論理回路入門（６）

組合せ回路１：マルチビット加算回路

李　亜民

2024 年 10 月 29 日 (火)

マルチビット加算回路

ポイント

- マルチビット
- マルチビットの加算
- リップルキャリー加算器 (RCA)

  RCA: Ripple-Carry Adder
- 桁上げ先見加算器 (CLA)

  CLA: Carry-Look-ahead Adder
- ツリー型桁上げ先見加算器の回路

復習：2進数の加算

```
  0     0     1     1
+ 0   + 1   + 0   + 1
---   ---   ---   ---
  0     1     1   1 0
```

繰り上がり (carry out)

変数で表す:

```
    1  ←― a
+   1  ←― b
-----
  1 0
```

入力信号

出力信号

c: carry out (上位桁への繰り上がり)

s: sum (和)

```
  ↑   ↑
  c   s
```

復習：半加算器設計

真理値表

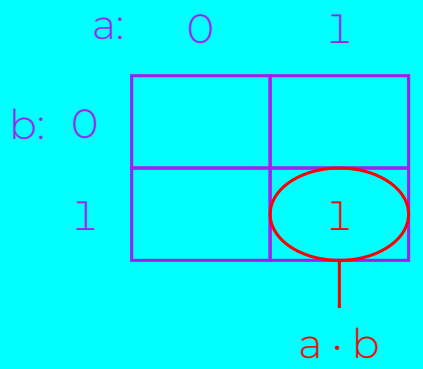
a	b	c	s	コメント
0	0	0	0	$0+0=0\,0$ (加算)
0	1	0	1	$0+1=0\,1$ (加算)
1	0	0	1	$1+0=0\,1$ (加算)
1	1	1	0	$1+1=1\,0$ (加算)

論理式

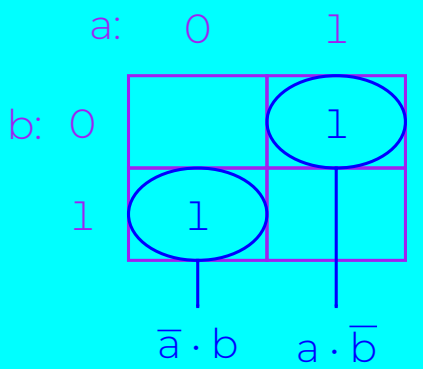
$c = a \cdot b$

$s = \overline{a} \cdot b + a \cdot \overline{b} = a \oplus b$

c のカルノー図



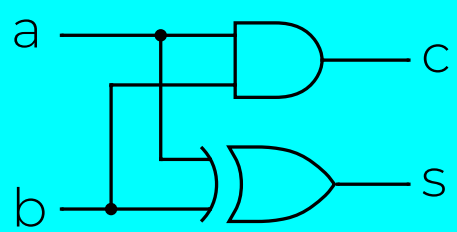
s のカルノー図



復習：半加算器設計 (回路)

論理式 $c = a \cdot b$

$$s = \overline{a} \cdot b + a \cdot \overline{b} = a \oplus b$$



Verilog HDL による回路（half_adder.v）

```
‘timescale 1ns/1ns

module half_adder (a, b, c, s);
    input  a, b;
    output c, s;

    assign c = a & b; // AND
    assign s = a ^ b; // XOR

endmodule
```

half_adder.v

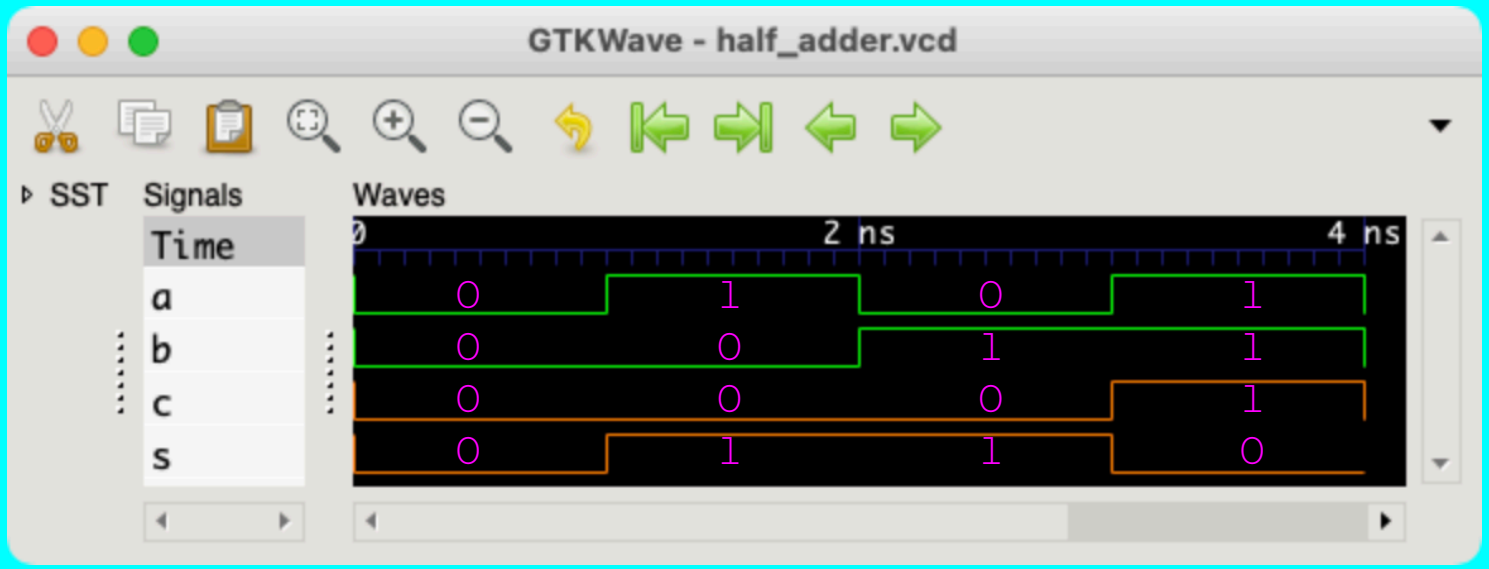
復習：半加算器設計 (テストベンチ)

```
‘timescale 1ns/1ns
module half_adder_tb;
    reg  a, b;
    wire c, s;
    half_adder ha (a, b, c, s);
    initial begin
        #0 a = 0;
        #0 b = 0;
        #4 $finish;
    end
    always #1 a = ~a;
    always #2 b = ~b;
    initial begin
        $dumpfile ("half_adder.vcd");
        $dumpvars;
    end
endmodule
```

