Verilog HDL 入門

松谷 宏紀 *

1 Verilog HDL の基本文法

1.1 基本的な構造

- 1 つの機能単位をモジュールとして設計する。
- モジュールは module で始まって endmodule で終わる。
- C 言語ではブロックを {} でくくることで制御構造を作るが、Verilog HDL では begin で始まり end で終わることで制御構造を作る。

以下に基本的なモジュールの記述方法を示す。各部の詳しい説明は後述する。

```
------------------------- ここから -------------------------
module モジュール名の記述部 ( 入出力ポートの記述部 );

  パラメータ、レジスタ、ネットの指定の記述

  下位モジュールの呼び出しの記述

  回路の記述

  ...

endmodule
------------------------- ここまで -------------------------
```

1. モジュール名の記述部
   モジュールの名前。C 言語の関数名に相当し、他のモジュールから呼び出されるときに用いる。
2. 入出力ポートの記述部
   モジュールに対して入出力される信号の一覧。C 言語の引数に相当するが、C 言語と異なり出力信号も記述する。
3. パラメータ、レジスタ、ネットの指定の記述
   モジュール内で用いる変数名とそのビット幅を指定する。
4. 下位モジュールの呼び出しの記述
   他のモジュールを呼び出す。これにより、回路の階層的な設計が可能。
5. 回路の記述
   実際の論理回路の動作を記述する。

*matutani@arc.ics.keio.ac.jp

1.2 数値

Verilog HDL で用いることができる数値表現を以下に示す。

1.2.1 論理値

実際の回路の動作を考えた場合に利用できる値は以下のものがある。

- 0：low
- 1：high
- x：不定値。値が 0 か 1 か定かではない状態
- z：ハイインピーダンス。電源とグランドの両方から切り放された状態。

1.2.2 定数

回路表現に定数を用いる場合には以下のような記述形式をとる。

<ビット幅>'<基数><数値>

- ＜ビット幅＞：数値を 2 進数に換算した場合、何桁分の数字とするのかを 10 進数で指定する。
- ＜基数＞：数値を何進法で表しているのかを示す。
  - b：2 進数
  - o：8 進数
  - d：10 進数
  - h：16 進数
- ＜数値＞：実際の数値を示す。
  - 基数が b のとき：0 もしくは 1 のみ使用可
  - 基数が d のとき：0 〜 9 まで使用可
  - 基数が h のとき：0 〜 9 と a 〜 f が使用可

例 ： 次はすべて 10 進数の 60 を表す（ビット幅は 6 ビット）

```
6'b111100
6'd60
6'h3c
```

これらの記述形式で記述せず、60 などと定数を直接表記した場合、Verilog HDL では 32 ビット 10 進数の値として解釈される。

1.2.3 変数

回路記述に用いることができる変数には以下の 2 つがある。

- レジスタ型：reg
  汎用の変数。記述によりラッチやフリップフロップ等の記憶素子を生成したり、組み合わせ回路を生成する事が可能（後述）。initial 文と always 文の手続き的ブロックで左辺値として使用可能。
- ネット型：wire
  レジスタ間の配線部分を示す。信号の伝達のみを行い、値は保持しない。assign 文でのみ左辺値として使用可能。

1.3 多ビット信号

1.3.1 信号線宣言時のビット幅指定

信号線は何も指定しなければ 1 ビットとなる。多ビットの信号線を用いるためには以下のような記述をする。

```
reg [7:0]   byte;                    // 8 ビット幅のレジスタ型変数
wire [31:0] word;                    // 32 ビット幅のネット型変数
```

1.3.2 多ビット信号からのビット選択

多ビットの信号として宣言した信号線から、必要な信号を選択するには以下のように記述する。

```
reg [15:0] half_word;               // 半語長のレジスタ型変数
wire [7:0] byte0, byte1;            // バイト長のネット型変数

assign     byte0 = half_word[15:8]; // half_word の 15~8bit を byte0 に継続代入
assign     byte1 = half_word[7:0];  // half_word の 7~0bit を byte1 に継続代入
```

1.3.3 レジスタ配列

Verilog 1995 では reg, integer, time 型のみ配列を作ることができる (ネット型はできない)。Verilog 2001 では全ての型で配列をサポートしている。

```
reg [31:0] reg_array [0:255];       // 32x256 のレジスタ配列
```

Verilog 1995 では、配列要素のビット選択ができない。

```
reg_array[100];      // OK
reg_array[100][7:0]; // NG
```

