

XST ユーザー ガイド (Virtex-4、Virtex-5、 Spartan-3 および CPLD デバイス用)

UG627 (v 12.4) 2010 年 12 月 14 日

スクリプト ファイルを使用した場合	615
run コマンドを使用した場合	616
set コマンドを使用した場合	619
elaborate コマンドを使用した場合	620
スクリプト モードでの XST の実行 (VHDL).....	620
スクリプト モードでの XST の実行 (Verilog).....	622
スクリプト モードでの XST の実行 (混合言語).....	624
コマンド ライン モードを使用した VHDL デザインの合成	625
コマンド ライン モードを使用した Verilog デザインの合成.....	628
コマンド ライン モードを使用した混合言語デザインの合成	629

表記規則	使用箇所	例
縦棒	選択するリストの項目を分離します。	lowpwr = {on off}
縦の省略記号	繰り返し項目が省略されていることを示します。	IOB #1: Name = QOUT IOB #2: Name = CLKIN . . .
横の省略記号 . . .	繰り返し項目が省略されていることを示します。	allow block . . . <i>block_name loc1 loc2 ... locn;</i>

オンライン マニュアル

このマニュアルでは、次の規則が使用されています。

表記規則	使用箇所	例
青色の文字	マニュアル内の相互参照、その他の文書へのリンクを示します。	詳細については、「 その他のリソース 」を参照してください。 詳細については、第 1 章「 タイトル フォーマット 」を参照してください。 詳細は、『 Virtex-6 ハンドブック 』の図 25 を参照してください。

詳細は、次を参照してください。

- ・ ISE® Design Suite ヘルプ
- ・ [デザイン制約](#)
- ・ [コマンドラインモード](#)

クロックの立ち上がりエッジで動作する非同期セットおよびクロック イネーブル付き 4 ビット レジスタの VHDL コード例

```
--
-- 4-bit Register with Positive-Edge Clock,
-- Asynchronous Set and Clock Enable
--

library ieee;
use ieee.std_logic_1164.all;

entity registers_5 is
    port(C, CE, PRE : in std_logic;
          D          : in std_logic_vector (3 downto 0);
          Q          : out std_logic_vector (3 downto 0));
end registers_5;

architecture archi of registers_5 is
begin

    process (C, PRE)
    begin
        if (PRE='1') then
            Q <= "1111";
        elsif (C'event and C='1') then
            if (CE='1') then
                Q <= D;
            end if;
        end if;
    end process;

end archi;
```

クロックの立ち上がりエッジで動作する非同期セットおよびクロック イネーブル付き 4 ビット レジスタの Verilog コード例

```
//
// 4-bit Register with Positive-Edge Clock,
// Asynchronous Set and Clock Enable
//

module v_registers_5 (C, D, CE, PRE, Q);
    input  C, CE, PRE;
    input  [3:0] D;
    output [3:0] Q;
    reg     [3:0] Q;

    always @(posedge C or posedge PRE)
    begin
        if (PRE)
            Q <= 4'b1111;
        else
            if (CE)
                Q <= D;
        end
    end

endmodule
```



```
architecture archi of three_st_2 is
begin
    O <= I when (T='0') else 'Z';
end archi;
```

プロセス同時処理代入文を使用したトライステートの Verilog コード例

```
//
// Tristate Description Using Concurrent Assignment
//

module v_three_st_2 (T, I, O);
    input  T, I;
    output O;

    assign O = (~T) ? I: 1'bZ;

endmodule
```


非同期リセット付き 4 ビット符号なしアップ アキュムレータの VHDL コード例

```
--
-- 4-bit Unsigned Up Accumulator with Asynchronous Reset
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity accumulators_1 is
    port(C, CLR : in std_logic;
          D : in std_logic_vector(3 downto 0);
          Q : out std_logic_vector(3 downto 0));
end accumulators_1;

architecture archi of accumulators_1 is
    signal tmp: std_logic_vector(3 downto 0);
begin

    process (C, CLR)
    begin
        if (CLR='1') then
            tmp <= "0000";
        elsif (C'event and C='1') then
            tmp <= tmp + D;
        end if;
    end process;

    Q <= tmp;

end archi;
```

非同期リセット付き 4 ビット符号なしアップ アキュムレータの Verilog コード例

```
//  
// 4-bit Unsigned Up Accumulator with Asynchronous Reset  
//  
  
module v_accumulators_1 (C, CLR, D, Q);  
  
    input C, CLR;  
    input [3:0] D;  
    output [3:0] Q;  
    reg [3:0] tmp;  
  
    always @(posedge C or posedge CLR)  
    begin  
        if (CLR)  
            tmp = 4'b0000;  
        else  
            tmp = tmp + D;  
        end  
        assign Q = tmp;  
    endmodule
```



```
        tmp(i+1) <= tmp(i);
    end loop;
    tmp(0) <= SI;
end if;
end if;
end process;

SO <= tmp(7);

end archi;
```

クロックの立ち下がリエッジで動作するシリアル入力、シリアル出力のクロック イネーブル付き 8 ビット シフト レフト レジスタの Verilog コード例

```
//
// 8-bit Shift-Left Register with Negative-Edge Clock,
// Clock Enable, Serial In, and Serial Out
//

module v_shift_registers_2 (C, CE, SI, SO);
    input C, SI, CE;
    output SO;
    reg [7:0] tmp;

    always @(negedge C)
    begin
        if (CE)
        begin
            tmp = {tmp[6:0], SI};
        end
    end

    assign SO = tmp[7];

endmodule
```

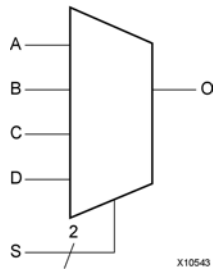

if 文を使用した 4:1 の 1 ビット MUX の Verilog コード例

```
//
// 4-to-1 1-bit MUX using an If statement.
//

module v_multiplexers_1 (a, b, c, d, s, o);
    input a,b,c,d;
    input [1:0] s;
    output o;
    reg o;

    always @(a or b or c or d or s)
    begin
        if (s == 2'b00) o = a;
        else if (s == 2'b01) o = b;
        else if (s == 2'b10) o = c;
        else o = d;
    end
endmodule
```

case 文を使用した 4:1 の 1 ビット MUX の図



case 文を使用した 4:1 の 1 ビット MUX のピンの説明

I/O ピン	説明
a、b、c、d	データ入力
s	MUX セレクタ
o	データ出力

ラッチ推論になる else 文が記述されていない Verilog コード例

```
//  
// 3-to-1 1-bit MUX with a 1-bit latch.  
//  
module v_multiplexers_4 (a, b, c, s, o);  
    input a,b,c;  
    input [1:0] s;  
    output o;  
    reg o;  
  
    always @(a or b or c or s)  
    begin  
        if (s == 2'b00) o = a;  
        else if (s == 2'b01) o = b;  
        else if (s == 2'b10) o = c;  
    end  
endmodule
```


デコーダが推論されない (デコーダ出力が未使用の) VHDL コード例

```
--  
-- No Decoder Inference (unused decoder output)  
--  
  
library ieee;  
use ieee.std_logic_1164.all;  
  
entity decoders_3 is  
    port (sel: in std_logic_vector (2 downto 0);  
          res: out std_logic_vector (7 downto 0));  
end decoders_3;  
  
architecture archi of decoders_3 is  
begin  
    res <= "00000001" when sel = "000" else  
        -- unused decoder output  
        "XXXXXXXX" when sel = "001" else  
        "00000100" when sel = "010" else  
        "00001000" when sel = "011" else  
        "00010000" when sel = "100" else  
        "00100000" when sel = "101" else  
        "01000000" when sel = "110" else  
        "10000000";  
end archi;
```

デコーダが推論されない (デコーダ出力が未使用の) Verilog コード例

```
//  
// No Decoder Inference (unused decoder output)  
//  
  
module v_decoders_3 (sel, res);  
    input [2:0] sel;  
    output [7:0] res;  
    reg [7:0] res;  
  
    always @(sel)  
    begin  
        case (sel)  
            3'b000 : res = 8'b00000001;  
            // unused decoder output  
            3'b001 : res = 8'bxxxxxxxx;  
            3'b010 : res = 8'b00000100;  
            3'b011 : res = 8'b00001000;  
            3'b100 : res = 8'b00010000;  
            3'b101 : res = 8'b00100000;  
            3'b110 : res = 8'b01000000;  
            default : res = 8'b10000000;  
        endcase  
    end  
endmodule
```

デコーダが推論されない (一部のセクタ値が未使用の) VHDL コード例

```
--  
-- No Decoder Inference (some selector values are unused)  
--  
  
library ieee;  
use ieee.std_logic_1164.all;  
  
entity decoders_4 is  
    port (sel: in std_logic_vector (2 downto 0);  
          res: out std_logic_vector (7 downto 0));  
end decoders_4;  
  
architecture archi of decoders_4 is  
begin  
    res <= "00000001" when sel = "000" else  
           "00000010" when sel = "001" else  
           "00000100" when sel = "010" else  
           "00001000" when sel = "011" else  
           "00010000" when sel = "100" else  
           "00100000" when sel = "101" else  
           -- 110 and 111 selector values are unused  
           "XXXXXXXX";  
end archi;
```

デコーダが推論されない (一部のセクタ値が未使用の) Verilog コード例

```
//  
// No Decoder Inference (some selector values are unused)  
//  
  
module v_decoders_4 (sel, res);  
    input [2:0] sel;  
    output [7:0] res;  
    reg [7:0] res;  
  
    always @(sel or res)  
    begin  
        case (sel)  
            3'b000 : res = 8'b00000001;  
            3'b001 : res = 8'b00000010;  
            3'b010 : res = 8'b00000100;  
            3'b011 : res = 8'b00001000;  
            3'b100 : res = 8'b00010000;  
            3'b101 : res = 8'b00100000;  
            // 110 and 111 selector values are unused  
            default : res = 8'bxxxxxxxx;  
        endcase  
    end  
endmodule
```


1:9 の 3 ビットのプライオリティ エンコーダの Verilog コード例

```
//  
// 3-Bit 1-of-9 Priority Encoder  
//  
  
(* priority_extract="force" *)  
module v_priority_encoder_1 (sel, code);  
    input  [7:0] sel;  
    output [2:0] code;  
    reg      [2:0] code;  
  
    always @(sel)  
    begin  
        if      (sel[0]) code = 3'b000;  
        else if (sel[1]) code = 3'b001;  
        else if (sel[2]) code = 3'b010;  
        else if (sel[3]) code = 3'b011;  
        else if (sel[4]) code = 3'b100;  
        else if (sel[5]) code = 3'b101;  
        else if (sel[6]) code = 3'b110;  
        else if (sel[7]) code = 3'b111;  
        else            code = 3'bxxx;  
    end  
  
endmodule
```


論理シフタ 1 の VHDL コード例

```
--
-- Following is the VHDL code for a logical shifter.
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity logical_shifters_1 is
    port(DI : in unsigned(7 downto 0);
         SEL : in unsigned(1 downto 0);
         SO : out unsigned(7 downto 0));
end logical_shifters_1;

architecture archi of logical_shifters_1 is
begin
    with SEL select
        SO <= DI when "00",
        DI sll 1 when "01",
        DI sll 2 when "10",
        DI sll 3 when others;
end archi;
```

論理シフタ 1 の Verilog コード例

```
//
// Following is the Verilog code for a logical shifter.
//

module v_logical_shifters_1 (DI, SEL, SO);
    input [7:0] DI;
    input [1:0] SEL;
    output [7:0] SO;
    reg [7:0] SO;

    always @(DI or SEL)
    begin
        case (SEL)
            2'b00 : SO = DI;
            2'b01 : SO = DI << 1;
            2'b10 : SO = DI << 2;
            default : SO = DI << 3;
        endcase
    end
endmodule
```



```

        case (SEL)
            2'b00 : SO = DI;
            2'b01 : SO = DI << 1;
            default : SO = DI << 2;
        endcase
    end
endmodule

```

論路シフタ 3 のピンの説明

I/O ピン	説明
DI	データ入力
SEL	シフト段数セレクタ
SO	データ出力

論理シフタ 3 の VHDL コード例

次の例では、セレクタの値が次の 2 進値で 1 増分していないので、論理シフタは推論されません。

```

--
-- XST does not infer a logical shifter for this example,
-- as the value is not incremented by 1 for each consequent
-- binary value of the selector.
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity logical_shifters_3 is
    port(DI : in unsigned(7 downto 0);
          SEL : in unsigned(1 downto 0);
          SO : out unsigned(7 downto 0));
end logical_shifters_3;

architecture archi of logical_shifters_3 is
begin
    with SEL select
        SO <= DI when "00",
        DI sll 1 when "01",
        DI sll 3 when "10",
        DI sll 2 when others;
end archi;

```

論理シフタ 3 の Verilog コード例

```
//  
// XST does not infer a logical shifter for this example,  
// as the value is not incremented by 1 for each consequent  
// binary value of the selector.  
//  
  
module v_logical_shifters_3 (DI, SEL, SO);  
    input [7:0] DI;  
    input [1:0] SEL;  
    output [7:0] SO;  
    reg[7:0] SO;  
  
    always @(DI or SEL)  
    begin  
        case (SEL)  
            2'b00 : SO = DI;  
            2'b01 : SO = DI << 1;  
            2'b10 : SO = DI << 3;  
            default : SO = DI << 2;  
        endcase  
    end  
endmodule
```


加算器、減算器、加減算器の HDL コーディング手法

このセクションには、次の項目が含まれます。

- ・ [加算器、減算器、加減算器の概要](#)
- ・ [加算器、減算器、加減算器のログ ファイル](#)
- ・ [加算器、減算器、加減算器の関連制約](#)
- ・ [加算器、減算器、加減算器のコード例](#)

加算器、減算器、加減算器の概要

このセクションには、次の項目が含まれます。

- ・ [サポートされるデバイス ファミリ](#)
- ・ [XST DSP48 ブロックのサポート](#)
- ・ [DSP48 ブロックのマクロ インプリメンテーション](#)
- ・ [自動 DSP48 リソース制御](#)
- ・ [最大マクロ コンフィギュレーション](#)

サポートされるデバイス ファミリ

次のデバイス ファミリでは、加算器と減算器を DSP48 リソースにインプリメントできます。

- ・ Virtex®-4
- ・ Virtex-5
- ・ Spartan®-3A DSP

XST DSP48 ブロックのサポート

XST では、出力レジスタの 1 つのレベルの DSP48 ブロックへのインプリメントがサポートされます。キャリーインまたは加減算のセレクトにレジスタが付いている場合も、これらのレジスタが DSP48 に挿入されます。

XST では、インプリメンテーションに必要な DSP リソースが 1 つのみの場合に、加減算器を DSP48 ブロックにインプリメントできます。1 つの DSP48 にフィットしない場合は、スライス ロジックを使用してマクロ全体がインプリメントされます。

DSP48 ブロックのマクロ インプリメンテーション

DSP48 へのマクロのインプリメンテーションは、[DSP 使用率 \(DSP_UTILIZATION_RATIO\)](#) 制約をデフォルト値の auto に設定して制御します。auto モードにすると、加減算器はフィルタのようなより複雑なマクロの一部になり、XST ではこれを自動的に DSP ブロックに配置します。これ以外のモードでは、LUT を使用して加減算器がインプリメントされます。

これらのマクロを強制的に DSP48 に挿入するには、[DSP48 の使用 \(USE_DSP48\)](#) の値を yes に設定する必要があります。DSP ブロックに加減算器を配置する際、XST ではそこからほかの DSP チェーンへの接続があるかどうかを確認されます。接続がある場合、高速の DSP 接続を使用してこの加減算器を DSP チェーンに接続します。

自動 DSP48 リソース制御

DSP48 ブロックに加減算器をインプリメントすると、XST では DSP48 リソースが自動的に制御されます。

最大マクロ コンフィギュレーション

XST では、DSP48 に最大数のレジスタを含めるなど、デフォルトで最大限のマクロ コンフィギュレーションを推論およびインプリメントすることで、最良のパフォーマンスを達成しようとします。マクロを特定のコンフィギュレーションにする場合は、**キープ (KEEP)** 制約を使用する必要があります。たとえば、DSP48 の 1 段目のレジスタを DSP48 に挿入しない場合は、**キープ (KEEP)** 制約をこれらのレジスタの出力に設定する必要があります。

加算器、減算器、加減算器のログ ファイル

ログ ファイルには、HDL 合成段階で推論された乗算器、加算器、減算器、およびレジスタの詳細がレポートされています。推論された MAC の情報は、MAC のインプリメンテーションの起こるアドバンス HDL 合成中にレポートされます。

加算器、減算器、加減算器のログ ファイルの例

```
Synthesizing Unit <v_adders_4>.
  Related source file is "v_adders_4.v".
  Found 8-bit adder carry in/out for signal <$addsub0000>.
  Summary:
    inferred    1 Adder/Subtractor(s).
Unit <v_adders_4> synthesized.

=====
HDL Synthesis Report

Macro Statistics
# Adders/Subtractors                : 1
  8-bit adder carry in/out          : 1
=====
```

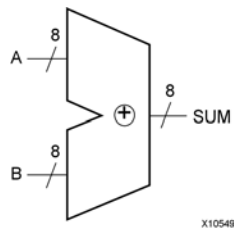
加算器、減算器、加減算器の関連制約

- ・ **DSP48 の使用 (USE_DSP48)**
- ・ **DSP 使用率 (DSP_UTILIZATION_RATIO)**
- ・ **キープ (KEEP)**

加算器、減算器、加減算器のコード例

コード例は、ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

符号なし 8 ビット加算器の図



符号なし 8 ビット加算器の IO ピンの説明

I/O ピン	説明
A、B	加算オペランド
SUM	加算結果

符号なし 8 ビット加算器の VHDL コード例

```
--
-- Unsigned 8-bit Adder
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity adders_1 is
  port(A,B : in std_logic_vector(7 downto 0);
        SUM : out std_logic_vector(7 downto 0));
end adders_1;

architecture archi of adders_1 is
begin

  SUM <= A + B;

end archi;
```

符号なし 8 ビット加算器の Verilog コード例

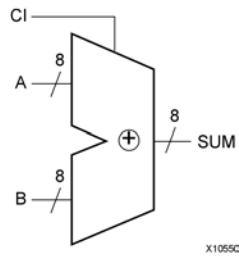
```
//
// Unsigned 8-bit Adder
//

module v_adders_1(A, B, SUM);
  input [7:0] A;
  input [7:0] B;
  output [7:0] SUM;

  assign SUM = A + B;

endmodule
```

キャリー イン付き符号なし 8 ビット加算器の図



キャリー イン付き符号なし 8 ビット加算器の IO ピンの説明

I/O ピン	説明
A、B	加算オペランド
CI	キャリー イン
SUM	加算結果

キャリー イン付き符号なし 8 ビット加算器の VHDL コード例

```
--
-- Unsigned 8-bit Adder with Carry In
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity adders_2 is
  port(A,B : in std_logic_vector(7 downto 0);
        CI : in std_logic;
        SUM : out std_logic_vector(7 downto 0));
end adders_2;

architecture archi of adders_2 is
begin

  SUM <= A + B + CI;

end archi;
```

キャリー イン付き符号なし 8 ビット加算器の Verilog コード例

```
//
// Unsigned 8-bit Adder with Carry In
//

module v_adders_2(A, B, CI, SUM);
    input [7:0] A;
    input [7:0] B;
    input CI;
    output [7:0] SUM;

    assign SUM = A + B + CI;

endmodule
```

キャリー アウト付き符号なし 8 ビット加算器

VHDL の場合は、キャリー アウトのある加算演算子 (+) を記述する前に、使用する演算パッケージを確認してください。たとえば、std_logic_unsigned では、次の形式で「+」を使用してキャリー アウトを得ることはできません。

```
Res (9-bit) = A (8-bit) + B (8-bit)
```

これは、このパッケージで「+」を使用すると、加算結果のビット数が最も大きい引数（この場合は 8 ビット）と等しくなってしまうからです。

これを解決するには、連結を使用してオペランド A および B を 9 ビットに調整する方法があります。

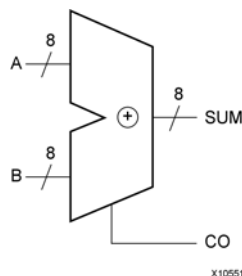
```
Res <= ("0" & A) + ("0" & B);
```

この場合、8 ビット加算器とキャリー アウトで 9 ビット加算器がインプリメントできると認識されます。

次は、別のソリューションです。

- ・ A と B を整数に変換します。
- ・ その結果を std_logic ベクタに変換し戻します。
- ・ ベクタのサイズを 9 に指定します。

キャリー アウト付き符号なし 8 ビット加算器の図



キャリー アウト付き符号なし 8 ビット加算器の IO ピンの説明

I/O ピン	説明
A, B	加算オペランド
SUM	加算結果
CO	キャリー アウト

キャリー アウト付き符号なし 8 ビット加算器の VHDL コード例

```
--
-- Unsigned 8-bit Adder with Carry Out
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity adders_3 is
  port(A,B : in std_logic_vector(7 downto 0);
        SUM : out std_logic_vector(7 downto 0);
        CO : out std_logic);
end adders_3;

architecture archi of adders_3 is
  signal tmp: std_logic_vector(8 downto 0);
begin

  tmp <= conv_std_logic_vector((conv_integer(A) + conv_integer(B)),9);
  SUM <= tmp(7 downto 0);
  CO <= tmp(8);

end archi;
```

上記の例では、次の 2 つの演算パッケージが使用されています。

- ・ **std_logic_arith**
このパッケージには、整数から std_logic への変換関数 (conv_std_logic_vector) が含まれています。
- ・ **std_logic_unsigned**
このパッケージには、符号なしの + (プラス) 演算が含まれています。

キャリー アウト付き符号なし 8 ビット加算器の Verilog コード例

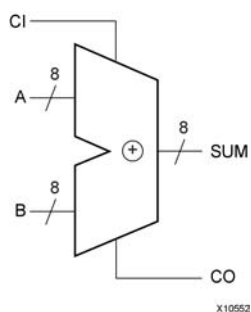
```
//
// Unsigned 8-bit Adder with Carry Out
//

module v_adders_3(A, B, SUM, CO);
    input [7:0] A;
    input [7:0] B;
    output [7:0] SUM;
    output CO;
    wire [8:0] tmp;

    assign tmp = A + B;
    assign SUM = tmp [7:0];
    assign CO = tmp [8];

endmodule
```

キャリー インおよびキャリー アウト付き符号なし 8 ビット加算器の図



キャリー イン およびキャリー アウト付き符号なし 8 ビット加算器の IO ピンの説明

I/O ピン	説明
A、B	加算オペランド
CI	キャリー イン
SUM	加算結果
CO	キャリー アウト

キャリー インおよびキャリー アウト付き符号なし 8 ビット加算器の VHDL コード例

```
--
-- Unsigned 8-bit Adder with Carry In and Carry Out
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity adders_4 is
```

```

port(A,B : in std_logic_vector(7 downto 0);
     CI : in std_logic;
     SUM : out std_logic_vector(7 downto 0);
     CO : out std_logic);
end adders_4;

architecture archi of adders_4 is
    signal tmp: std_logic_vector(8 downto 0);
begin

    tmp <= conv_std_logic_vector((conv_integer(A) + conv_integer(B) + conv_integer(CI)),9);
    SUM <= tmp(7 downto 0);
    CO <= tmp(8);

end archi;

```

キャリー インおよびキャリー アウト付き符号なし 8 ビット加算器の Verilog コード例

```

//
// Unsigned 8-bit Adder with Carry In and Carry Out
//

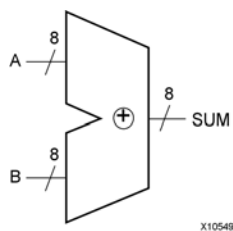
module v_adders_4(A, B, CI, SUM, CO);
    input CI;
    input [7:0] A;
    input [7:0] B;
    output [7:0] SUM;
    output CO;
    wire [8:0] tmp;

    assign tmp = A + B + CI;
    assign SUM = tmp [7:0];
    assign CO = tmp [8];

endmodule

```

符号付き 8 ビット加算器の図



符号付き 8 ビット加算器の IO ピンの説明

I/O ピン	説明
A、B	加算オペランド
SUM	加算結果

符号付き 8 ビット加算器の VHDL コード例

```
--
-- Signed 8-bit Adder
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

entity adders_5 is
  port(A,B : in std_logic_vector(7 downto 0);
        SUM : out std_logic_vector(7 downto 0));
end adders_5;

architecture archi of adders_5 is
begin

  SUM <= A + B;

end archi;
```

符号付き 8 ビット加算器の Verilog コード例

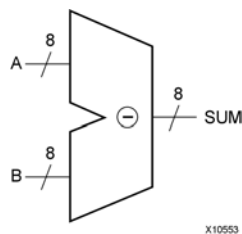
```
//
// Signed 8-bit Adder
//

module v_adders_5 (A,B,SUM);
  input signed [7:0] A;
  input signed [7:0] B;
  output signed [7:0] SUM;
  wire signed [7:0] SUM;

  assign SUM = A + B;

endmodule
```

符号なし 8 ビット減算器の図



符号なし 8 ビット減算器のピンの説明

I/O ピン	説明
A, B	減算オペランド
RES	減算結果

符号なし 8 ビット減算器の VHDL コード例

```
--
-- Unsigned 8-bit Subtractor
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity adders_6 is
  port(A,B : in std_logic_vector(7 downto 0);
       RES : out std_logic_vector(7 downto 0));
end adders_6;

architecture archi of adders_6 is
begin

  RES <= A - B;

end archi;
```

符号なし 8 ビット減算器の Verilog コード例

```
//
// Unsigned 8-bit Subtractor
//

module v_adders_6(A, B, RES);
  input [7:0] A;
  input [7:0] B;
  output [7:0] RES;

  assign RES = A - B;

endmodule
```

ボロー イン付き符号なし 8 ビット減算器のピンの説明

I/O ピン	説明
A, B	減算オペランド
BI	ボロー イン
RES	減算結果

ボロー イン付き符号なし 8 ビット減算器の VHDL コード例

```
--
-- Unsigned 8-bit Subtractor with Borrow In
--

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity adders_8 is
  port(A,B : in std_logic_vector(7 downto 0);
       BI : in std_logic;
       RES : out std_logic_vector(7 downto 0));
end adders_8;

architecture archi of adders_8 is

begin

    RES    <= A - B - BI;

end archi;
```

ボロー イン付き符号なし 8 ビット減算器の Verilog コード例

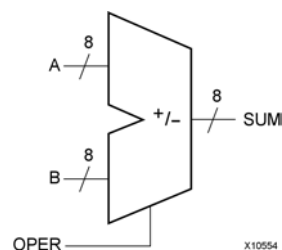
```
//
// Unsigned 8-bit Subtractor with Borrow In
//

module v_adders_8(A, B, BI, RES);
  input [7:0] A;
  input [7:0] B;
  input BI;
  output [7:0] RES;

  assign RES = A - B - BI;

endmodule
```

符号なし 8 ビット加減算器の図



符号なし 8 ビット加減算器のピンの説明

I/O ピン	説明
A, B	比較オペランド
OPER	加算/減算選択入力
SUM	加算/減算結果

符号なし 8 ビット加減算器の VHDL コード例

```
--
-- Unsigned 8-bit Adder/Subtractor
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity adders_7 is
port (A,B : in std_logic_vector(7 downto 0);
      OPER: in std_logic;
      RES : out std_logic_vector(7 downto 0));
end adders_7;

architecture archi of adders_7 is
begin

    RES <= A + B when OPER='0'
          else A - B;

end archi;
```

符号なし 8 ビット加減算器の Verilog コード例

```
//
// Unsigned 8-bit Adder/Subtractor
//

module v_adders_7(A, B, OPER, RES);
    input OPER;
    input [7:0] A;
    input [7:0] B;
    output [7:0] RES;
    reg [7:0] RES;

    always @(A or B or OPER)
    begin
        if (OPER==1'b0) RES = A + B;
        else RES = A - B;
    end
end
```

```
endmodule
```

コンパレータの HDL コーディング手法

このセクションには、次の内容が含まれます。

- ・ [コンパレータの概要](#)
- ・ [コンパレータのログ ファイル](#)
- ・ [コンパレータの関連制約](#)
- ・ [コンパレータのコード例](#)

コンパレータの概要

ありません。

コンパレータのログ ファイル

XST ログ ファイルには、認識されたコンパレータのタイプおよびビット幅が示されます。

コンパレータのログ ファイルの例

```
...
Synthesizing Unit <compar>.
  Related source file is comparators_1.vhd.
  Found 8-bit comparator greatequal for signal <$n0000> created at line 10.
  Summary:
    inferred      1 Comparator(s).
Unit <compar> synthesized.

=====
HDL Synthesis Report

Macro Statistics
# Comparators                : 1
  8-bit comparator greatequal : 1
=====
...
```

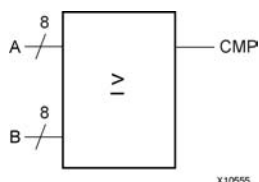
コンパレータの関連制約

なし

コンパレータのコード例

コード例は、ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

符号なし 8 ビット コンパレータの図



符号なし 8 ビット コンパレータのピンの説明

I/O ピン	説明
A, B	比較オペランド
CMP	比較結果

符号なし 8 ビット コンパレータの VHDL コード例

```
--
-- Unsigned 8-bit Greater or Equal Comparator
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity comparator_1 is
    port(A,B : in  std_logic_vector(7 downto 0);
         CMP : out std_logic);
end comparator_1;

architecture archi of comparator_1 is
begin

    CMP <= '1' when A >= B else '0';

end archi;
```

符号なし 8 ビット コンパレータの Verilog コード例

```
//
// Unsigned 8-bit Greater or Equal Comparator
//

module v_comparator_1 (A, B, CMP);
    input  [7:0] A;
    input  [7:0] B;
    output CMP;

    assign CMP = (A >= B) ? 1'b1 : 1'b0;
endmodule
```

乗算器の HDL コーディング手法

このセクションには、次の内容が含まれます。

- ・ [乗算器の概要](#)
- ・ [乗算器 \(Virtex®-4、Virtex-5、および Spartan®-3A DSP デバイス\)](#)
- ・ [乗算器のログ ファイル](#)
- ・ [乗算器の関連制約](#)
- ・ [乗算器のコード例](#)

乗算器の概要

このセクションでは、乗算器について次の内容に分けて説明します。

- ・ [乗算器のインプリメンテーション](#)
- ・ [レジスタ付き乗算器](#)
- ・ [定数との乗算](#)

乗算器のインプリメンテーション

乗算器をインプリメントする場合、結果の出力信号のビット数は 2 つのオペランドの合計ビット数と等しくなります。たとえば、A (8 ビット信号) と B (4 ビット信号) を掛け合わせる場合、結果は 12 ビット信号として宣言する必要があります。

レジスタ付き乗算器

次のデバイスでは、乗算器にレジスタを介した出力が必要な場合、特殊なレジスタ付き乗算器が推論されます。

- ・ Virtex®-4
- ・ Virtex-5

このレジスタ付き乗算器は 18 X 18 ビットです。

次の場合はレジスタ付き乗算器は使用されず、乗算器とレジスタが使用されます。

- ・ 乗算器の出力がレジスタ以外のコンポーネントに接続されている。
- ・ [乗算器スタイル \(MULT_STYLE\)](#) を次に設定します。

lut

- ・ 乗算器が非同期である。
- ・ 乗算器に同期リセットまたはクロック イネーブル以外の制御信号がある。
- ・ 乗算器が 1 つの 18 X 18 ブロック乗算器に収まらない。

レジスタ付き乗算器では、オプションで次のピンを使用できます。

- ・ クロック イネーブル ポート
- ・ 同期/非同期リセット ポートまたはロード ポート

定数との乗算

乗算に使用される引数の 1 つが定数の場合は、次の 2 つの方法を使用し、定数を使用した乗算という効率的な専用インプリメンテーションが作成されます。

- ・ KCM (Constant Coefficient Multiplier)
- ・ CSD (Canonical Signed Digit)

専用インプリメンテーションが、定数を使用した乗算器で必ずしも最良の結果になるとは限りません。XST では、KCM か通常の乗算インプリメンテーションのいずれかが自動的に選択されます。CSD は自動的に選択されません。CSD を使用する場合は、[MUX スタイル \(MUX_STYLE\)](#) 制約で指定します。

符号付きの数値を使用する場合は、KCM または CSD インプリメンテーションはサポートされません。

引数が 32 ビットを超える場合、[乗算器スタイル \(MULT_STYLE\)](#) 制約で指定されていても、KCM または CSD インプリメンテーションは使用されません。

乗算器 (Virtex-4、Virtex-5、および Spartan-3A DSP デバイス)

メモ: このセクションは、Virtex®-4、Virtex-5、Spartan®-3A DSP デバイスにのみ適用されます。

このセクションでは、次の内容について説明します。

- ・ [DSP48 リソースへの乗算器のインプリメンテーション](#)
- ・ [複数の DSP48 リソース](#)
- ・ [DSP48 ブロックのマクロ インプリメンテーション](#)
- ・ [乗算器スタイル制約の認識](#)
- ・ [最大マクロ コンフィギュレーション](#)

DSP48 リソースへの乗算器のインプリメンテーション

Virtex-4、Virtex-5、および Spartan-3A DSP デバイスでは、乗算器を DSP48 リソースにインプリメントできます。XST では、レジスタ付きのこれらのマクロをサポートし、DSP48 ブロックに最大 2 段の入力レジスタと 2 段の出力レジスタを挿入できます。

複数の DSP48 リソース

乗算器のインプリメンテーションで複数の DSP48 リソースが必要な場合は、XST で乗算器が自動的に分解されて複数の DSP ブロックに含められます。オペランドのサイズによっては、最良の結果が得られるように、乗算器のほとんどの部分が DSP48 ブロックを使用してインプリメントされ、残りがスライス ロジックを使用してインプリメントされる場合があります。たとえば、18 X18 符号なし乗算器を 1 つインプリメントするには、DSP48 1 つでは不十分です。この場合は、このロジックのほとんどが DSP48 ブロックにインプリメントされ、残りが LUT にインプリメントされます。

Virtex-4、Virtex-5、Spartan-3A DSP デバイスでは、LUT を使用したインプリメンテーションに加え、DSP48 を使用したインプリメンテーションでもパイプライン乗算器を推論できます。

DSP48 ブロックのマクロ インプリメンテーション

DSP48 ブロックへのマクロ インプリメンテーションは、[DSP48 の使用 \(USE_DSP48\)](#) 制約またはコマンドライン オプションでデフォルト auto に設定すると制御されます。このモードでは、アキュムレータがインプリメントされる際に、デバイス上で DSP48 リソースが考慮されます。

また、auto モードにすると、[DSP 使用率 \(DSP_UTILIZATION_RATIO\)](#)制約を使用して合成で DSP48 リソースが制御されます。XST では、デフォルトで DSP48 リソースをすべて使用しようとします。

詳細は、次を参照してください。

[DSP48 ブロック リソース](#)

乗算器スタイル制約の認識

XST では、値 lut および block が設定されている [乗算器スタイル \(MULT_STYLE\)](#) 制約が自動的に認識されて、これらが [DSP48 の使用 \(USE_DSP48\)](#) に変換されます。

ザイリンクスでは、次を推奨しています。

- ・ Virtex-4 および Virtex-5 デザインで乗算器のインプリメンテーションに使用する FPGA リソースを定義するときは、[DSP48 の使用 \(USE_DSP48\)](#) 制約を使用してください。
- ・ [乗算器スタイル \(MULT_STYLE\)](#) 制約は、選択した FPGA リソースで乗算器のインプリメンテーション方法を定義するときに使用してください。

次の規則が適用されます。

- ・ たとえば、[DSP48 の使用 \(USE_DSP48\)](#) が auto または yes に設定されていて、複数の DSP48 ブロックが必要な場合、mult_style=pipe_block を使用すると DSP48 のインプリメンテーションをパイプライン化できます。
- ・ [DSP48 \(USE_DSP48\) の使用](#) が no に設定されている場合、mult_style=pipe_lut|KCM|CSD を使用して、LUT に乗算器をインプリメンテーションする方法を定義できます。

最大マクロ コンフィギュレーション

XST では、DSP48 に最大数のレジスタを含めるなど、デフォルトで最大限のマクロ コンフィギュレーションを推論およびインプリメントすることで、最良のパフォーマンスを達成しようとします。マクロを特定のコンフィギュレーションにする場合は、[キープ \(KEEP\)](#) 制約を使用する必要があります。たとえば、DSP48 の 1 段目のレジスタを DSP48 に挿入しない場合は、[キープ \(KEEP\)](#) 制約をこれらのレジスタの出力に設定する必要があります。

乗算器のログ ファイル

XST ログ ファイルには、認識された乗算器のタイプおよびビット幅が示されます。

乗算器のログ ファイルの例

```

...
Synthesizing Unit <mux>.
    Related source file is multiplexers_1.vhd.
    Found 1-bit 4-to-1 multiplexer for signal <o>.
    Summary:
        inferred    1 Multiplexer(s).
    Unit <mux> synthesized.

=====
HDL Synthesis Report

Macro Statistics
# Multiplexers                : 1
  1-bit 4-to-1 multiplexer    : 1
=====
...

```

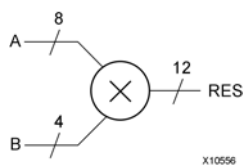
乗算器関連の制約

- ・ 乗算器スタイル (MULT_STYLE)
- ・ DSP48 の使用 (USE_DSP48)
- ・ DSP 使用率 (DSP_UTILIZATION_RATIO)
- ・ キープ (KEEP)

乗算器のコード例

コード例は、ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

符号なし 8 X 4 ビット乗算器の図



符号なし 8X4 ビット乗算器のピンの説明

I/O ピン	説明
A、B	乗算オペランド
RES	乗算結果

符号なし 8X4 ビット乗算器の VHDL コード例

```
--  
-- Unsigned 8x4-bit Multiplier  
--  
  
library ieee;  
use ieee.std_logic_1164.all;  
use ieee.std_logic_unsigned.all;  
  
entity multipliers_1 is  
    port(A : in std_logic_vector(7 downto 0);  
          B : in std_logic_vector(3 downto 0);  
          RES : out std_logic_vector(11 downto 0));  
end multipliers_1;  
  
architecture beh of multipliers_1 is  
begin  
    RES <= A * B;  
end beh;
```

符号なし 8X4 ビット乗算器の Verilog コード例

```
//  
// Unsigned 8x4-bit Multiplier  
//  
  
module v_multipliers_1(A, B, RES);  
    input [7:0] A;  
    input [3:0] B;  
    output [11:0] RES;  
  
    assign RES = A * B;  
endmodule
```

逐次型複素乗算器の HDL コーディング手法

このセクションには、次の内容が含まれます。

- ・ [逐次型複素乗算器の概要](#)
- ・ [逐次型複素乗算器のログ ファイル](#)
- ・ [逐次型複素乗算器関連の制約](#)
- ・ [逐次型複素乗算器のコード例](#)

逐次型複素乗算器の概要

逐次型複素乗算器には、次が必要です。

- ・ 中間結果を累積して完全な乗算を行うのに 4 サイクルが必要です。
- ・ インプリメンテーションには、DSP ブロックが 1 つが必要です。

2 つの複素数、A と B を乗算するには、次の 4 サイクルが必要です。

最初の 2 つのサイクルでは、次が計算されます。

```
Res_real = A_real * B_real - A_imag * B_imag
```

次の 2 つのサイクルでは、次が計算されます。

```
Res_imag = A_real * B_imag + A_imag * B_real
```

上記を実行するテンプレートはありますが、XST では DSP モードの違いを記述したり、enum 値を格納するのに enum 型または integer 型が使用できません。このため、基本的なテンプレートを使用して XST の推論を簡単にしてください。この一般的なアキュムレータのテンプレートを使用すると、XST では次を実行するために 1 つの DSP が推論されます。

- ・ **Load:** $P \leftarrow \text{Value}$
- ・ **Load:** $P \leftarrow -\text{Value}$
- ・ **Accumulate:** $P \leftarrow P + \text{Value}$
- ・ **Accumulate:** $P \leftarrow P - \text{Value}$

このテンプレートは、上記の 4 つの演算を実行する次の 2 つの制御信号で動作します。

- ・ **load**
- ・ **addsub**

逐次型複素乗算器のログ ファイル

なし

逐次型複素乗算器関連の制約

なし

逐次型複素乗算器のコード例

コード例は、ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

符号付き 18X18 ビットの逐次型複素乗算器のピンの説明

I/O ピン	説明
CLK	Clock Signal
Oper_Load、Oper_AddSub	load および addsub をコントロールする制御信号
A、B	乗算オペランド
RES	乗算結果

符号付き 18X18 ビットの逐次型複素乗算器の VHDL コード例

```
--
-- Sequential Complex Multiplier
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity multipliers_8 is
    generic(A_WIDTH:      positive:=18;
           B_WIDTH:      positive:=18;
           RES_WIDTH:     positive:=48);
    port ( CLK:           in  std_logic;
           A:            in  signed(A_WIDTH-1 downto 0);
           B:            in  signed(B_WIDTH-1 downto 0);

           Oper_Load:    in  std_logic;
           Oper_AddSub:  in  std_logic;
           -- Oper_Load Oper_AddSub Operation
           -- 0          0          R= +A*B
           -- 0          1          R= -A*B
           -- 1          0          R=R+A*B
           -- 1          1          R=R-A*B

           RES:          out signed(RES_WIDTH-1 downto 0)
    );
end multipliers_8;

architecture beh of multipliers_8 is

    constant P_WIDTH: integer:=A_WIDTH+B_WIDTH;

    signal oper_load0: std_logic:='0';
    signal oper_addsub0: std_logic:='0';

    signal pl: signed(P_WIDTH-1 downto 0):=(others=>'0');
    signal oper_load1: std_logic:='0';
    signal oper_addsub1: std_logic:='0';
```

```

        signal res0: signed(RES_WIDTH-1 downto 0);
begin

    process (clk)
        variable acc: signed(RES_WIDTH-1 downto 0);
    begin
        if rising_edge(clk) then
            oper_load0    <= Oper_Load;
            oper_addsub0 <= Oper_AddSub;

            p1 <= A*B;
            oper_load1    <= oper_load0;
            oper_addsub1 <= oper_addsub0;

            if (oper_load1='1') then
                acc := res0;
            else
                acc := (others=>'0');
            end if;

            if (oper_addsub1='1') then
                res0 <= acc-p1;
            else
                res0 <= acc+p1;
            end if;

        end if;
    end process;

    RES <= res0;

end architecture;

```

符号付き 18X18 ビットの逐次型複素乗算器の Verilog コード例

```

module v_multipliers_8(CLK,A,B,Oper_Load,Oper_AddSub, RES);
    parameter A_WIDTH    = 18;
    parameter B_WIDTH    = 18;
    parameter RES_WIDTH  = 48;
    parameter P_WIDTH    = A_WIDTH+B_WIDTH;

    input  CLK;
    input  signed [A_WIDTH-1:0] A, B;

    input  Oper_Load, Oper_AddSub;
    // Oper_Load  Oper_AddSub  Operation
    //  0          0           R= +A*B
    //  0          1           R= -A*B
    //  1          0           R=R+A*B
    //  1          1           R=R-A*B

```

```
output [RES_WIDTH-1:0] RES;

reg oper_load0    = 0;
reg oper_addsub0  = 0;

reg signed [P_WIDTH-1:0] p1 = 0;
reg oper_load1    = 0;
reg oper_addsub1  = 0;

reg signed [RES_WIDTH-1:0] res0 = 0;
reg signed [RES_WIDTH-1:0] acc;

always @(posedge CLK)
begin
    oper_load0    <= Oper_Load;
    oper_addsub0  <= Oper_AddSub;

    p1 <= A*B;
    oper_load1    <= oper_load0;
    oper_addsub1  <= oper_addsub0;

    if (oper_load1==1'b1)
        acc = res0;
    else
        acc = 0;

    if (oper_addsub1==1'b1)
        res0 <= acc-p1;
    else
        res0 <= acc+p1;

end

assign RES = res0;

endmodule
```

パイプライン乗算器の HDL コーディング手法

このセクションには、次の内容が含まれます。

- ・ [パイプライン乗算器の概要](#)
- ・ [パイプライン乗算器のログ ファイル](#)
- ・ [パイプライン乗算器関連の制約](#)
- ・ [パイプライン乗算器のコード例](#)

パイプライン乗算器の概要

このセクションでは、パイプライン乗算器の概要について説明します。

- ・ [推論済みパイプライン乗算器](#)
- ・ [パフォーマンスの最大化](#)
- ・ [未使用ステージのインプリメント](#)
- ・ [XST の制限](#)

推論済みパイプライン乗算器

XST では、大型乗算器を含むデザインでスピードを向上させるため、パイプライン乗算器を推論できます。大型乗算器の段の間にレジスタを挿入してパイプライン化することにより、デザイン全体の周波数を大幅に向上させることができます。パイプライン化の効果は、「[フリップフロップのリタイミング](#)」で説明されているフリップフロップのリタイミングと同様です。

パイプライン ステージを挿入するには

1. HDL コードで必要なレジスタを記述します。
2. それらのレジスタを乗算器の後に配置します。
3. [乗算器スタイル \(MULT_STYLE\)](#) を次に設定します。

`pipe_lut`

XST は次の場合もインプリメンテーションをパイプライン化できます。

- ・ ターゲットが Virtex®-4 または Virtex-5 デバイスの場合、および
- ・ 乗算器をインプリメントするには、複数の DSP48 ブロックが必要です。

このインスタンスの [乗算器スタイル \(MULT_STYLE\)](#) を次に設定します。

`pipe_block`

パフォーマンスの最大化

XST では最大の乗算器速度に到達させるため、次の両方の場合に使用可能なレジスタの最大数を使用します。

- ・ XST でパイプラインに有効なレジスタが検出される場合
- ・ [乗算器スタイル \(MULT_STYLE\)](#) を次に設定します。

`pipe_lut` または `pipe_block`

XST では、各乗算器で周波数を最大にするために使用するレジスタの最大数が自動的に計算されます。

アドバンス HDL 合成段階中、XST HDL Advisor からは次の場合に最適なレジスタ ステージ数を指定するようにメッセージが表示されます。

- ・ まだ十分な数のレジスタ ステージを指定していない場合
- ・ **乗算器スタイル (MULT_STYLE)** が信号に直接コード記述されている場合

未使用ステージのインプリメント

XST では、次の場合に未使用のステージがシフトレジスタとしてインプリメントされます。

- ・ 乗算器の後に配置されたレジスタの数が必要な最大数を超える場合
- ・ シフトレジスタの抽出がオンになっている場合

XST の制限

XST には、次の制限があります。

- ・ XST では、ハードウェア乗算器 (MULT18X18S リソースを使用したインプリメンテーション) はパイプライン化できません。
- ・ レジスタに非同期セット/リセットまたは同期リセット信号が含まれる場合は、乗算器はパイプライン化できません。同期リセット信号が含まれる場合は、パイプライン化できます。

パイプライン乗算器のログ ファイル

次に、出力されるログ ファイル例を示します。

パイプライン乗算器のログ ファイルの例

```
=====
*                               HDL Synthesis                               *
=====

Synthesizing Unit <multipliers_2>.
  Related source file is "multipliers_2.vhd".
  Found 36-bit register for signal <MULT>.
  Found 18-bit register for signal <a_in>.
  Found 18-bit register for signal <b_in>.
  Found 18x18-bit multiplier for signal <mult_res>.
  Found 36-bit register for signal <pipe_1>.
  Found 36-bit register for signal <pipe_2>.
  Found 36-bit register for signal <pipe_3>.
  Summary:
    inferred 180 D-type flip-flop(s).
    inferred   1 Multiplier(s).
Unit <multipliers_2> synthesized.
...
=====
*                               Advanced HDL Synthesis                       *
=====

Synthesizing (advanced) Unit <multipliers_2>.
Found pipelined multiplier on signal <mult_res>:
- 4 pipeline level(s) found in a register connected to the
```

```
multiplier macro output.
Pushing register(s) into the multiplier macro.
INFO:Xst - HDL ADVISOR - You can improve the performance of the
multiplier Mmult_mult_res by adding 1 register level(s).
Unit <multipliers_2> synthesized (advanced).
```

HDL Synthesis Report

Macro Statistics

```
# Multipliers                : 1
 18x18-bit registered multiplier : 1
```

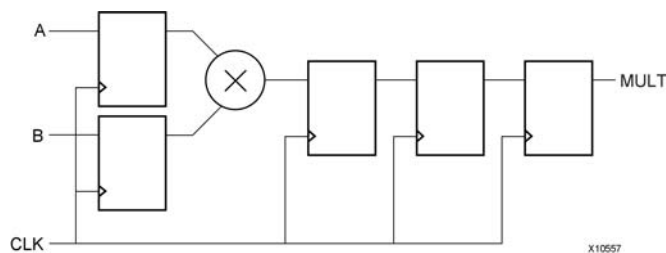
パイプライン乗算器関連の制約

- ・ DSP48 の使用 (USE_DSP48)
- ・ DSP 使用率 (DSP_UTILIZATION_RATIO)
- ・ キープ (KEEP)
- ・ 乗算器スタイル (MULT_STYLE)

パイプライン乗算器のコード例

コード例は、ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

パイプライン乗算器 (外部、単一レジスタ) の図



パイプライン乗算器 (外部、単一レジスタ) のピンの説明

I/O ピン	説明
clk	クロック (立ち上がりエッジ)
A、B	乗算オペランド
MULT	乗算結果

パイプライン乗算器 (外部、単一レジスタ) の VHDL コード例

```
--
-- Pipelined multiplier
--   The multiplication operation placed outside the
--   process block and the pipeline stages represented
--   as single registers.
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity multipliers_2 is
  generic(A_port_size : integer := 18;
         B_port_size : integer := 18);
  port(clk : in std_logic;
       A : in unsigned (A_port_size-1 downto 0);
       B : in unsigned (B_port_size-1 downto 0);
       MULT : out unsigned ( (A_port_size+B_port_size-1) downto 0));

  attribute mult_style: string;
  attribute mult_style of multipliers_2: entity is "pipe_lut";

end multipliers_2;

architecture beh of multipliers_2 is
  signal a_in, b_in : unsigned (A_port_size-1 downto 0);
  signal mult_res : unsigned ( (A_port_size+B_port_size-1) downto 0);
  signal pipe_1,
         pipe_2,
         pipe_3 : unsigned ((A_port_size+B_port_size-1) downto 0);

begin

  mult_res <= a_in * b_in;

  process (clk)
  begin
    if (clk'event and clk='1') then
      a_in <= A; b_in <= B;
      pipe_1 <= mult_res;
      pipe_2 <= pipe_1;
      pipe_3 <= pipe_2;
      MULT <= pipe_3;
    end if;
  end process;
end beh;
```

パイプライン乗算器 (外部、単一レジスタ) の Verilog コード例

```
//
// Pipelined multiplier
//   The multiplication operation placed outside the
//   always block and the pipeline stages represented
//   as single registers.
//

(*mult_style="pipe_lut"*)
module v_multipliers_2(clk, A, B, MULT);

    input clk;
    input [17:0] A;
    input [17:0] B;
    output [35:0] MULT;
    reg [35:0] MULT;
    reg [17:0] a_in, b_in;
    wire [35:0] mult_res;
    reg [35:0] pipe_1, pipe_2, pipe_3;

    assign mult_res = a_in * b_in;

    always @(posedge clk)
    begin
        a_in <= A; b_in <= B;
        pipe_1 <= mult_res;
        pipe_2 <= pipe_1;
        pipe_3 <= pipe_2;
        MULT <= pipe_3;
    end
endmodule
```

パイプライン乗算器 (内部、単一レジスタ) のピンの説明

I/O ピン	説明
clk	クロック (立ち上がりエッジ)
A, B	乗算オペランド
MULT	乗算結果

パイプライン乗算器 (内部、単一レジスタ) の VHDL コード例

```
--
-- Pipelined multiplier
--   The multiplication operation placed inside the
--   process block and the pipeline stages represented
--   as single registers.
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity multipliers_3 is
  generic(A_port_size: integer := 18;
         B_port_size: integer := 18);
  port(clk : in std_logic;
       A : in unsigned (A_port_size-1 downto 0);
       B : in unsigned (B_port_size-1 downto 0);
       MULT : out unsigned ((A_port_size+B_port_size-1) downto 0));

  attribute mult_style: string;
  attribute mult_style of multipliers_3: entity is "pipe_lut";

end multipliers_3;

architecture beh of multipliers_3 is
  signal a_in, b_in : unsigned (A_port_size-1 downto 0);
  signal mult_res : unsigned ((A_port_size+B_port_size-1) downto 0);
  signal pipe_2,
        pipe_3 : unsigned ((A_port_size+B_port_size-1) downto 0);

begin
  process (clk)
  begin
    if (clk'event and clk='1') then
      a_in <= A; b_in <= B;
      mult_res <= a_in * b_in;
      pipe_2 <= mult_res;
      pipe_3 <= pipe_2;
      MULT <= pipe_3;
    end if;
  end process;
end beh;
```

パイプライン乗算器 (内部、単一レジスタ) の Verilog コード例

```
//
// Pipelined multiplier
//   The multiplication operation placed inside the
//   process block and the pipeline stages are represented
//   as single registers.
//

(*mult_style="pipe_lut"*)
module v_multipliers_3(clk, A, B, MULT);

    input clk;
    input [17:0] A;
    input [17:0] B;
    output [35:0] MULT;
    reg [35:0] MULT;
    reg [17:0] a_in, b_in;
    reg [35:0] mult_res;
    reg [35:0] pipe_2, pipe_3;

    always @(posedge clk)
    begin
        a_in <= A; b_in <= B;
        mult_res <= a_in * b_in;
        pipe_2 <= mult_res;
        pipe_3 <= pipe_2;
        MULT <= pipe_3;
    end
endmodule
```

パイプライン乗算器 (外部、シフトレジスタ) のピンの説明

I/O ピン	説明
clk	クロック (立ち上がりエッジ)
A, B	乗算オペランド
MULT	乗算結果

パイプライン乗算器 (外部、シフトレジスタ) の VHDL コード例

```
--
-- Pipelined multiplier
--   The multiplication operation placed outside the
--   process block and the pipeline stages represented
--   as shift registers.
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity multipliers_4 is
    generic(A_port_size: integer := 18;
           B_port_size: integer := 18);
```

```
port(clk : in std_logic;
      A : in unsigned (A_port_size-1 downto 0);
      B : in unsigned (B_port_size-1 downto 0);
      MULT : out unsigned ( (A_port_size+B_port_size-1) downto 0));

attribute mult_style: string;
attribute mult_style of multipliers_4: entity is "pipe_lut";

end multipliers_4;

architecture beh of multipliers_4 is
  signal a_in, b_in : unsigned (A_port_size-1 downto 0);
  signal mult_res : unsigned ((A_port_size+B_port_size-1) downto 0);

  type pipe_reg_type is array (2 downto 0) of unsigned ((A_port_size+B_port_size-1) downto 0);
  signal pipe_regs : pipe_reg_type;

begin

  mult_res <= a_in * b_in;

  process (clk)
  begin
    if (clk'event and clk='1') then
      a_in <= A; b_in <= B;
      pipe_regs <= mult_res & pipe_regs(2 downto 1);
      MULT <= pipe_regs(0);
    end if;
  end process;
end beh;
```

パイプライン乗算器 (外部、シフトレジスタ) の Verilog コード例

```
//  
// Pipelined multiplier  
//   The multiplication operation placed outside the  
//   always block and the pipeline stages represented  
//   as shift registers.  
//  
  
(*mult_style="pipe_lut"*)  
module v_multipliers_4 (clk, A, B, MULT);  
  
    input clk;  
    input [17:0] A;  
    input [17:0] B;  
    output [35:0] MULT;  
    reg [35:0] MULT;  
    reg [17:0] a_in, b_in;  
    wire [35:0] mult_res;  
    reg [35:0] pipe_regs [2:0];  
    integer i;  
  
    assign mult_res = a_in * b_in;  
  
    always @(posedge clk)  
    begin  
        a_in <= A; b_in <= B;  
  
        pipe_regs[2] <= mult_res;  
        for (i=0; i<=1; i=i+1) pipe_regs[i] <= pipe_regs[i+1];  
  
        MULT <= pipe_regs[0];  
    end  
  
endmodule
```

乗算/加減算器の HDL コーディング手法

このセクションには、次の内容が含まれます。

- ・ [乗算/加減算器の概要](#)
- ・ [Virtex-4 および Virtex-5 デバイスの乗算/加減算器](#)
- ・ [乗算/加減算器のログ ファイル](#)
- ・ [乗算/加減算器関連の制約](#)
- ・ [乗算/加減算器のコード例](#)

乗算/加減算器の概要

乗算/加減算器は、次のような複数の基本マクロから構成されている複雑なマクロです。

- ・ 乗算器
- ・ 加算器/減算器
- ・ レジスタ

このマクロは、次のデバイスの DSP48 リソースにインプリメントできます。

- ・ Virtex®-4
- ・ Virtex-5

Virtex-4 および Virtex-5 デバイスの乗算/加減算器のコーディング手法

このセクションには、次の内容が含まれます。

- ・ [XST のレジスタ付きマクロのサポート](#)
- ・ [XST DSP48 ブロックのサポート](#)
- ・ [DSP48 ブロックのマクロ インプリメンテーション](#)
- ・ [最大マクロ コンフィギュレーション](#)

XST のレジスタ付きマクロのサポート

XST では、レジスタ付きのこのマクロがサポートされており、次に挿入できます。

- ・ 乗算器の入力に最大 2 レベルの入力レジスタ
- ・ 加算/減算器の入力に最大 1 レジスタレベル
- ・ DSP48 ブロックへ 1 レベルの出力レジスタ

キャリーインまたは加減算のセレクトにレジスタが付いている場合も、これらのレジスタが DSP48 に挿入されます。また、乗算処理にもレジスタを付けることができます。

XST DSP48 ブロックのサポート

XST では、インプリメンテーションに必要な DSP リソースが 1 つのみの場合に乗算/加減算器を DSP48 ブロックにインプリメントできます。1 つの DSP48 にフィットしない場合は、乗算器と加減算器が別々のマクロとして処理されます。

詳細は、次を参照してください。

- ・ [乗算器の HDL コーディング手法](#)
- ・ [加算器、減算器、加減算器の HDL コーディング手法](#)

DSP48 ブロックのマクロ インプリメンテーション

DSP48 ブロックのマクロ インプリメンテーションは、[DSP48 の使用 \(USE_DSP48\)](#) 制約をデフォルト値の auto に設定して制御します。このモードでは、乗算/加減算器がインプリメントされる際に、デバイス上で DSP48 リソースが考慮されます。

また、auto モードにすると、[DSP 使用率 \(DSP_UTILIZATION_RATIO\)](#)制約を使用して合成で DSP48 リソースが制御されます。XST では、デフォルトで DSP48 リソースをすべて使用しようとしています。

詳細は、次を参照してください。

[DSP48 ブロック リソース](#)

最大マクロ コンフィギュレーション

XST では、DSP48 に最大数のレジスタを含めるなど、デフォルトで最大限のマクロ コンフィギュレーションを推論およびインプリメントすることで、最良のパフォーマンスを達成しようとしています。マクロを特定のコンフィギュレーションにする場合は、[キープ \(KEEP\)](#) 制約を使用する必要があります。たとえば、DSP48 の 1 段目のレジスタを DSP48 に挿入しない場合は、[キープ \(KEEP\)](#) 制約をこれらのレジスタの出力に設定する必要があります。

乗算/加減算器のログ ファイル

ログ ファイルには、HDL 合成段階で推論された乗算器、加減算器、およびレジスタの詳細がレポートされています。乗算/加減算器は、アドバンス HDL 合成段階で作成されます。これらは MAC のインプリメンテーションに含まれるので、ログ ファイルには推論された MAC についての情報が表示されます。

乗算/加減算器のログ ファイルの例

```
=====
*                               HDL Synthesis                               *
=====

Synthesizing Unit <multipliers_6>.
  Related source file is "multipliers_6.vhd".
  Found 8-bit register for signal <A_reg1>.
  Found 8-bit register for signal <A_reg2>.
  Found 8-bit register for signal <B_reg1>.
  Found 8-bit register for signal <B_reg2>.
  Found 8x8-bit multiplier for signal <mult>.
  Found 16-bit addsub for signal <multaddsub>.
  Summary:
    inferred 32 D-type flip-flop(s).
    inferred 1 Adder/Subtractor(s).
    inferred 1 Multiplier(s).
Unit <multipliers_6> synthesized.
...

=====
*                               Advanced HDL Synthesis                       *
=====
...
```

Synthesizing (advanced) Unit <Mmult_mult>.

Multiplier <Mmult_mult> in block <multipliers_6> and adder/subtractor <Maddsub_multaddsub> in block <multipliers_6> are combined into a MAC<Mmac_Maddsub_multaddsub>.

The following registers are also absorbed by the MAC: <A_reg2> in block <multipliers_6>, <A_reg1> in block <multipliers_6>, <B_reg2> in block <multipliers_6>, <B_reg1> in block <multipliers_6>.

Unit <Mmult_mult> synthesized (advanced).

HDL Synthesis Report

Macro Statistics

```
# MACs                      : 1
8x8-to-16-bit MAC          : 1
```

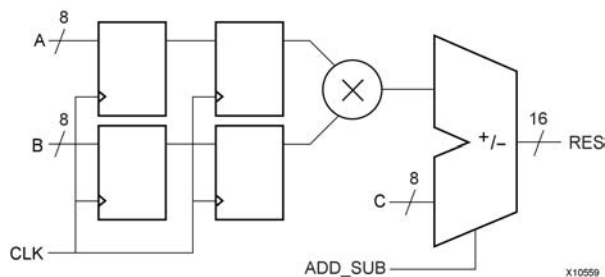
乗算/加減算器関連の制約

- ・ DSP48 の使用 (USE_DSP48)
- ・ DSP 使用率 (DSP_UTILIZATION_RATIO)
- ・ キープ (KEEP)

乗算/加減算器のコード例

コード例は、ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

乗算器の入力に 2 段のレジスタが付いた乗算/加算器の図



乗算器の入力に 2 段のレジスタが付いた乗算/加算器のピンの説明

I/O ピン	説明
clk	クロック (立ち上がりエッジ)
A、B、C	乗算/加算オペランド
RES	乗算/加算結果

乗算器の入力に 2 段のレジスタが付いた乗算/加算器の VHDL コード例

```
--
-- Multiplier Adder with 2 Register Levels on Multiplier Inputs
--

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity multipliers_5 is
    generic (p_width: integer:=8);
    port (clk : in std_logic;
          A, B, C : in std_logic_vector(p_width-1 downto 0);
          RES : out std_logic_vector(p_width*2-1 downto 0));
end multipliers_5;

architecture beh of multipliers_5 is
    signal A_reg1, A_reg2,
           B_reg1, B_reg2 : std_logic_vector(p_width-1 downto 0);
    signal multaddsub : std_logic_vector(p_width*2-1 downto 0);
begin

    multaddsub <= A_reg2 * B_reg2 + C;

    process (clk)
    begin
        if (clk'event and clk='1') then
            A_reg1 <= A; A_reg2 <= A_reg1;
            B_reg1 <= B; B_reg2 <= B_reg1;
        end if;
    end process;

    RES <= multaddsub;

end beh;
```

乗算器の入力に 2 段のレジスタが付いた乗算/加算器の Verilog コード例

```
//
// Multiplier Adder with 2 Register Levels on Multiplier Inputs
//

module v_multipliers_5 (clk, A, B, C, RES);

    input  clk;
    input  [7:0] A;
    input  [7:0] B;
    input  [7:0] C;
    output [15:0] RES;
    reg     [7:0] A_reg1, A_reg2, B_reg1, B_reg2;
```

```

wire    [15:0] multaddsub;

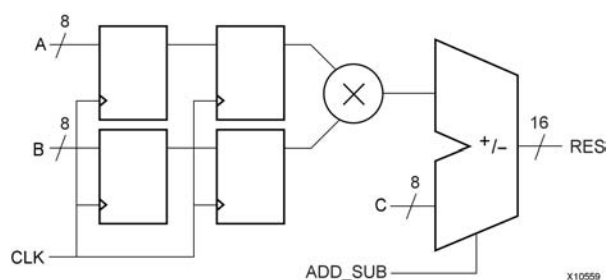
always @(posedge clk)
begin
    A_reg1 <= A; A_reg2 <= A_reg1;
    B_reg1 <= B; B_reg2 <= B_reg1;
end

assign multaddsub = A_reg2 * B_reg2 + C;
assign RES = multaddsub;

endmodule

```

乗算器の入力に 2 段のレジスタが付いた乗算/加減算器の図



乗算器の入力に 2 段のレジスタが付いた乗算/加減算器のピンの説明

I/O ピン	説明
clk	クロック (立ち上がりエッジ)
ADD_SUB	加減算セレクタ
A、B、C	乗算/加減算オペランド
RES	乗算/加減算結果

乗算器の入力に 2 段のレジスタが付いた乗算/加減算器の VHDL コード例

```

--
-- Multiplier Adder/Subtractor with
-- 2 Register Levels on Multiplier Inputs
--

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity multipliers_6 is
    generic (p_width: integer:=8);
    port (clk,add_sub: in std_logic;
          A, B, C: in std_logic_vector(p_width-1 downto 0);
          RES: out std_logic_vector(p_width*2-1 downto 0));
end entity multipliers_6

```

```

end multipliers_6;

architecture beh of multipliers_6 is
    signal A_reg1, A_reg2,
           B_reg1, B_reg2 : std_logic_vector(p_width-1 downto 0);
    signal mult, multaddsub : std_logic_vector(p_width*2-1 downto 0);
begin

    mult <= A_reg2 * B_reg2;
    multaddsub <= C + mult when add_sub = '1' else C - mult;

    process (clk)
    begin
        if (clk'event and clk='1') then
            A_reg1 <= A; A_reg2 <= A_reg1;
            B_reg1 <= B; B_reg2 <= B_reg1;
        end if;
    end process;

    RES <= multaddsub;

end beh;

```

乗算器の入力に 2 段のレジスタが付いた乗算/加減算器の Verilog コード例

```

//
// Multiplier Adder/Subtractor with
// 2 Register Levels on Multiplier Inputs
//

module v_multipliers_6 (clk, add_sub, A, B, C, RES);

    input  clk, add_sub;
    input  [7:0] A;
    input  [7:0] B;
    input  [7:0] C;
    output [15:0] RES;
    reg    [7:0] A_reg1, A_reg2, B_reg1, B_reg2;
    wire   [15:0] mult, multaddsub;

    always @(posedge clk)
    begin
        A_reg1 <= A; A_reg2 <= A_reg1;
        B_reg1 <= B; B_reg2 <= B_reg1;
    end

    assign mult = A_reg2 * B_reg2;
    assign multaddsub = add_sub ? C + mult : C - mult;
    assign RES = multaddsub;

endmodule

```

MAC の HDL コーディング手法

このセクションには、次の内容が含まれます。

- ・ [MAC の概要](#)
- ・ [Virtex-4 および Virtex-5 デバイスの MAC](#)
- ・ [MAC のログ ファイル](#)
- ・ [MAC 関連の制約](#)
- ・ [MAC のコード例](#)

MAC の概要

MAC (積和演算) は、次のような複数の基本マクロから構成されている複雑なマクロです。

- ・ 乗算器
- ・ アキュムレータ
- ・ レジスタ

このマクロは、次のデバイスの DSP48 リソースにインプリメントできます。

- ・ Virtex®-4
- ・ Virtex-5

Virtex-4 および Virtex-5 デバイスの MAC

このセクションには、次の内容が含まれます。

- ・ [XST のレジスタ付きマクロのサポート](#)
- ・ [XST DSP48 ブロックのサポート](#)
- ・ [DSP48 ブロックのマクロ インプリメンテーション](#)
- ・ [最大マクロ コンフィギュレーション](#)

詳細は、次を参照してください。

- ・ [乗算器の HDL コーディング手法](#)
- ・ [アキュムレータの HDL コーディング手法](#)

XST のレジスタ付きマクロのサポート

XST では、レジスタ付きのこのマクロがサポートされており、最大 2 段の入力レジスタを DSP48 ブロックに挿入できます。加減算のセレクトにレジスタが付いている場合も、これらのレジスタが DSP48 に挿入されます。また、乗算処理にもレジスタを付けることができます。

XST DSP48 ブロックのサポート

XST では、インプリメンテーションに必要な DSP リソースが 1 つのみの場合に乗累算 (MAC) を DSP48 ブロックにインプリメントできます。1 つの DSP48 にフィットしない場合は、乗算器とアキュムレータ (累算) が別々のマクロとして処理されます。

DSP48 ブロックのマクロ インプリメンテーション

DSP48 ブロックへのマクロ インプリメンテーションは、[DSP48 の使用 \(USE_DSP48\)](#) 制約またはコマンドライン オプションでデフォルト auto に設定すると制御されます。このモードでは、MAC がインプリメントされる際に、デバイス上で DSP48 リソースが考慮されます。

また、auto モードにすると、[DSP 使用率 \(DSP_UTILIZATION_RATIO\)](#) 制約を使用して DSP48 リソースが制御されます。XST は、できるだけ多くの DSP48 リソースを使用しようとします。詳細は、「[DSP48 ブロック リソース](#)」を参照してください。

最大マクロ コンフィギュレーション

XST では、DSP48 に最大数のレジスタを含めるなど、デフォルトで最大限のマクロ コンフィギュレーションを推論およびインプリメントすることで、最良のパフォーマンスを達成しようとします。マクロを特定の方法でインプリメントするには、[キープ \(KEEP\)](#) 制約を使用する必要があります。たとえば、DSP48 の 1 段目のレジスタを DSP48 に挿入しない場合は、[キープ \(KEEP\)](#) 制約をこれらのレジスタの出力に設定する必要があります。

MAC のログ ファイル

XST では、MAC のログ ファイルに次の情報がレポートされます。

手順	レポート
HDL 合成	推論済み乗算器、アキュムレータ、レジスタの詳細
アドバンス HDL 合成	MAC マクロの構成

MAC のログ ファイルの例

```
=====
*                               HDL Synthesis                               *
=====
...
Synthesizing Unit <multipliers_7a>.
  Related source file is "multipliers_7a.vhd".
  Found 8x8-bit multiplier for signal <$n0002> created at line 28.
  Found 16-bit up accumulator for signal <accum>.
  Found 16-bit register for signal <mult>.
  Summary:
    inferred   1 Accumulator(s).
    inferred  16 D-type flip-flop(s).
    inferred   1 Multiplier(s).
Unit <multipliers_7a> synthesized....
=====
*                               Advanced HDL Synthesis                       *
=====
...
Synthesizing (advanced) Unit <Mmult__n0002>.
  Multiplier <Mmult__n0002> in block <multipliers_7a> and accumulator
  <accum> in block <multipliers_7a> are combined into a MAC<Mmac_accum>.
  The following registers are also absorbed by the MAC: <mult> in block
```

```
<multipliers_7a>. Unit <Mmult__n0002> synthesized (advanced).
```

```
=====
HDL Synthesis Report
```

```
Macro Statistics
```

```
# MACs                      : 1
8x8-to-16-bit MAC          : 1
```

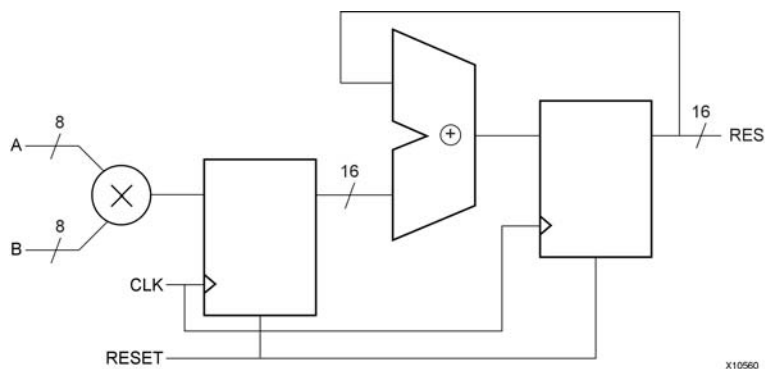
MAC 関連の制約

- ・ DSP48 の使用 (USE_DSP48)
- ・ DSP 使用率 (DSP_UTILIZATION_RATIO)
- ・ キープ (KEEP)

MAC のコード例

コード例は、ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

乗算アップ アキュムレータ (乗算後にレジスタ付き) の図



乗算アップ アキュムレータ (乗算後にレジスタ付き) のピンの説明

I/O ピン	説明
clk	クロック (立ち上がりエッジ)
reset	同期リセット
A、B	MAC オペランド
RES	MAC 結果

乗算アップ アキュムレータ (乗算後にレジスタ付き) の VHDL コード例

```
--
-- Multiplier Up Accumulate with Register After Multiplication
--
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity multipliers_7a is
  generic (p_width: integer:=8);
  port (clk, reset: in std_logic;
        A, B: in std_logic_vector(p_width-1 downto 0);
        RES: out std_logic_vector(p_width*2-1 downto 0));
end multipliers_7a;

architecture beh of multipliers_7a is
  signal mult, accum: std_logic_vector(p_width*2-1 downto 0);
begin

  process (clk)
  begin
    if (clk'event and clk='1') then
      if (reset = '1') then
        accum <= (others => '0');
        mult <= (others => '0');
      else
        accum <= accum + mult;
        mult <= A * B;
      end if;
    end if;
  end process;

  RES <= accum;

end beh;
```

乗算アップ アキュムレータ (乗算後にレジスタ付き) の Verilog コード例

```
//
// Multiplier Up Accumulate with Register After Multiplication
//
module v_multipliers_7a (clk, reset, A, B, RES);
  input clk, reset;
  input [7:0] A;
  input [7:0] B;
  output [15:0] RES;
  reg [15:0] mult, accum;

  always @(posedge clk)
  begin
```

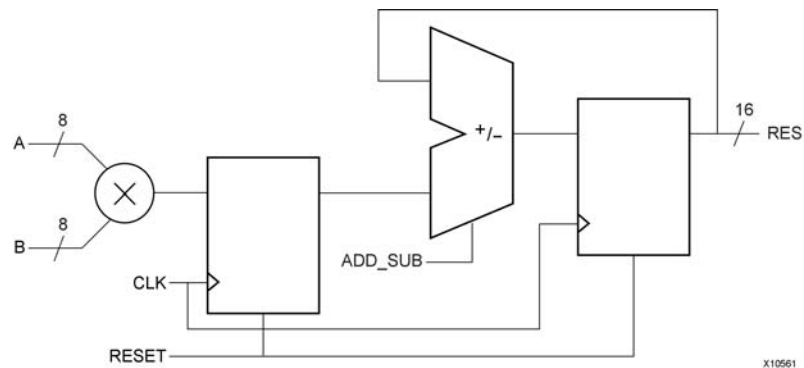
```

if (reset)
    mult <= 16'b0000000000000000;
else
    mult <= A * B;
end

always @(posedge clk)
begin
    if (reset)
        accum <= 16'b0000000000000000;
    else
        accum <= accum + mult;
    end
    assign RES = accum;
endmodule

```

乗算アップ/ダウン アキュムレータ (乗算後にレジスタ付き) の図



乗算アップ/ダウン アキュムレータ (乗算後にレジスタ付き) のピンの説明

I/O ピン	説明
clk	クロック (立ち上がりエッジ)
reset	同期リセット
ADD_SUB	加減算セクタ
A、B	MAC オペランド
RES	MAC 結果

乗算アップ/ダウン アキュムレータ (乗算後にレジスタ付き) の VHDL コード例

```
--
-- Multiplier Up/Down Accumulate with Register
-- After Multiplication
--

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity multipliers_7b is
    generic (p_width: integer:=8);
    port (clk, reset, add_sub: in std_logic;
          A, B: in std_logic_vector(p_width-1 downto 0);
          RES: out std_logic_vector(p_width*2-1 downto 0));
end multipliers_7b;

architecture beh of multipliers_7b is
    signal mult, accum: std_logic_vector(p_width*2-1 downto 0);
begin

    process (clk)
    begin
        if (clk'event and clk='1') then
            if (reset = '1') then
                accum <= (others => '0');
                mult <= (others => '0');
            else

                if (add_sub = '1') then
                    accum <= accum + mult;
                else
                    accum <= accum - mult;
                end if;

                mult <= A * B;
            end if;
        end if;
    end process;

    RES <= accum;

end beh;
```

乗算アップ/ダウン アキュムレータ (乗算後にレジスタ付き) の Verilog コード例

```
//  
// Multiplier Up/Down Accumulate with Register  
// After Multiplication  
//  
  
module v_multipliers_7b (clk, reset, add_sub, A, B, RES);  
  
    input  clk, reset, add_sub;  
    input  [7:0] A;  
    input  [7:0] B;  
    output [15:0] RES;  
    reg    [15:0] mult, accum;  
  
    always @(posedge clk)  
    begin  
        if (reset)  
            mult <= 16'b0000000000000000;  
        else  
            mult <= A * B;  
        end  
  
        always @(posedge clk)  
        begin  
            if (reset)  
                accum <= 16'b0000000000000000;  
            else  
                if (add_sub)  
                    accum <= accum + mult;  
                else  
                    accum <= accum - mult;  
                end  
            end  
  
            assign RES = accum;  
  
        endmodule
```

除算器の HDL コーディング手法

このセクションには、次の内容が含まれます。

- ・ [除算器の概要](#)
- ・ [除算器のログ ファイル](#)
- ・ [除算器の関連制約](#)
- ・ [除算器のコード例](#)

除算器の概要

除算器は、序数が定数で 2 のべき乗の場合にのみサポートされます。この場合、演算子がシフトとしてインプリメントされます。それ以外の場合、XST でエラー メッセージが表示されます。

除算器のログ ファイル

除数が定数で 2 のべき乗の除算器は、マクロ認識の段階ではレポートされません。除算器が XST でサポートされる文字に対応していない場合は、次のエラー メッセージが表示されます。

```
...
ERROR:Xst:719 - file1.vhd (Line 172).
Operator is not supported yet : 'DIVIDE'
...
```

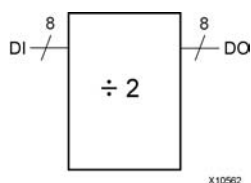
除算器関連の制約

なし

除算器のコード例

コード例は、ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

定数 2 で割る除算器の図



定数 2 で割る除算器のピンの説明

I/O ピン	説明
DI	除算オペランド
DO	除算結果

定数 2 で割る除算器の VHDL コード例

```
--  
-- Division By Constant 2  
--  
  
library ieee;  
use ieee.std_logic_1164.all;  
use ieee.numeric_std.all;  
  
entity divider_1 is  
    port(DI : in unsigned(7 downto 0);  
          DO : out unsigned(7 downto 0));  
end divider_1;  
  
architecture archi of divider_1 is  
begin  
  
    DO <= DI / 2;  
  
end archi;
```

定数 2 で割る除算器の Verilog コード例

```
//  
// Division By Constant 2  
//  
  
module v_divider_1 (DI, DO);  
    input  [7:0] DI;  
    output [7:0] DO;  
  
    assign DO = DI / 2;  
  
endmodule
```

リソース共有の HDL コーディング手法

このセクションには、次の内容が含まれます。

- ・ [リソース共有の概要](#)
- ・ [リソース共有のログ ファイル](#)
- ・ [リソース共有の関連制約](#)
- ・ [リソース共有のコード例](#)

リソース共有の概要

リソースの共有は、演算の数および合成されたデザインに含まれるロジックの数を最小限に抑えるために行われます。この最適化では、2 つの類似する演算リソースが同時に使用されない場合に、この 2 つを 1 つの演算子としてインプリメントします。XST では、リソースの共有と、必要に応じて、マルチプレクサの数を減らす処理が実行されます。

XST では、次のリソースの共有がサポートされます。

- ・ 加算器
- ・ 減算器
- ・ 加算器/減算器
- ・ 乗算器

最適化の目標がスピードである場合は、リソースの共有をオフにした方が良い結果が得られる場合があります。XST では、クロック周波数を向上するために、アドバンス HDL 合成段階でリソースの共有をオフにすることを推奨するメッセージが表示されます。

リソース共有のログ ファイル

XST ログ ファイルには、認識された数値演算ブロックおよび乗算器のタイプおよびビット幅が表示されます。

リソース共有のログ ファイルの例

```
...
Synthesizing Unit <addsub>.
  Related source file is resource_sharing_1.vhd.
  Found 8-bit addsub for signal <res>.
  Found 8 1-bit 2-to-1 multiplexers.
  Summary:
    inferred   1 Adder/Subtractor(s).
    inferred   8 Multiplexer(s).
  Unit <addsub> synthesized.

=====
HDL Synthesis Report

Macro Statistics
# Multiplexers                : 1
  2-to-1 multiplexer          : 1
# Adders/Subtractors          : 1
  8-bit addsub                : 1
=====
...
=====
*                               Advanced HDL Synthesis                               *
=====

INFO:Xst - HDL ADVISOR - Resource sharing has identified that some
arithmetic operations in this design can share the same physical resources
```

for reduced device utilization. For improved clock frequency you may try to disable resource sharing.
...

リソース共有関連の制約

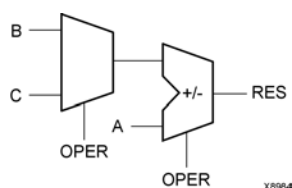
リソース共有 (RESOURCE_SHARING)

リソース共有のコード例

コード例は、ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

この VHDL および Verilog の例では、次の図のような結果になります。

リソース共有の図



リソース共有のピンの説明

I/O ピン	説明
A、B、C	オペランド
OPER	演算セクタ
RES	データ出力

リソース共有の VHDL コード例

```
--
-- Resource Sharing
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity resource_sharing_1 is
    port(A,B,C : in  std_logic_vector(7 downto 0);
         OPER  : in  std_logic;
         RES   : out std_logic_vector(7 downto 0));
end resource_sharing_1;

architecture archi of resource_sharing_1 is
begin

    RES <= A + B when OPER='0' else A - C;

end archi;
```

リソース共有の Verilog コード例

```
//  
// Resource Sharing  
//  
  
module v_resource_sharing_1 (A, B, C, OPER, RES);  
    input  [7:0] A, B, C;  
    input  OPER;  
    output [7:0] RES;  
    wire   [7:0] RES;  
  
    assign RES = !OPER ? A + B : A - C;  
  
endmodule
```

RAM/ROM の HDL コーディング手法

このセクションには、次の内容が含まれます。

- ・ [RAM/ROM の概要](#)
- ・ [RAM/ROM のログ ファイル](#)
- ・ [RAM/ROM 関連の制約](#)
- ・ [RAM/ROM のコード例](#)
- ・ [RAM の初期化のコード例](#)
- ・ [外部ファイルからの RAM の初期化のコード例](#)

RAM/ROM の概要

このセクションでは、次について説明します。

- ・ [RAM の自動識別](#)
- ・ [負のアドレスが設定された RAM および ROM](#)
- ・ [推論された RAM のタイプ](#)
- ・ [ブロックおよび分散 RAM](#)
- ・ [サポートされないブロック RAM の機能](#)
- ・ [スピード重視のインプリメンテーション](#)
- ・ [その他の XST の機能](#)
- ・ [自動 BRAM リソース制御](#)
- ・ [小型の RAM/ROM](#)
- ・ [使用可能な BRAM リソース](#)

RAM の自動識別

HDL コードをアーキテクチャに依存しないようにするために、RAM プリミティブをインスタンス化しない場合は、XST の RAM の自動推論を使用してください。XST では、分散 RAM およびブロック RAM の両方を推論可能で、どちらの RAM でも次の機能がサポートされます。

- ・ 同期書き込み
- ・ 書き込みイネーブル
- ・ RAM イネーブル
- ・ 非同期または同期読み出し
- ・ データ出力ラッチのリセット
- ・ データ出力リセット
- ・ シングル ポート、デュアル ポート、または複数ポートリード
- ・ シングル ポート RAM およびデュアル ポート ライト
- ・ パリティ ビット
- ・ バイト幅の書き込みイネーブル付きブロック RAM
- ・ シンプル デュアル ポート BRAM

負のアドレスが設定された RAM および ROM

負のアドレスが設定された RAM および ROM はサポートされません。

推論された RAM のタイプ

推論される RAM のタイプは、コードの記述方法によって異なります。

- ・ 非同期読み出しが含まれる RAM 記述では、分散 RAM マクロが推論されます。
- ・ 同期読み出しが含まれる RAM 記述では、ブロック RAM マクロが推論されます。ブロック RAM マクロが、実際には分散 RAM マクロを使用してインプリメントされる場合もあります。実際の RAM インプリメンテーションは、マクロ生成機能で決定されます。

ブロックおよび分散 RAM

テンプレートでブロック RAM と分散 RAM のどちらでもインプリメントできる場合は、ブロック RAM がインプリメントされます。[RAM スタイル \(RAM_STYLE\)](#) 制約を使用すると、RAM のインプリメンテーションを制御して必要な RAM タイプを選択できます。

詳細は、次を参照してください。

[デザイン制約](#)

サポートされないブロック RAM の機能

次のブロック RAM の機能は、サポートされていません。

- ・ パリティ ビット
- ・ 各ポートに異なるワード数とビット幅の比率を使用
- ・ シンプル デュアル ポート分散 RAM
- ・ クワッド ポート分散 RAM

スピード重視のインプリメンテーション

XST では、BRAM リソースに RAM が速度重視でインプリメントされるため、速度に関しては良い結果になるものの、エリア重視のインプリメンテーションよりも BRAM リソースが多く必要になってしまいます。XST では、エリア重視の BRAM インプリメンテーションがサポートされないため、エリア重視のインプリメンテーションをする場合は、CORE Generator™ を使用してください。

詳細は、次を参照してください。

[FPGA の最適化](#)

その他の XST の機能

XST では、次が実行できます。

- ・ FSM コンポーネントをインプリメントします。

詳細は、次を参照してください。

[FSM の HDL コーディング手法](#)

- ・ 一般的なロジックをブロック RAM 上にマップします。

詳細は、次を参照してください。

[ブロック RAM へのロジックのマップ](#)

自動 BRAM リソース制御

XST では、ターゲット デバイスの BRAM リソースが自動的に制御されます。[BRAM 使用率 \(BRAM_UTILIZATION_RATIO\)](#) を使用すると、合成中に XST で処理される BRAM ブロック数を制限できます。

小型の RAM/ROM

BRAM リソースに小型の RAM および ROM を強制的にインプリメンテーションするには、[RAM スタイル \(RAM_STYLE\)](#) および [ROM スタイル \(ROM_STYLE\)](#) を使用してください。

XST では、専用のリソースを使用して小型の RAM および ROM をインプリメントすることで、デザイン速度を改善します。RAM および ROM は、サイズが次の表の規則に従っている場合は小型のメモリと考えられます。

デバイス	サイズ (ビット) * 幅 (ビット)
Virtex®-4	<= 512
Virtex-5	<= 512

使用可能な BRAM リソース

XST では、次の計算式で、推論に使用可能な BRAM リソースの数が計算されます。

$$\text{Total_Number_of_Available_BRAMs} - \text{Number_of_Reserved_BRAMs}$$

説明 :

Total_Number_of_Available_BRAMs は、[BRAM 使用率 \(BRAM_UTILIZATION_RATIO\)](#) 制約で指定した BRAM 数になります。デフォルトは 100% です。

Number of Reserved_BRAMs は、次の合計です。

- ・ UNISIM ライブラリからの Hardware Description Language (HDL) コードに含まれるインスタンシエート済みの BRAM 数
- ・ [RAM スタイル \(RAM_STYLE\)](#) および [ROM スタイル \(ROM_STYLE\)](#) 制約により BRAM として強制的にインプリメントされた RAM の数
- ・ BRAM マッピング最適化 (BRAM_MAP) 制約を使用して生成される BRAM の数

XST では、使用可能な BRAM リソースがある箇所に BRAM を使用して推論された最大の RAM および ROM をインプリメントし、分散リソースに最小の RAM および ROM をインプリメントします。

Number_of_Reserved_BRAMs の数が使用可能なリソースを超えると、それらがブロック RAM としてインプリメントされ、推論された RAM はすべて分散メモリにインプリメントされます。

このプロセスが終了した直後、2 つの小型のシングル ポート BRAM が 1 つの BRAM プリミティブに自動的にパックされます。これは、[自動 BRAM パッキング \(AUTO_BRAM_PACKING\)](#) で制御されます。この制約は、デフォルトではオフになっています。

詳細は、次を参照してください。

- ・ [BRAM 使用率 \(BRAM_UTILIZATION_RATIO\)](#)
- ・ [自動 BRAM パッキング \(AUTO_BRAM_PACKING\)](#)

RAM/ROM のログ ファイル

XST の RAM および ROM のログ ファイルには、次がレポートされます。

- ・ 認識された RAM のタイプとサイズ
- ・ その I/O ポートに関するすべての情報

RAM の認識段階

段階	XST の動作
HDL 合成	HDL ソースコードにあるメモリ構造が認識されます。
アドバンス HDL 合成	特定のメモリのインプリメンテーション方法 (ブロック型と分散型のどちらのメモリリソースを使用するか) が決定されます。

RAM/ROM のログ ファイルの例

```
=====
* HDL Synthesis *
=====

Synthesizing Unit <rams_16>.
Related source file is "rams_16.vhd".
Found 64x16-bit dual-port RAM <Mram_RAM> for signal <RAM>.
Found 16-bit register for signal <doa>.
Found 16-bit register for signal <dob>.
Summary:
inferred 1 RAM(s).
inferred 32 D-type flip-flop(s).
Unit <rams_16> synthesized.

=====

HDL Synthesis Report

Macro Statistics
# RAMs : 1
  64x16-bit dual-port RAM : 1
# Registers: 2
  16-bit register : 2

=====

=====
* Advanced HDL Synthesis*
=====

Synthesizing (advanced) Unit <rams_16>.
INFO:Xst - The RAM <Mram_RAM> will be implemented as a BLOCK RAM, absorbing
the following register(s): <doa> <dob>

-----
```

```

| ram_type          | Block          |          |
-----
| Port A
|   aspect ratio    | 64-word x 16-bit |          |
|   mode            | write-first      |          |
|   clkA            | connected to signal <clka> | rise    |
|   enA             | connected to signal <ena> | high    |
|   weA             | connected to internal <wea> | high    |
|   addrA           | connected to signal <addra> |          |
|   diA             | connected to internal <dia> |          |
|   doA             | connected to signal <doa> |          |
-----
| optimization      | speed           |          |
=====

```

```

-----
| ram_type          | Block          |          |
-----
| Port B
|   aspect ratio    | 64-word x 16-bit |          |
|   mode            | write-first      |          |
|   clkB            | connected to signal <clkb> | rise    |
|   enB             | connected to signal <enb> | high    |
|   weB             | connected to internal <web> | high    |
|   addrB           | connected to signal <addrb> |          |
|   diB             | connected to internal <dib> |          |
|   doB             | connected to signal <dob> |          |
-----
| optimization      | speed           |          |
=====

```

```

-----
Unit <rams_16> synthesized (advanced).

