# ディジタル回路設計

## 1　ディジタル回路設計の概要

半導体集積回路技術の進歩により、計算機や携帯電話、さらには自動車や家電などあらゆる分野で大規模半導体集積回路 (LSI; Large Scale Integrated Circuit) が使用されている。LSI の規模と動作クロックは年々大きくなっており、これらの LSI の高集積化と高速化にともなって回路の設計はますます複雑で困難なものになってきている。一方で製品の市場サイクルは短くなっており、可能なかぎり製品の開発期間を短くしなければならなくなっている。このような設計の大規模化と設計期間の短縮を同時に達成するには計算機による設計の自動化が必須となっている。

計算機による LSI 設計の自動化は 1970 年代の半導体マスク設計から始まり、1980 年代には論理ゲートを用いた回路図入力とゲートレベルシミュレーションによる設計が行われるようになった。しかし、数万ゲートを超える大規模回路では、ゲートレベルシミュレータを用いても、要求仕様を満足する回路を短期間で開発することが難しくなってきている。そこで、従来のゲートレベル設計に対する次世代設計技術として、ハードウェア記述言語 HDL と論理合成ツールを用いた設計手法が実用化されてきた。

### 1.1　目的

- 電子系 CAD(Computer Aided Design) での基本素子と基本回路の理解
- 簡単な回路設計を通しての、CAD/EDA(Electronic Design Automation) システムによる設計方法の理解
- ハードウェア記述言語 HDL(Hardware Description Language) での回路設計
- FPGA(Field Programmable Gate Array) での動作設計

### 1.2　実験で取り扱うディジタル回路設計の流れ

図 1 に、論理合成と HDL を用いた場合の通常の ASIC 開発フローを示す。

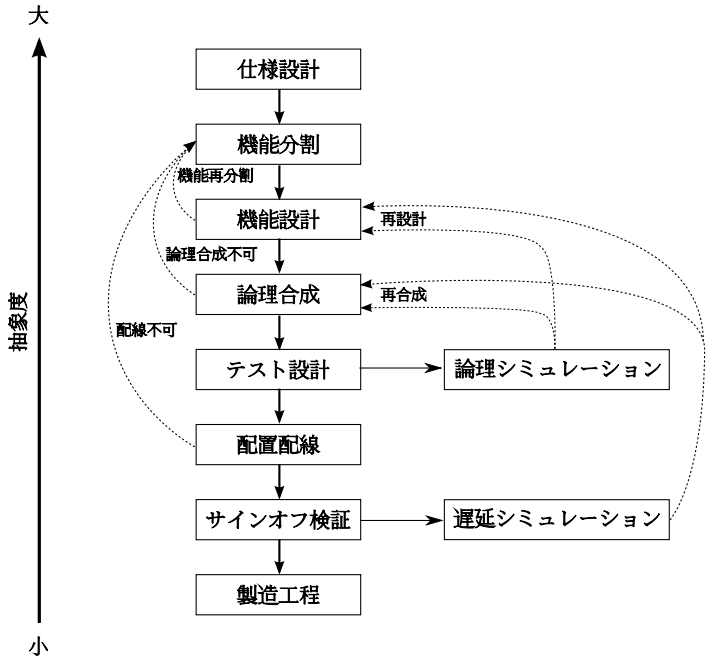



図 1: ディジタル回路設計の流れ

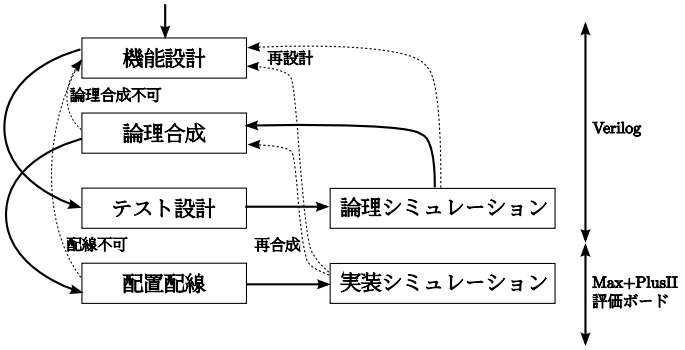



図 2: 本実験でのディジタル回路設計の流れ

本実験では、仕様設計 (システムの特性や機能について検討する) と機能分割 (どのようなハードウェア、ソフトウェアを使用するかを検討する) はすでに終わっているものとし、図 2 のような流れで開発を行う。実機によるシミュレーションを行わない場合には論理シミュレーションまでを行う。

