MICS 実験第一 J2 課題レポート

学籍番号 2210342, 鈴木謙太郎

2024 年 7 月 27 日

1　問題 1

1.1　モジュール eq の動作検証

実験資料で示されたコード 1 のようなモジュール eq を, 同じく示された eqSim を用いてシミュレーションした. このモジュール eq は, 入力 a, b に対して $(a \wedge b) \vee (\bar{a} \wedge \bar{b})$ を出力するもので,$a = b$ のときだけ $s = 1$ となる.

Code 1: モジュール eq の Verilog コード

```
1  module eq (
2      s,a,b
3  );
4      input a, b;
5      output s;
6      wire na, nb, s1, s2;
7      assign na = ~a, nb = ~b;
8      assign s1 = a & b, s2 = na & nb;
9      assign s = s1 | s2;
10 endmodule
```

このモジュール eq をシミュレーションするモジュール eqSim をコード 2 のように作成した. このモジュールは, 上のモジュール eq に対して予想される 4 通りの入力をすべて与えて, それぞれの出力を確認するものである.

Code 2: モジュール eqSim の Verilog コード

```
1  module eqSim; /* 一致検出回路の */
2    wire s; /* シミュレーション */
3    reg x, y;
4    eq g1(s, x, y);
5    initial
6      begin
7      $dumpfile("eq.vcd");
8      $dumpvars(0, eqSim);
```

```
9       $monitor(" %b %b  %b  %b %b", x, y,  g1.s1, g1.s2,s, $stime);
10      $display(" x y s1 s2 s       time");
11      x=0; y=0;
12      #50 y=1;
13      #50 x=1; y=0;
14      #50 y=1;
15      #50 $finish;
16      end
17  endmodule
```

その結果は下のようになった.

```
x y s time
0 0 1          0
0 1 0         50
1 0 0        100
1 1 1        150
eqSim.v:15: $finish called at 200 (1s)
```

また, その波形を gtkwave で表示した結果は図 1 のようになった.

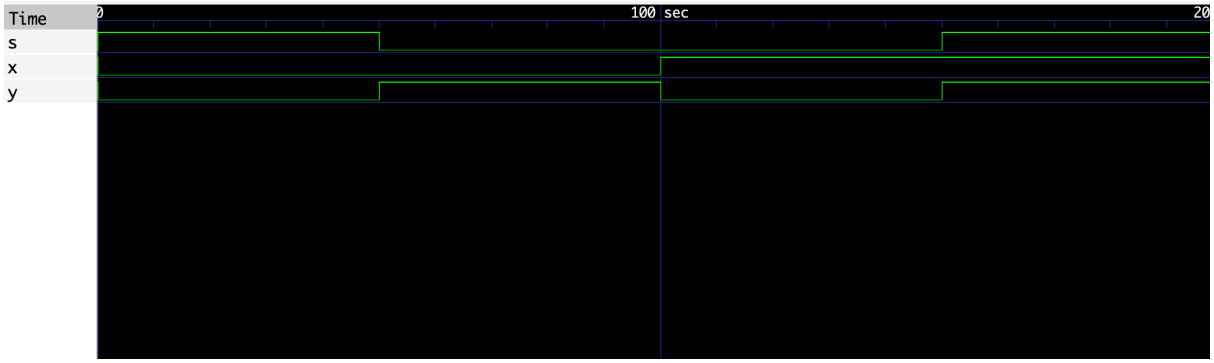

図 1: eqSim の波形

このシミュレーションはすべての入力例をカバーしており,$x = y$ のときのみ $s = 1$ となることが確認できている.

1.2 モジュール eqSim の改良

モジュール eqSim をコード 3 のように改良し,eq 内部の $s1, s2$ を表示するようにした.

