卒 業 研 究 報 告

題　　目

ハードウェア記述言語によるディジタル回路設計

指 導 教 員

矢野　政顕　　教授

報 告 者

小松　輝将

平成 13 年　 3 月　 1 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

われわれの身の回りにある電気・電子機器には LSI が多数搭載されている。たとえば、パソコンやゲーム機には CPU やメモリ、さらに周辺機器制御用 LSI などのさまざまな LSI が搭載されている。近頃ではテレビ、ビデオ、ステレオなどの AV 機器もディジタル化が進み、音声や画像を処理する LSI が組み込まれている。このような状況が出現したのは、LSI の製造技術の進歩により LSI に搭載できる回路の規模が増大し、高度で多様な機能を実現できるようになったからである。もっとも、回路が大規模で複雑なものになれば回路設計の困難は増大する。そこで、LSI の大規模化・複雑化に対応するため設計手法にも革命が起こった。すなわち、HDL（Hardware Description Language；ハードウェア記述言語）を用いてトップダウン的に設計する手法が開発され、広く用いられるようになってきた。

この卒業研究において順序回路とVHDL を勉強し、それを理解した上で設計例としてディジタル時計をとりあげ VHDL で設計をする。

第2章　順序回路

2.1 順序回路とは

組合せ回路は、入力の組合せだけで出力の状態が一意に決まる回路であるが、順序回路（sequencial logic circuit）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。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、従来の組合せ回路と記憶素子の組合せで構成されるものである。順序回路のブロック構成を図 2.1 に示す。

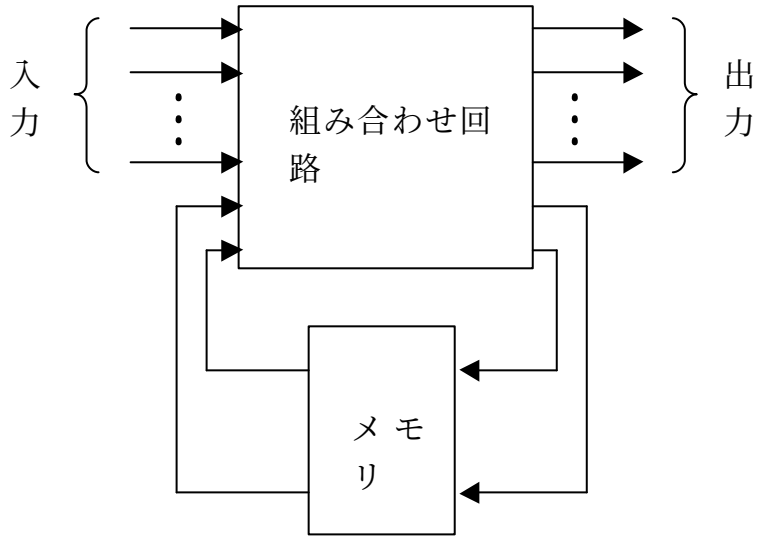


図 2.1 :順序回路のブロック構成

2.1.1 基本順序回路

（a）　メモリ

順序回路には、入力の履歴（状態、ステータス）を保存するためのメモリ（memory）が必要である。

簡単なメモリは 2 値の論理値（1ビット）を記録する機能だけ持てばよいので、1 ビットメモリはフリップフロップで（flip-flop）で実現できる。

(b)　基本順序回路

基本的な順序回路は組み合わせ回路とメモリとから構成される。
順序回路は回路動作タイミングから同期式順序回路と非同期式順序回路とに分けることができる。

(c)　クロック

順序回路における動作のタイミング（同期）をとるパルス信号がクロック（clock）である。クロックによって順序回路のメモリの状態を変更する事ができる。

(d)　同期式順序回路

回路動作がクロックに同期している順序回路を同期式順序回路という。メモリは 1 クロックパルス入力に対して 1 回だけ状態を変え得る。したがって入力が不変でも状態変化によって出力が変わり得る。動作速度がクロックパルスに依存するが、現在の状態が十分安定したと思われる時間間隔で動作させるので安定性がよく、コンピュータなどの設計に用いられる。

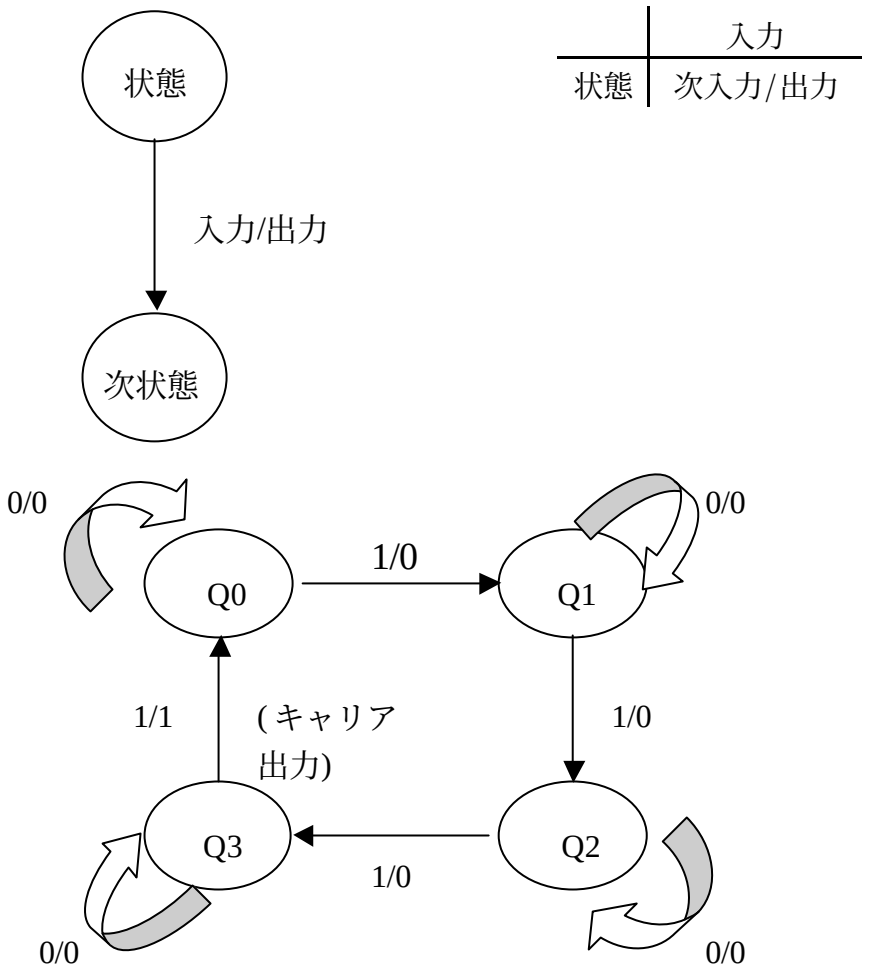


図２２:状態遷移図

・状態遷移表と状態遷移図

組合せ回路ではその設計に真理値表を用いたが、順序回路では真理値表の変わりに状態遷移表（state transition table）を用いる。順序回路は安定した現在の状態に入力信号が加わると出力が出るとともに次の状態に遷移する。この移りゆく状態を理解しやすくするために図示したものが状態遷移図（state transition diagram）という。

同期式順序回路動作の記述は状態遷移図によって表すことができる。状態遷移図においては、ノードが状態をリンク（矢印）が状態遷移を表す。また、その状態遷移をリンクに入力論理値とそれによる出力論理値を付記することによって表す。

(2)　非同期式順序回路

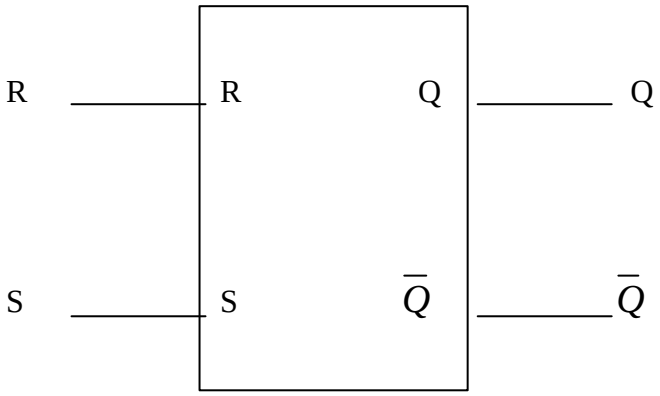
回路動作が任意の時刻に発生する入力の変化、およびその順序だけに依存する順序回路を非同期式順序回路という。最後の入力変化の影響が落ち着くことを安定という。原理的には同期式の方よりも高速だが、回路が複雑になると遅延時間の推定が困難になるので、むしろ時間に左右されない比較的遅いディジタル回路設計に使用される。

2.2 フリップフロップ（FF）

1 ビットのメモリは0か1とのいずれかの安定状態（双安定状態）をもつので、これをフリップフロップという。順序回路を構成する重要な部分であるメモリは多数のフリップフロップによって構成されている。

・SR フリップフロップ (SR-FF)

SR−FF (set reset) は 2個のNAND素子で構成される最小構成のＦＦである。実際のＦＦは、ＳＲ−ＦＦの機能と他の３つのＦＦを組み合わせたものがほとんどである。ＦＦとは図 2.2 に示す**2安定マルチバイブレータ** (multi vibrator) のことである。マルチバイブレータにはこの他に無安定 (安定のない発信器) と1安定 (安定点が 1つある単発パルス発信器) があり、いずれも実用ディジタル回路ではよく使用される。



(a) SR-FF の記号

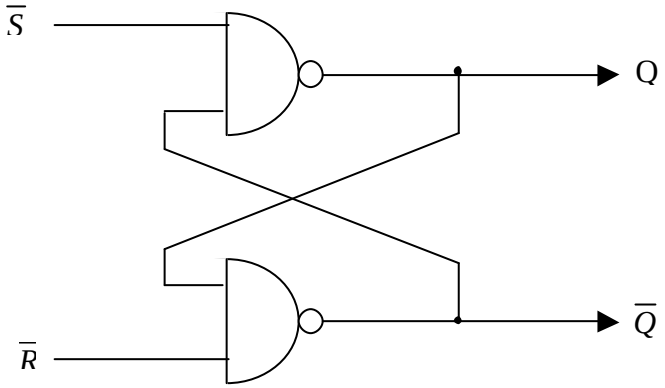
R	S	Q+
L	L	Qn
L	H	H
H	L	L
H	H	不定

(b) 真理値表

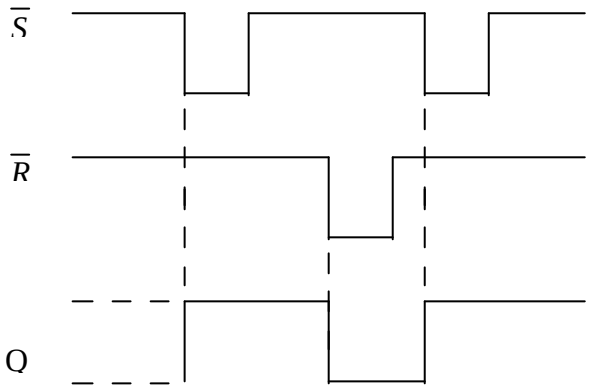
入力 S	入力 R	現在の状態 Q	次の状態 Q+	
0	0	0	0	
0	0	1	1	
0	1	0	0	リセット
0	1	1	0	
1	0	0	1	セット
1	0	1	1	
1	1	0	x	
1	1	1	x	

(c) SR-FF の遷移表

S	0	0	1	1
R	0	1	0	1
Q	Q’	0	1	x



(d) NAND 構成の SR-FF



（e）タイムチャート

図 2.3： SR フリップフロップ

・エッジトリガ型フリップフロップ

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）/立ち下がり（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。

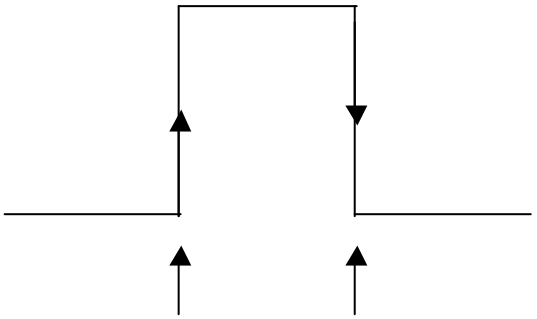
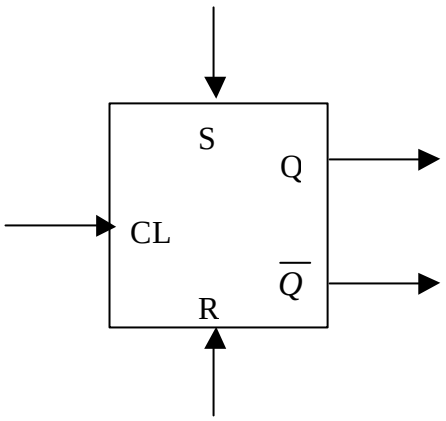


