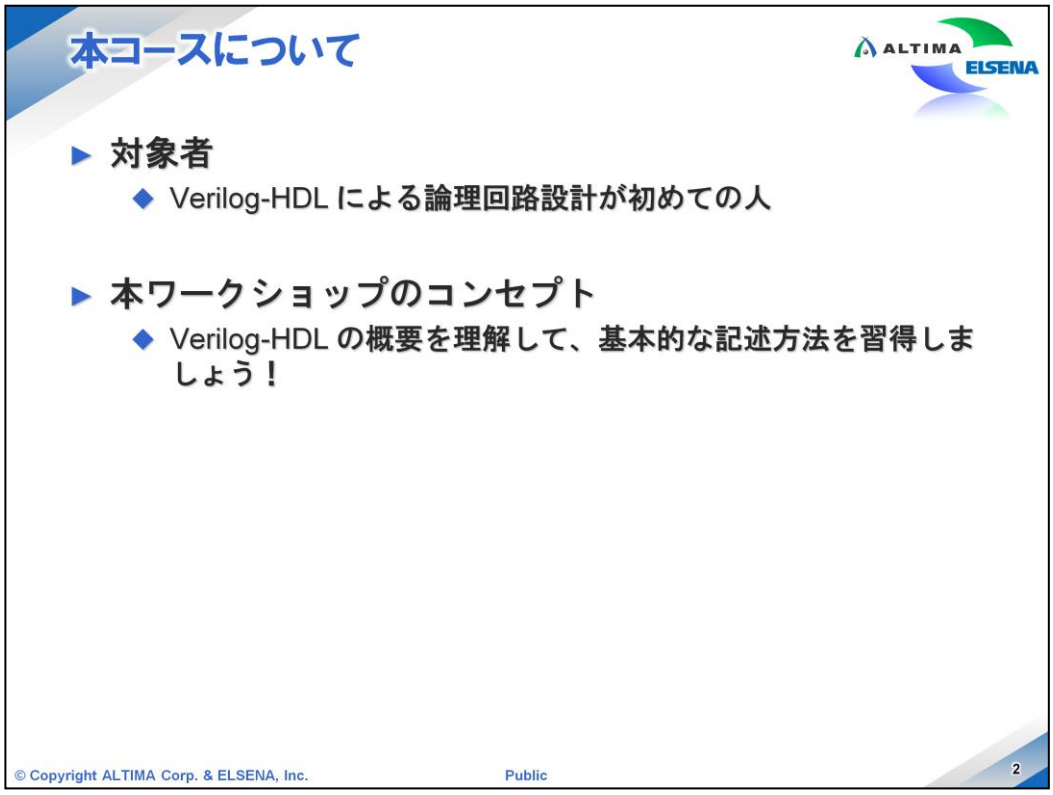
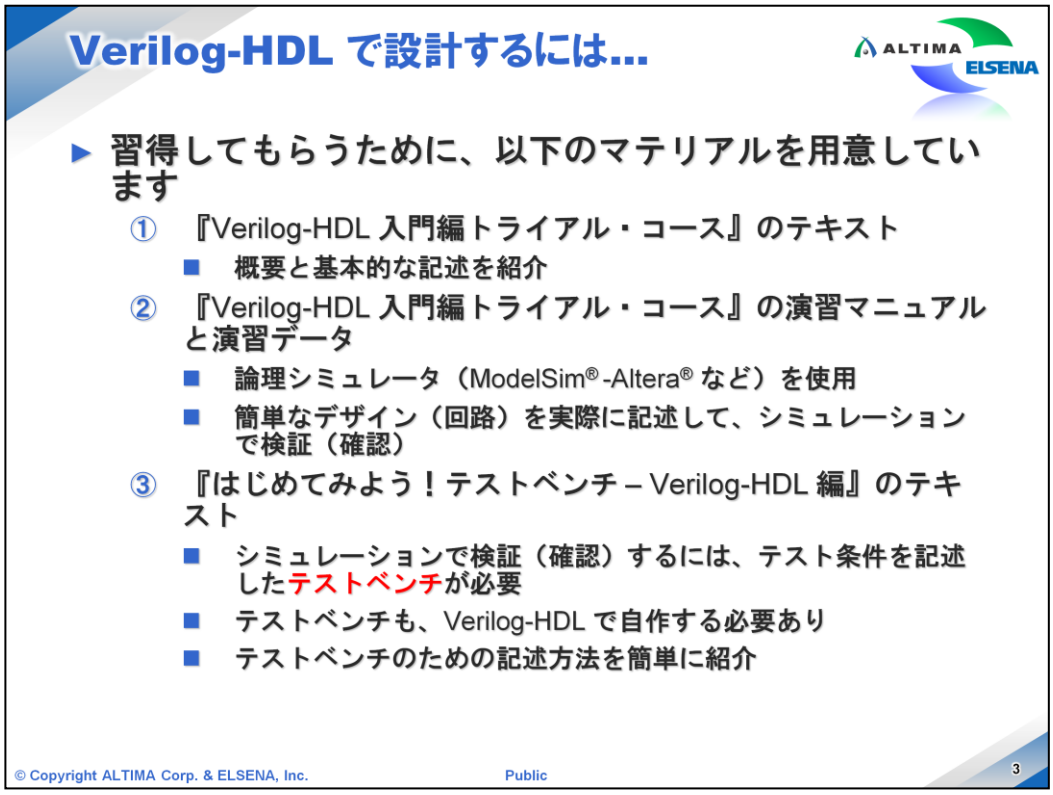
『Verilog-HDL 入門編トライアル・コース』を受講して頂き、誠にありがとうございます。



本ワークショップは、Verilog-HDL 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、Verilog-HDL の概要が理解でき、基本的な記述方法が習得できると思います。



Verilog-HDL 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『Verilog-HDL 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『Verilog-HDL 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。

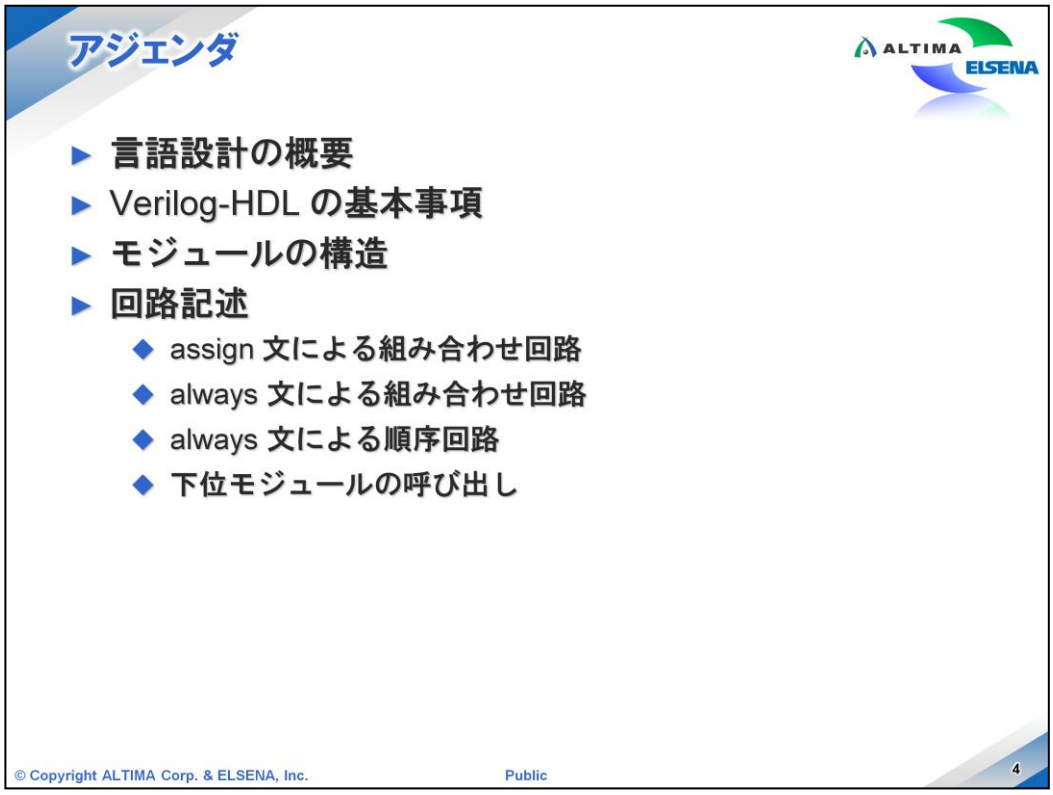
演習を実施するには、ModelSim-Altera などの論理シミュレータが必要です。

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。

Verilog-HDL 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 – Verilog-HDL 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。



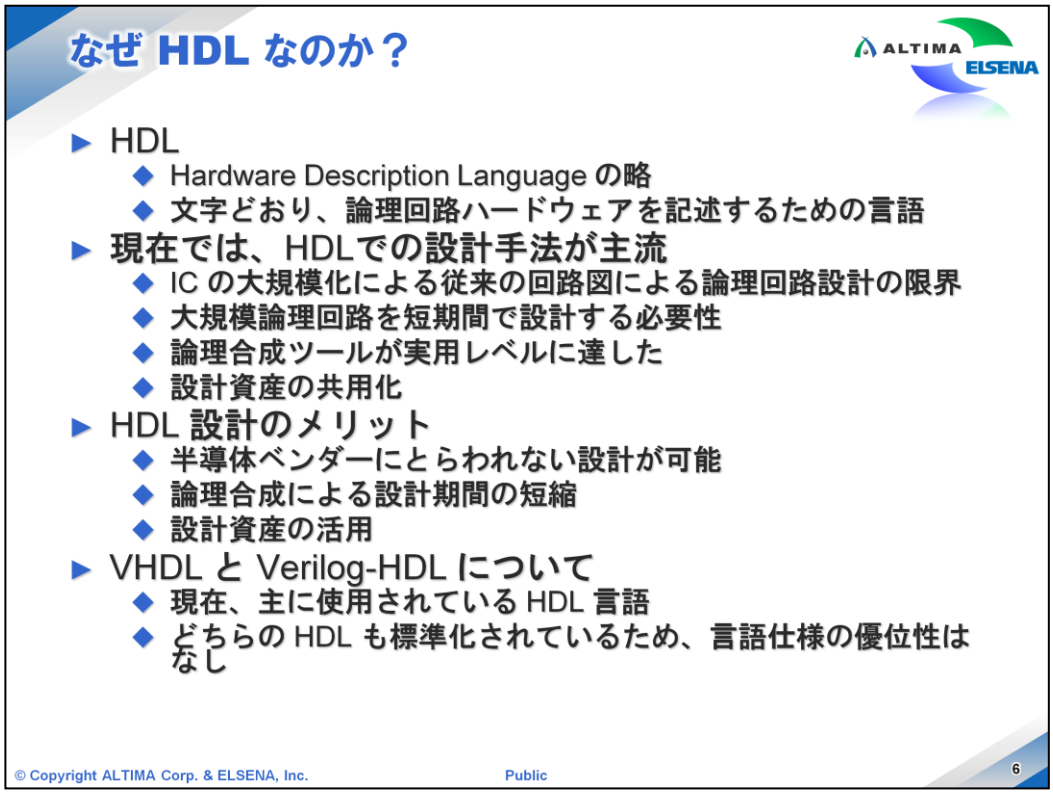
本ワークショップのアジェンダです。

最初に、言語設計の概要を説明し、続けて、Verilog-HDL の基本事項とモジュールの構造を説明します。

その後、実際の回路記述の方法として、assign 文による組み合わせ回路、always 文による組み合わせ回路、always 文による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モジュールの呼び出しの記述方法を説明します。

それでは、『言語設計の概要』に移ります。



なぜ HDL を習得する必要があるのでしょうか？

Hardware Description Language の各単語の頭文字をとって HDL と一般的には呼ばれており、文字通り、論理回路ハードウェアを記述するための言語です。

