MICS実験第一 J2課題

論理回路とVerilog-HDL

八巻隼人

(yamaki@uec.ac.jp)

西10号館-519

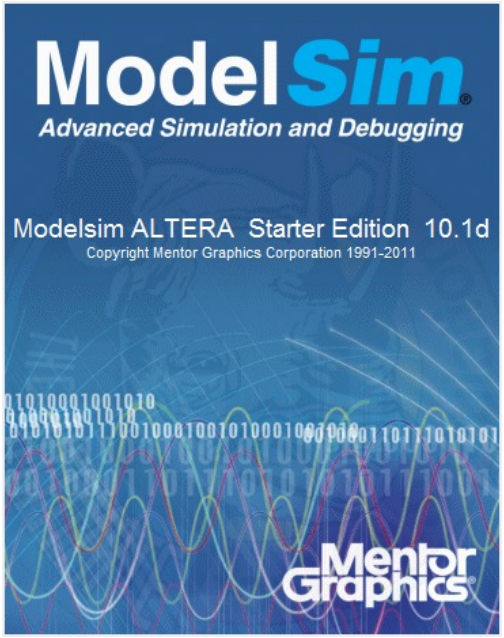
この実験資料について

- ダウンロード用 URL
  - http://www.hpc.is.uec.ac.jp/yamaki_lab/teaching-jp.html
  - 資料と共にsampleファイルが置いてあります.
    授業で出てくるvファイルはここに入っているので参考にしてください. （./~.sc というスクリプトを実行するとiverilogコンパイルから実行まで一気に行ってくれます）

- 佐藤証先生の「J2課題解説スライド」がベース
  - 佐藤証先生のオリジナル→ http://satoh.cs.uec.ac.jp/ja

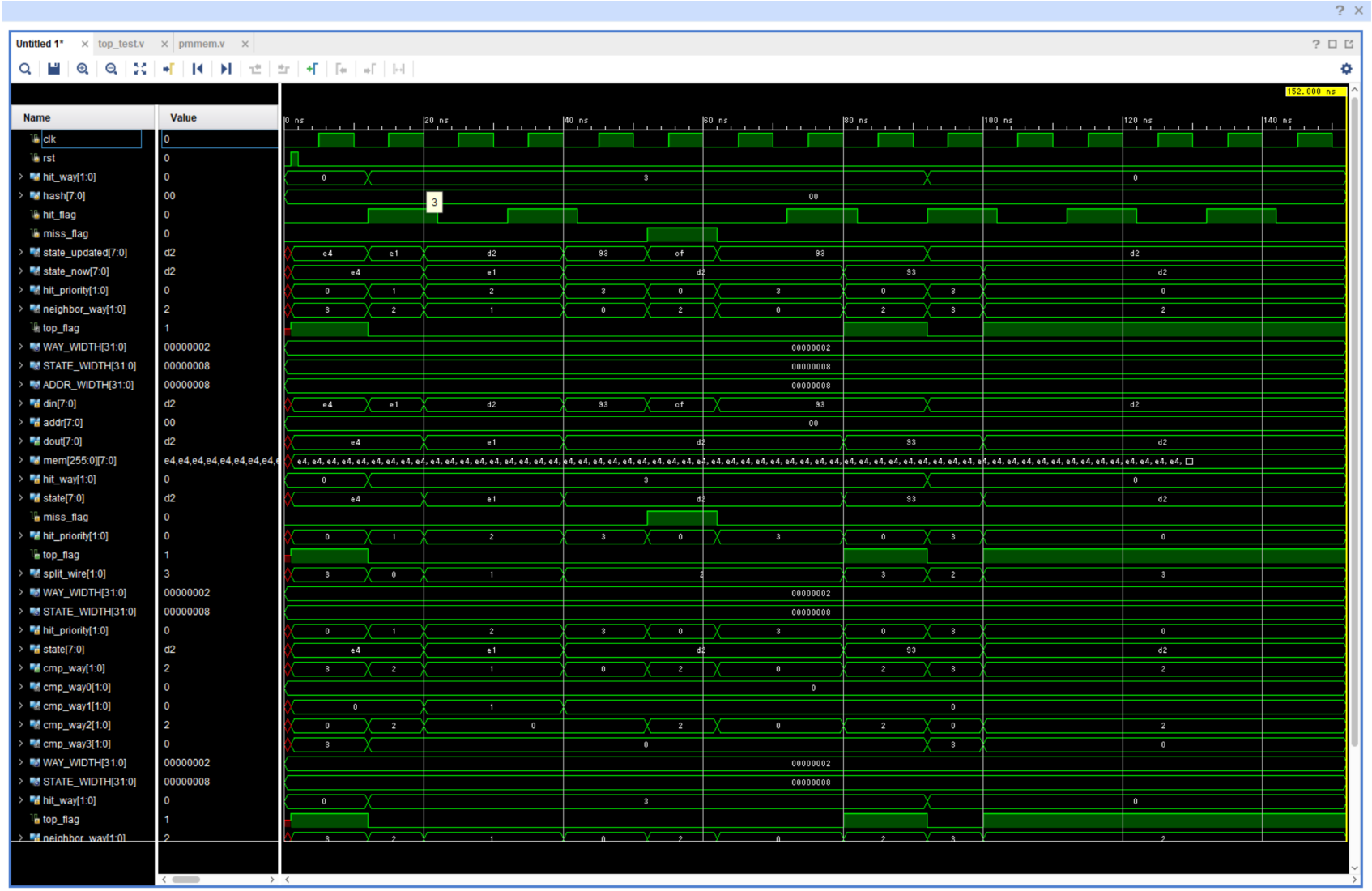
参考書

- 赤い本：
  - Verilog-HDL の勉強用
- 青い本：
  - ツールの使い方の勉強用



J2課題の目的

- 論理回路の特性 / 設計技術を習得する



J2課題の実施方法

・今年は対面実施となります．

・実施内容や課題は本資料に載っています

- それとは別に，問題のヒントを以下のリンクに随時upします．

  http://www.hpc.is.uec.ac.jp/yamaki_lab/teaching-jp.html

- わからないことがあったら対応するので私にメールください
  - 八巻（yamaki@uec.ac.jp）

実験の流れ

- 実験初日（~p43）
  ガイダンス
  簡単な組み合わせ回路の設計
  問題1～3

- 実験２日目（~p60）
  順序回路の設計
  問題4

- 実験３日目（~p71）
  加減算回路の設計
  問題5～7

レポート

記載内容と採点基準

- この資料の問題１～７を全て行い，その内容に考察を加えレポートとする
- 問題１～７全て回答できていることが修了の条件
- 更に出来る人は問題８を解くと加点
- 例年，問題７まで解けずに単位を落とす人が結構います．
  わからない部分は恐れず質問してください．

提出方法

- レポートPDF ファイルを WebClass 上で提出．
- 受付期間内なら何度でも再提出可．

受付期間

- 授業終了から３週間後の23:00まで
- 前半組： 7/30(水) の 23:00まで
- 後半組： 8/11(月) の 23:00まで

目次

- 本実験の内容とその意義（論理回路とハードウェア設計）
- verilogの使い方
- 簡単な組み合わせ回路（問題１～３）
- 順序回路（問題４）
- 加減算回路（問題５～７）

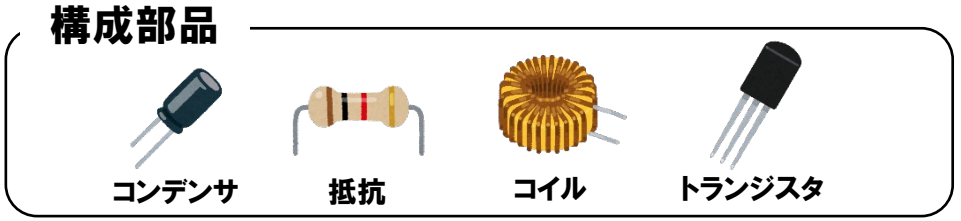
本実験の内容とその意義

論理回路とハードウェア設計

論理回路とは

- デジタル回路の数学的モデル
  - **デジタル回路：** １と０のデジタル信号のみを用いて様々な計算をする回路
    （一般的には電圧のHighとLowを１と０にデジタル化）

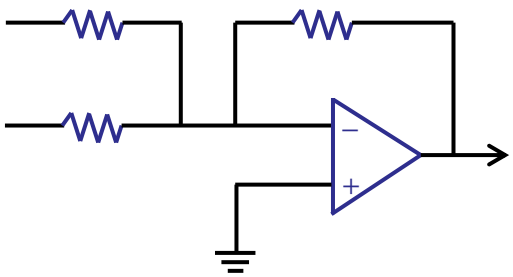
  - **アナログ回路：**連続するアナログ信号を用いて様々な計算をする回路



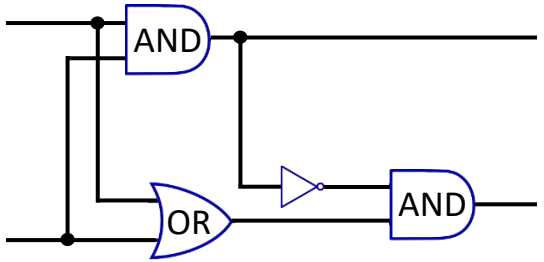
デジタル回路の利点

・ある機能を実現したい場合，アナログ/デジタル回路どちらでも可だが

＜例えば加算器＞



アナログ回路



デジタル回路

＋ 回路構成自体は簡潔にできる ↔ − トランジスタで構成するため複雑

− 集積化はしにくい ↔ ＋ 集積化できる！→ IC (Integrated Circuit)

− 汎用化しにくい（専用回路） ↔ ＋ 汎用化できる！→ CPU

ハードウェア？ソフトウェア？

- 文脈によりそれぞれが指す意味は多少異なる

一般的には

- ハードウェア：物理的実体を持つ機械部品．
  例．CPU，メモリ，マウス，キーボード，ディスプレイ
- ソフトウェア：物理的実体を持たない構成要素．主にプログラムのこと．
  例．OS，アプリケーションプログラム，ドライバ

処理において使われる時（ハードウェア処理，ソフトウェア処理）

- ハードウェア：専用の論理回路による処理
- ソフトウェア：CPUによる（プログラムでの）処理

本実験の文脈はこちら

ハードウェアとソフトウェア 何が違う？

～ソフトウェア (CPU) の場合～

- CPUの強みは何でも (プログラムで記述すれば) 処理できる ＝ **汎用的**
  ⇆　何かの処理に特化しているわけではない

例えば…

- 次のような計算をCPUで実行する場合

$$x = (a \times 10 + 20) + (b \times 100 - 50)$$

- CPUでは以下のように実行に 5 命令（ 5 サイクル）を要する
  乗算命令 → 加算命令 → 乗算命令 → 減算命令 → 加算命令

ハードウェアとソフトウェア 何が違う？

～ハードウェアの場合～

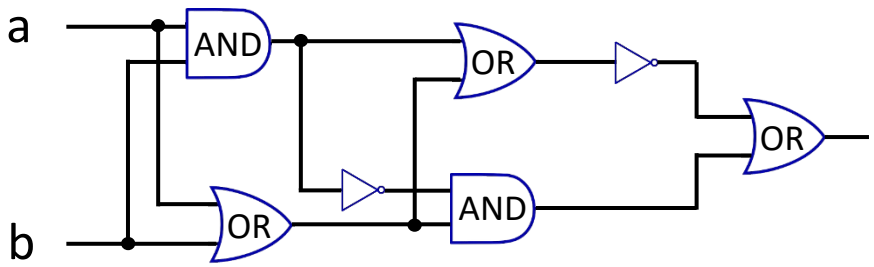
- その処理専用の回路（ASIC）を作れば高速に処理できる

例えば…

- 先ほどの例↓

$$x = (a \times 10 + 20) + (b \times 100 - 50)$$

- ハードウェアでは 1 サイクル で計算が完了する



x = (a×10 + 20) + (b×100−50) を計算する回路
（この回路は適当…）

Tips

ソート処理を考えてみるとより差がわかりやすい.

ソート処理はCPUの場合,値の大小で入れ替えを繰り返すので結構なサイクルを要する

一方でハードウェアの場合,ソート回路に値を入れると1サイクルで結果が得られる

超高速

ハードウェアとソフトウェア 何が違う？

～まとめると～

- ハードウェア処理は超高速
- また，（あまりにも複雑な処理でなければ）CPUやメモリを実装するより専用論理回路を実装したほうが小規模

小型とか超高速を重視する処理において有用

例えば…

- **動画像処理**
- **組み込み機器系（監視カメラ，IoTセンサとか）の処理**
- **ルータのパケット処理**

ハードウェア設計（この実験）

- ハードウェア（専用論理回路）はどうやって作る？

① まず仕様を満たす論理回路の設計図を作成する
  - 仕様から状態遷移図・真理値表を描き，入出力の論理式を求める
  - 導いた論理式をもとに回路図を作成

論理回路設計の講義でやった

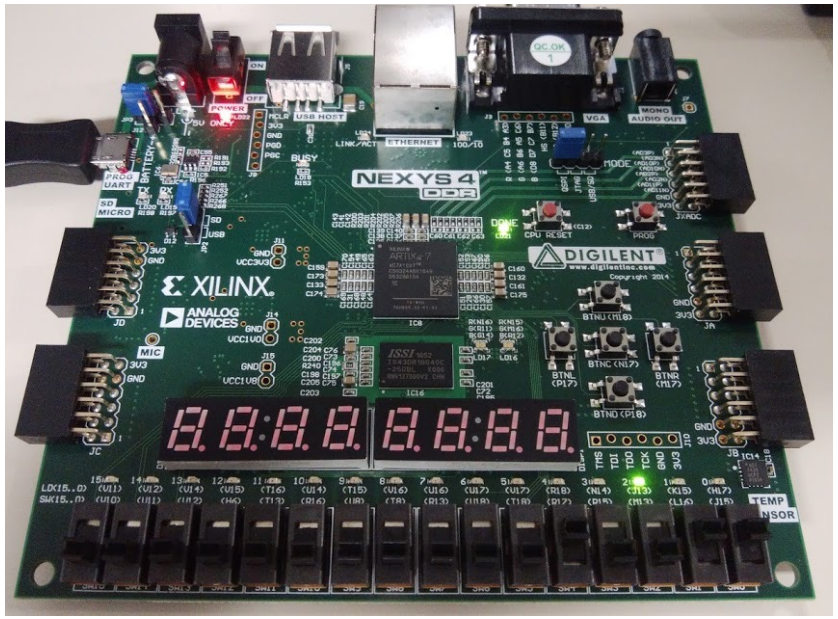
② テストベンチを用いて設計した回路の動作を検証
  - 回路作成には膨大なお金がかかるので失敗はできない
  - シミュレーションにより回路が所望の動作をするか徹底的に検証

ハードウェア記述言語　verilog

③ 論理回路（実物）を作成
膨大なお金をかけて外注する

ハードウェア設計の③について

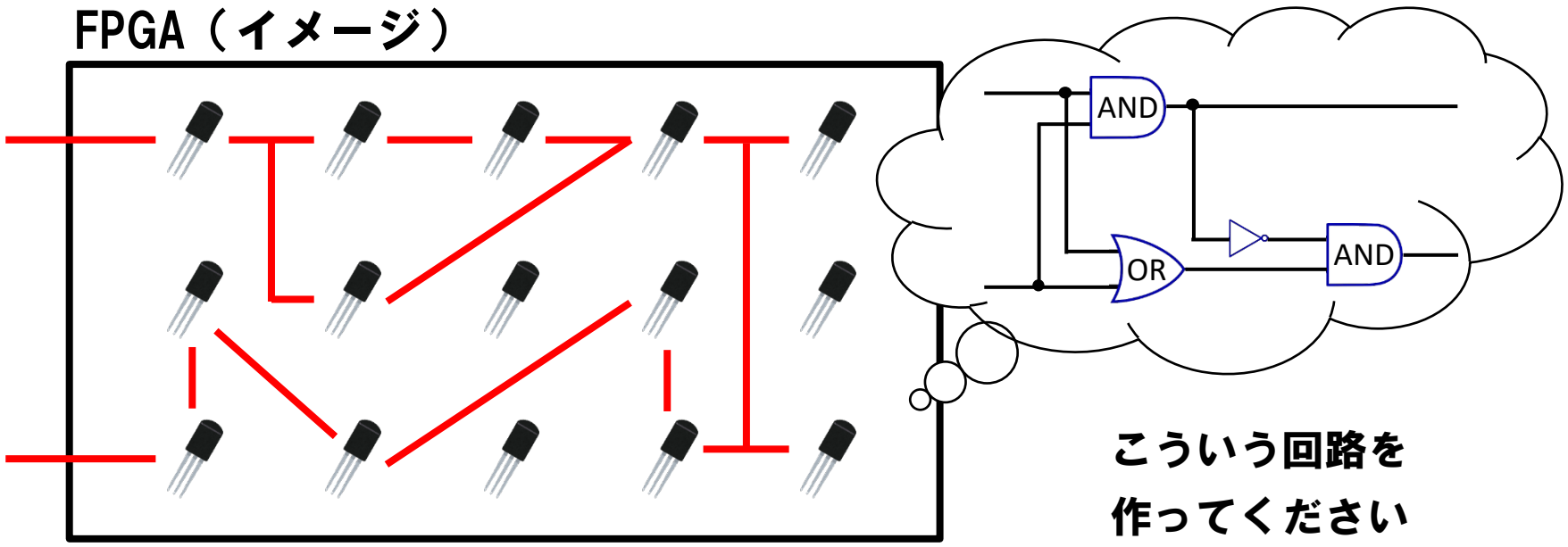
- 実物の回路を作ってもらうにはとにかくお金がかかる（数百万円～）
- そこで最近は **FPGA (Field Programmable Gate Array)** が出てきた
- FPGAは…
  - 論理回路を（外注せずとも）自作できるハードウェア
  - FPGAのお値段は数万円～
  - 回路を作る費用はタダ
  - しかも設計した論理回路を自分でPC上で焼ける
  - 何度でも焼き直し可能という夢のようなデバイス



FPGA（のイメージ）

Field Programmable Gate Array

- ゲート（トランジスタ）がアレイ（格子）状に配置されている
- Verilog で作成した回路となるようにFPGAのゲートを配線
  → 所望の論理回路を作成できる（配線しなおすことも可能）



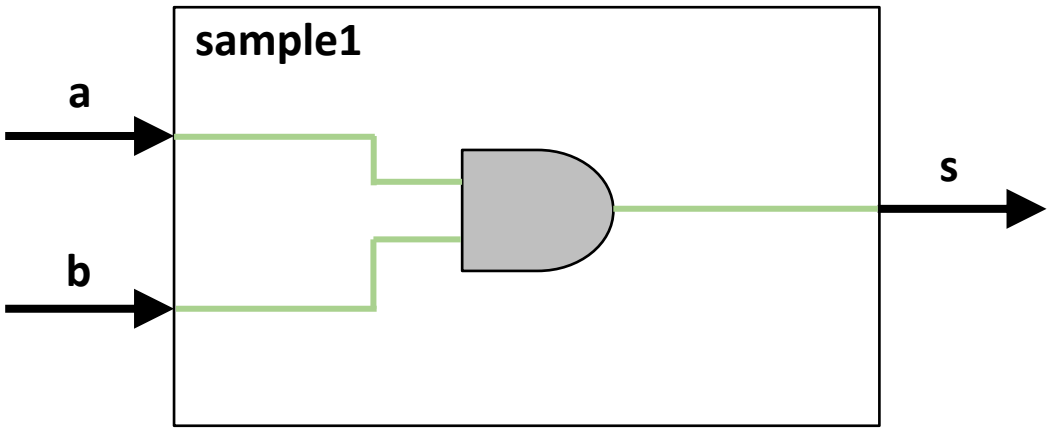
ハードウェア記述言語

verilog

Verilog-HDL の例：AND回路のモジュール（sample1.v）

- まずは以下のverilogプログラムを自分で作ってみましょう
  コマンドラインで $ vi sample1.v または $ emacs sample1.vとし以下を作成

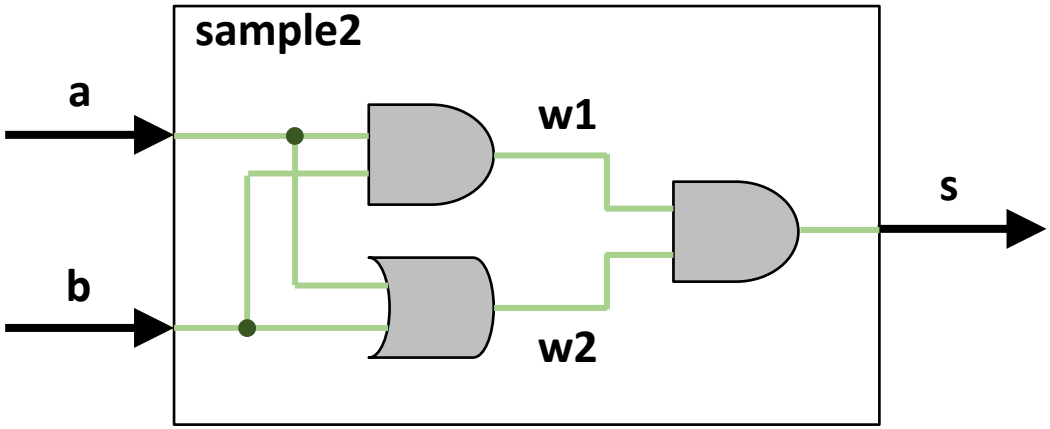
```
module      sample1(s, a, b);      /* モジュール名(入出力端子)
   input    a, b;                  /* 入力端子a, bの宣言
   output   s;                     /* 出力端子sの宣言
   assign   s = a & b;             /* s に a (and) b を接続
endmodule
```



Verilog-HDL の例： wireの使い方（sample2.v）

- モジュール内で必要になる配線はwireで宣言
- assign文は出力や配線に接続するための構文（後述するregはassignを使わない）

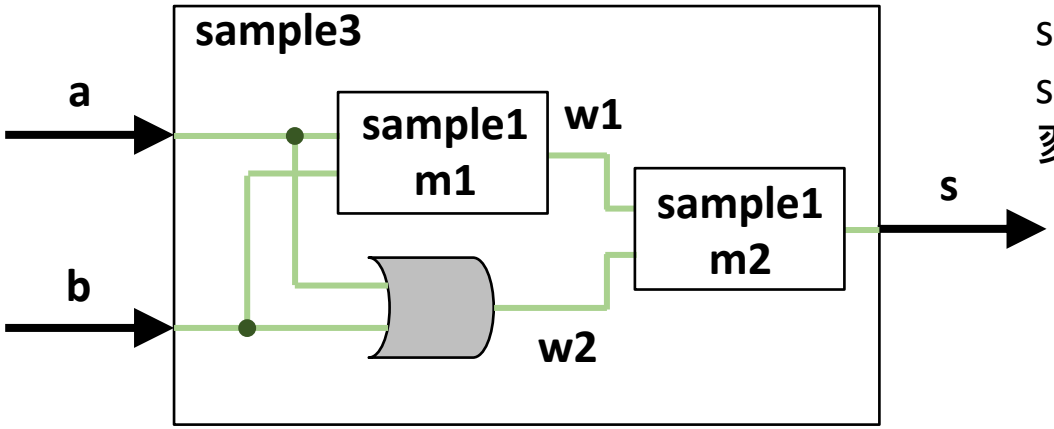
```
module      sample2(s, a, b);  /* モジュール名(入出力端子)
   input    a, b;                 /* 入力端子a, bの宣言
   output   s;                    /* 出力端子sの宣言
   wire     w1, w2;               /* 配線w1, w2の宣言
   assign   w1 = a & b;           /* w1 に a (and) b を接続
   assign   w2 = a | b;           /* w2 に a (or) b を接続
   assign   s = w1 & w2;          /* s に w1 (and) w2 を接続
endmodule
```



Verilog-HDL の例：モジュールの入れ子（sample3.v）

- モジュール内で別モジュールを呼び出すことも可能

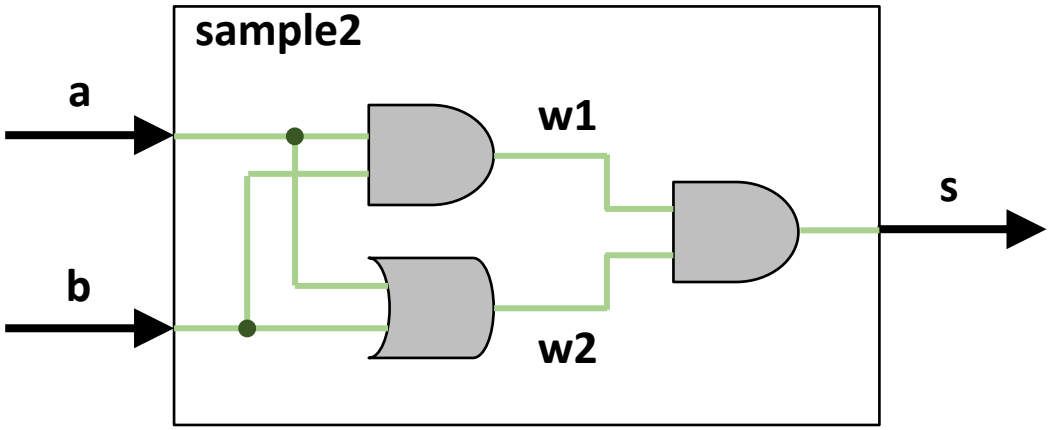
```
module      sample3(s, a, b);   /* モジュール名(入出力端子)
   input    a, b;                  /* 入力端子a, bの宣言
   output   s;                     /* 出力端子sの宣言
   wire     w1, w2;                /* 配線w1, w2の宣言
   sample1 m1(w1, a, b);           /* sample1モジュールを呼び出し
   sample1 m2(s, w1, w2);             引数の順番はsample1.vと対応
   assign   w2 = a | b;            /* w2 に a (or) b を接続
endmodule
```



sample2のand回路を
sample1モジュールに
変えてみた

モジュールの実装→動作確認（テストベンチ）

- 以上で簡単な回路ならもう記述できるようになりました
- モジュールが完成したら，その動作確認が必要
- 以下の回路なら，aとbにテスト信号を流し，正しいsが得られるか確認
  a, bが(0,0), (0,1), (1,0), (1,1) の時，sが 0, 0, 0, 1 になるかどうか
- 作ったモジュールに手を加えて動作確認するのはよろしくない
  そこで動作確認用のテストベンチモジュールを作る



Verilog-HDL の例：sample1のテストベンチ（sample1_test.v）

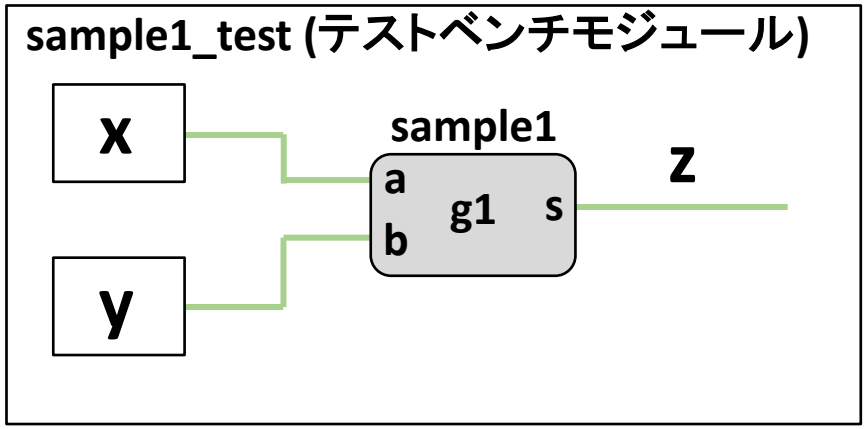
テストベンチの場合，
入出力端子は作らない．
入力側をregで
（詳しくは後述）
出力側をwireで宣言．
間違えやすいので注意

モニタで指定した信号に
（今回ならx,y,z,stime）
変化があった場合に
ディスプレイ出力

C言語でいう
Printf

```
module      sample1_test;
   wire     z;
   reg      x, y;
   sample1 g1(z, x, y);
   initial                          /* イベントの生成
      begin
         $monitor(" %b %b %b", x, y, z, $stime);
         $display(" x y z                  time");
               x=0; y=0;      /* 0サイクル目
         #50   y=1;           /* 50サイクル目
         #50   x=1; y=0;      /* 100サイクル目
         #50   y=1;           /* 150サイクル目
         #50   $finish;       /* 200サイクル目
      end
endmodule
```