図 2.4 :エッジトリガ型フリップフロップ

⑴ クロック入力つきSRフリップフロップ



S	0	0	1	1
R	0	1	0	1
Q	Q’	0	1	x

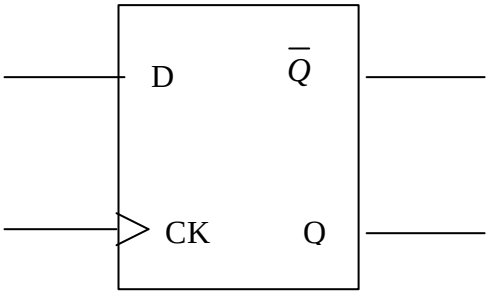
$Q = Q(t+1) \qquad Q' = Q(t)$

$Q = \overline{R}\, Q' + S \quad (S\ R = 0)$

図２５：クロック入力つきSRフリップフロップ

② Dフリップフロップ

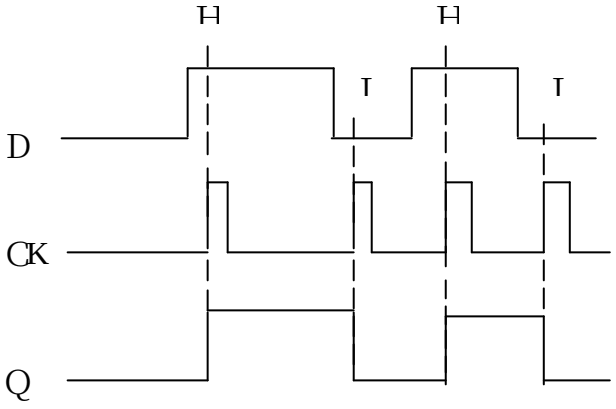
Q出力はD入力のみに依存し、状態信号の取り込み保存（ラッチ）用として利用できる。



ⓐD-FF の記号

D	Q	Q_+
0	0	0
0	1	0
1	0	1
1	1	1

ⓑD-FF の遷移表

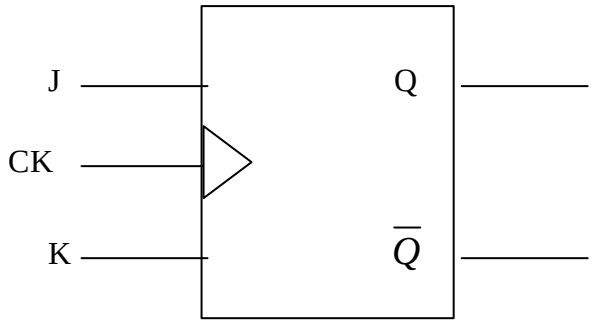


ⓒD-FF のタイミングチャート

図 2.6： D フリップフロップ

・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 入力と組み合せたフリップフロップ



ⓐJK-FF の記号

現在の状態 Q	入力の状態 Q_+ 入力 JK 00.	01	11	10
0	0	0	1	1
1	1	0	0	1

J	K	Q_+
0	0	Q
0	1	0
1	1	$\overline{Q}$
1	0	1

ⓑ状態表

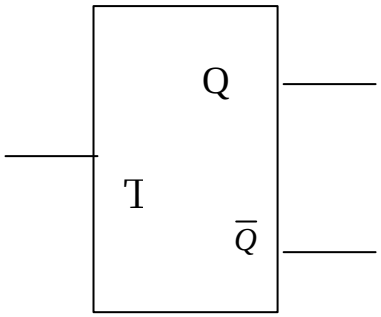
現在の状態 Q	入力後の状態 Q_+ 入力 CK 0 JK 00	01	11	10	1 JK 00	01	11	10
0	0	0	0	0	0	0	1	1
1	1	1	1	1	1	0	0	1

ⓒJK-FF の遷移表

図 2.7： JK フリップフロップ

・Tフリップフロップ

入力が加わるごとに出力の１と０が反転するトグル（toggle）動作を行う。



(a) JK-FF の記号

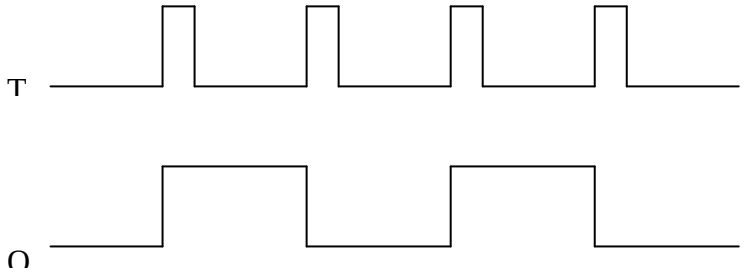
現在の状態	入力後の状態 Q_+	
Q	入力 T	
	0	1
0	0	1
1	1	0

(b) T-FF の遷移表

$$Q_+ = \overline{Q}T + Q\overline{T}$$

T	Q_+
0	Q
1	$\overline{Q}$

(c) 状態表



(d) T-FF のタイミングチャート

図 2.8： T フリップフロップ

・ マスタースレーブ型フリップフロップ

クロックの立ち上がり（マスタ）と立下り（スレーブ）の組で1つの動作（2段階動作）を行うフリップフロップ。動作タイミングが異なるだけで、種類はエッジトリガ型フリップフロップと同様のものがある。

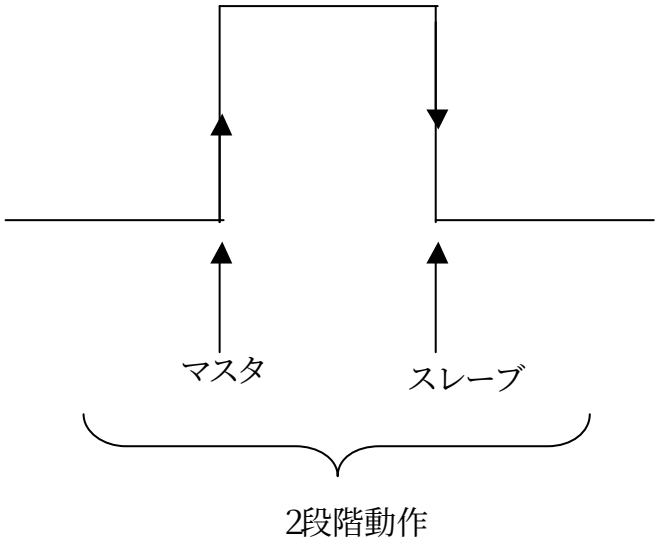


図 2.9 :マスタースレーブ型フリップフロップのクロック入力

・ ラッチとレジスタ

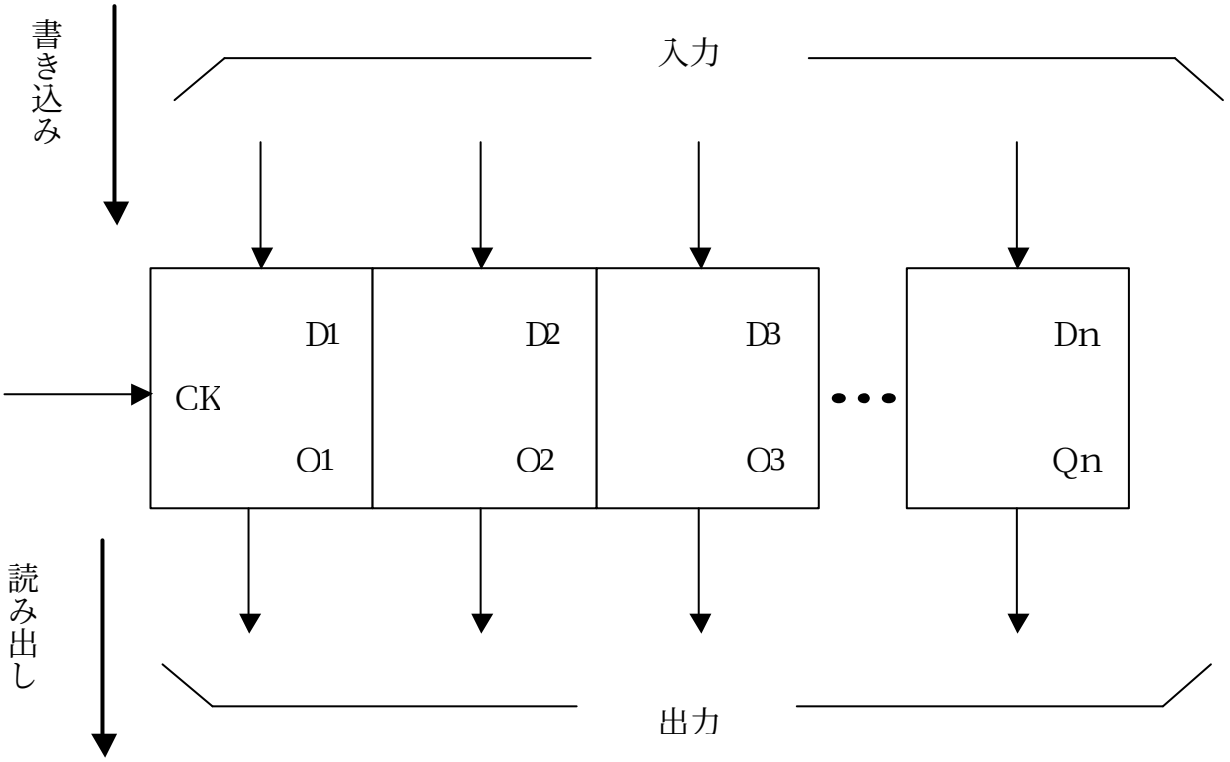


図 2.10：ラッチとレジスタ

上図に示すようなフリップフロップの集まり（複数個のフリップフロップを並べたもの）をレジスタという。レジスタ長（ビット）とは、並べたフリップフロップの個数のことであり、各フリップフロップは同期して動作する。

レジスタの動作にはフリップフロップ出力の読み出しとフリップフロップへの書き込み/状態更新とがあり、各ビットは同時（並列）に動作する。

・ シフタ

あるフリップフロップ出力を別のフリップフロップ出力を別のフリップフロップ入力とし、それらの相互に連結したフリップフロップに共通のクロックを入力することによって同期して動作するフリップフロップ群をシフタという。

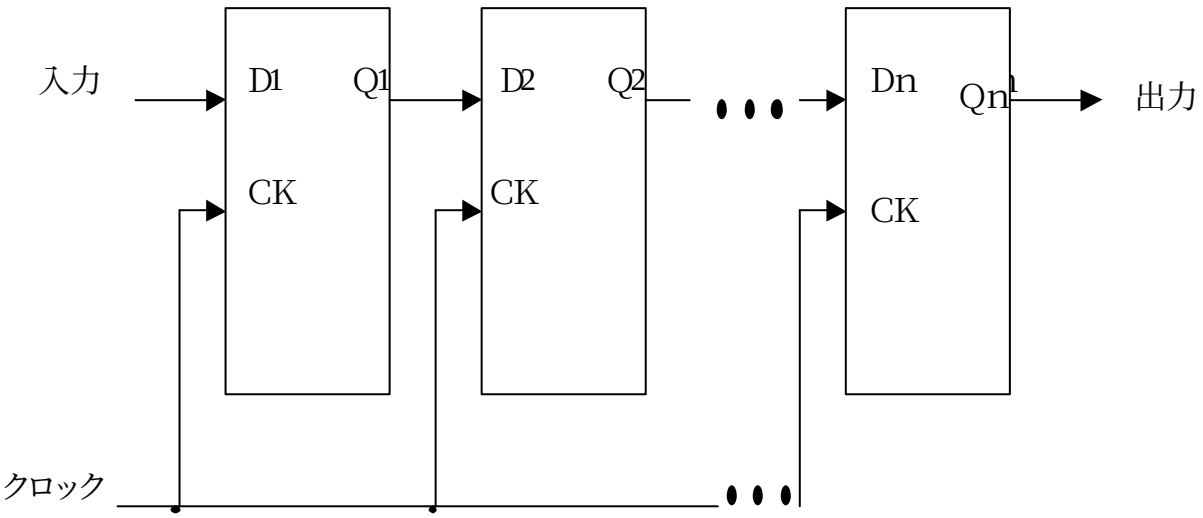


図 2.11 :シフタの原理図

・カウンタ

クロックパルスに同期してあらかじめ決められた順序で状態変化を起こす同期式順序回路のこと。カウンタはシフタの１種であるが隣接するフリップフロップ間を直結するのではなく簡単な組み合わせ回路を挿入する事によって、決められたビットパターンとしての状態変化を生起させる点が異なる。