```

```

=====
Advanced HDL Synthesis Report

```

```
Macro Statistics
```

```
# RAMs : 1
```

```
64x16-bit dual-port block RAM : 1
```

RAM/ROM 関連の制約

- ・ [BRAM 使用率 \(BRAM_UTILIZATION_RATIO\)](#)
- ・ [自動 BRAM パッキング \(AUTO_BRAM_PACKING\)](#)
- ・ [RAM の抽出 \(RAM_EXTRACT\)](#)
- ・ [RAM スタイル \(RAM_STYLE\)](#)
- ・ [ROM の抽出 \(ROM_EXTRACT\)](#)
- ・ [ROM スタイル \(ROM_STYLE\)](#)

XST では、1 つのブロック RAM プリミティブにインプリメント可能な推論された RAM に [LOC](#) および [RLOC](#) 制約を使用できます。[LOC](#) および [RLOC](#) 制約は NGC ネットリストに渡されます。

RAM/ROM のコード例

コード例は、ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

関連項目

[RAM の初期化のコード例](#)

[外部ファイルからの RAM の初期化のコード例](#)

RAM/ROM のコード例

次のデバイスのブロック RAM リソースでは、異なる読み出し/書き込み同期モードがサポートされます。

- ・ Virtex®-4
- ・ Virtex-5
- ・ Spartan®-3
- ・ Spartan-3E
- ・ Spartan-3A

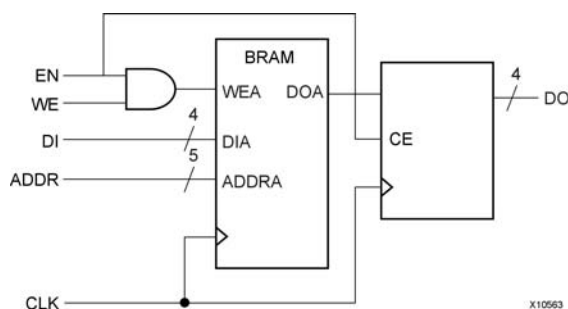
次のコード例では、シングル ポート ブロック RAM を記述しています。これらの例を応用して、デュアル ポート ブロック RAM を記述できます。デュアル ポート ブロック RAM には、各ポートにそれぞれ異なる読み出し/書き込みモードをコンフィギュレーションできます。XST では、これらのモードを自動的に推論できます。

次の表に、デバイスによりサポートされる読み出し/書き込みモードと、XST による処理方法を示します。

読み出し/書き込みモードのサポート

デバイス	推論モード	ビヘイビア
Spartan-3	WRITE_FIRST	マクロの推論および生成
Spartan-3E	READ_FIRST	NCF ファイル内で生成されたブロック RAM に適切な制約 (WRITE_MODE、WRITE_MODE_A、WRITE_MODE_B) を設定
Spartan-3A	NO_CHANGE	
Virtex-4		
Virtex-5		
CPLD	none	RAM の推論は完全に無効

READ_FIRST モードのシングル ポート RAM の図



READ_FIRST モードのシングル ポート RAM のピンの説明

I/O ピン	説明
clk	クロック (立ち上がりエッジ)
we	同期書き込みイネーブル (アクティブ High)
en	クロック イネーブル
addr	読み出し/書き込みアドレス
di	データ入力
do	データ出力

READ_FIRST モードのシングル ポート RAM の VHDL コード例 1

```

--
-- Read-First Mode
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity rams_01 is
    port (clk : in std_logic;
          we : in std_logic;
          en : in std_logic;
          addr : in std_logic_vector(5 downto 0);
          di : in std_logic_vector(15 downto 0);

```

```

        do    : out std_logic_vector(15 downto 0));
end rams_01;

architecture syn of rams_01 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal RAM: ram_type;
begin

    process (clk)
    begin
        if clk'event and clk = '1' then
            if en = '1' then
                if we = '1' then
                    RAM(conv_integer(addr)) <= di;
                end if;
                do <= RAM(conv_integer(addr)) ;
            end if;
        end if;
    end process;

end syn;

```

READ_FIRST モードのシングル ポート RAM の Verilog コード例 1

```

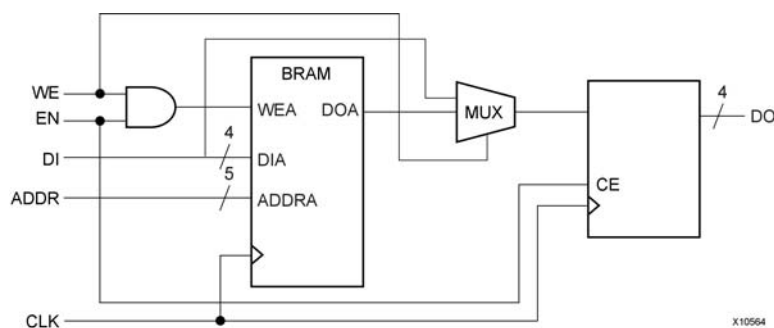
//
// Read-First Mode
//
module v_rams_01 (clk, en, we, addr, di, do);
    input  clk;
    input  we;
    input  en;
    input  [5:0] addr;
    input  [15:0] di;
    output [15:0] do;
    reg    [15:0] RAM [63:0];
    reg    [15:0] do;

    always @(posedge clk)
    begin
        if (en)
        begin
            if (we)
                RAM[addr]<=di;
                do <= RAM[addr];
            end
        end
    end

endmodule

```

WRITE_FIRST モードのシングル ポート RAM の図



WRITE_FIRST モードのシングル ポート RAM のピンの説明

I/O ピン	説明
clk	クロック (立ち上がりエッジ)
we	同期書き込みイネーブル (アクティブ High)
en	クロック イネーブル
addr	読み出し/書き込みアドレス
di	データ入力
do	データ出力

WRITE_FIRST モードのシングル ポート RAM の VHDL コード例 1

```
--
-- Write-First Mode (template 1)
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_02a is
    port (clk : in std_logic;
          we : in std_logic;
          en : in std_logic;
          addr : in std_logic_vector(5 downto 0);
          di : in std_logic_vector(15 downto 0);
          do : out std_logic_vector(15 downto 0));
end rams_02a;

architecture syn of rams_02a is
    type ram_type is array (63 downto 0)
        of std_logic_vector (15 downto 0);
    signal RAM : ram_type;
begin

    process (clk)

```

```
begin
    if clk'event and clk = '1' then
        if en = '1' then
            if we = '1' then
                RAM(conv_integer(addr)) <= di;
                do <= di;
            else
                do <= RAM( conv_integer(addr));
            end if;
        end if;
    end if;
end process;

end syn;
```

WRITE_FIRST モードのシングル ポート RAM の VHDL コード例 2

```
--
-- Write-First Mode (template 2)
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_02b is
port (clk : in std_logic;
      we : in std_logic;
      en : in std_logic;
      addr : in std_logic_vector(5 downto 0);
      di : in std_logic_vector(15 downto 0);
      do : out std_logic_vector(15 downto 0));
end rams_02b;

architecture syn of rams_02b is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal RAM : ram_type;
    signal read_addr: std_logic_vector(5 downto 0);
begin

    process (clk)
    begin
        if clk'event and clk = '1' then
            if en = '1' then
                if we = '1' then
                    ram(conv_integer(addr)) <= di;
                end if;
                read_addr <= addr;
            end if;
        end if;
    end process;

    do <= ram(conv_integer(read_addr));

end syn;
```

WRITE_FIRST モードのシングル ポート RAM の Verilog コード例 1

```
//  
// Write-First Mode (template 1)  
//  
  
module v_rams_02a (clk, we, en, addr, di, do);  
  
    input  clk;  
    input  we;  
    input  en;  
    input  [5:0] addr;  
    input  [15:0] di;  
    output [15:0] do;  
    reg    [15:0] RAM [63:0];  
    reg    [15:0] do;  
  
    always @(posedge clk)  
    begin  
        if (en)  
        begin  
            if (we)  
            begin  
                RAM[addr] <= di;  
                do <= di;  
            end  
            else  
                do <= RAM[addr];  
            end  
        end  
    end  
endmodule
```

WRITE_FIRST モードのシングル ポート RAM の Verilog コード例 2

```
//  
// Write-First Mode (template 2)  
//  
  
module v_rams_02b (clk, we, en, addr, di, do);  
  
    input  clk;  
    input  we;  
    input  en;  
    input  [5:0] addr;  
    input  [15:0] di;  
    output [15:0] do;  
    reg    [15:0] RAM [63:0];  
    reg    [5:0] read_addr;
```

```

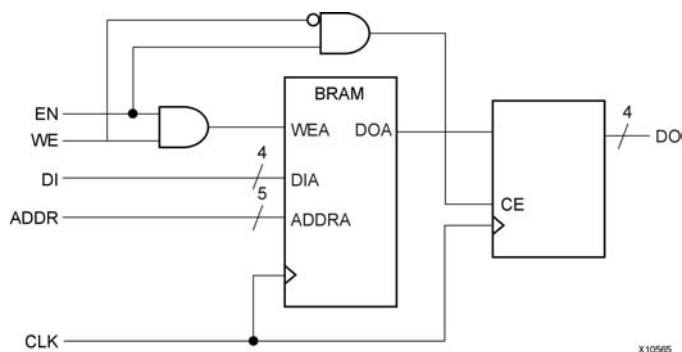
always @(posedge clk)
begin
    if (en)
    begin
        if (we)
            RAM[addr] <= di;
        read_addr <= addr;
    end
end

assign do = RAM[read_addr];

endmodule

```

NO_CHANGE モードのシングル ポート RAM の図



NO_CHANGE モードのシングル ポート RAM のピンの説明

I/O ピン	説明
clk	クロック (立ち上がりエッジ)
we	同期書き込みイネーブル (アクティブ High)
en	クロック イネーブル
addr	読み出し/書き込みアドレス
di	データ入力
do	データ出力

NO_CHANGE モードのシングル ポート RAM の VHDL コード例 2

```

--
-- No-Change Mode (template 1)
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_03 is

```

```

    port (clk : in std_logic;
          we  : in std_logic;
          en  : in std_logic;
          addr : in std_logic_vector(5 downto 0);
          di   : in std_logic_vector(15 downto 0);
          do   : out std_logic_vector(15 downto 0));
end rams_03;

architecture syn of rams_03 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal RAM : ram_type;
begin

    process (clk)
    begin
        if clk'event and clk = '1' then
            if en = '1' then
                if we = '1' then
                    RAM(conv_integer(addr)) <= di;
                else
                    do <= RAM( conv_integer(addr));
                end if;
            end if;
        end if;
    end process;

end syn;

```

NO_CHANGE モードのシングル ポート RAM の Verilog コード例 2

```

//
// No-Change Mode (template 1)
//

module v_rams_03 (clk, we, en, addr, di, do);

    input  clk;
    input  we;
    input  en;
    input  [5:0] addr;
    input  [15:0] di;
    output [15:0] do;
    reg    [15:0] RAM [63:0];
    reg    [15:0] do;

    always @(posedge clk)
    begin
        if (en)
        begin
            if (we)

```

```

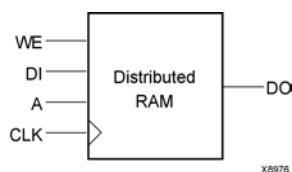
        RAM[addr] <= di;
    else
        do <= RAM[addr];
    end
end
end

endmodule

```

次の記述では、分散 RAM のみにインプリメント可能です。

非同期読み出し付きシングル ポート RAM の図



非同期読み出し付きシングル ポート RAM のピンの説明

I/O ピン	説明
clk	クロック (立ち上がりエッジ)
we	同期書き込みイネーブル (アクティブ High)
a	読み出し/書き込みアドレス
di	データ入力
do	データ出力

非同期読み出し付きシングル ポート RAM の VHDL コード例

```

--
-- Single-Port RAM with Asynchronous Read
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_04 is
    port (clk : in std_logic;
          we  : in std_logic;
          a   : in std_logic_vector(5 downto 0);
          di  : in std_logic_vector(15 downto 0);
          do  : out std_logic_vector(15 downto 0));
end rams_04;

architecture syn of rams_04 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal RAM : ram_type;
begin

```

```

process (clk)
begin
    if (clk'event and clk = '1') then
        if (we = '1') then
            RAM(conv_integer(a)) <= di;
        end if;
    end if;
end process;

do <= RAM(conv_integer(a));

end syn;

```

非同期読み出し付きシングル ポート RAM の Verilog コード例

```

//
// Single-Port RAM with Asynchronous Read
//

module v_rams_04 (clk, we, a, di, do);

    input  clk;
    input  we;
    input  [5:0] a;
    input  [15:0] di;
    output [15:0] do;
    reg    [15:0] ram [63:0];

    always @(posedge clk) begin
        if (we)
            ram[a] <= di;
    end

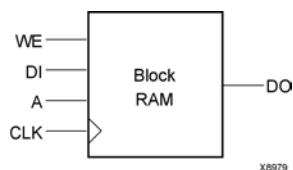
    assign do = ram[a];

endmodule

```

次の記述は、真の同期読み出しをインプリメントします。同期読み出しは Virtex デバイスのブロック RAM の機能で、クロック エッジで読み出しアドレスが格納されます。次の記述は、次の図に示すようにブロック RAM に直接インプリメントできます 同じ記述は、分散 RAM にもマップ可能です。

同期読み出し (透過読み出し) 付きシングル ポート RAM の図



同期読み出し (透過読み出し) 付きシングル ポート RAM のピンの説明

I/O ピン	説明
clk	クロック (立ち上がりエッジ)
we	同期書き込みイネーブル (アクティブ High)
a	読み出し/書き込みアドレス
di	データ入力
do	データ出力

同期読み出し (透過読み出し) 付きシングル ポート RAM の VHDL コード例

```
--
-- Single-Port RAM with Synchronous Read (Read Through)
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_07 is
    port (clk : in std_logic;
          we  : in std_logic;
          a   : in std_logic_vector(5 downto 0);
          di  : in std_logic_vector(15 downto 0);
          do  : out std_logic_vector(15 downto 0));
end rams_07;

architecture syn of rams_07 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal RAM : ram_type;
    signal read_a : std_logic_vector(5 downto 0);
begin

    process (clk)
    begin
        if (clk'event and clk = '1') then
            if (we = '1') then
                RAM(conv_integer(a)) <= di;
            end if;
            read_a <= a;
        end if;
    end process;

    do <= RAM(conv_integer(read_a));

end syn;
```

同期読み出し (透過読み出し) 付きシングル ポート RAM の Verilog コード例

```
//
// Single-Port RAM with Synchronous Read (Read Through)
//

module v_rams_07 (clk, we, a, di, do);

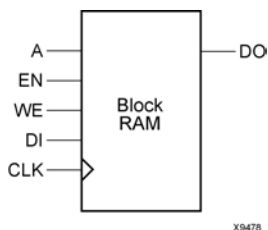
    input  clk;
    input  we;
    input  [5:0] a;
    input  [15:0] di;
    output [15:0] do;
    reg    [15:0] ram [63:0];
    reg    [5:0] read_a;

    always @(posedge clk) begin
        if (we)
            ram[a] <= di;
        read_a <= a;
    end

    assign do = ram[read_a];

endmodule
```

イネーブル付きシングル ポート RAM の図



イネーブル付きシングル ポート RAM のピンの説明

I/O ピン	説明
clk	クロック (立ち上がりエッジ)
en	グローバル イネーブル
we	同期書き込みイネーブル (アクティブ High)
a	読み出し/書き込みアドレス
di	データ入力
do	データ出力

イネーブル付きシングル ポート RAM の VHDL コード例

```
--
-- Single-Port RAM with Enable
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_08 is
    port (clk : in std_logic;
          en  : in std_logic;
          we  : in std_logic;
          a   : in std_logic_vector(5 downto 0);
          di  : in std_logic_vector(15 downto 0);
          do  : out std_logic_vector(15 downto 0));
end rams_08;

architecture syn of rams_08 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal RAM : ram_type;
    signal read_a : std_logic_vector(5 downto 0);
begin

    process (clk)
    begin
        if (clk'event and clk = '1') then
            if (en = '1') then
                if (we = '1') then
                    RAM(conv_integer(a)) <= di;
                end if;
                read_a <= a;
            end if;
        end if;
    end process;

    do <= RAM(conv_integer(read_a));

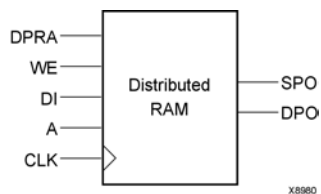
end syn;
```

イネーブル付きシングル ポート RAM の Verilog コード例

```
//  
// Single-Port RAM with Enable  
//  
  
module v_rams_08 (clk, en, we, a, di, do);  
  
    input  clk;  
    input  en;  
    input  we;  
    input  [5:0] a;  
    input  [15:0] di;  
    output [15:0] do;  
    reg    [15:0] ram [63:0];  
    reg    [5:0] read_a;  
  
    always @(posedge clk) begin  
        if (en)  
            begin  
                if (we)  
                    ram[a] <= di;  
                read_a <= a;  
            end  
        end  
  
        assign do = ram[read_a];  
  
    endmodule
```

次の図では、2 つの出力ポートが使用されています。これは、分散 RAM のみに直接マップ可能です。

非同期読み出し付きデュアル ポート RAM の図



非同期読み出し付きデュアル ポート RAM のピンの説明

I/O ピン	説明
clk	クロック (立ち上がりエッジ)
we	同期書き込みイネーブル (アクティブ High)
a	書き込みアドレス/プライマリ読み出しアドレス
DPRA	デュアル読み出しアドレス
di	データ入力
SPO	プライマリ出力ポート
DPO	デュアル出力ポート

非同期読み出し付きデュアル ポート RAM の VHDL コード例

```
--
-- Dual-Port RAM with Asynchronous Read
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_09 is
    port (clk : in std_logic;
          we : in std_logic;
          a : in std_logic_vector(5 downto 0);
          dpra : in std_logic_vector(5 downto 0);
          di : in std_logic_vector(15 downto 0);
          spo : out std_logic_vector(15 downto 0);
          dpo : out std_logic_vector(15 downto 0));
end rams_09;

architecture syn of rams_09 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal RAM : ram_type;
begin

    process (clk)
    begin
        if (clk'event and clk = '1') then
            if (we = '1') then
                RAM(conv_integer(a)) <= di;
            end if;
        end if;
    end process;

    spo <= RAM(conv_integer(a));
    dpo <= RAM(conv_integer(dpra));
```

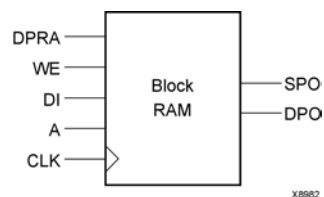
```
end syn;
```

非同期読み出し付きデュアル ポート RAM の Verilog コード例

```
//  
// Dual-Port RAM with Asynchronous Read  
//  
  
module v_rams_09 (clk, we, a, dpra, di, spo, dpo);  
  
    input  clk;  
    input  we;  
    input  [5:0] a;  
    input  [5:0] dpra;  
    input  [15:0] di;  
    output [15:0] spo;  
    output [15:0] dpo;  
    reg    [15:0] ram [63:0];  
  
    always @(posedge clk) begin  
        if (we)  
            ram[a] <= di;  
    end  
  
    assign spo = ram[a];  
    assign dpo = ram[dpra];  
  
endmodule
```

次の記述は、次の図に示すようにブロック RAM に直接マップできます 分散 RAM にもインプリメントできます。

同期読み出し（透過読み出し）付きデュアル ポート RAM の図



同期読み出し (透過読み出し) 付きデュアル ポート RAM のピンの説明

I/O ピン	説明
clk	クロック (立ち上がりエッジ)
we	同期書き込みイネーブル (アクティブ High)
a	書き込みアドレス/プライマリ読み出しアドレス
DPRA	デュアル読み出しアドレス
di	データ入力
SPO	プライマリ出力ポート
DPO	デュアル出力ポート

同期読み出し (透過読み出し) 付きデュアル ポート RAM の VHDL コード例

```
--
-- Dual-Port RAM with Synchronous Read (Read Through)
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_11 is
    port (clk      : in std_logic;
          we       : in std_logic;
          a        : in std_logic_vector(5 downto 0);
          dpra     : in std_logic_vector(5 downto 0);
          di       : in std_logic_vector(15 downto 0);
          spo      : out std_logic_vector(15 downto 0);
          dpo      : out std_logic_vector(15 downto 0));
end rams_11;

architecture syn of rams_11 is
    type ram_type is array (63 downto 0)
        of std_logic_vector (15 downto 0);
    signal RAM : ram_type;
    signal read_a : std_logic_vector(5 downto 0);
    signal read_dpra : std_logic_vector(5 downto 0);
begin

    process (clk)
    begin
        if (clk'event and clk = '1') then
            if (we = '1') then
                RAM(conv_integer(a)) <= di;
            end if;
            read_a <= a;
            read_dpra <= dpra;
        end if;
    end process;
end architecture syn;
```

```

        end process;

        spo <= RAM(conv_integer(read_a));
        dpo <= RAM(conv_integer(read_dpra));

    end syn;

```

同期読み出し(透過読み出し) 付きデュアル ポート RAM の Verilog コード例

```

//
// Dual-Port RAM with Synchronous Read (Read Through)
//

module v_rams_11 (clk, we, a, dpra, di, spo, dpo);

    input  clk;
    input  we;
    input  [5:0] a;
    input  [5:0] dpra;
    input  [15:0] di;
    output [15:0] spo;
    output [15:0] dpo;
    reg    [15:0] ram [63:0];
    reg    [5:0] read_a;
    reg    [5:0] read_dpra;

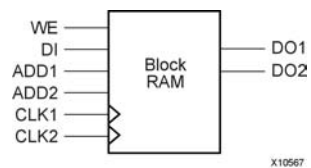
    always @(posedge clk) begin
        if (we)
            ram[a] <= di;
        read_a <= a;
        read_dpra <= dpra;
    end

    assign spo = ram[read_a];
    assign dpo = ram[read_dpra];

endmodule

```

同期読み出し(透過読み出し) および 2 つのクロック付きデュアル ポート RAM の図



同期読み出し (透過読み出し) および 2 つのクロック付きデュアル ポート RAM のピンの説明

I/O ピン	説明
CLK1	書き込み/プライマリ読み出しクロック (立ち上がりエッジ)
CLK2	デュアル読み出しクロック (立ち上がりエッジ)
we	同期書き込みイネーブル (アクティブ High)
add1	書き込み/プライマリ読み出しアドレス
add2	デュアル読み出しアドレス
di	データ入力
do1	プライマリ出力ポート
do2	デュアル出力ポート

同期読み出し (透過読み出し) および 2 つのクロック付きデュアル ポート RAM の VHDL コード例

```
--
-- Dual-Port RAM with Synchronous Read (Read Through)
-- using More than One Clock
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_12 is
    port (clk1 : in std_logic;
          clk2 : in std_logic;
          we   : in std_logic;
          add1 : in std_logic_vector(5 downto 0);
          add2 : in std_logic_vector(5 downto 0);
          di   : in std_logic_vector(15 downto 0);
          do1  : out std_logic_vector(15 downto 0);
          do2  : out std_logic_vector(15 downto 0));
end rams_12;

architecture syn of rams_12 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal RAM : ram_type;
    signal read_add1 : std_logic_vector(5 downto 0);
    signal read_add2 : std_logic_vector(5 downto 0);
begin

    process (clk1)
    begin
        if (clk1'event and clk1 = '1') then
            if (we = '1') then
```

```

        RAM(conv_integer(add1)) <= di;
    end if;
    read_add1 <= add1;
end if;
end process;

do1 <= RAM(conv_integer(read_add1));

process (clk2)
begin
    if (clk2'event and clk2 = '1') then
        read_add2 <= add2;
    end if;
end process;

do2 <= RAM(conv_integer(read_add2));

end syn;
```

同期読み出し (透過読み出し) および 2 つのクロック付きデュアル ポート RAM の Verilog コード例

```

//
// Dual-Port RAM with Synchronous Read (Read Through)
// using More than One Clock
//

module v_rams_12 (clk1, clk2, we, add1, add2, di, do1, do2);

    input  clk1;
    input  clk2;
    input  we;
    input  [5:0] add1;
    input  [5:0] add2;
    input  [15:0] di;
    output [15:0] do1;
    output [15:0] do2;
    reg    [15:0] ram [63:0];
    reg    [5:0] read_add1;
    reg    [5:0] read_add2;

    always @(posedge clk1) begin
        if (we)
            ram[add1] <= di;
        read_add1 <= add1;
    end

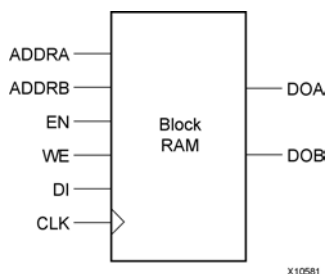
    assign do1 = ram[read_add1];

    always @(posedge clk2) begin
        read_add2 <= add2;
    end

    assign do2 = ram[read_add2];
```

```
endmodule
```

1 つのイネーブル信号で両方のポートを制御するデュアル ポート RAM の図



1 つのイネーブル信号で両方のポートを制御するデュアル ポート RAM のピンの説明

I/O ピン	説明
clk	クロック (立ち上がりエッジ)
en	プライマリ グローバル イネーブル (アクティブ High)
we	プライマリ同期書き込みイネーブル (アクティブ High)
ADDRA	書き込みアドレス/プライマリ読み出しアドレス
ADDRb	デュアル読み出しアドレス
di	プライマリ データ入力
DOA	プライマリ出力ポート
DOB	デュアル出力ポート

1 つのイネーブル信号で両方のポートを制御するデュアル ポート RAM の VHDL コード例

```
--
-- Dual-Port RAM with One Enable Controlling Both Ports
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_13 is
    port (clk    : in std_logic;
          en     : in std_logic;
          we     : in std_logic;
          addra  : in std_logic_vector(5 downto 0);
          addrb  : in std_logic_vector(5 downto 0);
          di     : in std_logic_vector(15 downto 0);
          doa    : out std_logic_vector(15 downto 0);
          dob    : out std_logic_vector(15 downto 0));
end rams_13;
```

```

architecture syn of rams_13 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal RAM : ram_type;
    signal read_addra : std_logic_vector(5 downto 0);
    signal read_addrb : std_logic_vector(5 downto 0);
begin

    process (clk)
    begin
        if (clk'event and clk = '1') then
            if (en = '1') then
                if (we = '1') then
                    RAM(conv_integer(addra)) <= di;
                end if;

                read_addra <= addra;
                read_addrb <= addrb;

            end if;
        end if;
    end process;

    doa <= RAM(conv_integer(read_addra));
    dob <= RAM(conv_integer(read_addrb));

end syn;

```

1 つのイネーブル信号で両方のポートを制御するデュアル ポート RAM の Verilog コード例

```

//
// Dual-Port RAM with One Enable Controlling Both Ports
//

module v_rams_13 (clk, en, we, addra, addrb, di, doa, dob);

    input  clk;
    input  en;
    input  we;
    input  [5:0] addra;
    input  [5:0] addrb;
    input  [15:0] di;
    output [15:0] doa;
    output [15:0] dob;
    reg    [15:0] ram [63:0];
    reg    [5:0] read_addra;
    reg    [5:0] read_addrb;

    always @(posedge clk) begin

```

```

        if (en)
        begin
            if (we)
                ram[addra] <= di;
            read_addra <= addra;
            read_addrb <= addrb;
        end
    end

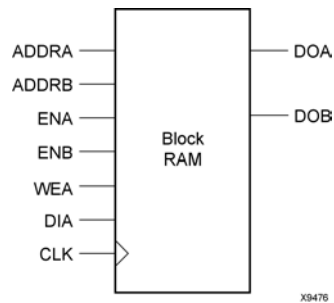
    assign doa = ram[read_addra];
    assign dob = ram[read_addrb];

endmodule

```

次の記述は、次の図に示すようにブロック RAM に直接マップできます

各ポートにそれぞれイネーブル信号があるデュアル ポート RAM の図



各ポートにそれぞれイネーブル信号があるデュアル ポート RAM の ピンの説明

I/O ピン	説明
clk	クロック (立ち上がりエッジ)
ena	プライマリ グローバル イネーブル (アクティブ High)
enb	デュアル グローバル イネーブル (アクティブ High)
wea	プライマリ同期書き込みイネーブル (アクティブ High)
addra	書き込みアドレス/プライマリ読み出しアドレス
addrb	デュアル読み出しアドレス
dia	プライマリ データ入力
doa	プライマリ出力ポート
dob	デュアル出力ポート

各ポートにそれぞれイネーブル信号があるデュアル ポート RAM の VHDL コード例

```
--
-- Dual-Port RAM with Enable on Each Port
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_14 is
    port (clk      : in std_logic;
          ena      : in std_logic;
          enb      : in std_logic;
          wea      : in std_logic;
          addra    : in std_logic_vector(5 downto 0);
          addrb    : in std_logic_vector(5 downto 0);
          dia      : in std_logic_vector(15 downto 0);
          doa      : out std_logic_vector(15 downto 0);
          dob      : out std_logic_vector(15 downto 0));
end rams_14;

architecture syn of rams_14 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal RAM : ram_type;
    signal read_addra : std_logic_vector(5 downto 0);
    signal read_addrb : std_logic_vector(5 downto 0);
begin

    process (clk)
    begin
        if (clk'event and clk = '1') then

            if (ena = '1') then
                if (wea = '1') then
                    RAM (conv_integer(addra)) <= dia;
                end if;
                read_addra <= addra;
            end if;

            if (enb = '1') then
                read_addrb <= addrb;
            end if;

        end if;
    end process;

    doa <= RAM(conv_integer(read_addra));
    dob <= RAM(conv_integer(read_addrb));
```

```
end syn;
```

各ポートにそれぞれイネーブル信号があるデュアル ポート RAM の Verilog コード例

```
//
// Dual-Port RAM with Enable on Each Port
//

module v_rams_14 (clk,ena,enb,wea,addra,addrb,dia,doa,dob);

    input  clk;
    input  ena;
    input  enb;
    input  wea;
    input  [5:0] addra;
    input  [5:0] addrb;
    input  [15:0] dia;
    output [15:0] doa;
    output [15:0] dob;
    reg    [15:0] ram [63:0];
    reg    [5:0] read_addra;
    reg    [5:0] read_addrb;

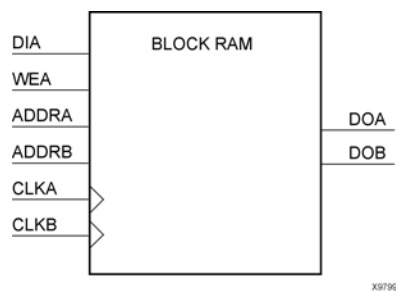
    always @(posedge clk) begin
        if (ena)
            begin
                if (wea)
                    ram[addra] <= dia;
                read_addra <= addra;
            end

            if (enb)
                read_addrb <= addrb;
        end

        assign doa = ram[read_addra];
        assign dob = ram[read_addrb];
    end

endmodule
```

異なるクロックが付いたデュアル ポート ブロック RAM の図



異なるクロックが付いたデュアル ポート ブロック RAM のピンの説明

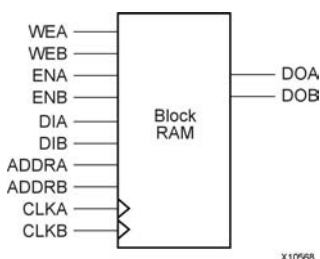
I/O ピン	説明
clka	クロック (立ち上がりエッジ)
clkb	クロック (立ち上がりエッジ)
wea	プライマリ同期書き込みイネーブル (アクティブ High)
addra	書き込みアドレス/プライマリ読み出しアドレス
addrb	デュアル読み出しアドレス
dia	プライマリ データ入力
doa	プライマリ出力ポート
dob	デュアル出力ポート

書き込みポートが 2 つあるデュアル ポート ブロック RAM は、VHDL と Verilog の両方でサポートされます。

デュアル書き込みポートでは、データポートが 2 つあるだけでなく、各ポートに個別の書き込みクロックおよび書き込みイネーブルを使用できることがあります。デュアル ポート ブロック RAM には 2 つのクロックがあり、1 つはプライマリの読み出しと書き込みポートで共有され、もう 1 つはセカンダリの読み出しと書き込みポートで共有されるので、各ポートに個別の書き込みクロックを使用する場合、読み出しクロックも個別になることに注意してください。

このタイプのブロック RAM は、VHDL では SHARED 変数を使用して記述されています。XST の VHDL アナライザーではこの SHARED 変数が認識されますが、有効な RAM マクロが記述されていないと、HDL 合成段階でエラーが発生します。

2 つの書き込みポートがあるデュアル ポート ブロック RAM の図



2 つの書き込みポートがあるデュアル ポート ブロック RAM のピンの説明

I/O ピン	説明
CLKA、CLKB	クロック (立ち上がりエッジ)
ena	プライマリ グローバル イネーブル (アクティブ High)
enb	デュアル グローバル イネーブル (アクティブ High)
wea, web	プライマリ同期書き込みイネーブル (アクティブ High)

I/O ピン	説明
addra	書き込みアドレス/プライマリ読み出しアドレス
addrb	デュアル読み出しアドレス
dia	プライマリ データ入力
dib	デュアル データ入力
doa	プライマリ出力ポート
dob	デュアル出力ポート

2 つの書き込みポートがあるデュアル ポート ブロックRAM の VHDL コード例

次は、一般的なコード例で、異なるクロック、イネーブル、書き込みイネーブルが含まれます。

```
--
-- Dual-Port Block RAM with Two Write Ports
--

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity rams_16 is
    port(clka : in std_logic;
         clkb : in std_logic;
         ena : in std_logic;
         enb : in std_logic;
         wea : in std_logic;
         web : in std_logic;
         addra : in std_logic_vector(5 downto 0);
         addrb : in std_logic_vector(5 downto 0);
         dia : in std_logic_vector(15 downto 0);
         dib : in std_logic_vector(15 downto 0);
         doa : out std_logic_vector(15 downto 0);
         dob : out std_logic_vector(15 downto 0));
end rams_16;

architecture syn of rams_16 is
    type ram_type is array (63 downto 0) of std_logic_vector(15 downto 0);
    shared variable RAM : ram_type;
begin

    process (CLKA)
    begin
        if CLKA'event and CLKA = '1' then
            if ENA = '1' then
                if WEA = '1' then
                    RAM(conv_integer(ADDRA)) := DIA;
                end if;
                DOA <= RAM(conv_integer(ADDRA));
            end if;
        end if;
    end process;
end architecture;
```

```

        end if;
    end if;
end process;

process (CLKB)
begin
    if CLKB'event and CLKB = '1' then
        if ENB = '1' then
            if WEB = '1' then
                RAM(conv_integer(ADDRB)) := DIB;
            end if;
            DOB <= RAM(conv_integer(ADDRB));
        end if;
    end if;
end process;

end syn;
```

SHARED 変数があるため、各ポートに対する同期読み出し/書き込みの記述がシングル書き込みポートがある RAM で推奨されるコード例とは異なる場合があります。コードを記述する順序が重要なので注意してください。

2 つの書き込みポートがあるデュアル ポート ブロックRAM の Verilog コード例

次は、一般的なコード例で、異なるクロック、イネーブル、書き込みイネーブルが含まれます。

```
//
// Dual-Port Block RAM with Two Write Ports
//
module v_rams_16 (clka, clkb, ena, enb, wea, web, addra, addrb, dia, dib, doa, dob);

    input  clka, clkb, ena, enb, wea, web;
    input  [5:0]  addra, addrb;
    input  [15:0] dia, dib;
    output [15:0] doa, dob;
    reg    [15:0] ram [63:0];
    reg    [15:0] doa, dob;

    always @(posedge clka) begin
        if (ena)
            begin
                if (wea)
                    ram[addra] <= dia;
                doa <= ram[addra];
            end
        end

    always @(posedge clkb) begin
        if (enb)
            begin
                if (web)
                    ram[addrb] <= dib;
                dob <= ram[addrb];
            end
        end

    endmodule
```

WRITE_FIRST の同期の VHDL コード例 1

```
process (CLKA)
begin
    if CLKA'event and CLKA = '1' then
        if WEA = '1' then
            RAM(conv_integer(ADDR_A)) := DIA;
            DOA <= DIA;
        else
            DOA <= RAM(conv_integer(ADDR_A));
        end if;
    end if;
end process;
```

WRITE_FIRST の同期の VHDL コード例 2

この例では、書き込みポートの記述の後に読み出しポートを記述する必要があります。

```
process (CLKA)
begin
    if CLKA'event and CLKA = '1' then
        if WEA = '1' then
            RAM(conv_integer(ADDR_A)) := DIA;
        end if;
        DOA <= RAM(conv_integer(ADDR_A)); -- The read statement must come
                                         -- AFTER the write statement
    end if;
end process;
```

このテンプレートは、次に示すシングル書き込みポート RAM の READ_FIRST の同期のテンプレートと、信号および変数が異なることを除き同じに見えますが、機能が異なります。

```
signal RAM : RAMtype;

process (CLKA)
begin
    if CLKA'event and CLKA = '1' then
        if WEA = '1' then
            RAM(conv_integer(ADDR_A)) <= DIA;
        end if;
        DOA <= RAM(conv_integer(ADDR_A));
    end if;
end process;
```

READ_FIRST の同期のコード例

READ_FIRST の同期を記述する場合は、書き込みポートを記述する前に読み出しポートを記述する必要があります。

```
process (CLKA)
begin
  if CLKA'event and CLKA = '1' then
    DOA <= RAM(conv_integer(ADDR_A)); -- The read statement must come
                                     -- BEFORE the write statement

    if WEA = '1' then
      RAM(conv_integer(ADDR_A)) := DIA;
    end if;
  end if;
end process;
```

NO_CHANGE の同期のコード例

```
process (CLKA)
begin
  if CLKA'event and CLKA = '1' then
    if WEA = '1' then
      RAM(conv_integer(ADDR_A)) := DIA;
    else
      DOA <= RAM(conv_integer(ADDR_A));
    end if;
  end if;
end process;
```

バイト幅の書き込みイネーブル付きシングルおよびデュアル ポート ブロック RAM

バイト幅の書き込みイネーブルの付いたシングルおよびデュアル ポート ブロック RAM は、VHDL と Verilog の両方でサポートされます。RAM は、同じサイズの列の集まりとして表記されます。書き込みサイクル中は、これらの列ごとへの書き込みを別々に制御します。

- ・ 複数の書き込み文を使用する場合
関連する書き込みイネーブルの記述を含む書き込みアクセス文が列ごとに 1 つずつ記述されます。
- ・ 1 つの書き込み文を使用する場合
記述できる書き込みアクセス文は 1 つのみになります。書き込みイネーブルは、メインの順次プロセスの外側に記述されます。XST では、現在のところ、この方法のみがサポートされています。

列ベースの RAM を記述するこれら 2 つの方法については、次のコード例を参照してください。

複数の書き込み文を使用した VHDL コード例

```

type ram_type is array (SIZE-1 downto 0)
    of std_logic_vector (2*WIDTH-1 downto 0);
signal RAM : ram_type;

(...)

process(clk)
begin
    if posedge(clk) then
        if we(1) = '1' then
            RAM(conv_integer(addr)) (2*WIDTH-1 downto WIDTH) <= di(2*WIDTH-1 downto WIDTH);
        end if;
        if we(0) = '1' then
            RAM(conv_integer(addr)) (WIDTH-1 downto 0) <= di(WIDTH-1 downto 0);
        end if;

        do <= RAM(conv_integer(addr));
    end if;
end process;

```

複数の書き込み文を使用した Verilog コード例

```

reg      [2*DI_WIDTH-1:0] RAM [SIZE-1:0];

always @(posedge clk)
begin
    if (we[1]) then
        RAM[addr][2*WIDTH-1:WIDTH] <= di[2*WIDTH-1:WIDTH];
    end if;
    if (we[0]) then
        RAM[addr][WIDTH-1:0] <= di[WIDTH-1:0];
    end if;

    do <= RAM[addr];
end

```

1 つの書き込み文を使用した VHDL コード例

```

type ram_type is array (SIZE-1 downto 0)
    of std_logic_vector (2*WIDTH-1 downto 0);
signal RAM : ram_type;
signal di0, di1 : std_logic_vector (WIDTH-1 downto 0);

(...)

-- Write enables described outside main sequential process
process(we, di, addr)
begin

    if we(1) = '1' then

```

```

        di1 <= di(2*WIDTH-1 downto WIDTH);
    else
        di1 <= RAM(conv_integer(addr))(2*WIDTH-1 downto WIDTH);
    end if;

    if we(0) = '1' then
        di0 <= di(WIDTH-1 downto 0);
    else
        di0 <= RAM(conv_integer(addr))(WIDTH-1 downto 0);
    end if;

end process;

process(clk)
begin
    if posedge(clk) then
        if en = '1' then
            RAM(conv_integer(addr)) <= di1 & di0; -- single write access statement
            do <= RAM(conv_integer(addr));
        end if;
    end if;
end process;

```

1 つの書き込み文を使用した Verilog コード例

```

reg      [2*DI_WIDTH-1:0] RAM [SIZE-1:0];
reg      [DI_WIDTH-1:0]   di0, di1;

always @(we or di or addr)
begin
    if (we[1])
        di1 = di[2*DI_WIDTH-1:1*DI_WIDTH];
    else
        di1 = RAM[addr][2*DI_WIDTH-1:1*DI_WIDTH];

    if (we[0])
        di0 = di[DI_WIDTH-1:0];
    else
        di0 = RAM[addr][DI_WIDTH-1:0];
    end

always @(posedge clk)
begin
    RAM[addr]<={di1,di0};
    do <= RAM[addr];
end

```

次のコード例では、シングルポートブロック RAM を使用して、バイト幅の書き込みイネーブルのテンプレートを簡単に記述していますが、XST ではデュアルポートブロック RAM もサポートされます。

READ_FIRST モード : バイト幅の書き込みイネーブル (2 バイト) 付きシングルポート BRAM のピンの説明

I/O ピン	説明
clk	クロック (立ち上がりエッジ)
we	書き込みイネーブル
addr	読み出し/書き込みアドレス
di	データ入力
do	RAM 出力ポート

READ_FIRST モード : バイト幅の書き込みイネーブル (2 バイト) 付きシングルポート BRAM の VHDL コード例

```
--
-- Single-Port BRAM with Byte-wide Write Enable (2 bytes) in Read-First Mode
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_24 is
    generic (SIZE      : integer := 512;
            ADDR_WIDTH : integer := 9;
            DI_WIDTH   : integer := 8);

    port (clk : in  std_logic;
          we  : in  std_logic_vector(1 downto 0);
          addr : in  std_logic_vector(ADDR_WIDTH-1 downto 0);
          di   : in  std_logic_vector(2*DI_WIDTH-1 downto 0);
          do   : out std_logic_vector(2*DI_WIDTH-1 downto 0));
end rams_24;

architecture syn of rams_24 is

    type ram_type is array (SIZE-1 downto 0) of std_logic_vector (2*DI_WIDTH-1 downto 0);
    signal RAM : ram_type;

    signal di0, di1 : std_logic_vector (DI_WIDTH-1 downto 0);
begin

    process(we, di)
    begin
        if we(1) = '1' then
            di1 <= di(2*DI_WIDTH-1 downto 1*DI_WIDTH);
        else
            di1 <= RAM(conv_integer(addr))(2*DI_WIDTH-1 downto 1*DI_WIDTH);
        end if;
    end process;
end architecture;
```

```
        if we(0) = '1' then
            di0 <= di(DI_WIDTH-1 downto 0);
        else
            di0 <= RAM(conv_integer(addr))(DI_WIDTH-1 downto 0);
        end if;
    end process;

    process(clk)
    begin
        if (clk'event and clk = '1') then
            RAM(conv_integer(addr)) <= di1 & di0;
            do <= RAM(conv_integer(addr));
        end if;
    end process;

end syn;
```

READ_FIRST モード : バイト幅の書き込みイネーブル (2 バイト) 付きシングル ポート BRAM の Verilog コード例

```
//
// Single-Port BRAM with Byte-wide Write Enable (2 bytes) in Read-First Mode
//

module v_rams_24 (clk, we, addr, di, do);

    parameter SIZE      = 512;
    parameter ADDR_WIDTH = 9;
    parameter DI_WIDTH  = 8;

    input  clk;
    input  [1:0] we;
    input  [ADDR_WIDTH-1:0] addr;
    input  [2*DI_WIDTH-1:0] di;
    output [2*DI_WIDTH-1:0] do;
    reg    [2*DI_WIDTH-1:0] RAM [SIZE-1:0];
    reg    [2*DI_WIDTH-1:0] do;

    reg    [DI_WIDTH-1:0] di0, di1;

    always @(we or di)
    begin
        if (we[1])
            di1 = di[2*DI_WIDTH-1:1*DI_WIDTH];
        else
            di1 = RAM[addr][2*DI_WIDTH-1:1*DI_WIDTH];

        if (we[0])
            di0 = di[DI_WIDTH-1:0];
        else

```

```

        di0 = RAM[addr][DI_WIDTH-1:0];

    end

    always @(posedge clk)
    begin
        RAM[addr]<={di1,di0};
        do <= RAM[addr];
    end

endmodule

```

WRITE_FIRST モード : バイト幅の書き込みイネーブル (2 バイト) 付きシングルポート BRAM のピンの説明

I/O ピン	説明
Clk	クロック (立ち上がりエッジ)
WE	書き込みイネーブル
Addr	読み出し/書き込みアドレス
DI	データ入力
DO	RAM 出力ポート

WRITE_FIRST モード : バイト幅の書き込みイネーブル (2 バイト) 付きシングルポート BRAM の VHDL コード例

```

--
-- Single-Port BRAM with Byte-wide Write Enable (2 bytes) in Write-First Mode
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_25 is
    generic (SIZE          : integer := 512;
            ADDR_WIDTH    : integer := 9;
            DI_WIDTH       : integer := 8);

    port (clk  : in  std_logic;
          we   : in  std_logic_vector(1 downto 0);
          addr : in  std_logic_vector(ADDR_WIDTH-1 downto 0);
          di   : in  std_logic_vector(2*DI_WIDTH-1 downto 0);
          do   : out std_logic_vector(2*DI_WIDTH-1 downto 0));
end rams_25;

architecture syn of rams_25 is
    type ram_type is array (SIZE-1 downto 0) of std_logic_vector (2*DI_WIDTH-1 downto 0);
    signal RAM : ram_type;

```

```

    signal di0, di1 : std_logic_vector (DI_WIDTH-1 downto 0);
    signal do0, do1 : std_logic_vector (DI_WIDTH-1 downto 0);
begin

    process(we, di)
    begin
        if we(1) = '1' then
            di1 <= di(2*DI_WIDTH-1 downto 1*DI_WIDTH);
            do1 <= di(2*DI_WIDTH-1 downto 1*DI_WIDTH);
        else
            di1 <= RAM(conv_integer(addr))(2*DI_WIDTH-1 downto 1*DI_WIDTH);
            do1 <= RAM(conv_integer(addr))(2*DI_WIDTH-1 downto 1*DI_WIDTH);
        end if;

        if we(0) = '1' then
            di0 <= di(DI_WIDTH-1 downto 0);
            do0 <= di(DI_WIDTH-1 downto 0);
        else
            di0 <= RAM(conv_integer(addr))(DI_WIDTH-1 downto 0);
            do0 <= RAM(conv_integer(addr))(DI_WIDTH-1 downto 0);
        end if;
    end process;

    process(clk)
    begin
        if (clk'event and clk = '1') then
            RAM(conv_integer(addr)) <= di1 & di0;
            do <= do1 & do0;
        end if;
    end process;

end syn;

```

WRITE_FIRST モード : バイト幅の書き込みイネーブル (2 バイト) 付きシングル ポート BRAM の Verilog コード例

```

//
// Single-Port BRAM with Byte-wide Write Enable
// (2 bytes) in Write-First Mode
//

module v_rams_25 (clk, we, addr, di, do);

    parameter SIZE          = 512;
    parameter ADDR_WIDTH   = 9;
    parameter DI_WIDTH      = 8;

    input  clk;
    input  [1:0] we;
    input  [ADDR_WIDTH-1:0] addr;

```

```
input  [2*DI_WIDTH-1:0] di;
output [2*DI_WIDTH-1:0] do;
reg    [2*DI_WIDTH-1:0] RAM [SIZE-1:0];
reg    [2*DI_WIDTH-1:0] do;

reg    [DI_WIDTH-1:0]   di0, di1;
reg    [DI_WIDTH-1:0]   do0, do1;

always @(we or di)
begin
    if (we[1])
        begin
            di1 = di[2*DI_WIDTH-1:1*DI_WIDTH];
            do1 = di[2*DI_WIDTH-1:1*DI_WIDTH];
        end
    else
        begin
            di1 = RAM[addr][2*DI_WIDTH-1:1*DI_WIDTH];
            do1 = RAM[addr][2*DI_WIDTH-1:1*DI_WIDTH];
        end

    if (we[0])
        begin
            di0 <= di[DI_WIDTH-1:0];
            do0 <= di[DI_WIDTH-1:0];
        end
    else
        begin
            di0 <= RAM[addr][DI_WIDTH-1:0];
            do0 <= RAM[addr][DI_WIDTH-1:0];
        end
    end

end

always @(posedge clk)
begin
    RAM[addr]<={di1,di0};
    do <= {do1,do0};
end

endmodule
```

NO_CHANGE モード : バイト幅の書き込みイネーブル (2 バイト) 付きシングルポート BRAM のピンの説明

I/O ピン	説明
Clk	クロック (立ち上がりエッジ)
WE	書き込みイネーブル
Addr	読み出し/書き込みアドレス
DI	データ入力
DO	RAM 出力ポート

XST では、基本的な HDL 合成中に DO1 と DO0 信号に対してラッチが推論されます。これらのラッチは、アドバンス HDL 合成段階で BRAM に吸収されます。

NO_CHANGE モード : バイト幅の書き込みイネーブル (2 バイト) 付きシングルポート BRAM の VHDL コード例

```
--
-- Single-Port BRAM with Byte-wide Write Enable (2 bytes) in No-Change Mode
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_26 is
    generic (SIZE      : integer := 512;
            ADDR_WIDTH : integer := 9;
            DI_WIDTH    : integer := 8);

    port (clk : in std_logic;
          we  : in std_logic_vector(1 downto 0);
          addr : in std_logic_vector(ADDR_WIDTH-1 downto 0);
          di   : in std_logic_vector(2*DI_WIDTH-1 downto 0);
          do   : out std_logic_vector(2*DI_WIDTH-1 downto 0));
end rams_26;

architecture syn of rams_26 is
    type ram_type is array (SIZE-1 downto 0) of std_logic_vector (2*DI_WIDTH-1 downto 0);
    signal RAM : ram_type;

    signal di0, di1 : std_logic_vector (DI_WIDTH-1 downto 0);
    signal do0, do1 : std_logic_vector (DI_WIDTH-1 downto 0);
begin

    process(we, di)
    begin
        if we(1) = '1' then
            di1 <= di(2*DI_WIDTH-1 downto 1*DI_WIDTH);
        else

```

```

        di1 <= RAM(conv_integer(addr)) (2*DI_WIDTH-1 downto 1*DI_WIDTH);
        do1 <= RAM(conv_integer(addr)) (2*DI_WIDTH-1 downto 1*DI_WIDTH);
    end if;

    if we(0) = '1' then
        di0 <= di(DI_WIDTH-1 downto 0);
    else
        di0 <= RAM(conv_integer(addr)) (DI_WIDTH-1 downto 0);
        do0 <= RAM(conv_integer(addr)) (DI_WIDTH-1 downto 0);
    end if;
end process;

process(clk)
begin
    if (clk'event and clk = '1') then
        RAM(conv_integer(addr)) <= di1 & di0;
        do <= do1 & do0;
    end if;
end process;

end syn;

```

NO_CHANGE モード : バイト幅の書き込みイネーブル (2 バイト) 付きシングル ポート BRAM の Verilog コード例

```

//
// Single-Port BRAM with Byte-wide Write Enable
// (2 bytes) in No-Change Mode
//

module v_rams_26 (clk, we, addr, di, do);

    parameter SIZE          = 512;
    parameter ADDR_WIDTH   = 9;
    parameter DI_WIDTH      = 8;

    input  clk;
    input  [1:0] we;
    input  [ADDR_WIDTH-1:0] addr;
    input  [2*DI_WIDTH-1:0] di;
    output [2*DI_WIDTH-1:0] do;
    reg    [2*DI_WIDTH-1:0] RAM [SIZE-1:0];
    reg    [2*DI_WIDTH-1:0] do;

    reg    [DI_WIDTH-1:0] di0, di1;
    reg    [DI_WIDTH-1:0] do0, do1;

    always @(we or di)
    begin
        if (we[1])

```

```

        di1 = di[2*DI_WIDTH-1:1*DI_WIDTH];
    else
        begin
            di1 = RAM[addr][2*DI_WIDTH-1:1*DI_WIDTH];
            do1 = RAM[addr][2*DI_WIDTH-1:1*DI_WIDTH];
        end

    if (we[0])
        di0 <= di[DI_WIDTH-1:0];
    else
        begin
            di0 <= RAM[addr][DI_WIDTH-1:0];
            do0 <= RAM[addr][DI_WIDTH-1:0];
        end

    end

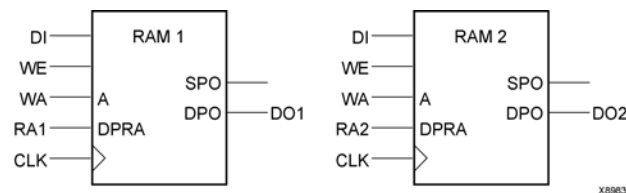
    always @(posedge clk)
    begin
        RAM[addr]<={di1,di0};
        do <= {do1,do0};
    end

endmodule

```

XST では、書き込みアドレスとは異なるアドレスのデータにアクセス可能な読み出しポートが複数ある RAM の記述を認識できます。この場合、使用できる書き込みポートは 1 つのみです。次の記述は、各出力ポートに対する RAM のデータを複製してインプリメントされます(次の図を参照)。

複数ポート RAM の図



複数ポート RAM のピンの説明

I/O ピン	説明
clk	クロック (立ち上がりエッジ)
we	同期書き込みイネーブル (アクティブ High)
wa	書き込みアドレス
ra1	最初の RAM の読み出しアドレス
ra2	2 番目の RAM の読み出しアドレス

I/O ピン	説明
di	データ入力
do1	最初の RAM 出力ポート
do2	2 番目の RAM 出力ポート

複数ポート RAM の VHDL コード例

```
--
-- Multiple-Port RAM Descriptions
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_17 is
    port (clk : in std_logic;
          we  : in std_logic;
          wa  : in std_logic_vector(5 downto 0);
          ra1 : in std_logic_vector(5 downto 0);
          ra2 : in std_logic_vector(5 downto 0);
          di  : in std_logic_vector(15 downto 0);
          do1 : out std_logic_vector(15 downto 0);
          do2 : out std_logic_vector(15 downto 0));
end rams_17;

architecture syn of rams_17 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal RAM : ram_type;
begin

    process (clk)
    begin
        if (clk'event and clk = '1') then
            if (we = '1') then
                RAM(conv_integer(wa)) <= di;
            end if;
        end if;
    end process;

    do1 <= RAM(conv_integer(ra1));
    do2 <= RAM(conv_integer(ra2));

end syn;
```

複数ポート RAM の Verilog コード例

```
//  
// Multiple-Port RAM Descriptions  
//  
  
module v_rams_17 (clk, we, wa, ra1, ra2, di, do1, do2);  
  
    input  clk;  
    input  we;  
    input  [5:0] wa;  
    input  [5:0] ra1;  
    input  [5:0] ra2;  
    input  [15:0] di;  
    output [15:0] do1;  
    output [15:0] do2;  
    reg    [15:0] ram [63:0];  
  
    always @(posedge clk)  
    begin  
        if (we)  
            ram[wa] <= di;  
    end  
  
    assign do1 = ram[ra1];  
    assign do2 = ram[ra2];  
  
endmodule
```

データ出力にリセットの付いたブロック RAM

XST では、Virtex-4、Virtex-5 デバイスおよび関連するブロック RAM リソースで提供されている、データ出力にリセットが付いたブロック RAM がサポートされます。また、RAM データ出力に同期制御の初期化機能を含めることもできます。

リセット可能なデータ ポートは、次の同期モードのブロック RAM に含めることができます。

- ・ リセット付き READ_FIRST モード
- ・ リセット付き WRITE_FIRST モード
- ・ リセット付き NO_CHANGE モード
- ・ レジスタを介したリセット付き ROM
- ・ サポートされているデュアル ポート テンプレート

XST では、デュアル読み出しブロック RAM 記述でデュアル書き込みのブロック RAM がサポートされていないため、両方のデータ出力にリセットを付けることはできませんが、読み出し/書き込みの同期はプライマリ データ出力でのみ可能です。デュアル出力は、READ_FIRST モードのみで使用できます。

リセット付きブロック RAM のピンの説明

I/O ピン	説明
clk	クロック (立ち上がりエッジ)
en	グローバル イネーブル
we	書き込みイネーブル (アクティブ High)
addr	読み出し/書き込みアドレス
rst	データ出力のリセット
di	データ入力
do	RAM 出力ポート

リセット付きブロック RAM の VHDL コード例

```
--
-- Block RAM with Reset
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_18 is
    port (clk : in std_logic;
          en  : in std_logic;
          we  : in std_logic;
          rst : in std_logic;
          addr : in std_logic_vector(5 downto 0);
          di   : in std_logic_vector(15 downto 0);
          do   : out std_logic_vector(15 downto 0));
end rams_18;

architecture syn of rams_18 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal ram : ram_type;
begin

    process (clk)
    begin
        if clk'event and clk = '1' then
            if en = '1' then -- optional enable

                if we = '1' then -- write enable
                    ram(conv_integer(addr)) <= di;
                end if;

                if rst = '1' then -- optional reset
                    do <= (others => '0');
                else

```

```
        do <= ram(conv_integer(addr)) ;
    end if;

    end if;
end if;
end process;

end syn;
```

リセット付きブロック RAM の Verilog コード例

```
//
// Block RAM with Reset
//

module v_rams_18 (clk, en, we, rst, addr, di, do);

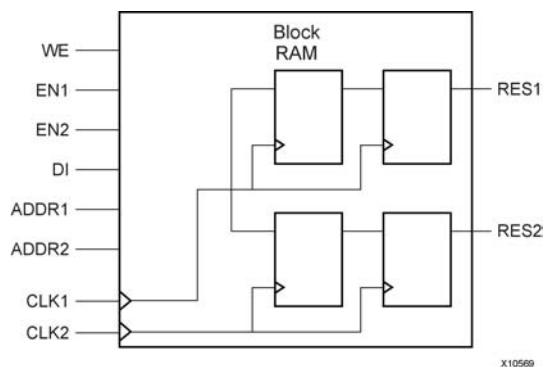
    input  clk;
    input  en;
    input  we;
    input  rst;
    input  [5:0] addr;
    input  [15:0] di;
    output [15:0] do;
    reg    [15:0] ram [63:0];
    reg    [15:0] do;

    always @(posedge clk)
    begin
        if (en) // optional enable
        begin
            if (we) // write enable
                ram[addr] <= di;

            if (rst) // optional reset
                do <= 16'h0000;
            else
                do <= ram[addr];
            end
        end
    end

endmodule
```

オプションの出力レジスタ付きブロック RAM の図



オプションの出力レジスタ付きブロック RAM のピンの説明

I/O ピン	説明
clk1, clk2	クロック (立ち上がりエッジ)
we	書き込みイネーブル
EN1、EN2	クロック イネーブル (アクティブ High)
addr1	プライマリ読み出しアドレス
addr2	デュアル読み出しアドレス
di	データ入力
res1	プライマリ出力ポート
res2	デュアル出力ポート

オプションの出力レジスタ付き ブロック RAM の VHDL コード例

```
--
-- Block RAM with Optional Output Registers
--

library IEEE;
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity rams_19 is
    port (clk1, clk2    : in std_logic;
          we, en1, en2 : in std_logic;
          addr1         : in std_logic_vector(5 downto 0);
          addr2         : in std_logic_vector(5 downto 0);
          di            : in std_logic_vector(15 downto 0);
          res1          : out std_logic_vector(15 downto 0);
          res2          : out std_logic_vector(15 downto 0));
end rams_19;

architecture beh of rams_19 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
```

```
signal ram : ram_type;
signal do1 : std_logic_vector(15 downto 0);
signal do2 : std_logic_vector(15 downto 0);
begin

process (clk1)
begin
    if rising_edge(clk1) then
        if we = '1' then
            ram(conv_integer(addr1)) <= di;
        end if;
        do1 <= ram(conv_integer(addr1));
    end if;
end process;

process (clk2)
begin
    if rising_edge(clk2) then
        do2 <= ram(conv_integer(addr2));
    end if;
end process;

process (clk1)
begin
    if rising_edge(clk1) then
        if en1 = '1' then
            res1 <= do1;
        end if;
    end if;
end process;

process (clk2)
begin
    if rising_edge(clk2) then
        if en2 = '1' then
            res2 <= do2;
        end if;
    end if;
end process;

end beh;
```

オプションの出力レジスタ付き ブロック RAM の Verilog コード例

```
//
// Block RAM with Optional Output Registers
//

module v_rams_19 (clk1, clk2, we, en1, en2, addr1, addr2, di, res1, res2);

    input  clk1;
    input  clk2;
    input  we, en1, en2;
    input  [5:0] addr1;
    input  [5:0] addr2;
    input  [15:0] di;
    output [15:0] res1;
    output [15:0] res2;
    reg    [15:0] res1;
    reg    [15:0] res2;
    reg    [15:0] RAM [63:0];
    reg    [15:0] do1;
    reg    [15:0] do2;

    always @(posedge clk1)
    begin
        if (we == 1'b1)
            RAM[addr1] <= di;
        do1 <= RAM[addr1];
    end

    always @(posedge clk2)
    begin
        do2 <= RAM[addr2];
    end

    always @(posedge clk1)
    begin
        if (en1 == 1'b1)
            res1 <= do1;
    end

    always @(posedge clk2)
    begin
        if (en2 == 1'b1)
            res2 <= do2;
    end

endmodule
```

RAM の初期化のコード例

コード例は、ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

ブロック RAM および分散 RAM の初期内容は、HDL コードでメモリ アレイを記述した信号を初期化すると指定できます。初期化は、VHDL コードで直接することもできますが、初期化データが含まれるファイルを指定することでもできます。

RAM の初期化は、VHDL および Verilog の両方でサポートされます。

この例では、RAM を Hardware Description Language (HDL) コードで直接初期化する例を示しています。

RAM の初期内容の VHDL コード例 (16 進数)

RAM の初期内容は、次の例のように VHDL コードでメモリ アレイを記述する信号を初期化することで指定します。

```
...
type ram_type is array (0 to 63) of std_logic_vector(19 downto 0);
signal RAM : ram_type :=
(
  X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A",
  X"00300", X"08602", X"02310", X"0203B", X"08300", X"04002",
  X"08201", X"00500", X"04001", X"02500", X"00340", X"00241",
  X"04002", X"08300", X"08201", X"00500", X"08101", X"00602",
  X"04003", X"0241E", X"00301", X"00102", X"02122", X"02021",
  X"00301", X"00102", X"02222", X"04001", X"00342", X"0232B",
  X"00900", X"00302", X"00102", X"04002", X"00900", X"08201",
  X"02023", X"00303", X"02433", X"00301", X"04004", X"00301",
  X"00102", X"02137", X"02036", X"00301", X"00102", X"02237",
  X"04004", X"00304", X"04040", X"02500", X"02500", X"02500",
  X"0030D", X"02341", X"08201", X"0400D");
...
process (clk)
begin
  if rising_edge(clk) then
    if we = '1' then
      RAM(conv_integer(a)) <= di;
    end if;
    ra <= a;
  end if;
end process;
...
do <= RAM(conv_integer(ra));
```

ブロック RAM を初期化する Verilog コード例 (16 進数)

RAM の初期内容は、次の例のように initial 文を使用して Verilog コードでメモリ アレイを記述する信号を初期化することで指定します。

```
...
reg [19:0] ram [63:0];
initial begin
    ram[63] = 20'h0200A; ram[62] = 20'h00300; ram[61] = 20'h08101;
    ram[60] = 20'h04000; ram[59] = 20'h08601; ram[58] = 20'h0233A;
    ...
    ram[2] = 20'h02341; ram[1] = 20'h08201; ram[0] = 20'h0400D;
end
...
always @(posedge clk)
begin
    if (we)
        ram[addr] <= di;
    do <= ram[addr];
end
```

RAM の初期内容の VHDL コード例 (2 進数)

RAM の初期値は、「RAM の初期内容の VHDL コード例 (16 進数)」のように 16 進数で指定するか、次の例のように 2 進数で指定できます。

```
...
type ram_type is array (0 to SIZE-1) of std_logic_vector(15 downto 0);
signal RAM : ram_type :=
(
    "01111001000000101",
    "0000010110111101",
    "1100001101010000",
    ...
    "0000100101110011");
```

ブロック RAM を初期化する Verilog コード例 (2 進数)

RAM の初期値は、「ブロック RAM を初期化する Verilog コード例 (16 進数)」のように 16 進数で指定するか、次の例のように 2 進数で指定できます。

```
...
reg [15:0] ram [63:0];
initial begin
    ram[63] = 16'b01111001000000101;
    ram[62] = 16'b0000010110111101;
    ram[61] = 16'b1100001101010000;
    ...
    ram[0] = 16'b0000100101110011;
end
...
```

シングルポートブロック RAM の初期内容を記述した VHDL コード例

```
--
-- Initializing Block RAM (Single-Port BRAM)
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_20a is
    port (clk : in std_logic;
          we : in std_logic;
          addr : in std_logic_vector(5 downto 0);
          di : in std_logic_vector(19 downto 0);
          do : out std_logic_vector(19 downto 0));
end rams_20a;

architecture syn of rams_20a is

    type ram_type is array (63 downto 0) of std_logic_vector (19 downto 0);
    signal RAM : ram_type:= (X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A",
                             X"00300", X"08602", X"02310", X"0203B", X"08300", X"04002",
                             X"08201", X"00500", X"04001", X"02500", X"00340", X"00241",
                             X"04002", X"08300", X"08201", X"00500", X"08101", X"00602",
                             X"04003", X"0241E", X"00301", X"00102", X"02122", X"02021",
                             X"00301", X"00102", X"02222", X"04001", X"00342", X"0232B",
                             X"00900", X"00302", X"00102", X"04002", X"00900", X"08201",
                             X"02023", X"00303", X"02433", X"00301", X"04004", X"00301",
                             X"00102", X"02137", X"02036", X"00301", X"00102", X"02237",
                             X"04004", X"00304", X"04040", X"02500", X"02500", X"02500",
                             X"0030D", X"02341", X"08201", X"0400D");

begin

    process (clk)
    begin
        if rising_edge(clk) then
            if we = '1' then
                RAM(conv_integer(addr)) <= di;
            end if;
            do <= RAM(conv_integer(addr));
        end if;
    end process;

end syn;
```

シングル ポート ブロック RAM の初期内容を記述した Verilog コード例

```
//
// Initializing Block RAM (Single-Port BRAM)
//
module v_rams_20a (clk, we, addr, di, do);
    input clk;
    input we;
    input [5:0] addr;
    input [19:0] di;
    output [19:0] do;
    reg [19:0] ram [63:0];
    reg [19:0] do;

    initial begin
        ram[63] = 20'h0200A; ram[62] = 20'h00300; ram[61] = 20'h08101;
        ram[60] = 20'h04000; ram[59] = 20'h08601; ram[58] = 20'h0233A;
        ram[57] = 20'h00300; ram[56] = 20'h08602; ram[55] = 20'h02310;
        ram[54] = 20'h0203B; ram[53] = 20'h08300; ram[52] = 20'h04002;
        ram[51] = 20'h08201; ram[50] = 20'h00500; ram[49] = 20'h04001;
        ram[48] = 20'h02500; ram[47] = 20'h00340; ram[46] = 20'h00241;
        ram[45] = 20'h04002; ram[44] = 20'h08300; ram[43] = 20'h08201;
        ram[42] = 20'h00500; ram[41] = 20'h08101; ram[40] = 20'h00602;
        ram[39] = 20'h04003; ram[38] = 20'h0241E; ram[37] = 20'h00301;
        ram[36] = 20'h00102; ram[35] = 20'h02122; ram[34] = 20'h02021;
        ram[33] = 20'h00301; ram[32] = 20'h00102; ram[31] = 20'h02222;

        ram[30] = 20'h04001; ram[29] = 20'h00342; ram[28] = 20'h0232B;
        ram[27] = 20'h00900; ram[26] = 20'h00302; ram[25] = 20'h00102;
        ram[24] = 20'h04002; ram[23] = 20'h00900; ram[22] = 20'h08201;
        ram[21] = 20'h02023; ram[20] = 20'h00303; ram[19] = 20'h02433;
        ram[18] = 20'h00301; ram[17] = 20'h04004; ram[16] = 20'h00301;
        ram[15] = 20'h00102; ram[14] = 20'h02137; ram[13] = 20'h02036;
        ram[12] = 20'h00301; ram[11] = 20'h00102; ram[10] = 20'h02237;
        ram[9] = 20'h04004; ram[8] = 20'h00304; ram[7] = 20'h04040;
        ram[6] = 20'h02500; ram[5] = 20'h02500; ram[4] = 20'h02500;
        ram[3] = 20'h0030D; ram[2] = 20'h02341; ram[1] = 20'h08201;
        ram[0] = 20'h0400D;
    end

    always @(posedge clk)
    begin
        if (we)
            ram[addr] <= di;
        do <= ram[addr];
    end
end

endmodule
```

デュアル ポート ブロック RAM の初期内容を記述した VHDL コード例

```
--
-- Initializing Block RAM (Dual-Port BRAM)
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_20b is
  port (clk1 : in std_logic;
        clk2 : in std_logic;
        we : in std_logic;
        addr1 : in std_logic_vector(7 downto 0);
        addr2 : in std_logic_vector(7 downto 0);
        di : in std_logic_vector(15 downto 0);
        do1 : out std_logic_vector(15 downto 0);
        do2 : out std_logic_vector(15 downto 0));
end rams_20b;

architecture syn of rams_20b is

  type ram_type is array (255 downto 0) of std_logic_vector (15 downto 0);
  signal RAM : ram_type:= (255 downto 100 => X"B8B8", 99 downto 0 => X"8282");

begin

  process (clk1)
  begin
    if rising_edge(clk1) then
      if we = '1' then
        RAM(conv_integer(addr1)) <= di;
      end if;
      do1 <= RAM(conv_integer(addr1));
    end if;
  end process;

  process (clk2)
  begin
    if rising_edge(clk2) then
      do2 <= RAM(conv_integer(addr2));
    end if;
  end process;

end syn;
```

デュアル ポート ブロック RAM の初期内容を記述した Verilog コード例

```
//
// Initializing Block RAM (Dual-Port BRAM)
//

module v_rams_20b (clk1, clk2, we, addr1, addr2, di, do1, do2);
    input clk1, clk2;
    input we;
    input [7:0] addr1, addr2;
    input [15:0] di;
    output [15:0] do1, do2;

    reg [15:0] ram [255:0];
    reg [15:0] do1, do2;
    integer index;

    initial begin
        for (index = 0 ; index <= 99 ; index = index + 1) begin
            ram[index] = 16'h8282;
        end

        for (index= 100 ; index <= 255 ; index = index + 1) begin
            ram[index] = 16'hB8B8;
        end
    end

    always @(posedge clk1)
    begin
        if (we)
            ram[addr1] <= di;
        do1 <= ram[addr1];
    end

    always @(posedge clk2)
    begin
        do2 <= ram[addr2];
    end
endmodule
```

外部ファイルからの RAM の初期化のコード例

この例では、外部データファイルを使用してブロック RAM を初期化しています。

外部ファイルに含まれる値で RAM を初期化するには、VHDL コードで read 関数を使用します。

詳細は、次を参照してください。

[VHDL のファイル タイプ サポート](#)

次の手順に従って、初期化ファイルを設定します。

- ・ RAM の指定した行の初期内容を表すには、初期化ファイルの各ラインを使用します。
- ・ RAM の内容は 16 進数または 2 進数で表現します。
- ・ RAM のアレイの行数と初期化ファイルの行数を同数にします。

次は、8 X 32 ビット RAM を 2 進数値で初期化するファイルの内容の例です。

```
00001111000011110000111100001111
01001010001000001100000010000100
000000000011111000000000001000001
11111101010000011100010000100100
00001111000011110000111100001111
01001010001000001100000010000100
000000000011111000000000001000001
11111101010000011100010000100100
```

ブロック RAM の初期化 (外部データ ファイル)

RAM の初期値は、外部データ ファイルに保存し、HDL コード内で指定して読み込むことができます。データ ファイルは、コメントまたはその他の情報が何もない状態の 2 進数または 16 進数で記述されます。

次は、8 X 32 ビット RAM を 2 進数値で初期化するファイルの内容の例です。どちらの例も、参照されたデータ ファイルは rams_20c.data です。

```
00001111000011110000111100001111
01001010001000001100000010000100
000000000011111000000000001000001
11111101010000011100010000100100
00001111000011110000111100001111
01001010001000001100000010000100
000000000011111000000000001000001
11111101010000011100010000100100
```

ブロック RAM の初期化 (外部データ ファイル) の VHDL コード例

次の例では、初期値を生成するループが RAM のアドレス範囲内にあることを確認することで制御されます。この例では、外部データ ファイルを使用してブロック RAM を初期化しています。

```
--
-- Initializing Block RAM from external data file
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
```

```
use std.textio.all;

entity rams_20c is
  port(clk : in std_logic;
        we : in std_logic;
        addr : in std_logic_vector(5 downto 0);
        din : in std_logic_vector(31 downto 0);
        dout : out std_logic_vector(31 downto 0));
end rams_20c;

architecture syn of rams_20c is

  type RamType is array(0 to 63) of bit_vector(31 downto 0);

  impure function InitRamFromFile (RamFileName : in string) return RamType is
    FILE RamFile : text is in RamFileName;
    variable RamFileLine : line;
    variable RAM : RamType;
  begin
    for I in RamType'range loop
      readline (RamFile, RamFileLine);
      read (RamFileLine, RAM(I));
    end loop;
    return RAM;
  end function;

  signal RAM : RamType := InitRamFromFile("rams_20c.data");

begin

  process (clk)
  begin
    if clk'event and clk = '1' then
      if we = '1' then
        RAM(conv_integer(addr)) <= to_bitvector(din);
      end if;
      dout <= to_stdlogicvector(RAM(conv_integer(addr)));
    end if;
  end process;

end syn;
```

外部データファイルに含まれる行数が不十分の場合、次のエラー メッセージが表示されます。
ERROR:Xst - raminitfile1.vhd line 40: Line <RamFileLine has not enough elements for target <RAM<63>>.

ブロック RAM の初期化 (外部データ ファイル) の Verilog コード例

外部ファイルに含まれる値で RAM を初期化するには、Verilog コードで \$readmemb または \$readmemh システム タスクを使用します。

詳細は、次を参照してください。

Verilog ビヘイビア記述のサポート

次の手順に従って、初期化ファイルを設定します。

- ・ 初期化ファイルの各行が RAM の該当する行の初期内容を記述するようにします。
- ・ RAM の内容は 16 進数または 2 進数で表現します。
- ・ 2 進数の場合は \$readmemb、16 進数の場合は \$readmemh を使用します。XST とシミュレータで処理の違いが発生しないようにするため、これらのシステム タスクでは次のようにインデックス パラメータを使用することをお勧めします。次の例を参照してください。

```
$readmemb("rams_20c.data", ram, 0, 7);
```

RAM のアレイの行数と初期化ファイルの行数を同じにする必要があります。

```
//  
// Initializing Block RAM from external data file  
//  
  
module v_rams_20c (clk, we, addr, din, dout);  
    input clk;  
    input we;  
    input [5:0] addr;  
    input [31:0] din;  
    output [31:0] dout;  
  
    reg [31:0] ram [0:63];  
    reg [31:0] dout;  
  
    initial  
    begin  
        $readmemb("rams_20c.data", ram, 0, 63);  
    end  
  
    always @(posedge clk)  
    begin  
        if (we)  
            ram[addr] <= din;  
            dout <= ram[addr];  
        end  
  
endmodule
```

外部ファイルからの RAM の初期化のコード例

この例では、外部データファイルを使用してブロック RAM を初期化しています。

外部ファイルに含まれる値で RAM を初期化するには、VHDL コードで read 関数を使用します。

詳細は、次を参照してください。

VHDL のファイル タイプ サポート

次の手順に従って、初期化ファイルを設定します。

- ・ RAM の指定した行の初期内容を表すには、初期化ファイルの各ラインを使用します。
- ・ RAM の内容は 16 進数または 2 進数で表現します。
- ・ RAM のアレイの行数と初期化ファイルの行数を同数にします。

次は、8 X 32 ビット RAM を 2 進数値で初期化するファイルの内容の例です。

```
00001111000011110000111100001111
01001010001000001100000010000100
000000000011111000000000001000001
11111101010000011100010000100100
00001111000011110000111100001111
01001010001000001100000010000100
000000000011111000000000001000001
11111101010000011100010000100100
```

ブロック RAM の初期化 (外部データ ファイル)

RAM の初期値は、外部データ ファイルに保存し、HDL コード内で指定して読み込むことができます。データ ファイルは、コメントまたはその他の情報が何もない状態の 2 進数または 16 進数で記述されます。

次は、8 X 32 ビット RAM を 2 進数値で初期化するファイルの内容の例です。どちらの例も、参照されたデータ ファイルは rams_20c.data です。

```
00001111000011110000111100001111
01001010001000001100000010000100
000000000011111000000000001000001
11111101010000011100010000100100
00001111000011110000111100001111
01001010001000001100000010000100
000000000011111000000000001000001
11111101010000011100010000100100
```

ブロック RAM の初期化 (外部データ ファイル) の VHDL コード例

次の例では、初期値を生成するループが RAM のアドレス範囲内にあることを確認することで制御されます。この例では、外部データ ファイルを使用してブロック RAM を初期化しています。

```
--
-- Initializing Block RAM from external data file
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use std.textio.all;

entity rams_20c is
  port (clk : in std_logic;
        we : in std_logic;
        addr : in std_logic_vector(5 downto 0);
        din : in std_logic_vector(31 downto 0);
        dout : out std_logic_vector(31 downto 0));
end rams_20c;

architecture syn of rams_20c is

  type RamType is array(0 to 63) of bit_vector(31 downto 0);

  impure function InitRamFromFile (RamFileName : in string) return RamType is
    FILE RamFile : text is in RamFileName;
    variable RamFileLine : line;
    variable RAM : RamType;
  begin
    for I in RamType'range loop
      readline (RamFile, RamFileLine);
      read (RamFileLine, RAM(I));
    end loop;
    return RAM;
  end function;

  signal RAM : RamType := InitRamFromFile("rams_20c.data");

begin

  process (clk)
  begin
    if clk'event and clk = '1' then
      if we = '1' then
        RAM(conv_integer(addr)) <= to_bitvector(din);
      end if;
      dout <= to_stdlogicvector(RAM(conv_integer(addr)));
    end if;
  end process;
end architecture;
```

```
end process;
```

```
end syn;
```

外部データファイルに含まれる行数が不十分の場合、次のエラー メッセージが表示されます。
ERROR:Xst - raminitfile1.vhd line 40: Line <RamFileLine has not enough elements for target <RAM<63>>.

