Synplifyソフトウェアによる FLEX 10Kデバイスの性能改善方法

1998年 10 月 ver.1.0 Application Note 101

イントロダクション

性能の改善への要求が高まると共に、ロジックの最適化が最高レベルで実現されるようにデザインを構築することが必要になります。VHDLおよびVerilog HDLのコーディング・テクニック、Synplifyソフトウェアのコンストレイント（制約条件）、およびMAX+PLUS® IIソフトウェアのオプションを活用することによって、FLEX® 10Kデバイスにさらに高い性能を実現することができます。これらのツールの活用は、デザイン・フローの簡素化、FLEX 10Kデバイスを使用するデザインの最適化、そして既存の高性能プログラマブル・ロジック・ファミリのスピードをさらに向上させるときに役立ちます。このアプリケーション・ノートは、アルテラとシンプリシティ（Synplicity）社のツールを使用したデザイン・フローにおいて、FLEX 10Kデバイスの性能をさらに改善させるためのテクニックについて解説したものです。

この資料を適切に理解するためには、FLEX 10Kファミリのアーキテクチャ、アルテラのMAX+PLUS IIソフトウェア、シンプリシティ社のSynplifyソフトウェアに対する基本的な理解が必要です。

このアプリケーション・ノートでは、下記の項目について解説します。

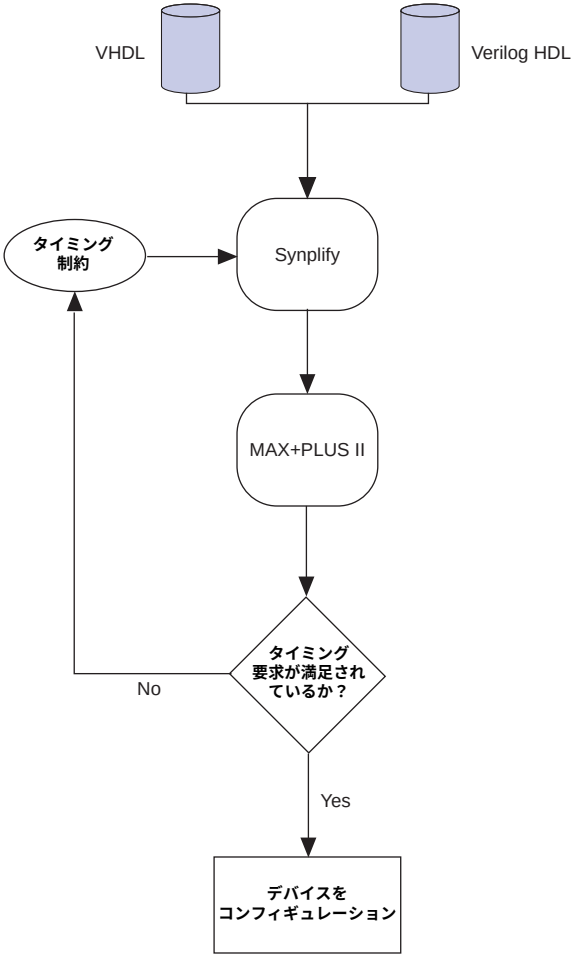
デザイン・フロー

アルテラ／シンプリシティのデザイン・フローでは、まず最初にSynplifyソフトウェアで論理合成を行うためのハードウェア記述言語（HDL）によるデザインの作成を行います。HDLによるデザインの作成方法についてのガイドラインは、「Synplifyソフトウェアにおける効率的なHDLデザイン・テクニック」および「FLEX 10Kデバイスに対するSynplifyソフトウェアでのデザイン・テクニック」の章で解説されています。HDLによるデザインの記述が完了すると、デザインをSynplifyソフトウェアで合成し、生成されたEDIF（Electronic Design Interchange Format）ファイル（.edf）をMAX+PLUS IIのソフトウェアに取り込みます。そして、性能の評価を行った上で、適切であればデザインをFLEX 10Kデバイスに実現し、このデザイン・フローが終了します。

多くのデザイン・テクニックには、エリアとタイミングとの間にトレード・オフが存在します。このため、エリアの使用効率を改善するテクニックを使用したときに、性能が低下することがあります。また、複製されたロジックを使用するテクニックでデザイン全体を改善することができますが、これには追加のロジック・リソースが必要になります。

図１はSynplifyソフトウェアとMAX+PLUS IIソフトウェアを使用したときに推奨されるデザイン・フローを示したものです。

図１　推奨されるデザイン・フロー



Synplifyソフトウェアにおける効率的なHDLデザイン・テクニック

Synplifyソフトウェアにおいて効率的なHDLデザイン・テクニックを使用することで、デザインの簡略化、ロジックの最適化、そしてロジック遅延の低減が実現され、デザイン全体の性能が改善されます。以下のセクションでは、こうした結果を得るための方法を下記の項目ごとに解説します。

- 階層化
- 組み合わせロジック
- プライオリティ・エンコーディング形式のIfステートメント
- ロジック最適化のための"Don't Care"条件
- シーケンシャル・ロジック
- ゲート付きクロック
- ステート・マシン
- エンベデッド・アレイ・ブロック（EAB）にステート・マシンを構成する方法

階層化

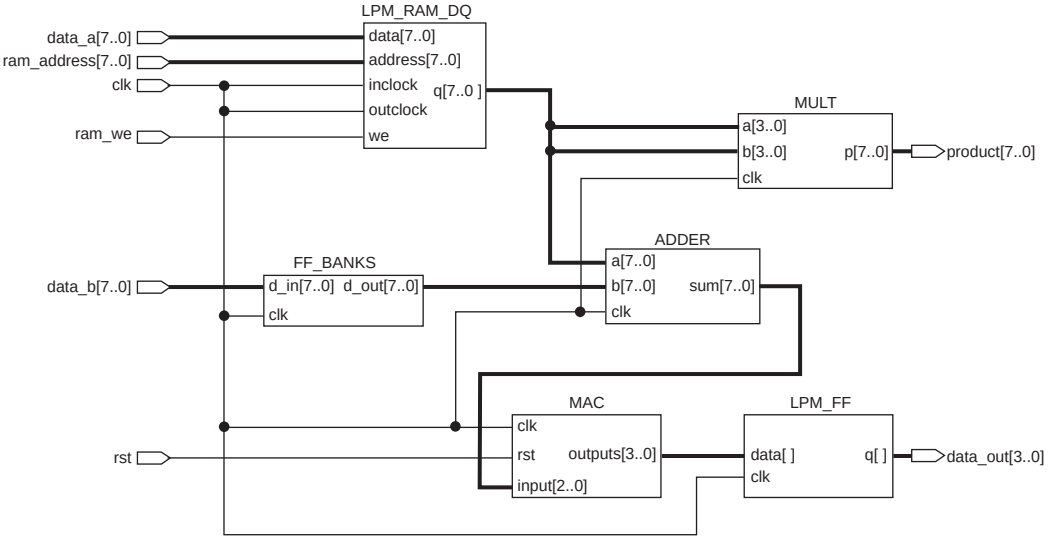
多くのデザインは、回路機能を１個のデザイン・ファイルで実現できないほど複雑なものとなります。Synplifyソフトウェアを使用した場合、デザインを複数のファイルで作成し、これらのファイルをひとつの階層でリンクさせることができます。デザインを標準的なVHDLまたはVerilog HDLで作成して、インスタンス化とシミュレーションを行い、デザイン全体ではなく各サブデザインごとに個別に最適化することができます。デザインの分割は、下記のガイドラインにしたがって行ってください。

- デザインを機能ごとに分割します。この場合、図２に示すようなブロック・ダイヤグラムやハイ・レベルな回路図を使用することで、回路を適切に分割することができます。データ・パス、トライ・ステート信号、ステート・マシン、レジスタのブロック、大規模なマクロファンクション、メモリ・エレメント、コントロール・ブロック、再利用可能なメガファンクションなどが適切に分割されるようにします。

- I/Oとの接続が最小になるようにデザインを分割します。あまりにも多くの本数のI/Oが存在すると、デザインとデバッグの工程が複雑になります。

- 可能な限り、各ブロックの出力にはレジスタを設けるようにします。

図2　階層化されたデザイン



組み合わせロジック

ある時間での出力がその時間の入力の状態のみで規定されるロジックは、回路の前の状態とは関係なく、組み合わせ回路として記述されます。組み合わせ回路の例としては、デコーダ、マルチプレクサ、アダーなどがあります。

組み合わせロジックを採用する場合、下記のセクションで解説されているテクニックを使用することによって、Synplifyソフトウェアの論理合成から得られる合成結果の性能を改善させることができます。

ラッチ

FLEXデバイスのシリコン上にはラッチではなく、レジスタが組み込まれています。このため、ラッチを使用したデザインでは、レジスタを使用した場合よりも多くのロジックが生成され、性能も低下します。例えば、MAX+PLUS IIソフトウェアでは、１個のラッチの構成にFLEXデバイス内の２個のロジック・エレメント（LE）が使用されます。

組み合わせ回路のロジックをデザインするときは、HDLのデザイン記述形式によって意図しないラッチが生成されないように注意する必要があります。例えば、CaseやIfを使用したステートメントが入力のすべての条件をカバーしていない場合は、組み合わせ回路のフィードバックによってラッチが形成されてしまう可能があります。

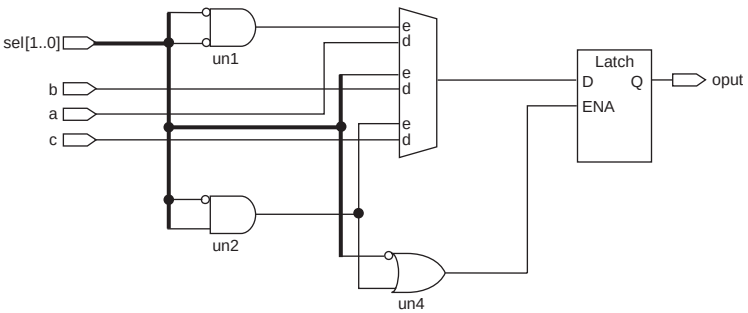
図３は、ラッチが生成されるVHDLの記述例を示したものです。

図３　ラッチを生成させるVHDLの記述例

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY bad IS
PORT  ( a,b,c  : IN STD_LOGIC;
        sel    : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
        oput   : OUT STD_LOGIC);
END bad;

ARCHITECTURE behave OF bad IS
BEGIN
    PROCESS (a,b,c,sel)
    BEGIN
        IF sel = "00" THEN
             oput <= a;
        ELSIF sel = "01" THEN
             oput <= b;
        ELSIF sel = "10" THEN
             oput <= c;
        END IF;
    END PROCESS;
END behave;
```

図４は、図３のVHDLコードを回路図で示したものです。

図４　意図しないラッチが生成される記述例を表した回路図



If文やCase文から、それぞれELSE節およびWHEN OTHERS節が抜けていると、１個のラッチが生成されてしまいます。

デフォルト条件に「Don't Care」の指定を行うと、Synplifyでは最も適切なロジックが生成される傾向があります。

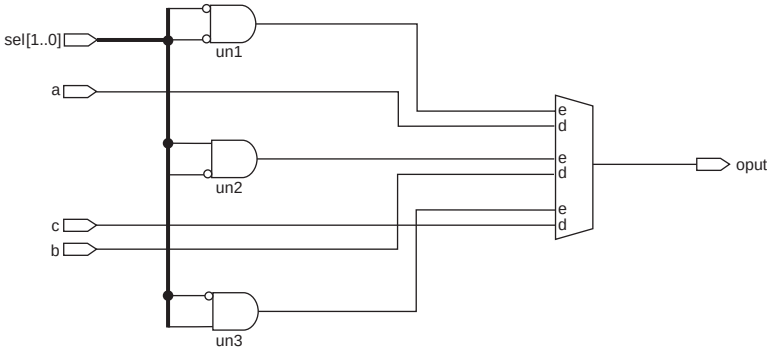
図５は意図しないラッチの生成を防ぐVHDLコードの例を示したものです。

図５　意図しないラッチの生成を防ぐVHDLの記述例

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY good IS
PORT ( a,b,c   : IN STD_LOGIC;
       sel     : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
       oput    : OUT STD_LOGIC);
END good;

ARCHITECTURE behave OF good IS
BEGIN
    PROCESS (a,b,c,sel)
    BEGIN
        IF sel = "00" THEN
             oput <= a;
        ELSIF sel = "01" THEN
             oput <= b;
        ELSIF sel = "10" THEN
             oput <= c;
        ELSE
             oput <= 'x'; -- removes latch
        END IF;
    END PROCESS;
END behave;
```

