卒 業 研 究 報 告

題　　目

電子サイコロの設計と製作

指 導 教 員

矢野 政顕 教授

報 告 者

学籍番号：1030195
氏名：貞野 雅俊

平成 15 年　 2 月　10 日

高知工科大学 電子・光システム工学科

目次

第 1 章　はじめに

今日のエレクトロニクス技術の発展はめざましいものがある。それを支えているのが、半導体技術とこれを応用した電子回路の設計技術である。

身近にあるものとして、携帯電話、家庭用電化製品、パソコンなどがあり、半導体技術の向上とともに、高性能かつ小型化を目指し、技術の発展は続くと考えられる。

卒業研究では、具体的な例として電子サイコロを題材とし、ディジタル回路の設計、製作と VHDL を利用した設計の一連の流れを習得しようと考え設計した。これらの設計を通して、標準ロジック IC によるディジタル回路の設計、VHDL による回路設計の理解を深めることができた。

本報告は 5 章から構成されている。第 1 章では研究の背景を述べ、第 2 章で順序回路の設計に関する基本事項について説明する。第 3 章では、実際に IC を使い具体的な例として電子サイコロを題材にして回路の設計方法、また実装についての流れについて述べる。第 4 章では、ハードウェア記述言語による設計手法と、その実際例として設計した電子サイコロについて報告する。第 5 章で全体をまとめる。

第 2 章　　順序回路

2-1　順序回路とは

順序回路（sequential circuit）とは、入力だけでなく過去の入力によって得られた現在の状態にも影響されて次の状態と出力が決まる回路である。順序回路のブロック構成を図 2.1 に示す。過去の状態をフィードバックして記憶素子で記憶させ、それを現在の状態（内部状態ともいう）とする。順序回路はこの現在の状態と入力を組合せ回路に入力して次の状態と出力を得るもので、組合せ回路と記憶回路の組合せで構成されるものである。[1]

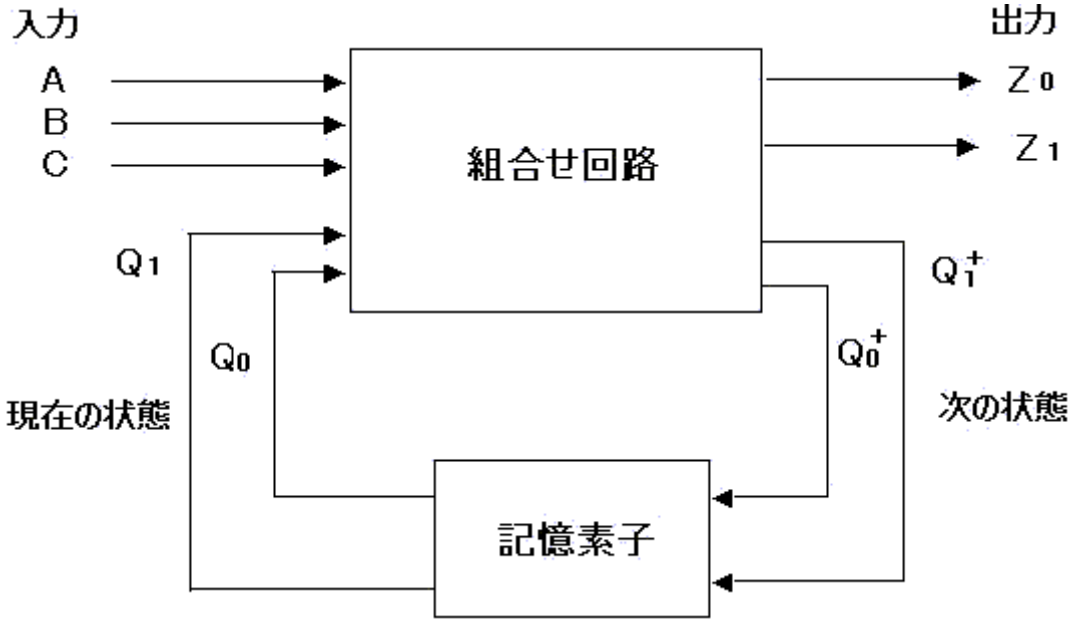


図 2.1 順序回路のブロック構成

順序回路の基礎

順序回路は組合せ回路と記憶回路からなる。図 2.1 に示すように組合せ回路は、その出力を記憶回路を経てその入力にフィードバックさせた回路である。過去の入力による論理操作の結果は記憶回路で記憶され（これを内部状態あるいは単に状態と呼ぶ)、現在の入力と統合し組合せ回路に入力される。このため、出力は過去の入力に依存することになる。すなわち記憶回路の遅れを Δt とすると、ある時刻 t の順序回路の出力は、その時刻の入力と記憶回路の内部状態として記憶されている Δt 前の組合せ論理回路の出力によって決まることになる。

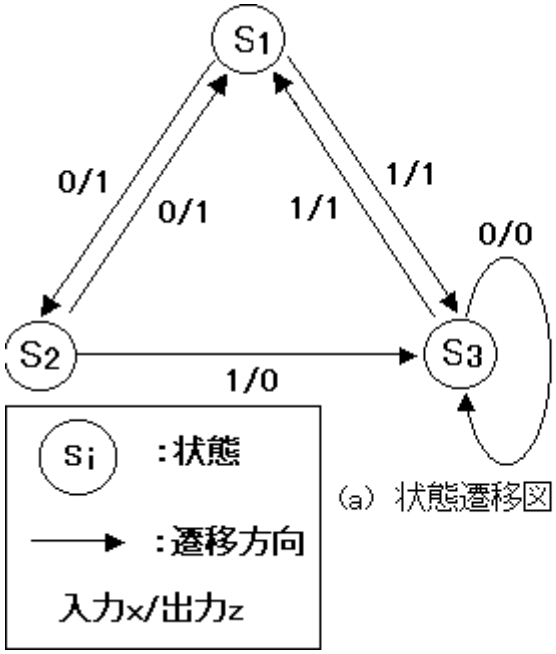
組合せ論理回路ではその時の入力と出力だけを考えればよかったが、順序回路を考えるにはこれまでに述べたように時刻ごとに異なる状態という概念が必要になる。

状態遷移図と遷移表

順序回路の動作をわかりやすくするため、状態遷移図（state transition diagram）や状態遷移表（state transition table）が用いられる。これらは単に状態図､状態表とも呼ばれる｡その状態遷移図と状態遷移表を図 2.2 に例を示す。

状態遷移図では､一般に状態は○印で表され､そのなかに状態名が書かれる。また、状態間の遷移は矢印で表され、その側に遷移時の入力（／の前）と出力（／の後）が書かれる。図 2.2（a）の例では s1～s3 の 3 つの状態があり、その間は矢印のように遷移することを示している。

遷移表では、一般に左側に現在の状態を書き、中央に次の状態、右側に遷移後の出力を各入力状態（入力値の組み合せ）ごとに書く。図 2.2（b）は図 2.2（a）を遷移表で表したものである。



(a) 状態遷移図

現在の状態	次の状態 入力x 0	次の状態 入力x 1	出力z 入力x 0	出力z 入力x 1
S1	S2	S3	1	1
S2	S1	S3	1	0
S3	S3	S1	0	1

(b) 遷移表

図 2.2 状態遷移図と遷移表

2-2　フリップフロップ

順序回路の状態を記憶するには、フリップフロップ（flip-flop：FF）が用いられる。フリップフロップは二つの安定状態をもつ 1 ビットの記憶素子であり、適切な入力によってほかの安定状態に移る。

フリップフロップの原理

図 2.3（a）に示すように、2 個の NOT 回路 N1、N2 の入出力を相互に接続した回路の動作を考えてみる。

N1 の出力 Q1 が H レベルであるとすると、N2 の入力は Q1 に直結されているから H レベルであり、出力 Q2 は L レベルとなる。よって、N1 の入力は L レベルとなり、その出力 Q1 は当然 H レベルである。これは最初に仮定した Q1 のレベルと一致し、回路の状態は、Q1 が H レベル、Q2 が L レベルを保持することになる。

次に、何らかの方法により、Q1 を L レベルに変えると、同様の考え方により、Q1 が L レベル、Q2 が H レベルに保持される。

このように、図 2.3（a）の回路は二安定状態を有しており、これを 2 安定回路、あるいは双安定回路という。

フリップフロップは、2 安定回路の二つの安定状態を外部より選択できるようにした回路で、基本的には図 2.3（b）に示すように、安定状態を決定するための二つの入力（R，S）と、互いに相補の関係にある二つの出力（Q，$\overline{Q}$）を有している。

フリップフロップは、出力の状態の変化のさせ方により、いくつかの種類のフリップフロップが存在する。[5]

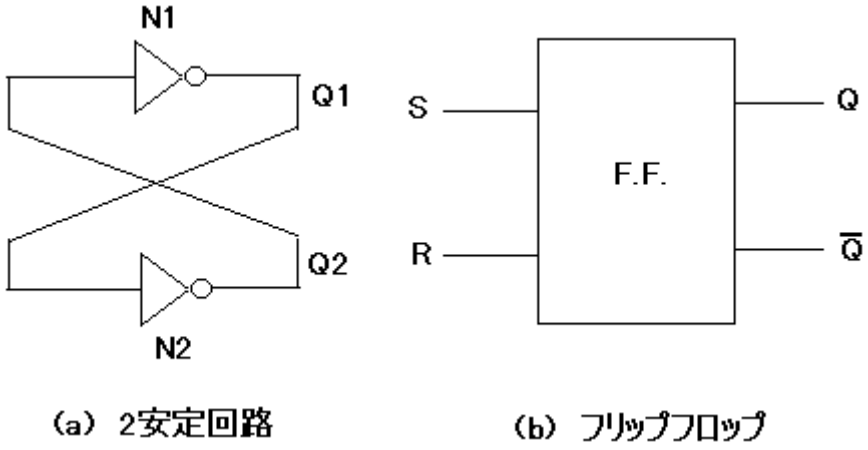


図 2.3　2 安定回路とフリップフロップ

SR フリップフロップ

SR フリップフロップ（set　reset　flip-flop：SR-FF）は文字どおり、入力としてセット S とリッセト R をもつ基本的なフリップフロップであり、その基本構成は図 2.4 に示す。すなわち、入力 S に 1 が印加されたときに出力 Q を 1 とし、入力 R に 0 が印加されたとき出力 Q を 1 とする（一般に、Q＝1 にすることを FF をセットするといい、Q＝0 にすることをリセットするという）。入力 S、R がともに 1 のときは以前の状態を保持する。また、入力 S、R がともに 0 のときは、Q、$\overline{Q}$ ともに 1 になるが、そもそも $\overline{Q}$ は Q の否定でなければならないので、禁止とする。

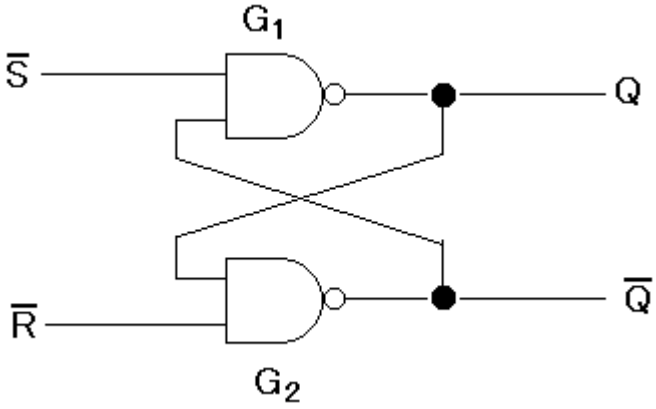
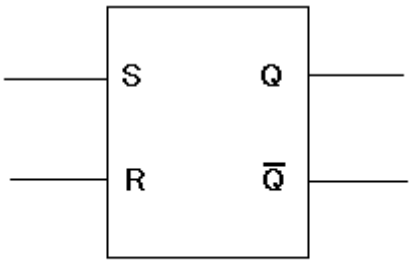


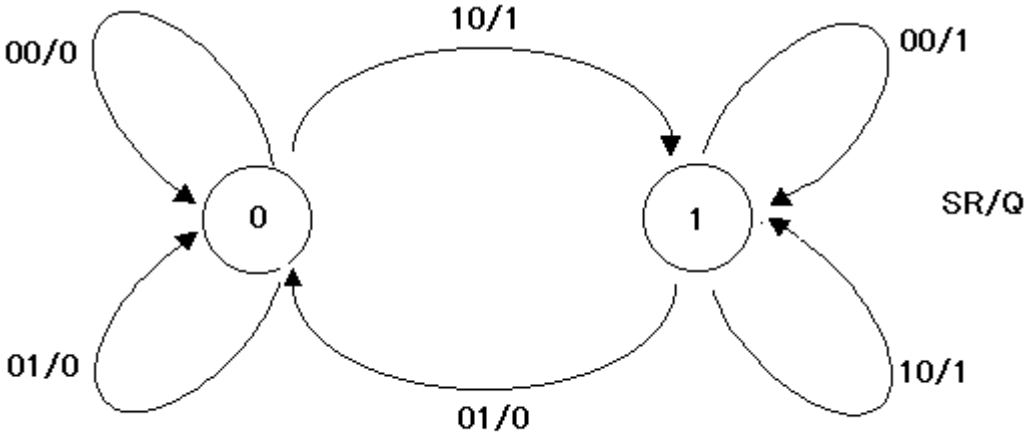
図 2.4　FF の基本構成（SR フリップフロップ）

SR-FF は図 2.5（a）のように表され、その動作は、出力 Q 自体を状態として扱うと図 2.5（b）の状態遷移図あるいは図 2.5（c）の遷移表で表される。ここ

で、Q、Q'はそれぞれ現在の状態、次の状態を表す。以上のように、FF でも一般的な順序回路と同様にそれぞれの動作を表せるが、その動作が比較的に簡単であるので図 2.5（d）のような表が使われる。これを特性表と呼ぶ。



(a) SR-FFの記号



(b)状態遷移図

現在の状態 Q	次の状態Q' 入力SR 00	01	10	11
0	0	0	1	×
1	1	0	1	×

(c) 遷移表

S	R	Q'
0	0	Q
0	1	0
1	0	1
1	1	×

×：禁止

(d) 特性表

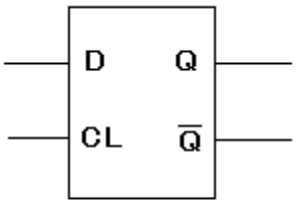
図 2.5　SR フリップフロップの表現

D フリップフロップ

D フリップフロップ（delayed　flip-flop：D-FF）は、D 入力とクロック（CL）入力をもち、CL が印加された時点の D 入力の値を記憶し、次に CL が印加されるまでその状態を保持する働きをする。なお、CL が印加されない時はもとの状態を保持する。このため、コンピュータにおいて、1 マシンサイクルの間各種のデータを保持するレジスタ回路あるいはシフトレジスタなどとして用いられる。

