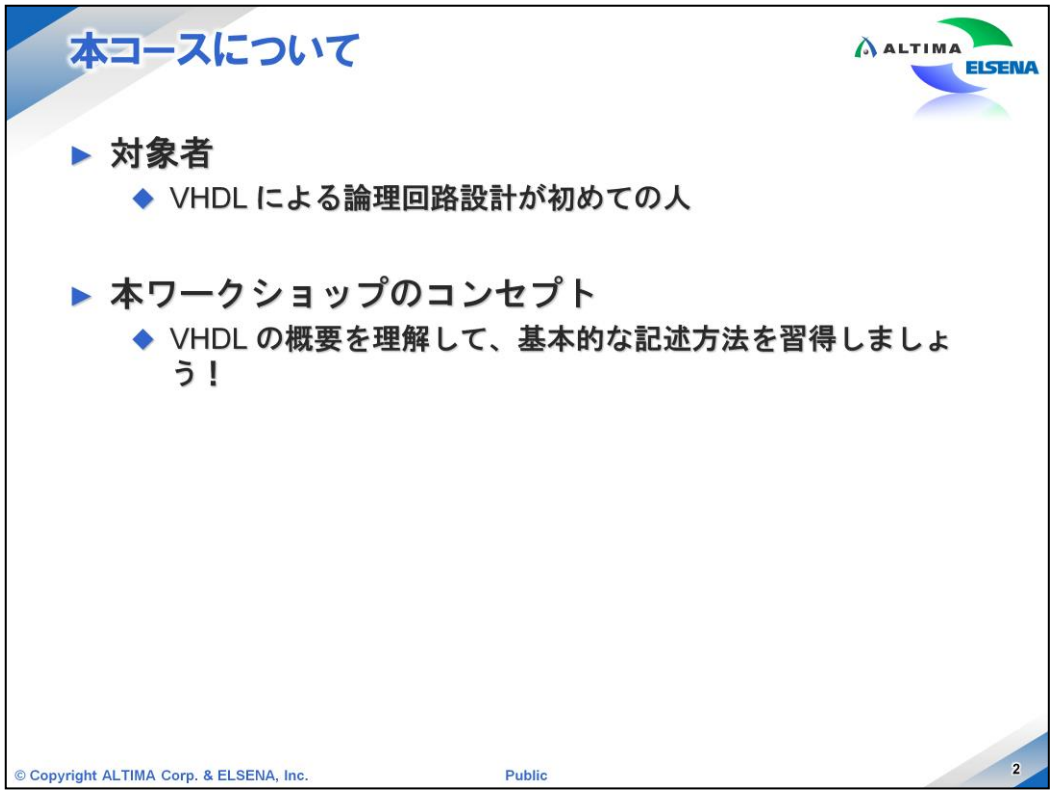
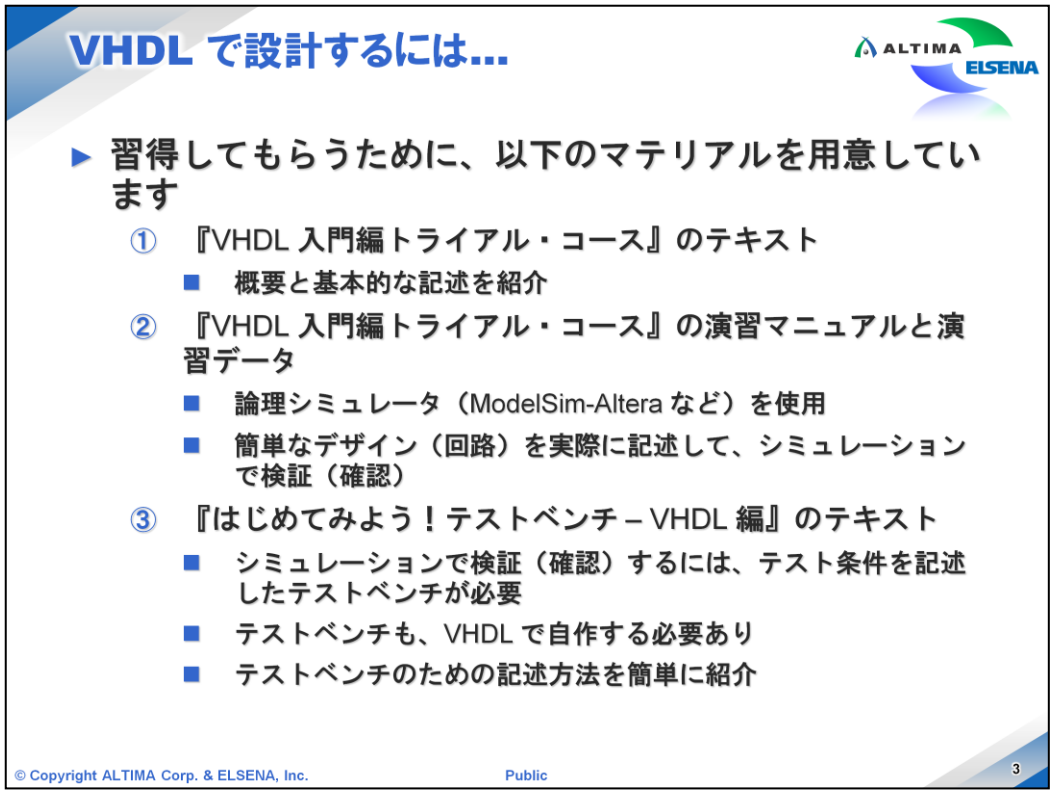
『VHDL 入門編トライアル・コース』を受講して頂き、誠にありがとうございます。



本ワークショップは、VHDL による論理回路設計が初めての方を対象にしています。従って、非常に簡単な内容となっています。

本ワークショックを受講すると、VHDL の概要が理解でき、基本的な記述方法が習得できる内容になっています。



VHDL で設計できるようになるために、以下のマテリアルを用意しています。

「①『VHDL 入門編トライアル・コース』のテキスト」では、概要と基本的な記述を紹介します。

そして、演習を通して理解度を上げてもらうため、「②『VHDL 入門編トライアル・コース』の演習マニュアルと演習データ」を用意しています。

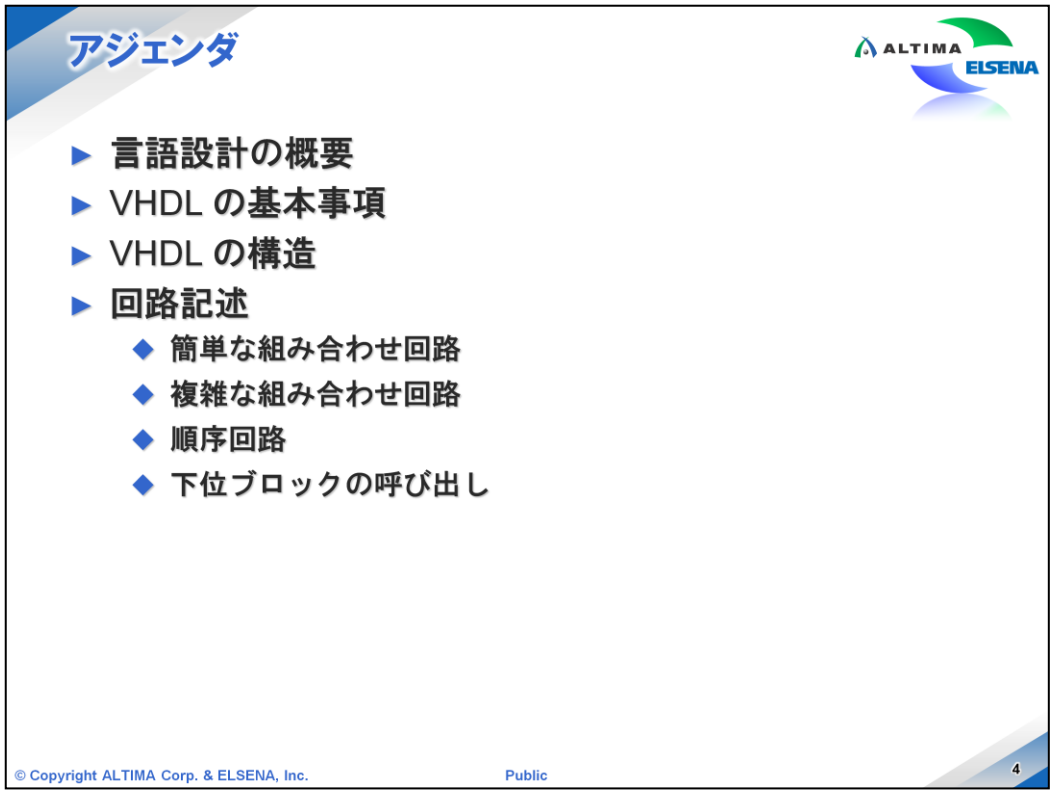
演習を実施するには、ModelSim-Altera などの論理シミュレータが必要です。

アルテラ社の Web サイトから、無償版の ModelSim-Altera Starter Edition を入手することができます。

VHDL で論理回路を記述したら、実機確認する前にシミュレーションで検証をしますが、シミュレーションをするには入力ピンのテスト条件を記述したテストベンチが必要です。

このテストベンチは、基本的に Verilog-HDL で自作する必要があります。

そのため、「③『はじめてみよう！テストベンチ - VHDL 編』のテキスト」でテストベンチのための記述方法を紹介します。



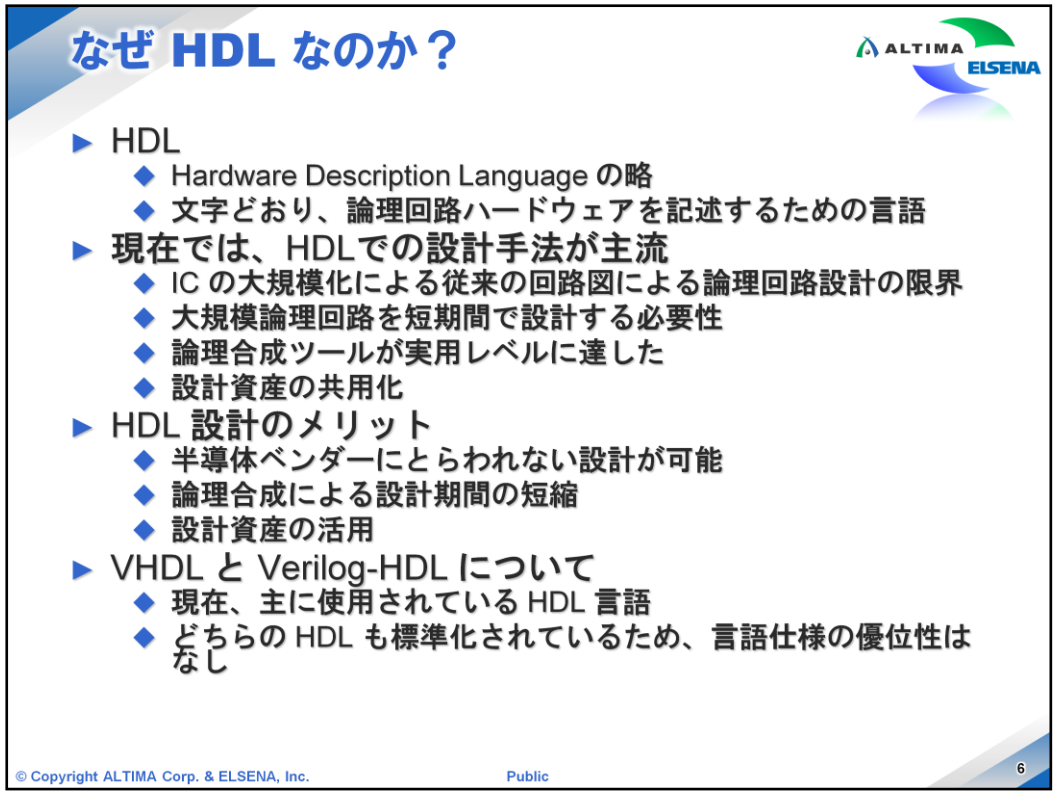
本ワークショップのアジェンダです。

最初に、言語設計の概要を説明し、続けて、VHDL の基本事項と構造を説明します。

その後、実際の回路記述の方法として、簡単な組み合わせ回路、複雑な組み合わせ回路、順序回路を説明します。

最後に、階層設計に必要不可欠な下位ブロックの呼び出しの記述方法を説明します。

次に、VHDL の基本構造がどのようになっているのかを確認しましょう。



HDL は文字どおりハードウェアを記述するための言語です。但しここで言うハードウェアとは論理回路のことです。アナログ回路ではありません。

現在ではデバイスも大規模になり、大規模論理回路を短期間で設計する必要があります。従来の回路図を使用する設計手法では HDL で設計をするのに比較すると多大な労力を必要とするため HDL での設計が主流となっています。