テストイベント
今回は50サイクル毎に
入力値を変化



iverilogを使った動作確認

- コンパイル

  iverilog -o 実行ファイル名 -s トップモジュール名 コンパイルしたい.vファイル群

```
[        @blue99 sample]$ iverilog -o sample1 -s sample1_test sample1_test.v sample1.v
```

→　エラーが出なければsample1というファイルが作成される

- シミュレーションの実行

  vvp 実行ファイル名　または　実行ファイルを直接指定 ./sample1

```
[        @blue99 sample]$ vvp sample1
a b s       time
0 0 0          0
0 1 0         50
1 0 0        100
1 1 1        150
```

$displayの内容

$monitorの内容

→　自分が意図した通りの結果となっているか確認

一致検出回路（eq.v）

```
module   eq(s, a, b); /* 一致検出回路 */
  input  a, b;
  output s;
  wire   na, nb, s1, s2;
  assign na = ~a, nb = ~b;
  assign s1 = a & b, s2 = na & nb;
  assign s  = s1 | s2;
endmodule
```

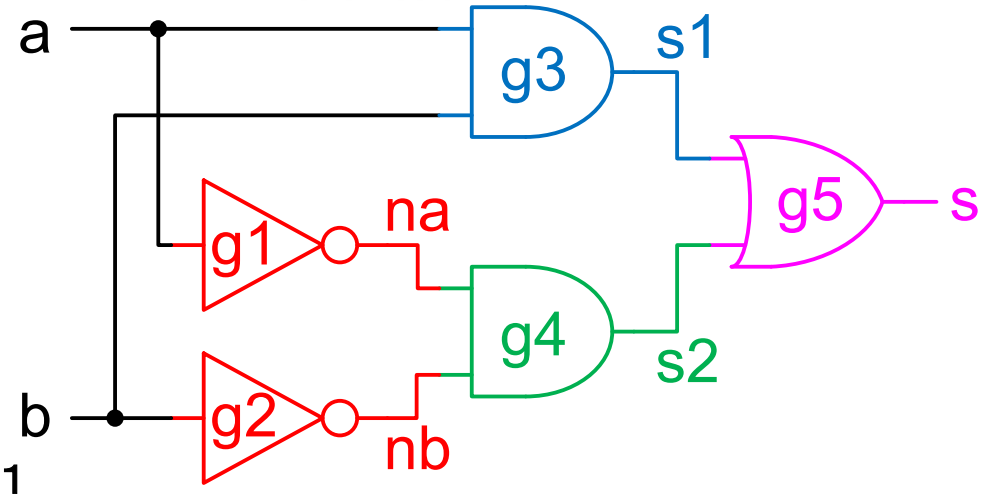
各記号の意味
- ~: NOT演算
- &: AND演算
- |: OR演算

真理値表

a	b	s
0	0	1
0	1	0
1	0	0
1	1	1

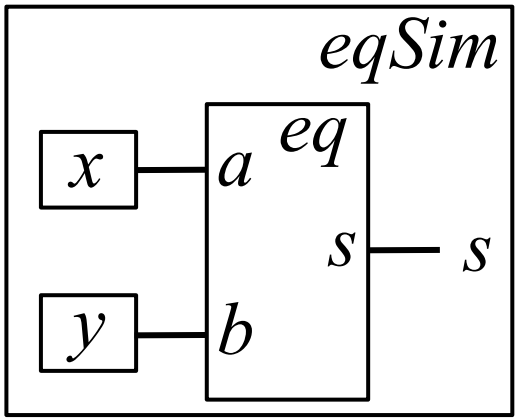
入力a,bが一致した時に出力sが１

回路図

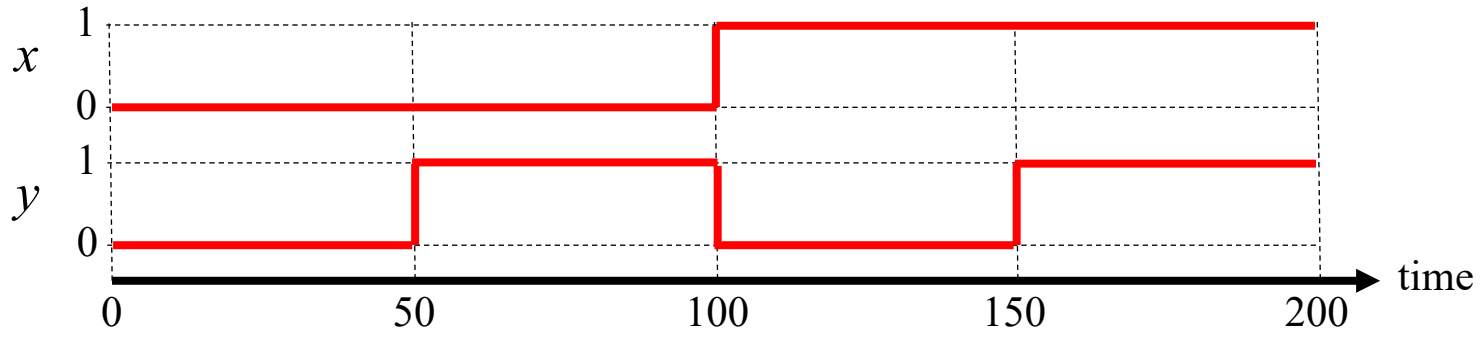


一致検出回路のテストベンチ（eqSim.v）

```
module   eqSim;          /* 一致検出回路の */
  wire   s;              /* シミュレーション */
  reg    x, y;           ← レジスタ宣言
  eq     g1(s, x, y);    ← モジュール呼び出し
  initial                ← 制御記述
    begin
      $monitor(" %b %b %b", x, y, s, $stime);
      $display(" x y s                time");
            x=0; y=0;  ┐
      #50   y=1;       │
      #50   x=1; y=0;  ├ ← テスト入力
      #50   y=1;       │
      #50   $finish;   ┘
    end                  ← 遅延
endmodule
```



[シミュレーション用回路]



[テスト入力の時間変化]

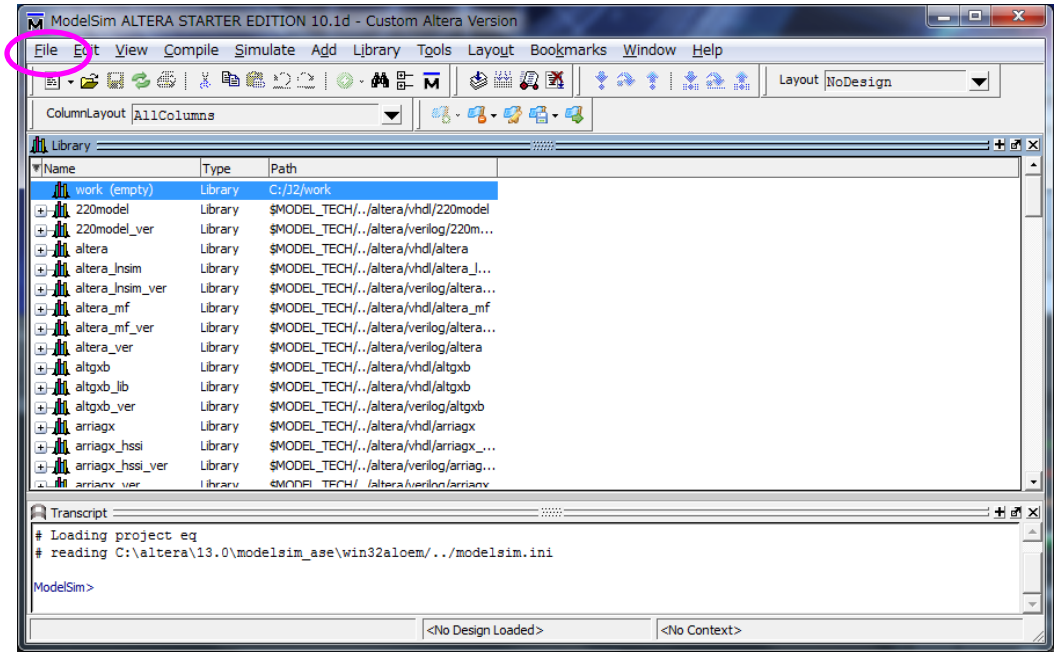
ModelSimの使い方

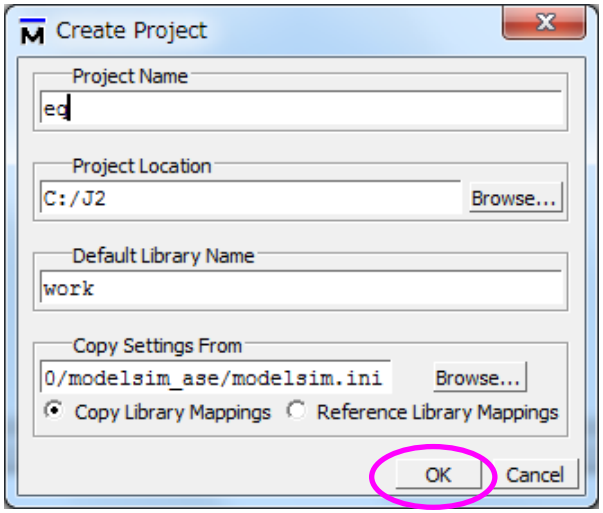
すみません，今年度はCEDの環境変更に伴いModelSimが使えません

代わりに gtkwave というツールを使います
使い方は Webclass の実験ページに書いてありますので
そちらを参照してください

ModelSimの起動とプロジェクト作成

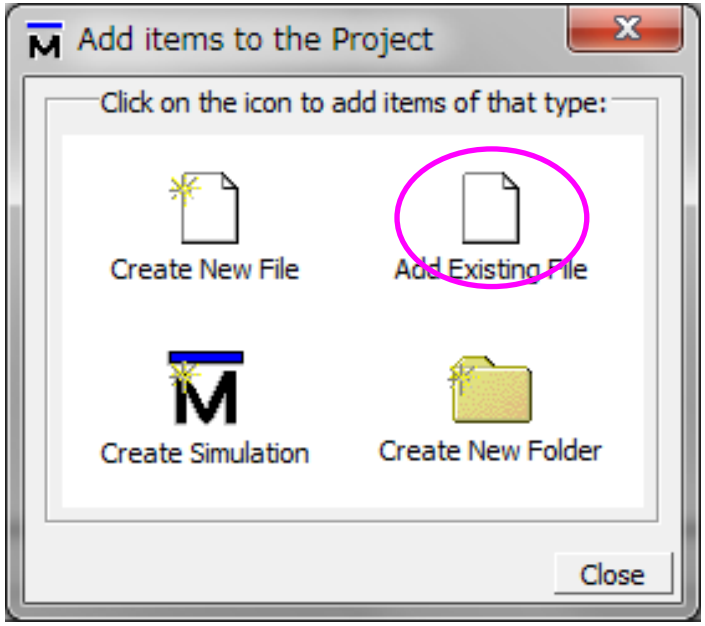
1. コンソールで”mkdir ~/mics_J2”と入力
2. コンソールで”vsim”（今年は”vsim2”？）と入力
3. File→New→Proeject... で“Create Project”ウィンドウが開く
4. Project Name： eq
5. Project Location： */path_to_homedir/*mics_J2

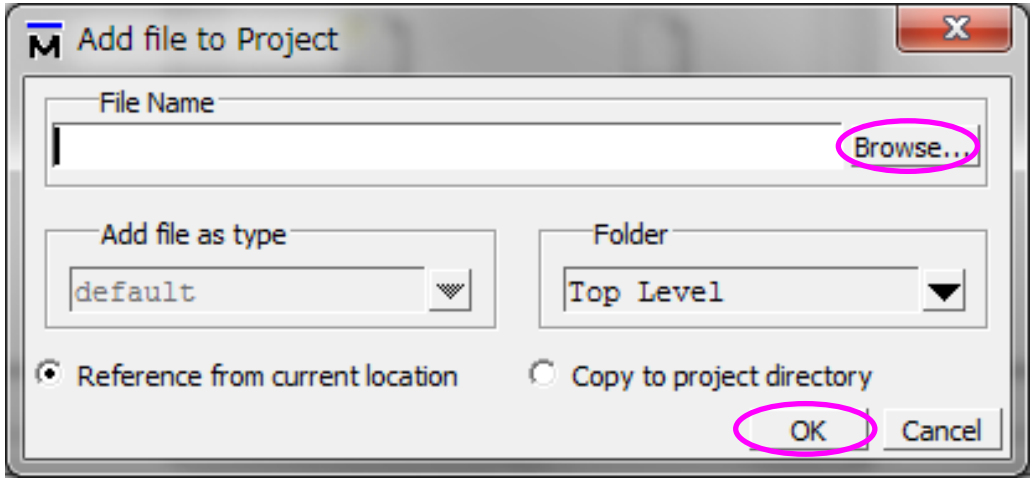




Verilog-HDLファイルの追加

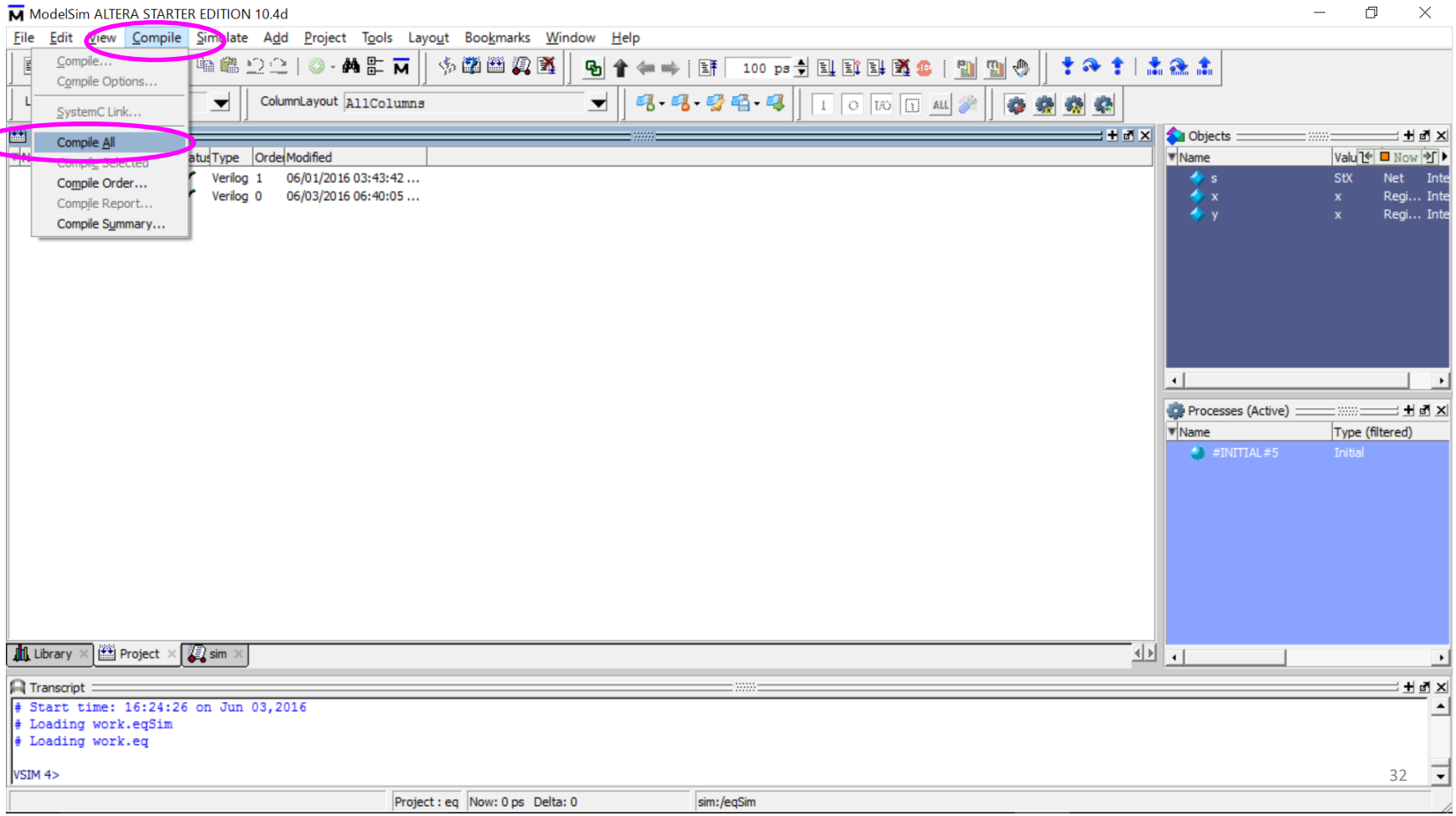
1. “Add items to the Project”ウィンドウ： Add Existing File
   またはメインウィンドウから Project→Add to Project→Existing File…
2. “Add file to Project”ウィンドウ： Browse
3. “Select files to add to project”ウィンドウ：
4. eq.v と eqSim.v を選択して開く.
5. “Add file to Project”ウィンドウ： OK
6. “Add items to the Project”ウィンドウ： Close





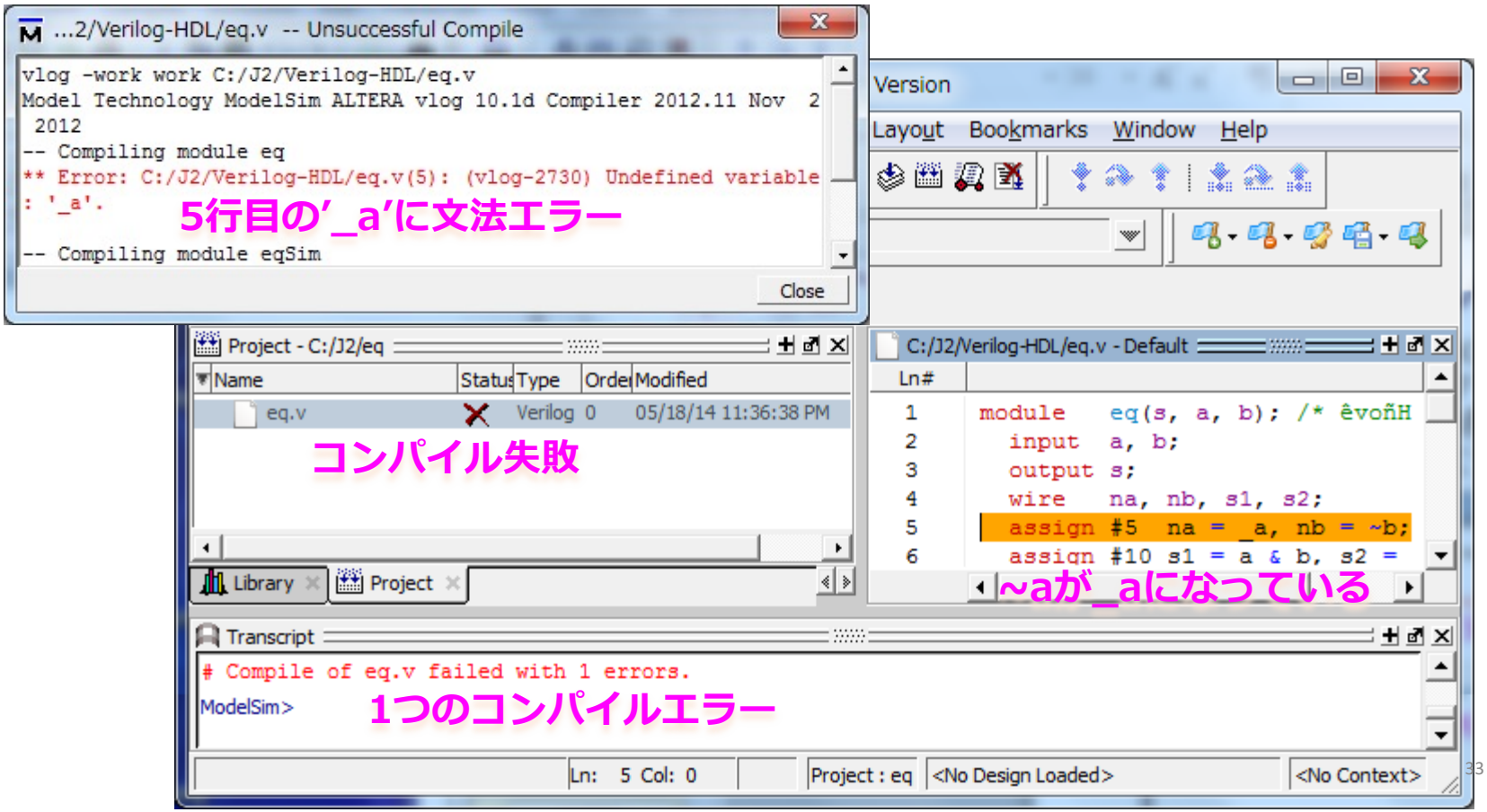
コンパイル

- メインウィンドウ：Compile→Compile All
- Verilog-HDLファイルのStatusが？からレに変わり、“Compile of eqSim.v was successful.” “Compile of eq.v was successful” と出る.



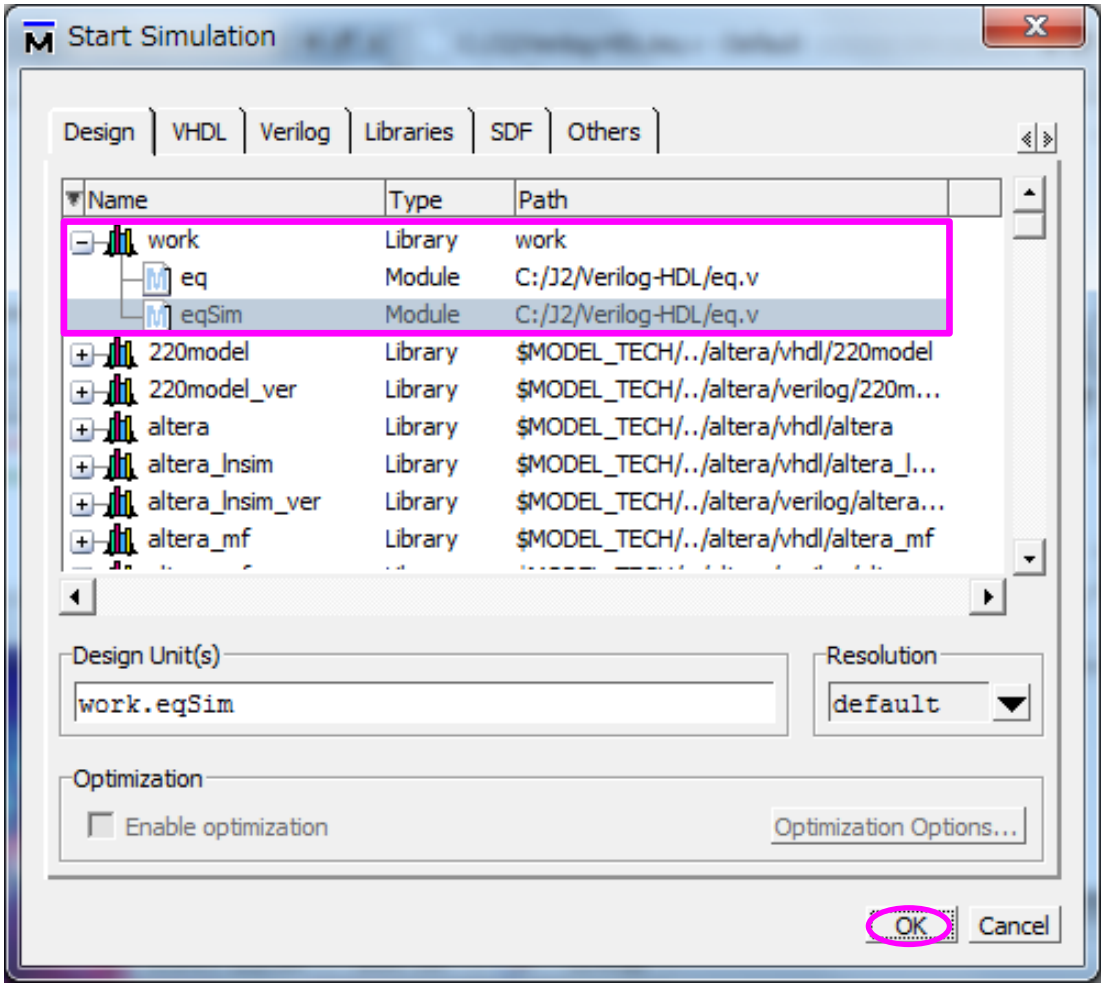
コンパイルエラー

- エラーがあるとTranscriptウィンドウに赤く表示されるので，そこをダブルクリックするとエラーの詳細が表示される
- さらに“** Error: “の赤い行をクリックするとソースコードウィンドウが開いてエラーの行がオレンジで表示される