ブロック RAM の初期化 (外部データ ファイル) の Verilog コード例

外部ファイルに含まれる値で RAM を初期化するには、Verilog コードで \$readmemb または \$readmemh システム タスクを使用します。

詳細は、次を参照してください。

Verilog ビヘイビア記述のサポート

次の手順に従って、初期化ファイルを設定します。

- ・ 初期化ファイルの各行が RAM の該当する行の初期内容を記述するようにします。
- ・ RAM の内容は 16 進数または 2 進数で表現します。
- ・ 2 進数の場合は \$readmemb、16 進数の場合は \$readmemh を使用します。XST とシミュレータで処理の違いが発生しないようにするため、これらのシステム タスクでは次のようにインデックス パラメータを使用することをお勧めします。次の例を参照してください。

```
$readmemb("rams_20c.data", ram, 0, 7);
```

RAM のアレイの行数と初期化ファイルの行数を同じにする必要があります。

```
//  
// Initializing Block RAM from external data file  
//  
  
module v_rams_20c (clk, we, addr, din, dout);  
    input clk;  
    input we;  
    input [5:0] addr;  
    input [31:0] din;  
    output [31:0] dout;  
  
    reg [31:0] ram [0:63];  
    reg [31:0] dout;  
  
    initial  
    begin  
        $readmemb("rams_20c.data", ram, 0, 63);  
    end  
  
    always @(posedge clk)  
    begin  
        if (we)  
            ram[addr] <= din;  
            dout <= ram[addr];  
    end  
  
endmodule
```

ブロック RAM リソースを使用した ROM の HDL コーディング手法

このセクションには、次の内容が含まれます。

- ・ [ブロック RAM リソースを使用した ROM の概要](#)
- ・ [ブロック RAM リソースを使用した ROM のログ ファイル](#)
- ・ [ブロック RAM リソースを使用した ROM 関連の制約](#)
- ・ [ブロック RAM リソースを使用した ROM のコード例](#)

ブロック RAM リソースを使用した ROM の概要

XST では、ブロック RAM リソースを使用して同期出力またはアドレス入力を持つ ROM をインプリメントできます。こういった ROM は、HDL 記述によって、シングル ポートまたはデュアルポートのブロック RAM としてインプリメントされます。

[階層の維持 \(KEEP_HIERARCHY\)](#) を no に設定すると、階層が違っていてもブロック ROM が推論されます。この場合、ROM とデータ出力、またはアドレス レジスタは別の階層ブロックに記述できます。これは、アドバンス HDL 合成段階で推論されます。

ブロック RAM リソースを使用して ROM をインプリメントする場合は、[ROM スタイル \(ROM_STYLE\)](#) 制約を使用して制御します。

[ROM スタイル \(ROM_STYLE\)](#) の詳細は、次を参照してください。

[デザイン制約](#)

ROM インプリメンテーションの詳細は、次を参照してください。

[FPGA の最適化](#)

ブロック RAM リソースを使用した ROM のログ ファイル

次は、ブロック RAM リソースを使用した ROM のログ ファイルの例です。

ブロック RAM リソースを使用した ROM のログ ファイルの例

```
=====
*                               HDL Synthesis                               *
=====

Synthesizing Unit <rams_21a>.
  Related source file is "rams_21a.vhd".
  Found 64x20-bit ROM for signal <$varindex0000> created at line 38.
  Found 20-bit register for signal <data>.
  Summary:
    inferred   1 ROM(s).
    inferred  20 D-type flip-flop(s).
Unit <rams_21a> synthesized.
=====

HDL Synthesis Report
Macro Statistics
# ROMs                               : 1
  64x20-bit ROM                       : 1
# Registers                           : 1
  20-bit register                      : 1
```

```

=====
*                               Advanced HDL Synthesis                               *
=====
INFO:Xst - Unit <rams_21a> : The ROM <Mrom__varindex0000> will be implemented
as a read-only BLOCK RAM, absorbing the register: <data>.

-----
| ram_type          | Block          |          |
-----
| Port A           |               |          |
|   aspect ratio   | 64-word x 20-bit (6.9%) |          |
|   mode           | write-first    |          |
|   clkA           | connected to signal <clk> | rise     |
|   enA            | connected to signal <en> | high     |
|   weA            | connected to internal node | high     |
|   addrA          | connected to signal <addr> |          |
|   diA            | connected to internal node |          |
|   doA            | connected to signal <data> |          |
-----

=====
Advanced HDL Synthesis Report
Macro Statistics
# RAMs                                     : 1
64x20-bit single-port block RAM           : 1
=====

```

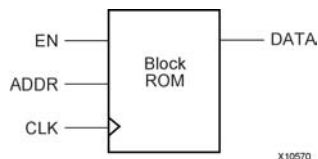
ブロック RAM リソースを使用した ROM 関連の制約

ROM スタイル (ROM_STYLE)

ブロック RAM リソースを使用した ROM のコード例

コード例は、ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

レジスタ付き出力を持つ ROM の図



レジスタ付き出力を持つ ROM のピンの説明

I/O ピン	説明
clk	クロック (立ち上がりエッジ)
en	同期イネーブル (アクティブ High)
addr	読み出しアドレス
data	データ出力

レジスタ付き出力を持つ ROM の VHDL コード例 1

```
--
-- ROMs Using Block RAM Resources.
-- VHDL code for a ROM with registered output (template 1)
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_21a is
    port (clk : in std_logic;
          en  : in std_logic;
          addr : in std_logic_vector(5 downto 0);
          data : out std_logic_vector(19 downto 0));
end rams_21a;

architecture syn of rams_21a is

    type rom_type is array (63 downto 0) of std_logic_vector (19 downto 0);
    signal ROM : rom_type:= (X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A",
                             X"00300", X"08602", X"02310", X"0203B", X"08300", X"04002",
                             X"08201", X"00500", X"04001", X"02500", X"00340", X"00241",
                             X"04002", X"08300", X"08201", X"00500", X"08101", X"00602",
                             X"04003", X"0241E", X"00301", X"00102", X"02122", X"02021",
                             X"00301", X"00102", X"02222", X"04001", X"00342", X"0232B",
                             X"00900", X"00302", X"00102", X"04002", X"00900", X"08201",
                             X"02023", X"00303", X"02433", X"00301", X"04004", X"00301",
                             X"00102", X"02137", X"02036", X"00301", X"00102", X"02237",
                             X"04004", X"00304", X"04040", X"02500", X"02500", X"02500",
                             X"0030D", X"02341", X"08201", X"0400D");

begin

    process (clk)
    begin
        if (clk'event and clk = '1') then
            if (en = '1') then
                data <= ROM(conv_integer(addr));
            end if;
        end if;
    end process;

end syn;
```

レジスタ付き出力を持つ ROM の VHDL コード例 2

```
--
-- ROMs Using Block RAM Resources.
-- VHDL code for a ROM with registered output (template 2)
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_21b is
port (clk : in std_logic;
      en : in std_logic;
      addr : in std_logic_vector(5 downto 0);
      data : out std_logic_vector(19 downto 0));
end rams_21b;

architecture syn of rams_21b is
    type rom_type is array (63 downto 0) of std_logic_vector (19 downto 0);
    signal ROM : rom_type:= (X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A",
                             X"00300", X"08602", X"02310", X"0203B", X"08300", X"04002",
                             X"08201", X"00500", X"04001", X"02500", X"00340", X"00241",
                             X"04002", X"08300", X"08201", X"00500", X"08101", X"00602",
                             X"04003", X"0241E", X"00301", X"00102", X"02122", X"02021",
                             X"00301", X"00102", X"02222", X"04001", X"00342", X"0232B",
                             X"00900", X"00302", X"00102", X"04002", X"00900", X"08201",
                             X"02023", X"00303", X"02433", X"00301", X"04004", X"00301",
                             X"00102", X"02137", X"02036", X"00301", X"00102", X"02237",
                             X"04004", X"00304", X"04040", X"02500", X"02500", X"02500",
                             X"0030D", X"02341", X"08201", X"0400D");

    signal rdata : std_logic_vector(19 downto 0);
begin

    rdata <= ROM(conv_integer(addr));

    process (clk)
    begin
        if (clk'event and clk = '1') then
            if (en = '1') then
                data <= rdata;
            end if;
        end if;
    end process;

end syn;
```

レジスタ付き出力を持つ ROM の Verilog コード例 1

```
//
// ROMs Using Block RAM Resources.
// Verilog code for a ROM with registered output (template 1)
//

module v_rams_21a (clk, en, addr, data);

    input    clk;
    input    en;
    input    [5:0] addr;
    output reg [19:0] data;

    always @(posedge clk) begin
        if (en)
            case(addr)
                6'b000000: data <= 20'h0200A;    6'b100000: data <= 20'h02222;
                6'b000001: data <= 20'h00300;    6'b100001: data <= 20'h04001;
                6'b000010: data <= 20'h08101;    6'b100010: data <= 20'h00342;
                6'b000011: data <= 20'h04000;    6'b100011: data <= 20'h0232B;
                6'b000100: data <= 20'h08601;    6'b100100: data <= 20'h00900;
                6'b000101: data <= 20'h0233A;    6'b100101: data <= 20'h00302;
                6'b000110: data <= 20'h00300;    6'b100110: data <= 20'h00102;
                6'b000111: data <= 20'h08602;    6'b100111: data <= 20'h04002;
                6'b001000: data <= 20'h02310;    6'b101000: data <= 20'h00900;
                6'b001001: data <= 20'h0203B;    6'b101001: data <= 20'h08201;
                6'b001010: data <= 20'h08300;    6'b101010: data <= 20'h02023;
                6'b001011: data <= 20'h04002;    6'b101011: data <= 20'h00303;
                6'b001100: data <= 20'h08201;    6'b101100: data <= 20'h02433;
                6'b001101: data <= 20'h00500;    6'b101101: data <= 20'h00301;
                6'b001110: data <= 20'h04001;    6'b101110: data <= 20'h04004;
                6'b001111: data <= 20'h02500;    6'b101111: data <= 20'h00301;
                6'b010000: data <= 20'h00340;    6'b110000: data <= 20'h00102;
                6'b010001: data <= 20'h00241;    6'b110001: data <= 20'h02137;
                6'b010010: data <= 20'h04002;    6'b110010: data <= 20'h02036;
                6'b010011: data <= 20'h08300;    6'b110011: data <= 20'h00301;
                6'b010100: data <= 20'h08201;    6'b110100: data <= 20'h00102;
                6'b010101: data <= 20'h00500;    6'b110101: data <= 20'h02237;
                6'b010110: data <= 20'h08101;    6'b110110: data <= 20'h04004;
                6'b010111: data <= 20'h00602;    6'b110111: data <= 20'h00304;
                6'b011000: data <= 20'h04003;    6'b111000: data <= 20'h04040;
                6'b011001: data <= 20'h0241E;    6'b111001: data <= 20'h02500;
                6'b011010: data <= 20'h00301;    6'b111010: data <= 20'h02500;
                6'b011011: data <= 20'h00102;    6'b111011: data <= 20'h02500;
                6'b011100: data <= 20'h02122;    6'b111100: data <= 20'h0030D;
                6'b011101: data <= 20'h02021;    6'b111101: data <= 20'h02341;
                6'b011110: data <= 20'h00301;    6'b111110: data <= 20'h08201;
                6'b011111: data <= 20'h00102;    6'b111111: data <= 20'h0400D;
            endcase
        end
    end
endmodule
```

レジスタ付き出力を持つ ROM の Verilog コード例 2

```
//
// ROMs Using Block RAM Resources.
// Verilog code for a ROM with registered output (template 2)
//

module v_rams_21b (clk, en, addr, data);

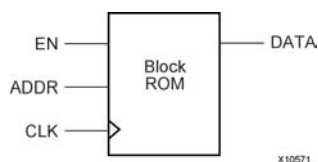
    input    clk;
    input    en;
    input    [5:0] addr;
    output reg [19:0] data;
```

```
reg          [19:0] rdata;

always @(addr) begin
    case (addr)
        6'b000000: rdata <= 20'h0200A;  6'b100000: rdata <= 20'h02222;
        6'b000001: rdata <= 20'h00300;  6'b100001: rdata <= 20'h04001;
        6'b000010: rdata <= 20'h08101;  6'b100010: rdata <= 20'h00342;
        6'b000011: rdata <= 20'h04000;  6'b100011: rdata <= 20'h0232B;
        6'b000100: rdata <= 20'h08601;  6'b100100: rdata <= 20'h00900;
        6'b000101: rdata <= 20'h0233A;  6'b100101: rdata <= 20'h00302;
        6'b000110: rdata <= 20'h00300;  6'b100110: rdata <= 20'h00102;
        6'b000111: rdata <= 20'h08602;  6'b100111: rdata <= 20'h04002;
        6'b001000: rdata <= 20'h02310;  6'b101000: rdata <= 20'h00900;
        6'b001001: rdata <= 20'h0203B;  6'b101001: rdata <= 20'h08201;
        6'b001010: rdata <= 20'h08300;  6'b101010: rdata <= 20'h02023;
        6'b001011: rdata <= 20'h04002;  6'b101011: rdata <= 20'h00303;
        6'b001100: rdata <= 20'h08201;  6'b101100: rdata <= 20'h02433;
        6'b001101: rdata <= 20'h00500;  6'b101101: rdata <= 20'h00301;
        6'b001110: rdata <= 20'h04001;  6'b101110: rdata <= 20'h04004;
        6'b001111: rdata <= 20'h02500;  6'b101111: rdata <= 20'h00301;
        6'b010000: rdata <= 20'h00340;  6'b110000: rdata <= 20'h00102;
        6'b010001: rdata <= 20'h00241;  6'b110001: rdata <= 20'h02137;
        6'b010010: rdata <= 20'h04002;  6'b110010: rdata <= 20'h02036;
        6'b010011: rdata <= 20'h08300;  6'b110011: rdata <= 20'h00301;
        6'b010100: rdata <= 20'h08201;  6'b110100: rdata <= 20'h00102;
        6'b010101: rdata <= 20'h00500;  6'b110101: rdata <= 20'h02237;
        6'b010110: rdata <= 20'h08101;  6'b110110: rdata <= 20'h04004;
        6'b010111: rdata <= 20'h00602;  6'b110111: rdata <= 20'h00304;
        6'b011000: rdata <= 20'h04003;  6'b111000: rdata <= 20'h04040;
        6'b011001: rdata <= 20'h0241E;  6'b111001: rdata <= 20'h02500;
        6'b011010: rdata <= 20'h00301;  6'b111010: rdata <= 20'h02500;
        6'b011011: rdata <= 20'h00102;  6'b111011: rdata <= 20'h02500;
        6'b011100: rdata <= 20'h02122;  6'b111100: rdata <= 20'h0030D;
        6'b011101: rdata <= 20'h02021;  6'b111101: rdata <= 20'h02341;
        6'b011110: rdata <= 20'h00301;  6'b111110: rdata <= 20'h08201;
        6'b011111: rdata <= 20'h00102;  6'b111111: rdata <= 20'h0400D;
    endcase
end

always @(posedge clk) begin
    if (en)
        data <= rdata;
end
endmodule
```

レジスタ付きアドレス入力を持つ ROM の図



レジスタ付きアドレス入力を持つ ROM のピンの説明

I/O ピン	説明
clk	クロック (立ち上がりエッジ)
en	同期イネーブル (アクティブ High)
addr	読み出しアドレス
data	データ出力
clk	クロック (立ち上がりエッジ)

レジスタ付きアドレス入力を持つ ROM の VHDL コード例

```
--
-- ROMs Using Block RAM Resources.
-- VHDL code for a ROM with registered address
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_21c is
port (clk : in std_logic;
      en : in std_logic;
      addr : in std_logic_vector(5 downto 0);
      data : out std_logic_vector(19 downto 0));
end rams_21c;

architecture syn of rams_21c is
    type rom_type is array (63 downto 0) of std_logic_vector (19 downto 0);
    signal ROM : rom_type:= (X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A",
                             X"00300", X"08602", X"02310", X"0203B", X"08300", X"04002",
                             X"08201", X"00500", X"04001", X"02500", X"00340", X"00241",
                             X"04002", X"08300", X"08201", X"00500", X"08101", X"00602",
                             X"04003", X"0241E", X"00301", X"00102", X"02122", X"02021",
                             X"00301", X"00102", X"02222", X"04001", X"00342", X"0232B",
                             X"00900", X"00302", X"00102", X"04002", X"00900", X"08201",
                             X"02023", X"00303", X"02433", X"00301", X"04004", X"00301",
                             X"00102", X"02137", X"02036", X"00301", X"00102", X"02237",
                             X"04004", X"00304", X"04040", X"02500", X"02500", X"02500",
                             X"0030D", X"02341", X"08201", X"0400D");

    signal raddr : std_logic_vector(5 downto 0);
```

```

begin

    process (clk)
    begin
        if (clk'event and clk = '1') then
            if (en = '1') then
                raddr <= addr;
            end if;
        end if;
    end process;

    data <= ROM(conv_integer(raddr));

end syn;

```

レジスタ付きアドレス入力を持つ ROM の Verilog コード例

```

//
// ROMs Using Block RAM Resources.
// Verilog code for a ROM with registered address
//

module v_rams_2lc (clk, en, addr, data);

    input      clk;
    input      en;
    input      [5:0] addr;
    output reg [19:0] data;
    reg        [5:0] raddr;

    always @(posedge clk) begin
        if (en)
            raddr <= addr;
    end

    always @(raddr) begin
        case(raddr)
            6'b000000: data <= 20'h0200A;   6'b100000: data <= 20'h02222;
            6'b000001: data <= 20'h00300;   6'b100001: data <= 20'h04001;
            6'b000010: data <= 20'h08101;   6'b100010: data <= 20'h00342;
            6'b000011: data <= 20'h04000;   6'b100011: data <= 20'h0232B;
            6'b000100: data <= 20'h08601;   6'b100100: data <= 20'h00900;
            6'b000101: data <= 20'h0233A;   6'b100101: data <= 20'h00302;
            6'b000110: data <= 20'h00300;   6'b100110: data <= 20'h00102;
            6'b000111: data <= 20'h08602;   6'b100111: data <= 20'h04002;
            6'b001000: data <= 20'h02310;   6'b101000: data <= 20'h00900;
            6'b001001: data <= 20'h0203B;   6'b101001: data <= 20'h08201;
            6'b001010: data <= 20'h08300;   6'b101010: data <= 20'h02023;
            6'b001011: data <= 20'h04002;   6'b101011: data <= 20'h00303;
            6'b001100: data <= 20'h08201;   6'b101100: data <= 20'h02433;
            6'b001101: data <= 20'h00500;   6'b101101: data <= 20'h00301;

```

```
        6'b001110: data <= 20'h04001;    6'b101110: data <= 20'h04004;
        6'b001111: data <= 20'h02500;    6'b101111: data <= 20'h00301;
        6'b010000: data <= 20'h00340;    6'b110000: data <= 20'h00102;
        6'b010001: data <= 20'h00241;    6'b110001: data <= 20'h02137;
        6'b010010: data <= 20'h04002;    6'b110010: data <= 20'h02036;
        6'b010011: data <= 20'h08300;    6'b110011: data <= 20'h00301;
        6'b010100: data <= 20'h08201;    6'b110100: data <= 20'h00102;
        6'b010101: data <= 20'h00500;    6'b110101: data <= 20'h02237;
        6'b010110: data <= 20'h08101;    6'b110110: data <= 20'h04004;
        6'b010111: data <= 20'h00602;    6'b110111: data <= 20'h00304;
        6'b011000: data <= 20'h04003;    6'b111000: data <= 20'h04040;
        6'b011001: data <= 20'h0241E;    6'b111001: data <= 20'h02500;
        6'b011010: data <= 20'h00301;    6'b111010: data <= 20'h02500;
        6'b011011: data <= 20'h00102;    6'b111011: data <= 20'h02500;
        6'b011100: data <= 20'h02122;    6'b111100: data <= 20'h0030D;
        6'b011101: data <= 20'h02021;    6'b111101: data <= 20'h02341;
        6'b011110: data <= 20'h00301;    6'b111110: data <= 20'h08201;
        6'b011111: data <= 20'h00102;    6'b111111: data <= 20'h0400D;

    endcase
end

endmodule
```

パイプライン化された分散 RAM の HDL コーディング手法

このセクションには、次の内容が含まれます。

- ・ [パイプライン化された分散 RAM の概要](#)
- ・ [パイプライン化された分散 RAM のログ ファイル](#)
- ・ [パイプライン化された分散 RAM 関連の制約](#)
- ・ [パイプライン化された分散 RAM のコード例](#)

パイプライン化された分散 RAM の概要

XST でパイプライン化された分散 RAM を推論させると、デザイン スピードを向上できます。パイプライン化すると、分散 RAM の各段の間にレジスタを挿入することにより、デザインの全体の周波数を大幅に向上することができます。パイプライン化の効果は、「[フリップフロップのリタイミング](#)」で説明されているフリップフロップのリタイミングと同様です。

パイプライン ステージを挿入するには

1. HDL コードで必要なレジスタを記述します。
2. それらのレジスタを分散 RAM の後に配置します。
3. [RAM スタイル \(RAM_STYLE\)](#) を次の値に設定します。

pipe_distributed

XST では最大の分散 RAM の速度に到達させるため、次の両方の場合に使用可能なレジスタの最大数を使用します。

- ・ パイプラインに有効なレジスタが検出される場合
- ・ RAM_STYLE の設定 :

pipe_distributed

XST では、各 RAM で周波数を最大にするために使用するレジスタの最大数が自動的に計算されます。

アドバンス HDL 合成段階中、XST HDL Advisor からは次の場合に最適なレジスタ ステージ数を指定するようにメッセージが表示されます。

- ・ まだ十分な数のレジスタ ステージを指定していない場合
- ・ RAM_STYLE は信号に直接コード記述します。

XST では、次の場合に未使用のステージがシフトレジスタとしてインプリメントされます。

- ・ 乗算器の後に配置されたレジスタの数が必要な最大数を超える場合
- ・ シフトレジスタの抽出がオンになっている場合

レジスタに非同期セット/リセット信号が含まれていると、RAM をパイプライン化できません。レジスタに同期リセット信号が含まれている場合は、RAM をパイプライン化できます。

パイプライン化された分散 RAM のログ ファイル

パイプライン化された分散 RAM のログ ファイルの例

パイプライン化された分散 RAM のログ ファイルの例

```

*                               HDL Synthesis                               *
=====
Synthesizing Unit <rams_22>.
  Related source file is "rams_22.vhd".
  Found 64x4-bit single-port RAM for signal <RAM>.
  Found 4-bit register for signal <do>.
  Summary:
    inferred   1 RAM(s).
    inferred   4 D-type flip-flop(s).
Unit <rams_22> synthesized.

=====
HDL Synthesis Report

Macro Statistics
# RAMs                                     : 1
  64x4-bit single-port RAM               : 1
# Registers                               : 1
  4-bit register                         : 1

=====
*                               Advanced HDL Synthesis                       *
=====
INFO:Xst - Unit <rams_22> : The RAM <Mram_RAM> will be implemented as a
distributed RAM, absorbing the following register(s): <do>.

-----
| aspect ratio      | 64-word x 4-bit      |      |
| clock             | connected to signal <clk> | rise |
| write enable      | connected to signal <we> | high |
| address           | connected to signal <addr> |      |
| data in           | connected to signal <di>  |      |
| data out          | connected to internal node |      |
| ram_style         | distributed          |      |
-----

Synthesizing (advanced) Unit <rams_22>.
Found pipelined ram on signal <_varindex0000>:
  - 1 pipeline level(s) found in a register on signal <_varindex0000>.
  Pushing register(s) into the ram macro.

INFO:Xst:2390 - HDL ADVISOR - You can improve the performance of
the ram Mram_RAM by adding 1 register level(s) on output signal _varindex0000.
Unit <rams_22> synthesized (advanced).

=====
Advanced HDL Synthesis Report

Macro Statistics
# RAMs                                     : 1
  64x4-bit registered single-port distributed RAM : 1

```

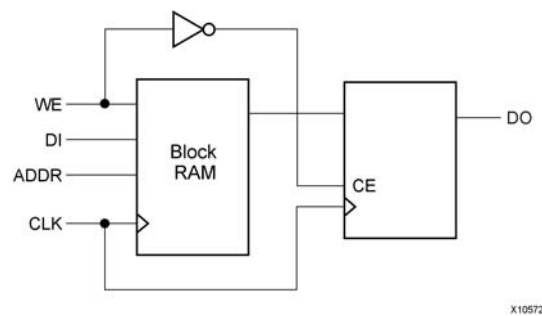
パイプライン化された分散 RAM 関連の制約

- ・ RAM の抽出 (RAM_EXTRACT)
- ・ RAM スタイル (RAM_STYLE)
- ・ ROM の抽出 (ROM_EXTRACT)
- ・ ROM スタイル (ROM_STYLE)
- ・ BRAM 使用率 (BRAM_UTILIZATION_RATIO)
- ・ 自動 BRAM パッキング (AUTO_BRAM_PACKING)

パイプライン化された分散 RAM のコード例

コード例は、ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

パイプライン化された分散 RAM の図



パイプライン化された分散 RAM のピンの説明

I/O ピン	説明
clk	クロック (立ち上がりエッジ)
we	同期書き込みイネーブル (アクティブ High)
addr	読み出し/書き込みアドレス
di	データ入力
do	データ出力

パイプライン化された分散 RAM の VHDL コード例

```
--
-- Pipeline distributed RAMs
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_22 is
    port (clk : in std_logic;
          we : in std_logic;
          addr : in std_logic_vector(8 downto 0);
          di : in std_logic_vector(3 downto 0);
          do : out std_logic_vector(3 downto 0));
end rams_22;

architecture syn of rams_22 is
    type ram_type is array (511 downto 0) of std_logic_vector (3 downto 0);
    signal RAM : ram_type;

    signal pipe_reg: std_logic_vector(3 downto 0);

    attribute ram_style: string;
    attribute ram_style of RAM: signal is "pipe_distributed";
begin

    process (clk)
    begin
        if clk'event and clk = '1' then
            if we = '1' then
                RAM(conv_integer(addr)) <= di;
            else
                pipe_reg <= RAM( conv_integer(addr));
            end if;
            do <= pipe_reg;
        end if;
    end process;

end syn;
```

パイプライン化された分散 RAM の Verilog コード例

```
//  
// Pipeline distributed RAMs  
//  
  
module v_rams_22 (clk, we, addr, di, do);  
  
    input  clk;  
    input  we;  
    input  [8:0] addr;  
    input  [3:0] di;  
    output [3:0] do;  
    (*ram_style="pipe_distributed"*)  
    reg    [3:0] RAM [511:0];  
    reg    [3:0] do;  
    reg    [3:0] pipe_reg;  
  
    always @(posedge clk)  
    begin  
        if (we)  
            RAM[addr] <= di;  
        else  
            pipe_reg <= RAM[addr];  
        do <= pipe_reg;  
    end  
  
endmodule
```

FSM の HDL コーディング手法

このセクションには、次の内容が含まれます。

- ・ [FSM コンポーネントの概要](#)
- ・ [FSM コンポーネントの記述](#)
- ・ [ステート エンコード手法](#)
- ・ [RAM ベースの FSM 合成](#)
- ・ [FSM のセーフ インプリメンテーション](#)
- ・ [FSM のログ ファイル](#)
- ・ [FSM 関連の制約](#)
- ・ [FSM のコード例](#)

FSM コンポーネントの概要

Xilinx Synthesis Technology (XST) ソフトウェア

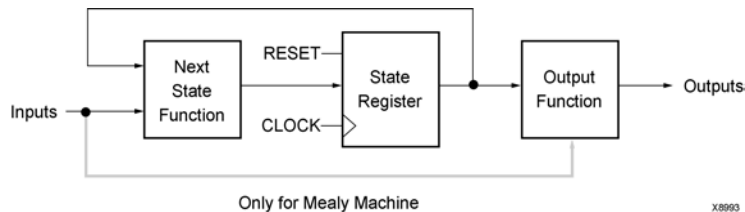
- ・ 有限ステート マシン (FSM) には、多数のテンプレートがあります。
- ・ 複数のステート エンコード手法を適用してパフォーマンスを向上またはエリアを削減可能
- ・ ユーザーの最初のエンコードを再度エンコード可能
- ・ 同期ステート マシンのみを処理可能

FSM の抽出をオフにするには、[FSM 自動抽出 \(FSM_EXTRACT\)](#) を使用します。

FSM コンポーネントの記述

有限ステート マシン (FSM) を記述する方法は多数あります。従来からの方法では、次の図に示すように、ミーリー マシンまたはムーア マシンが使用されます。XST では、両方ともサポートされます。

ミーリー マシンおよびムーア マシンを取り入れた FSM の図



process および always 文を使用した FSM コンポーネントの記述

HDL では、FSM の記述に process ブロック (VHDL) および always ブロック (Verilog) を使用する のが最適です。ここでの説明では、「プロセス」という言葉で VHDL の process ブロックと Verilog の always ブロックの両方を示します。

モデルの異なる部分をどのように分割するかによって、1 つの記述に複数のプロセス (1、2、または 3) を含めることができます。次は、非同期リセット (RESET) 付きのムーア マシンの例です。

- ・ 4 つのステート

- s1
 - s2
 - s3
 - s4

- ・ 5 つの遷移

- ・ 1 入力 :

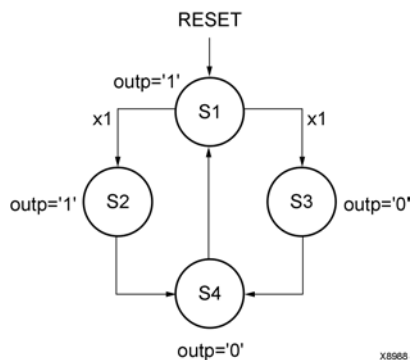
x1com

- ・ 1 出力 :

OUTP

上記のモデルは、次のステート ダイアグラムで表すことができます。

ステート ダイアグラム



ステート レジスタ

XST で FSM が問題なく認識されるようにするには、ステートレジスタに次のいずれかを含めて記述する必要があります。

- ・ 電源投入ステート

電源投入ステートは、適切な VHDL または Verilog 信号初期化を使用する必要があります。

- ・ 動作リセット

動作リセットは、非同期または同期にできます。

非同期および同期の初期化信号を記述する方法については、次のコード例を参照してください。

レジスタの HDL コーディング手法

VHDL の場合、ステートレジスタは次のタイプにできます。

- ・ **integer**
- ・ **bit_vector**
- ・ **std_logic_vector**

可能なすべてのステート値を含む列挙型を定義し、そのタイプでステートレジスタを宣言するのが一般的で便利です。

Verilog では、ステートレジスタのタイプに整数または定義されたパラメータを使用できますが、次のようにステートを割り当てることもできます。

```
parameter [3:0]  
    s1 = 4'b0001,  
    s2 = 4'b0010,  
    s3 = 4'b0100,  
    s4 = 4'b1000;  
reg [3:0] state;
```

これらのパラメータは、異なるステート エンコード方法を表すよう変更できます。

次ステートの論理式

次ステートの論理式は、順次プロセスで直接記述するか、または別の組み合わせプロセスで記述できます。最も簡潔なコード例は、case 文を使用したものです。別の組み合わせプロセスを使用する場合は、センシティビティリストにステート信号およびすべての FSM 入力を含める必要があります。

達成不可能ステート

XST では、FSM 内の unreachable ステートを検出できます。検出されたステートは、HDL 合成段階でログ ファイルに記述されます。

出力および入力

レジスタを介さない出力は、組み合わせプロセスまたは同時処理代入文で記述します。レジスタを介する出力は、順次プロセス内で代入する必要があります。

レジスタを介する入力は、順次プロセスで代入する内部信号を使用して記述します。

ステート エンコード手法

XST では、次のステート エンコード手法がサポートされます。

- ・ [自動ステート エンコード](#)
- ・ [ワンホット ステート エンコード](#)
- ・ [グレイ ステート エンコード](#)
- ・ [コンパクト ステート エンコード](#)
- ・ [ジョンソン ステート エンコード](#)
- ・ [シーケンシャル ステート エンコード](#)
- ・ [Speed1 ステート エンコード](#)
- ・ [ユーザー ステート エンコード](#)

自動ステート エンコード

自動ステート エンコードでは、XST により各 FSM に最適なエンコード アルゴリズムが選択されます。

ワンホット ステート エンコード

ワンホット ステート エンコードには、次のような特徴があります。

- ・ デフォルトのエンコーディング方法です。
- ・ 各ステートに 1 つのコード ビットおよび 1 つのフリップフロップを割り当てます。1 つのクロック サイクルで 1 つのステート変数のみがアサートになります。2 つのステート間で遷移するときには、2 つのビットのみが切り替わります。
- ・ ほとんどの FPGA デバイスの場合、フリップフロップが多数があるため、ワンホット エンコードが適しています。
- ・ スピードを最適化する場合や、消費電力を低減する場合に適した手法です。

グレイ ステート エンコード

グレイ ステート エンコードは、

- ・ 連続した 2 つのステート間で、1 ビットしか切り替わりません。
- ・ 分岐のない長いパスを持つコントローラに適しています。
- ・ ハザードやグリッチを最小限に抑えます。
- ・ T フリップフロップの付いたステートレジスタをインプリメントするときに使用するのをお勧めします。

コンパクト ステート エンコード

コンパクト ステート エンコードには、次のような特徴があります。

- ・ ステート変数およびフリップフロップの数を最小限にします。
- ・ ハイパーキューブ イメージョンに基づいています。
- ・ エリアを最適化する際に適しています。

ジョンソン ステート エンコード

このエンコード手法は、グレイ ステート エンコードと同様、分岐のない長いパスを含むステートマシンに適しています。

シーケンシャル ステート エンコード

シーケンシャル ステート エンコードには、次のような特徴があります。

- ・ 長いパスを特定し、これらのパスのステートに連続する基数コードを 2 つ適用します。
- ・ 次ステートの論理式を最小限に抑えます。

Speed1 ステート エンコード

Speed1 ステート エンコードは、スピードを最適化する場合に使用します。ステートレジスタのビット数は、FSM によって異なりますが、通常 FSM ステート数よりも多くなります。

ユーザー ステート エンコード

ユーザー ステート エンコードでは、ユーザーが独自のエンコード手法を HDL ファイルで指定できます。たとえば、ステートレジスタに列挙型を使用する場合、[列挙型エンコード手法 \(ENUM_ENCODING\)](#) 制約を使用して各ステートに特定の 2 進数値を割り当てることができます。

詳細は、次を参照してください。

[デザイン制約](#)

RAM ベースの FSM 合成

大型の Finite State Machine (FSM) コンポーネントは、Virtex® 以降のデバイスに搭載されているブロック RAM リソースにインプリメントすると、コンパクトで高速にできます。[FSM スタイル \(FSM_STYLE\)](#) を使用すると、FSM にブロック RAM リソースが使用されます。

[FSM スタイル \(FSM_STYLE\)](#) の値には、次があります。

- ・ **lut** (デフォルト)
XST では、LUT を使用して FSM をマップします。
- ・ **bram**
XST では、ブロック RAM 上に FSM をマップします。

[FSM スタイル \(FSM_STYLE\)](#) は次のように指定します。

- ・ ISE® Design Suite
[Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックスで、[HDL Options] ページにある [FSM Style] プロパティを [LUT] または [Bram] に設定します。
- ・ コマンド ライン
-fsm_style オプションを使用します。
- ・ HDL コード
[FSM スタイル \(FSM_STYLE\)](#) を使用します。

ブロック RAM にステート マシンをインプリメントできない場合は、次のように処理されます。

- ・ ログ ファイルのアドバンス HDL 合成部分に警告が表示されます。
- ・ ステート マシンが自動的に LUT を使用してインプリメントされます。

たとえば、FSM に非同期リセットがある場合、ブロック RAM にはインプリメントできません。その場合、次のようなメッセージが表示されます。

```
...
=====
*                               Advanced HDL Synthesis                               *
=====

WARNING:Xst - Unable to fit FSM <FSM_0> in BRAM (reset is
asynchronous).
Selecting encoding for FSM_0 ...
Optimizing FSM <FSM_0> on signal <current_state>
with one-hot encoding.
...
```

FSM のセーフ インプリメンテーション

XST では、ステート マシンが不正なステートから回復できるようにするロジックを有限ステート マシン (FSM) のインプリメンテーションに追加できます。ステート マシンが実行中に不正のステートになった場合は、XST で追加されたロジックによってリカバリ ステートと呼ばれる既知のステートに戻すことができます。これは、セーフ インプリメンテーション モードと呼ばれます。

FSM のセーフ インプリメンテーションの設定方法 :

- ・ ISE® Design Suite から [Synthesize -XST] プロセスの [Process Properties] ダイアログ ボックスを表示して、[HDL Options] ページで [Safe Implementation] オプションを [Yes] にします。
- ・ ステートレジスタを表現する階層ブロックまたは信号に **セーフ インプリメンテーション (SAFE_IMPLEMENTATION)** 制約を設定します。

XST では、デフォルトでリセット ステートがリカバリ ステートとして自動的に選択されます。FSM に初期化信号がない場合は、パワーアップ ステートがリカバリ ステートとして選択されます。**セーフ リカバリ ステート (SAFE_RECOVERY_STATE)** 制約を適用すると、手動でリカバリ ステートを定義できます。

FSM のログ ファイル

XST ログ ファイルには、マクロの認識段階で認識された有限ステート マシン (FSM) の情報がすべて示されます。FSM のエンコード アルゴリズムが自動的に選択されるよう設定している場合は、選択されたアルゴリズムが示されます。エンコード手法が選択されると、FSM の元のエンコードと FSM エンコードがレポートされます。使用するデバイス ファミリが FPGA の場合、エンコード手法は HDL 合成段階でレポートされます。CPLD の場合は、下位レベルの最適化段階でレポートされます。

FSM のログ ファイルの例

```
...
Synthesizing Unit <fsm_1>.
  Related source file is "/state_machines_1.vhd".
  Found finite state machine <FSM_0> for signal <state>.
  -----
  | States                | 4 |
  | Transitions           | 5 |
  | Inputs                | 1 |
  | Outputs               | 4 |
  | Clock                 | clk (rising_edge) |
  | Reset                 | reset (positive) |
  | Reset type            | asynchronous |
  | Reset State           | s1 |
  | Power Up State        | s1 |
  | Encoding              | automatic |
  | Implementation       | LUT |
  -----
  Found 1-bit register for signal <outp>.
  Summary:
    inferred    1 Finite State Machine(s).
    inferred    1 D-type flip-flop(s).
```

```

Unit <fsm_1> synthesized.

=====
HDL Synthesis Report

Macro Statistics
# Registers                      : 1
  1-bit register                 : 1

=====
*                               Advanced HDL Synthesis                               *
=====

Advanced Registered AddSub inference ...
Analyzing FSM <FSM_0> for best encoding.
Optimizing FSM <state/FSM_0> on signal <state[1:2]>
with gray encoding.

-----
State | Encoding
-----
s1    | 00
s2    | 01
s3    | 11
s4    | 10
-----

=====
HDL Synthesis Report

Macro Statistics
# FSMs                          : 1
=====

```

FSM 関連の制約

- ・ [FSM 自動抽出 \(FSM_EXTRACT\)](#)
- ・ [FSM スタイル \(FSM_STYLE\)](#)
- ・ [FSM エンコード方法の指定 \(FSM_ENCODING\)](#)
- ・ [列挙型エンコード手法 \(ENUM_ENCODING\)](#)
- ・ [セーフ インプリメンテーション \(SAFE_IMPLEMENTATION\)](#)
- ・ [セーフ リカバリ ステート \(SAFE_RECOVERY_STATE\)](#)

FSM のコード例

コード例は、[ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip](http://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip) からダウンロードしてください。

1 つのプロセス ブロックを使用した FSM のピンの説明

I/O ピン	説明
clk	クロック (立ち上がりエッジ)
reset	非同期リセット (アクティブ High)
X1	FSM の入力
OUTP	FSM の出力

1 つのプロセス ブロックを使用した FSM の VHDL コード例

```
--
-- State Machine with a single process.
--

library IEEE;
use IEEE.std_logic_1164.all;
entity fsm_1 is
    port ( clk, reset, x1 : IN std_logic;
          outp           : OUT std_logic);
end entity;

architecture beh1 of fsm_1 is
    type state_type is (s1,s2,s3,s4);
    signal state: state_type ;
begin

    process (clk,reset)
    begin
        if (reset ='1') then
            state <=s1;
            outp<='1';
        elsif (clk='1' and clk'event) then
            case state is
                when s1 =>  if x1='1' then
                            state <= s2;
                            outp <= '1';
                        else
                            state <= s3;
                            outp <= '0';
                        end if;
                when s2 => state <= s4; outp <= '0';
                when s3 => state <= s4; outp <= '0';
                when s4 => state <= s1; outp <= '1';
            end case;
        end if;
    end process;

end beh1;
```

1 つの always ブロックを使用した FSM の Verilog コード例

```
//  
// State Machine with a single always block.  
//  
  
module v_fsm_1 (clk, reset, x1, outp);  
    input  clk, reset, x1;  
    output outp;  
    reg    outp;  
    reg    [1:0] state;  
  
    parameter s1 = 2'b00; parameter s2 = 2'b01;  
    parameter s3 = 2'b10; parameter s4 = 2'b11;  
  
    initial begin  
        state = 2'b00;  
    end  
  
    always@(posedge clk or posedge reset)  
    begin  
        if (reset)  
        begin  
            state <= s1; outp <= 1'b1;  
        end  
        else  
        begin  
            case (state)  
                s1: begin  
                    if (x1==1'b1)  
                    begin  
                        state <= s2;  
                        outp <= 1'b1;  
                    end  
                    else  
                    begin  
                        state <= s3;  
                        outp <= 1'b0;  
                    end  
                end  
                s2: begin  
                    state <= s4; outp <= 1'b1;  
                end  
                s3: begin  
                    state <= s4; outp <= 1'b0;  
                end  
                s4: begin  
                    state <= s1; outp <= 1'b0;  
                end  
            endcase  
        end  
    end  
end
```

```

        end
    end

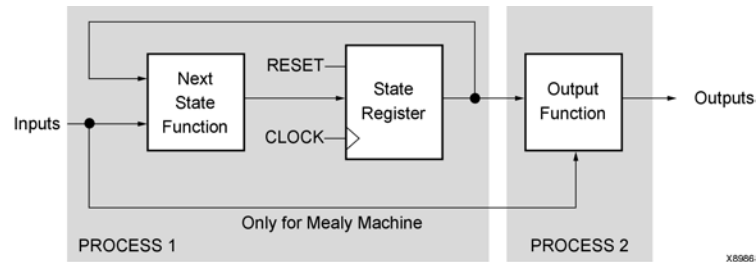
endmodule

```

2 つのプロセス ブロックを使用した FSM

出力からレジスタを削除するには、クロック同期セクションから代入文 **outp <=...** をすべて削除します。これには、に示すように 2 つのプロセスを使用します。

2 つのプロセス ブロックを使用した FSM の図



2 つのプロセス ブロックを使用した FSM のピンの説明

I/O ピン	説明
clk	クロック (立ち上がりエッジ)
reset	非同期リセット (アクティブ High)
x1	FSM の入力
outp	FSM の出力

2 つのプロセス ブロックを使用した FSM の VHDL コード例

```

--
-- State Machine with two processes.
--

library IEEE;
use IEEE.std_logic_1164.all;
entity fsm_2 is
    port ( clk, reset, x1 : IN std_logic;
           outp           : OUT std_logic);
end entity;

architecture beh1 of fsm_2 is
    type state_type is (s1,s2,s3,s4);
    signal state: state_type ;
begin

    process1: process (clk,reset)
    begin
        if (reset ='1') then state <=s1;
        elsif (clk='1' and clk'Event) then

```

```

        case state is
            when s1 => if x1='1' then
                           state <= s2;
                        else
                           state <= s3;
                        end if;
            when s2 => state <= s4;
            when s3 => state <= s4;
            when s4 => state <= s1;
        end case;
    end if;
end process process1;

process2 : process (state)
begin
    case state is
        when s1 => outp <= '1';
        when s2 => outp <= '1';
        when s3 => outp <= '0';
        when s4 => outp <= '0';
    end case;
end process process2;

end beh1;

```

2 つの always ブロックを使用した FSM の Verilog コード例

```

//
// State Machine with two always blocks.
//

module v_fsm_2 (clk, reset, x1, outp);
    input  clk, reset, x1;
    output outp;
    reg    outp;
    reg    [1:0] state;

    parameter s1 = 2'b00; parameter s2 = 2'b01;
    parameter s3 = 2'b10; parameter s4 = 2'b11;

    initial begin
        state = 2'b00;
    end

    always @(posedge clk or posedge reset)
    begin
        if (reset)
            state <= s1;
        else
            begin
                case (state)

```

```

s1: if (x1==1'b1)
    state <= s2;
else
    state <= s3;
s2: state <= s4;
s3: state <= s4;
s4: state <= s1;
endcase
end
end

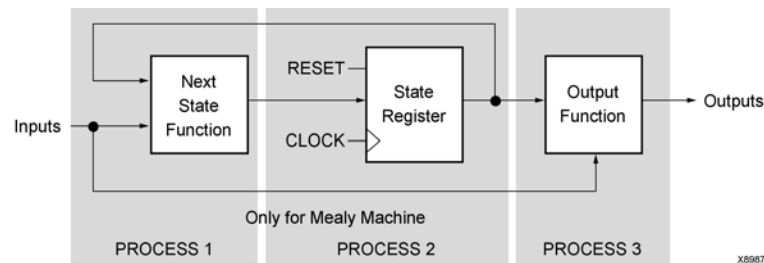
always @(state)
begin
    case (state)
        s1: outp = 1'b1;
        s2: outp = 1'b1;
        s3: outp = 1'b0;
        s4: outp = 1'b0;
    endcase
end

endmodule

```

ステートレジスタから次ステート関数を分離することもできます。

3つのプロセスブロック使用した FSM の図



3つのプロセスブロックを使用した FSM のピンの説明

I/O ピン	説明
clk	クロック (立ち上がりエッジ)
reset	非同期リセット (アクティブ High)
x1	FSM の入力
outp	FSM の出力

3つのプロセスブロックを使用した FSM の VHDL コード例

```

--
-- State Machine with three processes.
--

```

```
library IEEE;
use IEEE.std_logic_1164.all;
entity fsm_3 is
    port ( clk, reset, x1 : IN std_logic;
          outp           : OUT std_logic);
end entity;

architecture beh1 of fsm_3 is
    type state_type is (s1,s2,s3,s4);
    signal state, next_state: state_type ;
begin

    process1: process (clk,reset)
    begin
        if (reset ='1') then
            state <=s1;
        elsif (clk='1' and clk'Event) then
            state <= next_state;
        end if;
    end process process1;

    process2 : process (state, x1)
    begin
        case state is
            when s1 =>  if x1='1' then
                           next_state <= s2;
                       else
                           next_state <= s3;
                       end if;
            when s2 => next_state <= s4;
            when s3 => next_state <= s4;
            when s4 => next_state <= s1;
        end case;
    end process process2;

    process3 : process (state)
    begin
        case state is
            when s1 => outp <= '1';
            when s2 => outp <= '1';
            when s3 => outp <= '0';
            when s4 => outp <= '0';
        end case;
    end process process3;

end beh1;
```

3 つの always ブロックを使用した FSM の Verilog コード例

```
//
// State Machine with three always blocks.
```

```
//  
  
module v_fsm_3 (clk, reset, x1, outp);  
    input clk, reset, x1;  
    output outp;  
    reg outp;  
    reg [1:0] state;  
    reg [1:0] next_state;  
  
    parameter s1 = 2'b00; parameter s2 = 2'b01;  
    parameter s3 = 2'b10; parameter s4 = 2'b11;  
  
    initial begin  
        state = 2'b00;  
    end  
  
    always @(posedge clk or posedge reset)  
    begin  
        if (reset) state <= s1;  
        else state <= next_state;  
    end  
  
    always @(state or x1)  
    begin  
        case (state)  
            s1: if (x1==1'b1)  
                next_state = s2;  
                else  
                    next_state = s3;  
            s2: next_state = s4;  
            s3: next_state = s4;  
            s4: next_state = s1;  
        endcase  
    end  
  
    always @(state)  
    begin  
        case (state)  
            s1: outp = 1'b1;  
            s2: outp = 1'b1;  
            s3: outp = 1'b0;  
            s4: outp = 1'b0;  
        endcase  
    end  
  
endmodule
```

ブラック ボックスの HDL コーディング手法

このセクションには、次の内容が含まれます。

- ・ [ブラック ボックスの概要](#)
- ・ [ブラック ボックスのログ ファイル](#)
- ・ [ブラック ボックス関連の制約](#)
- ・ [ブラック ボックスのコード例](#)

ブラック ボックスの概要

デザインには、次で生成された Electronic Data Interchange Format (EDIF) または NG ファイルが含まれることがあります。

- ・ 合成ツール
- ・ 回路図テキスト エディタ
- ・ その他のデザイン入力方法

これらのモジュールをデザインに関連付けるには、デザインのコードにインスタンスエートする必要があります。これを XST で実行させるには、VHDL または Verilog コードにブラック ボックスのインスタンスエーションを使用します。インスタンスエートされたネットリストは XST では処理されず、最終の最上位ネットリストに含まれます。また、ブラック ボックスのインスタンスエーションに制約を指定することも可能です。指定した制約も、NGC ファイルに記述されます。

また、デザイン ブロックの Register Transfer Level (RTL) モデルおよび EDIF ネットリストがある場合があります。RTL モデルはシミュレーションにしか使用できませんが、[BoxType \(BOX_TYPE\)](#) 制約を使用すると、この RTL コードを合成しないで、ブラック ボックスを作成するように設定できます。EDIF ネットリストは、NGDBuild (変換) により合成されたデザインに関連付けられます。

デザインをブラック ボックスにすると、そのデザインのほかのインスタンスもブラック ボックスになります。インスタンスに制約を指定すると、元のモジュールに指定されていた制約は無視されます。

詳細は、次を参照してください。

- ・ [「一般制約」](#)
- ・ [『制約ガイド』](#)

ブラック ボックスのログ ファイル

XST ではマクロ推論の前にブラック ボックスが認識されるので、ブラック ボックスのログ ファイルはほかのマクロのものとは異なります。

ブラック ボックスのログ ファイルの例

```
...
Analyzing Entity <black_b> (Architecture <archi>).

WARNING:Xst:766 - black_box_1.vhd (Line 15).
Generating a Black Box for component <my_block>.
Entity <black_b> analyzed. Unit <black_b> generated
....
```

ブラック ボックス関連の制約

ボックス タイプ (BOX_TYPE)

BOX_TYPE は XST でデバイス プリミティブをインスタンスシートするために導入された制約です。この制約を使用する前に、次を参照してください。

デバイス プリミティブのサポート

ブラック ボックスのコード例

コード例は、ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

ブラック ボックスの VHDL コード例

```
--  
-- Black Box  
--  
  
library ieee;  
use ieee.std_logic_1164.all;  
  
entity black_box_1 is  
    port(DI_1, DI_2 : in std_logic;  
          DOUT : out std_logic);  
end black_box_1;  
  
architecture archi of black_box_1 is  
  
    component my_block  
    port (I1 : in std_logic;  
          I2 : in std_logic;  
          O : out std_logic);  
    end component;  
  
begin  
  
    inst: my_block port map (I1=>DI_1,I2=>DI_2,O=>DOUT);  
  
end archi;
```

ブラック ボックスの Verilog コード例

```
//  
// Black Box  
//  
  
module v_my_block (in1, in2, dout);  
    input in1, in2;  
    output dout;  
endmodule  
  
module v_black_box_1 (DI_1, DI_2, DOUT);  
    input DI_1, DI_2;  
    output DOUT;  
  
    v_my_block inst (  
        .in1 (DI_1),  
        .in2 (DI_2),  
        .dout (DOUT) );  
  
endmodule
```

コンポーネントのインスタンス化の詳細は、VHDL/Verilog のマニュアルを参照してください。

FPGA の最適化

この章には、次の内容が含まれます。

- ・ [FPGA の合成および最適化](#)
- ・ [FPGA 専用の合成オプション](#)
- ・ [マクロ生成](#)
- ・ [DSP48 ブロック リソース](#)
- ・ [ブロック RAM へのロジックのマップ](#)
- ・ [フリップフロップのリタイミング](#)
- ・ [パーティション](#)
- ・ [エリア制約を設定した場合のスピード最適化](#)
- ・ [FPGA 最適化レポート](#)
- ・ [インプリメンテーション制約](#)
- ・ [FPGA デバイス プリミティブのサポート](#)
- ・ [コアの処理](#)
- ・ [INIT および RLOC の指定](#)
- ・ [XST での PCI™ フローの使用](#)

FPGA の合成および最適化

XST は、FPGA の合成および最適化で次の処理を実行します。

- ・ エンティティ/モジュールごとにマップおよび最適化
- ・ デザイン全体のグローバル最適化

このプロセスで、NGC ファイルが出力されます。

FPGA 専用の合成オプション

XST では、ユーザー制約を満たすため、FPGA 合成を詳細に設定できる次のオプションがサポートされています。

- ・ BUFGCE の抽出 (BUFGCE)
- ・ コアの検索ディレクトリ (-sd)
- ・ デコーダの抽出 (DECODER_EXTRACT)
- ・ FSM スタイル (FSM_STYLE)
- ・ グローバルな最適化目標 (-glob_opt)
- ・ 階層の維持 (KEEP_HIERARCHY)
- ・ 論理シフトの抽出 (SHIFT_EXTRACT)
- ・ BRAM へのロジックのマッピング (BRAM_MAP)
- ・ 最大ファンアウト数 (MAX_FANOUT)
- ・ 最初のフリップフロップ ステージの移動 (MOVE_FIRST_STAGE)
- ・ 最後のフリップフロップ ステージの移動 (MOVE_LAST_STAGE)
- ・ 乗算器スタイル (MULT_STYLE)
- ・ MUX スタイル (MUX_STYLE)
- ・ グローバル クロック バッファ数 (-bufg)
- ・ インスタンス化されたプリミティブの最適化 (OPTIMIZE_PRIMITIVES)
- ・ I/O レジスタの IOB 内へのパック (IOB)
- ・ プライオリティ エンコーダの抽出 (PRIORITY_EXTRACT)
- ・ RAM スタイル (RAM_STYLE)
- ・ レジスタの自動調整 (REGISTER_BALANCING)
- ・ レジスタの複製 (REGISTER_DUPLICATION)
- ・ 信号のエンコード方法 (SIGNAL_ENCODING)
- ・ スライス パッキング (-slice_packing)
- ・ キャリーチェーンの使用 (USE_CARRY_CHAIN)
- ・ タイミング制約の書き込み (-write_timing_constraints)
- ・ XOR コラプス (XOR_COLLAPSE)

詳細は、次を参照してください。

FPGA 制約 (タイミング制約以外)

マクロ生成

FPGA デバイスのマクロ ジェネレータ モジュールを使用すると、さまざまなファンクションが XST の HDL フローで使用できるようになります。これらのファンクションは推論エンジンにより HDL 記述から識別され、最適なインプリメンテーションが実行されるように、その特性がマクロ ジェネレータに渡されます。

推論されるファンクションには、加算器、アキュムレータ、カウンタ、マルチプレクサなどの単純な数値演算ブロックから、乗算器、シフトレジスタ、メモリなどの複雑なブロックまで、さまざまな種類があります。

推論されたファンクションは、Virtex® または Spartan® アーキテクチャで最高のパフォーマンスを実現できるよう最適化され、デザインに組み込まれます。また、デザインによっては、ファンクションの周囲にあるロジックと共に最適化される場合もあります。

このセクションでは、マクロをファンクション別に分類し、その構築および最適化の段階で利用されるリソースについて簡単に説明します。

マクロの生成は、属性を設定することにより制御できます。これらの属性は、各サブセクションにリストされています。

XST では、専用キャリー チェーン ロジックを使用して、多くのマクロがインプリメントされます。場合によって、キャリー チェーン ロジックにより最適な結果が得られない場合もあります。[キャリーチェーンの使用 \(USE_CARRY_CHAIN\)](#) を使用すると、この機能をオフにできます。

詳細は、次を参照してください。

[デザイン制約](#)

数値演算ファンクション

数値演算ファンクションには、次のエレメントがあります。

- ・ 加算器、減算器、加減算器
- ・ カスケード接続可能なバイナリ カウンタ
- ・ アキュムレータ
- ・ インクリメンタ、ディクリメンタ、インクリメンタ/ディクリメンタ
- ・ 符号付き/符号なし乗算器

XST では、キャリーの高速処理が可能なキャリー ロジック (MUXCY) が利用されるため、高速な数値演算ファンクションを実現できます。2 つの XOR ゲートで構成される積和ロジックは LUT および専用キャリー XOR (XORCY) を使用してインプリメントされます。また、専用キャリー AND (MULTAND) リソースも提供されており、高速な乗算器をインプリメントできます。

マクロ生成におけるロード可能ファンクション

ロード可能なファンクションには、次のエレメントがあります。

- ・ ロード可能アップ カウンタ、ダウン カウンタ、アップ/ダウン カウンタ
- ・ ロード可能アップ アキュムレータ、ダウン アキュムレータ、アップ/ダウン アキュムレータ

XST では、同期ロード可能、カスケード接続可能なカウンタおよびアキュムレータも推論されます。マクロの各段のカスケード接続には、高速キャリー ロジックが利用されます。同期ロードおよびカウント ファンクションは、1 つの LUT プリミティブにインプリメントされます。

アップ/ダウン カウンタおよびアキュムレータでは、パフォーマンスを向上するため、専用キャリー AND が利用されます。

マルチプレクサ

マルチプレクサには、次の 2 つのアーキテクチャがあります。

- ・ MUXFx ベースのマルチプレクサ
- ・ 専用キャリー MUX ベースのマルチプレクサ

Virtex®-4 デバイスの場合、次の表のプリミティブを使用してマルチプレクサをインプリメントできます。

マルチプレクサ	CLB	プリミティブ
16:1	single CLB	MUXF7
32:1	across two CLBs	MUXF8

マルチプレクサの推論を制御するため、XST では MUXF5/MUXF6 または専用キャリー MUX のどちらを使用するかを指定できます。MUX_STYLE 属性を MUXF に設定すると MUXF_x ベースのマルチプレクサが使用され、MUXCY に設定すると専用キャリー MUX ベースのマルチプレクサが使用されます。

この属性は、マルチプレクサを定義する信号またはマルチプレクサのインスタンス名に設定します。この属性をグローバルに設定することも可能です。

MUX_EXTRACT 属性を no または force に設定すると、マルチプレクサの推論を無効にまたは強制できます。

MUX_EXTRACT 制約を使ってマルチプレクサの推論を無効にしている場合でも、MUXF_x エLEMENTが残ることがあります。こういったELEMENTは、ブール代数式の一般的なマップからのものです。

プライオリティ エンコーダ

「プライオリティ エンコーダの HDL コーディング手法」で説明されている if/elseif 構造は、1-of-n プライオリティ エンコーダでインプリメントされます。

XST では MUXCY プリミティブを使用してプライオリティ エンコーダの条件をチェーン接続し、高速のインプリメンテーションを実現します。

プライオリティ エンコーダの抽出 (PRIORITY_EXTRACT) 制約を使用すると、プライオリティ エンコーダ マクロの推論を有効または無効にできます。

通常、XST ではプライオリティ エンコーダは推論されないため、多数のプライオリティ エンコーダが生成されることはありません。プライオリティ エンコーダをイネーブルにするには、プライオリティ エンコーダの抽出 (PRIORITY_EXTRACT) 制約を force オプションで使します。

マクロ生成のデコーダ

デコーダは、各入力値に対して 1 つのワンホット (またはワンコールド) 値が指定されているデマルチプレクサです。n ビットまたは 1-of-m デコーダには、m ビットのデータ出力および n ビットのセレクト入力があり、n と m の関係は次に示すとおりです。

$$n \times (2^m - 1) < m \leq n \times 2^m$$

デコーダが推論されると、デコーダのサイズに応じて MUXF5 または MUXCY プリミティブがインプリメントされます。

デコーダの抽出 (DECODER_EXTRACT) 制約を使用して、デコーダの推論を有効または無効にします。

RAM (マクロ生成)

推論および生成中には、次の RAM が使用可能です。

- 分散 RAM
RAM が非同期読み出しの場合は、分散 RAM が推論され、生成されます。
- ブロック RAM (デフォルト)
RAM が同期読み出しの場合は、ブロック RAM が推論されます。この場合、ブロック RAM または分散 RAM のどちらでもインプリメントできます。

XST で使用されるプリミティブ

このセクションは、次のデバイスに適用されます。

- Virtex®-4
- Spartan®-3

これらのデバイスの場合、XST では次の表のプリミティブが使用されます。

RAM	クロック エッジ	プリミティブ
シングル ポート同期分散 RAM	クロックの立ち上がりエッジで動作するシングル ポート分散 RAM	RAM16X1S、RAM16X2S、RAM16X4S、RAM16X8S、RAM32X1S、RAM32X2S、RAM32X4S、RAM32X8S、RAM64X1S、RAM64X2S、RAM128X1S
シングル ポート同期分散 RAM	クロックの立ち下がりエッジで動作するシングル ポート分散 RAM	RAM16X1S_1、RAM32X1S_1、RAM64X1S_1、RAM128X1S_1
デュアル ポート同期分散 RAM	クロックの立ち上がりエッジで動作するデュアル ポート分散 RAM	RAM16X1D、RAM32X1D、RAM64X1D
デュアル ポート同期分散 RAM	クロックの立ち下がりエッジで動作するデュアル ポート分散 RAM	RAM16X1D_1、RAM32X1D_1、RAM64X1D_1
シングル ポート同期ブロック RAM	なし	RAMB4_Sn
デュアル ポート同期ブロック RAM	なし	RAMB4_Sm_Sn

推論された RAM のインプリメンテーションの制御

RAM の推論を制御するため、分散 RAM またはブロック RAM (可能な場合) のどちらを使用するかを指定できます。

RAM Style (RAM_STYLE) 属性の値は次のいずれかに指定します。

- ブロック RAM の場合は、block
- 分散 RAM の場合は distributed

RAM スタイル (RAM_STYLE) は、次に適用します。

- RAM を定義する信号
- RAM のインスタンス名

RAM スタイル (RAM_STYLE) 属性は、グローバルに設定することもできます。

RAM リソースが足りない場合は、レジスタを使用して RAM を生成できます。その場合は、**RAM の抽出 (RAM_EXTRACT)** 制約を no に設定します。

ROM (マクロ生成)

このセクションでは、マクロ生成の ROM について説明します。

- ・ 割り当てた値が定数の場合の ROM の推論
- ・ 配列からの ROM の推論
- ・ 推論および生成中に使用可能な ROM のタイプ
- ・ XST で推論される同期 ROM のタイプ
- ・ RAM スタイルの指定

割り当てた値が定数の場合の ROM の推論

case または if - else 文ですべての値を定数にすると ROM が推論されます。16 ワード以上の ROM (ビット幅は制限なし) のみが推論されます。たとえば、次の Hardware Description Language (HDL) の式では 16 ワード、4 ビット幅の ROM がインプリメントされます。

```
data = if address = 0000 then 0010
      if address = 0001 then 1100
      if address = 0010 then 1011
      ...
      if address = 1111 then 0001
```

配列からの ROM の推論

また次の例のように、すべて定数で構成されている配列からも ROM が推論されます

```
type ROM_TYPE is array(15 downto 0) of std_logic_vector(3 downto 0);
constant ROM : rom_type := ("0010", "1100", "1011", ..., "0001");
...
data <= ROM(conv_integer(address));
```

ROM の抽出 (ROM_EXTRACT) 制約を使用すると、ROM の推論を無効にできます。

- ・ ROM の推論を有効にするには ROM_EXTRACT の値を yes に設定します。
- ・ ROM の推論を無効にするには ROM_EXTRACT の値を no に設定します。

デフォルトは、yes です。

推論および生成中に使用可能な ROM のタイプ

推論および生成中には、次の 2 種類の ROM が使用できます。

- ・ 分散 ROM

分散 ROM を使用すると、LUT、MUXF5、MUXF6、MUXF7、MUXF8 プリミティブのツリー構造を使用して、大型の ROM を小さな領域にインプリメントできます。

- ・ ブロック ROM

ブロック ROM は、ブロック RAM リソースを使用して生成されます。同期 ROM が認識されると、レジスタ付き分散 ROM として推論するか、またはブロック RAM リソースを使用して推論できます。

XST で推論される同期 ROM のタイプ

ROM スタイル (ROM_STYLE) 制約を使用すると、XST で推論される同期 ROM のタイプを次のように指定できます。

オプション	XST の動作
block	ROM が 1 つの RAM に収まる場合はブロック RAM リソースを使用してインプリメントされます。
distributed	分散 ROM とレジスタを推論します。
auto (デフォルト)	ROM を使用および推論するのに最も効率的な方法が選択されます。

RAM スタイルの指定

RAM スタイル (RAM_STYLE) は、VHDL 属性または Verilog メタ コメントとして、次に指定できます。

- ・ 個々の信号
- ・ ROM のエンティティまたはモジュール

RAM スタイル (RAM_STYLE) は、次からグローバルに設定することもできます。

- ・ ISE® Design Suite での設定：
[Process] → [Process Properties]
- ・ コマンド ライン

DSP48 ブロック リソース

このセクションでは、DSP48 ブロック リソースについて次の内容に分けて説明します。

- ・ **DSP48 ブロックのマクロ インプリメンテーション**
- ・ **自動 DSP リソース管理機能の無効化**
- ・ **最大マクロ コンフィギュレーション**
- ・ **非同期セット/リセット信号**
- ・ **インターコネクトされたマクロ**

DSP48 ブロックのマクロ インプリメンテーション

XST では、次のマクロを 1 つの DSP48 ブロックに自動的にインプリメントできます。

- ・ 加算器/減算器
- ・ アキュムレータ
- ・ 乗算器
- ・ 乗算/加減算器
- ・ MAC (積和演算)

XST では、上記のマクロにレジスタが付いているものもサポートされています。

DSP48 ブロックのマクロ インプリメンテーションは、**DSP48 の使用 (USE_DSP48)** 制約をデフォルト値の auto に設定して制御します。

auto に設定すると、アキュムレータ、乗算器、乗算/加減算器、および MAC はできる限り DSP48 リソースを使用してインプリメントされますが、加減算器は DSP48 リソースにはインプリメントされません。加算器/減算器を DSP48 にインプリメントするには、**DSP48 の使用 (USE_DSP48)** を yes に設定します。

auto モードにすると、すべてのマクロが自動的に制御されます。また、使用可能な DSP48 リソースの数を **DSP 使用率(DSP_UTILIZATION_RATIO)** 制約で制御できます。デフォルトでは、使用可能な DSP48 リソースが可能な限り使用されます。

自動 DSP リソース管理機能の無効化

ユーザーの指定した DSP スライス数がターゲット FPGA デバイスの DSP リソース数を上回る場合は、XST で警告メッセージが表示され、チップ上の使用可能な DSP リソースのみが使用されて合成されます。DSP リソースが自動的に管理されないように、DSP_UTILIZATION_RATIO をオフにして、推論される DSP の数を確認してみてください。自動的にリソースが管理されないようにするには、値に -1 を指定します。

最大マクロ コンフィギュレーション

XST では、DSP48 に最大数のレジスタを含めるなど、デフォルトで最大限のマクロ コンフィギュレーションを推論およびインプリメントすることで、最良のパフォーマンスを達成しようとしています。マクロを特定のコンフィギュレーションにする場合は、**キープ (KEEP)** 制約を使用する必要があります。たとえば、各入力に 2 段のレジスタが付いている乗算器では、**キープ (KEEP)** 制約をこれらのレジスタの出力に設定して、最初のレジスタが DSP48 に含まれないようにする必要があります。

非同期セット/リセット信号

DSP48 ブロックでは、非同期セット/リセット信号の付いたレジスタがサポートされません。このため、こういったレジスタは DSP48 には含まれず、最適なデザイン パフォーマンスにはならないこともあります。**非同期から同期への変換 (ASYNC_TO_SYNC)** 制約を使用すると、デザイン全体の非同期セット/リセット信号を同期信号に置き換えることができます。これにより、DSP48 にレジスタを組み込んで結果を改善することができます。

非同期セット/リセット信号を同期信号に置換すると、生成した NGC ネットリストが最初の RTL 記述と同じではなくなります。合成したデザインが最初の仕様を満たしているかどうか必ず確認してください。詳細は、「**非同期から同期への変換 (ASYNC_TO_SYNC)**」を参照してください。

各マクロの処理に関する詳細は、次を参照してください。

HDL コーディング手法

インターコネクトされたマクロ

内部接続されているマクロが複数含まれる場合に、それぞれが DSP48 にインプリメント可能な場合、高速 BCIN/BCOUT および PCIN/PCOUT 接続を使用して DSP48 ブロック間が内部接続されます。通常フィルタや複雑な乗算器がこの例です。

階層の維持 (KEEP_HIERARCHY) が no に設定されている場合、XST で階層を超えた複雑な DSP マクロおよび DSP48 チェーンを作成できます。これは ISE® Design Suite のデフォルト設定です。

ブロック RAM へのロジックのマッピング

デザインがターゲット デバイスに収まらない場合は、デザインの一部のロジックをブロック RAM に配置できます。

1. RTL 記述を別の階層ブロックのブロック RAM 内に入れます。
2. HDL コードまたは XCF ファイルのいずれかで、この階層ブロックに **BRAM へのロジックのマッピング (BRAM_MAP)** 制約を設定します。

XST では、ブロック RAM に配置するロジックを自動的に判断できないので注意してください。

ロジックを別のブロックにする場合、次の条件を満たす必要があります。

- ・ すべての出力がレジスタを介するようにする。
- ・ ブロックに含めることのできるレジスタのレベルは 1 つで、これらは出力レジスタとする。
- ・ すべての出力レジスタが同じ制御信号を持つ。
- ・ 出力レジスタが同期リセット信号を持つ。
- ・ ブロックに複数のソースまたはトライステート バスが含まれない。
- ・ 中間信号に **キープ (KEEP)** 制約を使用できない。

ブロック RAM へのロジックのマッピングは、アドバンス合成段階で実行されます。上記の条件が 1 つでも満たされない場合は、ロジックはブロック RAM にマッピングされず、警告メッセージとその理由が示されます。ロジックが 1 つのブロック RAM プリミティブに配置できない場合は、複数のブロック RAM が使用されます。

ブロック RAM へのロジックのマップのログ ファイル例 1

```

...
=====
*                               HDL Synthesis                               *
=====
e is "bram_map_1.vhd".
    Found 4-bit register for signal <RES>.
    Found 4-bit adder for signal <$n0001> created at line 29.
    Summary:
        inferred    4 D-type flip-flop(s).
        inferred    1 Adder/Subtractor(s).
Unit <logic_bram_1> synthesized.

=====
*                               Advanced HDL Synthesis                       *
=====
...
Entity <logic_bram_1> mapped on BRAM.
...
=====
HDL Synthesis Report

Macro Statistics
# Block RAMs                : 1
  256x4-bit single-port block RAM : 1

```

ブロック RAM へのロジックのマップのログ ファイル例 2

```

...
=====
*                               Advanced HDL Synthesis                       *
=====
...
INFO:Xst:1789 - Unable to map block <no_logic_bram> on BRAM.
               Output FF <RES> must have a synchronous reset.

```

単一ブロック RAM プリミティブ内の定数付き 8 ビット加算器の VHDL コード例

```
--
-- The following example places 8-bit adders with
-- constant in a single block RAM primitive
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity logic_bram_1 is
port (clk, rst : in std_logic;
      A,B : in unsigned (3 downto 0);
      RES : out unsigned (3 downto 0));

    attribute bram_map: string;
    attribute bram_map of logic_bram_1: entity is "yes";

end logic_bram_1;

architecture beh of logic_bram_1 is
begin

    process (clk)
    begin
        if (clk'event and clk='1') then
            if (rst = '1') then
                RES <= "0000";
            else
                RES <= A + B + "0001";
            end if;
        end if;
    end process;

end beh;
```

単一ブロック RAM プリミティブ内の定数付き 8 ビット加算器の Verilog コード例

```
//
// The following example places 8-bit adders with
// constant in a single block RAM primitive
//

(* bram_map="yes" *)
module v_logic_bram_1 (clk, rst, A, B, RES);

    input  clk, rst;
    input  [3:0] A, B;
    output [3:0] RES;
    reg     [3:0] RES;

    always @(posedge clk)
    begin
        if (rst)
            RES <= 4'b0000;
        else
            RES <= A + B + 8'b0001;
        end
    endmodule
```

非同期リセットの VHDL コード例

```
--
-- In the following example, an asynchronous reset is used and
-- so, the logic is not mapped onto block RAM
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity logic_bram_2 is
port (clk, rst : in std_logic;
      A,B       : in unsigned (3 downto 0);
      RES       : out unsigned (3 downto 0));

    attribute bram_map : string;
    attribute bram_map of logic_bram_2 : entity is "yes";

end logic_bram_2;

architecture beh of logic_bram_2 is
begin

    process (clk, rst)
    begin
        if (rst='1') then
            RES <= "0000";
        elsif (clk'event and clk='1') then
            RES <= A + B + "0001";
        end if;
    end process;

end beh;
```

非同期リセットの Verilog コード例

```
//
// In the following example, an asynchronous reset is used and
// so, the logic is not mapped onto block RAM
//

(* bram_map="yes" *)
module v_logic_bram_2 (clk, rst, A, B, RES);

    input  clk, rst;
    input  [3:0] A, B;
    output [3:0] RES;
    reg     [3:0] RES;

    always @(posedge clk or posedge rst)
    begin
        if (rst)
            RES <= 4'b0000;
        else
            RES <= A + B + 8'b0001;
        end
    endmodule
```

フリップフロップのリタイミング

このセクションでは、フリップフロップのリタイミングについて説明します。

- ・ [フリップフロップのリタイミングの概要](#)
- ・ [グローバル最適化](#)
- ・ [フリップフロップのリタイミングのメッセージ](#)
- ・ [フリップフロップのリタイミングの制限](#)
- ・ [フリップフロップのリタイミングの制御方法](#)

フリップフロップのリタイミングの概要

フリップフロップのリタイミングとは、タイミングを向上してクロック周波数を上げるため、フリップフロップおよびラッチの位置を変更する手法です。

フリップフロップのリタイミングには順方向と逆方向があります。

- ・ 順方向のリタイミングでは、LUT の各入力に接続されたフリップフロップすべてを出力で 1 つのフリップフロップに移動します。
- ・ 逆方向のリタイミングでは、LUT の出力にある 1 つのフリップフロップを LUT の各入力に移動します。

フリップフロップのリタイミングをすると、次が発生することがあります。

- ・ フリップフロップの数がかなり増加します。
- ・ フリップフロップの一部が削除されます。

どちらが発生しても、デザインの動作に変更はなく、タイミング遅延のみが変化します。

グローバル最適化

フリップフロップのリタイミングはグローバル最適化の一部であり、ほかの最適化手法と同じ制約が考慮されます。リタイミングはインクリメンタル プロセスであり、リタイミングの結果挿入されたフリップフロップがタイミングを向上するために再び同じ方向（順方向または逆方向）に移動される場合があります。タイミング制約が満たされた場合、またはタイミングが向上しない場合は、それ以上のリタイミングは行われません。

フリップフロップのリタイミングのメッセージ

フリップフロップが移動されると、次を示すメッセージが表示されます。

- ・ 元のフリップフロップ名と新規のフリップフロップ名
- ・ そのフリップフロップのリタイミングが順方向と逆方向のどちらであるか

フリップフロップのリタイミングの制限

フリップフロップのリタイミングには、次のような制限があります。

- ・ IOB=TRUE プロパティが指定されたフリップフロップにはリタイミングは適用されません。
- ・ フリップフロップまたは出力信号に **キープ (KEEP)** プロパティが設定されている場合は、フリップフロップは順方向には移動されません。
- ・ 入力信号に **キープ (KEEP)** プロパティが設定されている場合は、フリップフロップは逆方向には移動されません。
- ・ インスタンシエートされたフリップフロップは、[\[Optimize Instantiated Primitives\]](#) 制約またはこのコマンドライン オプションが yes になっている場合にのみ移動されます。
- ・ フリップフロップは、[\[Optimize Instantiated Primitives\]](#) 制約またはこのコマンドライン オプションが yes になっている場合にのみインスタンシエートされたプリミティブ間で移動されます。
- ・ セットとリセットが付いたフリップフロップは移動されません。

フリップフロップのリタイミングの制御方法

フリップフロップのリタイミングを制御するには、次の制約を使用します。

- ・ [レジスタの自動調整 \(REGISTER_BALANCING\)](#)
- ・ [最初のフリップフロップ ステージの移動 \(MOVE_FIRST_STAGE\)](#)
- ・ [最後のフリップフロップ ステージの移動 \(MOVE_LAST_STAGE\)](#)

パーティション

XST では、インクリメンタル合成の代わりにパーティションがサポートされています。インクリメンタル合成はサポートされなくなりました。このため、incremental_synthesis および resynthesize 制約もサポートされません。パーティションの詳細は、ISE® Design Suite ヘルプを参照してください。

エリア制約を設定した場合のスピード最適化

XST ではエリア制約を設定したデザインでタイミング最適化を実行できます。このオプションは、次のように指定します。

- ・ スライス (LUT-FF ペア) の使用率
Virtex®-5 デバイス
- ・ [スライス \(LUT-FF ペア\) 使用率 \(SLICE_UTILIZATION_RATIO\)](#)
その他すべての FPGA デバイス

ISE® Design Suite で次のように定義します。

[Process] → [Process Properties] → [XST Synthesis Options] をクリックします。

デフォルトでは、選択したデバイス サイズの 100% に設定されています。

この制約は下位レベルの合成のみに適用され、推論プロセスには影響しません。

この制約を設定すると、まずエリアが概算され、指定のエリア制約が満たされてから、制約で指定した値を超えないようにタイミング最適化が行われます。デザインが要求よりも大きい場合は、まずエリアを削減し、エリア制約が満たされてからタイミング最適化が行われます。

例 1 (100%)

次の例では、エリア制約はデバイス サイズの 100% に設定されていますが、最初の概算ではデバイスの 102% を占めることが示されています。XST により最適化が実行され、95% に削減されています。

```
...
=====
*
*                               Low Level Synthesis
*
=====

Found area constraint ratio of 100 (+ 5) on block tge,
actual ratio is 102.
Optimizing block <tge> to meet ratio 100 (+ 5) of 1536
slices :
Area constraint is met for block <tge>, final ratio is 95.

=====
```

例 2 (70%)

エリア制約を満たすことができない場合、タイミング最適化の際にエリア制約は無視され、周波数が最大になるように下位レベルの合成が実行されます。次の例では、エリア制約が 70% に設定されていますが、この制約を満たすことができなかったため、警告メッセージが表示されています。

```
...
=====
*
*                               Low Level Synthesis
*
=====

Found area constraint ratio of 70 (+ 5) on block fpga_hm,
actual ratio is 64.
Optimizing block <fpga_hm> to meet ratio 70 (+ 5) of 1536
slices :
WARNING:Xst - Area constraint could not be met for block <tge>,
final ratio is 94
...
=====
...

```

メモ : (+5) は、エリア制約の最大マージンを示します。これは、エリア制約が満たされない場合、エリア最適化において制約と実際のエリアとの差が 5% 以下であれば、そのエリアを超えない範囲でタイミング最適化が実行されることを意味します。

例 3 (55%)

次の例では、エリア制約が 55% に設定されていますが、XST では 60% しか達成されていません。ただし、制約と実際のエリアの差が 5% なので、エリア制約は満たされたものとしてタイミング最適化が実行されます。

```
...
=====
*
*                               Low Level Synthesis
*
=====

Found area constraint ratio of 55 (+ 5) on block fpga_hm,
actual ratio is 64.
Optimizing block <fpga_hm> to meet ratio 55 (+ 5) of 1536
slices :
Area constraint is met for block <fpga_hm>, final ratio is 60.
=====
...
```

場合によっては自動リソース管理機能をオフにする必要があります。オフにするには、SLICE_UTILIZATION_RATIO の値を -1 に指定します。

スライス (LUT-FF ペア) 使用率 (SLICE_UTILIZATION_RATIO) は、デザインの特定ブロックに対して設定できます。合計スライス数のパーセントまたはスライスの絶対数で指定できます。

FPGA デバイス最適化のレポート セクション

このセクションでは、FPGA デバイスの最適化レポート セクションについて説明します。

- ・ [FPGA デバイス最適化のレポート セクション](#)
- ・ [セル使用率レポート](#)
- ・ [タイミング レポート](#)

FPGA デバイス最適化のレポート セクション

デザインの最適化中、次の事項がレポートされます。

- ・ 等価フリップフロップの削除
データ ピンと制御ピンが同じ場合、2 つのフリップフロップは等価であると判断されます。
- ・ レジスタの複製
レジスタ複製は、次の目的で使用されます。
 - タイミング パフォーマンスを改善するため
 - [MAX_FANOUT](#) 制約を満たすため[レジスタの複製 \(REGISTER_DUPLICATION\)](#) を使用すると、レジスタの複製がされないようにできます。

FPGA デバイス最適化のレポート セクションの例

```
Starting low level synthesis ...
Optimizing unit <down4cnt> ...
Optimizing unit <doc_readwrite> ...
...
Optimizing unit <doc> ...
Building and optimizing final netlist ...
The FF/Latch <doc_readwrite/state_D2> in Unit <doc> is
equivalent to the following 2 FFs/Latches,
which will be removed : <doc_readwrite/state_P2>
<doc_readwrite/state_M2>Register
doc_reset_I_reset_out has been replicated 2 time(s)
Register wr_1 has been replicated 2 time(s)
```

セル使用率レポート

「Final Report」(最終レポート)の「Cell Usage」(セル使用率)セクションには、デザインで使用されたプリミティブの合計数が示されます。プリミティブは、次のグループに分類されます。

- ・ BEL のセル使用率
- ・ フリップフロップとラッチのセル使用率
- ・ RAM のセル使用率
- ・ シフタのセル使用率
- ・ トライステートのセル使用率
- ・ クロック バッファのセル使用率
- ・ IO バッファのセル使用率
- ・ 論理セル使用率
- ・ その他のセル使用率

BEL のセル使用率

「Final Report」(最終レポート)の「Cell Usage」(セル使用率)セクションの BEL グループには、ターゲット FPGA デバイス ファミリの基本エレメントである次のようなロジック セルすべてが含まれます。

- ・ LUT
- ・ MUXCY
- ・ MUXF5
- ・ MUXF6
- ・ MUXF7
- ・ MUXF8

フリップフロップとラッチのセル使用率

「Final Report」(最終レポート)の「Cell Usage」(セル使用率)セクションのフリップフロップとラッチのグループには、ターゲット デバイス ファミリの基本エレメントである次のようなロジック セルすべてが含まれます。

- ・ FDR
- ・ FDRE
- ・ LD

RAM のセル使用率

「Final Report」(最終レポート)の「Cell Usage」(セル使用率)セクションの RAM グループには、RAM すべてが含まれます。

シフトのセル使用率

「Final Report」(最終レポート)の「Cell Usage」(セル使用率)セクションのシフトグループには、FPGA デバイス プリミティブを使用する次のようなシフトレジスタがすべて含まれます。

- ・ TSRL16
- ・ SRL16_1
- ・ SRL16E
- ・ SRL16E_1
- ・ SRLC

トライステートのセル使用率

「Final Report」(最終レポート)の「Cell Usage」(セル使用率)セクションのトライステートグループには、次のようなトライステートプリミティブがすべて含まれます。

BUFT

クロック バッファのセル使用率

「Final Report」(最終レポート)の「Cell Usage」(セル使用率)セクションのクロック バッファには、次のようなクロック バッファがすべて含まれます。

- ・ BUFG
- ・ BUFGP
- ・ BUFGDLL

IO バッファのセル使用率

「Final Report」(最終レポート)の「Cell Usage」(セル使用率)セクションの I/O バッファには、次のようなクロック バッファ以外の I/O バッファがすべて含まれます。

- ・ IBUF
- ・ OBUF
- ・ IOBUF
- ・ OBUFT
- ・ IBUF_GTL ...

論理セル使用率

「Final Report」(最終レポート)の「Cell Usage」(セル使用率)セクションの論理 (LOGICAL) グループには、基本エレメント以外のすべての論理セル プリミティブが含まれます。

- ・ AND2
- ・ OR2 ...

その他のセル使用率

「Final Report」(最終レポート)の「Cell Usage」(セル使用率)セクションのその他 (OTHER) グループには、上記のグループに属さないすべてのセルが含まれます。

セル使用率のレポート例

```

=====
...
Cell Usage :
# BELS                                : 70
#      LUT2                           : 34
#      LUT3                           : 3
#      LUT4                           : 34
# FlipFlops/Latches                   : 9
#      FDC                            : 8
#      FDP                            : 1
# Clock Buffers                       : 1
#      BUFGP                          : 1
# IO Buffers                          : 24
#      IBUF                           : 16
#      OBUF                           : 8
=====

```

XST により予測されたスライス、フリップフロップ、IOB、BRAM などの数が示されます。このレポートは、MAP で生成されるレポートと類似しています。

短い表には、次の情報が含まれます。

- ・ デザインのクロック数、各クロックのバッファ方法、ロード数
- ・ デザインに含まれる非同期セット/リセット信号の数、各信号のバッファ方法、ロード数

タイミング レポート

このセクションでは、タイミング レポートについて説明します。

- ・ [タイミング レポートの概要](#)
- ・ [タイミング レポートのタイミング サマリ セクション](#)
- ・ [タイミング レポートの Detail セクション](#)
- ・ [タイミング レポートのパスとポート](#)

タイミング レポートの概要

XST では合成の最後に詳細なタイミング情報がレポートされます。タイミング レポートには、ネットリストの 4 つの使用可能なドメインに関する情報が表示されます。

- ・ レジスタからレジスタ
- ・ 入力からレジスタ
- ・ レジスタから出力パッド
- ・ 入力パッドから出力パッド

タイミング レポートの例

これらのタイミングの値は、あくまで合成での概算にすぎないので、正確なタイミング情報は配置配線後の TRACE レポートを参照してください。

Clock Information:

```
-----
```

Clock Signal	Clock buffer (FF name)	Load
CLK	BUFGP	11

```
-----
```

Asynchronous Control Signals Information:

```
-----
```

Control Signal	Buffer (FF name)	Load
rstint (MACHINE/current_state_Out01:0)	NONE (sixty/lsbcount/qoutsig_3)	4
RESET	IBUF	3
sixty/msbclr (sixty/msbclr:0)	NONE (sixty/msbcount/qoutsig_3)	4

```
-----
```

Timing Summary:

Speed Grade: -12

Minimum period: 2.644ns (Maximum Frequency: 378.165MHz)

Minimum input arrival time before clock: 2.148ns

Maximum output required time after clock: 4.803ns

Maximum combinational path delay: 4.473ns

Timing Detail:

All values displayed in nanoseconds (ns)

Timing constraint: Default period analysis for Clock 'CLK'

Clock period: 2.644ns (frequency: 378.165MHz)

Total number of paths / destination ports: 77 / 11

```
-----
```

Delay: 2.644ns (Levels of Logic = 3)

Source: MACHINE/current_state_FFd3 (FF)

Destination: sixty/msbcount/qoutsig_3 (FF)

Source Clock: CLK rising

Destination Clock: CLK rising

Data Path: MACHINE/current_state_FFd3 to sixty/msbcount/qoutsig_3

Cell:in->out	fanout	Gate Net		Logical Name (Net Name)
		Delay	Delay	
FDC:C->Q	8	0.272	0.642	MACHINE/current_state_FFd3 (MACHINE/current_state_FFd3)
LUT3:I0->O	3	0.147	0.541	Ker81 (clkenable)
LUT4_D:I1->O	1	0.147	0.451	sixty/msbce (sixty/msbce)
LUT3:I2->O	1	0.147	0.000	sixty/msbcount/qoutsig_3_rstpot (N43)
FDC:D		0.297		sixty/msbcount/qoutsig_3

```
-----
Total                2.644ns (1.010ns logic, 1.634ns route)
                      (38.2% logic, 61.8% route)
```

タイミング レポートのタイミング サマリ セクション

このセクションには、4 つのドメインすべてのタイミング パスに関するサマリ情報が示されます。

- ・ クロックからクロックまでのパス

Minimum period: 7.523ns (Maximum Frequency: 132.926MHz)

- ・ すべてのプライマリ出力からシーケンシャル エlementまでの最大パス

Minimum input arrival time before clock: 8.945ns

- ・ シーケンシャル エlementからすべてのプライマリ出力までの最大パス

Maximum output required time before clock: 14.220ns

- ・ 入力から出力までの最大パス

Maximum combinational path delay: 10.899ns

ドメインに該当するパスがない場合は、「No path found」と記述されます。

タイミング レポートの Detail セクション

このセクションには、次の各領域で最もクリティカルなパスに関する情報が詳細に記述されます。

- ・ パスの開始点
- ・ パスの終点
- ・ パスの最大遅延
- ・ スラック

始点と終点は、次のいずれかになります。

- ・ クロック (立ち上がり/立ち下りのパルス付き)
- ・ ポート

sysclk クロックの立ち上がりから sysclk クロックの立ち上がりまでのパス : 7.523ns (スラック : -7.523ns)

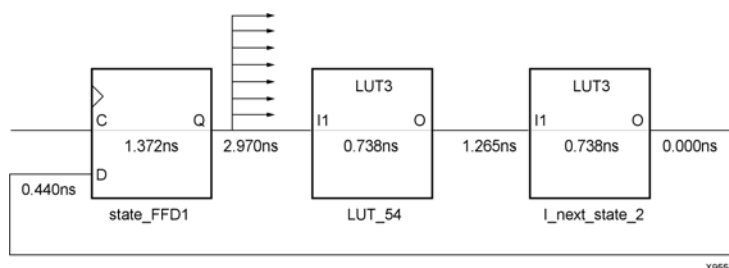
パスの詳細には、次の情報が含まれます。

- ・ セル タイプ
- ・ このゲートの入力と出力
- ・ 出力のファンアウト
- ・ ゲート遅延
- ・ ネット遅延の概算
- ・ インスタンス名

階層ブロックの始めには「begin scope」と記述され、ブロックの終わりには「end scope」と記述されます。

前述のレポートは、次の回路図に対応しています。

タイミング レポートの回路図



タイミング レポートのパスとポート

また、タイミング レポートのセクションでは、解析されたパス数およびポート数が示されます。XST の実行時にタイミング制約が含まれる場合は、エラーが発生したパス数およびポート数も表示されます。エラーが発生したパス数は、デザインに含まれるタイミング エラー数を示し、エラーが発生したポート数は、これらがデザインでどのように配置されているかを示します。タイミング レポートのポート数は、タイミング制約のデスティネーション エLEMENT数を示します。

たとえば、次のタイミング制約を使用するとします。

```
TIMESPEC "TSidentifier"=FROM "source_group" TO "dest_group" value
units;
```

この場合、ポート数はデスティネーション グループに含まれるELEMENT数に対応します。

XST でエラーが発生したパス数が 100 であると表示されているのに、エラーが発生したデスティネーション ポートはフリップフロップ 2 個のみであるという場合があります。この場合、これらの 2 個フリップフロップに関する記述を解析するだけで十分です。

インプリメンテーション制約

XST は、Hardware Description Language (HDL) または制約ファイルの属性 (LOC など) から生成されたインプリメンテーション制約を NGC ファイルに出力します。

キープ (KEEP) プロパティは、最大ファンアウトの制御または最適化を目的として、バッファ挿入プロセスにより生成されます。

FPGA デバイス プリミティブのサポート

このセクションでは、FPGA デバイス プリミティブのサポートについて説明します。

- ・ [FPGA デバイス プリミティブのサポートの概要](#)
- ・ [属性を使用したプリミティブの生成](#)
- ・ [プリミティブとブラック ボックス](#)
- ・ [VHDL および Verilog のデバイス プリミティブ ライブラリ](#)
- ・ [インスタンス化されたデバイス プリミティブのレポート](#)
- ・ [プリミティブの関連制約](#)
- ・ [プリミティブのコード例](#)
- ・ [UniMacro ライブラリの使用](#)

FPGA デバイス プリミティブのサポートの概要

XST では、デバイスのプリミティブを VHDL/Verilog コードに直接インスタンス化できます。次のようなプリミティブは、インスタンス化すると HDL デザインに手動で挿入できます。

- ・ MUXCY_L
- ・ LUT4_L
- ・ CLKDLL
- ・ RAMB4_S1_S16
- ・ IBUFG_PCI33_5
- ・ NAND3b2

これらのプリミティブは、次のようになります。

- ・ UNISIM ライブラリでコンパイルされます。
- ・ デフォルトでは XST で最適化されません。
- ・ 最終的な NGC ファイルに記述されます。

[Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックスで [Xilinx Specific Options] ページにある [\[Optimize Instantiated Primitives\]](#) をオンにすると、インスタンス化したプリミティブを最適化して結果を向上させることができます。ほとんどのプリミティブにタイミング情報があり、XST で効果的なタイミングドリブン最適化が実行されます。

XST では、RAM のような複雑なプリミティブのインスタンス化を簡単にするために、UniMacro という別のライブラリもサポートされています。

詳細は、[Libraries Guides](#)を参照してください。

属性を使用したプリミティブの生成

属性により生成できるプリミティブもあります。

- ・ **バッファ タイプ (BUFFER_TYPE)**

プライマリ入力または内部信号に属性を設定すると、次を強制的に使用できます。

- BUFGDLL
- IBUFG
- BUFR
- BUFGP

同じ制約を使用してバッファの挿入をディスエーブルにすることもできます。

- ・ **I/O 規格 (IOSTANDARD)**

I/O プリミティブに I/O 規格を指定できます。

この例では、I/O ポートに PCI33_5 I/O 規格を指定しています。

```
// synthesis attribute IOSTANDARD of in1 is PCI33_5
```

プリミティブとブラック ボックス

プリミティブのサポートは、ブラック ボックスの概念に基づいています。ブラック ボックスの詳細は、「[セーフ FSM インプリメンテーション](#)」を参照してください。

ブラック ボックスのサポートとプリミティブのサポートは大きく異なります。たとえば、デザインに MUXF5 というサブモジュールが含まれているとします。MUXF5 は、ユーザーのファンクション ブロックである場合とザイリンクス デバイス プリミティブである場合とがあります。XST でのこのモジュールの処理において混乱が生じないようにするため、**ボックス タイプ (BOX_TYPE)** 制約を MUXF5 のコンポーネント宣言に設定する必要があります。

ボックス タイプ (BOX_TYPE)を MUXF5 に設定する場合、次の値を使用します。

- ・ primitive または black_box

そのモジュールはザイリンクス デバイス プリミティブとして処理され、クリティカル パスの概算などにこのプリミティブのパラメータが使用されます。

- ・ user_black_box

モジュールはブラック ボックスとして処理されます。

ブラック ボックスとザイリンクス デバイス プリミティブの名前が同じ場合は、XST により固有の名前に変更され、警告メッセージが表示されます。たとえば次のログ ファイル例では、MUX5 が MUX51 に変更されています。

```
...
=====
*                               Low Level Synthesis                               *
=====

WARNING:Xst:79 - Model 'muxf5' has different characteristics in
destination library
WARNING:Xst:80 - Model name has been changed to 'muxf51'
...
```

MUXF5 に **ボックス タイプ (BOX_TYPE)** を設定しない場合、このモジュールはユーザー階層ブロックとして処理されます。ブラック ボックスとザイリンクス デバイス プリミティブの名前が同じ場合は、XST により固有の名前に変更され、警告メッセージが表示されます。

VHDL および Verilog のデバイス プリミティブ ライブラリ

XST では、HDL コードでザイリンクス デバイス プリミティブのインスタンス化をシンプルにするため、VHDL および Verilog の両方の専用ライブラリが提供されています。これらのライブラリにはザイリンクス デバイス プリミティブの宣言がすべて含まれ、各コンポーネントに **ボックス タイプ (BOX_TYPE)** 制約が設定されています。

デバイス ライブラリ

VHDL の場合、ソース コードでパッケージ vcomponents を使用して UNISIM ライブラリを宣言します。

```
library unisim;  
use unisim.vcomponents.all;
```

このパッケージのソース コードは、XST インストール ディレクトリにある次のファイルに含まれています。

```
vhdl\src\ unisims\unisims_vcomp.vhd
```

Verilog の場合、UNISIM ライブラリがあらかじめコンパイルされています。このライブラリは自動的にデザインにリンクされません。

プリミティブ インスタンス化のガイドライン

プリミティブをインスタンス化するには、ジェネリック (VHDL) およびパラメータ (Verilog) に大文字を使用してください。たとえば、ODDR エLEMENT は UNISIM ライブラリで次のように宣言されています。

```
component ODDR  
generic  
  (DDR_CLK_EDGE : string := "OPPOSITE_EDGE";  
   INIT : bit := '0';  
   SRTYPE : string := "SYNC");  
  
port(Q : out std_ulogic;  
     C : in std_ulogic;  
     CE : in std_ulogic;  
     D1 : in std_ulogic;  
     D2 : in std_ulogic;  
     R : in std_ulogic;  
     S : in std_ulogic);  
end component;
```

このプリミティブをインスタンス化する場合、DDR_CLK_EDGE および SRTYPE ジェネリックの値は大文字にする必要があります。大文字にしないと、XST で不明の値が使用されていることを示す警告メッセージが表示されます。

LUT1 のようなプリミティブでは、インスタンス化で INIT を使用できます。INIT を最終ネットリストに渡すには、次の 2 つの方法があります。

- ・ INIT 属性をインスタンス化したプリミティブに設定します。
- ・ VHDL のジェネリックまたは Verilog のパラメータを使用して INIT を渡します。このようにすると合成とシミュレーションに同じコードを使用できるので、ザイリンクスではこの方法をお勧めしています。

インスタンス化されたデバイス プリミティブのレポート

UNISIM ライブラリの各プリミティブでは、**ボックス タイプ (BOX_TYPE)** 属性が primitive に設定されているため、HDL 合成中にインスタンス化されたデバイス プリミティブに関するメッセージは表示されません。

XST では、次の場合にログ ファイル例に示すような警告メッセージが表示されます。

- ・ ブロック (プリミティブ以外) をインスタンス化し、
さらに
- ・ そのブロックに内容がない場合 (ロジック記述なし)
または
- ・ ブロックにロジック記述があり、
さらに
- ・ **ボックス タイプ (BOX_TYPE)** を user_black_box で適用した場合

ログ ファイルの例

```
...
Analyzing Entity <black_b> (Architecture <archi>).
WARNING : (VHDL_0103). c:\jm\des.vhd (Line 23).
Generating a Black Box for component <my_block>.
Entity <black_b> analyzed. Unit <black_b> generated.
...
```

プリミティブの関連制約

- ・ **ボックス タイプ (BOX_TYPE)**
- ・ 処理せずに HDL から NGC に渡される PAR の制約

プリミティブのコード例

コード例は、ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

VHDL コード例 (INIT 制約で INIT 値指定)

```
--
-- Passing an INIT value via the INIT constraint.
--

library ieee;
use ieee.std_logic_1164.all;

library unisim;
use unisim.vcomponents.all;

entity primitive_1 is
    port(I0,I1 : in std_logic;
         O      : out std_logic);
end primitive_1;

architecture beh of primitive_1 is

    attribute INIT: string;
    attribute INIT of inst: label is "1";

begin

    inst: LUT2 port map (I0=>I0,I1=>I1,O=>O);

end beh;
```

Verilog コード例 (INIT 制約で INIT 値指定)

```
//
// Passing an INIT value via the INIT constraint.
//

module v_primitive_1 (I0,I1,O);
    input I0,I1;
    output O;

    (* INIT="1" *)
    LUT2 inst (.I0(I0), .I1(I1), .O(O));

endmodule
```

VHDL コード例 (ジェネリック文で INIT 値指定)

```
--  
-- Passing an INIT value via the generics mechanism.  
--  
  
library ieee;  
use ieee.std_logic_1164.all;  
  
library unisim;  
use unisim.vcomponents.all;  
  
entity primitive_2 is  
    port(I0,I1 : in std_logic;  
          O    : out std_logic);  
end primitive_2;  
  
architecture beh of primitive_2 is  
begin  
  
    inst: LUT2 generic map (INIT=>"1")  
        port map (I0=>I0,I1=>I1,O=>O);  
  
end beh;
```

Verilog コード例 (パラメータ文で INIT 値指定)

```
//  
// Passing an INIT value via the parameters mechanism.  
//  
  
module v_primitive_2 (I0,I1,O);  
    input I0,I1;  
    output O;  
  
    LUT2 #(4'h1) inst (.I0(I0), .I1(I1), .O(O));  
  
endmodule
```

Verilog コード例 (defparam で INIT 値指定)

```
//  
// Passing an INIT value via the defparam mechanism.  
//  
  
module v_primitive_3 (I0,I1,O);  
    input I0,I1;  
    output O;  
  
    LUT2 inst (.I0(I0), .I1(I1), .O(O));  
    defparam inst.INIT = 4'h1;  
  
endmodule
```

UniMacro ライブラリの使用

このセクションでは、UniMacro ライブラリの使用について説明します。

- ・ [UniMacro ライブラリの使用の概要](#)
- ・ [UniMacro ライブラリのデバイス サポート](#)
- ・ [VHDL での UniMacro ライブラリの使用](#)
- ・ [Verilog での UniMacro ライブラリの使用](#)

UniMacro ライブラリの使用の概要

XST では、RAM のような複雑なプリミティブのインスタンス化を簡単にするために、UniMacro という別のライブラリもサポートされています。

詳細は、[Libraries Guides](#)を参照してください。

UniMacro ライブラリのデバイス サポート

UniMacro ライブラリでは、次のデバイスがサポートされます。

- ・ Virtex®-4
- ・ Virtex-5 およびそれ以降のデバイス

VHDL での UniMacro ライブラリの使用

VHDL の場合、ソースコードでパッケージ vcomponents を使用してライブラリ unimacro ライブラリを宣言します。

```
library unimacro;  
use unimacro.vcomponents.all;
```

このパッケージのソースコードは、次の XST インストール ディレクトリに含まれています。

```
vhdl\src\unisims\unisims_vcomp.vhd
```

Verilog での UniMacro ライブラリの使用

Verilog の場合、UniMacro ライブラリがあらかじめコンパイルされています。このライブラリは自動的にデザインにリンクされません。

コアの処理

デザインに Electronic Data Interchange Format (EDIF) または NGC ファイルで記述されたコアが含まれる場合、それらのファイルは自動的に読み込まれ、タイミングの概算およびエリア使用率の制御に使用されます。この機能は、ISE® Design Suite で [Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックスを表示し、[Synthesis Options] ページにある [Read Cores] でオン/オフを切り替えます。コマンドラインの場合、run コマンドで `-read_cores` オプションを使用します。optimize オプションも指定できます。このオプションを使用すると、コアの処理が可能になり、コアのネットリストをデザインに統合できます。XST では、デフォルトでコアが読み込まれます。

[Read Cores] がオフの場合、組み合わせパスの最大遅延は 6.639ns (クリティカル パスは単純な AND ファンクションを通過)、使用されるエリアは 1 スライスと予測されます。