half_adder_tb.v

復習：半加算器設計 (波形)

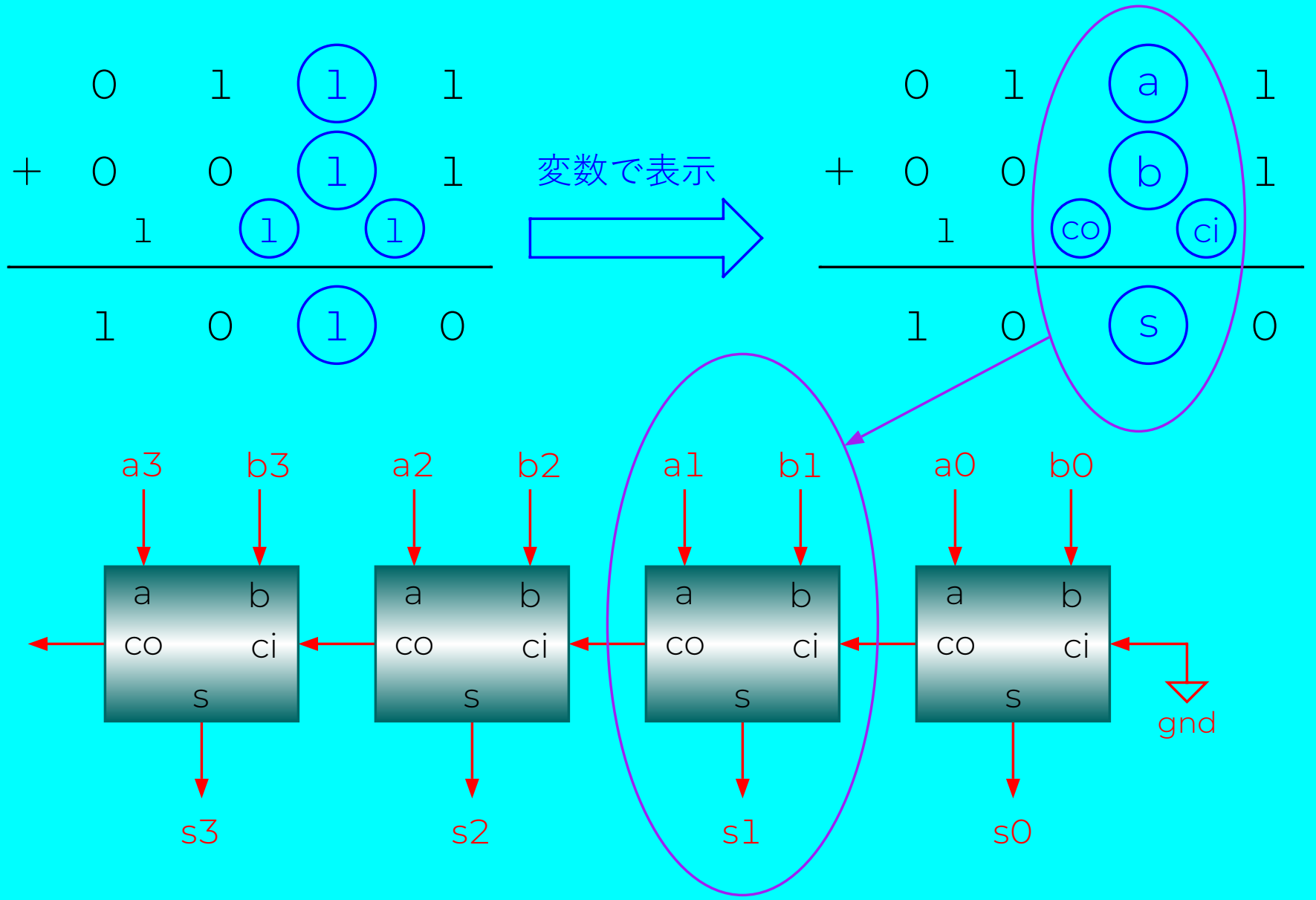
```
% iverilog -Wall -o half_adder \
  half_adder_tb.v half_adder.v
% vvp half_adder
% gtkwave half_adder.vcd
```



a b cs　a b cs　a b cs　a b cs

0+0=00　1+0=01　0+1=01　1+1=10

復習：全加算器設計


0 1 1 1
+ 0 0 1 1
1 1 1
1 0 1 0
変数で表示
0 1 a 1
+ 0 0 b 1
1 co ci
1 0 s 0
a3 b3 a2 b2 a1 b1 a0 b0
a b
co ci
s
gnd
s3 s2 s1 s0

