

2020年4月10日

ハードウェア設計論:2

# ハードウェアにおける設計表現 ハードウェア設計記述言語VerilogHDL

VerilogHDLの基本構文

代入、条件分岐、時間の表現、状態遷移と順序機械

TA2名: 中山、池田(健)が担当します

質問は随時はSLACK #2020s-ハードウェア設計論

**verilog**が実行できる状態にしておいてください。

<http://www.mos.t.u-tokyo.ac.jp/~ikeda/HWDesign/>

# 本日の課題(目標)

本日の課題:

目標1: 課題1を終えること

——本日講義時間内に課題1が終了することを目指す

目標2: 課題2・3を終えること

——次回4月17日までに課題2を終えておいていただく

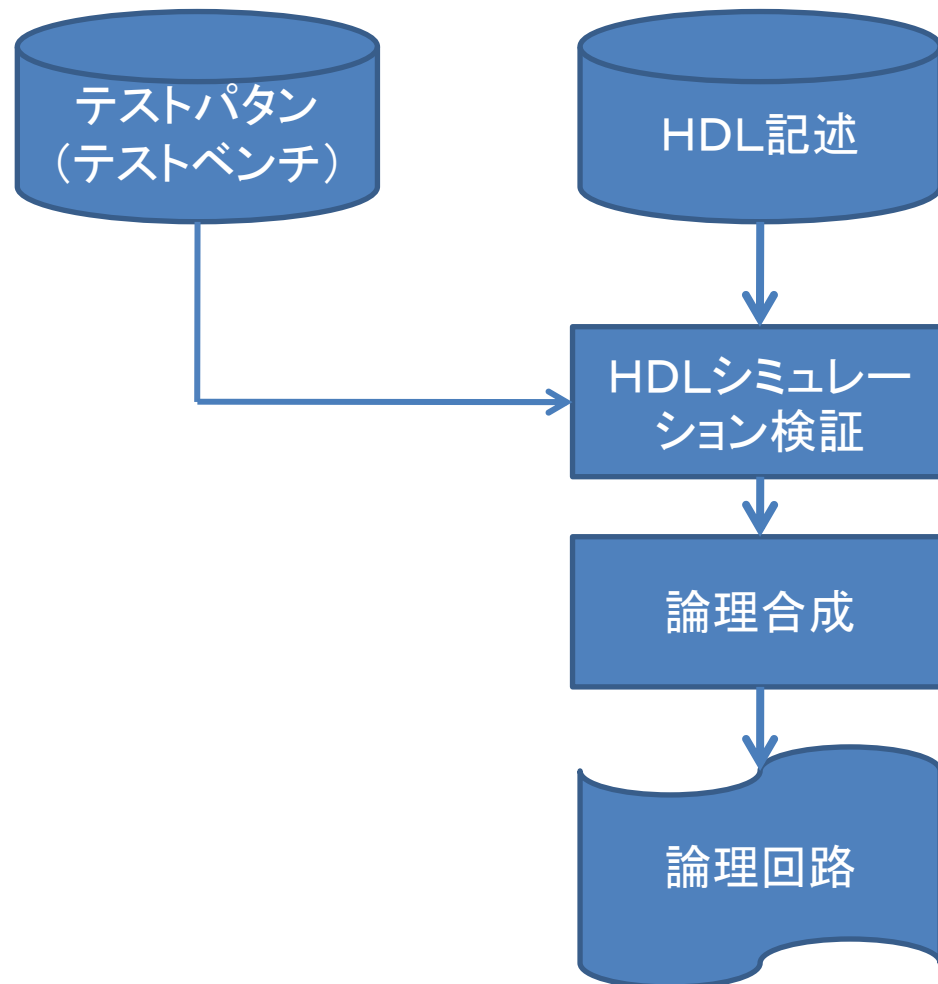
# ハードウェア記述言語

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

# ・・・なぜハードウェア記述言語・・・

- 言語の文法そのものは、ほぼソフト
- 大きな違いは「時間」の概念
  - すべては同時に動作する(ステップごとではない)
  - 時間を制御することが出来る
  - 変数が、記憶(不揮発性)と揮発性(配線)に区別
- 並列ソフトウェアと本質的には同じ
  - VerilogHDLのオリジンはProlog(並列化志向の言語)

