東京大学工学部機械情報工学科 メカトロニクス設計演習

FPGA・VHDL入門

演習資料

2022年9月28日版

森本 雄矢 y-morimo@hybrid.t.u-tokyo.ac.jp
聶 銘昊 nie@hybrid.t.u-tokyo.ac.jp

目次

第1章　FPGAとは

大規模集積回路 LSI (Large-Scale Integration) は **Fig. 1.1** のように分類できる．この演習で扱う FPGA (Field Programmable Gate Array) はその名の通り，その場でプログラムできるゲートアレイである．コンピュータ上でデザインした論理回路を書き込むことで，任意の LSI として利用することができる．FPGA は ASIC のプロトタイプとして動作を確認するのに適している．また ASIC に比べて開発期間が短く済み，初期コストが低いため，早期に製品を市場に出したい場合や少量生産の場合には量産にも用いられる．

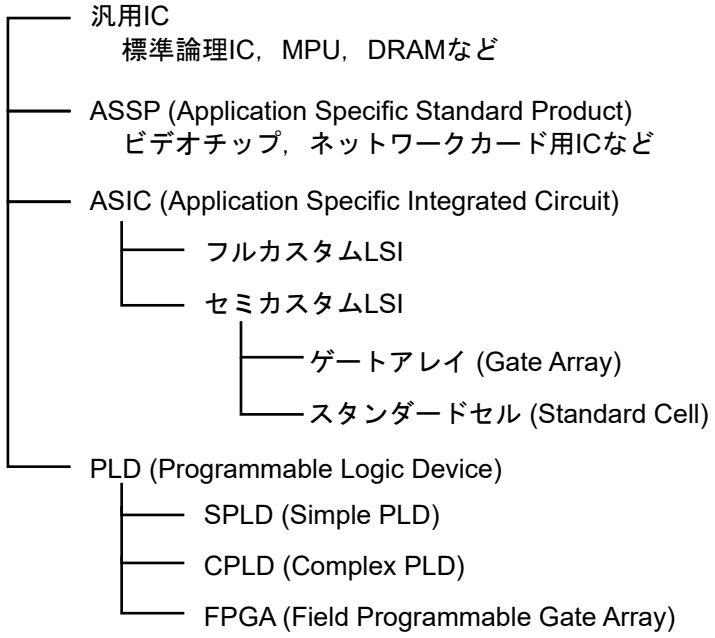


Fig. 1.1: LSI の分類.

FPGA の内部には，ファンクションジェネレータ，レジスタ，マルチプレクサなどからなる基本セル（CLB: Configurable logic block）があらかじめ相互に配線されており，スイッチマトリクス（PSM: Programmable switch matrix）をプログラムすることで接続を変更して，任意の回路を構成することができるようになっている．FPGA における回路設計とは，どの基本セルにどのような論理回路を割り当てるかを決定し，求める動作を実現するようにスイッチマトリクスをプログラムすることを意味する．

本演習で使用する FPGA ボードには，Xilinx（ザイリンクス）社の Spartan-6 シリーズの XC6SLX16 という FPGA が載っている．このチップには揮発性のメモリがあり，電源を切るとダウンロードしたプログラムが消えてしまう. ボードには不揮発性のメモリも実装されており, そちらに書き込むこともできる（演習では扱わない）.

第 2 章　回路設計の流れ

回路設計の大まかな流れを説明する (**Fig. 2.1**)．まず，回路の仕様を決める．それに従い，ハードウェア記述言語（HDL）により回路を記述する．次に，論理合成により論理ゲート回路を生成する．このゲート回路が論理的に正しく動作するかを検証するために，論理シミュレーションを行う．その後，配置配線を行う．これはゲート回路を実際の FPGA のどこに配置し，互いに配線するかを決めるものである．これにより配線長が決定するので，配置配線した回路に対してタイミングシミュレーションを行い，要求するタイミングを満たしているか確認する．ここまでは全てコンピュータ上で行う．最後にコンピュータと FPGA ボードを接続し，配置配線した回路を FPGA にダウンロード (コンフィギュレーションともいう) して終了である．また，設計した回路の FPGA 上での動作を検証するために，FPGA からの信号をコンピュータに取り込み，デバッグする方法もある．設計者は HDL で回路を記述することに集中することができ，論理合成以降はパラメータを入力する程度で自動的に処理が行われる．

2.1　ハードウェア記述言語（HDL）による回路記述

HDL(Hardware Description Language) とは，回路のハードウェア動作を記述する言語である．HDL を用いた設計は，回路図入力による設計よりもより抽象度の高いレベルで設計することができるため，より大規模な回路設計に対しても難しい論理式を考える必要がないという利点がある．また，設計の変更が容易であり，記述内容が理解しやすく，切り貼りもできることから，設計の共有や再利用がしやすい．

HDL は，C 言語などのプログラミング言語の構文に似ているが，入力信号の変化に従って並行的に動作するような並列処理や，論理値，遅延などのハードウェア特有のパラメータを記述することが可能である．現在広く普

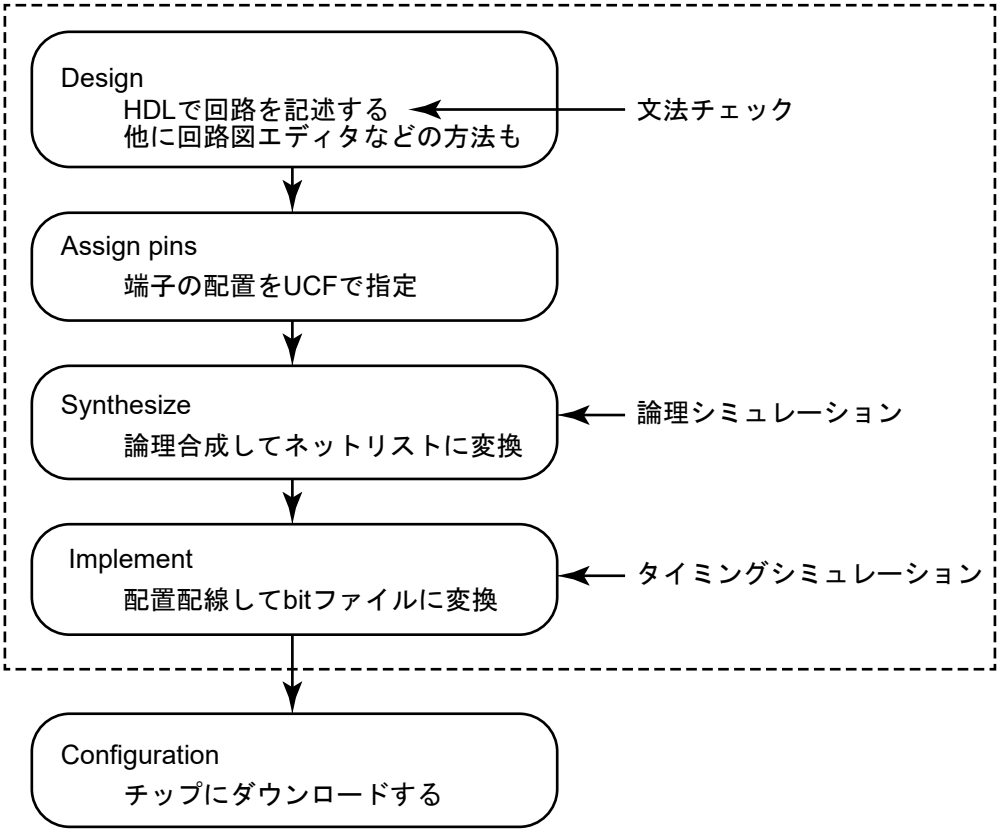


Fig. 2.1: 回路設計の流れ.

及している HDL に，VHDL (VHSIC (Very High Speed IC) HDL) と Verilog-HDL がある．本演習では VHDL を用いて設計を行う．

HDL で記述された回路を等価な論理回路表現（ネットリスト: 回路中の素子や接続状態を，文字列を使って記述したもの）へと変換していく．その際，設計者の与える設計制約条件や FPGA デバイスのテクノロジ条件により，以下に挙げるような処理が行われる．

ビヘイビアシミュレーション

プログラムの論理的な記述が設計どおりになっていることを確認するために，テストベンチ（入力を記述した VHDL）を用いて，合成した回路の動作をシミュレータ上で確認する．

2.2 論理合成

リソースの割り付け（Resource Allocation）

HDL で定義された演算子（+，-，&など）を判別して，ライブラリに登録されているリソース（加算器，乗算器など）を自動的に割り付ける．同じ演算を実行する加算器にも，リップル・キャリーやキャリー・ルックアヘッドなどの性能の異なるいくつかのアーキテクチャが存在する．論理合成では，それらのアーキテクチャの選択が，設計制約条件にしたがって自動的に行われる．たとえば，高速動作が必要な場合にはキャリー・ルックアヘッドが選択され，高速動作が必要なく回路面積を小さくする場合にはゲート数を減らすためにリップル・キャリーが選択される．このように適切なアーキテクチャが自動的に選択されるので，設計時に固有のアーキテクチャを指定する必要がない．

リソースの共有化（Resource Sharing）

HDL で記述された回路を解析し，複数の演算子が同時に使用されないときに，それらの演算子に対して一つのリソースを割り当てることである．例えば，**Fig. 2.2** の回路では記述中の 2 つの加算演算は if 文によって分岐されているために，同時に実行されることはない．この記述をそのまま実現した場合には，2 つの加算器の出力をマルチプレクサによって選択する回路が合成されるが，加算器の前にマルチプレクサを配置して，演算する入力データを選択することで加算器を共有することができる．しかし，マルチプレクサにおけるデータを選択する信号の遅延が大きい場合には，要求されるタイミング条件によってはこのようなリソースの共有が行われない．リソースの共有は，演算が同時に実行されないと判断され，さらに共有化してもタイミング条件を満足している場合に行われる．

レジスタの推定（Register Inference）

クロック・サイクル間に渡ってどの値を保持すべきかを決め，値を保持するために必要なラッチやレジスタなどの順序素子を呼び出す．順序素子に対して，適切なクロックや，非同期や同期のリセット/セット信号が接続される．HDL 記述の際に不充分な分岐をもつ条件式を記述すると，分岐中に当てはまる動作が存在していない場合には値を保持するとして不必要なラッチが生成される場合があるので，注意が必要である．

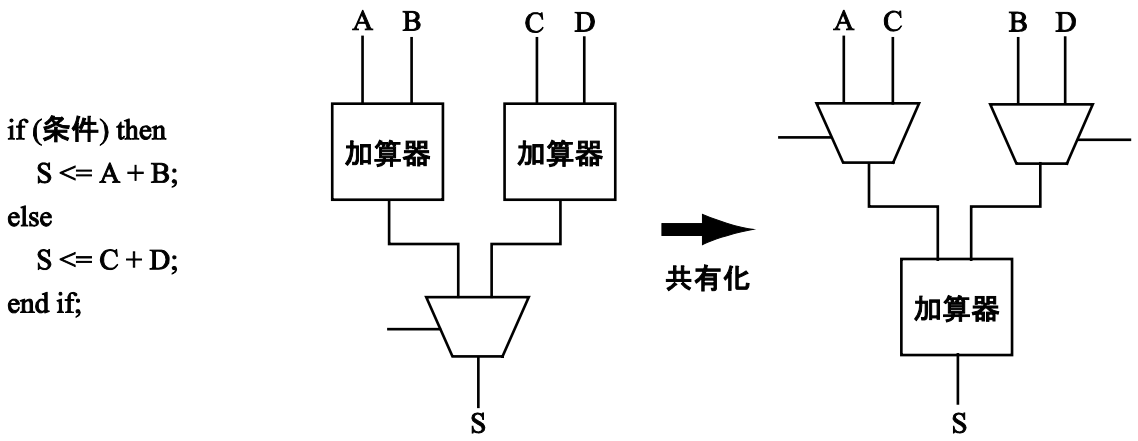


Fig. 2.2: リソースの共有化の例.

構造化（Structuring）と平坦化（Flattering）

構造化とは複数の論理回路の間で共有化可能な論理を見つけ出すことである．この処理によって論理回路を小さくすることができるが，選び出した共有項のファンアウトが増加することから，遅延値は増加する．逆に，平坦化は共有化された論理を展開することである．

冗長論理の削除（Redundancy Removal）

論理式内の冗長な論理を見つけ出して削除することである．冗長論理は回路面積を増やし，不必要なファンアウトによって回路性能を悪化させる．**Fig. 2.3** に冗長論理の削除例を示す.

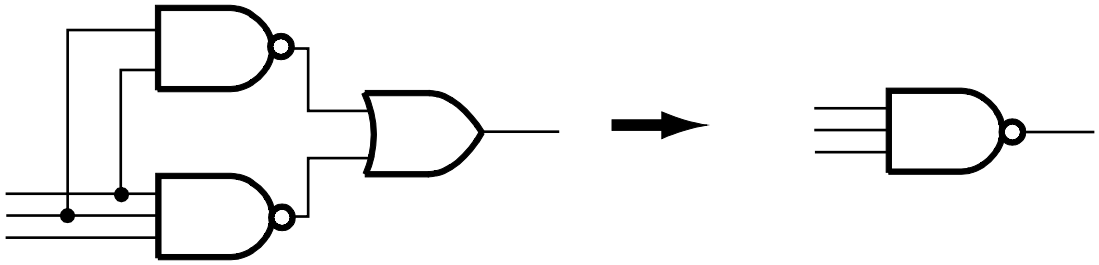


Fig. 2.3: 冗長論理の削除例.

