Artgraphics

SystemVerilog による設計と論理合成

Document Identification Number: ARTG-TD-001-2024
Document Revision: 2.2, 2025.04.07
アートグラフィックス
篠塚一也

SystemVerilog による設計と論理合成

© 2025 アートグラフィックス
〒124-0012 東京都葛飾区立石 8-14-1
www.artgraphics.co.jp

Design and Logic Synthesis with SystemVerilog

© 2025 Artgraphics. All rights reserved.
8-14-1, Tateishi, Katsushika-ku, Tokyo, 124-0012 Japan
www.artgraphics.co.jp

注意事項

- 本解説書により得られた知識・情報の使用から生じるいかなる損害についても、弊社および本書の著者は責任を負わないものとします。
- 弊社の技術資料の内容の一部、あるいは全部を無断で複写、複製、転載する事は、禁じます。
- 弊社の技術資料の譲渡、転売、模倣、又は、改造等の行為を禁止致します。

はじめに

現代では、ハードウェア記述言語はハードウェア設計に欠かせない手段になっています。本書は、RTL 記述の原則とルール、および、RTL 記述から合成回路を予測する手法を解説した技術資料です。論理合成の観点から合成回路を予測する手法を採用しているので、市販の論理合成システムに関する使用技術の習得にも繋がる効果をもたらします。ただし、本書は論理合成のアルゴリズムを解説しているわけではありません。本書は、RTL 記述が表現する回路を導くための技術を解説する目的をもっています。RTL 記述と生成される回路との対応を予測できれば、論理合成を効果的に使用する技術にも繋がります。そして、論理合成の働きを正しく理解できれば、優れた性能を持つ回路を効率よく開発できる機会に恵まれます。

論理合成は、ハードウェア記述言語で記述された仕様から回路表現を導く EDA ツールです。回路表現は、通常、ネットリストと呼ばれる別の抽象度を持つハードウェア記述言語で表現されます。先ずは、ハードウェア、ハードウェア記述言語、論理合成との関係を考察する事から始めます。本書では、ハードウェア記述言語として SystemVerilog を仮定します。

`a==b` と記述すると XNOR が生成されることは誰でも知っていると思います。直ぐにはピンと来ないかも知れませんが、以下のように書いても、やはり `a==b` を意味します。

```
{a,b} == 0 || {a,b} == 3   // XNOR : a == b
```

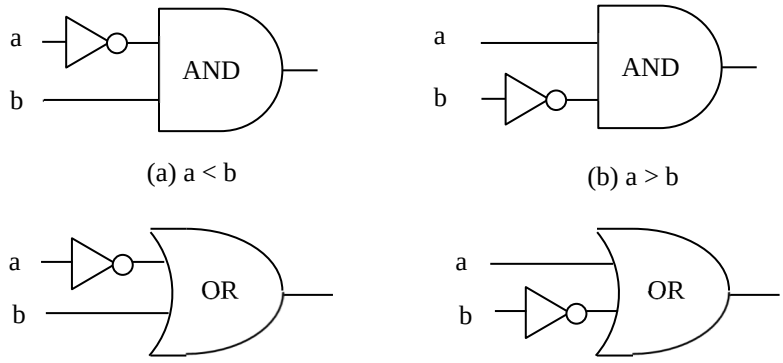
しかも、このような記述をしてしまう事もあるので、注意を怠ると冗長な表現をしてしまいます。同様に、以下の記述は XOR です。つまり、`a!=b` を意味しています。

```
{a,b} == 1 || {a,b} == 2   // XOR : a != b
```

実際問題として、この種の記述法をする事もあり得るので、冗長な表現を避けるためには記述を見直す必要があります。特に、変数とリテラルが用いられている式には、プリミティブな基本回路が生成されます。

さて、`{a,b} == 1` は、`a < b` を意味します。一方、`{a,b} == 2` は `a > b` を意味します。したがって、XOR は `a < b` と `a > b` から構築されているといえます。言い換えれば、XOR は `a!=b` を意味しています。寧ろ、これが XOR の定義であると考えるのが適切です。

組み合わせ回路は AND、OR、INV を用いて構築できますが、2:1 マルチプレクサを用いても組み合わせ回路を作り上げる事ができます。これらの基本回路はロジック設計において本質的な役割を果たしているので、本書の大半はこれらの回路の解説と応用法に費やされています。基本回路でも複合的な機能を持ちます。例えば、下図(b) に示す回路は `a > b` を示しています。そして、この種の回路を組み合わせる事により、複雑な比較演算回路を構築する事ができます。しかも、これらの回路を 2:1 マルチプレクサで簡単に表現できます。



(a) a < b　　(b) a > b

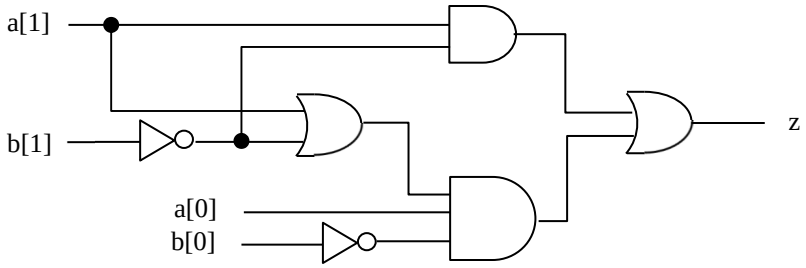
(c) a <= b　　(d) a >= b

Copyright © 2025 Artgraphics. All rights reserved.

