WP231 (1.1) 2006 年 1 月 6 日

デザイン パフォーマンス向上のための HDL コーディング法

いかなる FPGA 設計においても、最高のパフォーマンスを得るために欠かせない要因の 1 つが、デザインの適切な RTL 記述です。RTL レベルの設計中に、ちょっとした工夫をすることが、100MHz 未満で動作するデザインか 400MHz 以上で動作するデザインかの明暗を分けることにつながります。

安定したデザイン パフォーマンスとは、デザイン プロセスにおける様々な要因を熟慮した結果にほかなりません。まずは、デザインに最も適したハードウェア プラットフォームを選択しなくてはなりません。そして、選んだデバイス アーキテクチャと、インプリメンテーション ツールの設定や機能を検討する必要があります。さらに、これは本稿の目的でもありますが、ターゲット デバイス上に効果的に配置できる HDL コードを記述しなくてはなりません。これらの各項目の詳細を記載したリソースは、ウェブ サイトに多数掲載されています。本稿では、デザインのパフォーマンスを向上させるコードの記述方法とそのヒントに焦点を当てて紹介します。適切な FPGA コーディング方法を繰り返し述べつつ、あまり知られていないながらも、最新のザイリンクス FPGA アーキテクチャにそのまま応用可能な技術を紹介します。

リセットの使用とパフォーマンス

システム規模のオプションのうち、パフォーマンスやエリア、消費電力に大きな影響をおよぼすオプションがリセットです。システム設計者の中には、グローバル非同期リセットの使用を電源投入時の回路の初期化に限定する設計者もいますが、これは FPGA 設計にとっては不要なことです。ザイリンクスの FPGA アーキテクチャでは、リセットの使用とそのタイプがデザインのパフォーマンスと密接に関係しているため、次のようなリセット ストラテジがあります。

- シフト レジスタ ルックアップ テーブル (SRL) のような、デバイスのライブラリ コンポーネントの使用を抑制する。
- 専用のハードウェア ブロックの同期エレメントの使用を控える。
- ファブリック内のロジックを最適化しない。
- リセット信号のファンアウトは大きくなる場合があるため、配置配線を厳しく制約する。

SRL

すべての最新ザイリンクス FPGA アーキテクチャでは、ルックアップ テーブル (LUT) エレメントをロジック、ROM/RAM あるいは SRL としてコンフィギュレーションできます。合成ツールでは、RTL コードからどの構造が使用されているかが推論できます。ただし、SRL のライブラリ コンポーネントがリセット機能を持たないため、パフォーマンスを最適化したシフト レジスタ (SRL) を使用する場合にリセットがコーディングできません。リセットをコーディングしてシフト レジスタを推論する場合には、数個のフリップフロップ、あるいはロジックを周辺に追加してリセットを機能させる必要があります。図 1 に示すように、シフト レジスタでリセットを使用しないコーディングでは、通常、出力で形成されるレジスタは 1 つなので、エリアおよびパフォーマンスは最適となります。

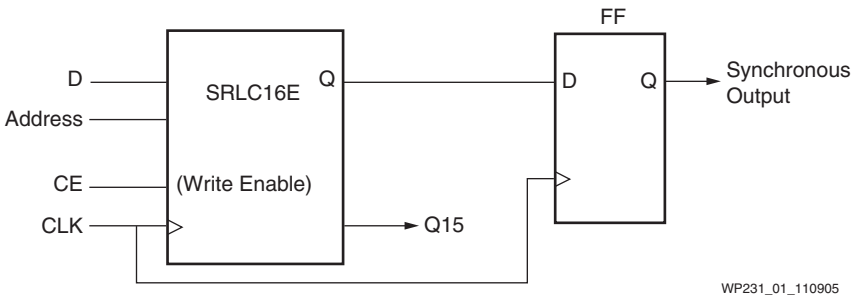

図 1 : パフォーマンスを最適化したシフトレジスタ

リセットを使用しない場合に比べ、使用した場合にエリアと消費電力に及ぼす効果は明らかですが、パフォーマンスに与える影響はそれほど明確ではありません。通常、レジスタ間のタイミング パス (フリップフロップの Clock-to-Out、配線遅延、次のフリップフロップのセットアップ タイム) はデザイン内の最長パスにならないため、フリップフロップで構築したシフト レジスタのパフォーマンスがクリティカルになることはありません。ただし、リソースの追加使用 (フリップフロップおよび配線) がデザインの他の部分の配置配線にマイナスの影響を与え、配線遅延が長くなる場合があります。SRL にさらにロジックを追加してリセット機能を持たせると、追加したロジック部分が SRL の Clock-to-Out に影響を与え、データが目的のロジックに到達するまで時間がかかり、パフォーマンスが低下します。

ヒント

- 最適化した SRL ライブラリ セルのエリアおよびパフォーマンスへの影響を避

け、シフト レジスタのリセットを使用しない。

乗算器と RAM

最新のザイリンクス FPGA アーキテクチャでは、すべてに専用の演算リソースがあります。このリソースは、多数の DSP アルゴリズムなどで乗算に使用できますが、バレル シフタなどのアプリケーションでも使用できます。

同様に、さまざまなサイズの RAM が、ほぼすべての FPGA で、アプリケーションを問わず使用されています。すべてのザイリンクス FPGA には、RAM、ROM、大規模 LUT、さらには汎用ロジックとして使用できるブロック RAM エレメントがあります。乗算器と RAM リソースの両方を利用することで、よりコンパクトで高性能なデザインが実現できます。

パフォーマンスにおいては、選択するリセットのタイプによってデザインに与える影響が違います。乗算器ブロックおよび RAM レジスタは、共に同期リセットだけを有しますので、これらの機能に対して非同期リセットをコーディングすると、ブロック内のレジスタが使用できなくなります。これによって、パフォーマンスに多大な影響を及ぼします。たとえば、最高速の Virtex™-4 をターゲット デバイスとして、完全にパイプライン化された乗算器で非同期リセットを使用すると、パフォーマンスは、約 200MHz に下がります。同期リセットを使うよう再コーディングすると、パフォーマンスは 2 倍以上の 500MHz に向上します。

Virtex-4 のブロック RAM は、乗算器と同様にオプションのレジスタを備えています。これらの出力レジスタを使用すると、RAM のクロックから出力までの時間が短縮し、デザイン全体の処理速度が向上します。オプションのレジスタにはリセット ポートがないため、リセットのビヘイビアをコーディングしても、結果的に出力レジスタはイネーブルされません。

また、RAM を LUT あるいは汎用ロジックとして使用する際に派生する問題があります。時には、エリアとパフォーマンスの理由から、ROM や汎用ロジックとしてコンフィギュレーションした複数の LUT を 1 個のブロック RAM に圧縮することが有効な場合があります。方法としては、これらの構造をマニュアルで指定するか、もしくは、合成ツールに制約を加え、自動的にロジック デザインのその部分を未使用のブロック RAM リソースにマップするかのいずれかです。ブロック RAM のリセット コンフィギュレーションにより、同期リセットを使用する場合や、リセットを全く使用しない場合には、デザイン機能を変更せずに、汎用ロジックが配置できます。