現在では、HDL による設計が主流となっています。

デバイスの大規模化によって従来の回路図による設計手法が限界となってきています。

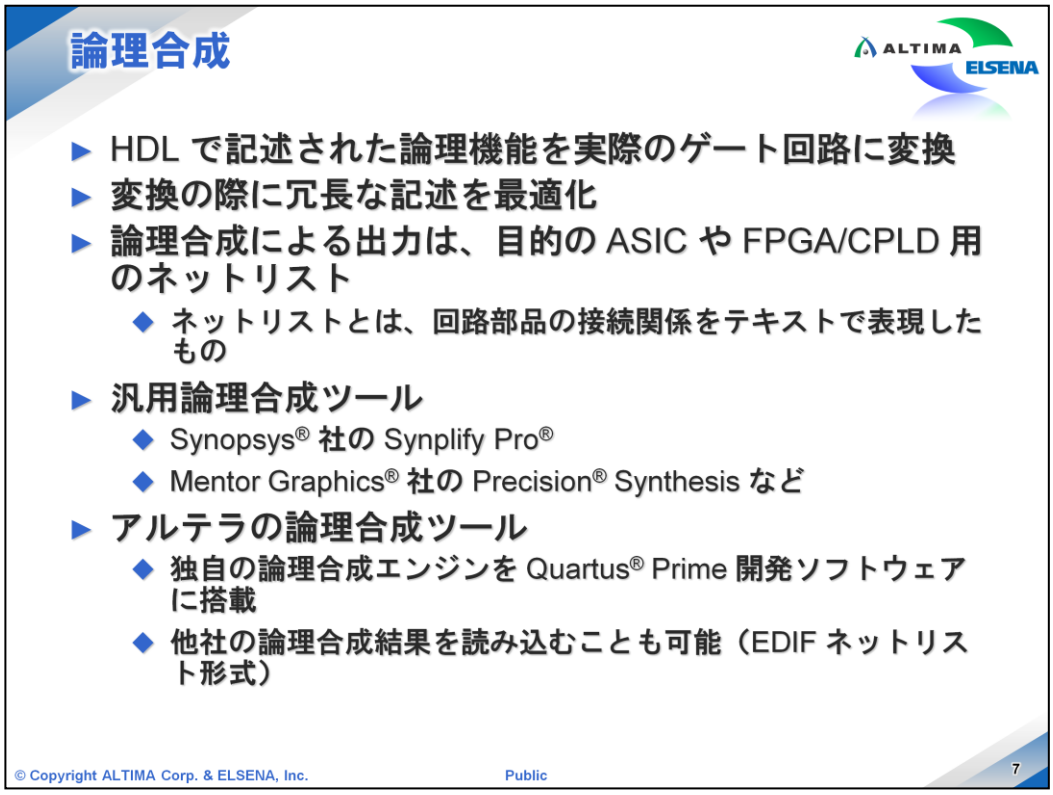
開発期間が年々短くなる傾向がある反面、大規模な論理回路を短期間で設計する必要に迫られているといった実情があります。

HDL による設計は、その点慣れれば効率良く設計することができます。

HDL のメリットは、半導体ベンダーにとらわれない設計が可能であることと、論理合成による設計期間の短縮、設計資産の活用などが挙げられます。

そして、もう 1つのメジャーな HDL として VHDL があります。

どちらもよく使用されている HDL で標準化されているため、言語仕様の優位性は特にありません。



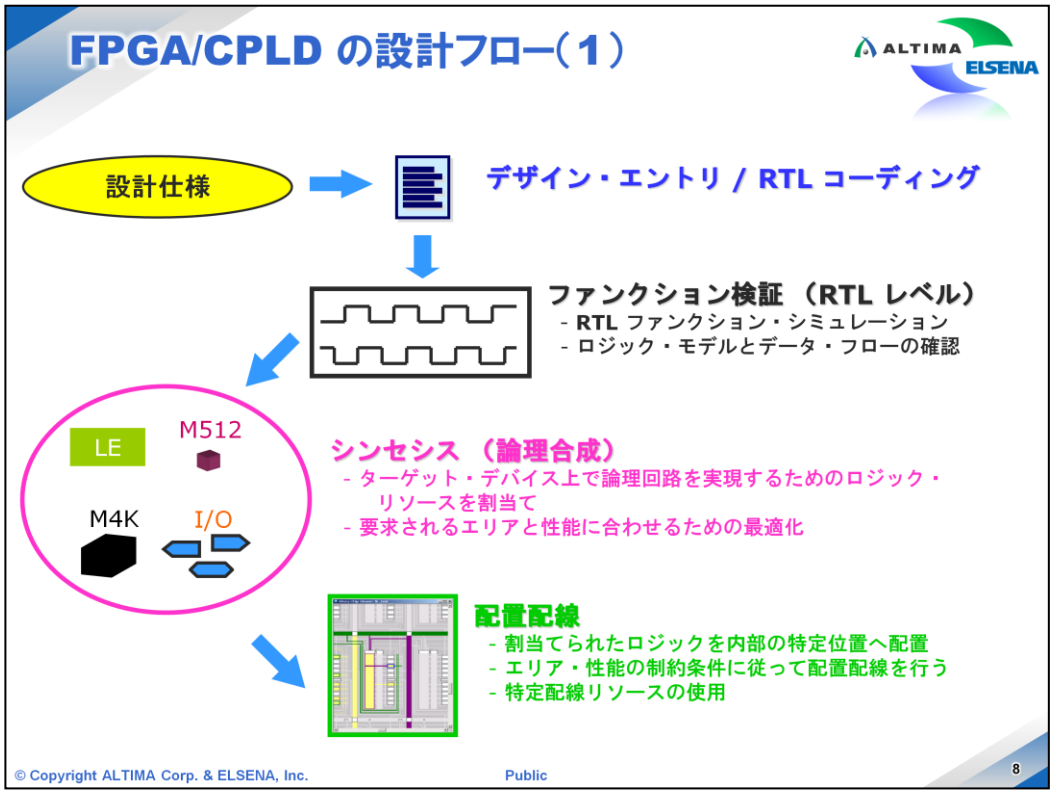
HDL で論理設計したら、実際のゲート回路に変換するために論理合成を行います。

もし、冗長な記述があると論理合成ツールが判断したら、自動的にツールが最適化してくれて、ASIC や FPGA/CPLD などターゲットのデバイス用のネットリストを出力します。

汎用論理合成ツールとしては、Synopsys 社の Synplify Pro や Mentor Graphics 社の Precision Synthesis が挙げられます。

アルテラの Quartus Prime 開発ソフトウェアにも論理合成エンジンが搭載されているので、そのまま配置配線のプロセスに移ることもできます。

また、汎用の論理合成ツールで生成されたネットリストを取り込んで、配置配線のプロセスに移ることもできます。



一般的な FPGA/CPLD の設計フローを紹介します。

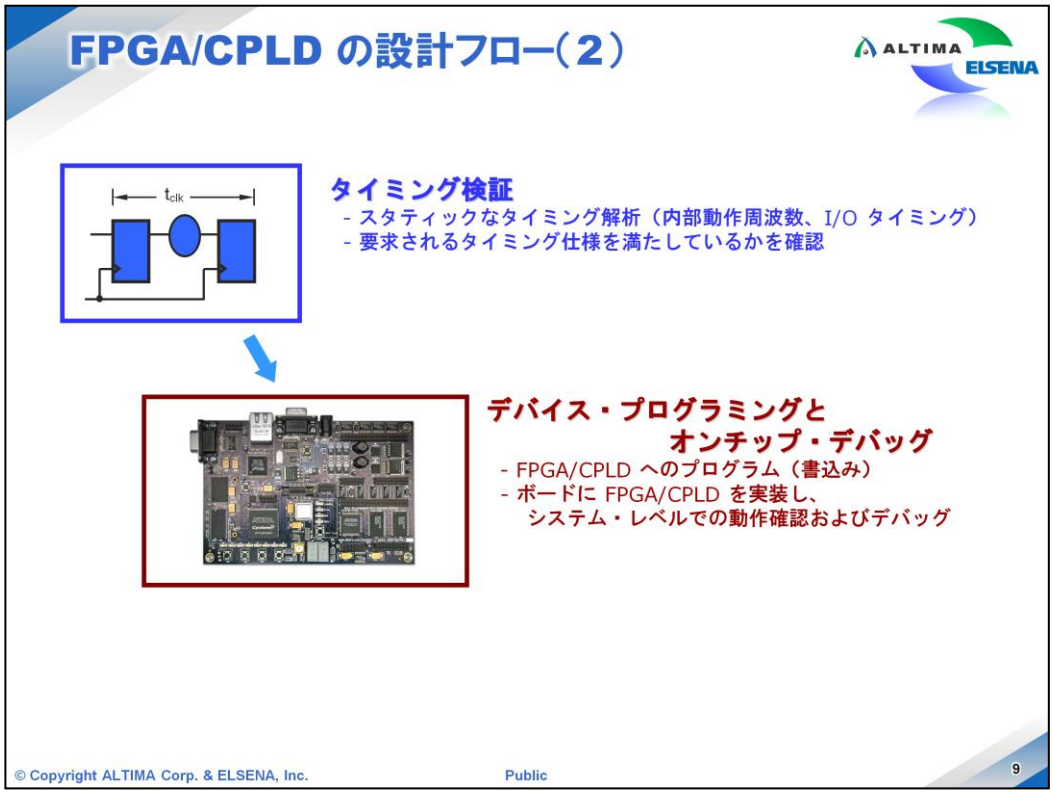
まず、装置やボード・レベル、FPGA/CPLD 内の設計仕様の検討を行った後に、HDL による実際の設計作業に入ります。

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。

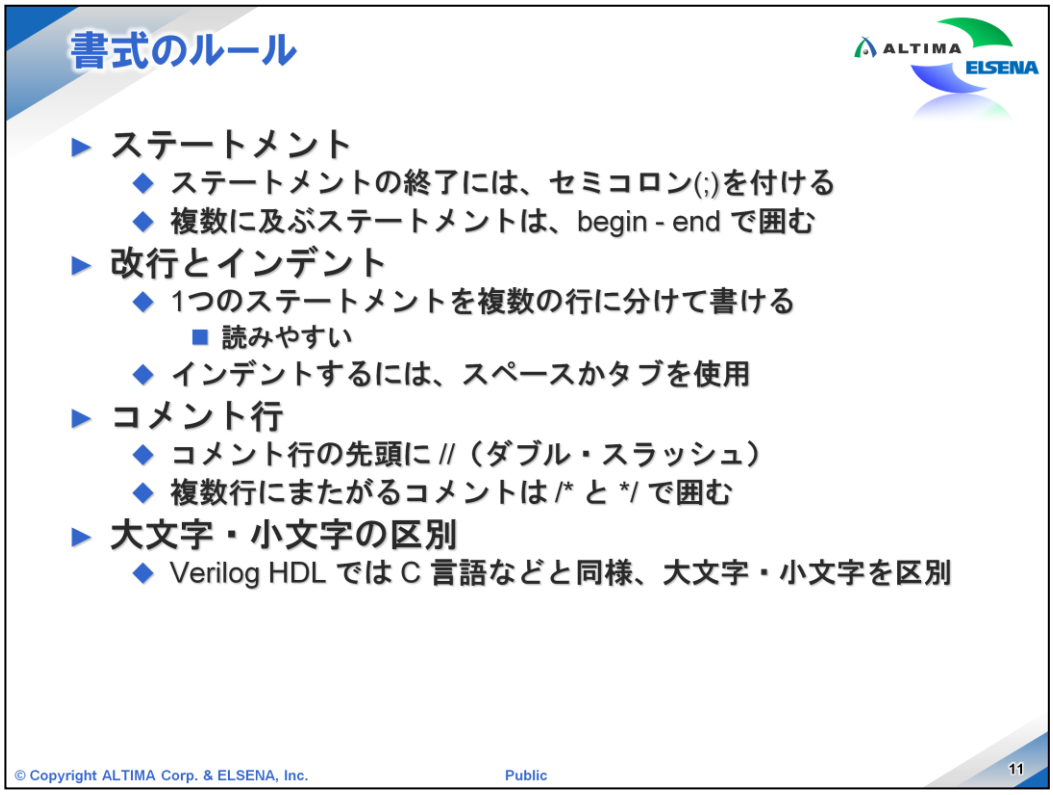
これらは、ツールが行ってくれます。



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。

次に、『Verilog-HDL の基本事項』の説明に移ります。



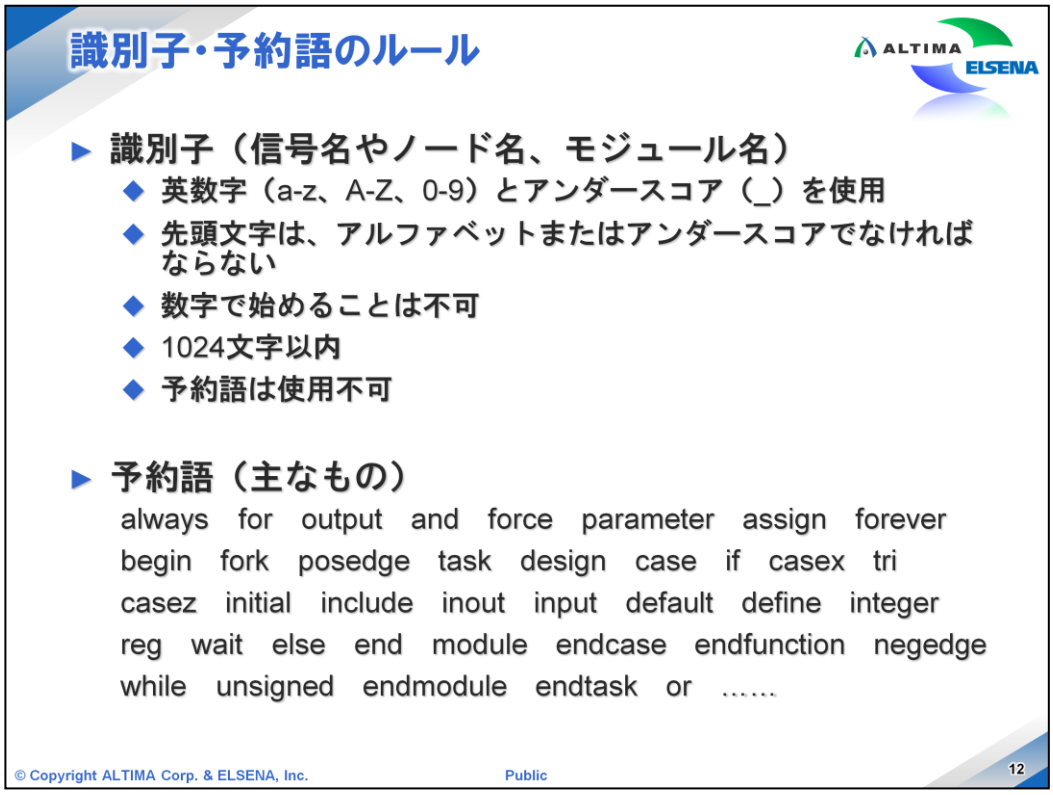
書式のルールについて、説明します。

まずステートメントですが、ステートメントの終了にはセミコロン（;）を付けます。
そして、ステートメントが複数に及ぶ場合は、begin – end で囲みます。

次に、改行とインデントですが、読みやすくするために 1つのステートメントを改行して複数の行に分けることができます。
インデントするには、半角スペースがタブを使用します。

