情報工学実験 II

第 2 回： 加減算器の実装

2018 年 12 月 6 日

柴田 裕一郎（shibata@cis.nagasaki-u.ac.jp）

1 加算器の記述

今回はもう少し実用的な組合せ回路を取り扱います．まずは 4 ビットの加算器（加算をする組合せ回路）を設計してみましょう．

ソースコード 1 に 4 ビット加算器の記述を示します．算術演算子「+」を使うことで，加算の部分は

```
sout = ain + bin;
```

のように簡単に記述できます．一口に加算器といっても，リプルキャリー型やキャリールックアヘッド型などさまざまな種類があります．しかし，ここではどんな種類の加算器を使うかについては一切記述していないことに注意してください．あくまで「加算をする」ということだけを記述しています．このように，実装に関する細かいことは後で決めることにし，とりあえず大事なこと（ここでは「加算する」こと）だけに着目した設計を**抽象度の高い**設計といいます．設計の抽象度を高くできることがハードウェアを言語で設計する大きなメリットのひとつです．どんな加算器を使用するのかは，論理合成時に必要な性能や使用できる面積などの制約をツールに与えることで，論理合成ツールに自動的に選択させることができます．

さて，このモジュールの 2 つの入力（ain と bin）と出力（sout）はそれぞれ 4 ビット幅のバスです．そこで，

```
output wire [3:0] sout
```

のように，「[3:0]」という記述を付加して 4 ビットのバスであることを示しています．SystemVerilog のバスは C 言語の配列と同様に 0 オリジンになっており，「[3:0]」と宣言すると，「sout[0]」（0 ビット目）から「sout[3]」（3 ビット目）の合計 4 ビットの信号が使えるようになります．「sout[0]」が LSB（Least Significant Bit: 最下位ビット），「sout[3]」が MSB（Most Significant Bit: 最上位ビット）です．また，例え

ソースコード 1　4 ビット加算器の記述例（add4.sv）

```
1  `default_nettype none
2  // 4-bit adder
3  module add4
4    (
5     input wire [3:0]  ain, bin,
6     output wire [3:0] sout
7     );
8
9     assign sout = ain + bin;
10 endmodule
11 `default_nettype wire
```

ば「sout[3:1]」と記述すると，3 ビット目から 1 ビット目までの 3 ビットを切り出したことを表します．

2 乱数を用いたシミュレーションによる検証

2.1 テストベクタ

一度 LSI を製造してしまうと多額の費用がかかるうえ，製造後に論理を修正するのはほとんど不可能なため，シミュレーションの段階ですべてのバグは取っておきたいものです．また，LSI は設計が完璧でも，半導体の特性のばらつきや不純物などの関係から製造過程でかならず不良品が発生します．これを効率よくチェックすることも必要です．

しかし，入力の組み合わせの数はビット幅に応じて指数的に増えるので，少し規模が大きくなると単純な組み合わせ回路であっても，すべての入力の組み合わせを試すことが難しくなります．回路をテストするための入力信号の列をテストパターンとか**テストベクタ**と呼びます．効率の良いテストベクタをいかに作るのかという問題は，現在でも盛んに研究されている分野のひとつです．あらかじめ LSI の中にテストのための回路を埋め込んでおき，自動で自分自身をテストするという方式（BIST: Built In Self Test）も研究されています．

この実験ではもう少し簡単にテストをするために，検証のためのテストベクタを乱数を用いて作ることにします．乱数ごときにハードウェアの検証を任せられるのかと少し心配になるかもしれません．もちろん完全にはできません．しかし，乱数による検証はかなり有効であることが経験的に知られています．実際の設計現場でもよく使われている手法です．

2.2 シミュレーションモジュール

乱数でテストベクタを与えるシミュレーションモジュールの例をソースコード 2 に示します．2 つの入力ポート（ain と bin）に乱数を与えて出力を観察する動作を 50 回繰り返しています．概ね前回説明したシミュレーションモジュールと同様の構造を持っていますが，以下に新しく登場した文法事項を中心に説明します．

■多ビットの変数・配線（5 行目・6 行目） 今回の検査対象モジュールの入力・出力は 4 ビット幅なので，それに合わせて，4 ビット幅の logic や wire を定義しています．入出力ポートと同様に「[3:0]」という記述を付加することで 4 ビット幅であることを示します．

■整数変数（11 行目） 今回のシミュレーションモジュールでは，同じ動作を 50 回繰り返すために for 文を使っています．そのために initial ブロックの中で整数の変数 i を定義しています．ほぼ C 言語と同じ文法ですが，SystemVerilog では int 型ではなく integer 型を使います．

■for 文（13 行目〜18 行目） シミュレーション記述では C 言語と同様に for 文が使えます．この例では変数 i を 1 ずつ増やしながら 50 回繰り返します．C 言語では中括弧（「{」と「}」）で繰り返す部分のブロックを示しますが，SystemVerilog ではキーワード「begin」と「end」を使います．

