まえがき

本書では，HDL（Hardware Description Language ；ハードウェア記述言語）設計の初心者向けに，Verilog HDL およびVHDL によるテストベンチの書き方と，基本的な検証のノウハウを解説します．

テストベンチというのは，HDL で設計した回路に対して，これが設計者の意図通りになっているか確認する環境を，同じHDL で記述したものです．テストベンチによる回路の検証は，数ある検証方法の一つでしかありませんが，最も重要で効率が良く，実際に最も多く実施されている方法です．

HDL には非常に多くの文法が用意されています．本書では，テストベンチの記述で用いられる文法を初心者に分かりやすいように，段階的に解説していきます．また，文法の解説とともに，検証における基本的なノウハウを紹介しています．そして，初心者の陥りがちな失敗を示すとともに，検証効率を上げる方法を解説しています．

本書は大きく３部構成になっています．

第１部は，そもそも検証とは何のために必要かということから，簡単なテストベンチを書いてシミュレーションし，波形を見て検証するというところまでを解説しています．第１部の内容を理解すれば，効率の良し悪しは別として，HDL で書かれた回路を検証することができるようになります．

第２部では，テストベンチの記述に必要なVerilog HDL，VHDL の文法を，検証の例を見ながら順に解説していきます．第２部を理解すれば，一般的なHDL の検証に必要な文法のほとんどすべてを習得したことになります．第１部までの文法に比べて，格段に効率良く検証を実施できるようになります．

そして，最後の第３部では，そこまで解説した文法を使った応用的なテストベンチの書き方と，そのほかの検証のノウハウについて解説します．第３部までを理解すれば，HDL 設計者として自分は何ができて，何ができないのかを知ることができます．そして自分に不足しているものがあれば，それを自分で調べることができるようになります．

本書では，HDL の記述例をなるべく多く記載しています．それは，多くの記述を見たことがある，あるいは知っている方が，やりたいことを実現する際の選択肢が増え，読者であるHDL 初心者の助けになると考えたからです．

また，本書の巻末には文法の書式をまとめた項目が付いています．文法を忘れてしまったときや，文法から使い方をたどりたいときには，これを参考にしてください．

本書で解説している知識やノウハウは，HDL の設計をはじめて３～５年経つ人であれば誰でも知っていることばかりです．しかし，HDL 設計をはじめたばかりの技術者に，そのノウハウを丁寧に教えてくれるような先輩や上司が，いつもいるとは筆者には思えませんでした．というのも，一人前のHDL 設計者のほとんどが，厳しい開発現場でスケジュールに追われていることが多く，後輩に丁寧な指導をするような余裕はないように思えます．そして技術者の中には，まだまだ技術やノウハウは，教えられるものではなく，

人の仕事を見て盗むものと考え，手取り足取りで教えることには否定的な方も多いように感じます．また，まだ十分な技術や知識を身につけていないにも関わらず，たった一人で客先の開発現場に派遣され，困っている方にお会いしたことがあります．

知識やノウハウのほとんどは，知っている人には当たり前で，取り立てていう程のことではないと思われています．しかしながら，それを知らない人は，知らなかったばかりにとんでもなく時間を掛けてしまったり，あるいはひどく品質を下げてしまったりといった羽目に陥ることがあります．これは本人だけでなく，一緒に開発している方々にとっても不幸なことです．

そこで筆者は，HDLの文法だけでなく，HDL初心者の方が開発現場で困る前に，基本的な検証のノウハウをお知らせしたいという思いで本書を執筆しました．もしあなたがすでに3～5年のHDL設計の経験をお持ちだとすると，本書の内容はすでに知っているものばかりかもしれません．しかし，そこにあなたの後輩や部下に知らせたい知識やノウハウがあったならば，本書があなたのお手伝いをできることでしょう．そして，気付きによって効率良く検証できるようになったならば，あるいは気付かなかったとしても本書に書かれているノウハウによって開発で苦労せずに済んだのであれば，筆者としてはとてもうれしく思います．

最後になりましたが，本書執筆の機会を下さった小林優氏，CQ出版の西野直樹氏，編集にあたって多大な迷惑をおかけしたCQ出版の方々に厚くお礼を申し上げるとともに，本書を手にとって頂いた方々に感謝いたします．

2010年8月　安岡貴志

Verilog HDL&VHDL テストベンチ記述の初歩

CONTENTS

本書は，Design Wave Magazine 2007年5月号～2009年3/4月号で連載された「初歩からのHDLテストベンチ」の記事をもとに，加筆・再編集したものです．

第１章

検証の重要性とテストベンチ

1.1　検証で用いるテストベンチとは

あなたがVerilog HDLやVHDL（以降，両方を指すときには単にHDLという）で回路を書いたとき，必ずそのコード（HDLの文法で書かれた文章やプログラム）が正しく書けているかを確認しなくてはなりません．なぜなら，そのコードには勘違いや書き間違いをして，思ったものと違う回路になってしまっている可能性があるからです．

ミスは人間であれば仕方がないことです．どんなに経験が豊かであろうと，どんなに注意深く書こうと，ミスというものはなくなりません．設計する回路が複雑になればなるほど，いくつものファイルに分かれるようになり，ソース・コードの量が増えれば増えるほど，ミスが入り込む可能性は上がります．

HDLで書いた回路が，意図した通りに出来ているかどうかを確認することを検証（Verification）といいます．検証では，論理シミュレータをはじめとする回路の動作を模擬（シミュレーション）するツールを使います．このとき，設計した回路とテストベンチ（Testbench）の二つが必要になります．

● テストベンチはHDLで記述する

テストベンチは，回路と同じようにHDLで記述します．

回路の記述ではHDLの文法のすべてが使えるわけではありませんでした．なぜなら，回路の記述は，後に回路を合成するために，その動作が物理的な回路として実現できる範囲に制限されるからです．