図 2.6（a）に D-FF の記号、（b）に特性表、（c）に基本構成、（d）にその動作を示します。特性表は CL が印加された時を示している。この FF は図 2.6（c）に示すように、基本的には同期式 SR-FF の R 入力を $\overline{S}$ とした構成をとる。また、特性方程式は特性表より次のようになる。

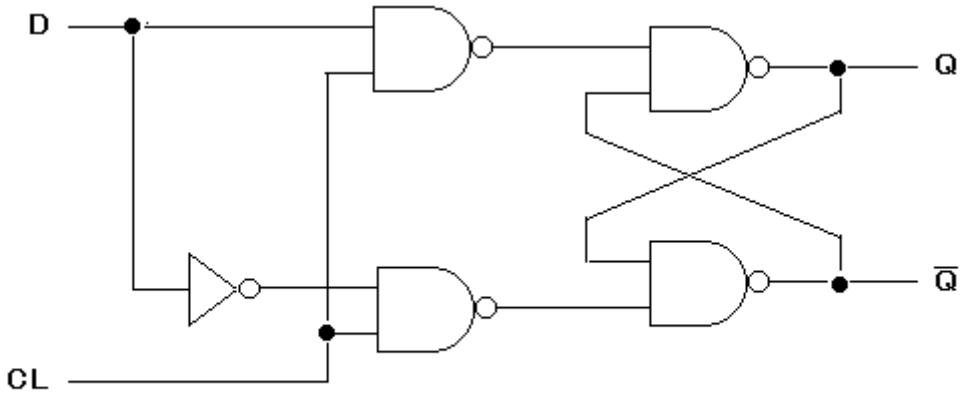
$$Q' = D$$



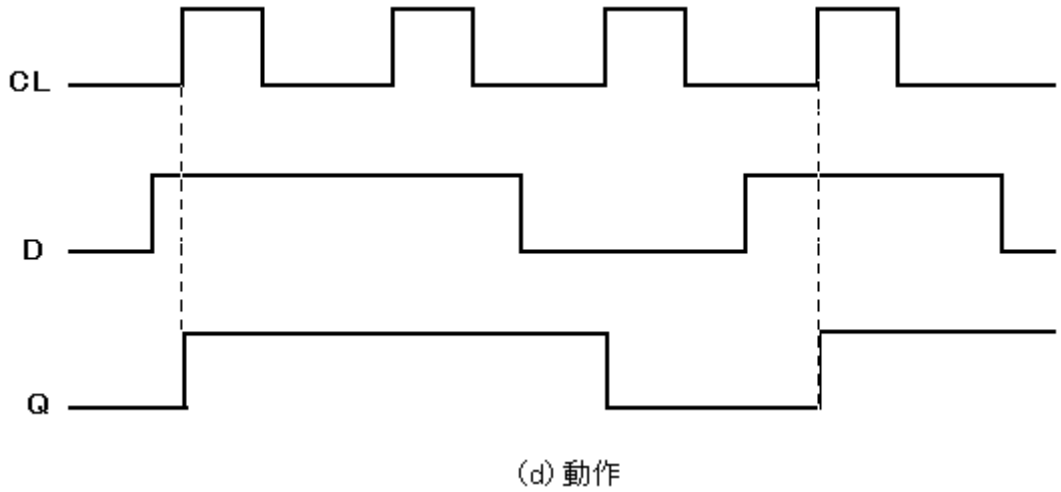
(a) D-FFの記号

D	Q'
0	0
1	1

(b) 特性表



(c) 構成



(d) 動作

図 2.6　D フリップフロップの構成と動作

フリップフロップはこの他にもいくつかの種類があり、動作や役割も異なっている。そのいくつかを例に挙げる。

・クロック付き SR フリップフロップ

・エッジトリガ型フリップフロップ

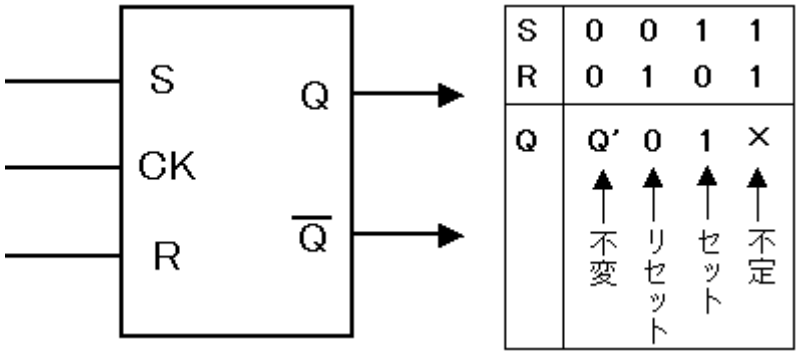
・JK フリップフロップ

・T フリップフロップ

ここで例に挙げたものについて、動作、回路構成などについて説明する。

クロック入力付き SR フリップフロップ

①S/R 入力とも“ 0 ”の時、Q 出力は不変；②S/R 入力とも“ 1 ”の時、出力は不定（“ 0 ”/“ 1 ”のいずれになるか分からない）；③S/R 入力のいずれかが“ 1 ”の時、Q 出力は強制的に“ 1 ”（セット）/“ 0 ”（リセット）；という動作をする。図 2.7 にクロック入力付き SR-FF を示す。[2]



Q＝Q（t+1）　Q’＝Q（t）

Q＝$\overline{R}$･Q’＋S　（S・R＝0）

図 2.7　クロック入力付き SR フリップフロップ

エッジトリガ型フリップフロップ

図 2.8 に示すように、状態変化がクロックパルスの立ち上がり（ポジティブエッジ）／立下り（ネガティブエッジ）によって生じるフリップフロップをエッジトリガ型フリップフロップという。エッジトリガ型フリップフロップには次のようなものがある。[2]

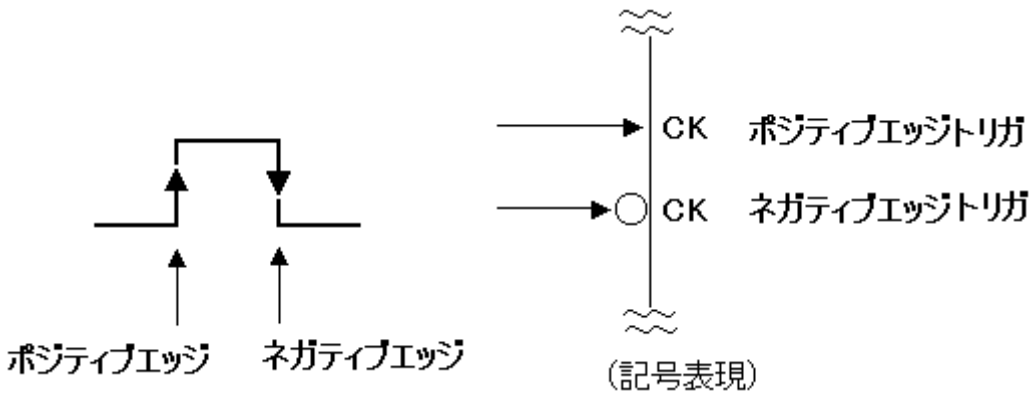


図 2.8　エッジトリガ型フリップフロップのクロック入力

JK フリップフロップ

SR フリップフロップの出力を入力にフィードバックし、J/K 入力と組み合わせたフリップフロップである。①J/K 入力のいずれかが“ 0 ”のとき、SR フリップフロップと同じ動作；②J/K 入力とも“ 1 ”の時、Q 出力が反転（トグル）；という動作をする。SR や D フリップフロップの機能を包含しているので、それらよりも適用範囲は広い。図 2.9 に JK-FF を示す。[2]

$$Q = \overline{K} \cdot Q' + J \cdot \overline{Q'}$$

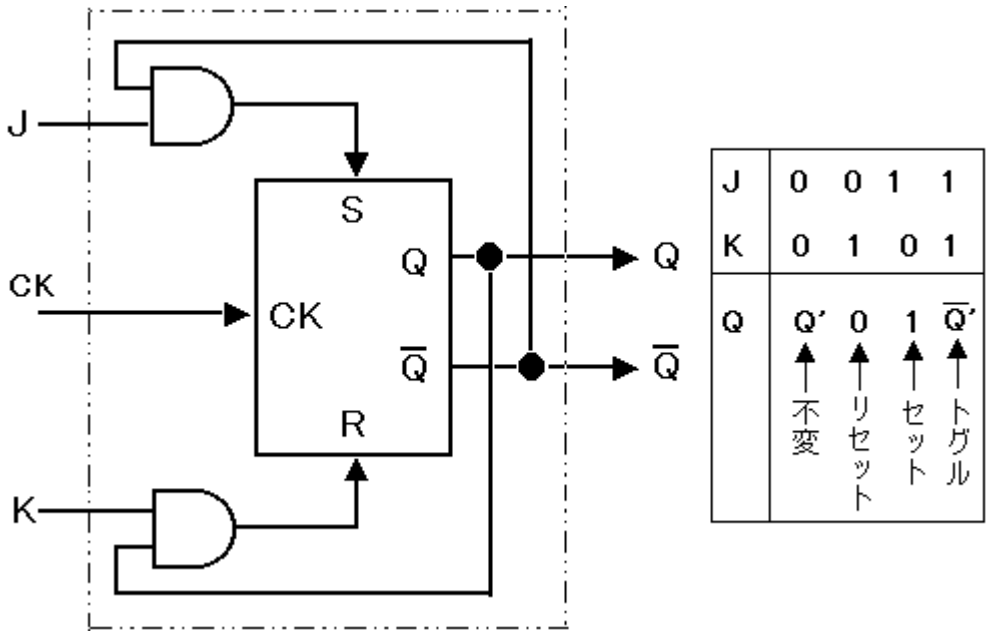
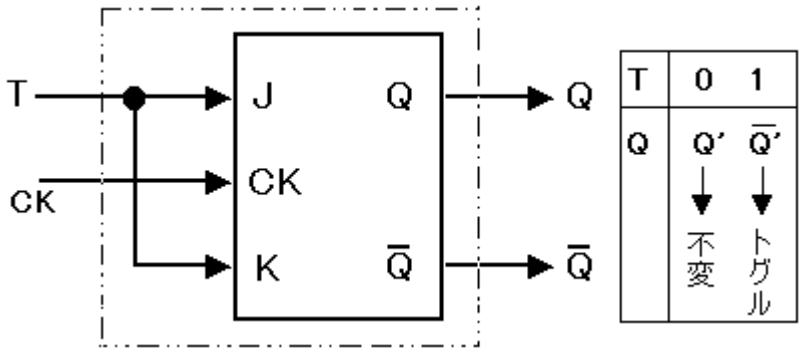


図 2.9　JK フリップフロップ

T フリップフロップ

JK フリップフロップの J 入力と K 入力を結合し T 入力としたフリップフロップである。①T 入力が“ 0 ”である時、状態（Q 出力）は不変；②T 入力が“ 1 ”である時、状態（Q 出力）を反転；という動作（これをトグル（toggle）動作という）をする。図 2.10 に T フリップフロップを示す。[2]



$Q = \overline{T} \cdot Q' + T \cdot = T \oplus Q'$

図 2.10　T フリップフロップ

2-3　ラッチとレジスタ

図 2.11 に示すような複数のフリップフロップを並べたものをラッチ（latch）とかレジスタ（register）という。“ レジスタ長（ビット）” とは、並べたフリップフロップ個数のことであり、各フリップフロップは同期して動作する。すなわち、ラッチやレジスタの動作にはフリップフロップ出力の読み出し（read）とフリップフロップへの書き込み（write）／状態更新とがあり、各ビットは同時（並列）に動作する。

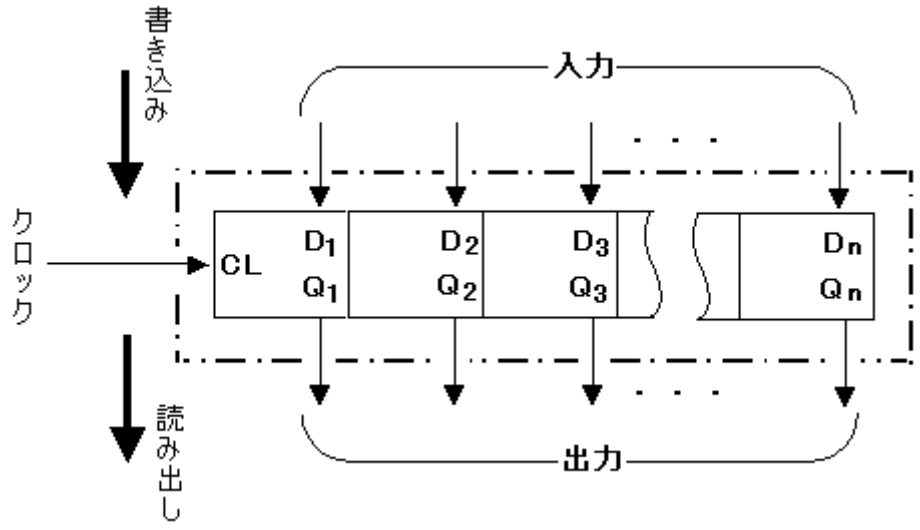


図 2.11　D フリップフロップによるラッチ／レジスタ（ｎビット）の構成

レジスタ

レジスタとは置数器のことで､1 バイトまたは 1 ワードの大きさでデータの一時記憶として使用する。例えば演算装置の汎用レジスタ、メモリのアドレスを指定するアドレスレジスタおよびメモリのデータを入出力するときの一時記憶に使用されるメモリレジスタなどがある。これらは並列に入力したデータをそのまま出力すればよいので D-FF の機能があればよい。図 2.12 に D-FF を用いた単方向並列入出力の 4 ビットレジスタを示す。[1]

