情報工学実験 II
第 2 回： 加減算器の実装

柴田裕一郎・松尾堅太郎・元島晃伸

shibata@cis.nagasaki-u.ac.jp
情報工学コース

2018 年 12 月 6 日

加減算器を作る

1. 加算器の記述
   1. バス信号の記述
   2. 乱数を用いたテストベクタの生成
   3. シミュレーションログの検証
2. 減算の手順と加減算器の記述
   1. 継続的代入による記述： 三項条件演算子
   2. 手続的代入による記述： if 文

4 ビット加算器の記述

```
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
```

- 「+」で加算を表現： どんな加算器なのかは記述していない
  ⇒ 抽象度の高い設計

バス信号線の宣言

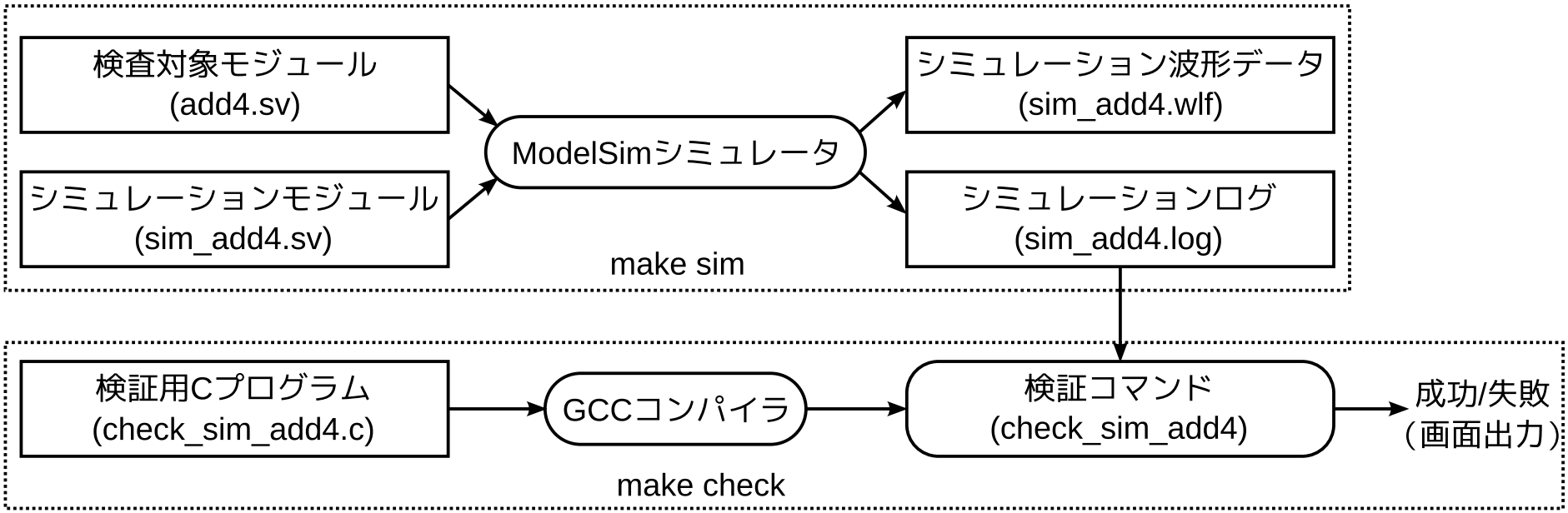
```
input wire [3:0]  ain, bin,
output wire [3:0] sout
```

- 信号名に [3:0] をつけて 4 ビットのバスであることを示す
- ビット番号は C 言語の配列と同様に 0 オリジン
- この例では，ain[0]（0 ビット目）から ain[3]（3 ビット目）までの合計 4 ビットの信号が使える
- ain[3:1] と記述すると 3 ビット目から 1 ビット目までの 3 ビットを表す

テストベクタの生成

- テストベクタの役割
  - 設計段階でのバグ（記述ミス，考え違い etc）の発見
  - LSI 製造後の不良品の発見
- 回路規模が大きくなると全パターンによる検証は無理
  - 回路の論理構造を考えたテストベクタの生成
  - BIST（Built-In Self Test）
  - 現在もホットな研究分野のひとつ
- 乱数に基づくテストベクタ生成
  - 全数検査が不可能な場合には有力な手段
  - 完璧ではない

設計検証の流れ


検査対象モジュール
(add4.sv)
シミュレーションモジュール
(sim_add4.sv)
ModelSimシミュレータ
シミュレーション波形データ
(sim_add4.wlf)
シミュレーションログ
(sim_add4.log)
make sim
検証用Cプログラム
(check_sim_add4.c)
GCCコンパイラ
検証コマンド
(check_sim_add4)
成功/失敗
(画面出力)
make check

乱数でテストベクタを生成

```
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
```

シミュレーションログ

(sim_add4.log にも保存される)

```
# 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 ....
```

目視チェックの欠点

- 根性が必要
- ビット幅が広くなってテストベクタ数が増えると不可能
- エラーを見逃しまう可能性が高く信頼できない

$\Rightarrow$ プログラムによる自動化

検証用 C プログラム（1/3）

```
6  int main(int argc, char *argv[])
7  {
8    FILE *fp;
9    char buf[BUF_SIZE];
10   int  lineNumber = 1;     /* 行数カウンタ */
11   int  checkedLines = 0;   /* 検査済みテストベクタ数 */
12
13   /* ログファイルが指定されていなければ終了 */
14   if (argc != 2) {
15     fprintf(stderr, "error: ログファイルが未指定\n");
16     exit(1);
17   }
18
19   /* ログファイルがオープンできなれば終了 */
20   if ((fp = fopen(argv[1], "r")) == NULL) {
21     fprintf(stderr, "error: ファイルオープン失敗: %s\n", argv[1]);
22     exit(1);
23   }
```