ヒント

- レジスタを専用リソースにパックできなくなり、パフォーマンス、使用率、ツールの最適化に影響を与えるため、非同期リセットの使用は避ける。

汎用ロジック

非同期リセットの影響は、汎用ロジック構造のパフォーマンスにも現れます。FPGA のすべての汎用レジスタがセットおよびリセットを同期または非同期のいずれにでもプログラムできるため、非同期リセットを使用しても問題がないように思えますが、それは得てして誤りです。非同期リセットがどのように最適化を妨げるか、図 2 にコード例を示します。非同期リセットを使用しない場合、信号が必要とするリソースが利用できるため、レジスタを駆動している他の同期パスが最適化できます (図 3 の FDRSE を参照) 。

例 1 :

非同期リセット コードをインプリメントする場合には、ロジック生成に 5 つの信号が使用されるため、合成ツールでデータ パスに対する 2 つの LUT の推論が必要になります。このコードは、図 2 のようにインプリメントできます。

VHDL	Verilog

process (CLK, RST)
begin
  if (RST = '1') then
  Q <= '0';
  elsif (rising_edge(clk)) then
  Q <= A or (B and C and D and E);
  end if;
end process;
	always @(posedge CLK, posedge RST)
	  if (RESET)
	  Q <= 1'b0;
	  else
	  Q <= A | (B & C & D & E);

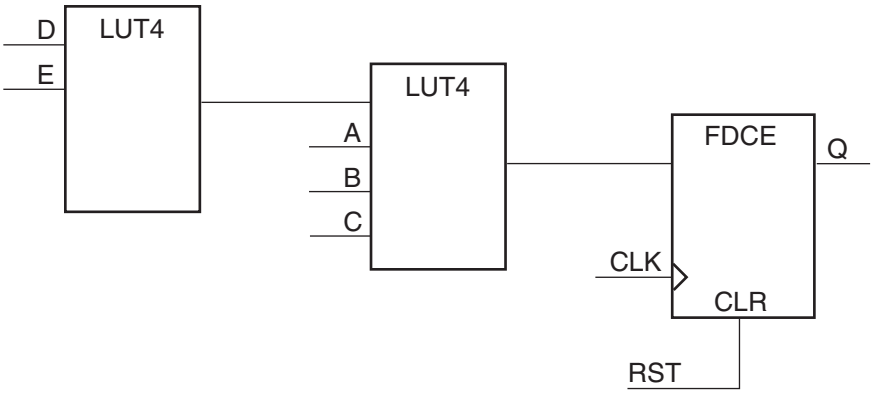

図 2 : 非同期リセットのインプリメント

例 2 :

同期リセットを使用してこのコードを書き換えると、合成ツールでは、機能のインプリメントにより柔軟に対応できます。このコードは、図 3 のようにインプリメントできます。

VHDL	Verilog

process (CLK)
begin
  if (rising_edge(clk)) then
  if (RST = '1') then
  Q <= '0';
  else
  Q <= A or (B and C and D and E);
  end if;
  end if;
end process;
	always @(posedge CLK)
	  if (RESET)
	  Q <= 1'b0;
	  else
	  Q <= A | (B & C & D & E);

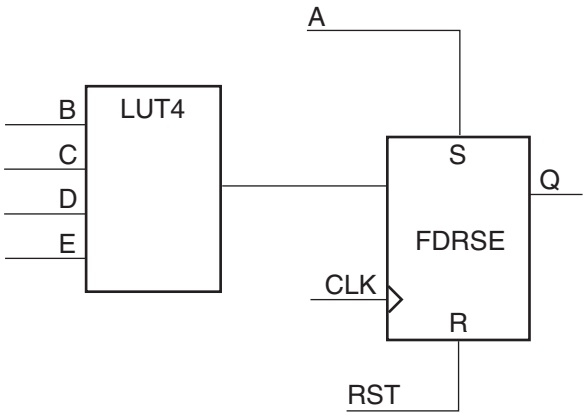

図 3 : 同期リセットのインプリメントでパフォーマンスが改善

図 3 で示したインプリメントでは、合成ツールは A がアクティブ High のとき、Q が常にロジック 1 であると認識します (OR ファンクション)。FDRSE レジスタは同期セットあるいはリセットとしてコンフィギュレーションされているため、セットは同期データ パスの一部として自由に使用できます。

例 3 :

非同期セットあるいはリセットがパフォーマンスに深く関わりがあることを示す、8 つの信号を使用したさらに複雑な機能を見てみます。この機能のインプリメントには、最低でも 3 つの LUT が必要です。このコードは、図 4 のようにインプリメントできます。

VHDL	Verilog

process (CLK, RST)
begin
  if (RST = '1') then
  Q <= '0';
  elsif (rising_edge(clk)) then
  Q <= (F or G or H) and
  (A or (B and C and D and E));
  end if;
end process;
	always @(posedge CLK, posedge RST)
	  if (RESET)
	  Q <= 1'b0;
	  else
	  Q <= (F | G | H) &
	  (A | (B & C & D & E));

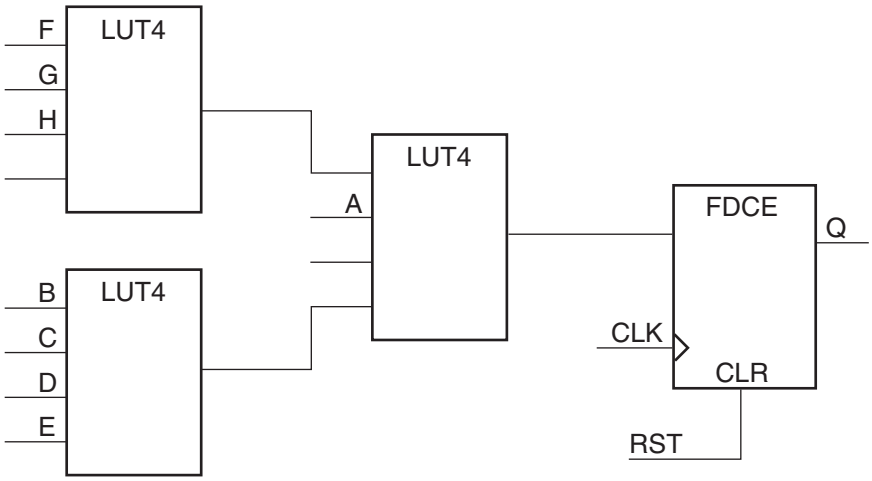

図 4 : 8 つの入力機能を使用した非同期リセットのインプリメント

例 4

このコードは、同期リセットを使用して 図 5 のように書き換え、インプリメントできます。

VHDL	Verilog

process (CLK)
begin
  if (rising_edge(clk)) then
  if (RST = '1') then
  Q <= '0';
  else
  Q <= (F or G or H) and
  (A or (B and C and D and E));
  end if;
  end if;
end process;
	always @(posedge CLK)
	  if (RESET)
	  Q <= 1'b0;
	  else
	  Q <= (F | G | H) &
	  (A | (B & C & D & E));

