```
DOUT <= SF_RAM(RPTR) when OE='1' else "ZZZZZZZZ";
FF <= FF_REG;
EF <= EF_TMP;
-- FIFO
process(CLK) begin
  if(CLK'event and CLK = '1') then
        if(WEN ='1' and FF_REG='0') then
          SF_RAM(WPTR) <= DIN;
    end if;
  end if;
end process;
-- Write Pointer
process(CLK, RST) begin
  if(RST='1') then
         WPTR <= 0;
  elsif(CLK'event and CLK = '1') then
        if(WEN = '1' and FF_REG='0') then
```

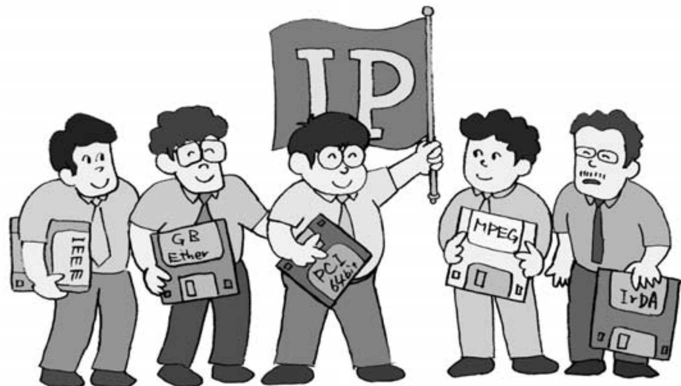
第1章 設計再利用を考慮してHDLを記述しよう

まずはじめに，設計再利用を考慮したHDL記述の考えかたについて，お話しさせていただきます．この手の話をするとき，キーワードとして欠かせないのがIP（intellectual property）ということばです．この「IP」ということばは，いろいろな意味が込められて使用されていると思います．本書では，特に断らないかぎり，「論理合成可能なモデル，および回路ライブラリ」を指して使うことにします．なお，本書に付属するCD-ROMには，次章以降で紹介するさまざまな機能マクロのサンプル記述が収録されています．

さて，現在，LSIの回路規模が大きくなるにつれて，IPを使用しないとLSIを短期間に作ることができないと言われるようになってきました．このIPを提供する，いわゆるIPプロバイダ（IPベンダとも呼ぶ）が注目を集めており，ここ数年，いろいろな新しい会社が出現しています（**図1.1**）．LSIの回路規模がさらに大きくなると，その面積の7～8割を（外部調達や自社開発の）IPが占めるようになるとも言われています．おそらくこのような時代はもう目の前に来ており，遅かれ早かれこうしたIPを使った設計が主流になっていくのだと思います．

その一方で，技術的な側面から見た場合，筆者自身は，「外部調達のIPを使った設計で本当によいのだろうか？」という疑問をもっています．たしかにビジネス的な観点から考えると，以下のような理由で外部調達のIPを使わざるをえないというのも理解できます．

- **短期間にモノ（LSI）を出したい．**
- **バグのために何度もリワーク（再設計）することは許されない．**
- **一つのプロジェクトに何人もの設計者を割くことができない．**



見本

〔図1.1〕IPを提供するベンチャ企業が続々登場

しかし，このエンジニアリングの世界で本当にそれでよいのでしょうか?

LSIが組み立てパソコンのようになる!?

さて，外部調達の（つまり市販の）IPを使用してみんながLSIを設計し始めると，どうなるのでしょうか．たとえが適当なのかどうかわかりませんが，それは秋葉原で売られているできあいのパーツを買ってきて作る組み立てパソコンのようになるのではないかと思います．つまりだれが作っても同じようなものができあがってしまいます．このような状態になると，いったいどのようにして製品の差異化を図るのでしょうか．おそらく，組み立てパソコンの場合と同じようにソフトウェア（OSやアプリケーション）の勝負，あるいはモノの値段だけの勝負になっていくのだと思います（**図1.2**）．

ところで組み立てパソコンの世界において，技術的な蓄積はどこに集まるのかというと，組み立てパソコンに使用されているボードやチップなどを提供しているメーカではないかと思います．こうしたメーカの技術競争があるからこそ，次にどのようなボードやチップを提供したらよいか，どのようなものを作るべきかなどが見えてくるのだと思います．

LSIを作る世界でも，やがて組み立てパソコンの世界と同じようなことが起こるのかもしれません（すでにそうなっているのかもしれないが…）．つまり技術的な蓄積はIPプロバイダの中にしか存在せず，LSIを作る側は単にそのパーツ（IP）を利用するだけで，価格競争や，どうでもいいオマケを付ける競争に向かっていくのかもしれません．ただ欧米の場合は，IPプロバイダの中に技術が蓄積されていくだけまだよいのですが，IPプロバイダがほとんど存在しない日本では，このような方向にすら行っていないような気がします．

設計者一人一人がIPプロバイダになろう

上記のような状況に陥らないためにはどうしたらよいのでしょうか．それはやはり，LSI設計に携わっている設計者自身がIPプロバイダを目指すくらいの気概をもって回路を設計していくことではないかと考えます．IPプロバイダに対して，使用するIPについていろいろと率直に意見を言ったり，的確な指示が出せるように，やはり一度はIPを作ってみるべきではないかと思います．

技術的な進歩は，それまでの失敗や苦労の上に成り立つものです．単に言われたまま，提供されるものを黙って使っているだけでは，しかも使えるか使えないかの判断を下しているだけでは，でき上がったLSIに技術的な差が現れてこないのではないかと思います．

見本

〔図1.2〕LSI設計は，だれが作っても差のない“組み立てパソコン”の世界になる!?

