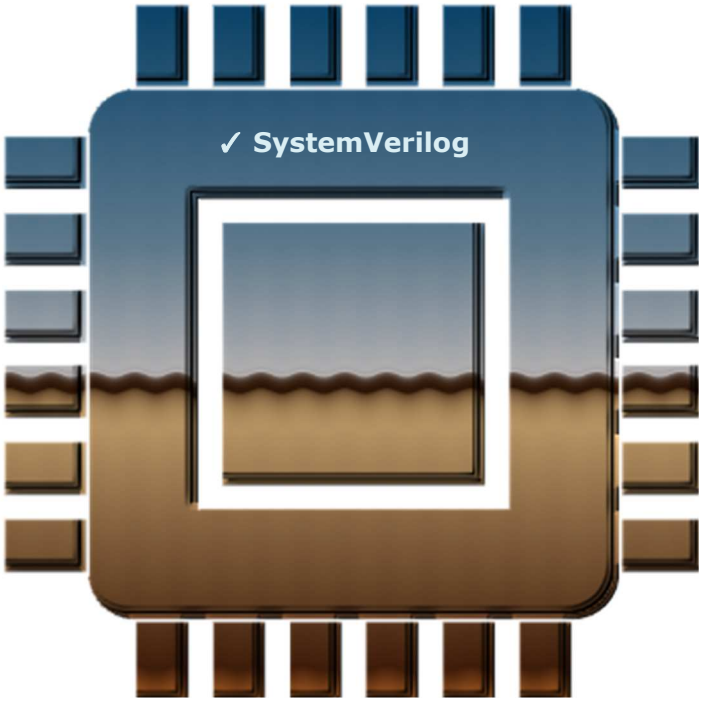
Artgraphics

SystemVerilog によるロジック設計の基礎

Document Identification Number: ARTG-TD-003-2020
Document Revision: 1.0, 2020.03.16
アートグラフィックス
篠塚一也



Logic Design with SystemVerilog

SystemVerilog によるロジック設計の基礎

〒124-0012 東京都葛飾区立石 8-14-1
www.artgraphics.co.jp

Logic Design with SystemVerilog

8-14-1, Tateishi, Katsushika-ku, Tokyo, 124-0012 Japan
www.artgraphics.co.jp

はじめに

SystemVerilogの最新仕様は、2018年2月21日に規格IEEE Std 1800-2017（以降、LRMと略称）として公開され、日本国内でも次第にVerilog HDL（以降Verilogと略称）からSystemVerilogへと移行する技術者の動向も顕著になり始めています。

一方、依然としてVerilogを主言語としているユーザが多い事も事実です。然し、その様なユーザにおいても、殆どの場合、ツール環境はSystemVerilogに移行を完了しています。EDAツールベンダーは既にSystemVerilogに移行しているので、もはやVerilogの開発環境でVerilogが使用されている訳ではありません。通常、ユーザが使用するVerilogは、SystemVerilogのサブセットとして実行しています。従って、Verilogユーザは、SystemVerilogの利点を活用していない状態でEDAツールを使用しているとも言えます。

SystemVerilogは、設計、仕様、検証の何れの作業も同じ言語で記述する事を可能にします。仕様、及び検証に関しては、SystemVerilogはVerilogには存在しない機能で構成されていますが、設計機能に限定するとSystemVerilogとVerilogはほぼ対等に使えると感じる技術者も多いと思います。それ故、Verilogが依然として使用されている理由の一つではないかと思えます。SystemVerilogでしか可能ではない設計機能が出現するまでは、Verilogが設計用の言語として使用され続けるのは合理的だと思います。

本書は、VerilogとSystemVerilogのどちらが良いか等の比較する目的を持ってはいません。目的は、ロジック設計をSystemVerilogでモデリングする際、どの様な記述法が望ましいかを解説する事です。また、それらの記述法を通してSystemVerilogの持つ機能を再確認する機会を提供する事です。従って、本書で紹介しているモデリングをVerilogで記述し直す事により、ユーザの環境に合わせてモデリング知識を適用する事ができます。

本書は、概要を含めて9つの章から構成されています。第1章の概要では、組み合わせ回路とシーケンシャル回路の記述ルールを概説し、第2章以降の指針を設定しています。モデリングルールの基本を総括している意味において、第1章の内容は極めて重要になります。

第2章では、SystemVerilogによるモデリングで必要なSystemVerilogに関する知識を復習します。特に重要と思われる知識に限定しているため、本書のモデリングで使用している機能を全てを復習しているわけではありません。例えば、if文やcase文のシンタックスや機能の解説を省略してあります。また、この章ではVerilogスタイルとSystemVerilogスタイルの記述上での差異も簡単にまとめてあります。

第3章では、データ表現の基本となる整数表現を簡単にまとめてあります。特に、重要な概念である演算オーバフロー検出を解説してあります。ここで、演算オーバフローとは、加算のcarry-outとは全く異なる概念なので注意が必要です。解説した知識を明確にするために、この章には演算オーバフロー検出機能を使用した加減算器のデザインも含まれています。