また、回路図で ASIC を開発する場合は、まず ASIC ベンダーとシリーズを決めてから設計作業に入る必要があります。理由は、回路図で使用するライブラリがベンダー／シリーズ毎に異なるためです。

HDL を使用すれば機能設計が終わった後、あるいは FPGA でプロトタイプを動かした後でも選定を変える事が可能となります。

ハードウェア言語として VHDL と並んで主流なのが　Verilog-HDL です。

VHDL は米国国防省によって、VHSIC プログラムで規定された標準言語であり、早い段階から IEEE で標準化されていました。

Verilog-HDL は個人により発案され、Open 環境で磨かれてきた標準言語です。

どちらの言語にも特徴があり、どちらが優れているなどと優劣をつけられるものではありません。現在では、Verilog-HDL も IEEE により標準化され、VHDL、Verilog-HDL どちらも各種合成ツールやシミュレータで対応されています。

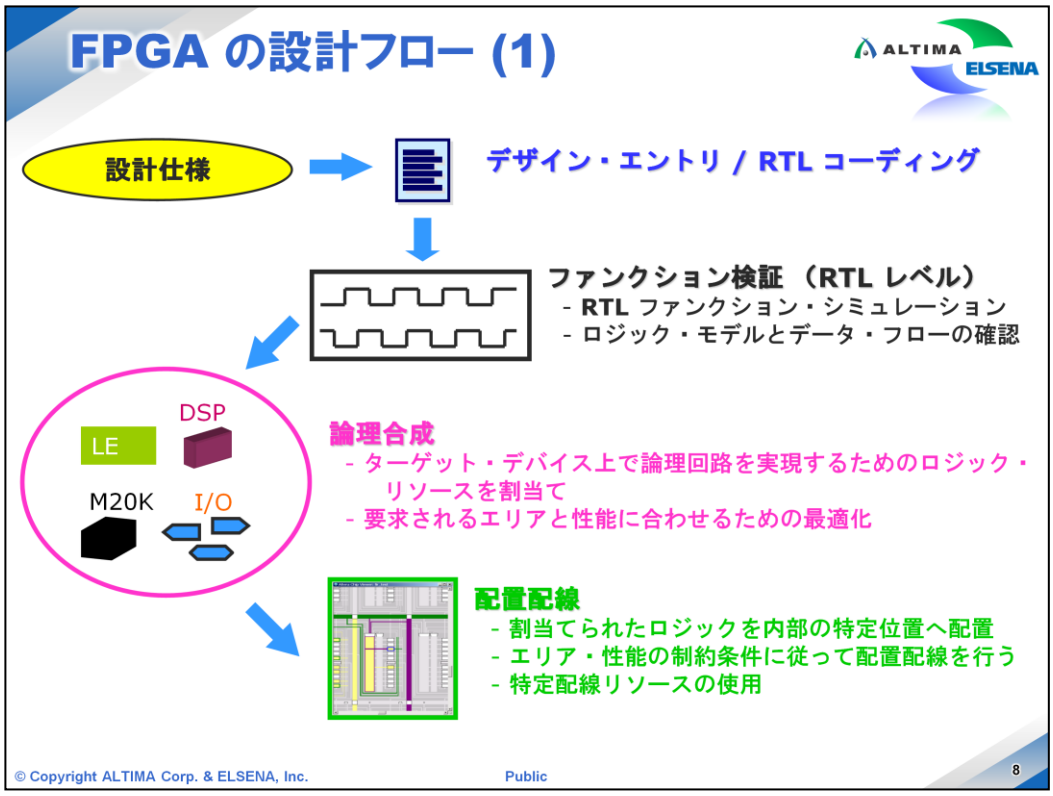
HDL で記述された論理回路（RTL: Register Transfer Level などとも呼ばれます）を実際にデバイス上に構築していくためには、ゲート回路に変換する必要があります。この作業を論理合成と呼び、一般的には、論理合成ツールと呼ばれるソフトウェアが使用されます。

論理合成を行う際には、RTL 記述をゲート回路に変換するほかにも、リソース削減のために冗長な記述の最適化も行います。

論理合成ツールには各種ありますが、いくつかあげるとすると MentorGraphics 社の Precisiton Synthesis や Synopsys 社の Synplify Pro などがあります。

ごれら論理合成ツールを使用して目的とする ASIC や FPGA 用のネットリストを生成します。ネットリストとはゲート回路に変換されたもので、各素子の接続関係がテキスト表現されたものです。

また、アルテラ社 FPGA を使用する場合には、アルテラ社専用の設計ツールである Quartus Prime に独自の論理合成機能を搭載しています。Quartus 内の論理合成ツールを使用しない場合には、汎用の論理合成ツールで生成したネットリストを Quartus Prime に取り込んで設計に使用することも可能です。



一般的な FPGA/CPLD の設計フローを紹介します。

まず、装置やボード・レベル、FPGA/CPLD 内の設計仕様の検討を行った後に、HDL による実際の設計作業に入ります。

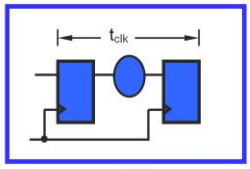
その後、内部遅延を含めない論理的なシミュレーションを行い、想定された動作をしているかどうかを確認します。

もし、想定する動作をしていない場合は、デザイン・エントリーに戻って修正して再度シミュレーションを実施します。

HDL による設計が概ね問題ないと判断できれば、論理構成 → 配置配線 → タイミング検証と進みます。

これらは、ツールが行ってくれます。

FPGA の設計フロー (2)



タイミング検証
- スタティックなタイミング解析（内部動作周波数、I/O タイミング）
- 要求されるタイミング仕様を満たしているかを確認

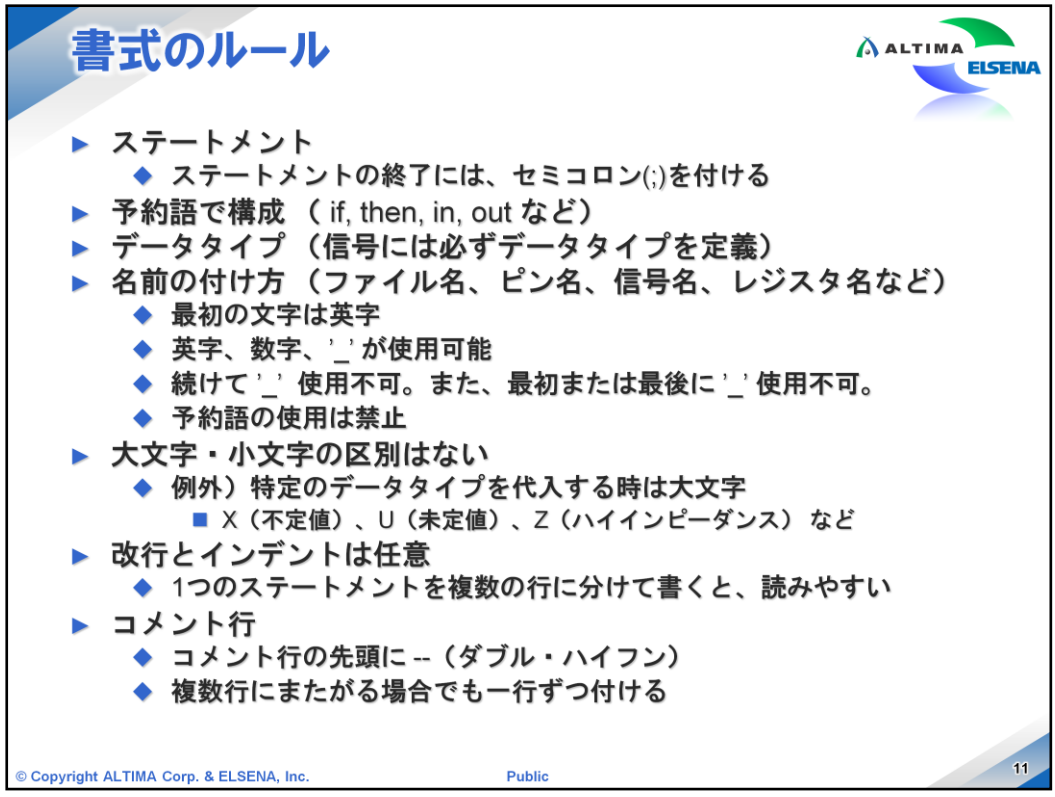
デバイス・プログラミングと
オンチップ・デバッグ
- FPGA/CPLD へのプログラム（書込み）
- ボードに FPGA/CPLD を実装し
  システム・レベルでの動作確認、および
  デバッグ

タイミング検証とは、FPGA/CPLD 内部のスタティックなタイミング解析を指します。

配置配線結果がユーザの満足するタイミング仕様となっているかを検証します。

最後に、ボードに実装されている FPGA/CPLD へデータを書き込んで、実機検証（デバッグ）を行います。

まずは VHDL の書式や予約語など基本ルールを確認しましょう。



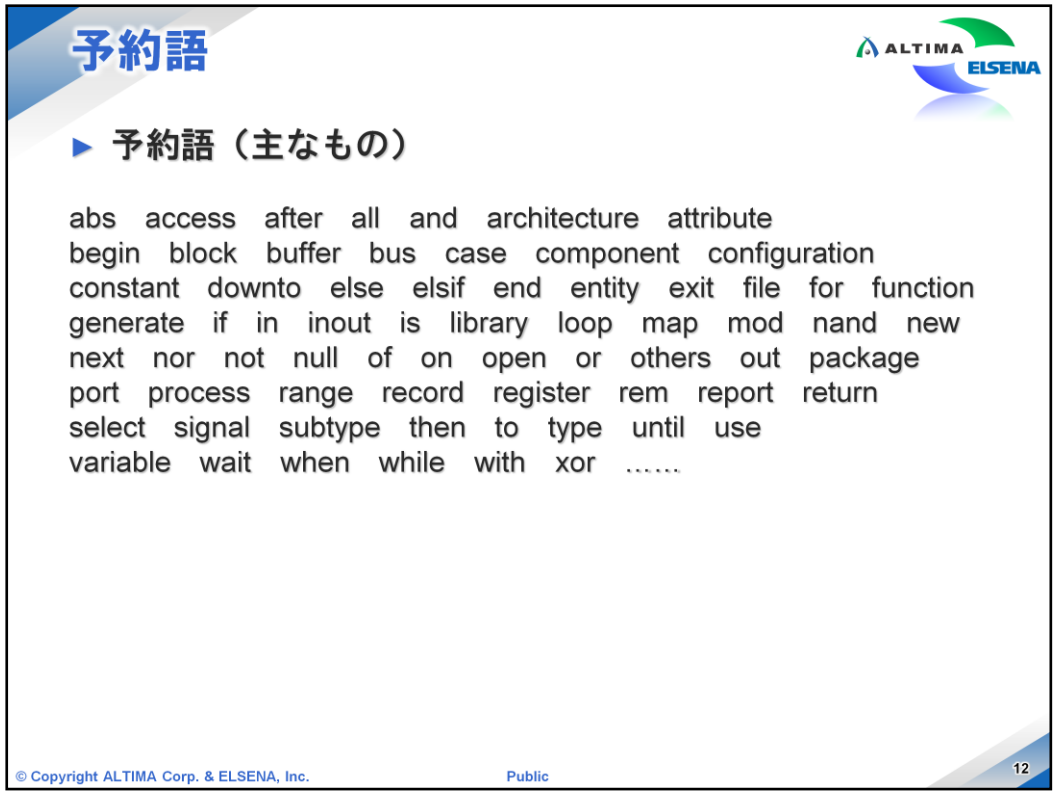
まずは書式ルールについてです。

1つのステートメント(処理)の終了には、セミコロン (;) を付けます。

指定されている予約語を使用します。(if, then, in, out など、次ページに記載しています)

VHDL では、コンポーネント間の接続などの配線 (signal) や各種変数 (variable) などが使用されますが、これらは必ずデータタイプを指定して使用する必要があります。データタイプには、標準で定義されているものや、ユーザが新しくデータタイプを定義して使用することができます。VHDL ではデータタイプを重視した言語であり、同じデータタイプでないとデータ代入することができません。

ファイル名やピン名、信号名、レジスタ名などには、いくつかルールがあります。最初の文字は必ず英字で始めるようにしなければならず、数字や記号を使用することはできません。使用できる文字は、英字、数字、記号は '_' (アンダーバー) で、それ以外の記号などは使用することが出来ません。'_' は2個以上続けて使用することはできません。また、予約語を信号名やピン名に使用することはできません。たとえば、ピンに "in" という名前を付けることはできません。



VHDL では、ファイル内の大文字・小文字を区別しません。たとえば信号名に “data_a” と定義した場合には、”data_a” と記述しても “DATA_A” と記述しても同じ信号と認識されます。ただし、例外として特定のデータタイプ（不定値 (X) や未定値 (U)、ハイインピーダンス (Z) など）を代入する際には大文字を使用します。参考として、Verilog-HDL では大文字・小文字の区別をしません。

