6. 推奨される HDL コーディング構文

この資料は英語版を翻訳したもので、内容に相違が生じる場合には原文を優先します。こちらの日本語版は参考用としてご利用ください。設計の際には、最新の英語版で内容をご確認ください。

QII51007-6.0.0

はじめに

HDL コーディング・スタイルは、プログラマブル・ロジック・デザインで得られる結果の品質に大きく影響する場合があります。合成ツールは、ロジック利用率とパフォーマンスの両面でHDLコードを最適化します。ただし、場合によっては、最適化を最大限に実行するために人間によるデザインの理解が要求されたり、合成ツールにデザインの目的や意図についての情報がまったく与えられないこともあります。最適化において結果の品質を改善できる機会は、数多くあります。

この章では、アルテラのデバイスをターゲットとしたときに、最適な合成結果を得るための HDL コーディング・スタイルの推奨事項について説明します。以下の項から構成されます。

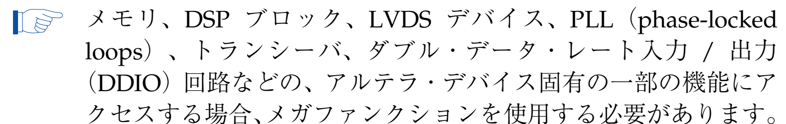
- アルテラのメガファンクション
- HDL コードでのアルテラ・メガファンクションのインスタンス化
- HDL コードからのアルテラ・メガファンクションの推測
- デバイス固有のコーディング・ガイドライン
- その他のロジック構造のコーディング・ガイドライン

デザインの構造化のガイドラインについて詳しくは、「Quartus II ハンドブック Volume 1」の「Design Recommendations for Altera Devices」の章を参照してください。

スタイルの推奨事項、オプション、または合成ツール（Quartus® II 合成機能とその他のサードパーティ EDA ツールを含む）に固有の HDL 属性については、ツール・ベンダのマニュアル、または「Quartus II ハンドブック Volume 1」の「合成」セクションの該当する章を参照してください。

アルテラのメガファンクションの使用

アルテラでは、アルテラ・デバイスのアーキテクチャ用に最適化された、パラメータ化されたメガファンクションを提供しています。独自のロジック・コーディングの代わりにメガファンクションを使用することで、貴重なデザイン時間が節約できます。さらに、アルテラが提供するメガファンクションを使用すると、より効率的にロジック合成とデバイス実装を行うことができます。メガファンクションのサイズをスケーリングしたり、またパラメータを設定することによって各種オプションを設定できます。メガファンクションには、パラメータ化されたモジュールのライブラリ（LPM）とアルテラ・デバイス固有のメガファンクションがあります。



メモリ、DSP ブロック、LVDS デバイス、PLL（phase-locked loops）、トランシーバ、ダブル・データ・レート入力 / 出力（DDIO）回路などの、アルテラ・デバイス固有の一部の機能にアクセスする場合、メガファンクションを使用する必要があります。

HDL コードでメガファンクションを使用するには、6–3 ページの「HDL コードでのアルテラ・メガファンクションの インスタンス化」で説明するようにインスタンス化します。デバイス・ファミリまたはベンダに依存しない独自のコードを作成する方が望ましい場合や、メガファンクションを直接インスタンス化したくない場合があります。メガファンクションのインスタンス化を希望しない場合は、6–6 ページの「HDL コード からの アルテラ・メガファンクションの推測」のガイドラインおよびコーディング例に従い、汎用 HDL コードで適切なアルテラ・メガファンクションが推測されるようにします。

一部のデザインでは、メガファンクションをインスタンス化するより、汎用 HDL コードを使用した方が良い結果が得られることがあります。標準 HDL コードを使用する場合とメガファンクションを使用する場合を説明した、以下の一般的なガイドラインと例を参照してください。

- 簡単な加算または減算ファンクションについては、LPM ファンクションではなく + または – シンボルを使用してください。簡単な算術演算に対して LPM ファンクションをインスタンス化した場合は、ファンクションがハードコードされ、合成アルゴリズムが基本的なロジック最適化を活用できなくなるため、効率の高い結果が得られないことがあります。同期ローダブル・カウンタなどの複雑な算術ロジックの場合は、LPM ファンクションを使用すると、HDL コードからは推測が困難なアーキテクチャ固有の詳細な機能にアクセスできます。
- 簡単なマルチプレクサおよびデコーダの場合、LPM ファンクションの代わりにアレイ表記法（`out = data[sel]` など）を使用してください。アレイ表記法は非常に有効に機能し構文も簡単です。APEX™ シリーズ・デバイスのカスケード・チェインなどのアーキテクチャ機能を活用するには `LPM_MUX` ファンクションを使用できますが、こ

の LPM ファンクションは、特定の実装を強制する場合にのみ使用してください。

- 可能であれば、除算演算は避けてください。除算は本質的に低速な演算です。設計者の多くは乗算を工夫して、除算結果を得ています。除算が必要な場合は、`LPM_DIVIDE` ファンクションを使用すると可能な最良の結果が得られます。

HDL コードでのアルテラ・メガファンクションの インスタンス化

以下の項では、HDL コードのメガファンクションを以下の方法でインスタンス化して使用する方法を説明します。

- MegaWizard® Plug-In Manager を使用したメガファンクションのインスタンス化 —MegaWizard Plug-In Manager を使用すると、ファンクションをパラメータ化し、ラッパ・ファイルを作成できます。
- サードパーティ合成ツール用クリア・ボックス・ネットリスト・ファイルの作成 — オプションで、ラッパ・ファイルの代わりにクリア・ボックス・ボディを作成することができます。
- ポート & パラメータ定義を使用したメガファンクションのインスタンス化—HDLコードで直接ファンクションをインスタンス化できます。

MegaWizard Plug-In Managerを使用したメガファンクションのインスタンス化

HDL コードでインスタンス化できるメガファンクションを Quartus II GUI で作成するには、この項の説明に従って MegaWizard Plug-In Manager を使用します。MegaWizard Plug-In Manager は、メガファンクションをカスタマイズおよびパラメータ化するためのグラフィカル・ユーザ・インタフェースを提供し、すべてのメガファンクション・パラメータを正しく設定できるようにします。パラメータの設定が終了したら、生成するファイルを指定できます。選択した言語に応じて、MegaWizard Plug-In Manager は正しいパラメータでメガファンクションをインスタンス化し、以下のファイル・セットのいずれかを生成します。

- AHDL テキスト・デザイン・ファイル（**.tdf**）ラッパ・ファイルおよびサンプルのインスタンス・テンプレートText Design File（**_inst.tdf**）。
- Verilog HDL（**.v**）ラッパ・ファイル、サンプルのインスタンス・テンプレート Verilog HDL ファイル（**_inst.v**）、およびブラック・ボックス Verilog HDL モジュール宣言。
- VHDL（**.vhd**）ラッパ・ファイルおよびサンプルのインスタンス・テンプレート VHDL ファイル（**_inst.vhd**）。

デザイン内のメガファンクション・ラッパ・ファイルは、対応するサンプルのインスタンス・ファイルを使用してインスタンス化できます。また、MegaWizard Plug-In Manager はデフォルトで、以下のデフォルト・ファイルも作成します。

- VHDL デザイン・ファイルで使用できるコンポーネント宣言ファイル（**.cmp**）
- テキスト・デザイン・ファイル（**.tdf**）で、または Verilog HDL デザイン・ファイルの参照用として使用できる ADHL インクルード・ファイル（**.inc**）

MegaWizard Plug-In Manager で生成されるファイルのリストと説明については、表 6–1 を参照してください。

表 6–1. MegaWizard Plug-In Manager 生成のファイル （１/２）

ファイル	説明
<*出力ファイル*>**.bsf**	ブロック・シンボル・ファイル —Quartus II ブロック・デザイン・ファイル（**.bdf**）で使用されます。
<*出力ファイル*>**.cmp**	コンポーネント宣言ファイル —VHDL デザインで使用されます。
<*出力ファイル*>**.inc**	ADHL インクルード・ファイル —AHDL デザインで使用されます。
<*出力ファイル*>**.tdf**(1)	AHDL ラッパ･ファイル—AHDL デザインでインスタンス化するためのメガファンクション・ラッパ・ファイル
<*出力ファイル*>**.vhd**(2)(4)	VHDL ラッパ・ファイル —VHDL デザインのインスタンス化のためのメガファンクション・ラッパ・ファイル、またはクリア・ボックス・ネットリスト・ファイル。
<*出力ファイル*>**.v**(3)(4)	Verilog HDL ラッパ・ファイル —Verilog HDL デザインのインスタンス化のためのメガファンクション・ラッパ・ファイル、またはクリア・ボックス・ネットリスト・ファイル。
<*出力ファイル*>**_bb.v**(3)	ブラック・ボックス Verilog HDL モジュール宣言 — サードパーティ合成ツールでブラック・ボックスを作成する場合、ポートの方向を指定するために Verilog HDL デザインで使用される中空モジュール宣言。
<*出力ファイル*>**_inst.tdf**(1)	テキスト・デザイン・ファイル・インスタンス・テンプレート — メガファンクション・ラッパ・ファイルにあるサブデザインのサンプル AHDL インスタンス。
<*出力ファイル*>**_inst.vhd**(2)	VHDL インスタンス・テンプレート — メガファンクション・ラッパ・ファイルにあるエンティティのサンプル VHDL インスタンス。

表 6–1. MegaWizard Plug-In Manager 生成のファイル （２/２）

ファイル	説明
<*出力ファイル*>**_inst.v**(3)	Verilog HDL インスタンス・テンプレート — メガファンクション・ラッパ・ファイルにあるモジュールのサンプル Verilog HDL インスタンス。

表 6–1 の注 :

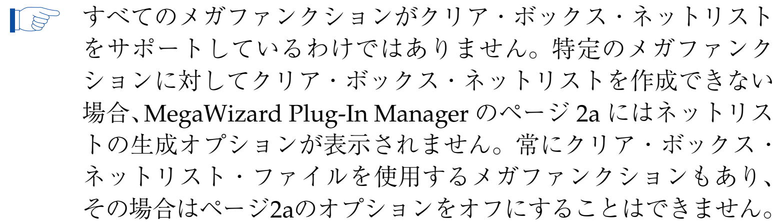
(1) MegaWizard Plug-In Manager は、AHDL 出力ファイルを選択した場合にのみこのファイルを生成します。
(2) MegaWizard Plug-In Manager は、VHDL 出力ファイルを選択した場合にのみこのファイルを生成します。
(3) MegaWizard Plug-In Manager は、Verilog HDL 出力ファイルを選択した場合にのみこのファイルを生成します。
(4) メガファンクション・ラッパ・ファイルは、ほとんどのメガファンクションに対してデフォルトで作成されます。クリア・ボックスの機能を活用するには、Tools メニューで **MegaWizard Plug-In Manager** をクリックし、**Generate clear box netlist file instead of a default wrapper file (for use with supported EDA synthesis tools only)** をオンにします。MegaWizard Plug-In Manager の使用方法について詳しくは、Quartus II ヘルプを参照してください。

サードパーティ合成ツール用クリア・ボックス・ネットリスト・ファイルの作成

サードパーティ合成ツールで特定のメガファンクションを使用する場合、オプションでラッパ・ファイルの代わりにクリア・ボックス・ボディを作成できます。クリア・ボックス・ボディは完全に合成されたメガファンクションで、特定のサードパーティ EDA 合成ツールで使用できます。メガファンクション・クリア・ボックス・ボディを含むネットリスト・ファイルは、サードパーティ合成ツール向けに、Quartus II ソフトウェアで使用されるアーキテクチャ上の詳細情報を提供します。特定の合成ツールはこの情報を利用することで、タイミングおよびリソース利用率の予測をより正しく報告できます。また、合成ツールはタイミング情報を使用して、タイミング・ドリブン最適化に集中できます、また結果の品質を向上させることができます。

合成ツールでのクリア・ボックスのサポートについて詳しくは、ツール・ベンダのマニュアル、または「Quartus II ハンドブック Volume 1」の「合成」セクションの該当する章を参照してください。

クリア・ボックス・ネットリストを生成するには、MegaWizard Plug-In Manager のメガファンクション選択ページ 2a で、**Generate clear box netlist file instead of a default wrapper file (for use with supported EDA synthesis tools only)** をオンにします。



すべてのメガファンクションがクリア・ボックス・ネットリストをサポートしているわけではありません。特定のメガファンクションに対してクリア・ボックス・ネットリストを作成できない場合、MegaWizard Plug-In Manager のページ 2a にはネットリストの生成オプションが表示されません。常にクリア・ボックス・ネットリスト・ファイルを使用するメガファンクションもあり、その場合はページ2aのオプションをオフにすることはできません。

ポートおよびパラメータ定義を使用したメガファンクションのインスタンス化

メガファンクションは、他のサブデザイン、モジュール、またはコンポーネントのように、メガファンクションを呼び出してパラメータを設定することによって、AHDL、Verilog HDL、または VHDL コードで直接インスタンス化できます。

メガファンクションのポートとパラメータのリストについては、Quartus II ヘルプの該当するメガファンクションを参照してください。Quartus II ヘルプには、VHDL コンポーネント宣言のサンプルと、各メガファンクションの AHDL ファンクションのプロトタイプも記載されています。

アルテラでは、PLL、トランシーバ、LVDS ドライバなどの複雑なメガファンクションには、MegaWizard Plug-In Manager の使用を推奨しています。MegaWizard Plug-In Manager の使用方法について詳しくは、6–3 ページの「MegaWizard Plug-In Manager を使用したメガファンクションのインスタンス化」を参照してください。

HDL コードからのアルテラ・メガファンクションの推測

Quartus II 合成機能を含む合成ツールは、特定のタイプの HDL コードを認識し、適切なメガファンクションを自動的に推測します。合成ツールは、デザインをコンパイルするときには、特にメガファンクションをインスタンス化しない場合でも、アルテラ・メガファンクション・コードを使用します。合成ツールは、アルテラ・デバイスに最適化されたロジックを利用するメガファンクションを推測します。このようなロジックの面積と性能は、同じ HDL コードの汎用ロジックを推測して得られる結果よりも良好な場合があります。

以下の項では、標準合成ツールが認識し、メガファンクションにマップするロジックのタイプについて説明します。合成ソフトウェアは、ここに挙げる特定のファンクションのみ推測します。合成ソフトウェアは、PLL、LVDS ドライバ、トランシーバ、DDIO 回路など、その他のファンクションを HDL コードから推測することはできません。これらのファンクションは HDL コードで完全に、あるいは効率的に説明できないためです。合成ツール・オプションを使用して、特定のメガファンクショ

ンの推測をオフにできるケースもあります。以下の項では、以下のメガファンクションを汎用 HDL コードから推測する方法について説明しています。

- lpm_mult — 乗算器を HDL コードから推測
- altmult_accum & altmult_add — 乗算累積器および乗算加算器を HDL コードから推測
- altsyncram & lpm_ram_dp — RAM ファンクションを HDL コードから推測
- lpm_rom — ROM を HDL コードから推測
- altshift_taps — シフト・レジスタを HDL コードから推測

合成ツールの機能とオプションについては、合成ツールのマニュアル、または「Quartus II ハンドブック Volume 1」の「合成」の項の該当する章を参照してください。

lpm_mult — 乗算器を HDL コードから推測

乗算器ファクションを推測する場合、合成ツールは乗算器を検索し、lpm_mult または altmult_add メガファンクションに変換します。あるいは乗算器デバイスの素子に直接マップすることもあります。DSP ブロックを含むデバイスの場合、ソフトウェアはデバイスの利用状況に応じて、ロジックの代わりに乗算を DSP ブロックに実装できます。Quartus II フィッタは、入力および出力レジスタを DSP ブロックに配置（すなわち、レジスタ・パッキングを実行）し、性能および面積利用率を改善することもできます。

DSP ブロックと DSP ブロックが実装できるファンクションについて詳しくは、該当するアルテラ・デバイス・ファミリのハンドブックと、アルテラ Web サイト **www.altera.com** の DSP ソリューション・センタを参照してください。