これに対してテストベンチでは，HDLの文法のすべてを，それこそC言語のプログラムのように使うことができます．逆に言えば，回路を書いている間には使わなかった文法がテストベンチを書く際には必要になってきます．

本書では，テストベンチの基礎からテストベンチ特有の文法までを，順を追って解説していきます．

● FPGA向けの回路でも検証が必要

FPGA（Field Programmable Gate Array）向けの回路の開発でHDLを使う場合，あなたはテストベンチを作ってシミュレーションしなくても，FPGAに搭載して検証すればいいじゃないか，と思うかもしれません．

しかし，期待通りに動かなかった場合には，途端に効率が下がってしまいます．FPGAのピンをオシロスコープやロジック・アナライザなどで観測しながら，解析しなければならないためです．FPGAの入出力信号だけで不具合の判断ができないときには，内部信号を空きピンに出力する記述を加えなければなりません．観測したい信号の方が空きピンよりも多ければ，ピンに接続する信号を切り替えながら観察するなど，大変な労力が必要になります．例えば，ある内部信号について，ある期間に1クロック分のパルスが17回出ているか確認しようとするだけでも非常に大変です．

これに対し，テストベンチを作ってシミュレーションし，結果を波形で観測する方法を使えば，任意の内部信号を必要なだけ並べて確認することができます．FPGAの開発であっても，最初にシミュレータによって回路の動きを確認し，動作を確実にしてから実機で検証する方が，圧倒的に効率が良くなります．

また，近年FPGAは回路規模が上がり，動作が複雑化しています．すべてが出来上がってから，実機だけで解析することは不可能になっています．

FPGAの開発であっても，テストベンチを作成してシミュレータにより検証することは，もはや必須です．

● 機能ブロック単位で検証する

テストベンチを作るためには，回路を作るのと同じか，場合によってはそれ以上の時間がかかります．ゲートやカウンタ，セレクタなどを一つずつ検証していては，いくら時間があっても終わりません．従って検証は「××処理部」や，「○○制御部」といった機能ブロック単位で行います（**図1.1**）．

そして，機能ブロックごとの検証が終わってから，検証済みの機能ブロックを接続して，全体の検証を行うようにします．

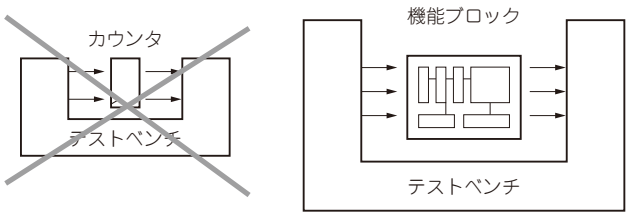


図1.1
機能ブロック単位で検証する
検証は「××処理部」や「○○制御部」といった機能ブロック単位で行う．機能ブロックごとの検証が終わってから，検証済みの機能ブロックを接続して，全体の検証を行う．

第2章

組み合わせ回路のためのテストベンチ

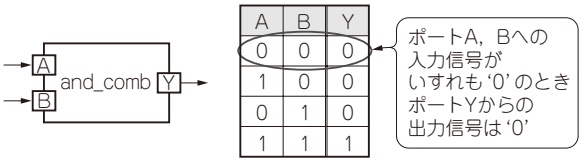
本章では**図2.1**に示す組み合わせ回路のためのテストベンチの回路（コード）を記述します．一番簡単なテストベンチの形は，**図2.2**のようになります．

このテストベンチを作る手順を書き並べると，以下のようになります．

1）検証環境を置く箱を作る
2）箱に検証対象の回路を置く
3）箱の中で入力波形を作る
4）信号をテスト対象の回路のポートにつなげる

2.1　検証環境を置く箱を作る

ここでいう箱とは，**図2.2**の一番外側の枠に相当します．HDLによる回路の記述では，Verilog HDLならばモジュール，VHDLならばエンティティという箱を作ったはずです．テストベンチも同じように，モジュールもしくはエンティティという箱を作ることになります．ただし，回路記述（回路の構成や機能をHDLで表現したコード）と違って，この箱には入出力のポートはありません．テストベンチは，シミュレーションを行う際の最上位階層（一番外側の箱）に当たるので，この箱から外に信号を出し入れする必要がな



（a）ブロック図　　（b）真理値表

図2.1
検証対象の回路
回路の名前は，and_combである．1ビットの入力ポートA，Bと1ビットの出力ポートYを持つ．記憶素子（フリップフロップなど）を含まない組み合わせ回路である

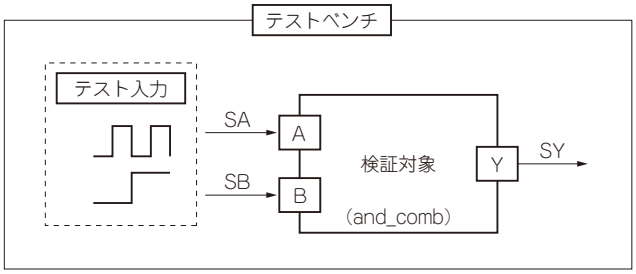


図2.2
テストベンチの構造
テストベンチの中には，検証対象の回路とテスト入力を生成するブロック，検証対象のポートに接続される信号がある．

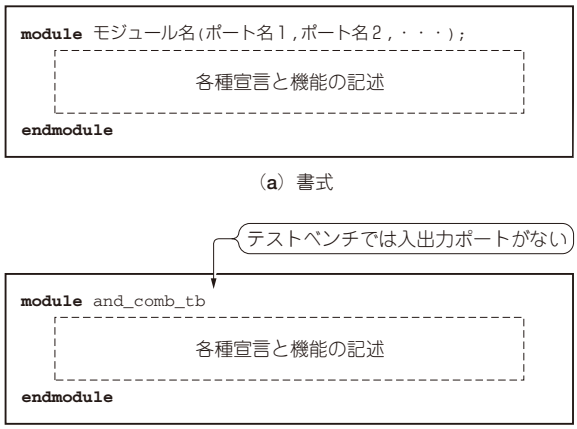


図2.3
Verilog HDLによるモジュールの書式と記述例

いからです．

この箱にand_comb_tbという名前を付けることにします．また，and_comb_tbを記述するファイルを，Verilog HDLであればand_comb_tb.v，VHDLであればand_comb_tb.vhdとします．

