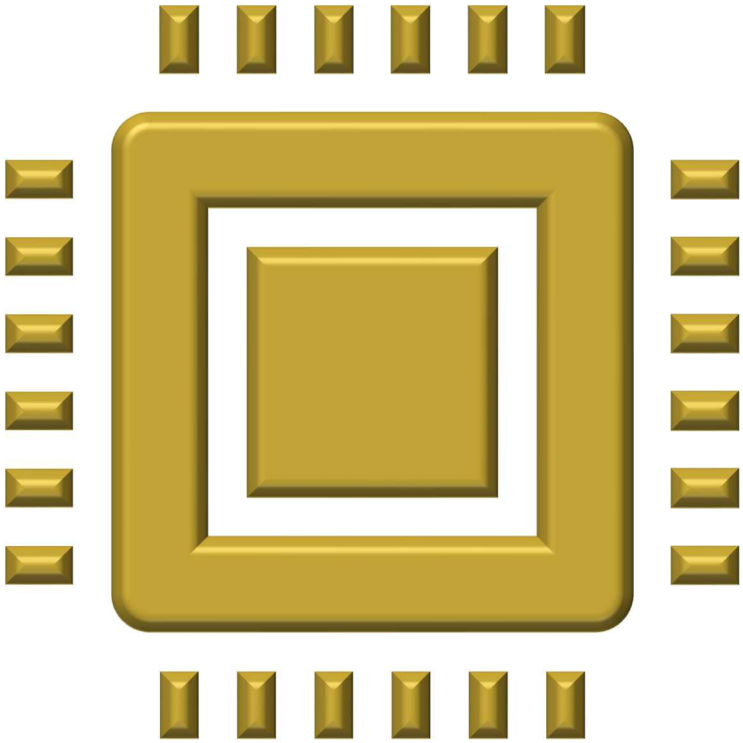
Artgraphics

SystemVerilog によるロジック設計入門

Document Identification Number: ARTG-TD-001-2022
Document Revision: 1.0, 2022.07.31
アートグラフィックス
篠塚一也



SystemVerilog によるロジック設計入門

〒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、および検証に関しては、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をする目的は持ってはいません。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。ただし、検証で使用される重要な機能の解説は本書に含まれていません。例えば、クラス、インターフェース、プログラム、チェッカー等の解説は含まれていませんが、本書ではそれらの知識を仮定しています。従って、本書はSystemVerilogに関する中級、および上級技術者に向いています。勿論、本書はSystemVerilog初心者にも適していますが、内容を選択して学習する必要があります。難易度の高い章は、後回しにして学習する事を勧めます。

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。この章には、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
2022.07.31	1.0	初版。

目次

```
    3: out = d;
    default out = 'x;
    endcase

endmodule
```

この記述によれば、機能は自明と言えます。また、記述に間違いがあったとしても容易に発見する事ができるのは明らかです。論理合成ツールは、この記述を multiplexer_gate に相当するネットリストを自動的に生成します。

SystemVerilogには、もう一つ別の表現形式があります。その記述法はUDPと呼ばれ、いわゆる、真理値表を用いてデザインを記述する方法です。以下は、LRM に紹介されている例です（[1]）。この例は2入力のマルチプレクサを記述しています。

```
primitive multiplexer(mux, control, dataA, dataB);
output mux;
input control, dataA, dataB;
   table
      // control dataA dataB mux
      0 1 ? : 1 ;
      0 0 ? : 0 ;
      1 ? 1 : 1 ;
      1 ? 0 : 0 ;
      x 0 0 : 0 ;
      x 1 1 : 1 ;
   endtable
endprimitive
```

この例で示すように、SystemVerilog の持つ機能を活用していません。UDP で記述する事ができるデザインには限りがあるので、本書ではUDPを使用したモデリングは割愛します。

1.3 モデリングの仕方

回路は、組み合わせ回路とシーケンシャル回路に分類されます。何れの回路をHDLで記述するにしても、記述ルールを順守しなければなりません。順守しないと、仕様に一致する回路を生成する事ができません。HDL記述からゲートへの変換は、EDAツールを使用して自動的に行われるため、ツールが仕様を正しく理解できるためには、誤解のないHDL記述が準備されなければなりません。記述ルールには、HDL言語で定めたルールとEDAツール自体が設定したルールがあります。本書では、前者のルールと、一般的にツールで定められていると考えられるルールを順守するように解説を進めます。

