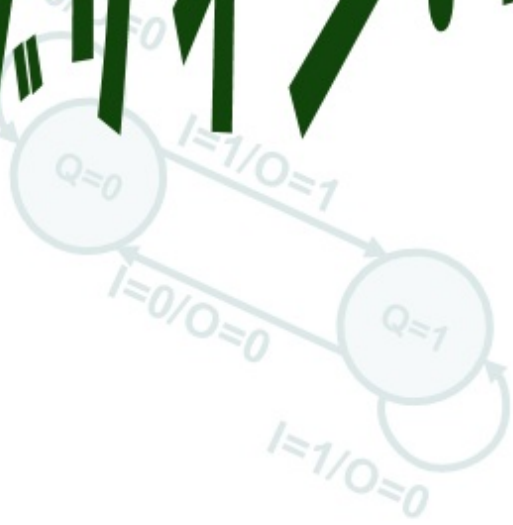
$$f(A,B,C)=(A+\overline{A})\cdot f(A,B,C)$$
$$=A\cdot f(A,B,C)+\overline{A}\cdot f(A,B,C)$$

Verilog HDL

デザイン・テンプレート



植木うてな

Verilog HDLの概要

歴史

1985年　Gateway Design Automation社（後にCadence社に買収）によって論理シミュレータ用言語として開発された。
1990年　OVI（Open Verilog International, 現Accellera）が設立され、言語仕様が公開された。
1995年　IEEE Std. 1364-1995として標準化された。
2001年　IEEE Std. 1364-2001として拡張された。
2005年　IEEE Std. 1800-2005としてシステムレベル記述への対応したSystemVerilogが標準化され，従来のVerilogはIEEE Std. 1364-2005として改定された。

特徴

・C言語に似た文法
・トランジスタレベルからシステムレベルまでの幅広い抽象度の記述が可能
・シミュレーションの入力パターン生成、結果表示、期待値比較まで一つの言語体系の中で記述できる

本テキストではIEEE Std. 1364-1995の範囲で、論理合成を前提としたRTL（Register Transfer Level）の記述について解説する。

文法（概要）

キーワード

module(入出力端子リスト）.. endmodule

　　この範囲が一つの回路モジュールとなる。

　　モジュールの集合によって回路全体が記述される。

begin .. end

　　汎用のブロック指定。

　　if文always文等の有効範囲を指定する。

initial

　　モジュールの中でシミュレーション上一回だけ実行されるブロック。

always @(イベント式)

　　シミュレーションにおいてイベント式にあるイベントが発生するたびに実行されるブロック。

　　if（条件式）else　..

　　条件判断文。

for(変数定義;条件式;変移)

　　繰り返し処理。

while(条件式)

　　繰り返し処理。

input、output

　　入力端子、出力端子の定義でモジュールの最初に記述。

reg、wire

　　wireは値を保持しない内部変数、regは値を保持する変数。

　　合成時には配線になるかレジスタになるかは記述による。

=

　　ブロッキング代入。

<=

　　ノンブロッキング代入。

assign

　　継続的代入。

　　シミュレーション上はイベントを伴わずタイムユニット毎に式が評価される。

function

　　関数定義。

//、/* .. */

　　コメント。

　　実行には影響しない。

task

関数定義の内、平行処理するもの。

オペレータ（演算子）

+、-、*、/

加減乗除（算術演算）。

%

除算の余り（剰余演算）。

~、&、|、^、~^(^~)

not、and、or、xor、xnor（ビット演算）。

&&、||、!

and、or、not（論理演算）。

!=、==、!==、===

等号演算（前者2つが論理等号演算、後者2つがケース等号演算）。

{}

連節演算。

?:

条件演算。

RTL記述とは

RTL（Register Transfer Level）

RTLは回路内のレジスタ間のデータの流れを明示して記述する設計レベルである。また、回路機能の記述は、ゲートを直接記述するのではなく、なるべく高い抽象度で書くことが望ましい。整理すると下記の様になる。