Code 3: 改良後のモジュール eqSim の Verilog コード

```
1  module eqSim; /* 一致検出回路の */
2    wire s; /* シミュレーション */
3    reg x, y;
```

```
4   eq g1(s, x, y);
5   initial
6     begin
7     $dumpfile("eq.vcd");
8     $dumpvars(0, eqSim);
9     $monitor(" %b %b  %b  %b %b", x, y,  g1.s1, g1.s2,s, $stime);
10    $display(" x y s1 s2 s       time");
11    x=0; y=0;
12    #50 y=1;
13    #50 x=1; y=0;
14    #50 y=1;
15    #50 $finish;
16    end
17 endmodule
```

このときの出力は下のようになった.

```
x y s1 s2 s       time
0 0  0  1 1         0
0 1  0  0 0        50
1 0  0  0 0       100
1 1  1  0 1       150
eqSim.v:15: $finish called at 200 (1s)
```

これにより, 内部的にも s1 と s2 が正しく計算され,$s = s1 \vee s2$ により正しい s が出力されていることが確認できた.

2　問題 2

入力信号 a, b, c, d を受け取り，$(a = b) \wedge (c = d)$ のとき出力信号 s を 1 に，それ以外のとき s を 0 にするモジュール doubleEq を作成する.

このモジュールの簡易的な回路図は図 2 のようになっている.eq と表記した部分は 1 節で扱ったモジュール eq を用いる.

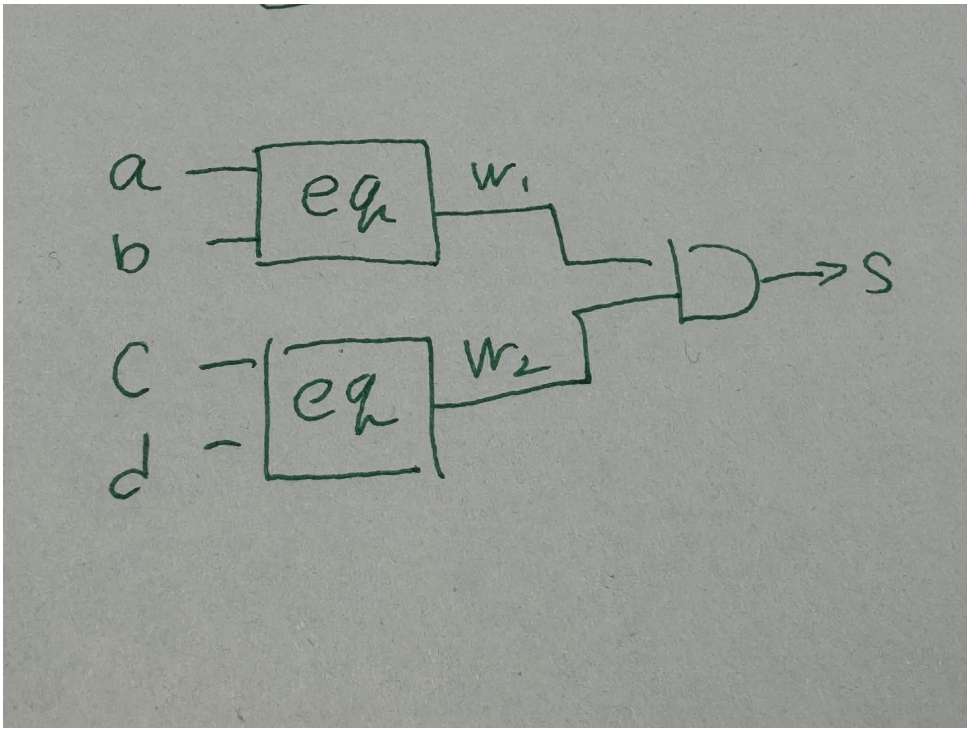

図 2: モジュール doubleEq の回路図

これをもとに, コード 4 のようにモジュール doubleEq を作成した.

Code 4: モジュール doubleEq の Verilog コード

```
1  module doubleEq(
2      s,a,b,c,d
3  );
4    input a, b, c, d;
5    output s;
6    wire w1, w2;
7    eq m1(w1, a, b);
8    eq m2(w2, c, d);
9    assign s = w1 & w2;
10 endmodule
```

3 問題 3

2 節で作成したモジュール doubleEq をシミュレーションするモジュール doubleEqSim をコード 5 のように作成した. 今回は入力ケースがたかだか 16 通りほどだったので, 想定されるすべての入力についてテストを行った.