最後に読みやすくするための工夫として、改行やインデント、コメントを使用することができます。

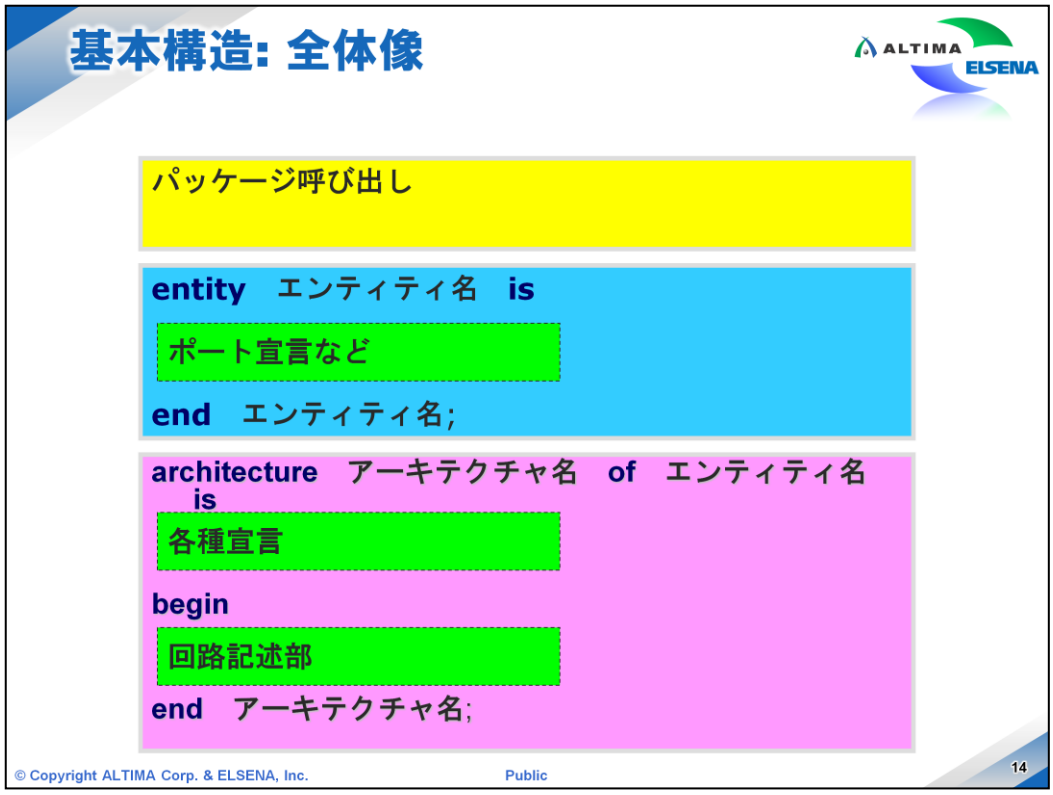
1つのステートメントが長くなる場合などには、複数行に分けて記述することができます。

コメントを入れる場合には、コメントの開始位置に ‘ー’ (ダブル・ハイフン)を付けます。ダブル・ハイフンを付けるとそれ以降、その行が改行されるまでがコメントと認識されます。

VHDL には複数行をまとめてコメントアウトする記述方法がありませんので、1行ずつダブル・ハイフンを付ける必要があります。

こちらが VHDL で使用される予約語です。よく使用されるものを記載しています。

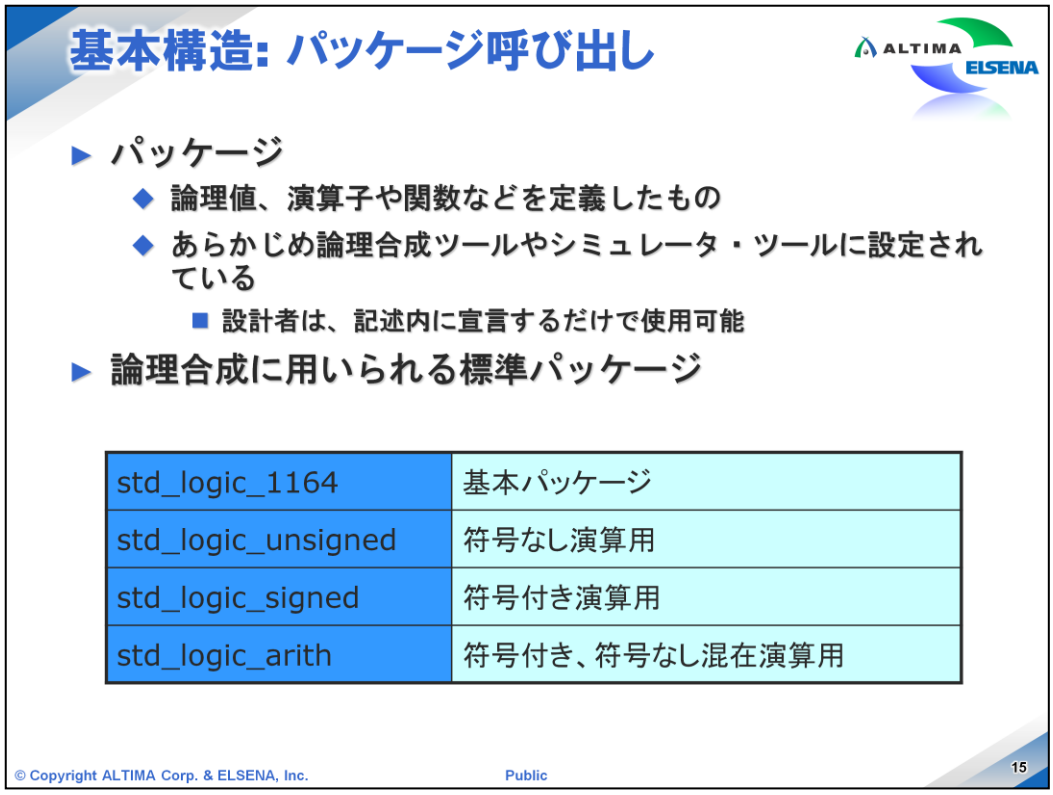
次に、VHDL の基本構造がどのようになっているのかを確認しましょう。



もっともシンプルな VHDL はエンティティ宣言部（水色）とそれに関連したアーキテクチャ本体部（ピンク）で構成されます。

エンティティ宣言は回路図のシンボルのようなもの、アーキテクチャ宣言はその中の回路、という風にイメージすると分かりやすいと思います。

また、パッケージの呼び出し（黄色）で演算子や標準的な関数等を使用できるようにします。



まずは、パッケージの呼び出しです。

パッケージとは理論値、演算子や標準関数などが定義されたものです。これらを呼び出すことによって、演算子や標準関数を使用できるようになります。

VHDL の各種パッケージは、論理合成ツールやシミュレータでは設定済み(コンパイル済み)の状態で組み込まれているため、ユーザはパッケージを宣言するだけで使用することができます。

回路記述では必ず最初にこのパッケージ呼び出しが必要です。一般的に用いられる標準パッケージを示します。

・std_logic_1164

基本パッケージです。std_logic データタイプや関数が定義されています。

・std_logic_unsigned

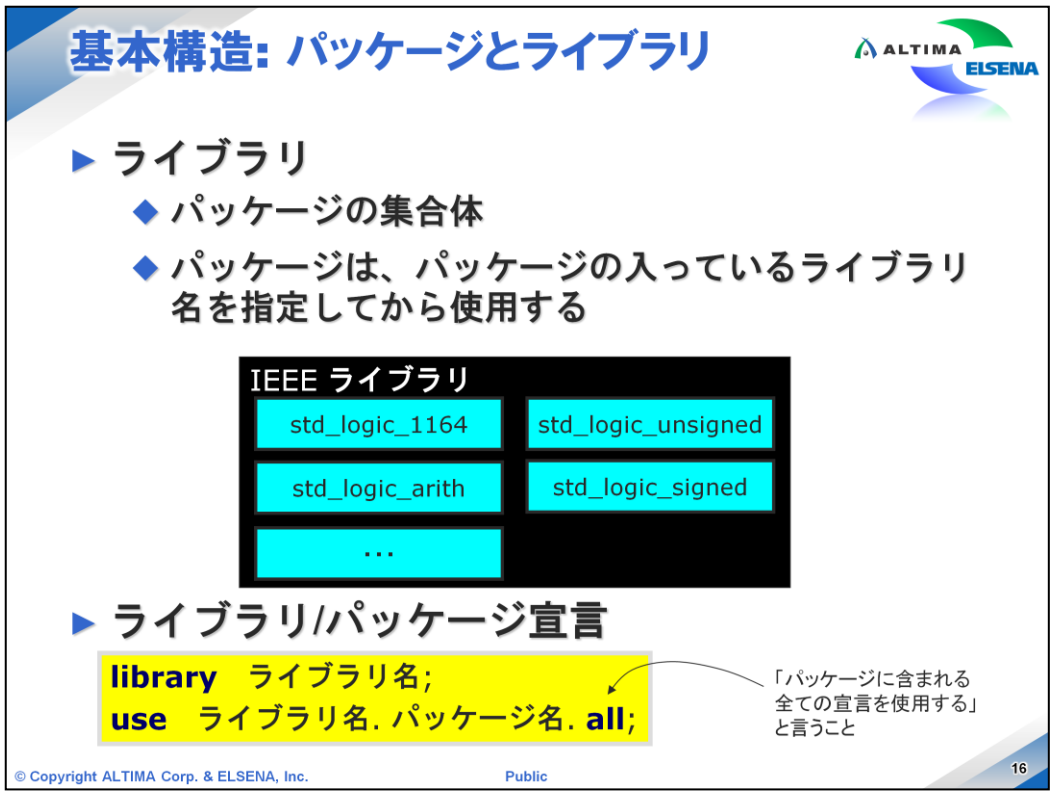
符号なし整数として std_logic_vector タイプを扱うために std_logic_arith を拡張したものです。算術演算子(+, −, *)、比較演算子 (<, <=, >, >=, =, /=)、シフト演算子 (>>, <<) などが含まれます。

・std_logic_signed

符号つき整数として std_logic_vector タイプを扱うために std_logic_arith を拡張したものです。算術演算子、比較演算子、シフト演算子などが含まれます。
std_logic_unsigned と std_logic_signed を同時に使用することはできません。

・std_logic_atith

基本的な算術演算子が定義されています。



このようにいくつかの標準パッケージがありますが、呼び出す際にはライブラリから指定する必要があります。

ライブラリとは、複数のパッケージをまとめて1つのディレクトリに格納したものです。IEEE ライブラリは、IEEE によって承認されたライブラリで、前頁で紹介した、std_logic_1164 や std_logic_arith 等が含まれています。

このライブラリやその中のパッケージを使用するために、パッケージの呼び出しが必要なのです。ライブラリ、パッケージを呼び出す前に、まず library ライブラリ名; とライブラリの宣言をします。

そのあとに、use ライブラリ名.パッケージ名.all; と記述し、パッケージを宣言します。all とすることによって、パッケージ内のすべての宣言を含むことができます。

std_logic_1164 を定義するには、

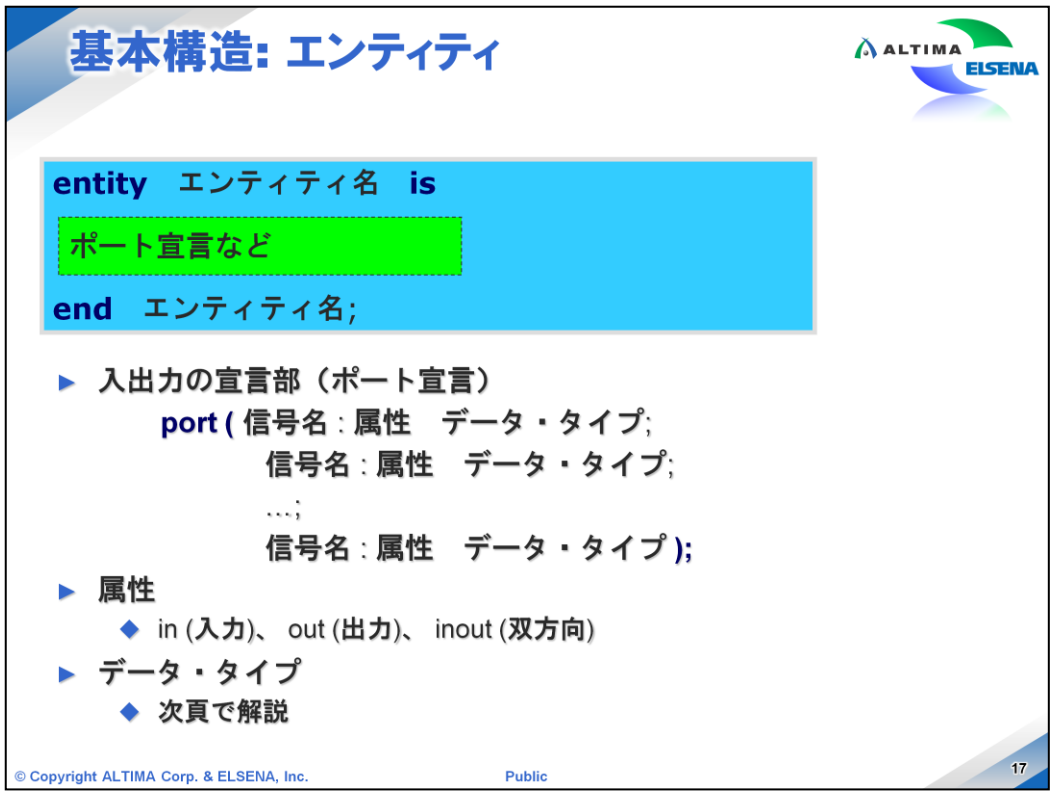
library ieee;

use ieee.std_logic_1164.all;

