2015年4月17日

# ハードウエア設計論

講義内容

Verilogシミュレーション環境の設定

VerilogHDLの基本構文

代入、条件分岐、時間の表現

池田　誠

TA3名：田村、池田、松川が担当します

分からないという心の叫びは
#EEHWDesign
にてどうぞ

# 使用する環境 / 本日の課題

- Ubuntu??.?? + iVerilog + gtkwave
- **困った場合には、、ファイルは http://www.mos.t.u-tokyo.ac.jp/~ikeda/HWDesign/ にあります**
- **本日の課題**
  - **Verilogのインストールと動作確認**
  - **簡単なVerilogの入力と動作確認**
  - **結果のUPLOAD（出欠を兼ねる）→メール送信**

# Verilogのインストール

- Linux(Ubuntu)を起動してログインしターミナルを開く
  % sudo apt-get install verilog
  % sudo apt-get install nvidia-settings
  % sudo apt-get install gtkwave
- パスワードを聞かれるので、自分のパスワードを入力
- インストールの実施の有無を聞かれるので y

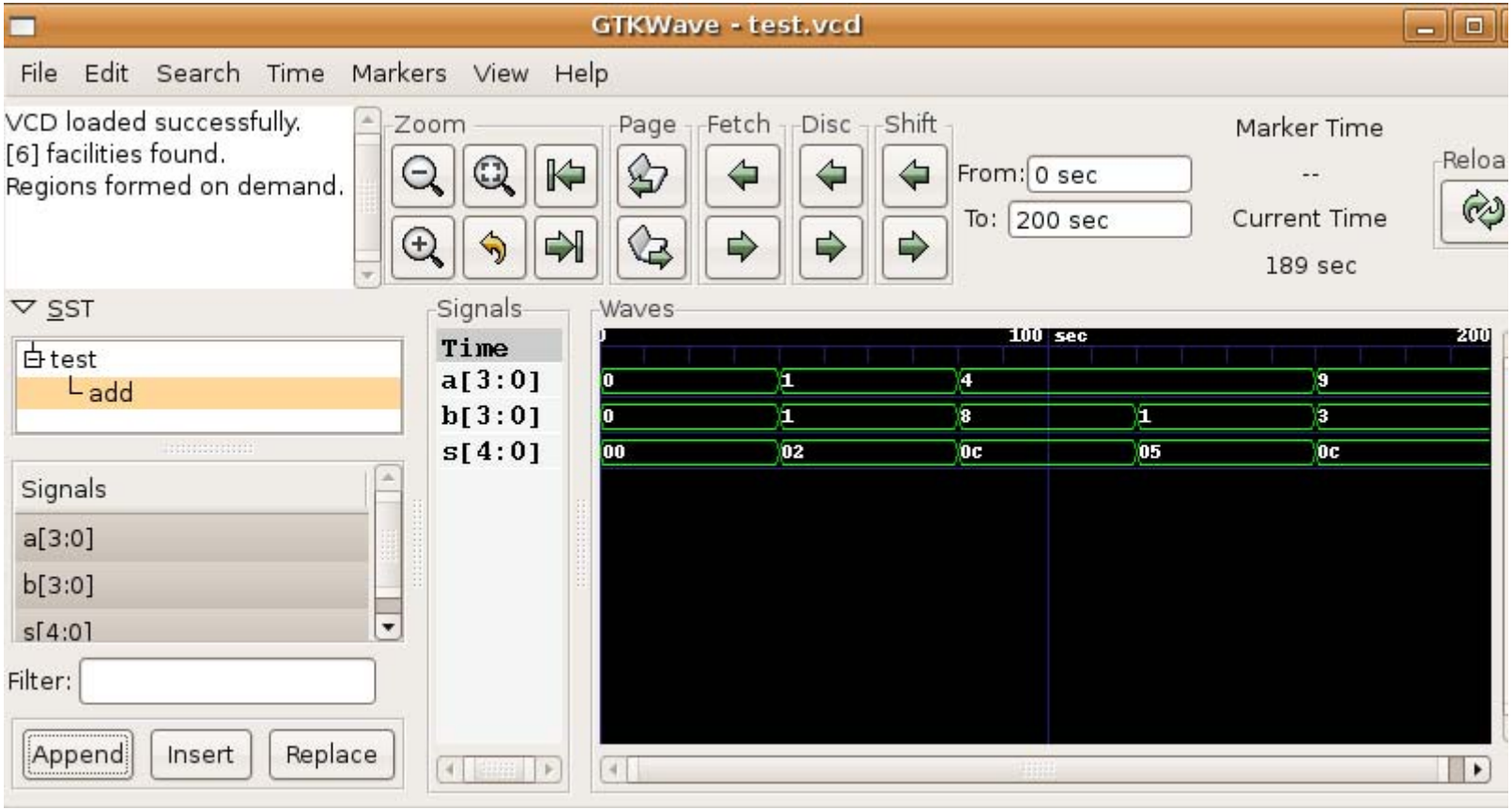
# Verilogの実行確認

- http://www.mos.t.u-tokyo.ac.jp/~ikeda/HWDesign/test.v をダウンロード
  % **iverilog　test.v**
  % **./a.out**
- 以下のように表示されればOK

```
0  0000 + 0000 = 00000  ( 0 +  0 =  0)
40  0001 + 0001 = 00010  ( 1 +  1 =  2)
 80  0100 + 1000 = 01100  ( 4 +  8 = 12)
120  0100 + 0001 = 00101  ( 4 +  1 =  5)
160  1001 + 0011 = 01100  ( 9 +  3 = 12)
200  1101 + 1101 = 11010  (13 + 13 = 26)
```

# Verilogの実行結果のGUIでの確認

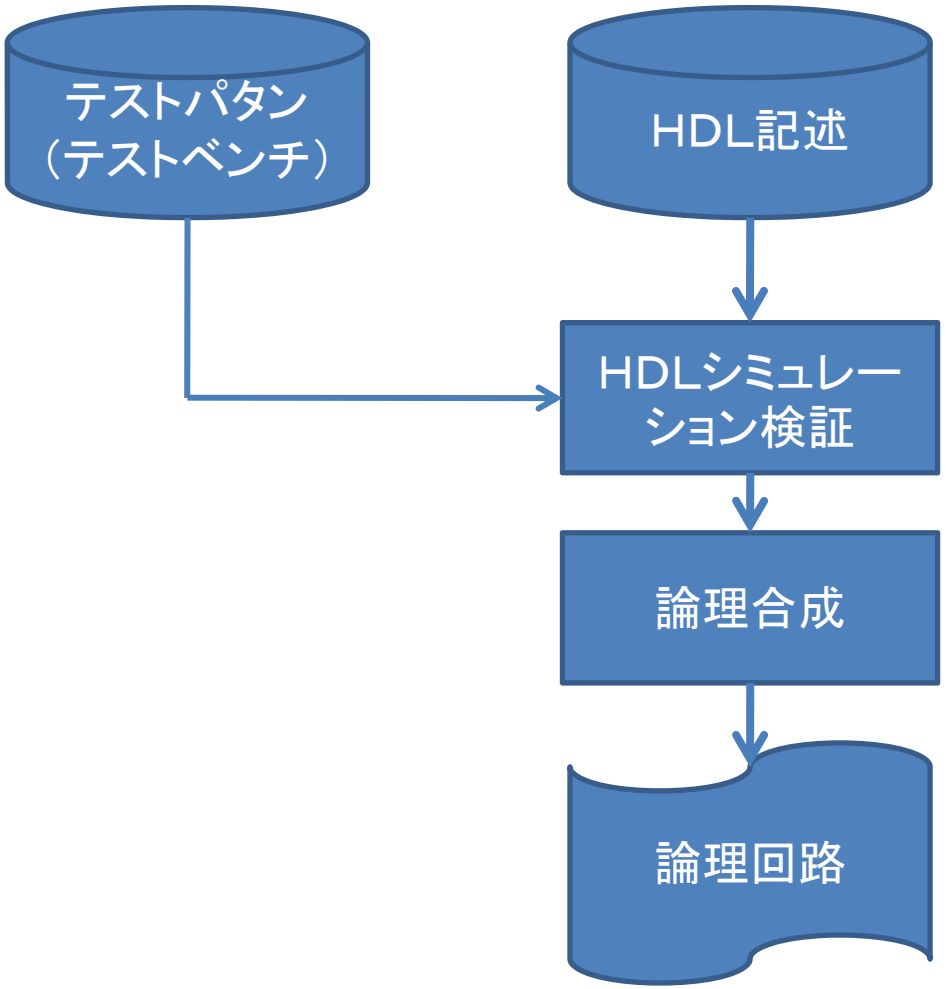
- % **gtkwave　test.vcd**



# ハードウエア記述言語

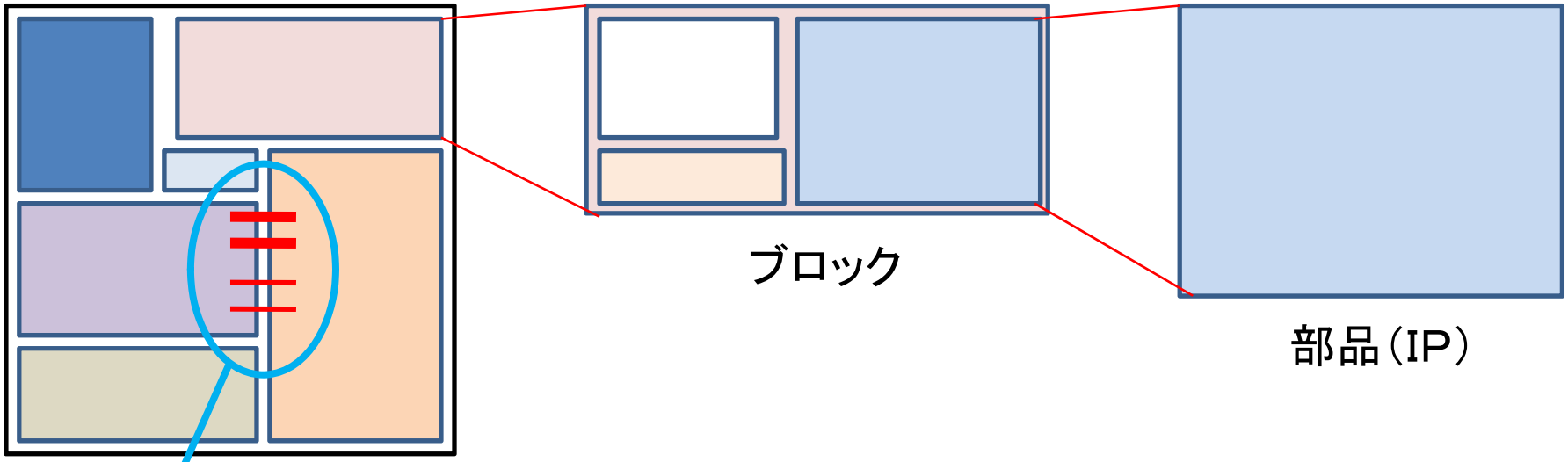
| 言語 | VHDL | VerilogHDL | UDL／I | SFL |
|---|---|---|---|---|
| 開発開始時期 | 1981 | 1984 | 1987 | 1981 |
| 開発組織 | IEEE | Cadence | 電子協 | NTT |
| 言語仕様公表 | 1987 | 1985 | 1990 | 1985 |
| 論理シミュレータ | 有 | 有 | 有 | 有 |
| 論理合成系 | 有 | 有 | 有 | 有 |
| 規格の見直し | 1993,2000, 2002, 2008 | 1995,2001, 2005 | 1992 | なし |

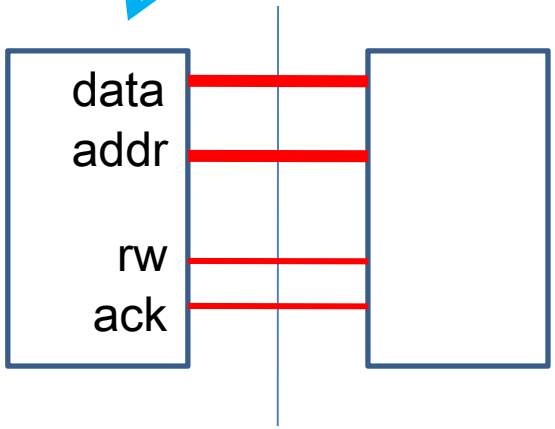
# HDLによる設計

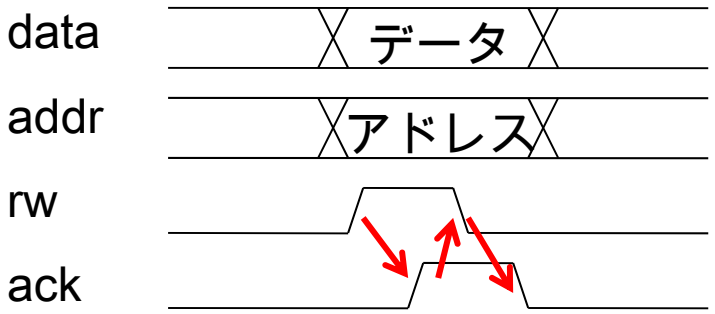



- 抽象度の高い記述による記述量の削減と設計効率向上
- ゲートレベル設計の自動化
- 設計の早期からのシミュレーションによる検証
- 制御回路の自動生成

# 階層設計とインターフェース




インターフェースとタイミング

# VerilogHDLの基本構文

**module　*モジュール名*　（ *ポート名*, *ポート名*, ・・・　）;**
**モジュールの入出力の宣言（全ポート名を宣言する）;**
**モジュール内信号の宣言（暗黙の定義は出来るだけ避ける）;**
**回路・機能の定義 ;**
**endmodule**

**全ての記述は module ～ endmoduleで囲う。**
**構文は セミコロン ; により閉じる**
**複数構文にまたがる場合には begin ～ endで囲う。**
**入出力の定義は、 入力　input, 出力 output, 双方向 inoutにより定義**
**例 :**

```
module   test ( inA, inB, outC );
  input  inA, inB;
  output  outC;
endmodule
```

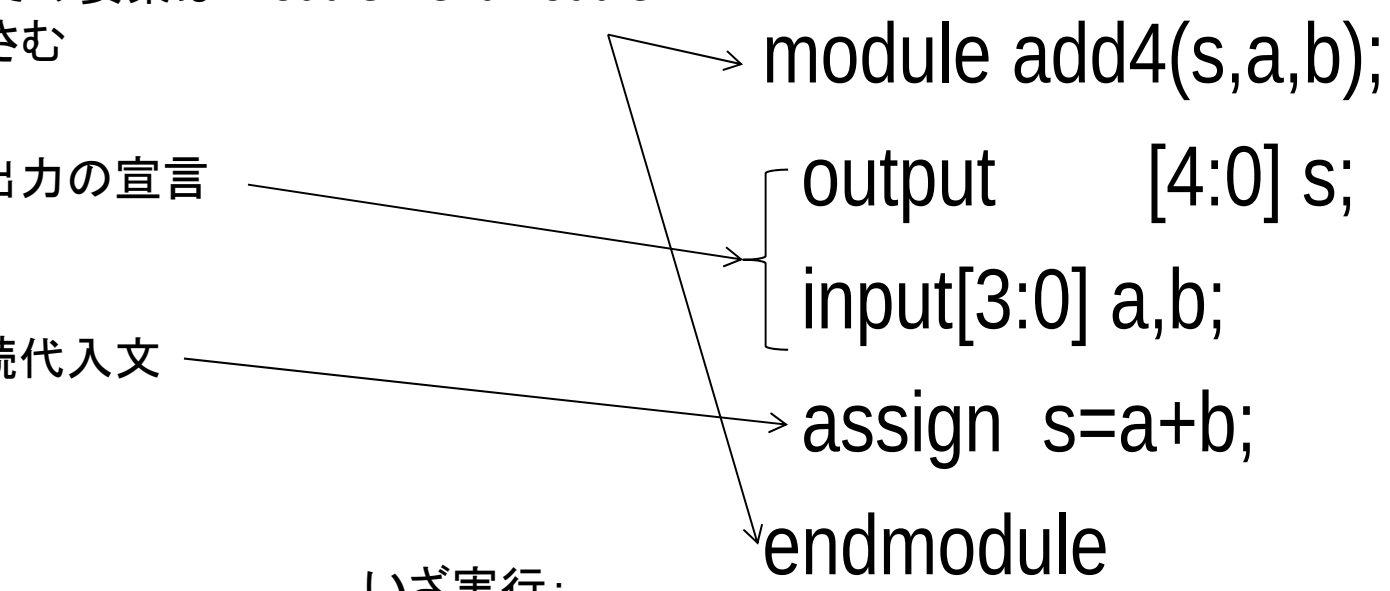
# VerilogHDLの基本構文

add4.v

すべての要素は module – endmodule
ではさむ

入出力の宣言

継続代入文




いざ実行:

% **iverilog add4.v**

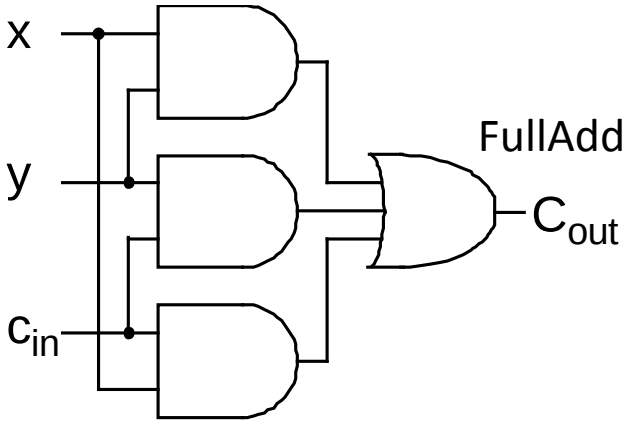
% **./a.out**

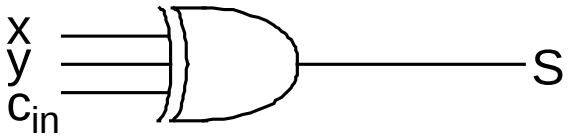
？？？

テストベンチが無い
（＝入出力が与えられて
いないので何も動かない）

| 名称 | 記号 | 定義 | 優先順位 | 名称 | 記号 | 定義 | 優先順位 |
|---|---|---|---|---|---|---|---|
| 算術演算 | + | 加算 | 3 | 論理演算 | ! | 論理値のNOT | 1 |
| | - | 減算 | 3 | | && | 論理値のAND | 9 |
| | * | 乗算 | 2 | | \|\| | 論理値のOR | 10 |
| | / | 除算 | 2 | 等号演算 | == | 論理等号 | 6 |
| | % | 剰余 | 2 | | != | 論理不等号 | 6 |
| ビット演算 | ~ | ビット毎の反転 | 1 | | === | ケース等号(X,Zも一致) | 6 |
| | & | ビット毎のAND | 7 | | !== | ケース不等号(X,Zも不一致) | 6 |
| | \| | ビット毎のOR | 8 | 関係演算 | < | 小なり | 5 |
| | ^ | ビット毎のExOR | 7 | | <= | 小なりイコール | 5 |
| | ~^ | ビット毎のExNOR | 7 | | > | 大なり | 5 |
| リダクション演算（単項演算） | & | 各桁ビットのAND | 1 | | => | 大なりイコール | 5 |
| | ~& | 各桁ビットのNAND | 1 | シフト演算 | << | 右オペランド分左シフト（空いたビットは0） | 4 |
| | \| | 各桁ビットのOR | 1 | | >> | 右オペランド分右シフト（空いたビットは0） | 4 |
| | ~\| | 各桁ビットのNOR | 1 | 条件演算 | ? : | 条件 ? 真の場合 : 偽の場合 | 11 |
| | ^ | 各桁ビットのExOR | 1 | | | | |
| | ~^ | 各桁ビットのExNOR | 1 | | | | |

# 簡単な論理式を実現してみよう




**左図の全加算器を実現してみる:**

FullAdderFunction.v

```
module  FullAdderFunction ( x, y, cin, cout, s );
input    x, y, cin;
output   cout, s;

assign   cout  = (x & y ) | ( x & cin )  | (y & cin ) ;
assign   s = x ^ y ^ cin ;
endmodule
```

# Verilogで定義される論理値と定数

**取りうる値：**

0：　　Low (論理 0)

1：　　High（論理　1）

x：　　不定値：0か1か不定であるがどちらかの値を取る

z：　　High Impedance：0でも1でもない（定義されない値）

**定数：**

<ビット幅>'<基数><数値>として表す

<基数>：　b, B: 2進数、o,O: 8進数, d,D:  10進数, h,H: 16進数

（例）

| 表記 | 基数 | ビット幅 | 二進数表記 |
|---|---|---|---|
| 8 | 10 | 32bit | 0000…..01000 |
| 4'd5 | 10 | 4bit | 0101 |
| 1'b0 | 2 | 1bit | 0 |
| 16'h 0f0f | 16 | 16bit | 0000111100001111 |
| 4'bx | 2 | 4bit | xxxx |

# Verilogで定義される型

**ネット型**:　　配線を表す。信号の論理値は接続されるノードの値として決定される

→　単なる配線であるため、何らかの演算結果が「接続」されているだけであり、代入操作としては接続、つまり　assign 文のみが使用可能

**レジスタ型**:　　レジスタ（記憶素子：いわゆるプログラミング的な変数）。信号の論理値が保持される

→　レベルを保持するラッチやフリップフロップに相当。always文, initial文, function, taskの中での手続き代入操作のみが可能。(assignは出来ない)

# 定義可能な型の種類

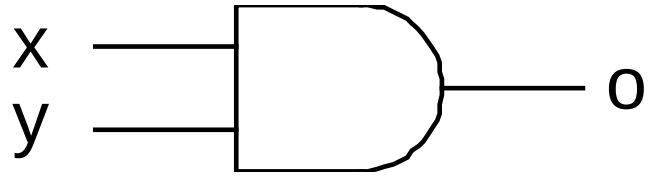
- wire型
  - 継続的代入されているときのみ値を保持する型。一般の配線と同様。通常は組み合わせ論理部を表現するのに使用
  - 1bitのwire型は定義を省略可能：ただしこの暗黙の定義は使わない方が賢明
- reg型
  - 任意ビット、記憶保持が可能な型（通常はFFなど状態、データを保存したいノードに対して使用）, signedを指定しない場合には符号なし
- signed指定
  - reg signed [7:0] a;　　aを符号付きレジスタ型として定義
  - wire signed [7:0] a;　aを符号付きwire型として定義
- integer型
  - 32bit幅の符号付き整数型
- real型(実数), time型(符号無し64ビット), realtime型(実数表記での時間)
- バス幅の定義
  - reg [7:0] aなど。a[0] からa[7]の8ビット幅として定義。降順
- アレイの定義
  - reg a[0:31]など。0から31番地までを確保。昇順

# VerilogHDLの基本構文：構造記述

回路の定義:
他のモジュールを呼び出す記述：構造記述
（ちょうど回路図を書いているようなもの）

モジュール名　　インスタンス名　（ポート）
モジュール名：呼び出すモジュールの名前
インスタンス名：任意（呼び出しの名前）:変数や他のインスタンス名と重複してはならない
ポート:
　定義順呼び出し:
　　ポート定義順に信号を記述
　名前呼び出し:　module (a, b, c); の場合
　　.a(sigA), .b(sigB), .c(sigC) と記述することで記述順は関係なくなる

```
module and2 (x, y, o);
  input  x, y;
  output   o;
  and   a1 ( o, x, y );
endmodule
```

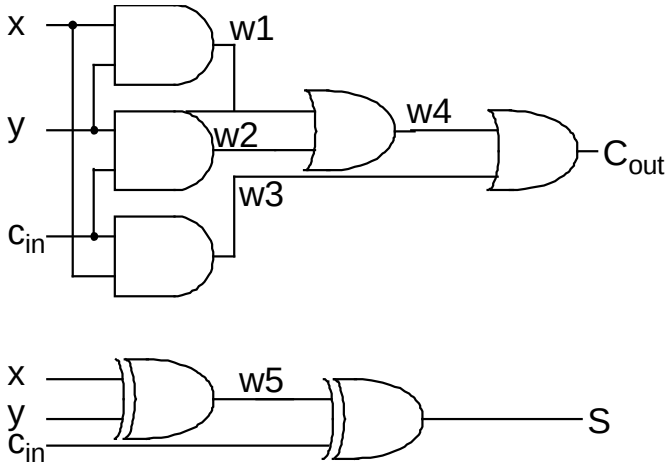


# Verilogで定義されるプリミティブゲート

Verilog-HDLにあらかじめ組み込まれているゲートで、module定義をすることなく使用することが可能。通常、ポートは 出力、入力、イネーブルの順となっている

| 種別 | ゲート名 | 出力 | 入力 | イネーブル | 機能 |
|---|---|---|---|---|---|
| 1入力ゲート | buf, not | OUT | IN | | |
| 2入力ゲート | and, nand, nor, or, xor, xnor | OUT | IN1, IN2 | | |
| 3state | bufif0, bufif1, notif0, notif1 | OUT | DATA | CONTROL | buf: バッファ, not: 論理反転,<br>if0: control=0で出力,<br>if1: CONTROL=1で出力 |
| switch | nmos, pmos, cmos, rnmos, rpmos, rcmos, tran, tranif0, tranif1, rtran, rtranif0, rtranif1 | | | | |
| pullup, pulldown | pullup, pulldown | | | | |

# 簡単な回路図を実現してみよう




左図の全加算器を構造記述により実現してみる:

FullAdderStructure.v

```
module  FullAdderStructure ( x, y, cin, cout, s );
input    x, y, cin;
output   cout, s;
wire   w1, w2, w3, w4, w5;

and   a1 ( w1, x, y );
and   a2 ( w2, y, cin );
and   a3 ( w3, cin, x );
or     o1 ( w4, w1, w2 );
or     o2 ( cout, w4, w3 );
xor    x1 ( w5, x, y );
xor    x2 ( s, w5, cin );
endmodule
```

# 基本構文：手続きと順序機械

4ビットカウンタ

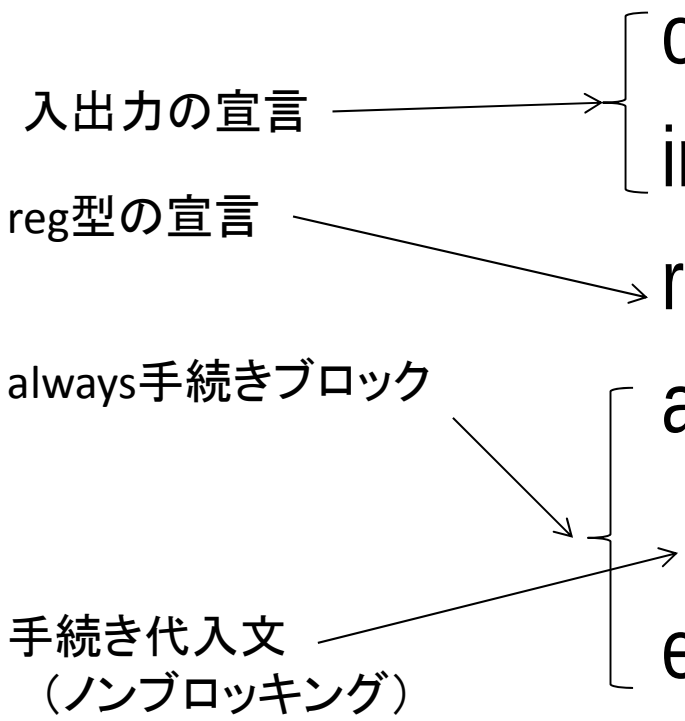
count4.v

入出力の宣言

reg型の宣言

always手続きブロック

手続き代入文
（ノンブロッキング）



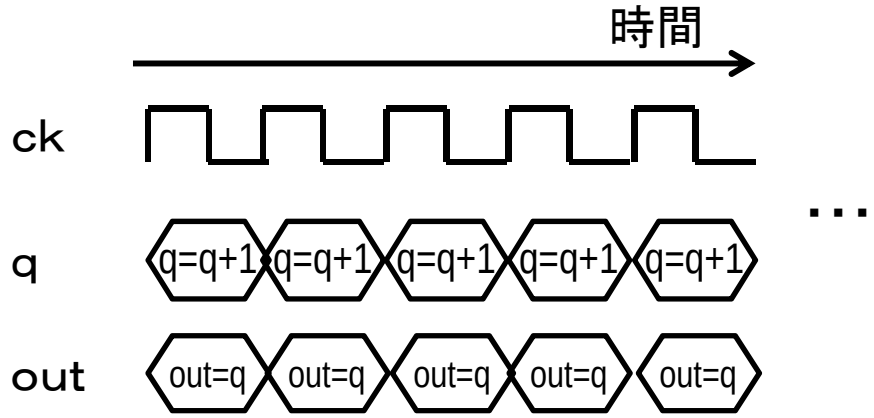
```
module  count4(out,ck);
  output      [3:0] out;
  input       ck;
  reg         [3:0] q;
  always @(posedge ck) begin
    q <= q+1;
  end
  assign out = q;
endmodule
```

# Verilogと時間・・

```
module  count4(out,ck);
  output    [3:0] out;
  input           ck;
  reg             [3:0] q;
  always @(posedge ck) begin
     q <= q+1;
  end
  assign out = q;
endmodule
```

always構文は@内の条件が満たされるたびに実行される

継続的に代入が実施される（つまりqが変化するごとにoutが変化する



# VerilogHDLの基本構文：プロセス文

1. always(イベント)ブロック
   イベントが発生した場合にブロック内を1回実行、
   次のイベントが発生するまで待つ
   1. always  #10 ck<=~ck;
      1. クロックの定義
   2. always @(posedge ck)
      1. 順序機械の定義: エッジセンシティブ
         posedge: 立ち上がりエッジ、negedge: 立下りエッジ
   3. always (reset)
      1. 組み合わせ回路の定義：レベルセンシティブ
2. Initial (イベント)ブロック
   1. ブロック内を1回実行し終了

# VerilogHDLの基本構文：手続き文

- 条件分岐
  - if文
    if 条件式
        文
    else
        文
  - case文：casez: zも条件に入れられる、casex: x, zも条件に入れられる
    case 条件式
      値1: 文
      値2: 文
      default: 文
    endcase
- ループ
  - for：「文1」を実行し、「条件式」が真の場合、「文２」、「文３」を実行、その後「条件式」が真の間、「文２」、「文３￥を繰り返す
    for( 文1 ; 条件式 ; 文3 )
        文2
  - while：「条件式」が真の間「文」を繰り返し実行する
    while( 条件式 )
        文
  - forever：繰り返し「文」を実行する。ループから脱出には disable文を用いる
    forever
        文
  - wait：「条件式」が真になるまで実行をストップする
    wait(条件式)

# 順序機械の構成の原則

always文を使用する場合、原則としてリセット（初期化）を必ず実装しておく

同期リセット：クロックに同期してリセット実行

```
always ( @(posedge ck) ) begin
   if( rst ) リセットの実行
   else
                順序機械の記述
end
```

非同期リセット：リセット信号入力で直ちにリセット実行

```
always ( @(posedge ck) or rst ) begin
   if( rst ) リセットの実行
   else
                順序機械の記述
end
```

count4r.v

# VerilogHDLの基本構文

全ての構文は
module / endmodule
で囲む

全ての構文は
セミコロンで区
切る

構文が複数行になる場
合には
begin / end
で囲う

if else 構文

```
module  count4r(out,ck,res);
  output        [3:0] out;
  input         ck, res;
  reg           [3:0] q;
  always @(posedge ck or negedge res) begin
    if( !res)   q<= 0;
    else        q <= q+1;
  end
  assign out = q;
endmodule
```

res=0のとき qに0を代入
（リセット）

# テストベンチを作成

- シミュレーションを行うためには設計回路に入力を与えるためのプログラムが必要
  - テストベンチ
- 同じverilogで記述する

# テストベンチ

testadd41.v

```
module testadd4;
  wire              [4:0] s;
  reg               [3:0] a, b;
  initial begin
    $monitor( “%t %b + %b = %b”, $time, a, b, s);
     a <= 0;   b <= 0;
     #40    a <= 1;   b <= 3;
     #40    a <= 4;   b <= 8;
     #40    a <= $random;   b <= $random;
     #40    a <= $random;   b <= $random;
     #40
     $finish;
  end
  add4    add ( s,a,b );
endmodule
```

システムタスク:
$monitor: 変化ごとに表示

initial手続きブロック

システムタスク:
$finish シミュレーション終了

被検証用モジュールの
呼び出し（インスタンス化宣言）

インスタンス名
（任意）

モジュール名

入出力信号名

# テストベンチ

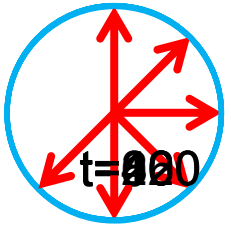
再度実行:

% **iverilog testadd41.v add4.v**

% **./a.out**

```
  0 0000 + 0000 = 00000
 40 0001 + 0011 = 00100
 80 0100 + 1000 = 01100
120 0100 + 0001 = 00101
160 1001 + 0011 = 01100
```

# Verilogと時間




```
module add4(s,a,b);
  output    [4:0] s;
  input     [3:0] a,b;
  assign  s=a+b;
endmodule
```

initial構文はプログラム言語同様に逐次実行

時間経過を表示：表示がない場合には0(δ時間)

```
module testadd4;
  wire                [4:0] s;
  reg          [3:0] a, b;
  initial begin
      $monitor( "%t %b + %b = %b", $time, a, b, s);
      a <= 0;   b <= 0;
      #40       a <= 1;   b <= 3;
      #40       a <= 4;   b <= 8;
      #40       a <= $random;   b <= $random;
      #40       a <= $random;   b <= $random;
      #40
      $finish;
  end
  add4    add ( s,a,b );
endmodule
```

```
  0 0000 + 0000 = 00000
 40 0001 + 0011 = 00100
 80 0100 + 1000 = 01100
120 0100 + 0001 = 00101
160 1001 + 0011 = 01100
```

# テストベンチ

testcount4.v

```
module testcount4;
  wire        [3:0] out;
  reg         ck;
  initial begin
    $monitor( "%t %b %b", $time, ck,  out);
    ck<=0;
    #350
    $finish;
  end
  always #10  ck <= ~ck;
  count4  cnt ( out, ck);
endmodule
```

システムタスク:
$monitor: 変化ごとに表示

initial手続きブロック

システムタスク:
$finish シミュレーション終了

initial構文はプログラム言語同様に逐次実行

```
%  iverilog  testcount4.v  count4.v
%  ./a.out
```

```
 0 0 xxxx
10 1 xxxx
20 0 xxxx
30 1 xxxx
```

？？？？？？
リセットしなくてはレジスタ型変数の値が不定

always構文は条件が満たされるたびに実行される(つまり10単位時間ごとにckが~ck (ckの論理反転)になる
→20周期のクロックが生成される

テスト対象のモジュールを呼び出す継続的代入同様に常に実行(変化が即時に伝搬する

# テストベンチ

testcount4r.v

```
module testcount4r;
  wire              [3:0] out;
  reg               ck, res;
  initial begin
    $monitor( “%t %b %b %b”, $time, ck, res, out);
    ck<=0;
    res<=0;
    #40
    res <= 1;
    #350
    $finish;
  end
  always #10  ck <= ~ck;
  count4r  cnt ( out, ck, res );
endmodule
```

システムタスク:
$monitor: 変化ごとに表示

initial手続きブロック

システムタスク:
$finish シミュレーション終了

被検証用モジュールの
呼び出し（インスタンス化宣言）

モジュール名

入出力信号名

インスタンス名
（任意）

# いざ実行

**% iverilog testcount4r.v count4r.v**
**% ./a.out**

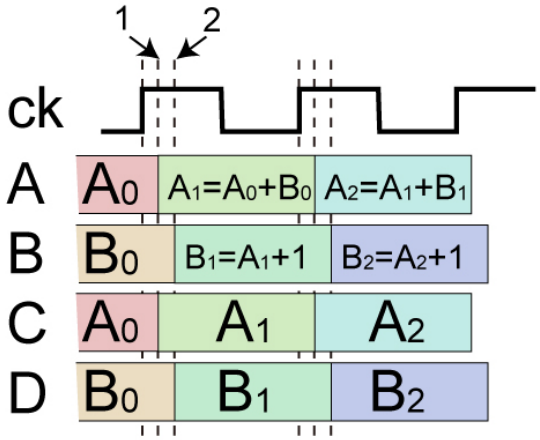
```
  0 0 0 0000
 10 1 0 0000
 20 0 0 0000
 30 1 0 0000
 40 0 1 0000
 50 1 1 0001
 60 0 1 0001
 70 1 1 0010
 80 0 1 0010
 90 1 1 0011
100 0 1 0011
110 1 1 0100

370 1 1 0001
380 0 1 0001
```

# VerilogHDLの基本構文：代入

- 代入の時間関係

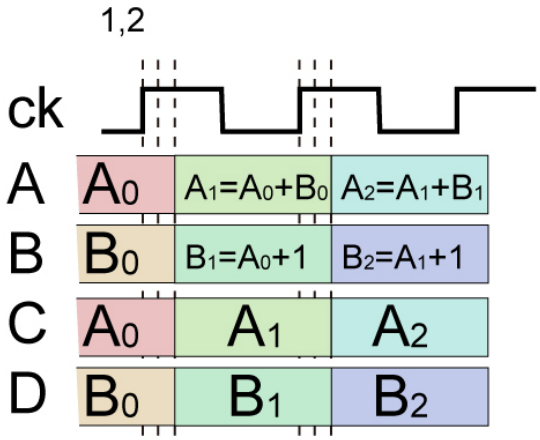
ブロッキング代入

```
assign C=A;
assign D=B;
always @posedge ck
1    A=A+B;
2    B=A+1;
end
```



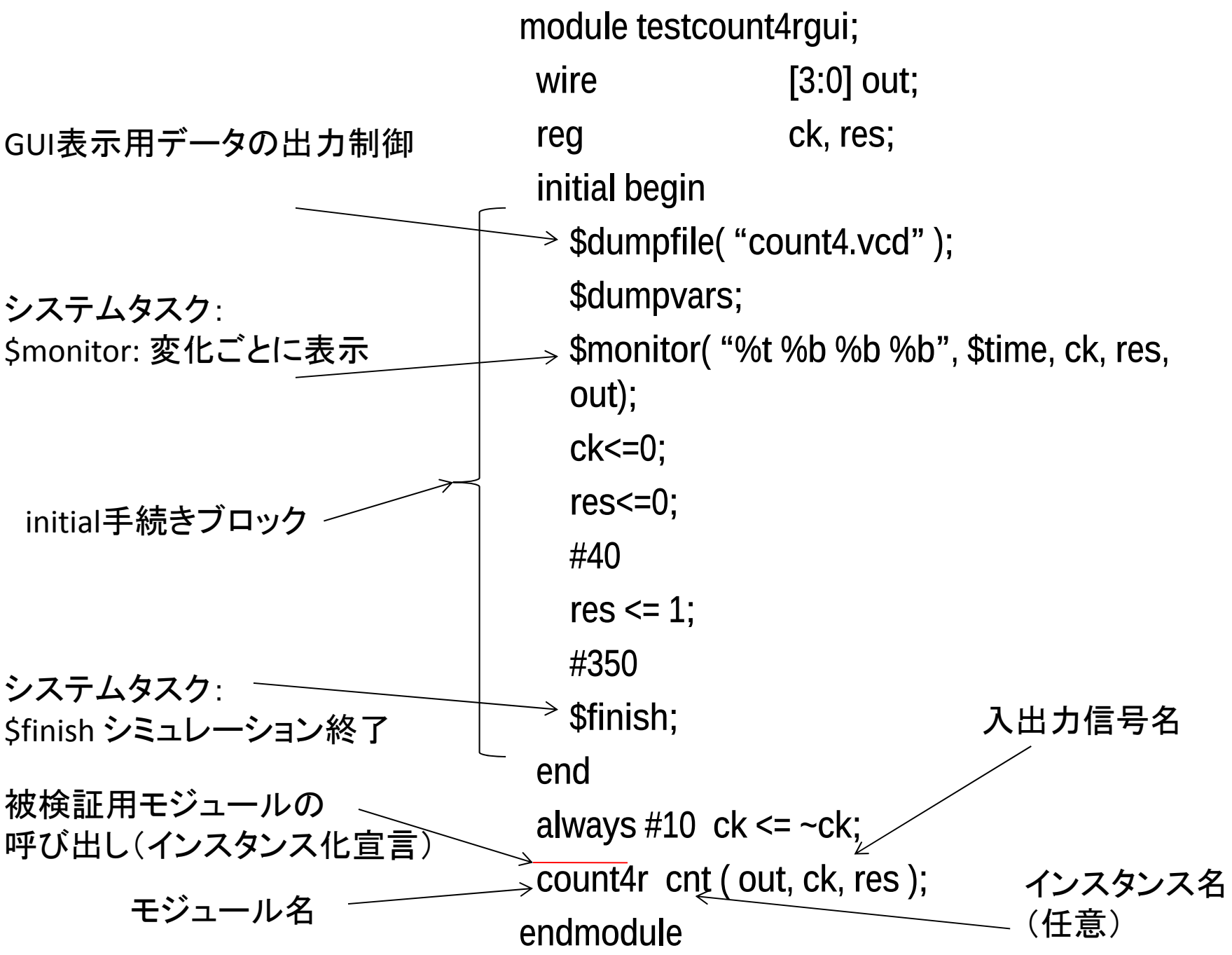

ノンブロッキング代入

```
assign C=A;
assign D=B;
always @posedge ck
    A<=A+B;
    B<=A+1;
end
```




ハードウエアの記述では必ずノンブロッキング代入を使用する

テストベンチの記述にはノンブロッキング代入を用いてもよい


module testcount4rgui;
wire [3:0] out;
reg ck, res;
initial begin
$dumpfile( “count4.vcd” );
$dumpvars;
$monitor( “%t %b %b %b”, $time, ck, res, out);
ck<=0;
res<=0;
#40
res <= 1;
#350
$finish;
end
always #10 ck <= ~ck;
count4r cnt ( out, ck, res );
endmodule
GUI表示用データの出力制御
システムタスク:
$monitor: 変化ごとに表示
initial手続きブロック
システムタスク:
$finish シミュレーション終了
被検証用モジュールの
呼び出し（インスタンス化宣言）
モジュール名
入出力信号名
インスタンス名
（任意）

# 演習３

- 論理式バージョン
  module  FullAdderFunction  ()
  を完成させ、simfulladd.vを作成して、シミュレーション結果を確認する
- 構造記述バージョン
  module  FullAdderStructure ()
  を完成させ、simfulladd.vを作成して、シミュレーション結果を確認する

# 本日の出欠は・・・

本日正午までに
演習3(テストベンチも含む)と実行結果の画面のコピーを

**HWDesign2015@vdec.u-tokyo.ac.jp**

までメールをしてください。
メールは題名に20150417_P
本文冒頭に
#StudentID: 学生証番号(03-540から明記すること)
#Name: 氏名(姓_名): 例 Ikeda_Makoto
を明記すること。氏名を含めすべて半角英数字のみにすること。機械的に処理をするのでこのフォーマットを守らない場合には出席にはならないことを注意すること。なお、出席登録可否のメールが返送されるので確認しておくこと。後日の登録漏れ申告は原則認めない。出欠一覧は、Webからリアルタイムで確認可能(必ず確認しておくこと)。なお、姓、名とも1文字目のみ大文字、あとは小文字とすること。HTML形式や、base64 encodedは認識しないので必ず平文、JISにて送信すること

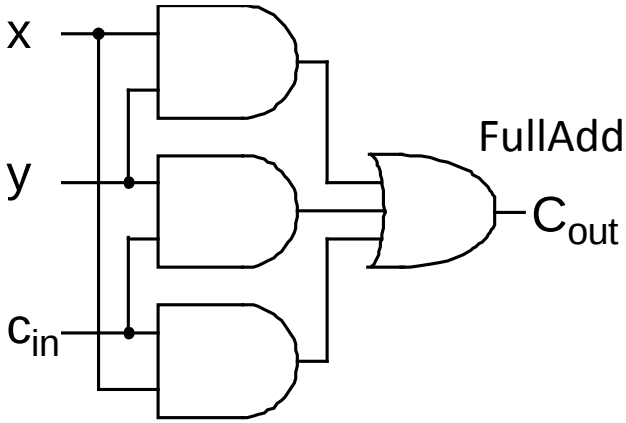
# ハードウエアにおける設計表現
# ハードウエア設計記述言語VerilogHDL
# ～状態遷移と順序機械～

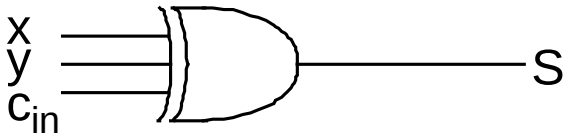
**Ubuntuを起動し**
**verilog**が実行できる状態にしておいてください。

# 本日の出欠は・・・

本日正午＋δまでに
**WEB**から課題１～４までを提出する。それまでに終わらない場合には、来週までに課題４まで終えてWEBにUPLOADしておくこと。

# 簡単な論理式を実現してみよう



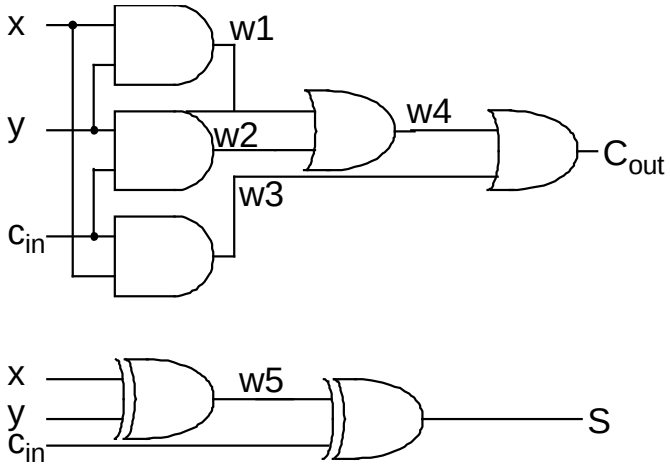

左図の全加算器を実現してみる:

FullAdderFunction.v

```
module  FullAdderFunction ( x, y, cin, cout, s );
input    x, y, cin;
output   cout, s;

assign   cout  = (x & y ) | ( x & cin )  | (y & cin ) ;
assign   s = x ^ y ^ cin ;
endmodule
```

## 課題３－１　FullAdderFunction.v

# 簡単な回路図を実現してみよう




左図の全加算器を構造記述により実現してみる：

FullAdderStructure.v

```
module  FullAdderStructure ( x, y, cin, cout, s );
input    x, y, cin;
output   cout, s;
wire   w1, w2, w3, w4, w5;

and   a1 ( w1, x, y );
and   a2 ( w2, y, cin );
and   a3 ( w3, cin, x );
or     o1 ( w4, w1, w2 );
or     o2 ( cout, w4, w3 );
xor    x1 ( w5, x, y );
xor    x2 ( s, w5, cin );
endmodule
```

課題３－２　FullAdderStructure.v

# テストベンチ・・1

simfulladd.v

```
module simfulladd;
  wire                  s, cout;
  reg                   x, y, cin;
  initial begin
      $monitor( “%t Input (x, y, cin) -> Output (s, cout):
      (%b, %b, %b) -> (%b, %b)”, $time, x, y, cin, s, cout);
      x <= 0;   y <= 0; cin<=0;
      #40       x <= 1;   y <= 0; cin<=0;
      #40       x <= 0;   y <= 1; cin<=0;
      #40       x <= 1;   y <= 1; cin<=0;
      #40       x <= 0;   y <= 0; cin<=1;
      #40       x <= 1;   y <= 0; cin<=1;
      #40       x <= 0;   y <= 1; cin<=1;
      #40       x <= 1;   y <= 1; cin<=1;
      #40
      $finish;
  end
FullAdderFunction  add ( x, y, cin, cout, s );
endmodule
```

FullAdderStructure add ( x, y, cin, cout, s );

simfulladd

```
 0 Input (x, y, cin) -> Output (s, cout): (0, 0, 0) -> (0, 0)
40 Input (x, y, cin) -> Output (s, cout): (1, 0, 0) -> (1, 0)
80 Input (x, y, cin) -> Output (s, cout): (0, 1, 0) -> (1, 0)
120 Input (x, y, cin) -> Output (s, cout): (1, 1, 0) -> (0, 1)
160 Input (x, y, cin) -> Output (s, cout): (0, 0, 1) -> (1, 0)
200 Input (x, y, cin) -> Output (s, cout): (1, 0, 1) -> (0, 1)
240 Input (x, y, cin) -> Output (s, cout): (0, 1, 1) -> (0, 1)
280 Input (x, y, cin) -> Output (s, cout): (1, 1, 1) -> (1, 1)
L15 "simfulladd.v": $finish at simulation time 320
```

