Text of reconfigurable device Documentation

リリース *0.0.1*

Masayuki Takagiwa

2019 年 06 月 02 日

目次

Contents:

第 1 章

はじめに

本書の目的は VHDL による FPGA (CPLD) への論理回路の実装を一通り体験することです。

到達目標に、「VHDL によるストップウォッチの実装」を設定し、VHDL の説明と実習を行います。

八戸高専の場合、「ディジタル回路」「ディジタル回路」「ディジタル信号処理」と重複するところがあります。それらの授業で学んだことが実習の助けになるでしょう。

1.1 受講にあたって

以下の点に注意、協力してください。

- お昼休みには基板の電源を切ってください。変な回路を書き込んでおいてデバイスに負荷をかけて壊してしまうことを避けるためです。
- 演習 = このテキスト通りに入力して動作を確認するもの、課題 = 自力でコードを書いて動作させるもの、としています。
- 演習、課題でどうしてもツールのエラーが消せないときは呼んでください。
- 課題はできあがったら基板を持って見せに来てください。達成度を記録します。
- 成績は、課題の達成度と筆記試験から求めます。
- 筆記試験は最終日に行い、テキストなどの持ち込みは不可です。
- 欠席せざるを得ないことがあると思います。そのとき課題は授業中、どうにもならないときは後からおこなってください。

第２章

ストップウォッチについて

ディジタルタイプのストップウォッチの例を図に示します。通常「スタート・ストップ」「ラップ・リセット」の２つのボタンと、６桁程度の数字のディスプレイがあります。

「スタート・ストップ」のボタンは、押す毎に「計時」の実行、停止が切り替わります。リセットしない限り、計時は停止した値から再開されます。

「ラップ・リセット」は、停止時に押すと値がクリアされます。計時を行っているときに押すと、通常計時中は変化し続けている数字が停止しますが、計時は続けられています。

実際にはここでは２つの動作があり、「スプリット」では値は引き続きカウントアップしていますが、「ラップ」ではラップボタンで計時は０から再開されます。

数字のディスプレイは通常上位から２桁ずつ、分、秒、1/100 秒のカウントを表示します。

2.1 実装目標

最低限の目標は以下の機能を VHDL で実装することです。

４桁の数字の表示　上位２桁は秒、下位２桁は 1/100 秒です。

「スタート・ストップ」スイッチの動作　一つのスイッチで行います。

「ラップ・リセット」スイッチはリセット動作の実装　停止中に値のリセットが行えること。

追加目標はラップ機能、スプリット機能の実装となります。

第 3 章

FPGA について

FPGA はディジタル信号を取り扱う IC、LSI の一種です。

ターゲットの回路が決まっており、予算が十分にあれば、専用の IC 、LSI を製造するのが、デバイス単価、動作周波数、消費電力のいずれも有利です。

それに対し FPGA は内部に組み込む回路を書き換えられることが利点となっています。

「試作・実験」「少量多品種」に向いたデバイスです。

使用例:

液晶テレビ　FPGA の単価は一般に同じ規模の LSI より高いため、家電製品に FPGA が使われることはありませんが、ある液晶テレビでは一時 FPGA が使われました。理由は明らかではありませんが、恐らく LSI の設計、製造が間に合わないか、組み込む回路に変更が入る恐れがあったものと考えられます。LSI は設計や製造に時間がかかり、また内部の回路変更には作り直しが必要ですが、FPGA では直ぐに回路を組み込むことができ、データを入れ替えるだけで変更も可能です。

携帯電話と基地局の試作機　携帯電話や基地局は世界的な規格に則って作られますが、規格が固まってから作っていると発売が遅くなってしまいます。規格が固まる前にシステムを作ってしまい、規格変更に合わせてあとから回路が変更できるよう、FPGA が大量に使われます。規格や回路が固まったら、必要に応じて LSI に移行します。

株・為替自動取引　コンピュータによる高頻度取引では１秒以下の周期で株価、為替の情報を監視し、自身に有利なタイミングで売買を行います。ソフトウェアだけでの処理は遅いと考えられており、FPGA による処理が併用されます。処理のアルゴリズムは使用者毎に異なるため、LSI 化はできず、FPGA が使用されます。

注釈: 実際にこの授業で使用するデバイスは「PLD」と呼ばれます。PLD は Programmable Logic Device の略です。プログラム可能な論理デバイス、となります。

対して FPGA は Field Programmable Gate Array の略です。現場 (Field) でプログラム可能なゲート (論理) の集合、となります。

できることはほぼ同じですが、以下のような特徴で分けています。

CPLD

- フラッシュメモリを内蔵し、電源を入れるとすぐ使える。
- 回路規模は小さめ。
- 基本的に論理回路しか入ってない。
- 動作速度が比較的高い。
- 構造上、XOR が少し苦手。

FPGA

- 動作のプログラムは揮発性のメモリに格納されるので、電源を切ると内容が失われる。そのため別途フラッシュメモリが必要。
- 回路規模はかなり大きい。
- 高機能 (乗算用ハードウェアが入っていたり、メモリが入っていたりする) 。
- 動作速度は今ではだいぶ高くなってきた。

いずれも D フリップフロップと組み合わせ回路がセットになっており、その入力をプログラムするような形です。このセットがデバイスの中に多数組み込まれ、相互に接続できるよう配線とスイッチも同様に多数用意されます。

CPLD と FPGA の間の大きな違いは組み合わせ回路の構成です。CPLD は大量の AND と多入力 OR を組み合わせていますが (実際には NOT も入ります) 、FPGA は AND 、OR に縛られない LUT (LookUp Table) を使用します。たとえば４入力の AND 、４入力の OR に限らず、"4" 等の任意の数値と一致したときに H を出力する、というようなことも簡単にできるよう、より柔軟な構造になっています。

3.1 HDL

ディジタル回路を組み込んだ IC や LSI の開発では、初期には AND 、OR 、NOT 、フリップフロップを CAD 上で回路図のように配置して接続していました。

こういった CAD は、IC 、LSI メーカ毎に異なるため、たとえば完成したデータを違う LSI メーカで製造しなければいけなくなったとき、互換性が無いため入力＆確認し直しが必要になっていました。

こういった状況を避けるために、IC 、LSI の開発も、ソフトウェアのようにテキストで行おうという動きがでてきました。このジャンルの言語は HDL (Hardware Description Language) と呼ばれます。

ソフトウェアと同様、HDL も様々なものが開発されましたが、現在の主流は VerilogHDL と VHDL です。

VerilogHDL、VHDL は FPGA だけでなく、IC、LSI の開発にも利用されます。

3.2 VerilogHDL と VHDL

各言語の特徴を以下に示します (筆者の独断と偏見を含む)

VerilogHDL

- 民間主導- ツールメーカが開発した。
- 比較的便利な記述ができるようになっている。
- 文法上記述ミスが許容される傾向がある＝ デバッグ時に大変なことが多い。
- 日本でのシェアは、こちらがかすかに多いかもしれない。
- C 言語に近い、と評されることがあるものの、上記のような動きがあることから、むしろ BASIC の方が近いと感じる。
- 無料のシミュレータがあるため、特に趣味の範囲での人気は VHDL より高い模様。

VHDL

- 米国国防省主導- ソフトウェアのプログラミング言語をベースに策定
- 融通が利かず、あまり高機能でない。コード量は多くなる傾向かもしれない。
- 海外でのシェアは、こちらがかすかに多いかもしれない。
- BASIC ににたキーワードもあるものの、文法エラーの検出の強さは C 言語に近い。

3.3 高位合成

HDL は一般にクロックや信号の衝突などソフトウェアとは異なる考え方が必要になります。それだけ、HDL のプログラマはソフトウェアのプログラマに比べ人口が少ないです。

それでも FPGA に魅力を感じている人々が長年ツールを開発し続け、2014 年頃からそれが安価に入手できるようになりました。

Xilinx 社は HLS という名前で、C 言語での FPGA 開発ができるようになっています。

Intel 社 (Altera 社を買収)、Xilinx 社[*1] とも OpenCL 対応のデータ処理向けの開発環境をそろえています。

より高レベル (ハードウェアから遠いという意味で) な言語によるアプローチもあり、これらは高位合成と呼ばれています。

現在は Java 、Python などの言語での FPGA 開発が可能になっています。

[*1] FPGA メーカーは現在はあまり多くなく、Intel 社と Xilinx 社でほぼ寡占、Lattice 社が単独 3 位のようなポジションで、あとは小規模なメーカーがいくつか。

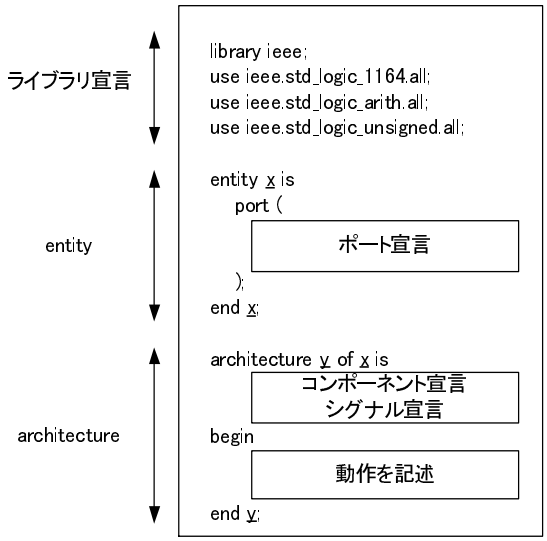
ただし FPGA に組み込む回路全てをこれらの言語だけで開発できないパターンもあり、そういった箇所には引き続き VerilogHDL、VHDL が使われます。

第 4 章

VHDL

4.1 基本形

VHDL のソースコードは、基本的には以下のような形です。



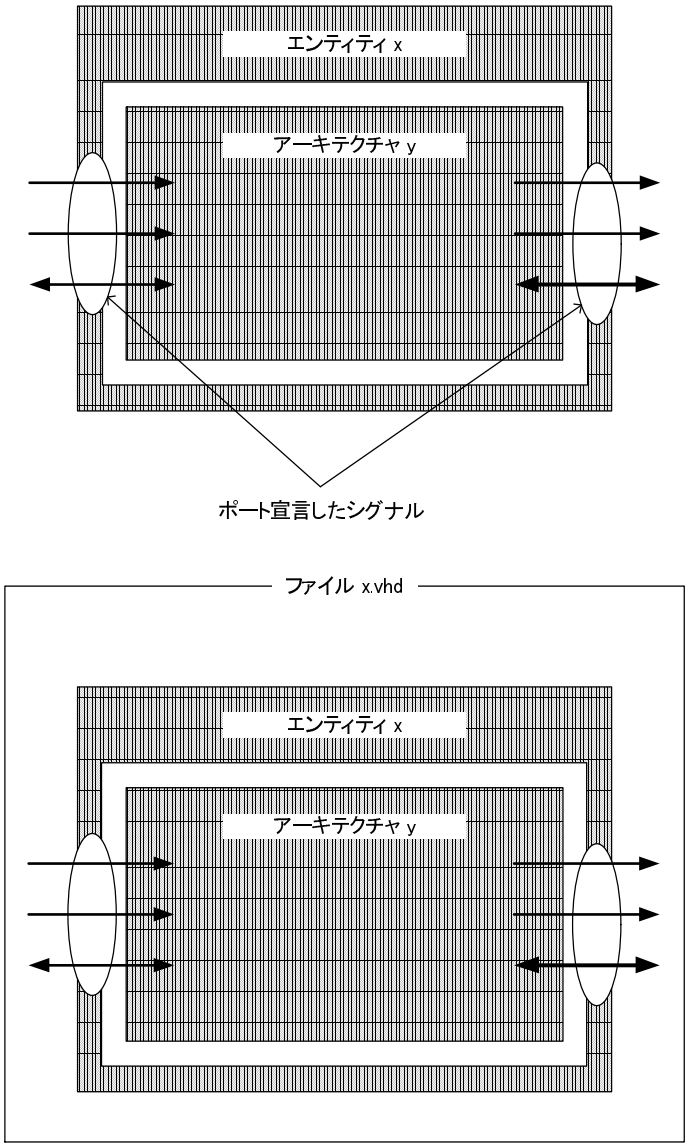
これがテンプレートとなります。コードを記述するときは、この中の x、y、３つの枠の中を、目的に合わせて書き換えます。