データ

　　レジスタ間のデータ転送を組合せ回路として記述する。

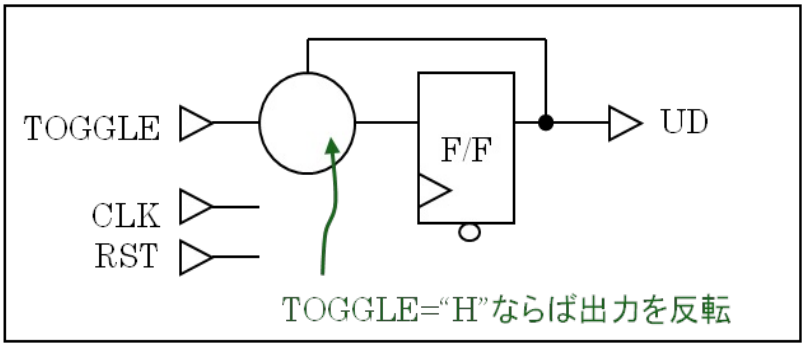
レジスタ

　　レジスタは構造記述するのではなく、機能記述から論理合成によって生成する（記述スタイルを参照）。

組合せ回路

　　組合せ回路はfunction文等を利用して、順序回路の記述と分離する。

下図のようなRTL図を作成してから記述すると書きやすくなる。



RTL図

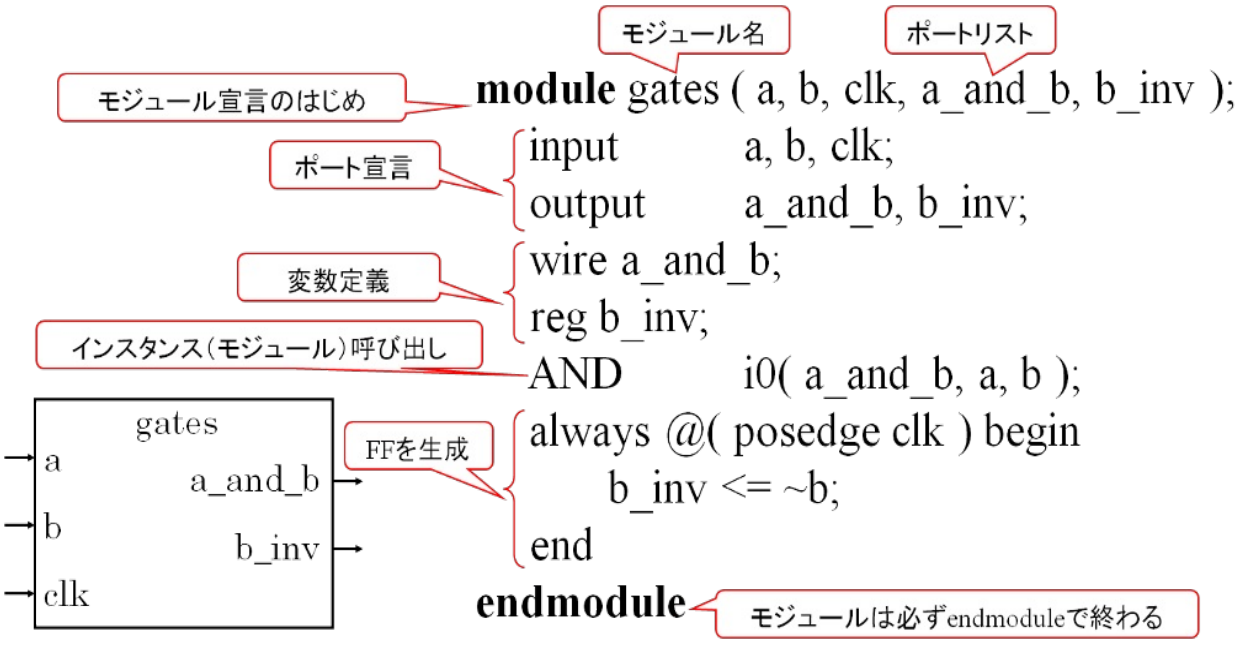
Verilog HDLによる設計の基本方針（７項目）

1. 論理合成で確実に動作する回路が得られることがわかっているのは同期回路。
2. 論理回路でクロックの分配回路を生成しない。
3. 記述レベルはRTL（レジスタと組合せ回路を分離した記述）。
4. 組合せ回路ではフィードバックの記述は書かない。
5. 論理合成後の回路構造をイメージして記述する。
6. 遅延指定（＃）やinitial文による初期化は使用しない（論理合成されない）。
7. 基本的に凝った記述はしない。