**機能設計**

ASIC の内部機能をさらに HDL を用いて詳細に定義する。HDL の記述レベルには、論理レベル、RTL(Register Transfer Level)、ビヘイビアレベル (Behaviour Level) などがある。記述した HDL は論理シミュレータ上でテストパターンを用いて検証する。

**論理合成**

実現するテクノロジを指定し、目標とするゲート回路の性能を設計制約条件とし、設計制約条件を満たすゲート回路を自動的に合成する。合成した回路は、タイミング解析を行い、要求するタイミングを満たしていることを確認する。

**テスト設計**

デバイスの動作保証をするためには故障検出率の高い回路を作成する必要がある。そのため、デバイスのテストを容易にするためのテスト容易化設計を行い、さらにテストパターンの自動生成を行う。

**配置配線 (レイアウト設計)**

チップを製造するための必要なマスクを作成するため、ゲート回路のネットリストに従って、セルの自動配置と自動配線を行う。

## 1.3 使用するツールについて

**EDA(Electronic Design Automation)**

Verilog のフリーの実装である Icarus Verilog を用いる。Icarus Verilog はコマンドラインで実行するツールであり、機能設計、機能シミュレーション、論理合成、論理シミュレーションを行うことができる。Icarus Verilog でゲートの接続情報 (ネットリスト, net list) を記録する EDIF(Electronic Design Interchange Format) ファイル (フォーマットがほぼ共通化されているおり、多数のベンダのソフトウェア同士でデータをやり取りすることができる) を出力し、Altera 社の MAX+PlusII で配置配線を行う。

**FPGA**

実装シミュレーションには FPGA を利用する。評価ボードには三菱電機マイコン機器ソフトウェア社製 MU200-EA30 を利用する。このボード上にはロータリースイッチ、プッシュスイッチ、7 セグメント LED などがある。全てのピンコネクタが引きだされており、柔軟な回路を設計、評価することができる。

搭載 FPGA は Altera 社の FLEX10K シリーズの EPF10K30RC208-4(3 万ゲート) である。FLEX 10K シリーズでは SRAM ベースの揮発性メモリで配置配線を記憶している。電源を落としてしまうと配線デー

タを失うが、ホスト計算機からダウンロードして何回でも書き換えが可能である[1]。

**拡張子について**

- **.edf**　ネットリストファイル (信号線の接続関係を記述)
- **.gdf**　回路図ファイル
- **.sym**　シンボルファイル (外部回路の形状、信号線の表示)
- **.v**　Verilog ファイル
- **.vcd**　波形ファイル

# 2　Verilog によるサンプルプロジェクトの作成

本章ではサンプル・プロジェクトを作成し、その実行を通して Verilog における開発の概要を簡単に説明します。Verilog での作業は次のようになります。

1. 作成する回路の仕様を設計
2. 1 を実現する Verilog ソースを作成
3. 2 をテストする Verilog のテストコードを作成
4. コンパイルしてテスト

## 2.1　仕様設計

今回作成するサンプルプロジェクトでは 1-bit コンパレータを作成します。これは図 3 のように 2 入力 1 出力であり、表 1 のような機能を持っています。

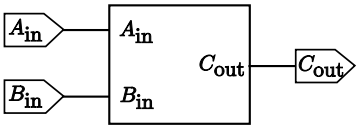



図 3: 1-bit コンパレータ

表 1: 1-bit コンパレータの機能

| | |
|---|---|
| **入力信号** | $Ain$, $Bin$ |
| **出力信号** | $Cout$ |
| **機　能** | $Ain = Bin$ のとき $Cout$ が “1”、それ以外のときは “0” |

## 2.2　ソースファイルの作成

エディタで次のような内容のファイル compare.v を作成してください。

```
module compare(a, b, c);
   input a;
   input b;
   output c;
   assign c = ~ (a ^ b);
endmodule
```

モジュールは Verilog の最小単位であり、キーワード module で始まります。compare がモジュールの名前で、括弧内がポートリスト (端子リスト) となっています。ポートの入出力はモジュール内で指定されています。2 行目から最後の endmodule までがモジュールの中身であり、この中ではパラメータ宣言、ポート宣言、レジスタ宣言、イベント宣言、ネット宣言、ステートメントなどを記述します。

