Verilogの基礎その2

ISHI会
フルデジタルで音声を操るためのロジック回路 2回目

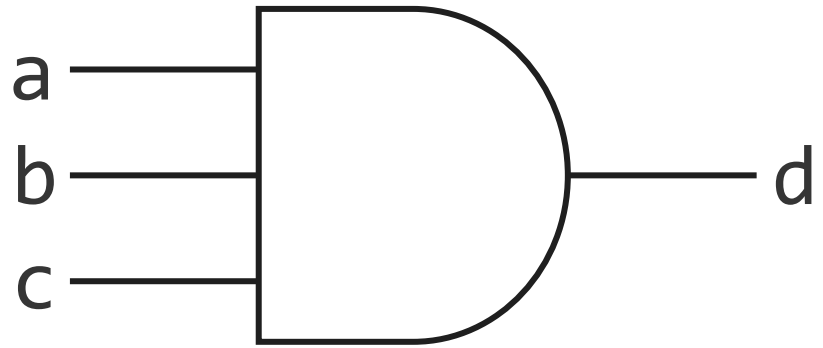
目次

- Verilog文法学習の続き
  - モジュール呼び出し
  - バス表記
  - リダクション演算
  - 三項演算子と数値リテラル
  - ビット連接
  - always文
    - ✓reg
    - ✓if
    - ✓case文
- 回路演習
- 練習問題

前回の回答例

3入力AND回路のVerilogコード

```
module three_and(
input wire a,
input wire b,
input wire c,
output wire d);

  assign d = a & b & c;
endmodule
```



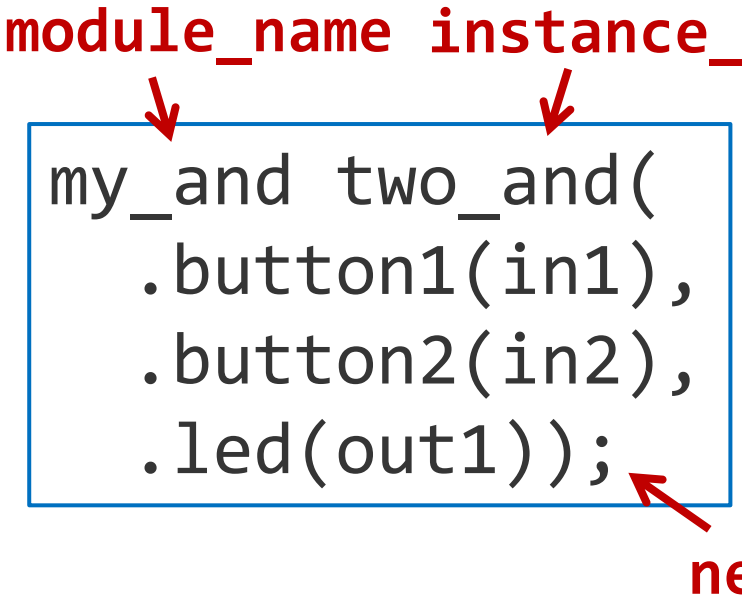
モジュール呼び出しの方法

モジュール呼び出し

Verilogではモジュール内でモジュールを呼び出すことができる。
モジュールは次のように呼び出す。

```
module_name instance_name (net);
```

module_nameは呼び出したいモジュールの名前、instance_nameはインスタンス名、netはモジュールへの配線を記述する。1週目のmy_andを呼び出すには以下のように記述する。



netの記述ルール

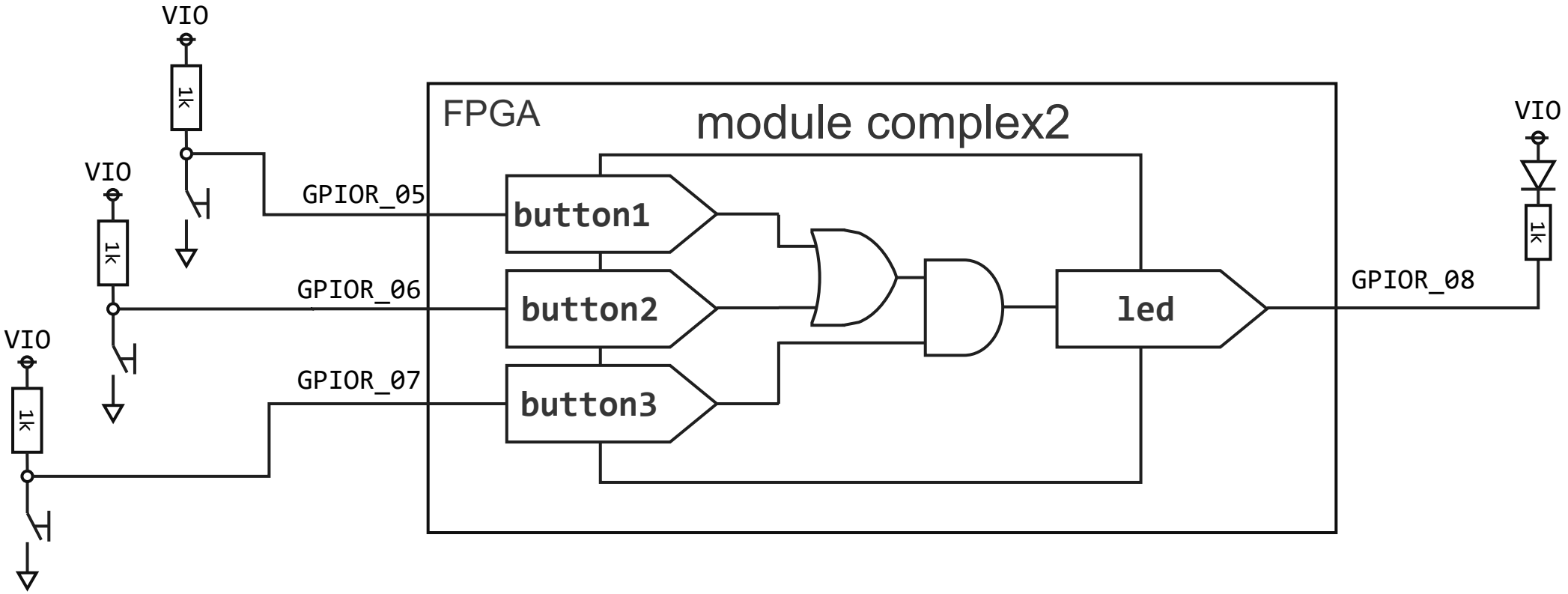
呼び出すモジュールのネット定義名

```
.a(in1)
```

呼び出し元モジュールのラベル

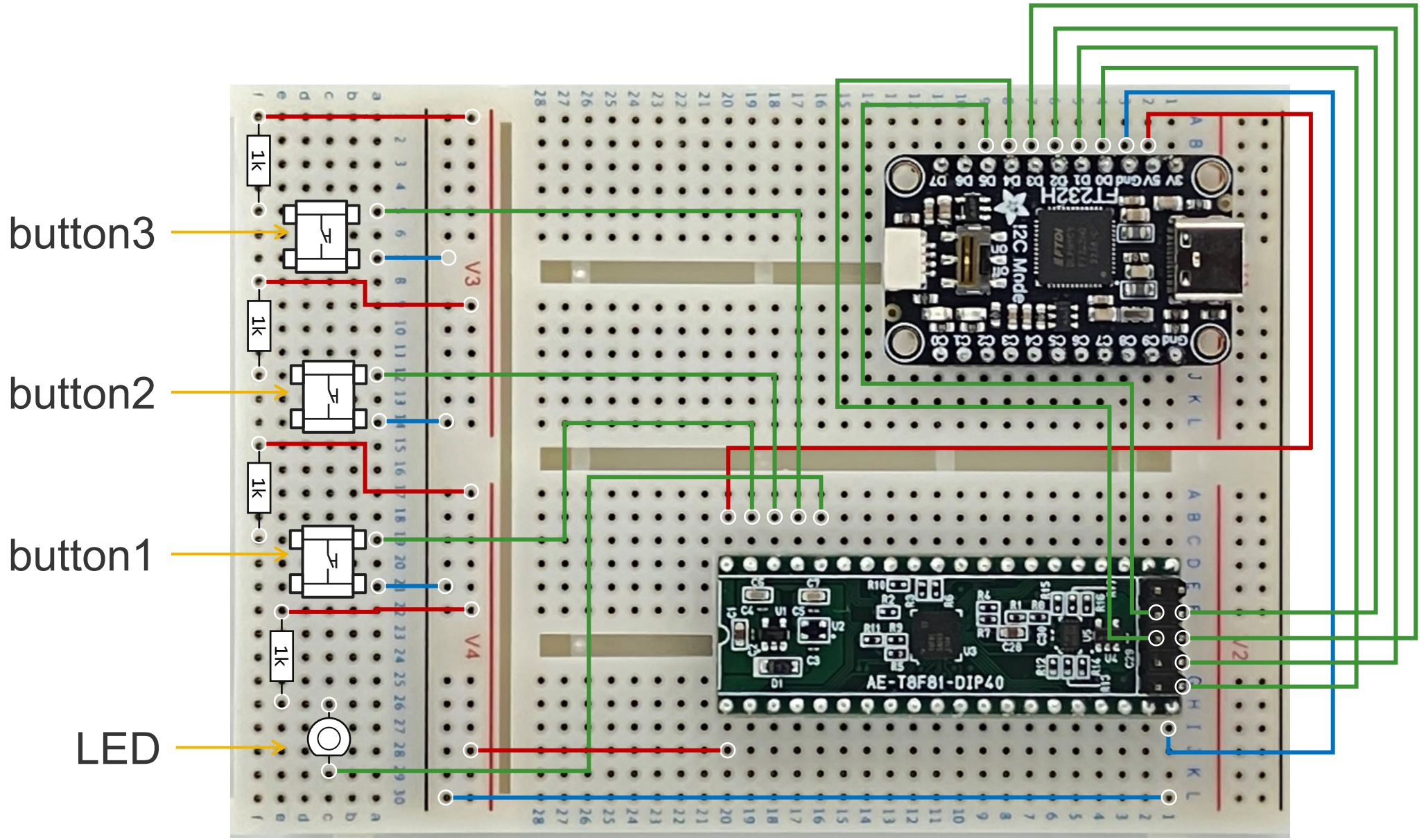
```
module my_and(
  input wire button1,
  input wire button2,
  output wire led);

  assign led = button1 & button2;
endmodule
```

モジュール呼び出しを使った組み合わせ回路



3個のボタンはGPIOR_05とGPIOR_06、GPIOR_07に接続する。
LEDはGPIOR_8に接続する。

配線図


button3
button2
button1
LED
1k
1k
1k
1k
V3
V4
AE-T8F81-DIP40
FT232H

モジュール呼び出しの練習

andとorを組み合わせた回路を作成する。
1回目に作成したmy_andとmy_orを使用する。

※Top Module/Entitiyをcomplex2に変更するのを忘れずに

```
1  module complex2(
2  input wire button1,
3  input wire button2,
4  input wire button3,
5  output wire led);
6
7  wire a_or_b;
8
9  my_or two_or
10 (
11     .button1(button1),
12     .button2(button2),
13     .led(a_or_b)
14 );
15
16 my_and two_and
17 (
18     .button1(button3),
19     .button2(a_or_b),
20     .led(led)
21 );
22
23 endmodule
```

