触って学ぼう FPGA 開発入門（1）
理論より実践！ FPGA 開発をスタートしよう

鳥海 佳孝　設計アナリスト　2007/1/25

本連載は、「これから FPGA を開発してみよう！」という入門者の方や仕事で FPGA の設計をされている方（特に新人の方）を対象にしています。また「理論より実践」を主眼とし、入手しやすい無償開発ツールと低価格の FPGA ボードを題材に解説していきます。実際にツールとトレーニングボードを動かして楽しみながら学んでいきましょう。（編集部）

開発環境の準備

ツールを選定しよう

まずは、FPGA 開発に必要なツールを選定します。FPGA メーカー各社から独自の開発ツールが提供されています。その中から筆者が選択したのは、ザイリンクスの「ISE WebPACK 8.2i」です。同ツールはアカウント登録さえ行えば、無償で利用できます。

ISE WebPACKは、ザイリンクスのサイトにある、「無償ダウンロード」からダウンロードできます。インストールの方法は、「WebInstall」と「シングル ファイル ダウンロード」の 2 種類があります。どちらでインストールしても構いません。また、インストールはデフォルトの設定で進めて特に問題ありません。

ちなみにサイト内の「無償 ISE WebPACK の登録およびダウンロード方法」に、ダウンロードに関する説明があるので参考にしてください。

参考：

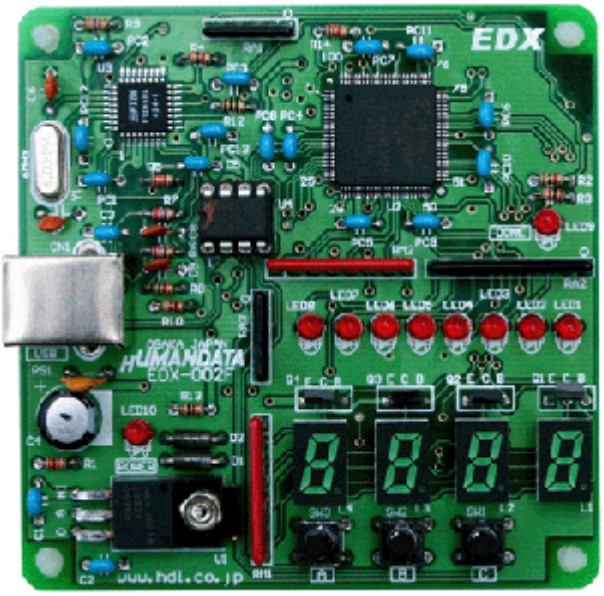
拙著【実践】C言語による組込みプログラミングスタートブックの「Xilinx社の開発環境（ISE WebPACK）のインストール」に、ツールのインストール方法が記載されています。

関連リンク：

- アルテラhttp://www.altera.co.jp/
- ザイリンクスhttp://japan.xilinx.com/

ボードを選定しよう

続いて、今回使用するFPGAボードを選定します。本連載は入門者を対象にしているので、誰にでも気軽に始められるような「低価格」で「使いやすい」ボードを選択したいと思います。世の中には多くのFPGAボードが出回っていますが、その中で低価格と使いやすさを考慮すると「ヒューマンデータ XILINX対応FPGAトレーナ EDX-002」が最良だと筆者は考えます。同製品を選択した主な理由は以下のとおりです。



- **価格が 1 万 4800 円（税抜き）と安い**
- **USB で設計データを FPGA ボードへダウンロードできる（パラレルポートのないノート PC でも利用可能）**
- **USB から電源供給できる**
- **入出力が豊富にある（プッシュスイッチ×3、赤色 LED×8、7 セグメント LED×4）**

写真 1　ヒューマンデータ XILINX 対応 FPGA トレーナ EDX-002

EDX-002 と PC を接続するには、EDX-002 用のドライバを PC にインストールする必要があります。また、開発・生成した FPGA 用のデータを EDX-002 にダウンロードするには「BitCfg」というツール

が必要となります。詳しくは、EDX-002 に付属しているマニュアルを参考にインストールしてください。

これで、開発ツール（ISE WebPACK）と実装対象となる FPGA ボード（EDX-002）の準備ができました。

関連リンク：
ヒューマンデータ　http://www.hdl.co.jp/

Verilog-HDL を記述しよう

開発を進める前に、回路設計に使用するハードウェア記述言語（HDL）を選定する必要があります。HDL には大きく分けて「Verilog-HDL」と「VHDL」がありますが、本連載では以下の理由から Verilog-HDL を採用することにします。

- **Verilog-HDL の方が歴史が古い（日本での浸透度はアプリケーション分野により異なる）**
- **記述量が少ない**
- **Verilog-HDL が主流である（筆者の独断と偏見）**

ここでは、Verilog-HDL の文法や言語仕様などの細かな説明は避けて、まずは「動かす」ことを第一に進めていきます。

今回は、3 つのプッシュスイッチ（以下スイッチ）を押して（入力）、8 つの赤色 LED を点灯（出力）させる**「3 入力 8 出力のデコーダ」**をサンプルとして開発してみましょう。

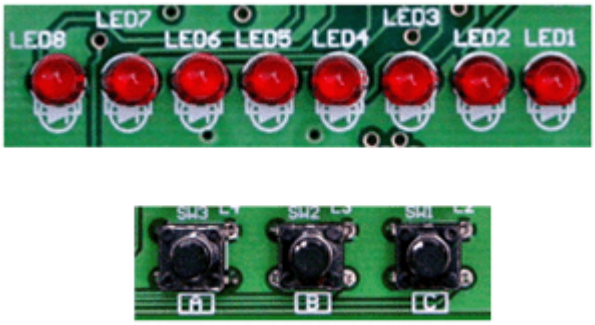
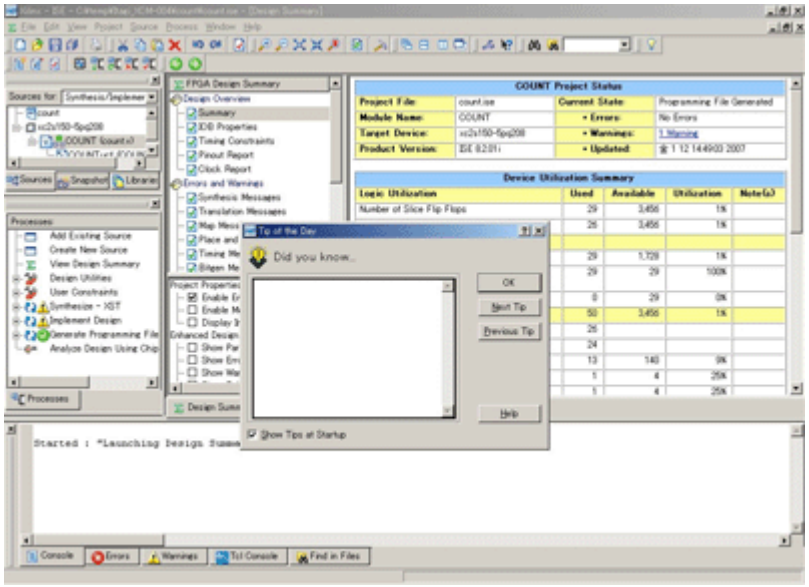


写真 2　8 つの赤色 LED（上）と 3 つのプッシュスイッチ（下）

下記リスト 1 に「3 入力 8 出力のデコーダ」のソースを示します。このソースをそのまま入力して「dec.v」というファイルを作成し、任意のフォルダに保存します（本連載では、C:¥temp¥IT_Media¥Chapter1 に保存しています）。

```
1  module DECODER(A,B,C,Y);
2  input A,B,C;
3  output [7:0] Y;
4  reg [7:0] Y;
5
6  always @(A or B or C)
7  begin
8     case({A,B,C})
9              3'b000:Y=8'b00000001;
10             3'b001:Y=8'b00000010;
11             3'b010:Y=8'b00000100;
12             3'b011:Y=8'b00001000;
13             3'b100:Y=8'b00010000;
14             3'b101:Y=8'b00100000;
15             3'b110:Y=8'b01000000;
16             3'b111:Y=8'b10000000;
17             default:Y=8'bxxxxxxxx;
18    endcase
19 end
20 endmodule
```

リスト 1　「3 入力 8 出力のデコーダ」の Verilog-HDL ソース（dec.v）



画面 1　ISE WebPACK（Xilinx ISE 8.2i－Project Navigator）

ISE WebPACK での開発

FPGA データを生成しよう

「3 入力 8 出力のデコーダ」のソースを作成したら、ISE WebPACK を使用して**「論理合成」「FPGA のピン固定」「配置配線」「FPGA のデータ生成」**を行います。

ISE WebPACK（Xilinx ISE 8.2i－Project Navigator）を起動します。最初に「Tip of the Day」というダイアログが表示されるので［OK］ボタンを押します。

次にプロジェクトの新規作成を行います。メニューから［File］－［New Project］を選択すると［New Project Wizard - Create New Project］ダイアログが表示されます。［Project Location］に c:¥temp¥IT_Media を指定し、［Project Name:］に Chapter1 と入力して［Next］ボタンを押します。

続いて、デバイスを指定します。下記を参考に選択してください。デバイスの指定が完了したら［Next］ボタンを押します。

Property Name	Value
Family	Spartan2
Device	XC2S15
Package	VQ100
Speed	-5

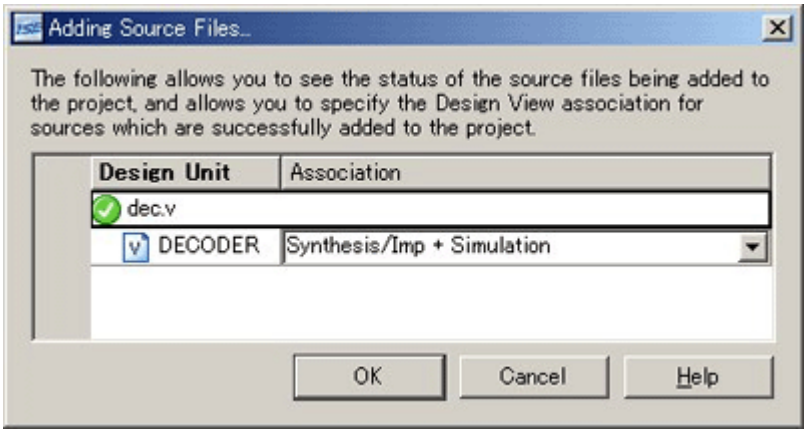
新しいソースファイルを作成するダイアログが表示されますが、すでにサンプルソース dec.v を作成済みなので、そのまま［Next］ボタンを押します。

次に既存のソースファイルを指定するダイアログが表示されるので［Add Source］ボタンを押し、dec.v を指定して［Next］ボタンを押します。サマリが表示されたら［Finish］ボタンで完了します。

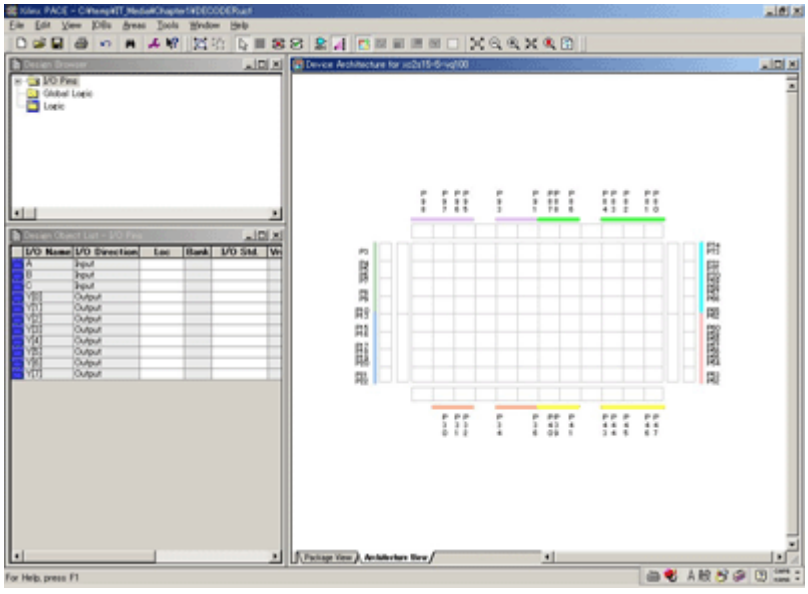
最後にソースの追加を確認するダイアログが表示されます。ソースが正常に追加されると dec.v の横に緑のチェックマークが表示されます（**注**）。［OK］ボタンを押して新規プロジェクトの作成を完了します。

注：チェックマークが緑の場合は正常です。エラーの場合、オレンジの「？」マークが表示されるのでソースファイルなどを修正する必要があります（以下同）。

ISE WebPACK の画面に戻ると、dec.v が Chapter1 というプロジェクトに追加されています。画面左の［Process］ツリーにある「Synthesize - XST」を右クリックして［Run］で「論理合成」を実行します。論理合成が正常に終了すると、緑のチェックマークが表示されます。次に「FPGA のピン固定」ツール（Xilinx PACE）を起動します。画面左の［Process］ツリーにある「User Constarints」の「+」をクリックして展開します。「Assign Package Pins」を右クリックして［Run］を選択すると、メッセージが表示されるので［Yes］ボタンを押して FPGA のピン固定ツールを起動します。



画面 2　ソースの追加を確認するダイアログ



画面 3　「FPGA のピン固定」ツール（Xilinx PACE）

画面左の［Design Object List - I/O Pins］にある［Loc］という項目を下記のように入力します。

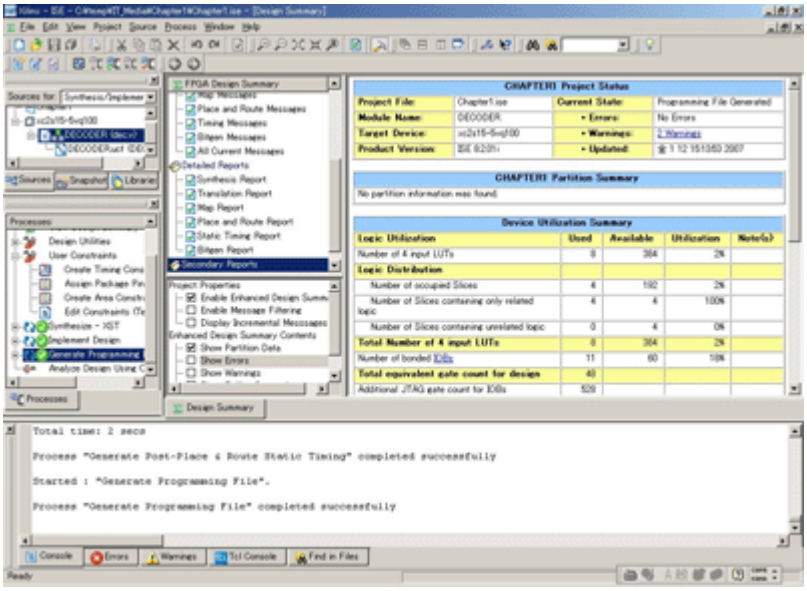
I/O Name	Loc
A	P17
B	P16
C	P15
Y<0>	P68
Y<1>	P67
Y<2>	P66
Y<3>	P65
Y<4>	P56
Y<5>	P55
Y<6>	P54
Y<7>	P53

設定が完了したら、メニューの［File］－［Save］でピン固定の情報をファイル（DECODER.ucf）に保存します（**注**）。

注：保存後、［Bus Delimiter］ダイアログが表示された場合は［Select IO Bus Delimiter］の「XST Default: <>」にチェックを入れて、［OK］を選択します。

ISE WebPACK に戻り、画面左の［Process］ツリーにある「User Constraints」の「+」をクリックして展開します。「Edit Constraints (Text)」を右クリックして［Run］を選択すると、画面に DECODER.ucf の中身が表示されます。

最後に「配置配線」「FPGA のデータ生成」を行います。画面左の［Sources］ツリーから「DECODER（dec.v）」を選択し、同じく画面左の［Process］ツリーにある「Generate Programming File」を右クリックして［Run］を実行します。正常に終了すると、緑のチェックマークが表示されます。

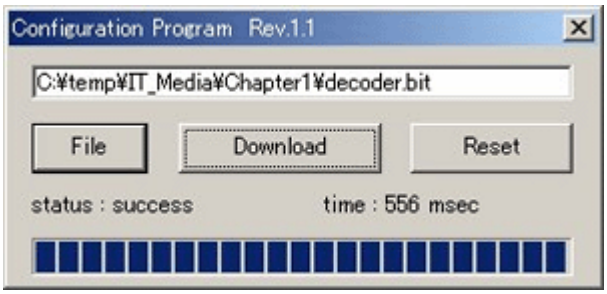


画面 4　配置配線・FPGA データの生成が正常終了した様子（画像をクリックすると拡大表示します）

以上で「論理合成」「FPGA のピン固定」「配置配線」「FPGA のデータ生成」の作業は完了です。

ボードにダウンロードして動作確認しよう

作成した FPGA 用のデータを早速 EDX-002 ボードにダウンロードしてみましょう。まず、BitCfg（EDX-002 ボードに付属）を起動します。メニューの［File］を選択し、C:¥Temp¥IT_Media¥Chapter1¥decoder.bit を指定します。次に［Download］ボタンをクリックします。ダウンロードが正常に完了すると、ダイアログに「status : success」と表示されます。



画面 5　ダウンロードが完了した様子

以上で EDX-002 ボードへのダウンロードは完了です。次は、いよいよ動作の確認です。それでは、ボードにあるスイッチを押してみましょう。

ボタンが何も押されていないときは一番左端の赤色 LED が消灯して、すべてのスイッチを押すと右端の赤色 LED が消灯します。いかがでしょうか？ わずか 20 行ほどのプログラムで開発できました。筆者が回路設計を始めた 20 年前に比べると、簡単に開発が行えるようになったとあらためて実感します。

プログラムの修正と反映

ソースを変更してみよう

次は、リスト 1 のプログラムを少し修正してみましょう。

変更後の仕様は、**「スイッチが何も押されていない（000）ときは右端の赤色 LED が点灯し、スイッチがすべて押されている（111）ときは左端の赤色 LED が点灯する」**です。

ソースを修正する前に、リスト 1 の動作のポイントをおさらいしておきましょう。

- **FPGA ボード上のスイッチ入力が負論理（「0」でスイッチが ON）になっている**
- **FPGA ボード上の赤色 LED 出力が負論理（「0」を出力すると赤色 LED が点灯）になっている**

上記のポイントを考慮して、リスト 1 を変更します。今回は特に詳しく説明はしませんが、入力の A、B、C と出力の 8bit の Y をそれぞれ反転させればよいのです。Verilog-HDL での反転の演算子は「~」です。これを使用して修正します。

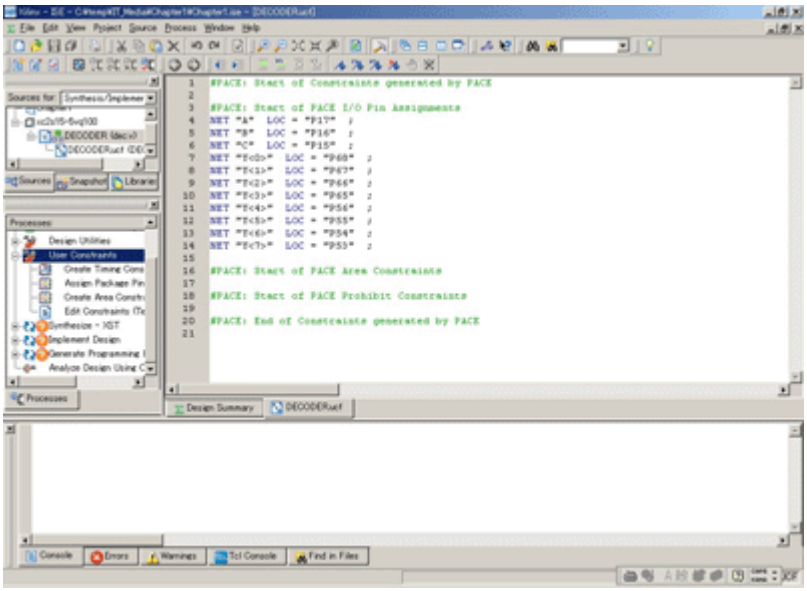
ISE WebPACK 上で dec.v をリスト 2 のように書き換えて、上書き保存（［Ctrl］＋［S］キー）します。

```
1  module DECODER(A,B,C,Y);
2  input A,B,C;
3  output [7:0] Y;
```

```
4  reg [7:0] Y;
5
6  always @(A or B or C)
7  begin
8     case({A,B,C})
9              3'b000:Y=~(8'b00000001);
10             3'b001:Y=~(8'b00000010);
11             3'b010:Y=~(8'b00000100);
12             3'b011:Y=~(8'b00001000);
13             3'b100:Y=~(8'b00010000);
14             3'b101:Y=~(8'b00100000);
15             3'b110:Y=~(8'b01000000);
16             3'b111:Y=~(8'b10000000);
17             default:Y=8'bxxxxxxxx;
18    endcase
19 end
20 endmodule
```

リスト 2　変更を加えた「3 入力 8 出力のデコーダ」の Verilog-HDL ソース

ソースを修正すると、画面左の［Processes］ツリーにある「Synthesize - XST」「Implement Design」「Generate Programming File」にオレンジの「？」マークが表示されます。これはソースファイルがアップデートされたことを示すので、再び FPGA データの生成を行う必要があります。



画面 6　ソースファイルがアップデータされたときの様子

それでは、もう一度「配置配線」「FPGA のデータ生成」を行いましょう。画面左の［Sources］ツリーにある「DECODER（dec.v）」を選択し、同じく画面左の［Process］ツリーにある「Generate Programming File」を右クリックして［Run］を実行します（**注**）。

注：「論理合成」も自動的に実行してくれます。

正常終了したら BitCfg を起動して、FPGA データをボードにダウンロードしてください。

さて、今度はいかがでしょうか？ スイッチが何も押されていないときには、右端の赤色 LED だけが点灯し、スイッチがすべて押されているときには左端の LED だけが点灯しているはずです。スイッチの押し方は全部で 8 通りありますので、一通りボード上で動作を確認してみましょう。

このように HDL を使用すると、入力や出力の反転程度の変更であれば非常に簡単に行えます。これを回路図で行うと、入力 3bit と出力 8bit のすべてに反転素子を置く必要があり、修正が非常に面倒です。この修正変更の容易さは、HDL で設計するメリットの 1 つであるといえます。また、回路修正をした場合にその変更をすぐに回路に反映できる点が FPGA の一番の特長といえます。

◇

いかがでしたでしょうか？ 今回は肩慣らしということで、詳しい説明抜きに FPGA ボードに回路を作成する手順を紹介しました。この先の連載では、もう少し詳しく FPGA の中身や Verilog-HDL の文法的な要素を解説していきたいと思います。