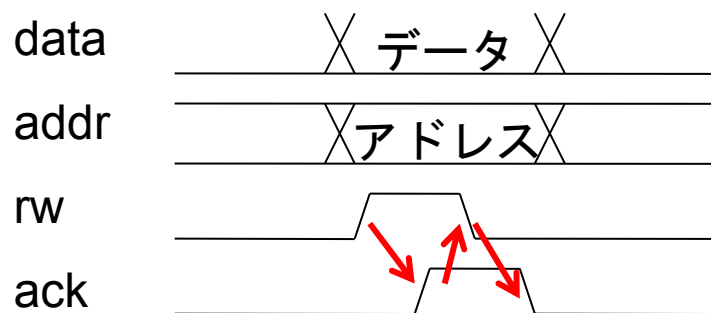
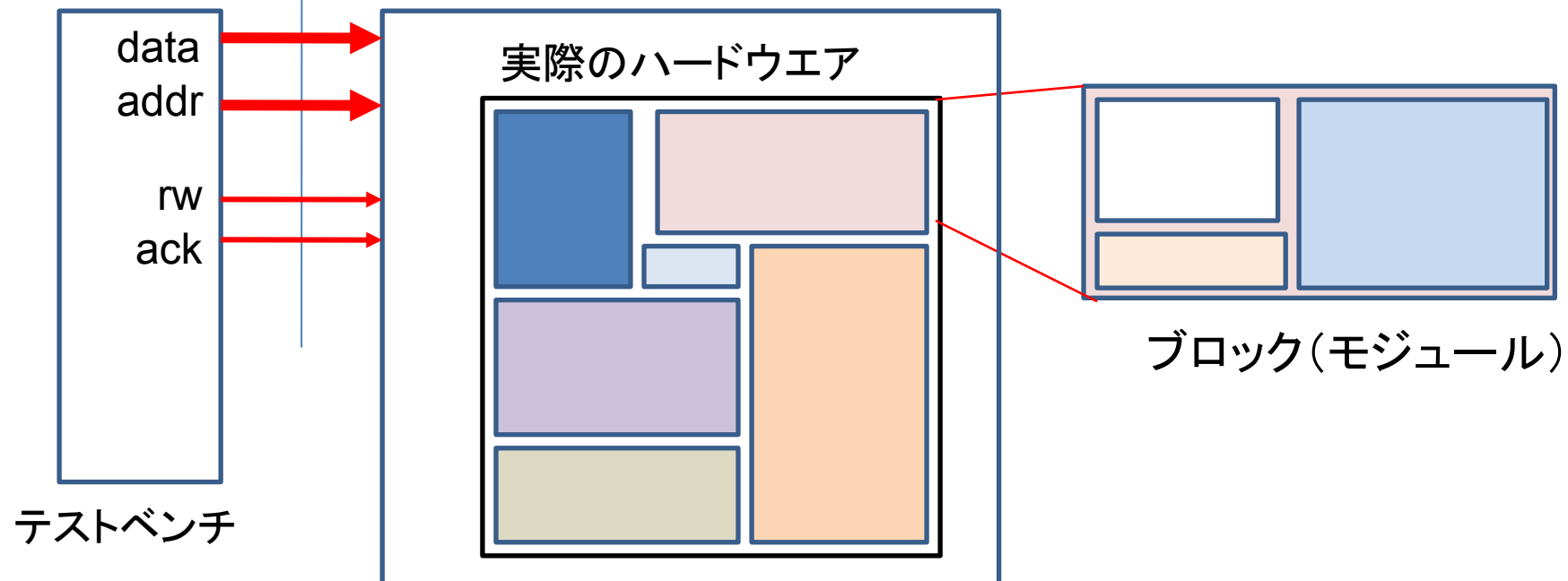
# HDLによる設計



- 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

# HDLの基本

入出力を生成



入出力信号とタイミング

# 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
```

モジュールとは「機能」もしくは「構造」のまとまり。プログラミングの関数\*のようなもの。全ての記述は module ~ endmoduleで囲う。

構文は セミicolon ; 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
```

\*ただしmodule内に function, taskといった手続きを記述することが可能であるのでmoduleを「関数」と表現するのは適切ではない

# 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  
(変数の宣言、代入、プロセス  
文の前におかなくてはならな  
い)

継続代入文  
(プロセス文中(後述)に  
入れてはならない)

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)