復習：全加算器の真理値表

input a, b, ci; // ci: carry in (下位桁からの繰り上がり)
output co, s; // co: carry out (上位桁への繰り上がり), s: sum (和)

a	b	ci	co	s	コメント
0	0	0	0	0	0 + 0 + 0 = 0 0 (加算)
0	0	1	0	1	0 + 0 + 1 = 0 1 (加算)
0	1	0	0	1	0 + 1 + 0 = 0 1 (加算)
0	1	1	1	0	0 + 1 + 1 = 1 0 (加算)
1	0	0	0	1	1 + 0 + 0 = 0 1 (加算)
1	0	1	1	0	1 + 0 + 1 = 1 0 (加算)
1	1	0	1	0	1 + 1 + 0 = 1 0 (加算)
1	1	1	1	1	1 + 1 + 1 = 1 1 (加算)

$s = \overline{a}\,\overline{b}\,ci + \overline{a}\,b\,\overline{ci} + a\,\overline{b}\,\overline{ci} + a\,b\,ci$ 最小項の論理和
$co = \overline{a}\,b\,ci + a\,\overline{b}\,ci + a\,b\,\overline{ci} + a\,b\,ci$ 最小項の論理和

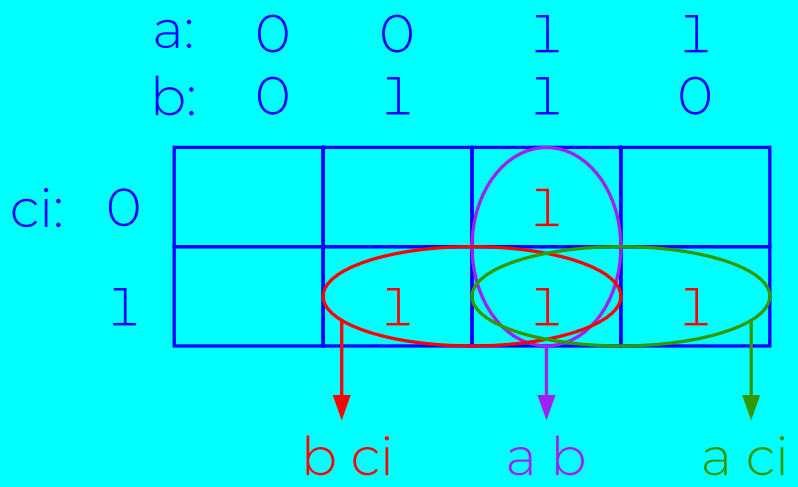
復習：全加算器設計

真理値表から論理式をつくる

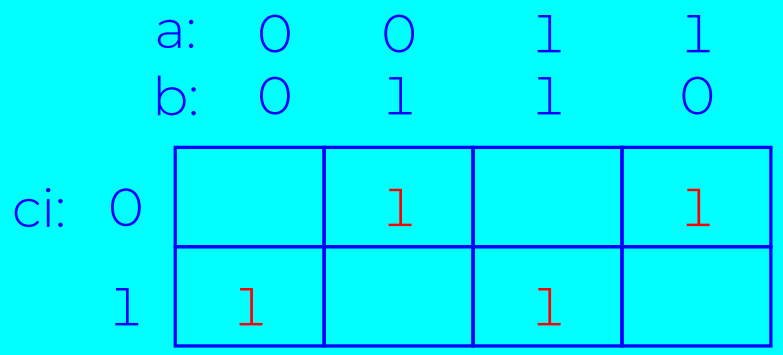
— どの組み合わせで出力が 1 になるか

カルノー図による論理式を簡単化する (co)