という風に記述します。

また、呼び出したパッケージの有効期間は、エンティティの終了までです。

1つのファイル内で複数のエンティティを記述している場合には、ライブラリの宣言とパッケージの宣言をやり直す必要があります。



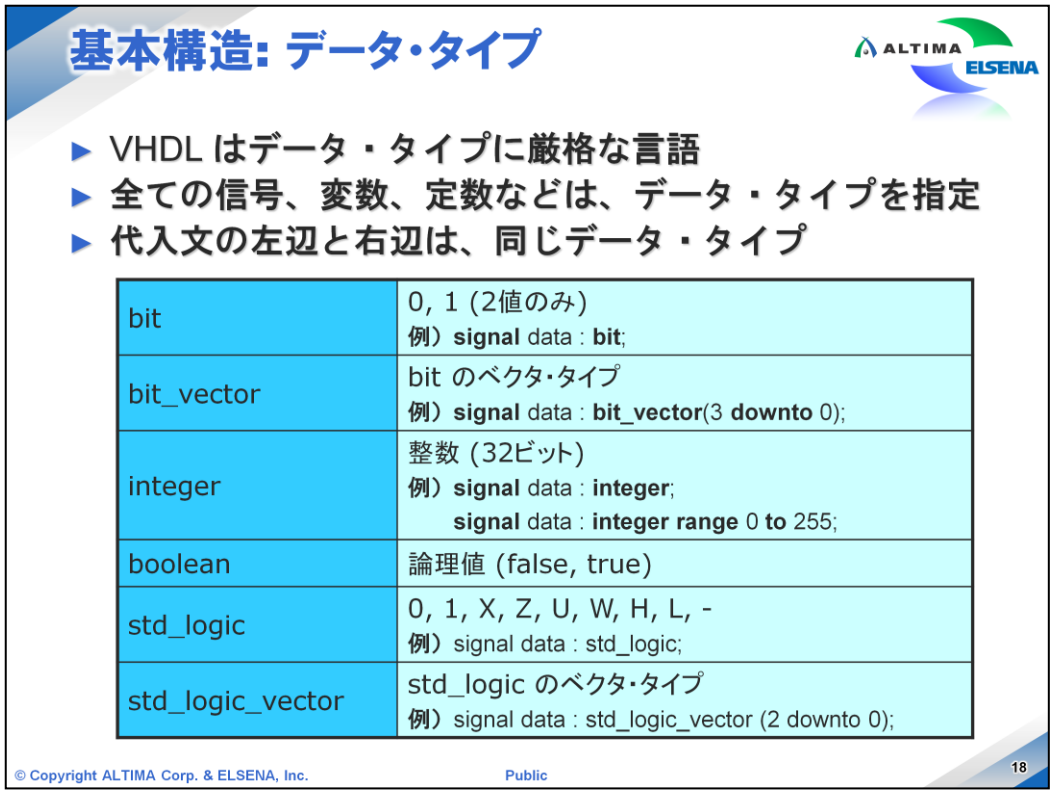
ライブラリやパッケージの呼び出しができたら、いよいよ回路に関する記述をしていきたいところですが、

回路記述をする前に、エンティティ宣言が必要です。エンティティ宣言では、作成するエンティティの入出力ポートの一覧、および、それらの属性やデータ・タイプを記述します。

エンティティ宣言の記述方法としては、entity から始まり、任意のエンティティ名が続き、is と続きます。この後に、エンティティのポート一覧を記述します。最後に end エンティティ名; で終了です。

入出力ポートの宣言は、port (); の括弧の中に、信号名 : 属性 データ・タイプ; という順番で記述し、複数ポートがある場合には、セミコロン (;) で区切り追加していきます。一覧の最後のポート宣言にはセミコロンは付けず、括弧で閉じてセミコロンを付けます。

属性は in (入力)、out (出力)、inout (双方向) を指定します。入力属性で宣言されたポートにエンティティ内部から値を代入することはできません。出力属性の信号を信号代入の右辺で使用することもできません。



ポートや信号を宣言するにはデータ・タイプの指定が必要です。

デザイン内で使用しているすべての信号(signal)、変数(variable)、定数(constant)にデータ・タイプを指定します。VHDL はデータ・タイプに厳格な言語であるため、異なるデータ・タイプ同士の代入はできません。また、同じデータ・タイプでもビット幅の異なるものは代入できません。

このリストは VHDL の標準データ・タイプです。

bit タイプは、'0' '1' の2値のみで論理値を表します。bit_vector タイプは、bit タイプのベクタ配列を表し、2bit 以上の信号となります。使用する際にはビット幅を指定します。

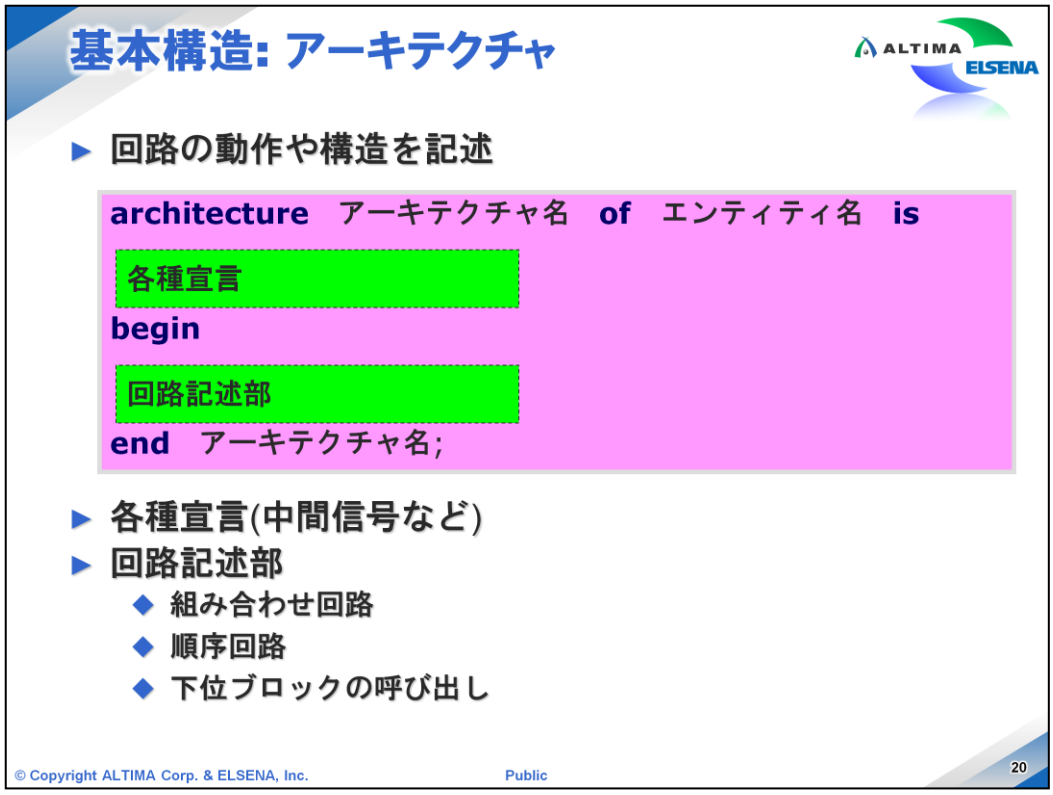
integer タイプは、32bit の整数値を表し、整数範囲を range で指定します。指定しない場合は、32bit として扱われます。(コンパイラの条件による)range 0 to 255 と指定すると、8bit の信号となります。

boolean タイプは、真(true)と偽(false)を表します。比較などの関係演算子で、その演算結果として真であれば true、偽であればfalse を返します。

std_logic タイプは、9値のロジック信号 ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-')を表すことができるデータ・タイプです。std_logic_vector タイプは、std_logic の配列ベクタとなります。std_logic はもっとも一般的に使われるタイプです。

bit, boolean, integer は standard パッケージで定義され、std_logic、std_logic_vector は std_logic_1164 パッケージで定義されます。

それでは、演習1を実施してください。
詳細は、演習マニュアルをご覧ください。



エンティティ宣言ができましたら、回路の動作や構造(アーキテクチャ)を記述していきます。

回路記述部は、組み合わせ回路や順序回路、下位ブロックを呼び出したりして、回路の動作や構造を記述します。

アーキテクチャはエンティティ宣言で指定したシンボル内の回路を表現していますので、アーキテクチャ名 of エンティティ名 is という風にどのエンティティに含まれる、という記述で始まり、回路記述部分を begin と end で囲った中に記述します。end のあとには再度アーキテクチャ名を記述します。

アーキテクチャ内で使用する中間信号や変数などがある場合には、is と begin の間(各種宣言部分)に記述します。

中間信号などの宣言にも、先ほどのデータ・タイプの指定が必要です。

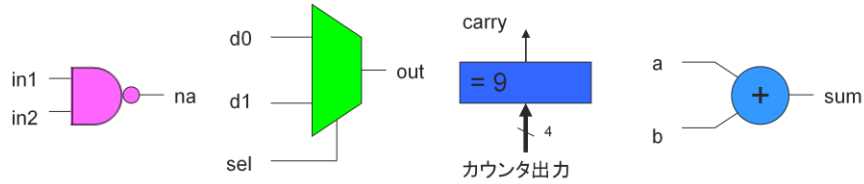
回路記述に使用されるさまざまな回路表現方法を見ていきましょう。

簡単な組み合わせ回路

ALTIMA ELSENA

▶ 論理式 1行 で記述できる組み合わせ回路

```
na  <= not (in1 and in2);                          --2入力 NAND
with sel select out <= d0 when '0', d1 when others;  --セレクタ
carry <= '1' when cnt10 = "1001" else '0';          --ケタ上がり信号
sum <= a + b;                                       --加算回路
```



2入力 NAND **セレクタ** **ケタ上がり信号** **加算回路**

まずは簡単な組み合わせ回路の表現方法についてです。

いくつか例を挙げていますが、このように論理式1行の記述で組み合わせ回路を表現することができます。

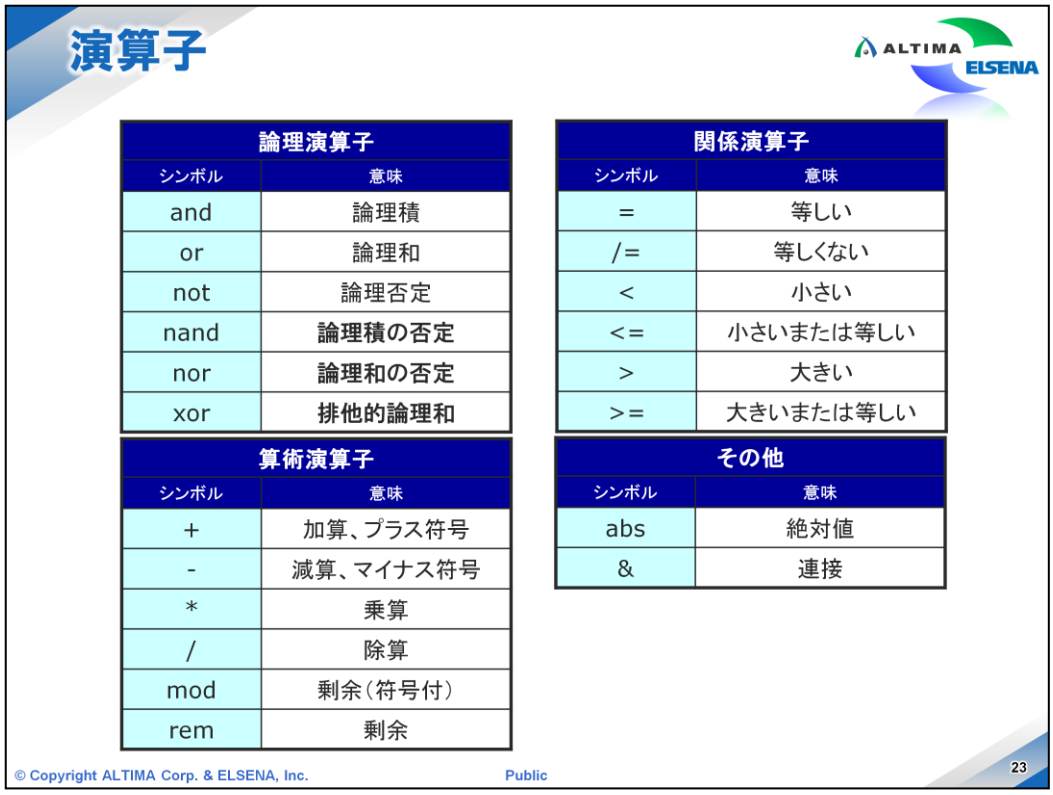
まずは、2入力 NAND の記述例で、論理演算子(and や not)を使用しています。優先順位を付ける場合には、このように括弧を付けて指定します。in1 と in2 の and をとり、not を付けて na に代入していますので、出力側に not が付きます。