ⓐ 2進カウンタ

T/JK フリップフロップのQ出力を次段のフリップフロップのクロック入力とするカウンタを2進カウンタという。これは、非同期式カウンタの例である。

ⓑ 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。これは、非同期式カウンタの例である。

(c) 同期式カウンタ

リプルカウンタに対して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状態変化するカウンタを同期式カウンタという。

(d) 可逆カウンタ（アップダウンカウンタ）

カウントアップのカウントダウンの両方が可能なカウンタを可逆カウンタという。

(e) リングカウンタ

シフトレジスタの最上位からの出力を最下位段への入力とし、フリップフロップをリング状に連結したカウンタをリングカウンタという。

(f) ジョンソンカウンタ

シフタの最上位段からの出力を反転して最下位段の入力とするリングカウンタをジョンソンカウンタという。

2.3 順序回路の設計

- 順序回路の応用方程式とＦＦの入力方程式

ＦＦの出力には常に Q と $\overline{Q}$ が存在するがこれらを用いた式ですべての順序回路を表すことができる。ある順序回路において現在の状態を Q, $\overline{Q}$, 次の状態を $Q+$ とするとき、これらの関係は一般的に次式で表すことができる。

$$Q_+ = g_1 Q + g_2 \overline{Q}$$

この方程式を応用方程式（application equation）という。
ＦＦの制御方法とはその入力端子への入力に関する論理関数、すなわち入力方程式を求めること

である。設計に使用するＦＦを決めればその特性方程式が一意に決まるのでその式と応用方程式から入力方程式を求めることができる。

順序回路の設計手順をまとめると以下のようになる。

(a) 状態遷移図で遷移状態を確認し,記憶すべき状態数を求める。
(b) 状態数を符号化し,状態割り当てを行う。
(c) 状態繊維表を作成する。
(d) カルノー図を描き、設計したい順序回路の特性方程式を求める。
(e) 特性方程式を応用方程式に対応させて、g1,g2 を求める。
(f) 使用するＦＦを決めて、その入力方程式に g1,g2 を代入し、その論理関数を求める。
(g) 順序回路を構成する。

以上のように,順序回路の設計とはまず設計したい回路の動作特性を特性方程式で表し、その動作特性を実現するために使用する特定のＦＦの入力方程式を求めることである。

第 3 章 ハードウェア記述言語によるディジタル回路設計

3.1 ハードウェア記述言語について

3.1.1 ハードウェア記述言語

ハードウェア記述言語（HDL:Hardware Description Language）が出現して以来ＶＬＳＩの設計は大きく変わった。従来はＲＴＬ回路の記述は仕様書によって行われており、計算機で動作が検証できるＲＴＬ表現も存在したが、ゲートレベルへの展開が可能な表現ではなかったので、人手で回路を構築していくボトムアップ的設計手法が取られていた。1980 年代後半になってVHDLやVerilogＨＤＬといったハードウェア記述言語が開発されてRTL回路はＨＤＬで記述することが一般的になり、ゲートレベルへの展開が可能になってきた。このように、ＨＤＬを用いてＲＴＬ設計を行うことをＨＤＬ設計と呼んでいる。

・VHDLの概要

ハードウェア記述言語（HDL）は、Cや Pascal などのような形式言語である。ただし、プログラミング言語とは異なり、ＨＤＬは、ハードウェアを設計する上で欠かすことのできない並列処理（同時処理）や時間、タイミングの概念などを記述できる特徴がある。

VHDL（語源はVHSIC HDL）は、ＨＤＬの一つであり、米国国防総省において、VHSIC（very high speed IC）プロジェクトの一環として、1981 年に開発がはじめられた。1983 年には,言語に対する要求仕様書がまとめられ、1987 年には言語マニュアル（language reference manual : LRM）が発行された、さらに1987 年末には、IEEE において、標準言語として承認され、現在では全世界に広く普及している。

IEEE:The Institute of Electrical and Electronics Engineers(米国電気電子技術者協会)の略称、エレクトロニクス全般に関する研究を目的とした世界的に権威のあるアメリカの学会。

表 3.1　HDLの歴史（発展の経過）

言語	VHDL	verilogHDL	UDL/Ⅰ	SFL
開発開始時期	1981	1984	1987	1981
開発組織	IEEE	Cadence	電子協	NTT
言語使用公表	1987	1985	1990	1985
論理シミュレータ	有	有	有	SECONDS
論理合成系	有	有	有	SFLEXP
企画の見直し	1933	1999（予定）	1992	予定なし

・ VHDL

VHDLは 1970 年代にスタートした米国防省のVHSIC（Very High Speed IC）プロジェクトで開発されたHDLであり、1987 年に IEEE std−1076 として標準化された。VHDLはシステムからスイッチに至るハードウェア全般の記述を対象としており、ハードウェアの構造と動作の両方を記述できる。VHDLでは回路を階層的にとらえ、ある回路は下位のより小さい回路が接続されたものとして記述する。最下層の回路を動作的に記述し、それより上位の回路を構造的に記述する。VHDLは、ユーザー定義のタイプ、抽象度の高いデータタイプ、回路をコントロールするためのコンフィギュレーションなどに特徴がある。

・ VHDLの生い立ち

VHDLは 1980 年代アメリカ国防省のVHSIC（超高速集積回路）計画から生まれました。ここで技術者たちは大規模な回路設計しようとしたが、従来の回路図を使った方法ではもはや不可能であるとの結論に達し、新たな設計手法が求められた。そのとき国防省の付託により IBM、テキサスインスルメンツ、インターメトリックの 3 社によって開発されたのがVHDLによる言語設計の手法である。1985年にその情報が公開され、以後 1987 年に IEEE1076として標準化され、世の中で初めて標準化されたハードウェア記述言語となり、その後1993年に機能拡張が行われ、現在に至っている。

3.1.2　シミュレーションのためのモデリング

ＶＨＤＬは電子回路の動作を記述するためのさまざまな機能を持っており、AND, ORといったゲートからマイクロプロセッサのような複雑な動作をするデバイスまで容易に記述できる。しかも複雑な動作を数式のような抽象的な形で記述するだけで実現できるため、複雑なロジック回路を書くことなく動作を表現できる。

また、VHDLはCやＢasicと同じくコンピューター上で動作を検証できるので回路の動作検証（シミュレーション）が容易にできるようになる。

3.1.3　設計図としてのVHDL

ＶＨＤＬからは論理合成ソフトにより、回路図をコンピューターに作らせることができる。ありがたいことにこれらの論理合成ソフトはたいていの場合、遅延時間やファンアウトを自動調整してくれるので、設計者の手間を大幅に省いてくれる。ＰＬＤやＦＰＧＡではＡＢＥＬ等の言語がすでに存在しているが、ＶＨＤＬはこれらの言語に比較してはるかに抽象性が高く、複雑で大規模な回路の記述に適している。また、ＶＨＤＬの記述はメーカーに依存していないので移植性が高い言語である。

3.1.4　**ＶＨＤＬでテストパターンを作る**

たいていの回路シミュレータの場合、回路をテストするためのテストパターンは別のツールで作ることになる。しかしVHDLシミュレータではテストパターンもVHDLで記述できる。そのため回路のデバッグとテストパターンのデバッグが同時にできる。

3.1.5　標準言語としてのVHDL

ＶＨＤＬはすでに標準的言語としてＩＥＥＥで制定され、多くのＥＤＡベンダーから支持されています。ＶＨＤＬで記述しておけば、将来使っている開発ソフトが変わったとしても、あるいはデバイスが変わったとしてもVHDLは生き続け、設計資産は残る。

3.1.6　HDLを用いたトップダウン設計の特徴とメリット

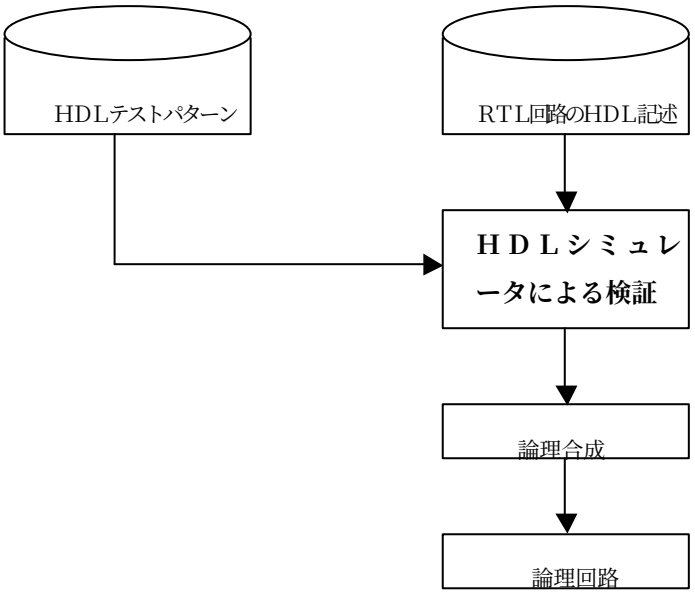


図 3.1　VHDL による設計フロー

(1)ゲートレベル設計の自動化

HDLを用いてRTL記述を行えば、論理合成が自動化される。その結果、RTL設計とゲートレベル設計を完全に分離できることから、設計者はRTL設計に専念できるようになり設計効率が向上する。しかもゲートレベルへの展開が自動化されたことは人手によるケアレスミスの発生を減らし、設計品質の向上をもたらす。なお、RTLはゲートレベルにより上位の階層なので構成部品は大きいが抽象度は高い。したがって、設計者は細部に精力を注ぐことなくより規模の大きい設計に対応でき、大規模回路を短期間で設計できる。

⑵シミュレーション早期開始が可能に

ボトムアップ的なゲートレベル設計では、シミュレーションはゲートレベルに展開してから行っていた。そのため、シミュレーションでバグが見つかっても、それが上位レベルのバグであると、バグ発生の個所の特定と修正に時間がかかってしまう。ところがHDLでRTLレベルの段階シミュレーションを行えるので、設計の早い段階で高い品質を確保できる。

⑶制御回路設計の簡単化

人手によるゲートレベル設計では、制御回路の設計は困難であり誤りが発生しやすかった。HDLによるRTL設計では、制御回路をステートマシンで構成しHDLで記述するので制御回路の設計が容易になり、誤り発生の確率は減少する。

以上の如く、HDL を用いる設計のメリットが非常に大きいということがわかる。

3.2 VHDLによるディジタル回路の記述

3.2.1 VHDLの記述例

簡単な記述例として,半加算回路の記述例を示す。これをもとに VHDL の記述について説明する。

リスト3.1 :半加算器の VHDL 記述

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

entity HAFE_ADDER is
        port (A ,B : in std_logic ;
              S , C : out std_logic );
end HALF_ADDER