第4章では、真理値表とブール代数に関しての知識を総括しています。組み合わせ回路を設計する場合、真理値表からブール式を導く必要性も少なからず出て来ると想定して、この章にはブール代数の基本的な知識をまとめてあります。特に、Shannonの展開定理に関係するロジック設計に関する代表的な話題がまとめられています。

第5章は、組み合わせ回路のモデリングを主題にしています。代表的な組み合わせ回路の記述法を詳細に紹介しています。組み合わせ回路をテストするテストベンチのコードと実行結果も含まれています。

第6章は、シーケンシャル回路のモデリングを主題にしています。この章には、各種のカウンター、シフトレジスタのモデリングが紹介されています。これらの記述を通して、シーケンシャル回路の記述法を理解する事ができます。

第7章は、FSMの解説が主題です。Moore FSMとMealy FSMの相違を明確にし、それぞれの

モデリングを詳しく解説しています。この章では、具体的な設計問題に対して、それぞれの FSM でモデリングし、記述法の差異を明確にしています。また、デザインした FSM をテストする際の相違に対しての解説も含んでいます。Mealy FSM は、クロックと非同期に動作するので、Moore FSM と同じ方法でテストをする事ができません。両者のモデリングによる結果を比較し易くするために、テストベンチも両 FSM の差異を意識して作りました。この章のモデリングとテストベンチ記述法を理解する事は、FSM の完全理解に繋がります。

第 8 章は、デザインを検証する際のテストベンチの構築法を解説しています。設計者にしても、実装したデザインを自身で検証する義務があります。組み合わせ回路、シーケンシャル回路、FSM では、検証のタイミングが大きく異なります。それぞれの回路の特性に合わせたテストベンチ構築法が必要です。従来のモジュール方式による検証に加えて、クラスを使用した検証環境の構築例が紹介されています。本章は、SystemVerilog に関する設計機能以外の機能に関する知識を習得する最適な内容です。

第 9 章は、SystemVerilog の基本であるシミュレーション実行のタイミングに関する知識を含んでいます。この章で解説しているスケジューリング領域の概念は、絶対的に必要な知識です。必ず、完全な理解をしてからモデリングを学習する様にして下さい。

また、紙面の都合上、一部の記述は小さな書体で記述されています。

アートグラフィックス
篠塚一也

変更履歴

日付	Revision	変更点
2020.03.16	1.0	初版。

目次

1　概要

1.1　HDL によるハードウェアモデリング

ハードウェア仕様を基にして、ハードウェアを物理的に実現する手段として、HDL が使用されます。自然言語による表現と異なり、HDL による表現は曖昧性の無い記述を導きます。記述された実装内容は、一般的にデザインと呼ばれ、様々な変換過程を経て、ゲートレベルに実現されます。ゲートレベルの表現は、一般的にはネットリストと呼ばれライブラリーセルをネット（配線を示すワイア）で結合した膨大な量の記述表現となります。このレベルは、セルインスタンスをネットで結合しているので、HDL の表現形式の一つと言えますが、実質的には、異なる言語による物理表現とも言えます。本書の目的は、ゲートレベルよりも高位のレベル、具体的には、RTL、及びそれよりも高位のレベルでデザインを表現する事です。本書では、HDL として最もよく知られている SystemVerilog を使用して解説を進めます。

1.2　デザインの表現形式

デザインは実装した内容で、HDL による機能的な記述形式を多く含みます。一方、前述のネットリストは構造的な表現形式であるため、表現自体から機能を理解する事はできません。最も概要的なデザインの表現形式は図 1-1 に示す様なブロックダイアグラムです。HDL によるデザイン記述は、inputs を outputs に変換するための手順を詳細に定義します。

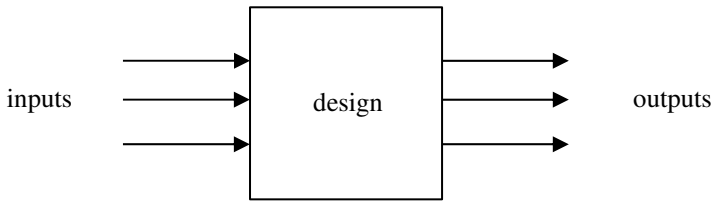


図 1-1　デザインのブロックダイアグラム表現

ブロックダイアグラムにおいては、入力、及び出力が明記され、仕様解説は自然言語で補足されます。仕様を HDL のどのレベルで表現し直すかで、記述の読み易さが大きく変わります。例として、ハーフアダーを仮定します。そのブロックダイアグラムは、図 1-2 の様に書く事ができます。

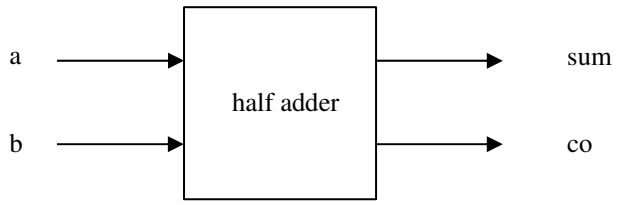


図 1-2　ハーフアダーのブロックダイアグラム

もし、ゲートレベルの表現を選択すれば、以下の様な記述になります。

```
module half_adder(input a,b,output co,sum);
xor(sum,a,b);
and(co,a,b);
endmodule
```