図６は、図５のVHDL記述を回路図で表したものです。

図６　ラッチの生成を防ぐVHDL記述例を表した回路図



プライオリティ・エンコーディング形式のIf文

デザイン内のクリティカル・パスの遅延を減少させるときに、If文を使用してプライオリティ・エンコーディングを実現させる方法があります。図７に示すVerilog HDLの記述例では、If文を使用してプライオリティ・エンコーディングを実現しています。この例では、クリティカル・パス内でもっとも遅延の大きいsel1にIf文を適用しています。この場合、sel1の優先順位（プライオリティ）が最も高くなります。

図７　プライオリティ・エンコーディング形式のIf文を使用したVerilog HDL記述例

```
module priority (a,b,c,d,sel1,sel2,sel3,sel4,oput);
input a,b,c,d,sel1,sel2,sel3,sel4;
output oput;
always @(a or b or c or d or sel1 or sel2 or sel3 or sel4)
begin
    oput = 1'b0;
    if (sel1)
         oput = a;
    else if (sel2)
         oput = b;
    else if (sel3)
         oput = c;
    else if (sel4)
         oput = d;
end
endmodule
```

ロジック最適化のための"Don't Care"条件

通常、Synplifyソフトウェアは、不定値を "don't care" 条件として扱って、ロジックを最適化します。デザイン内で、デフォルト値に特定のロジックの値を指定する替わりに "don't care" を指定することで、良好なロジックの最適化結果が得られます。

すべての "don't care" 条件は、シミュレーションで検証される必要があります。

図８は、デフォルト条件の値に特定のロジック値を指定し、オプション条件のない方法で回路を構成したVHDLの記述例を示したものです。

図８　VHDLでデフォルト条件の値に特定のロジック値を指定した記述例

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY value IS
    PORT (    data : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
              sel  : IN STD_LOGIC_VECTOR(4 DOWNTO 0);
              dout : OUT STD_LOGIC);
END value;

ARCHITECTURE behave OF value IS
BEGIN
    PROCESS(sel,data)
    BEGIN
       CASE sel IS
              WHEN "00000" => dout <= data(0);
              WHEN "00001" => dout <= data(1);
              WHEN "00010" => dout <= data(2);
              WHEN "00011" => dout <= data(3);
              WHEN "00100" => dout <= data(4);
              WHEN "00101" => dout <= data(5);
              WHEN "00110" => dout <= data(6);
              WHEN "00111" => dout <= data(7);
              WHEN "01000" => dout <= data(8);
              WHEN "01001" => dout <= data(9);
              WHEN "01010" => dout <= data(10);
              WHEN "01011" => dout <= data(11);
              WHEN "01100" => dout <= data(12);
              WHEN "01101" => dout <= data(13);
              WHEN "01110" => dout <= data(14);
              WHEN "01111" => dout <= data(15);
              WHEN OTHERS  => dout <= '0';    -- assigned a logic value.
       END CASE;
    END PROCESS;
END behave;
```

図８に示された記述方法では最適化された結果が得られず、19.6nsの伝搬遅延時間（t_{PD}）が発生します。これに対して、デフォルト条件の値に"don't care" を指定することで、18.3ns のt_{PD}が実現されます（図９を参照）。

図９　VHDLでデフォルト条件の値に "Don't Care" を指定した記述例

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY no_care IS
    PORT (    data :IN STD_LOGIC_VECTOR (15 DOWNTO 0);
              sel  :IN STD_LOGIC_VECTOR(4 DOWNTO 0);
              dout :OUT STD_LOGIC);
END no_care;

ARCHITECTURE behave no_care IS
BEGIN
    PROCESS(sel,data)
    BEGIN
       CASE sel IS
              WHEN "00000" => dout <= data(0);
              WHEN "00001" => dout <= data(1);
              WHEN "00010" => dout <= data(2);
              WHEN "00011" => dout <= data(3);
              WHEN "00100" => dout <= data(4);
              WHEN "00101" => dout <= data(5);
              WHEN "00110" => dout <= data(6);
              WHEN "00111" => dout <= data(7);
              WHEN "01000" => dout <= data(8);
              WHEN "01001" => dout <= data(9);
              WHEN "01010" => dout <= data(10);
              WHEN "01011" => dout <= data(11);
              WHEN "01100" => dout <= data(12);
              WHEN "01101" => dout <= data(13);
              WHEN "01110" => dout <= data(14);
              WHEN "01111" => dout <= data(15);
              WHEN OTHERS   => dout  <= 'x'; -- assigned a “don’t care” value.
       END CASE;
    END PROCESS;
END behave;
```

シーケンシャル・ロジック

ある時間の出力が、その時間の入力とそれ以前のすべての時間の入力で決定される場合、このロジックはシーケンシャル・ロジック（順序回路）となります。すべてのシーケンシャル回路には１個または複数のレジスタ（フリップフロップ）が含まれている必要があります。FLEXデバイスの各LEには１個のDタイプのフリップフロップが内蔵されており、回路をパイプライン化した場合でも追加のリソースは必要になりません。デザイン内のロジックを分割したり、デザイン内に特定の値をストアしたい場合でも、追加のLEや配線リソースが使用されることはありません。

意図しないフィードバック・マルチプレクサが生成されないように注意する必要があります。 If文を使用したときに、可能性のあるすべての入力条件が指定されていないと、フィードバック・マルチプレクサが生成されてしまいます。図10はフィードバック・マルチプレクサが生成されるVHDLの記述例を示したものです。

図10　フィードバック・マルチプレクサを生成するVHDLの記述例

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY seq IS
  PORT( a,b,c,d,clk,rst  : IN STD_LOGIC;
        sel              : STD_LOGIC_VECTOR(3 DOWNTO 0);
        oput             : OUT STD_LOGIC);
END seq;

ARCHITECTURE behave OF seq IS
BEGIN
    PROCESS(clk, rst)
    BEGIN
         IF rst = '1' THEN
               oput <= '1';
         ELSIF clk='1' AND clk'EVENT THEN
               IF sel(0) = '1' THEN
                     oput <= a AND b;
               ELSIF sel(1) = '1' THEN
                     oput <= b;
               ELSIF sel(2) = '1' THEN
                     oput <= c;
               ELSIF sel(3) = '1' THEN
                     oput <= d;
               END IF;
         END IF;
    END PROCESS;
END behave;
```

フィードバック・マルチプレクサが生成されると、図10に示した回路機能には５個のLEが必要になります。この場合、最後にELSE節を使用し、すべてのステートを指定してフィードバックを除去することで、LEの数を減少させることができます。図11で示す実現方法には、２個のLEが必要になるだけです。

図11　フィードバック・マルチプレクサの生成を防ぐVHDLの記述

```
LIBRARY ieee;
USE IEEE.std_logic_1164.ALL;

ENTITY seq2 IS
  PORT( a,b,c,d,clk,rst  : IN STD_LOGIC;
        sel              : STD_LOGIC_VECTOR(3 DOWNTO 0);
        oput             : OUT STD_LOGIC);
END seq2;

ARCHITECTURE behave OF seq2 IS
BEGIN
    PROCESS(clk, rst)
    BEGIN
         IF rst = '1' THEN
               oput <= '1';
         ELSIF clk='1' AND clk'EVENT THEN
               IF sel(0) = '1' THEN
                     oput <= a AND b;
               ELSIF sel(1) = '1' THEN
                     oput <= b;
               ELSIF sel(2) = '1' THEN
                     oput <= c;
               ELSIF sel(3) = '1' THEN
                     oput <= d;
               ELSE
                     oput<= 'x'; -- removes feedback
                                    multiplexer
               END IF;
         END IF;
    END PROCESS;
END behave;
```

ゲート付きクロック

ゲート付きのクロックはロジックの遅延とクロックのスキューを生成し、FLEXデバイス内の追加配線リソースが必要になります。さらに、これによって、ロジック・アレイ・ブロック（LAB）で使用できるクロックの本数が限定されることになります。このため、ゲート付きクロックの使用は可能な限り避けることが必要です。

デザイン内にゲート付きのクロックを使用する場合は、GLOBALのプリミティブを使用して、ゲート付きクロックがファン・アウトの大きい内部のグローバル信号のいずれか１本に配置されようにします。図12は、VHDLのデザイン内にGLOBALのプリミティブを使用してゲート付きクロックを実現した例を示したものです。

図12　VHDLでゲート付きクロックを実現する方法

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY gate IS
  PORT ( a,b       : IN STD_LOGIC;
         c,d       : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
         oput      : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END gate;

ARCHITECTURE behave OF gate IS
SIGNAL clock         : STD_LOGIC;
SIGNAL gclk          : STD_LOGIC;
SIGNAL count         : STD_LOGIC_VECTOR(3 DOWNTO 0);
ATTRIBUTE black_box  : BOOLEAN;

COMPONENT GLOBAL
  PORT ( a_in    : IN STD_LOGIC;
         a_out   : OUT STD_LOGIC);
END COMPONENT;
ATTRIBUTE black_box OF GLOBAL: COMPONENT IS TRUE;

BEGIN
  clock <= a AND b;
  clk_buf: GLOBAL PORT MAP (clock, gclk);

  PROCESS(gclk)
  BEGIN
        IF gclk='1' AND gclk'EVENT THEN
                      count <= c + d;
        END IF;
  END PROCESS;
  oput <= count;
END behave;
```

ステート・マシン

ステート・マシンを含むデザインの性能を改善するためには、すべての演算機能回路やデータ・パスからステート・マシンが分離されるようにすることが必要です。したがって、ステート・マシンは、コントロール・ロジックとして扱う必要があります。

図13は、32ビットのカウンタと８ビットの４対１マルチプレクサと接続されたステート・マシンの例をブロック図で示したものです。このステート・マシンは、デザイン内の他の２つのエレメントに対するコントロール・ロジックも提供しています。

図13　ステート・マシンの例

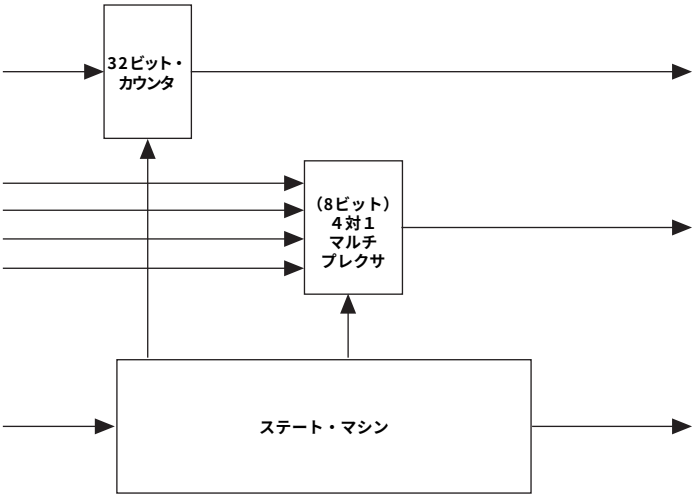


図14は、カウンタとマルチプレクサがステート・マシンの記述に関連しているために、非効率なデザイン・スタイルとなっている例を示したものです。この記述の結果として、１個のアップ／ダウン・カウンタが使用される替わりに、２個のカウンタが参照される可能性があります。また、マルチプレクサには、コントロール信号に関連した追加のロジックが生成される可能性があります。