Verilog HDL

図2.3(a)は，Verilog HDLによるモジュールの書式です．

点線の枠内は，回路記述ではモジュール内で使う信号の宣言や回路の機能を記述しました．テストベンチand_comb_tbもこの形式に従って書くことになります．

図2.3(b)は，and_comb_tbをVerilog HDLで記述したものです．

回路記述のときには，モジュール名の後ろに入出力ポートを記述していましたが，テストベンチではなくなっています．点線の枠内には，これから検証対象となる回路や，テスト入力を生成する記述を埋めていきます．

VHDL

図2.4(a)はVHDLの書式です．VHDLでは一つの箱がエンティティ宣言，アーキテクチャ宣言，コン

第3章

順序回路のためのテストベンチ

3.1　クロックを含むテストベンチの注意点

クロックを含むテスト対象の回路（順序回路）は，組み合わせ回路と違い独特の注意が必要です．

本章ではクロックを含む回路のテストベンチを作成します．検証対象の回路は，イネーブル付き[注3.1]の12進カウンタにします（コラム「幅のある信号の表記」を参照）．仕様（機能）を**図3.1**に示します[注3.2]．

● クロックを作る

第2章で説明した組み合わせ回路のテストベンチと大きく違う点として，クロックの存在があります．クロックは，一定周期で‘0’（Lレベル）と‘1’（Hレベル）を繰り返す信号です．

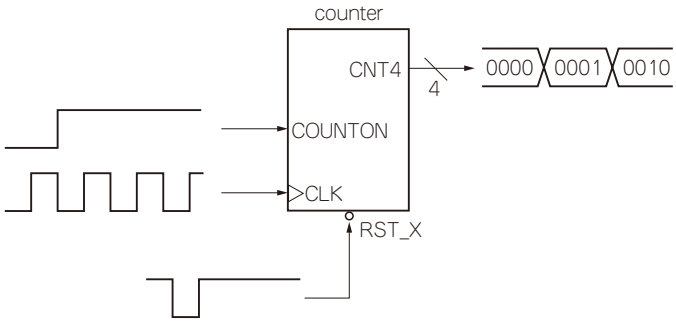
今回のテストベンチは**図3.2**のようになります．

Verilog HDL

リスト3.1は，Verilog HDLでCLKというクロックを作ったものです．CLKは，50nsで‘0’（Lレベル），‘1’（Hレベル）を繰り返します[注3.3]．CLKの周期は100nsになります．

クロックは，ほかのテスト入力とは別のブロック（always文）で作ります（コラム「クロックの記述」を参照）．Verilog HDLでは別々のブロック（always文やinitial文など）は，並列に（並行して）動きます．

注3.1：イネーブルとは，何らかの機能を許可する信号である．今回であればイネーブル信号COUNTONが‘1’のとき，カウント機能が許可になり，カウントを実行する．
注3.2：実際の開発ではこの規模でテストをすることはない．クロックを含む回路のテストベンチの要点を明確にするために，あえて小規模な回路にしている．
注3.3：本書では説明を簡単にするために，Verilog HDLにおける時間単位がnsである前提で説明する．時間単位の初期設定はシミュレータにより異なる．



（**a**）ブロック図

- ●回路の名前はcounterとする．
- ●カウント値は，出力ポートCNT4から出力される．カウント値は0から11までの値をとり，ビット幅は4ビット．
- ●非同期リセットが付いている．入力ポートRST_Xに接続された信号が‘0’（Lレベル）になると，カウント値は0に戻る．
- ●（RST_Xが1で）入力ポートCOUNTONに接続されている信号が‘1’のとき，クロック（入力ポートCLKに接続された信号）の立ち上がりごとに，カウント値は+1される（カウント値が11のときは0に戻る）．
- ●（RST_Xが1で）入力ポートCOUNTONに接続されている信号が‘0’のとき，カウント値は保存される．

（**b**）仕様

図3.1
イネーブル付きの12進カウンタの仕様
組み合わせ回路と異なりクロックが必要．

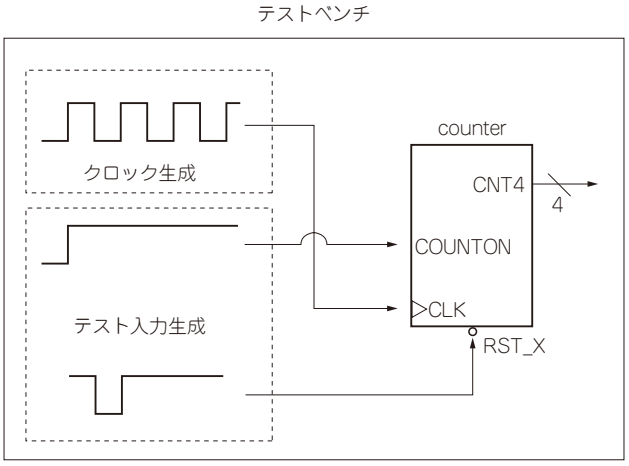


図3.2　イネーブル付きの12進カウンタのテストベンチ
クロックとテスト入力を生成する．

リスト3.1　Verilog HDL によるクロックの記述例

```
reg         CLK;

always begin
                CLK = 1'b1;
  #50           CLK = 1'b0;
  #50;
end
```

リスト3.2　VHDL によるクロックの記述例

```
（宣言部分）
signal CLK : std_logic;

begin

（機能部分）
process begin
  CLK <= '1'; wait for 50 ns;
  CLK <= '0'; wait for 50 ns;
end process;
```

