



『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。

本コースについて



▶ 対象者

- ◆ Verilog-HDL による論理回路設計が初めての人

▶ 本ワークショップのコンセプト

- ◆ Verilog-HDL の概要を理解して、基本的な記述方法を習得しましょう！

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。

本ワークショップを受講すると、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 で設計するには...



▶ 習得してもらうために、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
 - 概要と基本的な記述を紹介
- 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
 - 論理シミュレータ（ModelSim®-Altera® など）を使用
 - 簡単なデザイン（回路）を実際に記述して、シミュレーションで検証（確認）
- ③ 『はじめてみよう！テストベンチ – Verilog-HDL 編』のテキスト
 - シミュレーションで検証（確認）するには、テスト条件を記述した**テストベンチ**が必要
 - テストベンチも、Verilog-HDL で自作する必要あり
 - テストベンチのための記述方法を簡単に紹介

© Copyright ALTIMA Corp. & ELSENA, Inc.

Public

3

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。

- ▶ 言語設計の概要
- ▶ Verilog-HDL の基本事項
- ▶ モジュールの構造
- ▶ 回路記述
 - ◆ 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
 - ◆ 下位モジュールの呼び出し

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、階層設計に必要な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 なのか？



- ▶ HDL
 - ◆ Hardware Description Language の略
 - ◆ 文字どおり、論理回路ハードウェアを記述するための言語
- ▶ 現在では、HDLでの設計手法が主流
 - ◆ IC の大規模化による従来の回路図による論理回路設計の限界
 - ◆ 大規模論理回路を短期間で設計する必要性
 - ◆ 論理合成ツールが実用レベルに達した
 - ◆ 設計資産の共有化
- ▶ HDL 設計のメリット
 - ◆ 半導体ベンダーにとらわれない設計が可能
 - ◆ 論理合成による設計期間の短縮
 - ◆ 設計資産の活用
- ▶ VHDL と Verilog-HDL について
 - ◆ 現在、主に使用されている HDL 言語
 - ◆ どちらの HDL も標準化されているため、言語仕様の優位性はなし

© Copyright ALTIMA Corp. & ELSENA, Inc.

Public

6

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。

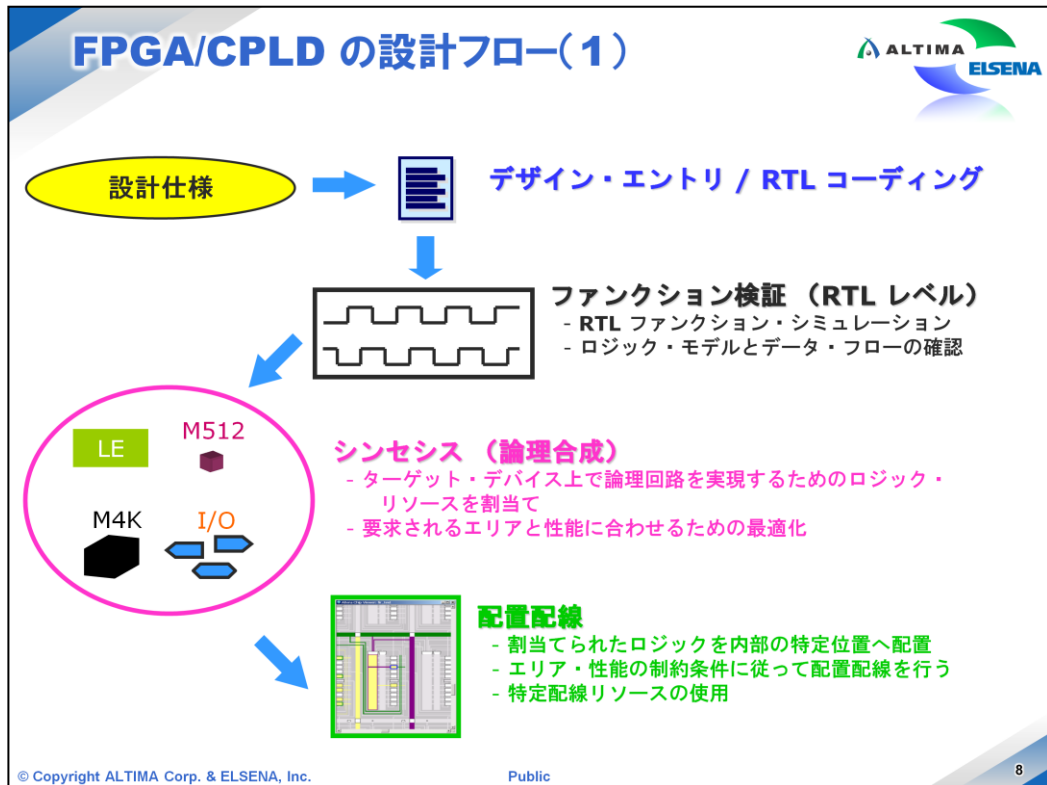
- ▶ HDL で記述された論理機能を実際のゲート回路に変換
- ▶ 変換の際に冗長な記述を最適化
- ▶ 論理合成による出力は、目的の ASIC や FPGA/CPLD 用のネットリスト
 - ◆ ネットリストとは、回路部品の接続関係をテキストで表現したもの
- ▶ 汎用論理合成ツール
 - ◆ Synopsys® 社の Synplify Pro®
 - ◆ Mentor Graphics® 社の Precision® Synthesis など
- ▶ アルテラの論理合成ツール
 - ◆ 独自の論理合成エンジンを Quartus® Prime 開発ソフトウェアに搭載
 - ◆ 他社の論理合成結果を読み込むことも可能（EDIF ネットリスト形式）