前述したように，ビジネス的な観点から見ると，自分でIPを作っている余裕などないというのが現実でしょう．しかし，そこを乗り越えていかないと，日本の電子業界や半導体業界が将来にわたって好ましい方向へ進んでいかないのではないか，と筆者は思うのです．

前置きがたいへん長くなりました．本書では，IP化や設計再利用を意識したHDL記述のポイントについて解説します．次章以降のサンプル記述の中にも，そういった配慮をして記述したものが含まれています．

じつは筆者は，8，9年ほど前にも，こうしたIPを設計したことがあります．当時はまだIPということばが存在しておらず（あったのかもしれないが，少なくとも筆者は知らなかった），論理合成可能なモデルということで「シンセサイザブル・モデル」と呼んでいました．そのとき設計したものは8ビットのCPUモデルと，その周辺回路（ペリフェラル）である割り込みコントローラやパラレルI/Oなどでした．以下では，これらを作ったときの経験を交えながら，使いやすいIPを開発するコツについて紹介していきます．

1 柔軟なモデルを実現する“パラメタライズ”

IP化や設計再利用を意識してモデルを作る場合，HDLを記述するうえで避けて通れないテクニックが「パラメタライズ」です．これは，インスタンス化のとき（Verilog HDLであれば`module`の，VHDLであれば`entity`の“箱”を置くとき）に，例えばビット幅をパラメータとして与え，そのIPのビット幅をユーザが自由にカスタマイズできるようにするテクニックです（**図1.3**）．

パラメタライズの手法を用いていないと，そのIPをあるときは4ビット構成で，あるときは8ビット構成で使いたいという場合に，4ビットと8ビットの両方の下位モジュール（`module`または`entity`）を用意しなければなりません．このような方法をとっていたのでは，同じ種類の“箱”に対して，さまざまなモジュールを用意する必要があり，管理しきれなくなります．しかも，どこまでのビット幅のものを用意すればよいのか見当がつきません．同じ種類の“箱”に対して，パラメータを与えることによって，ビット幅を柔軟に変更できるしくみを作り込んでおけば，こうした問題を回避できます（**図1.4**）．

ただし，パラメタライズのテクニックを使うと，シミュレーションや論理合成を行う際に，パラメータの受け渡しのようすが設計者側からはっきり見えなくなることがあります．そのため，パラメータの受け渡しの記述そのものを禁止している会社や設計チームもあります．この手法を利用する場合には，あなたのプロジェクトでパラメタライズがどのように取り扱われているのかをよく確認してください．

見本

〔図1.3〕パラメータ入力でさまざまなモジュールができあがる

なにをパラメタライズにすればよいか

パラメタライズにする項目（パラメータに対応させる項目）として，大きく分けて以下の2種類があると筆者は考えています．

(1) ビット幅に対応するパラメータ

(2) 不要な機能（ファンクション）を削るためのパラメータ

このほかに，いくつかのパラメータの組み合わせによって，いろいろな機能を生成させるといった方法も考えられなくはないのですが，RTL記述でそこまで行うのは，現実にはなかなか難しいのではないかと思います．

ビット幅のパラメタライズはすぐにでも導入できる

まず(1)のビット幅に対応したパラメタライズですが，簡単な機能マクロであれば，これはすぐにでも導入できます．詳しい記述方法については，次章以降のサンプル記述の解説を参照していただきたいと思います．基本的には，Verilog HDLでは，

　parameter文，define文，for文

を，VHDLでは，

　generic文，constant文，for文，function文

を使用します．特に，パラメータをビット幅に対応させる場合には，繰り返しの記述に適したfor文を駆使することになると思います．

ところで，ビット幅に対応したパラメタライズといっても，単純にビット幅を増減させるだけとはかぎりません．例えばCPUやDSPなどのプロセッサにおいて，データのビット幅をパラメタライズにしたとします．このとき，命令コード（OPコード）をこのデータ・バスから取ってくるとすると，もともと16ビットだったデータが24ビットや32ビットになってしまうことがあります．そこで，必要なOPコードのデコードのやりかたや不必要なビットの処理（たとえば‘0’で埋めるとか）を考える必要があります．またアドレシング処理で，そのプロセッサが即値（immediate）などのアドレシングをサポートしている場合，パラメータの与えかたによって取り込むデータのビット幅が異なってきます．このあたりの考慮も必要になります．

このように，パラメタライズの手法をちょっとしたプロセッサやコントローラに適用するだけでも，HDL

〔図1.4〕インスタンス化してパラメータを与える例

ある機能の“箱”をインスタンス化するとき，ビット幅などのパラメータを与えることによって所望の仕様のモジュールが得られる．この方法を用いれば，すべてのビット幅に対して個別にモジュールを作成する必要がなくなる．管理や保守の面でも扱いやすい．

Verilog HDL

```
DFF #(8) IO(.C(CLK),.D(DATA_IN),.Q(DATA_OUT);
//
// Verilog HDLのパラメータの受け渡し
```