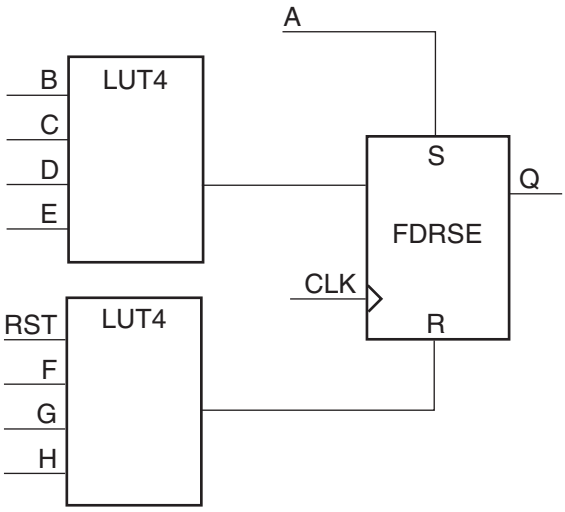

図 5 : 同期制御信号でロジック レベル数が削減できる

図 5 のインプリメント結果が示すように、同じロジック ファンクションのインプリメントに使用する LUT が少なくて済むうえ、このファンクションを生成する各信号でロジック レベルが減少するため、デザインがより高速化されます。デザイン内のロジックの多くは同期なので、同期を使用する場合、あるいはリセットをまったく使用しない場合には、デザインをさらに最適化し、エリアを縮小させ、パフォーマンスが向上できます。

ヒント

- グローバル リセットが必要かどうかを確認する。
- 非同期制御信号の使用を控える。

加算器ツリーではなく加算器チェーンを使用する

多くの信号処理アルゴリズムでは、サンプルの入力ストリームに四則演算を実行後、この四則演算のすべての出力の和を求めます。FPGA のようなパラレル アーキテクチャでは、和をインプリメントするため、加算器には一般的にツリー構造が使われます。加算器ツリーの使用で懸念されるのは、サイズが変化する点です。加算器の数は、加算器ツリーの入力数で決まります。加算器ツリーの入力数が多いほど、必要な加算器の数も増え、ロジック リソースや消費電力も増加します。また、大きなツリーほどツリーの末端に進むにしたがい、加算器の規模が大型化し、ロジック レベ

ル数が増え、システム パフォーマンスの悪化を招きます。デバイスの使用率と消費電力をできるだけ抑え、加算器ツリーのパフォーマンスを維持するには、加算器ツリーを専用のシリコン リソースとしてインプリメントします (図 6 参照)。しかし、シリコン エリアを縮小し、かつ加算器ツリーの大半を専用リソース内にインプリメントするのは、FPGA の製造メーカには不可能です。

Virtex-4 デバイス ファミリは、DSP48 専用シリコンの列を使用するという別の手段で和をインプリメントします。図 7 に示すように、加算器ツリーではなく、チェーン型の加算器を使用して和をインクリメントさせ、算出します。この方法は、従来の FPGA とは一線を画し、ロジックとインターコネクトの両者全体が専用シリコン内に存在することから、DSP アルゴリズムのパフォーマンスの最大化および消費電力の低減につながります。また、パイプライン化すると、DSP48 ブロックのパフォーマンスは、加算器の数にかかわらず、最速のスピードグレードで 500MHz となります。48 ビットの加算器あるいはアキュムレータを組み合わせたポートをカスケード接続すると、現在のサンプルの算出に加え、それまでに算出したすべてのサンプルの和も計算できます。