ここで x は entity 名、y は architecture 名です。ポート宣言はこの entity が外部とやりとりする信号 (たとえば実際のデバイスのピン) を宣言します。

entity 名はこの回路に対する自由な名前 (たとえば機能から名付けるなど) 、architecture 名はどの entity に対しても同じ「rtl」という名前をつけておけば通常問題ありません。

これらは以下のようなイメージになります。

ファイルにはいくつかの entity 、architecture を含めることができますが、この実習では一つのファイルに一つの entity 、architecture を保存し、ファイル名は entity 名に合わせて設定してください。



4.1.1 解説

4.1.1.1 entity とその周辺

entity と architecture はいずれもそれ単体では何もできません。わざわざ分かれているところから想像できるとおり、同じ entity に対して異なる architecture を適用させることもできますが、この授業では扱いません。

また entity 、architecture が動作するためには、entity の前にある 4 行のライブラリ宣言が必要です。

ここでは ieee の 1164 、arith 、unsigned を呼び出しています。この授業ではこの組み合わせを使用しますので、この 4 行のライブラリ宣言もそのまま使用します。

C 言語のインクルードファイルの宣言などと異なり、ファイルに複数の entity 、architecture を記述する場合、ライブラリ宣言は entity 毎に宣言し直す必要があります。 4 個の entity を記述していれば、その都度、合計 4 回のラ

イブラリ宣言が必要です。

4.1.2 ファイルとの関係

一般にファイル名と entity 名をそろえ、拡張子は「.vhd」とします。

一つのファイルに複数の entity 、architecture を記述することもできますが、その際はそれに対応した use 、library の宣言も必要です。

この授業では、一つのファイルに一つの entity 、architecture とするのがよいでしょう。

またライブラリのうち unsigned は名前から想像できるとおり符号なしの演算を行います。符号付きの演算を行いたい場合は代わりに signed を呼び出します。一つの entity 、architecture 内では混在はできません。

4.2 信号の種類

信号は、ソフトウェアで言う変数と同様のイメージから考えてください。

たとえば C 言語であれば char 、int 、float 等があるように、VHDL でもライブラリを読み込むことで以下のような信号が扱えます。

4.2.1 std_logic

1bit の信号。通常使用する値は 0、1、Z です。

直値を扱う場合、1bit 分ずつシングルクオートで括ります。

0、1 は信号の L レベル、 H レベルに対応します。

Z は「ハイインピーダンス」で、その信号が無い、何もつながっていない状態を作ります。この授業の範囲では使いません。

例

```
' 0 ' , ' 1 ' , 'Z '
```

4.2.2 std_logic_vector

std_logic を束ねたもので、任意のビット数を扱うことができます。

各ビットに代入できる値は std_logic と同じです。

束ねるビット数は宣言時に決めておきます。たとえば１０進数で０～１００までを扱うには 7bit 必要ですので、７本の std_logic を束ねるため、以下のような形式になります。

```
std_logic_vector(6 downto 0)
```

この場合、MSB[*1] が bit 6 、LSB[*2] が bit 0 という宣言になります。ここに代入する値は、たとえば１０進数の１０であれば

```
"0001010"
```

というふうにダブルクオーテーションで括ります。左が bit6 、右が bit0 です。代入する値は、代入先の信号とビット幅が一致している必要があります[*3] 。

例

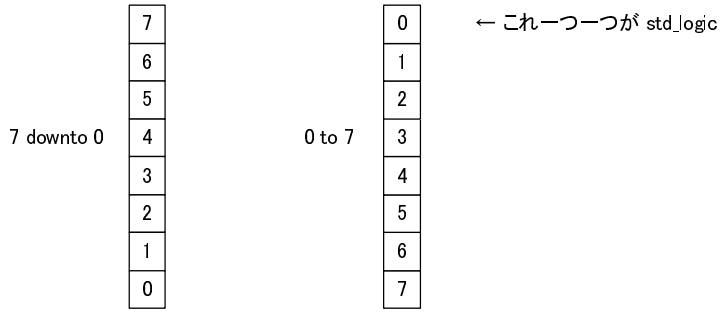
```
"0000", "010101010", "00Z00Z"
```

括弧 () でビット番号を指定することで、std_logic として 1bit 抜き出して扱うことができます。

注釈: std_logic_vector(0 to 6) という宣言の仕方もありますが、ソースやプロジェクトの中で混在させるのはバグのもとになるので通常はどちらかに統一します。

この授業では downto に統一します。

図の上を MSB、下を LSB とした場合、downto と to の関係は次の図のようになります。



4.2.3 integer

10 進数を直接扱います。bit は意識しません。std_logic や std_logic_vector とは直接接続することはできません。

[*1] 変化することにより全体の値が大きく変化するビット
[*2] 変化から値全体の変化が一番小さいビット
[*3] ビット幅が一致していなくてもツール上エラーにならない場合があり、発見しづらいバグになりやすいです。

4.3 ポート宣言

ポート宣言では、この entity (回路ブロック) が外部とやりとりする信号を定義します。

複数の信号を定義でき、それぞれ以下のような形です。

ポート名: 方向信号型

定義の区切りにセミコロンが必要です。定義の終わりを示すものでは無いので、最後の定義ではセミコロンは書きません。

例

```
extsignal1 : in std_logic;
extsignal2 : out std_logic_vector(3 downto 0);
extsignal3 : inout std_logic;
extsignal4 : buffer std_logic
```

ポート名は任意の名前をつけ、architecture 内からその信号にアクセスできます。

方向については上記の 4 パターンがあります。

in　この entity への入力です。architecture 内では読むことしかできません。

out　この entity からの出力です。architecture 内で書き込むことしかできません。

inout　入出力両方ができます。architecture 内では読み書きができますが、信号が衝突すると電気的に短絡 (ショート) となるため、エラーとなります。エラーにならない対応はこの授業では扱いません。

buffer　この entity からの出力です。out との違いは信号の再利用ができることですが、制約もあるので使うのは避けた方がよいでしょう。

それぞれのイメージを図に示します。読み書きは他の signal へ、または signal からの「代入」と読み替えてもよいでしょう。

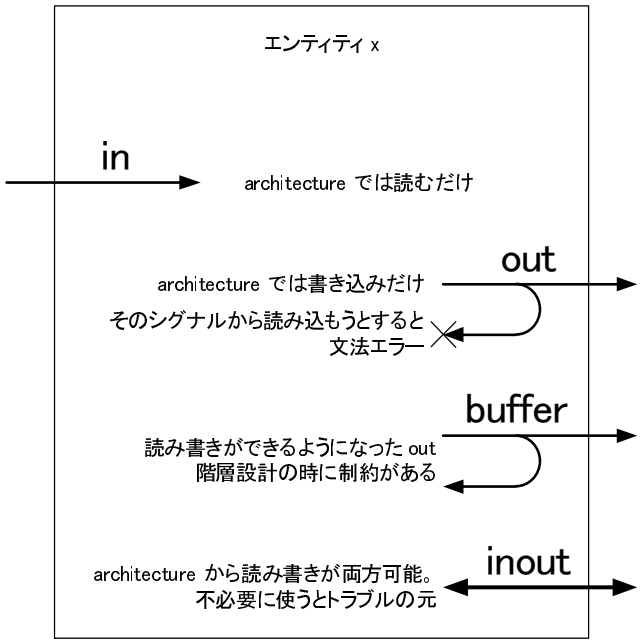
buffer と inout はこの授業では使用しません。

4.4 シグナル宣言

アーキテクチャの中で使用する信号を宣言します。アーキテクチャの記述の中で、begin の前に行います (begin より後には宣言できません) 。

例

```
signal intsignal1 : std_logic;
signal intsignal2 : std_logic_vector(3 downto 0);
```




```
signal intsignal3 : integer;
signal intsignal4 : std_logic;
```

「signal」 キーワードの後に、ポート宣言と同様に任意の名前をつけます。信号の種類もポート宣言と同様です。

アーキテクチャ内部でのみ使用するので、方向は記述しません。

また全ての宣言で末尾はセミコロンで閉じます。

4.5 値の代入

代入は 「<=」 で行います。

例 (1)

```
intsignal1 <= '1';
intsignal4 <= intsignal1;
intsignal2 <= "0000";
intsignal3 <= 5;
```

std_logic_vector の場合は値の扱い方にバリエーションがあるのでここで解説します。

合計したビット数が代入先の信号に一致していれば、&で結合できます。

括弧でビットを指定すれば、std_logic を代入したり、参照したりできます。

4.6 もう一つの信号 variable と代入

VHDL には port 、signal の他にもう一つ、variable という信号のタイプがあります。

variable は後述する process 文の中でのみ使用でき、信号のタイプは他と同様 std_logic 等を使うことができます。

代入には := を使います。

variable はこの授業では扱いません。

第 5 章

演習

5.1 注意事項

- 開発環境 Quatus II のプロジェクトは、実習項目毎に作り直してください。流用するとトラブルの元になります。
- Quatus II の中で日本語を入力するのは避けましょう。こちらもトラブルの元になります。

5.2 論理演算

5.2.1 演習

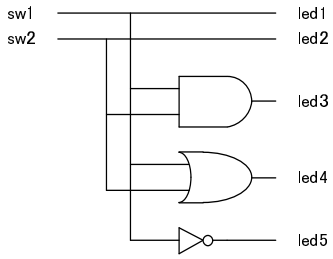
プロジェクト名 vhdl01

スイッチ入力に対して論理演算を行い、結果を LED に出力します。

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity vhdl01 is
  port (
    sw1  : in std_logic;
    sw2  : in std_logic;
    led1 : out std_logic;
    led2 : out std_logic;
    led3 : out std_logic;
    led4 : out std_logic;
    led5 : out std_logic
  );
end vhdl01;
```

