

# 電卓の設計

---

2007年12月VDECリフレッシュ教育

京都大学情報学研究科

小林和淑

# どうして電卓なの？

---

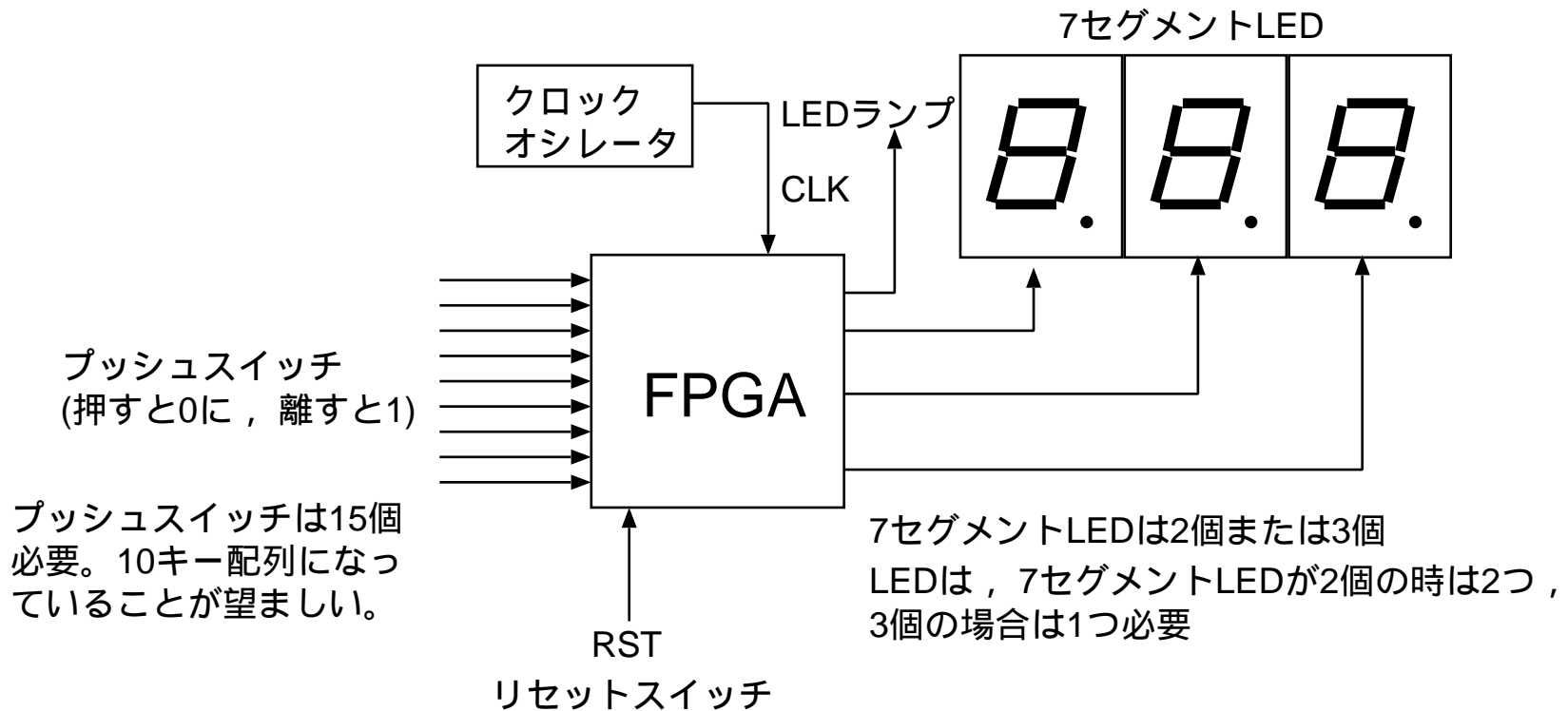
- ◆ その場で10キーを使って動かせる。
- ◆ プロセッサだと、プログラムを考えたり、メモリとのインタフェースが必要
- ◆ ただし、入力が非同期に入るので、同期変換しないといけない。
- ◆ 簡単なようで奥が深い。

# BCDと2進数

---

- ◆ 計算機は2進数だが、人間は10進数
  - BCD: 10進数を2進数であらわす  
94 = 101\_1100 2進数の場合  
1001(9)\_0100(4) BCDの場合
- ◆ BCDで計算することも可能だが、面倒
- ◆ 入力でBCDを2進数に変換し、出力を再び10進数に変換する。2進数で計算できる。

# FPGA回路（ボード）の仕様



# BCD2桁入力2進記憶回路

---

10キーを2回押して、2桁の10進  
数を入力する回路を設計

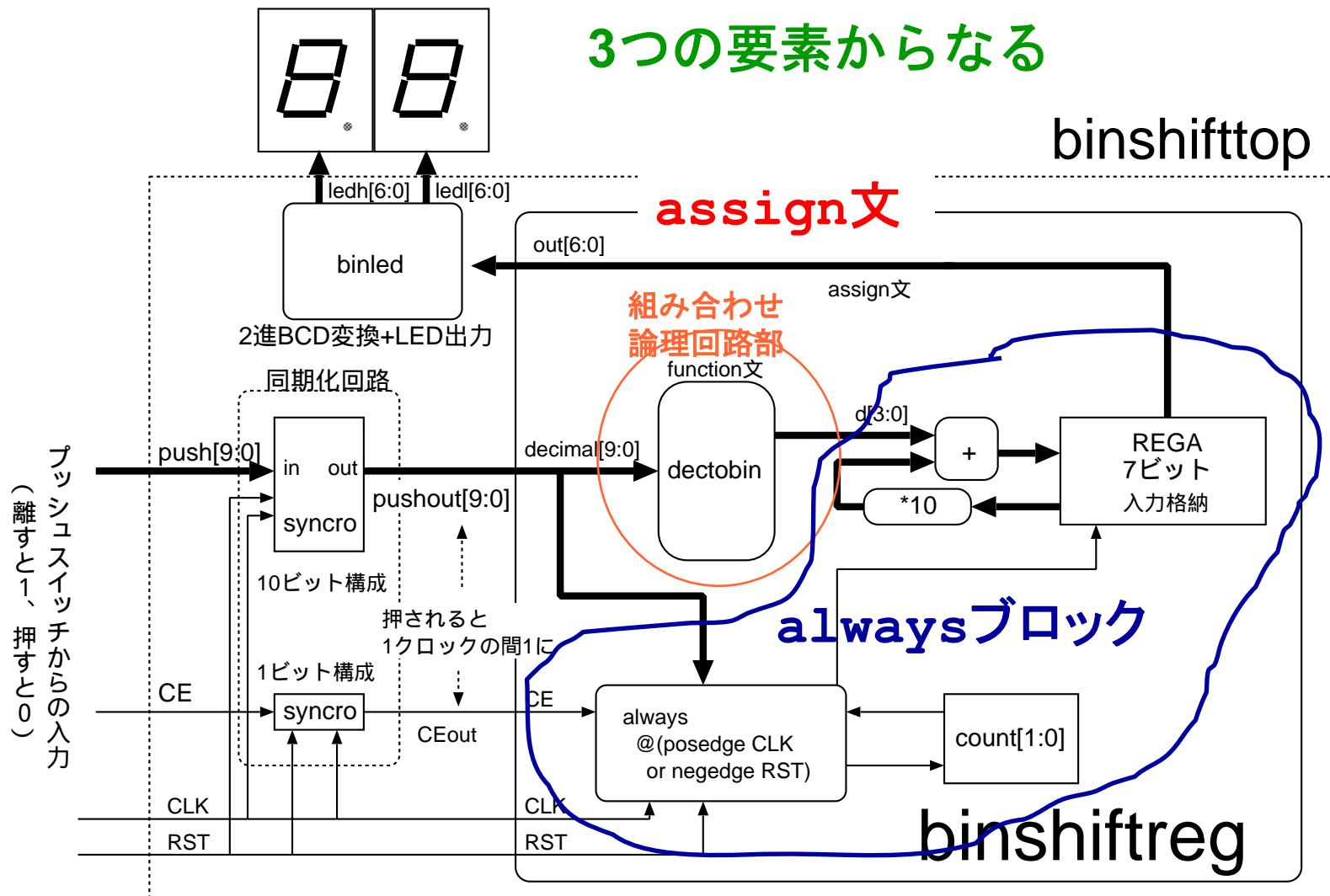
# BCD2桁入力2進記憶回路

プッシュスイッチから与えられたBCDを2進数に変換して保存する

module 名	binshiftreg	
入力ピン	decimal[9:0]	10 キーからの入力 decimal[0] が 0 キー, decimal[9] が 9 キーに対応.
	CE	入力をクリア
	CLK, RST	クロックとリセット
出力ピン	out[6:0]	格納した数の 2 進数出力. binled へ渡す.

# BCD2桁入力2進記憶回路

3つの要素からなる



# 処理の流れ

---

1. 10キーの入力を2進数に変換(dectobin)
  - function文で実現
2. 2進数をレジスタに入力。
  - 前の入元に10をかけて現在の入力を入れる。  
入力 2,1 → 表示 21
  - alwaysブロック
3. 10進数に変換して出力
  - assign文＋外付け回路(binled)

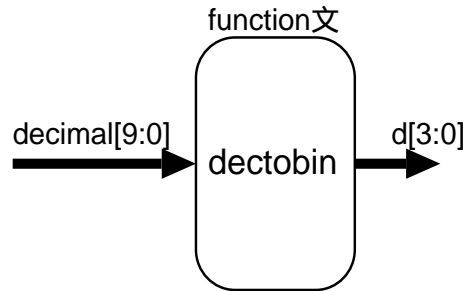


# 入出力ポートの定義

---

```
module binshiftreg(out,decimal,CLK,RST,CE);  
// 出力ポート、入力ポートの順に書くほうがよい  
output [6:0] out;  
input [9:0] decimal;  
input CLK,RST,CE;  
// ↑ input, outputの定義はビット幅毎に  
endmodule
```

# functionとassignによる組み合わせ論理回路

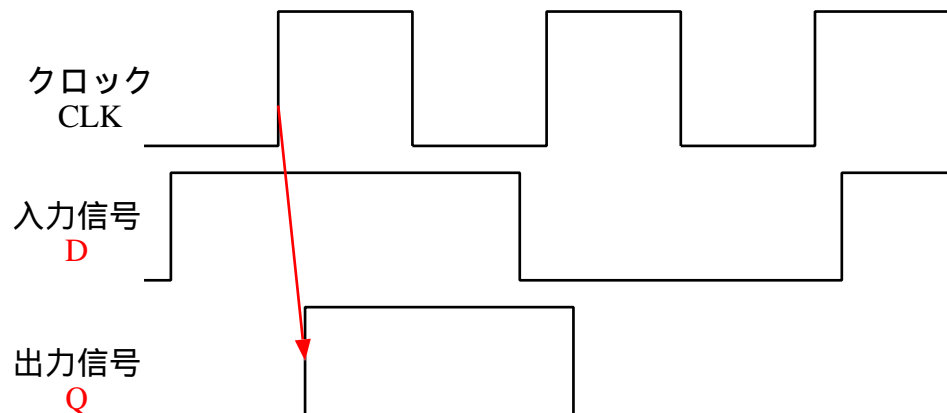
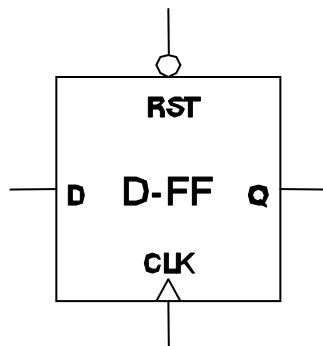


- 9を押すとdecimal[9]が1に
- decimal[x]をxに変換する

```
module binshiftreg(out,decimal,CLK,RST,CE);
output [6:0] out;
input [9:0] decimal;
input CLK,RST,CE;
wire [3:0] d; // ← assignで代入されるノードはwireで定義しておく.
assign d=dectobin(decimal); // ← function dectobinの出力をdに入力
function [3:0] dectobin; // ← [3:0]は出力のビット幅を定義
input [9:0] in; // ← function文の引数を定義
    if(in[9]) //← functionの中には自由にif, caseが書ける
        dectobin = 9;
    else if(in[8])
        dectobin = 8;
    else if(in[7])
        dectobin = 7;
    中略
    else if(in[0])
        dectobin = 0; // 最後のelseがなくても組み合わせ回路になる
endfunction
endmodule
```