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。

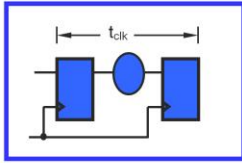
その後、内部遅延を含めない論理的なシミュレーションを行い、想定された動作をしているかどうかを確認します。

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。

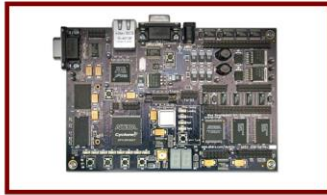
これらは、ツールが行ってくれます。

FPGA/CPLD 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。



次に、『Verilog-HDL の基本事項』の説明に移ります。

- ▶ ステートメント
 - ◆ ステートメントの終了には、セミコロン(;)を付ける
 - ◆ 複数に及ぶステートメントは、begin - end で囲む
- ▶ 改行とインデント
 - ◆ 1つのステートメントを複数の行に分けて書ける
 - 読みやすい
 - ◆ インデントするには、スペースかタブを使用
- ▶ コメント行
 - ◆ コメント行の先頭に // (ダブル・スラッシュ)
 - ◆ 複数行にまたがるコメントは /* と */ で囲む
- ▶ 大文字・小文字の区別
 - ◆ Verilog HDL では C 言語などと同様、大文字・小文字を区別

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。

識別子・予約語のルール



▶ 識別子（信号名やノード名、モジュール名）

- ◆ 英数字（a-z、A-Z、0-9）とアンダースコア（_）を使用
- ◆ 先頭文字は、アルファベットまたはアンダースコアでなければならない
- ◆ 数字で始めることは不可
- ◆ 1024文字以内
- ◆ 予約語は使用不可

▶ 予約語（主なもの）

always for output and force parameter assign forever
begin fork posedge task design case if casex tri
casez initial include inout input default define integer
reg wait else end module endcase endfunction negedge
while unsigned endmodule endtask or