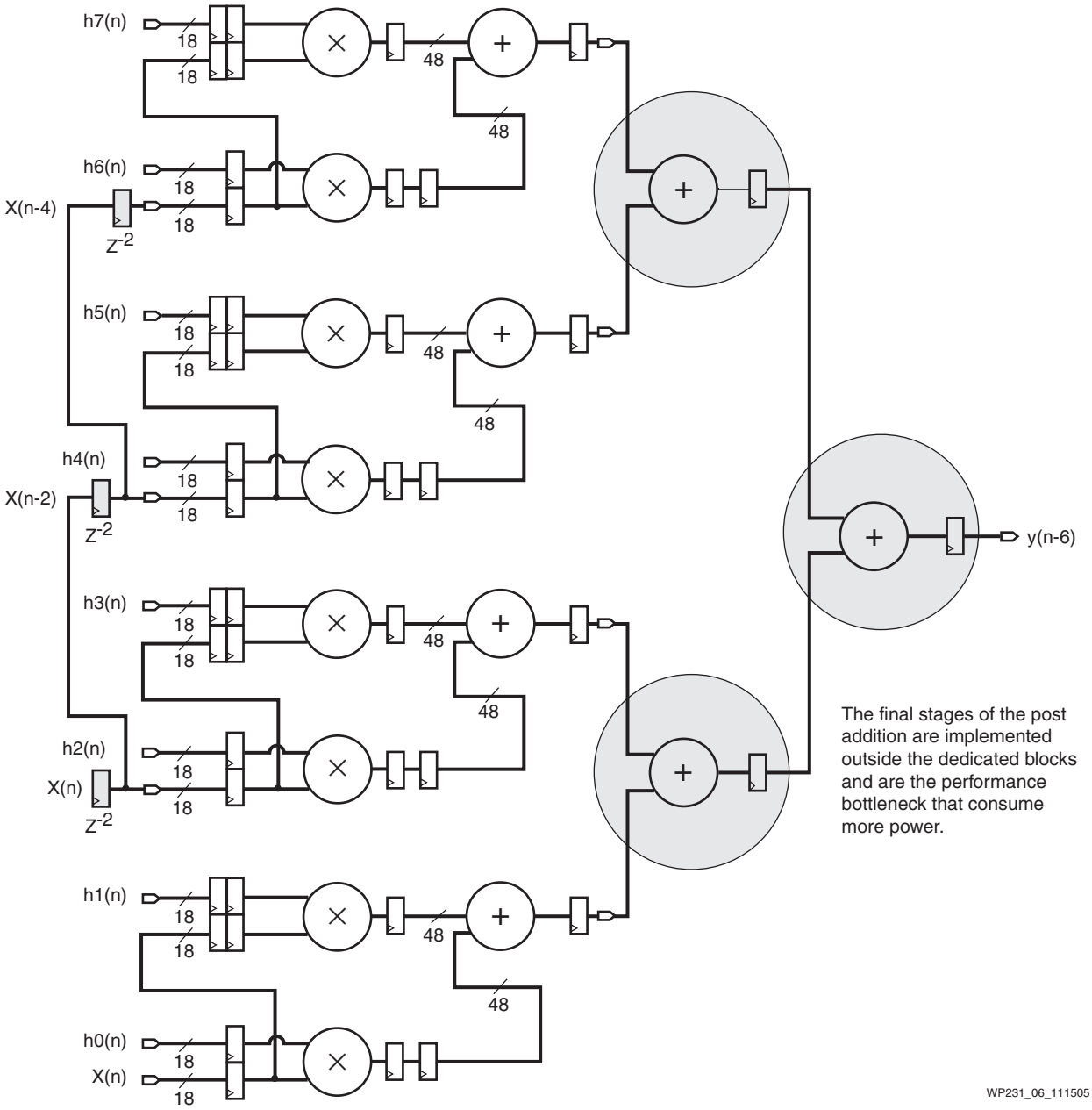

図 6： 加算器のツリー構造がパフォーマンスを低下させ、消費電力を増加させる

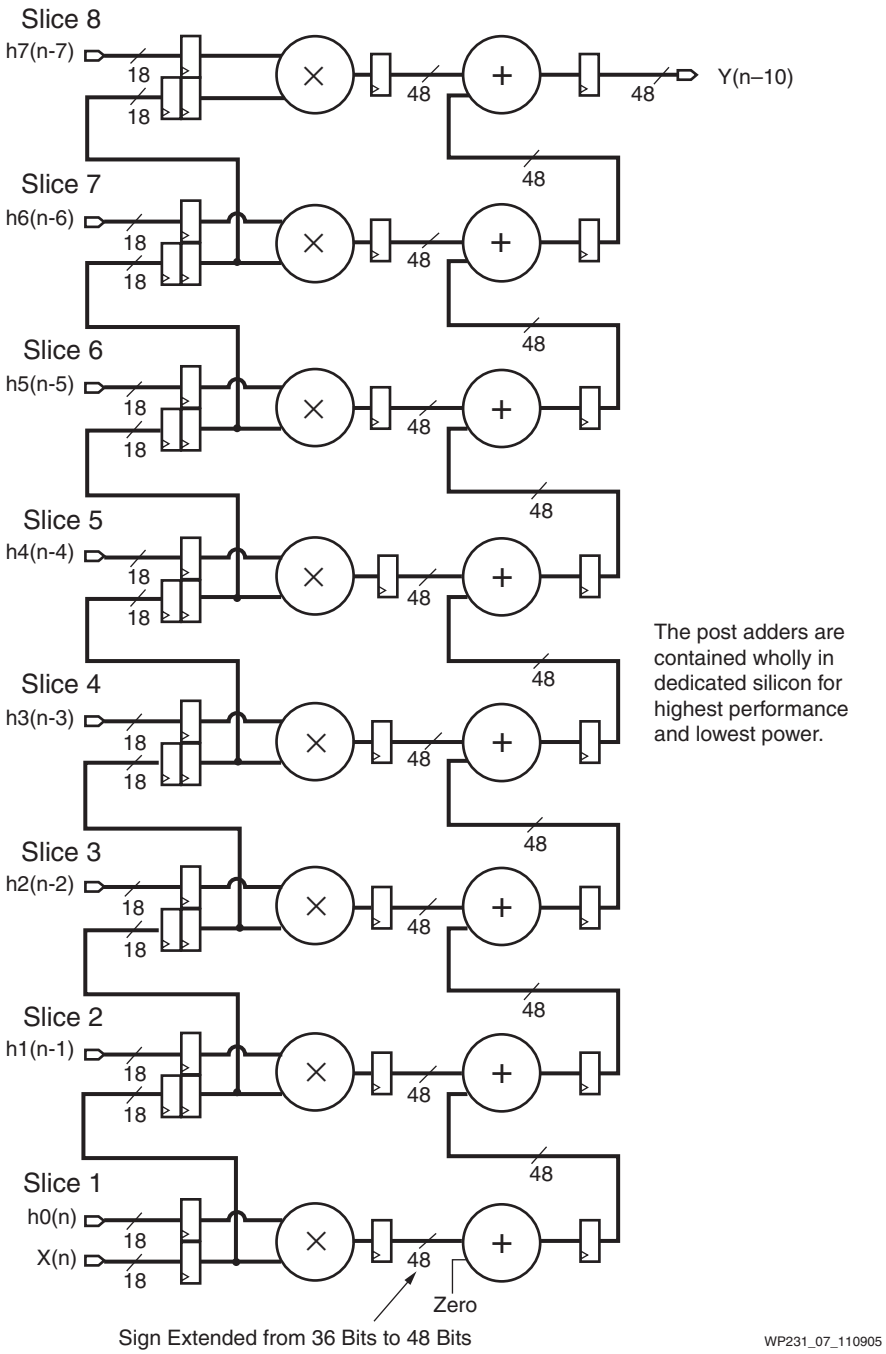

図 7 : チェーン型の加算器でパフォーマンスが予測できる

Virtex-4 の加算器のチェーン構造を利用するには、加算器ツリーの記述を加算器チェーンの記述に置き換えます。ダイレクト フォーム フィルタから転置フィルタへの変換、あるいはシストリック フォーム フィルタへの変換方法は、UG073:『FPGA の XtremeDSP ユーザーガイド』を参照してください。通常、この変換によって、デザインにレイテンシが追加されます。変換終了後、アルゴリズムはアプリケーションの要求する以上に高速で動作します。その場合、マルチチャネリングやフォールディングなどのいずれかの手法を使用して、デバイスの使用率や消費電力をさらに向上できます。いずれの手法も、小型デバイスへのデザインのインプリ

メント時、あるいは空いたリソースを使用して、デザインに機能を追加する場合に有効です。

- マルチチャネリングとは、非常に高速な演算エレメントを利用して、はるかに低いサンプル率で複数入力ストリーム (チャネル) を処理することです。この手法は、シリコン効率をチャネル数とほぼ等倍に高めます。マルチチャネル フィルタリングは、時分割多重化単一チャネル フィルタとして見ることができます。たとえば、一般的なマルチチャネル フィルタリングでは、各チャネルに個別のデジタル フィルタを使用して、複数入力チャネルがフィルタリングされます。Virtex-4 の DSP48 スライスを利用すると、8 倍クロックの単一フィルタでクロッキングできるため、単一デジタル フィルタで 8 つすべての入力チャネルがフィルタリングできます。したがって、FPGA に必要なリソース数が約 1/8 に削減できます。
- フォールディングは同様の概念で動作します。時分割多重化複数入力ストリームを使用せずに、個別フィルタのタップを時分割多重化すると、フィルタが使用するリソースが少なくなります。空いたリソースは別の用途に使用できます。

ヒント

- Virtex-4 の加算器をツリー構造にせず、カスケード接続した記述を使用する。

ブロック RAM のパフォーマンスを最大化

メモリ エレメントを効率的に推論する場合、パフォーマンスに影響を与える次の要因を考慮する必要があります。

- 専用ブロック RAM と分散 RAM のどちらを使用するか。
- 出力パイプライン レジスタを使用する。
- 非同期リセットの使用を控える。

他の要因としては、HDL のコーディング スタイルと合成ツールの設定がメモリのパフォーマンスに大きく影響します。

HDL コーディング スタイル

デュアル ポート ブロック メモリを推論する場合、2 つのポートが同じメモリ セルに同時にアクセスすることが考えられます。たとえば、両ポートが、異なる値を同時に同じメモリ セルに書き込みすると、競合が発生し、メモリ セルの内容が不安定になります。

また、メモリのコンフィギュレーションで懸念される例には、ターゲット デバイスによってメモリの出力値が異なる点があります。最新の Virtex および Spartan™ ファミリは、プログラマブルな 3 種の動作モードによって、書き込み中のメモリ出力が管理できます。デバイスのユーザーガイドに、これらの動作モードの詳細が記載されています。

例 5 :