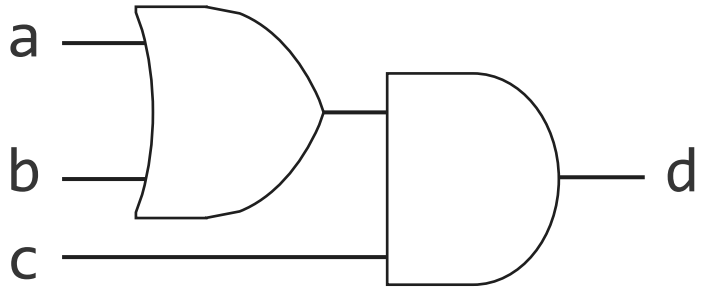
my_orを呼び出す

orの結果はa_or_bに接続

my_andを呼び出す

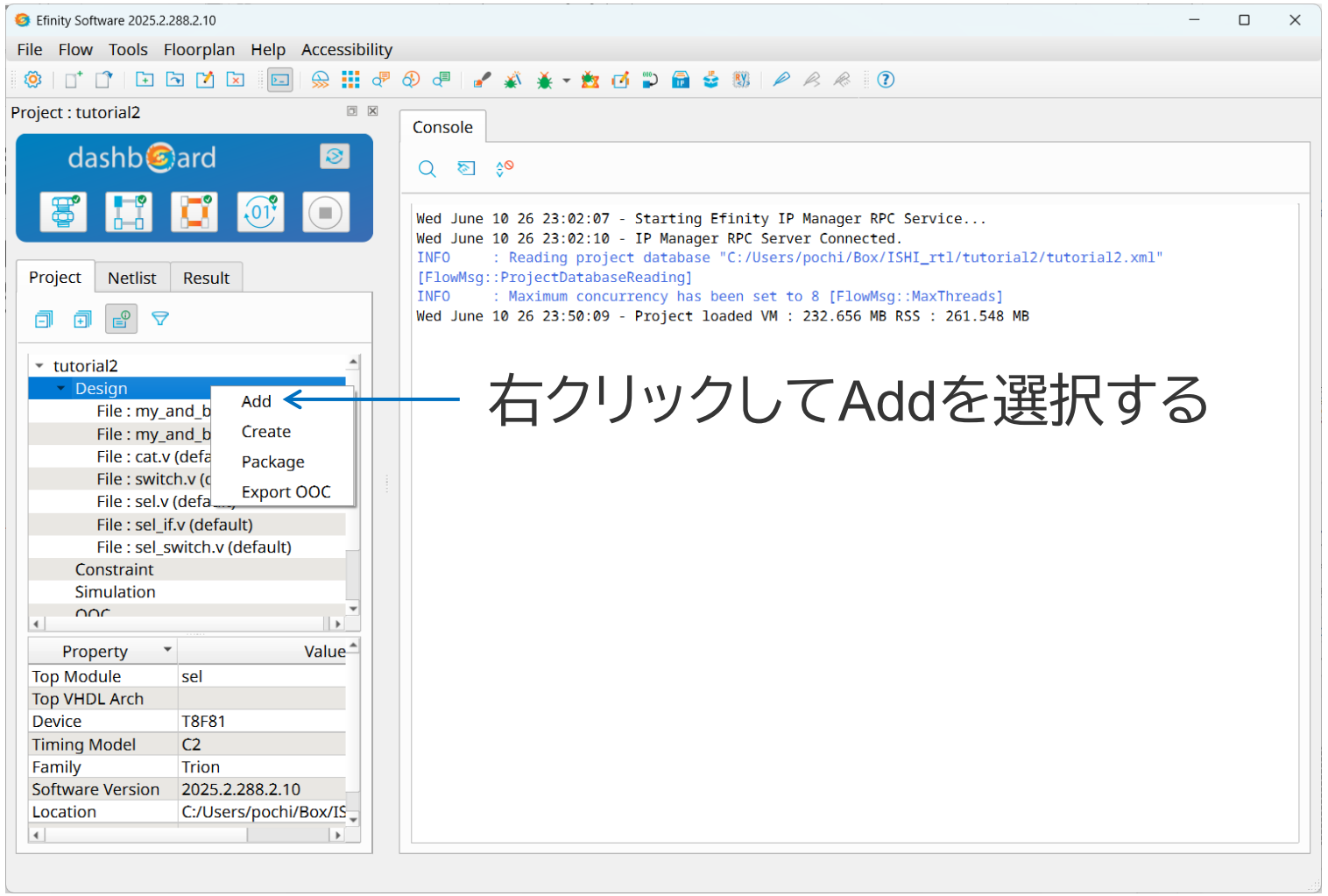
my_andの2つ目の入力はa_or_bにする。

出力はledに接続



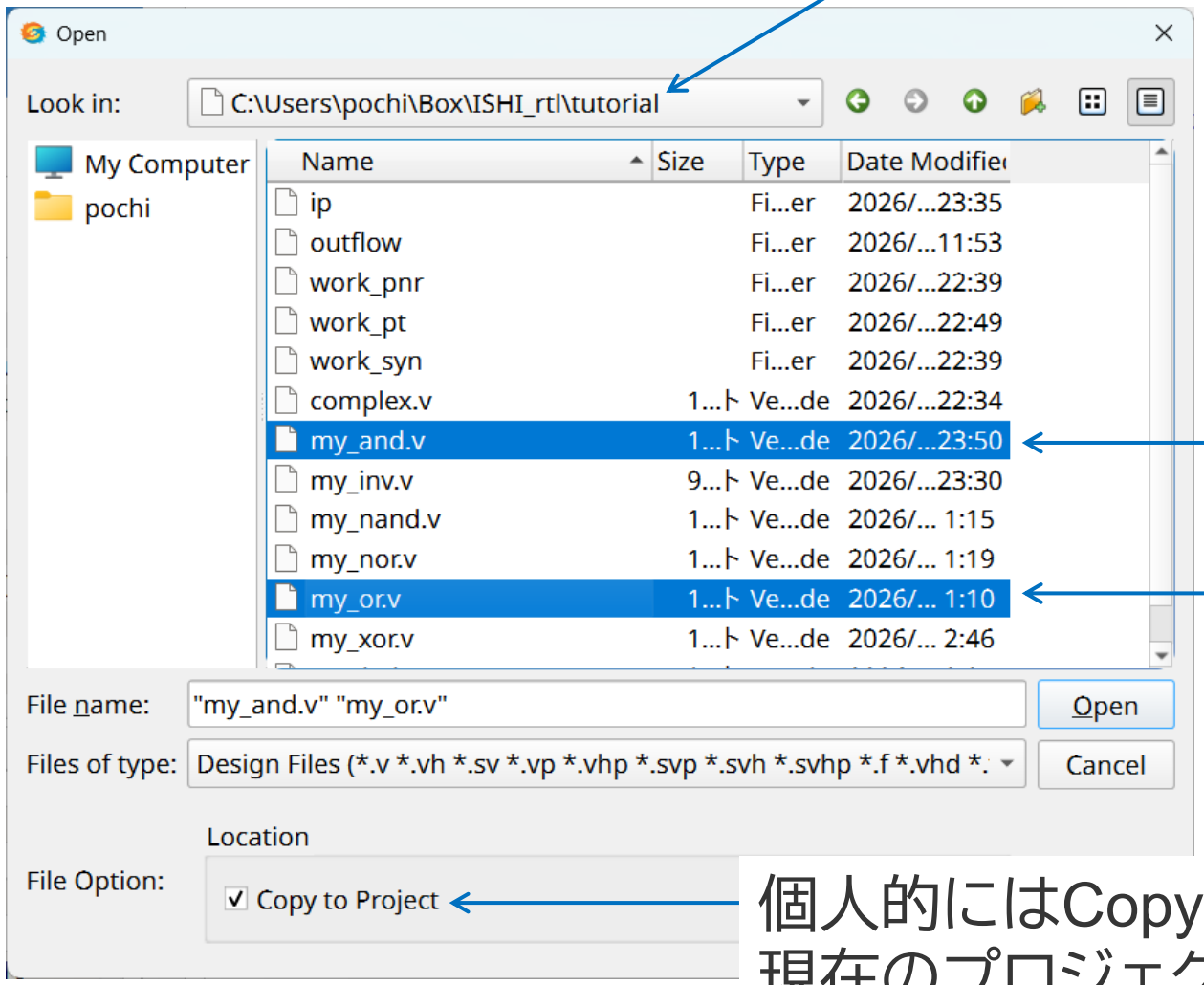
ファイルの読み込み1

※ 前回とは別のプロジェクトを作成した場合に必要な手順。別のプロジェクトから他のVerilogを読み込むときにも必要。



ファイル読み込み2

読み込みたいVerilogファイルのパスに移動

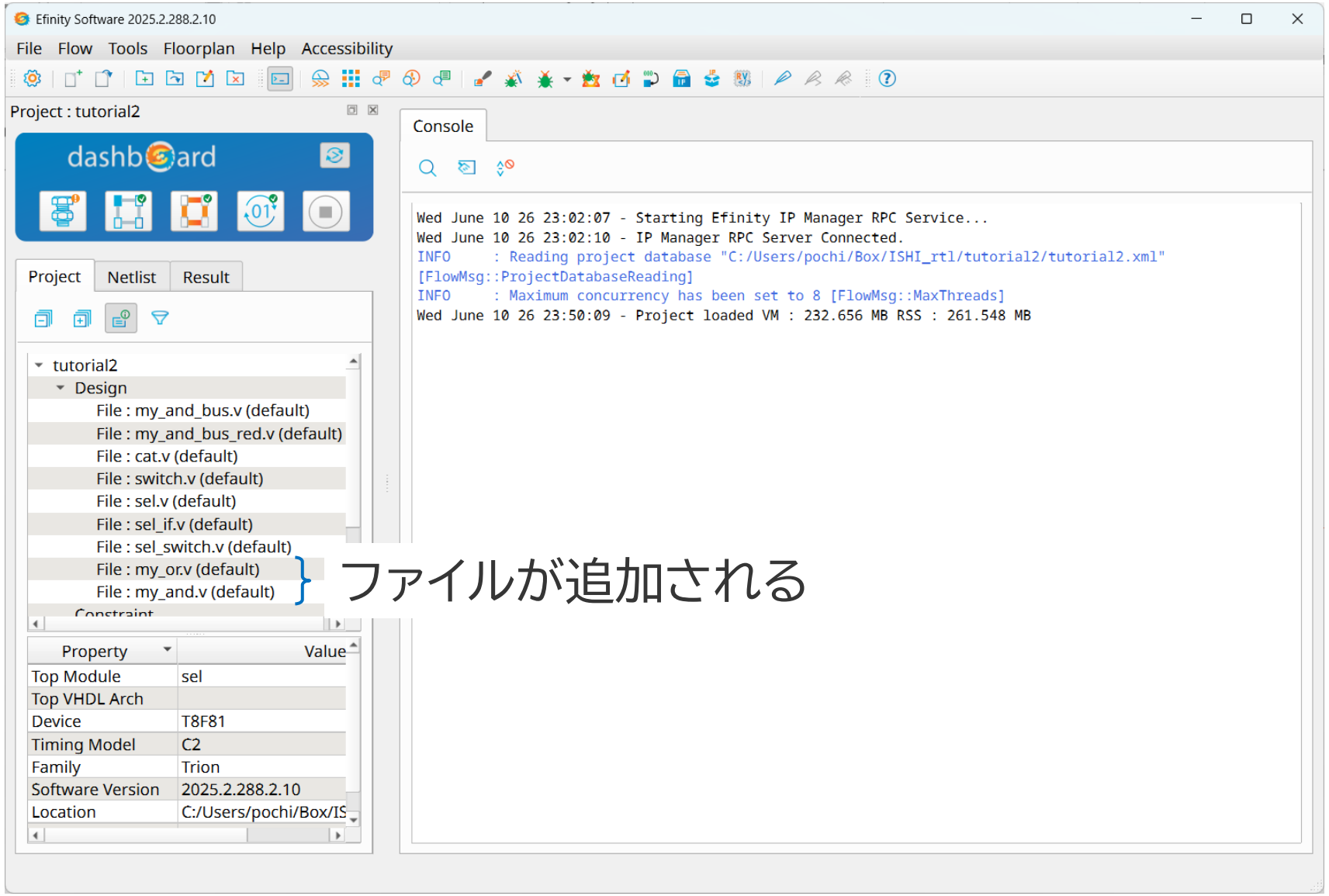


my_and.vを選択

Ctrlを押しながら、my_or.vを選択

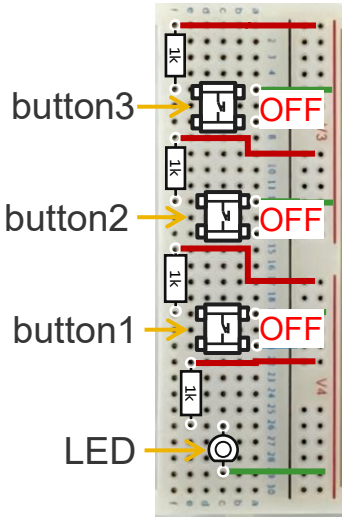
個人的にはCopy to Projectにチェックを入れるのがおすすめ
現在のプロジェクトにファイルがコピーされる

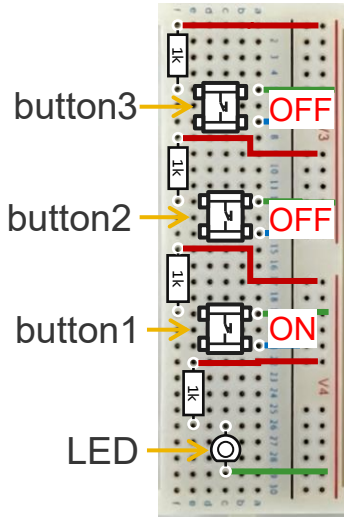
ファイル読み込み3

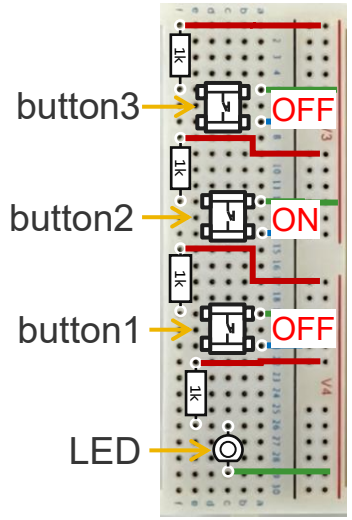


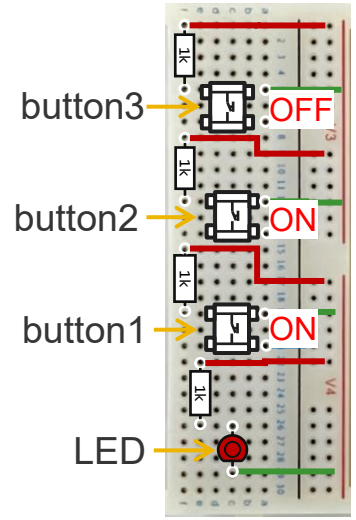
ファイルが追加される

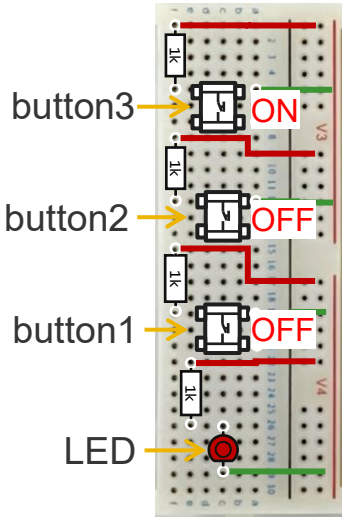
動作

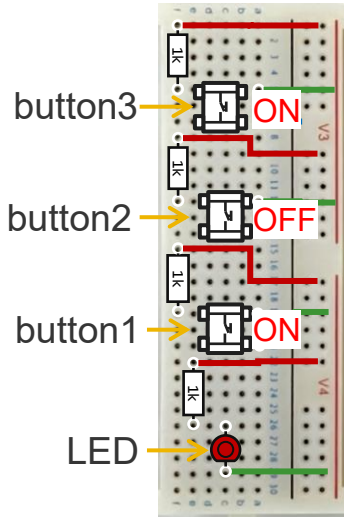

button3
OFF
button2
OFF
button1
OFF
LED

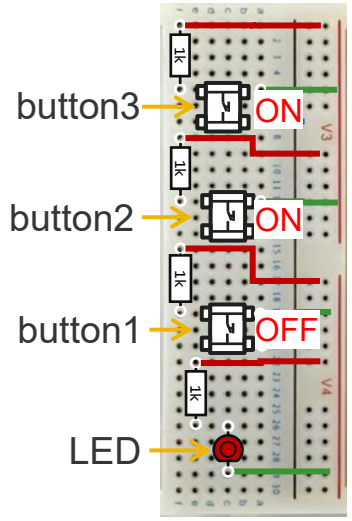

button3
OFF
button2
OFF
button1
ON
LED

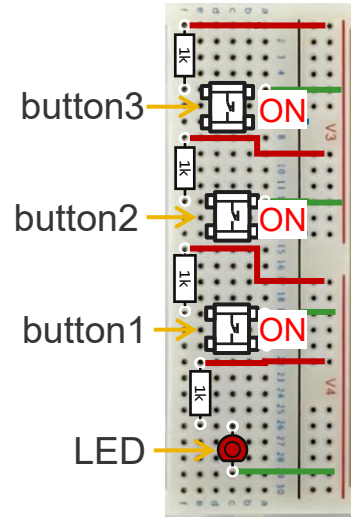

button3
OFF
button2
ON
button1
OFF
LED


button3
OFF
button2
ON
button1
ON
LED


button3
ON
button2
OFF
button1
OFF
LED


button3
ON
button2
OFF
button1
ON
LED


button3
ON
button2
ON
button1
OFF
LED


button3
ON
button2
ON
button1
ON
LED

バス表記

バス

```
input wire [3:0] button
```

このような記述を**バス**という。複数の配線を束ねて、配列(リスト)のように扱える。
バスの1つの配線を指定するには、

```
button[2]
```

のように記述する。複数の連続した配線を指定するには、

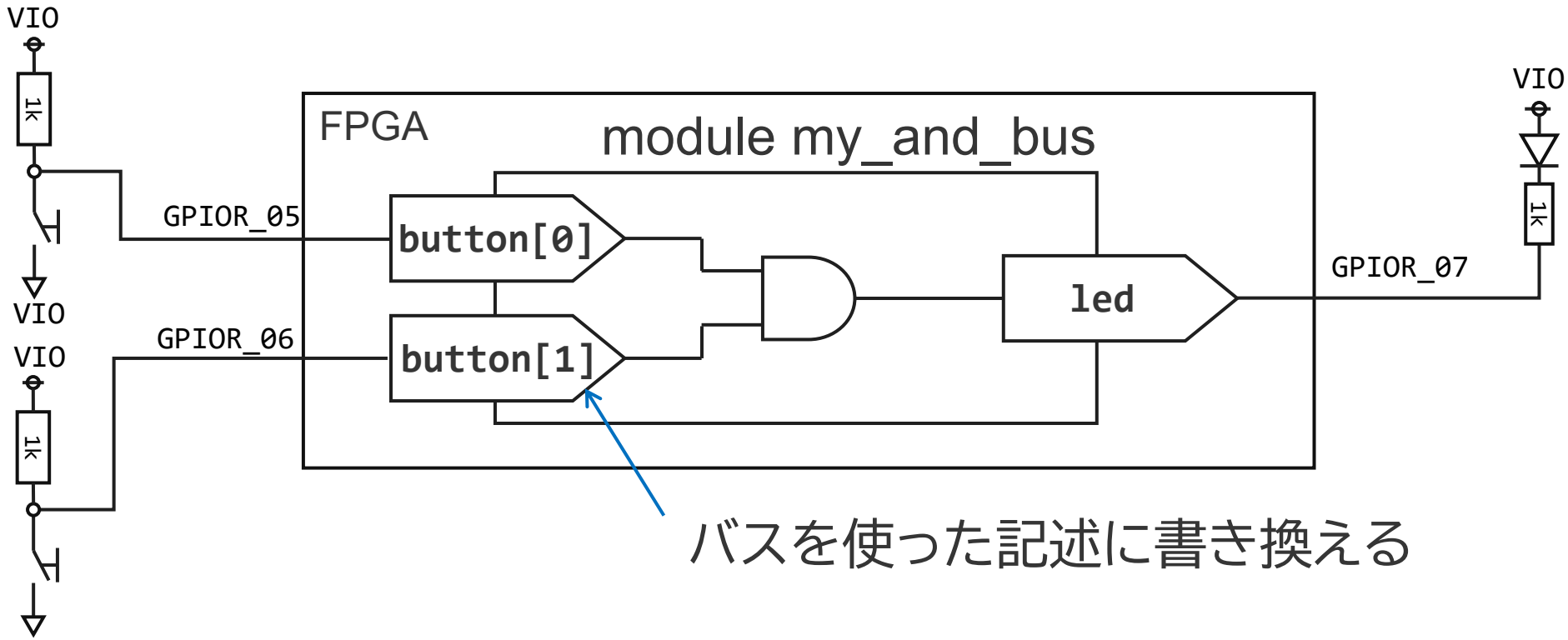
```
button[2:1]
```

のように記述する。
なお、バスはゼロから始まっている必要はなく、以下のような定義も可能。

```
input wire [4:1] button
```

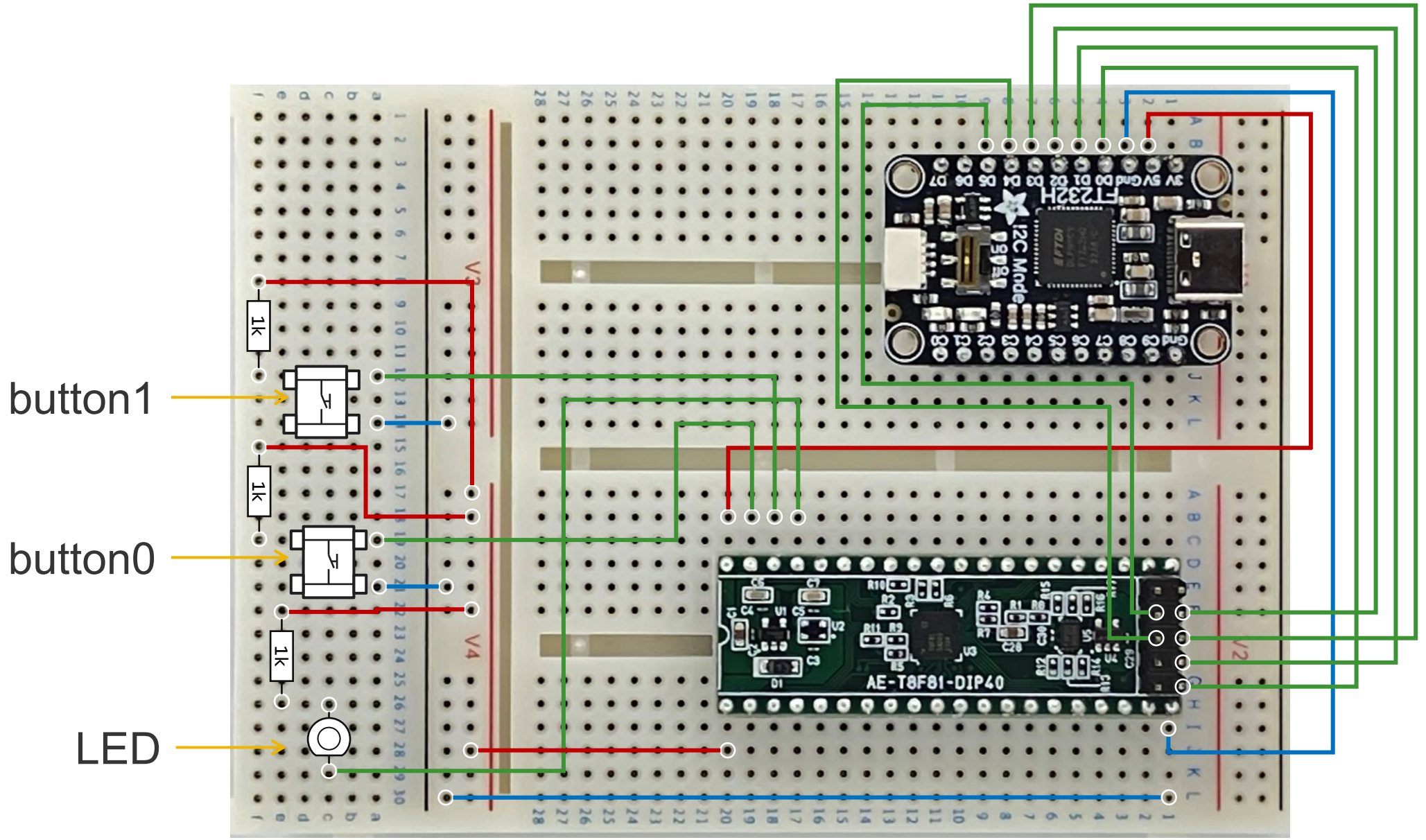
バスを使ったANDゲート

2入力ANDゲートとして動作する回路をFPGAに実装する。
入力として2つのボタンと1つのLEDを有する。



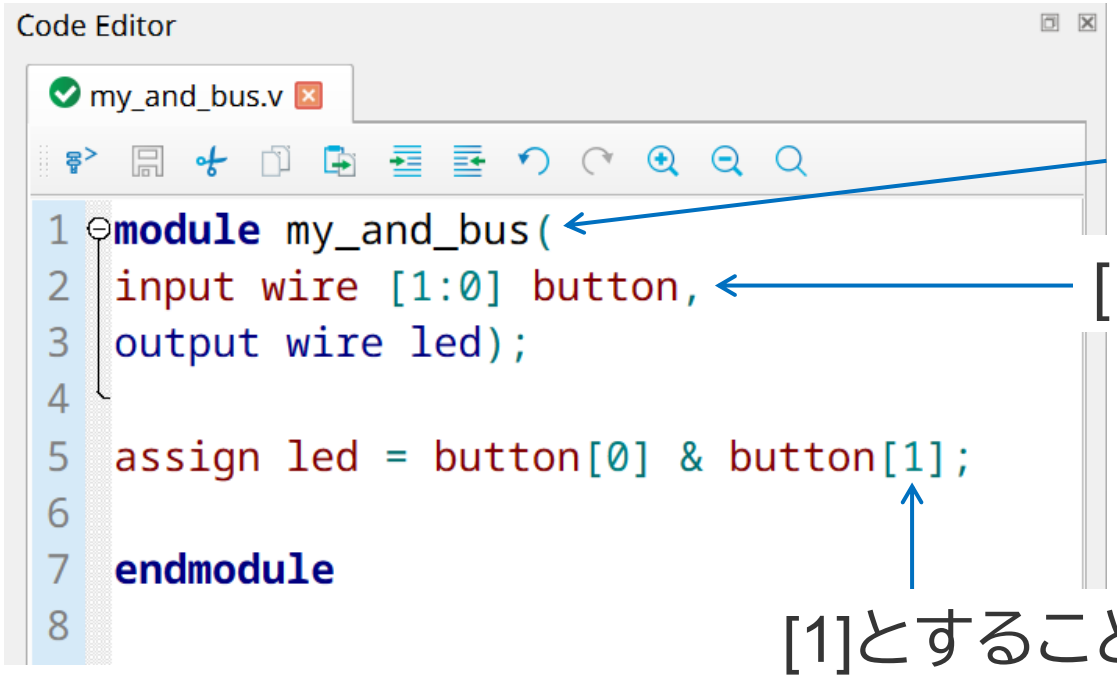
2個のボタンはGPIOR_05とGPIOR_06、LEDはGPIOR_07に接続する。