表 1 で示したように、1-bit コンパレータは 2 入力と 1 出力となっています。そのため、ポート宣言 (2〜4 行目) では a と b が入力、c が出力となっています。また、1-bit コンパレータは $Ain = Bin$ のとき $Cout$ が “1” で、それ以外のときは “0” なので、$Ain$ と $Bin$ の排他的論理和 (^) をとり、それを否定 (~) すればよいことがわかります。

[1]従来のプログラマブルロジックアレイは、マスク ROM、ヒューズ溶解によるライトワンスなチップや EEPROM(Electronically Erasable Programmable Read Only Memory) などの不揮発性メモリを利用したものである。EEPROM は書き換え可能だが、実際には紫外線による素子劣化のため頻繁に書き換えることはできない。

## 2.3 テストファイルの作成

先ほど作成した、compare をテストするためのコードを作成します。ファイル compare.v の続きに次の内容を追加してください。

```
‘timescale 1ns/1ps
module compare_test;
   parameter STEP = 10;
   reg ina, inb;
   wire outc;

   compare comp(ina, inb, outc);        // compare をインスタンス化

   initial begin
      $dumpfile("compare_result.vcd");  // 波形データ出力先を指定
      $dumpvars(0, comp);               // 出力変数を指定
      $monitor($stime, "\ta=%b, b=%b, c=%b", ina, inb, outc);

      ina <= 1’b0;  inb <= 1’b0;  #STEP
      ina <= 1’b0;  inb <= 1’b1;  #STEP
      ina <= 1’b1;  inb <= 1’b0;  #STEP
      ina <= 1’b1;  inb <= 1’b1;  #STEP
      ina <= 1’b0;  inb <= 1’b0;  #STEP
      $finish;
   end
endmodule
```

initial は初回に 1 度だけ実行されるステートメントであり、#は待機時間を示します。この例はテストとして A と B に 0 を入力して 10 ナノ秒待ち、次に A と B にそれぞれ 0 と 1 を入力して 10 ナノ秒待つ…ということを順番に行うように指示しています。

## 2.4 モジュールのコンパイルとリンク

コマンドプロンプト上で次のようにして iverilog でコンパイルを行います。

```
iverilog -o compare -s compare_test compare.v
```

-o で出力するファイル、-s でトップレベルモジュールを指定しており、残りはコンパイルしたいソースファイルです。エラーがなければ、なにもメッセージを表示せずに compare というファイルが作成されます。

## 2.5 シミュレーション

実行にはコマンドプロンプト上で vvp を次のように実行します。

```
vvp compare
```

実行したら、次のように表示されるでしょう。

```
VCD info: dumpfile compare_result.vcd opened for output.
                   0    a=0, b=0, c=1
                  10    a=0, b=1, c=0
                  20    a=1, b=0, c=0
                  30    a=1, b=1, c=1
                  40    a=0, b=0, c=1
```

コンパレータの真理値表は表 2 のようになりますが、これと同じ結果となっていることを確認してください。

表 2: コンパレータの真理値表

| Ain | Bin | Cout |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

### 2.6 波形の表示

vvp を実行したときに、結果が表示されるとともに compare_result.vcd というファイルが作成されます。これを波形ビューワで見ることができます。これがコンパレータの真理値表と同じ結果となっていることを確認してください。

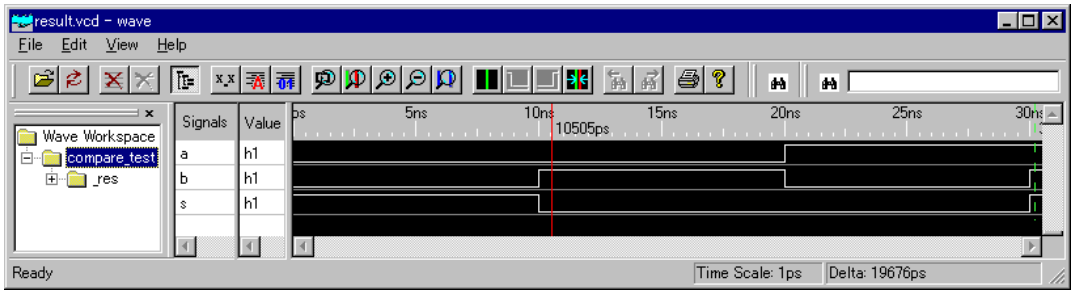



図 4: 波形ビューワ

**課題 1**

Verilog で次の基本回路を作成してシミュレーションを行いなさい。

- 2 入力 NAND
- 2 入力 NOR

## 3 簡単な回路の作成

論理回路は大きく組み合わせ論理回路 (combinational circuit) と順序論理回路 (sequential circuit) の 2 つに分けられます。