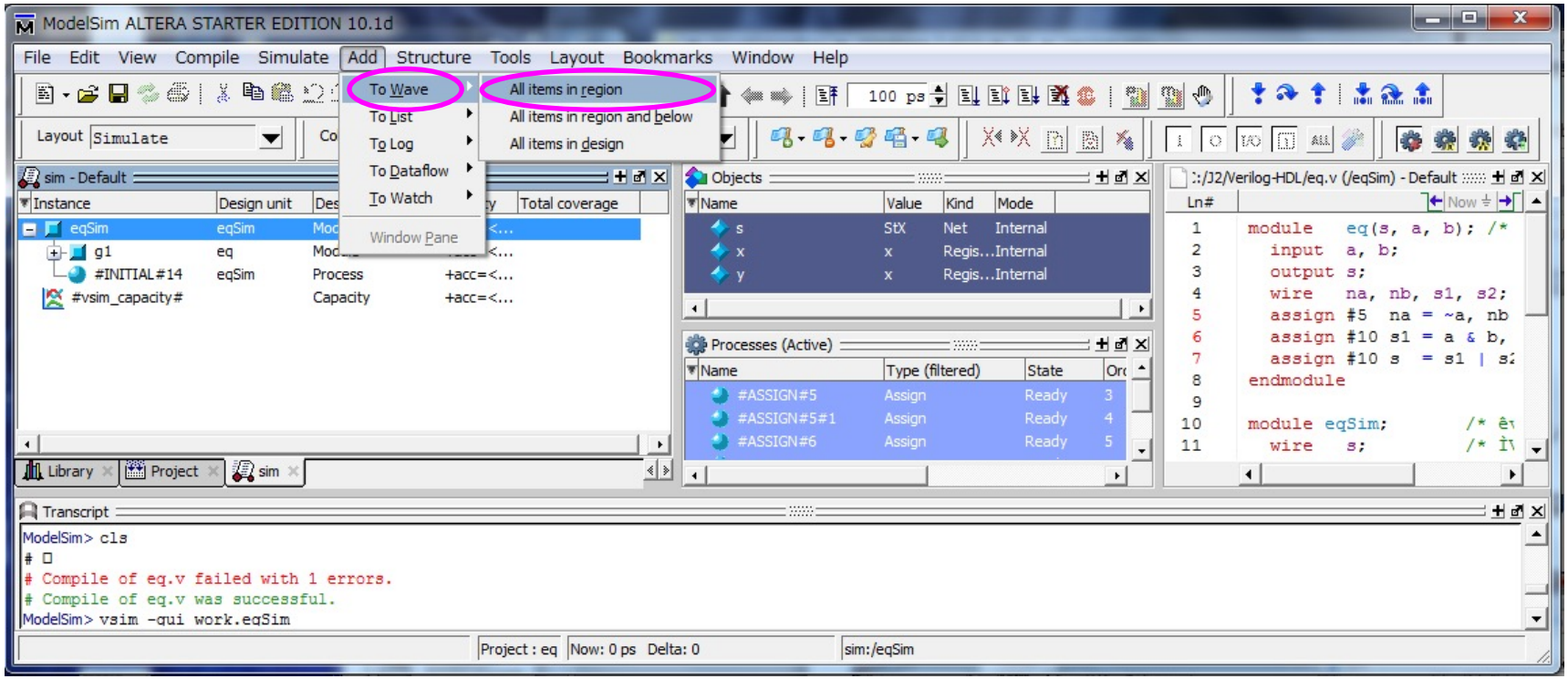
シミュレータ起動

- Simulate→Start Simulation
- Start Simulationウィンドウでwork/eqSimを選択



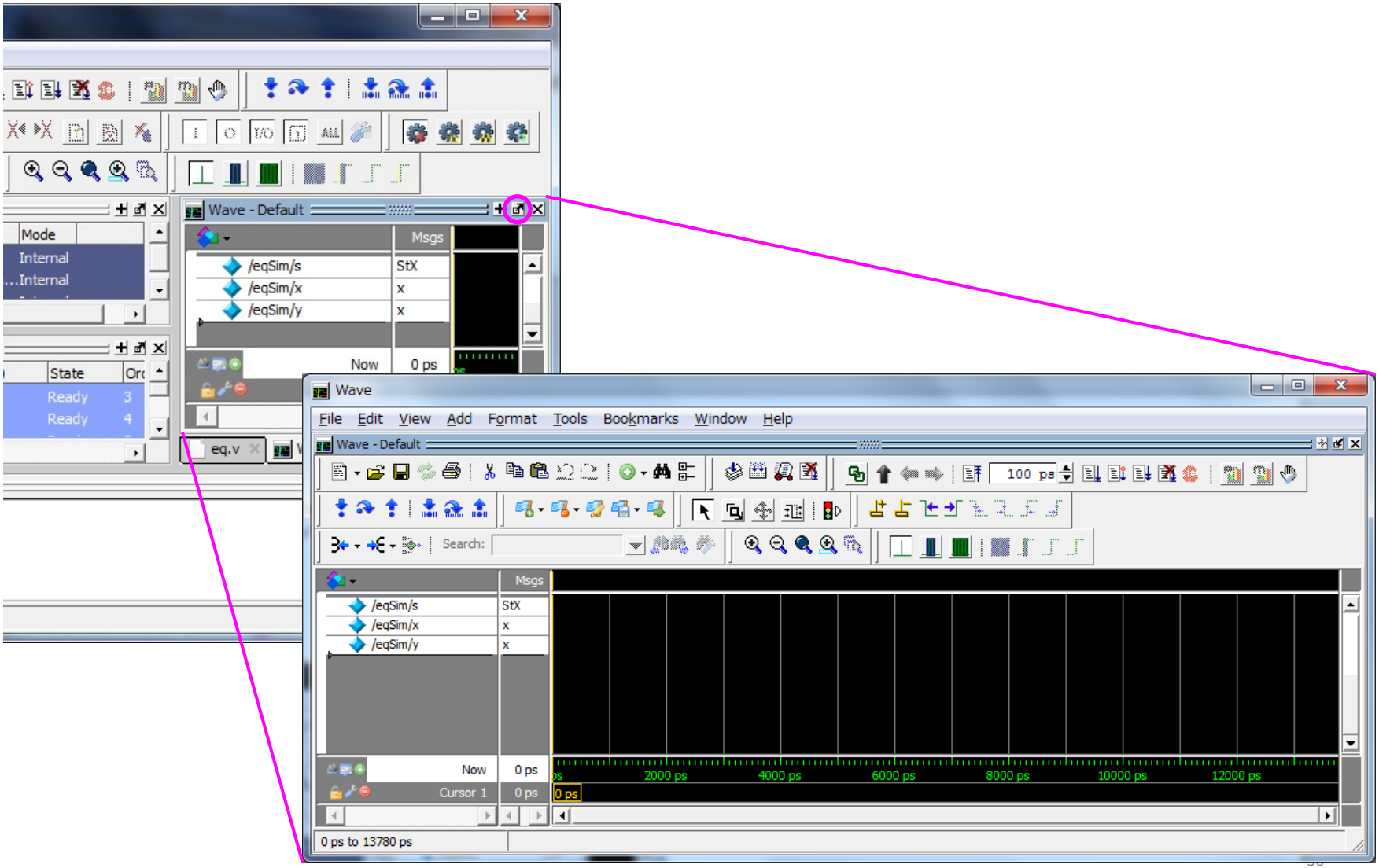
シミュレーション設定

- シミュレーションウィンドウが開く
- Add→add to wave→all items in regionで波形ウィンドウが開く
- “All items in region and below”と”All items in design”は下位レベルやソースの全ての信号線が表示される



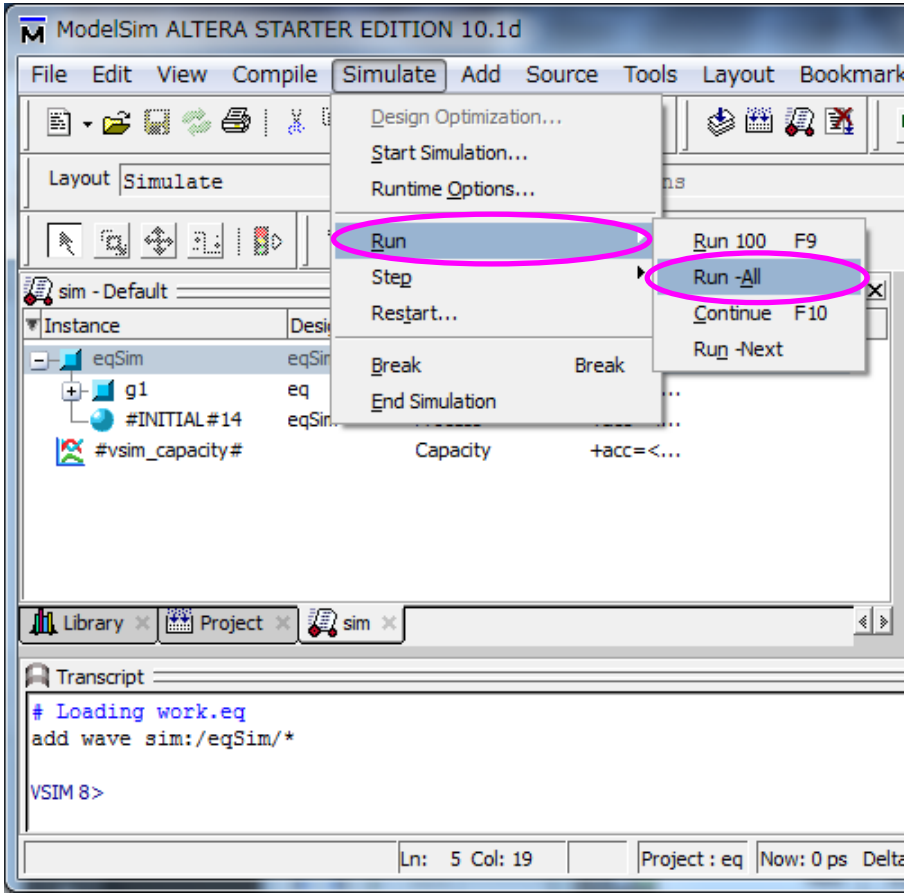
シミュレーション設定

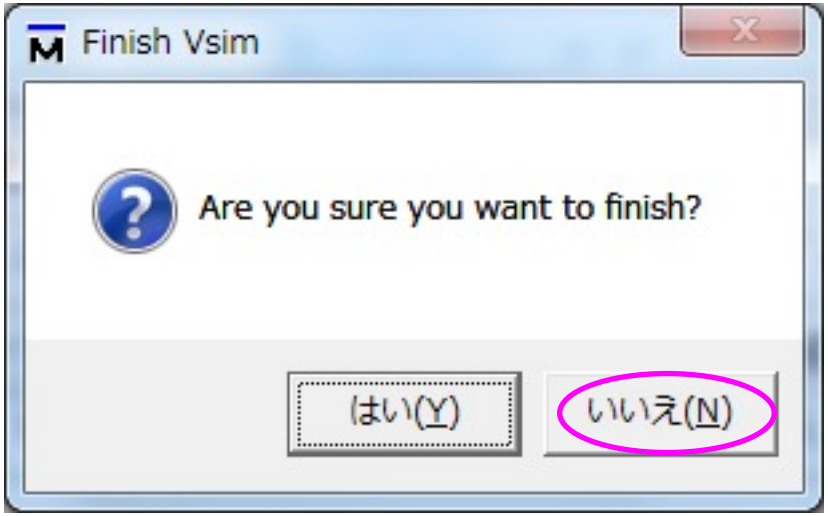
- ウィンドウが小さいので  ボタンを押して拡大



シミュレーション実行

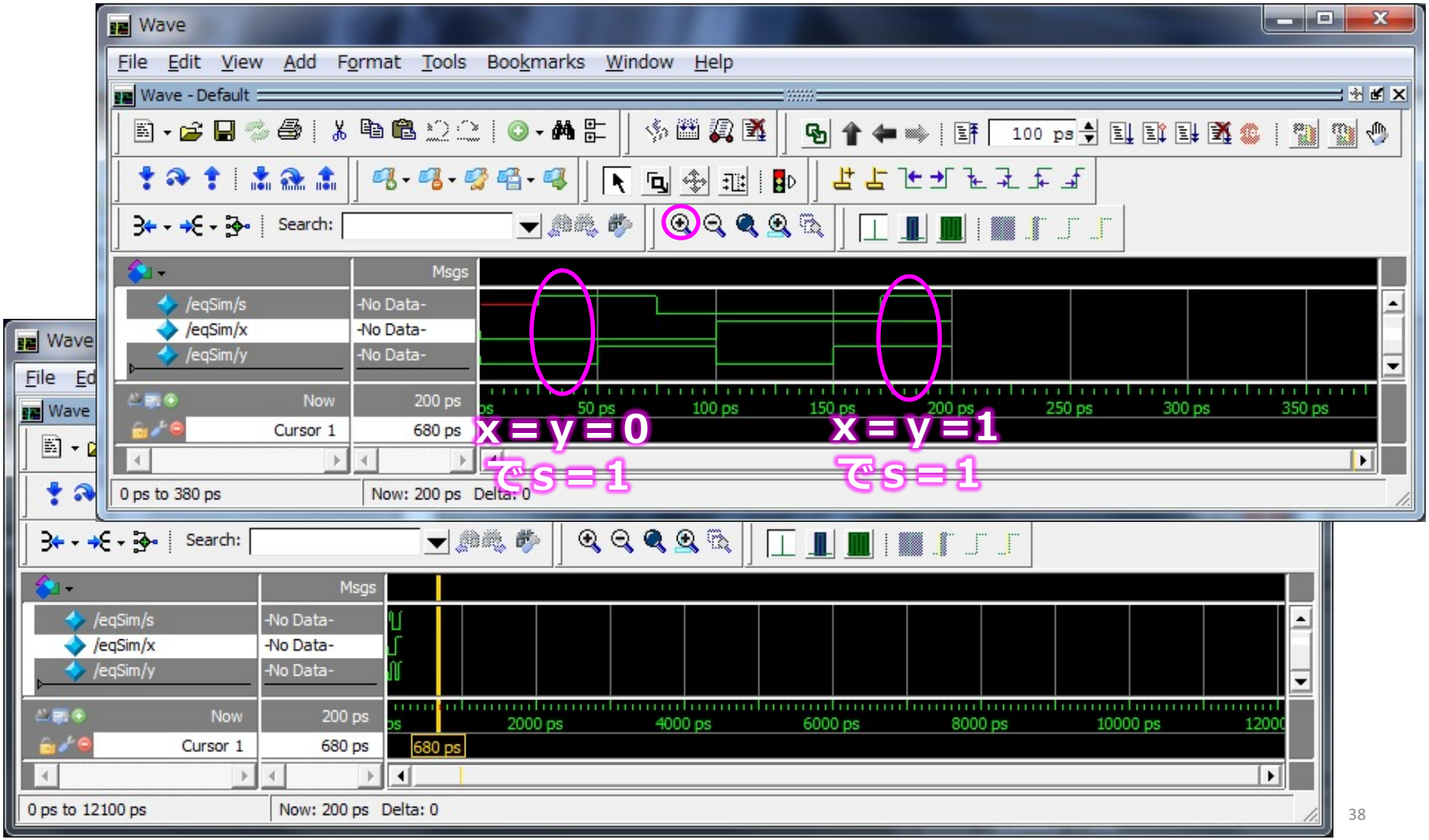
- メインウィンドウで，Simulate→Run→All
- Finsh Vsimウィンドウで「いいえ(N)」を選択
  - 「はい(Y)」を選択するとModelSimが終了してしまうので注意





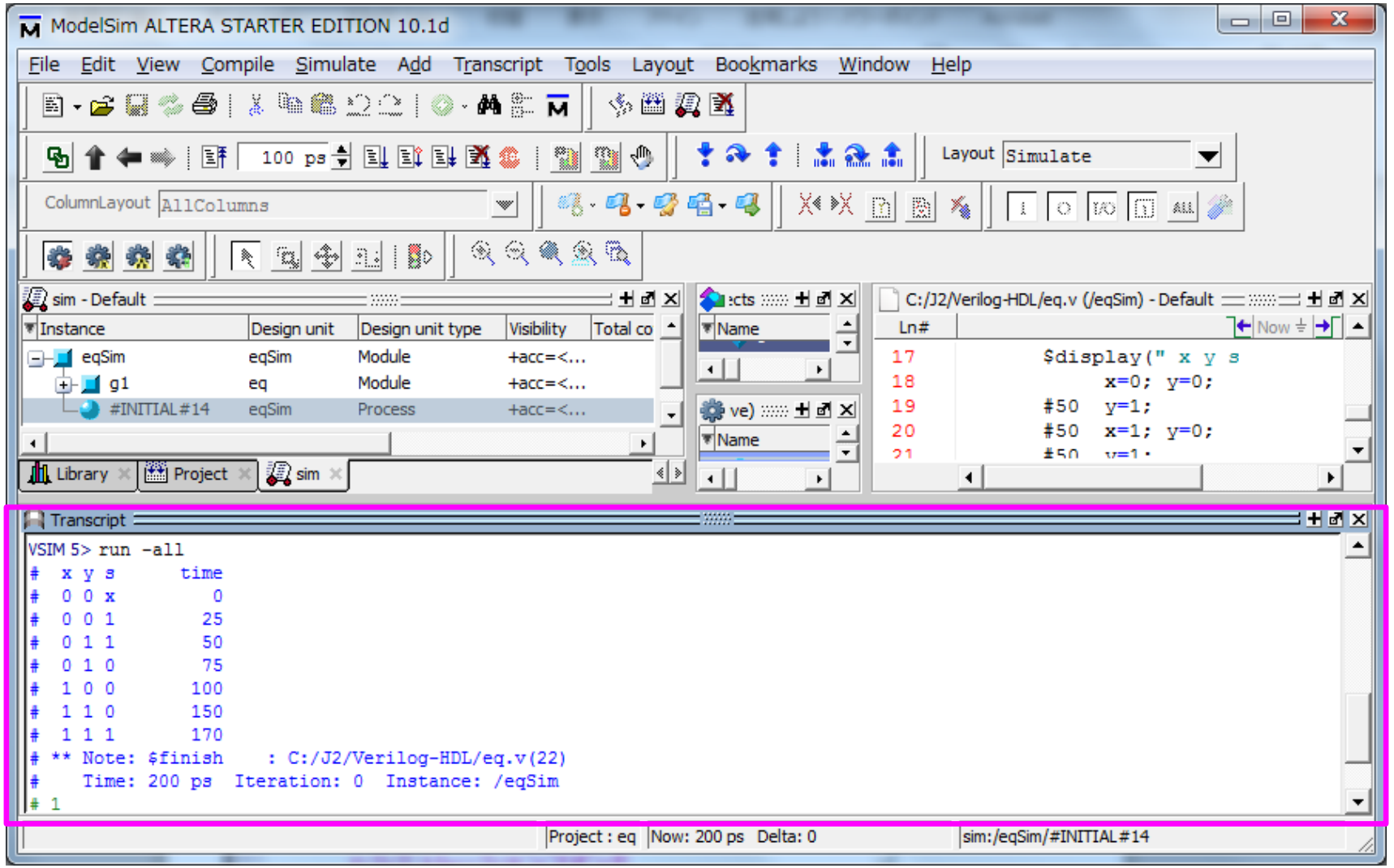
シミュレーション結果表示

- Waveウィンドウに結果が表示される
- 画面を虫眼鏡で拡大して表示



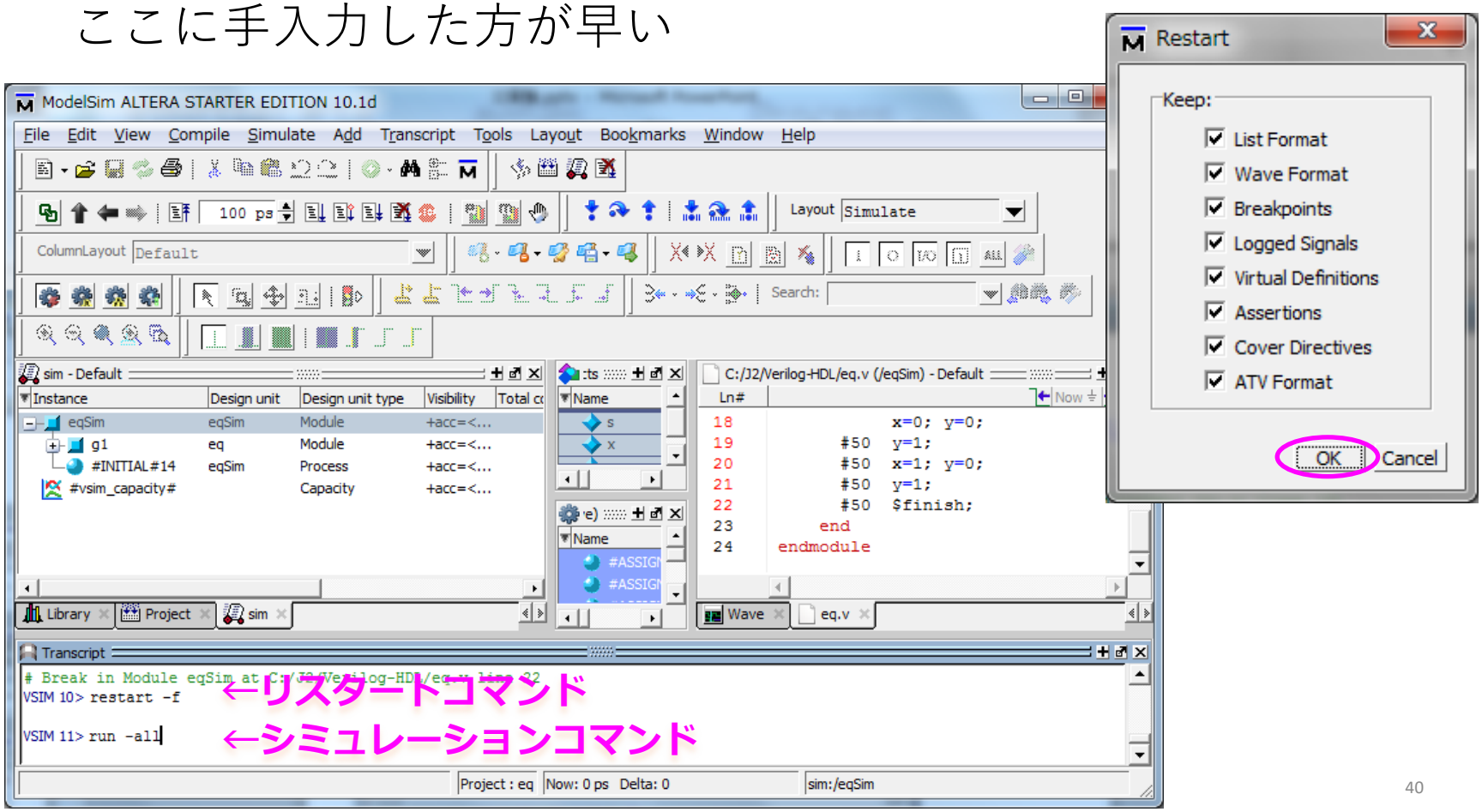
シミュレーション結果表示

- シミュレーションモジュールに$monitorコマンドを記述しているのでTranscriptウィンドウにテキストでも表示される



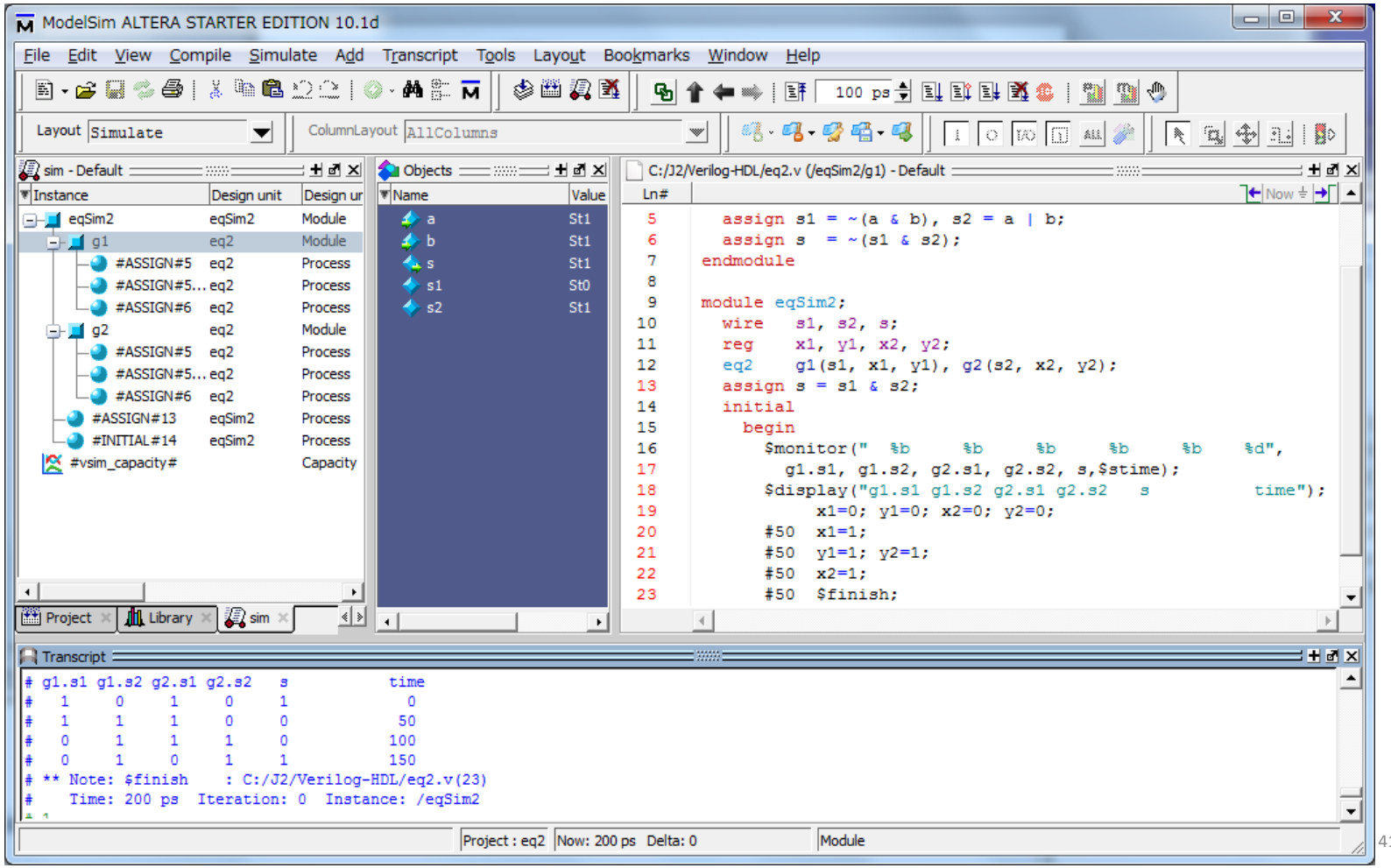
再シミュレーション

- メインウィンドウで Simulate→Restart
- Restartウィンドウが出るのでOKをクリック
- 「シミュレーションの実行」に戻って再シミュレーション
- 全てのコマンドはTranscriptウィンドウで実行されているので，ここに手入力した方が早い



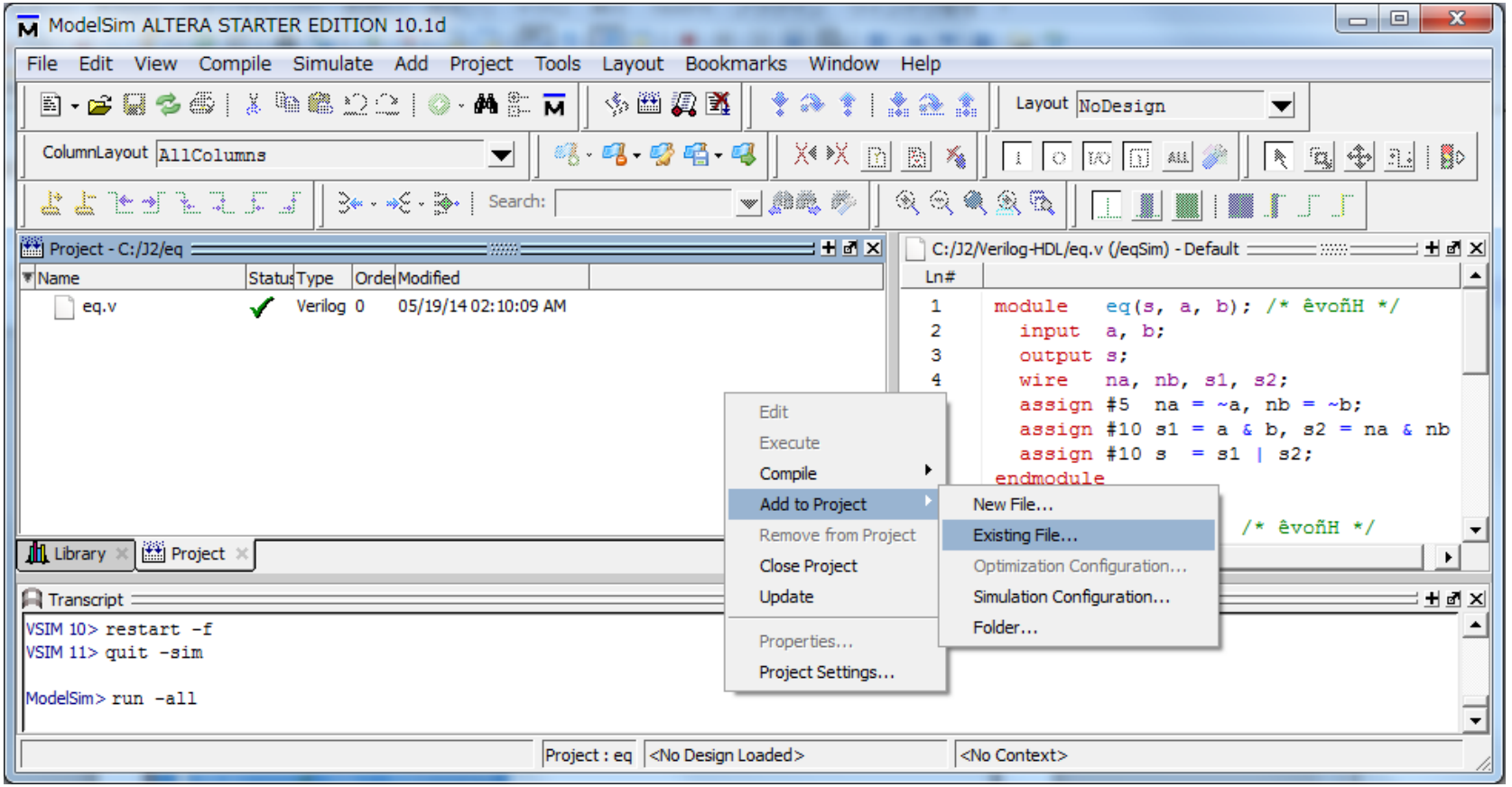
内部信号の表示

- Simウィンドウでモジュールを指定し,内部信号を表示
- $monitor関数では “g1.s1”のようにモジュール内の信号やさらに”.”でつないでモジュール内のモジュールを指定



別回路のシミュレーション

- Simulate→End Simulation
- Projectウィンドウで右クリックしてポップアップメニューを出し，Add to Project→Existing File...
- eq2.v と eqSim2.v を追加