第４章

遅延の記述方法

4.1　相対遅延と絶対遅延

相対遅延とは，前の信号の変化から何nsで代入を実行するというように，遅延を相対関係で表現する手法です〔**図4.1 (a)**〕．これに対して絶対遅延とは，シミュレーション開始時点（シミュレーション時間0）から何nsで代入するというような，絶対時間で表現する手法です〔**図4.1 (b)**〕．

ここでは，**図4.2**の仕様を持つ検証対象の回路に対して，**図4.3 (a)**のテストベンチから，**図4.3 (b)**の信

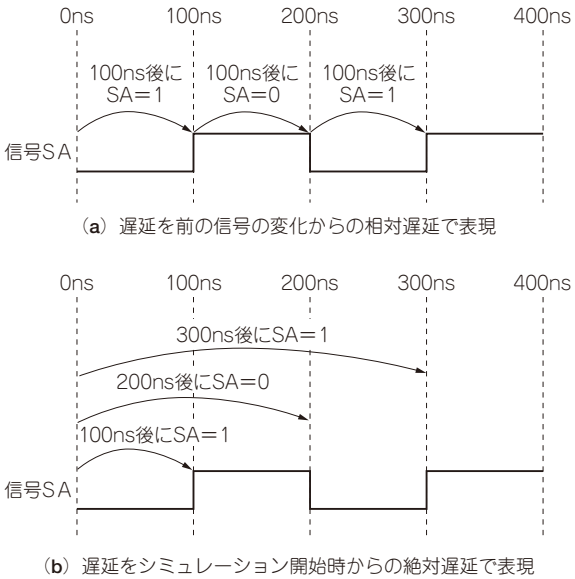


図4.1
遅延の表現
前の信号の変化からの相対関係で表現する相対遅延と，常に開始時点からの時間で表現する絶対遅延がある．

（a）ブロック図

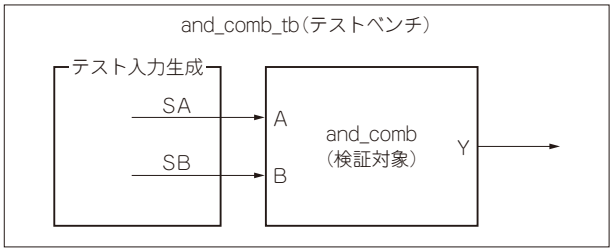
A	B	Y
0	0	0
1	0	0
0	1	0
1	1	1

ポートA，Bへの入力信号がいずれも‘0’のときポートYからの出力信号は‘0’

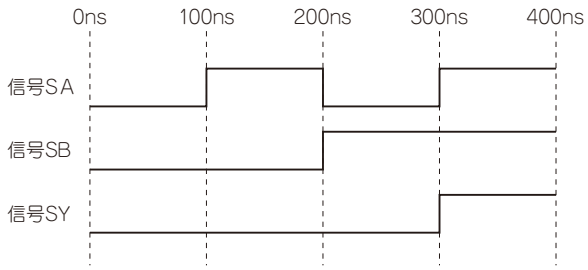
（b）真理値表

図4.2　検査対象の回路

回路の名前は，and_combである．1ビットの入力ポートA，Bと1ビットの出力ポートYを持つ．記憶素子（フリップフロップなど）を含まない組み合わせ回路である．



（a）ブロック図



（b）タイミング・チャート

図4.3　AND回路の検証

(a)のようなテストベンチから，(b)のテスト入力を与える．

号SAと信号SBのようなテスト入力を与える記述法を，相対遅延と絶対遅延を使って解説します．

Verilog HDL

リスト4.1（a）は，第2章で示したテスト入力の記述例です．ここで記述された信号SA，SBの変化のタイミングは**図4.3（b）**と同じです[注4.1]．

リスト4.1（b）は，**リスト4.1（a）**の記述を1行当たり一つの式だけの記述に直したものです．式の実行タイミングは，式の間の相対遅延で記述されています．**図4.4**のように，begin～endの間に記述された式は上から順番に実行され，その間の遅延は累積されます（式2は必ず式1の後に実行され，式3は必ず式2の後に実行される）．

図4.5はfork～joinという文法の書式を表しています．fork～joinは，begin～endと違い，式の間で遅延は累積しません．それぞれの式がシミュレーション開始時から独自の遅延値に従って並行に実行されます．つまり，各式の遅延値は絶対遅延となります．

リスト4.2は，**リスト4.1**とまったく同じ変化タイミングで信号SAとSBのテスト入力を記述したものです．各代入式の＃に続く遅延値は絶対遅延となり，式の間で累積しません．

注4.1：本書のVerilog HDLのすべての解説では，シミュレータのシミュレーション時間の単位が，nsに設定されているものとして解説している．

第５章

標準出力の記述方法

本章では波形以外のシミュレーション結果の確認方法として，標準出力による確認方法とその文法を紹介します．ただし，どちらの方法もシミュレーション実行直後に，内容を確認しなければ，結果は消えてしまいます．

実設計においては，回路の完成までに何度もシミュレーションを行い，その結果をテキスト・ファイルに保存しておくという手法をとるのが普通です．結果をテキスト・ファイルに保存するための文法は，標準出力のための文法と非常に似通っています．今回の標準出力のための文法がしっかり理解できれば，ファイルで残す手法も簡単に利用できます．

5.1　標準出力の書き方

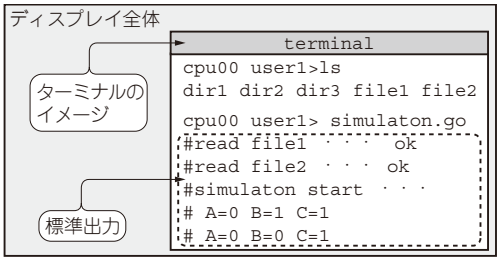
標準出力とは，処理結果を文字列で出力するものと考えてください．UNIXやLinuxなどのターミナル上でシミュレーションを実行しているのあれば，そのターミナル〔**図5.1 (a)**〕に表示されます．シミュレーション・ツールで専用ウィンドウを表示しているのであれば，その中でログなどが表示されるウィンドウ〔**図5.1 (b)**〕に表示されます．

