

Zybo Z7-20 ではじめる FPGA

第一部 基礎・RTL 編

わさらぼ合同会社
2018年3月19日版

イントロダクション

FPGA で何ができるか、どうすれば FPGA を使うことができるのか、の概略を俯瞰しましょう。

1 もし〇〇を制御したいけどソフトウェアじゃ難しい

実験などのために、何かの入力を観測しながら、それに応じて何かの処理をする、という機会がありませんか？そのような場合、パソコンやマイコンで実装してみようとしても

- モータやセンサをたくさんつなぎたいから 100 個くらい自由に使える入出力があればいいのに
- システムを乾電池 1 本で動作しないかなあ
- 決められた時間内できちんと処理を終わらせたい/繰り返したい
- 処理専用の特別な命令を実行できたら高速化できるのにな
- いくつもの処理を並行して実行できたらいいのにな
- 361 ビットの値の演算が一発でできたらすっきり書けるのに

など、ソフトウェアではあと一息痒いところに手が届かず、歯痒い思いをしたことはないでしょうか。

1.1 プロセッサがプログラムを実行する

パソコンはもちろんのこと、今やテレビや携帯電話、自動車などのあらゆる製品の中でソフトウェアが動作しています。それらのソフトウェアは、実現したいアプリケーションや動作させる環境に応じて、さまざまなプログラミング言語で記述されています。たとえば、JavaScript は Web アプリケーションを便利に華やかにしてくれますし、C で記述されたプログラムはシステムを細やかに制御し、高速に動作させることができます。

近年のソフトウェアが動作する環境は、プログラミング言語やオペレーティング・システム、ライブラリなどにより上手に隠蔽されていますので、抽象化された世界の上でプログラムを書けるソフトウェア・エンジニアは、ハードウェアを意識することが少ないかもしれません、どのような言語を使ったプログラムでも、ハードウェアであるプロセッサで処理が実行されます。

たとえば、パソコン上で動作するソフトウェアは Intel の Core i7 などのプロセッサの上で動作し、スマートフォンの上のソフトは ARM プロセッサの上で動作しています。また、組み込み機器でも各種マイコンの上で処理が実行されています。一般に、マイコンは、演算装置に加え様々なデバイスを制御するための I/O コントローラを備えているためセンサの入力を読み取ったり、モーターを回したりといった処理をソフトウェアで操作できます。「プロセッサ」や「マイコン」はデバイスそのものがアプリケーションに応じて変化するわけで

はなく、図 1) のように、ソフトウェアによって処理させる内容をその時々で決めることができるため、幅広い用途に活用されています。

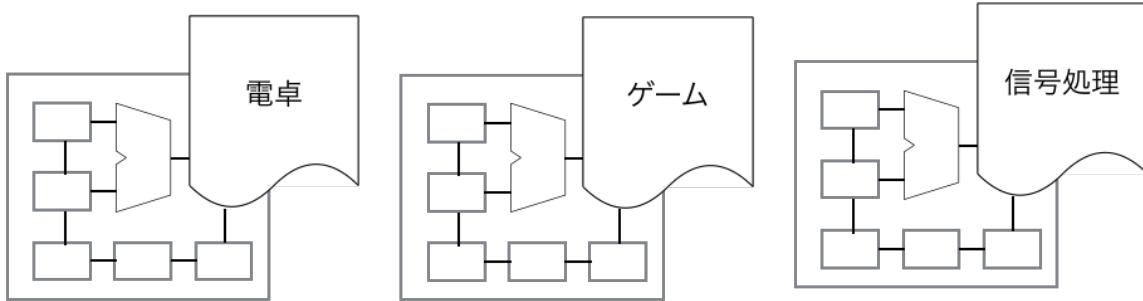


図 1 ソフトウェアプログラムはプロセッサの上で処理される。どんなプログラムを実行する場合でもプロセッサの構造は変わらない

一方で、自由に設計できるようにみえるソフトウェアも、実際に動作するときには、常にプロセッサの制約を受けます。たとえば、プロセッサの処理能力の限界を越えるような速さでの計算や、プロセッサがもっていない I/O を操作するといったことはできません。しかし、デバイスそのものをアプリケーションに応じて変化させて、冒頭に挙げたような「もし〇〇なプログラムが書けたなら」の希望を現実にできる手段があれば、今まで手を出しにくかったアプリケーションが実現できるのではないかでしょうか。

1.2 ハードウェア・デバイスを作るための便利な仕組み

そんなときには、オリジナルのハードウェアを作るという解決策があります。ハードウェアを作るといっても、げじげじの足が付いたデバイスを一つずつはんだ付けしたり、半導体工場に製作を依頼する必要はありません。ハードウェア記述言語 (HDL ; Hardware Description Language) を用いて作成したハードウェア・イメージを専用のデバイス (FPGA) に書き込むだけでオリジナルのハードウェア・デバイスを作る便利な仕組みがあります。FPGA は、図 2 のように、論理回路になる素や、論理回路同士を接続する素がパッケージされた LSI です。

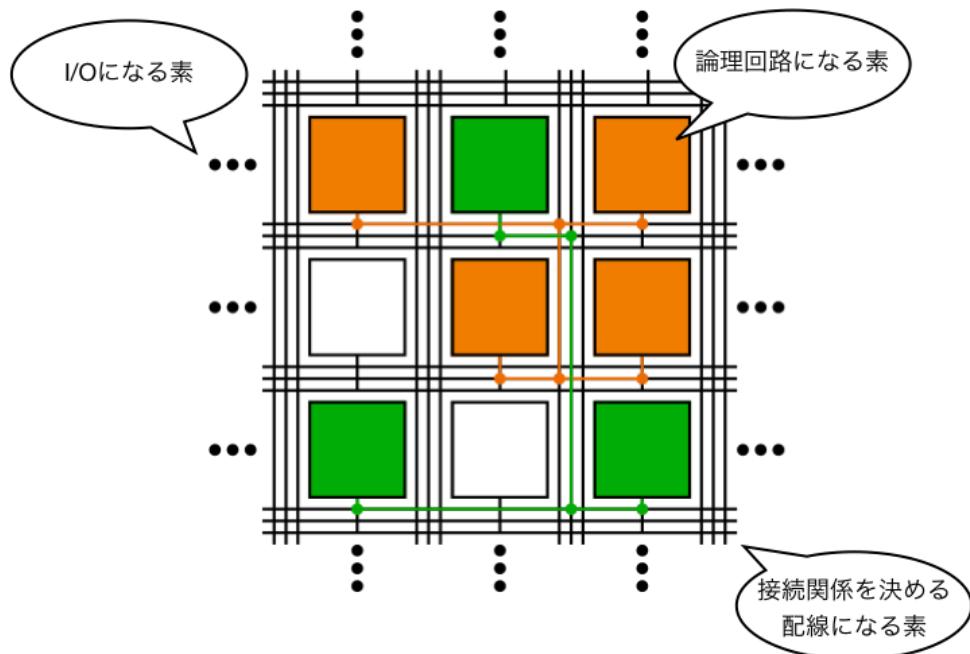


図 2 FPGA は、論理回路の素や配線回路の素がパッケージされた LSI

用途によって、とにかく短い時間で処理をしたい、低消費電力で処理をさせたい、同時にたくさんの I/O にアクセスしたいといったことが求められることもあるでしょう。FPGA は、そんな要求に応えることができる、やりたい処理をさせるための専用ハードウェアを自在に作れる柔らかいプログラム可能なハードウェア・デバイスです(図 3)。

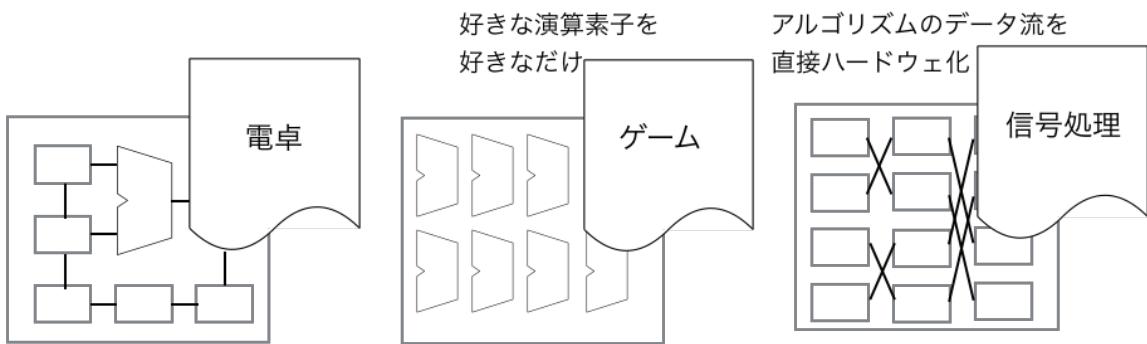
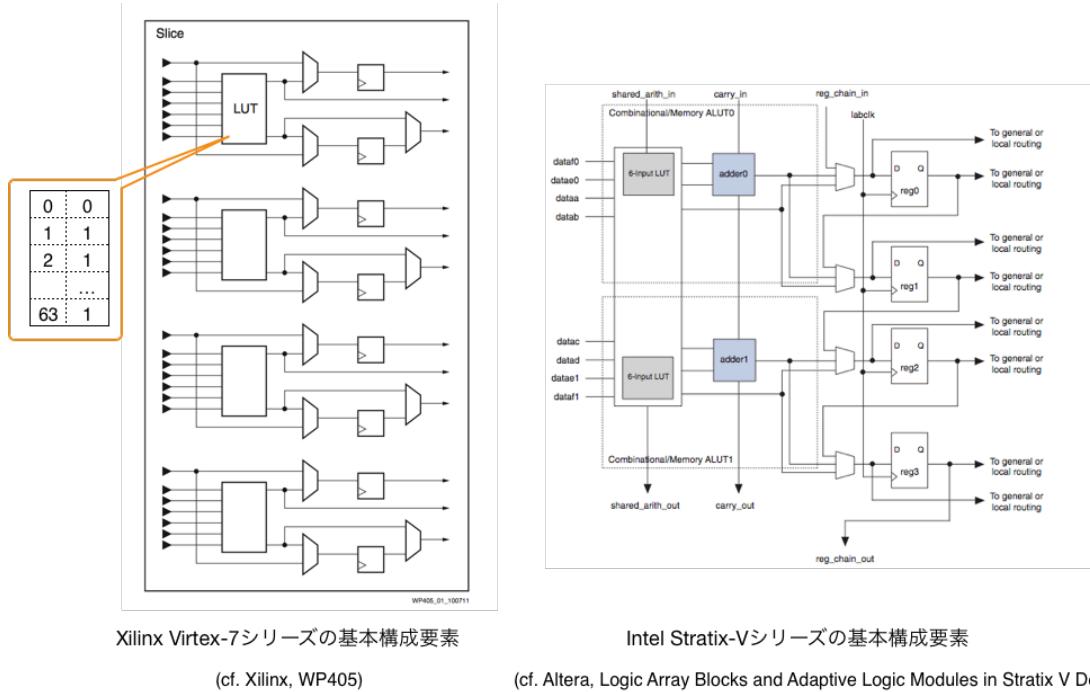


図 3 FPGA は、実現したい処理向けの専用ロジックを自分で作ることができる

1.3 FPGA には小さなメモリ (LUT) がたくさん入っている

FPGA の中身はどのようにになっているのか、もう少し詳しくのぞいてみましょう。図 4 は、代表的な FPGA メーカーである Xilinx と Intel の FPGA の内部構造です。多少違いはありますが、基本的な構成要素は、LUT(Look-up Table) と記憶素子 (D-FF) で構成されるロジック・セルです。



Xilinx Virtex-7シリーズの基本構成要素

(cf. Xilinx, WP405)

Intel Stratix-Vシリーズの基本構成要素

(cf. Altera, Logic Array Blocks and Adaptive Logic Modules in Stratix V Devices)

図 4 FPGA の内部構造

好きな論理演算を実現する鍵は LUT で、これは小規模なメモリのようなものです。入力されたデータに対して出力する信号を決定するためのテーブルの役割を持ちます。

たとえば、図 4 のように、アドレス 0 に 0、それ以外のアドレス 1~63 には、1 と書いてある LUT の場合には、6 入力 1 出力の OR 回路に相当します。同様に、アドレス 0~62 に 0 をアドレス 63 に 1 と書いておけば、6 入力 1 出力の AND 回路を作ることができます。また、1 が奇数個のアドレス、たとえば、アドレス 1, 2, 4, 7 など、のみに 1 と書いておけば、6 入力 1 出力の XOR 回路になります。

入力ビット幅の範囲で好きな論理演算を決められる LUT と、複数の LUT を好きに接続できる接続テーブルによって、大規模な論理演算を好きに構成することができますね。

1.4 ハードウェアと記述言語

LUT と接続テーブルを設定して、ハードウェアを自由に設計できるといっても、ソフトウェアでは簡単に書ける足し算や引き算といった基本的な演算を、一つ一つ論理回路で構成するのは骨が折れる作業です。そこで利用するのがハードウェア記述言語 (HDL; Hardware Description Language) です。

HDL は、C プログラミングなどと同じように変数への加減算や条件分岐などを用いてハードウェアを設計できるプログラミング言語です。ユーザは、まるでプログラムをロードするように、ハードウェア・データを FPGA に書き込むことで所望のデバイスを作ることができます。

2 ハードウェア・プログラミングを理解する三つのポイント

オリジナルのハードウェアを作成するための手段が、ハードウェア・プログラミングです。ソフトウェア・プログラミングでは、プロセッサ（というハードウェア）を動作させるための命令列を設計するのに対し、ハードウェア・プログラミングは、ハードウェアそのものを設計します。ハードウェア・プログラミングにチャレンジするにあたり、ハードウェアの基本概念となる「演算の決め方」と「データの単位」、「処理の動作方式」についてソフトウェア・プログラミングと比較しながら説明します。

2.1 演算の決め方

ソフトウェア・プログラミングでは、演算はプロセッサのもつ命令として決められています。一方でハードウェア・プログラミングでは、論理演算を組み合わせて自分で演算を決めることができます。

■ソフトウェア・プログラミング ソフトウェア・プログラミングは、プロセッサの演算ユニットが持つ機能をどのように利用するかを指示します。プロセッサの演算ユニットでは、足し算や掛け算などの算術演算や、比較演算、分岐などの処理を制御する命令を実行できます。また、信号処理を高速に処理できるように「掛け算して足し算」するといった命令を備えているプロセッサもあります。実際にプロセッサの演算ユニットをどのように使うか、を決定するのはコンパイラの仕事で、プログラマは、普段ほとんど意識する必要はありません。

■ハードウェア・プログラミング ハードウェア・プログラミングは、ハードウェアそのものを作成するので、ベースとなる演算ユニットというものは存在しません。演算処理は、基本的には、NOT, AND, OR の論理演算とそれらの組み合わせで実現します。

2.2 データの単位

ソフトウェア・プログラミングとハードウェア・プログラミングで取り扱い可能なデータの単位について比べてみます。

■ソフトウェア・プログラミング 一般に、プロセッサの作業領域であるレジスタの幅は固定されていて、ソフトウェア・プログラミングではそのサイズでしかデータを取り扱えません。たとえば、レジスタ幅が 32 ビットのプロセッサでは、1 という値も 4294967295 という大きな値も 32 ビット幅のデータとして処理されます。また、512 ビットという大きな値を取り扱うためには、 $512 \div 32$ の 16 回分のデータ処理を必要とします。

■ハードウェア・プログラミング ハードウェア・プログラミングでは、「0」または「1」の値である 1 ビット単位のデータを自由に組み合わせて利用できます。処理内容によって、好きなビット数のデータを定義できるので、大きいデータも小さいデータも自由に定義して処理できます。512 ビットのデータも、ハードウェア・プログラミングなら 1 回のデータ処理で取り扱えます。

2.3 処理の動作方式

ソフトウェア・プログラムは、プロセッサが記述した処理を順序よく実行してくれます。しかし、ハードウェア・プログラムは、生成されたハードウェア回路そのものがデータを処理します。そのため、処理の順序を守るための仕組みが必要な場合には自分でそのしくみを作る必要があります(図 5)。

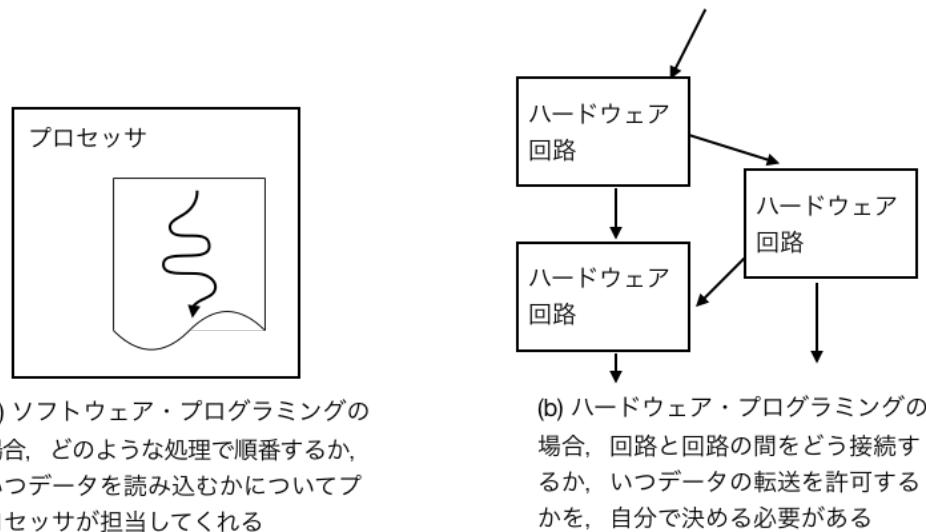


図 5 ソフトウェア・プログラミングではプロセッサが処理順序を決めてくれる。ハードウェア・プログラミングでは、ハードウェアを構成する回路同士の接続関係や、いつデータを転送するかを全部自分で決める必要がある。

■ソフトウェア・プログラミング 今日のプロセッサの基本的な構成方式は、図 6 に示すノイマン型アーキテクチャと呼ばれる方式です。ノイマン型アーキテクチャがプログラムを実行する手順はおよそ次の通りです。

1. 命令を読み込む
2. 読み込んだプログラムの命令を理解する
3. 演算ユニット上で命令を実行する
4. 必要に応じてメモリを読み書きする
5. 1. に戻る

パソコンのプロセッサに見られる「クロック 3GHz」といったスペックのプロセッサでは、プロセッサ内のユニットが 3GHz のクロックに同期して、すなわち 0.33n 秒程度の時間で動作することで、このサイクルが繰り返されます。

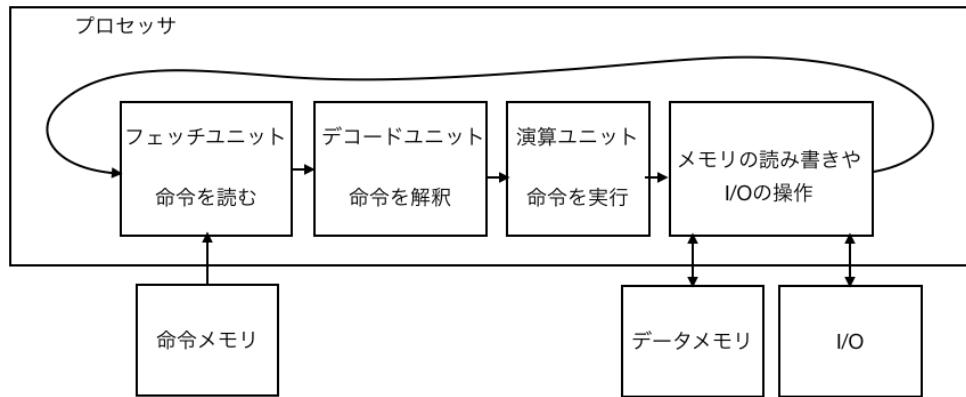


図6 今日のプロセッサの典型的な構成方式。処理性能向上のために、各ユニットは可能な限り並行に動作するように工夫されることが多い。

命令を実行する、演算ユニットでは、決められたビット幅の作業領域の値に足し算や掛け算などの演算を適用できます。高い処理性能を達成するために、複数の演算ユニットを持つプロセッサもあり、それらのプロセッサでは、演算ユニットの個数分だけ同時に命令を処理することができます。勿論、「演算Aの後に演算Bを計算しないと演算Aの結果が保存されない、とか、演算Bの入力が揃う前に計算をはじめてしまう」ということは普通はなく、プロセッサが演算の順番を守ってくれます。

■ハードウェア・プログラミング ハードウェアの基本となる論理演算回路において、入力信号を与えてから出力信号が得られるまでの時間は、トランジスタの特性や論理演算の段数で決定されます。そのため、複雑なゲートが関係し合うような回路で、データが正しく到達することを考慮するのは大変困難です(図7)。

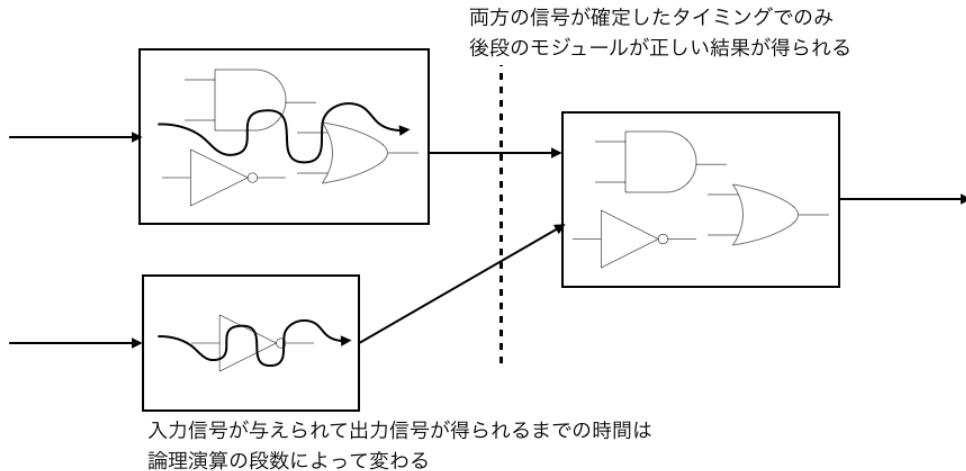


図7 入力された信号に対して出力が確定するまでの時間は、論理演算回路を構成する論理演算の段数で決まる。

ハードウェア・プログラミングでは、これを解決するために大抵の場合、「同期順序回路」という構成方法を利用します。同期順序回路では、独立に動作するハードウェア中のモジュールのデータの入力のタイミングを、共通の信号「クロック」に合わせて制御します。ある単位処理を受けもつハードウェア・モジュールは、クロックが立ち上がりや立ち下がりから一定期間の間で必ず信号を読み取ることにします(図8)。全ての

モジュールが、(1) クロックに間にあうように信号を決定し、(2) 一定の期間値を変更しない、というルールさえ守れば、全体の演算が正しく動作します。

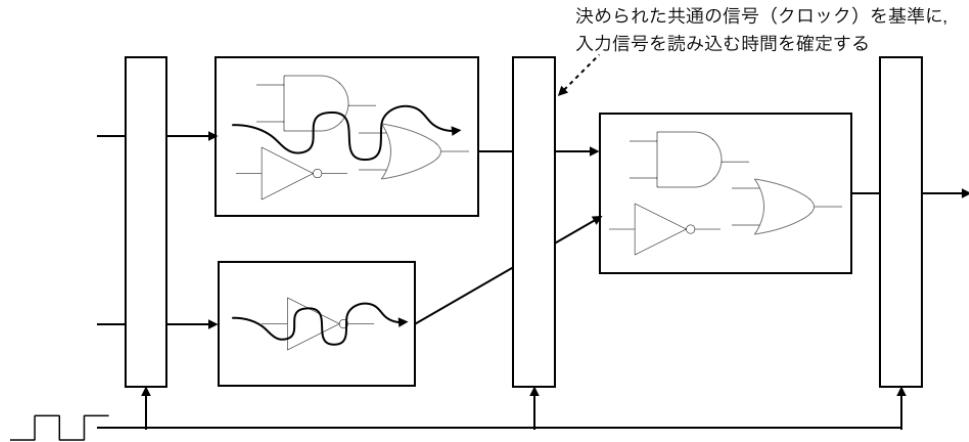


図 8 入力された信号に対して出力が確定するまでの時間は、論理演算回路を構成する論理演算の段数で決まる。

回路全体がクロックに合わせて動作するため、クロックの周期が短いほど全体が高速に動作します。しかし、やみくもにクロックを短くできるというわけではありません。

3 ハードウェア・プログラミングの流れ - 使用する言語と必要なツールは？

ハードウェア・プログラミングの詳細は、以降の章で実例と共に紹介しますが、まずは全体の流れのイメージをつかんでみましょう。

FPGA を使ったハードウェア・プログラミングは、図 9 のような流れで行ないます。流れ自体はソフトウェア・プログラミングとあまり変わりません。

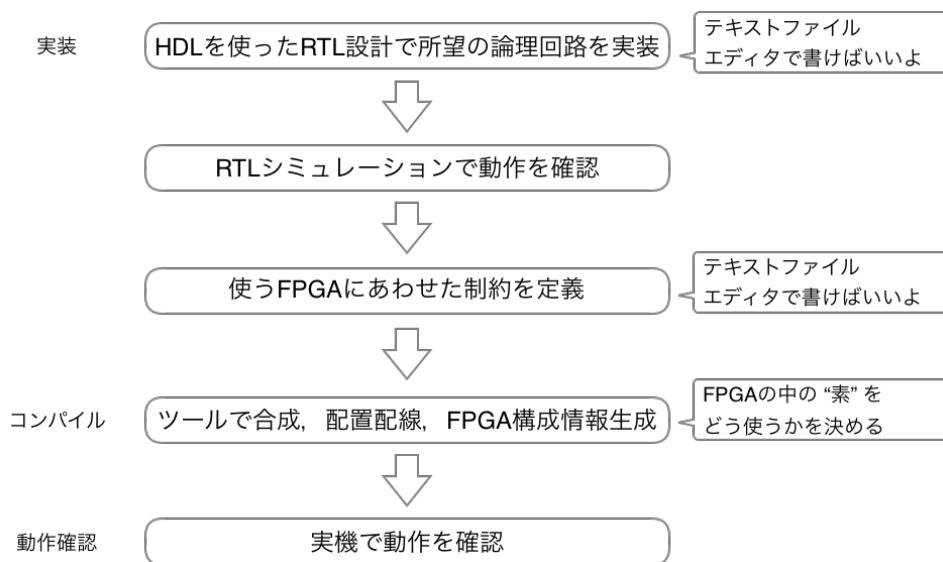


図9 FPGA を使ったハードウェア・プログラミングの流れ。

ハードウェア・プログラミングには、ハードウェア記述言語 (HDL; Hardware Description Language) を使用します。一般的に使用されているのは、VHDL と Verilog HDL の 2 種類です。記述自体は好きなテキストエディタで行なって構いません。図 10 は、emacs を使って VHDL を記述している例です。

図10はemacsでVHDLを記述する画面です。画面内にVHDLのコードが表示されています。右側にオレンジ色の枠で囲まれて「スタイルはソフトウェアプログラミングと類似」という説明文があります。

```

emacs@CIDER
<->[X] 0+ hoge.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity hoge is
port (
    clk : in std_logic;
    reset : in std_logic;
    q : out std_logic
);
end entity;

architecture RTL of hoge is
begin
    signal counter : unsigned(31 downto 0) := (others => '0');

    begin
        q <= counter(4);

        process(clk)
        begin
            if rising_edge(clk) then
                counter <= counter + 1;
            end if;
        end process;
    end RTL;

```

U\--- hoge.vhd All (16,0) [0] (VHDL) 2:37午前 1.25
Wrote c:/Users/miyo/hoge.vhd

図10 emacs を使って VHDL を記述している例。

HDL でハードウェアを設計したら、正しく動作を記述できているか RTL シミュレーションというフローで確認します。コマンドラインベースで使える簡単なシミュレーションツールや、回路の動作の流れを信号の変

化をグラフィカルに表示するツールなどがあります。図 11 は、図 10 で記述したハードウェアをシミュレーションし信号の変化を描画した様子です。

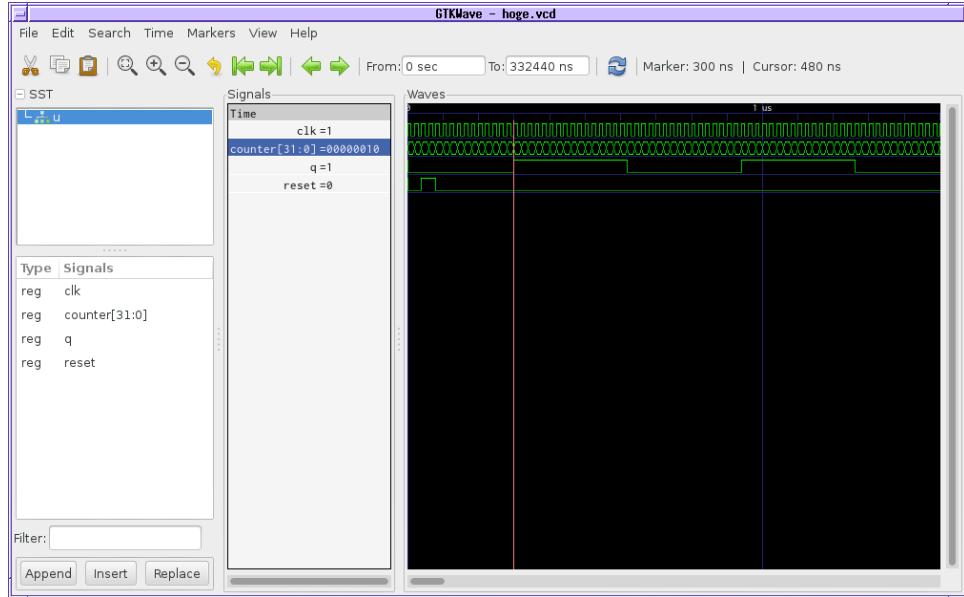


図 11 シミュレーション結果を確認している様子。

動作の確認ができた後、開発ツールの助けを借りてハードウェアに書き込む構成情報を生成します。開発ツールで行う作業は、合成と配置配線、構成情報生成の 3 段階です。必要なツールは、FPGA ベンダである米国の Intel 社と Xilinx 社の両方から、それぞれ専用のものが提供されています。また、最適化処理などに特色を持つサード・パーティ製の開発ツールもあり、必要に応じて使い分けるとよいでしょう。図 12 は、開発ツールの例です。

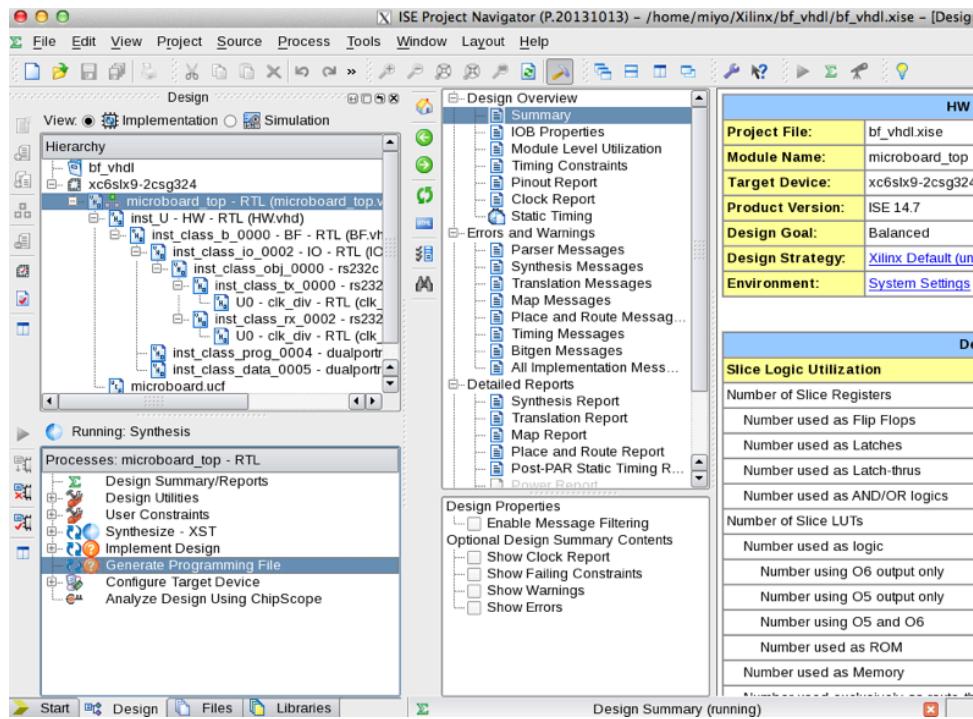


図 12 開発ツールを使って、合成・配置配線・構成情報の生成を行なう。

開発ツールの行う、ハードウェア・プログラミング特有の作業内容は次の通りです。

合成 - 論理回路に置き換える処理 記述された HDL を解釈し、論理回路に置き換えます。また、さまざまな最適化を適用し、小さく高速な回路を自動で生成します。

配置配線 - ロジック・セルの割り当てと接続を決定 合成して生成した論理回路を FPGA 内にあるロジック・セルを組み合わせて実現するための割り当てと接続関係を決定します。ここで、ユーザが「特定の回路同士を近付ける」や、「特定の出力を好きなピンにアサインする」などの設定を与えられます。

構成情報生成 - 書き込み用ファイルを作る 最後に FPGA に書き込み可能な構成情報を生成します。

ソフトウェア・プログラミングであれば、コンパイル、リンク、バイナリ生成に相当するステップだと思えばよいでしょう。

構成情報の生成が完了したら、そのデータを書き込みソフトウェアを使用して FPGA に書き込むことで設計したハードウェアを動かすことができます。

4 さいごに

本章では、オリジナルのハードウェア・デバイスを設計する方法として、ハードウェア記述言語と FPGA を用いた開発のプロセスについて説明しました。特に、ソフトウェア・プログラミングとハードウェア・プログラミングを対比させながらその違いについて説明しました。ハードウェア設計は自由度が高い分、自分で面倒を見なければならない部分も多いのですが、プロセッサの制約に縛られずにアイデアを実現できる可能性を秘めています。FPGA と HDL によるハードウェア設計に興味を持って、はじめの一歩を踏み出してもらえると

嬉しいです。

VHDL/Verilog HDL 入門

代表的なハードウェア記述言語 VHDL と Verilog HDL の基礎概念と文法を学ぼう

1 はじめに

本章では、ハードウェア記述言語 (HDL; Hardware Description Language) のうち、よく使用される VHDL と Verilog HDL の二つの HDL の基本文法を説明します。ちょっとした違いを発見しながら読み進めると面白いでしょう。

ソフトウェア・プログラミングで使用する C や Java、コミュニケーションで使用する英語についても、正しく使用するために文法の知識は欠かせません。同じように HDL で設計する際も文法の知識が必要です。ここで、基本文法をしっかりと押さえましょう。

2 ハードウェア記述言語の基本概念

プログラミング言語に多くの種類があるように、ハードウェア記述言語 (HDL) にもさまざまな種類があります。その中でもよく利用されるのが、VHDL と Verilog HDL です。VHDL と Verilog HDL は、どちらも、ハードウェアを表現するための似たような概念を取り扱うことができる言語です。

ただし、似たような概念でもそれぞれの言語で使用する言葉が違うので注意が必要です。両方の言語に共通する概念と、言語の特徴について説明します。

2.1 構造の基本 — エンティティ/モジュール

どの言語にも基本的な構造があります。たとえば、C では関数、Java ではクラスなどです。HDL では、与えられた入力に対して出力を生成するブロックが基本的な単位です(図 1)。このブロックを VHDL ではエンティティ (entity), Verilog HDL ではモジュール (module) と呼びます。ただし、この章では、特に VHDL や Verilog HDL に違いがない説明では、モジュールと呼ぶことにします。



図 1 ハードウェア・プログラミングの基本的な単位

通常のプログラミング言語と HDL の大きな違いは、エンティティ/モジュールは、最初から最後まで与えられた入力に対する出力を生成し続けるということです。C などで関数を呼び出す場合、main プログラムからその関数内へ処理が移ります(2)。処理を終えると戻り値を呼び出し元に返し、main プログラムが再び動き始めます。つまり、main プログラムは、呼び出した関数の処理が完了するまで待たれます。これは、プログラム・カウンタが、プログラムを順々に呼び出して実行するからです。

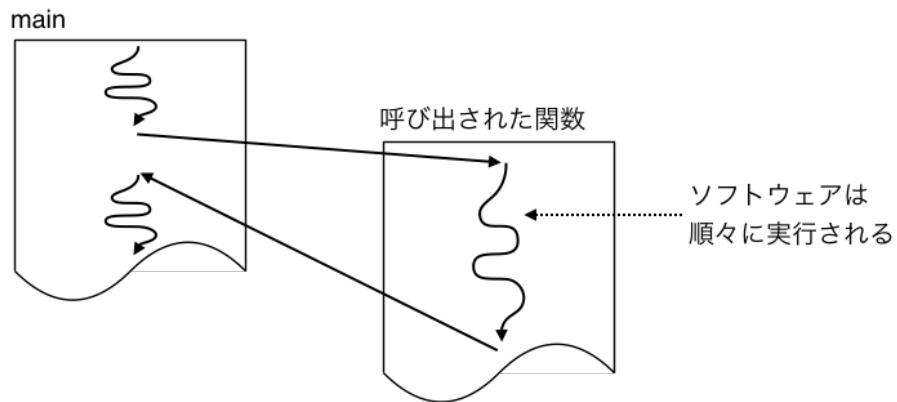


図 2 C で記述した一般的なソフトウェア・プログラムの実行の様子

一方、HDL で記述されたエンティティ/モジュールには、共通のプログラム・カウンタのような、複数の演算回路の動作を制御する仕組みはありません。(図 3)。どのモジュールも常に存在し、独立して動作します。したがって、特定の入力を与えると出力を返すよりも、入力されているデータに対して出力するデータを作り続けているというイメージになります。複数のモジュール間で制御が必要であれば自分で、そのように設計する必要があります。

複数の演算回路の動作の制御はなく、
回路は常に存在し、独立に動作する

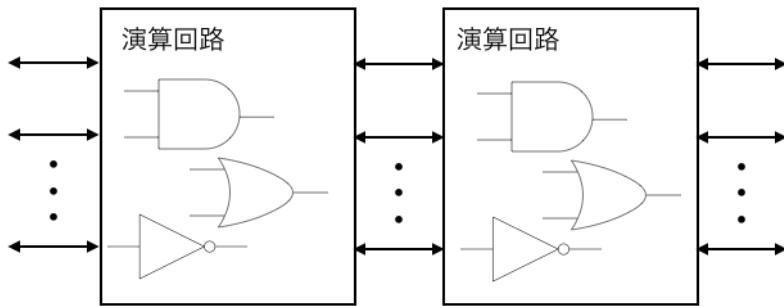


図 3 ハードウェアは常に存在し、演算回路の動作が制御されることはない（制御が必要なら自分で記述する必要がある）

2.2 2 種類の基本処理方法 — 同時処理文と順次処理文

繰り返しになりますが、ハードウェア・プログラミングでは、独立して動作するモジュールを扱う必要があります。すなわちハードウェアを記述するための言語では、独立して動作する同時並行的な処理を記述できる必要があります。とはいえ、実現したい処理によっては、条件分岐のような依存関係のある処理の記述が望まれます。これらの要求を満たすため、VHDL と Verilog HDL のどちらも、同時処理文と順次処理文と呼ばれる二種類の記述方式をサポートしています。具体的な記述方法は後で説明しますが、それぞれの考え方を頭に入れておいてください。

■同時処理文 同時処理文とは、周りの処理に依存せず独立して動作する処理です。複数の同時処理文は、ある特定の時点で一斉に処理されます。そのため、記述順や各処理文の間には、構文的な順序が存在せず、「ある時点」で入力された値に従って出力が生成されます。出力が確定するまでの時間は、物理的にデバイスの中を電気が流れる速さや信号遅延に依存します。

■順次処理文 順次処理文は、複数の処理同士に構文などによって順序が規定された処理です。たとえば、ソフトウェアには欠かせない分岐などの制御文の表現には順序が必要になります。

2.3 使用できる変数 — 数値と信号

プログラミング言語と同じように HDL でも変数を利用できます。VHDL でも Verilog HDL でも、変数はすべて型を持ちます。ハードウェアとして、基本的な型は 1 本の信号線です。また、信号線を束ねた配列もサポートされます。このほかに、整数や自分で定義した型も利用できます。

変数は、英数字からなる名前を付けることができます。変数名の先頭は英字または「_」で始める必要があります、末尾を「_」にしてはいけません。Verilog HDL では、大文字と小文字は区別されます。

2.4 演算の基本 — 算術/論理減算、比較、代入

VHDL および Verilog HDL では加減算や論理演算、比較などの演算子を利用することができます。ソフトウェア・プログラミングの場合は演算子を使って記述された処理はプロセッサに与える命令に変換されます

が、HDL の場合は、その演算に相当するハードウェア・ロジックとして LUT や FF などの組み合わせに合成されます。FPGA の中には、小さなデジタルシグナルプロセッサや乗算器を持つものがあり、条件にうまく合致すると、それらが使用されます。

HDL でもソフトウェア・プログラミング同様に、演算した結果を代入演算を利用して、ほかの（あるいは同じ）変数に代入することができます。HDL の代入には、ブロッキング代入とノンブロッキング代入の 2 種類があります。ブロッキング代入は、その時点で値を代入して次に進む代入です。一方、ノンブロッキング代入は、複数の代入文において、それらの代入の同時実行を規定します。C などで記述した単一スレッドのソフトウェア・プログラムの代入は、HDL でいうところのブロッキング代入に相当します。

2.5 値の基本 — '0', '1', 'Z', 'X'

ハードウェアの値は'0' と'1' の値をとります。加えて、ハードウェアにはハイ・インピーダンスという、「抵抗が無限大」を意味する状態が存在します。VHDL や Verilog HDL では'Z' で表されます。値として「抵抗が無限大」というのは、少しあまりにくいかかもしれません。物理的には、図 4 のようにスイッチを切った状態をイメージしてください。複数の信号が一つにまとめられるとき、'Z' は、「ほかの値に影響を与えない」ということを意味します。

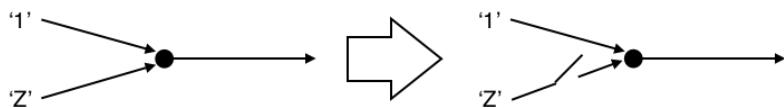


図 4 HDL ではハイ・インピーダンスでスイッチオフを記述できる

また、'0' でも'1' のどちらでもいい値として不定値という概念があります。これは、'X' と表現されます。

ソフトウェア・プログラミングでは、通常、'0' と'1' の 2 値をとる値をビットと呼びますが、'Z' と'X' も加えた 4 つの値をとる信号が便宜上ビットと呼ばれることがあります。

2.6 文末には「;」を付ける

VHDL も Verilog HDL も、演算処理や変数定義などの文の終わりには「;」(セミコロン) を付けます。ただし、両言語ともソフトウェアのプログラミング言語では少し首をかしげてしまうよう、「;」を付けないケースが存在するので注意が必要です。

3 VHDL の基本文法のルール

VHDL の基本的な文法を説明します。

3.1 コメント

多くのソフトウェア・プログラミング言語と同様に、VHDL でもソースコード中にコメントを書くことができます。VHDL では、「--」から行末までがコメントになります。

3.2 モジュールの構成

図 5 に、VHDL で記述するモジュールの概要を示します。VHDL では、対象とするモジュールを大きく entity と architecture に分けて記述します。entity には外部に接続される入出力ポートの宣言などの回路の外枠を、architecture には使用する関数の定義や処理内容など、回路の内部を定義します。

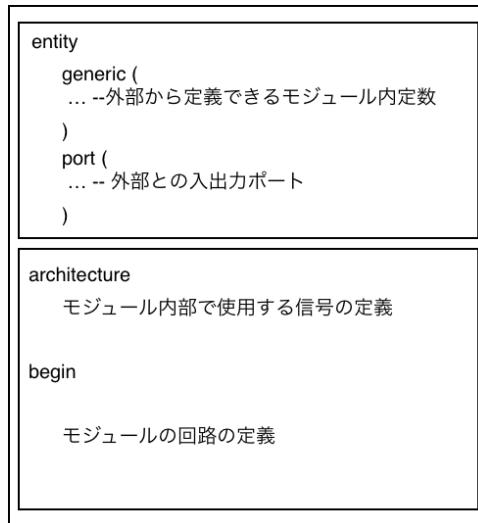


図 5 VHDL のモジュール定義は entity と architecture から構成される

3.3 即値の表現方法

VHDL では、ソース・コードの中に定数の値を記述できます（表 1）。複数の信号線を束ねた値は、信号線の本数分だけ各信号に相当する値を並べて表現します（たとえば 4bit なら "0000" など）。また、「X"01"」と記述することで 16 進数で数を表記できます。よく用いられる代表的な定数表現には、表 1 のようなものがあります。

説明	値の例
1 本の信号線がとる信号の値	'1', '0', 'Z', 'X'
複数の信号線がとる信号の値	"111", "0100", X"10"
整数	32, 8, 1000
真偽値	true, false

表 1 VHDL で記述できる定数の例

3.4 型

VHDL の変数はすべて型を持ちます。たくさんの型が定義されており、また、独自の型も定義できます。よく使用される 5 つの型を表 2 に示します。基本的には、`std_logic`, `std_logic_vector` はハードウェアの信号に相当する型、`signed` や `unsigned` は加減算などの算術演算ができる値を表現するために用いる型、一般的な数値を表現できる `integer` です。

型名	説明
<code>std_logic</code>	1bit の信号線
<code>std_logic_vector(n-1 downto 0)</code>	n-bit の信号線
<code>unsigned(n-1 downto 0)</code>	n-bit の符号なしの算術演算可能な値
<code>signed(n-1 downto 0)</code>	n-bit の符号ありの算術演算可能な値
<code>integer n to m</code>	n から m までの整数

表 2 VHDL で用いられる型の例

3.4.1 1-bit の信号

`std_logic` は、VHDL の基本となる 1bit の信号に相当する型です。`'0'`, `'1'` のほかに、ハイ・インピーダンスを示す`'Z'`、不定値を示す`'X'` を値としてとれます。これらの値は、ハードウェアにそのまま対応します。

3.4.2 n-bit の信号

`std_logic_vector(n downto 0)` は、`std_logic` が n 個並んだ n-bit の信号線に相当する型です。「n-bit の `std_logic_vector` 型」と呼びます。`std_logic_vector` 型の変数 a の中の要素を、`a(3)`, `a(4 downto 2)` などとして取り出せます。前者は `std_logic` 型、後者は 3bit の `std_logic_vector` 型です。「downto」は `std_logic` の並びに、MSB から降順で番号を付けることを意味します。つまり、`std_logic_vector(n-1 downto 0)` のビット列の場合、MSB が `std_logic_vector(n-1)` で、LSB が `std_logic_vector(0)` です。`to` を使うことで逆順に並べることもできます。その場合は、`std_logic_vector(n upto 0)` のように書きます。