1.3.1 組み合わせ回路と記述ルール

組み合わせ回路では、出力は入力によって一意的に定まります。すなわち、次のような特徴を持ちます。

- 出力は、現在の入力により完全に決定される。
- 入力信号に変化が発生すれば、出力も即座に変化する。
- 出力信号がフィードバックする事はない。すなわち、メモリーを持たない。

これらの特徴から、次のようなルールが導かれます。

- 組み合わせ回路の記述では、回路が依存する信号を漏れなく列挙しなければならない。
- 組み合わせ回路の記述が条件による分岐を含む場合、全ての分岐において出力を設定しなければならない。もし、出力信号に値を設定しないパスが存在すると、現在の出力信号値を保存する条件が生成され、メモリー要素を持つ回路が生成される。すなわち、組み合わせ回路ではなく、レベルセンシティブなラッチが生成される。
- 組み合わせ回路の記述では、ディレーやイベント制御を使用する事はできない。

回路が依存する信号を漏れなく列挙しないと、EDA ツールは HDL 記述を正しく解析する事ができないため、論理合成された回路と元の HDL の記述にはタイミング上の差異が発生してしまいます。

例えば、先程紹介した 4 入力のマルチプレクサのブロックダイアグラムは、図 1-3 のように表現されます。

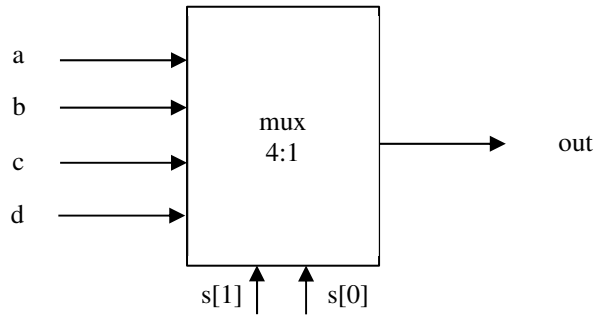


図 1-3　4 入力マルチプレクサのブロックダイアグラム

従って、このような機能を持つマルチプレクサを生成するためには、SystemVerilog 記述は回路が依存する信号を次のように明示的に宣言しなければなりません。このような信号のリストは、センシティビティリストと呼ばれます。

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

回路が依存する信号名を漏れなく列挙する

ここで、以下に示す記述は信号 a、b、c、d、s の何れかに変化があれば、いつでも case 文以降の再計算を行う事を指示しています。すなわち、入力の変化に同期して回路が動作する仕組みを表現しています。

```
always @(a,b,c,d,s)
    case (s)
    ...
```

回路が依存する信号に変化があれば、常に case 文を実行しなければならないので、キーワード always が必要になります。もし、always が存在しないと、一回だけのイベントになるため、論理合成ツールは正しい回路を生成できません。

しかし、次のように記述すると 4 入力のマルチプレクサが生成されるかどうかは分かりません。合成結果は、論理合成ツールの性能に依存します。

```
module mux(input a,b,c,d,[1:0] s,output out);
assign out = (s == 0) ? a : (s == 1) ? b : (s == 2) ? c : d;
endmodule
```

条件演算子（?:）は、基本的には、二者択一の選択機能であるため、4 入力の マルチプレクサの代わりに、2 入力の マルチプレクサが図 1-4 のように生成される可能性が極めて高いと考