次にセレクタの記述例です。セレクタ回路は回路設計でよく使用される回路で、セレクタ信号により複数の入力信号を切り替えて出力します。VHDL ではこのようにセレクタ回路を記述することができます。sel が‘0’の時に d0 を出力し、sel がその他値(‘1’)の時に d1 を出力します。

3つ目は桁上がり信号を作成する回路例です。セレクタ回路と似た記述をします。carry 信号に、cnt が“1001”(10進の‘9’)の時に‘1’が出力されるようになります。

このように構文を使用することにより、簡単に回路を記述することができます。

最後に、加算回路の記述例です。加算回路は、and や or、not を組み合わせて作成することができますが、VHDL 記述では算術演算子を使用することができるので、このように1行で記述することも可能です。



ここで回路記述で使用する演算子について確認しておきましょう。前頁の例で、論理演算子と算術演算子を使用していました。

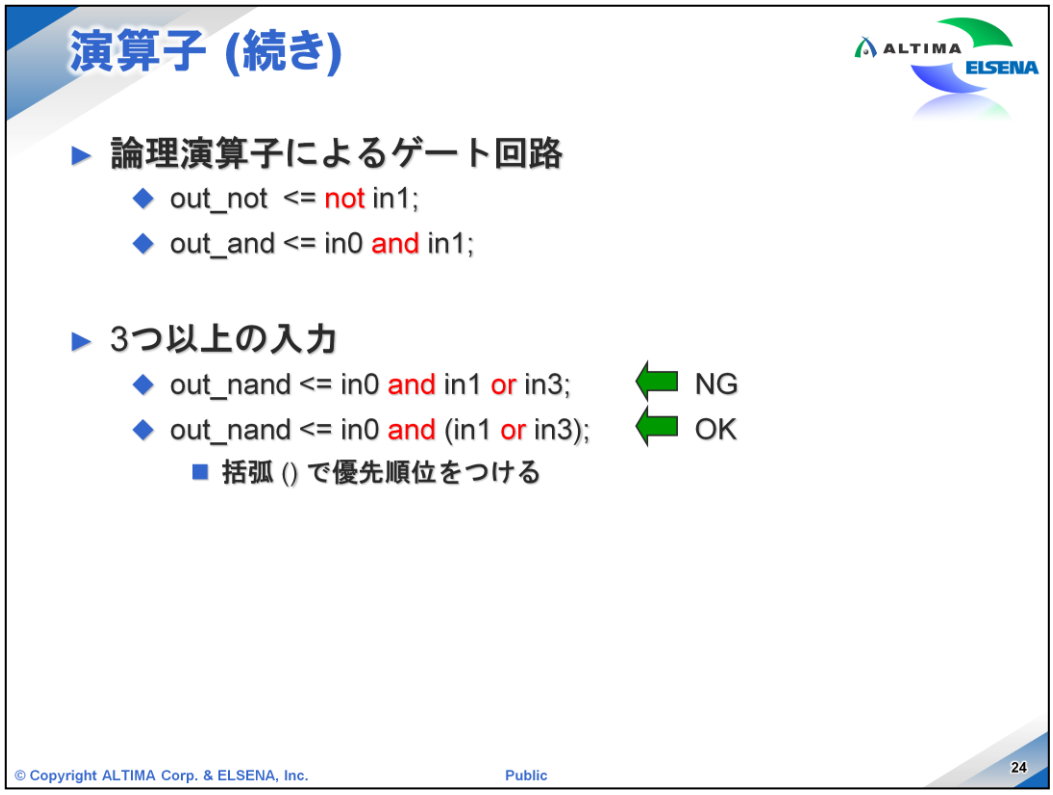
論理演算子は、and や or などの論理をそのまま記述することができます。

算術演算子は、加算や減算、乗算を記述することができます。

VHDL 標準パッケージでは、論理演算子は bit タイプでのみ、算術演算子は integer タイプでのみ使用可能で、std_logic_vector で定義した信号に対して算術演算子を使用するには、std_logic_unsigned（符号なし）もしくは std_logic_singed（符号付き）というパッケージを呼び出す必要があります。

関係演算子は、右辺と左辺の比較を行い、boolean タイプの”真（true）”、”偽（false）” を返し、if 文の条件式で使用されることが多い演算子です。

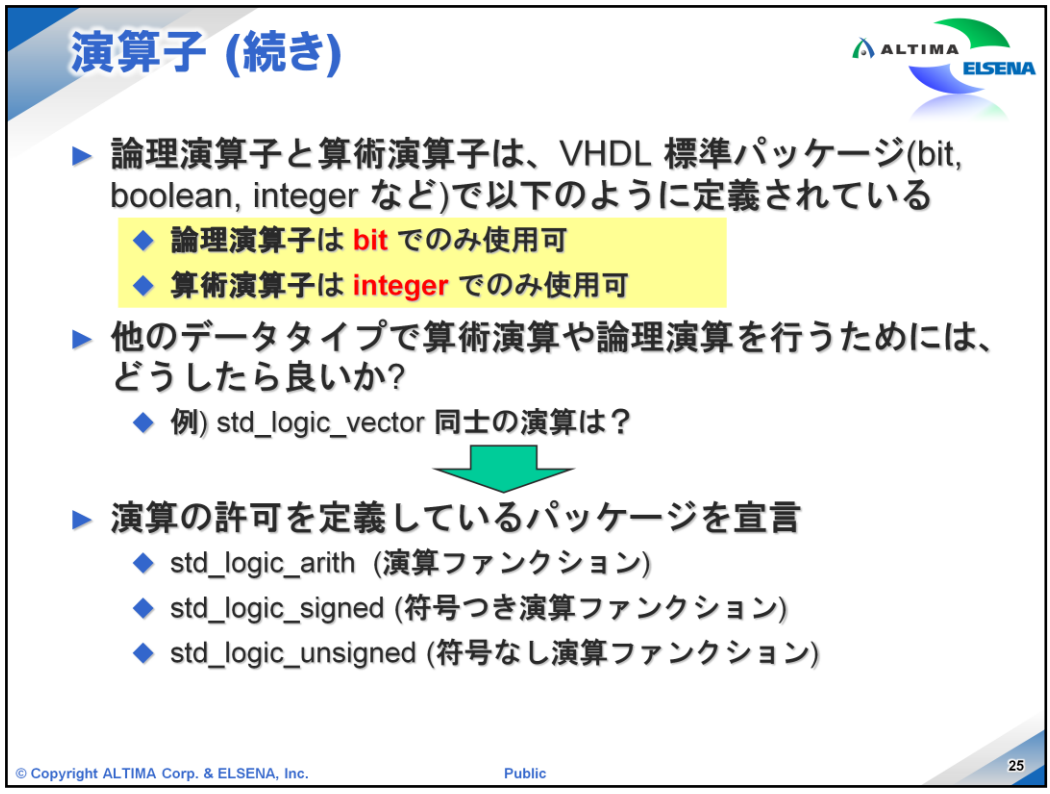
その他、abs は絶対値を返し、& を使用してビットやベクタを連結することができます。



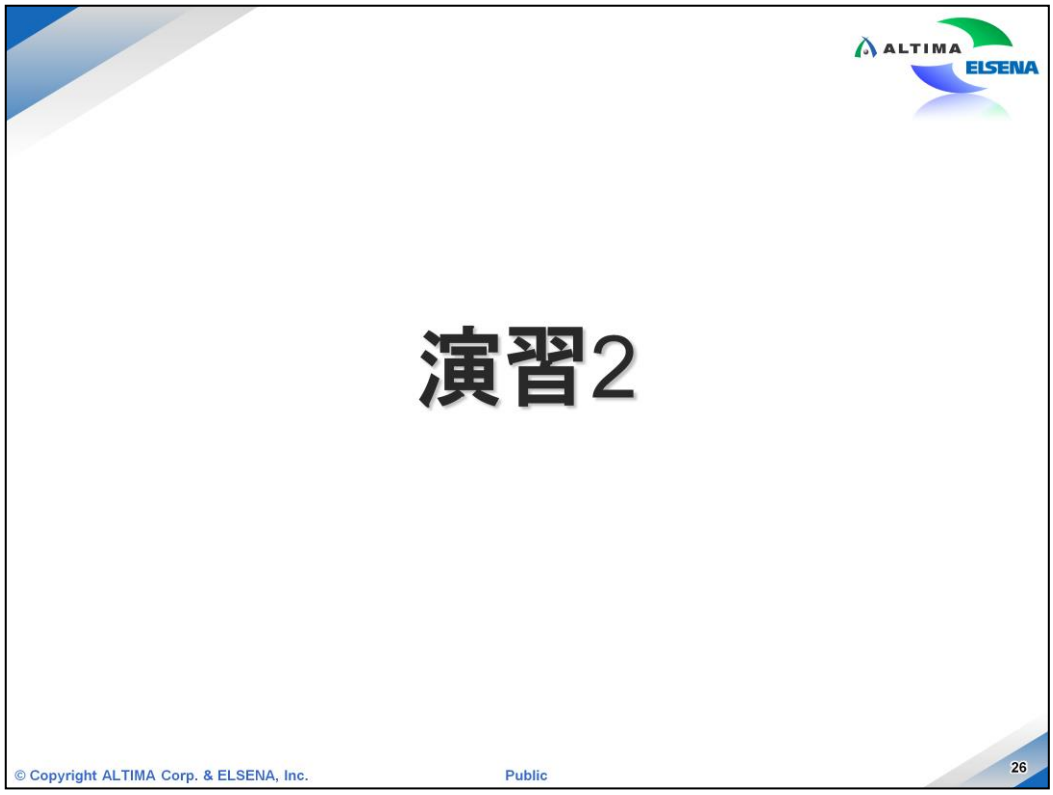
これらの演算子を使用してゲート回路を記述することができます。

オペランド1つに対し使用して、not を使用して反転、and を使用して各オペランドの and を左辺に代入します。

3つ以上のオペランドがある場合には、VHDL では左の論理式のほうが優先順位が高いなどがないため、括弧で優先順位を付けることが必要です。ただし、”not” だけは他の演算子よりも優先順位が高くなります。



VHDL 標準パッケージ(bit、boolean、integer など)では、論理演算子と算術演算子で使用できるデータ・タイプが一部のみのため、別のデータ・タイプで使用する場合には、演算子の使用を定義しているパッケージを呼び出す必要があります。

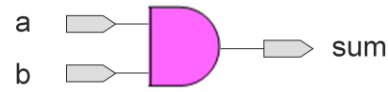


それでは、演習2を実施してください。
詳細は、演習マニュアルをご覧ください。

複雑な組み合わせ回路

- 複雑な組み合わせ回路は process 文で記述
  - process 文の後ろ、括弧内に全ての入力を記述
    - センシティビティ・リスト
  - process 文の中に動作を記述

```
process (a, b)
begin
        sum <= a and b;
end process;
```