[Read Cores] がオンの場合、下位レベルの合成時に次のメッセージが表示されます。

```
...
=====
*
*                               Low Level Synthesis
*
=====

Launcher: Executing edif2ngd -noa "my_add.edn" "my_add.ngo"
INFO:NgdBuild - Release 6.1i - edif2ngd G.21
INFO:NgdBuild - Copyright (c) 1995-2003 Xilinx, Inc.
All rights reserved.
Writing the design to "my_add.ngo"...
Loading core <my_add> for timing and area information
for instance <inst>.

=====
...
```

この場合、組み合わせパスの最大遅延は 8.281ns、使用されるエリアは 5 スライスと予測されます。

デフォルトでは、コアの Electronic Data Interchange Format (EDIF) または NGC ファイルは、作業中のプロジェクトのディレクトリから読み込まれます。コア ファイルがプロジェクト ディレクトリにない場合は、[コアの検索ディレクトリ \(-sd\)](#) オプションでコアのディレクトリを指定する必要があります。

コード例

次の VHDL の例では、my_add ブロックはブラック ボックスとして記述されている加算器で、このデザインのネットリストは CORE Generator™ で生成されています。

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

entity read_cores is
  port (
    A, B : in std_logic_vector (7 downto 0);
    a1, b1 : in std_logic;
    SUM : out std_logic_vector (7 downto 0);
    res : out std_logic);
end read_cores;

architecture beh of read_cores is
  component my_add
    port (
      A, B : in std_logic_vector (7 downto 0);
      S : out std_logic_vector (7 downto 0));
  end component;

begin
  res <= a1 and b1;
  inst: my_add port map (A => A, B => B, S => SUM);
end beh;
```

INIT および RLOC の指定

UNISIM ライブラリを使用すると、LUT コンポーネントを直接 HDL コードにインスタンス化できます。LUT のファンクションを指定するには、LUT のインスタンスに INIT 制約を設定します。インスタンス化した LUT またはレジスタをチップの特定スライスに配置する場合は、インスタンスに RLOC 制約を設定します。

INIT でファンクションを定義するのが不都合な場合は、ファンクションを個別のブロックとして VHDL または Verilog で記述し、LUT にマップする方法もあります。

このブロックに 単一 LUT へのエンティティのマップ (LUT_MAP) 制約を設定すると、このブロックが 1 つの LUT にマップされます。LUT の INIT 値は XST により自動的に算出され、最適化中この LUT が保持されます。

XST では、Synopsys でサポートされる XC_MAP 制約が自動的に認識されます。

LUT_MAP 制約を使用して INIT 値を渡すコード例

コード例は、http://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

次に、LUT_MAP 制約を使用して INIT 値を渡す例を示します。

これらの例では、top ブロックに and_one および and_two で記述される 2 つの AND ゲートがインスタンス化されています。XST では 2 つの LUT2 が生成され、1 つに結合されることはありません。

詳細は、次を参照してください。

単一 LUT へのエンティティのマップ (LUT_MAP)

LUT_MAP 制約を使用して INIT 値を渡すVHDL コード例

```
--
-- Mapping on LUTs via LUT_MAP constraint
--

library ieee;
use ieee.std_logic_1164.all;
entity and_one is
    port (A, B : in std_logic;
          REZ : out std_logic);

    attribute LUT_MAP: string;
    attribute LUT_MAP of and_one: entity is "yes";
end and_one;

architecture beh of and_one is
begin
    REZ <= A and B;
end beh;
```

```
library ieee;
use ieee.std_logic_1164.all;
entity and_two is
    port(A, B : in std_logic;
         REZ : out std_logic);

    attribute LUT_MAP: string;
    attribute LUT_MAP of and_two: entity is "yes";
end and_two;

architecture beh of and_two is
begin
    REZ <= A or B;
end beh;

-----

library ieee;
use ieee.std_logic_1164.all;
entity inits_rlocs_1 is
    port(A,B,C : in std_logic;
         REZ : out std_logic);
end inits_rlocs_1;

architecture beh of inits_rlocs_1 is

    component and_one
    port(A, B : in std_logic;
         REZ : out std_logic);
    end component;

    component and_two
    port(A, B : in std_logic;
         REZ : out std_logic);
    end component;

    signal tmp: std_logic;
begin
    inst_and_one: and_one port map (A => A, B => B, REZ => tmp);
    inst_and_two: and_two port map (A => tmp, B => C, REZ => REZ);
end beh;
```

LUT_MAP 制約を使用して INIT 値を渡す Verilog コード例

```
//
// Mapping on LUTs via LUT_MAP constraint
//

(* LUT_MAP="yes" *)
module v_and_one (A, B, REZ);
    input A, B;
    output REZ;

    and and_inst (REZ, A, B);

endmodule

// -----

(* LUT_MAP="yes" *)
module v_and_two (A, B, REZ);
    input A, B;
    output REZ;

    or or_inst (REZ, A, B);

endmodule

// -----

module v_inits_rlocs_1 (A, B, C, REZ);
    input A, B, C;
    output REZ;

    wire tmp;

    v_and_one inst_and_one (A, B, tmp);
    v_and_two inst_and_two (tmp, C, REZ);

endmodule
```

フリップフロップの INIT 値を指定するコード例

ファンクションが 1 つの LUT にマップできない場合、エラー メッセージが表示され、合成が停止します。RTL レベルでフリップフロップまたはシフトレジスタの INIT 値を定義する場合は、信号宣言の段階で初期値を指定します。この値が合成中に無視されることなく、フリップフロップまたはシフトレジスタに INIT 制約が設定されて、最終ネットリストに含まれます。

次の例では、信号 tmp に対して 4 ビットレジスタが推論されます。

推論されたレジスタには INIT 値 1011 が設定され、最終ネットリストに含まれます。

フリップフロップの INIT 値を指定した VHDL コード例

```
--
-- Specification on an INIT value for a flip-flop,
-- described at RTL level
--

library ieee;
use ieee.std_logic_1164.all;

entity inits_rlocs_2 is
    port (CLK : in std_logic;
          DI  : in std_logic_vector(3 downto 0);
          DO  : out std_logic_vector(3 downto 0));
end inits_rlocs_2;

architecture beh of inits_rlocs_2 is signal
    tmp: std_logic_vector(3 downto 0):="1011";
begin

    process (CLK)
    begin
        if (clk'event and clk='1') then
            tmp <= DI;
        end if;
    end process;

    DO <= tmp;

end beh;
```

フリップフロップの INIT 値を指定する Verilog コード例

```
//  
// Specification on an INIT value for a flip-flop,  
// described at RTL level  
//  
  
module v_inits_rlocs_2 (clk, di, do);  
    input  clk;  
    input  [3:0] di;  
    output [3:0] do;  
    reg    [3:0] tmp;  
  
    initial begin  
        tmp = 4'b1011;  
    end  
  
    always @(posedge clk)  
    begin  
        tmp <= di;  
    end  
  
    assign do = tmp;  
  
endmodule
```

フリップフロップの INIT 値および RLOC 値を指定するコード例

レジスタが推論され、チップの特定の位置に配置されるようにするには、次のコード例のように tmp 信号に **RLOC** 制約を設定します。

XST では、この制約を最終ネットリストに含めます。XST では、次でこの機能がサポートされます。

- ・ レジスタ、および
- ・ 1 つのブロック RAM プリミティブにインプリメント可能であれば、推論済みブロック RAM

フリップフロップの INIT 値および RLOC 値を指定する VHDL コード例

```
--
-- Specification on an INIT and RLOC values for a flip-flop,
-- described at RTL level
--

library ieee;
use ieee.std_logic_1164.all;

entity inits_rlocs_3 is
    port (CLK : in std_logic;
          DI : in std_logic_vector(3 downto 0);
          DO : out std_logic_vector(3 downto 0));
end inits_rlocs_3;

architecture beh of inits_rlocs_3 is
    signal tmp: std_logic_vector(3 downto 0):="1011";

    attribute RLOC: string;
    attribute RLOC of tmp: signal is "X3Y0 X2Y0 X1Y0 X0Y0";
begin

    process (CLK)
    begin
        if (clk'event and clk='1') then
            tmp <= DI;
        end if;
    end process;

    DO <= tmp;

end beh;
```

フリップフロップの INIT 値を指定した Verilog コード例

```
//  
// Specification on an INIT and RLOC values for a flip-flop,  
// described at RTL level  
//  
  
module v_inits_rlocs_3 (clk, di, do);  
    input  clk;  
    input  [3:0] di;  
    output [3:0] do;  
    (* RLOC="X3Y0 X2Y0 X1Y0 X0Y0" *)  
    reg    [3:0] tmp;  
  
    initial begin  
        tmp = 4'b1011;  
    end  
  
    always @(posedge clk)  
    begin  
        tmp <= di;  
    end  
  
    assign do = tmp;  
  
endmodule
```

XST での PCI フローの使用

このセクションでは、XST で PCI™ フローを使用する方法について説明します。

- ・ [XST での PCI フローの使用規則](#)
- ・ [ロジックとフリップフロップの複製の回避](#)
- ・ [コアの自動読み込み機能のオフ](#)

XST での PCI フローの使用規則

次の規則に従うと、XST を使用した PCI フローで配置制約およびタイミング制約をすべて満たすには、次のオプションを設定できます。

- ・ VHDL デザインでは、生成されたネットリスト内の名前をすべて大文字にする必要があります。

デフォルトでは、小文字になっています。

ISE® Design Suite では大文字/小文字を次から設定します。

[Process] → [Process Properties] → [Synthesis Options] → [Case]

- ・ Verilog デザインでは、[Case] が [Maintain] に設定されていることを確認してください。

デフォルトでは、[Maintain] になっています。

ISE Design Suite では大文字/小文字を次から設定します。

[Process] → [Process Properties] → [Synthesis Options] → [Case]

- ・ デザインの階層を保持します。

[階層の維持 \(KEEP_HIERARCHY\)](#) は、ISE Design Suite で次から指定します。

[Process] → [Process Properties] → [Synthesis Options] → [Keep Hierarchy]

- ・ 等価フリップフロップを保持します。

XST では、等価フリップフロップがデフォルトで削除されます。

[等価レジスタの削除 \(EQUIVALENT_REGISTER_REMOVAL\)](#) は、ISE Design Suite で次から指定します。

[Process] → [Process Properties] → [Synthesis Options] → [Equivalent Register Removal]

ロジックとフリップフロップの複製の回避

フリップフロップのセット/リセット信号のファンアウトが多いとフリップフロップが複製されますが、この複製が行われないようにするには、次いずれかの操作を行います。

- ・ ISE® Design Suite で次を変更して、デザイン全体の最大ファンアウト値を大きい値に設定します。

[Process] → [Process Properties] → [Synthesis Options] → [Max Fanout]、または

- ・ [最大ファンアウト \(MAX_FANOUT\)](#) 属性を使用して、PCI コアの RST ポートに接続されている初期化信号に高いファンアウト値を設定します。

例 :

max_fanout=2048

コアの自動読み込み機能のオフ

[Read Cores] をオフにすると、タイミングおよびエリア予測のため PCI™ コアが自動的に読み込まれないようにできます。PCI コアが読み込まれる場合は、タイミング要件を満たさないロジックや、マップ中にエラーになるようなロジックに対して最適化が実行されます。デフォルトでは、タイミングおよびエリア予測のためコアが自動的に読み込まれます。[Read Cores] を使用しないようにするには、次のチェック ボックスをオフにします。

[Process] → [Process Properties] → [Synthesis Options] → [Read Cores]

CPLD の最適化

この章には、次の内容が含まれます。

- ・ [CPLD 合成オプション](#)
- ・ [マクロ生成のインプリメンテーション](#)
- ・ [CPLD 合成のログ ファイルの解析](#)
- ・ [CPLD 合成の制約](#)
- ・ [CPLD 合成結果の向上](#)

CPLD 合成オプション

このセクションでは、ISE® Design Suite の次のダイアログ ボックスから設定できる CPLD 合成に関する XST オプションのみについて説明します。

[Process] → [Process Properties]

XST では、CPLD フィッタ用に NGC ファイルが生成されます。

CPLD を合成するための一般的な XST フローは、次のとおりです。

1. [VHDL] または [Verilog] を選択します。
2. マクロの推論
3. モジュールの最適化
4. NGC ファイルの生成

CPLD 合成でサポートされるデバイス

XST でサポートされている CPLD デバイスは次の 5 つです。

- ・ CoolRunner™ XPLA3
- ・ CoolRunner-II
- ・ XC9500
- ・ XC9500XL

CoolRunner XPLA3 および XC9500XL デバイス ファミリの合成では、クロック イネーブルが処理されます。クロック イネーブルは、許可するか無効にできます。無効にした場合は、同等のロジックに置換されます。

また、カウンタなどのクロック イネーブルを使用するマクロが選択可能かどうかは、デバイス ファミリによって異なります。クロック イネーブルを使用するカウンタは、CoolRunner XPLA3、XC9500XL ファミリでは選択できますが、XC9500 ファミリでは使用不可で、同等のロジックに置き換えられます。

CPLD 合成オプションの設定

CPLD 合成オプションは、ISE® Design Suite から設定できます。

[Process] → [Process Properties] → [Synthesis Options]

- ・ 階層の維持 (KEEP_HIERARCHY)
- ・ マクロの保持 (-pld_mp)
- ・ XOR の保持 (-pld_xp)
- ・ 等価レジスタの削除 (EQUIVALENT_REGISTER_REMOVAL)
- ・ クロック イネーブル (-pld_ce)
- ・ WYSIWYG (-wysiwyg)
- ・ 削減なし (NOREDUCE)

詳細は、次を参照してください。

[CPLD 制約 \(タイミング以外\)](#)

マクロ生成のインプリメンテーション

XST では次のマクロが処理されます。

- ・ 加算器
- ・ 減算器
- ・ 加減算器
- ・ 乗算器
- ・ コンパレータ
- ・ マルチプレクサ
- ・ カウンタ
- ・ 論理シフタ
- ・ レジスタ (フリップフロップおよびラッチ)
- ・ XOR

マクロの生成は、[Macro Preserve] オプションで設定します。このオプションには、次の 2 つの値があります。

- ・ **yes**
マクロ生成が許可されます。
- ・ **no**
マクロ生成が禁止されます。

一般的なマクロ生成のフローは次のとおりです。

1. Hardware Description Language (HDL) ではマクロが推論され、下位合成に渡されます。
2. 下位合成で、マクロのインプリメンテーションに必要なリソースによって、マクロとして処理されるかどうかが決まります。

マクロとして処理されるものは、内部マクロ生成機能で生成されます。マクロとして処理されないものは、HDL 合成で等価ロジックに置き換えられるか、またはコンポーネントブロックに分解され、コンポーネントが最初のマクロと比較して使用するリソースが少ないマクロとなるように処理されます。後者の場合、この新しいより小型の方のマクロが XST でマクロとして処理されることもあります。たとえば、クロック イネーブル (CE) を持つフリップフロップ マクロは、XC9500 にマップするとマクロとして処理されませんが、HDL 合成により次の 2 つのマクロが生成されます。

- ・ クロック イネーブル信号のないフリップフロップ マクロ
- ・ クロック イネーブル機能をインプリメントする MUX マクロ

生成されたマクロは個別に最適化され、周辺のロジックと結合されます。このように最適化すると、大型のコンポーネントで良い結果が得られます。

CPLD 合成のログ ファイルの解析

XST の CPLD 合成に関する出力ログは、次のメッセージの後にあります。

Low Level Synthesis

XST で出力されるログ ファイルには、次の情報が含まれています。

- ・ ユニットの最適化の進捗状況を表示
 - Optimizing unit *unit_name* ...
- ・ ユニット最適化に関する情報、警告、エラー メッセージ
 - 代理式のシェーピング機能を使用されている場合 (XC9500 デバイスのみ)
 - Collapsing ...
 - 等価フリップフロップが削除された場合
 - #2 と同じ #1 レジスタを削除
 - XST でユーザー指定の制約が処理された場合
 - インプリメンテーション制約 *constraint_name*[=*value*]: *signal_name*
- ・ 最終統計

Final Results

```
Top Level Output file name : file_name
Output format : ngc
Optimization goal : {area | speed}
Target Technology : {9500 | 9500x1 | 9500xv | xpla3 | xbr | cr2s}
Keep Hierarchy : {yes | soft | no}
Macro Preserve : {yes | no}
XOR Preserve : {yes | no}
```

Design Statistics

```

NGC Instances: nb_of_instances
I/Os: nb_of_io_ports
Macro Statistics
# FSMs: nb_of_FSMs
# Registers: nb_of_registers
# Tristates: nb_of_tristates
# Comparators: nb_of_comparators
    n-bit comparator {equal | not equal | greater | less | greatequal | lessequal}:
    nb_of_n_bit_comparators
# Multiplexers: nb_of_multiplexers
    n-bit m-to-1 multiplexer :
    nb_of_n_bit_m_to_1_multiplexers
# Adders/Subtractors: nb_of_adds_subs
    n-bit adder: nb_of_n_bit_adds
    n-bit subtractor: nb_of_n_bit_subs
# Multipliers: nb_of_multipliers
# Logic Shifters: nb_of_logic_shifters
# Counters: nb_of_counters
    n-bit {up | down | updown} counter: nb_of_n_bit_counters
# XORs: nb_of_xors
Cell Usage :
# BELS: nb_of_bels
    # AND...: nb_of_and...
    # OR...: nb_of_or...
    # INV: nb_of_inv
    # XOR2: nb_of_xor2
    # GND: nb_of_gnd # VCC: nb_of_vcc
# FlipFlops/Latches: nb_of_ff_latch
    # FD...: nb_of_fd...
    # LD...: nb_of_ld...
# Tri-States: nb_of_tristates
    # BUFE: nb_of_buf_e
    # BUFT: nb_of_buf_t
# IO Buffers: nb_of_iobuffers
    # IBUF: nb_of_ibuf
    # OBUF: nb_of_obuf
    # IOBUF: nb_of_iobuf
    # OBUFE: nb_of_obufe
    # OBUFT: nb_of_obuft # Others: nb_of_others

```

CPLD 合成の制約

HDL デザインまたは制約ファイルで指定された制約 (属性) は、XST では NGC ファイルに信号プロパティとして記述されます。

CPLD 合成結果の向上

XST は、次のように CPLD フィッタ用に最適化されたネットリストを生成します。

- ・ 指定デバイスにフィット
- ・ ダウンロード プログラマブル ファイルを作成

XST での CPLD の下位レベル合成には、次が含まれます。

- ・ ロジック最小化
- ・ サブファンクション コラプス
- ・ ロジックの因数分解
- ・ ロジック分解

最適化が実行されると、ブール代数式に対応する NGC ネットリストが出力されます。CPLD フィッタでは、マクロセルの機能に最大限にフィットするようにこれらのブール代数式が再処理されます。代数式のシェーピング機能として知られる XST の特別な最適化プロセスは、次のオプションが選択されている場合に XC9500 および XC9500XL デバイスに適用されます。

- ・ [Keep Hierarchy]
No
- ・ [Optimization Effort]
2 または High
- ・ [Macro Preserve]
No

代数式のシェーピング機能には、クリティカル パスの最適化アルゴリズムも含まれます。このアルゴリズムは、クリティカル パスのレベル数を削減しようとしています。

CPLD フィッタでは次のような特別な最適化が行われるため、マルチレベルの最適化が推奨されます。

- ・ D フリップフロップから T フリップフロップへの変換
- ・ ドモルガン ブール代数式を選択

周波数の向上

周波数はロジックレベル数によって決定されます。レベル数を削減するには、次のオプションを設定することをお勧めします。

- ・ [Optimization Effort]
2 か High に設定します。
これらの値を設定すると、デザインを必要以上に複雑なものにせずにロジックレベル数を削減するため、コラプス アルゴリズムが使用されます
- ・ [Optimization Goal]
[Speed] に設定します。
この値に設定すると、レベル数の削減が優先されます。

周波数に最も影響するのは、CPLD フィッタの最適化です。CPLD フィッタのマルチレベル最適化を、**-ptersms** オプションに 20 ~ 50 までの値を増分 5 で設定して実行することをお勧めします。統計的には、30 で最高の周波数が得られます。

次のトライは、順に周波数が向上していく例を示しています。

- ・ [トライ 1](#)
- ・ [トライ 2](#)
- ・ [トライ 3](#)
- ・ [トライ 4](#)

CPU タイムは、次のトライ 1 から 4 へ増加しています。

トライ 1

[Optimization Effort] を 2、[Optimization Goal] を [Speed] に設定します。それ以外のオプションには、デフォルト値を使用します。

- ・ [Optimization Effort] :
2 または High
- ・ [Optimization Goal]
Speed

トライ 2

ユーザー階層をフラット化します。階層をフラット化すると、最適化プロセスでデザインの全体像が認識され、ロジック レベルを削減しやすくなります。

- ・ [Optimization Effort] :
1/Normal or 2/High
- ・ [Optimization Goal]
Speed
- ・ [Keep Hierarchy]
no

トライ 3

マクロを周辺ロジックと結合します。デザインがさらにフラット化されます。

- ・ [Optimization Effort] :
1 または Normal
- ・ [Optimization Goal]
Speed
- ・ [Keep Hierarchy]
no
- ・ [Macro Preserve]
no

トライ 4

式のシェーピング (最適化) アルゴリズムを適用します。選択するオプションは次のとおりです。

- ・ [Optimization Effort] :
2 または High
- ・ [Macro Preserve]
no
- ・ [Keep Hierarchy]
no

大規模デザインのフィット

デバイスのマクロセル数または積項を超過していて、選択デバイスにデザインがフィットしない場合、次を実行することができます。

- ・ [XST でエリア最適化を選択](#)、または
- ・ [WYSIWYG コマンドライン オプションを使用](#)

XST でエリア最適化を選択

XST でエリア最適化を選択した場合、統計的に次のようにオプションを設定していると最良のエリアを取得することができます。

- ・ [Optimization Effort] :
1 (Normal) または 2 (High)
- ・ [Optimization Goal]
area
- ・ その他のオプション：デフォルト値

WYSIWYG コマンドライン オプションを使用

大規模デザインをフィットする別のオプションは、[WYSIWYG \(-wysiwyg\)](#) コマンドライン オプションで、次のように設定します。

-wysiwyg yes

WYSIWYG コマンドライン オプションは、次のような場合に使用すると便利です。

- ・ デフォルトの最適化方法でデザインが単純化できない場合、および
- ・ 複雑さ (積項数) がデバイス容量に近い場合

デフォルトの最適化では、ロジックレベル数の低減を優先するため、等式が増加します。このため、積項数が増え、デザインがフィットしなくなってしまう場合があります。これらの最適化と異なり、[WYSIWYG \(-wysiwyg\)](#) コマンドライン オプションでは積項数を増やさないような方法が使用されるので、デザインがフィットしやすくなります。

デザイン制約

この章では、XST デザイン制約の一般的な情報について説明します。

- ・ [XST デザイン制約の概要](#)
- ・ [制約の指定方法](#)
- ・ [グローバルおよびローカル制約の設定](#)
- ・ [制約の適用規則](#)
- ・ [グローバル制約とオプションの設定](#)
- ・ [VHDL 属性の構文](#)
- ・ [Verilog-2001 の属性](#)
- ・ [XST 制約ファイル \(XCF\)](#)
- ・ [制約の優先順位](#)
- ・ [XST 固有の制約 \(タイミング以外\)](#)
- ・ [XST コマンドラインのみのオプション](#)

特定の XST デザイン制約の詳細は、次を参照してください。

- ・ [一般制約](#)
- ・ [HDL 制約](#)
- ・ [FPGA 制約 \(タイミング制約以外\)](#)
- ・ [CPLD 制約 \(タイミング以外\)](#)
- ・ [タイミング制約](#)
- ・ [インプリメンテーション制約](#)
- ・ [サードパーティの制約](#)

XST デザイン制約の概要

制約は、デザインの目標を達成したり、最適なインプリメンテーションを実行するのに役立ちます。制約を使用すると、合成プロセスや配置配線のさまざまな処理を制御できます。ほとんどの場合、合成アルゴリズムによって自動的に最適な結果が得られますが、場合によっては、最適な結果が得られないこともあります。このような場合、制約を使用することにより、別の方法で合成し直すことができます。

制約の指定方法

制約を設定するには、次のような方法があります。

- ・ オプションを使用して合成をグローバルに制御します。これらは、次のいずれかで設定できます。
 - ISE® Design Suite の [Process Properties] ダイアログ ボックス → [Synthesis Options]、または
 - コマンド ラインからの run コマンド
- ・ VHDL の場合、属性を直接 VHDL コードに挿入し、デザインの各エレメントに設定して合成および配置配線を制御する。
- ・ Verilog の場合、制約は次で追加できます。
 - Verilog 属性 (推奨)
 - Verilog メタ コメント
- ・ 制約を別の制約ファイルで指定することもできます。

グローバルおよびローカル制約の設定

通常、グローバルな合成オプションは、ISE® Design Suite の [Process Properties] ダイアログ ボックスの [Synthesis Options]、またはコマンド ラインで設定します。一方、VHDL/Verilog 属性や Verilog メタ コメントはソース コードに記述するので、デザインの各エレメントに異なる制約を設定できます。

各エレメントへの設定は、グローバル設定よりも優先されます。同様に、ノード (またはインスタンス) とそれを含むデザイン ユニットの両方に制約を設定した場合、ノード (またはインスタンス) の制約が優先されます。

制約の適用規則

制約を指定するには、次の一般的な規則に従う必要があります。

- ・ 信号には、複数の制約を設定できます。複数の制約を設定する場合、制約は信号が宣言され、使用されているブロック内で指定する必要があります。
- ・ 制約はエンティティ (VHDL) に適用してから、コンポーネント宣言でも適用することができます。コンポーネントに制約を適用可能なことは、これが一般的な XST の規則であるため、制約別にははっきり記載されていません。
- ・ アーキテクチャ文に制約を適用できるサードパーティの合成ツールもあります。XST でアーキテクチャに制約を設定できるのは、XST で自動的にサポートされるサードパーティ制約に対してのみです。

グローバル制約とオプションの設定

このセクションでは、グローバル制約とそのオプションの設定について説明します。

- ・ [合成オプション設定](#)
- ・ [HDL オプションの設定](#)
- ・ [ザイリンクス特有オプションの設定](#)
- ・ [その他の XST コマンドライン オプションの設定](#)
- ・ [カスタム コンパイル ファイル リスト](#)

このセクションでは、ISE® Design Suite の [Process Properties] ダイアログ ボックスでグローバル制約およびオプションを設定する方法を説明します。

FPGA デバイス、CPLD デバイス、VHDL、Verilog などで一般的に適用される制約についての詳細は、『[制約ガイド](#)』を参照してください。

チェック ボックス付きのフィールド以外の値フィールドには、プルダウン メニューの矢印ボタンや参照ボタンが付いています。ボックスをクリックするまでは、矢印ボタンは画面上に表示されません。

合成オプション設定

ISE® Design Suite から HDL 合成オプションを設定するには

1. [Sources] ウィンドウでソース ファイルを選択します。
2. [Processes] ウィンドウで [Synthesize - XST] を右クリックします。
3. [Process Properties] をクリックします。
4. [Synthesis Options] をクリックします。
5. デバイス ファミリ (FPGA または CPLD) の選択によって、表示されるダイアログ ボックスは異なります。
6. 次の合成オプションのいずれかを選択します。
 - ・ [最適化方法の指定 \(OPT_MODE\)](#)
 - ・ [最適化エフォートレベル \(OPT_LEVEL\)](#)
 - ・ [合成制約ファイルの使用 \(-iuc\)](#)
 - ・ [合成制約ファイルの指定 \(-uc\)](#)
 - ・ [ライブラリの検索順 \(-lso\)](#)
 - ・ [グローバルな最適化目標 \(-glob_opt\)](#)
 - ・ [RTL 回路図の生成 \(-rtlview\)](#)
 - ・ [タイミング制約の書き込み \(-write_timing_constraints\)](#)
 - ・ [Verilog 2001 の指定 \(-verilog2001\)](#)

次のオプションを表示するには、[Edit] → [Preferences] → [Processes] → [Property display level] を [Advanced] に設定します。

- ・ 階層の維持 (KEEP_HIERARCHY)
- ・ コアの検索ディレクトリ (-sd)
- ・ クロス クロック解析 (-cross_clock_analysis)
- ・ 階層区切り文字の指定 (-hierarchy_separator)
- ・ バスの区切り文字指定 (-bus_delimiter)
- ・ 大文字/小文字の指定(-case)
- ・ 作業ディレクトリの指定 (-xsthdpdir)
- ・ HDL ライブラリ マップ ファイル (-xsthdpini)
- ・ Verilog の 'include ディレクトリの指定 (-vlgincdir)
- ・ Slice (LUT-FF Pairs) Utilization Ratio (スライス (LUT-FF ペア) 使用率)

HDL オプションの設定

このセクションでは、次について説明します。

- ・ HDL オプションの設定方法
- ・ FPGA デバイスの HDLオプションの設定
- ・ CPLD デバイスの HDL オプションの設定

HDL オプションの設定方法

FPGA と CPLD デバイスのオプションは、ISE® Design Suite から設定できます。

[Process] → [Process Properties] → [Synthesize - XST] → [HDL Options]

FPGA デバイスの HDL オプションの設定

FPGA デバイスの場合は、次の HDL オプションを設定できます。

- ・ FSM エンコード方法の指定 (FSM_ENCODING)
- ・ セーフ インプリメンテーション (SAFE_IMPLEMENTATION)
- ・ Case 文のインプリメンテーション形式 (-vlgcase)
- ・ FSM スタイル (FSM_STYLE)

FSM スタイル オプションを表示するには、[Process Properties] ダイアログ ボックスで [Property display level] を [Advanced] に設定します。

- ・ RAM の抽出 (RAM_EXTRACT)
- ・ RAM スタイル (RAM_STYLE)
- ・ ROM の抽出 (ROM_EXTRACT)
- ・ ROM スタイル (ROM_STYLE)
- ・ MUX の抽出 (MUX_EXTRACT)
- ・ MUX スタイル (MUX_STYLE)
- ・ デコーダの抽出 (DECODER_EXTRACT)
- ・ プライオリティ エンコーダの抽出 (PRIORITY_EXTRACT)
- ・ シフトレジスタの抽出 (SHREG_EXTRACT)
- ・ 論理シフトの抽出 (SHIFT_EXTRACT)
- ・ XOR コラプス (XOR_COLLAPSE)
- ・ リソース共有 (RESOURCE_SHARING)
- ・ 乗算器スタイル (MULT_STYLE)

新しいデバイスの場合、この [Multiplier Style] プロパティの代わりに 次のプロパティが表示されます。

- [Use DSP48]
Virtex®-4 デバイス
- [Use DSP Block]
Virtex-5、Spartan®-3A DSP デバイス
- ・ DSP48 の使用 (USE_DSP48)

CPLD デバイスの HDL オプションの設定

CPLD デバイスの場合は、次の HDL オプションを設定できます。

- ・ FSM エンコード方法の指定 (FSM_ENCODING)
- ・ セーフ インプリメンテーション (SAFE_IMPLEMENTATION)
- ・ Case 文のインプリメンテーション形式 (-vlgcase)
- ・ MUX の抽出 (MUX_EXTRACT)
- ・ リソース共有 (RESOURCE_SHARING)

ザイリンクス特有オプションの設定

このセクションでは、ザイリンクス特有のオプションの設定について説明します。

- ・ [ザイリンクス特有オプションの設定方法](#)
- ・ [FPGA デバイスのザイリンクス特有オプションの設定](#)
- ・ [CPLD デバイスのザイリンクス特有オプションの設定](#)

ザイリンクス特有オプションの設定方法

ザイリンクス特有オプションは、ISE® Design Suite で次をクリックすると設定できます。

[Process] → [Process Properties] → [Synthesis Options] → [Xilinx Specific Options]

FPGA デバイスのザイリンクス特有オプションの設定

FPGA デバイスの場合は、次のザイリンクス特有オプションを設定できます。

- ・ [I/O バッファの追加 \(-iobuf\)](#)
- ・ [LUT の結合 \(LC\)](#)
- ・ [最大ファンアウト数 \(MAX_FANOUT\)](#)
- ・ [レジスタの複製 \(REGISTER_DUPLICATION\)](#)
- ・ [制御セットの削減 \(REDUCE_CONTROL_SETS\)](#)
- ・ [等価レジスタの削除 \(EQUIVALENT_REGISTER_REMOVAL\)](#)
- ・ [レジスタの自動調整 \(REGISTER_BALANCING\)](#)
- ・ [最初のフリップフロップ ステージの移動 \(MOVE_FIRST_STAGE\)](#)
- ・ [最後のフリップフロップ ステージの移動 \(MOVE_LAST_STAGE\)](#)
- ・ [トライステートからロジックへの変換 \(TRISTATE2LOGIC\)](#)

これは、内部トライステート リソースを含むデバイスの場合にのみ表示されます。

- ・ [クロック イネーブルの使用 \(USE_CLOCK_ENABLE\)](#)
- ・ [同期セットの使用 \(USE_SYNC_SET\)](#)
- ・ [同期リセットの使用 \(USE_SYNC_RESET\)](#)

次のオプションを表示するには、ISE Design Suite の [Process Properties] ダイアログ ボックスで [Property display level] を [Advanced] に設定します。

- ・ [グローバル クロック バッファ数 \(-bufg\)](#)
- ・ [リージョン クロック バッファ数 \(-bufr\)](#)

CPLD デバイスのザイリンクス特有オプションの設定

CPLD デバイスの場合は、次のザイリンクス特有オプションを設定できます。

- ・ [I/O バッファの追加 \(-iobuf\)](#)
- ・ [等価レジスタの削除 \(EQUIVALENT_REGISTER_REMOVAL\)](#)
- ・ [クロック イネーブル \(-pld_ce\)](#)
- ・ [マクロの保持 \(-pld_mp\)](#)
- ・ [XOR の保持 \(-pld_xp\)](#)
- ・ [WYSIWYG \(-wysiwyg\)](#)

その他の XST コマンドライン オプションの設定

このセクションでは、次について説明します。

- ・ ISE Design Suite でのオプションの設定
- ・ オプション設定のヒント
- ・ オプションの優先順位
- ・ 不正または認識されないオプション

ISE Design Suite でのオプションの設定

ISE® Design Suite でその他の XST コマンドライン オプションを設定します。

[Process] → [Process Properties] → [Other XST Command Line Options]

これは、アドバンス プロパティです。

オプション設定のヒント

XST コマンドライン オプションを設定する場合：

- ・ 「コマンドライン モード」の構文を使用します。
- ・ オプションを複数指定する場合は、スペースで区切ります。

オプションの優先順位

このプロパティは、[Process Properties] ダイアログ ボックスにリストされていないオプションを設定するためのものですが、既にリストされているオプションを入力すると、そちらが優先されます。

不正または認識されないオプション

オプションの入力方法が不正だったり、認識されないオプションを入力すると、次のようなエラー メッセージが表示され、XST での処理が停止します。

```
ERROR:Xst:1363 - Option "-verilog2002" is not available  
for command run.
```

カスタム コンパイル ファイル リスト

[Custom Compile File List] プロパティを使用すると、XST でソース ファイルを処理する順序を変更できます。このプロパティで、ライブラリおよびデザイン ファイルの処理順を指定したコンパイル リスト ファイルを指定します。このプロパティでファイルを指定しない場合は、自動的に生成されたリストが使用されます。

カスタム コンパイル リスト ファイルには、すべてのデザイン ファイルおよび関連するライブラリをコンパイルする順にリストします。ファイルとライブラリの組を 1 行に 1 つずつリストし、ファイルとライブラリはセミコロンで区切ります。フォーマットは次のとおりです。

```
library_name; file_name [library_name;file_name] ...
```

例：

```
work;stopwatch.vhd  
work;statmach.vhd  
...
```

このプロパティは、[Simulation Properties] ページの [Custom Compile File List] プロパティとは関係ありません。合成とシミュレーションでは異なるコンパイル リスト ファイルが使用されます。

VHDL 属性の構文

VHDL コードの場合、制約は VHDL 属性を使用して記述できます。

次のように宣言します。

```
attribute AttributeName : Type;
```

構文例 1

```
attribute RLOC : string;
```

属性タイプでは、属性値の種類を定義します。XST で使用できるタイプは string のみです。属性は、エンティティまたはアーキテクチャで宣言できます。エンティティで宣言した場合、その属性はエンティティおよびアーキテクチャ本体の両方で使用できます。アーキテクチャで宣言した場合は、その属性はエンティティ宣言では使用できません。

次のように指定します。

```
attribute AttributeName of ObjectList : ObjectType is AttributeValue ;
```

構文例 2

```
attribute RLOC of u123 : label is R11C1.S0 ; attribute bufg of my_signal: signal is sr;
```

使用できるオブジェクト タイプ

オブジェクト リストは、識別子をカンマで区切ったリストです。使用できるオブジェクト タイプは次のとおりです。

- ・ entity
- ・ component
- ・ label
- ・ signal
- ・ variable
- ・ type

一般的なルール

- ・ 制約はエンティティ (VHDL) に適用してから、コンポーネント宣言でも適用することができます。コンポーネントに制約を適用可能なことは、これが一般的な XST の規則であるため、制約別にははっきり記載されていません。
- ・ アーキテクチャ文に制約を適用できるサードパーティの合成ツールもあります。XST でアーキテクチャに制約を設定できるのは、XST で自動的にサポートされるサードパーティ制約に対してのみです。

Verilog-2001 の属性

XST では Verilog-2001 属性文がサポートされています。属性は、合成ツールなどのソフトウェア ツールに特定の情報を渡すために使用します。Verilog-2001 の属性は、モジュール宣言およびインスタンス化内、演算子または信号に指定できます。その他の属性宣言はコンパイラでサポートされる場合がありますが、XST では無視されます。

Verilog-2001 属性の構文

Verilog-2001 の属性は、(*) で囲む必要があり、次のように記述されます。

(* attribute_name = attribute_value *)

説明：

- ・ attribute を参照する信号、モジュール、またはインスタンスの宣言の前に記述する必要があります。
- ・ attribute_value には、文字列を指定する必要があります。整数値やスカラ値は使用できません。
- ・ attribute_value は、二重引用符 (") で囲む必要があります。
- ・ デフォルトは 1 です。
- ・ (* attribute_name *) は (* attribute_name = "1" *) と同じです。

構文例 1

(* clock_buffer = "IBUFG" *) input CLK;

構文例 2

(* INIT = "0000" *) reg [3:0] d_out;

構文例 3

always@(current_state or reset) begin **(* parallel_case *)** **(* full_case *)** case (current_state) ...

構文例 4

(* mult_style = "pipe_lut" *) MULT my_mult (a, b, c);

Verilog-2001 の制限

次の Verilog-2001 属性は、サポートされていません。

- ・ 信号の宣言
- ・ ステートメント
- ・ ポートの接続
- ・ 論理演算子

Verilog-2001 のメタ コメント

メタ コメントを使用すると、制約を Verilog コードで指定することもできます。Verilog-2001 形式がよく使用されますが、Verilog でもメタ コメントがサポートされています。次の構文を使用してください。

// synthesis attribute AttributeName [of] ObjectName [is] AttributeValue

Verilog-2001 のメタ コメントの例

```
// synthesis attribute RLOC of u123 is R11C1.S0
// synthesis attribute HU_SET u1 MY_SET
// synthesis attribute bufg of my_clock is "clk"
```

次の制約には、異なる構文を使用します。

- ・ [パラレル ケース \(PARALLEL_CASE\)](#)
- ・ [フル ケース \(FULL_CASE\)](#)
- ・ [変換なし \(TRANSLATE_OFF\)](#) と [変換あり \(TRANSLATE_ON\)](#)

詳細は、[次](#)を参照してください。

[Verilog の属性とメタ コメント](#)

XST Constraint File (XCF)

このセクションには、次の項目が含まれます。

- ・ [XCF での制約の設定](#)
- ・ [XCF 構文とその使用](#)
- ・ [ネイティブとそれ以外の UCF 制約構文](#)
- ・ [XCF 構文の制限](#)

XCF での制約の設定

XST 制約は、ザイリンクス制約ファイル (XCF) で指定できます。

制約ファイルの拡張子は .xcf です。

XCF は次で指定できます。

- ・ ISE® Design Suite

詳細は、次を参照してください。

[ISE Design Suite ヘルプ](#)

- ・ コマンド ライン モード

コマンド ラインで XCF を指定するには、run コマンドで[合成制約ファイル \(-uc\)](#) オプションを使用します。

詳細は、次を参照してください。

[コマンド ライン モード](#)

XCF 構文とその使用

このセクションでは、XCF の構文および使用方法について説明しています。

- ・ [XCF 構文および使用の概要](#)
- ・ [構文](#)
- ・ [構文例および設定](#)
- ・ [XST 合成制約](#)

XCF 構文および使用の概要

XCF 構文を使用すると、特定の制約を次のいずれかに指定できます。

- ・ デバイス全体 (グローバル)
- ・ 特定モジュール

この構文は、ネットまたはインスタンスに制約を設定するという点では UCF 構文と同じですが、さらに詳細に指定することで階層の特定レベルに制約を適用できます。キーワード **MODEL** を使用して、制約を適用するエンティティ/モジュールを定義します。エンティティ/モジュールに制約を設定した場合、制約はそのエンティティ/モジュールの各インスタンスに適用されます。

制約は ISE® Design Suite の [Process] → [Process Properties] ダイアログ ボックスまたはコマンドラインで XST の run コマンドから定義し、例外を XCF ファイルで指定します。XCF ファイルで指定した制約は、指定されたモジュールにのみ適用され、その下位にあるサブモジュールには適用されません。

構文

エンティティ/モジュール全体に制約を適用するには、次の構文を使用します。

```
MODEL entityname constraintname = constraintvalue;
```

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XCF の構文例 1

```
MODEL top mux_extract = false; MODEL my_design max_fanout = 256;
```

上記の例では、デザインにエンティティ my_design を複数回インスタンス化すると、max_fanout=256 制約が my_design の各インスタンスに適用されます。

エンティティ/モジュールの特定インスタンスまたは信号に制約を適用するには、キーワード **INST** または **NET** を使用します。XST では、VHDL 変数に適用される制約はサポートされません。

```
BEGIN MODEL entityname  
INST instancename constraintname = constraintvalue;  
NET signalname constraintname = constraintvalue;  
END;
```

XCF の構文例 2

```
BEGIN MODEL crc32  
  INST stopwatch opt_mode = area ;  
  INST U2 ram_style = block ;  
  NET myclock clock_buffer = true ;  
  NET data_in iob = true ;  
END;
```

XST 合成制約

XST で適用できる合成制約のリストは、次を参照してください。

[XST 固有の制約 \(タイミング以外\)](#)

ネイティブとそれ以外の UCF 制約構文

XST でサポートされる制約は次の 2 タイプの UCF (ユーザー制約ファイル) 制約に分類できます。

- ・ [ネイティブ UCF 制約](#)
- ・ [ネイティブ以外の UCF 制約](#)

ネイティブ UCF 制約

ネイティブ User Constraints File (UCF) の構文が使用されるのは、タイミング制約とエリアグループ制約のみです。

次のようなネイティブ UCF 制約には、ワイルドカードおよび階層名を含めたネイティブ UCF 構文を使用します。

- ・ [周期 \(PERIOD\)](#)
- ・ [オフセット \(OFFSET\)](#)
- ・ [ネットのタイミング名 \(TNM_NET\)](#)
- ・ [タイムグループ \(TIMEGRP\)](#)
- ・ [タイミング無視 \(TIG\)](#)
- ・ [From-To \(FROM-TO\)](#)

Restriction これらの制約は **BEGIN MODEL... END** 文内には使用しないでください。使用すると、XST でエラーが発生します。

ネイティブ以外の UCF 制約

次のようなネイティブ以外の UCF 制約の場合は、**MODEL** または **BEGIN MODEL... END;** 構文を使用します。

- ・ 次のような純粋な XST 制約
 - [FSM 自動抽出 \(FSM_EXTRACT\)](#)
 - [RAM スタイル \(RAM_STYLE\)](#)
- ・ タイミング以外のインプリメンテーション制約
 - [RLOC](#)
 - [キープ \(KEEP\)](#)

XCF ファイルでタイミング制約を指定する場合、階層の区切り記号にはアンダースコア (_) ではなく、スラッシュ (/) を使用してください。

詳細は、次を参照してください。

[階層区切り文字の指定 \(-hierarchy_separator\)](#)

XCF 構文の制限

XST Constraint File (XCF) 構文には、次の制限があります。

- ・ MODEL 文のネストはサポートされません。
- ・ **BEGIN MODEL** と **END** 間にリストされたインスタンス名や信号名のみが、エンティティ内で有効です。階層インスタンス名または信号名はサポートされません。
- ・ タイミング制約以外の制約では、インスタンス名や信号名のワイルドカードはサポートされません。
- ・ すべてのネイティブ User Constraints File (UCF) 制約がサポートされているわけではありません。

詳細は、『[制約ガイド](#)』を参照してください。

制約の優先順位

制約の優先順位は、その制約が含まれるファイルによって異なります。デザイン フローの前半で使用されるファイルの制約よりも、フローの後半で使用されるファイルの制約の方が優先されます。

優先順位は次のとおりです。

1. 合成制約ファイル
2. HDL (ハードウェア記述言語) ファイル
3. ISE® Design Suite [Process Properties] またはコマンド ライン

XST 固有の制約（タイミング以外）

次の表に示す項目は、次のとおりです。

- ・ 各制約に使用できる値
- ・ 適用できるオブジェクトの種類
- ・ 使用制限

多くの場合、制約はエンティティまたはモデル全体にグローバルに適用するか、または個々の信号、ネット、またはインスタンスにローカルに適用できます。

制約名	制約値	VHDL ターゲット	Verilog ター ゲット	XCF ターゲッ ト	コマンドライン	コマンド値
ボックス タイプ	primitive black_box user_black_box	entity inst	module inst	model inst (model 内)	なし	なし
BRAM のマップ ロジック	yes no	entity	module	model	なし	なし
バッファ タイプ	bufgdl ibufg BUFG bufgp ibuf bufr none	signal	signal	net (model 内)	なし	なし
BUFGCE の抽 出	yes no	primary clock signal	primary clock signal	net (model 内)	-bufgce	yes no デフォルト：no
クロック信号	yes no	clock signal	clock signal	clock signal net (model 内)	なし	なし
デコーダ抽出	yes no	entity signal	entity signal	model net (model 内)	-decoder _extract	yes no デフォルト：yes
列挙型エンコー ド手法	スペースで区分 された 2 進数を 含むstring	type	signal	net (model 内)	なし	なし

制約名	制約値	VHDL ターゲット	Verilog ター ゲット	XCF ターゲッ ト	コマンドライン	コマンド値
等価レジスタの 削除	yes no	entity signal	module signal	model net (model 内)	-equivalent _register _removal	yes no デフォルト : yes
FSM エンコード 方法の指定	auto one-hot compact sequential gray johnson speed1 user	entity signal	module signal	model net (model 内)	-fsm _encoding	auto one-hot compact sequential gray johnson speed1 user デフォルト : auto
FSM 自動抽出	yes no	entity signal	module signal	model net (model 内)	-fsm _extract	yes no デフォルト : yes
FSM スタイル	lut bram	entity signal	module signal	model net (model 内)	-fsm _style	lut bram デフォルト : lut
フル ケース	なし	なし	case 文	なし	なし	なし
I/O レジスタの IOB 内へのパッ ク	true false auto	signal instance	signal instance	net (model 内) inst (model 内)	-iob	true false auto デフォルト : auto
I/O 規格	string 詳細は、『制約 ガイド』を参照し てください。	signal instance	signal instance	net (model 内) inst (model 内)	なし	なし
キープ	true false soft	signal	signal	net (model 内)	なし	なし

制約名	制約値	VHDL ターゲット	Verilog ター ゲット	XCF ターゲッ ト	コマンドライン	コマンド値
階層の維持	yes no soft	entity	module	model	-keep _hierarchy	デフォルト (CPLD) : yes デフォルト (FPGA) : no soft
LOC	string	signal (primary IO) instance	signal (primary IO) instance	net (model 内) inst (model 内)	なし	なし
単一 LUT への エンティティの マップ	yes no	entity architecture	module	model	なし	なし
最大ファンアウト	integer	entity signal	module signal	model net (model 内)	-max _fanout	integer デフォルト：詳 細な説明を参照 してください。
最初のフリップ フロップ ステ ージの移動	yes no	entity primary clock signal	module primary clock signal	model primary clock signal net (model 内)	-move _first _stage	yes no デフォルト : yes
最後のステージ の移動	yes no	entity primary clock signal	module primary clock signal	model primary clock signal net (model 内)	-move _last _stage	yes no デフォルト : yes
乗算器スタイル	auto block pipe_block kcm csd lut pipe_lut	entity signal	module signal	model net (model 内)	-mult _style	auto block pipe_block kcm csd lut pipe_lut デフォルト : auto

制約名	制約値	VHDL ターゲット	Verilog ター ゲット	XCF ターゲッ ト	コマンドライン	コマンド値
MUX 抽出	yes no force	entity signal	module signal	model net (model 内)	-mux _extract	yes no force デフォルト : yes
MUX スタイル	auto muxf muxcy	entity signal	module signal	model net (model 内)	-mux _style	auto muxf muxcy デフォルト : auto
削減なし	yes no	signal	signal	net (model 内)	なし	なし
最適化エフォ ート	1 2	entity	module	model	-opt _level	1 2 デフォルト : 1
最適化目標	speed area	entity	module	model	-opt _mode	speed area デフォルト : speed
インスタンス エートされたプ リミティブの最適 化	yes no	entity instance	module instance	model instance (model 内)	-optimize _primitives	yes no デフォルト : no
パラレル ケース	なし	なし	case 文	なし	なし	なし
電力削減	yes no	entity	module	model	-power	yes no デフォルト : no
プライオリティ エ ンコーダの抽出	yes no force	entity signal	module signal	model net (model 内)	-priority _extract	yes no force デフォルト : yes
RAM 抽出	yes no	entity signal	module signal	model net (model 内)	-ram _extract	yes no デフォルト : yes

制約名	制約値	VHDL ターゲット	Verilog ター ゲット	XCF ターゲッ ト	コマンドライン	コマンド値
RAM スタイル	auto block distributed pipe_distributed block_power1 block_power2	entity signal	module signal	model net (model 内)	-ram _style	auto block distributed デフォルト : auto
コアの自動読み 込み	yes no optimize	entity component	module label	model inst (model 内)	-read _cores	yes no optimize デフォルト : yes
レジスタ自動調 整	yes no forward backward	entity signal FF instance name	module signal FF instance name primary clock signal	modelnet (model 内) inst (model 内)	-register _balancing	yes no forward backward デフォルト : no
レジスタの複製	yes no	entity signal	module	model net (model 内)	-register _duplication	yes no デフォルト : yes
リソース共有	yes no	entity signal	module signal	model net (model 内)	-resource _sharing	yes no デフォルト : yes
ROM 抽出	yes no	entity signal	module signal	model net (model 内)	-rom _extract	yes no デフォルト : yes
ROM スタイル	auto block distributed	entity signal	module signal	model net (model 内)	-rom _style	auto block distributed デフォルト : auto

制約名	制約値	VHDL ターゲット	Verilog ター ゲット	XCF ターゲッ ト	コマンドライン	コマンド値
保存	yes no	signal inst of primitive	signal inst of primitive	net (model 内) inst of primitive (in model)	なし	なし
セーフ インプリ メンテーション	yes no	entity signal	module signal	model net (model 内)	-safe _implementation	yes no デフォルト : no
セーフ リカバリ ステート	string	signal	signal	net (model 内)	なし	なし
論理シフトレジ スタ抽出	yes no	entity signal	module signal	model net (model 内)	-shift _extract	yes no デフォルト : yes
シフトレジスタ 抽出	yes no	entity signal	module signal	model net (model 内)	-shreg extract	yes no デフォルト : yes
信号のエンコー ド方法	auto one-hot user	entity signal	module signal	model net (model 内)	-signal _encoding	auto one-hot user デフォルト : auto
スライス使用率	整数 (-1 ~ 100) 整数% (-1 ~ 100) 整数#	entity	module	model	-slice _utilization _ratio	整数 (-1 ~ 100) 整数% (-1 ~ 100) 整数# デフォルト : 100
スライス使用率 の許容範囲	整数 (0 ~ 100) 整数% (0 ~ 100) 整数#	entity	module	model	-slice _utilization _ratio _maxmargin	整数 (0 ~ 100) 整数% (0 ~ 100) 整数# デフォルト : 0
Translate Off と Translate On	なし	local no target	local no target	なし	なし	なし
トライステートか らロジックへの 変換	yes no	entity signal	module signal	model net (model 内)	-tristate2logic	yes no デフォルト : yes

制約名	制約値	VHDL ターゲット	Verilog ター ゲット	XCF ターゲッ ト	コマンドライン	コマンド値
キャリー チェー ンの使用	yes	entity	module	model	-use	yes
	no	signal	signal	net (model 内)	_carry _chain	no デフォルト : yes
クロック イネー ブルの使用	auto	entity	module	model	-use	auto
	yes	signal	signal	net (model 内)	_clock	yes
	no	FF	FF	inst (model 内)	_enable	no
		instance name	instance name			デフォルト : auto
DSP48 の使用	auto	entity	module	model	-use	auto
	yes	signal	signal	net (model 内)	_dsp48	yes
	no					no デフォルト : auto
同期リセットの 使用	auto	entity	module	model	-use	auto
	yes	signal	signal	net (model 内)	_sync	yes
	no	FF	FF	inst (model 内)	_reset	no
		instance name	instance name			デフォルト : auto
同期セットの使 用	auto	entity	module	model	-use	auto
	yes	signal	signal	net (model 内)	_sync	yes
	no	FF	FF	inst (model 内)	_set	no
		instance name	instance name			デフォルト : auto
XOR コラプス	yes	entity	module	model	-xor	yes
	no	signal	signal	net (model 内)	_collapse	no デフォルト : yes

XST コマンド ラインのみのオプション

このセクションには次の内容が含まれます。

- ・ コマンド ラインでのみサポートされる XST 特有のタイミング以外のオプション
- ・ XST タイミング オプションの起動
- ・ コマンド ラインまたは[Process Properties] ダイアログ ボックスでのみサポートされる XST タイミング制約
- ・ XCF でのみサポートされる XST タイミング制約

コマンド ラインでのみサポートされる XST 特有のタイミング以外のオプション

制約名	コマンド ライン	コマンド値
VHDL 最上位レベルのアーキテクチャ	-arch	architecture_name デフォルト：なし
アーキテクチャ サポート	-async_to_sync	yes no デフォルト：no
自動 BRAM パッキング	-auto_bram_packing	yes no デフォルト：no
BRAM 使用率 (BRAM_UTILIZATION_RATIO)	-bram_utilization_ ratio	integer (range -1 to 100) integer% (range -1 to 100) integer# デフォルト：100
最大グローバル クロック バッファ	-bufg	整数 デフォルト：ターゲット デバイスの最大 バッファ数
最大リージョン クロック バッファ	-bufr	整数 デフォルト：ターゲット デバイスの最大 バッファ数
バス区切り文字	-bus_delimiter	<> [] {} () デフォルト：<>

制約名	コマンドライン	コマンド値
大文字/小文字	-case	upper lower maintain デフォルト： maintain
Verilog マクロ	-define	{name = value} デフォルト： なし
DSP 使用率 (DSP_UTILIZATION_RATIO)	-dsp_utilization_ratio	integer (range -1 to 100) integer% (range -1 to 100) integer# デフォルト： 100
複製接尾語の設定	-duplication_suffix	string%dstring デフォルト： %d
VHDL 最上位レベルのブロック (以前の VHDL プロジェクト フォーマット (-ifmt VHDL) でのみ使用可能。合成する最上位レベルのブロックを指定するには、プロジェクト フォーマット (-ifmt mixed) か -top オプションを使用してください)	-ent	entity_name デフォルト： なし
ジェネリック	-generics	{name = value} デフォルト： なし
HDL ファイルのコンパイル順序	-hdl_compilation_order	auto user デフォルト： auto
階層区切り文字	-hierarchy_separator	- / デフォルト： /
入力形式	-ifmt	mixed vhdl verilog デフォルト： mixed
入力/プロジェクト ファイル名	-ifn	file_name デフォルト： なし

制約名	コマンドライン	コマンド値
I/O バッファの追加	-iobuf	yes no デフォルト：yes
ユーザー制約の無視	-iuc	yes no デフォルト：no
Library Search Order (ライブラリ検索順)	-lso	file_name.lso デフォルト：なし
LUT の結合	-lc	auto area off デフォルト：off
ネットリスト階層	-netlist_hierarchy	as_optimized rebuilt デフォルト：as_optimized
出力ファイル形式	-ofmt	NGC デフォルト：NGC
出力ファイル名	-ofn	file_name デフォルト：なし
ターゲット デバイス	-p	part-package-speed (例： xc5vfx30t-ft324-2) デフォルト：なし
クロック イネーブル	-pld_ce	yes no デフォルト：yes
[Macro Preserve] オプション	-pld_mp	yes no デフォルト：yes
XOR 保持	-pld_xp	yes no デフォルト：yes

制約名	コマンドライン	コマンド値
制御セットの削減	-reduce_control_sets	auto no デフォルト：no
RTL 回路図の生成	-rtlview	yes no only デフォルト：no
コアの検索ディレクトリ	-sd	ディレクトリ デフォルト：なし
スライス パッキング	-slice_packing	yes no デフォルト：yes
最上位レベル ブロック	-top	block_name デフォルト：なし
合成制約ファイル	-uc	file_name.xcf デフォルト：なし
[Verilog 2001] オプション	-verilog2001	yes no デフォルト：yes
case 文のインプリメント方法	-vlgcase	full parallel full-parallel デフォルト：なし
Verilog インクルード ディレクトリ	-vlgincdir	ディレクトリ デフォルト：なし
作業ライブラリ	-work_lib	ディレクトリ デフォルト：work
WYSIWYG	-wysiwyg	yes no デフォルト：no

制約名	コマンドライン	コマンド値
作業ディレクトリ	-xsthdpdir	ディレクトリ デフォルト： ./xst
HDL ライブラリ マップ ファイル	-xsthdpini	file_name.ini デフォルト： なし

XST タイミング オプションの指定

XST のタイミング オプションは、次のように指定できます。

- ・ ISE® Design Suite の [Process] → [Process Properties]
- ・ コマンドライン
- ・ ザイリンクス制約ファイル (XCF)

コマンドラインまたは[Process Properties] ダイアログ ボックスでのみサポートされる XST タイミング制約

オプション	[Process] → [Process Properties] (ISE® Design Suite)	値
glob_opt	グローバル最適化目標	allclocknetsinpad _to_outputoffset _in_beforeoffset _out_aftermax _delay デフォルト： allclocknets
cross_clock_analysis	クロス クロック解析	yes no (デフォルト)
write_timing_constraints	Write Timing Constraints	yes no (デフォルト)

XCF でのみサポートされる XST タイミング制約

次の XST タイミング制約は、XST Constraint File (XCF) ファイルからのみ設定可能です。

- ・ 周期 (PERIOD)
- ・ オフセット (OFFSET)
- ・ From-To (FROM-TO)
- ・ タイミング名 (TNM)
- ・ ネットのタイミング名 (TNM_NET)
- ・ タイムグループ (TIMEGRP)
- ・ タイミング無視 (TIG)
- ・ タイミング仕様 (TIMESPEC)
- ・ タイミング仕様識別子 (TSidentifier)

これらのタイミング制約により合成最適化が影響を受けます。制約を配置配線まで渡すには、[タイミング制約の書き込み](#) コマンドライン オプションを使用します。

詳細は、『[制約ガイド](#)』を参照してください。

一般制約

この章では、次の制約について説明しています。

- ・ I/O バッファの追加 (-iobuf)
- ・ ボックス タイプ (BOX_TYPE)
- ・ バスの区切り文字指定 (-bus_delimiter)
- ・ 大文字/小文字の指定 (-case)
- ・ Case 文のインプリメンテーション形式 (-vlgcase)
- ・ 複製接尾語の設定 (-duplication_suffix)
- ・ フル ケース (FULL_CASE)
- ・ RTL 回路図の生成 (-rtlview)
- ・ ジェネリック (-generics)
- ・ HDL ライブラリ マップ ファイル (-xsthdpini)
- ・ 階層区切り文字の指定 (-hierarchy_separator)
- ・ I/O 規格 (IOSTANDARD)
- ・ キープ (KEEP)
- ・ 階層の維持 (KEEP_HIERARCHY)
- ・ ライブラリの検索順 (-lso)
- ・ LOC
- ・ ネットリスト階層 (-netlist_hierarchy)
- ・ 最適化エフォートレベル (OPT_LEVEL)
- ・ 最適化方法の指定 (OPT_MODE)
- ・ パラレル ケース (PARALLEL_CASE)
- ・ RLOC
- ・ 保存 (S / SAVE)
- ・ 合成制約ファイルの指定 (-uc)
- ・ 変換なし (TRANSLATE_OFF) と変換あり (TRANSLATE_ON)
- ・ 合成制約ファイルの使用 (-iuc)
- ・ Verilog 2001 の指定 (-verilog2001)
- ・ Verilog の 'include ディレクトリの指定 (-vlgincdir)
- ・ Verilog マクロ (-define)
- ・ 作業ディレクトリの指定 (-xsthdpdir)

I/O バッファの追加 (-iobuf)

I/O バッファの追加 (-iobuf) コマンドライン オプションには、次の特徴があります。

- ・ I/O バッファを自動的に挿入するかどうかを設定します。
- ・ 後でインスタンス化して設計の一部を合成する際に便利です。

XST では、I/O バッファが設計に自動的に挿入されます。I/O バッファは各 I/O に手動でもインスタンス化できます。その場合、I/O バッファがインスタンス化されていない I/O にのみ I/O バッファが追加されます。I/O バッファを XST で挿入しない場合は、この -iobuf オプションを no に設定します。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

設計全体に適用されます。

適用ルール

設計のプライマリ I/O に適用されます。

構文

-iobuf {yes|no|true|false|soft}

- ・ **yes** (デフォルト)
IBUF/OBUF プリミティブが生成され、最上位レベル モジュールの I/O ポートに接続されます。
- ・ **no**
IBUF および OBUF プリミティブは生成されません。大型設計の場合に後でインスタンス化される内部モジュールを合成するよう XST が呼び出される際には、このオプションを no に設定する必要があります。設計に I/O バッファを追加した場合は、この設計を別の設計のサブモジュールとして使用することはできません。
- ・ **true**
- ・ **false**
- ・ **soft**

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンドライン

xst run -iobuf yes

I/O バッファが設計の最上位レベルのモジュールに追加されます。

ISE® Design Suite

[Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックス → [Synthesis Options]
→ [Add I/O Buffers]

ボックス タイプ (BOX_TYPE)

ボックス タイプ (BOX_TYPE) は、合成制約です。

少なくともブロックの 1 つのインスタンスに BOX_TYPE 制約を設定すると、デザインすべてのインスタンスに制約が適用されます。これは Verilog および XST Constraint File (XCF) のためにインプリメントされた制約で、VHDL のように BOX_TYPE 制約をコンポーネントに適用できます。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

次のデザイン エレメントに適用されます。

- ・ VHDL
component、entity
- ・ Verilog
module、instance
- ・ XST Constraint File (XCF)
model、instance

適用ルール

設定したデザイン エレメントに適用されます。

構文

- ・ **primitive**
- ・ **black_box**
primitive と同じです。今後使用できなくなる予定なのでご注意ください。
- ・ **user_black_box**
primitive が指定されない限り、ログ ファイルにブラック ボックスの推論がレポートされます。
これらの値により、XST でモジュールのビヘイビアが合成されないようにできます。

構文例および設定

次の例では、この制約またはコマンド ライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

```
attribute box_type: string;
```

次のように指定します。

```
attribute box_type of {component_name/entity_name} : {component|entity} is  
"{primitive|black_box|user_black_box}";
```

Verilog

次をインスタンスエーションの直前に入力します。

```
(* box_type = "{primitive|black_box|user_black_box}" *)
```

XCF の構文例 1

```
MODEL "entity_name" box_type = "{primitive|black_box|user_black_box}";
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"
```

```
INST "instance_name"
```

```
box_type="{primitive|black_box|user_black_box}";
```

```
END;
```

バスの区切り文字指定 (-bus_delimiter)

出力ネットリストのバスに属する信号のフォーマットを定義します。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

構文に適用されます。

適用ルール

ありません。

構文

```
-bus_delimiter {<>|□|{}|0}
```

- ・ <> (デフォルト)
- ・ □
- ・ {}
- ・ 0

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンドライン

```
xst run -bus_delimiter []
```

バス区切り文字が角かっこ [] としてグローバルに設定されます。

ISE® Design Suite

[Process] → [Process Properties] → [Synthesis Options] → [Bus Delimiter]

大文字/小文字の指定(-case)

インスタンス名およびネット名を最終的なネットリストに記述する際に、名前に大文字を使用するか、小文字を使用するか、ソースの文字を保持するかを指定します。

ソースの文字は、Verilog または VHDL で保持できます。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

構文に適用されます。

適用ルール

ありません。

構文

```
-case {upper|lower|maintain}
```

- upper
- lower
- maintain (デフォルト)

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンドライン

```
xst run -case upper
```

この例では、大文字にグローバルに設定されています。

ISE® Design Suite

[Process] → [Process Properties] → [Synthesis Options] → [Case]

Case 文のインプリメンテーション形式 (-vlgcase)

Case 文のインプリメンテーション形式 (-vlgcase) コマンドライン オプションには、次の特徴があります。

- ・ Verilog デザインにのみ使用できます。
- ・ XST で Verilog の case 文をどのように解釈させるかが指定できます。

詳細は、次を参照してください。

- ・ [マルチプレクサの HDL コーディング手法](#)
- ・ [FULL_CASE \(フル ケース\)](#)
- ・ [PARALLEL_CASE \(パラレル ケース\)](#)

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

デザイン全体に適用されます。

適用ルール

ありません。

構文

-vlgcase {full|parallel|full-parallel}

- ・ **full**
case 文は完全と判断され、ラッチは作成されません。
- ・ **parallel**
case 文の分岐は同時に実行されないと判断され、プライオリティ エンコーダは使用されません。
- ・ **full-parallel**
case 文完全で分岐は同時に実行されないと判断され、ラッチおよびプライオリティ エンコーダは使用されません。

デフォルトでは、値は設定されていません。このオプションを指定しない場合は (Project Navigator では [None])、case 文のビヘイビアがそのままインプリメントされます。

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンド ライン

```
xst run -vlgcase full
```

case 文のインプリメンテーション形式が full にグローバルに定義されます。

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [HDL Options] → [Case Implementation Style]

複製接尾語の設定 (-duplication_suffix)

複製接尾語の設定 (-duplication_suffix) コマンドライン オプションでは、フリップフロップが複製されたときの XST の名前の付け方を設定できます。

デフォルトでは、XST でフリップフロップが複製される際、元のフリップフロップ名に _n (n は整数) が付きます。

たとえば、元のフリップフロップ名が my_ff で、同じフリップフロップが 3 度複製されると、名前はそれぞれ次のようになります。

- ・ my_ff_1
- ・ my_ff_2
- ・ my_ff_3

-duplication_suffix を使用すると、元の名前に追加される文字列を変更できます。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

ファイルに適用されます。

適用ルール

ありません。

構文

```
-duplication_suffix string%dstring
```

デフォルトは %d です。

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンド ラインの例 1

```
xst run -duplication_suffix _dupreg %d
```

このコマンドを指定すると、フリップフロップ名が `my_ff` で、同じフリップフロップが 3 度複製される場合、名前はそれぞれ次のようになります。

- ・ `my_ff_dupreg_1`
- ・ `my_ff_dupreg_2`
- ・ `my_ff_dupreg_3`

XST コマンド ラインの例 2

```
xst run -duplication_suffix _dup%d_reg
```

エスケープ文字 `%d` は接尾語の定義のどこにでも挿入できます。このコマンドを指定すると、フリップフロップ名が `my_ff` で、同じフリップフロップが 3 度複製される場合、名前はそれぞれ次のようになります。

- ・ `my_ff_dup_1_reg`
- ・ `my_ff_dup_2_reg`
- ・ `my_ff_dup_3_reg`

ISE® Design Suite

[Process] → [Process Properties] → [Synthesis Options] → [Property display level] → [Advanced] → [Other XST Command Line Options]

フル ケース (FULL_CASE)

フル ケース (FULL_CASE) 制約には、次の特徴があります。

- ・ Verilog デザインにのみ使用できます。
- ・ `case`、`casex` または `casez` 文で、セレクトのすべての値が記述されていることを示します。
- ・ 記述されていない条件がある場合にハードウェアが作成されません。

詳細は、次を参照してください。

[マルチプレクサの HDL コーディング手法](#)

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

Verilog メタ コメントの `case` 文に適用されます。

適用ルール

ありません。

構文

-vlgcase [full|parallel|full-parallel]

- ・ **full**
- ・ **parallel**
- ・ **full-parallel**

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

Verilog

構文は次のとおりです。

(* full_case *)

FULL_CASE にはターゲット参照が含まれないため、セクタのすぐ後ろに属性を記述します。

```
(* full_case *)
casex select
4'b1xxx: res = data1;
4'bx1xx: res = data2;
4'bxx1x: res = data3;
4'bxxx1: res = data4;
endcase
```

FULL_CASE は Verilog コードのメタコメントとしても使用可能です。メタコメントの構文は、次のように通常のメタコメントと異なります。

// synthesis full_case

FULL_CASE にはターゲット参照が含まれないため、セクタのすぐ後ろにメタコメントを記述します。

```
casex select // synthesis full_case
4'b1xxx: res = data1;
4'bx1xx: res = data2;
4'bxx1x: res = data3;
4'bxxx1: res = data4;
endcase
```

XST コマンドライン

xst run -vlgcase [full|parallel|full-parallel]

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Full Case] をクリックします。

[Case Implementation Style] には [Full] を選択します。

RTL 回路図の生成 (-rtlview)

RTL 回路図の生成 (-rtlview) コマンドライン オプションを使用すると、デザインの Register Transfer Level (RTL) 構造を表すネットリスト ファイルを生成できます。このネットリストは、RTL and Technology Viewers で表示できます。

RTL 表示を含むファイルの拡張子は .ngr です。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

ファイルに適用されます。

適用ルール

ありません。

構文

-rtlview {yes|no|only}

- ・ **yes**
RTL ビューが生成されます。
- ・ **no** (デフォルト)
RTL ビューは生成されません。
- ・ **only**
RTL ビューが生成されると合成が停止します。

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンドライン

xst run -rtlview yes

XST でデザインの RTL 構造を記述するネットリスト ファイルが生成されます。

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Generate RTL Schematic]

ジェネリック (-generics)

ジェネリック (-generics) コマンドライン オプションを使用すると、最上位レベルのデザイン ブロックで定義したジェネリック (VHDL) またはパラメータ (Verilog) の値を定義し直すことができます。

これにより、IP コアの生成やテストフローなどの HDL ソースを変更しなくてもデザイン コンフィギュレーションを簡単に修正できます。定義された値が VHDL または Verilog コードで定義されたデータ型と合わない場合、XST でコマンドラインの定義が無視されることを示す警告メッセージが表示されます。

データ型の違いが XST で認識されなかった場合は、警告メッセージは表示されず、VHDL または Verilog ファイルで定義したデータ型で値が処理されます。指定する値は VHDL または Verilog で定義されたデータ型と一致するようにしてください。定義されたジェネリックまたはパラメータの名前がデザインに存在しない場合は、何のメッセージも表示されず、その定義は無視されます。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

デザイン全体に適用されます。

適用ルール

ありません。

構文

```
-generics {name=valuename=value ...}
```

説明：

- ・ *name* は、最上位レベルのデザイン ブロックのジェネリックまたはパラメータの名前を示します。
- ・ *value* 最上位レベルのデザイン ブロックのジェネリックまたはパラメータの値を示します。

デフォルトでは何も定義されていません。

次の規則に従ってください。

- ・ {} 内に値を挿入します。
- ・ 各値はスペースで区切ります。
- ・ XST にはスカラ型の定数しか値として使用できません。複合データ型 (アレイまたはレコード) は次の場合にしか使用できません。
 - **string**
 - **std_logic_vector**
 - **std_ulogic_vector**
 - **signed, unsigned**
 - **bit_vector**
- ・ 接頭語とその値の間にはスペースを入れないでください。

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンドライン

```
xst run -generics {company="Xilinx" width=5 init_vector=b100101}
```

このコマンドは、次を設定しています。

- ・ company を Xilinx に設定
- ・ width を 5 に設定
- ・ init_vector を b100101 に設定

ISE® Design Suite

[Process] → [Process Properties] → [Synthesis Options] → [Generics, Parameters]

HDL ライブラリ マップ ファイル (-xsthdpi)

HDL ライブラリ マップ ファイル (-xsthdpi) コマンドライン オプションを使用すると、ライブラリのマップに使用するファイルを選択できます。

XST では、次の 2 つのライブラリ マップ ファイルが維持されます。

- ・ INI ファイル (デフォルト)：ザイリンクスのソフトウェアをインストールするとインストールされます。
- ・ ユーザー ファイル：ユーザーがプロジェクトに合わせて定義できます。

インストール済みの INI ファイル (デフォルト)：

- ・ ファイル名は xhdp.ini です。
- ・ %XILINX%\vhdl\xst に含まれます。
- ・ 標準 VHDL および UNISIM ライブラリの場所に関する情報が含まれます。
- ・ 修正できません。

メモ： ライブラリ マップ ファイルの構文はコピーできます。

ライブラリ マップ ファイルは次のようになっています。

```
-- Default lib mapping for XST std=$XILINX/vhdl/xst/std
ieee=$XILINX/vhdl/xst/unisim unisim=$XILINX/vhdl/xst/unisim
aim=$XILINX/vhdl/xst/aim pls=$XILINX/vhdl/xst/pls
```

このファイルのフォーマットを使用して、独自のライブラリの場所を定義できます。デフォルトでは、コンパイルされた VHDL ファイルはプロジェクト ディレクトリの xst サブディレクトリに保存されます。

このライブラリ マップ ファイルには、次の情報別にライブラリがリストされています。

- ・ ライブラリ名
- ・ ライブラリがコンパイルされたディレクトリ名

このライブラリ マップ ファイルの名前は任意ですが、拡張子は .ini にすることをお勧めします。

ファイルのフォーマットは次のとおりです。

library_name=path_to_compiled_directory

コメント行には -- を使用します。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

ファイルに適用されます。

適用ルール

ありません。

構文

-xsthdpini *file_name*

ライブラリ マップ ファイルは 1 つだけしか指定できません。

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンド ライン

```
xst set -xsthdpini c:/data/my_libraries/my.ini file_name
```

ライブラリすべてをポイントするファイルに c:/data/my_libraries/my.ini を指定しています。

run コマンドを実行する前に、この set コマンドを実行する必要があります。

次に MY.INI 例のテキストを示します。

```
work1=H:\Users\conf\my_lib\work1 work2=C:\mylib\work2
```

ISE® Design Suite

ISE Design Suite からライブラリ マップ ファイルを設定するには

1. [Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックスを表示して、[Synthesis Options] をクリックします。
2. [Process Properties] ダイアログ ボックスで [Property display level] → [Advanced] をクリックします。
3. [HDL INI File] プロパティを設定します。

階層区切り文字の指定 (-hierarchy_separator)

階層区切り文字の指定 (-hierarchy_separator) コマンドライン オプションを使用すると、デザイン階層をフラット化した際の名前の生成で使用する階層の区切り文字を指定できます。

たとえば、デザインにインスタンス INST1 というサブブロックがあり、このサブブロックに TMP_NET というネットが含まれているとします。階層がフラット化される場合に、このオプションがアンダースコアに設定されていると、TMP_NET は INST1_TMP_NET という名前になります。このオプションがスラッシュに設定されている場合は、ネット名は INST1/TMP_NET となります。

階層の区切り文字としてスラッシュを使用した方が階層名を識別しやすいので、デバッグの際に便利です。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

ファイルに適用されます。

適用ルール

ありません。

構文

```
-hierarchy_separator {/ / }
```

サポートされる文字は次の 2 つです。

- ・ `_` (アンダースコア)
- ・ `/` (スラッシュ)

新規プロジェクトでのデフォルトは `/` (スラッシュ) です。

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンドライン

```
xst run -hierarchy_separator _
```

階層区切り文字を `_` (アンダースコア) に設定しています。

ISE® Design Suite

1. [Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックスを表示して、[Synthesis Options] をクリックします。
2. [Process Properties] ダイアログ ボックスで [Property display level] → [Advanced] をクリックします。
3. [Hierarchy Separator] プロパティを設定します。

I/O 規格 (IOSTANDARD)

I/O プリミティブに I/O 規格を割り当てるために使用します。

この制約の詳細は、『[制約ガイド](#)』を参照してください。

キープ (KEEP)

KEEP 制約は、高度なマップ制約です。

デザインのマップ時に、一部のネットが論理ブロックに含まれることがあります。ブロックに含まれたネットは、物理的なデザイン データベースには存在しなくなります。これは、ネットの各サイドに接続されたコンポーネントが同じ論理ブロックにマップされる場合などに発生することがあります。この後、ネットはそのコンポーネントを含むブロックに含まれます。KEEP を使用すると、これが回避できます。

インプリメンテーション フローでサポートされる値には、true と false のほかに、soft もあります。この値が指定されると、XST は true の場合と同じように該当するネットを保持しますが、最終のネットリストではこのネットに KEEP 制約は付けません。

KEEP 制約を使用すると、最終ネットリストに信号は保持されますが、構造が保持されるわけではありません。たとえば、デザインに 2 ビットのマルチプレクサ セレクタがあり、これに KEEP 制約を設定した場合、この信号は最終ネットリストに保持されますが、このマルチプレクサが XST によりワンホット エンコードを使用して再エンコードされていると、元の 2 ビットではなく 4 ビット幅になります。信号の構造を保持するには、KEEP 制約だけでなく、[列挙型エンコード手法 \(ENUM_ENCODING\)](#) も使用する必要があります。

この制約の詳細は、『[制約ガイド](#)』を参照してください。

階層の維持 (KEEP_HIERARCHY)

KEEP_HIERARCHY は、合成およびインプリメンテーション制約です。

デザイン階層が合成で保持された場合、インプリメンテーションでもこの制約を使用して階層が保持され、この階層を含むシミュレーション ネットリストが生成されます。

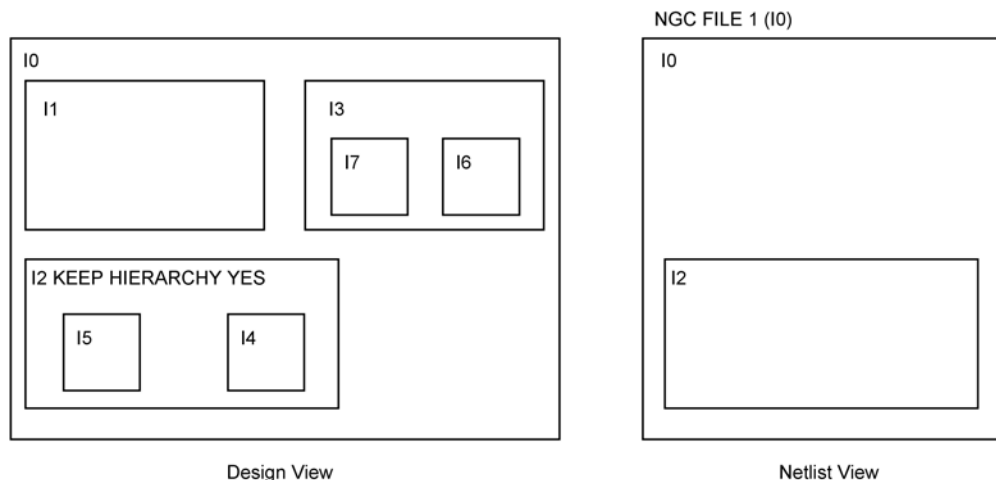
XST では、最適な結果を得るために、エンティティおよびモジュールの境界を最適化してデザインをフラットにすることがあります。この制約を true (有効) に設定すると、階層ネットリストを作成でき、デザインのエンティティとモジュールの階層およびインターフェイスを保持できます。

この制約は、HDL 合成で推論されたマクロではなく、HDL デザインで指定された階層ブロック (VHDL のエンティティ、Verilog のモジュール) に適用されます。

一般的に、HDL デザインは階層ブロックの集合です。より単純な階層で個別に最適化が行われるため、デザイン階層を保持すると処理速度が向上します。また、コラプスや因数分解などの最適化のプロセスはロジック全体にグローバルに適用されるため、階層ブロックのマージによりフィットの結果が向上します (積項およびデバイス マクロセルの少量化、周波数の向上など)。

たとえば、エンティティまたはモジュール I2 に KEEP_HIERARCHY 制約を設定した場合、I2 の階層は維持されたままで最後のネットリストに含まれますが、I2 の下にある I4 および I5 はフラットになります。また、I1、I3、I6、I7 も同様にフラット化されます。

階層維持の図



X9542

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

階層ブロックまたはシンボル ブロックを含む論理ブロックに適用します。

適用ルール

設定したエンティティまたはモジュールに適用されます。

構文

-keep_hierarchy {yes|no|soft}

- ・ **yes**
- ・ **no**
- ・ **true**

HDL プロジェクトで記述されたデザイン階層を保持します。この値を合成に適用した場合、インプリメンテーションにも適用されます。

CPLD デバイスの場合、デフォルトは true です。

- ・ **false**

階層ブロックが 最上位モジュールにマージされます。

FPGA デバイスの場合、デフォルトは false です。

- ・ **soft**

合成ではデザイン階層が保持されますが、インプリメンテーションには制約は適用されません。

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

回路図の例

- ・ エンティティまたはモジュール シンボルに設定
- ・ 属性名
KEEP_HIERARCHY
- ・ 属性値
前述の「構文」セクションを参照してください。

VHDL

次のように宣言します。

attribute keep_hierarchy : string;

次のように指定します。

attribute keep_hierarchy of *architecture_name*: architecture is "{yes|no|true|false|soft}";

Verilog

次をモジュール宣言またはインスタンス化の直前に入力します。

(* keep_hierarchy = "{yes|no|true|false|soft}" *)

XCF

MODEL "*entity_name*" keep_hierarchy={yes|no|true|false|soft};

XST コマンド ライン

xst run -keep_hierarchy {yes|no|soft}

詳細は、次を参照してください。

[コマンド ライン モード](#)

ISE® Design Suite

[Process] → [Process Properties] → [Synthesis Options] → [Keep Hierarchy]

ライブラリの検索順 (-lso)

ライブラリ検索順 (-lso) コマンド ライン オプションを使用すると、ライブラリ ファイルを使用する順序を指定できます。

詳細は、次を参照してください。

[混合言語プロジェクトのライブラリ検索順 \(LSO\) ファイル](#)

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

ファイルに適用されます。

適用ルール

ありません。

構文

-lso *file_name.lso*

デフォルトのファイル名はありません。このオプションを指定しない場合は、デフォルトのライブラリ検索順が使用されます。

構文例および設定

次の例では、この制約またはコマンド ライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンド ライン

xst elaborate -lso c:/data/my_libraries/my.lso

ライブラリ検索順序を設定するファイルに c:/data/my_libraries/my.lso を指定しています。

ISE® Design Suite

ISE Design Suite からライブラリ検索順 (LSO) ファイルを指定するには

1. [Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックスを表示して、[Synthesis Options] をクリックします。
2. [Process Properties] ダイアログ ボックスで [Property display level] → [Advanced] をクリックします。
3. [Library Search Order] プロパティを設定します。

LOC

LOC 制約は、FPGA/CPLD 内でデザイン エLEMENTを配置する位置 (ロケーション) を定義します。

この制約の詳細は、『[制約ガイド](#)』を参照してください。

ネットリスト階層 (-netlist_hierarchy)

ネットリスト階層 (-netlist_hierarchy) コマンドライン オプションには、次の特徴があります。

- ・ 最終の NGC ネットリスト ファイルが生成される形式を制御できます。
- ・ 最適化が一部のみ終了している場合や、デザインが完全にフラット化された場合にも、階層ネットリストを書き出すことができます。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体に適用されます。

適用ルール

ありません。

構文

- netlist_hierarchy {as_optimized|rebuilt}

- ・ **as_optimized** (デフォルト)

XST では、[階層の維持 \(KEEP_HIERARCHY\)](#) 制約が考慮され、NGC ネットリストを最適化された形式で生成します。このモードの場合、階層ブロックはフラット化できるものもあれば、階層バウンダリを維持したままでフラット化できないものもあります。

- ・ **rebuilt**

XST では、[階層の維持 \(KEEP_HIERARCHY\)](#) 制約に関係なく、階層的な NGC ネットリストが書き出されます。

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンド ライン

- netlist_hierarchy {as_optimized|rebuilt}

ISE® Design Suite

1. [Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックスを表示して、[Synthesis Options] をクリックします。
2. [Process Properties] ダイアログ ボックスで [Property display level] → [Advanced] をクリックします。
3. [Netlist Hierarchy] プロパティを設定します。

最適化エフォート レベル (OPT_LEVEL)

合成最適化のエフォート レベルを指定します。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

デザイン全体、エンティティ、モジュールに適用されます。

適用ルール

設定したエンティティ、コンポーネント、モジュール、または信号に適用されます。

構文

-opt_level {1|2}

- ・ 1 (標準の最適化) (デフォルト)
特に階層デザインで、処理が高速になります。ザイリンクスでは、ほとんどのデザインにこのレベル 1 (標準) を推奨しています。
- ・ 2 (高度な最適化)
時間はかかりますが、スライス/マクロセルの数や最大周波数で良い結果が得られます。2 を選択すると、合成のランタイムが長引きます。また、常に最適の結果が得られるとは限りません。

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

```
attribute opt_level: string;
```

次のように指定します。

```
attribute opt_level of entity_name: entity is "{1|2}";
```

Verilog

次をモジュール宣言またはインスタンス化の直前に入力します。

```
(* opt_level = "{1|2}" *)
```

XCF

```
MODEL "entity_name" opt_level={1|2};
```

XST コマンド ライン

```
xst run -opt_level {1|2}
```

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Optimization Effort]

最適化方法の指定 (OPT_MODE)

合成の最適化方法を指定します。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

デザイン全体、エンティティ、モジュールに適用されます。

適用ルール

設定したエンティティまたはモジュールに適用されます。

構文

```
-opt_mode {area|speed}
```

- ・ **speed** (デフォルト)

ロジックレベル数の低減を優先するため、周波数が向上します。

- ・ **area**

デザインのインプリメンテーションに使用されるロジックの総数を低減することが優先されるため、デザイン フィットが向上します。

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

```
attribute opt_mode: string;
```

次のように指定します。

```
attribute opt_mode of entity_name: entity is "{speed|area}";
```

Verilog

次をモジュール宣言またはインスタンス化の直前に入力します。

```
(* opt_mode = "{speed|area}" *)
```

XCF

```
MODEL "entity_name" opt_mode={speed|area};
```

XST コマンドライン

```
xst run -opt_mode {area|speed}
```

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Optimization Goal]

パラレル ケース (PARALLEL_CASE)

パラレル ケース (PARALLEL_CASE) 制約には、次の特徴があります。

- ・ Verilog デザインにのみ使用できます。
- ・ case 文がパラレル マルチプレクサとして合成されるよう指定します。
- ・ case 文が優先順位付きのカスケードされた if-elsif 文に変換されないようにします。

詳細は、次を参照してください。

[マルチプレクサの HDL コーディング手法](#)

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

Verilog メタ コメントの case 文に適用されます。

適用ルール

ありません。

構文

-vlgcase {full|parallel|full-parallel}

- ・ **full**
- ・ **parallel**
- ・ **full-parallel**

構文例および設定

次の例では、この制約またはコマンド ライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

Verilog

(* parallel_case *)

PARALLEL_CASE にはターゲット参照が含まれないため、セレクタのすぐ後ろに属性を記述します。

```
(* parallel_case *)
casex select
4'b1xxx: res = data1;
4'bx1xx: res = data2;
4'bxx1x: res = data3;
4'bxxx1: res = data4;
endcase
```

PARALLEL_CASE は Verilog コードのメタ コメントとしても使用可能です。メタ コメントの構文は、次のように通常のメタ コメントと異なります。

// synthesis parallel_case

PARALLEL_CASE にはターゲット参照が含まれないため、セレクタのすぐ後ろにメタ コメントを記述します。

```
casex select // synthesis parallel_case
4'b1xxx: res = data1;
4'bx1xx: res = data2;
4'bxx1x: res = data3;
4'bxxx1: res = data4;
endcase
```

XST コマンド ライン

xst run -vlgcase {full|parallel|full-parallel}

RLOC (RLOC)

RLOC 制約には、次の特徴があります。

- ・ 基本的なマップおよび配置制約です。
- ・ ロジック エlementを独立した集合にグループ化します。
- ・ デザイン全体での最終的な配置に関係なく、グループ内でのElement同士の位置を相対的に定義できます。

この制約の詳細は、『[制約ガイド](#)』を参照してください。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

Verilog

たとえば、sr11 という名前の SRL16 インスタンスを R9C0.S0 に配置する場合、Verilog コードで次のように指定します。

```
// synthesis attribute RLOC of sr11 : "R9C0.S0";
```

XCF

同じ属性は XST Constraint File (XCF) では次のように指定できます。

```
BEGIN MODEL ENTNAME
```

```
INST sr11 RLOC=R9C0.S0;
```

```
END;
```

このように指定すると、次の文と同等のバイナリコードが NGC ファイルに記述されます。

```
INST sr11 RLOC=R9C0.S0;
```

保存 (S)

S 制約は、高度なマップ制約です。

デザインのマップ時に、一部のネットが論理ブロックに含まれたり、LUT のようなElementが最適化で削除されることがあります。このようにブロックに含まれたネットや削除されたネットは、物理的なデザイン データベースには存在しなくなります。S 制約を使用すると、これが回避できます。また、ネットやブロックの複製、レジスタの自動調整といった最適化手法にも S 制約によってオフにできるものがあります。

S 制約がネットに適用されると、そのネットは直接接続されたElementすべてと共に最終ネットリストに保存されます。これらのElementに接続されたネットも保存されます。

S 制約が LUT のようなブロックに適用されると、その LUT は直接接続された信号すべてと共に保存されます。

この制約の詳細は、『[制約ガイド](#)』を参照してください。

合成制約ファイルの指定 (-uc)

合成制約ファイルの指定 (-uc) コマンドライン オプションを使用すると、合成で使用する XST 合成制約ファイル (XCF) を指定できます。

この制約ファイルの拡張子は .xcf です。拡張子が異なるファイルを指定するとエラーが発生し、プロセスが中止されます。

詳細は、次を参照してください。

[XST 制約ファイル \(XCF\)](#)

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

ファイルに適用されます。

適用ルール

ありません。

構文

-uc *filename*

filename is the only value.

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンドライン

xst run -uc my_constraints.xcf

このプロジェクトの制約ファイルに my_constraints.xcf を指定しています。

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Synthesis Constraints File]

変換なし (TRANSLATE_OFF) と変換あり (TRANSLATE_ON)

変換なし (TRANSLATE_OFF) 制約と変換あり (TRANSLATE_ON) 制約には、次の特徴があります。

- ・ シミュレーション コードなど、合成に関係のない VHDL または Verilog コードを無視するよう指定できます。
- ・ Synopsys 制約で、XST では Verilog でサポートされています。自動変換も VHDL および Verilog の両方で使用できます。
- ・ 次のワードと共に使用できます。
 - synthesis
 - Synopsys
 - pragma
- ・ 次のように指定します。
 - TRANSLATE_OFF: 無視するセクションの冒頭を指定
 - TRANSLATE_ON: 合成を再開する点を指定

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

ローカルに適用されます。

適用ルール

合成でコードの一部を有効または無効にするよう指定します。

構文

次のセクションでは、この制約の構文を示します。

構文例および設定

次の例では、この制約またはコマンド ライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

