Verilog - HDL
簡易文法書

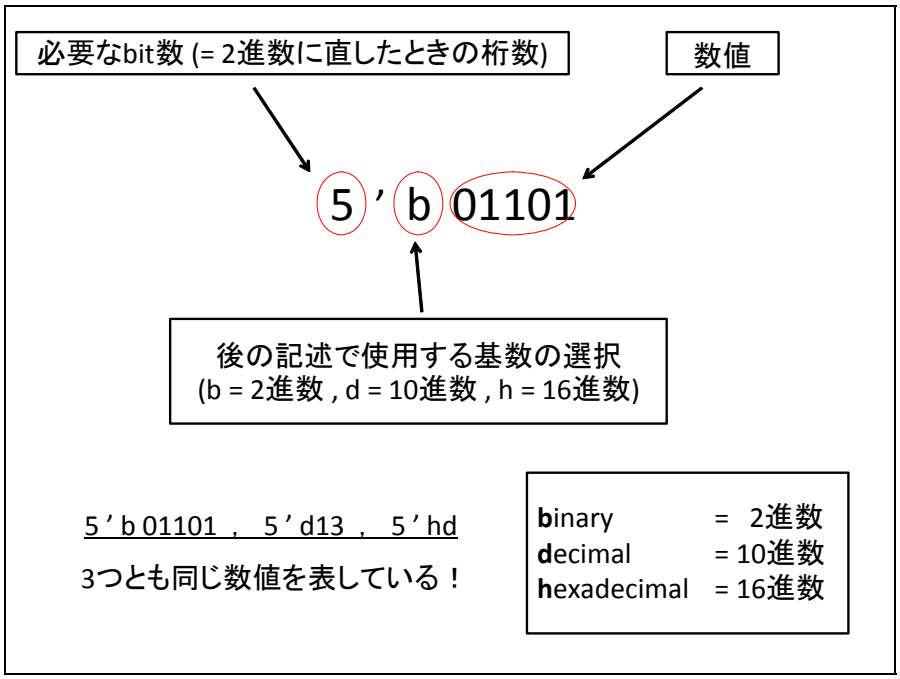
東京電機大学　工学部
情報通信工学科

町田　匠

＜定数の表記＞

「bit 数‘ 基数 数値」の順で記述する。

基数の表記が異なるものでも演算可能。



＜変数名の表記＞

・予約語を除いて任意の変数名を宣言できる。

・大文字小文字を区別し、数字と”_“を使用可能。(ただし一文字目に数字は使用できない)