Verilog HDL記述の基本

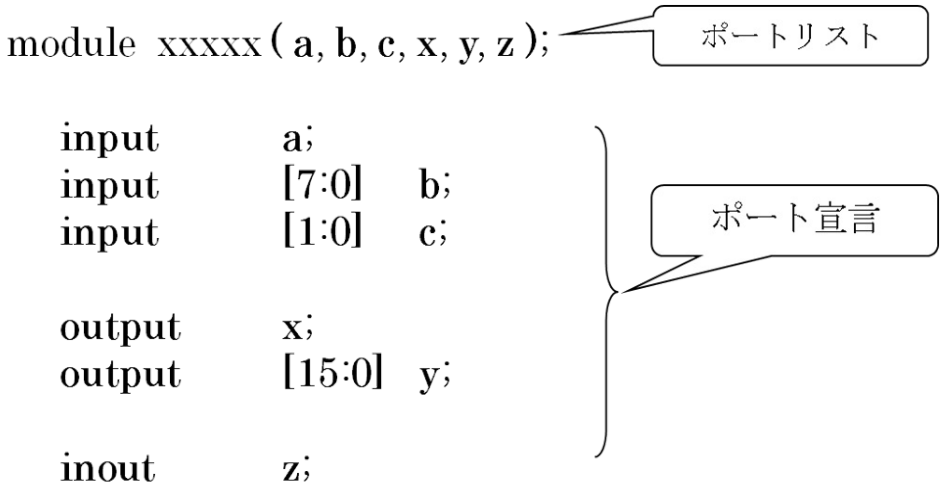
moduleはVerilog HDLにおける最も基本となる構文要素である。

設計する回路はすべてmoduleの集合からなる。



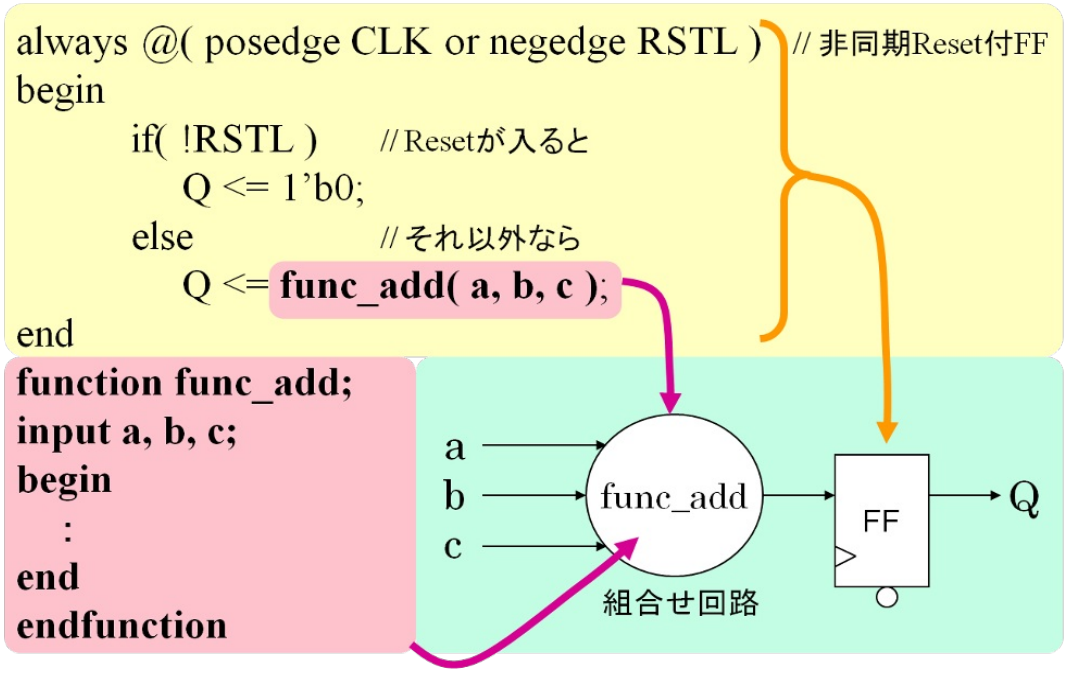
ポートリストとポート宣言

モジュールの入出力インタフェースを表す部分をポートリストという。
ポートリストはモジュール名の後に（）で括って定義する。
ポートリストに指定した各ポートはポート宣言を行う。
ポートリストには信号名のだけを記述、信号の幅はポート宣言で書く。



RTL記述の基本形

RTL記述は、下図のようにレジスタを生成するalways文と、組合せ回路を記述するfunction文の組合せで記述する。この記述であれば確実に論理合成できる。



インスタンス呼び出し

上位モジュールから下位モジュールを呼び出し配線する。これにより階層設計ができる。

モジュール名　　インスタンス名　（　ターミナルリスト　）；

《用語》インスタンス化(instantiation)，インスタンス(instance)とは？

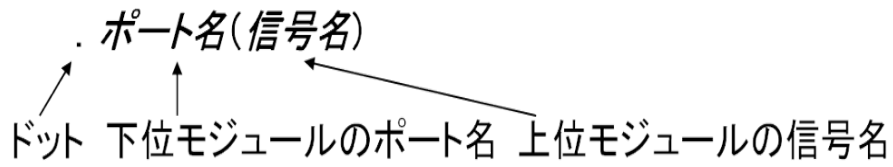
・インスタンス化(instantiation) ：実体化、定義されたモジュールを呼び出すこと

・インスタンス(instance)：実体≒部品として呼び出されたモジュール

下位モジュールのポート接続方法は①名前による方法と②位置による方法の２種類がある。

①名前による方法

ターミナルリストの各ポートについて下位モジュールのポート名を指定する。

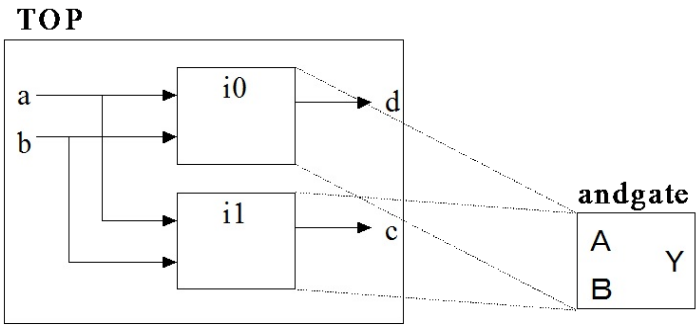


②位置による方法

ターミナルリストの各ポートについて下位モジュールで定義した順に指定する。

モジュールのインスタンス化の例

```
module top ( a, b, c, d );
input   a, b;
output  c, d;
    andgate  i0 ( .A(a), .B(b), .Y(d) );          ・・・（名前による指定）
    andgate  i1 ( c, a , b ) ;                          ・・・（位置による指定）
endmodule

module andgate ( Y, A, B );
input   A, B;
output  Y;
   assign Y = A & B;
endmodule
```



