

Zybo Z7-20 ではじめる FPGA

第二部 高位合成編

わさらぼ合同会社
2018年3月19日版

今どきのFPGA開発手法 - 高位合成の利用

FPGAを簡単に活用する手法として高位合成と呼ばれる開発手法に注目が集まっています。この章では、高位合成とは何か？について学びましょう。

1 高位合成とは？

FPGAを使うためには、長い間、VHDLあるいはVerilog HDLによるRTL設計が主流でした。しかし、HDLによるRTL設計は煩雑であり時間のかかる作業で簡単ではありません。そこで、最近では、高位合成という、HDLよりも高い水準での開発を可能とする技術に注目が集まっています。特に、C/C++を利用したFPGA開発手法は、FPGAメーカーであるXilinxとIntelの両方から無償でツールが提供され、簡単はじめられようになりました。

この章では、実際に高位合成を使ってみる前に、高位合成とは何か、ということについて学んでおきましょう。

1.1 FPGAとは - おさらい

Field Programmable Gate Array(FPGA)は、プログラム可能なハードウェアデバイスであり、ユーザが自由にハードウェアロジックをその上に構築できます。そのため、ASIC開発のプロトタイプ環境としての利用に加え、アプリケーションに応じたユーザ独自の専用ハードウェア開発の環境としても利用されます。

FPGAを用いたアプリケーション開発では、プロセッサ上でソフトウェアとして実装する場合に比べ、並列性の活用による低消費電力で高い処理能力の実現が期待される。アーキテクチャを工夫し、データ並列性とパイプライン並列性を活用することで、プロセッサと比較して数十倍から数百倍以上の性能向上が得られます。

また、単に高速に演算処理を実現できるだけではなく、FPGAはアプリケーションを構成する処理回路が入出力信号を直接操作することができるため、低レイテンシ、高スループットで物理デバイスにアクセスできる点に強みを持ちます。加えて、クロックレベルで決定的な処理を実装できることも、アプリケーションを専用ハードウェアとして実装するときの強みです。現代的なプロセッサで動作するソフトウェアでは、キャッシュなどの実行支援ユニットの動作や、複数のプログラムのコンテクスト切り替えなどによって、実行タイミングや処理にかかる時間を正確に管理することが難しくなっています。一方で、専用ハードウェアとしてアプリケーションを実装する場合には、クロック単位での信号の変化を自分で制御することができます。

1.2 FPGA を使う... のは辛い

FPGA の性能を効率良く活用するためには、一般に、VHDL や Verilog を用いた RTL 設計が必要で、プロセッサ上で動作するソフトウェア開発に比べて、人的、時間的な開発コストが大きいのが問題でした。特に、アルゴリズムとして複雑な処理の RTL 記述は繁雑で手間がかかり、時にはバグの温床となっています。また、アプリケーションに合ったアーキテクチャで処理を実装することができれば高い演算性能を達成することができる一方で、そうでない場合には処理性能でプロセッサや ASIC を凌駕することは困難です。そのため、FPGA で高性能処理を実現するためにアーキテクチャ上の試行錯誤をする必要があり、このこともまた、開発コストを大きくする要因です。

1.3 高位合成の登場

ソフトウェア開発のように、手軽に FPGA 開発を行えるように取り組みとして、プログラムをハードウェアロジックに変換する技術が高位合成です。特に、C/C++ で記述されたプログラムをハードウェアロジックに変換する高位合成処理系は、多数開発、販売されています。従来、実用に足るツールは高価だったのですが、2018 年現在では、FPGA メーカーである Xilinx と Intel の両方から Vivado HLS および Intel HLS コンパイラというツールが、無償で提供されるようになりました。

自由に C/C++ の記述能力を利用することはできないものの、C/C++ のループや条件分岐などの制御構文や、処理を関数にまとめてそれを呼び出す処理分割といったメリットを享受することができます。信号処理をソフトウェアで実装した場合にありがちな多重のループ文に対してもパイプライン化やアンロールでハードウェアの特徴を活用できるようになったこと、ターゲット周波数によって実装方式の探索をさせられることなどから、場合によっては手でがんばってハードウェアを設計するよりも、よい回路を生成できることもあるようです。

2 C/C++ ベースの FPGA 開発のながれ

C/C++ ベースの FPGA の開発、特に Xilinx の Vivado HLS を利用する流れを図 1 に示します。要は、C/C++ で記述したソースコードを Vivado HLS によって VHDL または Verilog HDL に変換し、その変換後のコードを Vivado で FPGA 用のビットストリームにする、というのが開発のおおまかな流れです。

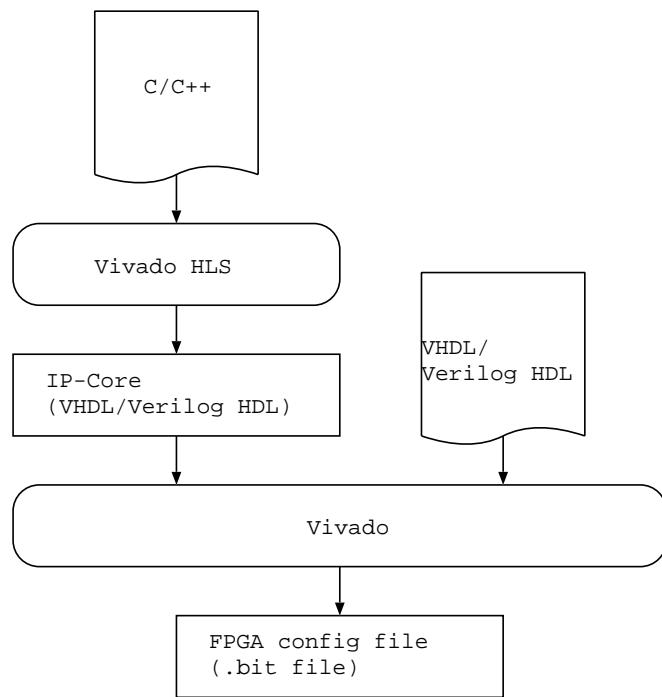


図 1 Vivado HLS を使った開発フロー

C/C++ ベースで作ったモジュールを設計に組み込むためには、ツールへの慣れは必要になりますが、特段難しいものではありません。次章以降、簡単なサンプルでステップを踏んでみていきましょう。

Vivado HLS を使った C ベース設計に挑戦

C/C++ ベースの FPGA 開発をはじめるにあたって、まずは Vivado HLS の使い方を FPGA で動作するビットストリームの作り方まで一通り学んでしまいましょう。

1 はじめに

この章では、とりあえず Vivado HLS を使った開発ができるようになるために、ツールを一通り使って C/C++ で書いたコードから FPGA で動作するビットストリームを生成する作業までを一通り体験してみることにします。習うより慣れろ、ですね。

2 Vivado HLS での開発ステップ

ポインタを使えない、多くの標準関数が使えないという制約はあるものの、基本的な制御構文の C/C++ コードを HDL モジュールにすることができます。ここでは、簡単なプログラムをハードウェア化してみましょう。

```
1 int bitcount(int a)
2 {
3     int i;
4     int s = 0;
5     int tmp = a;
6     for(i = 0; i < 32; i++){
7         if((tmp & 0x01) == 0x01){
8             s++;
9         }
10        tmp = tmp >> 1;
11    }
12    return s;
13 }
```

2.1 Vivado HLS プロジェクトの作成

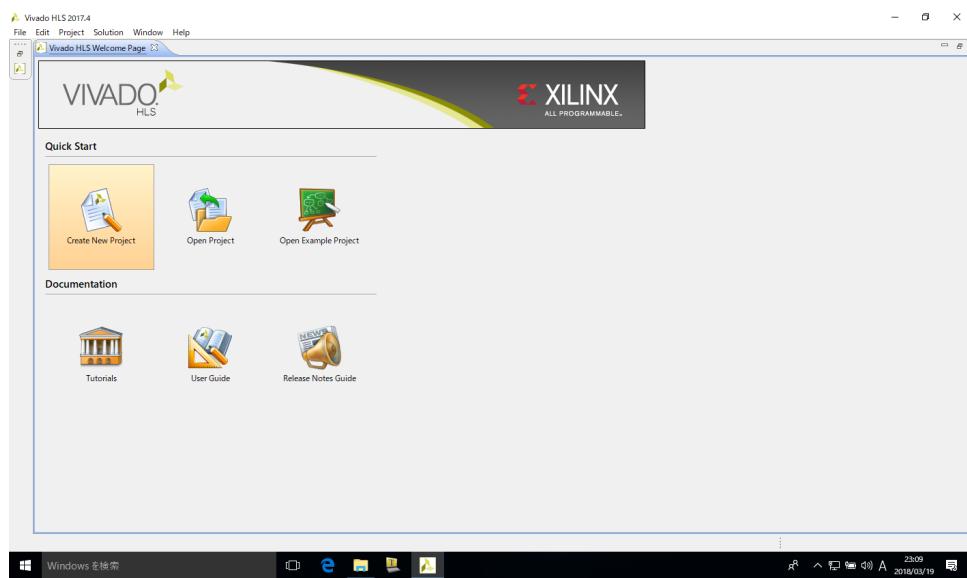


図1 Vivado HLS を起動したところ。Vivado HLS はデスクトップのショートカットアイコンやスタートメニューから起動する。

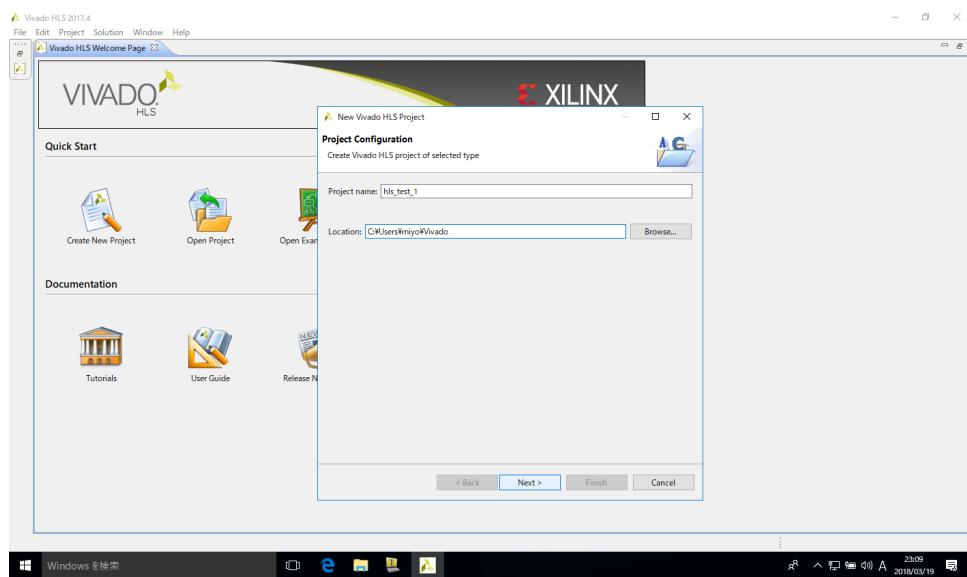


図2 プロジェクト名と格納フォルダを指定。ここでは、ホームの下の Vivado の下に格納することとし、名前を hls_test_1 とした

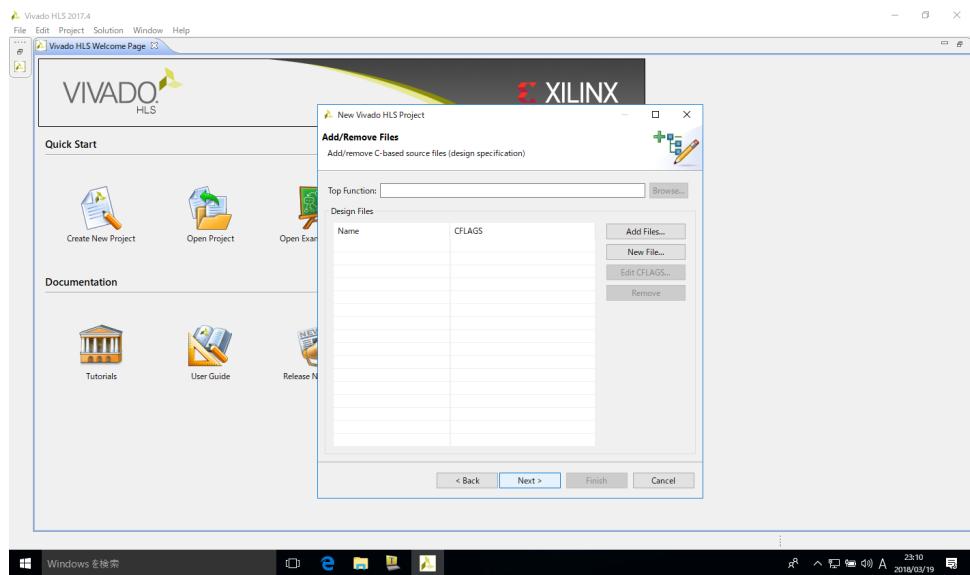


図3 既存の設計ソースコードがあれば、ここで追加。ないので Next で次へ。

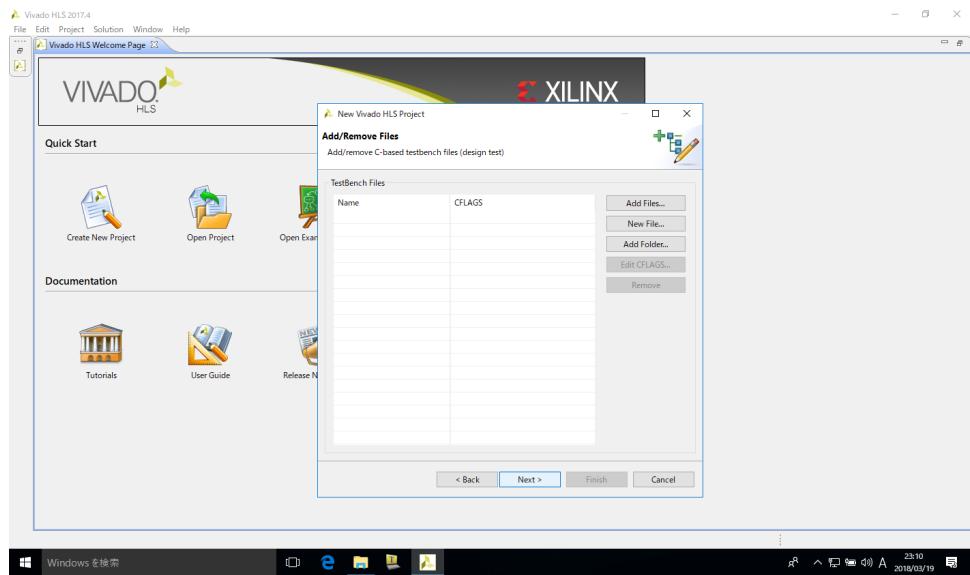


図4 既存のテストベンチ用のソースコードがあれば、ここで追加。ないので Next で次へ。

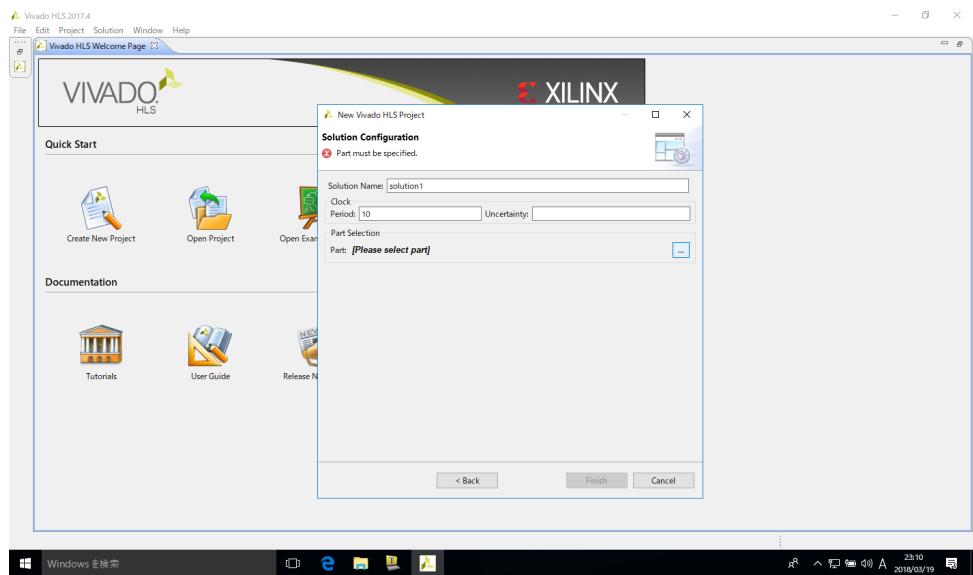


図 5 ターゲット FPGA を選択する。Part Selection の中の... ボタンをクリック

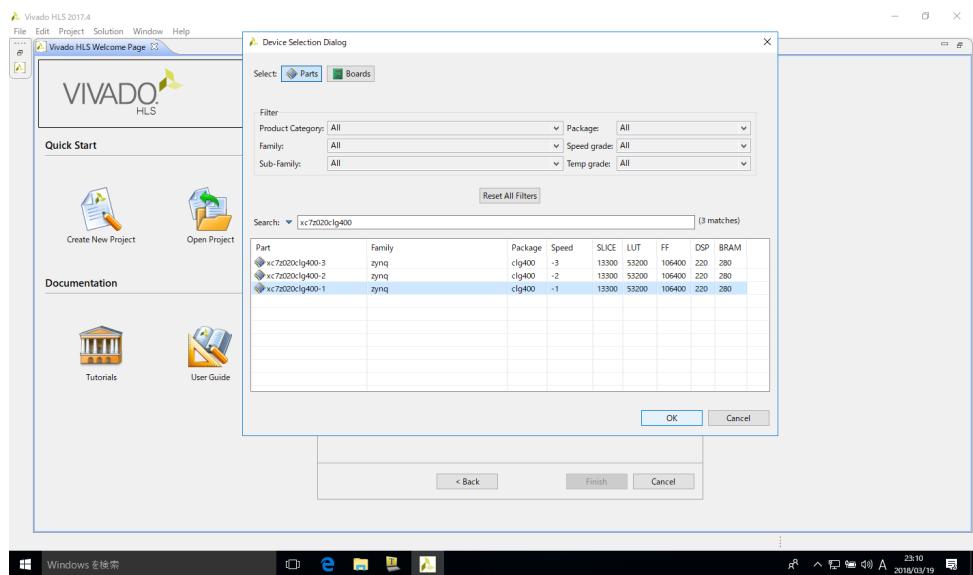


図 6 FPGA の型番を選択。xc7z020c1g400-1 を選択する。検索フィールドに xc7z020 と入力していくと楽に探せる。

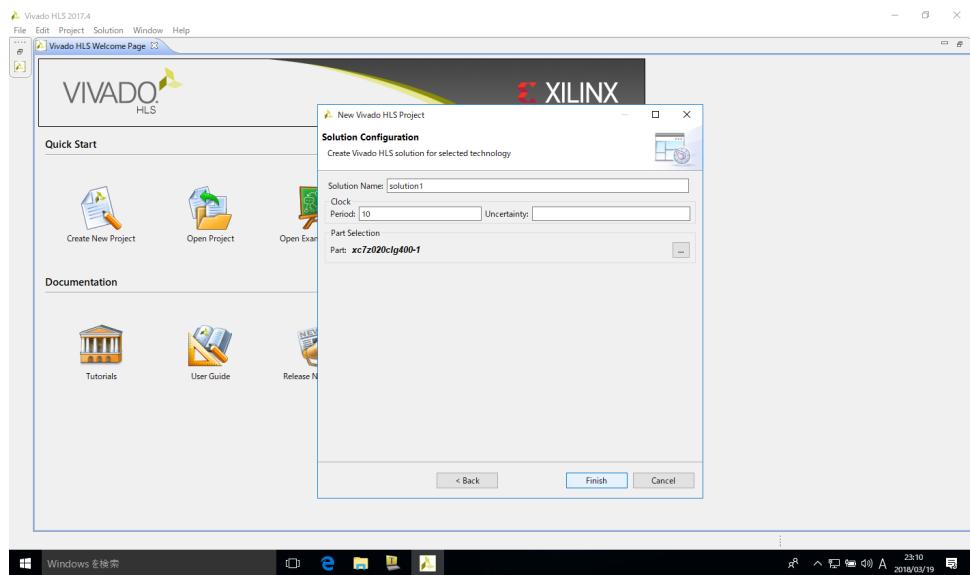


図 7 FPGA を選択し終えたら Finish.