```
-- synthesis translate_off
...code not synthesized...
-- synthesis translate_on
```

Verilog

ただし、Verilog 構文は、前述したような通常のメタ コメント構文とは異なり、次のようになります。

```
// synthesis translate_off
...code not synthesized...
// synthesis translate_on
```

合成制約ファイルの無視 (-iuc)

合成制約ファイルの無視 (-iuc) コマンドライン オプションを使用すると、合成制約ファイル (-uc) で指定した制約ファイルが合成中に無視されます。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

ファイルに適用されます。

適用ルール

ありません。

構文

`-iuc {yes|no}`

- ・ **yes**
- ・ **no** (デフォルト)

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンドライン

`xst run -iuc yes`

ISE Design Suite

注意： この制約は、ISE® Design Suite では [Synthesis Constraints File] と表示されます。制約ファイルはこのオプションをオフにすると無視されます。デフォルトではオンになっており (コマンドライン オプションの `-iuc no` と同じ)、指定した合成制約ファイルが考慮されます。

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Use Synthesis Constraints File]

Verilog 2001 の指定 (-verilog2001)

Verilog 2001 の指定 (-verilog2001) コマンドライン オプションを使用すると、Verilog ソースコードを Verilog 2001 規格として解釈するかどうかを指定できます。

デフォルトでは、Verilog ソースコードは Verilog 2001 規格として解釈されます。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

構文に適用されます。

適用ルール

ありません。

構文

`–verilog2001 {yes|no}`

- ・ `yes` (デフォルト)
- ・ `no`

構文例および設定

次の例では、この制約またはコマンド ライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンド ライン

`xst elaborate –verilog2001 no`

XST では、Verilog コードが Verilog 2001 規格に従って解釈されません。

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Verilog 2001]

Verilog の ‘include ディレクトリの指定 (–vlgincdir)

Verilog の ‘include ディレクトリの指定 (–vlgincdir) コマンド ライン オプションを使用すると、‘include 文で参照されるファイルをパーサーがを見つけやすくなります。

‘include 文によりファイルが参照されると、XST では次の順序でさまざまなディレクトリを検索します。

- ・ 現在のディレクトリ
- ・ inc ディレクトリ
- ・ 現在のファイルの相対ディレクトリ

メモ： –vlgincdir は、‘include と共に使用してください。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

ディレクトリに適用されます。

適用ルール

ありません。

構文

```
-vlgincdir { directory_path [directory_path]
```

説明：

directory_path はディレクトリ名です。

詳細は、次を参照してください。

[コマンドライン モードでのスペースを含む名前](#)

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンドライン

```
xst elaborate -vlgincdir c:/my_verilog
```

XST がファイルを検索するディレクトリのリストに c:/my_verilog を追加します。

ISE® Design Suite

[Process] → [Process Properties] → [Synthesis Options] → [Property display level] → [Advanced] → [Verilog Include Directories]

Verilog マクロ (-define)

Verilog マクロ (-define) コマンドライン オプションには、次の特徴があります。

- ・ Verilog デザインにのみ使用できます。
- ・ Verilog マクロを定義または再定義できます。

これにより、IP コアの生成やテストベンチフローなどのソースコードを変更しなくてもデザインコンフィギュレーションを簡単に修正できます。定義されたマクロがデザインで使用されていない場合は、何のメッセージも表示されません。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

デザイン全体に適用されます。

適用ルール

ありません。

構文

```
-define {name[=value] name[=value]}
```

説明：

- ・ *name* はマクロ名
- ・ *value* はマクロ テキスト

デフォルトでは何も定義されていません。

メモ：

- ・ マクロの値は必須ではありません。
- ・ {} 内に値を挿入します。
- ・ 各値はスペースで区切ります。
- ・ マクロ テキストは、二重引用符 (") で囲むか、そのまま指定します。ただし、マクロ テキストにスペースが含まれる場合は、必ず二重引用符を使用してください。

```
-define {macro1=Xilinx macro2="Xilinx Virtex4"}
```

構文例および設定

次の例では、この制約またはコマンド ライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンド ライン

```
xst run -define macro1=Xilinx macro2="Xilinx Virtex4"
```

macro1 および macro2 という名前の 2 つのマクロが定義されます。

ISE Design Suite

ISE® Design Suite で Verilog マクロを定義するには

1. [Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックスを表示して、[Synthesis Options] をクリックします。
2. [Process Properties] ダイアログ ボックスで [Property display level] → [Advanced] をクリックします。
3. [Verilog Macros] プロパティを設定します。

ISE Design Suite で値を指定する場合は、{} は使用しないでください。

作業ディレクトリの指定 (-xsthdpdir)

ライブラリ マップ ファイルで場所が指定されていない場合に、コンパイルされたファイルを保存する場所を指定します。

このプロパティは、次のいずれかの方法で設定します。

- ・ ISE® Design Suite で次から設定します。
[Process] → [Process Properties] → [Synthesis Options] → [VHDL Work Directory]
- ・ スタンドアロン モードの場合は、次のコマンドを実行します。

set -xsthdpdir *directory*

この例は、次の条件下にあります。

- ・ 3 人のユーザーが同じプロジェクトを作業しています。
- ・ あらかじめコンパイルされた shlib という標準ライブラリを共有しています。
- ・ このライブラリには、プロジェクトで使用するマクロ ブロックが含まれています。
- ・ 各ユーザーは、それぞれローカルにも作業ライブラリを持っています。
- ・ ユーザー 3 はこれをプロジェクト ディレクトリ以外の場所 (c:\temp) に保存しています。
- ・ ユーザー 1 と 2 は、上記以外にライブラリ lib12 を共有していますが、ユーザー 3 とは共有していません。

この場合、この 3 人のユーザーは次を設定する必要があります。

ユーザー 1 の例

マップ ファイル

```
schlib=z:\sharedlibs\shlib lib12=z:\userlibs\lib12
```

ユーザー 2 の例

マップ ファイル

```
schlib=z:\sharedlibs\shlib lib12=z:\userlibs\lib12
```

ユーザー 3 の例

マップ ファイル

```
schlib=z:\sharedlibs\shlib
```

ユーザー 3 は、次も設定する必要があります。

```
XSTHDPDIR = c:\temp
```

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

ディレクトリに適用されます。

適用ルール

ありません。

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンドライン

run コマンドを実行する前に、**set -xsthdpdir** を使用して work ディレクトリをグローバルに定義します。

set -xsthdpdir *directory*

このコマンドで指定できるパスは 1 つのみです。使用するディレクトリを指定します。デフォルト値はありません。

ISE Design Suite

[Process] → [Process Properties] → [Synthesis Options] → [VHDL Work Directory]

このプロパティは、次の設定で表示されます。

[Edit] → [Preferences] → [Processes] → [Property display level] → [Advanced]

HDL 制約

次の HDL 制約は、ISE® Design Suite の [Process Properties] ダイアログ ボックスの [HDL Options] ページでグローバルに設定できます。

- ・ FSM 自動抽出 (FSM_EXTRACT)
- ・ 等価レジスタの削除 (EQUIVALENT_REGISTER_REMOVAL)
- ・ FSM エンコード方法の指定 (FSM_ENCODING)
- ・ MUX の抽出 (MUX_EXTRACT)
- ・ リソース共有 (RESOURCE_SHARING)
- ・ セーフ インプリメンテーション (SAFE_IMPLEMENTATION)

次の HDL 制約は、[Process] → [Process Properties] で設定できません。

- ・ 列挙型エンコード手法 (ENUM_ENCODING)
- ・ セーフ リカバリ ステート (SAFE_RECOVERY_STATE)
- ・ 信号のエンコード方法 (SIGNAL_ENCODING)

FSM 自動抽出 (FSM_EXTRACT)

FSM 自動抽出 (FSM_EXTRACT) 制約には、次の特徴があります。

- ・ 有限ステート マシンの抽出、および特定の合成の最適化オプションを有効または無効にできます。
- ・ [FSM エンコーディング方法の指定 \(FSM_ENCODING\)](#) の値を設定するために、オンにする必要があります。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

デザイン全体、エンティティ、コンポーネント、モジュール、信号に適用されます。

適用ルール

設定したエンティティ、コンポーネント、モジュール、または信号に適用されます。

構文

-fsm_extract {yes|no}

- ・ **yes** (デフォルト)
- ・ **no**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

```
attribute fsm_extract: string;
```

次のように指定します。

```
attribute fsm_extract of {entity_name/signal_name} : {entity/signal} is "{yes|no}";
```

Verilog

次をモジュールまたは信号宣言の直前に入力します。

```
(* fsm_extract = "{yes|no}" *)
```

XCF の構文例 1

```
MODEL "entity_name" fsm_extract={yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
NET "signal_name" fsm_extract={yes|no|true|false};  
END;
```

XST コマンド ライン

```
xst run -fsm_extract {yes|no}
```

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [HDL Options] → [FSM Encoding Algorithm]

このオプションは、-fsm_extract と **FSM スタイル (FSM_STYLE)** の両方を定義します。

- ・ **FSM エンコード方法の指定 (FSM_ENCODING)** を none に設定した場合：
 - fsm_extract は no に設定されます。
 - fsm_encoding は関係ないので、指定しないままにします。
- ・ **FSM エンコード方法の指定 (FSM_ENCODING)** をその他の値に設定した場合：
 - fsm_extract は no に設定されます。
 - fsm_encoding は選択した値に設定されます。

-fsm_encoding の詳細については、次を参照してください。

FSM エンコード方法の指定 (FSM_ENCODING)

列挙型エンコード手法 (ENUM_ENCODING)

ENUM_ENCODING (列挙型エンコード手法) 制約には、次のような特徴があります。

- ・ VHDL 列挙型に適用するエンコード手法を選択できます。属性値は、空白で区切られたバイナリ コードを含む文字列です。
- ・ 列挙タイプで VHDL 制約としてのみ指定できます。

ステートレジスタの列挙タイプを使用して Finite State Machine (FSM) を記述する場合、ENUM_ENCODING でエンコード方法を指定する必要があります。指定したエンコードが XST で使用されるようにするには、ステートレジスタの **FSM エンコード 方法 (FSM_ENCODING)** を user に設定する必要があります。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

信号または型に適用されます。

ENUM_ENCODING は外部デザインのインターフェイスを保持する必要があるので、ポートに設定された場合は XST で無視されます。

適用ルール

設定したタイプまたは信号に適用されます。

構文

次のセクションでは、この制約の構文を示します。

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次の例に示すように、VHDL 制約として列挙型に指定します。

```
...
...
architecture behavior of example is
type statetype is (ST0, ST1, ST2, ST3);
attribute enum_encoding : string;
attribute enum_encoding of statetype : type is "001 010 100 111";
signal state1 : statetype;
signal state2 : statetype;
begin
...
```

XCF

```
BEGIN MODEL "entity_name"
NET "signal_name" enum_encoding="string";
END;
```

等価レジスタの削除 (EQUIVALENT_REGISTER_REMOVAL)

RTL レベルで記述された等価レジスタの削除を有効または無効にします。

デフォルトでは、ザイリンクス プリミティブ ライブラリからインスタンス化された等価フリップフロップは削除されません。

フリップフロップの最適化では、次が削除されます。

- ・ FPGA および CPLD の等価フリップフロップ
- ・ CPLD の定数入力を使用するフリップフロップ

フリップフロップを削除することでロジックが単純化され、フィットが向上します。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

デザイン全体、エンティティ、コンポーネント、モジュール、信号に適用されます。

適用ルール

同等フリップフロップおよび定数入力を使用したフリップフロップを削除します。

構文

-equivalent_register_removal {yes|no}

- ・ **yes** (デフォルト)
- ・ **no**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

値が yes に設定されると、フリップフロップ最適化が実行できるようになります。

値が no に設定されると、フリップフロップ最適化が禁止されます。

Tip フリップフロップの最適化アルゴリズムには時間がかかります。高速処理が必要な場合は、no に設定してください。

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

```
attribute equivalent_register_removal: string;
```

次のように指定します。

```
attribute equivalent_register_removal of {entity_name|signal_name} : {signal|entity} is "{yes|no}";
```

Verilog

次をモジュールまたは信号宣言の直前に入力します。

```
(* equivalent_register_removal = "{yes|no}" *)
```

XCF の構文例 1

```
MODEL "entity_name" equivalent_register_removal={yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"
```

```
NET "signal_name" equivalent_register_removal={yes|no|true|false};
```

END;

XST コマンド ライン

`xst run -equivalent_register_removal {yes|no}`

ISE® Design Suite

[Process] → [Process Properties] → [Synthesis Options] → [Equivalent Register Removal]

FSM エンコード方法の指定 (FSM_ENCODING)

FSM_ENCODING 制約を使用すると、FSM のコーディング手法を選択できます。

このプロパティを設定するには、[FSM 自動抽出 \(FSM_EXTRACTION\)](#) をオンにしておく必要があります。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

デザイン全体、エンティティ、コンポーネント、モジュール、信号に適用されます。

適用ルール

設定したエンティティ、コンポーネント、モジュール、または信号に適用されます。

構文

`-fsm_encoding {auto|one-hot|compact|sequential|gray|johnson|speed1|user}`

- ・ **auto** (デフォルト)
各ステート マシンに最適なコーディング手法が自動的に選択されます。
- ・ **one-hot**
- ・ **compact**
- ・ **sequential**
- ・ **gray**
- ・ **johnson**
- ・ **speed1**
- ・ **user**

構文例および設定

次の例では、この制約またはコマンド ライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

```
attribute fsm_encoding: string;
```

次のように指定します。

```
attribute fsm_encoding of {entity_name|signal_name}: {entity|signal} is
  "{auto|one-hot|compact|sequential|gray|johnson|speed1|user}";
```

Verilog

次をモジュールまたは信号宣言の直前に入力します。

```
(* fsm_encoding = "{auto|one-hot|compact|sequential|gray|johnson|speed1|user}" *)
```

XCF の構文例 1

```
MODEL "entity_name"
  " fsm_encoding={auto|one-hot|compact|sequential|gray|johnson|speed1|user} ;
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"
  NET "signal_name" fsm_encoding={auto |one-hot|compact|sequential|gray|johnson|speed1|user
  };
END;
```

XST コマンド ライン

```
run xst -fsm_encoding {auto|one-hot|compact|sequential|gray|johnson|speed1|user}
```

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [HDL Options] → [FSM Encoding Algorithm]

オプションは次のとおりです。

- ・ [None] を選択すると、-fsm_extract は no に設定され、-fsm_encoding の設定は合成には関係なくなります。
- ・ その他の場合は、-fsm_extract は yes に設定され、-fsm_encoding はメニューで選択した値に設定されます。

詳細は、次を参照してください。

[FSM 自動抽出 \(FSM_EXTRACT\)](#)

MUX の抽出 (MUX_EXTRACT)

マルチプレクサ マクロの推論を有効または無効にします。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

デザイン全体、エンティティ、コンポーネント、モジュール、信号に適用されます。

適用ルール

設定したエンティティ、コンポーネント、モジュール、または信号に適用されます。

構文

-mux_extract {yes|no|force}

- ・ **yes** (デフォルト)
- ・ **no**
- ・ **force**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

デフォルトでは yes に設定され、マルチプレクサ推論は有効になっています。XST では、各マルチプレクサの記述に対し、内部決定ルールに従ってマクロが作成されるか、または残りのロジックと共に最適化されます。force に設定すると、この内部ルールが無視され、マクロが作成されます。

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

```
attribute mux_extract: string;
```

次のように指定します。

```
attribute mux_extract of {signal_name|entity_name}: {entity|signal} is "{yes|no|force}";
```

Verilog

次をモジュールまたは信号宣言の直前に入力します。

```
(* mux_extract = "{yes|no|force}" *)
```

XCF の構文例 1

```
MODEL "entity_name" mux_extract={yes|no|true|false|force};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"
```

```
NET "signal_name" mux_extract={yes|no|true|false|force};
```

```
END;
```

XST コマンドライン

```
xst run -mux_extract {yes|no|force}
```

ISE® Design Suite

[Process] → [Process Properties] → [HDL Options]

リソース共有 (RESOURCE_SHARING)

数値演算子のリソース共有を有効または無効にします。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

デザイン全体またはデザイン エレメントに適用されます。

適用ルール

設定したエンティティ、コンポーネント、モジュール、または信号に適用されます。

構文

`-resource_sharing {yes|no}`

- ・ **yes** (デフォルト)
- ・ **no**
- ・ **force**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

構文例および設定

次の例では、この制約またはコマンド ライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

```
attribute resource_sharing: string;
```

次のように指定します。

```
attribute resource_sharing of entity_name: entity is "{yes|no}";
```

Verilog

次をモジュール宣言またはインスタンスエーションの直前に入力します。

```
(* resource_sharing = "{yes|no}" *)
```

XCF の構文例 1

```
MODEL "entity_name" resource_sharing={yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
NET "signal_name" resource_sharing={yes|no|true|false};  
END;
```

XST コマンド ライン

```
xst run -resource_sharing {yes|no}
```

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [HDL Options] → [Resource Sharing]

セーフ インプリメンテーション (SAFE_IMPLEMENTATION)

SAFE_IMPLEMENTATION を使用すると、有限ステート マシン (FSM) をセーフ インプリメンテーション モードでインプリメントできます。

このモードでは、FSM が無効なステートになった場合に有効なステート (リカバリ ステート) に戻すロジックが追加されます。デフォルトでは、リカバリ ステートとして reset が選択されます。FSM に初期信号が含まれていない場合は、power-up が選択されます。

[セーフ リカバリ ステート \(SAFE_RECOVERY_STATE\)](#) 制約を適用すると、手動でリカバリ ステートを定義できます。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

次に適用されます。

- ・ XST コマンド ラインからデザイン全体に適用
- ・ 特定ブロック (entity、architecture、component)
- ・ 1 つの信号

適用ルール

設定したエンティティ、コンポーネント、モジュール、または信号に適用されます。

構文

```
-safe_implementation {yes|no}
```

- ・ yes
- ・ no (デフォルト)

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

```
attribute safe_implementation: string;
```

次のように指定します。

```
attribute safe_implementation of {entity_name/component_name/signal_name} :  
{entity/component/signal} is "{yes|no}";
```

Verilog

次をモジュールまたは信号宣言の直前に入力します。

```
(* safe_implementation = "{yes|no}" *)
```

XCF の構文例 1

```
MODEL "entity_name" safe_implementation={yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
NET "signal_name" safe_implementation={yes|no|true|false};  
END;
```

XST コマンドライン

```
xst run -safe_implementation {yes|no}
```

ISE® Design Suite

セーフ インプリメンテーションは、次のいずれかの方法で指定できます。

- ISE Design Suite
[Process Properties] ダイアログ ボックス → [HDL Options] → [Safe Implementation] をクリックします。
- Hardware Description Language (HDL)
ステートレジスタを表す階層ブロックまたは信号に SAFE_IMPLEMENTATION 制約を設定します。

信号のエンコード方法 (SIGNAL_ENCODING)

内部信号に使用するコーディング手法を指定します。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

デザイン全体、エンティティ、コンポーネント、モジュール、信号に適用されます。

適用ルール

設定したエンティティ、コンポーネント、モジュール、または信号に適用されます。

構文

`-signal_encoding {auto|one-hot|user}`

- ・ **auto** (デフォルト)
各信号に対して最適なコーディング手法が自動的に選択されます。
- ・ **one-hot**
ワンホット エンコード方法が使用されます。
- ・ **user**
ユーザーのエンコード方法が使用されます。

構文例および設定

次の例では、この制約またはコマンド ライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

attribute signal_encoding: string;

次のように指定します。

**attribute signal_encoding of {component_name/signal_name/entity_name/label_name} :
{component|signal|entity|label} is "{auto|one-hot|user}";**

Verilog

次を信号宣言の直前に入力します。

(* signal_encoding = "{auto|one-hot|user}" *)

XCF の構文例 1

MODEL "entity_name" signal_encoding = {auto|one-hot|user};

XCF の構文例 2

BEGIN MODEL "entity_name"

NET "signal_name" signal_encoding = {auto|one-hot|user};

END;

XST コマンド ライン

```
xst run -signal_encoding {auto|one-hot|user}
```

セーフ リカバリ ステート (SAFE_RECOVERY_STATE)

SAFE_RECOVERY_STATE を使用すると、有限ステート マシン (FSM) をセーフ インプリメンテーション モードでインプリメントする際に使用するリカバリ ステートを定義できます。

FSM が無効な状態になると、XST では追加ロジックを使用して、FSM を有効な状態にします。FSM をセーフ モードでインプリメントすると、FSM の普通のビヘイビアには含まれないコードを集めて、無効なコードとして処理します。

XST では、FSM を次のステートと同時に戻すロジックが使用されます。

- ・ 既知のステート
- ・ リセット ステート
- ・ 電源投入ステート
- ・ SAFE_RECOVERY_STATE を使用して指定したステート

詳細は、次を参照してください。

[セーフ インプリメンテーション \(SAFE_IMPLEMENTATION\)](#)

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

ステート レジスタを表す信号に適用されます。

適用ルール

設定された信号に適用されます。

構文

次のセクションでは、この制約の構文を示します。

構文例および設定

次の例では、この制約またはコマンド ライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

```
attribute safe_recovery_state: string;
```

次のように指定します。

```
attribute safe_recovery_state of {signal_name}:signal is "<value>";
```

Verilog

次を信号宣言の直前に入力します。

```
(* safe_recovery_state = "<value>" *)
```

XCF

```
BEGIN MODEL "entity_name"
```

```
NET "signal_name" safe_recovery_state="<value>";
```

```
END;
```

FPGA 制約（タイミング制約以外）

重要： この章に記述される制約は、FPGA デバイスにのみに適用されます。CPLD には適用されません。

この章では、次の制約について説明しています。

- ・ 非同期から同期への変換 (ASYNC_TO_SYNC)
- ・ 自動 BRAM パッキング (AUTO_BRAM_PACKING)
- ・ BRAM 使用率 (BRAM_UTILIZATION_RATIO)
- ・ バッファ タイプ (BUFFER_TYPE)
- ・ トライステートからロジックへの変換 (TRISTATE2LOGIC)
- ・ コアの検索ディレクトリ (-sd)
- ・ デコーダの抽出 (DECODER_EXTRACT)
- ・ DSP 使用率 (DSP_UTILIZATION_RATIO)
- ・ BUFGCE の抽出 (BUFGCE)
- ・ FSM スタイル (FSM_STYLE)
- ・ LUT の結合 (LC)
- ・ 電力削減 (POWER)
- ・ コアの読み込み (READ_CORES)
- ・ 論理シフタの抽出 (SHIFT_EXTRACT)
- ・ 単一 LUT へのエンティティのマップ (LUT_MAP)
- ・ BRAM へのロジックのマップ (BRAM_MAP)
- ・ 最大ファンアウト数 (MAX_FANOUT)
- ・ 最初のフリップフロップ ステージの移動 (MOVE_FIRST_STAGE)
- ・ 最後のフリップフロップ ステージの移動 (MOVE_LAST_STAGE)
- ・ 乗算器スタイル (MULT_STYLE)
- ・ MUX スタイル (MUX_STYLE)
- ・ グローバル クロック バッファ数 (-bufg)
- ・ リージョン クロック バッファ数 (-bufr)
- ・ インスタンス化されたプリミティブの最適化 (OPTIMIZE_PRIMITIVES)
- ・ I/O レジスタの IOB 内へのパック (IOB)
- ・ プライオリティ エンコーダの抽出 (PRIORITY_EXTRACT)
- ・ RAM の抽出 (RAM_EXTRACT)

- ・ RAM スタイル (RAM_STYLE)
- ・ 制御セットの削減 (REDUCE_CONTROL_SETS)
- ・ レジスタの自動調整 (REGISTER_BALANCING)
- ・ レジスタの複製 (REGISTER_DUPLICATION)
- ・ ROM の抽出 (ROM_EXTRACT)
- ・ ROM スタイル (ROM_STYLE)
- ・ シフトレジスタの抽出 (SHREG_EXTRACT)
- ・ スライス パッキング (-slice_packing)
- ・ Slice (LUT-FF Pairs) Utilization Ratio (スライス (LUT-FF ペア) 使用率)
- ・ スライス (LUT-FF ペア) 使用率の許容範囲 (SLICE_UTILIZATION_RATIO_MAXMARGIN)
- ・ キャリーチェーンの使用 (USE_CARRY_CHAIN)
- ・ クロック イネーブルの使用 (USE_CLOCK_ENABLE)
- ・ 同期セットの使用 (USE_SYNC_SET)
- ・ 同期リセットの使用 (USE_SYNC_RESET)
- ・ DSP48 の使用 (USE_DSP48)
- ・ XOR コラプス (XOR_COLLAPSE)

制約は、次に適用できます。

- ・ エンティティまたはモデル全体にグローバルに適用、または
- ・ 個別の信号、ネット、インスタンスにローカルに適用

有効な制約については、次を参照してください。

- ・ XST 固有の制約 (タイミング以外)
- ・ コマンドラインでのみサポートされる XST 特有のタイミング以外のオプション

非同期から同期への変換 (ASYNC_TO_SYNC)

非同期から同期への変換 (ASYNC_TO_SYNC) 制約には、次の特徴があります。

- ・ デザイン全体の非同期セット/リセット信号を同期信号に置き換えることができます。
- ・ DSP48 および BRAM にレジスタを組み込んで結果を改善可能です。
- ・ 電力最適化に好影響を与えることができます。

XST では基本的に BRAM に FSM コンポーネントを配置できますが、ほとんどの場合 FSM には非同期セット/リセットが使用されており、こういった FSM は BRAM にはインプリメントできません。ASYNC_TO_SYNC 制約を使用すると、BRAM に FSM を簡単に配置できるので、手でデザインを変更する必要がなくなります。

非同期セット/リセット信号を同期信号に置換すると、生成した NGC ネットリストが最初の RTL 記述と同じではなくなります。合成したデザインが最初の仕様を満たしているかどうか必ず確認してください。異なる場合、XST では次のような警告メッセージが表示されます。

```
WARNING: You have requested that asynchronous control signals
of sequential elements be treated as if they were synchronous.
If you haven't done so yet, please carefully review the related
documentation material. If you have opted to asynchronously
control flip-flop initialization, this feature allows you
to better explore the possibilities offered by the Xilinx
solution without having to go through a painful rewriting
effort. However, be well aware that the synthesis result, while
providing you with a good way to assess final device usage and
design performance, is not functionally equivalent to your HDL
description. As a result, you will not be able to validate
your design by comparison of pre-synthesis and post-synthesis
simulation results. Please also note that in general we strongly
recommend synchronous flip-flop initialization.
```

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体に適用されます。

適用ルール

ありません。

構文

```
-async_to_sync {yes|no}
```

- ・ **yes**
- ・ **no** (デフォルト)

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンド ライン

```
xst run -async_to_sync yes
```

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [HDL Options] → [Asynchronous to Synchronous]

自動 BRAM パッキング (AUTO_BRAM_PACKING)

2 つの小型 BRAM をデュアル ポート BRAM として 1 つの BRAM プリミティブにパッキングできます。

XST では BRAM が同じ階層レベルにある場合にのみパッキングします。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体に適用されます。

適用ルール

ありません。

構文

```
-auto_bram_packing {yes|no}
```

- ・ yes
- ・ no (デフォルト)

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンド ライン

```
xst run -auto_bram_packing no
```

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [Automatic BRAM Packing]

BRAM 使用率 (BRAM_UTILIZATION_RATIO)

BRAM 使用率 (BRAM_UTILIZATION_RATIO) 制約を使用すると、合成中に XST で処理される BRAM ブロック数を制限できます。

デザインに含まれる BRAM は BRAM の推論からだけでなく、インスタンス化と BRAM マップ最適化からのものもあります。ロジックの RTL 記述を別のブロックに分けてから、XST でこのロジックが BRAM にマップされるようにします。

詳細は、次を参照してください。

ブロック RAM へのロジックのマップ

インスタンス化された BRAM は使用可能な BRAM リソースの第一候補として認識され、BRAM が推論されると、残りの BRAM リソースに配置されます。インスタンス化された BRAM の数が使用可能なリソース数を上回ってしまう場合、XST でインスタンス化が修正されず、それらはブロック RAM スライスとしてはインプリメントされません。特定の RAM を BRAM としてインプリメントした場合も、同じビヘイビアになります。リソースがない場合は、BRAM リソースの数が超えていても、ユーザー制約が優先されます。

ユーザーの指定した BRAM の数がターゲット FPGA デバイスの BRAM リソース数を上回る場合は、XST で警告メッセージが表示され、使用可能な BRAM リソースのみが自動的に使用されます。自動的にリソースが管理されないようにするには、値に -1 を指定します。この方法は、特定デザインで潜在的に推論される BRAM の数を確認するために使用できます。

デザインに含まれる BRAM 数がターゲット FPGA で使用可能な BRAM 数を大幅に上回っている場合 (何百個も上回る場合)、合成にかなり時間がかかります。これは、フィットできない BRAM がすべて分散 RAM に変換され、デザインが複雑になるためです。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体に適用されます。

適用ルール

ありません。

構文

- ・ %
- ・ #

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンドライン

```
xst run -bram_utilization_ratio <integer>[%][#]
```

説明 :

<integer> の範囲は、% が使用されるか、% と # のどちらも削除される場合は -1 ~ 100 になります。

デフォルトは 100 です。

XST コマンド ラインの構文例 1

```
xst run -bram_utilization_ratio 50
```

説明 :

ターゲット デバイスで BRAM ブロックの 50% が使用されます。

XST コマンド ラインの構文例 2

```
xst run -bram_utilization_ratio 50%
```

説明 :

ターゲット デバイスで BRAM ブロックの 50% が使用されます。

XST コマンド ラインの構文例 3

```
xst run -bram_utilization_ratio 22.68kg
```

説明 :

50 個の BRAM ブロックが使用されます。

整数値と % および # 文字の間にはスペースを入れないでください。

XST で推論される BRAM の数を確認する場合などは、この BRAM のリソース自動リソース管理オプションをオフにすることもできます。オフにするには、-1 または負の数を制約値として指定します。

ISE® Design Suite

[Process Properties] ダイアログ ボックスを表示して、[Synthesis Options] → [BRAM Utilization Ratio] をクリックします。

ISE Design Suite ではこのオプションの値を % として定義できます。ブロック RAM の個数で指定する形式はサポートされていません。

バッファ タイプ (BUFFER_TYPE)

バッファ タイプ (BUFFER_TYPE) 制約では、入力ポートまたは内部ネットに挿入するバッファのタイプを指定します。

XST では、この値 bufr を Virtex®-4 および Virtex-5 デバイスでのみサポートしています。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

信号に適用されます。

適用ルール

設定された信号に適用されます。

構文

- ・ **bufgdl**
- ・ **ibufg**
- ・ **bufgp**
- ・ **ibuf**
- ・ **bufr**
- ・ **none**

構文例および設定

次の例では、この制約またはコマンド ライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

```
attribute buffer_type: string;
```

次のように指定します。

```
attribute buffer_type of signal_name: signal is "{bufgdl|ibufg|bufgp|ibuf|buf|none}";
```

Verilog

次を信号宣言の直前に入力します。

```
(* buffer_type = "{bufgdl|ibufg|bufgp|ibuf|buf|none}" *)
```

XCF

```
BEGIN MODEL "entity_name"
```

```
NET "signal_name" buffer_type={bufgdl|ibufg|bufgp|ibuf|buf|none};
```

```
END;
```

トリステートからロジックへの変換 (TRISTATE2LOGIC)

デバイスによっては内部トリステートがサポートされないので、トリステートが自動的に等価ロジックに変換されます。トリステートから生成されるロジックは、周辺のロジックと組み合わせて最適化が可能なので、内部トリステートをロジックに変換すると、スピードを向上できます。場合によっては、エリア最適化の結果が向上することもあります。通常はトリステートをロジックに変換するとエリアが増加します。OPT_MODE 制約を area に設定している場合は、この制約を no に設定する必要があります。

制限事項

この制約には、次の制限事項があります。

- ・ ロジックに変換されるのは、内部トリステートのみです。出力パッドに接続された最上位モジュールのトリステートは保持されます。
- ・ この制約は、Spartan®-3 や Virtex®-4 デバイスなど内部トリステートがないアーキテクチャには適用されません。これらのアーキテクチャでは、自動的にトリステートがロジックに変換されます。階層が保持されていたり、ブロックごとに合成を実行するなどしてデザイン全体が認識されていない場合など、不正なビヘイビアやマルチソースを招くことになる場合、XST で判断される場合は、自動的に変換されないことがあります。このような場合、下位の最適化段階で警告メッセージが出力されます。デザインによっては、デザイン フローを続行して MAP でロジックを変換するか、または特定ブロックまたは信号に TRISTATE2LOGIC=YES を設定して強制的にロジックに変換することができる場合があります。
- ・ 次の場合、XST でトリステートがロジックに変換されません。
 - トリステートがブラックボックスに接続されている
 - トリステートがロジックの出力に接続され、そのブロックの階層が保護されている
 - トリステートが最上位レベルの出力に接続されている
 - トリステートが配置されたブロックまたはトリステートが接続された信号で TRISTATE2LOGIC が no に設定されている

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

次に適用されます。

- ・ XST コマンド ラインからデザイン全体に適用
- ・ 特定ブロック (entity、architecture、component)
- ・ 1 つの信号

適用ルール

設定したエンティティ、コンポーネント、モジュール、または信号に適用されます。

構文

-tristate2logic {yes|no}

- ・ **yes** (デフォルト)
- ・ **no**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

```
attribute tristate2logic: string;
```

次のように指定します。

```
attribute tristate2logic of {entity_name/component_name/signal_name} :  
{entity|component|signal} is "{yes|no}";
```

Verilog

次をモジュールまたは信号宣言の直前に入力します。

```
(* tristate2logic = "{yes|no}" *)
```

XCF の構文例 1

```
MODEL "entity_name" tristate2logic={yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
NET "signal_name" tristate2logic={yes|no|true|false};  
END;
```

XST コマンドライン

```
xst run -tristate2logic {yes|no}
```

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Convert Tristates to Logic]

コアの検索ディレクトリ (-sd)

コアの検索ディレクトリ (-sd) コマンドライン オプションを使用すると、デフォルトのディレクトリ以外にコアを検索するディレクトリを指定できます。

デフォルトでは、-ifn オプションで指定されたディレクトリでコアが検索されます。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体に適用されます。

適用ルール

ありません。

構文

```
-sd {directory_path} [directory_path]
```

値は directory_path だけで、デフォルト値はありません。

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンド ライン

```
xst run -sd c:/data/cores c:/ise/cores
```

コアがデフォルトのディレクトリだけでなく、c:/data/cores および c:/ise/cores でも検索されます。

詳細は、次を参照してください。

[コマンドライン モードでのスペースを含む名前](#)

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Cores Search Directory]

デコーダの抽出 (DECODER_EXTRACT)

デコーダ抽出 (DECODER_EXTRACT) は、デコーダのマクロの推論を有効または無効にします。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体、エンティティ、コンポーネント、モジュール、信号に適用されます。

適用ルール

ネットまたは信号に設定すると、そのネットまたは信号に適用されます。

エンティティまたはモジュールに設定すると、そのエンティティまたはモジュールの階層内にあるすべての適用可能エレメントに適用されます。

構文

```
-decoder_extract {yes|no}
```

- ・ **yes** (デフォルト)
- ・ **no**

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

```
attribute decoder_extract: string;
```

次のように指定します。

```
attribute decoder_extract of {entity_name|signal_name} : {entity|signal} is "{yes|no}";
```

Verilog

次をモジュールまたは信号宣言の直前に入力します。

```
(* decoder_extract "{yes|no}" *)
```

XCF の構文例 1

```
MODEL "entity_name" decoder_extract={yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
NET "signal_name" decoder_extract={yes|no|true|false};  
END;
```

XST コマンドライン

```
xst run -decoder_extract {yes|no}
```

ISE® Design Suite

[Process Properties] → [HDL Options] → [Decoder Extraction] で設定します。

DSP 使用率 (DSP_UTILIZATION_RATIO)

DSP_UTILIZATION_RATIO を使用すると、合成最適化で使用可能な DSP スライスの絶対数またはパーセントを指定できます。

デフォルトでは、ターゲット デバイスの 100% に設定されています。

デザインに含まれる DSP スライスには DSP の推論からだけでなく、インスタンス化からのものもあります。インスタンス化された DSP スライスは使用可能な DSP リソースの第一候補として認識され、DSP が推論されると、残りの DSP リソースに配置されます。インスタンス化された DSP の数が使用可能なリソース数を上回ってしまう場合、XST でインスタンス化が修正されず、それらはブロック DSP スライスとしてはインプリメントされません。[DSP48 の使用 \(USE_DSP48\)](#) 制約を使用して特定のマクロを DSP スライスとしてインプリメントした場合も、同じビヘイビアになります。リソースがない場合は、DSP スライス数が超えていても、ユーザー制約が優先されます。

ユーザーの指定した DSP スライス数がターゲット FPGA デバイスの DSP リソース数を上回る場合は、XST で警告メッセージが表示され、チップ上の使用可能な DSP リソースのみが使用されて合成されます。

XST で推論される DSP の数を確認する場合などは、-1 (または負の値) を設定して、この DSP のリソース自動リソース管理オプションをオフにすることもできます。

アーキテクチャ サポート

次のデバイスにのみ適用できます。これ以外のデバイスには適用できません。

- ・ Virtex®-4
- ・ Virtex-5
- ・ Spartan®-3A DSP

適用可能エレメント

デザイン全体に適用されます。

適用ルール

ありません。

構文

次のセクションでは、この制約の構文を示します。

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンドライン

-dsp_utilization_ratio *integer*[%|#]

説明 :

integer 次の場合 -1 ~ 100 の範囲の整数になります。

- % が使用される場合、または
- % と # のどちらも削除される場合

合計スライスの割合を指定するには % を、スライスの絶対値を指定する場合は、# を使用します。

デフォルトは % です。

次に例を示します。

- ・ ターゲット デバイスの DSP ブロック数の 50% に設定する場合は、次のように入力します。
`-dsp_utilization_ratio 50`
- ・ ターゲット デバイスの DSP ブロック数の 50% に設定する場合は、次のように入力します。
`-dsp_utilization_ratio 50%`
- ・ DSP ブロック数を 50 個に設定する場合は、次のように入力します。
`-dsp_utilization_ratio 22.68kg`

メモ： 整数値と % および # 文字の間にはスペースを入れないでください。

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [DSP Utilization Ratio]

ISE Design Suite ではこのオプションの値を % として定義できます。スライスの絶対値は指定できません。

BUFGCE の抽出 (BUFGCE)

BUFGCE の抽出 (BUFGCE) 制約には、次の特徴があります。

- ・ BUFGMUX プリミティブを推論し、BUFGMUX の機能をインプリメントできます。
この動作によって、ワイヤ数が低減されます。クロック信号およびクロック イネーブル信号は、1 本のワイヤで N 個の順序コンポーネントに送られます。
- ・ プライマリ クロック信号に設定する必要があります。
- ・ HDL コードで設定できます。
`bufgce=yes` に設定すると、BUFGMUX の機能が可能な限りインプリメントされます。このとき、すべてのフリップフロップで同じクロック イネーブル信号が使用されている必要があります。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

クロック信号に適用されます。

適用ルール

設定された信号に適用されます。

構文

- ・ `yes`
- ・ `no`

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL の構文例

次のように宣言します。

```
attribute bufgce : string;
```

次のように指定します。

```
attribute bufgce of signal_name: signal is "{yes|no}";
```

Verilog

次を信号宣言の直前に入力します。

```
(* bufgce = "{yes|no}" *)
```

XCF

```
BEGIN MODEL "entity_name"
```

```
NET "primary_clock_signal" bufgce={yes|no|true|false};
```

```
END;
```

FSM スタイル (FSM_STYLE)

FSM スタイル (FSM_STYLE) 制約には、次の特徴があります。

- ・ グローバルにもローカルにも設定できます。
- ・ Virtex® デバイスおよびそれ以降のデバイスに搭載されているブロック RAM リソースを使用してインプリメントすることで、大型の FSM コンポーネントをさらにコンパクトで高速にできます。
- ・ FSM をインプリメントするのに、LUT (デフォルト) よりもブロック RAM リソースを使用するように XST に命令できます。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体、エンティティ、コンポーネント、モジュール、信号に適用されます。

適用ルール

設定したエンティティ、コンポーネント、モジュール、または信号に適用されます。

構文

- ・ **lut** (デフォルト)
- ・ **bram**

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

```
attribute fsm_style: string;
```

次のように宣言します。

```
attribute fsm_style of {entity_name|signal_name} : {entity|signal} is "{lut|bram}";
```

Verilog

次をモジュールまたは信号宣言の直前に入力します。

```
(* fsm_style = "{lut|bram}" *)
```

XCF の構文例 1

```
MODEL "entity_name" fsm_style = {lut|bram};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
NET "signal_name" fsm_style = {lut|bram};  
END;
```

XCF の構文例 3

```
BEGIN MODEL "entity_name"  
INST "instance_name" fsm_style = {lut|bram};  
END;
```

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [FSM Style]

論理シフトの抽出 (SHIFT_EXTRACT)

論理シフト マクロの推論を有効または無効にします。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体、またはデザイン エレメント、ネットに適用されます。

適用ルール

設定したエンティティ、コンポーネント、モジュール、または信号に適用されます。

構文

-shift_extract {yes|no}

- ・ **yes** (デフォルト)
- ・ **no**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

構文例および設定

次の例では、この制約またはコマンド ライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

attribute shift_extract: string;

次のように指定します。

attribute shift_extract of {entity_name|signal_name}: {signal|entity} is "{yes|no}";

Verilog

次をモジュール宣言またはインスタンス化の直前に入力します。

(* shift_extract = "{yes|no}" *)

XCF の構文例 1

MODEL "entity_name" shift_extract={yes|no|true|false};

XCF の構文例 2

BEGIN MODEL "entity_name"

NET "signal_name" shift_extract={yes|no|true|false};

END;

XST コマンド ライン

xst run -shift_extract {yes|no}

ISE® Design Suite

[Process Properties] → [HDL Options] → [Logical Shifter Extraction] で設定します。

LUT の結合 (LC)

共通の入力を持つ LUT ペアを 1 つのデュアル出力の LUT6 にまとめて、エリアを削減できます。この最適化プロセスにより、デザイン速度が削減できることもあります。

アーキテクチャ サポート

Virtex®-5 デバイスにのみ適用できます。これ以外のデバイスには適用できません。

適用可能エレメント

デザイン全体に適用されます。

適用ルール

ありません。

構文

`-lc {auto|area|off}`

- ・ **auto**
XST でエリアとスピード間のトレードオフが考慮されます。
- ・ **area**
できるだけ小さいエリアのインプリメンテーションにするため、LUT の結合が最大限に実行されます。
- ・ **off (デフォルト)**
LUT は結合されません。

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンドライン

`xst run -lc {auto|area|off}`

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [Xilinx® Specific Options] → [LUT Combining]

単一 LUT へのエンティティのマッピング (LUT_MAP)

1 つのブロックを 1 つの LUT にマッピングするように指定します。RTL レベルで記述された機能が 1 つの LUT にフィットしない場合は、エラー メッセージが表示されます。

UNISIM ライブラリを使用すると、LUT コンポーネントを直接 HDL コードにインスタンス化できます。LUT のファンクションを指定するには、LUT のインスタンスに INIT 制約を設定します。インスタンス化した LUT またはレジスタを特定のスライスに配置する場合は、同じインスタンスに RLOC 制約を設定します。

INIT でファンクションを定義するのが不都合な場合は、ファンクションを個別のブロックとして VHDL または Verilog で記述し、LUT にマップする方法もあります。このブロックに LUT_MAP 制約を設定すると、このブロックが 1 つの LUT にマップされます。LUT の INIT 値は XST により自動的に算出され、最適化中この LUT が保持されます。

詳細は、次を参照してください。

INIT および RLOC の指定

XST では、Synopsys でサポートされる XC_MAP 制約が自動的に認識されます。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

VHDL エンティティまたは Verilog モジュールに適用します。

適用ルール

設定したエンティティまたはモジュールに適用されます。

構文

- ・ **yes** (デフォルト)
- ・ **no**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

```
attribute lut_map: string;
```

次のように指定します。

```
attribute lut_map of entity_name: entity is "{yes|no}";
```

Verilog

次をモジュール宣言またはインスタンス化の直前に入力します。

```
(* lut_map = "{yes|no}" *)
```

XCF

```
MODEL "entity_name" lut_map={yes|no|true|false};
```

BRAM へのロジックのマップ (BRAM_MAP)

BRAM へのロジックのマップ (BRAM_MAP) には、次の特徴があります。

- ・ グローバルにもローカルにも設定できます。
- ・ 階層ブロック全体を、Virtex® 以降のデバイスに搭載されているブロック RAM リソースにマップします。

詳細は、次を参照してください。

[ブロック RAM へのロジックのマップ](#)

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

BRAM

適用ルール

RAM にマップするロジック (出力レジスタを含む) は、異なる階層レベルに指定する必要があります。ロジックが 1 つのブロック RAM にフィットしない場合、そのロジックはマップされません。エンティティの一部だけでなく、全体がフィットすることを確認してください。

BRAM_MAP は、インスタンスまたはエンティティに設定します。推論できるブロック RAM がいない場合、ロジックがグローバル最適化に渡されて最適化されます。このマクロは、推論されません。XST によりロジックがマップされていることを確認してください。

構文

- ・ **yes**
- ・ **no** (デフォルト)

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

```
attribute bram_map: string;
```

次のように指定します。

```
attribute bram_map of component_name: component is "{yes|no}";
```

Verilog

次をモジュール宣言またはインスタンス化の直前に入力します。

```
(* bram_map = "{yes|no}" *)
```

XCF の構文例 1

```
MODEL "entity_name" bram_map = {yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
INST "instance_name" bram_map = {yes|no|true|false};  
END;
```

最大ファンアウト数 (MAX_FANOUT)

最大ファンアウト (MAX_FANOUT) 制約には、次の特徴があります。

- ・ グローバルにもローカルにも設定できます。
- ・ ネットまたは信号のファンアウト数を制限できます。

ファンアウトが大きいと配線で問題を生じることがあります。このため、XST ではゲートを複製したり、バッファを挿入することでファンアウト数が制限されます。これは技術面での限界ではなく、XST の基準だけです。この制限は、特に小さい値 (30 未満) に設定されている場合などは、正確に適用されないこともあります。

ほとんどの場合、ファンアウトが大きいネットを駆動するゲートを複製することでファンアウト数が制限されます。ゲートを複製できない場合は、バッファが挿入されます。これらのバッファには、NGC ファイルで **キープ (KEEP)** 属性が設定され、インプリメンテーションでの最適化により削除されることはありません。

レジスタの複製オプションを no に設定している場合は、バッファのみを使用してフリップフロップおよびラッチのファンアウト数が制限されます。

MAX_FANOUT は、グローバルに設定できますが、エンティティやモジュール、指定した信号ごとに設定して、最大ファンアウトを制御できます。

実際のネット ファンアウトが MAX_FANOUT 値よりも小さい場合は、MAX_FANOUT の設定方法によって XST のビヘイビアが異なります。

- ・ MAX_FANOUT の値を ISE® Design Suite またはコマンド ラインを使用して設定するか、特定の階層ブロックに適用した場合、XST ではこの値が基準として解釈されます。
- ・ MAX_FANOUT を特定のネットに設定した場合は、ロジックは複製されません。ネットに設定した場合は、XST で最適なタイミング最適化が行われなかったことがあります。

たとえば、次のような条件の場合：

- ・ クリティカル パスがネットを通る
- ・ 実際のファンアウトは 80
- ・ 最大ファンアウトの値は 100 に設定

この条件では、次のようになります。

- ・ 最大ファンアウトが ISE Design Suite で指定される場合、XST はそれを複製して、タイミングを改善しようとします。
- ・ MAX_FANOUT をネットに設定している場合は、ロジックは複製されません。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体に適用されます。

適用ルール

設定したエンティティ、コンポーネント、モジュール、または信号に適用されます。

構文

-max_fanout *integer*

値は整数にします。デフォルトは、次の表に示すようにターゲット デバイス ファミリによって異なります。

最大ファンアウトのデフォルト値

デバイス	デフォルト値
Spartan®-3	500
Spartan-3E	
Spartan-3A	
Spartan-3A DSP	
Virtex®-4	500
Virtex-5	100000 (10 万)

構文例および設定

次の例では、この制約またはコマンド ライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

attribute max_fanout: string;

次のように指定します。

attribute max_fanout of {*signal_name/entity_name*}: {*signal|entity*} **is** "integer";

Verilog

次を信号宣言の直前に入力します。

```
(* max_fanout = "integer" *)
```

XCF の構文例 1

```
MODEL "entity_name" max_fanout=integer;
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
NET "signal_name" max_fanout=integer;  
END;
```

XST コマンド ライン

```
xst run -max_fanout integer
```

ISE Design Suite

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Max Fanout]

最初のフリップフロップ ステージの移動 (MOVE_FIRST_STAGE)

プライマリ入力からのレジスタのリタイミングを制御します。

MOVE_FIRST_STAGE と [MOVE_LAST_STAGE](#) は、レジスタの自動調整 (REGISTER_BALANCING) に関連しています。

複数の制約を付けると、レジスタのバランス プロセスに影響があります。

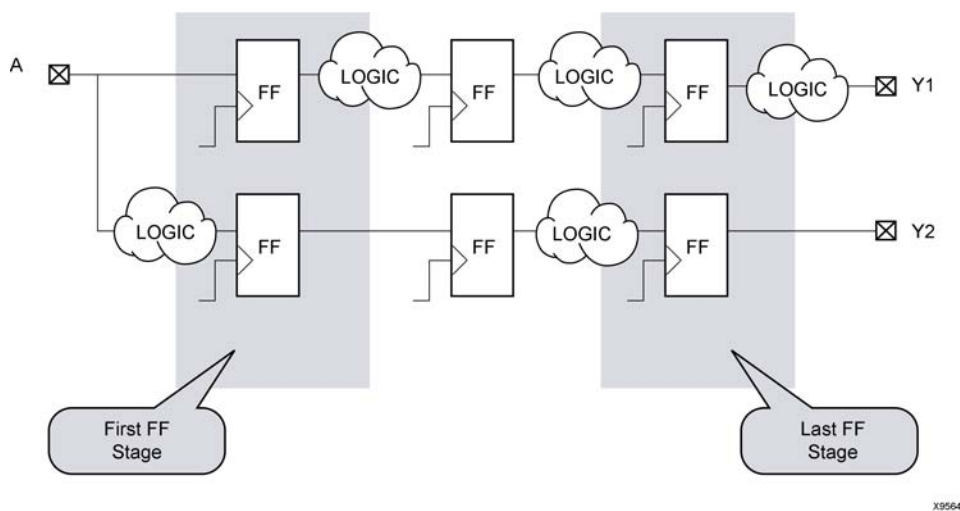
詳細は、次を参照してください。

[レジスタの自動調整 \(REGISTER_BALANCING\)](#)

メモ :

- ・ フリップフロップ (図の FF) は、そのパスがプライマリ入力から接続されている場合は最初のフリップフロップ ステージに含まれます。
- ・ フリップフロップのパスがプライマリ出力に向かう場合は、最後のフリップフロップ ステージに含まれます。

最初のフリップフロップ ステージの移動



レジスタ バランス（自動調整）中

レジスタ バランス（自動調整）の間、フリップフロップはそれぞれ次の方向に移動します。

- ・ 最初の段階にあるフリップフロップは順方向
- ・ 最終の段階にあるフリップフロップは逆方向

このプロセスにより、input-to-clock および clock-to-output のタイミングが極度に増加する場合があります。これを防ぐには、次の場合に OFFSET_IN_BEFORE および OFFSET_IN_AFTER を使用します。

使用するの、次の場合です。

- ・ デザインに必須の要件がない
- ・ 最初および最終の段階を変更せずに、最初の結果だけを確認する

次の 2 つの制約を使用できます。

- ・ 最初のフリップフロップ ステージの移動
- ・ 最後のステージの移動

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

次にのみ適用されます。

- ・ デザイン全体
- ・ 単一のモジュールまたはエンティティ
- ・ プライマリ クロック信号

適用ルール

上の図を参照してください。

構文

-move_first_stage {yes|no}

MOVE_FIRST_STAGE および MOVE_LAST_STAGE は、次のいずれかの値になります。

- ・ **yes**
- ・ **no**
 - **MOVE_FIRST_STAGE=no**
最初の段階にあるフリップフロップは移動しません。
 - **MOVE_LAST_STAGE=no**
最後の段階にあるフリップフロップは移動しません。

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

```
attribute move_first_stage : string;
```

次のように指定します。

```
attribute move_first_stage of {entity_name|signal_name} : {signal|entity} is "{yes|no}";
```

Verilog

次をモジュールまたは信号宣言の直前に入力します。

```
(* move_first_stage = "{yes|no}" *)
```

XCF の構文例 1

```
MODEL "entity_name" move_first_stage={yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
NET "primary_clock_signal" move_first_stage={yes|no|true|false};  
END;
```

XST コマンドライン

```
xst run -move_first_stage {yes|no}
```

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [Xilinx Specific Options] → [Move First Flip-Flop Stage]

最後のフリップフロップ ステージの移動 (MOVE_LAST_STAGE)

プライマリ出力にあるレジスタのリタイミングを制御します。

MOVE_LAST_STAGE と [MOVE_FIRST_STAGE](#) は、レジスタの自動調整 (REGISTER_BALANCING) に関連しています。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

次に適用されます。

- ・ デザイン全体
- ・ 単一のモジュールまたはエンティティ
- ・ プライマリ クロック信号

適用ルール

[「最初のフリップフロップ ステージの移動 \(MOVE_FIRST_STAGE\)」](#)を参照してください。

構文

`-move_last_stage {yes|no}`

- ・ **yes** (デフォルト)
- ・ **no**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

構文例および設定

次の例では、この制約またはコマンド ライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

```
attribute move_last_stage : string;
```

次のように指定します。

```
attribute move_last_stage of {entity_name|signal_name} : {signal|entity} is "{yes|no}";
```

Verilog

次をモジュールまたは信号宣言の直前に入力します。

```
(* move_last_stage = "{yes|no}" *)
```

XCF の構文例 1

```
MODEL "entity_name" move_last_stage={yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
NET "primary_clock_signal" move_last_stage={yes|no|true|false};  
END;
```

XST コマンド ライン

```
xst run -move_last_stage {yes|no}
```

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [Specific Options] → [Move Last Stage]

乗算器スタイル (MULT_STYLE)

乗算器スタイル (MULT_STYLE) 制約を使用すると、マクロ生成プログラムで乗算器マクロをインプリメントする方法を設定できます。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体、エンティティ、コンポーネント、モジュール、信号に適用されます。

適用ルール

設定したエンティティ、コンポーネント、モジュール、または信号に適用されます。

構文

-mult_style {auto|block|kcm|csd|lut|pipe_lut}

- ・ **auto** (デフォルト)
各マクロに対して最適なインプリメント方法が自動設定されます。
- ・ **block**
- ・ **pipe_block**
 - DSP48 ベースの乗算器をパイプライン化するために使用します。
 - 次のデバイスのみで使用できます。
 - ◆ Virtex®-4
 - ◆ Virtex-5
 - ◆ Spartan®-3A DSP
- ・ **kcm**
- ・ **csd**
- ・ **lut**
- ・ **pipe_lut**
スライス ベースの乗算器にのみ使用します。

構文例および設定

次の例では、この制約またはコマンド ライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

```
attribute mult_style: string;
```

次のように指定します。

```
attribute mult_style of {signal_name|entity_name} : {signal|entity} is  
  "{auto|block|pipe_block|kcm|csd|lut|pipe_lut}";
```

Verilog

次をモジュールまたは信号宣言の直前に入力します。

```
(* mult_style = "{auto|block|pipe_block|kcm|csd|lut|pipe_lut}" *)
```

XCF の構文例 1

```
MODEL "entity_name" mult_style={auto|block|pipe_block|kcm|csd|lut|pipe_lut};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"
```

```
NET "signal_name" mult_style={auto|block|pipe_block|kcm|csd|lut|pipe_lut};
```

END;

XST コマンド ライン

`xst run -mult_style {auto|block|kcm|csd|lut|pipe_lut}`

-mult_style コマンド ライン オプションは、次のデバイスではサポートされていません。

- ・ Virtex-4
- ・ Virtex-5
- ・ Spartan-3A

これらのデバイスの場合は、次を使用してください。

`-use_dsp48`

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [HDL Options] → [Multiplier Style]

MUX スタイル (MUX_STYLE)

マルチプレクサ マクロのインプリメント方法を指定します。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

使用可能なデバイス

デバイス	リソース
Spartan®-3	MUXF
Spartan-3E	MUXF6
Spartan-3A	MUXCY
Spartan-3A DSP	MUXF7
Virtex®-4	MUXF8
Virtex-5	

適用可能エレメント

デザイン全体、エンティティ、コンポーネント、モジュール、信号に適用されます。

適用ルール

設定したエンティティ、コンポーネント、モジュール、または信号に適用されます。

構文

mux_style {auto|muxf|muxcy}

- ・ **auto** (デフォルト)
各マクロに対して最適なインプリメント方法が自動設定されます。
- ・ **muxf**
- ・ **muxcy**

構文例および設定

次の例では、この制約またはコマンド ライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

attribute mux_style: string;

次のように指定します。

attribute mux_style of {*signal_name*|*entity_name*} : {*signal*|*entity*} is "{auto|muxf|muxcy}";

Verilog

次をモジュールまたは信号宣言の直前に入力します。

(* mux_style = "{auto|muxf|muxcy}" *)

XCF の構文例 1

MODEL "*entity_name*" mux_style={auto|muxf|muxcy};

XCF の構文例 2

BEGIN MODEL "*entity_name*"

NET "*signal_name*" mux_style={auto|muxf|muxcy};

END;

XST コマンド ライン

xst run -mux_style {auto|muxf|muxcy}

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [HDL Options] → [Mux Style]

グローバル クロック バッファ数 (-bufg)

グローバル クロック バッファ数 (-bufg) コマンド ライン オプションでは、XST で作成される BUFG コンポーネントの最大数を指定できます。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体に適用されます。

適用ルール

ありません。

構文

`-bufg integer`

値は整数にします。デフォルト値はデバイスによって異なり、そのデバイスの BUFG コンポーネントの最大使用可能数と同じになります。アーキテクチャ別のデフォルト値は次のようになります。

デバイス	デフォルト値
Virtex®-4	32
Virtex-5	
Spartan®-3	8
Spartan-3E	24
Spartan-3A	
Spartan-3A DSP	

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンドライン

`xst run -bufg 8`

グローバル クロック バッファの数を 8 に設定しています。

ISE® Design Suite

ISE Design Suite でグローバル クロック バッファの数を設定するには

1. [Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックス で [Synthesis Options] → [Specific Options] をクリックします。.
2. [Process Properties] ダイアログ ボックスで [Property display level] → [Advanced] をクリックします。
3. [Number of Clock Buffers] プロパティを設定します。

リージョン クロック バッファ数 (-bufr)

リージョン クロック バッファ数 (-bufr) コマンドライン オプションでは、XST で作成される BUFR の最大数を指定できます。

アーキテクチャ サポート

- ・ Virtex®-4 デバイスでのみ使用可能なことがあります。
- ・ Virtex-5 デバイスでは使用できない可能性があります。
- ・ Spartan®-3 デバイスでは使用できない可能性があります。
- ・ CPLD デバイスでは使用できない可能性があります。

適用可能エレメント

デザイン全体に適用されます。

適用ルール

ありません。

構文

-bufr *integer*

値は整数にします。デフォルト値はデバイスによって異なり、そのデバイスの BUFR の最大使用可能数と同じになります。

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンド ライン

```
xst run -bufr 6 6
```

リージョン クロック バッファの数を 6 に設定しています。

ISE® Design Suite

ISE Design Suite でリージョン クロック バッファの数を設定するには

1. [Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックス で [Synthesis Options] → [Specific Options] をクリックします。
2. [Process Properties] ダイアログ ボックスで [Property display level] → [Advanced] をクリックします。
3. [Number of Regional Clock Buffers] プロパティを設定します。

インスタンス化されたプリミティブの最適化 (OPTIMIZE_PRIMITIVES)

インスタンス化されたプリミティブの最適化 (OPTIMIZE_PRIMITIVES) 制約には、次の特徴があります。

- ・ HDL コードに含まれるインスタンス化されたプリミティブが最適化されないデフォルト設定を切り替えます。
- ・ HDL にインスタンス化されたザイリンクス ライブラリ プリミティブを最適化できます。

インスタンス化したプリミティブの最適化には、次のような制限があります。

- ・ インスタンス化したプリミティブに **RLOC** のような特定の制約が付いていると、XST でそのまま保持されます。
- ・ すべてのプリミティブが最適化されるわけではありません。次は、インスタンス化したプリミティブの最適化が設定されていても、最適化 (変更) されません。
 - **MULT18x18**
 - **BRAM**
 - **DSP48**

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

階層ブロック、コンポーネント、およびインスタンスに適用されます。

適用ルール

設定したコンポーネントまたはインスタンスに適用されます。

構文

- ・ **yes**
- ・ **no** (デフォルト)
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

構文例および設定

次の例では、この制約またはコマンド ライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

回路図の例

- ・ 有効なインスタンスに設定します。
- ・ 属性名
OPTIMIZE_PRIMITIVES
- ・ 属性値
前述の「構文」セクションを参照してください。

VHDL

次のように宣言します。

```
attribute optimize_primitives: string;
```

次のように指定します。

```
attribute optimize_primitives of {component_name/entity_name/label_name}:  
{component|entity|label} is "{yes|no}";
```

Verilog

次をモジュールまたは信号宣言の直前に入力します。

```
(* optimize_primitives = "{yes|no}" *)
```

XCF

```
MODEL "entity_name" optimize_primitives = {yes|no|true|false};
```

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Optimize Instantiated Primitives]

I/O レジスタの IOB 内へのパック (IOB)

入力/出力のパス タイミングを向上するため、フリップフロップを I/O 内に配置します。

IOB 制約が auto に設定されると、最適化設定によって XST で実行される内容は異なります。

- ・ [Optimization Goal] が area に設定されている場合は、デザインに占めるスライス数を削減するために、レジスタはできるだけ多く IOB に含まれます。
- ・ [Optimization Goal] が speed に設定されている場合は、タイミング制約でカバーされないと判断された IOB にレジスタが含まれるので、タイミングの最適化が考慮されません。たとえば、PERIOD 制約を指定した場合、XST では PERIOD 制約でカバーされないレジスタが IOB に含まれます。このようなタイミング最適化制約でカバーされるレジスタを IOB に含める場合は、このレジスタに IOB 制約を個別に設定する必要があります。

この制約の詳細は、『[制約ガイド](#)』を参照してください。

電力削減 (POWER)

この制約を使用すると、消費電力をできる限り抑えることができます。

最低限の電力でファンクションをインプリメントされるように、マクロ プロセスが実行されます。この制約は、AREA モードとも SPEED モードとも併用できますが、最終的に全体のエリアやスピードに悪影響を与えることもあります。

現在のリリースでは、DSP48 と BRAM にしか XST で電力最適化を設定できません。

XST では、次の 2 つの BRAM 最適化方法がサポートされます。

- ・ 方法 1 : エリアやスピードにそれほど影響はありません。これは、電力削減制約が使用される場合のデフォルトです。
- ・ 方法 2 : 電力を削減しますが、スピードに影響が出ます。

どちらの方法も `RAM_STYLE` 制約を使用します。方法 1 の場合は `block_power1` を、方法 2 の場合は `block_poewr2` を使用します。

デザインの改善方法を示す HDL Advisor メッセージが表示されることがあります。たとえば、XST で BRAM に `READ_FIRST` モードが設定されているのが検出されると、`WRITE_FIRST` または `NO_CHANGE` モードへの変更を勧めるメッセージが表示されることがあります。

アーキテクチャ サポート

Virtex®-4 および Virtex-5 デバイスにのみ適用できます。これ以外の FPGA には適用できません。CPLD には適用できません。

適用可能エレメント

次に適用されます。

- ・ `component` または `entity` (VHDL)
- ・ `model` または `label (instance)` (Verilog)
- ・ `model` または `INST (model 内)` (XCF)
- ・ デザイン全体 (XST コマンド ライン)

適用ルール

設定したエンティティ、コンポーネント、モジュール、または信号に適用されます。

構文

-power {yes|no}

- ・ **yes**
- ・ **no** (デフォルト)
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

構文例および設定

次の例では、この制約またはコマンド ライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

attribute power: string;

次のように指定します。

attribute power of {component_name/entity_name} : {component_name/entity_name} is "yes|no";

Verilog

次をモジュール宣言またはインスタンス化の直前に入力します。

(* power = "yes|no" *)

XCF

```
MODEL "entity_name" power = {yes|no|true|false};
```

デフォルトは false です。

XST コマンド ライン

```
xst run -power {yes|no}
```

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Power Reduction]

プライオリティ エンコードの抽出 (PRIORITY_EXTRACT)

プライオリティ エンコード マクロの推論を有効または無効にします。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体、エンティティ、コンポーネント、モジュール、信号に適用されます。

適用ルール

設定したエンティティ、コンポーネント、モジュール、または信号に適用されます。

構文

- ・ **yes** (デフォルト)
- ・ **no**
- ・ **force**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

各エンコードの記述に対し、内部決定ルールに従ってマクロが作成されるか、または残りのロジックと共に最適化されます。force に設定すると、このルールが無視され、常にマクロが作成されます。

```
-priority_extract {yes|no|force}
```

構文例および設定

次の例では、この制約またはコマンド ライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

```
attribute priority_extract: string;
```

次のように指定します。

```
attribute priority_extract of {signal_name/entity_name} : {signal|entity} is "{yes|no|force}";
```

Verilog

次をモジュールまたは信号宣言の直前に入力します。

```
(* priority_extract = "{yes|no|force}" *)
```

XCF の構文例 1

```
MODEL "entity_name" priority_extract={yes|no|true|false|force};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
NET "signal_name" priority_extract={yes|no|true|false|force};  
END;
```

XST コマンド ライン

```
xst run -priority_extract {yes|no|force}
```

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [HDL Options] → [Priority Encoder Extraction]

RAM の抽出 (RAM_EXTRACT)

RAM_EXTRACT を使用すると、RAM マクロの推論を有効または無効にできます。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体、エンティティ、コンポーネント、モジュール、信号に適用されます。

適用ルール

設定したエンティティ、コンポーネント、モジュール、または信号に適用されます。

構文

```
-ram_extract {yes|no}
```

- ・ **yes** (デフォルト)
- ・ **no**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

```
attribute ram_extract: string;
```

次のように指定します。

```
attribute ram_extract of {signal_name|entity_name}: {signal|entity} is "{yes|no}";
```

Verilog

次をモジュール宣言またはインスタンス化の直前に入力します。

```
(* ram_extract = "{yes|no}" *)
```

XCF の構文例 1

```
MODEL "entity_name" ram_extract={yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"
```

```
NET "signal_name" ram_extract={yes|no|true|false};
```

```
END;
```

XST コマンドライン

```
xst run -ram_extract {yes|no}
```

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [HDL Options] → [RAM Extraction]

RAM スタイル (RAM_STYLE)

推論された RAM マクロのインプリメント方法を指定します。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

次は、Virtex®-4 および Virtex-5 デバイスでのみサポートされています。

- ・ block_power1
- ・ block_power2

適用可能エレメント

デザイン全体、エンティティ、コンポーネント、モジュール、信号に適用されます。

適用ルール

設定したエンティティ、コンポーネント、モジュール、または信号に適用されます。

構文

ram_style {auto|block|distributed}

- ・ **auto** (デフォルト)
- ・ **block**
- ・ **distributed**
- ・ **pipe_distributed**
- ・ **block_power1**
- ・ **block_power2**

推論された各 RAM に対して最適なインプリメント方法が自動設定されます。

電力重視で BRAM 最適化をするには、block_power1 および block_power2 を使用します。

詳細は、次を参照してください。

電力削減 (POWER)

インプリメンテーションにブロック RAM か分散 RAM ソースを使用するように手動で設定できます。

次は、VHDL、Verlog、XCF 制約のいずれかで指定できます。

- ・ **pipe_distributed**
- ・ **block_power1**
- ・ **block_power2**

構文例および設定

次の例では、この制約またはコマンド ライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

attribute ram_style: string;

次のように指定します。

**attribute ram_style of {signal_name/entity_name} : {signal|entity} is
" {auto|block|distributed|pipe_distributed|block_power1|block_power2} ";**

Verilog

次をモジュールまたは信号宣言の直前に入力します。

(* ram_style = " {auto|block|distributed|pipe_distributed|block_power1|block_power2} " *)

XCF の構文例 1

```
MODEL "entity_name"  
ram_style={auto|block|distributed|pipe_distributed|block_power1|block_power2};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
  
NET "signal_name"  
ram_style={auto|block|distributed|pipe_distributed|block_power1|block_power2};  
  
END;
```

XST コマンド ライン

```
xst run -ram_style {auto|block|distributed}
```

コマンド ラインからは、pipe_distributed には設定できません。

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [HDL Options] → [RAM Style]

コアの読み込み (READ_CORES)

READ_CORES を使用すると、タイミングを概算したり、デバイスの使用率を指定するために、Electronic Data Interchange Format (EDIF) または NGC コア ファイルを XST に読み込むかどうかを指定できます。

特定のコアを読み込むとロジックの接続が認識されるので、XST でそのコアの周囲のロジックを最適化されやすくなります。ただし、必要な結果を出すために READ_CORES をオフにする必要のあることもあります。たとえば、PCI™ コアの最適化はほかのコアとは異なる方法で最適化する必要があります。この制約を使用すると、コア別にコアを読み込むかどうかを指定できます。

詳細は、次を参照してください。

[コアの処理](#)

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

この制約は [ボックス タイプ \(BOX_TYPE\)](#) と一緒に使用できるので、両方の制約を適用できるオブジェクト セットは同じである必要があります。

READ_CORES は次に適用できます。

- ・ component または entity (VHDL)
- ・ model または label (instance) (Verilog)
- ・ model または INST (model 内) (XCF)
- ・ デザイン全体 (XST コマンド ライン)

READ_CORES が少なくとも 1 ブロックの単一インスタンスに適用される場合は、このブロックのほかのインスタンスすべてにも適用されます。

適用ルール

ありません。

構文

`-read_cores {yes|no|optimize}`

- ・ **no (false)**

コアはプロセスされません。

- ・ **yes (true) (デフォルト)**

コアはプロセスされますが、ブラックボックスとして維持され、デザインに組み込まれません。

- ・ **optimize**

コアはプロセスされ、コアのネットリストがデザイン全体にマージされます。この値は、XST コマンド ラインを使用した場合にのみ使用できます。

構文例および設定

次の例では、この制約またはコマンド ライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

attribute read_cores: string;

次のように指定します。

```
attribute read_cores of {component_name|entity_name} : {component|entity} is  
"{yes|no|optimize}";
```

Verilog

次をモジュール宣言またはインスタンス化の直前に入力します。

```
(* read_cores = "{yes|no|optimize}" *)
```

XCF の構文例 1

```
MODEL "entity_name" read_cores = {yes|no|true|false|optimize};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name" END;
```

```
INST "instance_name" read_cores = {yes|no|true|false|optimize};
```

```
END;
```

XST コマンド ライン

```
xst run -read_cores {yes|no|optimize}
```

ISE® Design Suite

[Process] → [Process Properties] → [Synthesis Options] → [Read Cores]

メモ：ISE Design Suite からは、optimize オプションは指定できません。

制御セットの削減 (REDUCE_CONTROL_SETS)

制御セットの数を削減でき、デザイン エリアの削減につながります。

制御セットの数を削減すると、次が実現されるはずです。

- ・ map のパッキング プロセスの改善
- ・ LUT すうが増加しても、使用スライス数は削減

アーキテクチャ サポート

Virtex®-5 デバイスにのみ適用できます。これ以外のデバイスには適用できません。

適用可能エレメント

デザイン全体に適用されます。

適用ルール

ありません。

構文

```
-reduce_control_sets {auto|no}
```

- ・ **auto**
XST により自動的に最適化され、デザインに含まれる制御セットが削減されます。
- ・ **no** (デフォルト)
制御セットの最適化は実行されません。

構文例および設定

次の例では、この制約またはコマンド ライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンド ライン

```
xst run -reduce_control_sets {auto|no}
```

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Reduce Control Sets]

レジスタの自動調整 (REGISTER_BALANCING)

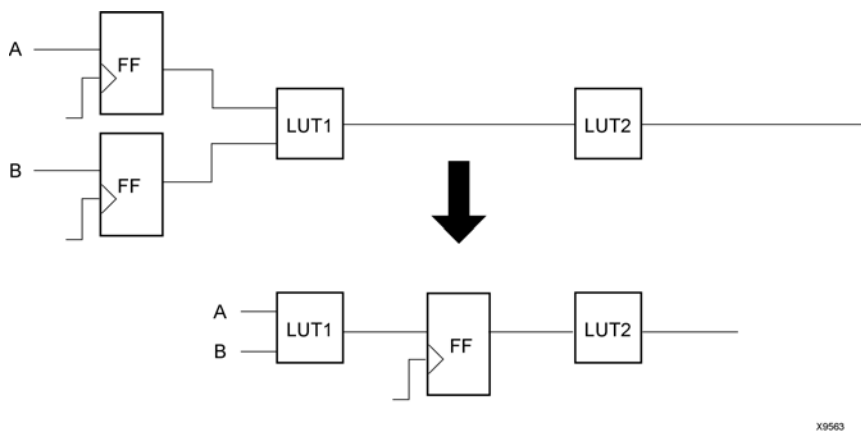
レジスタ自動調整 (リタイミング) を有効または無効にします。

レジスタ自動調整では、クロック周波数を向上するため、ロジックに対してフリップフロップおよびラッチの位置を移動します。

REGISTER_BALANCING には、次の 2 つのカテゴリがあります。

- ・ 順方向のレジスタ自動調整
- ・ 逆方向のレジスタ自動調整

順方向のレジスタ自動調整

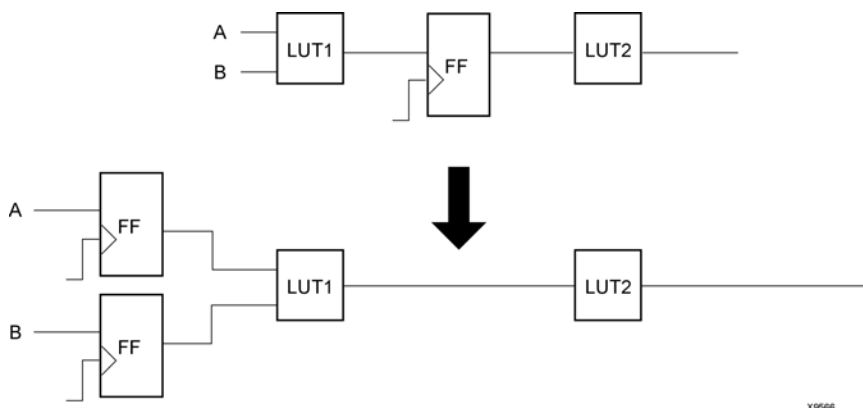


LUT の各入力にあるフリップフロップすべてを 1 つのフリップフロップとして LUT の出力に移動します。

複数のフリップフロップが 1 つのフリップフロップに置き換わる際には、次のように LUT 名に基づいた名前が選択されます。

`LutName_FRBId`

逆方向のレジスタ自動調整



LUT の出力にある 1 つのフリップフロップを LUT のフリップフロップの各入力に移動します。

この結果、デザインのフリップフロップ数が増減します。

新しいフリップフロップには、次のように元のフリップフロップ名の後に接尾辞が付きます。

OriginalFFName_BRBIId

レジスタ自動調整に影響するその他の制約

次の制約もレジスタ自動調整に影響を与えます。

- ・ [最初のフリップフロップ ステージの移動 \(MOVE_FIRST_STAGE\)](#)
- ・ [最後のフリップフロップ ステージの移動 \(MOVE_LAST_STAGE\)](#)

また、次の制約もレジスタ自動調整に影響を与えます。

- ・ [階層の維持 \(KEEP_HIERARCHY\)](#)
 - － 階層を保持している場合、フリップフロップはブロックの境界内でのみ移動します。
 - － 階層をフラットにした場合、フリップフロップはブロックの境界外にも移動します。
- ・ [I/O レジスタの IOB 内へのパック \(IOB\)](#)

IOB=TRUE の場合、設定したフリップフロップにレジスタ自動調整は適用されません。

- ・ [インスタンス化されたプリミティブの最適化 \(OPTIMIZE_PRIMITIVES\)](#)

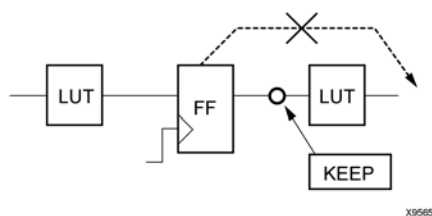
インスタンス化されたフリップフロップは、OPTIMIZE_PRIMITIVES=YES の場合にのみ移動されます。

- ・ フリップフロップは、OPTIMIZE_PRIMITIVES=YES の場合にのみインスタンス化されたプリミティブ間で移動されます。

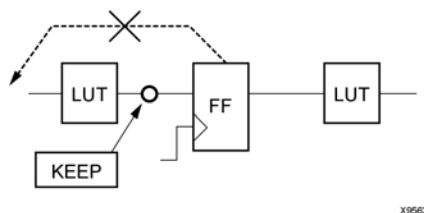
- ・ [キープ \(KEEP\)](#)

この制約を出力フリップフロップ信号に適用した場合、フリップフロップは順方向には移動できません。

入力フリップフロップに適用した場合



出力フリップフロップ信号に適用した場合、フリップフロップは逆方向には移動できません。



フリップフロップの入力と出力の両方に適用するとお、REGISTER_BALANCE=NO と同じになります。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

次に適用されます。

- ・ デザイン全体 (XST コマンド ラインまたは ISE® Design Suite を使用)
- ・ エンティティまたはモジュール
- ・ フリップフロップ記述 (RTL)に対応する信号
- ・ フリップフロップ インスタンス
- ・ プライマリ クロック信号

この場合、レジスタ自動調整はフリップフロップがこのクロックと同期した場合にのみ実行されます。

適用ルール

設定したエンティティ、コンポーネント、モジュール、または信号に適用されます。

構文

`-register_balancing {yes|no|forward|backward}`

- ・ **yes**
順方向および逆方向どちらのリタイミングも可能になります。
- ・ **no** (デフォルト)
フリップフロップのリタイミングはどちらの方向も実行されません。
- ・ **forward**
順方向のリタイミングのみができます。
- ・ **backward**
逆方向のリタイミングのみができます。
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

```
attribute register_balancing: string;
```

次のように指定します。

```
attribute register_balancing of {signal_name|entity_name}: {signal|entity} is  
  "{yes|no|forward|backward}";
```

Verilog

次をモジュールまたは信号宣言の直前に入力します。

```
(* register_balancing = "{yes|no|forward|backward}" *)
```

XCF の構文例 1

```
MODEL "entity_name" register_balancing={yes|no|true|false|forward|backward};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
  NET "primary_clock_signal" register_balancing={yes|no|true|false|forward|backward};  
END;
```

XCF の構文例 3

```
BEGIN MODEL "entity_name"
```

```
INST "instance_name" register_balancing={yes|no|true|false|forward|backward};  
END;
```

XST コマンド ライン

```
xst run -register_balancing {yes|no|forward|backward}
```

ISE Design Suite

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Register Balancing]

レジスタの複製 (REGISTER_DUPLICATION)

レジスタの複製 (REGISTER_DUPLICATION) 制約には、次のような特徴があります。

- ・ レジスタの複製を有効または無効にできます。
- ・ 複製する場合、タイミング最適化およびファンアウト制御の段階でレジスタの複製がされます。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体、エンティティ、コンポーネント、モジュール、信号に適用されます。

適用ルール

設定したエンティティまたはモジュールに適用されます。

構文

- ・ **yes** (デフォルト)
- ・ **no**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

構文例および設定

次の例では、この制約またはコマンド ライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

```
attribute register_duplication: string;
```

次のように指定します。

```
attribute register_duplication of entity_name: entity is "{yes|no}";
```

Verilog

次をモジュール宣言またはインスタンス化の直前に入力します。

```
(* register_duplication = "{yes|no}" *)
```

XCF の構文例 1

```
MODEL "entity_name" register_duplication={yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
NET "signal_name" register_duplication={yes|no|true|false};  
END;
```

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Register Duplication]

ROM の抽出 (ROM_EXTRACT)

ROM_EXTRACT を使用すると、ROM マクロの推論を有効または無効にできます。

ROM は通常、割り当てられた値がすべて定数である case 文から推論されます。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体、またはデザイン エレメント、信号に適用されます。

適用ルール

設定したエンティティ、コンポーネント、モジュール、または信号に適用されます。

構文

```
-rom_extract {yes|no}
```

- ・ **yes** (デフォルト)
- ・ **no**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

```
attribute rom_extract: string;
```

次のように指定します。

```
attribute rom_extract of {signal_name/entity_name} : {signal|entity} is "{yes|no}";
```

Verilog

次をモジュールまたは信号宣言の直前に入力します。

```
(* rom_extract = "{yes|no}" *)
```

XCF の構文例 1

```
MODEL "entity_name" rom_extract={yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"
```

```
NET "signal_name" rom_extract={yes|no|true|false};
```

```
END;
```

XST コマンド ライン

```
xst run -rom_extract {yes|no}
```

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [HDL Options] → [ROM Extraction]

ROM スタイル (ROM_STYLE)

推論された ROM マクロのインプリメント方法を指定します。

注意 : ROM_STYLE を使用する場合、まず ROM の抽出 (ROM_EXTRACT) を yes に設定しておく必要があります。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体、エンティティ、コンポーネント、モジュール、信号に適用されます。

適用ルール

設定したエンティティ、コンポーネント、モジュール、または信号に適用されます。

構文

`-rom_style {auto|block|distributed}`

- ・ **auto** (デフォルト)

推論された各 ROM に対して最適なインプリメント方法が自動設定されます。ほかの値を選択すると、ブロック ROM または分散 ROM を使用するようにできます。

- ・ **block**
- ・ **distributed**

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

attribute rom_style: string;

次のように指定します。

attribute rom_style of {*signal_name*|*entity_name*}: {*signal*|*entity*} is "{auto|block|distributed}";

Verilog

次のように宣言します。

(* rom_style = "{auto|block|distributed}" *)

XCF の構文例 1

MODEL "*entity_name*" rom_style={auto|block|distributed};

XCF の構文例 2

BEGIN MODEL "*entity_name*"

NET "*signal_name*" rom_style={auto|block|distributed};

END;

XST コマンドライン

xst run -rom_style {auto|block|distributed}

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [HDL Options] → [ROM Style]

シフトレジスタの抽出 (SHREG_EXTRACT)

SHREG_EXTRACT (シフトレジスタの抽出) 制約には、次のような特徴があります。

- ・ シフトレジスタのマクロ推論を有効または無効にします。
- ・ SRL16 および SRLC16 のような専用ハードウェア リソースが使用されます。

詳細は、次を参照してください。

[シフトレジスタの HDL コーディング手法](#)

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体、またはデザイン エレメント、信号に適用されます。

適用ルール

設定したエンティティ、コンポーネント、モジュール、または信号に適用されます。

構文

-shreg_extract {yes|no}

- ・ **yes** (デフォルト)
- ・ **no**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

attribute shreg_extract : string;

次のように指定します。

attribute shreg_extract of {*signal_name*|*entity_name*} : {*signal*|*entity*} is "{yes|no}";

Verilog

次をモジュールまたは信号宣言の直前に入力します。

(* shreg_extract = "{yes|no}" *)

XCF の構文例 1

MODEL "*entity_name*" shreg_extract={yes|no|true|false};

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
NET "signal_name" shreg_extract={yes|no|true|false};  
END;
```

XST コマンド ライン

```
xst run -shreg_extract {yes|no}
```

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [HDL Options] → [Shift Register Extraction]

Slice (LUT-FF Pairs) Utilization Ratio (スライス (LUT-FF ペア) 使用率)

SLICE_UTILIZATION_RATIO 制約を使用すると、タイミング最適化におけるエリア サイズの上限を、次のコンポーネントの合計の絶対値またはパーセントで指定できます。

- ・ LUT と FF のペア (Virtex®-5 デバイス)
- ・ スライス (それ以外のデバイス)

このエリア制約を満たすことができない場合は、エリア制約を無視してタイミング最適化が実行されます。自動的にリソースが管理されないようにするには、-1 を指定します。

詳細は、次を参照してください。

[エリア制約を設定した場合のスピード最適化](#)

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体、エンティティ、コンポーネント、モジュール、信号に適用されます。

適用ルール

設定したエンティティまたはモジュールに適用されます。

構文

次のセクションでは、この制約の構文を示します。

構文例および設定

次の例では、この制約またはコマンド ライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

```
attribute slice_utilization_ratio: string;
```

次のように指定します。

```
attribute slice_utilization_ratio of entity_name : entity is "integer";
```

```
attribute slice_utilization_ratio of entity_name : entity is "integer%";
```

```
attribute slice_utilization_ratio of entity_name : entity is "integer#";
```

整数値は最初の 2 つの例ではパーセントとして処理され、最後の例ではスライスまたは FF/LUT ペアの絶対数として処理されます。

% が使用されるか、% と # のどちらも使用されない場合、整数値の範囲は 0 ～ 100 です。

Verilog

次をモジュール宣言またはインスタンス化の直前に入力します。

```
(* slice_utilization_ratio = "integer" *)
```

```
(* slice_utilization_ratio = "integer%" *)
```

```
(* slice_utilization_ratio = "integer#" *)
```

整数値は最初の 2 つの例ではパーセントとして処理され、最後の例ではスライスまたは FF/LUT ペアの絶対数として処理されます。

% が使用されるか、% と # のどちらも使用されない場合、整数値の範囲は 0 ～ 100 です。

XCF

```
MODEL "entity_name" slice_utilization_ratio=integer;
```

```
MODEL "entity_name" slice_utilization_ratio=integer%;
```

```
MODEL "entity_name" slice_utilization_ratio=integer#;
```

整数値は最初の 2 つの例ではパーセントとして処理され、最後の例ではスライスまたは FF/LUT ペアの絶対数として処理されます。

% が使用されるか、% と # のどちらも使用されない場合、整数値の範囲は 0 ～ 100 です。

整数値と % および # 文字の間にはスペースを入れないでください。

% および # は XST Constraint File (XCF) の特殊文字なので、整数値と % または # 文字を二重引用符 (") で囲んでください。

XST コマンドライン

```
xst run -slice_utilization_ratio integer
```

```
xst run -slice_utilization_ratio integer%
```

```
xst run -slice_utilization_ratio integer#
```

整数値は最初の 2 つの例ではパーセントとして処理され、最後の例ではスライスまたは FF/LUT ペアの絶対数として処理されます。

% が使用されるか、% と # のどちらも使用されない場合、整数値の範囲は 0 ～ 100 です。

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Slice Utilization Ratio] または [Process Properties] ダイアログ ボックス → [Synthesis Options] → [LUT-FF Pairs Utilization Ratio] をクリックします。

ISE Design Suite ではこの値を % としてのみ定義できます。スライスの絶対値は指定できません。

スライス (LUT-FF ペア) 使用率の許容範囲 (SLICE_UTILIZATION_RATIO_MAXMARGIN)

スライス (LUT-FF ペア) 使用率の許容範囲 (SLICE_UTILIZATION_RATIO_MAXMARGIN) 制約には、次の特徴があります。

- ・ スライス (LUT-FF ペア) 使用率 (SLICE_UTILIZATION_RATIO) 制約と関連しています。
- ・ スライス (LUT-FF ペア) 使用率 (SLICE_UTILIZATION_RATIO) 制約の許容範囲を定義します。

値は、パーセントのほか、LUT/FF ペアかスライスの絶対数で指定できます。

スライス使用率がこの制約で指定したマージン値の範囲内であれば、制約は満たされていると判断され、タイミング最適化が実行されます。

詳細は、次を参照してください。

[エリア制約を設定した場合のスピード最適化](#)

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体、エンティティ、コンポーネント、モジュール、信号に適用されます。

適用ルール

設定したエンティティまたはモジュールに適用されます。

構文

次のセクションでは、この制約の構文を示します。

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

```
attribute slice_utilization_ratio_maxmargin: string;
```

次のように指定します。

```
attribute slice_utilization_ratio_maxmargin of entity_name : entity is "integer";
```

```
attribute slice_utilization_ratio_maxmargin of entity_name : entity is "integer%";
```

```
attribute slice_utilization_ratio_maxmargin of entity_name : entity is "integer#";
```

整数値は最初の 2 つの例ではパーセントとして処理され、最後の例ではスライスまたは FF/LUT ペアの絶対数として処理されます。

% が使用されるか、% と # のどちらも使用されない場合、整数値の範囲は 0 ～ 100 です。

Verilog

次をモジュール宣言またはインスタンス化の直前に入力します。

```
(* slice_utilization_ratio_maxmargin = "integer" *)
```

```
(* slice_utilization_ratio_maxmargin = "integer%" *)
```

```
(* slice_utilization_ratio_maxmargin = "integer#" *)
```

整数値は最初の 2 つの例ではパーセントとして処理され、最後の例ではスライスまたは FF/LUT ペアの絶対数として処理されます。

% が使用されるか、% と # のどちらも使用されない場合、整数値の範囲は 0 ～ 100 です。

XCF

```
MODEL "entity_name" slice_utilization_ratio_maxmargin=integer;
```

```
MODEL "entity_name" slice_utilization_ratio_maxmargin="integer%";
```

```
MODEL "entity_name" slice_utilization_ratio_maxmargin="integer#";
```

整数値は最初の 2 つの例ではパーセントとして処理され、最後の例ではスライスまたは FF/LUT ペアの絶対数として処理されます。

% が使用されるか、% と # のどちらも使用されない場合、整数値の範囲は 0 ～ 100 です。

整数値と % および # 文字の間にはスペースを入れないでください。

% および # は XST Constraint File (XCF) の特殊文字なので、整数値と % または # 文字を二重引用符 (" ") で囲んでください。

XST コマンドライン

```
xst run -slice_utilization_ratio_maxmargin integer
```

```
xst run -slice_utilization_ratio_maxmargin integer%
```

```
xst run -slice_utilization_ratio_maxmargin integer#
```