データタイプと数値表現

データタイプ

ネット宣言　　：wire, wand, wor, tri0, tri1, supply0, …
レジスタ宣言　：reg, trireg,…
整数宣言　　：integer
実数宣言　　：real

その他

パラメータ　　：parameter
デファイン　　：define

論理合成で使われるのは、ほぼ赤字の５種類。

数値表現

論理値　０，１，ｘ，ｚ
数値表現〈ビット幅〉’〈基数〉〈数値〉
基数 b,B：２進数，o,O：８進数，d,D：１０進数，h,H：１６進数

データタイプの詳細

① wire

・wireは回路内で配線を表す。

・各インスタンス間の配線を行う。

・wire自身は値を保持しない。

・継続的代入文を使用して代入する。

```
wire a;
wire [7:0] b;
```

② reg

・regは値を保持する変数。

・ある代入から次の代入まで値を保持する。

・reg変数≠Flip Flop（合成の結果がFFとは限らない）。

・手続き的代入文を使用して代入する。

```
reg a;
reg [7:0] b;
```

③ integer

・integerはfor-loop等で使用する整数変数。

・整数の長さはデフォルトは32ビット。

・論理合成を前提として記述では極力使わない。ただし、for-loopに限って使用することもある。

```
integer i;
for ( i=0; i < 8; i=i+1 )
     c[i] = a[i] & b[i] ;
```

④ parameter

・parameterは定数(constant value)を表す。

・parameterを用いることで可読性，汎用性の高い記述ができる。

・モジュール内で有効。

```
parameter TRUE=1 , FALSE=0 ;
parameter INIT=4’b0100 ,  DONE=4’b1011;
parameter [1:0] S0=3 ,  S1=1 ,  S2=0 ,  S3=2 ;
```

④ define

・defineは定数を表す（コンパイル指示子扱い）。

・defineもparameterと同様に可読性，汎用性の高い記述ができる。

・defineはデザイン全体で有効（モジュール単位で論理合成する場合は注意が必要）。

```
`define INIT  4’b0100
`define DONE   4’b1011
`define  size 4
```

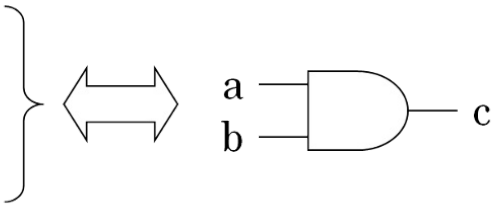
```
module padder ( a, b, c );
    input [`size-1:0] a, b;
    output [`size-1:0] c;
    assign  c = a + b;
endmodule
```

データタイプと数値表現（具体例）

下図のようにイベント文のalwaysによって、同じreg宣言であっても論理合成結果が異なる。

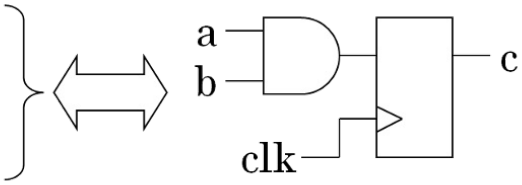
FFにならない例

```
reg     c;
always @( a or b )
begin
        c = a & b;
end
```



FFになる例

```
reg     c;
always @( posedge clk )
begin
        c <= a & b;
end
```



parameter文は、汎用性の高い記述にするために使う。また、インスタンス化時にデフォルトパラメータの値を変更することができる。

回路のパラメータ化例

```
module top ( A, B, C );
input [7:0] A , B;
output [7:0] C;
        padder  #(8)  i0( A, B, C );
endmodule

module padder ( a, b, c );
parameter     size=4;
input [size-1:0] a, b;
output [size-1:0] c;
    assign  c = a + b;
endmodule
```

パラメータで指定されているデフォルトの4ビットを8ビットに変更

デフォルトのサイズ

パラメータを用いて演算することも可能

信号のビット数をパラメータ化することで汎用性の高い記述になる

演算子の優先順位

優先順位で悩むより括弧を用いて演算の順序を明確にする方が良い

優先順位	演算タイプ	演算子	備考
	ビットセレクト，パートセレクト	[ ]	
	括弧	( )	
	論理否定，ビット反転	！ ～	
	リダクション演算	& | ~& ~| ^ ~^ ^~	全ビットの演算，結果は1ビット
	連接演算子	{ }	複数信号のベクタ化に使用
	乗算，除算，剰余	＊ ／ %	大きな回路になるので注意
	加算，減算	＋ －	
	シフト演算	<< >>	大きな回路になるので注意
	比較演算	< <= > >=	
	等号，不等号	== !=	
	ビット論理積	&	
	ビット排他的論理和	^ ^~ ~^	
	ビット論理和	|	
	論理積	& &	
	論理和	| |	
	条件演算	？：	右から左へ評価

演算子の使用例

算術演算子

```
parameter size=8;
wire [3:0] a, b, c, d, e;
assign c = size + 4'h2;
assign d = a + 4'h1;
assign e = a + b;
```

