2013/06/17

課題2 「論理回路とVerilog-HDL」

ハードウェア記述言語 Verilog-HDL によって論理回路の設計を行い、シミュレーションによって動作の確認と論理レベルでの遅延時間の評価をする。HDL は Hardware Description Language の略。1 章で組み合わせ回路、2 章で順序回路の取り扱いについて調べ、3 章で加減算回路の設計と評価をする。

Verilog-HDL によって設計をするときには、論理回路をそのまま記述する構造記述と、回路の振る舞いをプログラムのように記述する動作記述を、場合に応じて使う。この実験では、回路を構造記述で書き、シミュレーションの手続きを動作記述で書く。

1 組み合わせ回路

図 1 は、Verilog による一致検出回路 eq の記述 (1~8 行目) と、シミュレーション eqSim の記述 (10~24 行目) を 1 つの仕事にまとめたものである。

```
 1  module    eq(s, a, b);  /* 一致検出回路 */
 2    input   a, b;
 3    output  s;
 4    wire    na, nb, s1, s2;
 5    assign  #5   na = ~a, nb = ~b;
 6    assign  #10  s1 = a & b, s2 = na & nb;
 7    assign  #10   s = s1 | s2;
 8  endmodule
 9
10  module    eqSim;        /* 一致検出回路      */
11    wire    s;            /* のシミュレーター  */
12    reg     x, y;
13    eq      g1(s, x, y);
14    initial
15      begin
16        $monitor(" %b %b %b", x, y, s, $stime);
17        $display(" x y s                time");
18             x=0; y=0;
19        #50  y=1;
20        #50  x=1; y=0;
21        #50  y=1;
22        #50  $finish;
23      end
24  endmodule
```

図 1: 一致検出回路 eq.v の記述

この記述を、eq.v のように、うしろに .v のついた名前のファイルに用意し iverilog eq.v などとして実行形式[1] への変換をおこない、出力された a.out を実行するとシミュレーションが行われ結果が出力される (図 3 の (c) を参照)。

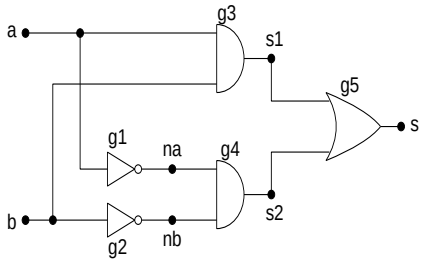

図 2: 一致検出回路 eq　(a) 論理回路図

x	y	s
0	0	1
0	1	0
1	0	0
1	1	1

```
x y s       time
0 0 x          0
0 0 1         25
0 1 1         50
0 1 0         75
1 0 0        100
1 1 0        150
1 1 1        170
```

図 3: 一致検出回路 eq　(b) 真理値表 (左図)　(c) シミューレーション出力 (右図)

1.1 モジュール

図 1 の eq と eqSim は、いずれも、基本になるゲート回路などを使って、モジュールとして表されている。モジュールの書式は、

```
module  モジュール名 (入出力端子のリスト);
  変数の宣言;
  内部の構造の記述;
endmodule
```

である。見出しのモジュール名は、他の名前と同様に、英文字で始まる英文字と数字の文字列で表す。入出力端子のリストは。入出力端子につけられた変数の名前を ", "で区切ったリストである。
コメントは、 `/*` と `*/` で囲まれる部分をコメントと

[1] 実際にはシミュレータへの入力ファイル

して使うことができる。

(a) 変数の宣言

変数の宣言は、入力端子、出力端子、レジスター、内部端子のそれぞれの型と、その型の変数の名前のリストを列挙する。

(a-1) 入力端子

`input` 変数名のリスト；

(a-2) 出力端子

`output` 変数名のリスト；

(a-3) 内部端子

`wire` 変数のリスト；

(a-4) レジスター

`reg` 変数名のリスト；

変数は基本的に 2 進であり、1 ビットの情報を表す。変数のとる値は、0 と 1 であるが、値が不確定の場合は `x` と表される。

(b) 内部構造の記述

システム内には基本論理演算子として、and, or, not などの論理演算子があらかじめ定義されており、回路はこれらの演算子を使って構成される。回路内にある各端子の “接続情報” は