Verilog HDL

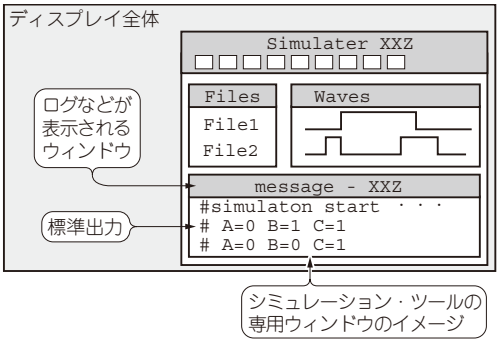
Verilog HDLでテキストを出力するためには，システム・タスク`$display`を使います．

図5.2 (a)に`$display`の書式を示します．かっこの中に書かれた信号の値や，ダブル・クォーテーション(" ")の中に書かれた文字列を標準出力に表示します．シミュレーション中にこのタスクが実行されると，その時点の信号の値とダブル・クォーテーションの中の文字列が標準出力に表示されます．

図5.2 (b)に`$display`の記述例と表示例を示します．



（a）UNIXやLinuxのターミナルのイメージ



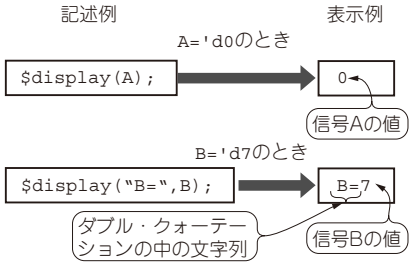
（b）シミュレータ専用ウィンドウのイメージ

図5.1　標準出力

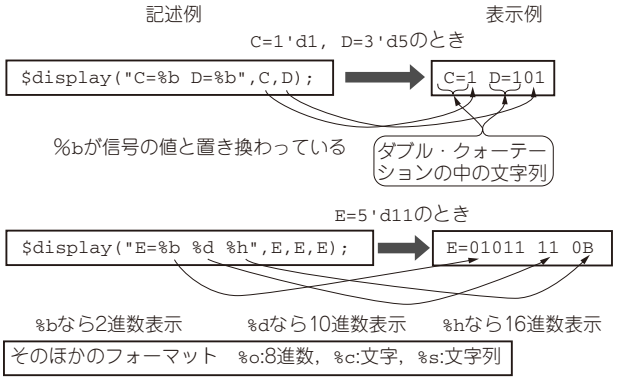
標準出力は，UNIXやLinuxなどのターミナル上でシミュレーションを実行していれば，そのターミナル，シミュレーション・ツールで専用ウィンドウを表示していれば，その中でログなどが表示されるウィンドウになる．

```
$display(信号名);
$display(信号名,信号名,・・・);
$display("文字列",信号名);
$display("文字列",信号名,"文字列",信号名,信号名,・・・);
```

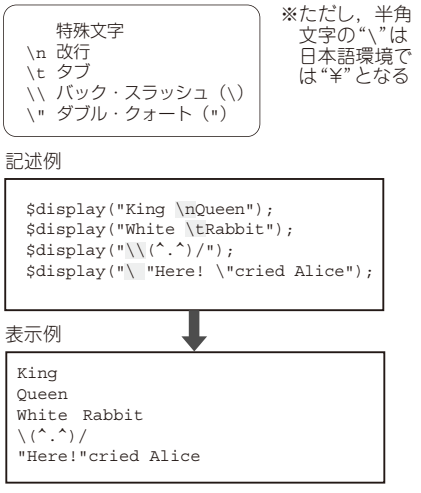
（a）書式



（b）記述例と表示例



（c）%を含む記述と表示例



（e）特殊文字

図5.2　システム・タスク$displayの使い方

Verilog HDLでテキストを入力したり出力したりするためには，システム・タスク$displayを使う．

第６章

ファイル入出力の記述方法

実際の開発では，設計中にはシミュレーションを繰り返しながら何度も回路（HDLで記述したRTLコード）を修正しなければなりません．シミュレーション結果を毎回人間が目視で確認していては，時間が掛かり，見逃しも発生してしまいます．そこで，どこかの時点で人間の目視に頼らず，機械によって自動的に結果を検証する仕組みを取り入れなければなりません．

UNIX環境などで設計をしている場合，シミュレーション結果をファイルに出力すると，diffコマンドなどで簡単に比較できるようになっています．実際の開発では，このようなコマンドを使って機械に比較させることを前提に，シミュレーション結果をファイルに出力させる場合がほとんどです（**図6.1**）．

6.1　ファイルによる検証結果の確認の方法

シミュレーション結果をファイルに出力する場合，同じ形式の期待値ファイル[注6.1]があれば簡単に結果を確

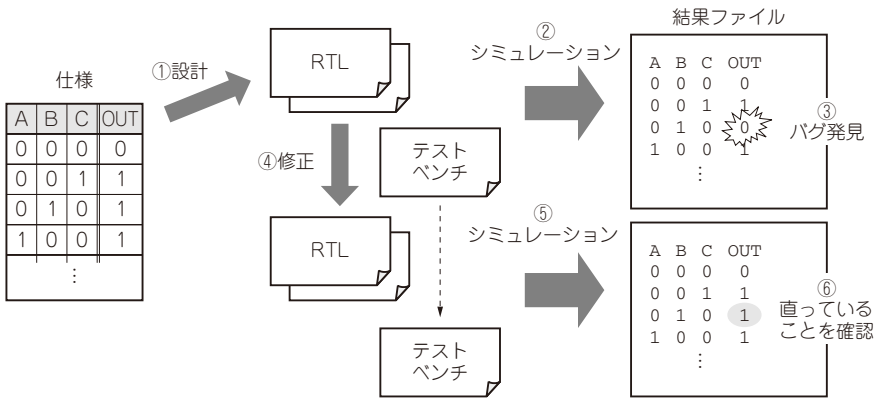


図6.1
ファイルによる検証結果確認
実際の開発では，シミュレーション結果をファイルに出力させる場合がほとんどである．仕様とシミュレーション結果とをコンピュータなどの機械を使って比較し確認する．