以下の 4 つのコード・サンプルに、合成ツールが lpm_mult または altmult_add メガファンクションとして推測できる符号なしおよび符号付き乗算器の Verilog HDL と VHDL の例を示します。それぞれの例は、1 つの DSP ブロックの 9 ビット・エレメントにフィットします。また、レジスタ・パッキングが実行されると、レジスタ用の特別なロジック・セルが不要になります。

Verilog HDL での符号付き宣言は、Verilog 2001 規格の 1 つの機能です。

例 6–1. Verilog HDL 符号なし乗算器

```
module unsigned_mult (out, a, b);
   output [15:0] out;
   input  [7:0] a;
   input  [7:0] b;
   assign out = a * b;
endmodule
```

例 6–2. 入力および出力レジスタを備えた Verilog HDL 符号付き乗算器（パイプライン化 = 2）

```
module signed_mult (out, clk, a, b);
   output [15:0] out;
   input clk;
   input signed [7:0] a;
   input signed [7:0] b;

   reg signed [7:0] a_reg;
   reg signed [7:0] b_reg;
   reg signed [15:0] out;
   wire signed [15:0] mult_out;

   assign mult_out = a_reg * b_reg;

   always @ (posedge clk)
   begin
      a_reg <= a;
      b_reg <= b;
      out <= mult_out;
   end
endmodule
```

例 6–3. 入力および出力レジスタを備えた VHDL 符号なし乗算器（パイプライン化 = 2）

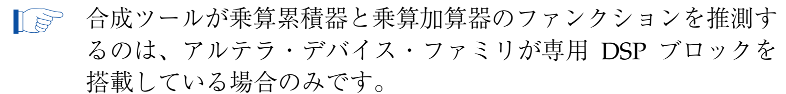
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.all;

ENTITY unsigned_mult IS
   PORT (
      a: IN UNSIGNED (7 DOWNTO 0);
      b: IN UNSIGNED (7 DOWNTO 0);
      clk: IN STD_LOGIC;
      aclr: IN STD_LOGIC;
      result: OUT UNSIGNED (15 DOWNTO 0)
   );
END unsigned_mult;

ARCHITECTURE rtl OF unsigned_mult IS
   SIGNAL a_reg, b_reg: UNSIGNED (7 DOWNTO 0);
BEGIN
   PROCESS (clk, aclr)
   BEGIN
      IF (aclr ='1') THEN
         a_reg <= (OTHERS => '0');
         b_reg <= (OTHERS => '0');
         result <= (OTHERS => '0');
      ELSIF (clk'event AND clk = '1') THEN
         a_reg <= a;
         b_reg <= b;
         result <= a_reg * b_reg;
      END IF;
   END PROCESS;
END rtl;
```

例 6–4. VHDL 符号付き乗算器

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

ENTITY signed_mult IS
   PORT (
      a: IN SIGNED (7 DOWNTO 0);
      b: IN SIGNED (7 DOWNTO 0);
      result: OUT SIGNED (15 DOWNTO 0)
   );
END signed_mult;

ARCHITECTURE rtl OF signed_mult IS
   SIGNAL a_int, b_int: SIGNED (7 downto 0);
   SIGNAL pdt_int: SIGNED (15 downto 0);
BEGIN
   a_int <= (a);
   b_int <= (b);
   pdt_int <= a_int * b_int;
   result <= pdt_int;
END rtl;
```

altmult_accum & altmult_add— 乗算累積器および乗算加算器を HDL コードから推測

合成ツールは、乗算累積器または乗算加算器を検出し、それぞれ `altmult_accum` または `altmult_add` メガファンクションに変換します。この後 Quartus II ソフトウェアは、配置配線中にこれらのファンクションを DSP ブロックに配置します。



合成ツールが乗算累積器と乗算加算器のファンクションを推測するのは、アルテラ・デバイス・ファミリが専用 DSP ブロックを搭載している場合のみです。

乗算累積器は、加算器と乗算器から構成されます。加算器の出力は後段のレジスタに供給され、そのレジスタ出力が加算器の入力となります。乗算加算器は、1 レベルまたは 2 レベルの加算、減算、または加算 / 減算演算子に供給する 2 ～ 4 個の乗算器から構成されます。加算が使用される場合には、常に第 2 レベルの演算子になります。Quartus II フィッタは、乗算累積器と乗算加算器に加えて、入力および出力レジスタを DSP ブロックに配置してレジスタをパックし、性能および面積利用率を改善します。

例 6–5 から 6–8 に示す Verilog HDL と VHDL コードのサンプルでは、乗算累積器と乗算加算器を推測しています。

例 6–5. 入力、出力、およびパイプライン・レジスタを備えた Verilog HDL 符号なし乗算累積器（レイテンシ = 3）

```
module unsig_altmult_accum (dataout, dataa, datab, clk, aclr, clken);
   input [7:0] dataa;
   input [7:0] datab;
   input clk;
   input aclr;
   input clken;
   output [31:0] dataout;
   reg [31:0] dataout;
   reg [7:0] dataa_reg;
   reg [7:0] datab_reg;
   reg [15:0] multa_reg;
   wire [15:0] multa;
   wire [31:0] adder_out;
   assign multa = dataa_reg * datab_reg;
   assign adder_out = multa_reg + dataout;
   always @ (posedge clk or posedge aclr)
   begin
      if (aclr)
      begin
         dataa_reg <= 0;
         datab_reg <= 0;
         multa_reg <= 0;
         dataout <= 0;
      end
      else if (clken)
      begin
         dataa_reg <= dataa;
         datab_reg <= datab;
         multa_reg <= multa;
         dataout <= adder_out;
      end
   end
endmodule
```

例 6–6. Verilog HDL 符号付き乗算加算器（レイテンシ = 0）

```
module sig_altmult_add (dataa, datab, datac, datad, result);
   input signed [15:0] dataa;
   input signed [15:0] datab;
   input signed [15:0] datac;
   input signed [15:0] datad;
   output [32:0] result;

   wire signed [31:0] mult0_result;
   wire signed [31:0] mult1_result;

   assign mult0_result = dataa * datab;
   assign mult1_result = datac * datad;
   assign result = (mult0_result + mult1_result);
endmodule
```

例 6–7. 入力、出力、およびパイプライン・レジスタを備えた VHDL 符号なし乗算加算器（レイテンシ = 3）

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

ENTITY unsignedmult_add IS
   PORT (
      a: IN UNSIGNED (7 DOWNTO 0);
      b: IN UNSIGNED (7 DOWNTO 0);
      c: IN UNSIGNED (7 DOWNTO 0);
      d: IN UNSIGNED (7 DOWNTO 0);
      clk: IN STD_LOGIC;
      aclr: IN STD_LOGIC;
      result: OUT UNSIGNED (15 DOWNTO 0)
   );
END unsignedmult_add;

ARCHITECTURE rtl OF unsignedmult_add IS
   SIGNAL a_int, b_int, c_int, d_int: UNSIGNED (7 DOWNTO 0);
   SIGNAL pdt_int, pdt2_int: UNSIGNED (15 DOWNTO 0);
   SIGNAL result_int: UNSIGNED (15 DOWNTO 0);
BEGIN
   PROCESS (clk, aclr)
   BEGIN
      IF (aclr = '1') THEN
         a_int <= (OTHERS => '0');
         b_int <= (OTHERS => '0');
         c_int <= (OTHERS => '0');
         d_int <= (OTHERS => '0');
         pdt_int <= (OTHERS => '0');
         pdt2_int <= (OTHERS => '0');
         result_int <= (OTHERS => '0');

      ELSIF (clk'event AND clk = '1') THEN
         a_int <= a;
         b_int <= b;
         c_int <= c;
         d_int <= d;
         pdt_int <= a_int * b_int;
         pdt2_int <= c_int * d_int;
         result_int <= pdt_int + pdt2_int;
      END IF;
      END PROCESS;
   result <= result_int;
END rtl;
```

例 6–8. 入力、出力、およびパイプライン・レジスタを備えた VHDL 符号付き乗算累積器（レイテンシ = 3）

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

ENTITY sig_altmult_accum IS
   PORT (
      a: IN SIGNED(7 DOWNTO 0);
      b: IN SIGNED (7 DOWNTO 0);
      clk: IN STD_LOGIC;
      accum_out: OUT SIGNED (15 DOWNTO 0)
   ) ;
END sig_altmult_accum;

ARCHITECTURE rtl OF sig_altmult_accum IS
   SIGNAL a_reg, b_reg: SIGNED (7 DOWNTO 0);
   SIGNAL pdt_reg: SIGNED (15 DOWNTO 0);
   SIGNAL adder_out: SIGNED (15 DOWNTO 0);
BEGIN
   PROCESS (clk)
   BEGIN
      IF (clk'event and clk = '1') THEN
         a_reg <= (a);
         b_reg <= (b);
            pdt_reg <= a_reg * b_reg;
         adder_out <= adder_out + pdt_reg;
      END IF;
     END process;
   accum_out <= (adder_out);
END rtl;
```

altsyncram & lpm_ram_dp—RAM ファンクションを HDL コードから推測

合成ツールは専用の RAM ブロックを搭載したデバイス・ファミリに対して、altsyncram または lpm_ram_dp メガファンクションに置換できるレジスタおよびロジックのセットを検出し、RAM ファンクションを推測します。

合成ツールは、シングル・ポートおよびシンプル・デュアル・ポート（1 つのリード・ポートと 1 つのライト・ポート）の RAM ブロックを認識します。汎用ロジックのレジスタを使用することによって、小さな RAM ブロックを効率的に実装できるので、通常は小さな RAM ブロックは推測しません。

Quartus II 合成機能を使用している場合､すべてのサイズの RAM ブロックを推測するよう指定できます。Assignments メニューの **Settings** をクリックします。**Category** リストで、**Analysis & Synthesis** をクリックします。**More Settings** をクリックします。**Existing Options Settings** で、オプション **Allow Any RAM Size for Recognition** を選択します。**Setting** 矢印をクリックし、**ON** を選択します。

デザインに合成ツールが認識も推測もしない RAM ブロックが含まれている場合、デザインが大量のシステム・メモリを要求し、これによってコンパイル問題が発生する可能性があります。

一部の合成ツールには、TriMatrix™ メモリ・ブロックを搭載するアルテラ・デバイス用に、推測された RAM ブロックの実装を制御するオプションが用意されています。例えば、Quartus II 合成機能では、値に“M512”、“M4K”、または “M-RAM” を使用してメモリ・ブロックのタイプを指定するか、または値 “logic” を使用して、専用メモリ・ブロックの代わりに通常のロジックの使用を指定する ramstyle 合成属性を使用します。

合成属性について詳しくは､「Quartus II ハンドブック Volume 1」の「合成」セクションの該当する章を参照してください。

フォーマル検証フローを使用している場合、アルテラでは RAM ロジックのみ搭載した別のエンティティまたはモジュールで RAM ブロックを作成することを推奨しています。Quartus II 合成機能を使用する場合などの特定のフォーマル検証フローでは、フォーマル検証ツールは RAM ブロックをサポートしないため、推測された RAM を持つエンティティまたはモジュールは自動的にブラック・ボックスに置かれます。Quartus II ソフトウェアは、このような場合は警告メッセージを発行します。エンティティまたはモジュールの RAM ブロックの外に、追加ロジックが搭載されている場合、このロジックもフォーマル検証ではブラック・ボックスとして処理する必要があるため検証できません。

デュアル・クロック同期 RAM

アルテラの TriMatrix メモリ・ブロックは同期型です。このため、これらの専用メモリ・ブロックを含むアーキテクチャをターゲットとした RAM デザインは、同期型でなければデバイス・アーキテクチャに直接マップすることはできません。同期メモリは、すべてのアルテラ・デバイス・ファミリでサポートされています。

同じ RAM アドレスに対する読み出しと書き込みが同時に起こると、アルテラ・デバイスの TriMatrix メモリ・ブロックは未定義のデータ値を返します。この点はオリジナルの HDL デザインの機能とは異なります。デザインで同じ RAM アドレスの読み出しと書き込みを行うときに特定の出力が必要になる場合、合成ツールにこれらのメモリに対する RAM 推測を無効にすることによって、デュアル・クロック・メモリに対する RAM ブロックを推測しないように指示します。

合成ツールの RAM 推測を無効にする具体的なオプションについては、合成ツールのマニュアルか、「Quartus II ハンドブック Volume 1」の「合成」セクションの該当する章を参照してください。

例 6–9 および 6–10 のコード・サンプルには、デュアル・クロック同期 RAM を推測する Verilog HDL と VHDL コードを示しています。

例 6–9. Verilog HDL デュアル・クロック同期 RAM

```
module ram_dual (q, addr_in, addr_out, d, we, clk1, clk2);
   output [7:0] q;
   input [7:0] d;
   input [6:0] addr_in;
   input [6:0] addr_out;
   input we, clk1, clk2;

   reg [6:0] addr_out_reg;
   reg [7:0] q;
   reg [7:0] mem [127:0];

   always @ (posedge clk1)
   begin
      if (we)
         mem[addr_in] <= d;
   end

   always @ (posedge clk2) begin
      q <= mem[addr_out_reg];
      addr_out_reg <= addr_out;
   end
endmodule
```

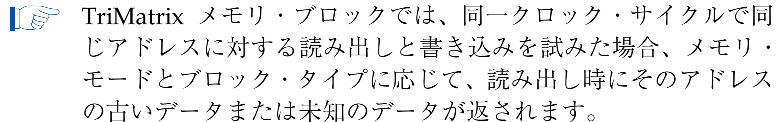
例 6–10. VHDL デュアル・クロック同期 RAM

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY ram_dual IS
    PORT (
        clock1, clock2: IN STD_LOGIC;
        data: IN STD_LOGIC_VECTOR (3 DOWNTO 0);
        write_address: IN INTEGER RANGE 0 to 31;
        read_address: IN INTEGER RANGE 0 to 31;
        we: IN STD_LOGIC;
        q: OUT STD_LOGIC_VECTOR (3 DOWNTO 0)
    );
END ram_dual;
ARCHITECTURE rtl OF ram_dual IS
    TYPE MEM IS ARRAY(0 TO 31) OF STD_LOGIC_VECTOR(3 DOWNTO 0);
    SIGNAL ram_block: MEM;
    SIGNAL read_address_reg : INTEGER RANGE 0 to 31;
BEGIN
    PROCESS (clock1)
    BEGIN
        IF (clock1'event AND clock1 = '1') THEN
            IF (we = '1') THEN
                ram_block(write_address) <= data;
            END IF;
        END IF;
    END PROCESS;
    PROCESS (clock2)
    BEGIN
        IF (clock2'event AND clock2 = '1') THEN
            q <= ram_block(read_address_reg);
            read_address_reg <= read_address;
        END IF;
    END PROCESS;
END rtl;
```

Read-Through-Write 動作のないシングル・クロック同期 RAM

この項のコード例には、シングル・クロック同期 RAM を推測する Verilog HDL と VHDL コードを示しています。アルテラの TriMatrix メモリ・ブロックは同期型です。このため、これらの専用メモリ・ブロックを含むアーキテクチャをターゲットとした RAM デザインは、同期型でなければデバイス・アーキテクチャに直接マップすることはできません。

これらの例では、TriMatrix メモリ・ブロックで直接サポートされていない、read-through-write 動作も回避しています。アルテラでは、デザインで RAM に read-through-write 動作が要求されない限り、すなわちデザインで同じ RAM 位置に対する同時読み出しおよび書き込みで、現在その RAM 位置に書き込み中の新しい値が要求されなければ、このコーディング・スタイルを使用することを推奨しています。



TriMatrix メモリ・ブロックでは、同一クロック・サイクルで同じアドレスに対する読み出しと書き込みを試みた場合、メモリ・モードとブロック・タイプに応じて、読み出し時にそのアドレスの古いデータまたは未知のデータが返されます。

Read-through-write 動作の RAM が必要な場合は、6–18 ページの「Read-Through-Write 動作対応シングル・クロック同期 RAM」の項を参照してください。

特定のデバイスの専用メモリ・ブロックについて詳しくは、アルテラ・ウェブサイト **www.altera.co.jp** の該当するアルテラ・デバイス・ファミリのデータ・シートを参照してください。

例 6–11 および 6–12 に示す RAM コード･サンプルは、アルテラ TriMatrix メモリに直接マップしています。

例 6–11. Read-Through-Write 動作非対応 Verilog HDL シングル・クロック同期 RAM