因みに、(c) の a <= b を例にとれば、「a が真(1)であれば b は真(1)である」を意味しています。しかも、a が偽(0)である場合には、常に、成立するので b はどうでも良い事になります。つまり、SystemVerilog の以下の式を意味します。この演算子は、logical implication として知られています。

$$a -> b$$

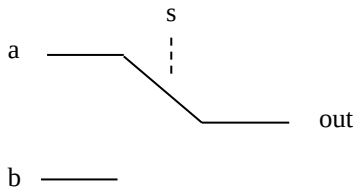
このように、簡単な論理回路でも論理を表現するツールとして有効に活用できます。また。下図 (e)に示す回路は上図 (b)の回路を応用して作り上げた機能です。(b)の機能を理解していれば(e)で示す回路の機能を特定することは難しくありません。



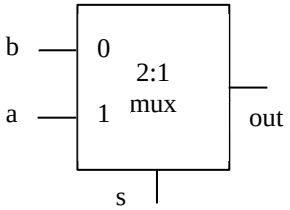
(e) 基本回路を適用した組み合わせ回路の機能

このように、本書では論理を構成する基本回路をつぶさに観察してロジック設計技術を身に付ける方法を解説しています。では、次に 2:1 マルチプレクサの役割について触れます。

トグルスィッチは、出力が、常に、二者択一の入力に接続されている回路を意味します。入力信号を a と b とし、出力を out、制御信号を s とすると下図(f)のような構成を得られます。信号 s はスイッチの役割を果たし、s == 1 (オン)であれば out は a に、s == 0 (オフ)であれば out は b に接続されます。



(f) トグルスウィッチ

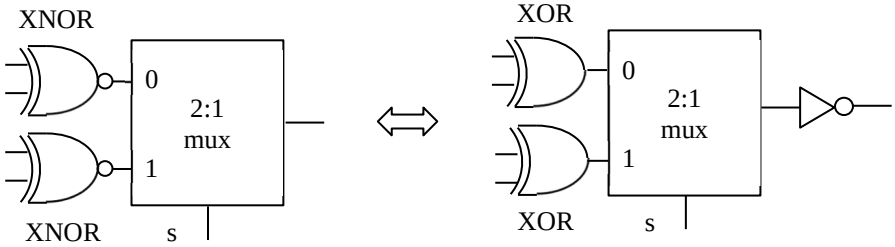


(g) 2:1 マルチプレクサ

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

(h) SystemVerilog 記述

トグルスウィッチは図(g)で示すような 2:1 マルチプレクサに他なりません。また、トグルスウィッチの定義をハードウェア記述言語で表現すれば図(h)のようになります。一方、マルチプレクサには面白い性質がありますが、ハードウェア記述言語を使用した記述では、自然にその性質を利用しています。下図は、その性質を分かり易く説明しています。



(i) 反転された入力　　(j) 出力を反転した構成

ブール代数を使用すれば、この性質が正しい事を厳密に証明できます。この性質は、インバ

Copyright © 2025 Artgraphics. All rights reserved.

ータを一つ節約できる事を示していますが、ハードウェア記述言語を使用している場合には、自然にこの性質を利用しています。ハードウェア記述言語を使用すると以下のように表現できますが、通常は左のように記述します。つまり、自然にインバータを節約する記述をしていると言えます。

図の(i)を表現した記述	図の(j)を表現した記述

`if( s == 1'b1 )`
`  z = a~^b;`
`else`
`  z = c~^d;`
	`assign z = ~w;`
	`always_comb`
	`  if( s == 1'b1 )`
	`  w = a^b;`
	`  else`
	`  w = c^d;`

このように、ハードウェア記述言語では無駄のない記述をしていれば、比較的良い回路が合成される事につながります。論理合成は、その意図を考慮して回路を合成します。

しかし、論理合成は万能ではありません。意図に即した最適な回路構成を得るためには、合成し易い記述が与えられなければなりません。例えば、jk―フリップフロップを以下に示す(k)のように記述すると、論理合成には難しい課題を与えるため最適な回路構成を得られるか否かは論理合成ツールの性能に依存します。しかし、(l)のように記述すれば、殆どの場合に最適な回路に合成されます。

(k) 論理合成を考慮しない記述	(l) 論理合成を考慮した記述

`always @(posedge clk) begin`
`  if( j == 1'b0 && k == 1'b0 )`
`  q <= q;`
`  else if( j == 1'b0 && k == 1'b1 )`
`  q <= 1'b0;`
`  else if( j == 1'b1 && k == 1'b0 )`
`  q <= 1'b1;`
`  else if( j == 1'b1 && k == 1'b1 )`
`  q <= ~q;`
`end`
	`always @(posedge clk) begin`
	`  if( {j,k} == 2'b00 )`
	`  q <= q;`
	`  else if( {j,k} == 2'b01 )`
	`  q <= 1'b0;`
	`  else if( {j,k} == 2'b10 )`
	`  q <= 1'b1;`
	`  else if( {j,k} == 2'b11 )`
	`  q <= ~q;`
	`end`