記述した部分をコメント行にしたい場合、ダブル・スラッシュ（//）を使います。
ダブル・スラッシュ以降は、その行末までがコメント扱いになります。
また、複数行をまとめてコメント行にしたい場合は、スラッシュとアスタリスク、アスタリスクとスラッシュで囲みます。

また、Verilog-HDL では、アルファベットの大文字と小文字を区別します。

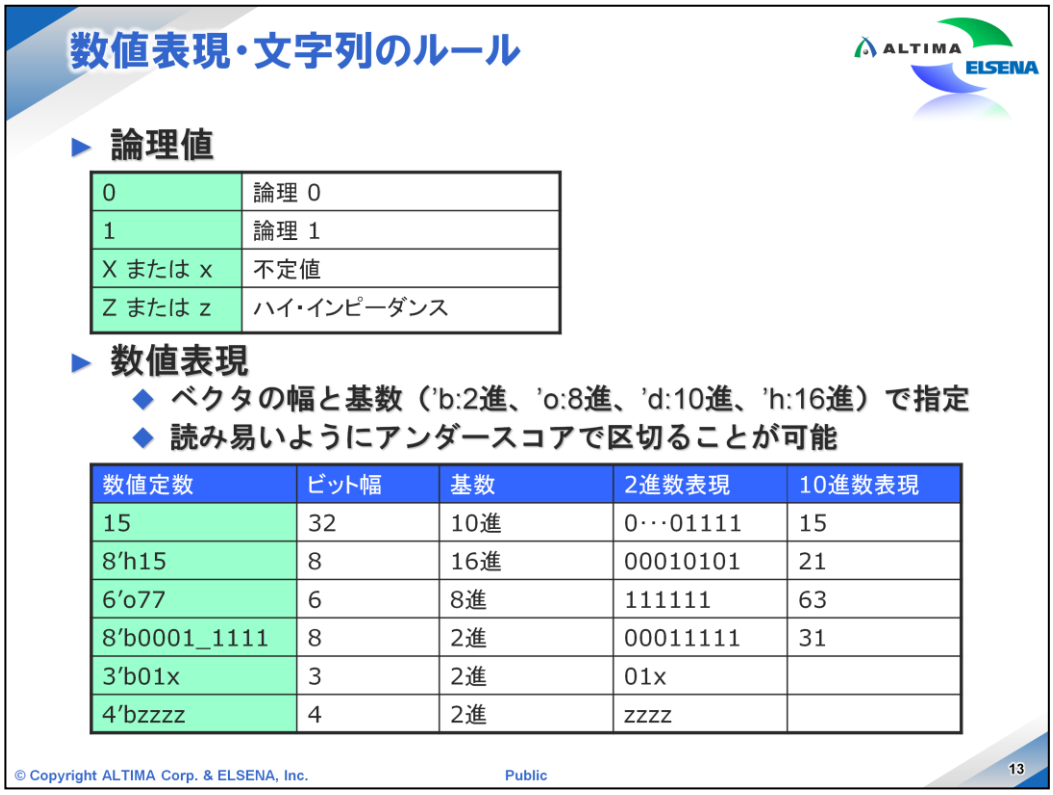


次に、信号名やノード名、モジュール名に使用する識別子に関するルールです。

識別子には、英数字とアンダースコアを使用することができますが、先頭の文字はアルファベットまたはアンダースコアでなければなりません。

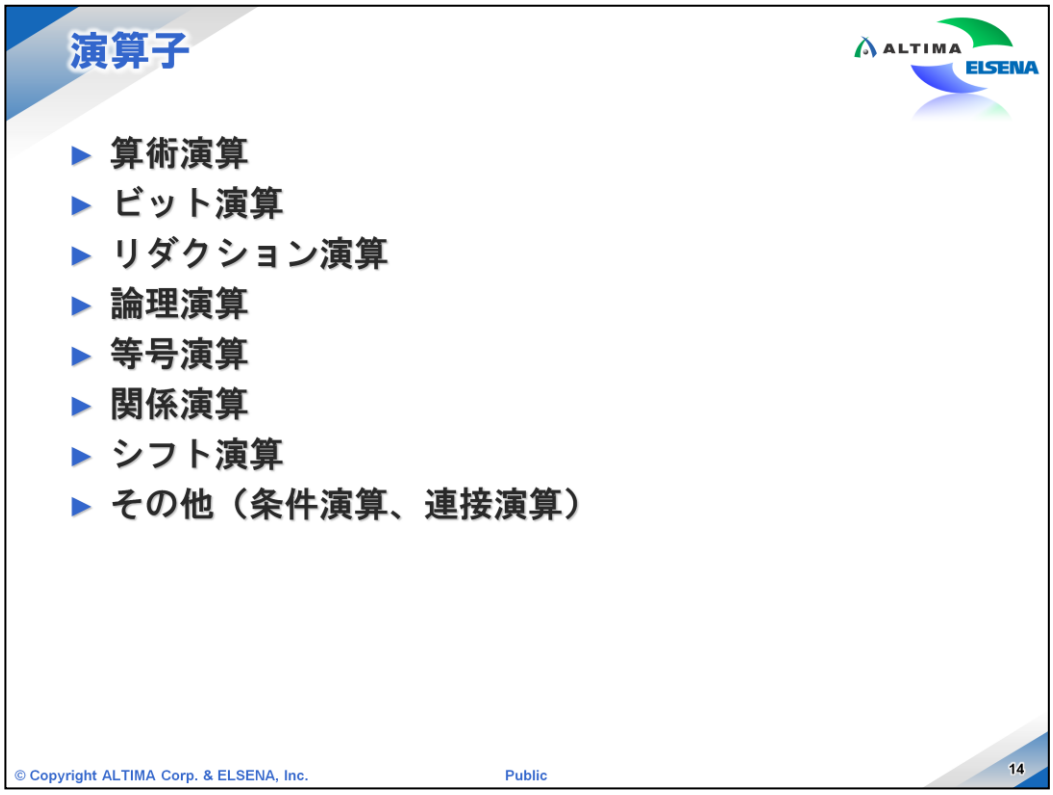
文字数は 1024 以内で、予約語を使用することはできません。

予約語は Verilog-HDL の文法で決められており、識別子として予約語をそのまま使用することはできませんので、注意してください。

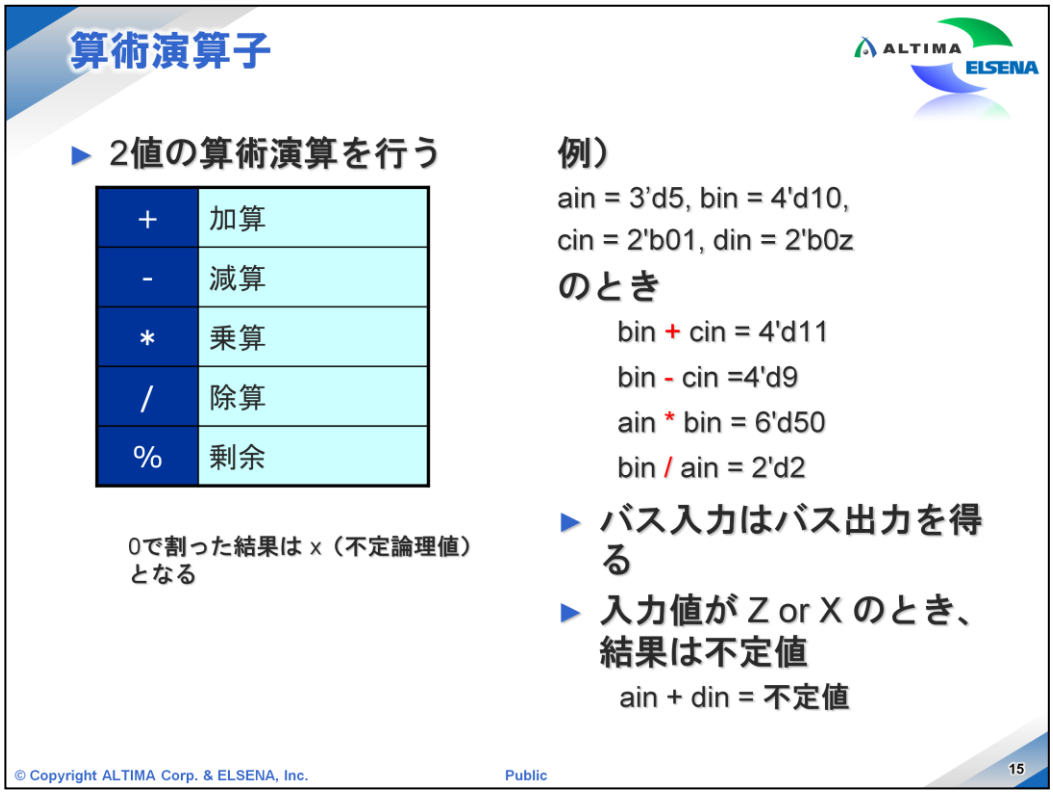


論理値は論理0 と論理 1、不定値、ハイ・インピーダンスの 4種類です。

数値表現については、ベクタの幅（ビット幅）と基数で指定します。
基数は何進数の数値かを表し、2進と 8進、10進、16進の 4種類があります。
読みやすくするために、例えば 4ビットずつアンダースコアで区切ることができます。