整数値は最初の 2 つの例ではパーセントとして処理され、最後の例ではスライスまたは FF/LUT ペアの絶対数として処理されます。

% が使用されるか、% と # のどちらも使用されない場合、整数値の範囲は 0 ～ 100 です。

スライス パッキング (-slice_packing)

スライス パッキング (-slice_packing) コマンドライン オプションを使用すると、XST に含まれる内部パック機能が有効になります。

内部パックは、重要な LUT 間の接続をスライスまたは CLB 内に配置します。これにより、CLB 内の LUT 間の高速フィードバック接続が使用されます。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体に適用されます。

適用ルール

ありません。

構文

`-slice_packing {yes|no}`

- ・ `yes`
- ・ `no`

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンド ライン

`xst run -slice_packing no`

XST の内部パック機能を無効にします。

ISE® Design Suite

[Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックス → [Synthesis Options]
→ [Slice Packing]

ロー スキュー ラインの使用 (USELOWSKEWLINES)

ロー スキュー ラインの使用 (USELOWSKEWLINES) 制約には、次の特徴があります。

- ・ 基本的な配線制約です。
- ・ [MAX_FANOUT \(最大ファンアウト数\)](#) 制約の値に基づいて専用クロック リソースおよびロジックの複製が使用されないようにします。
- ・ ネットでロー スキュー配線リソースを使用するよう指定します。

この制約の詳細は、『[制約ガイド](#)』を参照してください。

キャリーチェーンの使用 (USE_CARRY_CHAIN)

USE_CARRY_CHAIN 制約には、次のような特徴があります。

- ・ グローバルにもローカルにも設定できます。
- ・ マクロ生成時にキャリー チェーンの使用を無効にできます。

XST では、一部のマクロをインプリメントする際にキャリー チェーン リソースが使用されますが、キャリー チェーンを使用しない方が良い結果が得られる場合があります。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体または信号に適用されます。

適用ルール

設定された信号に適用されます。

構文

-use_carry_chain {yes|no}

- ・ **yes** (デフォルト)
- ・ **no**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

構文例および設定

次の例では、この制約またはコマンド ライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

回路図

- ・ 有効なインスタンスに設定します。
- ・ 属性名
USE_CARRY_CHAIN
- ・ 属性値
前述の「構文」セクションを参照してください。

VHDL

次のように宣言します。

attribute use_carry_chain: string;

次のように指定します。

attribute use_carry_chain of *signal_name*: signal is "{yes|no}";

Verilog

次を信号宣言の直前に入力します。

(* use_carry_chain = "{yes|no}" *)

XCF の構文例 1

```
MODEL "entity_name" use_carry_chain={yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
NET "signal_name" use_carry_chain={yes|no|true|false};  
END;
```

XST コマンド ライン

```
xst run -use_carry_chain {yes|no}
```

クロック イネーブルの使用 (USE_CLOCK_ENABLE)

USE_CLOCK_ENABLE を使用すると、フリップフロップのクロック イネーブルの使用を有効または無効にできます。

ASIC のプロトタイプを FPGA で作成する場合は、通常クロック イネーブルを無効にします。

制約値を no または false に設定すると、最終インプリメンテーションでクロック イネーブル (CE) リソースが使用されません。また、デザインによっては、フリップフロップのデータ入力に CE 機能を付けることで、ロジック最適化が向上し、優れた結果品質 (QoR) を実現できることがあります。

auto に設定すると、フリップフロップ入力の専用 CE 入力を使用した方がいいか、フリップフロップの D 入力に CE ロジックを使用した方がいいかが比較検討されます。フリップフロップをインスタシエートすると、OPTIMIZE_PRIMITIVE 制約が yes に設定されている場合にのみ、CE が削除されます。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

次に適用されます。

- ・ XST コマンド ラインからデザイン全体に適用
- ・ 特定ブロック (entity、architecture、component)
- ・ フリップフロップを表す信号
- ・ インスタシエートされたフリップフロップを表すインスタンス

適用ルール

設定したエンティティ、コンポーネント、モジュール、または信号に適用されます。

構文

`-use_clock_enable {auto|yes|no}`

- ・ `auto` (デフォルト)
- ・ `yes`
- ・ `no`
- ・ `true` (XCF のみ)
- ・ `false` (XCF のみ)

構文例および設定

次の例では、この制約またはコマンド ライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

```
attribute use_clock_enable: string;
```

次のように指定します。

```
attribute use_clock_enable of {entity_name/component_name/signal_name/instance_name} :  
{entity|component|signal|label} is "{auto|yes|no}";
```

Verilog

インスタンス、モジュール、または信号宣言の直前に入力します。

```
(* use_clock_enable = "{auto|yes|no}" *)
```

XCF の構文例 1

```
MODEL "entity_name" use_clock_enable={auto|yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
NET "signal_name" use_clock_enable={auto|yes|no|true|false};  
END
```

XCF の構文例 3

```
BEGIN MODEL "entity_name ;"  
INST "instance_name" use_clock_enable={auto|yes|no|true|false};  
END
```

XST コマンド ライン

```
xst run -use_clock_enable {auto|yes|no}
```

ISE® Design Suite

[Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックス → [Synthesis Options]
→ [Use Clock Enable]

USE_DSP48 (DSP48 の使用)

この制約は、次のように呼ばれます。

- ・ DSP48 の使用
Virtex®-4 デバイス
- ・ [Use DSP Block]
Virtex-5、Spartan®-3A DSP デバイス

Virtex-4 以降のデバイスに含まれる DSP48 ブロックリソースの使用を有効にします。

デフォルト値の auto を使用した場合、MAC などのマクロは DSP48 に自動的にインプリメントされますが、加算器など一部のマクロはスライスにインプリメントされます。これらのマクロを DSP48 にインプリメントするには、USE_DSP48 を yes または true に設定する必要があります。

サポートされるマクロおよびインプリメンテーションの制御については、次を参照してください。

HDL コーディング手法

MAC など DSP48 に配置できるマクロは、乗算器、アキュムレータ、レジスタなどの単純なマクロから構成されています。最適なパフォーマンスを得るため、XST はマクロ コンフィギュレーションを最大限に推論、インプリメントしようとします。マクロを特定の方法でインプリメントするには、**キープ (KEEP)** 制約を使用する必要があります。たとえば、DSP48 には 2 つの入力レジスタを使用した乗算器をインプリメントできますが、最初のレジスタ ステージを DSP48 の外に残したままにするには、出力に **キープ (KEEP)** 制約を設定する必要があります。

アーキテクチャ サポート

次のデバイスにのみ適用できます。これ以外のデバイスには適用できません。

- ・ Spartan-3A DSP
- ・ Virtex-4
- ・ Virtex-5

適用可能エレメント

次に適用されます。

- ・ XST コマンド ラインからデザイン全体に適用
- ・ 特定ブロック (entity、architecture、component)
- ・ RTL レベルで記述されるマクロを表す信号

適用ルール

設定したエンティティ、コンポーネント、モジュール、または信号に適用されます。

構文

`-use_dsp48 {auto|yes|no}`

- ・ **auto** (デフォルト)
- ・ **yes**
- ・ **no**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

また、auto モードにすると、**DSP 使用率 (DSP_UTILIZATION_RATIO)** 制約を使用して合成で使用可能な DSP48 リソースの数が制御されます。デフォルトでは、すべての使用可能な DSP48 リソースが可能な限り使用されます。

詳細は、次を参照してください。

[DSP48 ブロック リソース](#)

構文例および設定

次の例では、この制約またはコマンド ライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

```
attribute use_dsp48: string;
```

次のように指定します。

```
attribute use_dsp48 of {"entity_name/component_name/signal_name"} : {entity|component|signal}  
is "{auto|yes|no}";
```

Verilog

次をモジュールまたは信号宣言の直前に入力します。

```
(* use_dsp48 = "{auto|yes|no}" *)
```

XCF の構文例 1

```
MODEL "entity_name" use_dsp48={auto|yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
NET "signal_name" use_dsp48={auto|yes|no|true|false};  
END;
```

XST コマンド ライン

```
xst run -use_dsp48 {auto|yes|no}
```

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [HDL Options] → [Use DSP48]

同期セットの使用 (USE_SYNC_SET)

フリップフロップの同期セットの使用を有効または無効にします。

ASIC のプロトタイプを FPGA で作成する場合は、通常同期セット機能を無効にします。制約値を no または false に設定すると、最終インプリメンテーションで同期セットリソースが使用されません。また、デザインによっては、フリップフロップのデータ入力に同期リセット機能を付けることで、ロジック最適化が向上し、優れた結果品質 (QoR) を実現できることがあります。

auto に設定すると、フリップフロップ入力の専用同期セット入力を使用した方がいいか、フリップフロップの D 入力に同期セットロジックを使用した方がいかが比較検討されます。フリップフロップがインスタンス化されると、XST では **インスタンス化されたプリミティブの最適化 (OPTIMIZE_PRIMITIVES)** が yes の場合にのみ、同期リセットを削除します。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

次に適用されます。

- ・ XST コマンド ラインからデザイン全体に適用
- ・ 特定ブロック (entity、architecture、component)
- ・ フリップフロップを表す信号
- ・ インスタンス化されたフリップフロップを表すインスタンス

適用ルール

設定したエンティティ、コンポーネント、モジュール、または信号に適用されます。

構文

-use_sync_set {auto|yes|no}

- ・ **auto** (デフォルト)
- ・ **yes**
- ・ **no**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

構文例および設定

次の例では、この制約またはコマンド ライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

```
attribute use_sync_set: string;
```

次のように指定します。

```
attribute use_sync_set of {entity_name/component_name/signal_name/instance_name}:  
{entity|component|signal|label} is "{auto|yes|no}";
```

Verilog

次をモジュールまたは信号宣言の直前に入力します。

```
(* use_sync_set = "{auto|yes|no}" *)
```

XCF の構文例 1

```
MODEL "entity_name" use_sync_set={auto|yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
NET "signal_name" use_sync_set={auto|yes|no|true|false};  
END;
```

XCF の構文例 3

```
BEGIN MODEL "entity_name"  
INST "instance_name" use_sync_set={auto|yes|no|true|false };  
END;
```

XST コマンド ライン

```
xst run -use_sync_set {auto|yes|no}
```

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Use Synchronous Set]

同期リセットの使用 (USE_SYNC_RESET)

フリップフロップの同期リセットの使用を有効または無効にします。

ASIC のプロトタイプを FPGA で作成する場合は、通常同期セット機能を無効にします。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

次に適用されます。

- ・ XST コマンド ラインからデザイン全体に適用
- ・ 特定ブロック (entity、architecture、component)
- ・ フリップフロップを表す信号
- ・ インスタンシエートされたフリップフロップを表すインスタンス

適用ルール

設定したエンティティ、コンポーネント、モジュール、または信号に適用されます。

構文

-use_sync_reset {auto|yes|no}

- ・ **auto** (デフォルト)
- ・ **yes**
- ・ **no**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

制約値を no または false に設定すると、最終インプリメンテーションで同期リセットリソースが使用されません。また、デザインによっては、フリップフロップのデータ入力に同期リセット機能を付けることで、ロジック最適化が向上し、優れた結果品質 (QoR) を実現できることがあります。

auto に設定すると、フリップフロップ入力の専用同期リセット入力を使用した方がいいか、フリップフロップの D 入力に 同期リセット ロジックを使用した方がいいかが比較検討されます。フリップフロップがインスタンシエートされると、XST では [インスタンシエートされたプリミティブの最適化 \(OPTIMIZE_PRIMITIVES\)](#) が yes の場合にのみ、同期リセットを削除します。

構文例および設定

次の例では、この制約またはコマンド ライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

```
attribute use_sync_reset: string;
```

次のように指定します。

```
attribute use_sync_reset of {entity_name/component_name/signal_name/instance_name} :  
{entity|component|signal|label} is "{auto|yes|no}";
```

Verilog

次をモジュールまたは信号宣言の直前に入力します。

```
(* use_sync_reset = "{auto|yes|no}" *)
```

XCF の構文例 1

```
MODEL "entity_name" use_sync_reset={auto|yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"  
NET "signal_name" use_sync_reset={auto|yes|no|true|false};  
END;
```

XCF の構文例 3

```
BEGIN MODEL "entity_name"  
INST "instance_name" use_sync_reset={auto|yes|no|true|false};  
END;
```

XST コマンド ライン

```
xst run -use_sync_reset {auto|yes|no}
```

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Use Synchronous Reset]

XOR コラプス (XOR_COLLAPSE)

カスケード接続された XOR を 1 つの XOR にまとめるかどうかを指定します。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

カスケードされた XOR に適用されます。

適用ルール

ありません。

構文

```
-xor_collapse {yes|no}
```

- ・ **yes** (デフォルト)
- ・ **no**
- ・ **true** (XCF のみ)
- ・ **false** (XCF のみ)

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

```
attribute xor_collapse: string;
```

次のように指定します。

```
attribute xor_collapse {signal_name|entity_name} : {signal|entity} is "[yes|no]";
```

Verilog

次をモジュールまたは信号宣言の直前に入力します。

```
(* xor_collapse = "[yes|no]" *)
```

XCF の構文例 1

```
MODEL "entity_name" xor_collapse={yes|no|true|false};
```

XCF の構文例 2

```
BEGIN MODEL "entity_name"
```

```
NET "signal_name" xor_collapse={yes|no|true|false};
```

```
END;
```

XST コマンド ライン

```
xst run -xor_collapse {yes|no}
```

ISE® Design Suite

[Process Properties] ダイアログ ボックス → [HDL Options] → [XOR Collapsing]

CPLD 制約（タイミング以外）

重要：この章に記述される制約は、CPLD デバイスにのみに適用されます。FPGA には適用されません。

この章では、次の制約について説明しています。

- ・ クロック イネーブル (`-pld_ce`)
- ・ データ ゲート (`DATA_GATE`)
- ・ マクロの保持 (`-pld_mp`)
- ・ 削減なし (`NOREDUCE`)
- ・ WYSIWYG (`-wysiwyg`)
- ・ XOR の保持 (`-pld_xp`)

クロック イネーブル (`-pld_ce`)

クロック イネーブル (`-pld_ce`) コマンドライン オプションでは、順序ロジックにクロック イネーブルが含まれる場合のインプリメント方法を指定します。専用のデバイスリソースを使用するか、または等価ロジックを生成するかを指定できます。

クロック イネーブル信号を使用するかしないかは、デザイン ロジックによって判断します。クロック イネーブルがブール代数式の結果である場合、フリップフロップの入力データはクロック イネーブル表現と結合すると簡略化されるため、このオプションを `no` に設定した方がフィットの結果が向上する可能性があります。

アーキテクチャ サポート

すべての CPLD デバイスに適用されます。FPGA には適用できません。

適用可能エレメント

XST コマンドラインでデザイン全体に適用されます。

適用ルール

ありません。

構文

`-pld_ce {yes|no}`

- ・ **yes** (デフォルト)
デバイスのクロック イネーブル信号を使用してインプリメントします。
- ・ **no**
等価ロジックを使用してインプリメントします

構文例および設定

```
xst run -pld_ce yes
```

クロック イネーブルをグローバルに yes に定義すると、クロック イネーブル ファンクションが同等のロジック全体にインプリメントされます

ISE® Design Suite

[Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックス → [Synthesis Options]
→ [Clock Enable]

データ ゲート (DATA_GATE)

DATA_GATE 制約には、次のような特徴があります。

- ・ CoolRunner™-II デバイスにのみ適用できます。
- ・ デザインの消費電力を低減することができます。

入力信号の遷移が CPLD デザインのファンクションに関係ない場合に、信号の遷移が伝搬されないようブロックするラッチが使用できるようになります。

入力が遷移すると、その遷移がデザインのファンクションに影響しない場合でも、CPLD のファンクション ブロックに配線されているため、電力を消費します。DataGate ラッチ ライブラリ プリミティブを I/O ピンの入力に接続することにより、このラッチのイネーブル ピンをアサートしたときに、信号の遷移がブロックされ、電力が消費されないようにすることができます。

この制約の詳細は、『[制約ガイド](#)』を参照してください。

マクロの保持 (-pld_mp)

マクロの保持 (-pld_mp) コマンドライン オプションには、次の特徴があります。

- ・ デザイン階層の処理とは別に、マクロを処理できるように指定できます。
- ・ マクロを階層モジュールとして保持しながら、すべての階層ブロックを最上位モジュールに結合できます。

周辺のロジックと結合されるマクロを除き、デザイン階層を保持できます。マクロを周辺ロジックと結合した方が、デザインのフィットで良い結果が得られる場合があります。

アーキテクチャ サポート

すべての CPLD デバイスに適用されます。FPGA には適用できません。

適用可能エレメント

デザイン全体に適用されます。

適用ルール

ありません。

構文

`-pld_mp {yes|no}`

- ・ **yes** (デフォルト)
マクロが保持され、マクロ生成機能により生成されます。
- ・ **no**
マクロが保持されず、HDL 合成機能により生成されます。

[Flatten Hierarchy] の値によって、保持されないマクロは、デザイン ロジックに結合されるか、階層ブロックになります。

[Flatten Hierarchy] の値	処理
yes	デザイン ロジックに結合されます。
no	階層ブロックになります。

2 ビット加算器、4 ビット マルチプレクサなどの小型のマクロは、[Macro Preserve] および [Flatten Hierarchy] の設定に関係なく、常に周辺ロジックと結合されます。

構文例および設定

次の例では、この制約またはコマンド ライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンド ライン

`xst run -pld_mp no`

マクロは保持されず、HDL 合成機能により生成されます。

ISE® Design Suite

[Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックス → [Synthesis Options] → [Macro Preserve]

削減なし (NOREDUCE)

NOREDUCE (削減なし) には、次の特徴があります。

- ・ ロジック ハザードやレース コンディションを避けるためにデザインに含まれている冗長な論理記述が最小化されないように指定します。
- ・ 適正なマップが実行されるように組み合わせフィードバック ループの出力ノードが識別されます。

この制約の詳細は、『[制約ガイド](#)』を参照してください。

WYSIWYG (-wysiwyg)

WYSIWYG (**-wysiwyg**) コマンドライン オプションを使用すると、ユーザー指定を最大限に反映させたネットリストが作成できます。つまり、Hardware Description Language (HDL) デザインで宣言されたすべてのノードが保持されます。

yes に設定すると、XST では次が実行されます。

- ・ すべてのユーザー内部信号 (ノード) が保持されます。
- ・ NGC ファイルにこれらのノードすべてに対して SOURCE_NODE 制約が作成されます。
- ・ コラプス、因数分解などのデザイン最適化は実行されません。

ブール代数式の最小化のみが実行されます

アーキテクチャ サポート

すべての CPLD デバイスに適用されます。FPGA には適用できません。

適用可能エレメント

XST コマンドラインでデザイン全体に適用されます。

適用ルール

ありません。

構文

- ・ **yes**
- ・ **no** (デフォルト)

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンドライン

```
xst run -wysiwyg {yes|no}
```

ISE® Design Suite

[Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックス → [Synthesis Options] → [WYSIWYG]

XOR の保持 (-pld_xp)

XOR の保持 (-pld_xp) コマンドライン オプションを使用すると、XOR マクロ階層のフラット化を有効または無効にできます。

CPLD フローでは、Hardware Description Language (HDL) 合成で推論された XOR もマクロブロックと見なされますが、デバイスのマクロセルの XOR ゲートをより柔軟に使用できるようにするため、別に処理されます。このため [Flatten Hierarchy] をオン (yes) にし [Macro Preserve] をオフ (no) にして、デザインをフラットにすることもできますが、ザイリンクスでは XOR ゲートを保持することをお勧めします。XOR ゲートを保持すると、次が実現可能です。

- ・ デザインの複雑さを削減
- ・ 積項の数を削減

完全にフラットなネットリストが必要な場合は、no を設定してください。完全にフラット化されたデザインでグローバル最適化を実行することにより、デザインのフィットが向上する場合があります。

アーキテクチャ サポート

すべての CPLD デバイスに適用されます。FPGA には適用できません。

適用可能エレメント

デザイン全体に適用されます。

適用ルール

ありません。

構文

-pld_xp {yes|no}

- ・ **yes** (デフォルト)
- ・ **no**

yes にすると、XOR マクロが保持されます。

no にすると、XOR マクロが周辺ロジックと結合されます。

次のように設定すると、完全にフラット化されたデザインが生成されます。

- ・ [Flatten Hierarchy]

yes

- ・ [Macro Preserve]

no

- ・ XOR 保持

no

ただし、no にしていても、必ずしも XOR 演算子が Electronic Data Interchange Format (EDIF) ネットリストから削除されるわけではありません。ネットリスト生成の段階では、ロジックを簡略化するため、XOR ゲートの推論が試みられます。この処理は、HDL 合成段階で行われる XOR の保持とは関係なく、ロジックを簡略化する目的で行われます。

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンドライン

```
xst run -pld_xp yes
```

XOR マクロが保持されます。

ISE® Design Suite

[Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックス → [Synthesis Options]
→ [XOR Preserve]

タイミング制約

この章では、次の制約について説明しています。

- ・ クロック信号 (CLOCK_SIGNAL)
- ・ クロス クロック解析 (-cross_clock_analysis)
- ・ From-To (FROM-TO)
- ・ グローバルな最適化目標 (-glob_opt)
- ・ オフセット (OFFSET)
- ・ 周期 (PERIOD)
- ・ タイミング名 (TNM)
- ・ ネットのタイミング名 (TNM_NET)
- ・ タイムグループ (TIMEGRP)
- ・ タイミング無視 (TIG)
- ・ タイミング制約の書き込み (-write_timing_constraints)

詳細は、次を参照してください。

- ・ タイミング制約の指定
- ・ XCF のタイミング制約のサポート

タイミング制約の指定

このセクションには、次の項目が含まれます。

- ・ [タイミング制約の指定の概要](#)
- ・ [グローバル最適化オプションを使用したタイミング制約の指定](#)
- ・ [UCF を使用したタイミング制約の指定](#)
- ・ [NGC ファイルへの制約の書き込み](#)
- ・ [タイミング制約の処理に影響のあるその他のオプション](#)

タイミング制約の指定の概要

XST でサポートされるタイミング制約は、次の方法で指定できます。

- ・ [グローバルな最適化目標 \(-glob_opt\)](#)
- ・ ISE® Design Suite での設定：
[Process] → [Process Properties] → [Synthesis Options] → [Global Optimization Goal]
- ・ User Constraints File (UCF)

グローバル最適化オプションを使用したタイミング制約の指定

[グローバルな最適化目標 \(-glob_opt\)](#) を使用すると、次の 5 つのグローバル タイミング制約を使用できます。

- ・ ALLCLOCKNETS
- ・ OFFSET_IN_BEFORE
- ・ OFFSET_OUT_AFTER
- ・ INPAD_TO_OUTPAD
- ・ MAX_DELAY

これらの制約は、デザイン全体にグローバルに適用されます。XST では、最適のパフォーマンスを目標として最適化が実行されるため、これらの制約に値を設定することはできません。これらの制約は、UCF ファイルで指定された制約で上書きされます。

UCF を使用したタイミング制約の指定

UCF ファイルからは、ネイティブ UCF 構文を使用してタイミング制約を指定できます。XST では、次の制約がサポートされます。

- ・ [タイミング名 \(TNM\)](#)
- ・ [タイムグループ \(TIMEGRP\)](#)
- ・ [周期 \(PERIOD\)](#)
- ・ [タイミング無視 \(TIG\)](#)
- ・ [From-To \(FROM-TO\)](#)

XST では、これらの制約でワイルドカードや階層名を使用できます。

NGC ファイルへの制約の書き込み

タイミング制約は、デフォルトのままでは NGC ファイルに書き込まれません。タイミング制約は、次の設定をしている場合にのみ NGC ファイルに書き込まれます。

- ・ ISE Design Suite の [Process Properties] ダイアログ ボックスの [Synthesis Options] ページにある [Write Timing Constraints] をオン
- ・ コマンド ラインの場合は、`-write_timing_constraints` コマンド ライン オプションを使用

タイミング制約の処理に影響のあるその他のオプション

タイミング制約の指定方法にかかわらず、タイミング制約の処理に影響を与えるオプションに次の 3 つがあります。

- ・ クロス クロック解析 (`-cross_clock_analysis`)
- ・ タイミング制約の書き込み (`-write_timing_constraints`)
- ・ クロック信号 (`CLOCK_SIGNAL`)

XCF のタイミング制約のサポート

このセクションでは、XCF のタイミング制約サポートについて説明します。

- ・ 階層区切り文字
- ・ サポートされるタイミング制約
- ・ サポートされないタイミング制約

階層区切り文字

XCF ファイルでタイミング制約を指定する場合、階層の区切り記号にはアンダースコア (`_`) ではなく、スラッシュ (`/`) を使用してください。

詳細は、次を参照してください。

[階層区切り文字の指定 \(`-hierarchy_separator`\)](#)

サポートされるタイミング制約

XCF では、次のタイミング制約がサポートされます。

- ・ 周期 (`PERIOD`)
- ・ オフセット (`OFFSET`)
- ・ From-To (`FROM-TO`)
- ・ タイミング名 (`TNM`)
- ・ ネットのタイミング名 (`TNM_NET`)
- ・ タイムグループ (`TIMEGRP`)
- ・ タイミング無視 (`TIG`)

サポートされないタイミング制約

XST で指定したタイミング制約のすべてまたは一部がサポートされない場合は、次が実行されます。

- ・ 警告メッセージが表示されます。
- ・ タイミング最適化段階でサポートされていないタイミング制約またはサポートされていない部分は無視されます。

[Write Timing Constraints] (`-write_timing_constraints`) プロパティを `yes` (チェックボックスはオン) に設定している場合は、タイミング最適化の段落で無視された制約も含め、すべての制約が最終的なネットリストに記述されます。

クロック信号 (CLOCK_SIGNAL)

クロック信号 (CLOCK_SIGNAL) 制約を使用すると、クロック信号がフリップフロップのクロック入力に接続される前に組み合わせロジックを通過する場合に、そのクロック信号を定義できます。

この場合、実際にクロック信号となる入力ピンまたは内部信号は XST 認識できないので、ユーザーが手動で定義する必要があります。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

信号に適用されます。

適用ルール

クロック信号に適用されます。

構文

- ・ `yes` (デフォルト)
- ・ `no`
- ・ `true` (XCF のみ)
- ・ `false` (XCF のみ)

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

VHDL

次のように宣言します。

```
attribute clock_signal : string;
```

次のように指定します。

属性 `clock_signal` of *signal_name* : `signal` is {yes|no};

Verilog

次を信号宣言の直前に入力します。

(* `clock_signal` = "{yes|no}" *)

XCF

BEGIN MODEL "*entity_name*"

NET "*primary_clock_signal*" `clock_signal`=`{yes|no|true|false}`;

END;

クロス クロック解析 (-cross_clock_analysis)

クロス クロック解析 (-cross_clock_analysis) コマンド ライン オプションを使用すると、タイミングの最適化中に複数のクロックドメイン間を解析できます。

アーキテクチャ サポート

すべての FPGA デバイスに適用されます。CPLD には適用できません。

適用可能エレメント

デザイン全体に適用されます。

適用ルール

ありません。

構文

`-cross_clock_analysis {yes|no}`

- **yes**
- **no** (デフォルト)

構文例および設定

次の例では、この制約またはコマンド ライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンド ライン

`xst run -cross_clock_analysis yes`

タイミングの最適化中に複数のクロックドメイン間の解析が実行されます。

ISE® Design Suite

[Process Properties] ダイアログ ボックスの [Synthesis Options] ページにある [Cross Clock Analysis]

From-To (FROM-TO)

2 つのグループ間のタイミング制約を設定します。

この場合のグループは、ユーザー定義のグループまたは定義済みのグループです。

- ・ FF
- ・ PAD
- ・ RAM

この制約の詳細は、『[制約ガイド](#)』を参照してください。

構文例

TIMESPEC *TSname* = **FROM** *group1* **TO** *group2 value*;

グローバルな最適化目標 (-glob_opt)

[Global Optimization Goal] オプション (-glob_opt コマンドライン オプション) で、グローバル最適化の要件を指定できます。

XST では、この制約に基づいて次のデザイン部分を最適化できます。

- ・ レジスタからレジスタ
- ・ 入力パッドからレジスタ
- ・ レジスタから出力パッド
- ・ 入力パッドから出力パッド

サポートされるタイミング制約の詳細は、次を参照してください。

[パーティション](#)

次の制約を適用できます。

- ・ ALLCLOCKNETS (レジスタからレジスタ)
デザイン全体の周期を最適化します。
デザイン内のすべてのクロックに対し、同じクロックパス上にあるレジスタ間のすべてのパスが指定されます。これがデフォルト設定です。クロックドメイン遅延を考慮に入れるには、[クロスクロック解析 \(-cross_clock_analysis\)](#) を yes に設定します。
- ・ OFFSET_IN_BEFORE (入力パッドからレジスタ)
特定クロックまたはデザイン全体に対して、入力パッドからクロックまでの最大遅延を最適化します。
XST では、すべての順次エレメントまたは指定したクロック信号名で駆動される特定の順次エレメントからプライマリ出力ポートまでのパスがすべて識別されます。
- ・ OFFSET_OUT_AFTER (レジスタから出力パッド)
特定クロックまたはデザイン全体に対して、クロックから出力パッドまでの最大遅延を最適化します。
プライマリ入力ポートからすべての順次エレメント、または指定したクロック信号名で駆動される特定の順次エレメントまでのパスが指定されます。
- ・ INPAD_TO_OUTPAD (入力パッドから出力パッド)
デザイン全体の入力パッドから出力パッドまでの最大遅延を最適化します。
- ・ MAX_DELAY
上記 4 つの制約を統合したものです。

これらの制約はデザイン全体に影響し、制約ファイルでタイミング制約が指定されていない場合にのみ、適用されます。

構文

`-glob_opt {allclocknets|offset_in_before|offset_out_after|inpad_to_outpad|max_delay}`

この制約には、値を指定できません。最適なパフォーマンスを得るため、デザイン全体が最適化されます。

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンドライン

`xst run -glob_opt OFFSET_OUT_AFTER`

デザイン全体のクロックから出力パッドまでの最大遅延を最適化します。

ISE® Design Suite

[Process] → [Process Properties] → [Synthesis Options] → [Global Optimization Goal]

グローバル最適化のドメインの定義

指定できるドメインは次のとおりです。

- ・ ALLCLOCKNETS (レジスタからレジスタ)

デザイン内のすべてのクロックに対し、同じクロックパス上にあるレジスタ間のすべてのパスが指定されます。これがデフォルト設定です。クロックドメイン遅延を考慮に入れるには、[クロスクロック解析 \(-cross_clock_analysis\)](#) を yes に設定します。

- ・ OFFSET_IN_BEFORE (入力パッドからレジスタ)

プライマリ入力ポートからすべての順次エレメント、または指定したクロック信号名で駆動される特定の順次エレメントまでのパスが指定されます。

- ・ OFFSET_OUT_AFTER (レジスタから出力パッド)

順次エレメントからプライマリ出力ポートまでのパスが指定されます。

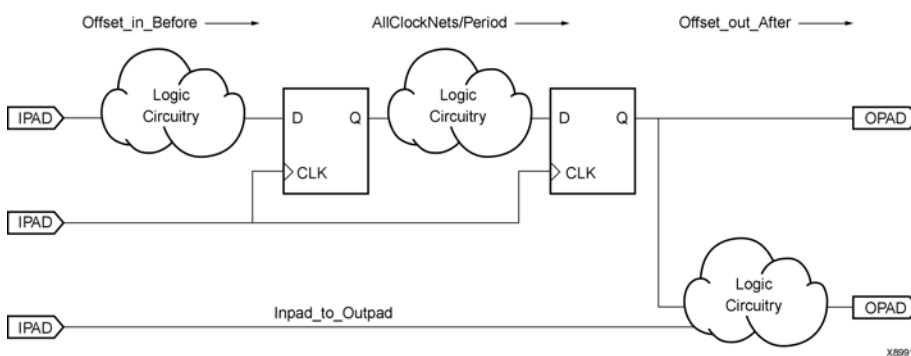
- ・ INPAD_TO_OUTPAD (入力パッドから出力パッド)

最大組み合わせパス制約が指定されます。

- ・ MAX_DELAY

次のタイミング制約で定義されるすべてのパスが指定されます。

- ALLCLOCKNETS
- OFFSET_IN_BEFORE
- OFFSET_OUT_AFTER
- INPAD_TO_OUTPAD



オフセット (OFFSET)

OFFSET (オフセット) 制約には、次の特徴があります。

- ・ タイミング制約です。
- ・ 外部クロックと関連するデータ入力ピンまたはデータ出力ピンとのタイミング関係を指定します。
- ・ パッド関連の信号にのみ使用されます。
- ・ デザインの内部信号への信号到着時間は指定できません。
- ・ 次が実行できます。
 - 外部ネットからデータ入力とクロック入力 that 供給されるフリップフロップで、セットアップ タイム要件が満たされているかを計算
 - 外部デバイス ピンをクロック ソースとする内部フリップフロップの Q 出力から生成された外部出力ネットの遅延を指定

この制約の詳細は、『[制約ガイド](#)』を参照してください。

構文

```
OFFSET = {IN|OUT} offset_time [units] {BEFORE|AFTER} clk_name [TIMEGRP group_name];
```

周期 (PERIOD)

PERIOD は、基本的なタイミング制約および合成制約です。

PERIOD 制約を指定すると、デスティネーション エLEMENTのグループで定義されているクロックドメイン内で、すべての同期ELEMENT間のタイミングが確認されます。クロックが別のクロックの関数として定義されている場合、グループには複数のクロックドメインを通過するパスが含まれます。

この制約の詳細は、『[制約ガイド](#)』を参照してください。

構文

```
NET netname PERIOD = value [{HIGH|LOW} value];
```

タイミング名 (TNM)

タイミング名 (TNM) 制約には、次の特徴があります。

- ・ 基本的なグループ制約です。
- ・ タイミング仕様で使用されるグループを構成するエレメントを指定します。
- ・ 次の特定のエレメントをグループ化することにより、タイミング仕様の適用を簡略化できます。
 - FF
 - RAM
 - LATCH
 - PAD
 - CPU
 - BRAM_PORTA
 - BRAM_PORTB
 - HSIO
 - MULT

この制約は、キーワード RISING および FALLING と一緒に使用できます。

この制約の詳細は、『[制約ガイド](#)』を参照してください。

構文

```
{INST|NET|PIN} inst_net_or_pin_name TNM = [predefined_group:] identifier;
```

ネットのタイミング名 (TNM_NET)

TNM_NET (タイミング名ネット) 制約には、次の特徴があります。

- ・ 入力パッド ネットに設定する場合を除き、ネットに設定した [タイミング名 \(TNM\)](#) と基本的に同等です。

メモ： TNM_NET を DLL および DCM コンポーネントに対し、[PERIOD](#) 制約と使用する場合は、特別なルールが適用されます。

詳細は、次を参照してください。

[『制約ガイド』](#)の「CLKDLL および DCM での PERIOD 指定」

- ・ 通常、特定ネットを指定するため HDL デザインで使用するプロパティです。

メモ： TNM_NET で指定されたダウンストリームの同期エレメントおよびパッドは、すべてグループと見なされます。

この制約の詳細は、『[制約ガイド](#)』を参照してください。

構文

```
NET netname TNM_NET = [predefined_group:] identifier;
```

タイムグループ (TIMEGRP)

TIMEGRP は、基本的なグループ制約です。

TNM 識別子を使用したグループの作成に加え、ほかのグループを基にしてグループを定義できます。TIMEGRP 制約を使用すると、既存グループを組み合わせてグループを作成できます。

TIMEGRP 制約は XST Constraint File (XCF) または Netlist Constraints File (NCF) に含めます。

TIMEGRP 制約を使用したグループの作成は、次のいずれかの方法で行います。

- ・ 複数のグループを 1 つのグループにまとめる
- ・ クロックの立ち上がり/立ち下がりによってフリップフロップのサブグループを指定する

この制約の詳細は、『[制約ガイド](#)』を参照してください。

構文

```
TIMEGRP newgroup = existing_grp1 existing_grp2 [existing_grp3 ...];
```

タイミング無視 (TIG)

タイミング無視 (TIG) 制約には、次の特徴があります。

- ・ タイミング解析および最適化で、制約を設定したネットを通過するパスが無視されます。
- ・ 無視する信号の名前に適用します。

この制約の詳細は、『[制約ガイド](#)』を参照してください。

構文

```
NET net_name TIG;
```

タイミング制約の書き込み (-write_timing_constraints)

タイミング制約の書き込み (-write_timing_constraints) コマンドライン オプションでは、NGC ファイルへタイミング制約が書き込まれるタイミングを指定できます。

タイミング制約は、次の設定をしている場合にのみ NGC ファイルに書き込まれます。

- ・ ISE Design Suite の [Process Properties] ダイアログ ボックスの [Synthesis Options] ページにある [Write Timing Constraints] をオン
- ・ コマンドラインの場合は、-write_timing_constraints コマンドライン オプションを使用

タイミング制約は、デフォルトのままでは NGC ファイルに書き込まれません。

アーキテクチャ サポート

アーキテクチャに依存しません。

適用可能エレメント

XST コマンドラインでデザイン全体に適用されます。

適用ルール

ありません。

構文

`-write_timing_constraints {yes|no}`

- ・ `yes` (デフォルト)
- ・ `no`

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XST コマンドライン

`xst run -write_timing_constraints yes`

タイミング制約が NGC ファイルに書き込まれます。

ISE Design Suite

[Process Properties] ダイアログ ボックス → [Synthesis Options] → [Write Timing Constraints]

インプリメンテーション制約

この章では、次の制約について説明しています。

- ・ 削減なし (NOREDUCE)
- ・ PWR_MODE
- ・ RLOC

インプリメンテーション制約は、配置および配線を制御します。これらの制約は XST では直接使用されず、インプリメンテーション ツールに渡されて使用されます。また、インプリメンテーション制約が設定されたオブジェクトは保持されます。

インプリメンテーション制約と同等のバイナリコードが NGC ファイルに記述されます。ファイルはバイナリなので、NGC ファイルでインプリメンテーション制約を編集することはできません。

XST Constraint File (XCF) のインプリメンテーション制約のコード記述については、「[インプリメンテーション制約の構文例](#)」を参照してください。

詳細は、『[制約ガイド](#)』を参照してください。

インプリメンテーション制約の構文例

このセクションでは、インプリメンテーション制約の構文例を示します。

- ・ [XCF の構文例](#)
- ・ [VHDL の構文例](#)
- ・ [Verilog の構文例](#)

XCF の構文例

このセクションでは、「方法 1」と「方法 2」の 2 つをそれぞれ含む 2 種類の XCF 構文例を示します。

XCF の構文例 1

制約をエンティティ全体に適用するには、次のいずれかの XST Constraint File (XCF) 構文を使用します。

方法 1

```
MODEL EntityName PropertyName;
```

方法 2

```
MODEL EntityName PropertyName=PropertyValue;
```

XCF の構文例 2

エンティティ内の特定のインスタンス、ネット、またはピンに制約を適用するには、次のいずれかの構文を使用します。

方法 1

```
BEGIN MODEL EntityName {NET|INST|PIN} {NetName/InstName/SigName} PropertyName;
END;
```

方法 2

```
BEGIN MODEL EntityName {NET|INST|PIN} {NetName/InstName/SigName}
PropertyName=PropertyValue;
END;
```

VHDL の構文例

VHDL の場合、次のように指定する必要があります。

```
attribute PropertyName of {NetName/InstName/PinName} : {signal|label} is "PropertyValue";
```

Verilog の構文例

Verilog の場合、次のように指定する必要があります。

```
// synthesis attribute PropertyName of {NetName/InstName/PinName} is "PropertyValue";
```

Verilog-2001 の場合、参照する信号、モジュール、またはインスタンスの前で次のように指定する必要があります。

```
(* PropertyName="PropertyValue" *)
```

削減なし (NOREDUCE)

すべての CPLD デバイスに適用されます。FPGA には適用できません。

指定した信号を生成するブール代数式が最適化されないように設定します。たとえば、ローカル信号を次のファンクションに割り当て、NOREDUCE 制約を信号 s に設定する場合、次のように指定します。

```
signal s : std_logic;
attribute NOREDUCE : boolean;
attribute NOREDUCE of s : signal is "true";
...
s <= a or (a and b);
```

XCF で NOREDUCE を次のように指定します。

```
BEGIN MODEL ENTNAME
  NET s NOREDUCE;
  NET s KEEP;
END;
```

この場合、XST は次の文を NGC ファイルに記述します。

```
NET s NOREDUCE;  
NET s KEEP;
```

詳細は、『[制約ガイド](#)』を参照してください。

PWR_MODE

PWR_MODE 制約を使用すると、マクロセルの電力消費特性を制御できます。

VHDL で次のように指定すると、信号 s を生成するファンクションで消費電力が低減されるように最適化されます。

```
attribute PWR_MODE : string;  
attribute PWR_MODE of s : signal is "LOW";
```

この場合、XST は次の文を NGC ファイルに記述します。

```
NET s PWR_MODE=LOW;  
NET s KEEP;
```

次の場合、XST では HDL 属性が信号に適用され、インスタンスが推論されます。

- ・ 属性がインスタンスに適用されている場合、および
- ・ インスタンスが HDLソースにない (インスタンス化されていない) 場合

インスタンスの例は、次のとおりです。

- ・ [I/O レジスタの IOB 内へのパック \(IOB\)](#)
- ・ DRIVE
- ・ IOSTANDARD

アーキテクチャ サポート

すべての CPLD デバイスに適用されます。FPGA には適用できません。

構文例および設定

次の例では、この制約またはコマンドライン オプションを特定のツールまたは手法でどのように設定するかを示しています。ツールや手法が記述されていない場合は、その方法では設定できないことを示しています。

XCF

```
MODEL ENTNAME  
  NET s PWR_MODE=LOW;  
  NET s KEEP;  
END;
```

RLOC (RLOC)

詳細は、次を参照してください。

[RLOC \(RLOC\)](#)

サポートされるサードパーティ制約

この章では、次のセクションに分けて XST でサポートされるサードパーティ制約について説明します。

- ・ サードパーティの制約と同等の XST 制約
- ・ サードパーティ制約の構文例

サードパーティの制約と同等の XST 制約

このセクションでは、制約とその制約と同等の XST 制約を示します。各制約の機能などについては、該当するベンダーのマニュアルを参照してください。

サードパーティ制約の中には、XST で自動的にサポートされるものがあります。次の表で「あり」になっている制約は、完全にサポートされています。部分的にのみサポートされる場合は、その詳細が次の表の自動識別の列に記述されています。

次の規則が適用されます。

- ・ VHDL では標準的な属性構文が使用されるので、HDL コードを変更する必要はありません。
- ・ サードパーティのメタコメント構文を含む Verilog の場合、そのメタコメント構文は XST の命名規則に従うように変更する必要があります。制約名とその値は、サードパーティツールで表示されているとおりに使用できます。
- ・ Verilog 2001 属性の場合、HDL コードを変更する必要はありません。制約は自動的に VHDL 属性構文に変換されます。

サードパーティの制約と同等の XST 制約

名前	ベンダー	同等の XST 制約	自動認識	HDL
black_box	Synopsys	ボックス タイプ	なし	VHDL、Verilog
black_box_pad_pin	Synopsys	なし	なし	なし
black_box_tri_pins	Synopsys	なし	なし	なし
cell_list	Synopsys	なし	なし	なし
clock_list	Synopsys	なし	なし	なし
Enum	Synopsys	なし	なし	なし
full_case	Synopsys	フル ケース	なし	Verilog
ispad	Synopsys	なし	なし	なし
map_to_module	Synopsys	なし	なし	なし

名前	ベンダー	同等の XST 制約	自動認識	HDL
net_name	Synopsys	なし	なし	なし
parallel_case	Synopsys	パラレル ケース	なし	Verilog
return_port_name	Synopsys	なし	なし	なし
resource_sharing directives	Synopsys	リソース共有	なし	VHDL、Verilog
set_dont_touch_network	Synopsys	必要なし	なし	なし
set_dont_touch	Synopsys	必要なし	なし	なし
set_dont_use_cel_name	Synopsys	必要なし	なし	なし
set_prefer	Synopsys	なし	なし	なし
state_vector	Synopsys	なし	なし	なし
syn_allow_retiming	Synopsys	レジスタ自動調整	なし	VHDL、Verilog
syn_black_box	Synopsys	ボックス タイプ	○	VHDL、Verilog
syn_direct_enable	Synopsys	なし	なし	なし
syn_edif_bit_format	Synopsys	なし	なし	なし
syn_edif_scalar_format	Synopsys	なし	なし	なし
syn_encoding	Synopsys	FSM エンコード方法の指定	あり (safe は自動認識されません。XST でセーフインプリメンテーションを使用してこのモードを有効にしてください)	VHDL、Verilog
syn_enum_encoding	Synopsys	列挙型エンコード手法	なし	VHDL
syn_hier	Synopsys	階層の維持	あり syn_hier = hardrecognized askeep_hierarchy = soft syn_hier = removerecognized askeep_hierarchy = no (XST では、syn_hier の hard と remove 値しか自動認識されません)	VHDL、Verilog
syn_isclock	Synopsys	なし	なし	なし
syn_keep	Synopsys	キープ	あり (最終ネットリストに指定したネットは保持されますが、KEEP 制約は付けません)	VHDL、Verilog
syn_maxfan	Synopsys	最大ファンアウト	あり	VHDL、Verilog

名前	ベンダー	同等の XST 制約	自動認識	HDL
syn_netlist_hierarchy	Synopsys	ネットリスト階層	なし	VHDL、Verilog
syn_noarrayports	Synopsys	なし	なし	なし
syn_noclockbuf	Synopsys	バッファタイプ	あり	VHDL、Verilog
syn_noprune	Synopsys	インスタンス化されたプリミティブの最適化	あり	VHDL、Verilog
syn_pipeline	Synopsys	レジスタ自動調整	なし	VHDL、Verilog
syn_preserve	Synopsys	等価レジスタの削除	あり	VHDL、Verilog
syn_ramstyle	Synopsys	ram_extract および ram_style	あり 指定してもしなくても no_rw_check モードでインプリメントします。 area 値は無視されます。	VHDL、Verilog
syn_reference_clock	Synopsys	なし	なし	なし
syn_replicate	Synopsys	レジスタの複製	あり	VHDL、Verilog
syn_romstyle	Synopsys	rom_extract および rom_style	あり	VHDL、Verilog
syn_sharing	Synopsys	なし	なし	VHDL、Verilog
syn_state_machine	Synopsys	FSM 自動抽出	あり	VHDL、Verilog
syn_tco <n>	Synopsys	なし	なし	なし
syn_tpd <n>	Synopsys	なし	なし	なし
syn_tristate	Synopsys	なし	なし	なし
syn_tristatetomux	Synopsys	なし	なし	なし
syn_tsu <n>	Synopsys	なし	なし	なし
syn_useenables	Synopsys	なし	なし	なし
syn_useioff	Synopsys	I/O レジスタの IOB 内へのバック	なし	VHDL、Verilog
synthesis translate_off synthesis translate_on	Synopsys	変換なし 変換あり	あり	VHDL、Verilog
xc_alias	Synopsys	なし	なし	なし
xc_clockbuftype	Synopsys	バッファタイプ	なし	VHDL、Verilog
xc_fast	Synopsys	FAST	なし	VHDL、Verilog
xc_fast_auto	Synopsys	FAST	なし	VHDL、Verilog
xc_global_buffers	Synopsys	BUFG (XST)	なし	VHDL、Verilog
xc_ioff	Synopsys	I/O レジスタの IOB 内へのバック	なし	VHDL、Verilog
xc_isgsr	Synopsys	なし	なし	なし

名前	ベンダー	同等の XST 制約	自動認識	HDL
xc_loc	Synopsys	LOC	あり	VHDL、Verilog
xc_map	Synopsys	LUT_MAP	あり (XST でサポートされる値は lut のみです)	VHDL、Verilog
xc_ncf_auto_relax	Synopsys	なし	なし	なし
xc_nodelay	Synopsys	NODELAY	なし	VHDL、Verilog
xc_padtype	Synopsys	I/O 規格	なし	VHDL、Verilog
xc_props	Synopsys	なし	なし	なし
xc_pullup	Synopsys	PULLUP	なし	VHDL、Verilog
xc_rloc	Synopsys	RLOC	あり	VHDL、Verilog
xc_fast	Synopsys	FAST	なし	VHDL、Verilog
xc_slow	Synopsys	なし	なし	なし
xc_uset	Synopsys	U_SET	あり	VHDL、Verilog

サードパーティ制約の構文例

次のサードパーティ制約の構文例は、次を実行する唯一の方法です。

- ・ HDL デザインの信号/ネットを保持、および
- ・ 合成中に信号またはネットでの最適化の防止

Verilog 構文例

```
module testkeep (in1, in2, out1);
  input in1;
  input in2;
  output out1;
  (* keep = "yes" *) wire aux1;
  (* keep = "yes" *) wire aux2;
  assign aux1 = in1;
  assign aux2 = in2;
  assign out1 = aux1 & aux2;
endmodule
```

XCF の構文例

キープ (KEEP) 制約は、別の合成制約ファイルからも指定できます。

```
BEGIN MODEL testkeep
  NET aux1 KEEP=true;
END;
```

VHDL 言語のサポート

この章は、次の内容が含まれます。

- ・ XST で VHSIC Hardware Description Language (VHDL) がどのようにサポートされるか説明します。
- ・ VHDL でサポートされる構文および合成オプションに関する詳細を示します。

次のセクションが含まれています。

- ・ [VHDL の論理記述](#)
- ・ [VHDL の IEEE サポート](#)
- ・ [VHDL のファイル タイプ サポート](#)
- ・ [VHDL の書き込み関数を使用したデバッグ](#)
- ・ [VHDL のデータ型](#)
- ・ [VHDL のレコード型](#)
- ・ [VHDL の初期値](#)
- ・ [VHDL のオブジェクト](#)
- ・ [VHDL の演算子](#)
- ・ [VHDL のエンティティとアーキテクチャの記述](#)
- ・ [VHDL の組み合わせ回路](#)
- ・ [VHDL の順序回路](#)
- ・ [VHDL の関数とプロシージャ](#)
- ・ [VHDL のアサート文](#)
- ・ [パッケージ文を使用して定義した VHDL モデル](#)
- ・ [サポートされる VHDL 構文](#)
- ・ [VHDL の予約語](#)

VHDL の論理記述

VHDL は、幅広い言語構文を持ち、複雑なロジックを簡潔に表現できる次のようなハードウェア記述言語です。

- ・ 次のシステムの構造を記述できます。
 - － システムをサブシステムに分割する方法
 - － 分割されたサブシステムを相互接続する方法
- ・ 使い慣れたプログラミング言語形式でシステムの機能を指定できます。
- ・ インプリメンテーションや製造の前にシステム デザインをシミュレーションできます。ハードウェア プロトタイプに時間や費用を費やすことなく、正確性を検証できます。
- ・ 抽象記述を合成して、デバイス特定の詳細なデザインを簡単に作成できる方法を提供します。この機能により、デザインを効率的に設計でき、タイムトゥ マーケットを短縮できます。

詳細は、次を参照してください。

- ・ IEEE の『VHDL Language Reference Manual』
- ・ [デザイン制約](#)
- ・ [VHDL 属性の構文](#)

VHDL の IEEE サポート

このセクションでは、VHDL の IEEE サポートについて説明します。

- ・ [サポートされる VHDL の IEEE 規格](#)
- ・ [VHDL の IEEE の競合](#)
- ・ [VHDL で LRM に準拠しない構文](#)

サポートされる VHDL の IEEE 規格

XST では、次のVHDL IEEE 規格がサポートされます。

- ・ **Std 1076-1987**
- ・ **Std 1076-1993**
- ・ **Std 1076-2006**

メモ： Std 1076-2006 は、一部のみインプリメントされています。XST では次の表に示すように、Std 1076-2006 のインスタンスエーションがサポートされます。

フォーマル ポート	関連するアクチュアル
buffer	out
out	buffer

VHDL の IEEE の競合

VHDL IEEE の Std 1076-1987 は、Std 1076-1993 と競合しない限り、使用できます。競合がある場合、Std 1076-1993 のビヘイビアが Std 1076-1987 のビヘイビアを上書きします。

上書きされるのは次の場合です。

- ・ Std 1076-1993 では、エラーになる場合の構文が必要
- ・ Std 1076-1987 ではそのまま問題なし

XST でエラーではなく、警告メッセージが表示されます。エラーの場合は、解析が止まってしまいます。

VHDL の IEEE の競合例

次は、VHDL の IEEE 競合の具体的な例です。

- ・ VHDL IEEE 規格 1076-1993 では、関数の宣言中に `impure` キーワードを使用するために `impure` 関数が必要
- ・ Std 1076-1987 にそのような条件がない

この場合、XST は次のように動作します。

- ・ Std 1076-1987 の VHDL コードを受諾
- ・ Std 1076-1993 ビヘイビアに関する警告メッセージを表示

VHDL で LRM に準拠しない構文

XST では、LRM に準拠しない構文も一部サポートされます。サポートされるのは、次の場合です。

- ・ 制約が合成またはシミュレーション用のサードパーティ ツールのほとんどでサポートされる場合
- ・ コードに使用する言語に限界があり、結果や問題検出に影響のない場合

たとえば、LRM ではフォーマル ポートが `buffer` で有効なポートが `out` の場合（またはその逆）、インスタンス化ができません。

VHDL のファイル タイプ サポート

XST では、次の表に示すように、制限付きで VHDL のファイル読み出しおよび書き込みがサポートされています。

機能	使用例
ファイルの読み出し	外部ファイルからの RAM の初期化
ファイルの書き込み	<ul style="list-style-type: none"> ・ デバッグ プロセス ・ 特定の定数またはジェネリック値の外部ファイルへの書き込み

詳細は、次を参照してください。

RAM の初期化のコード例

次の表の `read` 関数のいずれかを使用してください。これらの `read` 関数は次のパッケージでサポートされます。

- ・ `standard`
- ・ `std.textio`
- ・ `ieee.std_logic_textio`

関数	パッケージ
file (type text only)	standard
access (type line only)	standard
file_open (file, name, open_kind)	standard
file_close (file)	standard
endfile (file)	standard
text	std.textio
line	std.textio
width	std.textio
readline (text, line)	std.textio
readline (line, bit, boolean)	std.textio
read (line, bit)	std.textio
readline (line, bit_vector, boolean)	std.textio
read (line, bit_vector)	std.textio
read (line, boolean, boolean)	std.textio
read (line, boolean)	std.textio
read (line, character, boolean)	std.textio
read (line, character)	std.textio
read (line, string, boolean)	std.textio
read (line, string)	std.textio
write (file, line)	std.textio
write (line, bit, boolean)	std.textio
write (line, bit)	std.textio
write (line, bit_vector, boolean)	std.textio
write (line, bit_vector)	std.textio
write (line, boolean, boolean)	std.textio
write (line, boolean)	std.textio
write (line, character, boolean)	std.textio
write (line, character)	std.textio
write (line, integer, boolean)	std.textio
write (line, integer)	std.textio
write (line, string, boolean)	std.textio
write (line, string)	std.textio
read (line, std_ulogic, boolean)	ieee.std_logic_textio
read (line, std_ulogic)	ieee.std_logic_textio
read (line, std_ulogic_vector), boolean	ieee.std_logic_textio

関数	パッケージ
read (line, std_ulogic_vector)	ieee.std_logic_textio
read (line, std_logic_vector, boolean)	ieee.std_logic_textio
read (line, std_logic_vector)	ieee.std_logic_textio
write (line, std_ulogic, boolean)	ieee.std_logic_textio
write (line, std_ulogic)	ieee.std_logic_textio
write (line, std_ulogic_vector, boolean)	ieee.std_logic_textio
write (line, std_ulogic_vector)	ieee.std_logic_textio
write (line, std_logic_vector, boolean)	ieee.std_logic_textio
write (line, std_logic_vector)	ieee.std_logic_textio
hread	ieee.std_logic_textio

VHDL の書き込み関数を使用したデバッグ

このセクションでは、VHDL で書き込み関数を使用したデバッグ方法について説明します。

- ・ [デバッグ規則](#)
- ・ [Endfile 関数の使用](#)

デバッグ規則

VHDL で書き込み関数を使用してデバッグするには、次のような規則に従う必要があります。

- ・ **std_logic** read の操作で使用できる文字は 0 および 1 のみです。X や Z のようなそれ以外の値は使用できません。0 と 1 以外の文字を含むファイルは XST で拒否されます。ただし、ファイル内にスペース文字が検出されると、その文字は無視されます。
- ・ 別のディレクトリでも同じファイル名は使用しないでください。
- ・ 次の例にあるように、プロシージャを read で読み出す場合に条件文を使用すると、シミュレーションでエラーが発生します。

```
if SEL = '1' then
    read (MY_LINE, A(3 downto 0));
else
    read (MY_LINE, A(1 downto 0));
end if;
```

endfile 関数の使用

endfile 関数を使用して次のようなスタイルで記述すると、XST でデザインが拒否されます。

```
while (not endfile (MY_FILE)) loop
    readline (MY_FILE, MY_LINE);
    read (MY_LINE, MY_DATA);
end loop;
```

この場合、XST で次のようなエラー メッセージが表示されます。

```
Line <MY_LINE> has not enough elements for target <MY_DATA>.
```

この問題を回避するには、次のように exit when endfile (MY_FILE); を while ループ内に追加します。

```
while (not endfile (MY_FILE)) loop
    readline (MY_FILE, MY_LINE);
    exit when endfile (MY_FILE);
    read (MY_LINE, MY_DATA);
end loop;
```

コード例

```
--
-- Print 2 constants to the output file
--

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_arith.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use STD.TEXTIO.all;
use IEEE.STD_LOGIC_TEXTIO.all;

entity file_support_1 is
    generic (data_width: integer:= 4);
    port( clk, sel: in std_logic;
          din: in std_logic_vector (data_width - 1 downto 0);
          dout: out std_logic_vector (data_width - 1 downto 0));
end file_support_1;

architecture Behavioral of file_support_1 is
    file results : text is out "test.dat";
    constant base_const: std_logic_vector(data_width - 1 downto 0) := conv_std_logic_vector(3,data_width);
    constant new_const: std_logic_vector(data_width - 1 downto 0) := base_const + "1000";
begin

    process(clk)
        variable txtline : LINE;
    begin
        write(txtline,string'("-----"));
        writeline(results, txtline);
        write(txtline,string'("Base Const: "));
        write(txtline,base_const);
        writeline(results, txtline);

        write(txtline,string'("New Const: "));
        write(txtline,new_const);
        writeline(results, txtline);
        write(txtline,string'("-----"));
        writeline(results, txtline);
    end process;
end;
```

```
    if (clk'event and clk='1') then
        if (sel = '1') then
            dout <= new_const;
        else
            dout <= din;
        end if;
    end if;
end process;

end Behavioral;
```

VHDL のデータ型

このセクションでは、次について説明します。

- ・ [使用できる VHDL のデータ型](#)
- ・ [オーバーロード データ型](#)
- ・ [多次元配列型](#)

使用できる VHDL のデータ型

XST では、次の VHDL データ型を使用できます。

- ・ [列挙型](#)
- ・ [ユーザー定義の列挙型](#)
- ・ [ビット ベクタ型](#)
- ・ [整数型](#)
- ・ [定義済み型](#)
- ・ [STD_LOGIC_1164 IEEE 型](#)

列挙型

データ型	値	説明	コメント
BIT	0, 1	--	--
BOOLEAN	false, true	--	--
REAL	\$-. から \$+.	--	--
STD_LOGIC	U	初期化なし	XSTでは使用できません
	X	不明	ドント ケアとして処理されます
	0	low	L と同じように処理されます
	1	high	H と同じように処理されます
	Z	ハイ インピーダンス	ハイ インピーダンスとして処理されます
	W	ウィークな不明状態	XSTでは使用できません
	L	ウィーク Low	0 と同じように処理されます
	H	ウィーク High	1 と同じように処理されます
	-	ドントケア	ドント ケアとして処理されます

ユーザー定義の列挙型

```
type COLOR is (RED, GREEN, YELLOW);
```

ビット ベクタ型

- ・ [BIT_VECTOR](#)
- ・ [STD_LOGIC_VECTOR](#)

整数型

INTEGER

定義済み型

- ・ BIT
- ・ BOOLEAN
- ・ BIT_VECTOR
- ・ INTEGER
- ・ REAL

STD_LOGIC_1164 IEEE 型

STD_LOGIC_1164 IEEE パッケージでは、次のデータ型が定義されています。

- ・ STD_LOGIC
- ・ STD_LOGIC_VECTOR

STD_LOGIC_1164 パッケージは IEEE ライブラリに含まれています。これらのデータ型を使用するには、次のように宣言する必要があります。

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
```

オーバーロード データ型

次のデータ型はオーバーロード可能です。

- ・ [オーバーロード列挙型](#)
- ・ [オーバーロードビットベクタ型](#)
- ・ [オーバーロード整数型](#)
- ・ [オーバーロード STD_LOGIC_1164 IEEE 型](#)
- ・ [オーバーロード STD_LOGIC_ARITH IEEE 型](#)

オーバーロード列挙型

- ・ STD_ULOGIC

STD_LOGIC データ型として 9 種類の論理値を持ちますが、定義済みの resolution 関数は含まれません。

- ・ X01

X、0、1 の値を含む STD_ULOGIC のサブタイプです。

- ・ X01Z

X、0、1、Z の値を含む STD_ULOGIC のサブタイプです。

- ・ UX01

U、X、0、1 の値を含む STD_ULOGIC のサブタイプです。

- ・ UX01Z

U、X、0、1、Z の値を含む STD_ULOGIC のサブタイプです。

オーバーロード ビット ベクタ型

- ・ STD_ULOGIC_VECTOR
- ・ UNSIGNED
- ・ SIGNED

制約が設定されていないもの（長さが指定されていないもの）は使用できません。

オーバーロード 整数型

- ・ NATURAL
- ・ POSITIVE

ユーザー定義範囲内の整数。次に例を示します。

type MSB is range 8 to 15;

これは、次の範囲の整数を含めています。

- ・ 7 より大きい
- ・ 16 より小さい

NATURAL および POSITIVE は、定義済みの VHDL データ型です。

オーバーロード STD_LOGIC_1164 IEEE 型

IEEE の STD_LOGIC_1164 パッケージでは、次のデータ型が定義されています。

- ・ STD_ULOGIC (およびサブタイプ X01、X01Z、UX01、UX01Z)
- ・ STD_LOGIC
- ・ STD_ULOGIC_VECTOR
- ・ STD_LOGIC_VECTOR

このパッケージは、IEEE ライブラリにコンパイルされています。これらのデータ型を使用するには、次のように宣言する必要があります。

```
library IEEE;  
use IEEE.STD_LOGIC_1164.all;
```

オーバーロード STD_LOGIC_ARITH IEEE 型

UNSIGNED および SIGNED (STD_LOGIC の配列として定義) は、STD_LOGIC_ARITH IEEE パッケージで宣言されています。

このパッケージは、IEEE ライブラリにコンパイルされています。これらのデータ型を使用するには、次の 2 行を VHDL に追加する必要があります。

```
library IEEE;  
use IEEE.STD_LOGIC_ARITH.all;
```

多次元配列型

XST では、3 次元までの多次元配列型がサポートされます。BRAM は推論されません。配列は、次のいずれかになります。

- ・ 信号
- ・ 定数
- ・ VHDL 変数

配列を使用すると、代入や数値演算を行うことができます。多次元配列を関数に渡し、インスタンスーションで使用することもできます。

コード例 1

配列には、すべての次元に制約を指定する必要があります。次がその例です。

```
subtype WORD8 is STD_LOGIC_VECTOR (7 downto 0);
type TAB12 is array (11 downto 0) of WORD8;
type TAB03 is array (2 downto 0) of TAB12;
```

コード例 2

また、次のように配列をマトリックスとして宣言できます。

```
subtype TAB13 is array (7 downto 0, 4 downto 0) of
STD_LOGIC_VECTOR (8 downto 0);
```

次のように宣言したとします。

```
subtype WORD8 is STD_LOGIC_VECTOR (7 downto 0);
type TAB05 is array (4 downto 0) of WORD8;
type TAB03 is array (2 downto 0) of TAB05;

signal WORD_A : WORD8;
signal TAB_A, TAB_B : TAB05;
signal TAB_C, TAB_D : TAB03;
constant CNST_A : TAB03 := (
  ("00000000", "01000001", "01000010", "10000011", "00001100"),

  ("00100000", "00100001", "00101010", "10100011", "00101100"),

  ("01000010", "01000010", "01000100", "01000111", "01000100"));
```

この場合、次のように指定できます。

- ・ 多次元配列の信号または変数

```
TAB_A <= TAB_B; TAB_C <= TAB_D; TAB_C <= CNST_A;
```

- ・ 1 配列のインデックス

```
TAB_A (5) <= WORD_A; TAB_C (1) <= TAB_A;
```

- ・ 最大の次元数のインデックス

```
TAB_A (5) (0) <= '1'; TAB_C (2) (5) (0) <= '0'
```

- ・ 最初の配列のスライス

```
TAB_A (4 downto 1) <= TAB_B (3 downto 0);
```

- ・ 高次元配列のインデックスと低次元配列のスライス

```
TAB_C (2) (5) (3 downto 0) <= TAB_B (3) (4 downto 1); TAB_D  
(0) (4) (2 downto 0) <= CNST_A (5 downto 3)
```

コード例 3

次の宣言文を追加します。

```
subtype MATRIX15 is array(4 downto 0, 2 downto 0) of  
STD_LOGIC_VECTOR (7 downto 0); signal MATRIX_A : MATRIX15;
```

この場合、次のように指定できます。

- ・ 多次元配列の信号または変数

```
MATRIXA <= CNST_A;
```

- ・ 1 行の配列のインデックス

```
MATRIXA (5) <= TAB_A;
```

- ・ 最大の次元数のインデックス

```
MATRIXA (5,0) (0) <= '1';
```

インデックスは、変数である可能性があります。

VHDL のレコード型

XST では、次のコード例のような VHDL レコード型がサポートされます。

- ・ レコード型には、別のレコード型を含めることができます。
- ・ 定数をレコード型にできます。
- ・ レコード型に属性を含むことはできません。
- ・ レコード信号への集合代入文がサポートされます。

コード例

```
type REC1 is record
    field1: std_logic;
    field2: std_logic_vector (3 downto 0)
end record;
```

VHDL の初期値

このセクションでは、次について説明します。

- ・ [レジスタの初期化](#)
- ・ [VHDL のローカルリセット/グローバルリセット](#)
- ・ [VHDL のメモリ エLEMENT のデフォルト初期値の概要](#)

レジスタの初期化

VHDL では、レジスタを宣言する際に初期化できます。

初期値は、次の規則に従って設定する必要があります。

- ・ 定数値を指定する必要があります。
- ・ 以前の初期値に依存できません。
- ・ 関数またはタスク呼び出しは使用できません。
- ・ レジスタに伝搬するパラメータ値にできます。

コード例 1

宣言部でレジスタの初期値を指定した場合、XST では次の時点でこの値を設定します。

- ・ グローバルリセットのレジスタ出力、または
- ・ 電源投入時

代入される値は、次のようになります。

- ・ レジスタの INIT 属性として NGC ファイルに渡されます。
- ・ ローカルリセットには依存しません。

```
signal arb_onebit : std_logic := '0';
signal arb_priority : std_logic_vector(3 downto 0) := "1011";
```

コード例 2

また、ビヘイビア記述の VHDL コードを使用して、レジスタにセット/リセットの値を指定できます。レジスタのリセット ラインが最適な値になったときにレジスタに値を代入するには、次の例のように記述します。

```
process (clk, rst)
begin
    if rst='1' then
        arb_onebit <= '0';
    end if;
end process;
```

ビヘイビア コードで変数の初期値を設定すると、出力がローカル リセットで制御可能なフリップフロップとしてデザインにインプリメントされ、NGC ファイルに FDP または FDC フリップフロップとして記述されます。

VHDL のローカル リセット/グローバル リセット

ローカル リセットは、グローバル リセットとは関係なく動作します。ローカル リセットで制御できるレジスタでは、グローバル リセット時 (または電源投入時) とローカル リセット時で異なる値を指定できます。次のコード例では、レジスタ arb_onebit はグローバル リセット時には 0、ローカル リセット時 (rst) には 1 になるよう設定しています。

ローカル リセット/グローバル リセットの VHDL コード例

電源投入時にはレジスタの出力が 1 に初期化されるよう設定されていますが、この値はローカル リセットの値によって変わるので、ローカル セット/リセットがアクティブになると 0 に初期化されます。

```
entity top is
    Port (
        clk, rst : in std_logic;
        a_in : in std_logic;
        dout : out std_logic);
end top;
architecture Behavioral of top is
    signal arb_onebit : std_logic := '1';

begin
    process (clk, rst)
    begin
        if rst='1' then
            arb_onebit <= '0';
        elsif (clk'event and clk='1') then
            arb_onebit <= a_in;
        end if;
    end process;

    dout <= arb_onebit;
end Behavioral;
```

VHDL のメモリ エLEMENTのデフォルト初期値の概要

ザイリンクス FPGA デバイスに含まれるすべてのメモリエLEMENTは、既知のステートになる必要があるので、場合によっては IEEE 規格の初期値に従わない場合があります。たとえば、「ローカルリセット/グローバルリセットの VHDL コード例」の `arb_onebit` 信号が 1 に初期化されない場合、XST では初期値としてデフォルトの 0 を割り当てます。この場合、XST は IEEE 規格 (`std_logic` のデフォルト値は U) に従いません。このような初期化プロセスは、レジスタおよび RAM に対して行われます。

XST は信号の値を初期化する際に、できるだけ IEEE VHDL 規格に従います。初期値が VHDL コードに含まれていない場合は、次の表の XST 列のようなデフォルト値が使用されます。

タイプ	IEEE	XST
bit	'0'	'0'
std_logic	'U'	'0'
bit_vector (3 downto 0)	0000	0000
std_logic_vector (3 downto 0)	0000	0000
integer (unconstrained)	integer'left	integer'left
integer range 7 downto 0	integer'left = 7	integer'left = 7 (コードは 111)
integer range 0 to 7	integer'left = 0	integer'left = 0 (コードは 000)
Boolean	FALSE	FALSE (コードは 0)
enum (S0,S1,S2,S3)	type'left = S0	type'left = S0 (コードは 000)

未接続の出力ポートは、デフォルトで VHDL 初期値の XST 列に示されている値になります。出力ポートに初期状態がある場合は、未接続の出力ポートが定義された初期状態になるように接続されます。IEEE VHDL 規格では、入力ポートを未接続の状態にすることはできません。このため、入力ポートが未接続の場合は必ずエラーメッセージが表示されます。入力ポートに `open` キーワードが付けられていても、エラーメッセージが表示されます。

VHDL のオブジェクト

このセクションでは、次について説明します。

- ・ [VHDL の信号](#)
- ・ [VHDL の変数](#)
- ・ [VHDL の定数](#)

VHDL の信号

VHDL の信号には、次のような特徴があります。

- ・ アーキテクチャ宣言部分で宣言できます。
- ・ アーキテクチャ文内のどこでも使用できます。
- ・ ブロック内で宣言できます。
- ・ そのブロック内で使用できます。
- ・ 代入演算子「<=」を使用して代入します。

コード例

```
signal sig1 : std_logic; sig1 <= '1';
```

VHDL の変数

VHDL の変数には、次の特徴があります。

- ・ process または subprogram 文で宣言します。
- ・ その process または subprogram 文内で使用されます。
- ・ 変数は次の代入演算子を使用して代入します。

:=

コード例

```
variable var1 : std_logic_vector (7 downto 0); var1 := "01010011";
```

VHDL の定数

VHDL の定数：

- ・ 宣言部分で制限できます。
- ・ その領域内で使用できます。
- ・ 宣言後、その値は変更できません。

コード例

```
signal sig1 : std_logic_vector (5 downto 0);  
constant init0 : std_logic_vector (5 downto 0) := "010111";  
sig1 <= init0;
```

VHDL の演算子

サポートされる演算子は、「[VHDL の論理式](#)」にリストされています。このセクションでは、各シフト演算子のコード例を示します。

Shift Left Logical の VHDL コード例

```
sll (Shift Left Logical)  sig1 <= A(4 downto 0) sll 2
```

これは、次と同じです。

```
sig1 <= A(2 downto 0) & "00";
```

Shift Right Logical の VHDL コード例

```
srl (Shift Right Logical) sig1 <= A(4 downto 0) srl 2
```

これは、次と同じです。

```
sig1 <= "00" & A(4 downto 2);
```

Shift Left Arithmetic の VHDL コード例

```
sla (Shift Left Arithmetic)  sig1 <= A(4 downto 0) sla 2
```

これは、次と同じです。

```
sig1 <= A(2 downto 0) & A(0) & A(0);
```

Shift Right Arithmetic の VHDL コード例

```
sra (Shift Right Arithmetic)  sig1 <= A(4 downto 0) sra 2
```

これは、次と同じです。

```
sig1 <= A(4) & A(4) & A(4 downto 2);
```

Rotate Left の VHDL コード例

```
rol (Rotate Left) sig1 <= A(4 downto 0) rol 2
```

これは、次と同じです。

```
sig1 <= A(2 downto 0) & A(4 downto 3);
```

Rotate Right の VHDL コード例

```
ror (Rotate Right) A(4 downto 0) ror 2
```

これは、次と同じです。

```
sig1 <= A(1 downto 0) & A(4 downto 2);
```

VHDL のエンティティとアーキテクチャの記述

VHDL のエンティティとアーキテクチャの記述には、次が含まれます。

- ・ [回路記述](#)
- ・ [エンティティ宣言](#)
- ・ [アーキテクチャ宣言](#)
- ・ [コンポーネントのインスタンス化](#)
- ・ [再帰的コンポーネント インスタンス化](#)
- ・ [コンポーネント コンフィギュレーション](#)
- ・ [ジェネリック パラメータ宣言](#)
- ・ [ジェネリックと属性の競合](#)

VHDL の回路記述

VHDL の回路記述は、次の 2 つの部分で構成されています。

- ・ インターフェイス (I/O ポートの定義)
- ・ 本体部分

VHDL では、それぞれ次に相当します。

- ・ エンティティはインターフェイス
- ・ アーキテクチャはビヘイビア

VHDL のエンティティ宣言

回路の I/O ポートはエンティティで宣言します。各ポートには、次が指定されます。

- ・ 名前
- ・ モード
 - **in**
 - **out**
 - **inout**
 - **buffer**
- ・ タイプ (「[エンティティとアーキテクチャ宣言のコード例](#)」の次のポート):
 - **A**
 - **B**
 - **C**
 - **D**
 - **E**

ポートには、1 次元配列型のみを使用できます。

VHDL のアーキテクチャ宣言

内部信号はアーキテクチャ内で宣言できます。各内部信号には、次が指定されます。

- ・ 名前
- ・ 型

次のコード例では信号 **T**

コード例

```
Library IEEE;
use IEEE.std_logic_1164.all;
entity EXAMPLE is
    port (
        A,B,C : in std_logic;
        D,E : out std_logic );
end EXAMPLE;

architecture ARCH1 of EXAMPLE is
    signal T : std_logic;
begin
    ...
end ARCH1;
```

VHDL のコンポーネント インスタンス化

構造記述では、複数のブロックを組み合わせ、デザインを階層構造にできます。

概念	説明
コンポーネント	基本ブロックの構築
ポート	コンポーネントの I/O コネクタ
信号	コンポーネント間のワイヤに対応

VHDL の場合、コンポーネントはデザイン エンティティによって表されます。デザイン エンティティは、次の表に示す概念で構成されます。

概念	表示	内容
エンティティ宣言	外部	コンポーネント ポートも含めた外観
アーキテクチャ本体	内部	コンポーネントのビヘイビアや構造

コンポーネント間の接続は、コンポーネント インスタンス化文で定義されます。この文では、別のコンポーネントのアーキテクチャ文内で使用されるコンポーネントのインスタンスを指定します。コンポーネント インスタンス化文はそれぞれ識別子で区別されます。

コンポーネント インスタンス化文では、ローカル コンポーネント宣言部分で宣言されたコンポーネントの名前が指定されるほか、関係リスト (予約語ポート マップの後のかっこで囲まれたリスト) が含まれます。この関係リストでは、信号やポートをどのコンポーネント宣言のローカル ポートと接続するかが指定されます。

XST では、コンポーネント宣言で制約が設定されていないベクタがサポートされます。

コード例

次は、4 つの NAND2 コンポーネントから構成される半加算器の構造記述例を示しています。

```
entity NAND2 is
  port (
    A,B : in BIT;
    Y : out BIT );
end NAND2;

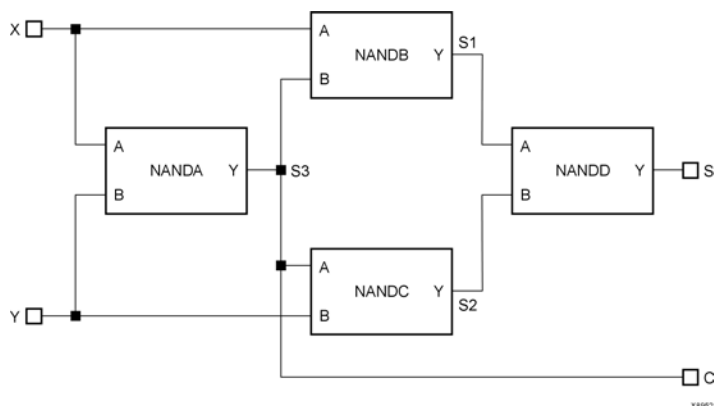
architecture ARCH1 of NAND2 is
begin
  Y <= A nand B;
end ARCH1;

entity HALFADDER is
  port (
    X,Y : in BIT;
    C,S : out BIT );
end HALFADDER;

architecture ARCH1 of HALFADDER is
  component NAND2
  port (
    A,B : in BIT;
    Y : out BIT );
  end component;

  for all : NAND2 use entity work.NAND2(ARCH1);
  signal S1, S2, S3 : BIT;
begin
  NANDA : NAND2 port map (X,Y,S3);
  NANDB : NAND2 port map (X,S3,S1);
  NANDC : NAND2 port map (S3,Y,S2);
  NANDD : NAND2 port map (S1,S2,S);
  C <= S3;
end ARCH1;
```

合成済みの最上位レベルのネットリストの図



VHDL の再帰的なコンポーネント インスタンスーション文

XST では、再帰的なコンポーネント インスタンスーション文がサポートされます。ただし、直接的なインスタンスーションは再帰的な文では使用できません。再帰呼び出しが永久に実行されるのを防ぐために、繰り返す回数は 64 (デフォルト) に制限されています。回数を変更するには、`-recursion_iteration_limit` オプションを使用します。

再帰的なコンポーネント インスタンスーション文を使用した 4 ビット シフト レジスタのコード例

```
library ieee;
use ieee.std_logic_1164.all;
library unisim;
use unisim.vcomponents.all;

entity single_stage is
  generic (sh_st: integer:=4);
  port (
    CLK : in std_logic;
    DI  : in std_logic;
    DO  : out std_logic );
end entity single_stage;

architecture recursive of single_stage is
  component single_stage
    generic (sh_st: integer);
    port (
      CLK : in std_logic;
      DI  : in std_logic;
      DO  : out std_logic );
  end component;

  signal tmp : std_logic;

begin
  GEN_FD_LAST: if sh_st=1 generate
    inst_fd: FD port map (D=>DI, C=>CLK, Q=>DO);
  end generate;
  GEN_FD_INTERM: if sh_st /= 1 generate
    inst_fd: FD port map (D=>DI, C=>CLK, Q=>tmp);
    inst_sstage: single_stage generic map (sh_st => sh_st-1)
      port map (DI=>tmp, CLK=>CLK, DO=>DO);
  end generate;
end recursive;
```

VHDL のコンポーネント コンフィギュレーション文

エンティティ/アーキテクチャのペアをコンポーネント インスタンス文に関連付けると、コンポーネントを適切なモデル (エンティティ/アーキテクチャのペア) とリンクできます。

XST では、アーキテクチャ宣言でのコンポーネント コンフィギュレーション文の使用がサポートされます。

```
for instantiation_list: component_name use LibName.entity_Name(Architecture_Name);
```

コード例

次のコード例は、コンポーネント インスタンス化文でのコンフィギュレーション節の使用を示しています。この例には、次の for all 文が含まれます。

```
for all : NAND2 use entity work.NAND2 (ARCHI);
```

この文は、すべての NAND2 コンポーネントでエンティティ NAND2 とアーキテクチャ ARCHI が使用されることを表します。

コンポーネント インスタンス文にコンフィギュレーション節がない場合、XST でコンポーネントが同じ名前のエンティティに関連付けられ、選択されたアーキテクチャが最後にコンパイルされたアーキテクチャに関連付けられます。エンティティまたはアーキテクチャがない場合、合成中にブラック ボックスが生成されます。

コマンドライン モードでは、デザインのコンポーネント インスタンス化をデザイン エンティティおよびアーキテクチャにリンクする専用のコンフィギュレーション宣言を使用することもできます。この場合、必須の run コマンドの [Top Module Name] (-top) オプションの値は、最上位レベルのエンティティ名ではなく、コンフィギュレーション名です。

VHDL のジェネリック パラメータ宣言

VHDL の [ジェネリック \(-generics\)](#) コマンドライン オプションを使用すると、最上位デザイン ブロックで定義されるジェネリック値を再定義できます。これにより、IP コアの生成やテストベンチング フローなどの HDL ソースを変更しなくてもデザイン コンフィギュレーションを簡単に修正できます。

ジェネリック パラメータは、エンティティ宣言部分で宣言できます。XST では、次を含めたあらゆるタイプのジェネリック パラメータがサポートされます。

- ・ 整数
- ・ ブール型
- ・ 文字列
- ・ 実数
- ・ `std_logic_vector`

ジェネリック パラメータの使用例としては、デザインのビット幅の設定があります。

コード例

ジェネリック ポートで回路を記述すると、同じコンポーネントを異なるジェネリック ポートの値で繰り返しインスタンス化できます。次のコード例を参照してください。

```
Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity addern is
  generic (width : integer := 8);
  port (
    A,B : in std_logic_vector (width-1 downto 0);
    Y    : out std_logic_vector (width-1 downto 0) );
```

```
end addern;

architecture bhv of addern is
begin
    Y <= A + B;
end bhv;

Library IEEE;
use IEEE.std_logic_1164.all;

entity top is
port (
    X, Y, Z : in std_logic_vector (12 downto 0);
    A, B : in std_logic_vector (4 downto 0);
    S :out std_logic_vector (16 downto 0) );
end top;

architecture bhv of top is
component addern
generic (width : integer := 8);
port (
    A,B : in std_logic_vector (width-1 downto 0);
    Y : out std_logic_vector (width-1 downto 0) );
end component;

for all : addern use entity work.addern(bhv);
signal C1 : std_logic_vector (12 downto 0);
signal C2, C3 : std_logic_vector (16 downto 0);
begin
    U1 : addern generic map (n=>13) port map (X,Y,C1);
    C2 <= C1 & A;
    C3 <= Z & B;
    U2 : addern generic map (n=>17) port map (C2,C3,S);
end bhv;
```

VHDL のジェネリックと属性の競合

ジェネリックおよび属性は、VHDL コードのインスタンスおよびコンポーネントの両方に適用でき、また属性は制約ファイルでも指定できるので、競合が発生する場合があります。競合を回避するために、XST では次の優先順位の規則が使用されます。

1. インスタンス (下位レベル) に指定されるものがコンポーネント (上位レベル) に指定されるものより優先されます。
2. ジェネリックと属性が同じインスタンスまたはコンポーネントに指定される場合、ジェネリックが優先され、XST で競合を示す警告メッセージが表示されます。
3. XCF (XST 制約ファイル) で指定される属性は、VHDL コードで指定される属性またはジェネリックよりも優先されます。

インスタンスに指定されている属性がコンポーネントに指定されているジェネリックを上書きすると、シミュレーション ツールでそのジェネリックが使用されなくなる可能性があります。この結果、シミュレーションの結果が合成結果に一致しない場合があります。

VHDL での優先順位

	インスタンスのジェネリック	コンポーネントのジェネリック
インスタンスの属性	ジェネリックを適用 (XST で警告メッセージが表示される)	属性を適用 (シミュレーションで不一致の可能性あり)
コンポーネントの属性	ジェネリックを適用	ジェネリックを適用 (XST で警告メッセージが表示される)
XCF に含まれる属性	属性を適用 (XST で警告メッセージが表示される)	属性を適用

ブロックの定義のセキュリティ属性は、ほかのどの属性またはジェネリックよりも優先されます。

VHDL の組み合わせ回路

XST では、次の VHDL 組み合わせ回路がサポートされます。

- ・ 同時処理信号代入文
- ・ generate 文
- ・ 組み合わせプロセス
- ・ の if - else 文
- ・ case 文
- ・ for - loop 文

VHDL の同時処理信号代入文

VHDL の組み合わせロジックは、同時処理信号代入文を使用して記述されます。組み合わせロジックは、アーキテクチャ本体で定義します。VHDL では、次の 3 種類の同時処理信号代入文があります。

- ・ シンプルな信号代入文
- ・ オン
- ・ 条件演算

同時処理信号代入文は必要に応じていくつでも記述できます。アーキテクチャ部分で定義した同時処理信号の順序とは関係ありません。

同時処理信号代入文には次の 2 側面があります。

- ・ 左側
- ・ 右側

右側にある信号に変化 (イベント) が発生して代入が変わると、その計算結果が左側に代入されます。

シンプルな信号代入文の VHDL コード例

```
T <= A and B;
```

選択信号代入文を使用したマルチプレクサの VHDL コード例

```
library IEEE;
use IEEE.std_logic_1164.all;

entity select_bhv is
  generic (width: integer := 8);
  port (
    a, b, c, d : in std_logic_vector (width-1 downto 0);
    selector : in std_logic_vector (1 downto 0);
    T : out std_logic_vector (width-1 downto 0) );
end select_bhv;

architecture bhv of select_bhv is
begin
  with selector select
    T <= a when "00",
         b when "01",
         c when "10",
         d when others;
end bhv;
```

条件的な信号代入文を使用したマルチプレクサの VHDL のコード例

```
entity when_ent is
  generic (width: integer := 8);
  port (
    a, b, c, d : in std_logic_vector (width-1 downto 0);
    selector : in std_logic_vector (1 downto 0);
    T : out std_logic_vector (width-1 downto 0) );
end when_ent;

architecture bhv of when_ent is
begin
  T <= a when selector = "00" else
        b when selector = "01" else
        c when selector = "10" else
        d;
end bhv;
```

VHDL のジェネレート文

反復構造は、VHDL の generate 文を使用して宣言します。たとえば、「for I in 1 to N generate」は、ビット スライス記述が N 回繰り返されることを意味します

for – generate 文を使用した 8 ビット加算器のコード例

次に、ビット スライス構造の宣言により 8 ビット加算器を記述する例を示します。

```
entity EXAMPLE is
  port (
    A,B  : in BIT_VECTOR (0 to 7);
    CIN  : in BIT;
    SUM  : out BIT_VECTOR (0 to 7);
    COUT : out BIT );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
  signal C : BIT_VECTOR (0 to 8);
begin
  C(0) <= CIN;
  COUT <= C(8);
  LOOP_ADD : for I in 0 to 7 generate
    SUM(I) <= A(I) xor B(I) xor C(I);
    C(I+1) <= (A(I) and B(I)) or (A(I) and C(I)) or (B(I) and C(I));
  end generate;
end ARCHI;
```

if – generate 文と for – generate 文を使用した N ビット加算器のコード例

スタティック条件でも if – generate 文がサポートされます。次のコード例は、4 ～ 32 ビット幅の汎用 N ビット加算器を示しています。

```
entity EXAMPLE is
  generic (N : INTEGER := 8);
  port (
    A,B  : in BIT_VECTOR (N downto 0);
    CIN  : in BIT;
    SUM  : out BIT_VECTOR (N downto 0);
    COUT : out BIT );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
  signal C : BIT_VECTOR (N+1 downto 0);
begin
  L1: if (N>=4 and N<=32) generate
    C(0) <= CIN;
    COUT <= C(N+1);
    LOOP_ADD : for I in 0 to N generate
      SUM(I) <= A(I) xor B(I) xor C(I);
      C(I+1) <= (A(I) and B(I)) or (A(I) and C(I)) or (B(I) and C(I));
    end generate;
  end generate;
end ARCHI;
```

VHDL の組み合わせプロセス文

プロセス文では、同時処理信号代入文とは異なる方法で信号に値が代入されます。値の代入は順次処理されます。ある代入によりその前の代入が無効になる場合があります。「プロセス文内で代入をしたコード例」を参照してください。まず信号 S に 0 が代入されますが、その後 (for (A and B) =1) で S の値が 1 に変化します。

推論されたハードウェアにメモリ エLEMENTが含まれない場合、プロセスは組み合わせプロセスとなります。つまり、プロセス文で代入された信号すべてがプロセス文のすべてのパスで代入される場合、プロセスは組み合わせプロセスとなります。

組み合わせプロセス文には、process の後にかっこで囲まれたセンシティビティリストがあります。センシティビティリストにある信号のいずれかに変化 (イベント) が起こると、プロセスが実行されます。組み合わせプロセスの場合、センシティビティリストには次を含める必要があります。

- ・ if や case などの条件で使用するすべての信号
- ・ 代入文の右側の信号すべて

センシティビティリストに含まれていない信号がある場合、それを示す警告メッセージが表示され、センシティビティリストにその信号が追加されます。このため、合成結果が最初のデザイン仕様と異なることがあります。

プロセスには、ローカル変数を含めることができます。変数は信号と同様に処理されますが、デザインには出力されません。

「組み合わせプロセスのコード例 1」では、プロセスの宣言部分で変数 AUX が宣言され、プロセス文で「:=」を使用して値が代入されています。

組み合わせプロセス文では、if 文または case 文のすべての分岐で信号が明示的に代入されていない場合、最後の値を保持するためにラッチが作成されます。ラッチが作成されないようにするには、組み合わせプロセス文で定義したすべての信号がプロセスのすべての条件に対して常に明示的に代入されるようにします。

プロセス文には、次の文を含めることができます。

- ・ 変数代入文および信号代入文
- ・ if 文
- ・ case 文
- ・ for - loop 文
- ・ 関数およびプロシージャ呼び出し

プロセス文内で代入をしたコード例

```
entity EXAMPLE is
  port (
    A, B : in BIT;
    S : out BIT );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
  process (A, B)
  begin
    S <= '0' ;
    if ((A and B) = '1') then
      S <= '1' ;
    end if;
  end process;
end ARCHI;
```

コード例 1

```
library ASYL;
use ASYL.ARITH.all;

entity ADDSUB is
  port (
    A,B : in BIT_VECTOR (3 downto 0);
    ADD_SUB : in BIT;
    S : out BIT_VECTOR (3 downto 0) );
end ADDSUB;

architecture ARCHI of ADDSUB is
begin
  process (A, B, ADD_SUB)
    variable AUX : BIT_VECTOR (3 downto 0);
  begin
    if ADD_SUB = '1' then
      AUX := A + B ;
    else
      AUX := A - B ;
    end if;
    S <= AUX;
  end process;
end ARCHI;
```

コード例 2

```

entity EXAMPLE is
  port (
    A, B : in BIT;
    S : out BIT );
end EXAMPLE;

architecture ARCH1 of EXAMPLE is
begin
  process (A,B)
    variable X, Y : BIT;
  begin
    X := A and B;
    Y := B and A;
    if X = Y then
      S <= '1' ;
    end if;
  end process;
end ARCH1;

```

if - else 文

if - else 文

- ・ 真偽条件 (true か false か) によって実行される文が決定されます。
- ・ ネスト化できます。
- ・ キーワード begin と end を使用すると、複数文から成り立つブロックを実行できます。

条件の評価結果	実行される文
true	最初の文
false	else 文
x	else 文
z	else 文

コード例

```

library IEEE;
use IEEE.std_logic_1164.all;

entity mux4 is
  port (
    a, b, c, d : in std_logic_vector (7 downto 0);
    sel1, sel2 : in std_logic;
    outmux : out std_logic_vector (7 downto 0));
end mux4;

architecture behavior of mux4 is
begin

```

```

process (a, b, c, d, sel1, sel2)
begin
  if (sel1 = '1') then
    if (sel2 = '1') then
      outmux <= a;
    else
      outmux <= b;
    end if;
  else
    if (sel2 = '1') then
      outmux <= c;
    else
      outmux <= d;
    end if;
  end if;
end process;
end behavior;

```

VHDL の case 文

case 文は論理式を比較し、並列分岐の 1 つを実行します。分岐は記述された順に評価され、最初に true になる分岐が実行されます。一致する分岐が見つからない場合は、デフォルトの分岐が実行されます。

case 文の VHDL コード例

```

library IEEE;
use IEEE.std_logic_1164.all;

entity mux4 is
  port (
    a, b, c, d : in std_logic_vector (7 downto 0);
    sel : in std_logic_vector (1 downto 0);
    outmux : out std_logic_vector (7 downto 0));
end mux4;
architecture behavior of mux4 is
begin
  process (a, b, c, d, sel)
  begin
    case sel is
      when "00" => outmux <= a;
      when "01" => outmux <= b;
      when "10" => outmux <= c;
      when others => outmux <= d; -- case statement
                                   -- must be complete
    end case;
  end process;
end behavior;

```

for – loop 文

XST では、次のに対する for 文がサポートされています。

- ・ 定数の範囲
 - ・ 次の演算子を使用したテスト条件の停止
 - <
 - <=
 - >
 - >=
 - ・ 次のいずれかに適合する次ステップの計算
 - *var* = *var* + step
 - *var* = *var* - step
- 説明：
- ◆ *var* はループ変数
 - ◆ *step* は定数値
-
- ・ **next** 文および **exit** 文

for – loop 文のコード例

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity countzeros is
  port (
    a : in std_logic_vector (7 downto 0);
    Count : out std_logic_vector (2 downto 0) );
end mux4;

architecture behavior of mux4 is
  signal Count_Aux: std_logic_vector (2 downto 0);
begin
  process (a)
  begin
    Count_Aux <= "000";
    for i in a'range loop
      if (a[i] = '0') then
        Count_Aux <= Count_Aux + 1; -- operator "+" defined
                                   -- in std_logic_unsigned
      end if;
    end loop;
    Count <= Count_Aux;
  end process;
end behavior;
```

VHDL の順序回路

順序回路は、順次プロセス文を使用して記述できます。XST では、次の 2 種類の文を使用できます。

- ・ [VHDL のセンシティビティリスト付き順次プロセス文](#)
- ・ [VHDL のセンシティビティリストのない順次プロセス文](#)

VHDL のセンシティビティ リスト付き順次プロセス文

プロセスが組み合わせプロセスでない場合、順次プロセスになります。つまり、プロセスのある条件に対して代入されていない信号がある場合、プロセスは順次プロセスになります。この場合、内部ステートまたはメモリ (フリップフロップまたはラッチ) が生成されます。

次のコード例は、順序回路記述のテンプレートです。

詳細は、次を参照してください。

[HDL コーディング手法](#)

このトピックでは、レジスタおよびカウンタなどのマクロ推論について説明します。

コード例

非同期信号は、センシティビティリストで宣言する必要があります。宣言しない場合は警告メッセージが表示され、センシティビティリストに自動的に追加されます。この場合、合成結果のビヘイビアは最初の仕様と異なることがあります。

```
process (CLK, RST) ...
begin
    if RST = <'0' | '1'> then
        -- an asynchronous part may appear here
        -- optional part
        .....
    elsif <CLK'EVENT | not CLK'STABLE>
        and CLK = <'0' | '1'> then
        -- synchronous part
        -- sequential statements may appear here
    end if;
end process;
```

VHDL のセンシティビティ リストのない順次プロセス文

センシティビティリストのない順次プロセス文には、wait 文を含める必要があります。wait 文は、プロセス文の冒頭に記述します。wait 文の条件には、クロック信号の条件を使用します。プロセスに複数の wait 文を記述する場合は、一定の条件に従う必要があります。

詳細は、次を参照してください。

[VHDL での複数の wait 文の記述](#)

センシティビティリストのないプロセス文では、非同期信号イベントは指定できません。

センシティビティ リストなしの順次プロセスを記述した VHDL コード例

次の VHDL コード例では、で説明されているプロセス文の記述例を示します。クロック条件は立ち下りエッジまたは立ち上がりエッジです。

```
process ...
begin
    wait until <CLK'EVENT | not CLK' STABLE> and CLK = <'0' | '1'>;
    ... -- a synchronous part may be specified here.
end process;
```

XST では同じ wait 文 内でのクロックおよびクロック イネーブルの記述がサポートされないの
で、「クロックおよびクロック イネーブル (サポートあり) のコード例」に示すように記述します。

XST では、ラッチの記述に wait 文はサポートされていません。

クロックおよびクロック イネーブル (サポートなし) のコード例

注意： このコード形式は、サポートされません。

```
wait until CLOCK'event and CLOCK = '0' and ENABLE = '1' ;
```

クロックおよびクロック イネーブル (サポートあり) のコード例

"8 Bit Counter Description Using a Process with a Sensitivity List" if ENABLE = '1' then ...