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

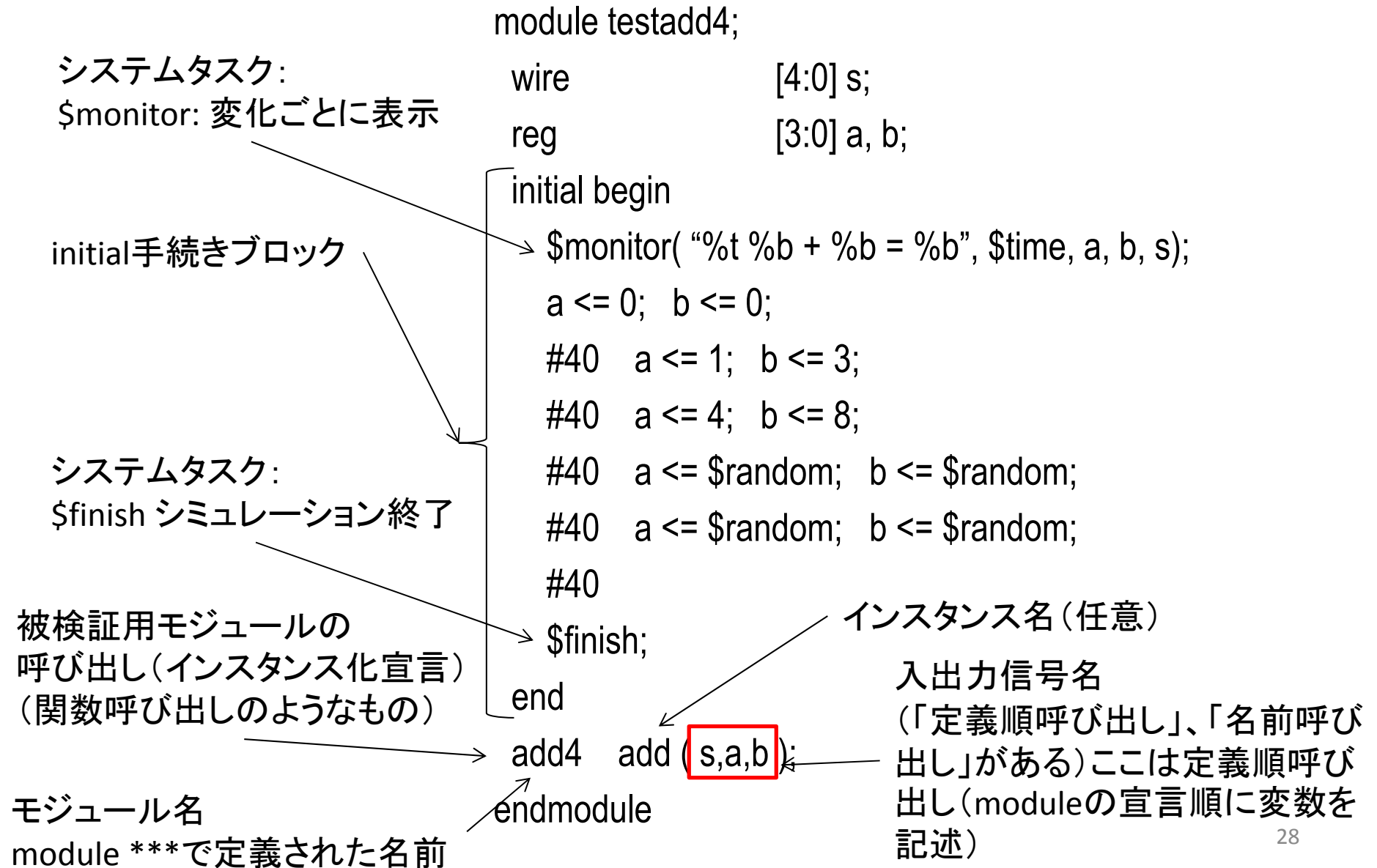


# テストベンチを作成

- 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



# テストベンチ

再度実行:

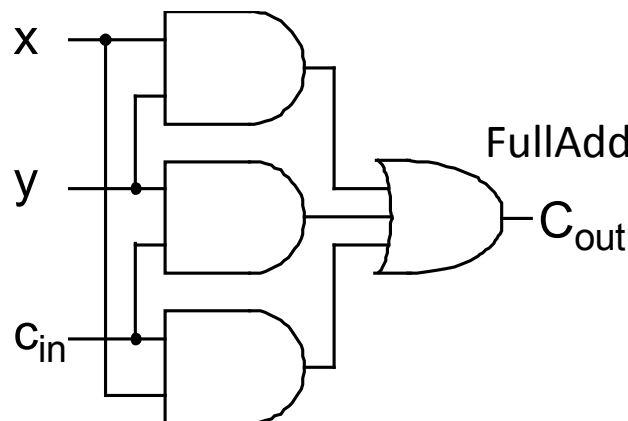
```
% iverilog testadd41.v add4.v
```

```
% ./a.out
```

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

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	関係演算	!==	ケース不等号(X,Zも不一致)	6
		ビット毎のOR	8		<	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				

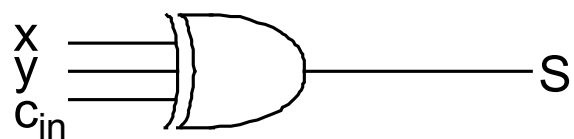
# 簡単な論理式を実現してみよう



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
```

論理式やプロセス文、手続き文を用いた記述を  
「**動作記述**」と呼ぶ

# 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: **組み合わせ回路の「値」**として用いる

→ 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: **順序機械の状態**として用いる

→ 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

ネット型  
レジスタ型

- 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(組み合わせ回路)」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
- 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)
- 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
- 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



# 式・数の表現

- 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  $a[16:13]$

# VerilogHDLの基本構文：構造記述と組み合わせ回路

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:

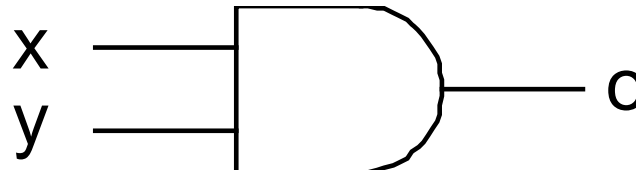
定義順呼び出し:

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
```



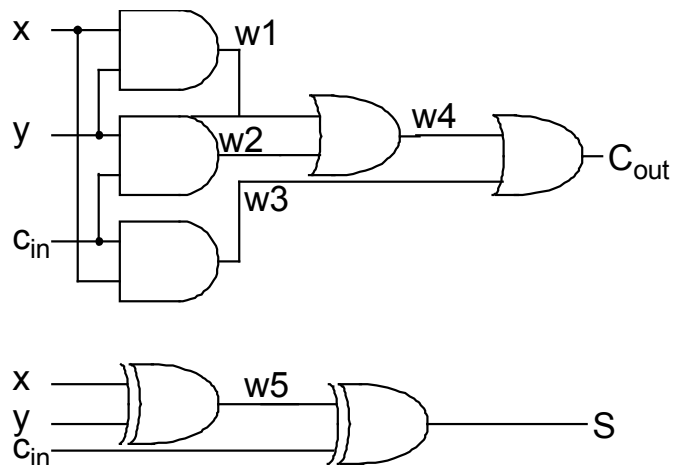
# Verilogで定義されるプリミティブゲート

Verilog-HDLにあらかじめ組み込まれているゲート。module定義なしに使用可能。  
通常、ポートは 出力、入力、イネーブルの順となっている

(プリミティブゲートも moduleとして定義されている)