次回は、**「7 セグメント LED を使った回路作成」**を中心に Verilog-HDL についても解説する予定です。

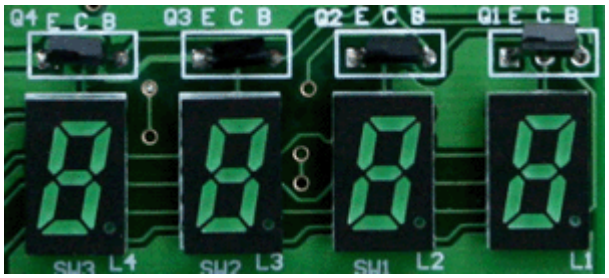


写真 3　7 セグメント LED

論理シミュレーションを行う癖を付けよう

触って学ぼう FPGA 開発入門（2）
論理シミュレーションを行う癖を付けよう

鳥海 佳孝　設計アナリスト　2007/2/16

本連載は、「これから FPGA を開発してみよう！」という入門者の方や仕事で FPGA の設計をされている方（特に新人の方）を対象にしています。また「理論より実践」を主眼とし、入手しやすい無償開発ツールと低価格の FPGA ボードを題材に解説していきます。実際にツールとトレーニングボードを動かして楽しみながら学んでいきましょう。（編集部）

前回は、**「理論より実践」**をテーマに「3 入力 8 出力のデコーダ」の開発と動作確認を行いました。第 2 回目は、以下の 3 点を中心に解説していきます。

1. 前回の verilog-HDL デコーダのソースを解説
2. 7 セグメント LED のデコーダの作成
3. テストベンチを用いた論理シミュレーションの実行

注：「ISE WebPACK」などツールの使用方法は、**第 1 回 理論より実践！ FPGA開発をスタートしよう**を参考にしてください。

関連記事：

- いまさら聞けない FPGA入門

Verilog-HDL ソースの中をのぞいてみよう

まずは、**第 1 回目のリスト 2**のVerilog-HDLソースの中身を詳しく見ていきましょう。

1 行目

「module」を宣言し、モジュール名として「DECODER」（モジュール名は任意の名前）を命名します。「( )」の中に入出力ポートの名前を列挙（ポートリストを指定）します。ここでは「DECODER」モジュールの入出力である「A」「B」「C」「Y」の 4 つの入出力ポートを指定します。

2～3 行目

ポートリストで指定した信号の方向を記述しています。入力には「input」、出力には「output」を宣言します。ちなみに、入出力双方向の場合には「inout」を宣言します。

4 行目

Verilog-HDL の代入には、assign 文による「継続的代入文」と always 文、initial 文による「手続的代入文」があります。継続的代入文で代入される、つまり左辺で使用される信号（変数）は wire 宣言し、「手続的代入文」の中で左辺で使用される信号（変数）は reg 宣言する必要があります。6 行目からの always 文で出力の「Y」が左辺で使用されているので、ここでは「Y」を reg 宣言します。

6 行目

always 文を使用して動作を記述します。「( )」内の記述は、入力信号である「A」「B」「C」をセンシティビティ・リストに指定しています。つまり、「A」「B」「C」のいずれかが変化したら always 文中の「begin」と「end」で囲まれている代入処理が行われます。

7～18 行目

case 文を使用して「A」「B」「C」のすべての組み合わせを記述します。「A」「B」「C」を 3bit の信号と見立てるため「{ }」を使用して、3bit に連接した信号を作成します。例えば、8 行目の case 文「case(~{A,B,C})」で「~」の反転演算子を利用して参照していますので「3'b010」というのは、

「A が 1：A が押されていない」
「B が 0：B が押されている」
「C が 1：C が押されていない」

場合を示しています。

入力信号は、3bit なので「1」「0」を使用したすべての組み合わせは 8 通りです。しかし、Verilog-HDL では「1」「0」以外にも「z：ハイ・インピーダンス」「x：アンノン」の状態が存在します。

「A が z」
「B が 0」
「C が x」

例えば、「3'bz0x」のような場合は「default」に処理が飛ぶようにします（「z」「x」まで考慮すると組み合わせが膨大になるので、default で記述するのがよいでしょう）。

case 文ですべての組み合わせを記述したら、最後に「endcase」を記述します。

20 行目

最後にモジュールの終わりを示す「endmodule」を記述して、実際に回路となる部分の記述は終了です。

注：Verilog-HDL の文法など詳しい解説は、書籍などを参考にしてください。

7 セグメント LED を点灯させよう（1）

今回は、ボードにある 7 セグメント LED を点灯させてみましょう。

ここでは、入力スイッチ「A」「B」「C」の押され方によって、7 セグメント LED の表示部分に「0～7」を表示させます。

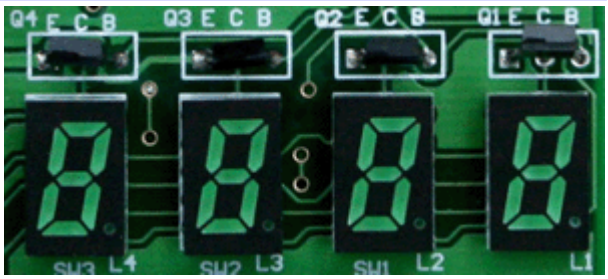


写真 1　7 セグメント LED

「7 セグメント LED のデコーダ」のソース

リスト 1に「7 セグメントLEDのデコーダ」のソースを示します。

```
1  module DECODER7(A,B,C,LED,SA);
2  input A,B,C;
3  output [7:0] LED;
4  output [3:0] SA;
5  reg [7:0] LED;
6
7  assign SA = 4'bzzz0;
8
9  always @(A or B or C)
10 begin
11    case({~A,B,C})        //ABCDEFG Dp
12          3'b000:LED <= 8'b0000001_1;
13          3'b001:LED <= 8'b1001111_1;
14          3'b010:LED <= 8'b0010010_1;
15          3'b011:LED <= 8'b0000110_1;
16          3'b100:LED <= 8'b1001100_1;
17          3'b101:LED <= 8'b0100100_1;
18          3'b110:LED <= 8'b0100000_1;
19          3'b111:LED <= 8'b0001101_1;
20        default:LED <= 8'b0110000_1;
21    endcase
22 end
23 endmodule
```

リスト 1　7 セグメントLEDのデコーダ（DECODER7.v）

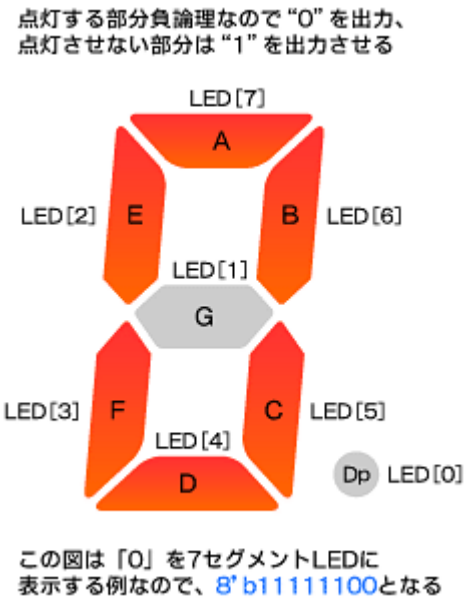


図 1　ビットアサインについて

前回作成した「3 入力 8 出力のデコーダ」と異なる主な点を以下に示します。

1. 出力の信号名を「Y」から「LED」に変更した
2. LED の出力の値を変更した（ビットアサインに関しては、**図 1** を参照）
3. 「 _ 」を使用すると代入する固定値を区切ることができる。ここでは Dp（ドットポイント）だけを区別した
4. 代入記号を「=」（ブロッキング代入）から「<=」（ノンブロッキング代入）に変更した（今回の場合は、どちらの記号を使用しても代入結果は同じ）
5. このボードの 7 セグメント LED は、ダイナミック点灯を使用している。使用する 7 セグメント LED を選択する必要があるので、一番右側にある 7 セグメント LED のけたを選択するために「0」を出力して、それ以外の選択しない 7 セグメント LED のけたに対しては「z」を与える（**リスト 1 の 7 行目**）。ハイ・インピーダンス「z」を与えることに関しては、ボードのマニュアルを参照

この Verilog-HDL ソースを使って、前回と同様に ISE WebPACK で「論理合成（XST）」を実行します。

7セグメントLEDを点灯させよう（2）

ピン固定の実行

論理合成が正常に終了したら、「FPGAのピン固定」ツール（Xilinx PACE）で7セグメントLEDの「ピン固定」を行います。

出力信号の名称が「Y」から「LED」へ変更になった点と出力先が7セグメントLEDになった点を考慮して、それぞれの信号に対してピン番号を与えます。あらかじめ、リスト2を作成してプロジェクト作成時にHDLと一緒に読み込んでも構いません。

```
1  #PACE: Start of Constraints generated by PACE
2  #PACE: Start of PACE I/O Pin Assignments
3  NET "A"  LOC = "P17"  ;
4  NET "B"  LOC = "P16"  ;
5  NET "C"  LOC = "P15"  ;
6  NET "LED<0>"  LOC = "P41"  ;
7  NET "LED<1>"  LOC = "P40"  ;
8  NET "LED<2>"  LOC = "P31"  ;
9  NET "LED<3>"  LOC = "P30"  ;
10 NET "LED<4>"  LOC = "P22"  ;
11 NET "LED<5>"  LOC = "P21"  ;
12 NET "LED<6>"  LOC = "P20"  ;
13 NET "LED<7>"  LOC = "P19"  ;
14 NET "SA<0>"  LOC = "P46"  ;
15 NET "SA<1>"  LOC = "P45"  ;
16 NET "SA<2>"  LOC = "P44"  ;
17 NET "SA<3>"  LOC = "P43"  ;
18
19 #PACE: Start of PACE Area Constraints
20
21 #PACE: Start of PACE Prohibit Constraints
22
23 #PACE: End of Constraints generated by PACE
```

リスト2　7セグメントLEDのピン固定ファイル（DECODER7.ucf）

I/O Name	Loc
A	P17
B	P16
C	P15
LED<0>	P41
LED<1>	P40
LED<2>	P31
LED<3>	P30
LED<4>	P22
LED<5>	P21
LED<6>	P20
LED<7>	P19
SA<0>	P46
SA<1>	P45
SA<2>	P44
SA<3>	P43

配置配線とダウンロード

ピン固定が完了したら、ISE WebPACKで「配置配線」を実行して、FPGA用のデータをボードにダウンロードしてみましょう。

いかがでしょうか？ 入力スイッチが押されていないときには「0」が表示され、すべて押したときには「7」が表示されるはずです。また、ほかのスイッチの組み合わせも試してみましょう。

論理シミュレータを使ってみよう（1）

これまでは、回路になる部分（いわゆるRTL）を記述して、ボード上にそのデータをすぐに転送して動作を確認していました。

今回の回路規模（7セグメントLEDのデコーダ）程度であれば、この方法でも問題はありません。しかし、もっと大規模な回路の場合にはそうはいきません。なぜなら、ボード上での動作に不具合があった場合に、実機からでは問題個所の特定が困難だからです。

一般的には、記述したHDLの動作が正しいかどうかを確認するために**「論理シミュレーション」**を実行します。論理シミュレーションで正しく動作しないものは、いくらボード上で試してもうまくいきません。必ず**「論理シミュレーションを実行してから、実機で検証する」**癖を付けた方がよいでしょう。

論理シミュレータのインストール

本連載で使用する論理シミュレータは「ModelSim」です。ModelSimは、Verilog-HDLだけではなくVHDLでも使えます。機能的な制限はありますが、ザイリンクスのホームページから無料で入手できます。

http://japan.xilinx.com/ise/mxe3/license.htmのライセンス契約画面で［I Agree］ボタンをクリックします。次に、「ModelSim Xilinx Edition III Download Page」の画面で「mxe_3_6.2c.zip」をダウンロードします（原稿執筆時点のバージョンは「6.2c」）。

ダウンロードが完了したら、mxe_3_6.2c.zip を解凍して「Setup.exe」を実行します。

「Select Components」ダイアログで［MXE III Starter］にチェックを入れて［Next］ボタンでインストールを進めます。途中、「Select Library Installation Option」ダイアログが表示されたら「Full Verilog」を選択して［Next］および［はい］でインストールを進めます。

最後に「ModelSim XE III Setup Complete」ダイアログが表示されるので［Finish］をクリックします。

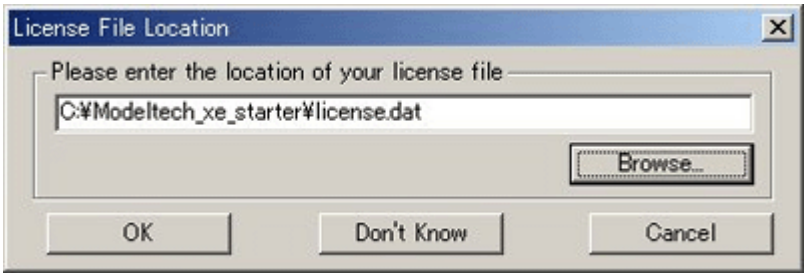
Web ブラウザが起動して「ModelSim Xilinx Edition License Request」が表示されます。ここで［Register］をクリックして、「User ID」「Password」を入力後［Sign In］します。そして、Web ブラウザの［戻る］を 2 回押して「ModelSim Xilinx Edition License Request」の画面に戻り［Continue］をクリックします。

「ModelSim Xilinx Edition - Starter License Request Form」画面が表示されるので、必須項目を入力して［Submit］をクリックします。続いて、ライセンスに関する情報ダイアログが表示されるので［OK］をクリックします。ライセンス送付画面表示後、電子メールでライセンスファイルが送付されます。

入手したライセンスファイル（license.dat）を任意のフォルダに保存します（本連載では、C:¥Modeltech_xe_starter 以下に保存しています）。

次に、［スタート］－［プログラム］－［ModelSim XE III 6.2c］－［Licensing Wizard］でライセンス設定ツールを起動します。「Welcome to the ModelSim License Wizard」ダイアログが表示されたら［Continue］をクリックします。「License File Location」ダイアログでライセンスファイル（license.dat）を指定して［OK］をクリックします。

続いて［Yes］ボタンをクリックします。ライセンスファイルが「LM_LICENSE_FILE 変数」に指定されたこと示すダイアログが表示されたら［OK］ボタンでライセンス設定を完了します。



画面 1　License File Location ダイアログ

論理シミュレータを使ってみよう（2）

テストベンチの記述

まず、**リスト 1** のRTLの動作を確認するためにテストベンチを記述します（**リスト 3**）。

```
1  module TEST_DECODER7;
2  reg A,B,C;
3  wire [7:0] LED;
4  wire [3:0] SA;
5
6  parameter CYCLE = 100;
7
8  DECODER7 i0(.A(A), .B(B), .C(C), .LED(LED), .SA(SA));
9
10 initial
11 begin
12     {A,B,C}=3'b000;
13     #CYCLE {A,B,C}=3'b001;
14     #CYCLE {A,B,C}=3'b010;
15     #CYCLE {A,B,C}=3'b011;
16     #CYCLE {A,B,C}=3'b100;
17     #CYCLE {A,B,C}=3'b101;
18     #CYCLE {A,B,C}=3'b110;
19     #CYCLE {A,B,C}=3'b111;
20     #CYCLE $finish;
21 end
22
23 endmodule
```

リスト 3　7 セグメントLEDデコーダのテストベンチ（TEST_DECODER7.v）

以下に、テストベンチの内容を解説します。

1 行目

テストベンチには入出力がないので、ポートリストは記述しません。モジュール名だけ記述します。

2～4 行目

7 セグメント LED のデコーダの入力端子につなぐ信号（変数）「A」「B」「C」は、手続的代入文（always 文、initial 文）の中で左辺で使われているので reg 宣言を行います。それ以外の信号（変数）「LED」「SA」は、wire 宣言を行います。

6 行目

parameter 文で 1 サイクルの長さを指定します。

8 行目

7 セグメント LED のデコーダをインスタンスします。

10～21 行目

「A」「B」「C」の入力ポートへの連接子「{ }」を使用した信号代入を記述します（「0」「1」で考えられるすべての組み合わせを入力）。parameter 文で指定した時間だけずらして「A」「B」「C」の値を変えています。

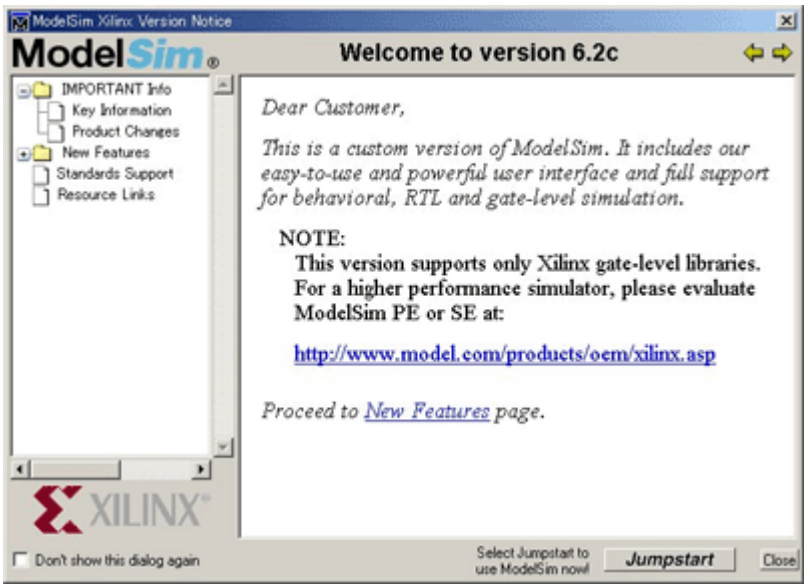
注：厳密には、RTL 中に記述されている case 文の「default」に処理が飛ぶかどうかを確認するために、8 つの組み合わせ以外の「z」「x」などを入力として与える必要があります。

このテストベンチでは、7 セグメントの LED デコーダの入力ポート「A」「B」「C」に「3'b000～3'b111」の 8 通りの組み合わせの信号を 100 ステップごとに変化させて代入し、出力ポート（「LED」「SA」）の変化をチェックします。

シミュレーションの実行

テストベンチの準備ができたら、シミュレーションを実行してみましょう。

［スタート］－［プログラム］－［ModelSim XE III 6.2c］－［ModelSim］で論理シミュレータを起動します。最初に「Welcome to Version 6.2c」ダイアログが表示されるので右下の［Close］ボタンで閉じます。

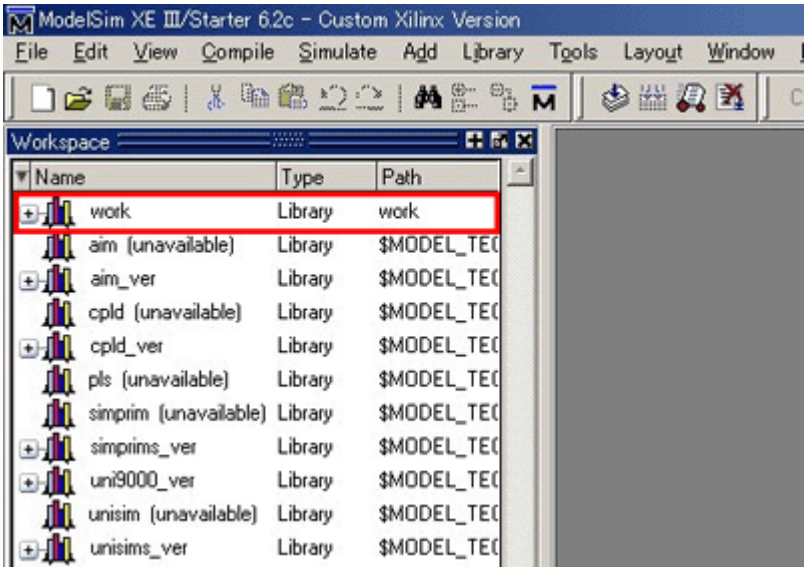


画面 2　Welcome to Version 6.2c ダイアログ

起動したらメニューの［File］－［Change Directory...］を選択して、「Choose folder」ダイアログを表示します。ここで、RTL とテストベンチのファイルがあるフォルダを設定して［OK］ボタンをクリックします。

続いて、メニューから［File］－［New］－［Library...］を選択します。「Create a New Library」ダイアログが表示されるので、そのまま［OK］ボタンをクリックします。

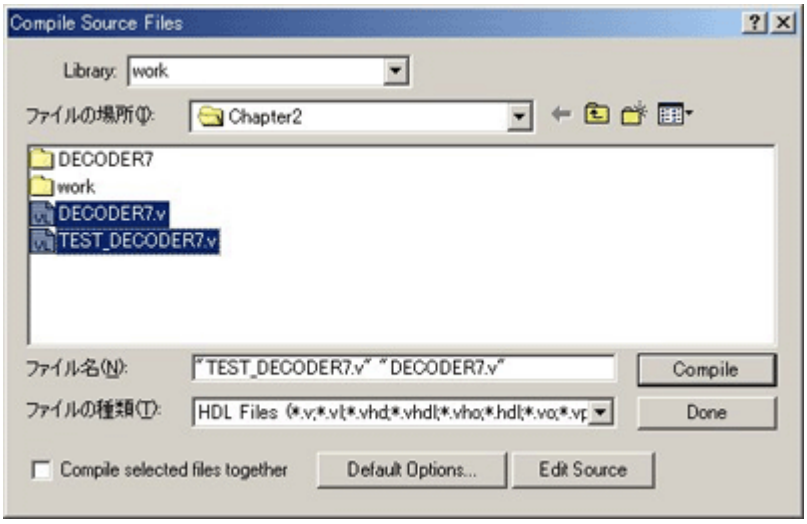
画面左の「Workspace」ツリーに「work」ライブラリが追加されます。



画面 3　work ライブラリが追加された様子

注：ModelSim では、コンパイル結果はすべて work と呼ばれるライブラリフォルダに格納されます。**画面 3** のように work ライブラリがすでに表示されていれば、設定の必要はありません。

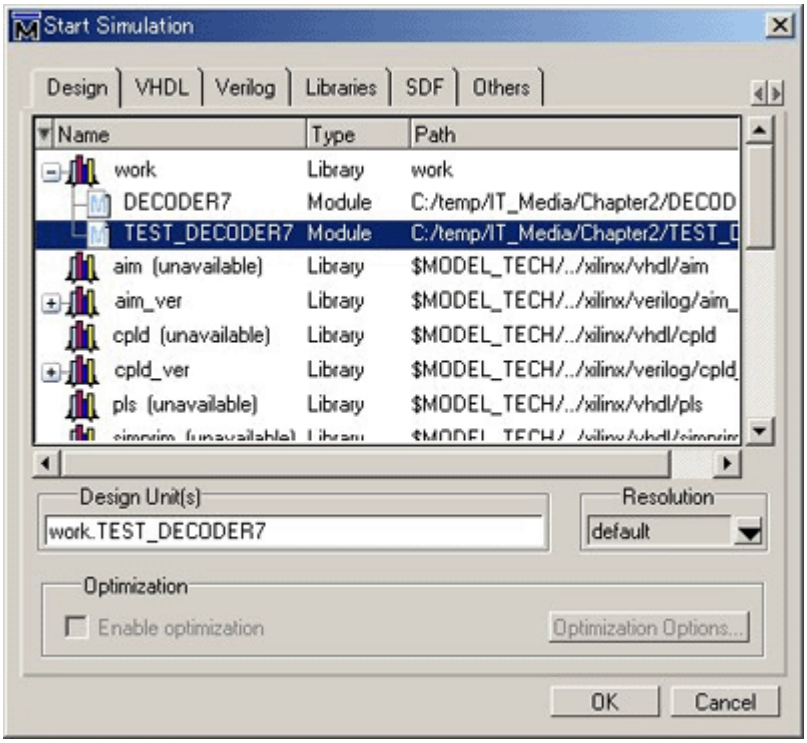
メニューから［Compile］－［Compile...］を選択します。「Compile Source Files」のダイアログが表示されるので、RTL とテストベンチのファイルを選択して［Compile］ボタンをクリックします。