では、どのような回路に合成されるか予測してみます。jk―フリップフロップの仕様から下表(1)を得ます。表(2)は、現在の状態 `q(t)` が 0 である場合の `q(t+1)` を示し、表(4)は現在の状態 `q(t)` が 1 である場合の `q(t+1)` を示しています。

j	k	q(t+1)
0	0	q(t)
0	1	0
1	0	1
1	1	~q(t)

(1) jk―フリップフロップの真理値表

j	k	q(t+1)
0	0	0
0	1	0
1	0	1
1	1	1

(2) q(t) == 0 の時

j	k	q(t+1)
0	0	1
0	1	0
1	0	1
1	1	0

(3) q(t) == 1 の時

j	k	q(t+1)
0	0	1
1	0	1
0	1	0
1	1	0

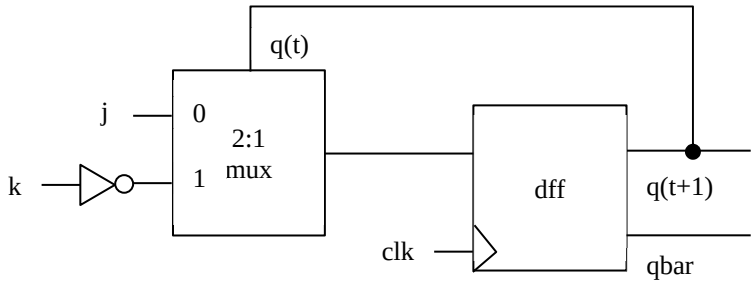
(4) q(t) == 1 時の状態を見易くした状態

上表から次の事がわかります。したがって、この回路は2:1マルチプレクサに他なりません。

- 表(2)から、`q(t)==0` であれば、`q(t+1)==j` である。
- 表(4)から、`q(t)==1` であれば、`q(t+1)==~k` である。

以上から、下図に示す回路構成が得られます。

Copyright © 2025 Artgraphics. All rights reserved.



この例が示すように、RTL 記述では意図を明確にする必要があります。例えば、上記の場合には信号 j と k が意味を持つ対として使用されている事を明示的に表現する必要があります。

本書は、基本的な構文を利用して、RTL 記述がどのような回路に合成されるかを習得する技術を養う目的を持っています。そのために、RTL 記述に必要な SystemVerilog の基礎知識、論理合成が仮定しているルール、組み合わせ回路の記述法、シーケンシャル回路の記述法、ラッチの記述法など幅広い観点から解説を進めています。

第 1 章では、SystemVerilog による動作記述法を基本回路をもとにして解説しています。そして、合成回路を予測する技術を典型的な回路を使用して解説します。また、二項演算子の基礎回路となる 2 変数ロジック関数の解説をします。

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全ての機能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。

第 3 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の代表的な話題がまとめられています。それに加えて、真理値表が表現する論理回路を SystemVerilog で記述する方法も紹介されているので、この章は非常に濃い内容を含んでいます。例えば、XOR、XNOR、NAND、NOR 等の論理回路を if-then-else で記述する技術を習得できます。

第 4 章では、RTL の動作記述をするための構文 `always` の使用法をセンシティビティリスト、ブロッキング代入文、ノンブロッキング代入文と関連させて詳しく解説しています。`always` の意義を正しく理解するための重要な章です。

第 5 章では、モデリングで必要とされるエンコーディングが解説されています。Gray コード、Johnson コード、one-hot コード等に関する基礎知識を再確認するために役立つ知識が解説されています。

第 6 章では、データ表現の基本となる整数表現を詳しくまとめてあります。特に、重要な概念である整数演算オーバーフロー検出を解説してあります。ここで、演算オーバーフローとは、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ー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。また、加減算器の応用例としてインクリメントとデクリメントの二つの機能を備えた回路も紹介します。

第 7 章は、RTL 記述の基礎を構成する演算子と式の解説を含んでいます。その他、`if` 文や `case` 文の合成技術の基礎知識が詳しく解説されています。

第 8 章は、ラッチに関する話題です。レベルセンシティブなラッチと RS ラッチに関する注意事項が詳しく解説されています。

Copyright © 2025 Artgraphics. All rights reserved.

第 9 章は、シーケンシャル回路のモデリングと論理合成を主題にしています。シーケンシャル回路の検証法の簡単な解説も含まれています。

第 10 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。

本書は、論理合成のルールと RTL 記述を回路に変換する技術を様々なデザイン例を通して詳しく解説しています。本書を精読することにより、SystemVerilog による適切な RTL 表現を準備する習慣が自然と身に付きます。

アートグラフィックス
篠塚一也

Copyright © 2025 Artgraphics. All rights reserved.

変更履歴

日付	Revision	変更点
2024.11.22	1.0	初版。
2025.01.17	1.1	全体的な校正を実施。
2025.03.14	2.0	全体的に内容を改訂。
2025.03.31	2.1	練習問題を追加。
2025.04.07	2.2	全体的な校正を実施。

Copyright © 2025 Artgraphics. All rights reserved.