Verilog HDLのパラメータの受け渡しには“#”を使う

VHDL

```
IO:DFF generic map (WIDTH => 8)
--
-- VHDLのパラメータの受け渡し
   port map (C => CLK, D => DATA_IN, Q => DATA_OUT);
```

VHDLのパラメータの受け渡しはgeneric mapで

いずれも下位階層で，それぞれparameter，genericを使用していることが条件

見本

記述の知識だけでなく，プロセッサやコントローラそのものの知識が必要になります．

機能を削るためのパラメタライズの導入には検討が必要

次に(2)の，不要な機能を削るためのパラメタライズですが，筆者の経験では，これは(1)のビット幅に対応するパラメタライズと比べるとかなりめんどうです．単純に考えてもこちらのほうがめんどうだということは，おわかりいただけると思います．

前述した(1)で使用する構文以外に，Verilog HDLでは，

`` `ifdef ``文

を，VHDLでは，

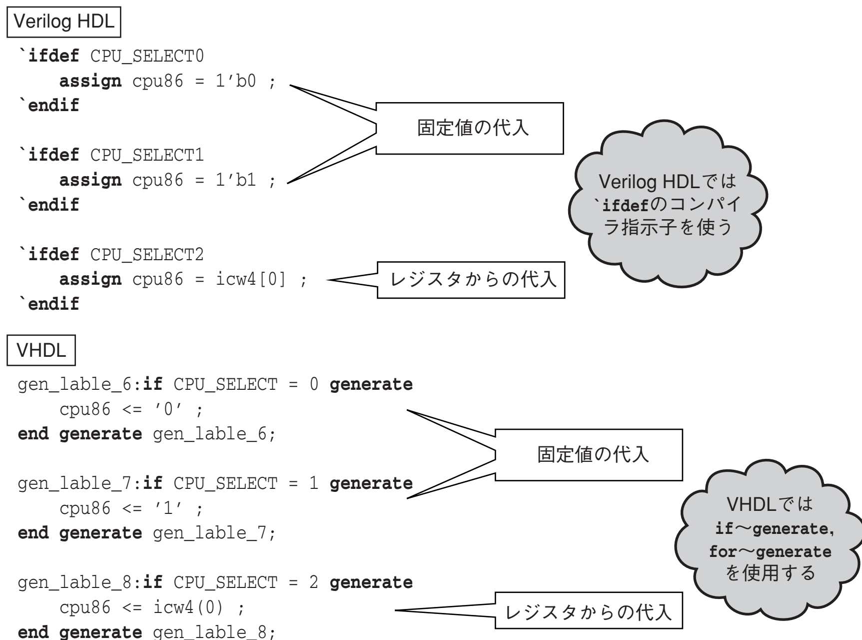
`for～generate`文，`if～generate`文

を使用することになると思います(**図1.5**)．特に，あらかじめHDL記述中に入っている機能を，パラメータを与えることによって削る場合，下位のモジュールのインスタンス化を行うかどうか，あるいは論理を生成するかどうかなどをコントロールする必要があります．

余談ですが，筆者が8，9年ほど前にパラメタライズの手法を利用してIPを作っていたころは，論理合成ツールがVerilog HDLの`` `ifdef ``文をサポートしていませんでした．そのため，Verilog HDLでインスタンス化をコントロールするのはけっこうたいへんでした．そのときにはどうしたかというと，変換プログラム(Cのプログラム)を作って，その変換プログラムにパラメータを与えると，ひな型となるHDLソースを

〔図1.5〕パラメータを与えて機能などを削る例

この例ではパラメータ(Verilog HDLでは`CPU_SELECT0`，`CPU_SELECT1`，`CPU_SELECT2`，VHDLでは`CPU_SELECT`に`0`，`1`，`2`のいずれかを指定)によって，信号`cpu86`に固定値(‘`0`’または‘`1`’)を代入するか，レジスタ(`icw4`の0ビット目)の値を代入するかを切り替えている．論理合成を行うと，`cpu86`が固定値になったほうが，回路的には小さくなることがわかる．つまり機能の削減を行っている．



見本

基に，所望の機能をもつHDLソースを生成するしくみを作りました（論理合成ツールが構文をサポートしていないからといって，すぐにあきらめてはいけない）．

面積の1/3，または500～1,000ゲートの削減が目安

表1.1に，筆者が当時作った8255相当のパラレルI/Oのコントローラに対する，ビット幅と機能をパラメタライズにした記述の合成結果を示します（元の米国Intel社の8255は非同期で設計されているが，筆者が設計したモデルはすべて同期設計にした）．ASICライブラリがかなり古くて申しわけないのですが，米国LSI Logic社のLSI10Kのものを用いた結果です．ここでは，mode_selectというパラメータが機能に対応し，port_widthというパラメータがビット幅に対応しています．

mode_selectについてもう少し説明を付け加えます．Intel社の8255は，三つのモードを持っており，モードによって外部ピンなどの役割が異なります．通常の8255は，コマンド・レジスタに値を設定することによってモードを指定します．ただし，8255を使用する場合，実際には一つのモードのみを使うことが多いようです．筆者はモード0のみを使用する場合がほとんどだろうと判断しました（このへんは，いろいろと異論があると思うが…）．そこで，これらのモードの中から不要なものを削れるように，パラメタライズの手法を適用しました．