# テストベンチ・・2

simfulladd2.v

```
module simfulladd;
  wire                    s, cout;
  reg          x, y, cin, ck, flag;
  initial begin
       $monitor( “%t Input (x, y, cin) -> Output (s, cout): (%b, %b, %b) -> (%b, %b)”, $time, x, y, cin, s,
       cout);
       x <= 0;   y <= 0; cin<=0;  ck <= 0; flag <= 0;
  end
  always  #10 ck <= ~ck;
  always @(negedge ck) begin
       if( s != (x ^ y ^ cin) || cout != (x & y | x & cin | y & cin) ) begin
         flag <= 1;
         $finish;
       end
       if( {cin,x,y}  == 3’b 111 ) begin
         $display( “OK¥n” );
         $finish;
       end
  end
  always @(posedge ck) begin
       {cin,x,y} <= {cin,x,y} + 1;
  end
  FullAdderFunction add ( x, y, cin, cout, s );
endmodule
```

FullAdderStructure add ( x, y, cin, cout, s );

simfulladd

```
  0 Input (x, y, cin) -> Output (s, cout): (0, 0, 0) -> (0, 0)
 10 Input (x, y, cin) -> Output (s, cout): (0, 1, 0) -> (1, 0)
 30 Input (x, y, cin) -> Output (s, cout): (1, 0, 0) -> (1, 0)
 50 Input (x, y, cin) -> Output (s, cout): (1, 1, 0) -> (0, 1)
 70 Input (x, y, cin) -> Output (s, cout): (0, 0, 1) -> (1, 0)
 90 Input (x, y, cin) -> Output (s, cout): (0, 1, 1) -> (0, 1)
110 Input (x, y, cin) -> Output (s, cout): (1, 0, 1) -> (0, 1)
130 Input (x, y, cin) -> Output (s, cout): (1, 1, 1) -> (1, 1)
OK
```

# VerilogHDLの基本構文：関数・タスク

- function：戻り値は関数名を左辺に指定した代入
  function ビット幅 関数名;
  　input ビット幅 変数名;
  　宣言
  　シーケンシャル文
  endfunction

  関数呼び出しは
  a <= 関数名(引数)
- task：戻り値はoutputもしくはinout変数として宣言
  task タスク名;
  　input ビット幅 変数名;
  　output ビット幅 変数名;
  　宣言
  　シーケンシャル文
  endtask

  タスク呼び出しは
  タスク名(引数1, 引数2・・・)

# その他・・1

- 連接
  - {式1, 式2}: 式1, 式2をつなげる {0101, 1100} → 01011100
  - {定数式 {式, 式} }: {}内を定数式の値だけ繰り返す { 5, {10} } → 1010101010
- レンジ式
  - [定数1:定数2] : a[6:3] → a[6], a[5], a[4], a[3]
  - [式+:定数] : a[P*8+:4] → P=0の時a[3:0], P=1の時 a[11:8]
  - [式-:定数] : a[P*8-:4] → P=1の時 a[8:5], P=2の時 a[16:13]

# 演習１

- add4.v, testadd41.vをダウンロードして実行
- add4.vを修正し減算をするsub4.v, 乗算をするmul4.vを作成し、それらに対応するテストベンチ testsub41.v, testmul41.vを作成して実行し結果を確認

**課題１－１　sub4.v**
**課題１－２　mul4.v**

# 演習１の答え

```
module sub4(s,a,b);
  output     [4:0] s;
  input      [3:0] a,b;
  assign  s=a-b;
endmodule
```

```
module testsub4;
  wire                    [4:0] s;
  reg          [3:0] a, b;
  initial begin
       $monitor( “%t %b - %b = %b”, $time, a, b, s);
       a <= 0;   b <= 0;
       #40       a <= 1;   b <= 3;
       #40       a <= 4;   b <= 8;
       #40       a <= $random;   b <= $random;
       #40       a <= $random;   b <= $random;
       #40
       $finish;
  end
  sub4    sub ( s,a,b );
endmodule
```

2の補数表現
1,0を反転させて1を足す

```
  0 0000 0- 0000 0= 00000 0
 40 0001 1- 0011 3= 11110 -2
 80 0100 4- 1000 8= 11100 -4
120 0100 4- 0001 1= 00011 3
160 1001 9- 0011 3= 00110 6
```

```
module mul4(s,a,b);
  output     [7:0] s;
  input      [3:0] a,b;
  assign  s=a*b;
endmodule
```

```
module testmul4;
  wire                    [7:0] s;
  reg          [3:0] a, b;
  initial begin
       $monitor( “%t %b * %b = %b”, $time, a, b, s);
       a <= 0;   b <= 0;
       #40       a <= 1;   b <= 3;
       #40       a <= 4;   b <= 8;
       #40       a <= $random;   b <= $random;
       #40       a <= $random;   b <= $random;
       #40
       $finish;
  end
  mul4    mul ( s,a,b );
endmodule
```

```
 0 0000 * 0000 = 00000000
 40 0001 * 0011 = 00000011
 80 0100 * 1000 = 00100000
120 0100 * 0001 = 00000100
160 1001 * 0011 = 00011011
```

# testadd42.v もう少し洒落たテストベンチ

```
module testadd42;
  wire                [4:0] s;
  reg          [3:0] a, b;
  reg          ck;
  initial begin
      $monitor( “%t %b + %b = %b”, $time, a, b, s);
      a <= 0;   b <= 0;
      ck <= 0;
      #400
      $finish;
  end
  always  #10 ck <= ~ck;
  always @(posedge ck) begin
      a <= $random;
      b <= $random;
  end
  add4 add ( s,a,b );
endmodule
```

% **iverilog testadd42.v add4.v**

% **./a.out**

```
  0 0000 + 0000 = 00000
 10 0100 + 0001 = 00101
 30 1001 + 0011 = 01100
 50 1101 + 1101 = 11010
 70 0101 + 0010 = 00111
 90 0001 + 1101 = 01110
110 0110 + 1101 = 10011
          ...
350 1010 + 1101 = 10111
370 0110 + 0011 = 01001
390 1101 + 0011 = 10000
```

# 全数をチェックしたければ

testadd43.v

```
module testadd43;
  wire          [4:0] s;
  reg           [3:0] a, b;
  reg                 ck;
  reg                 flag;
  initial begin
        $monitor( "%t %b + %b = %b", $time, a, b, s);
        a <= 0;   b <= 0; flag <= 0;
        ck <= 0;
  end
  always  #10 ck <= ~ck;
  always @(negedge ck) begin
        if( s != a + b ) begin
         flag <= 1;
         $finish;
       end
       if( a == 'h f && b == 'h f ) begin
         $display( "OK¥n" );
         $finish;
       end
  end
  always @(posedge ck) begin
        {b,a} <= {b,a} + 1;
  end
  add4 add ( s,a,b );
endmodule
```

```
% iverilog   testadd43.v   add4.v
% ./a.out
```

```
   0 0000 + 0000 = 00000
  10 0001 + 0000 = 00001
  30 0010 + 0000 = 00010
  50 0011 + 0000 = 00011
  70 0100 + 0000 = 00100
  90 0101 + 0000 = 00101
           ...
5030 1100 + 1111 = 11011
5050 1101 + 1111 = 11100
5070 1110 + 1111 = 11101
5090 1111 + 1111 = 11110
OK
```

# 演習２

- count4r.v, testcount4r.vをダウンロードして実行
- count4r.vを修正しデクリメント（クロックごとに－１）するcount4rs.v, ２づつ加算するcount4r2.v, 2倍づつ乗算するcount4r2m.vを作成し、それらに対応するテストベンチtestcount4rs.v, testcount4r2.v, testcount4r2m.v を作成して実行し結果を確認

**課題２－１　count4rs.v**
**課題２－２　count4r2.v**
**課題２－３　count4r2m.v**

# 演習2の期待される結果

count4rs.v

```
  0 0 0 0000
 10 1 0 0000
 20 0 0 0000
 30 1 0 0000
 40 0 1 0000
 50 1 1 1111
 60 0 1 1111
 70 1 1 1110
 80 0 1 1110
 90 1 1 1101
100 0 1 1101

370 1 1 1111
380 0 1 1111
390 1 1 1110
```

count4r2.v

```
  0 0 0 0000
 10 1 0 0000
 20 0 0 0000
 30 1 0 0000
 40 0 1 0000
 50 1 1 0010
 60 0 1 0010
 70 1 1 0100
 80 0 1 0100
 90 1 1 0110
100 0 1 0110


370 1 1 0010
380 0 1 0010
390 1 1 0100
```

初期値を1（0以外）にしておく必要あり

count4r2m.v

```
  0 0 0 0001
 10 1 0 0001
 20 0 0 0001
 30 1 0 0001
 40 0 1 0001
 50 1 1 0010
 60 0 1 0010
 70 1 1 0100
 80 0 1 0100
 90 1 1 1000
100 0 1 1000
110 1 1 0000
120 0 1 0000


390 1 1 0000
```

オーバーフローすると以後0となる

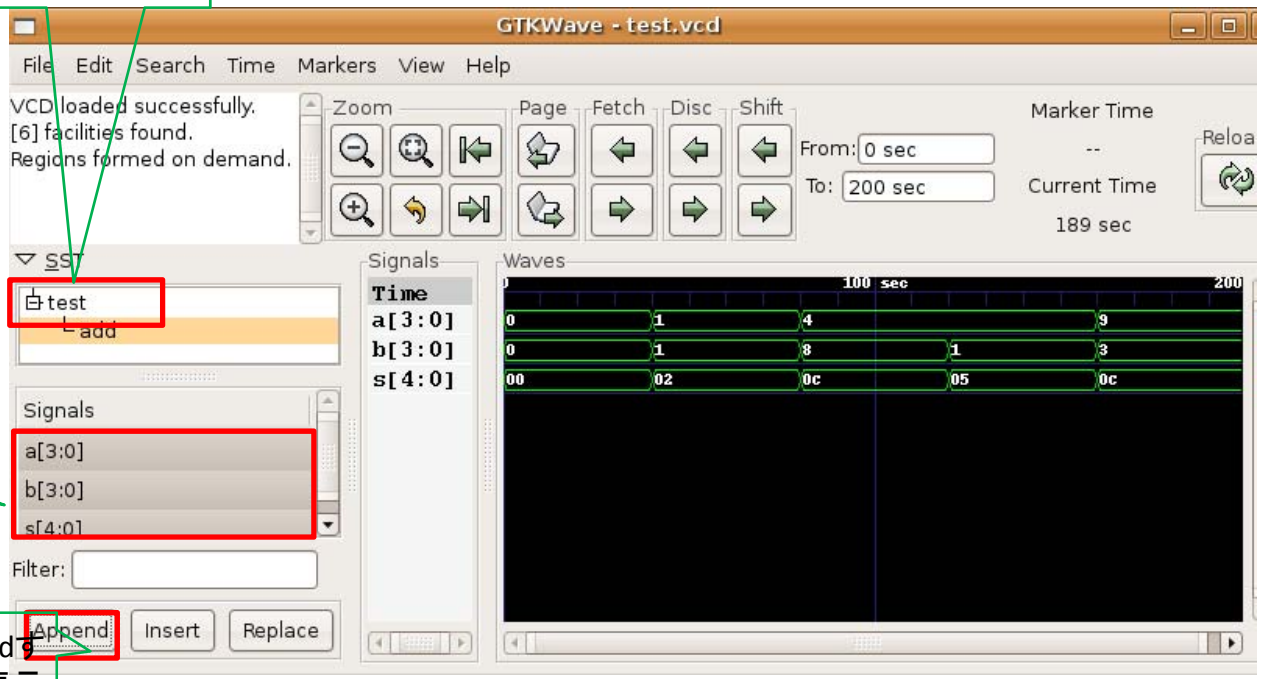
# グラフィカルに見たい

testadd4.v

```
module testadd4;
  wire          [4:0] s;
  reg           [3:0] a, b;
  reg                 ck;
  reg                 flag;
  initial begin
        $dumpfile("testadd.vcd");
        $dumpvars;
        a <= 0;   b <= 0; flag <= 0;
        ck <= 0;
  end
  always  #10 ck <= ~ck;
  always @(negedge ck) begin
        if( s != a + b ) begin
         flag <= 1;
         $finish;
       end
       if( a == 'h f && b == 'h f ) begin
         $display( "OK¥n" );
         $finish;
       end
  end
  always @(posedge ck) begin
        {b,a} <= {b,a} + 1;
  end
  add4 add ( s,a,b );
endmodule
```

```
% iverilog testadd4.v add4.v
% ./a.out
% gtkwave testadd.vcd
```

クリックすると下位モジュール名が表示される

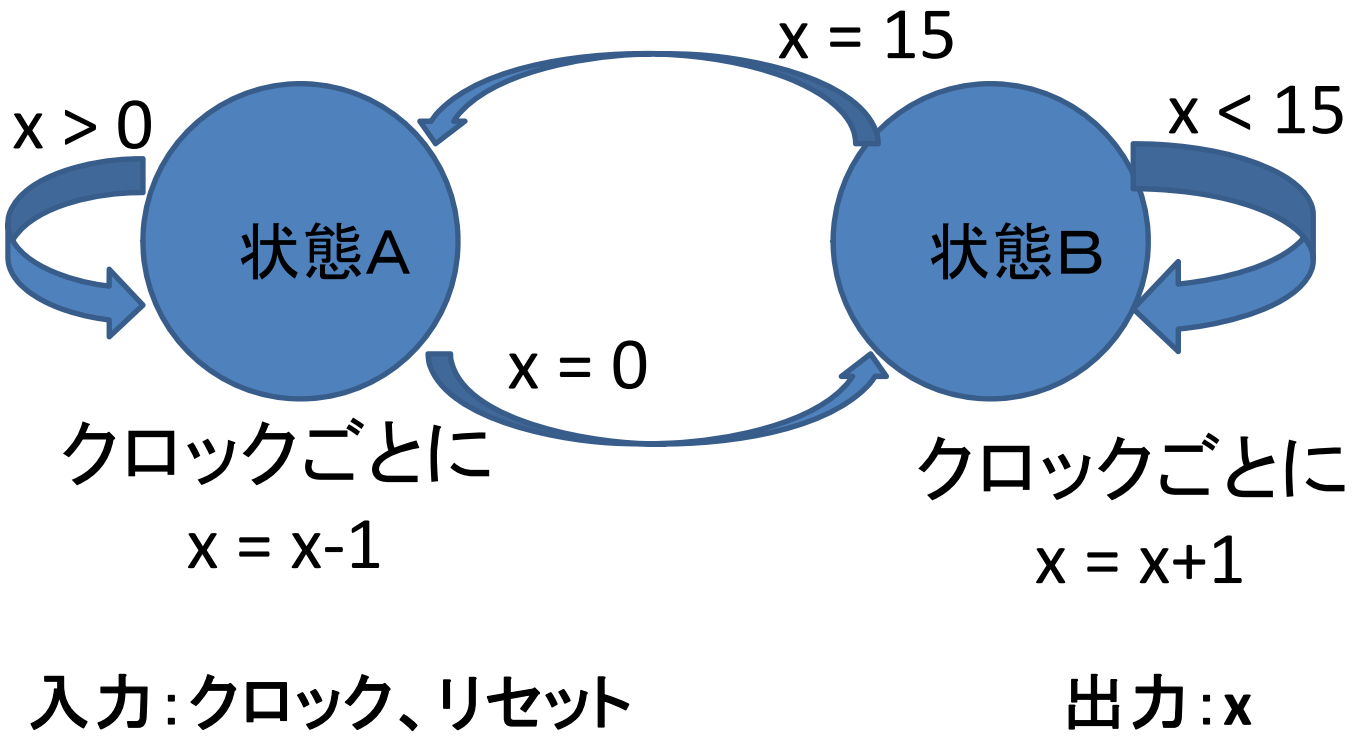



選択したモジュール内の信号が表示される

信号を選択してAppendすると右画面に波形が表示

# 演習４

- 簡単な状態機械を実現してみよう

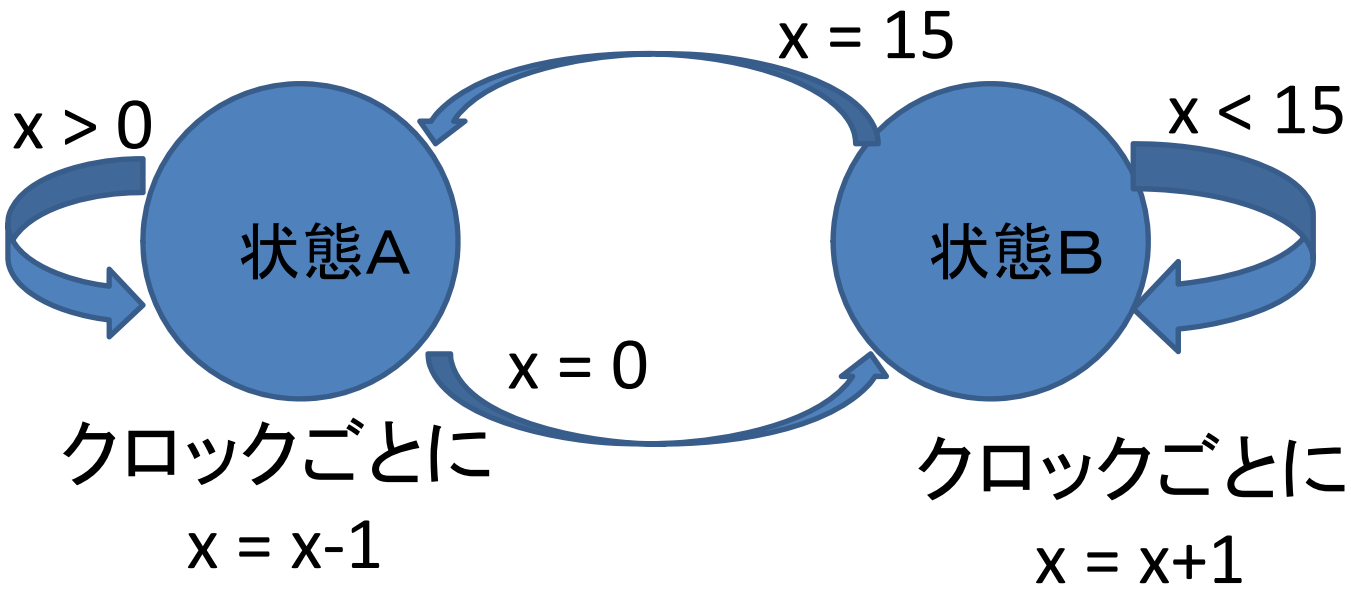


課題4　　stm.v

# 演習4：状態の定義

状態変数: st,
状態A: st=0, 状態B: st=1,
リセット時状態：A

変数: x
リセット時: x=0




入力：クロック、リセット

出力：x

状態機械の構成

```
always @(posedge ck) begin
    if( st == 0 ) begin   // State A
        if( x == 0 ) st <= 1;
        else     x<=x-1;
    end else begin        // State B
        if( x == 15 ) st <= 0;
        else     x<=x+1;
    end
end
```