・ビット幅の記述で複数ビットを宣言できる。[n - 1：0],[0：n - 1] ビット幅:n ビット

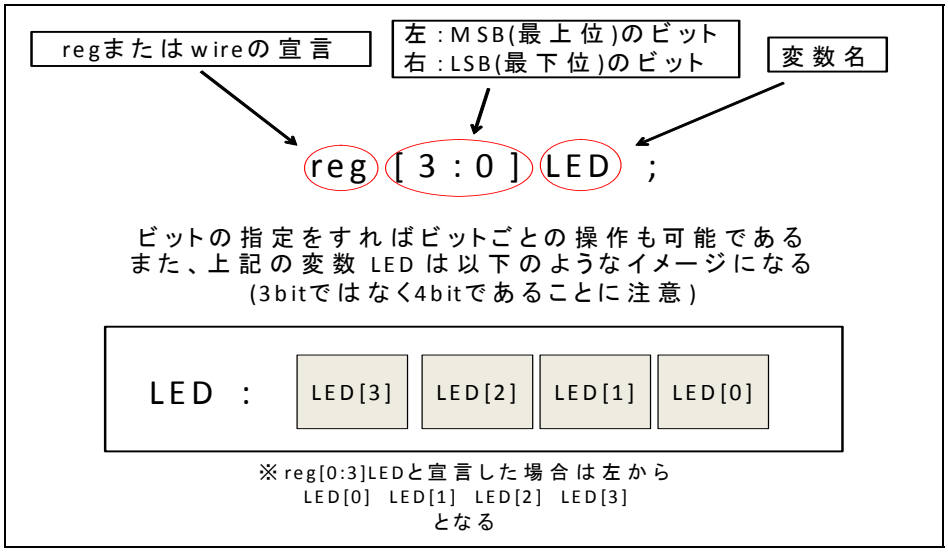
・「reg」と「wire」の二通りの宣言があり変数は必ずどちらかに属する。

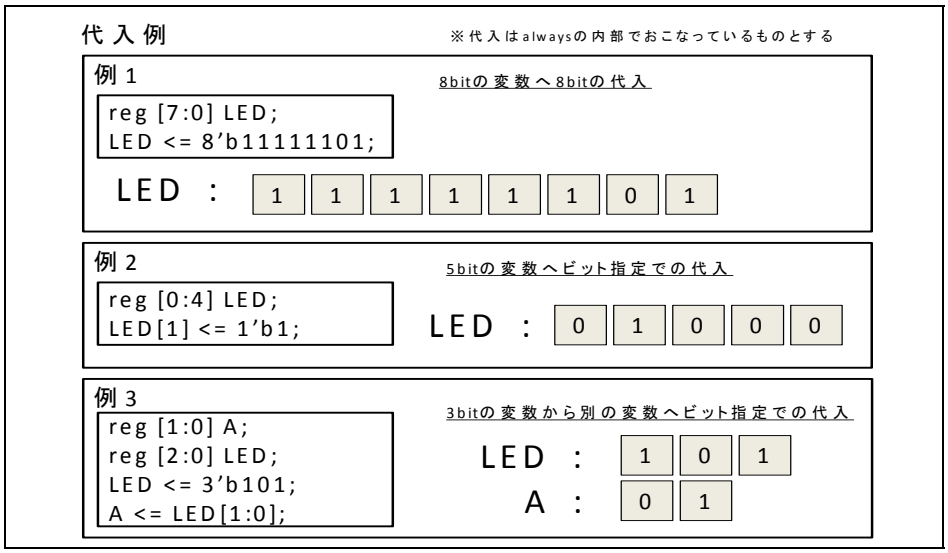
・reg,wire 宣言は必ず全ての変数におこなうようにすること。

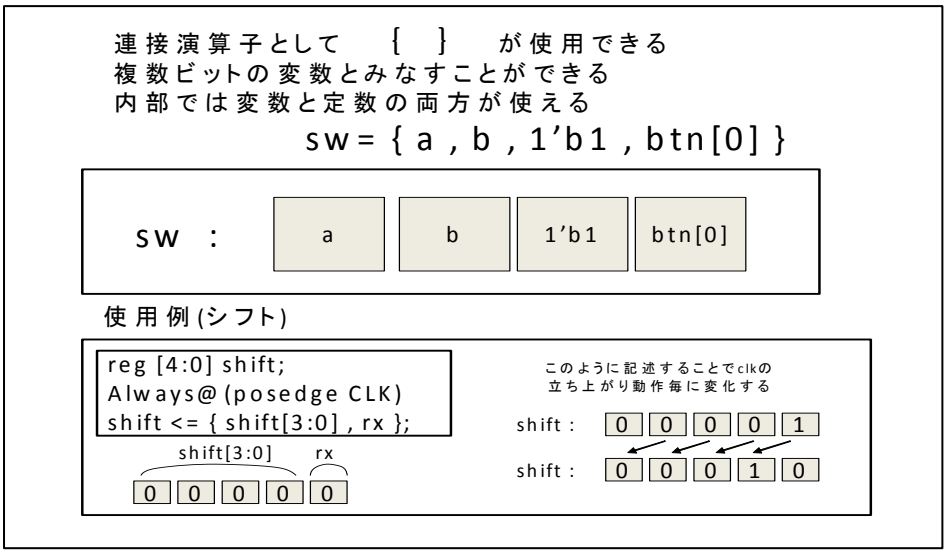
・reg , wire 宣言について

reg レジスタ宣言：
　値を保持する変数に対して宣言する。
wire ワイヤ宣言：
　配線として使用する変数に対して宣言する。
　(assign 文によって常時右辺の結果を出力する)
なれないうちは使い分けが難しいので
　・always 内で左辺として利用する場合は reg 宣言。
　・assign の左辺は wire 宣言
　・上記以外を wire 宣言
と覚えておくとよい。

