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）**

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

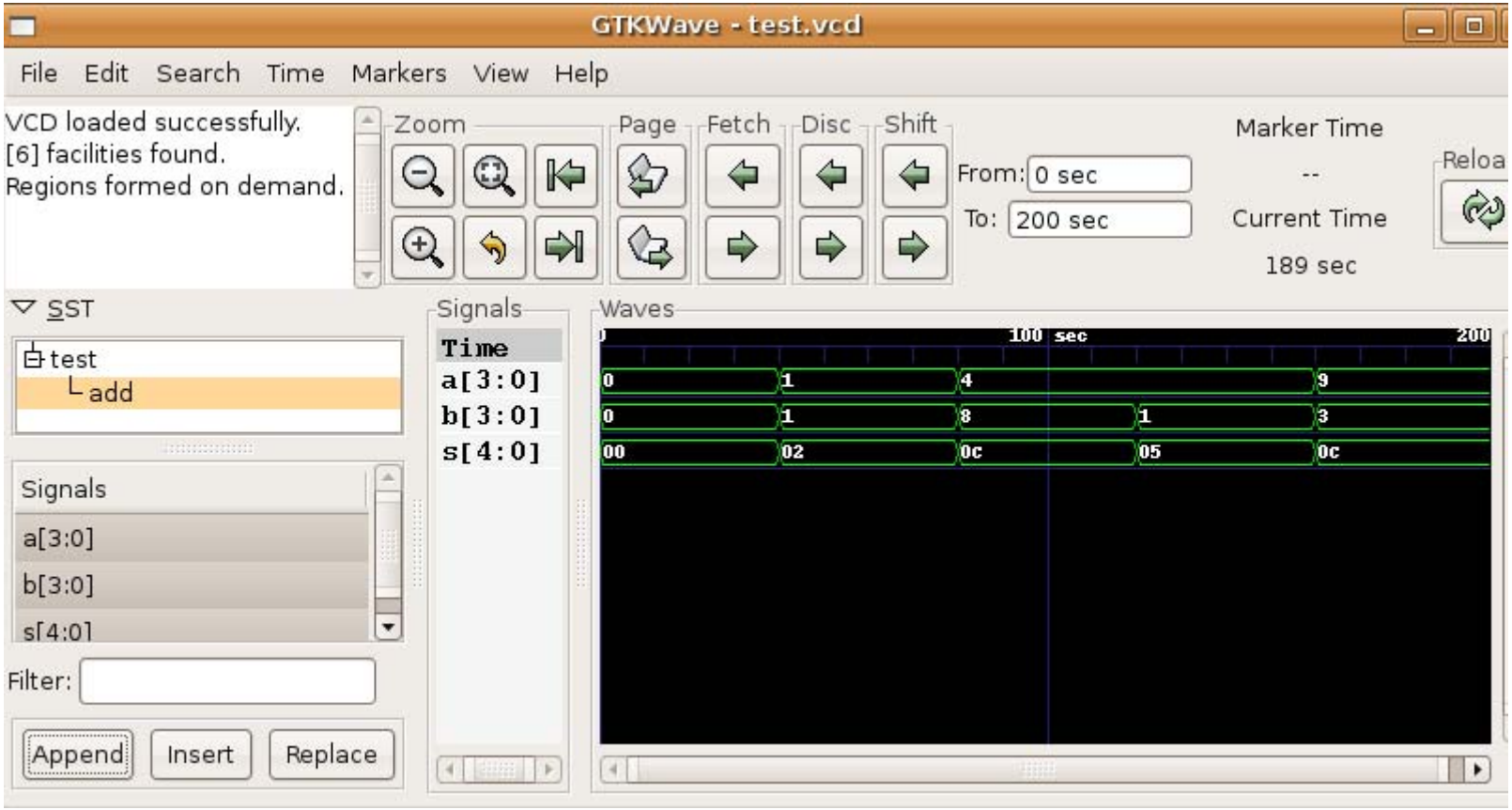
Verilogの実行確認

- 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

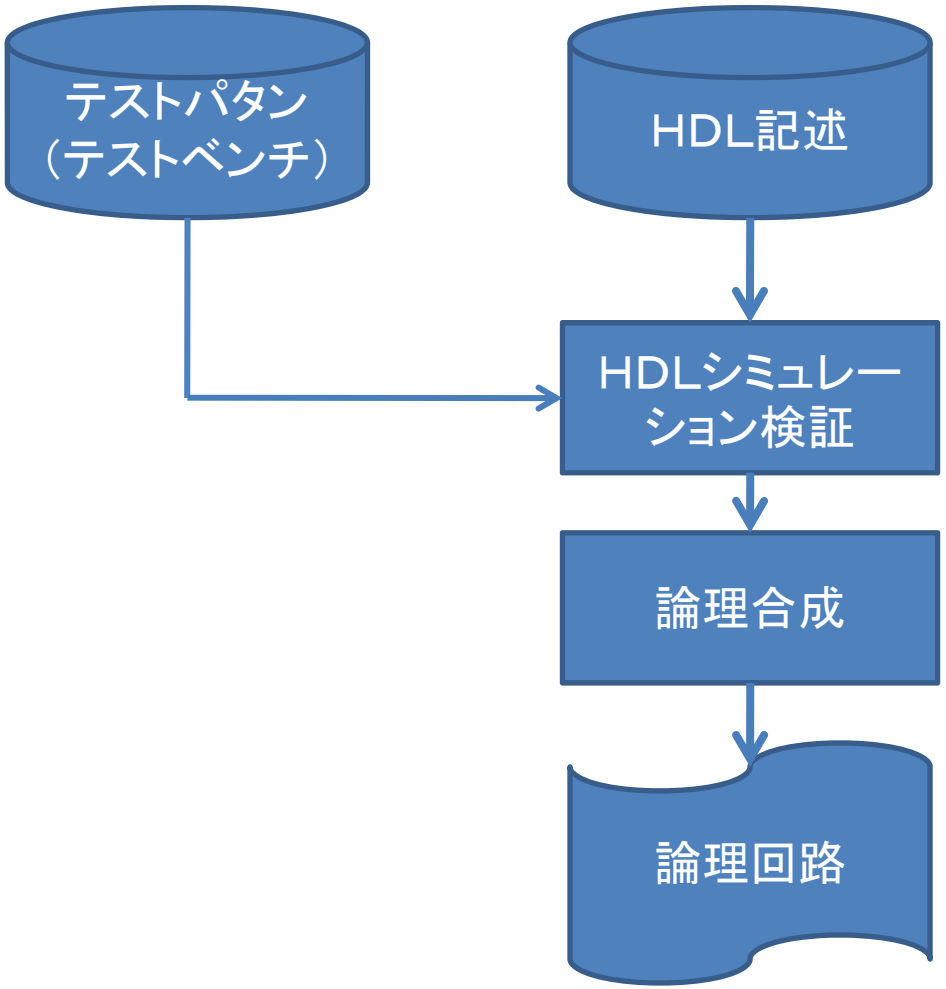