配線図



バスを使ったANDゲートのVerilogコード

※Top Module/Entitiyをmy_and_busに
変更するのを忘れずに



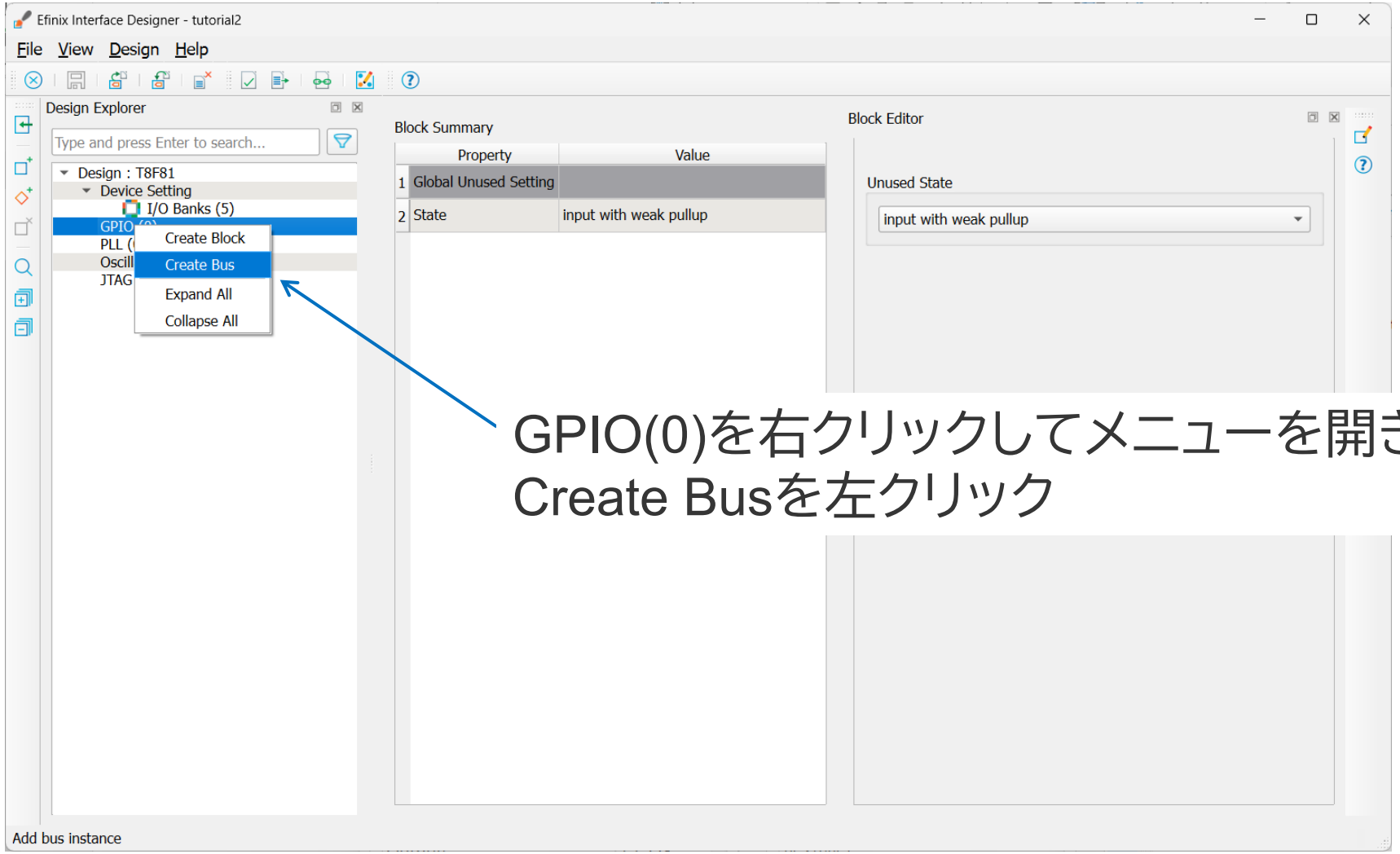
モジュール名はmy_and_bus

[1:0]とすることで2ビット分のバスが定義される

[1]とすることでバスの2個目の配線にアクセスできる

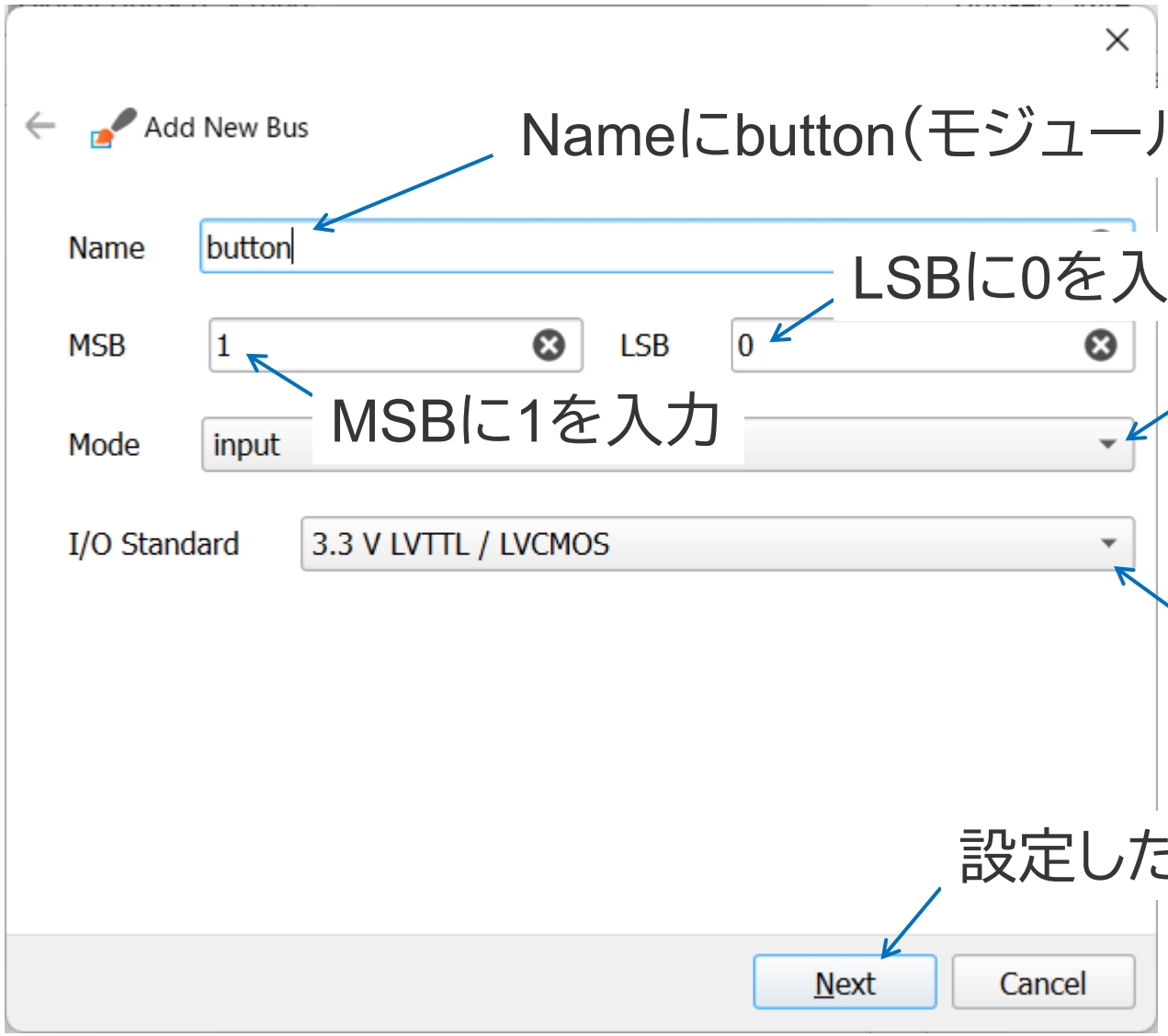
バスのピンアサイン

Interface Designerを開く



GPIO(0)を右クリックしてメニューを開き
Create Busを左クリック

バスのピンアサイン2



Nameにbutton(モジュール定義で指定したポート名)を入力

LSBに0を入力

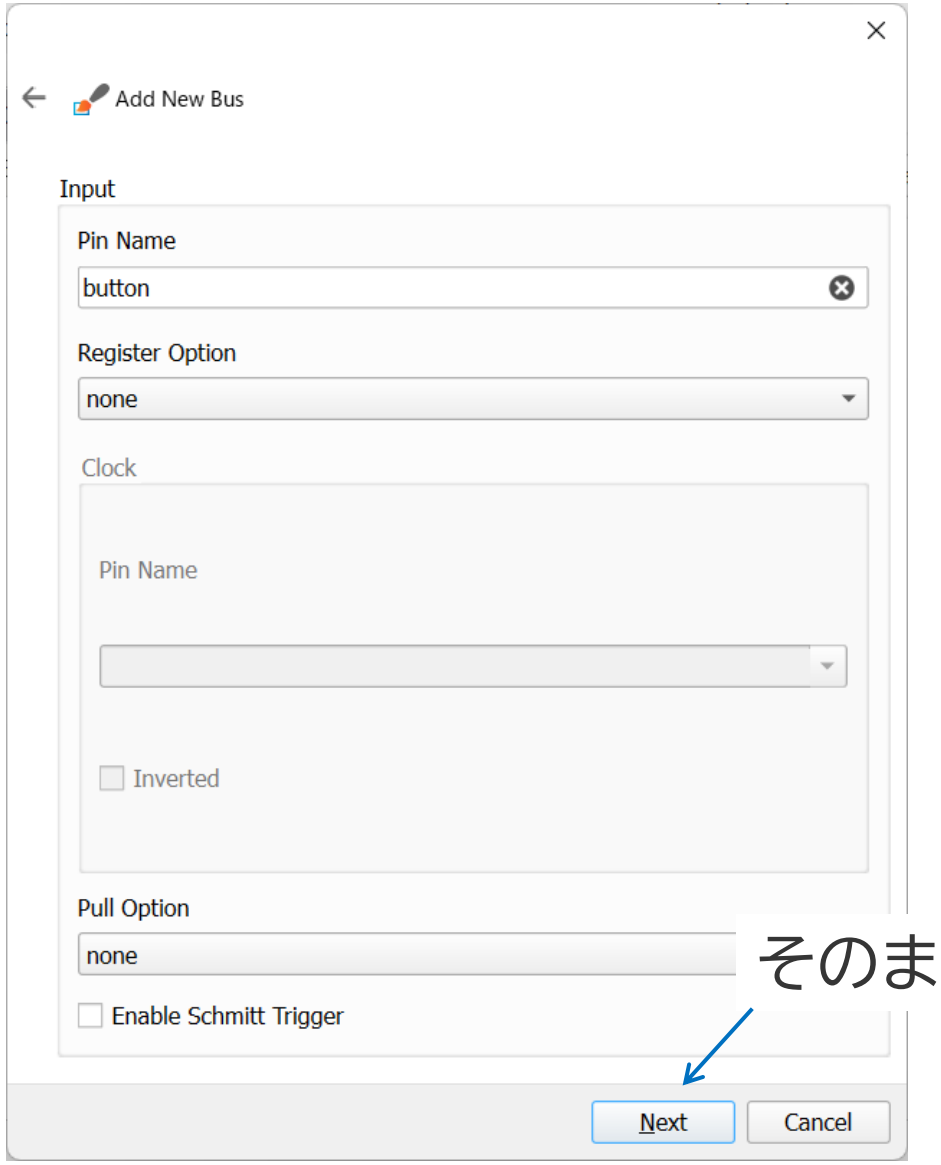
MSBに1を入力

Modeはinputにする

I/O Standardは3.3V LVTT/LVCMOSにする

設定したらNextをクリック

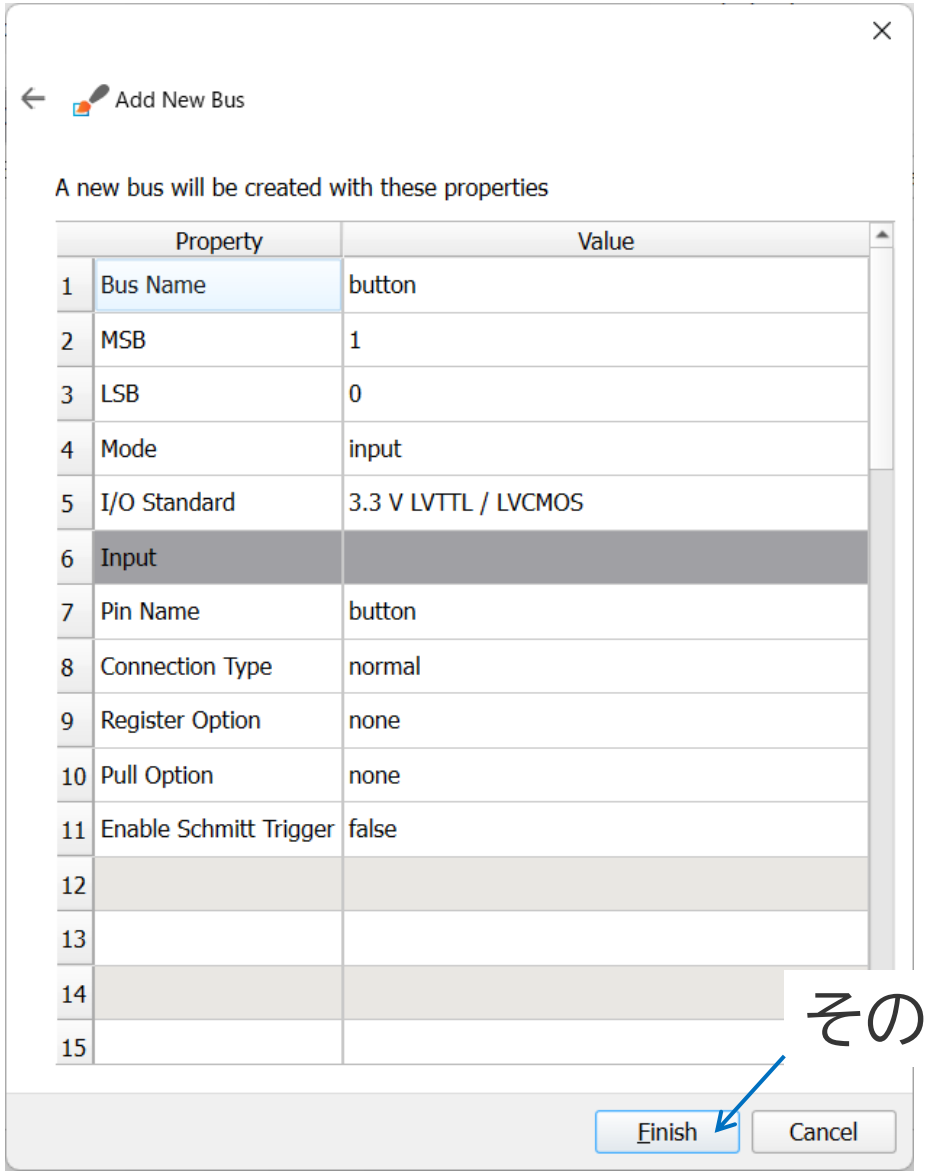
バスのピンアサイン3



ここは変更無しでよい

そのままNextを左クリック

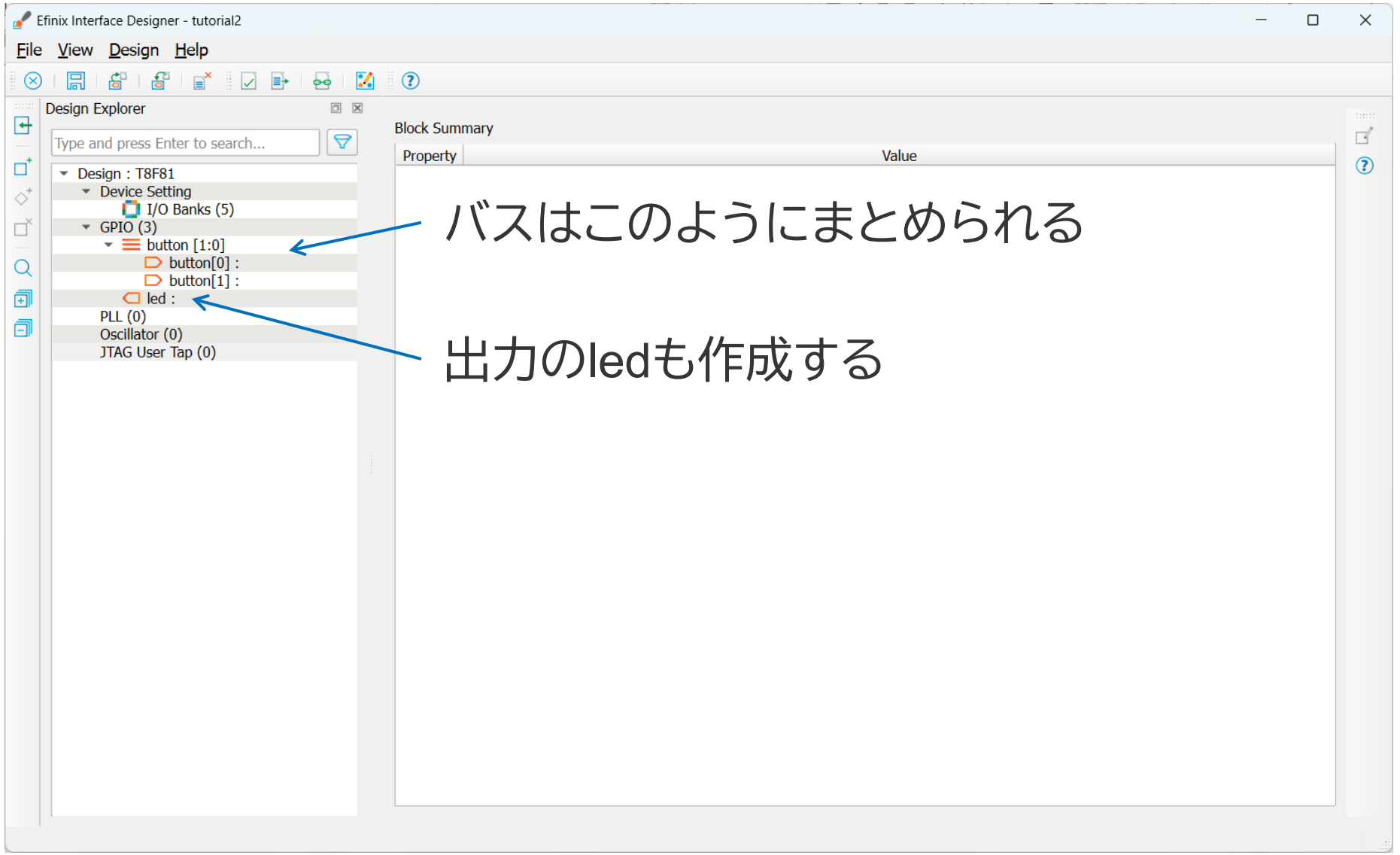
バスのピンアサイン4



確認画面が出るので間違いが無ければ
そのままFinishを左クリック

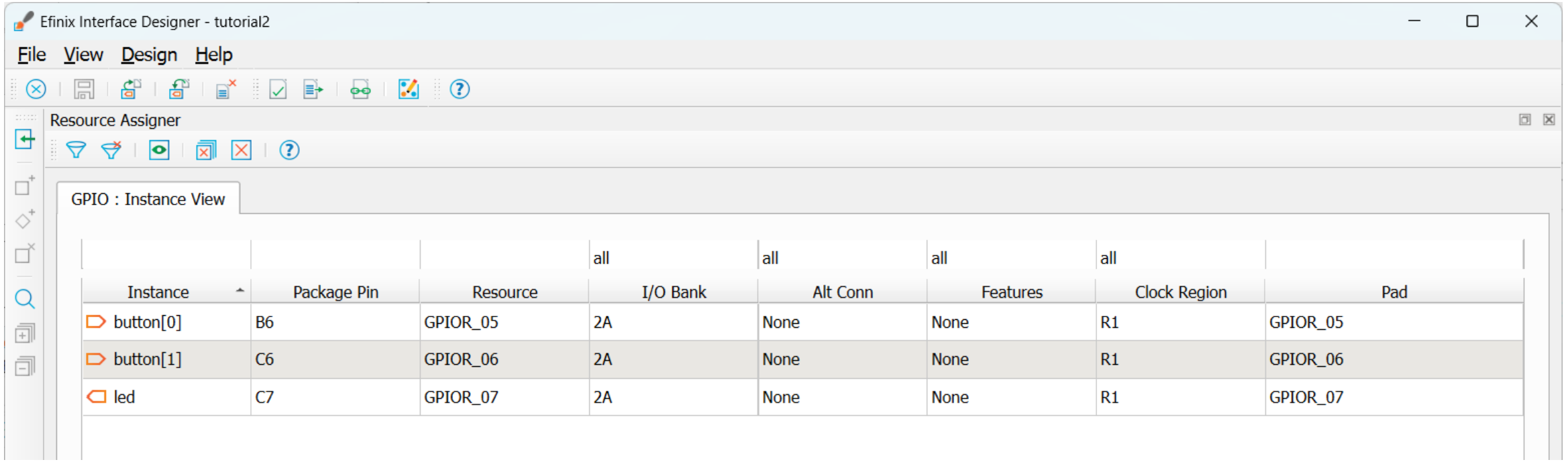
そのままFinishを左クリック

ピンアサイン1


Efinix Interface Designer - tutorial2
File View Design Help
Design Explorer
Type and press Enter to search...
Design : T8F81
Device Setting
I/O Banks (5)
GPIO (3)
button [1:0]
button[0] :
button[1] :
led :
PLL (0)
Oscillator (0)
JTAG User Tap (0)
Block Summary
Property
Value
バスはこのようにまとめられる
出力のledも作成する

ピンアサイン2

Design > Show/Hide GPIO Resource AssignerからResource Assignerを開く
次のように設定する。



リダクション演算

リダクション演算

```
wire [3:0] a;
wire b;
assign b = &a;
```

上記の&aは正しい文法である。&はバスに対しては1つでも作用できリダクションという。

リダクションはバスの全要素をオペランドとする演算を行う。

上記の例では、a[0] & a[1] & a[2] & a[3]と等価になる。

リダクションはAND、OR、ExORに対して適用できる。

ANDゲートのリダクションへの書き直し

Code Editor

my_and_bus.v | my_and_bus_red.v

```
1  module my_and_bus_red(
2  input wire [1:0] button,
3  output wire led);
4
5  assign led = &button;
6
7  endmodule
8
```