テストベンチ

```
module simstm;
reg ck, rst;
initial begin
    $dumpfile("stm.vcd");
    $dumpvars;
    $monitor( "st = %b: x=%x", s.st, s.x );
    ck=0; rst=0;
    #20 rst=1;
    #60 rst=0;
    #1000 $finish;
end
always #10  ck<=~ck;
    stm     s(ck,rst);
endmodule
```

全体

```
module stm(ck,rst);
input    ck,rst;
reg [3:0] x;
reg      st;
always @(posedge ck) begin
    if( rst == 1 ) begin
        st <= 0;
        x <= 0;
    end else begin

    end
end
endmodule
```

## stm.v

全体

```
module stm(ck,rst,x);
input       ck,rst;
output      [3:0] x;
reg [3:0] x;
regst;
always @(posedge ck) begin
    if( rst == 1 ) begin
            st <= 0;
            x <= 0;
    end else begin
            if( st == 0 ) begin
                    if( x == 0 ) st <= 1;
                    else x <= x-1;
            end else begin
                    if( x == 15 ) st <= 0;
                    else  x<= x+1;
            end
    end
end
endmodule
```

複数行にわたる場合には
必ず begin ～ endでくくる

## simstm.v

テストベンチ

```
module simstm;
reg ck, rst;
wire [3:0] x;
initial begin
    $dumpvars;
    $dumpfile(“stm.vdc”);
    $monitor( "st = %b: x=%x", s.st, s.x );
    ck=0; rst=0;
    #20 rst=1;
    #60 rst=0;
    #1000 $finish;
end
always #10          ck<=~ck;

stm         s(ck,rst,x);

endmodule
```

# ハードウエアにおける設計表現
# ハードウエア設計記述言語VerilogHDL
# ～種々の記述～

**Ubuntuを起動し**
**verilog**が実行できる状態にしておいてください。

# 本日の出欠は・・・

本日正午＋δまでに
WEBから課題５～７までを提出する。それまでに終わらない場合には、来週までに課題７まで終えてWEBにUPLOADしておくこと。（課題８，９は発展課題：挑戦してみてください）

# VerilogHDLの実行結果の確認

- 実行結果

% **iverilog simstm.v stm.v**

% **./a.out**

- エラー例

% iverilog simstm.v stm.v

simstm.v:9: syntax error

simstm.v:8: error: malformed statement

ありがちなエラー

wire型に手続き代入 ( <= )をしようとしている

reg型に継続代入( assign文)をしようとしている

# VerilogHDLの実行結果の確認

- 正常な場合

ck = 0, st = x: x=x
ck = 1, st = x: x=x
ck = 0, st = x: x=x
ck = 1, st = 0: x=0　　初期化されるまでは値は x を持つ
・・・・・・・
ck = 0, st = 1: x=e
ck = 1, st = 1: x=f
ck = 0, st = 1: x=f
ck = 1, st = 0: x=f　　x=f の時は状態のみを遷移させる
ck = 0, st = 0: x=f
ck = 1, st = 0: x=e　　状態遷移後、次のクロックからxの減算が始まる
・・・・・・・
ck = 0, st = 0: x=1
ck = 1, st = 0: x=0
ck = 0, st = 0: x=0
ck = 1, st = 1: x=0　　x=0 の時は状態のみを遷移させる
ck = 0, st = 1: x=0
ck = 1, st = 1: x=1　　状態遷移後、次のクロックからxの加算が始まる
・・・・・・・

# 記述誤りとエラーの例

**http://www.mos.t.u-tokyo.ac.jp/~ikeda/HWDesign/**
**から alu_e1.v .. alu_e5.vをダウンロードして iverilogでコンパイル**

**alu_e1.v**
alu_e1.v:8: syntax error
alu_e1.v:7: error: syntax error in reg variable list.
　　きちんと記述しているはずなのにエラーが出る場合たいていは、前の行の最後の　;　がない
　　その結果、” syntax error in reg variable list” となる。
**alu_e2.v**
alu_e2.v:13: error: C is not a reg/integer/time in alu.
alu_e2.v:7:　　: C is declared here as wire.
Elaboration failed
　　wireに対して手続き代入をしようとしている。
**alu_e3.v**
alu_e3.v:16: error: reg OUT; cannot be driven by primitives or continuous assignment.
2 error(s) during elaboration.
　　regに対して継続代入をしようとしている。
**alu_e4.v**
alu_e4.v:1: error: Port CTR (4) of module alu is not declared within module.
alu_e4.v:12: error: Unable to bind wire/reg/memory `CTR' in `alu'
Elaboration failed
　　入出力ポートの定義がない
**alu_e5.v**
alu_e5.v:6: error: CTR in module alu declared as input and as a reg type.
1 error(s) during elaboration.
　　入力ポートに対してreg定義をしようとしている

# モジュール間の変数の参照

```
module   A (*****)
reg  hoge1, hoge2, hoge3;
endmodule


module TOP_A;

A  InstanceNameA (**** );

endmodule
```

simstm.v
テストベンチ

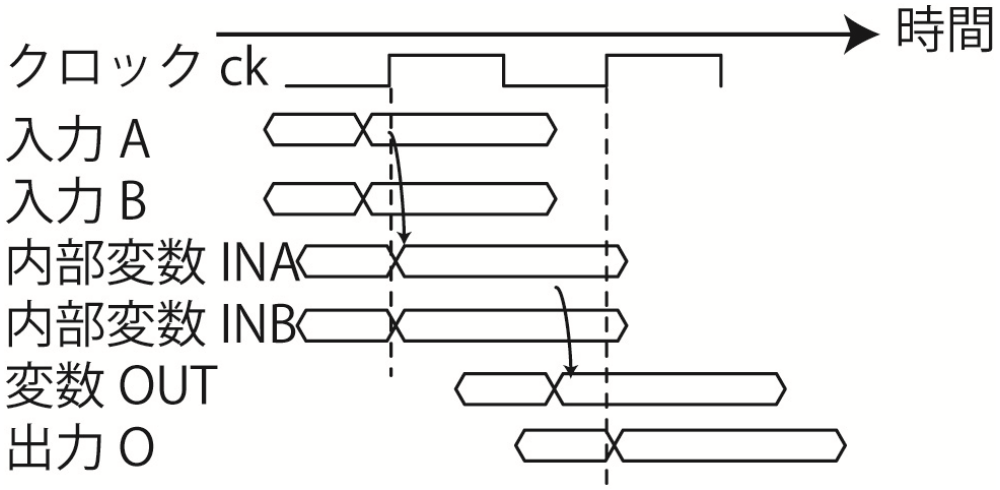
```
module simstm;
stm   s(ck,rst,x);
      $monitor( "st = %b: x=%x", s.st, s.x );
```

ハードウエアとしては、module Aの内部の変数を参照するためには、ポートから出力する必要がある。
→　不便なのでシミュレーションとしては、
TOP_Aモジュールから
InstanceNameA . hoge1
という形で InstanceNameAとして定義したモジュール内の変数を参照することができる

課題5　　alu.v

# 演習5：簡単な演算器・・1

- 入力A：8ビット、入力B：8ビット、出力O：8ビット
- 制御入力CTR：4ビット
  - 0000: 加算、0001: 減算
    1000: 論理積、1001: 論理和、1010: 排他的論理和、1011: 反転、
    1100: 1ビット右シフト（0で埋める）、1101: 1ビット左シフト(0で埋める), 1110: 1ビット右ローテーション(MSBをLSBで埋める),
    1111: 1ビット左ローテーション(LSBをMSBで埋める)
- 入力はクロックの立ち上がりで取り込み、1クロック後の立ち上がりで出力



# 実装例1

alu.v

**骨格**

```
module  alu(A,B,O,CTR,ck);
input       [7:0]       A, B;
input       [3:0]       CTR;
input       ck;
output      [7:0] O;
reg         [7:0]       INA, INB, C;
reg         [3:0]       C;
wire        [7:0]       OUT;

endmodule
```

**順序機械。。1**

```
always @(posedge ck) begin
            INA <= A;
            INB <= B;
            C <= CTR;
            O <= OUT;
end
```

**継続代入で実現**

```
assign  OUT=(C==’b0000 ? INA + INB :
            (C==’b0001 ? INA - INB :
            (C==’b1000 ? INA & INB :
            (C==’b1001 ? INA | INB :
            (C=’b1010 ? INA ^ INB :
            (C==’b1011 ? ~INA :
            (C==’b1100 ? INA>>1 :
            (C==’b1101 ? INA<<1 :
            (C==’b1110 ? {INA[0],INA[7:1]} :
            (C==’b1111 ? {INA[6:0],INA[7]} : 8’b0
));;;;;;;;;;
```

# 実装例2

alu2.v

**骨格**

```
module  alu(A,B,O,CTR,ck);
input       [7:0]       A, B;
input       [3:0]       CTR;
input       ck;
output      [7:0] O;
reg         [7:0]       INA, INB, OUT, O;
reg         [3:0]       C;

endmodule
```

**順序機械**

```
always @(posedge ck) begin
            C <= CTR;
            INA <= A;
            INB <= B;
            case  (C)
            'b0000 : O <= INA + INB;
            'b0001 : O <= INA - INB;
            'b1000 : O <= INA & INB;
            'b1001 : O <= INA | INB;
            'b1010 : O <= INA ^ INB;
            'b1011 : O <= ~INA;
            'b1100 : O <= INA>>1;
            'b1101 : O <= INA<<1;
            'b1110 : O <= {INA[0],INA[7:1]};
            'b1111 : O <= {INA[6:0],INA[7]};
            endcase
end
```

# 実装例2・・1

alu21.v

**骨格**

```
module  alu(A,B,O,CTR,ck);
input       [7:0]       A, B;
input       [3:0]       CTR;
input       ck;
output      [7:0] O;
reg         [7:0]       INA, INB, OUT, O;
reg         [3:0]       C;
```

```
endmodule
```

**順序機械。。2**

```
always @(posedge ck) begin
            C <= CTR;
            INA <= A;
            INB <= B;
            O <= OUT;
end
always  @(C or INA or INB) begin
            case  (C)
            'b0000 : OUT <= INA + INB;
            'b0001 : OUT <= INA - INB;
            'b1000 : OUT <= INA & INB;
            'b1001 : OUT <= INA | INB;
            'b1010 : OUT <= INA ^ INB;
            'b1011 : OUT <= ~INA;
            'b1100 : OUT <= INA>>1;
            'b1101 : OUT <= INA<<1;
            'b1110 : OUT <= {INA[0],INA[7:1]};
            'b1111 : OUT <= {INA[6:0],INA[7]};
            endcase
end
```

# 実装例2‥2

alu22.v

**骨格**

```
module  alu(A,B,O,CTR,ck);
input       [7:0]       A, B;
input       [3:0]       CTR;
input       ck;
output      [7:0] O;
reg         [7:0]       INA, INB, OUT, O;
reg         [3:0]       C;

endmodule
```

**順序機械**

```
always @(posedge ck) begin
            C = CTR;
            INA <= A;
            INB <= B;
            case  (C)
            'b0000 : O <= INA + INB;
            'b0001 : O <= INA - INB;
            'b1000 : O <= INA & INB;
            'b1001 : O <= INA | INB;
            'b1010 : O <= INA ^ INB;
            'b1011 : O <= ~INA;
            'b1100 : O <= INA>>1;
            'b1101 : O <= INA<<1;
            'b1110 : O <= {INA[0],INA[7:1]};
            'b1111 : O <= {INA[6:0],INA[7]};
            endcase
end
```

# 実装例3：functionを使用

alu3.v

**骨格**

```
module  alu(A,B,O,CTR,ck);
input       [7:0]       A, B;
input       [3:0]       CTR;
input       ck;
output      [7:0] O;
reg         [7:0]       INA, INB, O;
reg         [3:0]       C;

endmodule
```

**順序機械。。2**

```
always @(posedge ck) begin
            C <= CTR;
            INA <= A;
            INB <= B;
            O <= alufunc(INA,INB,C);
end
```

**function**

```
function  [7:0]  alufunc;
            input   [7:0] A;
            input   [7:0] B;
            input   [3:0] C;

            case  (C)
            'b0000 : alufunc = A + B;
            'b0001 : alufunc = A - B;
            'b1000 : alufunc = A & B;
            'b1001 : alufunc = A | B;
            'b1010 : alufunc = A ^ B;
            'b1011 : alufunc = ~A;
            'b1100 : alufunc = A>>1;
            'b1101 : alufunc = A<<1;
            'b1110 : alufunc = {A[0], A[7:1]};
            'b1111 : alufunc = {A[6:0], A[7]};
            endcase
endfunction
```