3.5 モジュールの外枠の記述 — entity

`entity` は、モジュールの外枠に相当し、モジュールの名前と入出力の信号で定義されます。たとえば、次の記述は、入力信号に `pClk` と `pReset`、出力信号に `Q` を持つ `test` という名前のモジュールの外枠の定義に相当します。

```

1 entity test is
2 port (
3     pClk    : in std_logic;
4     Q       : out std_logic;
5     pReset : in std_logic -- 最後の一つの後には";"をつけない
6 );
7 end entity;

```

3.5.1 ポートを定義する

ポートは、ハードウェア・モジュールの入出力です。entityの中の port(~);の中に信号を方向と型を指定して定義します。信号の方向には in(入力) と out(出力), inout(入出力) の 3 種類があります。各ポートは「名前 : 方向 型」で定義されます。たとえば、

```
1 pClk : in std_logic
```

という記述は、pClk という名前の型が std_logic の入力ポート (in) の定義に相当します。同じ方向、型の複数のポート名は「,」で並べて定義することもできます。たとえば、

```
1 pR, pG, pB : in std_logic
```

として、3つの入力信号 pR, pG, pB をまとめて定義できます。

3.5.2 定数を定義する

entityの中にモジュールの中で使用する定数を定義できます。

```

1 entity test is
2 generic (
3     width  : integer := 640;
4     height : integer := 480
5 );
6 port (
7     pClk : in std_logic;
8     Q   : out std_logic;
9     pReset : in std_logic
10 );
11 end test;

```

ここでは、width という名前で値が 640 の integer 型、すなわち整数の定数を定義しています。この定数は entity 内部、および内部処理を記述する architecture の中で使用できます。

3.6 内部処理の記述 — architecture

VHDL では、`architecture` にモジュールの処理内容を記述します。記述の基本的な流れは次の通りです。

```
1  architecture RTL of test is
2      (ここに変数の定義などを書く)
3  begin
4      (ここに処理内容を記述する)
5  end RTL;
```

上記は、`test` という名前のモジュールの中身を記述するためのブロックです。

3.7 変数の定義

`architecture` 内部で使用する変数として、変数を `signal` というキーワードを使って定義できます。初期値は省略可能です。

```
1  signal 名前 : 型 := 初期値;
```

たとえば、10 ビットの配列は下記のように定義します。

```
1  signal counter : std_logic_vector(9 downto 0);
```

また、`generic` で定義した値である `width` を使用して、幅 `width` の `std_logic_vector` 型の信号を次のように定義できます。ここで「`width - 1`」の値は合成時に決定されます。

```
1  signal counter : std_logic_vector(width-1 downto 0);
```

3.8 演算子と演算

代表的な演算子を表 3 にまとめます。論理演算子を `std_logic_vector` 型に定義した場合は、対応する各ビットの値同士に論理演算を適用した結果を返します。たとえば、「"10" and "11"」は「"10"」になり、「"10" or "11"」は「"11"」になります。比較演算の結果は、`true` か `false` の真偽値になります。よくあるプログラミング言語に備わっている演算が備わっていることがわかります。ただし、表 3 の説明に (*1) をついている、算術演算や数値の大小を比較する演算では `unsigned` 型、`signed` 型あるいは `integer` の変数や定数、あるいは数値に相当する即値にしか利用できません。

種類	演算子	説明
論理演算	<code>a and b</code>	論理積. <code>a</code> と <code>b</code> が'1' なら'1'. さもなければ'0'.
	<code>a or b</code>	論理和. <code>a</code> と <code>b</code> のどちらか又は両方が'1' なら'1'. さもなければ'0'.
	<code>a xor b</code>	排他的論理積. <code>a</code> と <code>b</code> の一方だけが'1' なら'1'. さもなければ'0'.
	<code>not a</code>	否定. <code>a</code> が'0' なら'1'. '1' なら verb—'0'—
比較演算	<code>a = b</code>	<code>a</code> と <code>b</code> が等しい場合 <code>true</code> . さもなければ <code>false</code>
	<code>a /= b</code>	<code>a</code> と <code>b</code> が等しくなければ場合 <code>true</code> . さもなければ <code>false</code>
	<code>a > b</code>	<code>a</code> が <code>b</code> がより大きいなら <code>true</code> . さもなければ <code>false</code> . (*1)
	<code>a < b</code>	<code>a</code> と <code>b</code> がより小さいなら <code>true</code> . さもなければ <code>false</code> . (*1)
	<code>a >= b</code>	<code>a</code> と <code>b</code> 以上なら <code>true</code> . さもなければ <code>false</code> . (*1)
	<code>a <= b</code>	<code>a</code> と <code>b</code> 以下なら <code>true</code> . さもなければ <code>false</code> . (*1)
算術演算	<code>a + b</code>	<code>a</code> と <code>b</code> の足し算 (*1)
	<code>a - b</code>	<code>a</code> と <code>b</code> の引き算 (*1)
	<code>a * b</code>	<code>a</code> と <code>b</code> の引き算 (*1)
	<code>a / b</code>	<code>a</code> と <code>b</code> の割り算 (*1)
	<code>a ** b</code>	<code>a</code> の <code>b</code> 乗 (*1)
配列操作	<code>a & b</code>	<code>a</code> と <code>b</code> をこの順に並べた信号線の束を作る
	<code>a(b)</code>	<code>a</code> の <code>b</code> 番目の信号を取り出す
	<code>a(b downto c)</code>	<code>a</code> の <code>b</code> 番目から <code>c</code> 番目の信号線の束を取り出す

表 3 VHDL で用いられる演算の例

3.8.1 演算結果の代入

演算の結果は代入文でほかの（あるいは同じ）変数へ代入できます。代入にはブロックング代入とノンブロックング代入があります。

```

1  := ブロックング代入
2  <= ノンブロックング代入

```

たとえば、

```

1  Q <= counter(width-1);

```

という記述は、`std_logic_vector` 型の変数 `counter` の `(width-1)` 番目を取り出し、`Q` に代入するハードウェアの記述に相当します。VHDL では、`signal` 変数には、初期化時以外でブロックング代入を使用することはできません。

3.8.2 型の変換

VHDL は型の制約が強い言語で、代入は同じ型の変数同士でしか認められません。また演算子も適用可能な型があらかじめ決められています。そのため、幅の違う `std_logic_vector` 同士で値を定義する場合には、ビット幅を削る/足すなどして同じ幅にしなければいけません。たとえば、`a` と `b` が、それぞれ幅 16-bit, 8-bit の `std_logic_vector` 型であれば、

```
1 a <= "00000000" & b; -- 足りない 8bit を 8bit の 0("00000000") で埋めている
2 b <= a(7 downto 0); -- a の下位 8bitだけを b に代入している
```

などとする必要があります。

型の変換には専用の関数を利用します。たとえば、`std_logic_vector` を `unsigned` 型あるいは `signed` 型に変換するためには、それぞれ `unsigned` 関数あるいは `signed` を用います。

```
1 unsigned(c);
```

と記述すると `std_logic_vector` 型の変数 `c` を `unsigned` 型に変換できます。

逆に、`unsigned` 型や `signed` 型の変数を `std_logic_vector` 型に変換する場合には、`std_logic_vector` 関数を用います。

```
1 std_logic_vector(d);
```

と記述すると `unsigned` 型の変数 `d` を `std_logic_vector` 型に変換できます。

`integer` 型の変数を `unsigned` 型や `signed` 型に変換する場合には、`to_unsigned` あるいは `to_signed` を用います。たとえば、`integer` 型の変数 `k` を `n`-bit の `unsigned` 型に変換する場合は、2 番目の引数にビット数 `n` を指定して、

```
1 to_unsigned(k, n);
```

と記述します。

一般に、VHDL では、`std_logic_vector` 型の変数に対して算術演算は記述できません。そのため、`std_logic_vector` と定数の加減算や比較演算する場合には、一度 `unsigned` 型に変換して演算する必要があります。たとえば、幅 `n` の `std_logic_vector` の変数 `counter` に定数 1 を加算する場合には、次のように、一度 `unsigned` 型に変換して演算した後で `std_logic_vector` 型に戻す必要があります。

```
1 counter <= std_logic_vector(unsigned(counter) + 1);
```

3.8.3 シフト演算

VHDL には、ソフトウェア・プログラミング言語で一般的なシフト演算子に相当する演算子がありません。VHDL では配列操作の演算を用いて似たような操作ができます。たとえば、幅 n-bit の std_logic_vector 型の変数 counter を右に 1 つシフトしたい場合には、次のように記述します。

```
1 counter <= '0' & counter(n-1 downto 1);
```

左に 2 つシフトしたい場合には、

```
1 counter <= counter(n-3 downto 0) & "00";
```

となります。

3.9 同時処理文

VHDL では単に並べて記述された文は、同時処理文として解釈されます。それらは合成ツールにより並列に解析され、回路に合成されます。すなわち、記述された内容はその順序に依存しません。たとえば、

```
1 c <= a and b;
2 e <= c and d;
```

と書いても、

```
1 e <= c and d;
2 c <= a and b;
```

と書いても、同じように図 6 の回路が合成されます。

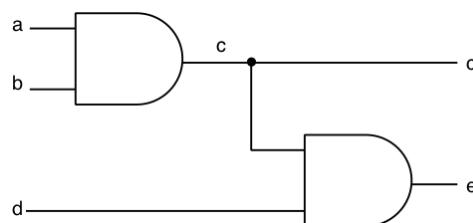


図 6 同時処理文は、記述順によらず解析・合成される

3.10 順次処理文 — process 文

順次処理文では記述された順序に従って意味が解析され、回路が合成されます。そのため、複雑な制御構文を使用できます。VHDLでは、`architecture` 中で `process` を使って順次処理文を記述するためのブロックを作ることができます。`process` 文の基本的な構文を次に示します。

```

1 process(a, b)
2 begin
3   c <= a and b;
4   d <= a or b;
5 end process;
```

ここで、`process()` の「`()`」内の変数のリストをセンシティビティ・リストといいます。このリストに列挙した変数の信号が変化すると `process` の中の回路が動作し出力値が変更されます。ノンブロッキング代入は、`process` 内の記述が順に解釈された後で、同時に信号が確定します。この `process` 文は、図 7 のような回路を生成します。あくまで文が順に解釈されるだけで、順に処理される回路が生成できるわけではない、ことに注意する必要があります。

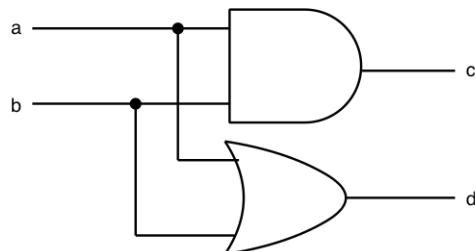


図 7 順次処理文中の複数のノンブロッキング文は順に解釈され、最後に値が同時に確定する

`process` 文の中にでてくる入力変数(式の右辺にでてくる変数)がすべてセンシティビティ・リストに列挙されている場合、入力が変化する度に回路が動作し、出力変数(式の左辺)の値が変更されます。つまり、その `process` 文から生成される回路では、何も状態を保存する必要がありません。そのため、図 7 のような記憶素子を必要としない組み合わせ回路として構成されます。

センシティビティ・リストにない変数が右辺に使われる。次のような `process` 文を考えてみます。

```

1 process(a)
2 begin
3   c <= a and b;
4   d <= a or b;
5 end process;
```

ここでは、センシティビティ・リストに「`b`」がなく、入力変数が全部列挙されていません。この回路では、`b` の値が変化しても出力先である `c` と `d` の値は変化しません。つまり、図 7 のように、入力である `a` を出

力の c と d に直接接続することができず、 c と d の値を保存する機構、記憶素子が必要となり、組み合わせ回路として合成されません。「同じように記述したつもりでも違う回路になるかもしれない」ということを覚えておいてください。

VHDL の `process` 文では、ブロッキング代入可能な `variable` 変数を利用できます。`variable` 変数は、順次処理文において便宜的に一時的な値を格納しておくためのものです。長く複雑な演算を行う場合に、ソース・コードの見通しをよくできます。たとえば、次のように使います。

```

1 process(a,b)
2     variable tmp0 : std_logic;
3     variable tmp1 : std_logic;
4 begin
5     tmp0 := a and b;
6     tmp1 := a or b;
7     c <= tmp0 xor tmp1;
8 end process;
```

ここでは、 tmp0 と tmp1 が `variable` 変数です。ブロッキング代入における演算の結果がそれぞれ代入されています。ブロッキング代入のため、合成ツールがこの構文に出会ったところで、 tmp0 の値は $(a \text{ and } b)$ に、 tmp1 の値は $(a \text{ or } b)$ にすぐさま置き換わります。つまり、これは、 $c \leq (a \text{ and } b) \text{ xor } (a \text{ or } b)$ という回路として合成されます。

3.11 制御構文

VHDL では、まるでソフトウェアを記述するように、条件分岐や C の `switch` 文のような制御構文が使えます。多くの制御構文は、`process` 文の中でのみ使用ができます。代表的なものを紹介します。

3.11.1 when ~ else

同時処理文中で記述可能な制御構文です。条件に従って出力する値を選択できます。下記は、 a と b の値が等しい場合は X を、等しくない場合には Y を c に代入する例です。

```
1   c <= X when a = b else Y;
```

制御構文というよりは、C などで、 $c = (a==b) ? X : Y$ と書く 3 項演算子に近いイメージですね。

3.11.2 if ~ then ~ elsif ~ else ~ end if — 条件分岐構文

`process` 文の中でのみ使用できる条件分岐構文です。次は、 a より b が大きい場合は処理文 X が、それ以外の場合は処理文 Y が有効になります。また、処理文の中で `if` をネスト(入れ子に)することもできます。

```

1 if a > b then
2   処理文 X
3 else
4   処理文 Y
5 end if;

```

また、`elsif` を使うと、`else` 節に重ねて次の条件を記述できます。

```

1 if a > b then
2   処理文 X
3 elsif a < b then
4   処理文 Y
5 else
6   処理文 Z
7 end if;

```

VHDL では、単に信号の値に対する条件だけでなく、信号が変化するタイミングを使用した条件式が書けます。たとえば、変数 `clk` が変化するタイミングは `clk'event` と書きますので、これを使って、

```

1 if clk'event and clk = '1' then
2   処理文
3 end if;

```

のような条件文を作ることができます。この例では、「`clk` が変化し、かつ、`clk` が '1' のときに処理文が実行されます。ハードウェアとしては、`clk` 信号が立ち上がった瞬間に相当します。それ意外のタイミングでは、処理文は動作せず、値が保存し続けられます。これは、決まったタイミングで処理を実行する順序同期回路の設計に欠かせない表現です。

ただし、2018 年現在では、多くの場合、直接「クロックの立ち上がり」を表す、`rising_edge` という関数を使って、

```

1 if rising_edge(clk) then
2   処理文
3 end if;

```

という記述が好まれます。

3.11.3 case～when — 選択

C や Java でいうところの `switch` 構文です。次に示す例では、`std_logic_vector` 型の変数 `a` の値によって処理文 X, Y, Z のどれかが実行されます。0, 1, 2 という整数と一致させられるように、`to_integer` を使って `a` を `integer` 型に変換しています。ここで、VHDL の `case ~ when` 構文では、どれかの `when` に必

ず該当するように記述する必要があることに注意しなければいけません。すべての `when` を列挙する代わりに「`others`」で残りすべての条件にマッチする場合を表現できます。

```
1 case to_integer(unsigned(a))
2   when 0 =>
3     処理文 X
4   when 1 =>
5     処理文 Y
6   when 2 =>
7     処理文 Z
8   when others => --そのほかのすべての場合
9     処理文 W
10 end case;
```

3.11.4 for ~ in ~ loop — 繰り返し

VHDL における繰り返し処理は、単に似たような処理を繰り返して記述する代わりに簡単に書けるようにするための構文です。実際には、合成時に繰り返し回数分のハードウェア回路が生成されます。

たとえば、下記に示す処理は、5bit の `std_logic_vector` 型の変数 `a` と `std_logic` 型の変数 `b` が定義されているときに、

```
1 process
2   variable i : integer := 0;
3   variable tmp : std_logic;
4 begin
5   tmp := '0';
6   for i in 0 to 4 loop
7     tmp := tmp or a(i);
8   end loop;
9   b <= tmp
10 end process;
```

という記述は、次のような記述を簡単に記述したことに相当します。

```

1 process
2     variable i : integer := 0;
3     variable tmp : std_logic;
4 begin
5     tmp := '0'
6     tmp := tmp or a(0);
7     tmp := tmp or a(1);
8     tmp := tmp or a(2);
9     tmp := tmp or a(3);
10    tmp := tmp or a(4);
11    b <= tmp
12 end process;

```

作成した回路が動作する時ではなく、回路を作成する時点での繰り返し処理が解釈されることに注意してください。

3.12 組み合わせ回路のサブモジュール

VHDL では、複雑な組み合わせ回路を生成するために、`function` というサブモジュールを記述できます。`function` は順次処理文で、`if` や `case` などの条件文を記述できますが、ノンブロッキング代入を用いることはできません。`function` は、次のように定義します。

```

1 function f (a : in std_logic; b : in std_logic)
2     return std_logic is
3     variable Q : std_logic;
4 begin
5     if (a = b) then
6         Q := '1';
7     else
8         Q := '0';
9     end if;
10    return Q;
11 end f;

```

これは、1 ビットの変数である `a` と `b` を入力とする名前 `f` の `function` の定義です。入力された二つの値が等しいときに 1 を、異なるときに 0 を出力する関数です。

呼び出し側では、

```

1 x <= f(x, y)

```

などとします。`x` と `y` は、関数呼び出しに対する実引数になります。

また次のように、配列型の変数を入力あるいは出力する関数を定義できます。

```
1  function g (x : in std_logic_vector(1 downto 0))
2      return std_logic_vector is
3          variable Q : std_logic_vector(1 downto 0);
4      begin
5          Q(0) := x(1);
6          Q(1) := x(0);
7          return Q;
8      end g;
```

Cなどのソフトウェア・プログラミング言語の関数とは違い、関数の計算が終了するまで呼び出し側の処理が待たされるということはありません。複雑な組み合わせ回路を見通しよく記述できる書き方です。

3.13 おまじない

実は、ここまで説明してきた `std_logic`などを利用するには、これらの機能が実装されたライブラリなどを読み込む必要があります。VHDL ソース・コードの先頭に以下を記述します。

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
```

4 Verilog HDL の基本文法のルール

Verilog HDL の基本的な文法を説明します。

4.1 コメント

多くのソフトウェア・プログラミング言語と同様に、Verilog HDL でもソースコード中にコメントを書くことができます。Verilog HDL では、C++ と同じように `/* ~ */` で囲んだ部分や `//` から行末までがコメントになります。

4.2 モジュールの構成

図 8 に、Verilog HDL で記述するモジュールの概要を示します。VHDL では、外枠の定義 `entity` と内部の定義 `architecture` が区別されていたのに対し、Verilog HDL にはそのような区別はありません。`module` の中に、外部と接続されるポートや関数内で使用する変数の宣言、処理内容などを記述します。

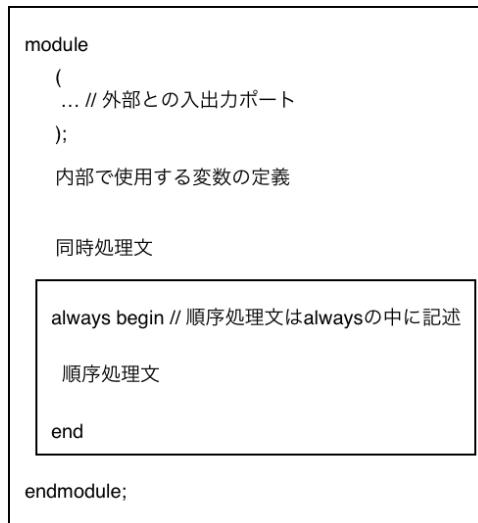


図 8 Verilog HDL のモジュール定義の概要

4.3 値の表現方法

Verilog HDL では、必要とするビット幅 (w) と基數 (f) を付けて「 $w'f$ 値」という形式で即値を記述します。ビット幅は 10 進数で記述します。ビット幅と基數を省略すると 10 進数、32bit の値になります。表 4 に、基數と記号と表現される数の関係を示します。

基數 (w)	基數記号 (f)	例	10 進数表記での値
2	b	8'b10	2
10	d	10'd10	10
		10	10
16	h	8'h10	16

表 4 Verilog HDL における値の表記例

4.4 型 — ネット型変数とレジスタ変数

Verilog HDL の変数には、ネット変数とレジスタ変数があります。どちらも 1bit の信号に相当する変数と、複数 bit の信号を束にした配列変数を作れます。ネット変数とレジスタ変数の違いは、ハードウェアに則したワイヤ (配線) とレジスタ (記憶素子) を想起させるものですが、後で説明するように使える場面と使い方に違いがあります。表 5 に、変数の型を示します。

型名	説明
<code>wire</code>	1-bit のネット変数
<code>wire[n-1:0]</code>	n-bit のネット変数
<code>reg</code>	n-bit のレジスタ変数
<code>reg[n-1:0]</code>	n-bit のレジスタ変数

表 5 Verilog HDL における型の例

4.4.1 ネット変数 — `wire`

モジュールやゲート同士を接続する配線に名前を付けた変数が、ネット変数 `wire` です。ネット変数自身は値を保持することはできず、他から代入された値を次に伝達する役目を担う変数です。ハードウェアの配線そのものに相当します。

4.4.2 レジスタ変数 — `reg`

レジスタ変数 `reg` は、値を保存する記憶素子になることができ、変数自身が値を保持できます。組み合わせ回路も順序回路も構成できます。

4.4.3 配列変数

配列変数 `wire[n-1:0]`, `reg[n-1:0]` は、それぞれ `wire` あるいは `reg` からなる n-bit の変数に相当します。幅 n のネット変数

```

1   wire[n-1:0] a;
2   reg[n-1:0] b;
```

と定義された変数 a の各要素を `a[0]`, `a[2:0]` などとして取り出すことができます。前者は `wire`, 後者は `wire[2:0]` の変数です。

4.5 モジュールの外枠の記述 — `module`

Verilog HDL では、`module` でハードウェア・モジュールの外枠を定義します。モジュールの名前と入出力の信号名を定義します。

```

1 module test(pClk, pReset, Q);
```

この文では、`pClk`, `pReset`, `Q` という名前の入出力信号を持つ `test` という名前のモジュールを定義しています。Verilog HDL では、この `module` 文から

```

1 endmodule
```

というキーワードまでがモジュールの定義に相当します。module の末尾には ‘;’ (セミコロン) が必要ですが、endmodule には付けないことに注意してください。ポート名だけを列挙する場合には、後で、各ポートの入出力方向と幅を定義する必要があります。

少し前の Verilog-95

```
1 module test(pClk, pReset, Q);
```

4.5.1 ポートの定義

ポートとは、モジュールの入出力信号のことです。ポートの名前は、モジュール名に続く () の中に記述します。それぞれのポートの入出力は、モジュールの中で、「方向型 変数名;」で宣言します。方向には、input(入力) と output(出力), inout(入出力) の 3 種類があります。module 文の () 内に与えた信号線の型と入出力方向はモジュールの内部で定義します。たとえば、pClk と pReset が 1bit の入力、Q が 1bit の出力信号であれば、

```
1 module test(pClk, pReset, Q);
2   input wire pClk;
3   input wire pReset;
4   output wire Q;
5
6   // 以降、モジュールの内部処理を記述する
7
8 endmodule
```

と記述されます。

通常 Verilog HDL では、1bit の wire 変数は定義せずに使えるため、wire が省略されることがあります。もちろん、n-bit のポートを定義することもできます。Q が幅 n-bit の出力ポートであれば、

```
1 output wire[n-1:0] Q
```

と書けます。

Verilog-2001 以降をサポートする処理系の場合には、VHDL のように信号名の定義に型と変数を付けることができます。

```
1 module test(
2     input wire pClk,
3     input wire pReset,
4     output wire Q
5 );
6
7     // 以降、モジュールの内部処理を記述する
8
9 endmodule
```

と記述できます。

4.5.2 定数を定義する

module宣言の後に、各モジュール内で有効な定数をparameter文で定義できます。

```
1 module test(pClk, pReset, Q);
2     parameter width = 640;
3     parameter height = 480;
```

上記に示したモジュールtestの中では、widthを640という値として利用できます。

4.6 内部処理の記述

ポートの宣言に引き続いで、内部で使用する変数の宣言および処理本体を記述します。

4.7 変数の定義

処理に必要な変数をあらかじめ定義しておく必要があります。変数は、ネット接続型あるいはレジスタ型の型を伴って定義されます。

```
1 wire a;
2 reg[15:0] b;
```

これは、ネット変数aと幅16ビットのレジスタ変数bを定義しています。

また、parameterで定義した定数widthを用いて、次のように変数を定義することもできます。

```
1 reg[width-1:0] c;
```

4.8 演算子と演算

代表的な演算子を表 6 にまとめました。論理演算における真と偽は、ネット変数やレジスタ変数の場合は'1' と'0' に対応します。n-bit の配列変数 `wire[n-1:0]` あるいは `reg[n-1:0]` の場合には、配列中に一つでも'1' である要素があれば真、さもなければ偽として判定されます。ビット論理演算子を配列変数に適用する場合は、対応する各要素同士について演算が適用されます。

種類	演算子	説明
論理演算	<code>a && b</code>	論理積。a と b が共に真なら真。さもなければ偽。
	<code>a — b—</code>	論理和。a と b のどちらか又は両方が真なら真。さもなければ偽。
	<code>!a</code>	否定。a が真なら偽。偽なら真
ビット論理演算	<code>a & b</code>	論理積。a と b が共に'1'なら'1'。さもなければ'0'。
	<code>a b—</code>	論理和。a と b のどちらか又は両方が'1'なら'1'。さもなければ'0'。
	<code>a ^ b</code>	排他的論理和。a と b のどちらか一方が'1'なら'1'。さもなければ'0'。
	<code>~a</code>	論理否定。a が'1'なら'0'。さもなければ'1'。
比較演算	<code>a == b</code>	a と b が等しい場合 <code>true</code> 。さもなければ <code>false</code>
	<code>a != b</code>	a と b が等しくなければ場合 <code>true</code> 。さもなければ <code>false</code>
	<code>a > b</code>	a が b がより大きいなら <code>true</code> 。さもなければ <code>false</code> 。(*1)
	<code>a < b</code>	a と b がより小さいなら <code>true</code> 。さもなければ <code>false</code> 。(*1)
	<code>a >= b</code>	a と b 以上なら <code>true</code> 。さもなければ <code>false</code> 。(*1)
	<code>a <= b</code>	a と b 以下なら <code>true</code> 。さもなければ <code>false</code> 。(*1)
算術演算	<code>a + b</code>	a と b の足し算
	<code>a - b</code>	a と b の引き算
	<code>a * b</code>	a と b の引き算
	<code>a / b</code>	a と b の割り算
	<code>a % b</code>	a と b の割り算の余り
シフト演算	<code>a >> b</code>	a と b ビット、右にシフト
	<code>a << b</code>	a を b ビット、左にシフト
条件演算	<code>a ? b : c</code>	a が真の時 b、偽のとき c
配列操作	<code>{a,b,c}</code>	a と b と c をこの順に並べた信号線の束を作る
	<code>a[b]</code>	a の b 番目の信号を取り出す
	<code>a[b:c]</code>	a の b 番目から c 番目の信号線の束を取り出す

表 6 Verilog HDL で用いられる演算の例

4.8.1 演算結果の代入

演算の結果をネット変数あるいはレジスタ変数に代入できます。ネット型変数への代入には `assign` 命令を使用して、下記のように記述します。

```
1 assign c = a & b;
```

これは、ハードウェア的には、演算の結果を信号線に接続することに相当します。ネット変数の代入は後述の同時処理文としてしか記述できません。

一方、レジスタ変数への代入では、ブロッキング代入とノンブロッキング代入の両方が使えます。Verilog HDL では、それぞれ下記のように記述します。

```
1 = // ブロッキング代入
2 <= // ノンブロッキング代入
```

たとえば、 $a + b$ の演算結果をノンブロッキング代入する場合には、

```
1 c <= a + b;
```

のように記述します。レジスタ変数への値の代入は、ブロッキング代入とノンブロッキング代入のどちらも、後で説明する順次処理文内でのみ使えます。

4.8.2 型の変換

VHDL が型に対して厳格であるのに対して、Verilog HDL では暗黙のうちに型が変換されます。意識せずに異なるビット幅の変数を演算、代入できます。

4.8.3 シフト演算

VHDL と異なり、Verilog HDL にはシフト演算子がありますが、シフト演算は大きな回路になってしまします。定数分のシフトを行いたい場合は、配列の結合演算を用いて実装する方が小さな回路として実現できます。たとえば、配列変数 $reg[n-1:0]$ の $counter$ を右に 1bit シフトしたい場合は、下記のようになります。

```
1 {1'b0, counter[n-1:1]};
```

また、左に 2 つシフトしたい場合には、

```
1 {counter [n-3:0] , 2'b00};
```

のように記述します。

4.9 同時処理文

ネット型変数の演算結果の代入は、同時処理文を記述します。記述されたすべての同時処理文は、合成ツールにより同時に解析され、回路として合成されます。つまり、記述された内容は、その順序に依存しません。たとえば、

```

1 assign c = a & b;
2 assign e = c & d;

```

という同時処理文の例も、

```

1 assign e = c & d;
2 assign c = a & b;

```

の例も、同じように図 8 の回路が合成されます。

4.10 順次処理文～always 文

always 文は、順次処理文を記述するためのブロックを作るものです。順次処理文では、記述された順序に従って意味が解析され、回路が合成されます。そのため、複雑な制御構文を使用できます。基本的な構文を下記に示します。変数 *c*, *d* は *reg* 変数です。

```

1 always @ (a, b) begin
2   c <= a & b;
3   d <= a | b;
4 end

```

ここで、**always** @() の「()」内の変数のリストをセンシティビティ・リストといいます。このリストに列挙した変数が変化すると、プロセス文の中の回路の値が変更されます。ノンブロッキング代入は、**always** 内の処理の解釈がすべて完了したタイミングで、同時更新されることに注意してください。この **always** 文は、図 7 のような回路になります。

always 文の中にある入力変数(式の右辺にでてくる変数)がすべてセンシティビティ・リストに列挙されている場合、入力が変化する度に回路が動作し、出力変数(式の左辺)の値が変更されます。つまり、その **always** 文は、何も状態を保存する必要がありません。そのため、記憶素子を必要としない組み合わせ回路として構成されます。ここで、入力に対して常に出力が生成されない場合とは、条件分岐などによって、入力信号の値によって値の代入が発生しない出力信号がある場合です。

次のような **always** 文を考えてみます。

```

1 always @ (a) begin
2   c <= a & b;
3   d <= a | b;
4 end

```

ここでは、センシティビティ・リストに「*b*」がなく、入力変数が全部列挙されていません。この回路では、*b* の値が変化しても出力先である *c* と *d* の値は変化しません。つまり、図 7 のように、入力である *a* と *b* を出力の *c* と *d* に直接接続することができず、*c* と *d* の値を保存する機構、記憶素子が必要となり、組み合わせ回

路としては合成されません。「同じように記述したつもりでも違う回路になるかもしれない」ということを覚えておいてください。

`always` の中では、`reg` 変数にブロッキング代入することもできます。ブロッキング代入では、`always` 中のほかの代入に関係なく、その時点での代入が発生し値が置き換わります。たとえば、次のような `always` 文を考えます。ここで、`tmp0`, `tmp1`, `c` はすべて `reg` 変数とします。

```

1  always @ (a, b) begin
2    tmp0 = a & b;
3    tmp1 = a | b;
4    c <= tmp0 ^ tmp1;
5  end

```

この場合、`tmp0` と `tmp1` とともに、ブロッキング代入で演算の結果が代入されています。ブロッキング代入なので、合成ツールがこの構文に出会ったところで、`tmp0` は $(a \& b)$ に、`tmp1` は $(a \vee b)$ —にすぐさま置き換われます。つまり、これは単に $c \leq (a \& b) \wedge (a \vee b)$ —という回路に合成されます。

センシティビティ・リストには、`@(a and b)` という条件式が記述できます。これは `a` と `b` のどちらかが変化した場合ではなく、`(a and b)` が変化した場合に `always` の中の処理を実行できるようにします。条件式には、信号の立ち上がり、あるいは立ち下がり条件を使用することもできます。立ち上がりは「`posedge`」、立ち下がりは「`negedge`」というキーワードを使います。

```

1  alwaysy @ (posedge clk) begin
2    処理文
3  end

```

これは、「`clk` の立ち上がり」のときに処理文が実行されることを意味します。ハードウェア的には、`clk` 信号が立ち上がった瞬間に相当し、それ意外のタイミングでは、処理文は動作せず、値が保存し続けられます。これは、決まったタイミングで処理を実行する順序同期回路の設計に欠かせない表現です。

4.11 制御構文

ソフトウェア・プログラミングのように、条件分岐の `if` や `C` でいう `switch` 構文のような制御構文が使えます。多くの制御構文は、`always` 文の中でのみ使用することができます。代表的なものを次に説明します。

4.11.1 `if ~ begin ~ end else begin ~ end`—条件分岐構文

`always` 文の中でのみ使用できる条件分岐構文です。次に示す例では、`a` が `b` より大きければ処理文 `X` が、それ以外の場合は処理文 `Y` が実行されます。また、処理文の中で、`if` をネスト（入れ子）にすることもできます。

```
1 if (a > b) begin
2   処理文 X
3 end else begin
4   処理文 Y
5 end
```

4.11.2 case — 選択

C や Java でいうところの switch 構文です。次に示す例では、変数 a の値によって処理文 X, Y, Z に分岐しています。case 構文では、どれかのケースに該当するように記述しましょう。もちろん、すべての条件を列挙してもよいのですが、「default」でどんな値にもマッチする場合を記述できます。

```
1 case(a)
2   0 :
3     処理文 X
4   1 :
5     処理文 Y
6   2 :
7     処理文 Z
8   default :
9     処理文 W
10  endcase
```

4.11.3 繰り返し構文 for

Verilog HDL における繰り返し処理は、単に似たような処理を繰り返して記述する代わりに簡単に書けるようにするための構文です。実際には、繰り返し回数分のハードウェア回路が生成されます。たとえば、下記に示す処理ですが、

```
1 tmp = 1'b0;
2 for (i = 0; i < 5; i = i + 1) begin
3   tmp = tmp | a[i];
4 end
5 b <= tmp;
```

これは、次のような代入文を記述することに相当します。

```

1  tmp = 1'b0;
2  tmp = tmp | a[0];
3  tmp = tmp | a[1];
4  tmp = tmp | a[2];
5  tmp = tmp | a[3];
6  tmp = tmp | a[4];
7  b <= tmp;

```

4.12 組み合わせ回路のサブモジュール

Verilog HDL では、複雑な組み合わせ回路を生成するために、`function` というサブモジュールを記述できます。`function` は順次処理文で、`if` や `case` などの条件文を記述できますが、ノンブロッキング代入を用いることはできません。`function` は、次のように定義します。

```

1  function f;
2    input a;
3    input b;
4    begin
5      if(a == b) begin
6        f = 1'b1;
7      end else begin
8        f = 1'b0;
9      end
10   end
11  endfunction

```

これは、1 ビットのネット型信号である `a` と `b` を入力とする名前 `f` の `function` の定義です。`function` の中に `f` に値を代入すると、関数の返り値としてセットされます。この例は、入力された二つの値が等しいときに'1'を、異なるときに'0'を出力する関数です。

呼び出し側では、

```
1  assign x = f(x, y)
```

または、`always` 文で

```
1  x <= f(x, y)
```

などとして呼び出せます。ここで、`x` と `y` は、関数呼び出しに対する実引数になります。

また、次のように、配列型の変数を入力あるいは出力する関数を定義できます。