※Top Module/Entitiyをmy_and_bus_red
に変更するのを忘れずに

モジュール名はmy_and_bus_red

buttonの後ろには何も付けずに、先頭に＆を付ける。

GPIO : Instance View

			all	all	all	all	
Instance	Package Pin	Resource	I/O Bank	Alt Conn	Features	Clock Region	Pad
button[0]	B6	GPIOR_05	2A	None	None	R1	GPIOR_05
button[1]	C6	GPIOR_06	2A	None	None	R1	GPIOR_06
led	C7	GPIOR_07	2A	None	None	R1	GPIOR_07

三項演算子と数値リテラル

三項演算子

```
<式1> ? <式2> : <式3>;
```

「?」と「:」から構成される演算子を**三項演算子**という。「:」に挟まれた式2と式3のどちらかを、式1の条件によって選択できる。

式1には**条件を表す式**を入れる。条件式には、2つの式の値を比較する「==」や「!=」、複数のビットを数値として比較する「>=」、「>」などが使用できる。

式1がTrueであれば、**「:」の左側である式2**が選択される。

式1がFalseであれば、**「:」の右側である式3**が選択される。

sの入力によってaとbを切り替えるような回路は、

```
assign c = (s == 1'b1) ? a : b;
```

のようにかける。これはsが1のときにaが、0のときにbがcに出力される回路である。

数値リテラル

```
1’b1
```

1ビットの1を表す式として、「1’b1」という表現を使用する。

ここで「b」は2進数を意味する。「b」は他に「o」、「d」、「x」がある。
それぞれ、8進数、10進数、16進数を意味する。

「1’」はビット数を表し、1ビットであることを意味する。
例えば2ビットの2であれば、「2’b10」と表記する。

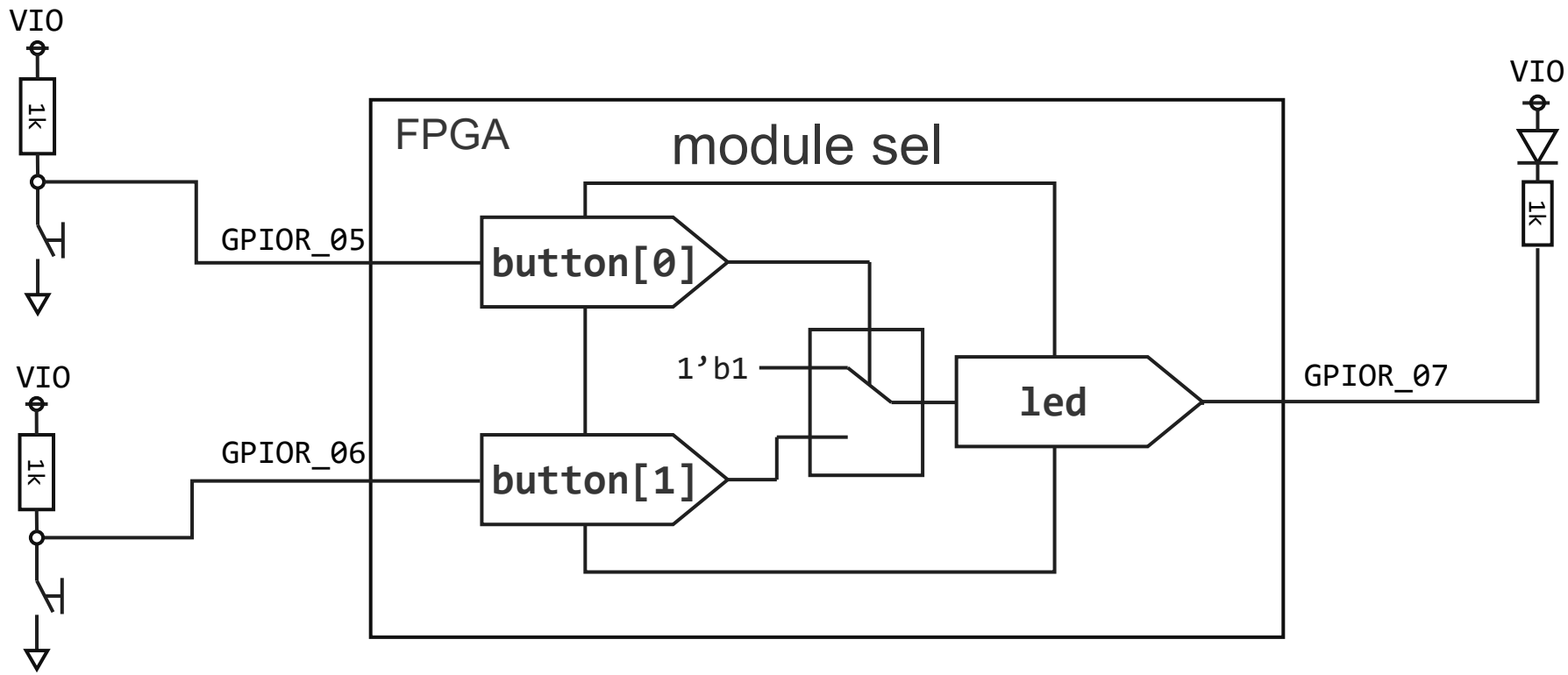
いずれの進数の表記であっても、「’」の手前の数字は2進数でのビット数を意味する。

例）16進数で10進数における10を意味する「0xa」は、

```
4’xa
```

16進数における1桁は4ビットとなるから、「4’」となる。

三項演算子を使った回路



button[0]でbutton[1]を通過させるか選択する回路を作る
button[0]を押している間はbutton[1]でledの点灯・消灯を制御できるようにする

三項演算子のVerilogコード

※Top Module/Entitiyをselに変更するのを忘れずに

Code Editor

sel.v

```
1  module sel
2  input wire [1:0] button,
3  output wire led);
4
5  assign led = (button[0] == 1'b1)? 1'b0: button[1];
6
7  endmodule
8
```

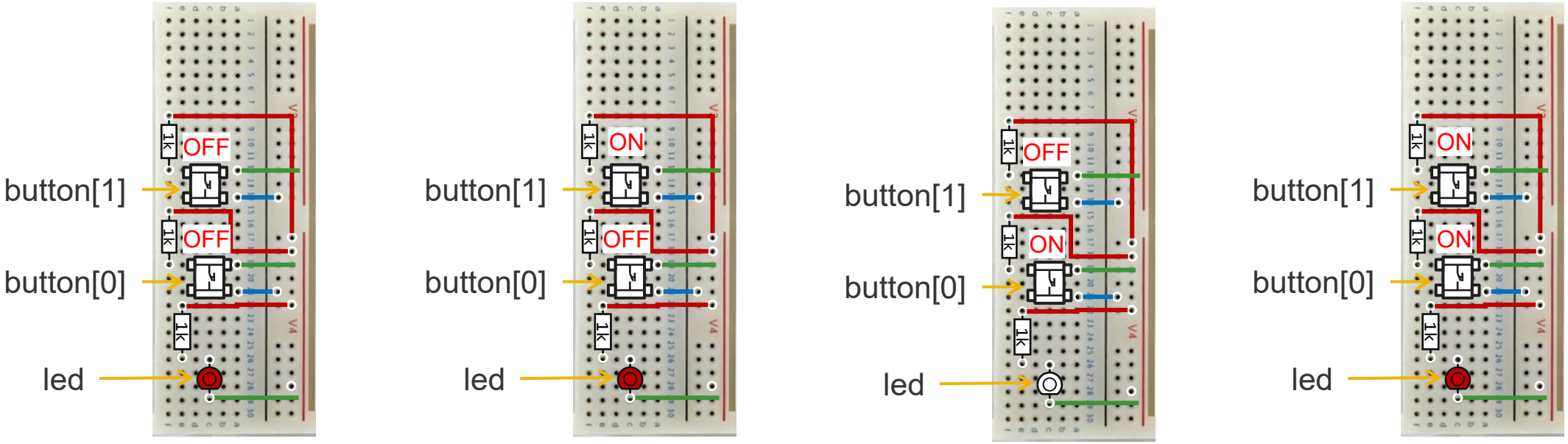
モジュール名はmy_and

カッコ内に条件を記述する。
三項演算子は「?」と「:」で記述

GPIO : Instance View

Instance	Package Pin	Resource	I/O Bank	Alt Conn	Features	Clock Region	Pad
			all	all	all	all	
button[0]	B6	GPIOR_05	2A	None	None	R1	GPIOR_05
button[1]	C6	GPIOR_06	2A	None	None	R1	GPIOR_06
led	C7	GPIOR_07	2A	None	None	R1	GPIOR_07

動作



- 何もボタンを押していない場合、LEDは0 → 点灯する
- button[0]を押した場合button[1]が透過になる。
  - button[1]を押した場合は点灯
  - button[1]を離した場合は消灯

ビット連接

ビット連接

```
{ a, b }
```

のように、{}内にカンマ区切りで配線を入れると新たなバスを定義できる。
左側にあるほど上位ビットとなる。

ビット連接は assign 構文の両辺で使うことができる。例えば、左辺に

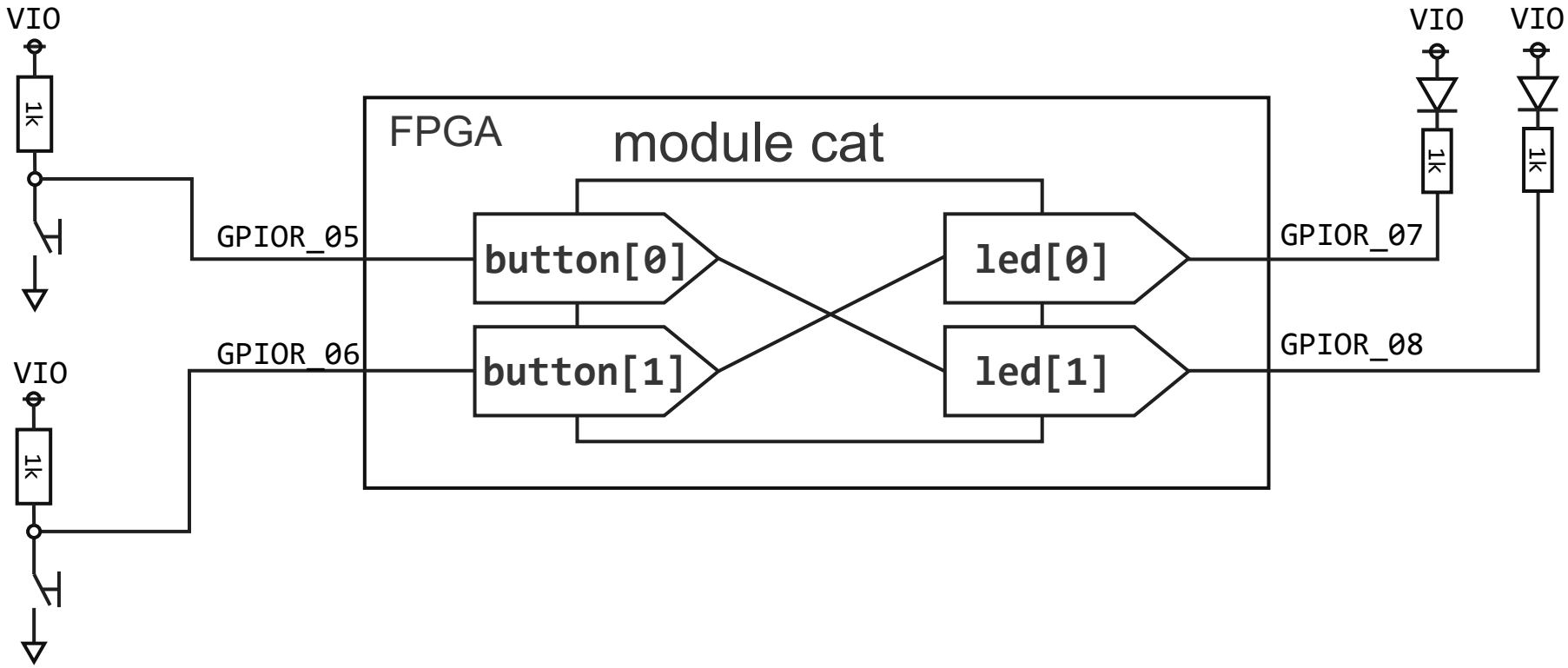
```
assign c = { a, b };
```

のように使うことで、cにaとbを連結したバスを接続できる。このとき、cは同じ長さのバスとして定義されている必要がある。右辺に使うことで、

```
assign { a, b } = c;
```

バスcをaとbに分けて接続できる。このとき、aとbのバス幅の合計はcのバス幅と一致する必要がある。(エラーにはならないが、バグの温床となるので合わせる)

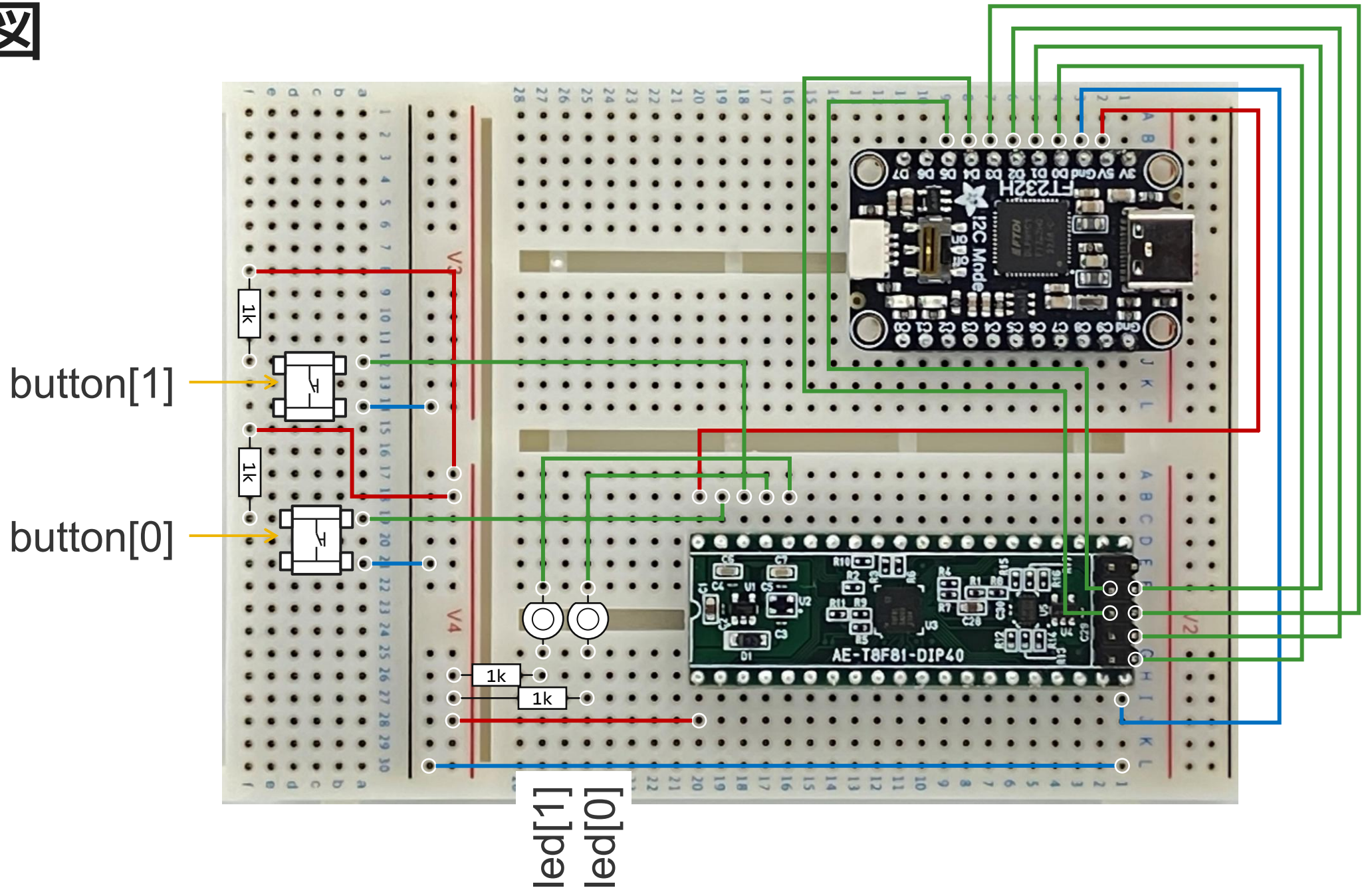
ビット連接を使った回路



スイッチ2個とLED2個の回路に変更する

スイッチとLEDの対応をスワップする回路を作る

配線図



ビット連接を使った回路のVerilogコード

※Top Module/Entitiyをcatに変更するのを忘れずに

Code Editor

cat.v

```
1 module cat(
2 input wire [1:0] button,
3 output wire [1:0] led);
4
5 assign led = {button[0], button[1]};
6
7 endmodule
8
```