テクノロジ・マッピング（Technology Mapping）

対象とする FPGA デバイスのテクノロジ条件に合致するように論理構造を変換し，ライブラリに登録された実際の素子を割り当てることである．その際，回路の速度や面積，消費電力などの設計者の与える制約条件を満たすために，最良の素子の組み合わせが割り当てられる．回路を高速にすることと，ゲート数を最小にして回路面積を小さくすることとは，トレードオフの関係にある．高速な回路が要求された場合には，入力から出力までの論理段数をできるだけ減らそうとするため，回路面積は増大する．逆に回路面積を抑えようとすると，論理の共通化を計り論理段数が増えるために回路の動作は遅くなる．

2.3 インプリメンテーション

インプリメンテーションでは，変換（Translate）→ マップ（Map）→ 配置配線（Place & Route）を順に行う．

配置配線（Place & Route）

論理合成されたゲート回路に従って，実際に FPGA にプログラミングするのに必要な情報に変換するために，論理回路を FPGA 内のセルに自動配置し，スイッチマトリックスの自動配線を行う．配置配線の中で重要な処理の一つにクロック・ツリーの構築がある．複数のレジスタを接続した回路では，配線によってクロックの到達時間にばらつき（クロック・スキュー（Clock Skew））が生じる．このばらつきにより，セットアップ・タイムやホールド・タイムなどのタイミング条件が満たされなくなる恐れがある．配置配線では，クロックの入力ボートから実際にレジスタに達するまでの遅延時間が均一化されるように，レジスタから等距離のところにバッファを配置するなどの処理が行われる．

タイミングシミュレーション

配線長が決まると，信号の遅延時間を満たしているか計算することができる．これをタイミングシミュレーションという．これにより仕様を満たしていることを確認する．

2.4 プログラムファイルの生成とコンフィギュレーション

FPGA にダウンロード可能な形式であるビットストリームファイルを生成し，そのビットストリームファイルを FPGA に書き込む（コンフィギュレーション）．本演習ではコンフィギュレーションに Xilinx 社の VIVADO を用いる．

コンフィギュレーションには，ボードの製造元である Digilent 社から提供されている Adept というソフトウェアを用いることもできる．演習で使用するボードには，複数のコンフィギュレーション方法がある．演習では書き込み時間の短い，FPGA の SRAM に直接書き込む方法を用いる．しかしこの方法で書き込んだプログラムは電源を OFF にすると消えてしまう．プログラムを保持するためには，ボード上に実装された不揮発メモリに書き込むとよい．この方法でコンフィギュレーションするときには，Adept を使うのが便利である．どのメモリに書き込むかは J8 で設定する．ジャンパーの設定を変えるときには「必ず電源を OFF にする」こと．

第 3 章　新規設計の流れ

本章では VHDL 言語による回路の新規設計を題材として，統合開発環境の操作手順に慣れてもらう．なお，この演習で使用する ISE Design Suite のバージョンは 14.7 である．

3.1　回路の仕様を決める

回路をプログラミングするにあたり，まずは回路の仕様として少なくとも回路の目的と入出力について決める．

例として，2 つのスイッチを同時に ON にすると 1 つの LED が点灯する機能を実装する．入力はボードに実装されているスライドスイッチ SW0 と SW1 とし，出力は LD0 とする．つまり，**Fig. 3.1** に示すような，2 入力 1 出力の AND 回路である．図において，入力が 2 ビットの信号 SW で，出力が 1 ビットの信号 LD であることを表す．

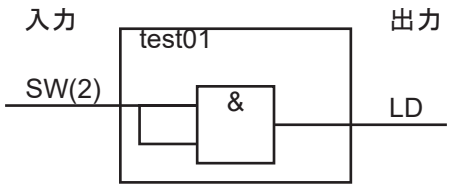
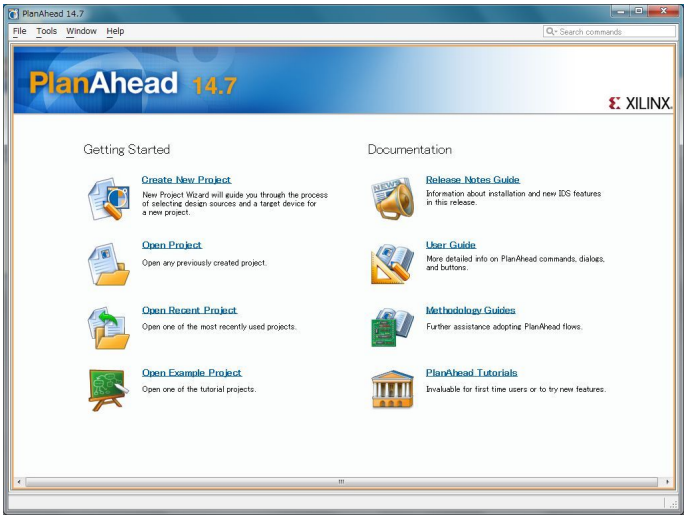


Fig. 3.1: 2 入力 1 出力 AND 回路．

3.2　プロジェクトの作成

「PlanAhead」の起動

Windows 画面のデスクトップにある PlanAhead のリンクをダブルクリックする．初回起動時はとても時間がかかる．

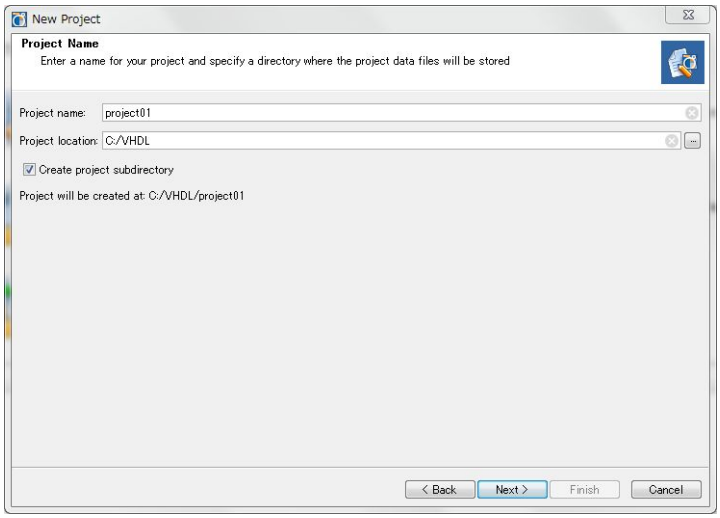


演習用のフォルダの作成

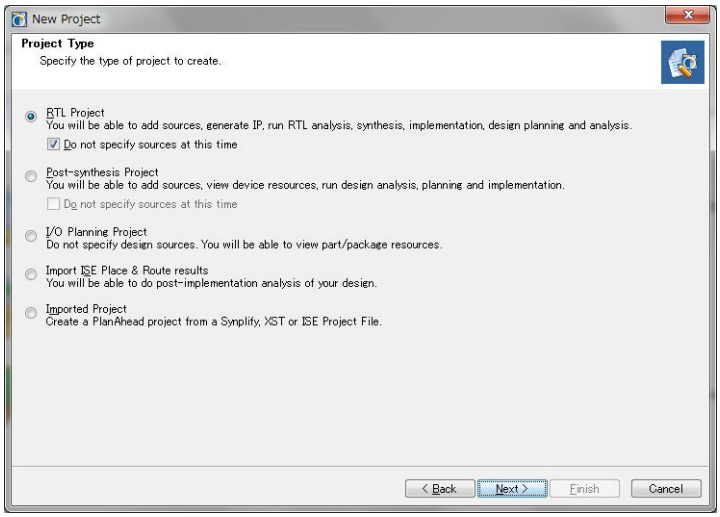
エクスプローラで C ドライブ直下にフォルダ「VHDL」を作成する．この資料では「C:¥VHDL」にプログラムファイルがあることを想定している．

新しいプロジェクトの作成

PlanAhead が起動したら，画面の左上のにある「Create new project」をクリックする．
出てきたウィンドウの「Next」をクリック.
「Name」にプロジェクト名を入力する（例では「project01」としている）．Project Location は,「C:/VHDL」とする.「Create project subdirectory」のチェックを入れておくと，自動でサブディレクトリが作られる．フォルダ名に日本語が入るとエラーが出るので注意すること.「Next」を選択.



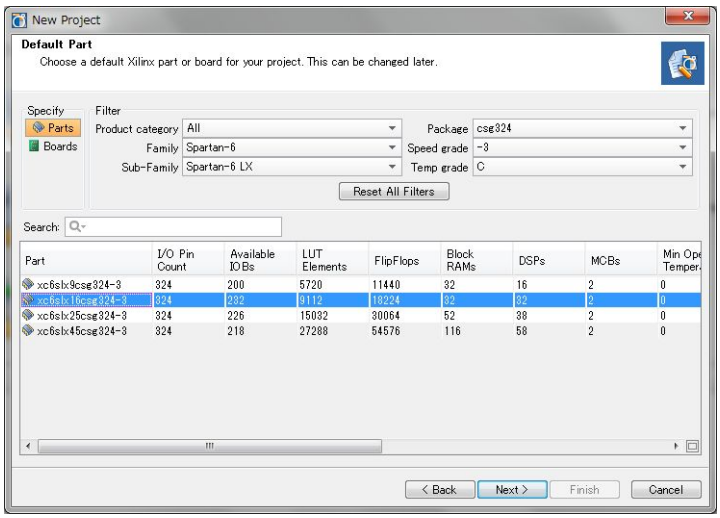
Project Type は「RTL Project」を選択する.「Do not specify sources at this time」にチェックを入れて,「Next」.



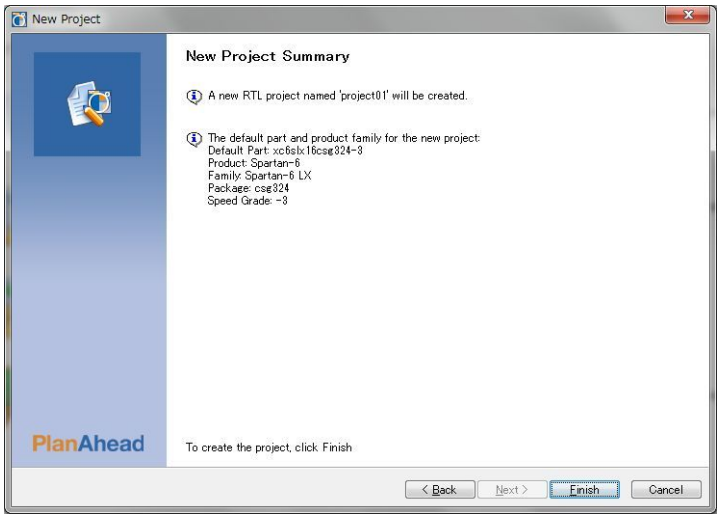
パーツ選択

Default Part で使用する FPGA を選択する．Filter で下記を選んでから，画面下で「xc6slx16csg324-3」を選択し，「Next」をクリック.「lx16」を選んでいることをよく確認すること．

Family	Spartan-6
Sub-Family	Spartan-6 LX
Package	csg324
Speed grade	-3

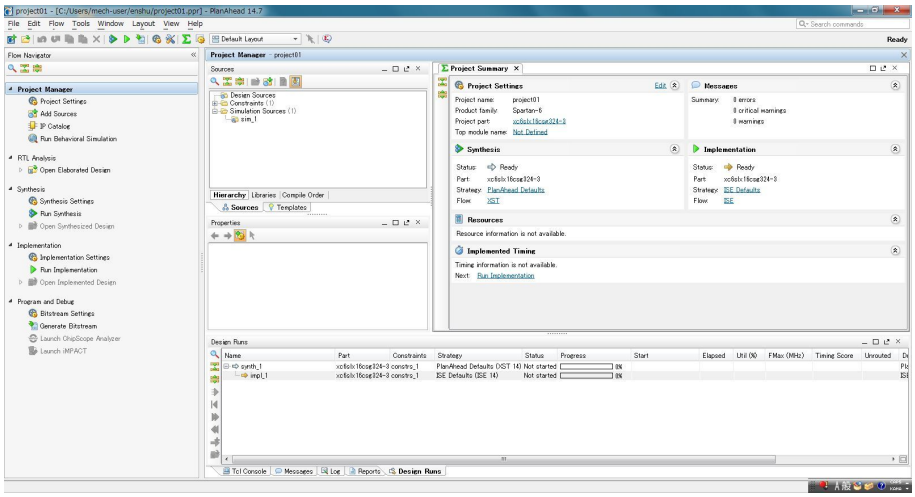


設定を確認して,「Finish」



統合開発環境の画面

PlanAhead の画面が表示される．基本的な流れとして，画面左の「Flow Navigator」に沿って進めていく．



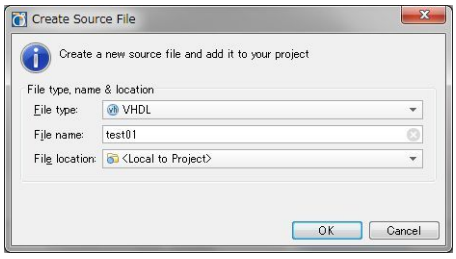
3.3 VHDL ファイルの作成

ファイルの新規作成

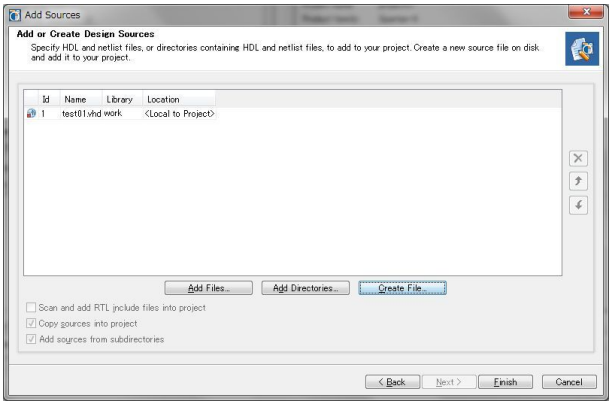
Flow Navigator の Project Manager の「Add source」をクリックし,「Add or Create Design Sources」を選択する．「Next」を選択.

次の画面で「Create File」をクリック.

「File type」を「VHDL」にして,「File name」に「test01」を入力.「OK」を選択.

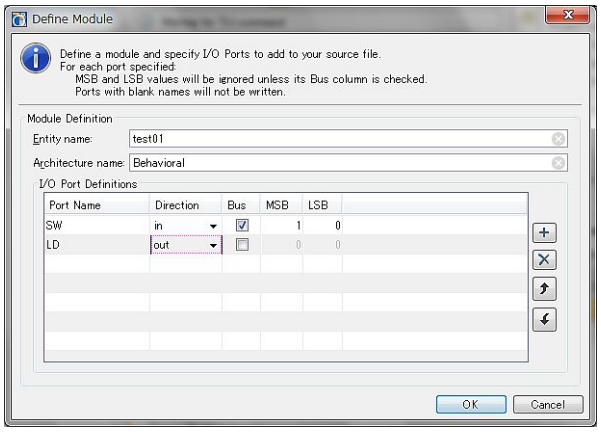


ファイル名などを確認して「Finish」.