目次

1 概要 1
1.1 SYSTEMVERILOG による動作記述 1
1.2 回路予測 8
1.2.1 ヒューリスティックな予測法 8
1.2.2 RTL 記述から予測する方法 9
1.2.3 正攻法 10
1.3 合成回路の予測技術 11
1.4 本書の目的と構成 11
1.5 例題に関して 13
1.6 本書の記法 14
1.7 練習問題 15
2 設計に必要な SYSTEMVERILOG の基礎知識 18
2.1 基礎知識 18
2.1.1 論理値 18
2.1.2 4-state 型 19
2.1.3 2-state 型 21
2.1.4 Integral データタイプ 22
2.1.5 enum データタイプ 22
2.1.6 string データタイプ 23
2.1.7 string リテラル 24
2.1.8 packed アレイと unpacked アレイ 25
2.1.9 ビットセレクトとパートセレクト 27
2.1.10 プロシージャ 27
2.1.11 モジュール 30
2.2 モデリングに役立つ基礎知識 33
2.2.1 オペレータとオペランド 33
2.2.2 {} オペレータ 34
2.2.3 inside オペレータ 36
2.2.4 ビットセレクト 37
2.2.5 プライオリティとパラレル 37
2.2.6 ブロッキング代入文とノンブロッキング代入文 38
2.2.7 パラメータとパッケージによる汎用的な記述 40
2.3 練習問題 41
3 ブール代数の基礎知識 45
3.1 重要な性質 45
3.2 SHANNON の展開定理（BOOLE の展開定理） 50
3.3 SHANNON の展開定理の応用 51
3.3.1 マルチプレクサによる階層設計 51
3.3.2 Shannon の展開定理と XOR 51
3.3.3 Shannon の展開定理とマルチプレクサ 52
3.4 ハーフアダーとフルアダー 54
3.4.1 ハーフアダーと真理値表 54
3.4.2 フルアダーと真理値表 55
3.4.3 ハーフアダーとフルアダーの計算アルゴリズムのまとめ 56
3.5 練習問題 57
4 論理合成と ALWAYS 60
4.1 センシティビティリスト 60
4.2 組み合わせ回路 60

Copyright © 2025 Artgraphics. All rights reserved.

4.2.1 センシティビティリスト 61
4.2.2 ブロッキング代入文 61
4.2.3 シングルビットとマルチビット 62
4.2.4 always_comb 63
4.3 シーケンシャル回路 64
4.3.1 センシティビティリスト 65
4.3.2 ノンブロッキング代入文 68
4.3.3 フリップフロップ数 68
4.4 練習問題 70

5 エンコーディング 72

5.1 GRAY コード 72
5.2 JOHNSON コード 74
5.3 ONE-HOT コード 75
5.4 練習問題 75

6 整数表現と演算 77

6.1 整数表現 77
6.2 SIGNED と UNSIGNED 78
6.3 符号の反転処理 80
6.4 加減算の精度 82
6.5 演算とオーバーフロー 83
6.6 練習問題 89

7 組み合わせ回路の論理合成 90

7.1 演算子 90
7.1.1 ビット演算 90
7.1.2 関係演算子 93
7.1.3 数値演算 94
7.2 IF /CASE 文の合成 95
7.3 デコーダー 98
7.4 記述のプライオリティ 101
7.5 3 ステートバス 105
7.6 練習問題 106

8 ラッチ 107

8.1 ラッチの記述法 107
8.2 ラッチを回避する方法 108
8.3 RS ラッチ 108
8.4 RS ラッチをレベルセンシティブラッチで表現する方法 110
8.5 RS ラッチと信号名 111
8.6 RS ラッチとマルチビット 111
8.7 練習問題 114

9 シーケンシャル回路の論理合成 116

9.1 シーケンシャル回路のモデリング 116
9.2 シーケンシャル回路の検証法 116
9.3 非同期信号を持つシーケンシャル回路 118
9.4 非同期信号とマルチビット 119
9.5 データシフタ 121
9.6 練習問題 124

10 FSM 126

10.1 概要 126

Copyright © 2025 Artgraphics. All rights reserved.

10.2 FSM の種類 128
10.3 MOORE FSM 129
10.3.1 Moore FSM の構成 129
10.3.2 Moore FSM のモデリング構造 129
10.4 MEALY FSM 130
10.4.1 Mealy FSM の構成 130
10.4.2 Mealy FSM のモデリング構造 130
10.5 FSM の適用例 131
10.5.1 パターン認識の問題 131
10.5.2 Moore FSM によるモデリング 131
10.5.3 Mealy FSM によるモデリング 134
10.6 練習問題 136

11 練習問題の解答 139

11.1 第 1 章の練習問題の解答 139
11.2 第 2 章の練習問題の解答 142
11.3 第 3 章の練習問題の解答 144
11.4 第 4 章の練習問題の解答 146
11.5 第 5 章の練習問題の解答 147
11.6 第 6 章の練習問題の解答 149
11.7 第 7 章の練習問題の解答 150
11.8 第 8 章の練習問題の解答 155
11.9 第 9 章の練習問題の解答 156
11.10 第 10 章の練習問題の解答 157

12 参考文献 162

Copyright © 2025 Artgraphics. All rights reserved.