# 同期順序回路の記述

- ◆ `always @(posedge CLK or negedge RST)`
  - クロック(CLK)の立ち上がりと、リセット(RST)の立下りに常に反応する。(非同期リセットつきD-FF)
  - RSTは回路構成上の都合で、0のときにリセットがかかるのが普通



# 同期順序回路の記述(2)

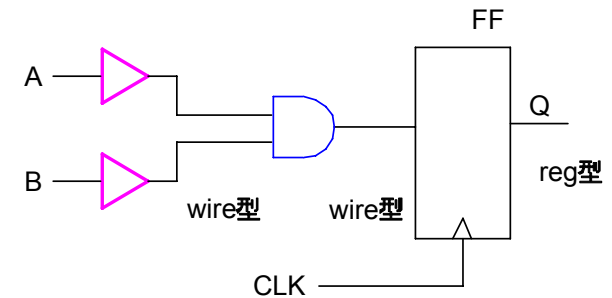
---

```
module ff (out,in,CLK,RST);  
    input in,CLK,RST;  
    output out;  
    reg out;// 記憶素子はreg型  
    always @(posedge CLK or negedge RST)  
        if(!RST) //リセットの場合  
            out<=0;  
        else  
            out<=in;  
endmodule
```

# 信号(変数)の型とビット幅

## ◆ 信号(変数)の型

- reg型: 値を記憶しておく
- wire型: 配線と等価（変化が常に伝搬）
  - » 宣言しないと, 1ビットのwireに
  - » ただし, 宣言しないとエラーにする処理系も多数出てきている。



## ◆ ビット幅

- [31:0] 32ビットの信号(バス)
  - » 特に必要な場合を除き、MSB>LSBとする  
(MSB=Most Significant Bit:最上位ビット)
- 利用時は, W[0], W[2:0], W

```
module m1(A,B,Q);  
    input A,B;  
    output Q;  
    reg Q;  
    wire w0, w1, w2;  
  
endmodule
```

# ブロッキング代入とノンブロッキング代入

## ◆ Verilogでの代入には2種類ある

- `<=` : ノンブロッキング代入(そこで処理をとめない (blockしない). 現時間のイベントをすべて評価していっせいに実行)
- `=` : ブロッキング代入(そこで処理をとめる). その時点で実行

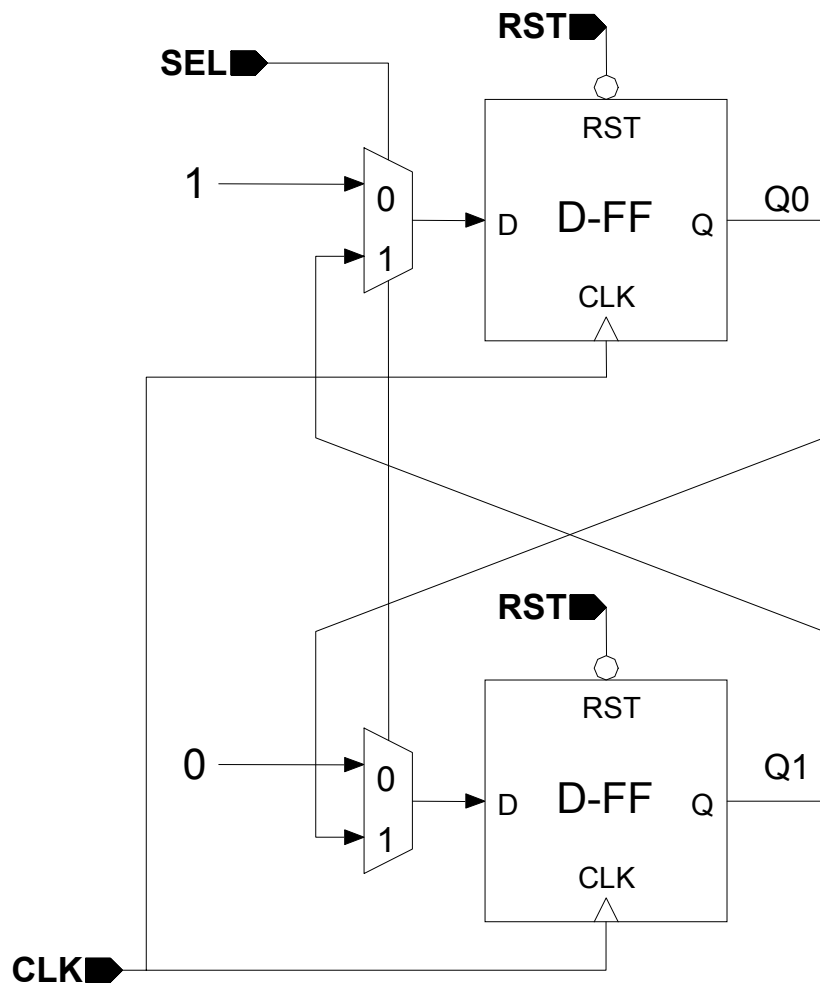
```
module nonblocking;
  reg a,b;
  initial
  begin
    a<=0;b<=1;
    #10
    a<=b;b<=a;
    #10
    $finish;
  end
  initial
    $monitor("a=%b,b=%b",a,b);
endmodule
```

値は交換  
される

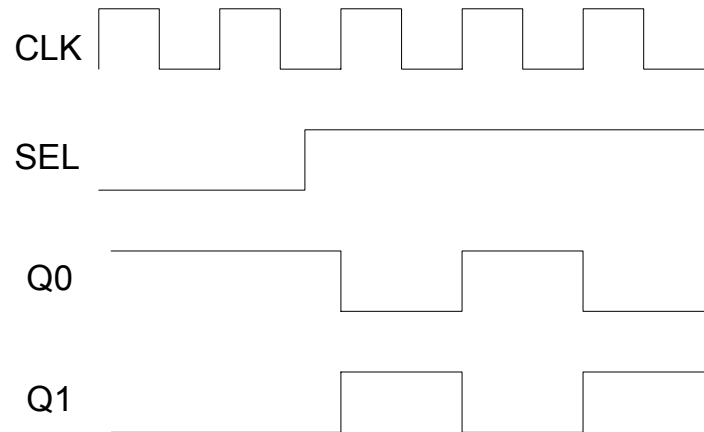
```
module blocking;
  reg a,b;
  initial
  begin
    a=0;b=1;
    #10
    a=b;b=a;
    #10
    $finish;
  end
  initial
    $monitor("a=%b,b=%b",a,b);
endmodule
```

値は交換  
されない

# D F F の動作(ノンブロッキング代入)



◆ CLKが入るとD F Fの値は？



◆ 毎回反転する

# ノンブロッキング代入

---

- ◆ always @(posedge CLK or negedge RST)内では必ず、ノンブロッキング代入文を使う
  - ブロッキング代入を使うと、RTLと合成後の回路の動作が合わないことになる
  - D-FFの動作はノンブロッキングだから！！
  - assign文の代入には、ノンブロッキング代入は使えない
- ◆ alwaysを使った組み合わせ回路の場合は、ブロッキング代入を使う



# alwaysによるレジスタ記述

```
reg [1:0] count; ← 入力された10キーの数を格納するレジスタ  
reg [6:0] REGA; ← BCDを2進化した数を格納するレジスタ
```

定義

```
assign d=dectobin(decimal);
```

```
function [3:0] dectobin;
```

中略

```
endfunction
```

```
always @(posedge CLK or negedge RST)
```

非同期リセット

```
if(!RST)
```

```
begin
```

```
    REGA<=0;count<=0; ← 初期化
```

初期化

```
end
```

```
else if((decimal != 0) && (count < 2))
```

```
begin ↑ decimalが0でなく(入力がある), countが2未満なら
```

```
    REGA<=(REGA*10)+d; ← REGAを10倍してdを足し, REGAに格納
```

```
    count<=count+1; ← countを1あげる.
```

```
end
```

```
else if(CE)
```

動作

```
begin
```

```
    REGA<=0;count<=0; ← CEですべてのレジスタをクリア
```

```
end
```

# assignによる出力ポート接続

```
reg [1:0] count; ← 入力された10キーの数を格納するレジスタ
reg [6:0] REGA; ← BCDを2進化した数を格納するレジスタ
assign d=dectobin(decimal);
function [3:0] dectobin;
  中略
endfunction
always @(posedge CLK or negedge RST)
  中略
  assign out=REGA;
```

- ◆ 出力ピンはreg型で定義可能
  - その場合は、同じ変数名になるのでassign文は不要
  - ただし、同じビット幅で定義する！
- ◆ 入力ピンはreg型で定義できない
  - 入力の外から与えられるので

# 同期化回路

- ◆ 非同期入力をそのまま同期回路に入れると、誤動作を起こす。
  - メタステーブル: 一時的に発振する
- ◆ 入力は1クロックのみアクティブなほうが回路が書きやすい。

例 : INが1になったらカウントアップ

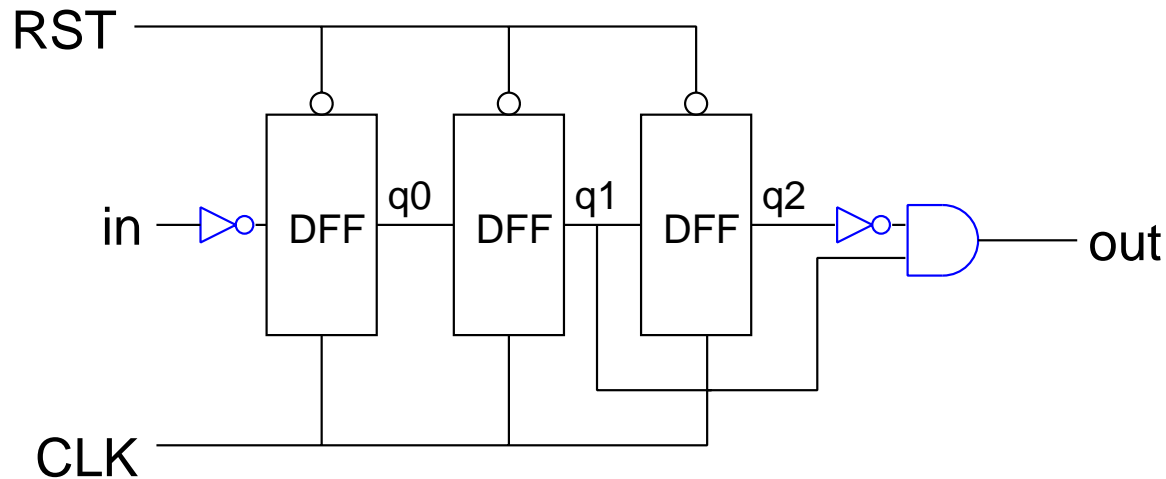
```
always @(posedge CLK)
  if(IN==0)
    0である.
  else if(IN==1)
    if(その前が0だったら)
      カウントアップ
```

同期化回路なし

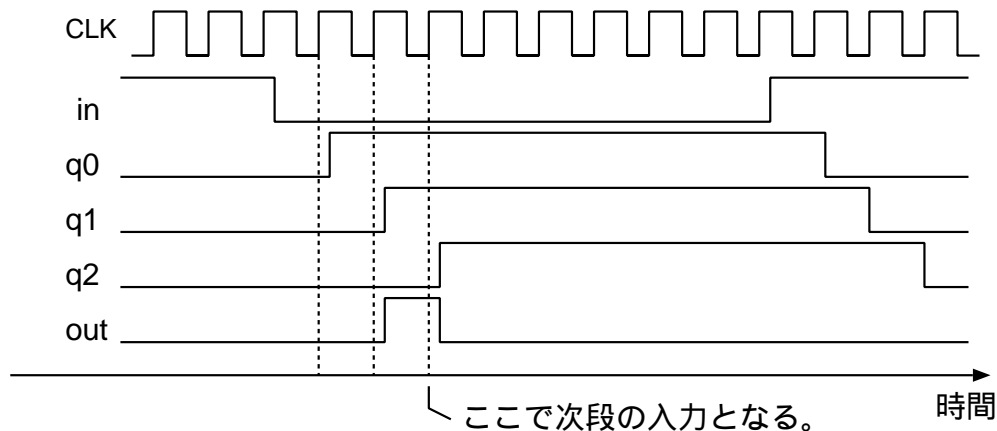
```
always @(posedge CLK)
  if(IN==1)
    カウントアップ
```