Code 5: モジュール doubleEqSim の Verilog コード

```
1  module doubleEqSim; /* 一致検出回路の */
2    wire s; /* シミュレーション */
3    reg a, b, c, d;
4    doubleEq g1(s, a, b, c, d);
5    initial
6      begin
7      $dumpfile("doubleEq.vcd");
8      $dumpvars(0, doubleEqSim);
9      $monitor(" %b %b %b %b  %b  %b  %b", a, b, c, d, s, g1.w1, g1.w2, $stime);
10     $display(" a b c d w1 w2  s      time");
11     /* test all case */
12     a=0; b=0; c=0; d=0;
13     #50 a=0; b=0; c=0; d=1;
14     #50 a=0; b=0; c=1; d=0;
15     #50 a=0; b=0; c=1; d=1;
16     #50 a=0; b=1; c=0; d=0;
17     #50 a=0; b=1; c=0; d=1;
18     #50 a=0; b=1; c=1; d=0;
19     #50 a=0; b=1; c=1; d=1;
20     #50 a=1; b=0; c=0; d=0;
21     #50 a=1; b=0; c=0; d=1;
22     #50 a=1; b=0; c=1; d=0;
23     #50 a=1; b=0; c=1; d=1;
24     #50 a=1; b=1; c=0; d=0;
25     #50 a=1; b=1; c=0; d=1;
26     #50 a=1; b=1; c=1; d=0;
27     #50 a=1; b=1; c=1; d=1;
28     end
29 endmodule
```

このテストを実行し,gtkwave で波形を表示した結果は図 3 のようになった.

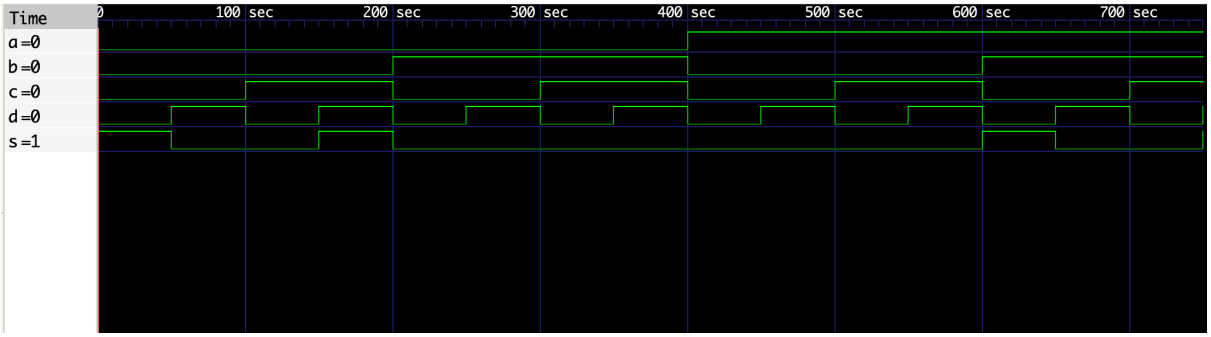

図 3: doubleEqSim の波形

この波形から,$a = b$ かつ $c = d$ のときだけ $s = 1$ となることが確認できた.

4 問題 4

実験資料で示された状態遷移表 m2 に従って, 図 4 のような状態遷移図と真理値表を作成した.