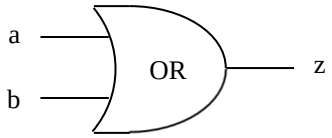
1　概要

本章では、SystemVerilog による RTL 記述と回路表現との関係を具体例を通して解説します。基本的な回路を使用して解説していますが、ここで紹介する変換技術は汎用的であり幅広い応用面を持っています。本章では回路記号、ブール代数、ブール式に触れるので、1.6 節の記法を先に読むと理解し易くなります。

周知のように組み合わせ回路は AND、OR、INV で表現できるので、これらの基本回路を SystemVerilog で記述する方法を解説します。また、全ての組み合わせ回路を 2:1 マルチプレクサで表現できるのも周知の事実なので SystemVerilog 記述とマルチプレクサとの関係も解説します。そして、jk―フリップフロップを例にとり、回路表現を導く手法を詳細に解説します。先ず AND、OR、INV を表現する方法の解説から始めます。

1.1　SystemVerilog による動作記述

オペレータ（｜）を使用すれば OR 回路（図 1-1(a)）を記述できるのは容易に想像できますが、動作記述ではどうすれば良いかを考察する事から始めます。結論を先に述べれば、OR 回路を図 1-1(b)のように記述できます。



(a)　OR 回路

```
if( a == 1'b1 )
    z = 1'b1;
else
    z = b;
```

(b) SystemVerilog による OR の表現

図 1-1　OR 回路と SystemVerilog による記述法

`if` 文の条件を変えると then と else の役割が変わるのに注意しなければなりません。例えば、`a` が偽となる条件を採用すると以下のような表現に変わります。

```
if( a == 1'b0 )
    z = b;
else
    z = 1'b1;
```

次に、なぜ上記のような動作表現が可能であるかを調べてみます。それには、OR の真理値表（表 1-1）を使用すると分かり易くなります。

表 1-1　OR の真理値表

a	b	z
0	0	0
0	1	1
1	0	1
1	1	1

このままでは結論を導くのは難しいので、真理値表に細工をして表 1-2 を得ます。この表は、信号 `a` の値をまとめる事により得られます。このようにすると、ほぼ結論が得られています。

Copyright © 2025 Artgraphics. All rights reserved.

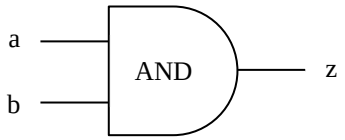
表 1-2　カラム a をまとめた真理値表

a	b	z
0	0	0
	1	1
1	0	1
	1	1

上記の表から以下の事実を得られます。したがって、図 1-1(b)の記述が得られる事がわかります。

- a が 1 であれば z==1 である。
- a が 0 であれば z==b である。

OR と同様にして AND 回路を図 1-2 (b)のように記述できます。

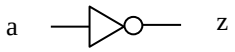



```
if( a == 1'b1 )
    z = b;
else
    z = 1'b0;
```

(a)　AND 回路　　(b) SystemVerilog による AND の表現

図 1-2　AND 回路と SystemVerilog による記述法

ではインバータはどのように記述できるでしょうか？図 1-3 (b)のようになる事は直ぐわかります。




```
if( a == 1'b1 )
    z = 1'b0;
else
    z = 1'b1;
```

(a)　INV 回路　　(b) SystemVerilog による INV の表現

図 1-3　INV 回路と SystemVerilog による記述法

組み合わせ回路は AND、OR、INV により構成できるので、全ての組み合わせ回路をハードウェア記述言語で表現できる事がわかります。

次に、組み合わせ回路で最も重要な役割を持つ 2:1 マルチプレクサを考察します（図 1-4）。

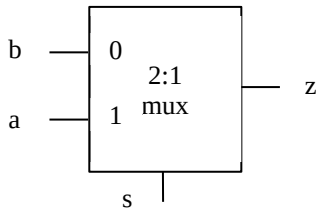


図 1-4　2:1 マルチプレクサの回路表現

マルチプレクサの SystemVerilog 表現を探るためには真理値表を使用します（表 1-3 ）。

Copyright © 2025 Artgraphics. All rights reserved.

表 1-3　2:1 マルチプレクサの真理値表

a	b	s	z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

この表を s に関して並べ替えると表 1-4 を得ます。

表 1-4　s の値に関して並べ替えた真理値表

a	b	s	z
0	0	0	0
0	1	0	1
1	0	0	0
1	1	0	1
0	0	1	0
0	1	1	0
1	0	1	1
1	1	1	1

そして、s の値を見やすくすると表 1-5 のようになります。

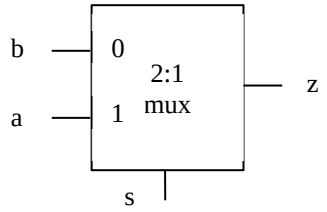
表 1-5　s の値をまとめた真理値表

a	b	s	z
0	0	0	0
1	0		0
0	1		1
1	1		1
0	0	1	0
0	1		0
1	0		1
1	1		1

このようにして得られた表から次の事が言えます。

- `s==0` であれば、`z==b` である。
- `s==1` であれば、`z==a` である。

この事実から、図 1-5 (b)の SystemVerilog 記述を得ます。



(a)　2:1 マルチプレクサ

```
if( s == 1'b1 )
    z = a;
else
    z = b;
```

(b) SystemVerilog によるマルチプレクサの表現

図 1-5　2:1 マルチプレクサの SystemVerilog による表現

Copyright © 2025 Artgraphics. All rights reserved.