```
architecture rtl of vhdl01 is
begin
  led1 <= sw1;
  led2 <= sw2;
  led3 <= sw1 and sw2;
  led4 <= sw1 or sw2;
  led5 <= not sw1;
end rtl;
```

このコードで出力される回路は図のようなものになります。



スイッチは基板奥の方 (7 セグ LED 側) に倒すと H 、手前で L のレベルになります。LED に対してはデバイスから H を与えると消灯、L を与えると点灯します。たとえば led1 はスイッチ 1 番を奥に倒すと消灯、手前に倒すと点灯します。led5 は逆の点灯の仕方になります。

注釈: この授業で使っている基板は 2 つのリビジョンがあり、リビジョン間でスイッチの極性が異なります。

期待した動きをしないときは、これが原因の場合があります。cd

5.3 数値演算

5.3.1 演習

プロジェクト名 vhdl02

スイッチ入力に対して数値演算を行い、LED に表示します。

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity vhdl02 is
  port (
```

```
    sw1     : in std_logic;
    sw2     : in std_logic;
    sw3     : in std_logic;
    sw4     : in std_logic;
    sw5     : in std_logic;
    sw6     : in std_logic;
    sw7     : in std_logic;
    sw8     : in std_logic;

    led1    : out std_logic;
    led2    : out std_logic;
    led3    : out std_logic;
    led4    : out std_logic;
    led5    : out std_logic;
    led6    : out std_logic;
    led7    : out std_logic;
    led8    : out std_logic;

    sled1a : out std_logic;
    sled1b : out std_logic;
    sled1c : out std_logic;
    sled1d : out std_logic;
    sled1e : out std_logic;
    sled1f : out std_logic;
    sled1g : out std_logic;
    sled2a : out std_logic;
    sled2b : out std_logic;
    sled2c : out std_logic;
    sled2d : out std_logic;
    sled2e : out std_logic;
    sled2f : out std_logic;
    sled2g : out std_logic
  );
end vhdl02;

architecture rtl of vhdl02 is
  signal s_sw1 : std_logic;
  signal s_sw2 : std_logic;
  signal s_sw3 : std_logic;
  signal s_sw4 : std_logic;
  signal s_sw5 : std_logic;
  signal s_sw6 : std_logic;
  signal s_sw7 : std_logic;
  signal s_sw8 : std_logic;

  signal sw : std_logic_vector(8 downto 1);

  signal add : std_logic_vector(4 downto 1);
  signal sub : std_logic_vector(4 downto 1);
```

```
  signal led71 : std_logic_vector(6 downto 0);
  signal led72 : std_logic_vector(6 downto 0);
begin
  -- fixed polarity
  -- revision 2
  -- s_sw1 <= sw1;
  -- s_sw2 <= sw2;
  -- s_sw3 <= sw3;
  -- s_sw4 <= sw4;
  -- s_sw5 <= sw5;
  -- s_sw6 <= sw6;
  -- s_sw7 <= sw7;
  -- s_sw8 <= sw8;
  -- revision 1 (no revision)
  s_sw1 <= not sw1;
  s_sw2 <= not sw2;
  s_sw3 <= not sw3;
  s_sw4 <= not sw4;
  s_sw5 <= not sw5;
  s_sw6 <= not sw6;
  s_sw7 <= not sw7;
  s_sw8 <= not sw8;

  -- left 4 switches & right 4 switches
  -- left switch is MSB
  sw <= s_sw1 & s_sw2 & s_sw3 & s_sw4 & s_sw5 & s_sw6 & s_sw7 & s_sw8;

  add(4 downto 1) <= sw(8 downto 5) + sw(4 downto 1);
  sub(4 downto 1) <= sw(8 downto 5) - sw(4 downto 1);

  -- make 7 seg LED bit map
  led71 <= "0010000" when (add = "1001")
      else "0000000" when (add = "1000")
      else "1111000" when (add = "0111")
      else "0000010" when (add = "0110")
      else "0010010" when (add = "0101")
      else "0011001" when (add = "0100")
      else "0110000" when (add = "0011")
      else "0100100" when (add = "0010")
      else "1111001" when (add = "0001")
      else "1000000" when (add = "0000")
      else "0000110" ;
  led72 <= "0010000" when (sub = "1001")
      else "0000000" when (sub = "1000")
      else "1111000" when (sub = "0111")
      else "0000010" when (sub = "0110")
      else "0010010" when (sub = "0101")
      else "0011001" when (sub = "0100")
```

```
      else "0110000" when (sub = "0011")
      else "0100100" when (sub = "0010")
      else "1111001" when (sub = "0001")
      else "1000000" when (sub = "0000")
      else "0000110" ;

  -- map to pin
  sled1a <= led71(0);
  sled1b <= led71(1);
  sled1c <= led71(2);
  sled1d <= led71(3);
  sled1e <= led71(4);
  sled1f <= led71(5);
  sled1g <= led71(6);
  sled2a <= led72(0);
  sled2b <= led72(1);
  sled2c <= led72(2);
  sled2d <= led72(3);
  sled2e <= led72(4);
  sled2f <= led72(5);
  sled2g <= led72(6);

  -- debug
  led1 <= not add(4);
  led2 <= not add(3);
  led3 <= not add(2);
  led4 <= not add(1);
  led5 <= not sub(4);
  led6 <= not sub(3);
  led7 <= not sub(2);
  led8 <= not sub(1);
end rtl;
```

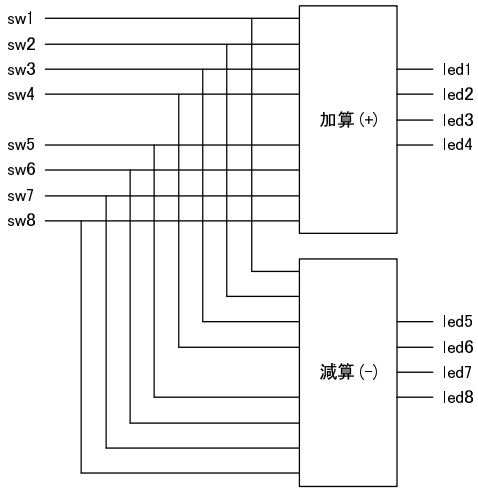
スイッチの上位 (基板上左側) 4 ビットと下位 (右側) 4 ビットの演算を行います。LED の上位に加算の結果、下位に減算の結果が 2 進数で表示されます。

たとえばスイッチを 8 から 1 まで、10100001 (1=ON, 0=OFF) とした場合、結果 (= LED の点灯パターン) は 10111001 となります。

また 7 セグ LED にも結果を表示します。 7 セグ LED は 0 〜 9 までの表示を行います。

注釈: 符号無し 4 ビット演算です。

このコードでは、10 以上の数値 (具体的には 0 〜 9 以外) は 7 セグ LED の表示は E になります。



5.4 条件分岐 when ～ else ～

5.4.1 演習

プロジェクト名 vhdl03

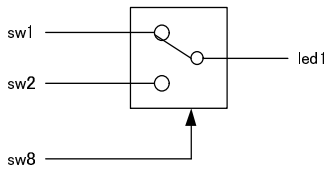
条件分岐の書き方の一つ、when ～ else ～ の例です。

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity vhdl03 is
  port(
    sw1  : in std_logic;
    sw2  : in std_logic;
    sw8  : in std_logic;
    led1 : out std_logic
  );
end vhdl03;

architecture rtl of vhdl03 is
begin
  led1 <= sw2 when (sw8 = '1')
     else sw1;
end rtl;
```

sw8 を H レベルにしておくと、LED1 は sw2 の操作に従って点灯します。

sw8 が L レベルの場合は、LED1 は sw1 の操作に従って点灯します。

このコードから生成される回路のイメージは次の図の通りです。



この記述ではいくつでも分岐させることができます。

```
ans <= x1 when (y1 = '1')
  else x2 when (y2 = '1')
  else x3 when (y3 = '1')
  else x4 when (y4 = '1')
  else x5;
```

ただし、判定は記述した順番に行われます。この例でたとえば y1 ～ y4 全てが 1 だった場合、y1 の条件が採用されます (プライオリティエンコーダ＝選択肢に優先順位のあるセレクタ) 。

5.5 条件分岐 with ～ select ～ when ～

5.5.1 演習

プロジェクト名 vhdl04

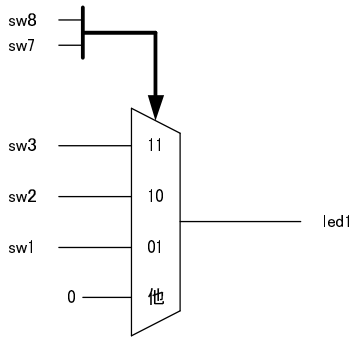
条件分岐の書き方の一つ、with ～ select ～ when ～ の例です。

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity vhdl04 is
  port(
    sw1  : in std_logic;
    sw2  : in std_logic;
    sw3  : in std_logic;
    sw7  : in std_logic;
    sw8  : in std_logic;
    led1 : out std_logic
  );
end vhdl04;

architecture rtl of vhdl04 is
  signal sw : std_logic_vector(1 downto 0);
begin
  sw <= sw8 & sw7;
  with sw select led1 <= sw3 when "11",
```

```
                        sw2 when "10",
                        sw1 when "01",
                        '0' when others;
end rtl;
```

sw7 と sw8 の設定により、LED の点滅を制御できるスイッチを切り替えられる回路です。

このコードから生成される回路のイメージは次の通りです。



中央のブロックはセレクタで、上から入力される値がブロック中の値と一致すると、その箇所の信号が出力されます。

選択肢に優先順位がある場合は when 〜 else 〜、ない場合は with 〜 select 〜 when 〜 を使います。

5.6 process 文

AND、OR、NOT などの組み合わせ回路や単純な条件分岐であればこれまでの内容で対応できますが、フリップフロップなどを組み合わせた順序回路を記述するには、process 文を使用します。

ヒント

フリップフロップ専用の記述ではありませんが、フリップフロップを使用する箇所のみ process 文を使うようにしていると、慣れていないうちはトラブルを避けやすくなります。

基本的な構成は以下の通りです。

```
LABEL: process (SENSITIVITY-LIST)
begin
  〜 EXPRESSION 〜
end process;
```

ラベル (LABEL) を付加するかどうかは任意ですが、識別のために固有の名前をつけておくのがよいでしょう。

センシティビティリスト (SENSITIVITY-LIST) は、この process 文を動作させるトリガになる信号のリストを記述します。

注釈: センシティビティリストに書いた信号は「トリガ」と書きましたが、その信号が変化した時に必ず process 文の内容が実行されるという話ではなく、その信号が変化したときに、process 文の一番最初の if 〜 then 〜 elsif 〜 else 〜 end if の条件で動作する、という点を覚えておきましょう。

信号名はカンマで区切ります。

式 (EXPRESSION) の部分には実際の動作を記述します。ただし条件分岐は、これまでの when 〜 else 〜 や with 〜 select 〜 when 〜 は使用できません。後ほどでてくる if 文や case 文を使用します (逆に if 文や case 文は process 文の外では使用できません) 。

表 5.1　組み合わせ

process 文の	when 〜 else 〜	with 〜 select 〜 when 〜	if 〜 elsif 〜 endif	case 〜 end case
外	○	○	×	×
中	×	×	○	○

5.7 フリップフロップの基本と if 文

フリップフロップは主に D 型、SR 型、JK 型、T 型があります。これらを VHDL で記述するときは一般に process 文を使用します。

フリップフロップのリセットが非同期式の場合、D フリップフロップは以下のように記述します。

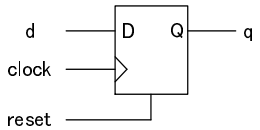
```
D_FF: process (reset, clock)
begin
  if (reset = '1') then
    q <= '0';
  elsif (clock'event and clock = '1') then
    q <= d;
  end if;
end process;
```

この例では、すべての信号は std_logic で定義されているとします。

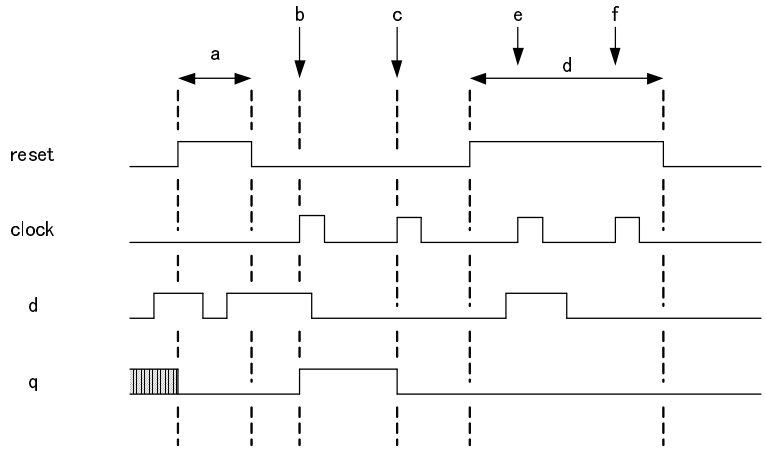
入力データは信号は d 、出力データ信号は q としています。

reset に H レベルの信号を与えるとリセット動作として、q を L レベルにセットしています。

クロック信号 clock が H へ変化すると、q へ d を代入します。



動作例を以下に示します。



a の期間では reset = 『1』 が成り立つので q <= 『0』 が実行され続けます。

b のタイミングでは clock ' event and clock = 『1』 が成り立ちます。これは、clock が変化したこと (clock ' event) と clock = 『1』 の組み合わせで、clock が 0 から 1 に変化したときに成り立つ、という記述の仕方です。このとき q <= d が実行されます。d が 1 なので最終的には q は 1 になります。

c のタイミングも b のタイミング同様、clock が 0 から 1 に変化したことが成り立っています。q には d の値 0 が代入されます。

d の期間では e や f のタイミングで clock ' event が成り立っていますが、if 文 (後述) で reset = 『1』 が先に記述されているため、e 、f は無視され、q には 0 が出力され続けます。

ちなみに FPGA が起動したとき、フリップフロップの値が H 、L どちらになっているかは決まっていません。そのため図で q はリセットがかかるまではどちらの値かわかりません (不定) 。