co のカルノー図



s のカルノー図



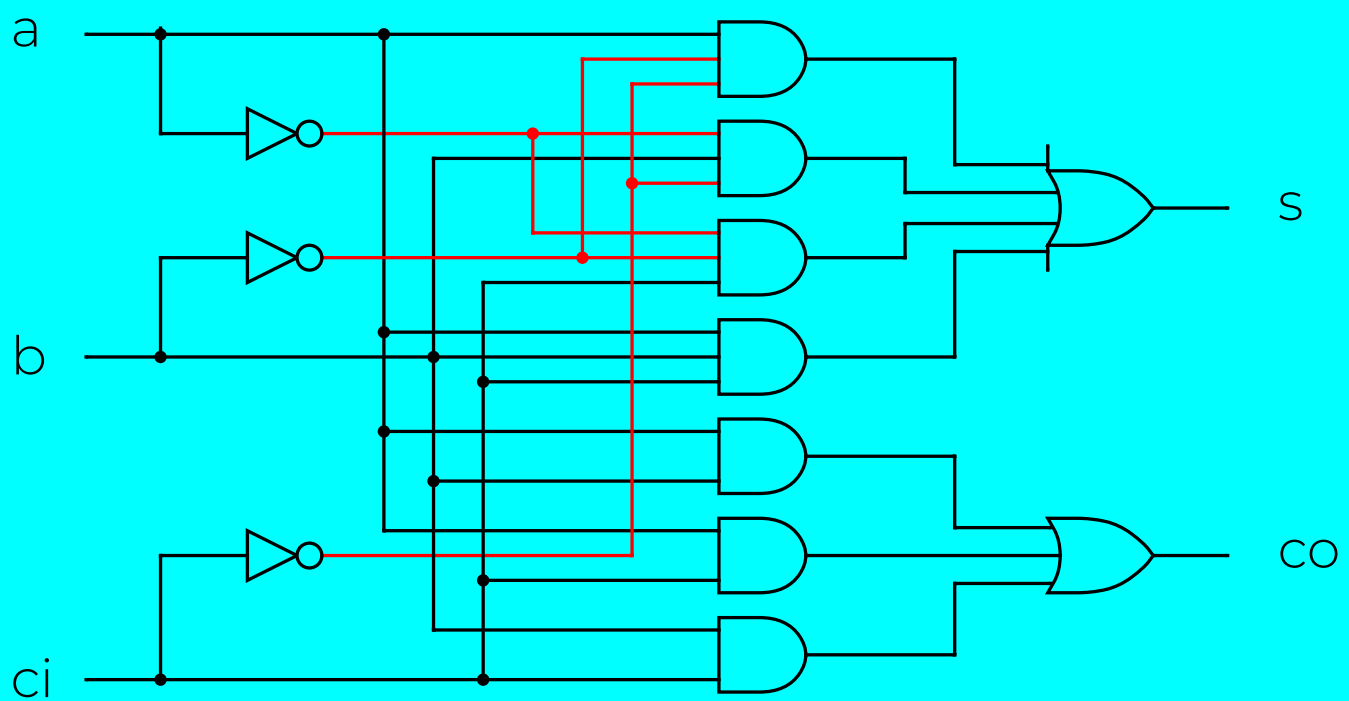
$co = a\,b + a\,ci + b\,ci$ (二つの入力が 1 1 であれば)

$s = a\,\overline{b}\,\overline{ci} + \overline{a}\,b\,\overline{ci} + \overline{a}\,\overline{b}\,ci + a\,b\,ci$

復習：全加算器設計 (回路)

$$co = a\,b + a\,ci + b\,ci$$ (二つの入力が 1 1 であれば)

$$s = a\,\overline{b}\,\overline{ci} + \overline{a}\,b\,\overline{ci} + \overline{a}\,\overline{b}\,ci + a\,b\,ci$$



復習：全加算器設計 (回路)

$co = a\,b + a\,ci + b\,ci$ (二つの入力が 1 であれば)

$s = a\,\overline{b}\,\overline{ci} + \overline{a}\,b\,\overline{ci} + \overline{a}\,\overline{b}\,ci + a\,b\,ci$

```
‘timescale 1ns/1ns

module fa (a, b, ci, co, s);
    input  a, b, ci;
    output co, s;

    assign co = a & b | a & ci | b & ci;

    assign s =  a & ~b & ~ci |
               ~a &  b & ~ci |
               ~a & ~b &  ci |
                a &  b &  ci;

endmodule
```

fa.v

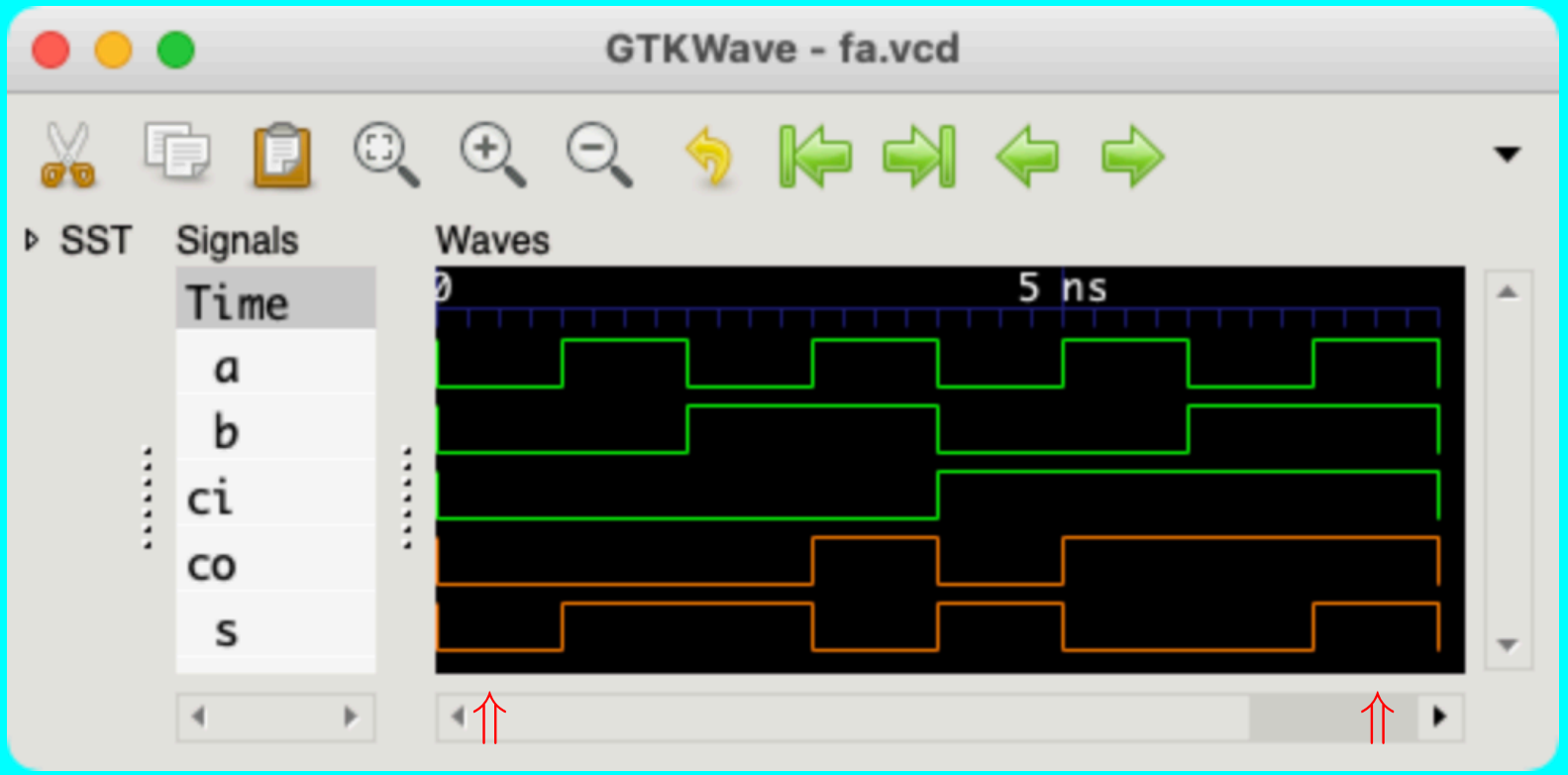
復習：全加算器設計 (テストベンチ)

```
‘timescale 1ns/1ns
module fa_tb;
    reg  a, b, ci;
    wire co, s;
    fa fadder (a, b, ci, co, s);
    initial begin
        #0 a  = 0; b  = 0; ci = 0;
        #8 $finish;
    end
    always #1 a  = ~a;
    always #2 b  = ~b;
    always #4 ci = ~ci;
    initial begin
        $dumpfile ("fa.vcd");
        $dumpvars;
    end
endmodule
```