そのため、Verilog 1995 で配列要素のにビット選択するには、以下のようにネット型を組み合わせる必要がある。一方、Verilog 2001 ではそのような制限はない。

```
wire [31:0] sel_data;
wire [7:0]  byte0;

// Verilog 1995
assign      sel_data = reg_array[100];  // 配列の要素の選択
assign      byte0    = sel_data[31:24]; // 任意のビットの参照

// Verilog 2001
assign      byte0    = reg_array[100][31:24]; // 一行で書ける
```

1.4 演算子と演算子の優先順位

Verilog HDL で扱える演算はほとんど C 言語と同じである。

1.4.1 Verilog HDL の演算子

以下に Verilog HDL の代表的な演算子を示す。Verilog 2001 から対応した演算子は (2001) と記してある。

算術演算	
+	加算、プラス符号
-	減算、マイナス符号
*	乗算
/	除算
%	剰余算
**	べき乗 (2001)

ビット演算	
~	NOT
&	AND
~&	NAND
|	OR
~|	NOR
^	Ex-OR
~^	Ex-NOR

論理演算	
!	論理否定
&&	論理 AND
||	論理 OR

リダクション演算	
&	AND
~&	NAND
|	OR
~|	NOR
^	Ex-OR
~^	Ex-NOR

等号演算	
==	等しい
!=	等しくない
===	等しい (x,z も比較)
!==	等しくない (x,z も比較)

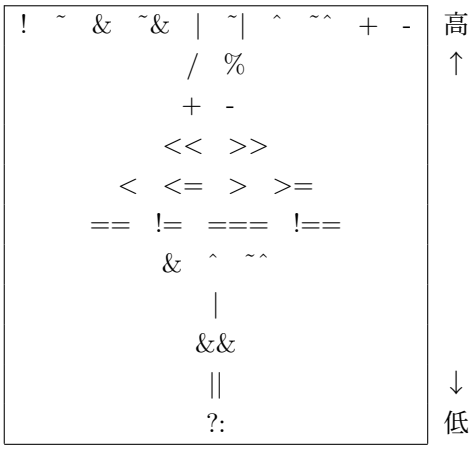
関係演算	
<	小
<=	以下
>	大
>=	以上

シフト演算	
<<	左シフト
>>	右シフト
>>>	算術右シフト (2001)

その他	
?:	条件演算
{ }	連接

1.4.2 演算子の優先順位

以下に演算子の優先順位を示す。



1.5 連結演算

連結演算子 {} を用いることで異なる信号をまとめて扱うことができる。
信号の接続の右辺に用いる場合を以下に示す。

```
wire [31:0] word;
wire [7:0]  byte0, byte1, byte2, byte3;

// byte0~3 を連結して word に継続代入
assign      word = { byte0, byte1, byte2, byte3 };
```

信号の接続の左辺に用いる場合。

```
wire [7:0]  in_a, in_b, out_c;
wire        carry;
```

```
// 加算結果の桁あふれ分が carry に代入される
assign { carry, out_c } = in_a + in_b;
```

1.5.1 繰り返しの記述

連結演算子を用いると同じ数値や信号を繰り返すことができる。

```
{ 4{ 2'b10 } }  →  8'b10101010
```

符号拡張を記述するには

```
wire [15:0] half_word;
wire [31:0] word;

// 上位 16bit を MSB(符合 bit) で拡張
assign word = { { 16{ half_word[15] } }, halfword };
```

1.6 リダクション演算

項の先頭に演算子をつけることで、複数のビット幅を持つ信号の全ビットに対してその演算を実行する。演算結果は 1 ビットの値で得られる。

```
wire [7:0] cnt;

assign     all_and = &cnt; // all_and = cnt[7] & ... & cnt[0]; と等価
assign     all_or  = |cnt; // all_and = cnt[7] | ... | cnt[0]; と等価
```

2 モジュールの記述

モジュールの記述には、モジュール名と入出力ポートを記述する必要がある。

```
// Verilog 1995
module sample_module (clk, reset, data_out);
    input wire clk;
    input wire reset_;
    output wire out;
.
.
endmodule

// Verilog 2001 からサポート
module sample_module (
    input wire clk,     // Clock
    input wire reset_,  // Reset
output wire out
);
.
.
endmodule
```

デフォルトの型は wire なので、wire は省略可能である。

3　論理回路の記述

3.1　組み合わせ回路と順序回路

- 組み合わせ回路
  現在の入力のみで出力が決まる回路。記憶素子を有さない。

- 順序回路
  過去の内部状態と現在の入力で出力が決まる回路。記憶素子を有する。

3.2　assign 文による継続代入

assign 文を用いることで複数の信号を接続することができる。assign 文は、右辺の信号の値が変化すると左辺の信号の値にすぐ代入され、これを継続的代入という。左辺にはネット型の変数が使用できる。

```
wire [31:0] word;
wire [7:0]  byte0, byte1, byte2, byte3;

assign { byte0, byte1, byte2, byte3 } = word;
```

