再構成型アーキテクチャ特論 （4）

osana@eee.u-ryukyu.ac.jp

前々回のまとめ

- 継続代入文 (assign)
  - 組み合わせ回路ができる。必要な演算器などは合成ツールが判断
    - wire 型の信号 (ポート宣言は暗黙の wire 宣言)
    - ブロッキング代入演算子 (=)
    - ? を使った条件付き代入も可能だが、マルチプレクサに注意

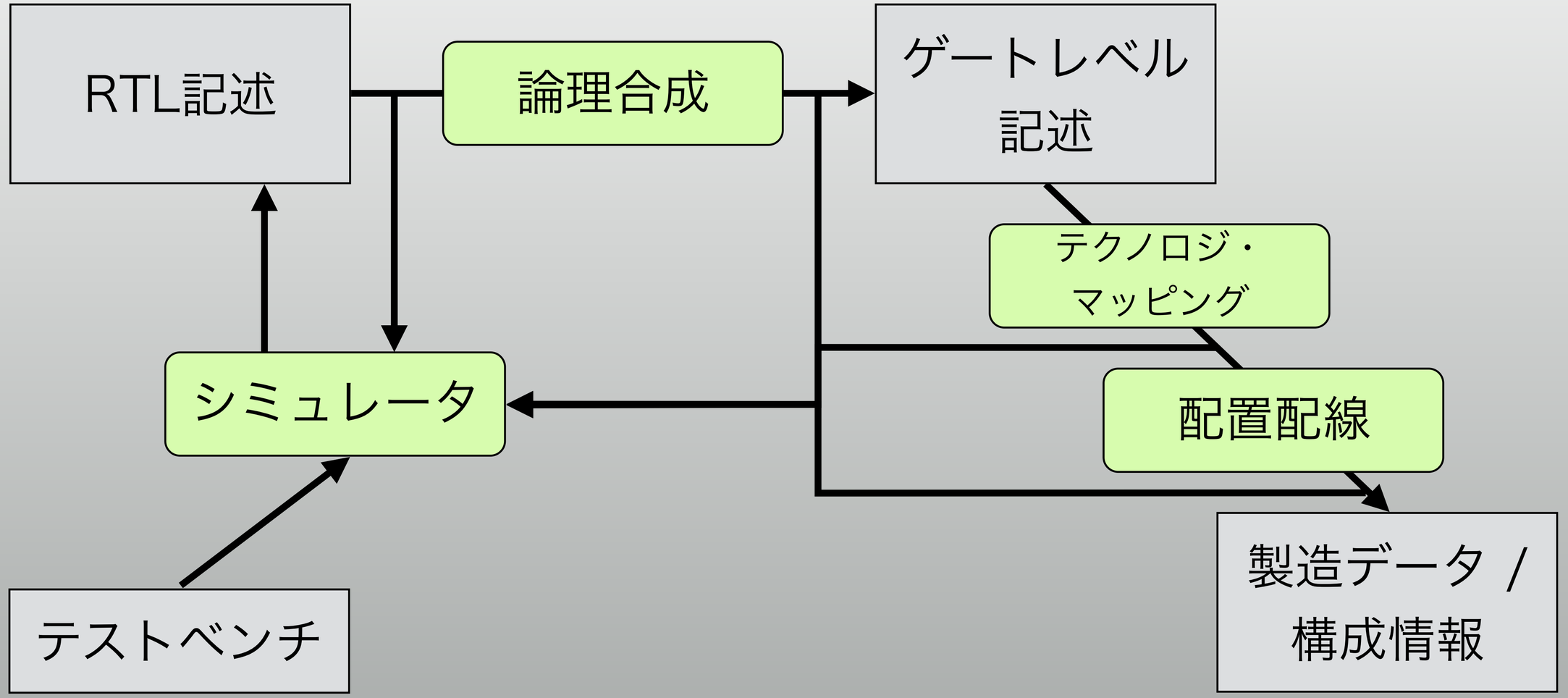
前回のまとめ

- 手続き代入文 (always)
  - always @ (…) の () の中にはクロック信号「だけ」を書く
  - reg 型の信号 (output regも): 必要に応じてFFが合成される
  - ノンブロッキング代入 (<=)
  - 条件付き代入に加えて if や case といった制御構造が使える

今回の予定

- HDL シミュレータによる動作検証
  - 検証の対象は先週までの内容を利用
  - テストベンチの書き方
  - Vivado simulator によるシミュレーションフロー

設計フロー


RTL記述
論理合成
ゲートレベル
記述
テクノロジ・
マッピング
シミュレータ
配置配線
製造データ /
構成情報
テストベンチ

シミュレーションは重要

- 設計のあらゆる局面で
  - RTL を書いて検証、論理合成して検証…
  - とはいえ、FPGA では気を付けていれば RTL だけでもなんとか
    - 実チップが動かなくてもやり直せる、というのは大きいメリット

テストベンチが必須

- 回路は勝手には動かない
  - 外からの入力が必要
- 波形を目で見て全部チェック、というのは実際ムリ
  - printf() 的なデバッグは結局のところ、偉大
  - 特定の状況が発生した時間などがわかれば検証の助けになる

シミュレーションしましょう

- Verilog シミュレータが必要 (講義ではXilinx Vivado simulatorを使用)
  - Cadence: NCsim (NC-Verilog)
  - Synopsys: VCS
  - Mentor Graphics: ModelSim
  - Stephen Williams: Icarus Verilog (フリーソフトウェア)

波形ビューワ

- シミュレータと統合されているもの
  - Vivado Simulator, ModelSim
- 分離されているもの
  - VCS, NCsim (simvision), Icarus Verilog (gtkwave)

波形 (waveform) ファイル

- VCD (Value Change Dump): 標準フォーマット。ASCII形式
  - VCD はどのシミュレータでも生成できる
- 各社独自フォーマットのほうがファイルサイズが(ずっと)小さい
- VPD (VCD Plus): Synopsys, SHM (Simulation History Manager): Cadence, WLF (Waveform Log File?): Mentor Graphics

波形ファイルを何に使うか

* 波形ビューワで見る: 当然です
* 消費電力推定ツールへ入力する: より正確な推定のために必要
* On-chip debug のツールから出力する: 実機の波形みながら問題解決

テストベンチとRTLの違い (1)

- テストベンチには時間の概念がある
  - initial 文と $finish、それから timescale
  - RTLではイベントはあっても、基本的に時間の経過はない

テストベンチとRTLの違い (2)

- テストベンチには入出力ポートがない
  - RTL の世界の外側全部を書きます
- テストベンチは論理合成しない: 論理合成を前提としない記述が自由
  - システムタスクなど、シミュレーションの制御に関するいろいろ

時間の経過に関連する記述 (1)

- `timescale : 単位時間と時間解像度を指定
  - “`timescale 1ns/1ps” とかが普通
    - #1 で 1ns の遅延が生じる
    - 1ps 未満の遅延は四捨五入する
- # : 文の実行や代入を遅延させる (timescale に依存)

時間の経過に関連する記述 (2)

- initial : 時刻ゼロで実行される手続き代入文
  - # を使って時系列でイベントを書いていくことができる
- always # : 一定の時間おきに実行される手続き代入文
  - always # (10) で 10ns おきに動く、とか
  - クロック作ったりするのに便利

テストベンチの例: Step 1

- クロックは10ns周期: 100MHz
  - 11ns でリセット
  - 31ns でリセット解除
- テスト対象はまだない

```
`timescale 1ns/1ps

module testbench ();
  reg CLK, RST;

  initial CLK <= 1;
  always # (5) CLK <= ~CLK;

  initial begin
    RST <= 0;
  #11
    RST <= 1;
  #20
    RST <= 0;
  end
endmodule
```

テストベンチの例: Step 2

- parameter を使用
  - 周期をわかりやすくする
  - 整数型だと *1.1 などで
    桁落ちの可能性がある

```
`timescale 1ns/1ps

module testbench ();
  reg CLK, RST;
  parameter real STEP = 10;

  initial CLK <= 1;
  always # (STEP/2) CLK <= ~CLK;

  initial begin
    RST <= 0;
  #(1.1*STEP)
    RST <= 1;
  #(2*STEP)
    RST <= 0;
  end
endmodule
```

RTL で書くものはなんでも書ける

- たとえばクロックのカウンタ
  - 波形ツールみながら、
    何クロック目かわかると便利
  - $display(後述)とかでも便利

```
initial CLK <= 1;
always # (STEP/2) CLK <= ~CLK;

reg [31:0] CLK;
always @ (posedge CLK)
  CNT <= RST ? 0 : CNT+1;

initial begin
  RST <= 0;
#(1.1*STEP)
  RST <= 1;
#(2*STEP)
  RST <= 0;
end
```

システムタスク

- 主にシミュレータが解釈していろいろするためのコマンド的なもの
  - シミュレーション中のメッセージ表示や波形ファイルの保存
  - 実数型の取り扱いをするための関数 (sin とか cos も) など
    - デバッグ用のメッセージを作るのに使う
  - 基本的に論理合成ツールは無視する (エラーになるものもある)

$display, $write: 標準出力 (1)

- $display (“フォーマット”, 信号, 信号…);
  - printf() 的なもの。ただし最後は改行される ($write は改行しない)
  - %b: 2進、%d: 10進、%h: 16進、%f: 実数　など
  - \t と \n でタブや改行も OK
  - 表示したいタイミングは initial / always の中で自分で決める

$monitor: 標準出力 (2)