- % **gtkwave　test.vcd**



ハードウエア記述言語

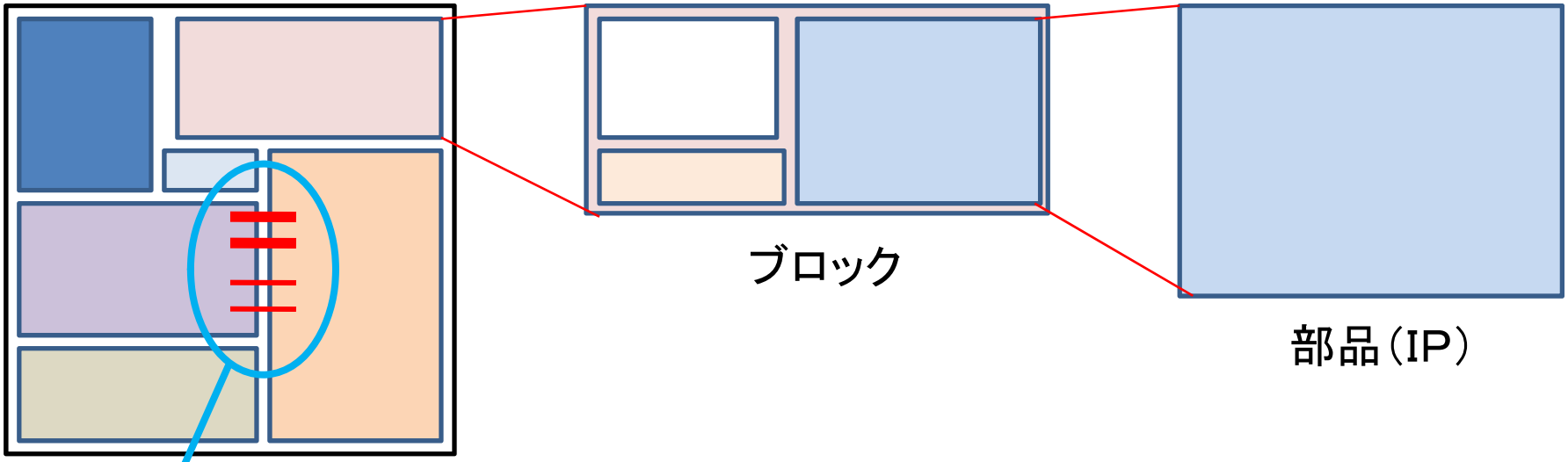
言語	VHDL	VerilogHDL	UDL／I	SFL
開発開始時期	1981	1984	1987	1981
開発組織	IEEE	Cadence	電子協	NTT
言語仕様公表	1987	1985	1990	1985
論理シミュレータ	有	有	有	有
論理合成系	有	有	有	有
規格の見直し	1993,2000, 2002, 2008	1995,2001, 2005	1992	なし