図14　サンプルのステート・マシンに対する非効率的なVHDL記述例（1/4)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY vhdl_bad IS
  PORT (
     clk   : IN STD_LOGIC;
```

図14　サンプルのステート・マシンに対する非効率的なVHDL記述例（2/4）

```
    rst     : IN STD_LOGIC;
    muxa    : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    muxb    : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    muxc    : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    muxd    : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    c_in    : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    sm_in   : IN STD_LOGIC_VECTOR(4 DOWNTO 0);
    sm_out  : OUT STD_LOGIC_VECTOR(4 DOWNTO 0);
    c_out   : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
    muxout  : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
 );
END vhdl_bad ;

ARCHITECTURE sm OF vhdl_bad IS

    TYPE STATE_TYPE IS (s0, s1, s2, s3, s4, s5, s6, s7);
    SIGNAL state: STATE_TYPE;
    SIGNAL count     : STD_LOGIC_VECTOR(31 DOWNTO 0);

BEGIN
    c_out <= count;
    PROCESS (clk, rst)
    BEGIN
      IF (rst = '1') THEN
          state <= s0;
          sm_out  <= "00000";
          count   <= (OTHERS => '0');
          muxout  <= (OTHERS => '0');
      ELSIF (clk'EVENT AND clk = '1') THEN
          muxout  <= muxa;
          CASE state IS
            WHEN s0 =>
              IF (sm_in(4) = '1' AND sm_in(3) = '1'AND
              sm_in(1)='1' AND sm_in(0) = '1')THEN
                  state  <= s6;
                  sm_out <= "00110";
              ELSIF (sm_in(1) = '1') THEN
                  state  <= s5;
                  sm_out <= "01110";
                  count <= count + "1";
              ELSE
                  state  <= s0;
                  sm_out <= "11110";
                  muxout <= muxb;
              END IF;
            WHEN s1 =>
```

図14　サンプルのステート・マシンに対する非効率的なVHDL記述例（3/4）

```
    IF (sm_in(4) = '1' AND sm_in(3) = '1' AND
    sm_in(1) ='1' AND sm_in(0) = '1') THEN
        state  <= s7;
        sm_out <= "11111";
    ELSIF (sm_in(4) = '1' AND sm_in(1) = '1') THEN
        state  <= s4;
        sm_out <= "10110";
        count  <= c_in;
        muxout <= muxc;
    ELSIF (sm_in(3) = '1') THEN
        state  <= s3;
        sm_out <= "00011";
    ELSIF (sm_in(4) = '1') THEN
        state  <= s2;
        sm_out <= "01101";
        muxout <= muxd;
    ELSE
        state  <= s1;
        sm_out <= "00101";
    END IF;
  WHEN s2 =>
    IF (sm_in(4) = '1' AND sm_in(2) = '1' AND
    sm_in(0) = '1') THEN
        state  <= s3;
        sm_out <= "11111";
    ELSIF (sm_in(0) = '1') THEN
        state  <= s2;
        sm_out <= "11010";
        muxout <= muxb;
    ELSIF (sm_in(4) = '1' AND sm_in(3) = '1' AND
    sm_in(1) = '1') THEN
        state  <= s1;
        sm_out <= "01010";
        count  <= count - "1";
    ELSIF (sm_in(4) = '1' AND sm_in(1) = '1') THEN
        state  <= s4;
        sm_out <= "10011";
    ELSE
        state  <= s2;
        sm_out <= "10100";
    END IF;
  WHEN s3 =>
    IF (sm_in(3) = '1') THEN
        state  <= s2;
        sm_out <= "01011";
```

図14　サンプルのステート・マシンに対する非効率的なVHDL記述例（4/4）

```
                muxout <= muxc;
              ELSE
                state  <= s3;
                sm_out <= "10100";
              END IF;
            WHEN s4 =>
              IF (sm_in(4) = '1' AND sm_in(3) = '1' AND
              sm_in(2)   ='1' AND sm_in(0) = '1') THEN
                state  <= s2;
                sm_out <= "11101";
              ELSE
                state  <= s4;
                sm_out <= "00111";
                count  <= c_in;
              END IF;
            WHEN s5 =>
              IF (sm_in(3) = '1' AND sm_in(1) = '1' AND
              sm_in(0) ='1') THEN
                state  <= s2;
                sm_out <= "00110";
              ELSIF (sm_in(4) = '1' AND sm_in(3) = '1') THEN
                state  <= s7;
                sm_out <= "00001";
              ELSIF (sm_in(4) = '1' AND sm_in(3) = '1' AND
              sm_in(2) = '1' AND sm_in(1) = '1' AND
              sm_in(0) = '1')THEN
                state  <= s3;
                sm_out <= "10100";
                muxout <= muxd;
              ELSE
                state  <= s5;
                sm_out <= "00101";
                count  <= count + "1";
              END IF;
            WHEN s6 =>
                state  <= s1;
                sm_out <= "10011";
            WHEN s7 =>
                state  <= s7;
                sm_out <= "10011";
            END CASE;
          END IF;
      END PROCESS;
END sm;
```

図15は、図14に示されたステート・マシンのデザインをより効率的な記述にしたものです。この記述例では、カウンタとマルチプレクサの機能がステート・マシンとは別個の処理で実現されるようになっています。

図15　サンプルのステート・マシンに対する効率的なVHDL記述例（1/5）

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY vhdl_good2 IS
  PORT (
        clk    : IN  STD_LOGIC;
        rst    : IN  STD_LOGIC;
        muxa   : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
        muxb   : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
        muxc   : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
        muxd   : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
        c_in   : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
        sm_in  : IN  STD_LOGIC_VECTOR(4 DOWNTO 0);
        sm_out : OUT STD_LOGIC_VECTOR(4 DOWNTO 0);
        c_out  : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
        muxout : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
        );
END vhdl_good2;

ARCHITECTURE sm OF vhdl_good2 IS

    TYPE STATE_TYPE IS (s0, s1, s2, s3, s4, s5, s6, s7);
    SIGNAL state STATE_TYPE;
    SIGNAL count    : STD_LOGIC_VECTOR(31 DOWNTO 0);
    SIGNAL count_en : STD_LOGIC;
    SIGNAL count_id : STD_LOGIC;
    SIGNAL count_ud : STD_LOGIC;
    SIGNAL muxel    : STD_LOGIC_VECTOR(1 DOWNTO0);

BEGIN
multiplexer:PROCESS(muxsel,muxa,muxb,muxc,muxd)

BEGIN
    CASE muxsel IS
       WHEN "00" =>
         muxout <= muxa;
       WHEN "01" =>
         muxout <= muxb;
       WHEN "10" =>
         muxout <= muxc;
```

図15　サンプルのステート・マシンに対する効率的なVHDL記述例（2/5）

```
        WHEN "11" =>
          muxout <= muxd;
        WHEN OTHERS =>
          muxout <= muxa;
    END CASE;
END PROCESS;

count: PROCESS (clk)
    VARIABLE direction : INTEGER;
    BEGIN
       IF (count_ud = '1') THEN
         direction := 1;
       ELSE
         direction := -1;
       END IF;
       IF rst = '1' THEN
         count <= "00000000000000000000000000000000";

       ELSIF (clk'EVENT AND clk = '1') THEN
            IF count_ld = '0' THEN
               count <= c_in;
            ELSE
               IF count_en = '1' THEN
                    count <= count + direction;
               END IF;
            END IF;
         END IF;

         c_out <= count;
    END PROCESS;

    PROCESS (clk, rst)
    BEGIN
      IF (rst = '1') THEN
         state <= s0;
         sm_out <= "00000";
         muxsel <= "00";
         count_ld <= '0';
         count_en <= '0';
         count_ud <= '0';
      ELSIF (clk'EVENT AND clk = '1') THEN
         count_ld <= '0';
         count_en <= '0';
         count_ud <= '1';
```

図15　サンプルのステート・マシンに対する効率的なVHDL記述例（3/5）

```
        muxsel    <= "00";
        CASE state IS
          WHEN s0 =>
            IF (sm_in(4) = '1' AND sm_in(3) = '1' AND
            sm_in(1) ='1' AND sm_in(0) = '1') THEN
              state  <= s6;
              sm_out <= "00110";
            ELSIF (sm_in(1) = '1') THEN
              state  <= s5;
              sm_out <= "01110";
              count_en <= '1';
            ELSE
              state  <= s0;
              sm_out <= "11110";
              muxsel <= "01";
            END IF;
          WHEN s1 =>
            IF (sm_in(4) = '1' AND sm_in(3) = '1' AND
            sm_in(1) ='1' AND sm_in(0) = '1') THEN
              state  <= s7;
              sm_out <= "11111";
            ELSIF (sm_in(4) = '1' AND sm_in(1) = '1') THEN
              state  <= s4;
              sm_out <= "10110";
              count_ld <= '1';
              muxsel <= "10";
            ELSIF (sm_in(3) = '1') THEN
              state  <= s3;
              sm_out <= "00011";
            ELSIF (sm_in(4) = '1') THEN
              state  <= s2;
              sm_out <= "01101";
              muxsel <= "11";
            ELSE
              state  <= s1;
              sm_out <= "00101";
            END IF;
          WHEN s2 =>
            IF (sm_in(4) = '1' AND sm_in(2) = '1' AND
            sm_in(0) ='1') THEN
              state  <= s3;
              sm_out <= "11111";
            ELSIF (sm_in(0) = '1') THEN
              state  <= s2;
              sm_out <= "11010";
```

図15　サンプルのステート・マシンに対する効率的なVHDL記述例（4/5）

```
      muxsel <= "01";
    ELSIF (sm_in(4) = '1' AND sm_in(3) = '1' AND
    sm_in(1) = '1') THEN
      state  <= s1;
      sm_out <= "01010";
      count_en <= '1';
      count_ud <= '0';
    ELSIF (sm_in(4) = '1' AND sm_in(1) = '1') THEN
      state  <= s4;
      sm_out <= "10011";
    ELSE
      state  <= s2;
      sm_out <= "10100";
    END IF;
  WHEN s3 =>
    IF (sm_in(3) = '1') THEN
      state  <= s2;
      sm_out <= "01011";
      muxsel <= "10";
    ELSE
      state  <= s3;
      sm_out <= "10100";
    END IF;
  WHEN s4 =>
    IF (sm_in(4) = '1' AND sm_in(3) = '1' AND
    sm_in(2) ='1' AND sm_in(0) = '1') THEN
      state  <= s2;
      sm_out <= "11101";
    ELSE
      state  <= s4;
      sm_out <= "00111";
      count_ld <= '1';
    END IF;
  WHEN s5 =>
    IF (sm_in(3) = '1' AND sm_in(1) = '1' AND
    sm_in(0) ='1') THEN
      state  <= s2;
      sm_out <= "00110";
    ELSIF (sm_in(4) = '1' AND sm_in(3) = '1') THEN
      state  <= s7;
      sm_out <= "00001";
    ELSIF (sm_in(4) = '1' AND sm_in(3) = '1' AND
    sm_in(2) = '1' AND sm_in(1) = '1' ANDD
    sm_in(0) = '1')THEN
      state  <= s3;
      sm_out <= "10100";
```