architecture STRUCTURE of HALF_ADDER
begin
        S <= A xor B ;
      C <= A and B ;
end STRUCTURE ;
```

ディジタル回路は、それに入力されたなんらかの信号（データ）に加工を施し、その加工結果を出力する電子回路である。このようなディジタル回路を設計するためには、

(1) どのような信号が入力されるのか？
(2) その信号にどのような加工を施すのか？
(3) どのような信号を出力させるのか？

上記 (1)および (3)の情報は、外部から入力される信号（入力信号）や外部へ出力する信号（出力信号）に関する情報、すなわち、インターフェースに関する情報である。これに対して上記 (2)の情報は、回路内部の動作や構造に関する情報である。VHDLではこれら外部,内部の情報をそれぞれ、エンティティ（entity）、アーキテクチャ（architecture）と呼び別々に記述する。

上のVHDLの記述例では、エンティティの記述部では,HALF_ADDERというエンティティ名（回路名）が、AとBの二つの入力ポート（入力線をもち）、SとCの二つの出力ポート（出力線）を持

っていることを宣言している。また、アーキテクチャの記述部では、HALF_ADDERに対して、STRUCTUREという名前のアーキテクチャを宣言し，アーキテクチャ本体として、半加算器の構造を記述している。

3.2.3 信号線

entity宣言内で宣言される信号は、それが入力信号線（入力ポート）なのか、出力信号線（出力ポート）なのか、あるいは入出力兼用の信号線なのかが定められている必要がある。

さらに、その信号線をどのようなタイプのデータ（信号）が通過するのかもあらかじめ宣言しておく必要がある。これらの宣言は、port文を用いて行う。

また、回路の内部でのみ用いられる内部信号線についても、データ・タイプを宣言しておく必要がある。この宣言は、architecture 宣言の宣言部（architecture ~ is begin と end の間）でsignal 文を用いて行う。

記述例では、entity 宣言内で信号線 A,B が入力ポートであり、かつ、それらのデータ・タイプがbit 型であることを表している。さらに、信号線S、C が出力ポートであり、かつ、それらのデータ・タイプが bit 型であることを表している。

3.2.4 信号の代入と演算子

architecture 本体（architecture 宣言の begin と end の間）では、ディジタル回路の動作や構造を記述する。データの流れを記述するために、信号代入文＜＝を用いる。信号代入文では、複数の信号間に論理演算などを施し、その結果を他の信号線に代入することもできる。

記述例には二つの信号代入文があり、それぞれ「信号 A とB の値の排他的論理和（xor）を信号Sに代入する」、「信号 A とB の値の論理積（and）を信号Cに代入する」を表している。

3.2.5 std_logic 型

VHDL には、論理方のデータ・タイプとして、論理地'0'または'1'しかとれない bit 型およびその配列タイプである bit_vector 型しかなかった。しかし、ディジタル回路の設計やシミュレーション、論理合成などを行う際には、不定値'X'、インピーダンス'Y'、ドント・ケア'-'などを使用できることが望ましい。

このための新しいデータ・タイプとして、std_logic 型とその配列タイプ std_logic_vector 型が、1993 年に IEEE で標準化された。現在ではディジタル回路を設計する際には、これらの型を用いるのが一般的になっている。ただし、これらのデータ・タイプを使用するためには VHDL 記述の最初の部分に、

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

の2行を加える必要がある。

3.2.6 階層設計

図に示すように、回路全体をいくつかのブロックに分割し、各ブロックごとに設計を行うような設計方法を階層設計という。このとき各ブロックは、必要に応じてさらに細かいブロックに分割され、階層設計される。階層設計は、大規模用な回路を設計する際に用いられる手法である。

VHDLでは、各ブロックのことをコンポーネントと呼んでいる。VHDLを用いて階層設計を行う場合、最上位階層のVHDL記述には、分割されたコンポーネント間の接続関係が記述される。また、各コンポーネントの記述には、さらに下位階層コンポーネント間の接続関係が記述される。

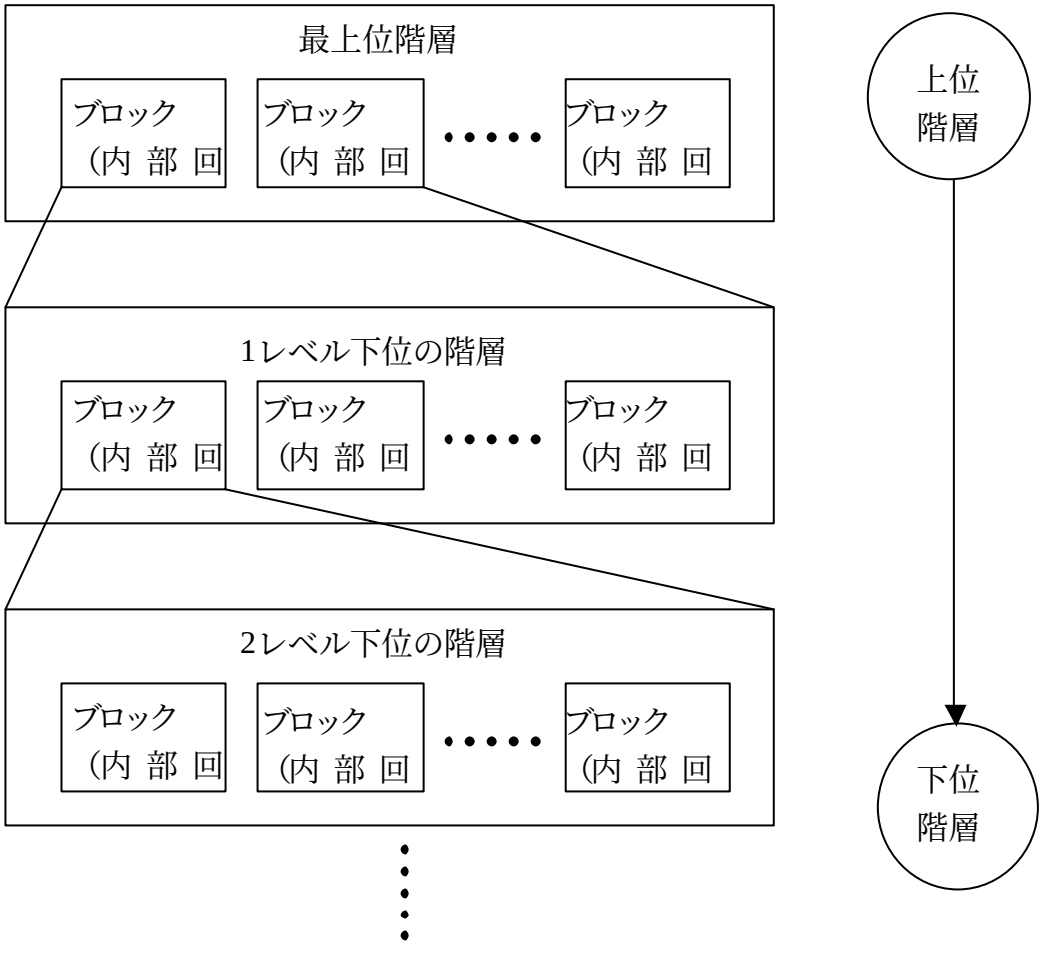


図 3.2　階層設計の概念

・ 設計例として全加算器を半加算器を二つ用いて階層設計する

リスト3.2 :半加算器のＶＨＤＬ記述

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

 entity   HAFE_ADDER   is
            port (A ,B : in std_logic ;
                   S , C : out std_logic );
 end HALF_ADDER

 architectureSTRUCTURE of HALF_ADDER
 begin
           S  <= A xor B ;
         C <=   A and B ;
 end    STRUCTURE ;
```

リスト3.3 :全加算器のＶＨＤＬ記述

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

 entity  FULL_ADDER  is
          port  (A ,B,CIN : in std_logic ;
                 SUM,  COUT  :  out  std_logic  );
 end  FULL_ADDER

architecture  STRUCTURE  of  FULL_ADDER is

component HALF_ADDER
          port  (A ,B : in std_logic ;
                 S  ,  C  :  out  std_logic  );
   end component ;

 signal  C1_C, C1_S, C2_C : std_logic;
begin
     COMP1 : HALF_ADDER port map (A, B, C1_S, C1_C);
     COMP2 : HALF_ADDER port map (C1_S, CIN, SUM, C2_C);
       COUT<= C1_C  or  C2_C
 end   STRUCTURE;
```

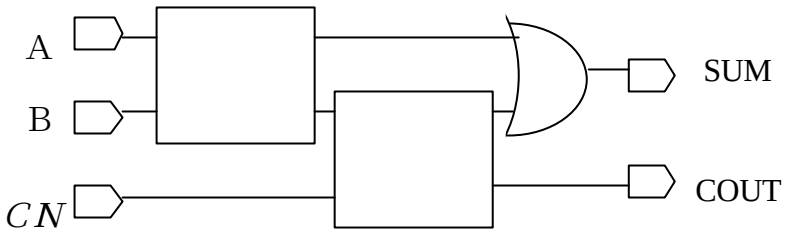


図 3.3　全加算器の論理合成結果

3.2.7 IEEE ライブラリの算術演算用パッケージ

std_logic_vector 型で算術演算や大小比較などを行う場合には、算術演算用のパッケージを使用する必要がある。算術演算用の主なパッケージを示す。

1) IEEE ライブラリ：numeric_std パッケージ
   符号ビットなし演算と符号ビット有り演算を混在させるためのパッケージ
2) IEEE ライブラリ：std_logic_unsigned パッケージ
   符号ビットなしの演算を行うためのパッケージ
3) IEEE ライブラリ：std_logic_signed パッケージ
   符号ビットありの演算を行うためのパッケージ
4) IEEE ライブラリ：std_logic_arith パッケージ
   符号ビットなし演算と符号ビットあり演算を混在させるためのパッケージ

標準パッケージ numeric_std を用いる場合、

```
A, B : in  std_logic;
C, D : in  unsigned  (7 downto 0);
E, F : out signed  (8 downto 0);
```

のように、unsigned 型、signed 型として、あらかじめ宣言しておく必要がある。なお、unsigned 型および signed 型のベースは std_logic 型なので、1 ビット幅の信号は上記のように std_logic 型として宣言をする。

また、パッケージstd_logic_unsigned、std_logic_signed を用いる場合、std_logic_vector 型として定義した信号をそのまま使用すれば、それぞれ符号ビットなし演算,符号ビットあり演算になる。なお、std_logic_unsigned とstd_logic_signed のパッケージを同時に私用することはできない。

さらに、パッケージstd_logic_arith を用いて,符号ビットなし演算と符号ビット有り演算を混在させる場合、