認できます．

● 期待値ファイルがない場合の検証手順

期待値ファイルがない場合には，初回は仕様とシミュレーション結果のファイルや波形表示を一つ一つ見比べて確認しなくてはなりません（**図6.2**）．しかし，2回目以降は前回のシミュレーション結果ファイルとをdiffコマンドなどで比較して，以下の項目を確認します．

- 修正箇所が直っているかどうか
- 前回正しく動いていた部分に新たに不具合が発生していないかどうか

単純な比較などの作業は，機械にやらせる方が人間が行うより早くて正確です．設計現場では少ない人手で，膨大な作業を要求されることが多いので，なるべく早い段階で機械ができることは，機械に任せられる環境を整えなければなりません．

● 期待値ファイルがある場合の検証手順

シミュレーション結果のファイルと同じ形式の期待値ファイルをC言語などで作成している場合は，初回からdiffコマンドなどを使った検証結果の確認が可能です（**図6.3**）．

● ファイル出力のための文法

ここでは，シミュレーション結果をファイルに出力する方法を解説します．

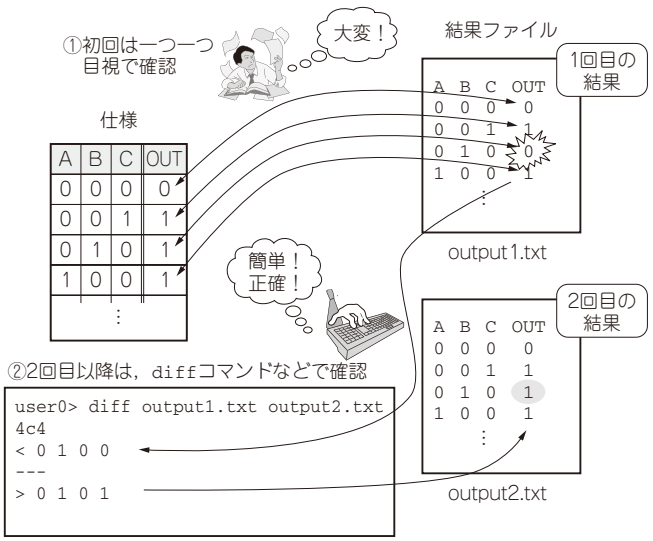


図6.2
期待値ファイルがない場合の検証手順
初回は仕様とシミュレーション結果のファイルや波形表示を見比べて一つ一つ確認する．2回目以降は，前回のシミュレーション結果ファイルとコンピュータなどの機械を使って比較する．ここでは，修正箇所が直っているかどうか，前回正しく動いていた部分に新たに不具合が発生していないかどうかを確認する．

注6.1：この場合の期待値ファイルとは，仕様通りに設計できていれば回路（RTL）が出力するはずのシミュレーション結果と同内容のファイルのこと．

第７章

タスク/プロシージャの記述方法

本章では，タスクとプロシージャを使用したテストベンチの構造を解説します．

タスク/プロシージャを使ったテストベンチの効果として，次の三つがあります．

- 記述量が減る
- ミスが減り，修正が容易になる
- テストベンチの見通しが良くなる

タスク/プロシージャを習得すると，皆さんの設計効率は格段に上がります．誤記や写し間違いを探すのに消費されるような無駄な時間を減らすことができるはずです．

初めて見ると，きっととっつきにくい文法で，その難しさに逃げ出したくなると思います．それでも，後で楽をするためにこれを習得し，エンジニアとして意義のある仕事により多くの時間を使っていただきたいと思います．

7.1　テストベンチの構造化

テストに必要な機能の記述，特にテスト入力をただ一つの順次処理文（initial 文，process 文）に書こうとすると，大変な行数になります．行が多くなればなるほど，書くのが大変になるだけではありません．一目でそのテストが何をしているのかが分からなくなり，思わぬ動きをしたときに不具合の原因を突き止めるのも大変になります．

そこで，テストの規模が大きくなると，テストベンチを構造化することが必須になってきます．テストベンチを構造化するには，Verilog HDL であればタスク，VHDL であればプロシージャを使用します．これらは，メインのルーチン（テスト入力）に対するサブルーチン（サブプログラム）の役割を果たします．記述量や間違いを減少させ，読解性やデバッグの効率を高めます（**図7.1**）．