* $display と違って、値が変化したら表示される
  * $monitoron / $monitoroff で停止 / 再開ができる

$f{display, write, monitor}

- ファイルに出力する版で、だいたい C と同じ
  - mcd = $fopen(“filename”);
  - $fdisplay(mcd, “format”, signal, signal…);
  - $fclose (mcd);

シミュレーション時間の関数

- $realtime
  - real 型で実時間
  - $display(“Time = %f ”, $realtime); とか
- $time
  - 64bit 整数で、timescale をかければ実時間

シミュレーション制御

- $finish: シミュレーションを終了
  - $finish(1) とか $finish(2) とするとCPU時間とかが出る (かも)
- Vivado simulator などではシミュレーションの終了時刻を指定してからシミュレーションを走らせるので必要ない
  - コマンドラインで使うシミュレータでは重要

データ型変換など

- $bitstoreal, $realtobits: real型と64bitの信号(IEEE-754倍精度)の変換
  - 実数計算をする回路のデバッグで重宝します
- $itor, $rtoi: real型とinteger型の変換
- $sin, $cos,…: いろいろな関数もあります。乱数とかも

波形ファイル保存

- $dumpfile(“hogehoge.vcd”); hogehoge.vcdを波形保存先として開く
- $dumpvars(0);　すべての信号を $dumpfile で開いたファイルに記録
- ベンダ固有のものもある
  - Vivado simulator などGUI統合型では指定しなくても波形が見られる

システムタスクまとめ

* $display, $monitor, $write, $fdisplay, $fmonitor, …
* $realtime, $time
* $finish
* $bitstoreal, $realtobits, $itor, $rtoi, $sin, $cos, …
* $dumpfile, $dumpvars

テスト対象モジュールの接続

- 通常のサブモジュールと同じ
  - inputは基本的にテストベンチの持つ reg で生成
    - テストベンチに他のシステムモデル (メモリなど) を含むなら、それとの間を wire で接続することも
  - output は wire に接続

たとえば…

```
`timescale 1ns/1ps

module sw_led_tb ();
  reg [3:0]  SW;
  reg        PUSH;
  wire [3:0] LED;

  sw_led uut (.SW(SW), .LED(LED),
              .PUSH(PUSH));

  initial begin
          SW <= 4'b0001; PUSH <= 0;
    #(10) SW <= { SW[2:0], SW[3] };
    #(10) SW <= { SW[2:0], SW[3] };
  end
endmodule
```

```
module sw_led
  (
   input  wire [3:0] SW,
   input  wire       PUSH,
   output wire [3:0] LED
  );

  wire  SW1_ = ~SW[1];
  assign LED[0] = PUSH & SW[0];
  assign LED[1] = SW1_;
  assign LED[2] = SW1_ & SW[2];
  assign LED[3] = |SW;

endmodule
```

信号を観測する

- 波形をビューワで見る: モジュール階層を下へたどっていける
  - つまりデザインに含まれるどの信号でも見ることができる
- 波形見てやるだけ、というのは意外としんどい

テスト用のロジックを HDL で書く

- テスト用のロジックをテストベンチに作る
  - テストベンチで “OK” みたいな信号を作るとか
  - 特定の条件で $display を使うとか
- テスト対象モジュールのポートに出ていない信号は見えないの？

見えます！

- そう、Verilog ならね…
  - uut.SW1_ とか
  - インスタンス名をつないでいけばもっと深い階層でもOK
  - 合成するコードでは使えない
- VHDL ではムリだそうです

```
module sw_led
  (
   input  [3:0] SW,
   input        PUSH,
   output [3:0] LED
  );

  wire  SW1_ = ~SW[1];
  assign LED[0] = PUSH & SW[0];
  assign LED[1] = SW1_;
  assign LED[2] = SW1_ & SW[2];
  assign LED[3] = |SW;

endmodule
```

やってみる

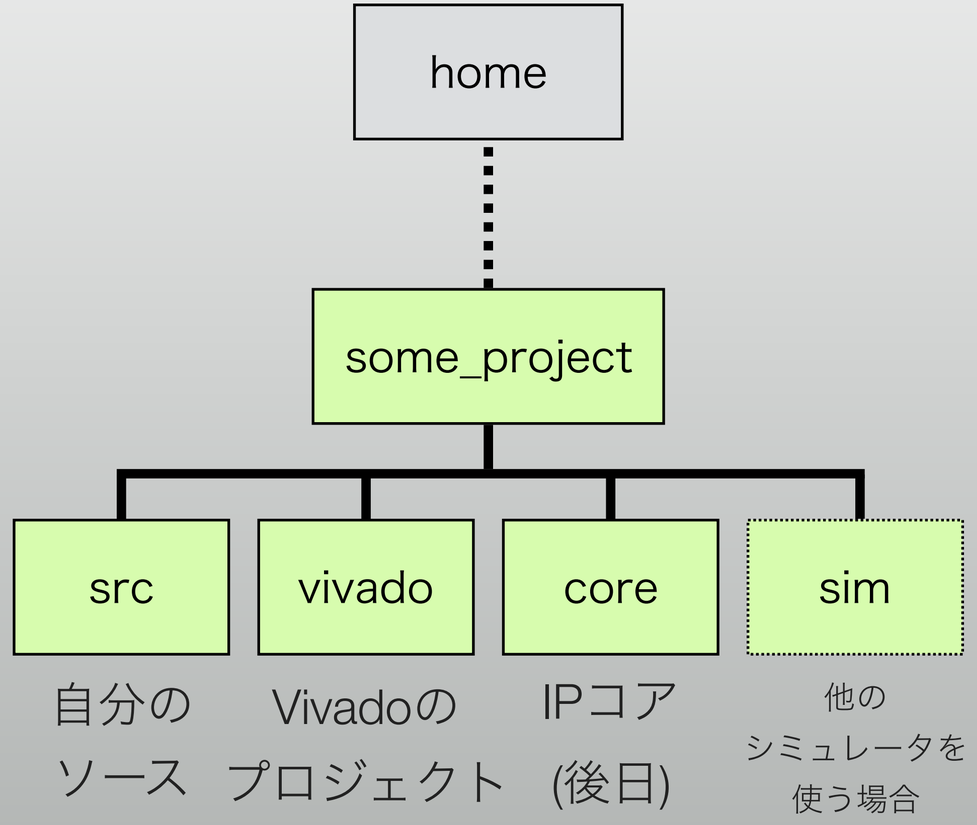
- Vivadoでシミュレーションする (合成までそのままいけます)
  - テストベンチとRTL記述も書きましょう
  - ターゲットデバイスを決めてプロジェクトを作る必要あり
- 普通のシミュレータを使う場合
  - 純粋なRTLのシミュレーションにはデバイスの指定は不要

ソースファイルをどこに置くか

- Vivado のデフォルトはプロジェクトフォルダの下
  - CAD はたいていたくさんのファイルを作ります
  - 自分のソースファイルと混じるとややこしい
    - プロジェクトを作り直したり、再利用するときに困る

おすすめのディレクトリ構成

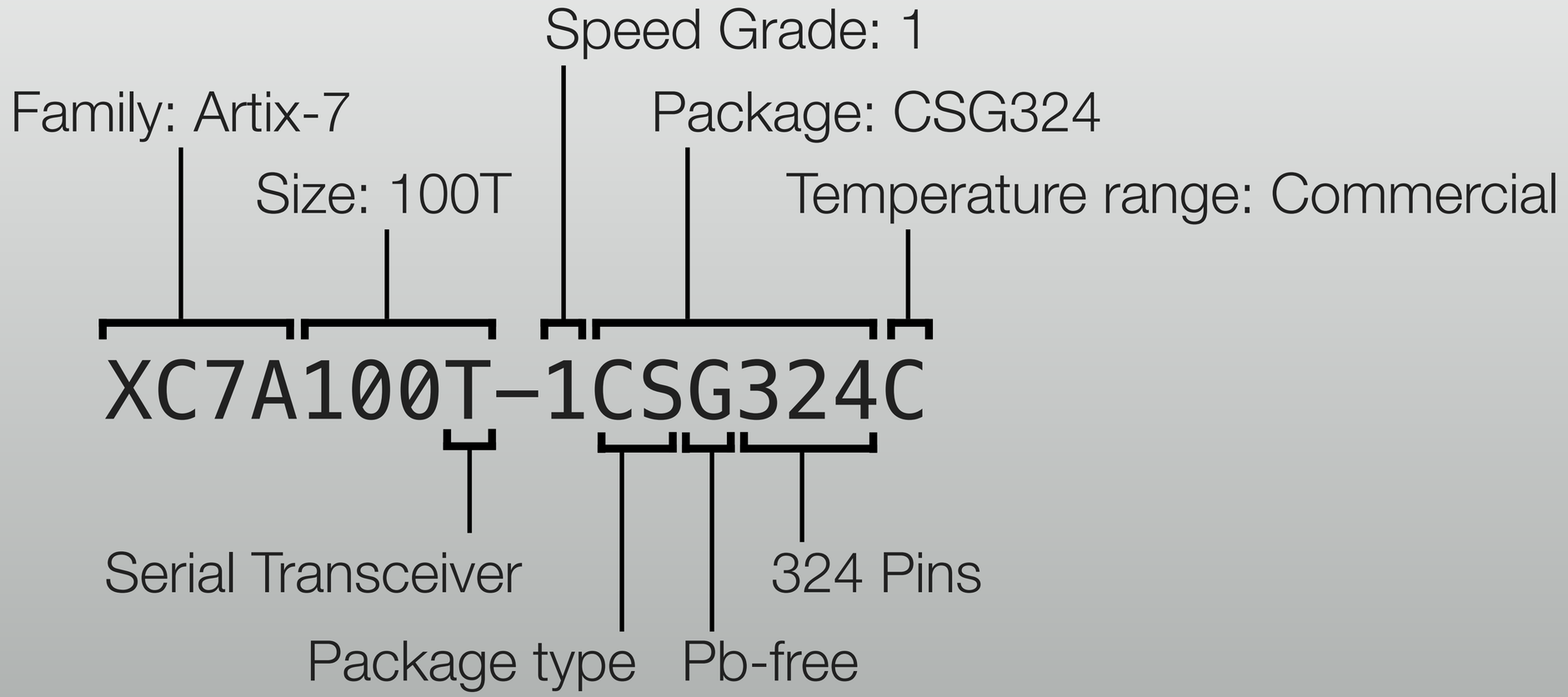
- ソースコードとプロジェクト
  - ソースコードのフォルダは
    自分で中身を管理する
  - プロジェクトからはソースを
    参照するだけにする



FPGAの型番って何…?