5.7.1 シミュレーション

順序回路の動作確認にはシミュレータが役に立ちます。ここでシミュレーションにより上記の動作を確認します。

D:\Altera\(学籍番号) に vhdl20 というフォルダを作ります。

Web ブラウザで http://vhdl.cottonrose.jp/2016/ を開き、vhdl20.vhd をダウンロードし、先ほどのフォルダにダウンロードします。

注釈: ダウンロードしたファイルが読み取り専用になっていることがあります。

別冊「Altera 開発環境（QuartusII）使用マニュアル」の 21 ページからのシミュレーションの手順を行います。

34 ページまで実行すると、上記のような波形が見られます (波形の赤いラインは不定を表します)。

注釈: 自分自身への代入について

process 文の中の if 文の中で、clock ’ event and clock = 『1』 の条件の中では、宣言されている signal や variable に自分自身を代入できます。

例えば、a <= a; はこのままでは無意味なコードですが、正常なものです。これは、この代入動作がクロックの変化点のみ行われるのがポイントです。

これを応用すると、a <= not a; であればクロックの変化点事に反転、a <= a + 1; であればカウントアップしていく動きになります。

これが process 文の外、clock ’ event and clock = 『1』 の外では、エラーになるか、予測できない動作になります。エラーにならなかった場合、例えば a <= not a; はデバイスの限界速度で反転を繰り返す信号を生成しますが、スイッチ動作が多すぎて電流消費、発熱から、何が起きるかわかりません。

5.7.2 同期、非同期

「非同期」動作は、クロックによらず動作する箇所、「同期」動作はクロックの変化点に合わせて動作することを示しています。

先ほどのサンプルでは、リセットは非同期動作 (特にこの場合は「非同期リセット」とも呼ばれる) 、データの保持は同期動作としています。

注釈: データをクロックに同期して保持するものを一般にフリップフロップ、同期せずに保持するものをラッチと呼びます。

これは「ディジタルコンピューティングシステム」p54 に書かれていることと逆ですが、会社などの組織毎に異なる場合があるので注意が必要です。

警告: FPGA に実装する回路において、一般にラッチは使用するべきではないとされています。

フリップフロップを使った方が動作が予測しやすいため、デバイス自体にフリップフロップが組み込まれています。それに対しラッチは大規模回路では動作の予測が難しく、バグの元になりやすいため、です。

慣れないうちは、できる限りすべてのフリップフロップは同じクロックで動作するようにし、動作を始める前にはすべて非同期リセットを行うように組むと失敗が少なくなります。

注釈: 「ディジタル回路」は信号レベル (縦軸) を離散的に扱います。「同期回路」は時間軸を離散的に扱うための

手法と考えられます。「非同期回路」は時間軸に不確定な要因を持つことになるため、動作の安定性に影響します。

この授業の実習では、その実習で使用するすべてのフリップフロップは同じクロックで動作させるようにします。

これらを守るためには、上の D フリップフロップの例のようなセンシティビティリスト (信号名は回路に合わせる) 、if 文の構成 (非同期リセットとクロックの動作しかなく、それ以上はその if 、elsif の中に書き足していく) を基本にしていきます。

5.7.3 if 文

条件分岐の基本的なもので、基本的な形は以下の通りです。

```
if (CONDITION) then
  〜 EXPRESSION 〜
end if;
```

条件 (CONDITION) が真であれば、処理 (EXPRESSION) が実行されます。複数の条件ごとに処理を分ける場合は以下のようになります。

```
if (CONDITION_A) then
  〜 EXPRESSION_A 〜
elsif (CONDITION_B) then
  〜 EXPRESSION_B 〜
elsif (CONDITION_C) then
  〜 EXPRESSION_C 〜
else
  〜 EXPRESSION_OTHER 〜
end if;
```

条件 A (CONDITION-A) が真であれば処理 A (EXPRESSION-A) を実行、それ以外で条件 B (CONDITION-B) が真であれば処理 B (EXPRESSION-B) 、それ以外で条件 C (CONDITION-C) が真であれば処理 C (EXPRESSION-C)、それ以外ではその他の処理 (EXPRESSION-OTHERS) を実行します。

注釈: 一般のプログラミング言語と異なり、「else if」ではなく「elsif」と書きます。

条件の書き方は、たとえば一致判定では

```
reset = '1'
```

不一致判定は

```
reset /= '0'
```

数値の大小の比較では、

```
count >= "1010"
```

となります。

条件は複数組み合わせることができ、そのときは and 、or 、not も使用できます。

たとえば

```
a = '1' and b = '0'
```

とすれば、 2つの条件が満たされた場合、と判定されます。クロックの変化点での動作、つまり同期動作について括る場合の条件は、

```
clock'event and clock = '1'
```

が一般的となります。これは clock の変化点と、clock が H という 2つの条件の AND となります。

注釈: 上記のような、クロックの変化点での動作の記述は、一つの if 文で複数書いても文法上は問題ありませんが、実際のデバイス上でそのような動作は行うことができません。

一般に process 文の最初の if 文の構成は最大で

```
process (set, reset, clock)
begin
  if (reset = '1') then
    〜
  elsif (set = '1') then
    〜
  elsif (clock'event and clock = '1') then
    〜
  end if ;
end process;
```

となります。

リセット時、セット (プリセット) 時、その他通常動作時を同じレベルで設定しています。

以下の書き方は許容されますが、理解できていないうちは行うべきではありません。

```
process (clock)
begin
```

```
  if (clock'event and clock = '1') then
    ～
  end if;
end process;
process (clock)
begin
  if (clock'event and clock = '0') then
    ～
  end if;
end process;
```

同じクロックで動作する書き方ですが、1 = クロックの立ち上がりと 0 = クロックの立ち下がりが混在しています。この 2 つの process 文の間でデータのやりとりがあるとき、デバイス上での動作条件が厳しくなります。

以下の書き方は実機には基本的には組み込めません。

```
process (clock)
begin
  if (clock'event and clock = '1') then
    ～
  elsif (clock'event and clock = '0') then
    ～
  end if;
end process;
```

同じ process 文内で、同じクロックの 1 = クロックの立ち上がりと 0 = クロックの立ち下がりが混在しています。同じ signal に対してこのような動作は指示できません。それぞれの処理に異なる signal を入れれば組み込むことはできますが、一つ前の例と同様使うべきではありません。

5.7.4 マルチソース

HDL では、基本的に全ての行が、常に「同時」に動作していると考える必要があります。この点が、ソースコードを逐次解釈していくソフトウェアと大きく異なる点です。

これにより、signal や port への値の代入の仕方に制限があります。

```
architecture rtl of test is
  signal a : std_logic;
begin
  a <= '0';
  a <= '1';
end rtl;
```

ソフトウェアであればこの場合、a に 0 が代入された後でさらに a に 1 が代入されます。しかし HDL の場合、0 と 1 の代入は同時に行おうとし、論理合成の段階でエラーとなります (エラーメッセージには multi source という言葉が含まれます) 。

```
entity x is
  port (
    d1  : in std_logic;
    d2  : in std_logic;
    clk : in std_logic;
    q   : out std_logic
  ) ;
end x;

architecture rtl of x is
  signal a : std_logic;
begin
  a_proc : process (clk)
  begin
    if (clk'event and clk = '1') then
      a <= d1;
    end if;
  end process;

  b_proc : process (clk)
  begin
    if (clk'event and clk = '1') then
      a <= d2;
    end if;
  end process;

  q <= a;

end rtl;
```

別々の process 文で動作させていても、同じ signal に代入しようとしているため、両方の条件が必ず衝突しない書き方出ない限り論理合成でエラーになります (そしてそのように書いたとしても不具合の修正なので書き換えているうちに条件が崩れ multi source のエラーになりやすいです)。

```
entity x is
  port (
    d1  : in std_logic;
    d2  : in std_logic;
    clk : in std_logic;
    q   : out std_logic
  ) ;
end x ;

architecture rtl of x is
  signal a : std_logic;
begin
  a proc : process (clk)
  begin
```

```
    if (clk'event and clk = '1') then
      a <= d1;
    end if;
  end process;

  a <= d2;
  q <= a;

end rtl;
```

このような process 文での順序回路と、通常の組み合わせ回路でも同様です。

5.7.5 process 文、if 文によるフリップフロップの記述のお約束

この教科書ではこの先順序回路を多く使用します。その際、後々のトラブルを避けるため、以下のような書き方を避けることをおすすめします。

```
process (reset, clock, a, b)
begin
  if (reset = '1') then
    〜 RESET-PROCEDURE 〜
  elsif (clock'event and clock = '1' and a = '1') then
    〜 PROCEDURE-A 〜
  elsif (clock'event and clock = '1' and b = '1') then
    〜 PROCEDURE-B 〜
  end if;
end process;
```

どうやら、リセット処理 (RESET-PROCEDURE)、クロックに同期した上で a が 1 の時の処理 (PROCEDURE-A)、クロックに同期した上で b が 1 の時の処理 (PROCEDURE-B) を行いたいらしい。

このような場合は以下のように、クロックによる動作と論理を分離して記述すると良いでしょう。

```
process (reset , clock)
begin
  if (reset = '1') then
    〜 RESET PROCEDURE 〜
  elsif (clock'event and clock = '1') then
    if (a = '1') then
      〜 PROCEDURE A 〜
    elsif (b = '1') then
      〜 PROCEDURE B 〜
    end if;
  end if;
end process ;
```

クロックの条件に追加で書いていると、この授業のレベルではその箇所でミスをしやすくなります。

またクロックの条件と分離することで、他へのコピー＆ペーストでのミスも減ります。

このほか、以下の箇所で代入を行っても、意図した動作にならない可能性が高いです。

```
process (reset , clock)
begin
  if (reset = '1') then
    〜 RESET PROCEDURE 〜
  elsif (clock'event and clock = '1') then
    〜 MAIN PROCEDURE 〜
  end if;
  a <= b;
end process;
```

a へ b を代入していますが、この位置ではセンシティビティリストに入る reset と clock の条件を無視した位置にあるため、どのような動作になるか保証できません。

process 文の外か、if 文の中に入れましょう。

5.7.6 演習

プロジェクト名 vhdl05

D フリップフロップを作る

```
library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity vhdl05 is
  port (
    sw1  : in std_logic ; -- data
    sw2  : in std_logic ; -- clock
    sw3  : in std_logic ; -- reset
    led1 : out std_logic
  );
end vhdl05;

architecture rtl of vhdl05 is
  signal d : std_logic;
begin
  dff : process (sw2, sw3)
  begin
    if (sw3 = '1') then
      d <= '0';
```

```
    elsif (sw2'event and sw2 = '1') then
      d <= sw1;
    end if;
  end process;

  led1 <= d;
end rtl;
```

ヒント

「–」 (マイナス 2 個) 以降は改行までコメントとして無視されます。

この例では、sw1 をデータ入力、sw2 をクロック、sw3 をリセットとして使用しています。

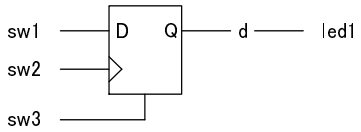
シグナル d を D フリップフロップの実態という意味で定義しています。

sw3 を H にすると、フリップフロップをリセットします。

sw2 が H に変化したタイミング (立ち上がりエッジ) で、sw1 の内容を d へ代入します。そのほかのタイミングでは sw1 の変化の影響を受けません。

process 文内の処理は sw2 と sw3 の変化でしか行われないため、センシティビティリストには sw2 と sw3 しか書いていません。

led1 には d の内容を出力しています。



動作確認

1. sw1、sw2、sw3 をすべて OFF にする。
2. まず sw3 を ON → OFF して、リセットする。
3. sw2 を ON → OFF して、led1 が変化しないことを確認する。 (入力データ sw1 が OFF なので、OFF のデータを改めてサンプルするだけ)
4. sw1 を ON にする。
5. sw2 を ON → OFF して、 led1 が変化することを確認する。
6. sw1 を OFF にしても led1 が変化しないことを確認する。
7. sw2 を ON → OFF して、 led1 が変化することを確認する。