```
1 function [1:0] g;
2   input [1:0] x;
3   begin
4     g[0] = x[1];
5     g[1] = x[0];
6   end
7 endfunction
```

ここで定義した `function` は、C の関数とは違い、関数の計算が終了するまで呼び出し側の処理が待たされるような制御を伴うことはなく、複雑な組み合わせ回路を見通し良く記述するための書き方です。

5 まとめ

ハードウェア記述言語である VHDL と Verilog HDL の基本について説明しました。どちらも一般のソフトウェア・プログラミング言語ではなじみの薄い、

- ノンブロッキング代入とブロッキング代入
- 同時処理文や順次処理文

といった、ハードウェアを設計するための特有の特徴をもっています。とはいえ、分岐構文などを使うと、ハードウェアで実装したい処理をソフトウェア的に設計できます。

今回は、基本的な文法の説明をかけあしで紹介しました。ソフトウェア・プログラミングを習得する際にはある程度、文法を覚えたあとは、他人の書いたコードをたくさん読むことで、言語をスムーズに習得できます。ハードウェアを設計するための HDL プログラミングでも同じです。たとえば、OpenCores.org には、さまざまな HDL コードが投稿されており、また、米国 Sun Microsystems 社からは、SPARC プロセッサの HDL コードが公開されています。習うより慣れろで、どんどん読み書きしてみましょう。

参考文献

1. Douglas L.Perry(著), メンター・グラフィックス・ジャパン株式会社 (翻訳); VHDL(Ascii software science—Language), 1996 年 10 月, アスキー.
2. Bhasker(著), デザインウェーブ企画室 (翻訳); VHDL 言語入門—ハードウェア記述言語によるロジック設計マスタリング (C&E TUTORIAL), 1995 年 7 月, CQ 出版社.
3. 長谷川 裕恭 ; VHDL によるハードウェア設計入門—言語入力によるロジック回路設計手法を身につけよう, 2004 年 4 月, CQ 出版社.
4. 小林 優 ; 入門 Verilog HDL 記述—ハードウェア記述言語の速習&実践 改訂, 2004 年 5 月, CQ 出版社.
5. 井倉 将実 ; FPGA ボードで学ぶ Verilog HDL, 2007 年 2 月, CQ 出版社.

Vivado を使った FPGA 開発 クイックスタート

FPGA 開発における Hello World “L チカ” を通して、開発の手順に慣れよう

1 はじめに

この章では、FPGA 開発における Hello World、すなわち最初に試してみるサンプルとして“L チカ”を題材に、開発の手順に慣れてみましょう。“L チカ”とは、「LED をチカチカさせる」の略です。

単に LED とチカチカさせるだけ、なのですが、開発の手順を身につけるには十分です。また、実際多くの信号制御は何かしらの規則にしたがって信号を ON/OFF していることですから、すべての信号制御は L チカの応用にあるとも言えます。簡単ですが大事ですよ。

2 実習

2.1 プロジェクトの用意

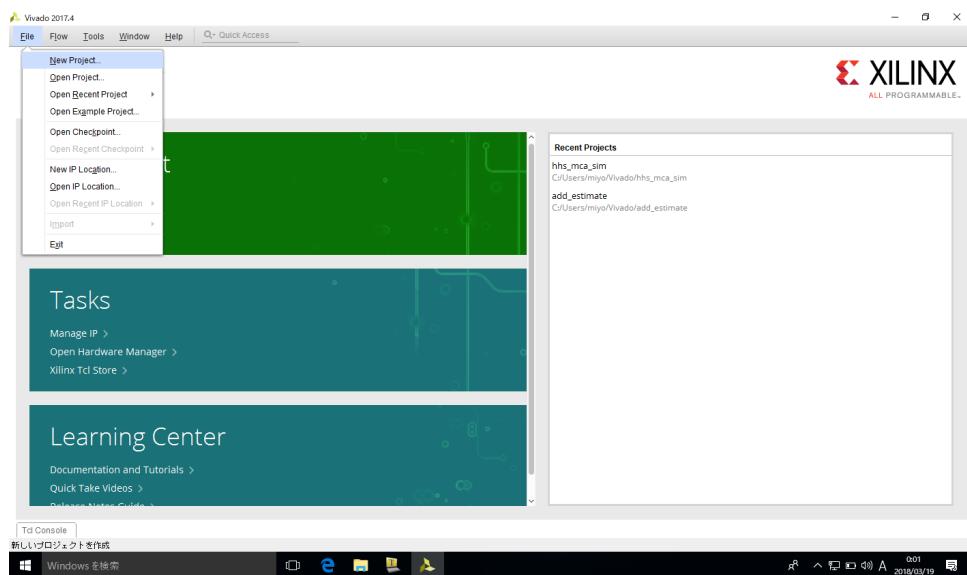


図 1 Vivado の File メニューから、New Project... を選択する

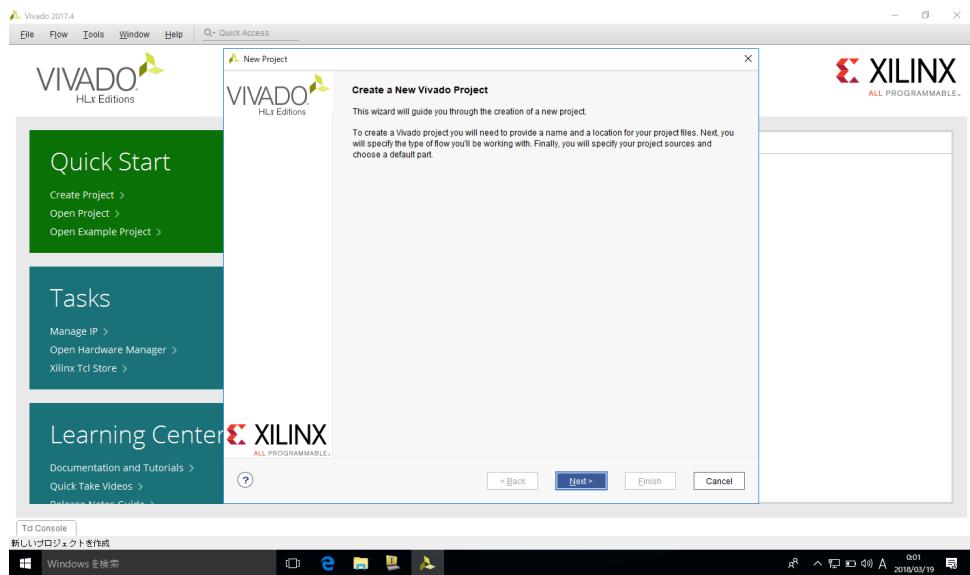


図2 プロジェクト作成ダイアログが開くので、Nextで次へ

プロジェクト名と格納先を決めます。ここでは、ホームディレクトリの下の Vivado というフォルダの下にプロジェクトを格納することとし、名前を project_1 としています。

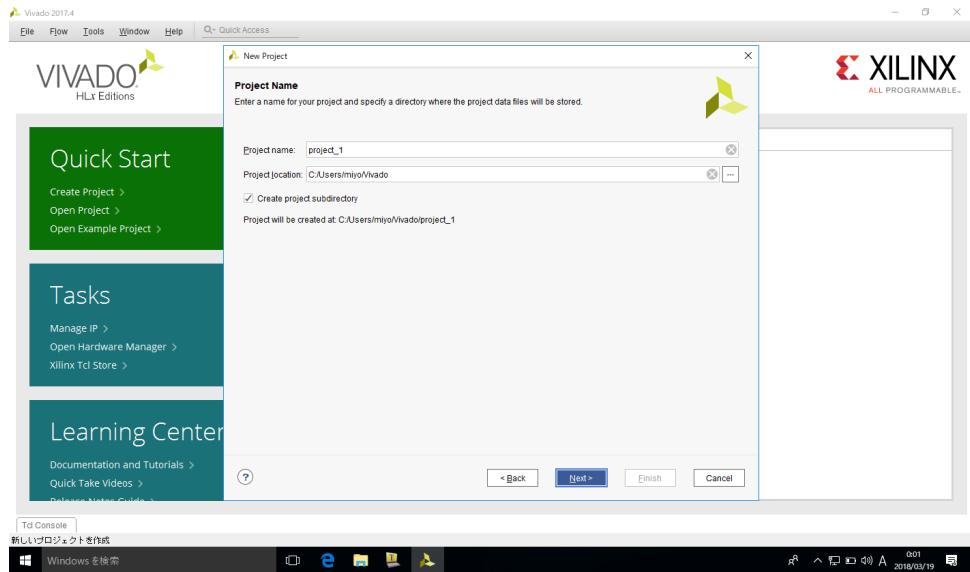


図3 プロジェクト名と格納先を決めます

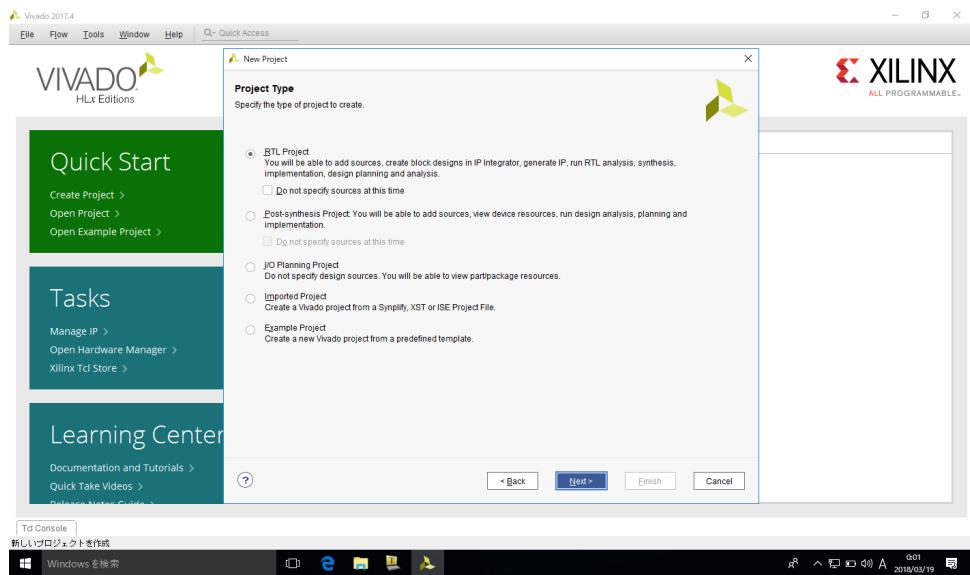


図 4 プロジェクトタイプの指定です。RTL プロジェクトを指定します

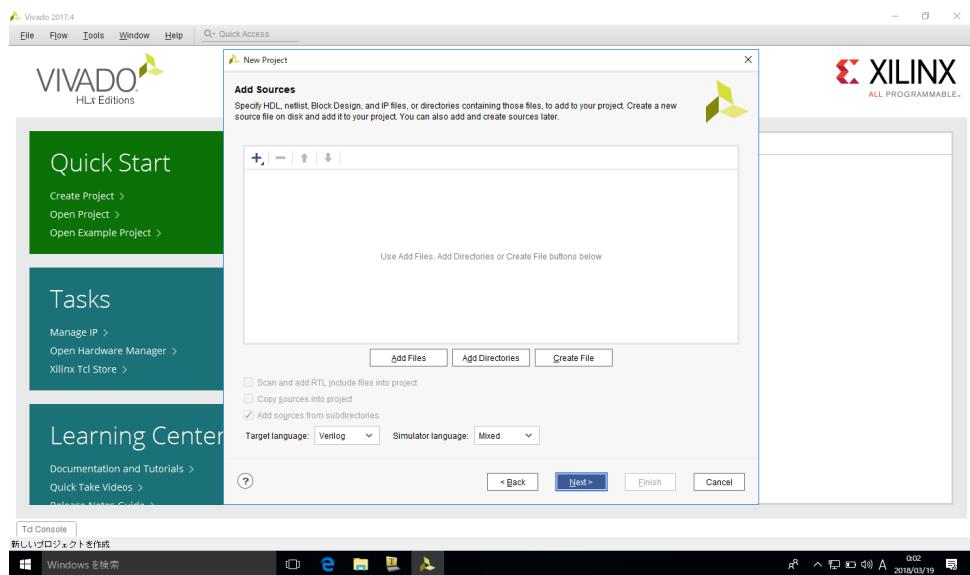


図 5 すでにソースコードがある場合にはここで追加できます。今回はないので Next で、そのまま次へ

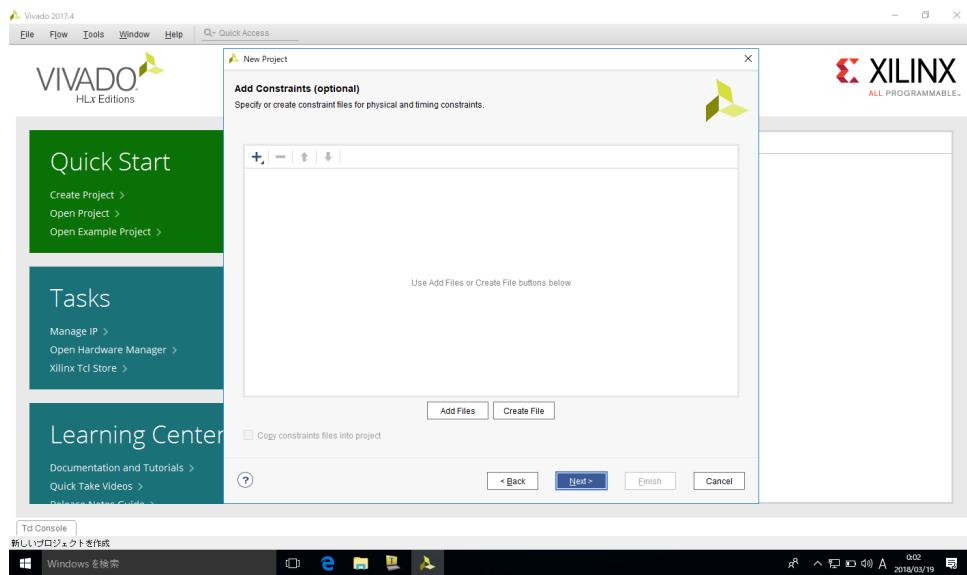


図 6 すでに制約ファイルがある場合にはここで追加できます。今回はないので Next で、そのまま次へ

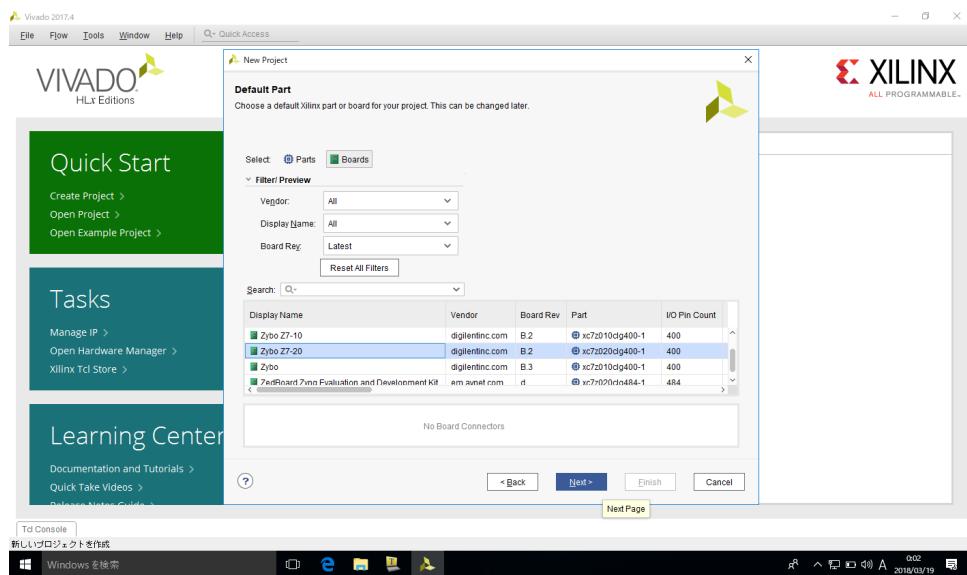


図 7 ターゲットとする FPGA の選択画面です。Board タブをクリックし Zybo Z7-20 を選択して で次へ

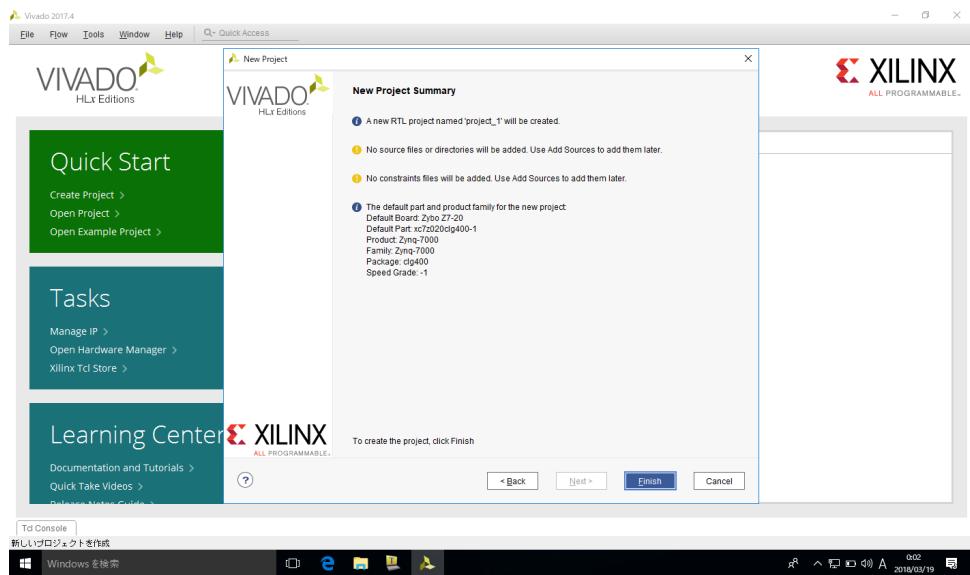


図 8 確認画面です。Finish をクリックするとプロジェクトの作成は完了です

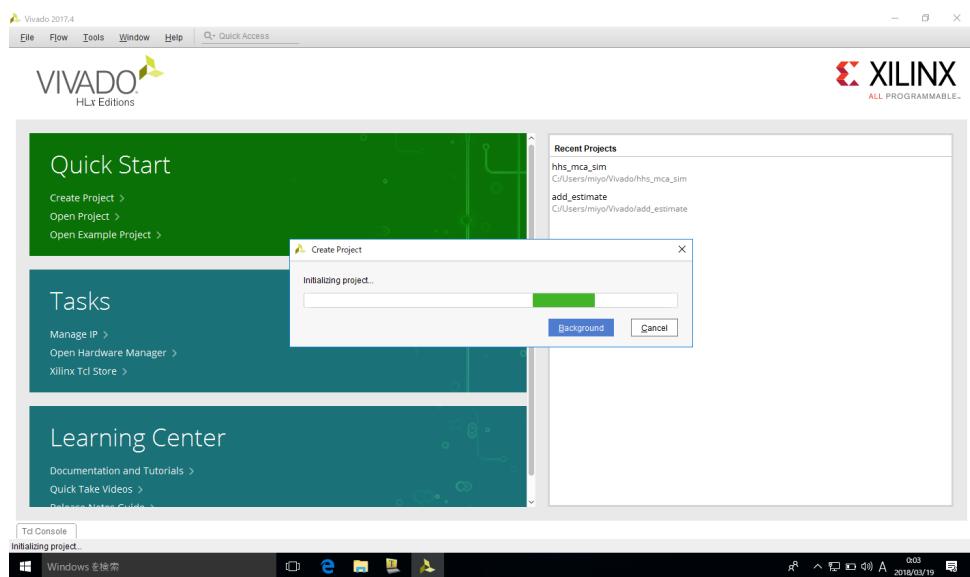


図 9 プロジェクト作成には少し時間がかかります

これでプロジェクトの完成です。Product Family が Zynq-7000 に、Project Summary の表示を見ると、Project part が Zybo Z7-20 と、Z7-20 向けのプロジェクトができていることがわかります。

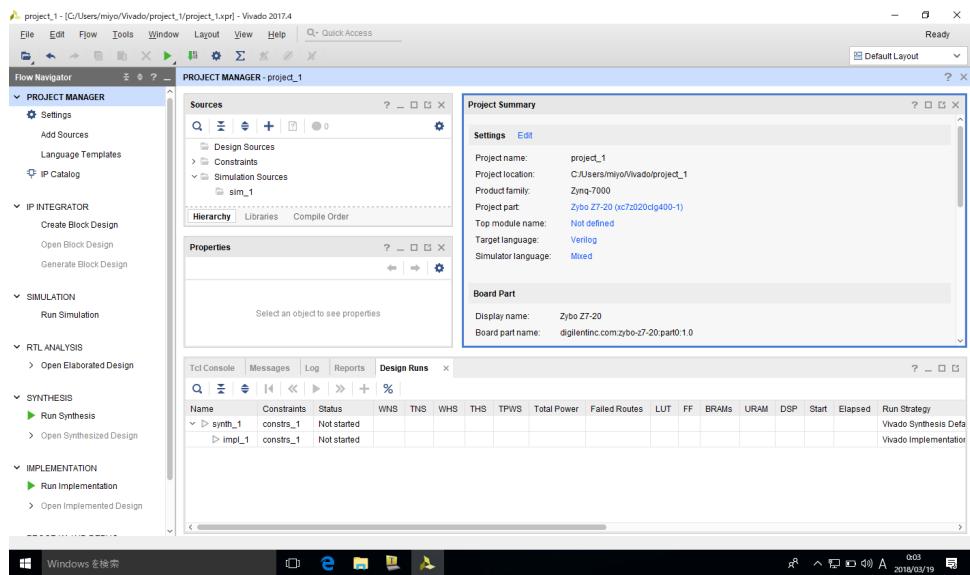


図 10 プロジェクトができあがりました。

2.2 ファイルの作成

作成したプロジェクトで、LED をチカチカさせるためのデザインを追加していきます。ここでは Vivado のウィザードを使ってデザインファイルを用意する方法ですすめていきます。

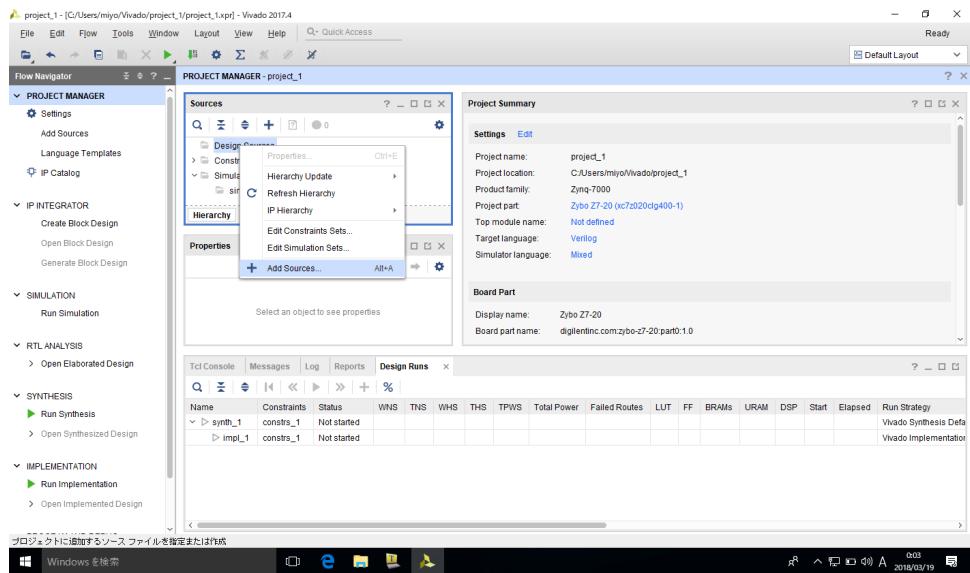


図 11 Sources の Design Source の上で右クリックし、Add Sources... を選択する

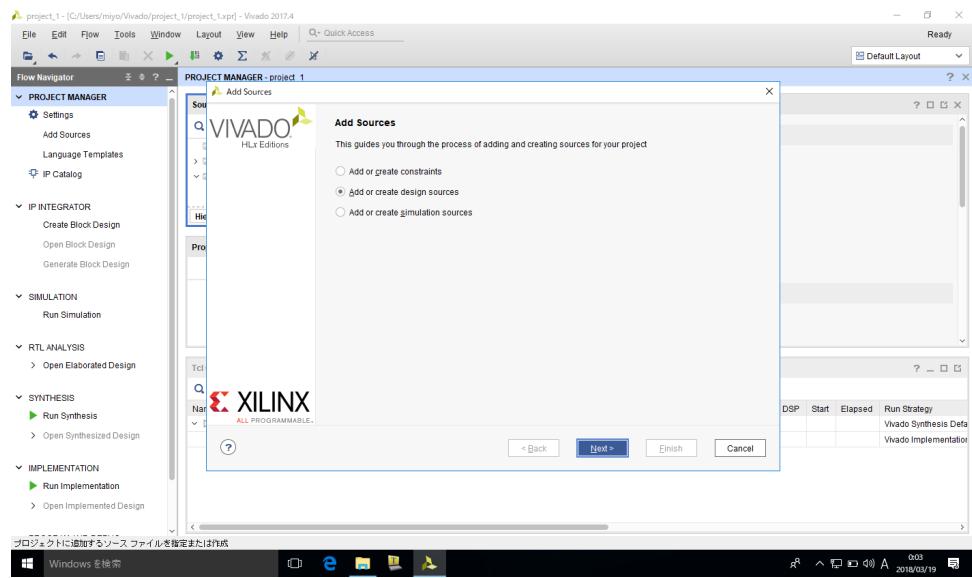


図 12 ファイル追加ダイアログが開く

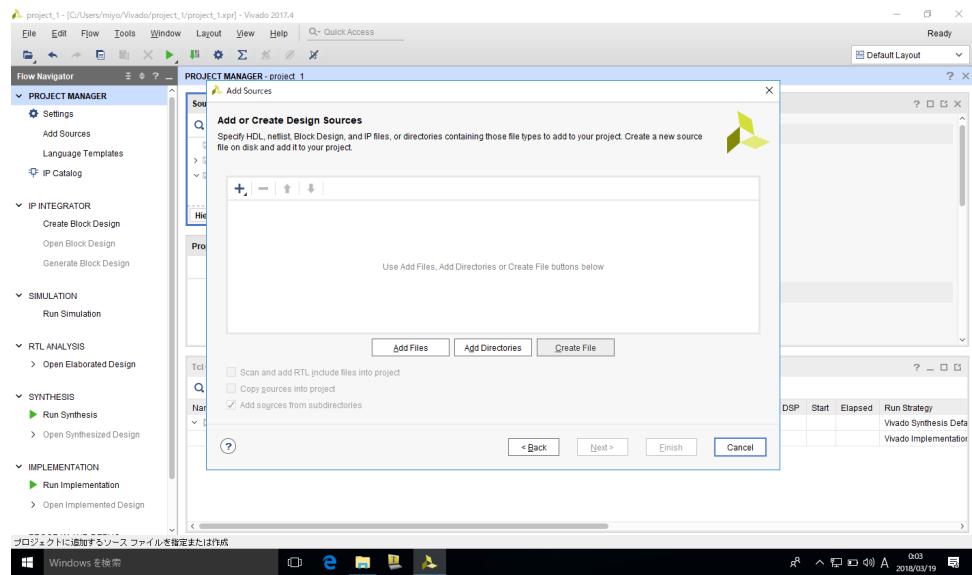


図 13 2 番目の Add or create design sources... にチェックをいれて Next をクリック

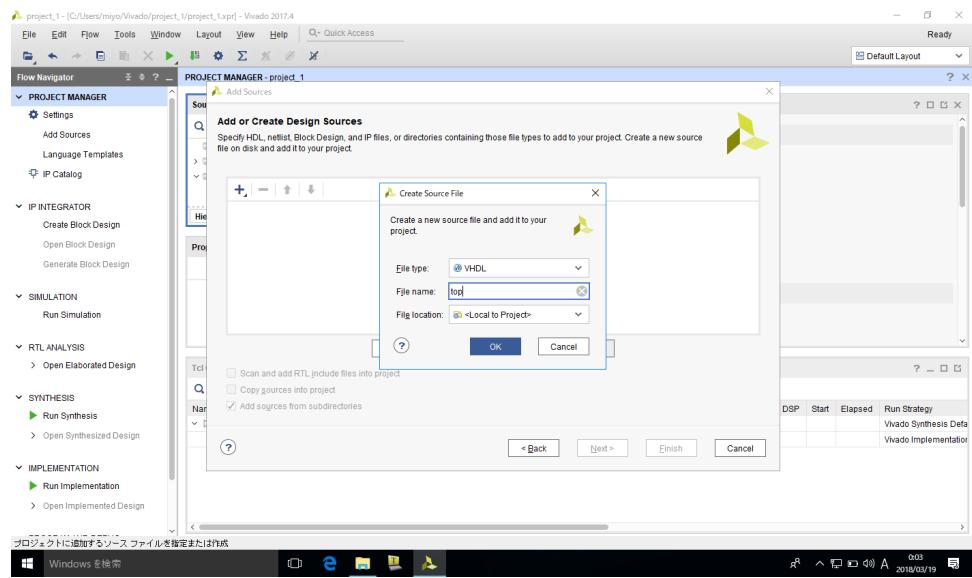


図 14 ここで新しくファイルを作るので、Create File をクリック

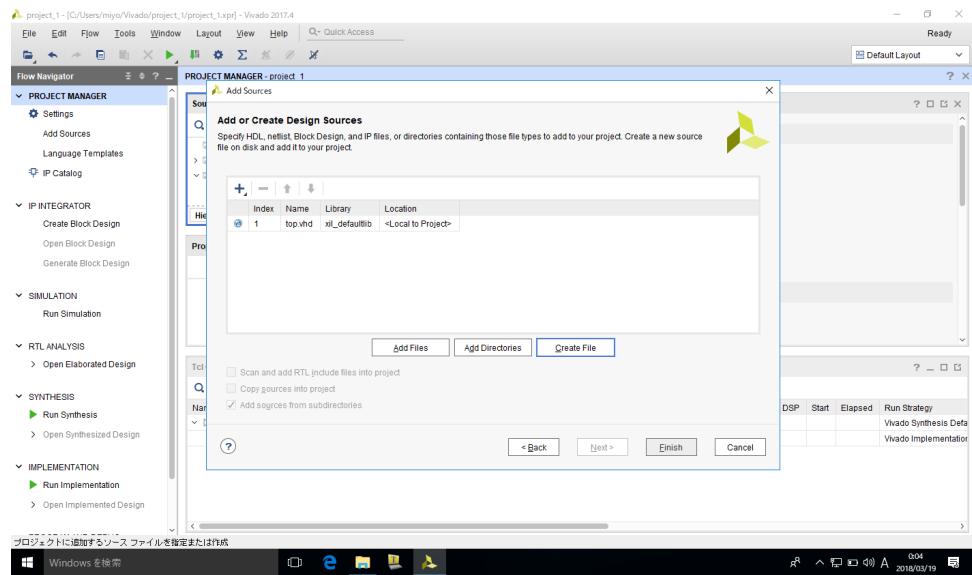


図 15 使用言語とモジュール名を決めます。ここでは VHDL を使うこととし、名前を top と決めて OK をクリック

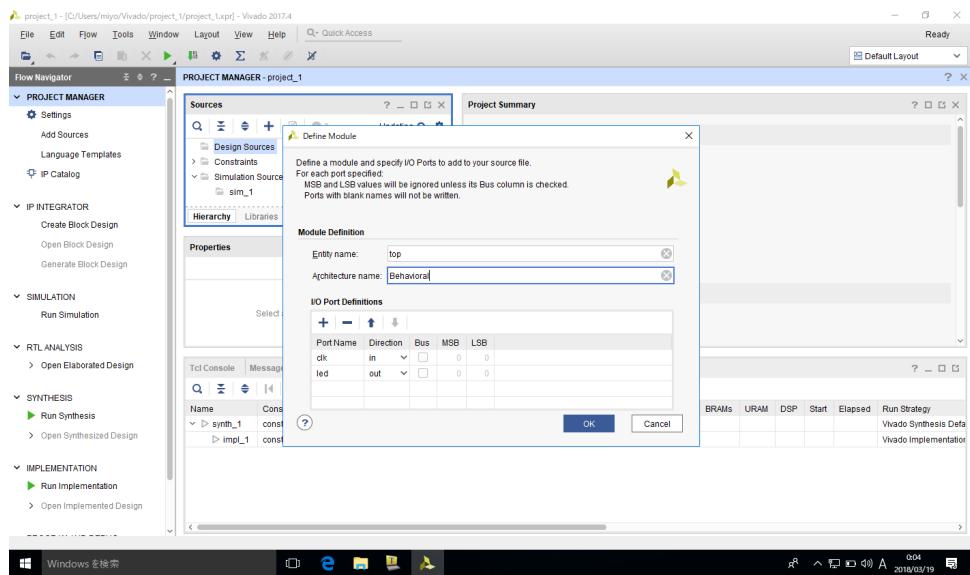


図 16 作成するファイルがリストに登録されたので、Finish をクリックします

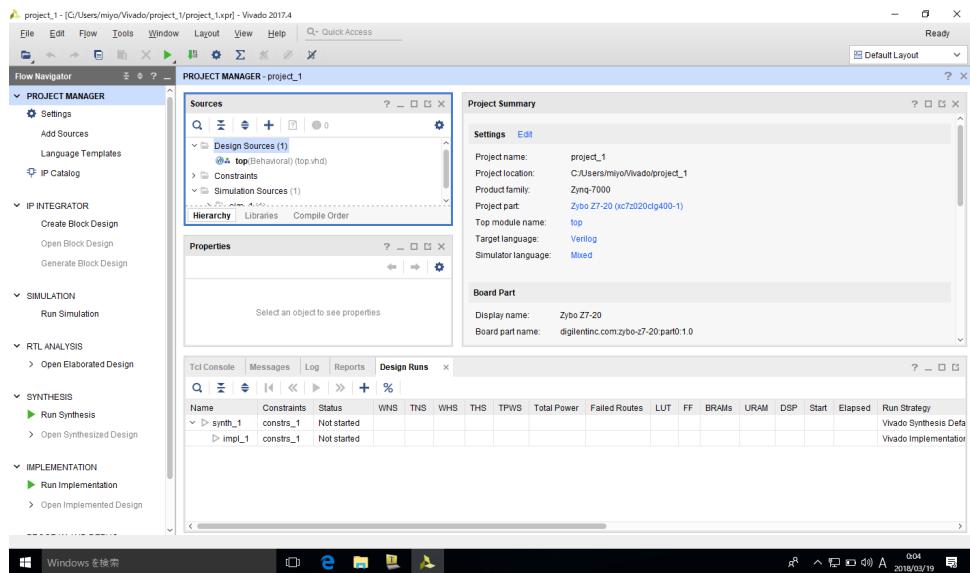


図 17 新たに作成した top モジュールのポートなどを定義することができます。あとで自分で書いてもいいのですが、入力ポートの clk と出力ポートの led を追加しましょう。OK をクリックします

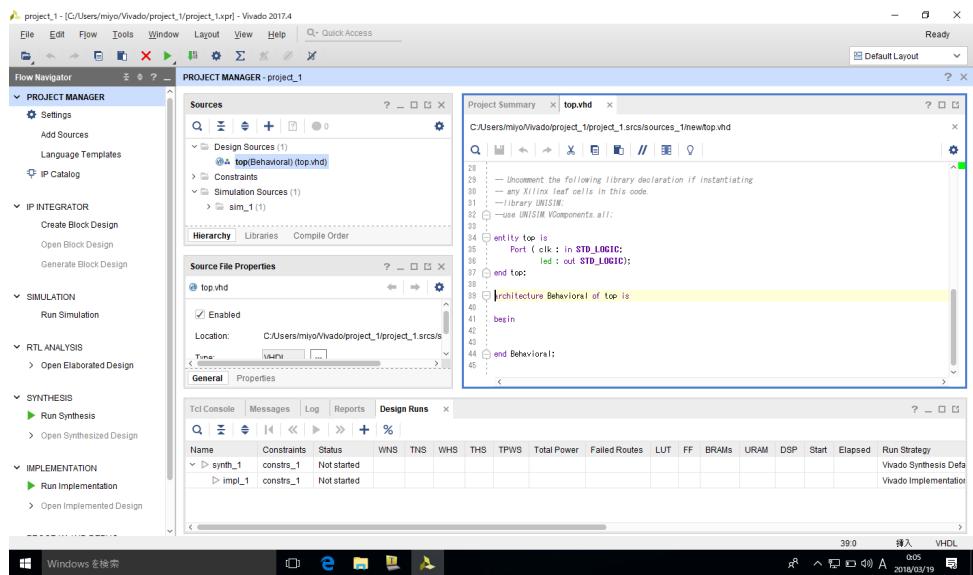


図 18 作成した top モジュールがプロジェクトに組み込まれました

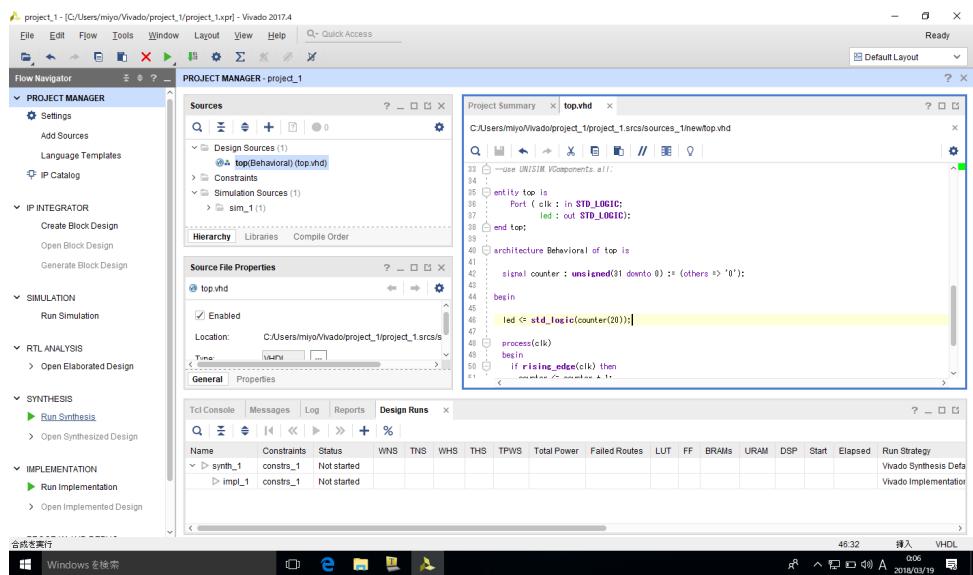


図 19 組み込まれた top をダブルクリックするとファイルを編集することができます

2.3 デザインの修正

ウィザードで作成した top モジュールは雛形でしかないので、このままでは、もちろん所望の動作はしません。次の内容で書き変えてください。スペースの都合上、作成されたコメントは取り除いています。ファイルを編集するときには気をつけてください。また、リスト中のコメントは説明上のものですので、実際には入力する必要はありません。

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use ieee.numeric_std.all; -- 追加する。数値演算に必要(1).
4
5 entity top is
6     Port ( clk : in STD_LOGIC;
7             led : out STD_LOGIC); -- このPortはウィザードで作成された
8 end top;
9
10 architecture Behavioral of top is
11
12     -- 次の行を追加。カウンタ変数の定義(2)。
13     signal counter : unsigned(31 downto 0) := (others => '0');
14
15 begin
16     -- 追加、ここから(3)
17     led <= std_logic(counter(23)); -- カウンタの23bit目をledに接続
18
19     process(clk) -- クロックの変化で動作するプロセス
20     begin
21         if rising_edge(clk) then -- クロックの立ち上がりであれば
22             counter <= counter + 1; -- カウンタをインクリメント
23         end if;
24     end process;
25     -- 追加、ここまで
26 end Behavioral;

```

追加箇所は多くないので、少し詳しくみてみましょう。

■追加箇所(1) 数値演算に必要なライブラリ、今回の場合でいうと“+”演算を利用するためには `use ieee.numeric_std.all` という前置きをしています。

■追加箇所(2) LED チカチカの、“チカチカ”というタイミングを制御するためのカウンタ変数を定義です。+ 演算を利用したいので `unsigned` 型で定義しています。

■追加箇所(3) いわば処理の本体です。一行目の代入文は、定義した `counter` 変数の 23bit 目を出力ポートである `led` に接続することを意味しています。`unsigned` の 1bit ですので `std_logic` に型変換しています。次の `process` 文では、`clk` が立ち上ることに `counter` に 1 を足すことを意味しています。すなわち、(3) で追加した部分では、`clk` の立ち上がり 8M 回毎に `led` の'1' と'0' が反転する回路を定義していることになります。

2.4 まずは合成してみる

ファイルの編集が終わったら合成してみましょう。といっても、左サイドにある Flow Navigator の中にある Run Synthesis をクリックするだけです。

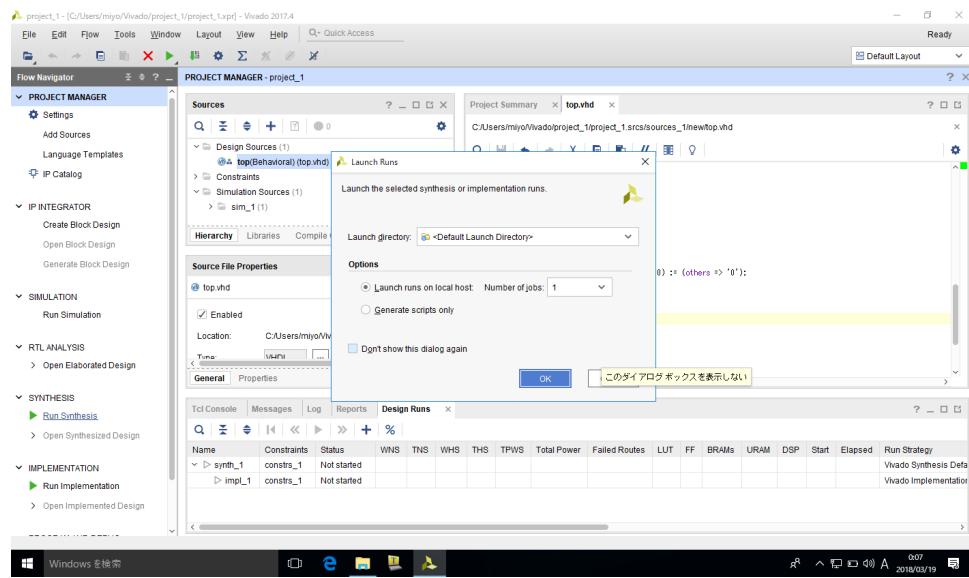


図 20 Run Synthesis をクリックすると合成開始ダイアログが開く。OK をクリックして合成を開始

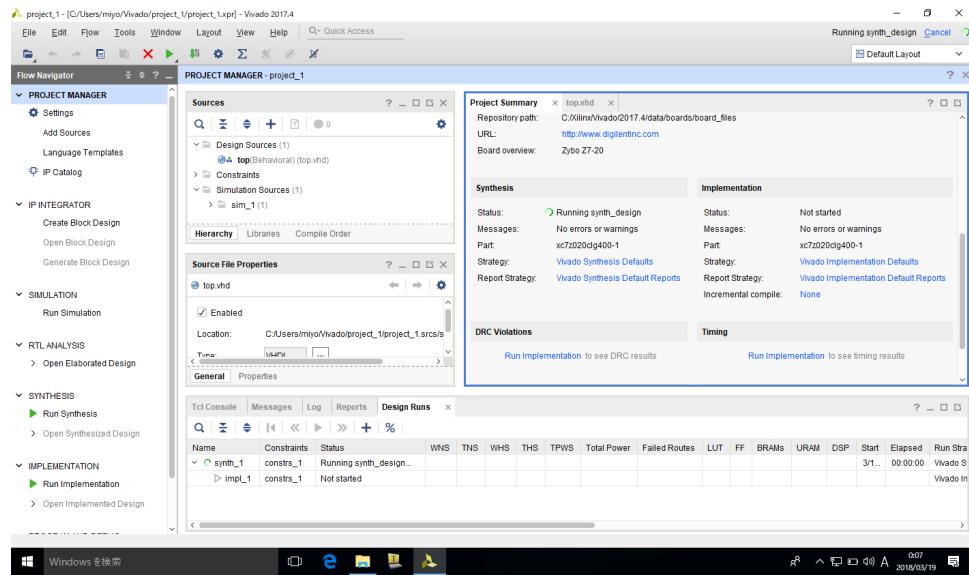


図 21 合成中は、Running Synth_design の横に、ぐるぐるアニメーションが表示される

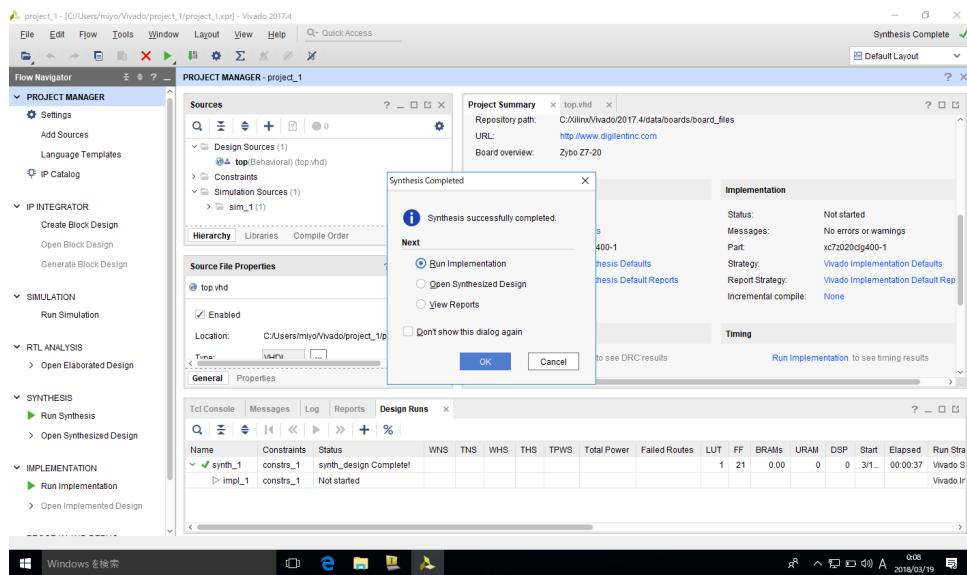


図 22 合成が完了した

2.5 I/O の設定

合成が完了したことでデザインが Vivado に受け付けられることが確認できました。しかし、これでそのまま FPGA でデザインが動くわけではありません。I/O ピンの設定をしていないからです。HDL のデザイン上は、入力ポート clk と出力ポートの led を定義しましたが、Vivado は、このポートを FPGA のどのピンに割り当てるか分かりません。FPGA を使った開発では人手でこの割り当てを決める必要があります。

ピンの割当ては合成結果を下敷に GUI で行なうことができます。まずは、合成結果を表示します。

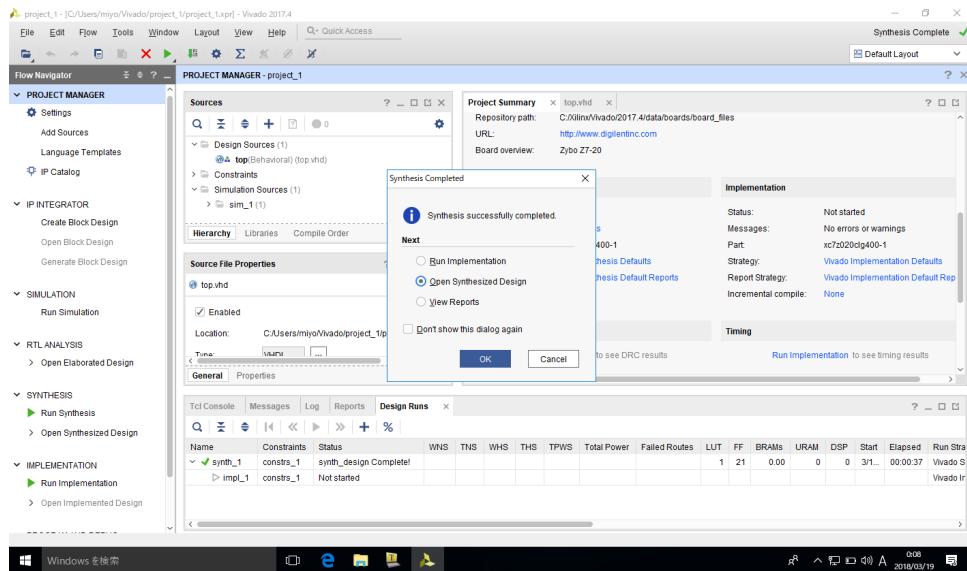


図 23 合成後のダイアログで、“Open Synthesized Design”を選択し、OK をクリックする

もし、うっかり合成後のダイアログを閉じてしまったり、ピン配置を指定しないまま次のステップにすすんでしまった場合には、Flow Navigator の中にある Open Synthesized Design をクリックしましょう。

合成結果が表示できたら、Layout の I/O Planning をクリックして I/O 設定用のモードに変更します。

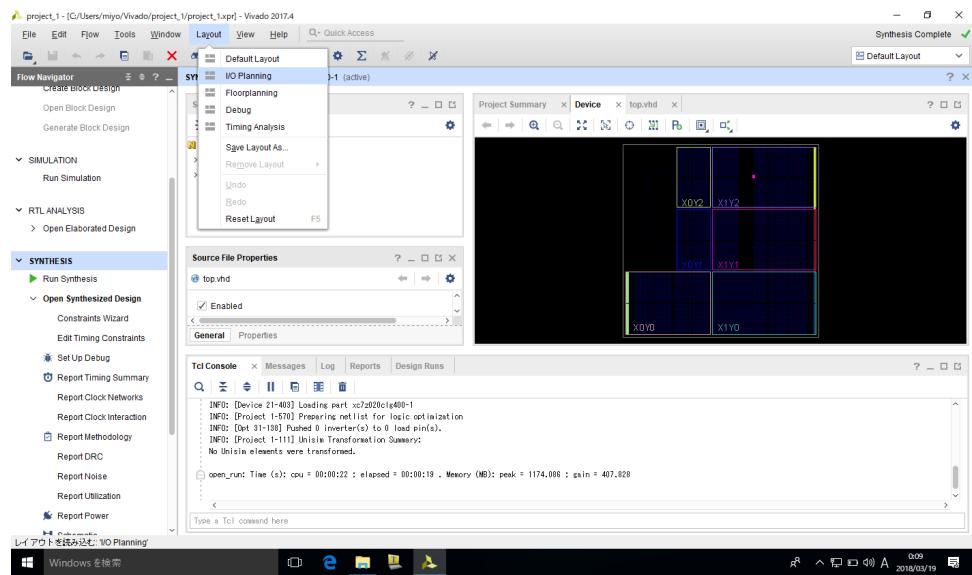


図 24 Layout の I/O Planning をクリックして I/O 設定用のモードに変更

画面下の I/O 割当て表でピン配置を決定します。clk と led の Package pin を、それぞれ、K17 と M14 に決め、どちらも I/O std を LVCMOS33 を選択します。

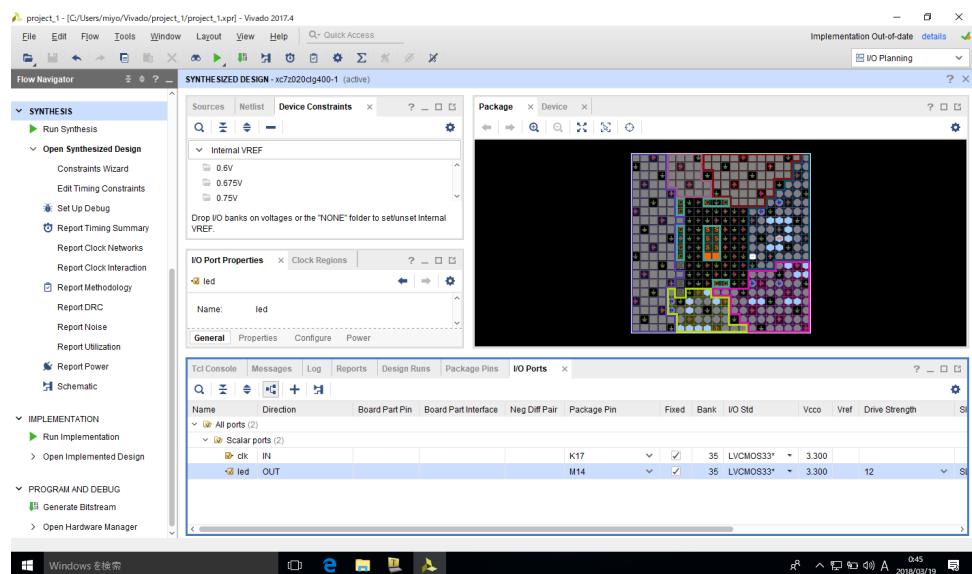


図 25 I/O のピン割り当てを決める

2.6 合成・配置配線

ピン配置がおわったので、FPGA 向けの書き込みファイルを作成します。Flow Navigator の中にある Generate bitstream をクリックするだけです。

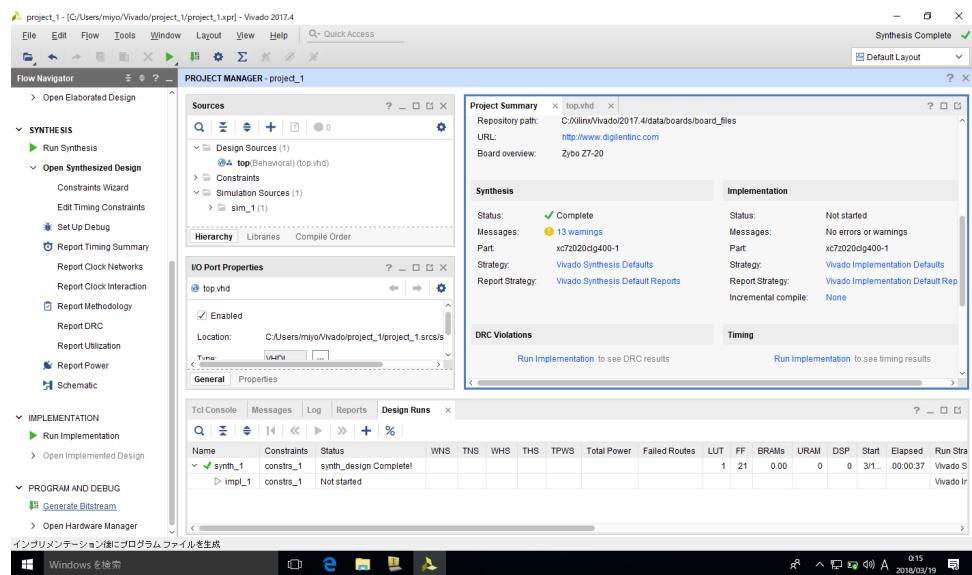


図 26 Flow Navigator の中にある Generate bitstream をクリックする。必要な合成・配置配線は一気通貫に実行される

2.6.1 制約ファイルの保存

おそらく先のピン配置の変更が保存されていないので、図 27 のダイアログが開くことでしょう。もちろん保存してください。

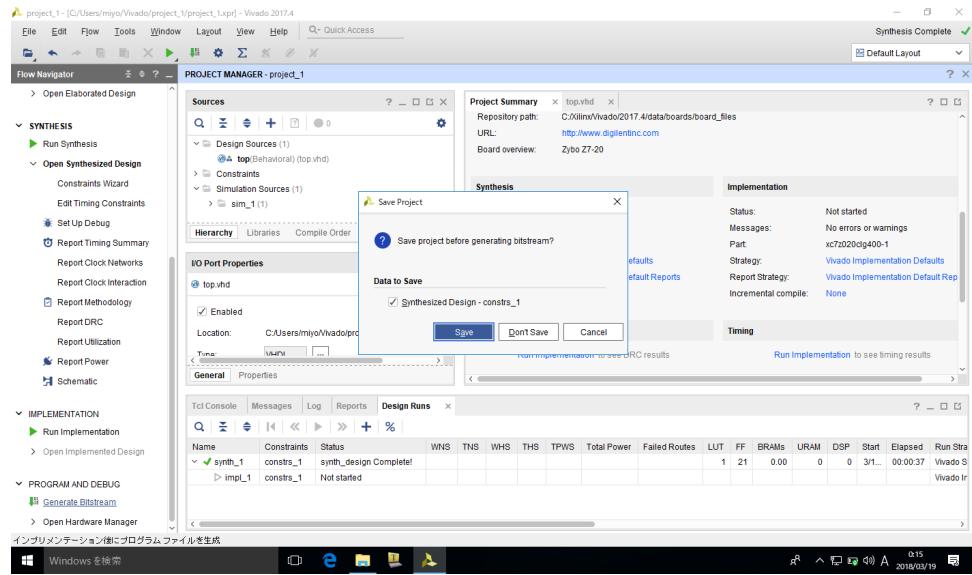


図 27 ピン配置の変更が保存されてないために表示されるダイアログ。Save をクリック。

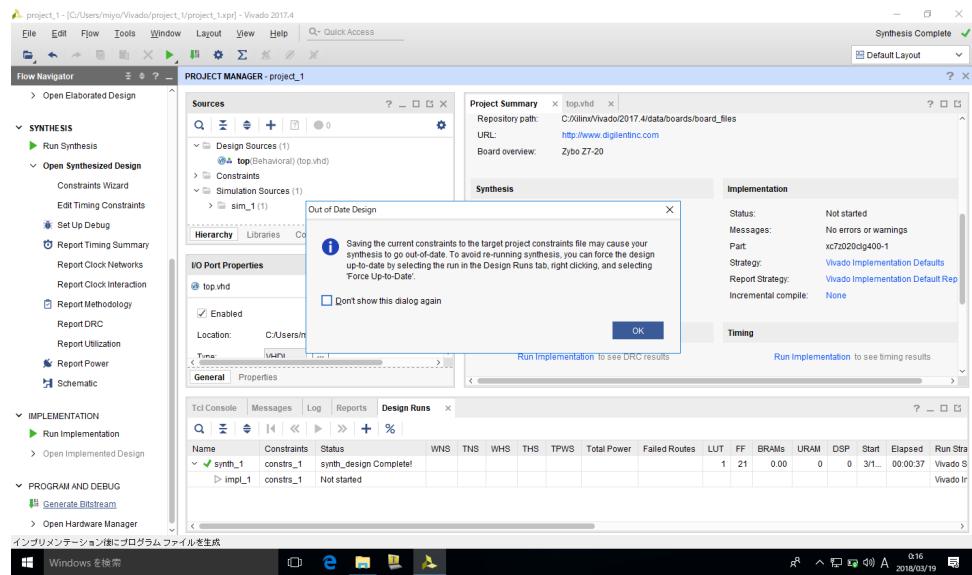


図 28 ファイル保存すると合成の結果が無効になるよという警告。OK で閉じる。

2.6.2 制約ファイルの指定(最初だけ)