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, if0: control=0で出力, 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
```

# 課題1

- 課題1－1:動作記述(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
- 課題1－2: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
- FullAdderFunction.v, FullAdderStructure.vを  
**<http://www.mos.t.u-tokyo.ac.jp/~ikeda/HWDesign/>**  
から提出

simfulladd.v

# 課題1のテストベンチ

```
module simfulladd;
  wire      s, cout;
  reg       x, y, cin;
  initial begin
    $monitor( "%t In (x, y, cin) -> Out (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;
    ..... 入力全条件を検証すること
    #40 x <= 1; y <= 1; cin <= 1;
    #40
    $finish;
  end
  FullAdderFunction add ( x, y, cin, cout, s );
endmodule
```

FullAdderStructureも同様

いやいや、、設計といえばアナログで  
しょ。。。デジタルなんて、、

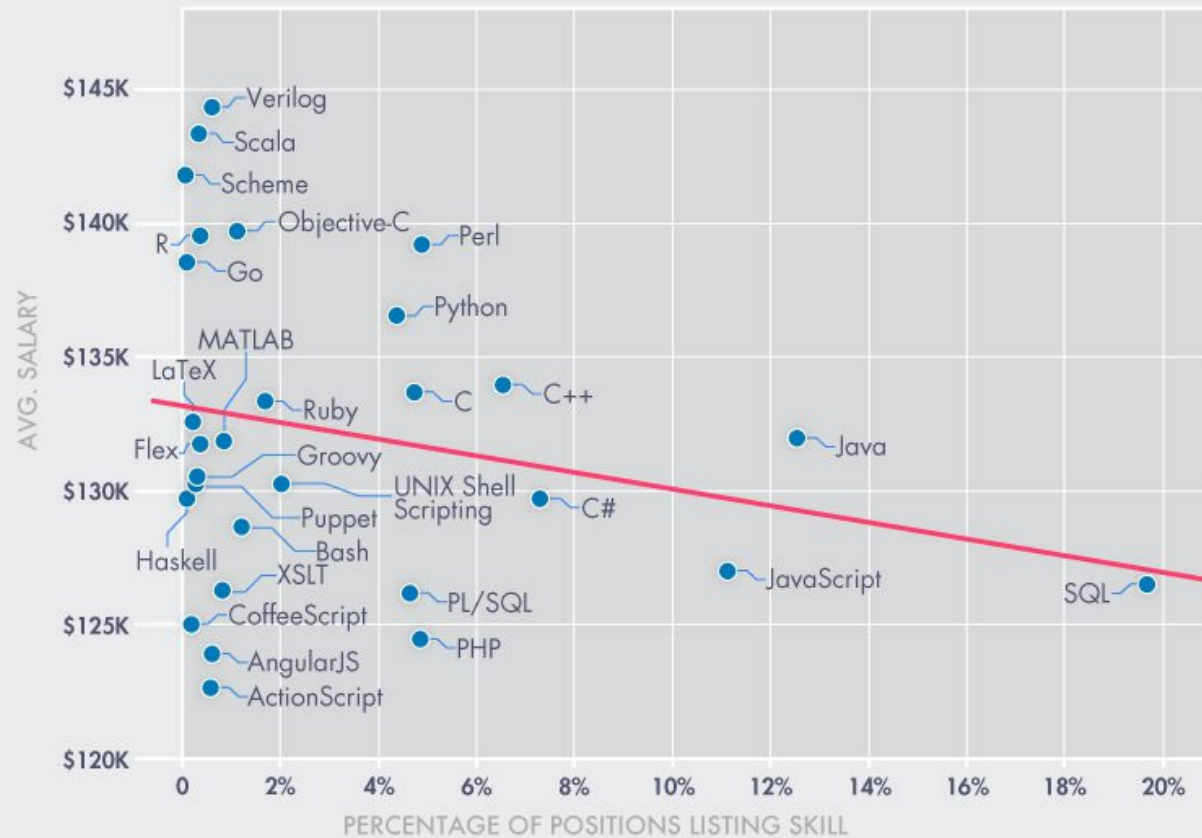
# なぜVerilogHDL?

- もっとも稼げるプログラミング言語は・・・



# MOST VALUABLE PROGRAMMING LANGUAGES

BY AVERAGE SALARY AND PERCENTAGE OF POSITIONS LISTING SKILL



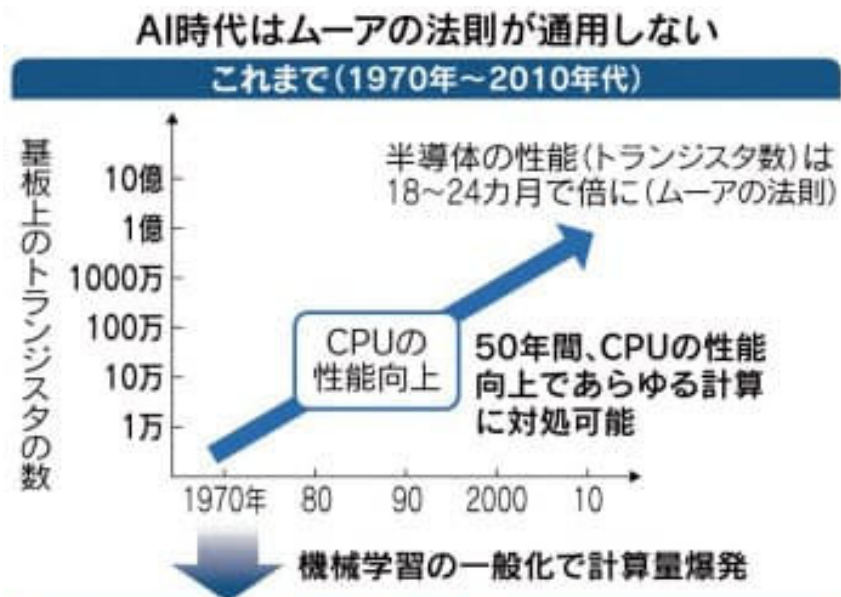
Source: [Paysa.com](https://paysa.com)



# なぜVerilogHDL?

- もっとも稼げるプログラミング言語は……
- 理由：需給のバランス
  - 希少価値の高さ：出来る人がほとんどいない  
ソフトははいて捨てるくらい書ける人がいる
  - 潜在需要の高さ：超高速トレーディングでMy Hardwareを駆使するトレーダが多数、そういった（お金が回る）需要がある。さらには、Google, Apple, Amazon, Facebookが本気で専用のハードウェアの開発のための人材を囲っている
  - PFNもソフトを捨ててハードに傾注

# AIはハードウェアを活用する時代



日本の人工知能(AI)分野をけん引するスタートアップ企業、プリファード・ネットワークス(東京・千代田)が事業の軸足をソフトウェアからハードウェアに移す。…深層学習の基盤ソフト「チェイナー」は、…開発を終え…自社用の半導体チップに力を入れる

「チェイナーは役目を終えた」 by秋葉拓哉執行役員

日本経済新聞 2020/3/23付

汎用と専用それぞれの半導体を理解し、使い分けられる人材はまだ少ない。エヌビディアなど米企業はこの問題に対応するため、ソフト人材の陣容強化を足元で進めている。シリコンバレーで人件費が高騰している理由のひとつには、こうした専門人材の囲い込みがあるとの指摘もある。

日本経済新聞 2020/3/31付

# 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
```