5.7.7 課題

プロジェクト名 vhdl06

下記のソースコードに追記して、JK フリップフロップを作れ

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity vhdl06 is
  port (
    sw1  : in std_logic; -- J
    sw2  : in std_logic; -- clock
    sw3  : in std_logic; -- K
    sw4  : in std_logic; -- reset
    led1 : out std_logic
  );
end vhdl06;

architecture rtl of vhdl06 is
  signal jk : std_logic;
begin
  jkff_proc : process (sw4, sw2)
  begin
    if (sw4 = '1') then
       -- put code below --

       -- put code above --
    elsif (sw2'event and sw2 = '1') then
       -- put code below --

       -- put code above --
    end if;
  end process;

  led1 <= jk;
end rtl;
```

2 箇所の」put code below」 から」put code above」 の間にコードを書いてください。

ソースコード中で指定しているようなスイッチのアサインで、JK フリップフロップを作成してください。

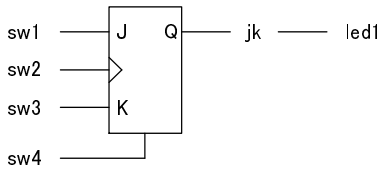


表 5.2　真理値表

sw2 (clock)	sw1(J)	sw3(K)	led1(Q → jk)
↑以外	X	X	維持
↑	0	0	維持
↑	1	0	1
↑	0	1	0
↑	1	1	反転

注釈: 「維持」は、自分自身を代入することで行えます。

動作確認は以下の通り行います。

1. sw1 〜 sw4 を OFF にする。
2. sw4 を ON → OFF として、リセットする (led1 が点灯する) 。
3. sw1 を ON → OFF として、led1 が変化しないことを確認する (J だけ動かしている) 。
4. sw3 を ON → OFF として、led1 が変化しないことを確認する (K だけ動かしている) 。
5. sw1 を ON にし、sw2 を ON → OFF として、led1 が消灯することを確認する。
6. sw1 を OFF にし、sw2 を ON → OFF としても、led1 が消灯したままであることを確認する。
7. sw3 を ON にし、sw2 を ON → OFF として、led1 が点灯することを確認する。
8. sw3 を OFF にし、sw2 を ON → OFF としても、led1 が点灯したままとなることを確認する。
9. sw1 、sw3 を ON にし、sw2 を ON → OFF として、led1 が消灯することを確認する (注記参照) 。
10. sw1 、sw3 を ON にしたままで、sw2 を ON → OFF として、led1 が点灯することを確認する (注記参照) 。

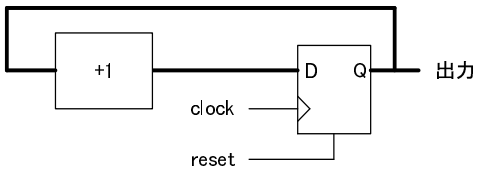
注釈: 最後の２つは、想定通りに動いたり、動かなかったりする。これは後述するチャタリングが原因。

5.8 カウンタ

フリップフロップはクロックに合わせて値を保持します。複数ビットのフリップフロップの組み合わせの出力に加算器を接続し、その結果をフリップフロップに戻すことで、次のクロックでは加算後の結果が保持されます。

加える値が１であれば、１ずつ増えていくカウンタとなります。

プロジェクト名 vhdl07




```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity vhdl07 is
  port (
    gclk0 : in std_logic;
    led1  : out std_logic;
    led2  : out std_logic;
    led3  : out std_logic;
    led4  : out std_logic;
    led5  : out std_logic;
    led6  : out std_logic;
    led7  : out std_logic;
    led8  : out std_logic
  );
end vhdl07;

architecture rtl of vhdl07 is
  signal c : std_logic_vector (24 downto 0);
begin
  count : process (gclk0)
  begin
    if (gclk0'event and gclk0 = '1') then
      c <= c + 1;
    end if;
  end process;

  led8 <= not c (24);
  led7 <= not c (23);
  led6 <= not c (22);
  led5 <= not c (21);
  led4 <= not c (20);
  led3 <= not c (19);
  led2 <= not c (18);
  led1 <= not c (17);
end rtl;
```

クロックはあらかじめ、32MHz を選択しておきます (JP3、JP4、JP5 のうち JP5 だけショートさせる)。

カウンタの信号は c で、process 文 count の中でクロックの立ち上がりエッジの度に 1 を加えます。

c の幅は 25bit なので、0 ～ 33,554,431 までの値を扱うことができます。入力しているクロックが 32MHz なので、約 1 秒で一周するよう、LED が点灯します。

LED には、上位 8bit のみ表示しています。

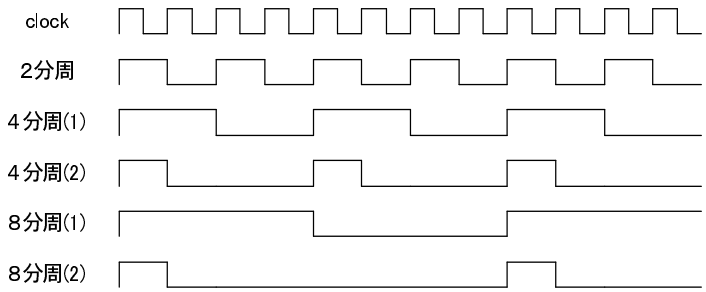
この記述では、ソフトウェアの for や while のようなループが書かれていませんが、センシティビティリストの gclk0 の変化 = クロックの変化毎にプロセス文の記述が呼び出されるため、自動的に繰り返し実行されます。

5.9 分周回路と T フリップフロップ

5.9.1 分周回路

カウンタを応用した回路構成の一つで、元の周波数の整数分の１の周波数を作り出します。

２分周 (1/2 の周波数) 、４分周 (1/4 の周波数) 、８分周 (1/8 の周波数) の例を図に示します。



２分周を超える場合、作られる信号の形は２つのパターンがあります。

一つは、デューティ比 (H レベルと L レベルの期間の比) ができるだけ５０：５０になるよう近づけたパターン。

もう一つは、作り出した周期の中で１クロック分だけ H レベルとし、残りは L とするパターン。

通常デバイス内では後者、デバイスの外では前者が使われることが多いです。

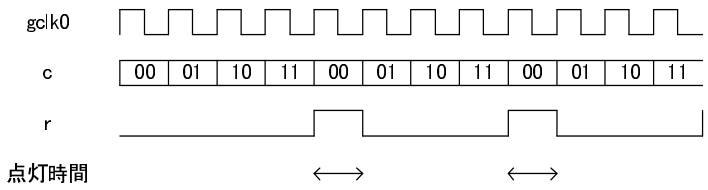
４分周の２つめのパターンの例を以下に示します。

プロジェクト名 vhdl08

```
library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity vhdl08 is
  port (
    gclk0 : in std_logic;
    led1  : out std_logic;
    led2  : out std_logic
```

```
  );
end vhdl08;

architecture rtl of vhdl08 is
  signal c : std_logic_vector (1 downto 0);
  signal r : std_logic;
begin
  div_proc : process (gclk0)
  begin
    if (gclk0'event and gclk0 = '1') then
      c <= c + 1;

      -- make 1/4 frequency
      if (c = "11") then
        r <= '1';
      else
        r <= '0';
      end if;
    end if;
  end process;

  led1 <= not r;
  led2 <= '0';
end rtl;
```

この中で c は１０進数で０～３までカウントし、r にはそのうち３のときだけ『1』がセットされます。その反転による LED の点灯がされるため、gclk0 32MHz で４周期のうち１サイクルだけ点灯することになり、結果として単純に点灯している led2 に比べ led1 が暗くなります。



5.9.2 課題

プロジェクト名 vhdl09

約１秒周期で led1 **の点灯、消灯を繰り返す回路を作れ**　(約 0.5 秒点灯、約 0.5 秒消灯を繰り返す)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
```

```
entity vhdl09 is
  port (
  gclk0 : in std_logic;
  led1  : out std_logic
  );
end vhdl09;

architecture rtl of vhdl09 is
  signal c : std_logic_vector (24 downto 0); -- counter
  signal r : std_logic; -- invert t when it is H level
  signal t : std_logic; -- T flipflop
begin
  cnt_proc : process (gclk0)
  begin
    if (gclk0'event and gclk0 = '1') then
       -- put code below --

       -- put code above --
    end if;
  end process;

  t_proc : process (gclk0)
  begin
    if (gclk0'event and gclk0 = '1') then
       -- put code below --

       -- put code above --
    end if;
  end process;

  led1 <= t;
end rtl;
```

分周回路と T フリップフロップ (のような動作) を組み合わせて作成します。基板上、JP5 のジャンパをショートし、gclk0 の周波数を 32MHz に設定します。

2 カ所のスペースのうち、一つ目には分周回路、二つ目には T フリップフロップを構成します。

分周回路は、 1 クロックだけ H レベルを作るような構成にする必要があります。先ほどの分周回路では 「11」 と比較していたところを、今回は (2^25)-1 、つまり」11111111111111111111111」と比較することになります。このときに r を 1 に、それ以外の値の時は 0 にセットします。

T フリップフロップは単純で、入力信号が 1 であれば反転、0 であればデータを維持します。

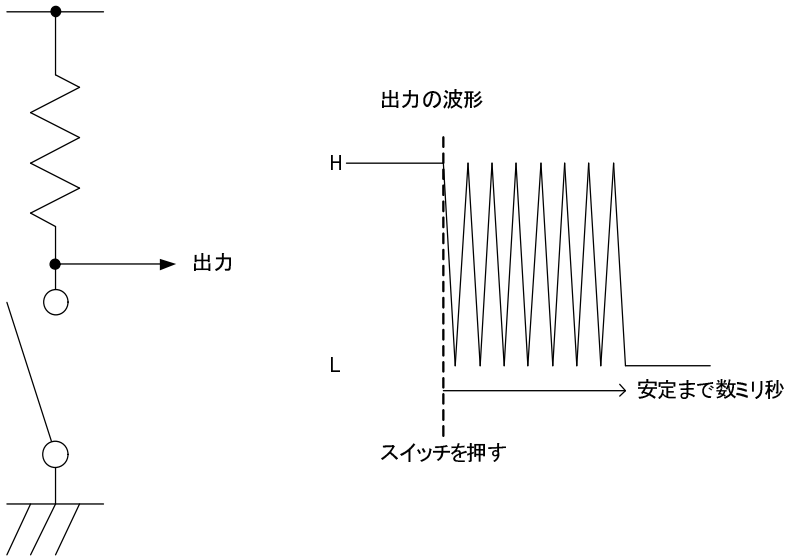
注釈: signal 宣言した信号は、その architecture 内でグローバル変数のように働きます。同じ architecture 内にある全ての process 文内から読み出すことができます。

---

r は、32MHz を 33,554,431 回カウントする毎に 1 になるため、そのたびに t が反転されます。それを led1 に出力すれば、目的の回路ができあがります。

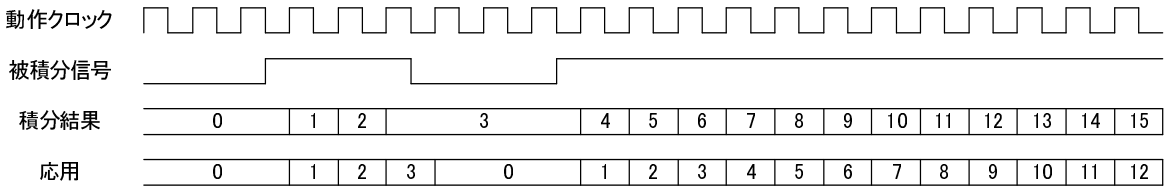
5.10 積分回路とチャタリング除去

機械的なスイッチは、切り替えたときに信号の状態が不安定な期間があります。



この不安定な期間は数 ms 程度続く場合もあります。人間にとっては感知できないほど短い時間ですが、FPGA は動作が十分速いため、この細かい変化を検出し、反応してしまいます。たとえば課題 vhdl06 ではこれにより不安定な動作となっています。