fa_tb.v

復習：全加算器設計 (波形)

```
% iverilog -Wall -o fa fa_tb.v fa.v
% vvp fa
% gtkwave fa.vcd
```



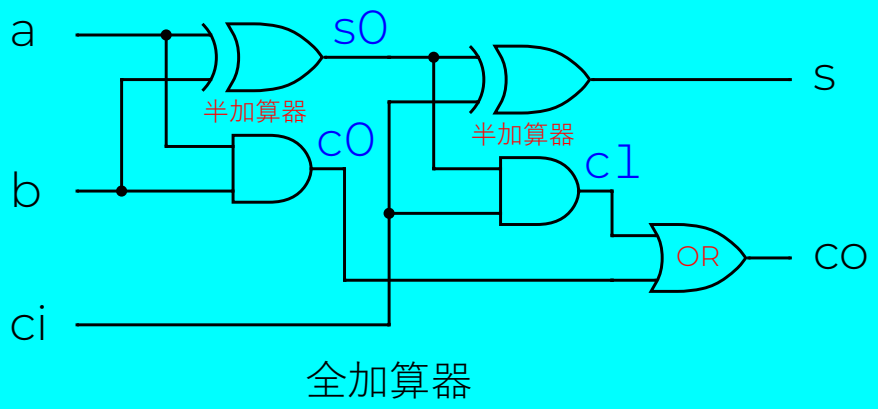
$0+0+0=00_2$ $1+1+1=11_2$

復習：半加算器を用いた全加算器

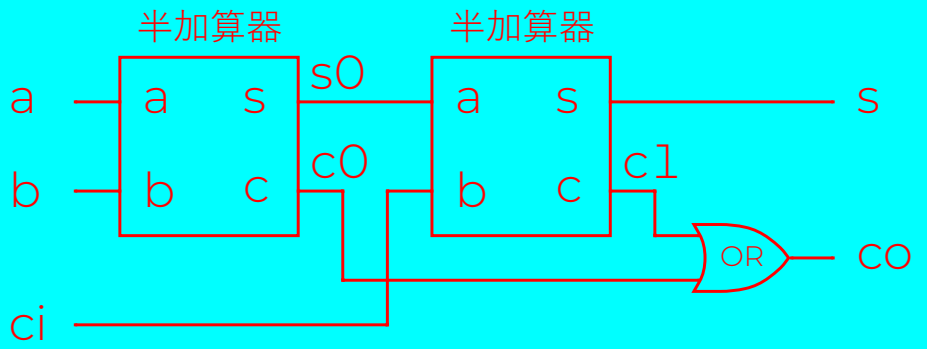
全加算器は、２つの半加算器と１つのOR回路を用いて構成することができる。

$$
\begin{aligned}
s &= a\,\overline{b}\,\overline{ci} + \overline{a}\,b\,\overline{ci} + \overline{a}\,\overline{b}\,ci + a\,b\,ci \\
&= (a \oplus b) \oplus ci = s0 \oplus ci \\
co &= \overline{a}\,b\,ci + a\,\overline{b}\,ci + a\,b\,\overline{ci} + a\,b\,ci \\
&= (\overline{a}\,b + a\,\overline{b})\,ci + a\,b\,(\overline{ci} + ci) \\
&= (a \oplus b)\,ci + a\,b = s0\,ci + a\,b = c1 + c0
\end{aligned}
$$

$$
\begin{aligned}
s0 &= a \oplus b \\
c0 &= a\ \ b \\
c1 &= s0\ \ ci
\end{aligned}
$$