入出力の設定

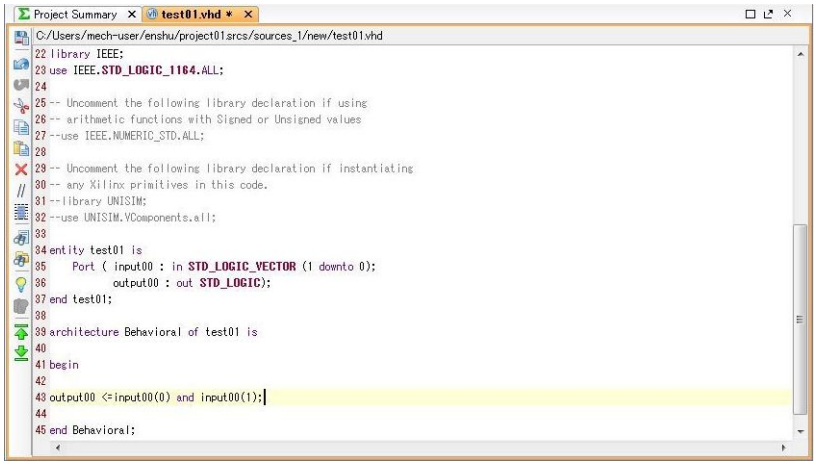
「Define Module」の画面で入出力の設定をする．この例の場合，2 つのスイッチ入力を 2 ビットの vector（SW）に入力し，1 ビットの出力データ LD を LED に出力する．vector 型にするときは，Bus にチェックを入れて，MSB（Most significant bit: 最上位ビット）と LSB（Least significant bit: 最下位ビット）を設定する．入力を終えたら「OK」を選択．これで test01.vhd が作成される．



Port Name	Direction	Bus	MSB	LSB
SW	in	check	1	0
LD	out			

VHDL の記述

画面上部左寄りの Hierarchy で test01.vhd をダブルクリックすると，画面右に次のような画面があらわれることを確認する．このウィンドウの VHDL プログラムを書き換えて，回路を記述する．



VHDL ファイルが生成された直後には，コメントを除くと **Table. 3.1** のとおりになっている．

Table. 3.1: 新規プロジェクトで作成されるひな形.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity test01 is
    Port ( SW : in STD_LOGIC_VECTOR (1 downto 0);
           LD : out STD_LOGIC);
end test01;

architecture Behavioral of test01 is

begin

end Behavioral;
```

architecture の記述

architecture に，**Table. 3.2** のように回路を記述する．新たに記入するのは，LD で始まる 1 行だけであることに注意してほしい．書き換えたら，忘れずにセーブしておくこと（Ctrl-s）．

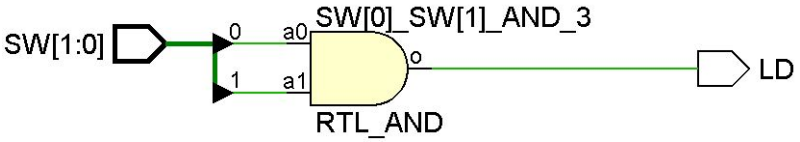
Table. 3.2: test01.vhd の architecture の記述例.

```
architecture Behavioral of test01 is

begin
    LD <= SW(0) and SW(1);
end Behavioral;
```

回路の確認（オプション）

Flow Navigator → RTL Analysis の Open Elaborated Design をクリックする．さらに，RTL Schematic をクリックすると，回路図が表示される．SW[1:0] というバスが AND に入力されて，LD が出力されている．



3.4 論理合成・インプリメンテーション・プログラミング

論理合成

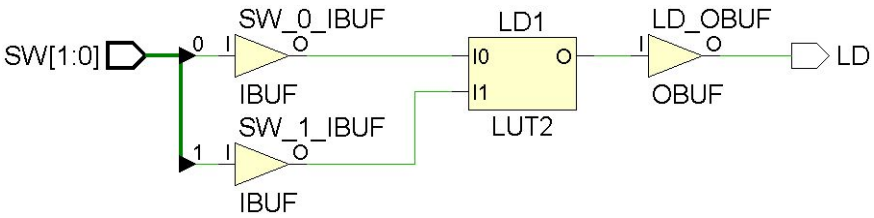
Flow Navigator → Synthesis の「Run Synthesis」をクリックする．論理合成には少し時間がかかる．コンパイル中は右上に処理中の工程が表示される．文法エラーがあると，画面下側の Console の「message」ウィンドウにエラーメッセージの抜粋が、「Log」ウィンドウにエラーメッセージの詳細が表示されるので，それを参考にデバッグする．

エラーがないときは「Synthesis Completed」のウィンドウが表示されるので,「Open Synthesis Design」を選択して「OK」．Elaborated Design を Close してよいか聞かれた場合は,「Yes」．

論理合成後の回路の確認（オプション）

Flow Navigator → Synthesis → Synthesized Design の Schmatic をクリックすると，回路図が表示される．入出力にバッファが挿入され，AND 回路は LUT（look up table）で構成されている．

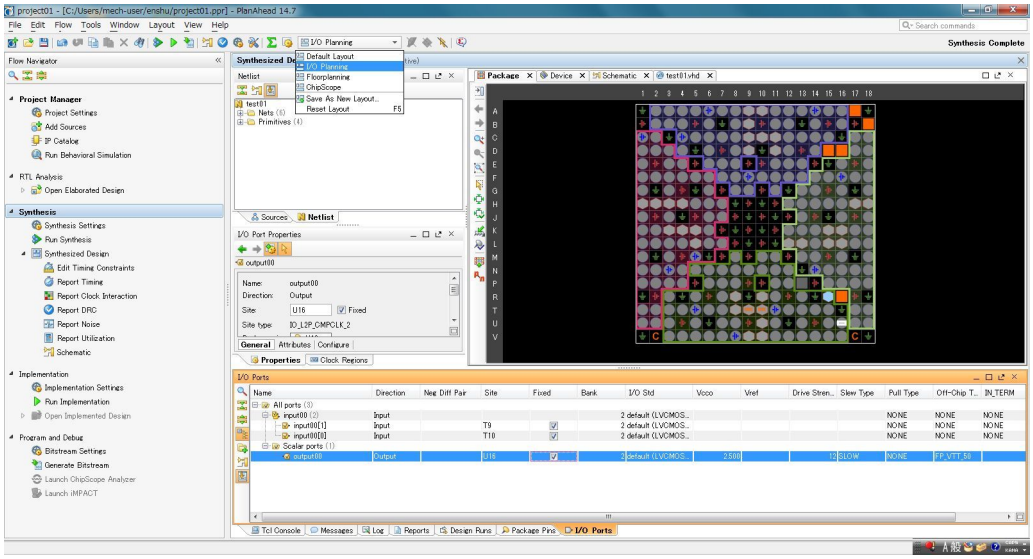
GUI の LD1 をクリックして選択してから，画面中央左寄りの Instance Properties で，タブ「Truth Table」を選択すると，AND の論理になっていることが確認できる．



ピンアサイン

画面上の「Default Layout」となっているところをクリックし,「I/O Planning」を選択．画面下の I/O Ports で，SW[1]，SW[0] 及び LD の信号の Site を設定する．Fixed のチェックが自動的に入ったことを確認する. Ctrl+s または，メニューバーの File → Save constraints で保存する．ファイル名は test01 としておく．これで test01.ucf が作成される．

Port Name	Site	実体
SW[1]	T9	SW1
SW[0]	T10	SW0
LD	U16	LD0

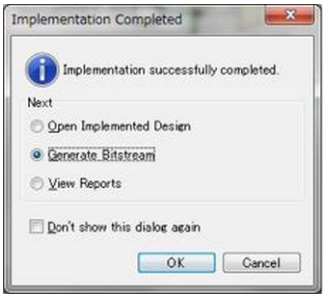


インプリメンテーション

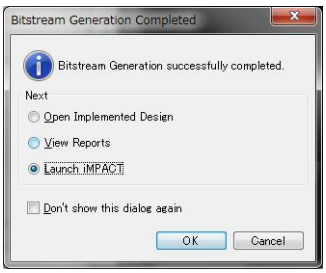
Flow Navigator → Implementation の「Run Implementation」をクリック.
しばらくすると Implementation Completed の画面が出る．

プログラムファイルの生成

Implementation Completed の画面で，Generate Bitstream を選んで「OK」. もしくは，Flow Navigator → Program and Debug の「Generate Bitstream」をクリック．インプリメンテーションの手順を飛ばした場合は「No Implementation Results Available」の確認画面が出るので「Yes」を選択.



しばらくすると Bitsteam Generation Completed の画面が出る．これで FPGA に書き込むための bit ファイルが生成される．



3.5 コンフィギュレーションと動作確認

FPGA ボードの接続

付属の USB ケーブルで FPGA ボードをノート PC に接続する．FPGA ボードの左上の「USB PROG」コネクタを使用すること．USB を接続したら，コネクタ近くの電源スイッチを入れる．

コンフィギュレーション

VIVADO Lab Edition を開き，Quick Start 内の「Create Project」をクリックする.「Project name」に「project_1-1」を入力し、「OK」をクリックする．画面上の「Open target」をクリックし、「Auto Connect」を選択する．画面左上の Hardware ウィンドウ内で，xc6slx16_0 が認識されていることを確認する．

画面上の「Program device」をクリックし、「Bitstream file」として、C:/VHDL/project_1/project_1.runs/impl_1 内にある「test01.bit」を選択，「Program」を押す.

動作確認

ボード上の SW0，SW1 を切換えて，LD0 の動作を確認する. 設計通りの AND 回路ができたか？

3.6 プロジェクトの再開

保存したプロジェクトを再開する際は、PlanAhead を起動した後に、画面左上二段目にある「Open Project」をクリックする。対象のフォルダ内にある ppr ファイルをクリックする。保存した段階でプロジェクト編集が再開可能となる。

第 4 章　VHDL の文法

4.1　エンティティとアーキテクチャ

VHDL では，回路の入出力インタフェースとなるポートをエンティティで定義し，具体的な処理をアーキテクチャで記述する．基本的な HDL 記述の例を **Fig. 4.1** に示す．これは VHDL で AND 回路を記述した例である．エンティティには，外部の回路と接続するためのポートが定義されているだけで，処理については何も書かれていない．回路の中身を記述するのがアーキテクチャである．エンティティとアーキテクチャが分離していることにより，ひとつのエンティティに対して複数のアーキテクチャを記述して異なる実装をすることもできる（この演習では扱わない）．

アーキテクチャの `C <= A and B` は，A と B の AND をとったものを信号 C に割り当てるという意味である．

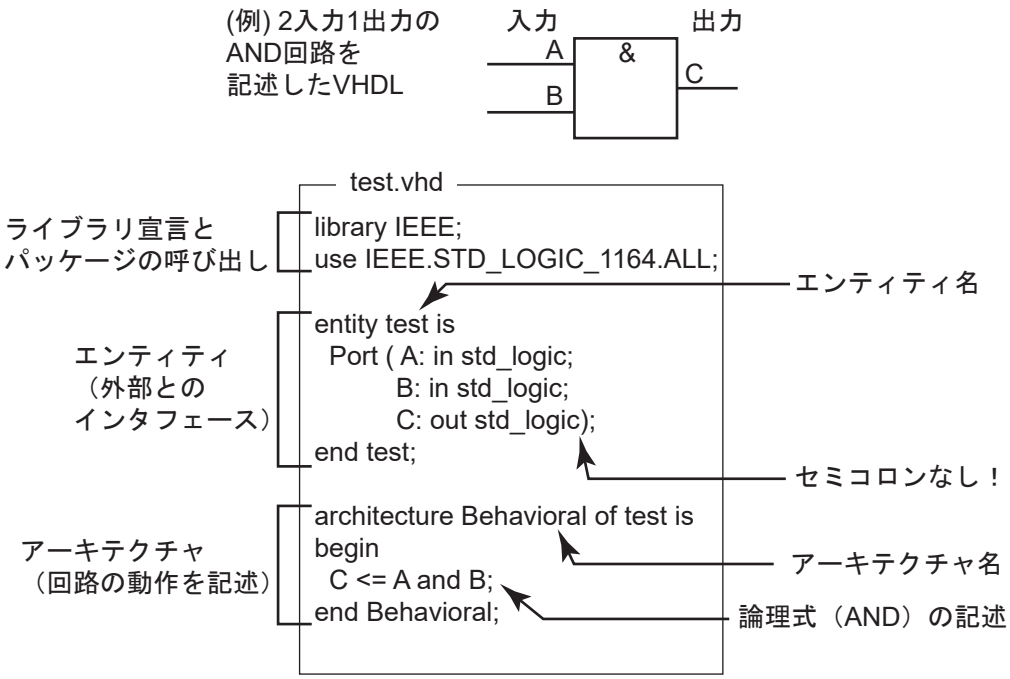


Fig. 4.1: HDL 記述の基本.

VHDL で使用することのできるエンティティ，アーキテクチャ，ポート，信号などの名前には，以下のような規則がある．

- 使用できる文字は，英字，数字，_（アンダースコア）である．
- 最初の文字は英字でなければならない．
- アンダースコアを続けて用いてはならない．また，名前の最後にアンダースコアを用いてはならない．
- 大文字と小文字の区別はない．ただし，’（シングルクォート）あるいは ”（ダブルクォート）で囲まれた文字は文字定数であり，大文字と小文字の区別をする．
- スペース，改行，タブを自由に使えるので，読みやすいように書くとよい．ただし，全角スペースを用いると文法エラーとなる．
- 文の終わりは;（セミコロン）である．

- コメントは，--（連続するハイフンふたつ）で始める．改行されるまで有効.

4.2 データの型

VHDL で使うことのできる型（のうち代表的なもの）を **Table. 4.1** に示す．入出力の信号の型は必ず，std_logic，またはその配列タイプの std_logic_vector にする．これらを使うにはライブラリ宣言とパッケージ呼び出しのために，必ず文頭に以下の 2 行の記述が必要である．

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

後述の演算子のうち，関係演算子，加算演算子を用いる場合は，signed，unsigned という型を使う．このとき，パッケージ NUMERIC_STD を使う必要がある．

```
use IEEE.NUMERIC_STD.ALL;
```

Table. 4.1 に示したものの他に，character, string, real といった型もある．

Table. 4.1: VHDL のデータタイプ（**本演習では std_logic，std_logic_vector，unsigned で充分である**）.

データタイプ	説明	使用例
std_logic	9 種の論理値（U，X，0，1，Z，W，L，H，-）	
std_logic_vector	std_logic の配列タイプ.	
signed	符号付きのベクトル	
unsigned	符号なしのベクトル	
integer	整数	
natural	自然数	
boolean	true（真）と false（偽）の 2 つの値をもつ．等価（=）や比較（<）などの関係演算で，演算結果が真であれば true，偽であれば false を戻す.	
bit	2 値の論理値 0 または 1 をもつ．Bit タイプへの値の代入は 2 進，8 進，16 進数で行うことができ，ビット基数として B（2 進数），O（8 進数），X（16 進数）を用いて指定する．2 進数の場合には省略できる.	B”11010” ”11010”（B を省略） O”32” X”1A”

VHDL はデータの型のチェックが厳密である．比較や加算をするときに signed 型や unsigned 型を使うのは，符号の有り無しを明示的に指定させるためである．signed の信号を std_logic_vector として取り出すときは，型変換（キャスト）しなければならない（**Table. 4.2**）.

論理値は 1 と 0 だけではない (**Table. 4.3**)．実際の LSI では電源を入れた際に，内部信号の値が 1 か 0 のどちらに決まるかは不確定である．シミュレーションでは，このように 1 か 0 かどちらに決まるかわからないような不確定な状態を不定値 X で表す．また，不定値 X は，回路内で 1 と 0 が同一信号上で競合した際に，どちらの論理値になるかわからない状態も表す．ゲート出力信号のドライブ強度をストレングス（Strength）という．強ストレングスと弱ストレングスが同一信号上で競合した場合には，強ストレングスの論理値が優先され，同一ストレングスで異なる論理値の値が競合した場合には不定値 X となる．複数のドライバをもつバス信号では，1 つだけのドライバから信号がドライブされ，他のドライバは全てハイインピーダンス Z でなければならない．シミュレーション実行時には信号は初期値 U から始まるため，初期化されていない回路部分を確認することができる．ドントケア (-) は論理合成時に使用する論理値である．出力の論理値が 1 でも 0 でも構わない場合にドントケアを用いれば，論理合成でゲート回路を最適化する際に回路性能が良くなる方の論理値を用いる．

Table. 4.2: 型変換（キャスト）の記述例.