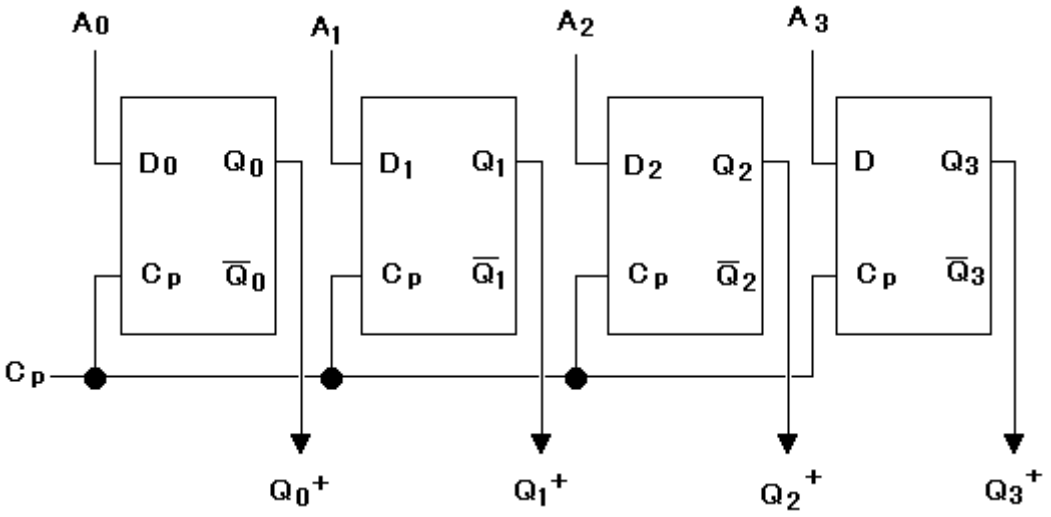


図 2.12　単方向並列入出力レジスタ

2-4　カウンタとは

カウンタ（Counter）回路は、時とともに到来するパルス状の信号の個数を計数（count）する回路である。この場合に、計数する信号が到来するごとに計数内容が 1 個増加（カウントアップ：count　up）する“ アップカウンタ ”と計数内容が 1 個減少（カウントダウン：count　down）する“ ダウンカウンタ ”とがある。

この計数をする形式は、任意の進数のものが設計可能であるが、純 2 進、10 進、12 進などが多用されている。また用途によっては、複数種の進数のカウンタをカスケードに接続する場合もある。そこでカウンタの出力は各桁ごとに“ 数値の重み ”が定義され、その組合せで計数内容が表現される。

付加機能としては

○計数内容を零にする“ クリヤ機能 ”（計数されるパルス状信号の到来に同期して実行する“ 同期型クリヤ機能 ”と、計数されるパルス状信号の到来に無関係に実行する“ 非同期型クリヤ機能 ”とがある。）

○任意の数値をあらかじめカウンタにセットする“ ロード機能 ”

○計数動作を実行しながら、ある瞬間の計数値を保持する“ ラッチ機能 ”とがある。

リプルカ・ウンタ

最下位段以外の各フリップフロップは、前段の出力（これをリプルキャリ（ripple　carry）という）によってトリガされるカウンタをリプル・カウンタという。図 2.13 にリプル・カウンタの構成を示す。

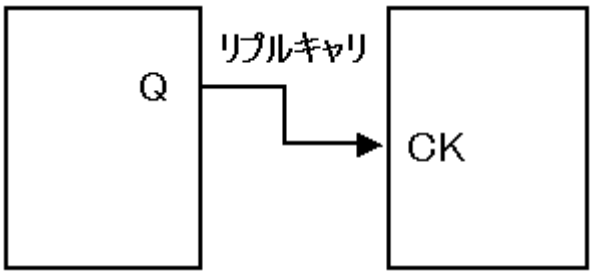


図 2.13　リプルキャリによるトリガ

リプル・カウンタでは、カウントパルスが最下位段から最上位段まで順番に伝播する。各フリップフロップの伝播遅延時間がカウンタの動作時間を左右し、フリップフロップは同期して動作しないので、リプル・カウンタを“ 非同期カウンタ ”ということもある。[2]

同期カウンタ

カウンタを構成するすべてのフリップフロップに同一のクロックが入力され、それらすべてのフリップフロップが同時に（同期して）状態変化するカウンタを同期カウンタという。[2]

2-5　デコーダとエンコーダ

デコーダ

デコード（decode）とは、入力の組み合わせ（2 進数パターン、コード）にしたがって対応する単一出力を生成することであり、コードの解読（復号）を行う機能である。デコード機能をもつ組み合わせ回路をデコーダ（decoder）という。図 2.14 に示すように、ｎ入力と 2^n（＝m）出力をもつデコーダを“ｎ×ｍデコーダ”という。デコーダの出力は相互に排他的であり、入力に対応する出力だけ 1 とし、残りすべては 0 となる。

デコーダは文字通りコードの解読機構であり、たとえば命令コードの解読などに利用できる。[2]

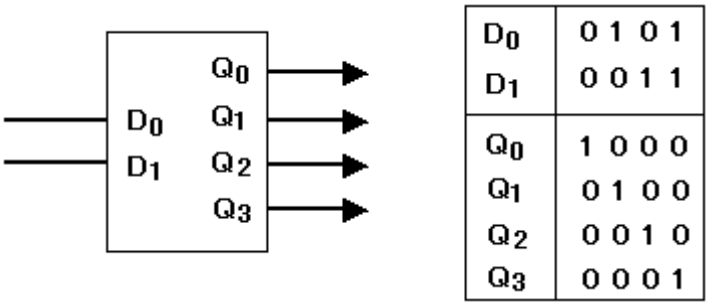


図 2.14　2×4 デコーダ

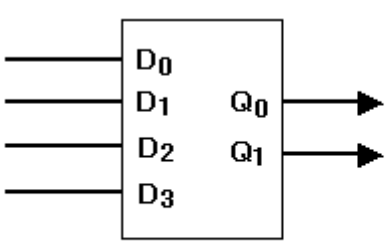
エンコーダ

エンコード（encode）とは、デコーダの反対の機能（すなわち符号化機能）であり、エンコーダ（encoder）はエンコードを行う組み合わせ回路である。エンコーダでは、複数入力のうち一つだけが 1 となり、1 になった入力に対応する出力パターン（コード）を生成する。図 2.15 に示すように、2^n（＝m）入力とｎ出力をもつエンコーダを“ｍ×ｎエンコーダ”という。[2]

単純なエンコーダでは、複数入力が 1 となると出力パターンから一意に入力を識別できない。たとえば図 2.15 において、$D_1 = D_2 = 1$、$D_0 = D_3 = 0$ の場合、Q＝11 となり、入力が D_3 だけであると誤認してしまいます。そこで、入力にプ

ライオリティ（priority）という優先順位づけをしてこの識別を可能にしたエンコーダをプライオリティエンコーダという。プレイ折ティエンコーダでは、優先順位の最も高い入力に対応する出力コードを生成する。上記の例では、$D_0 \rightarrow D_3$とプライオリティが高くなるというようにあらかじめ設定しておくと、“ $D_1 = D_2 = 1$ の場合でも Q＝10 で優先順位の高い入力がD_2である ”と識別することが可能となる。

プライオリティエンコーダの応用としては、割り込み検出回路などがある。



D_0	1 0 0 0
D_1	0 1 0 0
D_2	0 0 1 0
D_3	0 0 0 1
Q_0	0 1 0 1
Q_1	0 0 1 1

図 2.15　4×2 エンコーダ

第 3 章　　電子サイコロの作成

3-1　サイコロの動作

サイコロの使い方

サイコロの使い方としては、サイコロを振ってでた面の目の数を、その時の数字として扱いそれをゲームなどで使ったりするのが一般的である。サイコロとは、振ると 1 から 6 までの数字のうち一つだけが選ればれる。2 と 3 の間、例えば 2.5 などという数字はサイコロには存在しない。サイコロは 1～6 までの六つの数字だけを扱うことになる。そして、それぞれの数字の目が出る確率は 1/6 である。

サイコロの機能

サイコロの機能を考えると、次のような機能が考えられる。

サイコロの機能として大きく分けると 3 つの機能がある。まず、一つの機能としてサイコロを振るという機能がある。二つ目の機能としてサイコロが転がるという機能がある。三つ目の機能としては転がっているサイコロが止まりサイコロの目が決定するという機能とう 3 つに分けられる。

サイコロの機能として 1～6 までの数字のうち、毎回一つだけがランダムに選択される装置を実現させる。

電子サイコロの機能をもっと簡単に表現すれば、スイッチを ON にすると 1 から 6 までの中の数字が一つ選ばれる装置である。そして、スイッチ入れるたびにどの数字が選ばれるかわからないという点が重要である。[6]

機能から構成を決定

サイコロの機能が決定したところで、どのような構成でどのような回路を作るかというのを決定しなければならない。回路の構成を決定する際にもっとも大切なのが、入出力の部分である。外部からの信号をどのようにして取り入れるか、また回路動作をどのようにして表示するかという点が重要である。

外部からの入力信号、すなわちサイコロの目を決定するといった動作に当た

る信号は、スイッチを利用することで実現させる。そして、サイコロの目として 1〜6 までの 1 桁の数字をそのまま表示する場合は、7 セグメントの LED を用いれば可能である。

表示器として LED を利用してもサイコロの目を再現することは可能である。6 個の LED を用い、それぞれに 1 から 6 までの数字を当てはめて、一つだけ LED を点灯させれば、その点灯したところの数字を読むことで同じような機能が得られる。

しかし、もっと実際のサイコロに近い形の電子サイコロは考えられないだろうかと考え、サイコロの目の配置を図 3.1 に示す。この目の形に LED を配置することで、実際のサイコロに近いものが実現可能である。

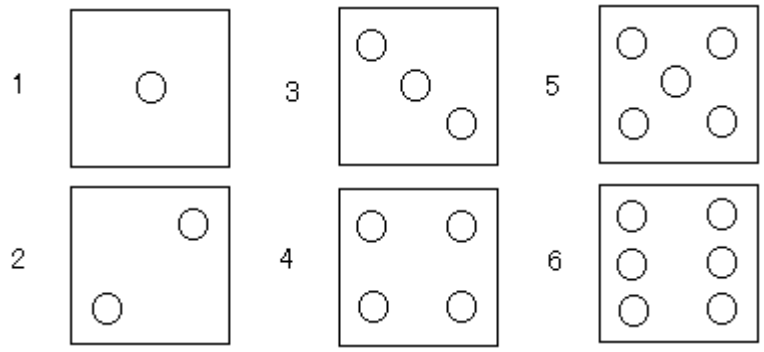


図 3.1　サイコロの目の配置

7 個の LED を実際のサイコロの目の形に配置しておき、それぞれの LED を点灯させることで数字を表現する。図 3.2 に LED の点灯例を示す。点灯した部分の LED が実際のサイコロの目の配置に近くなるように LED を配置し、どの LED を点灯させるかを回路によって構成する。[6]

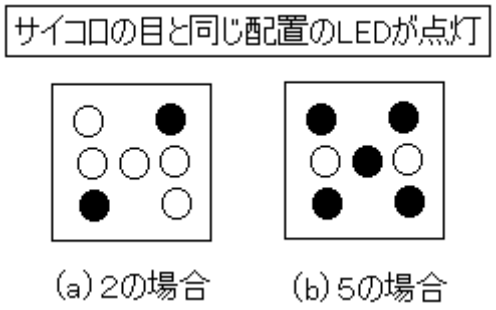


図 3.2　LED の点灯例

3-2　サイコロの目の構成

サイコロは 1～6 までの六つ目を選んで表示するわけであるが、サイコロのスタートスイッチを押すとサイコロが自動でサイコロの目を選択するといった機能がない回路の表示部のみの回路についての説明する。

7 個の LED でサイコロの目を実現

サイコロは 1 から 6 までの六つの数字しか使用しない。しかし、サイコロの LED の配置と実際のサイコロの目の配置から 7 個の LED が必要となる。そして、配置した LED のどれを点灯させるかで、サイコロの目を表現する。サイコロの 1～6 までの LED の点灯する位置を表した図を図 3.3 に示す。

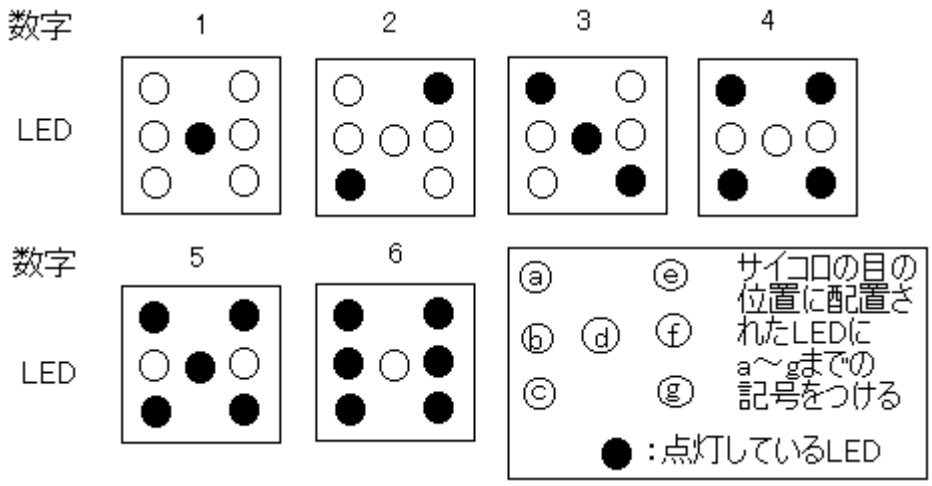
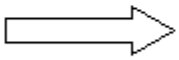


図 3.3　LED 点灯組み合わせ

配置した 7 個の LED にはそれぞれの位置を決めるために a～g までの記号を割り当てる。配置した 7 個の LED の点灯の組み合わせを全ての場合について考えると大変であるので点灯する組み合わせが同じ LED をグループ化する。グループ化の様子を図 3.4 示す。このグループ化した結果から四つのグループに分けられる。図 3.4 の（b）の表の意味は、ｃとｅの LED はサイコロの目の数の 2, 4，5，6 のとき点灯するという意味である。[6]

記号	数字
a	3,4,5,6
b	6
c	2,4,5,6
d	1,3,5
e	2,4,5,6
f	6
g	3,4,5,6

(a) LEDが点灯する時の数字



記号	数字
a , g	3,4,5,6
b , f	6
c , e	2,4,5,6
d	1,3,5

(b) 同じ組み合わせをまとめる

図 3.4　LED 点灯の組み合わせ

実際ロジック回路を実現させる時、簡単でわかりやすい方法でロジック回路を実現できないのだろうかと考えた。

例えばｃとｅのグループを考えてみる。この時、2，4，5，6 という四つの場合に点灯すればよいことになる。点灯するということは「サイコロの目になる」ということで、ロジックの“1”（“H”）を割り当てることにする。するとこの時は、4 入力の OR ゲートと同じ機能をもつことになる。

同様にして b，f は 1 入力の OR ゲート、a，g は 4 入力の OR ゲート、d は 3 入力の OR ゲートということになる。

多入力の OR ゲートはあまり一般的でないため、NAND ゲートを使用する。OR ゲートを NAND ゲートで代用するには図 3.5 のように入力信号をすべて反転させてやる。このようにすれば多入力の OR ゲートがなくても、NAND で同じ機能を持ったものを実現可能になる。