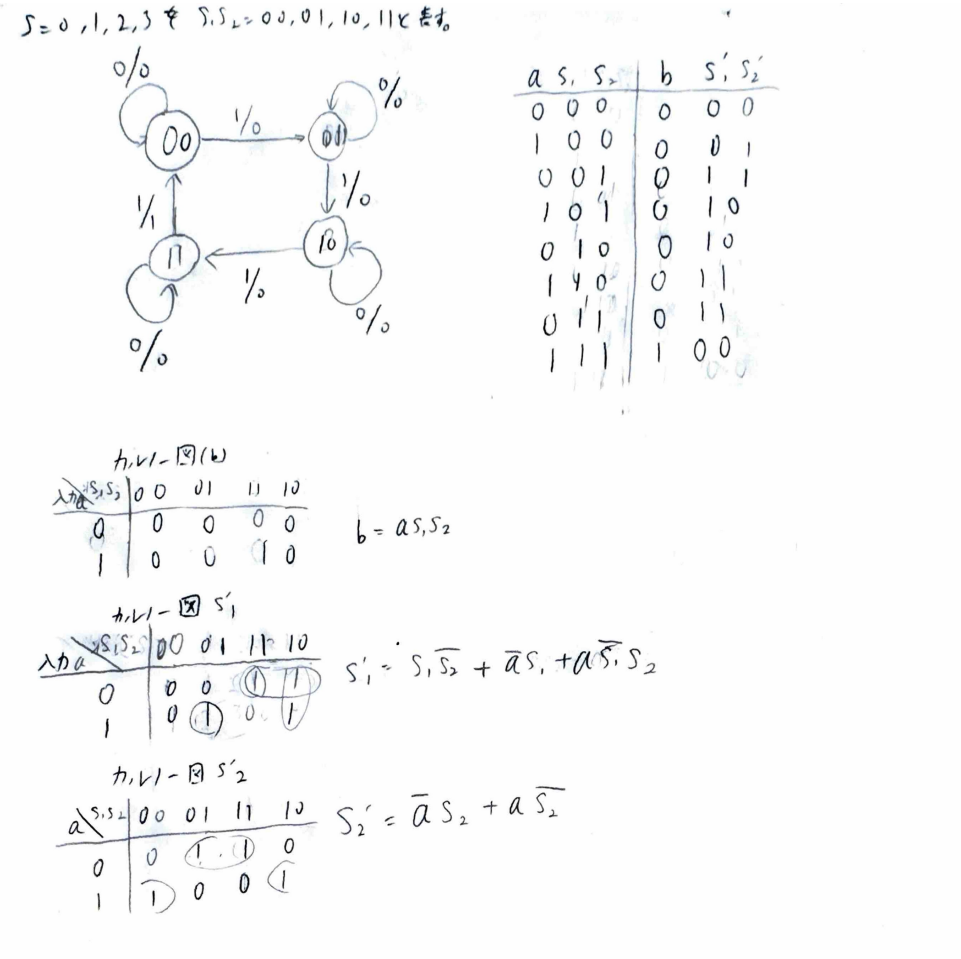

図 4: 作成した状態遷移図および真理値表

この結果,$b, s1', s2'$ の論理式は式 (1) のようになった.

$$\begin{aligned} b &= as_1s_2 \\ s1' &= s_1\bar{s_2} + \bar{a}s_1 + a\bar{s_1}s_2 \\ s2' &= \bar{a}s_2 + a\bar{s_2} \end{aligned} \tag{1}$$

この論理式を下に, コード 6 のようなモジュール count を作成した.

Code 6: モジュール count の Verilog コード

```
1 module count (
```

```
    a,ck,b
);
  input a, ck;
  output b;
  wire na;
  wire s1, s2, t;
  wire d1, d2, d3, d4;
  wire e1, e2, e3;
  dffn f1(s1,d1,ck);
  dffn f2(s2,e1,ck);
  assign na = ~a;
  assign ns1 = ~s1;
  assign ns2 = ~s2;

  assign d4 = s1 & ns2, d3 = na & s1, d2 = a & ns1 & s2, d1 = d4 | d3 | d2;
  assign e3 = na & s2, e2 = a & ns2, e1 = e3 | e2;
  assign b = a & s1 & s2;
endmodule
```

また, このモジュール count をシミュレーションするモジュール countSim をコード 7 のように作成した. clk が発するクロック信号は 50 サイクル周期で切り替わる. ここで a の値を切り替える間隔を 100 サイクルにすることで, a を 1 回切り替える間に clk が 0,1 両方の場合をテストできた. これにより, 少ない行数でより多くのテストケースをカバーできた.