```
signal A_VEC : STD_LOGIC_VECTOR (3 downto 0);
signal B_UNS : UNSIGNED (3 downto 0);
signal C_INT : INTEGER;
signal D_UNS : UNSIGNED (4 downto 0);
...
A_VEC <= std_logic_vector (B_UNS);
C_INT <= to_integer(unsigned(A_VEC)); --std_logic_vector は直接 integer に変換できない
A_VEC <= std_logic_vector(to_unsigned(C_INT, B_UNS'length));
  --B_UNS'length: B_UNS のビット数を返す
D_UNS <=  resize(unsigned(B_UNS),4);
```

Table. 4.3: std_logic の論理値（**本演習では 0 と 1 のみでよい**）.

Std_logic の論理値	論理値の意味
0，1，X	強ストレングスの論理値 0，論理値 1，不定値を示す.
L，H，W	弱ストレングスの論理値 0，論理値 1，不定値を示す.
Z	信号がドライブされていないハイインピーダンス状態を示す.
U	初期化されていない（Uninitialize）論理値を示す.
-	ドントケア（don't care）を示す.

4.3 演算子

演算子のうち，この演習で使用する (可能性がある) ものの一覧を **Table. 4.4** に示す．加算演算子と連結演算子の優先順位が等しいことに注意しなければならない.

繰り返しになるが，関係演算子または加算演算子を使うためにはパッケージ NUMERIC_STD を使う．このとき，演算子の左と右は同一のデータタイプでなければならないが，signed と integer，または unsigned と natural は混在可能である.

Table. 4.4: 主な演算子．下に行くほど優先順位が高い．**本演習の課題では論理演算子，等号，加算，減算，除算，剰余，連接を使う．**

論理演算子	and or nand nor xor	論理積 論理和 論理積の否定 論理和の否定 排他的論理和	std_logic，std_logic_vector，bit，bit_vector， boolean タイプで使用可能である．式の右辺と左辺および代入される値は，同一のデータタイプで同じ長さでなければならない．
関係演算子	= /= < > <= >=	等号 不等号 より小さい より大きい 以下 以上	結果は boolean で返される（つまり true または false）．std_logic_vector には使用不可．
四則演算 連接演算子 剰余演算子	+ - * / & mod	加算 減算 乗算 **除算** 連接 **剰余**	integer, signed, unsigned に使用可能． std_logic_vector にも使用したいなら, std_logic_signed や std_logic_unsigned パッケージが必要. 連接は配列をつなげるためのもの．例えば，4 ビットと 4 ビットを連結して 8 ビットにするなど． a modulo n, 略して a mod n とも表記される. a を b で除算し、余りを取得. 計算例: 32 mod 7 = 4
	not	否定	

4.4 信号

モデルの内部で各ゲートに接続される信号は，アーキテクチャ内で signal により宣言する．

信号割り当てにおける注意として，エンティティで in と指定したポートには出力できない（すなわち，信号 A が入力ならば，A <= B とは書けない）．逆に，エンティティで out と指定したポートを入力として使用することはできない（すなわち，信号 C が出力ならば，A <= C とは書けない）．

信号に初期値を持たせたいときは、:=（セミコロン イコール）を用いる．

```
signal A: STD_LOGIC;
signal B: STD_LOGIC := '0';
```

定数は constant で宣言する．

```
constant data: STD_LOGIC_VECTOR (3 downto 0) := "0101";
```

長いベクトルに信号を割り当てるときに,「残りは全て 0」といった指定をしたくなることがある．下記のように記述すると，ビット数を間違えることがなくなり便利である．

（例）ベクトル A の要素を全て'0' にしたいとき ··· `A <= (others => '0');`

（例）(7 downto 0) のベクトル A の 5 ビット目を'1' に，他を全て'0' にしたいとき ···

`A <= (5 => '1', others => '0');`．これは，`A <= "00100000";` に等しい．

Table. 4.5: 関係演算，加算演算は signed または unsigned 型に対して使う．

```
signal COUNT : UNSIGNED (7 downto 0);
...
if (COUNT < 100) then
COUNT <= COUNT + 1;
else
COUNT <= (others => '0');
end if;
```

ここまでの知識を使うと，2 ビット加算器は **Table. 4.6** のように記述される．std_logic_vector 型は加算ができないので，unsigned 型に明示的にキャストしていることに注意してほしい．

Table. 4.6: 2 ビット加算器の記述例.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;                              -- 算術演算するため

entity adder2 is
    Port (A, B: in STD_LOGIC_VECTOR (1 downto 0);  -- 2 ビットの入力の宣言
          S: out STD_LOGIC_VECTOR (1 downto 0);    -- 2 ビットの出力の宣言
          CO: out STD_LOGIC);                      -- 桁上がり信号の論理値の宣言
end adder2;

architecture Behavioral of adder2 is
signal TMP_I : STD_LOGIC_VECTOR (2 downto 0);      -- 信号の宣言
begin
    TMP_I <= std_logic_vector(unsigned('0' & A) + unsigned('0' & B));
                                                   -- &はビットの結合を表す
    CO <= TMP_I(2);                                -- 結果は信号代入文<= により
    S <= TMP_I(1 downto 0);                        -- TMP に代入される
end Behavioral;
```

4.5 同時処理文

アーキテクチャで記述した信号割り当ては，チップ内部の回路を配線していくものと考えるとわかりやすい．論理合成の結果は，割り当てを記述する順序によらず同じ結果となる．これは，MPU で実行される通常のプログラムが逐次的に進んでいくのとは対照的である．

Table. 4.7 は，testsignal 回路という 2 入力（IN1，IN2）1 出力（OUT1）の AND 回路の記述である．入力 IN1 と IN2 の AND をとり，OUT1 に結果を出力する．左右の記述を比較すると，`begin` から `end` までの記述の順番が異なるが，この部分は同時処理文であるため，結果は同じになる．

Table. 4.7: 二種類の AND 回路の記述. 結果は等しい.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity testsignal is
    Port (IN1: in STD_LOGIC;
          IN2: in STD_LOGIC;
          OUT1: out STD_LOGIC);
end testsignal;

architecture Behavioral of testsignal is
signal A, B : STD_LOGIC;
begin
    A <= IN1;
    B <= IN2;
    OUT1 <= A and B;
end Behavioral;
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity testsignal is
    Port (IN1: in STD_LOGIC;
          IN2: in STD_LOGIC;
          OUT1: out STD_LOGIC);
end testsignal;

architecture Behavioral of testsignal is
signal A, B : STD_LOGIC;
begin
    OUT1 <= A and B;
    A <= IN1;
    B <= IN2;
end Behavioral;
```

4.6 順序処理文と変数

記述の順序のとおりに処理するためには，if 文や case 文などの順次処理文を用いる．順次処理文は，**Table. 4.8** に示すようにプロセスの中に記述する．

process( ) のカッコの中には，プロセス文を実行する引き金となる信号を全て指定する．この信号の羅列をセンシティビティリスト（Sensitivity list）と呼ぶ．プロセスは，センシティビティリストに定義された信号が変化したときだけ評価される．プロセスでは，順次処理文が上から順番に評価される．

変数（variable）は，プロセスの中で値を保持するために使用され，process( ) と begin の間に宣言する．変数への初期値の代入は:=によって行う．変数の値は次に代入されるまで保持される（ラッチ）．変数の代入は右辺の計算と左辺への代入が同時に行われる．これは，信号代入文において，全ての信号代入文の右辺の処理が終了した後に，左辺への代入が行われるのと異なる．また，変数の値をプロセス文の外で利用するには，変数の値を信号に代入する．

Table. 4.8: 典型的なプロセス文の書き方.

```
architecture アーキテクチャ名 of エンティティ名 is

signal 信号の宣言;

begin

    process (センシティビティリスト)

    variable 変数の宣言;

    begin

        if 文，case 文，for-loop 文，while-loop 文などの順次処理文;

        信号 <= 変数;

    end process;

end アーキテクチャ名;
```

Table. 4.9: if 文の書式.

if(条件) then
  順次処理文;
end if;
	if(条件) then
	  順次処理文;
	else
	  順次処理文;
	end if;
		if(条件) then
		  順次処理文;
		elsif(条件) then
		  順次処理文;
		elsif(条件) then
		  順次処理文;
		else
		  順次処理文;
		end if;
			if(条件) then
			  順次処理文;
			elsif(条件) then
			  if(条件) then
			  if(条件) then
			  順次処理文;
			  else
			  順次処理文;
			  end if;
			  end if;
			end if;

【注意】全ての条件分岐について記述しなければならない．else は文法上は省略可能だが，省略してはならない.			

Table. 4.10: if 文の記述例（比較器）.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity comparator is
  Port ( A, B : in STD_LOGIC_VECTOR(3 downto 0);
         Y : out STD_LOGIC);
end comparator;

architecture Behavioral of comparator is
begin
  process(A,B)          -- センシティビティリストの定義
  begin
    if(unsigned(A) > unsigned(B)) then
      Y <= '1';
    else
      Y <= '0';
    end if;
  end process;
end Behavioral;
```

Table. 4.11: case 文の書式.

```
case 式 is
  when 値 => 順次処理文;
  when 値 => 順次処理文;
  when 値 | 値 | 値 …. => 順次処理文;
  when 値 to 値 => 順次処理文;
  when others => 順次処理文;
end case;
```

【注意】値が重複してはならず，式の取り得る全ての値に対応する処理を記述しなければならない．when others は最後に 1 回のみ，必ず記述する．

Table. 4.12: case 文の記述例（エンコーダ）.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity encoder is
  Port ( INPUT : in STD_LOGIC_VECTOR(7 downto 0);
         Y : out STD_LOGIC_VECTOR(2 downto 0));
end encoder;
architecture Behavioral of encoder is
begin
  process (INPUT)
  begin
    case INPUT is
      when "01111111" => Y <= "000";
      when "10111111" => Y <= "001";
      when "11011111" => Y <= "010";
      when "11101111" => Y <= "011";
      when "11110111" => Y <= "100";
      when "11111011" => Y <= "101";
      when "11111101" => Y <= "110";
      when "11111110" => Y <= "111";
      when others     => Y <= "---";          -- don't care 出力
    end case;
  end process;
end Behavioral;
```

Table. 4.13: for-loop 文の書式.

```
for ループ変数 in 整数 to 整数 loop
  順次処理文;
end loop;
```

【注意】ループ変数は変数宣言する必要がない．ループ変数は整数の変数であるが，データタイプをもたない．したがって，ループ変数の値を信号や変数に直接代入することはできない．ループ変数は配列のビット数の指定や，関係演算子を用いた数値の比較に使用する．

Table. 4.14: for-loop 文の記述例（パリティチェック）.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity parity_check is
    Port ( A: in STD_LOGIC_VECTOR(7 downto 0);
           Y : out STD_LOGIC);
end parity_check;

architecture Behavioral of parity_check is
begin
    process (A)
    variable TMP: STD_LOGIC;       -- 変数宣言
    begin
        TMP := '0';                -- 変数への値の割り当て
        for I in 0 to 7 loop       -- ループ変数 I は宣言なしに使用可能
            TMP := TMP xor A(I);
        end loop;
        Y <= TMP;                  -- 変数の値を信号に代入して，プロセス文の外に出力
    end process;
end Behavioral;
```

4.7 組み合わせ回路と順序回路（クロックの利用）

デジタル回路には大きく分けて，組み合わせ回路と順序回路の 2 種類がある．組み合わせ回路とは，現在の入力の組み合わせだけで出力が定まる回路である．代表的な例としては，インバータ (NOT)，AND，NAND，NOR，XOR，バッファ，マルチプレクサなどがある．

これに対して順序回路は，“データ取得時の” 入力と過去の状態から出力が決まる回路である．一般的な順序回路ではデータ取得のタイミングを制御信号によって決めるため，順序回路には入力・出力に加えて，クロック信号などの制御信号が必要となる．代表的な例としては，ラッチ，フリップフロップ，カウンタ，レジスタなどがある．

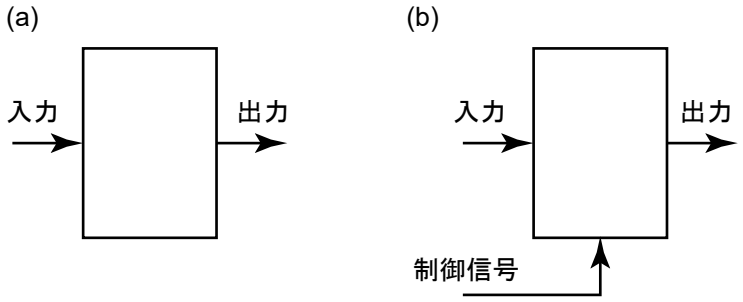


Fig. 4.2: (a) 組み合わせ回路と (b) 順序回路.

順序回路の制御信号としてはクロックが使われることが多い．交互に High と Low になる矩形の信号の立ち上がりまたは立ち下がりに同期して process の中を評価する．**Table. 4.15** にクロックのエッジの検出方法を示す．結局，立ち上がり検出は rising_edge，立ち下がり検出は falling_edge を使うとよい．

Table. 4.15: クロックの立ち上がりの利用.