```
module ram_infer (q, a, d, we, clk);
   output reg [7:0] q;
   input [7:0] d;
   input [6:0] a;
   input we, clk;

   reg [7:0] mem [127:0];

   always @ (posedge clk) begin
      if (we)
         mem[a] <= d;
      q <= mem[a]; // q doesn't get d in this clock cycle
   end
endmodule
```

例 6–12. Read-Through-Write 非対応 VHDL シングル・クロック同期 RAM

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

ENTITY ram IS
   PORT (
      clock: IN STD_LOGIC;
      data: IN STD_LOGIC_VECTOR (2 DOWNTO 0);
      write_address: IN INTEGER RANGE 0 to 31;
      read_address: IN INTEGER RANGE 0 to 31;
      we: IN STD_LOGIC;
      q: OUT STD_LOGIC_VECTOR (2 DOWNTO 0)
   );
END ram;

ARCHITECTURE rtl OF ram IS
   TYPE MEM IS ARRAY(0 TO 31) OF STD_LOGIC_VECTOR(2 DOWNTO 0);
   SIGNAL ram_block: MEM;
BEGIN
   PROCESS (clock)
   BEGIN
      IF (clock'event AND clock = '1') THEN
         IF (we = '1') THEN
            ram_block(write_address) <= data;
         END IF;
         q <= ram_block(read_address);
            -- VHDL semantics imply that q doesn't get data
            -- in this clock cycle
      END IF;
   END PROCESS;
END rtl;
```

Read-Through-Write 動作対応シングル・クロック同期 RAM

TriMatrix メモリ・ブロックは、混合ポートでの read-through-write 動作はサポートしていません。これは、同一クロック・サイクルで同じアドレスに対する読み出しと書き込みを試みた場合、メモリ・モードとブロック・タイプに応じて、読み出し時にそのアドレスの古いデータまたは未知のデータが返されることを意味します。ただし、同じ位置に対して同時に読み出しと書き込みを行うと、読み出し時に現在その RAM 位置に書き込み中の新しい値が読み出される HDL コードで、RAM ブロックを説明することができます。以下の例では、このタイプの RAM ロジックを推測するコードを示しています。この働きをターゲット・デバイスに実装するために、合成ソフトウェアは RAM ブロックの周囲にバイパス・ロジックを追加します。RAM ブロックがデザインのクリティカル・パスに含まれる場合、このバイパス・ロジックにより、デザインの面積利用率が増加し性能が低下します。

例 6–13 および 6–14 に示す RAM 例では、RAM ブロックの周辺にバイパス・ロジックが必要です。

例 6–13. Read-Through-Write 動作対応 Verilog HDL シングル・クロック同期 RAM

```
module ram_infer (q, a, d, we, clk);
    output [7:0] q;
    input [7:0] d;
    input [6:0] a;
    input we, clk;
reg [6:0] read_add;
    reg [7:0] mem [127:0];
always @ (posedge clk) begin
        if (we)
            mem[a] <= d;
        read_add <= a;
    end
assign q = mem[read_add];
endmodule
```

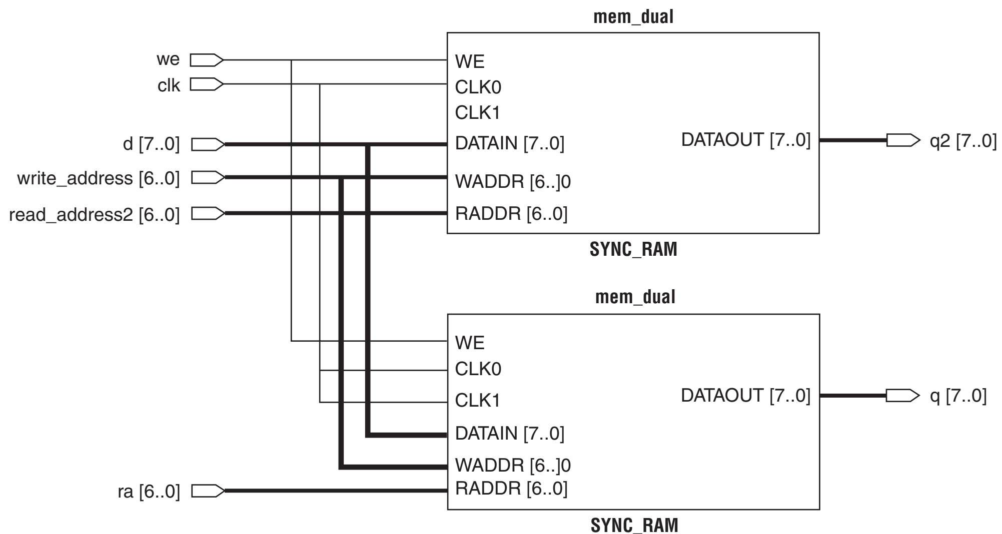
例 6–14. Read-Through-Write 動作対応 VHDL シングル・クロック同期 RAM

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

ENTITY ram IS
    PORT (
        clock: IN STD_LOGIC;
        data: IN STD_LOGIC_VECTOR (2 DOWNTO 0);
        write_address: IN INTEGER RANGE 0 to 31;
        read_address: IN INTEGER RANGE 0 to 31;
        we: IN STD_LOGIC;
        q: OUT STD_LOGIC_VECTOR (2 DOWNTO 0)
    );
END ram;

ARCHITECTURE rtl OF ram IS
    TYPE MEM IS ARRAY(0 TO 31) OF STD_LOGIC_VECTOR(2 DOWNTO 0);
    SIGNAL ram_block: MEM;
    SIGNAL read_address_reg: INTEGER RANGE 0 to 31;
BEGIN
    PROCESS (clock)
    BEGIN
        IF (clock'event AND clock = '1') THEN
            IF (we = '1') THEN
                ram_block(write_address) <= data;
            END IF;
            read_address_reg <= read_address;
        END IF;
    END PROCESS;
    q <= ram_block(read_address_reg);
END rtl;
```

2 つのリード・アドレスを持つ同期 RAM

Quartus II 合成機能は、2 つのリード・アドレスと 1 つのライト・アドレスを持つ RAM 記述から、RAM ブロックを推測できます。このタイプの RAM ブロックは、図 6-1 に示すように RAM ブロックを複製することによって実装できます。ブロックごとに独立しているリード・アドレスを除いて、両方の RAM ブロックのすべての入力が複製されます。

図 6-1. 2 つのリード・アドレスを持つ同期 RAM を示すブロック図



例 6–15 と 6–16 に示す 2 つのリード・アドレスを持つ RAM コード・サンプルは、RAM ブロックの複製により推測されています。

例 6–15. 2 つのリード・アドレスを持つ Verilog HDL シングル・クロック同期 RAM

```
module dual_ram_infer (q, q2, write_address, read_address, read_address2, d, we, clk);
      output reg [7:0] q;
      output reg [7:0] q2;
      input [7:0] d;
      input [6:0] write_address;
      input [6:0] read_address;
      input [6:0] read_address2;
      input we, clk;

      reg [7:0] mem [127:0];

      always @ (posedge clk) begin
            if (we)
                  mem[write_address] <= d;
            q <= mem[read_address];
            q2 <= mem[read_address2];
      end
endmodule
```

例 6–16. 2 つのリード・アドレスを持つ VHDL シングル・クロック同期 RAM

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

ENTITY dual_ram_infer IS
  PORT (
    clock: IN STD_LOGIC;
    data: IN STD_LOGIC_VECTOR (2 DOWNTO 0);
    write_address: IN INTEGER RANGE 0 to 31;
    read_address: IN INTEGER RANGE 0 to 31;
    read_address2: IN INTEGER RANGE 0 to 31;
    we: IN STD_LOGIC;
    q: OUT STD_LOGIC_VECTOR (2 DOWNTO 0);
    q2: OUT STD_LOGIC_VECTOR (2 DOWNTO 0)
    );
END dual_ram_infer;

ARCHITECTURE rtl OF dual_ram_infer IS
  TYPE MEM IS ARRAY(0 TO 31) OF STD_LOGIC_VECTOR(2 DOWNTO 0);
  SIGNAL ram_block: MEM;
BEGIN
  PROCESS (clock)
  BEGIN
    IF (clock'event AND clock = '1') THEN
      IF (we = '1') THEN
        ram_block(write_address) <= data;
      END IF;
      q <= ram_block(read_address);
      q2 <= ram_block(read_address2);
    END IF;
  END PROCESS;
END rtl;
```

非同期リード・アドレスを持つシングル・クロック同期 RAM

例 6–17 および 6–18 に示すコード・サンプルに、非同期リード・アドレスおよびレジスタ出力を備えた RAM の Verilog HDL と VHDL コードを示します。

以下の例の RAM コードの実装は、デバイス・ファミリの専用 RAM アーキテクチャによって異なります。APEX シリーズの RAM アーキテクチャは非同期リード・アドレスをサポートしているため、例えば、APEX デバイス・シリーズに非同期リード・アドレスを実装するのは簡単です。ただし、Stratix® デバイスおよび大部分の新しいデバイス・ファミリではリード・アドレスをラッチする必要があります。したがって、以下の例では非同期 RAM コードを直接実装することはできません。Stratix シリーズのデバイスなどに非同期 RAM を実装する場合、合成ツールで出力レジスタを RAM ブロックの入力に移動すれば、`altsyncram` メガファンクションを使用してロジックを実装できます。リード・クロックとライト・クロックが異なる場合、出力レジスタを RAM ブロックの入力に移動すると機能が多少変更されることがあります。このような状況では、合成ソフトウェアが警告を発行します。Quartus II 合成機能を使用している場合、Quartus II ヘルプで違いを説明しています。これらの RAM の例では、デバイスのアーキテクチャによっては直接 RAM ブロックにマップしない場合があります。

例 6–17. 非同期リード・アドレスを持つ Verilog HDL シングル・クロック同期 RAM

```
module ram (clock, data, write_address, read_address, we, q);
    parameter ADDRESS_WIDTH = 4;
    parameter DATA_WIDTH    = 8;
    input clock;
    input [DATA_WIDTH-1:0] data;
    input [ADDRESS_WIDTH-1:0] write_address;
    input [ADDRESS_WIDTH-1:0] read_address;
    input we;
    output [DATA_WIDTH-1:0] q;

    reg [DATA_WIDTH-1:0] q;
    reg [DATA_WIDTH-1:0] ram_block [2**ADDRESS_WIDTH-1:0];
    always @ (posedge clock)

    begin
        if (we)
            ram_block[write_address] <= data;
        q <= ram_block[read_address];
    end
endmodule
```

例 6–18. 非同期リード・アドレスを持つ VHDL シングル・クロック同期 RAM

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

ENTITY ram IS
    GENERIC (
        ADDRESS_WIDTH: integer := 4;
        DATA_WIDTH: integer := 8
    );
    PORT (
        clock: IN  std_logic;
        data: IN STD_LOGIC_VECTOR(DATA_WIDTH - 1 DOWNTO 0);
        write_address IN  STD_LOGIC_VECTOR (ADDRESS_WIDTH - 1 DOWNTO 0);
        read_address IN STD_LOGIC_VECTOR(ADDRESS_WIDTH - 1 DOWNTO 0);
        we: IN STD_LOGIC;
        q: OUT STD_LOGIC_VECTOR(DATA_WIDTH - 1 DOWNTO 0)
    );
END ram;

ARCHITECTURE rtl OF ram IS
    TYPE RAM IS ARRAY(0 TO 2 ** ADDRESS_WIDTH - 1) OF std_logic_vector(DATA_WIDTH - 1
DOWNTO 0);
    SIGNAL ram_block: RAM;
BEGIN
    PROCESS (clock)
    BEGIN
        IF (clock'event AND clock = '1') THEN
            IF (we = '1') THEN
                ram_block(TO_INTEGER(UNSIGNED(write_address))) <= data;
            END IF;
            q <= ram_block(TO_INTEGER(UNSIGNED(read_address)));
        END IF;
    END PROCESS;
END rtl;
```

初期メモリ内容の指定

合成ツールによっては、推測されるメモリの初期内容を指定できます。例えば、Quartus II 合成機能は、推測される RAM ブロックにメモリ初期化ファイル（**.mif**）を指定できる `ram_init_file` という合成属性をサポートしています。VHDL では、対応する信号にデフォルト値を指定することで、推測されるメモリの内容を初期化することもできます。Quartus II 合成機能では、推測される RAM のデフォルト値が MIF に自動的に変換されます。

`ram_init_file` 属性について詳しくは、「Quartus II ハンドブック Volume 1」の「Quartus II 合成機能」の章を参照してください。他の合成ツールの合成機能について詳しくは、ツール・ベンダのマニュアルを参照してください。

lpm_rom—ROM を HDL コードから推測

ROM ファンクションを推測する場合、合成ツールは専用の RAM ブロックを搭載したデバイス・ファミリに対してのみ、そのデバイス・ファミリに応じて `altsyncram` または `lpm_rom` メガファンクションに置換できるレジスタおよびロジックのセットを検出します。

フォーマル検証ツールは ROM メガファンクションをサポートしないため、Quartus II 合成機能はサードパーティ・フォーマル検証ツールが選択されている場合は、ROM メガファンクションを推測しません。

ROM が推測されるのは、`case` 文内のすべての選択肢に定数値が設定されている case 文が存在する場合です。小規模な ROM は、一般的に通常のロジックによるレジスタを使用して実装される場合に最高の性能を達成するため、各 ROM ファンクションが推測されメモリに配置されるには、最小サイズ要件を満たしている必要があります。

Quartus II 合成機能を使用している場合、ソフトウェアにすべてのサイズの ROM ブロックを推測するよう指示できます。Assignments メニューの **Settings** をクリックします。**Category** リストで、**Analysis & Synthesis** をクリックします。**More Settings** をクリックします。**Existing Options Settings** で、オプション **Allow Any ROM Size for Recognition** を選択します。**Setting** 矢印をクリックし、**ON** を選択します。

一部の合成ツールには、TriMatrix メモリ・ブロックを搭載するアルテラ・デバイス用に、推測された ROM ブロックの実装を制御するオプションが用意されています。例えば、Quartus II 合成機能では、値に “M512”、“M4K”、または “M-RAM” を使用してメモリ・ブロックのタイプを指定するか、または値 “logic” を使用して、専用メモリ・ブロックの代わりに通常のロジックの使用を指定する romstyle 合成属性を使用します。

合成属性について詳しくは、「Quartus II ハンドブック Volume 1」の「合成」セクションの該当する章を参照してください。フォーマル検証フローを使用している場合、アルテラでは ROM ロジックのみ搭載した別のエンティティまたはモジュールで ROM ブロックを作成することを推奨しています。これはフォーマル検証中に、エンティティとモジュールをブラック・ボックスとして処理する必要があるためです。

例 6–19 および 6–20 に示す Verilog HDL と VHDL コードのサンプルでは、同期 ROM ブロックを推測しています。デバイス・ファミリの専用 RAM アーキテクチャによっては、ROM ロジックが同期型でなければなりません。詳しくはデバイス・ファミリのハンドブックを参照してください。

Stratix シリーズ・デバイスや最近のデバイス・ファミリなど、同期 RAM ブロックを搭載したデバイス・アーキテクチャの場合、ROM コードを推測するためにアドレスまたは出力をラッチする必要があります。出力レジスタが使用される場合、これらのレジスタは Stratix RAM ブロックの入力レジスタを使用して実装されますが、ROM の機能は変更されません。アドレスをラッチする場合、推測される ROM のパワーアップ状態が HDL デザインと異なる場合があります。このようなシナリオでは、一般に合成ソフトウェアが警告を発行します。Quartus II ヘルプは、Quartus II 合成機能を使用するときに機能が変更される状況を説明しています。これらの ROM コード・サンプルは、アルテラ TriMatrix メモリ・アーキテクチャに直接マップしています。

例 6–19. Verilog HDL 同期 ROM