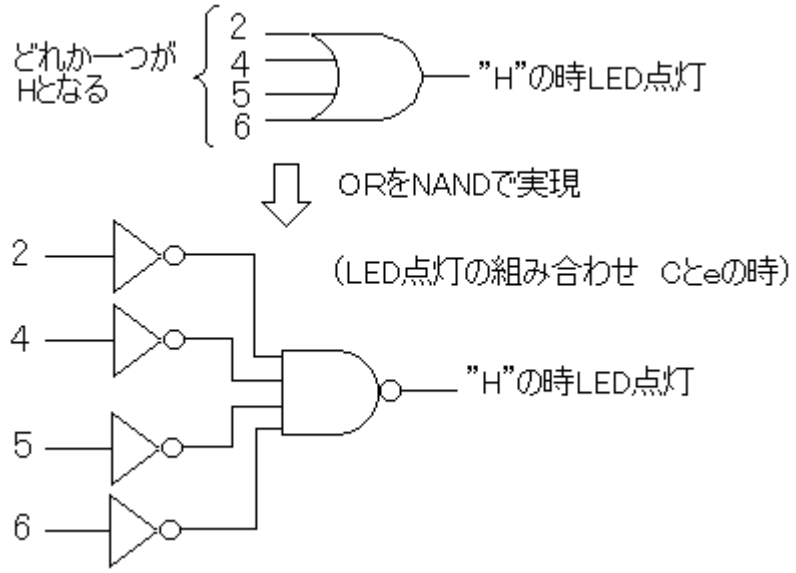


図 3.5　OR ゲートを NAND ゲートで実現

OR ゲートの NAND ゲートの真理値表を見比べて見ると同じロジックになっていることがわかる。OR ゲートと NAND ゲートで実現したロジックの真理値表を図 3.6 に示す。

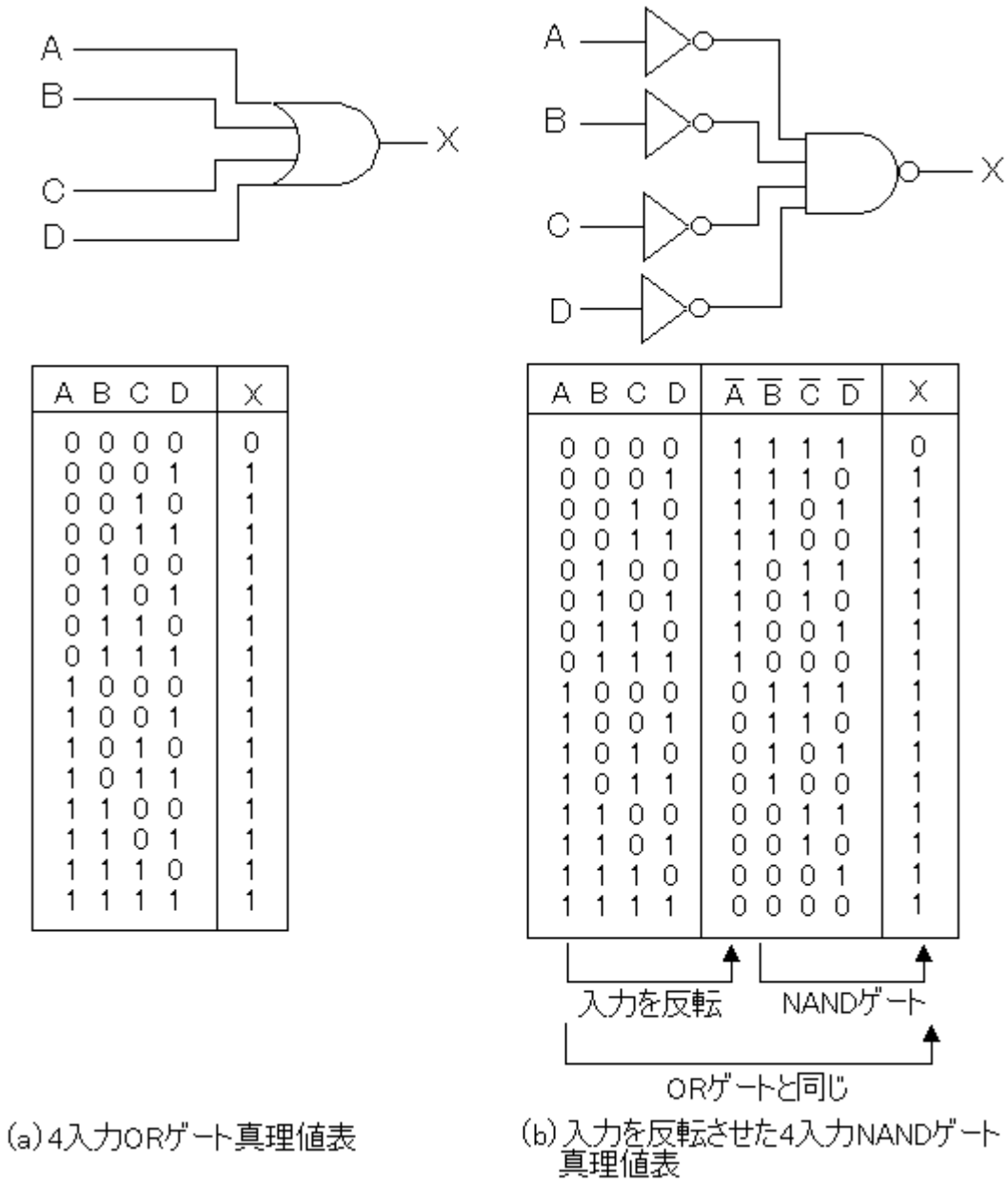
入力を反転させるためにはインバータが必要となるが、入力信号そのものを始めから反転させておけば省略することが可能である。

以上の考え方をもとに設計した回路の参考例を図 3.7 に示す。4 入力 NAND ゲートには 74HC20 を、3 入力 NAND ゲートには 74HC10 を使用したと想定している。

この回路は、回路の原理を理解のためであり実際に作成した回路とは異なる。NAND ゲートの出力が“ 1 ”になると、330Ωを通じて LED に電流が流れて LED が点灯する仕組みになっている。押しボタン・スイッチは 6 個使用している。それぞれの押しボタン・スイッチを押すと、押した時のみサイコロの目の形に LED が点灯します。押しボタン・スイッチを離すと OFF になるタイプのものを使用する。

また、二つ同時にスイッチを押すと正しいサイコロの目の形にはならない。

NAND ゲートを使用するため入力信号を反転する必要があると説明したが、スイッチを押すとグラウンド（“ 0 ”）、スイッチを離すと＋3V（47kΩを通じて）“ 1 ” になるので、反転したものと同じ信号レベルが NAND ゲートに加わるような回路構成を想定している。[6]



A	B	C	D	X
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

(a) 4入力ORゲート真理値表

A	B	C	D	$\overline{A}$	$\overline{B}$	$\overline{C}$	$\overline{D}$	X
0	0	0	0	1	1	1	1	0
0	0	0	1	1	1	1	0	1
0	0	1	0	1	1	0	1	1
0	0	1	1	1	1	0	0	1
0	1	0	0	1	0	1	1	1
0	1	0	1	1	0	1	0	1
0	1	1	0	1	0	0	1	1
0	1	1	1	1	0	0	0	1
1	0	0	0	0	1	1	1	1
1	0	0	1	0	1	1	0	1
1	0	1	0	0	1	0	1	1
1	0	1	1	0	1	0	0	1
1	1	0	0	0	0	1	1	1
1	1	0	1	0	0	1	0	1
1	1	1	0	0	0	0	1	1
1	1	1	1	0	0	0	0	1

(b) 入力を反転させた4入力NANDゲート真理値表

図 3.6　OR ゲートと入力を反転させた NAND ゲートの真理値表[6]

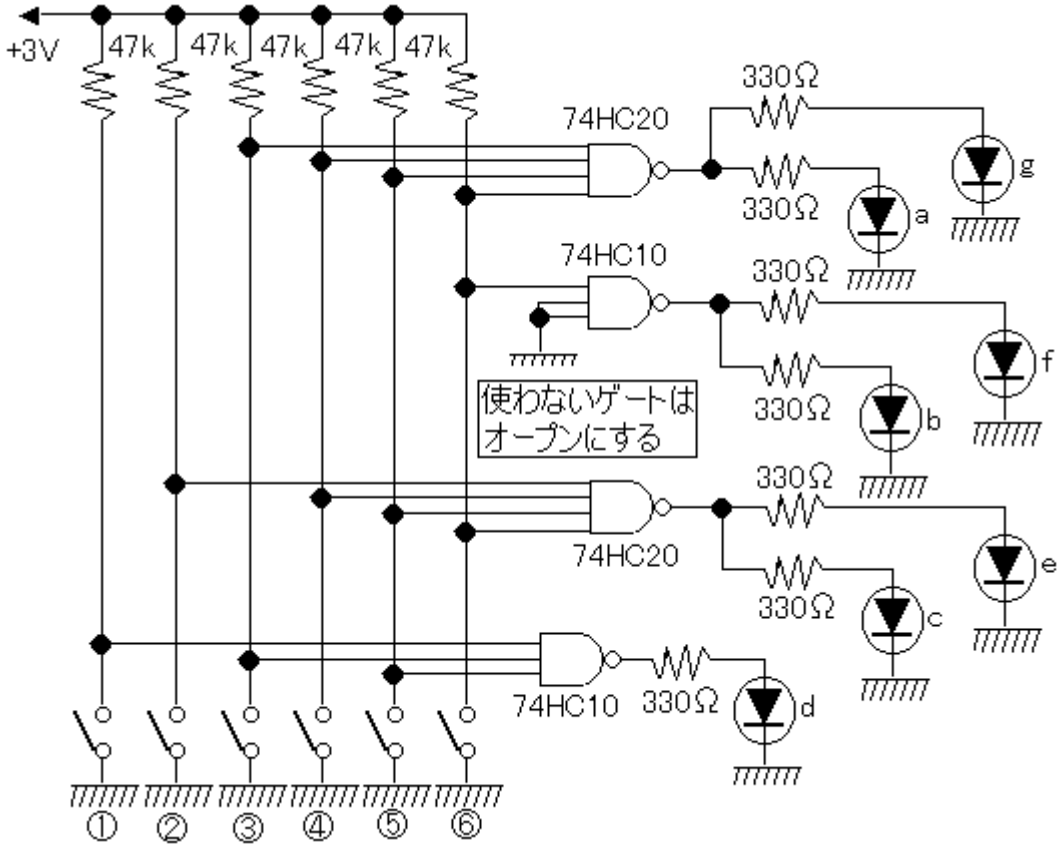


図 3.7　サイコロの目の実験回路[6]

3-3　機能ブロック図から回路設計

ブロック設計

サイコロの機能からブロック図を作成する時の考え方を図 3.8 に示す。回路設計を行う時はこの実際の機能からブロック図を作るという作業がもっとも重要な部分になる。

サイコロの機能は 3 つの機能に分けられる。サイコロの機能をもとに作成したブロック図を図 3.9 に示す。スイッチを入れることサイコロの機能としてはサイコロを振るという動作に当たるが、ここではカウンタのクロック制御としての役目をする。すなわち、クロック制御回路の出力が“ H ”になっているときだけ、カウンタがカウンタ動作をする。

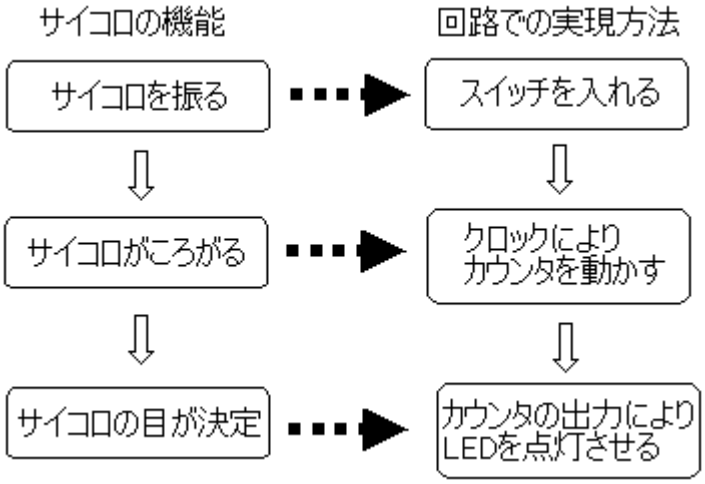


図 3.8　サイコロの機能からブロック図作成

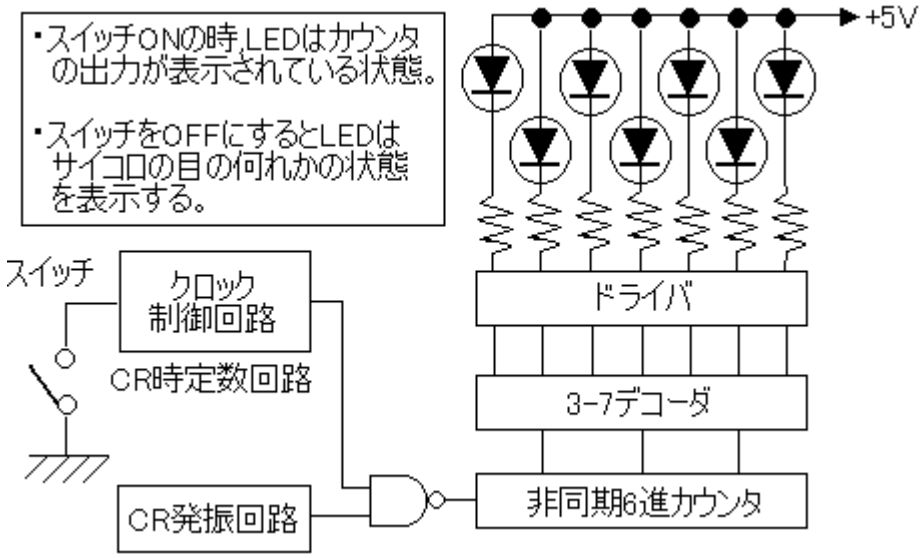


図 3.9　電子サイコロのブロック図

カウンタのカウンタ動作（1～6 まで数える）は、クロックを入れるか入れないかということで決定する。すなわち、カウンタにクロックを入力している時間というのがサイコロの機能としてサイコロが転がっている時の状態に当たる。

カウンタとしては非同期 6 進カウンタを用いる。サイコロの目は 1～6 までの 6 種類だから、出力状態（000　2 進）～（101　2 進）の 6 種類でよい。

そして、カウンタの 1～6 までの 6 種類の出力状態から、実際のサイコロの目の配置で LED を点灯するようにする。ここではデコーダを設計する作業が必要になる。

以上のようにサイコロとしての機能と構成がまとまったところで図 3.9 のブ

ロック図を基に、実際の回路を設計する。[6]

クロック発振回路

電子サイコロのクロック発振回路としては CR 発振回路を使用している。安定度という面からは、水晶発振回路を使用するのが最もよい方法である。しかし、CR 発振回路は抵抗やコンデンサの値を変更することで容易に発振周波数が変更できるとういうメリットがある。よって CR 発振器を使用することにする。クロック発振回路として使用した CR 発振回路を図 3.10 に示す。