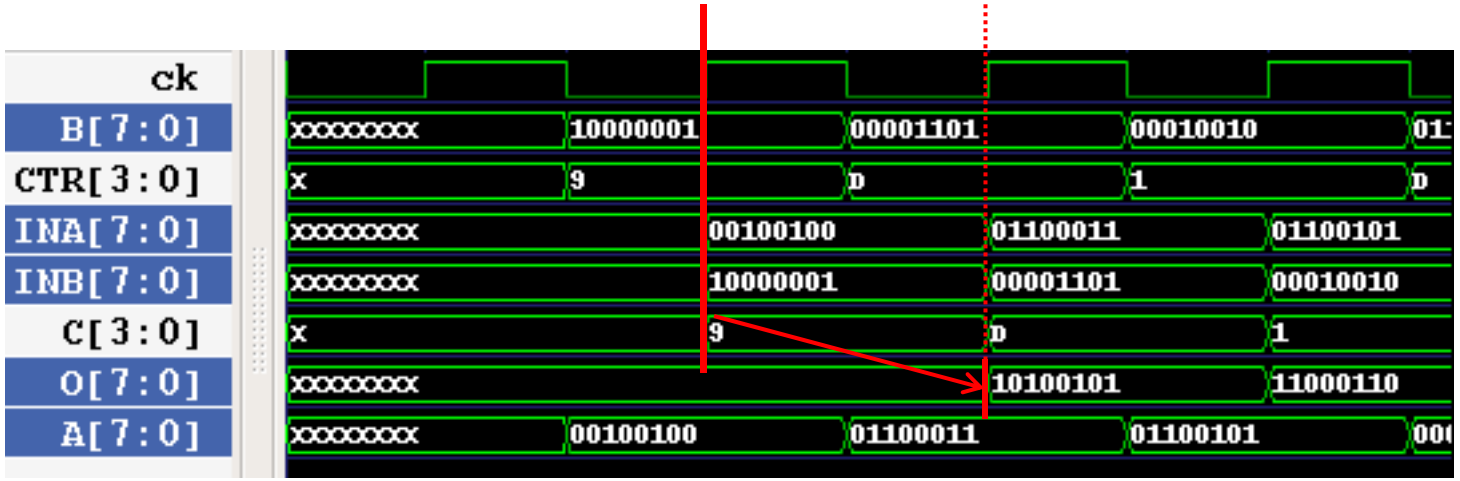
# テストベンチ

alutest.v

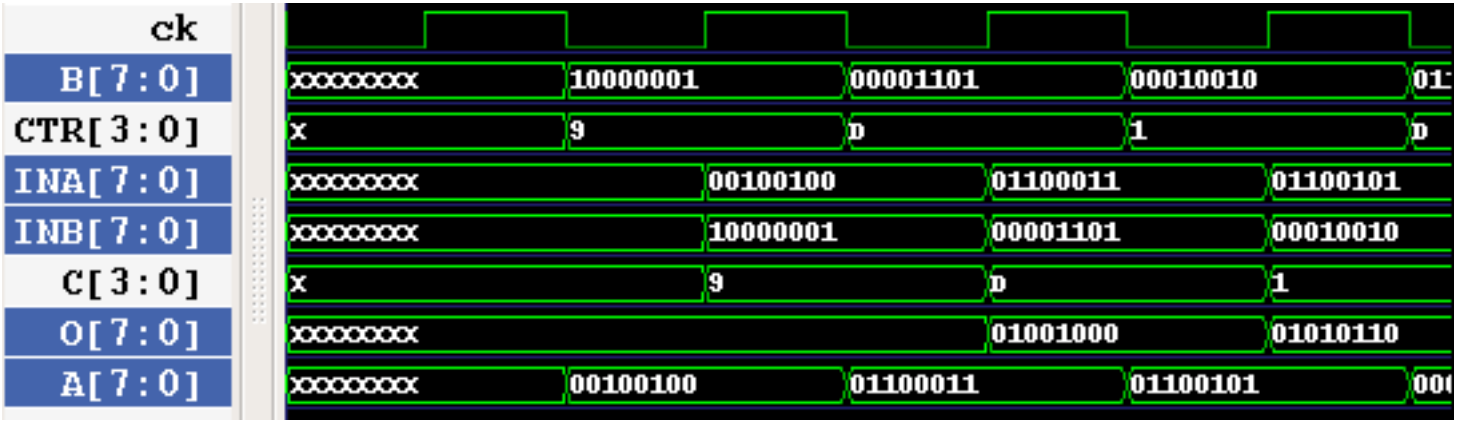
**骨格**

```
module  alutest;
reg         [7:0]       A, B;
reg         [3:0]       CTR;
reg         ck;
wire        [7:0] O;
initial begin
            ck=0;
            $monitor( “%t¥tA=%h, B=%h, CTR=%h, OUT=%h”, $time, A, B, CTR, O );
            #1000  $finish;
end
alu         ALU(A , B , O , CTR , ck);
always    #10         ck = ~ck;
always @(negedge ck) begin
            A = $random;
            B = $random;
            CTR  = $random;
end
endmodule
```

# 実行結果を確認してみましょう




alu.v



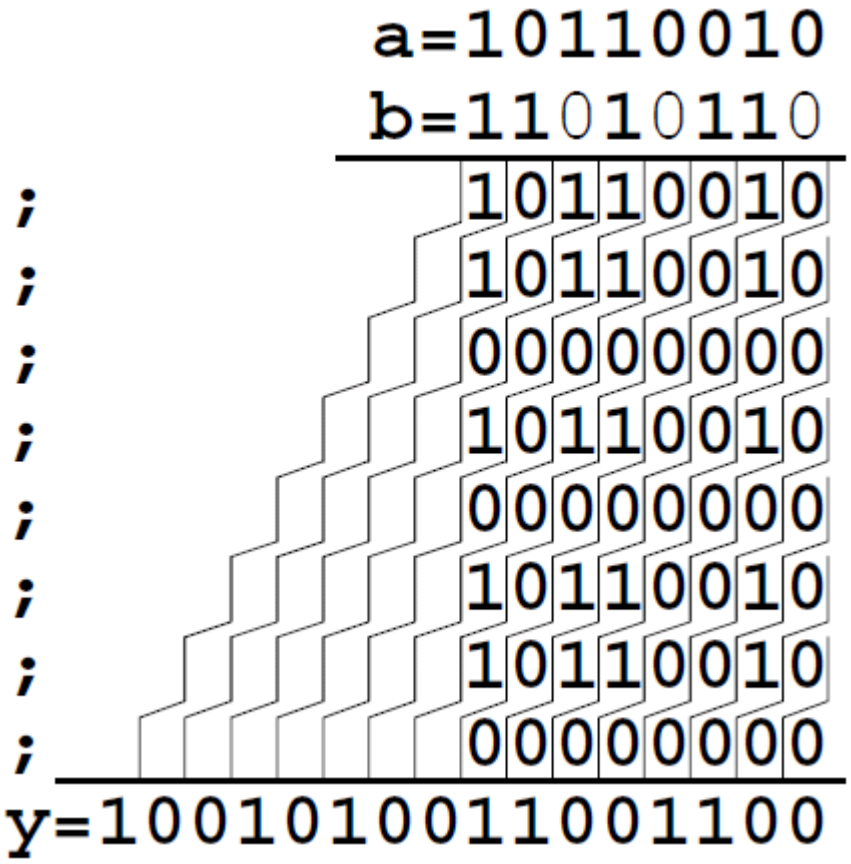

alu22.v

# 課題6　　mul.v (multest2.vを使用)

# 演習6：乗算の実装

```
    1101  被乗数 a
    1011  乗数 b
  -------
    1101  部分積 a*b[0]
   1101   部分積 a*b[1]
  0000    部分積 a*b[2]
 1101     部分積 a*b[3]
--------
10001111  積（ 部分積の総和）
```

```
y = 0;
y = (y<<1)+a*b[7];
y = (y<<1)+a*b[6];
y = (y<<1)+a*b[5];
y = (y<<1)+a*b[4];
y = (y<<1)+a*b[3];
y = (y<<1)+a*b[2];
y = (y<<1)+a*b[1];
y = (y<<1)+a*b[0];
```



# 乗算の実装：複数サイクルで実行

```
  1101  被乗数 a
  1011  乗数 b
 ------
  1101  部分積 a*b[0]
 1101   部分積 a*b[1]
0000    部分積 a*b[2]
1101    部分積 a*b[3]
 ------
10001111 積（部分積の総和）
```

入力 A, B, ck, start
start=1で A, Bを内部レジスタ AIN, BINに取り込み
状態変数 stを0、終了フラグ finを0とする
ck毎にstをインクリメント、以下のような演算を実行
st=7(演算終了）でfin=1とする
st=8でfin=0とする

start=1 `y = 0;`

st=0 `y = (y<<1)+a*b[7];`

st=1 `y = (y<<1)+a*b[6];`

st=2 `y = (y<<1)+a*b[5];`

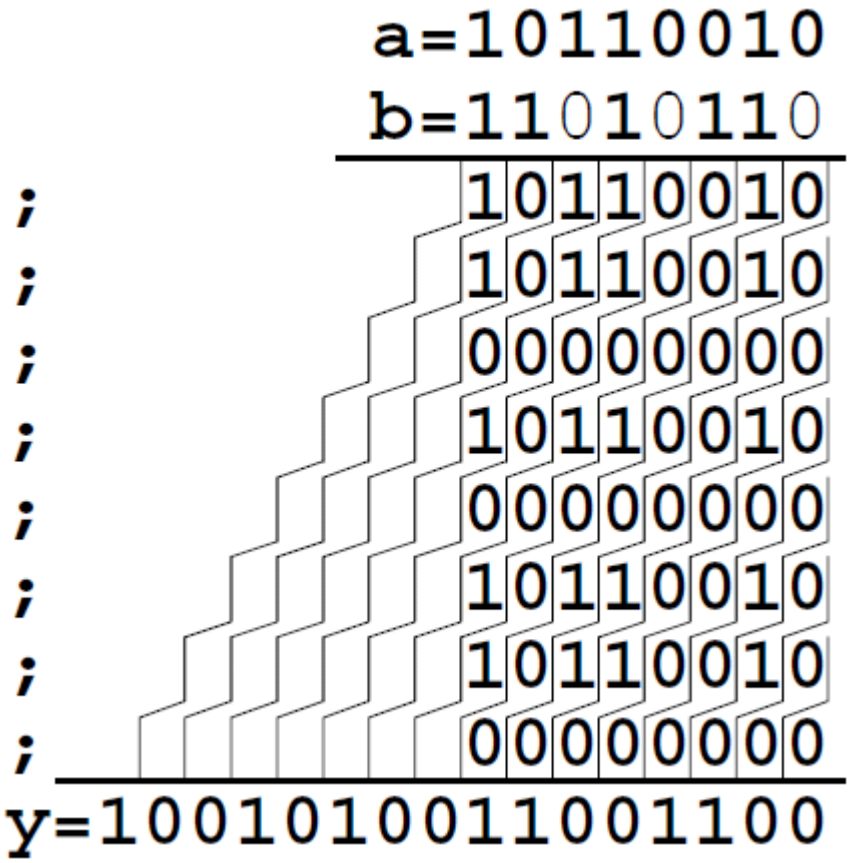
st=3 `y = (y<<1)+a*b[4];`

st=4 `y = (y<<1)+a*b[3];`

st=5 `y = (y<<1)+a*b[2];`

st=6 `y = (y<<1)+a*b[1];`

st=7 `y = (y<<1)+a*b[0];`



# 複数クロックでの実装

```
always @(posedge ck) begin
        if( start == 1 ) begin



        end else begin









        end
end
```

```
module mul(A,B,O,ck,start,fin);
input [7:0] A, B;
input ck,start;
output [16:0] O;
output fin;
```

変数（レジスタ等）の定義

実行

```
endmodule
```

multest.v

# 複数クロックでの実装（テストベンチ）

```
module  multest;
reg         [7:0]       A, B;
reg         ck;
reg         start;
reg         [3:0] st;
wire        [16:0] O;
reg         [16:0] OR;
initial begin
            ck=0;
            start=0;
            st=0;
            $monitor( "%t¥tA=%h, B=%h, CTR=%h, (OUT=%h) OUT=%h", $time, A, B, CTR, O, OR );
            #1000  $finish;
end
```

```
mul         MUL(A , B , O ,  ck, start,fin);
always      #10         ck = ~ck;
always @(negedge ck) begin
            if( st == 0 ) start <= 1;
            else start <= 0;
            if( fin == 1 ) OR <= O;
            st <= st+1;
            A = $random;
            B = $random;
end
endmodule
```

全数チェックするにはmultest2.v（WEBから取得）

# 演習7　課題7　　fifo.vの完成(simfifo.vを使用)

fifo.v

**モジュール構成の理解**

8ビット16段のFIFOを完成させよ（fifo.vをダウンロードして完成させ実行結果で確認）

モジュール名: fifo

入力：8ビット入力データ: Din, クロック: ck, データ入力フラグ: Wen, データ出力フラグ: Ren

リセット: rst

出力：8ビット出力データ: Dout, FIFOエンプティフラグ: Fempty, FIFOフルフラグ: Ffull

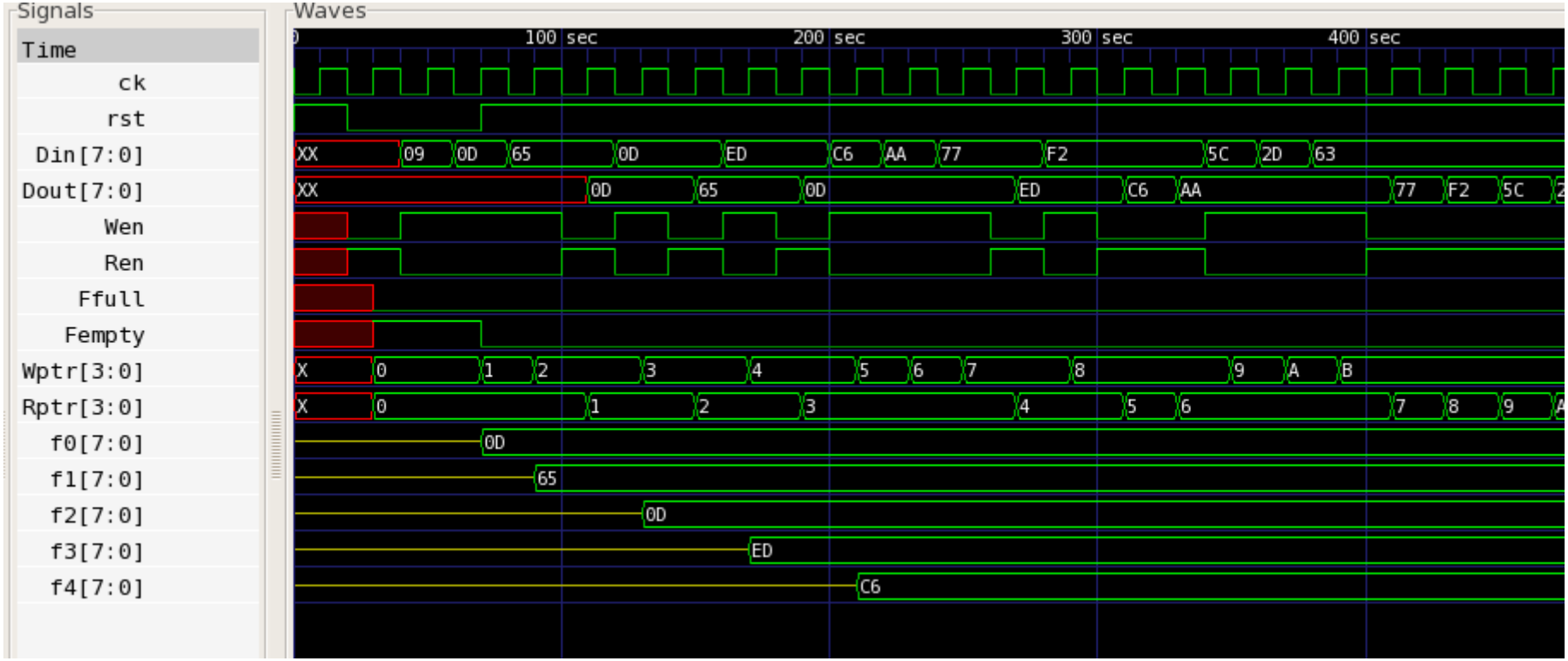
```
reg  [7:0] FMEM[0:15];
reg  [3:0] Wptr, Rptr;
reg  Fempty, Ffull;
reg  [7:0] obuf;
wire [3:0] NWptr, NRptr;

assign Dout = obuf;
assign NWptr = Wptr + 1;
assign NRptr = Rptr + 1;
```

```
always @(posedge ck) begin
   if( !rst ) begin
         Wptr <= 0;  Rptr <= 0;  Fempty <= 1;  Ffull <= 0;
   end else begin
         if( Ren == 1 && Fempty != 1 ) begin
            obuf <= FMEM[Rptr];  Rptr <= NRptr; Ffull <= 0;
            if( NRptr == Wptr ) Fempty <= 1;
            else Fempty <= 0;
         end
         if( Wen == 1 && Ffull != 1 ) begin
            FMEM[Wptr] <= Din; Wptr <= Wptr + 1; Fempty <= 0;
            if( NWptr == Rptr ) Ffull <= 1;
            else Ffull <= 0;
         end
   end
end
```