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。

数値表現・文字列のルール



▶ 論理値

0	論理 0
1	論理 1
X または x	不定値
Z または z	ハイ・インピーダンス

▶ 数値表現

- ◆ ベクタの幅と基数（'b:2進、'o:8進、'd:10進、'h:16進）で指定
- ◆ 読み易いようにアンダースコアで区切ることが可能

数値定数	ビット幅	基数	2進数表現	10進数表現
15	32	10進	0...01111	15
8'h15	8	16進	00010101	21
6'o77	6	8進	111111	63
8'b0001_1111	8	2進	00011111	31
3'b01x	3	2進	01x	
4'bzzzz	4	2進	zzzz	

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。

演算子



- ▶ 算術演算
- ▶ ビット演算
- ▶ リダクション演算
- ▶ 論理演算
- ▶ 等号演算
- ▶ 関係演算
- ▶ シフト演算
- ▶ その他（条件演算、接続演算）

次に、演算子について細かく見ていきたいと思います。

算術演算子



▶ 2値の算術演算を行う

+	加算
-	減算
*	乗算
/	除算
%	剰余

0で割った結果は x (不定論理値)
となる

例)

ain = 3'd5, bin = 4'd10,
cin = 2'b01, din = 2'b0z

のとき

bin + cin = 4'd11

bin - cin = 4'd9

ain * bin = 6'd50

bin / ain = 2'd2

▶ バス入力はバス出力を得る

▶ 入力値が Z or X のとき、 結果は不定値

ain + din = 不定値

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。

ビット演算子



▶ 2値のビット演算を行う

~	not
&	and
	or
^	xor
^^	xnor

例)

ain = 3'b101, bin = 3'b110,
cin = 3'b01x

のとき

~ain = 3'b010

ain & bin = 3'b100

bin & cin = 3'b010

ain | bin = 3'b111

ain ^ bin = 3'b011

ain ^ ~bin = 3'b100

- ▶ オペランドの各ビットごとに処理される
- ▶ 出力はオペランドの最大サイズで得られる
- ▶ サイズが違う場合は Left- 拡張される

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。

リダクション演算子



- ▶ バス入力のそれぞれのビット同士の演算を行いシングルビット出力を得る

&	and
~&	nand
	or
~	nor
^	xor
^~ ~^	xnor

例)

ain = 5'b10101, bin = 4'b0011,
cin = 3'bz00, din = 3'bx01

のとき

&ain = 1'b0

&din = 1'b0

~&ain = 1'b1

|ain = 1'b1

|cin = 1'bx

~|ain = 1'b0

^ain = 1'b1

~^ain = 1'b0

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。

関係演算子・論理演算子



▶ 関係演算子：2値の比較を行う

==	等しい
!=	等しくなく
===	等しい(X、Zも含む)
!==	等しくない(X、Zも含む)
<	小さい
<=	小さいまたは等しい
>	大きい
>=	大きいまたは等しい

▶ 論理演算子：2値の論理演算を行う

&&	論理積
	論理和
!	論理否定

例)

```
always @(posedge clk) begin
  if ((a == b) && (c == d))
    q <= 4'b1010;
  else if ((a != b) && (c != d))
    q <= 4'b0101;
end
```

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。

▶ 値のビット・シフトを行う

<<	左シフト
>>	右シフト

例)

A = 4'b1010, Y は 4ビットの reg または wire のとき

- ① assign Y = A << 1 ⇒ Y = 4'b0100
- ② assign Y = A >> 2 ⇒ Y = 4'b0010

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。

条件演算子



▶ 条件付きで値の選択を行う

? [条件式] ? [真の場合] : [偽の場合]

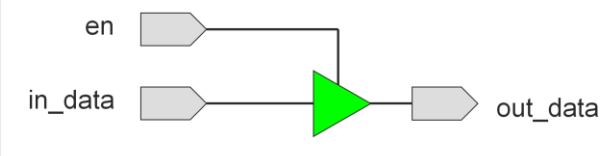
例)

```
assign out_data = (en == 1'b1) ? in_data : 4'bzzzz ;
```

en が 1 ならば、in_data を out_data に出力

en が 0 ならば、out_data が Hi-Z の状態

推論される回路



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。

▶ 信号の接続を行う

{ } ビットの結合、式の右辺にも使用可能

例)