然し、この記述方式はハーフアダーの計算アルゴリズムを理解している事を仮定しています。アルゴリズムに関する知識が無いか、又は記憶が定かではない場合には、この記述を確認するために少なからずの時間を必要とします。そもそも、HDL はハードウェアの実装に近い記述方式を回避するために存在します。上記の記述に間違いはありませんが、HDL という高位レベルの記述能力を十分に利用していないと言えます。もう少し高度な表現法は、以下の様

になります。

```
module half_adder(input a,b,output co,sum);
assign sum = a^b;
assign co = a&b;
endmodule
```

この記述法には多少の改善は見られますが、この記述がハーフアダーの計算アルゴリズムを理解している事を仮定している事実には変わりがありません。もし、計算アルゴリズムの記憶に間違いがあれば、後の検証で不可解な現象の解決に時間を要するだけとなります。寧ろ、以下の様に記述する事が望ましいと言えます。

```
module half_adder(input a,b,output co,sum);
assign {co,sum} = a+b;
endmodule
```

何故望ましいかと云えば、この表現方式であれば論理合成ツールが適切なゲートレベルへの変換をしてくれるからです。さて、もう一つの事例を紹介します。

4 入力の multiplexer をゲートレベルでモデリングすると以下の様な記述になります。and、or、not だけを用いて multiplexer の機能を構築する事ができます。

```
module multiplexer_gate(input [1:0] a,b,c,d,[1:0] s,
          output [1:0] out);
wire   _s1, _s0, wa0, wa1, wb0, wb1, wc0, wc1, wd0, wd1;

not(_s1,s[1]);
not(_s0,s[0]);
and(wa1,_s1,_s0,a[1]);
and(wb1,_s1,s[0],b[1]);
and(wc1,s[1],_s0,c[1]);
and(wd1,s[1],s[0],d[1]);
or(out[1],wa1,wb1,wc1,wd1);
and(wa0,_s1,_s0,a[0]);
and(wb0,_s1,s[0],b[0]);
and(wc0,s[1],_s0,c[0]);
and(wd0,s[1],s[0],d[0]);
or(out[0],wa0,wb0,wc0,wd0);

endmodule
```

確かに、上記の記述は 4 入力の multiplexer を実現していますが、これだけ複雑な構造体になると、機能の把握は自明ではありません。仮に、ネットの接続に間違いがあっても容易に発見する事ができないという問題があります。寧ろ、以下の様に HDL の特性を活用した記述スタイルの方が分かり易いと言えます。

```
module multiplexer_behavior(input [1:0] a,b,c,d,[1:0] s,
       output logic [1:0] out);

    always @(a,b,c,d,s)
       case (s)
       0: out = a;
       1: out = b;
       2: out = c;
       3: out = d;
       default out = 'x;
       endcase

endmodule
```

3 整数表現と演算

3.1 logic 型

SystemVerilog でモデリングする場合、整数を表現するためには logic 型を使用します。logic 型は、標準として unsigned です。符号付き整数を表現するためには、signed という修飾子を付加しなければなりません。例えば、以下の様に logic 型を使用します。宣言の効果は、表 3-1 にまとめられています。

```
logic [7:0]          ua;
logic signed [7:0]   sa;
```

表 3-1　logic 型の宣言

宣言	効果
`logic [7:0]  ua;`	ua は 8 ビットの符号なし整数を表現します。表現できる値は、0~255 となります。符号無しなので、8 ビット全体が値を表現します。最小値は、8'b0000_0000 として表現されます。
`logic signed [7:0]  sa;`	sa は 8 ビットの符号付き整数を表現します。表現できる値は、-128~127 となります。sa の MSB は符号を示し、値の一部ではありません。最小値は、8'b1000_0000 として表現されます。