作成した CR 発振回路の使用部品としては、クロック・バッファを含めてインバータ 4 個（74HC14）、抵抗 2 本（1MΩ,100kΩ）、コンデンサ 1 本（0.01μF）を使用する。

また、製作したサイコロのクロック発振回路の発振周波数は約 45Hz で製作している。

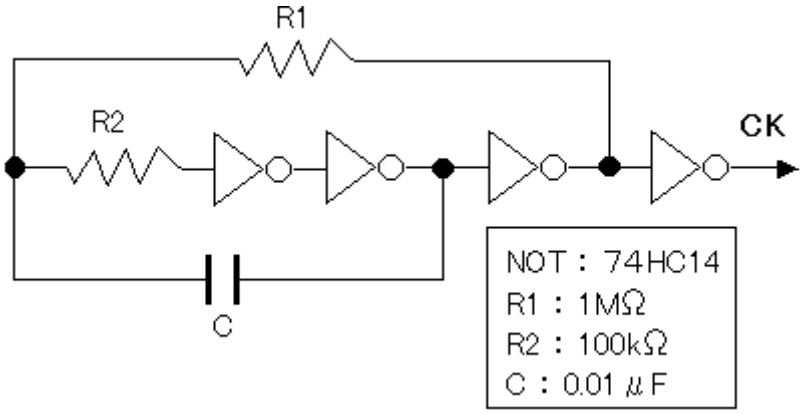


図 3.10　CR 発振回路

非同期 6 進カウンタ

電子サイコロの中心となるべき部分である。74 シリーズの中には、6 進カウンタの機能をもっている 7492 というものもあるが、古くてあまりファミリ化させていない（TTL スタンダード・タイプのみ）。そこで今回は、D フリップフロップ（74HC74）を使用してサイコロのカウンタを製作する。

6 進カウンタはｎ進カウンタのｎ＝6 の場合です。カウンタの種類としては、なるべく簡単な回路で作りたいということから、非同期カウンタとする。

非同期カウンタでｎ進のものを作成する場合、D フリップフロップを並べてその出力をデコードし、自分が設定している値（n=6）になった時にカウンタ全てをリセットすることで実現させる。

非同期 6 進カウンタの設計の手順および回路図を図 3.11 に示す。[6]

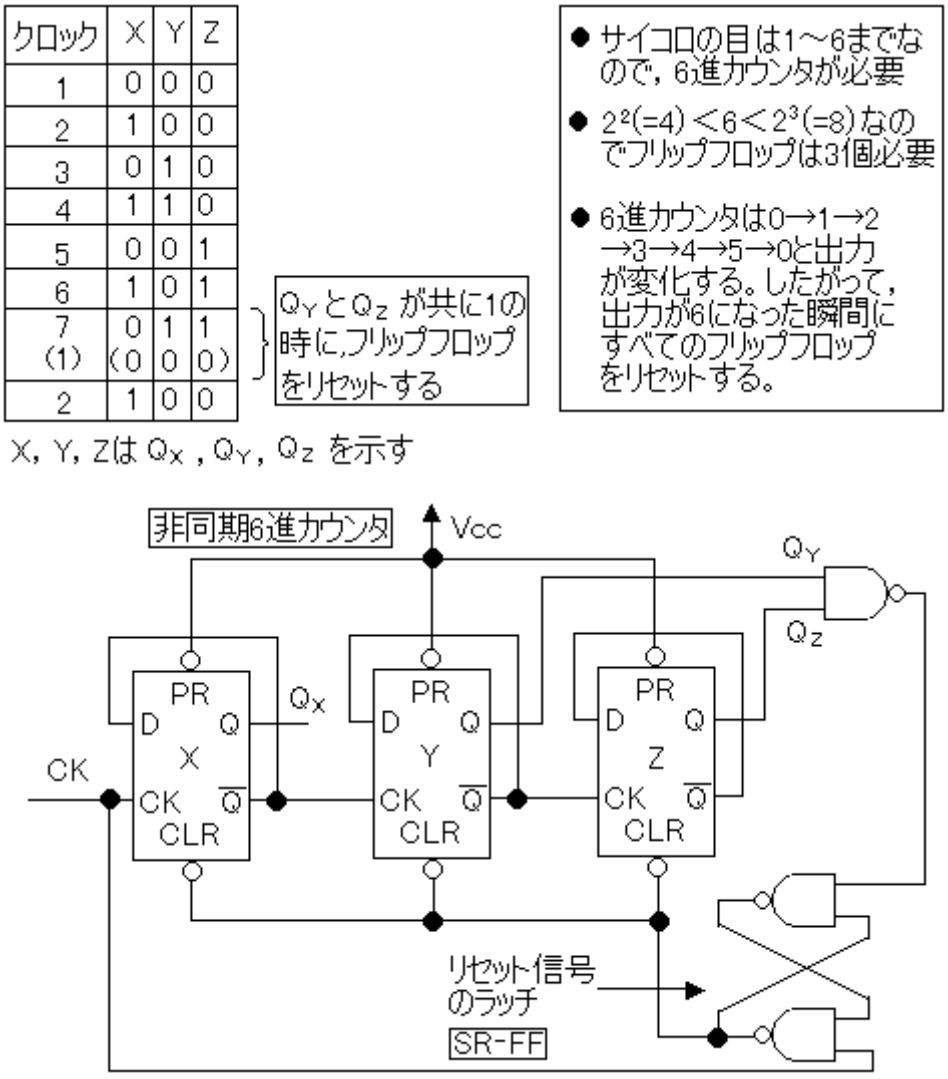


図 3.11　非同期 6 進カウンタ[6]

デコーダの設計

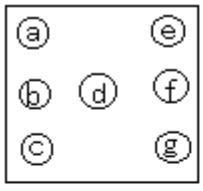
カウンタの出力に対応して、サイコロの目の形に LED を点灯させる信号を作る部分である。

6 進カウンタの出力は 10 進の 0（000　2 進）から 5（101　2 進）までの 6 種類である。それぞれの出力をそのまま LED で表示しても、2 進数での表示しか得られない。そこで 2 進数の 6 種類の値（000～101）からそれぞれの数のとき、どの LED を点灯させるかを決める。

この他にも 7 セグメントの数字表示器も、2 進数をセグメントのどの LED を点灯させるかということを決定するデコーダがあって、初めて数字として読める形になる。

サイコロの目の形に LED を配置すると、1～6 までの数値を 7 個の LED で表示することになる。サイコロ用デコーダの真理値表示を図 3.11 に示す。7 個配置した LED を a～g まで記号を割り当て、それぞれのカウンタ出力（X，Y，Z）に対して、どの LED を点灯させるかを決めたものが真理値表となる。

カウンタの出力とサイコロの目の対応としては、（X＝0，Y＝0，Z＝0）の時 1、（X＝1、Y＝0、Z＝0）の時に 2 というようになっている。したがって、カウンタ出力の 2 進数とはずれているが、特に問題はない。デコーダとしては、入力に対する出力が決定していればよいのであるから、かならずしも 2 進数とは一致していなくてもよい。[6]



a～gは
各LEDを表す

点灯LED
1 ： 点灯
0 ： 消灯

サイコロの目	LED							カウンタ出力		
	a	b	c	d	e	f	g	X	Y	Z
1	0	0	0	1	0	0	0	0	0	0
2	0	0	1	0	1	0	0	1	0	0
3	1	0	0	1	0	0	1	0	1	0
4	1	0	1	0	1	0	1	1	1	0
5	1	0	1	1	1	0	1	0	0	1
6	1	1	1	0	1	1	1	1	0	1

図 3.12　電子サイコロ用デコーダ

デコーダの入出力

サイコロのデコーダのブロック図を図 3.13 に示す。入力は X，Y，Z（3 ビット）、出力は a，b，c，d，e，f，g（7 ビット）です。X，Y，Z はカウンタの出力が接続される。したがって、（0，0，0）から（1，0，1）までの 6 種類の 2 進数が入力される。デコーダとしては、3bit 入力を 7bit 出力に変換するようなものを作成する必要がある。

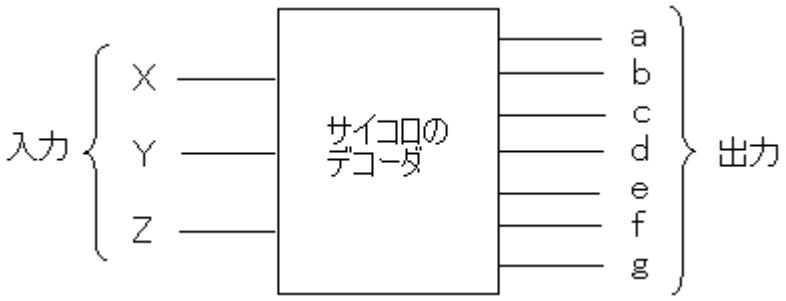


図 3-13　デコーダの入出力

これに対して、a，b，c，d，e，f，g の出力は、図 3.12 の真理値表の値によりデータが出力されるわけである。しかし、図 3.12 の真理値表を作成する前に確認しておかなければならないことがある。それは LED ドライバの極性を確認することである。“ 1 ”の時に LED を点灯させるのか、“ 0 ”の時に LED を点灯させるのかということである。

LED ドライバの極性を決定しないでデコーダ部のロジックを設計すると、回路を設計した後で余分なロジック回路を追加しなければならないこともある。しかし、今回使用する LED ドライバは“ 1 ”で LED を点灯するようになっているため、図 3.12 の真理値表をそのまま使えばよいことになる。[6]

カルノー図の簡単化

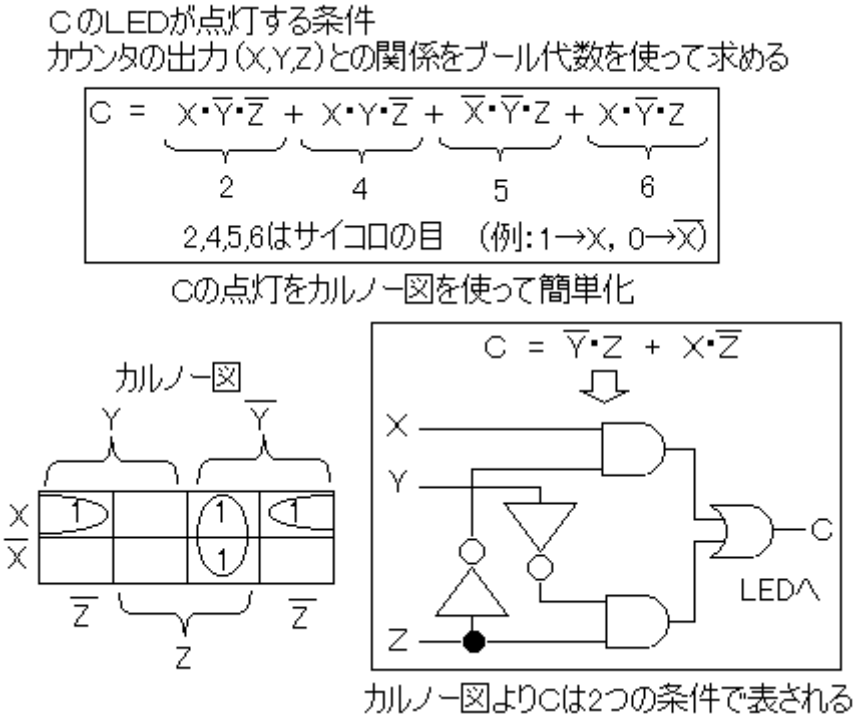
図 3.12 の真理値表をもとにデコーダ・ロジックを設計する。

まず c の LED を点灯させるための条件を求める。図 3.14 に論理を簡単化した過程を示す。最終的に得られたブール代数の結果によりロジックを組めばよいのであるが、もっと簡単にならないだろうかと考えた。図 3.15 にさらにロジックを簡単化する方法を示す。[6]

X，Y，Z は 3 ビットの組み合わせなので、（0，0，0）から（1，1，1）までの 8 通りある。しかし電子サイコロでは、そのうちの 6 通りしか使用しない。すなわち、（1，1，0）と（1，1，1）の組み合わせは、デコーダの入力データとして入ってこないことになる。

このことを利用して、c の点灯条件を見直してみる。入力として入ってこないデータは、デコーダの条件として使用しても使用しなくてもよいことになる。

入力条件が多くなると、デコーダのロジックが簡単になる。極端な例としては、すべての入力条件で出力が“ 1 ”の場合は、ロジック不要ということになる。



カルノー図よりCは2つの条件で表される

図 3.14　デコーダ・ロジック回路の設計[6]

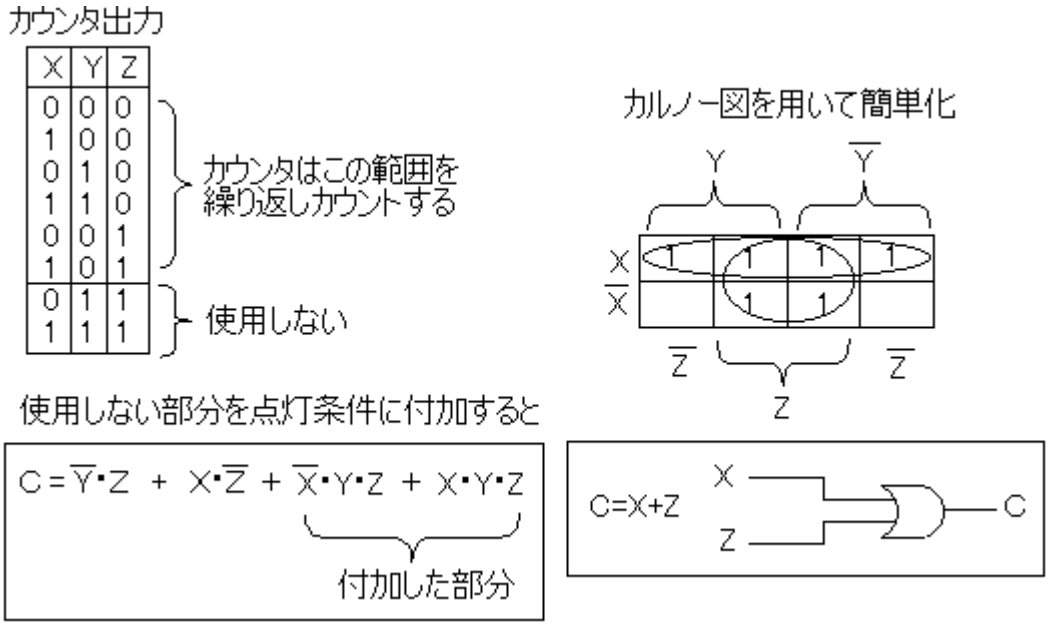


図 3.15　簡単化するための方法

図 3.15 の簡単化の結果、 c はXと Z の OR 出力ということになった。以上、同様の作業を a から g まで行った結果を図 3.16 に示す。

図 3.12 の真理値表を見ると、a と g、b と f、c と e はそれぞれ同じロジックでよいことになる。したがって a～g まで 7 回路分作らず、4 回路分でよいことになる。