- **組み合わせ回路**　内部には状態を持たず、現在の入力だけで出力が決まるような回路です。加算回路 (adder)、符号器 (encoder)、復号器 (decoder) などがあります。
- **順序論理回路**　内部に状態を持っており、現在の入力とそれ以前の入力によって出力が決まるような回路です。フリップフロップやカウンタなどがあります。

本実験ではこれらの回路のうちで基本的な回路を Verilog で設計します。

### 3.1 組み合わせ回路

組み合わせ回路は内部には状態を持たず、現在の入力だけで出力が決まるような回路です。加算回路は組み合わせ回路のひとつであり、1 ビットの 2 進数の加算を行う半加算回路 (half adder) と、桁上がりを含めた 1 ビットの加算を行うことができる全加算回路 (full adder) があります。複数の全加算回路を用いることで整数の加算回路を作ることができます。例えば、16 個の全加算回路を用いることで 2 つの 16 ビット整数の加算回路を作成することができます。

### 半加算回路

図 5 と表 3 はそれぞれ半加算回路の回路図とその真理値表です。入力 $A$ と $B$ とが足された結果が $C$ と $S$ に出力されます。このとき、$S$ はその桁の値 (Sum) で $C$ は桁上がりした値 (Carry) と考えてください。

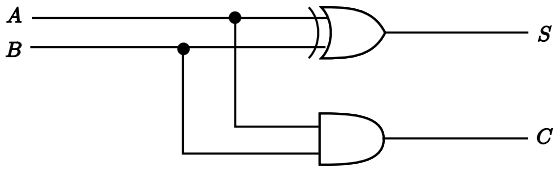



図 5: 半加算回路

表 3: 半加算回路の真理値表

| $A$ | $B$ | $C$ | $S$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

### 全加算回路

半加算回路は下の桁からの桁上がりを考慮していませんが、実際に 2 進表現された整数を加算するには下の桁上がりを考慮する必要があります。それを考慮する回路が全加算回路で、図 6 に示すように半加算回路 2 個で作ることができます[2]。全加算回路の真理値表は表 4 のようになります。

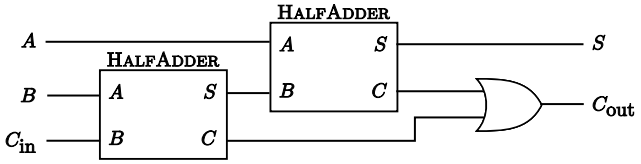



図 6: 全加算回路

表 4: 全加算回路の真理値表

| $A$ | $B$ | $C_{\text{in}}$ | $Cout$ | $S$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