同じビット表現でも signed と unsigned では示す値が異なる事に注意しなければなりません（図 3-1）。

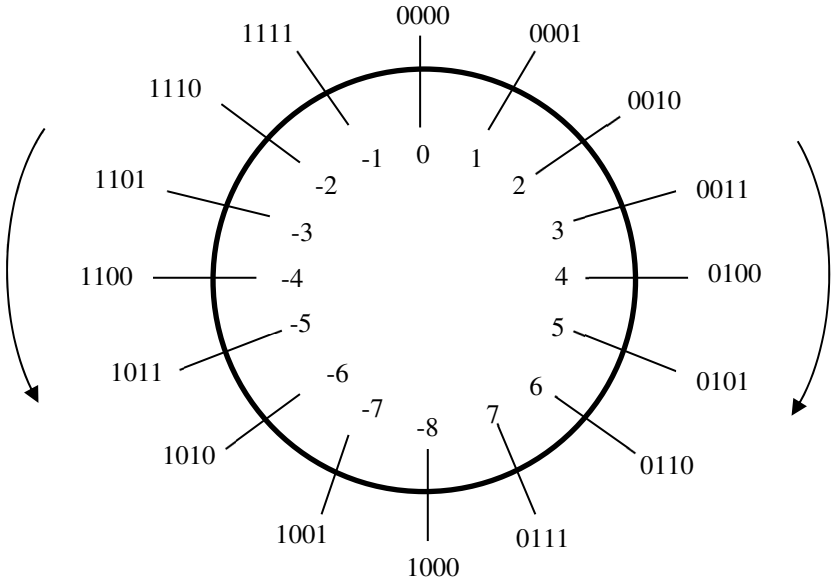


図 3-1　符号なし整数と符号付整数の関係（4 ビット）

4 ビットの整数 N が、{N3,N2,N1,N0}として表現されていれば、-N は、{~N3,~N2,~N1,~N1}+1 として計算する事ができます。これは、2 の補数表現です。例えば、3 は 0011 なので、次の様にして-3 を求める事ができます。

$$\{\sim 0, \sim 0, \sim 1, \sim 1\} + 1 = \{1,1,0,0\} + 1 = \{1,1,0,1\} = -3$$

SystemVerilog でモデリングする場合には、この様な整数の内部表現に煩わされる事が無く、ロジック設計自体に専念する事ができます。但し、整数演算を行う場合には、オーバフロー

に注意する事は必要です。

例 3-1　logic 型を使用した整数宣言例

符号付き整数の宣言効果を確認するための記述例を以下に示します。

```
module test;
logic [7:0]         ua;
logic signed [7:0]  sa;

    initial begin
        ua = '1;
        sa = '1;
        print;
        ua = 8'b0000_0000;
        sa = 8'b1000_0000;
        print;
    end

function void print();
    $display("%b(%0d) %b(%0d)",ua,ua,sa,sa);
endfunction
endmodule
```

ua と sa のビット表現は同じでも表現する値は異なる。

8'b1000_0000 が sa の最小値である事を確認する。

実行すると以下の様な結果を得ます。

```
11111111(255) 11111111(-1)
00000000(0) 10000000(-128)
```

■

実行時に符号なし数を符号付整数に変更する事ができます。そのためには、タイプ変換 signed'(expression)を使用します。その逆の操作は、unsigned'(expression)を使用します。

例 3-2　実行時の符号変換例

以下では、符号なし整数 c を符号付きに変更をしています。

```
module test;
logic [3:0]  a, b, c;

    initial begin
        a = 4'b1000;
        b = 4'b0001;
        c = a+b;
        $display("c=%b(%0d)",c,c);
        $display("signed c=%b(%0d)",signed'(c),signed'(c));
    end
endmodule
```

実行すると、以下の様な結果を得ます。

```
c=1001(9)
signed c=1001(-7)
```

■

時として、負の整数を内部的に作成しなければならない事もあるかと思います。以下に、2 の補数表現例を示しておきます。

4　真理値表とブール代数

与えられたハードウェア仕様をより明確に理解するために、時として、ビット表現のデータを使用して机上の動作シミュレーションが必要になります。或はまた、エンコーダー、デコーダー、カウンター等のデザインにおいては、ビット表現のデータを分析・解析して処理方式を導くための検討をします。この様な時、真理値表とブール代数は大きなツールとなります。本章では、これらの概念を復習整理します。

4.1　真理値表

最初に、表 4-1 の様な真理値表を仮定し、どのような回路を表現しているかを考察します。a、b、s が入力変数で out が出力変数です。

表 4-1　真理値表の例

a	b	s	out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

この真理値表から、以下の様なブール式を得る事ができます。

$$out = a' * b * s + a * b' * s' + a * b * s' + a * b * s$$

この式は、次の様に最適化されます。

$$out = b * s + a * s'$$

この式を観察すると、式が 2 入力の multiplexer を表現している事が分かります（図 4-1）。

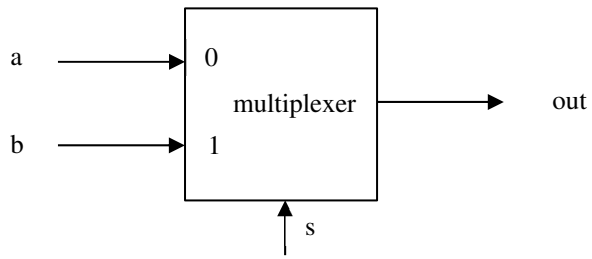


図 4-1　2 入力 multiplexer

真理値表からブール式を導く手順は、以下の様にまとめられます。

① 出力変数が 1 を取る行に対して、入力変数から積を構築する。
② それらの積の和を取る。

この手順で導かれたブール式は SOP 表現と呼ばれ、冗長な表現を含むため最適化が必要となります。そのためには、ブール代数に示されている定理を利用する事が必要になります。上記の最適化では、次の定理を使用しています。

$$x + x' = 1$$
$$x * (y + z) = x * y + x * z$$

複雑な最適化には、カルノー図等のツールが必要になります。カルノー図に関する解説は随所に見られるので、本書では解説を省略いたします。文献[2]には、カルノー図に関する詳しい解説があります。