・複数ビットの扱いについて







＜演算子＞

・代入演算子

ブロッキング代入「=」とノンブロッキング代入「<=」の二通りの代入演算子が存在する。
使い分けが難しいので
・wire 変数への代入には「=」(assign 文内部)
・reg 変数への代入には「<=」(always 文内部)
と覚えておくとよい。

・算術演算子

演算子	意味
＋	加算
－	減算
＊	乗算
／	除算

※一部の FPGA で「 ＊ 」「 / 」が使えない場合がある。

・ビット演算子

演算子	意味
&	AND
|	OR
~	NOT
^	EX-OR

※NOT と&は単項演算子である。

・関係演算子

演算子	意味
<	右辺未満
<=	右辺以下
>	右辺より大きい
>=	右辺以上
==	等しい
!=	等しくない

・　シフト演算子

演算子	意味
<<	左シフト
>>	右シフト

＜module の入出力宣言＞

FPGA への入力と出力を宣言する。

入力：input

出力：output

で宣言する。

ここでは、wire,reg,bit 数の宣言をしないで、内部で宣言することも可能であるが、一回でまとめて宣言してしまう方が二重宣言、宣言し忘れなどの間違いが少なくなるので、module の冒頭で宣言することとする。

```
module モジュール名( 入出力宣言 );
  動作記述
endmodule
```

例

```
module test(
        input wireCLK,
        input reg data,
        input reg [4:0] address,
        output wire [7:0] LED
);
```

入出力宣言

＜assign 文＞

・assign（割り当て）文として記述したものがひとつの回路（NET）となる。

・bit 演算による組み合わせ回路の記述、二つの変数を「=」で繋ぐ配線などの単純な組み合わせ回路を記述できる。(順序回路は記述できない)

```
assign 代入先変数 = 組み合わせ回路記述 ;
```

例

```
assign  A = ( B & C ) | ( D & E );
```

※上記の式は B , C の論理積と D , E の論理積の論理和

＜always 文＞

・always 文として記述したものがひとつの回路（NET）となる。
・組み合わせ回路と順序回路の記述が可能である。
・入力を列挙したものをセンシティビティリスト(*Sensitivity List*)と呼び、always@の直後のカッコ内に記述する。
・センシティビティリストの変数の変化(0 →1 , 1 →0 など)をトリガとして always 内の処理が開始される。
・すべての always 内の文は同時処理であるため記述の順番による処理の時間差はない。
・posedge によりポジティブエッジ（立ち上がり）、negedge によりネガティブエッジ(立ち下がり)を検出できる。

```
always@(センシティビティリスト)begin
  動作記述
end
```

例

```
always@(posedge CLK)begin
  if(count == 3'b111)
    count <= 3'b0;
  else
    count <= count + 3'b1
end

always@(count)begin
        case(count)
                3'b000:LED<=7'b0000000;
                3'b001:LED<=7'b0000001;
                3'b010:LED<=7'b0000010;
                3'b011:LED<=7'b0000100;
                3'b100:LED<=7'b0001000;
                3'b101:LED<=7'b0010000;
                3'b110:LED<=7'b0100000;
                3'b111:LED<=7'b1000000;
                default:LED=7'b0000000;
        endcase
  end
```

CLK の立ち上がりで起動

count が 111(2 進数)のときゼロリセット

インクリメント(1 増やす)

count の変化(立ち上がり,立ち下りなど)により起動

count の値により
LED の ON/OFF を制御

＜分岐文(if 文)＞

・分岐後の処理が複数になる場合は begin と end で囲み、記述する
・always 文内に if 文を用いて条件分岐を記述することができる。
・逐次処理のように見えるが順次記述した実行文はすべて同じタイミングで処理し、いっせいに代入を実行する。
・「 if 」,「 else if 」によって指定した条件に一致しない場合は else が実行される。

```
if (条件式 1)
   動作記述
else if (条件式 2)
   動作記述
else if (条件式 3)
   動作記述
else
   動作記述
```