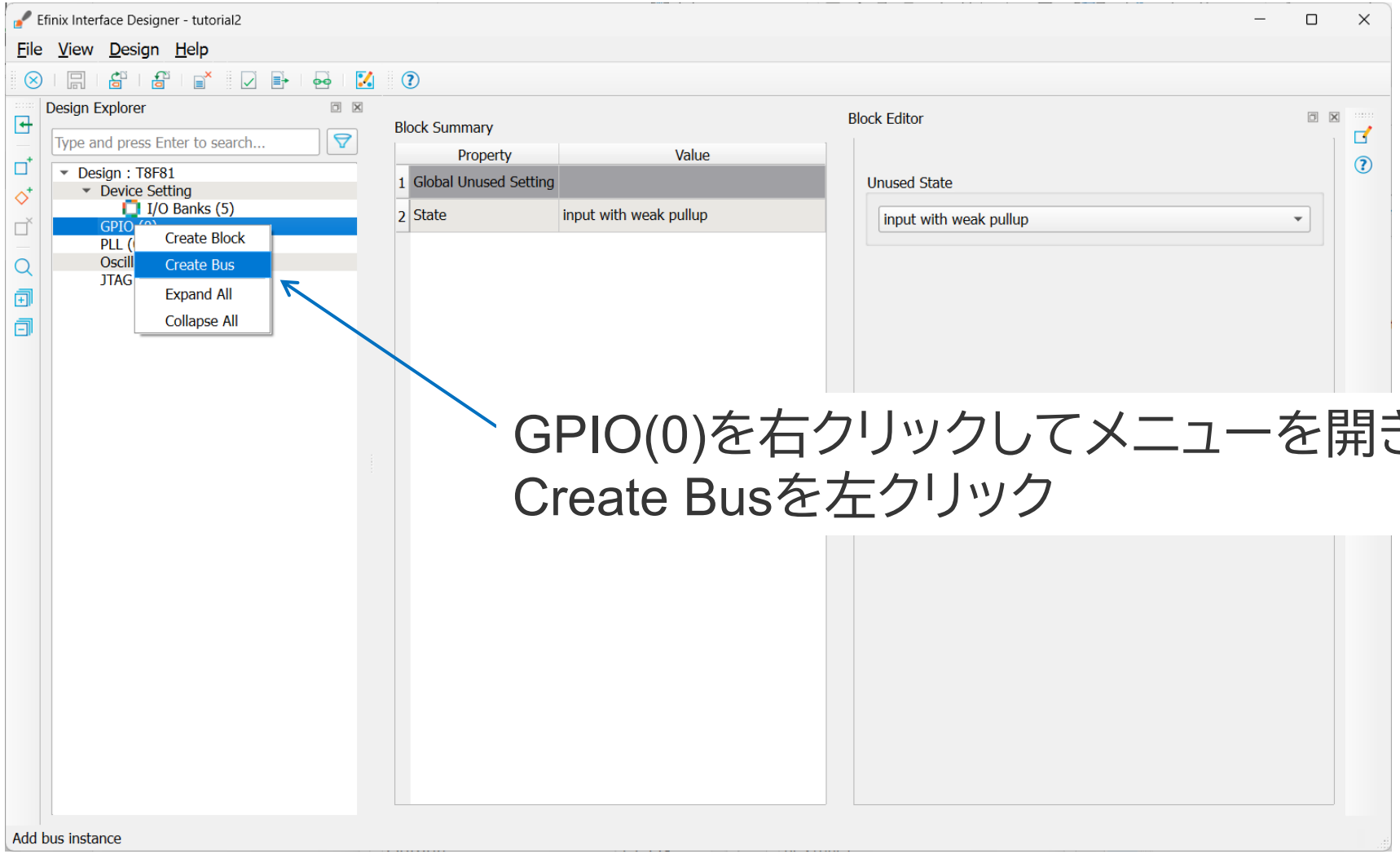
モジュール名はcat

入出力ともにバスにする

左側を[0]とする。左側のほうが上位ビットとなる。

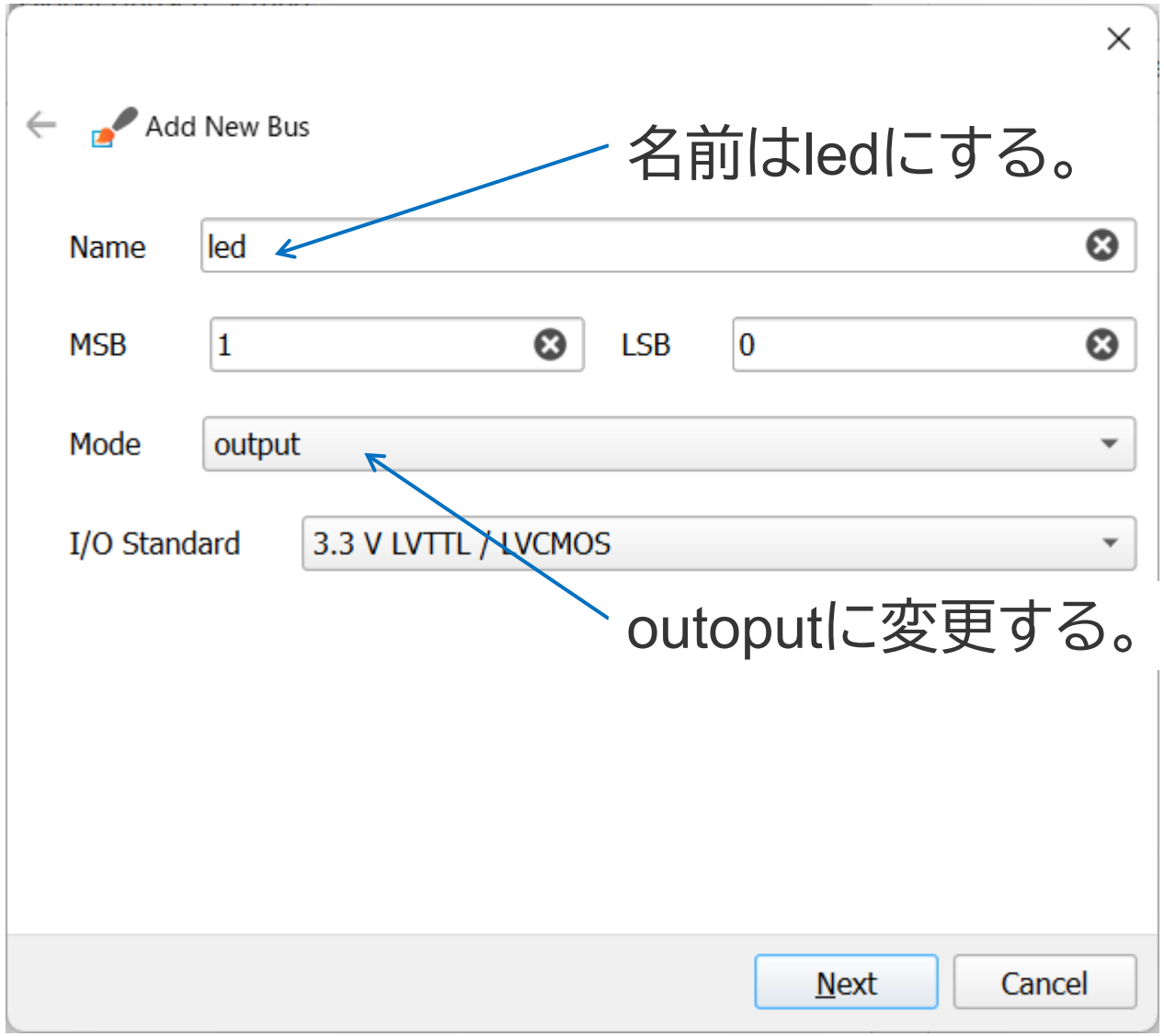
バスのピンアサイン1

Interface Designerを開く

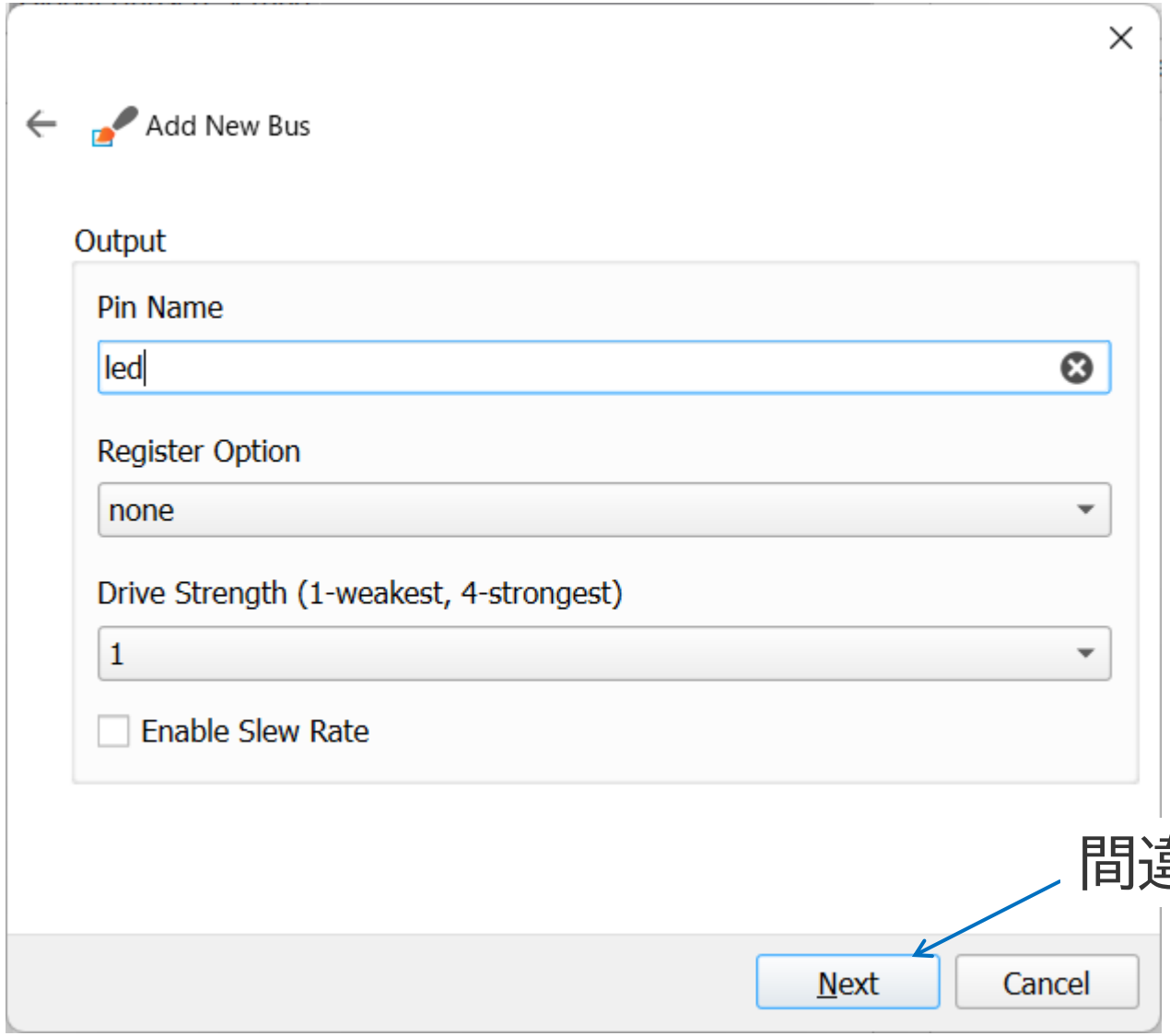


GPIO(0)を右クリックしてメニューを開き
Create Busを左クリック

バスのピンアサイン2


Add New Bus
名前はledにする。
Name
led
MSB
1
LSB
0
Mode
output
I/O Standard
3.3 V LVTTL / LVCMOS
outoputに変更する。
Next
Cancel

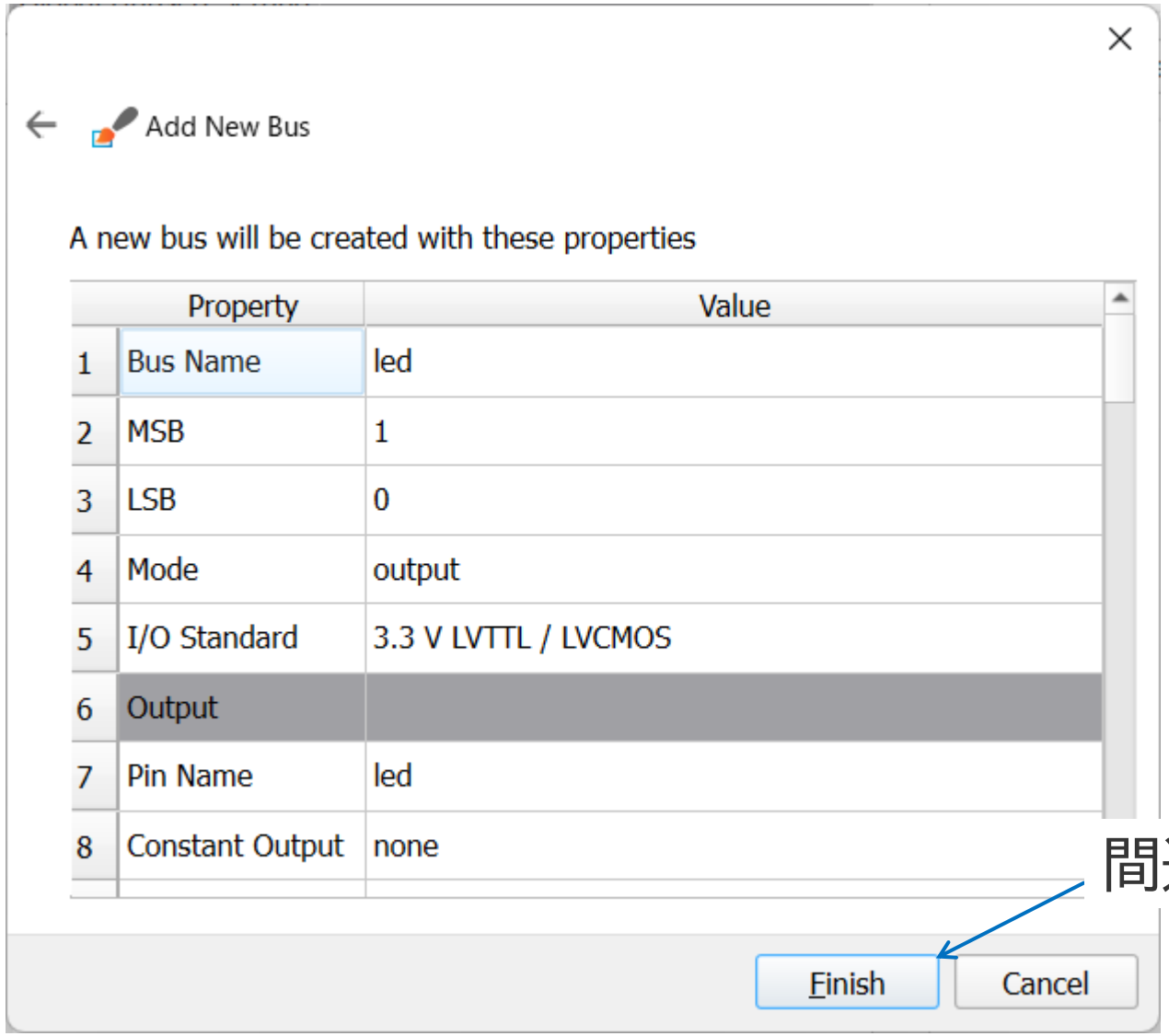
バスのピンアサイン3



ここは特に変更しない。
間違いがなければNext

間違いが無ければNextを左クリック。

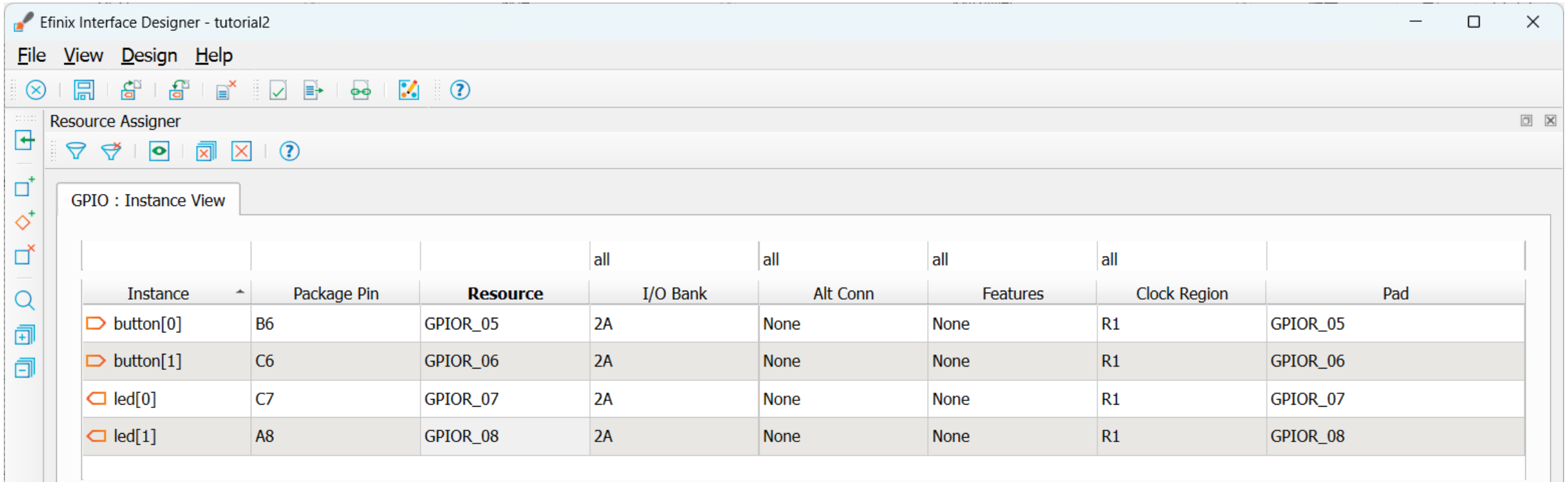
バスのピンアサイン4



最終確認。間違いが無ければFinish。

間違いが無ければNextを左クリック。

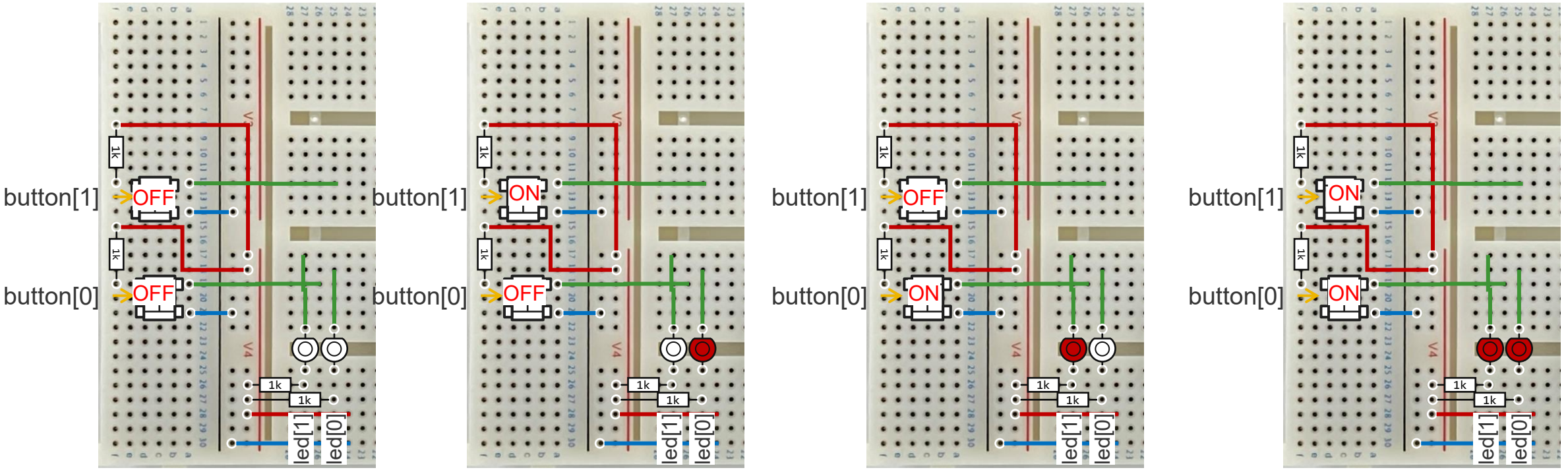
バスのピンアサイン



			all	all	all	all	
Instance	Package Pin	Resource	I/O Bank	Alt Conn	Features	Clock Region	Pad
button[0]	B6	GPIOR_05	2A	None	None	R1	GPIOR_05
button[1]	C6	GPIOR_06	2A	None	None	R1	GPIOR_06
led[0]	C7	GPIOR_07	2A	None	None	R1	GPIOR_07
led[1]	A8	GPIOR_08	2A	None	None	R1	GPIOR_08

- button を GPIO_R05とGPIOR_06に割り当てる
- ledをGPIOR_07とGPIOR_08に割り当てる

動作



button[1]はled[0]の点灯・消灯をコントロールし
button[0]はled[1]の点灯・消灯をコントロールする

always文

reg

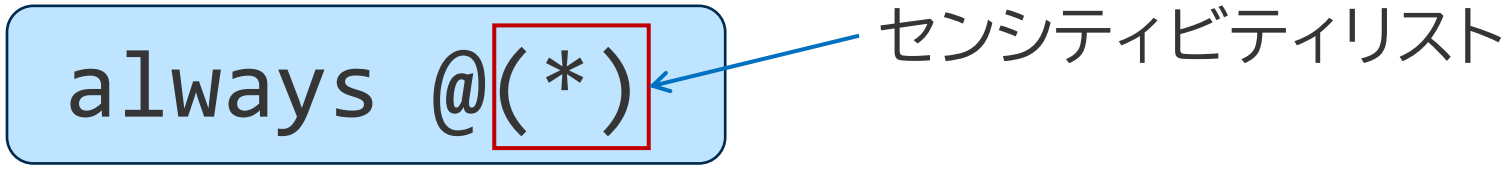
reg構文を用いることで**記憶素子を定義**できる。ただし、Verilogにおけるreg構文は合成結果として記憶素子にならないこともあり、**記憶素子にならないようなreg**の使い方を学習する。reg構文は、

```
wire reg_name;
```

のように記述する。regの後ろには、配線名をつける。これで、reg_nameというレジスタ(配線)ラベルを新たに定義できる。

always文

always文は**信号に変化があったときに実行する動作**を書くことができる構文。次のように記述する



「always」のあとの「@」から後ろをセンシティビティリストと言う。
センシティビティリストには変化を捕捉する信号を列挙する。
センシティビティリストに「*」を記述することで、モジュール内のすべての信号が列挙される。すなわち、モジュール内すべての信号の変化を補足することを意味する。

always文内ではwireを接続することはできない。
すなわち、「assign」を記述することができない。
always文内では代入される対象はすべて「reg」で定義される必要がある。

begin～end文

```
always @(*)
begin
    この範囲に
    動作を書く
end
```

beginからendまでの囲まれた範囲を1つのブロックとして扱うことができる。

複数の文を同時に動作させる範囲として指定するために使用する。

他のプログラミング言語では「{}」が使用されるが、
Verilogでは「{}」はビット連接ですでに使われているため、
このような表記となる。

なお、Verilogではインデントはすべて無視されるため、
あくまでも見やすさのために入れていることに注意。

alwaysのセンシティビティリストに続けて「begin」を書いても問題ない。
beginを改行して表記するのは個人的趣味。

if

```
always @(*)
begin
   if (s == 2’b00) e = a;
   else if (s == 2’b01) e = b;
   else if (s == 2’b10) e = c;
   else e = d;
end
```