えられます。

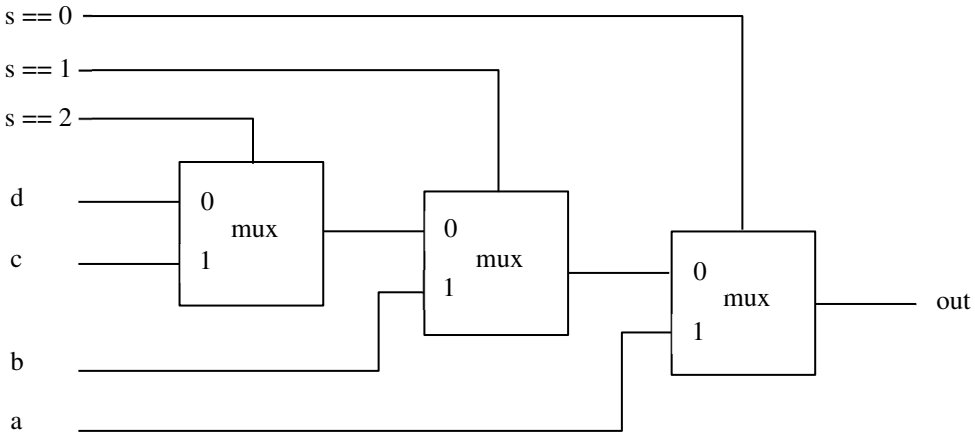


図 1-4　条件演算子（?:）による回路の生成

さて、組み合わせ回路における if 文や case 文のように、条件による分岐を伴う場合、全ての分岐において出力信号に値を設定しなければなりません。例えば、以下の組み合わせ回路は正しい記述です。

```
module mux2(input a,b,sel,output logic out);

always @(a,b,sel)
    if( sel == 1'b1 )
        out = a;
    else
        out = b;

endmodule
```

しかし、次の記述は正しい組み合わせ回路記述ではないため、ラッチが生成されてしまいます。信号 sel は 0~3 の値を取りますが、sel が 2 と 3 を取る場合の処理が不足しています。

```
module simple_latch(input [1:0] sel,output logic [3:0] out);

always @(sel)
    if( sel == 0 )
        out = 0;
    else if( sel == 1 )
        out = 4;

endmodule
```

sel が 0、1 以外であると
out には値が設定されない

この記述は正しいセンシティビティリストを持ちますが、信号 out に値が設定されないパスが存在するため、ラッチが生成されてしまいます。例えば、sel==2 であると、out は現在の値を維持します。すなわち、この回路はレベルセンシティブなラッチに他なりません。

1.3.2　シーケンシャル回路と記述ルール

シーケンシャル回路は、以下のような特徴を持ちます。

- シーケンシャル回路では、現在の入力と過去に発生した入力の履歴により、次の状態（next state）、および出力が決定される。
- 次の状態は適度なディレーを経過してから有効になり、次の計算サイクルで使用される。

- シーケンシャル回路のコンフィギュレーションを更新するタイミングを、特別な信号に合わせる場合と、そのような信号を持たない場合に分類される。前者のシステムは、同期システム、後者のシステムは非同期システムと呼ばれる。

本書では、主として同期なシーケンシャル回路を扱います。同期をとるために使用する特別な信号はクロックと呼ばれます。

シーケンシャル回路では、クロック信号に合わせて回路のコンフィギュレーションを更新するため、入力信号のサンプリングはクロッキングイベントに合わせて行われます。すなわち、クロッキングイベントが発生すると入力のサンプリングを行い、入力はコンフィギュレーションを更新する間はそのまま維持されます。実際問題として、サンプリングされた入力は、次のクロッキングイベントまで保持されます。すなわち、クロッキングイベントが起きた後に入力信号が変化しても、シーケンシャル回路には反映されずに、次回のクロッキングイベントまで影響を持ちません。コンフィギュレーション更新中にも即座に入力の影響を持たせるためには、その信号をクロックと非同期であるとして宣言しなければなりません。それらの非同期信号は、一般に、セット、リセット、プリセット、クリア等として知られています。

一般のデータ信号をクロックと非同期に動作するようにデザインする事も可能です。その方法は、Mealy FSM として知られています。Mealy FSM は、同期なシーケンシャル回路ですが、出力はクロックと非同期に動作するという特徴があります。

SystemVerilog では、シーケンシャル回路を `always` プロシージャを使用して記述しなければなりません。その際、以下のようなルールに従う必要があります。