AND、OR、INV は if-then-else で表現できたので、2:1 マルチプレクサとの関係を表 1-6 にまとめておきます。機能のカラムを見ると、単純な回路であるにも関わらず複合的な機能の役割をしている事が分かります。

表 1-6　主な 2 変数ロジック関数と同値な 2:1 マルチプレクサ表現

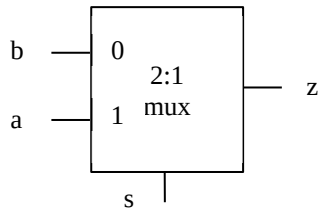
関数	マルチプレクサ表現	基本回路表現	機能
F1	0, b, 0, 1, 2:1 mux, out, a	a, b, AND, out	a && b
F2	a, 0, 0, 1, 2:1 mux, out, b	a, b, AND, out	a > b
F4	b, 0, 0, 1, 2:1 mux, out, a	a, b, AND, out	a < b
F7	b, 1, 0, 1, 2:1 mux, out, a	a, b, OR, out	a || b
F11	1, a, 0, 1, 2:1 mux, out, b	a, b, OR, out	a >= b

Copyright © 2025 Artgraphics. All rights reserved.

3　ブール代数の基礎知識

本章では、ブール代数の基礎知識を復習します。そして、divide-and-conquer アプローチのモデルとなる Shannon の展開定理を解説します。この定理は、Boole の展開定理とも呼ばれています。

先ず最初に、典型的な if-the-else をブール式で図 3-1 (c)のように表現できる事を注意しておきます。回路構成を予測する際には、計算し易い表現を選択する必要があります。



(a) 2:1 マルチプレクサ

```
if( s == 1'b1 )
    z = a;
else
    z = b;
```

(b) SystemVerilog 表現

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

(c) ブール式の表現

図 3-1　if-then-else の表現法

図 3-1 (c)のブール式では、`s==1` であれば `z==a`、`s==0` であれば `z==b` となります。このブール式表現を以下のように書く事にします。

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

ここで、`ite` は if-then-else の頭文字を意味しています。`ite` は 3 変数のロジック関数であると同時に 2:1 マルチプレクサを表現しています。`s`、`a`、`b` に対して様々な値を代入する事により、標準的な回路が得られます。例えば、以下のような関係を得られます。

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

つまり、2:1 マルチプレクサは、AND、OR、INV を生成する能力があるので、2:1 マルチプレクサは全ての組み合わせ回路を表現する能力を持つ事がわかります。

3.1　重要な性質

先ず、ブール代数の公理とも言える性質を表 3-1 にまとめておきます。

表 3-1　ブール代数の基本性質

使用例	呼称
$a + (b + c) = (a + b) + c$	結合律
$a * (b * c) = (a * b) * c$	結合律
$a + a = a$	冪等性
$a * a = a$	冪等性
$a + (a * b) = a$	吸収律
$a * (a + b) = a$	吸収律
$(a + b)' = a' * b'$	ドモルガン
$(a * b)' = a' + b'$	ドモルガン
$(a')' = a$	対合

Copyright © 2025 Artgraphics. All rights reserved.

机上でブール式を最適化する場合には、表 3-2 に示す定理が有効に働きます。この表において定理 3.1 と定理 3.2 は双対です。同様に、定理 3.3 と定理 3.4 も双対です。

表 3-2　ブール代数の定理

式	呼称
$a + a' * b = a + b$	定理 3.1
$a(a' + b) = ab$	定理 3.2
$a * b + a' * c + b * c = a * b + a' * c$	定理 3.3
$(a + b)(a' + c)(b + c) = (a + b)(a' + c)$	定理 3.4

表 3-2 に掲げた定理は非常に有用な機能なので記憶に値します。双対の定理となっているため、実質的には二つの定理となります。大事な定理なので読者自身で証明すると知識が付きます。以下では、参考のために証明をしておきます。

例 3-1　定理 3.1 の証明例

以下のように証明できます。以下の証明には、吸収律を使用しています。

$$a + a' * b = a + a * b + a' * b = a + (a + a') * b = a + 1 * b = a + b$$

■

例 3-2　定理 3.3 の証明例

以下のように証明できます。以下の証明では定理 3.1 と吸収律を使用しています。

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

■

例 3-3　定理 3.1 の応用例（その 1）

1.1 節で以下の記述は OR 回路を表現している事を示しましたが、定理 3.1 を使用しても証明できます。

```
if( a == 1'b1 )
    z = 1'b1;
else
    z = b;
```

上記の記述は、以下の式と同等です。

$$z = a * 1 + a' * b$$

この式は、定理 3.1 により次の式に等しくなります。

$$z = a + b$$

すなわち、上記の記述は OR を表現しています。
■

Copyright © 2025 Artgraphics. All rights reserved.