合成ツールは、表 1 に示したように、コーディング スタイルによってこれらのモードのいずれかを推論できます。

表 1 : ブロック RAM オペレーティング モード インターフェイスの用例

VHDL	Verilog

`-- 'write first' or transparent mode`
`process (clk)`
`begin`
`  if (rising_edge(clk)) then`
`  if (we = '1') then`
`  mem(conv_integer(addr)) <= di ;`
`  do <= di;`
`  else`
`  do <= mem(conv_integer(addr));`
`  end if;`
`  end if;`
`end process;`
	`// 'write first' or transparent mode`
	`always @(posedge clk) begin`
	`  if(we) begin`
	`  do <= data;`
	`  mem[address] <= data;`
	`  end else`
	`  do <= mem[address];`
	`end`

`-- 'read first' or read before write(slower)`
`process (clk)`
`begin`
`  if (rising_edge(clk)) then`
`  if (we = '1') then`
`  mem(conv_integer(addr)) <= di;`
`  end if;`
`  do <= mem(conv_integer(addr));`
`  end if;`
`end process;`
	`// 'read first' or read before write mode(slower)`
	`always @(posedge clk) begin`
	`  if (we)`
	`  mem[address] <= data;`
	`  do <= mem[address];`
	`end`

`-- 'no change' mode`
`process (clk)`
`begin`
`  if (rising_edge(clk)) then`
`  if (we = '1') then`
`  mem(conv_integer(addr)) <= di ;`
`  else`
`  do <= mem(conv_integer(addr));`
`  end if;`
`  end if;`
`end process;`
	`// 'no change' mode`
	`always @(posedge clk)`
	`  if (we)`
	`  mem[address] <= data;`
	`  else`
	`  do <= mem[address];`
	`end`

ヒント

- 「書き込み前に読み出し」 モードを使用せず、ブロック RAM のパフォーマンスの最大化を実現する。

合成ツールの設定

ブロック メモリのパフォーマンスに大きな影響を与える要因で、次に重要なのは合成ツールの設定です。合成ツール (Synplicity 社の Synplify など) によっては、RAM の周囲にバイパス ロジックを挿入し、RTL とハードウェア間で発生するおそれのあるミスマッチを回避します。この余剰ロジックには、読み出しおよび書き込みが同じメモリ セルで実行される場合の既知の値への RAM 出力を強化する目的があります。同じメモリ セルで絶対に読み出しと書き込みが同時に実行されない設計であれば、合成ツールでアプリケーションにバイパス ロジックが追加されないよう設定できます (表 2 を参照)。余剰ロジックには、メモリのすべての出力パス上にオーバーヘッド ロジックを追加するため、メモリ パフォーマンスへのマイナスの影響もあります。追加ロジックを回避、あるいは使用しないことが、メモリ パフォーマンスの維持につながります。

表 2 : 競合を避けるグルー ロジックの挿入をディスエーブルにする

VHDL	Verilog
type mem_type is array (127 downto 0) of std_logic_vector (7 downto 0); signal mem : mem_type; -- disable conflict avoidance logic attribute syn_ramstyle of mem : signal is "no_rw_check";	// disable conflict avoidance logic reg [7:0] mem [127:0] /* synthesis syn_ramstyle=no_rw_check*/;

ヒント

- 合成ツールの設定、テンプレートの推論、メモリ ブロックのパフォーマンスを最大化するための制限事項を確認する。

レジスタの一般的な使用方法

FPGA アーキテクチャでは、各 LUT に 1 つのレジスタがあり、I/O には、メモリ エレメントや DSP エレメントのような専用ブロックや追加レジスタがあります。このようなリソースを使用することが、パフォーマンスの最大化の実現に不可欠となります。レジスタには、デザイン パフォーマンスを向上させる複数の目的があります。これらのレジスタは、クリティカル パス、クリティカル ネットのファンアウト、セットアップ、I/O あるいは専用ブロックからの Clock-to-Out で、ロジック レベル数の削減に使用されます。使用可能なレジスタを最適化するためのガイドラインは後述します。

専用ブロック レジスタの使用

FPGA には、メモリや DSP といった、ほぼすべてのデザインで使用する機能の専用回路があります。これらのブロックにはオプションのレジスタがあります。これらのレジスタをイネーブルにすると、セットアップ、Clock-to-Out とブロック クロック速度のいずれかあるいは両方のブロック パフォーマンスが向上できます。合成ツールは、エリアを最小限に抑え、消費電力を削減し、パフォーマンスを最大に引き上げるため、これらのブロックにレジスタを自動的にパックしようとします。設計者は、合成ツールで制約を与えて、推論したコンポーネントのマッピングを制御できます。これらのブロックをインスタンシエートする際、アプリケーションのパフォーマンスを最大化するレジスタをイネーブルにする必要がありますが、合成ツールのデフォルト設定では、ユーザーがインスタンシエートしたアキテクチャのコンポーネントが最適化されません。最高のパフォーマンスを導くためにすべてのレジスタをイネーブルにすることが理想的ですが、レイテンシの要件を常に満たすとは限りません。その場合は、設計者の責任において適切なレジスタの集合をイネーブルにする必要があります。たとえば、乗算器 (MREG) と入力レジスタ (AREG および BREG) だけが使用される場合、Virtex-4 DSP48 のセル (高速のスピードグレード) は 1.8ns、Clock-to-Output が 2.3ns となります。同じ機能とレイテンシを維持しながら、MREG ではなくアキュムレータの出力レジスタ (PREG) をイネーブルにすると、セットアップ タイムが 3ns に延び、Clock-to-Out は 0.6ns に短縮します。詳細は、図 8 を参照してください。

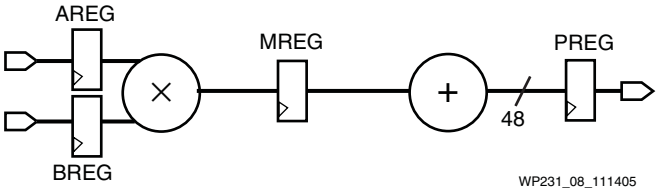

図 8 : DSP48 内部レジスタ

次の項目を考慮することが重要です。

- 駆動するロジックのタイミング、あるいは専用ブロックで駆動されるロジックのタイミング
- HDL コード内のレジスタの位置

ヒント

- 専用ブロックを推論する場合、合成ツールが、パフォーマンスを最良にするレジスタのセットをイネーブルにしない場合に合成制約を使用する (インスタンシエーションについて考慮が必要)。
- インスタンシエートしたコンポーネントにどのレジスタをイネーブルにするか選択する場合、レジスタの最大数が使用されることを確認し、ブロック内のレジスタからレジスタへの遅延、およびレジスタ間の遅延を考慮する。

I/O レジスタを使用する

ザイリンクスのすべての FPGA は、FPGA の入力パスと出力パスに専用のレジスタを備えています。このレジスタを利用して、入力パスのセットアップ タイムおよび出力パスの Clock-to-Out を短縮し、外部デバイスへのデータ供給のタイミング要件を容易に満たすことができます。ただし、場合によっては専用の I/O レジスタを使用することで、FPGA 内のタイミングにマイナスの影響が出る場合があります。内部ロジックへの配線遅延を促すケースです。これらのレジスタは、I/O のタイミングを満たすことが不可欠な場合には I/O 内に配置し、そうでない場合は FPGA ファブリック内に配置すべきです。Synplify のような合成ツールでは、タイミング仕様にしたがって、ファブリック内あるいは I/O 内に自動的にレジスタが配置されます。合成ツールが自動配置をサポートしていない場合、あるいはレジスタを手動で制御しながら配置したい場合は、次の手順を実行してください。