表1.1を見ていただければわかるように，同一のモード（mode_selectの値が同じ場合）では，回路規模はビット幅（port_widthの値）に比例していると考えられます．一方，モードを変えて一部の機能を削除した場合には，このような関係がありません．モード1（mode_select=111）とモード2（mode_select=011）は，回路規模にそれほどの差はありません．これはモード1とモード2の論理回路がきわめて似ているからです．一方，モード0（mode_select=001）では，回路規模がモード1の半分になっています．

以上のように，機能の削除に対応したパラメタライズでは，せっかく苦労してしかけを組み込んでも，劇的にゲート数が変化するとはかぎりません．むやみにパラメータを増やしても，その組み合わせに対する検証の手間が増えるだけです．トータルのゲート数を改善できないパラメタライズは，得策とは言えません．

ある米国人のエンジニアに，「機能を削るためのパラメタライズは，せめて500ゲート以上の削減が行われないのであれば無意味だ」と言われたことがあります．これを聞いたとき筆者は，上述のモデルを作る前だったこともあって，「たとえわずかでもゲート数が減れば，それだけでも価値がある」と考えていました．その後，実際にモデルを作り，たかだか三つのパラメータの組み合わせに対するモデルの検証を行うだけで

〔**表1.1**〕**パラメタライズの手法を適用した8255相当の合成結果**

使用した論理合成ツールは米国Synopsys社のDesign Compiler．論理合成時の制約条件はmax_area=0のみ．回路ライブラリは米国LSI Logic社のLSI10Kを用いた．port_widthはポートのビット幅のパラメータ．なお，mode_selectのパラメータに対応する機能は以下のとおり．mode_select[2]が‘1’のとき，モード2に必要な論理を生成．mode_select[1]が‘1’のとき，モード1に必要な論理を生成．mode_select[0]が‘1’のとき，モード0に必要な論理を生成．

mode_select[2:0]	port_width	ゲート数	
111	8	1477	モード選択のパラメータを変化
011	8	1382	
001	8	795	
111	4	1268	ポート幅のパラメータを変化
111	16	2308	
111	32	4028	

見本

もかなりの手間がかかることを経験しました．そこで初めて，米国人のエンジニアの言ったことばの意味がわかりました．

プロセス技術が日夜進化しているので，具体的に何ゲート以上と言うことは難しいのですが，せめて元のモデルの1/3の面積の削減，あるいは500～1,000ゲート以上の削減が行われないのであれば，機能を削るためのパラメタライズの手法は採用するべきではないでしょう．

ではどうやって，回路規模を削減できるかどうかを見通せばよいのでしょうか．これは一概にこうだと言えないところがあります．つまりモデルのアーキテクチャや機能を熟知していないと，このような判断を下せないからです．加えて，熟知していたとしても，よく見通せない場合さえあります．筆者の経験からすると，やってみないとわからない部分が，けっこう多いような気がします．

2 優れたIPを作るには…

実際にIPを作るとき，場合によってはかなりプリミティブなレベルで（すなわちゲート・レベルに近い表現で）記述する必要があります．たとえば動作速度が問題になるIPでは，論理合成の制約条件の与えかたによって，回路の速度が大きく変化するのは好ましくありません．これは，合成用スクリプトなどをきちんとサポートする必要が出てくるためです．そこで，多少，記述の可読性を犠牲にしても，ゲート・レベルに近い表現で記述する必要があります．ソースの可読性がよくて，性能もきちんと満足できるIPを作ることができれば，それに越したことはないのですが…．

ユーザの協力なしによいIPは作れない

IPを作る場合，以下のような二つのパターンがあると思います．

(1) すでに世の中にある回路をIP化する．

(2) 新規の回路をIPとして作る．

(2)の新規の回路をIPとして作る場合は，モデルの作成をある程度開発側が主導できます．IPを利用するユーザからいろいろ状況を聞いて，モデルを改良していけばよいでしょう（ユーザ側にとっても新規のモデルなので，ある程度，開発側に協力しようという姿勢がある）．

ところが(1)のすでに世の中にある回路をIP化する場合は，そうはいきません．このようなモデルを作る際には，すでに出回っているマニュアルを参照したり，チップが存在するのであればそれを入手し，そのチップにいろいろと具体的な信号を入れて観測したりします．マニュアルに書かれていない動作などは，入力信号を入れても詳細がわからないことが少なくありません．そして，このようなマニュアルに書かれていない機能をアプリケーションが使っていることもあります．加えて，チップにさまざまなバージョンがあり，それぞれ動作が異なることさえあります．こうなるとIPを開発する側だけではお手上げで，IPを利用する側の情報がないと対処しきれません．つまりIPのユーザがIPを開発する側に協力しないと，実用的なIPを作れないということです．

見本

筆者がIPを作っていたときは，(1)のすでに世の中にある回路をIP化するパターンでした．そのため，IPのユーザ（顧客）に相当手間を取らせましたし，お世話にもなりました．それによって，ようやく，どうにかこうにか使えそうなものができあがったのです．ほとんどのことは調べつくされていても，ユーザの利用

〔図1.6〕IPの開発側も利用側も相手に協力してあげようという姿勢がない

環境で動かなければアウトです．つまり，そのIPがどれほど洗練されているものであったとしても，「ユーザにとってよいIP」とは言えないのです．

海外ではどういう状況なのか，筆者はよく知りませんが，日本の場合には，以下のような問題があると思います．

- **ユーザ側はIPを買うだけだから，自分たちの環境でそのIPが動作して当然だと考える（動かなければ，ただ文句を言うだけ）．**
- **IPプロバイダは，自分たちが想定していない方法でIPが使われた場合に，なんのサポートもしようとしない．**