参考 4-1

2 入力の multiplexer を$f(s,a,b)$と書くと、以下の様に定義されます。

$$f(s,a,b) = s * b + s' * a$$

この基本表現から、以下の等式を得ます。

$f(s,a,1) = s + s' * a = s + a$
$f(s,0,b) = s * b$
$f(s,1,0) = s'$

即ち、AND、OR、INV が multiplexer で構成される事が分かります。従って、全ての組み合わせ回路は multiplexer で表現する事ができる事が分かります。multiplexer は、重要な組み合わせ回路であるだけでなく、組み合わせ回路のビルディングブロックです。
■

以下では、真理値表が有効な手段になり得る事を例を通して紹介します。

4.1.1 等号と真理値表

例 4-1　a==b の真理値表の例

a==b の真理値表は表 4-2 の様になります。

表 4-2　a==b の真理値表

a	b	eq
0	0	1
0	1	0
1	0	0
1	1	1

これらの値から、次の式を得る事ができます。

$$eq = a * b + a' * b'$$

このブール式は、XNOR に他なりません。逆に言えば、 a!=b は XOR で実現する事ができる事になります。
■

4.1.2 ハーフアダーと真理値表

ハーフアダーは、図 4-2 の様な入出力を持ちます。

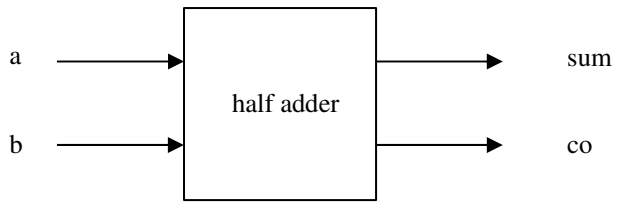


図 4-2　ハーフアダーののブロックダイアグラム

5.7 デコーダー

デコーダーは、データ信号（enable 信号）と n 個の制御信号を入力として持ち、2^n個の出力の一つだけを有効にする組み合わせ回路です。即ち、デコーダーは2^nビットの one-hot コードを生成する回路です（図 5-7）。

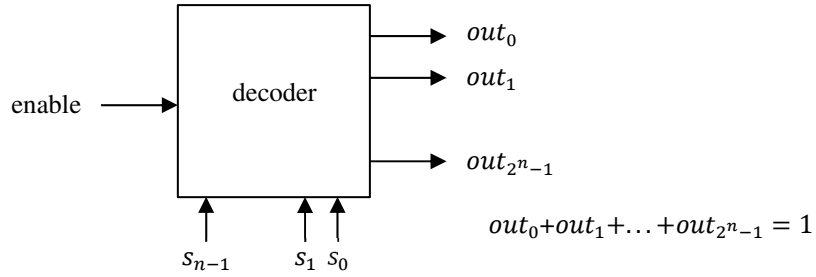


図 5-7　デコーダーのブロックダイアグラム

例 5-6　デコーダーの記述例

8 ビットの one-hot コードを生成するデコーダーを以下の様に記述します。SystemVerilog のルールにより、code は wire で、out は var になります。

```
module decoder(input enable,[2:0] code,output logic [7:0] out);
logic [7:0]  _out;

assign out = enable ? _out : '0;

   always @(code)
      case (code)
      0: _out = 8'b0000_0001;
      1: _out = 8'b0000_0010;
      2: _out = 8'b0000_0100;
      3: _out = 8'b0000_1000;
      4: _out = 8'b0001_0000;
      5: _out = 8'b0010_0000;
      6: _out = 8'b0100_0000;
      7: _out = 8'b1000_0000;
      default:  _out = 'x;
      endcase

endmodule
```

デコーダーのインスタンスを以下の様に作り、デコーダーへの入力 code を順に生成してテストします。

```
module test;
logic [2:0]  code;
logic [7:0]  out;
logic     enable;

decoder DUT(.*);

   initial begin
      $display("      enable code  out");
      enable = 1'b0;
      #10 code = 5;
      enable <= #1 1'b1;
      for(int i = 0; i < 8; i++ ) begin
```

```
            #10 code = i;
        end
    end

    initial forever @(code)
        #0 $display("@%3t: %b      %0d     %b",
            $time,enable,code,out);
endmodule
```

実行すると、enable==1'b1 の時のみ、デコーダーは one-hot コードを生成している事が分かります。

```
       enable code  out
@ 10: 0      5     00000000
@ 20: 1      0     00000001
@ 30: 1      1     00000010
@ 40: 1      2     00000100
@ 50: 1      3     00001000
@ 60: 1      4     00010000
@ 70: 1      5     00100000
@ 80: 1      6     01000000
@ 90: 1      7     10000000
```