1. 合成ツール (合成ツール資料を参照) のグローバル I/O レジスタ配置オプションをディスエーブルにする。
2. IOB=TRUE を UCF ファイルあるいはソース HDL コードに追加して、レジスタを I/O に配置指定する (制約ガイドの IOB 制約を参照)。
3. ISE Project Navigator で Map オプション [Pack I/O Registers/Latches into IOBs] をディスエーブルにする (あるいは、コマンドラインから -pr スイッチを使用しない)。この設定によって、レジスタは I/O に自動的に配置されません。図 9 を参照してください。

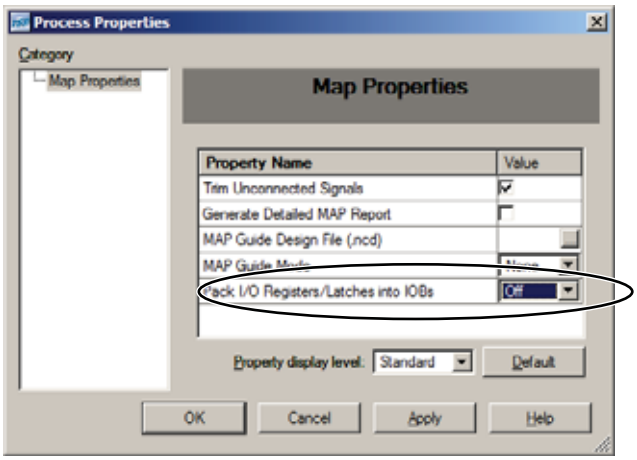

図 9 : レジスタが I/O セルに無差別にパックされないようにする

I/O レジスタの使用を制御することで、FPGA 内部のタイミング仕様を満たす必要のある FPGA への入出力のデータパスのタイミングのバランスを取ります。また、画期的な方法として、FPGA へのすべての入出力ポート上のレジスタが最上位の HDL コードで記述できます。コードの最上位階層でレジスタを指定すると、FPGA のインプリメントに階層デザイン手法を使用した場合の配置の競合が回避できます。また、ボード基板のキャプチャ ツールで使用できない場合があるポートの記述用の階層名を作成する必要もありません。

ヒント

- I/O セルへのレジスタのグローバル パックをディスエーブルにする。その代わり、回路基板上でタイミングがクリティカルなレジスタだけは、FPGA の I/O セルにパックされるよう制約します。

ファンアウトの大きいレジスタを複製する

レジスタの複製手法を使用すると、複製されたレジスタが信号のファンアウトを小さくし、クリティカル パスの動作速度を向上できます。この方法によって、インプリメンテーション ツールで異なるロード信号や関連ロジックの配置配線にゆとりができます。合成ツールでは、この手法が頻繁に用いられます。タイミング レポートで、配線遅延の長い、ファンアウトの小さいネットがクリティカル パスとされた場合には、合成ツールでの複製制約あるいは手動複製レジスタを考慮する必要があります。表 3 に、64 ロード信号を手動で一度に複製する方法を HDL コードで示します。

表 3 : レジスタの複製例

VHDL	Verilog

attribute EQUIVALENT_REGISTER_REMOVAL : string;
attribute EQUIVALENT_REGISTER_REMOVAL of
ce1 : signal is "NO";
attribute EQUIVALENT_REGISTER_REMOVAL of
ce2 : signal is "NO";
 	
begin
 	
-- Clock enable register with 64 fanout
-- replicated once
process (clk) begin
if (rising_edge(clk)) then
ce1 <= ce;
ce2 <= ce;
end if;
end process;
 	
process (clk) begin
if (rising_edge(clk)) then
if (ce1='1') then
res(31 downto 0) <= a_data(31 downto 0);
end if;
if (ce2='1') then
res(63 downto 32) <= a_data(63 downto 32);
end if;
end if;
end process;
	(*EQUIVALENT_REGISTER_REMOVAL="NO"*) reg ce1,
	ce2;
	
	// Clock enable register with 64 fanout
	// replicated once
	always @(posedge clk)
	begin
	ce1 = ce;
	ce2 = ce;
	end
	
	always @(posedge clk)
	begin
	if (ce1)
	res[31:0] <= a_data[31:0];
	if (ce2)
	res[63:32] <= a_data[63:32];
	end

例 6 :

通常、手動で複製したレジスタが合成ツールで自動的に最適化されないよう、合成制約をさらに追加する必要があります。上記の例では、XST 構文 (EQUIVALENT_REGISTER_REMOVAL) が使用されています。

合成ツールの多くは、ファンアウトのしきい値を利用して、レジスタを複製するかどうかを自動的に判別します。しきい値をグローバルに調整することで、ファンアウトの大きいネットを自動的に複製することができますが、どのレジスタを複製すべきかを設計者が制御できるほど高機能ではありません。さらなる措置として、特定のレジスタか階層レベルに属性を与えるか、どのレジスタを複製し、どのレジスタを複製すべきでないかを指定してください。

ヒント

- 配置配線ツールのレポートで、ファンアウトの大きい信号がデザインのパフォーマンスに影響を与えている場合、複製を考慮する。

パイプラインのレベルを追加する

パフォーマンスを向上させる他の手段としては、複数レベルのロジックを持つ長いデータ パスを再構築し、複数クロック サイクルに再分配する方法があります。この手法では、レイテンシおよびパイプライン化されたオーバーヘッド ロジックが増加しますが、データのスループットが向上し、クロック サイクルが高速化します。FPGA には優れたレジスタが豊富にあるので、追加レジスタやオーバーヘッド ロ

ジックがそれほど問題にはなりません。この手法を用いると、データパスは複数クロックに分散されるため、他のデザイン部分においては、追加したパスのレイテンシを反映させたデザインを特に考慮する必要があります。表 4 に、6 レベルのレジスタを 32 X 32 乗算器に加えた場合のコーディング方法を示します。合成ツールでは、最適化された Virtex-4 DSP48 のレジスタとこれらのレジスタをパイプライン化し、データのスループットを最大にします。

表 4 : パイプラインのレベルを追加する

VHDL	Verilog

type regbank6x64 is array(PIPE-1 downto 0) of
std_logic_vector(63 downto 0);
signal prod: regbank6x64;
 	
-- 32x32 multiplier with 4 DSP48 (PIPE=6)
prod(0) <= a * b;
 	
regbank: for i in 1 to PIPE generate begin
process (clk) begin
if (rising_edge(clk)) then
prod(i) <= prod(i-1);
end if;
end process;
end generate;
 	
mult_out <= prod(PIPE);
	parameter PIPE = 6;
	
	reg signed [63:0] prod [PIPE-1:0];
	
	// 32x32 multiplier with 4 DSP48 (PIPE=6)
	always @(posedge clk) begin
	prod[0] <= a * b;
	for (i=1; i<=PIPE-1; i=i+1)
	prod[i] <= prod[i-1];
	end
	assign mult_out = prod[PIPE-1];