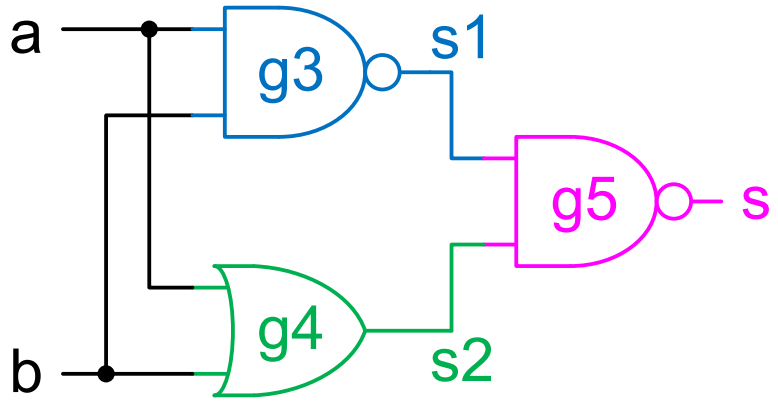
別回路のシミュレーション

```
module   eq2(s, a, b);
  input  a, b;
  output s;
  wire   s1, s2;
  assign s1 = ~(a & b), s2 = a | b;
  assign s  = ~(s1 & s2);
endmodule
```

真理値表

A	B	A=B
0	0	1
0	1	0
1	0	0
1	1	1

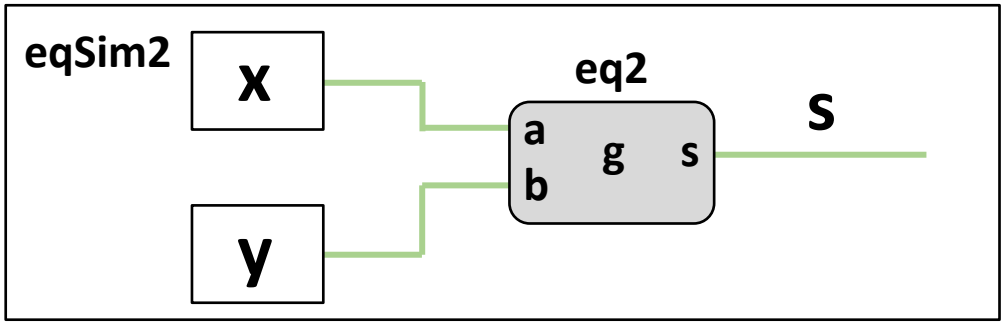
回路図



別回路のシミュレーション

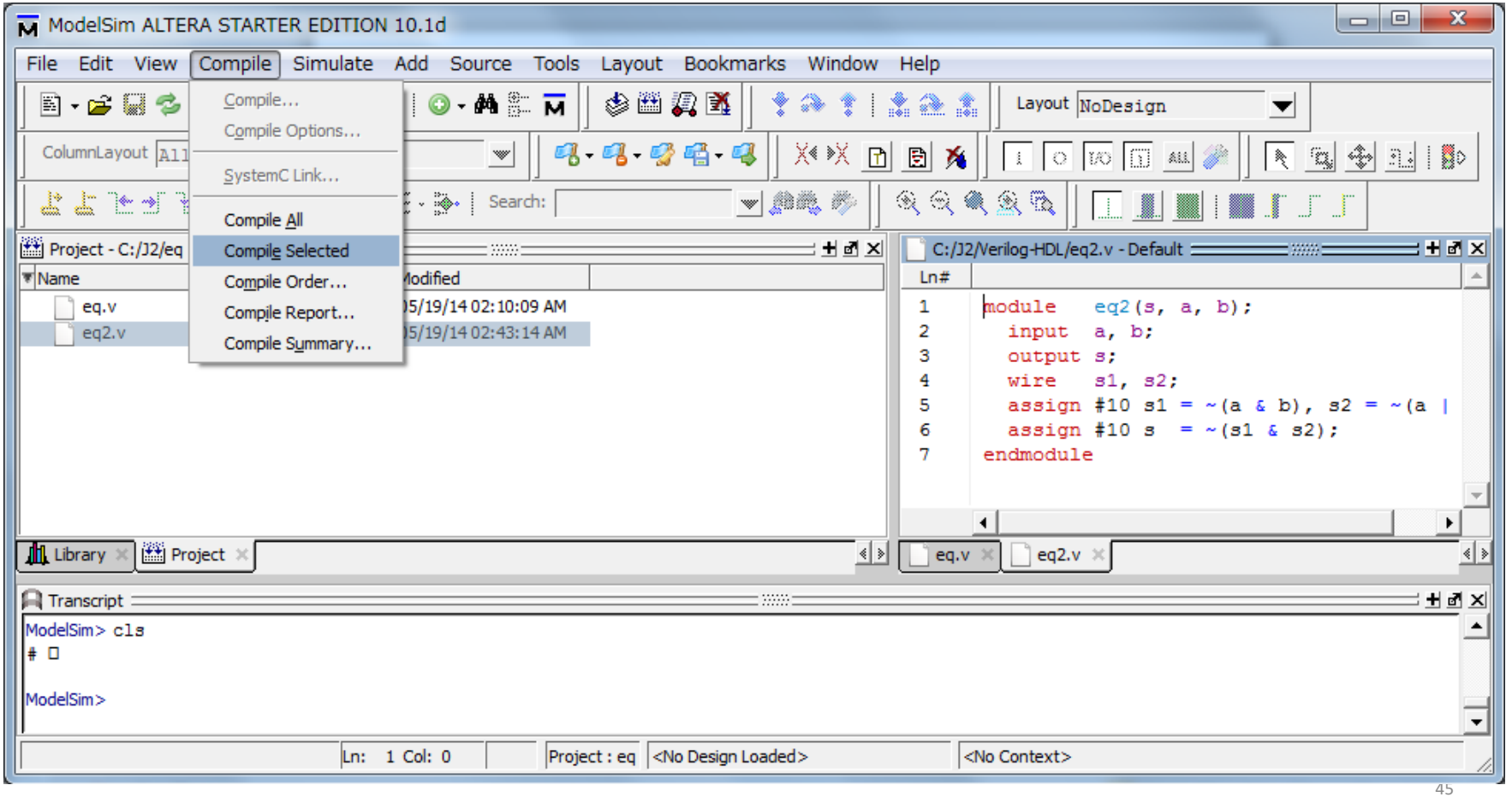
```
module eqSim2;
  wire   s;
  reg    x, y;
  eq2    g(s, x, y);

  initial
    begin
       $monitor( " %b %b", g.s1, g.s2, $stime);
      $display("g.s1 g.s2 time");
           x=0; y=0;
      #50  x=1;
      #50  y=1;
      #50  x=0;
      #50  $finish;
    end
endmodule
```



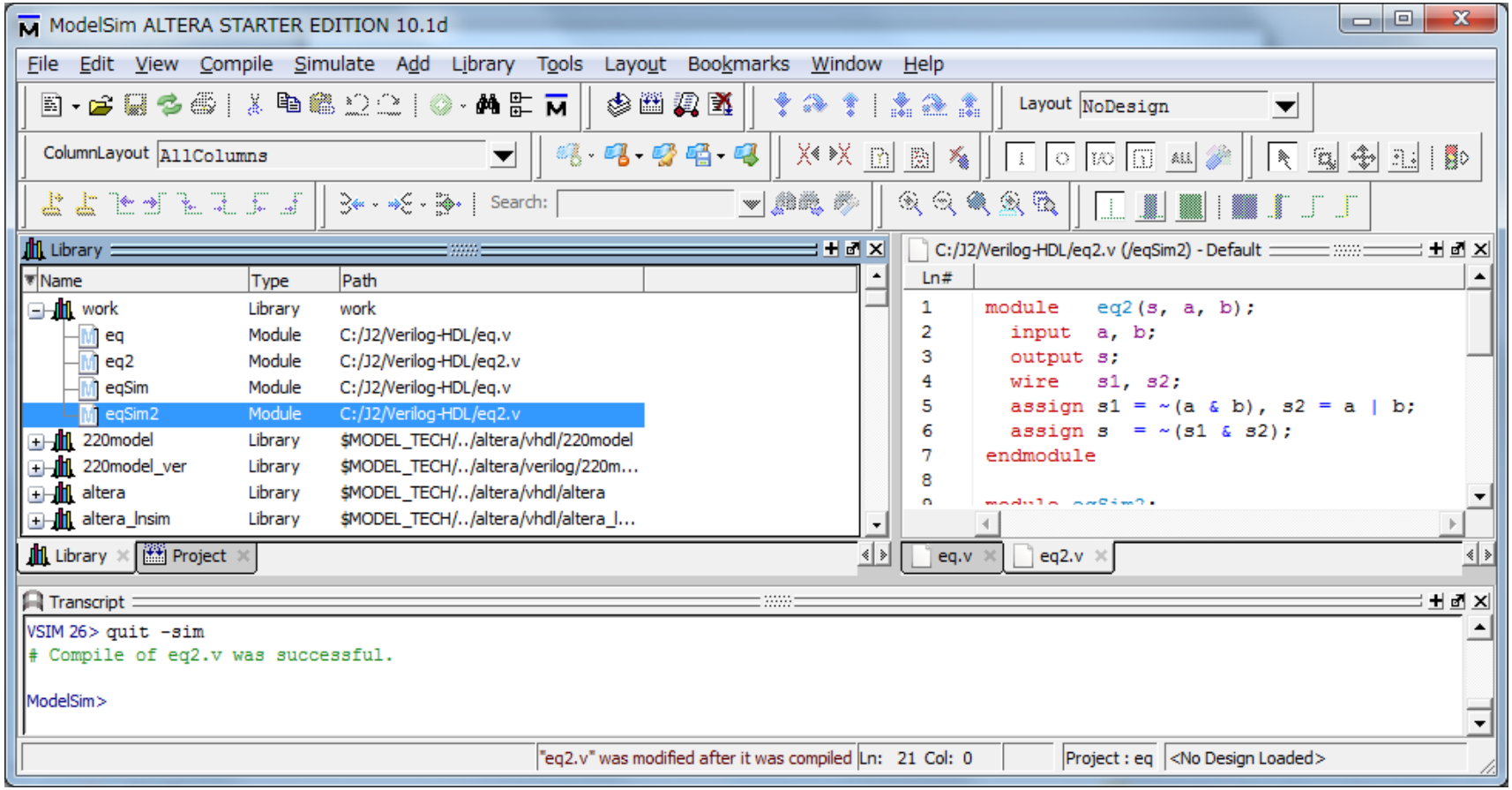
別回路のシミュレーション

- Window内の追加したファイルを選択して,
  Compile→Compile Selected(選択したファイルが対象)または
  Compile All(すべてのファイルが対象)でコンパイル



別回路のシミュレーション

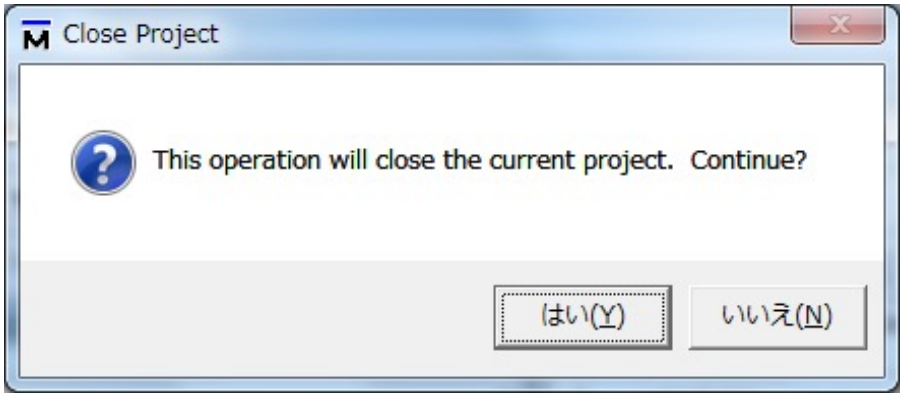
- Libraryタグのworkの下に，eq2とeqSim2ができている
- eqSimと同様にeqSim2をシミュレーション

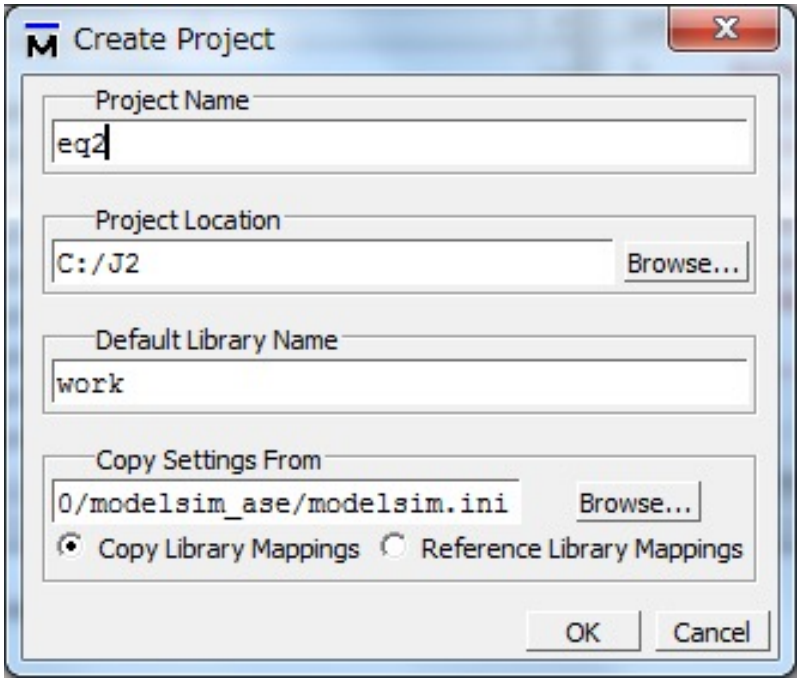


どうなりましたか？

プロジェクトの変更

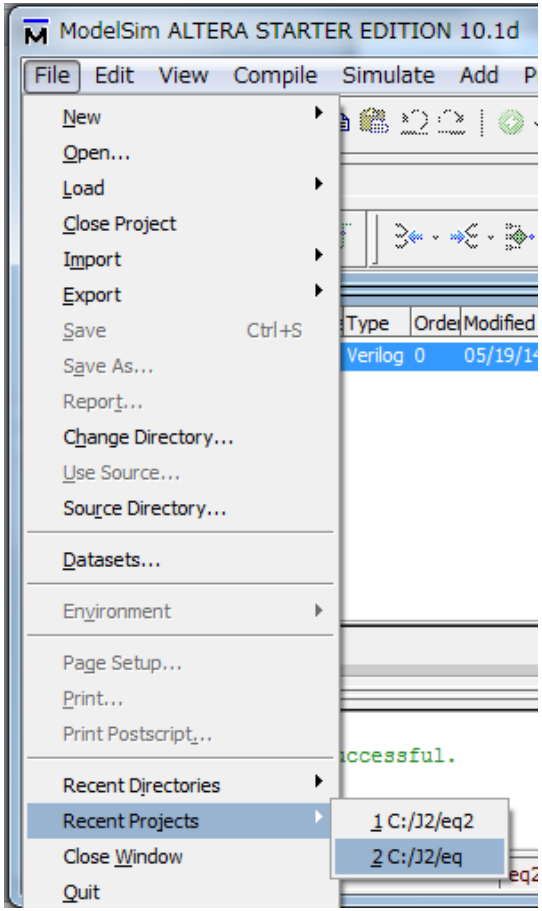
- File→Close Projectでプロジェクトを閉じる
- File→New Projectで新しいプロジェクトを作る
  - 前のプロジェクトを閉じていない時は閉じるかどうか聞かれる
- Project Name: eq2で新しいプロジェクトを開く
  - 他のプロジェクトとモジュールを共有するときや，名前が同じ異なるモジュールがなければDefault Library Name: workでよい

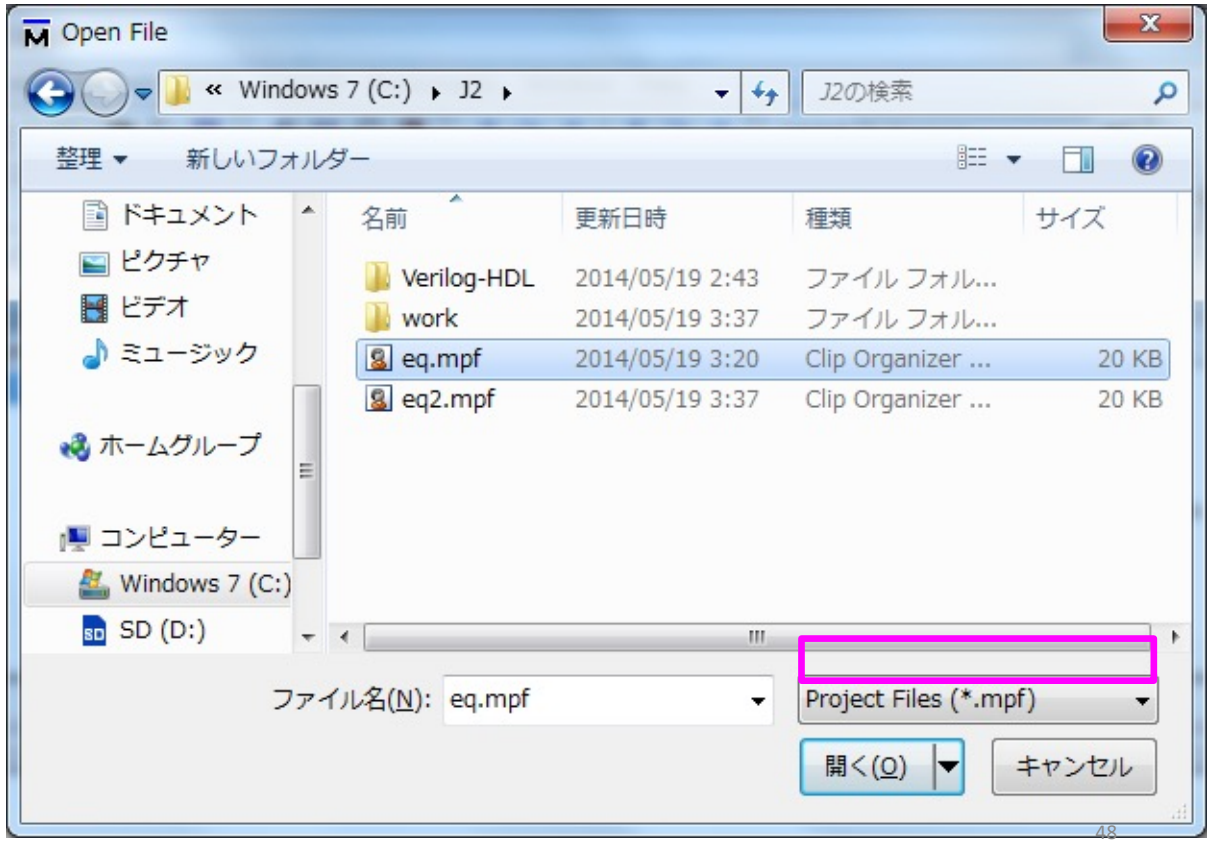




プロジェクトの変更

- プロジェクトeqに戻るときはFile→Resent Projectから選ぶか，File→Openでファイル選択のポップアップウィンドウの右下を“Project Files”に切り替えてmpf　(Modelsim Project File)ファイルを表示して選択





簡単な組み合わせ回路

（課題）

問題１

- モジュール eq の動作検証を eqSim を用いて行い，出力結果が正しくなるかどうか確認せよ．なお，検証には$monitorを用い，x, y, s をCUI上に表示して確認すること．

- また，eqSimを改良し，eqモジュールの内部信号であるs1, s2を$monitorでCUI上に表示できるようにせよ．
  $monitor文において，入れ子となっているモジュールの内部信号を観測するには“*モジュール名.信号名*”を使う．例として，p.35のeqSim2の$monitor文，$display文を参考にせよ．

問題２

- 入力信号 a, b, c, d を受け取り，$a = b$ と $c = d$ がともに成り立つとき出力信号 s を 1 に，それ以外のとき s を 0 にする回路のモジュールを，モジュールeq を 2 個使って作れ.

問題３

- 問題２で設計したモジュールの動作検証を行うテストベンチを作成せよ．また，作成したテストベンチモジュールを用いて，動作検証せよ．動作検証は，ModelSimを用いて，入出力信号の波形をGUI表示して確認すること．

順序回路

組み合わせ回路と順序回路

- 組み合わせ回路：**現在の入力のみ**によって出力が一意に決まる
  ここまで作成してきた回路は全て組み合わせ回路

- 順序回路：過去の入力により出力が変化する
  → すなわちデータを**記憶している**

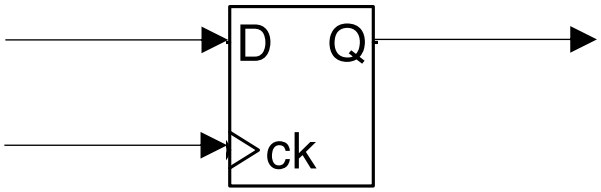
Dフリップフロップ

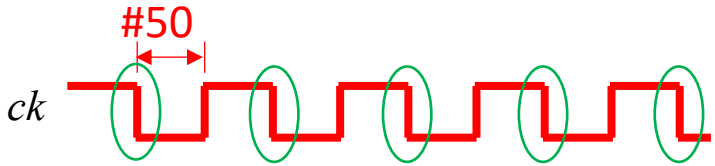
最も単純な順序回路

クロック信号に同期して出力を生成

従って次の周期に達するまでは入力が変化しても出力は不変

＝ 前周期の入力が次周期まで記憶される





Verilogでの順序回路作成

- レジスタ(reg)とalways文を使うことで，順序回路を作成できる
  - レジスタ…値を保存できる極小のメモリ

    wireが配線だったのに対し，レジスタ(regとして宣言)は

    データを格納する箱のイメージ
  - always文…always @(*イベント*) *処理内容*

    “*イベント* が起こる時，毎回 *処理内容* を実行する” という構文

Dフリップフロップ（Delay Frip Frop ）

クロック信号生成モジュール *clk*

```
module      clk(ck);
    output  ck;
    reg     ck;
    initial ck = 0;

    always #50 ck = ~ck;
endmodule
```

Reg の値を output したい時 同じ名前で定義してもよい

50サイクル経過する度に ck の値を反転

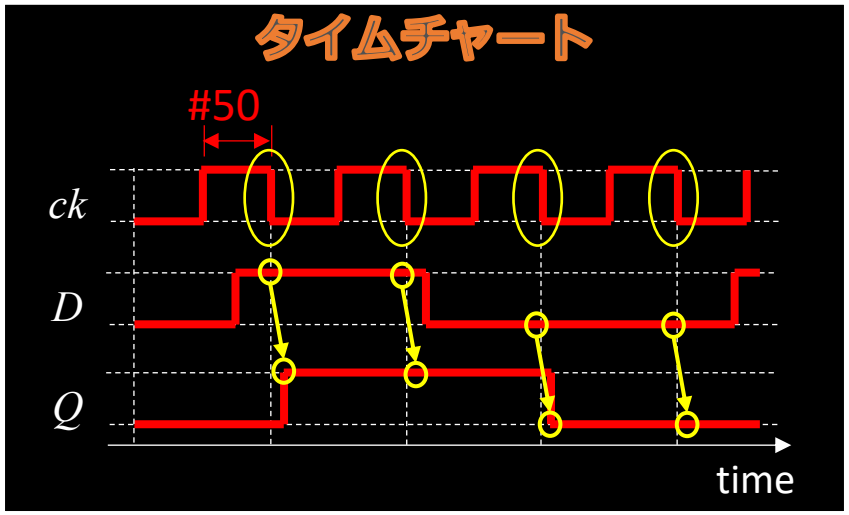
Dフリップフロップ *dffn*

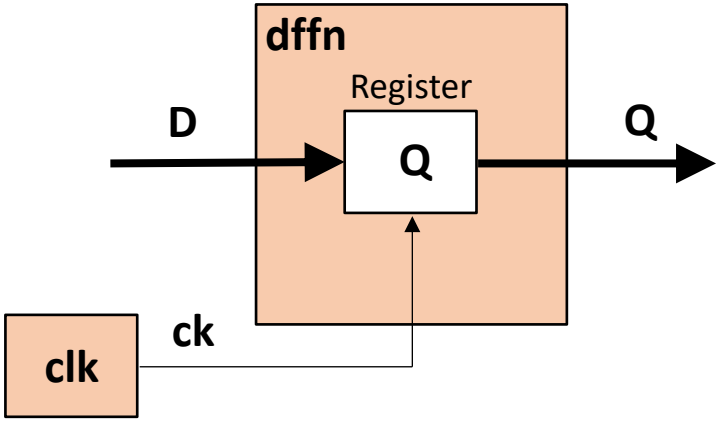
```
module      dffn(Q, D, ck);
    input   D, ck;
    output  Q;
    reg     Q;
    initial Q = 0;
    always @(negedge ck) Q = D;
endmodule
```

レジスタの初期化

Ckが下がる度にその時のDをQ1に保存

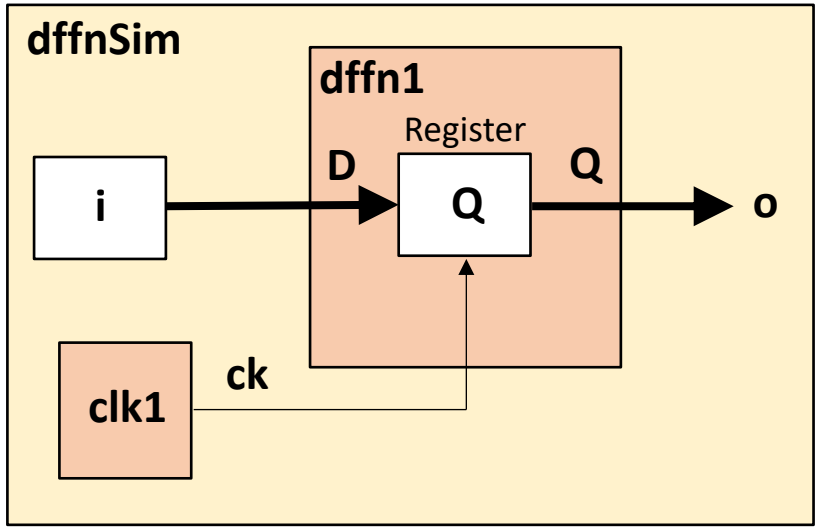
Ckが下がった瞬間の入力Dが出力Q





Dフリップフロップのテストベンチ

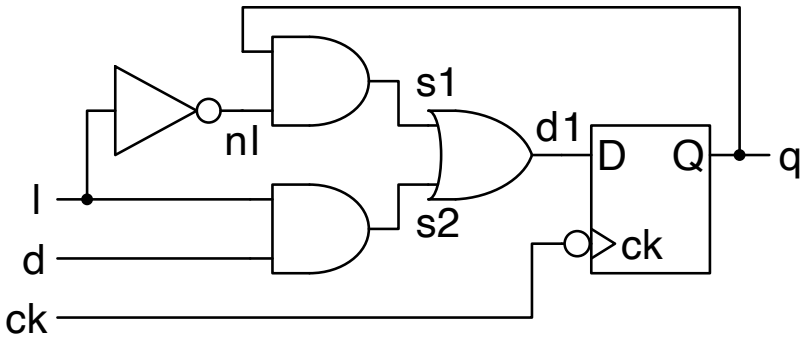
```
module      dffnSim;
    reg     i;
    wire    o;
    clk     clk1(ck);
    dffn    dffn1(o, i, ck);
    initial
      begin
        $monitor(“ %b %b %b”,
                  ck,i,o,$stime);
        $display(“ck i o time”);
             i = 0;
        #100 i = 1;
        #200 i = 0;
        #100 $finish;
      end
endmodule
```