はじめは、保存先のファイルが用意されていないので、図 29 のダイアログが開きます。ここでは、HDL モジュールと同じ名前の top というファイル名にします。なお、拡張子は xdc で自動的に付与されます。

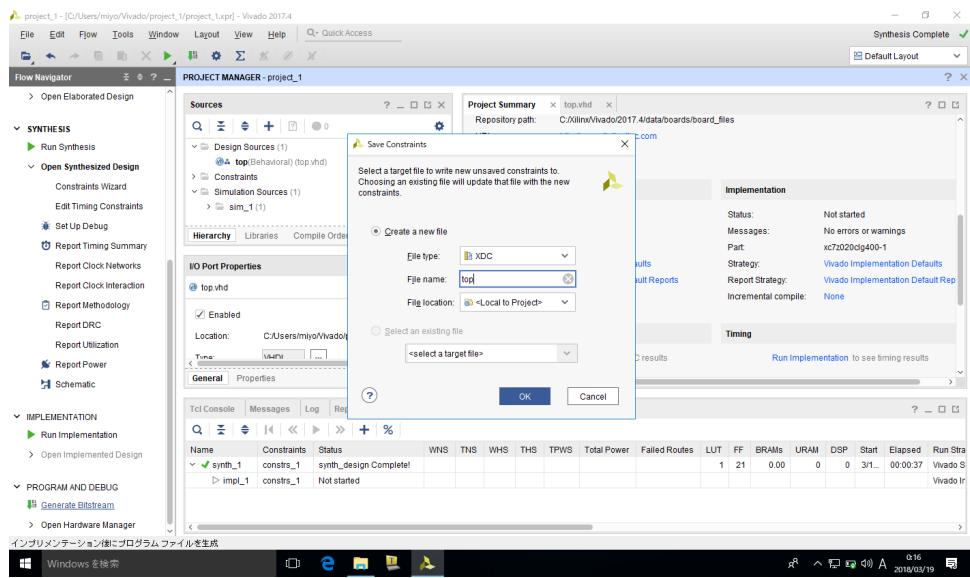


図 29 ピン配置を保存するファイルの作成を促すダイアログ。top という名前にして OK をクリック

2.6.3 合成・配置配線の開始

指定したピン配置の設定によって合成からやりなおしになります。“Yes”をクリックしてフローをすすめます。

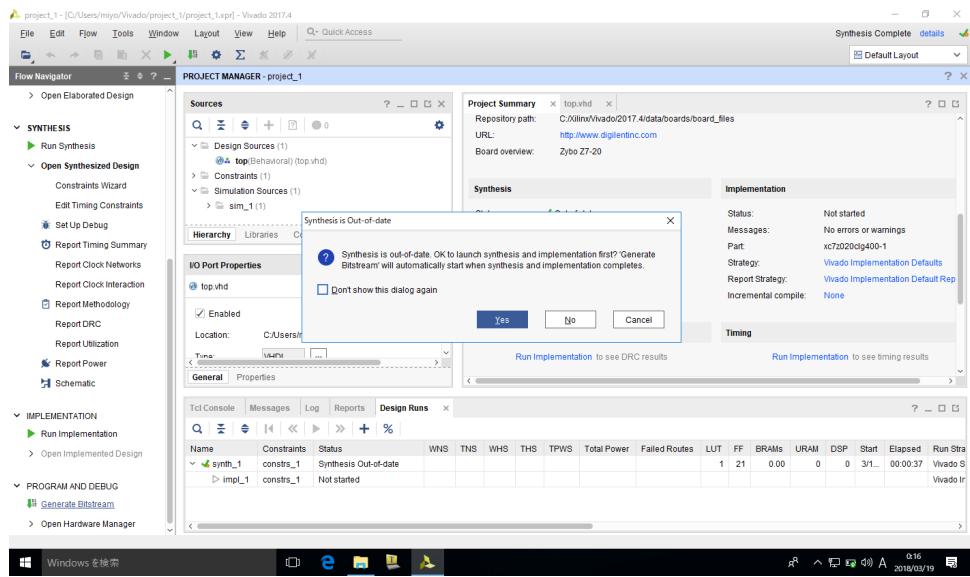


図 30 合成からやりなおし

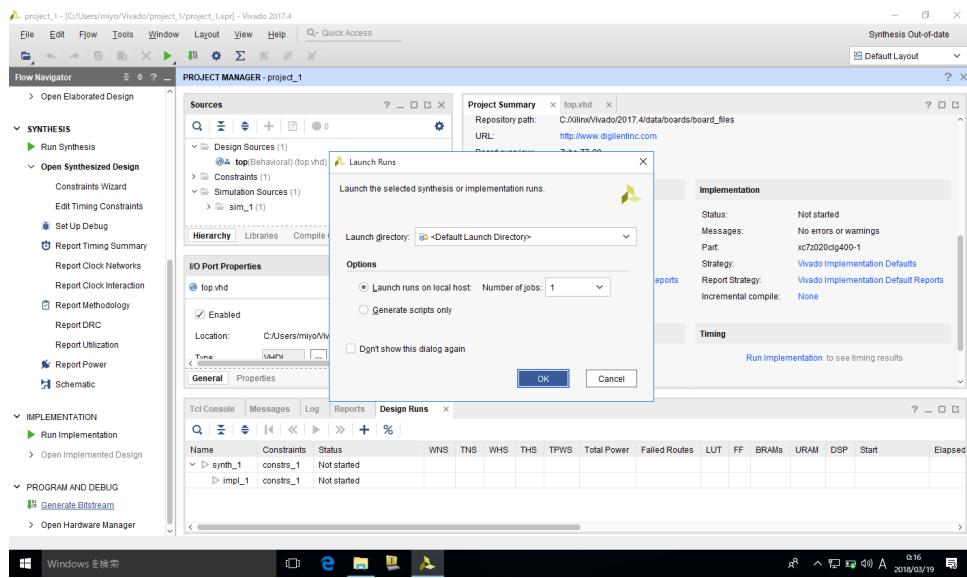


図 31 合成開始ダイアログ。OK で合成を開始。

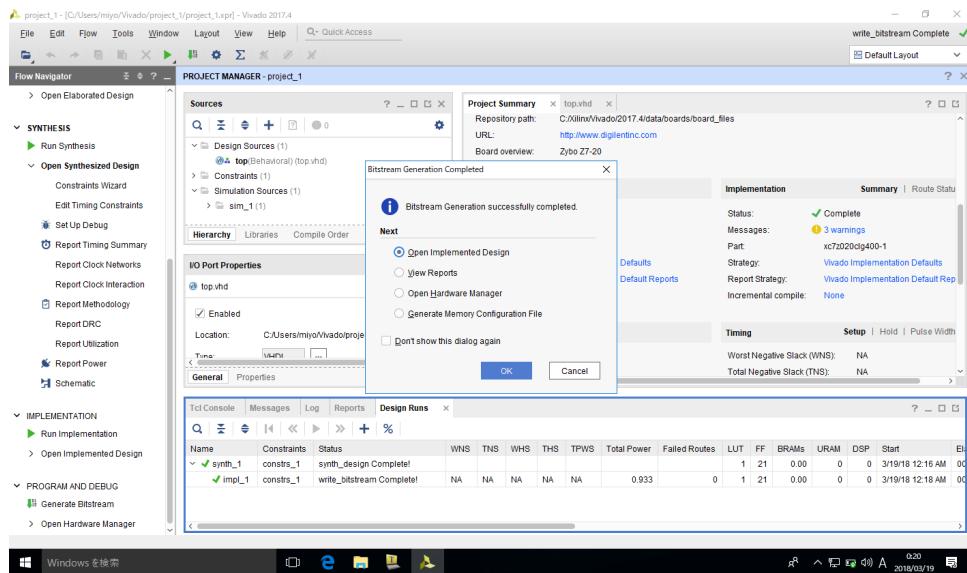


図 32 合成が無事に完了した。メインウインドウに Complete という文字と緑のチェックマークがついている

2.7 配置配線結果の確認

配置配線後、どのように FPGA 上に回路が配置されたかを確認することができます。今回は指定していませんが、動作クロックの指定を行った場合などには、正しくクロックの指定ができているか、その指定にあった回路ができているか、を確認する必要があります。

今回は、まず雰囲気だけみてみましょう。合成後の図 32 のダイアログで、Open Implementation Design に

チェックをいれて OK をクリックするか、Navigator Flow で Open Implementation Design をクリックします。

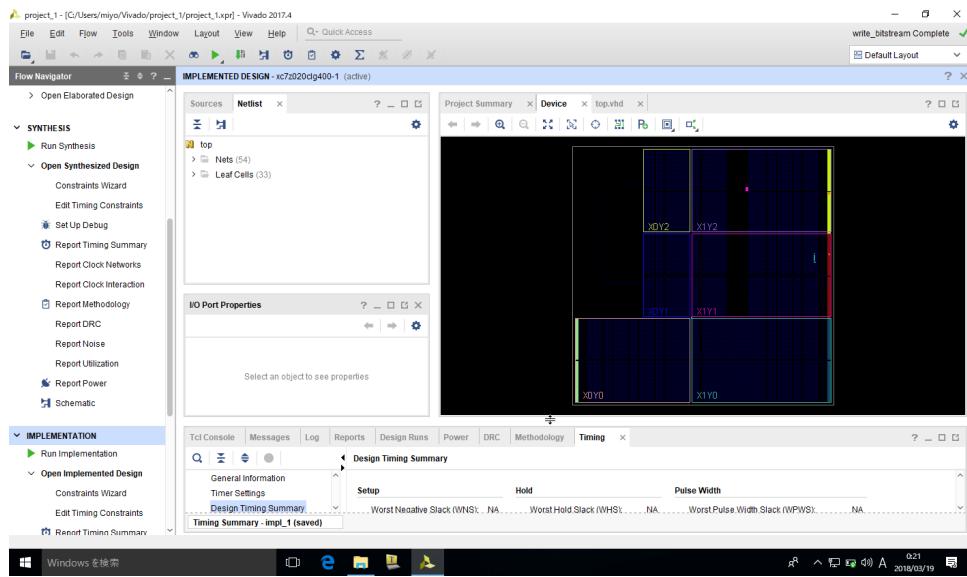


図 33 配置配線結果を開いてみたところ。右側の水色部分が使用しているハードウェアリソース

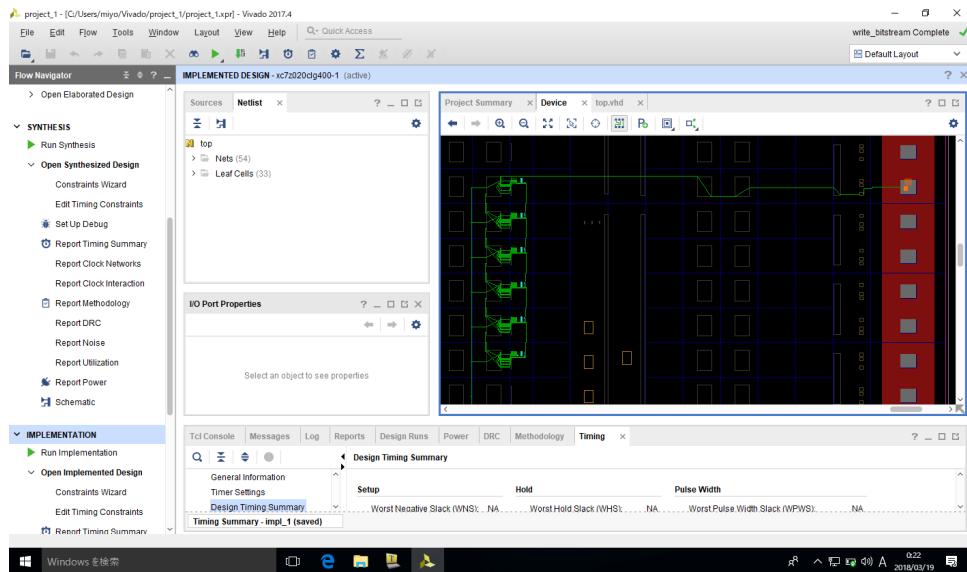


図 34 接続パスを表示し、拡大してみたところ

2.8 実機での動作確認

できあがったデータを FPGA ボードにダウンロードして動作させてみましょう。

まずは、FPGA とパソコンを USB ケーブルで接続します。ここで、JP5 のジャンパを JTAG に、JP16 の

ジャンパを USB 側にセットしておいてください。



図 35 FPGA とパソコンを USB ケーブルで接続。JP5 と JP16 の位置にも注意

書き込みの手順は次の通りです。

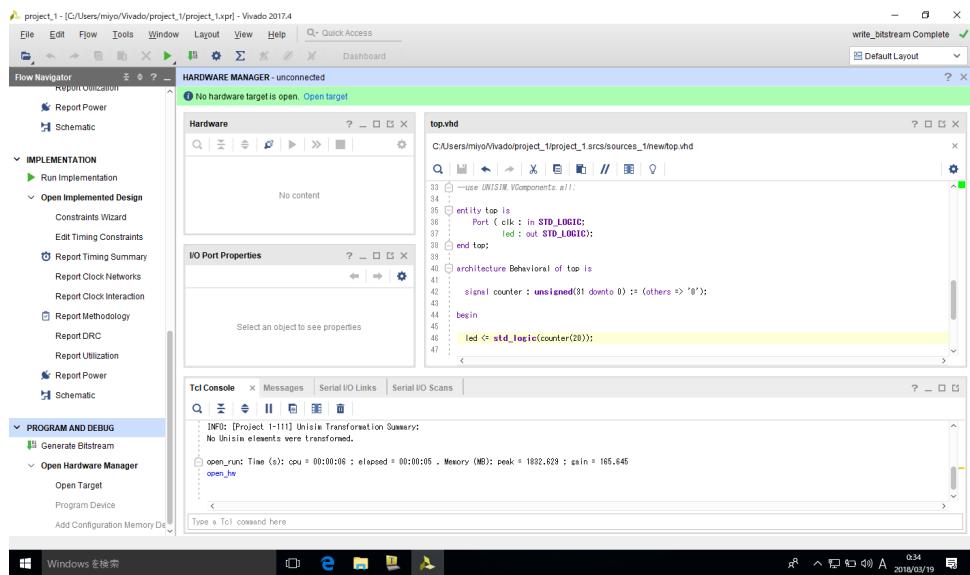


図 36 Navigator Flow で Open Hardware をクリックしたところ

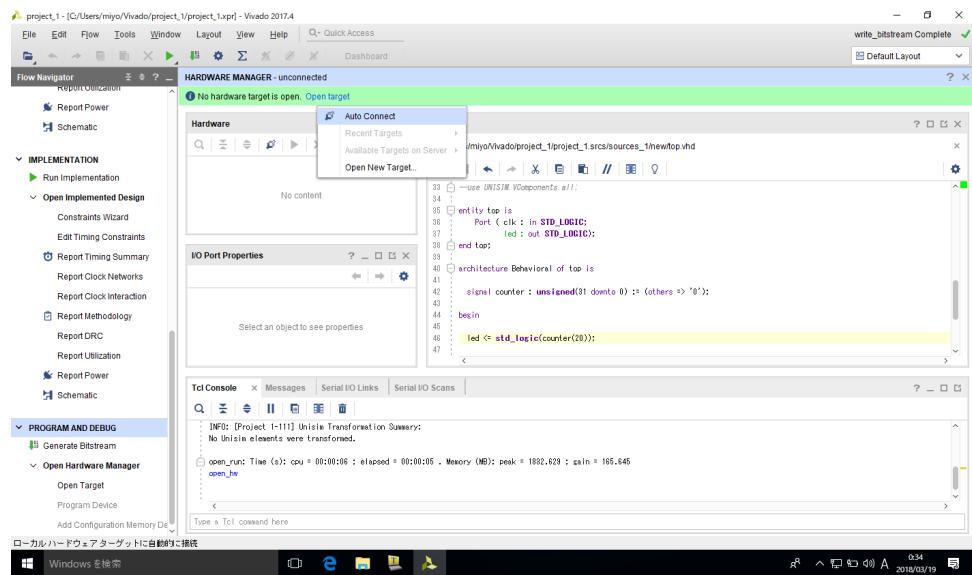


図 37 Open Target をクリックし、Auto Connect をクリック

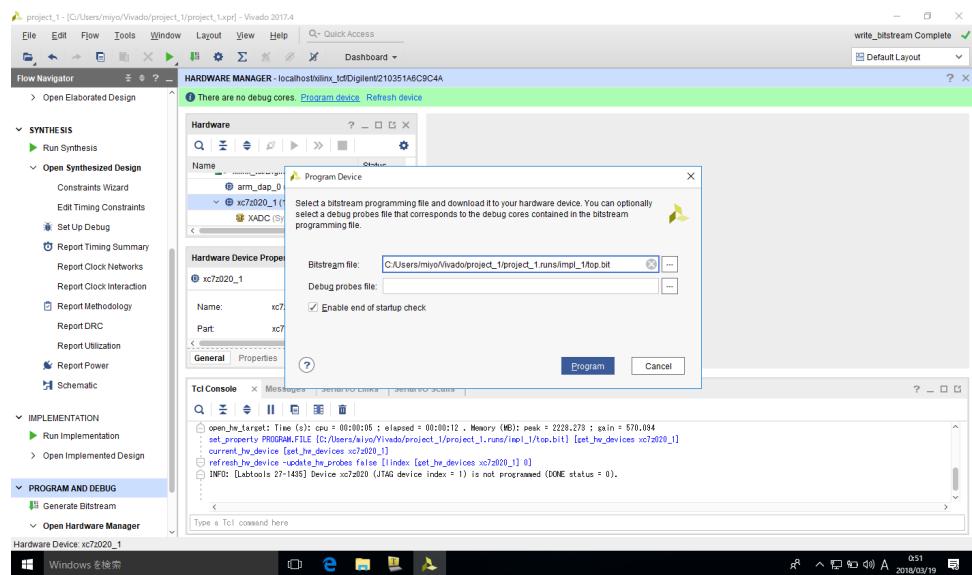


図 38 FPGA ボードと接続できたら、Program Device をクリック。開いたファイルダイアログで OK をクリック。

これで FPGA ボード上の LED LD0 が点滅するはずです。

3 課題

1. 点滅の間隔を変えてみよう
2. FPGA 上のほかの LED も点滅させてみよう

RTL シミュレーション

この章では，“Lチカ”を題材に設計したデザインがどのように動いているのか、Vivado 付属のシミュレータを使ったシミュレーションによって確認する方法を学びます。

1 はじめに

実際の開発現場で不具合を解析するときにもシミュレーションは大変有用な手段です。

HDL で記述したアプリケーション回路を FPGA 上で動作させるためには、合成や配置配線を実行して FPGA 書き込み用のコンフィギュレーション情報のファイルを生成する必要があります。回路規模が大きくなると、コンフィギュレーション情報を作成するのにも長い時間が必要になります。しかし、シミュレーションの場合は簡単なコンパイルでおしまいです。デバッグなどで、何度もソース・コードを変更するような場合には、シミュレータで動作を確認できれば圧倒的に短時間で済みます。

また、FPGA で動作しているアプリケーション回路の各信号が、どのように変化しているか、外から観測するのはなかなか難しいものです。一方で、シミュレーションでは自由に記述したロジックの信号の変化を観察できます。

そのため、シミュレーションを使用した内部の信号の観測が HDL レベルでの論理的なデバッグに有効です。この章では、この HDL コードの動作を Vivado シミュレータでシミュレーションする方法を学びましょう。

2 シミュレーションに必要なもの —— テスト・ベンチ

CPU を買ってきても、マザーボードがないとパソコンとして動かないように、FPGA も周辺部品の載った“マザーボード”がないと動きません。FPGA も FPGA 単体では動作せず、MicroBoard や DE0 nano に搭載されているような回路を駆動するクロック信号やりセット信号が必須です。

実際に FPGA を使ったシステムでは、クロックは外から与えられるのですが、シミュレーションでは、FPGA に実装した HDL モジュールが動作するのに必要なクロック信号やりセット信号なども自前で用意しなければなりません(図 1)。

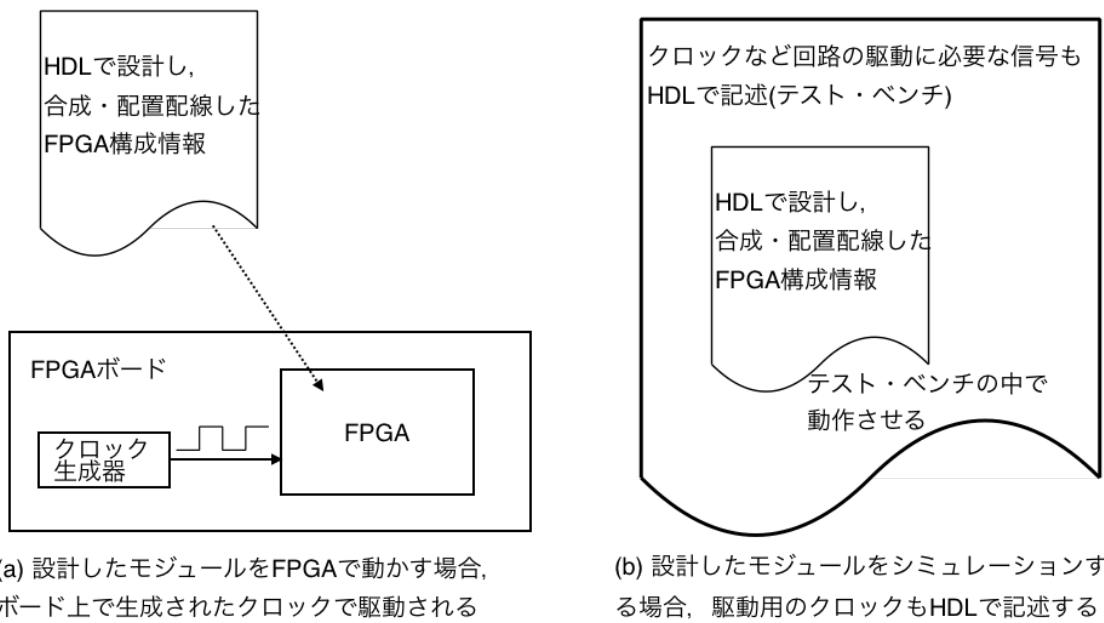


図 1 設計したモジュールを実機で動作させる場合 (a) とシミュレーションする場合 (b) の違い。シミュレーションする場合にはテスト・ベンチを用意する必要がある。

といって、とりたてて新しく特別なことを覚える必要があるわけではありません。外から与えられるべきクロック信号やリセット信号も HDL で記述できます。これをシミュレーション・コードやテスト・ベンチと呼びます。シミュレーション対象のモジュールで必要となる信号は、すべてテスト・ベンチで生成します。テスト・ベンチの記述方法は HDL ソースとほとんど同じで、違いは下記の 3 つになります。

- エンティティの中身(モジュールの入出力信号)が空である
- 時間を表す構文を使って信号の振る舞いを規定する
- ハードウェアでは実現できない仮想的な機能を利用できる

設計した HDL モジュールを動作させるのには必要な、マザーボードに相当する部分をすべて記述しなければなりません。逆にいって、テスト・ベンチは外部との入出力があつてはいけないということです。そのため、VHDL の場合はテスト・ベンチのエンティティの中身が、Verilog HDL の場合 module の引数が空になるというわけです。

2.1 テスト・ベンチでは“時間”を表現する必要がある

「スイッチを 10 秒押してから離す」や「50MHz の信号」などの振る舞いは時間を扱うので、HDL のビヘイビア・モデル（動作記述）を用いて記述します。

たとえば、「10ns の時間を待つ」という動作は HDL で次のように記述します。この記述はビヘイビア・モデルの基本中の基本です。VHDL で記述する場合は、

```
1 wait for 10ns;
```

Verilog HDL で記述する場合には、初めに ‘timescale を使って、

```
1  `timescale 1ns / 1ps
```

と、シミュレーションの単位時間を指定して、時間を挿入したい個所で

```
1  #10
```

と記述すると、指定した単位時間分の時間 (この例では 10ns) を作ることができます。

3 テストベンチの書き方

またか!!という感も否めませんが、「LED チカチカ」に再び登場してもらいましょう。コードは次の通りです。前章で利用したコードといいくつかの違いがあります。

■**reset** の追加 回路を初期化するための入力信号として **reset** を追加しています。

■**LED** に出力する **bit** を変更 実機ではカウンタの 23bit 目を **led** に接続しましたが、クロックの立ち上がりを 8M 回数つのは大変なので 3bit 目を **led** の出力に接続しています。

■モジュールのタイプを RTL に変更 モジュールのタイプを **Behavior** ではなく **RTL** としています。実際のところ今の Vivado では、このキーワードは特に意味をなさないのですがシミュレーションのためのテストベンチを **Behavior**、合成してハードウェア化もするファイルは **RTL** と分けておくと見分けるのが容易になります。

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 entity top is
6     Port ( clk      : in std_logic;
7             reset   : in std_logic;
8             led      : out std_logic
9         );
10 end top;
11
12 architecture RTL of top is
13
14     signal counter : unsigned(31 downto 0) := (others => '0');
15
16 begin
17
18     -- カウンタの 3bit 目を led に接続 (実機では 23bit 目を使った)
19     led <= std_logic(counter(3));
20
21     process(clk) -- クロックの変化で動作するプロセス
22     begin
23         if rising_edge(clk) then -- クロックの立ち上がりであれば
24             if reset = '1' then
25                 counter <= (others => '0');
26             else
27                 counter <= counter + 1; -- カウンタをインクリメント
28             end if;
29         end if;
30     end process;
31
32 end RTL;

```

図 2 シミュレーション対象のコード

FPGA 上に実装する場合には、第 3 章で紹介したように、3 つの入出力信号を FPGA 上の適切な I/O に接続しなければなりません。一方、ソフトウェアで動作をシミュレーションするためには、これらの信号テスト・ベンチで生成して外から与える、ことにします。

3.1 クロックを生成するテスト・ベンチ

順序回路の要がクロック信号です。従って、クロック信号の生成はテスト・ベンチの基本中の基本です。

カウンタ・モジュールの動作をシミュレーションするために、clk に 50MHz のクロック信号、すなわち 10ns ごとに ‘1’ と ‘0’ を繰り返す信号をテスト・ベンチで生成してみましょう。記述方法は、次の通りです。

3.1.1 VHDL で記述する場合

信号 `clk_i` を定義し, `clk_i` に ‘1’ を代入して 10ns 待ちます。その 10ns 後 `clk_i` に ‘0’ を代入, その 10n 後に再び ‘1’ を `clk_i` に代入, … ということを繰り返すことで, 10ns で周期的に $0 \rightarrow 1 \rightarrow 0 \rightarrow 1 \dots$ と変化する信号, すなわち 50MHz のクロック信号が生成できます。VHDL のコードで素直に実装すると, 「10ns 待つ」処理に相当する「`wait for 10ns`」を使って,

```
1 process begin
2   clk_i <= '1';
3   wait for 10ns;
4   clk_i <= '0';
5   wait for 10ns;
6   clk_i <= '1';
7   wait for 10ns;
8   clk_i <= '0';
9   wait for 10ns;
10  ...
11 end process;
```

となります。

この `process` 文には, きっかけになる信号なしで, シミュレーションの開始同時にすぐさま処理が開始されます。もちろん, これでもよいのですが, 実はプロセス文は最後まで到達すると, また先頭から開始されるので

```
1 process begin
2   clk_i <= '1'; wait for 10ns;
3   clk_i <= '0'; wait for 10ns;
4 end process;
```

という記述で, ずっとクロック信号を作り続けることができます。

3.1.2 Verilog HDL で記述する場合

Verilog HDL では, 「単位時間 XX が経過」を「#XX」で表現できますから, `clk_i` に'1' を代入したあと, 「単位時間 10 が経過」の後 `clk_i` に'0' を代入して, また「単位時間 10 が経過」の後 `clk_i` に'1' を代入…と繰り返すことで, $1 \rightarrow 0 \rightarrow 1 \rightarrow \dots$ というシーケンスが定義できます。

Verilog HDL では, `initial` 文で, シミュレーション実行開始時に一度だけ処理されるブロックを定義できます。すなわち, 次のコード片で, シミュレーション開始後から単位時間 10 毎に信号を反転させる処理, つまり 50MHz のクロック信号を生成できます。

```

1 initial begin
2   clk_i = 0; #10;
3   clk_i = 1; #10;
4   ...
5   forever #10 clk_i = !clk_i;
6 end

```

ずっと繰り返す処理を意味する `forever` を使って、

```

1 initial begin
2   clk_i = 0;
3   forever #10 clk_i = !clk_i;
4 end

```

と記述することもできます。

3.2 必要な入力信号を生成する

図 2 に示したリストの動作をシミュレーションするにはクロック信号 `clk` 以外に、人間が「えいやっ」とリセット・ボタンを押すことに相当する `reset` 信号もテスト・ベンチで生成しなければいけません。

ところで、人間がリセット・ボタンを押すのは、いつ、どのくらいの時間でしょうか？「リセット・ボタンは電源投入の 5 秒後に 10 ミリ秒間押される」などと決められるわけではありません。

従って、テスト・ベンチでは、シミュレーション対象の回路に対して、それらしい信号を想定して生成する必要があります。リセット・ボタンが押されるタイミング、であれば、特に制約があるわけではありませんので、電源投入後の適当な時刻に検知できる時間だけの信号が与えられる、という想定で十分でしょう。

リセット信号を生成する方法は、

1. クロックと同じプロセス・ブロックで記述する
2. 別のプロセスのブロック内で記述する

の 2 種類が考えられます。

ここでは、クロックとは別のプロセス・ブロック内で、変数 `reset_i` の値を $0 \rightarrow 1 \rightarrow 0$ と変化させることで、リセット・ボタンが押されたことに相当する信号を生成することにします。Verilog HDL であれば、リセットに相当するレジスタ変数 `reset_i` が適当なタイミングで変化するように記述できます。

```

1 initial begin
2   reset_i <= '0'; -- 最初はリセット信号は'0'
3   wait for 5ns;
4   reset_i <= '1'; -- 5n 秒後にリセット信号を'1' に
5   wait for 100ns;
6   reset_i <= '0'; -- しばらくしたら (100n 秒後), リセット信号を'0' に
7   wait; -- 以降は何もしない
8 end

```

3.3 テスト・ベンチとシミュレーション対象のモジュールを接続する

FPGA 上に書き込んだ回路に信号を与えるためには、物理的にクロックやスイッチの端子を配線することになります。シミュレーションする場合には、シミュレーション対象のモジュールとテスト・ベンチで生成した信号を HDL 的に接続する必要があります。

一般的には、テスト・ベンチをトップ・モジュールとして、その中でカウンタ・モジュールをサブモジュールとして呼び出すことにします。これはマザー・ボードがあって、その中に FPGA があって回路が動いている、という物理的な構造に上手くマッチします。VHDL も Verilog HDL も、モジュールを階層的に定義する仕組みをもっています。

VHDL では、

1. LED チカチカ回路のインターフェースを宣言
2. LED チカチカ回路のインスタンスを生成する

の 2 段階で、サブモジュールとして回路を呼び出すことができます。

具体的には、はじめに、次のように、

```

1 -- サブ・モジュールである test の素性を記述
2 component test
3   port (
4     clk    : in std_logic;
5     reset : in std_logic;
6     led    : out std_logic
7   );
8 end component;
```

component として、使用するモジュールの素性を宣言したあとで、次のように

```

1 -- サブ・モジュールをインスタンス化する
2 U: test port map(
3   clk    => clk_i,
4   reset => reset_i
5   led    => led,
6 );
```

このモジュールのインスタンスを生成、配線関係を記述します。

プログラミング言語 C でも、関数を呼び出す時には、関数の素性を明らかにするために「プロトタイプ宣言」をした上で、関数呼び出し文を書きますよね。VHDL でも同じようなものだな、と考えてもらえばよいでしょう。

3.4 テスト・ベンチの全容

説明が長くなりましたが、図 2 のリストをシミュレーションするためのテスト・ベンチの全容は、図 3 のようになります。

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4 entity top_sim is
5 end top_sim;
6 architecture Behavioral of top_sim is
7 -- シミュレーション対象のモジュールを宣言
8 component top
9     Port ( clk    : in std_logic;
10           reset : in std_logic;
11           led   : out std_logic
12       );
13 end component top;
14 -- シミュレーション対象のモジュールの信号用の変数を宣言
15 signal clk_i    : std_logic := '0';
16 signal reset_i : std_logic := '0';
17 signal led_i    : std_logic := '0';
18 begin
19 -- シミュレーション対象のモジュールのインスタンス生成
20 U : top port map(
21     clk => clk_i,
22     reset => reset_i,
23     led => led_i
24 );
25 -- クロックを生成
26 process
27 begin
28     clk_i <= '1'; wait for 10ns;
29     clk_i <= '0'; wait for 10ns;
30 end process;
31 -- リセット信号の生成
32 process
33 begin
34     reset_i <= '0';
35     wait for 5ns;
36     reset_i <= '1';
37     wait for 100ns;
38     reset_i <= '0';
39     wait;
40 end process;
41 end Behavioral;
```

図3 図2をシミュレーションするためのVHDLによるテスト・ベンチ

4 シミュレーションの実行手順

シミュレーションのテスト用に新しくプロジェクトを作成しましょう。プロジェクト名は、project_2としました。

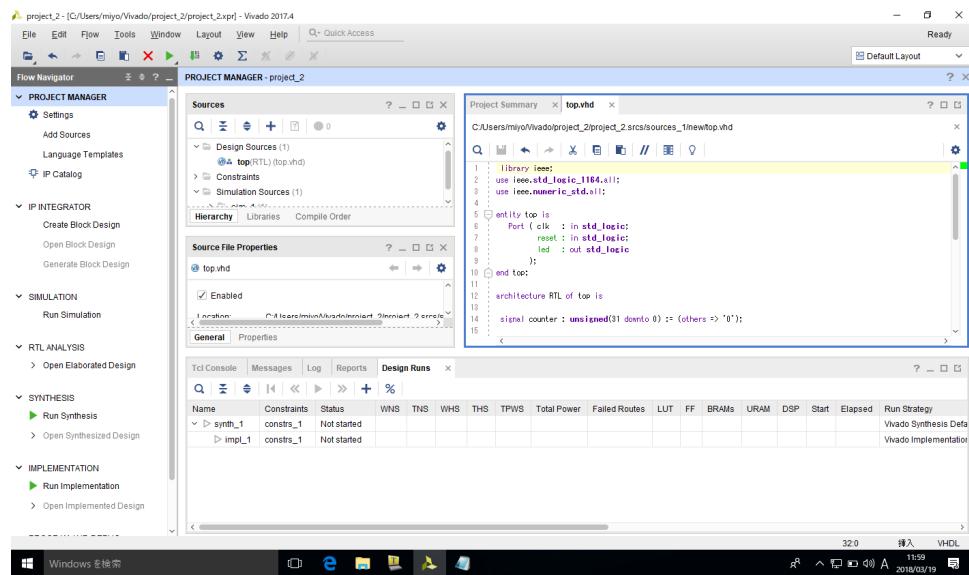


図4 新しく作成したプロジェクト

前章と同様に top モジュールを作成・追加してプロジェクトを作成したら、top モジュールの中身を図 2 のように変更しておきます。

以降の手順は次の通りです。

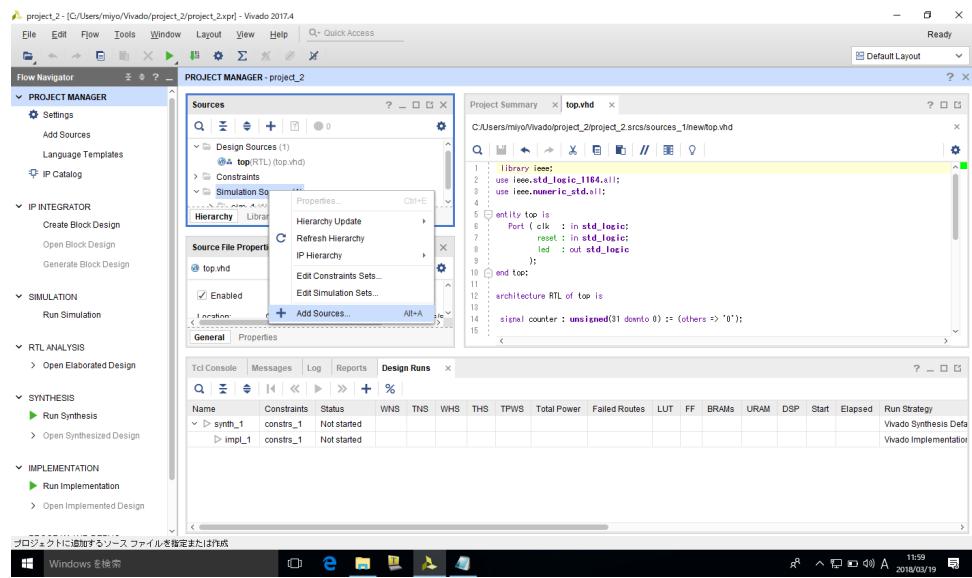


図 5 Sources の中の Simulation Sources の上で右クリックして Add Sources を選択

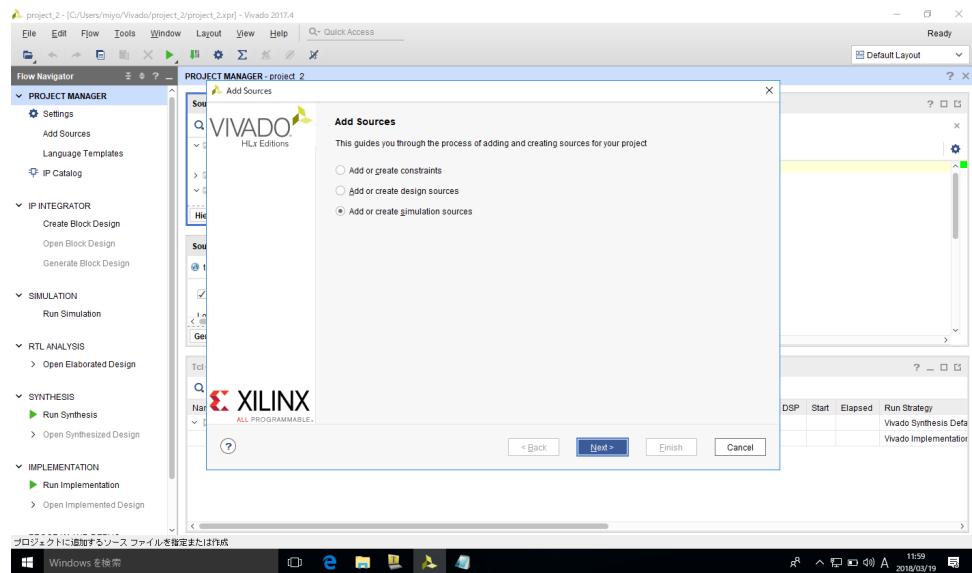


図6 “Add or create simulation sources”にチェックが入っていることを確認して Next をクリック

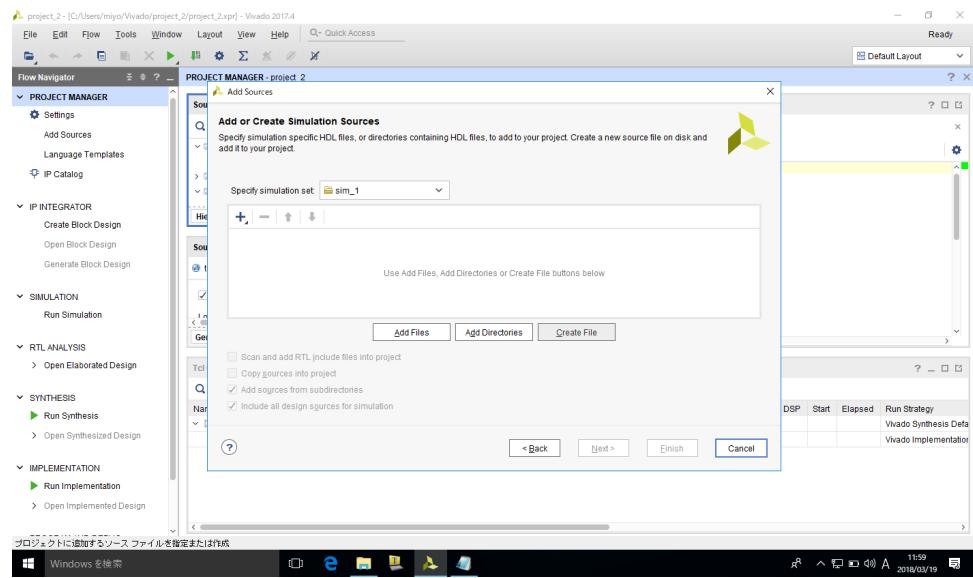


図 7 ファイル追加ダイアログで、Create File を選択

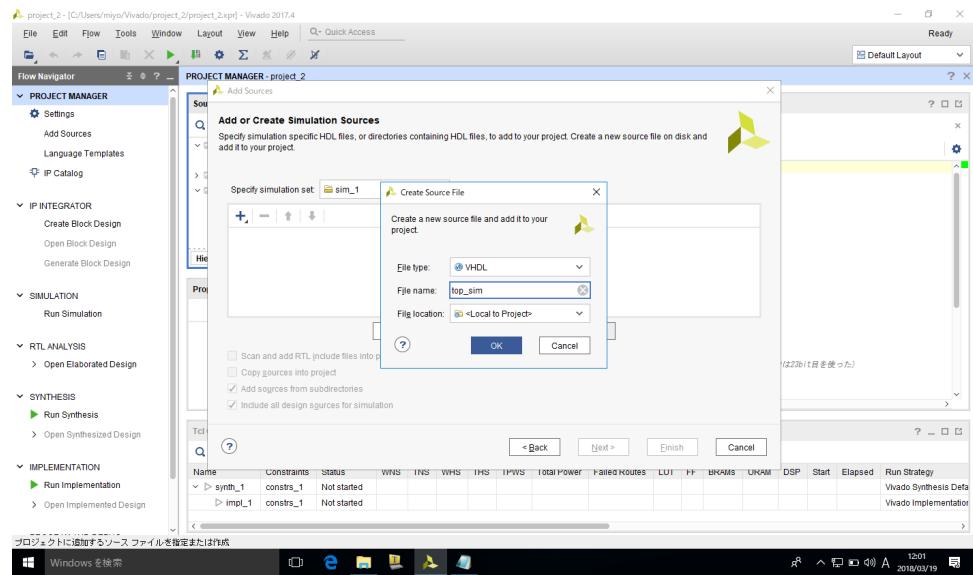


図 8 VHDL ファイルとして、top_sim という名前のモジュールを作成する

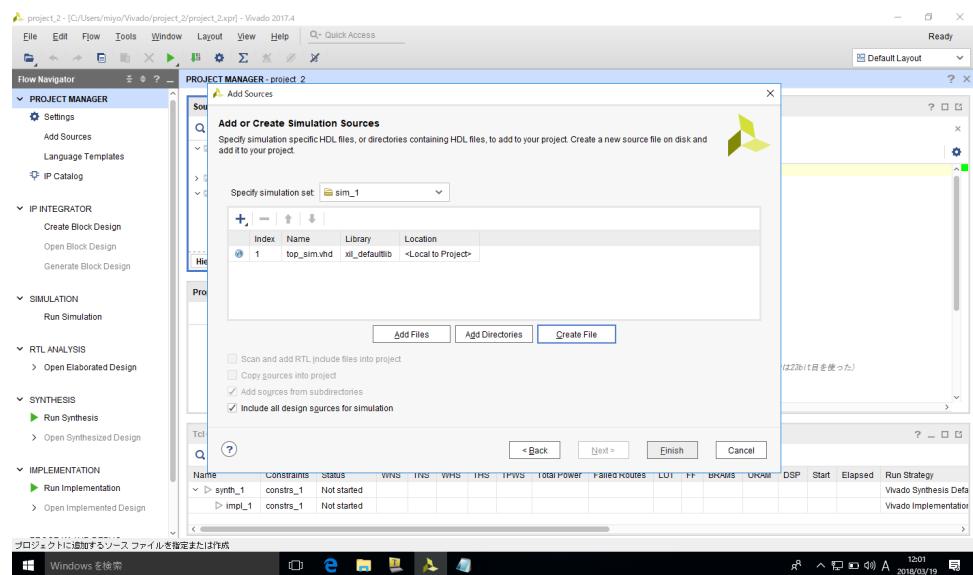


図 9 リストに追加された

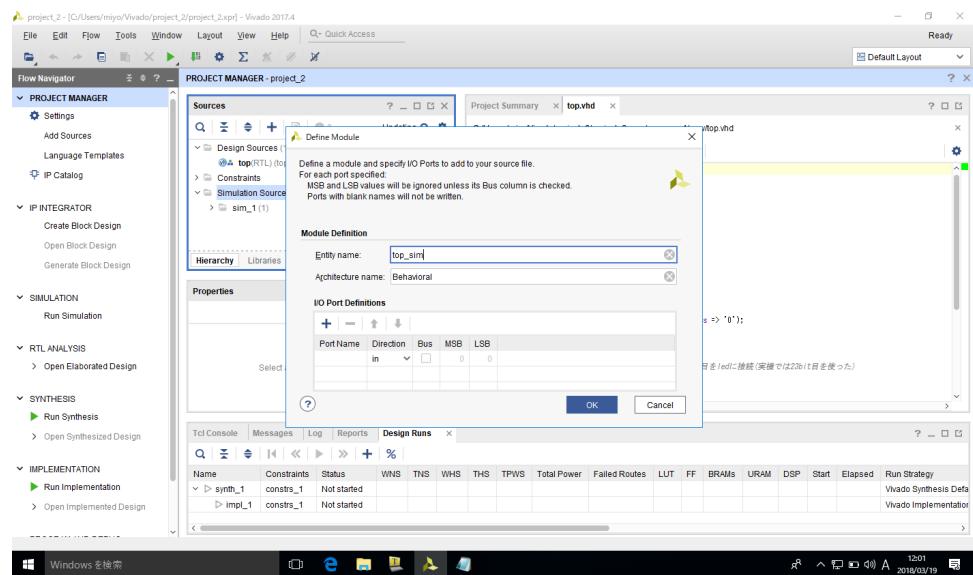


図 10 ポートの指定ダイアログ。今回は何もせずに OK をクリック

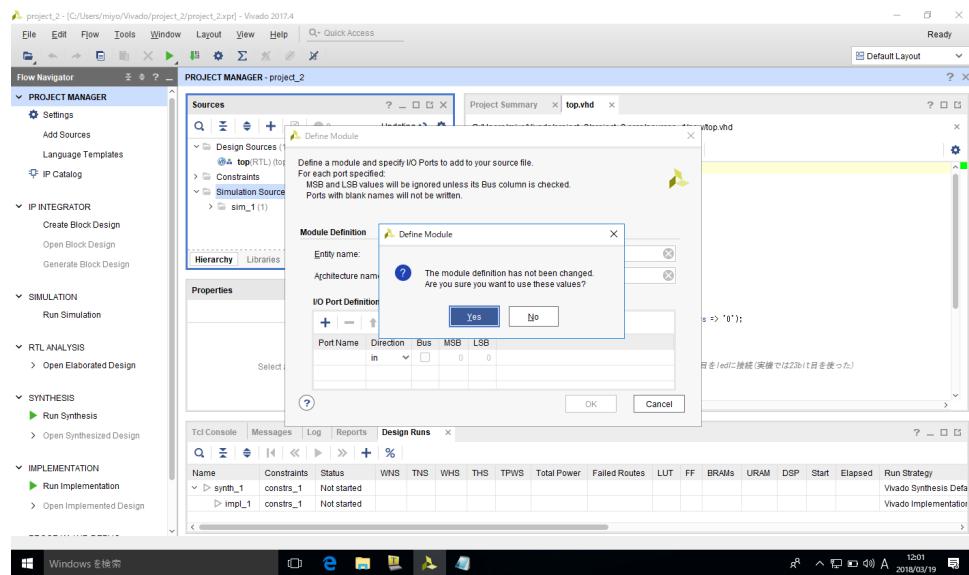


図 11 変更ないことを確認されるので Yes をクリックして、作業ステップをすすめる

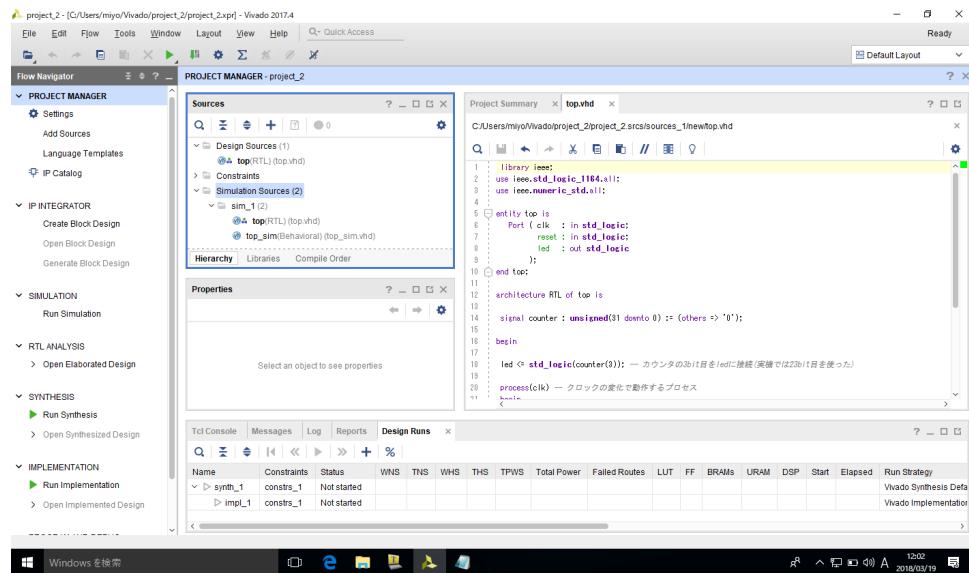


図 12 シミュレーション用のモジュールがプロジェクトに追加された

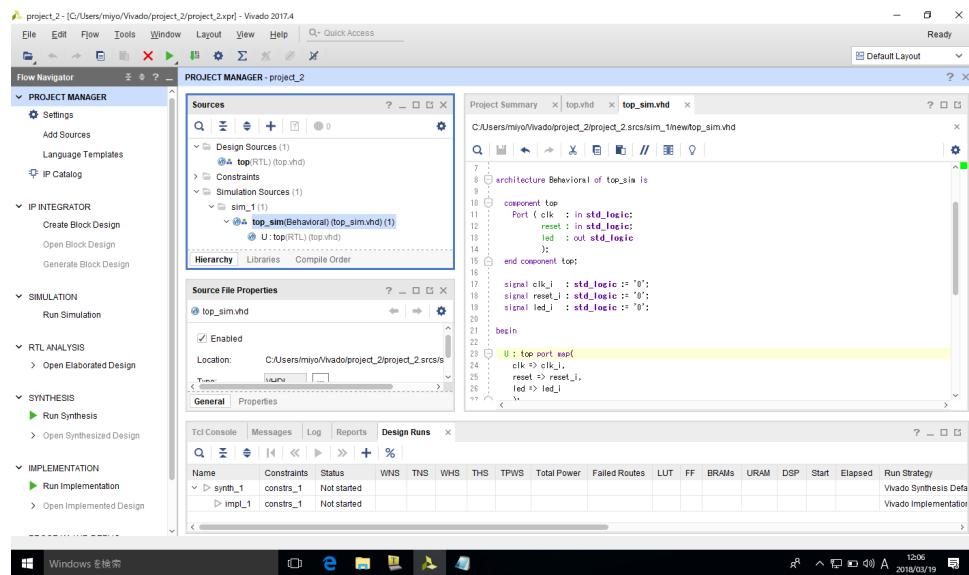


図 13 シミュレーションコードを top_sim に反映したところ。top_sim から top のモジュールが U という名前でインスタンス化されている様子が階層化されて表示される。