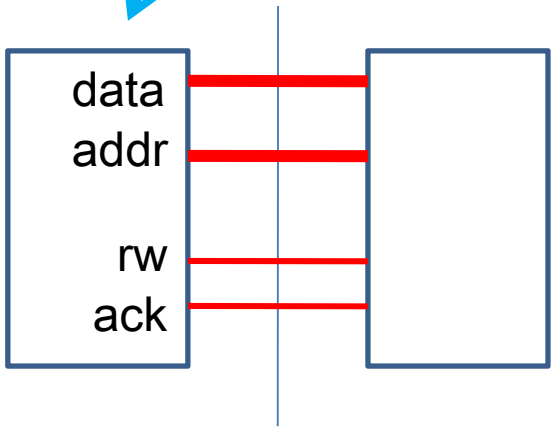
HDLによる設計

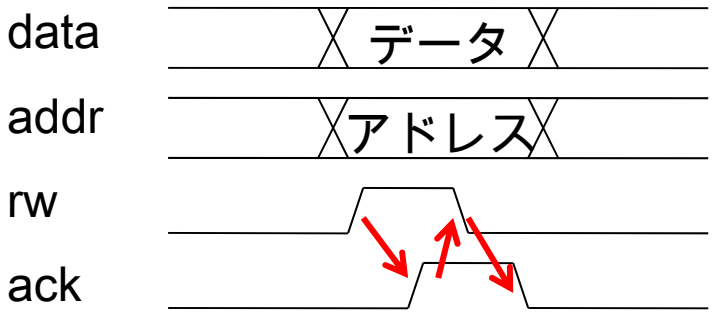


- 抽象度の高い記述による記述量の削減と設計効率向上
- ゲートレベル設計の自動化
- 設計の早期からのシミュレーションによる検証
- 制御回路の自動生成

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
```

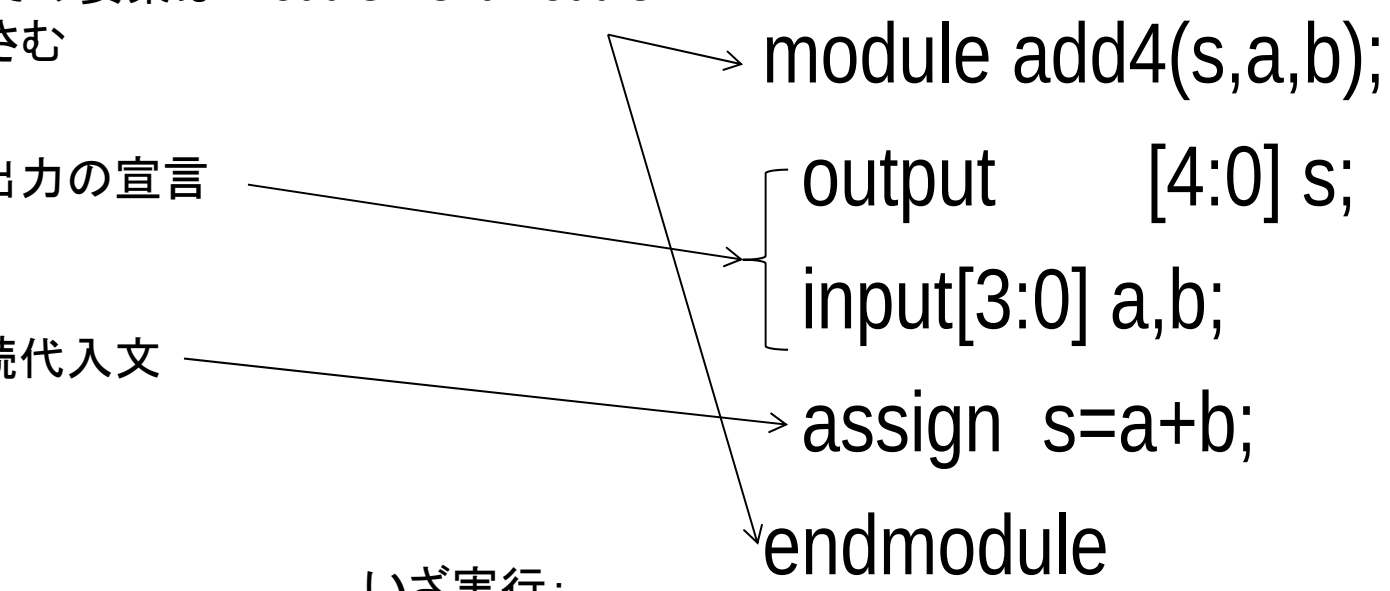
VerilogHDLの基本構文

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**

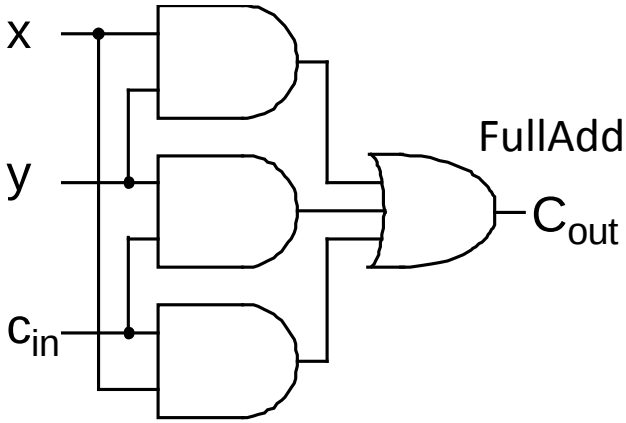
% **./a.out**

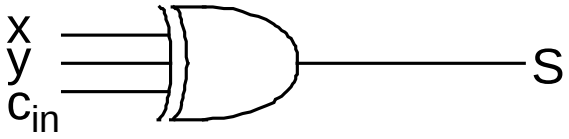
？？？

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 );
```

```
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

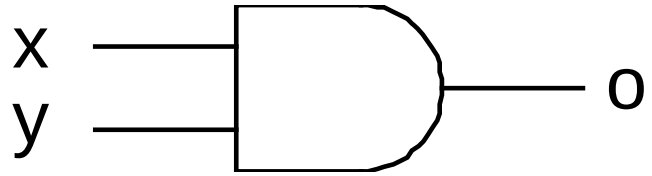
- wire型
  - 継続的代入されているときのみ値を保持する型。一般の配線と同様。通常は組み合わせ論理部を表現するのに使用
  - 1bitのwire型は定義を省略可能：ただしこの暗黙の定義は使わない方が賢明
- reg型
  - 任意ビット、記憶保持が可能な型（通常はFFなど状態、データを保存したいノードに対して使用）, signedを指定しない場合には符号なし
- signed指定
  - reg signed [7:0] a;　　aを符号付きレジスタ型として定義
  - wire signed [7:0] a;　aを符号付きwire型として定義
- integer型
  - 32bit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
```