■乱数先生（14 行目・15 行目） SystemVerilog ではシステムタスク $urandom_range を使って符号なし整数（つまりすべて正の数と考える）の乱数を生成できます．引数には生成する乱数の範囲を指定します．この例では，0 以上 15 以下の範囲の整数をランダムに生成し，ain と bin に代入しています．この代入は当然手続的代入です．ちなみに，urandom_range の「u」は unsigned（符号なし）の意味です（と思います）．

ソースコード 2　シミュレーションモジュールの記述例（sim_add4.sv）

```
1  `default_nettype none
2  `timescale 1ns/1ps
3  // Simulation module for add4
4  module sim_add4();
5     logic [3:0] ain, bin;
6     wire [3:0]  sout;
7
8     add4 add4_inst(.ain(ain), .bin(bin), .sout(sout));
9
10    initial begin
11       integer i;
12
13       for (i = 0; i < 50; i++) begin
14          ain <= $urandom_range(0, 15);
15          bin <= $urandom_range(0, 15);
16          #10
17            print();
18       end
19       $finish;
20    end
21
22    task print();
23       $write("ain= %d bin= %d sout= %d\n", ain, bin, sout);
24    endtask
25 endmodule
26 `default_nettype wire
```

2.3　C プログラムによるシミュレーションログの検証

このシミュレーションモジュールを用いてシミュレーションを実行すると，次のようなメッセージが画面に表示されます．また，同じ内容がログファイル（sim_add4.log）にも保存されます．

```
# vsim -do {add wave -r /sim_add4/add4_inst/*; run -all} -l sim_add4.log -c -wlf sim_add4.wlf sim_add4
# Loading sv_std.std
# Loading work.sim_add4
# Loading work.add4
# add wave -r /sim_add4/add4_inst/*
#  run -all
# ain=  3 bin= 11 sout= 14
# ain=  6 bin=  1 sout=  7
# ain=  1 bin=  4 sout=  5
# ain=  3 bin= 14 sout=  1
# ain=  5 bin= 10 sout= 15
.... 以下略 ....
```

これを見ながら，「3 + 11 = 14」，「6 + 1 = 7」，「1 + 4 = 5」…という具合に，加算が正しく行われているかを確認することができます．しかし，このように目視で 50 回分のチェックを行うのはかなりの根性が必要ですし，ビット幅が広くなってテストベクタが数万行などとなると，もはや不可能になります．第一，目視によるチェックではエラーを見逃しまう可能性が高く，信頼できません．そこで，シミュレーションログを自動的に

チェックするプログラムを C で書いて作ることにします．

この C プログラムの例はソースコード 3 の通りです．ログファイルのフォーマットに合わせて単純に ain, bin, sout の値を取り込み，期待値を計算して実際にファイルから読込んだシミュレーションの出力値とチェックします．

ソースコード 3　検証用プログラム（check_sim_add4.c）

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define BUF_SIZE 1024

int main(int argc, char *argv[])
{
  FILE *fp;
  char buf[BUF_SIZE];
  int  lineNumber = 1;    /* 行数カウンタ */
  int  checkedLines = 0;  /* 検査済みテストベクタ数 */

  /* ログファイルが指定されていなければ終了 */
  if (argc != 2) {
    fprintf(stderr, "error: ログファイルが未指定\n");
    exit(1);
  }

  /* ログファイルがオープンできなれば終了 */
  if ((fp = fopen(argv[1], "r")) == NULL) {
    fprintf(stderr, "error: ファイルオープン失敗: %s\n", argv[1]);
    exit(1);
  }

  /* ログファイルから 1行ずつ読み期待値と比較 */
  while (fgets(buf, BUF_SIZE, fp) != NULL) {
    /*「# ain=」で始まる行だけ処理 */
    if (strncmp(buf, "# ain=", 6) == 0)  {
      int ain, bin, sout, expectedValue;

      /* 入力値と出力値の読み込み */
      ain  = atoi(buf + 6);
      bin  = atoi(buf + 14);
      sout = atoi(buf + 23);

      /* 期待値の計算 */
      expectedValue  = (ain + bin) & 0xf;

      /* 出力と期待値を比較し異なれば終了 */
      if (sout == expectedValue)
        checkedLines++;
      else {
        fprintf(stderr, "error: %d 行目: ain=%d bin=%d sout=%d 期待値=%d\n",
                lineNumber, ain, bin, sout, expectedValue);
        exit(1);
      }
    }
    lineNumber++;
  }

```