確実な立ち上がり検出）

```
process (CLK、RESET)
begin
    if (RESET = '1') then
        Q <= '0';
    elsif rising_edge(CLK) then -- 確実
        Q <= D;
    end if;
end process;
```

悪い例）

```
process (CLK, RESET)
begin
    if (RESET = '1') then
        Q <= '0';
    elsif (CLK = '1') then -- 立ち上がり?
        Q <= D;
    end if;
end process
```

悪い例）

```
process (CLK)
begin
    wait until (CLK = '1'); -- 1 まで待つ
    Q <= D;
end process;
```

悪い例）

```
process (CLK)
begin
    if (CLK = '1') then   -- 1 だったら
        Q <= D;
    end if;
end process;
```

4.8 デザインの階層化とコンポーネント文

機能ごとに回路を分割（コンポーネント化）し，必要な個所でその回路を呼び出して使うことができる．これにより，複数の回路を階層化して設計することができる．設計が効率化できる．

例として NAND 回路を用いた AND 回路の実装を挙げて説明する（**Fig. 4.3**）．一般に，全ての論理回路は NAND を用いて記述することができる．AND 回路は，A and B = (A nand B) nand (A nand B) と表現できる．ひとつの AND 回路に 3 つの NAND 回路が含まれているので，NAND コンポーネントを再利用することを考える．AND 回路の中で 3 つの NAND 回路を下位階層として呼び出して使用するには，**Table. 4.16** のように記述すればよい．

コンポーネントを用いるときには，**Fig. 4.3** のようなブロック図を描くと，コンポーネントの入出力内部信号名，コンポーネント内の信号名が整理できるので便利である．

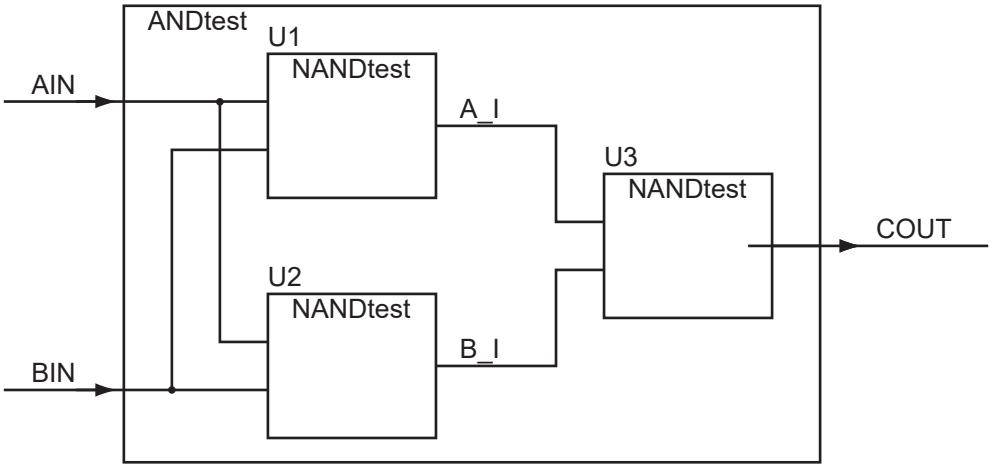


Fig. 4.3: NAND コンポーネントを用いた AND 回路. 予約語との混乱を避けるため，トップとコンポーネントの名称の最後に test をつけた．

ポートマッピングは，コンポーネントのポートと，呼び出し元の信号との結線を指定する．**Table. 4.16** において，コンポーネント U1 は，NANDtest の INPUT1 ポートに AIN，INPUT2 ポートに BIN，OUTPUT ポートに A_I を接続することを明示的に記述している．一方で，コンポーネント U2 は，コンポーネントのポート名（INPUT1, INPUT2，OUTPUT）を明示せずに，component で宣言した順番に接続する信号配線の名前を書いている．ポートマッピングの書き方は U1 と U2 のどちらでもかまわない．コンポーネントと呼び出し元のプログラムとで，ポート名が同じでも問題ない．ただし，ラベル名はインスタンスされたコンポーネントの固有名詞なので，コンポーネント名とラベル名は被らないようにすること．

前に開発した VHDL をコンポーネントとして使うためには，まずプロジェクトにファイルを読み込む．Flow Navigator → Project Manager の Add source をクリックし,「Add or Create Design Sources」を選択する．出てきたダイアログで，Add Files をクリックし，ファイルを選択する.「Copy sources into project」にチェックが入っていることを確認して，Finish．VHDL でコンポーネントを記述すると，Hierarchy が自動で更新される．

トップの VHDL を変更したいときは，Hierarchy で該当のファイルを右クリックして,「Set as Top」を選ぶ．この状態で論理合成を行えばコンポーネントを利用することができる．

Table. 4.16: ANDtest.vhd における NANDtest コンポーネントの呼び出し方.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ANDtest is
    Port ( AIN : in STD_LOGIC;
           BIN : in STD_LOGIC;
           COUT : out STD_LOGIC);
end ANDtest;

architecture Behavioral of ANDtest is

component NANDtest
    Port ( INPUT1: in STD_LOGIC;
           INPUT2: in STD_LOGIC;
           OUTPUT : out STD_LOGIC);
end component;
signal A_I : STD_LOGIC;
signal B_I : STD_LOGIC;

begin
    U1: NANDtest port map (INPUT1 => AIN, INPUT2 => BIN, OUTPUT => A_I); -- 1
    U2: NANDtest port map (AIN, BIN, B_I);                               -- 2
    U3: NANDtest port map (A_I, B_I, COUT);
end Behavioral;
```

第5章　シミュレーション

FPGA のプログラムのデバッグをするときにはある時刻においてそれぞれの信号がどの状態にあるかを知りたい．しかしながら，内部信号はその名の通りチップの内部にしかないため，プローブをあてて計測することはできない．

FPGA のシミュレーションには 2 種類ある．ビヘイビアシミュレーションとタイミングシミュレーションである．

ビヘイビアシミュレーションとは，回路の論理を確認するためのものである．

タイミングシミュレーションとは，配置配線が終わった後のシミュレーションのことで，配線長に応じた遅延の影響などを確認することができる．特に規模の大きな回路を作るときには意図せずに配線が長くなっていることも考えられるため，タイミングシミュレーションをしておく必要がある．

5.1　シミュレーションの方法

Xilinx 社から提供されているシミュレーションツール ISim を用いて，シミュレーションの方法を説明する．

回路のシミュレーションを実行するためには入力信号が必要である．これを記述したものをテストベンチと呼ぶ．

テストベンチファイルの作成

PlanAhead の画面左の Flow Navigator で Project Manager をクリックする．Add Sources をクリックする．出てきたウィンドウで,「Add or Create Simulation Sources」を選択して「Next」.「Create File...」をクリック．File type が VHDL であることを確認し，File name に「test01_tb」と入力して「OK」.「Finish」をクリック.

入出力の設定

モジュール設定のウィンドウが出る．テストベンチには入出力が不要なので，そのまま「OK」．生成されたファイルは entity のポートの指定で Syntax Error が出る．

Hierarchy の，Simulation Sources → sim_1 → Syntax_Error Files を開くと，生成された「test01_tb.vhd」がある．(Syntax error が起きると、 Syntax_Error Files にファイルが移動することに注意) ダブルクリックして開く．

まず，文法エラーを解消するために，ポートの指定を修正しておく（**Table. 5.1**）．修正後にセーブすると，Hierarchy が自動的に更新される．

Table. 5.1: ポートの指定を修正．左：修正前（生成された状態）．右：修正後.

`entity test01_tb is`
`    Port ( );`
`end test01_tb;`
	`entity test01_tb is`
	
	`end test01_tb;`

テストベンチファイルの例（クロックなし）

テストベンチの architecture の全体像を示す.
テストの対象である test01 は，コンポーネントとして呼ぶ.
test01 の入出力は同じ名前の内部信号として定義しておく．入力の信号は初期値を'0' としておく.

```
architecture Behavioral of test01_tb is

component test01
    Port ( SW : in STD_LOGIC_VECTOR (1 downto 0);
           LD : out STD_LOGIC);
end component;

signal SW : STD_LOGIC_VECTOR (1 downto 0) := (others => '0');

signal LD : STD_LOGIC;

begin
    -- Instantiate the Unit Under Test (UUT)
    uut: test01 port map (SW, LD);

    -- Stimulus process
    stim_process: process
    begin
        -- hold reset state for 100 ns.
        wait for 100 ns;

        -- insert stimulus here
        wait for 200 ns;    SW(0) <= '1';
        wait for 200 ns;    SW(1) <= '1';
        wait for 200 ns;    SW(0) <= '0';
        wait for 200 ns;    SW(1) <= '0';