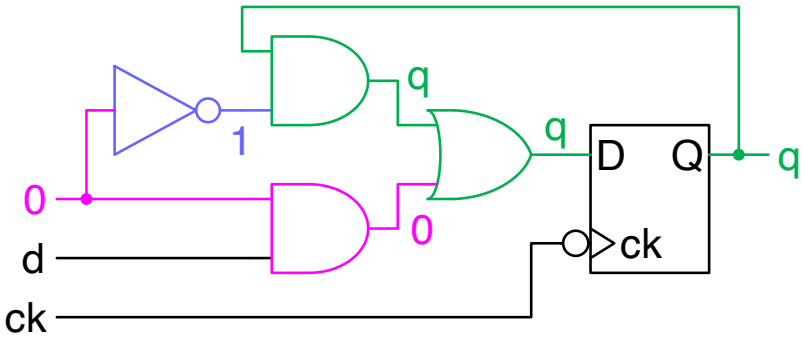
もう少し複雑な順序回路：1ビットの記憶回路

・制御信号 l でD-FFへの書き込みを制御

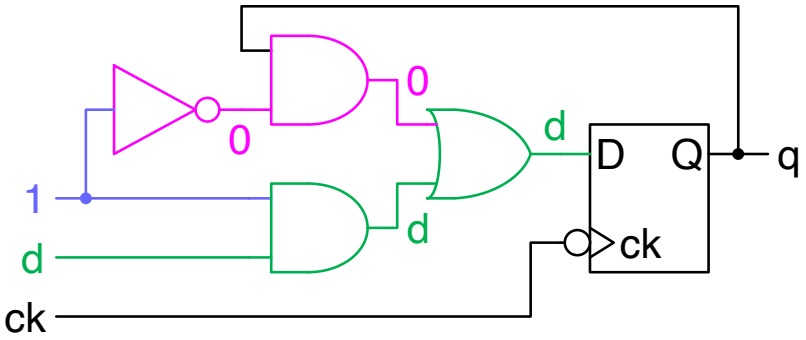
```
module   r1( q, l, d, ck );
  input    l, d, ck;
  output   q;
  wire     nl, s1, s2, d1;
  dffn     f( q, d1, ck );
  assign   nl = ~l;
  assign   s1 = nl & q,  s2 = l & d;
  assign   d1 = s1 | s2;
endmodule
```



l = 0 のとき q を保持



l = 1 のとき d を書込み



仕様からverilogで回路を作成するには？

- 前頁のように回路図が与えられれば，もうverilogで書けるはず
- それでは与えられた仕様から回路図を作るには？
  （論理回路の講義の復習）

  1．まず仕様から状態遷移図・真理値表を作成

  2．出力と次遷移状態を，入力と現遷移状態の論理式で表す
  (真理値表からカルノー図を用いて求める)

  3．得られた論理式から回路図を作成

  4．回路図をverilogで書き直す

- 次ページでは例として，**“スイッチaが押された状態でclkが立ち下がると，3クロックの間，１を出力する回路”**をverilogで作成する

逐次制御回路１（仕様→状態遷移表・遷移図）

- 仕様：
  スイッチaが押された状態でclkが立ち下がると，3クロックの間，１を出力
- パラメータ：
  - 入力a
  - 出力b
  - 現在の状態s1s0
  - 次の状態s1’s0’

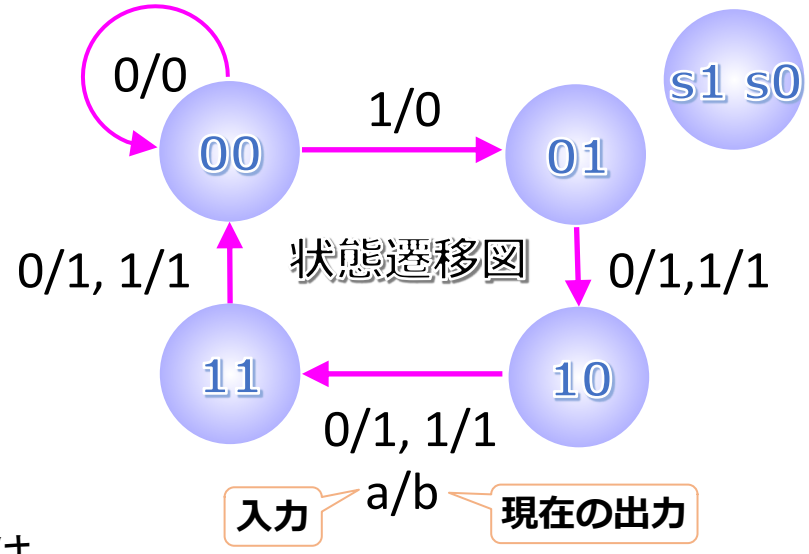
状態遷移表 m1

入力 a	現在の状態s1s0 00	01	10	11
0	00/0	10/1	11/1	00/1
1	01/0	10/1	11/1	00/1

次状態 s1’s0’/ b 現在の出力

現在状態が00の時
出力bは0
入力aが1なら
次の状態01に遷移

状態01, 10, 11の時は
出力bが1となる
入力aは0でも1でも
次の状態に遷移



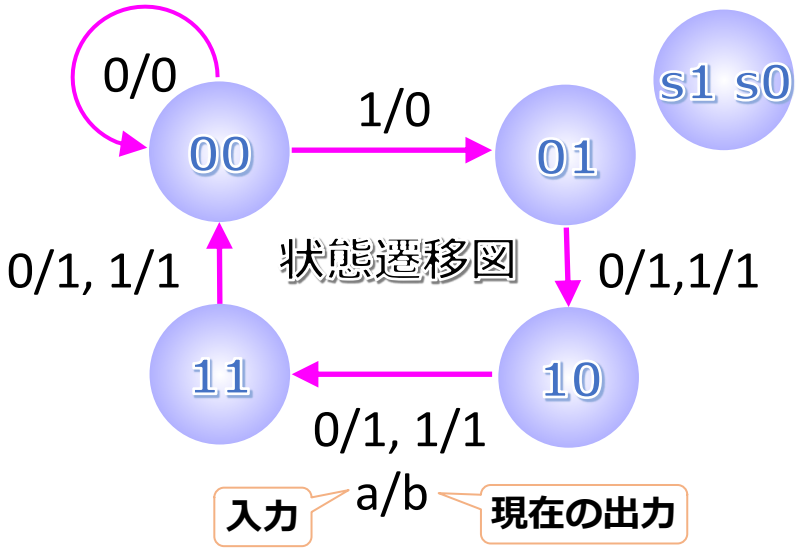
逐次制御回路１（状態遷移図→真理値表）

- 得られた状態遷移表，状態遷移図を基に真理値表を描く

状態遷移表 m1

		現在の状態s1s0			
		00	01	10	11
入力 a	0	00/0	10/1	11/1	00/1
	1	01/0	10/1	11/1	00/1

次状態 s1’s0’/ b 現在の出力



真理値表

a	s1	s0	b	s1’	s0’
0	0	0	0	0	0
1	0	0	0	0	1
x	0	1	1	1	0
x	1	0	1	1	1
x	1	1	1	0	0

x はdon’t careと言って，
0でも1でもどちらでも良いという意味

don’t care でまとめるのが難しいなら
a=0, a=1の２行に分けて描いても良い

逐次制御回路１（真理値表→論理式化）

- b, s1', s0' それぞれに関して**カルノー図**を作成し論理式化
- 状態s1s0の00 01 11 10という順番に注意
- カルノー図でまとめられる単位は1, 2, 4…なので1×1や1×2，2×2でまとめる

真理値表

a	s1	s0	b	s1'	s0'
0	0	0	0	0	0
1	0	0	0	0	1
x	0	1	1	1	0
x	1	0	1	1	1
x	1	1	1	0	0

カルノー図（b）

入力 a	状態s1s0 00	01	11	10
0	0	1	1	1
1	0	1	1	1

カルノー図（s1'）

入力 a	状態s1s0 00	01	11	10
0	0	1	0	1
1	0	1	0	1

カルノー図（s0'）

入力 a	状態s1s0 00	01	11	10
0	0	0	0	1
1	1	0	0	1

$b = s0 + s1$

$s1' = \overline{s1} \cdot s0 + s1 \cdot \overline{s0}$

$s0' = \overline{a} \cdot \overline{s0} + s1 \cdot \overline{s0}$

逐次制御回路１（論理式→回路図・実装）

$s1' = \overline{s1} \cdot s0 + s1 \cdot \overline{s0}$

$s0' = a \cdot \overline{s0} + s1 \cdot \overline{s0}$

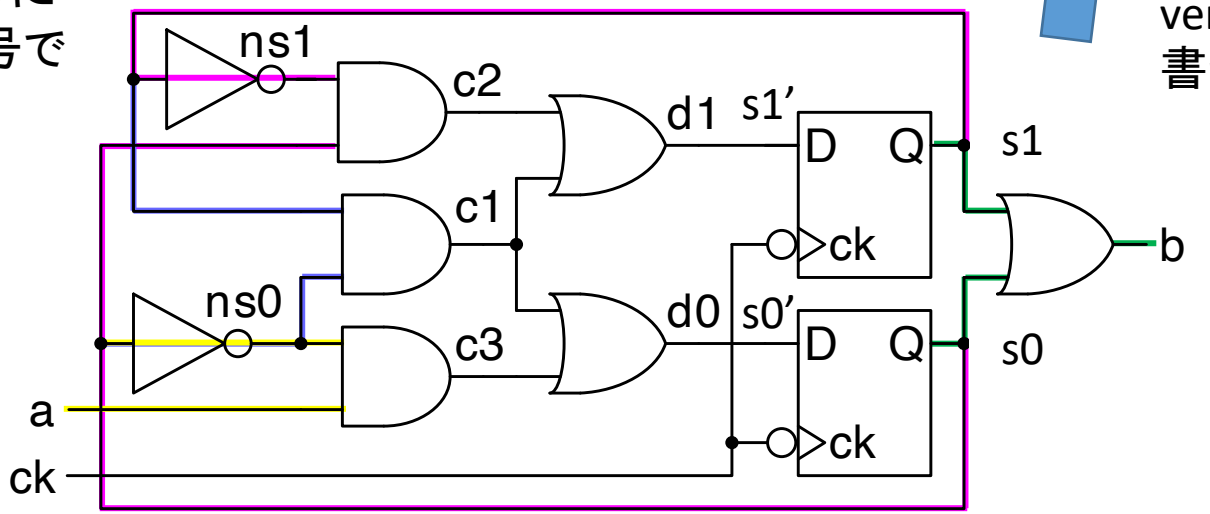
$b = s1 + s0$

今回は状態s1s0で
2bit分のFFが必要

2個のFFを中心に
b, s1', s0'を信号で
接続する

```
module m1( b, a, ck );
  input    a, ck;
  output   b;
  wire     ns1, ns0, s1, s0, d1, d0, c1, c2, c3;
  dffn     f1( s1, d1, ck ), f2( s0, d0, ck );
  assign   ns1 =  ~s1, ns0 = ~s0;
  assign   c1   =  s1 & ns0, c2 = ns1 & s0,
           c3   =  a & ns0;
  assign   d1   =  c1 | c2, d0 = c1 | c3,
           b    =  s1 | s0;
endmodule
```

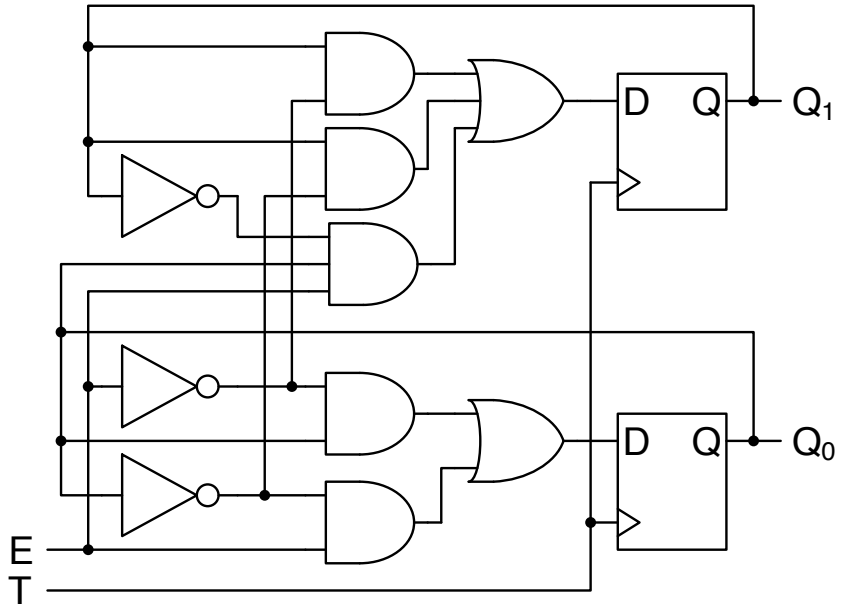
図が描けたら
verilogで
書き直すだけ



逐次制御回路２（回路図→状態遷移表→仕様）

- 順序回路を状態遷移図で表記して動作を解析，あるいは状態遷移図から順序回路を設計するときにはFFの値が「状態」を表す
- 入力と出力によってその状態間がどう遷移するかを調べる

- CS実験第一J1課題の回路の動作を調べる
  - 入力はE，出力はQ_1Q_0
  - 状態はQ_1Q_0（D-FFから直接なので出力と一致）
  - 2bitなので最大で4状態だが，全ての組合せを取らない場合もある



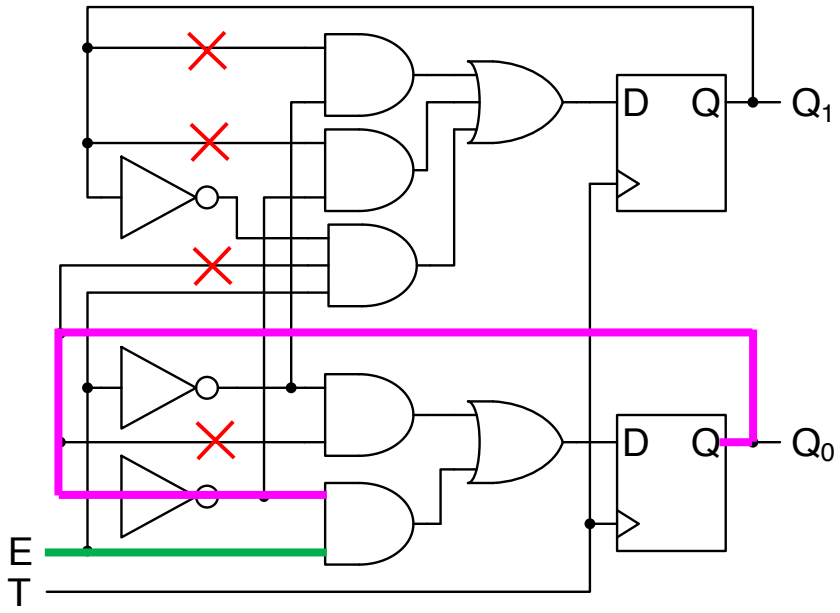
逐次制御回路２（状態遷移の確認）

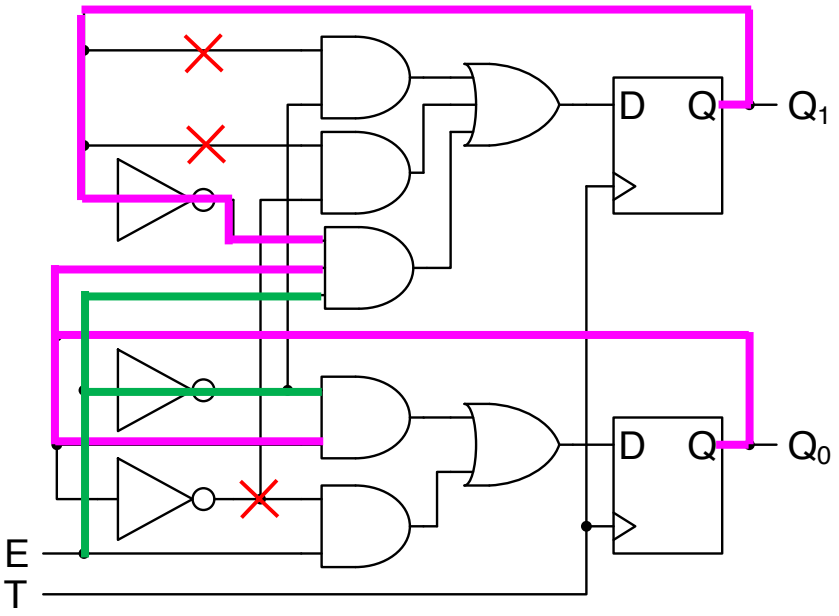
状態Q_1Q_0＝00の場合

- E＝0の時，　Q'_1＝0 Q'_0＝0となる
- E＝1の時，　Q'_1＝0 Q'_0＝1となる

状態Q_1Q_0＝01の場合

- E＝0の時，　Q'_1＝0 Q'_0＝1となる
- E＝1の時，　Q'_1＝1 Q'_0＝0となる





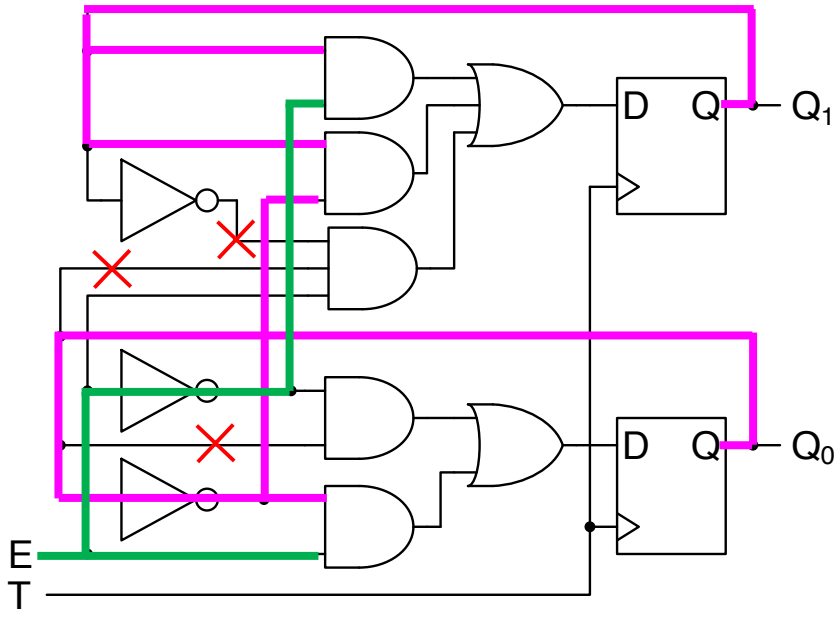
逐次制御回路２（状態遷移の確認）

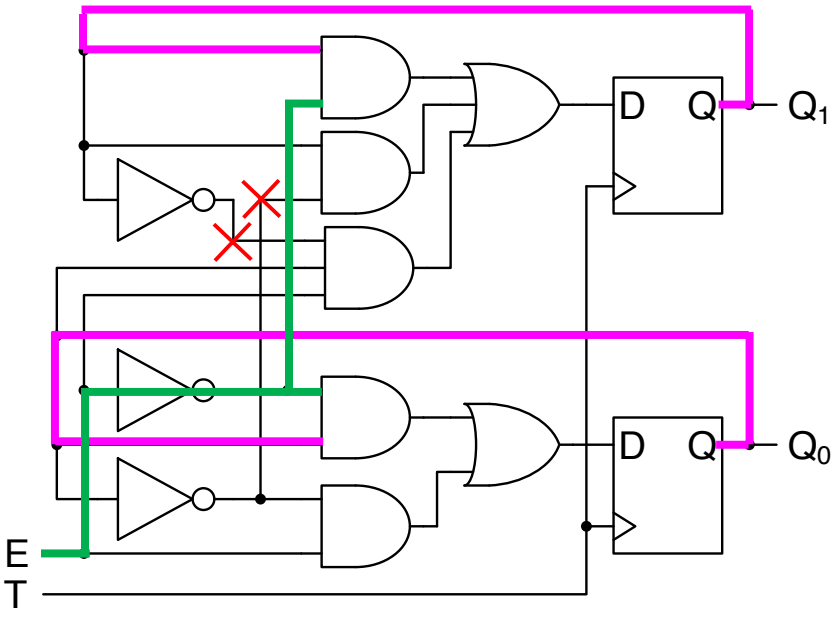
状態Q_1Q_0＝10の場合

- E＝0の時，　Q'_1＝1 Q'_0＝0となる
- E＝1の時，　Q'_1＝1 Q'_0＝1となる

状態Q_1Q_0＝11の場合

- E＝0の時，　Q'_1＝1 Q'_0＝1となる
- E＝1の時，　Q'_1＝0 Q'_0＝0となる





逐次制御回路２（真理値表・状態遷移図）

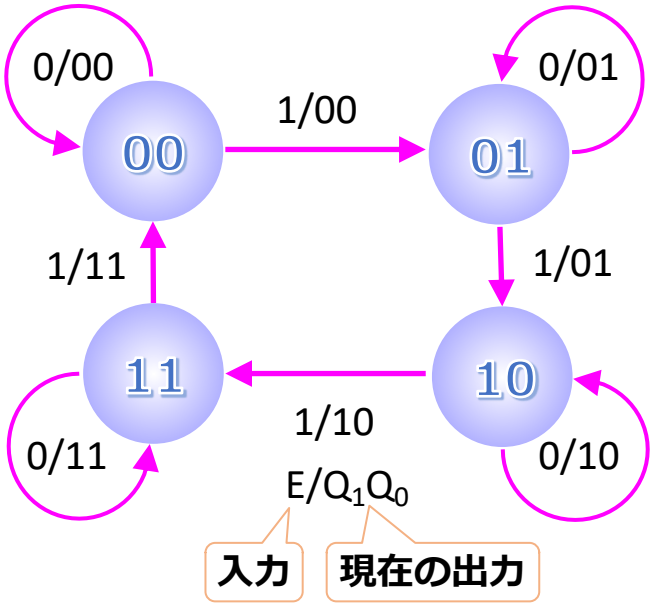
真理値表からE,Q_1Q_0と$Q'_1Q'_0$の関係を求める

真理値表

E	Q_1	Q_0	Q'_1	Q'_0
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	1
1	0	0	0	1
1	0	1	1	0
1	1	0	1	1
1	1	1	0	0

状態遷移図の作成

状態遷移図



１が入力されるたびに，カウントアップされた値（<4）を出力する回路だとわかる

逐次制御回路２（論理式の確認）

- カルノー図からQ_1, Q_0 の式が求められる

カルノー図（Q’1）

入力E	状態Q1Q0 00	01	11	10
0	0	0	1	1
1	0	1	0	1

カルノー図（Q’0）

入力E	状態Q1Q0 00	01	11	10
0	0	1	1	0
1	1	0	0	1

真理値表

E	Q_1	Q_0	Q'_1	Q'_0
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	1
1	0	0	0	1
1	0	1	1	0
1	1	0	1	1
1	1	1	0	0

$$Q'_1 = E \cdot \overline{Q_1} \cdot Q_0 + Q_1 \cdot \overline{Q_0} + \overline{E} \cdot Q_1$$

$$Q'_0 = \overline{E} \cdot Q_0 + E \cdot \overline{Q_0}$$

→ 回路図と照らし合わせて
一致しているかどうか

問題４

- 以下の状態遷移表m2に従って動く回路m2の状態遷移図を書き，回路を設計せよ．
- なお，回路m2は，初期状態 0 において入力 *a* に 1 が入力された状態でクロックが立ち下がると，状態 1 に遷移する．
  以降，同様の操作が行われる度に状態 2， 3 と遷移し，最終的に出力 *b* から 1 を出力し，初期状態 0 に戻る．
- 回路m2とそのテストベンチを実装し，シミュレーションで動作を確かめよ．ただし，回路m2の状態は，reg ではなく，2個の dffn を用いて実装すること．
- 上記のシミュレーション結果の何をもって動作確認としたのかも説明せよ．

状態遷移表 m2

		現在の状態s			
		0	1	2	3
入力 a	0	0/0	1/0	2/0	3/0
	1	1/0	2/0	3/0	0/1

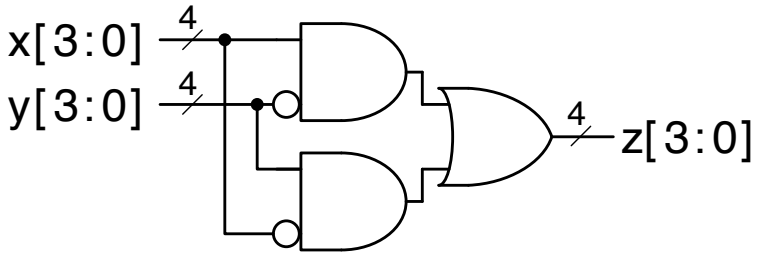
次状態 s’/b 現在の出力

加減算回路

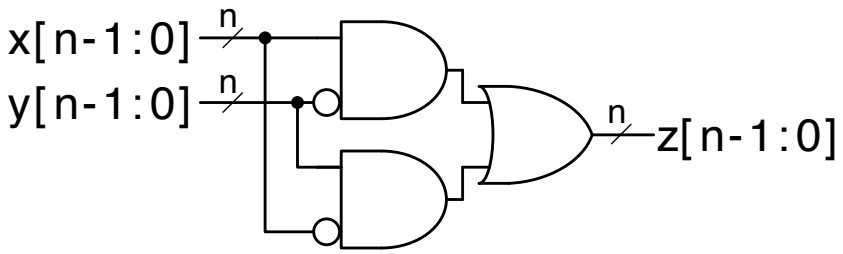
多ビットデータの表現

- 多ビットのデータは，x[3:0]のように配列で表す（＝x[3]x[2]x[1]x[0]の4bit値）
- Parameter 宣言を使うことでビット数を容易に変更できる

```
module eor4( z, x, y );
  input   [3: 0]  x, y;
  output  [3: 0]  z;
  wire    [3: 0]  nx, ny, z1, z2;
  assign  nx = ~x, ny = ~y;
  assign  z1 = x & ny, z2 = nx & y,
          z  = z1 | z2;
endmodule
```




```
module eorn( z, x, y );
  parameter  n = 8;
  input   [n-1: 0]  x, y;
  output  [n-1: 0]  z;
  wire    [n-1: 0]  nx, ny, z1, z2;
  assign  nx = ~x, ny = ~y;
  assign  z1 = x & ny, z2 = nx & y,
          z  = z1 | z2;
endmodule
```



eorn #4 e(g,a,b)
と呼び出すことで
n=4に指定を変えることも可能

ビットベクトル

- { }で信号を囲むことで複数の信号をまとめることが可能
  - 例えば，x[3:0]というのは{x[3:2], x[1:0]} {x[3], x[2], x[1], x[0]} と同じ
  - ビットベクトルを使うとシフト操作等が実現できる

    左シフト：{x[2:0], 1'b0} 左巡回シフト：{x[2:0], x[3]}

    右シフト：{1'b0, x[3:1]} 右算術シフト：{x[3], x[3:1]}

    ※　1'b0というのは1桁のb(2進数)の0という意味

    例えば2進数の11も2'b11と書かないと10進数の11扱いされてしまう

```
module   lshift( y, x );
  input   [3:0]   x;
  output  [3:0]   y;
  assign  y = {x[2:0], 1' b0};
endmodule
```

左1bitシフトのモジュール例

問題５の前提知識１（加算回路の設計）

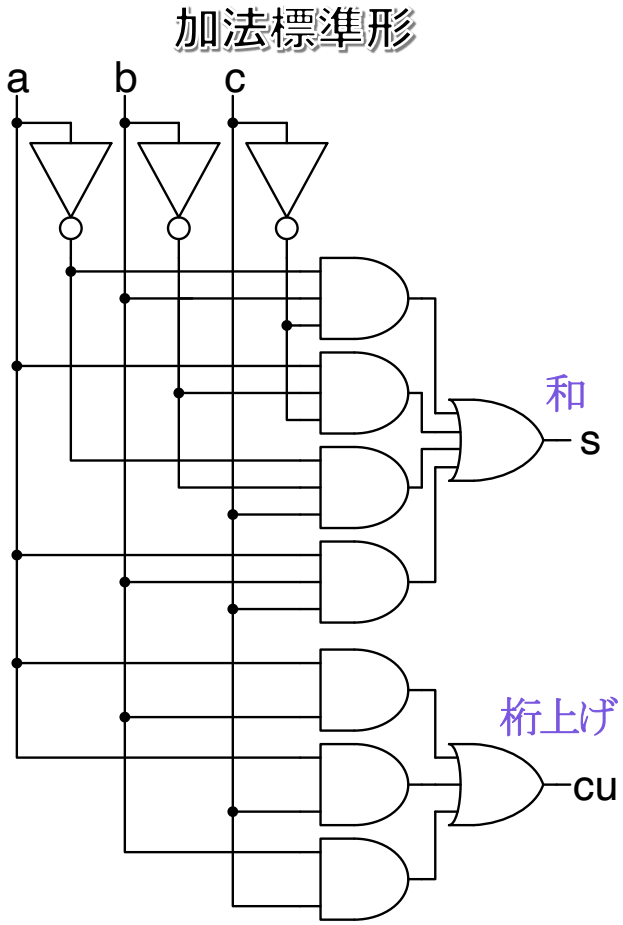
桁上げ　cu = a & b | b & c | c & a
和　　s = a & ~b & ~c | ~a & b & ~c | ~a & ~b & c | a & b & c
　　= a ^ b ^ c