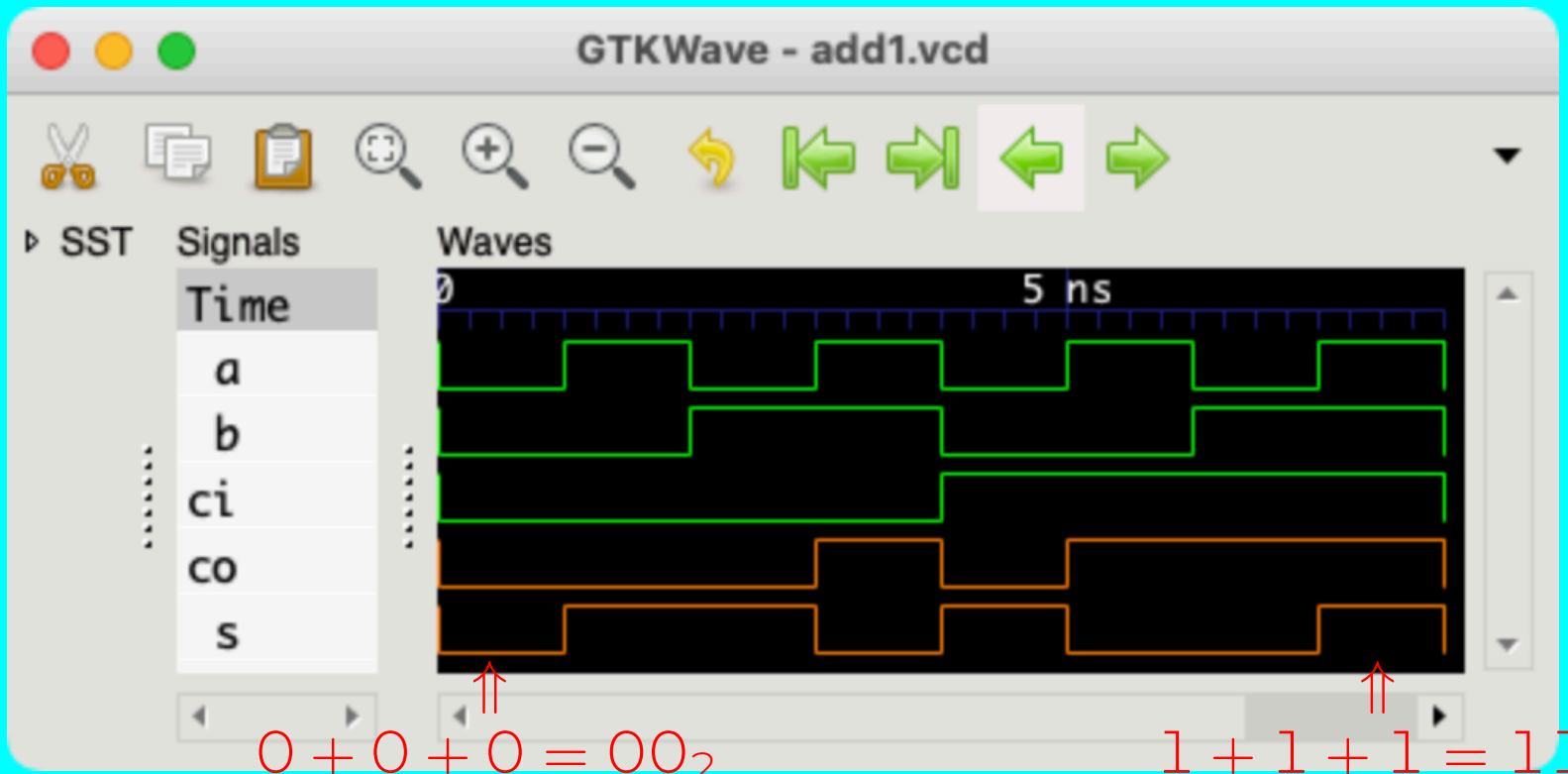
全加算器



全加算器

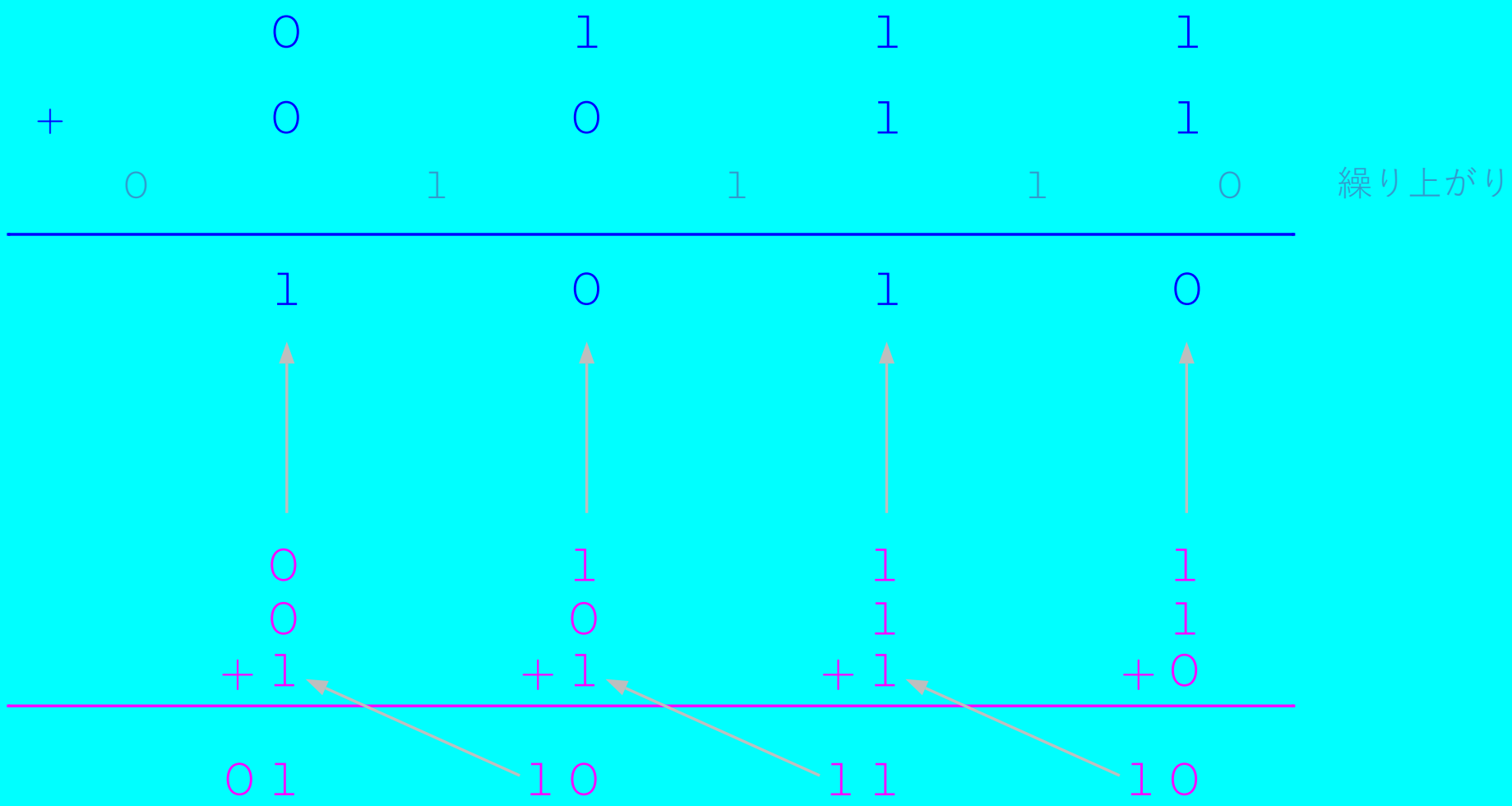
復習：半加算器を用いた全加算器

```
% iverilog -Wall -o add1 \