←定数＋定数　回路にならない（**定数**）
←変数＋定数　**インクリメンタ**が合成される
←変数＋変数　**加算器**が合成される

リダクション演算子

```
wire a, b;
wire [3:0] c, d;
assign a = &c;
assign b = ^d;
```

← a = c[0] & c[1] & c[2] & c[3]
← b = d[0] ^ d[1] ^ d[2] ^ d[3]

連接演算子

```
wire a, b, c, d;
wire [3:0] e;
wire [15:0] f;
assign e = {a, b, c, d };
assign f = {1'b1, {15{1'b0}}};
```

← e[3] = a, e[2] = b, e[1] = c, e[0] = d
← f = 16'b10000000_00000000

制御構文

【ブロック文】　begin～end

．２つ以上の文をまとめて構文上１つの文にしたもの。

．beginとendにはさまれた文を逐次的に実行する。

．beginとendは入れ子構造にすることが可能。

【論理合成に用いる制御構文】

if～else～, if～else if～else～

case～endcase（ casex, casez ）

for～

基本的にこれだけしか使わない。

これらはfunction文，always文で使用。

assign文で“？：”演算子を用いて制御することもある。

---

①if…else文

例）if・・・ else if・・・ else構造

```
if( s ) begin                  ←sの値が真(s=1)の時,
  a = 8'h0;                      begin以下を処理する
end
else if( e == 1'b0 ) begin     ←sの値が偽(s≠1)でかつeの値
  a = a + 8'h1;                  が0の時，begin以下を処理する
end
else begin                     ←sの値が偽でかつeの値が0で
  a = a;                         ない時，begin以下を処理する
end
```

誤ってラッチを生成しないようにelseはつける

②case, casex, casez文

・機能的にはif・・・ else文と同等

```
case ( s )                              ←条件分岐式(s)
  3'b000 : out = 8'b1111_1110;
  3'b001 : out = 8'b1111_1101;
  3'b010 : out = 8'b1111_1011;
  default: out = 8'b1111_1111;          ←sが上記以外の時,
endcase                                   outに1111_1111代入
```

- casez文
  各分岐式に**z, ?** を使用でき，各分岐式中の**z, ?** はdon't careとして扱われる
- casex文
  各分岐式に**x, z, ?** を使用でき，各分岐式中の**x, z, ?** はdon't careとして扱われる

組合せ回路の記述

組み合せ回路は、①function文，②always文，③assign文の３種類のうちのいずれかを用いて記述する。

3種類の記述方法の特徴

	function文	always文	assign文
長所	**組み合せ回路が生成されることを保証**	記述が容易（記述量が少ない）	**組み合せ回路が生成されることを保証**
短所	複数の出力を持つ回路が記述しにくい	**意図しないラッチを生成する危険性有り**	複雑な回路を記述しにくい
備考	モジュール内の変数が参照できるので，**使用されている変数が入力として宣言されているか確認**	不完全なif文（対応するelse文の無いif文）やdefaultの無いcase文を記述しない．また，**Sensitivity Listを十分にチェックする**	regなどに値を割り当てることはできない **可読性が落ちるので使用しない方が良い**

組合せ回路の記述例

function記述の例

任意のビット幅に対応可能な
インクリメンタのfunction記述例

```
parameter width = 4;
function [width-1:0] inc;
input [width-1:0] a;
reg c;
integer i;
begin
  c = 1'b1;
  for ( i=0; i<width; i=i+1 )  begin
       inc[i] = c ^ a[i];
       c = c & a[i];
  end
end
endfunction
```

functionの呼び出し例（1）

```
wire [3:0] y;
assign    y  =  inc( b );
```

functionの呼び出し例（2）

```
reg [3:0] y;
always @( posedge clk )
   y  <=  inc ( c );
```

always記述の例

手続き的代入文：ブロッキング代入文

データが16ビット幅の4to1セレクタ記述例

```
always @( a or b or c or d or s ) begin
  case( s )
    2'b00 : y = a;
    2'b01 : y = b;
    2'b10 : y = c;
    2'b11 : y = d;
    default : y = 16'bx;
  endcase
end
```

ここにalways内で入力に使われる信号（代入文の右辺や条件式）をすべて列挙する

assign記述の例

継続的代入文

データが16ビット幅の4to1セレクタ記述例

```
wire [15:0] y;
assign y=(s==2'b00)?a:(s==2'b01)?b:(s==2'b10)?c:(s==2'b11)?d:16'bx;
```

可読性が悪いのでこのような記述はしない方が良い

マスク処理の記述例

```
reg [15:0] mask;
wire [15:0] in_data;
wire [15:0] out_data;
assign out_data= in_data & mask;
```

順序回路の記述

順序回路は、always文を用いて記述する（基本記述スタイルを参照）。

順序回路で用いる代入文

　ブロッキング代入文（＝）とノンブロッキング代入文（＜＝）の２種類。

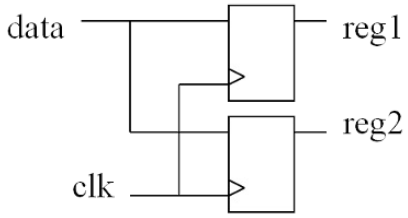
遅延について

　各々代入時に”= #5”や”<= #5”のように遅延を指定することも可能．ただし、論理合成時には無視される。

ブロッキング代入文（＝）とノンブロッキング代入文（＜＝）

例1） ブロッキング代入文を使用

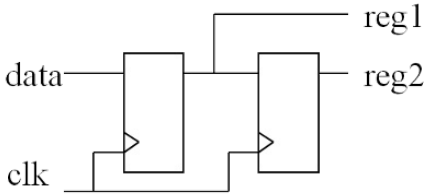
```
always @( posedge clk ) begin
  reg1 = data;
  reg2 = reg1;
end
```