デコーダーに関しては、ビヘイビアで記述するよりも、以下の様に式で直接記述する方が、分かり易いかも知れません。この記述法によると、デコーダーの出力が one-hot コードである事が明確になります。

```
module decoder(input enable,[2:0] code,output logic [7:0] out);
assign out[0] = enable&&(code==0);
assign out[1] = enable&&(code==1);
assign out[2] = enable&&(code==2);
assign out[3] = enable&&(code==3);
assign out[4] = enable&&(code==4);
assign out[5] = enable&&(code==5);
assign out[6] = enable&&(code==6);
assign out[7] = enable&&(code==7);
endmodule
```

或は、以下の様に、基本回路に分割して記述する事も出来ます。

```
module decoder(input enable,[2:0] code,output logic [7:0] out);
wire   c0, c1, c2, _c0, _c1, _c2;
assign c0 = code[0];
assign c1 = code[1];
assign c2 = code[2];
assign _c0 = ~c0;
assign _c1 = ~c1;
assign _c2 = ~c2;
assign out[0] = enable&_c2&_c1&_c0;
assign out[1] = enable&_c2&_c1&c0;
assign out[2] = enable&_c2&c1&_c0;
assign out[3] = enable&_c2&c1&c0;
assign out[4] = enable&c2&_c1&_c0;
assign out[5] = enable&c2&_c1&c0;
assign out[6] = enable&c2&c1&_c0;
assign out[7] = enable&c2&c1&c0;
endmodule
```

■

6　シーケンシャル回路

シーケンシャル回路として以下の様な回路の記述例を紹介します。

- アップダウンカウンター
- ラッチ
- JKーフリップフロップ
- データシフタ
- ユニバーサルシフトレジスタ
- Johnson カウンター
- Gray カウンター
- リングカウンター
- Gated clock
- インターフェースを使用するモジュール記述

6.1　シーケンシャル回路のモデリング

シーケンシャル回路のモデリングルールは、以下の様になります。

① always プロシージャを使用して、エッジセンシティブなセンシティビティリストを定義する。
② クロック信号、及び非同期信号をセンシティビティリストに指定する。
③ 非同期信号の判定を always プロシージャの先頭に記述する。
④ 非同期信号のエッジと判定基準は一致しなければならない。
⑤ always プロシージャ内では、ノンブロッキング代入を使用して出力変数に値を設定する。
⑥ クロック信号は、センシティビティリスト以外に使用されてはならない。

以下の記述は、代表的なシーケンシャル回路のモデリングを示しています。

```
module shift_register(input clk,reset,data_in,output q);
logic  q0, q1, q2, q3;

assign q = q3;

    always @(posedge clk,posedge reset)
        if( reset == 1'b1 ) begin
            q0 <= 1'b0;
            q1 <= 1'b0;
            q2 <= 1'b0;
            q3 <= 1'b0;
        end else begin
            q0 <= data_in;
            q1 <= q0;
            q2 <= q1;
            q3 <= q2;
        end

endmodule
```

クロック信号、及び非同期信号をセンシティビティリストに指定する。

posedge reset に合わせて条件判定を行う。

ノンブロッキング代入文を使用して出力変数に値を設定する。

6.2　アップダウンカウンター

この例で示すカウンターには、リセット信号（reset）、ロード信号（load）、アップダウン信号（up）が付いています（図 6-1）。load==1'b1 であれば、指定された値（data_in）をロードします。reset も load もセットされていない場合には、up の状態によりカウンターの内容を更新します。up==1'b1 であれば、カウンターをインクリメントし、そうでなければ、カウンターをデクリメントします。

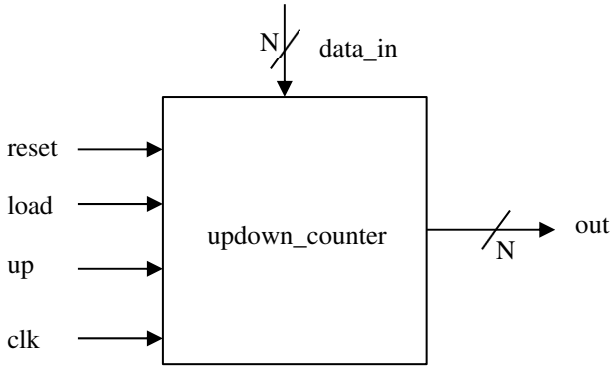


図 6-1　アップダウンカウンターのブロックダイアグラム

例 6-1　バイナリーカウンターの記述例

アップダウンカウンターを以下の様に定義します。

```
module updown_counter #(NBITS=2) (input clk,reset,load,up,
    [NBITS-1:0] data_in,output logic [NBITS-1:0] count);

    always @(posedge clk,posedge reset)
       if( reset == 1'b1 )
          count <= '0;
       else if( load == 1'b1 )
          count <= data_in;
       else if( up == 1'b1 )
          count <= count+1;
       else
          count <= count-1;

endmodule
```

次に、テストベンチを以下の様に定義します。updown_counter はシーケンシャル回路なので、その出力を確認するタイミングには注意しなければなりません。

```
module test;
parameter SIZE=4;
bit       clk;
logic        load, reset, up;
logic [SIZE-1:0] data_in, count;

clocking cb @(posedge clk); endclocking
clocking cbr @(posedge reset); endclocking
updown_counter #(.NBITS(SIZE)) DUT(.*);

   initial begin
      fork
         begin #5 reset = 1; up = 1; reset = #1 0; end
         begin #80 load = 1; up = 0; data_in = 12;
                @cb load = 0; end
      join
   end

   always @cb print();
   always @cbr print(1);
   initial $display("      reset up load data_in count");
   initial forever #10 clk = ~clk;
```