図15　サンプルのステート・マシンに対する効率的なVHDL記述例（5/5）

```
              muxsel <= "11";
            ELSE
              state  <= s5;
              sm_out <= "00101";
              count_en <= '1';
            END IF;
          WHEN s6 =>
              state  <= s1;
              sm_out <= "10011";
          WHEN s7 =>
              state  <= s7;
              sm_out <= "10011";
        END CASE;
      END IF;
    END PROCESS;
END sm;
```

ワン・ホット・エンコーディングでは、１ステートに１ビットのレジスタが使用されるため多くのステート・ビットが必要になりますが、デコーディングに要求されるロジックは減少します。ターゲット・デバイスがFLEX 10Kのときにステート・マシンの最適化を行う場合は、ワン・ホット・エンコーディングを使用してください。FLEXデバイスは多数のレジスタを内蔵したアーキテクチャとなっており、FLEXデバイスのルック・アップ・テーブル（LUT）をベースにしたアーキテクチャはファン・インの小さいデコーディング・ロジックの構成に最適となっています。ワン・ホット・エンコーディングの採用により、結果的にこれらのデバイスの使用効率と性能が改善されます。

エンベデッド・アレイ・ブロック（EAB）にステート・マシンを構成する方法

FLEX 10Kデバイスでステート・マシンの性能を向上させる他のデザイン・テクニックとして、ステート・マシンをEABに配置する方法があります。ステート・マシンをEABに配置するときのガイドラインを下記に示します。

- 限定されたI/O数の非常に複雑なステート・マシンは、EABに実現するのが理想的である。
- ステート・マシンには、必ずラッチが含まれないようにする。
- ステート・マシンは下位レベルのファイルに配置し、上位レベルへフィードバックする。
- 大規模なステート・マシンは、複数の小規模なステート・マシンに分割する。
- 入力はすべてレジスタを通るか、またはすべて通らないようにし、出力もすべてレジスタを通るか、またはすべて通らないようにする。
- 最良の結果を得るためには、EABのレジスタを使用する。

- ステート数と出力の本数は、１個のEABに構成されるステート・マシンに対する入力の本数を基準にする。

MAX+PLUS IIのソフトウェア・オプションを指定したり、Synplifyソフトウェアによる合成時に合成の属性を指定することによって、ステート・マシンをEABに実現させることができます。

Synplifyでは、ステート・マシンをEABに実現させるときに、下記のコードを使用します。

VHDL:

```
ATTRIBUTE altera_implement_in_eab : BOOLEAN;
ATTRIBUTE altera_implement_in_eab OF U1: LABEL IS TRUE;
```

ここで、U1がEABに実現されるファンクションになります。

Verilog HDL:

```
sqrtb sq (.z(sqa), .a(a)) /* synthesis
altera_implement_in_eab=1 */;
```

ここで、sqrtbがEABに実現されるモジュールになります。

図16から図19は、１個のEABに実現されるステート・マシンのVHDL記述を示したもので、このステート・マシンは下記のような属性を持っています。

- レジスタはLEに実現される。
- ステート・マシンには、ステート・マシンのコントロール・ロジックが含まれている。
- トップ・レベルのモジュールでレジスタをインスタンス化し、ステート・マシンがフィードバックを提供しているため、ステート・マシンがFLEX 10Kの１個のEABに実現される。

図16には、ステート・マシンの各ステートが規定されています。

図16　my_pack.vhdのVHDL記述

```
PACKAGE my_pack IS
    TYPE STATE_TYPE IS (s0, s1, s2, s3, s4, s5, serror);
END my_pack;
```

図17は、LEに実現されるレジスタの記述を示したものです。

図17 register.vhdのVHDL記述

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.my_pack.ALL;

ENTITY registers IS
PORT ( next_state    : IN STATE_TYPE;
       temp_out      : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
       clock, reset  : IN STD_LOGIC;
       out_state     : OUT STATE_TYPE;
       result        : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END registers;

ARCHITECTURE behave OF registers IS
BEGIN
    PROCESS (clock, reset)
    BEGIN
       IF (reset = '0') THEN
              result <= "0000";
       ELSIF (clock = '1' AND clock'EVENT) THEN
              result <= temp_out;
              out_state <= next_state;
       END IF;
    END PROCESS;
END behave;
```

図18は、フィードバックのない下位レベルのステート・マシンのデザイン、maclogic.vhdのVHDL記述を示したものです。

図18 maclogic.vhdのVHDL記述（1/3）

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.my_pack.ALL;

ENTITY maclogic IS
PORT ( input         : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
       in_state      : IN STATE_TYPE;
       next_state    : OUT STATE_TYPE;
       temp_out      : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
```

図18 maclogic.vhdのVHDL記述（2/3）

```
       clock, reset : IN STD_LOGIC);
END maclogic;

ARCHITECTURE behave OF maclogic IS

BEGIN
       PROCESS(in_state,input)
       BEGIN
           temp_out <="0000";

           CASE in_state IS
               WHEN s0 =>
                   IF (input = "000") THEN
                           next_state <= s1;
                   ELSE
                           next_state <= s0;
                   END IF;
                   temp_out <= "0000";
               WHEN s1 =>
                   IF (input = "001") THEN
                           next_state <= s2;
                   ELSE
                           next_state <= s1;
                   END IF;
                   temp_out <= "0001";
               WHEN s2 =>
                   IF (input = "010") THEN
                           next_state <= s3;
                   ELSE
                           next_state <= s2;
                   END IF;
                   temp_out <= "0010";
               WHEN s3 =>
                   IF (input = "011") THEN
                           next_state <= s4;
                   ELSE
                           next_state <= s3;
                   END IF;
                   temp_out <= "0011";
               WHEN s4 =>
                   IF (input = "100") THEN
                           next_state <= s5;
                   ELSE
                           next_state <= s4;
                   END IF;
                   temp_out <= "0100";
```

図18 maclogic.vhdのVHDL記述（3/3）

```
            WHEN s5 =>
                IF (input = "101") THEN
                        next_state <= s0;
                ELSE
                        next_state <= s5;
                END IF;
                temp_out <= "0101";
            WHEN serror =>
                IF (input = "111") THEN
                        next_state <= s0;
                ELSE
                        next_state <= serror;
                END IF;
                temp_out <= "1111";
            WHEN OTHERS =>
                        next_state <= serror;
                temp_out <= "1111";
       END CASE;
   END PROCESS;

END behave;
```

図19は、ステート・マシンのフィードバックを提供するトップ・レベルのファイル、mac.vhdのVHDL記述を示したものです。このmac.vhdのファイルは２つのデザイン・エレメントを構造的にリンクさせたものとなっています。

図19 mac.vhdのVHDL記述（1/2）

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.my_pack.ALL;

ENTITY mac IS
PORT(  clock,reset  : IN STD_LOGIC;
       input        : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
       outputs      : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END mac;

ARCHITECTURE behave OF mac IS
       COMPONENT maclogic
       PORT( input        : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
             temp_out     : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
             in_state     : IN STATE_TYPE;
             next_state   : OUT STATE_TYPE;
             clock, reset : STD_LOGIC);
```

図19 mac.vhdのVHDL記述（2/2）

```
        END COMPONENT;
        ATTRIBUTE altera_implement_in_eab      : BOOLEAN;
        ATTRIBUTE altera_implement_in_eab OF U1: LABEL IS TRUE;

        COMPONENT registers
        PORT(next_state     : IN STATE_TYPE;
             temp_out       : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
             clock, reset   : IN STD_LOGIC;
             out_state      : OUT STATE_TYPE;
             result         : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
        END COMPONENT;
        SIGNAL temp         : STATE_TYPE;
        SIGNAL tempstate    : STATE_TYPE;
        SIGNAL temp_result  : STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN
        u1 : maclogic
        PORT MAP(input => input, temp_out => temp_result, next_state =>
                 tempstate, in_state => temp, clock => clock, reset => reset);
        u2 : registers
        PORT MAP( next_state => tempstate, temp_out => temp_result, clock =>
                  clock, reset => reset, out_state => temp, result => outputs);
END behave;
```

Synplifyソフトウェアでの FLEX 10Kデバイスに対するデザイン・テクニック

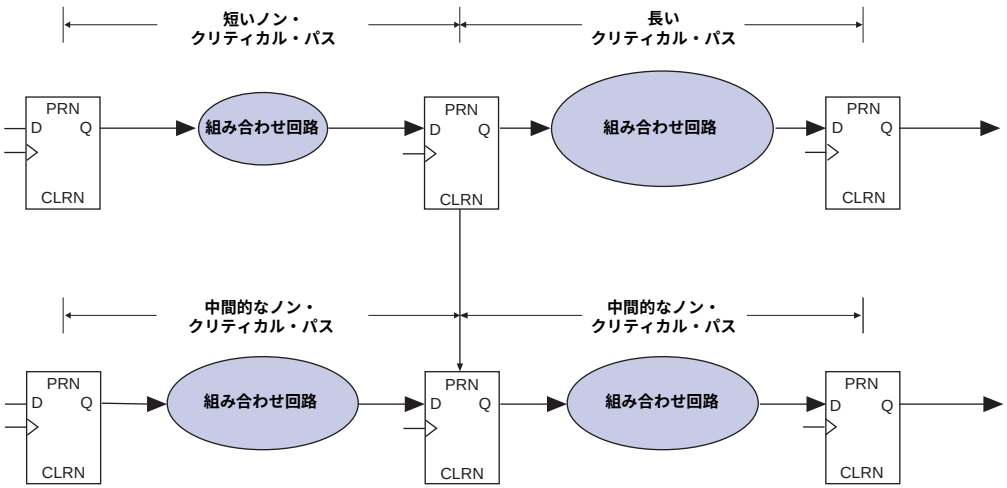
下記のデザイン・テクニックを使用することによって、FLEX 10Kデバイスの性能を最適化することができます。

- レジスタ・バランシング
- パイプライン化
- ロジックの複製
- LPMファンクション

レジスタ・バランシング

レジスタ・バランシングは長いパスの遅延を減少させたり、短いパスの遅延を増加させるときに使用されるテクニックです。このテクニックは、レジスタがレイテンシの目的だけに使用されるようなアプリケーションに有効です。図20を参照してください。

図20　レジスタ・バランシング

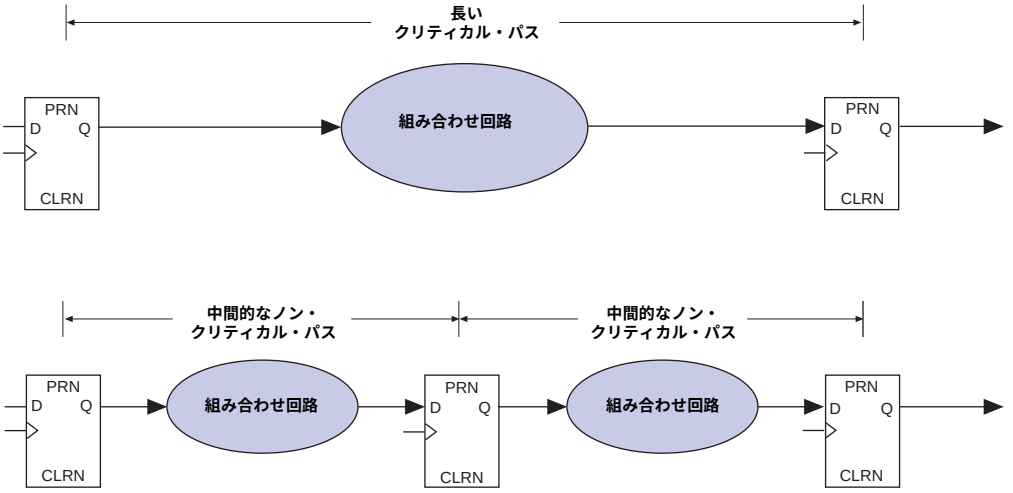


パイプライン化

パイプライン化はロジック値の保持に組み合わせ回路によるラッチではなく、レジスタを使用するテクニックです。デザインをパイプライン化することで、大規模な組み合わせ回路の遅延が追加のレジスタで分割されることになります。FLEX 10Kのアーキテクチャは LUTと接続されたレジスタを多数内蔵しているため、組み合わせ回路をパイプライン化した場合でも、一般的に追加のデバイス・リソースは必要になりません。したがって、レジスタを使用してパイプライン化を実現することで、デバイス内のLEの使用率を増加させることなく、性能を向上させることができます。図21を参照してください。

パイプライン化を実現した場合は、各レジスタがレイテンシを１段増加させることになります（カウンタでは、レイテンシの１段分に対して１クロック・サイクル前にデコードが行われる）。

図21　デザインのパイプライン化



ロジックの複製

ロジックの複製化によりファン・アウトを減少させるテクニックは、与えられたパスでのデザイン性能を改善させるための良好な手法となります。ロジックの複製化を実現することによって、信号がドライブする負荷の数が減少し、配線が容易になる可能性が高いため、ファン・アウトの大きなノードやフリップフロップに対しては、このロジックの複製化のテクニックが効果的です。

Synplifyソフトウェアでは、syn_preserveという属性の指定を行うことができるようになっています。この指定によって、Synplifyソフトウェアによる合成でレジスタが保護されるようにし、これに関係した部分が最適化されないようにすることができます。

Synplifyソフトウェアでsyn_preserveの指定を行うときは、下記の記述を行います。

VHDL:

```
ATTRIBUTE syn_preserve : BOOLEAN;
ATTRIBUTE syn_preserve OF signal_name : SIGNAL IS TRUE;
```

Verilog HDL:

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

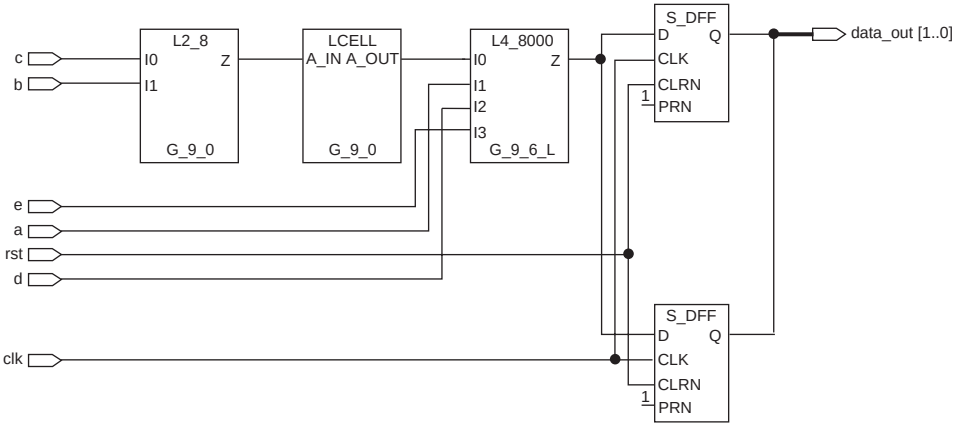
図22はロジックの複製を行うために、syn_preserveを使用した記述例を示したものです。

図22　ロジックの複製化を行うためのsyn_preserve**の使用例**

```
ENTITY split IS
    PORT(clk       : IN STD_LOGIC;
         a,b,c,d,e : IN STD_LOGIC;
         rst       : IN STD_LOGIC;
         data_out  : OUT STD_LOGIC_VECTOR(1 DOWNTO 0));
END split;

ARCHITECTURE behave OF split IS
    SIGNAL inter                    : STD_LOGIC_VECTOR(1 DOWNTO 0);
    ATTRIBUTE syn_preserve          : BOOLEAN;
    ATTRIBUTE syn_preserve OF inter : SIGNAL IS TRUE;
BEGIN
    reg: PROCESS(clk,rst)
        BEGIN
                      IF rst = '0' THEN
                           inter <= "00";
                      ELSIF (clk = '1' AND clk'EVENT) THEN
                           inter(0) <= (a AND b AND c AND d AND e);
                           inter(1) <= (a AND b AND c AND d AND e);
                      END IF;
        END PROCESS;
    data_out(0) <= inter(0); -- the registers inter(0) and inter(1) are
    data_out(1) <= inter(1);    duplicates.

END behave;
```

図23は、図22で示したVHDLの記述を回路図で表したものです。

図23　ロジックの複製化を行うためのロジック保護を行った記述の回路図



ここで、図24に示されるように、syn_preserveの属性が指定されなかった場合は、レジスタの１個が最適化され、削除されます。

図24　syn_preserveを使用しない記述（1/2）

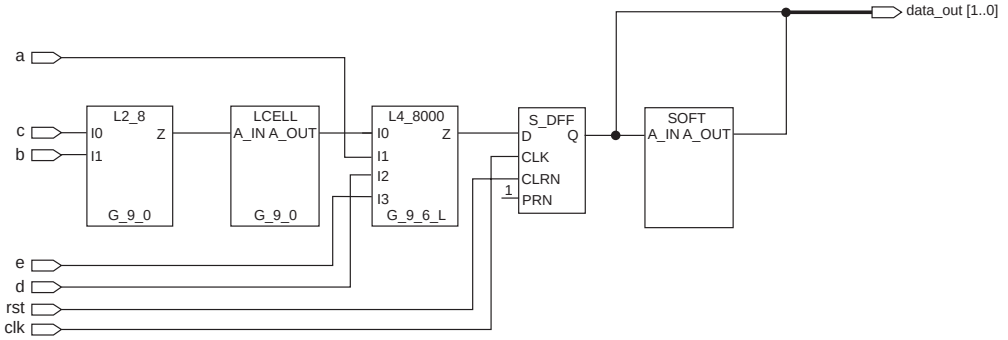
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY split3 IS
    PORT( clk       : IN STD_LOGIC;
          a,b,c,d,e : IN STD_LOGIC;
          rst       : IN STD_LOGIC;
          data_out  : OUT STD_LOGIC_VECTOR(1 DOWNTO 0));
END split3;

ARCHITECTURE behave OF split3 IS
SIGNAL inter: STD_LOGIC_VECTOR(1 DOWNTO 0);

BEGIN
    reg: process(clk,rst)
        BEGIN
                      IF rst = '0' THEN
                           inter <= "00";
```

図24　syn_preserveを使用しない記述（2/2）

```
                    ELSIF (clk = '1' AND clk'EVENT) THEN
                          inter(0) <= (a AND b AND c AND d AND e);
                          inter(1) <= (a AND b AND c AND d AND e);
                    END IF;
       END PROCESS;
   data_out(0) <= inter(0);
   data_out(1) <= inter(1);
END behave;
```

図25は、ロジックの複製化が行われない記述を回路図で示したものです。

図25　ロジックの複製化が行われない記述の回路図



LPMファンクション

LPM（Library of Parameterized Modules）のファンクションは、異なるサイズのポートやパラメータの指定を行うことで、各アプリケーションに簡単に最適化できる大規模なビルディング・ブロックとなっています。アルテラは、LPMファンクションをアルテラのデバイス・アーキテクチャに最適化しています。

Synplifyソフトウェアでは、LPMファンクションがブラック・ボックスとしてコンパイルされます。各パラメータの値は、Verilog HDLのメタ・コメント、またはVHDLの属性として表現され、EDIFファイルまたはテキスト・デザイン・ファイル（.tdf）に受け渡されます。

図26は、lpm_multファンクションを使用して、２段のパイプラインを持ったマルチプライヤを作成したVHDLのデザインを示したものです。

図26　VHDLでLPMファンクションをインスタンス化した記述例（1/2）

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY mult IS
PORT( a     : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
      b     : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
      clock : IN STD_LOGIC;
      p     : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END mult;

ARCHITECTURE a OF mult IS

COMPONENT my_mult

    PORT ( dataa  : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
           datab  : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
           aclr   : IN STD_LOGIC := '0';
           clock  : IN STD_LOGIC := '0';
           sum    : IN STD_LOGIC_VECTOR(7 DOWNTO 0) := (OTHERS => '0');
           result : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END COMPONENT;

ATTRIBUTE black_box     : BOOLEAN;
ATTRIBUTE lpm_widtha    : INTEGER;
ATTRIBUTE lpm_widthb    : INTEGER;
ATTRIBUTE lpm_widths    : INTEGER;
ATTRIBUTE lpm_widthp    : INTEGER;
ATTRIBUTE lpm_pipeline  : INTEGER;
ATTRIBUTE lpm_type      : STRING;

-- assign the appropriate attribute values

ATTRIBUTE black_box OF my_mult    : COMPONENT IS TRUE;
ATTRIBUTE lpm_widtha OF my_mult   : COMPONENT IS 4;
ATTRIBUTE lpm_widthb OF my_mult   : COMPONENT IS 4;
ATTRIBUTE lpm_widths OF my_mult   : COMPONENT IS 8;
ATTRIBUTE lpm_widthp OF my_mult   : COMPONENT IS 8;
ATTRIBUTE lpm_pipeline OF my_mult : COMPONENT IS 2;
ATTRIBUTE lpm_type OF my_mult     : COMPONENT IS "LPM_MULT";

SIGNAL tmp_result: STD_LOGIC_VECTOR(7 DOWNTO 0);
```

図26　VHDLでLPMファンクションをインスタンス化した記述例（2/2）

```
BEGIN
    u1: my_mult
         PORT MAP (dataa => a(3 DOWNTO 0), datab => b(3 DOWNTO 0), clock
         => clock, result => tmp_result);
    PROCESS(clock)
    BEGIN
         IF (clock'EVENT AND clock ='1') THEN
              p <= tmp_result;
         END IF;
    END PROCESS;
END a;
```

図27は、lpm_ram_dqファンクションを使用して、64×16のRAMを実現したVerilog HDLのデザインを示したものです。

図27　Verilog HDLでLPMファンクションをインスタンス化した記述例

```
// lpm ram example
//
// define the black box here (pick a name, such as myram_64x16),
// note that immediately after the port list, but before the semicolon ';'
// the black_box synthesis directive is included with the lpm_type specified
// as "lpm_ram_dq", and other parameter values are set.

module myram_64x16 (data,address,inclock,
    outclock,we,q) /* synthesis black_box lpm_width=16 lpm_widthad=6
    lpm_type="lpm_ram_dq" */ ;

    input [15:0] data;
    input [5:0] address;
    input inclock, outclock;
    input we;
    output [15:0] q;
    endmodule