画面 4　Compile Source Files ダイアログ

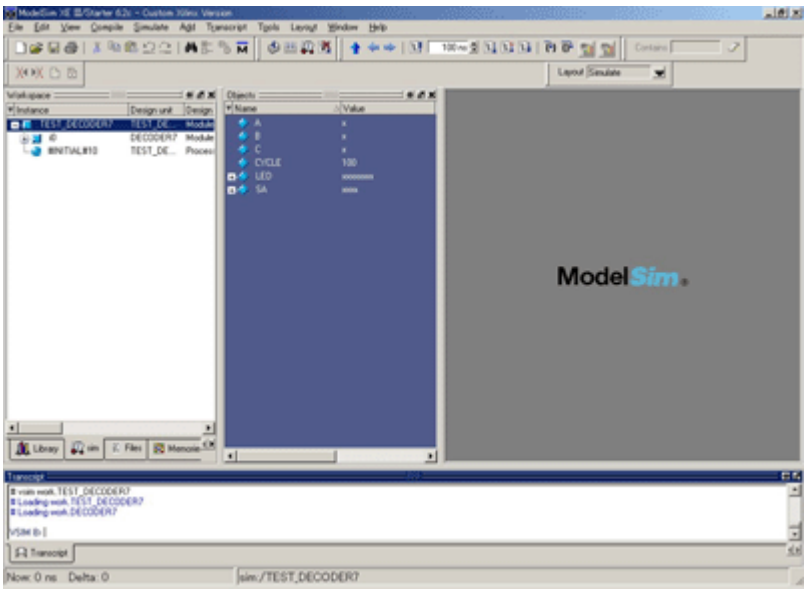
注：コンパイル結果は、画面下のログウィンドウに表示されます。エラーの場合は、赤い文字でエラーメッセージが出力されるので、それを基にファイルを修正してください（以下同）。

次に、メニューの［Simulate］－［Start Simulation...］をクリックします。「Start Simulation」ダイアログが表示されるので、［Design］タブのツリーから「work」を探して「+」をクリックして展開します。最上位階層、ここでは「TEST_DECODER7」を選択して［OK］ボタンをクリックします。



画面 5　Start Simulation ダイアログ

画面中央に「Objects」が表示されるので、波形表示したい信号を選択します（ここでは「A」「B」「C」「CYCLE」「LED」「SA」のすべてを選択します）。



画面 6　波形表示したい信号を選択する

メニューから［Add］－［Wave］－［Selected Signals］を選択します。選択された信号が画面右の「wave - default」に表示されます。続けて、メニューの［Simulate］－［Run］－［Run -All］を選択します。

テストベンチに「$finish;」が記述されている（**リスト 3 の 20 行目**）ので、シミュレーションを終了するかどうか尋ねてきます（**画面 7**）。ここでは、波形で動作を確認したいので［いいえ］ボタンをクリックします。ちなみに「$finish;」の代わりに「$stop;」を記述すれば、終了するかどうかを確認するダイアログは表示されません。

画面 7　Finish Vsim ダイアログ　ここでは［いいえ］を選択する

画面右「wave - default」の右上にある「+」（Zoom/Unzoom window）をクリックすると、波形表示によるシミュレーション結果が拡大表示されます。「A」「B」「C」の入力に対して、LED の出力が変化していることが確認できます。

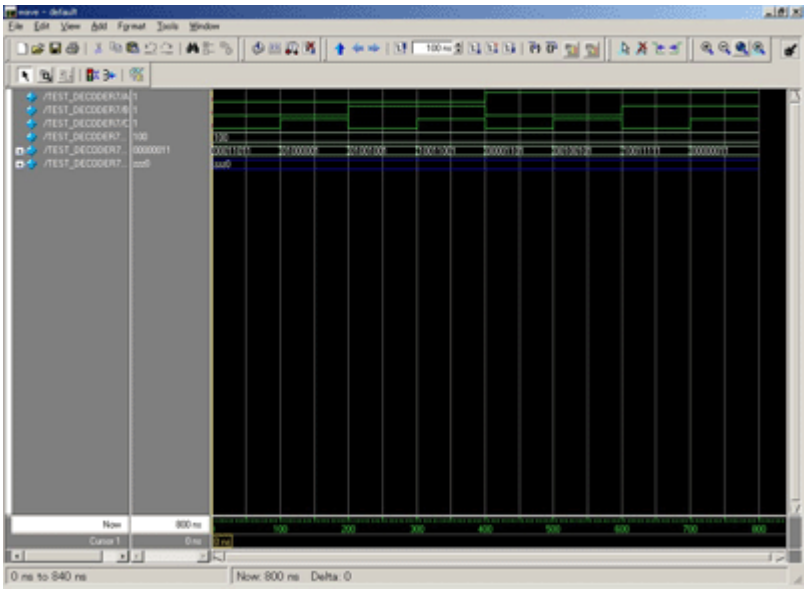
回路が大規模化すればするほど､実機動作から不具合を特定することは難しくなります｡そのため､論理シミュレータで回路になる部分､つまり RTL の記述が正しいかどうかを確認してから実機に転送するようにしましょう。論理シミュレーションで正しく動作しないデータをいくらボード上に転送しても、ボード上では絶対に正常に動作しません。

◇

いかがでしたでしょうか？ 今回は､7 セグメント LED の表示と論理シミュレーションに挑戦しました。このくらいの回路規模であれば「論理シミュレーションなんて……」と感じているのではないでしょうか？しかし、実機で動作させる前にシミュレーションを行うことはとても大切です。面倒くさいからと敬遠せずに、シミュレータの実行をぜひ心掛けてください。

次回は、**「順序回路（カウンタ）」**をボード上で動作させる予定です。

（次回に続く）



画面 8　波形表示によるシミュレーション結果の確認

触って学ぼう FPGA 開発入門（3）
順序回路の基本！ カウンタを作成しよう

鳥海 佳孝　設計アナリスト　2007/3/16

本連載は、「これから FPGA を開発してみよう！」という入門者の方や仕事で FPGA の設計をされている方（特に新人の方）を対象にしています。また「理論より実践」を主眼とし、入手しやすい無償開発ツールと低価格の FPGA ボードを題材に解説していきます。実際にツールとトレーニングボードを動かして楽しみながら学んでいきましょう。（編集部）

前回は、7 セグメントLEDのデコーダとテストベンチを作成してシミュレーションを行いました。皆さん正しく動作させることができたでしょうか？ しつこいようですが､実機で動かす前にシミュレーションを実施する癖を付けましょう。

さて、今回は「4bit のカウンタ」を作成しながら「**順序回路**」の基本について解説します。ちなみに、これまでの連載で紹介した回路は「**組み合わせ回路**」という種類の回路です。

補足：「組み合わせ回路」と「順序回路」
組み合わせ回路とは、「入力の変化に対応して、出力が変化する回路」のことを指します。組み合わせ回路の場合は、値を保持することができません。 順序回路とは、「現在の入力のみで出力が決まるのではなく、過去の入力にも依存する回路」のことを指します。最近の順序回路はほとんど、D フリップフロップです。この D フリップフロップにあるようにクロックを使用して値を保持します。

注：「ISE WebPACK」などツールの使用方法は、**第 1 回 理論より実践！ FPGA開発をスタートしよう**を参考にしてください。

関連記事：
いまさら聞けない FPGA入門
連載：触って学ぼう FPGA開発入門

4bit フリーランカウンタを作成しよう（1）

4bit のカウンタとは？

まず、今回のテーマ「4bit のカウンタ」について簡単に説明します。

4bit のカウンタとは、「0000（0x0）」～「1111（0xF）」の間を 1 つずつカウントアップ（ダウン）していく回路です。ちなみに、4bit なので出力は赤色 LED で行います。

このカウントは、入力されたクロック信号（0 から 1 への変化）によって行われます。クロックが 0 から 1 へ立ち上がったら、カウンタを＋1 にして値を保持します。そして、直接その値を赤色 LED に出力します。

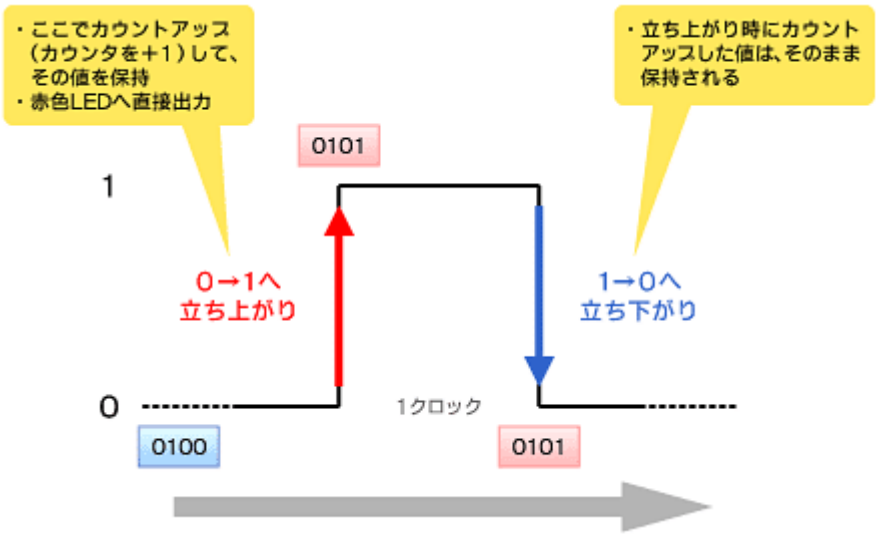


図 1　クロックの立ち上がり、立ち下がり時のイメージ

また、クロックの立ち下がり（1 から 0 への変化）が起きても、立ち上がり（0 から 1 への変化）の時にカウントアップしたカウンタの値はそのまま保持されます。

例えば、カウンタの値が「0101（0x5）」であれば、カウンタの値を保持して**図 2** のように赤色 LED が点灯します。

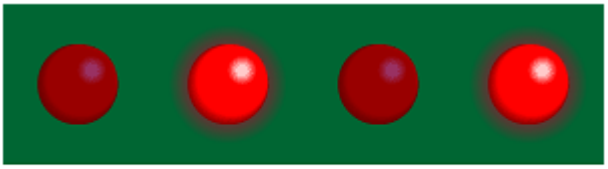


図 2　赤色 LED の点灯イメージ（0101 の場合）

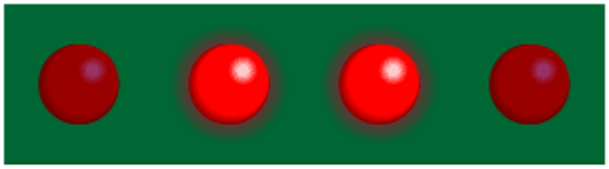


図 3　赤色 LED の点灯イメージ（0110 の場合）

クロックの立ち下がりのタイミングでも、カウンタの値（0101）はそのまま保持されます。

さらにクロックが立ち上がり、先ほど保持していた値（0101）をカウントアップして保持します（0110）。そして、その結果を赤色 LED に出力します（**図 3**）。

クロックの立ち下りの際は、先ほどと同様にカウンタの値（0110）はそのまま保持されます。

今回のポイントは、クロックの立ち上がり（0 から 1 への変化）時にカウントアップして保持した値をクロックが立ち下がった（1 から 0 へ変化）際にも、保持し続けているという点です。

この「値を保持する動き」こそが順序回路の動作なのです。

それでは、今回も「論理よりも実践」です！ まずは、単純な機能を持つフリーランのカウンタをシミュレーション上で動作させてみましょう。

4bit フリーランカウンタのソース

「4bit フリーランカウンタ」の Verilog-HDL ソースを以下に示します（**注**）。

注：前回までの連載で解説した文法については省略していますが、重要と思われる部分に関しては、再度解説を入れてあります（以下同）。

```
1  module COUNT4(RESET, CLK, COUNT);
2  input RESET, CLK;
3  output [3:0] COUNT;
4
5  reg [3:0] COUNT;
6
7  always @(posedge CLK or negedge RESET)
8  begin
9          if (RESET == 1'b0)
10                 COUNT <= 4'h0;
11         else
12                 COUNT <= COUNT + 4'h1;
13 end
```

リスト 1　4bitのフリーランカウンタ（COUNT4.v）

5 行目

「COUNT」という信号は、7～13 行目の always 文の中で左辺として使用されているため、4bit の reg 宣言を行う必要があります。

7 行目

always 文で、カウンタの動作を記述します（**注 1**）。「()」のセンシティビティ・リストには、クロック信号の立ち上がり（positive edge）「CLK」、またはクロック信号の立ち下がり（negative edge）「RESET」を指定します（**注 2**）。CLK の立ち上がり、RESET の立ち下がりごとに、always 文中の「begin」～「end」で囲まれた代入、つまりカウンタの動作が行われます。

注 1：Verilog-HDL の手続きは、「initial」か「always」ブロックの中に記述する必要があります。また、initial ブロックは時刻 0 に 1 度だけ、always ブロックは永久に繰り返し起動されます。

注 2：「立ち上がり」とは、クロック信号が 0 から 1 へと変化する瞬間を指し、「posedge」を使って指定します。また、「立ち下がり」とは、クロック信号が 1 から 0 へと変化する瞬間を指し、「negedge」を使って指定します。

8～13 行目

if 文を使用して、カウンタの動作を記述します。センシティビティ・リストに CLK の立ち上がり以外の信号として RESET の立ち下がりを記述していますので、最初に RESET 信号の条件について記述します（**9 行目**）。これにより RESET 信号が立ち下がれば（0 であれば）、この条件が真になり **10 行目**の代入が行われます。動作としては「非同期リセット」が掛かったことになります。

12 行目は、RESET 信号が「1」のときの条件が記述されています。ここでは、カウントアップ（＋1）の動作を行います。この代入は、CLK 信号が立ち上がっていて、かつ RESET 信号が「1」のときだけに実行されます。つまり、クロックの立ち上がりのときにカウンタがアップします。

以上のことから、この記述の動作は「非同期リセット付き同期カウンタ」になります。

4bit フリーランカウンタを作成しよう（2）

4bit フリーランカウンタのテストベンチ

早速、FPGA ボード上で動作させてみましょう！といいたいところですが、まずはテストベンチを作成し、シミュレーションを実行します。

以下に 4bit フリーランカウンタのテストベンチの記述を示します。

```
1  module TEST_COUNT4;
2  reg clk, reset;
3  wire [3:0] count;
4
5  parameter CYCLE = 100;
6
7  COUNT4 i1(.RESET(reset), .CLK(clk), .COUNT(count));
8
9  always #(CYCLE/2)
10         clk = ~clk;
11
12 initial
14         reset = 1'b0; clk = 1'b0;   ←時刻 0
15         #CYCLE reset = 1'b1;   ←時刻 100
16         #(20*CYCLE) reset = 1'b0;   ←時刻 2100
17         #CYCLE reset = 1'b1;   ←時刻 2200
18         #(20*CYCLE) $finish;   ←時刻 4200、シミュレーションを終了
19 end
20
21 initial
22         $monitor($time,,"clk=%b reset=%b count=%b", clk, reset, count);
23
24 endmodule
```

リスト 2　4bitのフリーランカウンタのテストベンチ（T_COUNT4.v）

2 行目

9 行目の always 文中の「clk」信号、**14～17 行目**の initial 文中の「clk」「reset」信号が手続き的代入文の中で使用されているので、それぞれの信号を reg 宣言します。

3 行目

「count」信号が、reg 宣言または wire 宣言をしなければならない信号としてまだ残っています。count 信号は、bit 幅が 1bit ではないので wire 宣言を省略できません（**注**）。そのため、4bit の wire 宣言を行います。

注：Verilog-HDL の文法では、1bit の wire 宣言は省略可能（宣言不要）となっています。

5 行目

parameter 文で、このシミュレーションにおける 1 クロックサイクルの長さを指定します。この parameter 文の値を変更すれば、クロックの周波数を変更できるので非常に便利です。

9～10 行目

クロックのような繰り返し「0」「1」になるパルスを生成する典型的な記述です。この場合には、時刻 50 ユニットごとに clk 信号が反転されます（**注**）。

注：「#」で時間の経過を指定することができます。

14 行目

reset 信号と clk 信号の時刻 0 での値を代入します。特にこの clk 信号の初期化は大変重要です。これを忘れてしまうと、うまく clk を生成することができません。

ちなみに、最新の Verilog-HDL の規格では、「reg clk = 1'b0;」のように reg 宣言時に初期値を与えることも可能です（この場合、initial 文中での初期化は不要）。

15 行目～18 行目

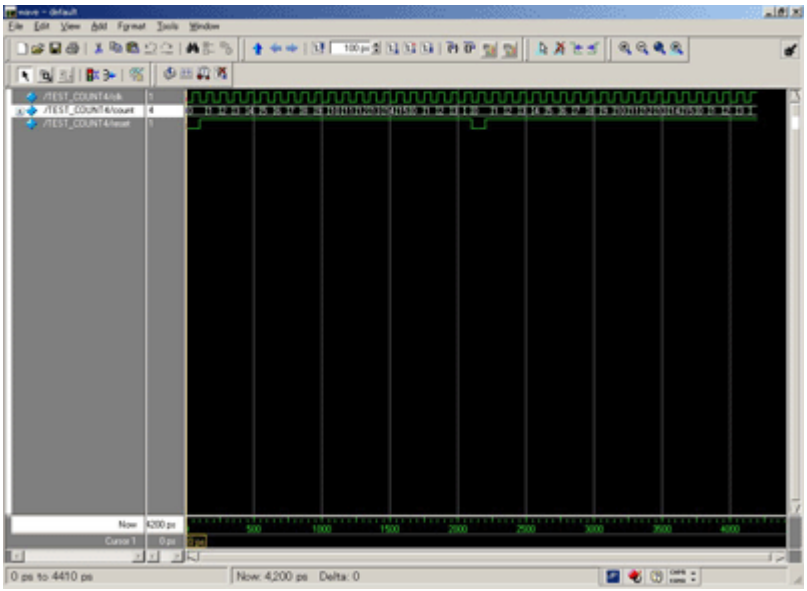
時刻 100 ユニットで reset 信号を「1'b1」にして、時刻 2100 ユニット（100＋2000）で reset 信号を「1'b0」にします。さらに、時刻 2200 ユニット（100＋2000＋100）で reset 信号を再び「1'b1」にして、時刻 4200 ユニット（100＋2000＋100＋2000）で$finish（シミュレーションを終了）します。

シミュレーションの実行

これで、RTL とテストベンチが完成しました。それでは、ModelSim を使ってシミュレーションを実行してみましょう。前回までの記事を参考にコンパイル、シミュレート、実行をして、波形表示で動作を確認してください。

画面 1 のように動作すれば問題ありません。

画面 1 では、結果を確認しやすくするためにカウンタの値を整数値で表示しています。この表示を行うには、「wave-default（波形表示のウィンドウ）」の左側リストから表示を変更したい信号を右クリックして［ショートカットメニュー］－［Radix］－［Unsigned］を選択します（**画面 2**）。

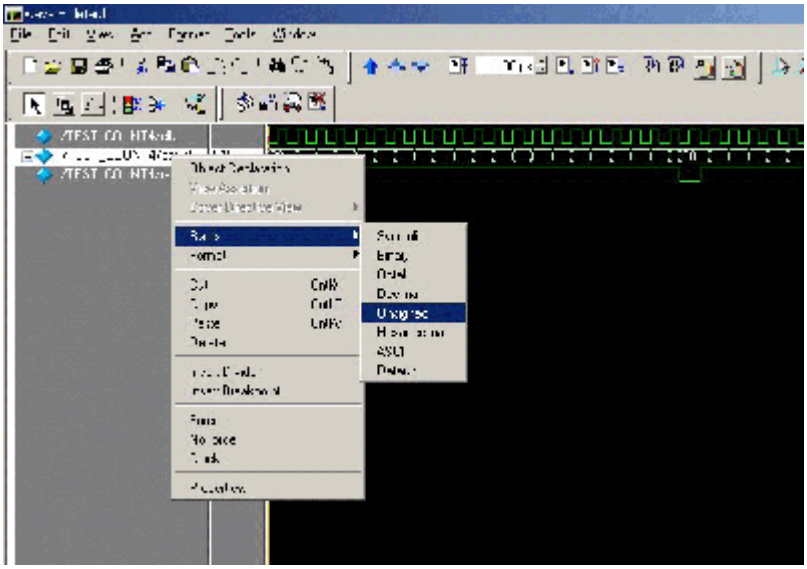


画面 1　波形表示によるシミュレーション結果の確認

FPGA ボード上での動作確認

これで、シミュレーションでの動作確認が取れました。それでは、RTL を FPGA のボード上で動かしてみましょう。

ここでは、手早く動作を確認するために 4bit のカウンタの値を赤色 LED にそのまま出力しています。以下のピンアサインを行って、論理合成、配置配線を実行してください。



画面 2　整数値でカウンタを表示

```
1  NET "CLK"  LOC = "P39"  ;
2  NET "RESET"  LOC = "P17"  ;
3  NET "COUNT<0>"  LOC = "P68"  ;
4  NET "COUNT<1>"  LOC = "P67"  ;
5  NET "COUNT<2>"  LOC = "P66"  ;
6  NET "COUNT<3>"  LOC = "P65"  ;
```

リスト 3　4bitのフリーランカウンタのピンアサイン（COUNT4.ucf）

特に問題がなければ、ボード上に設計データ（bit ファイル）をダウンロードします。

いかがでしょうか？

ボード上で確認してみると、すべての赤色 LED が点灯したような状態になっていると思います。

赤色 LED は、なぜカウント動作をしてくれないのでしょうか？

それは、カウンタが 6MHz で動作しているからです。実際は正しくカウント動作をしているのですが、高速過ぎて人間の目には「すべての LED が点灯しているだけ」のように見えているのです。

I/O Name	Loc
CLK	P39
RESET	P17
COUNT<0>	P68
COUNT<1>	P67
COUNT<2>	P66
COUNT<3>	P65

分周クロックの導入

このままでは、ボード（赤色 LED）上で正しくカウンタが動作しているかを確認することができません。そこで、カウンタのクロックを遅くする方法を紹介します。

あまり良い方法ではありませんが（その理由は、次回説明します）、今回は 6MHz のクロックを分周してカウンタに渡して動作させます。

分周するクロックを入れるために、**リスト 1** を以下のように変更します。

```
1  module COUNT4(RESET, CLK, COUNT);
2  input RESET, CLK;
3  output [3:0] COUNT;
4
5  reg [22:0] tmp_count;
6  reg [3:0] COUNT;
7
8  always @(posedge CLK or negedge RESET)
9  begin
10         if (RESET == 1'b0)
11                 tmp_count <= 23'h000000;
12         else
13                 tmp_count <= tmp_count + 23'h1;
14 end
15
16 assign DIVIDE_CLK = tmp_count[22];
17
18 always @(posedge DIVIDE_CLK or negedge RESET)
19 begin
20         if (RESET == 1'b0)
21                 COUNT <= 4'h0;
22         else
23                 COUNT <= COUNT + 4'h1;
24 end
25
26 endmodule
```

リスト 4　4bitのフリーランカウンタの修正版（COUNT4-2.v）

変更したポイントは、下記のとおりです。

- 6MHz 以上数えられるように 23bit のフリーランのカウンタ（tmp_count）を用意（**5 行目**）
- フリーランのカウンタの最上位 bit（22bit 目）をカウンタのクロックとして入力（**16 行目**）