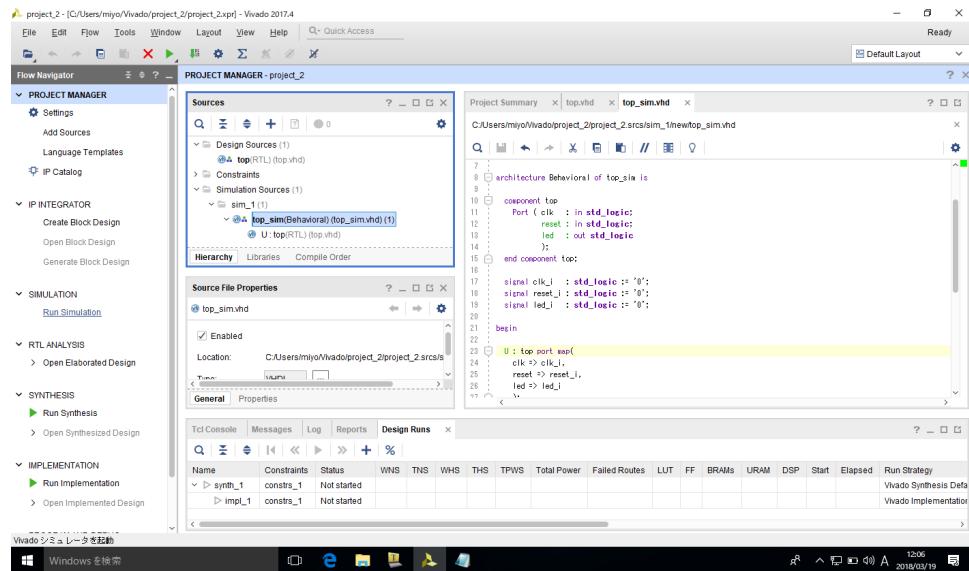


図 14 Flow Navigator の Run Simulation でシミュレーションの開始

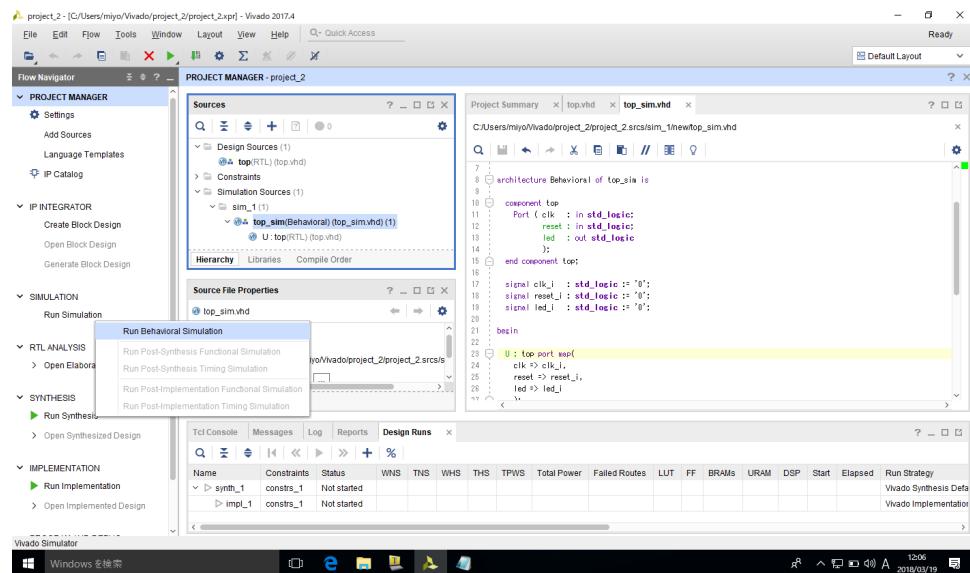


図 15 Run Behavioral Simultaion を選択

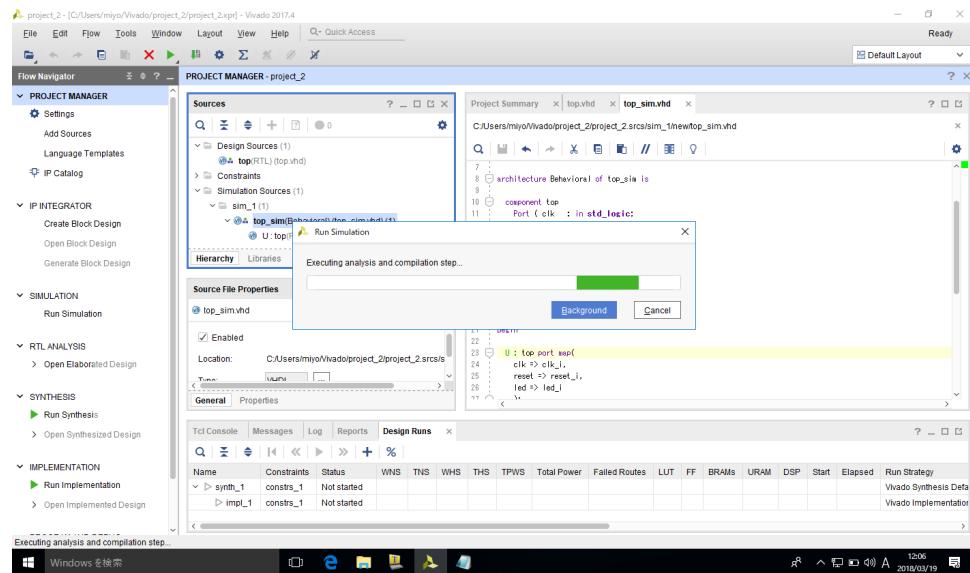


図 16 シミュレーション開始には少し時間がかかる

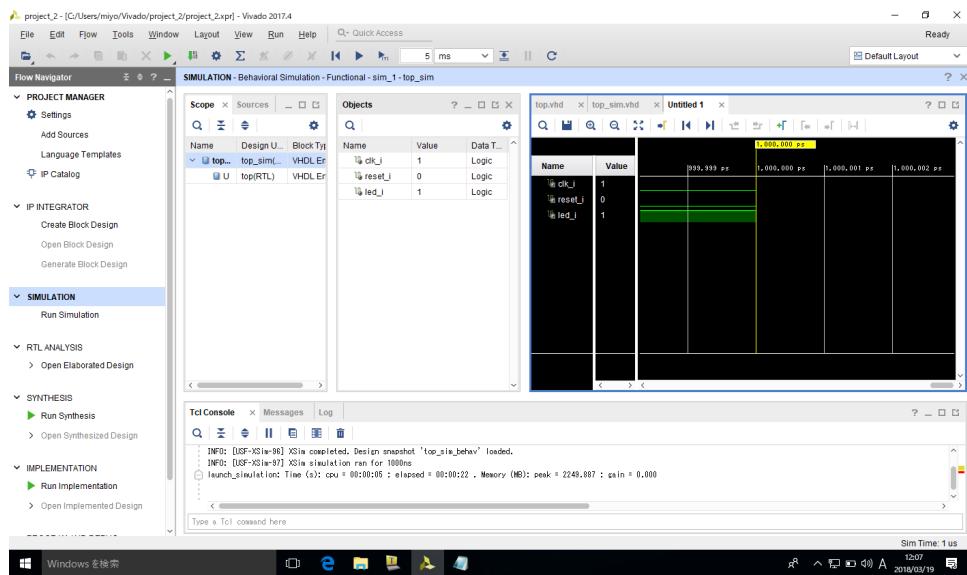


図 17 シミュレーションが開始された

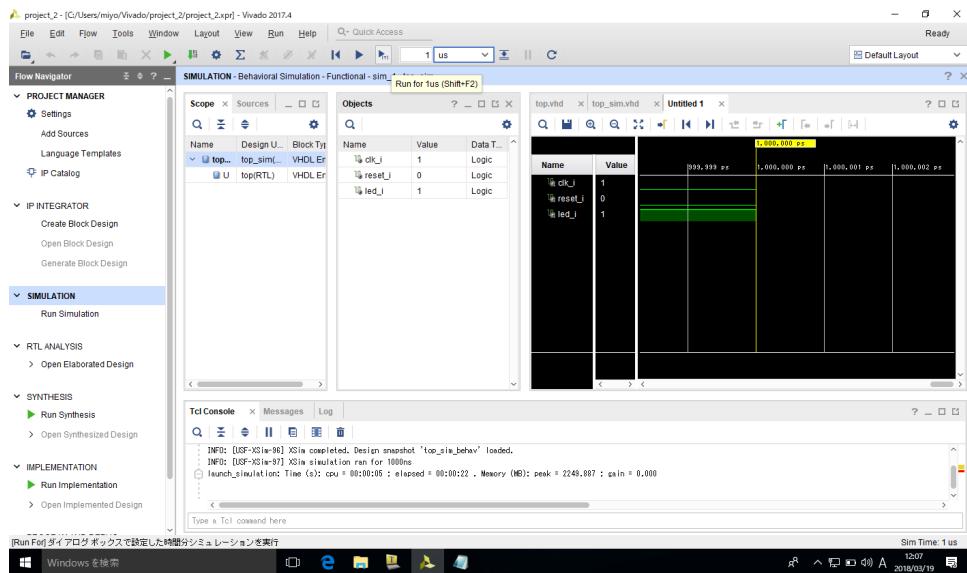


図 18 シミュレーション時間を指定して、シミュレーションステップをすすめる

指定した時間のシミュレーションが終わるとストップします。いつまでもストップしない場合には、一時停止アイコンをクリックして、強制的に止めることができます。

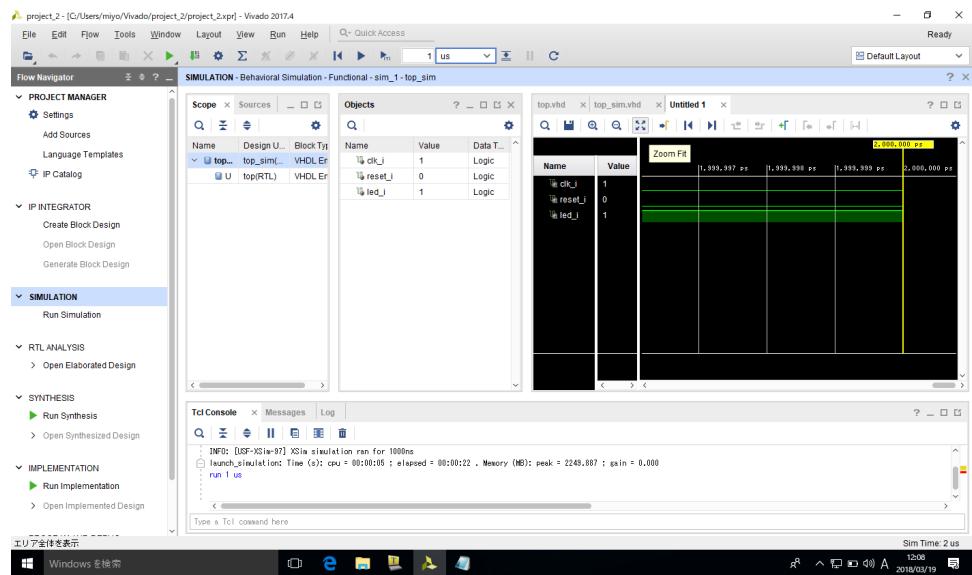


図 19 指定した時間のシミュレーションが終了

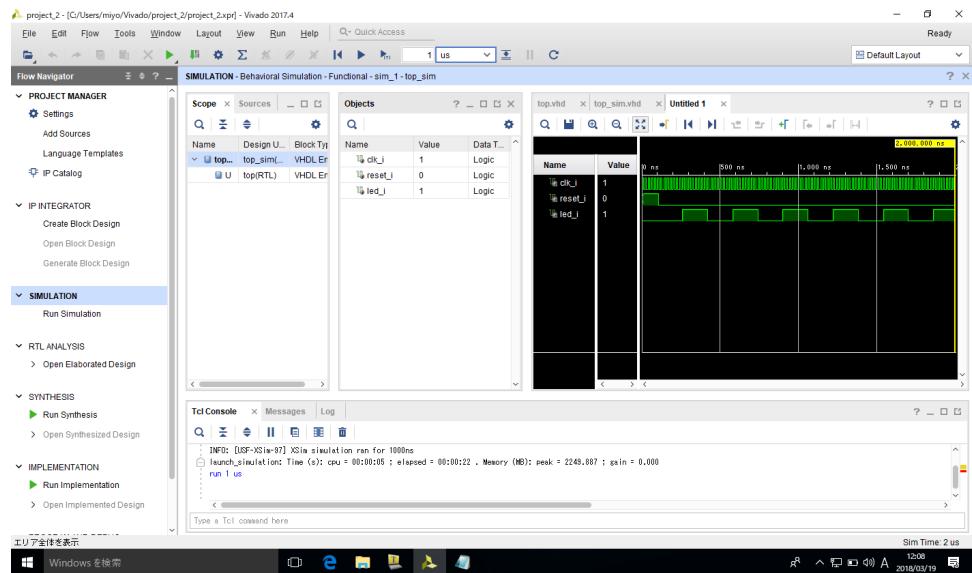


図 20 波形表示画面に全シミュレーション結果を表示 (FIT) させたところ

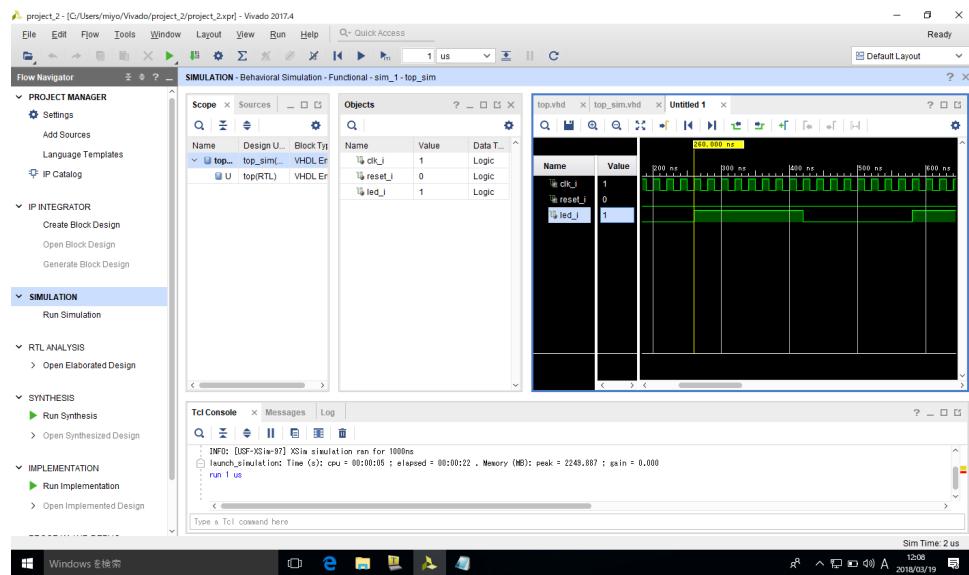


図 21 一部分を拡大した様子。3bit 目を led に接続しているため 8 クロック毎に led が ON/OFF している様子がみてとれる

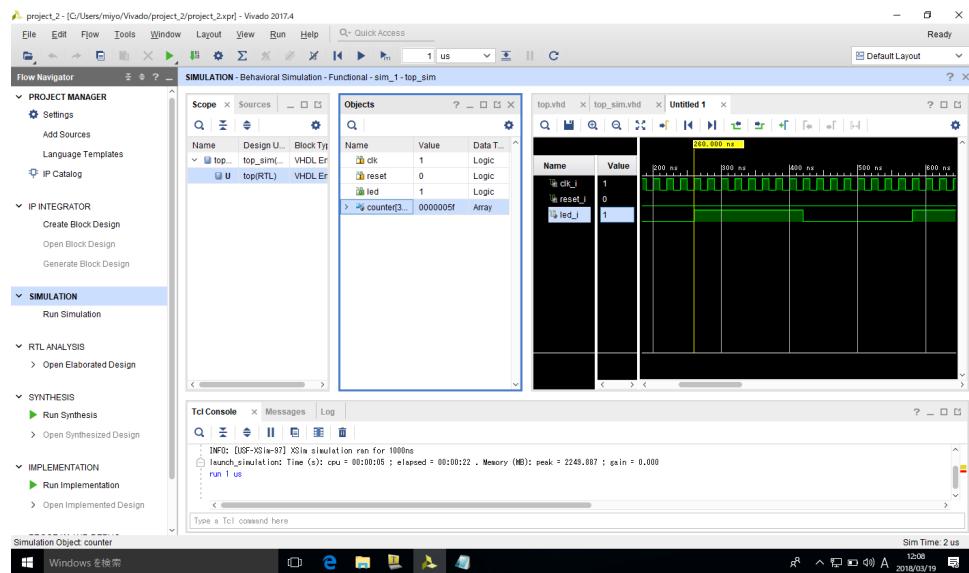


図 22 内部モジュール(今回でいうと top の中身を確認することもできる)

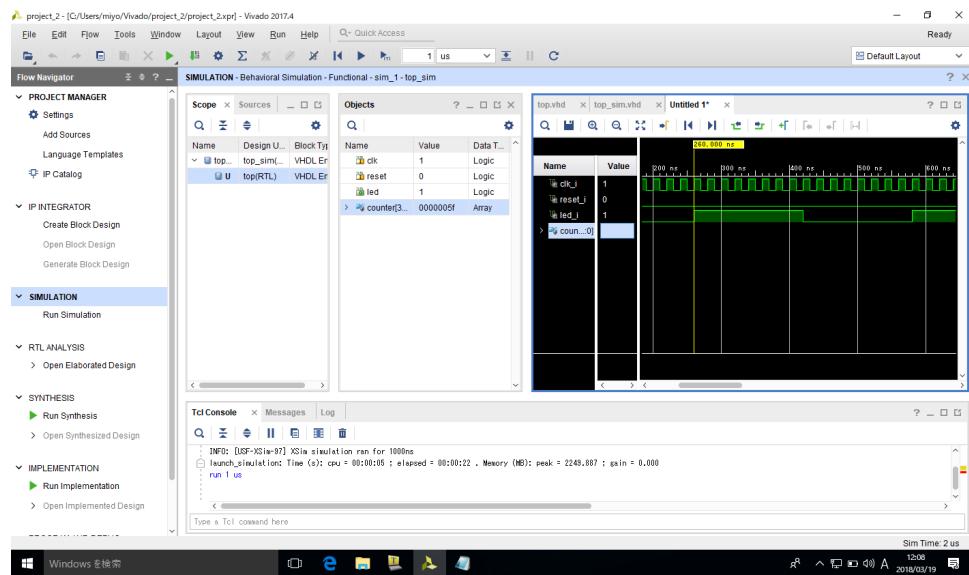


図 23 top モジュールの counter を波形表示画面に追加。ただし、内部の値の多くは非表示状態では保存されていないので、そのままでは値の変化を確認できない

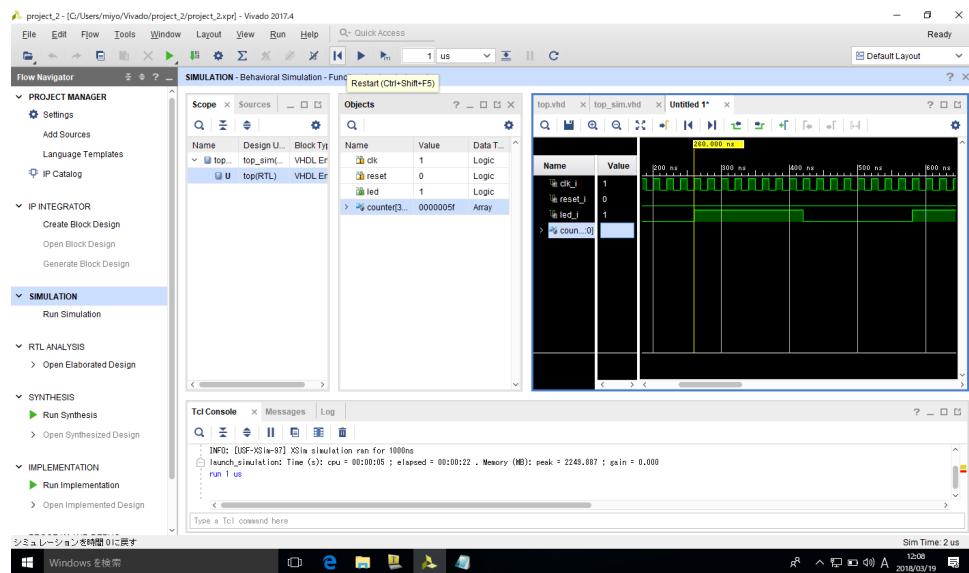


図 24 シミュレーションを一度リセット

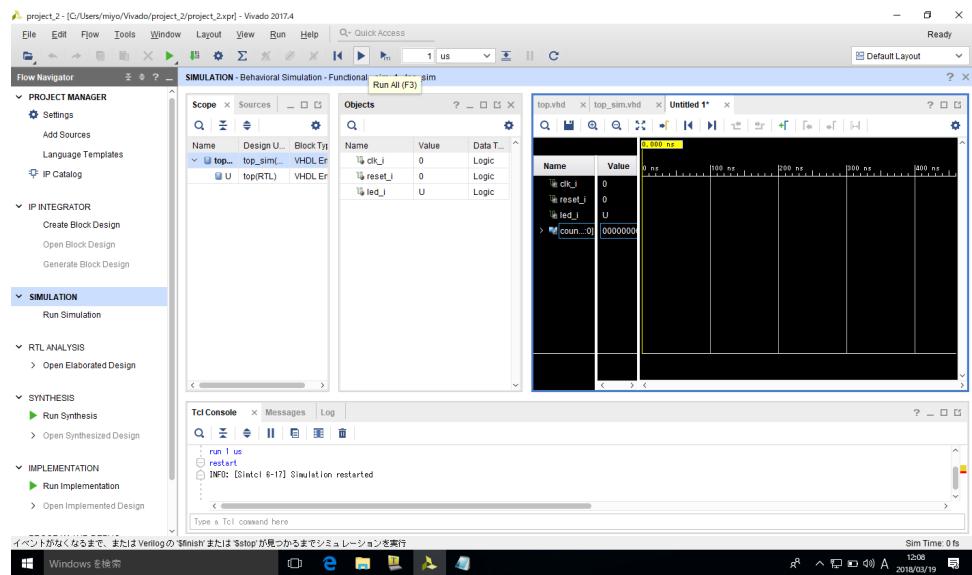


図 25 再度シミュレーション。今度は時間指定なくシミュレーションしてみる

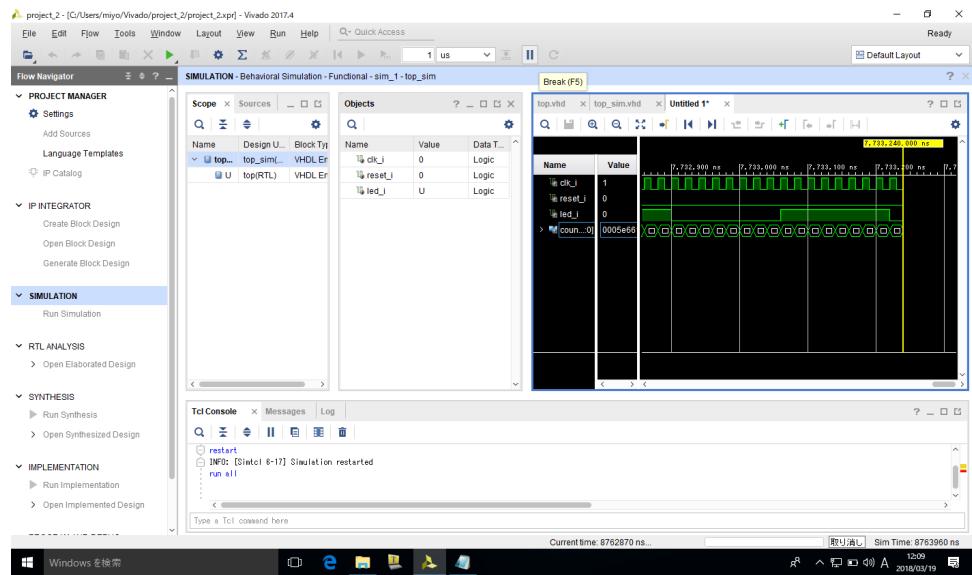


図 26 いつまでも終わらないので一時停止アイコンでシミュレーションをストップ

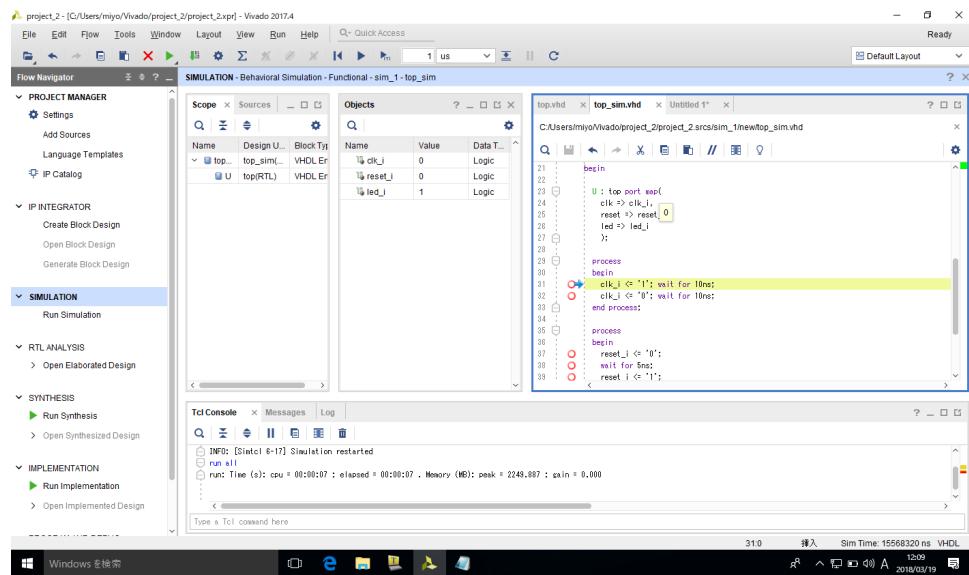


図 27 ストップした箇所のソースコードが表示される

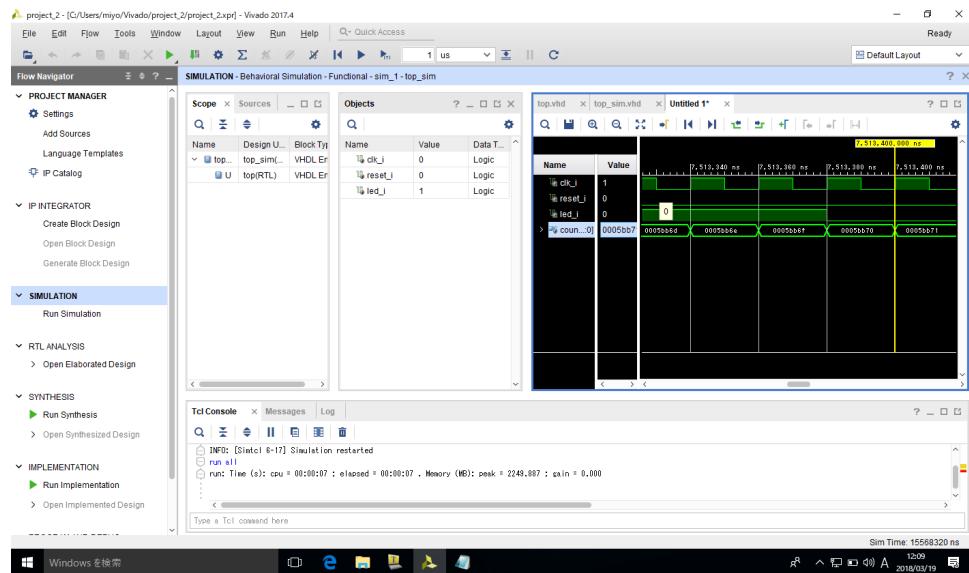


図 28 波形を確認してみると、内部の counter が clk にあわせてインクリメントしていること、counter の 3bit 目が led の '0'/'1' と同じであることが確認できる

5 課題

1. 点滅の間隔を変えてみたときの様子をシミュレーションしてみよう
2. reset を適当なタイミングで変化させてみて、counter や led の振る舞いを観察してみよう

基本実験

かんたんな実験を通じて FPGA でのハードウェア作りの基礎力を手に入れましょう。

1 はじめに

いくつかの簡単な HDL コードを書いて、FPGA でのハードウェア作りの基礎力を手に入れましょう。実際に HDL コードを書き、シミュレーションして、実機動作を確認することは、より複雑なハードウェアを設計するための第一歩です。

2 ILA を使って FPGA 内部の信号を観測する

FPGA は、FPGA の中で回路がどのように動作しているのかを知るための ILA(Internal Logic Analyzer) という仕組みを持っています。実験コードを書きはじめる前に、この ILA の使い方を学んでみましょう。