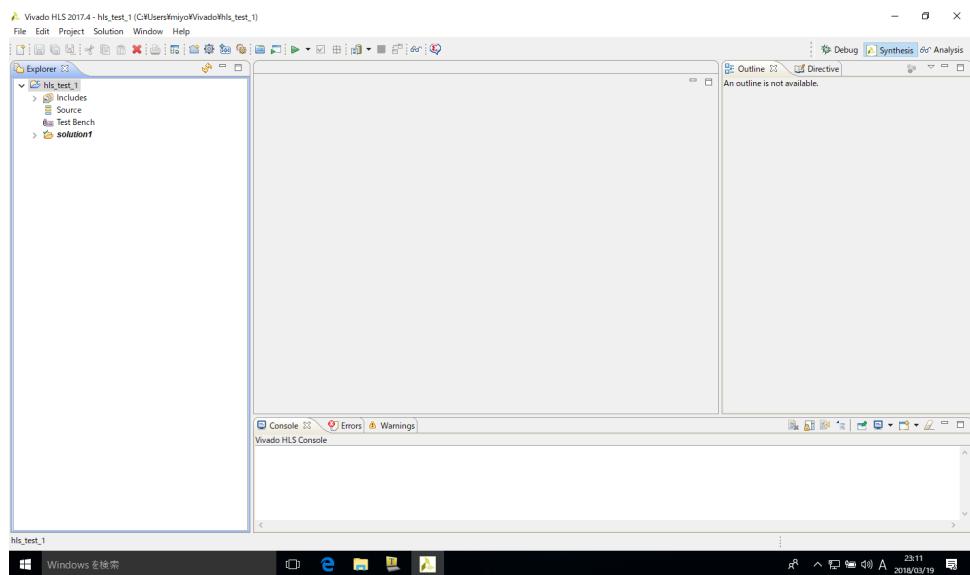


図 8 プロジェクトの作成が完了した

2.2 Vivado HLS 上での設計

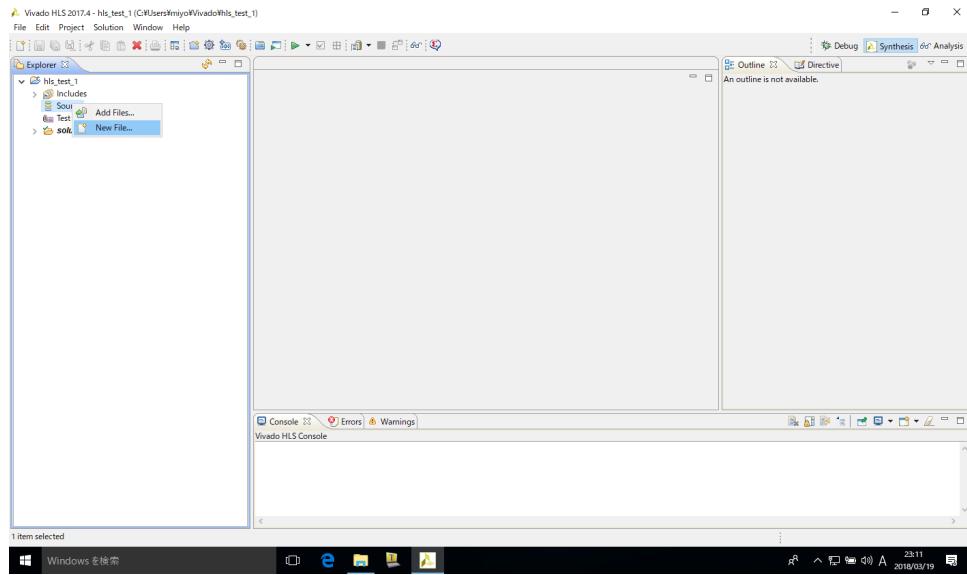


図9 左ペインの Sources アイコンの上で右クリック、New File... をクリック

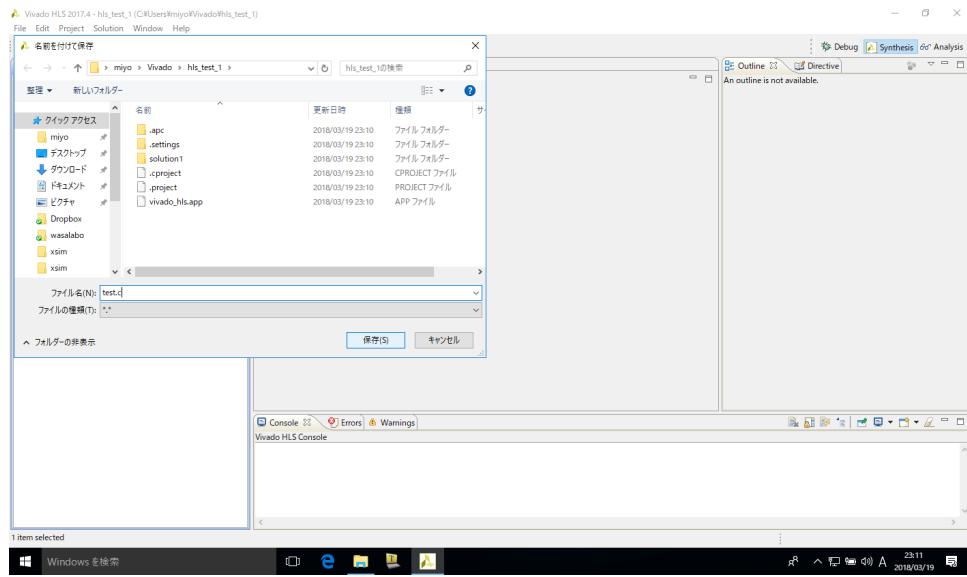


図10 開いたファイル選択ダイアログに、作成するファイル test.c と入力して保存をクリック

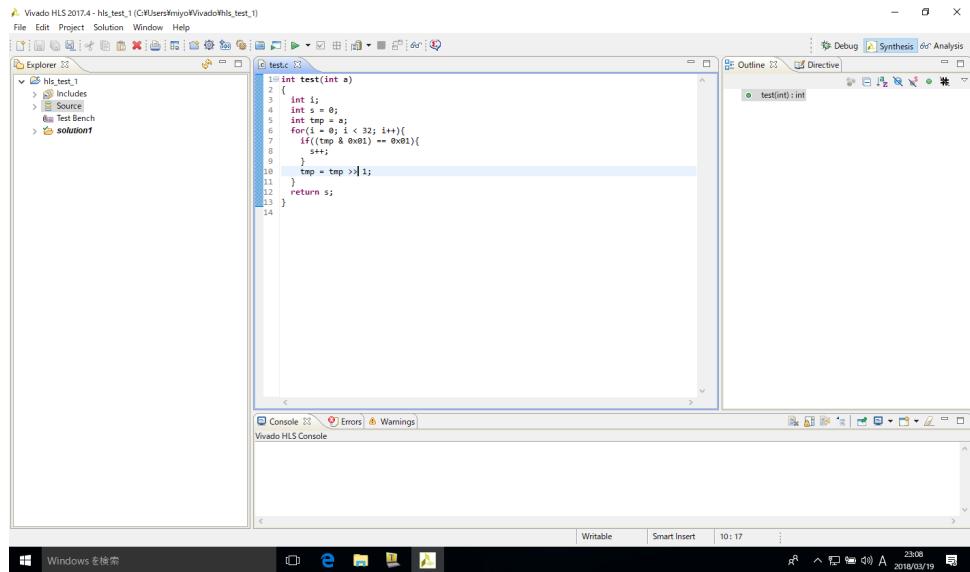


図 11 作成した test.c の中身として、先のソースコードを入力する。

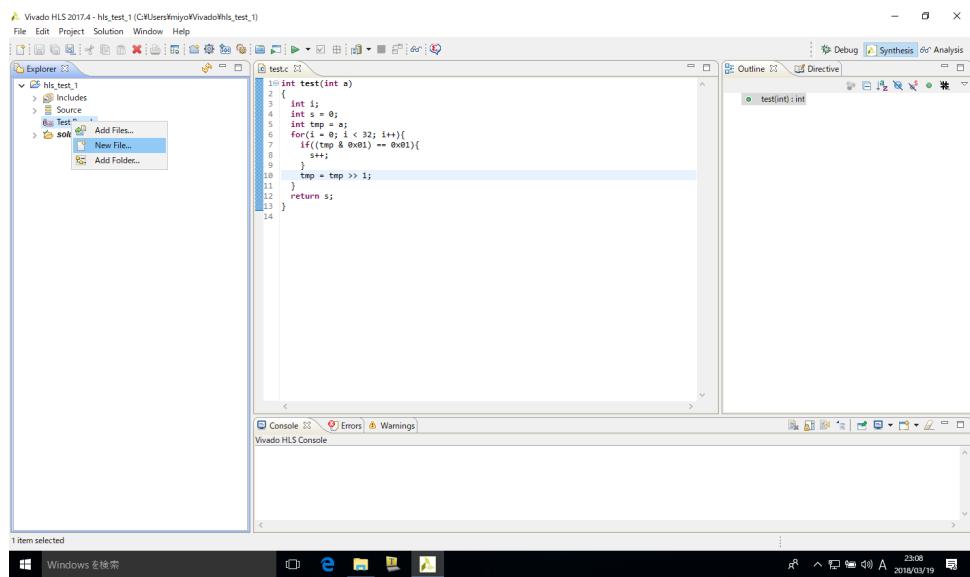


図 12 動作確認のために、テストベンチ用のソースコードを追加する。今度は Test Bench アイコンの上で右クリック、New File... をクリック

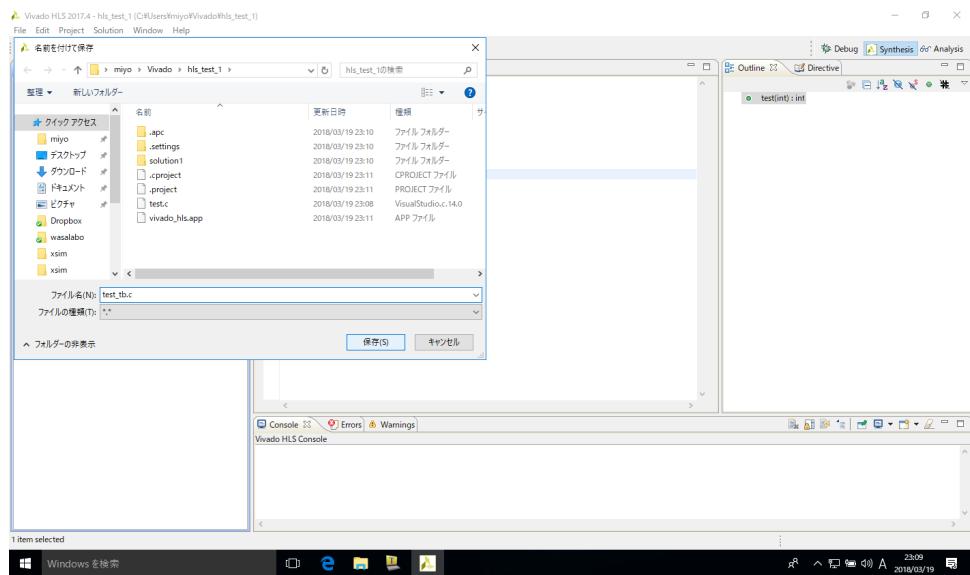


図 13 今度は test_tb.c というファイルを作成することにする。ファイル名を入力して保存をクリック

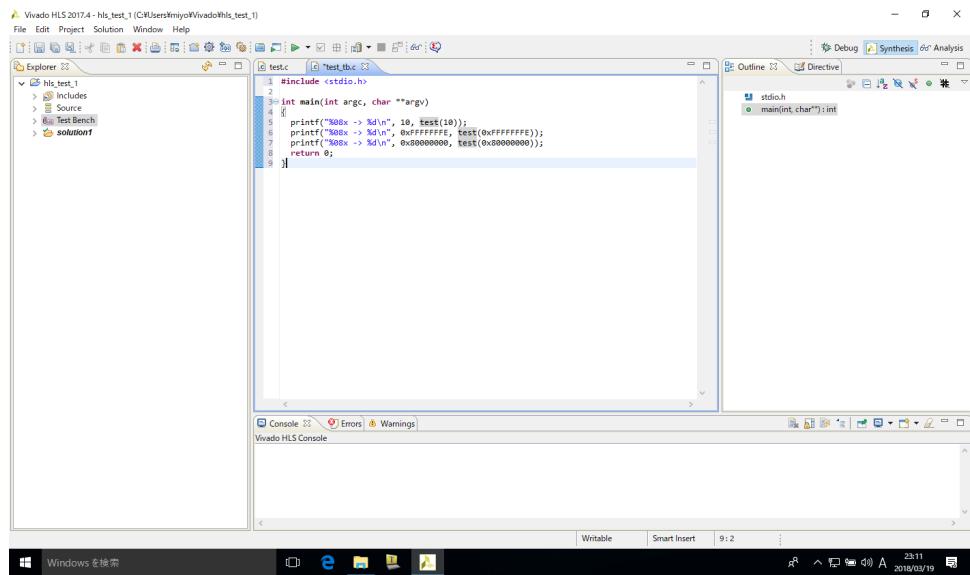


図 14 テストベンチの中身を記述する。テストベンチは単なる C プログラムなので stdio.h や printf 関数が使える

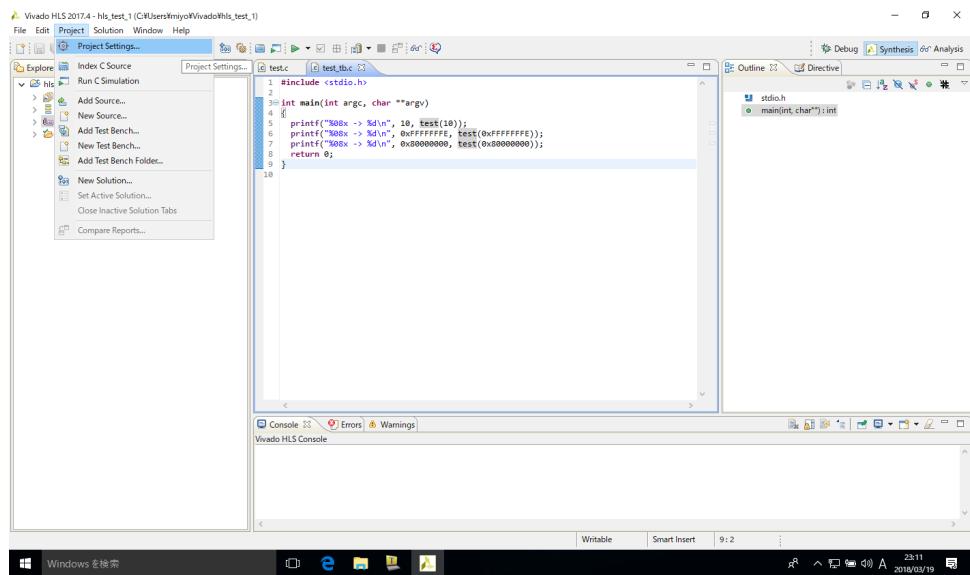


図 15 メニューの Project から Project Settings... をクリックして、プロジェクトの設定をする

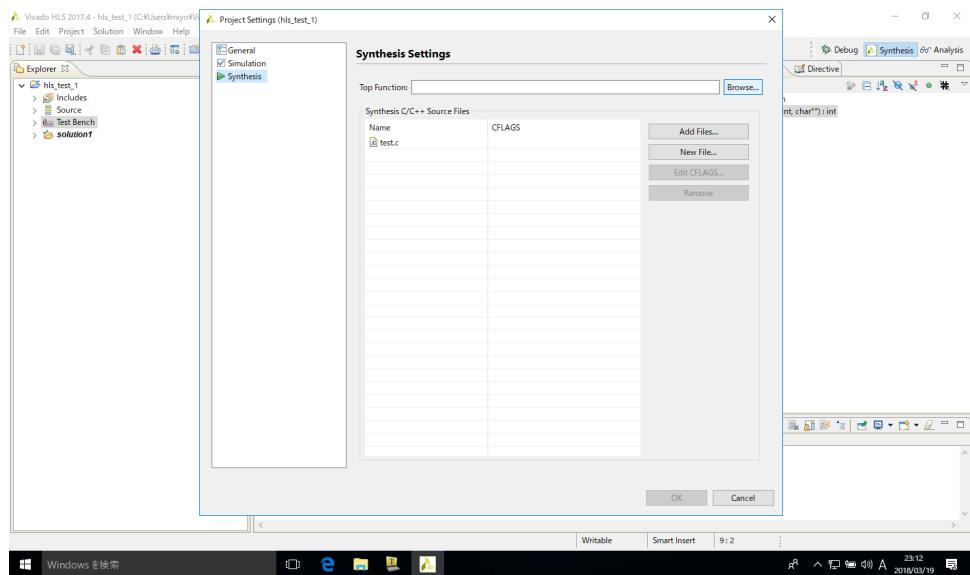


図 16 Synthesis をみると、Top Function が指定されていないことが確認できる

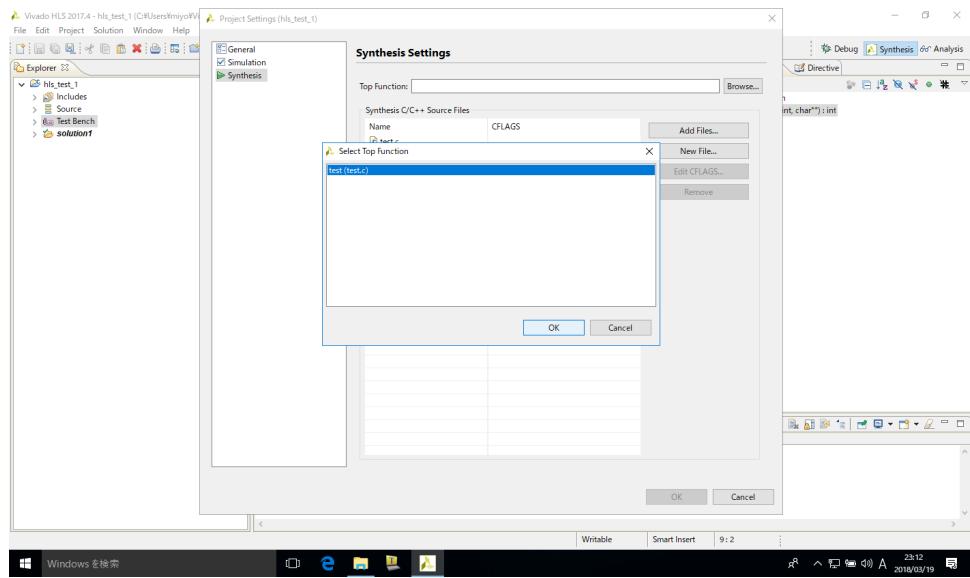


図 17 Browse... ボタンをクリックすると候補ができる。今回は一つだけ。候補に表示された test 関数を選択して OK をクリック

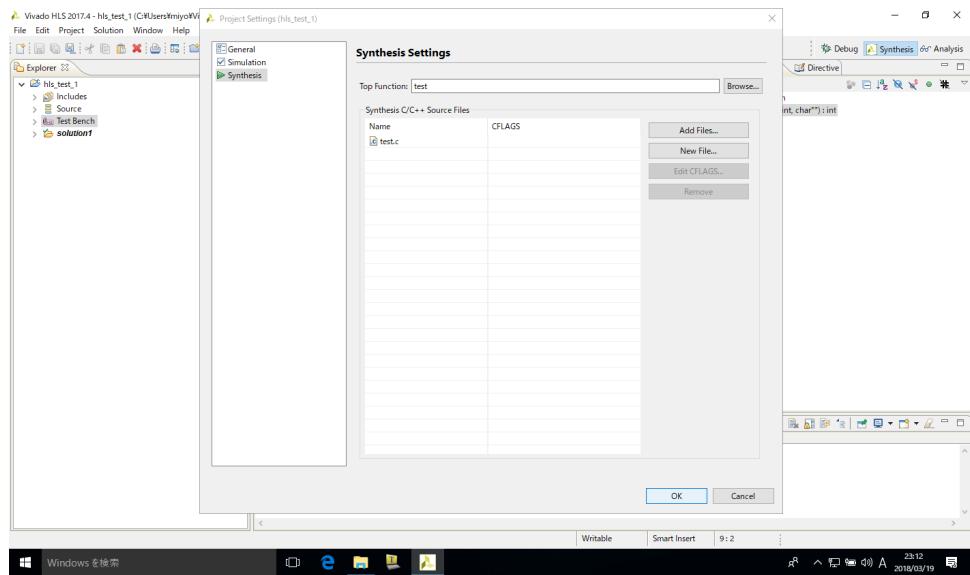


図 18 合成するトップの関数が test であることを指定できた

2.3 Vivado HLS での動作確認

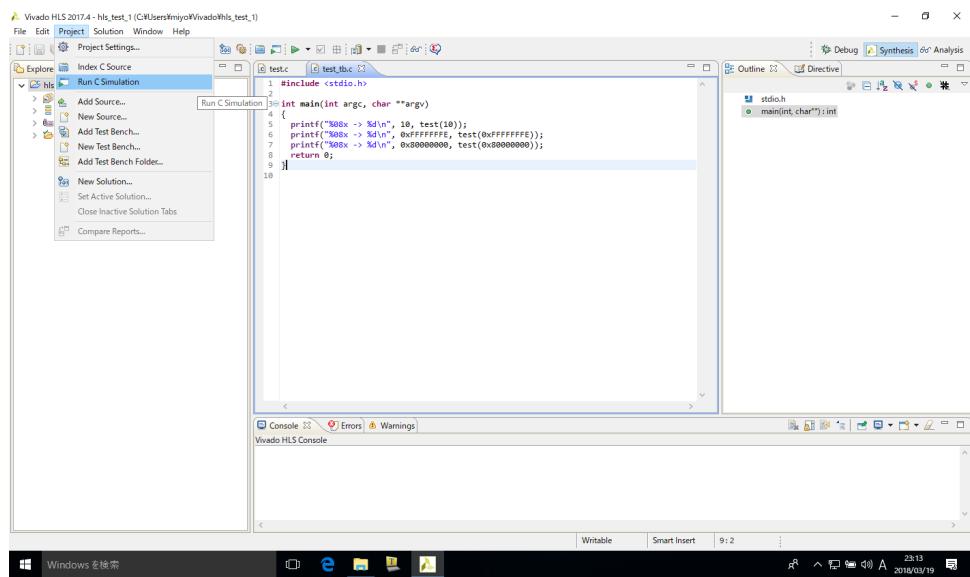


図 19 まずは C レベルでの動作を確認するため、メニューの Project から Run C Simulation をクリックする

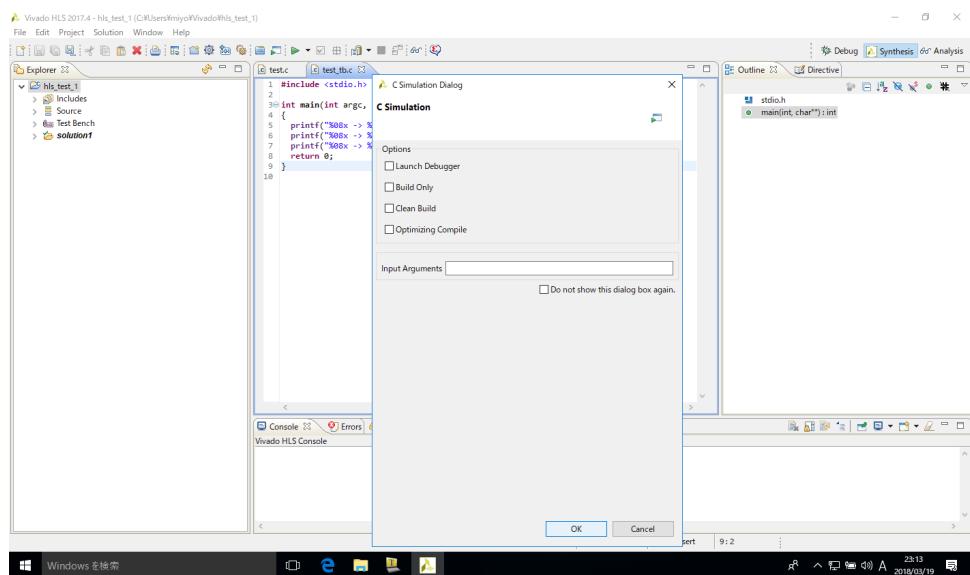


図 20 特にオプションは指定せずに OK をクリック

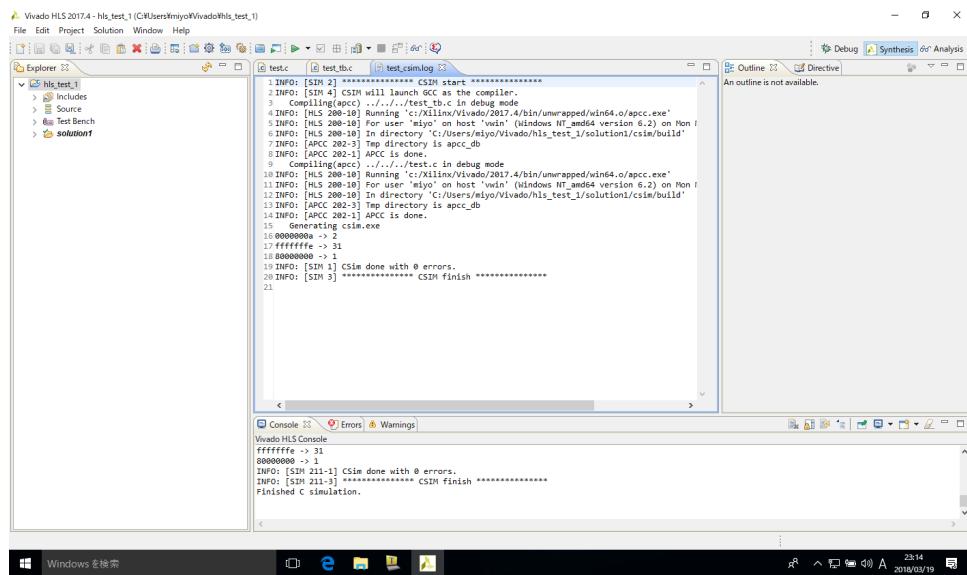


図 21 しばらくすると結果が表示される。たとえば 0xa の立っているビットは 2 個で答えが正しいことが確認できた

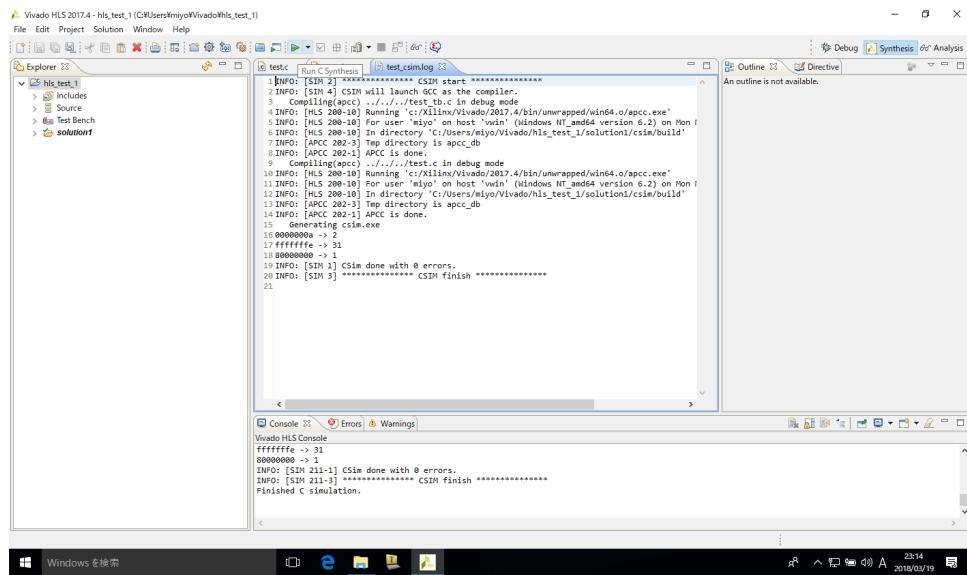


図 22 ツールバーの再生ボタンアイコンをクリックして C から HDL の合成を開始する

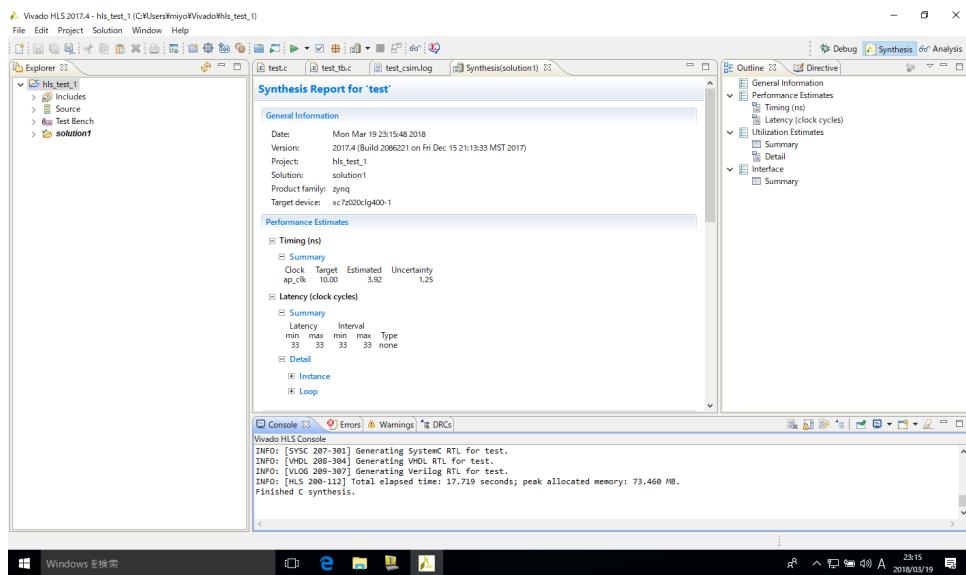


図 23 しばらく待つと合成が完了する。ここでは生成された回路のクリティカルパス遅延が 3.92ns であると表示されている。また、ひとつの答えを得るために 33 サイクル必要であることも確認できる。