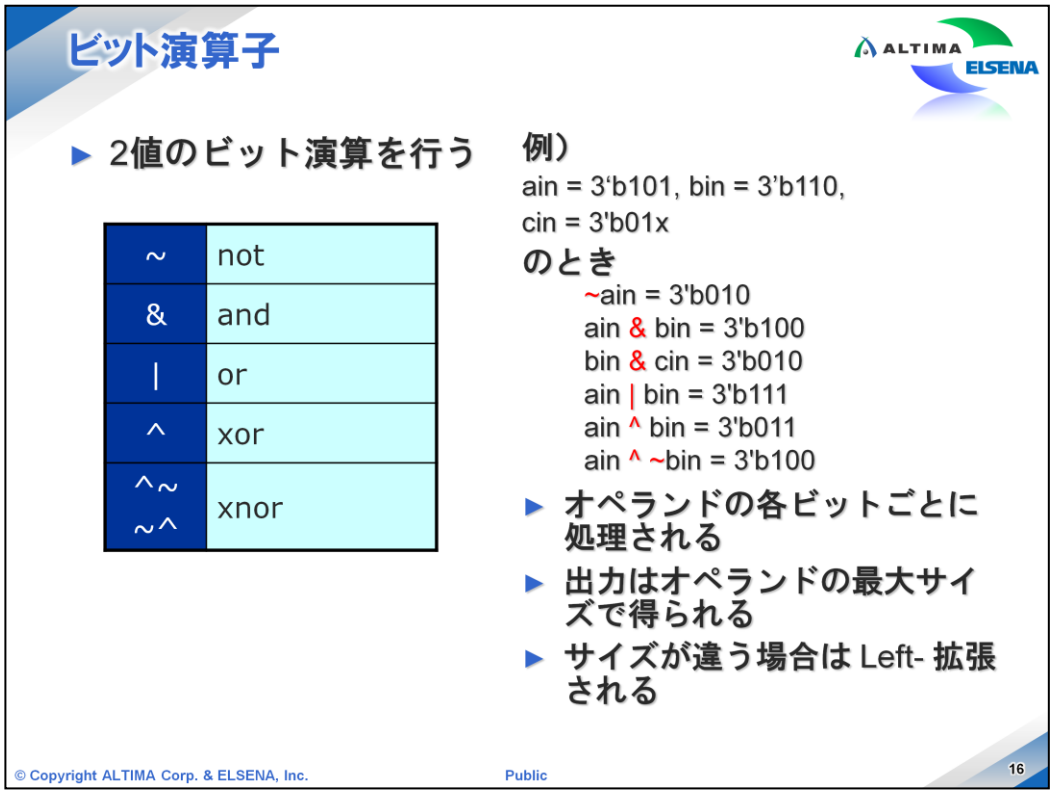


次に、演算子について細かく見ていきたいと思います。



まず、算術演算子です。
2値の算術演算を行うときに使用します。
加算と減算、乗算、除算、剰余があります。

右側に例を示しました。
バス入力の場合、バス出力を得ます。
そして、入力値がハイ・インピーダンスや不定値の場合、結果は不定値となります。



次に、ビット演算子です。

2値のビット演算を行うときに使用します。

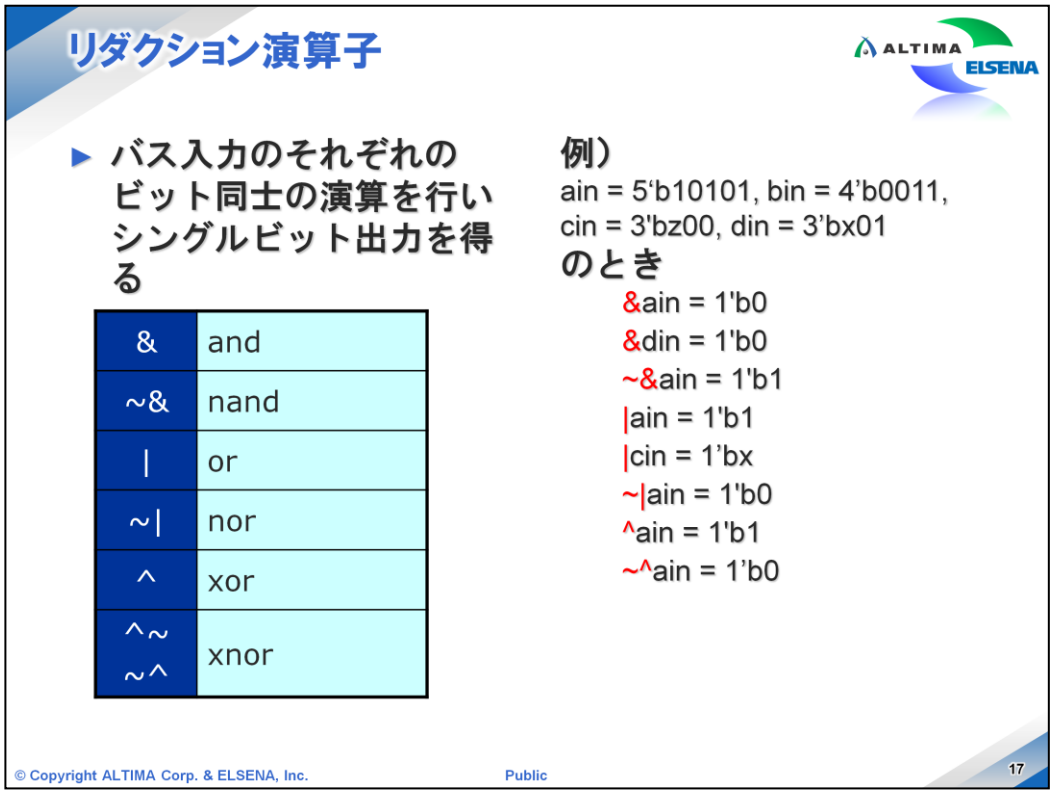
not と and、or、xor、xnor があります。

右側に例を示しました。

ビット演算は、演算の対象となる値や変数、つまりオペランドの各ビットごとに処理されます。

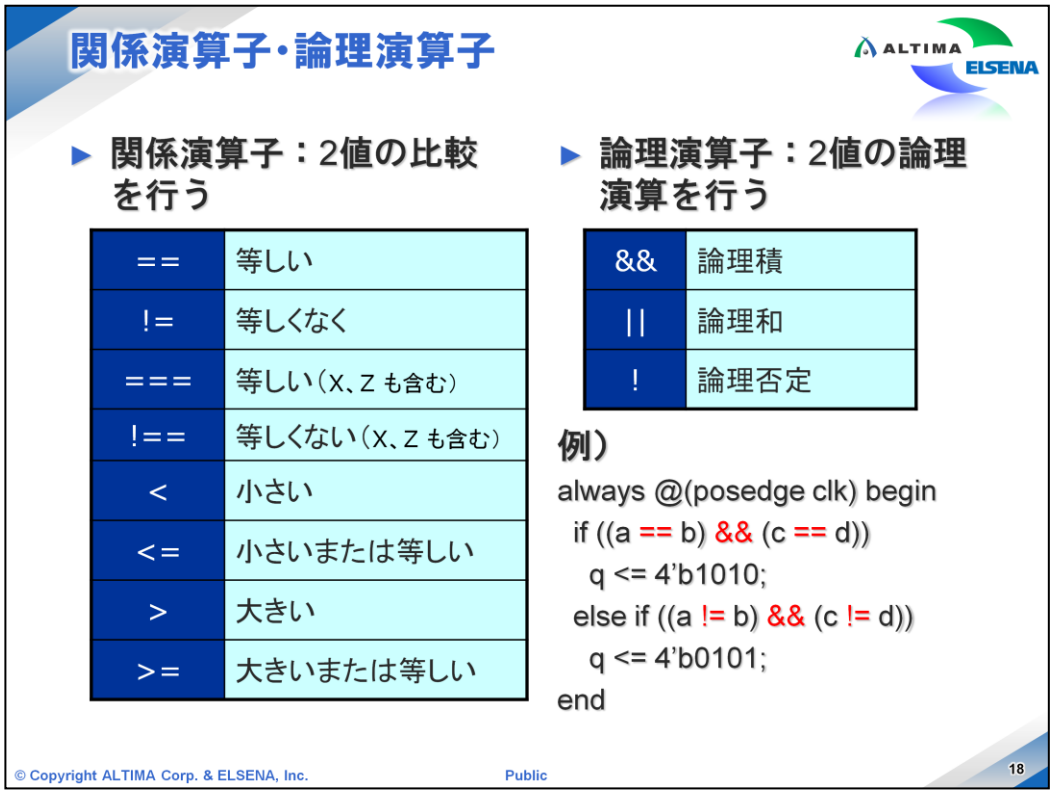
出力は、オペランドの最大サイズで得られます。

サイズが異なる場合、Left - 拡張されます。



次に、リダクション演算子です。
バス入力の各ビット同士の演算を行い、出力はシングル・ビットを得ます。

右側に例を示しました。
結果がシングル・ビットになっていることがわかると思います。



次に、関係演算子と論理演算子です。

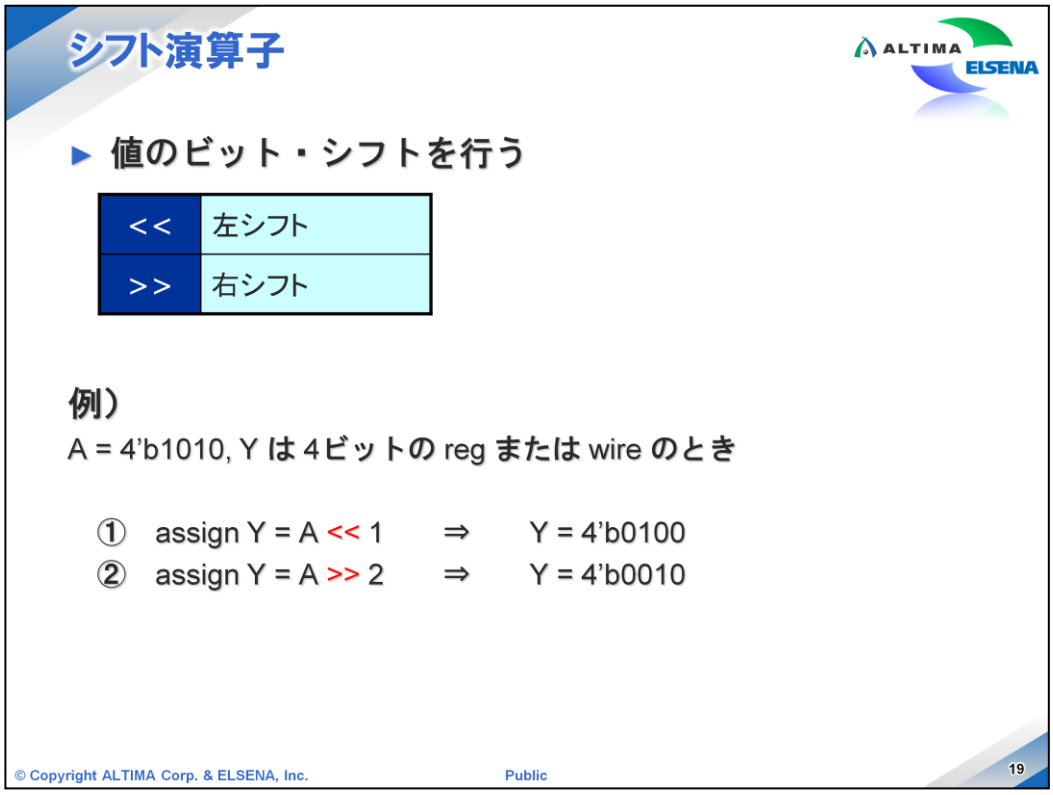
関係演算子は、2値の等しい、等しくない、大小を比較する時に使い、論理演算子は 2値の論理演算を行うときに使います。

右下に例を示しました。

if 文は後ほど説明しますが、if 文のところは「a と b が等しく、かつ c と d が等しい時」という意味になります。

関係演算子と論理演算子が両方使われています。

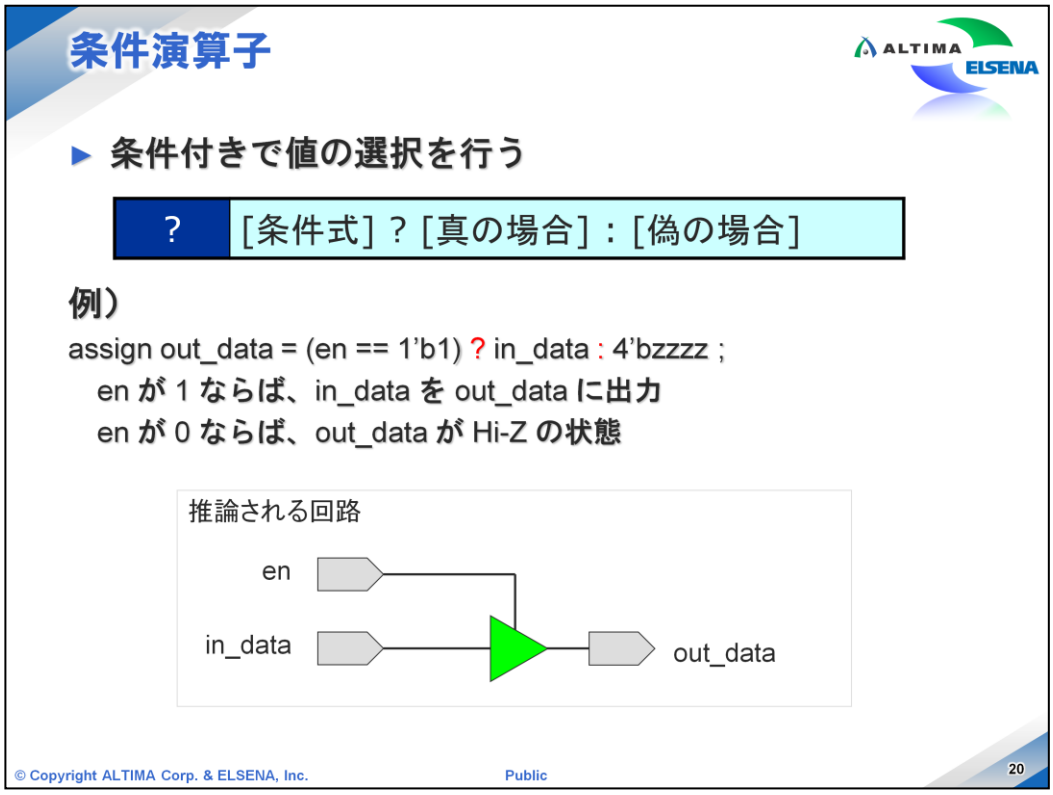
この条件が満たされたとき、つまり真の時に、1行下のステートメントが実行されます。



次に、シフト演算子です。

シフト演算子は、各ビットの値を左や右にシフトさせます。
左シフトは MSB 側にシフトされ、LSB 側には 0 が補充されます。
逆に右シフトは LSB 側にシフトされ、MSB 側には 0 が補充されます。