- ファミリ
  - Xilinx: Virtex, Kintex, Artix, Zynq, Spartan, …
  - Altera: Stratix, Arria, Cyclone, MAX, …
- デバイスサイズ・追加機能
- パッケージ・スピードグレード

型番の例

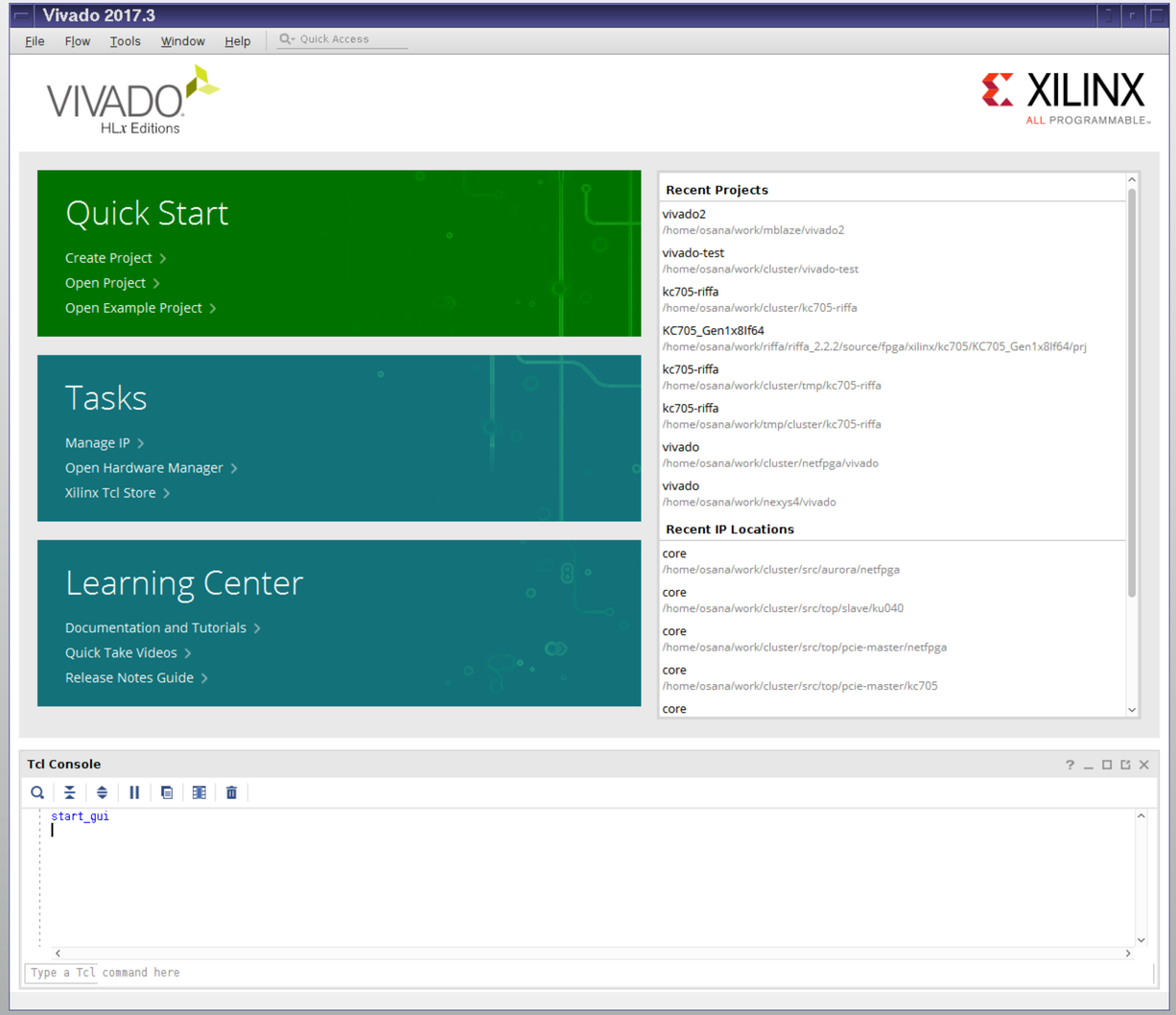

Speed Grade: 1
Family: Artix-7
Package: CSG324
Size: 100T
Temperature range: Commercial
XC7A100T–1CSG324C
Serial Transceiver
Package type
Pb-free
324 Pins

実習

- Vivado でプロジェクトを作成
  - 簡単なRTLとテストベンチを作成
  - シミュレーションしてみる
- 簡単にするためにプロジェクトフォルダの中にソースを作成

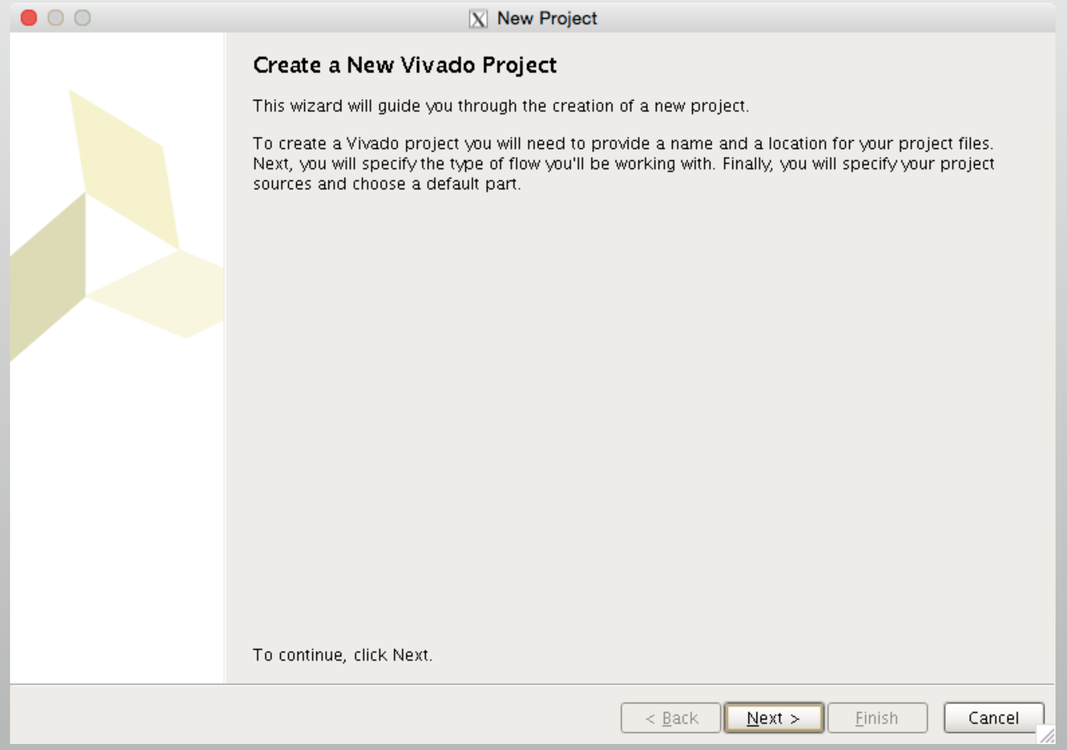
Vivadoを起動

* Create Project する



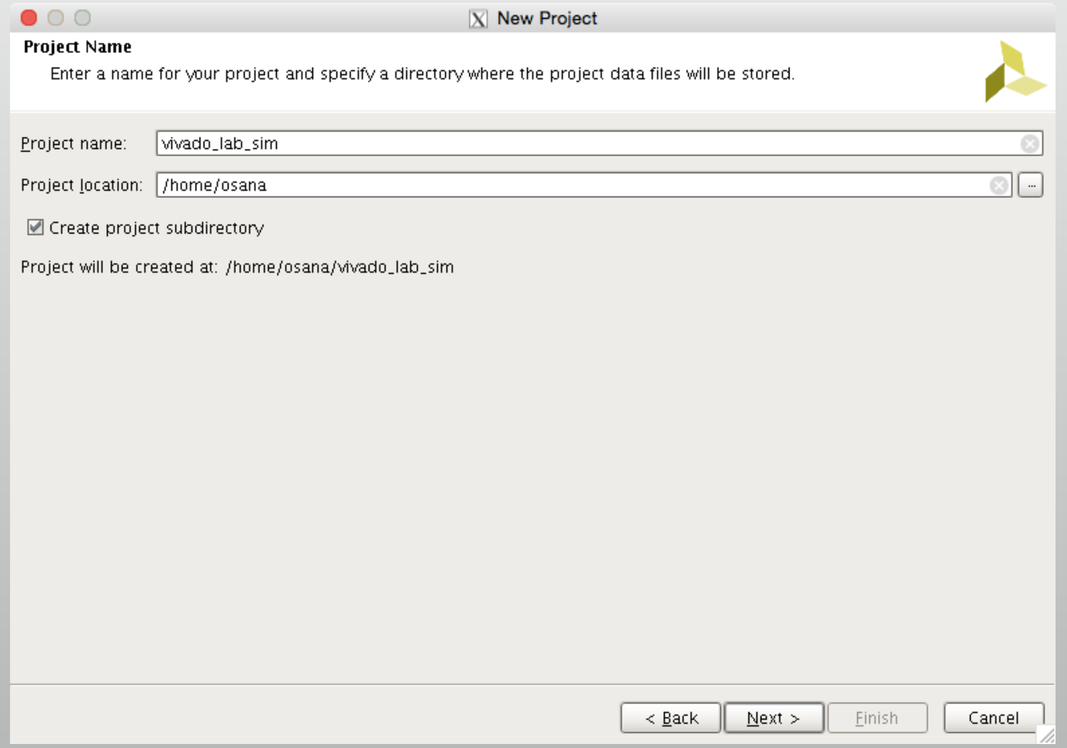
プロジェクトを作る (1/5)