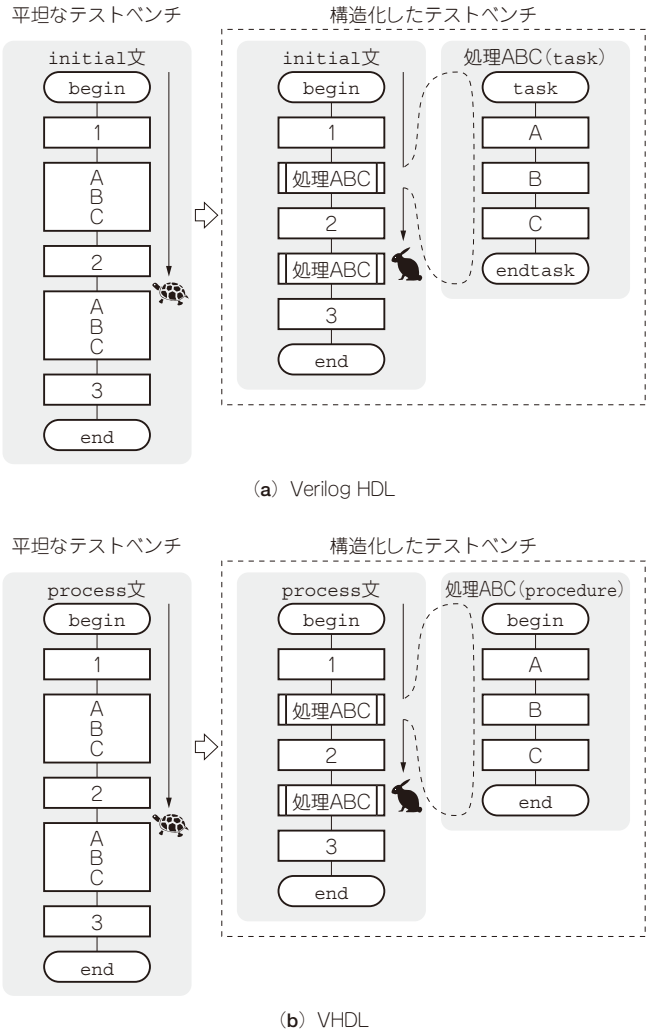


図7.1
テストベンチの構造化
テストの規模が大きくなってくると，テストベンチの構造化が必須になる．

● タスクの文法

Verilog HDL

図7.2(a)はタスク定義の書式です．内部信号宣言は，タスクの内部だけで使用する信号を宣言します．処理の記述には，**図7.1(a)**のA，B，Cの処理のように，構造化したい(まとめたい)処理を記述します．引き数宣言は後述します．

図7.2(b)にタスク定義の記述例を示します．仮引き数dirはタスクの中にしか現れない引き数です．タスクの呼び出し時には，与えられた引き数と置き換えられます．

処理の記述内にある信号UPは，タスクを呼び出しているテストベンチの階層に存在する信号です．タスクが呼び出されたのと同じ階層に存在する信号を参照したり，その信号に代入するのに，その信号を引き

第８章

階層化の記述方法

回路が複雑になってくると，画一的なテスト入力では効率的な検証ができません．そこでシミュレーション・モデルを使用すると，検証対象の回路の出力信号に合わせた信号を生成することが容易になります．シミュレーション・モデルは自作する以外にも，あなたの会社・グループで蓄積されてきた設計資産を使用することもできます．一般的なRAMやバス，CPUモデルは，EDAツール・ベンダが提供しているものを利用することもできます．

テスト環境を構築する際，画一的なテスト入力で事足りるのか，シミュレーション・モデルを使うのか，シミュレーション・モデルを使うのであれば，それをどう入手するかを判断する必要があります．状況に合わせて最も効率的な手法で検証に挑んでください．

また，シミュレーション・モデルを使うくらい検証対象が複雑になってくると，効率良く検証するために，各ファイルをどう配置するかもよく考えなければいけません．ファイルがよく整理されていなかったせいで，意図したテストと違うテストをして，結果に悩むようなことは大変な時間の無駄です．絶対に避けましょう．

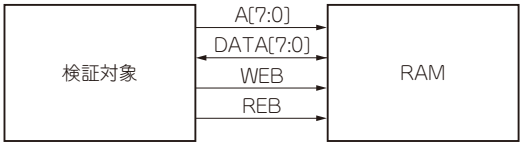
8.1　RAMのシミュレーション・モデル

Verilog HDL・VHDL共通

第7章までのテストベンチでは，テストベンチの記述されたモジュール（ファイル）は一つだけでした．そして，検証対象への入力データは，すべてこの中で`initial`文や`always`文を使って書かれていました．

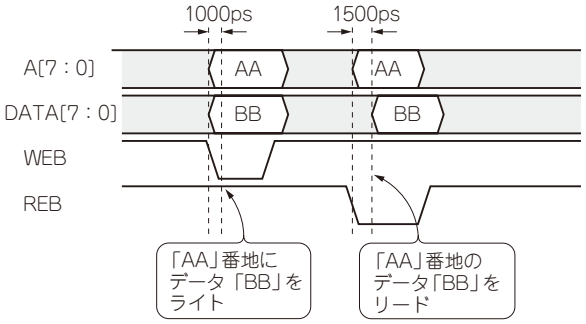
それではメモリとのインターフェースを持つ回路を検証するのに，この方法は効率的でしょうか．

図8.1は，RAMとのインターフェースを持つ回路の接続図と，RAMへのリード（読み出し）/ライト（書き込み）のタイミングを表しています．



入出力信号名	概要
A	リード/ライト共用のメモリのアドレス
DATA	ライト時：RAMに書き込むデータを入力 リード時：RAMから読み出すデータを出力
WEB	この信号が‘0’のときにライト
REB	この信号が‘0’のときにリード

（**a**）実際の接続の様子と信号の仕様



（**b**）リード/ライトのタイミング

図8.1　RAMとのインターフェース

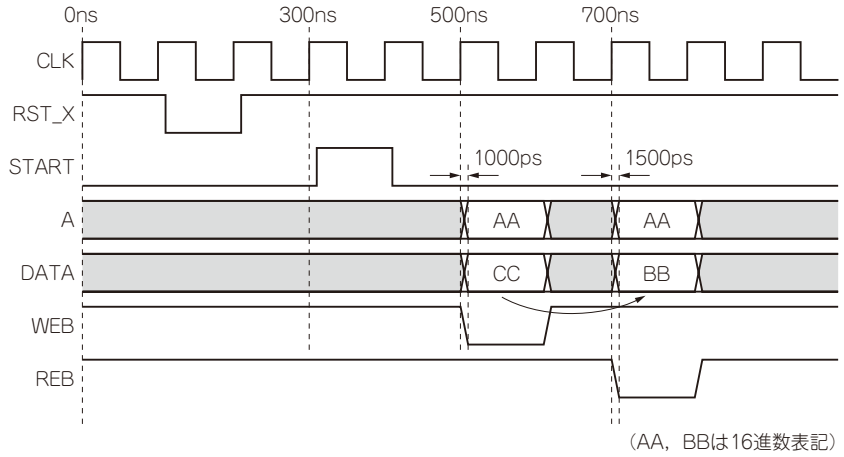


図8.2
RAMを含むテストで想定されるタイミング・チャート

このような回路を検証する際，第7章までのように検証対象からの出力と関係なくテスト入力を生成するやり方は，いくつかの問題を含みます．

例えば，次のような検証回路とテストを考えてみましょう．検証対象回路はSTART信号が‘1’になると，RAMに値を書き込み，さらに読み出すものとします．検証対象回路が正しく動いていれば，**図8.2**のようなタイミングになります．