- クロック信号と非同期信号以外は、センシティビティリストに指定しない。
- センシティビティリストに指定する信号には、`posedge`、または `negedge` を指定する。すなわち、シーケンシャル回路は、エッジセンシティブでなければならない。この場合、記述スタイルに応じて各種のフリップフロップが生成されます。
- 非同期信号を条件判定に使用する場合、条件判定とエッジが一致しなければならない。
- クロック信号は、センシティビティリスト以外に出現してはならない。
- 非同期信号は、必ず、条件判定に使用されなければならない。
- 一般的に、ノンブロッキング命令（<=）を使用して記述する。

これらのルールに従わない場合、論理合成ツールは正しい回路を生成する事ができません。また、これらのルールに加えて EDA ベンダー特有の制限が付加される事にも注意して下さい。ここで紹介したルールは、一般的に順守すべきルールです。

例えば、次の記述は非同期な `set`、および `reset` 信号付のフリップフロップの例です。この記述例は、上記の条件を満たしているので、RTL 論理合成可能なシーケンシャル回路です。

```
module async_flipflop(input logic clk,set,reset,data,
                      output logic q,q_bar);

assign q_bar = ~q;

always @(negedge set or negedge reset or posedge clk) begin
    if( reset == 0 )
        q <= 0;
    else if( set == 0 )
        q <= 1;
    else
        q <= data;
end

endmodule
```

エッジが negedge reset なので
    if(reset == 0)
または、
    if(!reset)
のように判定しなければならない

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での動作シミュレーションが必要になります。あるいはまた、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。このような時、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 のよう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、以下のよう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、次のように最適化されます。

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

この式を観察すると、式が 2 入力のマルチプレクサを表現している事が分かります（図 4-1）。

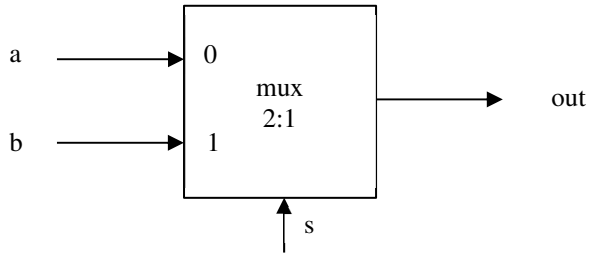


図 4-1　2 入力マルチプレクサ

真理値表からブール式を導く手順は、以下のようにまとめられます。

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 入力のマルチプレクサを$f(s,a,b)$と書くと、以下のように定義されます。

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

この基本表現から、以下の等式を得ます。

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

すなわち、AND、OR、INV がマルチプレクサで構成される事が分かります。したがって、全ての組み合わせ回路はマルチプレクサで表現する事ができる事が分かります。マルチプレクサは、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

等号演算は、最も良く使用されるオペレータです。例えば、if 文では以下のように等号を使用します。

```
if( s == 1 )
    out = a;
else
    out = b;
```

case 文でも同様に以下のように等号を使用します。この場合には、明示的ではありませんが、s==0、s==1、s==2、s==3 が使用されています。

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

例 4-1　a==b の真理値表の例

a==b の真理値表は表 4-2 のようになります。

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

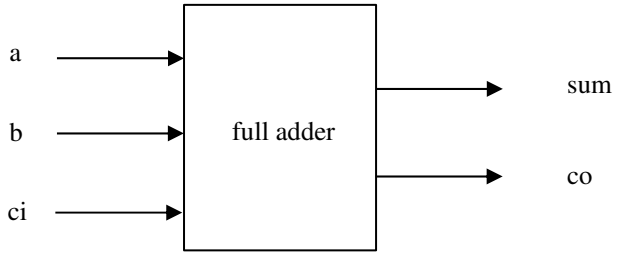


図 4-3　フルアダーのブロックダイアグラム

例 4-3　フルアダーの真理値表の例

フルアダーの真理値表は、表 4-4 のようになります。

表 4-4　フルアダーの真理値表

a	b	ci	co	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

これらの値から、次の式を得る事ができます。

$$sum = a' * b' * ci + a' * b * ci' + a * b' * ci' + a * b * ci$$
$$co = a' * b * ci + a * b' * ci + a * b * ci' + a * b * ci$$

これらの式は、次のように最適化されます。

$$sum = (a * b + a' * b') * ci + (a * b' + a' * b) * ci' = a \oplus b \oplus ci$$
$$co = a * b + b * ci + a * ci$$