# 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
```

本来は初期化が必要Reset=1  
のとき初期化だとすると

```
if (Reset==1) q<=0;  
else    q <= q+1;
```

# Verilogと時間・・・

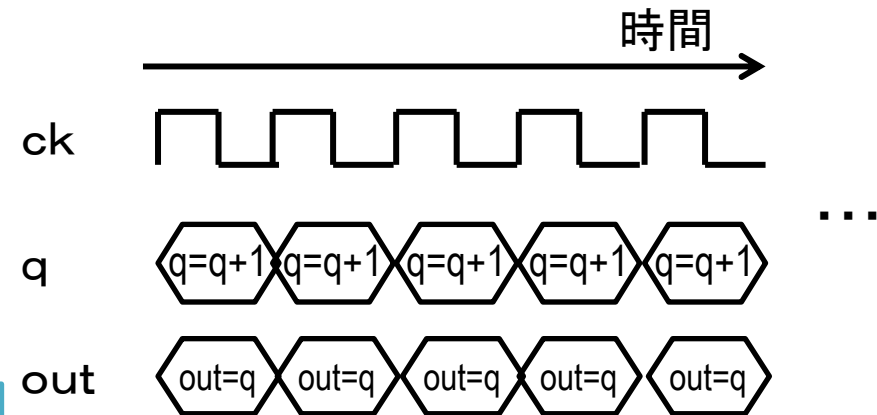
```
module count4(out,ck);  
  output  [3:0] out;  
  input    ck;  
  reg      [3:0] q;
```

always構文は@内の  
条件が満たされるた  
びに実行される

```
  always @(posedge ck) begin  
    q <= q+1;  
  end
```

```
  assign out = q;  
endmodule
```

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)





count4r.v

# VerilogHDLの基本構文

全ての構文は

module / endmodule  
で囲む

→ **module count4r(out,ck,res);**

← 全ての構文は  
セミコロンで区  
切る

**output [3:0] out;**

**input ck, res;**

構文が複数行になる場  
合には

**reg [3:0] q;**

begin / end  
で囲う

if else 構文

**always @(posedge ck or negedge res) begin**

→ **if( !res) q<= 0;**

**else q <= q+1;**

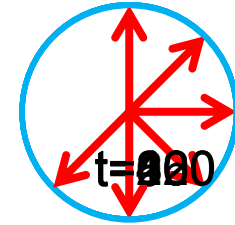
← res=0のとき qに0を代入  
(リセット)

**end**

**assign out = q;**

**endmodule**

# 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