同期化回路あり

# 同期化回路



◆ FFを複数段  
接続してメ  
タステーブ  
ルの伝搬を  
防ぐ



# binshifttop(最上位回路) の設計

---

## ◆ 構成要素

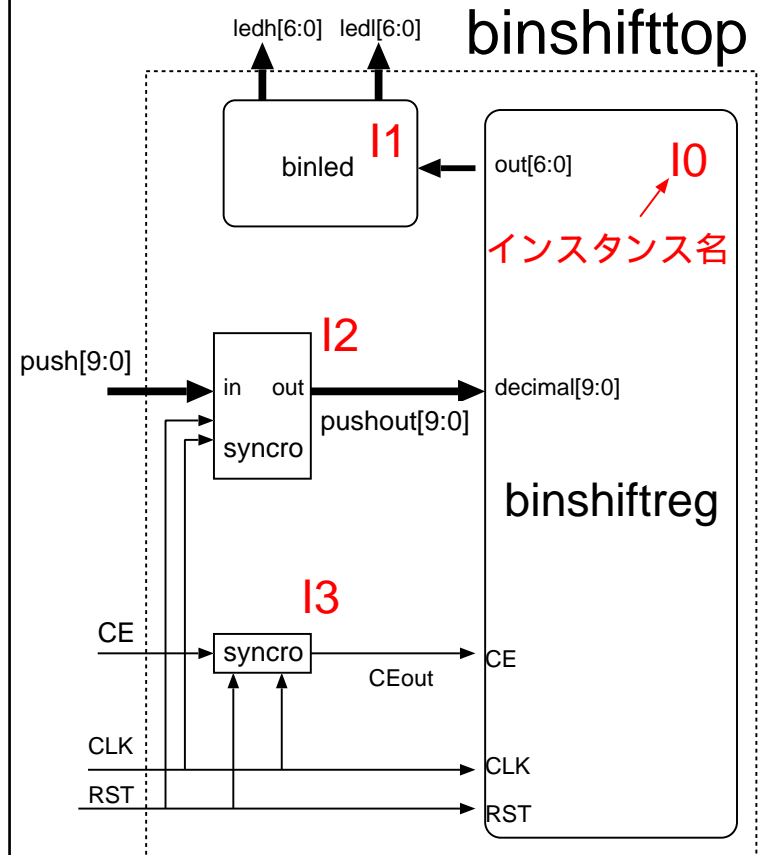
- 同期化回路
- binshiftreg
- 出力変換回路(binled)
  - » 2進出力を10進に変換し、さらにLEDへの出力に変換

## ◆ 構成要素をインスタンスとして階層的に記述する。

- インスタンス: 回路の中のサブ回路
- 適切な階層分割が回路の可読性を高める。

# binshifftopの設計

```
module binshifftop (push,ledl,ledh,CLK,CE,RST);
  input [9:0] push;// 10キー.
  input CLK,RST,CE;
  output [6:0] ledl, ledh;
  wire [6:0] out;
  wire [9:0] pushout; //←内部バスの信号定義
  wire CEout; //1ビットのワイヤでも必ず定義
  syncro #(1) I3(.in(CE),
    .out(CEout),.CLK(CLK),.RST(RST));
  syncro #(10) I2(.in(push),.out(pushout),
    CLK(CLK),.RST(RST));
  binled I1(.in(out),.ledl(ledl),.ledh(ledh));
  binshiftreg I0(.decimal(pushout),.CLK(CLK),
    .RST(RST),.CE(CEout),.out(out));
  // ↑ binshiftregのdecimal入力に, syncroのout出力を
  // pushoutを介して接続している.
endmodule
```



# 階層記述

```
binshiftreg I0(.decimal(pushout),.CLK(CLK),.RST(RST),.CE(CEout),.out(out));
```

- ◆ binshiftreg: 使用する回路のmodule名
- ◆ I0: インスタンス名 **上位回路中で一意**
  - 同じmoduleが階層中に1個しか使わないのなら、module名と同じでよい
- ◆ .decimal: 下位回路のピンの名前
- ◆ (pushout): 上位回路のネットの名前

## 書式

```
LowerModule InstName(.LowerModulePin(UpperModuleNet),..);
```

# 階層的な回路の記述

```
module moduleA (in,out,CLK,RST );  
  input in,CLK,RST;  
  output out;  
endmodule
```



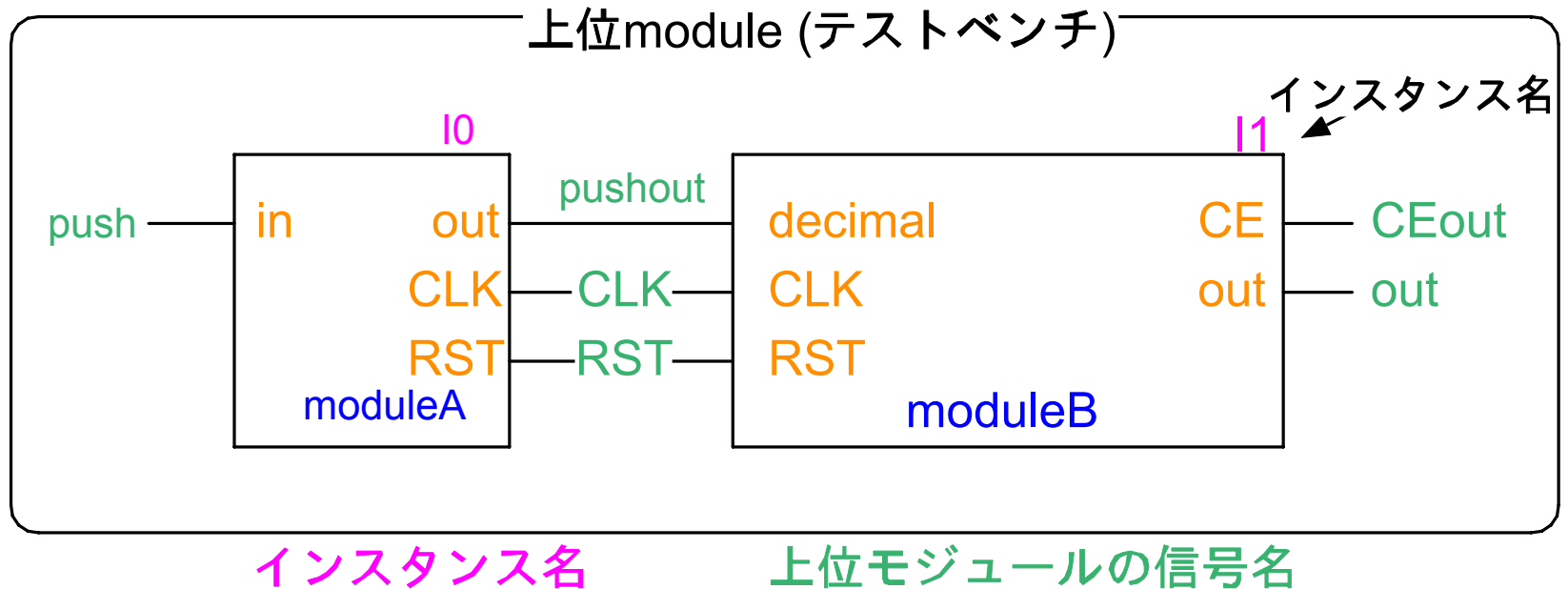
```
module moduleB (decimal, CLK, RST, CE, out );  
  input decimal, CLK, RST;  
  output CE, out;  
endmodule
```



◆ この二つのモジュールを回路中で接続したい。



# 階層的な回路の記述2



```
moduleA I0(.in(push),.out(pushout),.CLK(CLK),.RST(RST));
```

モジュール名      下位モジュールのピン名

```
moduleB I1(.decimal(pushout),.CLK(CLK), .RST(RST),  
.CE(CEout),.out(out));
```

# parameterによる可変長回路

```
module syncro(out,in,CLK,RST);  
    parameter WIDTH = 1; //パラメータと初期を定義  
    input [WIDTH-1:0] in; // 入力ピンをパラメータ化  
    output [WIDTH-1:0] out;  
    input CLK,RST;  
    reg [WIDTH-1:0] q0,q1,q2;  
    .....  
endmodule
```

- ◆ 同じ記述で、ビット幅が異なる回路を実現可能
- ◆ 書式 :  
parameter 名前=初期値;

# 上位回路でのパラメータの指定

- ◆ Verilogでは, 上位moduleから下位moduleのparameterを与える方法が2種類存在する.
  - defparam文を使用する
  - #(param1, param2, ...)で指定する。パラメータは, module内でparameter文で宣言した順番となる
  - シミュレータ, 合成系によって, サポートしている場合としていない場合があるので注意

```
module binshifftop (push,ledl,ledh,CLK,CE,RST);  
    defparam binshifftop.I2.WIDTH=10;  
    defparam binshifftop.I3.WIDTH=1;  
    syncro I3(.in(CE), .out(CEout),  
              CLK(CLK),.RST(RST));  
    syncro I2(.in(push),.out(pushout),  
              CLK(CLK),.RST(RST));  
endmodule
```

defparam

```
module binshifftop (push,ledl,ledh,CLK,CE,RST);  
  
    syncro #(1) I3(.in(CE), .out(CEout),  
                  CLK(CLK),.RST(RST));  
    syncro #(10) I2(.in(push),.out(pushout),  
                   CLK(CLK),.RST(RST));  
endmodule
```

#(param1, param2)

# シミュレーションによる動作確認

---

- ◆ あらかじめ、GUIを用いたテストフィクスチャを用意
  - C言語によりverilogシミュレータを拡張(PLI)
- ◆ WEBよりダウンロード
- ◆ verilogに必要なファイルをすべて引数で与える
- ◆ GUI上のLED出力により確認

```
gtksim.sh binshiftsimgtk.v binshifttop.v binshiftreg.v other.v
```

# 文法エラーのチェック

---

- ◆ シミュレーションを実行する前に，文法エラーがないかを先にチェックしておく
  - ほとんどすべてのverilogシミュレータで-cオプションにより，コンパイルのみを実行可能

## binshiftregのみのチェック

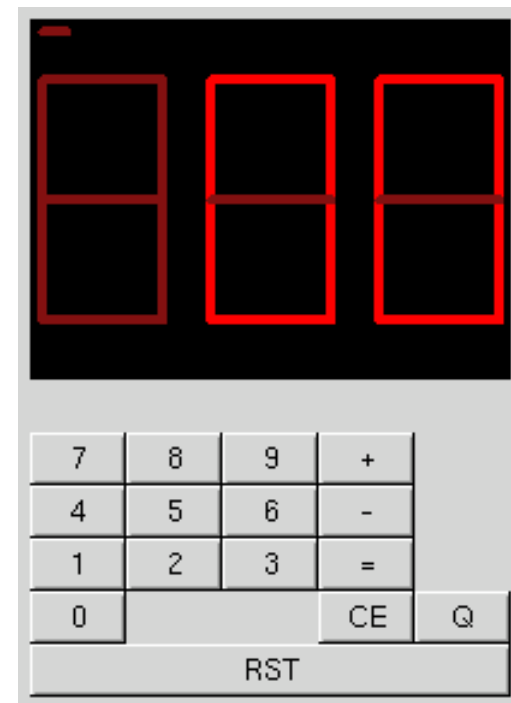
```
% verilog -c binshiftreg.v
```

## 全体のチェック

```
% verilog -c binshifttop.v binshiftreg.v other.v
```

# GUIの操作法

- ◆ 各ボタンはボード上のボタンに対応
  - ただし、Qをのぞく
- ◆ シミュレーションの終了はQ
- ◆ RSTはボードのRSTボタンに対応
- ◆ 左上のLEDは、電卓のオーバーフロー表示用



# デバッグの方法

---

- ◆ うまく動作しない場合は、シミュレーション終了後に波形ファイル(binshiftsim.vcd)をsimvisionを用いて表示する。

```
% simvision binshiftsim.vcd
```

- ◆ binshiftsimgtk.vの\$monitor文に見たい信号を追加しても良い
  - 同じmodule内に複数の\$monitorを書いてもひとつしか有効にならないので注意。