  add1_tb.v add1.v half_adder.v
% vvp add1
% gtkwave add1.vcd
```



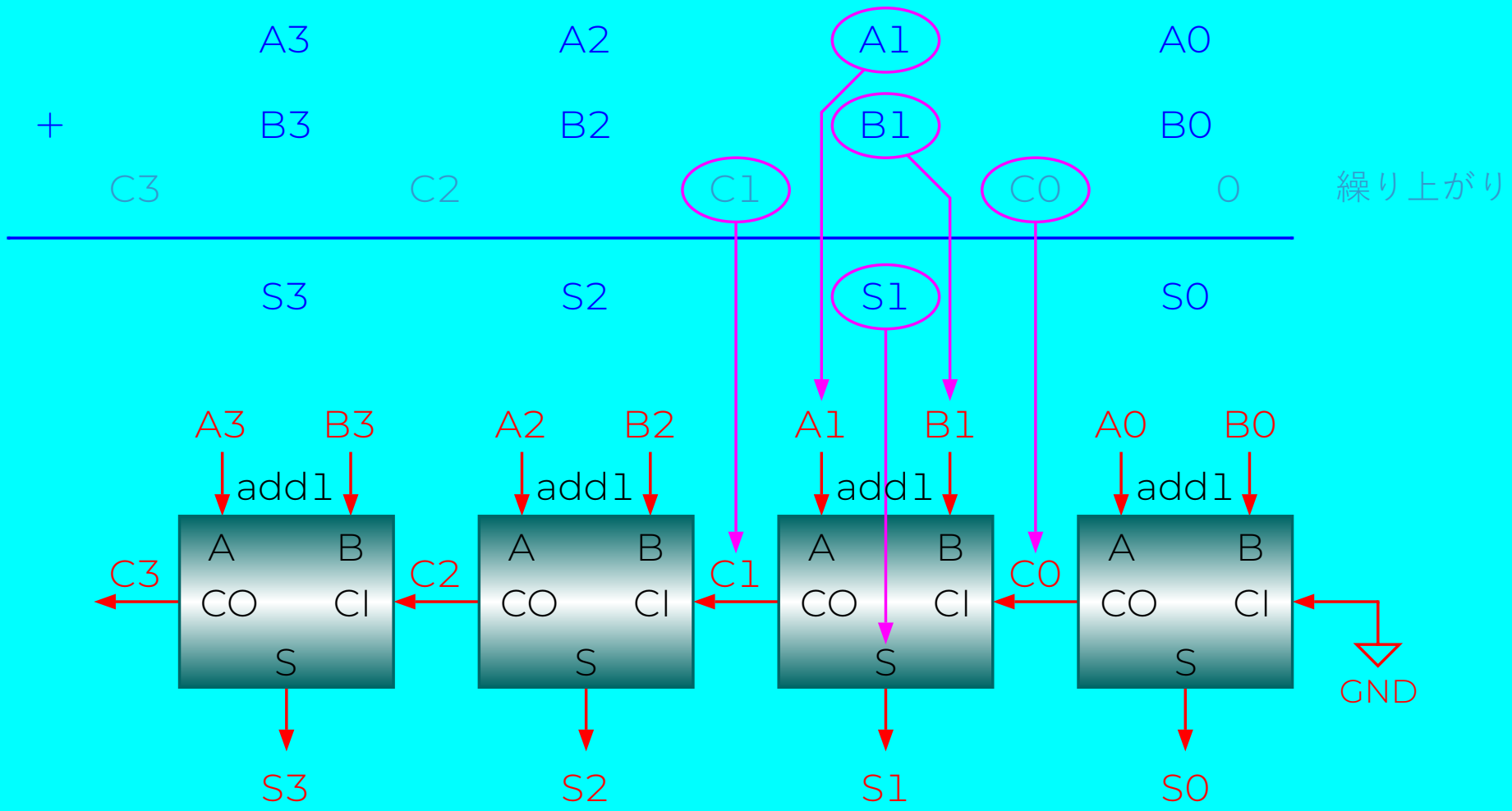
マルチビット (4 ビットの加算)