下に例を示しました。
例①は、左に1ビットシフトさせています。
例②は、右に 2ビットシフトさせています。



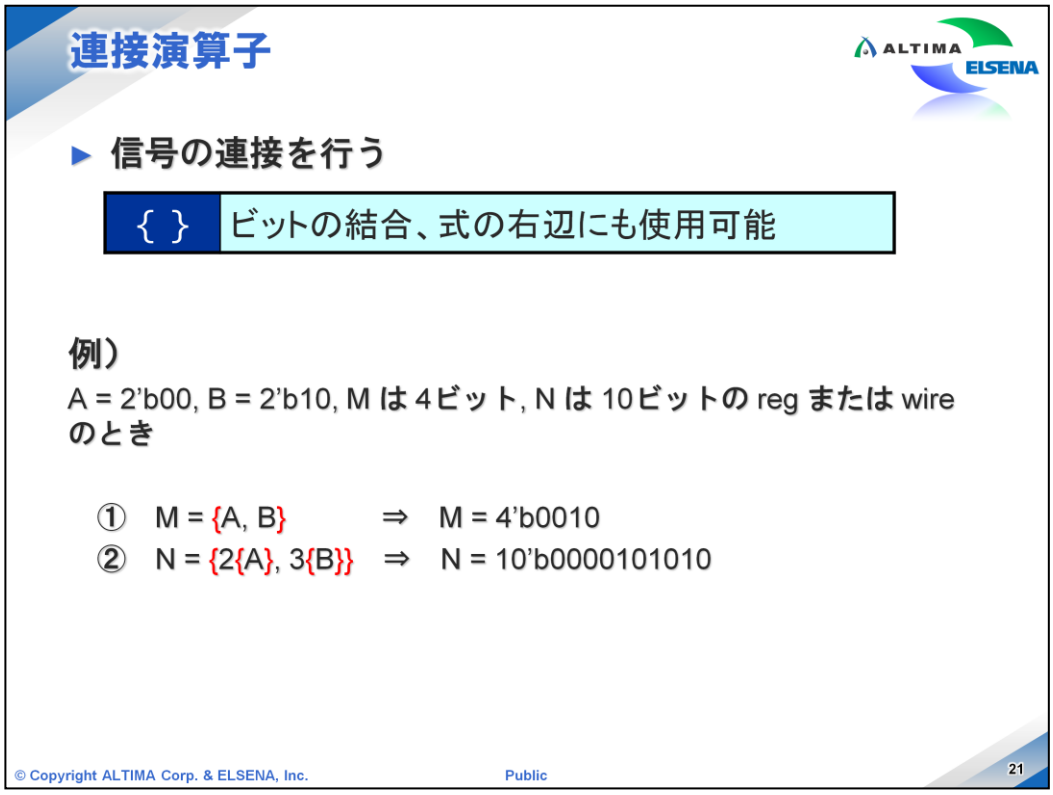
続いて、条件演算子です。

条件を記述して、真の場合と偽の場合の値や変数、信号の選択を行います。

下に例を示しました。

条件式のところに書かれているように en が 1 という条件に対して、真の場合は in_data が out_data に出力され、偽の場合は out_data がハイ・インピーダンスになります。

この時推論される回路は、下のようなイネーブル制御付きのバッファとなります。

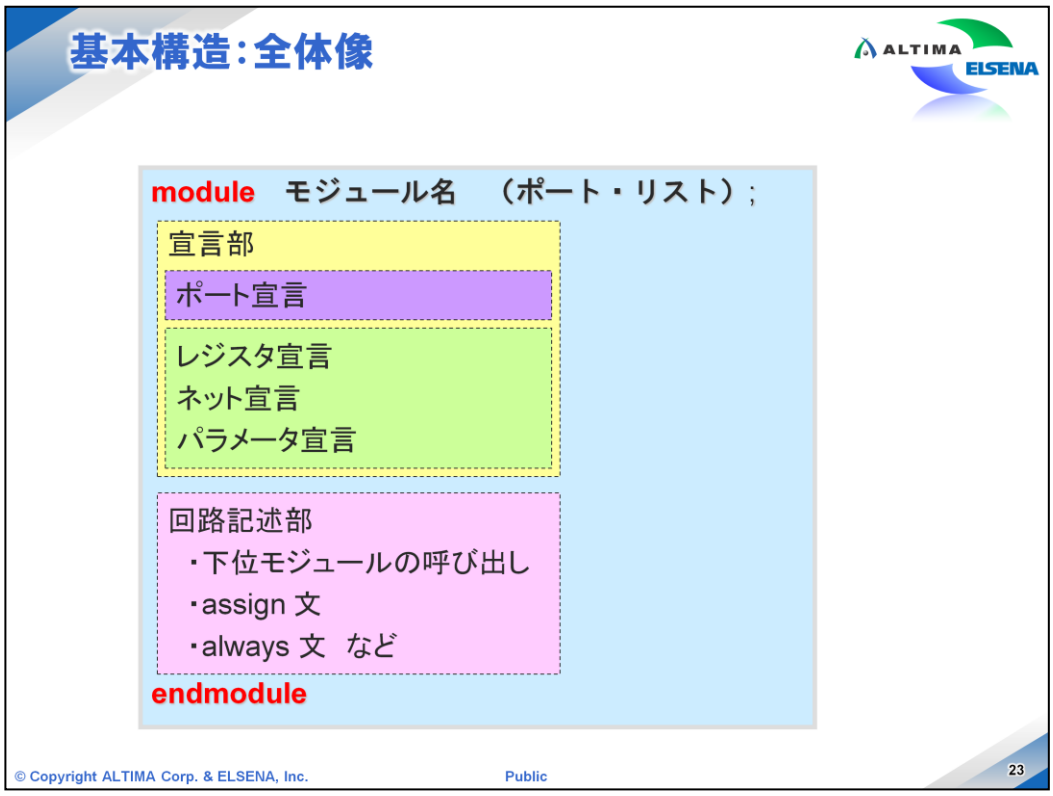


最後に、連接演算子です。

ビットを結合するときに使用します。
式の右辺にも使用できます。

下に例を示しました。
例①は、A の 2ビットと B の 2ビットを連接させています。
例②は、A を 2つ、B を 3つの計 10ビットに連接させています。

次に、『モジュールの構造』の説明に移ります。



Verilog-HDL の基本構造の全体像は、ここにある通りです。

まず module という予約語の後に任意のモジュール名を記述し、その後の括弧内にポート・リストを記述します。

ポート・リストに記述する複数のポートは、カンマ（, ）で区切ります。

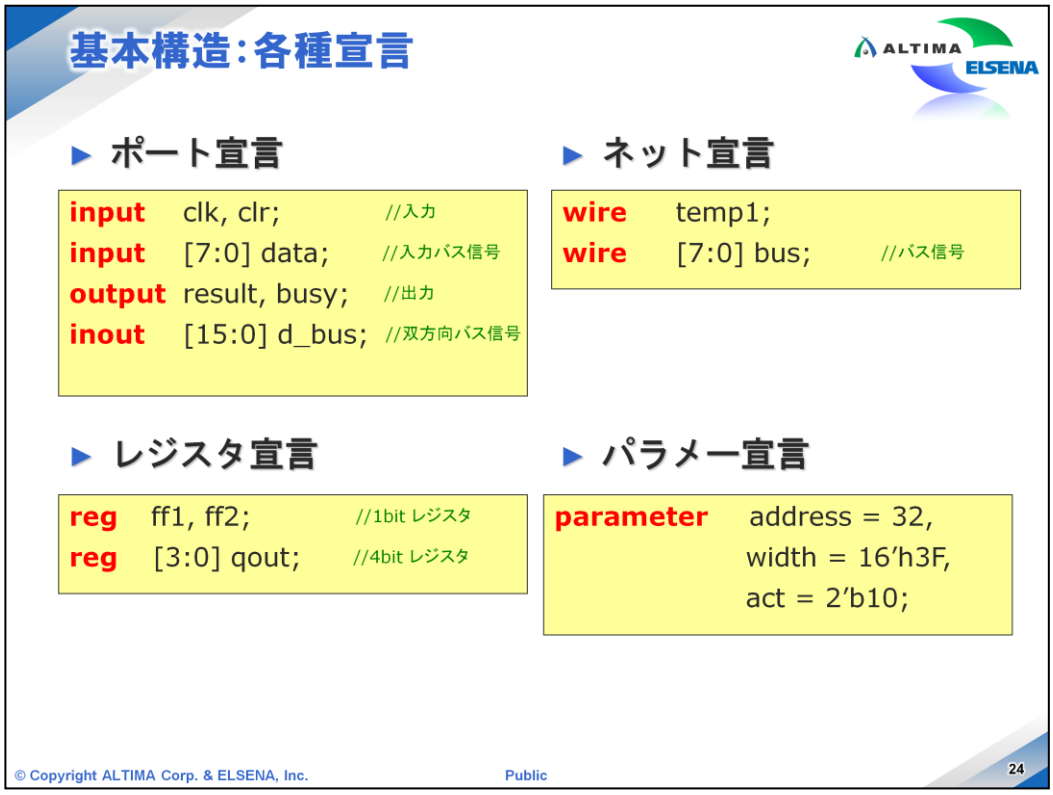
次に、宣言部が続きます。

宣言部には、ポート宣言やレジスタ宣言、ネット宣言、パラメータ宣言を必要に応じて記述します。

そして、回路記述部と続きます。

ここには、実際の回路の記述や下位モジュールの呼び出しなどを記述します。

最後に、endmodule で締めくくります。



宣言部ついて、少し細かく説明します。

まず、入力ポートや出力ポート、入出力ポートをポート宣言します。

入力の input や出力の output、双方向の inout の予約語を使用し、その後にポート名を記述します。

バス信号のように複数ビットのポートの場合は、四角括弧内にビット幅を記述します。

予約語やビット幅、ポート名の間は、半角スペースやタブを挿入します。

カンマ(,)で区切れば同一のステートメントに複数ポートを宣言できますが、同じビット幅である必要があります。

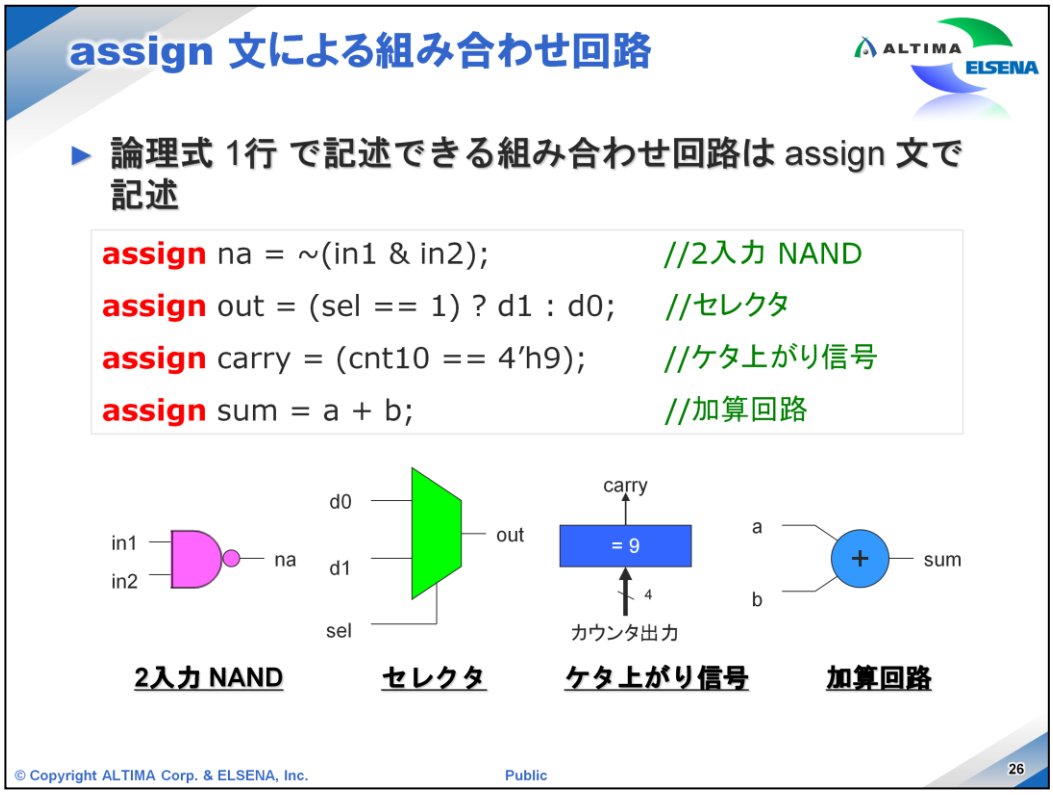
次に、レジスタ宣言は、モジュール内の信号のうち、値を保持する変数に対して宣言します。(後述する always 文の出力の信号は、レジスタ宣言をします。)

そして、ネット宣言は、モジュール内の信号のうち、配線として使用する変数に対して宣言します。

最後に、パラメータ宣言は、定数を文字(パラメータ)で置き換えることで、見やすさや編集のしやすさを狙ったものです。

必ず必要なものではありませんが、慣れれば非常に便利です。

ここからは、実際の『回路記述』の説明に入ります。



まず最初に、assign 文による組み合わせ回路のうち、論理式 1行で記述できるものを例を交えて紹介します。

1つ目は、NAND 回路です。

assign に続いて、左辺に出力信号名を記述し、右辺にはビット演算子を使用して記述しています。

2つ目は、セレクタ回路です。

assign に続いて、左辺に出力信号名を記述し、右辺には関係演算子を使って条件を記述して条件演算子の ? を記述し、続いて真の場合と偽の場合の値や変数、信号を記述しています。