```
D1 <= unsigned(A) + unsigned(B);
D2 <= signed(A) + signed(B);
```

のように、unsigned 型または signed 型に変換してから演算を行う。

なお、std_logic_unsigned、std_logic_signed,std_logic_arith の三つのパッケージは、IEEE ライブラリに格納されているが、IEEE で承認されたパッケージではなく、米国 Synopsys 社が提供しているパッケージである。

3.2.8 同時処理文と順序処理文

ディジタル回路を構成する個々のゲート回路は、それぞれ並列に動作している。VHDL では、このような並列動作を表現するために、architecture 本体に記述された信号代入文などは、すべて並列に処理される。このように並列処理される文を同時処理文(concurrent statement)という。

一方 C などプログラミング言語では、各文は順番に処理される。このように順番に処理される文を順次処理文（sequential statement）という。順序処理文は,ディジタル回路の動作を表すために不可欠であり、VHDL では、process 文によって、順序処理を表現する。

順序処理文には,if 文、case 文、for-loop 文などがあり、これらを記述する場合、process 文内に記述する必要がある。

なお、architecture 本体に process 文をいくつか記述した場合、process 文内の各文は同時処理されるが、各 process 文は平行に処理される。

3.3 VHDL によるディジタル回路の検証

設計した回路が所望の回路を表しているかどうかを確認することを、設計検証または単に検証という。ここでは、VHDL 記述が所望の回路を表しているかどうかを検証する方法を述べる。

3.3.1 シミュレーション

シミュレーションは、ディジタル回路を検証するもっとも一般的な方法である。ディジタル回路の動作を調べるには、実際に回路を作り、その回路に対してあらゆる可能な入力信号を加え、そのときの出力信号を調べればよい。しかし、この方法ではコストがかかりすぎる。シミュレーションは、この問題点を解決するための検証手段である。

シミュレーションでは、実際の回路を作る代わりに、コンピュータに回路の動作を模擬（シミュレート）させる。このコンピュータ上の模擬回路に対して、あらゆる可能な入出力信号を加え、そのときの出力信号を調べる方法がシミュレーションである。

3.3.2 形式的検証

形式的検証とは、一言でいえば、論理的の等価性判定に基づいた検証手法である。論理関数の等価性判定とは、与えられた二つの論理関数が、同一の真理値表をもつか否かを判定することである。

検証するディジタル回路が n 本の入力信号を持っている組合せ回路である場合、シミュレーションでは、2 ^n種類の入力信号を模擬回路に対して加え、そのときの出力信号を調べる必要がある。しかし、この2 ^nの値は、n の値が少し増加しただけでも,爆発的に増えてしまう。形式検証はこの問題点を解決するための検証手法である。

ディジタル回路は、何らかの論理関数を表している。形式的検証では、検証するディジタル回路を論理関数に変換し、この論理関数が所望の論理関数と対価であるかどうかを調べることによって、ディジタル回路の検証を行う。

形式的検証は、シミュレーションの問題点を解決する画期的な検証方法として、近年研究者の間で注目されるようになってきた。しかし、まだ多くの問題点を抱えており、広く普及するまでに至っていない。そのため一般には、シミュレーションに基づく検証が行われているのが実情である。

3.3.2 テストベンチによる検証

VHDL を使用している場合に限らず、HDL を用いた設計においてシミュレーションを行うには、テストベンチ（testbench）を記述すればよい。テストベンチとは、検証対象となる回路に対して、検証用の入力信号（これをテストベクタ（test vector）という）を加えたり、そのときの出力信号を観測するための HDL 記述である。テストベンチの記述といっても、何か特別な構文を用いるわけではない。これまでに使用した構文とconfiguration 宣言を用いるだけである。

- テストベンチの概要

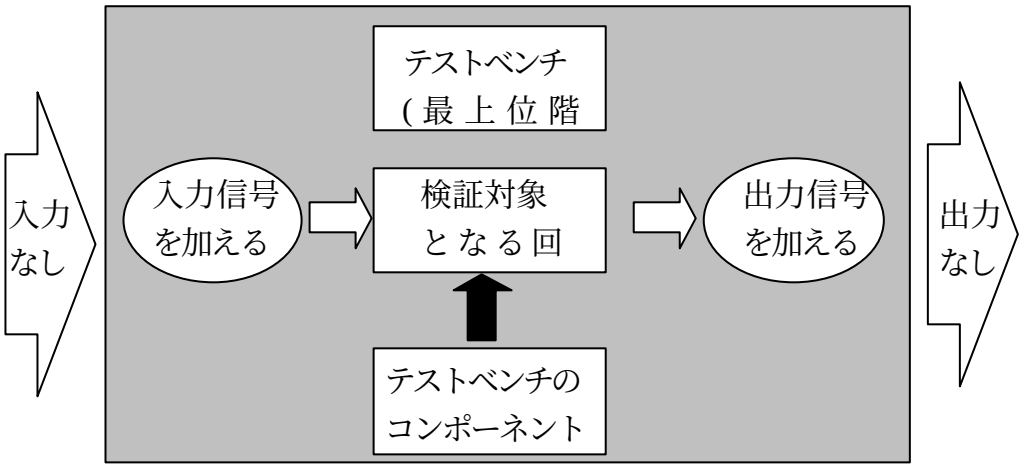


図 3.4 :テストベンチの階層

テストベンチは上図に示すように

1） 入出出力ポートを持たない最上位階層の記述であり、
2） 検証対象回路をコンポーネントとして呼び出し、
3） 検証対象回路にテストベクタを加え、
4） 必要に応じて、そのときの出力信号を観測するような HDL 記述である。

最上位階層の VHDL 記述に対して一つの注意点がある。VHDL では、一つのエンティティに対して、複数のアーキテクチャをもたせることができる。そのため一つのエンティティに対して,どのアーキテクチャを割り当てているのかを指定する必要がある。この指定を行うための構文が、

configuration 宣言である。VHDL では、この configuration 宣言を必ず最上位に記述しておかなければならない。

テストベンチの記述は、最上位階層の記述になるので、configuration 宣言が必要になる。また、コンポーネントを使用している記述を最上位階層とする場合は、その記述内にconfiguration 宣言を含ませる必要がある。

3.3.4 テストベンチの記述例

テストベンチの記述例として VHDL の設計例でかいた リスト3.1 の半加算器のＶＨＤＬ記述 のテストベンチを リスト3.4 に示す。

リスト3.4 に示した構造を順に追っていくテストベンチは、入出力を持たない記述であるために、エンティティの中身は空になっている。また、検証対象回路をコンポーネントとして呼び出すために、component 宣言を記述し、アーキテクチャ本体内でインスタンス化を行っている。この次にテストベクタが記述されている。そしてテストベンチ記述の最後には、configuration 宣言が記述されている。以上がテストベンチの構造である。

まず、リスト3.4 のプロセス P1 では、信号 SA に対して、‘1 ’を加え、その後に 50 nsの間待機する。次に、信号ＳＡに対して、‘0 ’を加え、その後に50 nsの間待機する。プロセスP1 は、この処理を永久に繰り返す。すなわち、100ns 周期の矩形波が SA に加えられていることになる。

このように記述することによって、半加算器に対して “00 ”01 “10 ”“11 ”の 4 種類すべてのテストベクタが 200ns 周期で加えられる。

リスト3.4 :半加算器のテストベンチの記述

```
library IEEE ;
 use IEEE .  std _logic_1164 . All

--テストベンチのエンティティは空
 entity  TEST_BENCH_HA  is
 end TEST_BENCH_HA

 architecture SIM_DATA  of TEST_BENCH_HA  is

-検証対象回路(コンポーネント)の宣言
  component HAFE_ADDER
          port (A ,B : in std_logic ;
               S , C : out std_logic );
 end component ;

signal SA,  SB,  SS,  SC  :  std_logic;

 begin
          --検証対象回路のインスタンス化
          M1 : HALF_ADDER port map (SA,  SB,  SS,  SC) ;
          --テストベクタ
           P1 : process
           begin
                    SA <= `0` ; wait for 50 ns ;
                    SA <= `1` ; wait for 50 ns ;
           end  process ;
          P2 : process
           begin
                     SB <= `0` ; wait for 100 ns ;
                    SB <= `1` ; wait for 100 ns ;
           end  process ;
end  SIM_DATE ;

--configuration宣言 最上位階層では必須）
```

```
configuration  CFG_HA  of  TEST_BENCH_HA  is
        for  SIM_DATA
        end  for ;
end  CFG_HA;
```

第 4章　ハードウェア記述言語による順序回路の設計例

•設計例　ディジタル時計

設計例として比較的身近なアプリケーションであるディジタル時計を設計する。

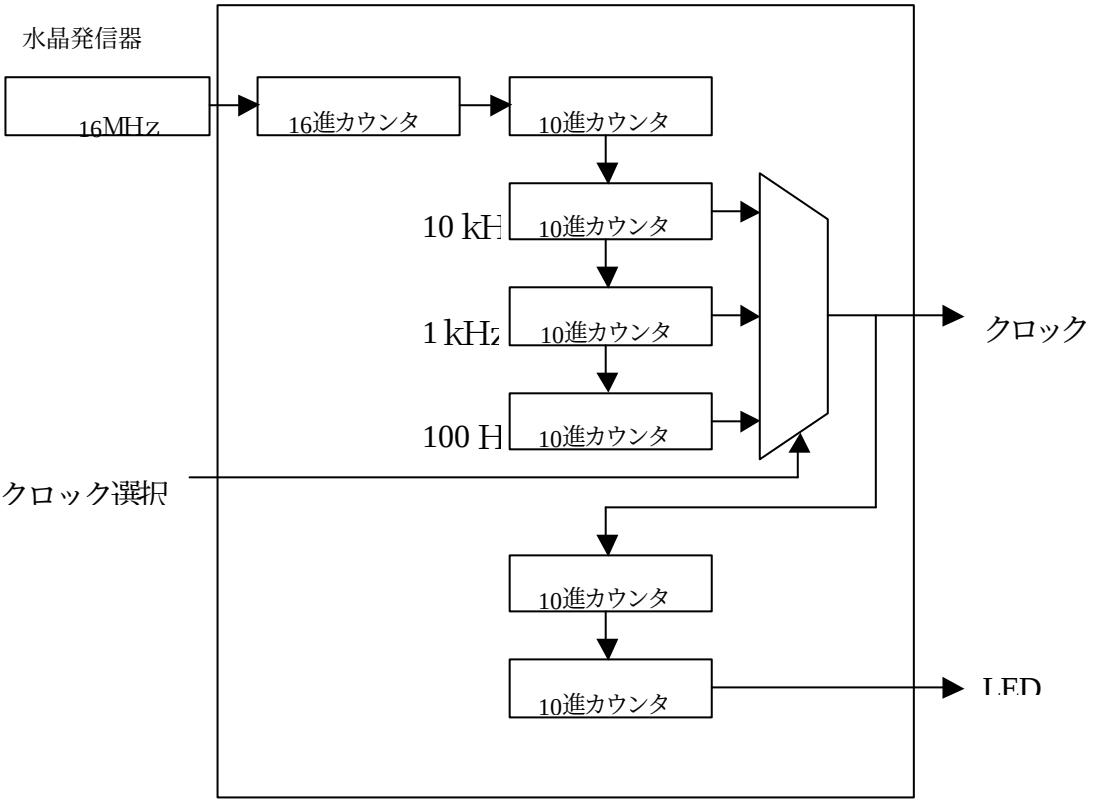


図 4.1 ：クロック発生機能ブロック図

図 4.1に示すように、水晶発振器から16MHz のクロックを入力し、時間カウントに必要な低い周波数のクロックを発生する機能をもたせる。正規のクロック周波数を 100Hz とし、時計動作確認のために外部からスイッチで周波数が正規の 10 倍、100 倍のクロックを選択できるようにする。また、クロックを 100 分周した出力（100Hz のとき 1Hz）に LED を接続し、クロックを目視できるようにする。

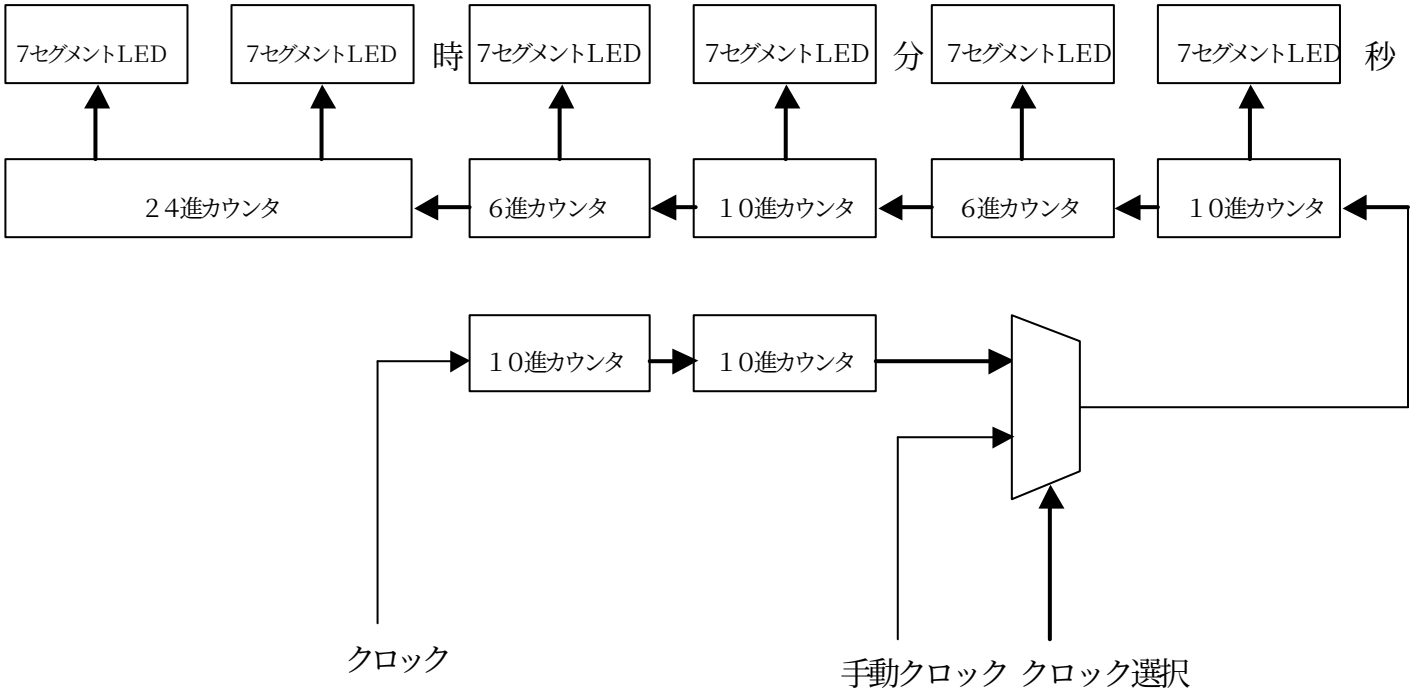


図 4.2 :時間カウントと時刻設定機能ブロック図

100Hz のクロックから「秒」,「分」,「時間」を順次カウントして 7 セグメントLED に表示する時間カウント機能と,外部のスイッチにより手動で時刻設定を行うための機能をもたせる。その、全体のブロック図を図 4.2 に示す。

4.1 クロック発生回路

水晶発振器の 16MHz のクロックが CLK に入力する。これを分周した時間カウント用 100Hz クロックが CLKOUT に出力される。100Hz のほか 1 kHz、10 kHz のクロックは INS3～INS0 によって選択できるようにする。LED 端子に LED を接続し,1 秒の点滅表示を行う。

表 4.1 :割り当てた信号

信号名	入出力	内容
CLK	入力	16Hz のクロックを入力
RESET	入力	リセット入力
INS_3～INS_0	入力	CLKOUT に出力されるクロックを選択
CLKOUT	出力	時間カウントのためのクロック出力
LED	出力	CLKOUT 出力クロックを 100 分週した出力

表 4.2 :クロック出力選択

INS_3	INS_2	INS_1	INS_0	CLKOUT に出力されるクロック周波数
L	L	L	L	100Hz
L	L	L	H	1kHz
L	L	H	L	10 kHz
上記以外				RESET 入力が出力される

4.2 時間カウント回路

CLK には、通常、100Hz のクロックを入力する。これをもとに時分秒カウントに必要なクロックを生成する。

INS3~INS0 端子には S10~S7 が接続され、これらのスイッチで時刻設定を行う。まず、S10 を OFF にすると時刻設定モードに入り、「時」を設定するには S9 を OFF にする。「分」を設定するときは S9 を ON,S8 を OFF にする。そして「秒」を設定するときは S9 と S8 を ON、S7 を OFF にする。時分秒の値には MCLK に接続されたスイッチ S11 を押すごとにそれぞれ更新される。S10 を ON にすると時間カウントをする。RESET はスイッチ S6 に接続し、S を ON にすると時分秒が 00 時 00 分 00 秒にリセットされる。

表 4.3：時刻設定選択表

INS3	INS2	INS1	INS0	説明
L	X	X	X	通常の時間カウント
H	H	X	X	時設定
H	L	H	X	分設定
H	L	L	H	秒設定
H	L	L	L	時分秒表示固定

4.3 VHDL記述とシミュレーション結果

クロック発生回路と時間カウント回路に必要なVHDL記述は、１０進カウンタ・7セグメントLED・クロック発生回路・時間カウント回路の四種類である。それらのVHDL記述とそのテストベンチ及びそのシミュレーション結果をそれぞれのせる。

リスト4.1：10進カウンタのVHDL記述

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity COUNT10 is
      port (
            CLK,RESET  :  in std_logic;
            CYIN       :  in std_logic;
            Q          :  out std_logic_vector(3 downto 0);
            CYOUT      :  out std_logic
      );
end COUNT10;
architecture RTL of COUNT10 is
signal TQ : std_logic_vector(3 downto 0);
begin
process (CLK,RESET,CYIN) begin
       if (RESET = '1') then
           TQ <= "0000" ;
       elsif (CLK'event and CLK ='1' and CYIN ='1') then
           if (TQ = "1001") then
               TQ <= "0000";
           else
               TQ <= TQ + '1';
           end if;
       end if;
end process;

process (TQ,CYIN) begin
      if (TQ = "1001" and CYIN ='1' ) then
          CYOUT <= '1' ;
      else
          CYOUT <= '0' ;
      end if ;
end process ;
Q <= TQ ;
end RTL ;
```