* とりあえず次へ



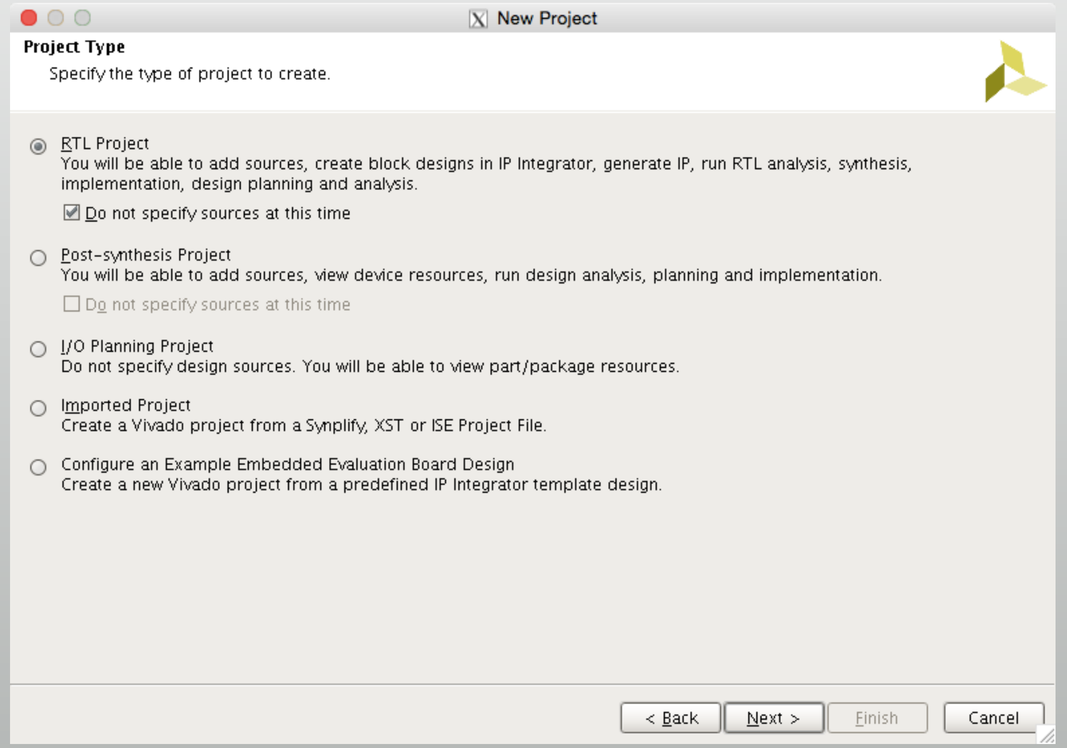
プロジェクトを作る (2/5)

- プロジェクトの名前と場所
  - “vivado_lab_sim” にしました
  - プロジェクト名でフォルダが作成されます



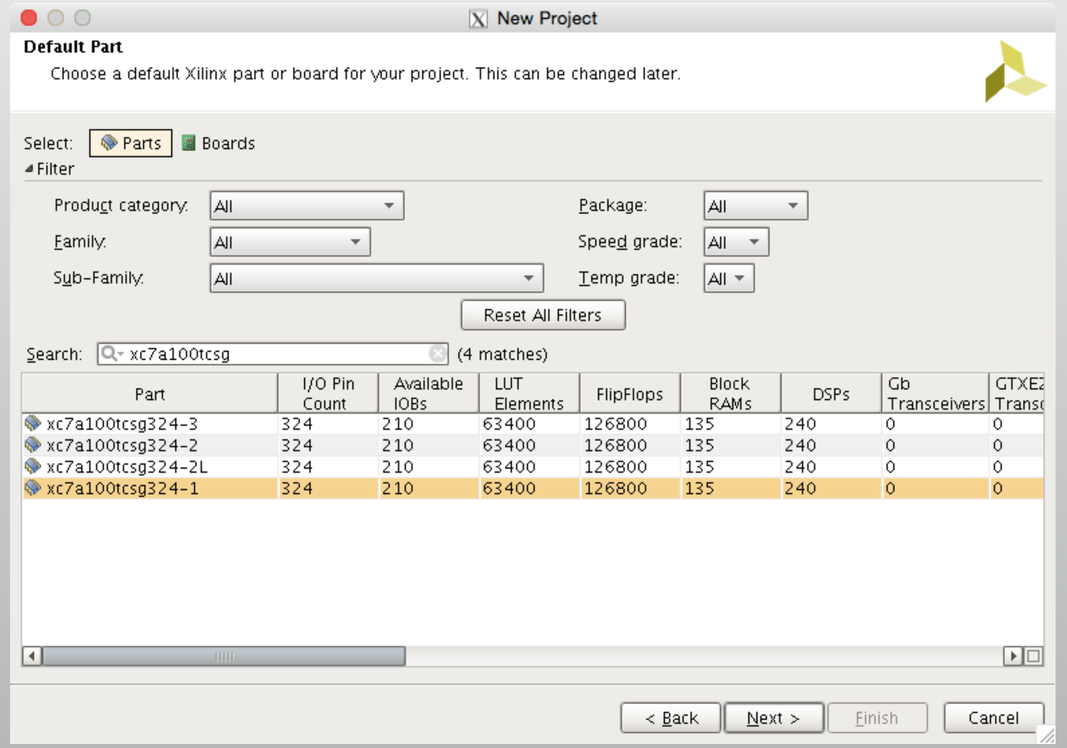
プロジェクトを作る (3/5)

- 基本は”RTL Project”
  - まだソースがないので、
    “Do not specify sources…”



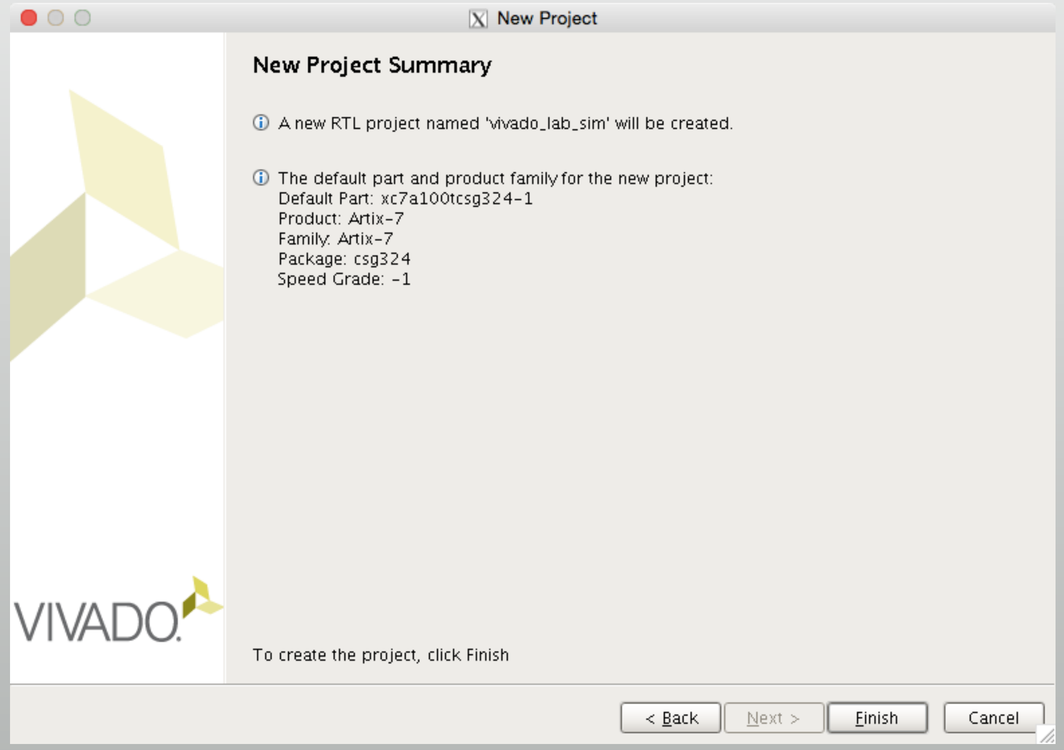
プロジェクトを作る (4/5)