(順次構文として認識される)

例2） ノンブロッキング代入文を使用

```
reg reg1, reg2;
always @( posedge clk ) begin
  reg1 <= data;
  reg2 <= reg1;
end
```



(正しくシフトレジスタとして認識される)

リセットについて

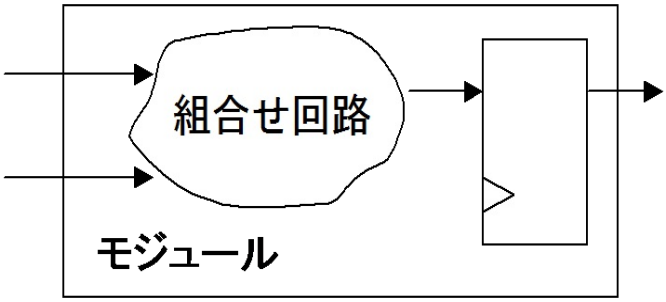
基本は非同期リセット。
同期リセットはリセット信号を生成する部分を合成によって作るため危険を伴う。
同じリセット信号で同期リセットと非同期リセットを混在させない。

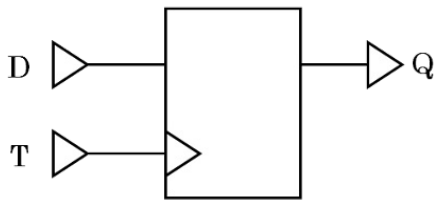
```
always @( posedge CLK or negedge RSTL )
   if( !RSTL )
     Q <= 1’ b0;
  else
     Q <= DATA;
```

回路分割について

ASICなどでは回路規模が大きくなり、一度に論理合成できない。
FPGA/CPLDも大規模になってきており、モジュールごとの論理合成する。
分割の方法に気をつけないと良い結果が得られない。

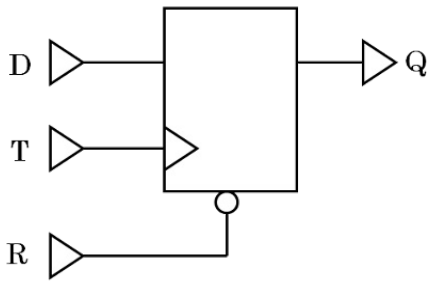
モジュールの出力をFFにする
合成時にタイミング制約を与えやすいし、最適な回路を生成しやすい。
ただし、FFの出力のファンアウトが大きい場合やタイミングが厳しい場合は、出力側のモジュールでFFのD入力に相当する信号を生成し、入力側のモジュールの受け口でFFでサンプリングするように回路分割を行った方が良い場合もある。





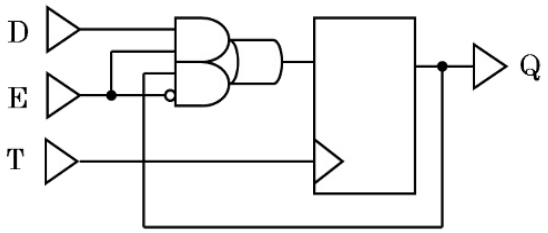

```
module  dff ( Q, D, T );
output Q;
input D, T;
reg Q;
always @( posedge T )
begin
        Q <= D;
end
endmodule
```

非同期リセット（セット）付きフリップ・フロップ



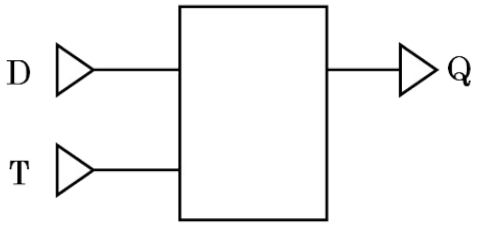

```
module  rdff ( Q, D, R, T );
output Q;
input D, R, T;
reg Q;
always @( posedge T or
negedge R )
begin
    if ( !R)
          Q <= 1'b0;
   else
          Q <= D;
end
endmodule
```

イネーブル付きフリップ・フロップ



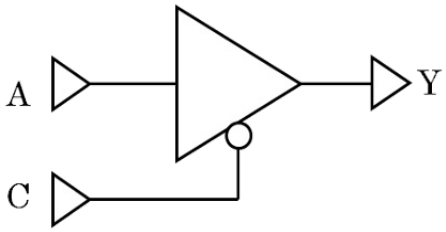

```
module  edff ( Q, D, E, T );
output Q;
input D, E, T;
reg Q;
always @( posedge T )
begin
   if ( E )
         Q <= D;
  else
         Q <= Q;
end
endmodule
```

ラッチ




```
module  dlatch ( Q, D, T );
output  Q;
input D, T;
reg Q;
always @( D or T )
if ( T )
        Q <= D;
endmodule
```