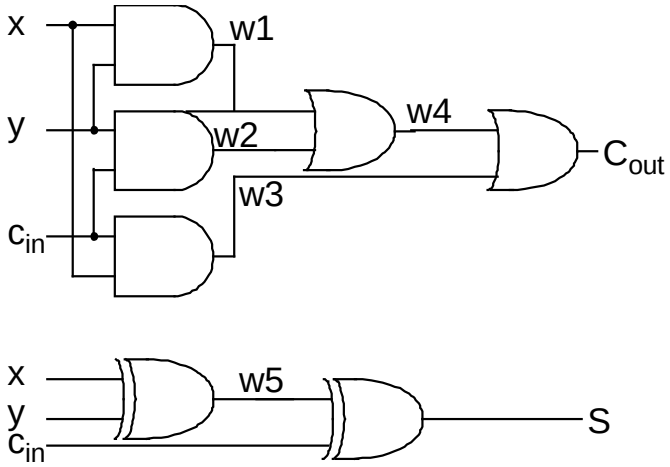


Verilogで定義されるプリミティブゲート

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, 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);

endmodule

基本構文：手続きと順序機械

4ビットカウンタ

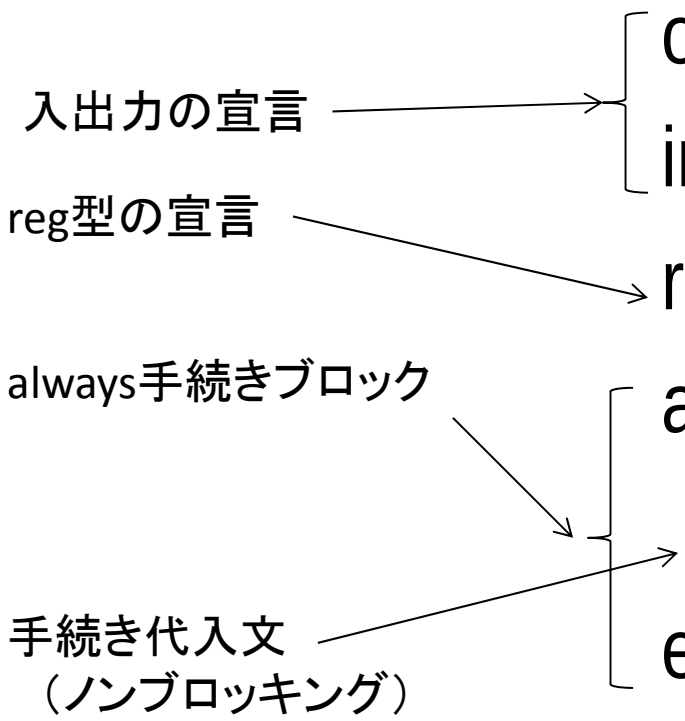
count4.v

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）



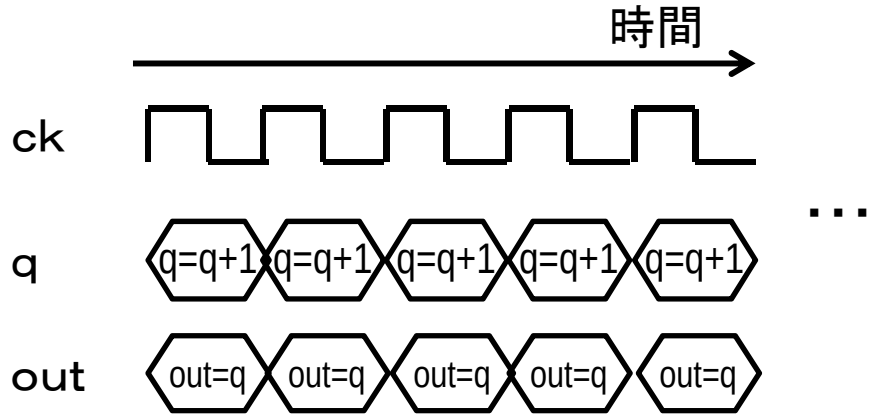
```
module  count4(out,ck);
  output      [3:0] out;
  input       ck;
  reg         [3:0] q;
  always @(posedge ck) begin
    q <= q+1;
  end
  assign out = q;
endmodule
```

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

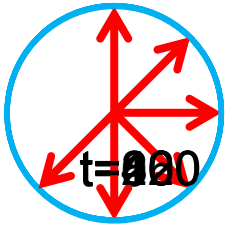
再度実行:

% **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

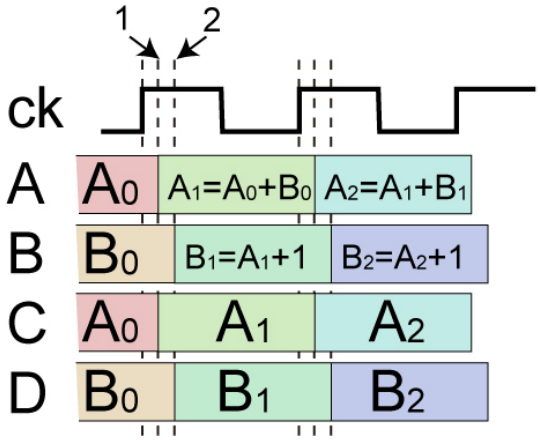
% iverilog testcount4r.v count4r.v
% ./a.out

