卒 業 研 究 報 告

題　　目

VHDL によるディスプレイを

表示させるためのカウンタの設計

指 導 教 員

橘　昌良助教授

報 告 者

木下　典子

平成 14　年 2　月 7　日

高知工科大学 電子･光システム工学科

目次

第 1 章

1. はじめに

現在、私たちの身の回りにはパソコン、携帯電話、ゲーム、テレビなどの映像や画像が表示されるものが多数出回り、生活をしていく上で目にするようになってきた。

私はパソコンやゲームのような動く画像に興味を持ち、卒業研究で液晶ディスプレイに絵を表示させ、その表示された絵が移動や変形したりするアニメーションになって動くような回路を設計したいと思った。今回本論文では、ハードウェア記述言語である VHDL とディスプレイを表示させるためのカウンタ回路の設計とを研究し理解を深めることを目的とし調査してまとめたものである。

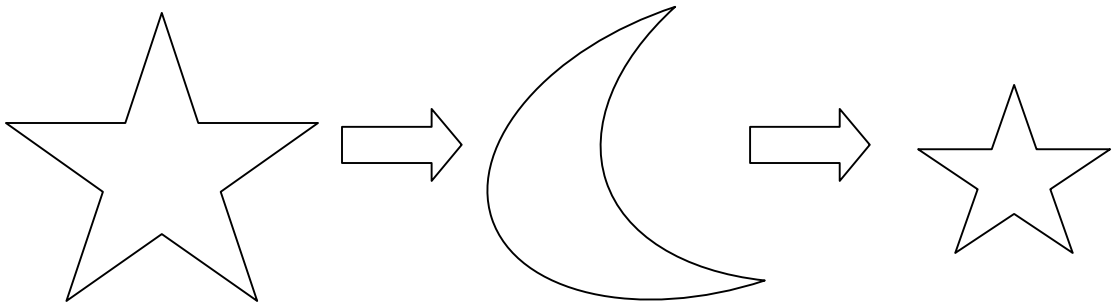


図 1.　表示させる絵の例

上の絵のように星型から月型に変形し、小さい星型になるような色々な形に変形・拡大・縮小する絵をディスプレイに表示される回路を設計する予定であった。だが、思っていたより時間がかかってしまい絵を表示させるところまでは到達できずに、ディスプレイを表示させるためのカウンタ回路を VHDL で設計するところまで行った。

本論文の構成は以下のようなものである。第 1 章のはじめにでは研究の目的・概要について、第 2 章では VHDL の基本的な文法などをまとめた。第 3 章ではディジタル画像の表現方法、第 4 章ではディスプレイに表示させるためのカウンタ回路の設計手順や、動作についてまとめた。VHDL のモジュール、テストベンチのソース、シミュレーション結果は付録にしてある。

第 2 章 VHDL（ハードウェア記述言語）について

2. VHDL とは

2.1　VHDL の歴史

ハードウェア記述言語である VHDL（Very high speed integrated circuit Hardware Description Language）は、米国国防省の VHSIC（Very Speed Integrate 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おり、その当時の国防省向け ASIC（特定用途向け集積回路）の開発は長いもので 3～4 年もかかっていた。一番スピードが速い ASIC を使用していたが、その間に半導体のプロセスは進歩し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。そこでロジックゲートを直接回路図で入力するのではなく、ハードウェア記述言語で設計することによって、開発終了時に一番スピードが速い ASIC を選択可能にする必要があった。

こうして VHDL が作られ、言語仕様書もできた。またその後、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。

2.2　VHDL の表現方法

○BEHAVIAR 記述

機能レベルでの振る舞いや動作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。だが、直接論理合成ができないから、ASIC や FPGA などの設計には使用しない。

○RTL 記述（Register Transfer Level）

直接論理合成ができ、クロックをベースにしたレジスタや、組み合わせ回路をまとまった論理式で記述するレベルである。これは ASIC や FPGA の設計に使用される。

○ゲート・レベル記述

ネットリストと呼ばれる、ASIC の内部のゲートやセルベースの接続関係で表したリストで記述されるレベルのこと。RTL 記述では表現できない高速性や、高密度を要求する場合に用いられる。

2.3　VHDL 設計のメリット

ハードウェア記述言語による設計手法は、ASIC などの大規模集積回路や比較的小規模な FPGA・PDL の設計などに利用されており、ハードウェアで設計する上で欠かすことのできない同時処理や時間、タイミングなどを記述できるという特徴も持っている。

VHDL によって

○ 設計期間が 1/2~1/3 短縮される

→VHDL による設計は、より抽象度の高いレベルでの設計をする事で難しい。論理式から設計者を解放し、設計期間を短縮する事が可能である。

○ 完成度の高いシステムが構築できる

○ 設計の再利用、変更が容易にできる

→抽象度の高い記述であるいうことは、それだけ設計の変更が容易になるという事で、設計者はより完成度の高いシステムを構築する事ができる。

以上のようなメリットがある。

回路図入力による設計と VHDL 入力による設計の違いは下図に示す。

	回路図入力	VHDL入力
1	回路図入力に時間がかかる	テキストで簡単に入力できる
2	論理式を考える必要がある	論理式を考える必要がない
3	回路変更が大変	回路変更が容易
4	設計者以外では、内容を理解しにくい	誰にでも理解しやすい
5	特定の半導体メーカのゲートライブラリを使用して回路入力をする	半導体メーカのゲートライブラリを使用しない。どのメーカでも作成可能

表1. 回路図入力による設計の違いとVHDLによる設計の違い

2.4　VHDL での記述方法

2.4.1　std_logic

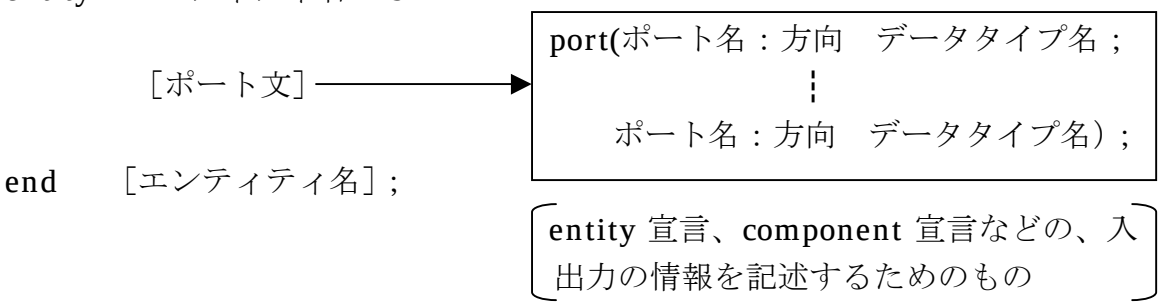
VHDL には、多くのデータタイプがあるが回路設計する際には“std_logic”、“std_logic_vector”を使用する。これらのデータタイプを使用する場合は VHDL の文頭に

```
library IEEE;
use IEEE.std_logic.all;
```

を必ず記述しなければならない。

2.4.2　エンティティとアーキテクチャ

VHDL で回路設計をする場合、どのようなことをさせたいのかなどの情報を記述する必要がある。どのような信号が入力されるか、どのような信号を出力させるかという、外部とのインターフェース（入力信号や出力信号）に関する情報をエンティティ内に記述する。これに対して、その信号にどのような加工をさせるかという情報は、回路の内部の動作や構造に関する事なのでアーキテクチャ内に記述する。また、一つのエンティティ宣言は、複数のアーキテクチャ宣言をもつことができる。すなわち、ビヘービァ・レベルの記述と RTL での記述、ゲート・レベルでの記述などの異なるレベルの記述が一つのエンティティでカバーができる。