        -- always end with wait
        wait;
    end process;
end Behavioral;
```

テストベンチファイルの例（クロックあり）

クロックを使う場合には，テストベンチの中にクロックの切り替わりを記述する．課題 6 で作成するクロック生成回路のシミュレーションのためのテストベンチの例を下記に示す．演習で使用するボードで使用できるクロックは 100 MHz である．周波数が 100 MHz のとき，周期は 10ns（ナノ秒）である．

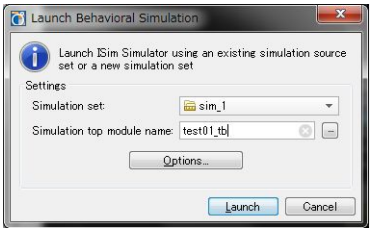
```
architecture Behavioral of test01_tb is
component test01
    Port ( CLK_SRC : in STD_LOGIC;
           CLK_OUT : out STD_LOGIC);
end component;
signal CLK_SRC : STD_LOGIC := '0';
signal CLK_OUT : STD_LOGIC;
constant CLK_period : time := 10 ns;
begin
    uut: clk_gen port map (CLK_SRC, CLK_OUT);
    clock_process :process
    begin
        CLK_SRC <= '0';
        wait for CLK_period / 2;
        CLK_SRC <= '1';
        wait for CLK_period / 2;
    end process;
end Behavioral;
```

実際の回路ではクロック入力と，スイッチなどのクロックでない入力が混在している．そのようなときには，上述の stim_process と clock_process を並列に書けばよい．

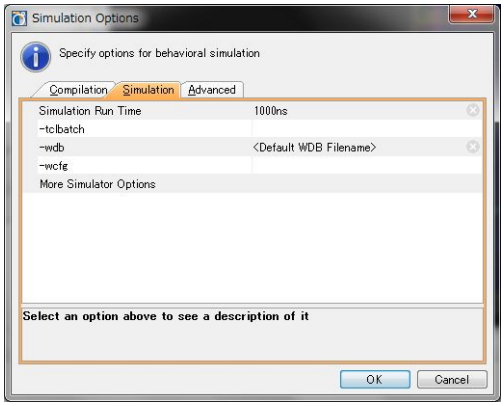
```
architecture Behavioral of test01_tb is
...
begin
    uut: top port map (CLK, SW, LD);
    clock_process :process
    begin
        ...
    end process;
    stim_process :process
    begin
    ...
    end process;
end Behavioral;
```

5.2 ビヘイビアシミュレーション

Flow Navigator → Project Manager の「Run Behavioral Simulation」をクリックする．出てきたウィンドウでテストベンチを選択する．



Option からシミュレーション時間を設定する．デフォルトでは 1000 ns になっている．



Launch でシミュレーションを開始する．

シミュレーションが終わると ISim のウィンドウが表示される．画面上部の Zoom to Full View を押して全体を表示する．

3 章で作成した And 回路のビヘイビアシミュレーションの結果を **Fig. 5.1** に示す．

Fig. 5.1: And 回路のビヘイビアシミュレーションの結果．

課題 6 のクロック生成回路のビヘイビアシミュレーションの結果を **Fig. 5.2** に示す．

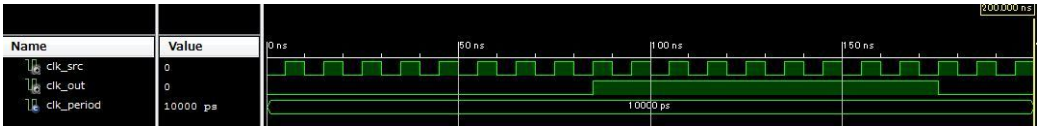


Fig. 5.2: クロック生成回路（18 分周回路）のビヘイビアシミュレーションの結果．

5.3 タイミングシミュレーション

配置配線が終わったあとの，信号の遅延を考慮したシミュレーションをタイミングシミュレーションという．本節ではタイミングシミュレーションの手順を示す．

Implementation が済んでいない場合は，Flow Navigator → Implementation の Run Implementation を実行する．Open Implemented Design を選択する．Implemented Design の Run Timing Simulation をクリックする．ビヘイビアシミュレーションと同様にシミュレーションの設定をして，Launch．

3 章で作成した And 回路のタイミングシミュレーションの結果を **Fig. 5.4** に示す．

結果を拡大すると，入力 SW[0] は 300 ns，SW[1] は 500 ns に'1' になり，時刻 507.071 ns において出力 LD が'1' になった．つまり，出力には 7.071 ns の遅延があることになる．この遅延は，ビヘイビアシミュレーションでは見られなかったものである．

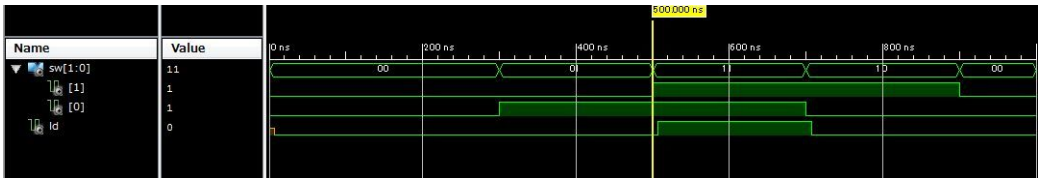


Fig. 5.3: And 回路のタイミングシミュレーションの結果．

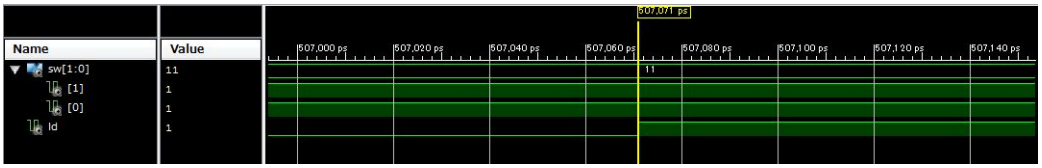


Fig. 5.4: And 回路のタイミングシミュレーションの結果（500 ns 付近を拡大したもの）．

クロック生成回路では何 ns 遅れるだろうか．

第 6 章　演習課題

【課題 1】基本的な論理回路（組み合わせ回路）

3 章で作成した，test01.vhd を拡張して，2 入力の AND，OR，XOR 回路を作成せよ．入力はスイッチ BTNL および BTNR とし，出力は AND を LD0，OR を LD1，XOR を LD2 とせよ.（ピンについては第 9 章を参照）

【課題 2】1-bit 全加算器

被加数を入力 A，加数を入力 B, 桁上がりの入力を C_{in} とする．加算結果を出力 S, 桁上がりを出力 C_{out} とする．下記の真理値表を完成させ，C_{in}, A, B をスイッチ SW2[V9], SW1[T9], SW0[T10] とする, C_{out} を LD1[V16], S を LD0[U16] とする.

(ヒント: 真理値表を Case 文で実現しよう)

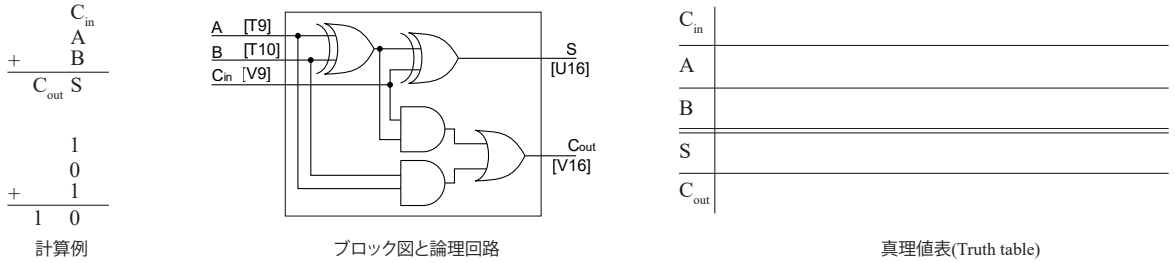


Fig. 6.1: 全加算器 (Full Adder).

【課題 3】4-bit 加算器 (unsigned)

被加数を入力 A(4)，加数を入力 B(4), 桁上がりの入力を無しとする．加算結果を出力 S(5)(桁上がりを S の最上位とする)．IEEE.NUMERIC_STD.ALL を利用して入力の桁数を調整せよ (例：**Table. 4.6**)．A(4) をスイッチ SW7-4[T5,V8,U8,N8], B(4) をスイッチ SW3-0[M8,V9,T9,T10], S(5) を LD4-0[M11, V15, U15, V16, U16] とする.

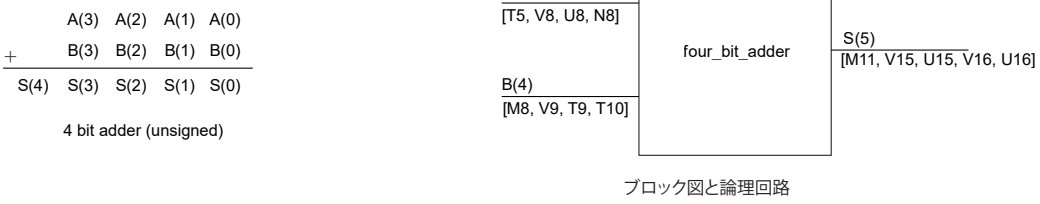


Fig. 6.2: 4bit 加算器 (4-bit Adder).

【課題 4】4-bit 加算器のビヘイビアシミュレーション

課題 3 で作成した 4-bit 加算器回路のビヘイビアシミュレーションを実行せよ．ランダムで 5 個以上の加算結果を確認する.

(Tip：**Fig. 6.3** のように， シミュレーションの表示が自由に変えられる. やり方: 信号を右クリック->Radix->表示したいフォーマットにすればよい.)

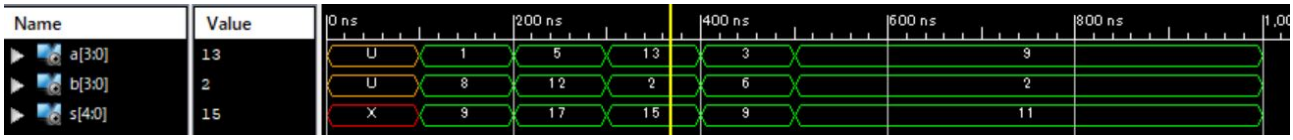


Fig. 6.3: ランダムで 5 個以上の加算結果を確認した例

【課題 5】7 セグメントディスプレイへの表示（組み合わせ回路）

7 セグメントディスプレイに 16 進数の数値を表示させる．7 セグメントディスプレイ構成の詳細は第 9 章に参考せよ．下記の真理値表を完成させ,「4 つのスイッチ（SW3 から SW0）で 2 進数入力した値」を 7 セグメントディスプレイの 1 つ（7seg0）に表示する回路を作成せよ．

真理値表について，通常，7 セグでは B と D は小文字の b と d で表現されることに注意すること（B と 8，D と 0 の区別がつかないため）．また，負論理であるため'0' のときに LED が点灯することにも注意．

エンティティの名称と構成は **Fig. 6.4** のとおりにすること．図中の SW(4) は SW 信号が 4 ビットのベクトルであることを意味する.「4 つのスイッチで 2 進数入力」とは，4 つのスイッチを ON，OFF，ON，OFF（"1010"）としたときに A を表示する，ということを意味する．4 つの 7 セグはアノードコモン接続されており，4 つのうちのどれを点灯させるかは ANx 信号で決める．この課題では 7seg0 だけを点灯させたいので，SEG_SEL 信号には常に"1110" を入力する．

入力	出力							
DATA	CA	CB	CC	CD	CE	CF	CG	DP
0000 (0)								
0001 (1)								
0010 (2)	0	0	1	0	0	1	0	1
0011 (3)								
0100 (4)								
0101 (5)								
0110 (6)								
0111 (7)								
1000 (8)								
1001 (9)								
1010 (A)								
1011 (b)								
1100 (C)								
1101 (d)								
1110 (E)								
1111 (F)								

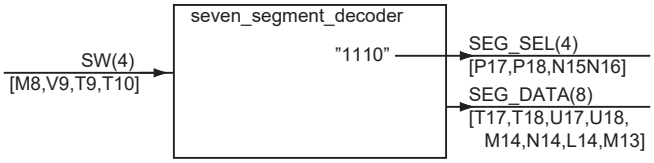


Fig. 6.4: 課題 5 のブロック図.

【課題 6】クロック生成回路（順序回路）

100 MHz のクロックから正確に 5 Hz のクロックを生成する回路を作り，その動作を LD0 で確認せよ. (ヒント：元になる 100 MHz のクロックにしたがうカウンタ回路を作成し，そのカウンタの値を使って，作りたい周波数のクロックとなるように信号をつくるとよい．カウンタ回路は **Table. 4.5** に参考.)

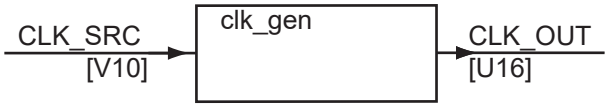


Fig. 6.5: 課題 6 のブロック図.

【課題 7】十進数で 4-bit 加算器の結果を表示（コンポーネントの利用）

4 つの 7 セグメントディスプレイを，各桁を時分割で表示（ダイナミック点灯）させて, 課題 3 で作成した 4-bit 加算器の結果を十進数で表示せよ. ブロック図を **Fig. 6.6** に示す.

課題 3 で作成した 4-bit 加算器の計算結果を二桁の十進数に変更し、それぞれの桁を 4-bit 二進数 (Sum_H, Sum_L) として出力すれば、課題 5 で作成した seven_segment に入力可能.

Sum_H, Sum_L の計算には, 加算結果のデータ型を integer にすると, 乗算, 除算, mod などの演算子が使えるので便利 (**Table. 4.4**). その後, 演算結果を integer から unsigned に戻せばよい (型変換は **Table. 4.2** に参考せよ).
(注意: **Table. 4.6** と同様に加算結果の中間シグナル TMP_I を std_logic_vector にしてしまうと Synthesis error が出るので, TMP_I を unsigned に変更せよ.)

また, seven_segment は，SEG_SEL 信号を削除すればそのまま使えるはずである．二桁表示するため，seven_segment を二個使おう.

課題 6 で作成した clk_gen は，分周比を調節すればそのまま使えるはずである．ブロック図中の clk_in は, およそ 60 Hz 以上であれば人の目には 4 桁が同時に表示されているように見えるはずである.

あとは，クロックに応じて SEG_SEL と SEG_DATA を生成するプログラム（図中の点線部）を書けばよい. SEG_SEL 信号は **Fig. 6.6** の波形図に示したように，表示数値と合わせて SEG_DATA を出力するのがよいだろう.

ある瞬間には AN0 から AN3 のどれか 1 つだけ'0' になることに注意して，SEG_SEL を作る.

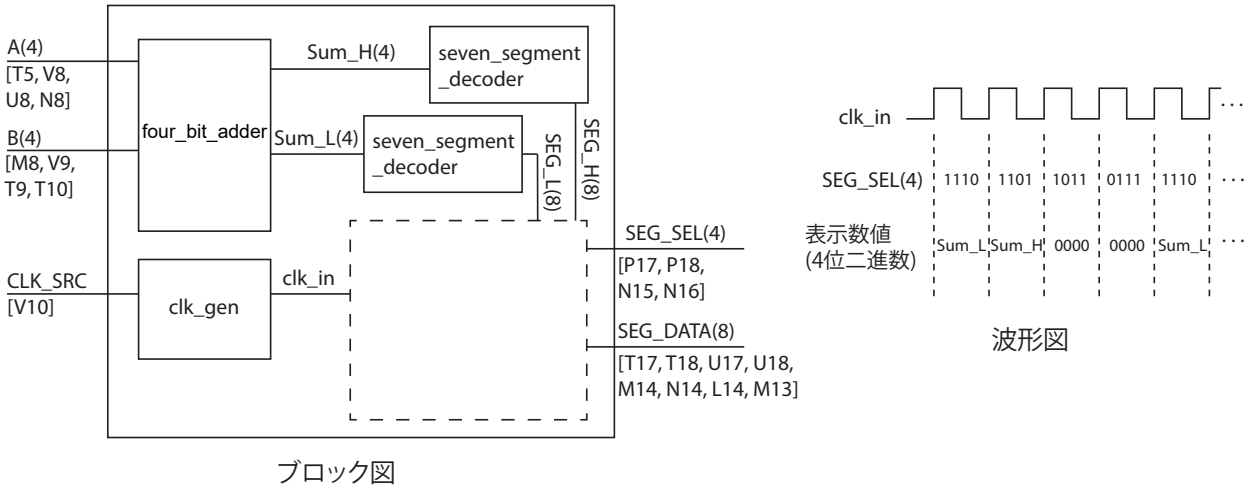


Fig. 6.6: 課題 7 のブロック図.

【課題 8】D フリップフロップ (dff) を使って加算器の計算結果を一時的維持

7 章にある D フリップフロップ (dff) を使うことで, 課題 2 で作成した 1 bit 全加算器の計算結果を 1Hz で更新せよ.Dff の 1 Hz クロック入力は課題 6 で作られた clk_gen を用いて生成できる. ブロック図は **Fig. 6.7** に参照. 同一

の入力に対し，全加算器の出力そのままのものに加えて，dff を介して同期出力させたものの計二種の出力を得られるようにせよ.

入力のスイッチを変えながら二種類の出力を比較せよ. 全加算器そのままの出力は入力の変化に瞬間的に追従する一方で, dff を介した出力 LED は 1 Hz クロックの影響を受けて遅延が生じるはずである.

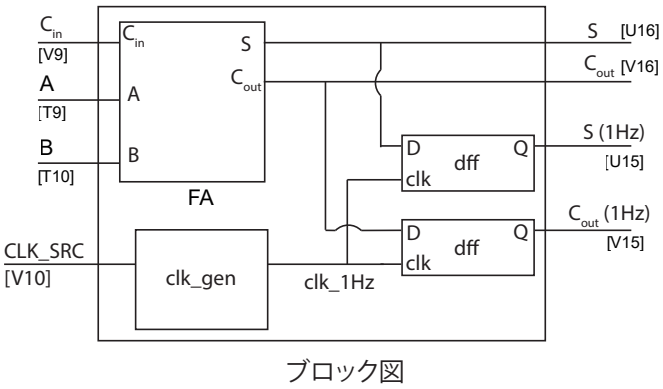


Fig. 6.7: 課題 8 のブロック図.

【課題 9】Pipeline のアーキテクチャで 1-bit 加算器から構成される 4-bit 加算器を実現する

加算器の bit 数が莫大になるとデジタル回路の複雑度も急劇に増え，やがて実際の回路面積も大きくなる．Pipeline アーキテクチャはその解決策の一つとして提案された．このアーキテクチャを用いて，M×N-bit の加算は M-bit の加算器を N 個があれば実現可能であり，実際の CPU にもよく使われていた．

本課題は **Fig. 6.8** のように，4-bit の加算を 4 つの 1-bit 全加算器 (FA) で実現する．また，7 章にある D フリップフロップ (dff) も必要となる．その使い方は課題 8 を参考にせよ．(注意: 表示を簡潔にするため，**Fig. 6.8** における dff の clk 入力は省略した．)

4-bit の加算を 4 回に分けて計算し，それぞれの回の計算結果を dff で保存して，次回の計算に入力する形で行う．

被加数を入力 A(4)，加数を入力 B(4)，桁上がりの入力を C_{in} とする．加算結果を出力 S(4)，桁上がりを出力 C_{out} とする．

課題 4 と同様に，ランダムで 5 個以上の加算結果を確認せよ．Pipeline の特徴として，**Fig. 6.9** のように，入力が変化した後 4 クロック周期の遅延を経て加算結果が出力されることを確認できたらよい．

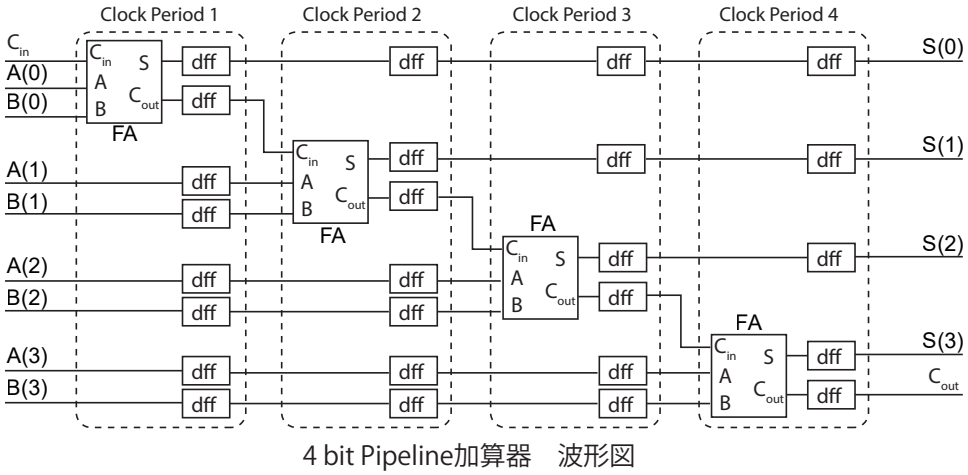


Fig. 6.8: Pipeline のアーキテクチャ．

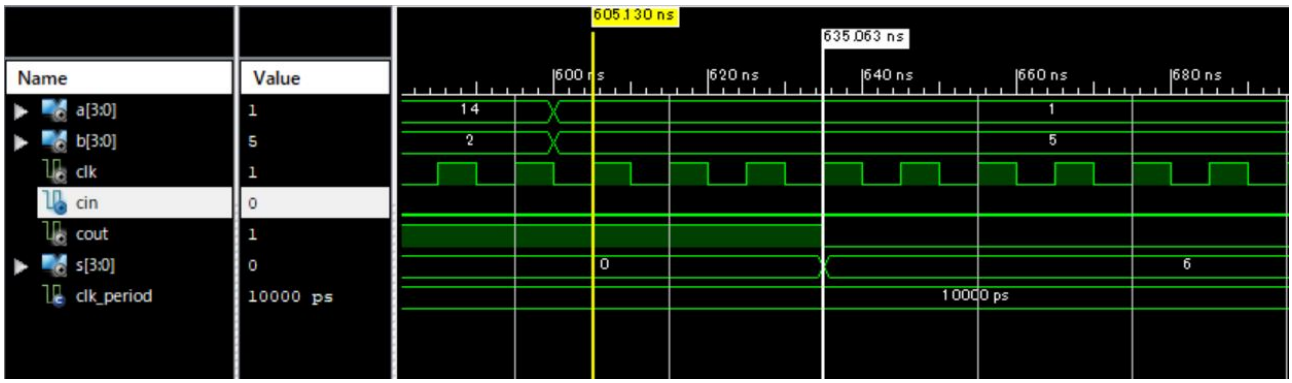


Fig. 6.9: ビヘイビアシミュレーションの実行例．

第7章　サンプルプログラム

7.1　Dフリップフロップ

D フリップフロップは最も基本的な順序素子である．クロック信号の立ち上がりに同期して，入力 D の値を読み込み，その値を Q に出力するとともに，値を保持する．D フリップフロップのブロック図を **Fig. 7.1** に，VHDL 記述の例を **Table. 7.1** に示す.

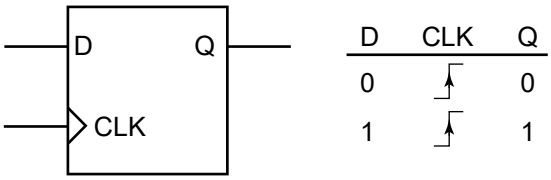


Fig. 7.1: D フリップフロップ.

Table. 7.1: D フリップフロップの VHDL.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity dff is
    Port (CLK, D : in STD_LOGIC;
          Q : out STD_LOGIC);
end dff;

architecture Behavioral of dff is
begin
  process (CLK) begin
    if rising_edge(CLK) then    -- クロックの立ち上がりエッジの検出
      Q <= D;
    end if;
  end process;
end Behavioral;
```