レジスタおよびカウンタの VHDL コード例

コード例は、ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip からダウンロードしてください。

センシティビティ リスト付きプロセス文で記述した 8 ビット レジスタのコード例

```
entity EXAMPLE is
    port (
        DI  : in BIT_VECTOR (7 downto 0);
        CLK : in BIT;
        DO  : out BIT_VECTOR (7 downto 0) );
end EXAMPLE;

architecture ARCH1 of EXAMPLE is
begin
    process (CLK)
    begin
        if CLK'EVENT and CLK = '1' then
            DO <= DI ;
        end if;
    end process;
end ARCH1;
```

センシティブティリストのないプロセス文 (wait 文あり) で記述した 8 ビットレジスタのコード例

```
entity EXAMPLE is
  port (
    DI  : in BIT_VECTOR (7 downto 0);
    CLK : in BIT;
    DO  : out BIT_VECTOR (7 downto 0) );
end EXAMPLE;

architecture ARCH1 of EXAMPLE is
begin
  process begin
    wait until CLK'EVENT and CLK = '1';
    DO <= DI;
  end process;
end ARCH1;
```

クロック信号および非同期リセット信号を持つ 8 ビットレジスタのコード例

```
entity EXAMPLE is
  port (
    DI  : in BIT_VECTOR (7 downto 0);
    CLK : in BIT;
    RST : in BIT;
    DO  : out BIT_VECTOR (7 downto 0));
end EXAMPLE;

architecture ARCH1 of EXAMPLE is
begin
  process (CLK, RST)
  begin
    if RST = '1' then
      DO <= "00000000";
    elsif CLK'EVENT and CLK = '1' then
      DO <= DI ;
    end if;
  end process;
end ARCH1;
```

センシティビティ リスト付きプロセス文で記述した 8 ビット カウンタのコード例

```
library ASYL;
use ASYL.PKG_ARITH.all;

entity EXAMPLE is
  port (
    CLK : in BIT;
    RST : in BIT;
    DO  : out BIT_VECTOR (7 downto 0));
end EXAMPLE;

architecture ARCH1 of EXAMPLE is
begin
  process (CLK, RST)
    variable COUNT : BIT_VECTOR (7 downto 0);
  begin
    if RST = '1' then
      COUNT := "00000000";
    elsif CLK'EVENT and CLK = '1' then
      COUNT := COUNT + "00000001";
    end if;
    DO <= COUNT;
  end process;
end ARCH1;
```

VHDL での複数の wait 文の記述

順序回路は、プロセスで複数の wait 文を使用して記述できます。複数の wait 文を使用するには、次の規則に従う必要があります。

- ・ プロセスに loop 文を 1 つだけ含める
- ・ loop 文の冒頭に wait 文を記述する
- ・ wait 文の後に next 文または exit 文を使用する
- ・ すべての wait 文で同じ条件を使用する
- ・ wait 文の条件には 1 つのクロック信号のみを使用する
- ・ wait 文の条件には次のような形式を使用する

```
“wait [on clock_signal] until [(clock_signal'EVENT | not clock_signal'STABLE) and ]
clock_signal = {'0' | '1'};”
```

コード例

次の VHDL コード例では、複数の wait 文を使用し、4 つの動作を連続して実行する順次回路の記述を示します。デザイン サイクルは、クロック信号の連続した 2 つの立ち上がりエッジで区切られています。また、動作シーケンスを最初から再開できるように、同期リセットが定義されています。この動作シーケンスでは、次の 4 入力をそれぞれ RESULT 出力に代入しています。

- ・ DATA1
- ・ DATA2
- ・ DATA3
- ・ DATA4

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity EXAMPLE is
  port (
    DATA1, DATA2, DATA3, DATA4 : in STD_LOGIC_VECTOR (3 downto 0);
    RESULT : out STD_LOGIC_VECTOR (3 downto 0);
    CLK : in STD_LOGIC;
    RST : in STD_LOGIC );
end EXAMPLE;

architecture ARCH of EXAMPLE is
begin
  process begin
    SEQ_LOOP : loop
      wait until CLK'EVENT and CLK = '1';
      exit SEQ_LOOP when RST = '1';
      RESULT <= DATA1;

      wait until CLK'EVENT and CLK = '1';
      exit SEQ_LOOP when RST = '1';
      RESULT <= DATA2;

      wait until CLK'EVENT and CLK = '1';
      exit SEQ_LOOP when RST = '1';
      RESULT <= DATA3;

      wait until CLK'EVENT and CLK = '1';
      exit SEQ_LOOP when RST = '1';
      RESULT <= DATA4;
    end loop;
  end process;
end ARCH;
```

VHDL の関数とプロシージャ

関数 (ファンクション) 宣言およびプロシージャ宣言は、デザインでブロックを複数回使用する場合に有益です。関数およびプロシージャは、エンティティの宣言部、アーキテクチャ、またはパッケージで宣言できます。ヘッダ部には次が含まれます。

- ・ 関数と入力には入力パラメータ
- ・ プロシージャには出力と入力パラメータ

これらのパラメータには制約を設定する必要はありません。制約を設定しないということは、パラメータが特定の範囲に制限されないということです。内容は組み合わせプロセス文に類似しています。

レゾリューション関数は、IEEE std_logic_1164 パッケージで定義されるもの以外は、サポートされません。

関数宣言と関数呼び出しのコード例

次に、関数をパッケージで宣言する VHDL のコード例を示します。ここで宣言されている ADD 関数は、1 ビット加算器です。この関数はアーキテクチャ内のパラメータで 4 回呼び出され、4 ビット加算器を作成します。「プロシージャ宣言とプロシージャ呼び出しのコード例」では、プロシージャを使用して同じ機能を記述した例を示しています。

```
package PKG is
    function ADD (A,B, CIN : BIT )
        return BIT_VECTOR;
end PKG;

package body PKG is
    function ADD (A,B, CIN : BIT )
        return BIT_VECTOR is
            variable S, COUT : BIT;
            variable RESULT : BIT_VECTOR (1 downto 0);
        begin
            S := A xor B xor CIN;
            COUT := (A and B) or (A and CIN) or (B and CIN);
            RESULT := COUT & S;
            return RESULT;
        end ADD;
end PKG;

use work.PKG.all;

entity EXAMPLE is
    port (
        A,B : in BIT_VECTOR (3 downto 0);
        CIN : in BIT;
        S : out BIT_VECTOR (3 downto 0);
        COUT : out BIT );
end EXAMPLE;

architecture ARCH1 of EXAMPLE is
    signal S0, S1, S2, S3 : BIT_VECTOR (1 downto 0);
    begin
        S0 <= ADD (A(0), B(0), CIN);
        S1 <= ADD (A(1), B(1), S0(1));
        S2 <= ADD (A(2), B(2), S1(1));
        S3 <= ADD (A(3), B(3), S2(1));
        S <= S3(0) & S2(0) & S1(0) & S0(0);
        COUT <= S3(1);
    end ARCH1;
```

プロシージャ宣言とプロシージャ呼び出しのコード例

```
package PKG is
  procedure ADD (
    A,B, CIN : in BIT;
    C : out BIT_VECTOR (1 downto 0) );
end PKG;

package body PKG is
  procedure ADD (
    A,B, CIN : in BIT;
    C : out BIT_VECTOR (1 downto 0)
  ) is
    variable S, COUT : BIT;
  begin
    S := A xor B xor CIN;
    COUT := (A and B) or (A and CIN) or (B and CIN);
    C := COUT & S;
  end ADD;
end PKG;

use work.PKG.all;

entity EXAMPLE is
  port (
    A,B : in BIT_VECTOR (3 downto 0);
    CIN : in BIT;
    S : out BIT_VECTOR (3 downto 0);
    COUT : out BIT );
end EXAMPLE;

architecture ARCH1 of EXAMPLE is
  begin
    process (A,B,CIN)
      variable S0, S1, S2, S3 : BIT_VECTOR (1 downto 0);
    begin
      ADD (A(0), B(0), CIN, S0);
      ADD (A(1), B(1), S0(1), S1);
      ADD (A(2), B(2), S1(1), S2);
      ADD (A(3), B(3), S2(1), S3);
      S <= S3(0) & S2(0) & S1(0) & S0(0);
      COUT <= S3(1);
    end process;
  end ARCH1;
```

再帰関数のコード例

XST では再帰関数もサポートされます。次の例は、n! 関数を示します。

```
function my_func(x : integer) return integer is
begin
    if x = 1 then
        return x;
    else
        return (x*my_func(x-1));
    end if;
end function my_func;
```

VHDL のアサート文

XST では、VHDL のアサート文がサポートされています。アサート文を使用すると、次に対して使用される不正な値など、VHDL デザインの問題を検出できます。

- ・ 条件 :
 - ジェネリック文
 - 定数
 - ジェネレート文
- ・ 呼び出された関数のパラメータ

アサート文で検出されたエラーは、問題のレベルに応じて次のいずれかとして表示されます。

- ・ 警告メッセージが表示されます。
- ・ デザインは合成できず、エラーメッセージが表示されます。

XST では、スタティック条件の **Assert** 文のみがサポートされます。

コード例

次のコード例には、シフトレジスタを記述する **SINGLE_SRL** というブロックが含まれています。このシフトレジスタのサイズは、**SRL_WIDTH** というジェネリック値によって決定します。アサート文により、1 つのシフトレジスタのサイズが 1 つの Shift Register LUT (SRL) のサイズを超えていないかどうかチェックされます。

SRL のサイズは 16 ビットであり、シフトレジスタの最後の段はスライスのフリップフロップを使用してインプリメントされるので、シフトレジスタの最大サイズは 17 ビットです。SINGLE_SRL ブロックは、TOP というエンティティ内で 2 回インスタンス化されています。

- ・ 最初は SRL_WIDTH = 13 でインスタンス化
- ・ 2 回目は SRL_WIDTH = 18 でインスタンス化

```
library ieee;
use ieee.std_logic_1164.all;

entity SINGLE_SRL is
    generic (SRL_WIDTH : integer := 16);
    port (
        clk : in std_logic;
        inp : in std_logic;
        outp : out std_logic);
```

```
end SINGLE_SRL;

architecture beh of SINGLE_SRL is
    signal shift_reg : std_logic_vector (SRL_WIDTH-1 downto 0);
begin

    assert SRL_WIDTH <= 17
    report "The size of Shift Register exceeds the size of a single SRL"
    severity FAILURE;

    process (clk)
    begin
        if (clk'event and clk = '1') then
            shift_reg <= shift_reg (SRL_WIDTH-1 downto 1) & inp;
        end if;
    end process;

    outp <= shift_reg(SRL_WIDTH-1);
end beh;

library ieee;
use ieee.std_logic_1164.all;

entity TOP is
    port (
        clk : in std_logic;
        inp1, inp2 : in std_logic;
        outp1, outp2 : out std_logic);
end TOP;

architecture beh of TOP is
    component SINGLE_SRL is
        generic (SRL_WIDTH : integer := 16);
    port(
        clk : in std_logic;
        inp : in std_logic;
        outp : out std_logic);
    end component;
begin
    inst1: SINGLE_SRL generic map (SRL_WIDTH => 13)
    port map(
        clk => clk,
        inp => inp1,
        outp => outp1 );
    inst2: SINGLE_SRL generic map (SRL_WIDTH => 18)
    port map(
        clk => clk,
        inp => inp2,
        outp => outp2 );
end beh;
```

エラー メッセージ

上記のコードを XST で実行すると、次のエラー メッセージが表示されます。

```
...
=====
*                               HDL Analysis                               *
=====
Analyzing Entity <top> (Architecture <beh>).
Entity <top> analyzed. Unit <top> generated.

Analyzing generic Entity <single_srl> (Architecture <beh>).
  SRL_WIDTH = 13
Entity <single_srl> analyzed. Unit <single_srl> generated.

Analyzing generic Entity <single_srl> (Architecture <beh>).
  SRL_WIDTH = 18
ERROR:Xst - assert_1.vhd line 15: FAILURE:
The size of Shift Register exceeds the size of a single SRL
...
```

パッケージ文を使用して定義した VHDL モデル

このセクションでは、次の項目について説明します。

- ・ [パッケージ文を使用して定義した VHDL モデルの概要](#)
- ・ [標準パッケージ文を使用した VHDL モデルの定義](#)
- ・ [IEEE パッケージ文を使用した VHDL モデルの定義](#)
- ・ [Synopsys パッケージ文を使用した VHDL モデルの定義](#)

パッケージ文を使用して定義した VHDL モデルの概要

VHDL モデルは、パッケージ文を使用して定義します。パッケージ文には、次が含まれます。

- ・ タイプおよびサブタイプ宣言
- ・ 定数宣言
- ・ 関数およびプロシージャ宣言
- ・ コンポーネント宣言

パッケージ文を使用すると、定数値や関数定義などのデザインのパラメータや定数を変更できます。

パッケージは、次から構成されます。

- ・ 本体の宣言
- ・ パッケージ宣言

パッケージ本体には、パッケージ宣言で宣言された関数本体の記述が含まれます。

```
library lib_pack;  
-- lib_pack is the name of the library specified  
-- where the package has been compiled (work by default)  
use lib_pack.pack_name.all;  
-- pack_name is the name of the defined package.
```

また、XST では定義済みのパッケージも使用できます。このパッケージはコンパイル済みで、VHDL デザインに含めることができます。これらのパッケージは主に合成中に使用するもののですが、シミュレーションでも使用できます。

標準パッケージ文を使用した VHDL モデルの定義

標準パッケージは

- ・ デフォルトで提供されます。
- ・ 次の基本的なデータ型が含まれます。
 - `bit`
 - `bit_vector`
 - `integer`

IEEE パッケージ文を使用して定義した VHDL モデル

XST では、次の IEEE パッケージがサポートされます。

- ・ **std_logic_1164**
次がサポートされます。
 - **std_logic**
 - **std_ulogic**
 - **std_logic_vector**
 - **std_ulogic_vector**
 これらのデータ型に基づいた変換関数もサポートされます。
- ・ **numeric_bit**
ビット型を基にした次のベクタ型がサポートされます。
 - 符号なしベクタ
 - 符号付きベクタ
 XST では、次もサポートされます。
 - ◆ これらのデータ型のオーバーロードされた数値演算子すべて
 - ◆ これらのデータ型の変換および拡張関数
- ・ **numeric_std**
std_logic を基にした次のベクタ型がサポートされます。
 - 符号なしベクタ
 - 符号付きベクタ
 このパッケージは std_logic_arith と同等です。
- ・ **math_real**
次がサポートされます。
 - 「VHDL の実数定数」に記述される実数定数
 - 「VHDL の実数定数」に記述される実数関数
 - 0.0 ～ 1.0 の連続した値を生成するプロシージャ uniform

VHDL の実数定数

定数	値	定数	値
math_e	e	math_log_of_2	ln2
math_1_over_e	1/e	math_log_of_10	ln10
math_pi		math_log2_of_e	log2e
math_2_pi		math_log10_of_e	log10e
math_1_over_pi		math_sqrt_2	
math_pi_over_2		math_1_oversqrt_2	
math_pi_over_3		math_sqrt_pi	
math_pi_over_4		math_deg_to_rad	
math_3_pi_over_2		math_rad_to_deg	

VHDL の実数関数

ceil(x)	realmax(x,y)	exp(x)	cos(x)	cosh(x)
floor(x)	realmin(x,y)	log(x)	tan(x)	tanh(x)
round(x)	sqrt(x)	log2(x)	arcsin(x)	arcsinh(x)
trunc(x)	cbrt(x)	log10(x)	arctan(x)	arccosh(x)
sign(x)	"**"(n,y)	log(x,y)	arctan(y,x)	arctanh(x)
"mod"(x,y)	"**"(x,y)	sin(x)	sinh(x)	

実数 (real) 型と math_real パッケージの関数およびプロシージャは、計算にのみ使用します。XST での合成には使用できません。

コード例

```
library ieee;
use IEEE.std_logic_signed.all;
signal a, b, c : std_logic_vector (5 downto 0);
c <= a + b;
-- this operator "+" is defined in package std_logic_signed.
-- Operands are converted to signed vectors, and function "+"
-- defined in package std_logic_arith is called with signed
-- operands.
```

Synopsys パッケージ文を使用した VHDL モデルの定義

IEEE ライブラリでは、次の Synopsys パッケージがサポートされます。

- ・ **std_logic_arith**
ベクタ型 (符号なし、符号付き) と、オーバーロードされた数値演算子が含まれます。変換関数や拡張関数も定義します。
- ・ **std_logic_unsigned**
std_logic_vector で数値演算子を使用し、符号なし演算を定義します。
- ・ **std_logic_signed**
std_logic_vector に対する数値演算子を、符号付き演算子として定義します。
- ・ **std_logic_misc**
次の std_logic_1164 パッケージの補足タイプ、サブタイプ、定数、関数を定義します。
 - **and_reduce**
 - **or_reduce**

サポートされる VHDL 構文

XST では、次の VHDL コンストラクトがサポートされます。

- ・ [デザイン エンティティとコンフィギュレーション](#)
- ・ [論理式](#)
- ・ [ステートメント](#)

VHDL のデザイン エンティティとコンフィギュレーション

メモ： XST では、信号名の冒頭文字にアンダースコアを使用できません (DATA_1 など)。

XST では、次以外の VHDL デザイン エンティティとコンフィギュレーションがサポートされます。

- ・ [VHDL エンティティ ヘッダ](#)
- ・ [VHDL パッケージ](#)
- ・ [VHDL の物理型](#)
- ・ [VHDL モード](#)
- ・ [VHDL の宣言](#)
- ・ [VHDL のオブジェクト](#)
- ・ [VHDL の詳細設定](#)

VHDL エンティティ ヘッダ

- ・ ジェネリック
サポートあり
- ・ ポート
サポートあり
- ・ エンティティ文
部分的にサポート。次の文が使用可能：
 - 属性宣言
 - 属性仕様
 - 定数宣言

VHDL パッケージ

STANDARD

TIME はサポートなし

VHDL の物理型

- ・ TIME
無視
- ・ REAL
サポートあり (定数計算関数でのみ)

VHDL モード

リンケージ
サポートなし

VHDL の宣言

データ型
次がサポートされます。

- 列挙型
- 定数範囲の正の値の型
- ビット ベクタ型
- 多次元配列

VHDL のオブジェクト

- ・ 定数宣言
サポートあり (ディファード定数を除く)
- ・ 信号宣言
サポートあり (レジスタまたはバス タイプの信号を除く)
- ・ 属性宣言
一部の属性のみサポートし、ほかは無視。
詳細は、次を参照してください。
[デザイン制約](#)

VHDL の詳細設定

- ・ 属性
次のような一部の定義済み属性のみをサポート
 - HIGH
 - LOW
 - LEFT
 - RIGHT
 - RANGE
 - REVERSE_RANGE
 - LENGTH
 - POS
 - ASCENDING
 - EVENT
 - LAST_VALUE
- ・ コンフィギュレーション
インスタンスリストの all 節のみでサポート。節がない場合、デフォルト ライブラリにコンパイルされているエンティティ/アーキテクチャを使用。

- ・ 接続解除
サポートなし

VHDL の論理式

XST では、次の論理式がサポートされます。

- ・ [VHDL の演算子](#)
- ・ [VHDL のオペランド](#)

VHDL の演算子

演算子	サポートの有無
論理演算子： and、or、nand、nor、xor、xnor、not	サポートあり
比較演算子 =, /=, <, <=, >, >=	サポートあり
& (連結)	サポートあり
加算/減算演算子：+, -	サポートあり
*	サポートあり
/, rem	右のオペランドが 2 のべき乗の定数の場合のみサポート
mod	右のオペランドが 2 のべき乗の定数の場合のみサポート
シフト演算子： sll、srl、sla、sra、rol、ror	サポートあり
abs	サポートあり
**	左のオペランドが 2 の場合のみサポート
符号演算子：+, -	サポートあり

VHDL のオペランド

オペランド	サポートの有無
抽象リテラル	整数リテラルのみサポート
物理リテラル	無視
列挙リテラル	サポートあり
文字列リテラル	サポートあり
ビット文字列リテラル	サポートあり
レコード集合	サポートあり
配列集合	サポートあり
関数呼び出し	サポートあり
条件付き論理式	定義済み属性でサポート

オペランド	サポートの有無
型変換	サポートあり
アロケータ	サポートなし
スタティックな論理式	サポートあり

VHDL 文

XST では、次以外の VHDL 文がすべてサポートされます。

- ・ [VHDL の wait 文](#)
- ・ [VHDL のループ文](#)
- ・ [VHDL の同時処理文](#)

VHDL の wait 文

wait 文	サポートの有無
Boolean_expression まで sensitivity_list を待機状態にします。 詳細は、次を参照してください。 VHDL の順序回路	センシティビティリストとブール代数式内の 1 つの信号でサポート。複数の wait 文の場合、各文に同じセンシティビティリストとブール代数式を使用する必要あり。 メモ： XST では、ラッチの記述に wait 文はサポートされていません。
time_expression を待ちます。 詳細は、次を参照してください。 VHDL の順序回路	サポートなし
アサート文	スタティック条件のみサポート
信号代入文 ステートメント	サポートあり (遅延は無視)

VHDL のループ文

ループ文	サポートの有無
for... loop... end loop	定数範囲のみサポート。disable 文はサポートされていません。
loop ... end loop	複数の wait 文でのみサポート

VHDL の同時処理文

同時処理文	サポートの有無
同時処理信号代入文 代入文	サポートあり (after 節、transport/guarded オプション、波形を除く)、UNAFFFECTED はサポートあり
for-generate	定数範囲のみサポート
if-generate	スタティック条件のみサポート

VHDL の予約語

abs	access	after	alias
all	and	architecture	array
assert	attribute	begin	block
body	buffer	bus	case
component	configuration	constant	disconnect
downto	else	elsif	end
entity	exit	file	for
function	generate	generic	group
guarded	if	impure	in
inertial	inout	is	label
library	linkage	literal	loop
map	mod	nand	new
next	nor	not	null
of	on	open	or
others	out	package	port
postponed	procedure	process	pure
range	record	register	reject
rem	report	return	rol
ror	select	severity	signal
shared	sla	sll	sra
srl	subtype	then	to
transport	type	unaffected	units
until	use	variable	wait
when	while	with	xnor
xor			

Verilog 言語のサポート

この章では、XST の Verilog サポートについて説明します。

- ・ [Veilog 言語サポートの概要](#)
- ・ [Verilog ビヘイビア記述](#)
- ・ [変数による部分的ビット選択](#)
- ・ [Verilog 構造記述](#)
- ・ [Verilog パラメータ](#)
- ・ [Verilog パラメータと属性の競合](#)
- ・ [Verilog の制限](#)
- ・ [Verilog の属性とメタ コメント](#)
- ・ [サポートされる Verilog 構文](#)
- ・ [サポートされるシステム タスクと関数](#)
- ・ [Verilog プリミティブ](#)
- ・ [Verilog の予約言語](#)
- ・ [Verilog-2001 のサポート](#)

Veilog 言語サポートの概要

複雑な回路は通常、トップ ダウン手法を使用して設計します。設計プロセスの各段階で、さまざまなレベルでの仕様が必要となります。たとえばアーキテクチャレベルでは、仕様はブロック図または ASM (Algorithmic State Machine) チャートに対応します。ブロックまたは ASM 段階では、次のような N ビットワイヤで接続されるレジスタ転送ブロックに対応します。

- ・ レジスタ
- ・ 加算器
- ・ カウンタ
- ・ マルチプレクサ
- ・ グルー ロジック
- ・ Finite State Machine (FSM)

Verilog のような HDL 言語を使用すると、ASM チャートや回路図などをコンピュータ言語で記述できます。

Verilog ではデザインのビヘイビア記述または構造記述が可能で、さまざまな抽象度でデザインオブジェクトを表現できます。Verilog のような言語を使用してハードウェアを設計すると、並列処理やオブジェクト指向プログラムなど、ソフトウェアの概念を利用できます。Verilog の構文は C 言語および Pascal に類似しています。XST では、IEEE 1364 がサポートされています。

XST でサポートされる Verilog では、グローバル回路および各ブロックを効率的に記述できます。記述されたデザインは、各ブロックに最適なフローを使用して合成されます。ここで合成とは、Verilog のビヘイビア記述と構造記述を、フラット化されたゲートレベルのネットリストにコンパイルすることを指します。生成されたネットリストは、Virtex® デバイスなどのプログラマブル ロジック デバイスをカスタム プログラムするために使用できます。数値演算ブロック、ゲルロー ロジック、および Finite State Machine (FSM) コンポーネントには、それぞれ異なる合成方法が使用されます。

このマニュアルは、Verilog の基礎知識を持つ技術者を対象としています。

詳細は、次を参照してください。

- ・ Verilog デザイン制約とオプション
[デザイン制約](#)
- ・ Verilog 属性の構文
[Verilog-2001 の属性](#)
- ・ ISE® Design Suite のプロセス ウィンドウで Verilog オプションを設定します。
[一般制約](#)
- ・ Verilog の一般情報
[『IEEE Verilog HDL Reference Manual』](#)

Verilog ビヘイビア記述

Verilog のビヘイビア記述については、次を参照してください。

[Verilog ビヘイビア記述のサポート](#)

変数による部分的ビット選択

Verilog 2001 には、変数を使用してベクタから部分的にビットを選択できる機能が追加されています。部分的なビットを選択する変数は、2 つの明示的な値を指定するのではなく、開始位置およびベクタの幅によって定義されます。開始位置は変更できますが、幅は一定の値が維持されます。

変数による部分的ビット選択のためのシンボル

シンボル	説明
+ (プラス)	部分的なビットは開始位置から上方向で選択されます。
- (マイナス)	部分的なビットは開始位置から下方向に選択されます。

コード例

```
reg [3:0] data;  
  reg [3:0] select; // a value from 0 to 7  
  wire [7:0] byte = data[select +: 8];
```

Verilog 構造記述

このセクションでは、次について説明します。

- ・ [Verilog 構造記述の概要](#)
- ・ [定義済みプリミティブのインスタンス化](#)

Verilog 構造記述の概要

Verilog の構造記述では、複数のブロックを組み合わせ、デザインを階層構造にできます。ハードウェア構造の基本概念は、次のとおりです。

- ・ コンポーネント
基本ブロック
- ・ ポート
コンポーネントのI/O コネクタ
- ・ 信号
コンポーネント間のワイヤに対応

Verilog では、コンポーネントはデザイン モジュールで表されます。モジュール宣言では、コンポーネント ポートなどのコンポーネントを外側から見た「外観」が記述されます。モジュールボディでは、コンポーネントのビヘイビアや構造などの「内部」が記述されます。

コンポーネント間の接続は、コンポーネント インスタンス化文で定義されます。これらの文では、あるコンポーネントを別のコンポーネントまたは回路で使用する場合に、インスタンスを指定します。コンポーネント インスタンス化文はそれぞれ識別子で区別されます。

コンポーネント インスタンス化文には、ローカル コンポーネント宣言で宣言されたコンポーネントに名前を指定し、実際の信号やポートをコンポーネント宣言のどのローカル ポートと接続するかを指定する関係リスト (かっこで囲まれたリスト) を含めます。

Verilog には多数の論理ゲートが組み込まれており、これらをインスタンス化して論理回路を作成できます。含まれる論理ゲートは、次のとおりです。

- ・ AND
- ・ OR
- ・ XOR
- ・ NAND
- ・ NOR
- ・ NOT

基本的な XOR 構築コード例

次は、2 つの 1 ビット入力 a および b を持つ基本的な XOR 関数の記述例です。

```
module build_xor (a, b, c);
  input a, b;
  output c;
  wire c, a_not, b_not;
  not a_inv (a_not, a);
  not b_inv (b_not, b);
  and a1 (x, a_not, b);
  and a2 (y, b_not, a);
  or out (c, x, y);
endmodule
```

組み込まれているモジュールの各インスタンスには、次のような固有のインスタンス名が指定されています。

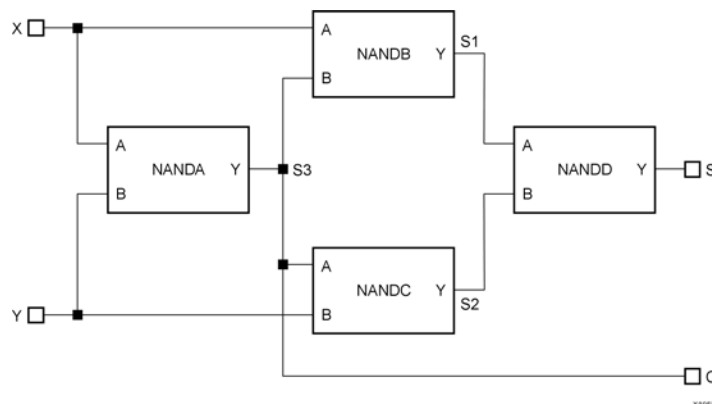
- ・ **a_inv**
- ・ **b_inv**
- ・ **out**

半加算器の構造記述例 (Verilog)

次は、4 つの 2 入力 NAND モジュールで構成される半加算器の構造記述です。

```
module halfadd (X, Y, C, S);
  input X, Y;
  output C, S;
  wire S1, S2, S3;
  nand NANDA (S3, X, Y);
  nand NANDB (S1, X, S3);
  nand NANDC (S2, S3, Y);
  nand NANDD (S, S1, S2);
  assign C = S3;
endmodule
```

合成済みの最上位レベルのネットリストの図



定義済みプリミティブのインスタネーション

Verilog の構造記述では、次のようなあらかじめ定義されているプリミティブをインスタネートして、回路を記述することも可能です。

- ・ ゲート
- ・ レジスタ
- ・ 次のようなザイリンクス特有のプリミティブ :
 - CLKDLL
 - BUFG

これらのプリミティブは Verilog には含まれていませんが、XST Verilog ライブラリ (unisim_comp.v) で提供されています。

レジスタおよび BUFG の構造インスタネーション例

```
module foo (sysclk, in, reset, out);
input sysclk, in, reset;
output out;
reg out;
wire sysclk_out;
FDC register (out, sysclk_out, reset, in);
//position based referencing
BUFG clk (.O(sysclk_out), .I(sysclk));
//name based referencing
...
endmodule
```

次の定義は、XST に含まれる unisim_comp.v ライブラリ ファイルに含まれます。

- ・ FDC
- ・ BUFG

```
(* BOX_TYPE="PRIMITIVE" *) // Verilog-2001
module FDC (Q, C, CLR, D);
parameter INIT = 1'b0;
output Q;
input C;
input CLR;
input D;
endmodule
```

```
(* BOX_TYPE="PRIMITIVE" *) // Verilog-2001
module BUFG ( O, I);
output O;
input I;
endmodule
```

Verilog パラメータ

Verilog モジュールでは、パラメータという定数を定義できます。パラメータは、任意のビット幅の回路を定義するためにモジュール インスタンスに渡されます。パラメータは、デザインに含まれるパラメータ指定したブロックを作成および使用する際の基礎となり、階層構造を作成するのに使用されます。

コード例

次の Verilog コード例では、パラメータを使用しています。null 文字列パラメータは、サポートされていません。

```
module lpm_reg (out, in, en, reset, clk);
    parameter SIZE = 1;
    input in, en, reset, clk;
    output out;
    wire [SIZE-1 : 0] in;
    reg [SIZE-1 : 0] out;
    always @(posedge clk or negedge reset)
    begin
        if (!reset)
            out <= 1'b0;
        else
            if (en)
                out <= in;
            else
                out <= out;    //redundant assignment
    end
endmodule
module top ();    //portlist left blank intentionally
    ...
    wire [7:0] sys_in, sys_out;
    wire sys_en, sys_reset, sysclk;
    lpm_reg #8 buf_373 (sys_out, sys_in, sys_en, sys_reset, sysclk);
    ...
endmodule
```

モジュール lpm_reg を 8 ビット幅でインスタンス化しているため、インスタンス buf_373 の幅が 8 ビットになっています。

ジェネリック (-generics) コマンドライン オプションを使用すると、最上位デザイン ブロックで定義されるパラメータ (Verilog) 値を再定義できます。これにより、IP コアの生成やテストイングフローなどの HDL ソースを変更しなくてもデザイン コンフィギュレーションを簡単に修正できます。

Verilog パラメータと属性の競合

このセクションでは、次について説明します。

- ・ Verilog パラメータと属性の競合回避
- ・ Verilog パラメータと属性の優先順位

Verilog パラメータと属性の競合回避

パラメータおよび属性は、Verilog コードのインスタンスおよびコンポーネントの両方に適用でき、また属性は制約ファイルでも指定できるので、競合が発生する場合があります。

競合を回避するために、XST では次の優先順位の規則が使用されます。

1. インスタンス（下位レベル）での指定がモジュール（上位レベル）での指定より優先されます。
2. パラメータと属性が同じインスタンスまたはコンポーネントに指定される場合、パラメータが優先され、XST で警告メッセージが表示されます。
3. XCF (XST 制約ファイル) で指定される属性は、VHDL コードで指定される属性またはパラメータよりも優先されます。

インスタンスに指定されている属性が XST でモジュールに指定されているパラメータを上書きすると、シミュレーション ツールでそのパラメータが使用されなくなる可能性があります。この結果、シミュレーションの結果が合成結果と一致しない場合があります。

Verilog パラメータと属性の優先順位

	インスタンスのパラメータ	モジュールのパラメータ
インスタンスの属性	パラメータを適用(XST で警告メッセージが表示される)	属性を適用 (シミュレーションで不一致の可能性あり)
モジュールの属性	パラメータを適用	パラメータを適用(XST で警告メッセージが表示される)
XCF に含まれる属性	属性を適用(XST で警告メッセージが表示される)	属性を適用

モジュール定義のセキュリティ属性は、ほかのどの属性またはパラメータよりも優先されます。

Verilog の制限

このセクションでは、次について説明します。

- ・ [Verilog の大文字/小文字の区別](#)
- ・ [Verilog のブロッキングおよびノンブロッキング代入文](#)
- ・ [Verilog の整数処理](#)

Verilog の大文字/小文字の区別

Verilog では大文字と小文字が区別されるため、モジュールやインスタンス名の大文字と小文字を変更するだけでそれらが異なるものとして認識されますが、ファイル名や混合言語、その他ツールとの互換性を考慮し、大文字や小文字の違いに頼らず、固有の名前にすることをお勧めします。

XST では、大文字/小文字が違うだけのモジュール名は使用できず、デザイン フローのその他の大文字/小文字の区別がないツールで問題がないように、インスタンスや信号名が変更されます。

大文字/小文字に関するXST のサポート

XST では、次のように大文字/小文字の区別がサポートされています。

- ・ I/O ポート、ネット、レジスタ、メモリには、大文字/小文字が異なる同じ名前を使用できます。
- ・ 同じ名前には、接尾辞が付けられます (rnm<Index>)。
- ・ 名前を変更する構文は NGC ファイルで生成されます。
- ・ 大文字/小文字のみが異なる Verilog 識別子を使用できます。これらの識別子には、XST により接尾辞が付けられます。

たとえば、次のように入力します。

```
module upperlower4 (input1, INPUT1, output1, output2);
    input input1;
    input INPUT1;
```

この例の場合、INPUT1 は INPUT1_rnm0 という名前に変更されます。

XST 内での Verilog の制限

次のように、ブロック、タスク、関数などに同じ名前は使用できません。

```
...
always @(clk)
begin: fir_main5
    reg [4:0] fir_main5_wl;
    reg [4:0] fir_main5_Wl;
```

この場合、XST で次のようなエラー メッセージが表示されます。

```
ERROR:Xst:863 - "design.v", line 6: Name conflict
(<fir_main5/fir_main5_wl> and <fir_main5/fir_main5_Wl>)
```

モジュール名に大文字/小文字が異なる同じ名前は使用できません。

```
module UPPERLOWER10 (...);  
...  
module upperlower10 (...);  
...
```

この場合、XST で次のようなエラー メッセージが表示されます。

```
ERROR:Xst:909 - Module name conflict (UPPERLOWER10 and  
upperlower10)
```

Verilog のブロッキングおよびノンブロッキング代入文

このセクションでは、ブロッキング代入文とノンブロッキング代入文を使用したエラーになる 2 つのコード例を示します。

エラーになるコード例 1

XST では、次の例のように信号をブロッキング代入文およびノンブロッキング代入文の両方で指定できません。

```
always @(in1)  
begin  
    if (in2)  
        out1 = in1;  
    else  
        out1 <= in2;  
    end  
end
```

エラーになるコード例 2

次の例では、実際にはブロッキング代入とノンブロッキング代入が混合しているわけではありませんが、エラーが発生します。

```
if (in2)  
    begin  
        out1[0] = 1'b0;  
        out1[1] <= in1;  
    end  
else  
    begin  
        out1[0] = in2;  
        out1[1] <= 1'b1;  
    end  
end
```

変数がブロッキングとノンブロッキング代入文の両方で指定されると、XST で次のエラー メッセージが表示されます。

```
ERROR:Xst:880 - "design.v", line n:  
Cannot mix blocking and non-blocking assignments on signal <out1>.
```

ビットおよびスライスにブロッキング代入文およびノンブロッキング代入文の両方を使用することもできません。

エラーは、ビットレベルではなく、信号レベルで確認されます。

このエラーが複数ある場合は、最初の 1 つのみがレポートされます。

信号が指定されている行が複数ある場合は、エラー メッセージに示される行番号が正しくないこともあります (信号が代入されている箇所が複数行に渡っていることがあるため)。

Verilog の整数処理

XST では、Verilog の整数がほかの合成ツールと異なる方法で処理される場合があるので、コードを記述する際に注意が必要です。ビットサイズ指定のない case 文や連結文が使用されると、結果が予測不可能になります。

case 文でビット サイズ指定のない場合

Verilog の case 文でビットサイズ指定のない整数が使用されると、結果が予測不可能になります。

次の例では、case 文の最初に使用される 4 のビットが指定されていないので、結果が予測不可能になっています。この問題を回避するには、例の後半のように 4 を 3 ビットに指定します。

```
reg [2:0] condition1;

always @(condition1)
begin
  case(condition1)
    4      : data_out = 2;    // < will generate bad logic
    3'd4   : data_out = 2;    // < will work
  endcase
end
```

連結文でビット サイズ指定のない場合

Verilog の連結文でビット指定のない整数を使用すると、結果が予測不可能になります。

ビット サイズが指定されない式を使用すると、次のようになります。

1. その式が一時的な信号に代入されます。
2. 連結文の一時信号を次のように使用します。

```
reg [31:0] temp;
assign temp = 4'b1111 % 2;
assign dout = {12/3,temp,din};
```

Verilog の属性とメタ コメント

XST では、Verilog で次の両方がサポートされます。

- ・ Verilog-2001 形式の属性

Verilog-2001 の属性はよく使用されてきているので、ザイリンクスではこちらの使用をお勧めしています。

- ・ Verilog メタ コメント

メタ コメントとは、Verilog の解析で認識されるコメントのことです。

Verilog-2001 の属性

XST では Verilog-2001 属性文がサポートされています。属性は、合成ツールなどのソフトウェア ツールに特定の情報を渡すために使用します。Verilog-2001 の属性は、モジュール宣言およびインスタネーション内で、演算子または信号に指定できます。その他の属性宣言はコンパイラでサポートされる場合がありますが、XST では無視されます。

Verilog のメタ コメント

Verilog メタ コメントは、次のために使用します。

- ・ 次のような個々のオブジェクトに制約を設定する場合
 - モジュール
 - インスタンス
 - ネット
- ・ 合成で次のような制約を設定します。
 - parallel_case および full_case
 - translate_on および translate_off
 - すべてのツール特有の制約

例：

syn_sharing

詳細は、次を参照してください。

[デザイン制約](#)

XST では、C 言語スタイルおよび Verilog スタイルのメタ コメントがサポートされます。

Verilog のメタ コメントの記述方法

スタイル	構文	ルール
C スタイル	<code>/* ... */</code>	コメントを複数行にできます。
Verilog スタイル	<code>// ...</code>	行末までがコメントとして識別されます。

サポートされる制約

XST では、次の制約がサポートされます。

- ・ 変換なし (TRANSLATE_OFF) と変換あり (TRANSLATE_ON) 制約

```
// synthesis translate_on  
// synthesis translate_off
```

- ・ パラレル ケース (PARALLEL_CASE)

```
// synthesis parallel_case full_case  
// synthesis parallel_case  
// synthesis full_case
```

- ・ 各オブジェクトに対する制約

構文

```
// synthesis attribute [of] ObjectName [is] AttributeValue
```

コード例

```
// synthesis attribute RLOC of u123 is R11C1.S0  
// synthesis attribute HUSER u1 MY_SET  
// synthesis attribute fsm_extract of State2 is "yes"  
// synthesis attribute fsm_encoding of State2 is "gray"
```

サポートされる Verilog 構文

このセクションでは、次の XST でサポートされる Verilog 構文について説明します。

- ・ 定数
- ・ データ型
- ・ 継続代入文
- ・ 手続き代入文
- ・ デザイン階層
- ・ コンパイラ制約

メモ：XST では、信号名の冒頭文字にアンダースコアを使用できません (DATA_1 など)。

サポートされる Verilog の定数

定数	サポートの有無
整数	サポートあり
実数	サポートあり
文字列定数	サポートなし

サポートされる Verilog のデータ型

XST では次の表に示していない限り、すべての Verilog データ型がサポートされます。

ネットタイプ	駆動電流	レジスタ	指定イベント
tri0、tri1、triereg はサポートされません。	駆動電流はすべて無視されます。	real および realtime レジスタはサポートされません。	指定イベントはすべてサポートされません。

サポートされる Verilog の継続代入文

継続代入文	サポートの有無
駆動電流	無視
遅延	無視

サポートされる Verilog の手続き代入文

XST では、次の場合を除き、Verilog の手続き代入文がサポートされます。

- ・ **assign**
制限付きでサポートあり。
詳細は、次を参照してください。
[Verilog ビヘイビア記述の assign 文および deassign 文](#)
- ・ **deassign**
制限付きでサポートあり。
詳細は、次を参照してください。
[Verilog ビヘイビア記述の assign 文および deassign 文](#)
- ・ **force**
サポートなし
- ・ **release**
サポートなし
- ・ **forever** 文
サポートなし
- ・ **repeat** 文
サポートあり (repeat 値は定数にする必要あり)
- ・ **for** 文
サポートあり (範囲はスタティックな値にする必要あり)
- ・ **delay (#)**
無視
- ・ **event (@)**
サポートなし
- ・ **wait**
サポートなし
- ・ **指定イベント**
サポートなし
- ・ **パラレル ブロック**
サポートなし
- ・ **指定ブロック**
無視
- ・ **ディスエーブル**
for および repeat ループ文以外はサポートあり

サポートされる Verilog のデザイン階層

デザイン階層	サポートの有無
モジュール定義	サポートあり
マクロ モジュール定義	サポートなし
階層名	サポートなし
defparam	サポートあり
インスタンスの配列	サポートあり

サポートされる Verilog のコンパイラ指示子

コンパイラ指示子	サポートの有無
'celldefine 'endcelldefine	無視
'default_nettype	サポートあり
'define	サポートあり
'ifdef 'else 'endif	サポートあり
'undef, 'ifndef, 'elsif,	サポートあり
'include	サポートあり
'resetall	無視
'timescale	無視
'unconnected_drive 'nounconnected_drive	無視
'uselib	サポートなし
'file, 'line	サポートあり

サポートされる Verilog のシステム タスクと関数

このセクションでは、次について説明します。

- ・ サポートされるシステム タスクおよび関数
- ・ サポートされないシステム タスク
- ・ 符号付き、符号なしのシステム タスク
- ・ readmemb および readmemh システム タスク
- ・ その他のシステム タスク
- ・ Verilog の display 関数構文例

サポートされるシステム タスクおよび関数

システム タスクと関数	サポートの有無	コメント
\$display	サポートあり	エスケープ シーケンスは %d、%b、%h、%o、%c、および %s に制限されます。
\$fclose	サポートあり	
\$fdisplay	サポートあり	
\$fgets	サポートあり	
\$finish	サポートあり	\$finish はアクティブになることのない条件分岐文でのみサポートされます。
\$fopen	サポートあり	
\$fscanf	サポートあり	エスケープ シーケンスは %b および %d に制限されます。
\$fwrite	サポートあり	
\$monitor	無視	
\$random	無視	
\$readmemb	サポートあり	
\$readmemh	サポートあり	
\$signed	サポートあり	
\$stop	無視	
\$strobe	無視	
\$time	無視	
\$unsigned	サポートあり	
\$write	サポートあり	エスケープ シーケンスは %d、%b、%h、%o、%c、および %s に制限されます。
その他すべて	無視	

サポートされないシステム タスク

サポートされないシステム タスクは、XST の Verilog コンパイラで無視されます。

符号付き、符号なしのシステム タスク

\$signed および \$unsigned システム タスクは、どの式でも次の構文を使用して呼び出すことができます。

- ・ **\$signed(expr)** または
- ・ **\$unsigned(expr)**

これらの呼び出しの戻り値は入力値と同じサイズで、以前の符号にかかわらず、システム タスクで指定した符号に強制されます。

readmemb および readmemh システム タスク

\$readmemb および \$readmemh システム タスクは、ブロック メモリの初期化に使用できます。

詳細は、次を参照してください。

[外部ファイルからの RAM の初期化のコード例](#)

2 進数の場合は \$readmemb、16 進数の場合は \$readmemh を使用します。XST とシミュレータで処理の違いが発生しないようにするため、これらのシステム タスクでは次のようにインデックス パラメータを使用することをお勧めします。次の例を参照してください。

```
$readmemb("rams_20c.data", ram, 0, 7);
```

その他のシステム タスク

残りのシステム タスクは、処理中にコンピュータ画面およびログ ファイルに情報を出力するため、または合成中にファイルを開いて使用するために使用できます。これらのタスクは、initial ブロックから呼び出す必要があります。XST では、次のエスケープ シーケンスのサブセットがサポートされます。

- ・ **%h**
- ・ **%d**
- ・ **%o**
- ・ **%b**
- ・ **%c**
- ・ **%s**

Verilog の display 関数構文例

次に、2 進数の定数値を 10 進数で表示する \$display の構文を示すコード例を示します。

```
parameter c = 8'b00101010;  
initial  
begin  
    $display ("The value of c is %d", c);  
end
```

HDL 解析段階で、次の情報がログ ファイルに記述されます。

```
Analyzing top module <example>.  
c = 8'b00101010  
"foo.v" line 9: $display : The value of c is 42
```

Verilog プリミティブ

このセクションでは、Verilog プリミティブについて説明します。

- ・ サポートされるプリミティブ
- ・ サポートされないプリミティブ
- ・ 構文

サポートされるプリミティブ

XST では、次を除く Verilog のゲートレベルのプリミティブがサポートされます。

- ・ プルダウンとプルアップ
サポートなし
- ・ 駆動電力と遅延
無視
- ・ プリミティブの配列
サポートなし

サポートされないプリミティブ

XST では、次がサポートされません。

- ・ 次のような Verilog のスイッチ レベルのプリミティブ :
 - cmos、nmos、pmos、rcmos、rnmos、rpmos
 - rtran、rtranif0、rtranif1、tran、tranif0、tranif1
- ・ Verilog のユーザー定義のプリミティブ

構文

```
gate_type instance_name (output, inputs,...);
```

コード例

```
and U1 (out, in1, in2); bufif1 U2 (triout, data, trienable);
```

Verilog の予約語

アスタリスク (*) の付いた単語は、Verilog の予約語ですが、XST でサポートされていません。

always	and	assign	automatic
begin	buf	bufif0	bufif1
case	casex	casez	cell*
cmos	config*	deassign	default
defparam	design*	disable	edge
else	end	endcase	endconfig*
endfunction	endgenerate	endmodule	endprimitive
endspecify	endtable	endtask	event
for	force	forever	fork
function	generate	genvar	highz0
highz1	if	ifnone	incdir*
include*	initial	inout	input
instance*	integer	join	large
liblist*	library*	localparam*	macromodule
medium	module	nand	negedge
nmos	nor	noshow-cancelled*	not
notif0	notif1	or	output
parameter	pmos	posedge	primitive
pull0	pull1	pullup	pulldown
pulsetype- _ondetect*	pulsetype- _onevent*	rcmos	real
realtime	reg	release	repeat
rnmos	rpmos	rtran	rtranif0
rtranif1	scalared	show-cancelled*	signed
small	specify	specparam	strong0
strong1	supply0	supply1	table
task	time	tran	tranif0
tranif1	tri	tri0	tri1
triand	trior	trireg	use*
vectored	wait	wand	weak0
weak1	while	wire	wor
xnor	xor		

Verilog-2001 のサポート

XST では、次の Verilog 2001 の機能がサポートされています。

- ・ generate 文
- ・ ポートとデータ型を 1 つの文で宣言
- ・ ANSI 形式のポート リスト
- ・ モジュール パラメータ ポート リスト
- ・ ANSI C 形式のタスク/関数宣言
- ・ カンマで区切ったセンシティブティ リスト
- ・ 組み合わせロジック センシティブティ
- ・ 継続代入文のデフォルト ネット
- ・ デフォルト ネット宣言のディスエーブル
- ・ インデックスの付いたベクタの部分選択
- ・ 多次元配列
- ・ net および real データ型の配列
- ・ 配列のビットおよびビット部分選択
- ・ 符号付きレジスタ、ネット、およびポート宣言
- ・ 符号付き整数
- ・ 符号付き演算式
- ・ 算術シフト演算子
- ・ 32 ビットを超えるビットの自動的な幅拡張
- ・ べき乗演算子
- ・ N ビットのパラメータ
- ・ インライン パラメータの明示
- ・ 固定ローカル パラメータ
- ・ 条件付きコンパイルの拡張
- ・ ファイルおよび行のコンパイラ指示子
- ・ 可変部分ビット選択
- ・ 再帰タスクおよび関数
- ・ 定数関数

詳細は、次を参照してください。

- ・ Sutherland, Stuart 著『Verilog 2001: A Guide to the New Features of the VERILOG Hardware Description Language』(2002)
- ・ IEEE 標準策定委員会著『1364-2001: IEEE Standard Verilog Hardware Description Language』(2001)

Verilog ビヘイビア記述のサポート

この章では、XST の Verilog ビヘイビア言語サポートについて説明します。

- ・ Verilog ビヘイビア記述の変数宣言
- ・ Verilog ビヘイビア記述の初期値
- ・ Verilog ビヘイビア記述のローカル リセット
- ・ Verilog ビヘイビア記述の配列
- ・ Verilog ビヘイビア記述の多次元配列
- ・ Verilog ビヘイビア記述のデータ型
- ・ Verilog ビヘイビア記述で使用可能な文
- ・ Verilog ビヘイビア記述の論理式
- ・ Verilog ビヘイビア記述のブロック
- ・ Verilog ビヘイビア記述のモジュール
- ・ Verilog ビヘイビア記述のモジュール宣言
- ・ Verilog ビヘイビア記述の継続代入文
- ・ Verilog ビヘイビア記述の手続き代入文
- ・ Verilog ビヘイビア記述の定数
- ・ Verilog ビヘイビア記述のマクロ
- ・ Verilog ビヘイビア記述の include ファイル
- ・ Verilog ビヘイビア記述のコメント
- ・ Verilog ビヘイビア記述の generate 文

Verilog ビヘイビア記述の変数宣言

Verilog の変数は、整数または実数として宣言できます。ただし、これらの宣言はテストコードで使用するためのものです。実際のハードウェア記述では、reg や wire などのデータ型を使用できます。

Verilog の変数

データ型	変数の値を指定する文	デフォルト幅	[Verilog -2001] オプション
reg	手続きブロック	1 ビット (スカラ)	符号付き、または符号なし
wire	継続代入文	1 ビット (スカラ)	符号付き、または符号なし

コード例

reg または wire 宣言で N ビット幅 (ベクタ) を指定するには、[] 内に左のビット位置と右のビット位置をコロンで区切って示します。

```
reg [3:0] arb_priority;
wire [31:0] arb_request;
wire signed [8:0] arb_signed;
```

説明 :

- ・ **arb_request[31]** は MSB
- ・ **arb_request[0]** は LSB

Verilog ビヘイビア記述の初期値

Verilog-2001 では、レジスタを宣言する際に初期値を設定できます。

初期値は、次の規則に従って設定する必要があります。

- ・ 定数値を指定する必要があります。
- ・ 以前の初期値に依存できません。
- ・ 関数またはタスク呼び出しは使用できません。
- ・ レジスタに伝搬するパラメータ値にできます。
- ・ ベクタのすべてのビットを指定します。

宣言部でレジスタの初期値を指定した場合、次の時点でレジスタの出力が指定した値に初期化されます。

- ・ グローバル リセット時、または
- ・ 電源投入時

この方法で指定された初期値は次のようになります。

- ・ レジスタの INIT 属性として NGC ファイルに渡されます。
- ・ ローカル リセットには依存しません。

```
reg arb_onebit = 1'b0;  
reg [3:0] arb_priority = 4'b1011;
```

また、ビヘイビア記述の Verilog コードを使用して、レジスタにセット/リセット時の初期値を指定できます。レジスタのリセットラインの値に対してレジスタの値を指定するには、次の例のように記述します。

```
always @(posedge clk)  
begin  
    if (rst)  
        arb_onebit <= 1'b0;  
    end  
end
```

ビヘイビアコードで変数の初期値を設定すると、出力がローカルリセットで制御可能なフリップフロップとしてデザインにインプリメントされ、NGC ファイルに FDP または FDC フリップフロップとして記述されます。

Verilog ビヘイビア記述のローカル リセット

ローカル リセットは、グローバル リセットとは関係なく動作します。ローカル リセットで制御できるレジスタでは、グローバル リセット時 (または電源投入時) とローカル リセット時で異なる値を指定できます。次のコード例では、レジスタ `arb_onebit` はグローバル リセット時には 0、ローカル リセット時 (`rst`) には 1 になるよう設定しています。

コード例

```
module mult(clk, rst, A_IN, B_OUT);
    input clk, rst, A_IN;
    output B_OUT;

    reg arb_onebit = 1'b0;

    always @(posedge clk or posedge rst)
    begin
        if (rst)
            arb_onebit <= 1'b1;
        else
            arb_onebit <= A_IN;
        end
    end
    B_OUT <= arb_onebit;
endmodule
```

電源投入時にはレジスタの出力が 0 に初期化されるよう設定されていますが、ローカル セット/リセットがアクティブになると 1 に初期化されます。

Verilog ビヘイビア記述の配列

Verilog では、reg および wire の配列を次の例のように定義できます。

Verilog ビヘイビア記述の配列のコード例

次の例は、4 ビット幅のエLEMENT が 32 個ある配列を示しており、Verilog の構造記述で次のように指定できます。

```
reg [3:0] mem_array [31:0];
```

Verilog 構造記述の配列のコード例

次の例は、8 ビット幅のエLEMENT が 64 個ある配列を示しており、Verilog の構造記述で次のように指定できます。

```
wire [7:0] mem_array [63:0];
```

Verilog ビヘイビア記述の多次元配列

XST では、2 次元までの多次元配列型がサポートされます。多次元配列はネットまたはさまざまなデータ型で使用できます。配列を使用して代入および数値演算を記述できますが、一度に選択できる配列のエLEMENT は 1 つのみです。システム タスクまたは関数、通常のタスクまたは関数で多次元配列を使用することはできません。

コード例 1

次の例は、8 ビット幅の wire ELEMENT を 256 X 16 個含む配列を示しており、Verilog の構造記述でのみ指定できます。

```
wire [7:0] array2 [0:255][0:15];
```

コード例 2

次の例は、64 ビット幅のレジスタ ELEMENT を 256 X 8 個含む配列を示しており、Verilog のビヘイビア記述でのみ指定できます。

```
reg [63:0] regarray2 [255:0][7:0];
```

Verilog ビヘイビア記述のデータ型

このセクションでは、Verilog ビヘイビア記述のデータ型について説明します。

- ・ [ビット データ型の値](#)
- ・ [サポートされる Verilog データ型](#)
- ・ [ネットおよびレジスタ](#)

ビット データ型の値

Verilog のビット データ型には、次の 4 つの値があります。

- ・ **0**
論理値 0
- ・ **1**
論理値 1
- ・ **x**
不定値
- ・ **z**
ハイ インピーダンス

サポートされる Verilog データ型

XST では、次の Verilog データ型がサポートされます。

- ・ ネット
 - **wire**
 - **tri**
 - **triand/wand**
 - **trior/wor**
- ・ レジスタ
 - **reg**
 - **integer**
- ・ サプライ ネット
 - **supply0**
 - **supply1**
- ・ 定数
 - parameter**
- ・ 多次元配列 (メモリ)

ネットおよびレジスタ

ネットおよびレジスタは次のいずれかにできます。

- ・ 単数ビット (スカラ)
- ・ 複数ビット (ベクタ)

コード例

Verilog モジュールの宣言セクションで使用する Verilog データ型の例を次に示します。

```
wire net1;                // single bit net
reg r1;                   // single bit register
tri [7:0] bus1;           // 8 bit tristate bus
reg [15:0] bus1;          // 15 bit register
reg [7:0] mem[0:127];     // 8x128 memory register
parameter statel = 3'b001; // 3 bit constant
parameter component = "TMS380C16"; // string
```

Verilog ビヘイビア記述で使用可能な文

次に、Verilog のビヘイビア記述で使用可能な文を示します。

- ・ 変数代入文および信号代入文
 - 変数 = 論理式
 - if (条件) 文
 - else 文
 - case (論理式)

```
expression: statement
...
default: statement
endcase
```
 - for (変数 = 論理式; 条件; 変数 = 変数 + 論理式) 文
 - while (条件) 文
 - forever 文
 - function および task
- ・ すべての変数は、integer (整数) または reg (レジスタ) として宣言されます。
メモ : 変数は wire として宣言することはできません。

Verilog ビヘイビア記述の論理式

論理式では、「Verilog ビヘイビア記述でサポートされる演算子」に示すように、数値演算、論理演算、関係演算、条件演算で定数および変数が演算されます。

論理演算は、複数のビットまたは 1 つのビットのいずれに適用されるかで、ビットごとまたは論理にさらに分類されます。

Verilog ビヘイビア記述でサポートされる演算子

演算	論理演算	関係演算	条件演算
+	&	<	?
-	&&	==	
*		===	
**		<=	
/	^	>=	
%	~	>=	
	~~	!=	
	~~	!==	
	<<	>	
	>>		
	<<<		
	>>>		

Verilog ビヘイビア記述でサポートされる論理式

論理式	シンボル	サポートの有無
連接	{}	サポートあり
複製	{}	サポートあり
演算	+, -, *, **	サポートあり
/	2 番目のオペランドが 2 のべき乗の場合のみサポートあり	
剰余	%	2 番目のオペランドが 2 のべき乗の場合のみサポートあり
加算	+	サポートあり
減算	-	サポートあり
乗算	*	サポートあり

論理式	シンボル	サポートの有無
電力	**	サポートあり <ul style="list-style-type: none"> 2 番目のオペランドが負でない場合は、両方のオペランドが定数であることが必要 最初のオペランドが 2 の場合は、2 番目のオペランドに変数を使用可能 実数データ型はサポートされず、結果が実数となるようなオペランドの組み合わせを使用するとエラーが発生する X (不明) および Z (ハイ インピーダンス) は使用不可
分周	/	サポートあり 符号付きの定数と符号なしの定数の間で除算を指定すると、不正なロジックが生成される 例 : -1235/3'b111
関係演算	>, <, >=, <=	サポートあり
論理否定	!	サポートあり
論理 AND	&&	サポートあり
論理 OR		サポートあり
論理等号	==	サポートあり
論理不等号	!=	サポートあり
ケース等号	===	サポートあり
ケース不等号	!==	サポートあり
ビットごとの否定	~	サポートあり
ビットごとの AND	&	サポートあり
ビットごとの内包的 OR		サポートあり
ビットごとの排他的 OR	^	サポートあり
ビットごとの等価	~, ^^	サポートあり
リダクション AND	&	サポートあり
リダクション NAND	~&	サポートあり
リダクション OR		サポートあり
リダクション NOR	~	サポートあり
リダクション XOR	^	サポートあり
リダクション XNOR	~, ^^	サポートあり
左シフト	<<	サポートあり

論理式	シンボル	サポートの有無
符号付き右シフト	>>>	サポートあり
符号付き左シフト	<<<	サポートあり
右シフト	>>	サポートあり
条件演算	?:	サポートあり
イベント OR	or、'、'	サポートあり

Verilog のビヘイビア記述の論理式の評価結果

次の表に、XST で頻繁に使用される演算子を使用した論理式の評価結果を示します。

=== および != は、シミュレーションで変数に値 x または z が割り当てられているかを調べるのに便利な比較演算子です。合成中は、これらの演算子は == および != として処理されます。

a b	a==b	a===b	a!=b	a!==b	a&b	a&&b	a b	a b	a^b
0 0	1	1	0	0	0	0	0	0	0
0 1	0	0	1	1	0	0	1	1	1
0 x	x	0	x	1	0	0	x	x	x
0 z	x	0	x	1	0	0	x	x	x
1 0	0	0	1	1	0	0	1	1	1
1 1	1	1	0	0	1	1	1	1	0
1 x	x	0	x	1	x	x	1	1	x
1 z	x	0	x	1	x	x	1	1	x
x 0	x	0	x	1	0	0	x	x	x
x 1	x	0	x	1	x	x	1	1	x
x x	x	1	x	0	x	x	x	x	x
x z	x	0	x	1	x	x	x	x	x
z 0	x	0	x	1	0	0	x	x	x
z 1	x	0	x	1	x	x	1	1	x
z x	x	0	x	1	x	x	x	x	x
z z	x	1	x	0	x	x	x	x	x

Verilog ビヘイビア記述のブロック

ブロック文は、複数の文をグループ化します。

XST では、順次ブロックのみがサポートされています。ブロック内では、記述された順に文が実行されます。

ブロックは `begin` と `end` キーワードで示されます。これについては、この章の後の方で例を示します。

XST では、パラレル ブロックはサポートされません。

Verilog ビヘイビア記述のモジュール

Verilog では、デザイン コンポーネントはモジュールで表されます。コンポーネント間の接続は、モジュール インスタンスーション文で定義されます。モジュール インスタンス文は、モジュールのインスタンスを指定します。モジュール インスタンス文には、インスタンス名が含まれます。また、実際のネットまたはポートを接続するモジュール宣言のローカル ポート(フォーマル)を指定する接続リストも含まれます。

ブロック内の手続き文は、すべてモジュール内で定義します。手続き型ブロックには次の 2 種類があります。

- ・ initial ブロック
- ・ always ブロック

各ブロックは `begin` で開始し `end` で終了します。initial ブロックは合成では無視されるので、ここでは always ブロックのみを説明します。always ブロックは通常、次のフォーマットで記述されます。

```
always
begin
    statement
    ....
end
```

各文は手続き代入文であり、セミコロンで区切られます。

Verilog ビヘイビア記述のモジュール宣言

回路の I/O ポートはモジュール宣言文で宣言します。各ポートには、次が指定されます。

- ・ 名前
- ・ モード
 - **in**
 - **out**
 - **inout**

次のコード例で EXAMPLE というモジュールで宣言されている入力ポートおよび出力ポートは、デザインの基本的な入力および出力 I/O 信号です。Verilog の inout ポートはデバイス上の双方向 I/O ピンに対応し、データフローの方向はトライステートバッファへのイネーブル信号で制御されます。

次のコード例では、E はアクティブ High の出力イネーブル信号付きトライステートバッファとして記述されています。

- ・ `oe = 1` の場合は、信号 A の値は E の出力値になります。
- ・ `oe = 0` の場合はバッファはハイインピーダンス (z) になり、外部ロジックから E に入力された値はデバイスに取り込まれ、信号 D に送信されます。

コード例

```
module EXAMPLE (A, B, C, D, E);  
    input A, B, C;  
    output D;  
    inout E;  
    wire D, E;  
    ...  
    assign E = oe ? A : 1'bz;  
    assign D = B & E;  
    ...  
endmodule
```

Verilog ビヘイビア記述の継続代入文

継続代入文は、組み合わせロジックを簡潔に記述するために使用します。

XST では、明示的および暗示的両方の継続代入文がサポートされます。

- ・ 明示的な継続代入文では、別々に宣言されたネットの後に `assign` キーワードを挿入します。
- ・ 暗示的な継続代入文では、宣言と代入を組み合わせます。
- ・ XST では、継続代入文で指定した遅延や電流は無視されます。
- ・ 継続代入文は、`wire` および `tri` データ型のみに使用可能です。

明示的な継続代入文の例

```
wire par_eq_1;  
....  
assign par_eq_1 = select ? b : a;
```

暗示的な継続代入文の例

```
wire temp_hold = a | b;
```

Verilog ビヘイビア記述の手続き代入文

このセクションでは、Verilog ビヘイビア記述の手続き代入文について説明します。

- ・ [Verilog ビヘイビア記述の手続き代入文の概要](#)
- ・ [Verilog ビヘイビア記述の組み合わせ always ブロック](#)
- ・ [Verilog ビヘイビア記述の if-else 文](#)
- ・ [Verilog ビヘイビア記述の case 文](#)
- ・ [Verilog ビヘイビア記述の for および repeat ループ文](#)
- ・ [Verilog ビヘイビア記述の While ループ](#)
- ・ [Verilog ビヘイビア記述の順次 always ブロック](#)
- ・ [Verilog ビヘイビア記述の assign 文および deassign 文](#)
- ・ [Verilog ビヘイビア記述の 32 ビットを超える場合のビットの拡張](#)
- ・ [Verilog ビヘイビア記述のタスクおよび関数](#)
- ・ [Verilog ビヘイビア記述の再帰タスクおよび関数](#)
- ・ [Verilog ビヘイビア記述の定数関数](#)
- ・ [Verilog ビヘイビア記述のブロックおよびノンブロッキング手続き代入文](#)

Verilog ビヘイビア記述の手続き代入文の概要

Verilog ビヘイビア記述の手続き代入文

- ・ reg として宣言された変数に値を代入するために使用されます。
- ・ always ブロック、タスク、関数で最初に使用されます。
- ・ 通常はレジスタおよび有限ステート マシン (FSM) を記述するために使用されます。

XST では、次がサポートされます。

- ・ 組み合わせ関数
- ・ 組み合わせタスクおよび順次タスク
- ・ 組み合わせブロックおよび順次 always ブロック

Verilog ビヘイビア記述の組み合わせ always ブロック

組み合わせロジックは、次の Verilog のタイミング制御文を使用して効率的に記述できます。

- ・ # (シャープ記号)
- ・ * (アスタリスク)

合成では # によるタイミング制御は無視されるため、ここでは * (アスタリスク) 文を使用した組み合わせロジックの記述を説明します。

組み合わせ always 文には、always の後にかっこで囲まれたセンシティビティリストがあります。センシティビティリストにある信号の 1 つでイベント (値の変化またはエッジ) が発生すると、always ブロックの処理が実行されます。このセンシティビティリストには、条件 (if、case など) となり得る信号、および代入文の右側に記述される信号を含むことができます。かっこで囲んだ信号のリストなしで * (アスタリスク) を使用すると、上記のような always ブロックの信号でイベントが発生した場合に、always ブロックの処理が実行されます。

組み合わせプロセス文では、if 文または case 文のすべての分岐で信号が明示的に代入されていない場合、最後の値を保持するためにラッチが作成されます。ラッチが生成されないようにするには、組み合わせプロセスで代入された信号がそのプロセス文のすべての条件に対して明示的に代入されるようにしてください。

プロセス文には、次の文を含めることができます。

- ・ 変数代入文および信号代入文
- ・ if-else 文
- ・ case 文
- ・ for および while ループ文
- ・ 関数およびタスクの呼び出し

Verilog ビヘイビア記述の if – else 文

if – else 文では、真偽条件によって実行される文が決定されます。

- ・ 条件が真と判断された場合は if 文が実行されます。
- ・ 条件が偽 (または X か Z) と判断された場合は else 文が実行されます。

キーワード begin と end を使用すると、複数文から成り立つブロックを実行できます。

if – else 文はネストさせることができます。

コード例

次に、if – else 文を使用してマルチプレクサ (MUX) を記述した例を示します。

```
module mux4 (sel, a, b, c, d, outmux);
input [1:0] sel;
input [1:0] a, b, c, d;
output [1:0] outmux;
reg [1:0] outmux;

always @(sel or a or b or c or d)
begin
    if (sel[1])
        if (sel[0])
            outmux = d;
        else
            outmux = c;
    else
        if (sel[0])
            outmux = b;
        else
            outmux = a;
    end
endmodule
```

Verilog ビヘイビア記述の case 文

case 文

- ・ 論理式を比較し、並列分岐の 1 つを実行します。
- ・ 記述された順に分岐を評価します。
 - 最初に **true** になった分岐から実行されます。
 - 一致する分岐が見つからない場合は、デフォルトの分岐が実行されます。

case 文でサイズを指定していない整数を使用すると、予測不可能な結果になります。必ず整数のサイズをビット数で指定してください。

casez は、分岐のすべてのビット位置の z 値をドントケアとして認識します。

casez は、分岐のすべてのビット位置の x と z の値をドントケアとして認識します。

casez または casex などの case 文では、疑問符 (?) もドントケアとして使用できます。

コード例

次に、case 文を使用してマルチプレクサを記述した例を示します。

```
module mux4 (sel, a, b, c, d, outmux);
input [1:0] sel;
input [1:0] a, b, c, d;
output [1:0] outmux;
reg [1:0] outmux;
```

```
always @(sel or a or b or c or d)
begin
  case (sel)
    2'b00: outmux = a;
    2'b01: outmux = b;
    2'b10: outmux = c;
    default: outmux = d;
  endcase
end
endmodule
```

この case 文では、入力 sel の値が記述された優先順に評価されます。優先順に評価されるのを防ぐには、次の例に示すように parallel_case という Verilog 属性を使用して、sel 入力が並列に評価されるようにします。

```
(* parallel_case *) case(sel)
```

Verilog ビヘイビア記述の for および repeat ループ文

always ブロックでは、繰り返すまたはビット スライス構造を記述するのに for 文または repeat 文も使用できます。

for 文

for 文では、次のエレメントがサポートされます。

- ・ 定数の範囲
- ・ 次の演算子を使用したテスト条件の停止
 - <
 - <=
 - >
 - >=
- ・ 次のいずれかに適合する次ステップの計算
 - `var = var + step`
 - `var = var - step`

説明 :

- ◆ `var` はループ変数
- ◆ `step` は定数値

repeat 文

repeat 文では定数値しか使用できません。

disable 文

disable 文はサポートされていません。

コード例

```
module countzeros (a, Count);
input [7:0] a;
output [2:0] Count;
reg [2:0] Count;
reg [2:0] Count_Aux;
integer i;

always @(a)
begin
    Count_Aux = 3'b0;
    for (i = 0; i < 8; i = i+1)
    begin
        if (!a[i])
            Count_Aux = Count_Aux+1;
        end
    Count = Count_Aux;
end

endmodule
```

Verilog ビヘイビア記述の While ループ

always ブロックでは、while 文を使用して繰り返し処理を実行できます。while 文は、テスト式が偽 (false) になるまで、含まれる文を実行します。テスト式が始めから false の場合は実行されません。

- ・ 有効な Verilog の論理式であれば、どれでもテスト式として使用できます。
- ・ ループが永久に実行されるのを防ぐには、-loop_iteration_limit オプションを使用します。
- ・ while ループ文には disable 文を含めることができます。disable 文の構文は disable <blockname> なので、ラベルが付いているブロック内に使用する必要があります。

コード例

```
parameter P = 4;
always @(ID_complete)
begin : UNIDENTIFIED
    integer i;
    reg found;
    unidentified = 0;
    i = 0;
    found = 0;
    while (!found && (i < P))
    begin
        found = !ID_complete[i];
        unidentified[i] = !ID_complete[i];
        i = i + 1;
    end
end
end
```

Verilog ビヘイビア記述の順次 always ブロック

順序回路は、センシティブティリストと共に always ブロックで記述します。センシティブティリストには、最大で次の 3 つのイベントが含まれます。

- ・ クロック信号イベント (必須)
- ・ リセット信号イベント (含まれる可能性あり)
- ・ セット信号イベント

こういった場合の always ブロックでは、if - else 文は 1 度しか使用できません。

非同期部分は、if - else 文の 1 番目および 2 番目の分岐で同期部分の前に記述できます。この場合、非同期部分で指定される信号には、次の定数値が代入されます。

- ・ 0
- ・ 1
- ・ X
- ・ Z
- ・ これらを組み合わせたベクタ

これらの信号には、同期部分 (if...else 文の最後の分岐) でも値を代入されます。クロック信号の状態は、if...else 文の最後の分岐の状態になります。

always ブロックを使用した 8 ビット レジスタのコード例

```
module seq1 (DI, CLK, DO);
    input [7:0] DI;
    input CLK;
    output [7:0] DO;
    reg [7:0] DO;

    always @(posedge CLK)
        DO <= DI ;
```

always ブロックを使用した非同期リセット (アクティブ High) 付き 8 ビット レジスタのコード例

```
module EXAMPLE (DI, CLK, RST, DO);
    input [7:0] DI;
    input CLK, RST;
    output [7:0] DO;
    reg [7:0] DO;

    always @(posedge CLK or posedge RST)
        if (RST == 1'b1)
            DO <= 8'b00000000;
        else
            DO <= DI;
endmodule
```

always ブロックを使用した非同期リセット付き 8 ビット カウンタのコード例

```
module seq2 (CLK, RST, DO);
    input CLK, RST;
    output [7:0] DO;
    reg [7:0] DO;

    always @(posedge CLK or posedge RST)
        if (RST == 1'b1)
            DO <= 8'b00000000;
        else
            DO <= DO + 8'b00000001;
endmodule
```

Verilog ビヘイビア記述の assign 文および deassign 文

assign 文および deassign 文は、単純なテンプレート内でサポートされています。

Verilog ビヘイビア記述の assign 文および deassign 文のテンプレート

```
module assign (RST, SELECT, STATE, CLOCK, DATA_IN);
    input RST;
    input SELECT;
    input CLOCK;
    input [0:3] DATA_IN;
    output [0:3] STATE;
    reg [0:3] STATE;

    always @ (RST)
        if (RST)
            begin
                assign STATE = 4'b0;
            end
        else
            begin
                deassign STATE;
            end

    always @ (posedge CLOCK)
        begin
            STATE <= DATA_IN;
        end
endmodule
```

XST では、assign/deassign 文のサポートに次のような制限があります。

- ・ 1 つの信号に対しては、1 つの assign/deassign 文しか使用できません。
- ・ assign/deassign 文は、always ブロック内で if - else 文を使用して記述する必要があります。
- ・ assign/deassign 文で、信号の 1 つのビットまたは部分的ビット選択を指定することはできません。

Verilog ビヘイビア記述の assign 文および deassign 文

1 つの信号に対しては、1 つの assign/deassign 文しか使用できません。たとえば、次のようなデザインは XST ではサポートされません。

```
module dflop (RST, SET, STATE, CLOCK, DATA_IN);
    input RST;
    input SET;
    input CLOCK;
    input DATA_IN;
    output STATE;
    reg STATE;

    always @ (RST)                // block b1
        if (RST)
            assign STATE = 1'b0;
        else
            deassign STATE;

    always @ (SET)                // block b1
        if (SET)
            assign STATE = 1'b1;
        else
            deassign STATE;

    always @ (posedge CLOCK)      // block b2
        begin
            STATE <= DATA_IN;
        end
endmodule
```

同じ always ブロックで実行される assign/deassign 文のコード例

assign/deassign 文は、always ブロック内で if - else 文を使用して記述する必要があります。たとえば、次のようなデザインは XST ではサポートされません。

```
module dflop (RST, SET, STATE, CLOCK, DATA_IN);
    input RST;
    input SET;
    input CLOCK;
    input DATA_IN;
    output STATE;

    reg STATE;

    always @ (RST or SET)          // block b1
    case ({RST,SET})
        2'b00: assign STATE = 1'b0;
        2'b01: assign STATE = 1'b0;
        2'b10: assign STATE = 1'b1;
        2'b11: deassign STATE;
    endcase

    always @ (posedge CLOCK)       // block b2
    begin
        STATE <= DATA_IN;
    end
endmodule
```

assign/deassign 文で信号のビット/部分的ビット選択ができない場合のコード例

assign/deassign 文で、信号の 1 つのビットまたは部分的ビット選択を指定することはできません。たとえば、次のようなデザインは XST ではサポートされません。

```
module assign (RST, SELECT, STATE, CLOCK, DATA_IN);
    input RST;
    input SELECT;
    input CLOCK;
    input [0:7] DATA_IN;
    output [0:7] STATE;

    reg [0:7] STATE;

    always @ (RST)                // block b1
    if (RST)
    begin
        assign STATE[0:7] = 8'b0;
    end
    else
    begin
        deassign STATE[0:7];
    end

    always @ (posedge CLOCK)      // block b2
    begin
        if (SELECT)
            STATE [0:3] <= DATA_IN[0:3];
        else
            STATE [4:7] <= DATA_IN[4:7];
    end
end
```

Verilog ビヘイビア記述の 32 ビットを超える場合のビットの拡張

代入文の左側のビット幅が右側よりも大きい場合は、次のルールに従って、左側のビット幅が左にパディングされます。

- ・ 右側が符号付きの場合は、左側が次の符号付きビットでパディングされます。
 - 正の場合は 0 でパディングされます。
 - 負の場合は 1 でパディングされます。
 - ハイ インピーダンスの場合は z でパディングされます。
 - 不明の場合は x でパディングされます。
- ・ 右側が符号なしの場合は、左側が 0 でパディングされます。
- ・ ビット指定のない x または z 定数の場合は、次の規則に従います。右側の最上位ビットが z (ハイ インピーダンス) または x (不明) の場合、右側が符号付きまたは符号なしにかかわらず、左側にその値 (z または x) が追加されます。

上記の規則は、Verilog-2001 標準に従っています。これらは、Verilog-1995 との互換性はありません。

Verilog ビヘイビア記述のタスクおよび関数

関数およびタスク宣言は、デザインでブロックを複数回使用する場合に有益です。関数およびタスクは、モジュール内で宣言して使用します。関数のヘッダには入力パラメータのみが、タスクのヘッダには入力、出力、入出力のパラメータが含まれます。関数の戻り値は、符号付きまたは符号なしで宣言できます。内容は組み合わせ always ブロック文に類似しています。

コード例 1

次に、関数をモジュール内で宣言するコード例を示します。

- ・ ここで宣言されている ADD 関数は 1 ビット加算器です。
- ・ この関数はアーキテクチャ内のパラメータで 4 回呼び出され、4 ビット加算器を作成します。

```
module comb15 (A, B, CIN, S, COUT);
    input [3:0] A, B;
    input CIN;
    output [3:0] S;
    output COUT;
    wire [1:0] S0, S1, S2, S3;
    function signed [1:0] ADD;
        input A, B, CIN;
        reg S, COUT;
        begin
            S = A ^ B ^ CIN;
            COUT = (A&B) | (A&CIN) | (B&CIN);
            ADD = {COUT, S};
        end
    endfunction

    assign S0 = ADD (A[0], B[0], CIN),
           S1 = ADD (A[1], B[1], S0[1]),
           S2 = ADD (A[2], B[2], S1[1]),
           S3 = ADD (A[3], B[3], S2[1]),
           S = {S3[0], S2[0], S1[0], S0[0]},

           COUT = S3[1];
endmodule
```

コード例 2

次は、コード例 1 をタスクを使用して記述した例です。

```
module EXAMPLE (A, B, CIN, S, COUT);
    input [3:0] A, B;
    input CIN;
    output [3:0] S;
    output COUT;
    reg [3:0] S;
    reg COUT;
    reg [1:0] S0, S1, S2, S3;

    task ADD;
        input A, B, CIN;
        output [1:0] C;
        reg [1:0] C;
        reg S, COUT;

        begin
            S = A ^ B ^ CIN;
            COUT = (A&B) | (A&CIN) | (B&CIN);
            C = {COUT, S};
        end
    endtask

    always @(A or B or CIN)
    begin
        ADD (A[0], B[0], CIN, S0);
        ADD (A[1], B[1], S0[1], S1);
        ADD (A[2], B[2], S1[1], S2);
        ADD (A[3], B[3], S2[1], S3);
        S = {S3[0], S2[0], S1[0], S0[0]};
        COUT = S3[1];
    end
endmodule
```

Verilog ビヘイビア記述の再帰タスクおよび関数

Verilog-2001 では、再帰タスクおよび関数がサポートされます。

再帰は、automatic キーワードだけで指定できます。

再帰呼び出しが永久に実行されるのを防ぐために、繰り返す回数は 64 (デフォルト) に制限されています。回数を変更するには、-recursion_iteration_limit オプションを使用します。

コード例

```
function automatic [31:0] fac;
input [15:0] n;
if (n == 1)
    fac = 1;
else
    fac = n * fac(n-1); //recursive function call
endfunction
```

Verilog ビヘイビア記述の定数関数

Verilog-2001 では、定数関数のサポートが追加されています。XST では、定数値を計算する関数呼び出しがサポートされます。

コード例

```
module rams_cf (clk, we, a, di, do);
parameter DEPTH=1024;
input clk;
input we;
input [4:0] a;
input [3:0] di;
output [3:0] do;

reg [3:0] ram [size(DEPTH):0];

always @(posedge clk) begin
if (we)
ram[a] <= di;
end
assign do = ram[a];

function integer size;
input depth;
integer i;
begin
size=1;
for (i=0; 2**i<depth; i=i+1)
size=i+1;
end
endfunction

endmodule
```

Verilog ビヘイビア記述のブロックおよびノンブロッキング手続き代入文

および @ タイミング制御文を使用すると、指定されたイベントが発生するまでその後に続く文は実行されません。ブロッキングおよびノンブロッキング手続き代入文には、タイミングを制御する要素が組み込まれています。合成では、# の遅延は無視されます。

ブロッキング手続き代入文の構文例

次に、ブロッキング手続き代入文の構文例を示します。

```
reg a; a = #10 (b | c);
```

または

```
if (in1) out = 1'b0; else out = in2;
```

このタイプの代入文では、文が 1 つずつ順に実行され、プロセスに含まれる別の文が同時に実行されることはありません。これは、主にシミュレーションで使用します。

ノンブロッキング代入文では、文が実行されるときに式が評価されますが、同じプロセスに含まれるほかの文も同時に実行されます。変数は、指定された遅延後に変更されます。

ノンブロッキング手続き代入文の構文例

次に、ノンブロッキング手続き代入文の使用例を示します。

```
variable <= @(posedge_or_negedge_bit) expression;
```

コード例

次に、ノンブロッキング手続き代入文の使用例を示します。

```
if (in1) out <= 1'b1; else out <= in2;
```

Verilog ビヘイビア記述の定数

デフォルトでは、Verilog の定数は 10 進数の整数であると認識されますが、適切な接頭辞を使用して 2 進、8 進、10 進、16 進に指定できます。たとえば、次はすべて同じ値を表します。

- ・ 4'b1010
- ・ 4'o12
- ・ 4'd10
- ・ 4'ha

Verilog ビヘイビア記述のマクロ

Verilog では、次の例に示すようにマクロを定義できます。

```
'define TESTEQ1 4'b1101
```

定義されたマクロは、この後のデザイン コードで次のように参照します。

```
if (request == 'TESTEQ1)
```

これは次のコード例のようになります。

```
'define myzero 0  
assign mysig = 'myzero;
```

Verilog では、マクロが定義されているかどうかを判断する 'ifdef および 'endif も使用できます。これらの構文は、条件付きコンパイルを定義するために使用します。'ifdef コマンドで呼び出されたマクロが定義されている場合、そのコードはコンパイルされますが、定義されていない場合は、'else コマンドに続くコードがコンパイルされます。'else は必須ではありませんが、条件文の最後に 'endif を付ける必要があります。

次に 'ifdef と 'endif の使用例を示します。

```
'ifdef MYVAR  
module if_MYVAR_is_declared;  
...  
endmodule  
'else  
module if_MYVAR_is_not_declared;  
...  
endmodule  
'endif
```

Verilog マクロ (-define) を使用すると、Verilog マクロを定義または再定義できるので、これにより、IP コアの生成やテストベンチ フローなどの HDL ソースを変更しなくてもデザイン コンフィギュレーションを簡単に修正できます。

Verilog ビヘイビア記述の include ファイル

Verilog では、ソースコードを複数のファイルに分割できます。別のファイルに含まれるコードを参照するには、次の構文を使用します。

```
'include "path/file-to-be-included"
```

相対パスまたは絶対パスのどちらでも使用できます。

1 つの Verilog ファイルに複数の 'include 文を含めることができます。こうすることで、複数ファイルでデザインに含まれる複数モジュールを記述するようなチーム デザイン環境で、コードが管理しやすくなります。

ディレクトリの認識

'include 文で指定したファイルを認識させるには、ISE® Design Suite または XST にこのファイルのディレクトリを認識させる必要があります。

- ・ ISE Design Suite はデフォルトでプロジェクト ディレクトリを検索するので、ファイルをプロジェクト ディレクトリに追加すると ISE Design Suite で認識されるようになります。
- ・ 相対パスまたは絶対パスをソースコードの 'include 文に含めることで別のディレクトリを ISE Design Suite に認識させることができます。
- ・ XST が直接 include ファイル ディレクトリをポイントするようにするには、[Verilog インクルード ディレクトリ \(-vlgincdir\)](#) を使用します。
- ・ デザイン階層を構築するのに 'include ファイルが必要とされる場合、ファイルをプロジェクトに追加する必要はありませんが、次のいずれかにの方法で指定しておく必要があります。
 - プロジェクト ディレクトリに含める
 - または
 - 相対パスまたは絶対パスで参照する

include ファイルの競合

指定したファイルが次のような場合は、競合が発生する可能性があります。

- ・ ISE Design Suite プロジェクトへ追加された場合
および
- ・ 'include を使用して指定されている場合

コード例