つまり，IPを開発する側も利用する側も，相手に協力してあげようという姿勢が少ないような気がします．たんにそのIPは使えるか使えないかという点に議論が集中してしまい，そのIPを改善しようという動きが見られないのではないでしょうか（**図1.6**）．

よいIPを作るには，IPの開発側とユーザ側の協力が不可欠だと筆者は思います（もっとも，今のこのご時世ではなかなか難しいのかもしれないが…）．

“組み立てパソコン”の世界ではエンジニア商売もあがったり

今後，半導体のプロセス技術が進歩すればするほど，チップに搭載できる論理回路の規模は大きくなり，配線の量が増え，配線遅延の影響が増大するでしょう．したがって，アプリケーションで使わない不要な機能を，パラメータを与えることによって削るという手法は合理的だと筆者は考えます．ただし，前述したように，そういったモデルを作るのが難しいという問題もあります．

結論としては，まずはビット幅を可変にする目的にパラメタライズの手法を適用するのがよいと思います．これだけでも十分効果があります．ただし，ビット幅を可変にするときにも，それなりの苦労はあります．データなどのビット幅も含めて，どういった項目をパラメタライズにするのが適当であるかを決めるには，IPを利用する側の情報が必要になります．ですから，IPを利用する側でも，「自分たちもIPプロバイダになるんだ」というくらいの気概をもって取り組まないと，本当の意味でクオリティの高いIPは手に入らないのだと思います．

見本

組み立てパソコンの世界と同じで，だれが作っても同じモノ（LSI）ができる世の中になってしまうと，筆者のようなエンジニアはとっとと商売変えしたほうがよいのかもしれません．とりあえずは，みんなで切磋琢磨し，協力し合って，よいIPを生み出していきたいものです．

第2章 実用回路のサンプル記述

第2章ではVHDLとVerilog HDLで作成したサンプル記述を紹介します．フリップフロップやアップダウン・カウンタのような基本回路から，同期FIFO，アドレス・デコーダ，バス・インターフェースのような実用回路まで，27種類の回路についてVHDLとVerilog HDLの記述（機能記述とテストベンチ）を紹介します（**表2.1**）．すべての記述は付属のCD-ROMに収録されています．また，いくつかの記述については，第1章で紹介したパラメタライズの手法を適用しています．

〔表2.1〕第2章で扱うサンプル回路と付属CD-ROMに収録したHDLデータの一覧

サンプル回路	VHDLの機能記述	VHDLのテストベンチ	Verilog HDLの機能記述	Verilog HDLのテストベンチ
1. RSフリップフロップ	rsff.vhd, rsff_inst.vhd	rsff_test.vhd	rsff.v, rsff_inst.v	rsff_test.v
2. トランスペアレント・ラッチ	latch.vhd	latch_test.vhd	latch.v	latch_test.v
3. Dフリップフロップ	dff.vhd	dff_test.vhd	dff.v	dff_test.v
4. イネーブル付きDフリップフロップ	e_dff.vhd	e_dff_test.vhd	e_dff.v	e_dff_test.v
5. ロード付きアップダウン・カウンタ（非同期リセット）	counter1.vhd, counter2.vhd	counter_test.vhd	counter1.v, counter2.v	counter_test.v
6. ロード付きアップダウン10進カウンタ	bcdcnt.vhd	bcdcnt_test.vhd	bcdcnt.v	bcdcnt_test.v
7. マルチプレクサ，デマルチプレクサ	mpx.vhd, dmpx.vhd	mpx_test.vhd, dmpx_test.vhd	mpx.v, dmpx.v	mpx_test.v, dmpx_test.v
8. シフト・レジスタ	sftreg.vhd, srreg.vhd	sftrg_test.vhd	sftreg.v, srreg.v	sftrg_test.v
9. プライオリティ・エンコーダ	penc1.vhd, penc2.vhd	penc_test.vhd	penc1.v, penc2.v	penc_test.v
10. バレル・シフタ	bshift.vhd	bshift_test.vhd	bshift.v	bshift_test.v
11. 加算器（ハーフ・アダー，フル・アダー）	hadder.vhd, fadder.vhd	hadder_test.vhd, fadder_test.vhd	hadder.v, fadder.v	hadder_test.v, fadder_test.v
12. 加減算器	addsub.vhd, fadder.vhd	addsub_test.vhd	addsub.v fadder.v	addsub_test.v
13. 乗算器	mltp.vhd, hadder.vhd	mltp_test.vhd	mltp.v, hadder.v	mltp_test.v
14. ALU（数値演算ユニット）	alu.vhd	alu_test.vhd	alu.v	alu_test.v
15. FIFO（同期バス）	fifo_sync.vhd	fifo_sync_test.vhd	fifo_sync.v	fifo_sync_test.v
16. デュアル・ポートSRAM（非同期バス）	dpram_async.vhd[注]	dpram_async_test.vhd	dpram_async.v[注]	dpram_async_test.v
17. ISAバス・インターフェース・コントローラ＋スクラッチパッド・レジスタ	isa.vhd	isa_test.vhd	isa.v	isa_test.v