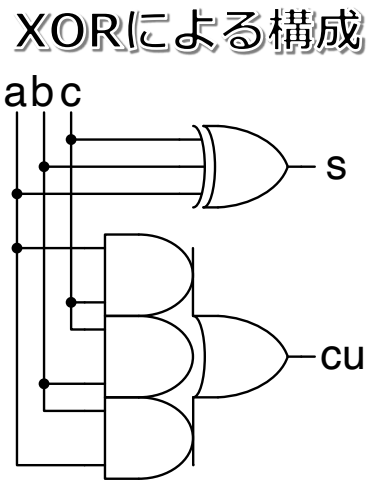
- 全加算器(Full Adder)は下からの桁上げを考慮した3入力の加算器

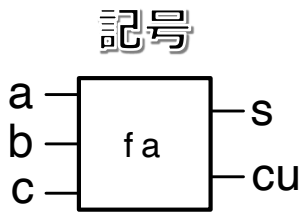
a	b	c	cu	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$cu = a \cdot b + a \cdot c + b \cdot c$

$s = a \cdot \bar{b} \cdot \bar{c} + \bar{a} \cdot b \cdot \bar{c} + \bar{a} \cdot \bar{b} \cdot c + a \cdot b \cdot c$



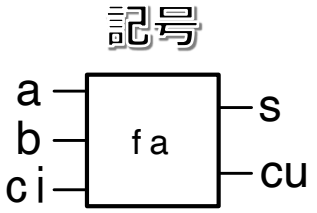




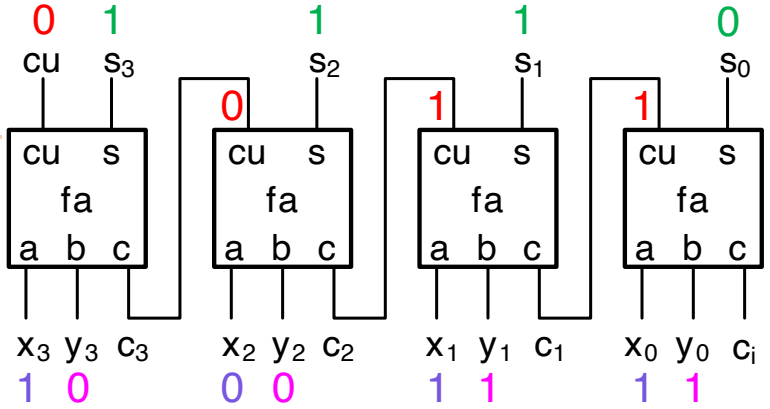
問題５の前提知識２（加算回路の設計）

・1ビット全加算器の場合（前頁を参考に）

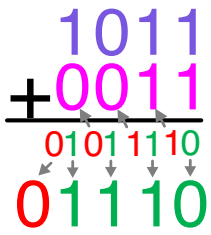
```
module   fa( cu, s, a, b, ci);
  input   a, b, ci;
  output  cu, s;
  assign  s = a ^ b ^ ci;
  assign  cu = a & b | b & ci | a & ci;
endmodule
```



・4ビット加算器を作るには？




```
module add4( cu, s, x, y, ci );
  input  [3: 0] x, y;
  input         ci;
  output        cu;
  output [3: 0] s;
  wire   [3: 0] c;
  fa            a3( cu, s[3], x[3], y[3], c[3]),
                a2(c[3], s[2], x[2], y[2], c[2]),
                a1(c[2], s[1], x[1], y[1], c[1]),
                a0(c[1], s[0], x[0], y[0], ci);
endmodule
```



問題５（加算回路の設計）

- 前頁による実装ではnビットの加算器は作れない．なぜなら，faモジュールをn個宣言することがverilogではできないからである．では，nビット加算器はどうすれば実装できるか？faモジュールを用いずにビットベクトルを使ってnビット加算器を作成せよ．

- このモジュールはadd4 のように、最上位の桁上げcu と、最下位に加えるci を持つこと。

- さらに、シミュレーションにより評価せよ。

（ヒント）
問題を解く肝になるのは，add4の”fa a3(~), a2(~)…a0(~);”の部分をいかにビットベクトルを用いて書き直すかである．次ページを参考に考えてみてください．

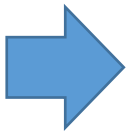
- 例： 4 bit加算器の場合
- 入力
  被加数 x[3:0]，加数 y[3:0]，最下位の桁上げ ci
- 出力
  加算結果 s[3:0]，最上位の桁上げ cu
- 途中の桁上げ　c[3:0]
  c[3]…3桁目からの桁上げ, c[2]…2桁目からの桁上げ
  c[1]…1桁目からの桁上げ, c[0]…0桁目からの桁上げ（=ci)

x[3] + y[3] + c[3] = □ □
cu s[3]

x[2] + y[2] + c[2] = □ □
c[3] s[2]

x[1] + y[1] + c[1] = □ □
c[2] s[1]

x[0] + y[0] + c[0] = □ □
c[1] s[0]



x[3] & y[3] | x[3] & c[3] | y[3] & c[3] = cu
x[2] & y[2] | x[2] & c[2] | y[2] & c[2] = c[3]
x[1] & y[1] | x[1] & c[1] | y[1] & c[1] = c[2]
x[0] & y[0] | x[0] & c[0] | y[0] & c[0] = c[1]

x[3] ^ y[3] ^ c[3] = s[3]
x[2] ^ y[2] ^ c[2] = s[2]
x[1] ^ y[1] ^ c[1] = s[1]
x[0] ^ y[0] ^ c[0] = s[0]

・4つの式をビットベクトルでまとめると？

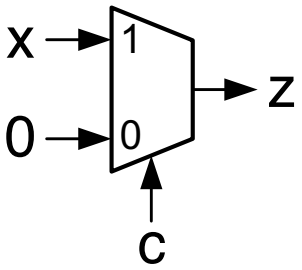
・4桁からn桁に拡張した場合どういう式になる？

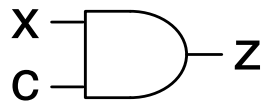
条件付きassign文

assign 変数= 条件? 条件が1(真) のときの値:
条件が0(偽) のときの値;

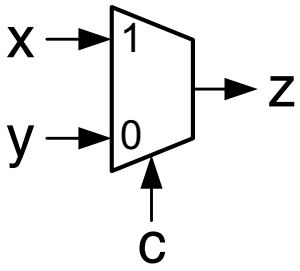
- 条件はC言語と同じ >,<, >=,<=, ==, != , <=などが, またその論理積(&&)や論理和(||)などが利用できる

assign z = c? x: 0;

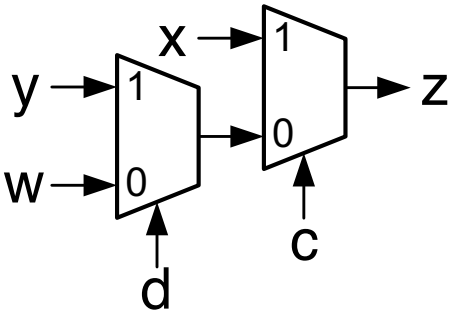




assign z = c? x: y;



assign z = c? x:
d? y: w;



繰り返し動作の記述（参考程度）

for (制御変数の初期設定; 継続条件;
制御変数の変更) 文;

```
module sel( z, x, c );
  parameter n = 4;
  input [n-1: 0] x;
  input c;
  output [n-1: 0] z;
  assign z = c? x: 0;
endmodule
```

```
module selSim;
  reg [3:0]  x;
  reg        c;
  wire [3:0] z;
  sel        sel1(z, x, c);
  initial
    begin
      $monitor("%b %b %b", x, c, z);
      c = 0;
      for(x=0; x<8; x=x+1) #50 c=~c;
      #50 $finish;
    end
endmodule
```

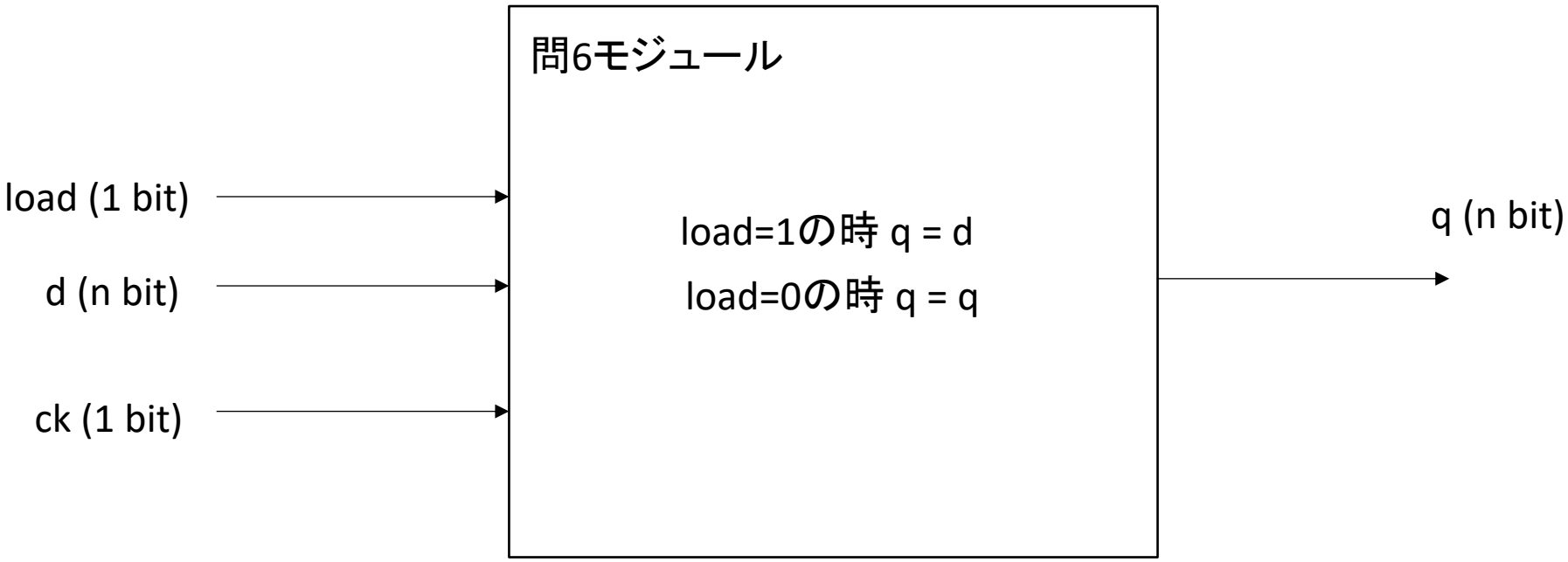
問題６（ *n* ビットレジスタ）

- クロック同期して動作し，以下に示す仕様を満たす *n* ビットレジスタのモジュール r を設計し、シミュレーションにより評価せよ.
- ちなみに１ビットレジスタの例はP.49に載せている.

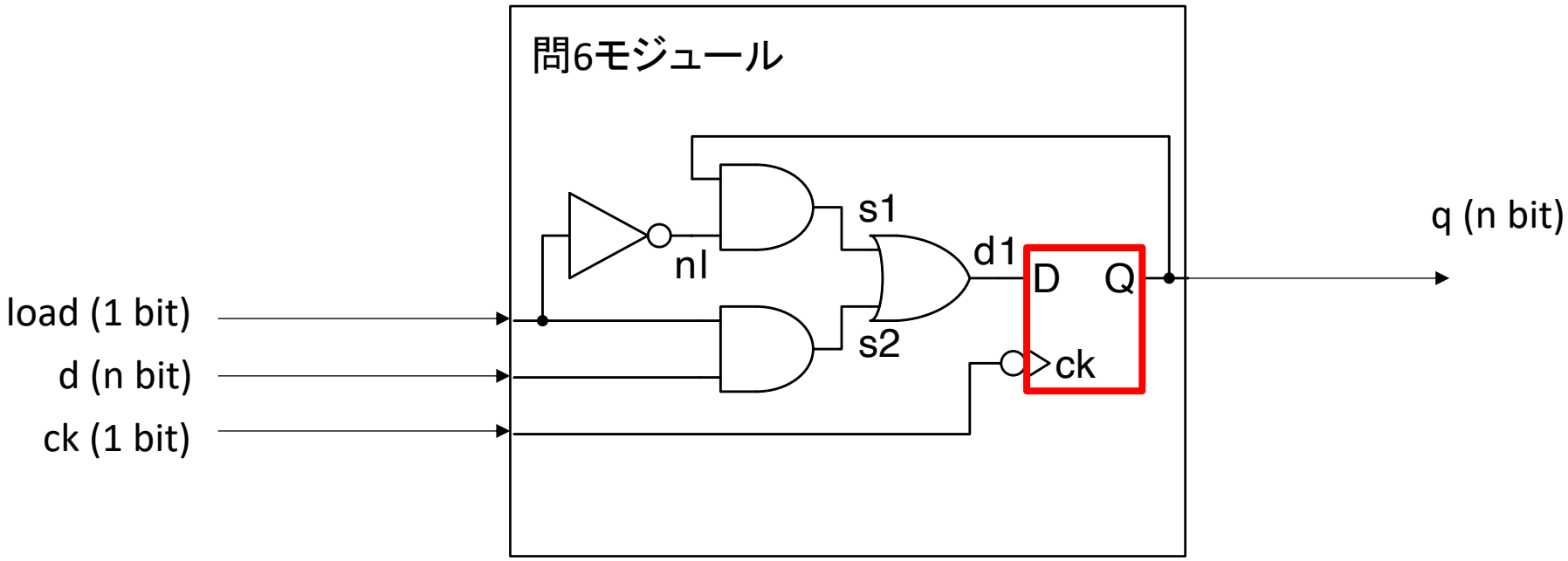
- この問題では，P.49の１ビットレジスタに対して
  ①データのビット幅を n ビットに拡張
  ②セレクタを用いて回路をより簡単化　すること

  - データ入力：　*d, ck*
  - 制御信号入力：　*load*
  - データ出力：　*q*
  - 動作：　*load* = 0 のとき *q* = *q*
    load = 1 のとき *q* = *d*

問題６（ n ビットレジスタ）



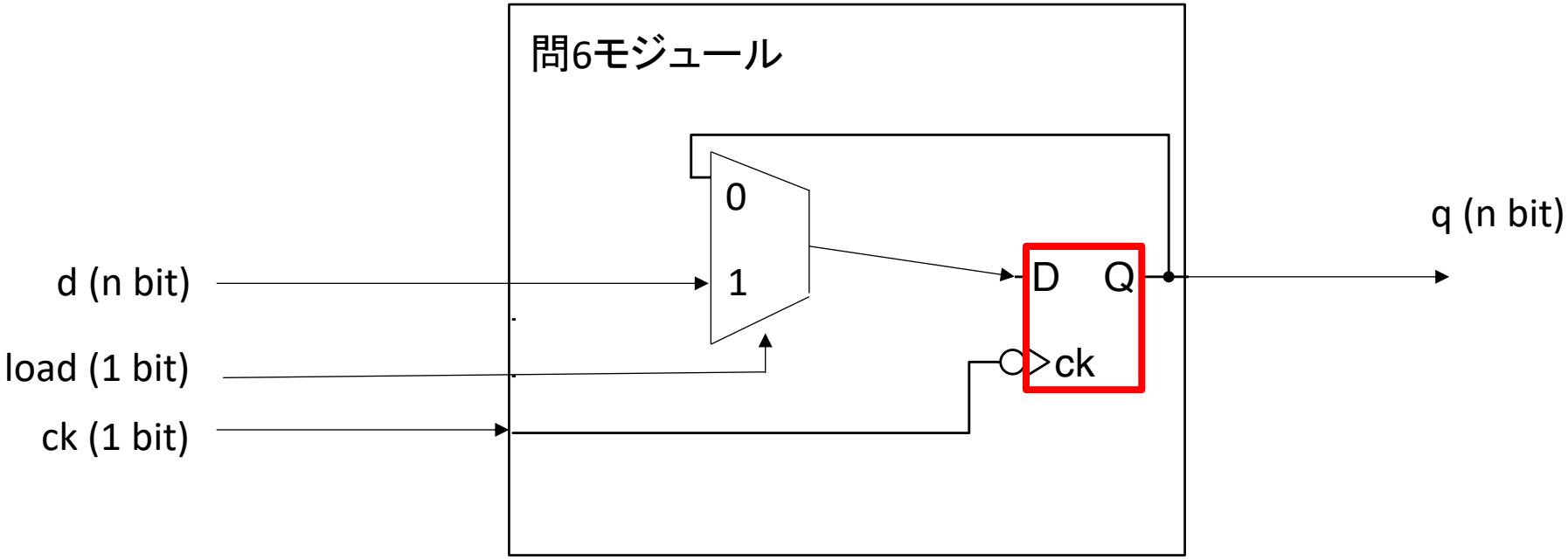
問題６（ *n* ビットレジスタ）



load=1の時 q = d

load=0の時 q = q

問題６（ *n* ビットレジスタ）



load=1の時 q = d

load=0の時 q = q

問題７（加減算回路の設計）

- 問題5の加算回路をベースに，以下の仕様の n 桁の加減算回路のモジュールを設計し、シミュレーションにより評価せよ．ただし負の数は2の補数表現とする．
  - 数の入力：　n 桁の整数 x, y
  - **制御信号入力**：　k
  - 出力：　桁上げ cu と n 桁の和 s
  - 動作：　$k = 0$ のとき $s = x + y$
    $k = 1$ のとき $s = x - y$

負数を2 の補数で表したとき、$x - y = x + \overline{y} + 1$ である．したがって，k の値によって y と $\overline{y}$ の何れかを選ぶ選択回路の出力と x を加算すればよく，1 の加算には、桁上げ入力を使う．

オプション課題

問題8 オプション課題

Xilinx VivadoとNEXYS 4を用いて,
加算器(問題5のadd4モジュール)を
FPGA上で動作させよ.

以下のURL からファイルをダウンロードし，作成したディレクトリに保存
http://www.hpc.is.uec.ac.jp/yamaki_lab/experiment/add4_nexys4.xdc
（コンソールでwget http://www.hpc.is.uec.ac.jp/yamaki_lab/experiment/add4_nexys4.xdc ~/mics_J2/）

NEXYS 4

Callout	Component Description	Callout	Component Description
1	Power select jumper and battery header	13	FPGA configuration reset button
2	Shared UART/ JTAG USB port	14	CPU reset button (for soft cores)
3	External configuration jumper (SD / USB)	15	Analog signal Pmod port (XADC)
4	Pmod port(s)	16	Programming mode jumper
5	Microphone	17	Audio connector
6	Power supply test point(s)	18	VGA connector
7	LEDs (16)	19	FPGA programming done LED
8	Slide switches	20	Ethernet connector
9	Eight digit 7-seg display	21	USB host connector
10	JTAG port for (optional) external cable	22	PIC24 programming port (factory use)
11	Five pushbuttons	23	Power switch
12	Temperature sensor	24	Power jack

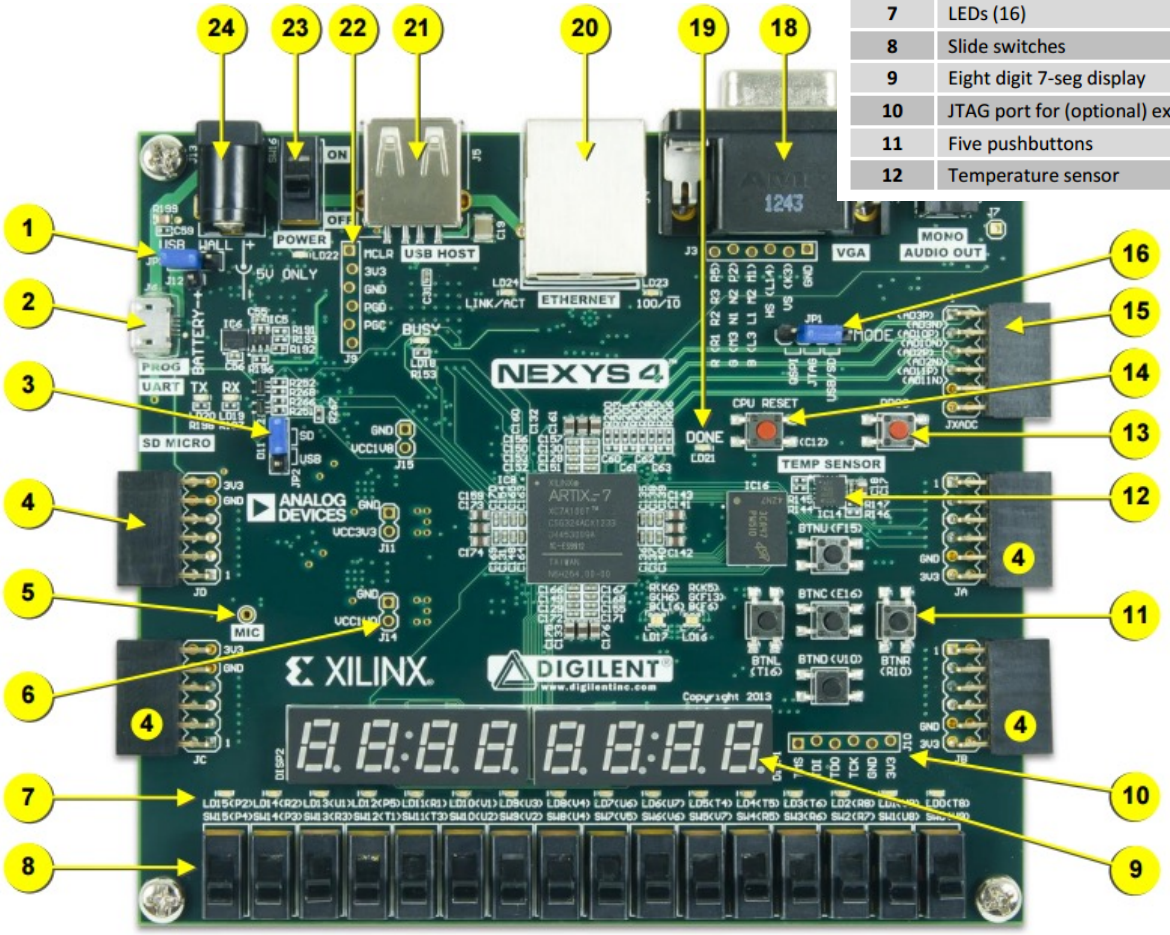
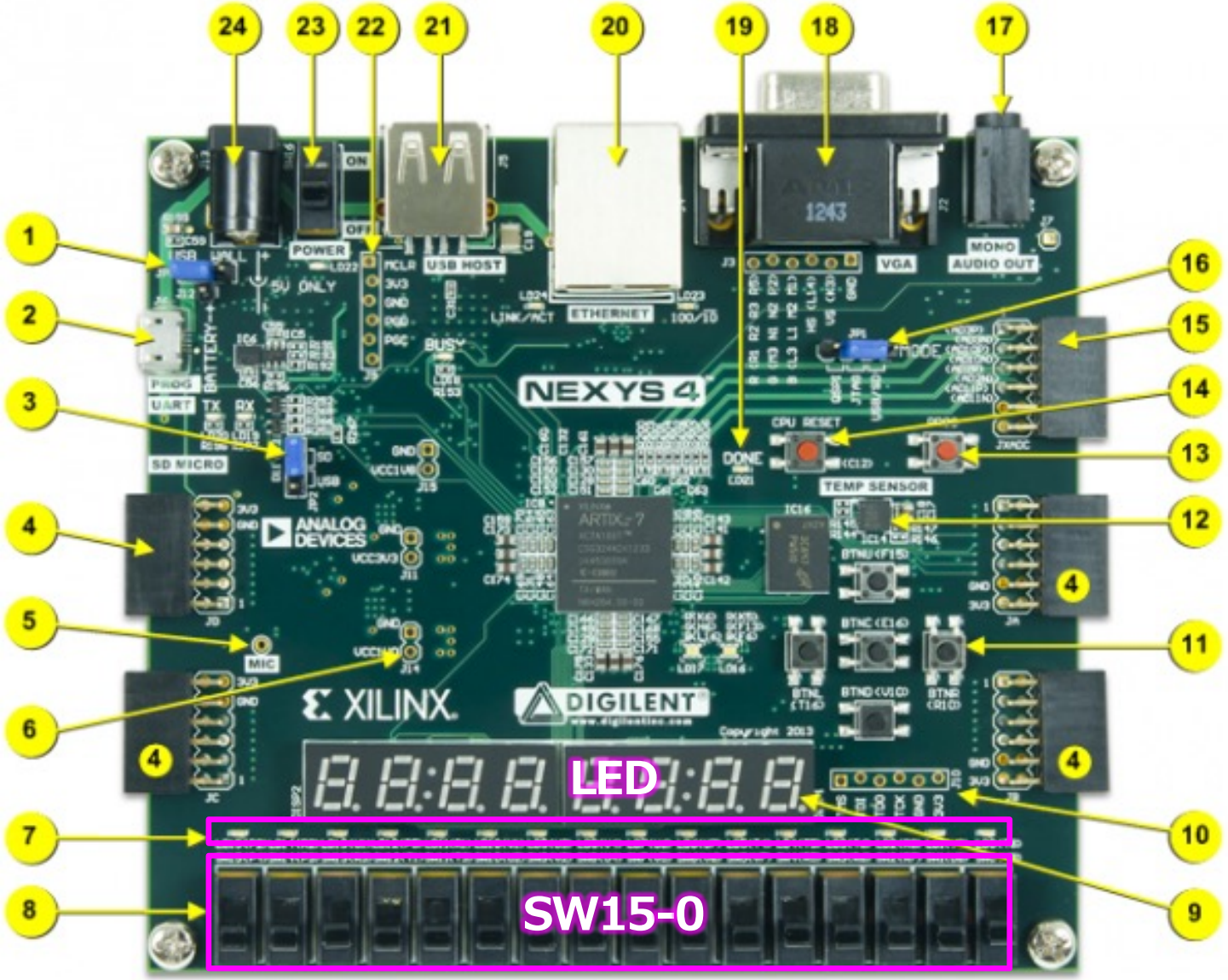


Figure 1. Nexys 4 board features

[出典] https://reference.digilentinc.com/_media/nexys:nexys4:nexys4_rm.pdf

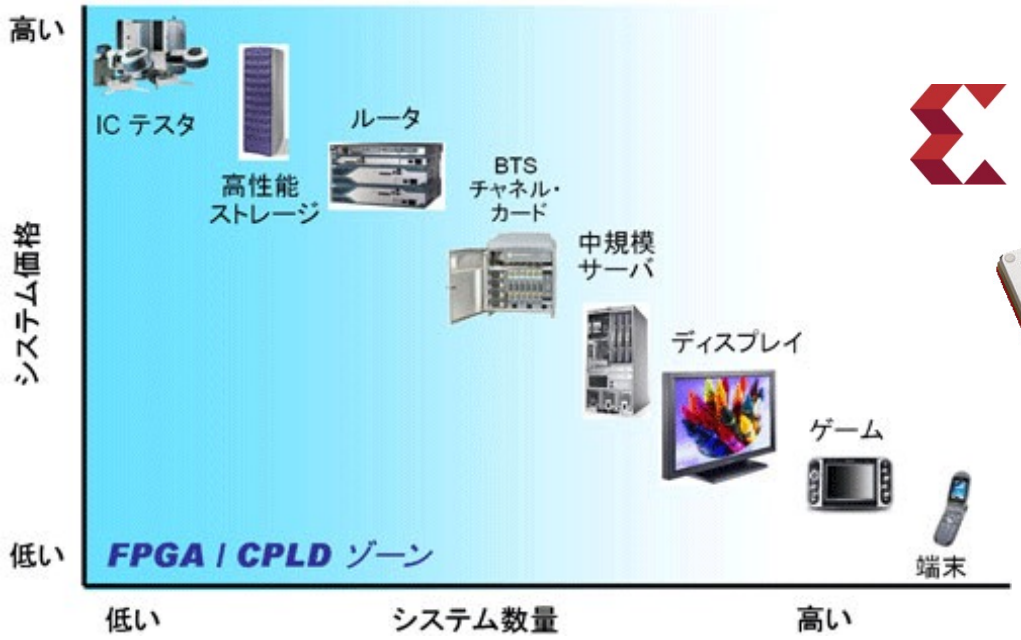
add4の実行

- SW7-0の切り替えに応じてLEDが点灯する



FPGAの用途

- FPGA
  - Field Programmable Gate Array
  - ユーザが論理機能を何度でも書き換え可能
  - 少量多品種・短ライフサイクル品
- ASIC
  - Application Specific Integrated Circuit
  - 回路の論理が固定の特定アプリケーション用のカスタム品
  - プロセッサ等の大量生産品