7.2 カウンタ

カウンタは，クロックに同期してデータの値のインクリメントやデクリメントを行うものであり，クロック信号の周波数を落とすための分周や回路内部の制御を行う際に多用される．カウンタのブロック図を **Fig. 7.2** に，VHDL 記述の例を **Table. 7.2** に示す．

`Q <= Q + 1` のようにしたくなるが，2 つの点で間違いである．1 つめの間違いは，Q は出力なので入力として扱えない（右辺には書けない）．2 つめの間違いは，Q は STD_LOGIC_VECTOR なので，整数を足すことはできない．

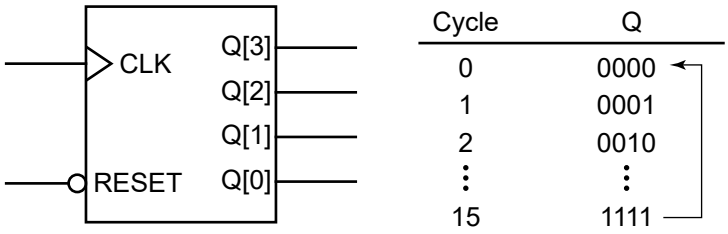


Fig. 7.2: 4 ビットカウンタ．

Table. 7.2: 4 ビットカウンタの VHDL．

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity counter is
    Port (CLK, RESET : in STD_LOGIC;
          Q : out STD_LOGIC_VECTOR (3 downto 0));
end counter;

architecture Behavioral of counter is
begin
    process (CLK, RESET)
    variable COUNT: UNSIGNED (3 downto 0);  -- カウンタ値を保持する 4 ビットの変数
    begin
        if (RESET = '1') then            -- 非同期のリセット（クロックに同期しない）
            COUNT := (others => '0');
        elsif rising_edge(CLK) then
            COUNT := COUNT + 1;
        end if;
        Q <= std_logic_vector(COUNT);
    end process;
end Behavioral;
```

7.3 シフトレジスタ

シフトレジスタは入力したデータをクロックに同期してシフトさせるものである．これを利用すれば，データを指定したクロック数だけ遅らせることや，シリアル入力をパラレル出力に変換することなどができる．シフトレジスタの VHDL 記述の例を **Table. 7.3** に示す.

Table. 7.3: シフトレジスタの VHDL.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity shift_reg8 is
    Port (QIN: in STD_LOGIC_VECTOR (7 downto 0);
          CLK, RESET: in STD_LOGIC;
          QOUT: out STD_LOGIC_VECTOR (7 downto 0));
end shift_reg8;

architecture Behavioral of shift_reg8 is
begin
  process (CLK, RESET)
  variable REG: STD_LOGIC_VECTOR (7 downto 0) := (others => '0');
  begin
    if rising_edge(CLK) then
      if (RESET = '1') then
        REG := QIN;
      else
        REG := REG(6 downto 0) & REG(7);
      end if;
    end if;
    QOUT <= REG;
  end process;
end Behavioral;
```

7.4 ステートマシン

順序回路の中で実践的によく使うのが「ステートマシン」である．ステートマシンとは，いくつかの状態（ステート）を持ち，条件を満たしたときに別の状態へと遷移するものである．まず **Fig. 7.3** のような状態遷移図を描く．このステートマシンは「MD0」,「MD1」,「MD2」という 3 つの状態を持ち，リセット信号（RESET = '1'）で「MD0」となる．この状態で，入力 1 = '1' となると（例えばスイッチが押されると）「MD1」に遷移し，INPUT1 = '1' の間は「MD1」を保持する．そして「MD1」のときに，INPUT1 = '0' となると（例えばスイッチを OFF にすると）「MD2」に遷移し，INPUT1 = '0' の間は「MD2」を保持する．同様に「MD2」のときに，INPUT2 = '1' となると「MD0」に遷移する．それぞれの状態での出力は状態遷移図に示した通りである.「MD1」のような場合もあり得る．

ステートマシンの VHDL を **Table. 7.4** に示す．type を用いて STATE という新しい型を宣言している．状態遷移図の矢印や出力に関する部分の記述は，case 文や with select 文を使って記述することが多い（process 文の中では case 文を，process 文の外では with select 文を用いる）．状態遷移図によっては，矢印の部分と出力の部分の記述を別の process 文として書くほうが良い場合もある．

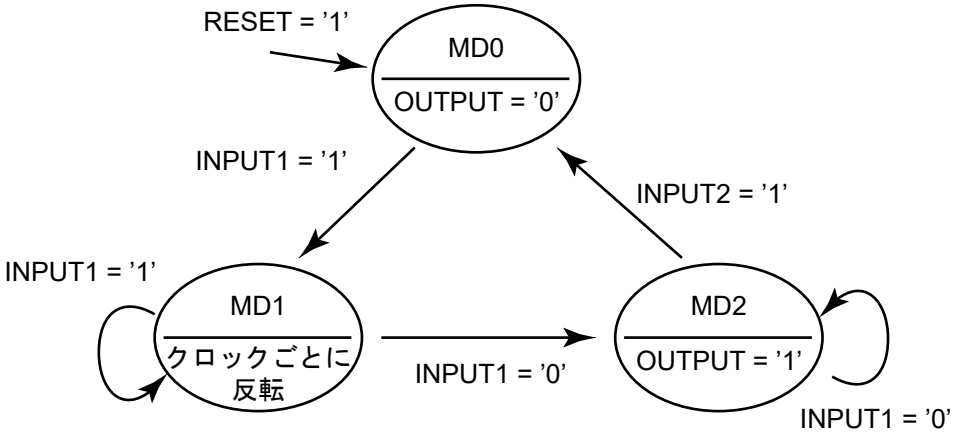


Fig. 7.3: 状態遷移図.

Table. 7.4: ステートマシンの VHDL.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity state_machine is
    Port (RESET, CLK, INPUT1, INPUT2 : in  STD_LOGIC;
          OUTPUT : out  STD_LOGIC);
end state_machine;
```

Table. 7.5: ステートマシンの VHDL のつづき.

```
architecture Behavioral of state_machine is
type STATE is (MD0, MD1, MD2);   -- MD0, MD1, MD2 からなる STATE という型を宣言
signal ST_I :STATE;              -- 現在の状態の信号
signal OUTPUT_I : STD_LOGIC;
begin
  OUTPUT <= OUTPUT_I;
  process (RESET, CLK) begin
    if (RESET = '1') then
      ST_I <= MD0;                    -- 初期化
    elsif rising_edge(CLK) then
      case ST_I is
        when MD0 =>
          OUTPUT_I <= '0';               -- 状態遷移図の出力にあたる部分の記述
          if (INPUT1 = '1') then           -- 状態遷移図の矢印にあたる部分の記述
            ST_I <= MD1;
          else
            ST_I <= MD0;
          end if;
        when MD1 =>
          OUTPUT_I <= not OUTPUT_I;
          if (INPUT1 = '0') then
           ST_I <= MD2;
          else
           ST_I <= MD1;
          end if;
        when MD2 =>
          OUTPUT_I <= '1';
          if (INPUT2 = '1') then
           ST_I <= MD0;
          else
           ST_I <= MD2;
          end if;
        when others =>
          OUTPUT_I <= '0';
          ST_I <= MD0;
      end case;
    end if;
  end process;
end Behavioral;
```

7.5 分周回路をコンポーネントとして利用した回路

1 秒ごとに 8 ビットカウンタの値を増やし，その値を LED アレイに表示する．clk_gen コンポーネントでは，クロック 100 MHz を分周して 1 Hz のクロックをつくっている．

Table. 7.6: コンポーネントを呼び出す top.vhd ファイル．

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity top is
    Port ( CLK_SRC : in STD_LOGIC;
           LD : out STD_LOGIC_VECTOR (3 downto 0));
end top;

architecture Behavioral of top is

signal CLK1s : STD_LOGIC;

component clk_gen                          -- 呼び出すコンポーネントの宣言
    Port ( CLK_SRC : in STD_LOGIC;
           CLK_OUT : out STD_LOGIC);
end component;

begin

    U1 : clk_gen port map (CLK_SRC, CLK1s);   -- コンポーネントの呼び出し

    U2 : process (CLK1s)                       -- 直接記述も可能
    variable COUNT : UNSIGNED (3 downto 0) := (others => '0');
    begin
        if rising_edge(CLK1s) then
           COUNT := COUNT + 1;
        end if;
        LD <= std_logic_vector(COUNT));
    end process;

end Behavioral;
```

7.6 ボタンを押している間だけカウントアップする回路

ボタンを押している間だけカウントアップする回路の例である．25 ビットのカウンタ変数 COUNT の上位のビットを LED に表示している．

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity top is
    Port ( CLK_SRC : in STD_LOGIC;
           BTN : in STD_LOGIC;
           LD : out STD_LOGIC_VECTOR(7 downto 0));
end top;

architecture Behavioral of top is

begin

    process
    variable COUNT : UNSIGNED (24 downto 0) := (others => '0');
    begin
        wait until rising_edge(CLK_SRC);
            if (BTN = '1') then
                COUNT <= COUNT + 1;
            end if;
        LD <= std_logic_vector(COUNT(24 downto 17));
    end process;

end Behavioral;
```

7.7 スイッチのチャタリングを防止する回路

BTN を押すと，LD4 と LD5 がカウントアップしていく．7seg0 に表示している COUNT3_I はチャタリング防止の処理をしていない BTN_I を使っているので，たまに 2 つ増えることがある．7seg1 に表示している COUNT2_I はチャタリング防止の処理をした CHAT_I を使っているので，そのようなことがおきないはず．BTN を何度も押し続けると，次第に LED4 と LED5 の数字がずれていく．clk_gen_10ms を 10 ms（100 Hz）ではなく 1 Hz 程度にして動作を確認するのも面白い．7seg2 に表示している COUNT1_I はこのクロックに従いカウントアップしている．

Table. 7.7: チャタリング防止回路の top.vhd ファイル．

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity top is
    Port ( CLK : in STD_LOGIC;
           BTN : in STD_LOGIC;
           SEG_DATA : out STD_LOGIC_VECTOR (7 downto 0);
           SEG_SEL : out STD_LOGIC_VECTOR (3 downto 0));
end top;

architecture Behavioral of top is
component clk_gen_10ms
    Port ( CLK_SRC : in STD_LOGIC;
           CLK10ms : out STD_LOGIC);
end component;
component clk_gen_1ms
    Port ( CLK_SRC : in STD_LOGIC;
           CLK1ms : out STD_LOGIC);
end component;
component seven_segment
    Port ( CLK : in STD_LOGIC;
           SEVENSEG_DATA : in STD_LOGIC_VECTOR (15 downto 0);
           SEG_DATA : out STD_LOGIC_VECTOR (7 downto 0);
           SEG_SEL : out STD_LOGIC_VECTOR (3 downto 0));
end component;
component chat
    Port ( BTN : in STD_LOGIC;
           CLK : in STD_LOGIC;
           CHAT_OUT : out STD_LOGIC);
end component;
```

Table. 7.8: チャタリング防止回路の top.vhd ファイル（つづき）.

```
signal BTN_I : STD_LOGIC;    -- チャタリング防止処理していない信号
signal CHAT_I : STD_LOGIC;   -- チャタリング防止処理済みの信号
signal CLK1ms_I : STD_LOGIC;    -- ダイナミック点灯のためのクロック
signal CLK10ms_I : STD_LOGIC;   -- チャタリング防止処理のためのクロック
signal SEVENSEG_DATA : STD_LOGIC_VECTOR (15 downto 0);  -- 7 セグの表示内容
signal SEG1_I : STD_LOGIC_VECTOR (3 downto 0);
signal SEG2_I : STD_LOGIC_VECTOR (3 downto 0);
signal SEG3_I : STD_LOGIC_VECTOR (3 downto 0);

begin
    U_CLK1ms : clk_gen_1ms port map (CLK, CLK1ms_I);
    U_CLK10ms : clk_gen_10ms port map (CLK, CLK10ms_I);
    U_CHAT : chat port map (BTN, CLK10ms_I, CHAT_I);
    U_SEVENSEGMENT : seven_segment port map
    (CLK1ms_I, SEVENSEG_DATA, SEG_DATA, SEG_SEL);

    BTN_I <= BTN;
    SEVENSEG_DATA <= "0000" & SEG1_I & SEG2_I & SEG3_I;

    process (CLK10ms_I, BTN_I)
    variable STATE3 : STD_LOGIC := '0';  -- ボタンが押されたかどうかを保持する変数
    variable COUNT3 : UNSIGNED (3 downto 0) := (others => '0');
    begin
        if rising_edge(CLK10ms_I) then
            if (BTN_I = '1') then
                if (STATE3 = '0') then  -- ボタンが押されて STATE が'0' のとき
                    COUNT3 := COUNT3 + 1;  -- カウントアップする
                    STATE3 := '1';  -- ボタンが押されたら STATE を'1' にする
                end if;
            else
                STATE3 := '0';  -- ボタンから手を離したら STATE を'0' にもどす
            end if;
        end if;
        SEG3_I <= std_logic_vector(COUNT3);
    end process;
```