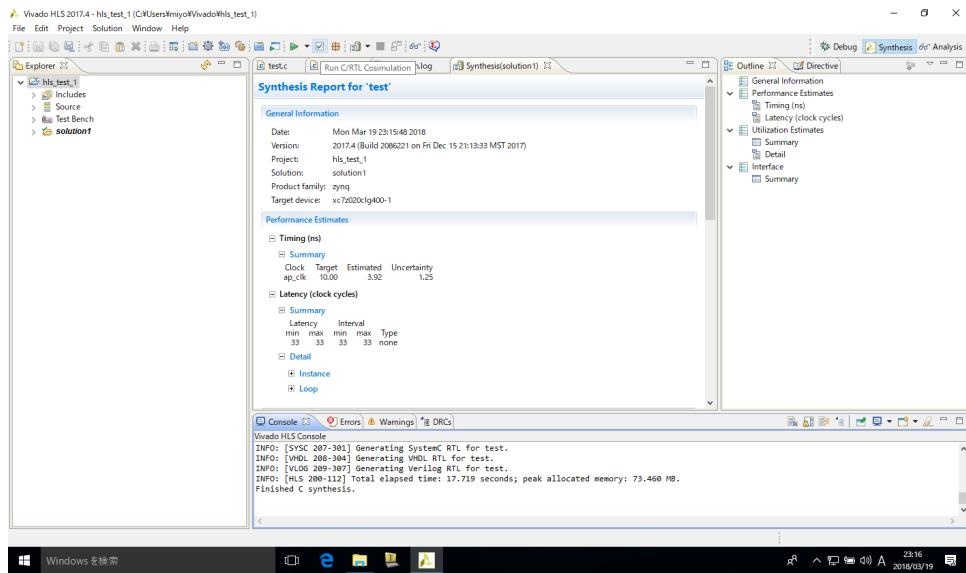


図 24 合成した HDL の動作検証を行う。テストベンチは C シミュレーションのときと同じ

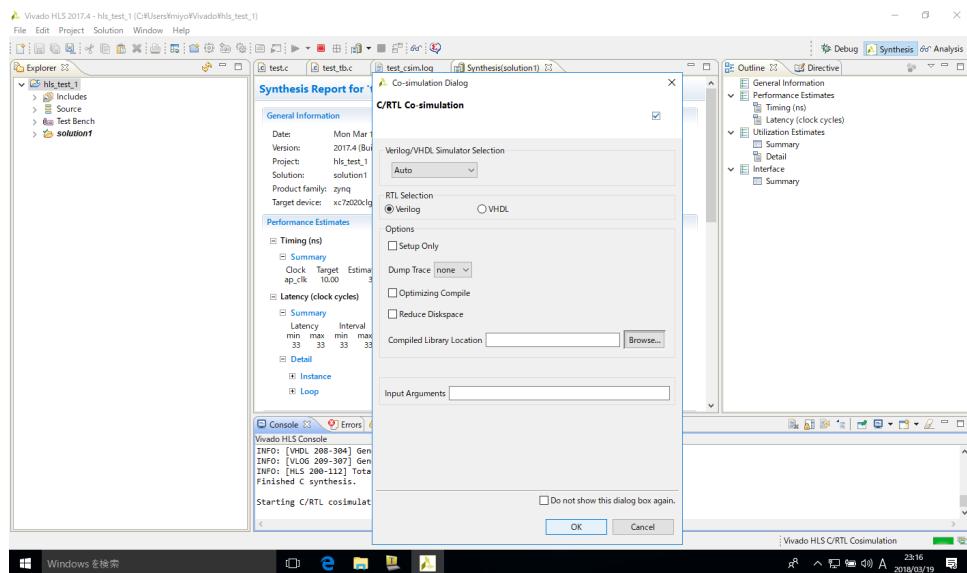


図 25 特にオプションは指定せずに OK をクリック

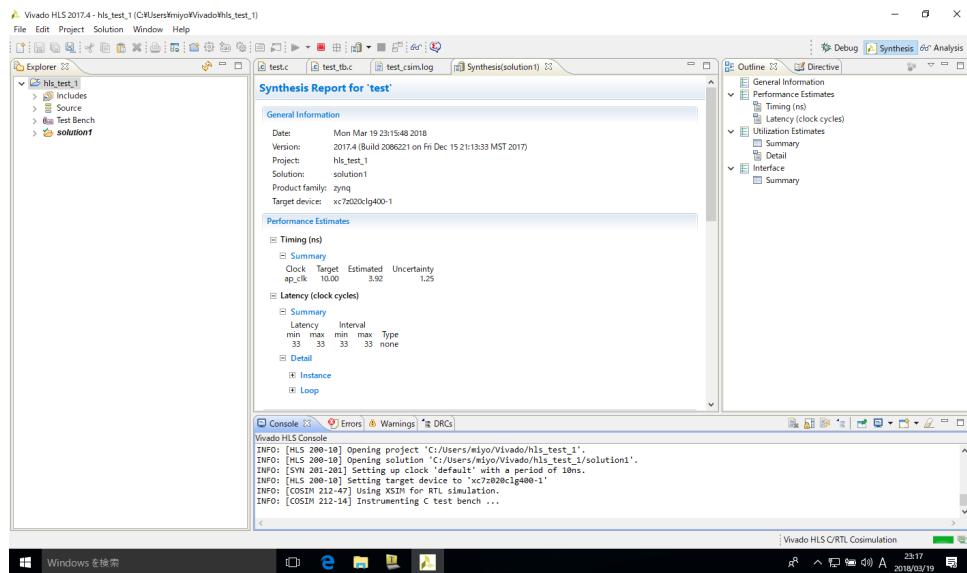


図 26 シミュレーションに XSIM(RTL シミュレータ)が使用されていることがわかる

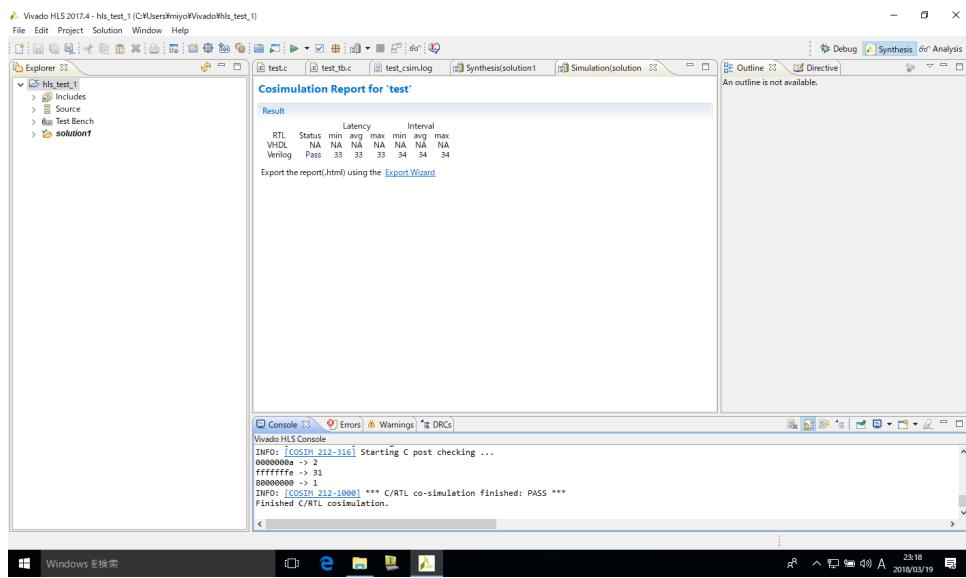


図 27 RTL シミュレーションでも望み通りの答えがえられることが確認できた

2.4 IP コアの生成

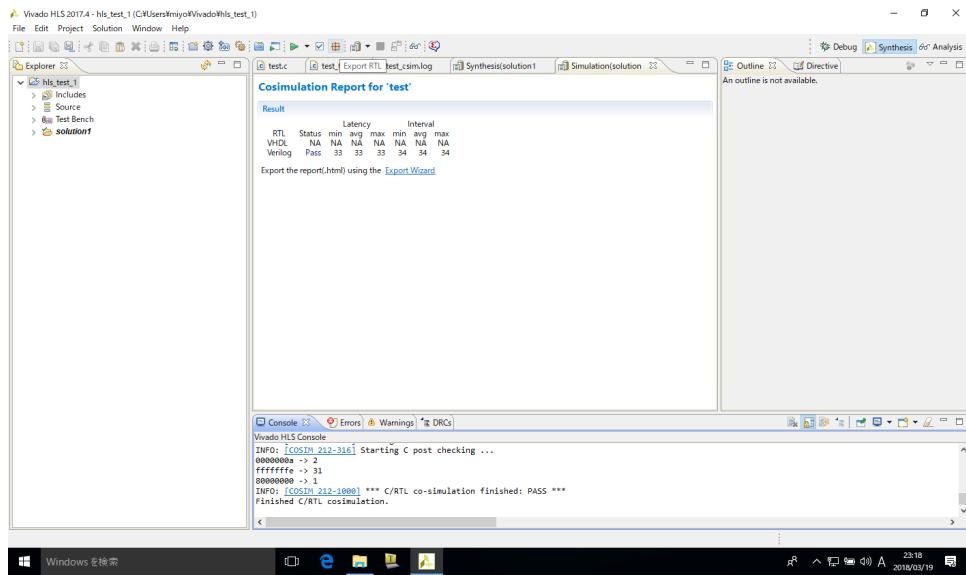


図 28 ツールバーの荷物のようなアイコンをクリックして IP コアを生成する

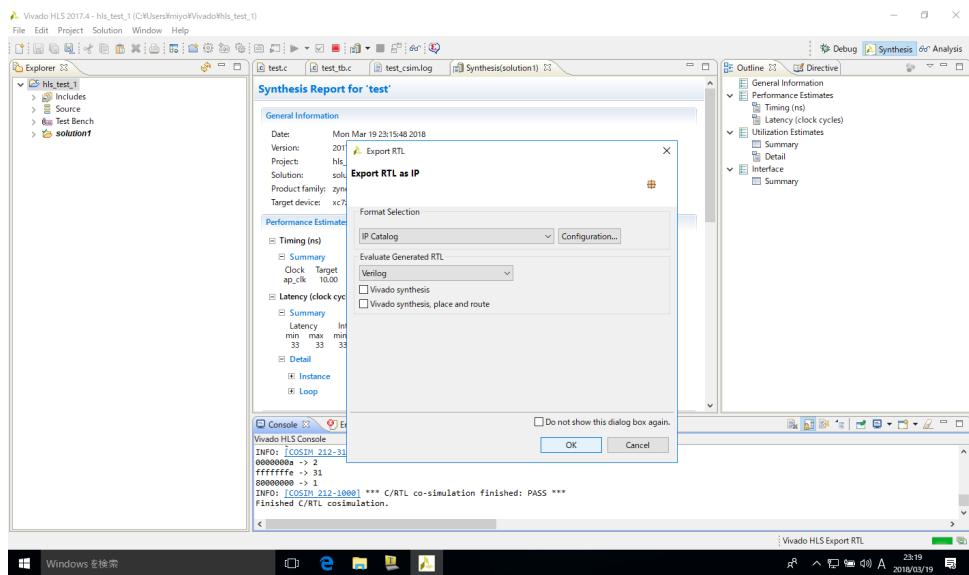


図 29 IP コア生成に関する設定ダイアログが開いたところ。特に変更の必要はないので OK をクリック

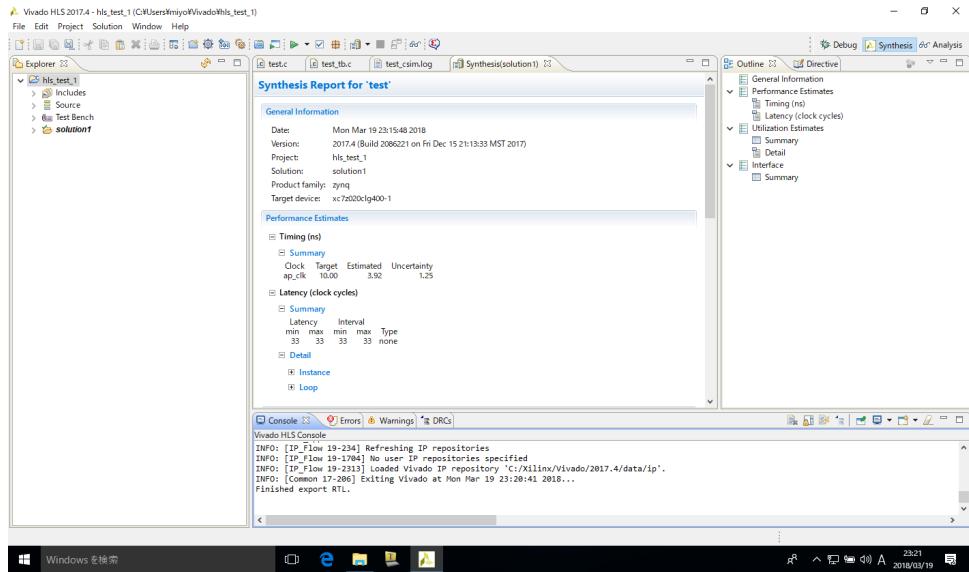


図 30 しばらく待つと、Vivado で利用可能な IP コアが生成される。

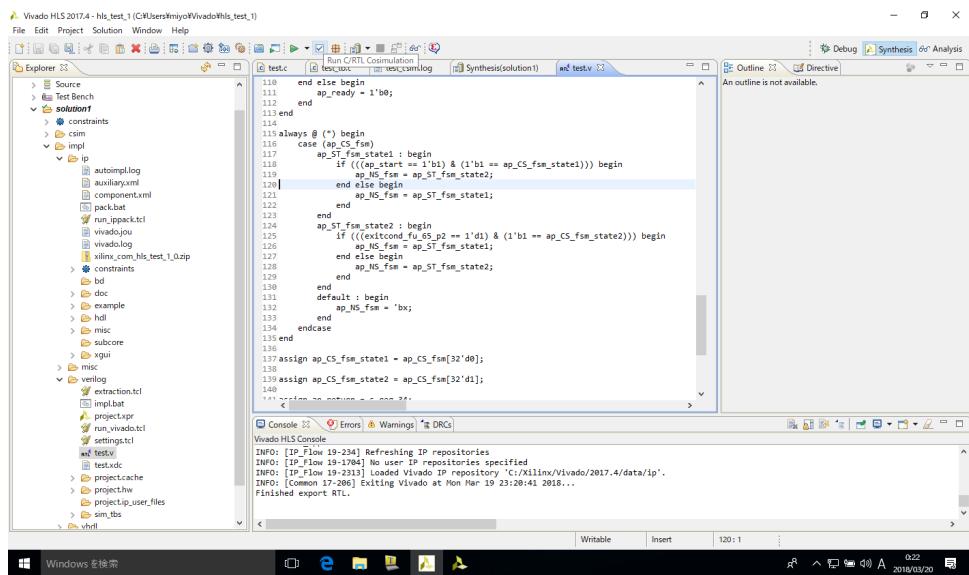


図 31 生成されたリソースやファイルは solution1 の下の impl の下に格納されていて自由に確認することができる

2.5 生成したモジュールを FPGA プロジェクトで利用する

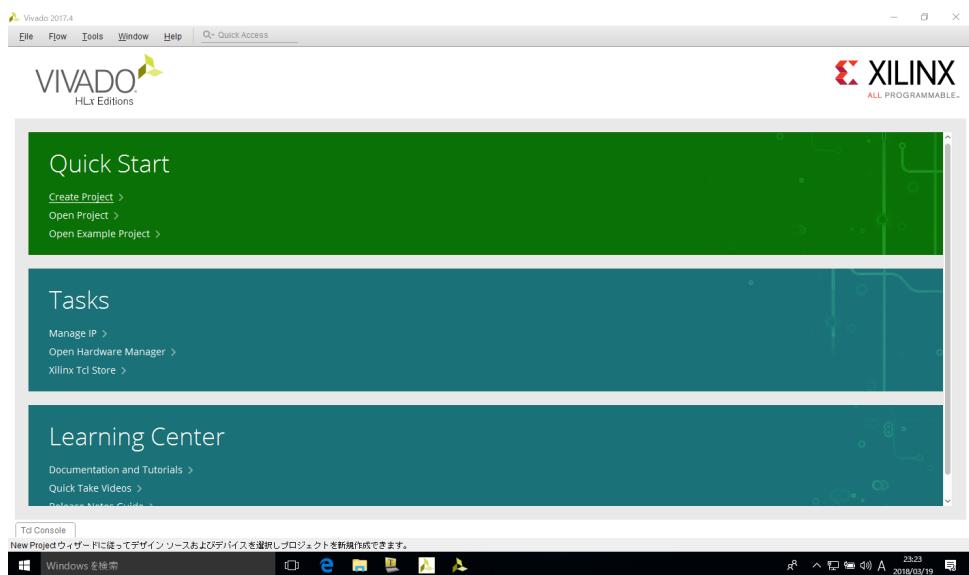


図 32 まずは Vivado プロジェクトを作成する

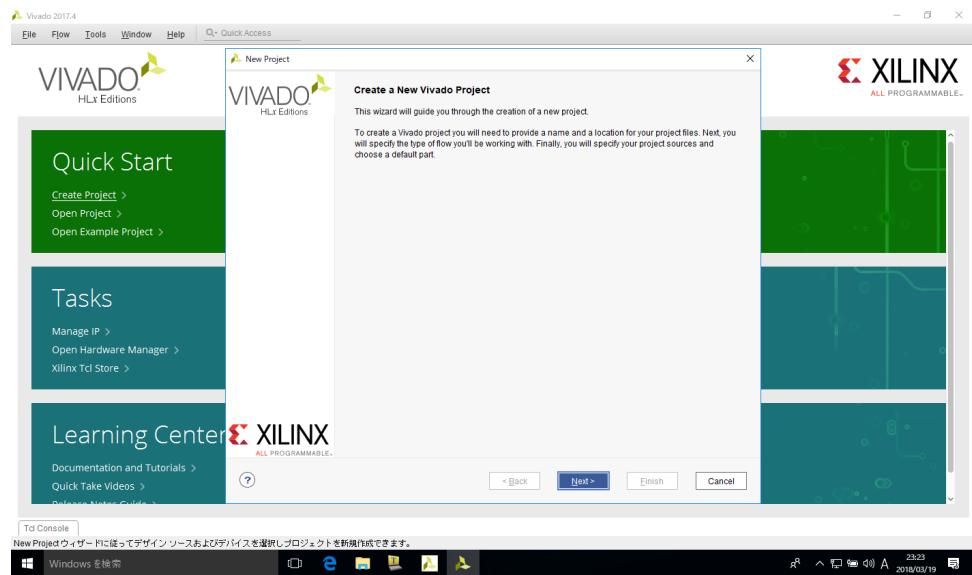


図 33 プロジェクト作成ダイアログ。Next で次へ

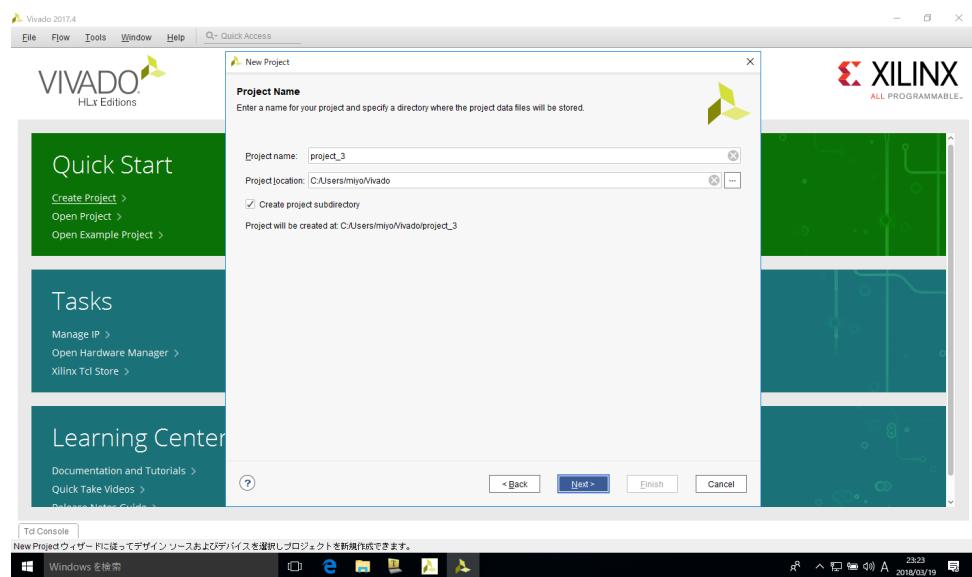


図 34 プロジェクト名を project_3 として、ホーム下の Vivado フォルダの下に保存することにする

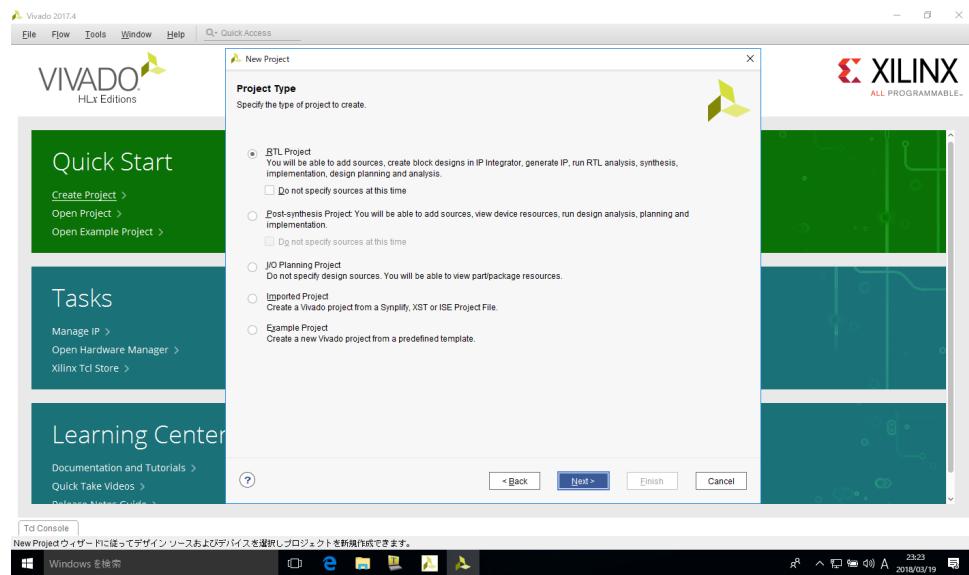


図 35 作成するプロジェクトは RTL プロジェクトを選択。

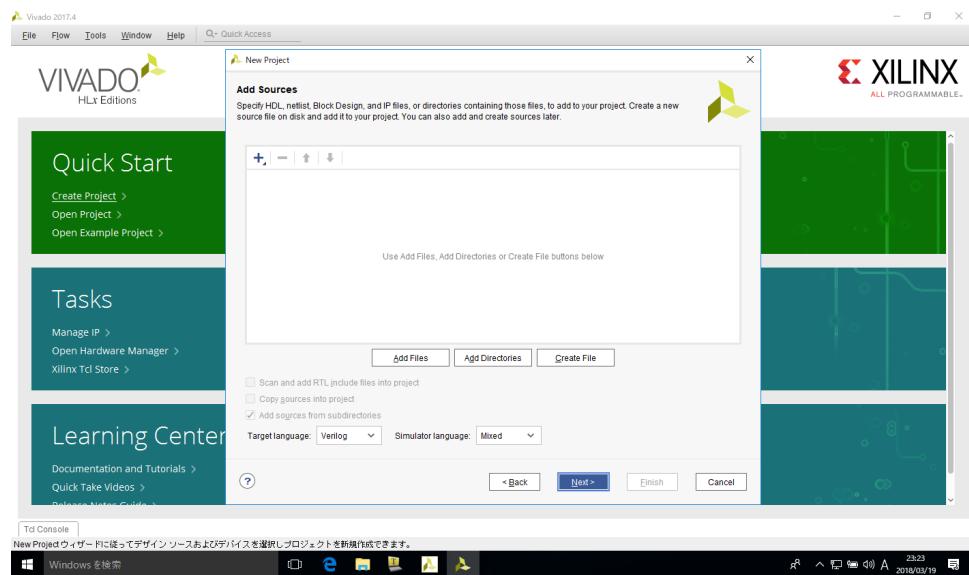


図 36 特にここで追加するファイルはないので Next で次へ

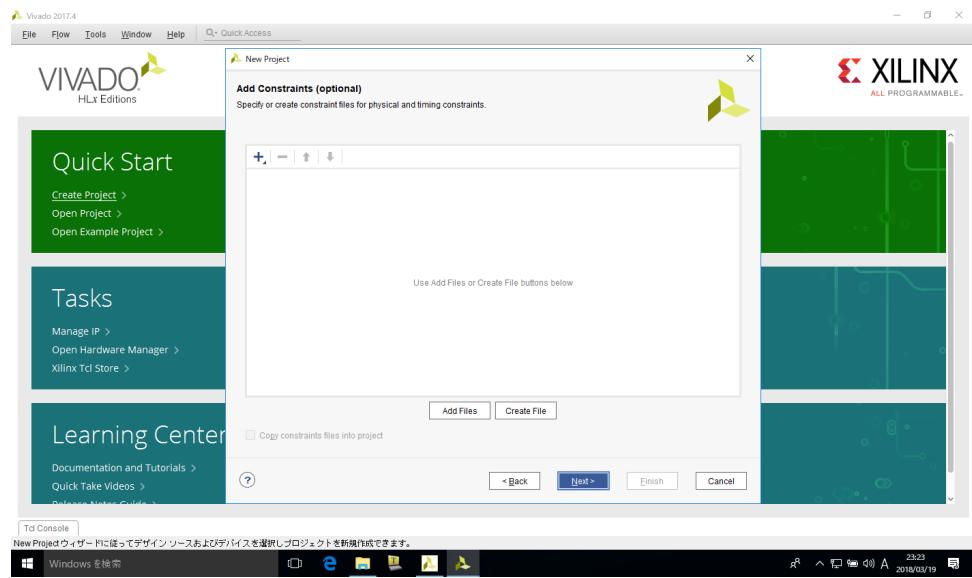


図 37 特にここで追加するファイルはないので Next で次へ

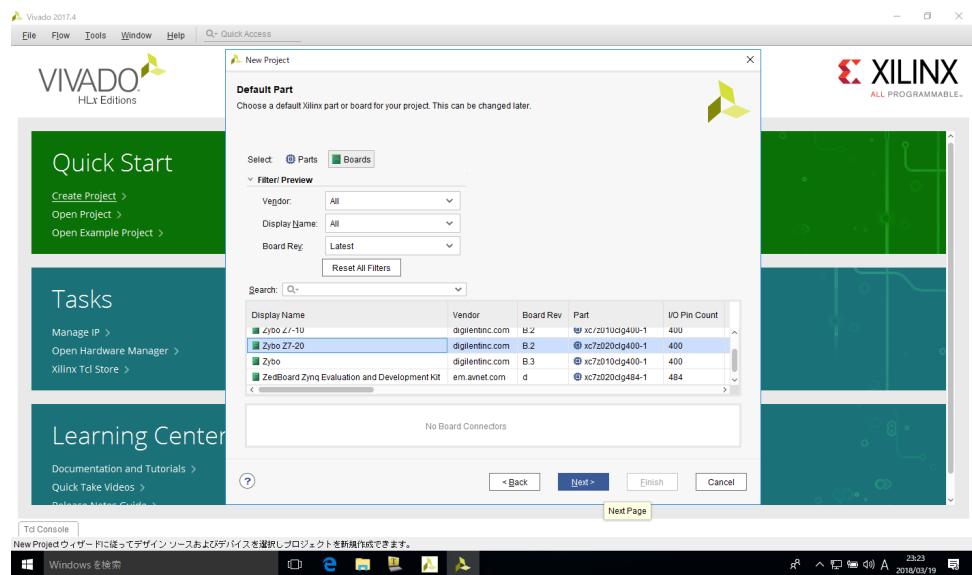


図 38 プロジェクトの開発ターゲットはボードリストから Zybo Z7-20 を選択

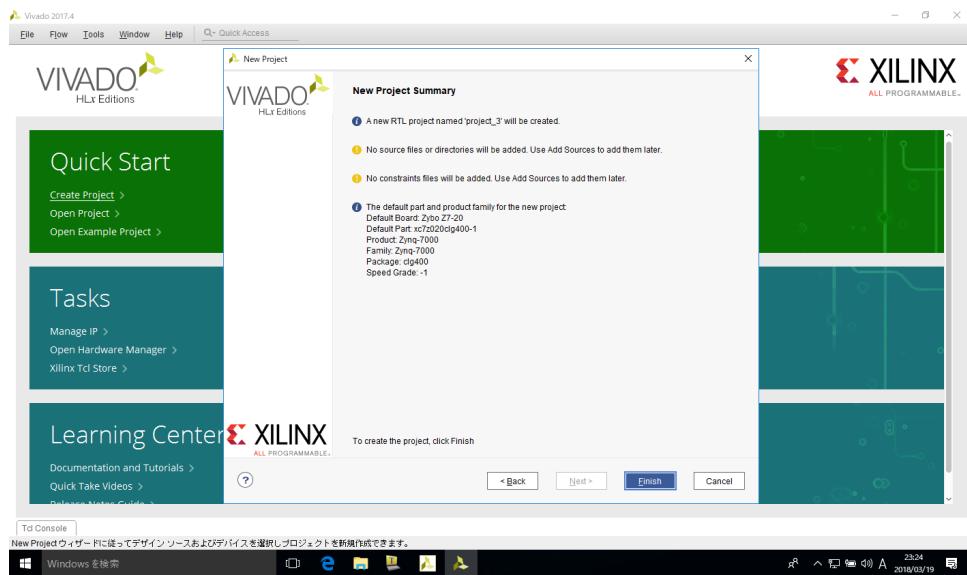