### 4 ビット加算回路

全加算回路で各ビットごとの桁上がりを考慮することで整数同士の加算回路をつくることができます。4 ビット整数同士の加算回路を図 7 に示します。

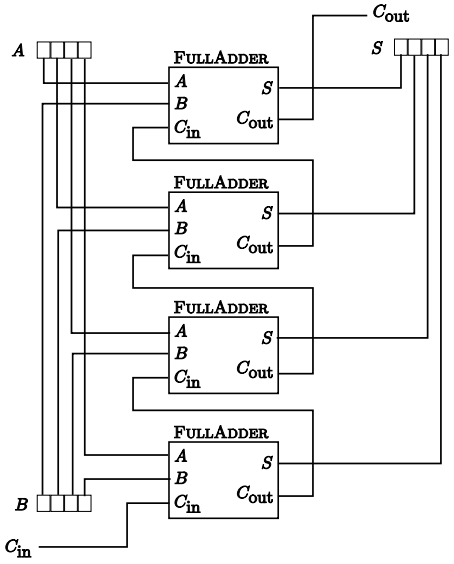



図 7: 4 ビット加算回路

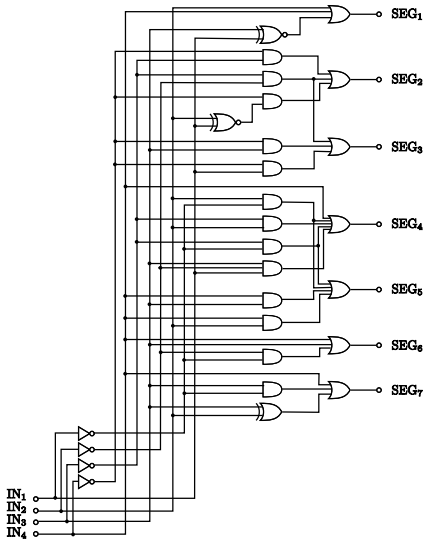



図 8: 7 セグメント LED の回路図

[2] $Cout$ の直前の OR は XOR でも構わない (両方が 1 となる組み合わせが存在しないため)。

**課題 2**

Verilog で次の回路を作成してシミュレーションを行いなさい。レポートには Verilog ソースとシミュレーション結果を添付すること。

(1) 半加算回路 (図 5 参照)

(2) 全加算回路 (図 6 参照)
(1) で作成した半加算回路を 2 つ利用すること。

(3) 4 ビット加算回路 (図 7 参照)
(2) で作成した全加算回路を 4 つ利用し、$A$, $B$, $S$ は 4 ビット幅の信号とすること。テストベンチでは意味がありそうな加算テストを 8 回行うこと (全網羅しなくてよい)

(4) 7 セグメント LED デコーダ (表 5 と図 9 参照)
入力を 4 ビット、出力を 8 ビット幅の信号とすること。論理式で書くこともできるが (図 8 を参照)、非常に複雑となるので case 文を用いよ。

表 5: 7 セグメント LED の信号線対応表

| **値** | $A$[7..0] | **値** | $A$[7..0] |
|---|---|---|---|
| 0 | 1111 1100 | 8 | 1111 1110 |
| 1 | 0110 0000 | 9 | 1111 0110 |
| 2 | 1101 1010 | A | 1110 1110 |
| 3 | 1111 0010 | B | 0011 1110 |
| 4 | 0110 0110 | C | 0001 1010 |
| 5 | 1011 0110 | D | 0111 1010 |
| 6 | 1011 1110 | E | 1001 1110 |
| 7 | 1110 0000 | F | 1000 1110 |

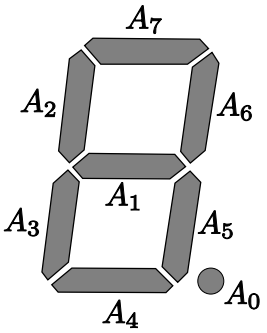


図 9: 7 セグメント LED

**課題 3**

課題 2 で作成した 4 ビット加算回路と 7 セグメント LED デコーダを利用して、図 10 の回路を Verilog で作成してシミュレーションを行いなさい。これは 4 ビットの数値を入力させ、それぞれの値とその加算結果を 7 セグメント LED に表示させるような回路である。レポートには Verilog ソースとシミュレーション結果を添付すること。

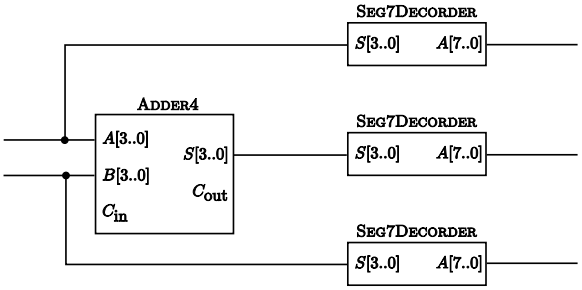


図 10: 課題 3 の回路構成概略図

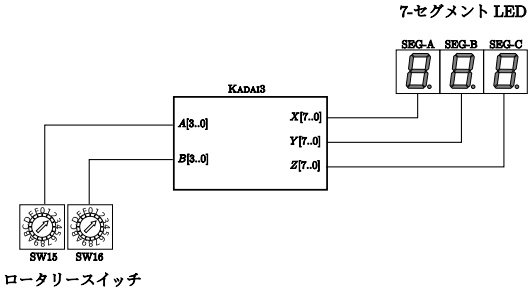


図 11: 課題 4 の回路構成概略図

**課題 4**

図 11 を参考に、課題 3 で作成した回路を評価ボード上で動かしなさい。

## 3.2 順序回路

順序回路は内部に状態を持っており、現在の入力とそれ以前の入力によって出力が決まるような回路です。順序回路のひとつであるフリップフロップ (FlipFlop) は 1 ビットの情報を一時的に保持することが