2.1 L チカで ILA の動作を学ぶ

三度目の登場ですが、L チカで ILA の動作を学びましょう。第 4 章で利用した `project_2` を使用します。

2.2 準備

ILA を使ってデバッグするために、`top` モジュールのソースコードを次のように書き変えてください。

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 entity top is
6     Port ( clk      : in std_logic;
7             reset   : in std_logic;
8             led      : out std_logic
9         );
10 end top;
11
12 architecture RTL of top is
13
14     attribute mark_debug : string; -- (1) 追加
15     signal counter : unsigned(31 downto 0) := (others => '0');
16     attribute mark_debug of counter : signal is "true"; -- (2) 追加
17
18 begin
19
20     -- カウンタの3bit目をledに接続(実機では23bit目を使った)
21     led <= std_logic(counter(3));
22
23     process(clk) -- クロックの変化で動作するプロセス
24     begin
25         if rising_edge(clk) then -- クロックの立ち上がりであれば
26             if reset = '1' then
27                 counter <= (others => '0');
28             else
29                 counter <= counter + 1; -- カウンタをインクリメント
30             end if;
31         end if;
32     end process;
33
34 end RTL;
```

図1 ILA 実験用に変更したリストの例

(1)と(2)が追加ポイントです。 (1)で `mark_debug` という attribute を利用することを宣言し、(2)で `counter` に `mark_debug` という attribute を付与しています。 attribute は、ツールに対する指示子です。 Vivado では、`mark_debug` を付与した信号はデバッグ対象の可能性があるとして特別扱いします。

コードが準備できたら、一度合成してピン配置まで終わらせてしまいましょう。

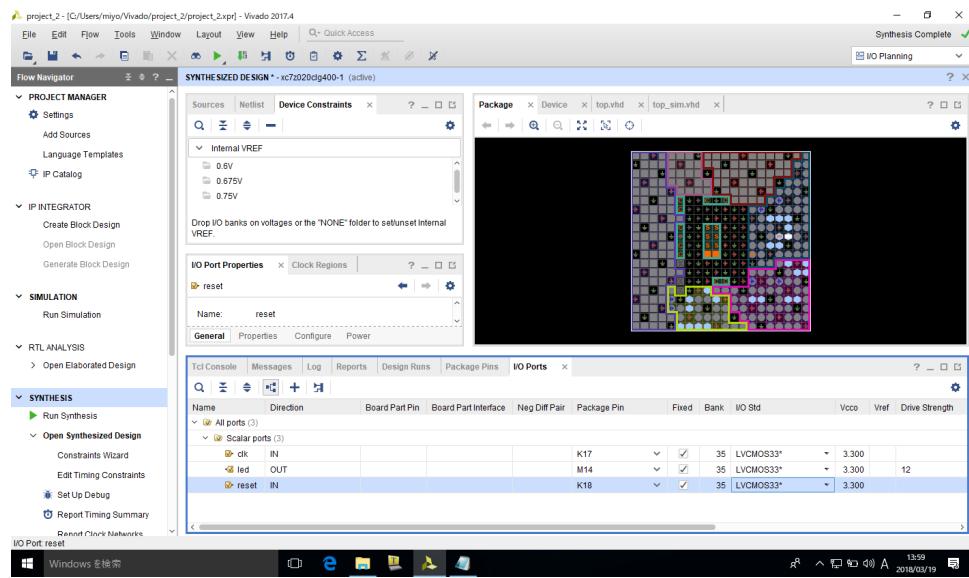


図2 I/O Planning で clk, reset, led のピン配置を決定する

2.3 ILA のセットアップと利用方法

準備ができたら ILA を追加して、その動作の様子を確認してみましょう。

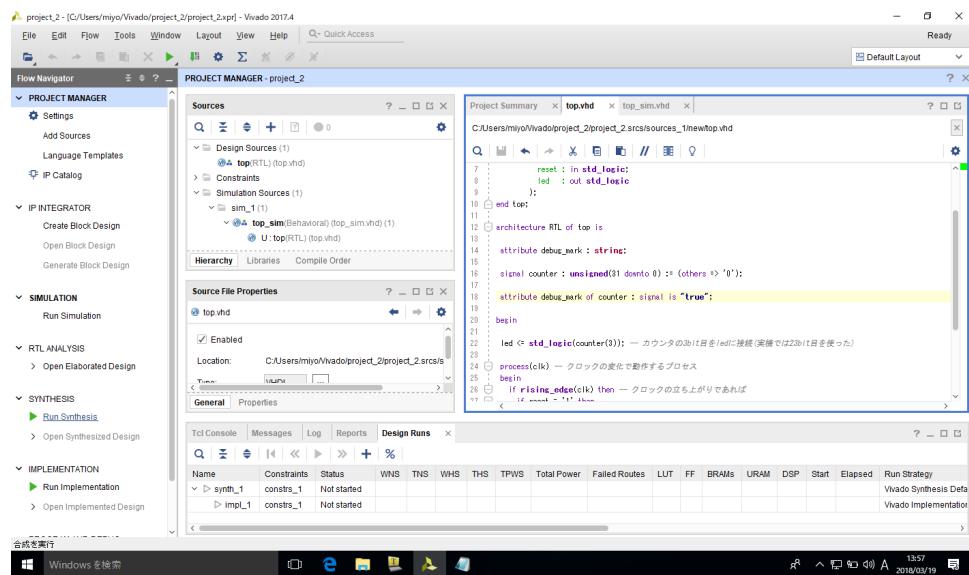


図3 一度合成する

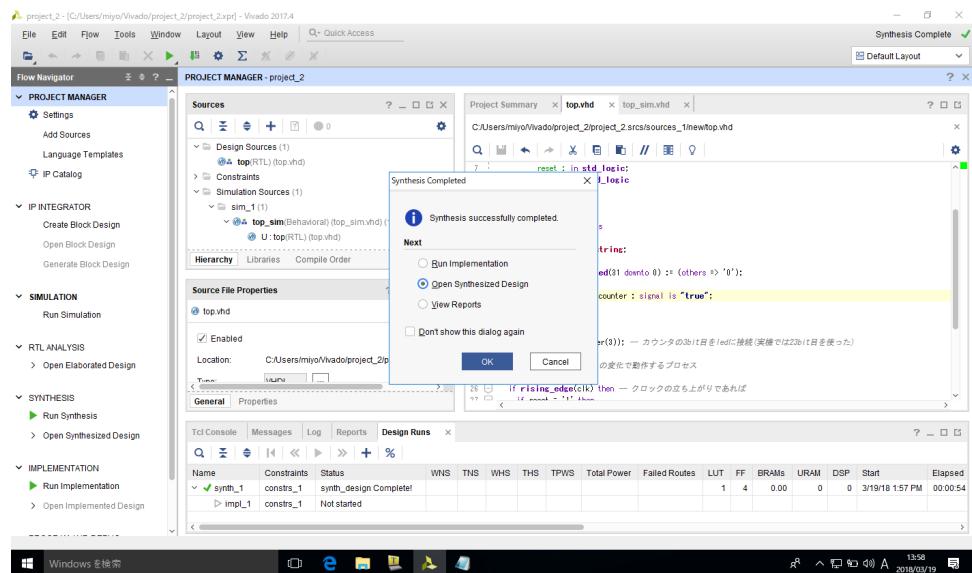


図4 合成が終わったら Open Synthesized Design で合成結果を開く

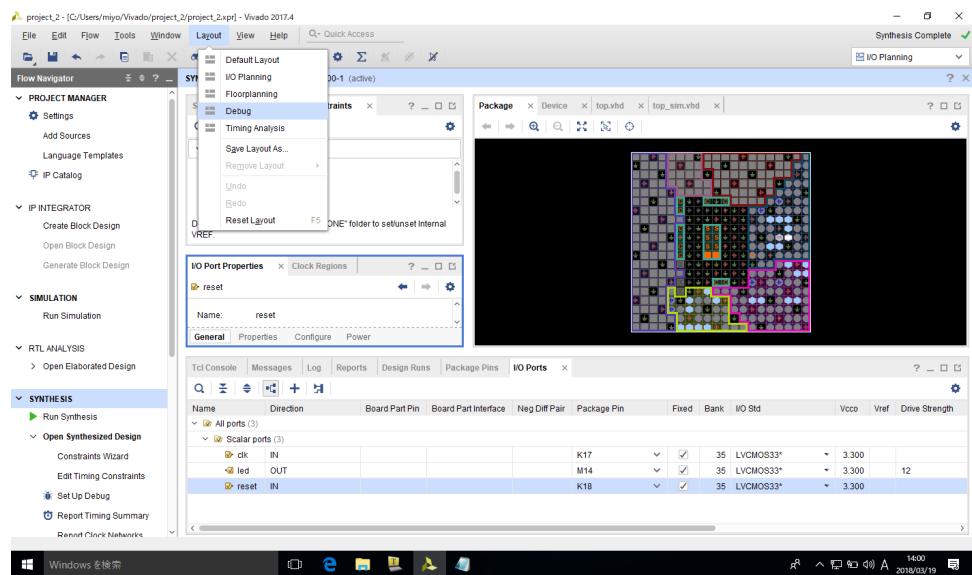


図5 Layout メニューの Debug をクリックしてデバッグビューに変更する

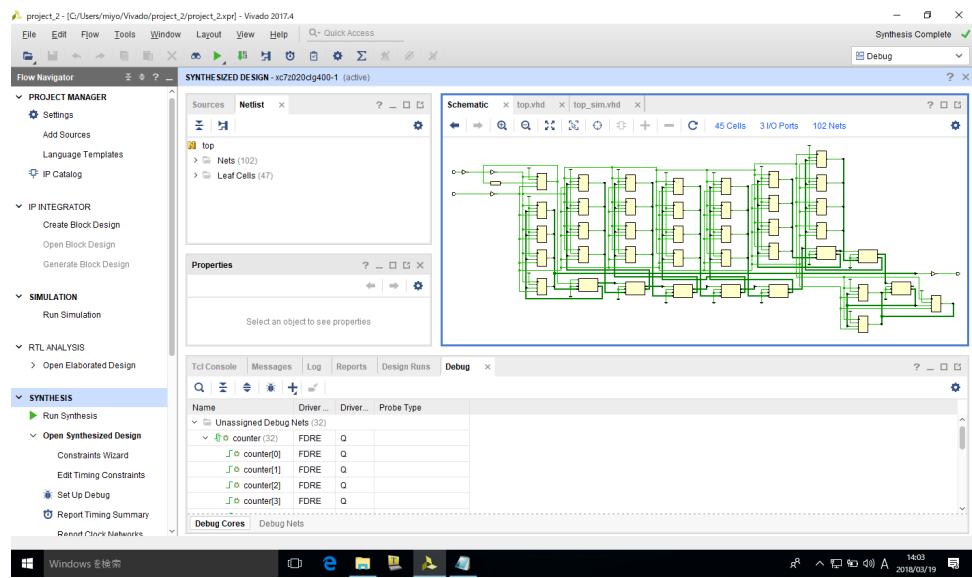


図 6 ILA 設定用の画面

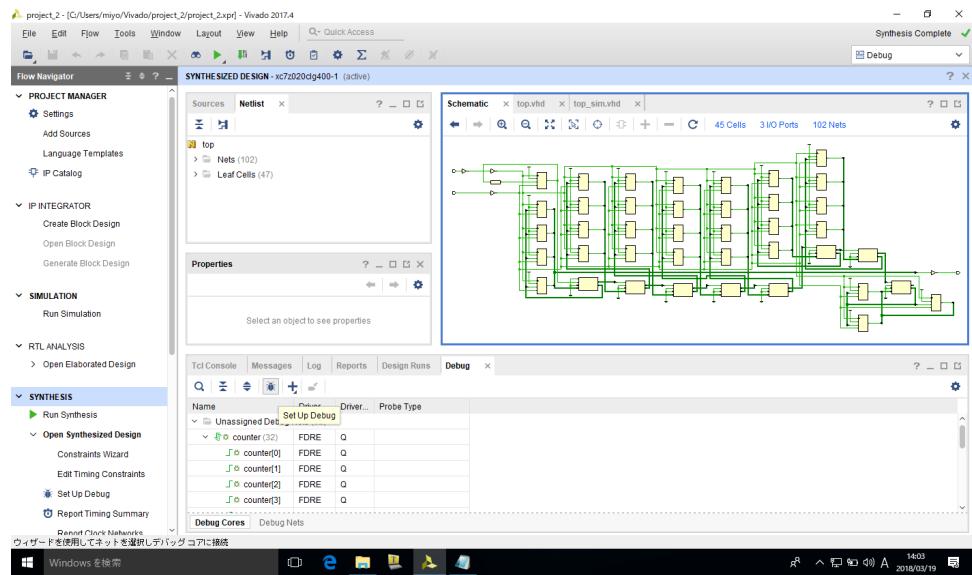


図 7 下にある虫みたいなアイコンをクリックして ILA 設定用のウィザードを開く

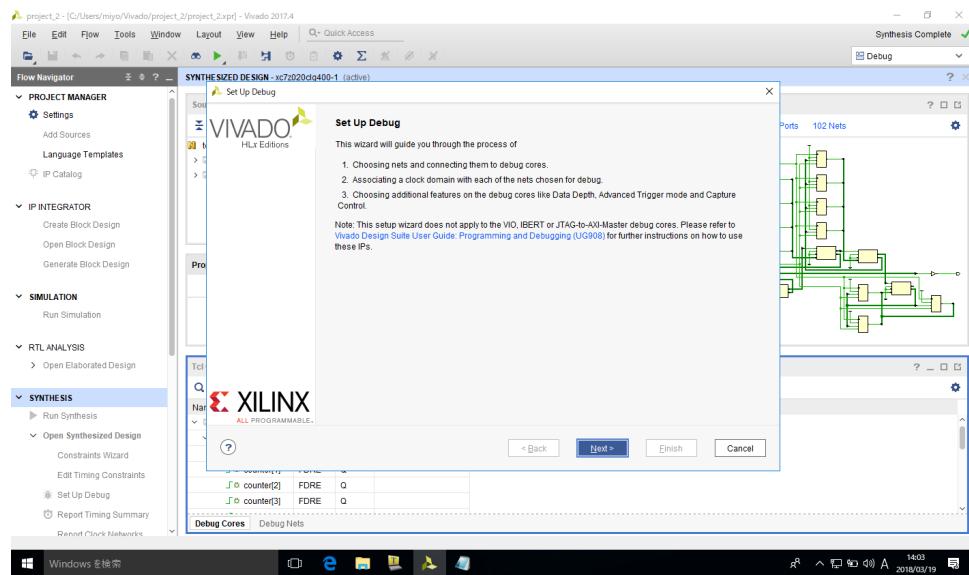


図8 ILA 設定用ウィザードの開始

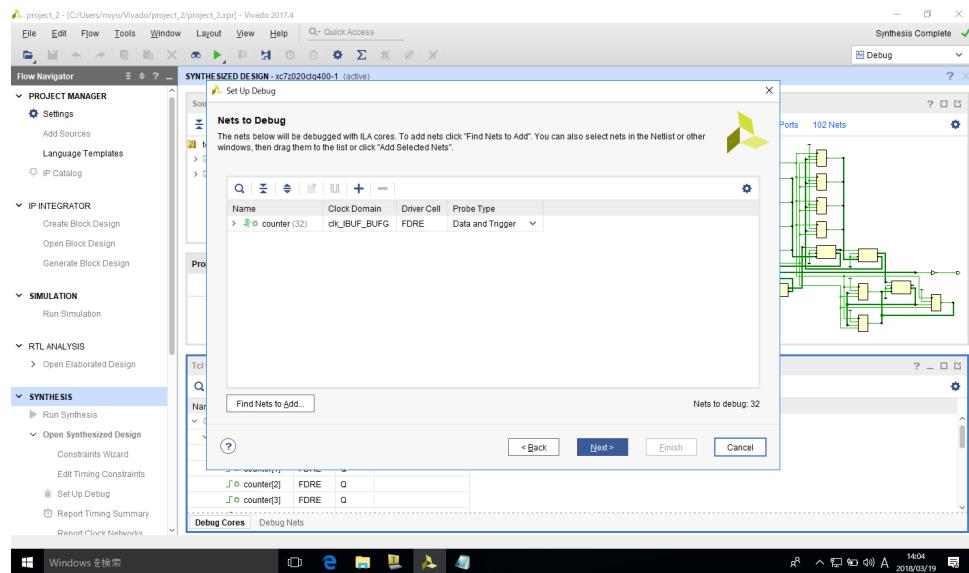


図9 mark_debug を付与した counter がリストに追加されているので、そのまま Next ですすむ。ここで新たに ILA による観測対象を追加したい場合には + アイコンをクリックすると信号を選ぶことができる。逆にリストにある信号を対象から取り除きたい場合には、取り除きたい信号を選択して - をクリックする

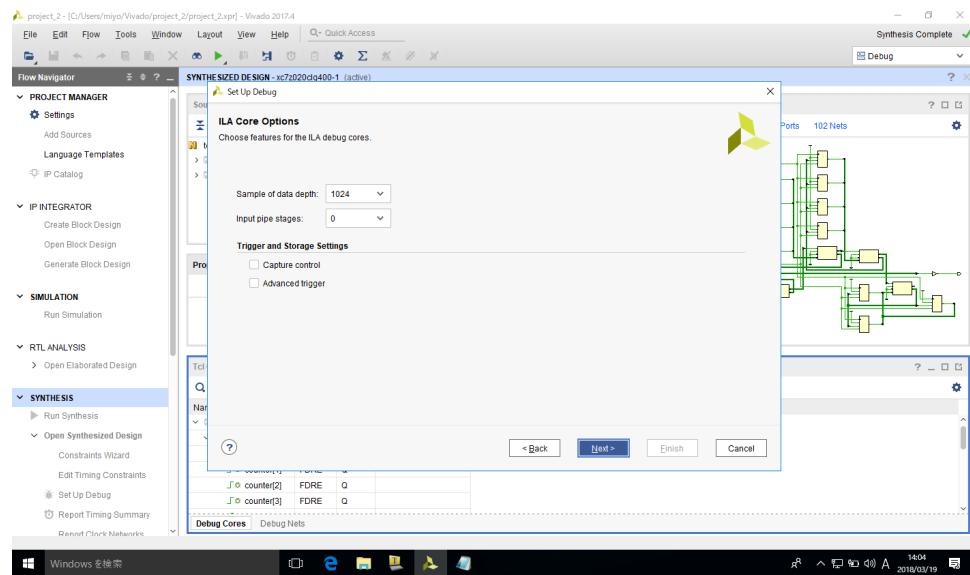


図 10 ILA で取得するデータ数の設定など。今回はそのままにして Next ですすむ

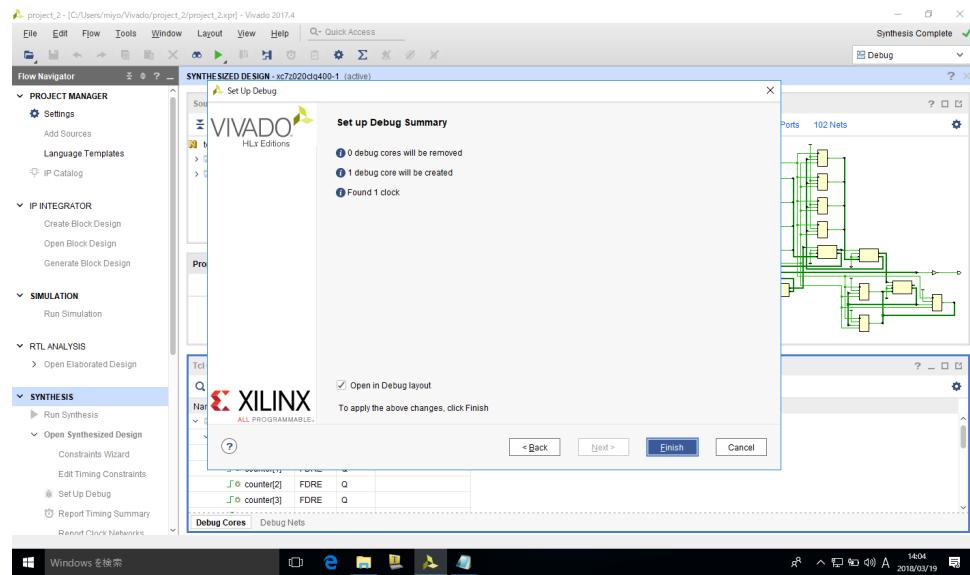


図 11 サマリの表示。Finish で完了

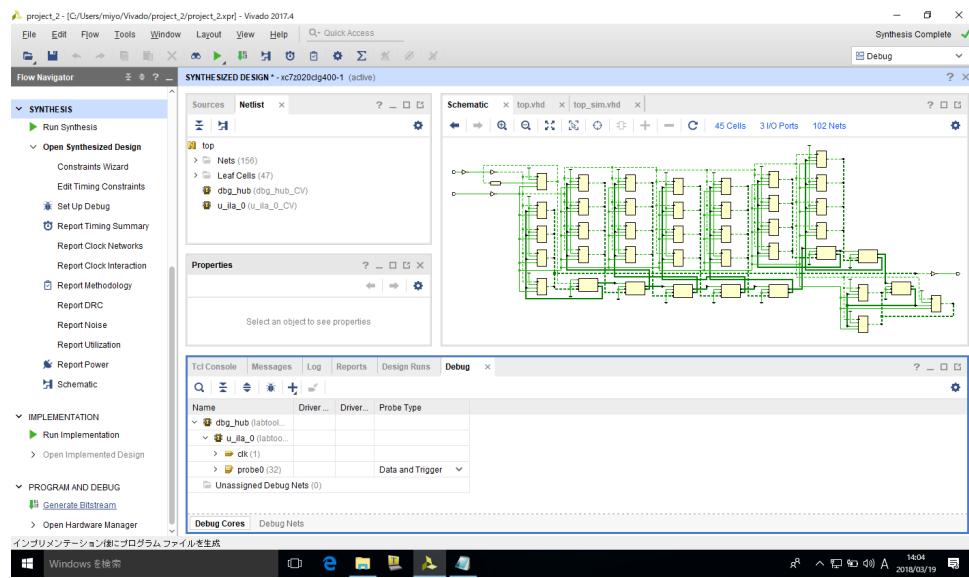


図 12 ウィザードが閉じて ILA の設定は完了。ILA が追加できていることがわかる。あとは、Generate Bitstream でビットファイルを作成すればよい。

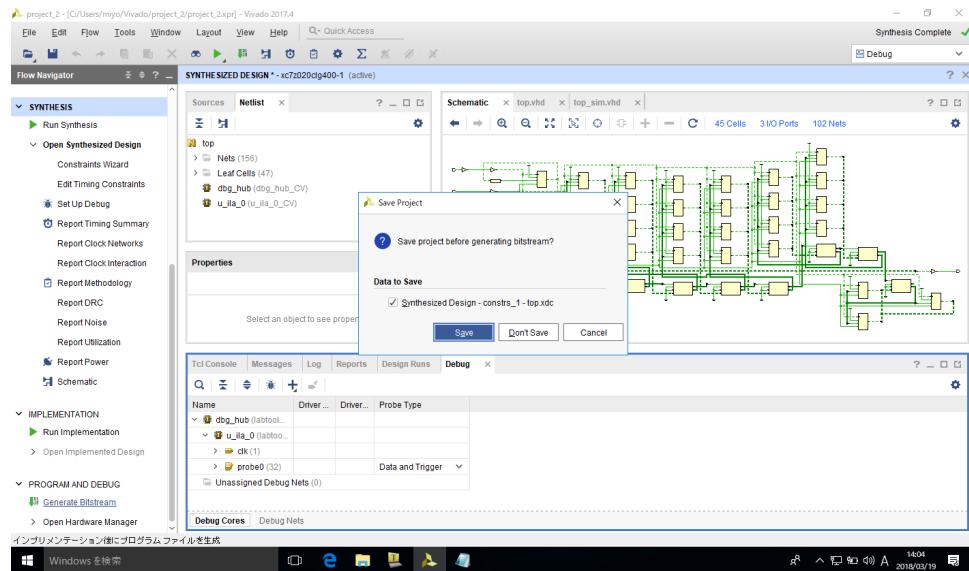


図 13 ILA の設定情報を xdc ファイルに保存してよいかの確認。Yes で次のステップにすすむ。

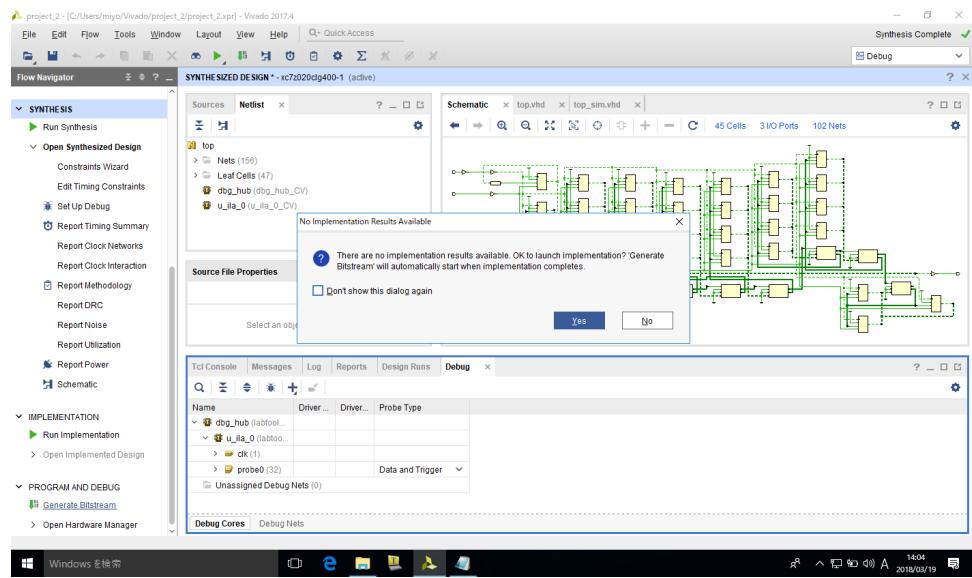


図 14 Generate bitstream の前に依存する他のタスクを実行します、という確認ダイアログ。Yes で次のステップへ

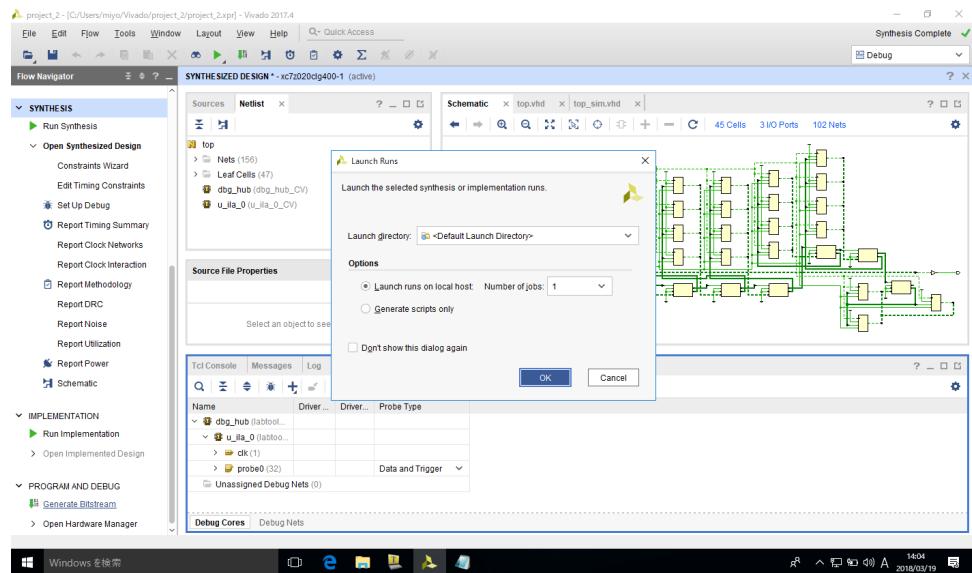


図 15 合成と配置配線の開始

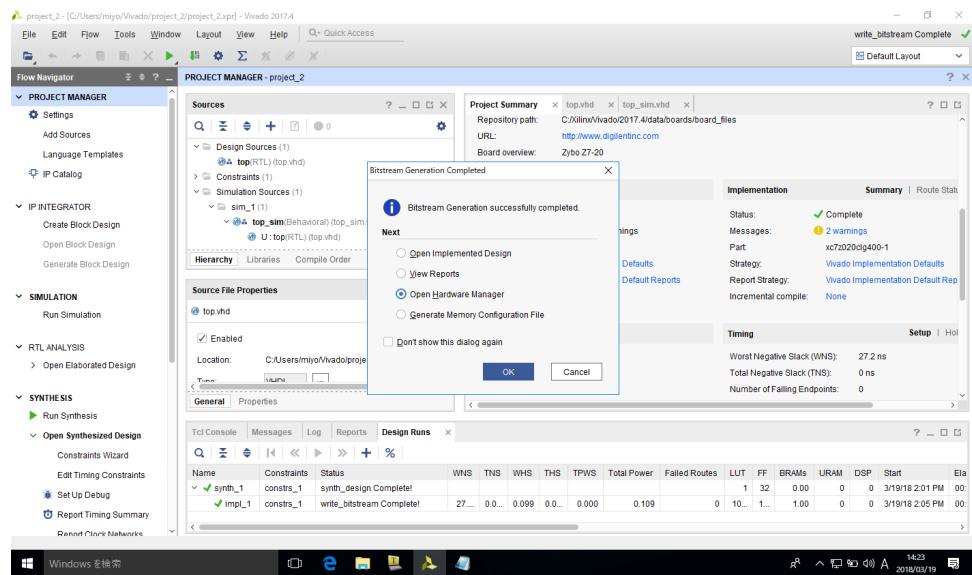


図 16 無事に合成と配置配線が終了しビットファイルができるところ。Open Hardware Manager を選択して OK をクリックすることで、ハードウェアマネージャの起動の手間を省くことができる

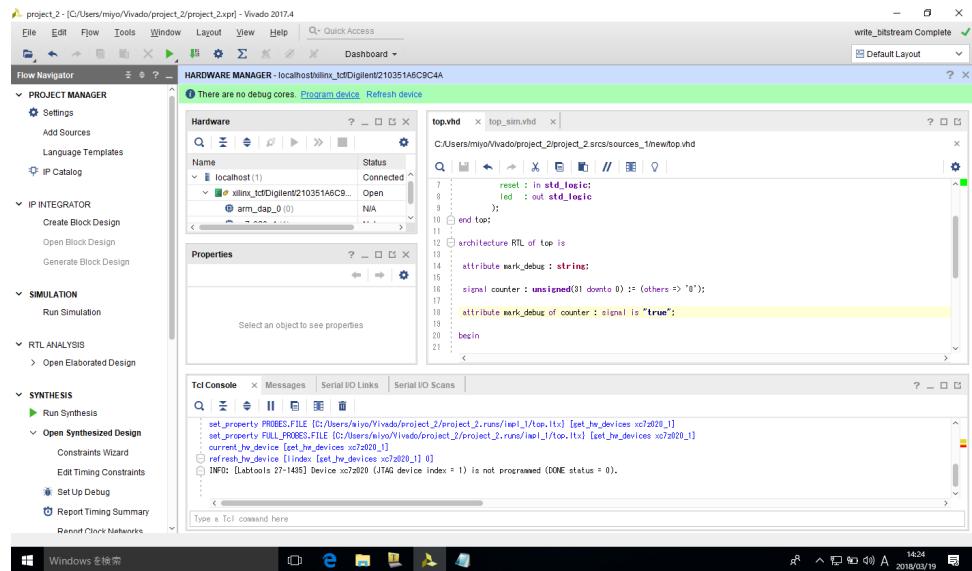


図 17 FPGA とパソコンを USB ケーブルで接続して Auto connect で認識させた後、Program Device をクリック

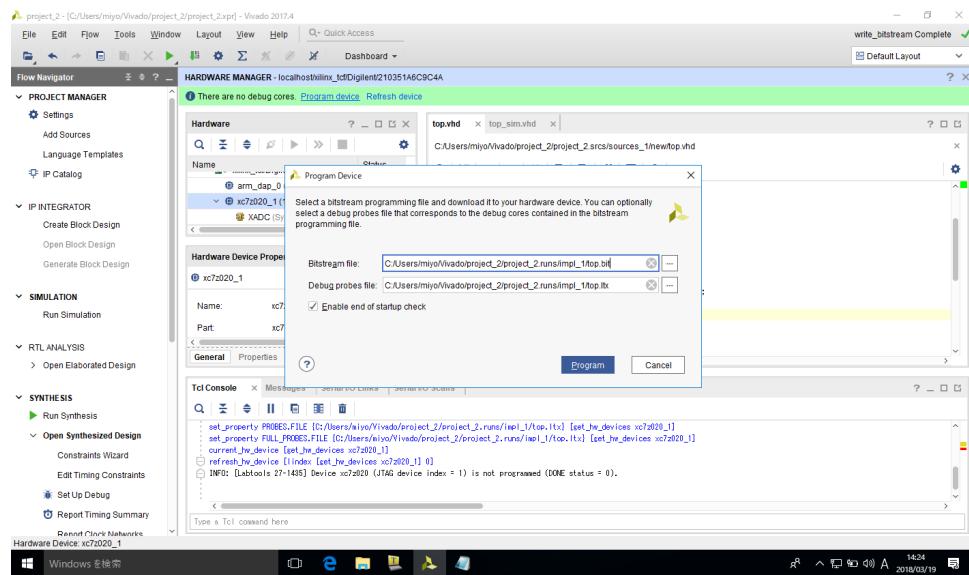


図 18 bit ファイルは FPGA に、ILA のパソコン側の定義ファイルである ilx は Vivado に読み込ませる。

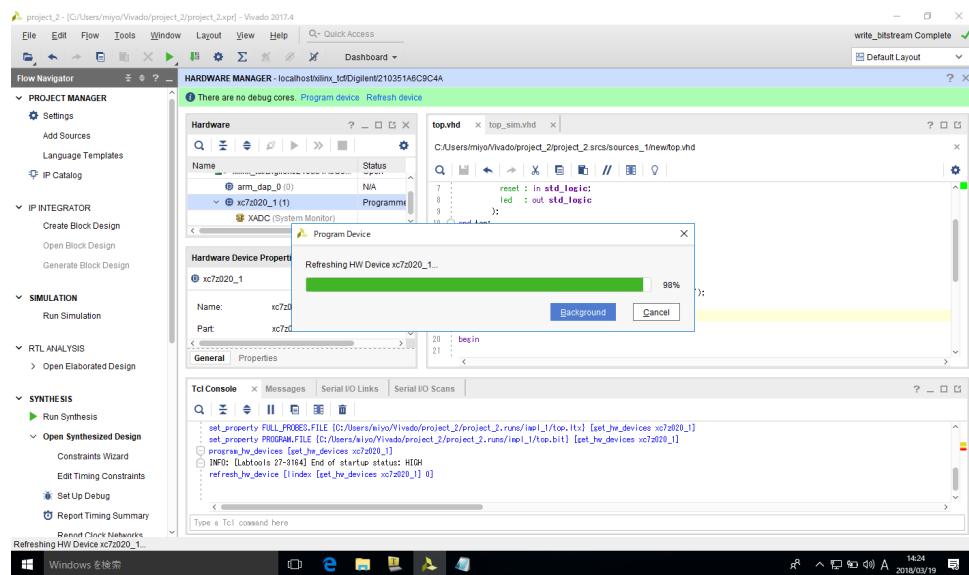


図 19 書き込み中

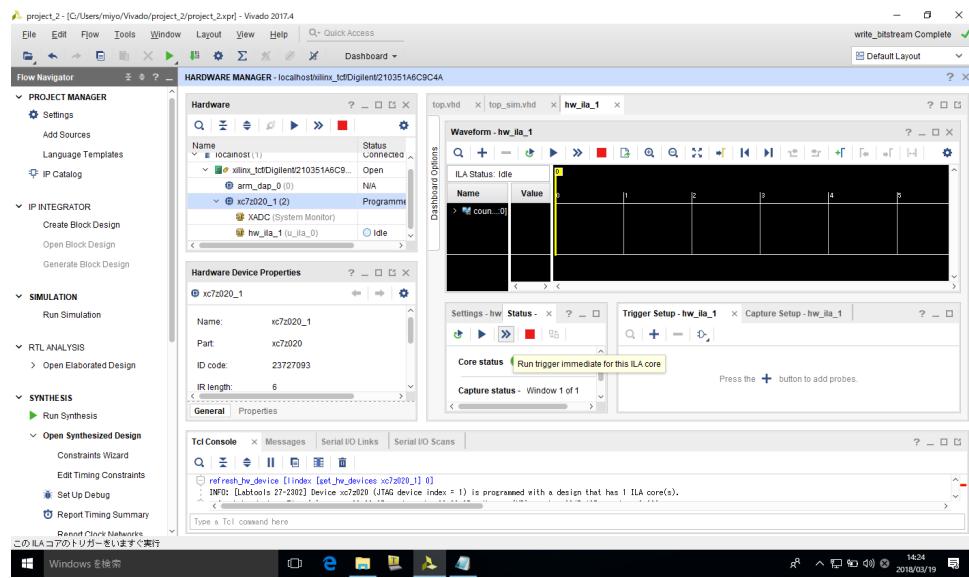


図 20 FPGA へのダウンロードが終了した。また、ILA による動作のモニタ画面が表示された

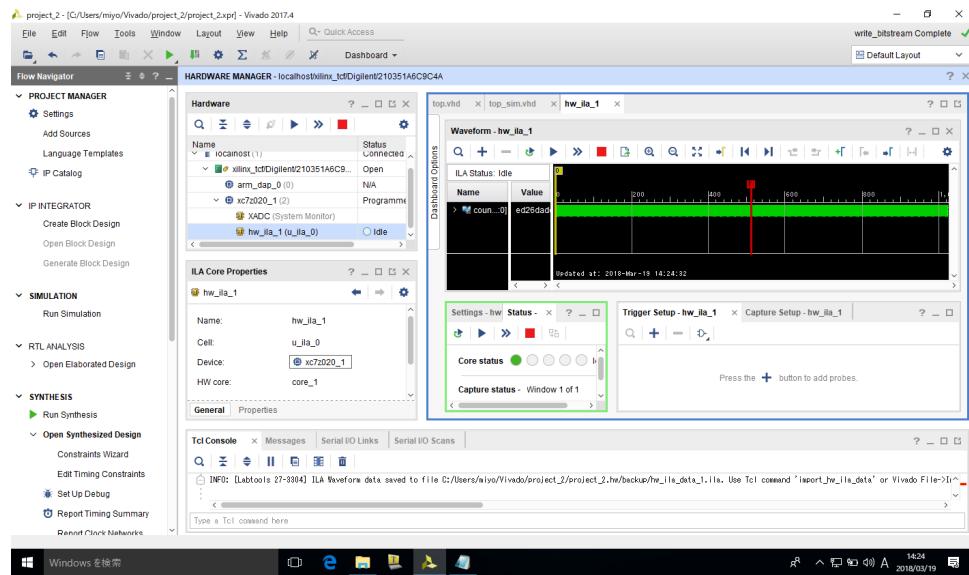


図 21 二重矢印のアイコンをクリックすると、その時点での値をキャプチャしてくれる

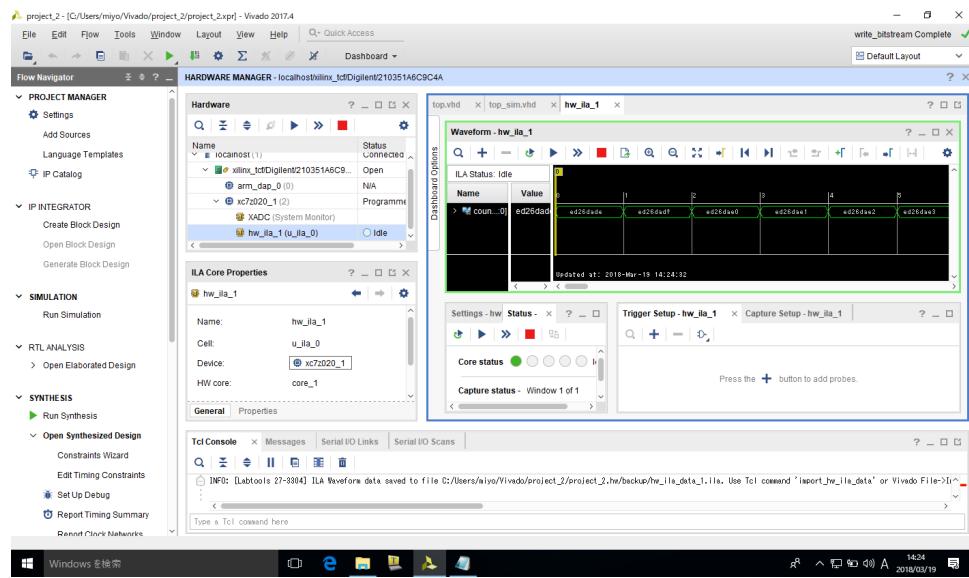


図 22 虫眼鏡アイコンで拡大すると、値が 1 ずつ増えていることが確認できる

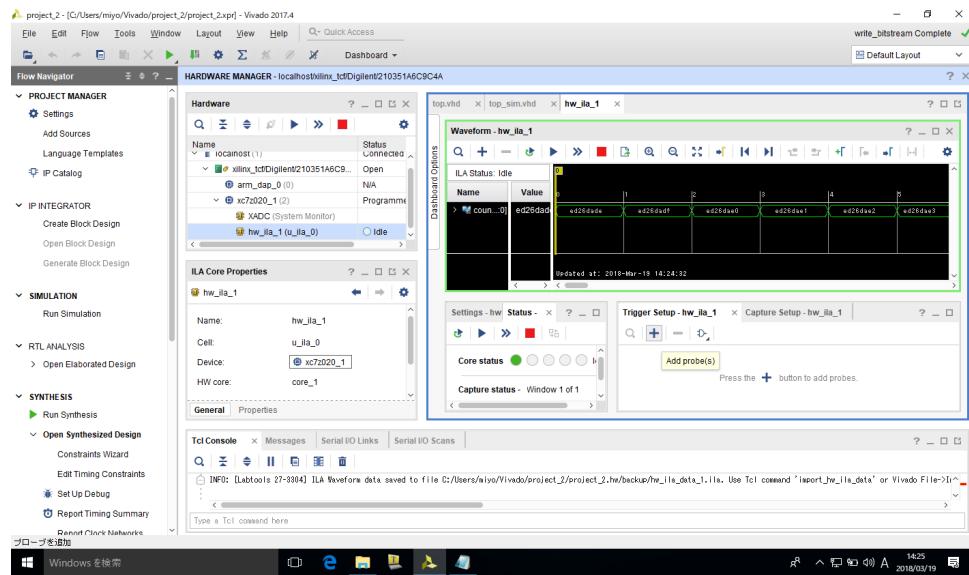


図 23 実機デバッガでは値をキャプチャする条件(トリガ条件)を指定する

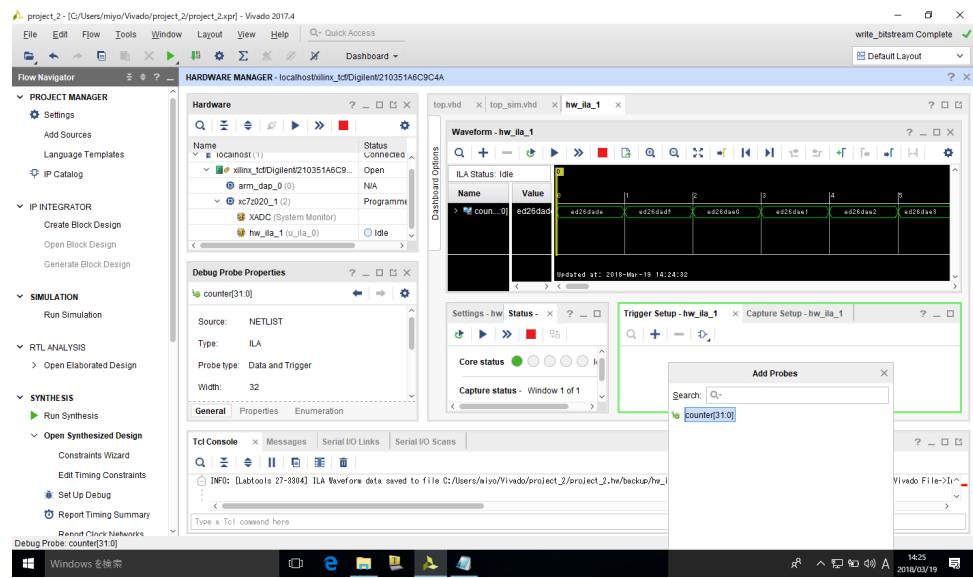


図 24 counter の値をトリガ条件に使用することとする

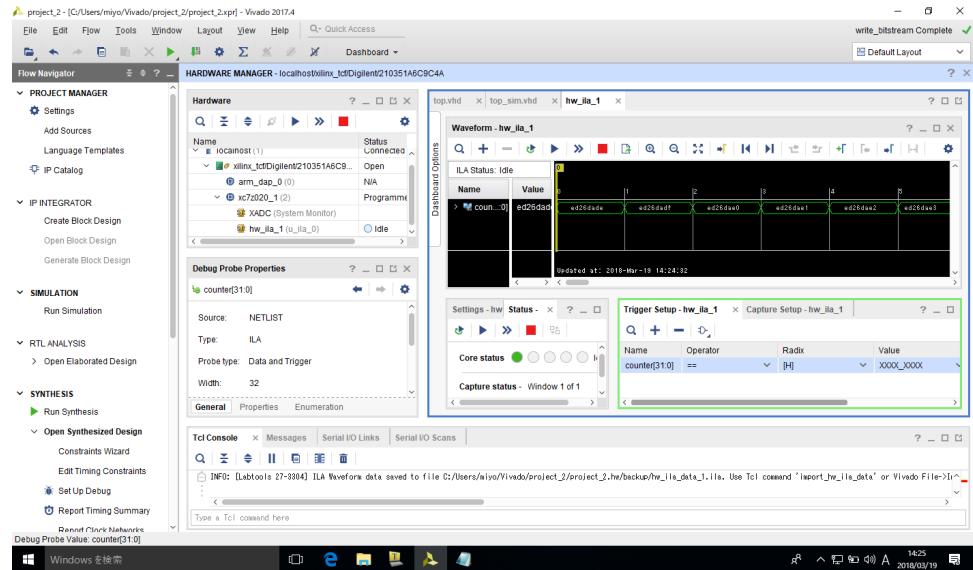


図 25 counter の値がトリガ条件として登録された

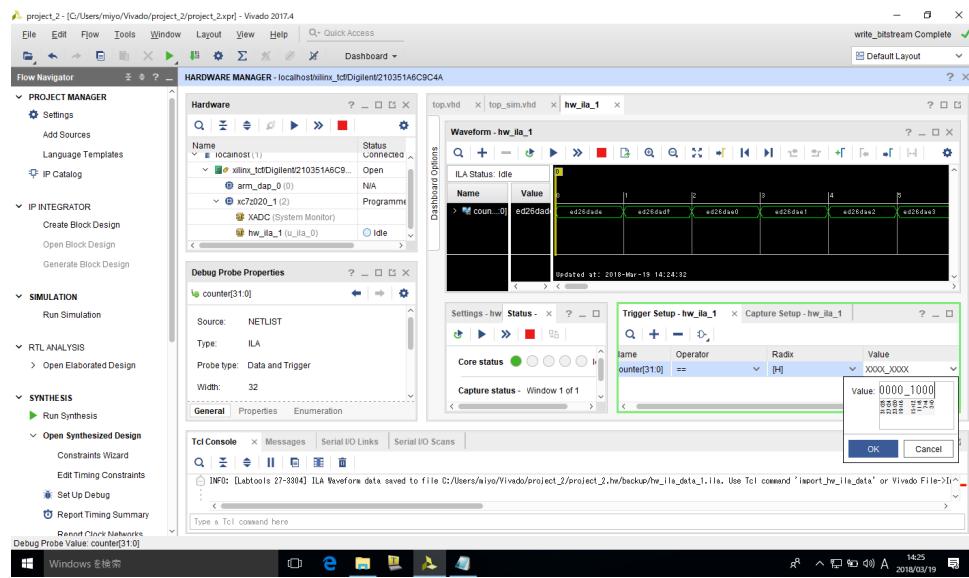


図 26 counter が 00001000 になった時点でキャプチャするように設定。トリガ値を指定したら三角アイコンでキャプチャを開始する

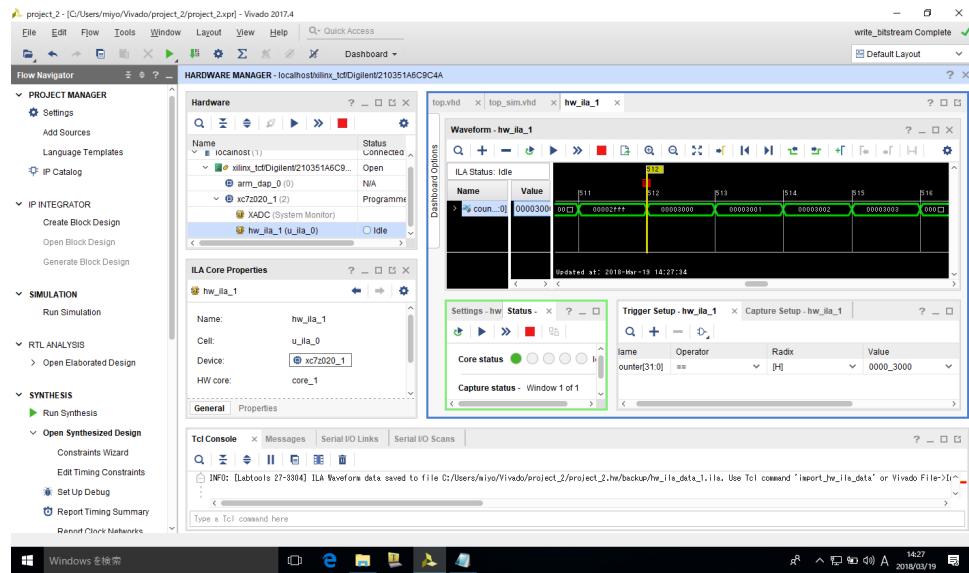


図 27 counter が 00001000 になった時点のデータをキャプチャすることができた

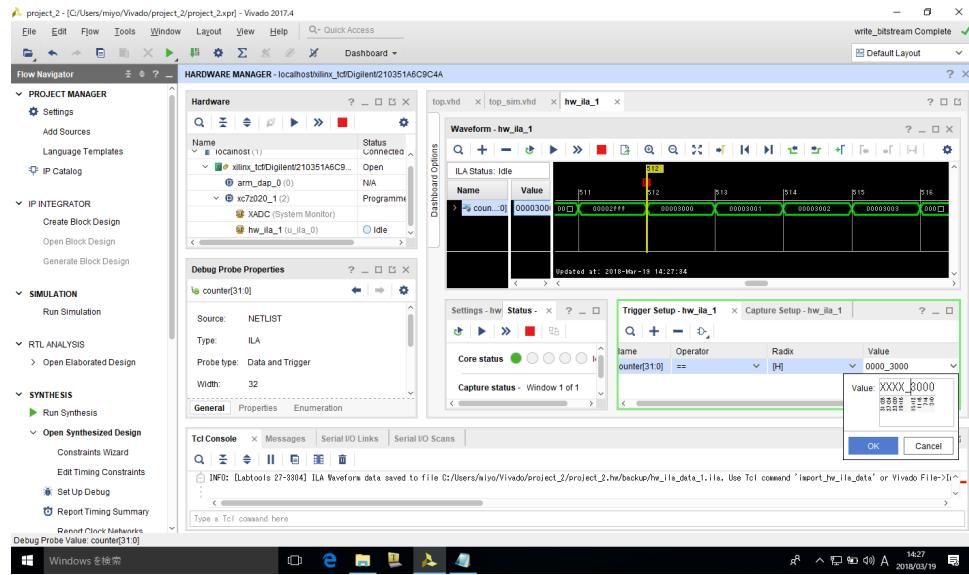


図 28 トリガにはドントケア(X)を指定することも可能。ここでは下位 16bit が 3000 になるデータを取得するように指定してみる

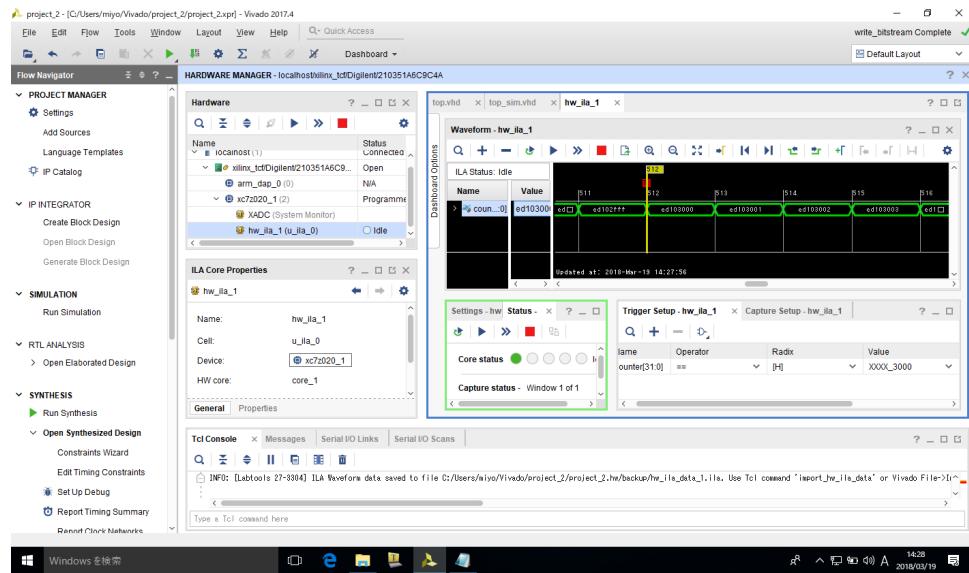


図 29 上位 16bit は指定なく、下位 16bit が 3000 の時点のデータがキャプチャできていることがわかる

2.4 ILA 插入すると回路は変わる

重要な点ですが、ILA を挿入すると、挿入前とは異なるハードウェアになることを理解しておく必要があります。ILA 向けのリソース使用量が増えるのはもちろん、観測対象の信号の接続関係も変化します。また観測のために残すべきレジスタの都合で最適化の結果もかわってきます。

たとえば、図 30 は、counter に mark_debug アトリビュートを付与せずに合成した場合のデバッグビューです。3bit 目を led に接続し、それ以上の bit 数の値は利用されていないため、ぱっさりと回路が小さくなっています。

しかし、mark_debug アトリビュートを付与して合成した場合には、もちろん最適化するわけにはいかないため、図 6 のように要/不要にかかわらず 32bit 分すべてのレジスタが回路として生成されています。

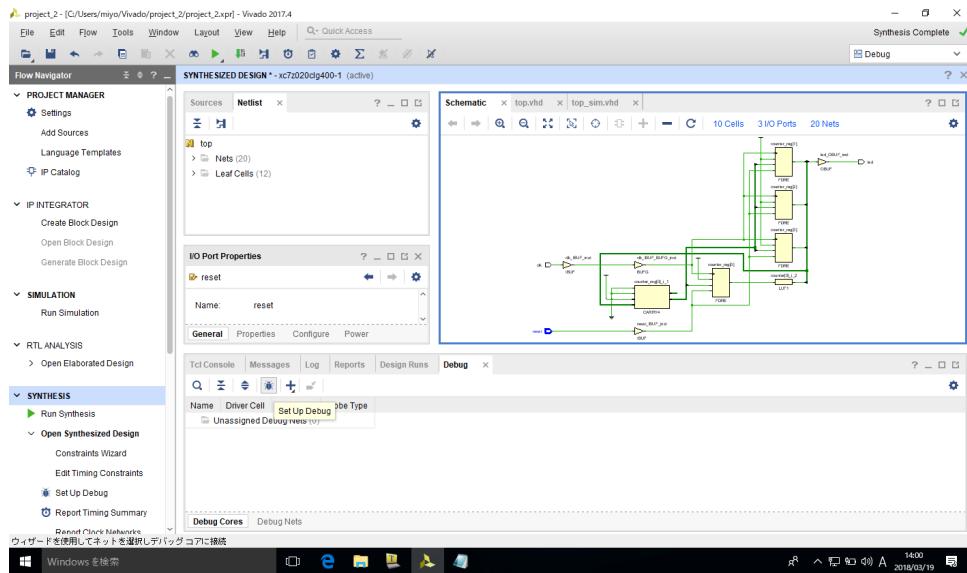


図 30 counter に mark_debug がない場合

3 基本実験の準備

FPGA を使った実験をする前に、動作の様子を確認しながら実験できるように簡単なテンプレートモジュールを用意しておくことにします。ここで作るのは、図 31 のように 4bit の入力と 4bit の出力ポートで構成されるモジュールです。ZYBO の DIP スイッチ SW0～SW3 を 4bit の入力に、LED LD0～LD3 を 4bit の出力にマッピングすることにします。

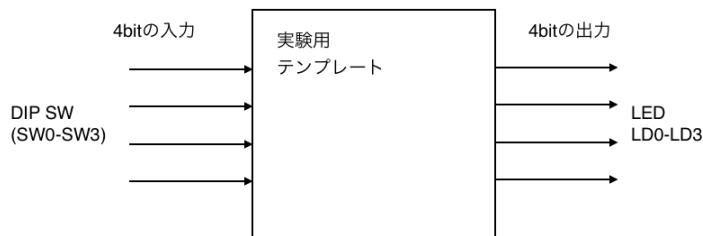


図 31 実験用の簡単なテンプレートモジュール

次のような内容の VHDL ファイルを用意します。

```
1 library ieee;
2
3 use ieee.std_logic_1164.all;
4 use ieee.numeric_std.all;
5
6 entity top is
7 port (
8     CLK : in std_logic;
9     SW : in std_logic_vector(3 downto 0);
10    LD : out std_logic_vector(3 downto 0)
11 );
12 end entity top;
13
14 architecture RTL of top is
15
16    signal sw_d0 : std_logic_vector(3 downto 0);
17    signal sw_d1 : std_logic_vector(3 downto 0);
18
19 begin
20
21    LD <= sw_d1;
22
23    process(CLK)
24    begin
25        if rising_edge(CLK) then
26            sw_d0 <= SW;
27            sw_d1 <= sw_d0;
28        end if;
29    end process;
30
31 end RTL;
```

ピン定義も用意しましょう。第3章で紹介したようにGUIで設定することもできますが、スクリプトファイルでピン定義を決めることもできます。実験に使用するZYBO Z7-20の全てのI/O定義は<https://github.com/Digilent/digilent-xdc/blob/master/Zybo-Z7-Master.xdc>にまとまっています。URL先の情報に基づいて、使用するピンの定義をまとめると次のようになります。top.xdcなどと、拡張子を.xdcとしてファイルに保存します。

```
1 set_property -dict {PACKAGE_PIN K17 IOSTANDARD LVCMOS33} [get_ports {CLK}];  
2 create_clock -add -name clk_pin -period 8.00 -waveform {0 4} [get_ports {CLK}];  
3  
4 set_property -dict {PACKAGE_PIN G15 IOSTANDARD LVCMOS33} [get_ports {SW[0]}];  
5 set_property -dict {PACKAGE_PIN P15 IOSTANDARD LVCMOS33} [get_ports {SW[1]}];  
6 set_property -dict {PACKAGE_PIN W13 IOSTANDARD LVCMOS33} [get_ports {SW[2]}];  
7 set_property -dict {PACKAGE_PIN T16 IOSTANDARD LVCMOS33} [get_ports {SW[3]}];  
8  
9 set_property -dict {PACKAGE_PIN M14 IOSTANDARD LVCMOS33} [get_ports {LD[0]}];  
10 set_property -dict {PACKAGE_PIN M15 IOSTANDARD LVCMOS33} [get_ports {LD[1]}];  
11 set_property -dict {PACKAGE_PIN G14 IOSTANDARD LVCMOS33} [get_ports {LD[2]}];  
12 set_property -dict {PACKAGE_PIN D18 IOSTANDARD LVCMOS33} [get_ports {LD[3]}];
```

作成した VHDL ファイルと定義ファイルをプロジェクトに追加して合成し、できあがった bit ファイルを ZYBO Z7-20 に書きこみましょう。DIP スイッチをオン・オフすることで LED が点灯、消灯するはずです。

4 基本演算の動作を確認してみよう

基本的な論理演算である、AND/OR/XOR の動作を実機の ILA を使って確認してみましょう。
AND/OR/XOR/NOT の動作を確認するためのモジュールとして次のようなモジュールを用意します。名前は `logic_test.vhd` として保存することにします。