『アーキテクチャ』

architecture　アーキテクチャ名　of　エンティティ名　is

＜宣言文＞ ……▶ (例)　signal count_in : in std_logic;

begin

　＜同時処理文＞

end　アーキテクチャ名；

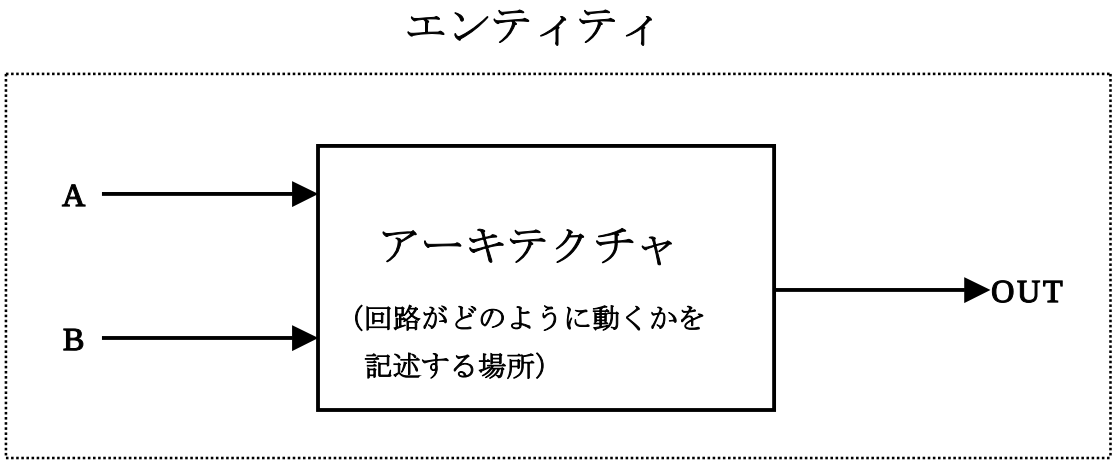


図 2.　エンティティとアーキテクチャ

2.4.3　名付け規則とコメント文

(1)名前の付け方

VHDL では大文字と小文字の区別がないので、どちらで書いても同じ語になる。ただし、例外が 1 つある。‘　’と“　”で囲まれた文字だけは大文字と小文字を区別する。

```
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

VHDL で使用する名前（信号名、ポート名、エンティティ名、アーキテクチャ名など）の付け方には、次のような規則がある。

○最初の文字は英字

○使用できる文字は英字・数字・‘__’

○続けて‘__’を使用してはならない。また最後の文字に‘__’を使用してはならない。

(2)コメント文

VHDL で“－－”の後に記述するとコメント文という文になる。これは設計者が名前、日付、記述した信号などの意味や動作が後から見てわかるようにという目的で記述する。コメント文は VHDL として処理されない。

2.5　階層設計

2.5.1　階層設計とは

階層設計とは、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おこなうような設計方法のことをいう。この時各ブロックは、必要に応じてさらに細かく分割されて階層設計される。大規模な回路を設計する時に用いられる。

2.5.2　コンポーネント

VHDL では各ブロックのことをコンポーネントと呼ぶ。VHDL でコンポーネントを使用する場合、あらかじめコンポーネント宣言をする必要がある。記述の仕方は次の通りである。

```
component　　コンポーネント名
［ポート文］
end　component ;
```

2.6　算術演算子

４ビットの加算器を記述する場合、フルアダー４個を並べて記述することができるが、そのようにすると何行も記述しなければならない。しかし、算術演算子を使用すれば、１行で記述ができ、ゲート数が少なくて速い回路を生成できる。実際、ロジック回路を生成できる算術演算子は、“＋”，“－”，“＊”だけである。簡単に記述ができるが、ビット長が長い場合は単純な記述で大量のロジックを生成するので慎重に使用しなければならない。

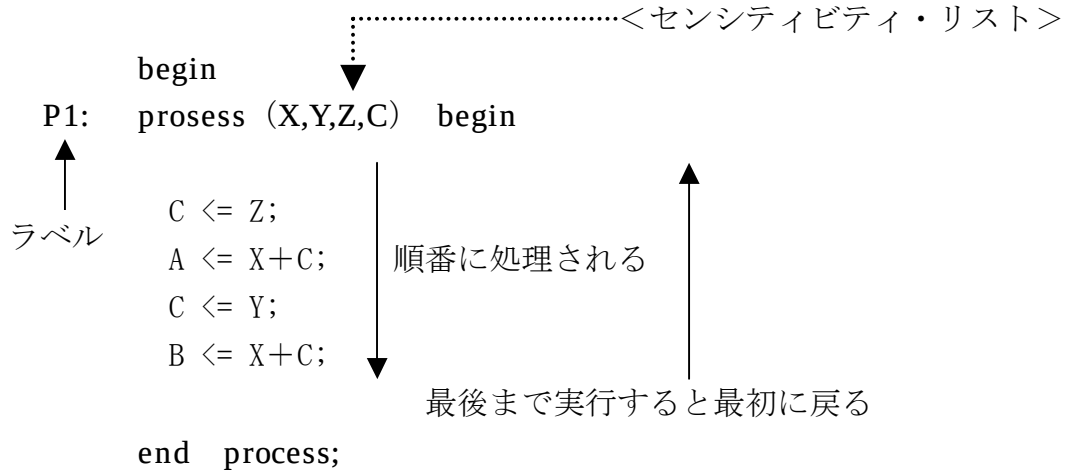
2.7　プロセス文

2.7.1　同時処理文

ディジタル回路を構成するゲート回路は、それぞれ並列に動作をしている。VHDL ではその並列動作を表現するために、同時処理文が用いられる。”architecture” “begin“ “end “の間に記述されたもののことを同時処理文という。

また、同時処理文には、ラベルを付けることができる。これは回路の記述を見やすくためや、シミュレーションの時の目印になるので用いられ、動作には影響を及ぼさない。（ジェネレート文除く）ラベルは、回路の中で他の記述で変数等に使われていない名前でなくてはならない。

2.7.2　プロセス文の記述



プロセス文はセンシティビティ･リストの値が変化すると実行が開始されて、記述の上から順番に処理していく。

プロセス文の中に if 文、case 文、for-loop 文などの順次処理文が記述できる。

2.8　if 文の記述

2.8.1　if 文

組み合わせ回路を生成するにはプロセス文のセンシティビティ･リストに入力信号の記述と、if 文の条件項にすべての場合を記述する必要がある。if 文の文法は下の通りである。

if 条件 then
{順次処理文}
end if;
	if 条件 then
	{順次処理文}
	else
	{順次処理文}
	end if;
		if 条件 then
		{順次処理文}
		{elsif 条件 then {順次処理分}}
		[else {順次処理文}]
		end if;

表 2.　　if 文の書式

2.8.2　if 文を使用するにあたっての注意

if 文では else 項を使用する。else は残りのすべての場合という意味である。これを記述してあれば、すべての場合を記述しているということになって、誤動作をすることはなくなる。必ず、else を使用しなければならないわけではない。省略する場合は、すべての場合の記述をしているかどうかを確認する必要がある。

2.9　case 文の記述

when 値　＝＞ 文

case 文は if 文とは異なり、値の順番がなく、すべてが並列に処理される。注意しなければならないのは、一度 when 項に記述した値を、その後で再び使うと文法エラーになるので、値が重複しないようにすることである。case 文の書式は以下の通りである。

```
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　other ＝＞　順次処理文
```

表 3.　case 文の書式

2.10　for-loop 文の記述

VHDL 記述中で規則的に繰り返し使用するものを、一つずつ記述するのは面倒である。何個も記述せずに、記述回数を減らすことができるのが for-loop 文である。

for-loop 文は、プロセス文内よりもサブプログラムでよく利用される。

2.11　関係演算子

2.11.1　関係演算子の特徴

○関係演算子の右辺と左辺は、同じデータ･タイプでないといけないが同じビット幅である必要はない。

○ 関係演算子は、算術演算子ほどではないが、ビット長が増えると巨大なロジックを生成する。

○ 関係演算子は、std_logic_vector などのベクタ･タイプ同士を比較する場合、1 番左のビットから順に比較していく。したがって、数値の比較では 1 番左が MSB ビットでないと正しい比較ができない。

=	等号	すべてのデータタイプで使用可能
/=	不等号	
<	より小さい	“integer”･“Real” “std_logic”･“std_logic_vector”で使用可能
>	より小さい大きい	
<=	より小さいか等しい	
>=	より大きいか等しい	

表 4.　関係演算子の 6 種類

2.12　順序回路の記述

2.12.1　DFF 生成の記述

ロジック回路設計にはロジックを生成させる記述のほかにフリップフロップのような順序回路も必要である。以下の記述がフリップフロップの記述である。