```
module sync_rom (clock, address, data_out);
    input clock;
    input [7:0] address;
    output [5:0] data_out;

    reg [5:0] data_out;

    always @ (posedge clock)
    begin
        case (address)
            8'b00000000: data_out = 6'b101111;
            8'b00000001: data_out = 6'b110110;
            ...
            8'b11111110: data_out = 6'b000001;
            8'b11111111: data_out = 6'b101010;
        endcase
    end
endmodule
```

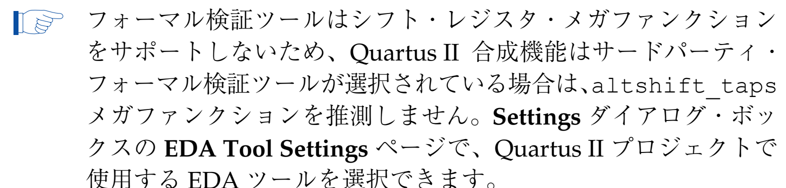
例 6–20. VHDL 同期 ROM

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

ENTITY sync_rom IS
    PORT (
        clock: IN STD_LOGIC;
        address: IN STD_LOGIC_VECTOR(7 downto 0);
        data_out: OUT STD_LOGIC_VECTOR(5 downto 0)
    );
END sync_rom;

ARCHITECTURE rtl OF sync_rom IS
BEGIN
PROCESS (clock)
    BEGIN
    IF rising_edge (clock) THEN
        CASE address IS
            WHEN "00000000" => data_out <= "101111";
            WHEN "00000001" => data_out <= "110110";
            ...
            WHEN "11111110" => data_out <= "000001";
            WHEN "11111111" => data_out <= "101010";
            WHEN OTHERS => data_out <= "101111";
        END CASE;
    END IF;
    END PROCESS;
END rtl;
```

altshift_taps— シフト・レジスタを HDL コードから推測

シフト・レジスタを推測する場合、合成ツールは同じ長さのシフト・レジスタ・グループを検出して、これらを altshift_taps メガファンクションに変換します。すべてのシフト・レジスタが検出されるには、以下の特性を備えている必要があります。

- 同じクロックおよびクロック・イネーブルを使用
- 他のセカンダリ信号がない
- タップがレジスタ 3 個以上分離して等間隔に配置されている。



フォーマル検証ツールはシフト・レジスタ・メガファンクションをサポートしないため、Quartus II 合成機能はサードパーティ・フォーマル検証ツールが選択されている場合は、altshift_taps メガファンクションを推測しません。**Settings** ダイアログ・ボックスの **EDA Tool Settings** ページで、Quartus II プロジェクトで使用する EDA ツールを選択できます。

フォーマル検証フローを使用している場合、アルテラではシフト・レジスタ・ロジックのみ搭載した別のエンティティまたはモジュールでシフト・レジスタ・ブロックを作成することを推奨しています。これはフォーマル検証中に、エンティティまたはモジュールをブラック・ボックスとして処理する必要があるためです。

合成ソフトウェアは専用の RAM ブロックを搭載するデバイス・ファミリのシフト・レジスタのみ認識し、特定のガイドラインに従って最適な実装を判断します。以下のガイドラインは Quartus II 合成機能で使用され、一般にサードパーティ EDA ツールでも使用されます。

- FLEX® 10K および ACEX® 1K デバイスの専用メモリ量は比較的小さいため、これらのデバイスの場合、`altshift_taps` メガファンクションは推測されません。
- APEX 20K および APEX II デバイスの場合、ソフトウェアはシフト・レジスタの総ビット数が 128 ビットを超える場合にのみ、`altshift_taps` メガファンクションを推測します。シフト・レジスタが小さい場合、通常は専用メモリへの実装で効果は得られません。
- Stratix および Cyclone™ シリーズのデバイスの場合、バスの幅（*W*）、各タップ間の長さ（*L*）、およびタップ数（*N*）に基づいて `altshift_taps` メガファンクションを推測するかどうかを判断します。
  - バス幅が 1 ($W = 1$) の場合、タップ数とタップ間の長 さを乗算した値が 64 以上 ($N \times L \geq 64$) であれば、`altshift_taps` が推測されます。
  - バス幅が 1 を超える場合 ($W > 1$)、バス幅、タップ数、各タップ間の長さを乗算した値が 32 以上であれば ($W \times N \times L \geq 32$)、`altshift_taps` が推測されます。

タップ間の長さ（*L*）が 2 の累乗にならない場合、リード・カウンタとライト・カウンタをデコードする別のロジックが使用されます。このような状況になるのは、サイズの異なるシフト・レジスタの場合、ファンクションの実装にロジック・エレメント（LE）またはアダプティブ・ロジック・モジュール（ALM）を使用する外部デコード・ロジックが必要になるためです。このデコード・ロジックによって、シフト・レジスタをメモリに実装する場合の性能上および利用上の利点が損なわれます。

`altshift_taps` メガファンクションにマップされ、RAM に配置されるレジスタは、合成後にはそれらのノード名が存在しないため、シミュレーション・ツールの Verilog HDL または VHDL ファイルでは使用できません。

以下の例ではシフト・レジスタが推測されます。

- Verilog HDL シングル・ビット幅、64 ビット長シフト・レジスタ
- タップが等間隔で配置された Verilog HDL 8 ビット幅、64 ビット長シフト・レジスタ
- タップが等間隔で配置された VHDL 8 ビット幅、64 ビット長シフト・レジスタ

Verilog HDL シングル・ビット幅、64 ビット長シフト・レジスタ

例 6–21 に示す Verilog HDL コード・サンプルは、シンプルなシングル・ビット幅、64 ビット長のシフト・レジスタを示しています。合成ソフトウェアは、サポートされるデバイスに対して、altshift_taps メガファンクションにレジスタ（$W = 1$ と $M = 64$）を実装します。レジスタの長さが 64 ビット未満の場合、シフト・レジスタはロジックに実装されます。

例 6–21. Verilog HDL シングル・ビット幅、64 ビット長シフト・レジスタ

```
module shift_1x64 (clk, shift, sr_in, sr_out);
    input clk, shift;
    input sr_in;
    output sr_out;

    reg [63:0] sr;

    always @ (posedge clk)
    begin
        if (shift == 1'b1)
        begin
            sr[63:1] <= sr[62:0];
            sr[0] <= sr_in;
        end
    end
    assign sr_out = sr[63];
endmodule
```

タップが等間隔で配置された Verilog HDL 8 ビット幅、64 ビット長シフト・レジスタ

例 6–22 と 6–23 に示すコード・サンプルには、タップが 15、31、47 番目と等間隔で配置された Verilog HDL および VHDL 8 ビット幅、64 ビット長のシフト・レジスタ（$W > 1$ と $M = 64$）を示しています。合成ソフトウェアはこのファンクションを 1 つの altshift_taps メガファンクションに実装し、それをサポートされるデバイスの RAM にマップします。

例 6–22. タップが等間隔で配置された Verilog HDL 8 ビット幅、64 ビット長シフト・レジスタ

```
module shift_8x64_taps (clk, shift, sr_in, sr_out, sr_tap_one, sr_tap_two, sr_tap_three );
    input clk, shift;
    input [7:0] sr_in;
    output [7:0] sr_tap_one, sr_tap_two, sr_tap_three, sr_out;

    reg [7:0] sr [63:0];
    integer n;

    always @ (posedge clk)
    begin
        if (shift == 1'b1)
        begin
            for (n = 63; n>0; n = n-1)
            begin
                sr[n] <= sr[n-1];
            end
            sr[0] <= sr_in;
        end

    end
    assign sr_tap_one = sr[15];
    assign sr_tap_two = sr[31];
    assign sr_tap_three = sr[47];
    assign sr_out = sr[63];
endmodule
```

例 6–23. タップが等間隔で配置された VHDL 8 ビット幅、64 ビット長シフト・レジスタ

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY shift_8x64_taps IS
    PORT (
        clk: IN STD_LOGIC;
        shift: IN STD_LOGIC;
        sr_in: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        sr_tap_one: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        sr_tap_two : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        sr_tap_three: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
        sr_out: OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
    );
    END shift_8x64_taps;

ARCHITECTURE arch OF shift_8x64_taps IS
    SUBTYPE sr_width IS STD_LOGIC_VECTOR(7 DOWNTO 0);
    TYPE sr_length IS ARRAY (63 DOWNTO 0) OF sr_width;
    SIGNAL sr: sr_length;
BEGIN
    PROCESS (clk)
    BEGIN
        IF (clk'EVENT and clk = '1') THEN
            IF (shift = '1') THEN
                sr(63 DOWNTO 1) <= sr(62 DOWNTO 0);
                sr(0) <= sr_in;
            END IF;
        END IF;
    END PROCESS;
    sr_tap_one <= sr(15);
    sr_tap_two <= sr(31);
    sr_tap_three <= sr(47);
    sr_out <= sr(63);
END arch;
```

デバイス固有のコーディング・ガイドライン

この項では、アルテラ・デバイスのアーキテクチャに対するデバイス固有のコーディングの推奨事項について説明します。作成した HDL コードを合成ツールがどのようにターゲットのアルテラ・デバイスにマップするかを理解することが重要です。レジスタおよび特定のロジック構造を、該当するアルテラ・デバイスのアーキテクチャに合わせて設計することによって、デザインの結果の品質を大幅に改善できます。

アルテラ・デバイスのレジスタ・パワーアップ値

デバイス・コアのレジスタは、すべてのアルテラ・デバイスで常に Low (0) ロジック・レベルでパワーアップします。ただし、レジスタが High (1) ロジック・レベルでパワーアップしたのと同様に動作するようロジックを実装する方法があります。

レジスタ・アーキテクチャでプリセットをサポートしないデバイス上で、プリセット信号を使用する場合、合成ツールはプリセット信号をクリア信号に変換することがありますが、それには合成で NOT ゲート・プッシュバックと呼ばれる最適化を実行する必要があります。NOT- ゲート・プッシュバックにより、レジスタの入力および出力にインバータが追加されるため、リセット状態とパワーアップ状態は High になりますが、デバイスは期待どおりに動作します。この場合、合成ツールはパワーアップ状態を伝えるメッセージを発行することがあります。レジスタ自体は Low でパワーアップしますが、レジスタ出力が反転されるので、すべてのディスティネーションで到着する信号は High です。

このような影響により、特定のリセット値(0 以外)を指定すると、合成ツールはレジスタで利用できる同期クリア(`aclr`)信号を使用して、`NOT` ゲート・プッシュバックで High ビットを実装する場合があります。その場合、レジスタは指定されたリセット値にパワーアップしているように見えます。この動作が見られるのは、デザインが FLEX 10KE または ACEX デバイスをターゲットにしている場合です。

デバイスでロード信号が使用できる場合、合成ツールは 1 または 0 の非同期ロードを使用して 1 または 0 値のリセットを実装できます。合成ツールが非同期ロード信号を使用する場合、`NOT` ゲート・プッシュバックは実行しないため、レジスタは 0 ロジック・レベルでパワーアップします。

詳しくは、該当するデバイス・ファミリのハンドブック、またはアルテラ・ウェブサイト www.altera.co.jp で該当するハンドブックを参照してください。

設計者は、通常デザインに対して明示的なリセット信号を使用します。この信号によって、必ずしもパワーアップ時でなくても、リセット後にすべてのレジスタが適切な値に強制されます。非同期リセットによってボードが安全な状態で動作できるようにデザインを作成しておき、リセットがアクティブな状態でデザインを立ち上げることができます。これがデバイスのパワーアップ状態に依存しない、適切な方法です。

レジスタの非同期コントロール・ポートをドライブする前に、デバイス・アーキテクチャの外部ロジックまたは組み合わせロジックを同期させることによって、デザインの安定性を高め、潜在的なグリッチの発生を防止することができます。

適切な同期デザイン方法について詳しくは、「Quartus II ハンドブック Volume 1」の「Design Recommendations for Altera Devices」の章を参照してください。

デザインに特定のパワーアップ状態を強制する場合は、合成ツールで使用可能な合成オプションを使用します。Quartus II 合成機能により、**Power-Up Level** ロジック・オプションを適用できます。またソース・コードで `altera_attribute` アサインメントを指定したオプションも適用可能です。このオプションを使用すると、合成ツールは実際にはコア・レジスタのパワーアップ状態を変更できないため、合成で強制的に `NOT` ゲート・プッシュバックが実行されます。

Quartus II 合成機能の **Power-Up Level** アサインメントは、特定のレジスタまたはデザインのエンティティ、モジュール、またはサブデザインに適用できます。この場合、適用されたブロック内のすべてのレジスタに対するアサインメントとなります。レジスタはデフォルトでは 0 でパワーアップするため、このアサインメントを使用すると、`NOT` ゲート・プッシュバックを使用して、すべてのレジスタが 1 でパワーアップするように設定することができます。

`NOT` ゲート・プッシュバックをグローバル・アサインメントとして使用すると、多数のインバータが必要になるため、結果の品質が多少低下する場合があることに注意してください。状況によっては、イネーブルまたはセカンダリ・コントロール・ロジックの推測によって問題が生じます。またこのようなデザインを、ASIC または HardCopy® デバイスに移行するのが困難な場合もあります。初期化を行っている場合は、機能シミュレーションでパワーアップ動作をシミュレートできます。

Power-Up Level オプションと `altera_attribute` については、「Quartus II ハンドブック Volume 1」の「Quartus II 合成機能」の章で説明しています。

VHDL では、合成ツールによっては、レジスタの初期化デバイスに実装することも可能です。例えば、Quartus II 合成機能は、レジスタの VHDL デフォルト値を Power-Up Level 設定に変換します。このように合成された動作は、機能シミュレーションでの VHDL コードのパワーアップ状態に一致します。

例えば、以下のコードでは q のレジスタが推測され、そのパワーアップ・レベルを High（リセット値は 0）に設定します。

```
SIGNAL q : STD_LOGIC := '1'; -- q has a default value of '1'

PROCESS (clk, reset)
BEGIN
   IF (reset = '1') THEN
      q <= '0';
   ELSIF (rising_edge(clk)) THEN
      q <= d;
   END IF;
END PROCESS;
```

ほとんどの合成ツールと同様に、Quartus II ソフトウェアは Verilog HDL 初期ブロックを合成しません。したがって、このツールは初期ブロックにおける変数への値の代入を合成しません。

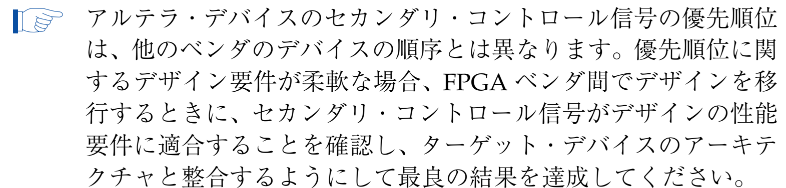
クリア & クロック・イネーブルなどのセカンダリ・レジスタ・コントロール信号

FPGA デバイス・アーキテクチャは、フリップフロップとしても知られるレジスタをベースにしています。アルテラ FPGA のレジスタは複数のセカンダリ・コントロール信号（クリア信号やイネーブル信号など）を提供しており、この信号を使用すると、特別なロジック・セルを使用しないで、各レジスタ用のコントロール・ロジックを実装できます。セカンダリ信号のサポートはデバイス・ファミリごとに異なるため、デバイス・ファミリのデータシートを参照して、ターゲット・デバイスで使用可能な信号を確認してください。

デバイスで信号を最も効率的に活用するには、HDL コードが可能な限りデバイス・アーキテクチャに一致している必要があります。アーキテクチャの性質のために、コントロール信号には一定の優先順位が設定されているため、可能な場合はその優先順位に従って HDL コードを作成する必要があります。

合成ツールは通常のロジックを使用して、すべてのコントロール信号をエミュレートできるため、常に適切な機能の結果が得られます。ただし、コントロール信号が使用される条件および優先順位の点でデザイン要件に柔軟性がある場合は、デザインをターゲット・デバイスのアーキテクチャに一致させ、最も効率的な結果を達成してください。デザインの信号の優先順位がターゲット・アーキテクチャと一致しない場合は、コントロール信号を実装するための特別なロジックが必要になることがあります。この特別なロジックでは、さらに別のデバイス・リソースが使用され、コントロール信号の遅延が増加する可能性があります。

また、状況によっては、デバイス・アーキテクチャで専用コントロール・ロジック以外のロジックを使用すると大きな影響を及ぼす可能性があります。例えば、クロック・イネーブル信号は、デバイス・アーキテクチャの同期リセット信号やクリア信号よりも優先順位が高くなります。クリア信号は同期信号ですが、クロック・イネーブルはロジック・アレイ・ブロック（LAB）のクロック・ラインをオフにします。このため、デバイス・アーキテクチャでは、同期クリアが有効になるのはクロック・エッジが起こった場合に限定されます。クロック・イネーブル信号よりも優先順位の高い同期クリア信号を使用してレジスタをコーディングする場合、レジスタのデータ入力を使用して、クロック・イネーブル機能をエミュレートする必要があります。信号はレジスタのクロック・イネーブル・ポートを使用しないため、Clock Enable Multicycle 制約は適用できません。この場合、これらのコントロール信号の優先順位は、デバイスで使用できる信号の優先順位に従うのが最適であるのは明らかであり、異なる優先順位を使用すると、クロック・イネーブル信号へのアサインメントにより予測とは異なる結果が生じます。



アルテラ・デバイスのセカンダリ・コントロール信号の優先順位は、他のベンダのデバイスの順序とは異なります。優先順位に関するデザイン要件が柔軟な場合、FPGA ベンダ間でデザインを移行するときに、セカンダリ・コントロール信号がデザインの性能要件に適合することを確認し、ターゲット・デバイスのアーキテクチャと整合するようにして最良の結果を達成してください。

アルテラのすべてのデバイス・ファミリで信号の順序は同じですが、前述したとおり、すべてのデバイス・ファミリがすべての信号を供給するとは限りません。以下の優先順序を遵守してください。

1. Asynchronous Clear, `aclr`—highest priority
2. Preset, `pre`
3. Asynchronous Load, `aload`
4. Enable, `ena`
5. Synchronous Clear, `sclr`
6. Synchronous Load, `sload`
7. Data In, `data`—lowest priority

以下の例では、前述した aclr、aload、ena コントロール信号を使用してレジスタを作成する Verilog HDL および VHDL コードを示しています。

dff_all.v にはセンシティビティ・リストに adata が含まれていませんが、**dff_all.vhd** は含まれています。これは Verilog HDL 言語の制限です。— 非同期ロード信号を記述する方法はありません（aload が High の間、adata が切り替わると q が切り替わる）。このような制限が課せられていても、すべての合成ツールはこの構造から aload 信号を推測する必要があります。合成ツールがこのような推測を実行する場合、合成ツールから情報または警告メッセージが発行されることがあります。

例 6–24. ena、aclr、aload コントロール信号を使用した Verilog HDL D 型フリップフロップ（レジスタ）