`assign` 遅延時間 代入文，. . . ，代入文；

の書式で列記する。代入文は、

出力端子（または内部端子）= 論理式

の形式で書かれる。右辺は、演算記号 `~`(not), `&`(and), `|`(or) を使った, 入力端子と内部端子にたいする論理式である。遅延時間は対象とする論理式をゲートで実際に実現する際の遅延時間[2] を記述する。同じ種類の同じ遅延時間を持つ代入文は ”, ”で区切って続けて書いてよい。遅延時間は ”#正の整数 ” によって表すが、 書かなければ 0 になる。入力端子と出力端子にはゲートに接続される端子などを表す変数を書く。図 1 の組み合わせ回路 eq の記述においては、もちろん、回路の接続情報を以下のように、1 つの式にまとめることができる。

```
assign  s = ( a & b ) | ( ~a & ~b );
```

このとき、代入文も内部端子も少なくてすむが、遅延時間の指定はアサイン文にたいしてのみ可能で、個々の基本論理演算 (and, or, not 等) を実現する際のゲートの遅延時間の指定はできない。

1.2 シミュレーション

図 1 では、10~24 行目にあるモジュール eqSim がシミュレーションの手続きを記述している。この例もそうであるが、一番外側にあるモジュールには入出力端子はない。そして、 eqSim の記述は

```
module モジュール名;
  変数の宣言；
  内部の構造の記述；
  制御の記述；
endmodule
```

となっている。

(a) レジスター変数

入出力変数と内部端子変数が受け取った値をそのまま伝えるのにたいして、レジスター変数（12 行目）は値を保持する能力を持っている。すなわち、代入文などで値が決まると、次に代入が行われるまでその値を取り続ける。レジスター変数は、本来、動作記述でレジスターを表現する目的で導入されたのであるが、eqSim では、テストされる回路を制御する信号を与えるのに便利であるために使われている。

(b) モジュールの利用

eqSim の内部は、直前に定義されたモジュール eq を使って作られている。一般に、モジュールは、それまでに定義されたモジュールを使って定義される。eq の定義に使われたゲートは、システム内であらかじめ基本要素として定義されたモジュールと考えることができる。モジュールを使うときの書式は、

モジュール名 識別名（変数のリスト）;

である。識別名は必ず書かなければならない。

回路は、モジュールを使ってモジュールを定義することを繰り返して、階層的に設計するのがよい。

(c) 制御の記述

eqSim の 14~23 行目の initial に始まる initial 文は、構造の記述の中でどこからも信号を受けていないレジスター変数に信号を与え、それにたいする回路の動作を表示する処理を記述している。その書式は、

```
initial
  begin
    複数の文
  end
```

である。単一の文で構成されているときには、begin と end はなくてよい。文には、代入文、条件文、繰り返し制御文などがあり、プログラミング言語 C と似た書式で書くことができる。文は書かれた順に 1 つずつ実行されるが、時間の制御を行う指令などを挿入することができる。文の書式は、基本的に、

遅延時間 動作の内容；

である。遅延時間は、ゲートの場合と同様に、#整数 に

[2] 通常、ゲートへの入力が確定してからそのゲートが正しい出力値に落ち着くまでには時間遅れが生じる。これをゲートの遅延時間と呼ぶ。

よって表す。動作の内容を記述する機能は多いが、ここでは、eqSim に含まれているものについて説明する。

(c-1) 代入

代入文の書式は

変数名 = 式；

である。

(c-2) 動作の監視と表示

```
$monitor("書式"、 変数のリスト)；
```

このコマンドは、変数のリストに書かれている何れかの変数の値が変化するとリストにあるすべての変数の値を書式にしたがって表示する。書式は、変数の表示、図形文字、スペースなどで、c と同様の方法で表す。ただし、変数の表示形式は、`%b`(2 進), `%d`(10 進), `%h`(16 進) であり、桁数の表示はない。`$stime` を変数のリストの任意の位置に挿入すると、そのときの時刻が 20 桁の整数として表示される。`$stime` については、書式に書かなくてよい。