検証用 C プログラム（2/3）

```
25    /* ログファイルから 1行ずつ読み期待値と比較 */
26    while (fgets(buf, BUF_SIZE, fp) != NULL) {
27      /*「# ain=」で始まる行だけ処理 */
28      if (strncmp(buf, "# ain=", 6) == 0)  {
29        int ain, bin, sout, expectedValue;
30
31        /* 入力値と出力値の読み込み */
32        ain  = atoi(buf + 6);
33        bin  = atoi(buf + 14);
34        sout = atoi(buf + 23);
35
36        /* 期待値の計算 */
37        expectedValue  = (ain + bin) & 0xf;
```

検証用 C プログラム（3/3）

```
39        /* 出力と期待値を比較し異なれば終了 */
40        if (sout == expectedValue)
41          checkedLines++;
42        else {
43          fprintf(stderr, "error: %d 行目: ain=%d bin=%d sout=%d 期
      待値=%d\n",
44                  lineNumber, ain, bin, sout, expectedValue);
45          exit(1);
46        }
47      }
48      lineNumber++;
49    }
50
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    }
```

シミュレーションと検証の手順

- シミュレーションの実行

```
% make sim
```

- 検証プログラムのコンパイルと検証の実行

```
% make check
```

- これらを連続して行うこともできる

```
% make sim check
```

結果の確認

- 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
```

のように出力されれば OK.

- 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，

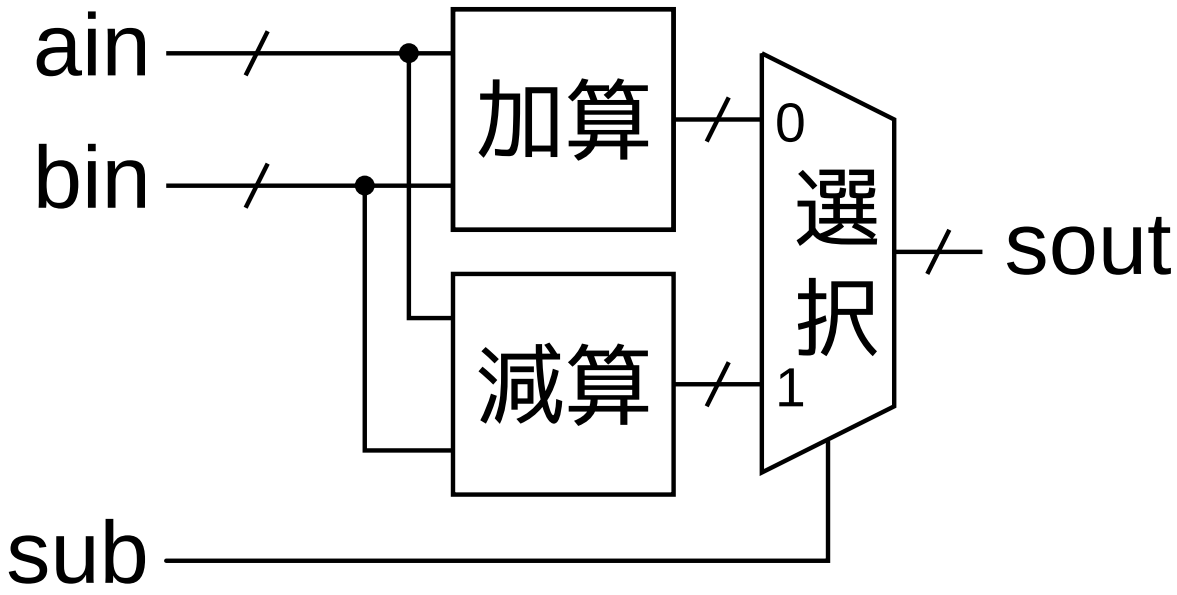
```
error: 27 行目: ain=3 bin=4 sout=12 期待値=7
make: *** [check] エラー 1
```

のように表示される.

2 の補数を用いた減算の方式（$A - B$）

1. B の中の ‘1’ と ‘0’ を反転させる
   「B の 1 の補数」ができる.
2. この数に 1 を足す
   「B の 2 の補数」ができる.
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）
   「B の 2 の補数」と A の加算は $A - B$ を意味する

加減算器の論理的構成



- 加算・減算を指定する 1 ビットの sub 入力を付加
- sub が ‘0’ なら加算，‘1’ なら減算
- 出力をマルチプレクサで選択
- 加算器と減算器の共有化などは合成ツールに任せる

継続的代入による 4 ビット加減算器の記述

```
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
```

- assign 文で三項条件演算子（?:）を使用
- 継続的代入によりマルチプレクサの動作を記述

if 文の利用

- if 文を使って書きたいとき…
  - if 文は継続的代入（assign 文）では使えない
- 手続的代入を使って組み合わせ回路を記述
  - always_comb 文を使う
  - 出力変数を logic 型にする
  - 代入記号「<=」を使う
- always_comb 文
  - 組み合わせ回路（combination circuit）を手続的に記述するために用意された構文
  - 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
  - 出力には必ず何らかの値が代入される必要

手続的代入による 4 ビット加減算器の記述

```
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
```

どちらの記述がよいのか？

- 事実上，合成される回路に違いはない
  - 好きなスタイルで記述してよい
- 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ことも
  - 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