また、三項演算子等を利用して組み合わせ回路を記述することも可能。但し、複雑な記述は推奨しない。

```
wire [7:0] byte0, byte1, byte2, byte3;
wire [7:0] data;
wire [1:0] sel;

assign data = ( sel == 2'b00 ) ? byte0 :
              ( sel == 2'b01 ) ? byte1 :
              ( sel == 2'b10 ) ? byte2 : byte 3;
```

この assign 文は以下の論理を実現する。

sel の値	論理
2'b00	data = byte0
2'b01	data = byte1
2'b10	data = byte2
2'b11	data = byte3

3.3　function 文による組み合わせ回路

function 文を用いると assign 文を用いるよりも複雑な記述で論理式を記述できる。

```
wire    [7:0] data;
wire    [7:0] d0, d1;
wire          sel;

assign data = data_sel( sel, d0, d1 ); // function の呼び出し

function [7:0] data_sel; // function の名前
```

```
    input sel;
    input [7:0] d0, d1;

    begin
       if( sel == 1'b0 ) begin
          data_sel = d0;
       end else begin
          data_sel = d1;
       end
    end
endfunction
```

function 文では、最初に function の名前と出力信号のビット幅を指定する。また、function の中で使用する信号線は、function を呼び出すときに引数として与え、function 文の中では入力信号として最初に指定する。function 文の中では制御構文として if 文や case 文、for 文を用いることができる。

function 文中でのみ一時的に使用する信号線を宣言する場合は、input の宣言と begin の間で、reg 型の信号線を宣言する。

3.4 always 文による組み合わせ回路

always 文を用いると assign 文よりも複雑な記述で論理式を記述できる。具体的には、制御構文として if 文や case 文、for 文を用いることができる。always 文を用いて assign 文の説明で使用した回路を記述すると以下のようになる。

```
wire [7:0] byte0, byte1, byte2, byte3;
reg [7:0]  data;
wire [1:0] sel;

always @( * ) begin
    case ( sel )
        2'b00 : data = byte0;
        2'b01 : data = byte1;
        2'b10 : data = byte2;
        2'b11 : data = byte3;
    endcase
end
```

always 文で組み合わせ回路を記述する際には、`always @( * ) begin ~ end` で制御構造を記述する必要がある。（但し、Verilog 1995 では * をサポートしていないため、@() の中に使用する全ての信号線を記述する必要がある。）

気をつけなければいけない点は、data 信号を reg 宣言してる事である。これは、always 文中では左辺値に wire 型をとれないからである。組み合わせ回路として always 文を記述した場合、レジスタ型の変数によって生成されるのは組み合わせ回路であり、記憶素子は生成されない。（つまり、Verilog では、reg 型だからと言ってレジスタ（Flip-Flop）が生成されるとは限らない。）

但し、always ブロック内で、case 文でフルケース（全パターン）が記述されていない場合や if 文で else が記述されていない場合に、ラッチが生成されてしまうので注意する必要がある。

```
always @( * ) begin
    case ( sel )
        2'b00 : data = 4'h1;
        2'b01 : data = 4'h2;
        2'b10 : data = 4'h4;
```

```
    endcase // sel が 2'b11 の場合、前回の値を出力
end
```

上記の例では、sel=2'b11 の場合に前回の値を保持する必要があるので、ラッチが生成されてしまう。(sel が 2'b11 の場合、data の値の決定には前回の値が必要→過去の状態を記憶しておく必要がある→ラッチを生成)

3.5 always 文による順序回路

always 文を用いることで、順序回路を記述することもできる。

```
module sample_module (
    input wire clk,     // Clock
    input wire reset_,  // Reset
    ....
);

   reg [1:0]   state;   // State

   /********** Sequential Logic **********/
   always @( posedge clock or negedge reset_ ) begin
       if( reset_ == 1'b0 ) begin // Reset
           state <= 2'b00;
       end else begin // Logic
           case( state )
               2'b00 : state <= #1 2'b01;
               2'b01 : state <= #1 2'b10;
               2'b10 : state <= #1 2'b11;
               2'b11 : state <= #1 2'b00;
           endcase
       end
   end

endmodule
```

always 文では

```
always@( posedge clock or negedge reset_ ) begin
```

という文の (posedge clock or negedge reset_) の部分で値を代入するタイミングを指定する。この文の意味は clock 信号の立ち上がり (0 から 1 へ変化) のときか、reset_ 信号の立ち下がり (1 から 0 への変化) のときに

```
always@( posedge clock or negedge reset_ ) begin ... end
```