- デバイスを選ぶ
  - XC7A100TCSG324-1
  - “Search” に入力しても、
    フィルタで絞り込んでもOK

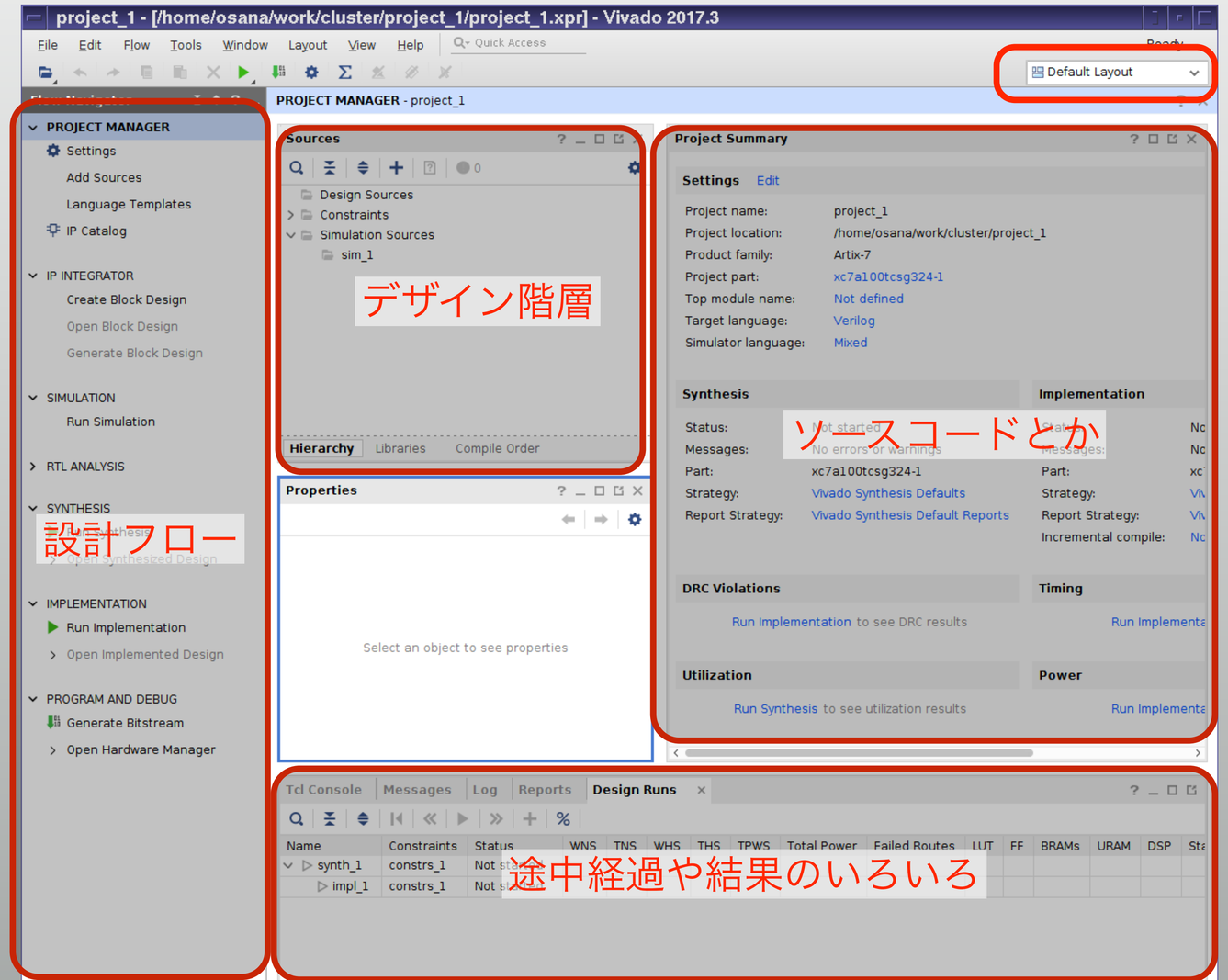


プロジェクトを作る (5/5)

* 最後確認しましょう



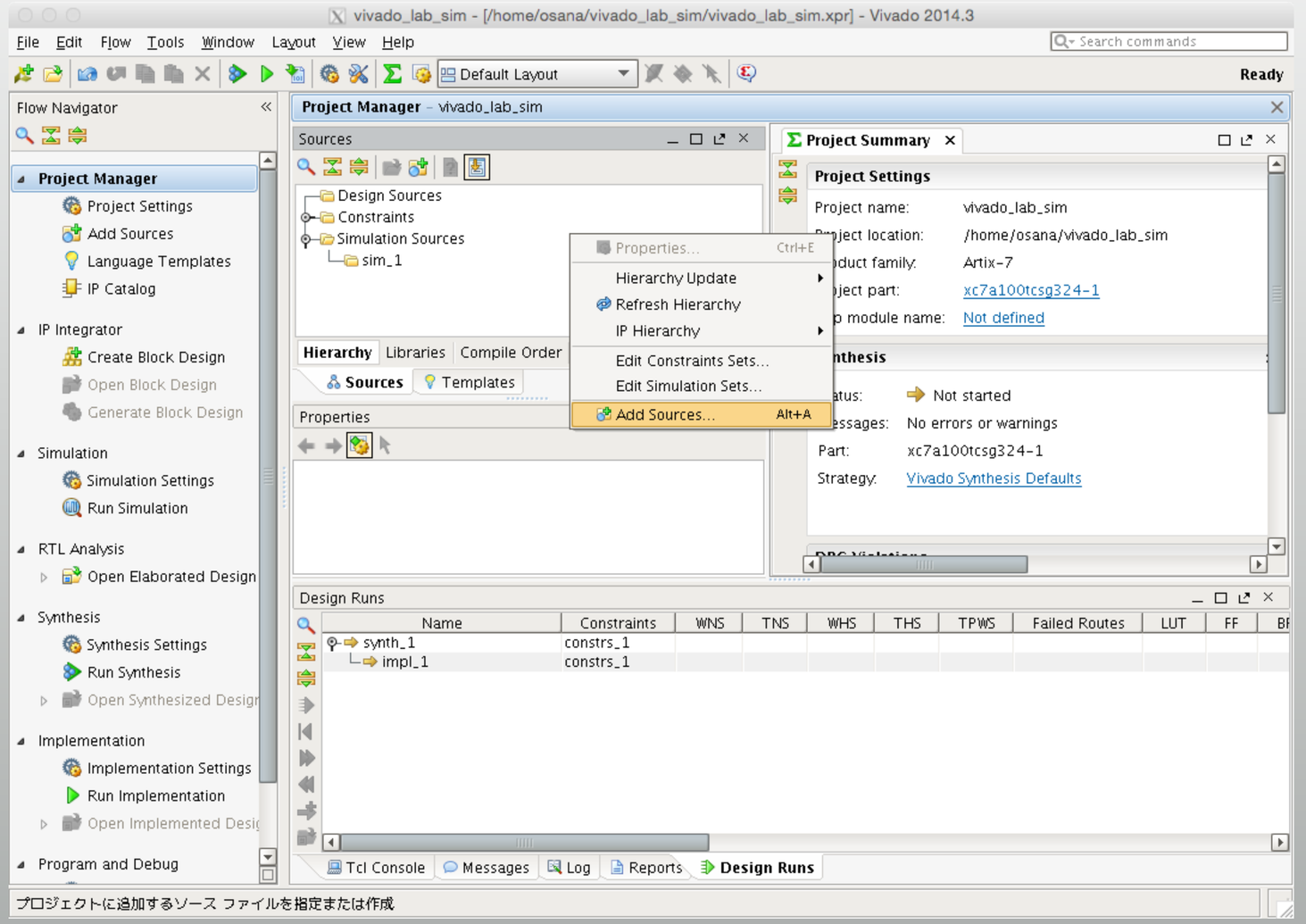
Vivadoの画面



画面のレイアウトが
わからなくなったら
ここを “Default Layout”

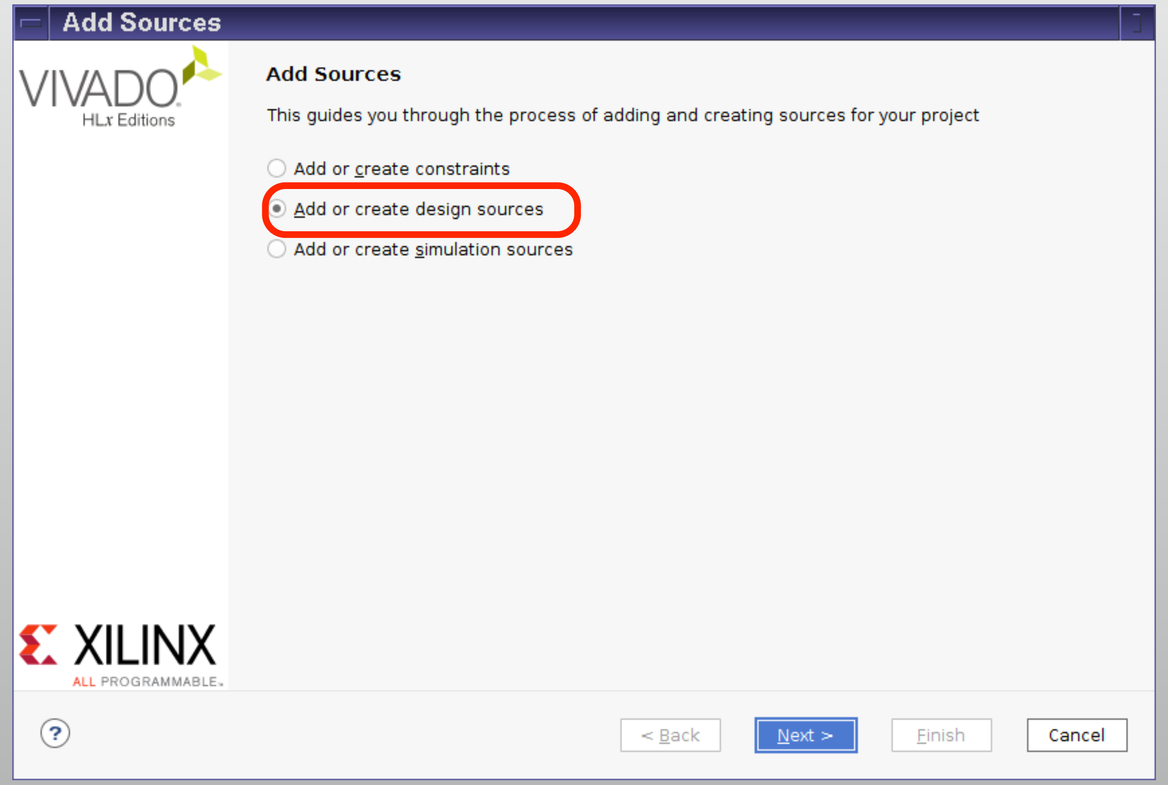
ソースコードを追加 (1/4)