Code 7: モジュール countSim の Verilog コード

```
module countSim;
  reg a;
  wire b;

  clk clk1(ck);
  count dut (a,ck,b);

  initial
    begin
    $dumpfile("countSim.vcd");
    $dumpvars(0, countSim);
    $monitor("%b  %b %b  %b  %b", a, ck, b, dut.s1, dut.s2, $stime);
    $display("a ck b s1 s2       time");

    a = 0;
    #100 a = 1;
    #100 a = 0;
```

```
18      #100 a = 1;
19      #100 a = 0;
20      #100 a = 1;
21      #100 a = 0;
22      #100 a = 1;
23      #100 a = 0;
24      $finish;
25      end
26  endmodule
```

このテストの出力から, 状態 $s1$ と $s2$ が正しく遷移していることが確認できた. また,gtkwave を用いて波形を表示すると図 5 のようになった.

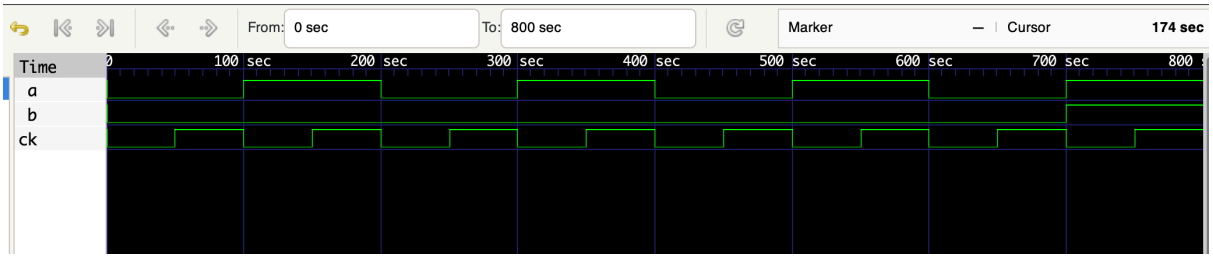

図 5: countSim の波形

この波形から, 正しい状態遷移の結果 $s1s2$ が 11 から 00 に遷移するときに $b = 1$ となっていることが確認できた.

5 問題 5

実験資料で示された 1 ビット加算器を参考にして,N ビット加算器モジュール addN をコード 8 のように作成した.

Code 8: モジュール addN の Verilog コード

```
1  module addN (
2      a, b, sum, ci, cu
3  );
4    parameter N = 8;
5
6    input [N-1:0] a, b;
7    input ci;
8    output [N-1:0] sum;
9    output cu;
10
11   wire [N-1:0] sum;
12   wire [N:0] c; // carries
```

```
13    assign c[0] = ci;
14
15
16    assign sum = a ^ b ^ c[N-1:0];
17    assign c[N:1] = (a & b) | (b & c[N-1:0]) | (a & c[N-1:0]);
18
19
20    assign cu = c[N];
21  endmodule
```

また, モジュール addN をシミュレーションするモジュール addNSim をコード 9 のように作成した.

Code 9: モジュール addNSim の Verilog コード

```
1  module addNSim;
2    reg [7:0] a, b;
3    reg ci;
4    wire [7:0] sum;
5    wire cu;
6    addN #8 g1(a, b, sum, ci, cu);
7
8    initial begin
9      $dumpfile("addN.vcd");
10     $dumpvars(0, addNSim);
11     $monitor(" %b  %b    %b  %b   %b", a, b,ci, sum, cu, $stime);
12     $display("        a          b   ci       sum  cu      time");
13
14     // test normal
15     a = 8'b00000011;
16     b = 8'b00000011;
17     ci = 0;
18
19     // test carry in
20     #10;
21     a = 8'b00000011;
22     b = 8'b00000011;
23     ci = 1;
24
25     #10;
26     // test overflow
27     a = 8'b11111111;
28     b = 8'b00000001;
29     ci = 0;
```

```
30
31      #10 $finish;
32    end
33  endmodule
```

このテストの出力は下のようになった.

a	b	ci	sum	cu	time
00000011	00000011	0	00000110	0	0
00000011	00000011	1	00000111	0	10
11111111	00000001	0	00000000	1	20

また,gtkwave を用いて波形を表示すると図 6 のようになった.