6.4 JKーフリップフロップ

JK-フリップフロップを紹介します。先ず、JK―フリップフロップの機能を表 6-2 の様な真理値表で表現する事が出来ます。

表 6-2 JK―フリップフロップの機能（[2]）

j(t)	k(t)	q(t)	q(t+1)	機能
0	0	0	0	HOLD
		1	1	
0	1	0	0	RESET
		1	0	
1	0	0	1	SET
		1	1	
1	1	0	1	TOGGLE
		1	0	

例 6-4 JK―フリップフロップの記述例

最初に、JK―フリップフロップのオペレーションに必要なコードをパッケージに定義しておきます。

```
package pkg;
typedef enum logic [1:0]
   { HOLD=2'b00, RESET=2'b01, SET=2'b10, TOGGLE=2'b11 } jk_op_e;
endpackage
```

JK―フリップフロップを以下の様に実装する事ができます。

```
`include "pkg_jk_flipflop.sv"

module jk_ff import pkg::*; (input logic clk,j,k,output logic
q,qbar);

assign qbar = ~q;

   always @(posedge clk) begin
      case ({j,k})
      HOLD:     q <= q;
      RESET:    q <= 0;
      SET:      q <= 1;
      TOGGLE:   q <= ~q;
      endcase
   end

endmodule
```

テストベンチでは、JK―フリップフロップに許される操作を順に試してテストします。

```
module test;
import pkg::*;
bit    clk;
logic  j, k, q, qbar;
jk_op_e   op;

clocking cb @(posedge clk); endclocking
jk_ff DUT(.*);

   initial begin
```

```
        $display("     j  k  q  qbar  op");
        fork
            {j,k} = RESET;
            #20 {j,k} = HOLD;
            #40 {j,k} = SET;
            #60 {j,k} = TOGGLE;
        join_none

        forever begin
            @cb;
            $cast(op,{j,k});
            $display("@%2t: %b  %b  %b  %b     %s",
               $time,j,k,q,qbar,op.name);
        end
    end
    initial forever #10 clk = ~clk;
    initial #100 $finish;
endmodule
```

実行すると以下の様な結果を得ます。

```
      j  k  q  qbar  op
@10: 0  1  0  1     RESET
@30: 0  0  0  1     HOLD
@50: 1  0  1  0     SET
@70: 1  1  0  1     TOGGLE
@90: 1  1  1  0     TOGGLE
```

■

6.5 データシフタ

ここでは、以下の様な機能を持つデータシフタを紹介します（図 6-3）。

- 非同期なリセット機能を持つ。
- load の指定により、外部からデータをロードする。
- クロックに同期してレジスタの内容を左右にシフトする。sl==1'b1 であれば、左に 1 ビットシフトし、sr==1'b1 であれば、右に 1 ビットする。sl と sr が 1'b1 ではない場合には、レジスタの内容を維持する。

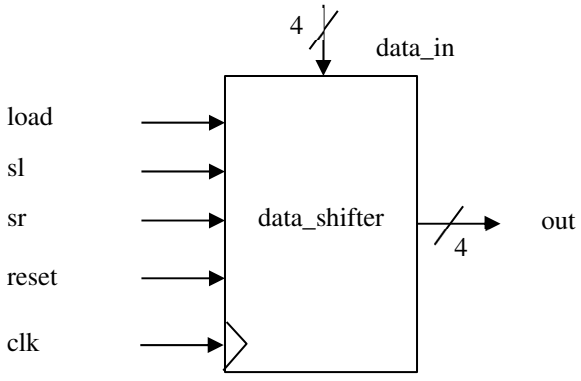


図 6-3　データシフタのブロックダイアグラム

例 6-5　4 ビットのデータシフタの記述例

4 ビットのデータシフタは以下の様になります。

7.2 FSM の種類

FSM では、入力と現在の状態から次の状態と出力が決定されます。FSM としては、表 7-5 に示す様な二種類のタイプが知られています。

表 7-5　Moore FSM と Mealy FSM

FSM のタイプ	回路の動作
Moore	Moore タイプの FSM では、出力は現在の状態にのみ依存します。 ① 組み合わ回路により、入力と現在の状態から次の状態を計算してレジスタに保存します。 ② 出力は、現在の状態から組み合わせ回路で計算します。 ③ 出力は現在の状態に対応します。 ④ 出力は、クロックに同期します。
Mealy	Mealy タイプの FSM では、出力は入力と現在の状態に依存します。 ① 出力は入力の変化に対応して即座に変化します。従って、出力は、クロックに対して非同期となります。 ② 出力は、クロックに同期する状態の変化にも対応します。

Mealy タイプの FSM では、出力を状態の変遷に対応させる事ができるため、Moore タイプの FSM よりも少ない状態数で済む場合が多いと云われています。