# Verilog HDLを記述する上での注意

- ◆ 常に回路を意識して記述する
  - 回路にならない記述はシミュレーションできても合成できない
- ◆ 例えば,
  - 複数のalwaysブロックで同じreg型変数に代入する
  - moduleの外に出ていない信号を他のmoduleで使用する
    - » Verilogではmodule内部の信号は、すべてローカル変数

```
always @(posedge CLK or negedge RST)
begin
    a<=b;
end
always @(posedge CLK or negedge RST)
begin
    a<=c;
end
```

駄目な例



# function文の落とし穴

- ◆ function内では， inputで定義した変数のみ使用が許されているはず。
  - module内で定義されている変数を使ってもシミュレーションは動くが，動作が変になる
  - input文で宣言する変数も， module内で使用していないものに

```
module m(a,b,c);  
  input a,b;  
  output c;  
  function fa;  
    input a; //同じ変数名は使用しない！  
    fa=a+b;  
  //module内で使用している信号は使用しない  
endfunction
```

こちら  
は駄目

```
module m(a,b,c);  
  input a,b;  
  output c;  
  function fa;  
    input ai,bi;  
    fa=ai+bi;  
  endfunction
```

こちら  
はOK

# alwaysを使った組み合わせ回路

- ◆ assignではかけないような複雑な組み合わせ回路を記述できる
- ◆ 気をつけないと意図しないラッチが生成される
  - ブロック内部で利用する信号はすべて always@の後に列挙
  - case文を利用する場合は、すべての場合を列挙する

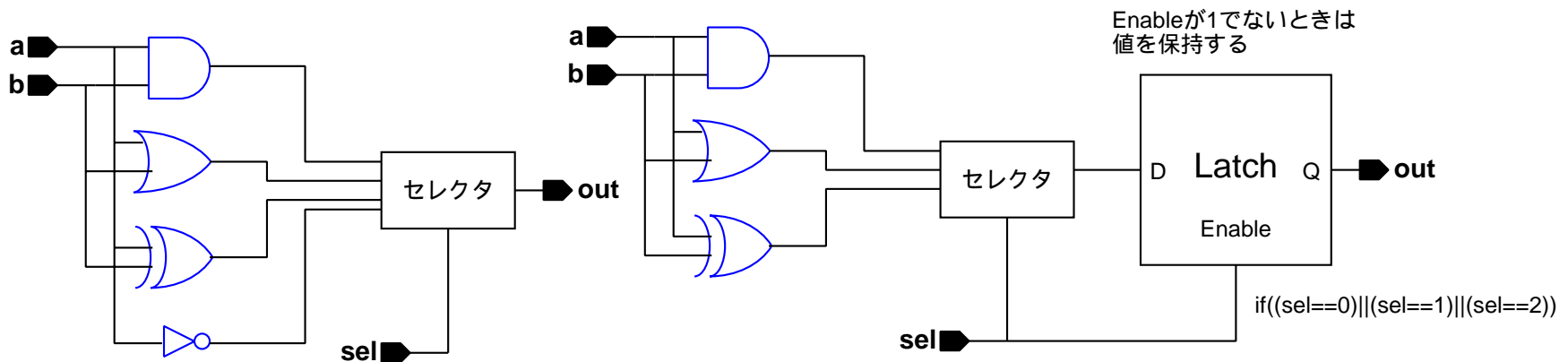
```
module combi (out,sel,a,b);
  input [1:0] sel;
  input a,b;
  output out;
  reg out;//必ずreg型にする
  always @(sel or a or b) //すべての信号を列挙
  begin
    case (sel)
      0:
        out=a&b;
      1:
        out=a|b;
      2:
        out=a^b;
      default: //すべての場合を列挙
        out=~a;
    endcase // case(sel)
  end // always @ (sel or a or b )
endmodule // combi
```

# ラッチの生成

## ◆ 同期回路にラッチは不要！

- ラッチとは、クロックに同期しないで値を保持する素子のこと

## ◆ always文で組み合わせ回路を書くと、ラッチが生成されることがある。



Latchなし

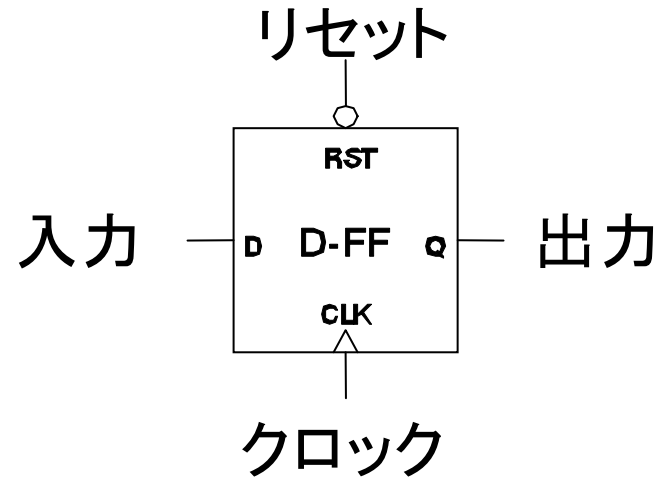
Latchあり

# Verilog-HDLにおける組み合わせ回路

- ◆ always文, function文のどちらかで記述する
  - functionの場合, 代入されるネットはwire型
  - alwaysの場合, 代入されるネットはreg型
- ◆ 単一出力の場合は, function文で記述するほうが簡単
- ◆ 複数出力の場合は, alwaysを使ったほうが簡単
- ◆ always文で記述するとラッチが生成される場合があるので注意
  - 詳細は越智先生の資料を参照
- ◆ 必ずすべてのネットをwireで定義する
  - 1ビットのwireは定義しなくても使用できるが, バグの温床になるので必ず定義する。
  - 最近のシミュレータ, 合成系は定義していないとエラーになる場合もある。
  - `wire [3:0] d;`を消しても, 正常にシミュレーション可能(verilog-xiではwarningすら出ない)

# 非同期リセットと同期リセットの混在

```
always @(posedge CLK or negedge RST)
begin
    if(!RST || CE) ← やってはいけない.
    begin
        REGA<=0;
        count<=0;
    end
end
```



- ◆ 非同期リセットと同期リセットの混在は合成不可
  - 現実のFFでは不可能だから
- ◆ 非同期リセットは電源投入時、誤動作時の初期化にのみ用いる。(PCのリセットボタン)
- ◆ 動作中の初期化は同期的に行う。(PCではCtrl-Alt-Del)

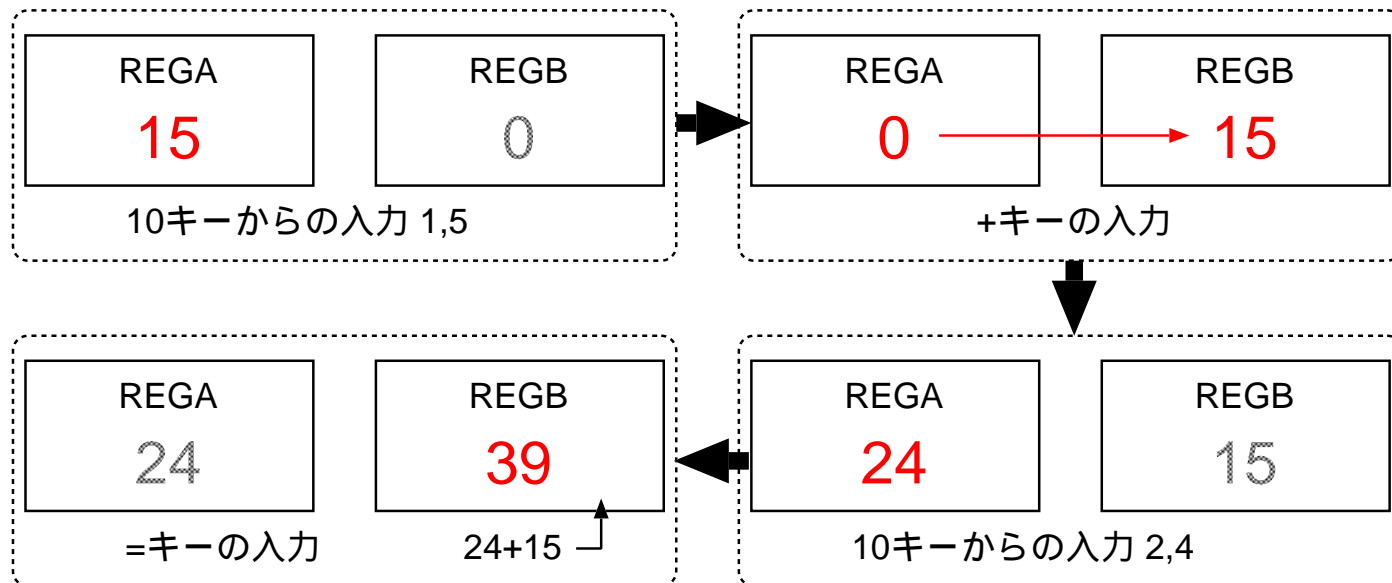
# 演算回路の実現

---

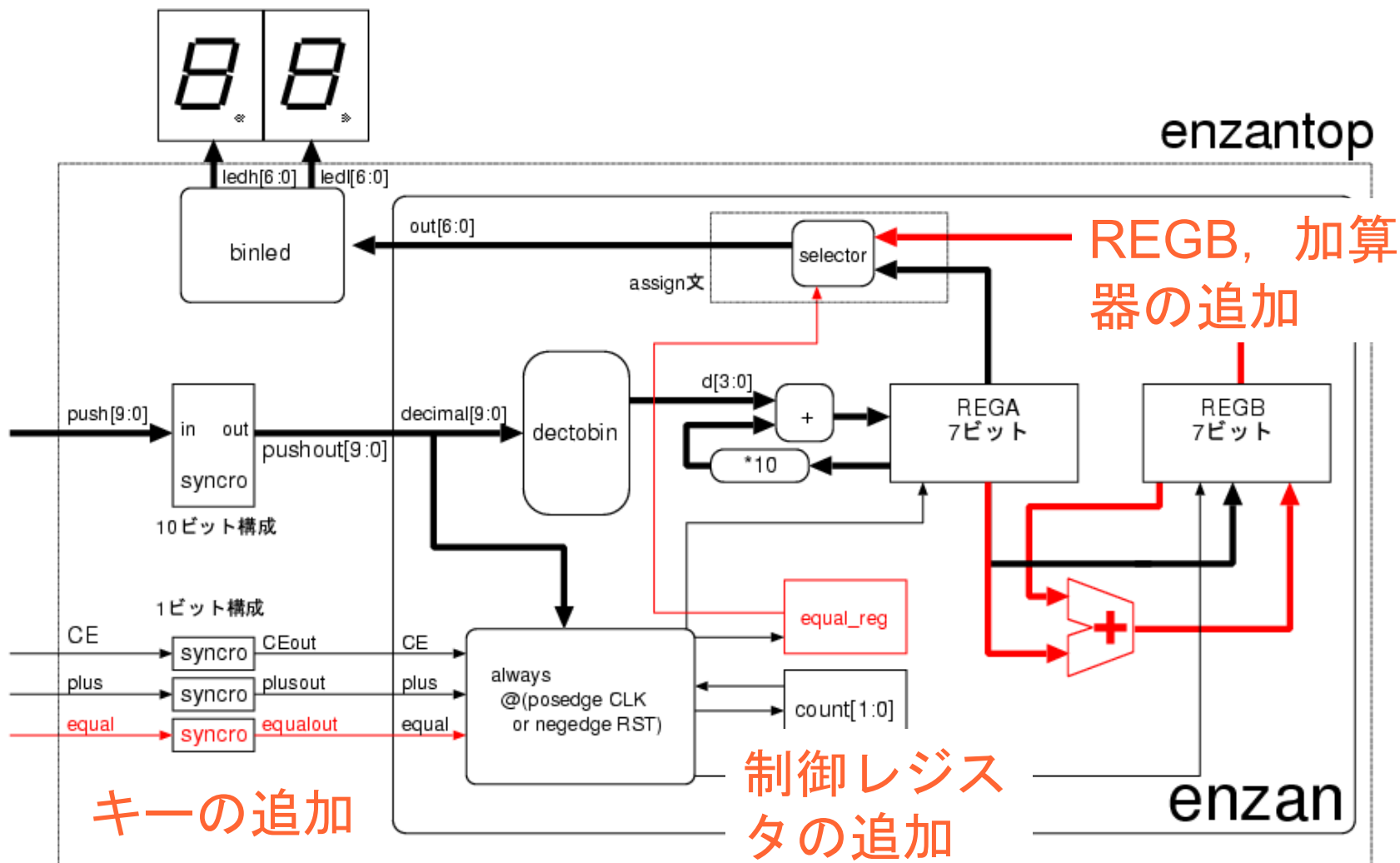
加算機能を付加します。

# 演算回路の実現

- ◆  $10+25=$ が実現できるようにする
- ◆ REGAのほかにもうひとつREGBを用意
- ◆ 手順



# 演算回路ブロック図





# 演算回路のVerilog-HDL記述

---

- ◆ 先ほどのbinshiftregを改造する。

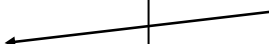
```
module enzan(decimal,plus,equal,CLK,RST,CE,out);  
  input [9:0] decimal;  
  input CLK,CE,RST,plus,equal; ←ピンの追加  
  中略  
endmodule
```