この変更を行ったら、ISE WebPACK で論理合成、配置配線を行い、設計データを FPGA のボードにダウンロードします。

今回はいかがでしょうか？ カウンタの動作を確認できたと思います。ただし、赤色 LED は負論理なので LED の動作はカウントダウンしています。

このシミュレーションを行うには、かなりのクロック数が必要となりますので、まともにシミュレーションするのは現実的ではありません（カウントが 1 つアップするだけでも、気の遠くなるようなクロック数が必要となります）。この課題を克服するためのシミュレーションテクニックに関しては「**単相同期回路設計**」の解説と併せて次回紹介する予定です。

スイッチによるカウントダウン動作を追加しよう

次は、スイッチにアップ・ダウンの機能を割り当てます。スイッチが押されたらカウントダウンするように変更してみましょう。

4bit アップ・ダウンカウンタのソース

4bit アップ・ダウンカウンタの Verilog-HDL ソースを以下に示します。

```
1  module UPDOWN(RESET, CLK, DEC, COUNT);
2  input RESET, CLK, DEC;
3  output [3:0] COUNT;
4
5  reg [22:0] tmp_count;
6  reg [3:0] COUNT_TMP;
7
8  always @(posedge CLK or negedge RESET)
9  begin
10         if (RESET == 1'b0)
11                 tmp_count <= 23'h000000;
12         else
13                 tmp_count <= tmp_count + 23'h1;
```

```
14 end
15
16 assign DIVIDE_CLK = tmp_count[22];
17
18 always @(posedge DIVIDE_CLK or negedge RESET)
19 begin
20         if (RESET == 1'b0)
21                 COUNT_TMP <= 4'h0;
22         else if (DEC == 1'b1)
23                 COUNT_TMP <= COUNT_TMP + 4'h1;
24         else
25                 COUNT_TMP <= COUNT_TMP - 4'h1;
26 end
27
28 assign COUNT = ~COUNT_TMP;
29
30 endmodule
```

リスト 5　4bitのアップ・ダウンカウンタ（UPDOWN.v）

以下に重要と思われる部分を説明します。

1～2 行目

カウントダウンさせるためのスイッチを割り当てたので、入力ポート「DEC」を追加します。

6 行目

出力する赤色 LED が負論理なので、最終的にカウントの値を反転させる必要があります。そのため一時的にカウンタの値を入れるための信号（変数）として「COUNT_TMP」を reg 宣言します。

22～25 行目

「DEC」のスイッチが押されていないときは「1'b1」なので、カウントアップの動作を記述し、「DEC」のスイッチが押されているときは「1'b0」なので、カウントダウンの動作を記述します。

28 行目

赤色 LED の出力が負論理なので、反転して出力します。

FPGA ボード上での動作確認

それでは、完成した RTL を FPGA のボード上で動かしてみましょう。

リスト 6 のピンアサインを行って、論理合成、配置配線を実行します。

I/O Name	Loc
CLK	P39
RESET	P17
DEC	P16
COUNT<0>	P68
COUNT<1>	P67
COUNT<2>	P66
COUNT<3>	P65

```
1  NET "CLK"  LOC = "P39"  ;
2  NET "RESET"  LOC = "P17"  ;
3  NET "DEC"  LOC = "P16"  ;
4  NET "COUNT<0>"  LOC = "P68"  ;
5  NET "COUNT<1>"  LOC = "P67"  ;
6  NET "COUNT<2>"  LOC = "P66"  ;
7  NET "COUNT<3>"  LOC = "P65"  ;
```

リスト 6　4bitのアップ・ダウンカウンタのピンアサイン（UPDOWN.ucf）

特に問題がなければ、ボード上に設計データ（bit ファイル）をダウンロードします。

いかがでしょうか？ 今度は、何もしないと順調にカウントアップして、真ん中のスイッチを押し続けるとカウントダウンするはずです。

10 進アップ・ダウンカウンタへ変更してみよう

ここまでは、単なるバイナリ（0～F）のカウンタの動作でした。次に、カウンタを 10 進（0～9）で動作させてみましょう。

10 進アップ・ダウンカウンタのソース

10 進アップ・ダウンカウンタの Verilog-HDL ソースを以下に示します。

```
1  module UPDOWN(RESET, CLK, DEC, COUNT);
2  input RESET, CLK, DEC;
3  output [3:0] COUNT;
4
5  reg [22:0] tmp_count;
6  reg [3:0] COUNT_TMP;
```

```
7
8  always @(posedge CLK or negedge RESET)
9  begin
10         if (RESET == 1'b0)
11                 tmp_count <= 23'h000000;
12         else
13                 tmp_count <= tmp_count + 23'h1;
14 end
15
16 assign DIVIDE_CLK = tmp_count[22];
17
18 always @(posedge DIVIDE_CLK or negedge RESET)
19 begin
20         if (RESET == 1'b0)
21                 COUNT_TMP <= 4'h0;
22         else if (DEC == 1'b1)
23                 if (COUNT_TMP == 4'h9)
24                         COUNT_TMP <= 4'h0;
25                 else
26                         COUNT_TMP <= COUNT_TMP + 4'h1;
27         else
28                 if (COUNT_TMP == 4'h0)
29                         COUNT_TMP <= 4'h9;
30                 else
31                         COUNT_TMP <= COUNT_TMP - 4'h1;
32 end
33
34 assign COUNT = ~COUNT_TMP;
35
36 endmodule
```

リスト 7　4bitの 10 進アップ・ダウンカウンタ（UPDOWN10.v）

以下に重要と思われる部分を説明します。

23～31 行目

カウントアップ時とカウントダウン時のどのタイミングで、0 または 9 に戻すのかがポイントとなっています。プログラムが得意な方であれば、一度カウントアップするか、あるいはカウントダウンしてからその値を 0 ないし 9 に戻す、という方法を思い付くでしょう。その辺りの理屈は次回に譲ることにして、今回は、

- カウントアップ時：COUNT_TMP の値が 9 のときに 0 を代入
- カウントダウン時：COUNT_TMP の値が 0 のときに 9 を代入

と記述をします。

ポイントは、条件を分離してそれに対して if 文をネスティング（入れ子）にして記述することです。通常は、順調にカウントアップしてほしいので、上記の条件がある意味「例外」です。このような条件では、if 文をネスティングさせて優先順位付けをしっかりと行います（**リスト 5** からの変更点はこれだけです）。

FPGA ボード上での動作確認

完成した RTL を FPGA のボード上で動かしてみましょう。

リスト 6をそのまま使用してピンアサイン、論理合成、配置配線を実行します。特に問題がなければ、ボード上に設計データ（bitファイル）をダウンロードします。

いかがでしょうか？ 何もしない状態だと順調にカウントアップし、なおかつ 9（両側の赤色 LED が点灯）の次が 0（すべて消灯）になるはずです。また、真ん中のスイッチを押し続けるとカウントダウンし、なおかつ 0（すべて消灯）の次が 9（両側の赤色 LED が点灯）になるはずです。

◇

次回は、FPGA 設計を行ううえで基本となる「**単相同期設計**」を紹介します。また、分周クロックを使用せずにカウンタの動作を遅らせる方法とその際のシミュレーション方法について解説する予定です。（次回に続く）

単相同期回路で設計する理由

触って学ぼう FPGA 開発入門（4）

単相同期回路で設計する理由

鳥海 佳孝　設計アナリスト　2007/4/12

本連載は、「これから FPGA を開発してみよう！」という入門者の方や仕事で FPGA の設計をされている方（特に新人の方）を対象にしています。また「理論より実践」を主眼とし、入手しやすい無償開発ツールと低価格の FPGA ボードを題材に解説していきます。実際にツールとトレーニングボードを動かして楽しみながら学んでいきましょう。（編集部）

第 3 回「順序回路の基本！ カウンタを作成しよう」では、順序回路の 1 つである「4bitのカウンタ」を設計して、FPGAボード上にインプリメント（作成）しました。ただし、カウンタを動作させるクロックに関しては元の 6MHzのクロックではなく、分周して作成したクロックで動作させました。

しかし、この方法は最近の設計方法としてはあまり好ましいものではありません。FPGA 設計の基本は（FPGA に限ったことではないですが）、**“単相同期回路”**つまり**“1 本のクロックですべてのフリップフロップ（順序回路）を駆動する”**ことです。今回はこのあたりの内容を徹底的に解説します。

関連記事：
いまさら聞けない FPGA入門
連載：触って学ぼう FPGA開発入門

なぜ単相同期回路なのか？

ここでは、“なぜ単相同期回路で作成する必要があるのか”を解説します。

スタッティックな遅延解析（Static Timing Analysis：STA）

ここでは、単相同期回路について説明する前に、「ISE WebPACK」などのツール（論理合成、配置配線）がどのような方針の下に回路を評価し、最終的にどのようにして回路にするのかを解説します。

これらのツールは**「静的なタイミング解析」**、いわゆる**“スタティックな遅延解析”**を行い作成する回路を評価して決定します。では、スタティックな遅延解析とは、どのようなものなのでしょうか？

実は、それほど難しいものではありません。

図 1 にあるように、NOT 素子（1ns の素子遅延）、NAND 素子（2ns の素子遅延）、NOR 素子（3ns の素子遅延）、それぞれの各素子に入力されてから出力されるまでの遅延値が**図 1** のように決まっているとします（この例では配線による遅延は考慮しません）。

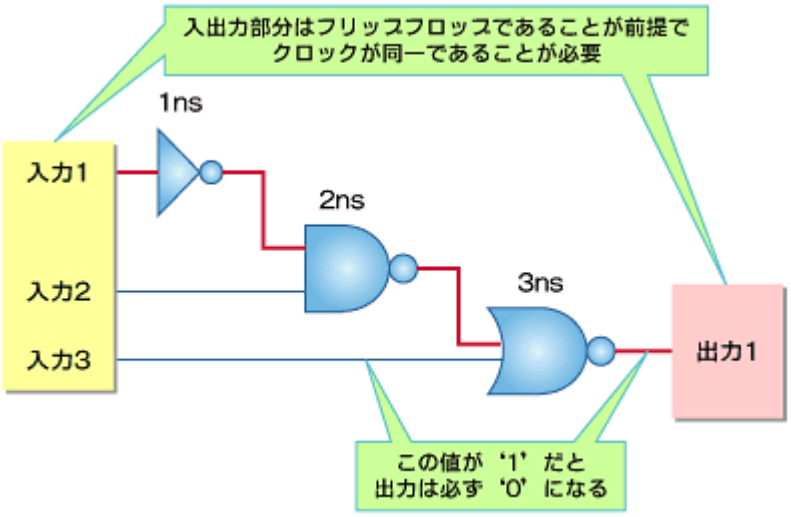


図 1　スタティックな遅延解析の仕組み

この前提で考えると、入力から出力まで一番時間がかかるパスは、1ns（NOT 素子）＋2ns（NAND 素子）＋3ns（NOR 素子）＝6ns のパスと評価されます（**図 1** では赤の太線で示すパス）。この評価が、その回路の実力（最大動作周波数（1/6ns＝約 166MHz））として判断されるのです。

さて、最大動作周波数が求められましたが、本当にこれ以上速いクロックで動作しないものなのでしょうか？

ここで論理的なことを考慮してみましょう。もし、［入力 3］の値が「1」だったらどのようになるでしょうか？ **図 1** にあるように最終段の NOR の入力には「1」が入力されます。この場合だと、NOR の片方の入力が「1」なので、もう片方の入力は「0」でも「1」でも出力には依存しません。つまり、この場合のように“論理値”を考慮すると、最大動作周波数は、

3ns（NOR 素子のみ）＝3ns

になります。シミュレーションのテストベンチのように、それぞれの素子の論理値を考慮して遅延を評価する方法を**「ダイナミック（動的）な遅延解析」**と呼びます。

それでは、「動的な遅延解析」と「静的な遅延解析」を比べてみましょう。

ツール上での計算量はどちらが多いでしょうか？ これは明らかで、「ただ単に素子遅延を足し込んで遅延解析を行っているものより、各素子の論理値まで考慮して遅延解析する方が計算量が多い」ということは直感的に分かると思います。

つまり、最近の大規模回路の場合、ダイナミックな遅延解析を行うとかなりのマシンパワーが必要となります。そのため、性能的な評価を行う際は、スタティックな遅延解析を用いることが多いようです。

図 1 の回路は、スタティックな遅延解析により「6ns」と評価されました。しかし、ここには大きな前提があります。この回路の入出力部分に相当する、フリップフロップ（F/F）が遅延解析の**“始点”**と**“終点”**であるということです（同じクロックを F/F のクロック端子に入力しないと前提が崩れてしまいます）。

一般的に FPGA のツール（に限ったことではありませんが）は、上述したようにスタティックな遅延解析を行っているので、これらのツールの一番得意な回路というのは、**図 2** のように入力される信号をいったん F/F で受けて、出力する信号を必ず F/F で出力し、同位相のクロックですべての F/F を駆動するというものです。つまり、各 F/F のクロック端子に入力されている回路が得意なのです。

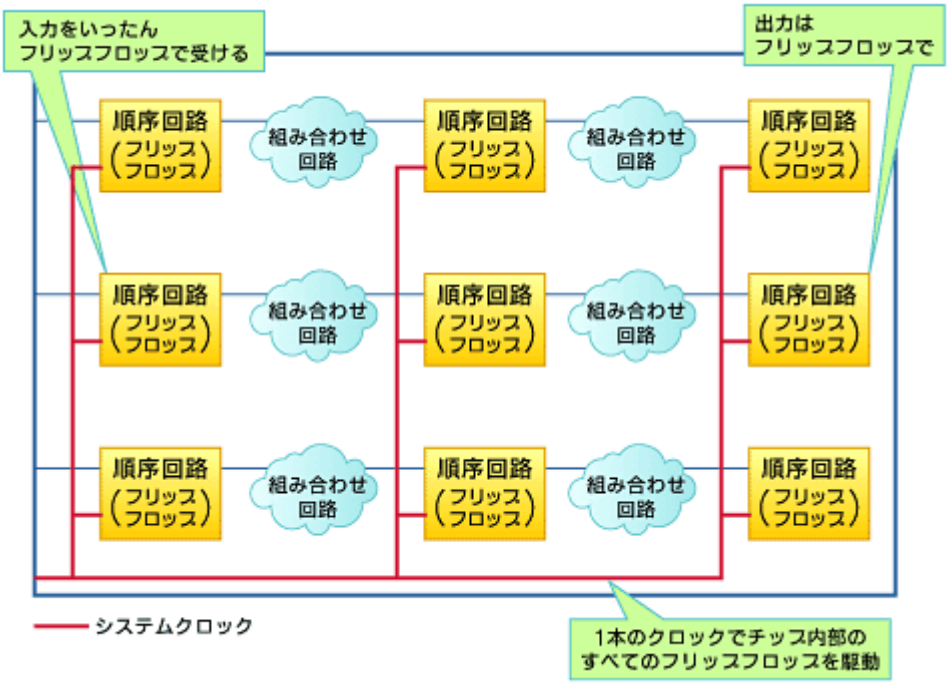


図 2　単相同期回路のイメージ

設計する際に、

「非同期回路じゃないといけない！」

「FPGA 内部はシステムクロックを多重的に使わざるを得ない！」

などの理由があれば別ですが、静的な遅延解析を効率良く行うためには、**“単相同期回路”**で設計するべきです。

これを踏まえて、第 3 回の**リスト 4**、**リスト 5**を見てみると、

```
assign DIVIDE_CLK = tmp_count[22];
always @(posedge DIVIDE_CLK or negedge RESET)     ←システムクロックである CLK 以外の信号がこの順序回路のクロックとして入力されている
begin
        if (RESET == 1'b0)
                COUNT <= 4'h0;
        else
                COUNT <= COUNT + 4'h1;
end
```

上記にコメントしてありますが、システムクロック（ここでは「CLK」）以外の信号（DIVIDE_CLK）が F/F のクロック入力端子に接続しています。これは、上記で説明した内容（“FPGA ツールは単相同期回路が得意”）に反します。例えば、Verilog-HDL を単相同期回路に準じて記述するということは、

```
always @(posedge CLK ……
```

となるように記述しなければなりません。

つまり、単相同期回路ではすべての F/F が「CLK」（この例の場合）という信号名を持つシステムクロックで駆動されるように、できるだけ上記ポリシー（CLK 信号以外は、クロック信号として扱わない）に従って記述する必要があります。こうすることで、静的な遅延解析が行いやすく、パフォー

マンスの良い回路を作ることができます。これが単相同期回路で設計する必要がある最大の理由なのです。

イネーブル方式によるカウンタ記述

それでは、どのようにしたら**前回**作成したカウンタを“単相同期回路の記述スタイル”に合わせることができるでしょうか？

前回の順序回路をクロック信号に乗り換えずに設計するためには、これから解説する**「イネーブル方式」**の回路記述を採用します。

前回作成した「10 進カウンタ」を“1 秒”で動作させて、イネーブル方式による記述スタイルに変更します。具体的には**リスト 1** のように記述します。

```
1  module UPDOWN(RESET, CLK, DEC, COUNT);
2  input RESET, CLK, DEC;
3  output [3:0] COUNT;
4
5  parameter SEC1_MAX = 6000000; // 6MHz
6
7  reg [22:0] tmp_count;
8  reg [3:0] COUNT_TMP;
9  wire ENABLE;
10
11  always @(posedge CLK or negedge RESET)
12  begin
13          if (RESET == 1'b0)
14                  tmp_count <= 23'h000000;
15  //      else
16          else if (ENABLE == 1'b1)
17                  tmp_count <= 23'h000000;
18          else
19                  tmp_count <= tmp_count + 23'h1;
20  end
21
22  // assign DIVIDE_CLK = tmp_count[22];
23  assign ENABLE = (tmp_count == (SEC1_MAX - 1))? 1'b1 : 1'b0;
24
25  //always @(posedge DIVIDE_CLK or negedge RESET)
26  always @(posedge CLK or negedge RESET)
27  begin
28          if (RESET == 1'b0)
29                  COUNT_TMP <= 4'h0;
30          else if (ENABLE == 1'b1)
31  //      else if (DEC == 1'b1)
32                  if (DEC == 1'b1)
33                          if (COUNT_TMP == 4'h9)
34                                  COUNT_TMP <= 4'h0;
35                          else
36                                  COUNT_TMP <= COUNT_TMP + 4'h1;
37                  else
38                          if (COUNT_TMP == 4'h0)
39                                  COUNT_TMP <= 4'h9;
40                          else
41                                  COUNT_TMP <= COUNT_TMP - 4'h1;
42  end
43
44  assign COUNT = ~COUNT_TMP;
45
46  endmodule
```

リスト 1　イネーブル方式による 4bitの 10 進アップ・ダウンカウンタ（UPDOWN10-2.v）

まず、**リスト 1 の 11～20 行目**にあるように、1 秒で動作するカウンタを作成します。具体的には、今回扱う FPGA ボードの 6MHz のクロックを使用していますので、「0～5999999」までカウントするカウンタを作成します。

次に **23 行目**にあるように、1 秒カウンタが「5999999」のときにだけ'1'となるような信号（ENABLE）を作成します（**図 3**）。この記述は、C 言語の条件演算子と同様な記述をします。

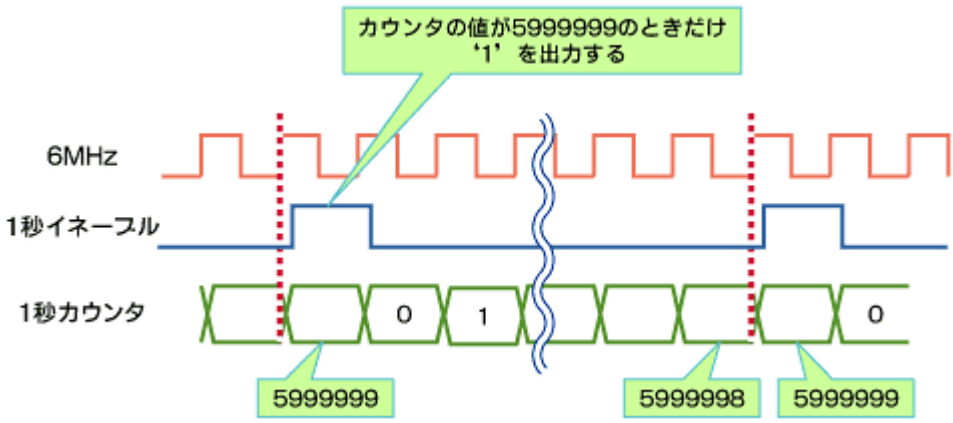


図 3　1 秒イネーブルの作り方

```
assign ENABLE = (tmp_count == (SEC1_MAX - 1))? 1'b1 : 1'b0;
```

- **tmp_count == (SEC1_MAX - 1)が真のとき→1'b1**
- **tmp_count == (SEC1_MAX - 1)が偽のとき→1'b0**

この ENABLE 信号により、tmp_count は「5999999」の次に「0」となります。ちょうど 10 進カウンタのアップカウントで「9」のときに「0」に戻し、ダウンカウントで「0 のときに「9」にしているのと同じ役目をすると考えるとよいでしょう。

なお、この ENABLE 信号の中で使用されている SEC1_MAX は、**5 行目**にあるように、1 秒カウンタの最大値となる値を parameter 文で定義します（この parameter 文を使用するところが、後でこのカウンタをシミュレーションするうえで重要なポイントとなります。詳しくは後述します）。

これで、1 秒カウンタは出来上がりです。次に、この ENABLE 信号を使ってカウンタの動作を 1 秒ごとに行うように変更します。具体的には、**30 行目**のように ENABLE 信号が「1」になったときだけ動作するように、アップ・ダウンするカウント動作の記述部部分を if 文で囲っています。こうすることで、ENABLE が「1」のとき（1 秒に 1 回）だけカウント動作が行われます。

くれぐれも、

```
always @(posedge ENABLE ……
```

などとしないでください。あくまでも、単相同期回路の記述スタイルに持ち込むことが重要なのです。

シミュレーション（1）

第 3 回「順序回路の基本！ カウンタを作成しよう」では、「動作を遅くした 10 進カウンタ」のシミュレーションを実施しませんでした。今回は、きちんと単相同期回路にしたことですし、確実にシミュレーションを行うことにします。

ここで前回使用したテストベンチにちょっと手を加えて（DEC という信号を追加）、シミュレーションしてみましょう（**リスト 2**）。

```
1  module TEST_UPDOWN10;
2  reg clk, reset, dec;
3  wire [3:0] count;
4
5  parameter CYCLE = 100;
6
7  UPDOWN i1(.RESET(reset), .CLK(clk), .DEC(dec), .COUNT(count));
8
9  always #(CYCLE/2)
10          clk = ~clk;
11
12  initial
13  begin
14          reset = 1'b0; clk = 1'b0; dec = 1'b1;
15          #CYCLE reset = 1'b1;
16          #(15*CYCLE) dec = 1'b0;
17          #(10*CYCLE) $finish;
18  end
```

```
19
20  initial
21          $monitor($time,,"clk=%b reset=%b count=%b", clk, reset, count);
22
23  endmodule
```