5　エンコーディング

モデリングする際、事象に対して適切なエンコーディングが要求されます。表 5-1 に頻繁に使用されるコードの一覧をまとめておきます。

表 5-1　バイナリー、BCD，one-hot、Gray、Johnson コード

#	binary	BCD	one-hot	Gray	Johnson
0	0000	0000	0000000000000001	0000	00000000
1	0001	0001	0000000000000010	0001	00000001
2	0010	0010	0000000000000100	0011	00000011
3	0011	0011	0000000000001000	0010	00000111
4	0100	0100	0000000000010000	0110	00001111
5	0101	0101	0000000000100000	0111	00011111
6	0110	0110	0000000001000000	0101	00111111
7	0111	0111	0000000010000000	0100	01111111
8	1000	1000	0000000100000000	1100	11111111
9	1001	1001	0000001000000000	1101	11111110
10	1010	xxxx	0000010000000000	1111	11111100
11	1011	xxxx	0000100000000000	1110	11111000
12	1100	xxxx	0001000000000000	1010	11110000
13	1101	xxxx	0010000000000000	1011	11100000
14	1110	xxxx	0100000000000000	1001	11000000
15	1111	xxxx	1000000000000000	1000	10000000

Gray コードと Johnson コードは、連続するコード間では 1 ビットしか変化しないという特徴を備えていますが、Johnson コードは Gray コードより多くのビット数を必要とします。一方、Gray コードは、バイナリーコードと同じビット数で充分です。

これらのエンコーディングをカウンターとして使用するとき、Gray と Johnson コードの数え上げは自明ではありません。本章では、それらの数え上げの仕方を考察します。

5.1　Gray コード

Gray コードを応用した回路に Gray カウンターがあります。Gray カウンターは、Gray コードを順に生成するシーケンシャル回路です。簡単に言えば、Gray カウンターはバイナリーコードを Gray コードに変換するシーケンシャル回路です。図 5-1 は 3 ビットの Gray カウンターの様子を示しています。Gray カウンターでは状態か変化する時に 1 ビットしか変化しないので、通常のバイナリーカウンターとは値の順序が異なります。

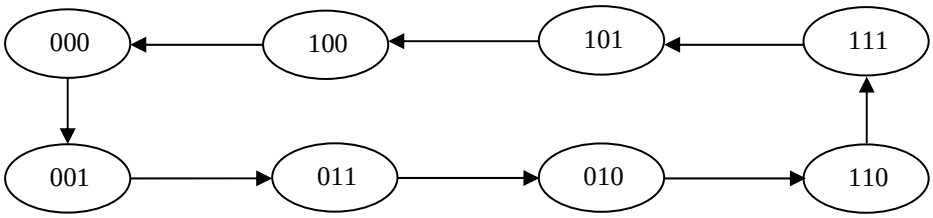


図 5-1　3 ビット Gray カウンターの状態遷移図

Gray コードを構成するビットを MSB から順に `g[n-1:0]` とし、バイナリーコードを構成するビットを MSB から順に `b[n-1:0]` とします。Gray コードは、以下のようにして求まります。

- Gray コードの MSB は、バイナリーコードの MSB と一致する。すなわち、`g[n-1]==b[n-1]` です。
- MSB 以外の Gray コード `g[i]` は、XOR で求まる。すなわち、`g[i]==b[i+1]^b[i]` です。

Copyright © 2025 Artgraphics. All rights reserved.

上記のコード変換を SystemVerilog の記述例としてまとめておきます。

例 5-1　Gray カウンターの記述例

前述の変換アルゴリズムを使用して、Gray カウンターを以下のように実装します。以下の記述では、バイナリーカウンターで値（binary）を更新し、更新されたバイナリーコード（binary）を Gray コード（q）に変換します。

```
module gray_counter #(NBITS=4)
    (input clk,reset,output logic [NBITS-1:0] q);
logic [NBITS-1:0]   binary;

always @(posedge clk,posedge reset)
    if( reset == 1 )
        binary <= 0;
    else
        binary <= binary+1;

always @(binary) begin
    q[NBITS-1] = binary[NBITS-1];

    for( int i = NBITS-2; i >= 0; i-- )
        q[i] = binary[i+1] ^ binary[i];
end

endmodule
```

バイナリーで数え上げる

バイナリー値を Gray コードに変換する

変数 binary に対して NBITS 個のフリップフロップが生成されます。そして、組み合わせ回路を表現する always プロシージャは NBITS-1 個の XOR で構成されます。

■

Gray カウンターの回路構成はほとんど明らかであると思えます。バイナリーの数え上げでは加算器が必要となりますが、ハーフアダーだけで十分です。この場合には、3 つのハーフアダー（ha）と一つの INV 回路だけで十分です。図 5-2 は 4 ビットの変数をインクリメントする回路構成を示しています。sum[0]は a[0]^1 となりますが、a[0]^1 は~a[0]のように簡略表現できます。なお、a[0]は現在の状態の binary[0]です。そして、sum[0]は次の状態の binary[0]です。以下同様です。

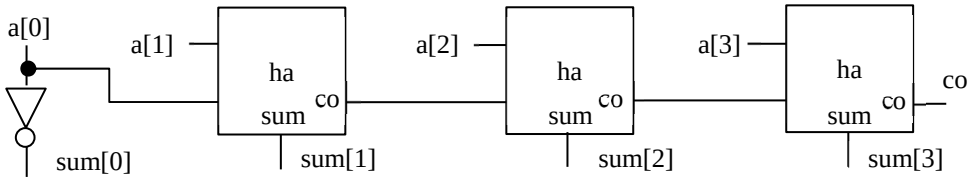


図 5-2　4 ビット変数のインクリメント回路の構成

以上を表 5-2 にまとめておきます。

表 5-2　Gray カウンターに必要な回路構成

必要な回路	必要な数
フリップフロップ	NBITS
ハーフアダー	NBITS-1
XOR	NBITS-1

Copyright © 2025 Artgraphics. All rights reserved.