```
library  IEEE;
use  IEEE.std_logic_1164.all;
entity  DFF  is
    port( CLK, D   : in   std_logic;
          Q        : out  std_logic);
end  DFF;

architecture  RTL  of  DFF  is
begin                          ←―― CLK のみ記述

    process  (CLK) begin       ←―― 立ち上がりエッジの検出
                                    （立下りの場合は CLK='0'にする）
      if (CLK' event  and  CLK='1')  then

        Q  <=  D;              ←―― 入力データ D を出力 Q に代入
      end  if;
    end  process;
end  RTL;
```

図 3.　D フリップフロップの記述例

2.12.2　強制リセットの記述

フリップフロップに強制リセットを付け加える場合は、強制リセット入力“RESET”を記述する。強制リセットは CLK よりも優先される。上記の DFF の記述に強制リセットを入れた記述は以下の通りである。

```
library  IEEE;
use  IEEE.std_logic_1164.all;
entity  DFF  is
    port( CLK, D, RESET  : in  std_logic;
          Q          : out  std_logic);
end  DFF ;

architecture  RTL  of  DFF  is
begin                                  CLK と RESET を記述

    process  (CLK, RESET) begin

      if (RESET='1')  then   ←――――――   強制リセット記述
                                          （CLK より優先）
        Q  <＝ '0';

        elsif (CLK' event  and  CLK='1')  then
        Q＜＝  'D';
      end  if;
    end  process;
end  RTL;
```

図 4.　強制リセットの記述例

2.13 ディジタル回路の検証

2.13.1　シミュレーション

ディジタル回路を設計したら、その回路が思い通りに動作をするかどうかをシミュレーションして確認しなければならない。シミュレーションというのは「回路動作に必要な入力を与え、出力の変化を観測する」ことである。

VHDL で回路を設計した場合もシミュレーションをする。方法としては、テストベンチを VHDL で記述すればよい。

2.13.2　テストベンチとは

VHDL で回路を記述したら、その回路が正しく動作するかどうかをテストするために記述をしなければならない。これをテストベンチという。

2.13.3　テストベンチの概要

テストベンチは

①入出力ポートを持たない最上階層の記述である。

②検証対象回路をコンポーネントとして呼び出す。

③検証対象回路にテスト・ベクタを加える。

④必要に応じて、その時の出力信号を観測する。

というような VHDL 記述である。

テストベンチ記述としては、プロセス文・wait 文・コンフィグレーション文を用いて記述をしていく。

VHDL はひとつのエンティティに対して、複数のアーキテクチャを持たすことができる。コンフィグレーション文というのは、ひとつのエンティティに、アーキテクチャを割り当てるかを指定する必要があるので、この指定を行う構文のことである。このコンフィグレーションの宣言を必ず最上階層に記述しておかなければならない。

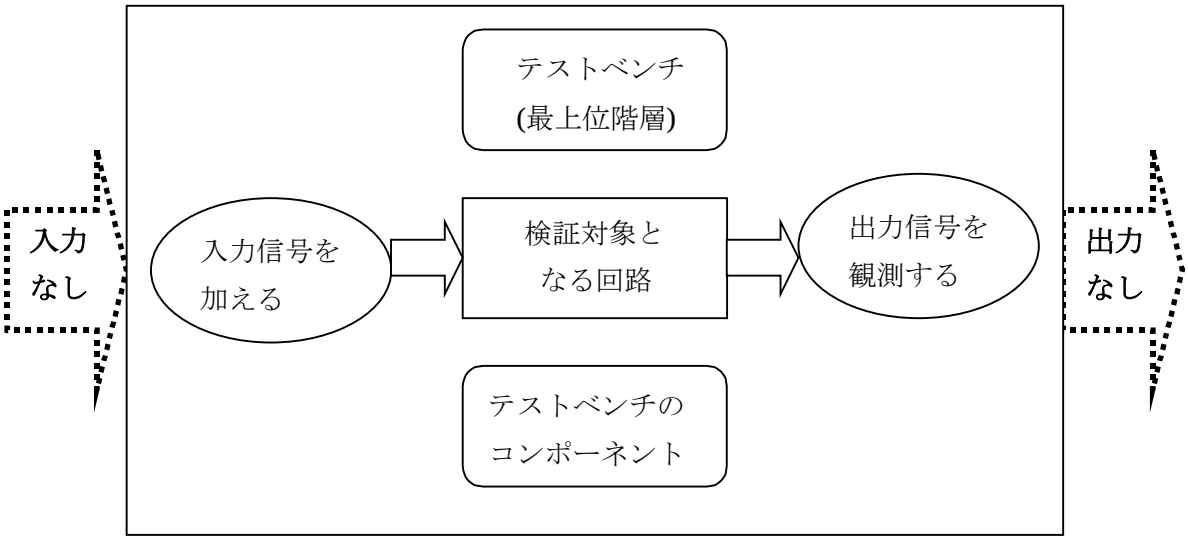


図 5.　テストベンチの階層

2.14　VHDL によるテストベンチの記述方法

テストベンチにはいくつかの記述方法があるが、最も直接的な方法としては、検証対象となる回路に印加する信号の値と時間を信号線ごとに指定するという方法である。他の記述方法としてはデータファイルを用いた記述やプログラム的な記述の仕方がある。

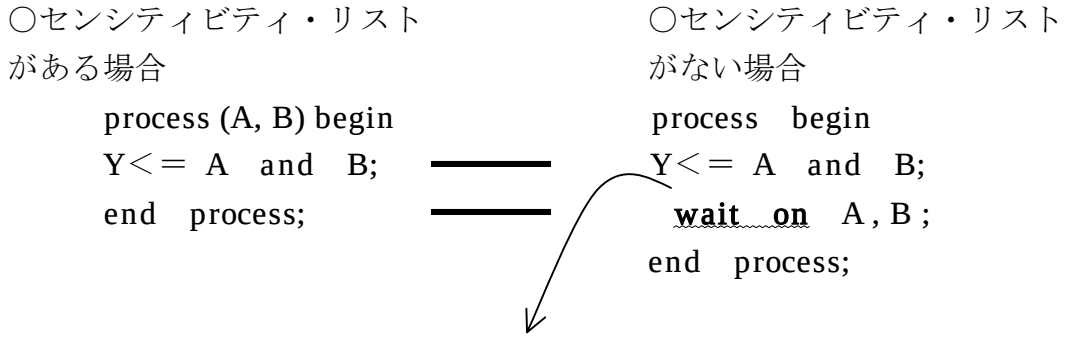
データファイルを用いた記述は、C 言語などのプログラミング言語で、テスト・ベクタを生成するプログラムを作成するなどして、データファイル（テキストファイル）を準備しておく。そのテキストをアクセスする為に TEXTIO パッケージがある。

プログラム的な記述は、VHDL の構文を利用する方法である。VHDL の if 文、case 文、loop 文などの構文を用いてプログラム的な記述をすることにより、シミュレーションを行うことも可能である。

2.15　シミュレーション記述の基本的な文法

シミュレーションの記述は最上位の階層に記述して、そこからシミュレーションしたいエンティティを呼び出す。エンティティを呼び出すが、回路の記述とは違って、エンティティ宣言の中のポートには記述しなくてもよいので何も書かずに空の状態にする。次にシミュレーションさせるものをコンポーネント宣言をして、インスタンス文で呼び出す。クロック信号はプロセス文によって生成される。プロセス文は、end　process まで行くと初めの process　begin に戻る。この記述は、パルスを永遠に出し続けるという繰り返し信号の特性を生かして記述する。

テストベンチ記述で回路の動作を発生させたり、次の動作を待たせたりするために wait 文を使用する。使用するのに注意しなければならないのは、プロセス文にセンシティビティ・リストがある場合、プロセス文の中で wait 文を使用することはできないことである。その例は以下に示す。