```
1 library ieee;
2
3 use ieee.std_logic_1164.all;
4 use ieee.numeric_std.all;
5
6 entity logic_test is
7 port (
8     CLK : in std_logic;
9     a, b : in std_logic;
10    q_and : out std_logic;
11    q_or : out std_logic;
12    q_xor : out std_logic;
13    q_not : out std_logic
14 );
15 end entity logic_test;
16
17 architecture RTL of logic_test is
18
19     attribute mark_debug : string;
20
21     q_and_i : std_logic;
22     q_or_i : std_logic;
23     q_xor_i : std_logic
24
25     attribute mark_debug of q_and_i : signal is "true";
26     attribute mark_debug of q_or_i : signal is "true";
27     attribute mark_debug of q_xor_i : signal is "true";
28     attribute mark_debug of q_not_i : signal is "true";
29
30 begin
31
32     q_and <= q_and_i;
33     q_or  <= q_or_i;
34     q_xor <= q_xor_i;
35     q_not <= q_not_i;
36
37     process(CLK)
38 begin
39         if rising_edge(CLK) then
40             q_and_i <= a and b;
41             q_or_i  <= a or b;
42             q_xor_i <= a and b;
43             q_not_i <= xor a;
44         end if;
45     end process;
46
47 end RTL;
```

先に用意したテンプレートに組み込んで、実機で動作を確認するために、`top.vhd` を次のように変更します。

```
1 library ieee;
2
3 use ieee.std_logic_1164.all;
4 use ieee.numeric_std.all;
5
6 entity top is
7 port (
8     CLK : in std_logic;
9     SW : in std_logic_vector(3 downto 0);
10    LD : out std_logic_vector(3 downto 0)
11 );
12 end entity top;
13
14 architecture RTL of top is
15
16 signal sw_d0 : std_logic_vector(3 downto 0);
17 signal sw_d1 : std_logic_vector(3 downto 0);
18
19 component logic_test
20 port (
21     CLK : in std_logic;
22     a, b : in std_logic;
23     q_and : out std_logic;
24     q_or : out std_logic;
25     q_xor : out std_logic;
26     q_not : out std_logic
27 );
28 end component logic_test;
29
30 begin
31
32 -- LD <= sw_d1;
33
34 process(CLK)
35 begin
36     if rising_edge(CLK) then
37         sw_d0 <= SW;
38         sw_d1 <= sw_d0;
39     end if;
40 end process;
41
42 U : logic_test port map(
43     CLK      => CLK,
44     a        => sw_d1(0),
45     b        => sw_d1(1),
46     q_and   => LD(0),
47     q_or    => LD(1),
48     q_xor   => LD(2),
49     q_not   => LD(3)
50 );
51
52 end RTL;
```

5 ランダムな振る舞いを実現する擬似乱数の生成

ゲームなどで、ランダムな振る舞いをさせたいときに用いられるのが乱数です。本物の乱数を作るのは非常に難しいため、一般的には数式で導いた擬似乱数で代用します。ソフトウェアで乱数を作成する場合は、`rand`関数などを呼び出すことで乱数系列に従って生成された値を利用できます。

乱数系列の作り方には、いろいろな方法があります。今回は、ビット操作の練習として、シフトと XOR 演算のみで構成できる XORSHIFT 法(参考文献 1)に基づく乱数生成器を実装してみましょう。32 ビットの XORSHIFT 法による乱数生成を C で記述すると、

```
1 unsigned long xor() {
2     static unsigned long y=2463534242;
3     y ^= (y << 13);
4     y ^= (y >> 17);
5     return (y ^= (y<<5));
6 }
```

という関数になります。この関数と同等の操作をするハードウェア・モジュールを作成し、シミュレーションと実機で動作を確認してみましょう。

```
1 library ieee;
2
3 use ieee.std_logic_1164.all;
4 use ieee.numeric_std.all;
5
6 entity xorshift is
7     port (
8         CLK      : in  std_logic;
9         Q        : out std_logic_vector(31 downto 0)
10    );
11 end entity xorshift;
12
13 architecture RTL of xorshift is
14
15     attribute mark_debug : string;
16
17     -- 2463534242 = 0x92d68ca2
18     signal y : std_logic_vector(63 downto 0) := X"0000000092d68ca2";
19     signal y0_d, y1_d : std_logic_vector(63 downto 0);
20
21     attribute mark_debug of y : signal is "true";
22     attribute mark_debug of y0_d : signal is "true";
23     attribute mark_debug of y1_d : signal is "true";
24
25 begin
26
27     Q <= y(31 downto 0);
28
29     process(CLK)
30         variable y0 : std_logic_vector(63 downto 0);
31         variable y1 : std_logic_vector(63 downto 0);
32     begin
33         if rising_edge(CLK) then
34             -- y ^= (y << 13);
35             y0 := y xor (y(63-13 downto 0) & "0000000000000000");
36             -- y ^= (y >> 17);
37             y1 := y0 xor ("0000000000000000" & y0(63 downto 17));
38             -- y ^= (y << 5);
39             y <= y1 xor (y1(63-5 downto 0) & "00000");
40
41             -- to debug
42             y0_d <= y0;
43             y1_d <= y1;
44         end if;
45     end process;
46
47 end RTL;
```

```
1 library ieee;
2
3 use ieee.std_logic_1164.all;
4 use ieee.numeric_std.all;
5
6 entity top is
7 port (
8     CLK : in std_logic;
9     SW : in std_logic_vector(3 downto 0);
10    LD : out std_logic_vector(3 downto 0)
11 );
12 end entity top;
13
14 architecture RTL of top is
15
16 signal sw_d0 : std_logic_vector(3 downto 0);
17 signal sw_d1 : std_logic_vector(3 downto 0);
18
19 component xorshift
20 port (
21     CLK : in std_logic;
22     Q : out std_logic_vector(31 downto 0)
23 );
24 end component xorshift;
25
26 begin
27
28 -- LD <= sw_d1;
29
30 process(CLK)
31 begin
32 if rising_edge(CLK) then
33     sw_d0 <= SW;
34     sw_d1 <= sw_d0;
35 end if;
36 end process;
37
38 U : xorshift
39 port map(
40     CLK      => CLK,
41     Q(3 downto 0)  => LD(3 downto 0),
42     Q(31 downto 4) => open
43 );
44
45 end RTL;
```

6 ビット加算器を作ってみよう

ビット加算器、つまり足算の基本要素を作ってみましょう。ビット加算器では、足される数、足す数、および、繰り上がりの 3bit の入力から、その桁の結果と繰り上がりの 2bit を出力します。全加算器は半加算器 2 個と OR で作ることができます。以下のリストを参考に、加算器が正しく動作することをシミュレータおよび ILA を使って実機で確認してください。

```
1 -- 半加算器
2 library ieee;
3 use ieee.std_logic_1164.all;
4 use ieee.numeric_std.all;
5
6 entity half_addr is
7     port ( a : in std_logic;
8             b : in std_logic;
9             s : out std_logic;
10            c : out std_logic
11        );
12 end half_addr;
13
14 architecture RTL of half_addr is
15
16     attribute mark_debug : string;
17
18     signal s_i : std_logic;
19     signal c_i : std_logic;
20
21     attribute mark_debug of s_i : signal is "true";
22     attribute mark_debug of c_i  : signal is "true";
23
24 begin
25
26     s <= s_i;
27     c <= c_i;
28
29     process(a, b)
30     begin
31         s_i <= a xor b;
32         c_i <= a and b;
33     end process;
34
35 end RTL;
```

```
1 -- 全加算器
2 library ieee;
3 use ieee.std_logic_1164.all;
4 use ieee.numeric_std.all;
5
6 entity full_addr is
7     Port ( a : in std_logic;
8             b : in std_logic;
9             ci : in std_logic;
10            s : out std_logic;
11            co : out std_logic
12         );
13 end full_addr;
14
15 architecture RTL of full_addr is
16
17     attribute mark_debug : string;
18
19     signal s_i : std_logic;
20     signal co_i : std_logic;
21
22     attribute mark_debug of s_i : signal is "true";
23     attribute mark_debug of co_i : signal is "true";
24
25 component half_addr
26     Port ( a : in std_logic;
27             b : in std_logic;
28             s : out std_logic;
29             c : out std_logic
30         );
31 end component half_addr;
32
33 signal s0 : std_logic;
34 signal c0 : std_logic;
35 signal c1 : std_logic;
36
37 begin
38
39     s <= s_i;
40     co <= co_i;
41
42     U0: half_addr port map( a => a, b => b, s => s0, c => c0);
43     U1: half_addr port map( a => s0, b => ci, s => s_i, c => c1);
44     co_i <= c0 or c1;
45
46 end RTL;
```

7 HDL の四則演算を試してみよう

VHDL や Verilog HDL では、加算器のレベルでハードウェアを設計する必要はなく、実際には定義されている算術演算を利用することができます。 $+/-/*$ の動作をシミュレータおよび実機で確認してみてください。たとえば、次のような VHDL コードを書いて試すことができます。

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 entity arith_test is
6     port (
7         a          : in  std_logic_vector(1 downto 0);
8         b          : in  std_logic_vector(1 downto 0);
9         q_a_add_b : out std_logic_vector(2 downto 0);
10        q_a_sub_b : out std_logic_vector(2 downto 0);
11        q_a_mult_b: out std_logic_vector(3 downto 0)
12    );
13 end arith_test;
14
15 architecture RTL of arith_test is
16
17     attribute mark_debug : string;
18
19     signal q_a_add_b_i : unsigned(2 downto 0);
20     signal q_a_sub_b_i : unsigned(2 downto 0);
21     signal q_a_mult_b_i: unsigned(3 downto 0);
22
23     attribute mark_debug of q_a_add_b_i : signal is "true";
24     attribute mark_debug of q_a_sub_b_i : signal is "true";
25     attribute mark_debug of q_a_mult_b_i : signal is "true";
26
27 begin
28
29     q_a_add_b <= std_logic_vector(q_a_add_b_i);
30     q_a_sub_b <= std_logic_vector(q_a_sub_b_i);
31     q_a_mult_b <= std_logic_vector(q_a_mult_b_i);
32
33     process(a, b)
34     begin
35         q_a_add_b_i <= unsigned('0' & a) + unsigned('0' & b);
36         q_a_sub_b_i <= unsigned('0' & a) - unsigned('0' & b);
37         q_a_mult_b_i <= unsigned(a) * unsigned(b);
38     end process;
39
40 end RTL;
```

8 合計値の計算

算術演算の応用問題として、与えられたデータ（たとえば32ビットのビット列）の中に、1がいくつあるかを数えて値を返すというモジュールを作成してみましょう。たとえば、0x00000001の場合はビット列の中に1は1個、0xFFFFFFFFの場合は16個という値を出力する、モジュールです。

もちろん逐次的に、1クロックで1bitずつ'0'か'1'かを検査するという、ソフトウェア的な実装も考えられます、ここでは、入力された値に対して即座に結果を返す組み合わせ回路として設計し、シミュレーションと実機で動作を確認してみてください。

たとえば、次のようなVHDLコードを書いて試すことができます。

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 entity bitcount is
6     port (
7         a : in std_logic_vector(31 downto 0);
8         q : out std_logic_vector(4 downto 0)
9     );
10 end bitcount;
11
12 architecture RTL of bitcount is
13
14     attribute mark_debug : string;
15
16     signal q_i : unsigned(4 downto 0);
17     attribute mark_debug of q_i : signal is "true";
18
19 begin
20
21     q <= std_logic_vector(q_i);
22
23     process(a)
24         variable sum : integer := 0;
25     begin
26         sum := 0;
27         for i in 0 to a'length-1 loop
28             if a(i) = '1' then
29                 sum := sum + 1;
30             end if;
31         end loop;
32         q_i <= to_unsigned(sum, q_i'length);
33     end process;
34
35 end RTL;
```

9 PWM

ある値を、1bitの信号の'1'と'0'の幅で表現するPWMという変調方式があります。デジタルで簡単に信号の強度を変える方法としてもよく利用されます。

次のVHDLコードは、PWMを実装してみた例です。

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 entity pwm is
6     port (
7         clk : in std_logic;
8         a   : in std_logic_vector(3 downto 0);
9         d   : in std_logic;
10        q   : out std_logic
11    );
12 end pwm;
13
14 architecture RTL of pwm is
15
16     attribute mark_debug : string;
17
18     signal counter : unsigned(3 downto 0) := (others => '0');
19     signal q_i      : std_logic := '0';
20
21     attribute mark_debug of q_i      : signal is "true";
22     attribute mark_debug of counter : signal is "true";
23
24 begin
25
26     q <= q_i;
27
28     process(clk)
29     begin
30         if rising_edge(clk) then
31             counter <= counter + 1;
32             if counter >= unsigned(a) and unsigned(a) < 15 then
33                 q_i <= d;
34             else
35                 q_i <= not d;
36             end if;
37         end if;
38     end process;
39
40 end RTL;
```

次のようにスイッチを PWM の幅に、出力を LED に割り当てて合成したときの動作を実機で確認してみましょう。

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 entity top is
6     port (
7         CLK : in std_logic;
8         SW  : in std_logic_vector(3 downto 0);
9         LD  : out std_logic_vector(3 downto 0)
10    );
11 end entity top;
12
13 architecture RTL of top is
14     signal sw_d0 : std_logic_vector(3 downto 0);
15     signal sw_d1 : std_logic_vector(3 downto 0);
16     component pwm
17         port (
18             clk : in std_logic;
19             a  : in std_logic_vector(3 downto 0);
20             d  : in std_logic;
21             q  : out std_logic
22        );
23     end component pwm;
24     signal pwm_q : std_logic;
25 begin
26
27     process(CLK)
28     begin
29         if rising_edge(CLK) then
30             sw_d0 <= SW;
31             sw_d1 <= sw_d0;
32         end if;
33     end process;
34
35     U: pwm
36         port map(
37             clk => clk,
38             a   => sw_d1,
39             d   => '1',
40             q   => pwm_q
41        );
42     LD(0) <= pwm_q;
43     LD(1) <= pwm_q;
44     LD(2) <= pwm_q;
45     LD(3) <= pwm_q;
46 end RTL;
```

10 ステートマシンの作り方と利用方法

基本実験の最後に、ハードウェア・プログラミングで逐次的に処理するために必要不可欠な概念であるステート・マシンと、その実装方法を学びましょう。ハードウェアでは処理を並列に実行できますが、世の中には順番にしか実行できない物事もたくさんあります。たとえば、そうめんを茹でるとき、鍋の水がまだ湯になる前に並行してそうめんを鍋に入れても、とても食べられるものはできあがりません。何事でも順序を守ることが大事なときもありますよね。

10.1 料理は逐次的な処理

何かを実行し、その次に何かを実行し、その次に…、という決まった手順に従った処理の実装は、ソフトウェアによく見られます。たとえば、そうめんをゆでるときには、

- 鍋に水を入れる
- 鍋を火にかける
- 沸騰するまで待つ
- そうめんを入れる
- 1分くらい待つ
- 十分やわらかいか確認する
- やわらかくなったら取り出して、できあがり

という手順が必要となります。ソフトウェアであれば、このような手順をそのままプログラミング言語で記述して実装できます。しかし、ハードウェア・プログラミングでは逐次的な処理をそのまま記述できません。逐次的に処理を進めることそのものを自分で記述する必要があります。これを簡単に扱う道具がステート・マシン(状態遷移機械)です。

10.2 ステート・マシンとは

図32は、そうめんをゆでる手順をステート・マシン的に表現した例です。楕円は「状態」を矢印は「状態遷移」を示しています。矢印に条件が書かれている場合は、その条件が満たされたときだけ状態が遷移する、ということを意味します。

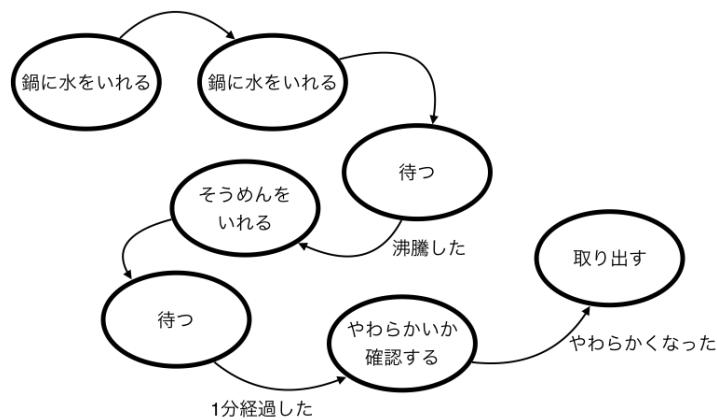


図 32 そうめんを茹でる流れをステートマシン的に表現してみた例.

ステート・マシンとは、処理を「状態」と「状態遷移」で抽象化した概念です。列挙された状態を定義された状態遷移に従って順々にたどっていくことで所望の処理が実現できます。各状態での処理を定義することで、逐次的な処理をステート・マシンを使って記述できます。

10.3 ハードウェア・プログラミングと相性の良いステート・マシン

ステート・マシンをハードウェア・プログラミングで実装するのは意外と簡単です。そうめんを茹でる手順をハードウェア記述言語の一つである VHDL で記述した擬似コードを示します。各状態を `std_logic_vector` 型の変数 `state` で管理しています。`state` の値が各状態に対応しています。変数 `state` をクロックごとに参照し、その時点で実行すべき処理を判断します。各状態では、次の状態に遷移するための条件判断と、遷移のための状態変数の更新を行います。

```
1  -----
2  -- そうめんをゆでるステートマシンの擬似コード
3  -----
4  architecture RTL of somen
5
6      signal state : std_logic_vector(2 downto 0) := (others => '0');
7
8      process(clk)
9      begin
10         if rising_edge(clk) then
11             case conv_integer(state) -- 変数「state」によって状態の場合分けをする
12                 when 0 =>
13                     鍋に水をいれる
14                     state <= conv_std_logic_vector(1, 3);
15                 when 1 =>
16                     鍋を火をかける
17                     state <= conv_std_logic_vector(2, 3);
18                 when 2 =>
19                     if 水が湧いたか? = true then
20                         state <= conv_std_logic_vector(3, 3);
21                     end if;
22                 when 3 =>
23                     そうめんをいれる
24                     state <= conv_std_logic_vector(4, 3);
25                 when 4 =>
26                     if 時間 = 1分 then
27                         state <= conv_std_logic_vector(5, 3);
28                     end if;
29                 when 5 =>
30                     if やわらかさが十分か? = true then
31                         state <= conv_std_logic_vector(6, 3);
32                     end if;
33                 when 6 =>
34                     -- おしまい
35                     when others => -- 「上記以外のその他」に相当。これで、全条件を列挙できた。
36                         null;
37                     end case;
38                 end if;
39             end process;
40
41 end RTL;
```

ところで、リスト 1 のコードには、0, 1, 2, ... という状態を識別するための番号が振られています。これらの値に意味はなく、単にほかと区別するために便宜的に付けられたマジック・ナンバです。マジック・ナンバはコードの可読性を下げ、後の変更を加えづらくします。ソフトウェアでもマジック・ナンバは忌み嫌われるよう、ハードウェア・プログラミングでもできれば避けたいものです。

VHDL では自分で型を定義することで、Verilog では define や localparam を使って値に名前を付けること

で、ソース・コードからマジック・ナンバを取り除くことができます。VHDL で状態を表す型を定義して、マジック・ナンバをなくした例が次の通りです。

```
1 -----  
2 -- そうめんをゆでるステートマシンの擬似コード  
3 -- 状態変数のための型を定義しマジックナンバをなくしたバージョン  
4 -----  
5 architecture RTL of somen  
6  
7     -- 状態を表わす型を定義する。これは、enum のような列挙型に相当。  
8     type StateType is (WATER, FIRE, HOT_WATER, PUT_SOMEN, WAIT_A_MIN, BOIL, FIN)  
9     signal state : StateType := WATER;  
10  
11    process(clk)  
12    begin  
13        if rising_edge(clk) then  
14            case conv_integer(state)  
15                when WATER =>  
16                    鍋に水をいれる  
17                    state <= FIRE;  
18                when FIRE =>  
19                    鍋を火をかける  
20                    state <= HOT_WATER;  
21                when HOT_WATER =>  
22                    if "水が湧いたか？ = true" then  
23                        state <= PUT_SOMEN;  
24                    end if;  
25                when PUT_SOMEN =>  
26                    そうめんをいれる  
27                    state <= WAIT_A_MIN;  
28                when WAIT_A_MIN =>  
29                    if 時間 = 1 分 then  
30                        state <= BOIL  
31                    end if;  
32                when BOIL =>  
33                    if やわらかさが十分か？ = true then  
34                        state <= conv_std_logic_vector(6, 3);  
35                    end if;  
36                when FIN =>  
37                    -- おしまい  
38                when others =>  
39                    null;  
40            end case;  
41        end if;  
42    end process;  
43  
44 end RTL;
```

次のリストを参考に、ステートマシンを利用したハードウェアを設計し、動作をシミュレーションと ILA で確認してみてください。

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 entity stmt_test is
6     port ( clk : in std_logic;
7             a,b : in std_logic;
8             led : out std_logic_vector(2 downto 0)
9         );
10 end stmt_test;
11
12 architecture RTL of stmt_test is
13     attribute mark_debug : string;
14
15     signal led_i : std_logic_vector(2 downto 0) := (others => '0');
16     attribute mark_debug of led_i : signal is "true";
17
18     type StateType is (BLACK, RED, GREEN, BLUE);
19     signal state : StateType := BLACK;
20
21     signal a_d, b_d : std_logic := '0';
22     signal a_rising, b_rising : std_logic := '0';
23
24 begin
25
26     led <= led_i;
27     a_rising <= '1' when a_d = '0' and a = '1' else '0';
28     b_rising <= '1' when b_d = '0' and b = '1' else '0';
29
30     process(clk)
31     begin
32         if rising_edge(clk) then
33             a_d <= a;
34             b_d <= b;
35
36             case state is
37             when BLACK =>
38                 led_i <= "000";
39                 if a_rising = '1' then
40                     state <= RED;
41                 elsif b_rising = '1' then
42                     state <= BLUE;
43                 end if;
44             when RED =>
45                 led_i <= "001";
46                 if a_rising = '1' then
47                     state <= GREEN;
48                 elsif b_rising = '1' then
49                     state <= BLACK;
50                 end if;
51             when GREEN =>
52                 led_i <= "010";
53                 if a_rising = '1' then
54                     state <= BLUE;
55                 elsif b_rising = '1' then
56                     state <= RED;
57                 end if;
58             when BLUE =>
59                 led_i <= "100";
60                 if a_rising = '1' then
61                     state <= BLACK;
62                 elsif b_rising = '1' then
63                     state <= GREEN;
64                 end if;
65             when others =>
66                 led_i <= "000";
67                 state <= BLACK;
68             end case;
69         end if;
70     end process;
71 end RTL;
```

参考文献

1. George Marsaglia, "Xorshift RNGs", The Florida State University, <http://www.jstatsoft.org/v08/i14>

アプリケーション実験

HDL のいろいろな記述方法を使って、少し複雑なモジュールの設計に取り組んでみましょう。

1 はじめに

これまでに学んできた内容を利用して、少し応用的な実験に取り組んでみましょう。

2 ストップウォッチを作ってみよう

ここでは、

- アクションボタンを押すとミリ秒毎のカウントアップを開始する
- カウント中に、アクションボタンを押すとカウントを一時停止する
- 一時停止中に、アクションボタンを押すとカウントを再開する
- リセットボタンを押すとカウントを 0 にクリアする

という機能をもったストップウォッチを作ってみましょう。

残念ながら、ZYBO Z7-20 には、7 セグメント LED のような数字を表示する分かりやすい表示器はありませんので、4 つの LED と 2 つの 3 色 LED を使って数を表現するとよいでしょう。

次のリストはストップウォッチを実装するためのトップモジュールの例です。3 色 LED の明るさを適当に調節するために、PWM モジュールを使って出力を変調しています。PWM モジュールによって、たとえば '1' を出力する場合でも、適当なタイミングで '1' と '0' がまぜられることで明るさを抑えることができます。

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 entity stopwatch_z7_20 is
6 port (
7     CLK : in std_logic;
8
9     led6_r : out std_logic;
10    led6_g : out std_logic;
11    led6_b : out std_logic;
12
13    led5_r : out std_logic;
14    led5_g : out std_logic;
15    led5_b : out std_logic;
16
17    LD : out std_logic_vector(3 downto 0);
18
19    btn : in std_logic_vector(1 downto 0)
20 );
21
22 end entity stopwatch_z7_20;
23
24 architecture RTL of stopwatch_z7_20 is
25
26 attribute ASYNC_REG : string;
27 attribute mark_debug : string;
28
29 component stopwatch
30 generic (
31     FREQ_MHz : integer := 125
32 );
33 port (
34     clk      : in std_logic;
35     reset    : in std_logic;
36     action   : in std_logic;
37     msec_out : out std_logic_vector(9 downto 0);
38     sec_out  : out std_logic_vector(5 downto 0);
39     min_out  : out std_logic_vector(5 downto 0);
40     hour_out : out std_logic_vector(4 downto 0)
41 );
42 end component stopwatch;
43
44 component pwm
45 port (
```

次のページに続く

```
1      clk : in  std_logic;
2      a    : in  std_logic_vector(3 downto 0);
3      d    : in  std_logic;
4      q    : out std_logic
5      );
6  end component pwm;
7
8  signal btn_d0 : std_logic_vector(1 downto 0);
9  signal btn_d1 : std_logic_vector(1 downto 0);
10
11 attribute ASYNC_REG of btn_d0, btn_d1 : signal is "TRUE";
12
13 signal msec_out : std_logic_vector(9 downto 0);
14 signal sec_out  : std_logic_vector(5 downto 0);
15 signal min_out  : std_logic_vector(5 downto 0);
16 signal hour_out : std_logic_vector(4 downto 0);
17
18 attribute mark_debug of msec_out : signal is "true";
19 attribute mark_debug of sec_out  : signal is "true";
20 attribute mark_debug of min_out  : signal is "true";
21 attribute mark_debug of hour_out : signal is "true";
22
23 begin
24
25 process(CLK)
26 begin
27   if rising_edge(CLK) then
28     btn_d0 <= btn;
29     btn_d1 <= btn_d0;
30   end if;
31 end process;
32
33 U: stopwatch
34 generic map(
35   FREQ_MHz => 125
36   )
37 port map(
38   clk      => CLK,
39   reset    => btn_d1(0),
40   action    => btn_d1(1),
41   msec_out => msec_out,
42   sec_out   => sec_out,
43   min_out   => min_out,
44   hour_out  => hour_out
45   );
```

次のページに続く

```

1 LD <= sec_out(3 downto 0);
2
3 PWM0 : pwm
4     port map(clk => clk, a => "1100", d => msec_out(0), q => led5_r);
5 PWM1 : pwm
6     port map(clk => clk, a => "1100", d => msec_out(1), q => led5_g);
7 PWM2 : pwm
8     port map(clk => clk, a => "1100", d => msec_out(2), q => led5_b);
9 PWM3 : pwm
10    port map(clk => clk, a => "1100", d => msec_out(3), q => led6_r);
11 PWM4 : pwm
12    port map(clk => clk, a => "1100", d => msec_out(4), q => led6_g);
13 PWM5 : pwm
14    port map(clk => clk, a => "1100", d => msec_out(5), q => led6_b);
15
16
17 end RTL;

```

stopwatch モジュールを実装して、完成させてみましょう。

3 シリアル通信に挑戦してみよう

最近のパソコンの外部機器の I/O のほとんどは USB です。ノート・パソコンは勿論、省スペースのデスクトップ・パソコンにさえ、RS-232-C のシリアル通信のポートが搭載されなくなっていました。それでも、送信と受信の 2 本で通信可能な RS-232-C は、パソコンと FPGA の間や FPGA と FPGA の間でデータを手軽にやりとりする手段として基本的かつ必要不可欠な存在です。

シリアル・ポートのないノート・パソコンでも、市販の USB-シリアル変換ケーブルや Bluetooth-シリアル変換モジュールを使って簡単に接続できます。

3.1 シリアル通信のしくみ

シリアル通信では、受信と送信で独立したポートを持ちます。機器を直結する場合には、お互いの送受信ポートをクロスして結線します(図 1)。

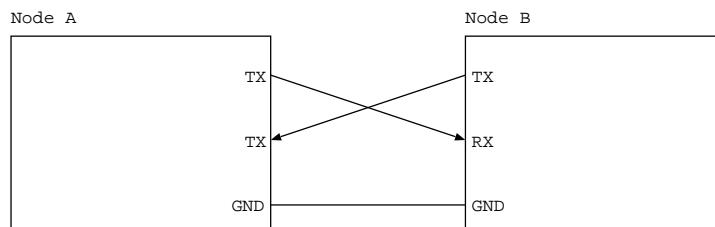


図 1 シリアル通信の結線

RS-232-C では、図 2 のように、8 ビットのデータにスタート・ビット ('0') とストップ・ビット ('1') を

付加してデータを通信します。接続した機器同士であらかじめ決めた速度(たとえば 19200bps など)で信号を送受信することで、共通したクロックを持つことなくデータの送受信ができます。

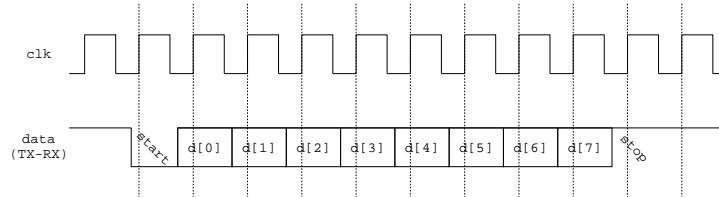


図 2 シリアル通信の信号の伝送方式

3.2 送信モジュール

決められた速度で信号をパタパタと変化させるだけです。ただし、一般に、シリアル通信の速度は回路の動作クロックに対して、とても遅いので、データを送信している途中に送るべき信号を更新してしまわないようブロックする仕組みが必要です。VHDL による実装例を示します。

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 entity uart_tx is
6     -- 定数宣言
7     generic (
8         sys_clk : integer := 14000000;           --クロック周波数
9         rate    : integer := 9600                --転送レート，単位は bps(ビット毎秒)
10    );
11
12    -- 入出力ポート宣言
13    port (
14        clk      : in  std_logic;              -- クロック
15        reset   : in  std_logic;              -- リセット
16        wr      : in  std_logic;              -- 送信要求
17        din     : in  std_logic_vector(7 downto 0); --送信データ
18        dout    : out std_logic;             --シリアル出力
19        ready   : out std_logic;             --送信要求を受け付けられるか
20    );
21
22 end uart_tx;
23
24
25 architecture rtl of uart_tx is
26     -- クロック分周モジュールの呼び出しの準備
27     component clk_div is
28         port (clk      : in  std_logic;
29                rst      : in  std_logic;
30                div      : in  std_logic_vector(15 downto 0);
31                clk_out : out std_logic);
32     end component;
33
34     -- 内部変数定義
35     signal in_din  : std_logic_vector(7 downto 0);  -- 送信データ一時保存用レジスタ
36     signal buf     : std_logic_vector(7 downto 0);  -- 一時的にしようするバッファ
37     signal load    : std_logic := '0';               -- 送信データを読み込んだかどうか
38     signal cbit    : unsigned(2 downto 0) := (others => '0'); -- 送信するビット番号
39
40     signal run     : std_logic := '0';               -- 送信状態にあるかどうか
41
42     signal tx_en   : std_logic; -- 送信用クロック
43     signal tx_en_d : std_logic := '0'; -- 送信用クロックの立ち上がり検出用
44
45     signal tx_div  : std_logic_vector(15 downto 0); -- クロック分周の倍率
46
47     signal status  : unsigned(1 downto 0); -- 状態遷移用レジスタ
48
49 begin
```

次のページに続く

```

1   -- クロック分周モジュールの呼び出し
2   -- clk_div の入出力ポートにこのモジュールの内部変数を接続
3   tx_div <= std_logic_vector(to_unsigned((sys_clk / rate) - 1, 16));
4   U0 : clk_div port map (clk=>clk, rst=>reset, div=>tx_div, clk_out=>tx_en);

5
6   -- ready への代入、常時値を更新している
7   ready  <= '1' when (wr = '0' and run = '0' and load = '0') else '0';

8
9   process(clk) --変化を監視する信号を記述する、この場合クロック
10 begin
11     if rising_edge(clk) then --クロックの立ち上がり時の動作
12       if reset = '1' then           --リセット時の動作、初期値の設定
13         load <= '0';
14       else
15         if(wr = '1' and run = '0') then    --送信要求があり、かつ送信中でない場合
16           load  <= '1';                  --データを取り込んだフラグを立てる
17           in_din <= din;                --一時保存用レジスタに値を格納
18         end if;
19         if(load = '1' and run = '1') then  --送信中で、かつデータを取り込んだ
20                                         --ことを示すフラグが立っている場合
21           load <= '0';                  --データを取り込んだフラグを下げる
22         end if;
23       end if;
24     end if;
25   end process;

26
27   process(clk)                         --変化を監視する信号(クロック)
28 begin
29     if rising_edge(clk) then
30       if reset = '1' then           --リセット時の動作、初期値の設定
31         dout    <= '1';
32         cbit    <= (others => '0');
33         status  <= (others => '0');
34         run     <= '0';
35         tx_en_d <= '0';
36       else
37         tx_en_d <= tx_en;
38         if tx_en = '1' and tx_en_d = '0' then -- tx_en の立ち上がりで動作
39           case to_integer(status) is          --status の値に応じて動作が異なる
40             when 0      =>                 --初期状態
41               cbit <= (others => '0');        --カウンタをクリア
42               if load = '1' then            -- データを取り込んでいる場合
43                 dout  <= '0';              -- スタートビット 0 を出力
44                 status <= status + 1;       -- 次の状態へ
45                 buf    <= in_din;          -- 送信データを一時バッファに退避

```

次のページに続く

```
1      run    <= '1';           -- 送信中の状態へ遷移
2      else
3          dout <= '1';
4          run  <= '0';         --送信要求受付可能状態へ
5      end if;
6      when 1 =>               --データを LSB から順番に送信
7          cbit <= cbit + 1;   -- カウンタをインクリメント
8          dout <= buf(to_integer(cbit)); --一時バッファの cbit 目を取出して出力
9          if(to_integer(cbit) = 7) then -- データの 8 ビット目を送信したら,
10             status <= status + 1; -- ストップビットを送る状態へ遷移
11         end if;
12     when 2 =>               -- ストップビットを送信
13         dout  <= '1';        --ストップビット 1
14         status <= (others => '0'); --初期状態へ
15     when others =>          --その他の状態の場合
16         status <= (others => '0'); -- 初期状態へ遷移
17     end case;
18 end if;
19 end if;
20 end if;
21 end process;

22
23
24 end rtl;
```

内部で利用している `clk_div` の実装は次の通りです。

```

1 library ieee;
2 use ieee.std_logic_1164.ALL;
3 use ieee.numeric_std.all;
4
5 entity clk_div is
6 port(
7     clk      : in std_logic;
8     rst      : in std_logic;
9     div      : in std_logic_vector(15 downto 0);
10    clk_out : out std_logic
11 );
12 end clk_div;
13
14 architecture RTL of clk_div is
15
16    signal counter : unsigned(15 downto 0) := (others => '0');
17
18 begin
19
20    process(clk)
21    begin
22        if(clk'event and clk = '1') then
23            if(rst = '1') then
24                counter <= (others => '0');
25            elsif (counter = unsigned(div)) then
26                counter <= (others => '0');
27                clk_out <= '1';
28            else
29                counter <= counter + 1;
30                clk_out <= '0';
31            end if;
32        end if;
33    end process;
34
35 end RTL;

```

`clk_div` モジュールで生成した、シリアル送信用の信号にあわせて、送信すべきデータを 1bit ずつ出力します。また、出力中は ready 信号を'0' にすることで、送信モジュールが動作中であることを上位のモジュールに伝えられるようになっています。

このモジュールの動作をシミュレーションで確認してみましょう。

3.3 受信モジュール

受信モジュールは送信モジュールに比べて少し複雑になります。送信側は自分のタイミングで信号を送ればいいのに対し、受信側ではスタート・ビットを頼りにデータの開始を検出し、以降の信号を正しい間隔でビットに分割しなければならないからです。また、データにはノイズが乗っている可能性もあるので、データを正

しく取得する仕組みも必要です。VHDL と Verilog HDL による実装の例が次のリストです。この例では、通信速度の 16 倍のクロックでデータをサンプリング（一定間隔で読み込む）することで、データを受信しています。

```

1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 entity uart_rx is
6   -- 定数宣言
7   generic(
8     sys_clk : integer := 14000000;           --クロック周波数
9     rate    : integer := 9600                --転送レート，単位は bps(ビット毎秒)
10    );
11  -- 入出力ポート宣言
12  port(
13    clk    : in  std_logic;                  -- クロック
14    reset : in  std_logic;                  -- リセット
15    din   : in  std_logic;                  -- シリアル入力
16    rd    : out std_logic;                 -- 受信完了を示す
17    dout  : out std_logic_vector(7 downto 0) -- 受信データ
18  );
19 end uart_rx;
20
21 architecture rtl of uart_rx is
22
23   --クロック分周モジュールのインスタンス生成の準備
24   component clk_div is
25     port(
26       clk      : in  std_logic;
27       rst      : in  std_logic;
28       div      : in  std_logic_vector(15 downto 0);
29       clk_out : out std_logic
30     );
31 end component;
32  --内部変数宣言
33  signal buf      : std_logic_vector(7 downto 0);  --受信データ系列の一時保存用
34  signal receiving : std_logic;                   --受信しているかどうか
35  signal cbit     : integer range 0 to 150; -- データの取り込みタイミング用カウンタ
36  signal rx_en    : std_logic;                   --受信用クロック
37  signal rx_en_d  : std_logic := '0';  --受信用クロック立ち上がり判定用レジスタ
38  signal rx_div   : std_logic_vector(15 downto 0); --クロック分周の倍率
39
40 begin
41   --クロック分周モジュールのインスタンス生成
42   --受信側は送信側の 16 倍の速度で値を取り込み処理を行う
43   rx_div <= std_logic_vector(to_unsigned(((sys_clk / rate) / 16) - 1, 16));
44   U0 : clk_div port map (clk=>clk, rst=>reset, div=>rx_div, clk_out=>rx_en);
45

```

次のページに続く

```

1   process(clk)                                --変化を監視する信号を記述, この場合クロック
2     begin
3       if rising_edge(clk) then
4           if reset = '1' then                  --リセット時の動作, 初期値の設定
5               receiving <= '0';
6               cbit      <= 0;
7               buf       <= (others => '0');
8               dout      <= (others => '0');
9               rd        <= '0';
10              rx_en_d  <= '0';
11          else
12              rx_en_d <= rx_en;
13              if rx_en = '1' and rx_en_d = '0' then    --受信用クロック立ち上がり時の動作
14                  if receiving = '0' then            --受信中でない場合
15                      if din = '0' then             --スタートビット0を受信したら
16                          rd <= '0';                --受信完了のフラグをさげる
17                          receiving <= '1';
18                      end if;
19                  else
20                      case cbit is
21                          when 6 =>
22                              if din = '1' then
23                                  receiving <= '0';
24                                  cbit      <= 0;
25                              else
26                                  cbit <= cbit + 1;
27                                  end if;
28                          when 22 | 38 | 54 | 70 | 86 | 102 | 118 | 134 =>  --data
29                              cbit <= cbit + 1;
30                              buf   <= din & buf(7 downto 1);  -- 新しい入力と受信済みデータを連結
31                      when 150 =>                    --stop
32                          rd        <= '1';
33                          dout      <= buf;
34                          receiving <= '0';        -- 受信完了
35                          cbit      <= 0;
36                      when others =>
37                          cbit <= cbit + 1;
38                      end case;
39                  end if;
40              end if;
41          end if;
42      end if;
43  end process;
44
45 end RTL;

```

通信速度の 16 倍でデータをサンプリングしているので、スタート・ビットは 16 サイクル分が取得できるはずです。スタートビットのはじまりと思われる'0'というデータを確認してから 6 サイクル目のデータをも

う一度取得し、そのデータがやっぱり'0'であれば、データ送信が開始されたと解釈します（ソース・コード中の状態 1）。データが送信されていることを確認した後は、16 サイクル毎にデータを取得します（ソース・コード中の状態 2）。8bit 分データを取得したら完了です。

送信モジュールと同じように、テストベンチを書いてシミュレーションで動作を確認してみましょう。

3.4 実機で動作確認

シリアル通信を実機で動作確認してみましょう。入出力に PMOD コネクタ JE の 1 ピンと 7 ピンを割当て、物理的に両者をジャンパピンで接続すれば、通信ができるはずです。動作を ILA を使って確認してみましょう。注意: PMOD コネクタの 6 ピンと 12 ピンは 3.3V、5 ピンと 11 ピンは GND です。間違って 6 ピンと 5 ピンなどのように 3.3V と GND を直結しないようにしましょう。

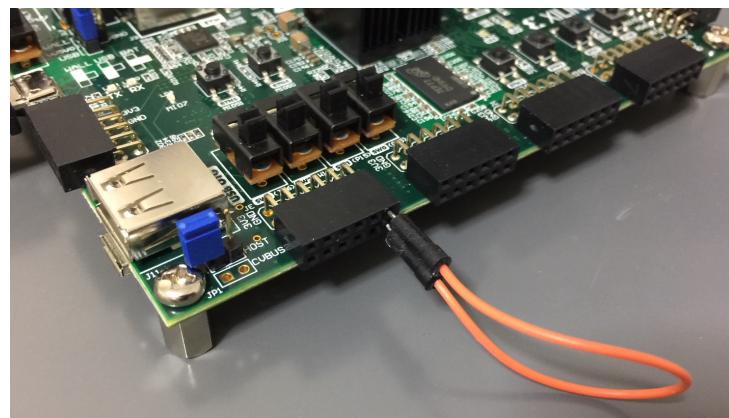


図 3 PMOD コネクタの JE の 1 ピンと 7 ピンをジャンパケーブルで接続した様子

4 音声信号を観測してみよう

ZYBO Z7-20 には音声入出力用のジャックが備わっています。これを利用すると外部からの音声信号を取り込み FPGA で処理することができます。信号は、SSM2603 という IC で A/D 変換されます。SSM2603 と FPGA は I2S で音声データをやりとりすることができます。

SSM2603 とやりとりする I2S 信号は図 4 の通りです。

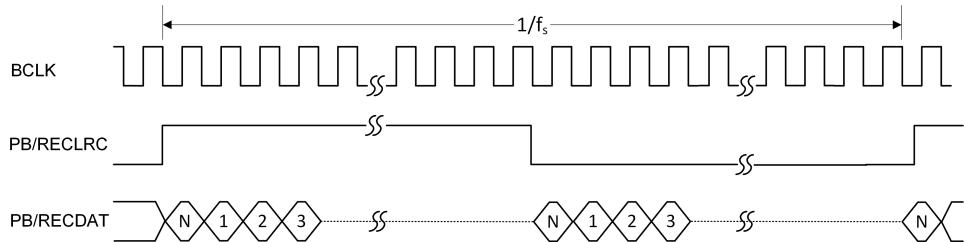


図 4 SSM2603 との通信インターフェース

これを送受信するモジュールを実装して音声信号の入出力をやってみましょう。でてくる信号が少し増えて複雑に見えるかもしれません、方針は RS-232-C の送受信と同じです。

4.1 I2S の送信

次のリストは、I2S の送信モジュールの実装例です。シミュレーションによって、信号が図 4 に示したように出力されることを確認してみましょう。

```
1 library ieee;
2
3 use ieee.std_logic_1164.all;
4 use ieee.numeric_std.all;
5
6 entity i2s_encoder is
7     generic (
8         WIDTH : integer := 24
9     );
10    port (
11        CLK : in std_logic; -- BCLK 4x
12
13        BCLK : in std_logic;
14        LRC : in std_logic;
15        DAT : out std_logic;
16
17        LIN : in std_logic_vector(WIDTH-1 downto 0);
18        RIN : in std_logic_vector(WIDTH-1 downto 0)
19    );
20 end entity i2s_encoder;
21
22 architecture RTL of i2s_encoder is
23
24     attribute mark_debug : string;
25
26     signal left_data : std_logic_vector(WIDTH-1 downto 0) := (others => '0');
27     signal left_cnt : unsigned(7 downto 0) := (others => '0');
28
29     signal right_data : std_logic_vector(WIDTH-1 downto 0) := (others => '0');
30     signal right_cnt : unsigned(7 downto 0) := (others => '0');
31
32     signal lrc_d : std_logic := '0';
33     signal bclk_d : std_logic := '0';
34
35     attribute mark_debug of left_data : signal is "true";
36     attribute mark_debug of left_cnt : signal is "true";
37
38     attribute mark_debug of right_data : signal is "true";
39     attribute mark_debug of right_cnt : signal is "true";
40
41 begin
42
43     process(CLK)
44     begin
45         if rising_edge(CLK) then
```

次のページに続く

```
1      lrc_d  <= LRC;
2      bclk_d <= BCLK;
3
4
5      if lrc_d = '1' and LRC = '0' then
6          left_cnt  <= (others => '0');
7          left_data <= LIN;
8      else
9          if left_cnt <= 4*WIDTH-1 then
10              left_cnt  <= left_cnt + 1;
11              if 3 <= left_cnt then
12                  DAT      <= left_data(WIDTH-1);
13                  if left_cnt(1 downto 0) = "10" then
14                      left_data <= left_data(WIDTH-2 downto 0) & '0';
15                  end if;
16              end if;
17          end if;
18      end if;
19
20      if lrc_d = '0' and LRC = '1' then
21          right_cnt <= (others => '0');
22          right_data <= RIN;
23      else
24          if right_cnt <= 4*WIDTH-1 then
25              right_cnt <= right_cnt + 1;
26              if 3 <= right_cnt then
27                  DAT      <= right_data(WIDTH-1);
28                  if right_cnt(1 downto 0) = "10" then
29                      right_data <= right_data(WIDTH-2 downto 0) & '0';
30                  end if;
31              end if;
32          end if;
33      end if;
34
35      end if;
36  end process;
37
38 end RTL;
```