A = 2'b00, B = 2'b10, M は 4ビット, N は 10ビットの reg または wire のとき

- ① $M = \{A, B\} \Rightarrow M = 4'b0010$
- ② $N = \{2\{A\}, 3\{B\}\} \Rightarrow N = 10'b0000101010$

最後に、接続演算子です。

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ビットを接続させています。

例②は、A を 2つ、B を 3つの計 10ビットに接続させています。



次に、『モジュールの構造』の説明に移ります。

module モジュール名 (ポート・リスト);

宣言部

ポート宣言

レジスタ宣言

ネット宣言

パラメータ宣言

回路記述部

- ・下位モジュールの呼び出し
- ・assign 文
- ・always 文 など

endmodule

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。

基本構造:各種宣言



▶ ポート宣言

```
input  clk, clr;           //入力
input  [7:0] data;         //入力バス信号
output result, busy;       //出力
inout  [15:0] d_bus;       //双方向バス信号
```

▶ ネット宣言

```
wire    temp1;
wire    [7:0] bus;         //バス信号
```

▶ レジスタ宣言

```
reg  ff1, ff2;             //1bit レジスタ
reg  [3:0] qout;           //4bit レジスタ
```

▶ パラメー宣言

```
parameter address = 32,
              width  = 16'h3F,
              act    = 2'b10;
```

宣言部について、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。



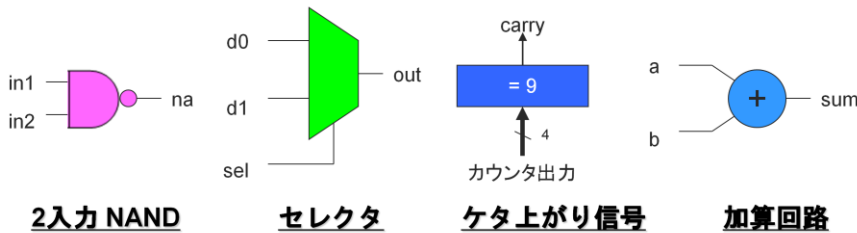
ここからは、実際の『回路記述』の説明に入ります。

assign 文による組み合わせ回路



- ▶ 論理式 1行 で記述できる組み合わせ回路は assign 文で記述

```
assign na = ~(in1 & in2);           //2入力 NAND
assign out = (sel == 1) ? d1 : d0;  //セレクタ
assign carry = (cnt10 == 4'h9);     //ケタ上がり信号
assign sum = a + b;                 //加算回路
```



© Copyright ALTIMA Corp. & ELSENA, Inc.

Public

26

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。

演習 1

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。

- ▶ 用途
 - ◆ 組み合わせ回路
 - ◆ 順序回路
- ▶ 代入される左辺はレジスタ型 (reg)
 - ◆ ネット型 (wire) は指定できない
- ▶ always 文内のステートメントは順次処理
 - ◆ ステートメントの順番がポイント
- ▶ 複数の always 文同士は並列処理
- ▶ always 文はネスティングできない
 - ◆ always 文の中に always 文を記述することは不可
- ▶ always 文の信号代入
 - ◆ ブロッキング代入
 - ◆ ノン・ブロッキング代入