リスト4.2：10進カウンタのテストベンチ（その1）

```
library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;
use work.COUNT10;

entity TESTBNCH is
end TESTBNCH;

architecture stimulus of TESTBNCH is

component COUNT10 is
    port (
        CLK,RESET: in std_logic;
        CYIN: in std_logic;
        Q: out std_logic_vector(3 downto 0);
        CYOUT: out std_logic
    );

end component;


signal CLK: std_logic;
signal RESET: std_logic;
signal CYIN: std_logic;
signal Q: std_logic_vector(3 downto 0);
signal CYOUT: std_logic;
signal Clock_cycle : natural := 0;
begin
    DUT: COUNT10 port map (
            CLK,
            RESET,
            CYIN,
            Q,
            CYOUT
        );
```

リスト4.2：１０進カウンタのテストベンチ（その2）

```
    CLOCK1: process
    begin
        Clock_cycle <= Clock_cycle + 1;
        CLK <= '1' ; wait for 0.5sec;
        CLK <= '0' ; wait for 0.5sec;
    end process CLOCK1;

    STIMULUS1: process
    begin
        RESET <= '1'; wait for 1sec;
        RESET <= '0';
        CYIN <= '1' ; wait;
    end process STIMULUS1;

end stimulus;
```

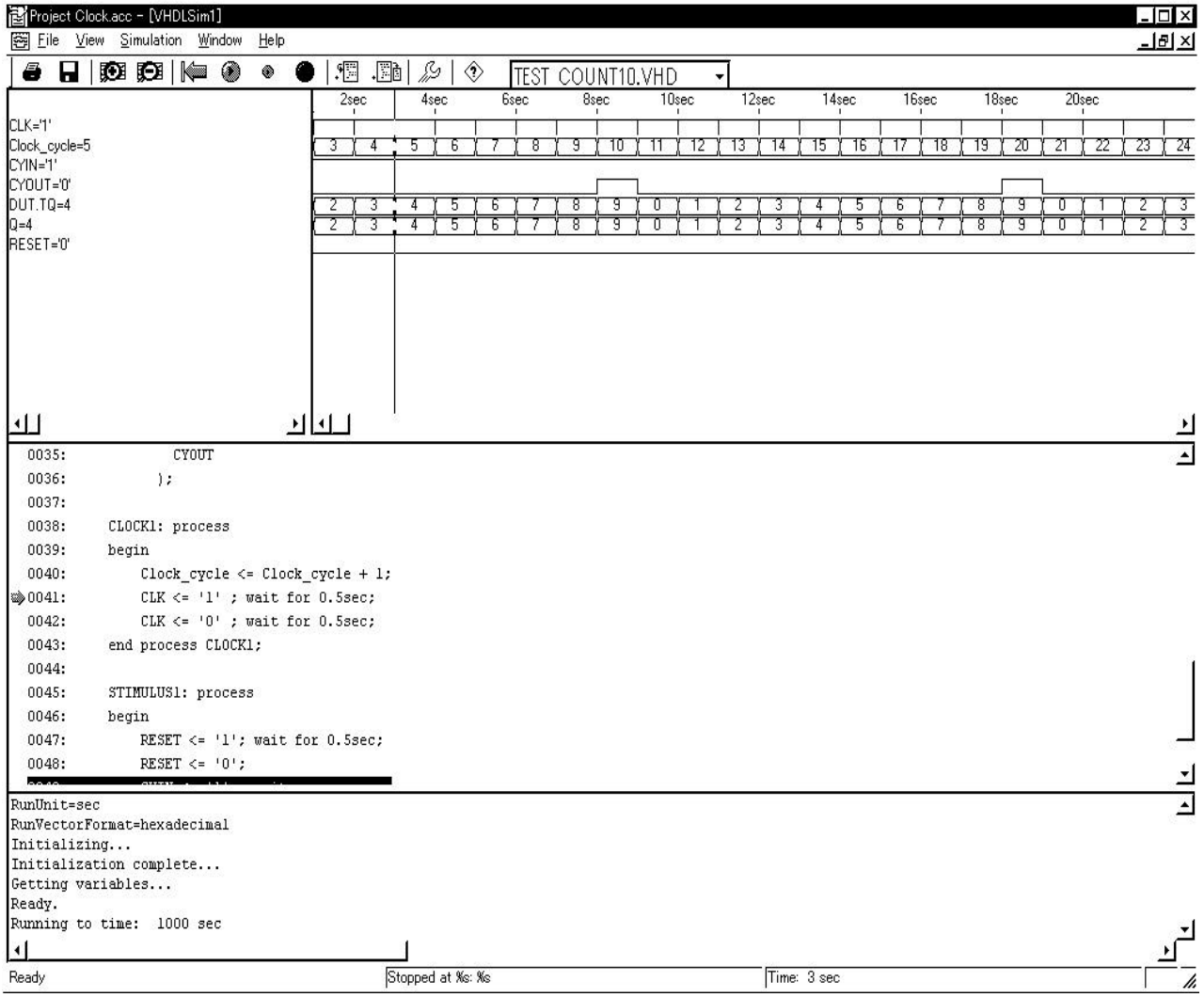


図4.3：10進カウンタのVHDL記述のシミュレーション結果

リスト4.3：7セグメントLEDのVHDL記述

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

entity DECODER_7SEG is
    port (DI: in std_logic_vector(3 downto 0);
          DO:out std_logic_vector(6 downto 0));

end DECODER_7SEG;

architecture DATAFLOW of DECODER_7SEG is
begin
   process(DI)
   begin
        case DI is
            when "0000" => DO <= "0111111"; --0
            when "0001" => DO <= "0000110"; --1
            when "0010" => DO <= "1011011"; --2
            when "0011" => DO <= "1001111"; --3
            when "0100" => DO <= "1100110"; --4
            when "0101" => DO <= "1101101"; --5
            when "0110" => DO <= "1111101"; --6
            when "0111" => DO <= "0000111"; --7
            when "1000" => DO <= "1111111"; --8
            when "1001" => DO <= "1101111"; --9
            when others => DO <= "0000000";
        end case;
   end process;
end DATAFLOW;
```

リスト4.4：7セグメントLEDのテストベンチ

```
library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;
use work.DECODER_7SEG;

entity TEST_BENCH_SEG7 is
end TEST_BENCH_SEG7 ;

architecture SIM_DATA of  TEST_BENCH_SEG7 is
    component DECODER_7SEG
           port (DI: in std_logic_vector(3 downto 0);
               DO:out std_logic_vector(6 downto 0));
end component;

signal DI:std_logic_vector (3 downto 0);
signal DO:std_logic_vector (6 downto 0);

begin
    M1 : DECODER_7SEG port map (DI,DO);
    P1 : process
        begin
          DI <= "0000" ; wait for 1sec;
          DI <= "0001" ; wait for 1sec;
          DI <= "0010" ; wait for 1sec;
          DI <= "0011" ; wait for 1sec;
          DI <= "0100" ; wait for 1sec;
          DI <= "0101" ; wait for 1sec;
          DI <= "0110" ; wait for 1sec;
          DI <= "0111" ; wait for 1sec;
          DI <= "1000" ; wait for 1sec;
          DI <= "1001" ; wait for 1sec;
  end process;
end SIM_DATA ;
```

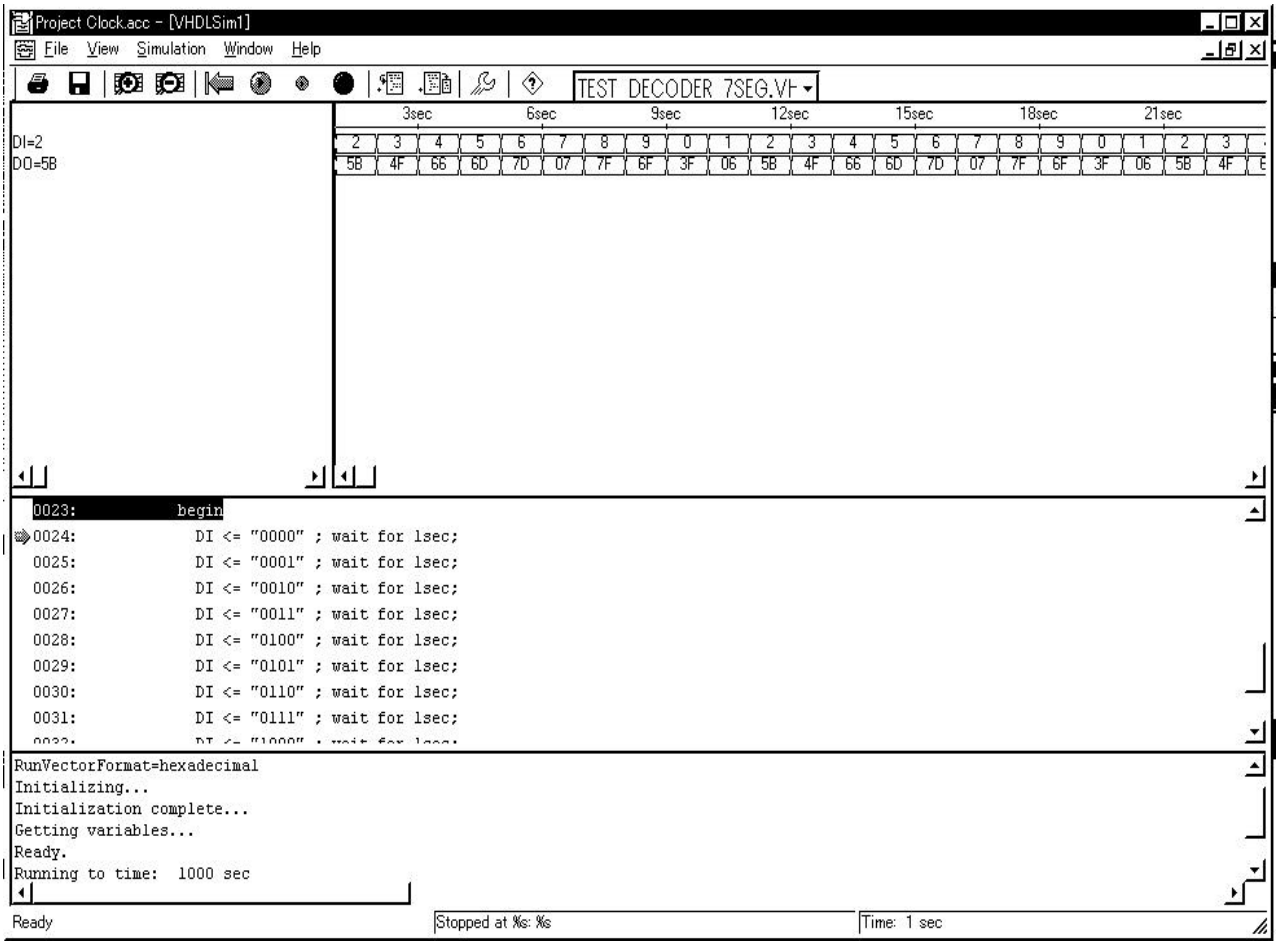


図４．４:7セグメントＬＥＤのシミュレーション結果

リスト4.5：クロック発生回路のＶＨＤＬ記述（その1）