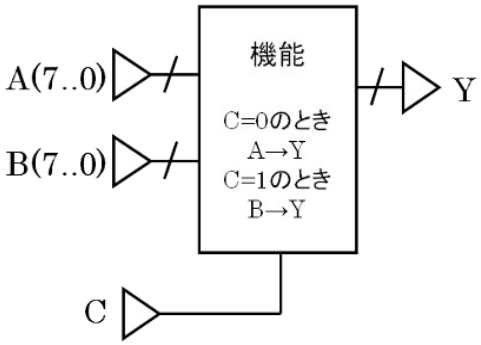
トライステート・バッファ




```
module  trist ( Y, A, C );
output Y;
input A, C;
reg Y;
always @( A or C )
if ( !C )
        Y <= A;
else
        Y <= 1'bz;
endmodule
```

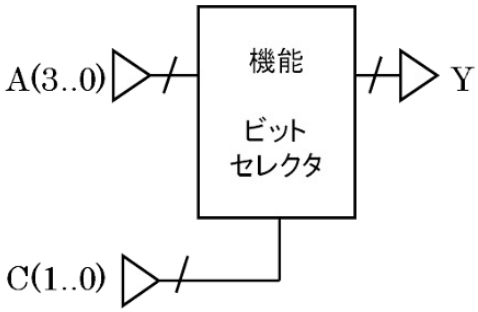
assign Y = C ? 1'bz : A; でも可

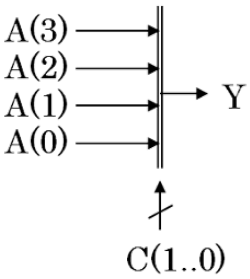
マルチプレクサ（セレクタ）その１




```
module  mux ( Y, A, B, C );
output [7:0] Y;
input [7:0] A, B, C;
  assign Y = func_mux ( A, B, C );
  function [7:0] func_mux;
  input [7:0] A, B;
  input C;
    if ( C == 1'b0 )
       func_mux = A;
    else if ( C == 1'b1 )
       func_mux = B;
    else
       func_mux = 8'bxxxxxxxx;
  endfunction
endmodule
```

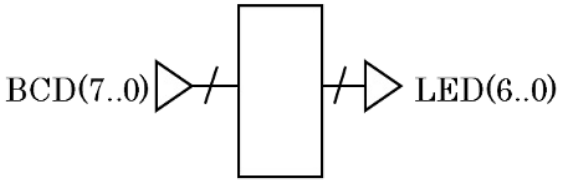
マルチプレクサ（セレクタ）その２





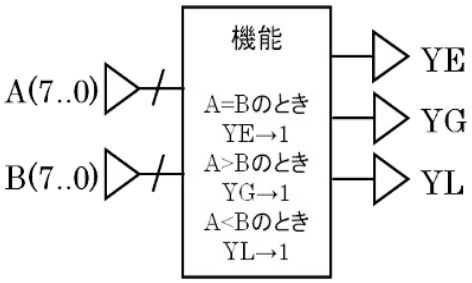

```
module  mux ( Y, A, C );
output Y;
input [3:0] A;
input [1:0] C;

  assign Y = A[C];

endmodule
```

デコーダ



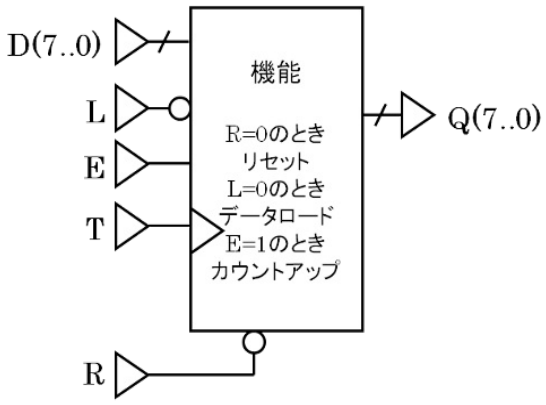

```
module  dec ( LED, BCD );
output [6:0] LED;
input [3:0] BCD;
  assign LED = func_dec ( BCD );
  function [6:0] func_dec;
  input [3:0] BCD;
    case ( BCD )
       4'b0000 : func_dec = 7'b1111110;
       4'b0001 : func_dec = 7'b1100000;
       4'b0010 : func_dec = 7'b1011011;
       4'b0011 : func_dec = 7'b1110011;
       4'b0100 : func_dec = 7'b1100101;
       4'b0101 : func_dec = 7'b0110111;
       4'b0110 : func_dec = 7'b1011111;
       4'b0111 : func_dec = 7'b1110000;
       4'b1000 : func_dec = 7'b1111111;
       4'b1001 : func_dec = 7'b1110111;
       default  : func_dec = 7'bxxxxxxx;
    endcase
  endfunction
endmodule
```