# +キーに対する動作追加

```
reg [6:0] REGA, REGB; ← レジスタの定義
always @(posedge CLK or negedge RST)
begin
  if(!RST)
  begin
    REGA<=0;
    REGB<=0; ← REGBの初期化の追加
    count<=0;
  end
  else if((decimal!=0) && (count < 2))
  begin
    REGA<=REGA*10+d; count<=count+1;
  end
  else if(plus) ← +キーが押されたら
  begin
    count<=0; REGA<=0;
    REGB<=REGA; ← REGAをREGBに移す
  end
```

+が押されると  
REGAの内容をREGB  
に移す。

REGBはそのまま  
なので何も書か  
ない



# =キーに対する動作の追加

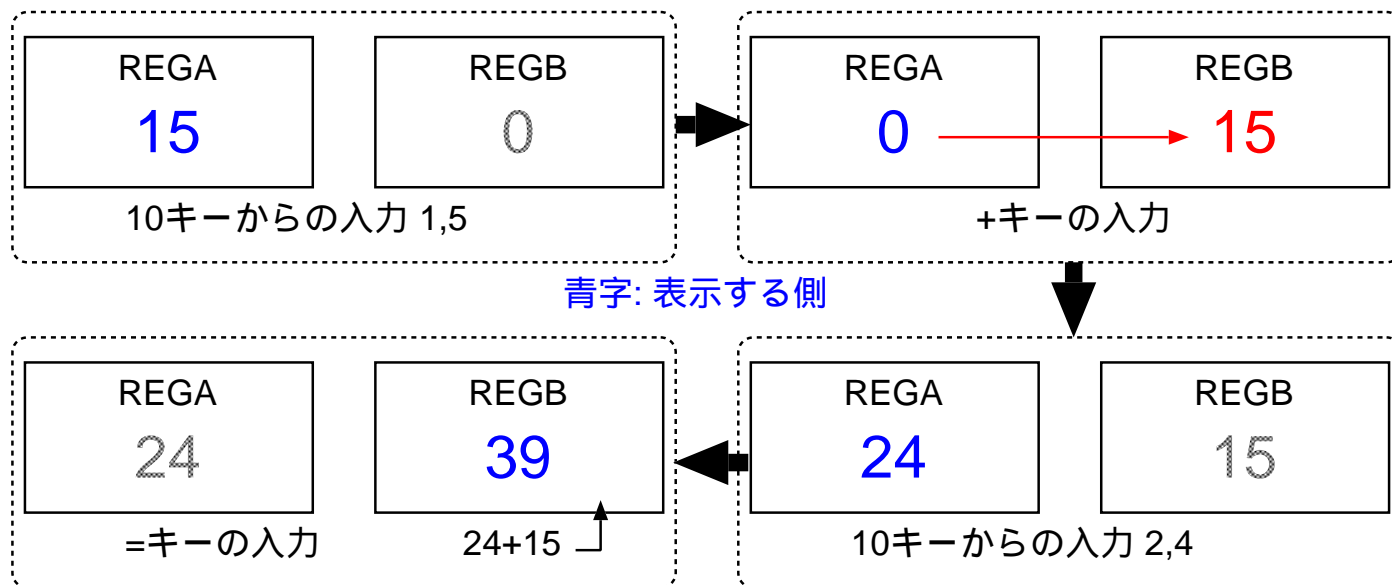
---

=が押されると、加算を行う。

```
always @(posedge CLK or negedge RST)
begin
  if(!RST)
  中略
  else if(equal) //← =キーが押されたら
  begin
    count<=0;
    REGB<=REGA+REGB;//←REGA+REGBをREGBに格納
  end
```

# 出力outの論理

- ◆ =が入力されたら、REGBで、それまではREGAをoutに出力する。
- ◆ =が入力されたことを覚えておくレジスタが必要      `equal_reg`



# 出力outの論理

**reg equal\_reg; 定義**

always @(posedge CLK or negedge RST)

中略

if(!RST)

begin

REGA<=0;REGB<=0;count<=0;

**equal\_reg<=0; ← 初期化**

end

else if(equal)

begin

count<=0;

REGB<=REGA+REGB;

**equal\_reg<=1; ← =が押されたら1にする.**

end

**assign out=(equal\_reg==0)?REGA:REGB;**

**↑ equal\_reg==1ならREGBを出力(selector)**

# enzantopの設計

---

- ◆ binshifftop.vを改造する。
- ◆ RSTを押さなくても、演算が再びはじめられるように改造しても面白い。

# 演算回路のシミュレーション

---

## ◆ シミュレーション方法

- enzansimgtk.vをダウンロードして実行する

```
gtksim.sh enzansimgtk.v enzantop.v enzan.v other.v
```

# 電卓の設計

---

演算回路を電卓にします。



# 電卓の動作

入力	表示	
12	12	
+	12	＋を押してもそのまま
20	20	次の数字を押せば変化
+	32	2回目の＋でその前の加算を実行
5	5	
=	37	
20	20	新しく演算を始める
+	20	
5	5	
=	25	

- ◆ 演算回路では、加算は＝を押した時点しか行っていない。
- ◆ 電卓では次の値を入力するまで、前の値を表示する。

# 電卓の設計

---

- ◆ -99から99までの値を取り扱う。
- ◆ 加算と減算が可能である。演算は＋，－，＝キーを押した時点で行い，10キーから次に入力があるまで，現在の入力もしくは演算結果をLEDに表示する。
- ◆ 加減算の結果が-99より小さいか，99を超える場合，オーバーフローLEDを点灯させて，動作を停止する。
- ◆ 累算ができる。

# 電卓の動作

## ◆ REGAとREGBをうまく制御する。

	入力値 REGA	演算結果 REGB	演算	
キーの入力順 ↓	1	0	+	
	+	1	+	演算実行 REGB<=REGB+REGA;
	5	1	+	
	-	6	+	演算実行 REGB<=REGB+REGA;
	9	6	-	
	=	-3	-	演算実行 REGB<=REGB-REGA;

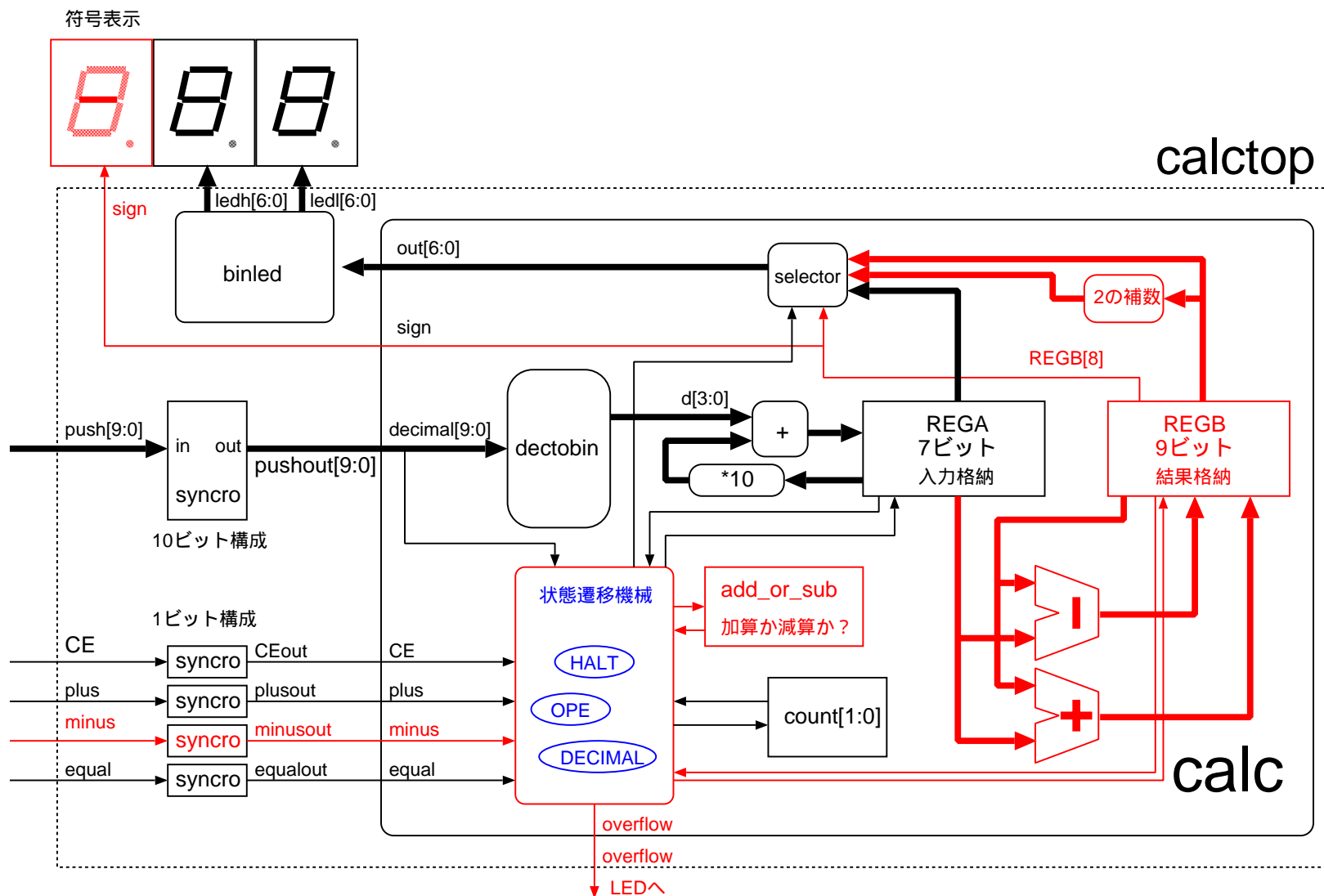
 LEDに表示する側

# 電卓の動作

---

- ◆ 10キーからの入力をREGA に格納する. REGA をLEDに出力
- ◆ +, -キーが来たら, 前回入力された+, -キーにしたがって, REGA, REGB の演算を実行して, REGB に格納する. REGB をLEDに出力する.
- ◆ 10キーからの入力があった時点で, LEDへの出力をREGA にする.
- ◆ +, -, =キーで, REGA とREGB の演算を実行して, REGB に格納する.