```
module dff_control(clk, aclr, aload, ena, data, adata, q);
    input clk, aclr, aload, ena, data, adata;
    output q;

    reg q;

    always @ (posedge clk or posedge aclr or posedge aload)
    begin
        if (aclr)
            q <= 1'b0;
        else if (aload)
            q <= adata;
        else
            if (ena)
                q <= data;
    end
endmodule
```

例 6–25. ena、aclr、aload コントロール信号を使用した VHDL D 型フリップフロップ（レジスタ）

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY dff_control IS
    PORT (
      clk: IN STD_LOGIC;
      aclr: IN STD_LOGIC;
      aload: IN STD_LOGIC;
      adata: IN STD_LOGIC;
      ena: IN STD_LOGIC;
       data: IN STD_LOGIC;
      q: OUT STD_LOGIC
    );
END dff_control;

ARCHITECTURE rtl OF dff_control IS
BEGIN
    PROCESS (clk, aclr, aload, adata)
    BEGIN
        IF (aclr = '1') THEN
          q <= '0';
        ELSIF (aload = '1') THEN
          q <= adata;
        ELSE
            IF (clk = '1' AND clk'event) THEN
                IF (ena ='1') THEN
                q <= data;
                END IF;
            END IF;
        END IF;
    END PROCESS;
END rtl;
```

プリセット信号は、より柔軟な aload 信号に置き換えられるため、多くのデバイス・ファミリには備わっていません。したがって、この例にはプリセット信号は含まれていません。

異なる sload および sclr 信号を備えたレジスタを数多く作成すると、sclr および sload 信号は LAB ワイド信号（LAB 内で共通の信号）なので、Quartus II フィッタでレジスタを LAB にパッキングするのが困難になることがあります。また LAB ワイドの sload 信号を使用すると、デバイスが持っている高速フィードバック・パスを使用してレジスタをパッキングできなくなります。これは一部のレジスタは同一 LAB 内の他のロジックでパッキングできないことを意味します。

したがって、合成ツールはルック・アップ・テーブル（LUT）にスペースがある場合は通常、sload または sclr 信号の使用を避けます。LUT が使用できる場合は、LUT を使用してこれらの信号を実装したほうが常

に柔軟性が高くなります。sload 信号および sclr 信号の用途は通常、演算チェイン（カウンタ）、または良好な LAB パッキングを可能にする共通信号を備えた十分なレジスタ数を持つワイド・マルチプレクサなどの特定のファンクションに限定されています。デバイス・ファミリごとにコントロール信号の数が異なるため、これらの信号の推測もデバイス独自に行われます。例えば、Stratix II デバイスはセカンダリ・コントロール信号については Stratix デバイスよりも柔軟なため、合成ツールはより多くの Stratix II デバイス用の sload および sclr 信号を推測する可能性があります。

これらの追加コントロール信号を使用する場合は、デバイス・アーキテクチャに合致する優先順序で使用してください。最も効率的な結果を達成するためには、先の例の aclr が aload よりも優先順位が高いのと同様に、sclr 信号の優先順位が sload 信号よりも高くなるように設計してください。デザインが上記の条件を満たさない場合、これらのレジスタ専用信号は推測されないことを覚えておいてください。このような場合、Quartus II ソフトウェアは汎用ロジック・リソースを使って HDL で記述されたとおりの回路を合成します。

Verilog HDL では、sload および sclr を用いた以下のコードは（モジュール宣言にコントロール信号を追加した後）、例 6–24 に示す Verilog HDL の例の q <= data 文を置き換えることができます。

例 6–26. Verilog HDL sload & sclr

```
if (sclr)
    q <= 1'b0;
else if (sload)
    q <= sdata;
else
        q <= data;
```

VHDL では、sload と sclr を用いた以下のコードは（エンティティ宣言にコントロール信号を追加した後）、例 6–25 に示す VHDL の例の q <= data 文を置き換えることができます。

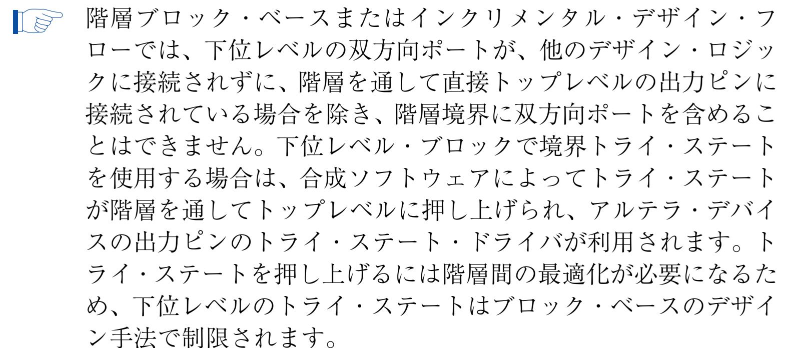
例 6–27. VHDL sload & sclr

```
IF (sclr = '1') THEN
    q <= '0';
ELSIF (sload = '1') THEN
    q <= sdata;
ELSE
        q <= data;
```

Tri-State Signals

アルテラ・デバイスをターゲットにしている場合、トップレベルの双方向ピンまたは出力ピンに接続されているときにのみトライ・ステート信号を使用する必要があります。下位レベルの双方向ピンは避け、出力ピンまたは双方向ピンをドライブしている場合を除き、Z ロジック値を使用するのは避けてください。

合成ツールはマルチプレクサ・ロジックを使用して、内部トライ・ステート信号を含むデザインをアルテラ・デバイスに正しく実装しますが、アルテラではこのコーディング方法は推奨していません。



階層ブロック・ベースまたはインクリメンタル・デザイン・フローでは、下位レベルの双方向ポートが、他のデザイン・ロジックに接続されずに、階層を通して直接トップレベルの出力ピンに接続されている場合を除き、階層境界に双方向ポートを含めることはできません。下位レベル・ブロックで境界トライ・ステートを使用する場合は、合成ソフトウェアによってトライ・ステートが階層を通してトップレベルに押し上げられ、アルテラ・デバイスの出力ピンのトライ・ステート・ドライバが利用されます。トライ・ステートを押し上げるには階層間の最適化が必要になるため、下位レベルのトライ・ステートはブロック・ベースのデザイン手法で制限されます。

例 6–28 および 6–29 に示すコード例は、トライ・ステート双方向信号を作成する Verilog HDL と VHDL コードを示しています。

例 6–28. Verilog HDL トライ・ステート信号

```
module tristate (myinput, myenable, mybidir);
    input myinput, myenable;
    inout mybidir;
    assign mybidir = (myenable ? myinput : 1'bZ);
endmodule
```

例 6–29. VHDL トライ・ステート信号

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

ENTITY tristate IS
PORT (
   mybidir : INOUT STD_LOGIC;
   myinput : IN STD_LOGIC;
   myenable : IN STD_LOGIC
     );
END tristate;

ARCHITECTURE rtl OF tristate IS
BEGIN
    mybidir <= 'Z' WHEN (myenable = '0') ELSE myinput;
END rtl;
```

アダー・ツリー

ターゲットにしたアルテラ・デバイスのアーキテクチャに合わせてアダー・ツリーを適切に構築すると、性能と集積度が大幅に改善される場合があります。大きなアダー・ツリーを使用するアプリケーションの例として、有限インパルス応答（FIR）フィルタがあります。パイプライン化されたバイナリまたはターナリ・アダー・ツリーを適切に使用すると、結果の品質を大幅に改善できます。

この項では、アルテラの 4 入力 LUT デバイス、および現在 Stratix II デバイスでのみ提供されている 6 入力 LUT ロジック構造について、コーディング推奨事項が異なる理由を説明します。

ロジック・エレメントに 4 入力 LUT を使用したアーキテクチャ

Stratix シリーズ、Cyclone シリーズ、APEX シリーズ、および FLEX シリーズのデバイスなどのアーキテクチャは、LE の標準組み合わせ構造として 4 入力 LUT を使用しています。

デザインがパイプライン化可能な場合、4 入力ルックアップ・テーブルを使用するデバイスにおいて 3 つの数値（`A`、`B`、`C`）を最も高速で加算する方法は、`A + B` を計算し、出力をラッチし、ラッチされた出力を `C` に加算することです。`A + B` の加算には 1 レベルのロジック（1 つの LE で 1 ビットが加算される）が必要なので、最大クロック速度で実行されます。これは必要なだけ拡張できます。

例 6–30 に示すコード・サンプルでは、5 つの数字（A、B、C、D、E）が追加されています。4 入力ルックアップ・テーブルを使用するデバイスで 5 つの数字を追加する場合、合計 64 の LE（16 ビット数値の場合）に 4 つのアダーと 3 レベルのレジスタが必要です。

例 6–30. Verilog HDL パイプライン・バイナリ・ツリー

```
module binary_adder_tree (A, B, C, D, E, CLK, OUT);
    parameter WIDTH = 16;
    input [WIDTH-1:0] A, B, C, D, E;
    input CLK;
    output [WIDTH-1:0] OUT;

    wire [WIDTH-1:0] sum1, sum2, sum3, sum4;
    reg [WIDTH-1:0] sumreg1, sumreg2, sumreg3, sumreg4;
    // Registers

    always @ (posedge CLK)
        begin
            sumreg1 <= sum1;
            sumreg2 <= sum2;
            sumreg3 <= sum3;
            sumreg4 <= sum4;
        end

    // 2-bit additions
    assign sum1 = A + B;
    assign sum2 = C + D;
    assign sum3 = sumreg1 + sumreg2;
    assign sum4 = sumreg3 + E;
    assign OUT = sumreg4;
endmodule
```

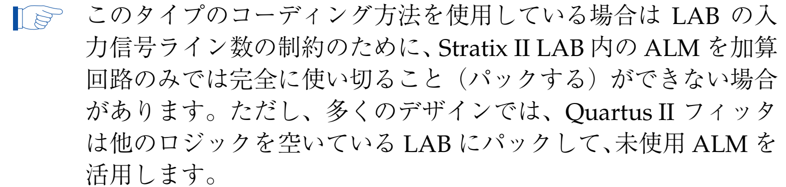
アダプティブ・ロジック・エレメントに 6 入力 LUT を使用したアーキテクチャ

Stratix II アーキテクチャは基本ロジック構造に 6 入力 LUT を使用するため、Stratix II デバイスは上記の 4 入力 LUT を用いた例とは異なるコーディング・スタイルのほうが有利です。特に Stratix II デバイスの ALM は、3 ビットを同時に加算できます。したがって、上記の例のツリーは深度が 2 レベルで、4 つの add-by-two の代わりに、2 つの add-by-three で実現できることになります。

Stratix II デバイスの上記の例のコードはコンパイルに成功しますが、このコードは非効率であり、6 入力アダプティブ・ルック・アップ・テーブル（ALUT）を活用していません。ツリーをターナリ・ツリーとして再構築することによって、デザインはより効率化され、集積度の利用率も大幅に改善されます。したがって、Stratix II デバイスをターゲットに

する場合、Stratix II デバイス・アーキテクチャを活用するには、4 入力 LUT アーキテクチャ向けにデザインされたパイプライン・バイナリ・アダー・ツリーは書き直す必要があります。

例 6–31 は Stratix II デバイスの 32 個の ALUT のみで実現できます。



このタイプのコーディング方法を使用している場合は LAB の入力信号ライン数の制約のために、Stratix II LAB 内の ALM を加算回路のみでは完全に使い切ること（パックする）ができない場合があります。ただし、多くのデザインでは、Quartus II フィッタは他のロジックを空いている LAB にパックして、未使用 ALM を活用します。

例 6–31. Verilog HDL パイプライン・ターナリ・ツリー

```
module ternary_adder_tree (A, B, C, D, E, CLK, OUT);
    parameter WIDTH = 16;
    input [WIDTH-1:0] A, B, C, D, E;
    input CLK;
    output [WIDTH-1:0] OUT;

    wire [WIDTH-1:0] sum1, sum2;
    reg [WIDTH-1:0] sumreg1, sumreg2;
    // Registers

    always @ (posedge CLK)
        begin
            sumreg1 <= sum1;
            sumreg2 <= sum2;
        end