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。

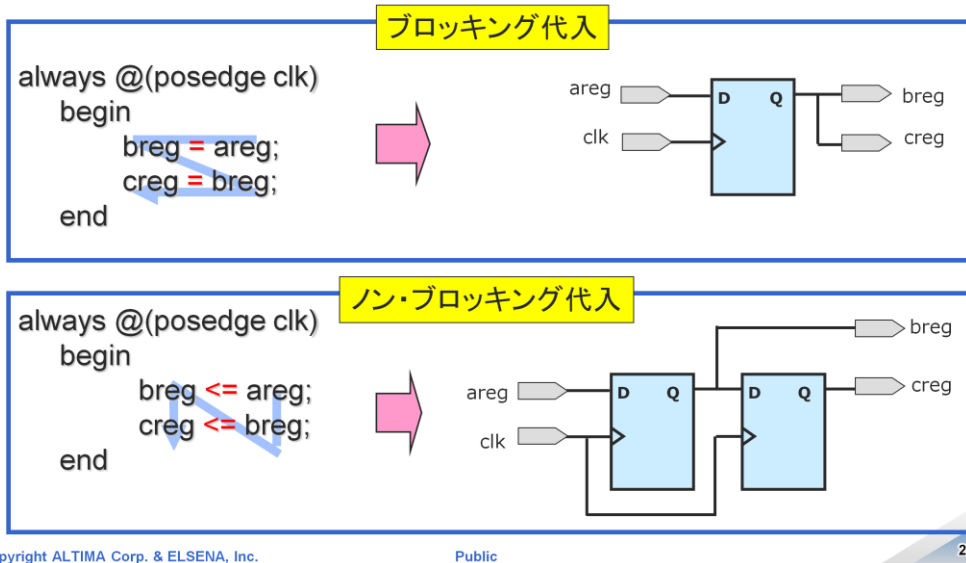
つまり、always 文の中に always 文を記述することはできません。

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



▶ always 文内での値の代入方法



© Copyright ALTIMA Corp. & ELSENA, Inc.

Public

29

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。

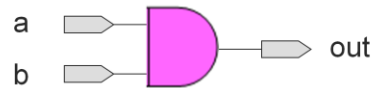
always 文による組み合わせ回路



▶ 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。

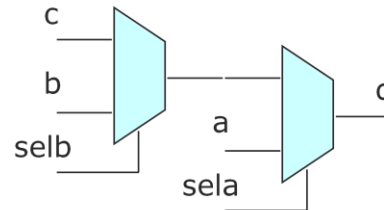
その後に、動作を記述します。

演習 2

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。

```
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。

演習 3

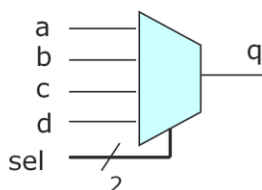
それでは、演習3を実施してください。
詳細は、演習マニュアルをご覧ください。

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



© Copyright ALTIMA Corp. & ELSENA, Inc.

Public

34

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。

演習 4

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。

always 文による順序回路



- ▶ フリップ・フロップ回路
- ▶ always 文の @ 以降に、クロック信号と非同期制御信号を記述
 - ◆ センシティブティ・リスト
- ▶ always @(センシティブティ・リスト)
 - ◆ 信号の立上り : posedge
 - ◆ 信号の立下り : negedge

例)

```
always @(posedge clock or negedge reset)
```

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ブティ・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。

クリア信号付きフリップ・フロップの記述



同期クリア付きフリップ・フロップ

```
module ff_sync_clr (d, clear, clock, q);  
  input  d, clear, clock;  
  output q;  
  
  reg    q;  
  
  always @(posedge clock)  
  begin  
    if (clear == 1'b0)  
      q <= 1'b0;  
    else  
      q <= d;  
    end  
  end  
endmodule
```

非同期クリア付きフリップ・フロップ

```
module ff_async_clr (d, aclear, clock, q);  
  input  d, aclear, clock;  
  output q;  
  
  reg    q;  
  
  always @(posedge clock or negedge aclear)  
  begin  
    if (aclear == 1'b0)  
      q <= 1'b0;  
    else  
      q <= d;  
    end  
  end  
endmodule
```

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、センシティブティ・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、センシティブティ・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 **negedge 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。

クロック・イネーブル付きフリップ・フロップ

```
module clk_enb (d, enable, clock, q) ;  
  input  d, enable, clock ;  
  output q ;  
  
  reg    q ;  
  
  always @( posedge clock )  
    if (enable == 1'b1)  
      q <= d ;  
  
endmodule
```

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、センシティブティ・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。

演習 5

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。

下位モジュールの呼び出し



- ▶ 既存のモジュールを呼び出し、上位階層へインプリメント
- ▶ 頻繁に用いるファンクションはコンポーネント化しておけば、何度でも呼び出せるから便利

下位モジュール名 インスタンス名 (ポート接続);

例)