例

```
always@(posedge CLK or negedge RESET)begin
        if (RESET == 1'b0)
                count <= 1'b0;          ―――― ①
        else if (count == 4'd9)
                begin
                        count<= 1'b0;
                        flag <= 1'b1    同時実行 ―――― ②
                end
        else
                count <= count + 4'd1   ―――― ③
        end
end
```

CLK の立ち上がり
または
RESET の立ち下がり
で起動

条件分岐により①,②,③のいずれかを実行。

＜分岐文(case 文)＞

・always 文内に case 文を用いて条件分岐を記述することができる。
・条件により 1 つの文のみを実行する。
・begin と end で囲むことで複数の動作文を記述できる。
・ひとつの変数の値のみによって出力が決定する場合には if 文よりも見やすいコードになる。
・指定した値に一致しない場合は default が実行される。

```
case(条件変数)
                条件数値 1： 動作記述
                条件数値 2： 動作記述
                条件数値 3： 動作記述
                条件数値 4： 動作記述
                default： 動作記述
endcase
```

例

```
  always@(count)begin
        case(count)
                4'b0000:LED<=7'b1111110;   //0
                4'b0001:LED<=7'b0110000;   //1
                4'b0010:LED<=7'b1101101;   //2
                4'b0011:LED<=7'b1111001;   //3
                4'b0100:LED<=7'b0110011;   //4
                4'b0101:LED<=7'b1011011;   //5
                4'b0110:LED<=7'b1011111;   //6
                4'b0111:LED<=7'b1110000;   //7
                4'b1000:LED<=7'b1111111;   //8
                4'b1001:LED<=7'b1111011;   //9
                default:LED<=7'bx;  ←――― X は不定（どちらでもよい）の表現
        endcase
  end
```

＜サンプル集＞

解析用サンプル１

１秒毎に LED が点灯と消灯を繰り返す(水晶発振子は 100MHz)

```
module test(
	input wire CLK,
	output wire LED
	);

	reg [26:0] count;
	reg T1s;
	reg toggle;

	always@(posedge CLK)begin
		if(count == 27'd99999999)
			begin
				count <= 27'd0;
				T1s <= 1'b1;
			end
		else
			begin
				count <= count + 27'd1;
				T1s <= 1'b0;
			end
	end

	always@(posedge T1s)begin
		toggle <= ~toggle;
	end

	assign LED = toggle;

endmodule
```

「CLK」は水晶発振子、「LED」は LED に接続されているものとする。
この動作がイメージできればだいたいのソースは読めるはずである。

解析用サンプル 2

デコーダ回路

```
module decoder(
        input wire A,
        input wire B,
        input wire C,
        output reg[6:0]LED);
always@(A,B,C)
begin
        case({A,B,C})
                3'b000:LED=7'b0000000;
                3'b001:LED=7'b0000001;
                3'b010:LED=7'b0000010;
                3'b011:LED=7'b0000100;
                3'b100:LED=7'b0001000;
                3'b101:LED=7'b0010000;
                3'b110:LED=7'b0100000;
                3'b111:LED=7'b1000000;
                default:LED=7'bx;
        endcase
end
endmodule
```

解析用サンプル 3

チャタリング除去回路

```
module chattering_eliminator(
                input clk,
                input enable,
                input i_sw,
                output o_sw);
reg [4:0] shift;
always@(posedge clk)begin
        if(enable == 1'b1)
                shift <= {shift[3:0], i_sw};
end
assign o_sw = &shift;
endmodule
```