可能であれば、デザインの検証方法およびツール セットでリタイミングとパイプラインをまとめて、デザイン パフォーマンスを向上させるよう考慮すべきです。リタイミングとは、組み合わせロジックのレジスタを調整して自動的に移動する合成や配置配線のアルゴリズムで、デザインの主要な入出力において同等のビヘイビアを維持しながら、タイミングを向上させます。

リタイミングによって、コード変更をせずにパフォーマンスを向上させ、RTL デザインを簡略化できます。ただし、リタイミングによってデザイン検証をより複雑化させてしまう場合があります。レジスタ名、その位置、機能が RTL との記述に合致しなくなることが理由です。そのため、リタイミングを使用しない設計者もいます。リタイミングを使用しない場合には、デバイスに関する設計者の知識を活かして RTL でレジスタを記述し、使用可能なデバイス リソースのレジスタを効率的に配置できます。パフォーマンスにおいて、内在するアーキテクチャのロジックのインプリメンテーションは常に考慮すべきです。また、ロジックのレベル数と信号の見込まれるファンアウト数に該当する各コードも考慮すべきでしょう。RTL コードでは、ロジックのレベルやファンアウトに偏りが出ないよう、レジスタの均等配置を選択する必要があります。このガイドラインにそってロジックを配置すれば、可能な限りデザイン パフォーマンスを向上させることができます。

ヒント

- レジスタ間のロジック レベルを調整すると、デザインのパフォーマンスが向上します。RTL コードでパイプラインのレベルを追加するか、合成ツールにリタイミングのオプションを適切に適用するか、あるいはその両方を実行してください。

推論かインスタンシエートかの選択

通常、ビヘイビアからデザインを記述し、FPGA で使用可能なゲートに、合成ツールでコードをマッピングすることが望ましいとされています。コードを移植しやすくするには、推論したロジックすべてを合成ツールで可視し、機能同士の最適化が実行できるようにします。この最適化に含まれるのは、ロジックの複製、結合と再構築、レジスタ間のロジック遅延を調整するためのリタイミングです。デバイスのライブラリ セルをインスタンシエートしても、合成ツールではデフォルトで最適化されません。したがって、デバイスのライブラリ セルの最適化を命令しても、合成ツールでは RTL を使用した同レベルでの最適化は通常実行できません。そのため、合成ツールではセルを介した最適化ではなく、パスへ、もしくはパスからのセルの最適化を実行します。たとえば、SRL をインスタンシエートした結果、この SRL が駆動するロジックのコーンが長い場合には、このパスがボトルネックとなる可能性があります (前述の SRL を参照してください)。SRL の Clock-to-Out は、一般的なレジスタよりも遅延が長くなります。Clock-to-Out 遅延を改善させつつ SRL が使用するエリアの縮小を妨げないようにすると、一般的なフリップフロップで最終的にインプリメントされる SRL の遅延が、実際の遅延より短く生成されることになります。

とはいえ、インスタンシエーションに都合のいい場合もあります。合成ツールでマッピングすると、タイミング、消費電力、エリア制約が満たされない場合や、FPGA 内のある特別な機能が推論できなくなる場合です。インスタンシエーションにおいては、設計者が合成ツール全体を管理します。たとえば、設計者はパフォーマンスを向上させるために、合成ツールが通常実行する LUT とキャリー チェーンのエレメント結合を選択せずに、LUT だけを使用したコンパレータをインプリメントできます。その他の場合、インスタンシエーションだけがデバイスの複雑なリソースを使用可能にする手段となります。理由は次の通りです。

- HDL 言語に限界があります。具体例では、ダブル データ レート (DDR) 出力信号を駆動するのに 2 つの異なるプロセスが必要となるため、VHDL で記述できません。
- ハードウェアが複雑化します。Virtex-4 の I/O の SerDes エレメントをインスタンシエートする方が、ビヘイビアを記述するより容易です。
- 合成ツールでの推論には限度があります。たとえば、現状の合成ツールでは、Virtex-4 の FIFO あるいは DSP48 対称型丸め込みやサチュレートのビヘイビアの記述からの推論ができません。つまり、設計者がインスタンシエートしない限り、それらの回路の使用はできないということです。

ヒント

- デザインのビヘイビアを最大限に記述する。
- ビヘイビアのコードを合成してもタイミングが満たされない場合には、合成ツールのタイミング制約や設定を確認してから、デバイスのライブラリ コンポーネントをインスタンシエーションするコードと交換する。
- 一般的な Verilog や VHDL のビヘイビア構築では、言語テンプレートを考慮して記述する。
- COREGenerator™、Architecture Wizard、ISE の言語テンプレートを考慮して、デバイスのライブラリ コンポーネントをインスタンシエートする。

クロック イネーブルとゲート付きクロックの違い

ザイリンクスでは、CLB レジスタには通常、クロック ポートをゲートするのではなく、専用のクロック イネーブル ポートの使用を推奨しています。ゲート付きクロックはグリッチが発生しやすく、クロック遅延、クロック スキュー、その他の悪影響

を招きます。クロック イネーブルを使用すると、クロック リソースの使用を抑え、タイミング特性、デザイン解析への効果が高まります。デバイスで使用できるクロック イネーブル リソースの使用には、いくつかの方法があります。消費電力の削減としてクロック ドメイン全体のゲートに望ましいのは、BUFGCE というクロック イネーブルのグローバル バッファ リソースを使用する方法です (図 10 を参照してください)。

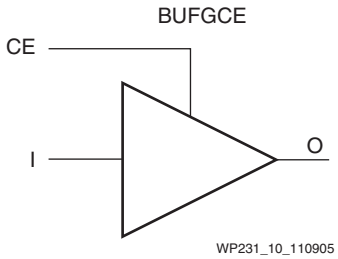

図 10 : BUFGCE

デザインの特定の狭いエリアで、クロックを数サイクル間停止させることのみを目的としたアプリケーションに適しているのは、FPGA レジスタのクロック イネーブルピンを使用する方法です。まず、図 11 に示してあるのが、クロック信号の不適切なゲート方法の例です。次に、図 12 に示すのが、コードを書き直して、クロック イネーブルピンを効果的にマップしたものです。

VHDL	Verilog

GATECLK <= (IN1 and IN2 and CLK);
process (GATECLK)
begin
  if (rising_edge(GATECLK)) then
  if (LOAD = '1') then
  OUT1 <= DATA;
  end if;
  end if;
end process;
	assign GATECLK = (IN1 & IN2 & CLK);
	always @(posedge GATECLK)
	  begin
	  if (LOAD)
	  OUT1 <= DATA;
	end