3つ目は、ケタ上がり信号です。

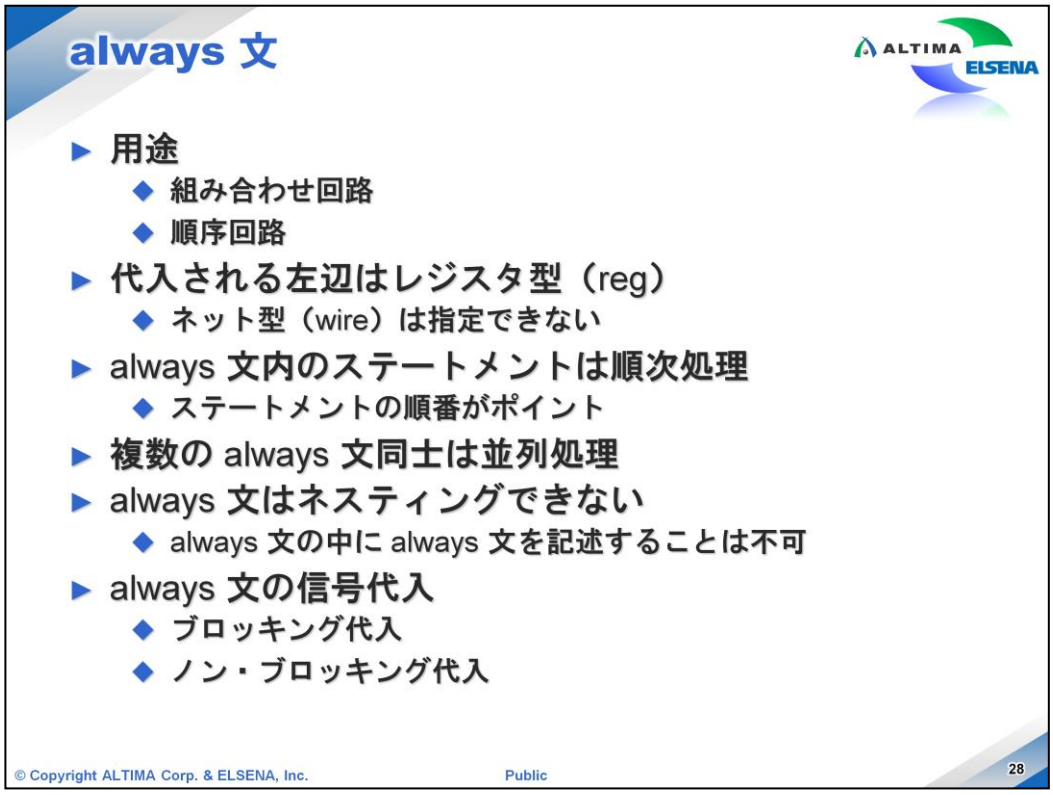
assign に続いて、左辺に出力信号名を記述し、右辺には関係演算子を使って条件を記述しています。

この例ではカウンタ出力の cnt10 が 9 の時に、carry が 1 となります。

4つ目は、加算回路です。

assign に続いて、左辺に出力信号名を記述し、右辺には算術演算子を使って計算式を記述しています。

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。



次に、always 文について説明します。

always 文は、組み合わせ回路や順序回路で使用されます。

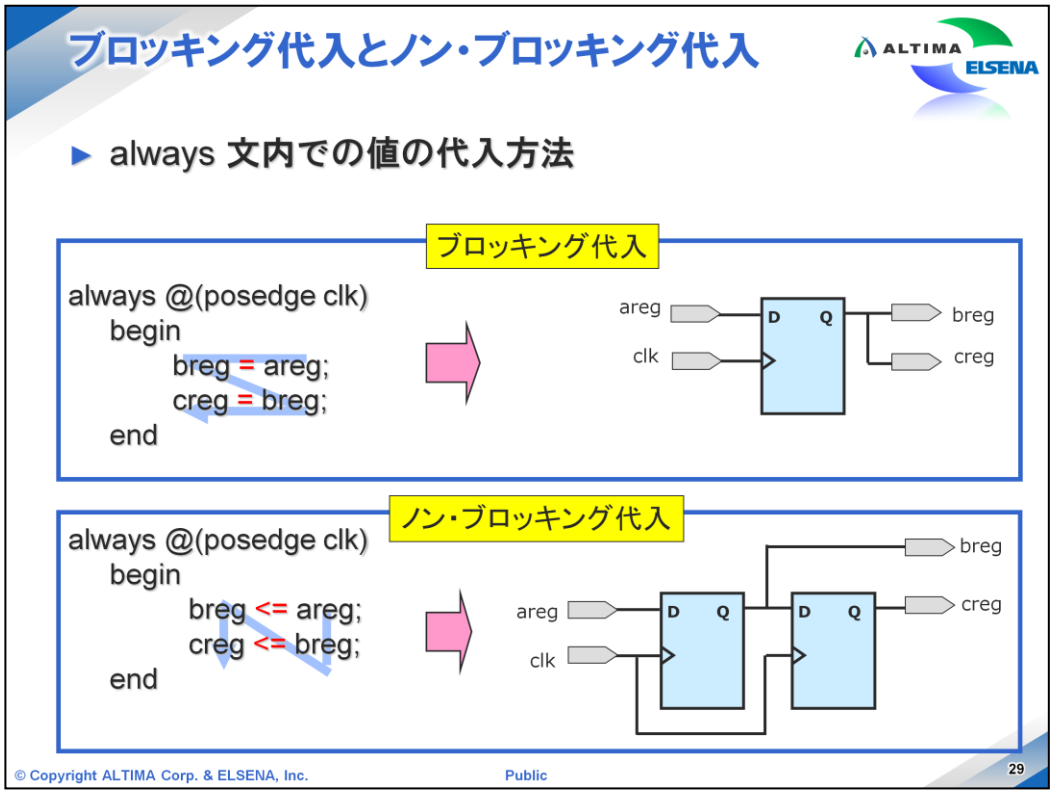
代入される左辺はレジスタ型である必要があります。

always 文内のステートメントは順次処理されるので、ステートメントの順番がポイントとなります。

複数の always 文がある場合、それらは並列処理されます。

always 文はネスティングできません。
つまり、always 文の中に always 文を記述することはできません。

always 文の信号代入には、ブロッキング代入とノン・ブロッキング代入があります。



ブロッキング代入とノン・ブロッキング代入の違いは、レジスタを例にするとこのスライドの通りです。

右辺から左辺に代入するところの違いだけで、論理合成後で推論される回路は大きく異なります。

レジスタに値が代入されると、次にレジスタに代入されるまで値を保持します。

ブロッキング代入では式が評価すると同時に代入を実行しますが、ノン・ブロッキング代入では右辺の評価が完了しても、代入はその時刻での他の代入文の評価が完了するまで先延ばしされます。
つまり、ノン・ブロッキング代入では右辺の値が確定してから左辺への代入が実行され、この例のように areg と creg の間には 2つのレジスタが挿入されます。

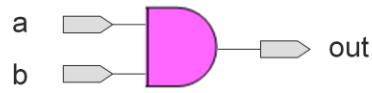
一般的に、ハードウェア設計では、ノン・ブロッキング代入を使用するケースが多いです。

always 文による組み合わせ回路

- 複雑な組み合わせ回路は always 文で記述
  - 出力は reg 宣言
  - always 文の @ 以降に、全ての入力を記述
    - センシティビティ・リスト
  - always 文の中に動作を記述

```
module adder (a, b, out);
 input       a, b;
 output      out;
 reg         out;

always @ (a or b)
         out <= a & b;
endmodule
```



それでは、always 文を使用した組み合わせ回路について紹介します。

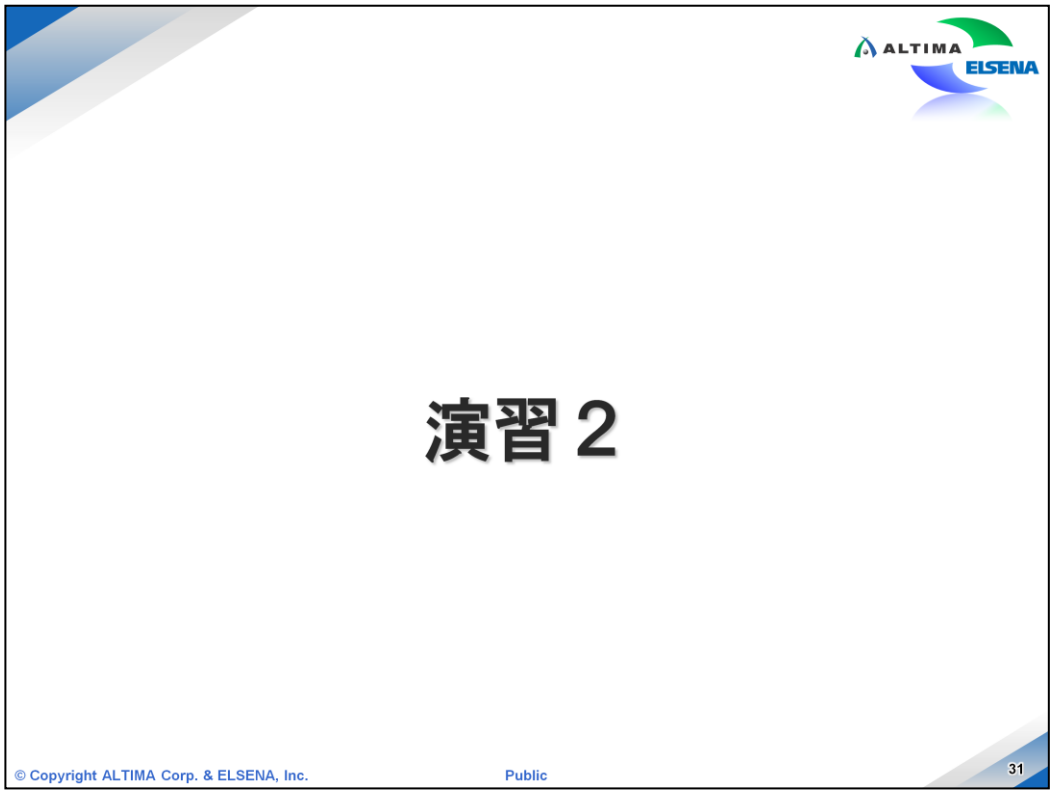
複雑な組み合わせ回路は always 文で記述しますが、ここでは簡単な組み合わせ回路である AND 回路で説明します。

先ほども説明しましたが、always 文の出力はレジスタ型となります。

そして、always の後にアット・マーク(@)を記述し、その後の括弧内にすべての入力を記述します。

これをセンシティビティ・リストと言います。

その後に、動作を記述します。



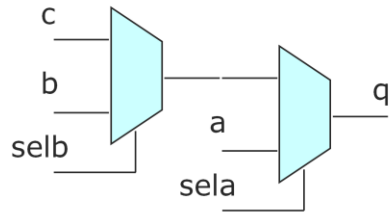
それでは、演習2を実施してください。
詳細は、演習マニュアルをご覧ください。

if 文

```
module case_sel (a, b, c, sela, selb, q);
   input   a, b, c;
   input   sela, selb;
   output   q ;
   reg      q;

always @(sela or selb or a or b or c)
 begin
   if (sela == 1'b1)
      q <= a;
   else if (selb == 1'b1)
      q <= b;
   else
      q <= c;
 end
endmodule
```

- 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
- always ブロック中で使用可能



続いて、if 文を紹介します。

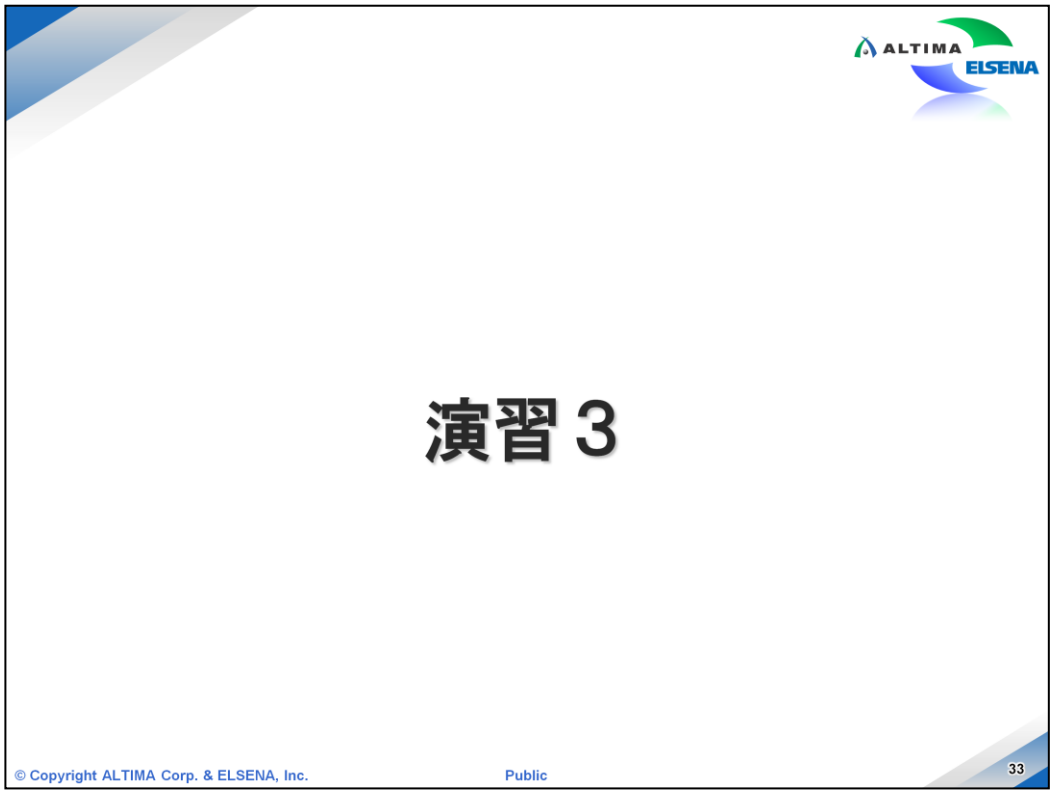
if 文は、条件を記載して、その条件を満足した場合、つまり真の時に実行するステートメントを続けて記載します。

