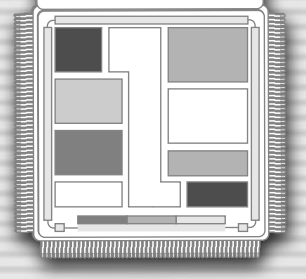


第2回

IPコア設計と検証における指針

森岡澄夫，高野光司，大庭信之

前回(本誌2002年8月号)紹介したように，IPコアへの要求の多様化に伴い，設計や検証の方法をケース・バイ・ケースでくふうしていく必要性が高まっています．そこで今回は，さまざまなIPコアに共通する設計や検証の指針について解説します．また，将来のIPコア設計の技術について，筆者らがどのように考えているかを説明します．（筆者）

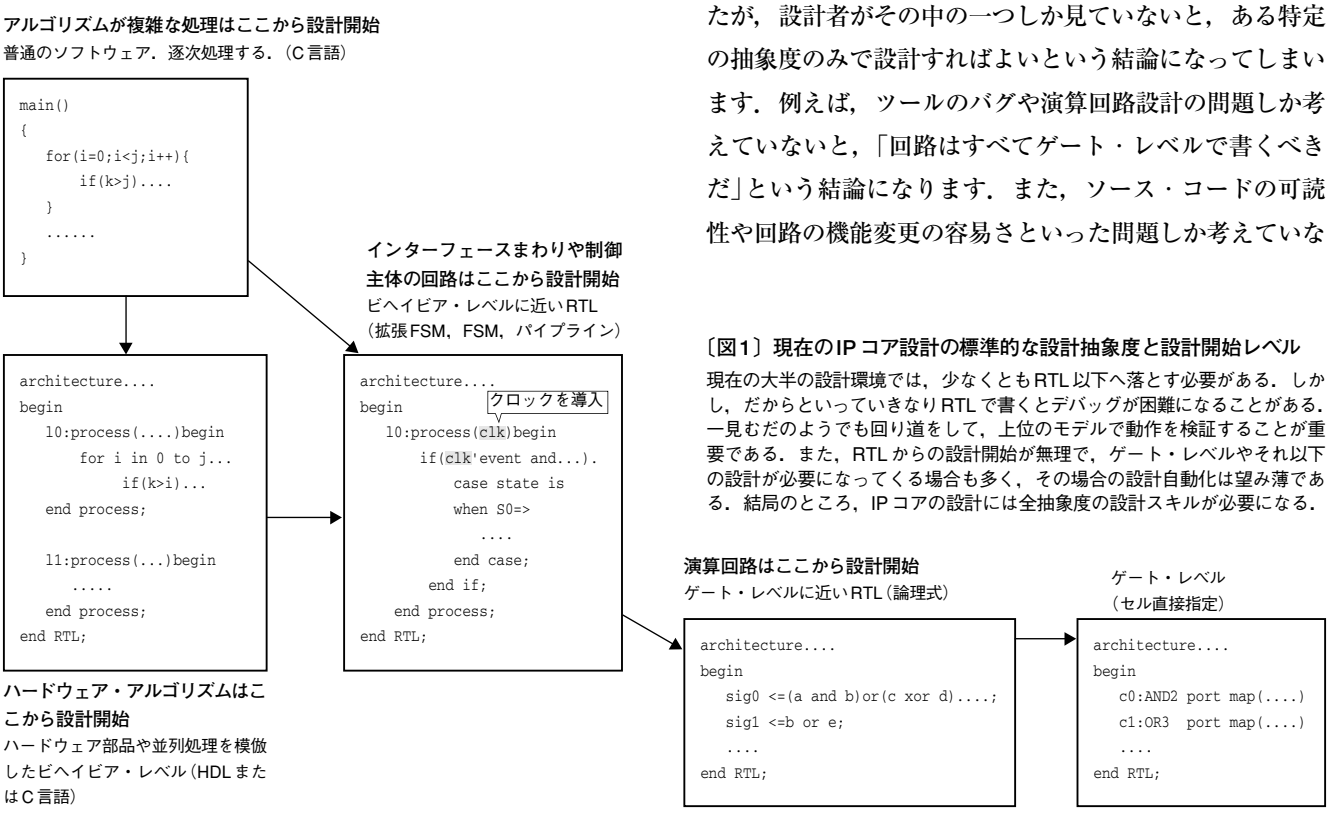
まず初めに，IPコアの設計について，筆者らが心がけているいくつかの指針を紹介します．

適切な抽象度で設計を行う

設計する回路の処理内容や要求性能に応じて，適切な設計抽象度を選んで設計を始めることが重要です．ビヘイビア・レベルだけ，RTLだけ，ゲート・レベルだけというように，一つの抽象度だけですべてを設計・検証しようとするのはナンセンスですし，実際問題としても無理な話です．

●単一の抽象度ですべてを設計・検証するのは困難

前回，IPコア設計におけるさまざまな課題を紹介しましたが，設計者がその中の一つしか見ていないと，ある特定の抽象度のみで設計すればよいという結論になってしまいます．例えば，ツールのバグや演算回路設計の問題しか考えていないと，「回路はすべてゲート・レベルで書くべきだ」という結論になります．また，ソース・コードの可読性や回路の機能変更の容易さといった問題しか考えていな