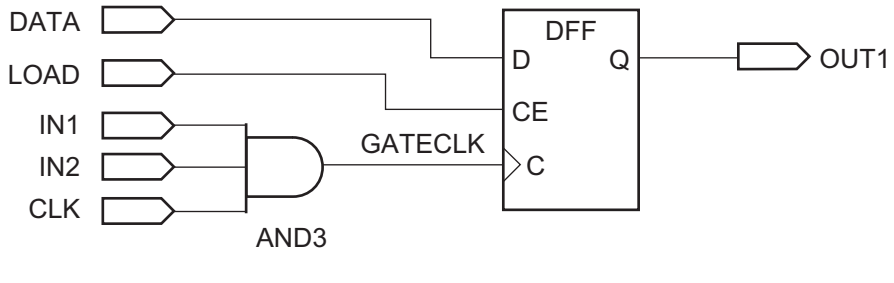

図 11 : ゲート付きクロック - ザイリンクスで推奨しないコーディング方法

VHDL	Verilog

ENABLE <= IN1 and IN2 and LOAD;
process
begin
  if (rising_edge(CLOCK)) then
  if (ENABLE = '1') then
  DOUT <= DATA;
  end if;
  end if;
end process;
	assign ENABLE = (IN1 & IN2 & LOAD);
	always @(posedge CLOCK)
	begin
	  if (ENABLE)
	  DOUT <= DATA;
	end

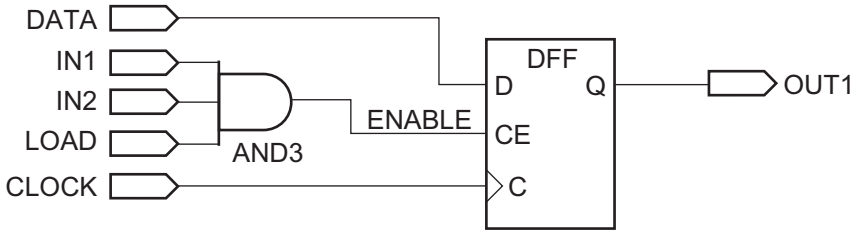

図 12： クロック イネーブル - クロック信号のゲートに効果的な手法

ヒント

- ゲート付きクロックを使用しない。
- グローバル クロック バッファのクロック イネーブル ポートを使用して、クロック ドメイン全体のクロックを停止する。
- ローカルにクロックをディスエーブルにする場合は、レジスタのクロック イネーブル ポートを使用する。
- クロック イネーブル信号の複製は、パスの一部がタイミング要件を満たさない場合に考慮する。

ネストした If-Then-Else、Case 構文および組み合わせ For ループ

if 構文および case 構文のネストや他の構文中に構文として含めることは避け、コード中に含むネット数を最小限に抑えてください。if 構文中に if 構文を入れすぎると、行が長くなり、構文の最適化に支障をきたします。構文のネストを最小限にすることで、一般的にはコードが読みやすく、移植しやすくなり、プリントする場合にもフォーマットが容易になります。

HDL で for ループを記述する場合、特に演算ロジック、特定のオペレーションのロジックに対して望ましいのは、データパスに最低 1 つのレジスタを配置することです。すると、コンパイル中に、合成ツールではループが展開されません。これらの同期エレメントがないと、合成ツールは各ループの反復で生成されたロジックを連結するので、組み合わせパスが非常に長くなり、結果としてデザイン パフォーマンスに影響をきたします。

ヒント

- シーケンシャルな構文のネスト数を最小限にする。
- 長い組み合わせパスを作成する場合には、for ループにレジスタを追加する。

階層

デザインの階層分離の選択では、HDL コード記述の容易性に左右されるケースがたびたび発生します。ただし、デザイン過程に費やす時間を短縮しつつ、パフォーマンスの観点からデザイン階層全体を最高の状態にするには、デザインの最適化、インプリメント方法、検証を熟考することが、より重要となる場合もあります。最適化という点から見ると、合成ツールの多くは論理的なデザイン階層を「ソフト」として扱い、それは、階層をできるだけ維持すること、最適化に影響が出ない範囲での階層を分離すること、または階層下のフットプリントおよびロジック内容を変更することを意味します。インクリメンタル デザインや階層維持を検証対象とするような階層的デザイン手法を用いると、論理的境界に跨る最適化は実行できません。ガイドラインに適切に従わない場合には、ロジック レベルあるいは配置制約が多くなり、デザインの最適化に影響を与えます。非階層デザインのインプリメンテーション フローを使用したとしても、ガイドラインに従うことで、より容易に実行ができ、配置配線ツールでも、ロジックの最適化、および配置にとって最良の選択がなされます。次にガイドラインを説明します。

1. 階層内のすべての入出力にレジスタを付けてください。それが難しい場合には、出力だけでも必ずレジスタを付けてください。一般的なロジックの最適化において、階層の影響を受けなくなります。
2. インスタンシエートした I/O バッファ、レジスタ、DDR 回路、SerDes、遅延エレメントを含むすべての I/O コンポーネントをデザインの最上位に配置してください。最上位にすべてを配置できない場合には、1 階層にすべてを収めるようにしてください。
3. FPGA の共有機能やリソースに配置する必要のあるレジスタまたはロジックのセットは、同位層に収めてください。たとえば、Virtex-4 の DSP48 に乗算器、アキュムレータ、関連レジスタを配置する必要のあるデザインでは、これらすべてのエレメントを同層のモジュールに収めなければなりません。
4. リソースを共有している合成ツールのロジックは、同位に配置してください。
5. 階層境界でファンアウトの大きいレジスタは、手動で複製してください。

以上の簡単なガイドラインを遵守するならば、選択した階層がデザインの最適化や、デザインのパフォーマンスに影響を及ぼすことはありません。この規定に従えない場合には、インプリメントしたデザインのクリティカル パスを確認して、階層に変更を加えることが最終的なデザインのパフォーマンスに影響があるかを判断するようお勧めします。

まとめ

合成および配置配線アルゴリズムにおける昨今の進歩がもたらした影響で、特定のデバイスから最大のパフォーマンスを引き出すことが、より容易になりました。合成ツールでは、複雑な演算やメモリ記述を推論し、専用のハードウェア ブロックにマッピングできますし、リタイミングのような最適化や、ロジックやレジスタの複製もできます。また、配置配線の輻輳を緩和するため、タイミング制約に基づいて、配置配線ツールでネットリストを再構築し、タイミング ドリブン パックや配置ができるようになりました。とはいえ、特定の RTL の記述では、ツールが最大化できるパフォーマンスは限られています。デザインのパフォーマンスをさらに改善する必要のある場合には、ターゲット デバイス、ツールの制約やオプションにより一層精通し、本稿で説明したコーディングのガイドラインに従うのが非常に効率的な方法といえます。

その他のリソース

デザイン パフォーマンスを最大化させるための、さらなるヒントが記載されたリソースを紹介します。

『合成シミュレーション デザイン ガイド』
http://www.xilinx.co.jp/support/software_manuals.htm

ザイリンクス TechXclusives
http://www.xilinx.co.jp/xlnx/xweb/xil_tx_home.jsp

『XST ガイド』
http://www.xilinx.co.jp/support/software_manuals.htm

Sunburst Design 社の Verilog コーディング スタイルおよび資料
http://www.sunburst-design.com/papers/

『ステートマシン設計を最適化する StateCAD XE』
http://www.xilinx.co.jp/xcell/xl38/xcell38_24.pdf

改訂履歴

次に、この資料の改訂履歴を示します。

日付	リビジョン	改訂内容
2005/12/05	1.0	初版リリース
2006/01/06	1.1	誤字の修正。