2大メーカー

XILINX®

ALTERA®

xdcファイル:FPGAのピン配置を決めるファイル
まずはFPGAボードを確認

#LED vivado　計算結果を出力するLED(1だと点灯、0だと無灯)

```
set_property PACKAGE_PIN R18 [get_ports {cu}]
set_property IOSTANDARD LVCMOS33 [get_ports {cu}]
set_property PACKAGE_PIN N14 [get_ports {s[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {s[3]}]
set_property PACKAGE_PIN J13 [get_ports {s[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {s[2]}]
set_property PACKAGE_PIN K15 [get_ports {s[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {s[1]}]
set_property PACKAGE_PIN H17 [get_ports {s[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {s[0]}]

#PUSH BUTTON
set_property PACKAGE_PIN N17 [get_ports {ci}]
set_property IOSTANDARD LVCMOS33 [get_ports {ci}]
```

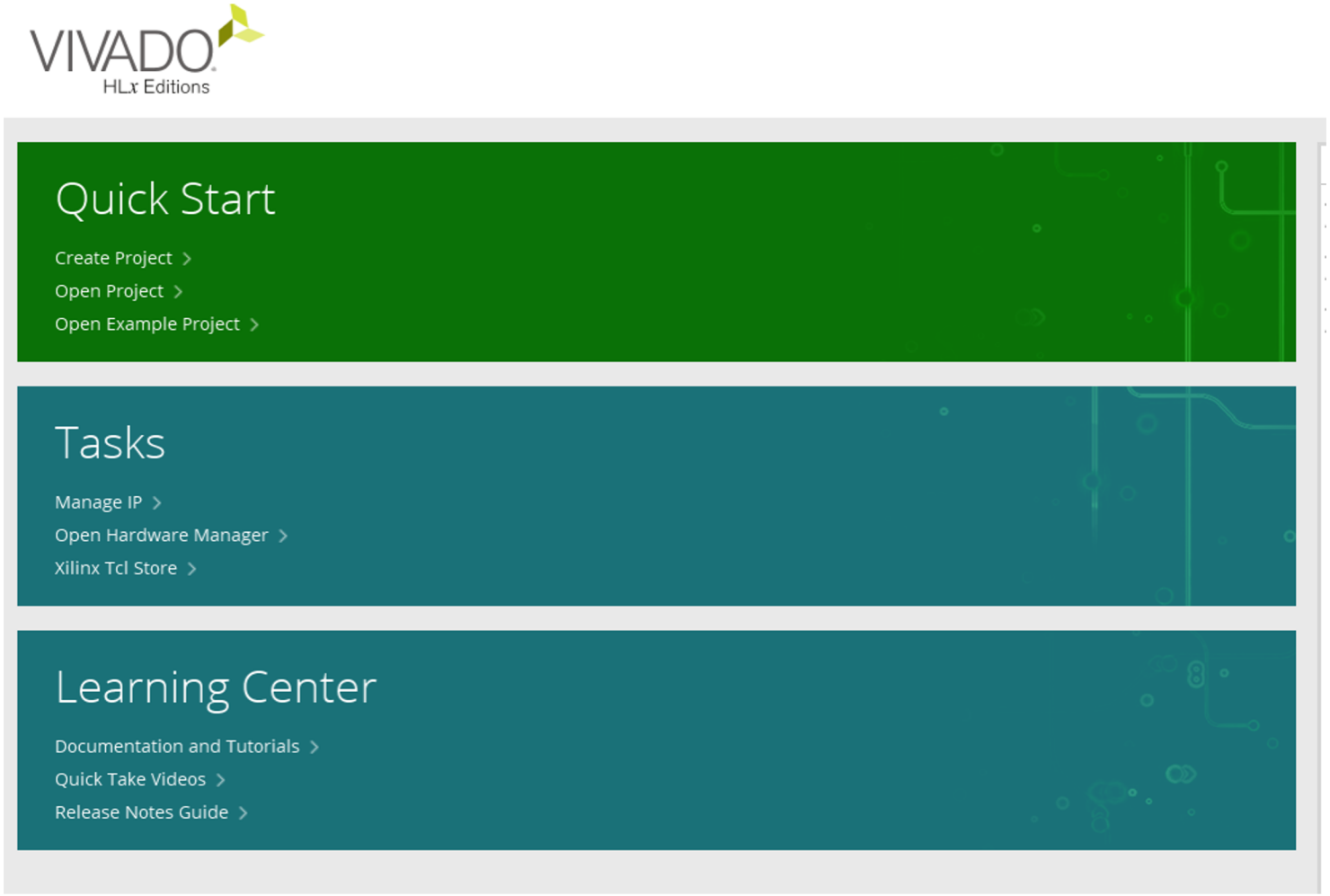
xdcファイル:FPGAのピン配置を決めるファイル
まずはFPGAボードを確認

#SWITCH vivado　4ビットの値を入力するスイッチ

```
set_property PACKAGE_PIN R13 [get_ports {x[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {x[3]}]
set_property PACKAGE_PIN U18 [get_ports {x[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {x[2]}]
set_property PACKAGE_PIN T18 [get_ports {x[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {x[1]}]
set_property PACKAGE_PIN R17 [get_ports {x[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {x[0]}]
set_property PACKAGE_PIN R15 [get_ports {y[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {y[3]}]
set_property PACKAGE_PIN M13 [get_ports {y[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {y[2]}]
set_property PACKAGE_PIN L16 [get_ports {y[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {y[1]}]
set_property PACKAGE_PIN J15 [get_ports {y[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {y[0]}]
```

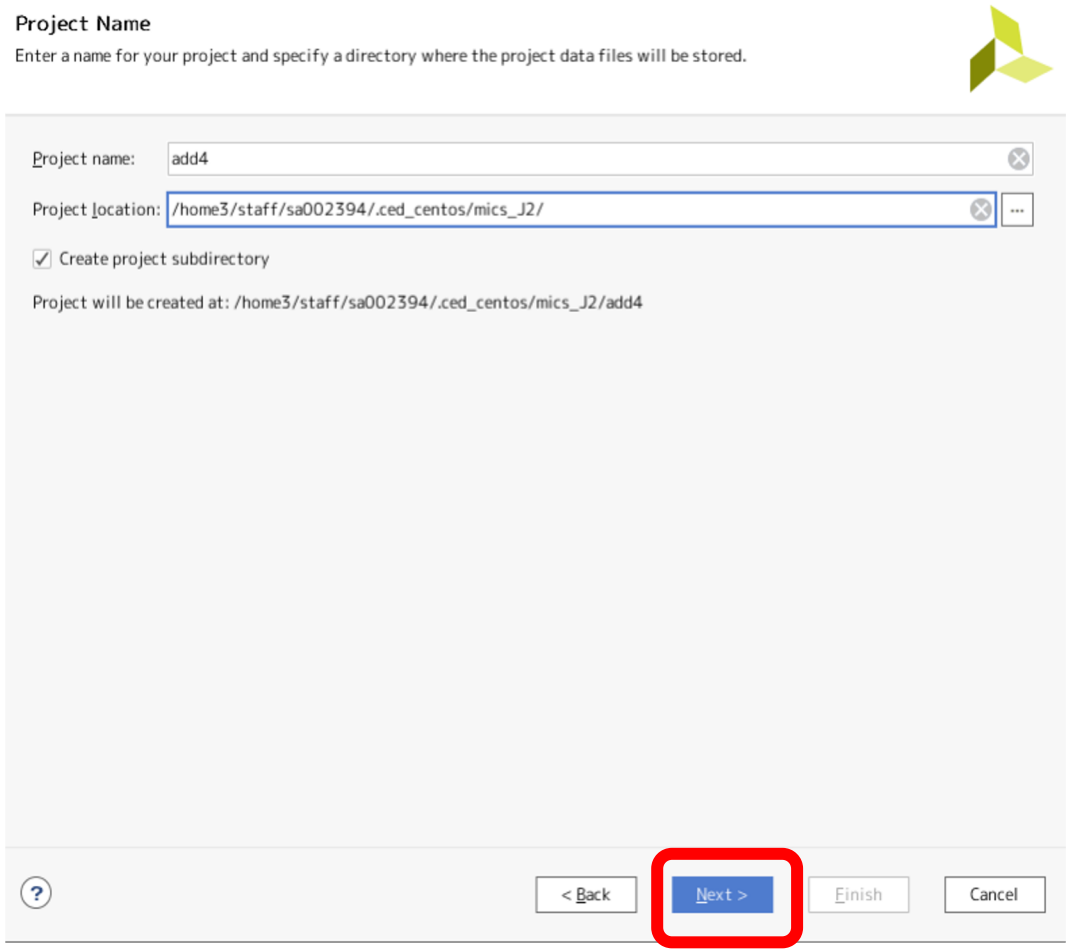
Vivadoを起動

- Terminal で vivado と入力
- Create New Projectをクリック



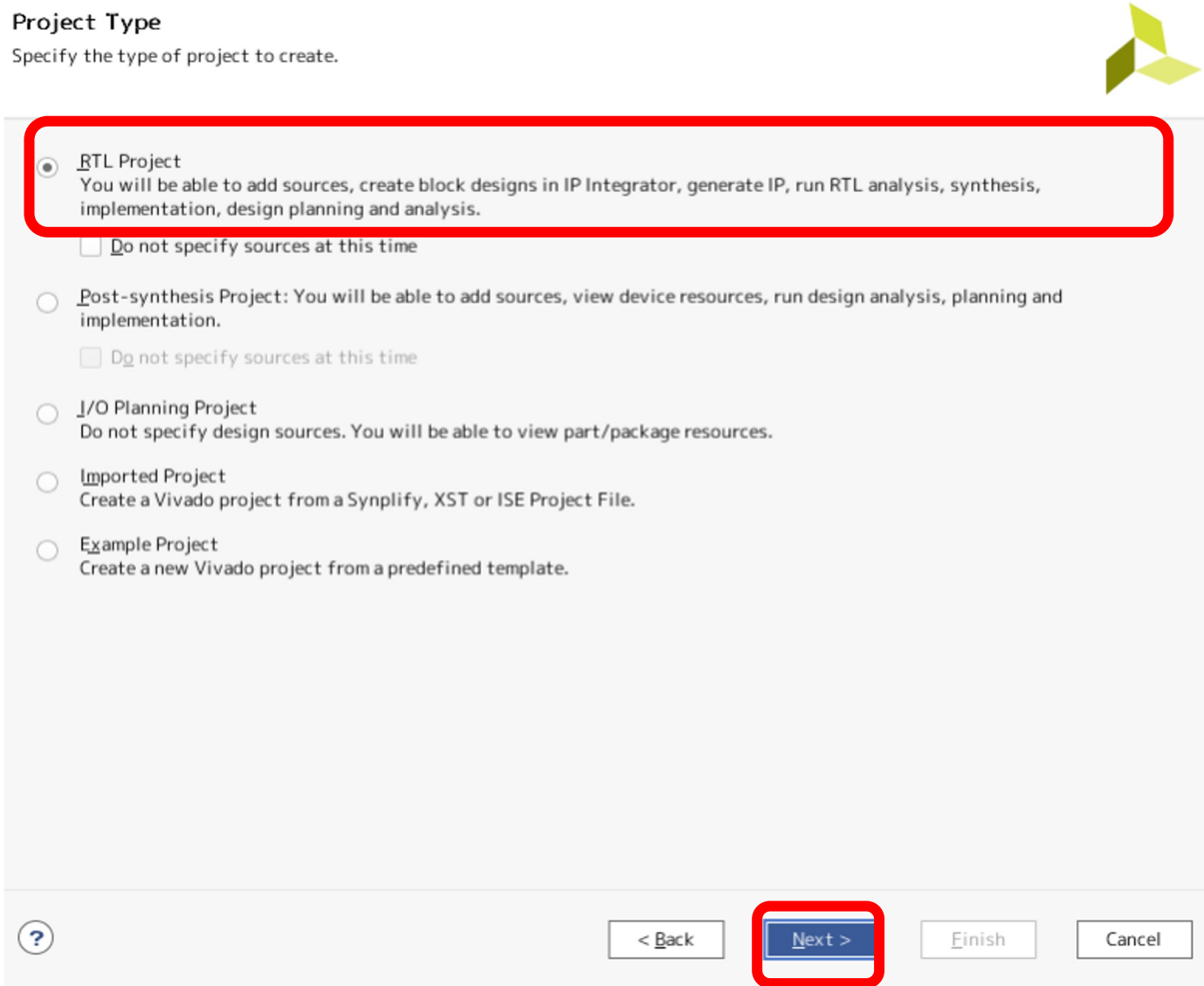
プロジェクトディレクトリを指定

- Project name: add4
- Project location: 任意のディレクトリ
- Create project Subdirectoryにチェック



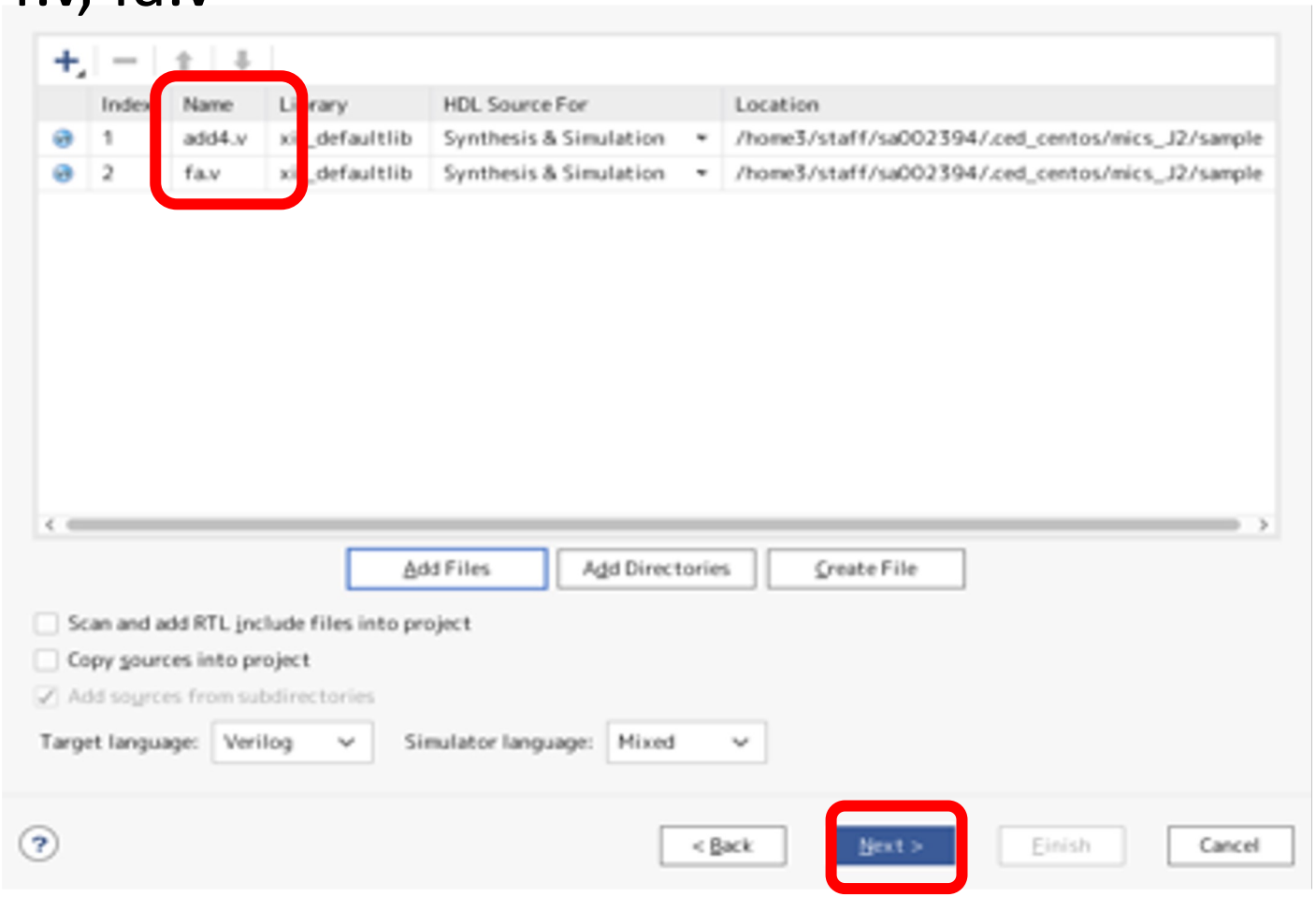
プロジェクトタイプの指定

RTLプロジェクトにチェック



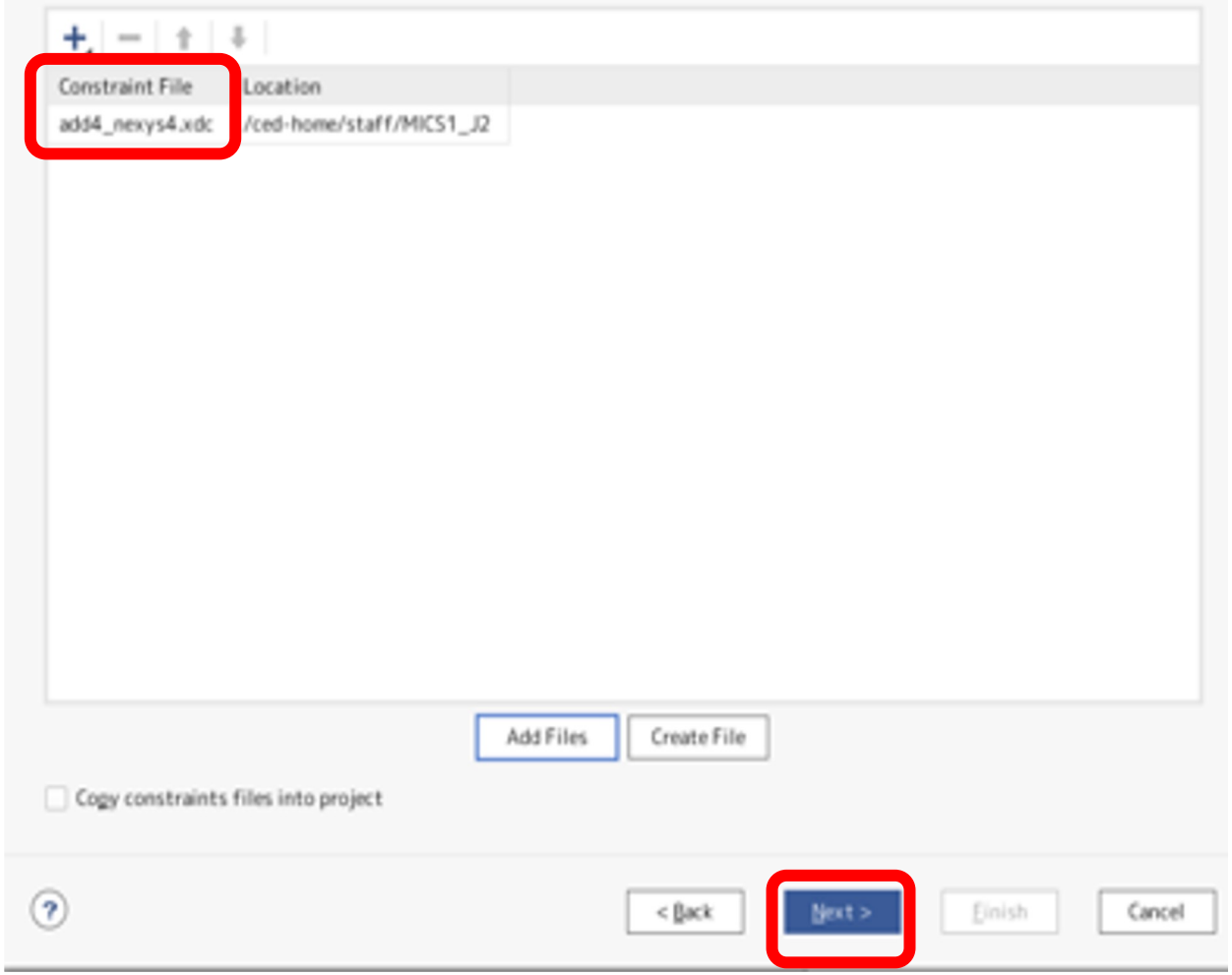
ソースファイルの指定

- Add filesで次のファイルを追加
  add4.v, fa.v



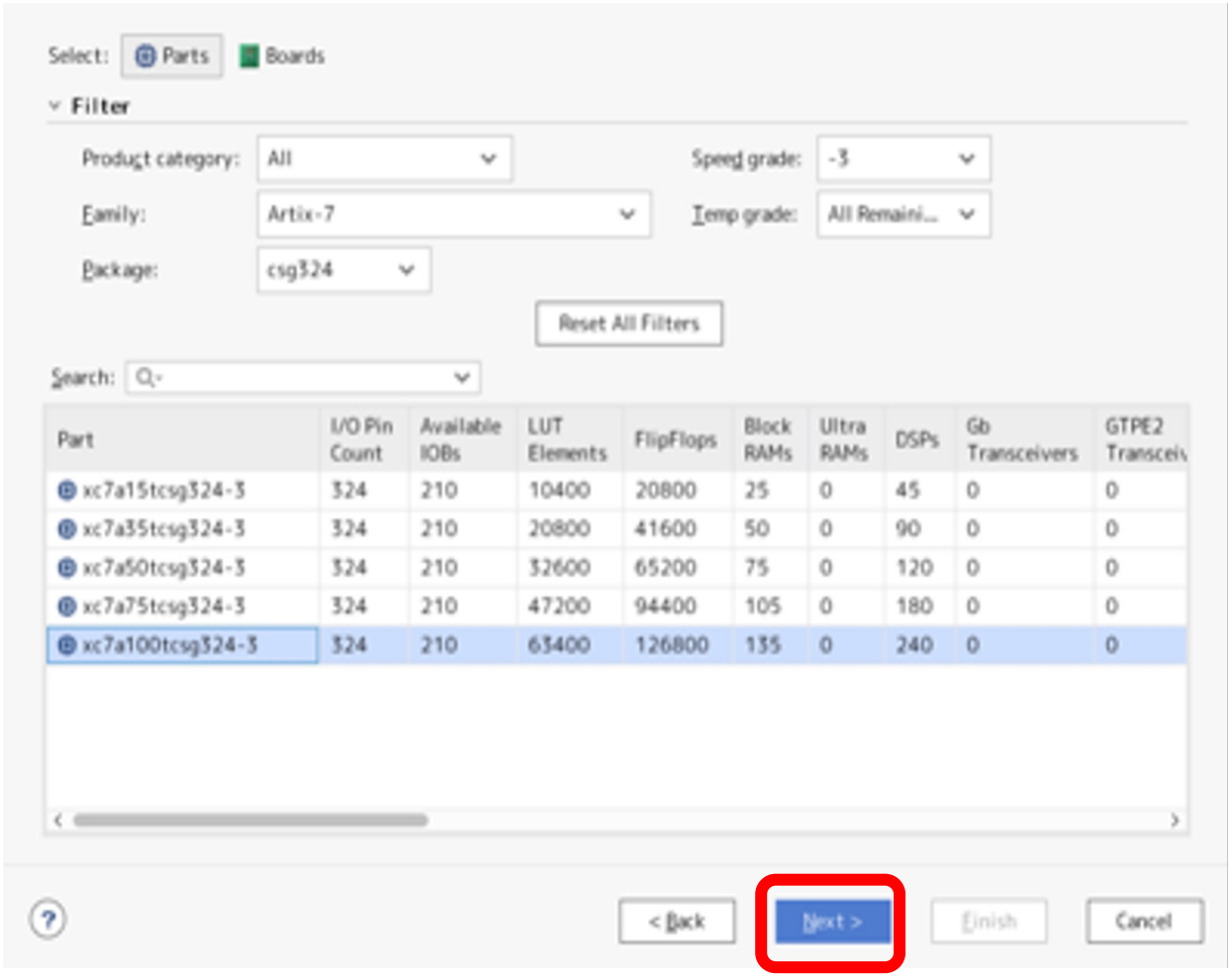
制約ファイルを追加

- Add filesで次のファイルを追加　add4_nexys4.xdc

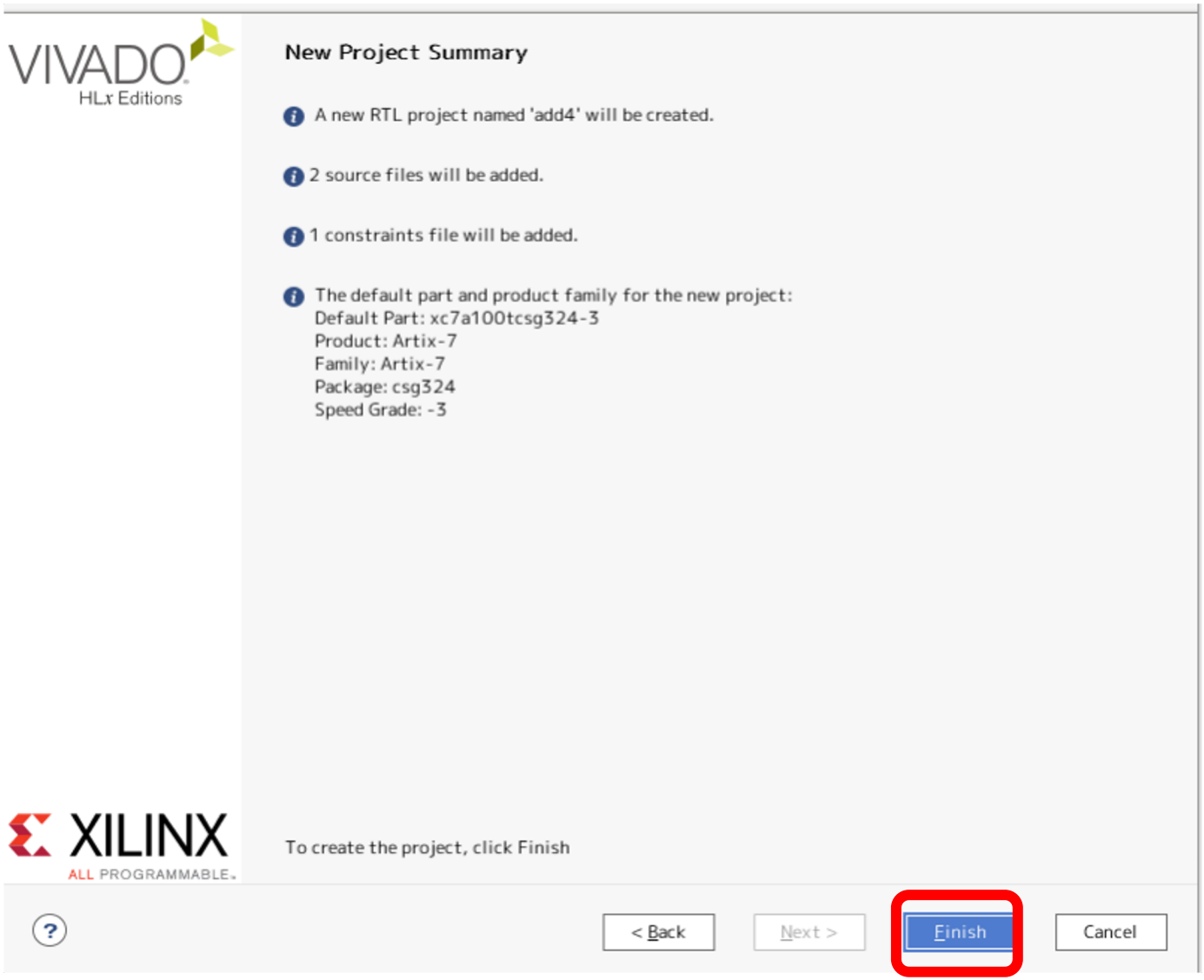


デバイスの指定: XC7A100T-CSG324を選択

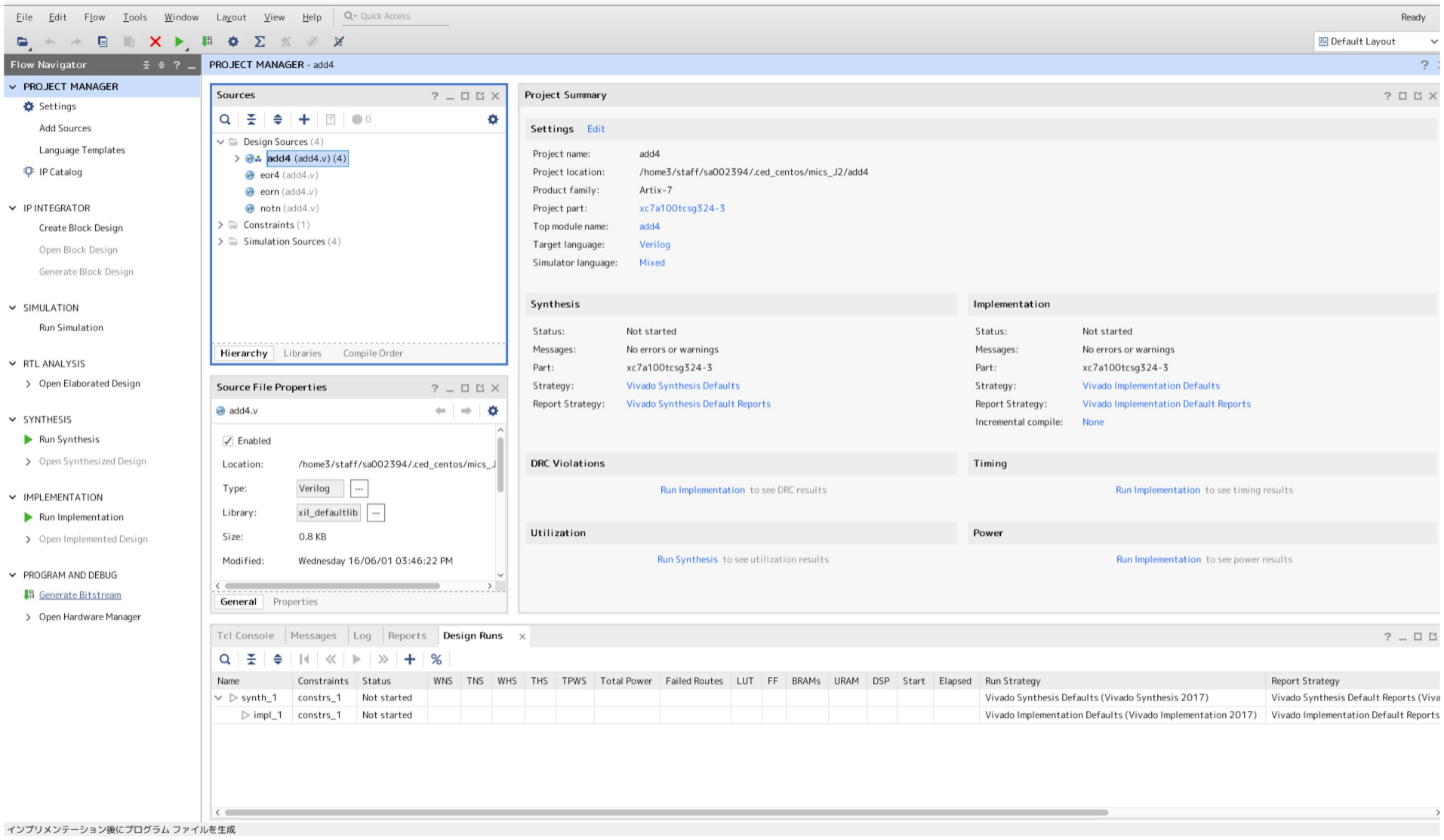
- Family: Artix-7
- Package: csg324
- Speed grade: -3