見本

〔表2.1〕第2章で扱うサンプル回路と付属CD-ROMに収録したHDLデータの一覧（つづき）

サンプル回路	VHDLの機能記述	VHDLのテストベンチ	Verilog HDLの機能記述	Verilog HDLのテストベンチ
18. パリティ・ジェネレータ，パリティ・チェッカ	user_func_pkg.vhd，ptygen.vhd，ptychk.vhd	pty_gen_chk_test.vhd	ptygen.v，ptychk.v	pty_gen_chk_test.v
19. 水平パリティ・ジェネレータ，水平パリティ・チェッカ	pty_gen_h.vhd，pty_chk_h.vhd	pty_h_test.vhd	pty_gen_h.v，pty_chk_h.v	pty_h_test.v
20. パルス・ジェネレータ	pg.vhd，pgi.vhd	pg_test.vhd	pg.v	pg_test.vhd
21. パラレル-シリアル・コンバータ	ps.vhd	sp_ps_test.vhd	ps.v	sp_ps_test.v
22. シリアル-パラレル・コンバータ	sp.vhd	sp_ps_test.vhd	sp.v	sp_ps_test.v
23. アラーム保護	hogo.vhd	hogo_test.vhd	hogo.v	hogo_test.v
24. フレーム同期検出	sync.vhd	sync_test.vhd	sync.v	sync_test.v
25. アドレス・デコーダ	adr_dec.vhd	—	adr_dec.v	—
26. クロック同期アドレス/データ多重バス・インターフェース	admux_busif.vhd，bus_reg.vhd，adr_dec.vhd	admux_bus_test.vhd	admux_busif.v，bus_reg.v，adr_dec.v	admux_bus_test.v
27. クロック同期アドレス/データ分離バス・インターフェース	busif.vhd，bus_reg.vhd，adr_dec.vhd	busif_test.vhd	busif.v，bus_reg.v，adr_dec.v	busif_test.v

注：ビヘイビア・モデル（シミュレーション用）．

1 RSフリップフロップ

- **作成者名：鳥海佳孝**
- **サンプル記述：リスト2.1（rsff.v，rsff_inst.v），リスト2.2（rsff.vhd，rsff_inst.vhd）**
- **モデルの種類：RTLモデル**
- **検証に使用したシミュレータ：VeriLogger（Verilog HDL），PeakVHDL（VHDL）**
- **端子表**
  入力：R，S
  出力：Q

RSフリップフロップ（**図2.1**，**図2.2**）は割り込み（インタラプト）の受け付けなどに使用されます．RSフリップフロップが取り扱う信号は非同期となるので，静的タイミング解析をともなう論理合成の際には注意が必要です．

検証だけならタスキがけの表現でOK

非同期入力のRSフリップフロップの記述は，ただ単純にRSフリップフロップの機能を示すだけ（つまり，合成しない）であれば，いわゆるタスキがけの表現でOKです．例えば，次のように記述します．

見本

〔リスト2.1〕RSフリップフロップのVerilog HDL記述（rsff.v，rsff_test.v，rsff_inst.v）

```
module RSFF_TEST;

parameter CYCLE = 100 ;
reg [3:0] R, S ;
wire [3:0] Q ;
wire Q0, Q0_tmp ;
integer I ;

RSFF_INST i0(R, S, Q);

initial
   begin
      for(I=0;I<=15;I=I+1)
          begin
             R=I;S=17-I;
             #CYCLE;
          end
   $finish;
   end

assign Q0 = ~(R[1] | Q0_tmp) ;
assign Q0_tmp = ~(S[1] | Q0) ;

initial
   $monitor($time,,"R0=%b S0=%b Q0=%b",
                                R[0],S[0],Q[0]);
endmodule

module RSFF_INST(R, S, Q);
parameter WIDTH = 4 ;
input [WIDTH-1:0] R,S;
output [WIDTH-1:0] Q ;

RSFF RS0(.CLK(S[0]), .RESET(R[0]), .D(1'b1)
                                       .Q(Q[0]));
RSFF RS1(.CLK(S[1]), .RESET(R[1]), .D(1'b1)
                                       .Q(Q[1]));
RSFF RS2(.CLK(S[2]), .RESET(R[2]), .D(1'b1)
                                       .Q(Q[2]));
RSFF RS3(.CLK(S[3]), .RESET(R[3]), .D(1'b1)
                                       .Q(Q[3]));

endmodule

module RSFF(CLK, RESET, D, Q);
input CLK, RESET ;
input D ;
output Q ;

reg Q ;

always @(posedge CLK or posedge RESET)
   begin
      if(RESET == 1'b1)
         Q <= 1'b0 ;
      else
         Q <= D ;
   end

endmodule
```