図 3.16（b）は AND や OR といったロジックで構成されているが、実際の IC では NAND や NOR がよく使用されている。そこで NAND ゲート（74HC00）1 個ですべてのデコーダ回路を構成する例を図 3.16（c）に示す。

ここで、$\overline{X}$, $\overline{Y}$, $\overline{Z}$は D フリップフロップの$\overline{Q}$出力を利用できるので、ロジックは追加となりません。[6]

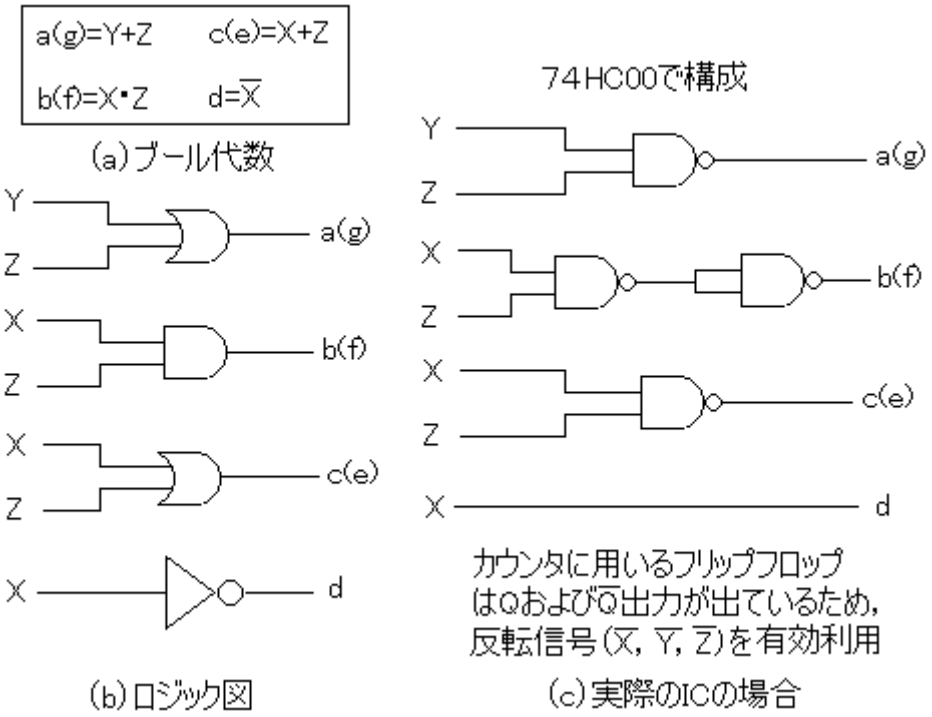


図 3.16　最も簡単化したサイコロ用デコーダ

LED ドライバ

LED を点灯させるドライバとしては、ドライバ IC（74HC04）を用いる。この IC はロジックの“ H ”レベルを加えると、出力トランジスタが ON（“ L ”レベル）となるものである。＋5V から LED と抵抗を通してドライバ IC に接続します。電流制限用の抵抗として 470Ωを使用しているので、LED には約 6mA 程度流れることになる。[6]

クロック制御回路

クロックの入力を ON または OFF と決定する制御回路としては、CR の時定数回路を利用する。使用する制御回路を図 3.17 に示す。

スイッチを ON にすると、4.7μF のコンデンサの電荷が 0 となる。そして＋5V に接続されている 100kΩの抵抗を通じてスイッチに電流が流れる。そして、インバータの入力には“ L ”レベルが得られるため、出力は“ H ”となる。

したがって、クロックを制御する NAND ゲートが開き、発振器からクロックがカウンタに供給されるわけである。そして、スイッチを OFF にすると、4.7μF のコンデンサに 100kΩを通じて＋5V から電流が流れ、電位が徐々に上昇

してくる。また、スイッチの ON と OFF を見分ける為の LED を制御回路につけている。スイッチ ON の時に制御回路の LED が点灯する。OFF にすると消える。

コンデンサの電位がインバータのスレッショルド・レベルを超えると出力が“L”となり、クロックを制御する NAND ゲートを閉じ、カウンタへのクロックを閉じることになる。[6]

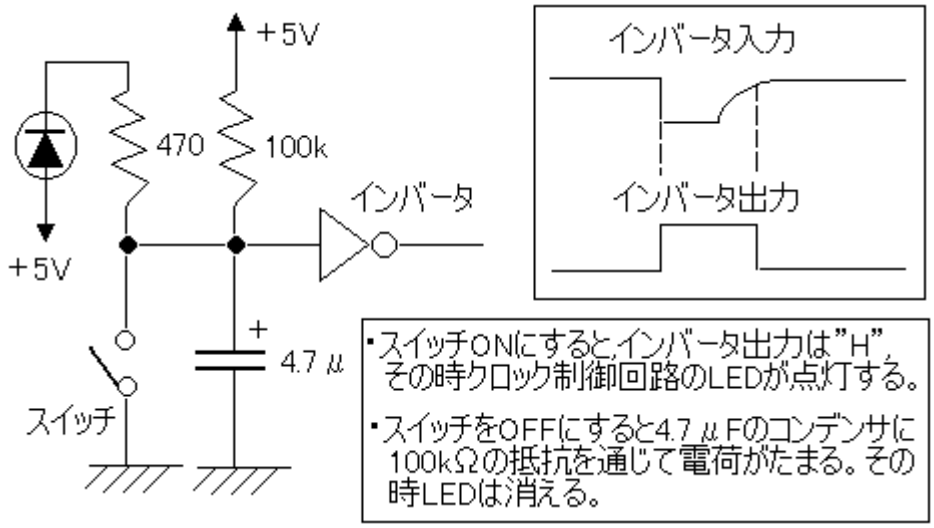


図 3.17　CR 時定数回路

電子サイコロの回路図および製作

以上の各ブロックごとの設計をまとめると、回路が完成である。製作した回路図を図 3.18 と図 3.19 に示す。実際に基盤に実装したものを図 3.20 に示す。

全部の部品の組み立てが終了したら、配線を確認してから電源を入れる。

電源を入れると出力が出され、このカウンタが動いている状態がサイコロが転がっている状態なので、スイッチを ON にすることで、サイコロの目が決定する。また、スイッチを OFF にする事でカウンタが再び動きだすようになっている。

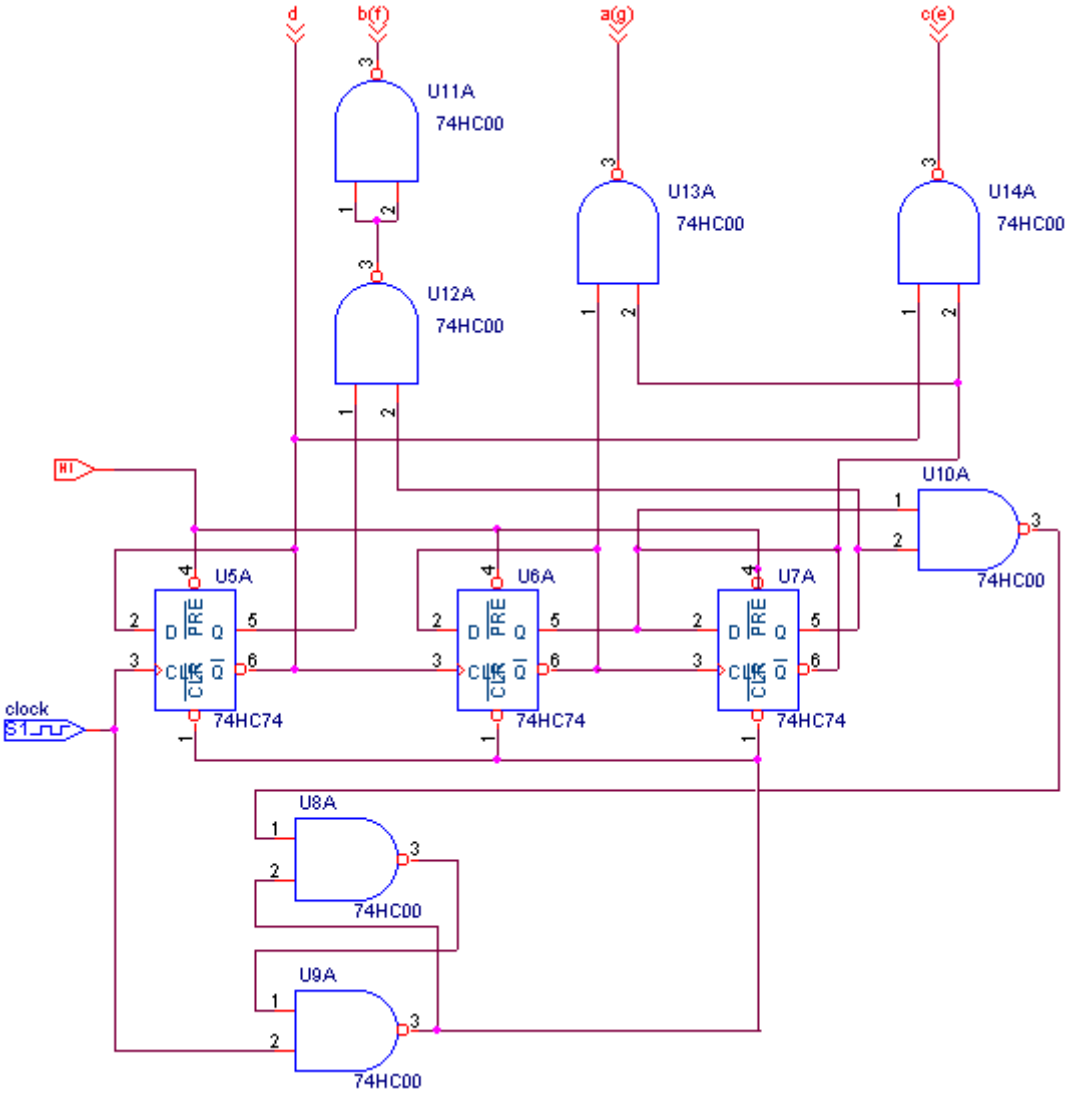


図 3.18　電子サイコロのカウンタと出力部の回路

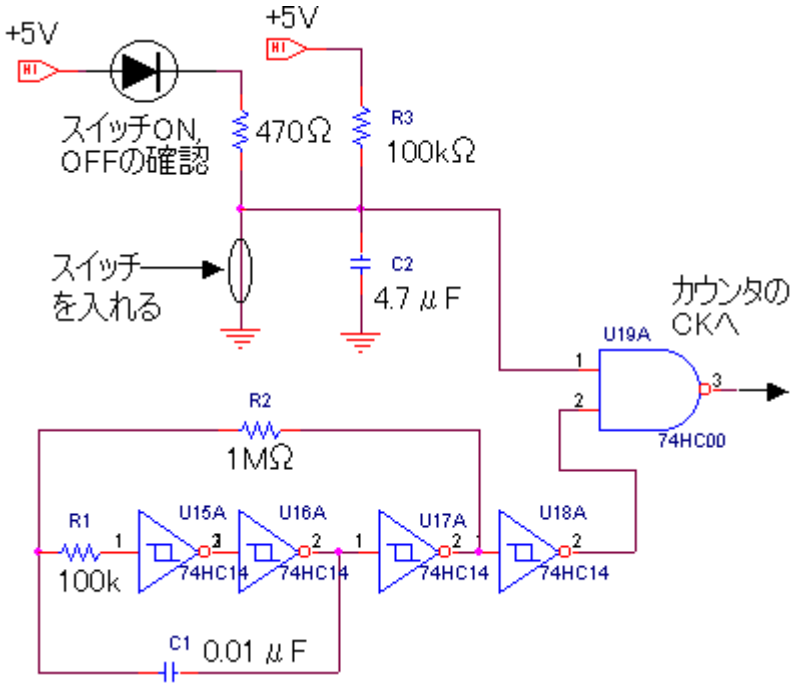


図 3.19　電子サイコロのクロック制御回路と CR 時定数回路

実際に実装した回路

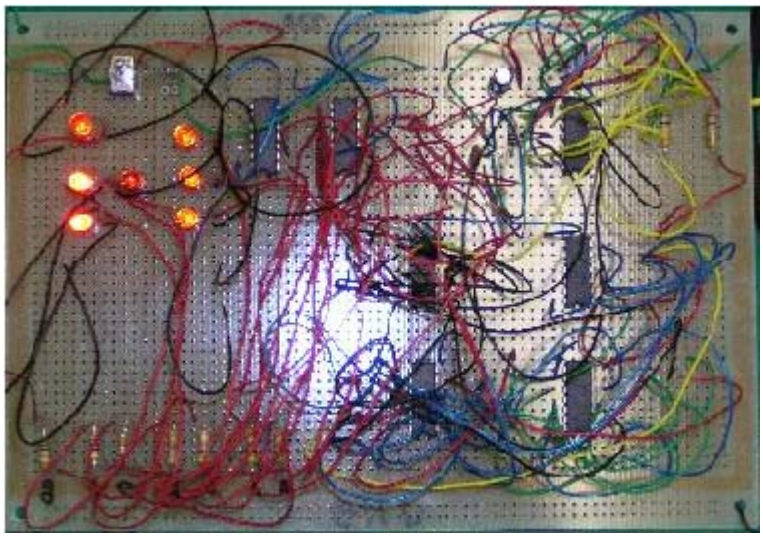


図 3.20　電子サイコロ

上の図 3.20 はカウンタを動かしスイッチを押して止めた時の画像（サイコロの目は 6 を示している）である。

製作した回路の動作

回路が完成して電源を入れるとカウンタが動作し、サイコロの目がついたり、消えたりする。実際はそのついたり消えたりしているのは目では確認することはできない。

カウンタを動作している時はスイッチ ON,OFF の確認のための LED が点灯している。しかし、サイコロの目がスイッチによりサイコロの目を止めると同時にスイッチ確認の LED が消える。

第 4 章　　ハードウェア記述言語

4-1　VHDL とは

HDL 設計のメリット

HDL（ハードウェア記述言語）による設計手法は、すでに ASIC（特定用途向け集積回路）などの大規模集積回路の設計でさかんに利用されている。HDL による設計は ASIC に限らず、FPGA や PLD などを使用した比較的小規模な設計にもさまざまなメリットをもたらす。

HDL による設計は、より抽象度の高いレベルで設計することにより、設計期間を短縮することができる。また、抽象度の高い記述であるということは、それだけ設計の変更が容易になるということで、完成度の高いシステムを構築することが可能である。

