第3回

設計ガイドラインの裏を知る

小林 優

今回は「設計ガイドラインの裏側」を解説する。業界標準や社内標準のガイドラインであっても、その本質的な意味を考えずに盲目的に従うと、ぐあいの悪い場合がある。あらゆる局面で完璧に適用できるガイドラインは存在しない。ここでは STARC (半導体理工学研究センター) と エッチ・ディー・ラボ がまとめた『設計スタイルガイド』を例に、HDL の設計ガイドラインの利用上の注意点について考える。 (編集部)

どんな分野でも、主義主張には論じられている背景に見逃せない重要な問題が隠されています。設計におけるガイドラインについても同じことです。そこで今回は、設計ガイドラインの裏側を解説します

筆者の会社(エッチ・ディー・ラボ)では、設計資産をスムーズに活用するための HDL 設計ガイドラインである『設計スタイルガイド』を発行しています。多くの企業に導入していただいているので、読者のみなさんも目にしたことがあるかもしれません。

このガイドに従えば、いわゆる「IP の再利用」が容易になります。しかし個々の項目について、なぜそう規定しているのか、従わなかったらどうなるのか、といったことを考えている方は少ないと思います。与えられたガイドラインをうのみにして、盲目的に従うという姿勢は、ちょっといただけません。

世の中に「絶対」というものがないように、ガイドラインといえども、あらゆる局面で完璧とはいかないのです。ここでは、いくつかの事例を紹介します。ガイドラインの根拠や背景について、設計者のみなさんが自身で考えてみてください。

● always の組み合わせ回路でブロッキング代入を使う!?

always 文の中の reg 信号への代入には、二つの代入記号が使えてしまいます。そこでガイドラインの中では、

● 組み合わせ回路 …… ブロッキング代入(=)

● 順序回路 …… ノンブロッキング代入(<=)

を推奨しています。実にやっかいです。裏を探ってみましょう。

順序回路では、記述の順序に動作が影響されないようにするため、ノンブロッキング代入(<=)を使います(本誌 2001 年 9 月号, pp.105-111 の本連載第 1 回を参照)。

同じように、組み合わせ回路でノンブロッキング代入(<=)を使うと、次のような問題を生じることがあります。リスト 1 (a) は、ラッチを生成してしまう記述です。case 文の中で default を使っている、すべての条件で出力信号の値が決まっていなくて、値を保持する回路、つまりラッチを生成してしまいます。

これを回避するために、分岐前に何らかの値を設定しておけば、ラッチは生成されません(リスト 1 (b))。しかしこの記述では、あるシミュレータで正しく動作しないことがありました。出力 P、Q への代入が 2 回発生しますが、記述の順に代入されていないことがあったのです。そこで、ガイドラインでは確実に動作させるために、順次処理であるブロッキング代入(=)を使うことを推奨しています(リスト 1 (c))。

が、しかしです。そもそも同一信号に対して 2 回の代入があることがマズイのであって、どんな条件下でも 1 回しか代入がなければ、順番の問題は発生しません。つまりリスト 1 (d) のように記述すれば、組み合わせ回路、順序回路の区別なくノンブロッキング代入(<=)だけでよくなります。こちらのほうがずっとシンプルなルールかもしれません。

〔リスト1〕 always文による組み合わせ回路における代入記号

同じ目的であっても手段はいろいろある。できるだけ単純なほうが望ましい。このルールにおいても、always文における代入記号をノンブロッキング代入(<=)に統一したうえで、「同一信号には、どんな条件下でも代入が1回しか生じないように記述する」としたほうが単純。

```
wire [1:0] D;
reg      P, Q;

always @( D ) begin
  case ( D )
    2'b00:  begin P<=1'b0; Q<=1'b1; end
    2'b01:  begin P<=1'b1;          end
    2'b10:  begin          Q<=1'b1; end
    default: begin P<=1'b0; Q<=1'b0; end
  endcase
end
```

(a) ラッチを生成してしまう記述

```
wire [1:0] D;
reg      P, Q;

always @( D ) begin
  P<=1'bx; Q<=1'bx;
  case ( D )
    2'b00:  begin P<=1'b0; Q<=1'b1; end
    2'b01:  begin P<=1'b1;          end
    2'b10:  begin          Q<=1'b1; end
    default: begin P<=1'b0; Q<=1'b0; end
  endcase
end
```

(b) 初期値代入による対策

```
wire [1:0] D;
reg      P, Q;

always @( D ) begin
  P=1'bx; Q=1'bx;
  case ( D )
    2'b00:  begin P=1'b0; Q=1'b1; end
    2'b01:  begin P=1'b1;          end
    2'b10:  begin          Q=1'b1; end
    default: begin P=1'b0; Q=1'b0; end
  endcase
end
```

(c) 初期値代入はブロッキング代入が確実

```
wire [1:0] D;
reg      P, Q;

always @( D ) begin
  case ( D )
    2'b00:  begin P<=1'b0; Q<=1'b1; end
    2'b01:  begin P<=1'b1; Q<=1'bx; end
    2'b10:  begin P<=1'bx; Q<=1'b1; end
    default: begin P<=1'b0; Q<=1'b0; end
  endcase
end
```

(d) どんな条件が成立しても代入を1ヵ所にすればOK

〔リスト2〕 内部参照のある記述

always文にいろいろ詰め込むのは得策ではない。内部で代入した信号を参照する記述は要注意である。入力と出力の関係を整理し、always文を分けたほうが論理合成上も有利。

```
wire [7:0] A, B;
reg [7:0] SUM;
reg      ZERO;

always @( A or B ) begin
  SUM <= A + B;
  if ( SUM==8'h00 )
    ZERO <= 1'b1;
  else
    ZERO <= 1'b0;
end
```

(a) 出力をalways内で参照した記述

```
wire [7:0] A, B;
reg [7:0] SUM;
reg      ZERO;

always @( A or B ) begin
  SUM = A + B;
  if ( SUM==8'h00 )
    ZERO = 1'b1;
  else
    ZERO = 1'b0;
end
```

(b) 確かにブロッキング代入なら動作するが…

```
wire [7:0] A, B;
reg [7:0] SUM;
reg      ZERO;

always @( A or B ) begin
  SUM <= A + B;
end

always @( SUM ) begin
  if ( SUM==8'h00 )
    ZERO <= 1'b1;
  else
    ZERO <= 1'b0;
end
```

(c) 正しくはalwaysを分ける

また、リスト2(a)のように内部参照のある記述では、正しいシミュレーション結果にならない(ZEROの変化が、次の入力の変化まで遅れる)ことがあります。確かにブロッキング代入を使えば正しく動作しますが(リスト2(b))、そもそもalways文の中で一つの信号に対して代入と参

照を行う記述は避けるべきです。リスト2(c)のようにalways文を分けるべきでしょう。

設計経験の少ない設計者の中には、組み合わせ回路と順序回路の違いをきちんと理解していない方もいます。筆者が講師を務めたセミナーでも、このような方が多数お