の begin と end で囲った記述が実行されるという意味である。

- posedge : 信号の立ち上がりに同期させる
- negedge : 信号の立ち下がりに同期させる

複数の信号に同期させる場合はそれらを"or"(Verilog 2001 では "," が使用可能) で結ぶ。always 文の中でも、if 文や case 文、for 文などを使うことができる。assign 文や組み合わせ回路の記述では"=" を用いて値を代入したが、レジスタ宣言した場合は"<=" を用いて値を代入することに注意。
(これをノンブロッキング代入という)
順序回路における代入文では、"<=" の後に"#1" と記述することを推奨する。

3.6 制御構文

以下に示す制御構文はモジュール中に直接記述することはできず、always 文の中で用いる。

3.6.1 if 文

if 文は指定された条件が真ならばその直後の記述を実行する回路を生成する。

```
if ( 条件式 ) begin
    // 条件式が真ならば実行
end
```

else 文、else if 文も使用可能。

```
if ( 条件式 1 ) begin
    // 条件式 1 が真ならば実行
end else if ( 条件式 2 ) begin
    // 条件式 1 が偽でかつ条件式 2 が真ならば実行
end else begin
    // 条件式 1、2 共に偽ならば実行
end
```

3.6.2 case 文

case 文は単純な信号の値を参照して動作を決めるような場合に、if 文よりも簡潔に回路を記述することができる。C 言語のように break 文は不用。

```
case ( 変数 )
    定数 1 : begin
        // 変数が定数 1 の場合に実行
    end
    定数 2 : begin
        // 変数が定数 2 の場合に実行
    end
    ....
    default : begin
        // 変数がどの定数とも等しくない場合に実行
    end
endcase
```

case 文では 0, 1, x, z をそれぞれ別の値とみなして区別するが、x や z を don't care として扱う casez 文と casex 文がある。

- case: 0, 1, x, z を区別して判定する。
- casez: 0, 1, x を区別して判定する。z, ? は don't care とする。
- casex: 0, 1 を区別して判定する。x, z, ? は don't care とする。

case 文で複数の記述が真になる場合は、先に記述したものが優先される。

以下に case 文と casez 文の使用例を示す。

```
case ( in0 )
    1'b0: 式 // in0 が 0 の場合に実行
    1'bz: 式 // in0 が z の場合に実行
endcase // in0 が 1, x のときは何もしない

casez ( in1 )
    3'b000: 式 // in1 が 000 の場合に実行
    3'b1??: 式 // in1 が 100, 101, 110, 111 のいずれかの場合に実行
    3'b11z: 式 // in1 が 110, 111 のいずれかの場合に実行
    3'b0xx: 式 // in1 が 0xx の場合に実行
endcase // in1 が 00x などの場合は何もしない
```

3.6.3 for 文

for 文を用いると、レジスタ配列などに対して同じような処理を行う場合に簡単に回路を記述できる。イテレータには integer 型を使用する（論理合成はされない)。

```
for( i=0 ; i<256 ; i=i+1 ) begin
    // 繰り返し処理
end
```

- integer は 32 ビットの整数を表す。
- Verilog HDL では C 言語にあるインクリメント (++) やデクリメント (−−) は使えない。

基本的に for 文はレジスタ配列の初期化等に使うだけである。以下に例を示す。

```
reg [31:0]  mem [255:0]; // 32x256 Memory

integer        i;           // Initialize Iterator

/********** Sequencial Logic **********/
always @( posedge clock or negedge reset_ ) begin
    if( reset_ == 1'b0 ) begin // Reset
        for ( i=0; i<256; i=i+1 ) begin
            mem[i] = 32'h0;
        end
    end else begin // Logic
        .....
    end
end
```

最終的に論理合成されて回路になる事を考え、あまり無茶な記述はしないように。。。

3.7 下位モジュールからの呼び出し

回路規模が大きくなってくると、1 つのモジュールにすべての機能を盛り込むことは難しくなる。そこで機能単位に複数のモジュールを作り、それらをさらに回路記述内で呼び出すことにより大規模な回路を記述する。他のモジュールを呼び出すモジュールを上位モジュール、呼び出されるモジュールを下位モジュールと呼ぶ。

例えば、以下のような"adder" というモジュールがあるとする。

```
module adder (
    input wire  ina,  // Input A
    input wire  inb,  // Input B

    output wire out  // Output
);

    assign      out = ina + inb;

endmodule
```