// instantiate the lpm function in the
// higher-level module myram,
module myram(clock, we, data, address, q);
    input clock, we;
    input [15:0] data;
    input [5:0] address;
    output [15:0] q;

myram_64x16 inst1 (data,address,clock, clock, we, q);
endmodule
```

LPMファンクションに対するMAX+PLUS IIのサポート状況は、MAX+PLUS IIのヘルプ機能を使用して確認することができます。

Synplifyソフトウェアの設定

このセクションでは、アルテラのデバイスをターゲットにしたファイルを処理するときに、Synplifyソフトウェアに最適となる設定について解説します。

ロジックのLCELLへのマッピング

「*Map Logic to LCELLs*」のオプションをオンに設定すると、Synplifyソフトウェアはロジックを複数のLUT、キャリー・チェイン、カスケード・チェインにマッピングし、デザインを性能に対して最適化します。

ただし、このオプションをオフに設定することによって、デバイスの使用効率が改善されるため、MAX+PLUS IIがデザインをフィッティングできなかった場合は、この設定をオフにする必要があります。

クリークの設定

「*Perform Cliquing*」のオプションは、「*Map Logic to LCELLs*」のオプションがオンに設定されたときで、出力ネットリストがEDIFに設定されているときにのみ選択可能になっています。この設定を行うことによって、Synplifyソフトウェアがクリティカル・パスに対して自動的にクリークを適用するようになります。Synplifyソフトウェアから、クリークがEDIFのネットリスト出力ファイルに定義されます。なお、このオプションはTDFには提供されていません。

自動クリーキングの指定により、性能がクリティカルなパスの配置配線がコントロールされます。これによって、平均で性能が10%から15%まで改善されます。

高性能を実現するMAX+PLUS IIのオプション

Synplifyソフトウェアによる論理合成が完了すると、生成されたEDIFのネットリスト・ファイルをMAX+PLUS IIに取り込むことができます。以下のセクションでは、推奨される合成スタイルと最適化された性能を得るために使用可能なMAX+PLUS IIのその他のオプションについて解説します。

合成スタイル

Synplifyソフトウェアで「*Map Logic to LCELLs*」のオプションがオンに設定されている場合は、Synplifyがデザインを自動的にFLEX 10Kのアーキテクチャに最適化します。このため、アルテラはMAX+PLUS II ソフトウェアで*WYSIWYG*の論理合成スタイルを使用することを推奨します（*NOT-Gate Push-back*のオプションはオフに設定）。このスタイルで要求される性能が得られなかった場合は、*FAST*の合成スタイルを選択してみてください。一般的には、*WYSIWYG*の論理合成スタイルで最高の結果が得られますが、場合によっては*FAST*の論理合成スタイルでより良い結果が得られることがあります。

MAX+PLUS IIのコンパイレーション・オプションの設定は、以下の手順で行ってください。

1. MAX+PLUS IIのソフトウェアを起動します。
2. Project Name（Fileメニュー）を選択します。Project Nameのダイアログ・ボックスで、Synplifyが生成したEDIFファイル、<ワーキング・ディレクトリ>/<プロジェクト名>.edfを選択し、OKをクリックします。

アルテラはSynplifyが生成したEDIFファイルをHDLのソース・ファイルと別のディレクトリにストアすることを推奨します。

3. Compiler（MAX+PLUS IIメニュー）を選択します。
4. EDIF Netlist Reader Settingsのダイアログ・ボックス（Interfacesメニュー）で、vendorとして*Synplicity*を選択します。
5. Deviceのダイアログ・ボックス（Assignメニュー）で適切なデバイスを選択し、OKをクリックします。
6. 適切なネットリスト・ライタのコマンドをオンに設定します（Interfacesメニュー）。VHDL出力ファイル（.vho）を作成したい場合は、VHDL Netlist Writerのコマンドをオンに設定します。
7. 適切なネットリスト・ライタに対する設定をオンにします（Interfacesメニュー）。VHDL出力ファイル（.vho）を作成したい場合は、*VHDL Output File [.vho]*のオプションをオンに設定します。OKをクリックします。

8. Global Project Logic Synthesis（Assignメニュー）を選択します。Global Project Logic Synthesisのダイアログ・ボックスで、 SynplifyのEDIFファイルでLE にマッピングされているFLEXデバイスのデザインに対する合成スタイルを*WYSIWYG*に設定します。Define Synthesis Styleをクリックします。Define Synthesis Styleのダイアログ・ボックスで、Advanced Optionsをクリックします。Advanced Optionsのダイアログ・ボックスで、「*NOT-Gate Push-back*」のオプションをオフに設定します。OKを３回クリックしてダイアログ・ボックスをクローズします。

9. Startボタンをクリックして、MAX+PLUS IIのコンパイラを起動します。

デザインの作成方法、 MAX+PLUS IIによるコンパイル方法の詳細については、MAX+PLUS II ACCESS℠ Key GuidelinesのSynplicity Synplify softwareを参照してください。

高速I/Oロジック・オプションの使用

FLEX 10KデバイスはI/Oピンの近傍にレジスタが内蔵されています。これらのIOE（Input/Output Element）レジスタは、デバイス内部に埋め込まれているレジスタよりも高速の「clock-to-output」遅延（t_{CO}）を提供します。

*Fast I/O*のロジック・オプションの設定により、これらのレジスタの使用が可能になります。このオプションの設定は、LEのレジスタを使用するか、入力またはI/Oピンとのダイレクトな高速接続を提供するIOEのレジスタを使用するかをMAX+PLUS IIのコンパイラに指示します。*Fast I/O*のオプションをオンに設定することによって、高速のセットアップ・タイムが実現されるため、タイミング性能を最大にすることが可能になります。

*Fast I/O*のロジック・オプションは、下記のように各ピンに適用することができます。

- 入力ピンでは、MAX+PLUS IIのコンパイラが入力と接続されるレジスタをIOEまたはLEに移動させます。
- 出力ピンでは、MAX+PLUS IIのコンパイラが出力と接続されるレジスタをI/Oセルまたロジック・セルに移動させます。

下記の手順の設定を行うことによって、MAX+PLUS IIのコンパイラはプロジェクト全体に対して*Fast I/O*を指定するようになります。

1. Global Project Logic Synthesis（Assignメニュー）を選択します。

2. Global Project Logic Synthesisのダイアログ・ボックスで、「*Automatic Fast I/O* 」のオプションをオンに設定し、OKをクリックします。

各レジスタごとに*Fast I/O*の指定を行う場合は、下記の手順で行います。

1. Logic Options（Assignメニュー）を選択します。Logic Optionsのダイアログ・ボックスで、レジスタのノード名を入力し、Individual Logic Optionsをクリックします。
2. Individual Logic Optionsのダイアログ・ボックスで、「*Fast I/O*」のオプションをオンに設定します。
3. 該当するダイアログ・ボックスでOKを２回クリックして、変更した内容をセーブします。
4. Startボタンをクリックして、MAX+PLUS IIのコンパイラにコンパイルを開始させます。

タイミング・ドリブン・コンパイレーション

タイミング・ドリブン・コンパイレーションを使用することで、デバイス・リソースの使用率に応じて、デバイス性能が平均で15%から30%まで改善されます。

一般的に、より高いリソース使用率となっているデバイスには、コンパイルに長い時間がかかります。リソースがフルに使用されている場合でも、MAX+PLUS IIのソフトウェアは、すべてのタイミング要求を満足させながら、デザインをフィッティングさせようとします。このようなリソースをフルに使用したデザインでは、コンパイル時間が10倍まで増加することがあります。

プロジェクトに対して、表１に示す４種類のタイミング・コンストレイントを指定することができます。

表１　タイミング・ドリブン・コンパイレーションに使用されるタイミング・コンストレイント

パラメータ	定　　義	MAX+PLUS IIによる実現
t_{PD}	t_{PD}（入力からレジスタなし出力までの遅延）は、入力ピンからの信号が組み合わせ回路を通って外部の出力ピンに現れるまでに必要な時間。	要求されるt_{PD}をプロジェクト全体、または任意の入力ピン、出力ピン、出力ピンと接続されるTRIバッファに対して指定することができる。
t_{CO}	t_{CO}（「clock-to-output」遅延）は、許容される「clock-to-output」の最大遅延時間を規定する。レジスタのクロックとなる入力ピンからレジスタと接続された出力ピンまでの遅延（クロック信号の遷移後の遅延）。この時間は常に外部ピン間遅延として表われる。	要求されるt_{CO}をプロジェクト全体、または任意の入力ピン、出力ピン、出力ピンと接続されるTRIバッファに指定することができる。
t_{SU}	t_{SU}（クロックのセットアップ・タイム）は、クロック・ピンにレジスタのクロック信号がアサートされるまで、レジスタに接続される入力データまたはイネーブル入力が入力ピンに保持される必要のある時間の長さ。	要求されるt_{SU}をプロジェクト全体、または任意の入力ピン、双方向ピンに指定することができる。
f_{MAX}	f_{MAX}（最高クロック周波数）は、内部のセットアップ・タイムおよびホールド・タイムの規定に違反することなく達成できるクロックの最高周波数。	要求されるf_{MAX}をプロジェクト全体、または任意の入力ピン、双方向ピン、レジスタに指定することができる。

☞ タイミングの要求をグローバルに指定するときは、Global Project Timing Requirements（Assignメニュー）を選択してください。個別のクリティカル・パスに対するタイミング要求を指定する場合は、Timing Requirements（Assignメニュー）を選択してください。

Synplifyのコンストレイントで性能を改善する方法

前述のMAX+PLUS IIソフトウェアに対するオプションを使用したデザインが要求される性能を達成できなかった場合は、３ページの図１に示されているデザイン・プロセスを繰り返し実行する必要があります。この場合は、下記のセクションに記述されているテクニックを適用して、Synplifyソフトウェアでデザインを改めて処理する必要があります。このセクションでは、以下の項目について解説します。

- Symbolic FSM（Finite State Machine）Compiler
- タイミング・コンストレイント
- リソースの共有
- HDL Analyst
- ピン・ロッキング
- EABのメモリへの使用
- EABのロジックへの使用

Symbolic FSM Compiler

SynplifyのSymbolic FSM Compilerユーティリティ・プログラムを使用して、ステート・マシンのデザインの性能を改善することができます。

このSymbolic FSM Compilerは、ステート・マシンを自動的に検出して再エンコードします。FLEXデバイスの場合は、ステート・マシンがワン・ホット形式に再エンコードされます（ワン・ホット・エンコーディングでは、１つのステートに１ビットが使用されるため、多くのステート・レジスタが必要になりますが、要求されるコントロール・ロジックは減少します）。プロジェクトのウィンドウで「*Symbolic FSM Compiler*」のオプションをオンに設定してください。

図28は、バイナリ・エンコーディング形式のステート・マシンとして実現されたVerilog HDLによるステート・マシンの記述例を示したものです。

図28　Verilog HDLによるバイナリ・エンコーディング形式のステート・マシン（1/3）

```
/* prep3 contains a small state machine */

module stmch1(clk, rst, in, out);
input clk, rst;
input [7:0] in;
output [7:0] out;

reg [7:0] out;
reg [3:0] current_state; // holds the current state

parameter // these parameters represent state names
     start = 3'b000,
     sa = 3'b001,
     sb = 3'b010,
     sc = 3'b011,
```

図28 Verilog HDLによるバイナリ・エンコーディング形式のステート・マシン (2/3)

```
     sd = 3'b100,
     se = 3'b101,
     sf = 3'b110,
     sg = 3'b111;

always @ (posedge clk or posedge rst)

begin
    if (rst) begin
         current_state = start;
         out = 8'b0;
    end else begin
         case (current_state)
         start: if (in == 8'h3c) begin
                     current_state = sa;
                     out = 8'h82;
              end else begin
                     out = 8'h00;
                     current_state = start;
              end
         sa: if (in == 8'h2a) begin
                     current_state = sc;
                     out = 8'h40;
              end else if (in == 8'h1f) begin
                     current_state = sb;
                     out = 8'h20;
              end else begin
                     current_state = sa;
                     out = 8'h04;
              end
         sb: if (in == 8'haa) begin
                     current_state = se;
                     out = 8'h11;
              end else begin
                     current_state = sf;
                     out = 8'h30;
              end
         sc: begin
              current_state = sd;
              out = 8'h08;
              end
         sd: begin
              current_state = sg;
              out = 8'h80;
              end
         se: begin
              current_state = start;
```

図28 Verilog HDL**によるバイナリ・エンコーディング形式のステート・マシン** (3/3)