```
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 ); 0 0000 + 0000 = 00000
```

```
endmodule          40 0001 + 0011 = 00100
```

```
                   80 0100 + 1000 = 01100
```

```
                  120 0100 + 0001 = 00101
```

```
                  160 1001 + 0011 = 01100
```

時間経過を表示: 表示がない場合には 0(δ時間)

# VerilogHDLの基本構文:代入

## • 代入の時間関係

## ブロッキング代入

## ノンブロッキング代入

```
assign C=A;  
assign D=B;  
always @posedge ck
```

```
assign C=A;  
assign D=B;  
always @posedge ck
```

1

2

**A=A+B;**

**B=A+1;**

end

1の後に2が  
実行

end

1と2は同時に実行

ck

**A**

# B

C

D

$A_0$	$A_1 = A_0 + B_0$	$A_2 = A_1 + B_1$
-------	-------------------	-------------------

$B_0$	$B_1 = A_1 + 1$	$B_2 = A_2 + 1$
-------	-----------------	-----------------

$A_0$	$A_1$	$A_2$
-------	-------	-------

$B_0$	$B_1$	$B_2$
-------	-------	-------

ck

A

# B

C

D

$A_0$	$A_1 = A_0 + B_0$	$A_2 = A_1 + B_1$
-------	-------------------	-------------------

$B_0$	$B_1 = A_0 + 1$	$B_2 = A_1 + 1$
-------	-----------------	-----------------

$A_0$	$A_1$	$A_2$
-------	-------	-------

$B_0$	$B_1$	$B_2$
-------	-------	-------

## ➡ 手続代入

## 繼續代入

ハードウェアの  
記述では必ず  
ンブロッキング  
代入を使用する

テストベンチの  
記述にはブロッ  
キング代入を用  
いてもよい

# テストベンチ

testcount4.v

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

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)

```
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
```

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 xxxxx  
10 1 xxxxx  
20 0 xxxxx  
30 1 xxxxx
```

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)

??????

リセットしなくてはレジスタ型変数の値が不定

# テストベンチ

testcount4r.v

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

```
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
```

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
```

GUI出力用のテストベンチ

```
module testcount4rgui;
```

```
    wire            [3:0] out;
```

GUI表示用データの出力制御

```
    reg            ck, res;
```

```
    initial begin
```

```
        $dumpfile( "count4.vcd" );
```

```
        $dumpvars;
```

システムタスク:

\$monitor: 変化ごとに表示

```
        $monitor( "%t %b %b %b", $time, ck, res,
```

```
        out);
```

```
        ck<=0;
```

```
        res<=0;
```

```
        #40
```

```
        res <= 1;
```

```
        #350
```

```
        $finish;
```

```
    end
```

システムタスク:

\$finish シミュレーション終了

```
    always #10 ck <= ~ck;
```

被検証用モジュールの  
呼び出し(インスタンス化宣言)