(c-3) 表示コマンド

```
$display("書式"、変数のリスト);
```

直前の文の実行直後の変数の値を書式にしたがって表示する。改行をしたくないときは、`display` を `write` にする。

(c-4) 終了指令

```
$finish
```

シミュレーションを終了させる。

(c-5) 定数の定義

同じ値の定数を使うとき、それにあらかじめ名前をつけることができる。定義は

```
‘define 名前 整数
```

によって行い、式の中で

```
‘名前
```

によって引用する。たとえば、図 1 の記述の最初に

```
‘define  w  50
```

と定数 `w` を定義すると、initial 文の中の `#50` は、`#‘w` と書くことができる。ゲートなどの遅延時間などは、ゲートの種類ごとに定数として最初に定義しておき、適宜変更してシミュレーションを行うと便利である。とくに、遅延時間を 0 にしたときの結果は、論理的な検証に有用である。

1.3 変数の有効範囲

図 1 において、モジュール eqSim にはモジュール eq が使われている。このとき、内側の eq で宣言された変数は、外側の eqSim からは直接は見えず、逆に外側の eqSim で宣言された変数は eq からは見えない。したがって、同じ名前を別のモジュールで異なる変数として使うことができる。使用したモジュールの内部の変数を外側から見るためには、モジュールの識別名を変数名の前に . をはさんでかく。たとえば、eqSim から eq の中の s1 端子は g1.s1 によって引用することができる。

問題 1 図 1 のゲートの遅延時間を 0 にしてシミュレーションを行い、その結果を図 3 (c) と比較せよ。それぞれのシミュレーション結果について、各変数の時間にたいする変化を図に表し、違いを明確にせよ。また、eq の内部端子の変化状況が分かるように表示してみよ。□

問題 2 変数 a, b, c, d を受け取り、$a = b$ と $c = d$ がともに成り立つとき出力 s を 1 に、それ以外のとき s を 0 にする回路のモジュールを、図 1 のモジュール eq を 2 個使って作れ。このとき、シミュレーション手続きから eq の内部端子はどのように参照すればよいか。適当なシミュレーションをおこない、実行例とともに示せ。□

問題 3 設計した組み合わせ回路が正しく動作することを検証するためには、シミュレーターをどのように作ったらよいかを考えよ。そして、レポートにて説明せよ。□

2 順序回路

同期式論理回路の基本回路は、クロックとクロックに同期して値を変える (同期式) フリップフロップであり、これらとゲート回路を使って回路を作る。

2.1 クロック

クロックは、図 4 に示すような、一定の周期 W でパルスを送り出す回路である。実際の回路では水晶発振器で実現するのであるが、シミュレーションには擬似的なクロックを使う。図 5 は動作記述で書いたクロックである。このモジュールは、時刻 0 に出力端子 clock を 1 とし、その後 50 単位時間経過するたびに clock の値を反転する。したがってパルスの周期 W は 100 単位時間になる。

動作を記述しているのは、4 行目の initial 文と 5 行目にある連続実行文である。その書式は

```
always 文
```

あるいは

```
always  begin  複数の文  end
```

であり、文 (のリスト) の実行を繰り返す、ことを意味する。これは、電源が ON であると回路は常に動いていることを表している。また、~ は否定 (not) を表す。

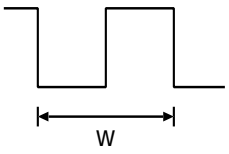

図 4: クロック　(a) 出力信号

```
module   clk( ck );
  output   ck;
  reg      ck;
  initial  ck = 1;
  always   #50  ck = ~ck;
endmodule
```

図 5: クロック　(b) モジュール

2.2 フリップフロップ

図 6 は同期式フリップフロップのモジュールである。これは TTL のエッジトリガー D 型フリップフロップに対応する。動作は、

```
Q'=D
```

である。すなわち、あるクロック時間の入力D の値が、次のクロック時間に出力Q から得られる。このモジュールの動作は動作記述で書かれており、クロックが 1 から 0 に変わるときに入力端子 D の値を記憶し、その値を指定された遅れ時間 (10 ユニット時間) 後に出力端子Q から出力する、ことを示している。この動作は、7 行目にあるイベント制御文が表している。イベント制御文の書式は

```
@( 変化、 変数名 )  文
```