〔図1〕現在のIPコア設計の標準的な設計抽象度と設計開始レベル
現在の大半の設計環境では，少なくともRTL以下へ落とす必要がある．しかし，だからといっていきなりRTLで書くとデバッグが困難になることがある．一見むだのようでも回り道をして，上位のモデルで動作を検証することが重要である．また，RTLからの設計開始が無理で，ゲート・レベルやそれ以下の設計が必要になってくる場合も多く，その場合の設計自動化は望み薄である．結局のところ，IPコアの設計には全抽象度の設計スキルが必要になる．

いと，「回路はすべてビヘイビア・レベルで書くべきだ」という結論になります．いずれも極端な考えかたで，現実的ではありません．実際には，自分の設計でいちばん優先度が高いものは何かを考え，バランスをとって臨機応変に設計抽象度を使い分けることがたいせつです．

●回路の種類によって設計開始の抽象度を変える

設計を開始する設計抽象度の選択について，以下に筆者らのガイドラインを示します．これは多くのディジタル回路設計者に共通であると思います（**図1**）．

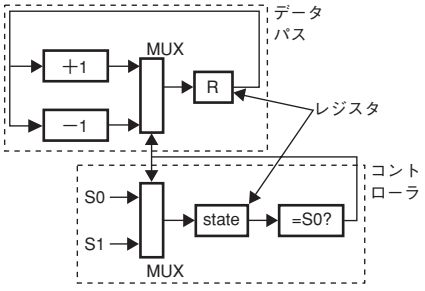
1）ツール（回路ライブラリ）に組み込まれていない演算回路，あるいはチップ・インターフェースの中で特にタイミング・クリティカルな部分は，ゲート・レベルから設計を開始する（ハード・コード化[注1]する場合も多い）．それ以外の設計抽象度では設計できないのでしかたがない．

2）チップ・インターフェース（wrapperを含む）や通信プロトコルなど制御が主体のものは，RTLから設計を開始する．これらの仕様は，普通，タイミング・チャートなどでクロックごとの動作が規定されるので，RTL以下で設計せざるをえない．一方，いきなりゲート・レベルで設計を始めると修正が困難になるので，それはなるべく避けたい．RTLでプロトコルの大筋をデバッグしてから，必要に応じてゲート・レベルに落として回路をチューンアップするというケースが経験上もっとも多い．

注1：セルなどの使用プリミティブを直接指定する．

〔図2〕RTL記述のスタイルによる抽象度の切り替え

目標とする回路が同じでも，RTL記述のしかたは複数ある．そのポイントの一つは，部品やそれらの結線（signal）を明示的に書くかどうかである．（b）はレジスタやセレクタを明示せず（ツールに生成させる），しかも部品間の結線をsignal宣言しない記述である．一方，（e）は部品やそれらの接続を完全に固定した記述である．（c）や（d）は中間的な記述である．どれが良くてどれがいけないというのではなく，それぞれ利害得失があるので目的別に切り替えて使う．



状態S0のときレジスタRをインクリメント，S1のときデクリメント

（a）サンプル回路

```
レジスタR，stateへの代入値を書く
10: process (clk) begin
    if (clk' event and ...) ...
        case state is
        when S0 =>
            R <= R + 1;
            state <= S1;
        when S1 =>
            R <= R - 1;
            state <= S0;
        end case;
    end if;
end process;
```

（b）モデリング手法①

```
コントローラの記述
10: process (clk) begin
    if (clk' event and ...) ...
        case state is
        when S0 =>
            state <= S1;
        when S1 =>
            state <= S0;
        end case;
    end if;
end process;

state_s0 <= '1' when state = S0 else '0' ;
```

```
データパスの記述
11: process (clk) begin
    if (clk' event and ...)...
        if (state_s0 = '1' ) then
            R <= R + 1;
        else
            R <= R - 1;
        end if;
    end if;
end process;
```

（c）モデリング手法②

```
レジスタstateの記述
10: process (clk) begin
    if (clk' event and ...) ...
        state <= next_state;
    end if;
end process;

レジスタRの記述
11: process (clk) begin
    if (clk' event and ...)...
        R <= next_R;
    end if;
end process;

演算回路やセレクタの記述
inc_R      <= R + 1;
dec_R      <= R - 1;
next_R     <= inc_R when state_s0 = '1' else dec_R;
state_s0   <= '1' when state = S0 else '0' ;
next_state <= S1 when state_s0 = '1' else S0;
```

（d）モデリング手法③

```
全部品を下位の回路ブロックにして呼び出し

r0: REG   port map (next_state, clk, state);
c0: CMP   port map (state, S0, state_s0);
s0: MUX   port map (S1, S0, state_s0, next_state);
r1: REG   port map (next_R, clk, R);
i0: INC   port map (R, inc_R);
d0: DEC   port map (R, dec_R);
s1: MUX   port map (inc_R, dec_R, state_s0, next_R);
```

（e）モデリング手法④