記述の拡張

＜インスタンス化(階層化)＞

・HDL はモジュールの再利用が可能となっているので、同じ機能の回路を複数作りたいときに何度も回路を記述しなくて済む。また、回路内に既に完成した回路を組み込みたい場合にも有効である。
・モジュール内の parameter は受け渡した値により上書きされる。
・パラメータの受け渡しはしなくてもよい。
・インスタンスの入出力ポートには必ず信号を接続する。(ポート名が正しければ順不同)
・wire と reg の区別がつかない人には少々難しいのでお勧めしない。
・ひとつの回路内で同じインスタンス名をつけることはできない。

```
モジュール名　#(.モジュール内パラメータ名 (受け渡すパラメータ))
                    インスタンス名( . モジュールのポート名(接続信号) , ・ ・ ・);
```

例

```
filter    #(.coefficient(cf1))   I1(.toggle(toggle), .slot_in (slot3_in), .slot_out (slot31_out));
filter    #(.coefficient(cf2))   I2(.toggle(toggle), .slot_in (slot3_in), .slot_out (slot32_out));
filter    #(.coefficient(cf3))   I3(.toggle(toggle), .slot_in (slot3_in), .slot_out (slot33_out));
```

＜定数の表記 2＞

・係数、レジスタのアドレスなどは変化しないので定数を用いることになる。
その際に wire や reg の宣言ではなく parameter を宣言しておくとインスタンス化したときに簡単に書き換えることができる。
・値の渡し方は前述の「インスタンス化」を参照。

```
parameter [ビット幅 s ] 変数名 = 定数;
```

例

```
parameter [19:0] address = 20'h04040;
```

＜分岐文(assign)＞

・always 文の中で if 文、または case 文を使用して分岐をおこなうこともできるが、エッジの検出をおこなわない場合は assign 文で単純に書くことができる。
・また、「条件が真のときの値」や「条件が偽のときの値」の部分に「条件式」と「?」を加えることでさらに条件分岐をさせることが可能である。しかし多用すると読みにくいソースとなるので、分岐が多い場合は if や case を使ったほうが良い。

```
assign 変数名 = (条件式) ? 条件が真のときの値 : 条件が偽のときの値;
```

例

```
assign enable = (count == 14'd9999) ? 1'b1 : 1'b0;
```

＜二次元配列＞

・二次元配列としてメモリを構成することができる。
・二次元配列はパラメータとして上書きすることは可能であるが、信号の配線としてインスタンス化したモジュールの入出力ポートにつなぐことはできない。
・メモリ列ごとのアクセス、1 ビットごとのアクセスが可能。

```
変数型 [ビット幅] 変数名 [メモリ幅] ;
```

例

```
parameter [15:0] coefficient [0:50];
reg signed [15:0] memory [0:50];

always@(posedge clk)begin
        if(count == 4'b1111)begin
                memory[15] <= 16'h0f0f;
                memory[2][0] <= 1'b0;
                memory[2][15:1] <= 15'h7fff
        end
end
```

＜繰り返し(for 文)＞

・用途としては単純に記述の省略である。
・配列をシフトさせるときに少ない記述でシフトさせることができる。
・for 文であるためクロックを用いてループ処理をするように思えるが、1 クロックで全ての処理を終える。つまり、複数の代入文のならびになる。

```
for(初期化式; 条件式; 反復式; )begin
        動作記述
end
```

例

```
integer i;
for(i=49; i>=0; i=i-1)begin
        left  [i+1] <= left  [i];
        right [i+1] <= right [i];
end
```

＜ノンブロッキング代入,ブロッキング代入＞

・「<=」での代入がノンブロッキング代入。
・「=」での代入がブロッキング代入。
・ノンブロッキング代入は右辺を全て評価した後に全て同時に代入する。
・ブロッキング代入は右辺を評価した直後に左辺に代入する。(逐次処理に似ている)

代入イメージ

初期値を X が 1,　A が 5,　B が 0 とする。

<u>ノンブロッキング代入</u>

```
X <= A;
B <= X;
```

X が 5,　A が 5,　B が 1　となる。

<u>ブロッキング代入</u>

```
X = A;
B = X;
```

X が 5,　A が 5,　B が 5　となる。