```
    count4r cnt ( out, ck, res );
```

モジュール名

```
endmodule
```

入出力信号名

インスタンス名  
(任意)

# 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 ; が不在  
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.

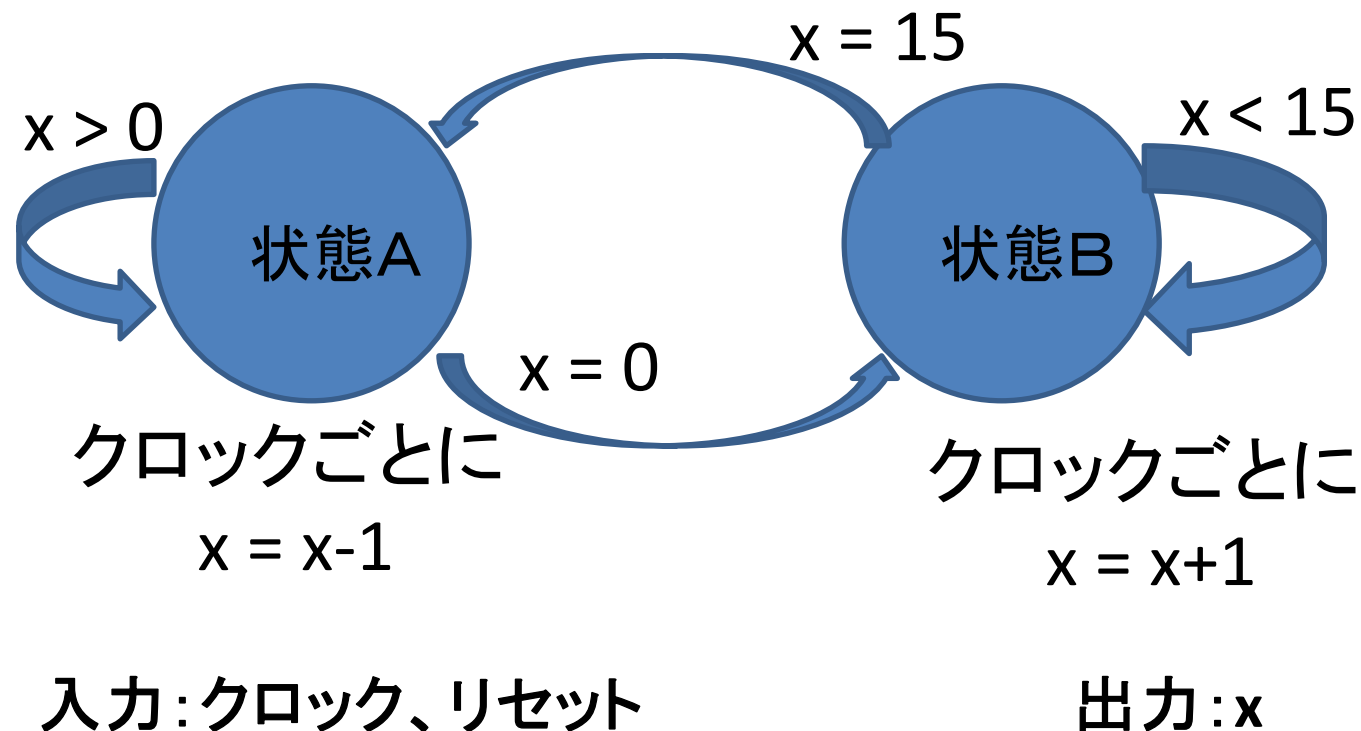
1 error(s) during elaboration.

入力ポートに対して reg 定義をしようとしている



# 順序機械の実現

- 簡単な状態機械を実現してみよう



# 状態の定義

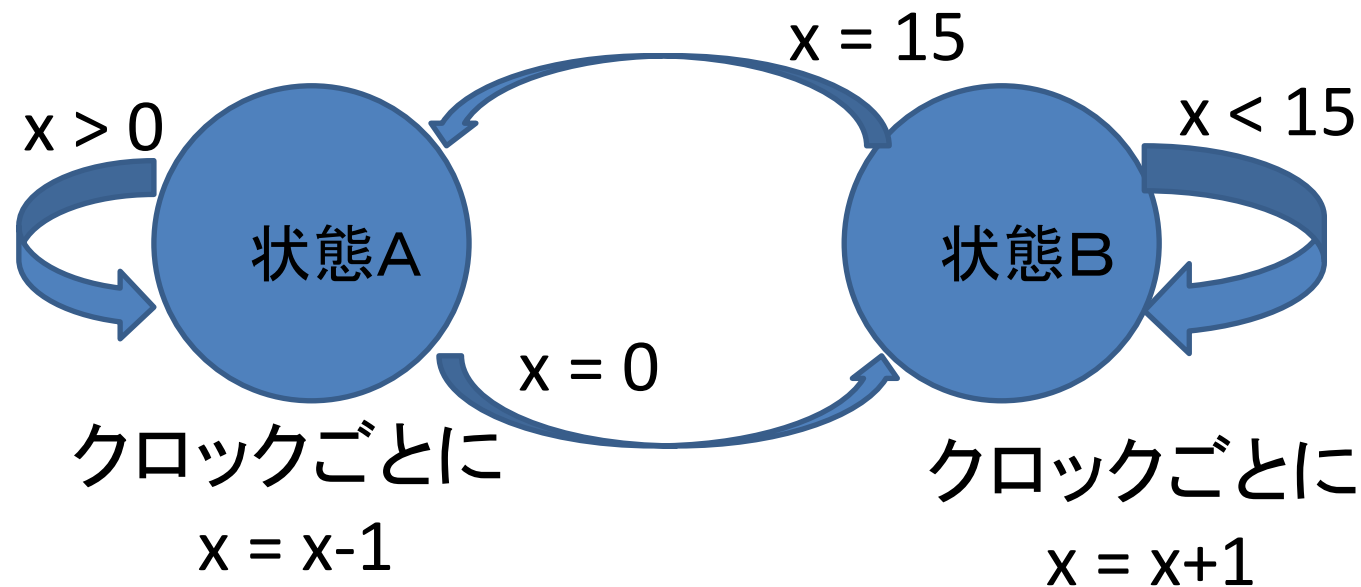
状態変数:  $st$ ,

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:  $x=0$



## 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
end
```

## 全体

```
module stm(ck,rst,x);
input  ck,rst;
output [3:0] x;
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でくる

## simstm.v

テストベンチ

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

stm    s(ck,rst,x);

endmodule
```

↑  
モジュール内  
変数の参照

# 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

.....

ck = 0, st = 1: x=e

ck = 1, st = 1: x=f

ck = 0, st = 1: x=f

ck = 1, st = 0: x=f

ck = 0, st = 0: x=f

ck = 1, st = 0: x=e

.....

ck = 0, st = 0: x=1

ck = 1, st = 0: x=0

ck = 0, st = 0: x=0

ck = 1, st = 1: x=0

ck = 0, st = 1: x=0

ck = 1, st = 1: x=1

.....