```
library  IEEE ;
use  IEEE .STD_LOGIC_1164. all;
use  IEEE .STD_LOGIC_ARITH. all;
use  ieee.std_logic_unsigned.all;

entity  Prescaler_r2  is
   port (
         CLK , RESET   :  in  std_logic ;
         INS           :  in std_logic_vector (3 downto 0) ;
         CLKOUT , LED   : out std_logic ;
         OT             : out std_logic_vector (3 downto 0)
    ) ;
end Prescaler_r2 ;

architecture  behavioral  of  Prescaler_r2  is

component  COUNT10
    port  (
        CLK, RESET  : in  std_logic ;
            CYIN    : in  std_logic ;
            Q       :  out std_logic_vector (3 downto 0) ;
            CYOUT    :  out std_logic
    );
end component ;

signal TQ16     : std_logic_vector (3 downto 0) ;
signal CY16     : std_logic ;
signal TQ100K     : std_logic_vector (3 downto 0) ;
signal CY100K     : std_logic ;
signal TQ10K     : std_logic_vector (3 downto 0) ;
signal CY10K     : std_logic ;
signal TQ1K     : std_logic_vector (3 downto 0) ;
signal CY1K     : std_logic ;
```

リスト4.5：クロック発生回路のＶＨＤＬ記述（その2）

```
signal TQ100     : std_logic_vector (3 downto 0) ;
signal CY100     : std_logic ;
signal TCLK     : std_logic ;
signal TTQ10     : std_logic_vector (3 downto 0) ;
signal TCY10     : std_logic ;
signal TTQ1     : std_logic_vector (3 downto 0) ;
signal TCY1     : std_logic ;

begin
process (RESET,CLK) begin
   if (RESET = '1') then
     TQ16 <= "0000";
   elsif (CLK'event and CLK='1')then
       if (TQ16="1111") then
          TQ16 <= "0000" ;
       else
           TQ16 <= TQ16 + '1' ;
       end if ;
   end if;
end process ;

process (TQ16) begin
  if (TQ16="1111") then
    CY16 <= '1' ;
  else
     CY16 <= '0';
  end if ;
end process;

H0 : COUNT10 port map (CLK, RESET, CY16, TQ100K, CY100K) ;
H1 : COUNT10 port map (CLK, RESET, CY100K, TQ10K, CY10K) ;
H2 : COUNT10 port map (CLK, RESET, CY10K, TQ1K, CY1K) ;
H3 : COUNT10 port map (CLK, RESET, CY1K, TQ100, CY100) ;
```

リスト4.5：クロック発生回路のＶＨＤＬ記述（その3）

```
process (INS, TQ100(3), TQ1K(3), TQ10K(3), RESET) begin
  case INS is
    when "0000" => TCLK <= TQ100(3) ;
    when "0001" => TCLK <= TQ1K(3) ;
    when "0010" => TCLK <= TQ10K(3) ;
    when others => TCLK <= RESET ;
  end case ;
end process;
CLKOUT <= TCLK ;
H6 : COUNT10 port map (TCLK, RESET, '1', TTQ10, TCY10) ;
H7 : COUNT10 port map (TCLK, RESET, TCY10, TTQ1, TCY1) ;
LED <= TTQ1(3) ;
OT <= INS;
end behavioral ;
```

リスト4.6：クロック発生回路のテストベンチ（その1）

```
library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;
use work.PRESCALER_R2;

entity TESTBNCH is
end TESTBNCH;

architecture stimulus of TESTBNCH is
component PRESCALER_R2 is
    port (
        CLK: in std_logic;
        RESET: in std_logic;
        INS: in std_logic_vector(3 downto 0);
        CLKOUT: out std_logic;
        LED: out std_logic;
        OT: out std_logic_vector(3 downto 0)
    );

end component;

signal CLK: std_logic;
signal RESET: std_logic;
signal INS: std_logic_vector(3 downto 0);
signal CLKOUT: std_logic;
signal LED: std_logic;
signal OT: std_logic_vector(3 downto 0);
signal done: boolean := false;
signal Clock_cycle : natural :=0;
```

リスト4.6：クロック発生回路のテストベンチ（その2）

```
begin
    DUT: PRESCALER_R2 port map (
           CLK,
           RESET,
           INS,
           CLKOUT,
           LED,
           OT
         );

    CLOCK1: process
```

リスト：クロック発生回路のテストベンチ（その2）

```
   begin
       Clock_cycle <= Clock_cycle + 1;
       CLK <= '1' ;wait for 0.5sec;
       CLK <= '0' ;wait for 0.5sec;
    end process CLOCK1;

    STIMULUS1: process
    begin
       RESET <= '1';  -- Reset the system
       wait for 1sec; -- Wait one clock cycle

       RESET <= '0';

       wait;          -- Suspend simulation
    end process STIMULUS1;

       STIMULUS2: process
    begin
        INS <= "0000";wait  ;
    end process;
end stimulus;
```

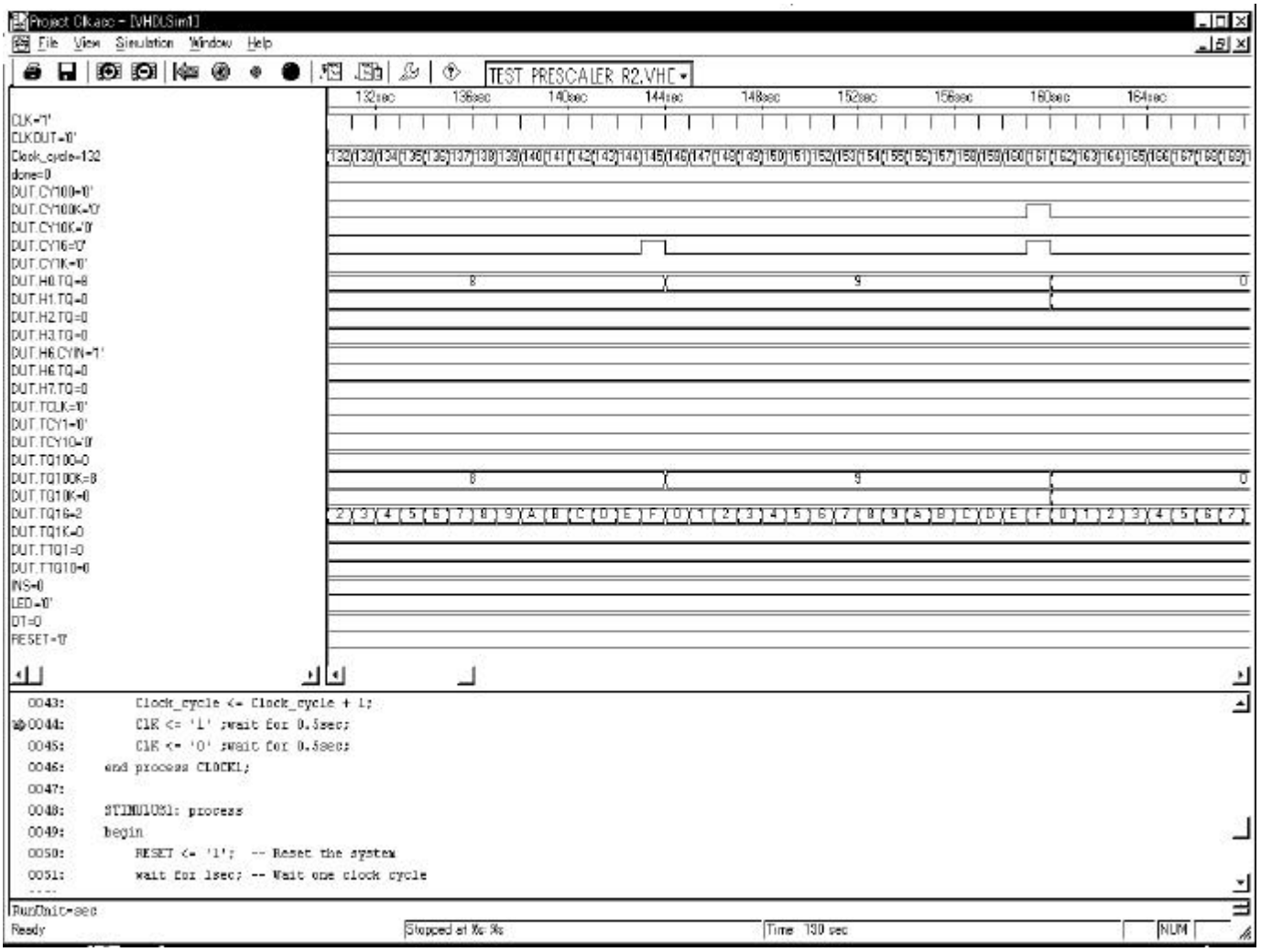


図4.5：クロック発生回路のシミュレーション結果

リスト4.7 1時間カウント回路（その1）

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

entity Clock is
     port (
           RESET,CLK : in std_logic;
           INS       : in std_logic_vector(3 downto 0);
           MCLK      :in std_logic;
           HHI,HLO   : out std_logic_vector(6 downto 0);
           MHI,MLO   : out std_logic_vector(6 downto 0);
           SHI,SLO   : out std_logic_vector(6 downto 0);
           HDP,MDP   : out std_logic
);
end Clock;

architecture behavioral of Clock is

component COUNT10
   port (
            CLK,RESET  :  in std_logic;
            CYIN       :  in std_logic;
            Q          :  out std_logic_vector(3 downto 0);
            CYOUT      :  out std_logic
      );
end component;

component DECODER_7SEG
           port (DI: in std_logic_vector(3 downto 0);
               DO:out std_logic_vector(6 downto 0)
);
end component;
```

リスト4.7 1時間カウント回路（その2）

```
component DPOINT
   port (
          DPIN : in std_logic;
          DPOUT : out std_logic
);
end component;


signal TQ10 : std_logic_vector (3 downto 0);
signal CY10 : std_logic;
signal TQ1 : std_logic_vector (3 downto 0);
signal CY1 : std_logic;
signal QHH,QHL,QMH,QML,QSH,QSL : std_logic_vector (3 downto 0);
signal CYHL,CYMH,CYML,CYSH,CYSL : std_logic;
signal TCLK : std_logic ;
signal TCYSH : std_logic ;
signal TCYMH : std_logic ;
signal CLK1S : std_logic ;
signal MANCLK : std_logic ;
signal SFR : std_logic_vector (2 downto 0);


begin
H1:COUNT10 port map(CLK,RESET
                    ,'1',TQ10,CY10);
H2:COUNT10 port map(CLK,RESET
                    ,CY10,TQ1,CY1);
CLK1S <= TQ1(3);
```

リスト4.7 1時間カウント回路（その3）

```
process (MCLK,CLK)begin
         if (MCLK='1') then
             SFR <= "000";
         elsif (CLK'event and CLK = '1') then
             SFR(0) <= not MCLK;
             SFR(1) <= SFR(0);
             SFR(2) <= SFR(1);
         end if;
end process;

MANCLK <= SFR(2);

process(INS,CLK1S,MANCLK)begin
     if (INS(3)='0')then
         TCLK <= CLK1S;
     else
        TCLK <= MANCLK;
     end if;
end process;

process (TCLK, RESET,INS) begin
      if (RESET = '1') then
          QSL <= "0000";
      elsif (INS(3) = '0' or INS ="1001")then
         if (TCLK'event and TCLK='1')then
             if (QSL = "1001") then
                QSL <="0000";
             else
                QSL <= QSL + '1';
             end if;
         end if;
     end if;
end process;
```