コンパレータ



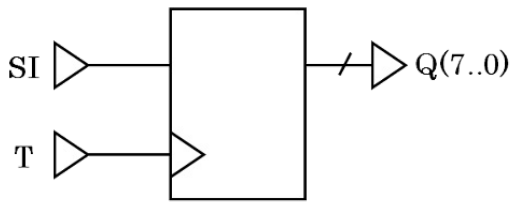

```
module  cmp ( YE, YG, YL, A, B );
output YE, YG, YL;
input [7:0] A, B;
 assign { YE, YG, YL } = func_cmp ( A, B );
 function [2:0] func_cmp;
 input [7:0] A, B;
 begin
   if ( A == B )
      func_cmp[2] = 1'b1;
   else
      func_cmp[2] = 1'b0;
   if ( A > B )
      func_cmp[1] = 1'b1;
   else
      func_cmp[1] = 1'b0;
   if ( A < B )
      func_cmp[0] = 1'b1;
   else
      func_cmp[0] = 1'b0;
 endfunction
endmodule
```

カウンタ

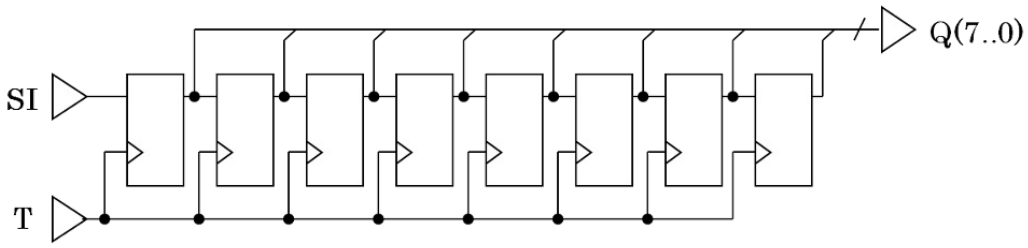



```
module  count ( Q, D, E, L, R, T );
output [7:0] Q;
input [7:0] D;
input E, L, R, T;
reg [7:0] Q;
  always @ ( posedge T or negedge R)
  begin
     if ( !R )
       Q <= 8'h00;
     else if ( !L )
       Q <= D;
     else if ( !E )
       ;
     else if ( Q == 8'hff )
       Q <= 8'h00;
     else
       Q <= Q + 8'h01;
  end
endmodule
```

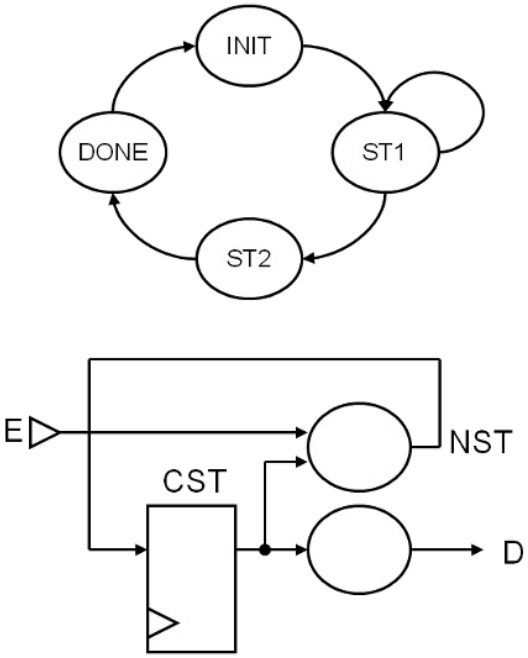
シフトレジスタ




```
module  sht ( Q, SI, T );
output [7:0] Q;
input SI, T;
reg [7:0] Q;
 always @( posedge T )
 begin
    Q <= { Q[6:0], SI };
 end
endmodule
```



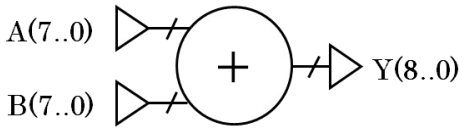
ステートマシン



ムーア型ステートマシン

```
module  count ( D, E, RSTL, CLK );
output D;
input E, RSTL, CLK;
`define INIT 2'b00
`define ST1 2'b01
`define ST2 2'b10
`define DONE 2'b11
reg [1:0] CST, NST;
wire D;
assign D = ( CST == `DONE );
always @ ( posedge CLK or negedge RSTL )
    if( !RSTL )  CST <= `INIT;
    else             CST <= NST;
always @ ( E or CST )
  case( CST )
     `INIT : NST = `ST1;
     `ST1 : if ( E == 1'b1 ) NST = `ST2;
              else NST = `ST1;
     `ST2 : NST = `DONE;
     `DONE : NST = `INIT;
     default : NST = `INIT;
  endcase
endmodule
```

加算器




```
module  add ( Y, A, B );
output [8:0] Y;
input [7:0] A, B;
  assign Y = { 1'b0, A } + { 1'b0, B };
endmodule
```

おわりに

論理合成を前提としたVerilog HDLによるRTL設計について、ノウハウとデザイン・テンプレートをざっくりまとめてみました。説明が大雑把なところもありますが、まぁ、参考資料程度には役に立つのではないでしょうか（タダですし）。

できれば、フリーのIcarus Verilogシミュレータでも使って動かしていただくと、より実感が湧くのではないかと思います。時間がとれればシミュレーション編も書きたいですが…、先のことは分かりません。

Verilog HDL デザイン・テンプレート
2011年2月8日
植木うてな