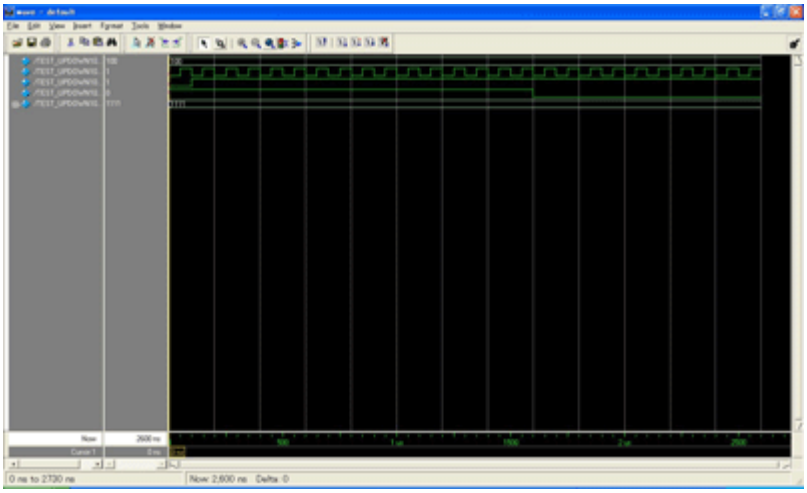
リスト 2　10 進のアップ・ダウンカウンタのテストベンチ　その 1（T_UPDOWN10.v）

テストベンチの流れとしては、

1. **reset を「0」にすることで、10 進カウンタを初期化**
2. **reset を「1」にすることで初期化を解除して、15 回アップカウントさせる（0→9→0→5）**
3. **dec を「0」にすることで、10 回ダウンカウントさせる（5→0→9→5）**

となります。一番検証したいアップカウント時の「9」→「0」の変化とダウンカウント時「0」→「9」の変化を確認できるはずです。

このテストベンチを使用して、シミュレーションを行うと**画面 1** のようになります（カウンタの出力は LED が負論理であることを考慮して、反転して出力されます）。



画面 1　波形表示によるシミュレーション結果の確認

画面 1 にもあるようにリセットが正常に動作して、最初のカウンタの値を「0」（**画面 1** では 1111）にしていますが、その後はまったく動作していません。なぜでしょうか？

それは、10 進のアップ・ダウンカウンタを 1 秒動作にしたので、6000000 クロック進まないと、この 10 進のアップ・ダウンカウンタが動作しないからです。つまり、クロック数が足りていないということです。

単純に「分かった！ クロック数を増やせば解決だ」とは決して思わないでください。これくらい小さい回路ですから、検証に必要なサイクル数（6000000 クロック×（1＋15＋10）＝156000000 クロック）を行っても何とかシミュレーションできるかもしれません。しかし、このクロック数をまともにシミュレーションするとなるとかなり非効率的です。

ここでは、まともに 1 秒のシミュレーションは行わないで「ENABLE 信号がきちんと役目を果たしているのか？」そこに焦点を当ててシミュレーション行います。

ここでポイントになるのが、前述した 10 進のアップ・ダウンカウンタの RTL 記述内で使用した parameter 文です。この parameter の値を減らせば、現実的な時間で 10 進のアップ・ダウンカウンタを動作できます。ここで、「そんなの RTL 記述の parameter 文を（以下のように）変更すれば簡単にできるよ」と思った方もいらっしゃるでしょう。

```
//parameter SEC1_MAX = 6000000; // 6MHz

parameter SEC1_MAX = 4; // For Simulation
```

確かにこうすればシミュレーションのときには下の parameter 文を使用し、FPGA に回路を作成するときにはコメントアウトされている 6MHz 用の parameter を使用すればいいことになります。しかし、この方法では目的に応じて書き換えなければならないので、間違いが生じる（誤ってシミュレーション用の parameter 文を使用するなど）可能性が高くなります。それではあまり効率的とはいえません。

シミュレーション（2）

そこで、parameter の値の受け渡し機能（Verilog-HDL の機能）を使用します。この方法を用いれば、RTL 記述に手を加えることなくシミュレーションを実行できます。

具体的には、**リスト 3 の 8 行目**にあるように、テストベンチの「10 進のアップ・ダウンカウンタ」をインスタンスしている部分に渡したい parameter の値を記述します。

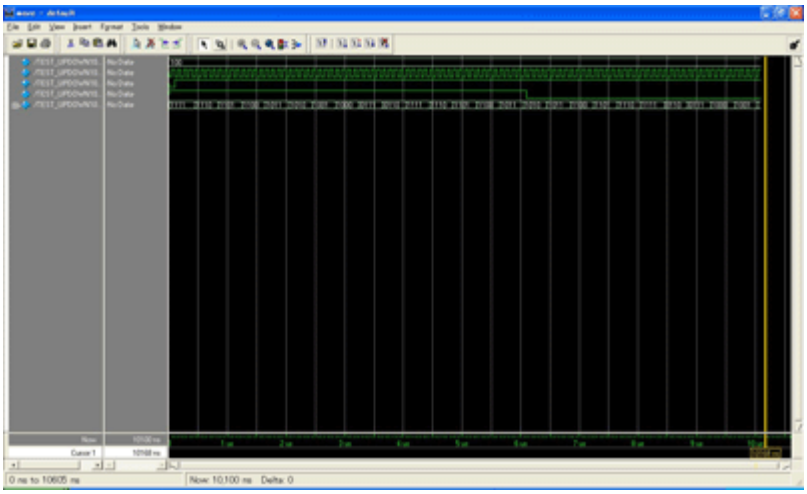
```
1  module TEST_UPDOWN10;
2  reg clk, reset, dec;
3  wire [3:0] count;
```

```
4
5  parameter CYCLE = 100;
6  parameter SIM_SEC1_MAX = 4;
7
8  UPDOWN #(.SEC1_MAX(SIM_SEC1_MAX)) i1(.RESET(reset), .CLK(clk), .DEC(dec), .COUNT(count));
9
10 always #(CYCLE/2)
11          clk = ~clk;
12
13  initial
14  begin
15          reset = 1'b0; clk = 1'b0; dec = 1'b1;
16          #CYCLE reset = 1'b1;
17          #(15*CYCLE*SIM_SEC1_MAX) dec = 1'b0;
18          #(10*CYCLE*SIM_SEC1_MAX) $finish;
19  end
20
21  initial
22          $monitor($time,,"clk=%b reset=%b count=%b", clk, reset, count);
23
24  endmodule
```

リスト 3　10 進のアップ・ダウンカウンタのテストベンチ　その 2（T_UPDOWN10-2.v）

6 行目のように記述することで、10 進のアップ・ダウンカウンタ内で使用されている parameter の値が「4」として扱われて、シミュレーションが実行されます。

インスタンスしたポートと信号を接続するイメージで記述すれば大丈夫です。このテストベンチと**リスト 1**のRTL記述を使用してシミュレーションを行います。この結果を**画面 2** に示します。



画面 2　波形表示によるシミュレーション結果の確認

いかがでしょうか？ 見事に 4 クロックに 1 回、カウンタの値が変化している様子が分かります。このようにシミュレータ上では**“1 秒の動作を確認する”**のではなく、**“ENABLE 信号が正しく 10 進のアップ・ダウンカウンタに効いているか”**を検証するのです。1 秒動作の確認は実機で行えばよいのです。

このように検証する項目を分けて、効率的に RTL の動作を確認することがとても重要です。

FPGA ボード上での動作確認

それでは、出来上がった 10 進のアップ・ダウンカウンタを FPGA 上で動作させてみましょう。

テストベンチでparameterの値を渡したので、**リスト 1**のRTL記述を特に変更する必要はありません。論理合成・配置配線を行って、FPGA上にダウンロードします。

注：「ISE WebPACK」などツールの使用方法は、**第 1 回　理論より実践！　FPGA開発をスタートしよう**を参考にしてください。

見事に単相同期回路の方式を採用して、LED が 1 秒でアップ・ダウンの動作をしているはずです。

◇

今回は、「単相同期回路」と「1 秒で動作するようなカウンタをいかにしてシミュレーションするか」について解説しました。実践でも役に立つ重要な内容となります。特に単相同期回路に慣れていな

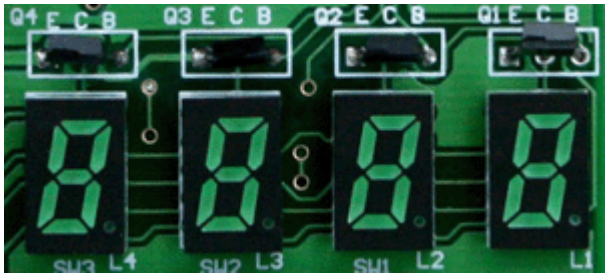


写真 1　7 セグメント LED

い設計者の方は、この記述スタイルや考え方をしっかりとマスターしてください。高速・複雑な設計になればなるほど必要な技術要素になるはずです。

さて、次回は“10 進カウンタの出力を 7 セグメント LED に出力”させます。すでに 10 進のアップ・ダウンカウンタとデコーダが完成していますので、これらをどのように接続して記述するのかを中心に解説します。（次回に続く）

関連記事：
強気のザイリンクス「2010 年にはASICに追い付く」
CPLDからFPGA、ASICまでそろうアルテラ

階層構造を意識した設計スタイルとは？

触って学ぼう FPGA 開発入門（5）
階層構造を意識した設計スタイルとは？

鳥海 佳孝　設計アナリスト　2007/5/18

本連載は、「これから FPGA を開発してみよう！」という入門者の方や仕事で FPGA の設計をされている方（特に新人の方）を対象にしています。また「理論より実践」を主眼とし、入手しやすい無償開発ツールと低価格の FPGA ボードを題材に解説していきます。実際にツールとトレーニングボードを動かして楽しみながら学んでいきましょう。（編集部）

連載第 4 回「単相同期回路で設計する理由」では、いわゆる**“単相同期回路”**を中心に解説しました。単相同期回路の考え方に基づいた設計をするには慣れが必要だと思いますが、現在のトレンドとなる設計手法ですし、設計ツールが基本的に単相同期回路を前提に作られていますから、ぜひこの設計スタイルをマスターしてください。

さて、**これまでの連載**で「7 セグメントLEDのデコーダ」と「10 進のアップ・ダウンカウンタ」を作成しました。今回は、これらを接続してFPGAボード上に実現してみましょう。また、生成された回路の評価を論理合成ツールのログで見てみましょう。

関連記事：
いまさら聞けない FPGA入門
連載：触って学ぼう FPGA開発入門

7 セグメント LED デコーダの修正

連載第 2 回「論理シミュレーションを行う癖を付けよう」で作成した 7 セグメント LEDのデコーダは、スイッチが 3bit入力でした。今回は、**図 1** のような接続構造になるので、デコーダの入力は 4bitになります。

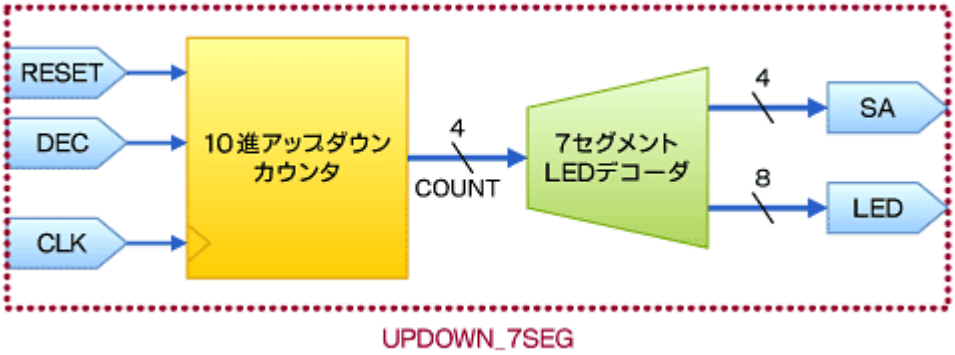


図 1　10 進アップ・ダウンカウンタ＋7 セグメント LED デコーダ

これにより、連載第 2 回で作成した 7 セグメント LED デコーダの RTL 記述を変更する必要があります。変更した RTL 記述を**リスト 1** に示します。

```
1   module DECODER7(COUNT,LED,SA);
2   input [3:0] COUNT;
3   output [7:0] LED;
4   output [3:0] SA;
5   reg [7:0] LED;
6
7   assign SA = 4'bzzz0;
8
9   always @(COUNT)
10  begin
11   case(COUNT)          //ABCDEFG Dp
12        4'b0000:LED <= 8'b0000001_1;
13        4'b0001:LED <= 8'b1001111_1;
14        4'b0010:LED <= 8'b0010010_1;
15        4'b0011:LED <= 8'b0000110_1;
16        4'b0100:LED <= 8'b1001100_1;
17        4'b0101:LED <= 8'b0100100_1;
18        4'b0110:LED <= 8'b0100000_1;
```

```
19          4'b0111:LED <= 8'b0001101_1;
20          4'b1000:LED <= 8'b0000000_1;
21          4'b1001:LED <= 8'b0000100_1;
22          default:LED <= 8'b0110000_1;
23    endcase
24  end
25  endmodule
```

リスト 1　修正した 7 セグメントLEDのデコーダ記述（DECODER7.v）

主な変更点は、以下のとおりです。

1～2 行目

入力が 4bit に変更になったので、1bit ずつバラバラで入力していたものをベクター記述（[3:0]）に変更し、入力の名前も「COUNT」にします。

9～11 行目

デコーダの入力、つまり always 文のセンシティビティ・リストと case 文の「()」の部分を「COUNT」に変更します。また、今回はカウンタからの出力が正論理となるため、反転の演算子「~」は使用しません。

20～21 行目

0～9 の表示となるため、「8」と「9」の表示部分を追加しました。

以上で、7 セグメント LED デコーダの修正が完了しました。

10 進アップ・ダウンカウンタの修正

続いて、**連載第 4 回「単相同期回路で設計する理由」**で作成した 10 進アップ・ダウンカウンタです。連載第 4 回では、FPGAボード上のLEDが負論理なのでカウンタの値を反転させて出力させていました。しかし、今回は前述のように 7 セグメントLEDのデコーダの入力を正論理で扱っているため、その部分を修正します。

修正した記述を以下（**リスト 2**）に示します。

```
1   module UPDOWN(RESET, CLK, DEC, COUNT);
2   input RESET, CLK, DEC;
3   output [3:0] COUNT;
4
5   parameter SEC1_MAX = 6000000; // 6MHz
6
7   reg [22:0] tmp_count;
8   reg [3:0] COUNT_TMP;
9   wire ENABLE;
10
11  always @(posedge CLK or negedge RESET)
12  begin
13          if (RESET == 1'b0)
14                  tmp_count <= 23'h000000;
15  //      else
16          else if (ENABLE == 1'b1)
17                  tmp_count <= 23'h000000;
18          else
19                  tmp_count <= tmp_count + 23'h1;
20  end
21
22  // assign DIVIDE_CLK = tmp_count[22];
23  assign ENABLE = (tmp_count == (SEC1_MAX - 1))? 1'b1 : 1'b0;
24
25  //always @(posedge DIVIDE_CLK or negedge RESET)
26  always @(posedge CLK or negedge RESET)
27  begin
28          if (RESET == 1'b0)
29             COUNT_TMP <= 4'h0;
30          else if (ENABLE == 1'b1)
31  //      else if (DEC == 1'b1)
32                  if (DEC == 1'b1)
```

```
33                          if (COUNT_TMP == 4'h9)
34                                  COUNT_TMP <= 4'h0;
35                          else
36                                  COUNT_TMP <= COUNT_TMP + 4'h1;
37                  else
38                          if (COUNT_TMP == 4'h0)
39                                  COUNT_TMP <= 4'h9;
40                          else
41                                  COUNT_TMP <= COUNT_TMP - 4'h1;
42  end
43
44  assign COUNT = COUNT_TMP;
45
46  endmodule
```

リスト 2　修正した 10 進アップ・ダウンカウンタ記述（UPDOWN10-2.v）

44 行目

カウンタの出力の反転を止め、COUNT に COUNT_TMP そのまま代入する。

以上で、10 進アップ・ダウンカウンタの修正が完了しました。

インスタンスの記述方法

10 進アップ・ダウンカウンタと 7 セグメント LED デコーダを接続します。**リスト 3** のように 1 つのファイルに両方を記述する、つまり階層構造を用いない記述でも構いませんが、ここでは大規模回路で用いられる階層構造を意識した記述方法を用います。

```
1   module UPDOWN_7SEG(RESET, CLK, DEC, LED, SA);
2   input RESET, CLK, DEC;
3   output [7:0] LED;
4   output [3:0] SA;
5
6   parameter SEC1_MAX = 6000000; // 6MHz
7
8   assign SA = 4'bzzz0;
9
10  reg [22:0] tmp_count;
11  reg [3:0] COUNT_TMP;
12  wire ENABLE;
13  reg [7:0] LED;
14
15  always @(posedge CLK or negedge RESET)
16  begin
17      if (RESET == 1'b0)
18          tmp_count <= 23'h000000;
19  //  else
20      else if (ENABLE == 1'b1)
21          tmp_count <= 23'h000000;
22      else
23          tmp_count <= tmp_count + 23'h1;
24  end
25
26  // assign DIVIDE_CLK = tmp_count[22];
27  assign ENABLE = (tmp_count == (SEC1_MAX - 1))? 1'b1 : 1'b0;
28
29  //always @(posedge DIVIDE_CLK or negedge RESET)
30  always @(posedge CLK or negedge RESET)
31  begin
32      if (RESET == 1'b0)
33              COUNT_TMP <= 4'h0;
34      else if (ENABLE == 1'b1)
```

```
35  //  else if (DEC == 1'b1)
36              if (DEC == 1'b1)
37                      if (COUNT_TMP == 4'h9)
38                              COUNT_TMP <= 4'h0;
39                      else
40                              COUNT_TMP <= COUNT_TMP + 4'h1;
41              else
42                      if (COUNT_TMP == 4'h0)
43                              COUNT_TMP <= 4'h9;
44                      else
45                              COUNT_TMP <= COUNT_TMP - 4'h1;
46  end
47
48  always @(COUNT_TMP)
49  begin
50     case(COUNT_TMP)       //ABCDEFG Dp
51          4'b0000:LED <= 8'b0000001_1;
52          4'b0001:LED <= 8'b1001111_1;
53          4'b0010:LED <= 8'b0010010_1;
54          4'b0011:LED <= 8'b0000110_1;
55          4'b0100:LED <= 8'b1001100_1;
56          4'b0101:LED <= 8'b0100100_1;
57          4'b0110:LED <= 8'b0100000_1;
58          4'b0111:LED <= 8'b0001101_1;
59          4'b1000:LED <= 8'b0000000_1;
60          4'b1001:LED <= 8'b0000100_1;
61          default:LED <= 8'b0110000_1;
62     endcase
63  end
64  endmodule
```

リスト 3　階層構造を用いない 10 進アップ・ダウンカウンタ＋7 セグメントLEDデコーダの記述（UPDOWN_7SEG-2.v）

階層構造を用いるといっても、それほど難しいことではありません。単に**図 1**の接続となるように 10 進アップ・ダウンカウンタと 7 セグメントLEDデコーダのモジュールをそれぞれインスタンス（箱を置く）して接続するだけです。インスタンスした記述を**リスト 4** に示します。

```
1   module UPDOWN_7SEG(RESET, CLK, DEC, LED, SA);
2   input RESET, CLK, DEC;
3   output [7:0] LED;
4   output [3:0] SA;
5
6   wire [3:0] COUNT;
7
8   parameter SEC1_MAX = 6000000; // 6MHz
9
10  UPDOWN #(.SEC1_MAX(SEC1_MAX)) i0(.RESET(RESET), .CLK(CLK), .DEC(DEC), .COUNT(COUNT));
11  DECODER7 i1(.COUNT(COUNT), .LED(LED), .SA(SA));
12
13  endmodule
```

リスト 4　10 進アップ・ダウンカウンタ＋7 セグメントLEDでコーダのインスタンス記述（UPDOWN_7SEG.v）

これまで、インスタンスの記述方法に関してあまり詳しく説明してこなかったので、ここで少し解説します。

10、11 行目に 10 進アップ・ダウンカウンタと 7 セグメント LED デコーダのモジュールをそれぞれインスタンスしています。記述方法は以下のとおりです。

```
UPDOWN #(.SEC1_MAX(SEC1_MAX)) i0(.RESET(RESET), .CLK(CLK), .DEC(DEC), .COUNT(COUNT));
DECODER7 i0(.COUNT(COUNT), .LED(LED), .SA(SA));
```

↑

同じインスタンス名を使用してはいけない

また、この例ではドット付きの（例えば.RESET）パラメータ名、ポート名と「()」内の接続したいパラメータ、信号名の名前を一致させていますが、必ずしも同じである必要はありません。

さらに、このインスタンスの記述部分が RTL の最上位記述となるため、少し冗長に見えますが、parameter 文で再度 SEC1_MAX を 6000000 に指定して、下位の 10 進アップ・ダウンカウンタにそのパラメータの値を渡しています。

defparam構文で階層的にパラメータを渡すことができる方法もありますが、論理合成ツールによってはこのパラメータを階層的に渡すという記述方法に対応していないものもありますので、ここでは**リスト 4**の記述を採用しました。

```
モジュール名 #(.モジュール内のパラメータ名(渡したいパラメータ), ……)
        インスタンス名(.モジュールのポート名(接続したい信号名), ……);
```

この記述で特に気を付けなければいけないのが、インスタンス名です。任意の名前を付けられますが、同じモジュール内ではユニークな名前である必要があります。つまり、同じインスタンス名を用いることができないという文法的なルールがあるのです。

以下のように、モジュール名が違っているからといって、同じ「i0」というインスタンス名を使用してはいけません。

テストベンチ記述の修正

テストベンチは、10 進アップ・ダウンカウンタで使用したものを流用します。RTL 記述からの出力が「COUNT」から「LED」「SA」に変更されたので、その部分を変更します。変更した記述を**リスト 5** に示します。

```
1   module TEST_UPDOWN10;
2   reg clk, reset, dec;
3   wire [7:0] led;
4   wire [3:0] sa;
5
6   parameter CYCLE = 100;
7   parameter SIM_SEC1_MAX = 4;
8
9   UPDOWN_7SEG #(.SEC1_MAX(SIM_SEC1_MAX)) i1(.RESET(reset), .CLK(clk), .DEC(dec), .LED(led),
    .SA(sa));
10
11  always #(CYCLE/2)
12      clk = ~clk;
13
14  initial
15  begin
16      reset = 1'b0; clk = 1'b0; dec = 1'b1;
17      #CYCLE reset = 1'b1;
18      #(15*CYCLE*SIM_SEC1_MAX) dec = 1'b0;
19      #(10*CYCLE*SIM_SEC1_MAX) $finish;
20  end
21
22  initial
23      $monitor($time,,"clk=%b reset=%b count=%b", clk, reset, i1.COUNT);
24
25  endmodule
```

リスト 5　テストベンチの記述（T_UPDOWN10-2.v）

変更点は以下のとおりです。

3～4 行目

led と sa のポートに接続するための wire 宣言（複数 bit あるので省略不可）

9 行目

LED と SA ポートに led と sa の信号の接続

23 行目

$monitor でカウンタの値を出力させている部分を、UPDOWN_7SEG モジュールの中の COUNT 信号に変更。具体的には「i1.COUNT」にする。

以上で、テストベンチ記述の修正が完了しました。

シミュレーションの実行

それでは、シミュレーションを実行しましょう！ 基本的にシミュレーションの実行は、いままで解説してきたとおりです。

必要なファイル 4 つ（**T_UPDOWN10-2.v**、**UPDOWN_7SEG.v**、**UPDOWN10-2.v**、**DECODER7.v**）を用います。