である。変化は、posedge(正のエッジ) と negedge(負のエッジ) の何れかであり、それぞれ、0 から 1 への変化と 1 から 0 への変化を意味する。そしてイベント制御文は、変数に指定された変化が起きたら、文を実行する。すなわち、変数の指定された変化に同期して文の示す動作が実行される。

図 7 は、フリップフロップ dffn のシミュレーションを行なうモジュールである。このとき、D への入力は、クロックの negedge に同期して変化させる。

```
module   dffn( Q, D, ck );
  input    D, ck;
  output   Q;
  reg      Q1;
  initial  Q1 = 0;
  assign   #10 Q = Q1;
  always   @( negedge ck )  Q1 = D;
endmodule
```

図 6: フリップフロップ

```
module   dffnSim;
  reg      x;
  wire     q, ck;
  clk      ck1(ck);
  dffn     g1(q,x,ck);
  initial
    begin
      $monitor(" %b %b %b ",q,x,ck,$stime);
      $display(" q x ck      time");
      x = 0;
      @( negedge ck )  #5  x = 1;
      @( negedge ck )  #5  x = 0;
      #150;  $finish;
    end
endmodule
```

図 7: フリップフロップのシミュレーション

2.3 記憶回路 (1 ビットのレジスター)

1 ビットの記憶回路は、入力端子として記憶指令 l とデータ入力 d、それに状態 q を持つ回路であり、クロックに同期して以下のように動く：

$l = 0$ ならば $q' = q$

$l = 1$ ならば $q' = d$

この回路の論理式は $q' = \overline{l} * q + l * d$ であり、それを実現したモジュールが図 8 である。

```
module  r1(q, l, d, ck);
  input   l, d, ck;
  output  q;
  wire    nl, s1, s2, d1;
  dffn    f( q, d1, ck );
  assign  #5   nl = ~l;
  assign  #10  s1 = nl & q, s2 = l & d;
  assign  #10  d1 = s1 | s2;
endmodule
```

図 8: 1 ビットの記憶回路

2.4 逐次制御回路

表 1 に示す状態遷移表に従って動く回路 m1 を考える。m1 は、停止状態 0 にあるとき入力 a から 1 を受けると、つづく 3 クロックの間出力 b から 1 を出力して再び状態 0 に戻る。

$a \setminus s$	0	1	2	3
0	0/0	2/1	3/1	0/1
1	1/0	2/1	3/1	0/1

表 1: m1 の状態遷移表 s'/b (次状態/出力)

m1 の 4 つの状態、0,1,2,3 を 2 ビットの符号 ($s1$,$s0$) によって (0,0), (0,1), (1,0), (1,1) と表すと、この回路は以下に示す論理式で表される。

$s1' = \overline{s1} * s0 + s1 * \overline{s0}$,

$s0' = a * \overline{s0} + s1 * \overline{s0}$,

$b = s1 + s0$.

これをモジュールとして実現したものが図 9 である。

```
module  m1( b, a, ck );
  input   a, ck;
  output  b;
  wire    ns1, ns0, s1, s0, d1, d0, c1, c2, c3;
  dffn    f1( s1, d1, ck ), f2( s0, d0, ck );
  assign  #5   ns1 = ~s1, ns0 = ~s0;
  assign  #10  c1 = s1 & ns0, c2 = ns1 & s0,
               c3 = a & ns0;
  assign  #10  d1 = c1 | c2, d0 = c1 | c3,
                b = s1 | s0;
endmodule
```

図 9: モジュール m1

このモジュールでは、次状態 $s1'$, $s0'$ をそれぞれ dffn の入力とし、状態 $s1$, $s0$ をそれぞれ dffn $f1$, $f2$ に保持している。各クロックの立ち下がりで状態遷移がおこなわれるようにしている。