    // 3-bit additions
    assign sum1 = A + B + C;
    assign sum2 = sumreg1 + D + E;
    assign OUT = sumreg2;
endmodule
```

上記の例では、パイプライン化された加算器を示していますが、加算演算を分割すると非パイプライン化加算器でも結果が向上します。デザインがパイプライン化されていない場合、ターナリ・ツリーはバイナリ・ツリーよりもはるかに性能が高くなります。例えば、合成ツールによっては、HDL コード `sum = (A + B + C) + (D + E)` の方が括弧を使用しないコードに比べ最適な実装、すなわち 3 入力アダー（`A + B + C` に対応）とそれに続く 3 入力アダー（`sum1 + D + E` に対応）が生成される可能性が高くなります。括弧を追加しない場合、合成ツールはアーキテクチャに最適ではない方法で加算を分割する可能性があります。

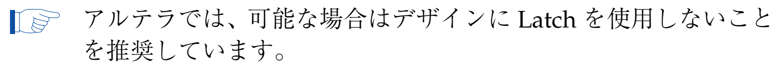
その他のロジック構造のコーディング・ガイドライン

この項では、以下のロジック構造のコーディングのガイドラインを示します。

- Latch— アルテラは可能であればLatch を使用しないことを推奨していますが、この項では Latch が必要な場合に、正しく使用するためのガイドラインも記載しています。
- ステート・マシン — この項は、ステート・マシンを使用するときに最良の結果を得るのに役立ちます。
- マルチプレクサ — この項では一般的な問題に対処し、マルチプレクサ・デザインに対して最適なリソース利用を実現するためのデザイン・ガイドラインを示しています。
- CRC（Cyclic Redundancy Check）チェック機能 — この項では、CRC 機能をデザインする場合に良好な結果を得るためのガイドラインを示します。

Latch

Latch は、新しい値が割り当てられるまで信号の値を保持する、小さな組み合わせループです。



アルテラでは、可能な場合はデザインに Latch を使用しないことを推奨しています。

Latch およびすべての組み合わせループを使用したデザインに関連する問題について詳しくは、「Quartus II ハンドブック Volume 1」の「Design Recommendations for Altera Devices」の章を参照してください。

Latch を使用する意図がないときは、「意図しない Latch の生成」で詳述するように、HDL コードから Latch を推測できます。Latch を推測させる意図がある場合は、6–44 ページの「Latch の正しい推測」で詳述するように、正しく推測して適切なデバイス動作を保証する必要があります。

意図しない Latch の生成

組み合わせロジックを設計する場合、特定のコーディング・スタイルでは意図しない Latch が生成される可能性があります。例えば、`CASE` 文または `IF` 文で可能なすべての条件に対応していないときには、新しい出力値が割り当てられない場合に出力を保持する Latch が必要になることがあります。合成ツールのメッセージで、推測される Latch への参照を確認してください。コードで意図せずに Latch が作成された場合は、コードの変更を行って Latch を削除します。

Latch はフォーマル検証ツールでのサポートが制限されていました。したがって、デザイン・フローでフォーマル検証を使用している場合は、不完全な CASE 文などを通じて、意図しない Latch が推測されないようにしてください。

Verilog HDL デザインで full_case 属性を使用すると、指定されていない case を don't care 値(X)として扱うことができます。ただし、full_case 属性を使用すると、シミュレーションのミスマッチが生じる可能性があります。この属性は合成専用属性であり、シミュレーション・ツールは指定されていない case を Latch として扱うためです。

合成ツールでの属性の使用について詳しくは、「Quartus II ハンドブック Volume 1」の「合成」セクションの該当する章を参照してください。「Quartus II 合成機能」の章には、可能性のあるシミュレーションのミスマッチを説明した例が記載されています。

IF 文または CASE 文の最後の ELSE 句または WHEN OTHERS 句を省略しても、Latch が生成される可能性があります。デフォルト条件の Don't care(X)アサインメントは、Latch の生成を防止するのに有効です。最高のロジック最適化を達成するには、デフォルトの CASE または最終的な ELSE 値を、ロジック値ではなく don't care(X)に割り当ててください。

例 6–32 に示す VHDL サンプル・コードでは、意図しない Latch 生成が防止されます。最後の ELSE 句を省略すると、このコードでは sel 入力の残りの組み合わせに対応する意図しない Latch が作成されます。このコードで Stratix デバイスをターゲットにする場合、最後の ELSE 条件を省略すると、合成ソフトウェアは最大 6 つの LE を使用してしまいますが、ELSE 文を使用すれば 3 つで済みます。また、ELSE 文で X の値を割り当てる代わりに 1 を割り当てると、合成ソフトウェアは don't care 値の指定により、より良い最適化を実行できます。LE が増加することがあります。

例 6–32. 意図しない Latch 作成を防ぐ VHDL コード

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

ENTITY nolatch IS
    PORT (a,b,c: IN STD_LOGIC;
        sel: IN STD_LOGIC_VECTOR (4 DOWNTO 0);
        oput: OUT STD_LOGIC);
END nolatch;

ARCHITECTURE rtl OF nolatch IS
BEGIN
    PROCESS (a,b,c,sel) BEGIN
        IF sel = "00000" THEN
            oput <= a;
        ELSIF sel = "00001" THEN
            oput <= b;
        ELSIF sel = "00010" THEN
            oput <= c;
        ELSE                  --- Prevents latch inference
            oput <= ''X'; --/
        END IF;
    END PROCESS;
END rtl;
```

Latch の正しい推測

合成ツールは、一般に組み合わせループに関連した問題を含まない安全な Latch を推測できます。

Quartus II 合成機能を使用する場合、ソフトウェアで推測される Latch はコンパイル・レポートの **User-Specified** and **Inferred Latches** セクションで報告されます。このレポートには、その Latch が安全かどうか、タイミング・ハザードがないかどうかが示されます。

設計したデザインの Latch または組み合わせループが、**User-Specified and Inferred Latches** レポートに記載されていない場合、ソフトウェアによって安全な Latch として推測されておらず、グリッチ・フリーと見なされていないことを意味します。

Compilation Report の **Analysis & Synthesis Logic Cells Representing Combinational Loops** テーブルにリストされるすべての組み合わせループは、タイミング・ハザードが危険な状態にあります。これらのエントリは、調査が必要なデザインの問題があることを示しています。ただし、組み合わせループを含む正しいデザインを作成することは可能です。例えば、組み合わせループを非センシティブにすることは可能です。これはハードウェア中に存在する経路について、設計者はその経路をアクティブにするデータが回路に発生することがないこと、あるいはデータ

入力には関係なく、その経路がセンシティブになるのを防止する相互排他的な方法で周辺ロジックがセットアップされていることを把握しています。

MAX® 7000AE および MAX 3000A などのマクロセル・ベースのデバイスの場合、**Analysis & Synthesis User-Specified and Inferred Latches** テーブルにリストされているすべての data（D 型）Latch と set-reset（S-R）Latch が、グリッチなどのタイミング・ハザードなしで実装されています。グリッチの発生を防ぐ補正項が実装され、1 つのマクロセルによりフィードバック・ループが形成されます。

Stratix デバイス、Cyclone シリーズ、および MAX II デバイスなどの 4 入力 LUT ベースのデバイスの場合、**User-Specified and Inferred Latches** テーブルのフィードバック・ループに 1 つの LUT を持つすべての Latch は、1 つの入力のみが変化するのであればタイミング・ハザードは生じません。ある 1 つの入力が変化しても同じ出力値になるのであれば、出力にはグリッチが発生しないという LUT の性質があるためです。例えば、イネーブル入力が非アクティブのときに D 型入力がトグルしたり、または優先順位の高いリセット入力がアクティブのときに set 入力がトグルする場合です。LUT のこのハードウェア動作は、1 つの LUT のみで構成されたループには補正項が不要であることを意味します。Quartus II ソフトウェアは、可能な場合は常にフィードバック・ループで 1 つの LUT を使用します。入力にフィードバックされる出力に加えて、data、enable、set、reset 入力を備えた Latch は、1 つの 4 入力 LUT には実装できません。入力が多すぎるため、Quartus II ソフトウェアが 1 つの LUT ループで Latch を実装できない場合は、**User-Specified and Inferred Latches** テーブルに Latch にタイミング・ハザードがあることが示されます。

Stratix II などの 6 入力 LUT ベースのデバイスの場合、1 つのアダプティブ・ルック・アップ・テーブル（ALUT）で、組み合わせループ内にすべての Latch 入力を実装できます。したがって、**User-Specified and Inferred Latches** テーブルのすべての Latch では、1 つの入力のみが変化するのであればタイミング・ハザードはありません。

ハザードのない動作を確実にするために、同時に変更できるのは 1 つのコントロール入力のみです。Set と reset を同時にデアサートするなど、2 つの入力を同時に変更したり、または data と enable を同時に変更すると、Latch は不正な動作を行います。

Quartus II 合成機能では、Verilog HDL の `always` ブロック、および VHDL の `process` 文から Latch が推測されますが、Verilog HDL での連続アサインメントや VHDL の同時信号アサインメントからは推測され

ません。これらのルールはレジスタの推測の場合と同じです。レジスタまたはフリップフロップは、always ブロックおよび process 文から推測されます。以下の例では Latch が推測されます。

Verilog HDL set-reset Latch の例

例 6–33 に示す Verilog HDL コードのサンプルでは、Quartus II ソフトウェアで S-R Latch が正しく推測されます。

例 6–33. Verilog HDL set-reset Latch

```
module simple_latch (
   input SetTerm,
   input ResetTerm,
   output reg LatchOut
   );

   always @ (SetTerm or ResetTerm) begin
      if (SetTerm)
         LatchOut = 1'b1;
      else if (ResetTerm)
         LatchOut = 1'b0;
      end
endmodule
```

HDL Data 型 Latch の例

例 6–34 に示す VHDL コードのサンプルでは、Quartus II ソフトウェアで D 型 Latch が正しく推測されます。

例 6–34. HDL Data 型 Latch

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

ENTITY simple_latch IS
   PORT (
      enable, data     : IN STD_LOGIC;
      q                : OUT STD_LOGIC
   );
END simple_latch;

ARCHITECTURE rtl OF simple_latch IS
BEGIN

latch : PROCESS (enable, data)
   BEGIN
      IF (enable = '1') THEN
         q <= data;
      END IF;
   END PROCESS latch;
END rtl;
```

以下の例は、Quartus II ソフトウェアで Latch が推測されない Verilog HDL の連続アサインメントを示します。動作は Latch に類似していますが、Latch としては正しく機能しない場合があり、Latch としてのタイミング解析は行われません。

```
assign latch_out = en ? data : latch_out;
```

Quartus II 合成機能では、lpm_latch メガファンクションのインスタンス化が可能な場合は安全な Latch を作成します。このメガファンクションを使用して、data、enable、set、reset の各入力を組み合わせて Latch を作成します。安全な Latch を作成する場合、HDL コードから Latch を推測する場合と同じ制限が適用されます。

アルテラのlpm_latchファンクションを別の合成ツールで推測すると、この実装も Quartus II ソフトウェアは Latch として認識します。サードパーティ合成ツールで、lpm_latch メガファンクションを使用して Latch を実装する場合、Quartus II 合成機能では、HDL ソース・コードで作成された Latch がリストされるのと同様に、**User-Specified and Inferred Latches** テーブルに Latch がリストされます。lpm_latch 実装の生成に必要なコーディング・スタイルは、使用する合成ツールによって異なります。サードパーティ合成ツールによっては、推測された lpm_latch ファンクション数をリストする場合があります。

Quartus II 合成機能によって Latch が **User-Specified and Inferred Latches** テーブルにリストされ、Analysis & Synthesis で安全な Latch として実装される場合、フィッタでのフィジカル・シンセシス・ネットリストの最適化により、ハザードのない性能が維持されます。

LUT ベースのファミリの場合、フィッタは Analysis & Synthesis で Latch イネーブルとして識別される信号を含むコントロール信号に対して、グローバル配線を使用します。場合によっては、グローバル挿入遅延により、タイミング性能が低下することがあります。必要であれば、Quartus II の **Global Signal** ロジック・オプションをオフにして、手動でグローバル信号の使用を防ぐことができます。グローバルLatchイネーブルは、コンパイル・レポートの **Global & Other Fast Signals** テーブルにリストされます。

ステート・マシン

合成ツールは、Verilog HDL と VHDL のステート・マシンを認識し、エンコードできます。この項では、ステート・マシンを使用する場合に最高の結果を得るためのガイドラインを示します。合成ツールが HDL コードをステート・マシンとして認識できるようにすると、ツールによる状態変数の再コーディングが可能になり、結果の品質が向上します。また、

ツールでステート・マシンのプロパティを使用できるようになり、デザインの他の部分が最適化されます。合成でステート・マシンが認識されるとき、多くの場合はデザインの面積と性能を改善できます。

最高の結果を達成するために、合成ツールは FPGA デバイスには one-hot エンコーディングを、CPLD デバイスには minimal-bit エンコーディングを使用することがよくあります。ただし、実装の選択はステート・マシンやデバイスごとに異なります。ステート・マシンのエンコーディング方法を制御する具体的な方法については、使用する合成ツールのマニュアルを参照してください。

Quartus II 合成機能でのステート・マシンのエンコーディングについて詳しくは、「Quartus II ハンドブック Volume 1」の「Quartus II Integrated Synthesis」の章にある「State Machine Processing」の項を参照してください。

ステート・マシンの適切な認識と推測を確実にし、結果の品質の向上させるために、アルテラでは Verilog HDL および VHDL の両方に適用される以下のガイドラインに従うことを推奨しています。

- 合成で不要な Latch が生成されないように、ステート・マシンから得られた出力にデフォルト値を割り当ててください。
- 出力値の割り当てを含むすべての演算ファンクションおよびデータ・パスから、ステート・マシンのロジックを切り離します。
- デザインに複数の状態で使用される演算が含まれる場合、ステート・マシン外部での演算を定義し、ステート・マシンの出力ロジックでこの値が使用されるようにします。
- シンプルな非同期または同期リセットを使用して、定義済みのパワーアップ状態にします。ステート・マシンのデザインに、非同期リセットと非同期ロードの両方が存在するなど、より込み入ったリセット・ロジックが含まれる場合、Quartus II ソフトウェアはステート・マシンを推測しないで、通常のロジックを生成します。

デバイスに問題があるためステート・マシンが不正な状態に移行した場合、ステート・マシンが次にリセットされるまで、デザインの正しい機能が停止する可能性があります。デフォルトでは、合成ツールはこのような状況への対処を提供しません。システムに何らかの障害がある場合、ステート・マシン以外の他のレジスタでも同様の問題が発生します。`default` または `when others` 句を指定しても、デザインが意図的にこの状態に移行しない場合は、この動作には影響しません。デフォルト・ステートにより生成されたロジックは、合成ツールにより削除されます。

合成ツールによっては、default case が存在しない場合はそれを挿入し、不正な状態を処理するデザインのロジックを保持する安全なステート・マシンを実装するオプションを提供するものがあります。ステート・マ

シンが何らかの理由で不正な状態に移行した場合、次のクロック・エッジでリセット状態に戻ります。当然ながら、このオプションではステート・マシンのみが保護され、この方法ではデザインの他のすべてのレジスタは保護されません。

ステート・マシンを実装するためのツール固有のオプションについて詳しくは、ツール・ベンダのマニュアルか、「Quartus II ハンドブック Volume 1」の「合成」セクションの該当する章を参照してください。

以下の「Verilog HDL ステート・マシン」および「VHDL ステート・マシン」の 2 つの項では、言語別ガイドラインとコーディング例を説明します。

Verilog HDL ステート・マシン

Verilog HDL ステート・マシンを適切に認識および推測するために、以下の Verilog HDL の補足ガイドラインに従ってください。これらのガイドラインの一部は、Quartus II 合成機能のみ対象としています。具体的なコーディングの推奨事項については、使用する合成ツールのマニュアルを参照してください。

- SystemVerilog 規格を使用する場合は、列挙型を使用してステート・マシンを記述します（6–52 ページの「SystemVerilog ステート・マシンのコーディング例」を参照）。
- ステート・マシンの状態を、Verilog-1995 および -2001 の `parameter` データ型で表し、パラメータを使用してステート・アサインメントを作成します、（以下の「Verilog HDL ステート・マシンのコーディング例」を参照）。この実装によりステート・マシンを読みやすくなり、コーディング時にエラーが発生するリスクが低減されます。

アルテラでは、`next_state <= 0` など、状態変数に整数値を直接使用しないこと推奨しています。ただし、整数を使用しても Quartus II ソフトウェアで推測が防止されることはありません。

■ 状態遷移ロジックにおいて以下の例のように算術演算が使用された場合、Quartus II ソフトウェアではステート・マシンは推測されません。

```
case (state)
    0: begin
        if (ena) next_state <= state + 2;
        else next_state <= state + 1;
    end
    1: begin
    ...
endcase
```

■ 状態変数が出力となる場合、Quartus II ソフトウェアではステート・マシンは推測されません。

Verilog HDL ステート・マシンのコーディング例

以下のモジュール verilog_fsm は、標準的な Verilog HDL ステート・マシンの実装の例です。

このステート・マシンには 5 つの状態があります。非同期リセットは変数ステートを state_0 に設定します。in1 と in2 の和は、state_1 と state_2 のステート・マシンの出力になります。差（in1 - in2）も state_1 と state_2 で使用されます。一時変数 tmp_out_0 と tmp_out_1 は、in1 と in2 の和と差を保存します。これらの一時変数をステート・マシンのさまざまな状態で使用すると、相互排他的状態間でリソース共有が適切に行われます。

例 6–35. Verilog-2001 ステート・マシン