10　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。

10.1　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。現在の状態は、レジスタで記憶されます。既に紹介したカウンター（例えば、Gray 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 10-1 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。

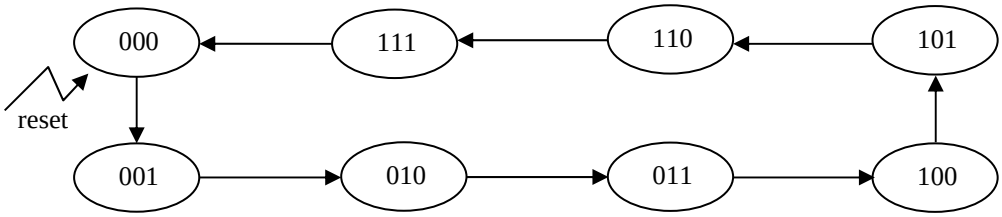


図 10-1　3 ビットのバイナリーカウンターの状態遷移

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 10-2 は Odd パリティチェッカーを示しています。

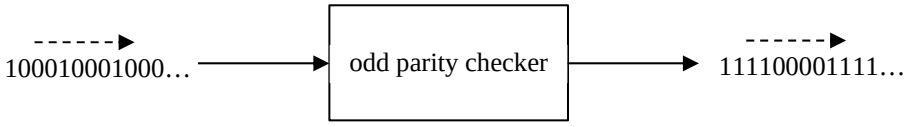


図 10-2　パリティチェッカーのブロックダイアグラム

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 10-3）。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。

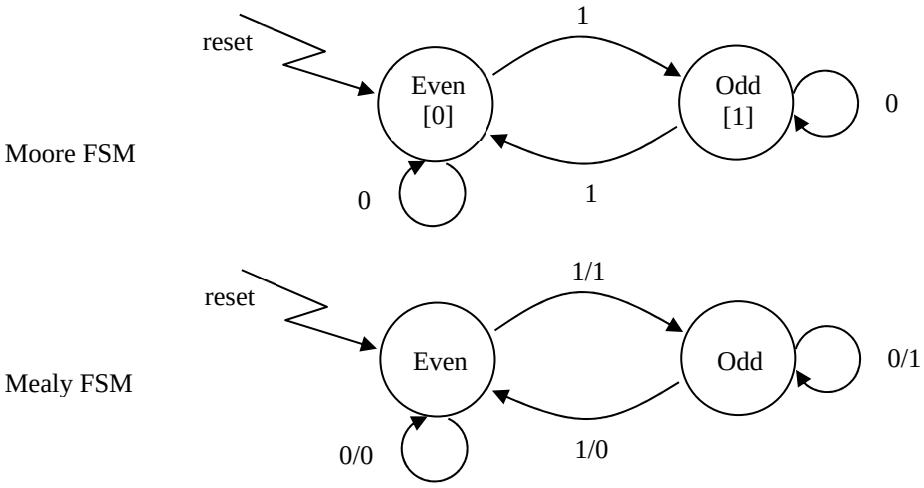


図 10-3　パリティチェッカーの状態遷移図

Copyright © 2025 Artgraphics. All rights reserved.

Mealy FSM では、入力に対して出力をアークに記します。例えば、現在 Even の状態にあり、ビット 1 を入力すると、次の状態は Odd（つまり 1）に移るのでアークには 1/1 が付けられます。他の場合も同様です。

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、図 10-3 の Moore FSM の状態遷移図から表 10-1 のような関係が得られます。Moore FSM の場合には、出力は現在の状態と一致するので、出力と次の状態は異なります。

表 10-1　パリティチェッカーの状態遷移表（Moore FSM）

現在の状態 (PS)	入力(I)	次の状態(NS)	出力 (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、表 10-2 のような真理値表が得られます。Moore FSM では出力と現在の状態が一致します。

表 10-2　パリティチェッカーの真理値表（Moore FSM）

現在の状態 (PS)	入力(I)	次の状態(NS)	出力 (O)
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

これから、以下のようなブール式を得ます。

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

一方、Mealy FSM の状態遷移図から表 10-3 のような関係が得られます。この場合には、出力は入力と現在の状態により決定されます。つまり、出力は次の状態に一致します。

表 10-3　パリティチェッカーの状態遷移表（Mealy FSM）

現在の状態 (PS)	入力(I)	次の状態(NS)	出力 (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 10-4 のような真理値表が得られます。

表 10-4　パリティチェッカーの真理値表（Mealy FSM）

現在の状態 (PS)	入力 (I)	次の状態 (NS)	出力(O)
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

これから、以下のようなブール式を得ます。

Copyright © 2025 Artgraphics. All rights reserved.