図 39 設定内容を確認して Finish をクリック。ウィザードを終了する。

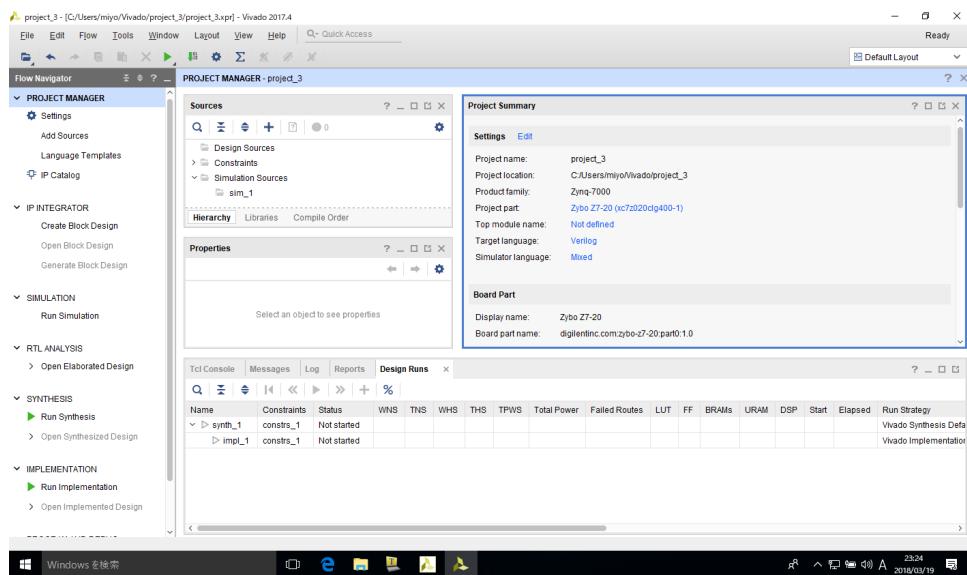


図 40 PROJECT NAVIGATOR の Settings をクリックして設定画面を呼び出す

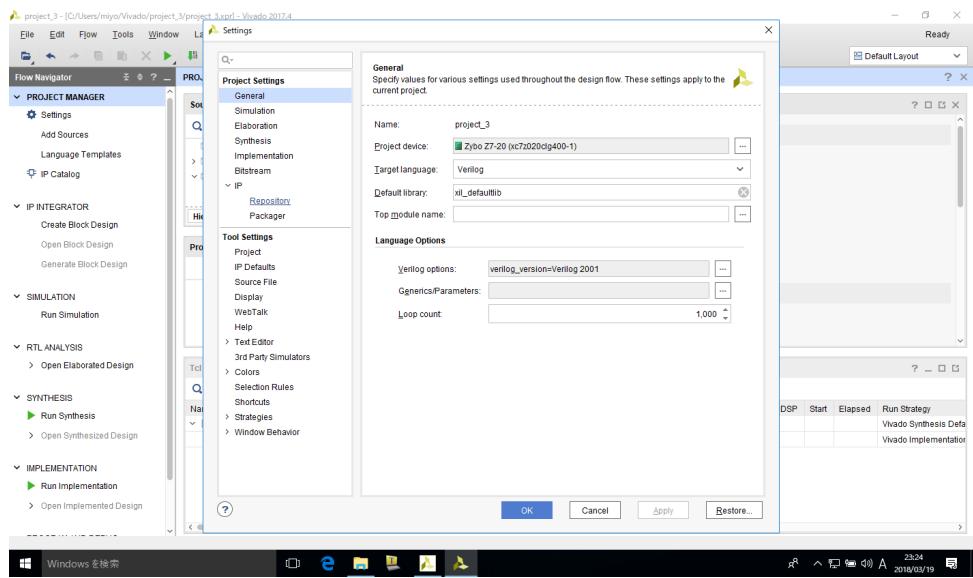


図 41 リストの IP を展開し、Repositories を選択する

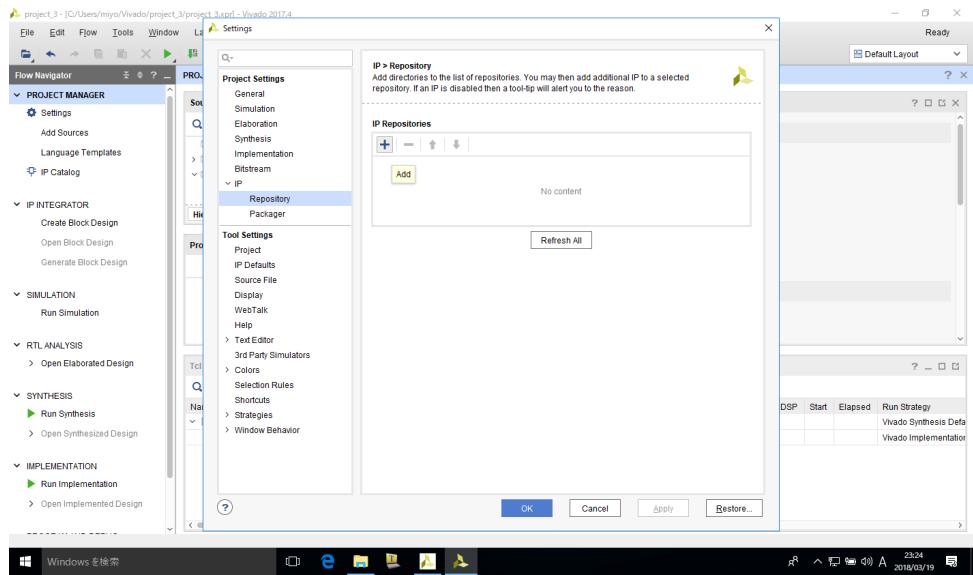


図 42 ここに Vivado HLS で作成した IP コアのフォルダを指定すれば利用できるようになる。+ アイコンをクリック。

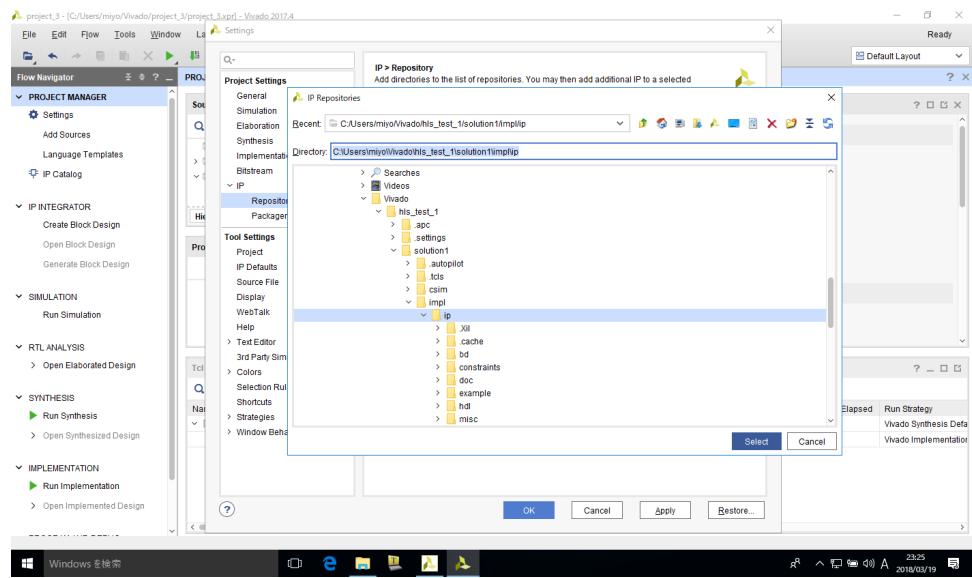


図 43 IP コア検索用のフォルダとして Vivado HLS で作成したプロジェクトフォルダ (hls_test_1) の下の, solution1\ip を選択して、Select をクリック

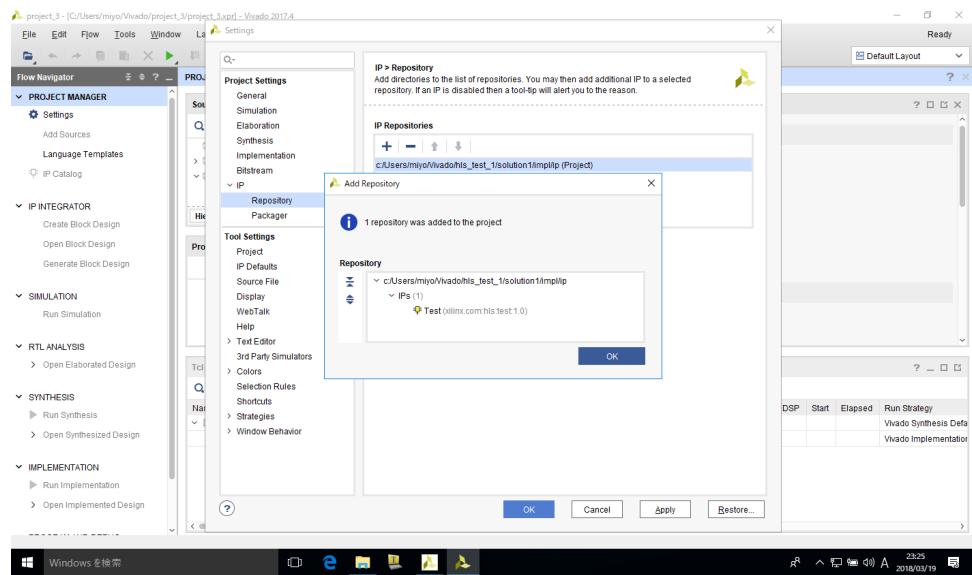


図 44 追加したリポジトリ情報が表示される。Test という IP コアが含まれていれば設定は正しい。OK でダイアログを閉じる

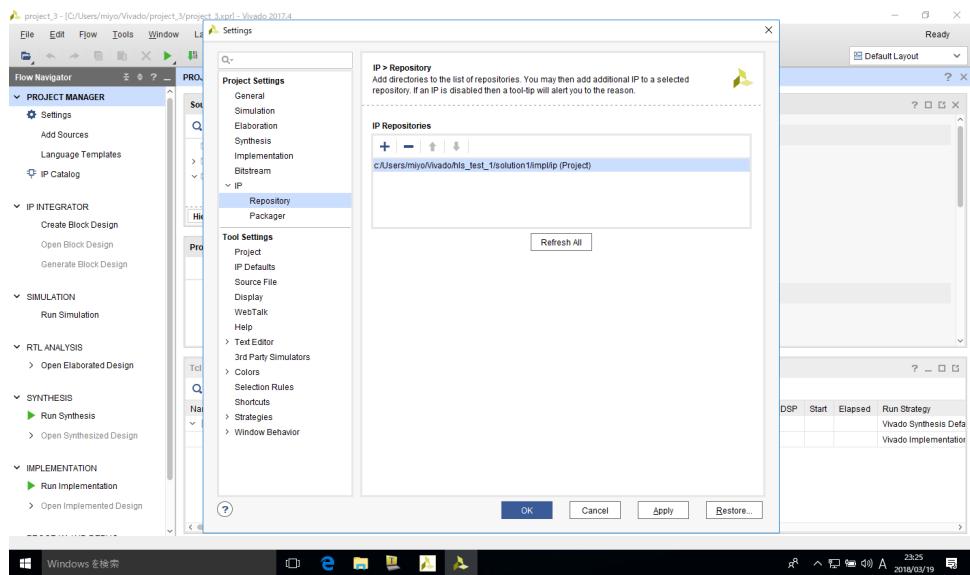


図 45 リポジトリに追加できたので、OK で設定ダイアログを閉じる

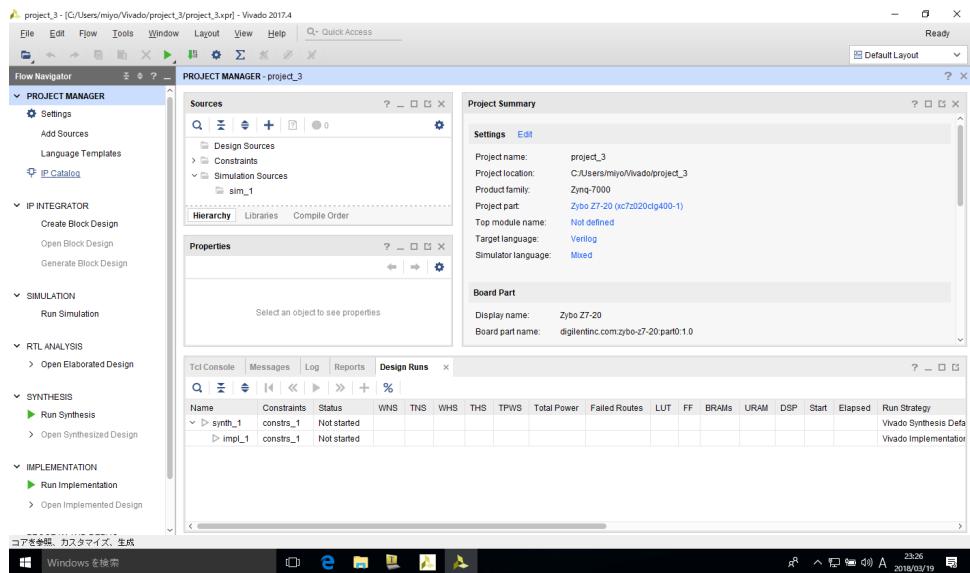


図 46 IP Catalogs を選択して IP を呼び出す

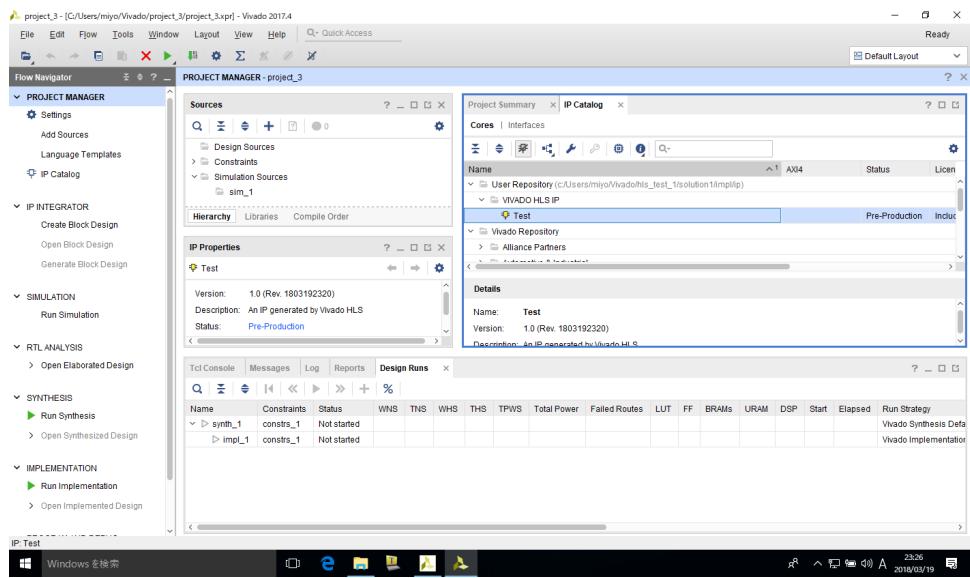


図 47 IP Catalog の中に Vivado HLS で作成した Test があるので、ダブルクリックする

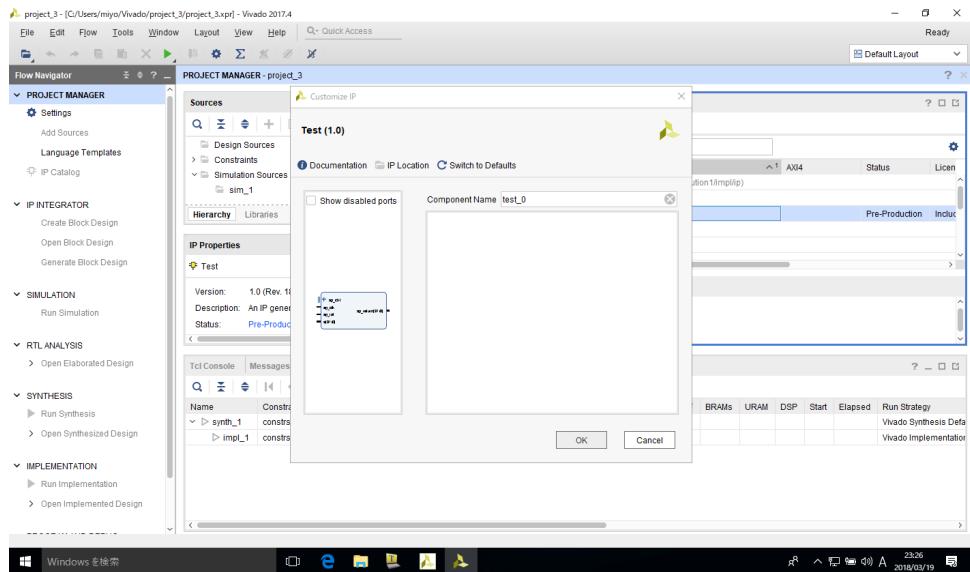


図 48 IP コアの設定画面が開くが、することもない。OK を押してダイアログを閉じる

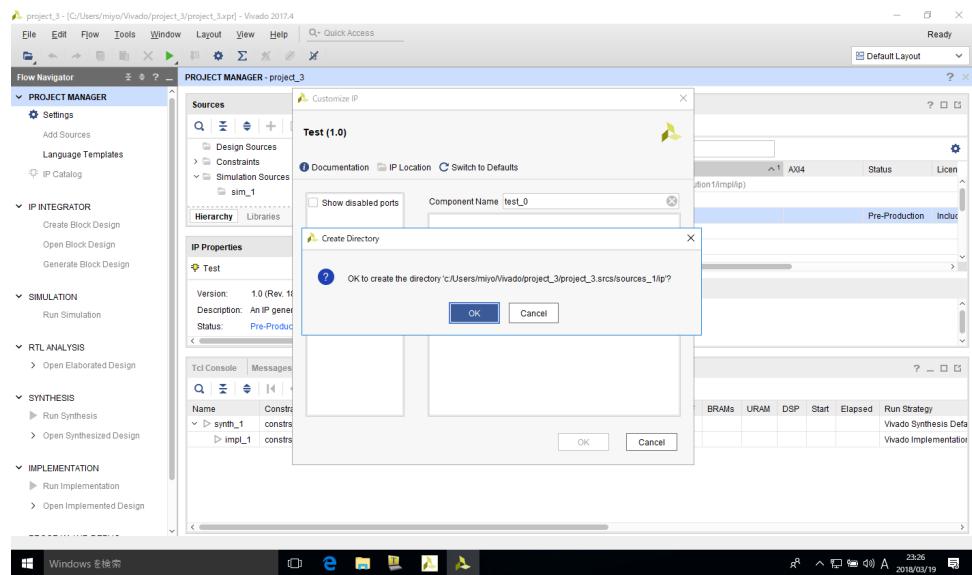


図 49 Vivado プロジェクト内に IP コア保存用のフォルダを作る確認を求める。OK をクリック

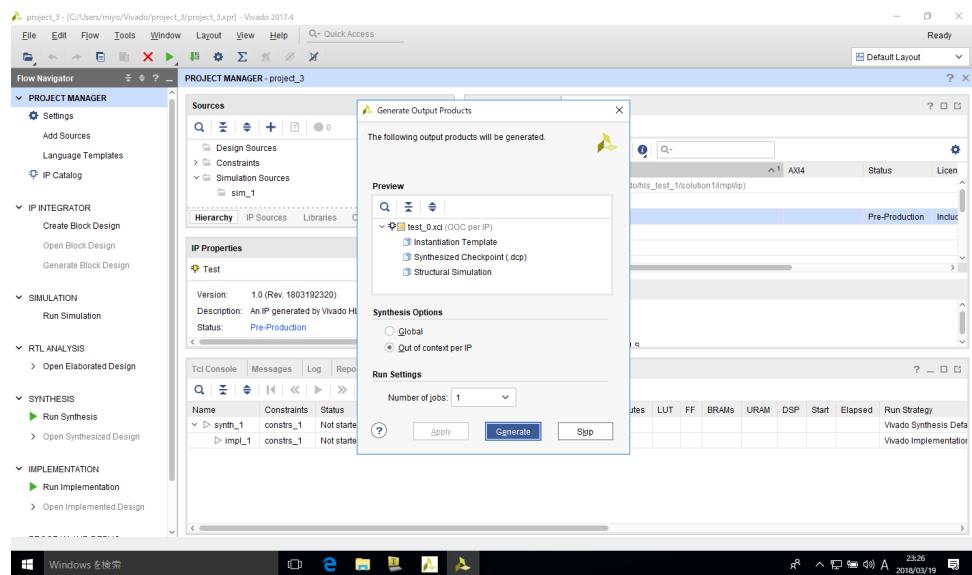


図 50 IP コア関連のファイルを Vivado に生成させるため Generate をクリック

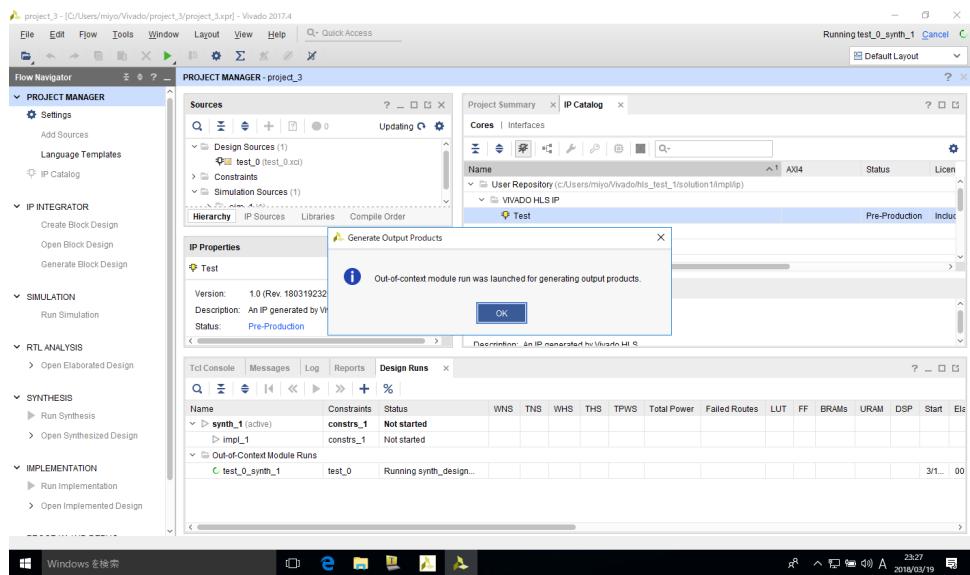


図 51 合成がかかる旨のメッセージが表示されたら OK で閉じる

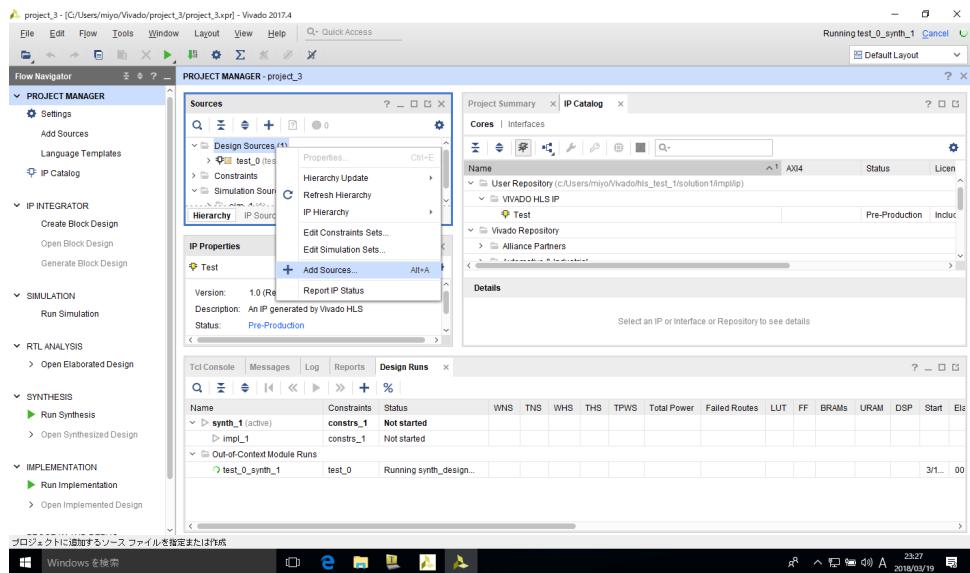


図 52 呼びだした IP コアのインスタンスを持つためのトップモジュールを作成する。モジュールの作成は、Design Sources の上で右クリックして Add Sources... 選択すればよい