今回は、**図 1**にもあるとおり 7 セグメントLEDの結果もさることながら、RTLの最上位に相当する「UPDOWN_7SEG」の内部信号である「COUNT」信号を波形表示させたいところです。

基本的には画面左にある［Workspace］ウィンドウの［sim］タブにある、インスタンス名が出力されている部分「i1」を選択すると、［Objects］ウィンドウに「COUNT」信号が現れますので、これを右クリックしてショートカットメニュー［Add to Wave］－［Selected Signals］を選択し、波形表示［wave］ウィンドウに加えるだけです。以下のように「COUNT」が動作している様子を見ることができます（**図 2-1～2-4**）。

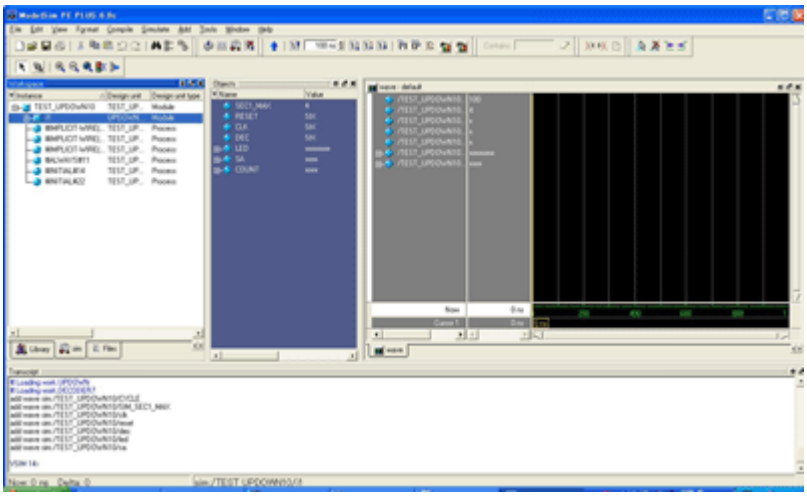


図 2-1　見たい信号のあるインスタンス名を選択

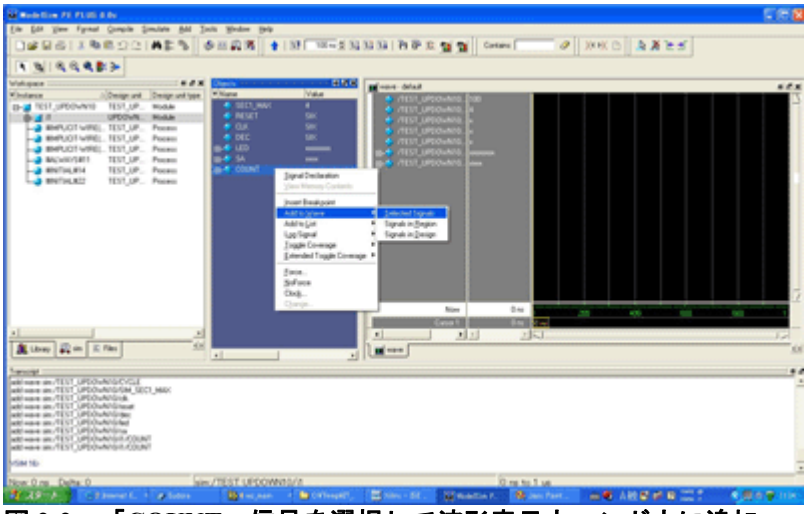


図 2-2　「COUNT」信号を選択して波形表示ウィンドウに追加

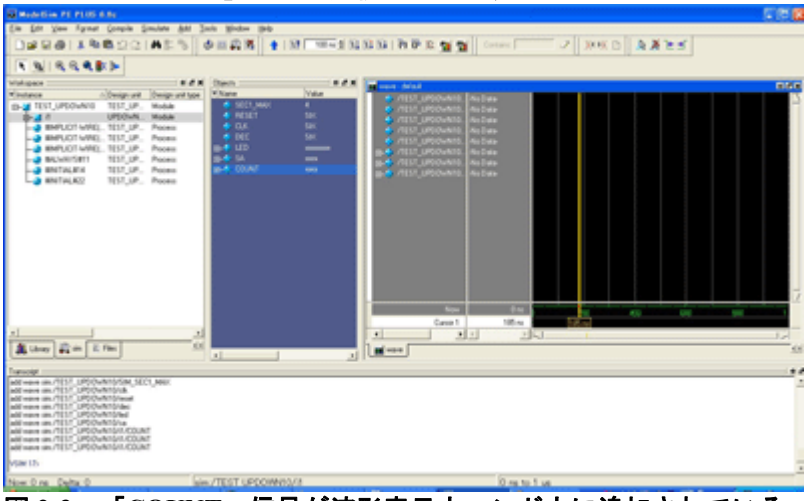


図 2-3　「COUNT」信号が波形表示ウィンドウに追加されている

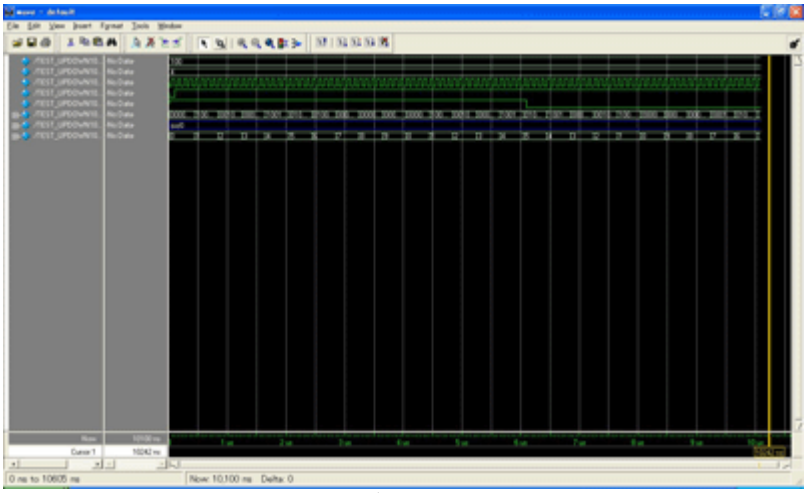


図 2-4　「COUNT」信号の値が表示される

論理合成、配置配線、ダウンロード

論理シミュレーションの結果特に問題がなければ、「ISE WebPACK」で**論理合成**、**配置配線**を行います。セットアップ時の注意点は、ファイルを選択する際に**図 3-1** のようにRTL記述に関連する下位のモジュール（**UPDOWN10-2.v**、**DECODER7.v**、**UPDOWN_7SEG.v**）をすべて選択することです。

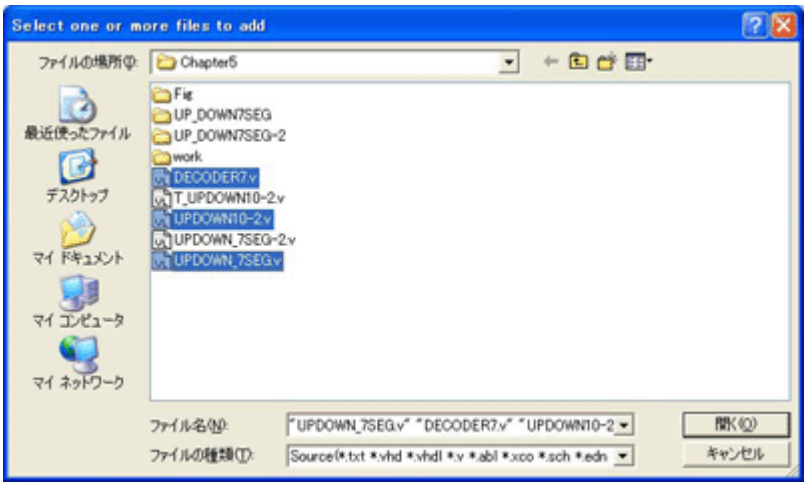


図 3-1　回路作成に必要なファイルをすべて選択する

画面左上の「Sources」ウィンドウに回路に必要なファイル（ピン固定のファイル含む）が、登録されます（**図 3-2**）。

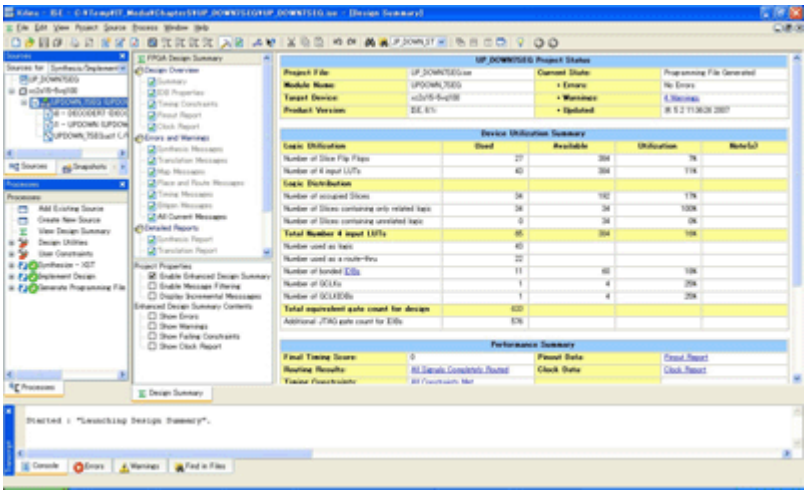


図 3-2　ISE WebPACK への登録（画像をクリックすると拡大します）

最上位のモジュール（**UPDOWN_7SEG.v**）を選択すれば、勝手に下位のモジュールを自動的に呼んできてくれるわけではありませんので注意してください。回路になるモジュールはすべて選択します。また、今回使用するピン固定のファイルを**リスト 6** に示します。

```
1   NET "CLK"  LOC = "P39"  ;
2   NET "RESET"  LOC = "P17"  ;
3   NET "DEC"  LOC = "P16"  ;
4   NET "LED<0>"  LOC = "P41"  ;
5   NET "LED<1>"  LOC = "P40"  ;
```

```
6   NET "LED<2>"  LOC = "P31"  ;
7   NET "LED<3>"  LOC = "P30"  ;
8   NET "LED<4>"  LOC = "P22"  ;
9   NET "LED<5>"  LOC = "P21"  ;
10  NET "LED<6>"  LOC = "P20"  ;
11  NET "LED<7>"  LOC = "P19"  ;
12  NET "SA<0>"  LOC = "P46"  ;
13  NET "SA<1>"  LOC = "P45"  ;
14  NET "SA<2>"  LOC = "P44"  ;
15  NET "SA<3>"  LOC = "P43"  ;
```

リスト 6　ピン固定ファイル（UPDOWN_7SEG.ucf）

後はいままでどおり論理合成、配置配線、FPGA 用のデータ作成を行います。

正常終了したらダウンロードを行い、動作を確認してみましょう。

ここまでの作業が問題なく完了していれば、真ん中のプッシュスイッチを押していない間は、右側の 7 セグメント LED が 0～9 まで 1 秒ごとにカウントアップし、真ん中のプッシュスイッチを押している間は、右側の 7 セグメント LED が 9～0 まで 1 秒ごとにカウントダウンするはずです。

いかがでしょうか？ うまく動いたでしょうか？

論理合成のログの見方

続いて、論理合成のログを見てみましょう。論理合成の結果のログは、**図 4-1** のようにしてレポートを参照します。

画面左中央の「Processes」ウィンドウの「Synthesize － XST」を展開して、「View Synthesis Report」をダブルクリックしてください。

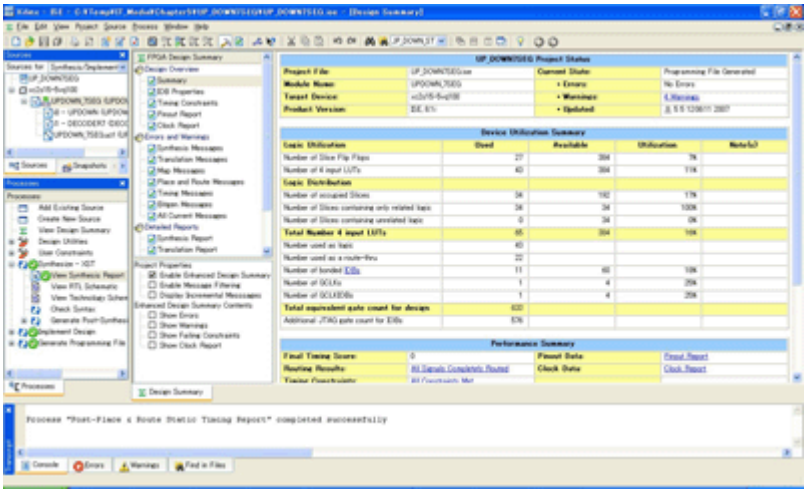


図 4-1　「View Synthesis Report」をダブルクリック（画像をクリックすると拡大します）

図 4-2 のように合成結果のログが表示されます。

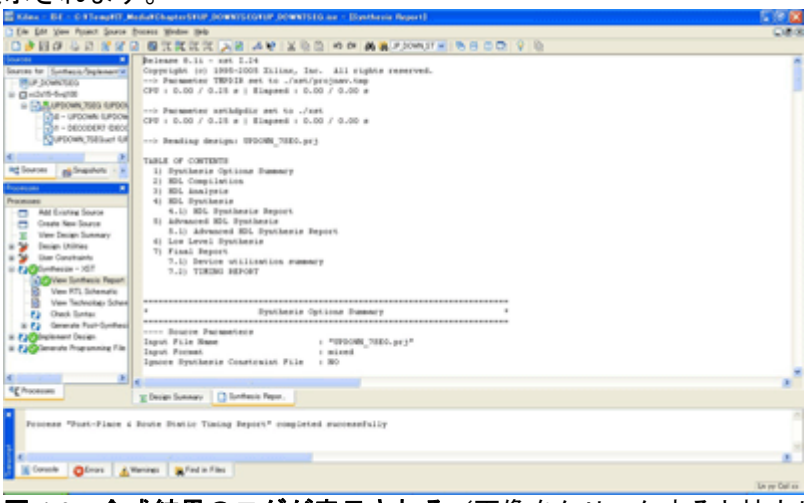


図 4-2　合成結果のログが表示される（画像をクリックすると拡大します）

レポートの内容としては、

1）　使用した合成オプションの一覧
2）　HDL のコンパイル結果
3）　HDL の解析
4）　HDL の合成
4.1）　HDL の合成のレポート
5）　アドバンスド HDL の合成
5.1）　アドバンスド HDL の合成のレポート

6） 下位階層レベル合成
7） 最終レポート
7.1） 使用したデバイスの占有率の一覧
7.2） タイミングレポート
となっています。この中で興味深いところだけをピックアップすると、まずは HDL の合成（HDL Synthesis）の部分です。ここを見てみると、

```
Synthesizing Unit <DECODER7>.
    Related source file is "DECODER7.v".
    Found 16x8-bit ROM for signal <LED>.
    Found 3-bit tristate buffer for signal <SA<3:1>>.
    Summary:
        inferred   1 ROM(s).
        inferred   3 Tristate(s).
Unit <DECODER7> synthesized.

Synthesizing Unit <UPDOWN>.
    Related source file is "UPDOWN10-2.v".
    Found 4-bit 4-to-1 multiplexer for signal <$n0000>.
    Found 4-bit addsub for signal <$n0001>.
    Found 4-bit register for signal <COUNT_TMP>.
    Found 23-bit up counter for signal <tmp_count>.
    Summary:
        inferred   1 Counter(s).
        inferred   4 D-type flip-flop(s).
        inferred   1 Adder/Subtractor(s).
        inferred   4 Multiplexer(s).
Unit <UPDOWN> synthesized.
```

となっていて、デコーダの部分は ROM として推定され、SA[3:1]の信号に関してはトライステートの回路として推定されています。また、カウンタの方ではアップ・ダウンさせるため加減算器（addsub）が推定されており、23bit フリーランのアップカウンタ（tmp_count）が推定されています。
　次に注目したいところは、最終レポートの使用したデバイスの占有率の一覧とタイミングレポートです。占有率としては、

```
Selected Device : 2s15vq100-5

 Number of Slices:                       35  out of    192    18%
 Number of Slice Flip Flops:             27  out of    384     7%
 Number of 4 input LUTs:                 64  out of    384    16%
 Number of bonded IOBs:                  15  out of     64    23%
 Number of GCLKs:                         1  out of      4    25%
```

となっており、この中で特に重要なのは Flip Flop と LUT（Look Up Table）の部分です。ボード上の FPGA デバイスには、384 個のフリップフロップと LUT が載っています。この数を超えてしまうとデバイスに収まらないことになります。上記の結果では、23bit のフリーランのカウンタと 4bit の 10 進カウンタでフリップフロップがそれぞれ使われていますので、23＋4 で確かに 27 個のフリップフロップが使用されていることが確かめられます。
　最後にタイミングレポートですが、以下のようになっています。

```
Timing Detail:
--------------
All values displayed in nanoseconds (ns)

=========================================================================
Timing constraint: Default period analysis for Clock 'CLK'
  Clock period: 9.275ns (frequency: 107.817MHz)
  Total number of paths / destination ports: 918 / 31
-------------------------------------------------------------------------
Delay:               9.275ns (Levels of Logic = 8)
  Source:            i0/tmp_count_8 (FF)
  Destination:       i0/tmp_count_0 (FF)
```

```
  Source Clock:      CLK rising
  Destination Clock: CLK rising

  Data Path: i0/tmp_count_8 to i0/tmp_count_0
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDC:C->Q              2   1.292   1.340  i0/tmp_count_8 (i0/tmp_count_8)
     LUT3_L:I0->LO         1   0.653   0.000  i0/_n0003_wg_sel (N01)
     MUXCY:S->O            1   0.784   0.000  i0/_n0003_wg_cy (i0/_n0003_wg_cy)
     MUXCY:CI->O           1   0.050   0.000  i0/_n0003_wg_cy_rn_0 (i0/_n0003_wg_cy1)
     MUXCY:CI->O           1   0.050   0.000  i0/_n0003_wg_cy_rn_1 (i0/_n0003_wg_cy2)
     MUXCY:CI->O           1   0.050   0.000  i0/_n0003_wg_cy_rn_2 (i0/_n0003_wg_cy3)
     MUXCY:CI->O           1   0.050   0.000  i0/_n0003_wg_cy_rn_3 (i0/_n0003_wg_cy4)
     MUXCY:CI->O          27   0.050   3.550  i0/_n0003_wg_cy_rn_4 (i0/_n0003_wg_cy5)
     LUT2_L:I0->LO         1   0.653   0.000  i0/tmp_count_Eqn_211 (i0/tmp_count_Eqn_21)
     FDC:D                     0.753          i0/tmp_count_21
    ----------------------------------------

    Total                      9.275ns (4.385ns logic, 4.890ns route)
                                       (47.3% logic, 52.7% route)

==========================================================================
Timing constraint: Default OFFSET IN BEFORE for Clock 'CLK'
  Total number of paths / destination ports: 6 / 3
--------------------------------------------------------------------------
Offset:              4.555ns (Levels of Logic = 3)
  Source:            DEC (PAD)
  Destination:       i0/COUNT_TMP_1 (FF)
  Destination Clock: CLK rising

  Data Path: DEC to i0/COUNT_TMP_1
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     IBUF:I->O             6   0.924   1.850  DEC_IBUF (DEC_IBUF)
     LUT4_L:I1->LO         1   0.653   0.000  i0/_n0000<1>1111_F (N511)
     MUXF5:I0->O           1   0.375   0.000  i0/_n0000<1>1111 (i0/_n0000<1>)
     FDCE:D                    0.753          i0/COUNT_TMP_1
    ----------------------------------------
    Total                      4.555ns (2.705ns logic, 1.850ns route)
                                       (59.4% logic, 40.6% route)

==========================================================================
Timing constraint: Default OFFSET OUT AFTER for Clock 'CLK'
  Total number of paths / destination ports: 28 / 7
--------------------------------------------------------------------------
Offset:              11.152ns (Levels of Logic = 2)
  Source:            i0/COUNT_TMP_1 (FF)
  Destination:       LED<7> (PAD)
  Source Clock:      CLK rising

  Data Path: i0/COUNT_TMP_1 to LED<7>
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDCE:C->Q            13   1.292   2.500  i0/COUNT_TMP_1 (i0/COUNT_TMP_1)
     LUT4:I1->O            1   0.653   1.150  Mrom_data_i1/Mrom_LED6 (LED_7_OBUF)
```

```
    OBUF:I->O                     5.557          LED_7_OBUF (LED<7>)
    ----------------------------------------
    Total                        11.152ns (7.502ns logic, 3.650ns route)
                                          (67.3% logic, 32.7% route)
```

3 種類出力されていますが、それぞれ

1）フリップフロップからフリップフロップ間（Default period analysis for Clock 'CLK'）
2）入力ピンからフリップフロップまで（Default OFFSET IN BEFORE for Clock 'CLK'）
3）フリップフロップから出力ピンまで（Default OFFSET OUT AFTER for Clock 'CLK'）

を表しています（**図 5**）。

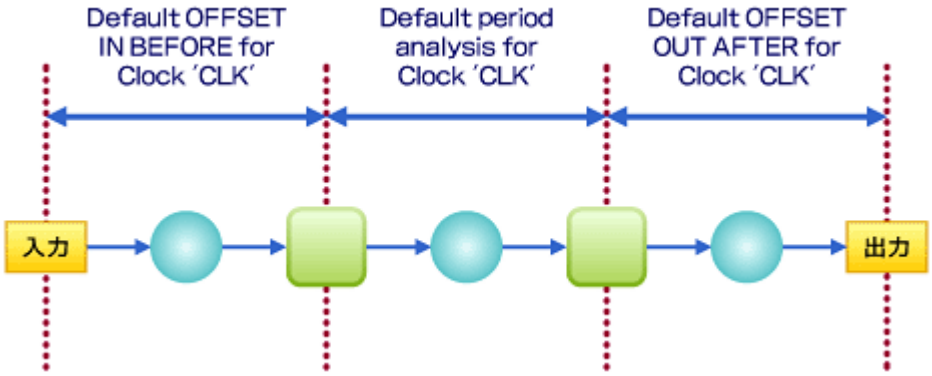


図 5　タイミングレポートで出力される遅延解析の定義
※青丸：組み合わせ回路／緑四角：フリップフロップ

ログには、まだ配置配線していないものの、仮の配線長を想定して一番長い遅延と検出された部分を表示しています。今回はそれぞれ、

1）tmp_count の 8bit 目から tmp_count の 0bit 目の間が 9.275ns かかっている
2）DEC の入力信号から COUNT_TMP の 1bit 目までが 4.555ns かかっている
3）COUNT_TMP の 1bit 目から LED の 7bit 目までが 11.152ns かかっている

ものが最長パスとして検出されています。いずれも 6MHz（約 166ns 以下）で動作させることを考えれば、結果的には十分満足していると考えてよいと思います。

実際の回路設計では、これらの結果を見ながらパフォーマンスが上がるように HDL を修正したり、制約条件（どのくらいのスピードで動作してほしいかの定義など）を駆使して作業を進めています。

◇

今回は、階層構造を利用して 10 進のアップ・ダウンカウンタの動作を 7 セグメント LED に出力しました。

回路が大規模化してくると、1 つのモジュールだけで設計することが困難になってくるので、今回のような階層構造を保って設計することが必要となります。このくらいの規模だと、わざわざ階層構造にするのはちょっと面倒に感じるかもしれませんが、ぜひこの設計スタイルに慣れておく方がよいでしょう。