```
wait　on 信号, 信号･･･
wait　until 条件
wait　for 時間
wait
```

図 6.　　wait 文の種類

```
library  IEEE;
use  IEEE.std_logic_1164.all;
                                        テストベンチのエンティティ宣言は記述す
                                        る必要がないから空
entity  TEST_BENCH_HA  is
end   TEST_BENCH__HA;

architecture  SIM_DATA  of  TEST_BENCH_HA  is
                                                  コンポーネント宣言
component  HALF_ADDER
     port  ( A, B : in    std_logic;
             S, C : out   std_logic );
end  component;


signal  SA, SB, SS, SC   : std_logic;

begin
      M1 : HALF_ADDER  port  map ( SA, SB, SS, SC );
                                    センシティビティ・リストなし
process begin
      SA  <=  '0' ; wait  for  50 ns;  ──────►  SA に 0 を入れて、50ns
      SA  <=  '1' ; wait  for  50 ns;           待てという意味
end  process;

process  begin
      SB  <=  '0' ; wait  for  100 ns;
      SB  <=  '1' ; wait  for  100 ns;
end  process;

end  SIM_DATA;
```

図 7.　テストベンチの記述例

2.16　テストベンチ記述の注意点

RTL 記述には決まりがあり、その通りに記述しないと回路が生成できない。それに対してシミュレーション記述は、VHDL の文法が合っていればどんな記述をしてもよい。if 文や for-loop 文を利用すれば複雑なパターンを作成できる。

ただし、シミュレーション記述では、時間に注意を払わなければならない。そこを注意しないと、その時々によって結果がことなってしまう。

RTL 設計では、RTL での記述でのシミュレーションに加え、ロジック回路生成後、ロジックゲートでのシミュレーションが必要である。

ロジック･ゲート回路でのシミュレーションの場合、RTL でのシミュレーションで使用したテスト・パターンをそのまま利用した方が簡単である。

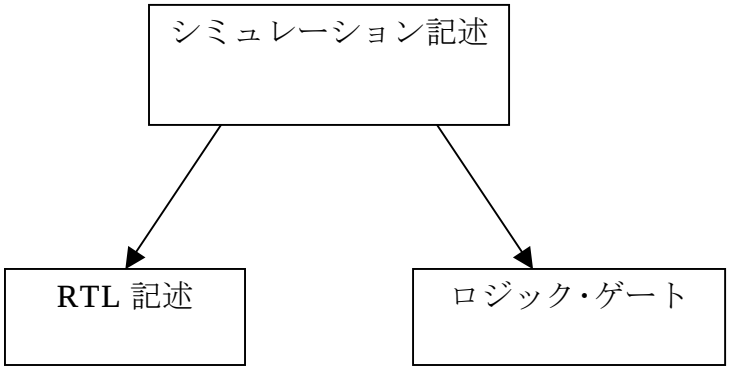


図 8.　同じシミュレーション記述を使用する

第 3 章　ディジタル画像の表現方法

3. ディジタル画像の表現

3.1　画像の分類

画像には色々な種類があり、その表現方法は異なる。どのようなものがあるのかを以下に分類する。

○白黒画像･･･明暗の情報を持っている画像のこと。
○ カラー画像･･･R(red)、G(green)、B(blue)という 3 原色で表される画像のこと。
○ 静止画像･･･時間的に変化しない画像のこと。
○ 動画像･･･テレビなどのように時間的に変化する画像のこと。

3.2　ディジタル画像の表現方法

3.2.1　次元について

ディスプレイに映し出される絵や図などは 2 次元で表されている。2 次元というのはコンピュータ等に与える情報が(x, y)座標であるかどうかということである。静止画像の場合、場所 x，y の関数 f (x，y)としてされ、各場所の明るさの情報を持つ。動画像の場合、2 次元のものが時間 t と共に変化するものであり、f (x，y，t)で表される。

3.2.2　画素（ピクセル）

コンピュータや液晶ディスプレイなどはピクセルというとても小さい四角形の集まりとして絵や文字を表現している。１つひとつのピクセルは１つの色しか持っていない。だが、そのピクセルをとても小さくすると人間の目には絵や文字が描かれているように見える。１つの絵に見えるようにするには、膨大な数のピクセルが必要である。

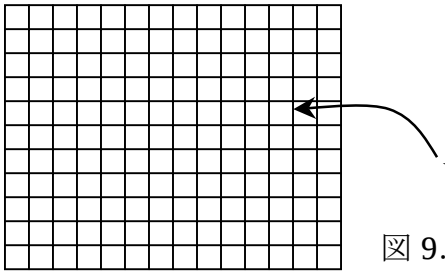


図 9.　　ピクセル

3.2.3　色の表現

ピクセルの色は RGB の三原色を用いて、3 つの色を組み合わせて様々な色を表現している。１画素に何色表示できるかは，１画素に割り当てられている メモリのビット数とディスプレイ装置によって決まる。例えば、二進数で表してみると 00000000 が一番暗くて、11111111 が一番明るい色であり、その他は 00000000～11111111 の間で色々と組み合わされて表現されている。例えば赤＋青＝紫､赤＋緑＝黄色となるように 3 色を混ぜ合わせて多数の色を作っている。

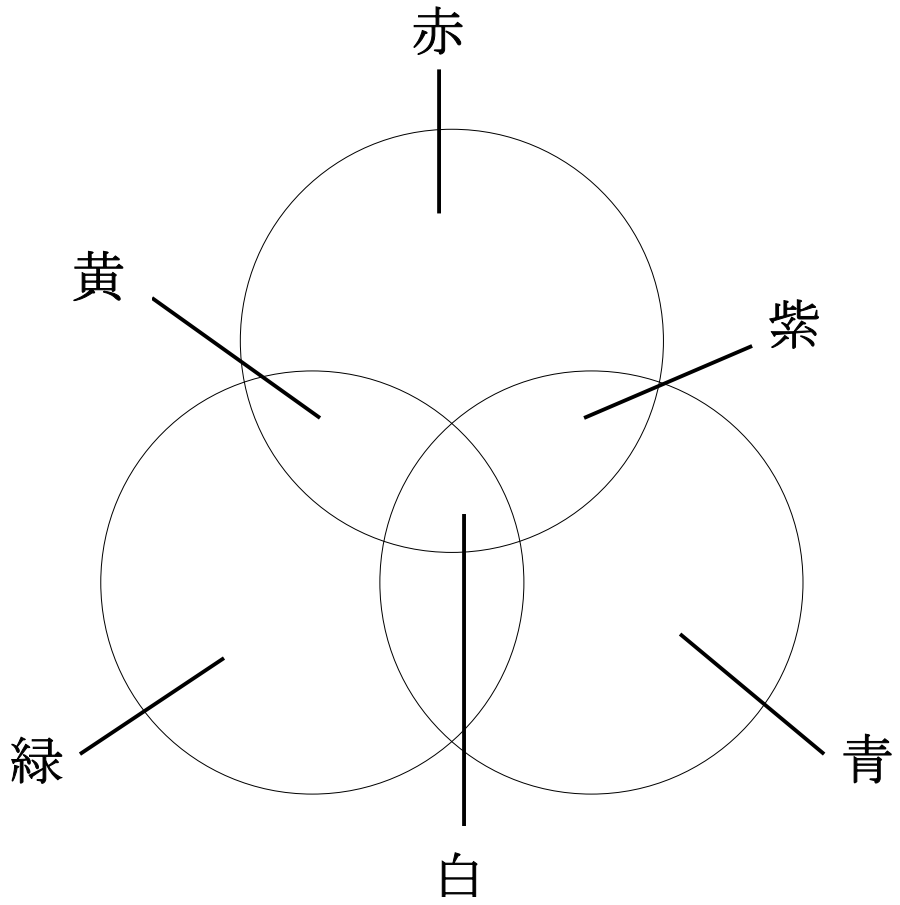