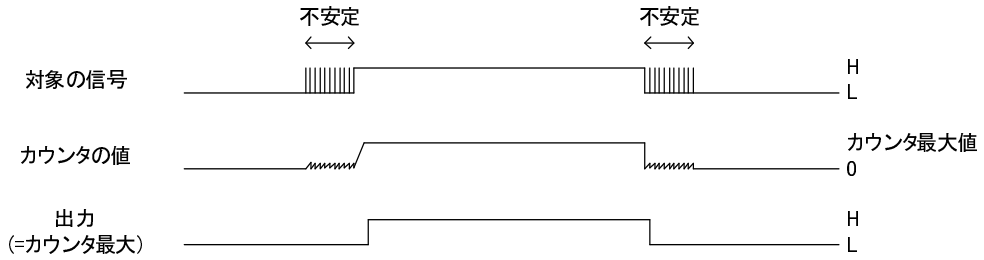
そのため、それを除去する回路を組み込む必要があります。これには積分回路を応用します。



積分回路は基本的にはカウンタです。入力を単純に加算していきます。これを応用し、H が入力されている間は上限まで加算 (上限まで来たら値を維持) 、一度でも L が来たら値をクリアします。

この応用した回路に、スイッチからの信号を入力します。スイッチを切り替えない期間では、カウンタの値は 0 か最大のいずれかで安定します。

スイッチを切り替えた時、入力は H と L が激しく切り替わります。応用した回路では入力の H が一定期間維持されると H を出力するよう動作しますので、この「一定期間」が数ミリ秒となるようなカウンタの最大値を設定すれば良いことになります。



応用回路はたとえば以下のようなソースになります。clock は積分していくサイクル、sw1 がチャタリングのあるスイッチからの入力、r がチャタリング除去後の出力です。

```
process (clock)
begin
  if (clock'event and clock = '1') then
    if (sw1 = '0') then
      c <= "000000000000000";
    elsif (c /= "111111111111111") and (sw1 = '1')) then
      c <= c + 1;
    end if;

    if (c = "111111111111111") then
      r <= '1';
    else
      r <= '0';
    end if;
  end if;
end process;
```

最初の if 文では、カウンタによる積分回路を構成しています。

スイッチを 0 側に倒されたら (sw1 = 『0』) カウンタの値は即 0 とします。

スイッチが 1 側に倒されたら (sw1 = 『1』) 、カウンタがいっぱいでなければ (c /=「111111111111111」、」/=」は」一致していない」 の意味) カウントします (c <= c + 1) 。

このカウンタは、スイッチが 1 側に倒されれば、カウンタのビットがすべて 1 になるまでカウントを続け、0 側に倒されればすぐにすべて 0 に戻します。

スイッチからの信号が不安定な状態では、少しでも 0 がくればカウンタも 0 に戻されますが、十分安定すれば最後までカウントします。

32MHz であれば 1 周期は約 31ns なので、15bit で 32000 回カウントすれば約 1ｍｓになります。スイッチが約 1ｍｓ安定すれば、最後までカウントされることになります。

次の if 文では、前のカウンタが最後までカウントされたら 1、それ以外は 0 を出力します。スイッチ入力が安定したことを判定することができます。

このままでは 1ｍｓしか対応できないため、チャタリングが除去し切れていないように見える場合はカウンタの

ビットを追加する必要があります。

注釈: std_logic_vector の桁数が、比較演算子の左右で一致していない場合、正しく比較されない場合があります。思ったように動作しない場合は、桁数も確認してください。

5.10.1 シミュレーション

チャタリング除去の動作をシミュレーションで確認します。

D:\Altera\(学籍番号) に vhdl21 というフォルダを作ります。

Web ブラウザで http://vhdl.cottonrose.jp/2016/ を開き、vhdl21.vhd をダウンロードし、先ほどのフォルダにダウンロードします。

注釈: ダウンロードしたファイルが読み取り専用になっていることがあります。

別冊「Altera 開発環境（QuartusII）使用マニュアル」の 21 ページからのシミュレーションの手順を行います。

signal の s_switch が、人間がイメージしているスイッチの ON/OFF です。

sw1 は FPGA からみた入力信号を模擬しています。

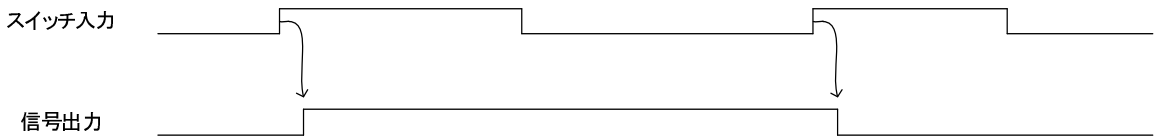
r が、チャタリング除去後の信号で、s_switch よりは時間が短いものの、sw1 のような信号のばたつきが無くなっていることが分かります。

シミュレーション完了後、Wave のエリアで signal c に対し右クリック → Radix → Unsigned 、もう一度右クリック → Format → Analog (Automatic) で、チャタリング除去に使用したカウンタの値の変化を違った形で見ることができます。チャタリングが出ているときは、ここでは 1 までしかカウントできず直ぐに 0 に戻されていることが分かります。

5.11 微分回路

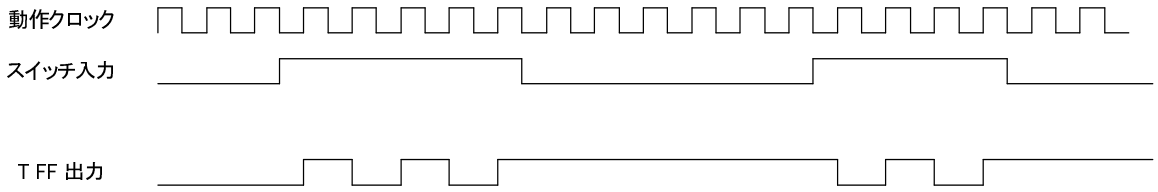
人間や機械からの信号= 遅い信号と高速な内部の回路とのインタフェースをしやすくする回路です。

たとえばスイッチを動かす度に LED の点灯、消灯を切り替えたいとします。タイミングチャートは以下のようなものです。

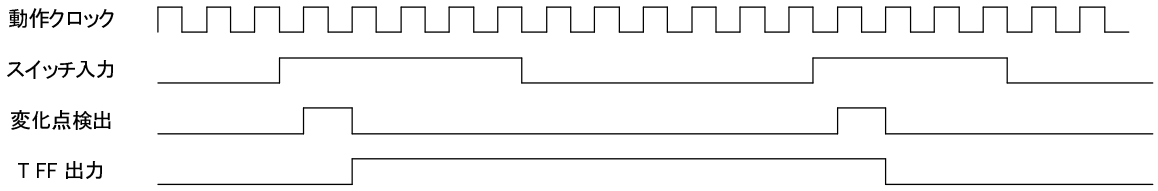


スイッチ入力をクロックとして T フリップフロップに入力できれば実現できますが、同期動作ではないためトラブルの原因になりやすく、採用できません。

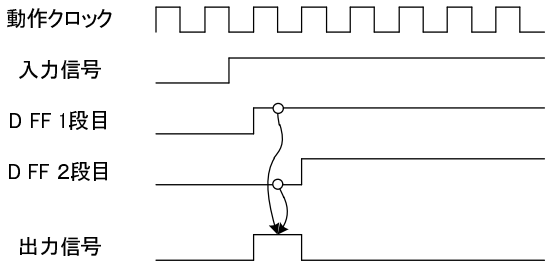
出力の信号を生成しているフリップフロップのクロックが十分遅ければやはり T フリップフロップで対応できそうですが、今回のボードに限らず通常は MHz 級の動作周波数ですのでそのような動作はできません。



スイッチ入力の変化点を検出する回路ができれば、T フリップフロップでも望みの動作が可能になります。

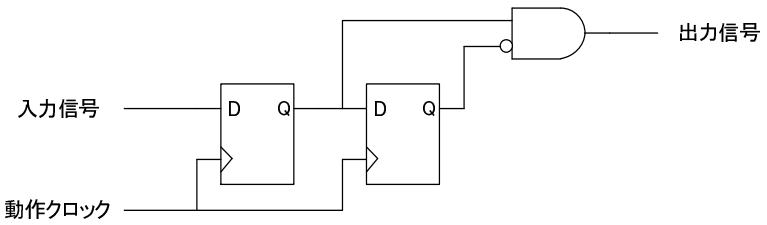


変化点を検出するため、微分回路と呼ばれる回路を使用します。



原理は簡単で、フリップフロップ２段を通して１クロックずつ遅延した信号の間の特定の差だけを抽出します。

実際の回路は以下のようになります。



5.11.1 演習

プロジェクト名 vhdl10

スイッチを往復させる毎に LED の点灯、消灯を切り替える回路を作る。

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity vhdl10 is
  port (
    gclk0 : in std_logic;
    sw1   : in std_logic;
    led1  : out std_logic
  );
end vhdl10;

architecture rtl of vhdl10 is
  signal c  : std_logic_vector (14 downto 0);
  signal r  : std_logic;
  signal d1 : std_logic;
  signal d2 : std_logic;
  signal s  : std_logic;
  signal t  : std_logic;
begin
  -- chattering cancel
  sum_proc : process (gclk0)
  begin
    if (gclk0'event and gclk0 = '1') then
      if (sw1 = '0') then
        c <= "000000000000000";
      elsif ((c /= "111111111111111") and (sw1 = '1')) then
        c <= c + 1;
      end if;

      if (c = "111111111111111") then
        r <= '1';
      else
        r <= '0';
      end if;
    end if;
  end process;

  -- difference
  diff_proc : process (gclk0)
  begin
    if (gclk0'event and gclk0 = '1') then
      d2 <= d1;
      d1 <= r;
    end if;
  end process;
  s <= d1 and (not d2);
```

```
  -- T flipflop
  t_proc : process (gclk0)
  begin
    if (gclk0'event and gclk0 = '1') then
      if (s = '1') then
        t <= not t;
      end if;
    end if;
  end process;

  led1 <= t;
end rtl;
```

diff proc とその直後の s への代入が微分回路になります。

5.12 １０進カウンタ

5.12.1 課題

プロジェクト名 vhdl11

０～９まで、１秒間に１ずつカウントする、１０進カウンタを作れ。カウント値は９の次は０に戻る。signal r は１秒に１回、１サイクルだけ H レベルになる信号となる。led1 が点滅する場合は誤りがある。

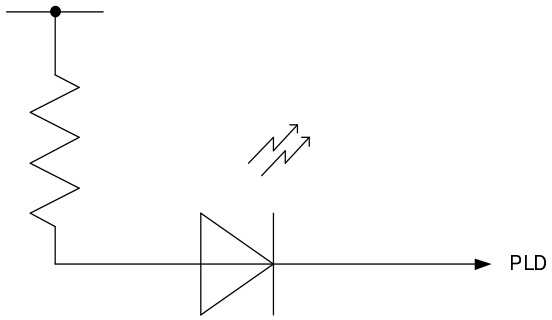
プロセス文 div_proc が１秒を作る。それを利用してプロセス文 cnt_proc 内で signal c で０～９をカウントする。

```
library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity vhdl11 is
  port (
    gclk0  : in std_logic;
    led1   : out std_logic;
    sled1a : out std_logic;
    sled1b : out std_logic;
    sled1c : out std_logic;
    sled1d : out std_logic;
    sled1e : out std_logic;
    sled1f : out std_logic;
    sled1g : out std_logic
  );
end vhdl11;

architecture rtl of vhdl11 is
```

```
  signal d    : std_logic_vector (24 downto 0);
  signal r    : std_logic;
  signal c    : std_logic_vector (3 downto 0);
  signal sled : std_logic_vector (6 downto 0);
  signal ck   : std_logic;
begin
  div_proc : process (gclk0)
  begin
    if (gclk0'event and gclk0 = '1') then
      if (d = "1111010000100011111111111") then
        r <= '1';
        d <= "0000000000000000000000000";
      else
        r <= '0';
        d <= d + 1;
      end if;
    end if;
  end process;

  cnt_proc : process (gclk0)
  begin
    if (gclk0'event and gclk0 = '1') then
       -- put code below --