これらの結果から、`sum` は XOR、`co` は majority 回路から構築する事ができるのが分かります。3 つの変数、`a`、`b`、`ci` の内で少なくとも 2 つの変数が 1 であれば、`co` が 1 となるので、多数決の原理が成立します。そのため、$a * b + b * ci + a * ci$ は majority 関数と呼ばれています。

■

4.1.4　7 セグメント表示と真理値表

次の例が示すように、真理値表の全ての項目が埋まるとは限りません。そのような場合、do-not-care の意味で X を記します。

例 4-4　7 セグメント表示の制御例

7 セグメントに対して、図 4-4 のようにラベルを割り当てる事ができます。

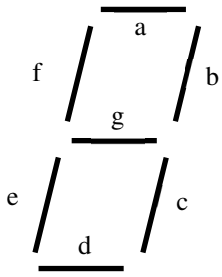


図 4-4　7 セグメントのラベル

BCD から 7 セグメントに変換するために、表 4-5 を 利用する事ができます。

表 4-5　BCD を 7 セグメントに変換する真理値表

d3	d2	d1	d0	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	X	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	X	0	1	1
1	0	1	0	X	X	X	X	X	X	X
1	0	1	1	X	X	X	X	X	X	X
1	1	0	0	X	X	X	X	X	X	X
1	1	0	1	X	X	X	X	X	X	X
1	1	1	0	X	X	X	X	X	X	X
1	1	1	1	0	0	0	0	0	0	0

BCD は 4 ビットですが、0~9 の値を示すだけであるため、10~15 は使用されていません。そのため、X で表を埋めています。最後の行は、7 セグメントをブランク表示にするためのコードを意味します。

SystemVerilog によるモデリングは以下のようになります。下記のモジュールは、BCD を入力して、7 セグメントコードに変換をします。

```
typedef enum logic [6:0] {
       D0 = 7'b1111_110,
       D1 = 7'b0110_000,
       D2 = 7'b1101_101,
       D3 = 7'b1111_001,
       D4 = 7'b0110_011,
       D5 = 7'b1011_011,
       D6 = 7'b0011_111,
       D7 = 7'b1110_000,
       D8 = 7'b1111_111,
       D9 = 7'b1110_011,
       BLANK = 7'b0000_000
   } seven_segment_e;

module seven_segment(input logic [3:0] number,
       output logic [6:0] out);
```

① 車のイグニッションにキー（K）が差し込まれていてドア（D）が開いている。
② イグニッションにキーが差し込まれていなくて、ブレーキ（B）が掛かっていない。
③ ドアが開いていて、ブレーキ（B）が掛かっていない。

これらの条件で使用する変数を表 4-10 のように定義します。

表 4-10 ブール変数の定義

変数	意味
K	車のイグニッションにキーが差し込まれている。
D	車のドアが開いている。
B	ブレーキが掛かっていない。

このように準備すると、ブザーがなる状況を以下のようなブール式で表現する事ができます。

$$Alarm = K * D + K' * B + D * B$$

しかし、このブール式は最適ではありません。consensus 定理により以下のように簡略化され、AND ゲートを一つ削減する事ができます。

$$Alarm = K * D + K' * B$$

つまり、条件③は蛇足と言えます。
■

4.2.2 Shannon の展開定理（Boole の展開定理）

n 変数のブール式に関する Shannon の展開定理（[3]）とは、以下の等式を意味します。

$$f(x_1, x_2, \dots, x_n) = x_i * f_{x_i} + x_i' * f_{x_i'}$$

ここで、f_{x_i}は、$f(x_1, x_2, \dots, x_n)$において、$x_i == 1$を代入した式を意味します。同様に、$f_{x_i'}$ は、$f(x_1, x_2, \dots, x_n)$において、$x_i == 0$を代入した式を意味します。Shannon の展開定理を繰り返し適用する事により$f(x_1, x_2, \dots, x_n)$を n 変数の SOP 形式で表現する事ができます。