できる論理回路で、順序回路の基本要素です。組み合わせ論理回路に遅延回路 (入力に対して遅延した出力を入力側へフィードバックする) を付け加えて情報を保持します。

### D-フリップフロップ

出力 $Q$ は常にクロック立ち上がり (rising edge) 時に与えられた $D$ の値を取ります。D-フリップフロップのタイミングチャートを図 13 に示します。

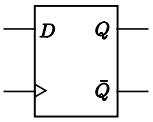



図 12: D-フリップフロップ

表 6: D-フリップフロップの真理値表

| clk | $D$ | $Q$ | $\bar{Q}$ |
|---|---|---|---|
| 立ち上がり時 | 0 | 0 | 1 |
| 立ち上がり時 | 1 | 1 | 0 |

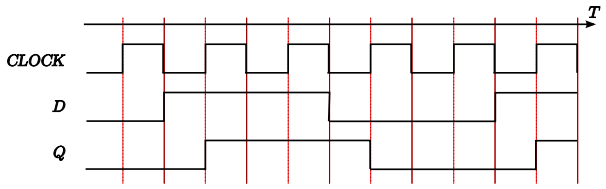



図 13: D-FF のタイミングチャート

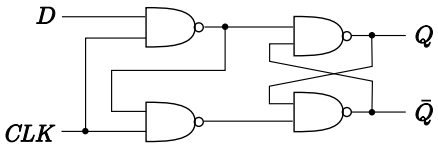



図 14: D-フリップフロップ

これを実現する回路は図 14 のようになりますが、Verilog では次に示すソースのように表 6 の内容を直感的に実現できます。

```
module DFF(clk, rst, d, q, qbar);
   input  clk;
   input  rst;
   input  d;
   output q;
   output qbar;
   reg    q;

   assign qbar = ~q;

   always @ (posedge clk or posedge rst) begin
      if (rst)
        q = 1'b0;
      else
        q = d;
   end
endmodule
```

always 節は括弧の中で指定された条件に変化があったときに begin〜end 内を 1 回実行するようになっています。また、posedge は指定した信号が 0 から 1 に変化したことを示しています。この例では clk か rst のどちらかが変化した場合に always 節内を実行することになります。

### カウンタ

フリップフロップを接続することでカウンタをつくることができます。4 ビットカウンタの例を図 15 に、4 ビットカウンタのタイミングチャートを図 16 に示します。

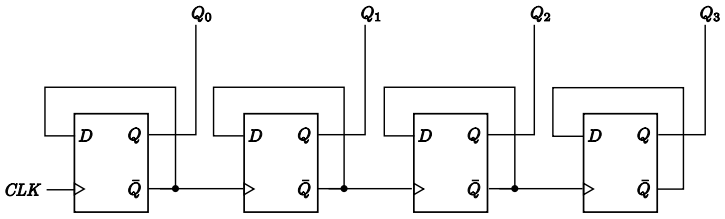



図 15: 4 ビットカウンタ

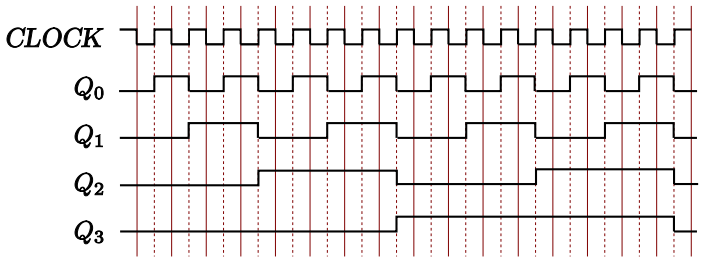



図 16: 4 ビットカウンタのタイミングチャート

**課題 5**

Verilog で次の回路を作成してシミュレーションを行いなさい。レポートには Verilog ソースとシミュレーション結果を添付すること。

(1) リセット付き D-フリップフロップ
$\overline{\text{RESET}}$ は”0”になったら $Q$ を 0、$\bar{Q}$ を 1 とするような入力とする

(2) リセット付き非同期 4 ビットカウンタ (図 15)
$\overline{\text{RESET}}$ は”0”になったらカウントを 0 にする (すべてのビットを 0 にする) ような入力とする。出力 $Q_0$〜$Q_3$ は 4 ビット幅の信号とすること。