  - if 文、case 文を用いて、より複雑に記述

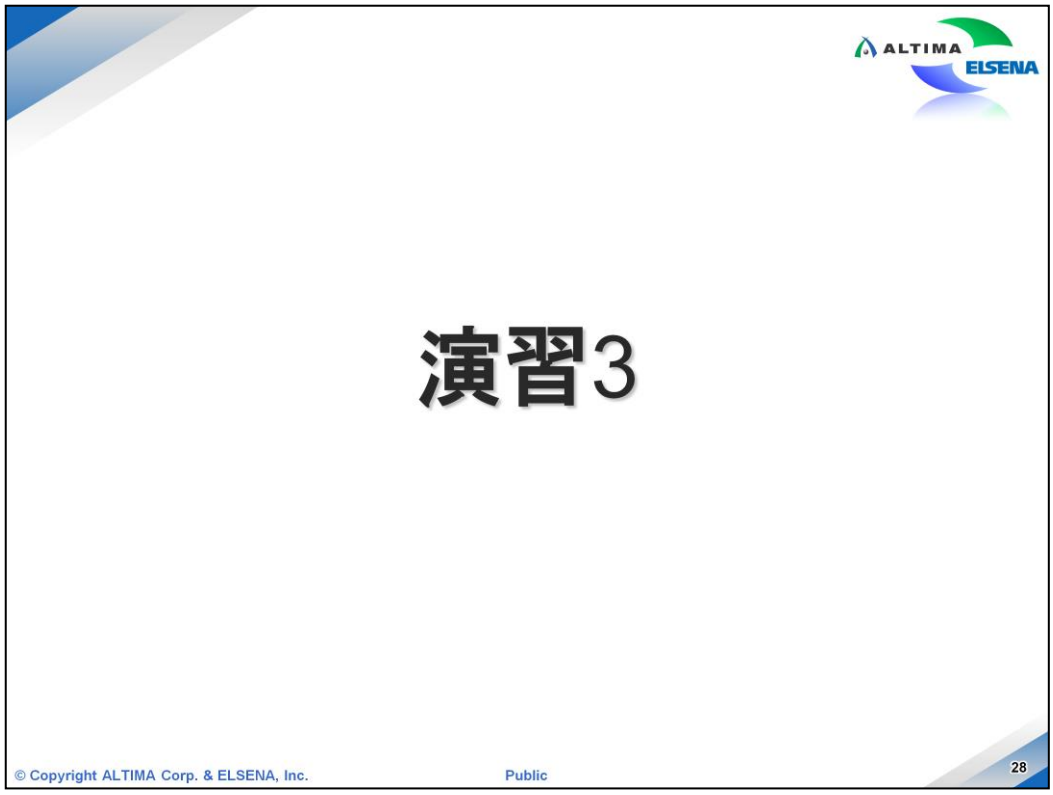
1行では記述することができない複雑な組み合わせ回路は、process 文を使用して記述します。

process 文の記述は、まず process (a, b) のようにまず記述し、括弧の中の信号いずれかの値が変化した時に、process 文の中の記述が実行されます。この括弧の中をセンシティビティ・リストと呼びます。

最後は end process; で1つの process 文が終了します。end まで実行されると、またセンシティビティ・リストのいずれかの信号が変化するまで動作しません。

この例のように、and 回路を process 文を使用して組み合わせ回路を記述する場合には、センシティビティ・リストにすべての入力信号を記述します。

if 文や case 文を使用して、条件などを付けることによってより複雑な回路を記述することができますが、if 文や case 文は必ずこの process 文内で使用することになります。

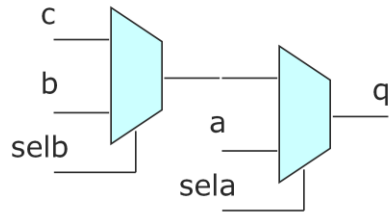


それでは、演習3を実施してください。
詳細は、演習マニュアルをご覧ください。

if 文

```
architecture logic of if_sel is
begin
    process (a, b, c, sela, selb)
    begin
        if (sela = '1') then
            q <= a;
        elsif (selb = '1') then
            q <= b;
        else
            q <= c;
        end if;
    end process;
end logic;
```

- 条件は、記述した順に評価される
  - 優先順位
- 条件が“真”である場合、該当のステートメントが実行される
- 全ての条件が“偽”であれば else のステートメントが実行される
- process ブロック中で使用可能



次に if 文を使用した組み合わせ回路の記述方法です。if 文は必ず process 文内で使用します。

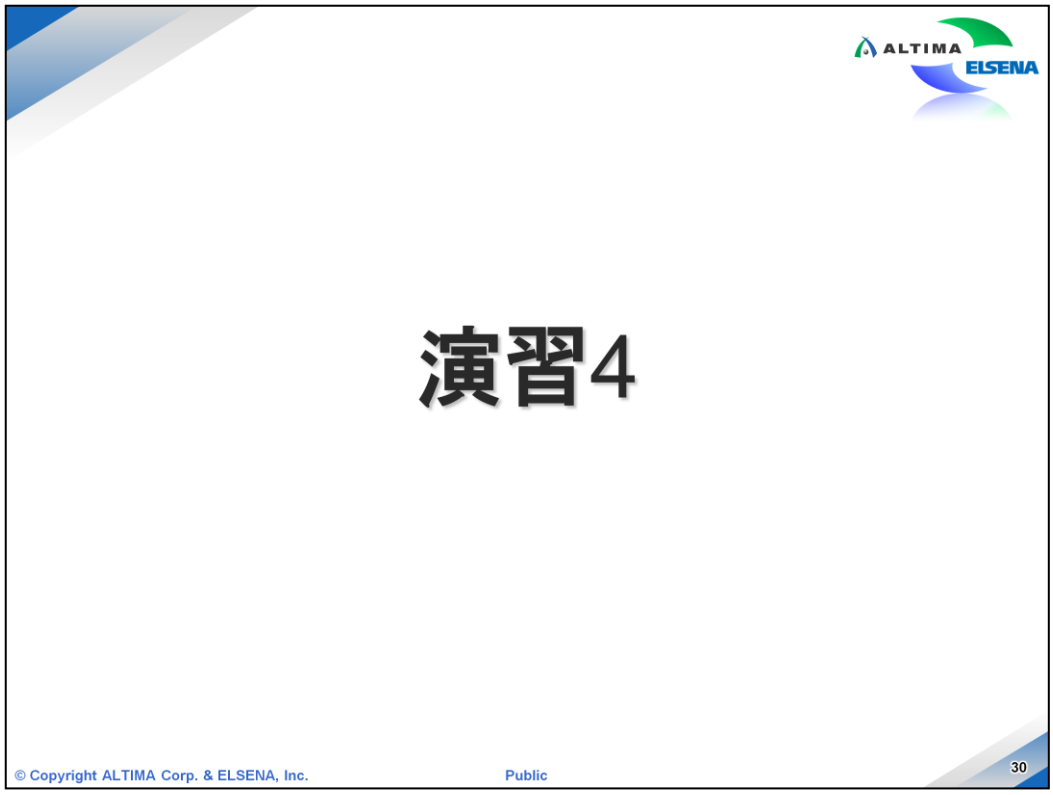
まずは、センシティビティ・リストにすべての入力信号を記述します。

if () then の括弧に条件を記述します。2つ目以降の条件は else () then で記述します。条件は上の記述から順番に評価され、条件が “真(true)”となった場合にその記述内が実行されます。

すべての if、ifelse 条件が“偽(false)”となった場合には最後に記述した else が実行されます。

この例文のように、if 文を使用してセレクタ回路を記述することができます。

この場合は、if、ifelse で2つのセレクタが記述されており、1つ目の if 条件が最初に評価されるため優先順位が高くなるので、構成される回路は左下の図のようになります。

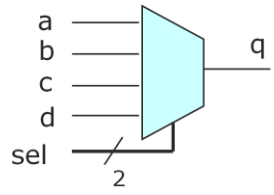


それでは、演習4を実施してください。
詳細は、演習マニュアルをご覧ください。

case 文

```
architecture logic of case_sel is
begin
    process (a, b, c, d, sel)
    begin
      case (sel) is
       when “00” =>
                    q <= a;
       when “01” =>
                    q <= b;
       when “10” =>
                    q <= c;
       when others =>
                    q <= d;
      end case;
    end process;
end logic;
```

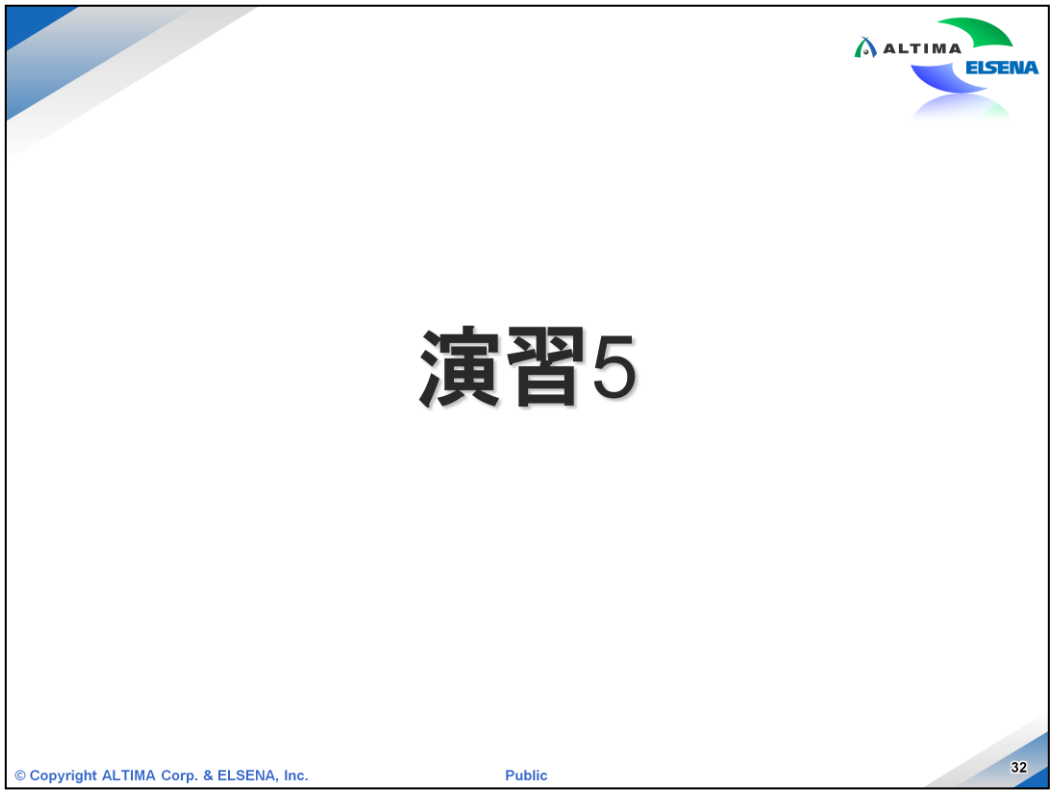
- 条件は、同時に評価される
  - 優先順位はない
- 全ての条件を考慮しなければならない
  - 重なる条件が無いようにする必要がある
- when others は、条件に無いその他の場合に実行される
- process ブロック中で使用可能



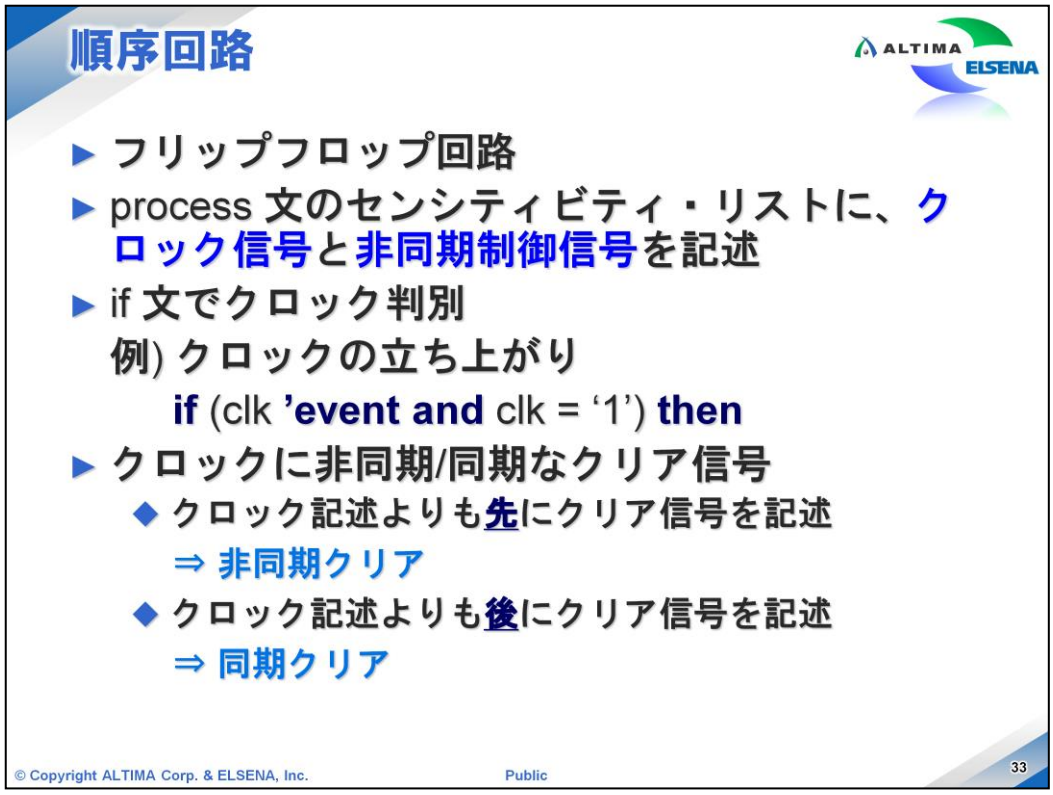
if 文と同じくよく使用されるのが case 文です。case 文も process 文内で使用します。

case () is で括弧内に、条件となる信号を記述し、when 条件となる信号の値 => 実行されるステートメント、という風にすべての取り得る条件の場合を考慮する必要があり、条件は重なりが無いように記述する必要があります。when others を使用して条件にない場合を記述します。最後に end case; で終了します。

if 文は上から順番に評価されますが、case 文はすべての条件が同時に評価されます。前頁の if 文の条件式では優先順位付きのセレクタ回路が構成されましたが、case 文を使用するとこの例のように優先順位のないセレクタ回路が構成されます。