初期化されるまでは値は x を持つ

x=f の時は状態のみを遷移させる

状態遷移後、次のクロックからxの減算が始まる

x=0 の時は状態のみを遷移させる

状態遷移後、次のクロックからxの加算が始まる

## simstm.v

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

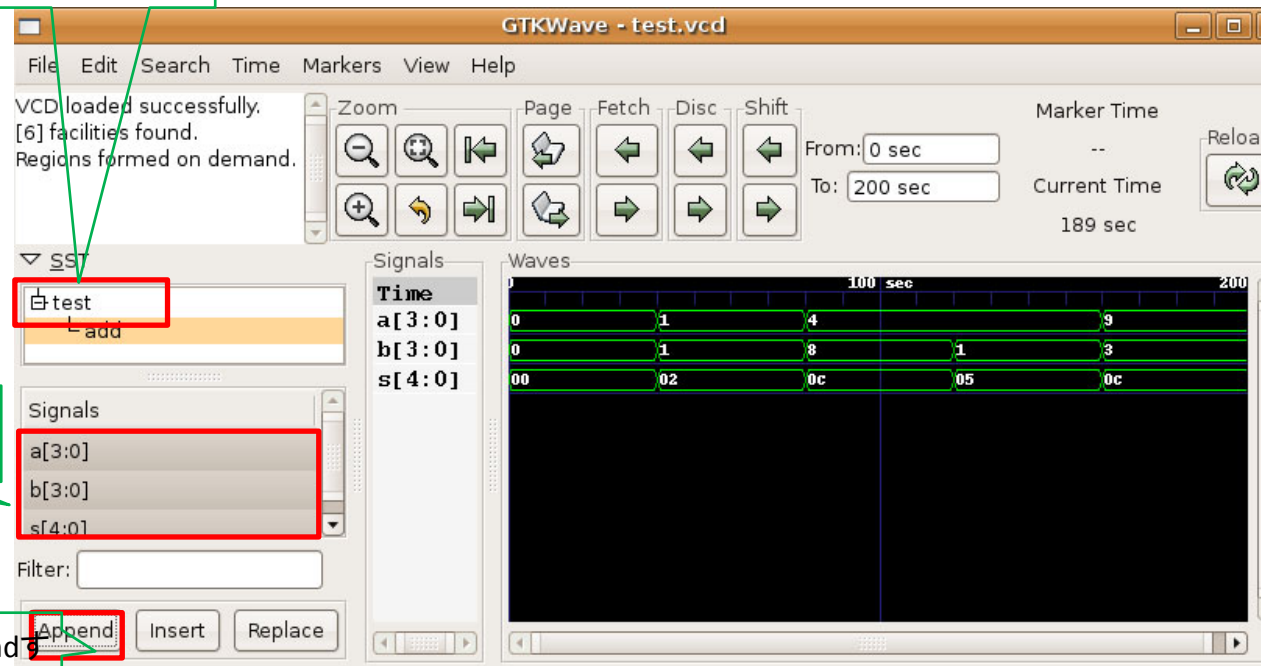
# グラフィカルに見たい

```
% iverilog simstm.v stm.v  
% ./a.out  
% gtkwave stm.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



# 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 );
```

ハードウェアとしては、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



# 演習2

- 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2-1 sub4.v

課題2-2 mul4.v

# 演習2のヒント

```
module sub4(s,a,b);
```

```
  output  [4:0] s;
```

```
  input   [3:0] a,b;
```



```
endmodule
```

ここを埋めてください

```
module mul4(s,a,b);
```

```
  output  [7:0] s;
```

```
  input   [3:0] a,b;
```



```
endmodule
```

結果の例

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

2の補数表現  
1,0を反転させて1を足す

結果の例

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

# 演習2のテストベンチの例

```
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
```

```
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
```

注意: ハードウェア記述そのものでは「ノンブロッキング代入を用いなくてはならないが、テストベンチ記述においては、「ノンブロッキング」、「ブロッキング」どちらを用いても結果に差は生じない

# 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
```

testadd43.v

# 全数をチェックしたければ

```
module testadd43;
  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;
  end
  always #10 ck = ~ck;
  always @(negedge ck) begin
    if( s != a + b ) begin
      $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
```

接続

% 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

# テストベンチ

```
simfulladd.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
```

# 演習3

- 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つつ加算するcount4r2.v, 2倍つつ乗算する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3-1 count4rs.v

課題3-2 count4r2.v

課題3-3 count4r2m.v

# 演習3の期待される結果

count4rs.v	count4r2.v	初期値を1(0以外)に しておく必要あり	count4r2m.v
0 0 0 0000	0 0 0 0000		0 0 0 0001
10 1 0 0000	10 1 0 0000		10 1 0 0001
20 0 0 0000	20 0 0 0000		20 0 0 0001
30 1 0 0000	30 1 0 0000		30 1 0 0001
40 0 1 0000	40 0 1 0000		40 0 1 0001
50 1 1 1111	50 1 1 0010		50 1 1 0010
60 0 1 1111	60 0 1 0010		60 0 1 0010
70 1 1 1110	70 1 1 0100	オーバーフローす ると以後0となる	70 1 1 0100
80 0 1 1110	80 0 1 0100		80 0 1 0100
90 1 1 1101	90 1 1 0110		90 1 1 1000
100 0 1 1101	100 0 1 0110		100 0 1 1000
			110 1 1 0000
370 1 1 1111	370 1 1 0010		120 0 1 0000
380 0 1 1111	380 0 1 0010		
390 1 1 1110	390 1 1 0100		390 1 1 0000