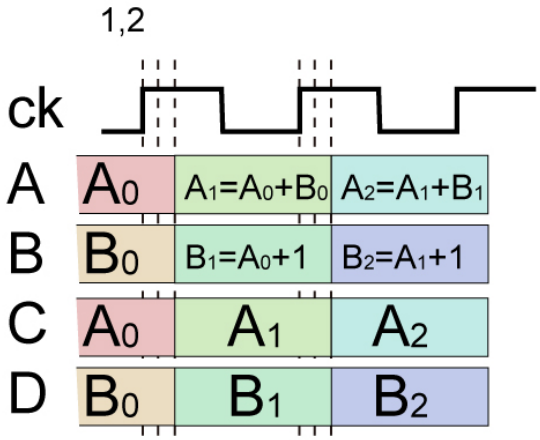
```
  0 0 0 0000
 10 1 0 0000
 20 0 0 0000
 30 1 0 0000
 40 0 1 0000
 50 1 1 0001
 60 0 1 0001
 70 1 1 0010
 80 0 1 0010
 90 1 1 0011
100 0 1 0011
110 1 1 0100

370 1 1 0001
380 0 1 0001
```

VerilogHDLの基本構文：代入

- 代入の時間関係

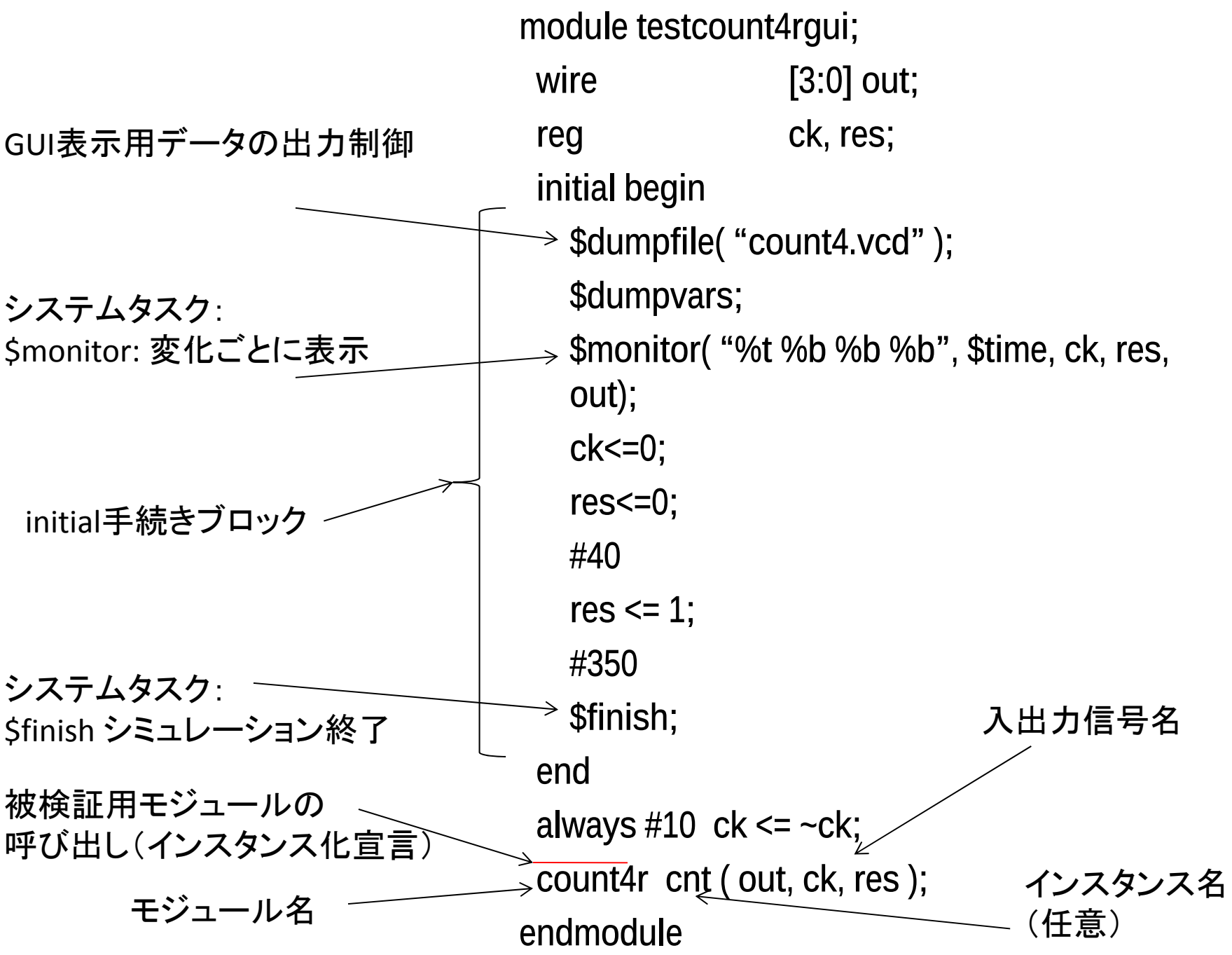
	ブロッキング代入	ノンブロッキング代入
	assign C=A;	assign C=A;
	assign D=B;	assign D=B;
	always @posedge ck	always @posedge ck
1	A=A+B;	A<=A+B;
2	B=A+1;	B<=A+1;
	end	end