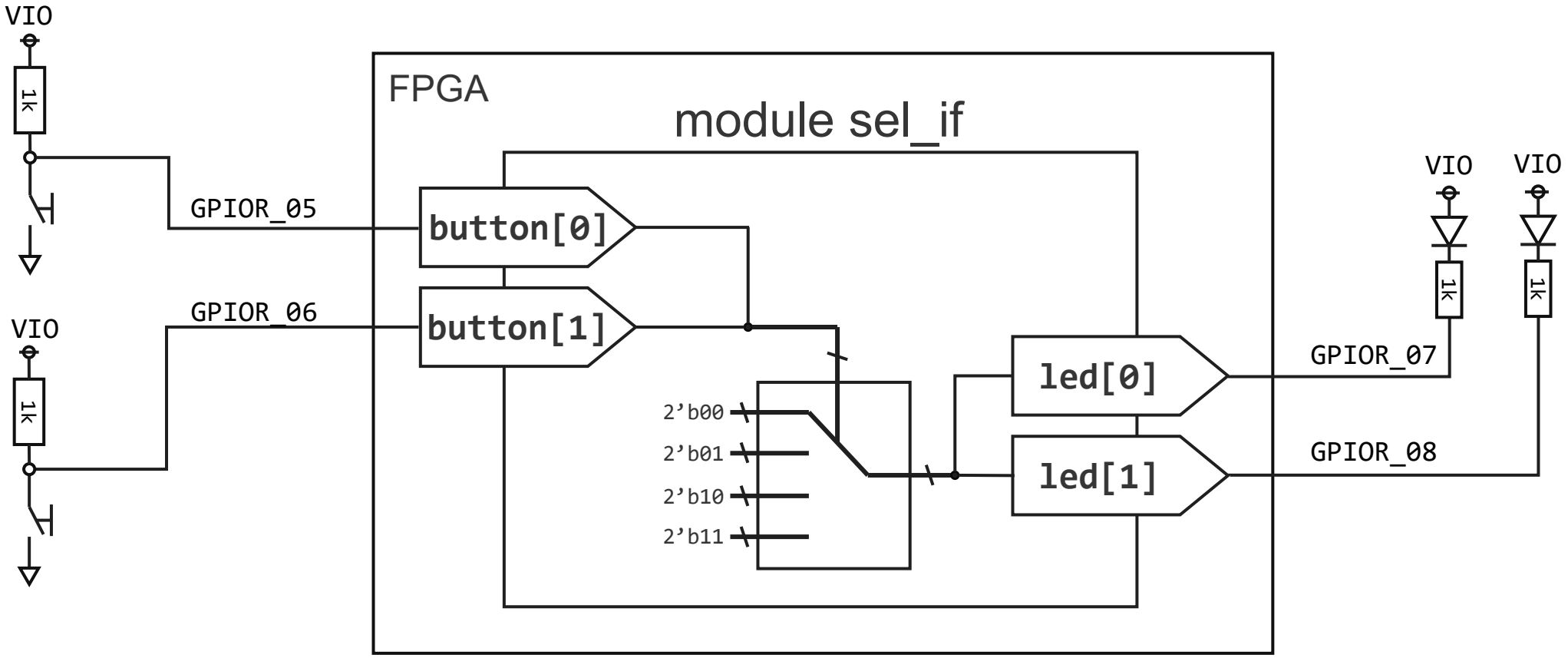
if文はalways文の中でのみ使うことができる。

条件がTrueの場合にその行の文が実行される。

条件には三項演算子と同様に等値、非等値、大小が使用できる。

組み合わせ回路を合成する場合は必ず「else」文が必要となる
「else」文がない場合、記憶回路（ラッチ）が合成される事があるので注意!
とりあえず、「else」が絶対に必要なことだけを絶対に覚える。

alwaysとifを用いたledの点灯回路



2つのボタンで2つのLEDのON/OFFをコントロールするセレクタ回路を作る

alwaysとifを用いたled点灯のVerilogコード

※Top Module/Entitiyをsel_ifに変更するのを忘れずに

Code Editor

sel_if.v

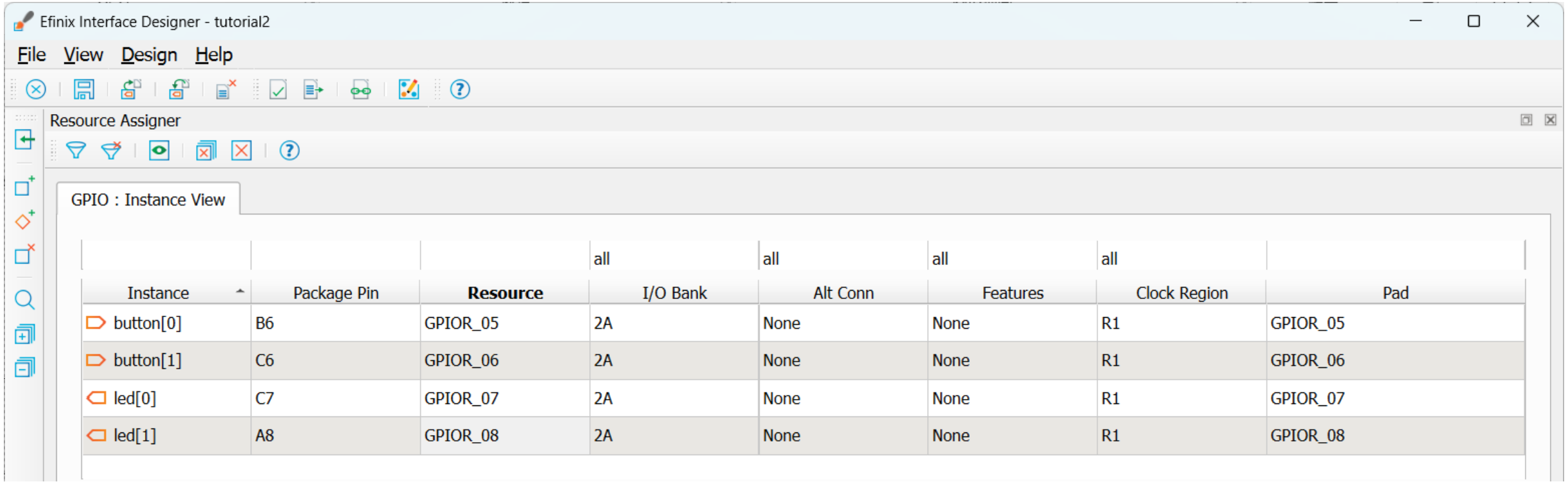
```
1  module sel_if(
2  input wire [1:0] button,
3  output reg [1:0] led);
4
5  always @(*)
6  begin
7      if (button == 2'b11) led = 2'b11;
8      else if (button == 2'b10) led = 2'b10;
9      else if (button == 2'b01) led = 2'b01;
10     else led = 2'b00;
11 end
12
13 endmodule
```

モジュール名はsel_if

ledはregで定義する

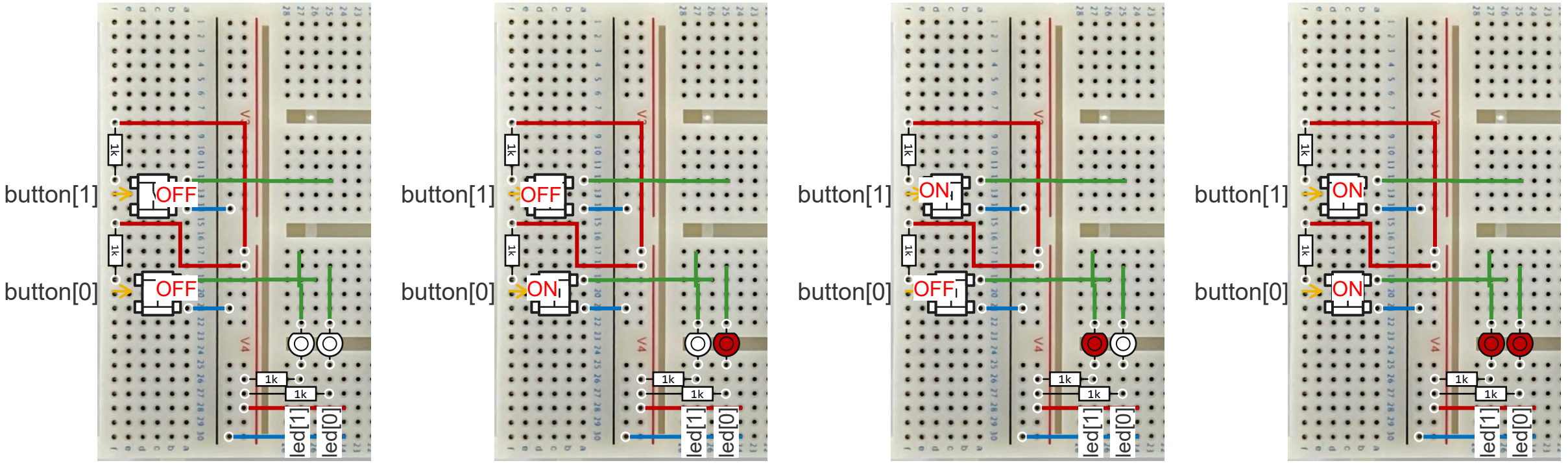
組み合わせ回路を合成するには
全パターンを列挙
またはelseを用いて列挙されていない
条件が生じないようにする

ピンアサイン



			all	all	all	all	
Instance	Package Pin	Resource	I/O Bank	Alt Conn	Features	Clock Region	Pad
button[0]	B6	GPIOR_05	2A	None	None	R1	GPIOR_05
button[1]	C6	GPIOR_06	2A	None	None	R1	GPIOR_06
led[0]	C7	GPIOR_07	2A	None	None	R1	GPIOR_07
led[1]	A8	GPIOR_08	2A	None	None	R1	GPIOR_08

動作



button[1:0] の論理に対応したLEDが点灯する

case～endcase文

```
case(s)
   2’b00: b = 1’b0;
   2’b01: b = 1’b1;
   2’b10: b = 1’b1;
   2’b11: b = 1’b0;
   default: b = 1’b0;
endcase
```

case(x)のxの値によって動作を切り替える事ができる。
case文では一致のみ比較することができる。(大小の比較はif文でしかできない。)
case文はalways文内でのみ記述することができる。
組み合わせ回路を合成するには、すべての値が列挙されなければならない。
defaultケースはケース不足を補うことができる。

alwaysとcase文を用いたled点灯のVerilogコード

if文をcase文に置き換える

※Top Module/Entitiyをsel_ifに変更するのを忘れずに

Code Editor

sel_if.v | sel_switch.v

```
1  module sel_switch(
2  input wire [1:0] button,
3  output reg [1:0] led);
4
5  always @(*)
6  begin
7      case(button)
8          2'b00: led = 2'b00;
9          2'b10: led = 2'b10;
10         2'b01: led = 2'b01;
11         2'b11: led = 2'b11;
12         default: led = 2'b11;
13     endcase
14 end
15
16 endmodule
```

モジュール名はsel_if

ledはregで定義する

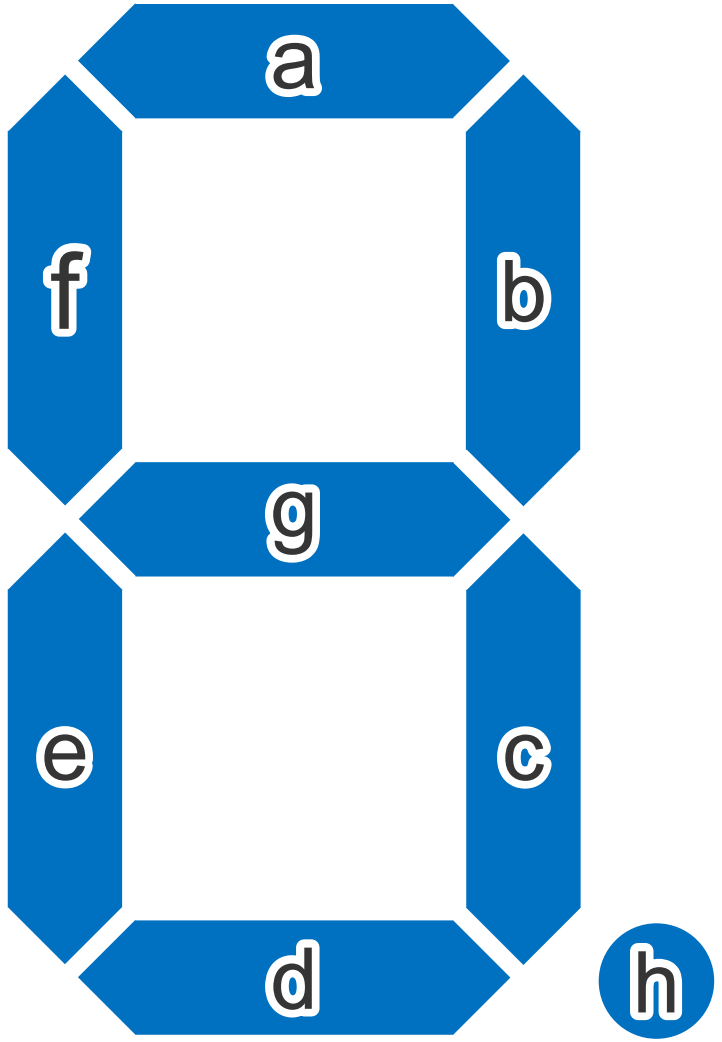
組み合わせ回路を合成するには全パターンを列挙
defaultケースでケース不足を補うようにする

ピンアサインはif文のときと同じにする。

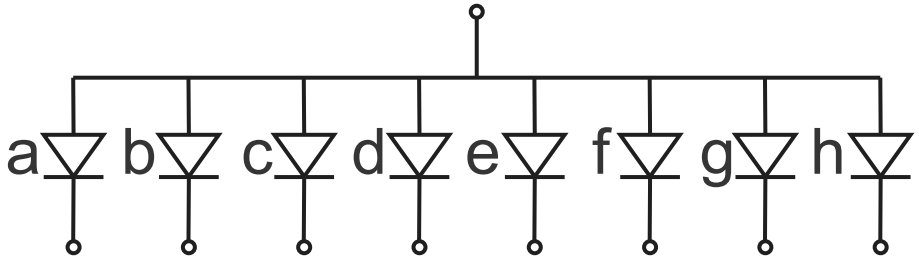
7セグメントLEDを用いた回路

7セグメント表示器

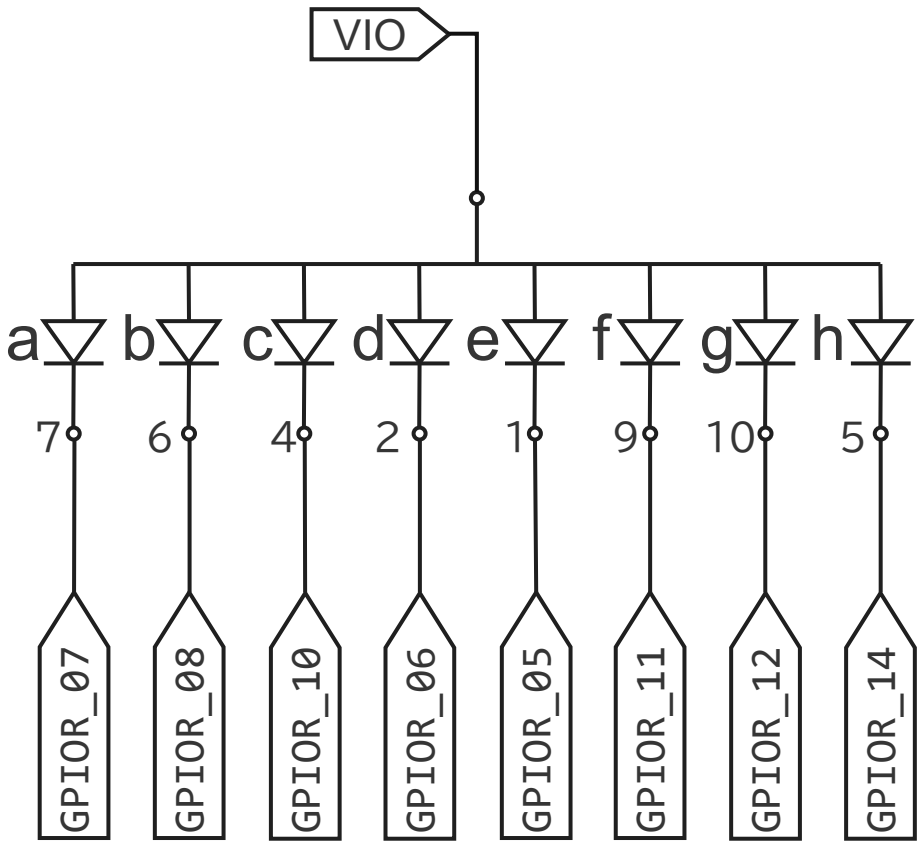
- 7個のLEDを数字を表示するように配列
- ドットを表すLEDが1つの合計8個のLED
- LEDはそれぞれ個別に点灯・消灯を制御可
- **カソード**を共通にした**カソードコモン**、
  アノードを共通にした**アノードコモン**がある。
  - ✓ 今回は**アノードコモン**を使用

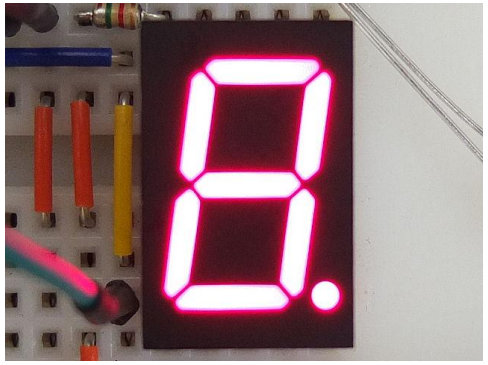


LEDの回路図はこんな感じ



7セグメントLEDとFPGAの接続

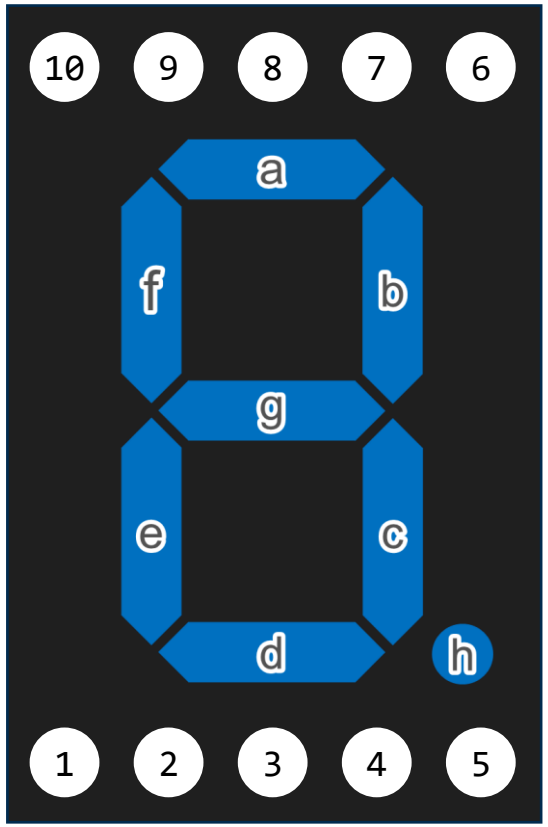




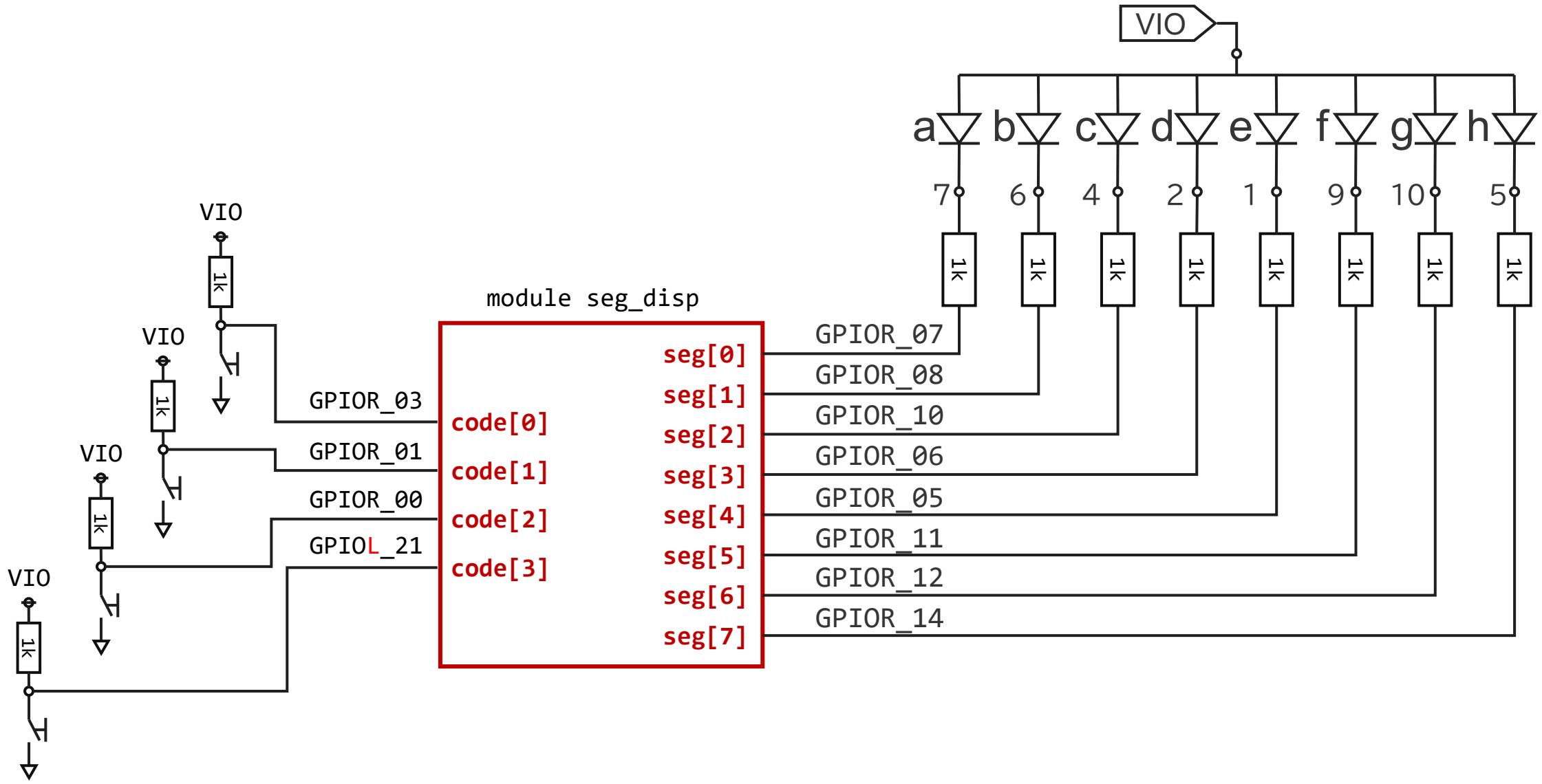
PARA LIGHT ELCTRONICS CO., LTD.