問題 4 表 2.2 に示す状態遷移表に従って動く回路 m2 の状態遷移図を書き、設計せよ。さらに、モジュールを書き、シミュレーションで動作を確かめよ。ただし、モジュール内では reg を使わず、dffn を用いて状態を保持せよ。シミュレーションでは、クロックと入力とのタイミングについて注意せよ。そのシミュレーション結果の何をもって動作確認としたのかも説明せよ。回路 m2 は、初期状態 0 にあるとき入力 a から 1 を受けとり、さらに、3 回 1 を受けると出力 b から 1 を出力し、初期状態 0 に戻る。

$a \setminus s$	0	1	2	3
0	0/0	1/0	2/0	3/0
1	1/0	2/0	3/0	0/1

表 2.2. m2 の状態遷移表 s'/b (次状態/出力)

3 加減算回路

この章では、与えられた2つの整数を指示にしたがって加算あるいは減算を行い、その結果を出力する加減算回路を設計する。

3.1 数の表現

演算回路で取り扱う数は2進整数であり、ビットの配列、あるいは、ビットのベクトルで表される。

(a) 定数

定数は、2進、8進、10進、16進で表すことができる。基本の形式は、

桁数　基数　数値

である。桁数は10進整数、4つの基数は、それぞれ、'b, 'o, 'd, 'h で表し、数値はそれぞれ の基数の表現の数字で表す。1ビットの場合と 10 進数の場合には、数値をそのまま書いてよい。たとえば、4'b1 は4ビットの2進数 0001 を表す.

(b) 配列

変数を宣言するときに変数のリストの前に範囲の指定をすると、それらの変数は指定された範囲の添字で指定されるビットの配列となる。範囲は、[最上位の添字、最下位の添字]、の書式で指定する。たとえば、

```
input  [3: 0]  x, y;
```

によって4ビットの変数 x と y が宣言される。式の中などで配列全体を1つの数として使うときはその配列名をそのまま x あるいは y などと書き、一部を使うときは x[3: 1] のように変数名に範囲を続けて書く。1ビットのときは x[2] のように添字で表す。

(c) ビットベクトル

n ビットの2進数は、n 組の1ビットの2進数で構成されているビットベクトルと考えることができる。ビットベクトルは、その構成要素のリストを { と } でかこんで表す。たとえば、x[3: 0] は、{x[3], x[2], x[1], x[0]} あるいは {x[3], x[2: 0]} と同じである。また、x[3: 0] の左シフトの結果は {x[2: 0], 0} 、右算術シフトの結果は {x[3], x[3: 1]} と書ける。

3.2 ビットベクトルを用いた接続情報の記述

(a) ビットベクトルにたいする論理式

assign 文の論理式に現れる変数は、ビットベクトルであってもよい。このとき、論理演算は各ビット毎に行われる。図10 に示す eor4 は、4ビットの入力 x[3: 0] と y[3: 0] の2進和 `{ x[3] & ~y[3] | ~x[3] & y[3], ... , x[0] & ~y[0] | ~x[0] & y[0] }` を z[3: 0] から出力する回路であり、以下のように、1ビットの回路と同様な呼び方で利用できる。

```
eor4  e( g, a, b )
```

(b) パラメータ (parameter)

ビットベクトルの各々のビットに一様な演算を行う回路にたいしては、異なる長さのデータにたいして対応できるように、長さをパラメーターとしてモジュールの定義をすることができる。たとえば、図10 のかわりに、パラメーター宣言を使って、図11 に示す定義をすると、

```
eorn  #(4)  e( g, a, b )
```

によって4ビットの2進和回路を利用できる。モジュールの定義の内部でパラメーターに与えられている値は、利用する文にパラメーターの値の指定がないときに使われるディフォルト値である。

パラメーターは、データの長さだけでなく、遅延時間などにもつかうことができる。また、複数のパラメーターの指定もできるが、利用するときは、# に続けて、指定した順にパラメーターの整数値のリストを書く。

```
module          eor4( z, x, y );
  input   [3: 0]  x, y;
  output  [3: 0]  z;
  wire    [3: 0]  nx, ny, z1, z2;
  assign  #5      nx = ~x, ny = ~y;
  assign  #10     z1 = x & ny, z2 = nx & y,
                   z = z1 | z2;
endmodule
```