図 10. 光の三原色

3.2.4　解像度

解像度とは、画像が表現できる細かさの程度を示すものである。表示されている画面が、水平方向に何ドット、垂直方向に何ラインで構成されているかを表す。コンピュータ・テレビなどの画像表示する物のディスプレイ画面は、ピクセルの数が多ければ多いほど絵がなめらかに、きれいに見える。これを高解像度画像という。逆に、ピクセルの数が少ないほど荒くザラザラしたように見える。これを低解像度画像という。このように絵を表現するにはピクセルをどれくらいの数を使うのかで、きれいに見えるか見えないかが決まる。例えば、640×480 の解像度では、30 万 7200 個のピクセルが集まっている。

映画はテレビなどの何倍ものピクセルが必要である。ピクセルが多いときれいに見えるが、それだけ計算時間がかかり、大容量のメモリやハードディスクが必要になるという欠点もある。

3.2.5　画面表示の表し方

液晶画面に絵や文字を表示させて、それの形を移動・変化させたりするには、以下の図のように画面の横方向（320 ピクセル）を X 座標、縦方向（240 ピクセル）を Y 座標として考える。

○ 表示させたい箇所の座標を決める。

○ それが座標のアドレスとなり、その指定した場所（アドレス）の画素が点灯して表示される。

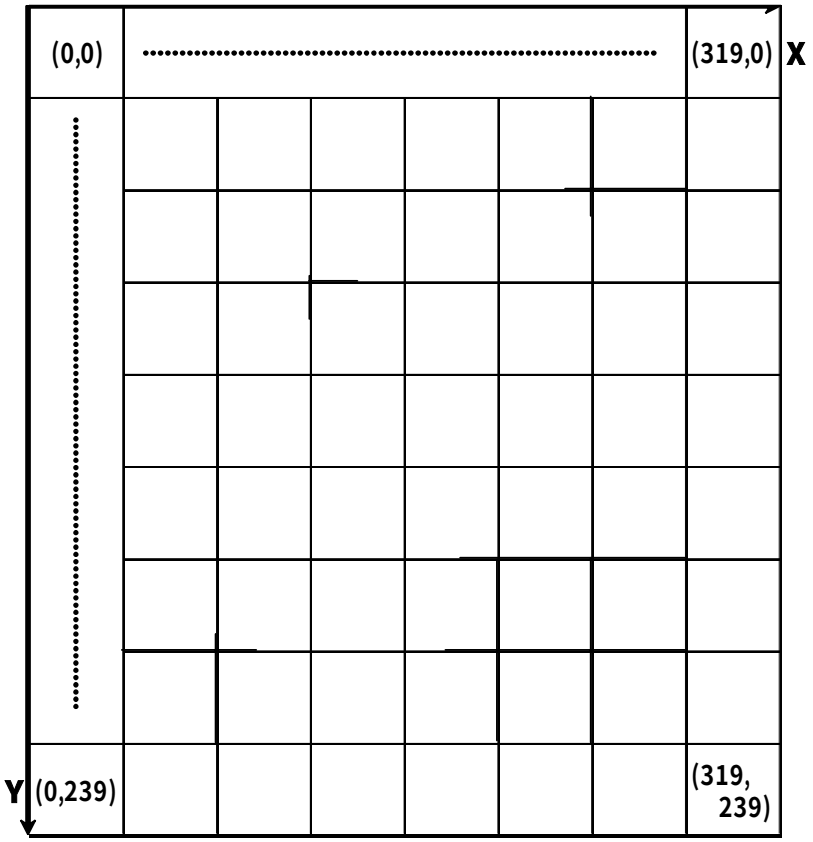


図11.　画面表示の表し方

3.3　表示するための画像について

画像表示の構成としては、計算機本体と、画像を表示するグラフィックディスプレイから成っている。表示するための画像はメモリに記憶され、メモリの内容が読み出されディスプレイに表示される。

3.4　ディスプレイについて

ディスプレイは図形表示コマンドをディスプレイプロセッサによって走査変換し、スクリーン上の画素に対応した記憶装置に記憶される。このデータはディスプレイコントローラによって画面に表示される。現在は CRT（陰極線管）だけでなく、液晶パネルなどのように薄型の物も出現して使われるようになっている。

第 4 章　ディスプレイを表示させるためのカウンタの設計

4.1　カウンタの役割

クロック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順序回路をカウンタという。カウンタは簡単に言えば数を数える回路であるが、タイマ回路や分周回路、計算機の制御、モータ回転数の計測、時計など幅広く用いられる回路でもあり、また記憶する回路でもある。同期式と非同期式がある。

カウンタの動作を表現する手段としては状態遷移表と状態遷移図がある。これは現在の状態が与えられた時に、入力が加えられると次の状態がどのように変わり、出力として何が出るかを示すものである。

4.2　カウンタの設計手順

（1）カウンタの回路への入出力信号を考え、ブロック図を作成する。
（2）カウンタがどのような状態（ステート）であるかを決め、各状態に信号を割り当てる。
（3）状態遷移図を書く。
（4）状態遷移図をもとにして、VHDL で設計していく。

4.3　ステートマシン

ステートマシンとは出力が現在の入力だけでは決まらずに、過去の出力と現在の状態と順序によって決まり、順序回路をコントロールする複雑な回路のことをいう。代表的なものでいうとフリップフロップがある。ステートマシンには 2 種類ある。

○ ミーリ型ステートマシン
出力信号を内部状態と入力信号によって生成される。回路の構成は簡単だが入力ノイズが回路に直接影響する。

○ ムーア型ステートマシン
出力信号は内部状態のみから生成される。入力信号の出力信号への影響は直接的にはない。

今回設計したカウンタ回路はムーア型ステートマシンである

4.4　h_counter と v_counter の設計

4.4.1　実験手順

320×240 の液晶ディスプレイに絵を表示させ、動作するようにするにはまず水平方向 h_counter と垂直方向 v_counter を設計しなければならない。設計手順を以下に示す。

①どのような仕組みで、ディスプレイに絵が映し出されているかを簡単な図を書く。

②仕様書にある水平方向のタイミング図を見ながら表示する部分としない部分とに分ける。

③②で各々の状態がどうやって動作をしてカウントしているかの状態遷移図を書いていく。

④1 列目のカウントが終わったら 2 列目にカウントされるようにイネーブル信号を考える。

⑤h_counter と v_counter にクロック、リセット、h_sync、v_sync などの入力・出力がどのようになっているかの図を書く。

⑥以上のことをふまえ、それぞれの状態の動作を考えながら VHDL で水平方向 h_counter と垂直方向 v_counter の回路を設計する。h_counter は常に水平方向にカウントされているのに対し、v_counter はイネーブル信号が‘1’でクロックが入った時だけカウントアップするものである。

⑦回路を記述してコンパイルが成功したら、それが正しく動くかどうかを確認するために VHDL でテストベンチを記述する。

⑧テストベンチを記述し、それもコンパイルが成功したら、シミュレーションして動作を確認し、波形がおかしかったらモジュール・テストを見直す。

4.4.2　ディスプレイへの入力と出力