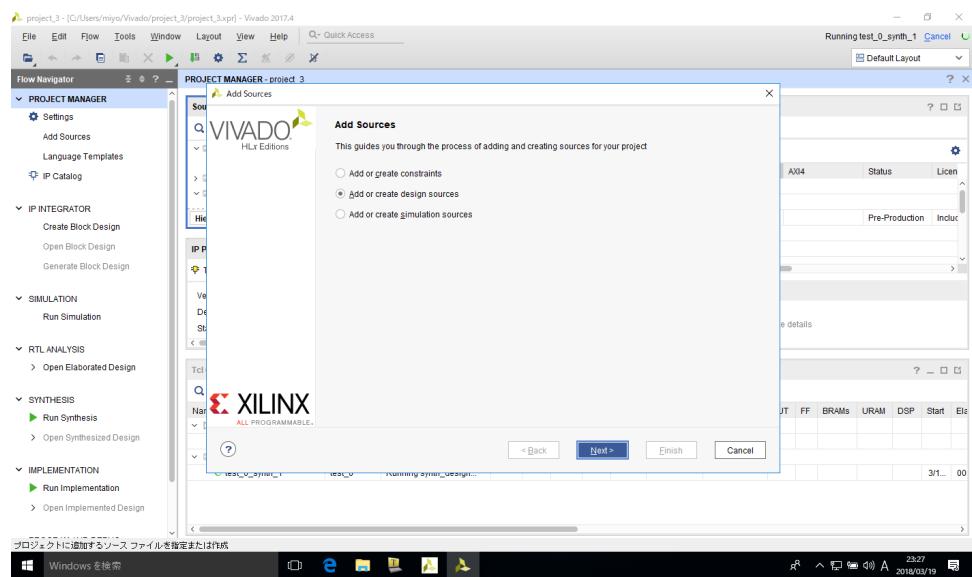


図 53 今回はデザインファイルを作る所以、Add or create design sources を選択

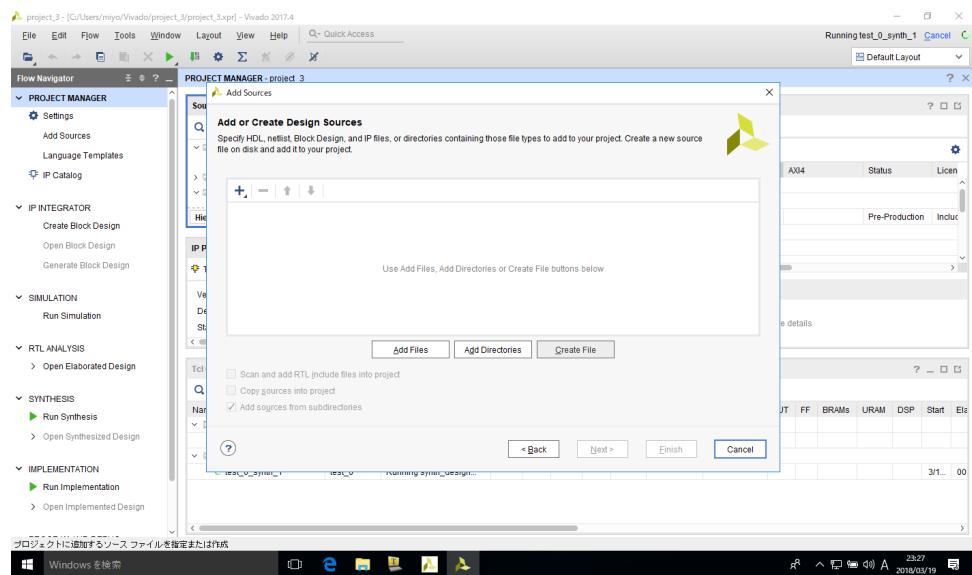


図 54 Create File をクリック。

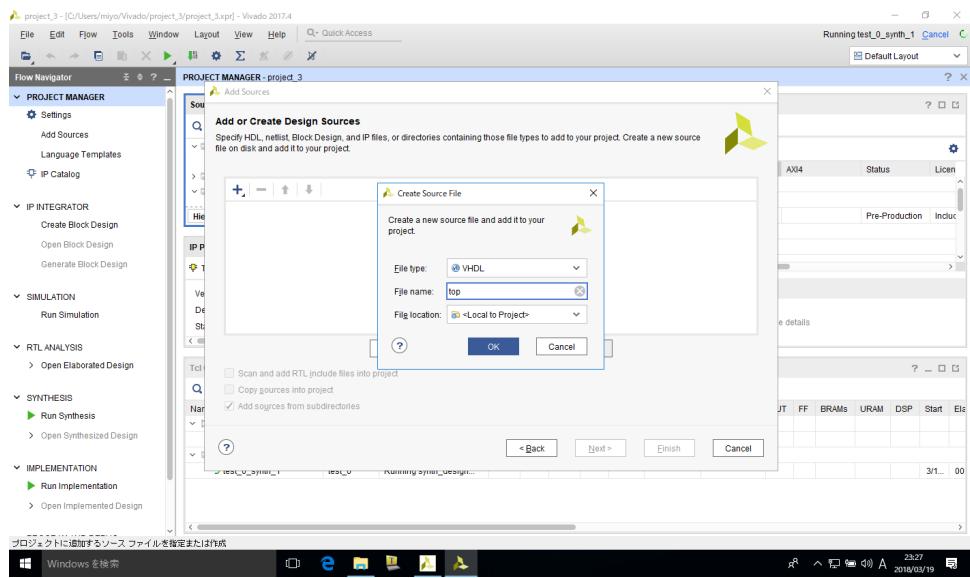


図 55 VHDL で、top という名前のモジュールを作ることにして OK をクリック。

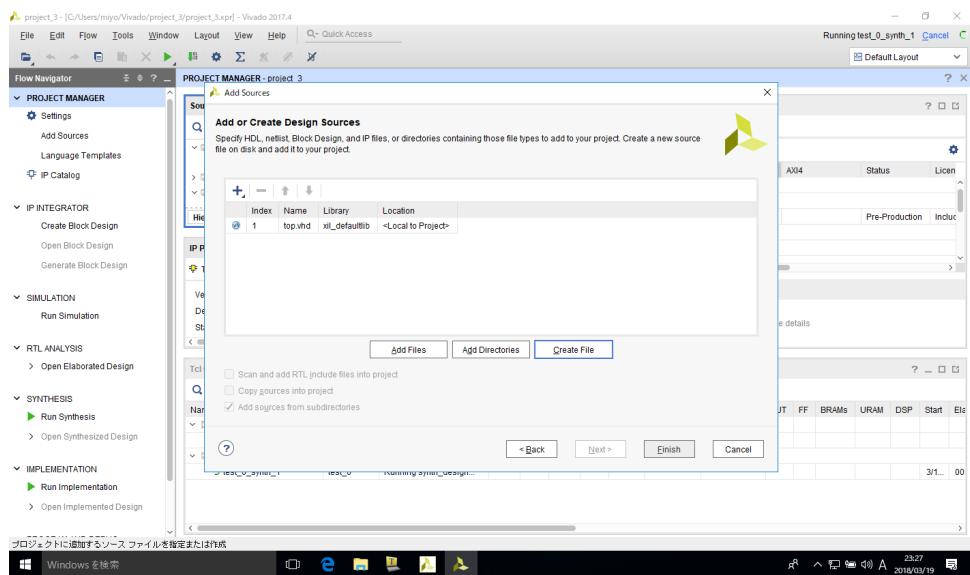


図 56 作成したモジュールがリストに追加されたので Finish で終了。

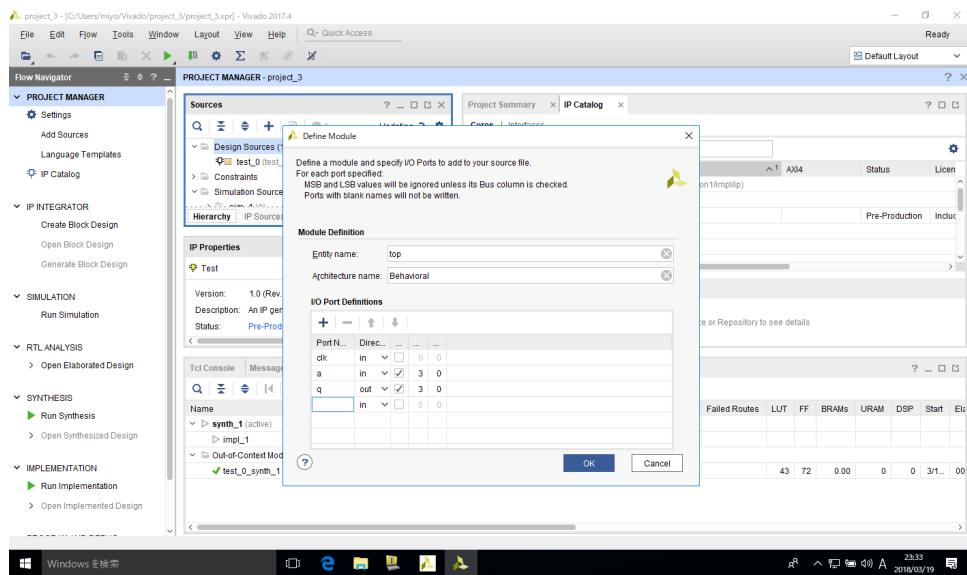


図 57 top モジュールの入出力ポートの生成ウィザードが開くので、clk と入出力用のポートを定義して OK をクリックする（あとでテキストで記述してもよいので無視して OK をクリックしても構わない）。