ハードウエアの記述では必ず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）

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

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
```

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

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

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
```

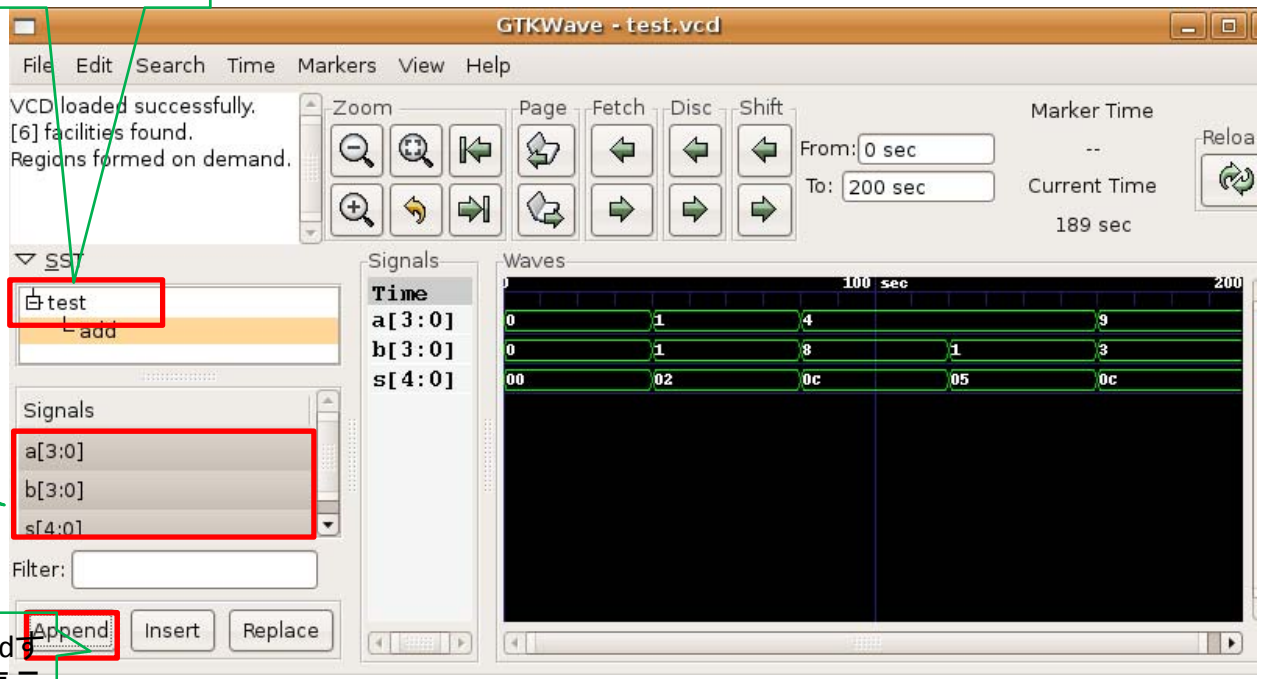
グラフィカルに見たい

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