図 1 入力と出力がどのようにディスプレイに表示されるかを示したものである。h_sync（水平同期信号)、v_sync（垂直同期信号)、クロックがディスプレイに直接入力され、どこに何を映し出すかを表すアドレスは一度メモリに記憶されてからディスプレイに表示される。

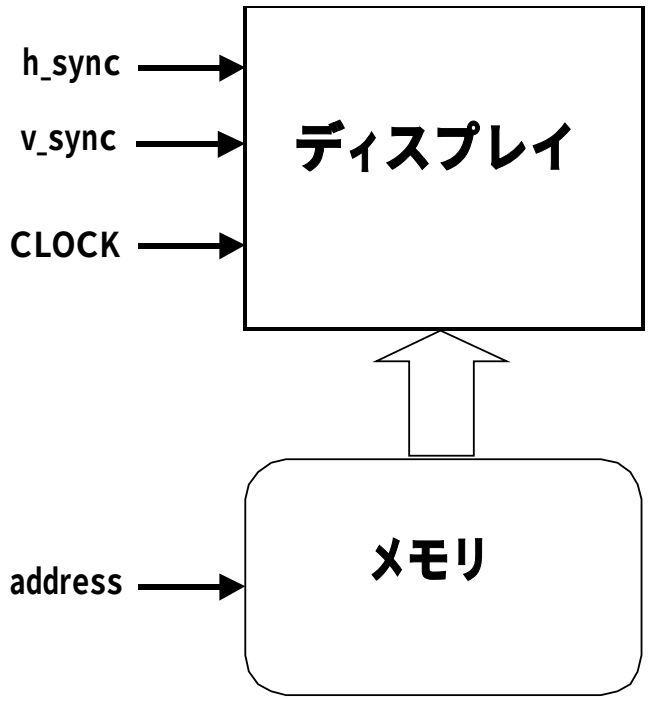
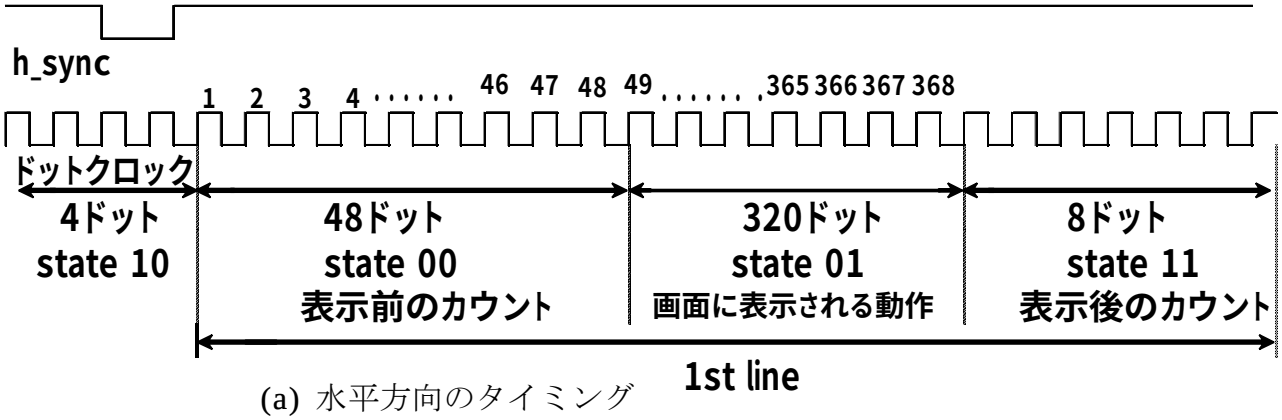


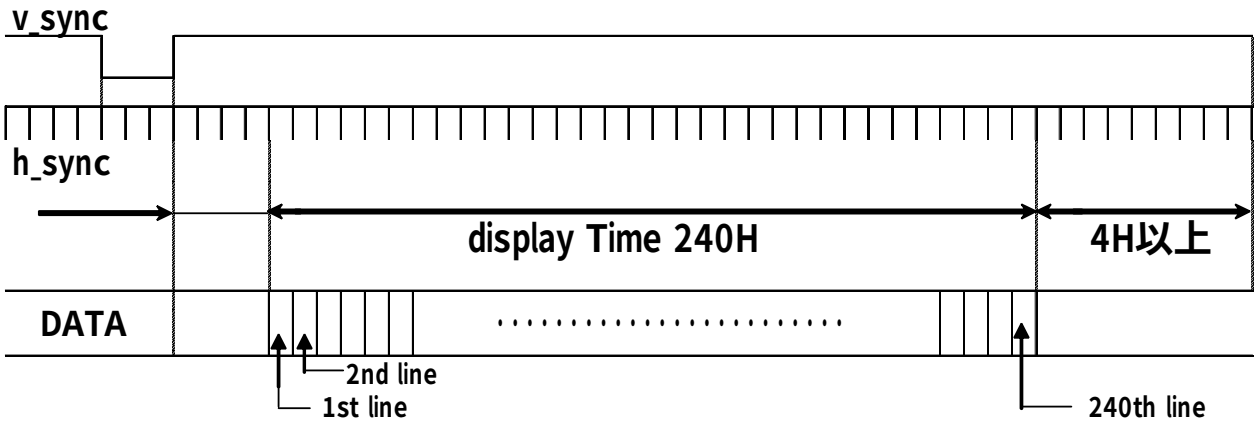
図 12.　ディスプレイへの入力と出力

4.4.3　水平方向と垂直方向のカウンタのタイミング図と動作の説明

図2は320×240の液晶ディスプレイの水平方向と垂直方向のデータタイミング図である。 図(a)は水平方向のデータ入力の動作を表したものである。いくつかの状態に分けおり、それぞれのカウントする値は違うが全体を見ると一つのカウンタ中での動作である。48 ドット（state00）の状態の部分は表示する前のカウントで、8 ドット（state11）と 4 ドット（state10）の状態の部分は表示した後のカウントである。これらの状態は表示の前と後のものであり、だだの数える部分であるから、絵は表示されない。絵を表示する部分は 320 ドット（state10）の状態のときである。この動作が 1 列で行われる。1 列目が終わると、2 列目・3 列目…と順番に同じことを繰り返しながら、240 列目まで行われる。



(a) 水平方向のタイミング



(b)垂直方向のタイミング

図 13.　データタイミング図

4.4.4　状態遷移図

4.4.3 のデータタイミング図を元に、４つの状態に分けて、h_counter の内部がどのように遷移しているか調べて、状態遷移図にしたものである。

v_counter についても同様な状態遷移図により行アドレスがもとめられる。

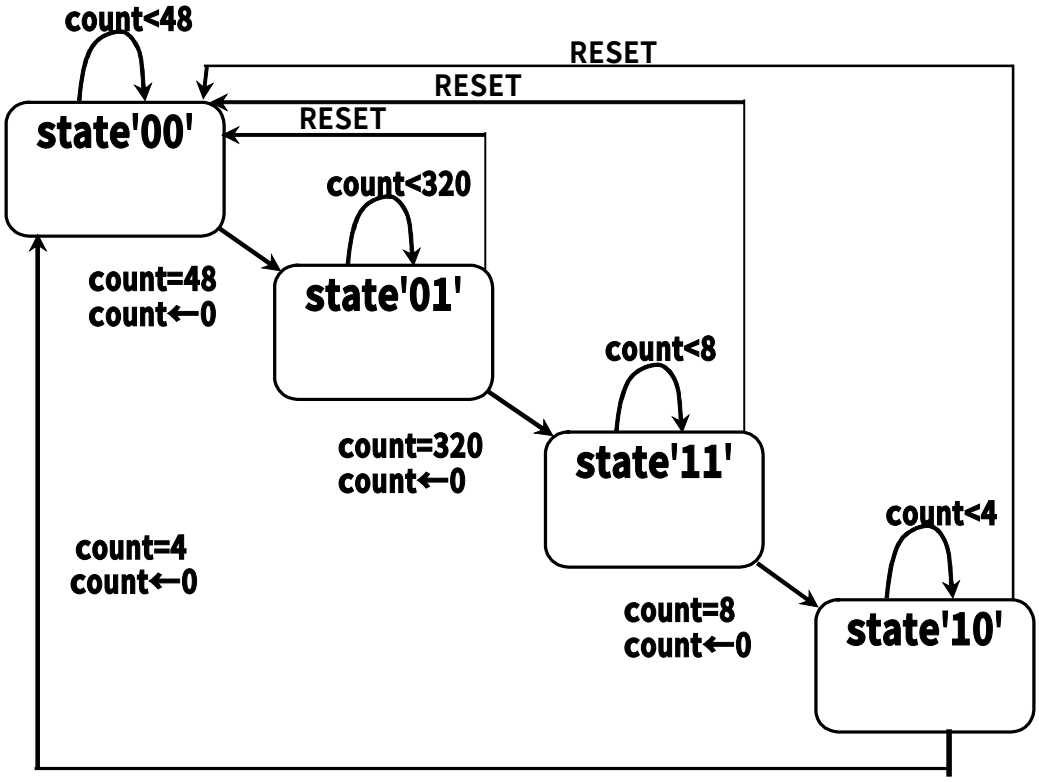


図14.　h_counterの状態遷移図

4.4.5　画面上でのデータの動作

図 15 は図 13 のデータが画面上でどのように動作しているかを表したものである。水平方向のカウンタは常にずっとカウントしている。(1,1)から始まり、(1,320)が終わる。先にも書いたように、(1,320)までカウントされたら、イネーブル信号によって‘1’上がり次の列になって(2,1)から同じ動作を繰り返していく。垂直方向のカウンタは 320 進カウンタのイネーブル信号(出力)が入力信号となって 1 回ずつカウントするだけである。水平方向のように常にカウントしない。

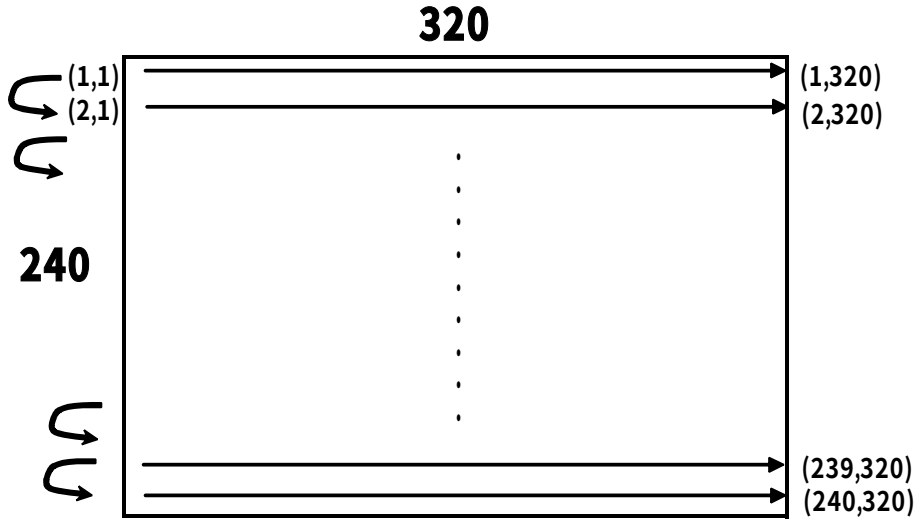


図 15.　画面上でのタイミングの動作

4.4.6　カウンタの入力信号と出力信号

図 16 は各々のカウンタの入力信号と出力信号を表したものである。それぞれの入出力信号の意味は以下に示す。

H_COUNTER
h_address･･･横方向（X 座標）に表示される場所
h_sync･･･ディスプレイの水平同期信号
h_grant･･･電圧が high の時カウントの値が画面の水平方向のアドレスになる信号
V_COUNTER
v_address･･･縦方向（Y 座標）に表示される場所
v_sync･･･ディスプレイの垂直同期信号
v_grant･･･電圧が high の時カウントの値が画面の垂直方向のアドレスになるような信号
v_en…h_counter からの桁上げ信号（イネーブル信号）
h_counter の出力信号で、その出力が v_counter の入力信号となる

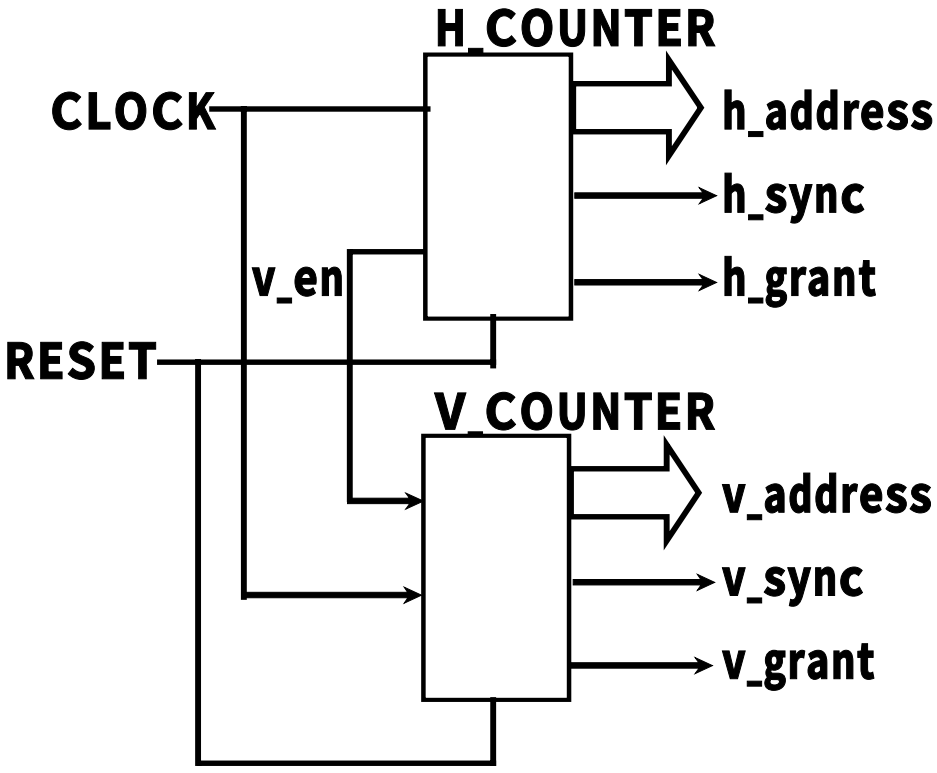


図 16.　カウンタの入力信号と出力信号

VHDL で記述したモジュールとテストベンチのソース、シミュレーション波形は付録として本論文の最後に載せてある。

第 5 章　おわりに

今回の卒業研究で、VHDL についてとディジタル画像の表現方法を学習してきた。実験としては、VHDL でディスプレイに絵を表示させるためのカウンタのモジュールとテストベンチを記述し、シミュレーションをした。設計した二つのカウンタは記述通りの波形が出た。目標としていたディスプレイに絵を表示させることをするまでには至らず、表示させるためのカウンタの設計・シミュレーションまでで終わってしまった。

VHDL に中々馴染めず、カウンタを設計するのにも苦労したが、記述したモジュールなどがコンパイル成功し、シミュレーションをして設計した回路が正しく動作している波形が出た時は達成感があった。諦めずに最後まで卒業研究をしたことで自分に自信がついた。

謝辞

今回の卒業研究に際して、丁寧なご助言、ご鞭撻して下さりました橘　昌良助教授に心から感謝致します。

研究中、親切丁寧なご助言下さった、原　央教授ならびに矢野　政顕教授に厚くお礼申し上げます。

また、同じ研究室において、たくさんの助言をしてくださった大学院生の先輩方、4 年生の皆様にお礼申し上げます。

参考文献

[1] VHDL によるハードウェア設計入門　長谷川裕恭　CQ 出版社
[2] Design Wave　magazine　2000 年 12 月号　CQ 出版社