さて、次回はいよいよ最終回となります。ここまでで 10 進のアップ・ダウンカウンタが作成できたので、次は 60 進のアップ・ダウンカウンタを作成します。かなり設計のエッセンスが含まれた題材となりますので、そのエッセンスを中心にお話ししたいと思います。（次回に続く）

関連記事：
強気のザイリンクス「2010 年にはASICに追い付く」
CPLDからFPGA、ASICまでそろうアルテラ

順序回路と組み合わせ回路を意識した記述を！

触って学ぼう FPGA 開発入門（6）
順序回路と組み合わせ回路を意識した記述を！

鳥海 佳孝　設計アナリスト　2007/6/15

本連載は、「これから FPGA を開発してみよう！」という入門者の方や仕事で FPGA の設計をされている方（特に新人の方）を対象にしています。また「理論より実践」を主眼とし、入手しやすい無償開発ツールと低価格の FPGA ボードを題材に解説していきます。実際にツールとトレーニングボードを動かして楽しみながら学んでいきましょう。（編集部）

連載第 5 回「階層構造を意識した設計スタイルとは？」では、10 進のアップ・ダウンカウンタと 7 セグメントLEDのデコーダのモジュールをそれぞれインスタンスして接続し、動作させました。

最終回となる今回は、7 セグメント LED を 2 つ使用して、60 進のアップ・ダウンカウンタを作成します。「それほど難しくないだろう」と思われるかもしれませんが、実際に何の情報もないと結構手間取ります。今回の内容は、この連載で一番強調したかった**“HDL によるハードウェア設計のエッセンス”**が盛り込まれていますので期待してください。

関連記事：
いまさら聞けない FPGA入門
連載：触って学ぼう FPGA開発入門

ダイナミック点灯の原理

今回使用している「EDX-002」は、ボード上での配線の引き回しを少なくするために、7 セグメント LEDの点灯に“ダイナミック点灯”の手法を使用しています。具体的な原理は**図 1** のとおりです。

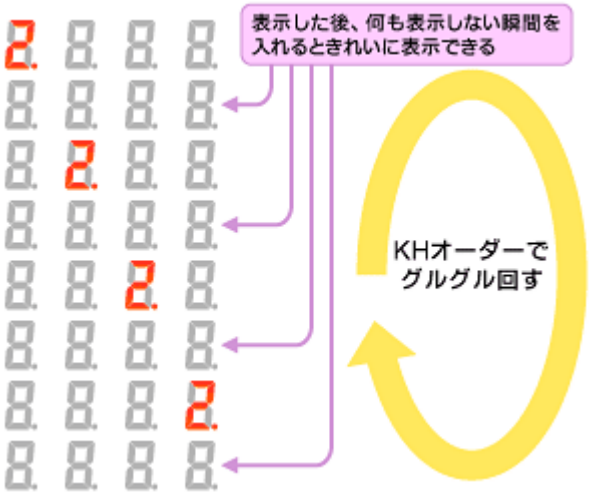


図 1　ダイナミック点灯の原理

「点灯させたい 7 セグメント LED の選択を行う信号（SA）を出力し、その選択された 7 セグメント LED に表示したい値を出力する」というのが基本動作です。また、EDX-002 の仕様では選択されていない 7 セグメント LED にはハイ・インピーダンスの信号を割り当てることになっています。

この一連の動作を**リスト 1** に示します。

```
1   module DCOUNT (CLK, ENABLE, L1, L2, L3, L4, SA, L);
2   input CLK, ENABLE;
3   input [7:0] L1, L2, L3, L4;
4   output [3:0] SA;
5   output [7:0] L;
6
7   parameter MAX_COUNT = 3'b111;
8   reg [2:0] sa_count_tmp;
9   reg [3:0] sa_count;
10  reg [7:0] L_tmp;
11
12  assign SA[3] = (sa_count[3]==1'b0)? 1'b0 : 1'bz;
13  assign SA[2] = (sa_count[2]==1'b0)? 1'b0 : 1'bz;
14  assign SA[1] = (sa_count[1]==1'b0)? 1'b0 : 1'bz;
15  assign SA[0] = (sa_count[0]==1'b0)? 1'b0 : 1'bz;
16  assign L = L_tmp;
17
18  always @(posedge CLK)
19  begin
20      if (ENABLE==1'b1)
```

```
21          if (sa_count_tmp==MAX_COUNT)
22              sa_count_tmp <= 3'b000;
23          else
24              sa_count_tmp <= sa_count_tmp + 1'b1;
25  end
26
27  always @(posedge CLK)
28  begin
29      if (sa_count_tmp[0]==1'b0)
30          begin
31             sa_count <= 4'b1111;L_tmp <= L_tmp;
32          end
33      else
34          case (sa_count_tmp[2:1])
35                2'b00:begin
36                          sa_count <= 4'b1110;L_tmp <= L4;
37                        end
38                2'b01:begin
39                          sa_count <= 4'b1101;L_tmp <= L3;
40                        end
41                2'b10:begin
42                          sa_count <= 4'b1011;L_tmp <= L2;
43                        end
44                2'b11:begin
45                          sa_count <= 4'b0111;L_tmp <= L1;
46                        end
47              default:begin
48                          sa_count <= 4'bxxxx;L_tmp <= 8'bxxxxxxxx;
49                      end
50          endcase
51  end
52
53  endmodule
```

リスト 1　ダイナミック点灯させるためのモジュール（dcount.v）

18～25 行目

sa_count_tmp という 3bit のカウンタで 0～7 までカウント。このときに ENABLE 信号（kHz オーダーで 1 回有効になる信号）が‘1’のときにのみカウントアップ

27～51 行目

このカウンタの値が奇数（sa_count_tmp[0]が‘1’）のときに、出力したい 7 セグメント LED のデコーダの信号を選択

12～15 行目

選択していない 7 セグメント LED にはハイ・インピーダンスを出力

今回の 7 セグメント LED の 2 けたの点灯には、このモジュール（**リスト 1**）を用います。

回路構成の検討

まず、60 進カウンタを設計するに当たり、どのようにこのカウンタを構成するのかを考えてみましょう。

単純に考えると「0～59」、すなわち 60 回カウントすればいいので、「6bit のカウンタを作成すれば簡単に実現できる！ 」と思い付きます。

しかし、最終的に 7 セグメント LED に出力するためには、10 の位と 1 の位に 6bit のカウンタの値を分けなくてはなりません。

分ける方法はいくつかありますが、ソフトウェア開発が得意な方の場合、

- **1 の位：60 進カウンタ%10（10 で割った余り）**
- **10 の位：60 進カウンタ/10（10 で割った商）**

で、求めることを思い浮かべるのではないでしょうか？ 確かに、アルゴリズム的には正しいのですが、残念ながら上記のような「%」や「/」の使い方は、論理合成ツールがサポートしていません。つまり、

回路が作成できないので、RTL ではないということになります。ということで、この方法は「使用不可」です。

気を取り直して、さらに考えてみましょう。

次に思い付くのは、「6bit のカウンタの値を 10 の位と 1 の位にデコードして分ける」という方法です（**リスト 2**）。

```
（省略）
8   always @(count60)
9       case (count60)
10              6'd0:{cnt10,cnt1}=7'h0_0;
11              6'd1:{cnt10,cnt1}=7'h0_1;
12              6'd2:{cnt10,cnt1}=7'h0_2;
13              6'd3:{cnt10,cnt1}=7'h0_3;
14              6'd4:{cnt10,cnt1}=7'h0_4;
15              6'd5:{cnt10,cnt1}=7'h0_5;
16              6'd6:{cnt10,cnt1}=7'h0_6;
17              6'd7:{cnt10,cnt1}=7'h0_7;
18              6'd8:{cnt10,cnt1}=7'h0_8;
19              6'd9:{cnt10,cnt1}=7'h0_9;
20              6'd10:{cnt10,cnt1}=7'h1_0;
21              6'd11:{cnt10,cnt1}=7'h1_1;
22              6'd12:{cnt10,cnt1}=7'h1_2;
23              6'd13:{cnt10,cnt1}=7'h1_3;
24              6'd14:{cnt10,cnt1}=7'h1_4;
25              6'd15:{cnt10,cnt1}=7'h1_5;
26              6'd16:{cnt10,cnt1}=7'h1_6;
27              6'd17:{cnt10,cnt1}=7'h1_7;
28              6'd18:{cnt10,cnt1}=7'h1_8;
29              6'd19:{cnt10,cnt1}=7'h1_9;
30              6'd20:{cnt10,cnt1}=7'h2_0;
31              6'd21:{cnt10,cnt1}=7'h2_1;
32              6'd22:{cnt10,cnt1}=7'h2_2;
33              6'd23:{cnt10,cnt1}=7'h2_3;
34              6'd24:{cnt10,cnt1}=7'h2_4;
35              6'd25:{cnt10,cnt1}=7'h2_5;
36              6'd26:{cnt10,cnt1}=7'h2_6;
37              6'd27:{cnt10,cnt1}=7'h2_7;
38              6'd28:{cnt10,cnt1}=7'h2_8;
39              6'd29:{cnt10,cnt1}=7'h2_9;
40              6'd30:{cnt10,cnt1}=7'h3_0;
41              6'd31:{cnt10,cnt1}=7'h3_1;
42              6'd32:{cnt10,cnt1}=7'h3_2;
43              6'd33:{cnt10,cnt1}=7'h3_3;
44              6'd34:{cnt10,cnt1}=7'h3_4;
45              6'd35:{cnt10,cnt1}=7'h3_5;
46              6'd36:{cnt10,cnt1}=7'h3_6;
47              6'd37:{cnt10,cnt1}=7'h3_7;
48              6'd38:{cnt10,cnt1}=7'h3_8;
49              6'd39:{cnt10,cnt1}=7'h3_9;
50              6'd40:{cnt10,cnt1}=7'h4_0;
51              6'd41:{cnt10,cnt1}=7'h4_1;
52              6'd42:{cnt10,cnt1}=7'h4_2;
53              6'd43:{cnt10,cnt1}=7'h4_3;
54              6'd44:{cnt10,cnt1}=7'h4_4;
55              6'd45:{cnt10,cnt1}=7'h4_5;
56              6'd46:{cnt10,cnt1}=7'h4_6;
57              6'd47:{cnt10,cnt1}=7'h4_7;
58              6'd48:{cnt10,cnt1}=7'h4_8;
59              6'd49:{cnt10,cnt1}=7'h4_9;
60              6'd50:{cnt10,cnt1}=7'h5_0;
61              6'd51:{cnt10,cnt1}=7'h5_1;
62              6'd52:{cnt10,cnt1}=7'h5_2;
63              6'd53:{cnt10,cnt1}=7'h5_3;
64              6'd54:{cnt10,cnt1}=7'h5_4;
65              6'd55:{cnt10,cnt1}=7'h5_5;
66              6'd56:{cnt10,cnt1}=7'h5_6;
67              6'd57:{cnt10,cnt1}=7'h5_7;
68              6'd58:{cnt10,cnt1}=7'h5_8;
69              6'd59:{cnt10,cnt1}=7'h5_9;
70          default:{cnt10,cnt1}=7'hx;
71      endcase
（省略）
```

リスト 2　10 の位と 1 の位分離デコーダ部分（cnt60_dec.v）

しかし、この方法では**図 2** のように 7 セグメント LED のデコーダの前に 1 の位と 10 の位を分けるデコーダが入るので、回路が大きくなり、LED の出力まで考えるとい ままでのデコーダ出力よりも遅くなってしまいます（今回のような LED 表示に関しては、本来それほど神経をとがらせる必要はありません）。つまり、今回の場合には 60 進カウンタを作成するという方法は、あまり得策とはいえません。

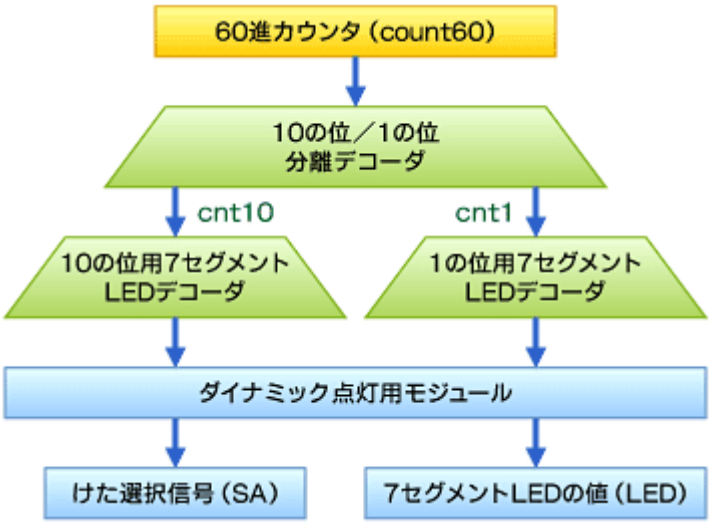


図 2　60 進カウンタを作成したときのハードウェア構成

そこで、いちいち 10 の位と 1 の位を分けるデコーダを作成することがないように、カウンタをあらかじめ「10 進カウンタ」と「6 進カウンタ」に分けて設計します（**図 3**）。こうすれば、いままでと同様にカウンタの出力をそのまま 7 セグメント LED のデコーダに接続できます。

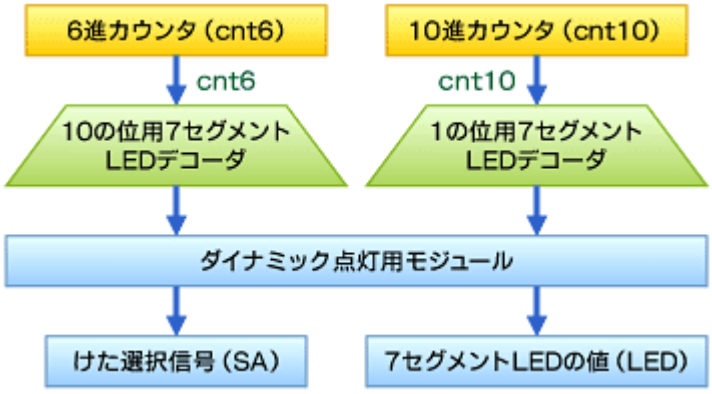


図 3　10 進と 6 進カウンタを作成したときのハードウェア構成

キャリーによるけた上げ

回路構成が決まったら早速設計に入ります。ここでは、本連載の集大成ともいうべきエッセンスを紹介します。

この「10 進のカウンタ」と「6 進のカウンタ」を分けて設計したときの最大のポイントは、10 進カウンタから 6 進カウンタへのけた上げ信号、つまり「キャリー」をいかに記述するかです。キャリーが**図 4** のようにアップカウントする場合は、10 進カウンタの値が‘9’になったときです。

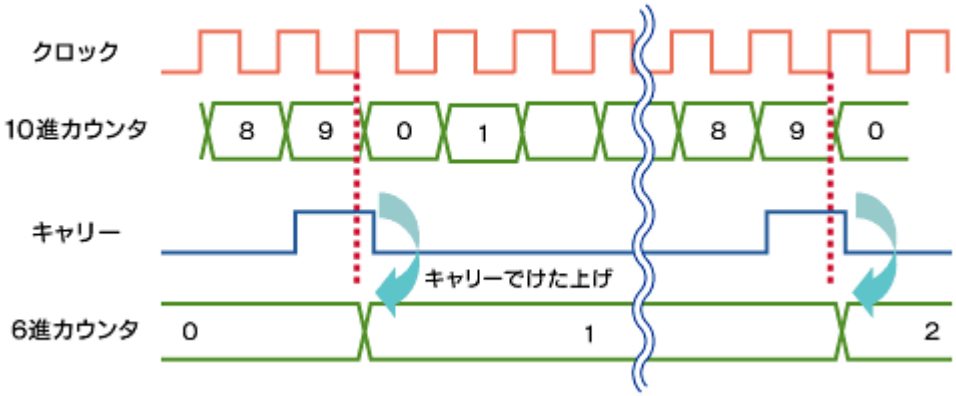


図 4　キャリーによるけた上げ

この考えから「キャリー信号の代入は“if (COUNT_TMP == 4'h9)”となっているところで行えば良い！ 」と思うでしょう。つまり、**前回のリスト 2**に変更を加えて、**リスト 3** のような記述をイメージすると思います。

（省略）

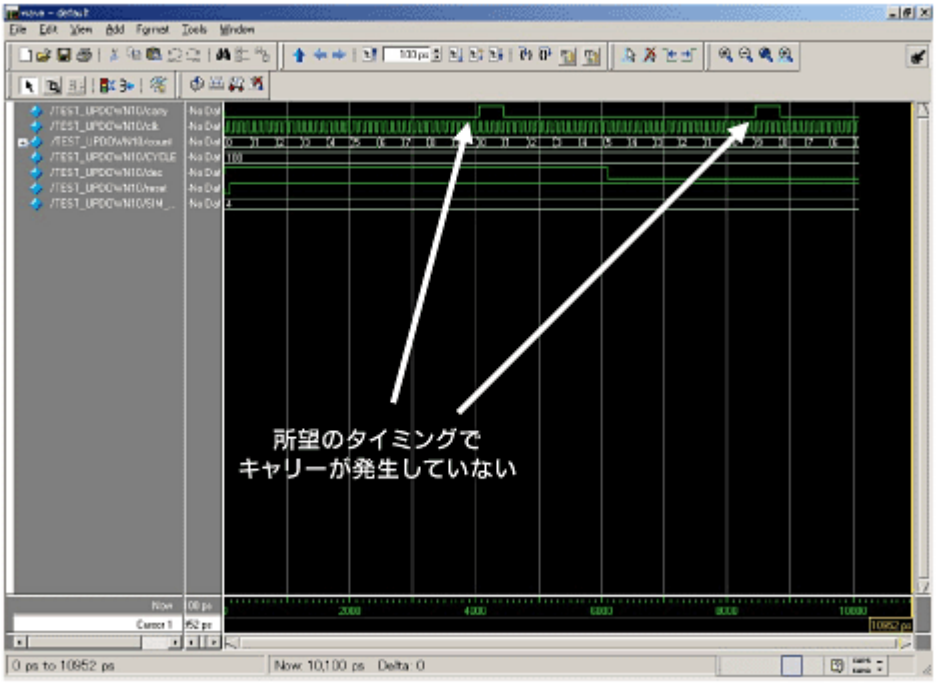
```
 28 always @(posedge CLK or negedge RESET)
 29 begin
 30         if (RESET == 1'b0)
 31             begin
 32                 COUNT_TMP <= 4'h0;
 33                 CARRY <= 1'b0;
 34             end
 35         else if (ENABLE == 1'b1)
 36 //       else if (DEC == 1'b1)
 37                     if (DEC == 1'b1)
 38                         if (COUNT_TMP == 4'h9)
 39                             begin
 40                                 COUNT_TMP <= 4'h0;
 41                                 CARRY <= 1'b1;
 42                             end
 43                         else
 44                             begin
 45                                 COUNT_TMP <= COUNT_TMP + 4'h1;
 46                                 CARRY <= 1'b0;
 47                             end
 48                     else
 49                         if (COUNT_TMP == 4'h0)
 50                             begin
```

```
51                              COUNT_TMP <= 4'h9;
52                              CARRY <= 1'b1;
53                          end
54                      else
55                          begin
56                              COUNT_TMP <= COUNT_TMP - 4'h1;
57                              CARRY <= 1'b0;
58                          end
59 end
```

（省略）

リスト 3　不正なキャリーの作り方（UPDOWN10.v）

一見するとこれで良さそうに思えますが、これが落とし穴の始まりです……。ここにキャリーの代入を書いて、シミュレーションしてみると**画面 1** のようになります。



画面 1　不正なタイミングでのキャリーによるけた上げ

アップカウントの値が‘0’のときと、ダウンカウントの値が‘9’のときににキャリーが出力されています……。ここで「なぜこうなるんだろう？」と追究していただければよいのですが、「じゃあ、10 進のカウンタの値を‘9’で見ているのが悪いので、‘8’で見れば OK だろう！」と、**リスト 4** のように力ずくで記述を変更すると**画面 2** のような結果になってしまいます。

（省略）

```
28 always @(posedge CLK or negedge RESET)
29 begin
30         if (RESET == 1'b0)
31             begin
32                 COUNT_TMP <= 4'h0;
33                 CARRY <= 1'b0;
34             end
35         else if (ENABLE == 1'b1)
36 //          else if (DEC == 1'b1)
37                     if (DEC == 1'b1)
38                         if (COUNT_TMP == 4'h8)
39                             begin
40                                 COUNT_TMP <= 4'h0;
41                                 CARRY <= 1'b1;
42                             end
43                         else
44                             begin
45                                 COUNT_TMP <= COUNT_TMP + 4'h1;
46                                 CARRY <= 1'b0;
47                             end
48                     else
49                         if (COUNT_TMP == 4'h1)
```

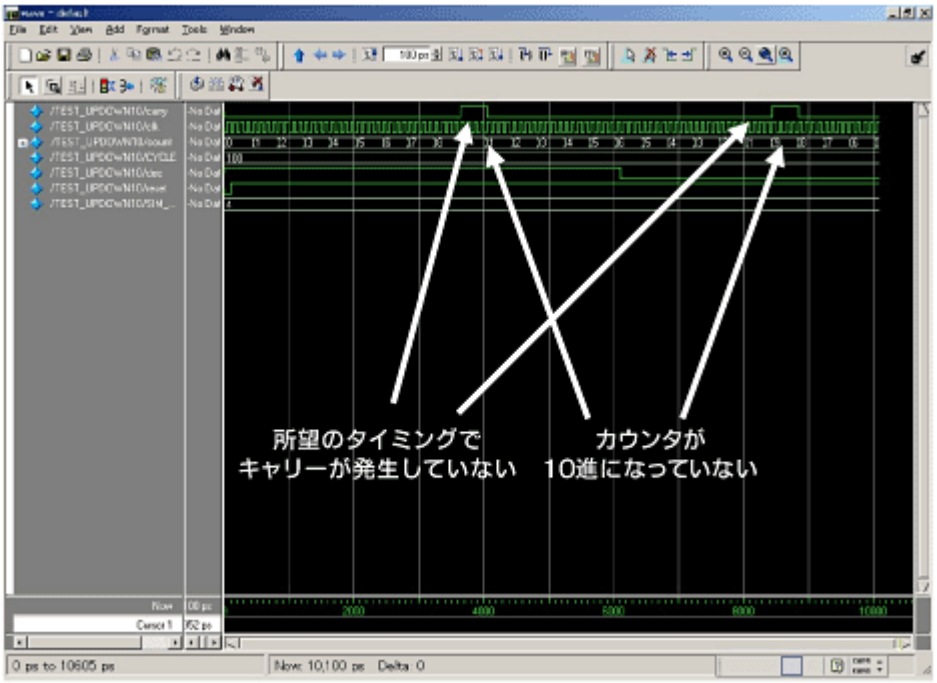
```
50                          begin
51                              COUNT_TMP <= 4'h9;
52                              CARRY <= 1'b1;
53                          end
54                      else
55                          begin
56                              COUNT_TMP <= COUNT_TMP - 4'h1;
57                              CARRY <= 1'b0;
58                          end
59 end
```

（省略）

リスト 4　不正な変更によるキャリーの作り方
（UPDOWN10-2.v）

　確かに‘8’の次の値でキャリーは‘1’になりましたが、10 進カウンタが 9 進になってしまいました……。これはいけません。

　次に「アップカウント時に 10 進を戻す値のデコードを‘9’にして、キャリーの生成を‘8’でデコードすればOK だ！ 」と思い付くのではないでしょうか。その記述を**リスト 5** に示します。