ここは Wptr <= NWptrでもよい

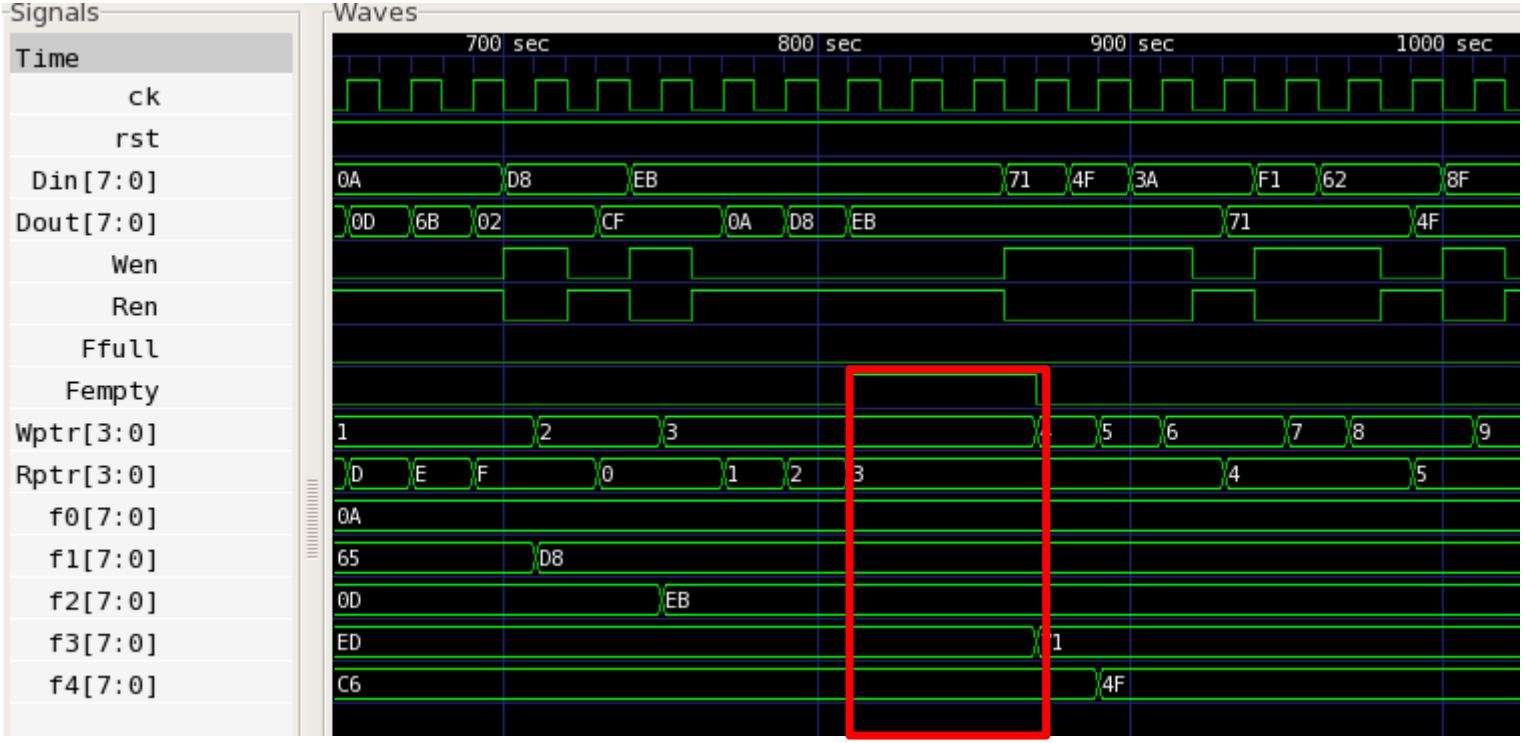
テストベンチはsimfifo.v

# FIFOの動作・・1（正常動作）



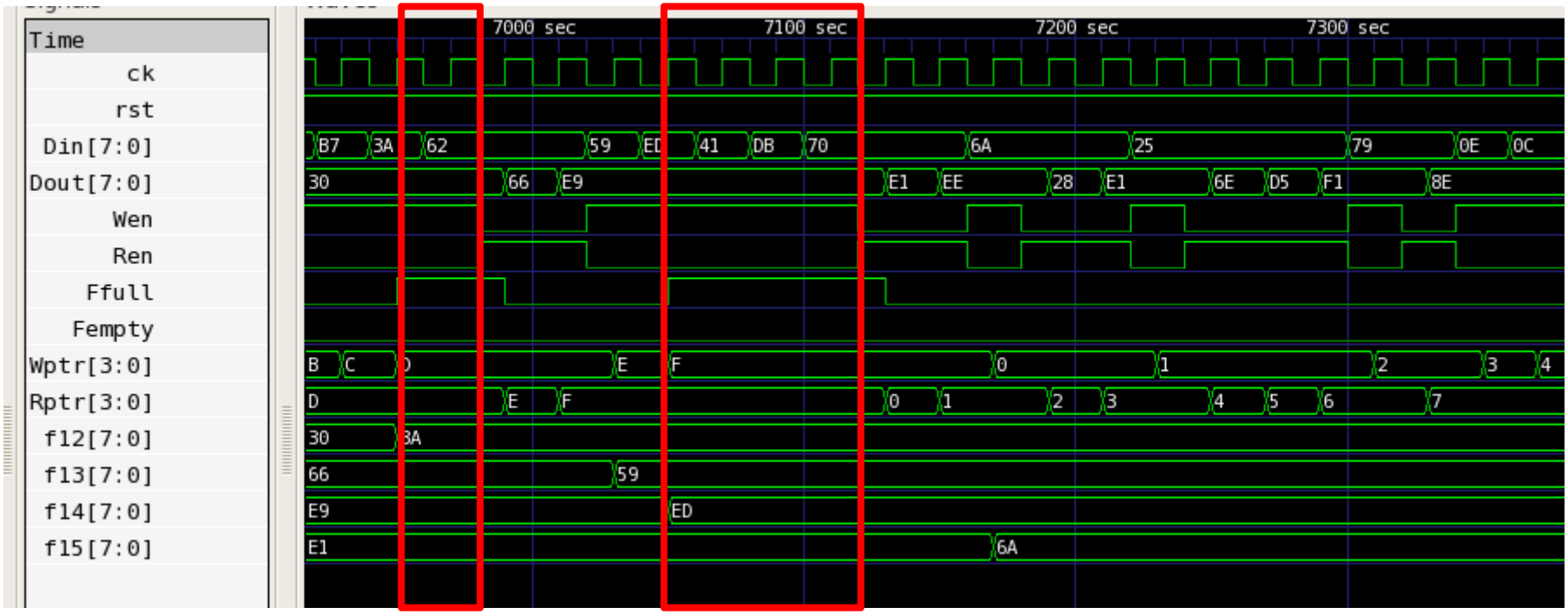

f0, f1,,,はFMEM[0..15]の内容を表す
Wen＝１の時には、クロックごとにDinが内部に書き込まれる
　　同時に、Wptrがインクリメントされる
Wen＝１の時には、クロックごとにFMEMの内容がDoutに出力される
　　同時に、Rptrがインクリメントされる

# FIFOの動作・・1（FIFO empty）




Ren＝1の時に Wptr == Rptr（Fempty=1)だと、FIFOが空っぽであるため読み出しは行われない

# FIFOの動作・・1（FIFO full）




Wen＝１の時に Wptr == Rptr（Ffull=1)だと、FIFOがいっぱいであるため書き込みは行われない

# crc.v 演習８ 課題８　　crc.vの完成(simcrc.vを使用)

**定義の理解**

エラーチェック・訂正符号（crc.vをダウンロードして完成させ実行結果で確認）
モジュール名: crc
$x^5+x^2+1$の剰余系（ただし、初期値は 11111）
変数 lfsr_q, lfsr_cの定義を行え

```
module crc( data_in, crc_en,
crc_out, rst, ck );
input [7:0] data_in;
input crc_en;
output [4:0] crc_out;
input rst;
input ck;

assign crc_out = lfsr_q;
assign lfsr_c[0] = lfsr_q[0] ^ lfsr_q[2] ^ lfsr_q[3]
        ^ data_in[0] ^ data_in[3] ^ data_in[5] ^ data_in[6];
assign lfsr_c[1] = lfsr_q[1] ^ lfsr_q[3] ^ lfsr_q[4]
        ^ data_in[1] ^ data_in[4] ^ data_in[6] ^ data_in[7];
assign lfsr_c[2] = lfsr_q[0] ^ lfsr_q[3] ^ lfsr_q[4] ^ data_in[0]
        ^ data_in[2] ^ data_in[3] ^ data_in[6] ^ data_in[7];
assign lfsr_c[3] = lfsr_q[0] ^ lfsr_q[1] ^ lfsr_q[4] ^ data_in[1]
        ^ data_in[3] ^ data_in[4] ^ data_in[7];
assign lfsr_c[4] = lfsr_q[1] ^ lfsr_q[2] ^ data_in[2] ^ data_in[4]
        ^ data_in[5];
always @(posedge ck) begin
   if(rst)   lfsr_q <= 'b 11111;
   else      lfsr_q <= crc_en ? lfsr_c : lfsr_q;
end
endmodule
```

# sbox.v 演習９ 課題９

## 手続き文の理解

DESのSボックス処理の実施
ck毎に右表に示す通りinに従い
sを出力するモジュール sboxを
完成させる

## sbox.vの完成(simsbox.v使用)

| in[0] | in[3] | {in[1],in[2]} | | | |
|---|---|---|---|---|---|
| | | 00 | 01 | 10 | 11 |
| 0 | 0 | s=14 | s=4 | s=13 | s=1 |
| 0 | 1 | s=0 | s=15 | s=7 | s=4 |
| 1 | 0 | s=4 | s=1 | s=14 | s=8 |
| 1 | 1 | s=15 | s=12 | s=8 | s=2 |

```
module sbox(in, s, ck);
input [3:0] in;
output [3:0] s;
input ck;

reg [3:0] s;

always @(posedge ck) begin

end
endmodule
```

テストベンチはsimsbox.v

# ハードウエアにおける設計表現
# ハードウエア設計記述言語VerilogHDL
# ～さらなる応用～

**Ubuntuを起動し**
**verilog**が実行できる状態にしておいてください。

# 本日の出欠は・・・

本日正午＋δまでに
**WEB**から課題１０－２を提出する。それまでに終わらない場合には、次回（６月５日）までに課題１０－２まで終えてWEBにUPLOADしておくこと。

# 課題5のテストベンチ

alutest2.v

**骨格**

```
module  alutest;
・・・・
alu         ALU(A , B , O , CTR , ck);
・・・・
always @(negedge ck)
   if( !(O == O0 || ( (CC == 'b0010 || CC== 'b0011 || CC == 'b0100 ||
              CC == 'b0101 || CC=='b0110 || CC == 'b0111) && O0 == 8'b0)) )
              $finish;
always @(posedge ck) begin
       C <= CTR;    CC <= C;    INA <= A;    INB <= B;    O <= OUT;
end
always @(C or INA or INB) begin
        case  (C)
        'b0000 : OUT <= INA + INB;
・・・・・・・・・
        'b1111 : OUT <= {INA[6:0], INA[7]};
//        default : OUT <= 0;
        endcase
end
```

CTRが無定義のコードの場合には
演算結果は0もしくは直前の値

CTRは演算結果判定2クロックあ
とで使用・・2クロック分遅らせる

multest2.v

# 課題6のテストベンチ

```
module  multest;
reg     [7:0]   A, B;
reg         [3:0] st;
・・・・・・
initial begin
・・・・・・・・・
end
mul     MUL(A , B , O ,  ck, start,fin);
always @(negedge ck) begin
        if( st == 0 )  start <= 1;
        else start <= 0;
        if( fin == 1 ) begin
            OR <= O;
            st <= 0;
            {B,A} <= {B,A} + 1;
            if( O != A*B )    $finish;
            if( A == 'h f && B == 'h f ) begin
              $display( "OK¥n" );   $finish;
            end
        end  else  st <= st+1;
end
```


stが0になるまで待って（stは4ビットなので乗算は16クロック以内に終わることを前提としている、そうでないとテストベンチが誤動作）演算実施（＝start→1）


finが出力されると即次の演算実施（＝start→1）

# fifo.v 演習7

8ビット16段のFIFOを完成させよ（fifo.vをダウンロードして完成させ実行結果で確認）
モジュール名: fifo
入力：8ビット入力データ: Din, クロック: ck, データ入力フラグ: Wen, データ出力フラグ: Ren
　　リセット: rst
出力：8ビット出力データ: Dout, FIFOエンプティフラグ: Fempty, FIFOフルフラグ: Ffull

```
module fifo ( Din, Dout, Wen, Ren, rst, ck, Fempty, Ffull );
input [7:0] Din;
output [7:0] Dout;
input Wen, Ren, rst, ck;
output Fempty, Ffull;

reg  [7:0] FMEM[0:15];
・・・・・・以下変更なし
assign f0 = FMEM[0];
```

メモリーの内容は通常は参照することができない
→アドレス毎にassign文で切り分けてあげることでシミュレーション中に参照（デバッグ）ができるようになる

# 演習7：続き

fifo.vの実行部分

```
always @(posedge ck) begin
   if( !rst ) begin
            Wptr <= 0;  Rptr <= 0;  Fempty <= 1;  Ffull <= 0;
   end else begin
            if( Ren == 1 && Fempty != 1 ) begin
               obuf <= FMEM[Rptr];  Rptr <= NRptr; Ffull <= 0;
               if( NRptr == Wptr ) Fempty <= 1;
               else Fempty <= 0;
            end
            if( Wen == 1 && Ffull != 1 ) begin
               FMEM[Wptr] <= Din; Wptr <= Wptr + 1; Fempty <= 0;
               if( NWptr == Rptr ) Ffull <= 1;
               else Ffull <= 0;
            end
   end
end
```

ここは Wptr <= NWptrでもよい

テストベンチはsimfifo.v

# crc.v 演習8

**定義の理解**

エラーチェック・訂正符号（crc.vをダウンロードして完成させ実行結果で確認）

モジュール名: crc

$x^5+x^2+1$の剰余系（ただし、初期値は 11111）

変数 lfsr_q, lfsr_cの定義を行え

```
module crc( data_in, crc_en,
crc_out, rst, ck );
input [7:0] data_in;
input crc_en;
output [4:0] crc_out;
input rst;
input ck;

reg [4:0] lfsr_q;
wire [4:0] lfsr_c;
```

simcrc2.v

```
module top;
・・・
crc crc(data_in, crc_en, crc_out, rst, ck);
initial begin
・・・
end

always @(negedge ck) begin
  if( crc_out != lfsr_q) $finish;
end

always @(posedge ck) begin
if(rst)   lfsr_q <= {5{1'b1}};
else lfsr_q <= crc_en ? lfsr_c : lfsr_q;
end
endmodule
```

# sbox.v 演習９ 課題９

## 手続き文の理解

DESのSボックス処理の実施
ck毎に右表に示す通りinに従い
sを出力するモジュール sboxを
完成させる

# sbox.vの完成(simsbox.v使用)

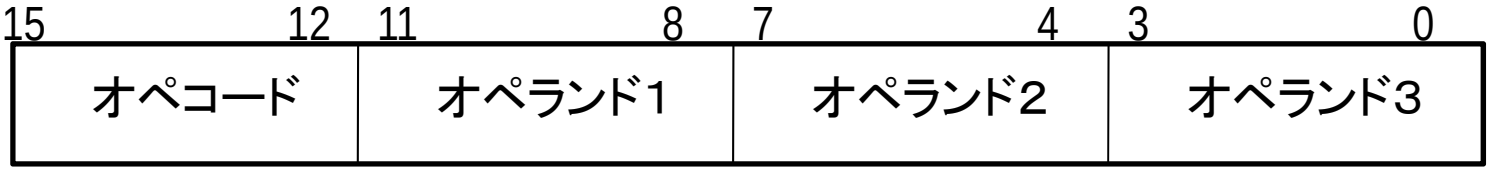
| in[0] | in[3] | {in[1],in[2]} | | | |
|---|---|---|---|---|---|
| | | 00 | 01 | 10 | 11 |
| 0 | 0 | s=14 | s=4 | s=13 | s=1 |
| 0 | 1 | s=0 | s=15 | s=7 | s=4 |
| 1 | 0 | s=4 | s=1 | s=14 | s=8 |
| 1 | 1 | s=15 | s=12 | s=8 | s=2 |

```
module sbox(in, s, ck);
input [3:0] in;
output [3:0] s;
input ck;
reg [3:0] s;
always @(posedge ck) begin
  case ({in[0],in[3]})
  0: begin    case ({in[1],in[2]})     0: s <= 14;     1: s <= 4;      2: s <= 13;     3: s <= 1;     endcase  end
  1: begin    case ({in[1],in[2]})    0: s <= 0;      1: s <= 15;     2: s <= 7;      3: s <= 4;    endcase    end
  2: begin    case ({in[1],in[2]})     0: s <= 4;      1: s <= 1;      2: s <= 14;     3: s <= 8;    endcase  end
  3: begin    case ({in[1],in[2]})     0: s <= 15;      1: s <= 12;      2: s <= 8;      3: s <= 2;    endcase  end
  endcase
end
endmodule
```