```
comp1 comp1_ins (.a(A), .b(B), .y(Y));
```

a, b, y : 下位モジュールのポート名

A, B, Y : 上位モジュールの信号名

※ 別々の名前でももちろん可

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、そうす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



```
module add4 (ci, a, b, sum, co); // 下位モジュール
```

```
input  [3:0]    a, b;
```

```
input          ci;
```

```
output [3:0]    sum;
```

```
output co;
```

```
assign {co, sum} = ci + a + b;
```

```
endmodule
```

```
module add8 (CI, A, B, SUM, CO); // 上位モジュール
```

```
input [7:0]    A, B;
```

```
input          CI;
```

```
output [7:0]    SUM;
```

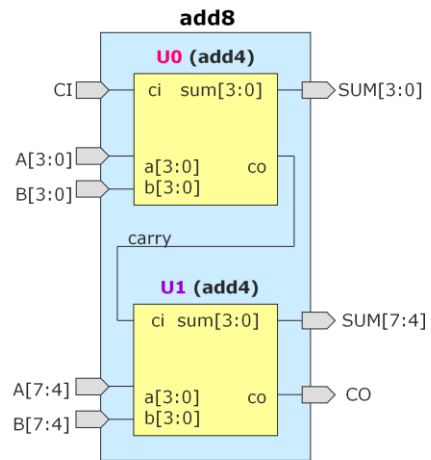
```
output CO;
```

```
wire  carry;
```

```
add4 U0 (.ci(CI), .a(A[3:0]), .b(B[3:0]), .sum(SUM[3:0]), .co(carry));
```

```
add4 U1 (.ci(carry), .a(A[7:4]), .b(B[7:4]), .sum(SUM[7:4]), .co(CO));
```

```
endmodule
```



© Copyright ALTIMA Corp. & ELSENA, Inc.

Public

41

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。

アジェンダ(おさらい)



- ▶ 言語設計の概要
- ▶ Verilog-HDL の基本事項
- ▶ モジュールの構造
- ▶ 回路記述
 - ◆ 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
 - ◆ 下位モジュールの呼び出し

ここまで、このアジェンダに沿って、説明してきました。

Verilog-HDL で設計するには...



▶ 習得してもらうために、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
 - 概要と基本的な記述を紹介
- 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
 - 論理シミュレータ（ModelSim-Altera など）を使用
 - 簡単なデザイン（回路）を実際に記述して、シミュレーションで検証（確認）
- ③ 『はじめてみよう！テストベンチ – Verilog-HDL 編』のテキスト
 - シミュレーションで検証（確認）するには、テスト条件を記述した**テストベンチ**が必要
 - テストベンチも、Verilog-HDL で自作する必要あり
 - テストベンチのための記述方法を簡単に紹介

© Copyright ALTIMA Corp. & ELSENA, Inc.

Public

43

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。



ALTIMA



ELSENA

**ワークショップはこれで終了です。
お疲れ様でした。**

弊社より資料を入手されましたお客様におかれましては、下記の使用上の注意を一読いただいた上でご使用ください。

1. 本テキストは非売品です。許可無く転売することや無断複製することを禁じます。
2. 本テキストは予告なく変更することがあります。
3. 本テキストの作成には万全を期していますが、万一ご不明な点や誤り、記載漏れなどお気づきの点がありましたら、弊社までご一報いただければ幸いです。
4. 本テキストで取り扱っている回路、技術、プログラムに関して運用した結果の影響については、責任を負いかねますのであらかじめご了承ください。
5. 本テキストは製品を利用する際の補助的な資料です。製品をご使用になる場合は、英語版のメーカー資料もあわせてご利用ください。

Verilog-HDL_Trial_Text_r1

Public

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。