Table. 7.9: チャタリング防止回路の top.vhd ファイル（さらにつづき）.

```
    process (CLK10ms_I, CHAT_I)
    variable STATE2 : STD_LOGIC := '0';
    variable COUNT2 : UNSIGNED (3 downto 0) := (others => '0');
    begin
        if rising_edge(CLK10ms_I) then
            if (CHAT_I = '1') then
                if (STATE2 = '0') then
                    COUNT2 := COUNT2 + 1;
                    STATE2 := '1';
                end if;
            else
                STATE2 := '0';
            end if;
        end if;
        SEG2_I <= std_logic_vector(COUNT2);
    end process;

    process (CLK10ms_I)
    variable COUNT1 : UNSIGNED (3 downto 0) := (others => '0');
    begin
        if rising_edge(CLK10ms_I) then
            COUNT1 := COUNT1 + 1;
        end if;
        SEG1_I <= std_logic_vector(COUNT1);
    end process;
end Behavioral;
```

Table. 7.10: チャタリング防止回路の chat.vhd ファイル.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity chat is
    Port (BTN : in STD_LOGIC;
          CLK : in STD_LOGIC;
          CHAT_OUT : out STD_LOGIC);
end chat;

architecture Behavioral of chat is
signal BTN_I : STD_LOGIC;
signal CHAT_OUT_I : STD_LOGIC;

begin
    BTN_I <= BTN;
    process (CLK) begin
        if rising_edge(CLK) then
            CHAT_OUT <= BTN_I;
        end if;
    end process;
end Behavioral;
```

第 8 章　FAQ & Tips

Project フォルダーに vhdl ファイルの場所

新規作成した source file: project 名.srcs/sources_1/new
bit file: project 名.runs/impl_1
新規作成した test bench file: project 名.srcs/sim_1/new

Syntax Error が消えない

エラーメッセージをクリックすると，Error が検出された行にカーソルが移動する．間違いはたいていエラーメッセージが指している行より前にある．Warning メッセージが解決のヒントになることもある．VHDL の中に全角スペースがあるとエラーになる．

Syntax Error の箇所が表示されない

Hierarchy の上部にメッセージが出ることがある．Hierarchy を更新すると表示される場合がある．または，プロジェクトを一度閉じて，再度開くと表示される場合がある．

unsigned が定義されていないというエラーが出る

use IEEE.NUMERIC_STD を追加しよう．

Synthesis の結果を Open できない

出力に何も接続されていない状態で Synthesis をすると，中身のないものができる．この状態では Open できない．

I/O Planning でポート（入出力信号）名が表示されない

文法チェックは通っても，回路にどこかおかしなところがある場合がある．使われていない端子，接続されていない端子は，自動的に省略される．例えば，出力につながっていない入力信号にはアサインすることはできない．

論理合成で multiple drivers とかいうエラーが出た

Signal xxx in unit yyy is connected to following multiple drivers
ある信号に複数の入力が同時に接続されうる状況になっている．**Table. 8.1** は，SW1 を High にすると LED が点灯し，SW 2 を High にすると LED が消える回路が，SW1 と SW2 が同時に High になったらどうなるだろうか？

Table. 8.1: 【悪い回路】複数の入力が接続されうる回路の例.

```
process (SW1)
    if (SW1 = '1')) then
        LD <= '1';
    end if;
end process;

process (SW2)
    if (SW2 = '1')) then
        LD <= '0';
    end if;
end process;
```

インプリメンテーションで A clock IOB... というエラーが出た

A clock IOB / BUFGMUX clock component pair have been found that are not placed at an optimal clock IOB / BUFGMUX site pair.

これはスイッチやボタンの入力がクロックと認識されてしまっている.（そのように意図していなくてもプログラムの書き方からクロックと判断されている）解決策として、スイッチやボタンには rising_edge 使わないこと. スイッチやボタンは何らかのクロックに同期して読むことが考えられる.

別の解決方法として，ucf ファイルを編集してクロックではないと制約を書いておくとエラーがでなくなる. 例として,「BTN というポートがクロックではないという制約」は下記のようになる.

```
NET "BTN" CLOCK_DEDICATED_ROUTE = FALSE;
```

Table. 8.2: クロックに同期してボタンを読み込む回路の例.

```
process
    if rising_edge(CLK) then
        ...
        if (BTN = '1')
            ...
        end if;
    end if;
```

ISim が起動しない

シミュレーションがエラーで止まっていないか，Message を確認する．ISim は複数起動できないので，前に起動したものは閉じておく．

シミュレーションを実行すると，Metavalue Detected といわれる

値の定まっていない信号があるので，信号の初期値を明示するとよい．どの信号が引っかかっているかを探すには，ISim の画面左の「Instance and Process Name」で信号を選択し，値が'U' のものを探すとよい．

シミュレーションで内部信号を見たい

ISim の画面で，コンポーネントを選択して内部信号を波形表示に追加する．その状態ではまだ波形は見えないので，もう一度シミュレーションを走らせる必要がある．

プログラムを書き換えるたびにすべての項目を Run していくのが面倒

ピンアサインなどを変更する必要がないなら，VHDL を書き換えてから「Generate Bitstream」をすると，それよりも前の段階までの Run を自動で実行する．

プログラムしていない動作をする

Configuration に使っているビットストリームファイルが正しいことを確認する．ビヘイビアシミュレーションをして内部信号を確認する．

コンフィギュレーションしても動かない

bit ファイルを作りときに最上位のモジュールをきちんと選択しているか確認する. Hierarchy で vhd を右クリックして「Set as Top」とする．iMPACT で以前のプロジェクトの.bit ファイルを読み込んでいないか確認する．

第 9 章　Nexys3 ボードについて

演習で使用する Nexys3 ボードについて, 演習で使用する機能を中心に説明する.

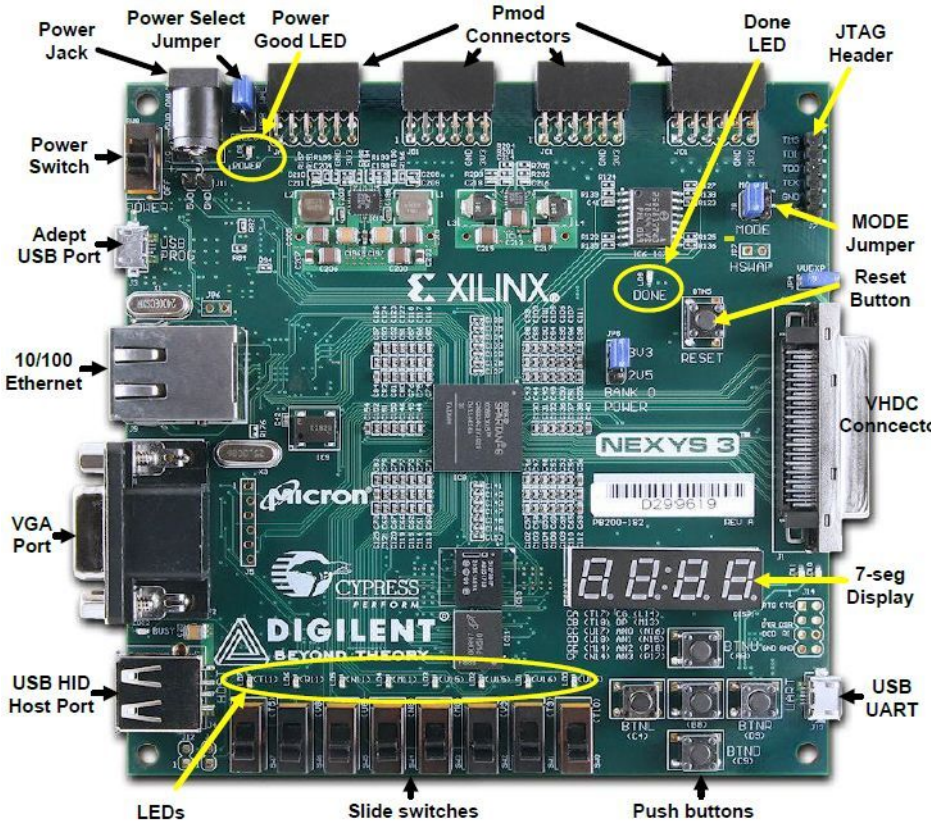


Fig. 9.1: Nexys3 ボードの概観（Digilent 社資料（Nexys3_rm_V2.pdf）より抜粋）.

パソコンとの接続と電源

パソコンとの接続にはボード左上の「USB PROG」を使用する. 演習では USB で給電する. 電源スイッチは USB コネクタの近くにある.

スイッチとボタン

8 個のスライドスイッチ（SW0 から SW7）と 5 個のボタン（BTNL, BTNR, BTNU, BTND, BTNS）が実装されている．スライドスイッチは上方向にスライドすると’1’ になる. ボタンは押すと’1’ になる.

LED アレイと 7 セグメントディスプレイ

スライドスイッチの近くに 8 個の LED アレイ（LD0 から LD7）が実装されている．LD0 から LD7 を’1’ にすると，対応する LED が点灯する．LED に直列接続されている抵抗器の抵抗値は 390Ω である．

4 桁の 7 セグメントディスプレイが実装されている. 制御信号（AN0 から AN3）によってどの桁を点灯させるか指定する．**負論理**であるため，’0’ にすることで, 対応する 7 セグが点灯可能な状態になる.（例：AN0 を’0’ にす

ると，最も右の 7 セグが点灯する）. 7 セグの個々の LED は 8bit の共通バス（CA から CG,DP）に接続され，どの LED を光らせるかを制御できる．このバスも**負論理**であるため，’0’ のときに点灯する．トランジスタのベースに接続された抵抗器の抵抗値は 2.2 kΩ であり，7 セグの各 LED に直列接続されている抵抗器の抵抗値は 100Ω である．

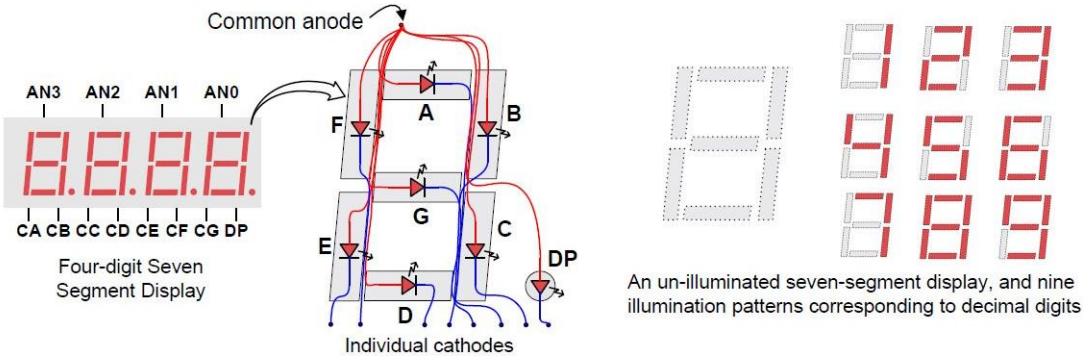


Fig. 9.2: 7 セグの配線（Digilent 社資料（Nexys3_rm_V2.pdf）より抜粋）.

クロック, その他

図示されていないが，100 MHz のクロックのピンは「V10」である．

その他の入出力として，8×4 の Pmod（Peripheral Module）ポート，USB-UART，Ethernet，8-bit VGA がある．

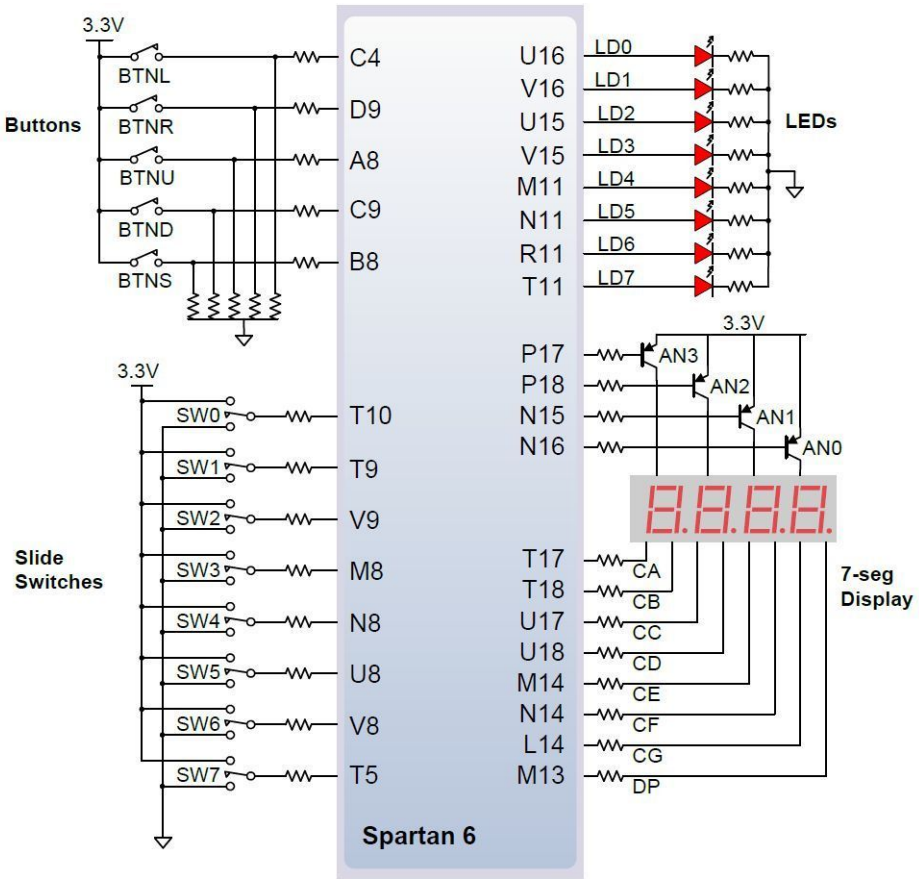


Fig. 9.3: Nexys3 のピン番号（Digilent 社資料（Nexys3_rm_V2.pdf）より抜粋）.