A-551SRD-NW-A B/W

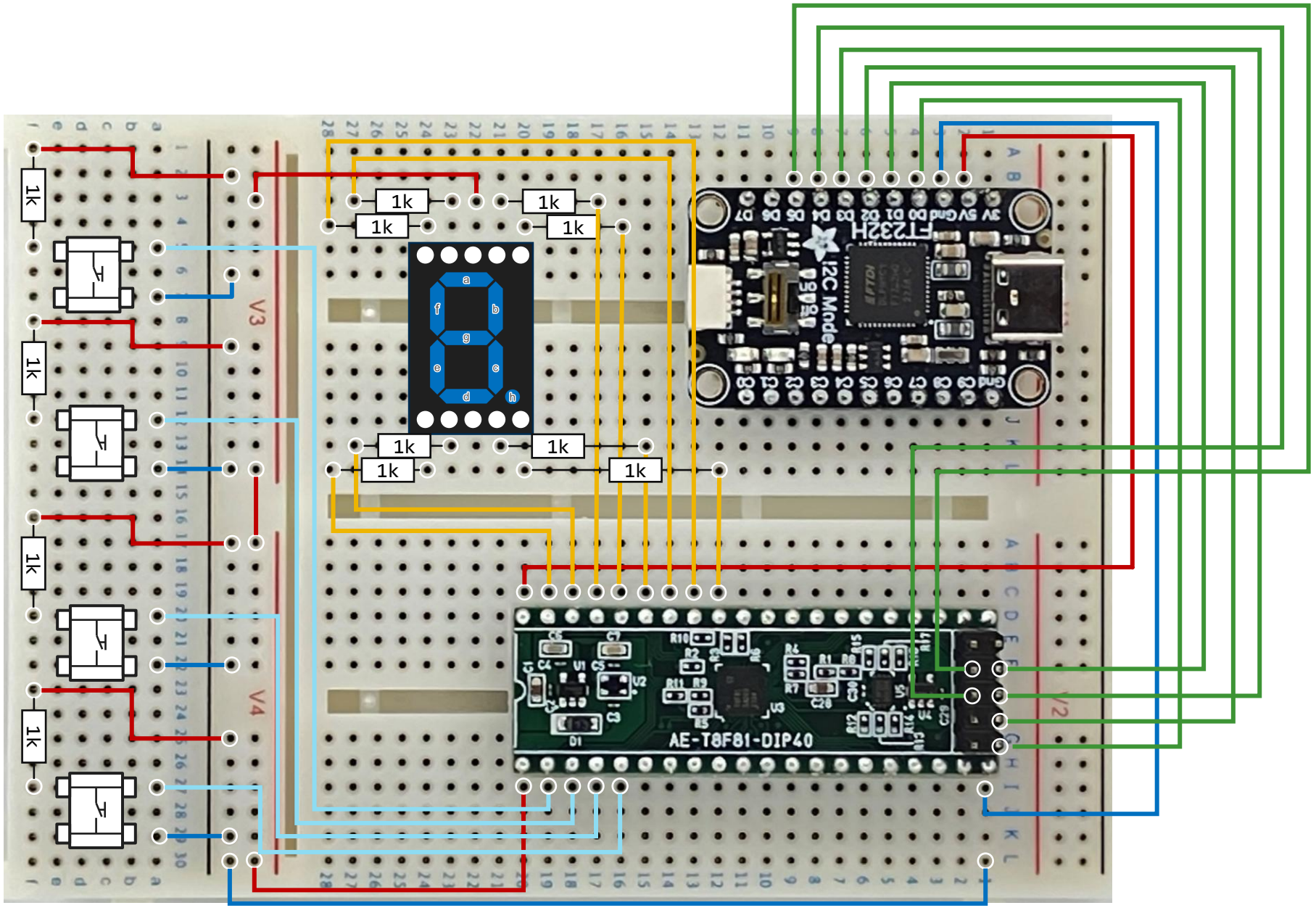
https://akizukidenshi.com/catalog/g/g100639/



7セグメントディスプレイとカウンター回路の接続

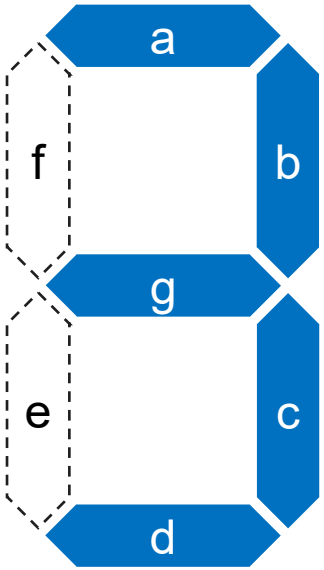


配線図



数字の表示方法

3を表示することを考える。消灯するセグメントは1、点灯するセグメントは0が出力されるようなコードを書く。



h	g	f	e	d	c	b	a
seg[7]	seg[6]	seg[5]	seg[4]	seg[3]	seg[2]	seg[1]	seg[0]
1	0	1	1	0	0	0	0

→ 8’b10110000

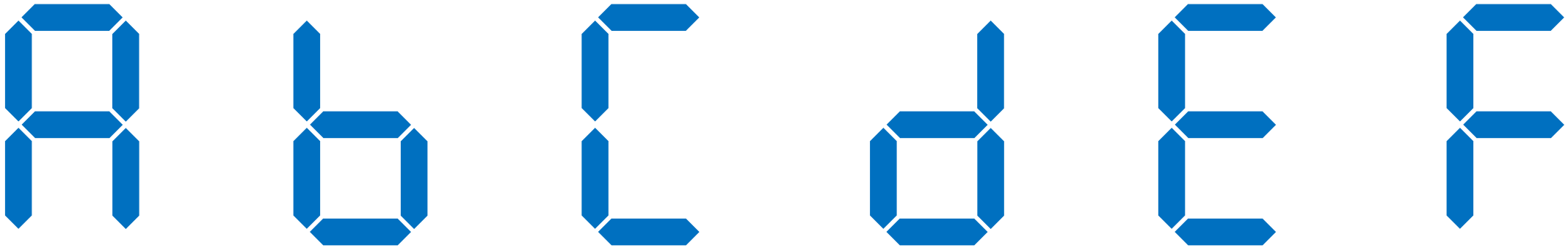
3を表示するときは seg = 8’b10110000 となるようにコードを書く。
他の数字についても同様に調べる。

7セグメントLEDに数字を表示するために書き込む値

表示する数字	seg[7]	seg[6]	seg[5]	seg[4]	seg[3]	seg[2]	seg[1]	seg[0]	書き込む数値（2進数：8’b）
0									
1									
2									
3	1	0	1	1	0	0	0	0	8’b10110000
4									
5									
6									
7									
8									
9									
a									
b									
c									
d									
e									
f									

7セグメントLED表示器a~fのフォントについては次ページに記載

A B C D E F



7セグメントでアルファベットを表示するのには無理があるので、
大文字・小文字を織り交ぜて表示することになる。
人間の目は偉いので違和感ありながらなんとなくわかる。

seg_coderのVerilogコード

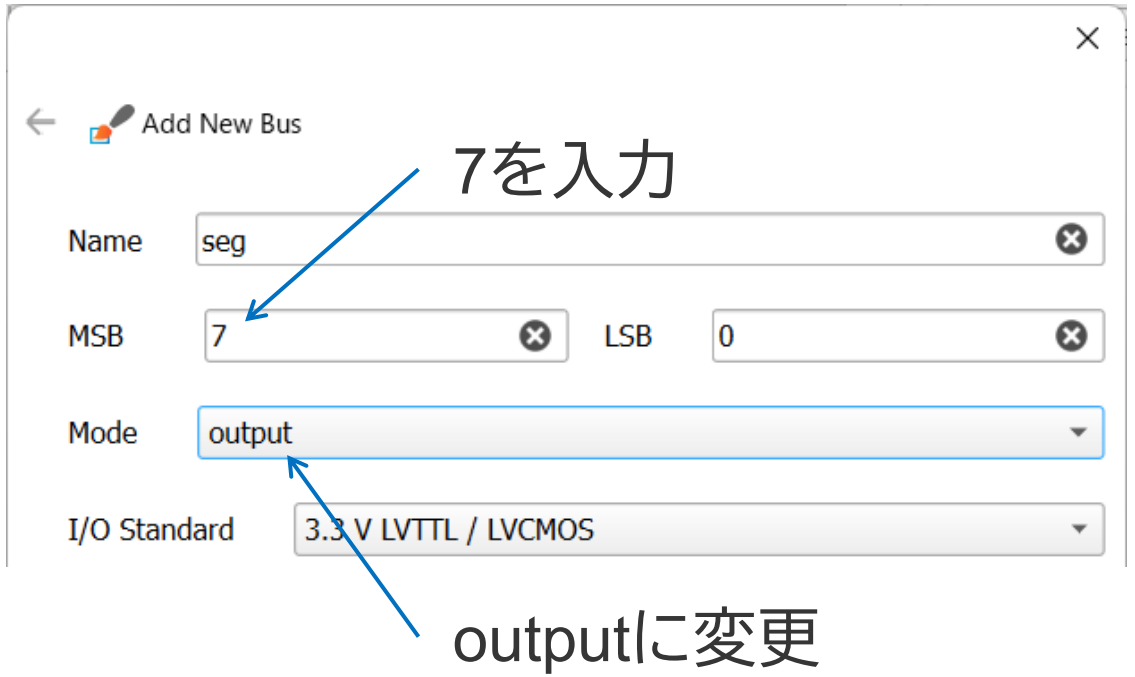
seg_coderは組み合わせ回路で作成できる。ここではalwaysを用いた組み合わせ回路の作成方法を用いる。

```
module seg_disp(
Input wire [3:0] code,
Output reg [7:0] seg);
always @(*)
begin
        case (code)
        4'b0000: seg = 8'b         ; // 0
        4'b0001: seg = 8'b         ; // 1
        4'b0010: seg = 8'b         ; // 2
        4'b0100: seg = 8'b10110000; // 3
        4'b1000: seg = 8'b         ; // 4

        default: seg = 8'b11111111; // 全部消灯
        endcase
end
endmodule
```

前のページで作成した表を用いて入力する。

バスブロックの作成


Add New Bus
3を入力
Name
code
MSB
3
LSB
0
Mode
input
I/O Standard
3.3 V LVTTL / LVCMOS


Add New Bus
7を入力
Name
seg
MSB
7
LSB
0
Mode
output
I/O Standard
3.3 V LVTTL / LVCMOS
outputに変更

ピンアサイン

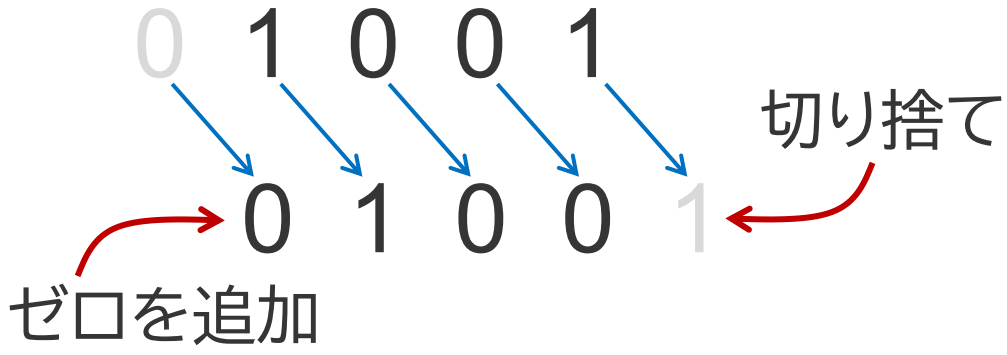
GPIO : Instance View

			all	all	all	all	
Instance	Package Pin	Resource	I/O Bank	Alt Conn	Features	Clock Region	Pad
code[0]	A6	GPIOR_03	2A	None	None	R1	GPIOR_03
code[1]	B5	GPIOR_01	2A	None	None	R1	GPIOR_01
code[2]	A5	GPIOR_00	2A	None	None	R1	GPIOR_00
code[3]	B3	GPIOL_21	1B	None	None	L1	GPIOL_21_NSTATUS
seg[0]	C7	GPIOR_07	2A	None	None	R1	GPIOR_07
seg[1]	A8	GPIOR_08	2A	None	None	R1	GPIOR_08
seg[2]	A9	GPIOR_10	2A	None	None	R1	GPIOR_10
seg[3]	C6	GPIOR_06	2A	None	None	R1	GPIOR_06
seg[4]	B6	GPIOR_05	2A	None	None	R1	GPIOR_05
seg[5]	B8	GPIOR_11	2A	None	None	R1	GPIOR_11
seg[6]	C8	GPIOR_12	2A	None	None	R1	GPIOR_12
seg[7]	B9	GPIOR_14	2A	None	None	R1	GPIOR_14

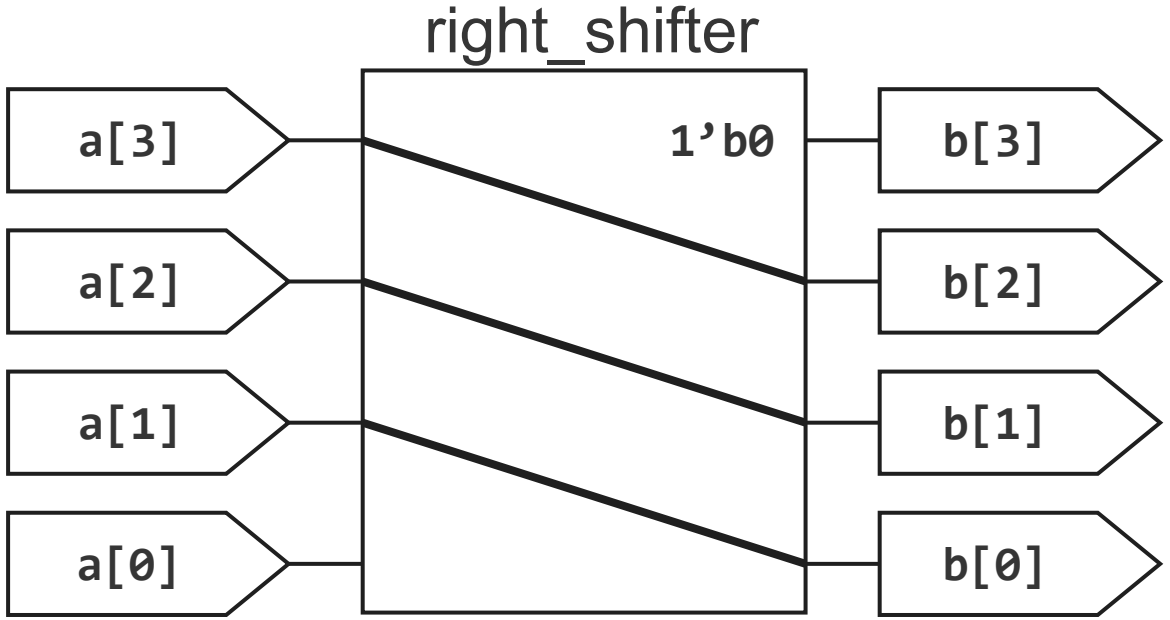
今日の発展課題

右シフター（バスを用いた回路）

各ビットを下記のように順序を維持したまま右側にシフトする回路を右シフターと呼ぶ。
このとき、最上位ビットはゼロを付け加え、右にはみ出す最下位ビットは切り捨てられる。



右シフターの回路は右図のように実現することができる。
入力の最下位ビットは切り捨て、
出力の最上位ビットにゼロを追加する。



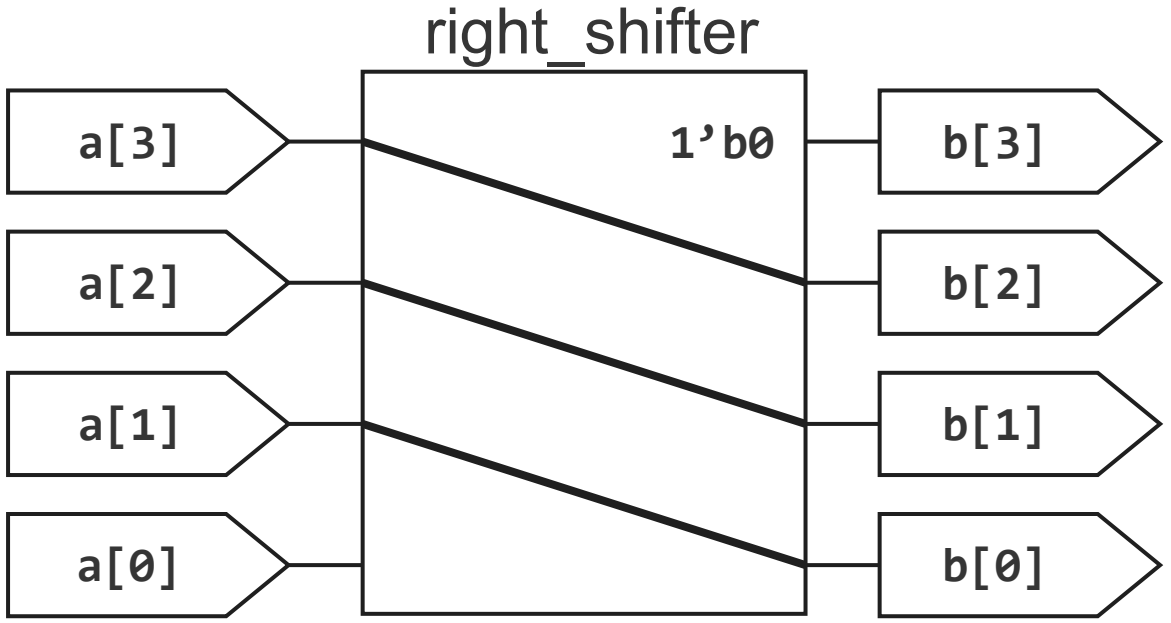
右シフターの記述

右の回路をVerilogで記述する

```
module right_shifter(
  input
  output                    );

  assign b =                ;
endmodule
```