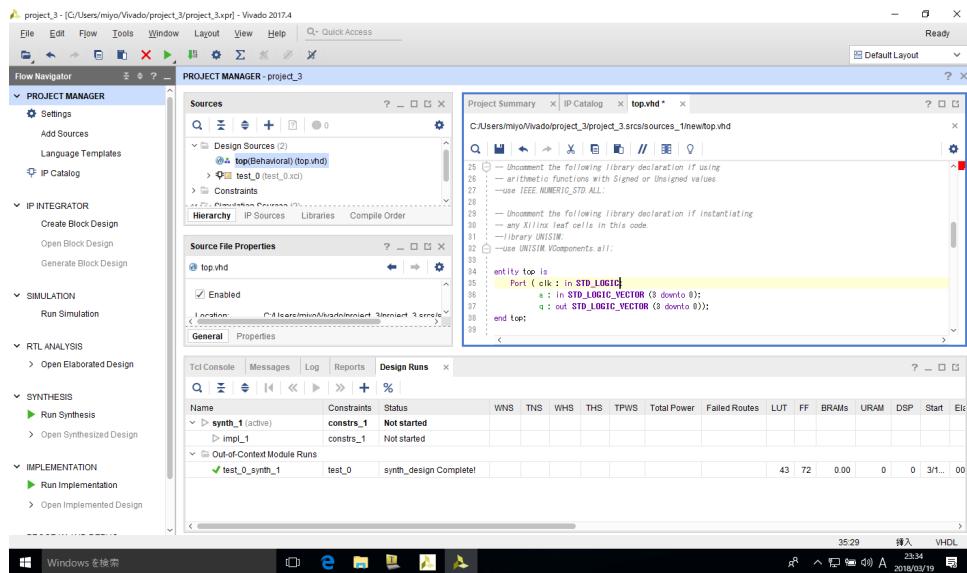


図 58 生成したトップモジュールを Vivado のエディタで開いたところ

top.vhd の内容は次の内容で書きかえる。

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity top is
5     Port ( clk : in STD_LOGIC;
6             a : in STD_LOGIC_VECTOR (3 downto 0);
7             q : out STD_LOGIC_VECTOR (3 downto 0));
8 end top;
9
10
11 architecture Behavioral of top is
12
13     component test_0
14         port (
15             ap_clk : in std_logic;
16             ap_rst : in std_logic;
17             ap_start : in std_logic;
18             ap_done : out std_logic;
19             ap_idle : out std_logic;
20             ap_ready : out std_logic;
21             a : in std_logic_vector(31 downto 0);
22             ap_return : out std_logic_vector(31 downto 0)
23         );
24     end component;
25
26     attribute mark_debug : string; -- 動作確認のために mark_debug アトリビュートを使う
27
28     signal q_i : std_logic_vector(31 downto 0);
29
30 begin
31
32     q <= q_i(3 downto 0);
33
34     U : test_0
35         port map(
36             ap_clk => clk,
37             ap_rst => '0',
38             ap_start => '1',
39             ap_done => ap_done,
40             ap_idle => ap_idle,
41             ap_ready => ap_ready,
42             a(31 downto 4) => (others => '0'),
43             a(3 downto 0) => a,
44             ap_return => q_i
45         );
46
47 end Behavioral;
```

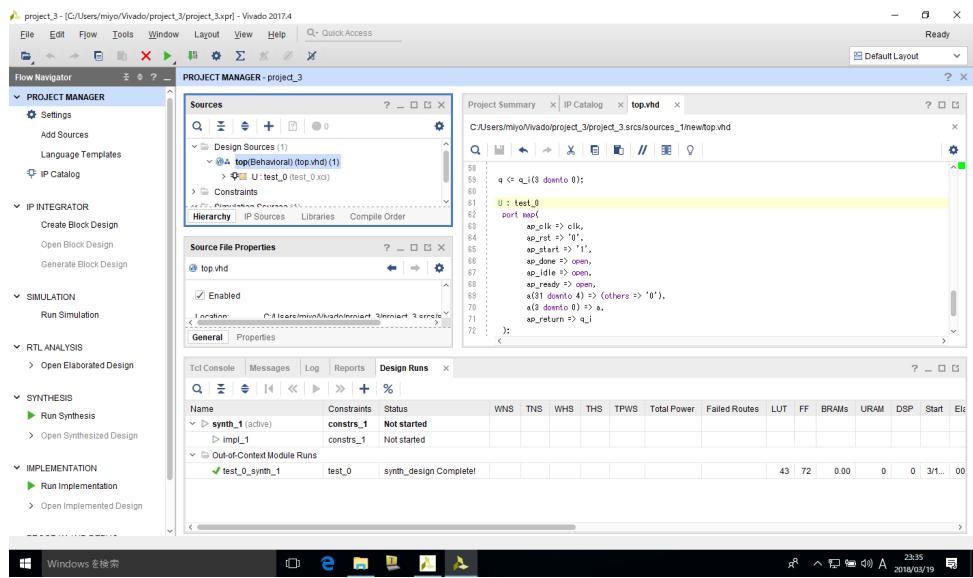


図 59 コードを正しく記述し終えると、`top → test_0` というデザインツリーが作られる

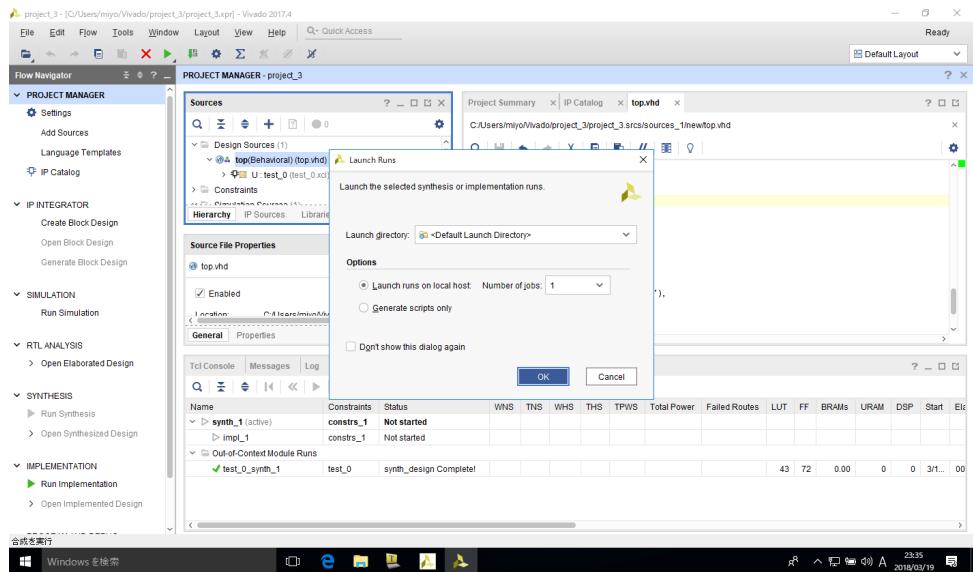


図 60 PROJECT NAVIGATOR の Run Synthesis をクリックして合成を開始。ダイアログは OK で閉じてステップを進める。

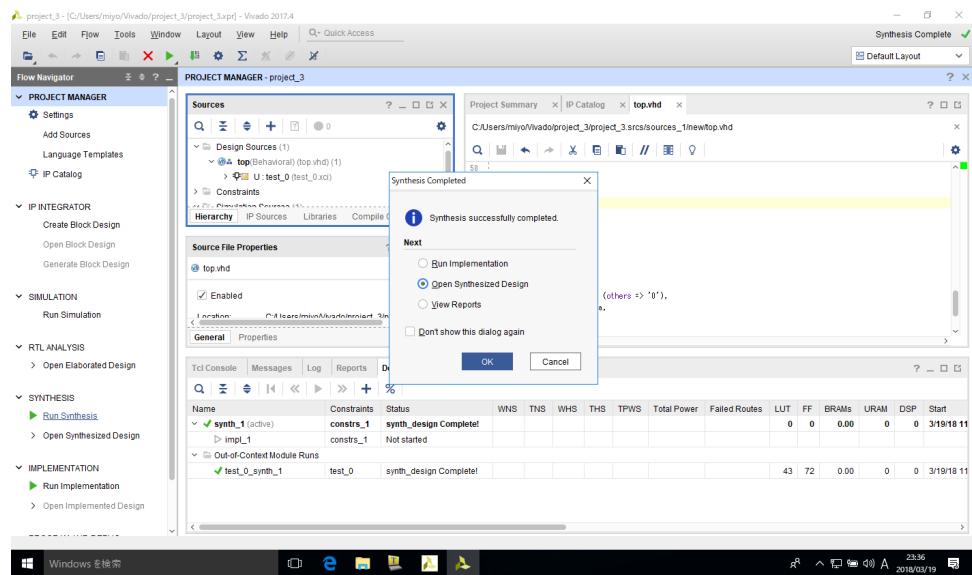


図 61 合成を終えたら、一度 Open Synthesized Design で合成結果を開く

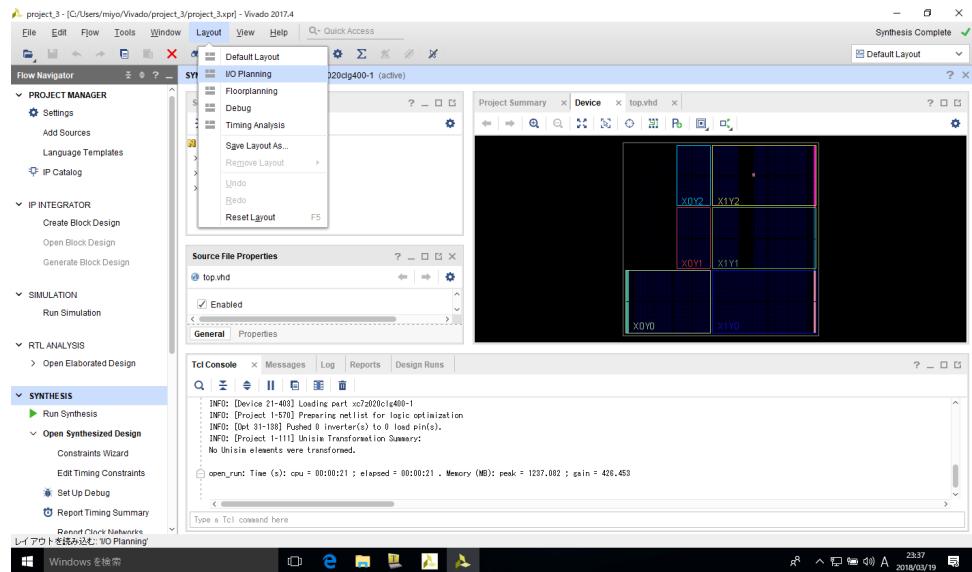


図 62 メニューの Layout から I/O Planning をクリックしてピン配置設定モードを呼び出す

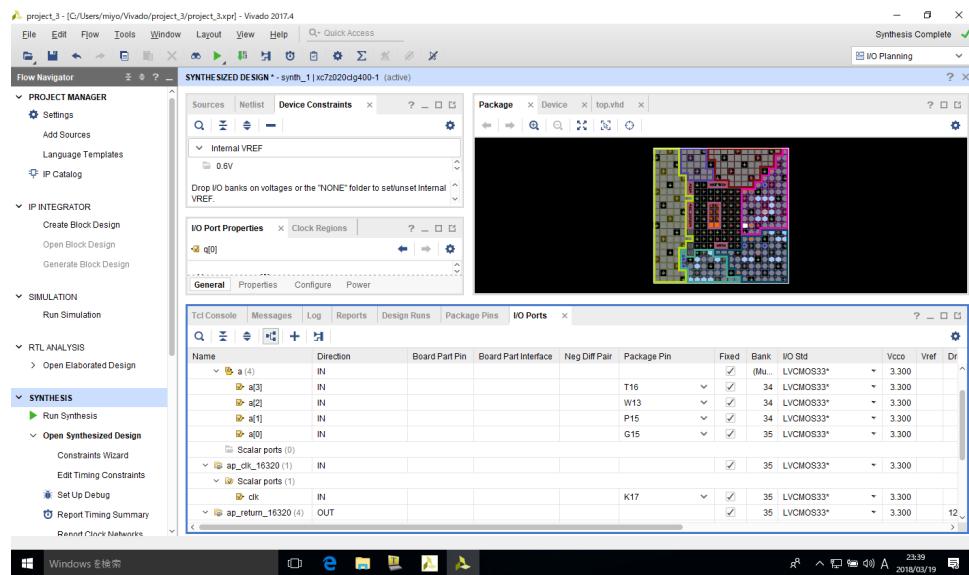


図 63 クロック (K17), 入力 4bit(T16, W13, P15, G15), 出力 4bit(D18, G14, M15, M14)を設定. 電圧は、すべて LVCMS33 に設定.

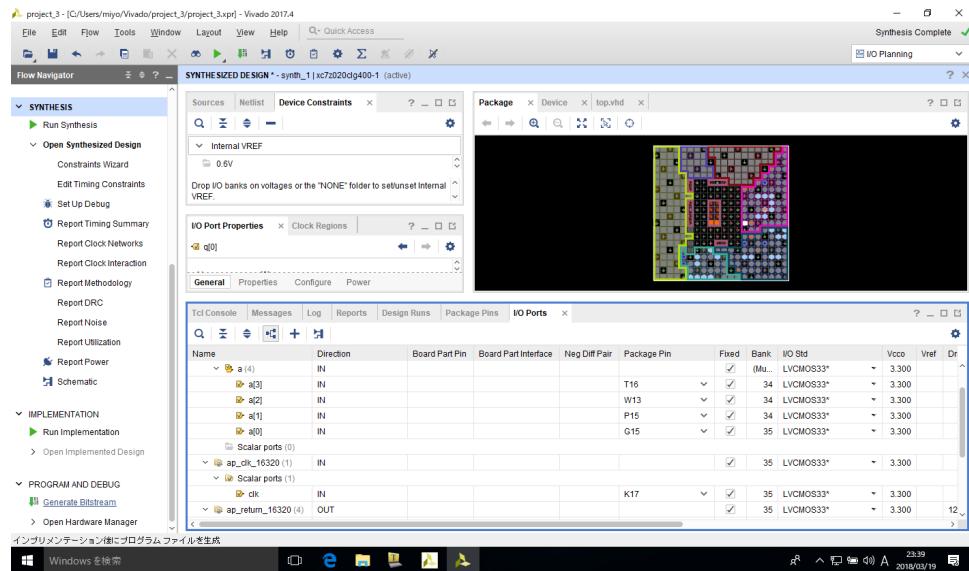


図 64 設定を終えたら Generate Bitstream をクリック

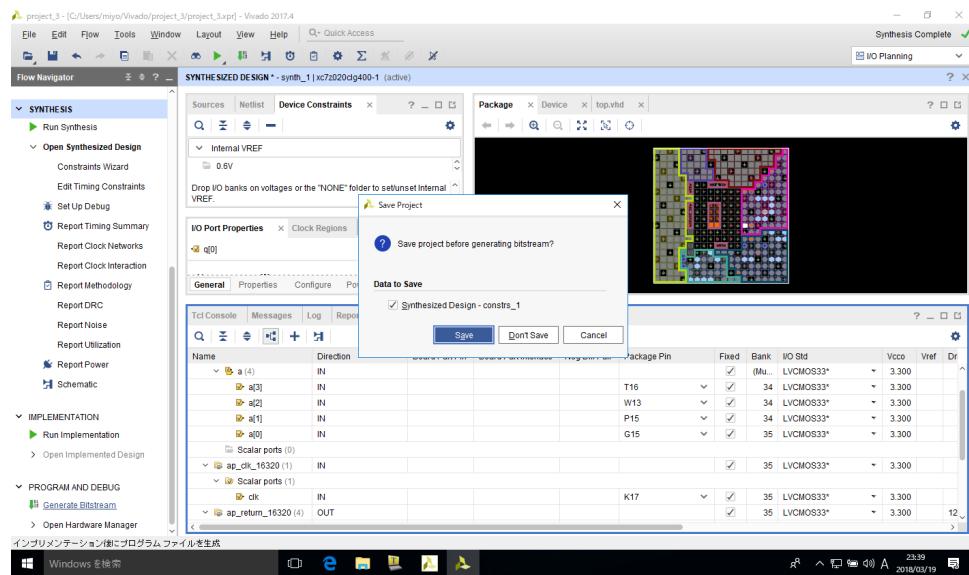


図 65 設定したピン配置を保存していいか聞いてくるので、Save で保存

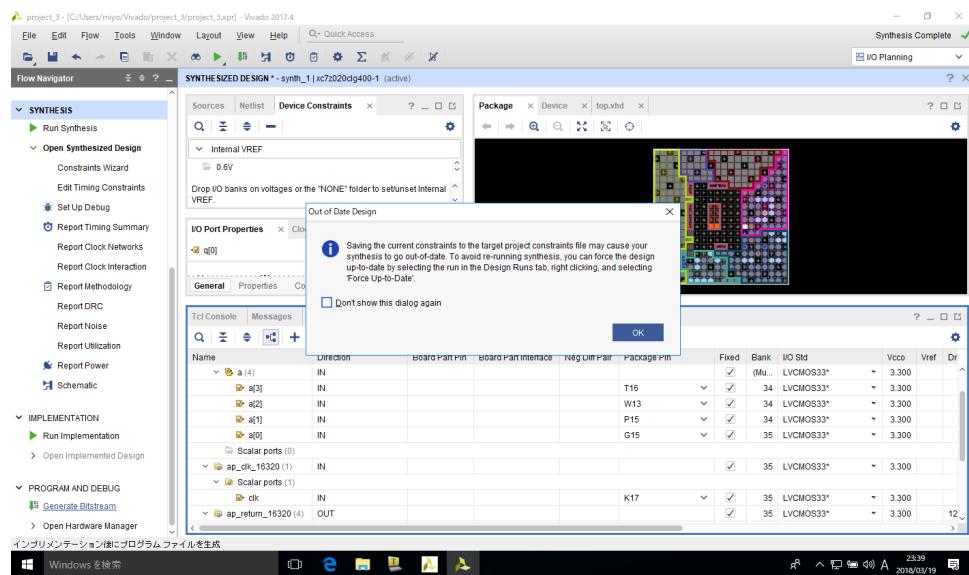


図 66 設定の保存で Synthesis が無効になる可能性があるという案内がでたら、OK でステップを進める

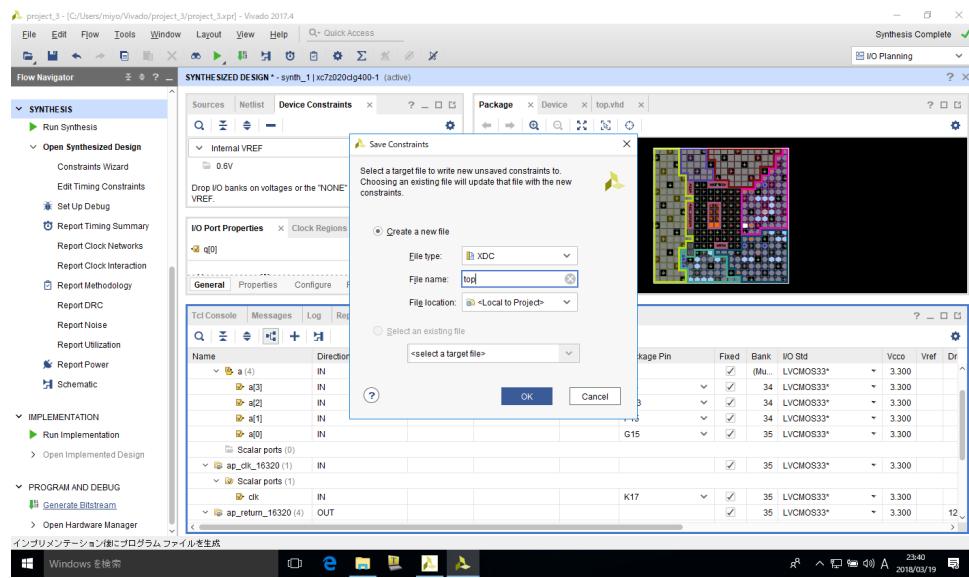


図 67 制約保存用のファイルがないので作成ダイアログに従って作成。名前を top として OK をクリックして完了

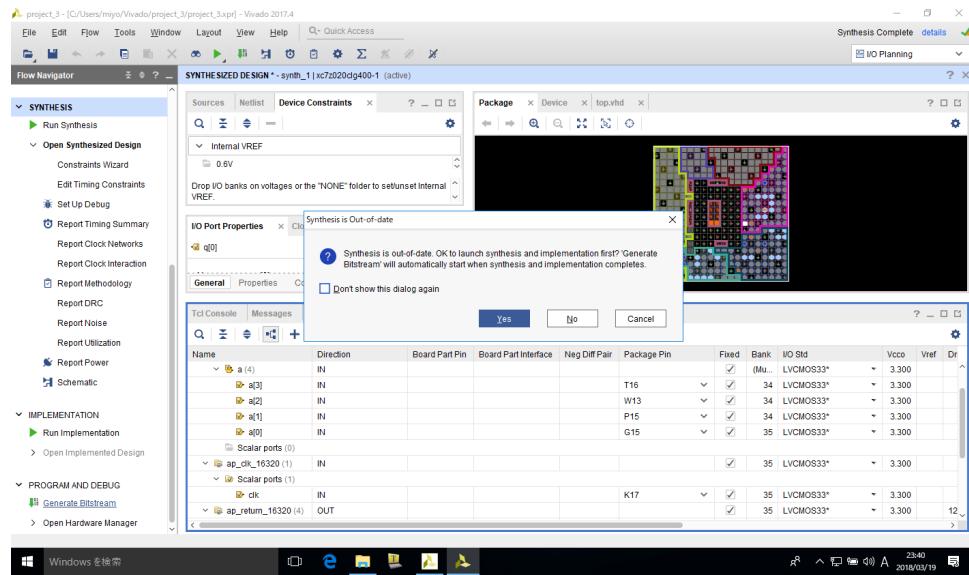


図 68 Synthesis からやりなおすことの許可を求められるので Yes でステップをすすめる

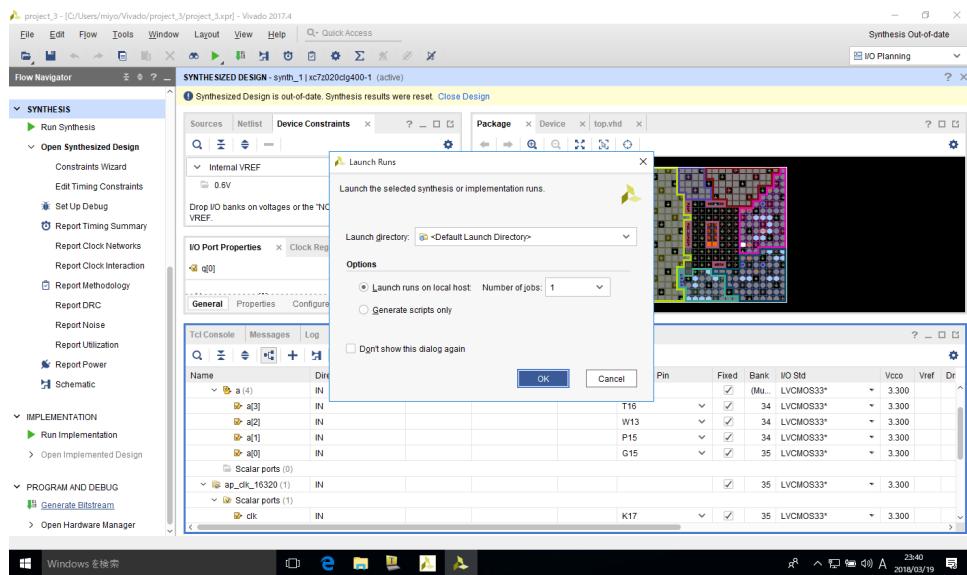


図 69 合成の開始。パラメタはそのままで OK で次へ。

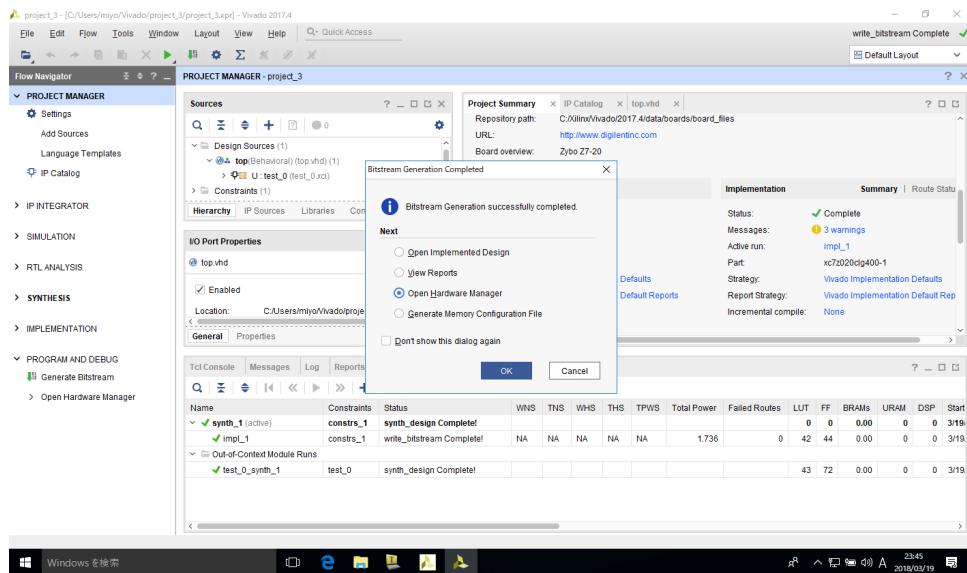


図 70 しばらく待つと合成、配置配線が完了して FPGA 用のビットストリームが生成される。Open Hardware Manager を選択して OK をクリック。

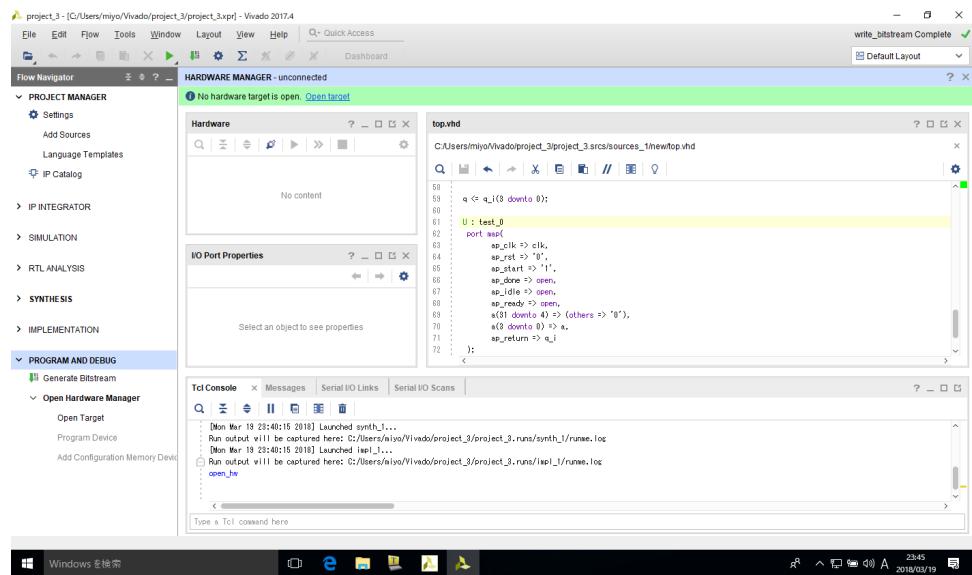


図 71 Hardware Manager が開いたところ。Open target をクリック。

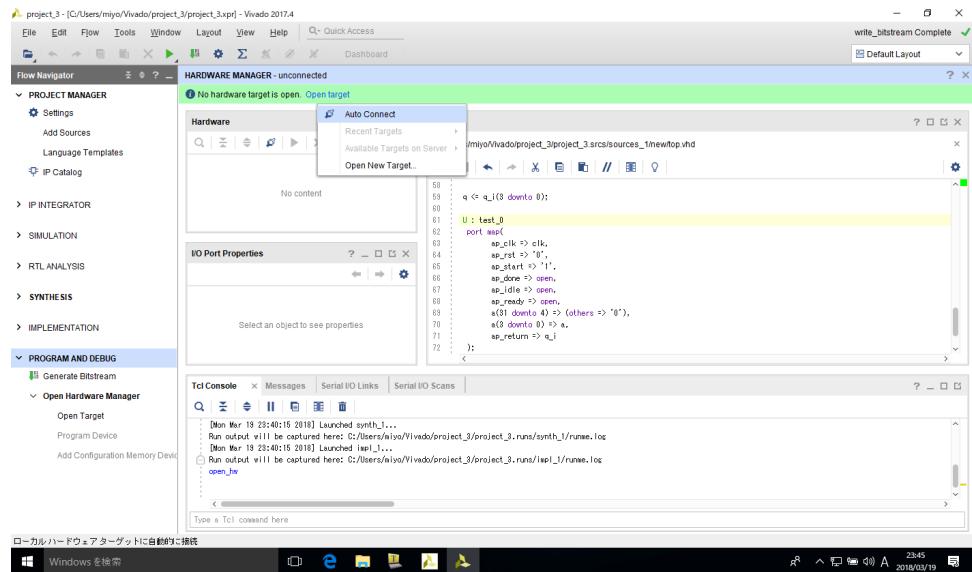


図 72 Auto Connect で FPGA ボードと接続する

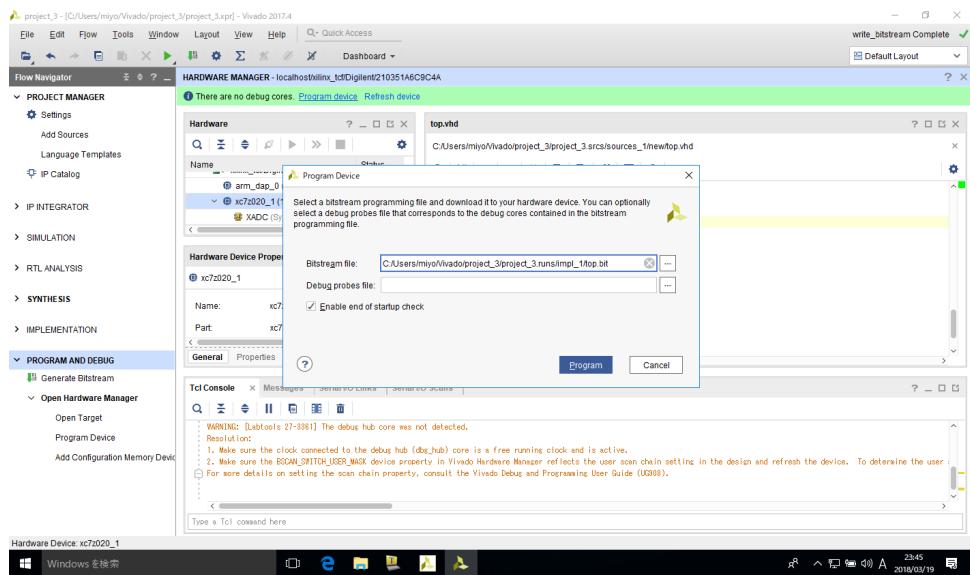


図 73 Program device で生成した bit ファイルを FPGA に書き込む

2.6 実機で動作を確認

実機での動作を確認できます。DIP スイッチを ON/OFF したときに ON の個数がカウントできていることがわかります。が、残念ながらよくみてみると、DIP スイッチを 2 つ ON にした状態、つまり 1bit 目の LED のみが点灯すべきケースで、0bit 目の LED が若干点灯していることがわかります。これはバグなので原因を探してみましょう。

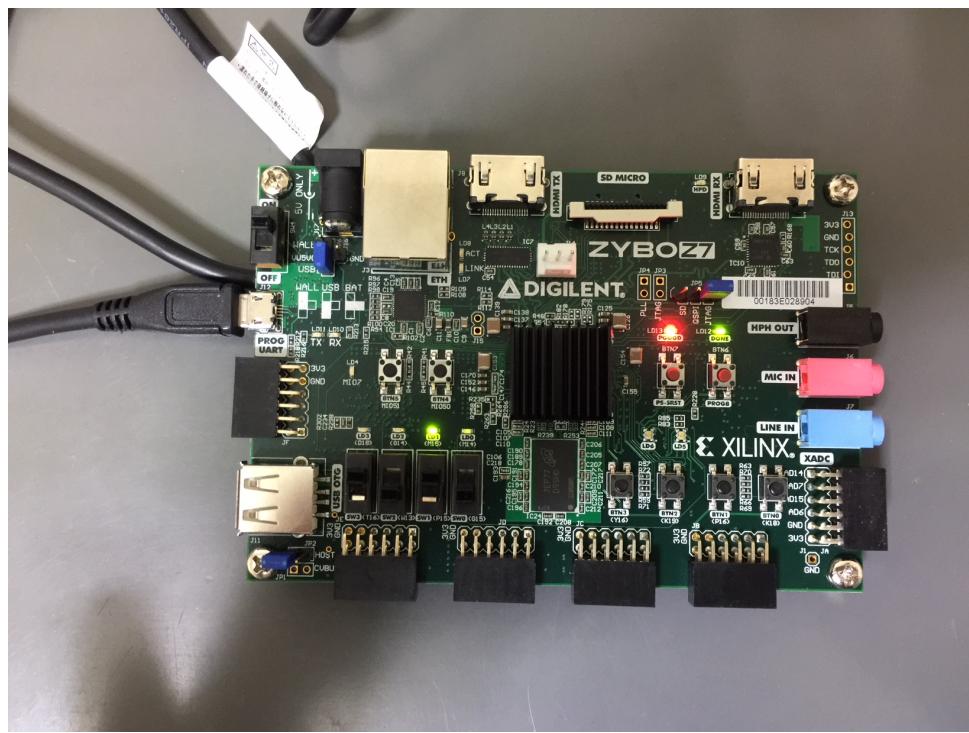


図 74 LED が一つだけ点灯するはずが 0bit 目の LED が若干明るい

ここでは、ILA を使って、実機で動作を確認してみます。まず、top モジュールを次のように書き変えて、いくつかの信号を ILA で観測できるようにしましょう。

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity top is
5     Port ( clk : in STD_LOGIC;
6             a : in STD_LOGIC_VECTOR (3 downto 0);
7             q : out STD_LOGIC_VECTOR (3 downto 0));
8 end top;
9
10
11 architecture Behavioral of top is
12
13     component test_0
14         port (
15             ap_clk : in std_logic;
16             ap_rst : in std_logic;
17             ap_start : in std_logic;
18             ap_done : out std_logic;
19             ap_idle : out std_logic;
20             ap_ready : out std_logic;
21             a : in std_logic_vector(31 downto 0);
22             ap_return : out std_logic_vector(31 downto 0)
23         );
24     end component;
25
26     attribute mark_debug : string; -- 動作確認のために mark_debug アトリビュートを使う
27
28     signal q_i : std_logic_vector(31 downto 0);
29     attribute mark_debug of q_i : signal is "true"; -- mark_debug に指定
30
31     -- ステータス信号を引出し ILA で観測するために追加
32     signal ap_done : std_logic;
33     signal ap_idle : std_logic;
34     signal ap_ready : std_logic;
35     attribute mark_debug of ap_done : signal is "true";
36     attribute mark_debug of ap_idle : signal is "true";
37     attribute mark_debug of ap_ready : signal is "true";
38     -- ステータス信号を引出し ILA で観測するために追加, ここまで
39
40 begin
41
42     q <= q_i(3 downto 0);
43
44     U : test_0
```

次のページに続く

```

1  port map(
2      ap_clk => clk,
3      ap_rst => '0',
4      ap_start => '1',
5      ap_done => ap_done,
6      ap_idle => ap_idle,
7      ap_ready => ap_ready,
8      a(31 downto 4) => (others => '0'),
9      a(3 downto 0) => a,
10     ap_return => q_i
11 );
12
13 end Behavioral;