* デザイン階層に
  ソースを追加



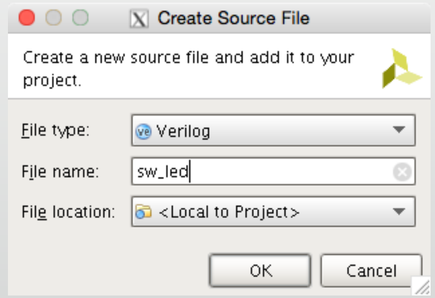
ソースコードを追加 (2/4)

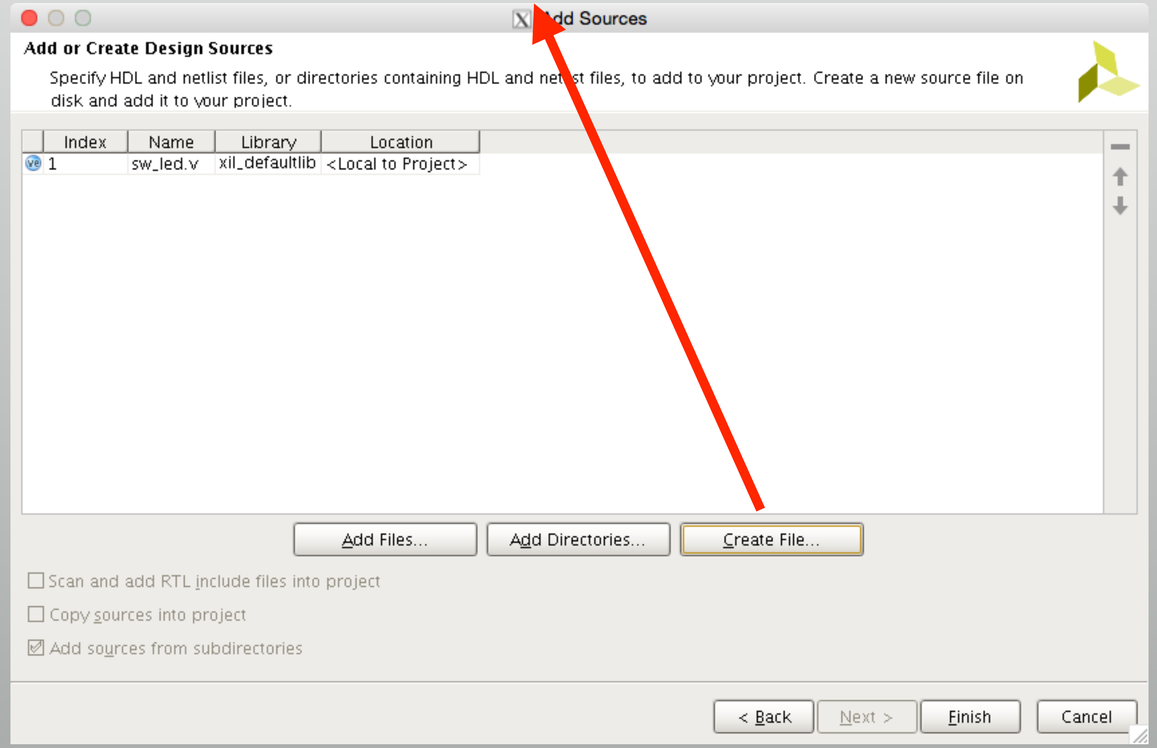
- RTLは “Design Source”



ソースコードを追加 (3/4)

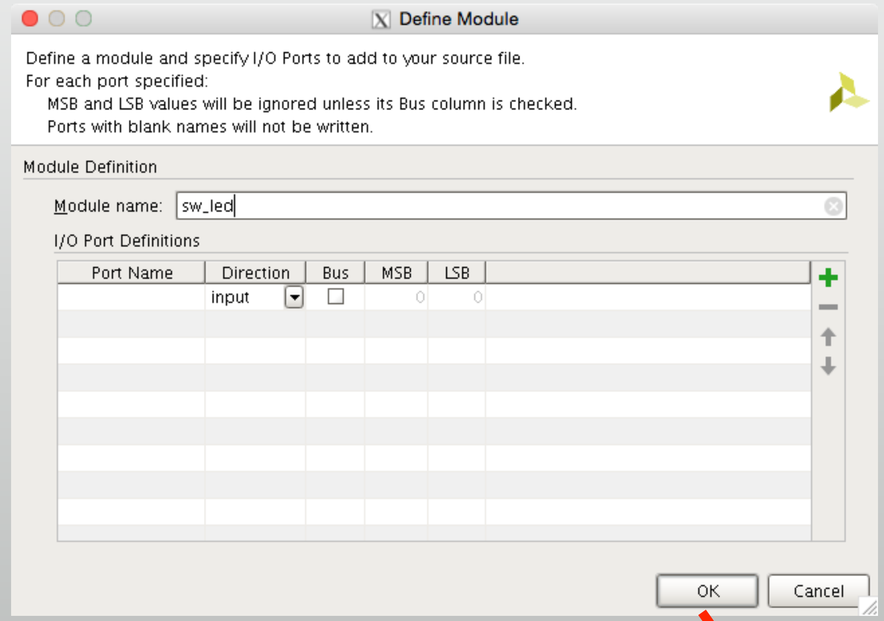
- これから書くので“Create File”
  - 今回はsw_led という名前で
  - 他のフォルダにあるのを
    使う場合は “Add Files”

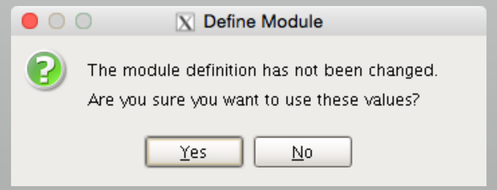




ソースコードを追加 (4/4)