〔リスト2.2〕RSフリップフロップのVHDL記述（rsff.vhd，rsff_test.vhd，rsff_inst.vhd）

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

entity RSFF is
    port (CLK,RESET: in std_logic;
          D: in std_logic;
          Q: out std_logic
         );
end RSFF;

architecture RTL of RSFF is
begin

    process(CLK,RESET)
    begin
       if(RESET = ‘1’) then
          Q <= ‘0’;
       elsif(CLK’event and CLK=’1’) then
          Q <= D ;
       end if ;
    end process;

end RTL;

library IEEE;
use IEEE.std_logic_1164.all;

entity RSFF_INST is
    generic (WIDTH : integer := 4) ;
    port (R,S: in std_logic_vector(WIDTH-1
                                   downto 0);
          Q: out std_logic_vector(WIDTH-1
                                   downto 0)
         );
end RSFF_INST;

architecture RTL of RSFF_INST is
component RSFF
```

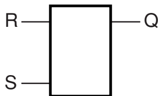
見本

〔リスト2.2〕RSフリップフロップのVHDL記述(rsff.vhd，rsff_test.vhd，rsff_inst.vhd)(つづき)

```
    port (CLK,RESET: in std_logic;
          D: in std_logic;
          Q: out std_logic
         );
end component;

constant VALUE_1 : std_logic := '1' ;
— VHDL'87のための修正
signal VALUE_1_SIG : std_logic ;

begin

— VHDL'87のための修正
VALUE_1_SIG <= VALUE_1 ;

GEN_RSFF:for I in 0 to WIDTH-1 generate
  RS: RSFF port map (CLK => S(I),RESET => R(I),
               D => VALUE_1_SIG, Q => Q(I));
end generate;
end RTL;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
entity RSFF_TEST is
end RSFF_TEST;

architecture STIMULUS of RSFF_TEST is
component RSFF_INST
    generic (WIDTH : integer := 4) ;
    port (R,S: in std_logic_vector(WIDTH-1
                                       downto 0);
          Q: out std_logic_vector(WIDTH-1
                                       downto 0)
         );
end component;

constant WIDTH : integer := 4 ;
constant PERIOD : time := 100ns ;
signal R,S : std_logic_vector(WIDTH-1 downto 0);
signal Q : std_logic_vector(WIDTH-1 downto 0);

begin

U0:RSFF_INST generic map (WIDTH => 4)
          port map (R => R, S => S, Q => Q);

process
variable J : integer ;
begin
   for I in 0 to 15 loop
       if (I = 0 or I = 1) then
           J := 1 - I ;
       else
           J := 17 - I ;
       end if ;
       R <= conv_std_logic_vector(I,4);
       S <= conv_std_logic_vector(J,4);
       wait for PERIOD;
   end loop ;
   wait ;
end process;

end STIMULUS;
```

〔図2.1〕
RSフリップフロップのブロック図

入力信号はR(リセット入力)とS(セット入力)，出力信号はQ(RSフリップフロップの出力)．



〔図2.2〕**RSフリップフロップのタイム・チャート**

1ビットのRSフリップフロップを四つインスタンス化して行ったシミュレーションの結果である．それぞれのRSフリップフロップに対して，リセット入力とセット入力をテストベンチとして作成した4ビット・カウンタで与えている．本モデルはリセット優先なので，リセット入力とセット入力が同時に入力された場合，リセットになるように設計されている．シミュレーション結果もそれを示している．

Q	0001	0000	1101	1100	1001
R	0000	0001	0010	0011	0100
S	0001	0000	1111	1110	1101

見本

● Verilog HDL
```
assign Q = ~(R | Q_tmp) ;
assign Q_tmp = ~(S | Q) ;
```
● VHDL
```
Q <= not (R or Q_tmp) ;
Q_tmp <= not (S or Q) ;
```

ただし，上記のような記述スタイルは，論理合成を意識した記述（RTL記述）としては適当と言えません．この理由として，上記の記述では回路にループが含まれているため，パスの遅延解析（静的タイミング解析）などを行いにくい，または行えなくなってしまうからです．そのため，ASICの設計では，上記のようなタスキがけの回路の使用を禁止している場合もあります．

Dフリップフロップを使用する場合にはタイミング制約に注意

では，論理合成を意識して記述する場合は，どのようにすればよいのでしょうか．方法は2通りあります．

(1) 使用するASICなどのライブラリを利用する．

(2) 非同期リセットのDフリップフロップを利用する．

まず(1)の方法ですが，使用するライブラリの中に非同期のRSフリップフロップがあれば，これを使用するのが無難です．なぜなら，ASICベンダがいろいろな意味でこの回路の動作を保証してくれるからです．実際に記述する場合，このRSフリップフロップのモジュール名（Verilog HDL），ないしはエンティティ名（VHDL）をインスタンス化して利用します．

もう一つの(2)の方法は，通常の非同期リセットのDフリップフロップを使用して，Rの信号を非同期リセット端子に，Sの信号をCLK端子に接続します．このとき，D入力には固定値‘1’を入れておきます．

機能的にはこれでOKです．ただし，この記述を直接RTLデータの中に入れてしまうと，論理合成時にタイミング制約をコントロールするのがめんどうになります．つまり，CLK端子にシステム・クロック以外の信号（Sの信号）が入力されているので，論理合成のタイミング制約を与えるときなどに，この部分に対する配慮が必要となるのです．そこで，あらかじめこのRSフリップフロップに相当するDフリップフロップは，一つの階層を設けて“箱”にしておき，その箱をインスタンス化したほうが取り扱いやすくなります．

2　トランスペアレント・ラッチ

- 作成者名：鳥海佳孝
- サンプル記述：リスト2.3（latch.v），リスト2.4（latch.vhd）
- モデルの種類：RTLモデル
- 検証に使用したシミュレータ：VeriLogger（Verilog HDL），PeakVHDL（VHDL）
- 端子表　　入力：G，D　　出力：Q
- パラメータ：WIDTH（D，Qのビット幅）

見本

Verilog HDLの場合もVHDLの場合も，トランスペアレント・ラッチ（**図2.3**，**図2.4**）のRTLの記述スタイルは組み合わせ回路とよく似ています．Verilog HDLでは，always文中でif文とcase文のいずれかを用いた場合，その参照している条件が不完全ならば，論理合成ツールはラッチを生成してしまいます．VHDLでは，process文中でif文を用いた場合，その参照している条件が不完全ならば，論理合成ツールはラッチを生成してしまいます．自分では組み合わせ回路を正しく記述したつもりでも，組み合わせ回路の論理が複雑で深くなってしまうと，条件を完結しきれずにラッチを生成させてしまうことがよくあります．

論理合成ツールがラッチをチェック

このような誤りを防ぐためのチェック機能をサポートしている論理合成ツールもあります．下記の例は，米国Synplicity社のFPGA用論理合成ツール「Synplify」を使用した例です．トランスペアレント・ラッチのVHDL記述をSynplifyにかけると合成結果に問題があったことを示し，次のようなワーニング・メッセージがログ・ファイルに現れます．

〔リスト2.3〕トランスペアレント・ラッチのVerilog HDL記述（latch.v，latch_test.v）