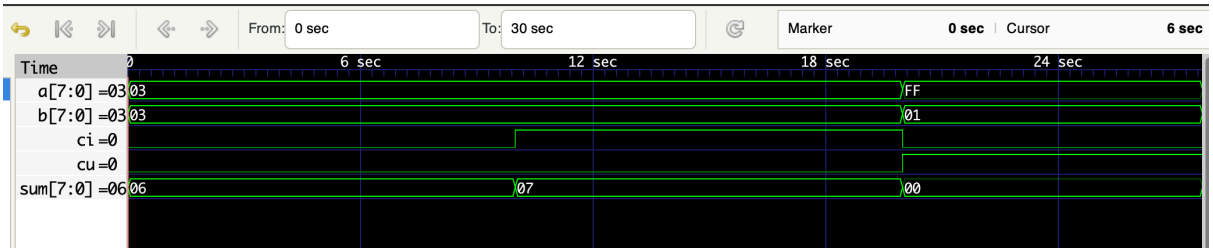

図 6: addNSim の波形

これらの結果から, 通常の演算と桁あふれが発生する演算の両方でモジュール addN が正しく動作していることが確認できた. ビットベクトルを用いることで,1 ビット加算器の実装を大きく変えることなく N ビット加算器を実装できることがわかった.

6 問題 6

実験資料で示された 1 ビットレジスタを参考にして,N ビットレジスタ regN を作成した.

6.1 モジュール dffn の改良

まず, モジュール dffn を N ビットの入出力に対応させる必要があると考え, コード 10 のように改良した. なお, 入出力ビット数 N のデフォルト値を 1 にすることで既にモジュール dffn を使用している他のモジュールに影響を与えないようにした.

Code 10: 改良後のモジュール dffn の Verilog コード

```
1  module dffn (
2      Q,D,ck
3  );
4    parameter N = 1;
5
6    input [N-1:0] D;
```

```
7    input ck;
8    output [N-1:0] Q;
9    reg [N-1:0] Q;
10   initial
11       Q = {N{1'b0}};
12   always @(negedge ck)
13       Q = D;
14 endmodule
```

あわせて, モジュール dffn をシミュレーションするモジュール dffnSim もコード 11 のように改良した.

Code 11: 改良後のモジュール dffnSim の Verilog コード

```
1  module dffnSim;
2    reg[1:0] i;
3    wire[1:0] o;
4    clk clk1(ck);
5    dffn #2 dffn1(o, i, ck);
6    initial
7      begin
8      $dumpfile("dffnSim.vcd");
9      $dumpvars(0, dffnSim);
10     $monitor(" %b %b %b", ck,i,o,$stime);
11     $display("ck  i  o      time");
12     i = 2'b00;
13     #100 i = 2'b01;
14     #200 i = 2'b10;
15     #100 $finish;
16     end
17 endmodule
```

このテストの出力は下のようになった.

```
 ck  i  o      time
 0 00 00         0
 1 00 00        50
 0 01 01       100
 1 01 01       150
 0 01 01       200
 1 01 01       250
 0 10 10       300
 1 10 10       350
dffnSim.v:15: $finish called at 400 (1s)
 0 10 10       400
```

gtkwave を用いて波形を表示すると図 7 のようになった.