[3] ディジタル信号処理　貴家仁志　昭晃堂
[4] コンピュータアーキテクチャの基礎　柴山潔　近代科学社
[5] HDL による VLSI 設計　深山正幸・北川彰夫・秋田純一・鈴木正國　共立出版
[6] VHDL によるディジタル回路入門　並木秀明・永井亘道　技術評論社
[7] コンピュータグラフィックスの基礎知識　塩川厚　ohmsha
[8] ディスプレイの仕様書　5.5 インチパッシブカラー液晶モジュール　LM32C041

付録

h_counter　モジュール

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

entity v_counter is
        port (CLK,RESET,v_en : in std_logic;
              count : out std_logic_vector(7 downto 0);
         v_grant : out std_logic;
         v_sync : out std_logic
         );
end v_counter;

architecture RTL of v_counter is

signal count_in : std_logic_vector(7 downto 0);
signal state,next_state : std_logic_vector(1 downto 0);

begin
      count <= count_in;
      state <= next_state;

    process(CLK,RESET,state) begin
       if(RESET='1') then
       count_in<="00000000";
          elsif(CLK'event and CLK='1') then
        if(v_en='1') then

        if (state ="00") then
              if(count_in = "00000100") then
                    count_in<="00000000";
                       next_state <= "01";
                 else
                    count_in<=count_in + '1';
```

```
            end if;
         elsif (state="01") then
           if(count_in = "11101111") then
                 count_in<="00000000";
                    next_state<="11";
              else
                 count_in<=count_in + '1';
            end if;

          elsif (state="11") then
           if(count_in = "00000111") then
                 count_in <= "00000000";
                    next_state<="10";
              else
                 count_in<=count_in + '1';
            end if;

          else
           if (count_in = "00000011") then
                 count_in <= "00000000";
                  next_state <="00";
              else
                 count_in<=count_in + '1';
            end if;

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


process(CLK,RESET,state) begin

     if (state="00") then
         v_grant<='0';
        v_sync<='1';
```

```
            elsif (state="01") then
               v_grant<='1';
               v_sync<='1';

                elsif (state="11") then
               v_grant<='0';
               v_sync<='1';

         else
               v_grant<='0';
               v_sync<='0';

           end if;

      end process;

end RTL;
```