リスト4.7 時間カウント回路（その4）

```
process(QSL,INS)begin
    if (INS(3)='0') then
        if (QSL = "1001")then
           CYSL <= '1';
        else
           CYSL <= '0';
        end if;
    elsif (INS(2) = '0' and  INS(1) = '0' and INS(0) = '1')then
        if (QSL = "1001")then
           CYSL <= '1';
        else
           CYSL <= '0';
        end if;
    else
        CYSL <= '0';
    end if;
end process;


process (RESET,TCLK,CYSL) begin
      if (RESET = '1') then
          QSH <= "0000";
      elsif (TCLK'event and TCLK='1' and CYSL ='1')then
            if (QSH = "0101") then
               QSH <="0000";
            else
                 QSH <= QSH + '1';


            end if;
         end if;
end process;
```

リスト4.7 1時間カウント回路（その5）

```
process(QSH,CYSL,INS)begin
    if (INS(3)='0') then
        if (QSH = "0101" and CYSL = '1')then
           CYSH <= '1';
        else
           CYSH <= '0';
        end if;
    elsif (INS(2) = '0' and  INS(1) = '0' and INS(0) = '1')then

           CYSH <= '0';
    end if;
end process;

process (RESET,TCLK,CYSH,INS) begin
      if (INS(3) = '0') then
          TCYSH <= CYSH;
      elsif (INS(2) = '0') then
          TCYSH <= '1';
      else
          TCYSH <= '0';
      end if;
      if (RESET ='1' ) then
      QML <= "0000";
      elsif  (INS(3)='0' or (INS(3)='1' and INS(2)='0'
                                and INS(1) ='1'))then
      if (TCLK'event and TCLK='1' and TCYSH ='1')then
         if(QML = "1001")then
            QML <= "0000";
         else
            QML <= QML + '1';
         end if;
      end if;
    end if;
end process;
```

リスト4.7 1時間カウント回路（その6）

```
process (QML,TCYSH) begin
   if(QML = "1001" and TCYSH = '1') then
      CYML <= '1';
   else
      CYML <= '0';
   end if;
end process;

process(RESET,TCLK,CYML)begin
   if(RESET = '1')then
      QMH <= "0000";
   elsif (TCLK'event and TCLK='1' and CYML='1')then
      if(QMH = "0101")then
         QMH <= "0000";
      else
         QMH <= QMH + '1';
      end if;
   end if;
end process;

process(QMH,CYML,INS)begin
      if (INS(3)='0')then
        if(QMH="0101"and CYML='1')then
           CYMH <= '1';
        else
           CYMH <= '0';
        end if;
     else
         CYMH <= '0';
     end if;
end process;
```

リスト4.7 1時間カウント回路（その7）

```
process (RESET,TCLK,CYMH,QHH,INS) begin
      if (INS(3)='0') then
          TCYMH <= CYMH;
      elsif (INS(2)='1') then
          TCYMH <= '1';
      else
          TCYMH <= '0';
     end if;

      if (RESET = '1') then
          QHL <= "0000";
      elsif(TCLK'event and TCLK='1' and TCYMH='1')then
         if (QHL="1001" or (QHH="0010" and QHL="0011"))then
             QHL <= "0000";
         else
             QHL <= QHL + '1';
         end if;
      end if;
end process;

process (QHL,TCYMH,QHH) begin
      if((QHL = "1001" or (QHL = "0011" and QHH="0010"))
                                    and TCYMH = '1') then

          CYHL <='1';
      else
          CYHL <='0';
      end if;
end process;
```

リスト4.7 1時間カウント回路（その8）

```
process (RESET,TCLK,CYHL) begin
      if (RESET = '1') then
         QHH <= "0000";
      elsif(TCLK'event and TCLK ='1' and CYHL='1') then
         if (QHH="0010") then
            QHH <="0000";
         else
             QHH <= QHH + '1';
         end if;
      end if;
end process;

U0 : DECODER_7SEG  port map  (QHH,HHI);
U1 : DECODER_7SEG  port map  (QHL,HLO);
U2 : DECODER_7SEG  port map  (QMH,MHI);
U3 : DECODER_7SEG  port map  (QML,MLO);
U4 : DECODER_7SEG  port map  (QSH,SHI);
U5 : DECODER_7SEG  port map  (QSL,SLO);
V0 : DPOINT port map ('0',HDP);
V1 : DPOINT port map ('0',MDP);

end behavioral;
```

リスト4.8 :時間カウント回路のテストベンチ（その1）

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

entity TESTBNCH is
end TESTBNCH;

architecture stimulus of TESTBNCH is
component CLOCK is
    port (
        RESET,CLK: in std_logic;
        INS: in std_logic_vector(3 downto 0);
        MCLK: in std_ulogic;
        HHI,HLO: out std_logic_vector(6 downto 0);
        MHI,MLO: out std_logic_vector(6 downto 0);
        SHI,SLO: out std_logic_vector(6 downto 0);
        HDP,MDP: out std_logic
    );

end component;

signal RESET,CLK: std_logic;
signal INS: std_logic_vector(3 downto 0);
signal MCLK: std_ulogic;
signal HHI,HLO: std_logic_vector(6 downto 0);
signal MHI,MLO: std_logic_vector(6 downto 0);
signal SHI,SLO: std_logic_vector(6 downto 0);
signal HDP,MDP: std_logic;
signal done: boolean := false;
signal Clock_cycle : natural := 0;
```

リスト4.8 時間カウント回路のテストベンチ（その2）

```
signal MHI,MLO: std_logic_vector(6 downto 0);
signal SHI,SLO: std_logic_vector(6 downto 0);
signal HDP,MDP: std_logic;
signal done: boolean := false;
signal Clock_cycle : natural := 0;

begin
    DUT: CLOCK port map (
           RESET,CLK,
           INS,
           MCLK,
           HHI,HLO,
           MHI,MLO,
           SHI,SLO,
           HDP,MDP
         );

    CLOCK1: process
    begin
        Clock_cycle <= Clock_cycle +1;
        CLK <= '1' ; wait for 0.05sec;
        CLK <= '0' ; wait for 0.05sec;
    end process CLOCK1;

    STIMULUS1: process
    begin
        RESET <= '1';  -- Reset the system
        wait for 0.1sec; -- Wait one clock cycle
        RESET <= '0';
        MCLK <= '0';
        wait;          -- Suspend simulation
    end process STIMULUS1;
```

リスト4.8 時間カウント回路のテストベンチ（その3）

```
        STIMULUS2: process
    begin
         INS(3) <= '0';wait;

    end process STIMULUS2;
end stimulus;
```

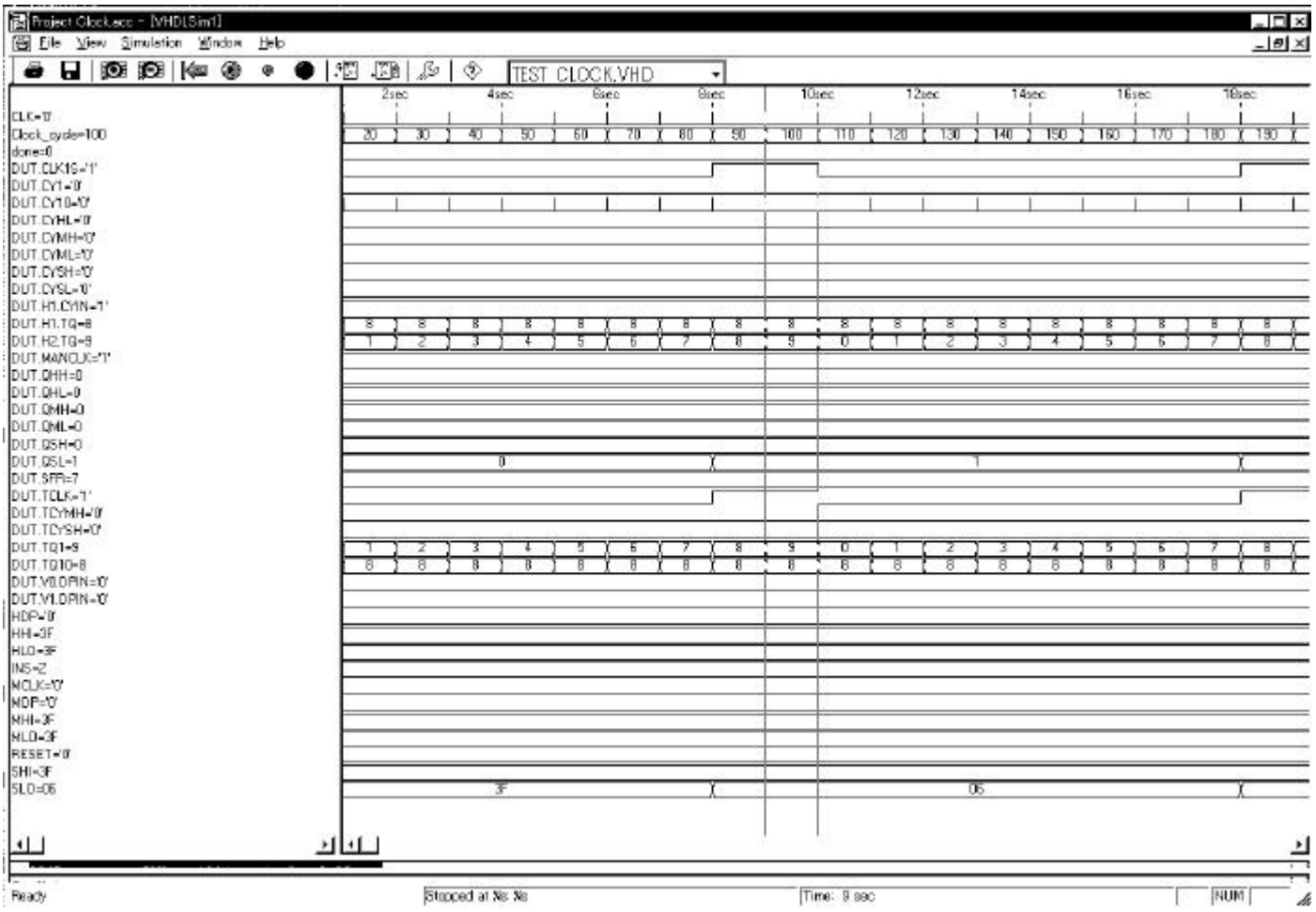


図４．６:時間カウンタ回路のVHDL記述のシミュレーション結果

第 5 章　まとめ

今回、この実験を行うについて、第 2 章で順序回路について第 3 章で VHDL について勉強してきた。そして、第 4 章で設計例としてディジタル時計をハードウェア記述言語（VHDL）で設計をした。

ディジタル時計回路記述自体はそれほど複雑ではなく、そして単体で機能するので VHDL の入門の題材に適していたと思われたため、ディジタル時計を選び設計をした。今回の製作では、たんに時分秒を表示する時計を記述し、その後時刻設定機能を持たせる記述を行った。このようにハードウェア記述言語を用いて回路を設計をすれば、はんだごてを使って配線を変えることなく回路を変更していろいろな機能を実現させて楽しむことができた。

謝辞

本実験を行うに際し、終始、懇切丁寧な御指導、御鞭撻を賜わりました。高知工科大学 電子・光システム工学科 矢野政顕 教授に心から感謝いたします。

研究中、懇切丁寧なご指導を賜わ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、高知工科大学 電子・光システム工学科 河津哲 教授、高知工科大学 電子・光システム工学科 橘昌良 助教授に厚くお礼申し上げます。

また終始、適切な御助言、御助力をいただきました高知工科大学 電子・光システム工学科専攻 北野克幸氏、小泉義信氏、小島大輔氏、幾井崇博氏、中村基継氏、田口禎譲氏ほか計算機応用講座研究室の皆様に心から感謝しお礼申し上げます。

参考文献

「ディジタル回路の考え方」 雨宮他　オーム社
「基礎からのディジタル IC」 宮本義博　林正義　技術評論社
「よくわかるディジタル IC 回路の基礎」松田巖　伊原充博　技術評論社
「論理回路の基礎（過程版）」 田丸啓吉　工学図書株式会社
「論理回路の設計」 浅川毅　オーム社
「論理回路入門」 渡辺隆司　森北出版株式会社
「HDL による VLSI 設計」深山正幸　北川章夫　秋田純一　鈴木政国
共立出版株式会社
「VHDL 言語入門」Jayaram　Bhasker 著　CQ 出版社
「VHDL の基礎」Z・ナビバ　著/王一幸　訳
Design　Wave　MAGAZIN　CQ 出版社