       -- put code above --
    end if;
  end process;

  chk_proc : process (gclk0)
  begin
    if (gclk0'event and gclk0 = '1') then
      if ((c = "1010") and (r = '1')) then
        ck <= not ck;
      end if;
    end if;
  end process;

  sled <= "0010000" when (c = "1001")
     else "0000000" when (c = "1000")
     else "1111000" when (c = "0111")
     else "0000010" when (c = "0110")
     else "0010010" when (c = "0101")
     else "0011001" when (c = "0100")
     else "0110000" when (c = "0011")
     else "0100100" when (c = "0010")
     else "1111001" when (c = "0001")
     else "1000000" when (c = "0000")
     else "0000110";
```

```
  sled1a <= sled (0);
  sled1b <= sled (1);
  sled1c <= sled (2);
  sled1d <= sled (3);
  sled1e <= sled (4);
  sled1f <= sled (5);
  sled1g <= sled (6);

  led1 <= ck;
end rtl;
```

7セグメント LED の点灯パターンはコードの通りになる、LED の配置は図のようになっています。port に L を出力すると、電位差で電流が流れ LED が点灯します。



5.13 階層設計

規模が大きくなってくると、すべてのコードを一つの architecture に書ききることが問題になります。そのためコードを機能ブロック毎に分割し呼び出すことができるようになっています。例を以下に示します。

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity Y_child is
  port (
  ～ PORT_DEFINITION(CHILD) ～
  );
end Y child ;

architecture A_Y_child of Y_child is
  ～ SIGNAL_DEFINITION(CHILD) ～
begin
  ～ EXPRESSIONS(CHILD) ～
end A Y chi ld ;
```

```
library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity X_parent is
  port (
    〜 PORT_DEFINITION(PARENT) 〜
  );
end X_parent;

architecture A_X_parent of X_parent is

  component Y_child
    port (
      〜 PORT_DEFINITION(CHILD) 〜
    );
  end component;

  〜 SIGNAL_DEFINITION(PARENT) 〜

begin

  i_Y_child : Y_child
    port map (
      〜 PORT_CONNECTION(CHILD PARENT) 〜
    );

  〜 OTHER_EXPRESSIONS(PARENT) 〜

end A_X_parent;
```

処理を抜き出した、他から呼び出される側の entity は、通常通り作成します。

処理を呼び出す場合は２段階の手続きがあります。

まず architecture から begin までの間で、component 宣言を行います。ここで、呼び出す entity はすべて宣言します。

次に begin の後で、実際に呼び出します。複数個呼び出すこともできますが、その場合は entity 名の前のコロンの前、インスタンス名 (ここでは i_Y_child) はそれぞれ固有のものにします。

5.13.1 演習

１０進数カウンタに階層化を適用します。

注釈: counter10.vhd と ledconv.vhd は、vhdl11b プロジェクトを作成した後、それぞれ vhdl11b.vhd と同様に file → new → VHDL file で作成します。

counter10.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity counter10 is
  port (
    gclk0 : in std_logic;
    sw2 : in std_logic;

    e_in : in std_logic;
    c_in : in std_logic;

    c_out : out std_logic;

    cnt : out std_logic_vector (3 downto 0)
) ;
end counter10;

architecture rtl of counter10 is
  signal c : std_logic_vector (3 downto 0);
begin
  cnt_proc : process (sw2, gclk0)
  begin
    if (sw2 = '1') then
      c <= "0000";
    elsif (gclk0'event and gclk0 = '1') then
      if ((e_in = '1') and (c_in = '1')) then
        -- put code below --

        -- put code above --
      end if;
    end if;
  end process;

  c_out <= '1' when (c = "1001")
      else '0';
  cnt <= c;
end rtl;
```

c_in は下の桁からの桁上げのリクエストを受け付けるポート、c_out は上の桁への桁上げのリクエストを出力するポートです。

e_in は H レベルが入るとカウントを行うポートです。

```
if ((e_in = '1') and (c_in = '1')) then
```

の箇所は、これまでのカウンタのソースでの r = 『1』 に相当します。

ledconv.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity ledconv is
  port (
    cnt : in std_logic_vector (3 downto 0);
    sledxa : out std_logic;
    sledxb : out std_logic;
    sledxc : out std_logic;
    sledxd : out std_logic;
    sledxe : out std_logic;
    sledxf : out std_logic;
    sledxg : out std_logic
  );
end ledconv;

architecture rtl of ledconv is
  signal sled : std_logic_vector (6 downto 0);
begin
  sled <= "0010000" when (cnt = "1001")
     else "0000000" when (cnt = "1000")
     else "1111000" when (cnt = "0111")
     else "0000010" when (cnt = "0110")
     else "0010010" when (cnt = "0101")
     else "0011001" when (cnt = "0100")
     else "0110000" when (cnt = "0011")
     else "0100100" when (cnt = "0010")
     else "1111001" when (cnt = "0001")
     else "1000000" when (cnt = "0000")
     else "0000110";
  sledxa <= sled (0);
  sledxb <= sled (1);
  sledxc <= sled (2);
  sledxd <= sled (3);
  sledxe <= sled (4);
  sledxf <= sled (5);
  sledxg <= sled (6);
end rtl;
```

sledxa から sledxg は 7 セグ LED の各端子に相当します。vhdl11b では 1 桁のみ使うため、sled1a 〜 sled1g と対応させています。これが 4 桁になる場合、ledconv を 4 個用意し、sled1a 〜 sled1g から sled4a 〜 sled4g までに対応させます。

プロジェクト名 vhdl11b

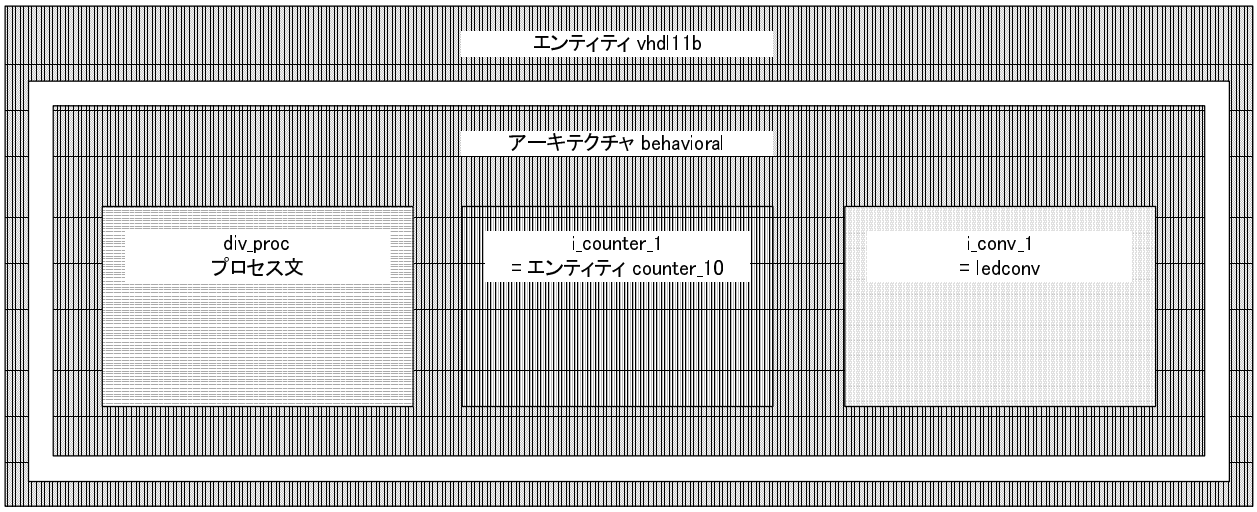
vhdl11b.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity vhdl11b is
  port (
    gclk0  : in std_logic;
    sw2    : in std_logic;
    sled1a : out std_logic;
    sled1b : out std_logic;
    sled1c : out std_logic;
    sled1d : out std_logic;
    sled1e : out std_logic;
    sled1f : out std_logic;
    sled1g : out std_logic
  );
end vhdl11b;

architecture rtl of vhdl11b is
  component counter10
    port (
      gclk0 : in std_logic;
      sw2   : in std_logic;

      e_in  : in std_logic;
      c_in  : in std_logic;

      c_out : out std_logic;

      cnt   : out std_logic_vector (3 downto 0)
    );
  end component;

  component ledconv
    port (
      cnt    : in std_logic_vector (3 downto 0);
      sledxa : out std_logic;
      sledxb : out std_logic;
      sledxc : out std_logic;
      sledxd : out std_logic;
```

```
      sledxe : out std_logic;
      sledxf : out std_logic;
      sledxg : out std_logic
    );
  end component;

  signal d      : std_logic_vector (24 downto 0);
  signal r      : std_logic;
  signal c_in1  : std_logic;
  signal c_out1 : std_logic;
  signal cnt1   : std_logic_vector (3 downto 0);
begin
  div_proc : process (gclk0)
  begin
    if (gclk0'event and gclk0 = '1') then
      if (d = "11110100001000011111111111") then
        d <= "00000000000000000000000000";
        r <= '1';
      else
        d <= d + 1;
        r <= '0';
      end if;
    end if;
  end process;

  c_in1 <= '1';

  i_counter1 : counter10
    port map (
      gclk0 => gclk0,
      sw2   => sw2,
      e_in  => r,
      c_in  => c_in1,
      c_out => c_out1,
      cnt   => cnt1
    );

  i_conv1 : ledconv
    port map (
      cnt    => cnt1,
      sledxa => sled1a,
      sledxb => sled1b,
      sledxc => sled1c,
      sledxd => sled1d,
      sledxe => sled1e,
      sledxf => sled1f,
      sledxg => sled1g
    );
```

```
end rtl;
```

この状態で、ソースコードの関係は以下のようになります。



counter10 を呼び出しているときのキーワード 「i_counter1」 や ledconv に対する 「i_conv1」 がインスタンス名となります。固有のインスタンス名をつけることで、同じ回路ブロックを複数回呼び出すことができます。

ポートの接続はこのコードのように、左辺に呼び出される回路のポート名、=> をはさんで右辺に呼び出し側の port 名または signal 名を書き、カンマで区切ります。カンマは区切りなので最後の接続の後には書きません。

5.13.2 シミュレーション & 演習

課題: タイミングチャートの読み方がわからない

先ほどの階層設計を応用して、10 進数 3 桁のカウンタを作り、シミュレータ上で動作を確認します。

D:\Altera\(学籍番号) に vhdl22 というフォルダを作ります。

Web ブラウザで http://vhdl.cottonrose.jp/2016/ を開き、vhdl22.vhd をダウンロードし、先ほどのフォルダにダウンロードします。

注釈: ダウンロードしたファイルが読み取り専用になっていることがあります。

---

vhdl11b で使った counter10.vhd も、このフォルダにコピーしておきます。

別冊「Altera 開発環境（QuartusII）使用マニュアル」の 21 ページからのシミュレーションの手順を行います。

途中のコンパイルでは **先に counter10.vhd をコンパイル** しておきます。

このソースコードは、そのままでは 10 進数のカウンタが 3 個並び、同じようにカウントするだけになっています。

これを **改造** し、10 進数 3 桁のカウンタにしてください。

ソースコードの抜粋を以下に示します。

```
-- COUNTERS ---------------------------------------------------------------

i_counter_0: counter10
  port map (
      gclk0 => gclk0,
      sw2   => sw2,
      e_in  => s_e_in_to_0,
      c_in  => s_c_in_to_0,
      c_out => s_c_out_from_0,
      cnt   => s_cnt_0
  );

i_counter_1: counter10
  port map (
      gclk0 => gclk0,
      sw2   => sw2,
      e_in  => s_e_in_to_1,
      c_in  => s_c_in_to_1,
      c_out => s_c_out_from_1,
      cnt   => s_cnt_1
  );

i_counter_2: counter10
  port map (
      gclk0 => gclk0,
      sw2   => sw2,
      e_in  => s_e_in_to_2,
      c_in  => s_c_in_to_2,
      c_out => s_c_out_from_2,
      cnt   => s_cnt_2
  );