```
                out = 8'h40;
                end
            sf: begin
                current_state = sg;
                out = 8'h02;
                end
            sg: begin
                current_state = start;
                out = 8'h01;
                end
            default: begin
            /* set current_state to 'bx (don't care) to tell the synplify software
            that all used states have been already been specified */
                current_state = 'bx;
                out = 'bx;
                end
            endcase
        end
end
endmodule
```

このデザインをSynplifyソフトウェアで論理合成し、MAX+PLUS IIソフトウェアでコンパイルしたとき、このデザインのf_{MAX}は75.18MHzとなり、27個のLEが使用されます。

Synplifyソフトウェアで「*Symbolic FSM Compiler*」のオプションをオンに設定してデザインを再合成すると、バイナリ・エンコーディングがSynplifyソフトウェアによってワン・ホット・エンコーディングに再エンコーディングされます。これをMAX+PLUS IIで再コンパイルすると、85.47MHzのf_{MAX}が実現され、27個のLEが使用される結果となります。

「*Symbolic FSM Compiler*」のオプションをオフに設定した場合でも、state_machineの属性を指定することによって、ステート・レジスタごとの最適化を行うことができます。

Synplifyソフトウェアでは、state_machineの属性を下記のような形式で使用することができます。

VHDL:

```
SIGNAL curstate : STATE_TYPE;
ATTRIBUTE state_machine : BOOLEAN;
ATTRIBUTE state_machine OF curstate : SIGNAL IS TRUE;
```

Verilog HDL:

```
reg [3:0] curstate /* synthesis state_machine */ ;
```

タイミング・コンストレイント

Synplifyソフトウェアは、論理合成の結果に影響を与える多くのタイミング・コンストレイントを提供しており、これらによってデザイン性能を改善することができます。

49ページの「HDL Analyst」に記述されているように、タイミング・コンストレイントを「HDL Analyst」（Technology view）からの結果に応じて使用します。

タイミング・コンストレイントは、Synplicity Design Constraintファイル（.sdc）に含まれています。プロジェクトにコンストレイント・ファイルを含めるときは、Synplify Project WindowでAddボタンをクリックして該当するファイルを選択し、「*Source Files*」のリストに追加します。

また、tclスクリプトから合成を行っている場合は、add_file -constraintコマンドを使用してコンストレイント・ファイルを読み込むこともできます。

クロック周波数の設定

プロジェクトのウィンドウで、デザインに対するデフォルトのクロック周波数を入力することができます。ただし、デザインで複数のクロックが使用されている場合は、下記に示す属性の指定を行って各クロックごとに周波数を規定する必要があります。

```
define_clock <クロック名> {-freq <MHz> | -period <ns>}
```

例：`define_clock sysclk -freq 33 MHz`

レジスタの入力に接続されるパスの遅延の改善方法

define_reg_input_delayのタイミング・コンストレイントを使用し、下記のフォーマットのコマンドでナノ・セカンド単位の数値を与えることによって、レジスタの入力と接続されるパスを高速化することができます。

```
define_reg_input_delay <レジスタ名> [-improve <ns>]
     [-route <ns>]
```

例：`define_reg_input_delay data_reg -improve 5 ns`

ここで、*<レジスタ名>*はバスではなく、1ビットのレジスタとなっている必要があります。複数のレジスタ・ビットに対して指定を行うときは、define_reg_input_delayのコマンドを複数回使用してください。

HDL AnalystまたはLog File Timing Reportでフリップフロップがネガティブなセットアップ・タイムやホールド・タイムとなるようなパスを持っているとレポートされた場合は、このレジスタに対するパスが原因となって目標のクロック周波数が達成されません。このような場合は、このレジスタに対して、define_reg_input_delayコマンドを-improveのオプション付きで使用してください。-improveのオプションを使用することで、Synplifyソフトウェアは最適化の実行時に目標のクロック周波数が達成されるようにデザインを再構成します。ここで、-improveには負の値を入力することが可能であり、これは時間に余裕があり、特定のフリップフロップに対するコンストレイントを緩和したいときに便利です。この方法でコンストレイントを緩和することによって、Synplifyはデザイン内の他のパスのタイミングを改善することができます。

レジスタ出力のパスの遅延を改善する方法

下記のコンストレイントはdefine_reg_input_delayと類似していますが、クリティカル・パスのソースがレジスタになっているときに、指定したレジスタ出力からのパスに対して適用されます。

```
define_reg_output_delay <レジスタ名> [-improve <ns>]
    [-route <ns>]
```

入力ピンが関連したパスの遅延を改善する方法

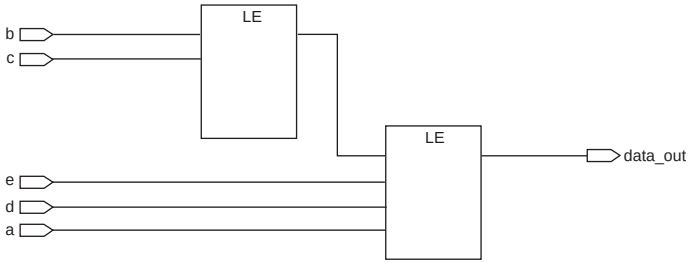
この遅延の改善方法を、５入力のANDゲートを記述した図29のVHDLコードで示します。

図29　**５入力の**AND**ゲート**

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;
USE ieee.std_logic_arith.ALL;

ENTITY split IS
    PORT( a,b,c,d,e :IN STD_LOGIC;
          data_out  :OUT STD_LOGIC);
END split;

ARCHITECTURE behave OF split IS
BEGIN
         data_out <= (a AND b AND c AND d AND e);
END behave;
```

この回路は、図30に示すようにFLEXデバイスの２個のLEで実現されます。

図30　２個のLEを使用して実現された５入力ANDゲート



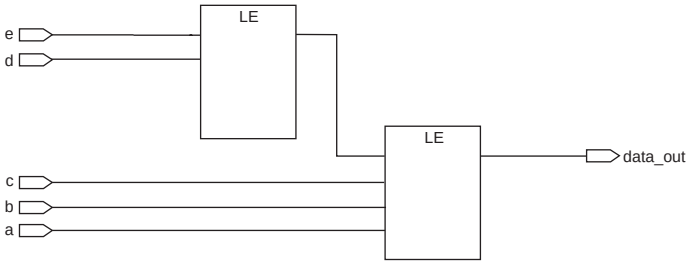
ここで、入力bのタイミングがクリティカルになっている場合は、下記のようにdefine_input_delayのコンストレイントを使用して、入力bが１個のLEのみを通るようにすることができます。

```
define_input_delay {<入力ポート名> | -default} <ns>
    [-improve <ns>] [-route <ns>]
```

例えば、下記のコマンドを使用すると、図31に示すような結果になります。

```
define_input_delay b 5
```

図31　クリティカル信号のパスをコントロールした回路



リソースの共有

Synplicity社のソフトウェアは、ロジック・ファンクション内のリソースの共有をコントロールするsyn_sharingの属性を提供しています。デフォルトで、Synplicity社のソフトウェアは、リソースを共有する設定を使用します。図32はデフォルトのリソース共有を使用したVerilog HDLのデザインを示したものです。

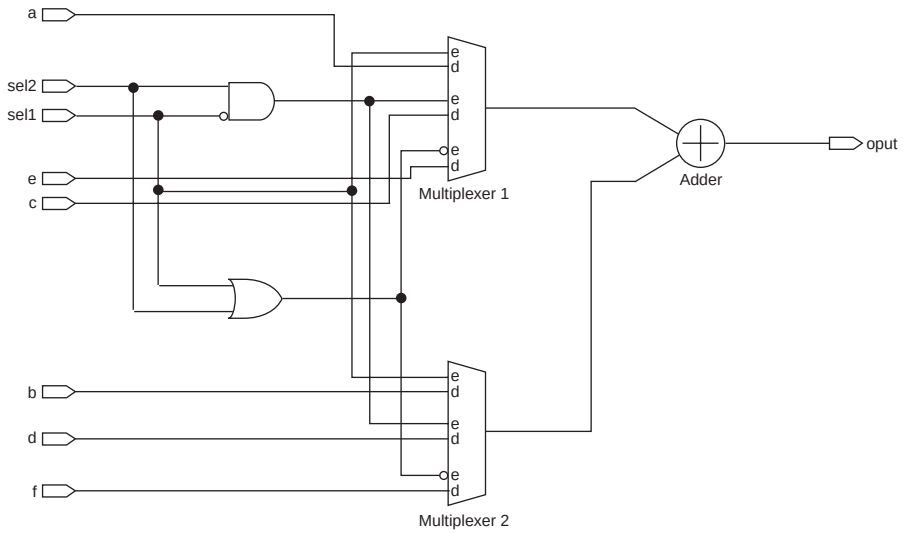
図32　デフォルトのリソース共有を使用したVerilog HDLのデザイン

リソースの共有に特別な属性の指定は必要ない。

```
module share (a,b,c,d,e,f,sel1,sel2,oput);
    input a,b,c,d,e,f;
    input sel1, sel2;
    output oput;
    reg oput;

    always @(a or b or c or d or e or f or sel1 or sel2)
    begin
         if (sel1)
              oput = a + b;
         else if (sel2)
              oput = c + d;
         else
              oput = e + f;
    end
endmodule
```

図33は、図32のVerilog HDLデザインを回路図で示したものです。

図33　リソースの共有



Synplifyソフトウェアはデフォルトでリソースを共有する設定を使用するため、このデザインは１個のアダーとアダーに対する入力を切り換えるためのマルチプレクサで実現されます。

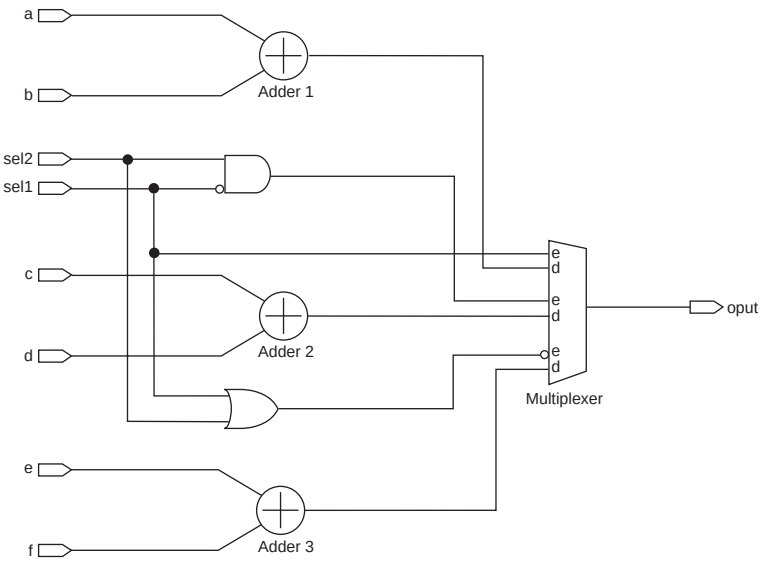
syn_sharingの属性は、図34に示すようにリソースの共有をディセーブルする場合にも使用できます。

図34　リソースの共有をディセーブルするときのsyn_sharingの使用方法

```
module no_share (a,b,c,d,e,f,sel1,sel2,oput); /*synthesis syn_sharing="off"*/
    input a,b,c,d,e,f;
    input sel1, sel2;
    output oput;
    reg oput;

    always @(a or b or c or d or e or f or sel1 or sel2)
    begin
      if (sel1)
            oput = a + b;
      else if (sel2)
            oput = c + d;
      else
            oput = e + f;
    end
endmodule
```