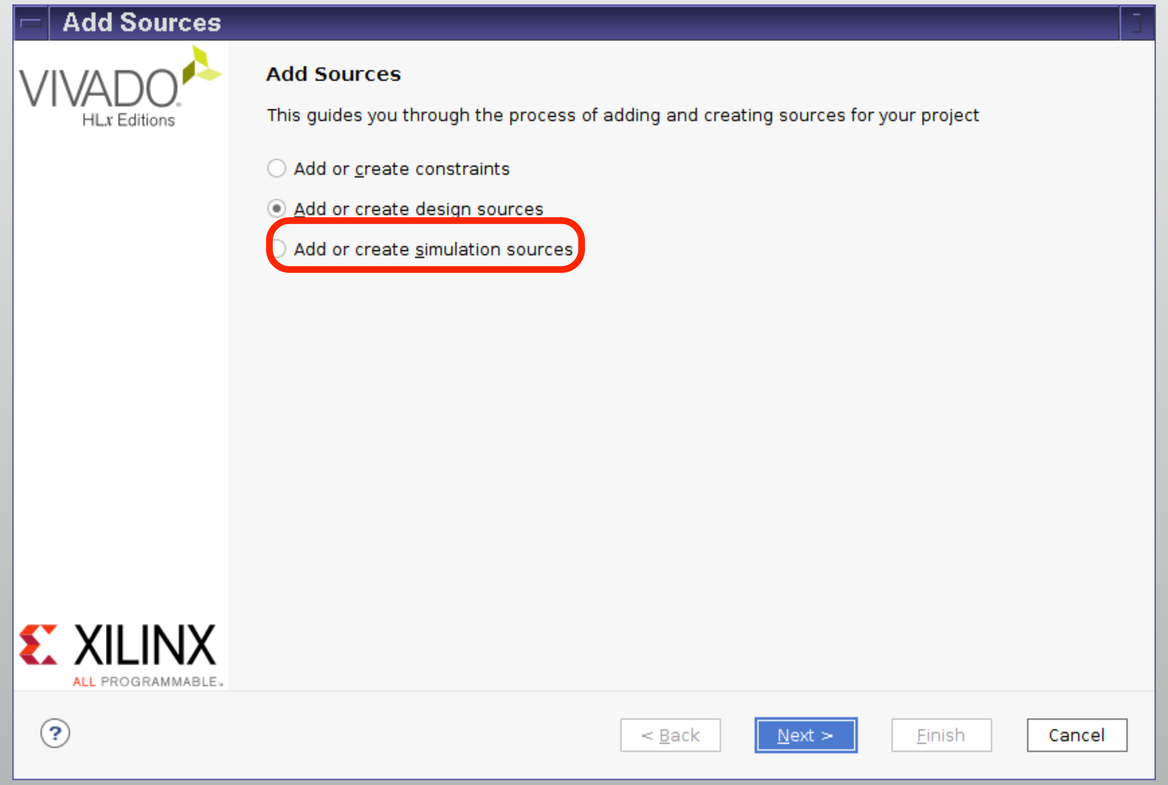
* モジュールのポートどうする？
  * 聞かれるけど気にしない





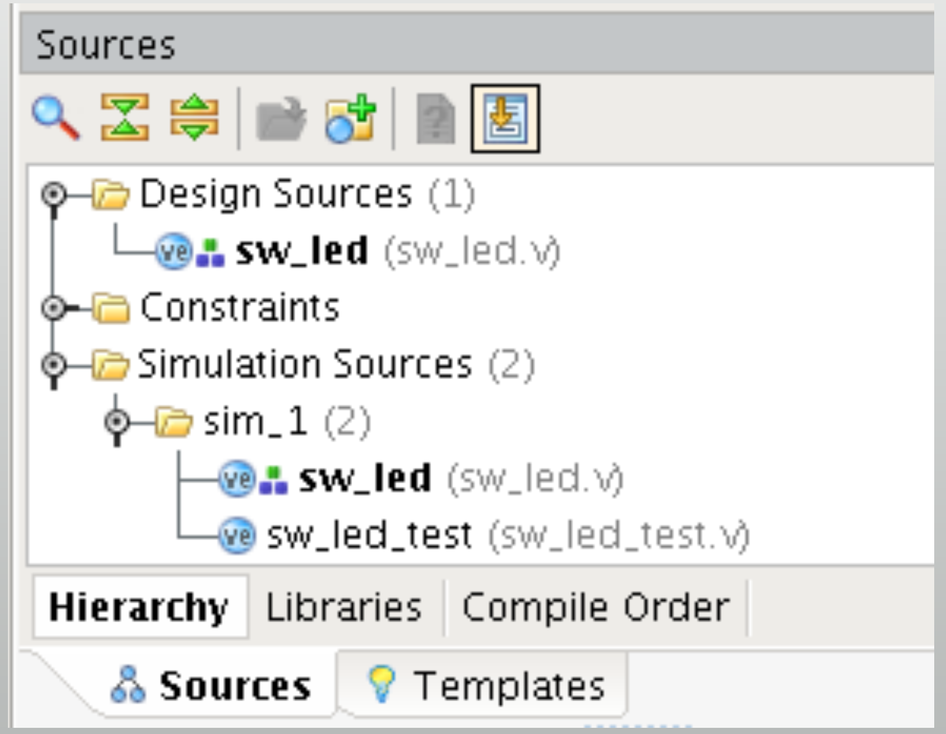
テストベンチも追加

- “Simulation Source” がそれ
  - sw_led_test という名前で
  - 手順は RTL と同じ



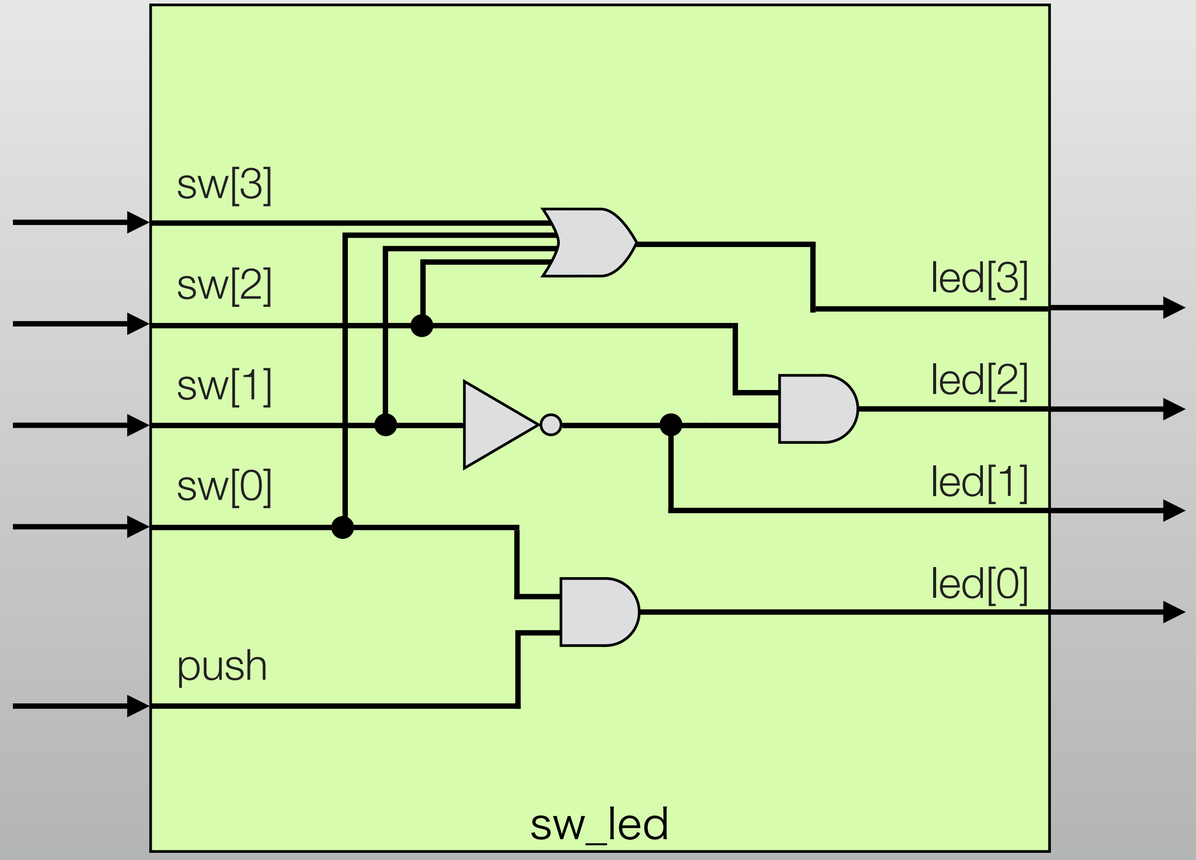
デザイン階層

- Design Sources
  - RTL
- Simulation Sources
  - テストベンチ+RTL
  - 複数のセットを作れる(sim_1)