それでは、演習5を実施してください。
詳細は、演習マニュアルをご覧ください。



ここまでは組み合わせ回路の記述方法を見てきました。回路設計をするにはもう一つフリップフロップという順序回路を使用する必要があります。

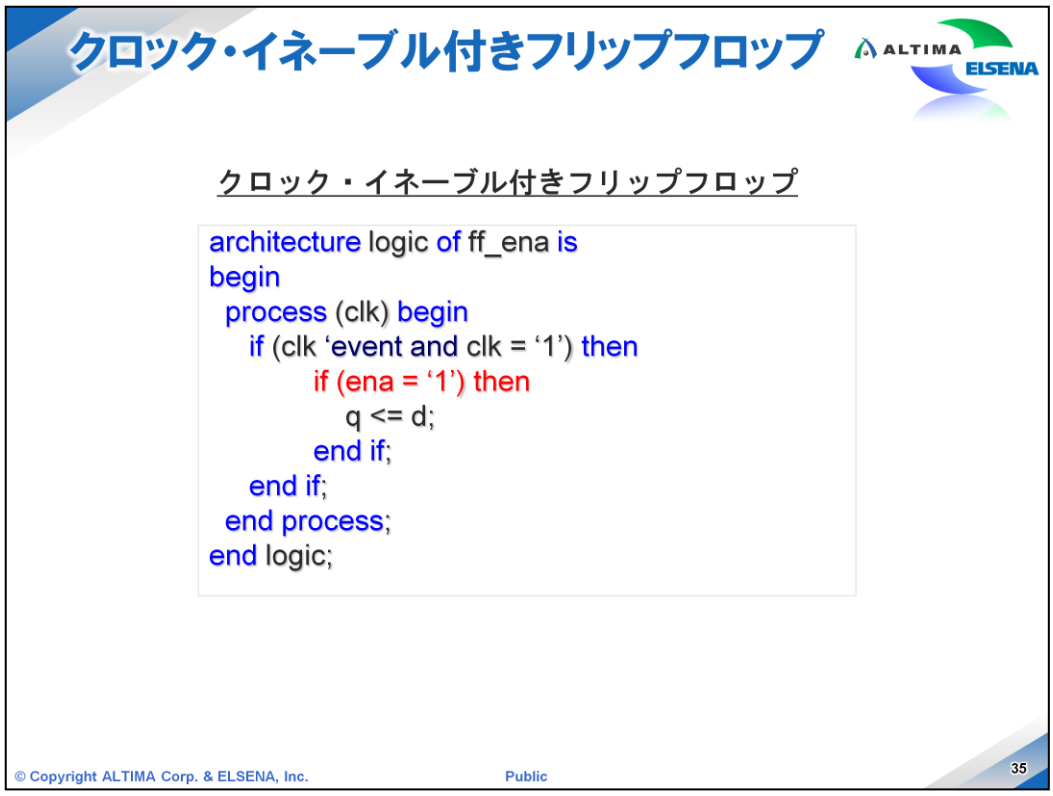
フリップフロップを記述するには、process 文で if 文を使用します。センシティビティ・リストにクロック信号と、非同期制御信号のみを記述します。if 文の条件で if (clk 'event and clk =1) then という風に、clk 信号が立ち上った時という条件を記述し、そのあとに出力を記述していきます。

フリップフロップにはクリア信号を使用しますが、クロック記述(if (clk 'event and clk =1))よりも先に if (reset =1) などリセットを記述すると非同期クリアとなり、クロックよりも後に記述すると同期クリアとなります。

同期クリアと非同期クリアを使用したフリップフロップ記述例です。

同期クリア付きフリップフロップでは、センシティビティリストに“clk”のみが記載され、クロックイベントの記述のあとにクリアの記述が書かれています。

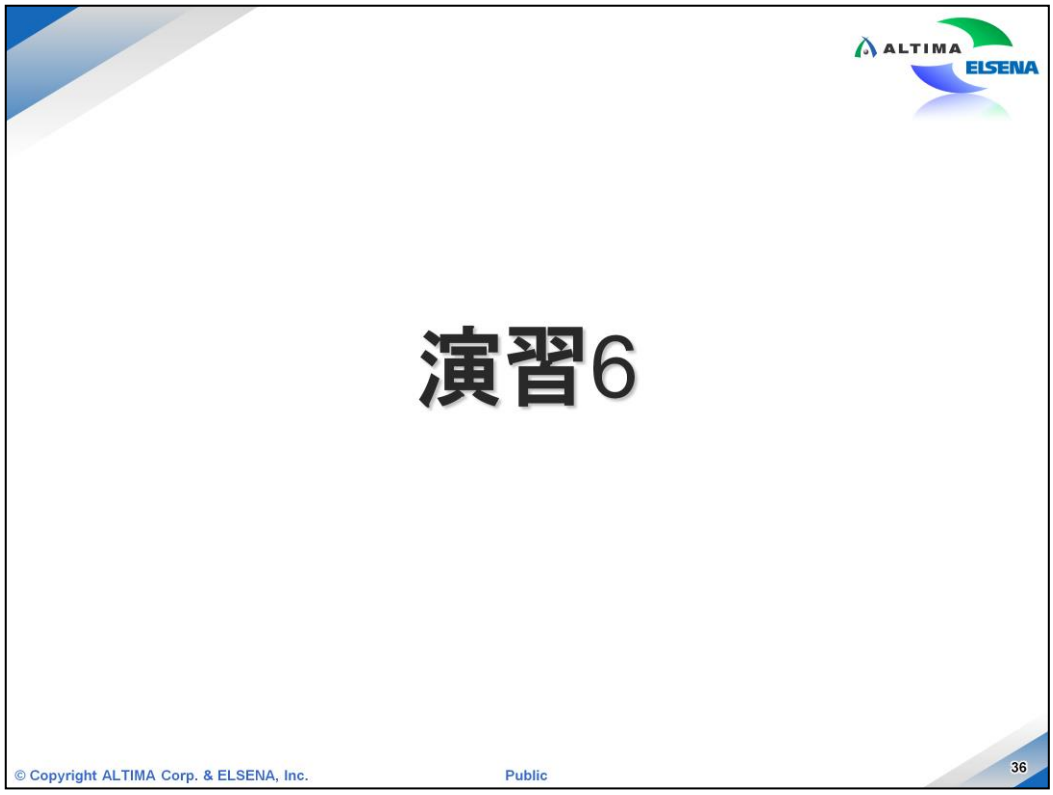
右側の非同期クロック付きフリップフロップでは、センシティビティリストに、クリア信号も記述されています。また、クリア記述がクロックよりも先に記述され、クリア信号が有効になった際には、直ちにフリップフロップの値がクリアされるという動作をします。



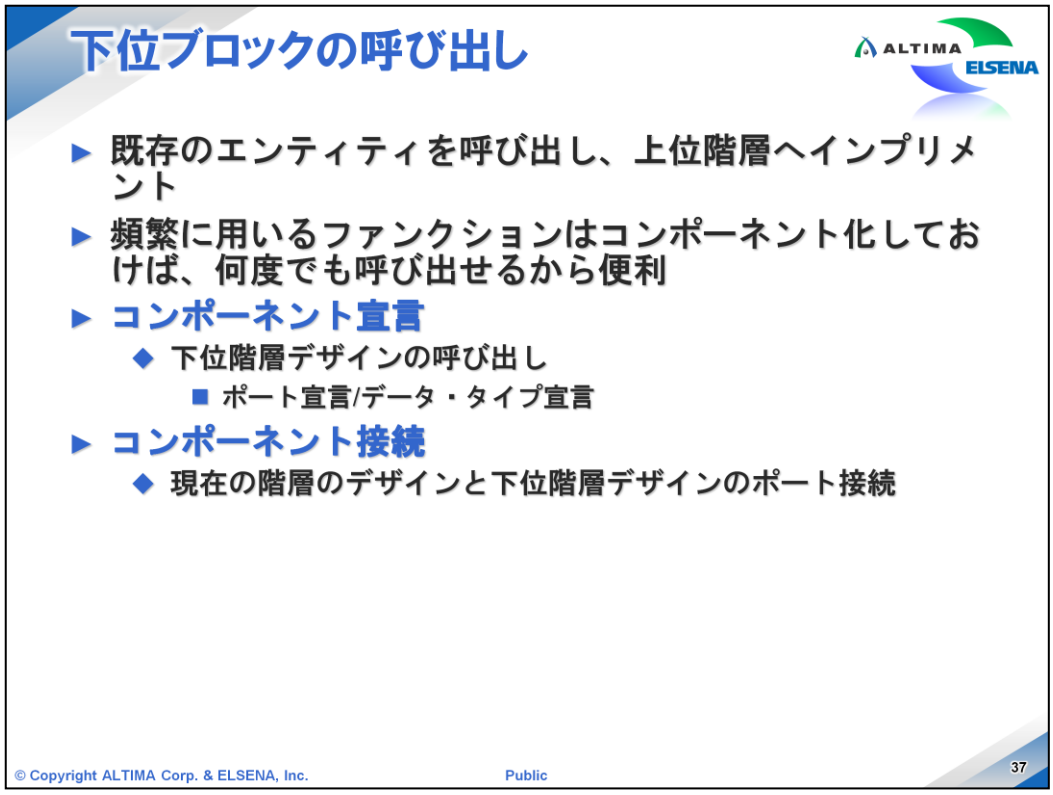
クロック・イネーブル付きのフリップフロップも同じように記述することができます。

if (ena = '1') then というようにイネーブル信号の条件式を追加することによって、クロックが入力され、かつイネーブルが有効になった時のみデータをアップデートできるフリップフロップの動作となります。

この例では、簡単にするためにクリア信号を省いています。



それでは、演習6を実施してください。
詳細は、演習マニュアルをご覧ください。

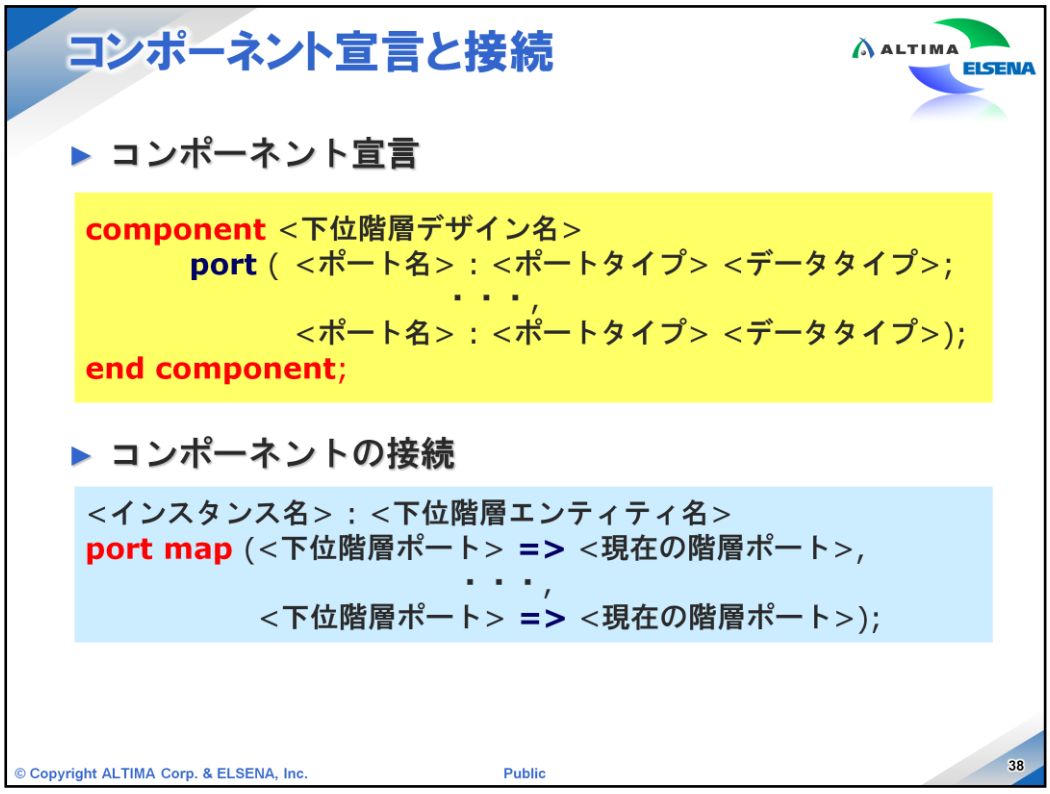


最後に VHDL を使用した階層の記述方法について説明いたします。

実際に回路設計を進めていく際には、既存のエンティティをコンポーネントし、上位階層から呼び出して使用することが多いです。このようにすると1ファイル内の記述が無駄に長くなったりすることを防ぐことができますし、よく使用する機能などをコンポーネント化し汎用的に使用することができます。