# 電卓のブロック図



# 設計手順

---

- ◆ module 部の記述を行う。
- ◆ 状態遷移機械を記述する。
  - 状態遷移をリセットの次に記述
- ◆ 各状態での動作を記述する。
  - 状態と入力によるレジスタの動作、その後の状態遷移
- ◆ 出力の部分の論理を記述する。

# module 部の設計

## ◆ 必要なレジスタの決定

- REGA: 入力用 0~99まで7ビット
- REGB: 計算結果格納 演算結果は-99-99=-198, 99+99=198まで9ビット
- add\_or\_sub: 演算が加算か減算か覚えておく 1ビット

	入力値 REGA	演算結果 REGB	演算
1	1	0	+
+	1	1	+
5	5	1	+
-	5	6	+
9	9	6	-
=	9	-3	-

キーの入力順

  LEDに表示する側

# 状態を作る（状態遷移機械）

- ◆ 表示する値にあわせて、状態を作成する。

動作	LED への出力	状態
10 キー入力時	10 キーからの入力値 (REGA)	DECIMAL
+, -, = 入力時	演算結果 (REGB)	OPE
オーバーフロー時	オーバーフロー (overflow) を示す LED を点灯	HALT



状態遷移図



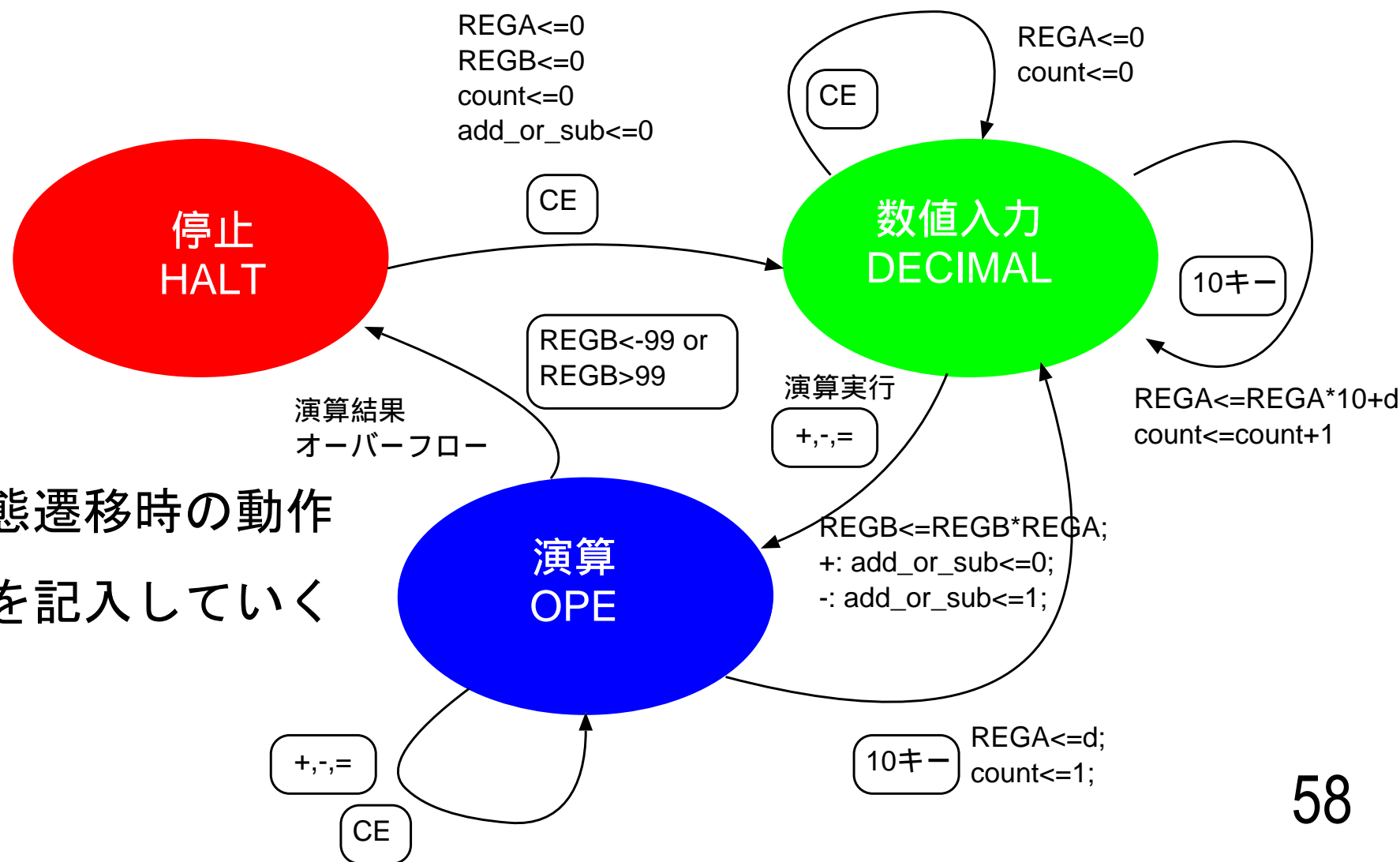


# 状態の記述

- ◆ `define文を使って、状態を名前で定義する。
- ◆ 状態遷移による条件分岐をリセットの次に高い条件とする。

```
`define DECIMAL 0
`define OPE 1
`define HALT 2
.....
reg [1:0] state; ←2ビットで定義する。
if(!RST)
    begin .... end
else begin
    case(state)
        `DECIMAL:
            .....
        `OPE:
```

# 各状態での動作の決定



# リソースシェアリング（資源の共有）

- ◆ 同時に使用しない演算器を共有する。
- ◆ 記述の仕方によって、共有されたりされなかったりする。

例 REGB<=(add\_or\_sub==0)?REGB+REGA:REGB-REGA;

☆add\_or\_subの値で、減算か加算か切り替える  
☆加減算は同時に行わないので、加算器ひとつで実行できる

# リソースシェアリング(2)

## ◆ 加減算器の3つの書き方

```
REGB<=(add_or_sub==0)?  
    REGB+REGA:REGB-REGA;
```

(i)

```
if(add_or_sub==0)  
    REGB<=REGB+REGA;  
else if(add_or_sub==1)  
    REGB<=REGB-REGA;
```

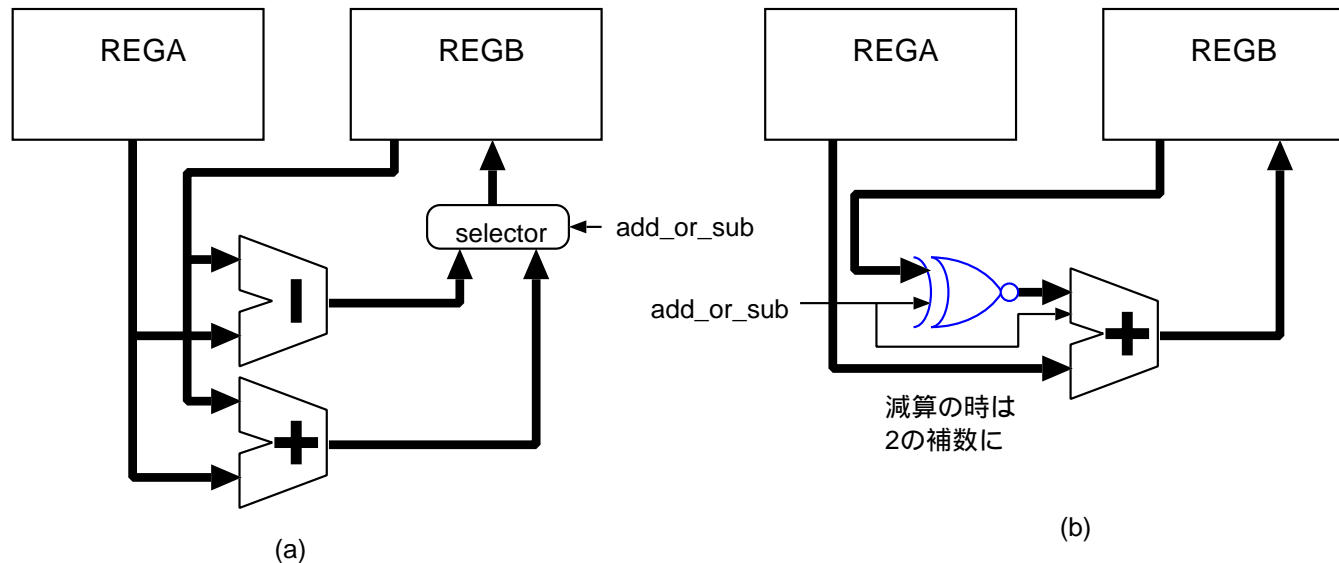
(ii)

```
REGB<=  
    REGB+((add_or_sub?)?REGA:(~REGA+1))
```

(iii)

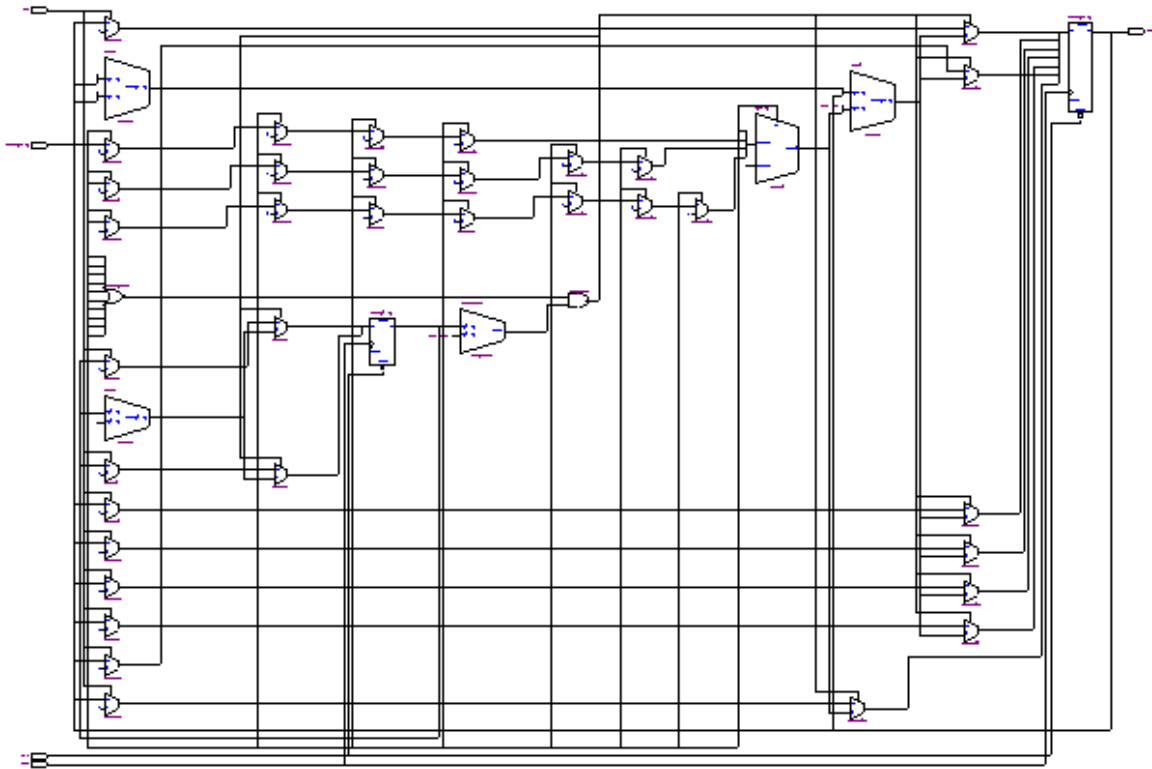
# リソースシェアリング(3)

- ◆ 記述・ツールによって生成される回路はまちまち
- ◆ Synopsys社のDesign Compilerだと (I), (ii), (iii) とともに(b)となる。
- ◆ Synplify\_proだと、(i), (ii)のみ、(b)になる。



# QuartusでのRTL確認方法

- ◆ Compile終了後， Tools→RTL Viewerを実行する



binshiftreg

# 減算および負の数の取り扱い

- ◆ 負の数は2の補数で取り扱う
- ◆ 2の補数=ビット反転+1
  - 正の数を表すのに必要なビット数+1で表す。
  - C言語では
    - » char 符号付8ビット -128~127まで
    - » unsigned char 符号なし8ビット 0~255まで
- ◆ 最上位ビットは符号ビット

## 例題

8ビットで表現するときの-25

25=0001\_1001 → ひっくり返して1110\_0110

→ 1を足して 1110\_0111

# Verilogにおける負の数の取り扱い

---

- ◆ Verilogでは、reg型で明示的に負の数が扱えなかった
  - したがって、教科書の記述のように、面倒なことをしないといけない
- ◆ それを解消するために、Verilog 2001が制定
  - signedによる負の数の取り扱い
  - 多次元配列
  - その他
  - 詳しくは、“VERILOG 2001” by Stuart Sutherland, Kluwer Academic Publishers



# signedによる負の数の取り扱い

- ◆ reg型、wire型は、そのままでは負の数の評価ができない。
- ◆ signedをつけると負の数で評価可能

```
reg signed [4:0] A;
```

```
reg [4:0] B;
```

```
if(A<-10) ○評価可能
```

```
if(B<-10) ×評価不可能: 常に成立するか, 常に不成立
```

# signed拡張ありの負の数の取り扱い

```
module inverse;
  reg signed [4:0] A,B,C;
  initial
  begin
    A=3;B=-2;
    $display("A=%d,%b, B=%d,%b",A,A,B,B);
    C=8-5; //←結果は3
    #100
    C=5-8; //←結果は-3
    #100
    C=-10-8; //←結果は-18(オーバーフロー)
    #100
    C=10+10; //←結果は20(オーバーフロー)
  end
  initial
    $monitor("%d: ",$time,"C=%d, %b",C,C);
endmodule
```