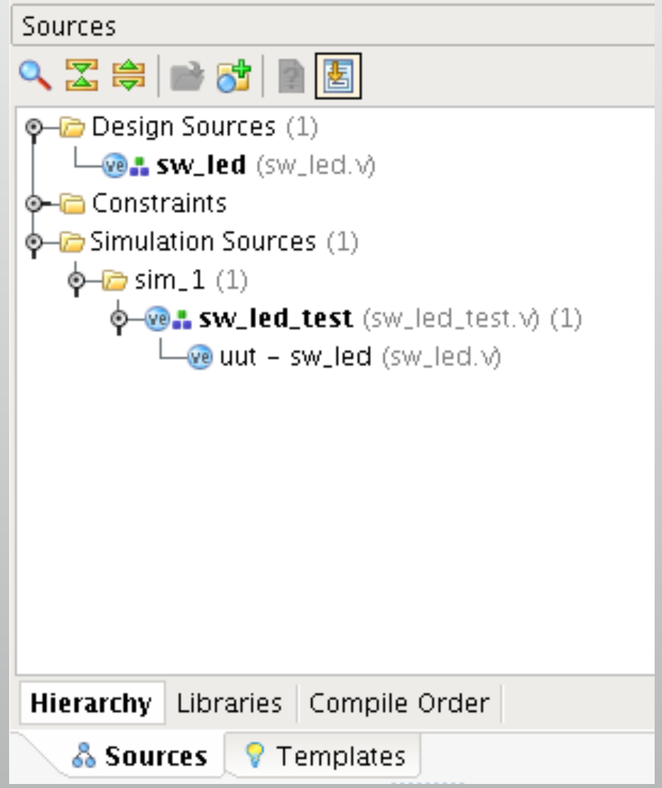
これ書きましょう

* さっきのスライドのと同じ



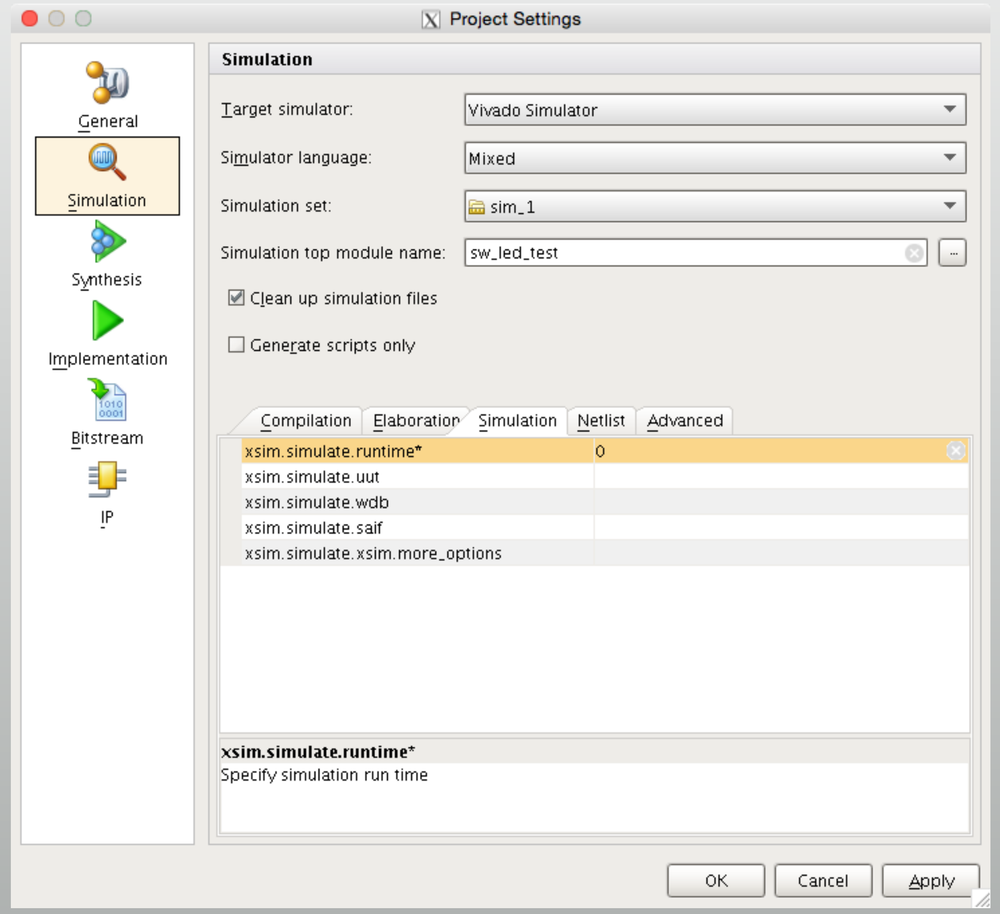
デザイン階層ふたたび

* テストベンチの下にRTLが入る
  * インスタンス名-モジュール名



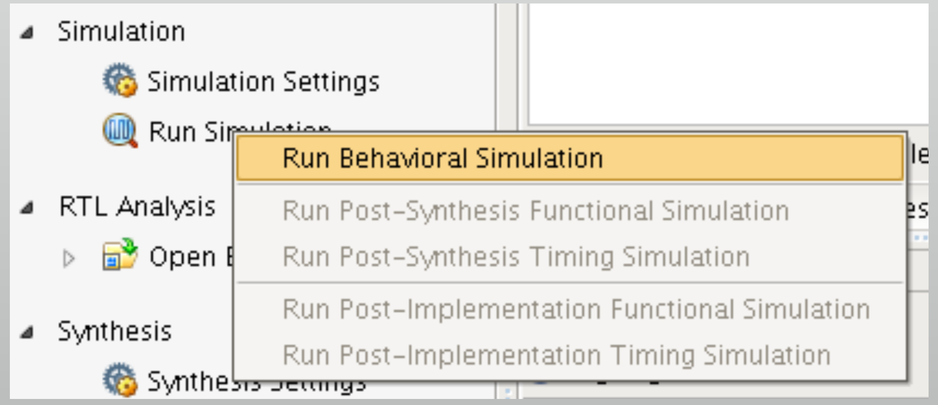
シミュレーションの設定

- Flow navigator から
  - Simulation settings → Simulation で runtime を 0
  - デフォルトは 1000ns

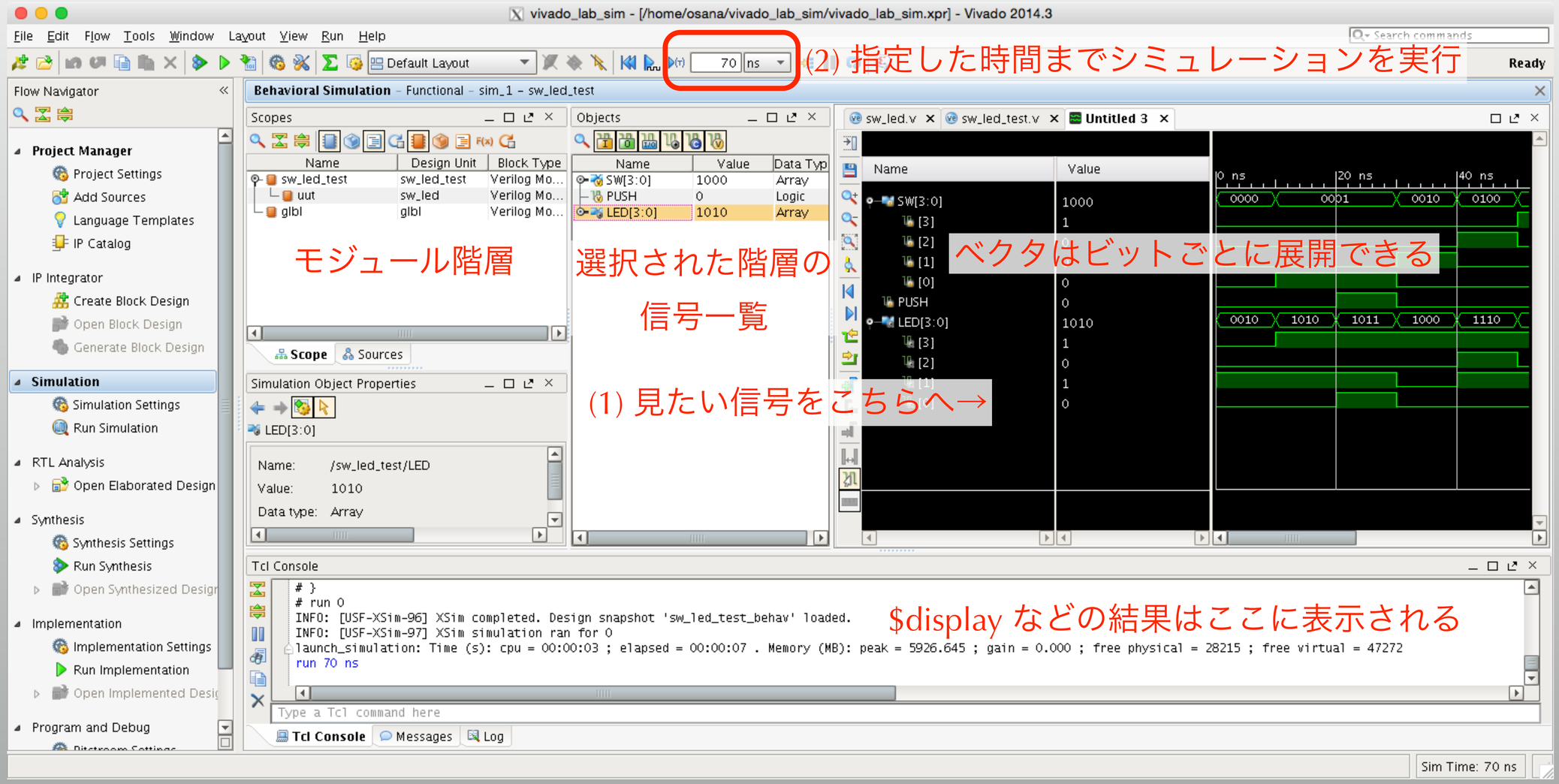


コンパイルを実行

- Run Simulation →
  Run Behavioral Simulation
  - 論理合成後のモデルなどでの
    シミュレーションも可能
  - シミュレーションするための
    コンパイルが実行される

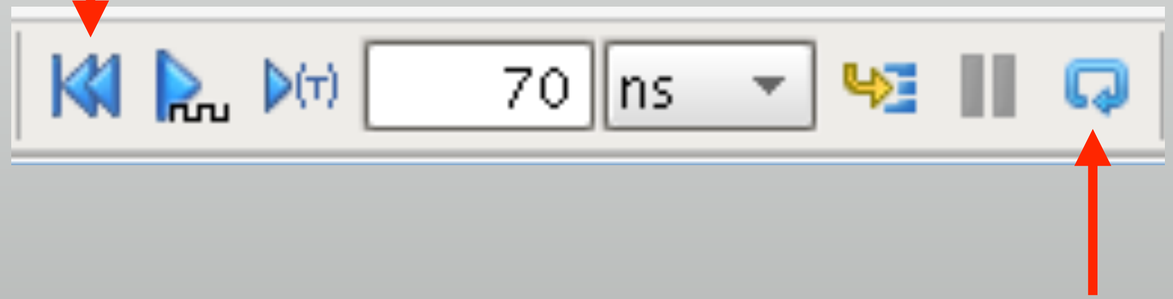


シミュレーション


vivado_lab_sim - [/home/osana/vivado_lab_sim/vivado_lab_sim.xpr] - Vivado 2014.3
(2) 指定した時間までシミュレーションを実行
モジュール階層
選択された階層の
信号一覧
ベクタはビットごとに展開できる
(1) 見たい信号をこちらへ→
$display などの結果はここに表示される

やり直しが必要な場合

波形に信号を追加したら一度リセットが必要



ソースを修正したら再コンパイルが必要

おまけ: 今日のソース

```
`timescale 1ns/1ps

module sw_led_tb ();
  reg [3:0]  SW;
  reg        PUSH;
  wire [3:0] LED;

  sw_led uut (.SW(SW), .LED(LED),
              .PUSH(PUSH));

  initial begin
    $monitor("%t SW: %b, PUSH: %b", $time, SW, PUSH);

        SW <= 4'b0000; PUSH <= 0;
    #10 SW <= 4'b0001;
    #10 PUSH <= 1;
    #10 SW <= { SW[2:0], SW[3] };  PUSH <= 0;
    #10 SW <= { SW[2:0], SW[3] };
    #10 SW <= { SW[2:0], SW[3] };
  end
endmodule
```

```
module sw_led
  (
   input  wire [3:0] SW,
   input  wire       PUSH,
   output wire [3:0] LED
  );

  wire  SW1_ = ~SW[1];
  assign LED[0] = PUSH & SW[0];
  assign LED[1] = SW1_;
  assign LED[2] = SW1_ & SW[2];
  assign LED[3] = |SW;

endmodule
```