画面 2　不正なタイミングでのキャリーによるけた上げと不正なカウンタの動作

（省略）

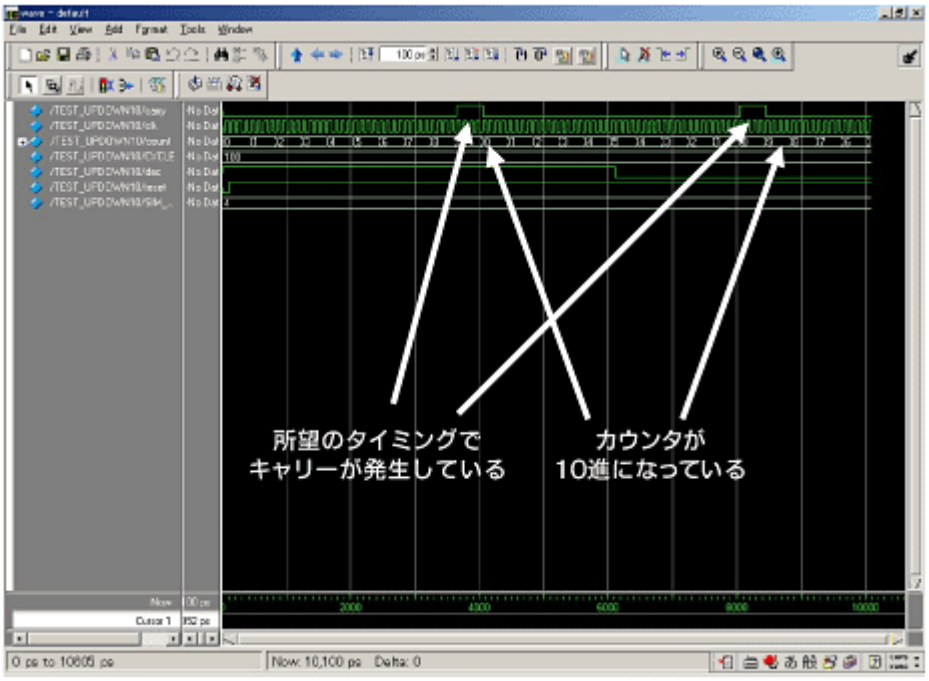
```
28 always @(posedge CLK or negedge RESET)
29 begin
30         if (RESET == 1'b0)
31             begin
32                 COUNT_TMP <= 4'h0;
33                 CARRY <= 1'b0;
34             end
35         else if (ENABLE == 1'b1)
36 //       else if (DEC == 1'b1)
37             if (DEC == 1'b1)
38                 begin
39                     if (COUNT_TMP == 4'h9)
40                         COUNT_TMP <= 4'h0;
41                     else
42                         COUNT_TMP <= COUNT_TMP + 4'h1;
43                     if (COUNT_TMP == 4'h8)
44                         CARRY <= 1'b1;
45                     else
46                         CARRY <= 1'b0;
47                 end
48             else
49                 begin
50                     if (COUNT_TMP == 4'h0)
51                         COUNT_TMP <= 4'h9;
52                     else
53                         COUNT_TMP <= COUNT_TMP - 4'h1;
```

```
54                      if (COUNT_TMP == 4'h1)
55                          CARRY <= 1'b1;
56                      else
57                          CARRY <= 1'b0;
58                  end
59 end
```

（省略）

リスト 5　不正なハードウェア構成によるキャリーの作り方（UPDOWN10-3.v）

　これをシミュレーションすると、確かに 10 進カウンタの動作で、キャリーも思っていたとおりのタイミングで出力されます(**画面 3**)。



画面 3　正確なタイミングでのキャリーによるけた上げと正確なカウンタの動作

　「めでたし、めでたし」といいたいところですが、ハードウェア設計としては実はこのやり方は“最低の対処方法”です。

　この設計方法のハードウェア構成は**図 5** のようになります。

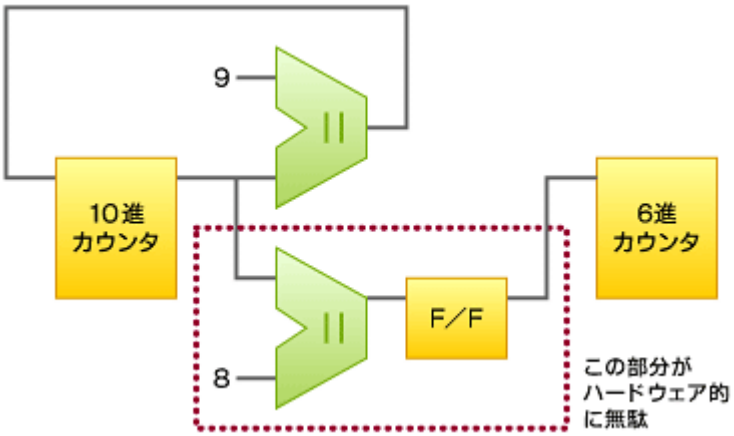


図 5　誤った 60 進カウンタのハードウェア構成

　図 5 のハードウェア構成ですと、アップカウント時のカウンタの値を‘8’と比較して、その結果をさらに F／F で受けて出力しています。そうです。この F／F によりキャリー信号が 1 サイクル遅れてしまうのです。

　なぜ、F／F が使われたのでしょうか？

　なぜなら、always @(posedge CLK .....) と記述されている部分は、その出力、つまり代入されている左辺の信号に必ず F／F が付いてくるからです。

　元の設計では、このキャリー信号を組み合わせ回路の出力として作成するべきでした（**図 6**）。つまり、キャリー信号の作成は、この順序回路になる always 文中で行ってはいけないのです。

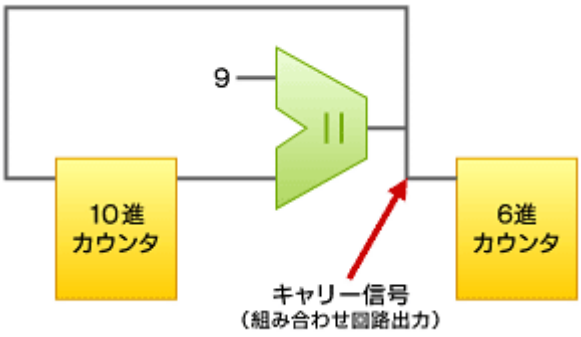


図 6　正しい 60 進カウンタのハードウェア構成

　ここが普通のソフトウェアプログラミングとの大きな違いです。**“HDL を記述するときには、順序回路を記述しているのか、組み合わせ回路を記述しているのかを意識して記述する必要がある”**のです。これが今回の連載の最大のポイントです。

最終的なハードウェア構成

キャリー信号は、組み合わせ回路の出力として作成する必要があるので、最終的には**リスト 6** のようにキャリー信号を作成します（最終的な 60 進カウンタの**リスト 7**では、10 進カウンタに相当する部分の信号名を「COUNT_TMP」→「CNT10」に変更しています）。

（省略）

```
28 always @(posedge CLK or negedge RESET)
29 begin
30         if (RESET == 1'b0)
31             begin
32                COUNT_TMP <= 4'h0;
33                CARRY <= 1'b0;
34             end
35         else if (ENABLE == 1'b1)
36 //       else if (DEC == 1'b1)
37             if (DEC == 1'b1)
38                 begin
39                    if (COUNT_TMP == 4'h9)
40                        COUNT_TMP <= 4'h0;
41                    else
42                        COUNT_TMP <= COUNT_TMP + 4'h1;
43                 end
44             else
45                 begin
46                    if (COUNT_TMP == 4'h0)
47                        COUNT_TMP <= 4'h9;
48                    else
49                        COUNT_TMP <= COUNT_TMP - 4'h1;
50                 end
51 end
52
53 always @(COUNT_TMP or DEC)
54 begin
55         if (DEC == 1'b1)
56             if (COUNT_TMP == 4'h9)
57                 CARRY <= 1'b1;
58             else
59                 CARRY <= 1'b0;
60         else
61             if (COUNT_TMP == 4'h0)
62                 CARRY <= 1'b1;
63             else
64                 CARRY <= 1'b0;
65 end
```

（省略）

リスト 6　正しいハードウェア構成によるキャリーの作り方（UPDOWN10-4.v）

作成したキャリー信号は 6 進カウンタと 10 進カウンタで参照します。このけた上げ・けた下げの信号は、ある意味 1 秒のイネーブルと同じ意味合いがあるととらえてもよいので、ここは if 文でネストせずに、つまり優先順位としては ENABLE と同じと考えて“&&”の条件演算子で記述します。こうすることで 60 進カウンタがきちんと動作するのです（**リスト 7**）。

（省略）

```
10  always @(posedge CLK or negedge RESET)
11  begin
12          if (RESET == 1'b0)
13              begin
14                 CNT10 <= 4'h0;
15              end
16          else if (ENABLE == 1'b1)
17  //       else if (DEC == 1'b1)
```

```
18              if (DEC == 1'b1)
19                  begin
20  //                   if (CNT10 == 4'h9)
21                      if (CARRY == 1'b1)
22                          CNT10 <= 4'h0;
23                      else
24                          CNT10 <= CNT10 + 4'h1;
25                  end
26              else
27                  begin
28  //                   if (CNT10 == 4'h0)
29                      if (CARRY == 1'b1)
30                          CNT10 <= 4'h9;
31                      else
32                          CNT10 <= CNT10 - 4'h1;
33                  end
34  end
35
36  always @(CNT10 or DEC)
37  begin
38          if (DEC == 1'b1)
39              if (CNT10 == 4'h9)
40                  CARRY <= 1'b1;
41              else
42                  CARRY <= 1'b0;
43          else
44              if (CNT10 == 4'h0)
45                  CARRY <= 1'b1;
46              else
47                  CARRY <= 1'b0;
48  end
49
50  always @(posedge CLK or negedge RESET)
51  begin
52          if (RESET == 1'b0)
53              begin
54                 CNT6 <= 3'b000;
55              end
56          else if (ENABLE == 1'b1 && CARRY == 1'b1)
57  //       else if (DEC == 1'b1)
58              if (DEC == 1'b1)
59                  begin
60                      if (CNT6 == 3'b101)
61                          CNT6 <= 3'b000;
62                      else
63                          CNT6 <= CNT6 + 3'b001;
64                  end
65              else
66                  begin
67                      if (CNT6 == 3'b000)
68                          CNT6 <= 3'b101;
69                      else
70                          CNT6 <= CNT6 - 3'b001;
71                  end
72  end
```

（省略）

リスト 7　最終的な 60 進カウンタ（CNT60.v）

図 7、**リスト 8**のようにモジュールを接続したときのテストベンチを**リスト 9**に示します。

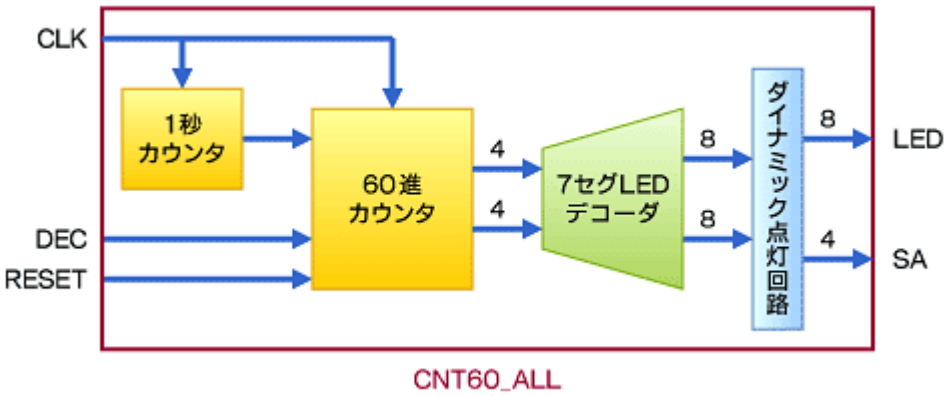


図 7　最終的な 60 進カウンタ＋7 セグメント LED デコーダのハードウェア構成

```
1   module CNT60_ALL(CLK, RESET, DEC, LED, SA);
2   input CLK, RESET, DEC;
3   output [7:0] LED;
4   output [3:0] SA;
5
6   reg [22:0] tmp_count;
7
8   wire [3:0] CNT10;
9   wire [2:0] CNT6;
10  wire ENABLE, ENABLE_kHz;
11  wire [7:0] LED10, LED6;
12
13  parameter SEC1_MAX = 6000000; // 6MHz
14
15  always @(posedge CLK or negedge RESET)
16  begin
17          if (RESET == 1'b0)
18                  tmp_count <= 23'h000000;
19          else if (ENABLE == 1'b1)
20                  tmp_count <= 23'h000000;
21          else
22                  tmp_count <= tmp_count + 23'h1;
23  end
24
25  assign ENABLE = (tmp_count == (SEC1_MAX - 1))? 1'b1 : 1'b0;
26  assign ENABLE_kHz = (tmp_count[11:0] == 12'hfff)? 1'b1 : 1'b0;
27
28  CNT60 i0(.CLK(CLK), .RESET(RESET), .DEC(DEC), .ENABLE(ENABLE),
29           .CNT10(CNT10), .CNT6(CNT6));
30  DECODER7 i1(.COUNT(CNT10), .LED(LED10));
31  DECODER7 i2(.COUNT({1'b0,CNT6}), .LED(LED6));
32  DCOUNT i3(.CLK(CLK), .ENABLE(ENABLE_kHz), .L1(LED10), .L2(LED6),
33          .L3(8'hff), .L4(8'hff), .SA(SA), .L(LED));
34
35  endmodule
```

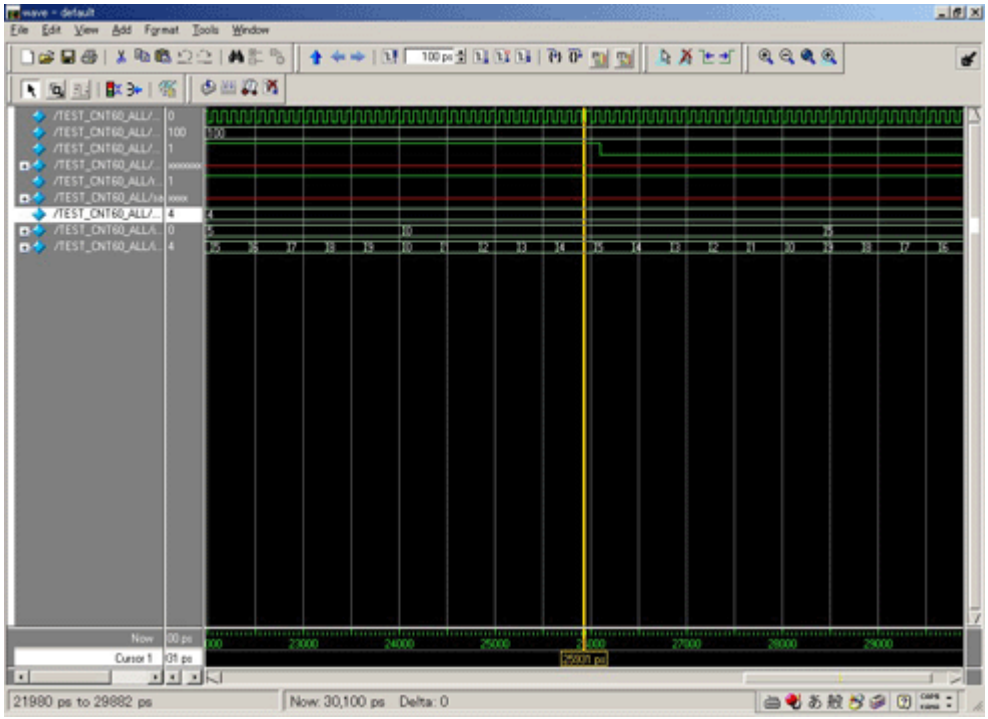
リスト 8　各モジュールのインスタンス（CNT60_ALL.v）

```
1   module TEST_CNT60_ALL;
2   reg clk, reset, dec;
3   wire [7:0] led;
4   wire [3:0] sa;
5
6   parameter CYCLE = 100;
7   parameter SIM_SEC1_MAX = 4;
8
9   CNT60_ALL #(.SEC1_MAX(SIM_SEC1_MAX)) i1(.RESET(reset),
```

```
10                  .CLK(clk), .DEC(dec), .LED(led), .SA(sa));
11
12    always #(CYCLE/2)
13        clk = ~clk;
14
15    initial
16    begin
17        reset = 1'b0; clk = 1'b0; dec = 1'b1;
18        #CYCLE reset = 1'b1;
19        #(65*CYCLE*SIM_SEC1_MAX) dec = 1'b0;
20        #(10*CYCLE*SIM_SEC1_MAX) $finish;
21    end
22
23    initial
24        $monitor($time,,"clk=%b reset=%b dec=%b count60=%d%d", clk, reset, dec, i1.i0.CNT6 , i
                  1.i0.CNT10);
25
26    endmodule
```

リスト 9　最終的な 60 進カウンタのテストベンチ（TEST_CNT60_ALL.v）

シミュレーション結果を**画面 4** に示します。



画面 4　最終的な 60 進カウンタのシミュレーション結果

順序回路と組み合わせ回路を意識

これで 60 進の動作を確認できます。なお、LED の表示がアン・ノンになっていますが、これはダイナミック点灯用のイネーブル信号（ENABLE_kHz）が、このシミュレーションでは出力されないためです。すでに実績のあるモジュールなので、あえてこの部分のシミュレーションは省略しました。

リスト 8のCNT60_ALL.vをRTLのトップモジュールとして、CNT60.v、dcount.v、DECODER7.vをプロジェクトに追加し、**リスト 10** をピン固定ファイルとし（前回のピン固定ファイルとSAの部分が異なっているので注意）、論理合成、配置配線を行い、その後ボードにダウンロードします。

いかがでしょうか？ 7 セグメント LED の右側 2 けたで、60 秒のアップダウン動作を確認できるはずです。

（省略）

```
4   NET "CLK"  LOC = "P39"  ;
5   NET "RESET"  LOC = "P17"  ;
6   NET "DEC"  LOC = "P16"  ;
7   NET "LED<0>"  LOC = "P41"  ;
8   NET "LED<1>"  LOC = "P40"  ;
9   NET "LED<2>"  LOC = "P31"  ;
10  NET "LED<3>"  LOC = "P30"  ;
11  NET "LED<4>"  LOC = "P22"  ;
12  NET "LED<5>"  LOC = "P21"  ;
13  NET "LED<6>"  LOC = "P20"  ;
```

```
14  NET "LED<7>"  LOC = "P19"  ;
15  NET "SA<0>"  LOC = "P46"  ;
16  NET "SA<1>"  LOC = "P45"  ;
17  NET "SA<2>"  LOC = "P44"  ;
18  NET "SA<3>"  LOC = "P43"  ;
```

（省略）

リスト 10　ピン固定ファイル（CNT60_ALL.ucf）

「いちいちキャリーの作成なんて面倒！ 」と思う方もいらっしゃるかもしれません。そこで、6 進カウンタの方で 10 進カウンタの値をそのまま見るという例を**リスト 11** に示します。

```
1   module CNT60(CLK, RESET, DEC, ENABLE, CNT10, CNT6);
2   input CLK, RESET, DEC, ENABLE;
3   output [3:0] CNT10;
4   output [2:0] CNT6;
5
6   reg [3:0] CNT10;
7   reg [2:0] CNT6;
8   //reg CARRY;
9
10  always @(posedge CLK or negedge RESET)
11  begin
12          if (RESET == 1'b0)
13              begin
14                 CNT10 <= 4'h0;
15              end
16          else if (ENABLE == 1'b1)
17  //       else if (DEC == 1'b1)
18              if (DEC == 1'b1)
19                  begin
20                     if (CNT10 == 4'h9)
21  //                   if (CARRY == 1'b1)
22                        CNT10 <= 4'h0;
23                     else
24                        CNT10 <= CNT10 + 4'h1;
25                  end
26              else
27                  begin
28                     if (CNT10 == 4'h0)
29  //                   if (CARRY == 1'b1)
30                        CNT10 <= 4'h9;
31                     else
32                        CNT10 <= CNT10 - 4'h1;
33                  end
34  end
35
36  /*
37  always @(CNT10 or DEC)
38  begin
39          if (DEC == 1'b1)
40              if (CNT10 == 4'h9)
41                  CARRY <= 1'b1;
42              else
43                  CARRY <= 1'b0;
44          else
45              if (CNT10 == 4'h0)
46                  CARRY <= 1'b1;
47              else
48                  CARRY <= 1'b0;
49  end
```

```
50  */
51
52  always @(posedge CLK or negedge RESET)
53  begin
54          if (RESET == 1'b0)
55              begin
56                 CNT6 <= 3'b000;
57              end
58          else if (ENABLE == 1'b1)
59  //        else if (ENABLE == 1'b1 && CARRY == 1'b1)
60  //       else if (DEC == 1'b1)
61              if (DEC == 1'b1)
62                  begin
63                     if (CNT10 == 4'h9)
64                         if (CNT6 == 3'b101)
65                             CNT6 <= 3'b000;
66                         else
67                             CNT6 <= CNT6 + 3'b001;
68                  end
69              else
70                  begin
71                     if (CNT10 == 4'h0)
72                         if (CNT6 == 3'b000)
73                             CNT6 <= 3'b101;
74                         else
75                             CNT6 <= CNT6 - 3'b001;
76                  end
77  end
78
79  endmodule
```

リスト 11　最終的な 60 進カウンタ（CNT60-2.v）

この方法では、確かにキャリーの信号を作成する必要はありません。しかし、この部分も非常に大事で「==4'h9」と書いた部分は「組み合わせ回路（コンパレータ）からの出力なんだ」ということを意識しないといけないのです。

順序回路を記述している場合でも、**“if 文の「()」の中の条件などは、組み合わせ回路で作成されているんだ”**ということを忘れないでください。

HDLソース中のif文などで同じ条件式を繰り返し記述していると、論理合成が同じ組み合わせ回路を“至る所で”作成している可能性があり、それによって回路を大きくしている場合もあるのです。無駄な資源を食わない回路の方がハードウェアとしては優秀です。特に、FPGAの資源が豊富にあるからといって、**図 5**のような無駄な回路を作成しているようでは、ハードウェア設計者として合格点はもらえないでしょう。

◇

最終回である今回は、設計の肝となる「順序回路と組み合わせ回路を意識しながら HDL を記述する」ことについて解説しました。HDL は、あくまでもハードウェアを設計するための記述言語ですので、設計の良しあしは記述する人がすべて握っています。できるだけ無駄なハードウェアを生成しないで、良い設計を行いましょう。

では、どうしたらそうなれるのか？ ひとえに場数しかないと思います。HDL を記述してはツールでハードウェアに落として、FPGA のボードで試してみることです。昔は、こんなことをやりたくても途方もないお金を掛けないとできませんでした。現在は FPGA というデバイスがあって、ミスしても書き直せますし、非常に短時間で本物の回路を作成できます。

ハードウェア設計者の減少が叫ばれている今日このごろ、これを機会に 1 人でも多くの方々が FPGA 設計を通して、ハードウェア設計に興味を持ってもらえたらと、切に願う次第です。

関連記事：
強気のザイリンクス「2010 年にはASICに追い付く」
CPLDからFPGA、ASICまでそろうアルテラ