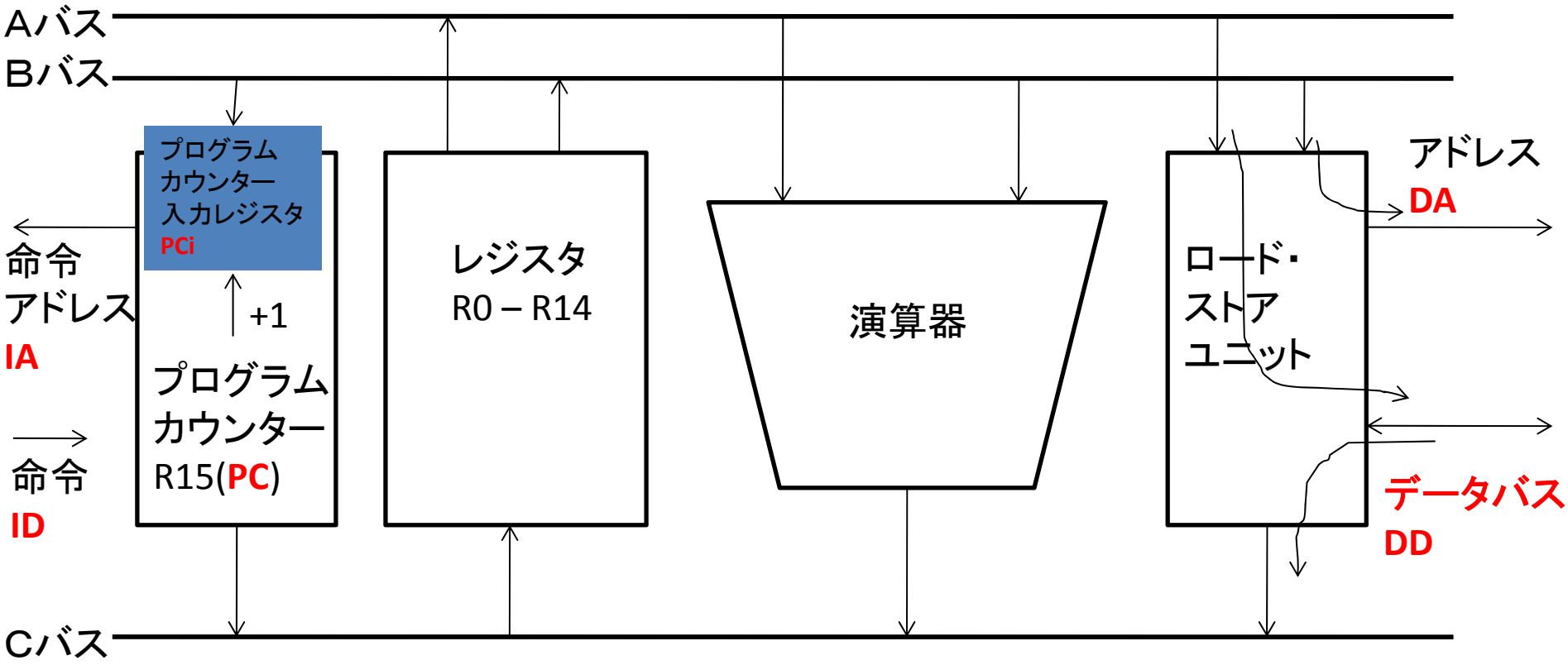
# 演習10

- 簡単なマイクロプロセッサを作ってみよう
  - 命令16ビット
    - 加算、減算、右シフト、左シフト、論理和、論理積、論理反転、排他的論理和
    - ジャンプ、条件分岐（ゼロ）
    - ロード、ストアー、下位ビットセット
  - データ16ビット、ゼロフラグ
  - レジスタ：16本、ただし0番レジスタは常に0、15番レジスタはプログラムカウンタ

命令語



# 演習10：CPUの構造


Aバス
Bバス
プログラム
カウンター
入力レジスタ
PCi
+1
命令
アドレス
IA
プログラム
カウンター
R15(PC)
命令
ID
レジスタ
R0 – R14
演算器
ロード・
ストア
ユニット
アドレス
DA
データバス
DD
Cバス

# 演習10：命令セット

| 命令 | オペコード | オペランド1 | オペランド2 | オペランド3 | 命令の詳細 |
|---|---|---|---|---|---|
| 加算 | 0000 | RC | RA | RB | [RA] + [RB] -> [RC] |
| 減算 | 0001 | RC | RA | RB | [RA] – [RB] -> [RC] |
| 右シフト | 0010 | RC | RA | RB | [RA] >> [RB] -> [RC] |
| 左シフト | 0011 | RC | RA | RB | [RA] << [RB] -> [RC] |
| 論理和 | 0100 | RC | RA | RB | [RA] \| [RB] -> [RC] |
| 論理積 | 0101 | RC | RA | RB | [RA] & [RB] -> [RC] |
| 論理反転 | 0110 | RC | RA | RB | ~[RA] -> [RC] |
| 排他的論理和 | 0111 | RC | RA | RB | [RA] ^ [RB] -> [RC] |
| 下位ビットセット | 1100 | RC | 即値データ | | {‘8b0,IMM} -> [RC] |
| ジャンプ | 1000 | RC | 0000 | RB | [RB] -> [PC] , [PC]+1 -> [RC] |
| 条件分岐（ゼロ） | 1001 | 0000 | 0000 | RB | If( flag ) [RB] -> [PC] |
| ロード | 1011 | RC | 0000 | RB | #[RB] -> [RC] |
| ストアー | 1010 | 0000 | RA | RB | [RA] -> #[RB] |

# 演習10：CPUの動作

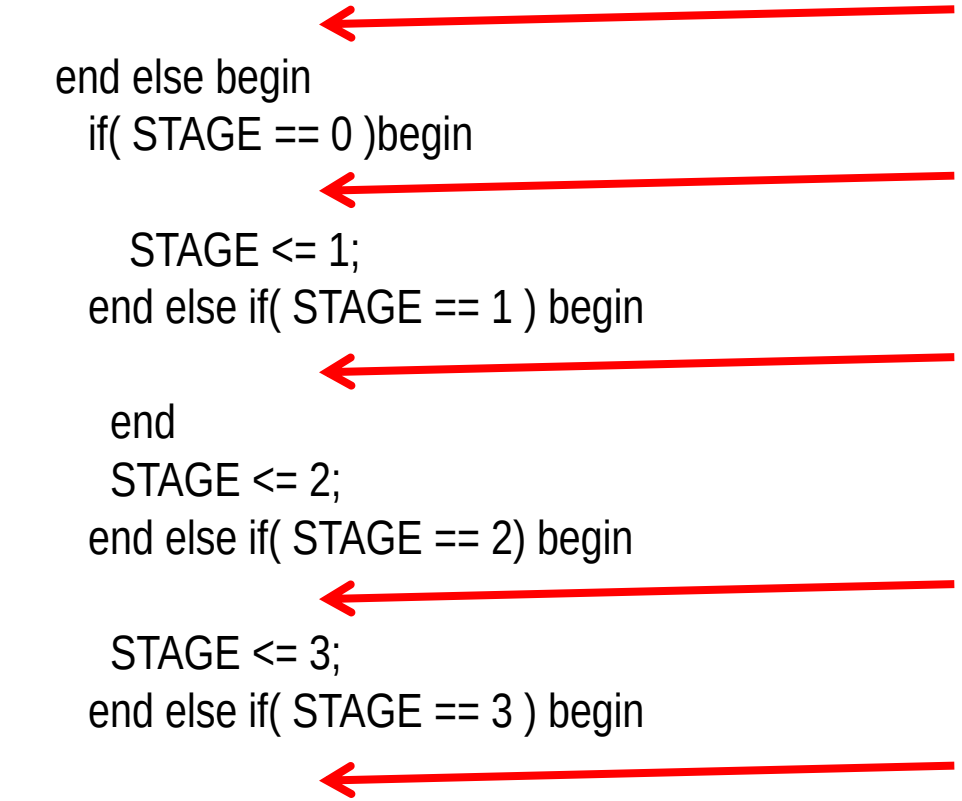
- 1クロック目：命令フェッチ
  - 命令アドレスの番地から命令を取り込む
- 2クロック目：命令デコード、レジスタ読み出し
  - 命令のOP2，OP3のレジスタを読み出しBUSA, BUSBへ
  - オペコード0xxxの場合に演算器の入力レジスタA, BにBUSA, BUSBの内容を取り込む
  - オペコード101xの場合にロードストアユニットの入力レジスタA, Bに BUSA, BUSBの内容を取り込む
  - オペコード1000の場合プログラムカウンタ入力レジスタPCiに BBUSの内容を取り込む
  - オペコード1001かつフラグレジスタが1の場合プログラムカウンタ入力レジスタPCiに BBUSの内容を取り込む
  - オペコードが100x以外の場合には、プログラムカウンタ入力レジスタPCiに PC + 1を取り込む
- 3クロック目：演算実行
  - オペコード0xxxの場合に、xxxに応じた演算結果を演算器出力レジスタFUcに取り込む
  - オペコード101xの場合、LSUAをデータアドレスに、x=1のとき RW=1,とし、データバスの結果をLSUCに取り込む、x=0のとき、RW=0とし、データバスにLSUAを出力する
  - オペコード1000のときPC出力レジスタPCcにPC+1値を取り込む
- 4クロック目：書き込み
  - オペコード0xxxの場合に、演算器出力レジスタ値をCBUSに出力する
  - オペコード101xの場合、LSU出力レジスタ値FUcをCBUSに出力する
  - オペコード1100の場合、即値データIMMをCBUSに出力する（ただし上位8ビットは0とする）
  - オペコード1000のときPC出力レジスタ値PCcをCBUSに出力する
  - CBUS値をOP1のレジスタに書き込む
  - PCにプログラムカウンタ入力レジスタ値を書き込む

# 演習１０－１：CPUの動作の状態遷移図を描いてみよう

状態遷移図からVerilogHDLのひな型ができるはず・・・

```
module CPU(CK,RST,IA,ID,DA,DD,RW);
  input CK,RST;
  input [15:0] ID;
  output RW;
  output [15:0] IA,DA;
  inout [15:0] DD;

  reg [1:0] STAGE;
```

```
always @(posedge CK)  begin
  if( RST == 1 ) begin
  end else begin
    if( STAGE == 0 )begin
      STAGE <= 1;
    end else if( STAGE == 1 ) begin
      end
      STAGE <= 2;
    end else if( STAGE == 2) begin
      STAGE <= 3;
    end else if( STAGE == 3 ) begin
      STAGE <= 0;
    end
  end
end
endmodule
```



# 継続代入

プログラムカウンタの出力はIA(命令アドレス)として外部に出力

オペコードは命令語(INST)の15-12ビット目
オペランド1は命令語の11-8ビット目
オペランド2は命令語の7-4ビット目
オペランド3は命令語の3-0ビット目
即値データは命令語の7-0ビット目

バスAには常にOPR2で指示されるレジスタファイルの値を出力
バスBには常にOPR3で指示されるレジスタファイルの値を出力

```
assign IA = PC;
assign OPCODE = INST[15:12];
assign OPR1 = INST[11:8];
assign OPR2 = INST[7:4];
assign OPR3 = INST[3:0];
assign IMM = INST[7:0];
assign BUSA = RF[OPR2];
assign BUSB = RF[OPR3];
```

# LSU・バス周りの制御

３クロック目
オペコード101xの場合、
**x=1(Load)のとき RW=1,とし**、データバスの結果をLSUCに取り込む、
**x=0(Store)のとき、RW=0とし**、データバスにLSUAを出力する

４クロック目
　オペコード0xxxの場合に、演算器出力レジスタ値をCBUSに出力する
　オペコード101xの場合、LSU出力レジスタ値FUcをCBUSに出力する
　オペコード1100の場合、即値データIMMをCBUSに出力する（ただし上位8ビットは0とする）
　オペコード1000のときPC出力レジスタ値PCcをCBUSに出力する

```
assign DD =(RW==0? LSUA : 'b Z);
assign DA = LSUB;
assign CBUS = (OPCODE[3]==0 ? FUC : (OPCODE[3:1]=='b 101 ? LSUC :
      (OPCODE=='b 1100 ? {8'b 0,IMM} : OPCODE=='b 1000 ? PCC : 'b z)));
```

# 手続き代入：１，２クロック目

１クロック目
命令アドレスの番地から命令（ID）を取り込む（INSTに）

```
INST <= ID;
```

２クロック目
オペコード0xxxの場合に演算器の入力レジスタA, BにBUSA, BUSBの内容を取り込む

```
if( OPCODE[3] == 0 ) begin
        FUA <= ABUS; FUB <= BBUS;
end
```

オペコード101xの場合にロードストアユニットの入力レジスタA, Bに BUSA, BUSBの内容を取り込む

```
if( OPCODE[2:1] == 'b01) begin
        LSUA <= ABUS; LSUB <= BBUS;
end
```

オペコード1000の場合プログラムカウンタ入力レジスタPCiに BBUSの内容を取り込む
オペコード1001かつフラグレジスタが1の場合プログラムカウンタ入力レジスタPCiに BBUSの内容を取り込む
オペコードが100x以外の場合には、プログラムカウンタ入力レジスタPCiに PC + 1を取り込む

```
if( (OPCODE[3:0] == 'b 1000) ||
(OPCODE[3:0] == 'b 1001 && FLAG == 1 ) )
                PCI <= BBUS;
else            PCI <= PC + 1;
```

# 手続き代入：３･４クロック目

３クロック目
オペコード0xxxの場合に、xxxに応じた演算結果を演算器出力レジスタFUcに取り込む

```
if( OPCODE[3] == 0 ) begin
    case (OPCODE[2:0])
    'b 0000 :  FUC <= FUA + FUB;
    'b 0001 :  FUC <= FUA - FUB;


    endcase
end
```

オペコード101xの場合
**x=1のとき RW=1**,とし、データバスの結果をLSUCに取り込む
**x=0のとき、RW=0とし**、データバスにLSUAを出力する
オペコード1000のときPC出力レジスタPCcにPC+1値を取り込む

```
if( OPCODE[3:1] == 'b 101 ) begin
     if( OPCODE[0] == 0 ) begin
          RW <= 0;  LSUC <= DD;
      end else begin
           RW <= 1;
      end
end

if( OPCODE[3:0] == 'b 1000 )
          PCC <= PC;
```

４クロック目
CBUS値をOP１のレジスタに書き込む
PCにプログラムカウンタ入力レジスタ値を書き込む

```
RF[OPR1] <= CBUS;
PC <= PCI;
```

# 演習10－2：CPUの完成

cpu.vを完成させ simcpuで動作を確認する

```
module simcpu;
reg CK, RST;
wire RW;
wire [15:0] IA, DA, DD;
reg [15:0] ID, DDi;
reg [15:0] IMEM [0:127], DMEM[0:127];
CPU c(CK,RST,IA,ID,DA,DD,RW);
assign DD = ( RW == 1 ? DDi : 'b Z);
initial begin
        $dumpfile("cpu.vcd");
        $dumpvars;
                CK = 0;
                RST = 0;
#5              RST = 1;
#100            RST = 0;
        #10000 $finish;
end
always @(negedge CK) begin
                ID = IMEM[IA];
end
always @(negedge CK) begin
                if( RW==1 )DDi = DMEM[DA]; else DMEM[DA]=DD;
end
initial begin
IMEM[0]='b 1100_0000_0000_0000;          // IMM  R0, [0]
IMEM[1]='b 1100_0001_0000_0001; // IMM  R1, [1]
IMEM[2]='b 1100_0010_0000_0010; // IMM  R2, [2]
IMEM[3]='b 1100_0011_0000_0011; // IMM  R3, [3]
IMEM[4]='b 1100_0100_0000_0100; // IMM  R4, [4]
IMEM[5]='b 0000_0101_0001_0011; // ADD  R5, R1, R3
IMEM[6]='b 1010_0000_0101_0000; // ST  R5, R0
end
always #10 CK = ~CK;
endmodule
```

# 演習10－3：CPUで計算させる

- 1～10までの整数を足し算し結果をデータメモリの0番地に出力するプログラム
  - R1：アキュムレータ
  - R2：加算する数（初期値1）
  - R3：加数（1）
  - R4：ループ回数（9）
  - R5：ジャンプアドレス(12)
  - R6：ジャンプアドレス(7)
  - R7：出力メモリアドレス（0）

# 演習１０－４：CPUで乗算の実行

- データメモリ0番地のデータと1番地のデータを掛け算して2番地に格納

# 演習１０－５：CPUのパイプライン化

- パイプライン化について検討する