例 4-8 Shannon の展開定理の応用例

この例では、Shannon の展開定理を使用して consensus 定理を証明します。式 $a * b + a' * c + b * c$ に対して、a に関して Shannon の展開定理を適用すると以下のようになります。

$$a * (b + b * c) + a' * (c + b * c)$$

これは、以下の式に等しくなります。

$$a * b + a' * c$$

従って、以下の等式を得ます。

$$a * b + a' * c + b * c = a * b + a' * c$$
■

4.2.3 Shannon の展開定理の応用

4.2.3.1 マルチプレクサによる階層設計

Shannon の展開定理とマルチプレクサには密接な関係があります。その関係を利用するとロジック設計のヒントに繋がります。基本的な Shannon の展開定理は、以下のように表現されます。

$$f(x_1, x_{2,} \ldots, x_n) = x_i * f_{x_i} + x_i' * f_{x_i'}$$

つまり、Shannon の展開定理は、n 変数の組み合わせ回路は、(n−1) 変数の組み合わせ回路とマルチプレクサで記述する事ができる事を示しています。そこで、x_1を 2 入力マルチプレクサの制御変数とすれば、$f(x_1, x_{2,} \ldots, x_n)$は、(n−1) 変数の関数から階層的に構築する事ができるようになります（図 4-5）。

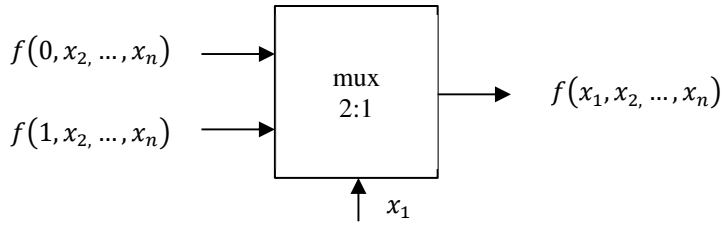


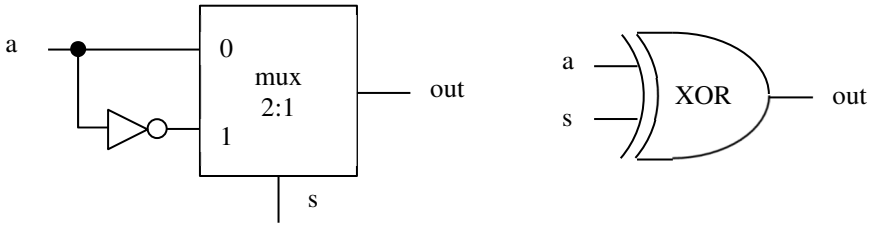
図 4-5　n 変数の組み合わせ回路を (n−1) 変数の組み合わせ回路から構築する方法

4.2.3.2　Shannon の展開定理と XOR

以下の SystemVerilog 記述を考察します。Shannon の展開定理より、図 4-6 (a)に示すような回路構成が予測されますが、最適化されると図 4-6 (b)のような回路構成になります。

```
logic  a, s, out;

always_comb
    if( s )
        out = ~a;
    else
        out = a;
```



(a)　素朴な回路表現　　(b)　最適化された回路表現

図 4-6　マルチプレクサによる XOR 表現

与えられた SystemVerilog 記述を$f(s, a)$と記すと、以下のようなブーリアン式が得られます。この式が XOR を表現している事は明らかです。

$$f(s, a) = s * a' + s' * a = s \oplus a$$

尚、元の SystemVerilog 記述を以下のように変更すると、XNOR が得られます。

```
logic  a, s, out;

always_comb
    if( s )
        out = a;
    else
        out = ~a;
```

表 4-11　2 変数のロジック関数

a	b	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F 10	F 11	F 12	F 13	F 14	F 15
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

表 4-12 には、それらの関数が表現する式を明記しました。多くの関数は、良く知られている回路を表現しています。例えば、F6 と F7 を考えてみます。それらの真理値表は、表 4-13 の様になります。この真理値表から、以下の等式を得ます。

$F6 = a' * b + a * b' = a \oplus b$
$F7 = a' * b + a * b' + a * b = a' * b + a = a + b$

実は、F7 はもっと簡単に求める事ができます。F7==0 となる場合の方が、F7==1 となる場合よりも遥かに少ないので、$F7'$を求めた方が効率が良いです。例えば、