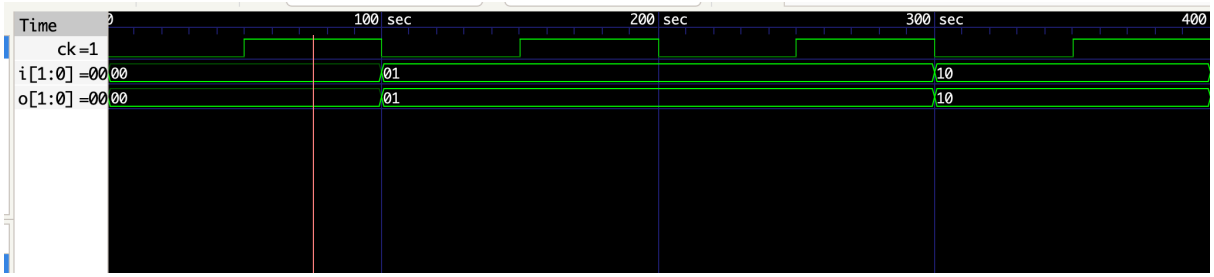

図 7: dffnSim の波形

このテストでは 2 ビットの信号の入出力をテストしている. クロックの立ち下がりに同期して 2 ビットの値を更新・保持できているので, 仕様を崩すことなくモジュール dffn を N ビット入出力に対応させることができたと判断した.

6.2 モジュール regN の作成

次に,N ビットレジスタのモジュール regN をコード 12 のように作成した.

Code 12: モジュール regN の Verilog コード

```
1  module regN (
2      l, d, ck, q
3  );
4    parameter N = 8;
5
6    input l;
7    input [N-1:0] d;
8    input ck;
9    output [N-1:0] q;
10   wire [N-1:0] d1;
11
12   dffn #8 f1(q,d1,ck);
13   assign d1 = l ? d : q;
14 endmodule
```

実験資料で示された conditional assign を用いることで, 少ない記述量でも load 信号の値に応じて $d1$ に送られる信号を制御することができた.

また, モジュール regN をシミュレーションするモジュール regNSim をコード 13 のように作成した.

Code 13: モジュール regNSim の Verilog コード

```
1    module regNSim ();
2    reg l;
3    reg [7:0] d;
```

```
4   clk c1(ck);
5   wire [7:0] q;
6
7   regN #8 g1(l, d, ck, q);
8
9   initial begin
10  $dumpfile("regN.vcd");
11  $dumpvars(0, regNSim);
12  $monitor("   %b    %b  %b %b", l, d, ck, q, $stime);
13  $display("load        data ck        q      time");
14
15  l = 0;
16  d = 8'b00000000;
17  #50;
18
19  // pass first data
20  l = 1;
21  d = 8'b00000011;
22  #50;
23
24  // get first data
25  l = 0;
26  #50;
27
28  // pass second data
29  l = 1;
30  d = 8'b00001111;
31  #50;
32
33  // get second data
34  l = 0;
35  #50;
36
37  #50 $finish;
38  end
39 endmodule
```

このテストの出力は下のようになった.

```
load        data ck        q      time
   0    00000000  0 00000000         0
   1    00000011  1 00000000        50
```

```
   0     00000011   0 00000011        100
   1     00001111   1 00000011        150
   0     00001111   0 00001111        200
   0     00001111   1 00001111        250
   0     00001111   0 00001111        300
regNSim.v:39: $finish called at 300 (1s)
```

gtkwave を用いて波形を表示すると図 8 のようになった.

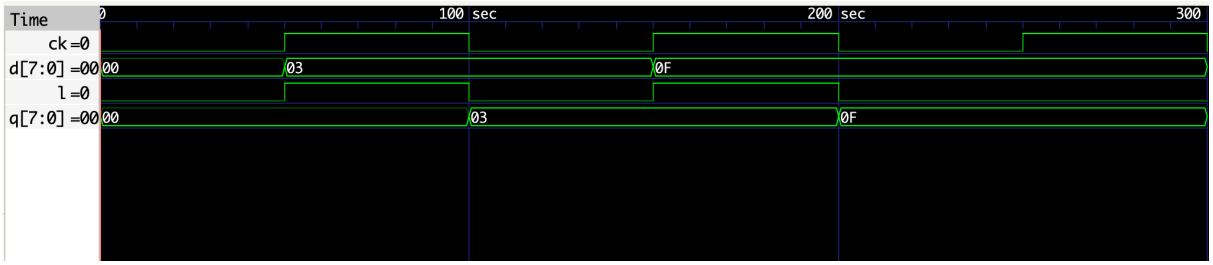

図 8: regNSim の波形

クロックが立ち下がる際に load に 1 が入力されているときだけ, 出力 q が d の値に更新されることが視覚的に確認できた.