```
51   /* 検査したテストベクタ数が0なら終了 */
52   if (checkedLines == 0) {
53     fprintf(stderr, "error: テストベクタが見つかりませんでした\n");
54     exit(1);
55   }
56
57   /* 検査したテストベクタ数を表示して正常終了 */
58   printf("success: 検査テストベクタ数: %d\n", checkedLines);
59   return (0);
60 }
```

2.4 検証の流れ

シミュレーション実行後，この検証用プログラムを

```
% gcc -o check_sim_add4 -Wall -O2 check_sim_add4.c
```

とコンパイルし，

```
% ./check_sim_add4 sim_add4.log
```

とシミュレーションログのファイル名を引数に与えて実行すればよいのですが，配布した Makefile を使えば，

```
% make check
```

とすることで，これらを自動的に行えます．また，

```
% make sim check
```

とすることで，シミュレーションの実行と，シミュレーション結果のチェックを一息に行うこともできます．すべてのテストベクタの検証がパスし，

```
success: 検査テストベクタ数: 50
```

のように出されれば OK です．もし期待値とシミュレーション結果に不一致が生じた場合には，

```
error: 27 行目: ain=3 bin=4 sout=12 期待値=7
make: *** [check] エラー 1
```

のように表示されますので，設計を見直す必要があります．以上の設計検証の流れを図 1 にまとめます．

3 加算器から加減算器へ

さて，加算ができれば次は減算をやりたくなりますが，減算は専用の減算器を使ったりはしないのが普通です．これは，2 の補数をとって加算する方式を使えば，減算器は加算器に少し回路を付加するだけで実現できるからです．A から B を引く演算（$A - B$）は以下の手順で処理できます．

（1）B の ‘1’ と ‘0’ をすべて反転させる
　すべての桁が ‘1’ である数から B を引いた数，すなわち「B の 1 の補数」ができます．

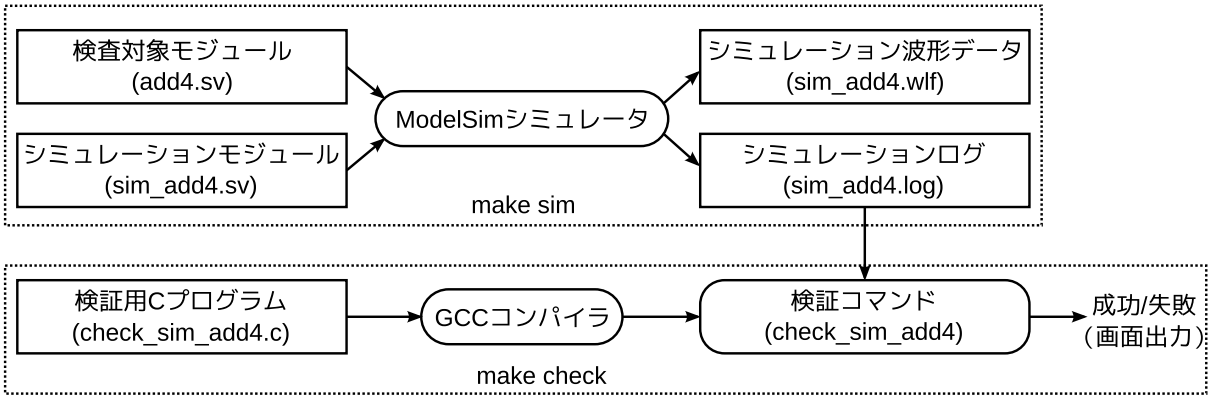

図 1　設計検証の流れ

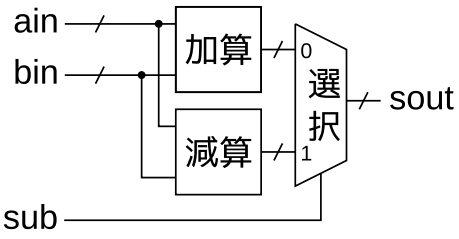

図 2　加減算器の論理的構造

（2）この数に 1 を足す

1 の補数より 1 大きい数，つまりすべての桁が ‘0’ で，はみ出した最上位の桁が ‘1’ である数から B を引いた数，すなわち「B の 2 の補数」ができます．

（3）この「B の 2 の補数」と A の加算を行う（ただし桁上げは無視する）

「B の 2 の補数」と A の加算は $A - B$ を意味します．

このように加算器と減算器では，かなりの部分を共通に使うことができます．このため，1 つのハードウェアで加算と減算を切り替えて使うことができる**加減算器**がよく用いられます．

3.1　加減算器の論理構成と記述

加減算器の論理的な構成を図 2 に示します．加算をさせるのか減算をさせるのかを指定する 1 ビットの sub という入力端子を新たに付け加えました．この sub に ‘1’ を入力したときには減算が，‘0’ を入力したときには加算が行われます．入力は 2 つに分岐し，片方は加算器，もう片方は減算器を通ってマルチプレクサ（図中の「選択」の部分）に入力されています．**マルチプレクサ**（**データセレクタ**ともいいます）は複数の入力のどれかひとつを選択して出力するハードウェアのことで，この例では sub によって 2 つの入力の切り替えを行います．

SystemVerilog でも，基本的に図 2 の論理構造を記述すればよいことになります．ただし，これでは加算器と減算器が別々に存在し，「加算器と減算器は似ているのでかなりの部分を共通化できる」という構造にはなっていません．しかし，そのような細かいハードウェアの簡単化についてはあまり気にせず，ツールに任せてしまおう，ということができるのが HDL 設計の利点です．

ソースコード 4　4 ビット加減算器の記述例