**課題 6**

Verilog で次の回路を作成してシミュレーションを行いなさい。レポートには Verilog ソースとシミュレーション結果を添付すること。

(1) 16 ビットカウンタ
**他のモジュールは一切使わないこと**。CLOCK, $\overline{\text{RESET}}$, ENABLE を 1 ビット入力、$Q$ を 16 ビット出力として考えること。

- CLOCK はクロック
- $\overline{\text{RESET}}$ は”0”になったらカウント $Q$ をリセットする入力
- ENABLE は”1”のときにはクロック立ち上がり時に $Q$ をカウントアップするような入力 (ストップウォッチのストップボタンのようなもの)。
- $Q$ はカウンタの値

(2) (1) のカウンタを 4 ビットごとに 16 進数表示させる
課題 2-(4) の 7 セグメント LED デコーダを 4 つ用いなさい

**課題 7**

課題 6-(2) で作成した回路を評価ボード上で動かしなさい。クロックとリセットは入力信号 CLK と RST をそれぞれ用いなさい。

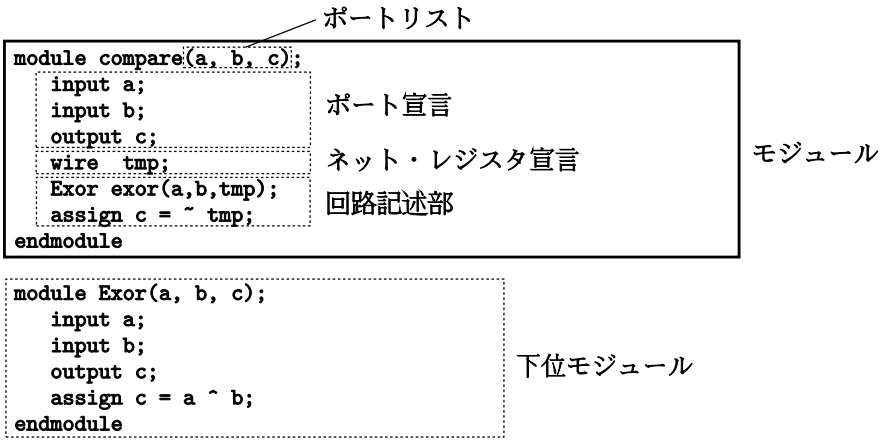
# A　Verilog 簡易マニュアル

ハードウェア記述言語 (HDL) は、信号の流れによって並行的に動作するハードウェアを記述するために開発された言語である。HDL には手続き的に実行するソフトウェアのプログラミング言語とは異なり、論理値やその信号強度、信号に対する動作、タイミング、並行処理などのハードウェアの表現に適した構文が用意されている。

Verilog は Pascal や C に似た文法を持つハードウェア・モデリング言語として、Automated Integrated Design Systems の Phil Moorby と Prabhu Goel によって 1984 年頃開発された。その後、同種のハードウェア記述言語である VHDL の IEEE 標準化に伴ない、Cadence Design Systems(Automated Integrated Design Systems を 1990 年に買収) は Verilog の規格を公開して、Verilog を IEEE 1364-1995 として規格化した。これを Verilog-1995 と呼ぶ。標準化にともない Verilog シミュレータは、Cadence 以外の会社やフリーのものも登場するようになった。

## A.1　Verilog の言語仕様

- キーワードの大文字小文字は区別される
- コメントアウトは//と/* */




**ポートリスト**　外部とのインタフェースを宣言する。

**ポート宣言**　モジュールで用いている信号のモードやタイプを宣言する。ポートリストに記述したものを全て宣言する必要がある。

**ネット宣言、レジスタ宣言**　モジュールで用いる信号のモードやタイプを宣言する

```
wire      c1;  // 1bit ネット
reg [3:0] q;   // 4bit レジスタ
```

**パラメータ宣言**　モジュールで用いる定数を宣言する

```
parameter STEP = 10;
```

**回路記述部**　モジュールの内部動作を記述する

**モード**　信号の方向であり、input か output かを指定する。**in の信号に値を書き込むことはできず、out の信号の値を読み込むことはできない** (エラーとなる)。

**タイプ**　信号の種類であり、wire か reg かを指定する。

- wire は値を状態として保持することができない (配線上のノードとして用いる)。wire の代入は assign 文でのみ可能 (配線の接続を意味する)
- reg は値を状態として保持することができる (レジスタ)