h_counter テストベンチ

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

entity TEST_h_counter is
end TEST_h_counter;

architecture stimulus of TEST_h_counter  is

component h_counter is
   port (CLK,RESET : in std_logic;
        COUNT : out std_logic_vector (8 downto 0);
        h_sync : out std_logic_vector;
        v_en : out std_logic;
        h_grant : out std_logic

        );

end component;

signal CLK,RESET,h_sync,v_en,h_grant : std_logic;
signal COUNT : std_logic_vector (8 downto 0);
--signal state,next_state : std_logic_vector(1 downto 0);

begin
   U0 : h_counter port map (CLK=>CLK
    ,RESET=>RESET,count=>count,
                        h_sync=>h_sync,v_en=>v_en,
                        h_grant=>h_grant
                        );

process begin
  CLK <= '1';
  wait for 10ns;
```

```
  CLK <= '0';
  wait for 10ns;
end process;

process begin

RESET <='1';
wait for 50ns;
RESET <='0';
wait for 50ns;
wait;

end process;

end stimulus;
```

v_counter モジュール

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

entity v_counter is
        port (CLK,RESET,v_en : in std_logic;
              count : out std_logic_vector(7 downto 0);
            v_grant : out std_logic;
            v_sync : out std_logic
            );
end v_counter;

architecture RTL of v_counter is

signal count_in : std_logic_vector(7 downto 0);
signal state,next_state : std_logic_vector(1 downto 0);

begin
      count <= count_in;
      state <= next_state;

    process(CLK,RESET,state) begin
         if(RESET='1') then
       count_in<="00000000";
              elsif(CLK'event and CLK='1') then
        if(v_en='1') then

        if (state ="00") then
                    if(count_in = "00000100") then
                            count_in<="00000000";
                                   next_state <= "01";
                        else
                           count_in<=count_in + '1';
                       end if;
```

```
                elsif (state="01") then
                  if(count_in = "11101111") then
                          count_in<="00000000";
                                 next_state<="11";
                       else
                          count_in<=count_in + '1';
                      end if;

                elsif (state="11") then
                  if(count_in = "00000111") then
                          count_in <= "00000000";
                                 next_state<="10";
                       else
                          count_in<=count_in + '1';
                      end if;

                else
                  if (count_in = "00000011") then
                          count_in <= "00000000";
                             next_state <="00";
                       else
                          count_in<=count_in + '1';
                      end if;

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


process(CLK,RESET,state) begin

       if (state="00") then
             v_grant<='0';
        v_sync<='1';
```

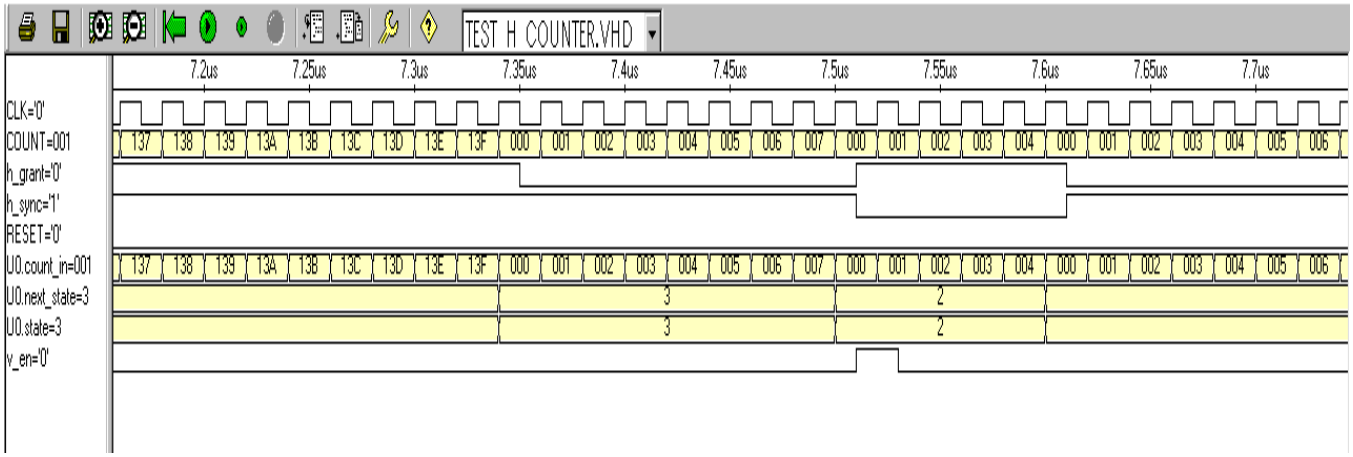
```
            elsif (state="01") then
                 v_grant<='1';
                 v_sync<='1';

            elsif (state="11") then
                 v_grant<='0';
                 v_sync<='1';

            else
                 v_grant<='0';
                 v_sync<='0';

           end if;

      end process;

end RTL;
```

v_counter テストベンチ

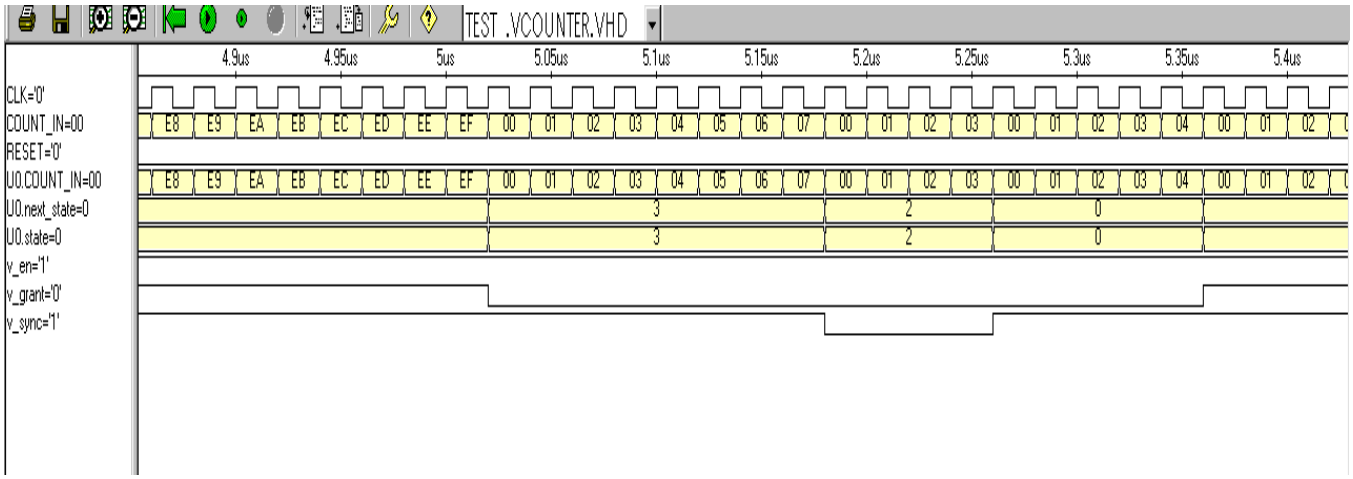
```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity TEST_v_counter is
end TEST_v_counter;

architecture stimulus of TEST_v_counter  is

component v_counter is
   port (CLK,RESET,v_en : in std_logic;
        COUNT : out std_logic_vector (7 downto 0);
        v_grant : out std_logic;
        v_sync : out std_logic
        );

end component;

constant CLK_CYCLE : Time :=100ns;

signal CLK,RESET,v_sync,v_en,v_grant : std_logic;
signal COUNT_IN :std_logic_vector (7 downto 0);
--signal state,next_state : std_logic_vector(1 downto 0);

begin
   U0 : v_counter port map (CLK=>CLK
    ,RESET=>RESET,COUNT=>COUNT_IN,

v_en=>v_en,v_grant=>v_grant,v_sync=>v_sync

                        );

process begin
  CLK <= '1';
```

```
  wait for 10ns;
  CLK <= '0';
  wait for 10ns;
end process;

process begin
RESET <= '1';
wait for 50ns;
RESET <= '0';
v_en <='1';
wait for 50ns;

wait ;

end process;

end stimulus;
```

シミュレーション結果



① h_counter のシミュレーション波形



② v_counter のシミュレーション波形

今回の研究で参考にした液晶モジュール

○5.5 インチパッシブカラー液晶モジュール LM32CO41

○垂直同期信号、水平同期信号、ドットクロック、RGB 各 3 ビットの表示データを入力する事によって、RGB 各 8 階調、512 色を表示可能

写真１　液晶モジュールの表

写真 2　液晶モジュールの裏