設定内容を確認

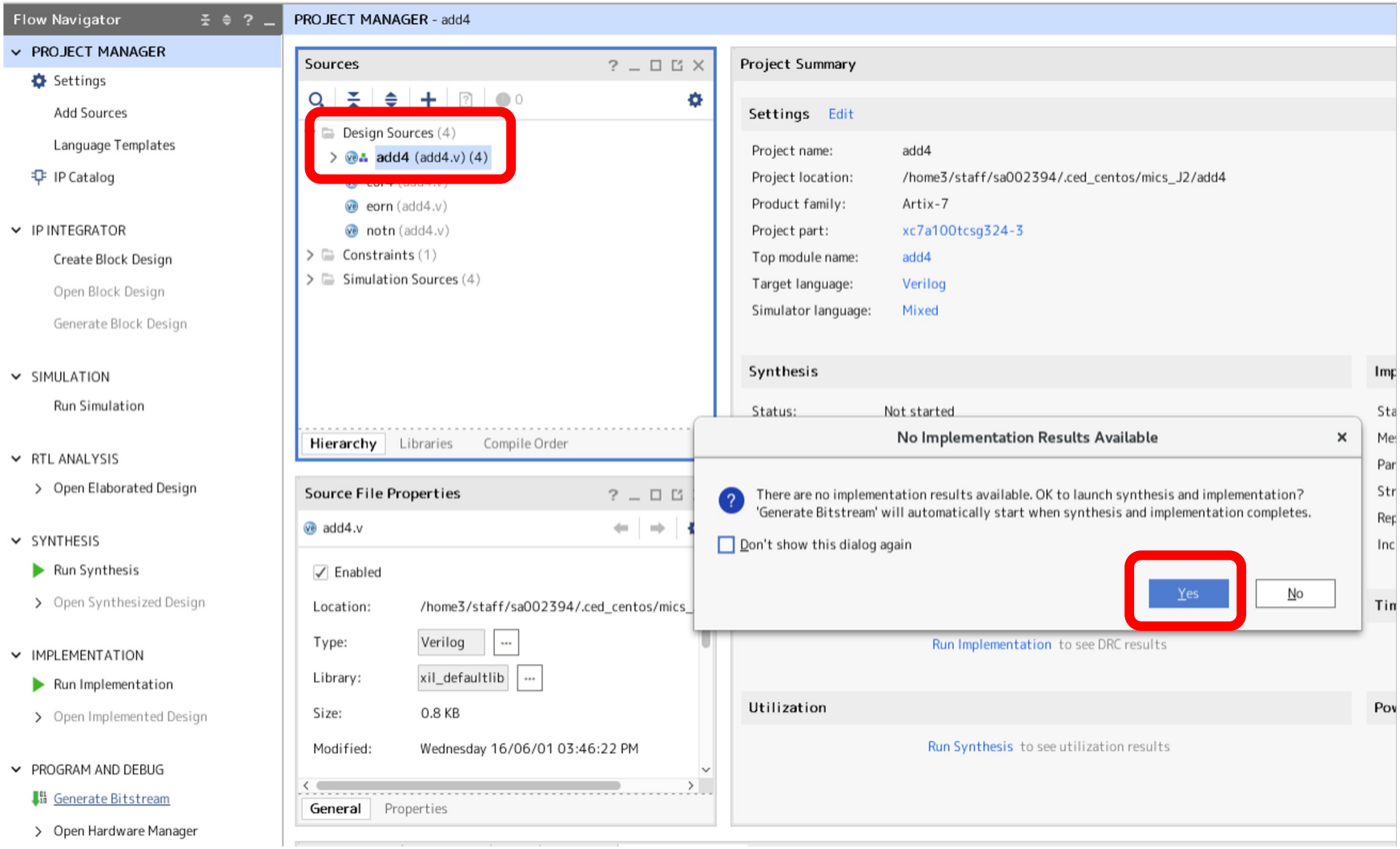

VIVADO
HLx Editions
New Project Summary
A new RTL project named 'add4' will be created.
2 source files will be added.
1 constraints file will be added.
The default part and product family for the new project:
Default Part: xc7a100tcsg324-3
Product: Artix-7
Family: Artix-7
Package: csg324
Speed Grade: -3
XILINX
ALL PROGRAMMABLE.
To create the project, click Finish
< Back
Next >
Finish
Cancel

Vivadoプロジェクトトップページ



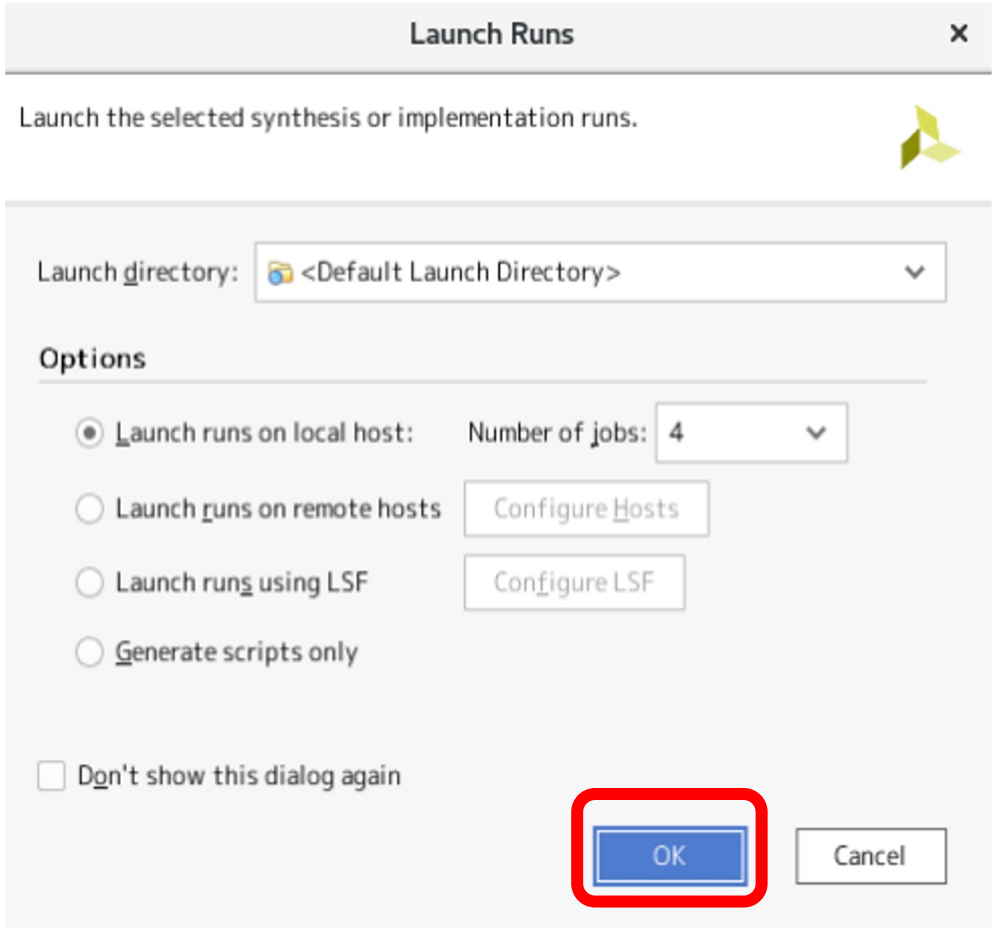
コンパイルの実行

Sources ウィンドウで add4 を選んで Generate Bitstream を実行し Yes をクリック



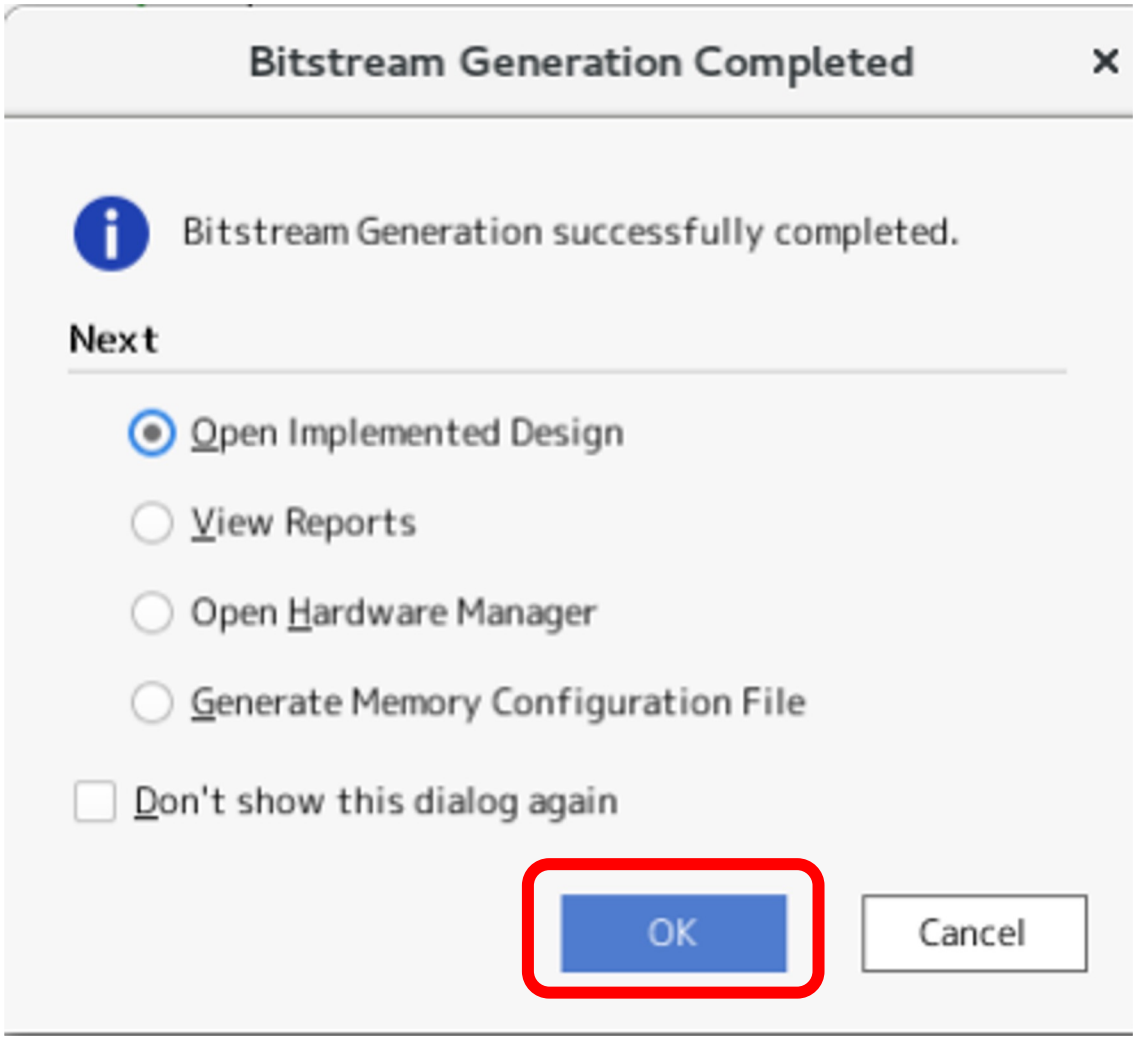
コンパイルの実行

OKをクリック



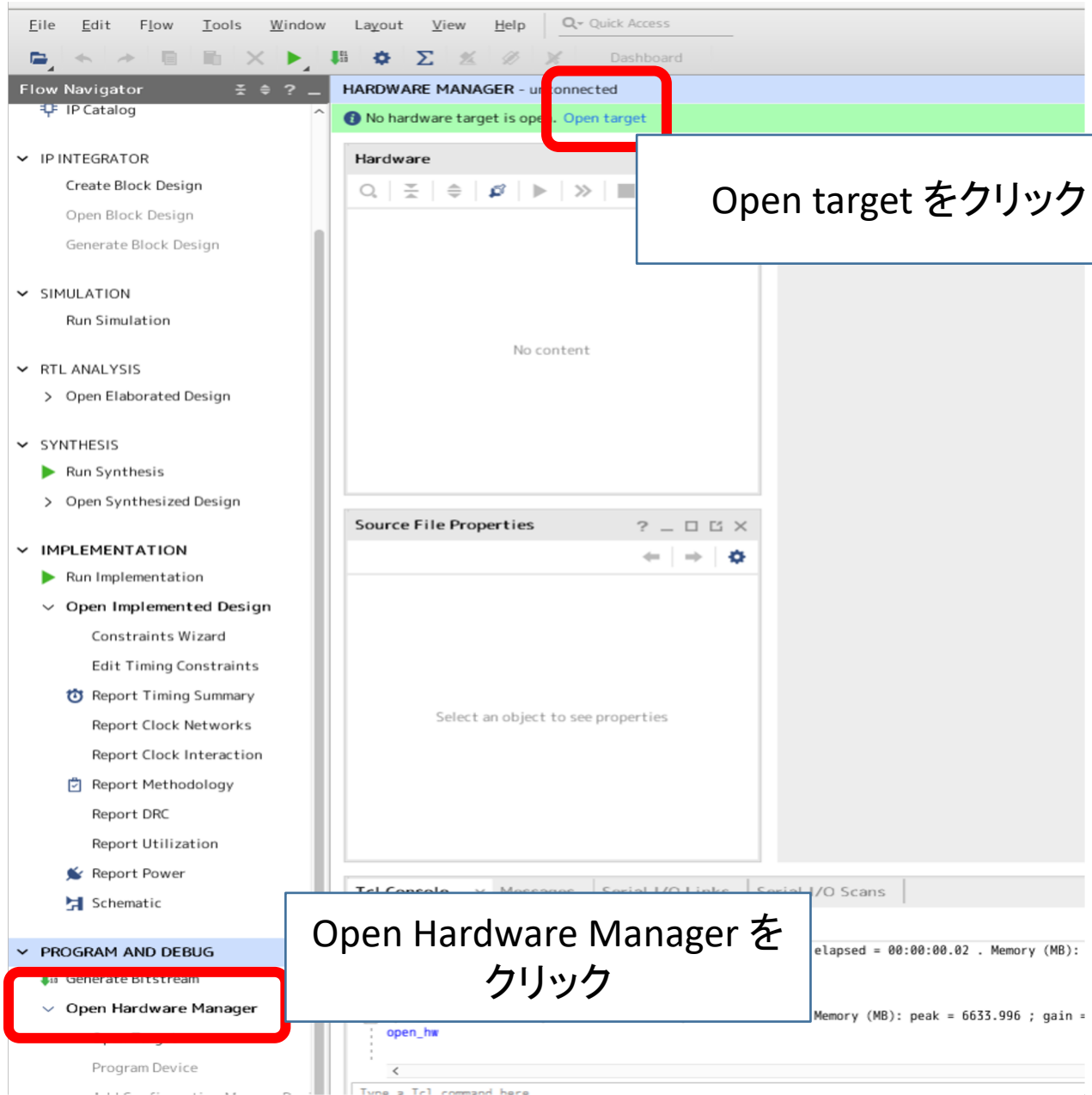
コンパイル成功

Bitstream Generation successfully completed がでたら成功

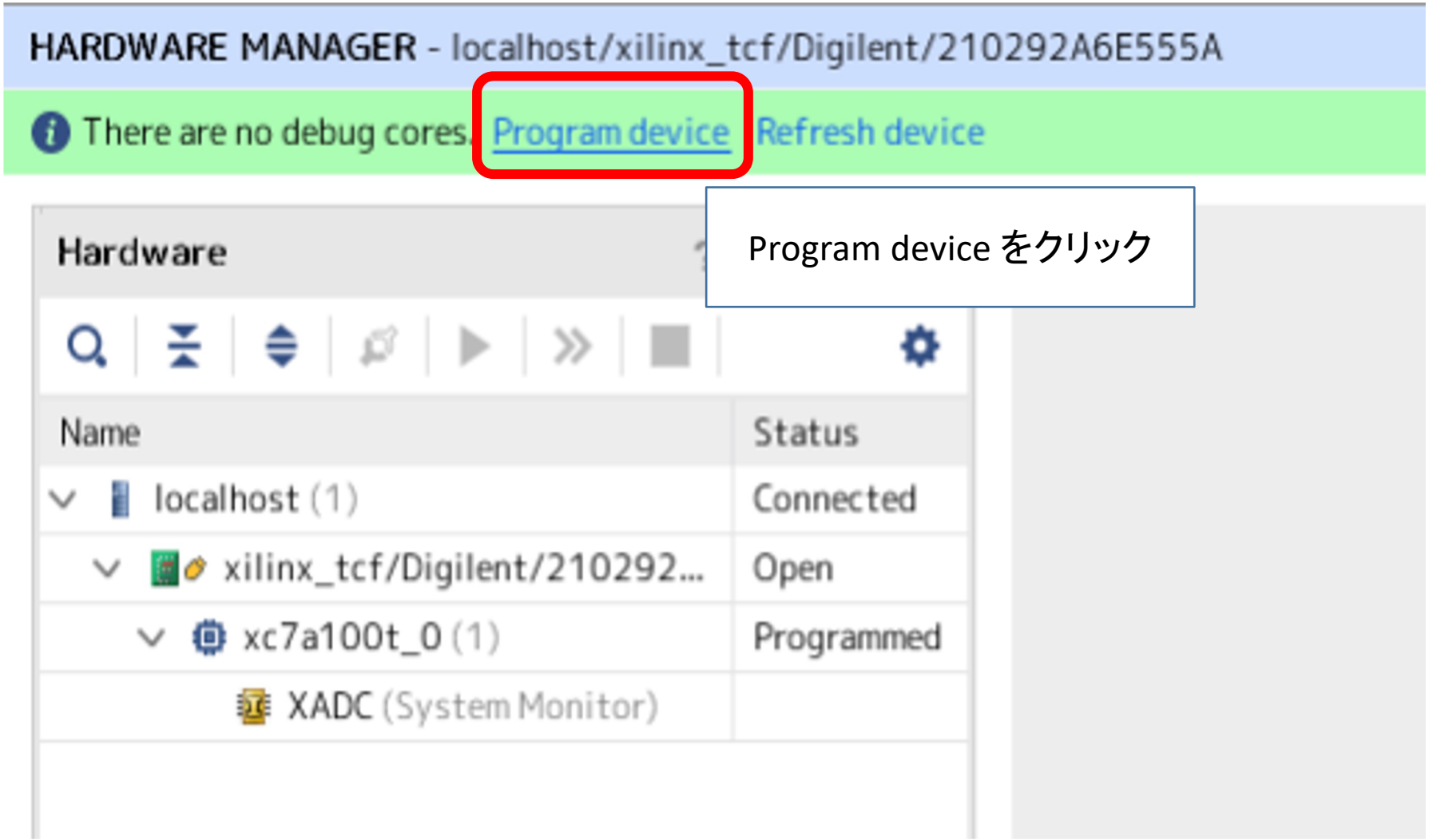


FPGAの電源を入れる

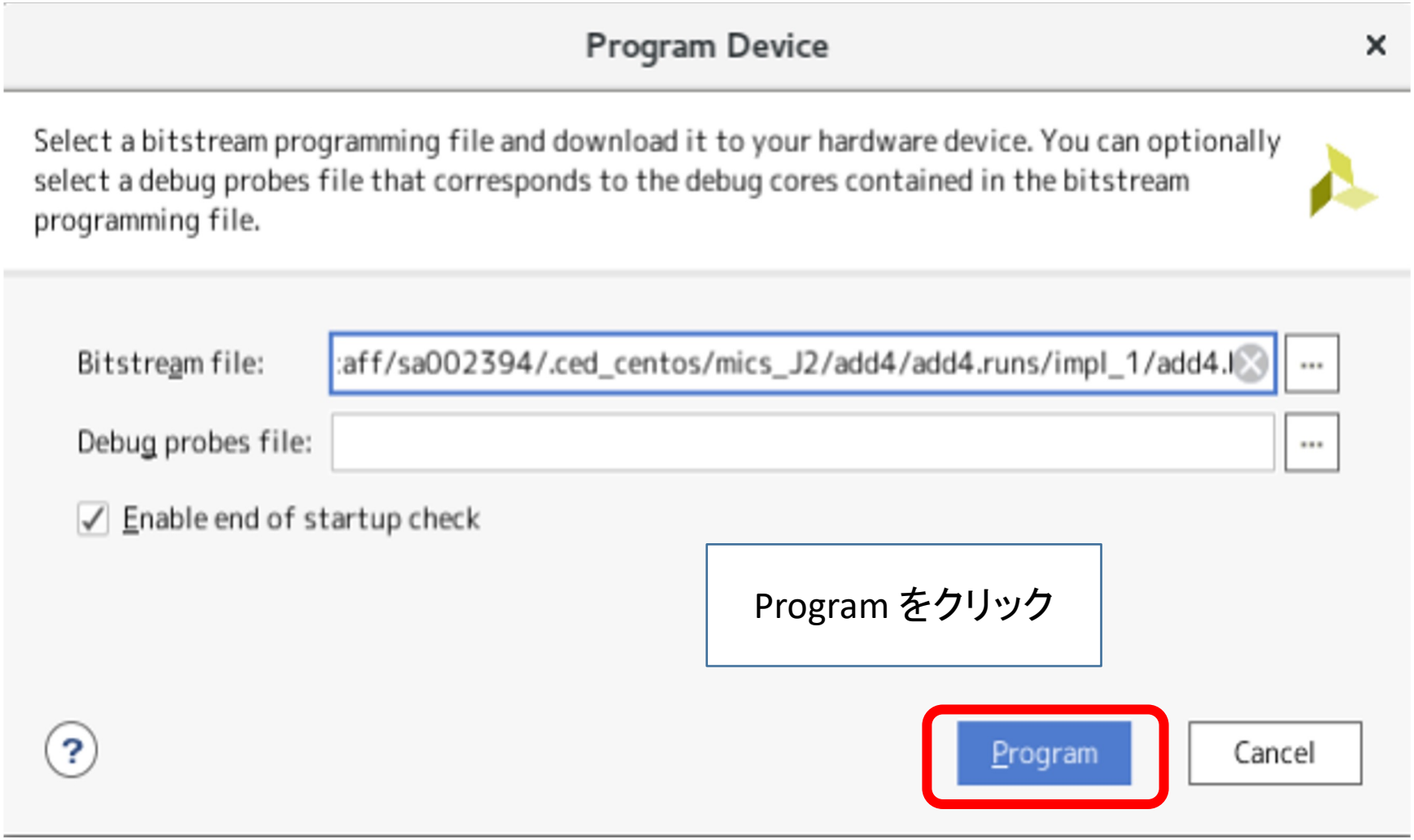
FPGAへ書き込み


File Edit Flow Tools Window Layout View Help
Quick Access
Dashboard
Flow Navigator
IP Catalog
IP INTEGRATOR
Create Block Design
Open Block Design
Generate Block Design
SIMULATION
Run Simulation
RTL ANALYSIS
Open Elaborated Design
SYNTHESIS
Run Synthesis
Open Synthesized Design
IMPLEMENTATION
Run Implementation
Open Implemented Design
Constraints Wizard
Edit Timing Constraints
Report Timing Summary
Report Clock Networks
Report Clock Interaction
Report Methodology
Report DRC
Report Utilization
Report Power
Schematic
PROGRAM AND DEBUG
Generate Bitstream
Open Hardware Manager
Program Device
HARDWARE MANAGER - unconnected
No hardware target is ope. Open target
Hardware
No content
Source File Properties
Select an object to see properties
Tcl Console
Messages
Serial I/O Links
Serial I/O Scans
elapsed = 00:00:00.02 . Memory (MB):
Memory (MB): peak = 6633.996 ; gain =
open_hw
Type a Tcl command here
Open target をクリック
Open Hardware Manager をクリック

FPGAへ書き込み


HARDWARE MANAGER - localhost/xilinx_tcf/Digilent/210292A6E555A
There are no debug cores.
Program device
Refresh device
Program device をクリック
Hardware
Name
Status
localhost (1)
Connected
xilinx_tcf/Digilent/210292...
Open
xc7a100t_0 (1)
Programmed
XADC (System Monitor)

FPGAへ書き込み


Program Device
Select a bitstream programming file and download it to your hardware device. You can optionally select a debug probes file that corresponds to the debug cores contained in the bitstream programming file.
Bitstream file:
:aff/sa002394/.ced_centos/mics_J2/add4/add4.runs/impl_1/add4.l
Debug probes file:
Enable end of startup check
Program をクリック
Program
Cancel

add4の動作確認

例：0011 + 0001 = 0100