-- edit connection here ---------------------------------------------------

s_e_in_to_0 <= r;
s_c_in_to_0 <= '1';
-- s_c_out_from_0

s_e_in_to_1 <= r;
s_c_in_to_1 <= '1';
-- s_c_out_from_1
```

```
s_e_in_to_2 <= r;
s_c_in_to_2 <= '1';
-- s_c_out_from_2


--------------------------------------------------------------------------------
```

ソースコード中の edit connection here にある 9 つの signal の接続を編集することで実現できます。

counter10 の e_in と c_in は、両方に 1 が入力されるとカウントするように働きます。c_out は、カウント値が 9 の時に 1 になります。

counter10 のポートの役割は以下の通りです。

gclk0　クロック入力

sw2　リセット入力

e_in　カウントしたいタイミングの入力

c_in　下の桁からの桁上げ信号の入力

c_out　上の桁への桁上げ信号の出力

3 桁分の counter10 から合計 9 つの signal が引き出されています。

一度シミュレータでこれらの信号の動きを確認し、要求通りの動作をするように、signal の接続を変更してください。

シミュレーション用のコードは判定機能を持っています。

Transcript エリアに 「Simulation finished with 0 errors.」 と表示されると、正しく動作したと判定されます。

もし数字が 0 でなかったり、」Simulation timed out with * **errors.」** (* は数字) と表示されたら、何かしらのエラーがあることになります。

5.14 状態遷移と、条件分岐 case 〜 when 〜

例えばカップラーメンを食べようとした場合でも、 (誰かが作ってくれる場合を除いて) 手順があります。

1. カップラーメンの包装をといてふたを開けて小袋をとりだす。必要なものはここで入れる。
2. やかんに水を入れてコンロにかけて沸かす。
3. 沸騰するのを待つ。
4. 沸騰したら火を止めて、お湯をカップに注ぐ。
5. ３分 (または５分) 待つ。

6. 必要に応じて小袋の中身をカップに入れる。まぜる。

7. 食べてよし。

小袋が入ってなかったり電気ポットでお湯を沸かしたり、湯切りタイプだったりとバリエーションがありますが、オーソドックなものではこのように順番があり、例えば１、２は入れ替えたり並列に行ってもいいですが、基本的に手順通りに行います。

プログラミングもこういったところに似ていますが、このように手順通りに処理を行いたい場合、ソフトウェアと同様状態遷移の考え方を使います。

状態遷移は通常は図で状態間のつながりを描きながら設計をし、ソースコードには人間が考えながら書き込むしかありません (図を描くことでソースコードを出力してくれるツールもあります) 。

ソースコード上で、各「状態」を数字で表しながらコーディングすることもできますが、各「状態」に名前をつけて管理しやすくすることができます。

(C 言語の enum と同じような働きです)

使い方はたとえば以下のようになります。

```
type RAMEN is (MAKE_READY_CUP, BOILWATER, POURWATER, WAIT_MINUTES, READY_TO_EAT);
signal ramen_stat : RAMEN;
begin
  ramen_stat <= MAKEREADY;
```

１行目では RAMEN という型を新しく定義しています。この型のとりうる値は」MAKE_READY_CUP」以降の５種類です。

２行目では、RAMEN の型の signal 、ramen_stat を宣言します。

ramen_stat には MAKE_READY_CUP など、定義した名前を代入したり、if 文で比較したりすることができます。ただし勝手に作った型ですので、そのままでは port から出力しても使えませんし、例えば LED の点灯パターンに対しては全く対応がとれないため使えません。あくまで内部で使うのが基本です。

このようにしなくても、自分でこれらの状態を管理すれば、たとえば integer の signal ででも管理できます。ただしわかりやすい名前をつけることで、ソースコードが理解しやすくなります。

5.15 条件分岐 case ～ when ～

注釈: case 文は process 文の中でのみ使用できます。

状態遷移専用ではありませんが、case 文はよく組み合わされて使われます。基本的な形は以下の通りです。

```
case (CONDITION_SIGNAL) is
  when (VALUE_A) => 〜 EXPRESSION A 〜
  when (VALUE_B) => 〜 EXPRESSION B 〜
  when others => 〜 EXPRESSION OTHER 〜
end case;
```

ある一つの signal の値毎に処理を分岐させることができます。処理は、次の when まで何行でも書くことができます。

5.15.1 演習

スイッチによって、特定の LED を点灯させる回路を作る。* LED1 が点灯している場合、sw2 を操作することで LED1 が消灯し、LED2 が点灯する。* LED2 が点灯している場合、sw1 を操作することで LED2 が消灯し、LED1 が点灯する。sw2 を操作した場合は、LED2 が消灯し、LED3 が点灯する。* LED3 が点灯している場合、sw1 を操作することで LED3 が消灯し、LED2 が点灯する。

swfilter.vhd

```
library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity swfilter is
  port (
    gclk0 : in std_logic;
    sw : in std_logic;
    sw_out : out std_logic
  );
end swfilter;

architecture rtl of swfilter is
  signal c : std_logic_vector (14 downto 0);
  signal r : std_logic;
  signal d1 : std_logic;
  signal d2 : std_logic;
  signal s : std_logic;
begin
  sum_proc : process (gclk0)
  begin
    if (gclk0'event and gclk0 = '1') then
      if (sw = '0') then
        c <= "000000000000000";
      elsif ((c /= "111111111111111") and (sw = '1')) then
        c <= c + 1;
      end if;
```

```
      if (c = "11111111111111") then
        r <= '1';
      else
        r <= '0';
      end if;
    end if;
  end process;

  diff_proc : process (gclk0)
  begin
    if (gclk0'event and gclk0 = '1') then
      d1 <= r;
      d2 <= d1;
      s <= d1 and (not d2);
    end if;
  end process;

  sw_out <= s;
end rtl;
```

プロジェクト名 vhdl12

vhdl12.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity vhdl12 is
  port (
    gclk0 : in std_logic;
    sw1   : in std_logic;
    sw2   : in std_logic;
    led1  : out std_logic;
    led2  : out std_logic;
    led3  : out std_logic
  );
end vhdl12;

architecture rtl of vhdl12 is
  component swfilter
    port (
      gclk0  : in std_logic;
      sw     : in std_logic;
      sw_out : out std_logic
    );
  end component;
```

```
  signal sw1d : std_logic;
  signal sw2d : std_logic;

  type LED_STAT is (P_LED1, P_LED2, P_LED3);
  signal lstat : LED_STAT;

  signal led : std_logic_vector (3 downto 1);
begin
  sw1_filter : swfilter
    port map (
      gclk0  => gclk0,
      sw     => sw1,
      sw_out => sw1d
    );

  sw2_filter : swfilter
    port map (
      gclk0  => gclk0,
      sw     => sw2,
      sw_out => sw2d
    );

  stat_proc : process (gclk0)
  begin
    if (gclk0'event and gclk0 = '1') then
      case lstat is
        when P_LED1 =>
          if (sw2d = '1') then
            lstat <= P_LED2;
          end if;
          led <= "001";

        when P_LED2 =>
          if (sw1d = '1') then
            lstat <= P_LED1;
          elsif (sw2d = '1') then
            lstat <= P_LED3;
          end if;
          led <= "010";

        when P_LED3 =>
          if (sw1d = '1') then
            lstat <= P_LED2;
          end if;
          led <= "100";

        when others =>
          lstat <= P_LED1;
```

```
          led <= "111";
      end case;
    end if;
  end process;

  led1 <= not led (1);
  led2 <= not led (2);
  led3 <= not led (3);
end rtl;
```

case 文とは関係ありませんが、swfilter は sw にスイッチからの信号を接続し、sw out にチャタリング除去後の信号を出力する回路ブロックです。

5.16 ストップウォッチ

1/100 秒単位のストップウォッチを実装する。

表示は 4 桁の 7 セグ LED で行い、１０進数のカウントで、99.99 秒までカウントできること。

5.16.1 課題：最小限の構成

ラップ・スプリット機能の無いストップウォッチを実装する。

操作は 2 つのスイッチで行う。

スタート・ストップスイッチ　往復させる毎にカウント動作とストップを切り替える。

　カウント動作中にスタート・ストップスイッチを往復させてストップした後、再度スイッチを往復させると、続きからカウント動作を行う。

リセットスイッチ　往復させるとカウントしていた値をクリアする。

　カウント動作中にリセットスイッチを往復させた場合の動作は定義しない (どのように動作してもよい) 。

99.99 秒の次のカウントは未定義 (どのように動作してもよい) 。

5.16.1.1 ヒント

vhdl11b に 10 進カウンタ counter10 、7 セグ LED への変換を行う ledconv がひとつずつあって 1 桁の 10 進カウンタができている。vhdl22 の要領で、これを 2 組、3 組、4 組と増やしていき、4 桁の 10 進カウンタを作るのがよい。いきなり 4 桁に挑戦しても躓くことが多い。

1 秒毎にカウントしているので、これを 1/100 毎にカウントするとよい。

スイッチの入力には、vhdl12 のチャタリング除去回路 swfilter が応用できる。

スタート・ストップの切り替えは vhdl10 が応用できる。

今カウント中なのか止まっているのか、判定する signal を作って制御する。ついでにそれを LED に表示するとわかりやすい。

お試しで signal を作るのにためらいは要らない。それを LED に表示しても誰もとがめない。

時々、作った VHDL ファイルを別名でとっておくと良い。内容が把握できなくなったときは、良くない方向に進んでいる可能性が高いので、前の状態からやり直してみる勇気も必要。

5.16.2 課題：応用

ラップ、またはスプリット、または両方の機能のついたストップウォッチを実装する。

操作は２つのスイッチで行う。

スタート・ストップスイッチ　往復させる毎にカウント動作とストップを切り替える。

カウント動作中にスタート・ストップスイッチを往復させてストップした後、再度スイッチを往復させた場合、続きからカウント動作を行う。

ラップ/スプリット・リセットスイッチ　往復させたときの動作状況に合わせて、このスイッチの動作も変化する。

カウントがストップしている時に往復させるとリセット動作として、カウントしていた値をクリアする。

カウント動作中に往復させるとスプリット状態に移る。表示している値は止めるが、カウント動作は継続する。

スプリット状態で再度往復させると、スプリット状態が解除され、カウントしている値の表示を再開する。

スプリット状態でスタート・ストップスイッチを往復させると、表示している値は維持したまま、カウント動作を止める。

スプリット状態 → カウント停止からこのこのスイッチを往復させると、カウントがストップした値を表示する（スプリット状態が解除される)。

この状態でもう一度このスイッチを往復させると、カウントはクリアする (リセット) その前にスタートのスイッチを往復させると、その値からカウントを再開する。

ラップの場合、ボタンを往復させると表示している値をとめ、カウントは０から再開する。

ラップ機能、スプリット機能を両方実装する場合は、スイッチの一つをモードの切り替えに割り振る。

99.99 秒の次のカウントは 00.00 秒とする。

5.16.2.1 ヒント

現在どの状況にいるのかを保持する signal を作り、状態遷移を応用すると良い。

また状況を LED に表示すると良い。

5.16.3 その他課題

タイマを作成する。

表示は、上位２桁が「分」、下位２桁が「秒」とする。

第一段階は、決められた時間 (たとえば３分など) のカウントダウンを行う。０までカウントダウンが完了したらそこで停止する。

リセットするとカウントする時間をリロードする。

第二段階は、カウントする時間を２つから選べるようにする (たとえば３分と５分) 。

最後は、任意の時間を設定できるようにする。

第 6 章

付録

6.1 参考情報

VHDL Language Reference Manual

言語の規格

http://ieeexplore.ieee.org/xpl/freeabs all.jsp?reload=true&arnumber=4772740

VHDL によるハードウェア設計入門

CQ 出版/ 長谷川裕恭著

HDL サンプル記述集

CQ 出版

トランジスタ技術 SPECIAL No. 79 初歩の HDL 設計学習帳

CQ 出版

トランジスタ技術 SPECIAL for フレッシャーズ No. 105 ロジック回路設計はじめの一歩

CQ 出版

例題で学ぶ論理回路設計

森北出版/ 富川武彦著

ディジタルコンピューティングシステム

昭晃堂/ 亀山充隆著