VHDL で下位階層を呼び出して使用するには、コンポーネント宣言と、コンポーネント接続記述、のセットが必ず必要です。

コンポーネント宣言は、下位階層のポートの宣言とデータ・タイプ宣言を記述します。コンポーネント接続では、現在の階層内の信号と呼び出した下位階層のポートの接続関係を記述します。

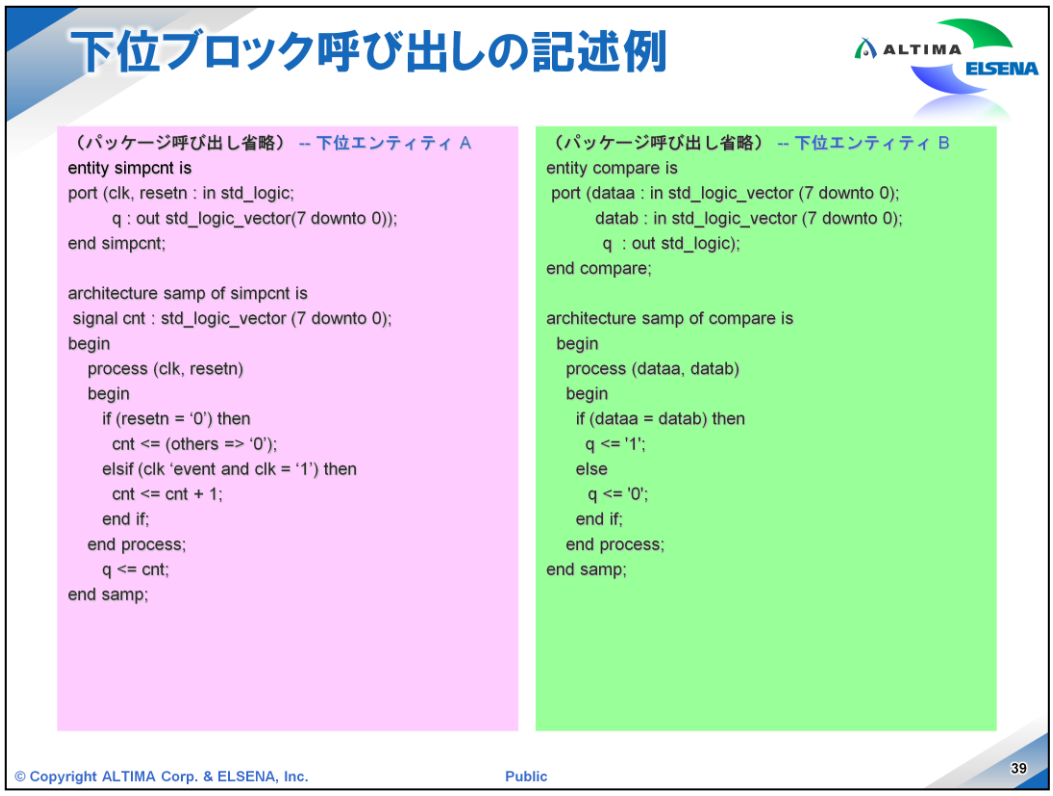


コンポーネント宣言とコンポーネント接続の記述方法です。

コンポーネント宣言（黄色）は下位階層のエンティティ宣言に似ています。呼び出す下位階層のエンティティ宣言をコピーし entity の部分を component に変更し is を消します。最後も entity を component に変更し、end component; とすると簡単に記述できます。

接続記述（水色）には、まずインスタンス名を記述します。インスタンス名とはこのコンポーネントにつけられる名前のようなもので、現在の階層上での識別名となりますので、同じコンポーネントを複数呼び出す場合にはそれぞれ固有のインスタンス名を付けます。

次に下位階層のエンティティ名を記述します。port map に続けて 下位階層のポートと現在の階層の信号を接続していきます。 => を使用し、複数ポートがある場合はカンマ（,）で区切って、次の接続を記述し、最後は); で閉じます。



下位ブロック呼び出しの記述例です。

このように simpcnt（エンティティ A ）と compare（エンティティ B） という2つのエンティティがあるとします。

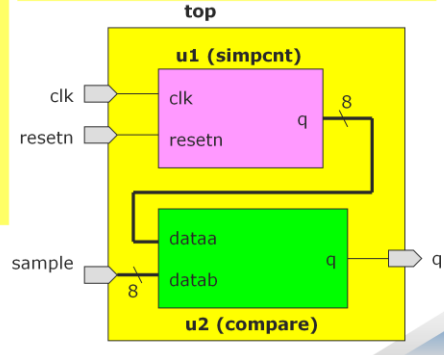
エンティティ A はフリップフロップを使用したカウンタとなっています。エンティティ B は比較器です。

これら2つのエンティティを使用して回路を作ってみましょう。（次頁）

下位ブロック呼び出しの記述例(続き)

```
（パッケージ呼び出し省略） -- 上位エンティティ
entity top is
port (clk, resetn : in std_logic;
      sample : in std_logic_vector (7 downto 0);
      q : out std_logic);
end top;

architecture top of top is
 signal count : std_logic_vector(7 downto 0);

 component simpcnt
 port (clk, resetn : in std_logic;
       q : out std_logic_vector(7 downto 0));
 end component;

 component compare
 port (dataa : in std_logic_vector (7 downto 0);
       datab : in std_logic_vector (7 downto 0);
       q  : out std_logic);
 end component;
```

```
begin
 U1 : simpcnt port map (
                        clk => clk,
                        resetn => resetn,
                        q => count
                        );

 U2 : compare port map (
                        dataa => count,
                        datab => sample,
                        q => q
                        );
end top;
```

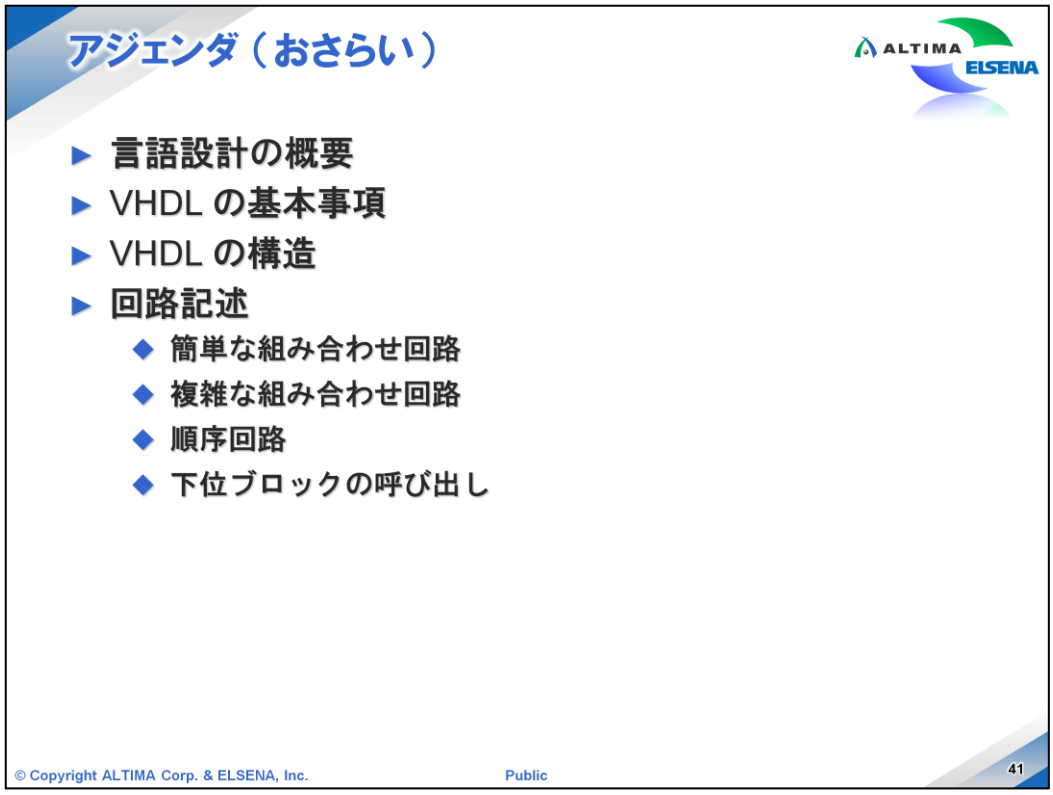


それでは上位階層から前頁の2つのエンティティを呼び出してみましょう。

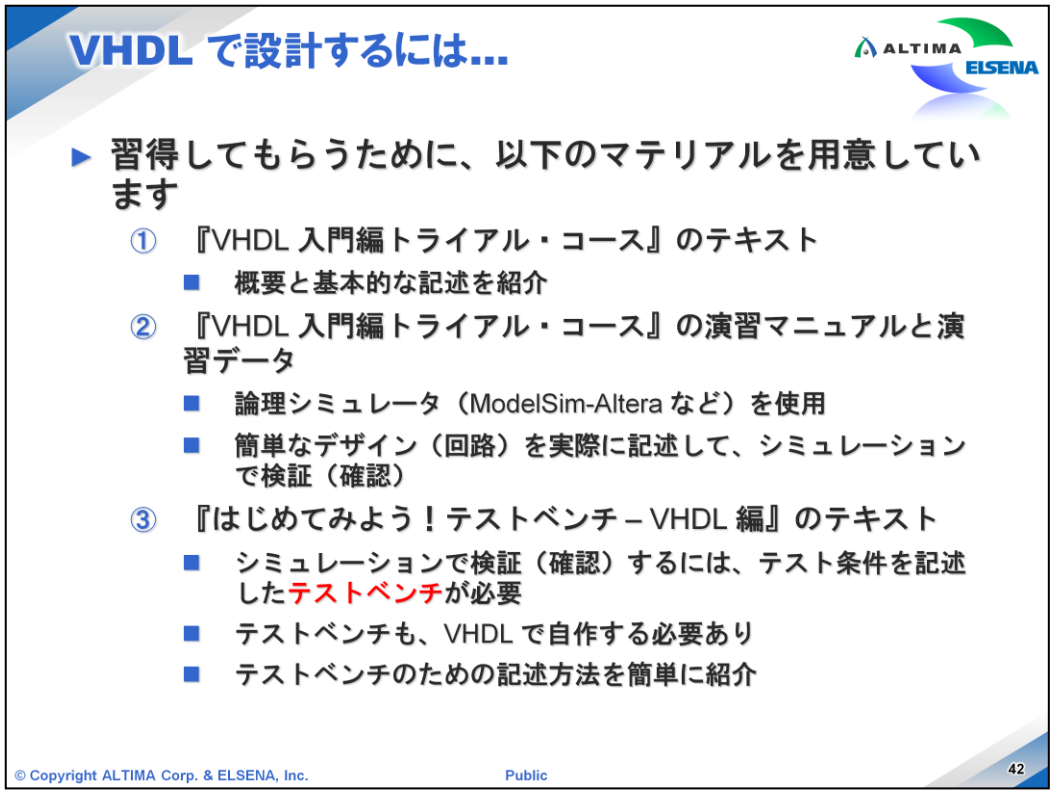
まずはそれぞれ simpcnt と compare のコンポーネント宣言を architecture 内で記述します。

次にポートの接続を記述していきますが、右下のように、simpcnt の q ポートと compare の dataa ポートを接続しますので、接続するための count という名前の signal を宣言しておきます。

最後にポートマップにそれぞれのポートの接続を記述して階層構造の完成です。



ここまで、このアジェンダに沿って、説明してきました。



冒頭でも触れましたが、VHDL で設計できるようになるために、以下のマテリアルを用意しています。

「①『VHDL 入門編トライアル・コース』のテキスト」では、概要と基本的な記述を紹介します。

そして、演習を通して理解度を上げてもらうため、「②『VHDL 入門編トライアル・コース』の演習マニュアルと演習データ」を用意しています。

演習を実施するには、ModelSim-Altera などの論理シミュレータが必要です。

アルテラの Web サイトから、無償版の ModelSim-Altera Starter Edition を入手することができます。

VHDL で論理回路を記述したら、実機確認する前にシミュレーションで検証をしますが、シミュレーションをするには入力ピンのテスト条件を記述したテストベンチが必要です。

このテストベンチは、基本的に VHDL で自作する必要があります。

そのため、「③『はじめてみよう！テストベンチ – VHDL 編』のテキスト」でテストベンチのための記述方法を紹介します。

いかがでしたか？

これで、『VHDL 入門編トライアル・コース』のワークショップは終了です。

受講して頂き、誠にありがとうございました。