そして、if 文には優先順位が存在し、記述した順番に評価されます。

すべての条件が偽であれば、else のステートメントが実行されます。

途中で真の条件がありステートメントが実行された場合は、それ以降は評価されずに if から抜け出します。

if 文は、always ブロック内で使用できます。

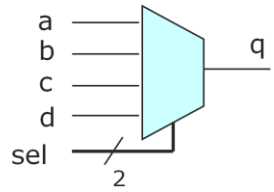


それでは、演習3を実施してください。
詳細は、演習マニュアルをご覧ください。

case 文

```
module case_sel (a, b, c, d, sel, q);
   input   a, b, c;
   input   [1:0]   sel;
   output   q ;
   reg      q;
always @(sel or a or b or c or d)
begin
   case (sel)
      2'b00 :
         q <= a;
      2'b01 :
         q <= b;
      2'b10 :
         q <= c;
      default :
         q <= d;
   endcase
end
endmodule
```

- 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
- default は、条件に無いその他の場合に実行される
- always ブロック中で使用可能



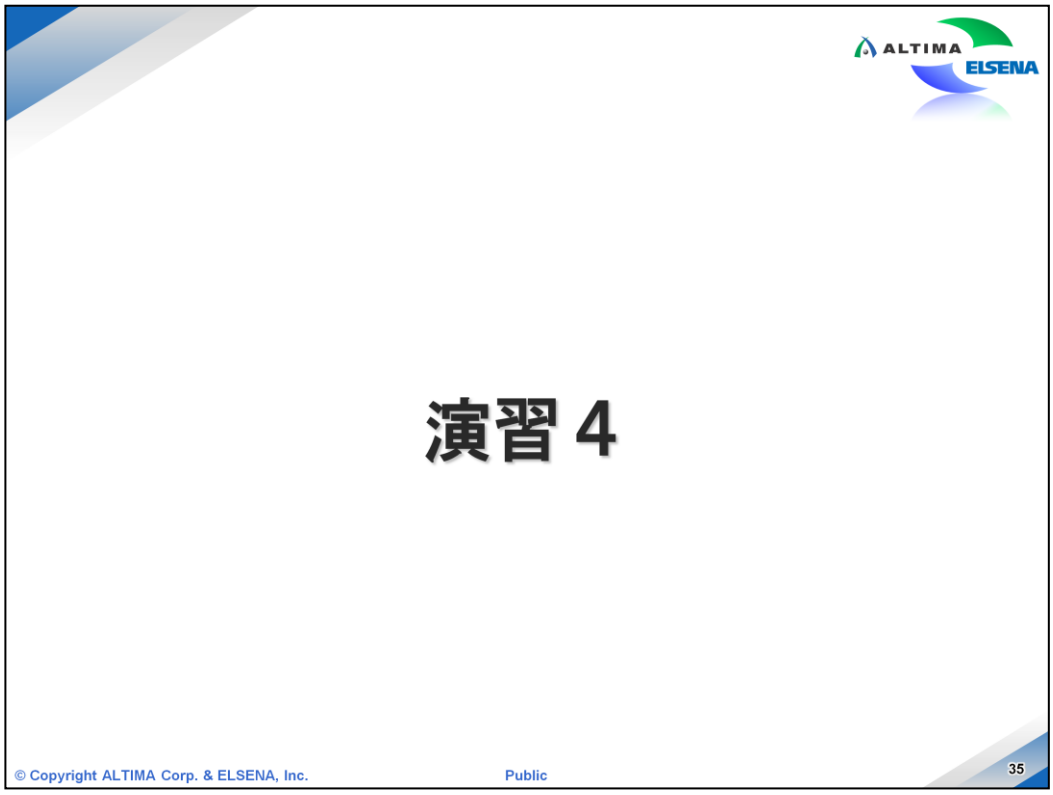
次に、case 文を紹介します。

case 文も条件と実行するステートメントを記述しますが、優先順位は存在しません。複数記述した条件は、同時に評価されます。

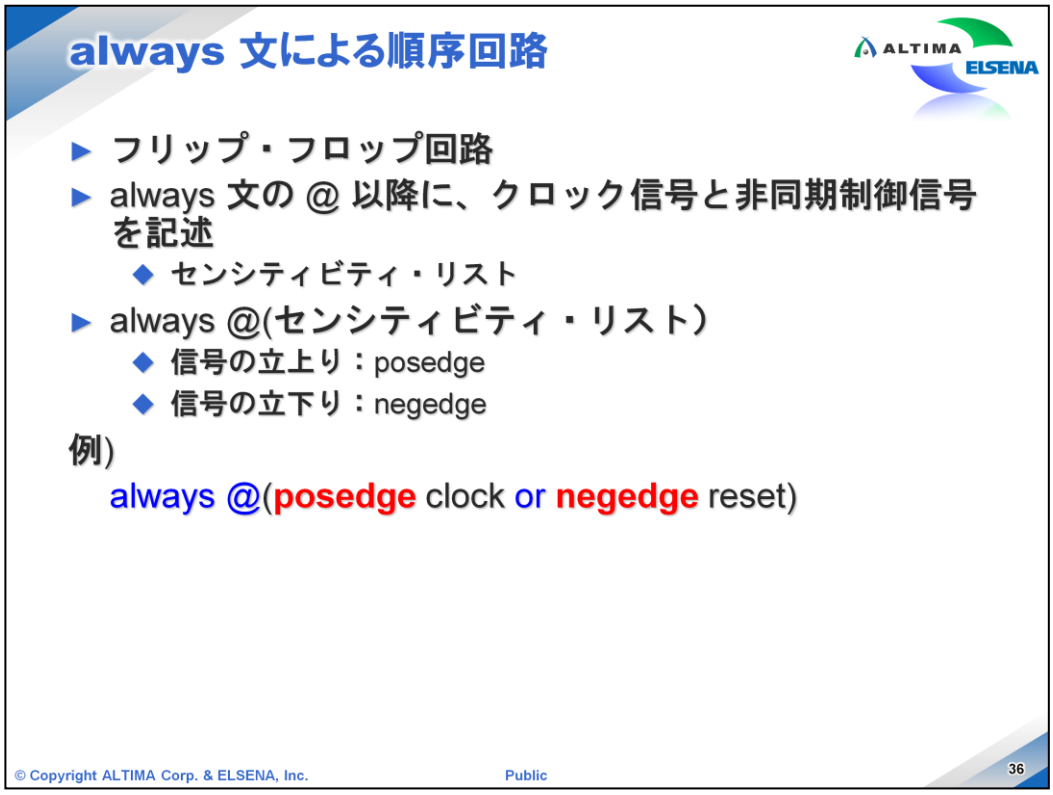
すべての条件を考慮する必要があり、条件が重ならないように気を付ける必要があります。

すべての条件が偽であれば、default のステートメントが実行されます。

case 文は、always ブロック内で使用できます。



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。

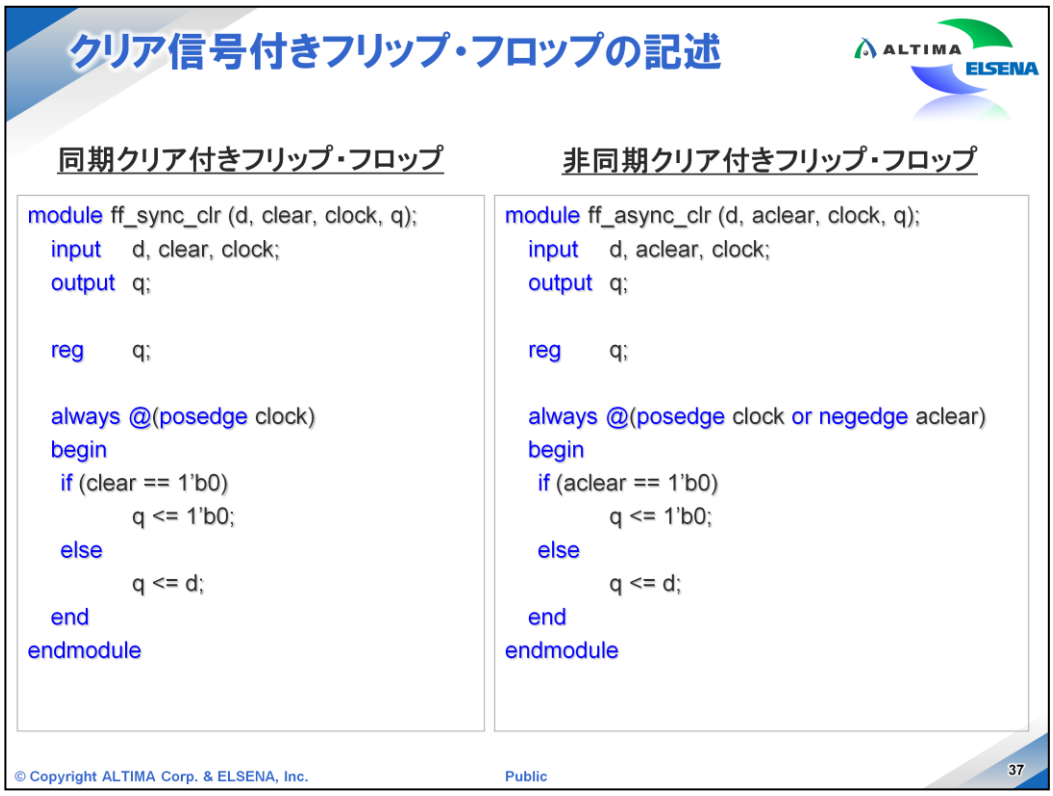


次に、always 文を使用した順序回路の説明をします。

always の後のアット・マーク（@）以降の括弧内に記述するセンシティビティ・リストにクロックや非同期制御信号を記述します。

信号の立ち上がりや立ち下がりに限定したい場合、ポスエッジ（posedge）やネグエッジ（negedge）を使います。

例えば、クロックの立ち上がりの場合、posedge clock と記述します。



クリア信号付きのフリップ・フロップの記述について説明します。

まずは、同期クリア付きフリップ・フロップです。

同期クリア付きフリップ・フロップなので、すべての動作がクロックのエッジに同期します。

従って、センシティビティ・リストにはクロックのエッジ、ここではクロックの立ち上がりである posedge clock と記述しています。

そして、if 文を使って、最初にクリアのための条件（clear 信号がゼロ（0）の時という条件）と実行するステートメントを記述します。

その後に、クリアされていない時に実行するステートメントを記述します。

右側は、非同期クリア付きフリップ・フロップです。

非同期クリア付きフリップ・フロップなので、一般的に非同期クリアが入った場合は最優先で出力がゼロにクリアされます。

従って、センシティビティ・リストにはクロックのエッジと非同期クリアのエッジ、ここではクロックの立ち上がりである posedge clock と非同期クリアの立ち下がりである negadge aclear と記述しています。