```

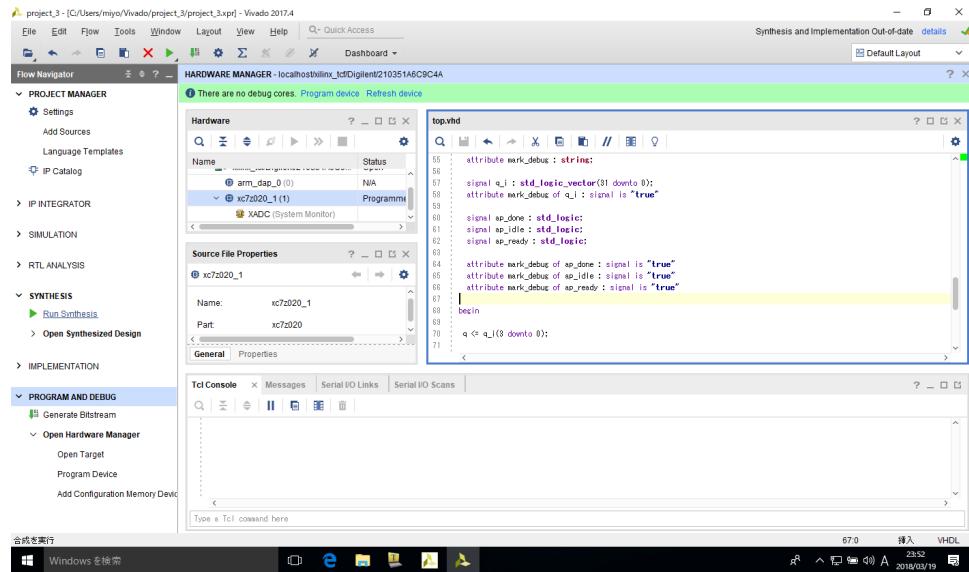


図 75 コードを書き変えたら Run Synthesis をクリックして合成を行なう

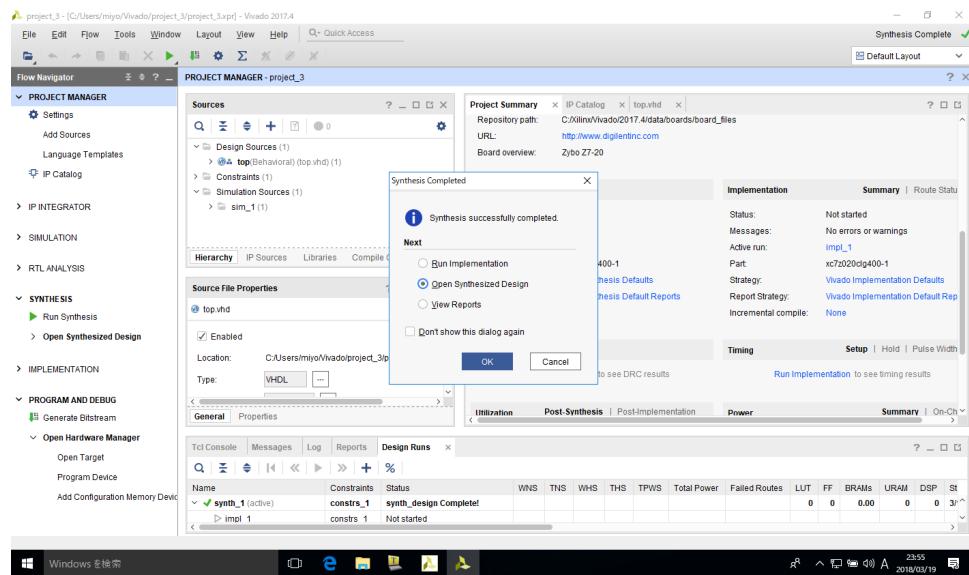


図 76 合成がおわったら Open Synthesized Design で合成結果を開く

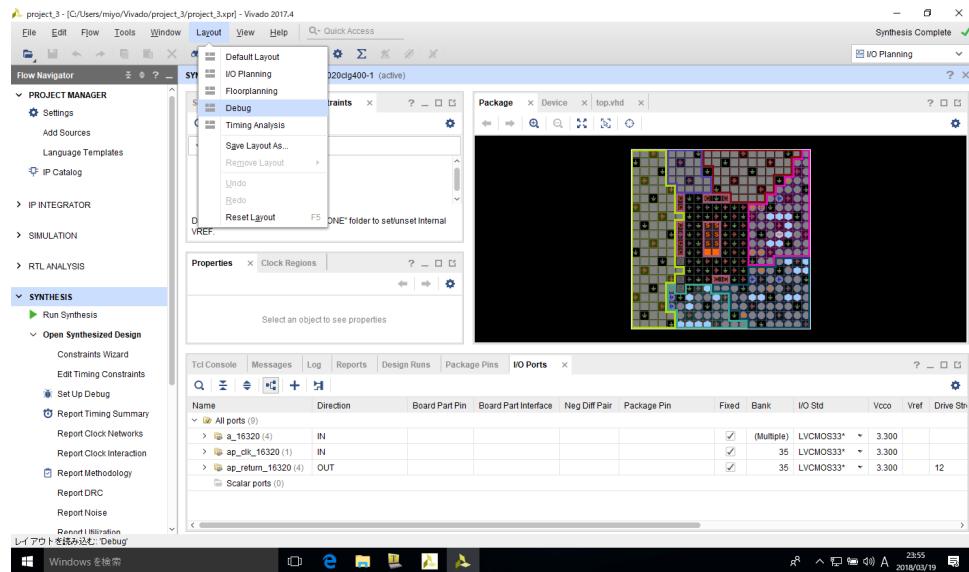


図 77 ILA の設定をしたいので、メニューの Layout から Debug を選択

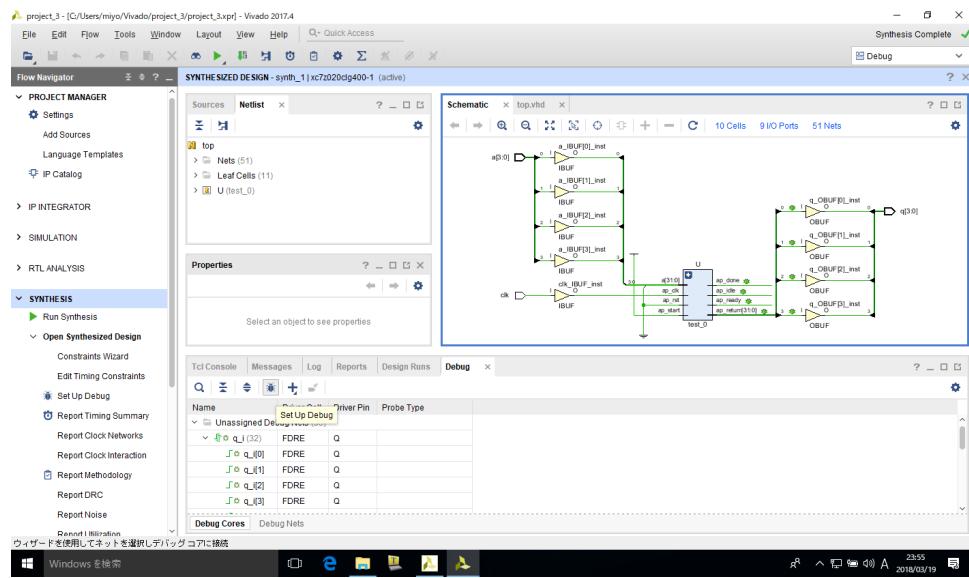


図 78 虫のアイコンをクリックして ILA 設定ウィザードを開く

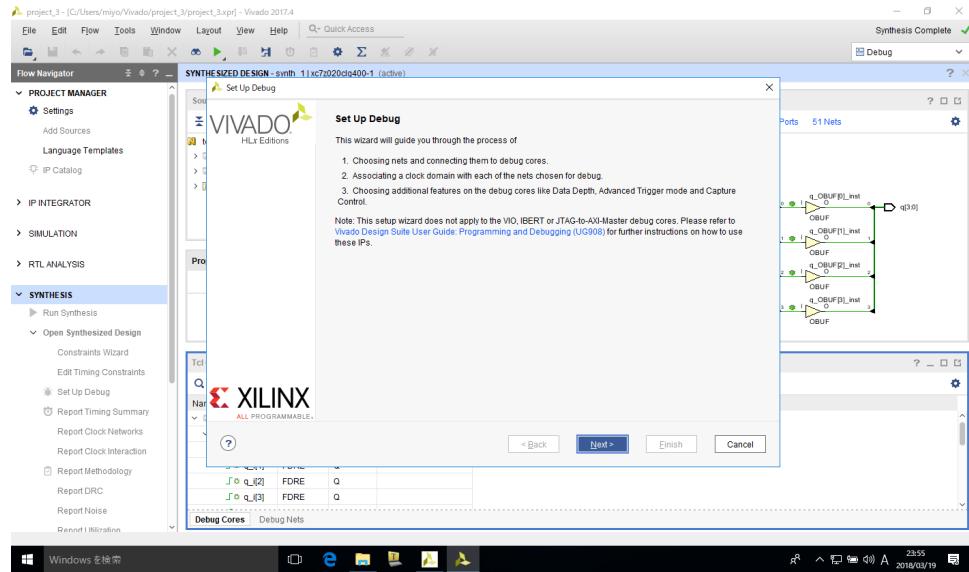


図 79 ILA 設定ウィザードの開始

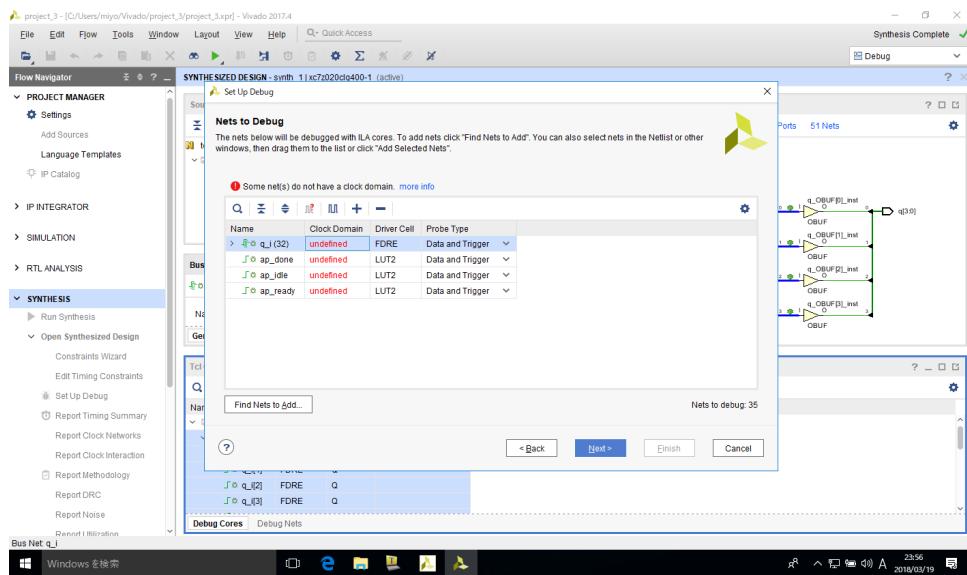


図 80 観測したい信号のクロックをみつけることができず undefined になっている

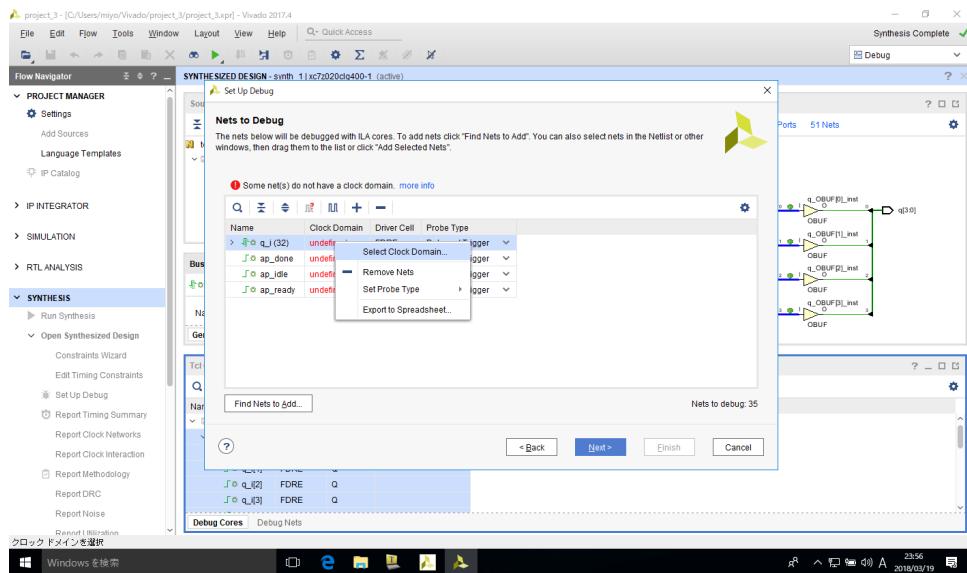


図 81 クロックを設定したいアイテムの上で右クリックして Select Clock Domain を選ぶ

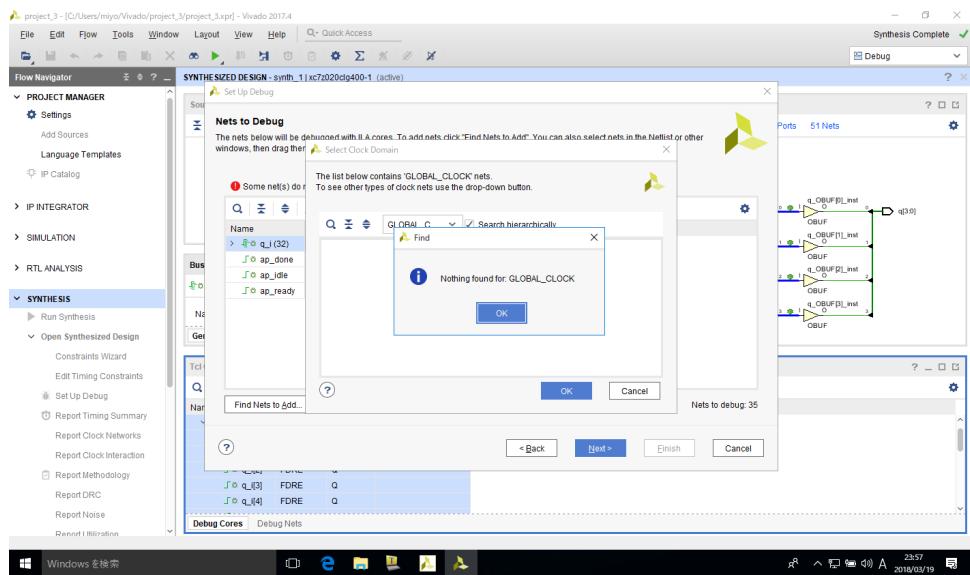


図 82 Global_CLOCKがないという案内が表示されるがOKで閉じる

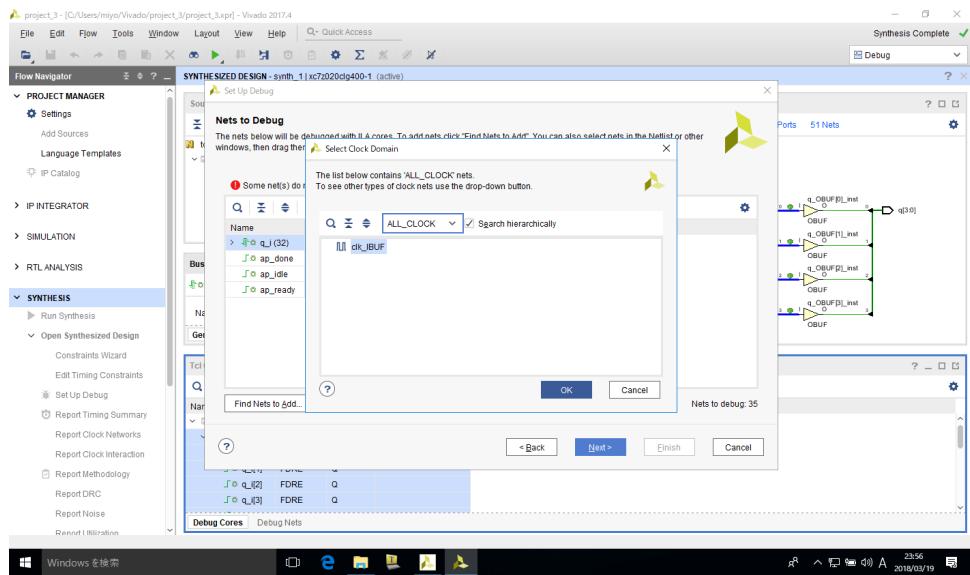


図 83 Global_CLOCKのかわりに ALL_CLOCKを選択すると clk_IBUFが見つかるので選択、OKをクリックする

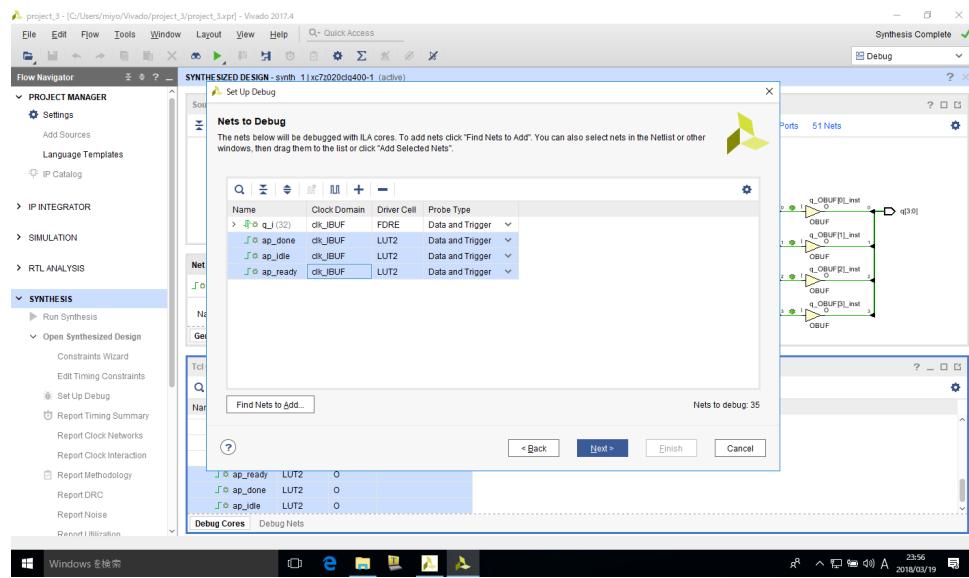


図 84 同様の手順で全ての信号の Clock Domain を clk_IBUF に設定したら Next で次へ

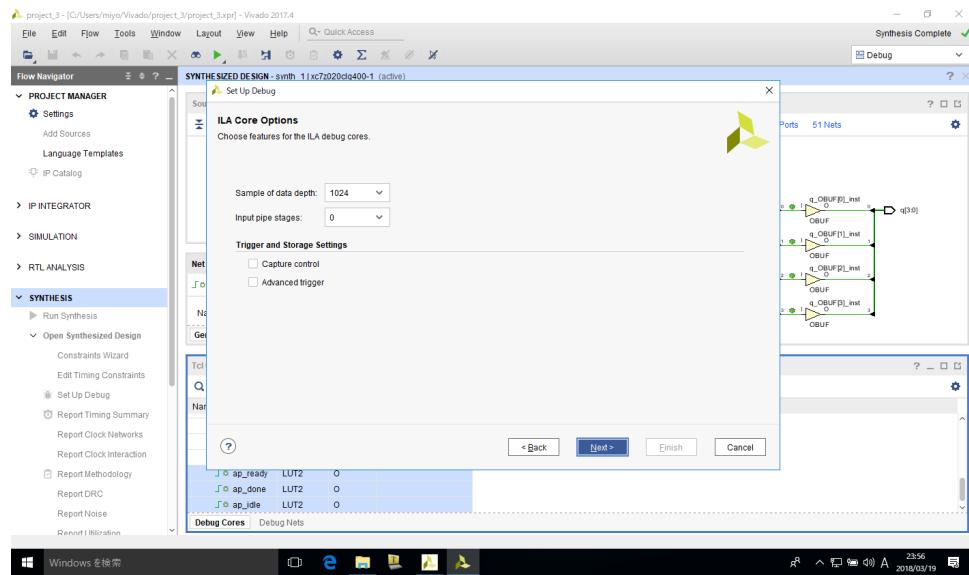


図 85 サンプル数や利用機能はデフォルトのままでよいので Next で次へ

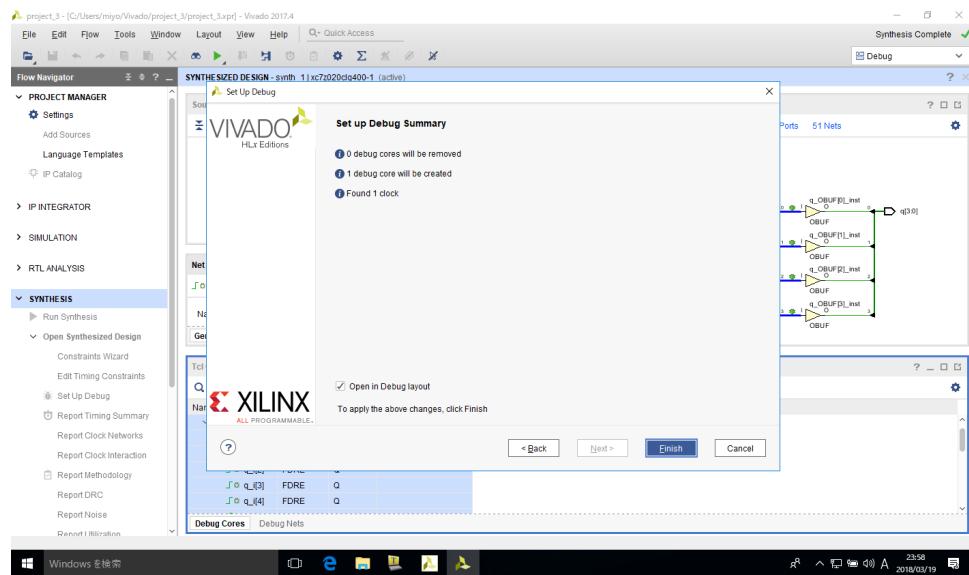


図 86 サマリを確認したら Finish でウィザードを閉じる

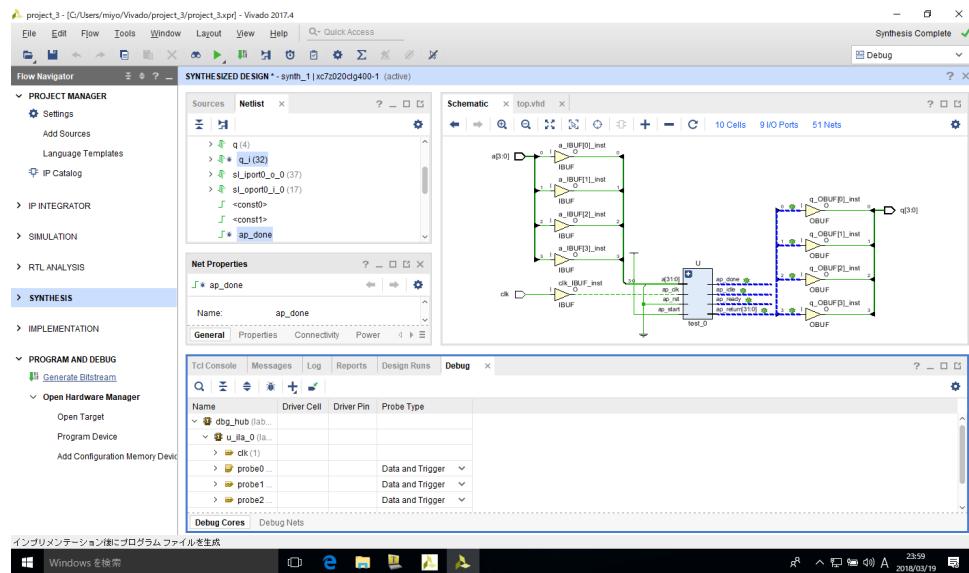


図 87 ILA の挿入ができたので、あらためて Generate Bitstream で合成と配置配線を行なう

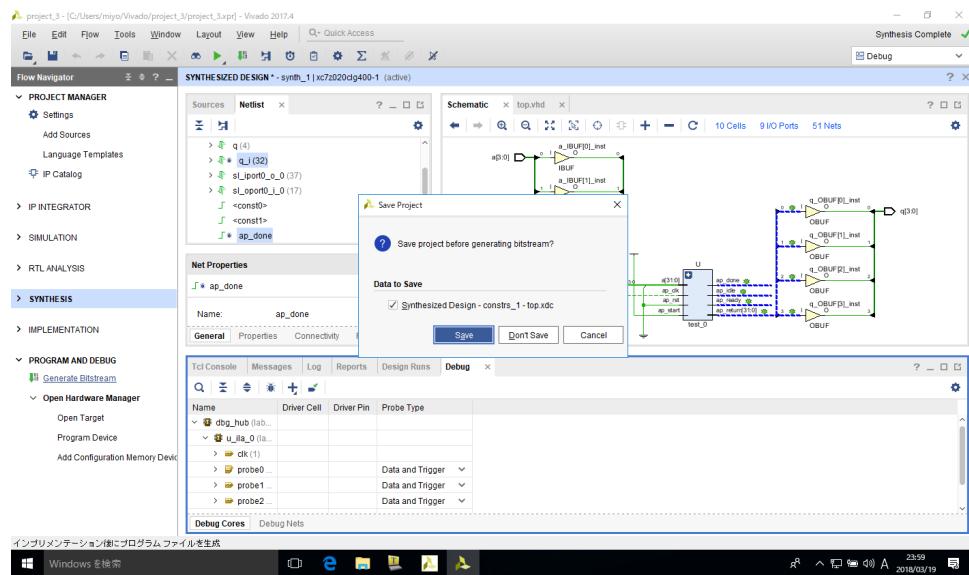


図 88 ILA に関する設定を保存するか確認を求められたら Save をクリック

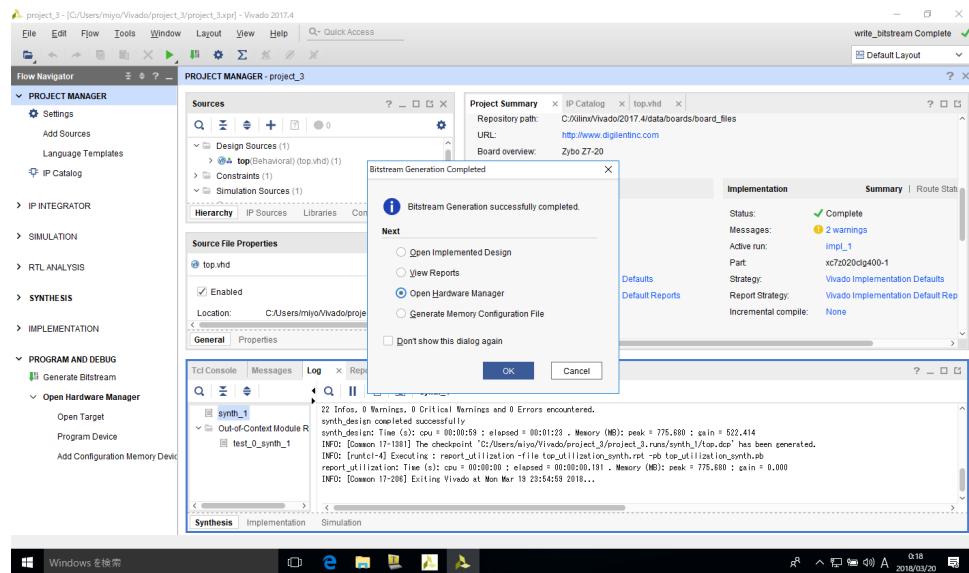


図 89 しばらく待つと ILA を仕込んだビットストリームが生成される。Open Hardware Manager を選択して OK をクリック

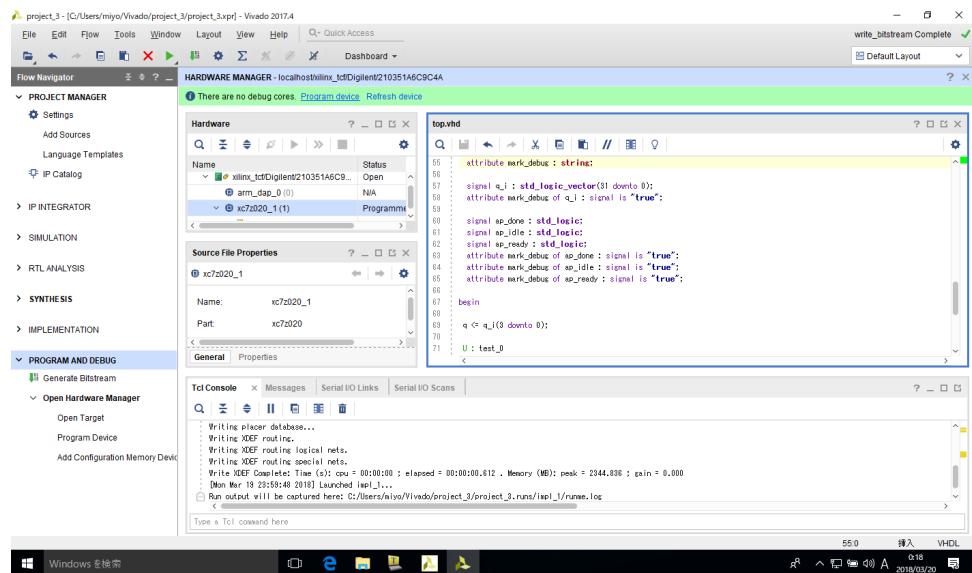


図 90 新しく作ったビットストリームを書き込むために Program Device をクリック

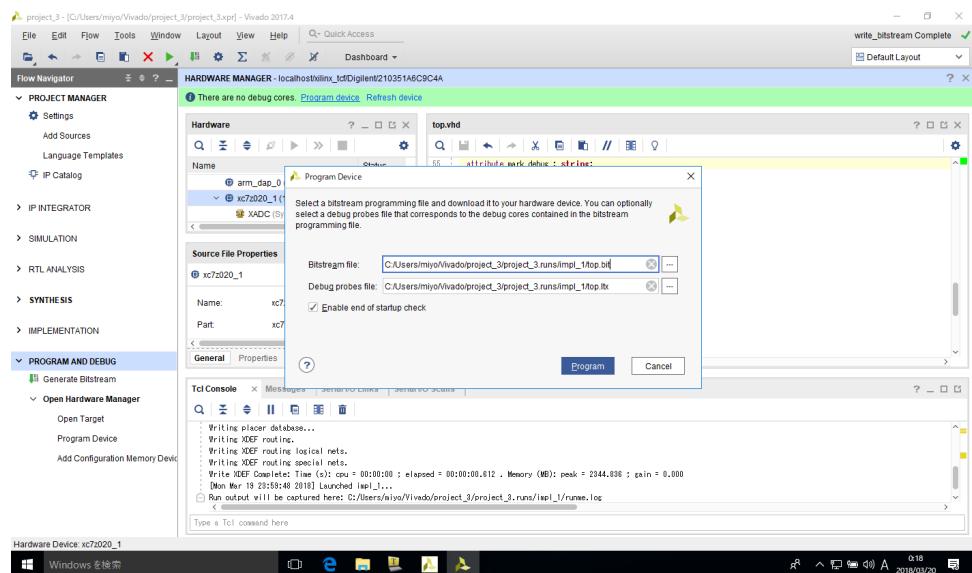


図 91 ビットストリームと ILA 用の定義ファイルがセットされていることを確認して Program をクリック

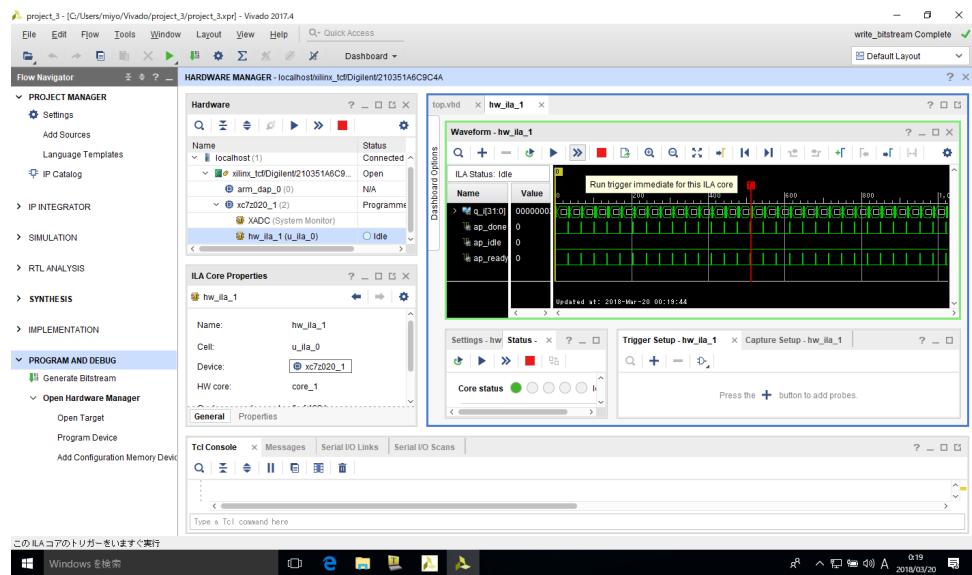


図 92 二重三角マークをクリックすると内部信号が確認できる

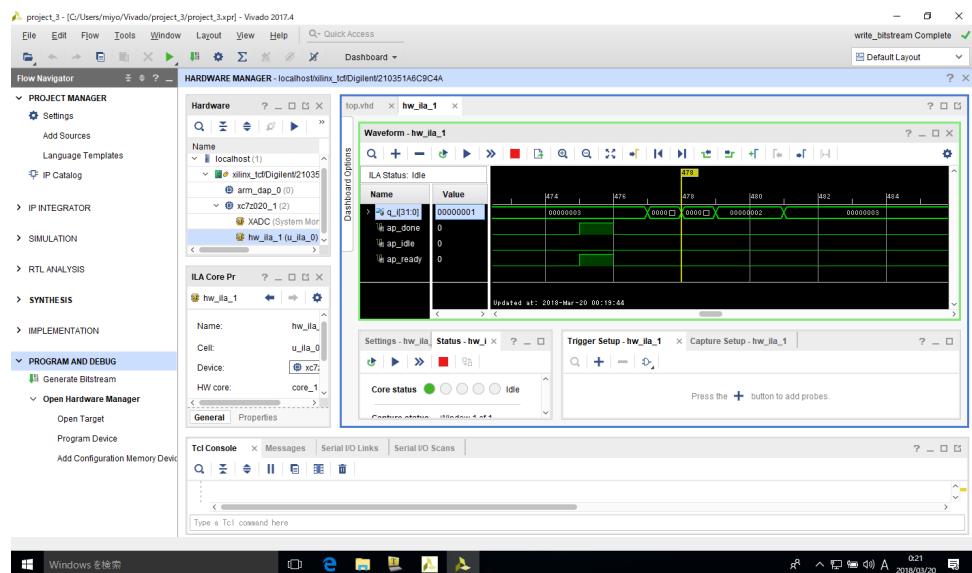


図 93 ap_done が'1'でない間に値がふらふらしていることが確認できた。

2.7 Vivado HLS のシミュレーション結果を詳細に確認する

実はILAを挿入するまでもなく、Vivado HLSのC/RTL協調シミュレーションの結果を注意深く確認することで、今回のバグは防ぐことができます。試してみましょう。

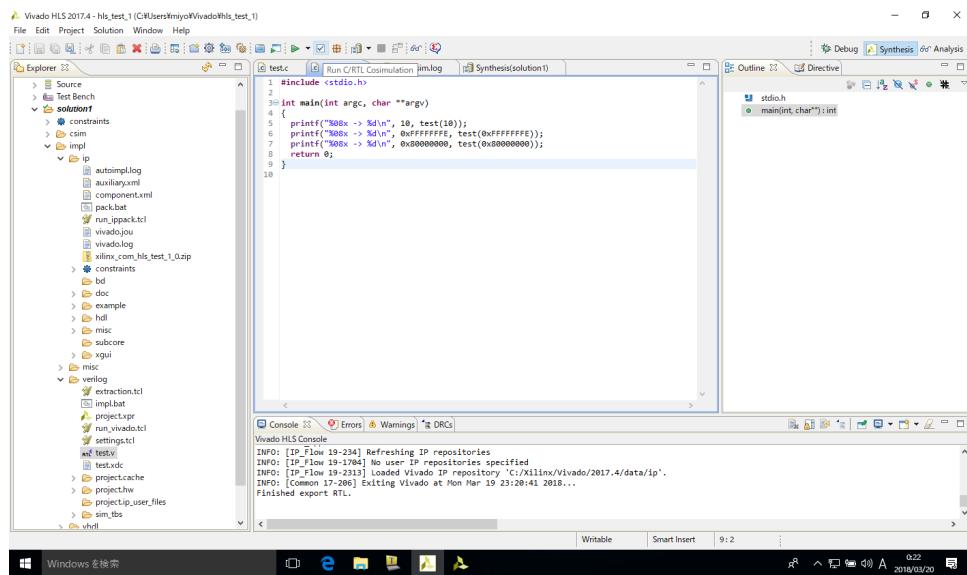


図 94 Vivado HLS で C/RTL シミュレーションを再度実行する

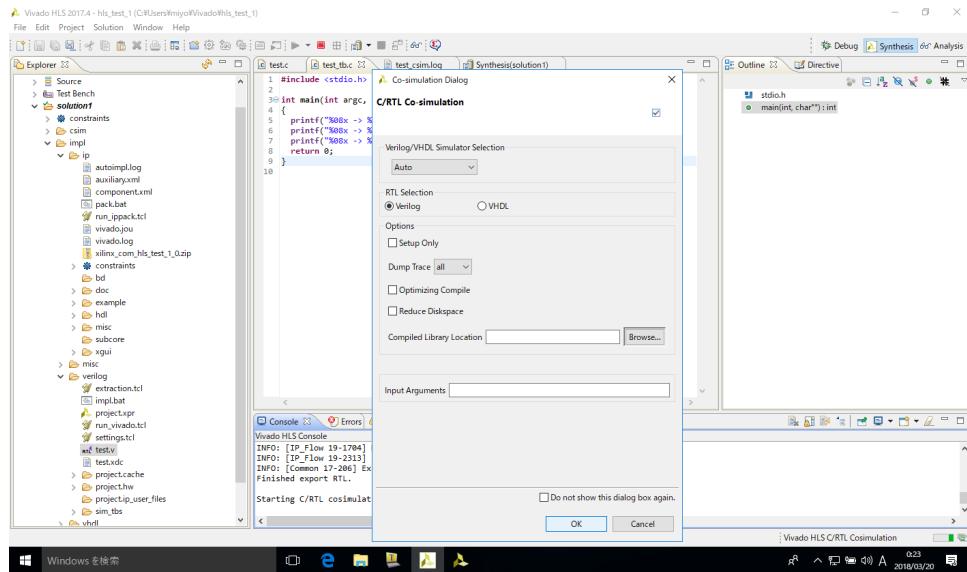


図 95 シミュレーション実行パラメタの Dump Trace で all を選択して OK をクリック

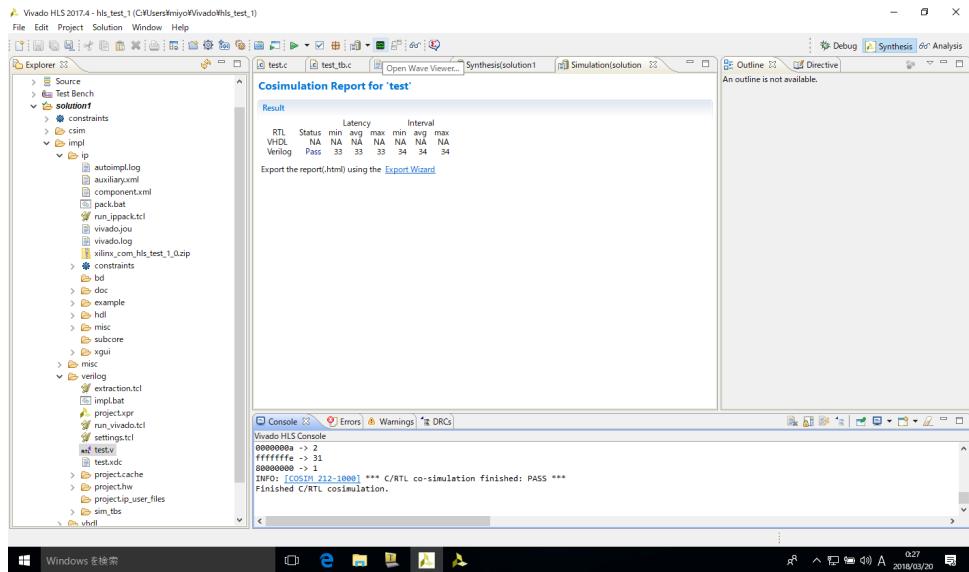


図 96 シミュレーションが完了すると、ツールバーの Open Wave Viewer をクリックする

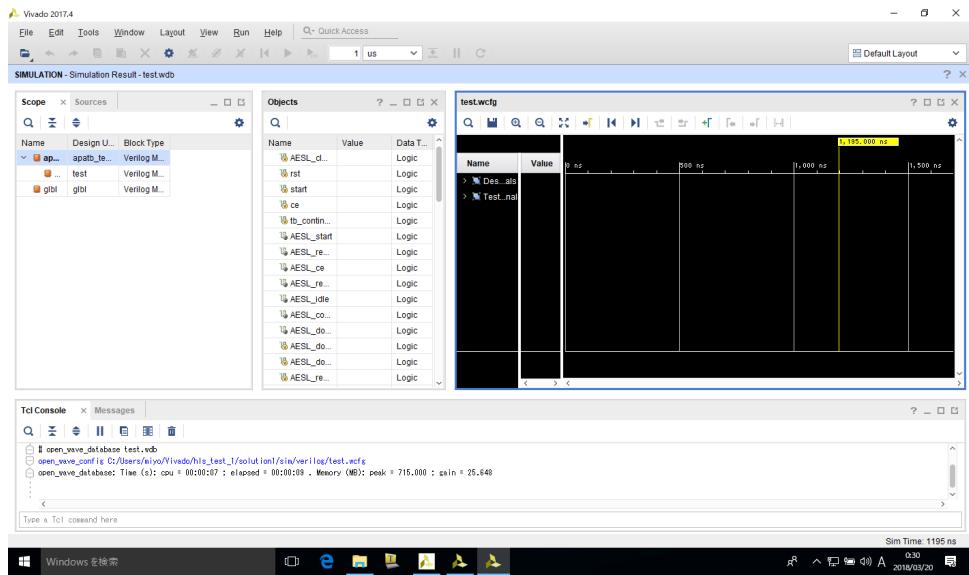


図 97 しばらく待つと Vivado シミュレータの波形ビューワが開く。ここで C/RTL シミュレータの結果を確認できる。

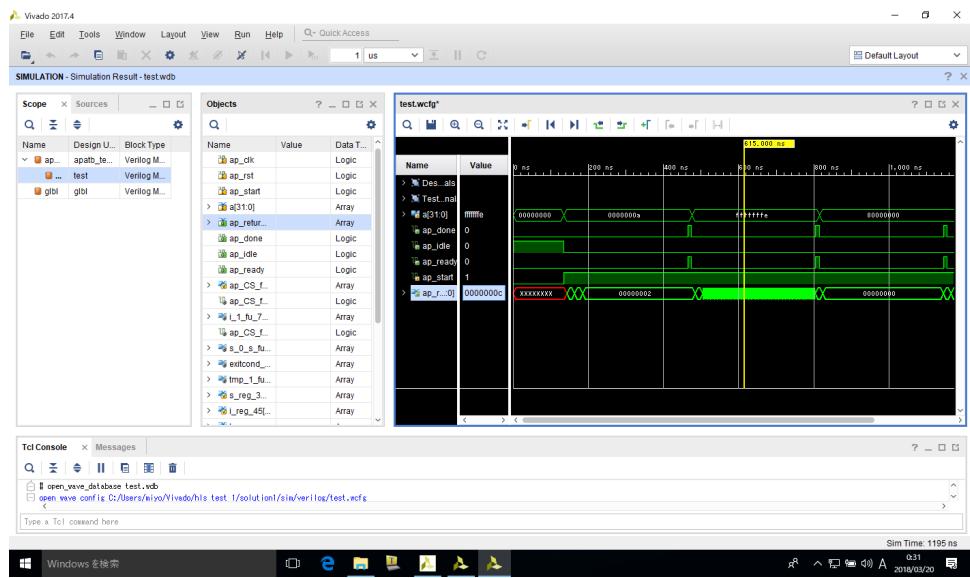


図 98 結果の信号や ap_done などを波形ビューワに追加してみたところ。ap_done が'1'でない場合に値が確定していないため注意しなければならない、ということがみてとれる。

2.8 再度実機での動作確認

バグ修正を反映して、もう一度、実機で動作を確認してみましょう。

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity top is
5     Port ( clk : in STD_LOGIC;
6             a : in STD_LOGIC_VECTOR (3 downto 0);
7             q : out STD_LOGIC_VECTOR (3 downto 0));
8 end top;
9
10
11 architecture Behavioral of top is
12
13     component test_0
14         port (
15             ap_clk : in std_logic;
16             ap_rst : in std_logic;
17             ap_start : in std_logic;
18             ap_done : out std_logic;
19             ap_idle : out std_logic;
20             ap_ready : out std_logic;