```
module verilog_fsm (clk, reset, in_1, in_2, out);
    input clk;
    input reset;
    input [3:0] in_1;
    input [3:0] in_2;output [4:0] out;
    parameter state_0 = 3'b000;
    parameter state_1 = 3'b001;
    parameter state_2 = 3'b010;
    parameter state_3 = 3'b011;
    parameter state_4 = 3'b100;

    reg [4:0] tmp_out_0, tmp_out_1, tmp_out_2;
    reg [2:0] state, next_state;

    always @ (posedge clk or posedge reset)
    begin
        if (reset)
            state <= state_0;
        else
            state <= next_state;
```

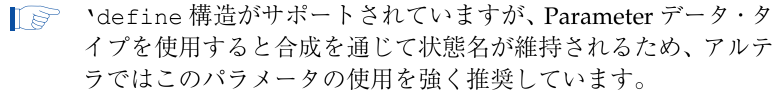
```
    end
    always @ (state or in_1 or in_2)
    begin
        tmp_out_0 = in_1 + in_2;
        tmp_out_1 = in_1 - in_2;
        case (state)
            state_0: begin
               tmp_out_2 <= in_1 + 5'b00001;
               next_state <= state_1;
            end
            state_1: begin
                if (in_1 < in_2) begin
                    next_state <= state_2;
                    tmp_out_2 <= tmp_out_0;
                end
                else begin
                    next_state <= state_3;
                    tmp_out_2 <= tmp_out_1;
                end
            end
            state_2: begin
                tmp_out_2 <= tmp_out_0 - 5'b00001;
                next_state <= state_3;
            end
            state_3: begin
                tmp_out_2 <= tmp_out_1 + 5'b00001;
                next_state <= state_0;
            end
            state_4:begin
                tmp_out_2 <= in_2 + 5'b00001;
                next_state <= state_0;
            end
            default:begin
                tmp_out_2 <= 5'b00000;
                next_state <= state_0;
            end
        endcase
    end
    assign out = tmp_out_2;
endmodule
```

このステート・マシンの同等の実装は、以下のように、parameter データ型の代わりに `define を使用すると達成できます。

```
`define state_0 3'b000
`define state_1 3'b001
`define state_2 3'b010
`define state_3 3'b011
`define state_4 3'b100
```

このケースでは、state アサインメントと next_state アサインメントには、以下の例に示すように state_x ではなく `state_x が割り当てられます。

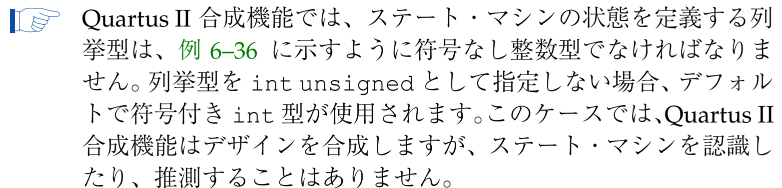
```
next_state <= 'state_3;
```



`define 構造がサポートされていますが、Parameter データ・タイプを使用すると合成を通じて状態名が維持されるため、アルテラではこのパラメータの使用を強く推奨しています。

SystemVerilog ステート・マシンのコーディング例

例 6–36 に示すモジュール enum_fsm は、列挙型を使用する System Verilog ステート・マシンの実装例です。アルテラでは、このコーディング・スタイルを使用して SystemVerilog のステート・マシンを記述することを推奨しています。



Quartus II 合成機能では、ステート・マシンの状態を定義する列挙型は、例 6–36 に示すように符号なし整数型でなければなりません。列挙型を int unsigned として指定しない場合、デフォルトで符号付き int 型が使用されます。このケースでは、Quartus II 合成機能はデザインを合成しますが、ステート・マシンを認識したり、推測することはありません。

例 6–36. 列挙型を使用する SystemVerilog ステート・マシン

```
module enum_fsm (input clk, reset, input int data[3:0], output int o);

   enum int unsigned { S0 = 0, S1 = 2, S2 = 4, S3 = 8 } state, next_state;

   always_comb begin : next_state_logic
      next_state = S0;
      case(state)
        S0: next_state = S1;
        S1: next_state = S2;
        S2: next_state = S3;
        S3: next_state = S3;
      endcase
   end

   always_comb begin
      case(state)
         S0: o = data[3];
         S1: o = data[2];
         S2: o = data[1];
         S3: o = data[0];
      endcase
   end

   always_ff@(posedge clk or negedge reset) begin
      if(~reset)
         state <= S0;
      else
         state <= next_state;
   end
endmodule
```

VHDL ステート・マシン

VHDL ステート・マシンが正しく認識および推測されるように、ステート・マシンの状態は列挙型で表し、対応する型を使用してステート・アサインメントを実行します。この実装によりステート・マシンを読みやすくなり、コーディング時にエラーが発生するリスクが低減されます。状態が列挙型で表されていない場合、合成ソフトウェア（Quartus II 合成機能など）はステート・マシンを認識しません。代わりに、ステート・マシンは通常のロジック･ゲートおよびレジスタとして実装され､ステート・マシンは Compilation Report の **Analysis & Synthesis** セクションにステート・マシンとしてリストされません。

VHDL ステート・マシンのコーディング例

以下のエンティティ `vhd1_fsm` は、VHDL の通常のステート・マシンの実装例です。

このステート・マシンには5つの状態があります。非同期リセットは変数状態を state_0 に設定します。in1 と in2 の和は、state_1 と state_2 のステート・マシンの出力になります。差（in1 - in2）も state_1 と state_2 で使用されます。一時変数 tmp_out_0 と tmp_out_1 は、in1 と in2 の和と差を保存します。これらの一時変数をステート・マシンのさまざまな状態で使用すると、相互排他的状態間でリソース共有が適切に行われます。

例 6–37. VHDL ステート・マシン

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY vhdl_fsm IS
   PORT(
      clk: IN STD_LOGIC;
      reset: IN STD_LOGIC;
      in1: IN UNSIGNED(4 downto 0);
      in2: IN UNSIGNED(4 downto 0);
      out_1: OUT UNSIGNED(4 downto 0)
      );
END vhdl_fsm;

ARCHITECTURE rtl OF vhdl_fsm IS
   TYPE Tstate IS (state_0, state_1, state_2, state_3, state_4);
   SIGNAL state: Tstate;
   SIGNAL next_state: Tstate;
BEGIN
   PROCESS(clk, reset)
   BEGIN
         IF reset = '1' THEN
            state <=state_0;
      ELSIF rising_edge(clk) THEN
            state <= next_state;
      END IF;
   END PROCESS;
   PROCESS (state, in1, in2)
      VARIABLE tmp_out_0: UNSIGNED (4 downto 0);
      VARIABLE tmp_out_1: UNSIGNED (4 downto 0);
   BEGIN
      tmp_out_0 := in1 + in2;
      tmp_out_1 := in1 - in2;
      CASE state IS
         WHEN state_0 =>
            out_1 <= in1;
            next_state <= state_1;
         WHEN state_1 =>
            IF (in1 < in2) then
               next_state <= state_2;
               out_1 <= tmp_out_0;
            ELSE
               next_state <= state_3;
               out_1 <= tmp_out_1;
            END IF;
```

```
        WHEN state_2 =>
           IF (in1 < "0100") then
              out_1 <= tmp_out_0;
           ELSE
              out_1 <= tmp_out_1;
           END IF;
              next_state <= state_3;
        WHEN state_3 =>
              out_1 <= "11111";
              next_state <= state_4;
        WHEN state_4 =>
              out_1 <= in2;
              next_state <= state_0;
        WHEN OTHERS =>
              out_1 <= "00000";
              next_state <= state_0;
     END CASE;
  END PROCESS;
END rtl;
```

マルチプレクサ

多くの FPGA デザインで、マルチプレクサはロジック利用の大きな部分を占めます。マルチプレクサ・ロジックを最適化すると、アルテラ・デバイスに最も効率的に実装することができます。この項では一般的な問題に対処し、マルチプレクサ・デザインに対して最適なリソース利用を実現するためのデザイン・ガイドラインを示しています。また各種のマルチプレクサ、およびアルテラの Stratix デバイスなど、多くの FPGA アーキテクチャで使用されている 4 入力 LUT へのマルチプレクサの実装方法について説明します。

Stratix II デバイスでは 6 入力 LUT が使用されますが、ここでは特に説明しません。最適化の原理と手法の多くは類似していますが、Stratix II の 6 入力 LUT デバイスではデバイス利用が異なります。例えば、Stratix II デバイスは、LE の 4 入力 LUT に実装されるものより幅の広いマルチプレクサを 1 つの ALM に実装できます。

マルチプレクサのタイプ

この最初の項では、さまざまなタイプの HDL コードからマルチプレクサを作成する方法を説明します。`CASE` 文、`IF` 文、およびステート・マシンはすべて、デザインでのマルチプレクサ・ロジックの共通のソースになります。これらの HDL 構造から、バイナリ・マルチプレクサ、セレクタ・マルチプレクサ、プライオリティ・マルチプレクサなど、さまざまなタイプのマルチプレクサが作成されます。HDL コードからのマル

チプレクサの作成方法および合成時の実装方法を理解することが、マルチプレクサ構造を最適化し最良の結果を得るための最初の一歩になります。

バイナリ・マルチプレクサ

バイナリ・マルチプレクサは、バイナリ・エンコード選択ビットに基づいて入力を選択します。以下の「Verilog HDL バイナリ・エンコード Case 文」の例は、簡単な 4:1 バイナリ・マルチプレクサを記述する Verilog HDL コードを示しています。

例 6–38. Verilog HDL バイナリ・エンコード Case 文

```
case (sel)
   2'b00: z = a;
   2'b01: z = b;
   2'b10: z = c;
   2'b11: z = d;
endcase
```

4:1 バイナリ・マルチプレクサは、2 つの 4 入力 LUT を使用すると効率的に実装されます。4:1 マルチプレクサを使用する大規模なバイナリ・マルチプレクサを構築できます。4:1 マルチプレクサのツリーから *N* 入力マルチプレクサ（*N*:1 マルチプレクサ）を構築すると、わずか 0.66*（*N* - 1）の LUT を使用した構造となります。

セレクタ・マルチプレクサ

セレクタ・マルチプレクサには、各データ入力に個別の選択ラインがあります。マルチプレクサの選択ラインは、one-hot エンコードされます。以下の「Verilog HDL one-hot エンコード Case 文」の例には、one-hot セレクタ・マルチプレクサを記述する簡単な Verilog HDL コードの例を示しています。

例 6–39. Verilog HDL one-hot エンコード Case 文

```
case (sel)
   4'b0001: z = a;
   4'b0010: z = b;
   4'b0100: z = c;
   4'b1000: z = d;
   default: z = 1'bx;
endcase
```

セレクタ・マルチプレクサは、AND ゲートと OR ゲートのツリーとして一般に構築されます。この方式を使用すると、2 つの AND ゲートと 1 つの OR ゲートを使用する 1 つの 4 入力 LUT の 2 本の選択ラインを使用し

て、2 つの入力を選択できます。これらの LUT の出力は、ワイド OR ゲートと組み合わせることができます。この構造の *N*-入力セレクタ・マルチプレクサには、最低でも 0.66*（*N*-0.5）の LUT が必要です。これは最良のバイナリ・マルチプレクサよりもわずかに多くなります。

プライオリティ・マルチプレクサ

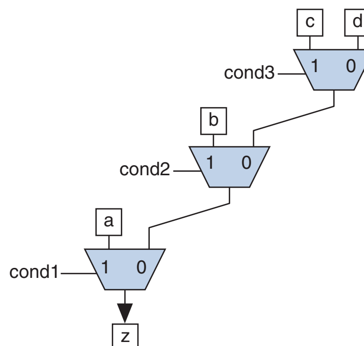
プライオリティ・マルチプレクサでは、選択ロジックが優先順位を示します。信号の優先順位に基づく特定の順序で、正しい項目を選択するオプションをチェックする必要があります。これらの構造は一般に、VHDL または Verilog HDL の `IF`、`ELSE`、`WHEN`、`SELECT` または `?:` 文から作成されます。「優先順位を示す VHDL IF 文」の項の VHDL コード例は、図 6-2 に示すような回路図を実装します。

例 6–40. 優先順位を示す VHDL IF 文

```
IF cond1 THEN z <= a;
ELSIF cond2 THEN z <= b;
ELSIF cond3 THEN z <= c;
ELSE z <= d;
END IF;
```

図 6-2 に示すマルチプレクサは、チェインを形成し、各条件または選択ビットを一度に 1 つずつ評価します。

図 6-2. IF 文のプライオリティ・マルチプレクサの実装



N-入力プライオリティ・マルチプレクサは、チェイン内の 2:1 マルチプレクサごとに 1 つの LUT を使用するため、*N*-1 の LUT を必要とします。マルチプレクサのこのチェインにより一般に遅延が増加します。これはロジックを通過するクリティカル・パスがチェイン内のすべてのマルチプレクサを横断しているためです。

マルチプレクサのタイミング遅延を改善するために、優先順位が要求されない場合はプライオリティ・マルチプレクサの使用を回避してください。デザインで選択順序が重要ではない場合は、CASE 文を使用して、プライオリティ・マルチプレクサの代わりに、バイナリまたはセレクタ・マルチプレクサを実装します。優先順位を要求する多重化デザインで、構造全体での遅延が重要な場合は、デザインを再コーディングしてロジック・レベル数を減らし、特にクリティカル・パスに沿った遅延を最小化することを検討してください。

デフォルトまたはその他の Case アサインメント

CASE 文で case を完全に指定するには、DEFAULT（Verilog HDL）アサインメントまたは OTHERS（VHDL）アサインメントを含めます。このアサインメントは、選択ラインの多くの組み合わせが未使用の one-hot エンコーディング方式で特に重要になります。未使用の選択ラインの組み合わせに case を指定し、合成ツールにこれらの case の合成方法の情報を提供してください。これは Verilog HDL および VHDL 言語仕様で要求されます。

デザインによっては、主に設計者がこれらの case が起こらないと想定するという理由から、未使用 case の結果を考慮する必要がない場合があります。これらのタイプのデザインでは、DEFAULT アサインメントまたは OTHERS アサインメントに任意の値を選択できます。ただし、選択するアサインメント値がデザインの実装に必要なロジック利用率に大きく影響する可能性があることに注意してください。これはアサインメントの値ごとに合成ツールの処理方法が異なり、また速度および面積の最適化方法が異なるためです。

一般に、最良の結果を得るには、無効な case を定義済みの case の 1 つと組み合わせるのではなく、無効な CASE 選択を単独の DEFAULT 文または OTHERS 文で明示的に定義します。

無効な case の値が重要でない場合は、X（don't care）ロジック値を割り当てることによって、無効な case を明示的に指定します。このアサインメントにより、合成ツールで最良の面積の最適化を実行できます。

HDL デザインおよび合成ツールに異なる DEFAULT または OTHERS アサインメントを使って実験し、デザインのロジック利用に及ぼす影響をテストできます。

暗黙のデフォルト

Verilog HDL および VHDL の IF 文は、CASE 型では容易に対応できない条件を指定する便利な方法といえます。ただし、IF 文を使用すると、マルチプレクサ・ツリーが複雑になって、容易に合成ツールで最適化できなくなります。

特に、すべての IF 文に、指定されていない場合でも、暗黙の ELSE 条件が指定されます。これらの暗黙のデフォルトにより、多重化デザインでは一層複雑になる可能性があります。

「暗黙のデフォルトの VHDL IF 文」の以下のコードの例では、4 入力（a、b、c、d）と 1 出力（z）のマルチプレクサを示しています。

例 6–41. 暗黙のデフォルトの VHDL IF 文

```
IF cond1 THEN
   IF cond2 THEN
      z <= a;
   END IF;
ELSIF cond3 THEN
   IF cond4 THEN
      z <= b;
   ELSIF cond5 THEN
      z <= c;
   END IF;
ELSIF cond6 THEN
   z <= d;
END IF;
```

このコードは 4:1 マルチプレクサを実装しているようですが、コード内の 3 つの IF 文のそれぞれには、指定されていない暗黙の ELSE 条件が含まれています。ELSE case の出力値が指定されていないため、合成ツールはこれらのcaseに対して同じ出力値を維持することが意図されていると仮定します。