```
1  `default_nettype none
2  // addsub4: addition when sub = 0, subtraction when sub = 1
3  module addsub4
4    (
5     input wire [3:0]  ain, bin,
6     input wire        sub,
7     output wire [3:0] sout
8     );
9
10    assign sout = sub? ain - bin: ain + bin;
11 endmodule
12 `default_nettype wire
```

図 2 の論理構造をストレートに SystemVerilog で記述した例をソースコード 4 に示します．10 行目の assign 文で三項条件演算子（?:）を使うことで，継続的代入によりマルチプレクサの動作を記述しています．

3.2　if 文を利用した記述

C 言語に慣れていると，マルチプレクサの部分を if 文を使って書きたいと思う人も多いでしょう．もちろん書けるのですが，**if 文は継続的代入（assign 文）では使えない**ことに注意する必要があります．なぜかを厳密に説明すると話がややこしくなるので簡単に説明します．継続的代入（assign 文）というのは「ずっと接続されている」ことを示すものでした．しかし，if 文は文法的には else 節を省くことが認められているので，条件が不成立の際には「何も代入されない」という記述が可能です．というわけで，assign 文の中に if 文を書いてしまうと，「ずっと接続されている」のに「何も代入されない」という矛盾した記述が可能になってしまいます．このような理由から禁止されているのです．

ではどすれば良いのかというと，手続的代入を使って組み合わせ回路を記述します．シミュレーションモジュールで手続的代入を使うためには initial 文を使いましたが，SystemVerilog には組み合わせ回路を手続的に記述するための always_comb 文という構文が用意されており，これを使います．always_comb の comb は combination circuit（組み合わせ回路）の略です．この構文を用いた加減算器の記述例をソースコード 5 に示します．ソースコード 4 と比べると以下の点が変更されています．

■logic 型出力（7 行目）　出力 sout の型が wire から logic に変更されています．これは sout に if 文を用いた手続的代入を行うためです．前回も説明したように，wire は配線を表しており継続的代入しか許されていません．そこでこの設計では sout を logic にしています．

■always_comb ブロック（10 行目〜15 行目）　組み合わせ回路の動作を手続的に記述するためのブロックです．always_comb 文は組み合わせ回路への入力（各代入文の右辺に現れる信号）が変化する度に起動され，begin から end までの文が順番に実行されます．

■if 文（11 行目〜14 行目）　if 文の文法は C 言語と同様です．ただし，if 節や else 節に複数の文を記述したい場合には，中括弧（「{」と「}」）ではなく「begin」と「end」を用いてブロックの範囲を表現します．また，組み合わせ回路なので，出力変数（ここでは sout）には条件によらず always_comb ブロックの中で必ず何らかの値が代入される必要があります．したがって，ここでは else 節を省くことはできません．

ソースコード 5　手続的代入を使った加減算器の記述例

```
1  `default_nettype none
2  // addsub4: addition when sub = 0, subtraction when sub = 1
3  module addsub4
4    (
5     input wire [3:0]   ain, bin,
6     input wire         sub,
7     output logic [3:0] sout
8     );
9
10    always_comb begin
11       if (sub)
12         sout <= ain - bin;
13       else
14         sout <= ain + bin;
15    end
16 endmodule
17 `default_nettype wire
```

■手続的代入（12 行目・14 行目）　alwyas_comb ブロック内部では，initial ブロックと同様にそれぞれの文が「順番に実行される」ので，「手続的代入」を用いる必要があります．代入の記号には「<=」を用います．左辺の変数は logic 型にしておきます．

3.3　それで，結局どっちの記述がいいの？

これまで，同じ加減算器の機能を持つ異なる 2 つの SystemVerilog 記述（ソースコード 4 とソースコード 5）を示しました．すると，どちらがより良い記述なのかが気になりますが，この程度の規模では合成される回路に違いはないと考えて差し支えありません．好きなスタイルで記述してください．ただし，ツールによっては得意な記述や苦手な記述など癖がある場合もありますので，使用するツールのドキュメントを読み，ツールベンダの推奨ガイドラインにしたがうことをお勧めします．