例えば、ビットシーケンスを入力する FSM において、ビット 1 を 2 回連続して入力すると 1 を出力するとします。それぞれの FSM の状態遷移図は、図 7-4 の様になります。Moore FSM の方が状態数を余計に必要とする事が分かります。

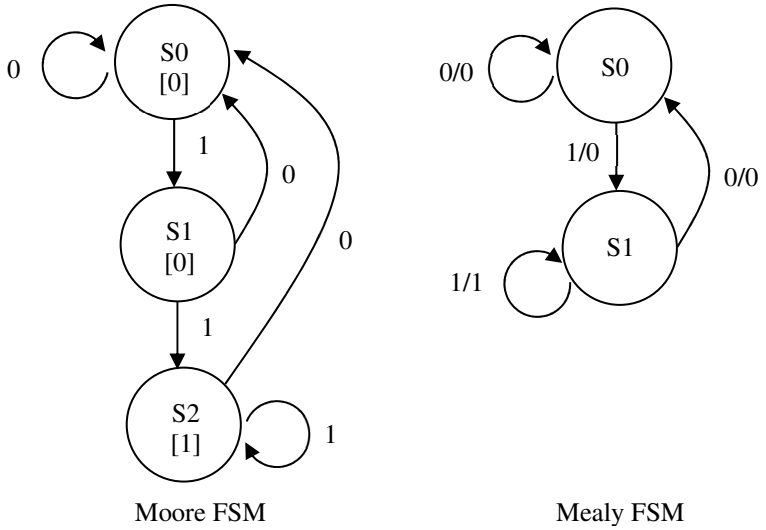


図 7-4　連続した 2 つの 1 を認識する FSM の状態遷移図

この場合には、Moore FSM の S1 と S2 は出力を除くと、全く同じ機能を持っています。そのため、Mealy FSM では、S1 と S2 をマージして状態数を減少させる事ができます。

7.3 Moore FSM

7.3.1 Moore FSM の構成

Moore タイプの FSM は、図 7-5 に様な構成になります。出力は、レジスタの内容から計算されます。

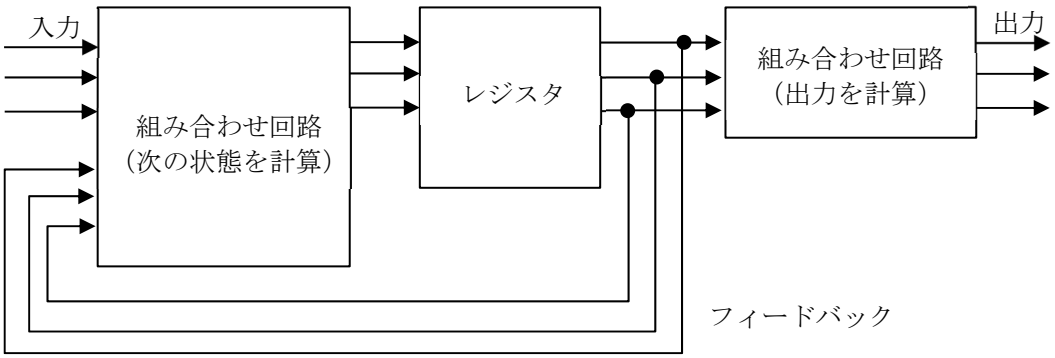


図 7-5　Moore FSM（[2]）

Moore FSM では、クロッキングイベントに同期してレジスタの内容が更新された後に、組み合わせ回路により出力が計算されます。レジスタの内容は NBA 領域で確定するので、同じタイムスロットのその後の Active 領域で出力計算が行われます。従って、同じタイムスロットのその後の Observed 領域で出力の確認を行う事ができます。つまり、クロッキングブロックを使用すれば、競合状態なしに Moore FSM の出力を確認する事ができます。

7.3.2　Moore FSM のモデリング構造

一般的な、Moore FSM のモデリングの手順は以下の様になります。

① クロッキングイベントを持つ always プロシージャで次の状態を計算する。
② 現在の状態にのみ依存する別の always プロシージャで出力計算を行う。
③ 状態を表現するための変数を宣言する。

一般的な Moore FSM のモデリング構造は、以下の様になります。

```
module moore_fsm(input clk,reset,...,output logic out);
state_e   state;
```

現在の状態を示す変数を宣言する。

```
    always @(posedge clk,posedge reset)
        if( reset )
            state <= S0;
        else
            case (state)
            S0: state <= ...
            S1: state <= ...
            ...
            endcase

    always @(state)
        case (state)
        S0: out = ...;
        S1: out = ...;
        endcase

endmodule
```

次の状態を計算して、レジスタに保存する。ここでは、出力の計算を行わない。

出力を計算する。

7.3.3　パリティチェッカーの Moore FSM によるモデリング

例 7-1　パリティチェッカーの実装例（Moore FSM）

先ず、FSM の状態を enum で定義しておきます。

```
typedef enum bit { EVEN, ODD } checker_state_e;
```