21             a : in std_logic_vector(31 downto 0);
22             ap_return : out std_logic_vector(31 downto 0)
23         );
24     end component;
25
26     attribute mark_debug : string;
27
28     signal q_i : std_logic_vector(31 downto 0);
29     attribute mark_debug of q_i : signal is "true";
30
31     signal ap_done : std_logic;
32     signal ap_idle : std_logic;
33     signal ap_ready : std_logic;
34     attribute mark_debug of ap_done : signal is "true";
35     attribute mark_debug of ap_idle : signal is "true";
36     attribute mark_debug of ap_ready : signal is "true";
37
38 begin
39     -- バグ修正のために追加
40     process(clk)
41     begin
42         if rising_edge(clk) then
43             if ap_done = '1' then
44                 q <= q_i(3 downto 0);
45             end if;

```

次のページに続く

```

1      end if;
2  end process;
-- バグ修正のために追加、ここまで
3
4
5 U : test_0
6 port map(
7     ap_clk => clk,
8     ap_rst => '0',
9     ap_start => '1',
10    ap_done => ap_done,
11    ap_idle => ap_idle,
12    ap_ready => ap_ready,
13    a(31 downto 4) => (others => '0'),
14    a(3 downto 0) => a,
15    ap_return => q_i
16 );
17
18 end Behavioral;

```

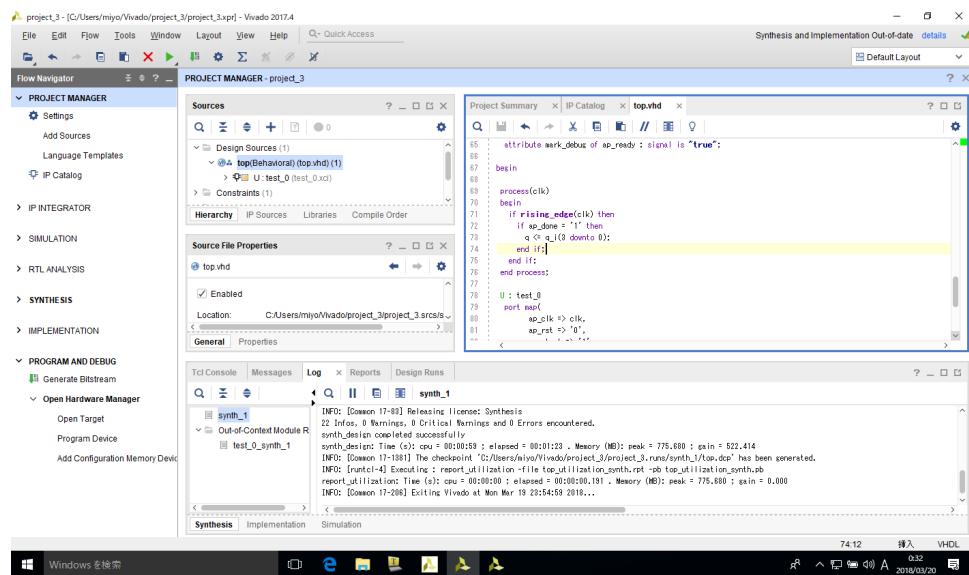


図 99 コードを書き変えたら Generate Bitstream をクリックして、再度ビットストリームを生成する

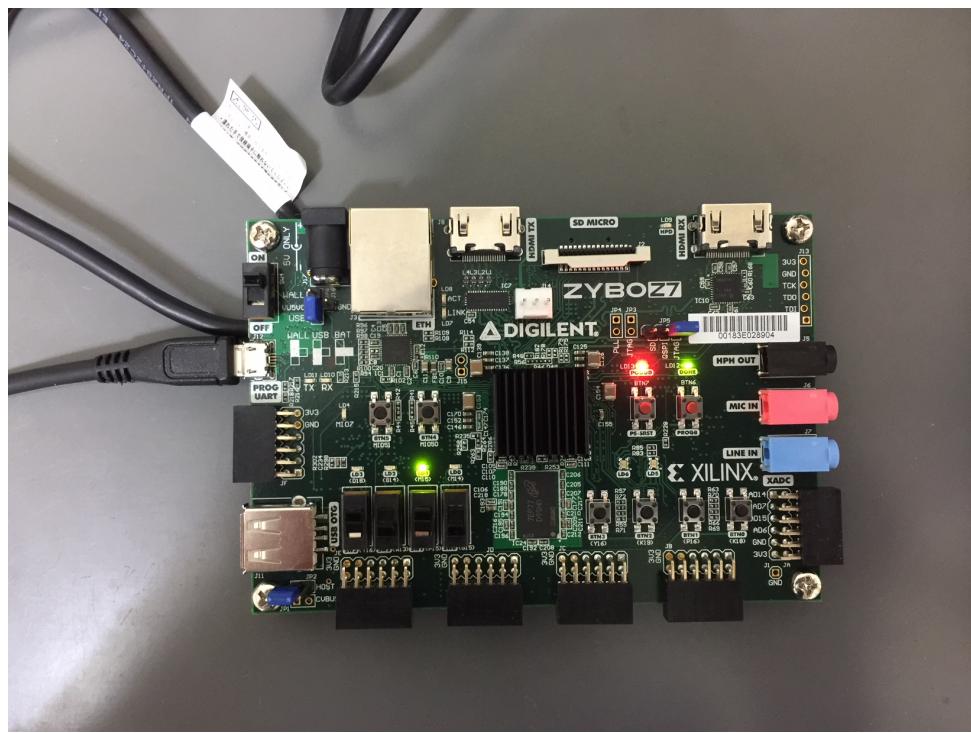


図 100 再び実機で動作を確認したところ。LED が正しく一つだけ点灯していることがわかる。

Vivado HLS 基本実験

VivadoHLS では、C/C++ からよりよいハードウェアを作成するために、いくつかの pragma で指示をあたえることができるようになっています。どんな pragma があり、どのような効果があるのか、試してみましょう。

1 データに関する pragma の調査と実験

Vivado HLS ではトップモジュールに選択した関数の引数が、RTL モジュールとの入出力になります。RTL モジュールとのインターフェースをどのようにしたいかは、引数に対する pragma で指定することができます。pragma によって、

- std_logic_vector のような単純な信号
- ブロックメモリ相当のインターフェース
- FIFO 相当のインターフェース
- AXI4-Stream 相当のインターフェース

を選択できます。ユーザガイドを参考に pragma を指定し、合成される HDL モジュールを調査してみましょう。C/RTL シミュレーションでインターフェースの振る舞いを観察してみるとよいでしょう。

2 回路生成方式に関する pragma の調査と実験

高速あるいは効率良く動作するハードウェアの設計では、データ並列性やパイプライン並列性の活用が必要不可欠です。Vivado HLS ではデータ並列性やパイプライン並列性の抽出は、ほぼ人手による pragma に委ねられています。言いかえれば、正しく pragam を指定できなければ、Vivado HLS をうまく使いこなすことはできません。

次のキーワードを参考に pragma を調べて、合成される HDL モジュールを調査してみましょう。

- DATAFLOW
- UNROLL
- PIPELINE

C/RTL シミュレーションで、pragma の指定の有無による実行サイクル数の変化を観察してみるとよいでしょう。

Vivado HLS 応用実験

Vivado HLS をつかった応用実験に取り組んでみましょう。

1 C/C++ ベースの音声信号処理に挑戦

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。音声信号の送受信は HDL モジュールで記述したものを利用することとして、ここでは、音声信号処理のカーネルを C/C++ で実装してみましょう。

たとえば、

1. 規定値より大きな値がきたときだけ音声を出力する,
2. 平滑化フィルタで音を滑らかにする,
3. 他の音源とまぜあわせる
4. 圧縮・伸長する

などが考えられます。pragma をうまく活用してパイプライン化することで、音声信号をリアルタイムに処理することができます。

2 発展: C/C++ ベースの画像処理に挑戦

2.1 準備

ZYBO Z7-20 には HDMI の入力と出力があります。この入出力を利用するアプリケーションを開発してみましょう。

2.2 画像処理の課題例

Xilinx からは、OpenCV 関数相当の画像処理ルーチン xfOpenCV が提供されています。それらを使用することで比較的容易に複雑な画像処理を FPGA 上に実装できます。それらを使用するか、あるいは、自分で C/C++ で画像処理ルーチンを書いて FPGA での画像処理に挑戦してみましょう。

たとえば、

1. 左右の反転.
2. アップコンバージョン/ダウンコンバージョン
3. 平滑化フィルタで画像を滑らかにする,
4. 圧縮・伸長する

などが考えられます。

音声信号処理同様、あるいは、音声信号処理以上に、pragma をうまく活用してパイプライン化することでリアルタイムに処理できるハードウェアの設計がもとめられます。