4.2 I2S の受信

次は受信モジュールです。送信モジュールと組み合わせてシミュレーションすることで、入力した音声信号を I2S フォーマットに変換し、それが復号される様を観測することができます。

```
1 library ieee;
2
3 use ieee.std_logic_1164.all;
4 use ieee.numeric_std.all;
5
6 entity i2s_decoder is
7     generic (
8         WIDTH : integer := 24
9     );
10    port (
11        CLK : in std_logic;
12
13        BCLK : in std_logic;
14        LRC : in std_logic;
15        DAT : in std_logic;
16
17        LOUT : out std_logic_vector(WIDTH-1 downto 0);
18        ROUT : out std_logic_vector(WIDTH-1 downto 0);
19        LOUT_VALID : out std_logic;
20        ROUT_VALID : out std_logic
21    );
22 end entity i2s_decoder;
23
24 architecture RTL of i2s_decoder is
25
26     attribute mark_debug : string;
27
28     signal left_data : std_logic_vector(WIDTH-1 downto 0) := (others => '0');
29     signal left_cnt : unsigned(5 downto 0) := (others => '0');
30     signal left_valid : std_logic := '0';
31
32     signal right_data : std_logic_vector(WIDTH-1 downto 0) := (others => '0');
33     signal right_cnt : unsigned(5 downto 0) := (others => '0');
34     signal right_valid : std_logic := '0';
35
36     signal lrc_d : std_logic := '0';
37
38     attribute mark_debug of left_data : signal is "true";
39     attribute mark_debug of left_valid : signal is "true";
40     attribute mark_debug of left_cnt : signal is "true";
41
42     attribute mark_debug of right_data : signal is "true";
43     attribute mark_debug of right_valid : signal is "true";
44     attribute mark_debug of right_cnt : signal is "true";
45
```

次のページに続く

```
1      signal bclk_d : std_logic := '0';
2
3 begin
4
5   LOUT_VALID <= left_valid;
6   ROUT_VALID <= right_valid;
7
8   LOUT <= left_data;
9   ROUT <= right_data;
10
11  process(CLK)
12  begin
13    if rising_edge(CLK) then
14      bclk_d <= BCLK;
15
16    if bclk_d = '0' and BCLK = '1' then
17
18      left_data <= left_data(WIDTH-2 downto 0) & DAT;
19      right_data <= right_data(WIDTH-2 downto 0) & DAT;
20
21      lrc_d <= LRC;
22
23      if lrc_d = '1' and LRC = '0' then
24        left_cnt <= (others => '0');
25      else
26        if left_cnt = WIDTH-1 then
27          left_valid <= '1';
28        else
29          left_valid <= '0';
30        end if;
31        if left_cnt <= WIDTH-1 then
32          left_cnt <= left_cnt + 1;
33        end if;
34      end if;
35
36      if lrc_d = '0' and LRC = '1' then
37        right_cnt <= (others => '0');
38      else
39        if right_cnt = WIDTH-1 then
40          right_valid <= '1';
41        else
42          right_valid <= '0';
43        end if;
44        if right_cnt <= WIDTH-1 then
45          right_cnt <= right_cnt + 1;
46        end if;
47      end if;
48    end if;
49  end if;
50  end process;
51
52 end RTL;
```

5 発展

紹介したアプリケーション実験を応用して、次のような課題に挑戦してみましょう。

1. ストップウォッチモジュールを改造して、カウントダウンできるようにしてみましょう。
2. いろいろな通信速度で RS-232-C 通信を試してみましょう。
3. 音声入出力ができるようになったので、たとえば、規定値より大きな値がきたときだけ音声を出力する、平滑化フィルタで音を滑らかにする、などが簡単には考えられるでしょう。

Vivado Design Suite のインストール

Vivado をインストールして、いつでも FPGA の開発ができる環境を整えておこう。実際に FPGA を使うだけではなく、文法の確認やシミュレーションによる動作の確認にも利用できますよ。

1 はじめに

ZYBO Z7-20 をはじめ、Xilinx の FPGA を利用するためには Xilinx の提供する開発環境 Vivado Design Suite(以下 Vivado)をパソコンにインストールしておく必要があります。Vivado は Windows および Linux で利用することができます。ここでは、Windows10 へのインストールをステップ毎に紹介します。

実際に FPGA を使わない場合でも、文法のチェックや合成・配置配線によるリソース使用量のチェック、シミュレーションによる動作の確認を行なうことができますのでインストールしておくと便利です。C/C++ を使った FPGA 開発用のツールである Vivado HLS も一緒にインストールできます。

2 インストーラのダウンロード

まずはインストーラを、Xilinx の Web ページからダウンロードします。

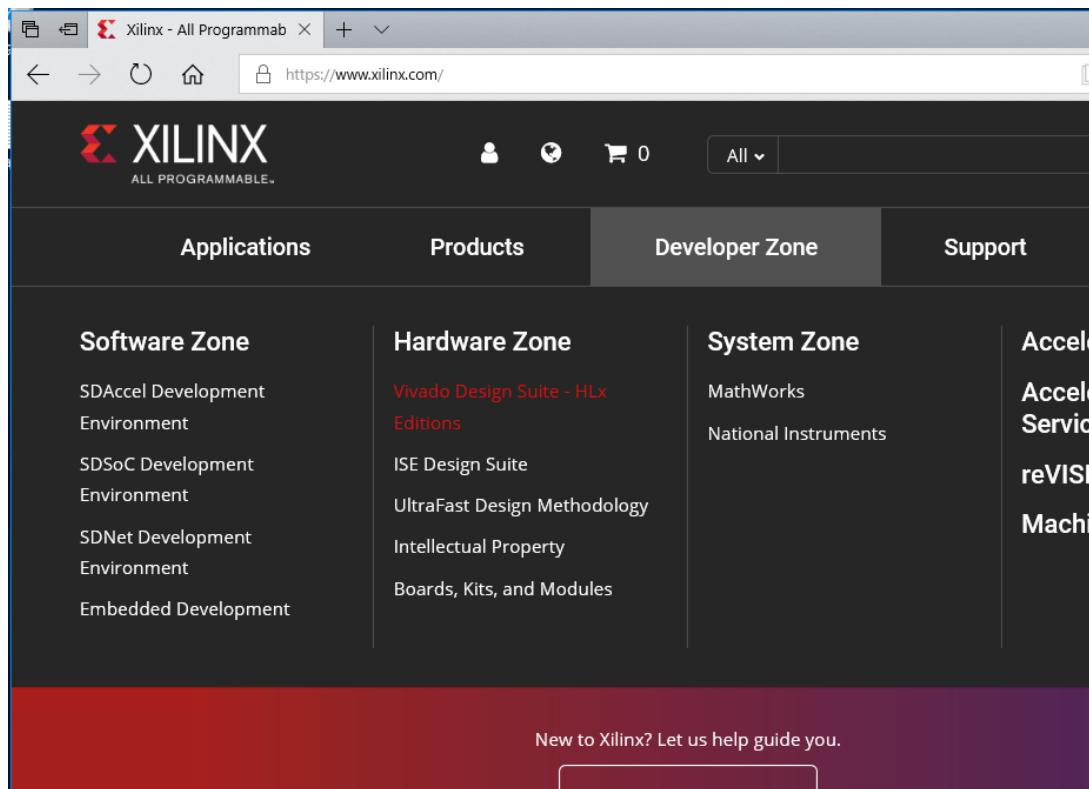


図 1 Xilinx の Web ページを訪れ, “Developer Zone” から “Vivado Design Suite - HLx Editions” を選ぶ

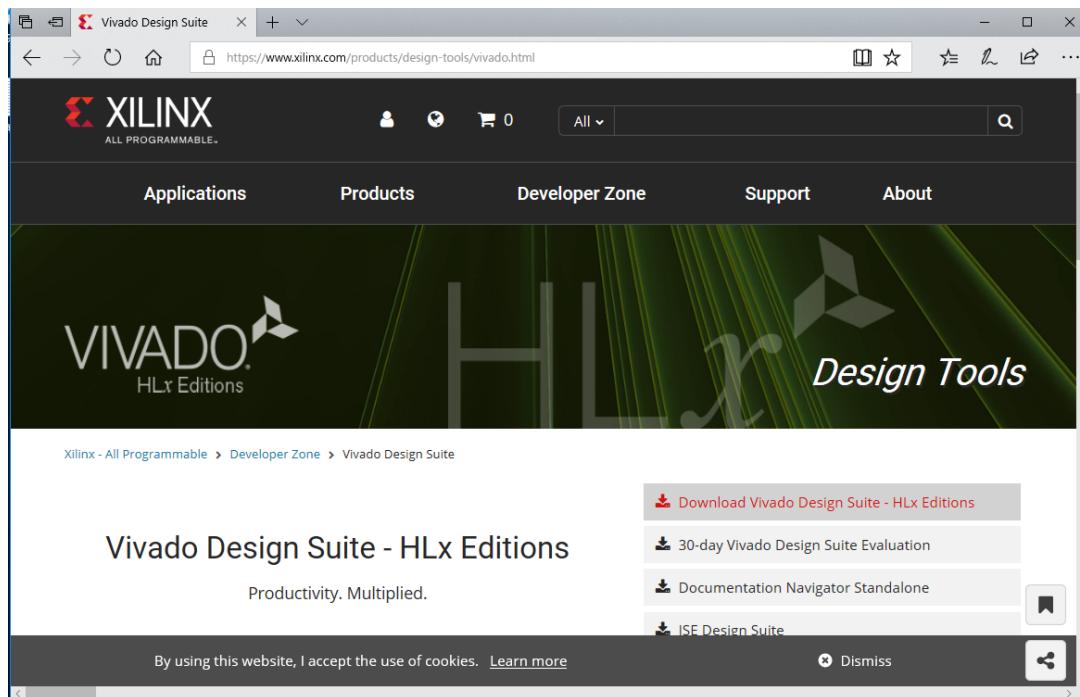


図 2 “Download Vivado Design Suite - HLx Editions” を選択

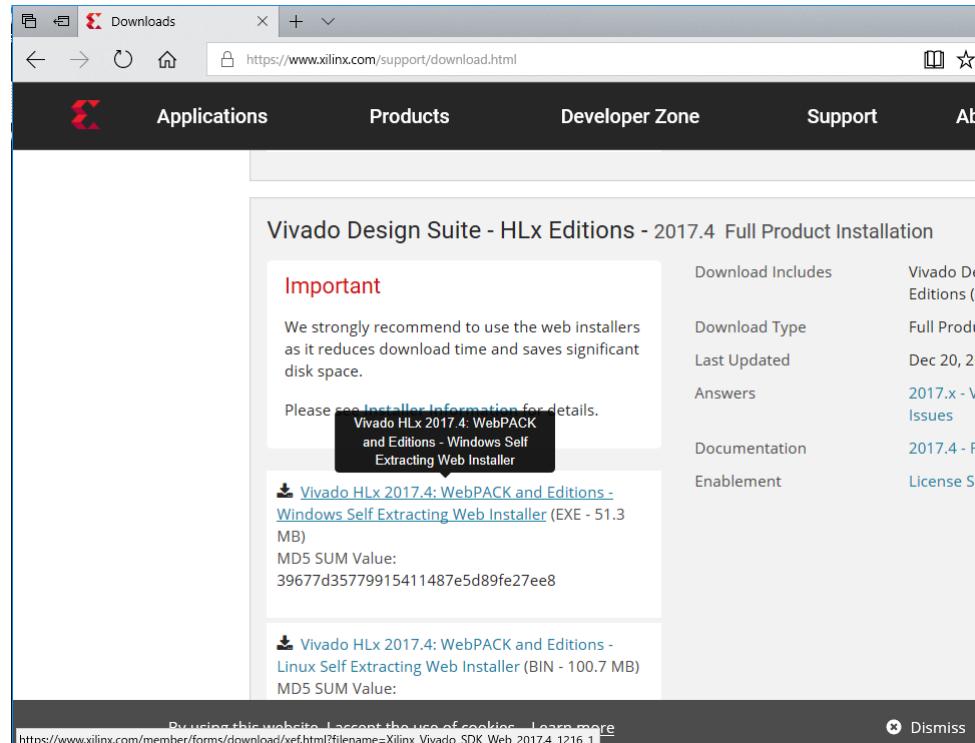


図 3 ページ半程にある Web インストーラをダウンロード。Windows 用と Linux 用があるので注意

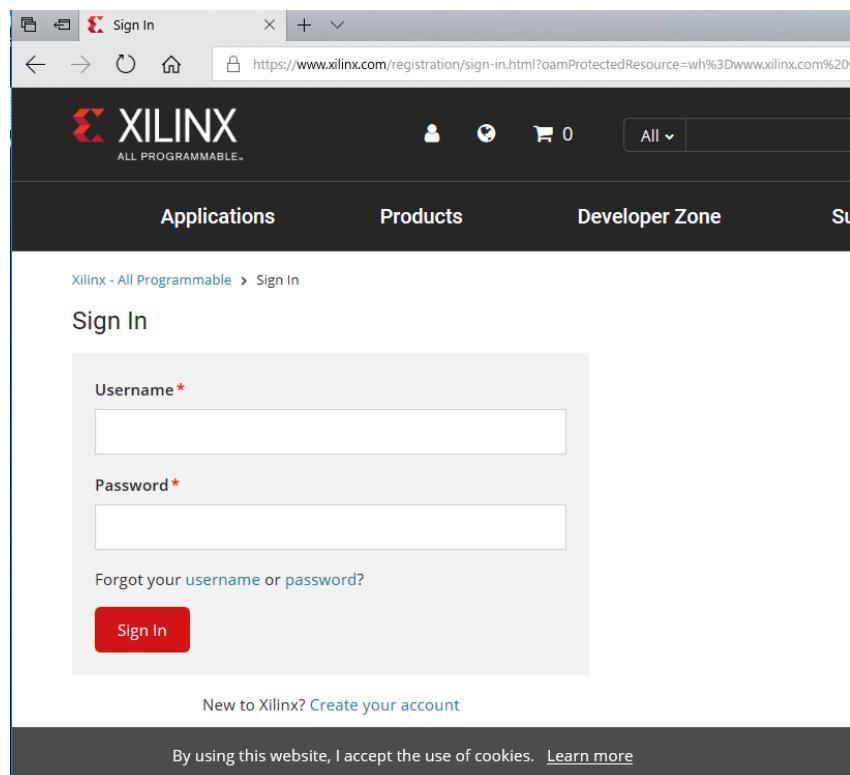


図 4 ダウンロードにはユーザ登録が必要。Create your account からアカウントを作成しましょう

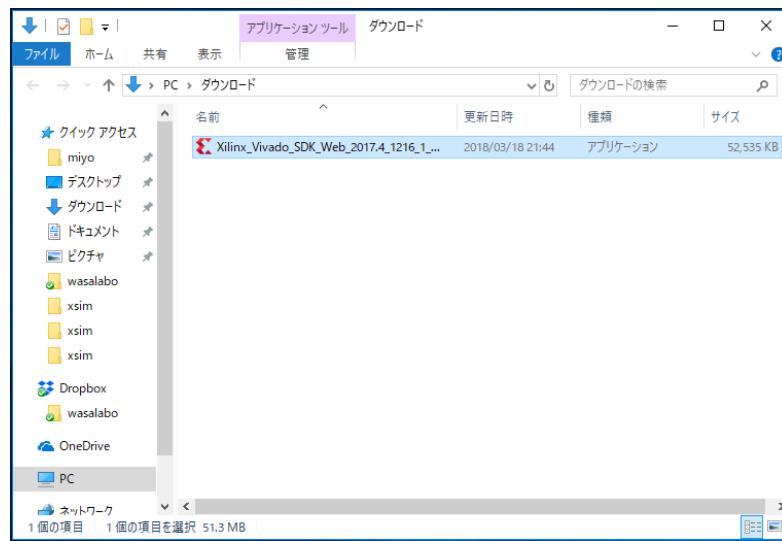


図 5 必要な項目を入力するとインストーラのダウンロードができます

3 インストーラの実行

ダウンロードしたインストーラをダブルクリックで起動して、インストールしましょう。

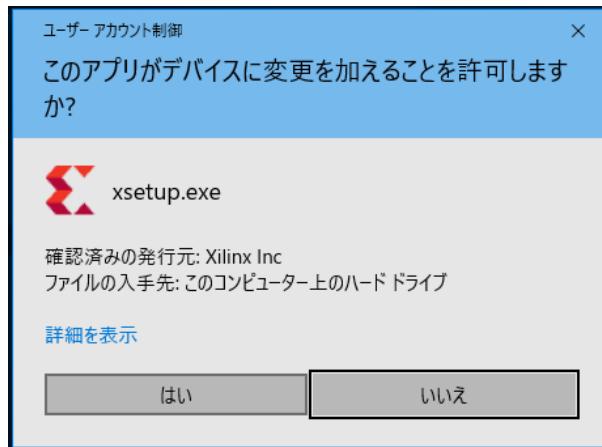


図 6 デバイスへの変更は、もちろん許可してください

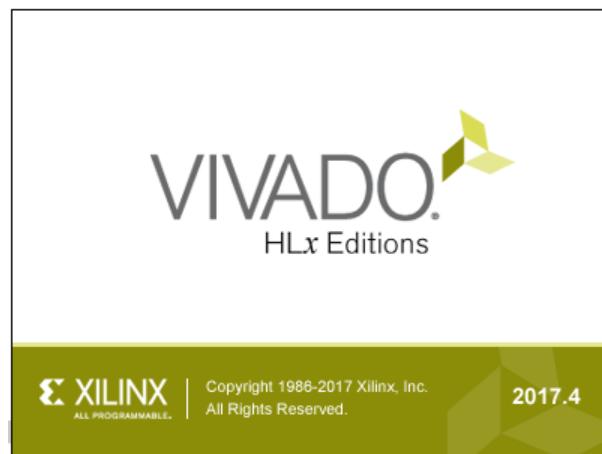


図 7 インストーラが起動するとスプラッシュが表示されます

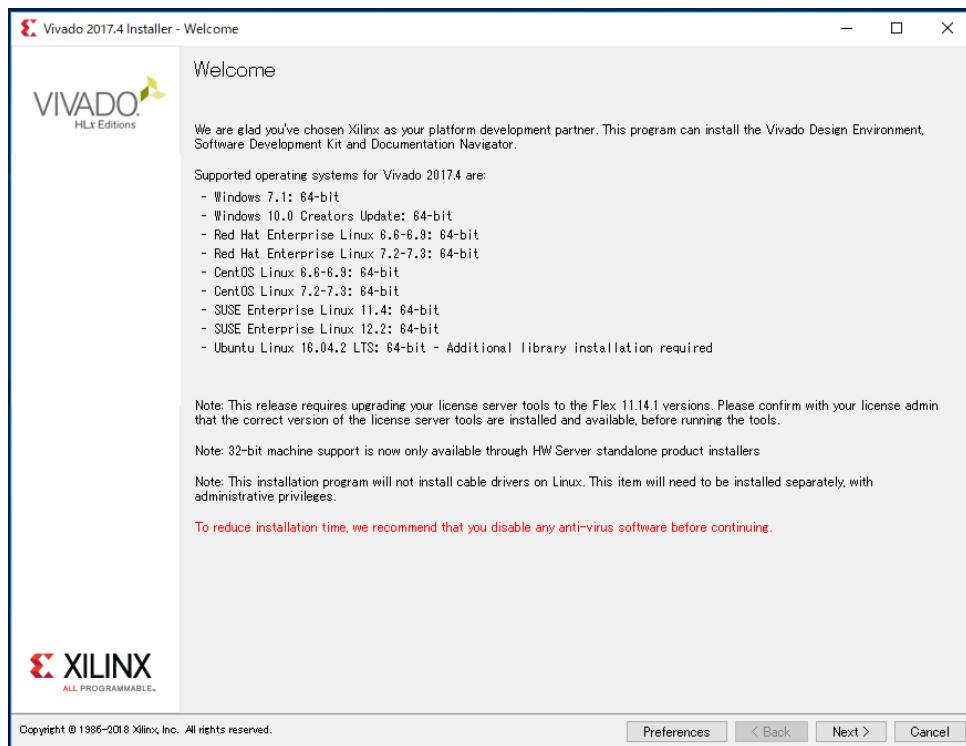


図 8 インストーラが起動しました。ダイアログ下の Next をクリックして次にすすみましょう

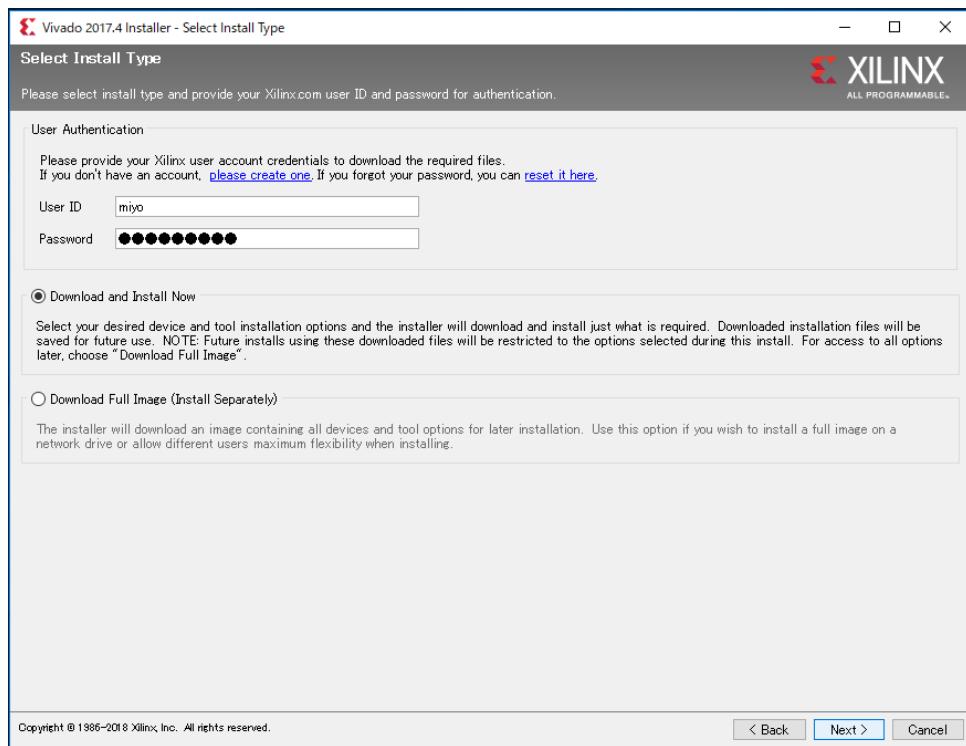


図9 インストーラのダウンロード時に入力したユーザ名とパスワードを、User ID と Password のテキストボックスに入力しましょう。“Download and Install Now”にチェックがはいっていることを確認して、Next をクリックします

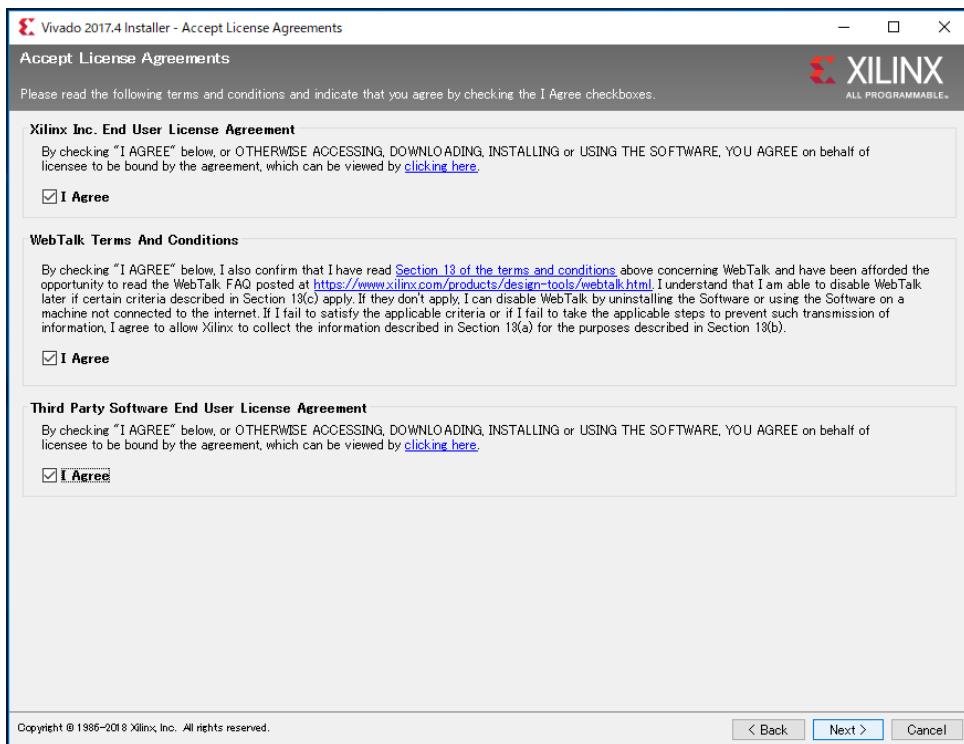


図 10 インストールされる各種ソフトウェアのライセンス確認です。3箇所の“*I Agree*”にチェックを入れて、*Next*をクリックします

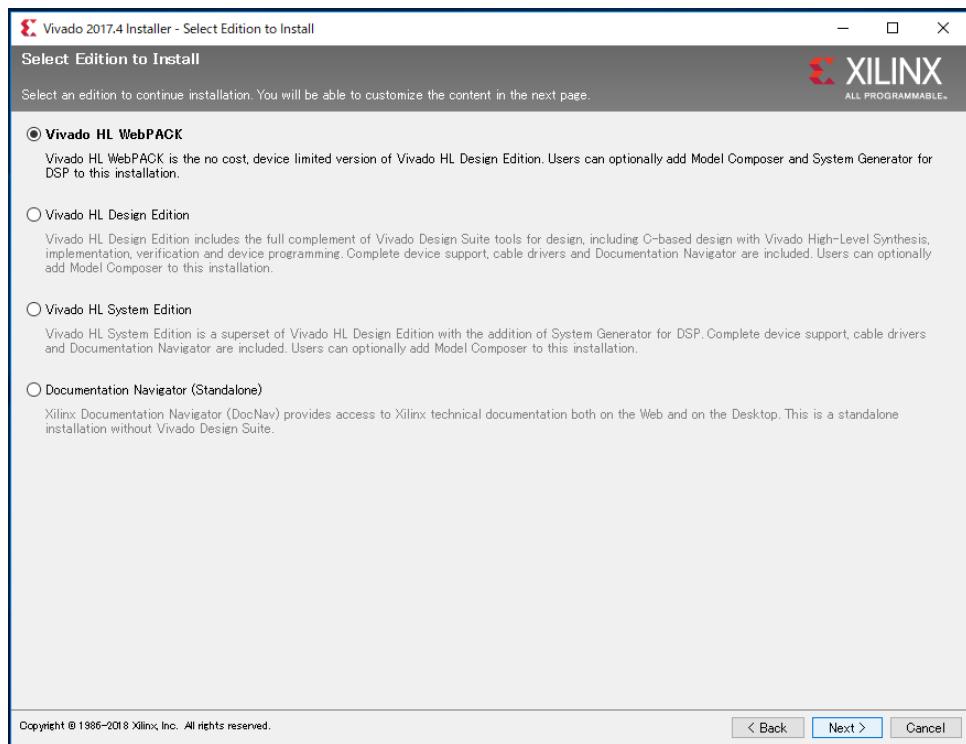


図 11 同じインストーラで様々なグレードの Vivado をインストールできます。ここでは無償で利用可能な “Vivado HL WebPACK” を選択して、Next をクリックします。もし将来的に WebPACK が対応していない大きな FPGA を使用する可能性がある場合には Design Edition を選択しても構いませんが、インストール後に追加することもできます。

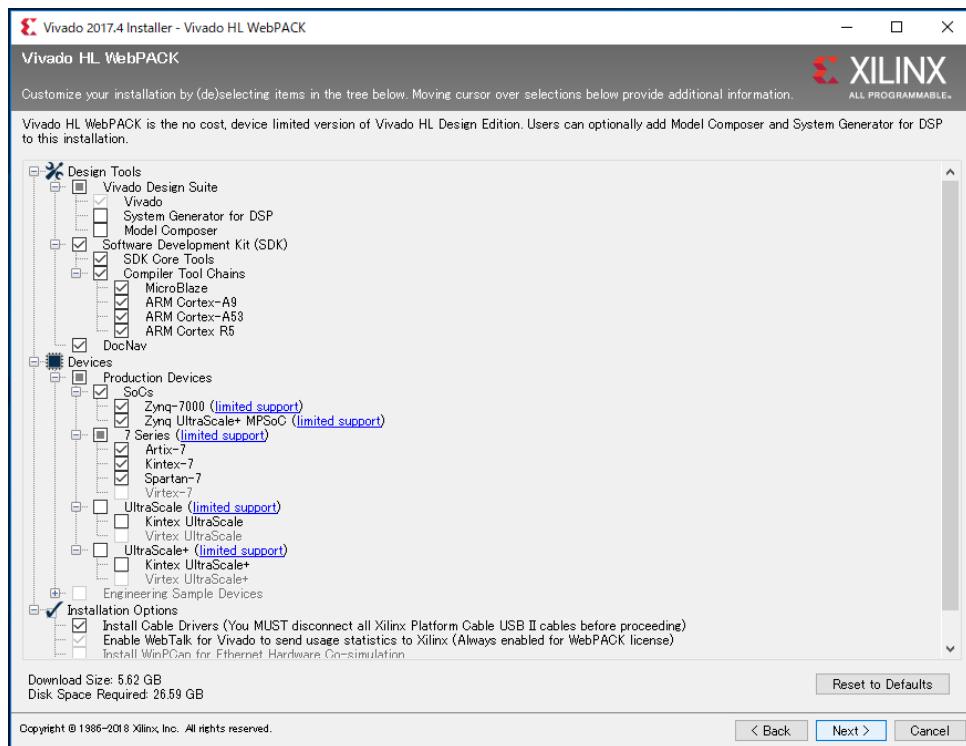


図 12 インストールするツールと対象デバイスを選択します。後々のことを考えると“Software Development Kit のすべてのチェックボックスにチェックを入れておくとよいでしょう。また使用するデバイスは、少なくとも“Zynq-7000”は選択しておきます。他は選択してしなくても構いません。“Install Cable Drivers”にチェックが入っていることを確認して、Next をクリックします。

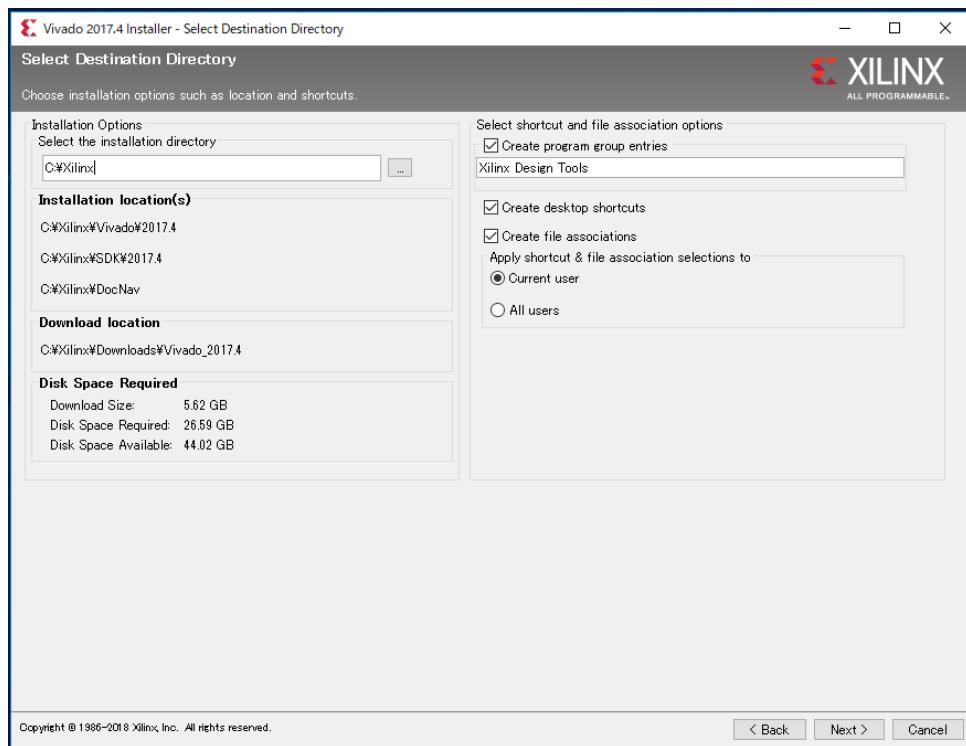


図 13 インストール先の指定です。特に変更する必要はないでしょう。ここでは、今回 26.59GB のディスク容量が必要になることがわかります。Next をクリックして次にすすみます。

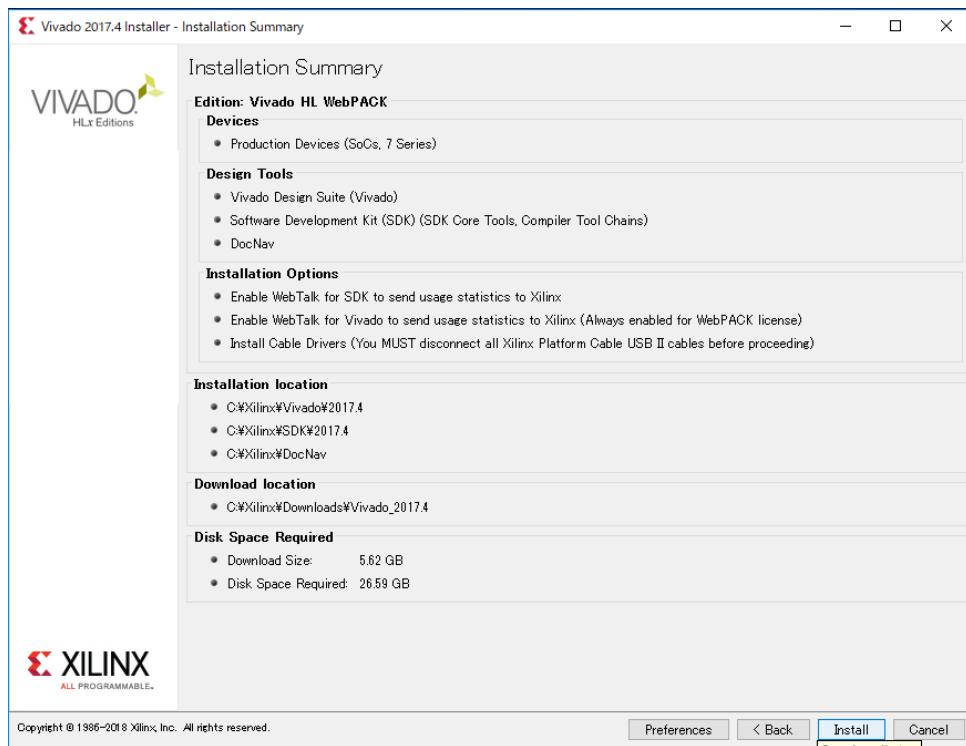


図 14 インストール前の最後の確認です。Next をクリックして次にすすみましょう。

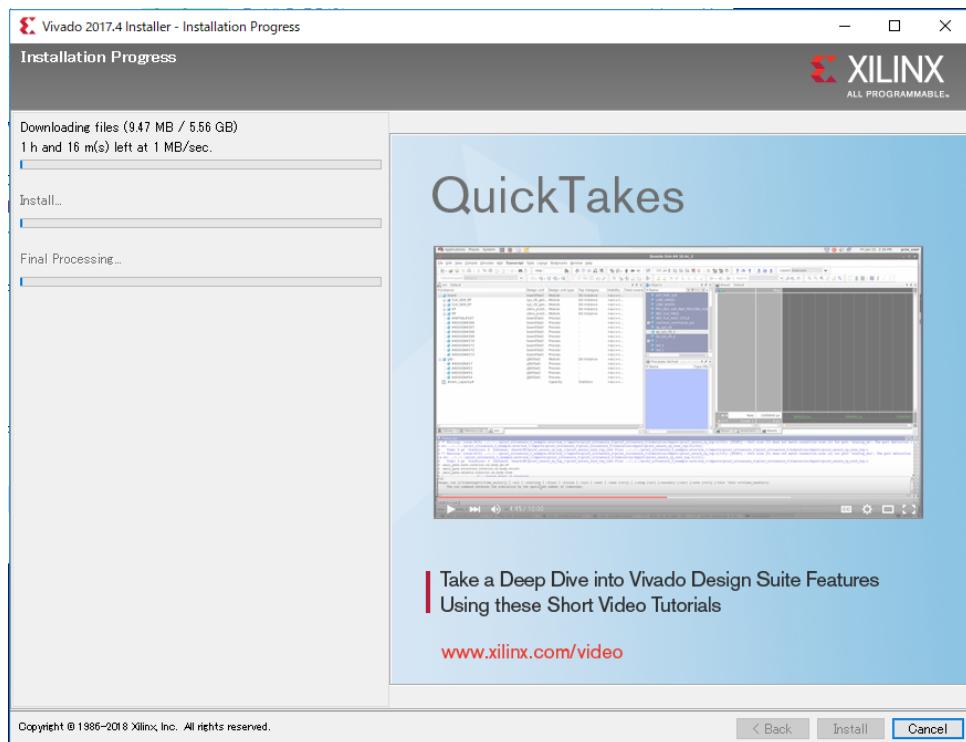


図 15 インストール進行中です。ネットワーク環境やパソコンの性能によりますが、少なくとも 1-2 時間程度の時間は覚悟するとよいでしょう。

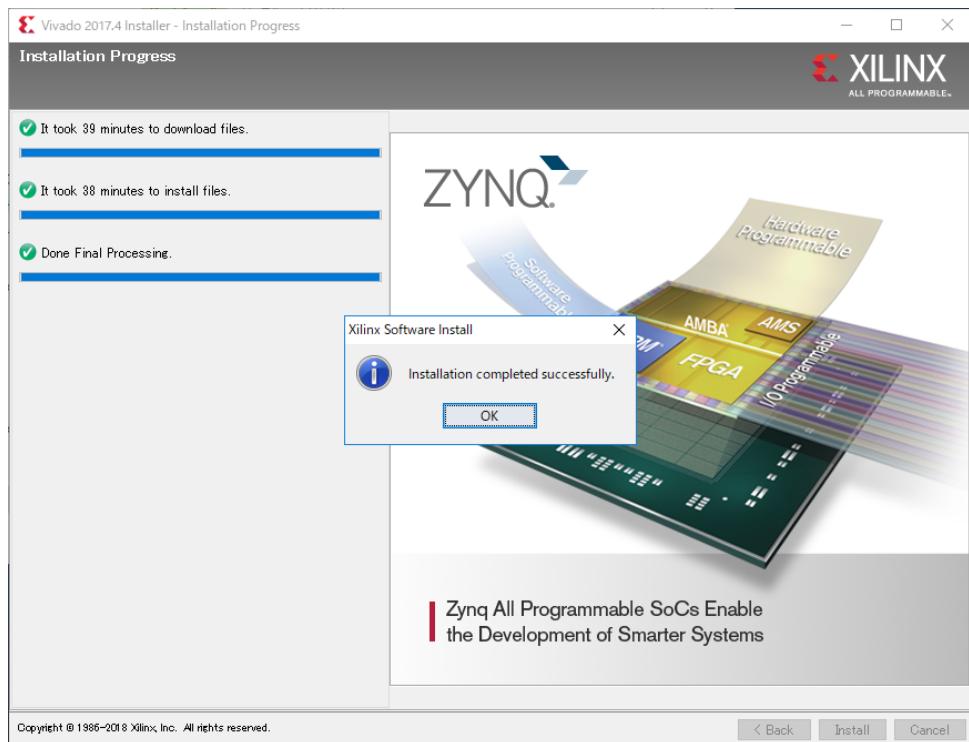


図 16 インストールが完了。この環境では 1 時間 20 分程度必要でした。

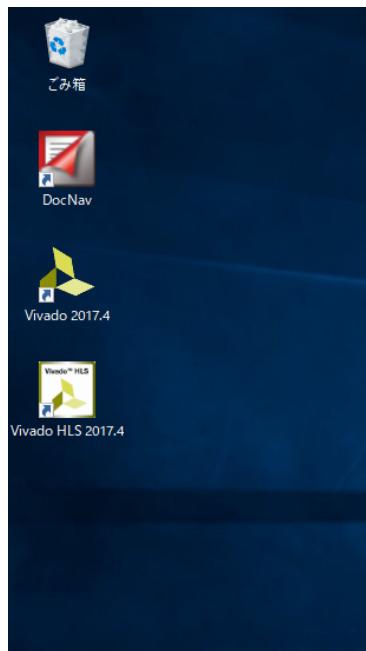


図 17 起動用のショートカット “Vivado 2017.4” がデスクトップに作成されました。

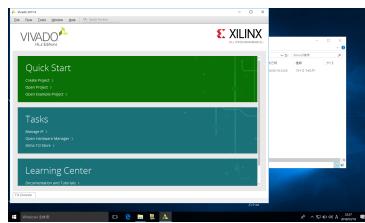


図 18 ショートカット “Vivado 2017.4” をクリックすると Vivado が起動します。

4 ボード定義ファイルのダウンロードとインストール

Vivado のインストールが完了したら、最後に Digilent の提供するボード定義ファイルをダウンロードしてインストールしましょう。これをインストールしておくと、Digilent の提供する FPGA ボードに搭載されている FPGA やメモリ、I/O の構成などのボード情報をライブラリから呼び出すことができるようになります。開発が簡単になります。

まず、Digilent の Web ページである、Installing Vivado and Digilent Board Files(<https://reference.digilentinc.com/vivado/installing-vivado/start>) の半ほどにある、archive というリンク (<https://github.com/Digilent/vivado-boards/archive/master.zip>) からダウンロードします。

ダウンロードしたら、すべて展開し、展開してきた、vivado-boards-master¥vivado-boards-master¥new の下にある board_files をコピーし、C:¥Xilinx¥Vivado¥2017.4¥data¥boards にはりつけましょう。最初から board_files フォルダは存在していますが、構わずはりつけると、Digilent の提供するボード定義ファイルが追加されます。

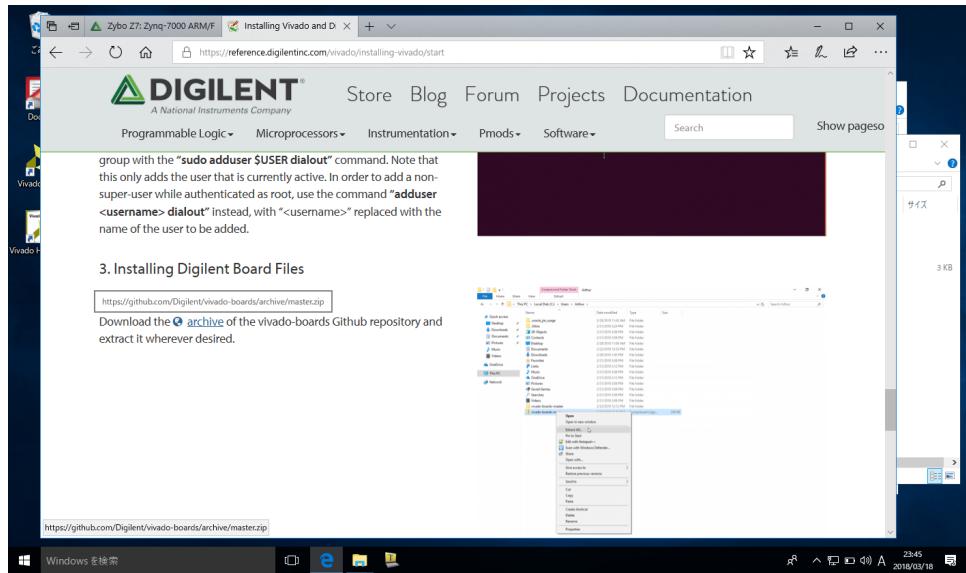


図 19 まずは、アーカイブファイルをダウンロードする。

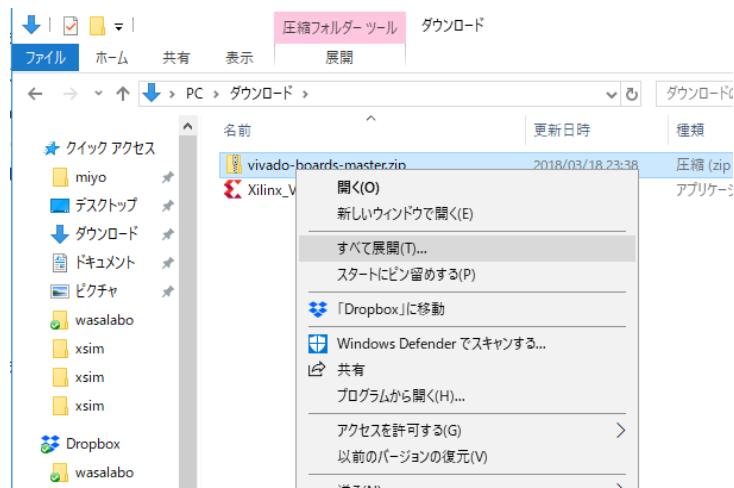


図 20 ダウンロードしたファイルを展開する

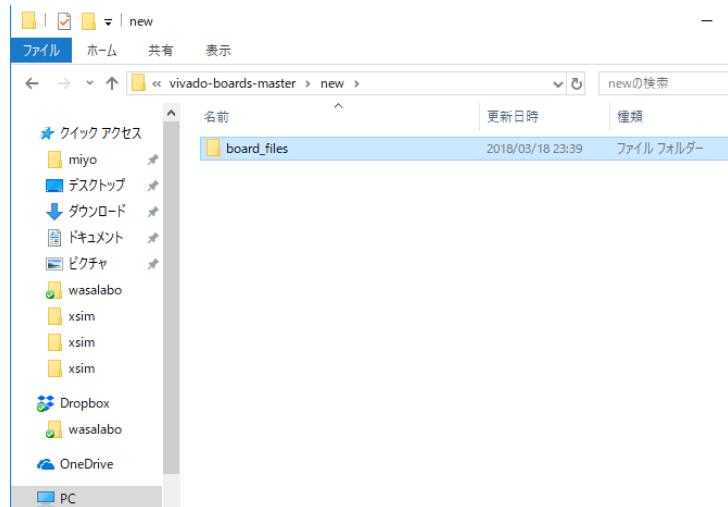


図 21 展開後の vivado-boards-master¥vivado-boards-master¥new の board_file をコピー

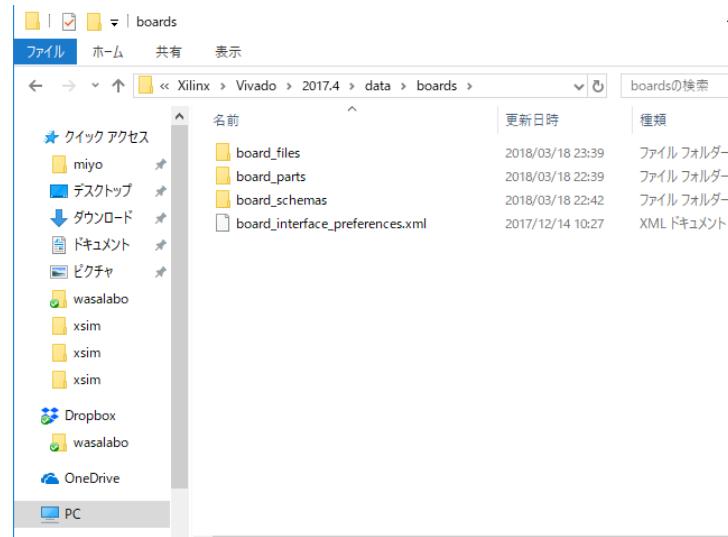


図 22 C:\Xilinx\Vivado\2017.4\data\boards にはりつける