このモジュールを他のモジュールから呼び出す場合は以下のようになる。

```
adder adder (     // Module_Name Instance_Name (
    .ina ( ina ), // Port Connection : Input A
    .inb ( inb ), // Port Connection : Input B

    .out ( out )  // Port Connection : Output
);
```

ポートの接続は、
”. 呼び出しモジュールのポート名 (接続する信号名) ”
として行う。

4 回路のシミュレーション

4.1 テストベクトル

C 言語とは異なり、Verilog HDL で記述した回路はそれ単体ではシミュレーションを行うことができない。外部より信号を与えることで記述した回路がはじめて動作する。

そこでテストベクトルというものを用意する。テストベクトルも Verilog HDL で記述されたモジュールあるが、テスト用の信号を生成するための記述がなされている。以下に例を示す。

```
‘timescale 1ns/1ps

module test;
    reg        clk, reset_;
    reg [7:0]  in_data;
    wire [7:0] out_data;

    parameter  STEP = 10;

    always #( STEP/2 ) begin
        clk <= ~clk;
    end

    ff ff0 (
        .clk      ( clk ),
        .reset_   ( reset_ ),
        .in_data  ( in_data ),
```

```
        .out_data ( out_data )
    );

    initial begin
        #0 begin
            clk     <= 1'b1;
            reset_  <= 1'b0;
            in_data <= 8'h00;
        end
        #(STEP) begin
            in_data <= 8'h0f;
        end
        #(STEP) begin
            in_data <= 8'hf0;
        end
        #(STEP) begin
           $finish;
        end
    end

    initial begin
       $shm_open();
       $shm_probe("AC");
    end

endmodule
```

- ‘timescale はシミュレーションで用いる単位時間を指定する。この場合シミュレーション中の 1 単位時間が 1ns、シミュレーションでの誤差が 10ps になることを示す。
- parameter は、そのモジュール内で扱える定数を宣言する。この場合、STEP を 10 として扱っている。
- always 文で#を使うと、引数に指定した単位時間分だけシミュレーション時間が経過すると記述を実行する。この場合パラメータ STEP の値が 10 であり、1 単位時間が 1ns であるので、5ns 経過するとレジスタ clk の値が反転する。こうすることで 100MHz のクロックを生成している。
- テストするモジュールはテストベクトルの下位モジュールとして呼び出す。
- initial 文は 1 回だけ実行される部分で、シミュレーションでの時間経過をここに記述することが多い。
- #0 は時間 0 を表すので、レジスタの初期化を行っておくと良い。
- #(STEP) と記述すると、STEP 時間経過してから次の記述を実行する。
- $display で改行付き printf、$write で改行無し printf を行える。
- $finish; はシミュレーションの終りを示す。これを記述しないとシミュレーションが終わらない。
- $shm_open と$shm_probe() は組み込みタスクで、これらを記述しておくと回路の信号の時間変化を表示する波形ファイルを生成する。この波形ファイルは simvision で波形を見るときに必要となる。
  - $shm_open(“出力ファイル名”) としても良い。

4.2 Icarus Verilog でのシミュレーション

論理回路の記述が終わり、テストベクトルを用意したら、設計した回路の動作をテストする。テストベクトルのファイル名を add_test.v、回路のファイル名を add.v とすると、

```
% iverilog add_test.v add.v
```

とコマンドを打つと Verilog HDL ソースのコンパイルを行う。もし、複数のモジュールを必要とするなら、必要なモジュールが書かれているファイル名を順次追加していく。テストベクトルは 1 番最初に指定すること。実際のシミュレーションは、

```
% vvp a.out
```

とコマンドを打つと実行できる。
以下簡単なオプションの説明。

- -h
  ヘルプ
- -o ファイル名
  出力ファイル名を指定（デフォルトでは a.out）
- -I パス
  インクルードパスに追加するディレクトリを指定
- -D マクロ名 (=値)
  define を追加. 値はなくてもよい

4.2.1 波形の表示

テストベクトルにシミュレーションを行った時の信号変化を波形として生成するような記述 (例えば$dumpfile() と$dumpvars() の 2 つなど) を書いておくと、シミュレーションの進み具合を観測することができる。

```
% gtkwave &
```

GTKWave は波形表示ツールである。
以下簡単な使い方。

- 波形ファイルのオープン
  File → Open New Tab
  VCD ファイル名（例えば dump.vcd）をオープン
  モジュールの階層が表示される
- 波形の表示
  見たい信号を選択して、右上の波形の形をしたボタンをクリック