$F7 = (F7')' = (a' * b')' = a + b$

のように簡単に求まります。$F14$も同様です。

$F14 = (F14')' = (a * b)'$

他の関数も同様にして確認する事ができます。

表 4-12　2 変数のロジック関数

関数	ブール式	呼称
F0	0	PASS 0
F1	a*b	AND
F2	a*b'	
F3	a	PASS a
F4	a'*b	
F5	b	PASS b
F6	a⊕b	XOR
F7	a+b	OR
F8	(a+b)’	NOR
F9	(a⊕b)'	XNOR
F10	b’	NOT b
F11	a+b'	
F12	a’	NOT a
F13	a'+b	
F14	(a*b)'	NAND
F15	1	PASS 1

表 4-13　F6 と F7 の真理値表

a	b	F6	F7
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	1

7 FSM

本章では、FSM の種類、および FSM の SystemVerilog によるモデリングを解説します。FSM の出力と次の状態は、入力と現在の状態から組み合わせ回路により決定されます。そして、計算された次の状態は、レジスタ（フリップフロップ）に保存されます。

7.1 概要

FSM は、有限個の異なる状態から構成されるシーケンシャル回路です。既に紹介した各種のカウンターは、最もシンプルな FSM の例です。カウンターにおいては、出力と状態が一致し、次のクロッキングイベントで遷移する次の状態も明確です。図 7-1 は、3 ビットのバイナリーカウンターの状態遷移を示しています。

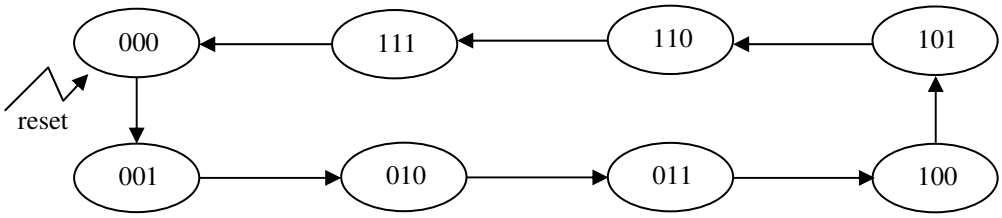


図 7-1　3 ビットのバイナリーカウンターの状態遷移

カウンターに次いで、最も良く知られている FSM は、Odd、または Even パリティチェッカーです。Odd パリティチェッカーでは、現在までのビット 1 の数が奇数であれば 1 を出力し、1 の数が偶数であれば、出力は 0 となります。図 7-2 は Odd パリティチェッカーを示しています。

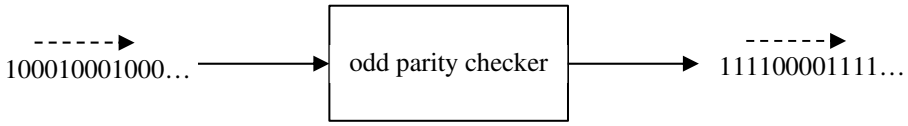


図 7-2　パリティチェッカーのブロックダイアグラム

このパリティチェッカーは、入力と現在の状態から次の状態と出力が決定されるので、FSM になります。この FSM の状態は、Even と Odd の二つの状態から構成されます（図 7-3）。状態遷移図は、出力を状態に割り当てる場合と状態遷移を示すアークに出力を割り当てる場合で異なります。前者は、Moore FSM、後者は Mealy FSM と呼ばれます。