7 問題 7

実験資料で示された N ビット加減算器の図解を参考にして,N ビット加減算器 calcN をコード 14 のように作成した.

Code 14: モジュール calcN の Verilog コード

```
1  module calcN (
2    a, b, sum, k, cu
3  );
4    parameter N = 8;
5
6    input [N-1:0] a, b;
7    input k;
8    output [N-1:0] sum;
9    output cu;
10
11   wire[N-1:0] B = k ? ~b : b;
12   addN #(
13        .N(N)
14   ) g1(a, B, sum, k, cu);
15
```

```
16 endmodule
```

また, モジュール calcN をシミュレーションするモジュール calcNSim をコード 15 のように作成した.

Code 15: モジュール calcNSim の Verilog コード

```
1  module calcNSim;
2  reg [7:0] a, b;
3  reg k;
4  wire [7:0] sum;
5  wire cu;
6  calcN #8 g1(a, b, sum, k, cu);
7
8  initial begin
9    $dumpfile("calcN.vcd");
10   $dumpvars(0, calcNSim);
11   $monitor(" %b  %b  %b  %b  %b", a, b, k, sum, cu, $stime);
12   $display("        a          b  k       sum  cu      time");
13
14   // test add
15   a = 8'b00000011;
16   b = 8'b00000011;
17   k = 0;
18
19   // test sub
20   #10;
21   a = 8'b00000011;
22   b = 8'b00000011;
23   k = 1;
24   #10;
25
26   // test add overflow
27   a = 8'b11111111;
28   b = 8'b00000001;
29   k = 0;
30   #10;
31
32   // test sub overflow
33   a = 8'b00000000;
34   b = 8'b00000001;
35   k = 1;
36   #10 $finish;
37 end
```

```
38 endmodule
```

このテストの出力は下のようになった.

```
       a          b  k        sum  cu      time
 00000011   00000011  0   00000110  0         0
 00000011   00000011  1   00000001  1        10
 11111111   00000001  0   00000000  1        20
 00000000   00000001  1   11111111  0        30
calcNSim.v:41: $finish called at 40 (1s)
```

$k = 0$ のときに加算,$k = 1$ のときに減算を行うことが確認できた. また, 加算時は $cu = 1$ ならオーバーフロー, 減算時は $cu = 0$ ならアンダーフローとなることも確認できた.

gtkwave を用いて波形を表示すると図 9 のようになった. a と b に同じ入力をしていても,k を切り替えることで加算減算が切り替わり異なる値を正しく出力していることが確認できた.

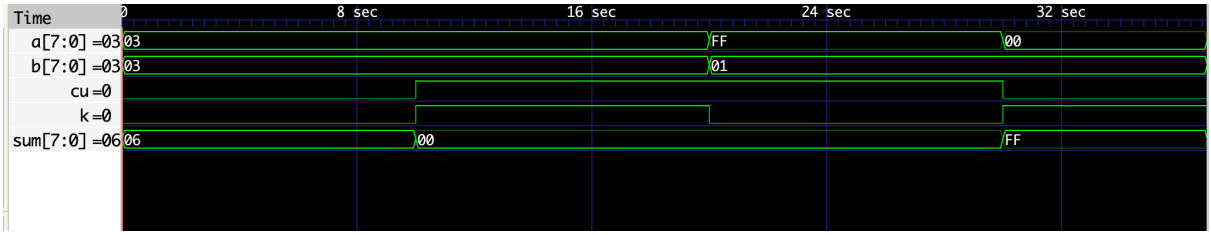

図 9: calcNSim の波形