図35は、図34で示したVerilog HDLのデザインを回路図で示したものです。

図35　リソースの共有をディセーブルした回路



この実現方法では、アダーの３本の出力がマルチプレクサで切り換えられるようになっています。リソースの共有から得られる結果を検討し、各デザインに対してリソースの共有の設定が適切かどうかをケース・バイ・ケースで判断する必要があります。

HDL Analyst

HDL AnalystはSynplifyソフトウェアの合成環境で生産性の高いグラフィック・ツールとなっています。このツールは、合成結果をビジュアルで確認して、デバイスの性能とエリアを改善するときに役立ちます。

HDL Analystは、VHDLおよびVerilog HDLのデザインから階層化されたRTL（Register Transfer Language）レベルの記述とテクノロジ・プリミティブ・レベルの回路図（SynplifyソフトウェアがLUTにマッピングした状態）を自動的に生成します。この機能により、RTLレベルおよびテクノロジ・プリミティブ・レベルの回路図と作成したソース・コードとを比較するクロス・プロービングの機能が提供されます。例えば、ソース・コード内のある行をハイライトしておくと、回路図上の対応するロジックもハイライトされます。逆に、回路図のロジックをダブル・クリックすることによって、対応するソース・コードを確認することができます。

このHDL Analystのクロス・プロービング機能を使用することによって、デザインの解析が可能になるため、デザインの変更個所をビジュアルで確認したり、デバイス・エリアの縮小やデバイス性能の向上を実現するためのタイミング・コンストレイントを追加するときに便利な機能になります。

HDL Analystはデザイン内のクリティカル・パスをハイライトしたり、分離することもできるようになっており、短時間で問題のある領域を解析し、タイミング・コンストレイントを追加して、再度合成を実行することが可能です。

HDL Analystによるクリティカル・パスの確認は、以下の手順で行います。

1. デザインを合成します。

2. Technology View（HDL Analystメニュー）を選択します。

3. Show Critical Path（HDL Analystメニュー）を選択して、Technology Viewでクリティカル・パスがハイライトされるようにします。これは、Synplifyのツールバー上にあるShow Critical Pathのボタン（ストップウオッチのボタン）をクリックするか、ポップ・アップ・メニューからマウスのライト・ボタンをクリックすることでも行えます。クリティカル・パスのハイライトをオンまたはオフにするときも、このShow Critical Pathのコマンドを使用します。

   Show Critical Pathのコマンド（HDL Analystメニュー）はデザイン内のクリティカル・パスをハイライトし、各インスタンスの上にタイミング値を表示します。このウィンドウからクリティカル・パスを解析したり、これらを別の回路図に分離して解析が容易になるようにすることができます。

4. Filter Schematic コマンド（HDL Analystメニュー）を選択して、クリティカル・パスを分離します。これは、Synplifyのツールバーにある Filter Schematicボタン（バッファ・ボタン）をクリックするか、ポップ・アップ・メニューからマウスのライト・ボタンをクリックすることでも行えます。

   Filter Schematic コマンド（HDL Analystメニュー）は選択されたすべてのオブジェクト（この場合は、ハイライトされたクリティカル・パスのインスタンス）に対して適用され、これらが１つの回路図にグループ化され、他のすべてのオブジェクトが画面から除去されます。クリティカルなインスタンスが多くの回路図にある場合は、それらがどのオリジナル回路図にあったかとは関係なく、すべてのクリティカルなインスタンスが収集され、１つの回路図で表示されます。

5. 再度、Filter Schematic コマンド（HDL Analystメニュー）を選択して、オリジナルの回路図をリストアします。

6. Show Critical Pathコマンドをオンに設定してクリティカル・パスを解析します。Synplifyは各インスタンスの上に２種類のタイミング値（遅延とスラック時間）を表示します。この情報をベースにして性能を改善する方法については、40ページの「Synplifyのコンストレイントで性能を改善する方法」をご覧ください。

HDL Analystに関する詳細については、Synplifyのオンライン・ヘルプを参照してください。

ピン・ロッキング

MAX+PLUS IIソフトウェアでは、Assignメニューのコマンドを選択して現在のプロジェクトに対するアサインメントを入力したり、フロアプラン・エディタで特定のノードやピンを移動させたり、アサインメント・アンド・コンフィギュレーション・ファイル（.acf）をマニュアルで修正することができるようになっています。これらのアサインメント方法は、すべてACFをエディットすることになります。ただし、ACFの変更は１つのアプリケーションで１回のみ可能です。テキスト・エディタのウィンドウでACFの内容をマニュアルで修正した場合は、Assignメニューのコマンドを選択するときや、フロアプラン・エディタに切り換えるときには、必ず修正したACFをセーブする必要があります。マニュアルによるACFの変更ではエラーが発生する可能性が高いため、アルテラはAssignメニューのコマンドやフロアプラン・エディタを使用したアサインメントの入力を行うことを推奨します。

また、アルテラは、まず最初にアサインメントなしでプロジェクトをコンパイルすることを推奨します。最初のコンパイル前に特定のアサインメントを行いたい場合は、下記の手順で行ってください。

1. Pin/Location/Chip（Assignメニュー）を選択します。 Pin/Location/Chipのダイアログ・ボックスで、アサインメントを入力したいノードまたはピンの名前を入力します。

2. *Chip Resources*からピン、ロジック・セル、または他のリソースを選択します。

3. Addをクリックして、次にOKのボタンをクリックします。

4. MAX+PLUS IIのCompilerのウィンドウでStartボタンをクリックして、コンパイルを開始させます。

また、Synplifyソフトウェアでは、ロジックや信号がアルテラ・デバイスの特定の位置に配置されるような指定が行えます。これらのアサインメントはSynplifyが生成するEDIFのネットリストを介してMAX+PLUS IIのソフトウェアに受け渡されます。

図36はVerilog HDLで信号のピン・アサインメントを行った例を示したものです。

図36　Verilog HDLによるピンのアサインメント（1/2）

```
module adder(cout, sum, a, b, cin);

    parameter size = 1; /* declare a parameter. default required */
    output cout;
    output [size-1:0] sum; // sum uses the size parameter
    input cin;
```

図36　Verilog HDLによるピン・アサインメント（2/2）

```
    input [size-1:0] a, b; // 'a' and 'b' use the size parameter

    assign {cout, sum} = a + b + cin;

endmodule

module adder8(cout, sum, a, b, cin);

    output cout /* synthesis altera_chip_pin_lc="adder8@159" */;

    output [7:0] sum /* synthesis
       altera_chip_pin_lc="@17,@166,@191,@152,@15,@148,@147,@149" */;

    input [7:0] a /* synthesis
       altera_chip_pin_lc="adder8@194,adder8@177,adder8@70,adder8@96,
       adder8@109,adder8@6,adder8@174,adder8@204" */;

    input [7:0] b;
    input cin;

    adder my_adder (cout, sum, a, b, cin);
    defparam my_adder.size = 8;

endmodule
```

図37はVHDLで信号のピン・アサインメントを行った例を示したものです。

図37　VHDLによるピン・アサインメント（1/2）

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY adder8 IS
    GENERIC (num_bits  : INTEGER := 8) ;
    PORT ( a,b         : IN STD_LOGIC_VECTOR (NUM_BITS -1 DOWNTO 0);
           result      : OUT STD_LOGIC_VECTOR(NUM_BITS -1 DOWNTO 0));

ATTRIBUTE altera_chip_pin_lc : STRING;

ATTRIBUTE altera_chip_pin_lc OF RESULT : SIGNAL IS
"@17,@166,@191,@152,@15,@148,@147,@149";

ATTRIBUTE altera_chip_pin_lc OF a : SIGNAL IS
"adder8@194,adder8@177,adder8@70,adder8@96,adder8@109,adder8@6,adder8@174,
    adder8@204";
```

図37　VHDLによるピン・アサインメント（2/2）

```
END adder8;

ARCHITECTURE behave OF adder8 IS

BEGIN
    result <= a + b;
END;
```

EABのメモリへの使用

EABにROMまたはRAMを実現する場合は、表２に示すLPMファンクションを使用してください。

表２　EABに実現可能なLPMファンクション

LPMファンクション名	機　　能
lpm_ram_dq	同期／非同期RAM（リード／ライトのデータ・ポートを分離）
lpm_ram_io	同期／非同期RAM（リードとライトのデータ・ポートが双方向）
lpm_rom	同期／非同期ROM
csdpram	デュアル・ポートRAM
csfifo	FIFOバッファ
altdpram	デュアル・ポートRAM
scfifo	シングル・クロックFIFO
dcfifo	デュアル・クロックFIFO

EABはメモリ・ファンクションの構成に最適化されているため、LEを他のロジックに使用することができます。

EABの使用方法の詳細については、「*FLEX 10K Embedded Programmable Logic Family*」のデータシートをご覧ください。

33ページの図26はメモリ・ファンクションをインスタンス化した例を示したものです。

EABのロジックへの使用

FLEX 10KのEABは、組み合わせロジックを実現するときにも使用できます。EABには、大規模なLUTを効率的に実現することができます。

FLEX 10Kデバイスでは、１個のEABが20個のLEのダイ面積とコストに相当しています。このため、１個のEABで20個以上のLEを必要とする複雑なロジックを１段のロジック・レベルで実現することもできます。

表３は、１個のEABに実現できる入力（*X*）と出力（*Y*）の本数を示したものです。

表３　１個のEABに実現可能な入力（X）と出力（Y）の本数

入力数（X）	出力数（Y）
8	8
9	4
10	2
11	1

また、さらに大規模なファンクション（EPF10K20-3デバイスに実現される8×8のマルチプライヤなど）の構成には、EABをカスケード接続したり、LEと結合させて使用することができます。ロジックにEABを使用することによって、タイミング遅延を増加させることなく、27個のLEの使用を節減することができます（1EAB=20LE）。表４を参照してください。

表４　EABとLEの比較

実現方法	使用されるリソース	最長遅延
LEのみ	136個のLE	40.0ns
複数のEAB	4個のEABと29個のLE	39.0ns

EABにステート・マシンをインスタンス化した例が、図16から図19に示されていますので参照してください。

まとめ

プログラマブル・ロジックの業界では、設計時間の短縮と性能の向上が重要な要素となっています。このアプリケーション・ノートには、デザインの効率化により、目標とする性能の達成や設計時間の短縮に役立つ多様なテクニックが示されています。VHDLやVerilog HDLのコーディング・テクニック、Synplifyソフトウェアのコンストレイント、MAX+PLUS IIソフトウェアのオプションを使用することで、FLEX 10Kデバイスに実現されるデザイン性能を改善し、最終的にはデザイン全体を改善することができます。

Altera、MAX、MAX+PLUS、MAX+PLUS II、FLEX、FLEX 10KはAltera Corporationの米国および該当各国におけるtrademark またはservice markです。この資料に記載されているその他の製品名およびサービス名は該当各社のtrademarkです。なお、SynplifyはSynplicity社のregistered trademarkです。Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

Copyright © 1998 Altera Corporation. All rights reserved.

I.S. EN ISO 9001

日本アルテラ株式会社

〒163-0436
東京都新宿区西新宿2-1-1
新宿三井ビル私書箱261号
TEL. 03-3340-9480　FAX. 03-3340-9487
http://www.altera.com/japan/

本社 Altera Corporation

101 Innovation Drive,
San Jose, CA 95134
TEL : (408) 544-7000
http://www.altera.com

この資料に記載された内容は予告なく変更されることがあります。最新の情報は、アルテラのウェッブ・サイト（http://www.altera.com）でご確認ください。この資料はアルテラが発行した英文のアプリケーション・ノートを日本語化したものであり、アルテラが保証する規格、仕様は英文オリジナルのものです。