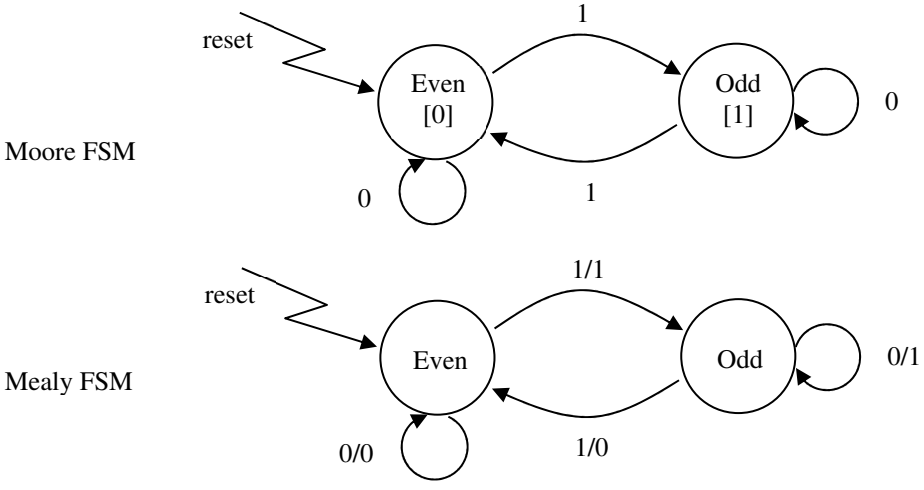


図 7-3　パリティチェッカーの状態遷移図

以降では、これらの状態遷移図を基にして、次の状態、および出力を決定するアルゴリズムを算出する方法について解説する事にします。

先ず、Moore FSM の状態遷移図から表 7-1 のような関係が得られます。

表 7-1　パリティチェッカーの状態遷移表（Moore FSM）

Present State (PS)	Input (I)	Next State (NS)	Present Output (O)
Even	0	Even	0
Even	1	Odd	0
Odd	0	Odd	1
Odd	1	Even	1

次に、Even と Odd にコード 0 と 1 を割り当てると、以下のような真理値表が得られます。

表 7-2　パリティチェッカーの真理値表（Moore FSM）

Present State (PS)	Input (I)	Next State (NS)	Present Output (O)
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

これから、以下のような式を得ます。

$NS = PS \oplus I$
$O = PS$

一方、Mealy FSM の状態遷移図から表 7-3 のような関係が得られます。

表 7-3　パリティチェッカーの状態遷移表（Mealy FSM）

Present State (PS)	Input (I)	Next State (NS)	Present Output (O)
Even	0	Even	0
Even	1	Odd	1
Odd	0	Odd	1
Odd	1	Even	0

次に、Even と Odd にコード 0 と 1 を割り当てると、表 7-4 のような真理値表が得られます。

表 7-4　パリティチェッカーの真理値表（Mealy FSM）

Present State (PS)	Input (I)	Next State (NS)	Present Output (O)
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

これから、以下のような式を得ます。

$NS = PS \oplus I$
$O = NS$

以上では、状態遷移図を基にして出力と次の状態を計算する手順を示しましたが、以降では、状態遷移図からシーケンシャル回路を SystemVerilog を使用して記述する手法を解説します。

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 に示すよう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。したがって、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 のようになります。Moore FSM の方が状態数を余計に必要とする事が分かります。

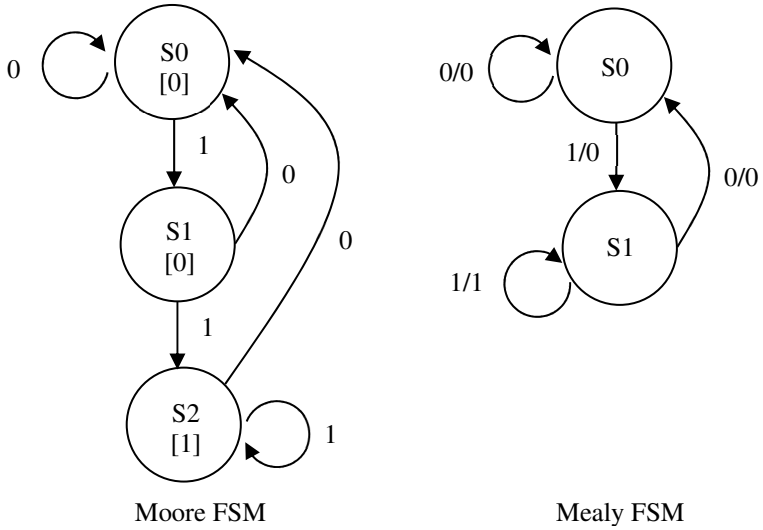


図 7-4　連続した 2 つの 1 を認識する FSM の状態遷移図

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 にような構成になります。出力は、レジスタの内容から計算されます。