# シミュレーション結果

---

A= 3,00011, B=-2,11110 // 2の補数で格納

0: C= 3, 00011 // 8-5

100: C=- 3, 11101 // 5-8

200: C=14, 01110 //-10-8;

300: C=-12, 10100 //10+10;

# 負の数

---

- ◆ 負の数を正しく表示するには最上位ビットの値で判断する。
- ◆ ただし、オーバーフローしたら駄目
- ◆ オーバーフローしないようにビット幅を決める。
- ◆ signedを使う上での注意点
  - 現在はほとんどすべてのツールがsignedをサポートしている
  - 一部の古いツールではサポートされていない場合もあり

# 負の数による条件判断

---

- ◆ REGB (9ビット) のオーバーフロー判定
  - -99より小さいか、99より大きい
  - `if((REGB<-99)|| (REGB>99))`
  - Signedにより非常に簡単に
  - 出力(out)の2の補数化
  - LEDへの出力は2の補数にする。
  - 条件判断は `if(REGB>0)` でOK。

# ボードの仕様による変更

---

- ◆ 今回使っているボードは、LEDをドライブしないと点灯するので、`calctop.v`の出力ピンが、`sign`ではなく、`ledsign[6:0]`となっている。
- ◆ `calc.v`は変更なし

# 回答例の不具合

---

- ◆ 現在の回答例では、 $=$ の後に演算を続けることができません。
  - $60+5=-5$ ができない
  - こちらは数行加えると修正可能
- ◆  $=$ の後に新規に演算を行うこともできない
  - $60+5=60-5$ ができない
  - 10キー入力が $=$ を押した後か、 $+-$ を押した後かを判断する必要あり
    - » enzan.vのequal\_regを利用する。

# Verilog Simulator

---

- ◆ Cadence社 verilog-xl, ncverilog(高速なverilog simulator)
  - VDECのメディアではIUS, LDVに含まれる
  - 波形を見るのはsimivision
- ◆ Synopsys社 vcs
  - VDECのメディアでは, vcsに含まれる
- ◆ Mentor Graphics社 modelsim
  - VDECのメディアでは, Modelsim
  - XILINX, ALTERAのFPGAソフトに機能限定版
- ◆ Plagmatic C社 GPL Cver
  - GPLライセンスにより, フリーで利用可能
  - Windows(cygwin), linux, Solaris, OS Xなどで動作
  - 波形を見るのはgtkwave
- ◆ Veritakwin (菅原システムズ)
  - 純国産Verilogシミュレータ
  - 無料で使えるCQ版もあり [http://verilogician.net/tools/Veritak/CQ\\_Version/](http://verilogician.net/tools/Veritak/CQ_Version/)



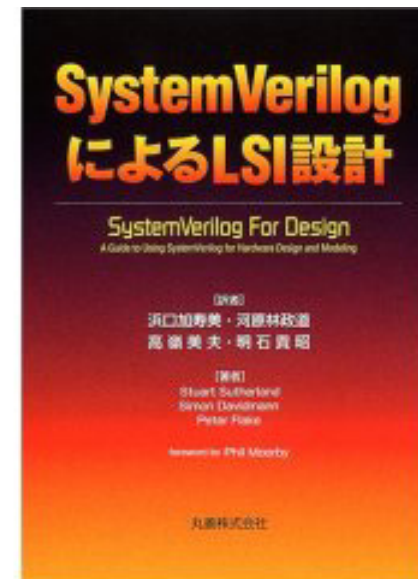
# System Verilog

---

- ◆ VerilogにVHDL, C++の要素を加えた規格
- ◆ Verilogの持つあいまい性を徹底的に排除
  - Verilog: always文は, 組み合わせ回路, 順序回路の両方を記述可能
  - System Verilog: always\_comb, always\_ff
  - logic: wire, regを統合した信号タイプ
- ◆ 既存のVerilogとの互換性を維持
- ◆ SystemVerilog 対応状況
  - Design Compiler, ncverilog, Quartusなどすでに対応済み
  - 詳しくは, [www.systemverilog.org](http://www.systemverilog.org)

# 参考書

- ◆ RTL設計スタイルガイド
  - STARCホームページより購入可能
    - » <http://www.starc.jp/>
  - RTL設計に関するさまざまな規約，推奨記述法などを紹介
- ◆ SystemVerilogによるLSI設計(SystemVerilog for Design – A Guide to using SystemVerilog for Hardware Design and Modeling)
  - SystemVerilogの記述法



# 演習を会社や大学で行うには？

---

- ◆ 本演習は, WindowsのPCが1台あれば, 評価用のライセンスを手して, 実際の設計までの流れを実習して頂くことが可能である
- ◆ 詳しくは配布資料参照

<http://kazunoko.kuee.kyoto-u.ac.jp/~kobayasi/refresh>

# LSI設計時のシミュレーション

---

## ◆ LSI設計

- 様々なレベルでのシミュレーションが必須

## ◆ ネットリストはツールから自動的に.

- HDL, ネットリスト, 回路レベル...

## ◆ テストベンチは自分で.

- ツール、シミュレータ毎に書き方を覚えないと駄目



**ST**

単一の記述から様々なシミュレータ  
のテストベンチに変換するツール

# 記述例: 4ビット累算回路

```
#!/usr/local/bin/perl  
use ST;
```

シミュレーション対象の定義

```
target "verilog";
```

```
module "fourbitaccum";  
vcd "fourbitaccum.vcd",0;
```

波形出力

```
pin "in[3:0]", "input";
```

```
pin "CLK", "clock";
```

```
pin "RST", "input";
```

ピンの定義

```
pin "out[7:0]", "output";
```

タイミングの定義

```
timing 1e-09,1e-07,10;
```

```
clock "CLK", "1111100000";
```

```
waveform "input", "dnrz", "%.....", "in", "RST";
```

```
waveform "output", "edge", ".....%", "out";
```

```
pinorder "in", "RST", "out";
```

```
beginvector;
```

```
vector 0,0,0;
```

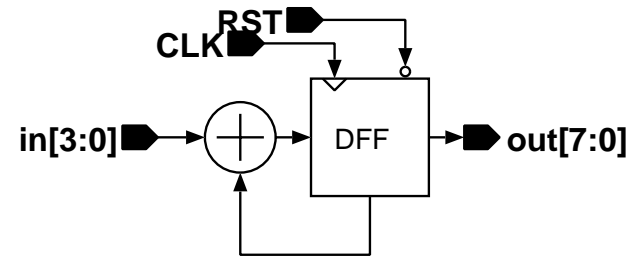
```
vector 1,1,0;
```

```
vector 5,1,1;
```

```
vector 4,1,6;
```

```
endvector;
```

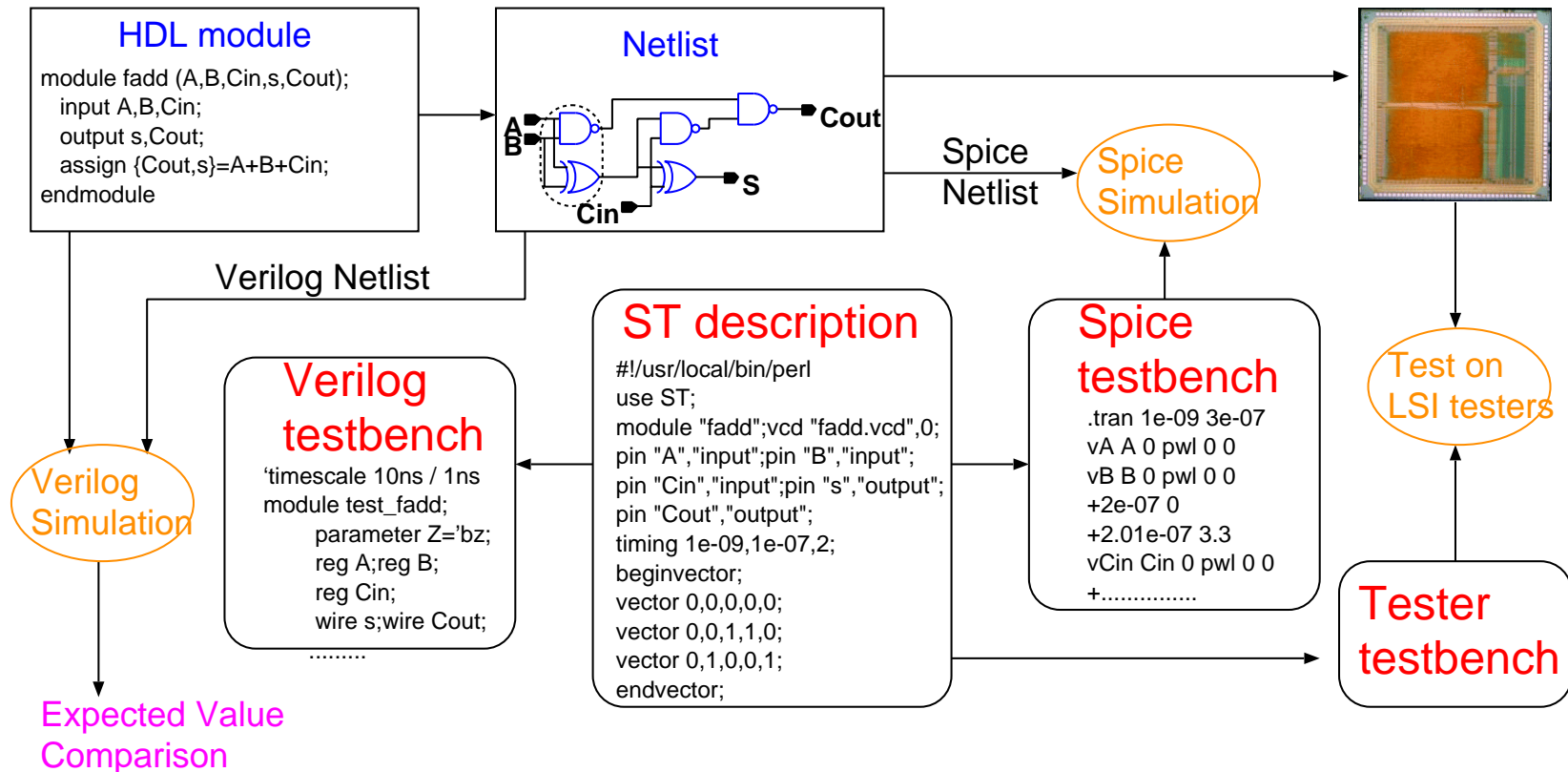
サイクル毎の  
入力ベクタ、期待値の定義



fourbitaccum

波形の定義

# STを使ったシミュレーションとテスト



ST記述を各種シミュレータ用のテストベンチに変換できる

# 期待値比較

STから出力されたテストベンチ

```
//cycle 3
in<=0;
#9          期待値
if(out == 5) ; else begin
    $display(_f,"      3(%0t): out:   5!=%0d",$realtime,out);
    _error=1;
end
```

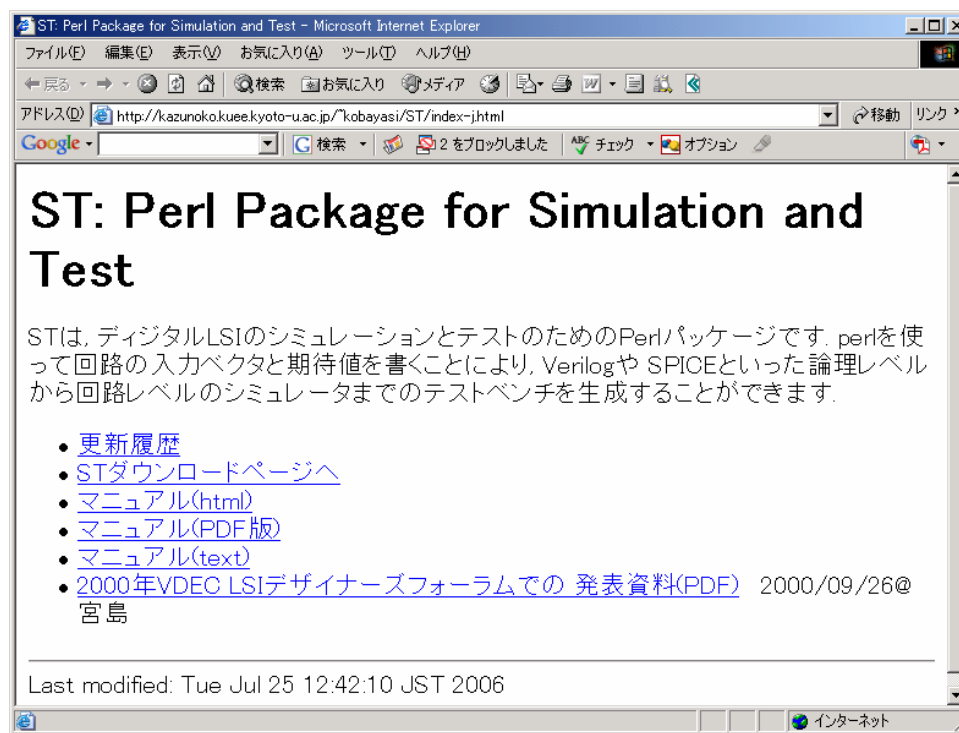
- ◆ Verilog, HSPICEそのものの機能により期待値比較
  - PLIを使わなくてよい!
  - シミュレーションだけで結果が判明
- ◆ 期待値比較が行えるシミュレータ
  - すべてのVerilogシミュレータ, HSPICE, nanosim