```
module LATCH(G, D, Q);
parameter WIDTH = 1 ;
input G ;
input [WIDTH-1:0] D ;
output [WIDTH-1:0] Q ;

reg [WIDTH-1:0] Q ;

always @(G or D)
   begin
      if (G == 1'b1)
          Q <= D ;
   end

endmodule

module LATCH_TEST;

parameter LATCH_WIDTH = 4 ;
parameter CYCLE = 100 ;

reg CLK ;
reg [LATCH_WIDTH-1:0] DIN ;
wire [LATCH_WIDTH-1:0] DOUT ;
integer I ;

LATCH #(LATCH_WIDTH) I0(.G(CLK), .D(DIN),
.Q(DOUT));

always #(CYCLE/2)
   CLK = ~CLK ;

initial
   begin
      CLK = 1'b0 ;
      #(CYCLE/4);
      for (I=0;I<=15;I=I+1)
          begin
             if (I == 8)
                 #(CYCLE/2);
             DIN <= I ;
             #CYCLE;
          end
      $finish;
   end

initial
   $monitor($time,,"CLK=%b DIN=%b
DOUT=%b",CLK,DIN,DOUT);

endmodule
```

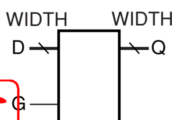
見本

〔リスト2.4〕トランスペアレント・ラッチのVHDL記述（latch.vhd，latch_test.vhd）

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

entity LATCH is
    generic (WIDTH : integer := 1) ;
    port (G: in std_logic;
          D: in std_logic_vector(WIDTH-1
                                 downto 0);
          Q: out std_logic_vector(WIDTH-1
                                 downto 0)
         );
end LATCH;

architecture RTL of LATCH is
begin

    process(G,D)
    begin
       if(G ='1') then
          Q <= D ;
       end if ;
    end process;

end RTL;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity LATCH_TEST is
end LATCH_TEST;

architecture STIMULUS of LATCH_TEST is
component LATCH
    generic (WIDTH : integer) ;
    port (G: in std_logic;
          D: in std_logic_vector(WIDTH-1
                                 downto 0);
          Q: out std_logic_vector(WIDTH-1
                                 downto 0)
          );
end component;
constant LATCH_WIDTH : integer := 4 ;
constant CYCLE : time := 100ns ;
signal G : std_logic := '0' ;
signal D,Q : std_logic_vector(LATCH_WIDTH-1
                                 downto 0) ;
begin

U0: LATCH generic map(WIDTH => LATCH_WIDTH)
          port map(G => G, D => D, Q => Q);

process
begin
   wait for CYCLE/2 ;
   G <= not G ;
end process;

process
variable J : integer ;
begin
   wait for CYCLE/4;
   for I in 0 to 15 loop
      if (I = 8) then
         wait for CYCLE/2;
      end if ;
      D <= conv_std_logic_vector(I,LATCH_WIDTH);
      wait for CYCLE;
   end loop;
   wait ;
end process ;

end STIMULUS;
```

〔図2.3〕
トランスペアレント・ラッチのブロック図

入力信号はG（クロック入力）とD［WIDTH-1：0］（データ入力），出力信号はQ［WIDTH-1：0］（トランスペアレント・ラッチの出力）．WIDTHは任意に与えられる整数値で，DとQのビット幅を決める．



見本

〔図2.4〕トランスペアレント・ラッチのタイム・チャート

トランスペアレント・ラッチのシミュレーション結果の一部を示す．WIDTHの値は4にしている（4ビットのトランスペアレント・ラッチとして動作している）．入力には，テストベンチとして作成した4ビット・カウンタの値を与えており，その値が所望のタイミングで出力されているかどうかを確かめている．入力が7以下のときは，Gに入力されているクロックが“L”の期間にDの入力を変化させている．入力が8以上のときはGのクロックが“H”の期間にDの入力を変化させている．つまりトランスペアレント・ラッチは，Gのクロックの“H”の期間にデータ入力が変化した場合，そのデータを伝搬する．