図 10: eor4

```
module            eorn( z, x, y );
  parameter         n = 8;
  input   [n-1: 0]  x, y;
  output  [n-1: 0]  z;
  wire    [n-1: 0]  nx, ny, z1, z2;
  assign  #5        nx = ~x, ny = ~y;
  assign  #10       z1 = x & ny, z2 = nx & y,
                     z = z1 | z2;
endmodule
```

図 11: eorn

図 12 に、ビットベクトルとパラメータを用いた応用例を示す。これは、入力 x に not ゲートを n 回直列に接続した結果の y を出力する回路を表している。

以前述べたように、assign 文は接続情報を表すために使われている。各ビットごとに接続情報を書き下せば、

「x が not ゲートを通して w[0] に接続し、

```
module          notn( y, x );
  parameter       n = 4;
  input           x;
  output          y;
  wire    [n-1:0] w;
  assign  #5      { y, w[n-1: 0] } = ~{ w[n-1: 0], x };
endmodule
```

図 12: notn

w[0] が not ゲートを通して w[1] に接続し、...、
w[i] が not ゲートを通して w[i+1] に接続し、...、
w[n-1] が not ゲートを通して y に接続している」
ことを表しているにすぎない。したがって、w という内部端子を表すビットベクトル変数が代入文の両辺に現れていることに矛盾はない。assign 文における代入文が持つ「意味」が、C などのプログラミング言語における場合と異なり、“接続情報” を表していることに注意されたい。

問題 5 (加算回路の設計)　2 つの n ビットの 2 進整数 x, y と桁上げ ci を加え、n ビットの和 s と上位への桁上げ cu を求める n ビットの加算回路は、1 ビットの加算回路である全加算器を n 個使って作ることができる。全加算器は、入力 a, b, c から

桁上げ　`cu = a & b | b & c | c & a`

和　`s = a & ~b & ~c | ~a & b & ~c | ~a & ~b & c | a & b & c`

を求める。図 13 は、全加算器 fa を 4 個使って書いた 4 ビット加算器のモジュール add4 である。このように、全加算器をサブモジュールとして使うと、ビットの長さ n をパラメーターにした記述はできない。ビットのベクトルを使って、n 桁の加算器をひとつのモジュールで実現せよ。このモジュールは add4 のように、最上位の桁上げ cu と、最下位に加える ci を持つこと。さらに、シミュレーションにより評価せよ。　□

```
module   add4( cu, s, x, y, ci );
  input  [3: 0]  x, y;
  input          ci;
  output         cu;
  output [3: 0]  s;
  wire   [3: 0]  c;
  fa             a3(  cu, s[3], x[3], y[3], c[3]),
                 a2(c[3], s[2], x[2], y[2], c[2]),
                 a1(c[2], s[1], x[1], y[1], c[1]),
                 a0(c[1], s[0], x[0], y[0],   ci);
endmodule
```

図 13: 4 桁加算回路 add4

(c) 条件付き assign 文

ビットベクトルにたいする論理演算において、2 つの演算数の長さが違う場合には、最下位の桁を合わせ、短いビットベクトルの上位に 0 を補って演算を行う。このため、ビットベクトル `{ x[n-1], ... , x[0] }` の各ビットと 1 ビットの変数 c の論理積
`{ x[n-1] & c, ... , x[0] & c }` は、条件付き assign 文をつかって書くのが便利である。条件付き assign 文の書式は、

```
assign  変数 = 条件? 条件が 1(真) のときの値:
                     条件が 0(偽) のときの値;
```

であり、条件は、C 言語の場合と同様に、変数、あるいは、変数と定数を比較演算子 `>, >=, ==, <=, <` で比較して得られる。これらの論理積あるいは論理和が必要ならば、 `&&, ||` によって式を書けばよい。

上記の論理積の結果を z に与えるには、

```
assign   z = c? x: 0;
```

例として、2 つの入力 x と y の何れか 1 つを c の値によって選んで出力する, 以下の仕様の選択回路の記述を示そう：

データ入力:　x, y;
制御入力:　c;
データ出力:　z;
動作:　$c = 0$ のとき $z = x$,
　　$c = 1$ のとき $z = y$.