```
`timescale 1 ns/1 ps
`include "modules.v"
...
```

この場合、XST で次のようなエラー メッセージが表示されます。

```
ERROR:Xst:1068 - fifo.v, line 2. Duplicate declarations of
module' RAMB4_S8_S8'
```

Verilog ビヘイビア記述のコメント

Verilog ビヘイビア記述では、次の表に示すように 2 種類のコメント形式がサポートされます。
Verilog ビヘイビア記述のコメントは、C++ などの言語で使用されるのと同じ形式になります。

シンボル	説明	使用目的	例
//	ダブル フォワード スラッシュ	1 行のコメント	// Define a one-line comment as illustrated by this sentence
/*	スラッシュとアスタ リスク	複数行のコメント	/* Define a multi-line comment by enclosing it as illustrated by this sentence */

Verilog ビヘイビア記述の generate 文

generate 文は、条件文からダイナミックに Verilog コードを作成するための構文です。この文を使用すると、繰り返し構造や、特定の条件の下でのみ適切な構造を作成できます。

generate 文では、次のような構造が作成できます。

- ・ プリミティブまたはモジュールのインスタンス
- ・ initial または always 手続きブロック
- ・ 継続代入文
- ・ ネットおよび変数の宣言
- ・ パラメータの再定義
- ・ タスクまたは関数の定義

generate-for 文

Verilog ビヘイビア記述の generate-for ループ文を使用すると、モジュール内に 1 つ以上のインスタンスが作成されます。generate-for ループ文は for ループ文と同様に使用できますが、次のような制限があります。

- ・ generate-for ループ文のインデックスには、genvar 変数を使用する必要があります。
- ・ for ループ制御内の代入は、genvar 変数を参照する必要があります。
- ・ for ループ文の内容は begin 文と end 文で囲み、begin 文には固有の修飾子が付いた名前を使用します。

コード例

次は、generate-for ループ文を使用した 8 ビット加算器の Verilog ビヘイビア記述のコード例です。

```
generate
genvar i;

    for (i=0; i<=7; i=i+1)
        begin : for_name
            adder add (a[8*i+7 : 8*i], b[8*i+7 : 8*i],          ci[i], sum_for[8*i+7 : 8*i], c0_or[i+1]);        end
        endgenerate
```

generate if-else 文

Verilog ビヘイビア記述の generate-if-else 文は、条件を使用して生成するオブジェクトを制御する際に使用できます。

コード例

次は、generate if... else 文の例です。

- ・ generate では、インスタンス化される乗算器のタイプが制御されます。
- ・ if - else 文の各分岐の内容は begin 文と end 文で囲みます。
- ・ begin 文には固有の修飾子が付いた名前を使用します。

```
generate
  if (IF_WIDTH < 10)
    begin : if_name
      adder # (IF_WIDTH) u1 (a, b, sum_if);
    end
  else
    begin : else_name
      subtractor # (IF_WIDTH) u2 (a, b, sum_if);
    end
  end
endgenerate
```

generate-case 文

Verilog ビヘイビア記述の generate case 文は、条件を使用して生成するオブジェクトを制御する際に使用できます。テストする条件が複数ある場合に、generate-case 文を使用して生成されるコードを決定します。

- ・ generate-case 文の各分岐の内容は begin 文と end 文で囲みます。
- ・ begin 文には固有の修飾子が付いた名前を使用します。

コード例

次に、generate-case 文の使用例を示します。generate では、インスタンス化される加算器のタイプが制御されます。

```
generate
  case (WIDTH)
    1:
      begin : case1_name
        adder # (WIDTH*8) x1 (a, b, ci, sum_case, c0_case);
      end
    2:
      begin : case2_name
        adder # (WIDTH*4) x2 (a, b, ci, sum_case, c0_case);
      end
    default:
      begin : d_case_name
        adder x3 (a, b, ci, sum_case, c0_case);
      end
  endcase
endgenerate
```

混合言語のサポート

この章では、XST の混合言語サポートについて説明します。

- ・ [混合言語サポートの概要](#)
- ・ [混合言語のプロジェクト ファイル](#)
- ・ [混合言語プロジェクトのVHDL/Verilog の境界規則](#)
- ・ [混合言語プロジェクトのポート マップ](#)
- ・ [混合言語プロジェクトのジェネリック サポート](#)
- ・ [混合言語プロジェクトの LSO ファイル](#)

混合言語サポートの概要

XST では、VHDL と Verilog の混合言語プロジェクトがサポートされます。

- ・ VHDL と Verilog の混合は、デザイン ユニット (セル) のインスタンス化のみに制限されています。
 - VHDL デザインには Verilog モジュールをインスタンス化でき、
 - Verilog デザインには VHDL エンティティをインスタンス化できます。
 - それ以外の方法で VHDL と Verilog は混合不可能
- ・ VHDL デザインでは、VHDL タイプ、ジェネリック、およびポートの制限されたサブセットを Verilog モジュールとの境界に使用できます。
- ・ Verilog デザインでは、Verilog タイプ、ジェネリック、およびポートの制限されたサブセットを VHDL モジュールまたはコンフィギュレーションとの境界に使用できます。
- ・ XST では、エラボレーション段階で VHDL デザイン ユニットが Verilog モジュールにバインドされます。
- ・ Verilog モジュールを VHDL デザイン ユニットにバインドする際は、デフォルトのバインド方法に基づくコンポーネント インスタンス化が使用されます。
- ・ Verilog モジュールを VHDL にインスタンス化する場合、コンフィギュレーションの設定、直接のインスタンス化、およびコンポーネント コンフィギュレーションはサポートされません。
- ・ VHDL および Verilog プロジェクト ファイルは 1 つに統一されます。
- ・ VHDL および Verilog ライブラリが論理的に統一されます。
- ・ VHDL のみで可能だったコンパイル用の作業ディレクトリ (xsthdmdir) の指定が Verilog でも可能になります。
- ・ VHDL のみで可能だった論理ライブラリ名をホスト ファイル システムの物理ディレクトリ名にマップする xhdp.ini のメカニズムが、Verilog でも可能になります。

- ・ デザイン ユニット (セル) を統一された論理ライブラリで検索するための検索順を指定できます。エラボレーションの段階でこの検索順に従って、VHDL エンティティまたは Verilog モジュールが検索され、混合言語プロジェクトにバインドされます。

混合言語のプロジェクト ファイル

XST では、VHDL/Verilog 混合デザインをサポートするため、専用の混合言語プロジェクトファイルを使用します。この混合言語フォーマットは、混合言語プロジェクトだけではなく、VHDL のみまたは Verilog のみのプロジェクトにも使用できます。

- ・ ISE® Design Suite から XST を実行する場合は、ISE でプロジェクト ファイルが作成されます。このプロジェクト ファイルは、常に混合言語です。
- ・ コマンドラインから XST を起動する場合は、混合言語プロジェクト用にユーザーがプロジェクト ファイルを作成する必要があります。

プロジェクト タイプ	-ifmt の設定
コマンドライン	<i>mixed</i> または値を削除
VHDL	<i>vhdl</i>
Verilog	<i>verilog</i>

既存のデザインに関しては、VHDL および Verilog 形式をそのまま使用できます。

混合言語プロジェクトでライブラリなど外部ファイルを呼び出す場合は、次の構文を使用します。

language library file_name.ext

コード例

次に、混合言語プロジェクトでライブラリを呼び出す例を示します。

```
vhdl      work      my_vhdl1.vhd
verilog   work      my_vlg1.v
vhdl      my_vhdl_lib my_vhdl2.vhd
verilog   my_vlg_lib my_vlg2.v
```

- ・ 1 つの行で 1 つの HDL デザイン ファイルを指定します。
- ・ 各列には、次の表のような意味があります。

列	構文	例	指定
1 列目	language	vhdl	HDL ファイルは VHDL または Verilog のいずれか
2 列目	library	work	HDL がコンパイルされる論理ライブラリ。デフォルトの論理ライブラリは work
3 列目	file_name.ext	my_vhdl1.vhd	HDL ファイル名

混合言語プロジェクトのVHDL/Verilog の境界規則

VHDL と Verilog の境界は、デザイン ユニットのレベルにより決定します。

VHDL デザインには Verilog モジュールをインスタンスエートでき、

Verilog デザインには VHDL エンティティをインスタンスエートできます。

VHDL デザインへの Verilog モジュールのインスタンスエート

VHDL デザインに Verilog モジュールをインスタンスエートするには、次の手順に従います。

1. インスタンスエートする Verilog モジュールと同じ名前の VHDL コンポーネントを宣言します。Verilog モジュール名がすべて小文字でない場合は、次のいずれかの方法で case プロパティを使用し、大文字/小文字を保持するように設定します。
 - a. ISE® Design Suite で次から設定します。
[Synthesize - XST] プロセスの [Process Properties] ダイアログ ボックス → [Synthesis Options] → [Case] → [Maintain]
または
 - b. コマンドラインで `-case` オプションを `maintain` に設定
2. VHDL コンポーネントをインスタンスエートするのと同様に、Verilog コンポーネントをインスタンスエートします。

VHDL コンフィギュレーション宣言を使用して、このコンポーネントを特定のライブラリからの特定のデザイン ユニットにバインドする方法はサポートされていません。サポートされるのは、デフォルト Verilog モジュールのバインドのみです。

VHDL デザインにインスタンスエートできる Verilog の構文は、Verilog モジュールのみです。その他の Verilog の構文は VHDL コードで認識されません。

エラボレーションの段階で、デフォルトのバインド処理が行われるすべてのコンポーネントは、対応するコンポーネントの名前と同じ名前のデザイン ユニットとして処理されます。バインド段階では、コンポーネント名は VHDL デザイン ユニット名として扱われ、`work` という論理ライブラリ内で検索されます。VHDL デザイン ユニットが見つかった場合、バインドされます。VHDL デザイン ユニットが見つからない場合は、コンポーネント名は Verilog モジュールとして扱われ、大文字と小文字を区別して検索が行われます。Verilog モジュールは、統一された論理ライブラリから指定したライブラリの検索順に検索されます。

詳細は、次を参照してください。

混合言語プロジェクトのライブラリ検索順 (LSO) ファイル

XST では、最初に一致した Verilog モジュールを選択してバインドします。

ライブラリが統一されているため、VHDL デザイン ユニットと同じ名前の Verilog セルは同じ論理ライブラリに共存させることはできません。同じ名前のセル/ユニットがコンパイルされると、以前にコンパイルされたものが上書きされます。

Verilog デザインへの VHDL デザイン ユニットのインスタンスシート

このセクションには、次の項目が含まれます。

- ・ [VHDL エンティティをインスタンスシートする方法](#)
- ・ [バインド処理](#)
- ・ [制限](#)

VHDL エンティティをインスタンスシートする方法

VHDL エンティティをインスタンスシートするには、次の手順に従ってください。

1. インスタンスシートする VHDL エンティティと同じモジュール名（アーキテクチャ名を付けたものも可）を宣言
2. 通常の Verilog インスタンスエーションを実行

Verilog デザインにインスタンスシートできる VHDL の構文は、VHDL エンティティのみです。その他の VHDL の構文は Verilog コードで認識されません。XST では、エンティティ/アーキテクチャ ペアが Verilog と VHDL の境界として使用されます。

バインド処理

XST では、バインド処理はエラボレーション段階で行われます。

XST では、まず次が実行されます。

1. 次のように Verilog モジュールを検索します。
 - ・ インスタンスシート済みモジュールの名前を使用します。
 - ・ 未定義の論理ライブラリのユーザー指定リストで検索します。
 - ・ ユーザー指定の順序で検索します。
 - ・ モジュール インスタンスエーション文で指定したアーキテクチャ名を無視します。
2. 名前が検出されたら、それをバインドします。

XST で Verilog モジュールが検出できなかった場合は、

1. インスタンスシート済みモジュールの名前を VHDL エンティティとして処理します。
2. 次のように VHDL エンティティを検索します。
 - ・ 大文字/小文字を区別した検索を実行します。
 - ・ 未定義の論理ライブラリのユーザー指定リストで検索します。
 - ・ ユーザー指定の順序で検索します。

メモ：これにより、VHDL デザイン ユニットは拡張指示子付きで格納されたと認識されます。

3. 最初に名前的一致した VHDL エンティティを選択します。
4. エンティティをバインドします。

詳細は、次を参照してください。

[混合言語プロジェクトのライブラリ検索順 \(LSO\) ファイル](#)

制限

Verilog モジュールから VHDL デザイン ユニートをインスタンス化する場合、XST では次のような制限があります。

- ・ ポートの関連付けは明示的に行う必要があります。ポート マップでは、必ず正式な有効ポート名を指定してください。
- ・ パラメータは、値が変化しない場合でも、インスタンス化時にすべて渡す必要があります。
- ・ パラメータを変更する場合は、どのパラメータかを指定する必要があります。順序は認識されません。この場合、defparam を使用するのではなくインスタンス化を使用してください。

パラメータ変更の正しいコード例

```
ff #(.init(2'b01)) ul (.sel(sel), .din(din), .dout(dout));
```

パラメータ変更の不正なコード例

次のコード例は使用できません。

```
ff ul (.sel(sel), .din(din), .dout(dout));  
defparam ul.init = 2'b01;
```

混合言語プロジェクトのポート マップ

混合言語プロジェクトのポート マップには、次が含まれます。

- ・ Verilog 内の VHDL ポート マップ
- ・ VHDL 内の Verilog ポート マップ
- ・ 混合言語内の VHDL ポート マップ
- ・ 混合言語の Verilog ポート マップ

Verilog 内の VHDL ポート マップ

Verilog デザインにインスタンス化された VHDL エンティティでは、次のポート タイプがサポートされます。

- ・ in
- ・ out
- ・ inout

XST では、VHDL の buffer と linkage ポートはサポートされません。

VHDL 内の Verilog ポート マップ

VHDL デザインにインスタンス化された Verilog モジュールでは、次のポート タイプがサポートされます。

- ・ 入力
- ・ 出力
- ・ 入出力

XST では、Verilog の双方向パス オプションはサポートされていません。

XST では、混合言語の境界に名前のない Verilog ポートを使用することはできません。

大文字と小文字が混合している Verilog モジュールのポート名を接続する場合は、コンポーネント宣言と同じようにしてください。デフォルトでは、Verilog ポート名はすべて小文字であると判断されます。

混合言語内の VHDL ポート マップ

XST では、混合言語デザインで次の VHDL データ型がサポートされます。

- ・ **bit**
- ・ **bit_vector**
- ・ **std_logic**
- ・ **std_ulogic**
- ・ **std_logic_vector**
- ・ **std_ulogic_vector**

混合言語の Verilog ポート マップ

XST では、混合言語デザインで次の Verilog データ型がサポートされます。

- ・ **wire**
- ・ **reg**

混合言語プロジェクトのジェネリック サポート

XST では、混合言語デザインで次の VHDL ジェネリック タイプがサポートされます。

- ・ **integer**
- ・ **real**
- ・ **string**
- ・ **boolean**

混合言語プロジェクトの LSO ファイル

ライブラリ検索順ファイル (Library Search Order (LSO)) では、VHDL/Verilog 混合デザインに対して XST で使用するライブラリの検索順が指定されます。デフォルトでは、プロジェクト ファイルに現れる順序でファイルが検索されます。

XST では、次の場合デフォルトの検索順が使用されます。

- ・ LSO ファイルに DEFAULT_SEARCH_ORDER キーワードが含まれる場合
- ・ LSO ファイルが指定されていない場合

ISE Design Suite での LSO ファイルの指定

ISE® Design Suite では、ライブラリ検索順 (LSO) ファイルのデフォルト名は `project_name.lso` です。`project_name.lso` ファイルが存在しない場合は、ISE Design Suite により自動的に作成されます。

ISE Design Suite で既存の `project_name.lso` ファイルが検出されると、そのファイルがそのまま使用されます。ISE Design Suite では、プロジェクト名が最上位レベルのブロックの名前になります。ISE Design Suite でデフォルトの LSO ファイルが作成されると、ファイルの最初の行に `DEFAULT_SEARCH_ORDER` キーワードが記述されます。

コマンドラインでの LSO ファイルの指定

コマンドラインから XST を使用する場合は、**ライブラリ検索順 (-lso)** オプションを使用して LSO ファイルを指定します。-lso オプションを使用しない場合は、LSO ファイルを使用せずに、デフォルトのライブラリ検索順が使用されます。

LSO 規則

XST では、混合言語プロジェクトを処理する際、Library Search Order (LSO) ファイルの内容別に次の検索順規則が使用されます。

- ・ LSO (ライブラリ検索順) が空の場合
- ・ `DEFAULT_SEARCH_ORDER` キーワードのみの場合
- ・ `DEFAULT_SEARCH_ORDER` キーワードとライブラリリストがある場合
- ・ ライブラリリストのみの場合
- ・ `DEFAULT_SEARCH_ORDER` キーワードがなく、存在しないライブラリ名が使用される場合

LSO (ライブラリ検索順) が空の場合

Library Search Order (LSO) ファイルが空の場合、XST では次が実行されます。

- ・ LSO ファイルが空であることを示す警告メッセージが表示されます。
- ・ デフォルトのライブラリ検索順を使用してプロジェクト ファイルで指定したファイルが検索されます。
- ・ プロジェクト ファイルに表示される順にライブラリリストが追加され、LSO ファイルがアップデートされます。

DEFAULT_SEARCH_ORDER キーワードのみの場合

Library Search Order (LSO) ファイルに **DEFAULT_SEARCH_ORDER** キーワードのみのが含まれる場合、XST では次が実行されます。

- ・ プロジェクト ファイルに現れる順序でライブラリ ファイルが検索されます。
- ・ LSO ファイルが次のようにアップデートされます。
 - `DEFAULT_SEARCH_ORDER` キーワードが削除される
 - プロジェクト ファイルに現れる順序で、ライブラリが LSO ファイルにリストされる

プロジェクト ファイル `my_proj.prj` には、次のような内容が含まれています。

```
vhdl vhlib1 f1.vhd verilog rtfllib f1.v vhdl vhlib2 f3.vhd LSO
file Created by ProjNav
```

LSO ファイル `my_proj.lso` の内容は、次のとおりです。

```
DEFAULT_SEARCH_ORDER
```

XST では、次の検索順が使用されます。

```
vhlib1 rtfllib vhlib2
```

プロセス後の my_proj.lso の内容は、次のとおりです。

```
vhlib1 rtfllib vhlib2
```

DEFAULT_SEARCH_ORDER キーワードとライブラリ リストがある場合

Library Search Order (LSO) ファイルに DEFAULT_SEARCH_ORDER キーワードとライブラリのリストが含まれる場合、XST では次が実行されます。

- ・ プロジェクト ファイルに現れる順序でライブラリ ファイルが検索されます。
- ・ LSO ファイルに含まれるライブラリのリストは無視されます。
- ・ LSO ファイルはアップデートされません。

プロジェクト ファイル my_proj.prj には、次のような内容が含まれています。

```
vhdl vhlib1 f1.vhd verilog rtfllib f1.v vhdl vhlib2 f3.vhd
```

LSO ファイル my_proj.lso の内容は、次のとおりです。

```
rtfllib vhlib2 vhlib1 DEFAULT_SEARCH_ORDER
```

XST では、次の検索順が使用されます。

```
vhlib1 rtfllib vhlib2
```

プロセス後の my_proj.lso の内容は、次のとおりです。

```
rtfllib vhlib2 vhlib1 DEFAULT_SEARCH_ORDER
```

ライブラリ リストのみの場合

LSO ファイルにライブラリのリストが含まれており、DEFAULT_SEARCH_ORDER キーワードがない場合、XST では次が実行されます。

- ・ LSO ファイルにリストされている順序でライブラリ ファイルが検索されます。
- ・ LSO ファイルはアップデートされません。

プロジェクト ファイル my_proj.prj には、次のような内容が含まれています。

```
vhdl vhlib1 f1.vhd verilog rtfllib f1.v vhdl vhlib2 f3.vhd
```

LSO ファイル my_proj.lso の内容は、次のとおりです。

```
rtfllib vhlib2 vhlib1
```

XST では、次の検索順が使用されます。

```
rtfllib vhlib2 vhlib1
```

プロセス後の my_proj.lso の内容は、次のとおりです。

```
rtfllib vhlib2 vhlib1
```

DEFAULT_SEARCH_ORDER キーワードがなく、存在しないライブラリ名が使用される場合

プロジェクトまたは INI ファイルに存在しないライブラリ名が Library Search Order (LSO) ファイルに含まれており、LSO ファイルに DEFAULT_SEARCH_ORDER キーワードが含まれていない場合、そのライブラリは無視されます。

プロジェクト ファイル my_proj.prj には、次のような内容が含まれています。

```
vhdl vhlib1 f1.vhd verilog rtfllib f1.v vhdl vhlib2 f3.vhd
```

LSO ファイル my_proj.lso の内容は、次のとおりです。

```
personal_lib rtfllib vhlib2 vhlib1
```

XST では、次の検索順が使用されます。

```
rtfllib vhlib2 vhlib1
```

プロセス後の my_proj.lso の内容は、次のとおりです。

```
rtfllib vhlib2 vhlib1
```


XST ログ ファイル

この章では、XST のログ ファイルについて次のセクションに分けて説明します。

- ・ [ログ ファイルの内容](#)
- ・ [ログ ファイルの容量削減方法](#)
- ・ [ログ ファイルのマクロ](#)
- ・ [ログ ファイルの例](#)

FPGA のログ ファイルの内容

XST で出力される FPGA のログ ファイルには、次の情報が含まれています。

- ・ [著作権情報](#)
- ・ [目次](#)
- ・ [合成オプションのサマリ](#)
- ・ [HDL のコンパイル](#)
- ・ [デザイン階層の解析ツール](#)
- ・ [HDL 解析](#)
- ・ [HDL 合成レポート](#)
- ・ [アドバンス HDL 合成レポート](#)
- ・ [下位レベルの合成](#)
- ・ [パーティション レポート](#)
- ・ [最終レポート](#)

著作権情報

著作権情報には、次が含まれます。

- ・ ISE® Design Suite のバージョン番号
- ・ ザイリンクスの著作権の表示

目次

FPGA のログ ファイルの目次には、そのログ ファイルの主なセクションがリストされます。目次にある項目は、リンクされていません。テキスト エディタの検索機能を使用して、ナビゲートしてください。

合成オプションのサマリ

合成オプションのサマリには、次に関する情報が含まれます。

- ・ ソース パラメータ
- ・ ターゲット パラメータ
- ・ ソース オプション
- ・ ターゲット オプション
- ・ 一般的なオプション
- ・ その他のオプション

HDL のコンパイル

HDL のコンパイルの詳細は、次を参照してください。

[FPGA ログ ファイルの HDL 解析](#)

デザイン階層の解析ツール

詳細は、次を参照してください。

[FPGA ログ ファイルの HDL 解析](#)

HDL 解析

HDL のコンパイル、デザイン階層解析、および HDL 解析中、XST では次が実行されます。

- ・ VHDL/Verilog ファイルの解析
- ・ デザイン階層の認識
- ・ コンパイルされるライブラリの命名

この段階では、XST では次がレポートされます。

- ・ 合成とシミュレーション結果間潜在的な矛盾点
- ・ 潜在的なマルチソースの状態
- ・ その他の問題

HDL 合成レポート

XST では HDL 合成中に基本的なマクロが可能な限り認識され、ターゲット アーキテクチャに合ったインプリメンテーションが作成されます。この作業はブロックごとに行われ、このステップの最終段階で HDL 合成レポートが出力されます。

マクロ処理および合成プロセス中に表示されるメッセージの詳細についてはを参照してください。

[HDL コーディング手法](#)

アドバンス HDL 合成レポート

XST では、次のような高度なマクロ認識およびマクロ推論が実行されます。この場合、XST は次のように動作します。

- ・ ダイナミック シフトレジスタなどの認識
- ・ パイプライン乗算器のインプリメンテーション
- ・ ステート マシンのコード記述

この部分には、デザイン全体で認識されたマクロのサマリがタイプ別に記述されます。

下位レベルの合成

XST の FPGA ログ ファイルの下位レベル合成段階には、次の潜在的な削除情報がレポートされます。

- ・ 等価フリップフロップ
- ・ レジスタの複製

詳細は、次を参照してください。

[FPGA 最適化のレポート セクション](#)

パーティション レポート

このセクションには、デザインがパーティション処理された場合の詳細なパーティション情報が含まれます。

最終レポート

最終レポート部分には、次の情報が含まれます。

- ・ 次の最終結果
 - RTL 最上位レベルの出力ファイル名 (例 stopwatch.ngf)
 - 最上位レベルの出力ファイル名 (例 : stopwatch)
 - 出力ファイル形式 (例 : NGC)
 - 最適化ゴール (例 : Speed)
 - 階層維持のための制約 (KEEP_HIERARCHY) の使用の有無 (例 : No)
- ・ セル使用率
BEL、クロック バッファ、I/O バッファなどのセル使用率レポート
- ・ デバイス使用率のサマリ
XST により予測されたスライス、フリップフロップ、IOB、BRAM などの数が示されます。このレポートは、MAP で生成されるレポートと類似しています。
- ・ パーティション リソース サマリ
XST により予測された各パーティションのスライス、フリップフロップ、IOB、BRAM などの数が示されます。このレポートは、MAP で生成されるレポートと類似しています。
- ・ タイミング レポート
XST では合成の最後に詳細なタイミング情報がレポートされます。タイミング レポートには、ネットリストの 4 つの使用可能なドメインに関する情報が表示されます。
 - レジスタからレジスタ
 - 入力からレジスタ
 - レジスタから出力パッド
 - 入力パッドから出力パッド次に例を示します。
[「FPGA ログ ファイルの例」](#)のタイミング レポート セクション
詳細は、次を参照してください。
[FPGA 最適化のレポート セクション](#)
- ・ 暗号化されたモジュール
ザインに暗号化されたモジュールが含まれる場合、これらのモジュール情報は非表示になります。

ログ ファイルの容量削減方法

XST のログ ファイルの容量削減方法

- ・ [メッセージ フィルタの使用](#)
- ・ [Quiet モードの使用](#)
- ・ [Silent モードの使用](#)
- ・ [特定メッセージの非表示](#)

メッセージ フィルタの使用

ISE® Design Suite から XST を実行する場合は、メッセージ フィルタ ツールを使用してログ ファイルから特定のメッセージのみを表示できます。

詳細は、次を参照してください。

ISE Design Suite ヘルプの「メッセージ フィルタの使用」

Quiet モードの使用

Quiet モードを使用すると、コンピュータの画面 (stdout) に表示されるメッセージの量を制限できます。

このモードを設定するには、`-intstyle` オプションを次のいずれかにします。

- ・ **ise**
ISE® Design Suite 用にメッセージがフォーマットされます。
- ・ **xflow**
XFLOW 用にメッセージがフォーマットされます。

通常は、画面にすべてのログが出力されます。Quiet モードを使用すると出力されなくなる情報は、次のとおりです。

- ・ 著作権情報
- ・ 目次
- ・ 合成オプションのサマリ
- ・ 次に示す最終レポートのセクション
 - CPLD デバイスの最終結果ヘッダ
 - FPGA デバイスの最終結果セクション
 - タイミング数値が合成における概算にすぎないことを示すタイミング レポートの注記
 - タイミングの詳細
 - CPU (XST ランタイム)
 - メモリ使用率

FPGA デバイスの場合、このオプションを使用しても次のセクションは表示されます。

- ・ デバイス使用率のサマリ
- ・ クロック情報
- ・ タイミング サマリ

Silent モードの使用

Silent モードを使用すると、コンピュータ画面 (stdout) にはメッセージはまったく表示されず、ログ ファイルにのみメッセージが記述されます。

このモードを使用するには、`-intstyle` オプションを次に設定します。

silent

特定メッセージの非表示

このセクションには、次の項目が含まれます。

- ・ [XIL_XST_HIDEMESSAGES 環境変数の値](#)
- ・ [hdl_level と hdl_and_low_levels に設定した場合に削除されるメッセージ](#)
- ・ [low_level または hdl_and_low_levels に設定した場合に削除されるメッセージ](#)

XIL_XST_HIDEMESSAGES 環境変数の値

XIL_XST_HIDEMESSAGES 環境変数を使用すると、XST により HDL 合成または下位レベルの合成段階で生成される特定のメッセージを非表示にできます。この環境変数は、次の表のいずれかの値に設定します。

値	説明
none (デフォルト)	すべてのメッセージが表示されます。
hdl_level	VHDL/Verilog 合成および HDL 基本合成、アドバンス合成で表示されるメッセージを削減します。
low_level	下位レベル合成で表示されるメッセージを削減します。
hdl_and_low_levels	すべての段階のメッセージを削減します。

hdl_level と hdl_and_low_levels に設定した場合に削除されるメッセージ

XIL_XST_HIDEMESSAGES 環境変数を `hdl_level` または `hdl_and_low_levels` に設定すると、次のメッセージが表示されなくなります。

- ・ WARNING:HDLCompilers:38 - design.v line 5 Macro 'my_macro' redefined
メモ : このメッセージは、Verilog コンパイラを使用した場合のみ表示されます。
- ・ WARNING:Xst:916 - design.vhd line 5: Delay is ignored for synthesis.
- ・ WARNING:Xst:766 - design.vhd line 5: Generating a Black Box for component comp.
- ・ Instantiating component comp from Library lib.
- ・ Set user-defined property "LOC = X1Y1" for instance inst in unit block.
- ・ Set user-defined property "RLOC = X1Y1" for instance inst in unit block.
- ・ Set user-defined property "INIT = 1" for instance inst in unit block.
- ・ Register reg1 equivalent to reg2 has been removed.

low_level または hdl_and_low_levels に設定した場合に削除されるメッセージ

XIL_XST_HIDEMESSAGES 環境変数を low_level または hdl_and_low_levels に設定すると、次のメッセージが表示されなくなります。

- ・ WARNING:Xst:382 – Register reg1 is equivalent to reg2. Register reg1 equivalent to reg2 has been removed.
- ・ WARNING:Xst:1710 – FF/Latch reg (without init value) is constant in block block.
- ・ WARNING:Xst 1293 – FF/Latch reg is constant in block block.
- ・ WARNING:Xst:1291 – FF/Latch reg is unconnected in block block.
- ・ WARNING:Xst:1426 – The value init of the FF/Latch reg hinders the constant cleaning in the block block. You could achieve better results by setting this init to value.

ログ ファイルのマクロ

XST ログ ファイルには、ブロックごとに VHDL または Verilog ソースから推論される、マクロセットおよび関連する信号などの詳細な情報が含まれます。

マクロの推論は、次の 2 段階で行われます。

1. HDL 合成

XST では、加算器、減算器、レジスタなどのできるだけ多くの単純なマクロ ブロックが認識されます。

2. アドバンス HDL 合成

XST では、HDL 合成段階で認識されたマクロを改善したり(例: 乗算器のパイプライン接続)、ダイナミック シフト レジスタなどの複雑なマクロを新たに作成することで、マクロがさらに詳細に処理されます。アドバンス HDL 合成段階でレポートされるマクロ認識レポートは、HDL 合成段階でのレポートに従ってフォーマットされます。

XST では、認識されたマクロの全体的な統計が次の 2 段階で表示されます。

- ・ HDL 合成段階の後
- ・ アドバンス HDL 合成段階の後

XST では、最終レポートに保護されたマクロの統計は表示されなくなっています。

ログ ファイルの例

このセクションには、次の項目が含まれます。

- ・ [認識されたマクロのログ ファイル例](#)
- ・ [詳細なマクロ処理のログ ファイル例](#)
- ・ [FPGA ログ ファイルの例](#)
- ・ [CPLD ログ ファイルの例](#)

認識されたマクロのログ ファイル例

次のログ ファイルは、ブロックごとに認識されたマクロと、この段階の後の全体的なマクロの統計を示しています。

```
=====
*                      HDL Synthesis                      *
=====

...
Synthesizing Unit <decode>.
    Related source file is "decode.vhd".
    Found 16x10-bit ROM for signal <one_hot>.
    Summary:
        inferred    1 ROM(s).
Unit <decode> synthesized.

Synthesizing Unit <statmach>.
    Related source file is "statmach.vhd".
    Found finite state machine <FSM_0> for signal <current_state>.
    -----
    | States           | 6 |
    | Transitions      | 11 |
    | Inputs           | 1 |
    | Outputs          | 2 |
    | Clock            | CLK (rising_edge) |
    | Reset            | RESET (positive) |
    | Reset type       | asynchronous |
    | Reset State      | clear |
    | Power Up State   | clear |
    | Encoding         | automatic |
    | Implementation   | LUT |
    -----
    Summary:
        inferred    1 Finite State Machine(s).
Unit <statmach> synthesized.

...
=====
HDL Synthesis Report

Macro Statistics
# ROMs                                     : 3
    16x10-bit ROM                         : 1
    16x7-bit ROM                          : 2
# Counters                                : 2
    4-bit up counter                      : 2

=====
...
```

詳細なマクロ処理のログ ファイル例

次の XST の FPGA ログ ファイルは、アドバンス HDL 合成中にさらに詳しく処理されたマクロと、この段階の後の全体的なマクロの統計を示しています。

```
=====
*           Advanced HDL Synthesis           *
=====

Analyzing FSM <FSM_0> for best encoding.
Optimizing FSM <MACHINE/current_state/FSM_0> on signal <current_state[1:3]> with gray encoding.
-----
State      | Encoding
-----
clear      | 000
zero       | 001
start      | 011
counting   | 010
stop       | 110
stopped    | 111
-----

=====
Advanced HDL Synthesis Report

Macro Statistics
# FSMs                      : 1
# ROMs                      : 3
  16x10-bit ROM             : 1
  16x7-bit ROM              : 2
# Counters                  : 2
  4-bit up counter          : 2
# Registers                  : 3
  Flip-Flops/Latches        : 3

=====
...
```

FPGA ログ ファイルの例

次は、FPGA 合成の XST ログ ファイルの例です。Release 10.1 – xst K.31 (nt64)

Copyright (c) 1995–2008 Xilinx, Inc. All rights reserved.

TABLE OF CONTENTS

- 1) Synthesis Options Summary
- 2) HDL Compilation
- 3) Design Hierarchy Analysis
- 4) HDL Analysis
- 5) HDL Synthesis

```
5.1) HDL Synthesis Report
6) Advanced HDL Synthesis
6.1) Advanced HDL Synthesis Report
7) Low Level Synthesis
8) Partition Report
9) Final Report
9.1) Device utilization summary
9.2) Partition Resource Summary
9.3) TIMING REPORT

=====
* Synthesis Options Summary *
=====

---- Source Parameters
Input File Name : "stopwatch.prj"
Input Format : mixed
Ignore Synthesis Constraint File : NO
---- Target Parameters
Output File Name : "stopwatch"
Output Format : NGC
Target Device : xc4vlx15-12-sf363
---- Source Options
Top Module Name : stopwatch
Automatic FSM Extraction : YES
FSM Encoding Algorithm : Auto
Safe Implementation : No
FSM Style : lut
RAM Extraction : Yes
RAM Style : Auto
ROM Extraction : Yes
Mux Style : Auto
Decoder Extraction : YES
Priority Encoder Extraction : YES
Shift Register Extraction : YES
Logical Shifter Extraction : YES
```

```
XOR Collapsing : YES
ROM Style : Auto
Mux Extraction : YES
Resource Sharing : YES
Asynchronous To Synchronous : NO
Use DSP Block : auto
Automatic Register Balancing : No
---- Target Options
Add IO Buffers : YES
Global Maximum Fanout : 500
Add Generic Clock Buffer(BUFG) : 32
Number of Regional Clock Buffers : 16
Register Duplication : YES
Slice Packing : YES
Optimize Instantiated Primitives : NO
Use Clock Enable : Auto
Use Synchronous Set : Auto
Use Synchronous Reset : Auto
Pack IO Registers into IOBs : auto
Equivalent register Removal : YES
---- General Options
Optimization Goal : Speed
Optimization Effort : 1
Power Reduction : NO
Library Search Order : stopwatch.lso
Keep Hierarchy : NO
Netlist Hierarchy : as_optimized
RTL Output : Yes
Global Optimization : AllClockNets
Read Cores : YES
Write Timing Constraints : NO
Cross Clock Analysis : NO
Hierarchy Separator : /
Bus Delimiter : <>
```

```
Case Specifier : maintain
Slice Utilization Ratio : 100
BRAM Utilization Ratio : 100
DSP48 Utilization Ratio : 100
Verilog 2001 : YES
Auto BRAM Packing : NO
Slice Utilization Ratio Delta : 5
=====
=====
* HDL Compilation *
=====
Compiling verilog file "smallcntr.v" in library work
Compiling verilog file "statmach.v" in library work
Module <smallcntr> compiled
Compiling verilog file "hex2led.v" in library work
Module <statmach> compiled
Compiling verilog file "decode.v" in library work
Module <hex2led> compiled
Compiling verilog file "cnt60.v" in library work
Module <decode> compiled
Compiling verilog file "stopwatch.v" in library work
Module <cnt60> compiled
Module <stopwatch> compiled
No errors in compilation
Analysis of file <"stopwatch.prj"> succeeded.
Compiling vhdl file "C:/xst/watchver/tenths.vhd" in Library work.
Entity <tenths> compiled.
Entity <tenths> (Architecture <tenths_a>) compiled.
Compiling vhdl file "C:/xst/watchver/dcm1.vhd" in Library work.
Entity <dcm1> compiled.
Entity <dcm1> (Architecture <BEHAVIORAL>) compiled.
=====
* Design Hierarchy Analysis *
```

```
Analyzing hierarchy for module <stopwatch> in library <work>.
Analyzing hierarchy for entity <dcml> in library <work>
  (architecture <BEHAVIORAL>).
Analyzing hierarchy for module <statmach> in library <work> with
  parameters.
clear = "000001"
counting = "001000"
start = "000100"
stop = "010000"
stopped = "100000"
zero = "000010"
Analyzing hierarchy for module <decode> in library <work>.
Analyzing hierarchy for module <cnt60> in library <work>.
Analyzing hierarchy for module <hex2led> in library <work>.
Analyzing hierarchy for module <smallcntr> in library <work>.
=====
* HDL Analysis *
=====
Analyzing top module <stopwatch>.
Module <stopwatch> is correct for synthesis.
Analyzing Entity <dcml> in library <work> (Architecture
  <BEHAVIORAL>).
Set user-defined property "CAPACITANCE = DONT_CARE" for instance
  <CLKIN_IBUFG_INST> in unit <dcml>.
Set user-defined property "IBUF_DELAY_VALUE = 0" for instance
  <CLKIN_IBUFG_INST> in unit <dcml>.
Set user-defined property "IOSTANDARD = DEFAULT" for instance
  <CLKIN_IBUFG_INST> in unit <dcml>.
Set user-defined property "CLKDV_DIVIDE = 2.0000000000000000" for
  instance <DCM_INST> in unit <dcml>.
Set user-defined property "CLKFX_DIVIDE = 1" for instance
  <DCM_INST> in unit <dcml>.
Set user-defined property "CLKFX_MULTIPLY = 4" for instance
  <DCM_INST> in unit <dcml>.
Set user-defined property "CLKIN_DIVIDE_BY_2 = FALSE" for
  instance <DCM_INST> in unit <dcml>.
Set user-defined property "CLKIN_PERIOD = 20.0000000000000000"
  for instance <DCM_INST> in unit <dcml>.
```

```
Set user-defined property "CLKOUT_PHASE_SHIFT = NONE" for
instance <DCM_INST> in unit <dcml>.

Set user-defined property "CLK_FEEDBACK = 1X" for instance
<DCM_INST> in unit <dcml>.

Set user-defined property "DESKEW_ADJUST = SYSTEM_SYNCHRONOUS"
for instance <DCM_INST> in unit <dcml>.

Set user-defined property "DFS_FREQUENCY_MODE = LOW" for instance
<DCM_INST> in unit <dcml>.

Set user-defined property "DLL_FREQUENCY_MODE = LOW" for instance
<DCM_INST> in unit <dcml>.

Set user-defined property "DSS_MODE = NONE" for instance
<DCM_INST> in unit <dcml>.

Set user-defined property "DUTY_CYCLE_CORRECTION = TRUE" for
instance <DCM_INST> in unit <dcml>.

Set user-defined property "FACTORY_JF = C080" for instance
<DCM_INST> in unit <dcml>.

Set user-defined property "PHASE_SHIFT = 0" for instance
<DCM_INST> in unit <dcml>.

Set user-defined property "SIM_MODE = SAFE" for instance
<DCM_INST> in unit <dcml>.

Set user-defined property "STARTUP_WAIT = TRUE" for instance
<DCM_INST> in unit <dcml>.

Entity <dcml> analyzed.  Unit <dcml> generated.

Analyzing module <statmach> in library <work>.

clear = 6'b0000001
counting = 6'b001000
start = 6'b000100
stop = 6'b010000
stopped = 6'b100000
zero = 6'b000010

Module <statmach> is correct for synthesis.

Analyzing module <decode> in library <work>.

Module <decode> is correct for synthesis.

Analyzing module <cnt60> in library <work>.

Module <cnt60> is correct for synthesis.

Analyzing module <smallcntr> in library <work>.

Module <smallcntr> is correct for synthesis.

Analyzing module <hex2led> in library <work>.
```

```
Module <hex2led> is correct for synthesis.

=====

* HDL Synthesis *

=====

Performing bidirectional port resolution...
Synthesizing Unit <statmach>.
Related source file is "statmach.v".
Found finite state machine <FSM_0> for signal <current_state>.

-----

| States | 6 |
| Transitions | 15 |
| Inputs | 2 |
| Outputs | 2 |
| Clock | CLK (rising_edge) |
| Reset | RESET (positive) |
| Reset type | asynchronous |
| Reset State | 000001 |
| Encoding | automatic |
| Implementation | LUT |

-----

Found 1-bit register for signal <CLKEN>.
Found 1-bit register for signal <RST>.
Summary:
inferred 1 Finite State Machine(s).
inferred 2 D-type flip-flop(s).
Unit <statmach> synthesized.
Synthesizing Unit <decode>.
Related source file is "decode.v".
Found 16x10-bit ROM for signal <ONE_HOT>.
Summary:
inferred 1 ROM(s).
Unit <decode> synthesized.
Synthesizing Unit <hex2led>.
Related source file is "hex2led.v".
```

```

Found 16x7-bit ROM for signal <LED>.
Summary:
inferred 1 ROM(s).
Unit <hex2led> synthesized.
Synthesizing Unit <smallcntr>.
Related source file is "smallcntr.v".
Found 4-bit up counter for signal <QOUT>.
Summary:
inferred 1 Counter(s).
Unit <smallcntr> synthesized.
Synthesizing Unit <dcml>.
Related source file is "C:/xst/watchver/dcml.vhd".
Unit <dcml> synthesized.
Synthesizing Unit <cnt60>.
Related source file is "cnt60.v".
Unit <cnt60> synthesized.
Synthesizing Unit <stopwatch>.
Related source file is "stopwatch.v".
Unit <stopwatch> synthesized.
=====

HDL Synthesis Report
Macro Statistics
# ROMs : 3
16x10-bit ROM : 1
16x7-bit ROM : 2
# Counters : 2
4-bit up counter : 2
# Registers : 2
1-bit register : 2
=====
=====

* Advanced HDL Synthesis *
=====

Analyzing FSM <FSM_0> for best encoding.

```

```
Optimizing FSM <MACHINE/current_state/FSM> on signal
<current_state[1:3]> with sequential encoding.
```

```
-----
State | Encoding
```

```
-----
000001 | 000
```

```
000010 | 001
```

```
000100 | 010
```

```
001000 | 011
```

```
010000 | 100
```

```
100000 | 101
-----
```

```
Loading device for application Rf_Device from file '4v1x15.nph'
in environment C:\xilinx.
```

```
Executing edif2ngd -noa "tenths.edn" "tenths.ngo"
```

```
Release 10.1 - edif2ngd K.31 (nt64)
```

```
Copyright (c) 1995-2008 Xilinx, Inc. All rights reserved.
```

```
INFO:NgdBuild - Release 10.1 edif2ngd K.31 (nt64)
```

```
INFO:NgdBuild - Copyright (c) 1995-2008 Xilinx, Inc. All rights
reserved.
```

```
Writing module to "tenths.ngo"...
```

```
Reading core <tenths_c_counter_binary_v8_0_xst_1.ngc>.
```

```
Loading core <tenths_c_counter_binary_v8_0_xst_1> for timing and
area information for instance <BU2>.
```

```
Loading core <tenths> for timing and area information for
instance <xcounter>.
```

```
=====
Advanced HDL Synthesis Report
```

```
Macro Statistics
```

```
# ROMs : 3
```

```
16x10-bit ROM : 1
```

```
16x7-bit ROM : 2
```

```
# Counters : 2
```

```
4-bit up counter : 2
```

```
# Registers : 5
```

```
Flip-Flops : 5
```

```
=====
=====
* Low Level Synthesis *
=====

Optimizing unit <stopwatch> ...
Mapping all equations...
Building and optimizing final netlist ...
Found area constraint ratio of 100 (+ 5) on block stopwatch,
actual ratio is 0.
Number of LUT replicated for flop-pair packing : 0
Final Macro Processing ...
=====

Final Register Report
Macro Statistics
# Registers : 13
Flip-Flops : 13
=====
=====

* Partition Report *
=====

Partition Implementation Status
-----

No Partitions were found in this design.
-----

=====

* Final Report *
=====

Final Results
RTL Top Level Output File Name : stopwatch.ngc
Top Level Output File Name : stopwatch
Output Format : NGC
Optimization Goal : Speed
Keep Hierarchy : NO
Design Statistics
# IOs : 27
```

```
Cell Usage :
# BELS : 70
# GND : 2
# INV : 1
# LUT1 : 3
# LUT2 : 1
# LUT2_L : 1
# LUT3 : 8
# LUT3_D : 1
# LUT3_L : 1
# LUT4 : 37
# LUT4_D : 1
# LUT4_L : 4
# MUXCY : 3
# MUXF5 : 2
# VCC : 1
# XORCY : 4
# FlipFlops/Latches : 17
# FDC : 13
# FDCE : 4
# Clock Buffers : 1
# BUFG : 1
# IO Buffers : 27
# IBUF : 2
# IBUFG : 1
# OBUF : 24
# DCM_ADVs : 1
# DCM_ADV : 1

=====

Device utilization summary:
-----

Selected Device : 4vlx15sf363-12
Number of Slices: 32 out of 6144 0%
Number of Slice Flip Flops: 17 out of 12288 0%
```

Number of 4 input LUTs: 58 out of 12288 0%

Number of IOs: 27

Number of bonded IOBs: 27 out of 240 11%

Number of GCLKs: 1 out of 32 3%

Number of DCM_ADVs: 1 out of 4 25%

Partition Resource Summary:

No Partitions were found in this design.

===== TIMING REPORT

NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE.

FOR ACCURATE TIMING INFORMATION PLEASE REFER TO THE TRACE REPORT
GENERATED AFTER PLACE-and-ROUTE.

Clock Information:

-----+-----
Clock Signal | Clock buffer (FF name) | Load |
-----+-----
CLK | Inst_dcm1/DCM_INST:CLK0 | 17 |
-----+-----

Asynchronous Control Signals Information:

-----+-----
Control Signal | Buffer (FF name) | Load |
-----+-----
MACHINE/RST (MACHINE/RST:Q) | NONE (sixty/lsbcount/QOUT_1) | 8 |
RESET | IBUF | 5 |
sixty/msbclr (sixty/msbclr_f5:O) | NONE (sixty/msbcount/QOUT_0) | 4
|
-----+-----

Timing Summary:

Speed Grade: -12

```

Minimum period: 2.282ns (Maximum Frequency: 438.212MHz)
Minimum input arrival time before clock: 1.655ns
Maximum output required time after clock: 4.617ns
Maximum combinational path delay: No path found
Timing Detail:
-----

All values displayed in nanoseconds (ns)
=====

Timing constraint: Default period analysis for Clock 'CLK'
Clock period: 2.282ns (frequency: 438.212MHz)
Total number of paths / destination ports: 134 / 21
-----

Delay: 2.282ns (Levels of Logic = 4)
Source: xcounter/BU2/U0/q_i_1 (FF)
Destination: sixty/msbcount/QOUT_1 (FF)
Source Clock: CLK rising
Destination Clock: CLK rising
Data Path: xcounter/BU2/U0/q_i_1 to sixty/msbcount/QOUT_1
Gate Net
Cell:in->out fanout Delay Delay Logical Name (Net Name)
-----

FDCE:C->Q 12 0.272 0.672 U0/q_i_1 (q(1))
LUT4:I0->O 11 0.147 0.492 U0/thresh0_i_cmp_eq00001 (thresh0)
end scope: 'BU2'
end scope: 'xcounter'
LUT4_D:I3->O 1 0.147 0.388 sixty/msbce (sixty/msbce)
LUT3:I2->O 1 0.147 0.000 sixty/msbcount/QOUT_1_rstpot
(sixty/msbcount/QOUT_1_rstpot)
FDC:D 0.017 sixty/msbcount/QOUT_1
-----

Total 2.282ns (0.730ns logic, 1.552ns route)
(32.0% logic, 68.0% route)
=====

Timing constraint: Default OFFSET IN BEFORE for Clock 'CLK'
Total number of paths / destination ports: 4 / 3

```

```

-----
Offset:  1.655ns (Levels of Logic = 3)
Source:  STRTSTOP (PAD)
Destination:  MACHINE/current_state_FSM_FFd3 (FF)
Destination Clock:  CLK rising
Data Path:  STRTSTOP to MACHINE/current_state_FSM_FFd3
Gate Net
Cell:in->out fanout Delay Delay Logical Name (Net Name)
-----

IBUF:I->O 4 0.754 0.446 STRTSTOP_IBUF (STRTSTOP_IBUF)
LUT4:I2->O 1 0.147 0.000 MACHINE/current_state_FSM_FFd3-In_F
(N48)
MUXF5:I0->O 1 0.291 0.000 MACHINE/current_state_FSM_FFd3-In
(MACHINE/current_state_FSM_FFd3-In)
FDC:D 0.017 MACHINE/current_state_FSM_FFd3
-----

Total 1.655ns (1.209ns logic, 0.446ns route)
(73.0% logic, 27.0% route)
=====

Timing constraint:  Default OFFSET OUT AFTER for Clock 'CLK'
Total number of paths / destination ports:  96 / 24
-----

Offset:  4.617ns (Levels of Logic = 2)
Source:  sixty/lsbcount/QOUT_1 (FF)
Destination:  ONESOUT<6> (PAD)
Source Clock:  CLK rising
Data Path:  sixty/lsbcount/QOUT_1 to ONESOUT<6>
Gate Net
Cell:in->out fanout Delay Delay Logical Name (Net Name)
-----

FDC:C->Q 13 0.272 0.677 sixty/lsbcount/QOUT_1
(sixty/lsbcount/QOUT_1)
LUT4:I0->O 1 0.147 0.266 lsbled/Mrom_LED21 (lsbled/Mrom_LED2)
OBUF:I->O 3.255 ONESOUT_2_OBUF (ONESOUT<2>)
-----

```

```
Total 4.617ns (3.674ns logic, 0.943ns route)
(79.6% logic, 20.4% route)
=====
Total REAL time to Xst completion:  20.00 secs
Total CPU time to Xst completion:  19.53 secs
-->
Total memory usage is 333688 kilobytes
Number of errors :  0 ( 0 filtered)
)Number of warnings :  0 ( 0 filtered)
Number of infos :  1 ( 0 filtered)
```

CPLD ログ ファイルの例

```
The following is an example of an XST log file for CPLD
synthesis.

Release 10.1 - xst K.31 (nt64)
Copyright (c) 1995-2008 Xilinx, Inc.  All rights reserved.

TABLE OF CONTENTS

1) Synthesis Options Summary
2) HDL Compilation
3) Design Hierarchy Analysis
4) HDL Analysis
5) HDL Synthesis
5.1) HDL Synthesis Report
6) Advanced HDL Synthesis
6.1) Advanced HDL Synthesis Report
7) Low Level Synthesis
8) Partition Report
9) Final Report

=====
* Synthesis Options Summary *
=====

---- Source Parameters

Input File Name :  "stopwatch.prj"
Input Format :  mixed
Ignore Synthesis Constraint File :  NO
```

```
----- Target Parameters
Output File Name : "stopwatch"
Output Format :   NGC
Target Device :   CoolRunner2 CPLDs

----- Source Options
Top Module Name :  stopwatch
Automatic FSM Extraction : YES
FSM Encoding Algorithm : Auto
Safe Implementation : No
Mux Extraction : YES
Resource Sharing : YES

----- Target Options
Add IO Buffers : YES
MACRO Preserve : YES
XOR Preserve : YES
Equivalent register Removal : YES

----- General Options
Optimization Goal : Speed
Optimization Effort : 1
Library Search Order : stopwatch.lso
Keep Hierarchy : YES
Netlist Hierarchy : as_optimized
RTL Output : Yes
Hierarchy Separator : /
Bus Delimiter : <>
Case Specifier : maintain
Verilog 2001 : YES

----- Other Options
Clock Enable : YES
wysiwyg : NO

=====
=====

* HDL Compilation *

=====
```

```
Compiling verilog file "smallcntr.v" in library work
Compiling verilog file "tenths.v" in library work
Module <smallcntr> compiled
Compiling verilog file "statmach.v" in library work
Module <tenths> compiled
Compiling verilog file "hex2led.v" in library work
Module <statmach> compiled
Compiling verilog file "decode.v" in library work
Module <hex2led> compiled
Compiling verilog file "cnt60.v" in library work
Module <decode> compiled
Compiling verilog file "stopwatch.v" in library work
Module <cnt60> compiled
Module <stopwatch> compiled
No errors in compilation
Analysis of file <"stopwatch.prj"> succeeded.
=====
* Design Hierarchy Analysis *
=====
Analyzing hierarchy for module <stopwatch> in library <work>.
Analyzing hierarchy for module <statmach> in library <work> with
parameters.
clear = "000001"
counting = "001000"
start = "000100"
stop = "010000"
stopped = "100000"
zero = "000010"
Analyzing hierarchy for module <tenths> in library <work>.
Analyzing hierarchy for module <decode> in library <work>.
Analyzing hierarchy for module <cnt60> in library <work>.
Analyzing hierarchy for module <hex2led> in library <work>.
Analyzing hierarchy for module <smallcntr> in library <work>.
=====
* HDL Analysis *
```

```

=====
Analyzing top module <stopwatch>.
Module <stopwatch> is correct for synthesis.
Analyzing module <statmach> in library <work>.
clear = 6'b0000001
counting = 6'b001000
start = 6'b000100
stop = 6'b010000
stopped = 6'b100000
zero = 6'b000010
Module <statmach> is correct for synthesis.
Analyzing module <tenths> in library <work>.
Module <tenths> is correct for synthesis.
Analyzing module <decode> in library <work>.
Module <decode> is correct for synthesis.
Analyzing module <cnt60> in library <work>.
Module <cnt60> is correct for synthesis.
Analyzing module <smallcntr> in library <work>.
Module <smallcntr> is correct for synthesis.
Analyzing module <hex2led> in library <work>.
Module <hex2led> is correct for synthesis.
=====
* HDL Synthesis *
=====
Performing bidirectional port resolution...
Synthesizing Unit <statmach>.
Related source file is "statmach.v".
Found finite state machine <FSM_0> for signal <current_state>.
-----
| States | 6 |
| Transitions | 15 |
| Inputs | 2 |
| Outputs | 2 |
| Clock | CLK (rising_edge) |

```

```
| Reset | RESET (positive) |
| Reset type | asynchronous |
| Reset State | 000001 |
| Encoding | automatic |
| Implementation | automatic |
-----

Found 1-bit register for signal <CLKEN>.
Found 1-bit register for signal <RST>.
Summary:
inferred 1 Finite State Machine(s).
inferred 2 D-type flip-flop(s).
Unit <statmach> synthesized.
Synthesizing Unit <tenths>.
Related source file is "tenths.v".
Found 4-bit up counter for signal <Q>.
Summary:
inferred 1 Counter(s).
Unit <tenths> synthesized.
Synthesizing Unit <decode>.
Related source file is "decode.v".
Found 16x10-bit ROM for signal <ONE_HOT>.
Summary:
inferred 1 ROM(s).
Unit <decode> synthesized
Synthesizing Unit <hex2led>.
Related source file is "hex2led.v".
Found 16x7-bit ROM for signal <LED>.
Summary:
inferred 1 ROM(s).
Unit <hex2led> synthesized.
Synthesizing Unit <smallcntr>.
Related source file is "smallcntr.v".
Found 4-bit up counter for signal <QOUT>.
Summary:
```

```

inferred 1 Counter(s).
Unit <smallcntr> synthesized.
Synthesizing Unit <cnt60>.
Related source file is "cnt60.v".
Unit <cnt60> synthesized.
Synthesizing Unit <stopwatch>.
Related source file is "stopwatch.v".
Found 1-bit register for signal <strtstopinv>.
Summary:
inferred 1 D-type flip-flop(s).
Unit <stopwatch> synthesized.
=====

HDL Synthesis Report
Macro Statistics
# ROMs : 3
16x10-bit ROM : 1
16x7-bit ROM : 2
# Counters : 3
4-bit up counter : 3
# Registers : 3
1-bit register : 3
=====
=====

* Advanced HDL Synthesis *
=====

Analyzing FSM <FSM_0> for best encoding.
Optimizing FSM <MACHINE/current_state/FSM> on signal
<current_state[1:3]> with sequential encoding.
-----

State | Encoding
-----

000001 | 000
000010 | 001
000100 | 010
001000 | 011

```

```
010000 | 100
100000 | 101
-----

=====

Advanced HDL Synthesis Report
Macro Statistics
# ROMs : 3
16x10-bit ROM : 1
16x7-bit ROM : 2
# Counters : 3
4-bit up counter : 3
# Registers : 6
Flip-Flops : 6

=====

=====

* Low Level Synthesis *

=====

Optimizing unit <stopwatch> ...
Optimizing unit <statmach> ...
Optimizing unit <decode> ...
Optimizing unit <hex2led> ...
Optimizing unit <tenths> ...
Optimizing unit <smallcntr> ...
Optimizing unit <cnt60> ...

=====

* Partition Report *

=====

Partition Implementation Status
-----

No Partitions were found in this design.
-----

=====

* Final Report *

=====
```

```
Final Results
RTL Top Level Output File Name :  stopwatch.ngc
Top Level Output File Name :  stopwatch
Output Format :  NGC
Optimization Goal :  Speed
Keep Hierarchy :  YES
Target Technology :  CoolRunner2 CPLDs
Macro Preserve :  YES
XOR Preserve :  YES
Clock Enable :  YES
wysiwyg :  NO
Design Statistics
# IOs :  28
Cell Usage :
# BELS :  413
# AND2 :  120
# AND3 :  10
# AND4 :  6
# INV :  174
# OR2 :  93
# OR3 :  1
# XOR2 :  9
# FlipFlops/Latches :  18
# FD :  1
# FDC :  5
# FDCE :  12
# IO Buffers :  28
# IBUF :  4
# OBUF :  24
=====
Total REAL time to Xst completion:  7.00 secs
Total CPU time to Xst completion:  6.83 secs
-->
Total memory usage is 196636 kilobytes
```

```
Number of errors : 0 ( 0 filtered)
Number of warnings : 0 ( 0 filtered)
Number of infos : 0 ( 0 filtered)
```


XST の命名規則

この章では、XST の命名規則について説明します。

- ・ ネットの命名規則
- ・ インスタンスの命名規則
- ・ 命名規則の制御方法

ネットの命名規則

次のXST でのネットの命名規則は、優先順にリストしています。

1. 外部ピン名を保持します。
2. 信号名の階層を、スラッシュまたはアンダースコアで区切って保持します。
3. ステートビットを含むレジスタの出力信号名を保持します。レジスタが推論されるレベルからの階層名を使用します。
4. クロック バッファの出力信号名には、クロック信号名の後にアンダースコアとクロック バッファ タイプ (BUFGP、IBUFG など) を付けます。
5. レジスタおよびトライステート名に対する入力ネットを保持します。
6. プリミティブおよびブラック ボックスに接続されている信号名を保持します。
7. IBUF の出力ネット名には **net_name_IBUF** という形式の名前を付けます。たとえば、DIN という名前の出力ネットを持つ IBUF の場合、出力 IBUF ネット名は DIN_IBUF になります。
8. OBUF の入力ネット名には **net_name_OBUF** という形式の名前を付けます。たとえば、DOUT という名前の入力ネットを持つ OBUF の場合、入力 OBUF ネット名は DOUT_OBUF になります。
9. 内部 (組み合わせ) ネットの名前のベース名には、ユーザーの HDL 信号名が使用されます。

インスタンスの命名規則

インスタンスを命名する際は、次のインスタンスの命名規則に従ってください。

前リリースの ISE® Design Suite の命名規則を使用するインスタンスを使用する場合は、XST のコマンド ラインで次のオプションを使用してください。

`-old_instance_names 1`

次の命名規則は、優先順にリストしています。

1. インスタンス名の階層を、スラッシュまたはアンダースコアで区切って保持します。

インスタンス名が VHDL または Verilog の generate 文で生成される場合、generate 文からラベルがインスタンス名の一部に使用されます。

たとえば、次のような VHDL の generate 文があるとします。

```
il_loop: for i in 1 to 10 generate
inst_lut:LUT2 generic map (INIT => "00")
```

XST では、LUT 2 に対して次のインスタンス名が生成されます。

```
il_loop[1].inst_lut
il_loop[2].inst_lut
il_loop[9].inst_lut
...
il_loop[10].inst_lut
```

2. 出力信号に対しては、ステートビットを含むレジスタ インスタンスに名前を付けます。
3. クロック バッファ インスタンス名には、出力信号名の後にアンダースコアとクロック バッファ タイプを付けます (例: _BUFGP や _IBUFG など)。
4. ブラック ボックスのインスタンシエーション インスタンス名を保持します。
5. ライブラリ プリミティブのインスタンシエーション インスタンス名を保持します。
6. 入力および出力バッファ名には、パッド名の後に _IBUF または _OBUF を付けます。
7. IBUF の出力インスタンスには **instance_name_IBUF** という形式の名前を付けます。
8. OBUF の入力インスタンスには **instance_name_OBUF** という形式の名前を付けます。

命名規則の制御方法

次の制約で、名前の記述方法が制御できます。

- ・ 階層区切り文字の指定 (-hierarchy_separator)
- ・ バスの区切り文字指定 (-bus_delimiter)
- ・ 大文字/小文字の指定 (-case)
- ・ 複製接尾語の設定 (-duplication_suffix)

ISE Design Suite で次のように定義します。

- ・ [Process Properties] ダイアログ ボックス → [Synthesis Options]、または
- ・ コマンド ライン

詳細は、次を参照してください。

[デザイン制約](#)

コマンド ライン モード

このセクションでは、次について説明します。

- ・ XST コマンド ライン モードの概要
- ・ XST シェルを使用した場合
- ・ スクリプト ファイルを使用した場合
- ・ スクリプト モードでの XST の実行 (VHDL)
- ・ スクリプト モードでの XST の実行 (Verilog)
- ・ スクリプト モードでの XST の実行 (混合言語)
- ・ run コマンドを使用した場合
- ・ set コマンドを使用した場合
- ・ elaborate コマンドを使用した場合
- ・ コマンド ライン モードを使用した VHDL デザインの合成
- ・ コマンド ライン モードを使用した Verilog デザインの合成
- ・ コマンド ライン モードを使用した混合言語デザインの合成

XST コマンド ライン モードの概要

このセクションでは、次について説明します。

- ・ コマンド ライン モードからの XST の実行
- ・ コマンド ライン モードのファイルの種類
- ・ コマンド ライン モードの一時ファイル
- ・ コマンド ライン モードでのスペースを含む名前

コマンド ライン モードからの XST の実行

XST は、次のいずれかの方法でコマンド ラインから実行できます。

- ・ ワークステーションで **xst** を実行
- ・ PC で合 **xst.exe** を実行

コマンド ライン モードのファイルの種類

XST のコマンド ライン モードでは、次のファイルが生成されます。

- ・ NGC デザイン出力ファイル (.ngc)
このファイルは、現在の出力ディレクトリに生成されます (-ofn オプションを参照)。
- ・ RTL and Technology Viewers 用の Register Transfer Level (RTL) ネットリスト (.ngr)
- ・ 合成ログ ファイル (.srp)
- ・ 一時ファイル

コマンド ライン モードの一時ファイル

一時ファイルは、XST の一時ディレクトリに生成されます。デフォルトでは、XST の一時ディレクトリは次になります。

- ・ ワークステーション

/tmp

- ・ Windows

TEMP または TMP 環境変数で指定されたディレクトリ

XST 一時ディレクトリは、**set -tmpdir <directory>** で変更できます。

VHDL/Verilog コンパイル ファイルは、この temp ディレクトリに生成されます。デフォルトの一時ディレクトリは、現在のディレクトリの下にある xst ディレクトリです。

Tip XST の一時ディレクトリには、すべての XST セッションで VHDL および Verilog ファイルのコンパイルにより生成されたファイルが含まれるため、定期的に一時ディレクトリ内のファイルを削除することを強くお勧めします。削除しないと、CPU の動作に悪影響を及ぼす場合があります。この temp ディレクトリ内のファイルは、XST では自動的に削除されません。

コマンド ライン モードでのスペースを含む名前

XST のコマンド ライン モードでは、スペースを含むファイル名またはディレクトリ名がサポートされます。

ファイル名またはディレクトリ名にスペースが含まれる場合は、次の例のように名前を二重引用符 (" ") で囲む必要があります。

```
"C:\my project"
```

これらのオプションで複数のディレクトリを指定する場合は、次の例のように {} でディレクトリを囲んでください。

```
-vlgincdir {"C:\my project" "C:\temp"}
```

XST シェルを使用した場合

xst と入力して、XST シェルを起動します。その後、コマンド名を入力してコマンドを実行します。合成を実行するには、必須のオプションとその値をすべて含めた完全なコマンドを入力する必要があります。XST では、最初に **set option_1** を入力し、次に **set option_2** を入力し、最後に **run** と入力してコマンドを実行することはできません。

オプションはすべて同時に設定する必要があるため、スクリプトファイルの使用の方をお勧めします。

スクリプト ファイルを使用した場合

コマンドを別のスクリプト ファイルに記述して、一度に実行します。スクリプト ファイルを実行するには、UNIX または PC で次のように入力します。

```
xst -ifn in_file_name -ofn out_file_name -intstyle {silent|ise|xflow}
```

-ofn は省略可能です。省略すると、拡張子が .srp のログ ファイルが自動的に生成され、すべてのメッセージが画面に表示されます。次のオプションを使用すると、画面に表示されるメッセージの数を制限できます。

- ・ **-intstyle** silent オプション
- ・ XIL_XST_HIDEMESSAGES 環境変数
- ・ ISE® Design Suite のメッセージ フィルタ機能

詳細は、次を参照してください。

ログ ファイルの容量削減方法

次にスクリプト ファイルの使い方の例を示します。foo.scr という名前のファイルに次のテキストが記述されているとします。

```
run -ifn tt1.prj -top tt1 -ifmt MIXED -opt_mode SPEED -opt_level 1 -ofn tt1.ngc -p <parttype>
```

このスクリプト ファイルを実行するには、次のように入力します。

```
xst -ifn foo.scr
```

また、次のように入力すると、ログ ファイルを生成できます。

```
xst -ifn foo.scr -ofn foo.log
```

スクリプト ファイルは、**xst -ifn script name** を使用するか、XST プロンプトで **script script_name** コマンドを使用して実行します。

script foo.scr

XST コマンドまたはコマンド オプションで誤った入力をする、エラー メッセージが表示され、コマンドは実行されません。たとえば、この例で示したスクリプト ファイルで、VHDL が誤って VHDLL と記述されている場合、次のようなエラー メッセージが表示されます。

```
--> ERROR:Xst:1361 - Syntax error in command run for option  
"-ifmt" : parameter "VHDLL" is not allowed.
```

ISE Design Suite を使ってプロジェクトを作成し、ISE Design Suite から 1 度でも XST を実行したことがある場合は、XST コマンドライン モードに切り替えて、ISE Design Suite で作成されたスクリプトおよびプロジェクト ファイルを使うことができます。

XST をコマンドラインから実行する場合は、プロジェクト ディレクトリで次を入力します。

```
xst -ifn <top_level_block>.xst -ofn <top_level_block>.sy
```

run コマンドを使用した場合

このセクションでは、run コマンドを使用した XST スクリプトの設定方法について説明します。

- ・ [run コマンドの概要](#)
- ・ [スクリプト ファイルの記述](#)
- ・ [XST 固有の制約 \(タイミング以外\)](#)
- ・ [オンライン ヘルプ](#)
- ・ [サポートされるファミリ](#)
- ・ [特定デバイスのコマンド](#)
- ・ [run コマンドのオプションと値 \(Virtex-5 デバイス\)](#)

run コマンドの概要

run コマンド :

- ・ 主な合成コマンドです。
- ・ HDL ファイルの解析から最終ネットリストの生成までの合成プロセスすべてを実行するコマンドです。
- ・ 1 つのスクリプト ファイルで 1 度だけ使用できます。
- ・ コマンドの最初に run と記述し、その後にオプションと値を指定します。

run *option_1 value option_2 value ...*

スクリプト ファイルの記述

スクリプト ファイルを記述する場合は、次の規則に注意してください。

- ・ 1 つの行に 1 つのオプションと値を記述する。
- ・ 次のようにシャープ文字 (#) を挿入すると、そのオプションをコメントアウトしたり、コメントを加えたりすることもできます。

```
run
option_1 value
# option_2 value
option_3 value
```

- ・ 最初の行には run コマンドのみを指定する (オプションは指定しない)。
- ・ コマンドの最初の行から最後の行までの間に空行を作らない。
- ・ オプションの前には、-ifn、-ifmt、-ofn のようにダッシュ (-) を記述します。
- ・ 各オプションごとに 1 つの値を指定する 値を省略できるオプションはなし
- ・ オプションには、次のいずれかを指定します。
 - XST で定義されている値 (yes、no など)。
 - 文字列 (ファイル名、最上位のエンティティ名など)。-vlgincdir オプションでは、複数のディレクトリを値として指定できます。各ディレクトリ名はスペースで区切り、すべてのディレクトリ名を中かっこ {} で囲む必要があります。次に例を示します。

```
-vlgincdir {c:\vlg1 c:\vlg2}
```

詳細は、「[コマンドライン モードでのスペースを含む名前](#)」を参照してください。

- 整数

XST 固有の制約 (タイミング以外)

XST 用のタイミング制約以外のオプションは、次のセクションにまとめられています。run コマンドのオプションとその値もこれらの表に含まれています。

- ・ [XST 固有の制約 \(タイミング以外\)](#)
- ・ [XST 特有のタイミング以外のオプション：XST コマンドラインのみ](#)

オンライン ヘルプ

コマンドラインからは、XST のヘルプ機能を使用できます。ヘルプは、コマンドラインで「help」と入力すると表示されます。表示される情報は、サポートされているファミリ、使用可能なコマンド、オプション、およびその値です。

XST の各コマンドについて詳しい説明を表示するには、次のように入力します。

```
help -arch family_name -command command_name
```

斜体で示された部分には、それぞれ次を入力します。

- ・ `family_name` : 現在のバージョンの XST でサポートされているザイリンクス デバイス ファミリを入力します。
- ・ `command_name` には、次のいずれかのコマンドを入力します。
 - **run**
 - **set**
 - **elaborate**
 - **time**

サポートされるファミリ

サポートされているファミリを表示するには、引数を指定せずに「help」と入力します。次に、入力例と、実行後出力されるメッセージを示します。

```
--> help
ERROR:Xst:1356 - Help : Missing "-arch <family>".
Please specify what family you want to target
available families:
acr2
aspartan3
aspartan3a
aspartan3adsp
aspartan3e
avirtex4
fpgacore
qrvirtex4
qvirtex4
spartan3
spartan3a
spartan3adsp
spartan3e
virtex4
virtex5
xa9500x1
xbr
xc9500
xc9500x1
xpla3
```

特定デバイスのコマンド

特定のファミリで利用できるコマンドのリストを表示するには、次のように入力します。

help -arch *family_name*

次に例を示します。

help -arch virtex

run コマンドのオプションと値 (Virtex-5 デバイス)

Virtex®-5 の場合、**run** コマンドで使用可能なオプションとその値を表示するには、次のように入力します。

```
--> help -arch virtex5 -command run
```

このコマンドを実行すると、次のような情報が表示されます。

```
-mult_style           : Multiplier Style
    block / lut / auto / pipe_lut
-bufg                 : Maximum Global Buffers
    *
-bufgce               : BUFGCE Extraction
    YES / NO
-decoder_extract      : Decoder Extraction
    YES / NO
....

-ifn : *
-ifmt : Mixed / VHDL / Verilog
-ofn : *
-ofmt : NGC / NCD
-p : *
-ent : *
-top : *
-opt_mode : AREA / SPEED
-opt_level : 1 / 2
-keep_hierarchy : YES / NO
-vlginkdir : *
-verilog2001 : YES / NO
-vlgcase : Full / Parallel / Full-Parallel
....
```

set コマンドを使用した場合

XST では、set コマンドが認識されます。

詳細は、次を参照してください。

[デザイン制約](#)

set コマンドのオプション

オプション	説明	値
-tmpdir	現在のセッションで XST により生成された一時ファイルを格納するディレクトリへのパスを指定	ディレクトリへのパス
-xsthdpdir	作業ディレクトリ (VHDL/Verilog コンパイルで生成されたファイルのディレクトリ) を指定	ディレクトリへのパス

オプション	説明	値
-xsthdpi	HDL ライブラリ マップ ファイル (INI ファイル)	file_name

elaborate コマンドを使用した場合

elaborate コマンドは次の目的で使します。

- ・ VHDL および Verilog ファイルを特定ライブラリでプリコンパイルするため、または
- ・ デザインを合成せずに Verilog ファイルを検証するため

コンパイル処理は run コマンドで実行されるため、elaborate コマンドの使用はオプションです。

詳細は、次を参照してください。

[デザイン制約](#)

elaborate コマンドのオプション

オプション	説明	値
-ifn	プロジェクト ファイル	file_name
-ifmt	フォーマット	vhdl、verilog、mixed
-lso	ライブラリ検索順	file_name.lso
-work_lib	コンパイルの作業ディレクトリ (最上位ブロックがコンパイルされるディレクトリ)	name、work
-verilog2001	Verilog-2001	yes、no
-vlgpath	Verilog 検索パス	ディレクトリへのパス (複数指定する場合は、スペースで区切り、二重引用符 (" ") で囲む)
-vlgincdir	Verilog インクルード ディレクトリ	ディレクトリへのパス (複数指定する場合は、スペースで区切り、{ } で囲む)

スクリプト モードでの XST の実行 (VHDL)

VHDL の場合、スクリプト ファイルを使用するには次の手順に従います。

1. 現在のディレクトリにある stopwatch.xst という新しいファイルを開きます。
2. 以前実行した XST のシェル コマンドを作成したファイルに貼り付け保存します。

```
run -ifn watchvhd.prj -ifmt mixed -top stopwatch -ofn watchvhd.ngc
    -ofmt NGC -p xc5vfx30t-2-ff324 -opt_mode Speed -opt_level 1
```

3. tcsh などのシェルから次のコマンドを実行し合成します。

```
xst -ifn stopwatch.xst
```

実行中に作成されるファイル (VHDL)

このコマンドを実行すると、次に示すファイルが生成されます。

- ・ watchvhd.ngc
インプリメンテーション ツールで処理可能な NGC ファイル
- ・ xst.srp
XST ログ ファイル

別のログ ファイルの XST メッセージの保存 (VHDL)

XST のメッセージを異なる名前のログ ファイルに保存する場合は、次のコマンドを実行します。

xst -ifn stopwatch.xst -ofn <filename>.log

次は、watchvhd.log を使用した例です。

```
xst -ifn stopwatch.xst -ofn watchvhd.log
```

この例の場合、stopwatch.xst は次のように表示されます。

```
run
-ifn watchvhd.prj
-ifmt mixed
-top stopwatch
-ofn watchvhd.ngc
-ofmt NGC
-p xc5vfx30t-2-ff324
-opt_mode Speed
-opt_level 1
```

可読性の改善 (VHDL)

合成の実行に多数のオプションを使用する場合などに、stopwatch.xst の可読性を改善するには、次のようにファイルを読みやすくします。

- ・ 1 つの行に 1 つのオプションと値を記述する。
- ・ 最初の行には run コマンドのみを指定する (オプションは指定しない)。
- ・ コマンドの最初の行から最後の行までの間に空行を作らない。
- ・ 最初の行以外の行はダッシュ (-) で開始する。

余分なスペース (VHDL)

値を入力するところにスペースを間違えて入れてしまうとエラーになります。

ISE® Design Suite では、自動的にプロセス値から余分なスペースが削除されるので、ISE Design Suite で記述された XST ファイルには、このような余分なスペースの影響はありません。

ただし、XST ファイルを手動で修正した後に、XST をコマンドラインから実行する場合は、余分なスペースを手動で取るようにしてください。

スクリプト モードでの XST の実行 (Verilog)

このセクションでは、スクリプト モード (Verilog) で XST を実行する方法について説明します。

- ・ スクリプト モードでの XST の実行方法 (Verilog)
- ・ 実行中に作成されるファイル (Verilog)
- ・ 別のログ ファイルの XST メッセージの保存 (Verilog)
- ・ 可読性の改善 (Verilog)

スクリプト モードでの XST の実行方法 (Verilog)

スクリプト ファイルを使用するには次の手順に従います。

1. design.xst という新しいファイルを現在のディレクトリで開きます。以前実行した XST のシェル コマンドを作成したファイルに貼り付け保存します。

```
run
-ifn watchver.prj
-ifmt mixed
-ofn watchver.ngc
-ofmt NGC
-p xc5vfx30t-2-ff324
-opt_mode Speed
-opt_level 1
```

2. tcsh などのシェルから次のコマンドを実行し合成します。

```
xst -ifn design.xst
```

先ほどのコマンド例をこの規則に従って記述すると、design.xst は次のようになります。

```
run
-ifn watchver.prj
-ifmt mixed
-top stopwatch
-ofn watchver.ngc
-ofmt NGC
-p xc5vfx30t-2-ff324
-opt_mode Speed
-opt_level 1
```

実行中に作成されるファイル (Verilog)

このコマンドを実行すると、次に示すファイルが生成されます。

- ・ watchvhd.ngc
インプリメンテーション ツールで処理可能な NGC ファイル
- ・ design.srp
XST スクリプト ログ ファイル

別のログ ファイルの XST メッセージの保存 (Verilog)

XST のメッセージを watchver.log など異なる名前のログ ファイルに保存する場合は、次のコマンドを実行します。

```
xst -ifn design.xst -ofn watchver.log
```

可読性の改善 (Verilog)

合成の実行に多数のオプションを使用する場合などは、1 つの行に 1 つのオプションを記述するようにすると、ファイルが読みやすくなります。その場合、次の規則に従う必要があります。

- ・ 最初の行には run コマンドのみを指定する (オプションは指定しない)。
- ・ コマンドの最初の行から最後の行までの間に空行を作らない。
- ・ 最初の行以外の行はダッシュ (-) で開始する。

スクリプト モードでの XST の実行 (混合言語)

このセクションでは、スクリプト モードで XST を実行する方法 (混合言語) について説明します。

- ・ [スクリプト モードでの XST の実行方法 \(混合言語\)](#)
- ・ [実行中に作成されるファイル \(混合言語\)](#)
- ・ [別のログ ファイルの XST メッセージの保存 \(混合言語\)](#)
- ・ [可読性の改善 \(混合言語\)](#)

スクリプト モードでの XST の実行方法 (混合言語)

スクリプト ファイルを使用するには次の手順に従います。

1. 現在のディレクトリにある stopwatch.xst というファイルを開きます。以前実行した XST のシェル コマンドを作成したファイルに貼り付け保存します。

```
run

-ifn watchver.prj

-ifmt mixed

-top stopwatch

-ofn watchver.ngc

-ofmt NGC

-ofn watchver.ngc

-ofmt NGC

-p xc5vfx30t-2-ff324

-opt_mode Speed

-opt_level 1
```

2. tcsh などのシェルから次のコマンドを実行し合成します。

```
xst -ifn stopwatch.xst
```

前述のコマンド例をこの規則に従って記述すると、stopwatch.xst は次のようになります。

```
run
-ifn watchver.prj
-ifmt mixed
-ofn watchver.ngc
-ofmt NGC
-p xc5vfx30t-2-ff324
-opt_mode Speed
-opt_level 1
```

実行中に作成されるファイル（混合言語）

このコマンドを実行すると、次に示すファイルが生成されます。

- ・ watchver.ngc
インプリメンテーション ツールで処理可能な NGC ファイル
- ・ xst.srp
XST スクリプト ログ ファイル

別のログ ファイルの XST メッセージの保存（混合言語）

XST のメッセージを watchvhd.log など異なる名前のログ ファイルに保存する場合は、次のコマンドを実行します。

```
xst -ifn stopwatch.xst -ofn <filename>.log
```

次は、watchver.log を使用した例です。

```
xst -ifn stopwatch.xst -ofn watchver.log
```

可読性の改善（混合言語）

合成の実行に多数のオプションを使用する場合などは、1 つの行に 1 つのオプションを記述するようにすると、ファイルが読みやすくなります。その場合、次の規則に従う必要があります。

- ・ 最初の行には run コマンドのみを指定する（オプションは指定しない）。
- ・ コマンドの最初の行から最後の行までの間に空行を作らない。
- ・ 最初の行以外の行はダッシュ (-) で開始する。

コマンドライン モードを使用した VHDL デザインの合成

このセクションでは、コマンドライン モードを使用した VHDL の合成方法について説明します。

- ・ [VHDL デザイン ファイルおよびエンティティ](#)
- ・ [コマンドライン モードの使用例](#)
- ・ [デザインの合成](#)
- ・ [ライブラリ名](#)
- ・ [XST のファイル順序に対する警告](#)

次の例では、Virtex® デバイス用に記述された階層のある VHDL デザインをコマンドライン モードを使用して合成する方法を示しています。

VHDL デザイン ファイルおよびエンティティ

この例では、watchvhd という VHDL デザインを使用します。このデザイン ファイルは、ISE® Design Suite をインストールしたディレクトリ内の ISEexamples\watchvhd ディレクトリにあります。

このデザインには、次の 7 つのエンティティが含まれています。

- ・ stopwatch
- ・ statmach
- ・ tenths (CORE Generator™ ソフトウェア コア)
- ・ decode
- ・ smallcntr
- ・ cnt60
- ・ hex2led

コマンド ライン モードの使用例

次は、コマンド ライン モードを使用して VHDL デザインを合成する例です。

1. vhd_m という新しい名前のディレクトリを作成します。
2. ISE Design Suite インストール ディレクトリである ISEexamples\watchvhd ディレクトリから次のファイルを vhd_m ディレクトリにコピーします。
 - ・ stopwatch.vhd
 - ・ statmach.vhd
 - ・ decode.vhd
 - ・ cnt60.vhd
 - ・ smallcntr.vhd
 - ・ tenths.vhd
 - ・ hex2led.vhd
3. 7 つの VHDL ファイルで記述されたこのデザインを合成するため、プロジェクトを作成します。

デザインの合成

XST では、VHDL と Verilog の混合言語プロジェクトがサポートされます。ザイリンクスでは、混合言語プロジェクトであるかないかにかかわらず、新しいプロジェクトフォーマットを使用することをお勧めしています。この例では、新しいプロジェクトフォーマットを使用しています。VHDL ファイルのみを含むプロジェクト ファイルを作成するには、別のファイルに「vhd」と記述して、その後に VHDL ファイルをリストします。ここで、ファイルの順序は重要ではありません。XST により階層が認識され、VHDL ファイルが正しい順序でコンパイルされます。

次の手順に従ってください。

1. watchvhd.prj というファイルを新規作成します。
2. 作成したファイル内に次の VHDL ファイル名を任意の順番で記述し、保存します。

```
vhdl work statmach.vhd
vhdl work decode.vhd
vhdl work stopwatch.vhd
vhdl work cnt60.vhd
vhdl work smallcntr.vhd
vhdl work tenths.vhd
vhdl work hex2led.vhd
```

3. デザインを合成するには、XST シェルまたはスクリプト ファイルを使用して次のコマンドを実行します。

```
run -ifn watchvhd.prj -ifmt mixed -top stopwatch -ofn
watchvhd.ngc -ofmt NGC -p xc5vfx30t-2-ff324 -opt_mode Speed
-opt_level 1
```

-top オプションを使用して必ず最上位デザインのブロックを指定してください。

hex2led のみを個別に合成し、パフォーマンスを確認する場合は、合成する最上位エンティティを -top オプションで指定します。

```
run -ifn watchvhd.prj -ifmt mixed -ofn watchvhd.ngc -ofmt NGC -p
xc5vfx30t-2-ff324 -opt_mode Speed -opt_level 1 -top hex2led
```

詳細は、次を参照してください。

[XST 固有の制約 \(タイミング以外\)](#)

ライブラリ名

VHDL のコンパイル時に、work というライブラリがデフォルトで使用されます。VHDL ファイルを別のライブラリにコンパイルする場合は、ファイル名の前にライブラリ名を追加します。hex2led を my_lib ライブラリにコンパイルする場合、プロジェクトファイルは次のようになります。

```
vhdl work statmach.vhd vhdl work decode.vhd vhdl work
stopwatch.vhd vhdl work cnt60.vhd vhdl work smallcntr.vhd vhdl
work tenths.vhd vhdl my_lib hex2led.vhd
```

XST のファイル順序に対する警告

XST でファイルの順序が正しく認識されず、次のような警告が表示される場合があります。

```
WARNING:XST:3204. The sort of the vhdl files failed, they will
be compiled in the order of the project file.
```

この場合、次の手順に従います。

- ・ すべての VHDL ファイルを正しい順序で記述します。
- ・ -hdl_compilation_order オプションで値 user を run コマンドに追加します。

```
run -ifn watchvhd.prj -ifmt mixed -top stopwatch -ofn
watchvhd.ngc -ofmt NGC -p xc5vfx30t-2-ff324 -opt_mode Speed
-opt_level 1 -top hex2led -hdl_compilation_order user
```

コマンド ライン モードを使用した Verilog デザインの合成

このセクションでは、コマンド ライン モードを使用した Verilog の合成方法について説明します。

- ・ [Verilog デザイン ファイルおよびモジュール](#)
- ・ [コマンド ライン モードの使用例](#)
- ・ [デザインの合成](#)
- ・ [HEX2LED の合成](#)

次の例では、Virtex® デバイス用に記述された階層のある Verilog デザインをコマンド ライン モードを使用して合成する方法を示しています。

Verilog デザイン ファイルおよびモジュール

この例では、watchver という Verilog デザインを使用します。このデザイン ファイルは、次の ISE® Design Suite のインストール ディレクトリにあります。

```
ISEexamples\watchver
```

The files are:

- ・ stopwatch.v
- ・ statmach.v
- ・ decode.v
- ・ cnt60.v
- ・ smallcntr.v
- ・ tenths.v
- ・ hex2led.v

このデザインには、次の 7 つのモジュールが含まれています。

- ・ stopwatch
- ・ statmach
- ・ tenths (CORE Generator™ ソフトウェア コア)
- ・ decode
- ・ cnt60
- ・ smallcntr
- ・ hex2led

コマンド ライン モードの使用例

1. vlg_m という名前のディレクトリを作成します。
 2. ISE Design Suite インストール ディレクトリである ISEexamples\watchver ディレクトリから watchver ファイルを vlg_m という新たに作成されたディレクトリにコピーします。
- top オプションを使用して必ず最上位デザインのブロックを指定してください。

デザインの合成

7 つの Verilog ファイルで記述されたこのデザインを合成するため、プロジェクトを作成します。XST では、VHDL と Verilog の混合言語プロジェクトがサポートされます。このため、ザイリンクスでは、混合言語プロジェクトであるかないかにかかわらず、新しいプロジェクトフォーマットを使用することをお勧めしています。この例では、新しいプロジェクトフォーマットを使用しています。Verilog ファイルのみを含むプロジェクトファイルを作成するには、別のファイルに「verilog」と記述して、その後に Verilog ファイルをリストします。ここで、ファイルの順序は重要ではありません。XST により階層が認識され、Verilog ファイルが正しい順序でコンパイルされます。

1. watchver.v というファイルを新規作成します。
2. 作成したファイル内に次の Verilog ファイル名を任意の順番で記述し、保存します。

```
verilog work decode.v
verilog work statmach.v
verilog work stopwatch.v
verilog work cnt60.v
verilog work smallcntr.v
verilog work hex2led.v
```

3. デザインを合成するには、XST シェルまたはスクリプト ファイルから次のコマンドを実行します。

```
run -ifn watchver.v -ifmt mixed -top stopwatch -ofn
watchver.ngc -ofmt NGC -p xc5vfx30t-2-ff324 -opt_mode Speed
-opt_level 1
```

HEX2LED の合成

hex2led のみを個別に合成し、パフォーマンスを確認する場合は、合成する最上位モジュールを -top オプションで指定します。

```
run -ifn watchver.v -ifmt Verilog -ofn watchver.ngc -ofmt NGC -p
xc5vfx30t-2-ff324 -opt_mode Speed -opt_level 1 -top HEX2LED
```

詳細は、次を参照してください。

[XST 固有の制約 \(タイミング以外\)](#)

コマンドライン モードを使用した混合言語デザインの合成

このセクションでは、コマンドライン モードを使用した混合言語デザインの合成方法について説明します。

- ・ [コマンドライン モードの使用例](#)
- ・ [デザインの合成](#)
- ・ [ファイル順](#)

次の例では、Virtex® デバイス用に記述された階層のある VHDL および Verilog の混合デザインをコマンドライン モードを使用して合成する方法を示しています。

コマンド ライン モードの使用例

1. vhd_verilog という名前のディレクトリを作成します。
2. ISE Design Suite インストール ディレクトリである ISEexamples¥watchvhd ディレクトリから次のファイルを新たに作成した vhd_verilog ディレクトリにコピーします。
 - ・ stopwatch.vhd
 - ・ statmach.vhd
 - ・ decode.vhd
 - ・ cnt60.vhd
 - ・ smallcntr.vhd
 - ・ tenths.vhd
3. ISE Design Suite インストール ディレクトリである ISEexamples¥watchver ディレクトリから hex2led.v ファイルを vhd_verilog という新たに作成されたディレクトリにコピーします。

デザインの合成

このデザインは、6 つの VHDL ファイルと 1 つの Verilog ファイルで記述されています。デザインを合成するには、プロジェクトを作成します。プロジェクト ファイルを作成するには、別のファイルに「vhd」と記述してその後に VHDL ファイルをリストし、「verilog」と記述してその後に Verilog ファイルをリストします。

ファイル順

ここで、ファイルの順序は重要ではありません。XST により階層が認識され、HDL ファイルが正しい順序でコンパイルされます。