# 公開先

## ◆ GPLに基づき公開中

<http://kazunoko.kuee.kyoto-u.ac.jp/~kobayasi/ST/>

## ◆ 詳細なドキュメントもあり





# VDEC LSI測定用FPGAボード

---

## ◆ 目的

- 机の上でLSIをテストする
- 安価にテスト環境を供給する

## ◆ 手法

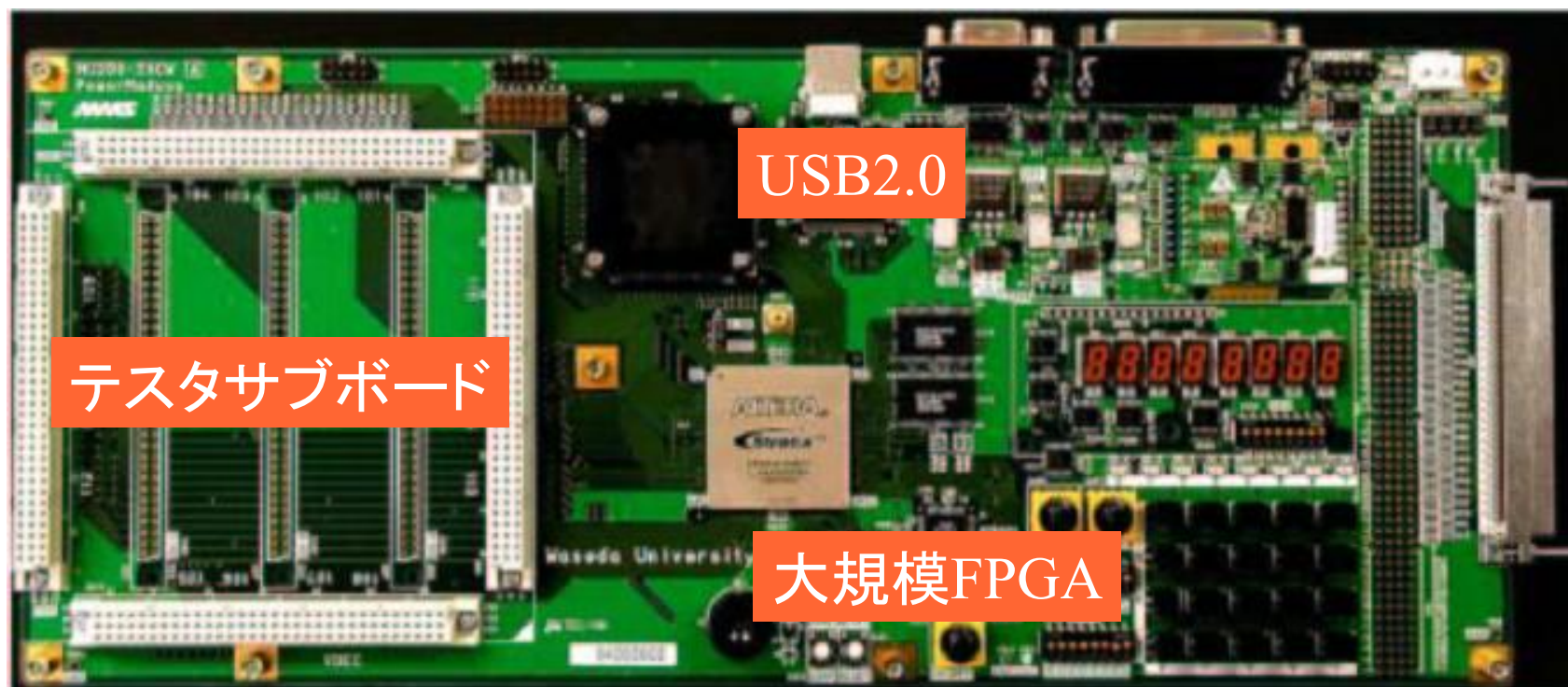
- FPGAを用いる
- LSIを測定する回路をFPGA内に構成

## ◆ STを用いて入力ベクタ作成可能



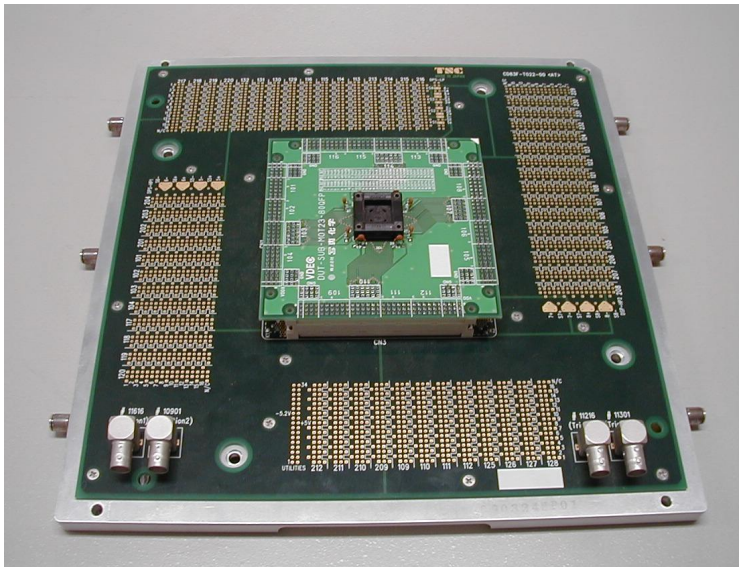
# ボード構成

- ◆ FPGA (Altera: Stratix)+テストサブボード
  - VDEC全試作チップ対応
- ◆ PCとはUSB2.0で通信
- ◆ 価格は50万より

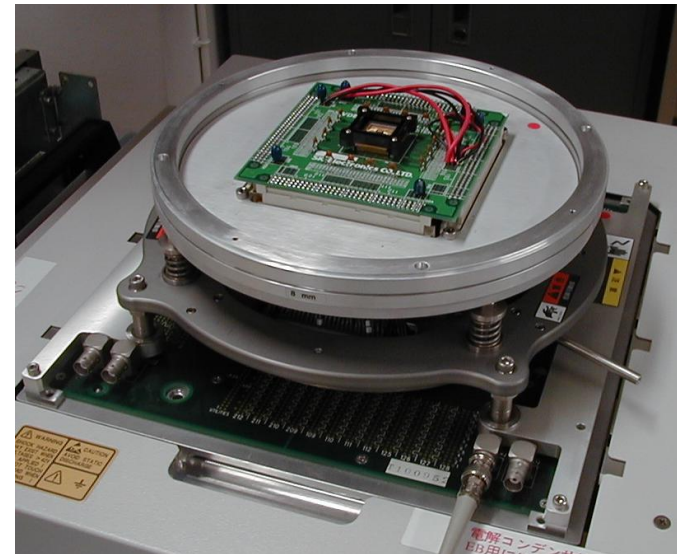


# LSIテスト用サブボード

- ◆ LSIテスト用の標準ボードを用意
  - パソコンのはんだ付けのみでテスト可能
- ◆ VDEC/サブセンターのLSIテストで使用可能



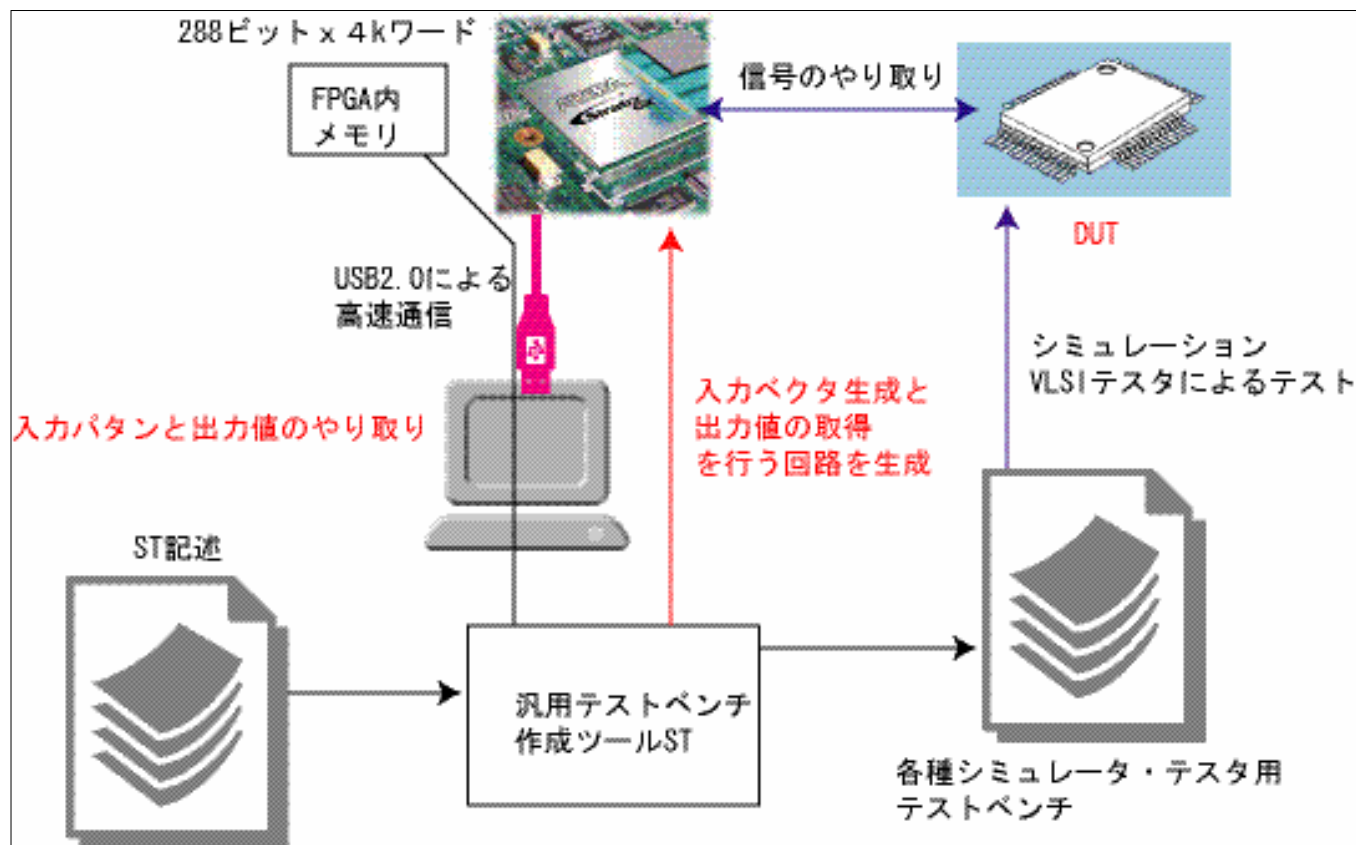
親ボードと子ボード



EBテスト用

# 汎用テストとして

- ◆ FPGA内蔵メモリに格納した入力値によりATスピードテスト可能



# 最後に

---

- ◆ 動作をよく考えて、電卓を作ってみよう。
- ◆ 市販の電卓とまったく同じ動作のものを作るのは結構大変
- ◆ がんばって回路を小さくしてみる。
- ◆ 乗除算を加えるとか、自分なりに改造する。