これを第7章までに説明してきた方法でテストベンチに記述すると，**リスト8.1**と**リスト8.2**のようになります．しかしこれらのテストベンチでは，**図8.3（a）**のように検証対象回路からライトされる値が想定と違っていたり，**図8.3（b）**のようにライト，リードのタイミングが違っていても，テストベンチからは同じタイミングでRAMの出力に当たる信号が入力されてしまいます．これでは検証対象回路の不具合を見逃してしまう恐れがあります．

図8.4の構成では，RAMモデルは，テスト対象と同じように一つのモジュール（Verilog HDL）/エン

第9章

期待値比較の記述方法

9.1　期待値の比較を自動化する

Verilog HDL・VHDL共通

第6章では，期待値ファイルとシミュレーションの結果ファイルを，diffコマンドなどで比較する方法を紹介しました．しかし，テスト・パターン数が増えてくると，すべてのパターンを手動で期待値チェックすることも非常に煩わしくなります．また，人手が入ることでケアレス・ミスを生みやすくなります（**図9.1**）．

Verilog HDLやVHDLでは，期待値ファイルさえあれば，テストベンチの中で比較することができます．ここでは，期待値の比較を自動化する方法を説明します．

図9.2は，クリップ機能付きの加算回路です．この回路は入力A，Bの加算結果をQに出力します．た

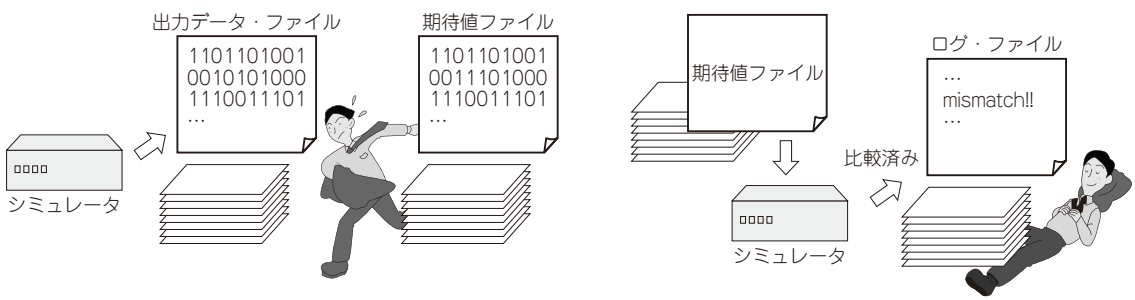


図9.1　出力結果の比較

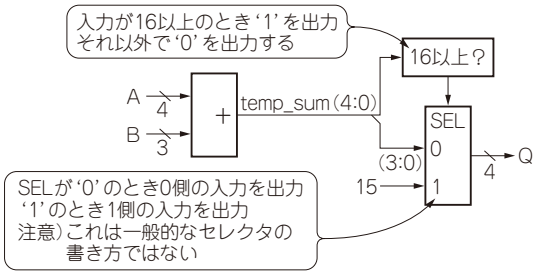


図9.2
クリップ機能付き加算回路
addexの回路構造を示す．

リスト9.1　自動比較機能付きテストベンチ（Verilog HDL）

```
module addex_tp;
parameter STEP   = 100,
          STROBE = 10,
          PATNUM = 8;
reg  [7:0] mem1 [0:PATNUM-1];
reg  [3:0] mem2 [0:PATNUM-1];
reg  [3:0] a,b;
wire [3:0] q;
integer    i,j;

addex addex( .A(a), .B(b[2:0]), .Q(q) );

initial begin
  $readmemh("input.txt",mem1);
  for(i=0;i<PATNUM;i=i+1)begin
    {a,b} = mem1[i];
    #STEP;
  end
  $finish;
end
```

①期待値データを入れる配列

⑥b[3]は捨てるだけ

```
initial begin
  $readmemh("expect.txt",mem2);
  for(j=0; j<PATNUM; j=j+1) begin

    #(STEP-STROBE) if( mem2[j] !== q )begin
      $display("Mismatch Error!");
      $display("q=%h expect=%h",q,mem2[j]);
    end
    #STROBE;
  end
end
endmodule
```

②期待値読み出し

④期待値比較

③タイミング調整

⑤不一致時のメッセージ

表示されるメッセージ
Mismatch Error!
q= 1 expect= F

回路の出力　　期待値

だし，加算した結果が16以上になった場合には，15を出力します．この回路の名前はaddexとします．

最初にこの回路を検証するテストベンチについて解説します．なお，今回のテストベンチでは期待値不一致のメッセージを標準出力に出しています．ツール側でログ・ファイルを出力する場合はこのままで構いませんが，そうでない場合には標準出力ではなくファイルにメッセージを残すようにしてください．

Verilog HDL

リスト9.1は，自動比較機能付きのテストベンチです．

リスト9.1の①は，期待値ファイルを読み出すための配列を宣言しています．出力qは4ビットの信号なので，配列のビット幅は4ビットです．期待値データの数は，テスト入力のデータ数と同じでPATNUM（=8）としているので，配列の番地の数は8個（0番地～7番地）です．

リスト9.1の②は，期待値ファイルexpect.txtを配列mem2に読み出しています．

リスト9.1の③は，期待値を比較するタイミングを調整しています．入力信号は，0，100，200，…，700ns（#1=1nsのとき）で切り替わるようになっていますが，期待値比較は90，190，290，…，790ns