ハードウェア記述言語には、VHDL（VHSIC　HDL），Verilog-HDL，UDL／I（Unified Design Language for Integrated Circuit），SFL（Structured Function description Language）などがある。それぞれの特徴を表 4.1 に示す。記述言語にはそれぞれ一長一短があるが、そのなかでもっとも広く普及しているのが VHDL である。[7]

＜表 4.1＞　各種 HDL の比較

言 語 名	特　　徴
VHDL	幅広い分野の記述が可能、高い記述能力
Verilog-HDL	幅広い分野の記述が可能だがVHDLほど記述能力は高くない
SFL	RTLでの記述のみ可能。完全同期式の回路に限定している。単純でわかりやすい記述
UD/I	RTLでの記述のみ可能。同期式の回路記述は単純化されている

VHDL の歴史

VHDL は、米国国防省の HSIC（Very High Speed Integrated Circuit）委員会で 1981 年に提唱された。大規模 IC の開発には、より上位レベルでの検証が求められていた。またその当時、国防省向け ASIC の開発は長いのもで 3 年から 4 年もかかっていた。その間、半導体のプロセスは進歩し、開発当初の時点では一番スピードが速い ASIC を使用していたのが、開発が完了する時点では時代遅れになってしまうという問題が生じていた。そこで直接ロジック･ゲートを回路図で入力するのではなく、ハードウェア記述言語（HDL）で設計することによって、開発終了時に一番スピードの速い ASIC を選択できるようにする必要があった。

こうして、1983 年に VHDL の仕様作成が開始され、1985 年に（VHDL の仕様作成）作業が完了した。1986 年にはマニュアルにまとめられ、バージョン 7.2 として公開された。現在では、米国国防省が調達するすべての ASIC は、VHDL 記述付きで納入するように義務づけられた。その後、1986 年には IEEE（米国電気電子技術者協会）での標準化作業が、VASG（VHDL Analysis & Standardization Group）委員会で始まった。1987 年 5 月には LRM（言語仕様書　Language Reference Manual）が作成された。12 月に IEEE Std 1076-1987 として承認されている。

IEEE は、米国での技術者の集まりという位置づけであるが、ここで承認されたものが世界の標準として認められる権威のある団体である。VHDL も全世界の標準 HDL として幅広く普及している。

1989 年には、VHDL シミュレータや VHDL 記述からロジック回路を生成するソフトウェア（論理合成ツール）が EDA ベンダから販売されるようになり、実際にロジック回路設計に用いられるようになった。[7]

4-2　VHDL の記述方法

VHDL にはさまざまな記述方法ができるようになっており、一般的には下記 3 種類の表現方法がある。

○RTL 記述

直接論理合成可能な、クロックをベースにしたレジスタや、組み合わせ回路に使用される。

○Behaviar 記述

機能レベルでの振る舞いや動作の内容だけを記述するレベルで、抽象的な機能の検証に使用する。しかし、直接論理合成が出来ないので、ASIC や FPGA などの設計には使用しない。

○ゲートレベル記述

ネットリストと呼ばれる。ASIC 内部のゲートやセルベースの接続関係で表現したリストで記述をするレベルである。

4-3　ＶＨＤＬによる電子サイコロの設計例

4-3-1　リプル・カウンタの記述

サイコロの目が次に何が出るかわかっていたらサイコロの意味がないのでそのサイコロの重要な動作の一つとしてサイコロを振るという動作が必要になる。その動作を実現するのがリプル･カウンタである。なぜ、リプル･カウンタが適用であるかというと、同期式カウンタでは次に何がでるかということは決まっているので、サイコロの振るという動作のプログラムとして使用することは不適当である。

＜リスト 4.1＞にリプル・カウンタの記述、＜リスト 4.2＞にテストベンチ記述を示す。テストベンチを実際にシミュレーションした結果を図 4.1 に示す。

リプル･カウンタの記述

リプル･カウンタは回路構造は単純であるが、RTL 記述ではきれいに記述方法がなく、フリップフロップを単純に並べるしかない。＜リスト 4-1＞にリプル・カウンタの記述を示す。

6 進の非同期カウンタのＶＨＤＬによる記述

＜リスト 4.1＞リプル・カウンタの記述

```
library ieee;
use ieee.std_logic_1164.all;
entity DFFR is
    port(
        CLK,RESET1,D : in std_logic;
        Q,QN :out std_logic
       );
end DFFR;

architecture RTL of DFFR is

signal Q_IN:std_logic;
begin
   QN <= not Q_IN;
   Q <= Q_IN;
 process(CLK,RESET1)begin
   if(RESET1='1')then
     Q_IN <= '0';
   elsif(CLK'event and CLK = '1')then
     Q_IN <= D;
   end if;
 end process;
end RTL;


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
```

```
entity COUNTER is

  port(
      CLK,RESET : in std_logic;
      COUNT : out std_logic_vector(2 downto 0)
      );

end COUNTER;
architecture RTL of COUNTER is

component DFFR

   port(CLK,RESET1,D : in std_logic;
       Q,QN : out std_logic);

end component;

 signal COUNT_IN_BAR : std_logic_vector(3 downto 0);
 signal COUNT_IN : std_logic_vector(2 downto 0);
 signal RESET_IN : std_logic;

begin

 COUNT <= COUNT_IN;

 COUNT_IN_BAR(0) <= CLK;

 GEN1:for I in 0 to 2 generate
   U:DFFR port map(
           CLK => COUNT_IN_BAR(I),
           RESET1 => RESET_IN,
           D => COUNT_IN_BAR(I+1),
           Q => COUNT_IN(I),
           QN => COUNT_IN_BAR(I+1)
            );
 end generate;
```

```
   RESET_IN <= RESET or (COUNT_IN(1) and COUNT_IN(2));

end RTL;
```

＜リスト 4.2＞リプル・カウンタのテストベンチ

```
library ieee;
use ieee.std_logic_1164.all;
use work.std.textio.all;
use work.DFFR;

entity TESTBNCH is
end TESTBNCH;

architecture stimulus of TESTBNCH is
component COUNTER is
  port (

       CLK : in std_logic;
       RESET : in std_logic;
       COUNT : out std_logic_vector(2 downto 0)
);

end component;

signal CLK : std_logic;
signal RESET : std_logic;
signal COUNT : std_logic_vector(2 downto 0);

begin
   DUT : COUNTER port map (

         CLK,
         RESET,
         COUNT
);
```

```
CLOCK1:process

  begin

    CLK <= '1'; wait for 10 ns;
    CLK <= '0'; wait for 10 ns;

end process CLOCK1;

STIMULUS1 : process

   begin

     RESET <= '0'; wait for 5 ns;
     RESET <= '1'; wait for 10 ns;
     RESET <= '0'; wait for 180 ns;
     RESET <= '1'; wait for 20 ns;
     RESET <= '0'; wait;

end process STIMULUS1;

end stimulus;
```

シミュレーションするにあたり、出力される波形を見るのが、目的である。そのため、カウンタが入力 CLK を入力することで記述通りの出力波形 COUNT が出るのかを確かめる。

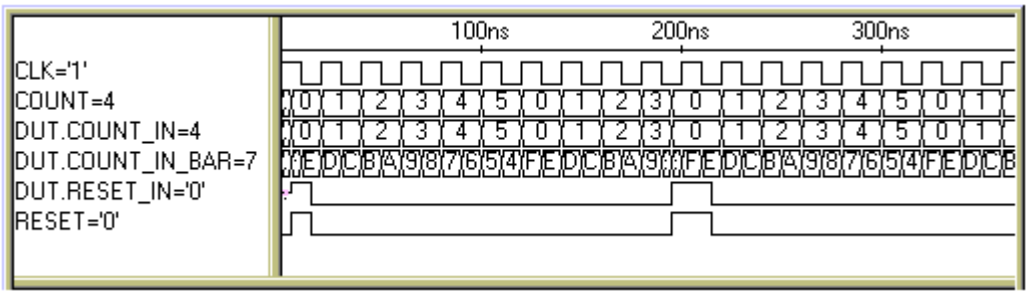


図 4.1　リプル･カウンタのシミュレーション結果

4-3-2　デコーダの記述

サイコロの目は 1 から 6 までの数字が必要となるしかし、サイコロの目を実際のサイコロの目の形に近づけるために LED を使用する。その時、LED の個数は 7 個必要になる。よって、出力は 7 出力が必要となるのでカウンタからの 3 入力を 7 出力に変換するような記述を考える。

次に、デコーダの記述説明をする。デコーダはカウンタからの 3 ビットの出力 COUNT をデコーダの入力（A，B，C）とする。そして、サイコロの目を点灯するためにサイコロの目の数だけ出力が必要であるから、入力 3 ビットを 7 ビットに変換するよにしたものである。

また、デコーダ部の（START，EN）がサイコロ振る動作とサイコロの目を決定させるための入力部となる。（START，EN）が（1,0）の時にデコーダから出力される状態である。また、（START，EN）が（1,1）の時がデコーダの出力が停止する。その時、デコーダ部の出力は保持した状態になる。

＜リスト 4.3＞にデコーダの記述、＜リスト 4.4＞にデコーダのテストベンチ記述を示す。また、テストベンチを実際にシミュレーションした結果を図 4.2 に示す。

デコーダの記述

3 入力 7 出力のデコーダのＶＨＤＬによる記述

＜リスト 4.3＞3 入力 7 出力デコーダの記述

```
library ieee;
use ieee.std_logic_1164.all;

entity DECODER3TO7 is
    port (
        A,B,C : in std_logic;
        START,EN : in std_logic;
        Y : out std_logic_vector(6 downto 0)
    );
end DECODER3TO7;

architecture RTL of DECODER3TO7 is
```

```
signal INDATA : std_logic_vector(2 downto 0);

begin

INDATA <= C & B & A;

 process(INDATA,START,EN)begin

  if(START = '1' and EN = '0')then --EN = STOP

     case INDATA is

         when "000" => Y <= "0001000"; --1
         when "001" => Y <= "0010100"; --2
         when "010" => Y <= "1001001"; --3
         when "011" => Y <= "1010101"; --4
         when "100" => Y <= "1011101"; --5
         when "101" => Y <= "1110111"; --6
         when others => Y <= "XXXXXXX"; --X

     end case;

  else
    null;
    --Y <= "1111111";
  end if;

 end process;

end RTL;
```

3 入力をして 7 出力されているか確認のためのテストベンチ記述

＜リスト 4.4＞3 入力 7 出力デコーダのテストベンチ記述

```
library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;
use work.DECODER3TO7;

entity TESTBNCH is
end TESTBNCH;

architecture stimulus of TESTBNCH is
component DECODER3TO7 is
    port (
        A,B,C : in std_logic;
        START,EN : in std_logic;
        Y: out std_logic_vector(6 downto 0)
    );

end component;
constant PERIOD: time := 100 ns;

signal A,B,C : std_logic;
signal START,EN : std_logic;
signal Y: std_logic_vector(6 downto 0);
signal done: boolean := false;

begin
    DUT: DECODER3TO7 port map (
           A,B,C,
           START,EN,
           Y
         );

 STIMULUS1 : process

  begin
```

```
   START <= '0';wait for 50 ns;
   START <= '1'; wait;

end process STIMULUS1;

STIMULUS2 : process

 begin

   EN <= '0'; wait for 245 ns;
   EN <= '1'; wait for 55 ns;
   EN <= '0'; wait for 333 ns;
   EN <= '1'; wait for 55 ns;
   EN <= '0'; wait for 300 ns;
   EN <= '1'; wait for 55 ns;

end process STIMULUS2;

STIMULUS3 : process

 begin

   A <= '0'; wait for 25 ns;
   A <= '1'; wait for 25 ns;

end process STIMULUS3;

STIMULUS4 : process

 begin

   B <= '0'; wait for 50 ns;
   B <= '1'; wait for 50 ns;
   B <= '0'; wait for 50 ns;
```

```
 end process STIMULUS4;

 STIMULUS5 : process

  begin

    C <= '0'; wait for 100 ns;
    C <= '1'; wait for 50 ns;

 end process STIMULUS5;

end stimulus;
```

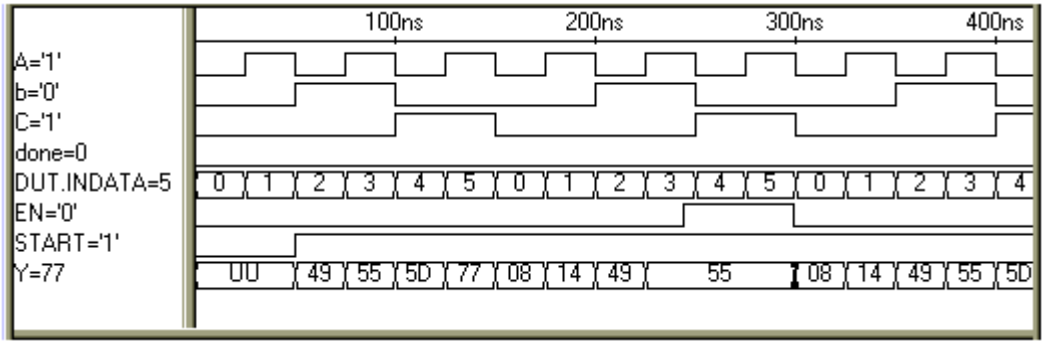


図 4.2　デコーダのシミュレーション結果

4-3-3　サイコロの記述

＜リスト 4.1＞と＜リスト 4.3＞で記述したプログラムを次に一つにする。その記述を＜リスト 4.5＞に示す。また、そのテストベンチを＜リスト 4.6＞に示す。テストベンチを実際にシミュレーションした結果を図 4.3 に示す。

サイコロの記述をする前に大まかなサイコロの動作の説明をする。

まず、入力として CLK が入力されカウンタが動作し、出力 COUNT（3 ビット）を出力する。そして、デコーダ部の（START，EN）が（1,0）と入力された時にカウンタからの出力 COUNT がデコーダの入力（A，B，C）に入力されるその値に応じて出力 Y（7 ビット）が出力されるというような記述をしている。この状態が、サイコロを振っている状態である。

また、デコーダ部の（START，EN）が（1,1）の時、デコーダの出力は現在の状態を保持した状態になる。この状態が、サイコロを振ってサイコロの目が決定した状態と同じことになる。そしてまた、（START，EN）が（1,0）に戻ればまた出力が変化する。

サイコロの記述方法の概念として大きな箱（DICE）を作る。そこに＜リスト 4.1＞、＜リスト 4.3＞で記述したものをコンポーネントを使用し呼び出すという記述をした。その記述を下記に示す。