そして、if 文を使って、最初にクリアのための条件（aclear 信号がゼロ（0）の時という条件）と実行するステートメントを記述します。

その後に、クリアされていない時に実行するステートメントを記述します。

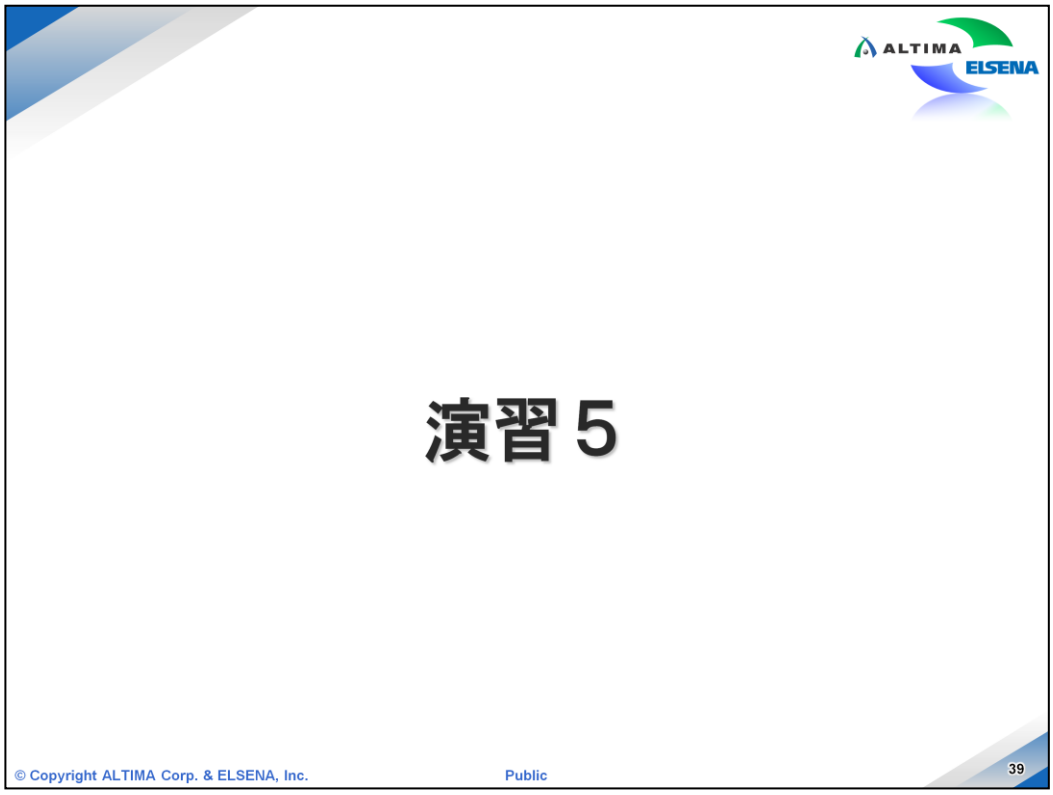
次に、クロック・イネーブル付きフリップ・フロップです。

一般的に、クロック・イネーブルがアクティブで、かつクロックのエッジに同期します。

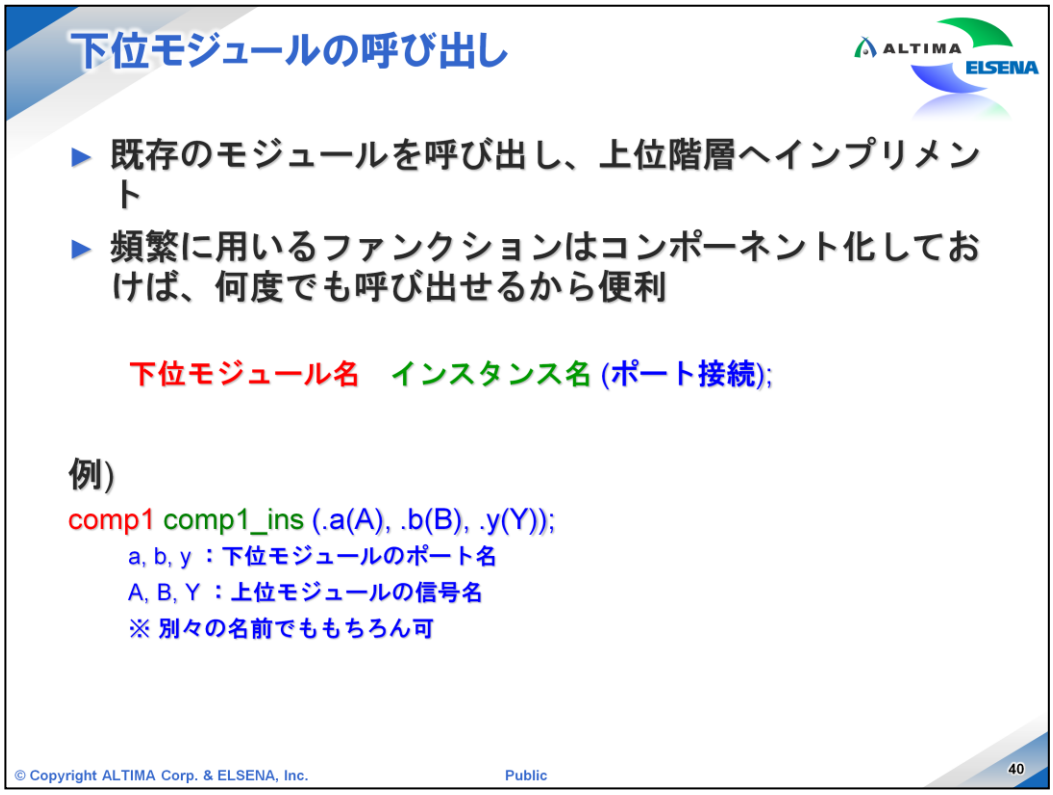
従って、センシティビティ・リストにはクロックのエッジ、ここではクロックの立ち上がりである posedge clock と記述しています。

そして、if 文を使って、クロック・イネーブルがアクティブの時という条件（enable 信号がイチ（1）の時という条件）と実行するステートメントを記述します。

ここではクリア信号がないタイプのフリップ・フロップを例にしているので、クリア信号に関する記述はありません。



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。



次に、下位モジュールの呼び出しについて説明します。

一般的に階層構造による設計をすることが多いと思いますが、そうなると上の階層から下の階層を呼び出す必要が生じます。

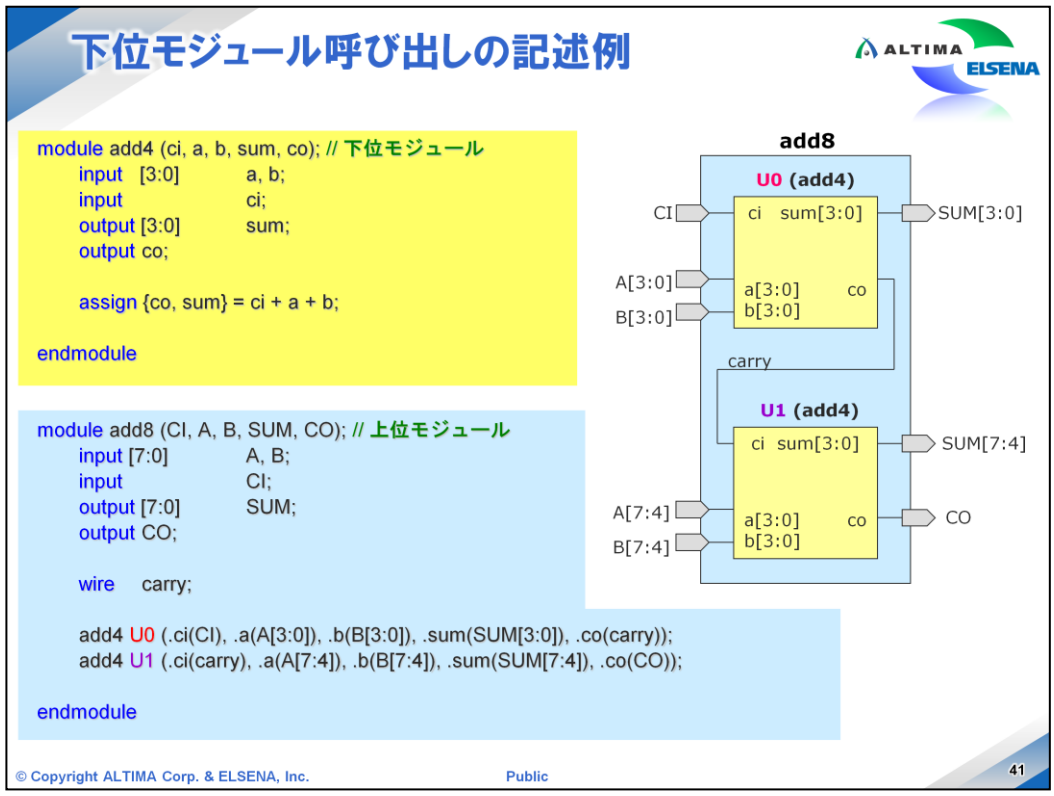
また、良く使用するファンクションをコンポーネント化しておけば、何度でも呼び出せて使用できるので、非常に便利です。

下位モジュールを呼び出すには、下位モジュール名に続いてインスタンス名、ポート接続を記述します。

例のように、下位モジュールのポート名と上位モジュールの信号名を記述し、複数ポート同士はカンマ(,)で区切ります。

上位モジュールの信号は、更に括弧で囲みます。

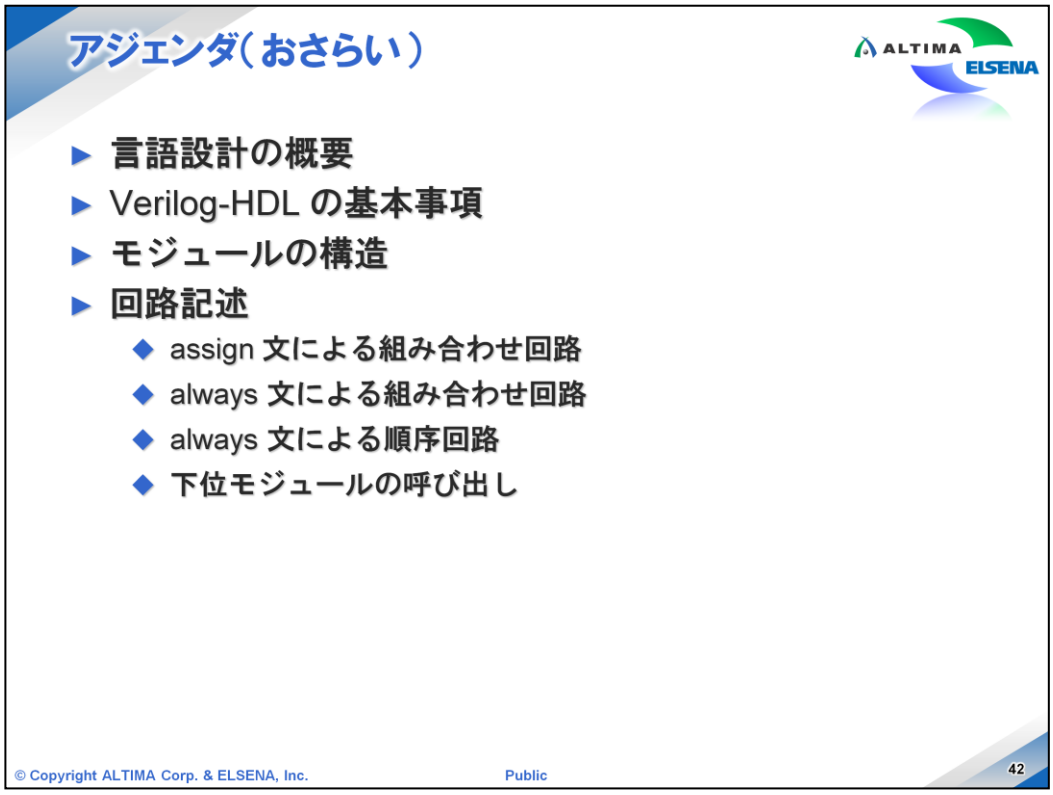
同一階層に同じインスタンス名のモジュールを置くことはできません。



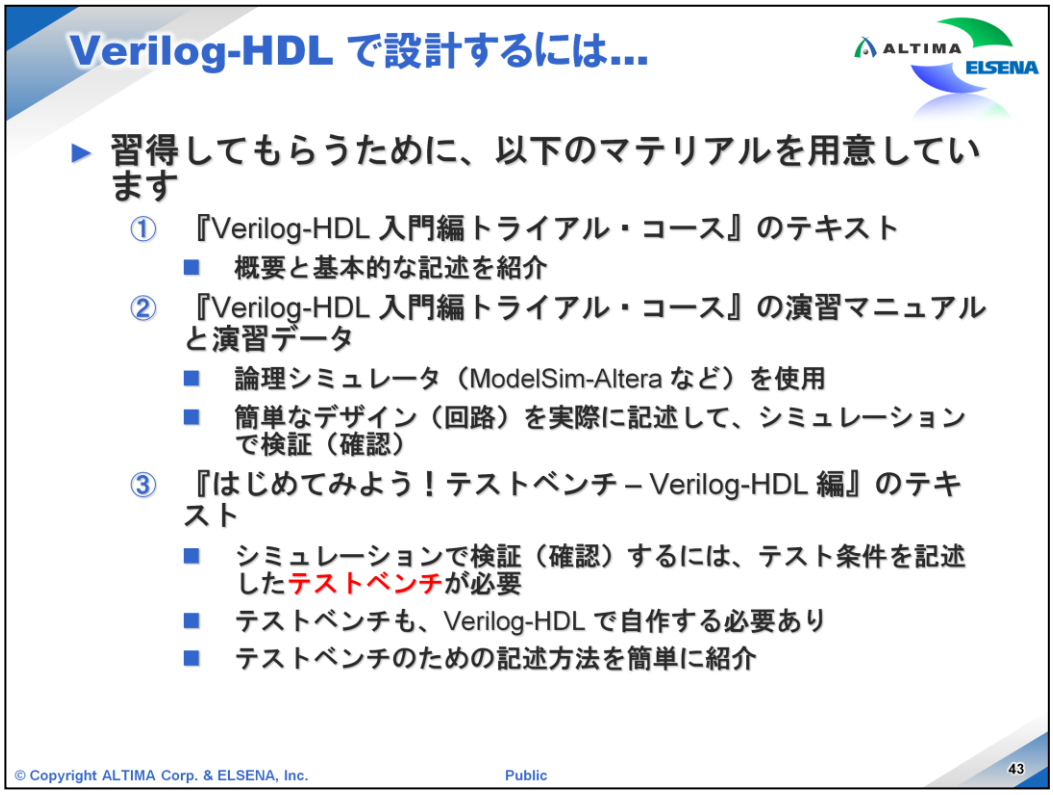
下位モジュール呼び出しの記述例を示します。

まず、黄色のブロックは、下位モジュールに相当します。
今まで説明してきた通り、ポート宣言や回路記述などが書かれています。

そして、青色の上位モジュールですが、まずはポート宣言やネット宣言が書かれています。
続いて回路記述部に、下位モジュールの呼び出し記述があります。
この例では下位に同じモジュールが 2個置かれており、別々のインスタンス名 U0 と U1 で定義されています。
1つ目の下位モジュール名とインスタンス名を書き、ポート接続を書きます。
続いて、2つ目以降の下位モジュールも同じように記述します。



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、Verilog-HDL 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『Verilog-HDL 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『Verilog-HDL 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。

Verilog-HDL 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 – Verilog-HDL 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、『Verilog-HDL 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。