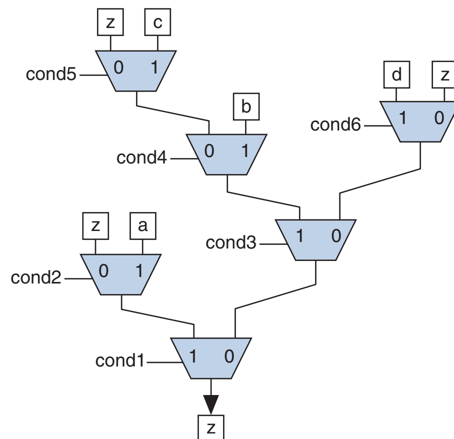
例 6–42 に示すコード・サンプルには、例 6–41 に示すコードと同じ機能を持つコードが示されていますが、ELSE case を明示的に指定しています。

例 6–42. デフォルト条件を明示的に指定した VHDL IF 文

```
IF cond1 THEN
   IF cond2 THEN
      z <= a;
   ELSE
      z <= z;
   END IF;
ELSIF cond3 THEN
   IF cond4 THEN
      z <= b;
   ELSIF cond5 THEN
      z <= c;
   ELSE
      z <= z;
   END IF;
ELSIF cond6 THEN
   z <= d;
ELSE
   z <= z;
END IF;
```

図 6-3 は例 6–42 のコードを表す回路図です。4 つの入力しかありませんが、マルチプレクサ・ロジックが基本の 4:1 マルチプレクサよりもかなり複雑であることを示しています。

図 6-3. 暗黙のデフォルトでの IF 文のマルチプレクサ実装



多重化ロジックを簡略化し、不要なデフォルトを削除できる方法がいくつかあります。最適な方法は、ロジックが 4:1 CASE 文の構造を使用するようにデザインを再コーディングすることです。あるいは、優先順位が重要な場合、コードを再構築してデフォルトの case を推測し、マルチプレクサをフラット化することができます。この例では、IF cond1 THEN IF cond2 の代わりに、同じ機能を実行する IF（cond1 AND cond2）を使用してください。また、デフォルトが don't care case かどうかを確認します。この例では、他に有効な case が起こらなければ、最後の ELSIF cond6 文を ELSE 文に変更することができます。

マルチプレクサ・ロジック内の不要なデフォルト条件をなくすことで、デザインの実装を簡単にし、ロジック利用率を低減することができます。

縮退マルチプレクサ

縮退マルチプレクサは、可能性のある case のすべてが固有のデータ入力に使用されていないマルチプレクサです。不要な case は、これらのマルチプレクサのロジック利用の効率低下の原因になる傾向があります。degenerate マルチプレクサが、完全なバイナリ・マルチプレクサで効率的なロジック利用を行えるように、このマルチプレクサを再コーディングできます。

バイナリ・マルチプレクサの選択ライン数によって通常、希望の機能を実装するのに必要なマルチプレクサのサイズが決まります。例えば、図 6-4 に示すマルチプレクサ構造には、16 入力を備えたバイナリ・マルチプレクサを実装可能な 4 本の選択ラインがあります。しかし、このデザインは 16 入力をすべて使用しておらず、このマルチプレクサは 16:1 縮退マルチプレクサになります。

例 6–43. 縮退マルチプレクサを記述する VHDL CASE 文

```
CASE sel[3:0] IS
   WHEN "0101" => z <= a;
   WHEN "0111" => z <= b;
   WHEN "1010" => z <= c;
   WHEN OTHERS => z <= d;
END CASE;
```

図 6-4. バイナリ縮退マルチプレクサ

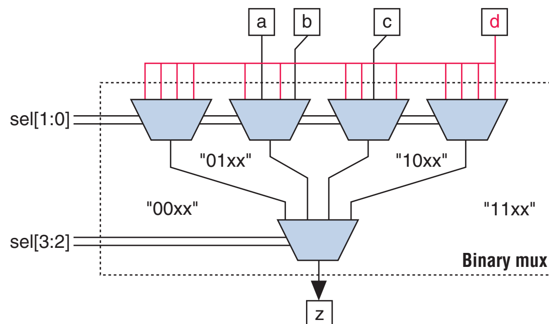
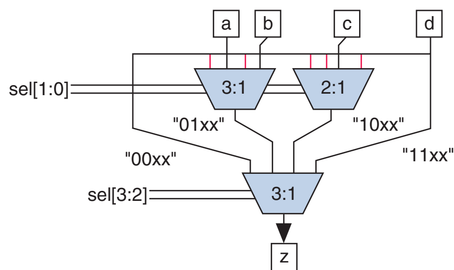


図 6-4 の例では、トップ・レベルの 1 番目と 4 番目のマルチプレクサを容易になくすことができます。これは各マルチプレクサの 4 入力がすべて同じ値であり、図 6-5 に示すように、他のマルチプレクサの入力数を減らすことができるためです。

図 6-5. 縮退バイナリ・マルチプレクサの最適化バージョン



このバージョンのマルチプレクサを実装する場合、残りの 3:1 マルチプレクサのそれぞれに 2 個ずつ、2:1 マルチプレクサに 1 個と最低でも 5 個の 4 入力 LUT が必要になります。このデザインは 4 入力からのみ出力を選択します。4:1 バイナリ・マルチプレクサでは、最適に実装できるのは 2 個の LUT なので、この縮退マルチプレクサ・ツリーではロジックの効率が低下します。

この構造のロジック利用率は、完全な 4:1 バイナリ・マルチプレクサを実装できるように選択ラインを再コーディングすることによって改善できます。例 6–44 のコード・サンプルは、オリジナルの選択ラインをバイナリ・エンコーディングにより 1 つの `z_sel` に変換するレコーダ・デザインを示しています。

例 6–44. 縮退バイナリ・マルチプレクサ用 VHDL レコーダ・デザイン

```
CASE sel[3:0] IS
   WHEN "0101" => z_sel <= "00";
   WHEN "0111" => z_sel <= "01";
   WHEN "1010" => z_sel <= "10";
   WHEN OTHERS => z_sel <= "11";
END CASE;
```

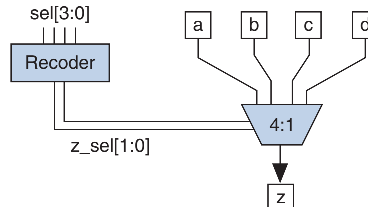
例 6–45 のコード・サンプルに、完全なバイナリ・マルチプレクサの実装方法を示します。

例 6–45. VHDL4:1 バイナリ・マルチプレクサ・デザイン

```
CASE z_sel[1:0] IS
   WHEN "00" => z <= a;
   WHEN "01" => z <= b;
   WHEN "10" => z <= c;
   WHEN "11" => z <= d;
END CASE;
```

レコーダ・デザインの新しい z_sel コントロール信号を使用して、4 つの入力 a、b、c、d のいずれかを選択する 4:1 バイナリ・マルチプレクサを制御します（図 6-6 を参照)。選択ラインの複雑さは Recoder デザインで処理され、データ・マルチプレクシングは最も効率的な実装を可能にする、シンプルなバイナリ選択ラインで実行されます。

図 6-6. Recoder によるバイナリ・マルチプレクサ



このレコーダのデザインは 2 つの LUT に実装でき、また効率的な 4:1 バイナリ・マルチプレクサが 2 つの LUT を使用するため、LUT の合計は 4 つになります。オリジナルの縮退マルチプレクサは 5 つの LUT を要求していました。このため、再コーディング・バージョンで使用するロジックはオリジナルよりも 20% 少なくなります。

マルチプレクサのロジック利用率は、選択ラインを完全なバイナリ case に再コーディングすると改善されることがよくあります。エンコーディングを実行するためのロジックが必要ですが、多くの場合は全体的なロジック利用率が改善されます。

マルチプレクサのバス

多くの場合、マルチプレクサの入力は、一連のデータ入力バスで同じマルチプレクサ機能が実行されるデータ入力バスです。このようなケースでは、マルチプレクサの非効率性はバスのビット数で乗算されます。上記の項で説明した問題は、ワイド・マルチプレクサ・バスの場合はより重要になります。

例えば、選択ラインを上記の項で詳述した完全なバイナリ case に再コーディングする方法は、多重化バスでよく使用されます。選択ラインの再コーディングは、バスのすべてのマルチプレクサに対して 1 回のみ実行する必要があります。バスのすべてのビットが Recoder ロジックを共有するため、マルチプレクサのバスのロジック効率を大幅に改善できます。

上記の項にある縮退マルチプレクサは、実装に 5 つの LUT を必要とします。入力と出力が 32 ビット幅の場合、ファンクションはバス全体で 32 × 5 または 160 の LUT を必要とします。このレコーダ・デザインは 2 つの LUT を使用しており、全体のバスに対して選択ラインを 1 回のみ再コーディングする必要があります。4:1 バイナリ・マルチプレクサは、バスのビットあたり 2 つの LE を必要とします。再コーディングしたバージョンの合計ロジック利用は、オリジナル・バージョンの 160 LUT と比べた場合、バス全体で 2 + (2 × 32) または 66 の LUT になります。ワイド・マルチプレクサ・バスでは、ロジックの節約がより重要になります。

縮退マルチプレクサを最適化する手法の使用、不要な暗黙のデフォルトの削除、最適な `DEFAULT` または `OTHERS` case の選択は、マルチプレクサのバスを最適化するときに重要な役割を果たす場合があります。

Quartus IIソフトウェアのマルチプレクサ再構築のためのオプション

Quartus II 合成機能には、合成中にマルチプレクサのバスを抽出し、最適化する **Restructure Multiplexers** ロジック・オプションがあります。特定の状況では、このオプションにより、デザインの HDL コードを変更しないで、説明した再コーディング機能の一部が自動的に実行されます。Optimization 手法が **Balanced**（ほとんどのデバイス・ファミリのデフォルト）または **Area** に設定されている場合、デフォルトによってこのオプションはオンになります。

詳しくは、「Quartus II ハンドブック Volume 1」の「Quartus II Integrated Synthesis」の章の「Restructure Multiplexers 」項を参照してください。

CRC（Cyclic Redundancy Check）機能

CRC（Cyclic Redundancy Check）計算は、データの破損を検出するために通信プロトコルおよびストレージ・デバイスで非常に多く使用されます。これらのファンクションは非常に効果的であり、破損したデータが 32 ビット CRC 検査に合格できる確率は非常に低いものです。

CRC 機能は、通常はワイド XOR ゲートを使用してデータを比較します。合成ツールがこれらの XOR ゲートをフラット化および分解して、FPGA LUT にロジックを実装する方法は、デザインの面積および性能の結果に大きな影響を及ぼす可能性があります。XOR ゲートには莫大な数の妥当なファクタリングの組み合わせを作成するキャンセレーション・プロパティがあるため、合成ツールがデフォルトで常に最良の結果を選択できるとは限りません。

Stratix II デバイスの 6 入力 ALUT は、これらのデザインでは 4 入力 LUT よりも非常に有利です。CRC 処理を行うデザインは、Stratix II デバイスにおいて非常に高速で動作させることが可能です。

アルテラ・デバイスでの CRC デザインの結果の品質を向上させるには、以下のガイドラインが役立ちます。

速度最適化による性能向上

合成ツールは XOR ゲートをフラット化し、面積とロジックのレベルの深度を最小化します。Quartus II 合成機能などの合成ツールは、このようなロジック構造に対してはデフォルトで面積の最適化をターゲットにしています。したがって、深度の低減に的を絞るには、合成の最適化手法を速度に設定します。

深度のフラット化によって、面積が大幅に増加する場合があります。

カスケード・ステージの代わりに別の CRC ブロックを使用

一部の設計者は、CRC デザインを最適化し、8 ビットの 4 ステージなどのカスケード・ステージを使用します。このようなデザインでは、データ幅に応じて、また必要に応じて（8、24、または 32 ビット後の計算など）中間計算が使用されます。このデザインは FPGA デバイスでは最適ではありません。CRC デザインで実行できる XOR キャンセレーションは、このファンクションでは最終結果を求めるのに、すべての中間計算が必要ないことを意味します。したがって、中間計算の使用を強制する

と、ファンクションの実装に必要な面積が増加し、またカスケード接続のためにロジック深度も増加します。通常は、デザインで必要なデータ幅ごとに完全に独立した CRC ブロックを作成し、次にそれらをまとめて多重化して、特定のタイミングで適切なモードを選択するのが得策です。

ブロックのマージを許可しないで独立した CRC ブロックを使用

合成ツールは、XOR ロジックでのファクタリング・オプションのために、通常は 2 つの異なる CRC ブロックでリソースを共有し、重複を抽出することによって CRC デザインの最適化を試みる場合がよくあります。前述したとおり、CRC ロジックでは大幅な面積削減が行えますが、各 CRC 機能を個別に最適化したときに有効に機能します。共通のデータ信号でドライブされる、または同じディスティネーション信号を供給する複数の CRC 機能がある場合は、重複抽出をチェックします。

結果の品質に問題があり、2 つの CRC 機能がロジックを共有している場合は、以下の方法のいずれかを使用して、ブロックが個別に合成されるようにしてください。

- 各 CRC ブロックを独立したプロジェクトとして合成し、次に各 CRC ブロックに対する個別の VQM または EDIF ネットリスト・ファイルを書き出します。
  - Quartus II 合成機能を使用して VQM ファイルを作成するには、Processing メニューで **Start** をクリックし、**Start VQM Writer** をクリックします。
- インクリメンタル・コンパイル・デザイン・フローでは、各 CRC ブロックを独立したデザイン・パーティションとして定義します。
  - 詳しくは、Quartus II ハンドブック、Volume 1 の階層ベースおよびチーム・ベースのデザインのための Quartus II インクリメンタル・コンパイルの章を参照してください。
- 合成オプションを使用して、CRC ブロックの階層境界を維持します。
  - Assignments メニューで **Assignment Editor** をクリックします。**Preserve Hierarchical Boundary** を **Firm** に設定します。

可能な場合はレイテンシを使用

デザインで CRC 機能を実装するために複数のサイクルを使用できる場合、レジスタを追加しデザインをリ・タイミングすると、面積の縮小、性能の改善、および消費電力の削減に役立つことがあります。合成ツールにリ・タイミング機能（Quartus II ソフトウェアの **Perform gate-level register retiming** オプションなど）がある場合、入力時に特別なレジス

タ・バンクを挿入し、リ・タイミング機能でレジスタを移動させて結果を向上させることができます。また、1/2 幅の CRC ユニットを構築し、各クロック・サイクルでデータの半分を切り替えることができます。

未使用時の CRC ブロックを無効にして消費電力を削減

CRC デザインは、入力に変化があると常にロジックがトグルするため、ダイナミック消費電力が大幅に増加します。消費電力を節約するには、CRC が必要とされないすべてのクロック・サイクルで、クロック・イネーブルを使用して CRC 機能を無効にしてください。一部のデザインは、他のロジックの実行中には、数クロック・サイクルの間 CRC 結果を検査しません。このような短時間でも、CRC 機能を無効にすることが重要です。

デバイス同期ロード（sload）信号を使用した初期化

多くの CRC デザインでは、演算前にデータを `1` に初期化する必要があります。ターゲット・デバイスが `sload` 信号の使用をサポートしている場合は、演算前にそれを使用して、デザイン内のすべてのレジスタを `1` に設定する必要があります。`sload` 信号の使用を可能にするには、6–33 ページの「クリア & クロック・イネーブルなどのセカンダリ・レジスタ・コントロール信号」の項で説明するコーディング・ガイドラインに従ってください。タイミング・クロージャ・フロアプランまたはチップ・エディタでレジスタの等価性を検査し、信号が予測どおりに使用されていることを確認します。

`sload` 信号を使用して、レジスタの実装を強制する必要がある場合、「Introduction to Low-Level Primitives Design User Guide」の説明に従って、低レベル・デバイスのプリミティブを使用できます。

まとめ

コーディング・スタイルとメガファンクションの実装は、デザインの性能に非常に大きな影響を及ぼす可能性があるため、デザイン・プロセスの当初から、コーディング・スタイルをデバイス・アーキテクチャに合わせることが重要です。デザインの性能および面積利用率を改善するには、この章で説明したコーディングの推奨事項に従って、メモリや DSP ブロックなどの最新デバイス機能、およびターゲットとなるアルテラ・デバイスのロジック・アーキテクチャを活用してください。

その他の最適化の推奨事項については、「Quartus II ハンドブック Volume 2」の「Area & Timing Optimization」の章を参照してください。