**多ビット信号**　[:] を用いて多ビット信号を宣言することができる。[最上位ビット:最下位ビット] で指定する。代入のときに範囲指定することもできる。右辺信号のビットが余る場合には切り捨てられ、右辺信号のビットが足りない場合には 0 で埋められる。

```
wire [7:0] x;       // 8bit 幅
wire [3:0] y;       // 4bit 幅
wire       z;       // 1bit 幅
assign y = x[7:4]; // 範囲指定した代入
assign z = x[2];   // 1 ビット指定代入
```

## A.2　回路記述部

回路の内部動作を記述する部分である。

**assign 文**　ネット信号の代入に用いる。reg 型には用いることができない。

```
assign c = ~ (a ^ b);
```

**インスタンス化**　下位モジュールと接続する

```
reg ina, inb;
wire outc;
compare comp(ina, inb, outc);
```

**always 文**　ある条件がトリガとなって動作する順序回路の記述に用いる (詳細は後述)

## A.3　演算子

### 論理演算子

| | |
|---|---|
| & | and |
| \| | or |
| ~ | not |
| ^ | xor |
| ~\| | nor |

### ビット連結演算子

```
wire a;
wire b;
wire [1:0] c;
assign c = {a, b}
```

## A.4　ビット表現

$w$'$cv$ という形式で指定する。

- $w$ にはビット幅を指定する (省略時 32bit)

- $c$ には基数を指定する。2 進数、8 進数、10 進数、16 進数をそれぞれ b、o、d、h で指定する (省略時 d)
- $v$ には値を指定する。間に_を入れることが可能

例を次に示す

```
79            // 00000000000000000000000001001110
1'b0          // 0
16'hffff      // 1111111111111111
8'h79         // 01111001
8'd79         // 01001110
8'b1100_0110  // 11000110
```

## A.5 always

always 節は指定された条件に変化があったときに文を 1 回実行する。文を複数書きたいときは、begin と end で囲う。

```
always @ (条件) 文
```

always で用いることができる文を次に示す。

- if else
- case 文
- ブロッキング代入

### if 文

if の後ろの条件式を評価し、真であれば直後の文を実行する。そうでなければ else の後ろの文を実行する。文を複数書きたいときは、begin と end で囲う。if 文を用いて compare を実装した場合の例を次に示す。

```
module compare(a, b, c);
   input a;
   input b;
   output c;
   reg    c;
   always @ (a or b) begin
      if (a == b)
        c = 1'b1;
      else
        c = 1'b0;
   end
endmodule
```

### case 文

case の後ろの式を評価し、値が一致した文を実行する。値が一致するものがなければ defalut の文を実行する。case 文を用いて compare を実装した場合の例を次に示す。

```
module compare(a, b, c);
   input a;
   input b;
   output c;
   reg    c;
   wire [1:0] tmp;
   assign tmp = {a, b}; // a と b の連結
```

```
   always @ (a or b) begin
      case (tmp)
        2'b00 : c = 1'b1;
        2'b01 : c = 1'b0;
        2'b10 : c = 1'b0;
        2'b11 : c = 1'b1;
      endcase
   end
endmodule
```

### ブロッキング代入

<=は非ブロッキング代入といい、実行時に並列に評価される。次の例の場合、c が変化すると a と b への代入が同時に行われ、結果として a と b の値が交換される。

```
reg a, b;
wire c;
always @(c) begin
   a <= b;
   b <= a;
end
```

## B　テストベンチでのクロックの使用例

always #(STEP/2) inclk = ~ inclk; とすることで、クロック時間の 1/2 (= 5ns) ごとにビットを反転させることができる。

```
`timescale 1ns/1ps
module counter_test();
   parameter STEP = 10;
   reg inrst, inclk, inenb;
   wire [15:0] v;
   counter _res(inrst, inclk, inenb, v);

   always #(STEP/2) inclk = ~ inclk;

   initial begin
      $dumpfile("result.vcd");  // 波形データ出力先を指定
      $dumpvars(0, _res);       // 出力変数を指定
      $monitor($stime, "\t clk=%b, rst=%b, enb=%b, v=%b",
               inclk, inrst, inenb, v);

      #0           inrst <= 1'b0;                inclk <= 1'b0;
      #STEP        inrst <= 1'b1; inenb <= 1'b1;
      #(STEP * 30)                inenb <= 1'b0;
      #(STEP * 5)  inrst <= 0'b1;
      #STEP        inrst <= 1'b0; inenb <= 1'b1;
      #(STEP * 30)

      $finish;
   end
endmodule
```