ここにコードを書く

ここにコードを書く



Half Adder 1

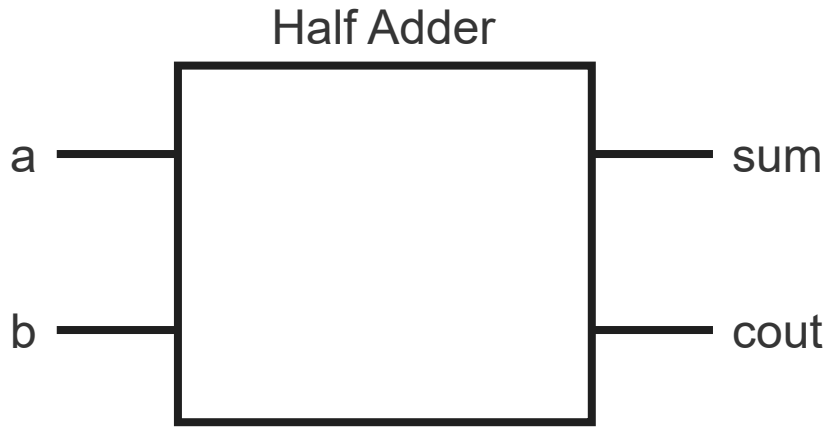
1ビットの2つの信号の和を求める回路をHalf Adder（半加算器）という。

1ビット信号の足し算

0	1	0	1
+) 0	+) 0	+) 1	+) 1
00	01	01	10

繰り上がりの桁が必要になるので1ビット同士の足し算の出力は2ビットになる。
繰り上がりの桁はキャリー（Carry）と呼ばれる

Half Adder 2



真理値表

a	b	sum	cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

↑ 排他的論理和　↑ 論理積

- Half Adderは2つの入力、2つの出力がある。
- 真理値表の関係を2つの出力それぞれにVerilogコードを書く。

2. Half Adder のVerilogコード

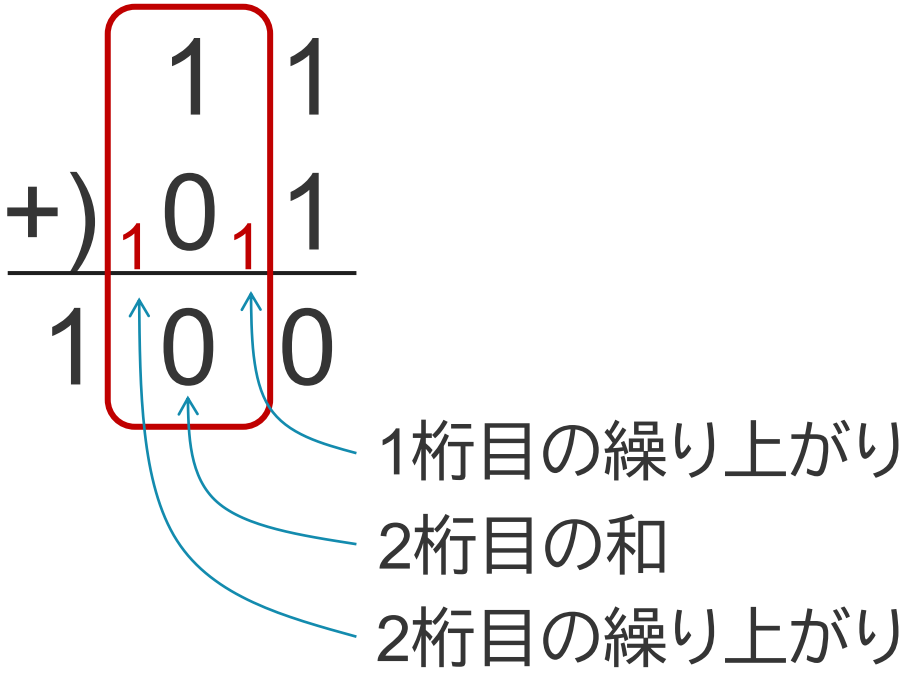
2つの信号aとbの論理和と論理積を求め同時に出力する回路を作成する。

```
module half_adder(
  input wire a,
  input wire b,
  output wire sum,
  output wire cout);

  ここにコードを書く

endmodule
```

全加算器

全加算器は加算器の最小要素を抜き出した回路である。

加算器の最小要素を考えるために2桁(2ビット)の足し算を考える。



2桁目で加算される数値は、1桁目の繰り上がり、2桁目同士の値の3つである。

したがって、入力は3つ必要になる。

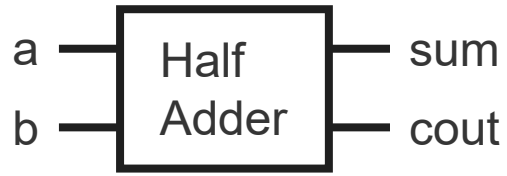
和は2桁となり、2桁目の値と繰り上がりの値になる。

したがって、出力は2つとなる。

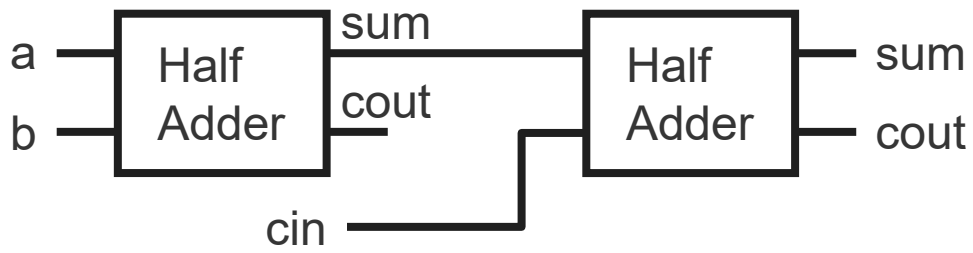
この回路を全加算器と呼び、加算回路を一般化した最小構成となる

Full Adderの実現方法

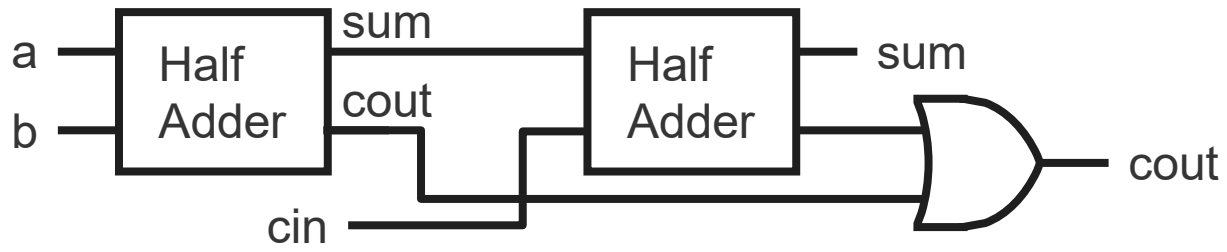
まずaとbの和を考える。これはHalf Adderにより実現される。



次に繰り上がりのcinを加算する。これはHalf Adderを使う。



最後にcoutを考える。Full Adderのcoutが1になるのは、それぞれの半加算器のcoutのどちらか(どちらも)が1になるときである。したがって、2つのcoutの論理和を取れば良い。



全加算器の真理値表

a	b	cin	sum	cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Full Adderの実装

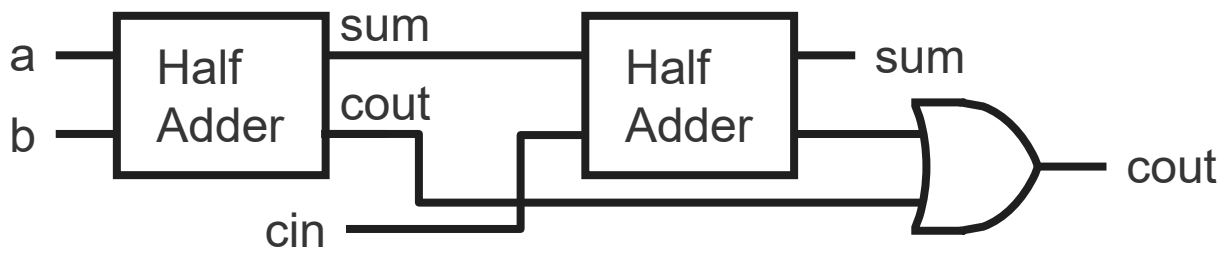
```
module full_adder(
  input wire a,
  input wire b,
  input wire cin,
  output wire sum,
  output wire cout);

  wire first_sum;
  wire first_cout;
  wire second_cout;

  half_adder first(
    .a(a),
    .b(b),
    .sum(first_sum),
    .cout(first_cout)
  );

  half_adder second(
    .a(first_sum),
    .b(cin),
    .sum(sum),
    .cout(second_cout)
  );

  assign cout = first_cout | second_cout;
endmodule
```

1つ目のHalf_adder

2つ目のHalf_adder

coutについて

Half adderを用いてFull Adderを実装する。



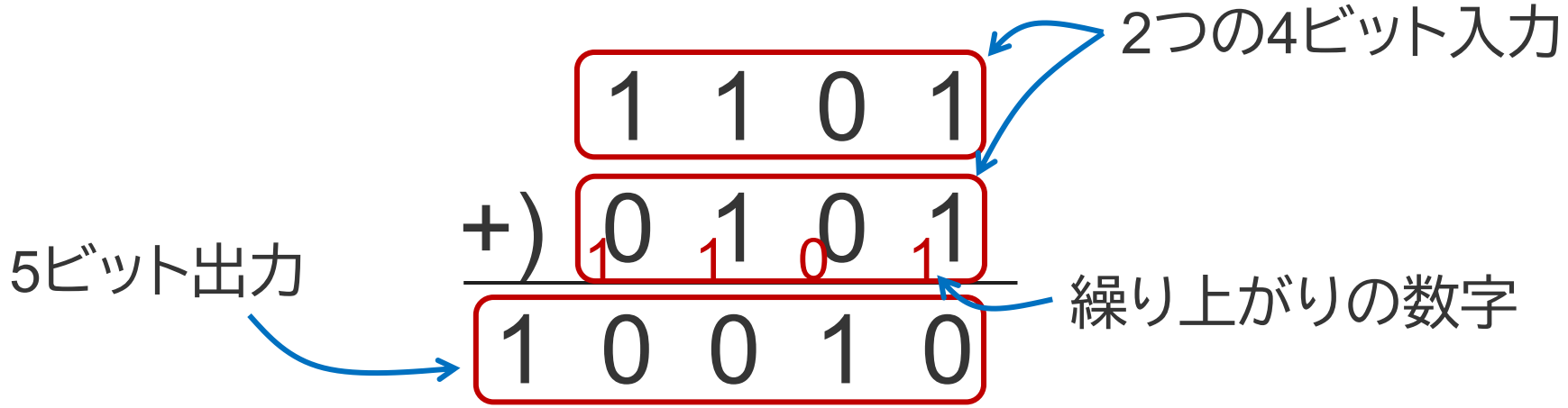
Half adderは1つ前の練習で作成したモジュールを使用する。ファイル名はhalf_adder.svとする。

Compile Optionsには「half_adder.sv」を追加する。

4ビット加算回路の実装

Full Adderを用いて加算回路を実装する。

2進数で4桁の足し算は筆算から次のように考える。

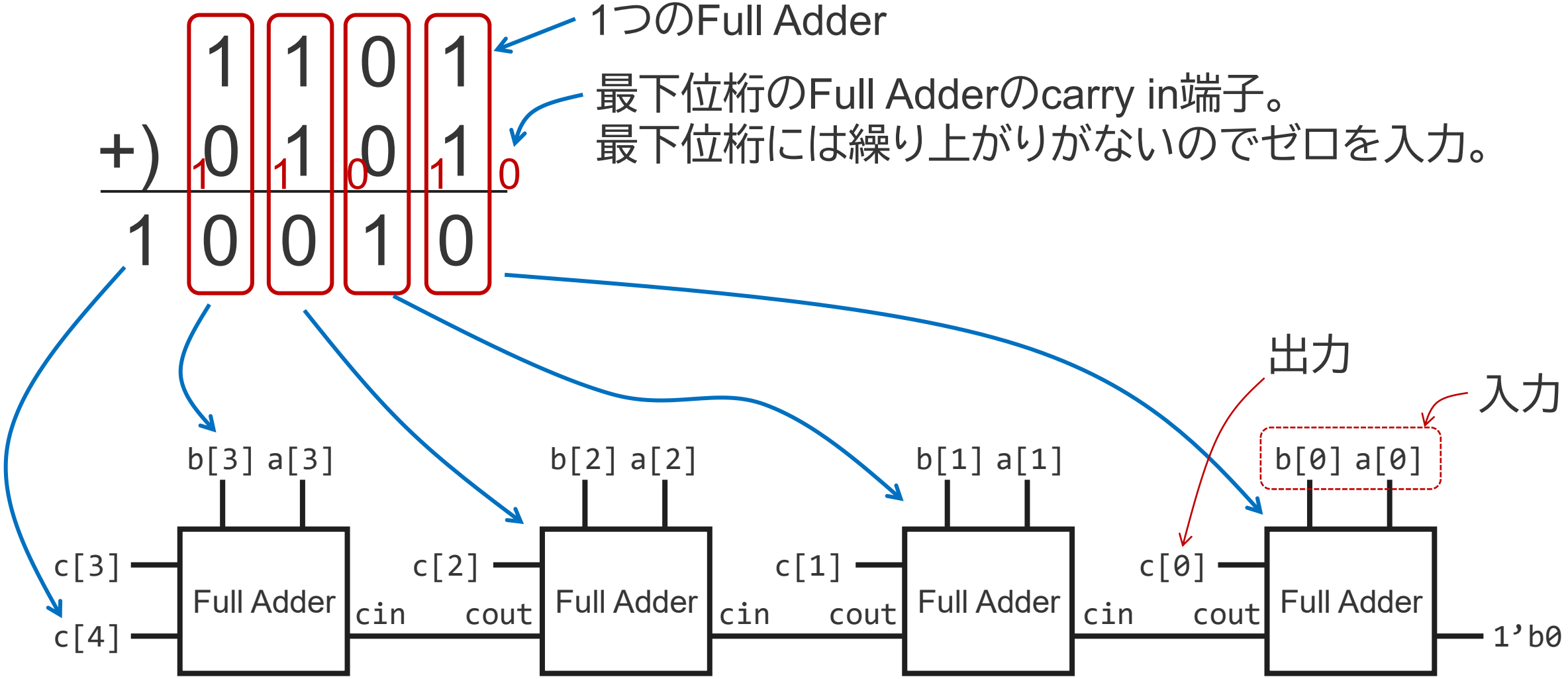


Nビット同士の足し算は繰り上がりからN+1ビットが必要になる。

このことから、4ビット加算回路では入力は4ビットの信号が2つ、出力は4ビットの信号が1つとなる。

Full Adderを用いた4ビット加算回路

多ビットの加算回路を実現するには各桁を加算回路によって実装する。



Full Adderを用いた4ビット加算回路の記述

```
module four_bit_adder(
    input wire [3:0] a,
    input wire [3:0] b,
    output wire [4:0] c);
```

続く

続き


```
endmodule
```

full_adder 及び half_adderモジュールは実習内で作成したものを使用する。それぞれ、full_adder.sv、half_adder.svとして追加する。
Compile Optionsには「half_adder.sv full_adder.sv」を追加する。

2ビット乗算回路1

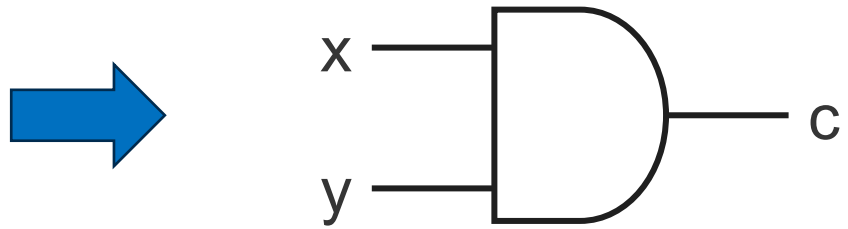
2進数であっても乗算の手順は10進数と同様である。筆算は下記のように計算する。

```
    1 1
×)  1 1
-------
    1 1
  1 1
1 1 1
-------
1 0 0 1
```

これを回路として実現する方法を考える。回路も筆算の計算順序と同様に実現していく。はじめに1ビット同士の乗算を考える。

1ビット同士の乗算

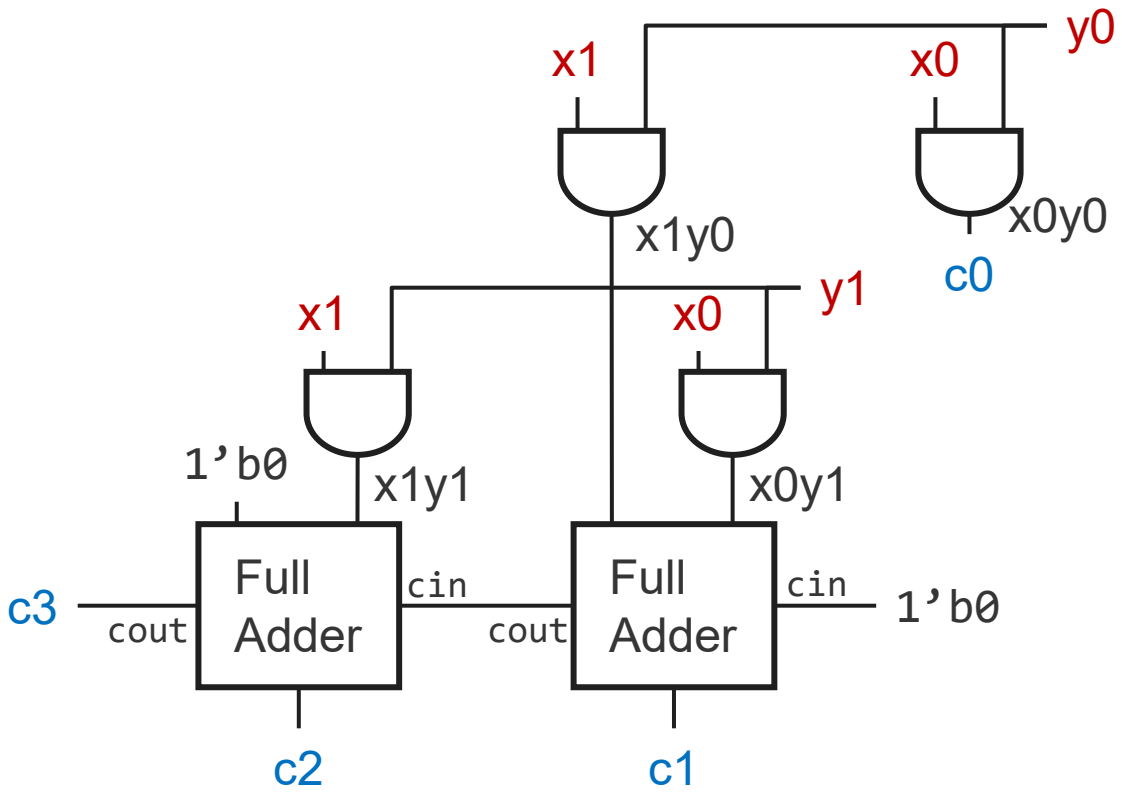
x	y	c
0	0	0
0	1	0
1	0	0
1	1	1



1ビット同士の乗算の真理値表は論理積と同じであり、1ビット同士の乗算は論理積として表現できる。したがって、筆算の各値は論理積として考えることができる。

2ビット乗算回路2

各桁の論理積を並べ、加算する桁にFull Adderをつなぐと下記のようになる。
1つ目のFull Adderはキャリー入力にゼロをつなぎ、2つ目のFull Adderは何も繋がない入力にはゼロを接続する。



入力は赤、出力は青で表記

$$
\begin{array}{rcccc}
 & & x1 & x0 & \\
\times) & & y1 & y0 & \\
\hline
 & x1y0 & x0y0 & & \\
 & x1y1 & x0y1 & & \\
\hline
c3 & c2 & c1 & c0 &
\end{array}
$$

2ビット乗算回路のVerilog記述

```
module two_bit_multiply(
    input wire [1:0] a,
    input wire [1:0] b,
    output wire [3:0] c);
```

ここにコードを書く

続く

続き

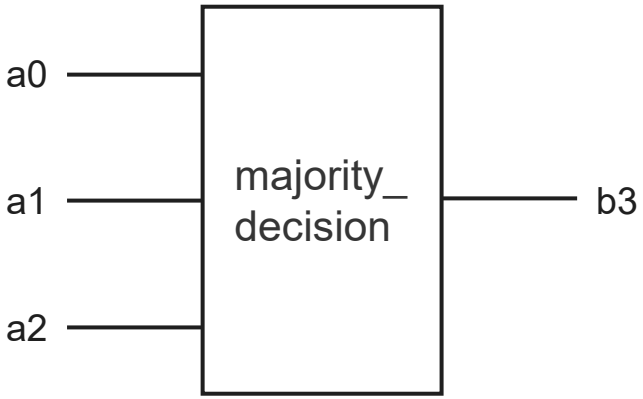
ここにコードを書く

```
endmodule
```

full_adder 及び half_adderモジュールは実習内で作成したものを使用する。それぞれ、full_adder.sv、half_adder.svとして追加する。Compile Optionsには「half_adder.sv full_adder.sv」を追加する。

多数決回路

多数決回路は入力3ビットのうち、2ビット以上が1のときに1が出力される回路である。



a0	a1	a2	b0
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

多数決回路のVerilog記述

```
module majority_decision(
  input wire [2:0] a,
  output reg b);

```


```
endmodule
```