VHDL によるサイコロの記述

＜リスト 4.5＞サイコロの記述

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity DICE is

      port(
          CLK,RESET : in std_logic;
          Y : out std_logic_vector(6 downto 0);
          --COUNT : out std_logic_vector(2 downto 0);
          START,EN : in std_logic
          );

end DICE;

architecture RTL of DICE is

component COUNTER

    port (
         CLK,RESET : in std_logic;
         COUNT : out std_logic_vector(2 downto 0)
         );

end component;
```

```
component DECODER3to7
    port (

          A,B,C : in std_logic;
          START,EN : in std_logic;
          Y : out std_logic_vector(6 downto 0)

          );
end component;

signal U0_COUNT : std_logic_vector(2 downto 0);

begin


   U0 : COUNTER port map ( CLK,RESET,U0_COUNT);
   U1 : DECODER3to7 port map (
              U0_COUNT(0),U0_COUNT(1),U0_COUNT(2),
              START,EN,
              Y
               );
end RTL;
```

＜リスト 4.6＞サイコロのテストベンチ記述

```
library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;
use work.DICE;

entity TESTBNCH is
end TESTBNCH;

architecture stimulus of TESTBNCH is
component DICE is
```

```
   port (
        CLK,RESET : in std_logic;
        Y : out std_logic_vector(6 downto 0);
        --COUNT : out std_logic_vector(2 downto 0);
        START,EN : in std_logic
        );

end component;
constant PERIOD: time := 100 ns;

signal CLK,RESET : std_logic;
signal Y : std_logic_vector(6 downto 0);
--signal COUNT : std_logic_vector(2 downto 0);
signal START,EN : std_logic;
signal done : boolean := false;

begin
   DUT: DICE port map (
                    CLK,RESET,
                    Y,--COUNT,
                    START,EN
                    );

 CLOCK1: process

  begin

    CLK <= '1'; wait for 10 ns;
    CLK <= '0'; wait for 10 ns;

 end process CLOCK1;

 STIMULUS1 : process

   begin
```

```
     START <= '0'; wait for 20 ns;
     START <= '1'; wait;

 end process STIMULUS1;

 STIMULUS2 : process

    begin

     EN <= '0'; wait for 145 ns;
     EN <= '1'; wait for 35 ns;
     EN <= '0'; wait for 233 ns;
     EN <= '1'; wait for 35 ns;
     EN <= '0'; wait for 70 ns;
     EN <= '1'; wait for 35 ns;
     EN <= '0'; wait for 125 ns;
     EN <= '1'; wait for 35 ns;
     EN <= '0'; wait;

end process STIMULUS2;

 STIMULUS3 : process

    begin

     RESET <= '0'; wait for 5 ns;
     RESET <= '1'; wait for 10 ns;
     RESET <= '0'; wait for 245 ns;
     RESET <= '1'; wait for 10 ns;
     RESET <= '0'; wait;

 end process STIMULUS3;

end stimulus;
```

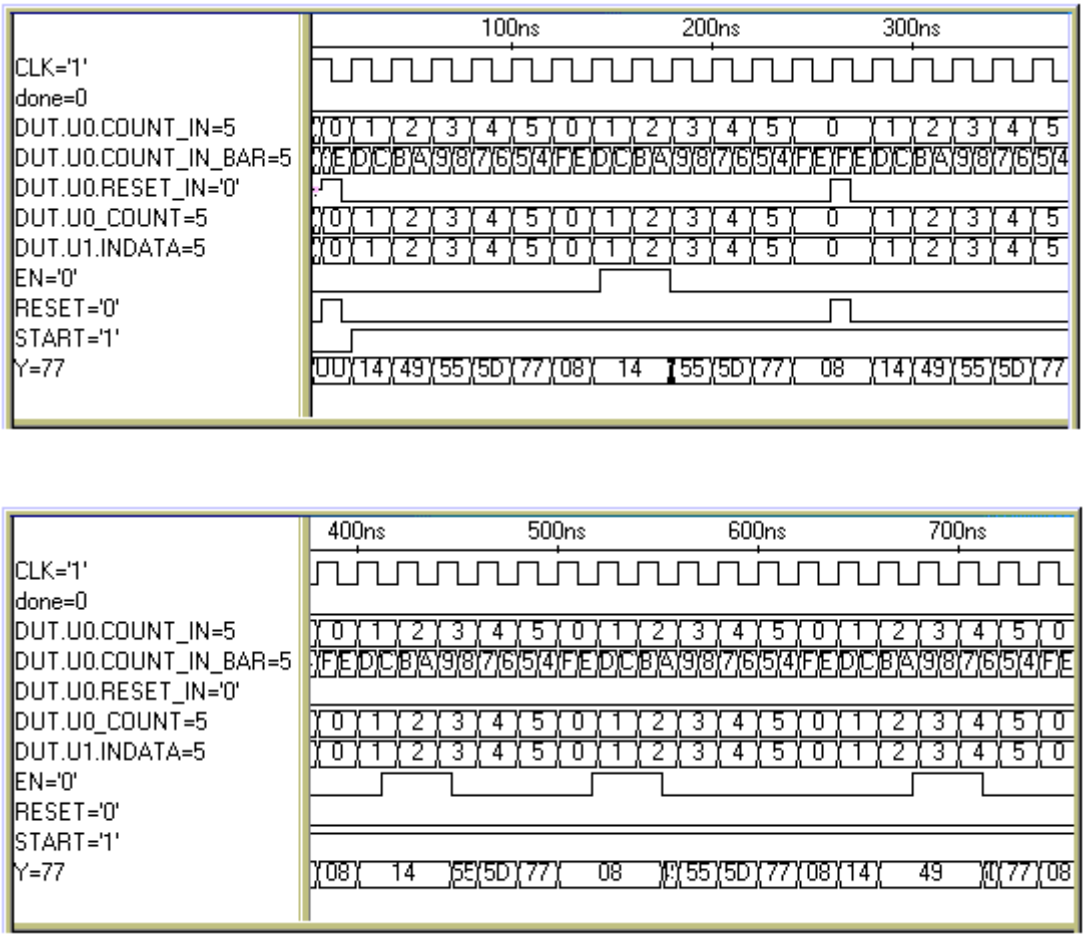


図 4.3　サイコロのシミュレーション結果

第 5 章　まとめ

今回の卒業研究において、標準論理 IC を用いた回路設計、VHDL 記述を用いた設計方法について学習してきた。実験としては、標準論理 IC を用いての実装と VHDL でリプル･カウンタと 3-7 デコーダの VHDL 記述とテストベンチを記述し、そのカウンタとデコーダをコンポーネントで関連付けしたサイコロを作成し、シミュレーションをした。最終的な目標であったサイコロを VHDL で記述し動作をシミュレーションで確認し、記述通りにシミュレーションの波形がでているかを確認するところまでが目標としていた。実験の成果としては、当初の目標である VHDL を用いてサイコロを記述するということを達成することができた。実験の成果としては、ディジタル回路の設計方法、VHDL の記述方法についての理解を深めることができたと思う。

今回の卒業研究においての課題としては、一つのことを短期間ではあるが目標を立て、その最終目標の実現のために努力をするということでした。この経験は社会にでてもいずれかの形で役に立つのではないかと考えている。

謝辞

本研究に際して、貴重な助言、ご指導をいただきました高知工科大学 電子・光システム工学科 矢野政顕 教授に深く感謝いたします。研究中、懇切丁寧なご指導を賜りました高知工科大学 電子･光システム工学科 学科長 原央 教授、高知工学科 電子･光システム工学科 橘昌良 助教授に深く感謝申し上げます。また、本研究に関して貴重なご意見をいただきました、大学院生、同じ研究室のグループの皆さん、色々な面で貴重な助言や励ましを下さった研究室の皆様に深く感謝いたします。

参考文献

[1] 論理回路入門　浜辺隆二　森北出版株式会社

[2] コンピュータアーキテクチャの基礎　柴山潔　近代科学社

[3] コンピュータ理解のための論理回路入門　村上国男　石川勉／共著　共立出版株式会社

[4] 論理設計スイッチング回路理論　笹尾勉　近代科学社

[5] －集積回路時代の－ディジタル電子回路　藤井信生　株式会社　昭晃堂

[6] ディジタル回路の設計･製作　湯山俊夫　CQ 出版

[7] VHDL によるハードウェア設計入門　長谷川裕恭　CQ 出版社

付録

名付け規則とコメント文

名前の付け方

VHDL での記述では大文字と小文字の区別がなく、大文字で書いても小文字で書いても、または混在しても同じ語句として認識される。しかし、ただ一つだけ例外がある。文字定数と呼ばれる‘　’（シングル・クォート）あるいは“　”（ダブル・クォート）で囲まれた文字だけは小文字と大文字の区別をする。[7]

＜リスト 1＞　名前の付け方の例

```
signal A : std_logic;
signal B : std_logic_vector(3 downto 0);
A <= 'X'  -- 小文字'x'ではエラー
B <= "XXXX" -- 小文字"xxxx"ではエラー
```

コメント文

VHDL では“－－”から行末までコメント文になる。コメント文は後から見てわかるようにするという目的で記述したりする。コメント文は VHDL では処理されない。

VHDL 記述の方法

signal 宣言

ポートとして宣言できない内部信号は、宣言文としてアーキテクチャ内で宣言します。

例：　signal　BBB　：　std_logic_vector（4　downto　0）；

ポートで出力指定（out）し、尚且つ内部でも再利用する必要のある信号は、一旦この信号名は区別する必要がある。

内部信号で機成し out ポートに代入する方法をとります。この場合、ポート名と内部信号名は区別する必要がある。[7]

ジェネレート文

同じコンポーネントの繰り返しを表現する場合にはジェネレート文を使用する。ジェネレート文には for-generate 文と if-generate 文の 2 種類がある。

ラベル：for　ジェネレート変数　in　不連続範囲　generate
＜同時処理文＞
end　generate　[ラベル]；

ラベル：if　条件　generate
＜同時処理＞
end　generate　[ラベル]；

for-generate 文は for-loop 文と異なり、アーキテクチャ文の中に記憶する同時処理文になる。したがって、内部の記述も順番に処理されるのではなく、並列に処理されることになる。また、exit 文や next 文を使用することはできない。

If-generate 文は、条件が“ TRUE ”のときだけ内部の記述を生成する。If 文と異なり、else 項はない。[7]

wait 文

単純に“ wait； ”と記述すると無限実行停止という意味になり、このプロセス文はそこから永遠に動作しなくなる。

Wait 文には、この他に“ wait　until ”や“ wait　on ”とプロセス文のセンシティビティ・リストの違いは“ wait　on ”がプロセス文中に何度も使用できるのに対し、センシティビティ・リストは文頭に一度しか使用できない点である。

プロセス文にセンシティビティ・リストがある場合は、プロセス文内で wait 文を使用することはできない。wait 文の種類を＜表 1＞に示す。[7]

＜表 1＞　wait 文の種類

wait on　信号, 信号…
wait nutil 条件
wait for 時間
wait

case 文の記述

case 文は、

when　値　=＞　文　　注：　“ =＞ ”は関係演算子ではない

という形式で記述する。

If 文の場合は、最初の条件が処理されたと、次の条件が処理されますが、case 文の場合は値の順番がなく、すべてが並列に処理される。したがって、1 度“ when ”項に記述した値を、そのあとで再び使用すると文法エラーになる。値が重複しないように注意をする。また、すべての場合を記述しないと文法エラーになる。よって、使わない値は“ others ”と記述する。“ others ”という記述は残りすべての場合という意味になる。

＜表 2＞に case 文の書式を示す。[7]

＜表 2＞　case 文の書式

```
case 式 is
      条件式
end case；
条件式： when 値 => 順次処理文
         when 値 | 値 | 値… => 順次処理文
         when 値 to 値 => 順次処理文
         when others => 順次処理文
 注：when othersは最後の行に1回のみ使用可能
     値は重なりがあってはならない。また、式のとりうるすべて
     の値を記述しなければならない
```

論理演算子

“ and ”, “ or ”などで記述されたものは論理式レベルの記述とばれている。論理式には、

not　and　or　nand　nor　xor

の 6 種類がある。

論理演算子は、“ std_logic ”, “ bit ” などのロジック型のデータ・タイプ、“ std_logic_vector ” などのロジック型の配列タイプ、および “ Boolean ” タイプが使用可能です。論理演算を使用する場合は、式の右辺と左辺および代入される信号は、すべて同じデータ・タイプである必要がある。

一つの文に二つ以上の論理式がある場合，C 言語などは左から順に優先順位が高くなりますが、VHDL では右左で優先順位の差はなく、＜リスト 2＞の①のように括弧でくくらないと文法エラーになる。ただし、例外があり、“ and ”, “ or ”, “ xor ” だけで構成されるものは順序を変更しても論理が変わらないので括弧を省略することができる。[7]

例：A ＜= B and C and D and E；
A ＜= B or C or D or E；
A ＜= B xor C xor D xor E；

＜リスト 2＞ AND-OR セレクタの記述

```
library IEEE;
use IEEE . std_logic_1164. all;

entity AND_OR_SELECTOR is

   port (A,B : in std_logic;
         SEL : in std_logic;
         Y   : out std_logic);

end AND_OR_SELCTOR;

architecture DATAFLOW of AND_OR_SELCTOR is

begin

   Y <= (A and SEL) or (not SEL and B);

end DATFLOW;
```

①論理式の記述

例：A ＜= （(B nand C）nand D）nand E； --必ず括弧が必要

A ＜= （B and C）or （D and E)； --必ず括弧が必要

ただし、論理演算子の中で“ not ”だけは優先順位が他の演算子（算術演算子、関係演算子）よりも高くなっている。①では “ not ” は “ and ” よりも先に演算される。

＜リスト 2＞は、AND-OR セレクタとよばれる回路で、ロジック回路設計ではよく使用される回路である。“ SEL ”に‘ 1 ’が入力された場合は信号‘ A ’を出力し、“ SEL ”に‘ 0 ’が入力された場合は‘ B ’を出力する。[7]

コンポーネント宣言

VHDL では、コンポーネント・インスタンス文で下位のモジュールを呼び出す前に、“ architecture ”と“ begin ”の間にそのモジュールがどのような形で構成されているかを表すコンポーネント宣言をしなければならない。コンポーネント宣言の記述方法を下記に示す。

```
component　コンポーネント名
        [ジェネリック文]
        [ポート文]
end　component；
```

コンポーネント宣言は、“ architecture ”と“ begin ”の間以外に、パッケージ文で宣言することもできる。[7]

コンポーネント・インスタンス文

コンポーネント・インスタンス文は

ラベル名　：　コンポーネント名　port　map　（信号, …）；

と記述する。ラベル名は、このコンポーネントに付けられる名前で、そのアーキテクチャ宣言の中で唯一のものでなければならない。

下位階層のポートと信号は、“ port　map ”で結合する。結合のさせかたは、「位置による関連付け」と「名前による関連付け」の 2 種類がある。

名前による関連付けは、

ポート名　＝＞　信号名

と記述する。[7]