この回路の第 i ビットの出力は、

```
z[i] = x[i] & ~c | y[i] & c
```

で与えられるから、その動作は、たとえば、

```
assign  #5   nc = ~c;
assign  #10  z0 = nc? x: 0,
             z1 = c? y: 0,
             z  = z0 | z1;
```

によって表すことができる。遅延時間を細かく指定しなくてよければ、

```
assign  #25  z = c? y: x;
```

でよい。

問題 6 (n ビットレジスター)　以下に示す仕様のクロックに同期して動作する n ビットレジスターのモジュール r を設計し、シミュレーションにより評価せよ。

データ入力：　d,
制御信号入力：　$load$,
データ出力：　q.
動作：　$load = 0$ のとき $q' = q$,
　　$load = 1$ のとき $q' = d$.

□

3.3 繰り返し動作の記述

Verilog には、for 文などの繰り返し文が備えられていて、動作記述に使うことができる。ここでは、for 文を使ったシミュレーションについて例によって説明する。for 文の書式は、

for（制御変数の初期設定; 継続条件;
制御変数の変更）文;

であり、C 言語の場合と同様に書くことができる。図 14,15,16 は、変数 c が 1 のとき入力 x を、c が 0 のとき 0 を、z から出力するモジュール sel とそのシミュレーションを行うモジュール selSim である。モジュール sel において、入力 c は、データ x を出力するかどうかを決めるという意味 で、制御信号と呼ぶことができる。

```
module    sel( z, x, c );
  parameter         n = 8;
  input   [n-1: 0]  x;
  input             c;
  output  [n-1: 0]  z;
  assign  #10       z = c? x: 0;
endmodule

module    selSim;
  wire    [1:0]  p;
  reg     [2:0]  v;
  sel     #2     g(p, v[1: 0], v[2]);
  initial
    begin
      $monitor("%b  %b", v, p, $stime);
      for ( v=0; v<7; v=v+1 )  #100;
      #200  $finish;
    end
endmodule
```

図 14: (a) モジュール sel と selSim

```
module            sel2( z, x, c );
  input   [1: 0]  x;
  input           c;
  output  [1: 0]  z;
  assign  #10     z[1] = x[1] & c,
                  z[0] = x[0] & c;
endmodule
```

図 15: モジュール sel　(b) 基本論理演算を用いて書いた sel2

入力　x[n-1: 0], c;
出力　z[n-1: 0];
動作　c=1 のとき z=x,
　　c=0 のとき z=0.

図 16: モジュール sel　(c) 動作の説明

問題 7 (加減算回路の設計)　以下に示す仕様の n 桁の加減算回路のモジュールを設計し、シミュレーションにより評価せよ。ただし、負の数は 2 の補数表現で表されているものとせよ。

数の入力 ：　n 桁の整数 x, y；
制御信号入力：　k；
出力 ：　桁上げ cu と n 桁の和 s；
動作 ：　$k = 0$ のとき $s = x + y$、
　　$k = 1$ のとき $s = x - y$。

何れの場合にも、cu は addn からの桁上げを与える。

負数を 2 の補数で表したとき、$x - y = x + \overline{y} + 1$ である。したがって、k の値によっ て y と $\overline{y}$ の何れかを選ぶ選択回路の出力と x を加算すればよく、1 の加算には、桁上げ入力を使う。　□

問題 8 (オプション課題)　各課題の回路について、FPGA ロジックトレーナーと QuartusII を用いて、実際にロジックトレーナー上で動作させ、RTL viewer で表示させた回路図について考察せよ。

参考文献

[1] D.E. トーマス、P.R. モアビー著　飯塚哲也、浅田邦博訳：設計言語 Verilog-HDL 入門、 培風館 (1995).

[2] 小林優：「改訂 入門 Verilog HDL 記述」, CQ 出版社 (2004 年 6 月).

注意

1. verilog 実行時に無限ループ等に入った場合、Ctrl+c → Ctrl+z 後に kill %% を実行する事により、実行を停止できる。