4 ビットの加算



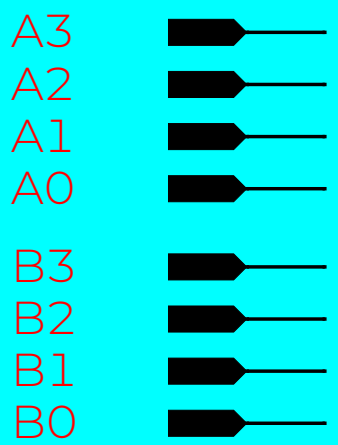
マルチビット (信号の名前と回路)

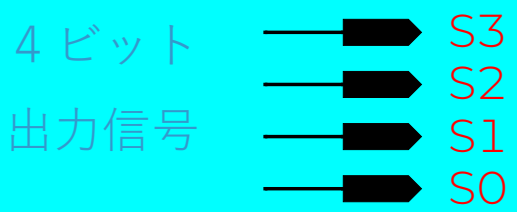
4 ビットの加算（信号の名前と回路）



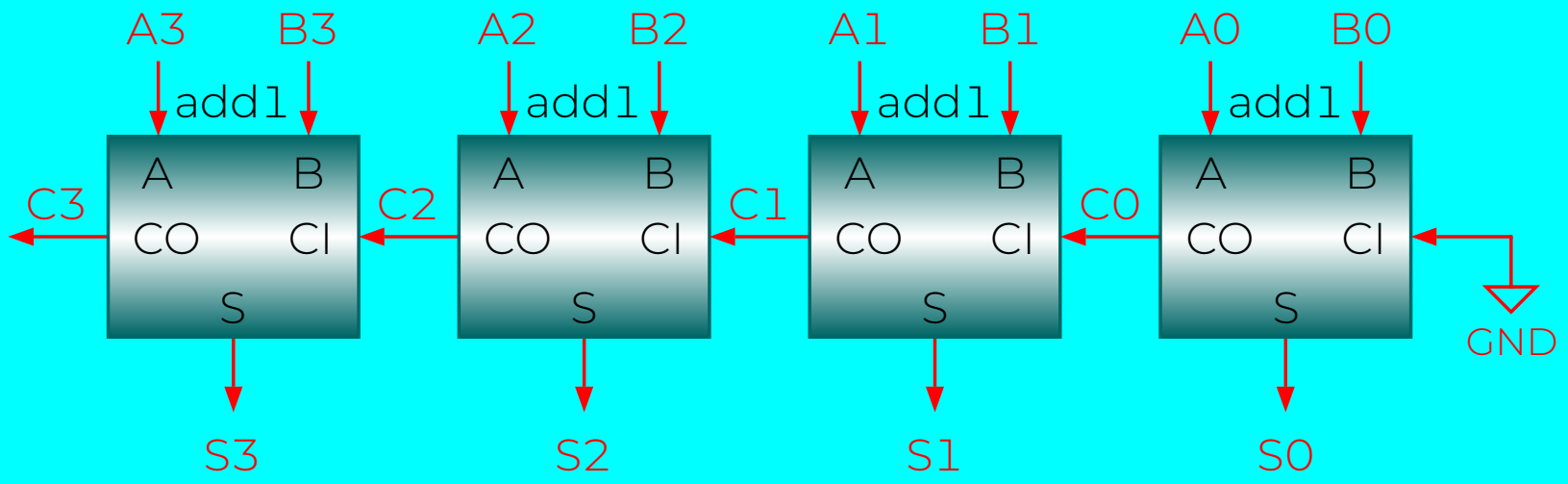
マルチビット (問題点)

4 ビットの加算（入力信号 8 本、出力信号 4 本）





ピン数が多いと作るのは大変



マルチビット (ビットの幅)

4 ビットの加算（入力信号 8 本、出力信号 4 本）

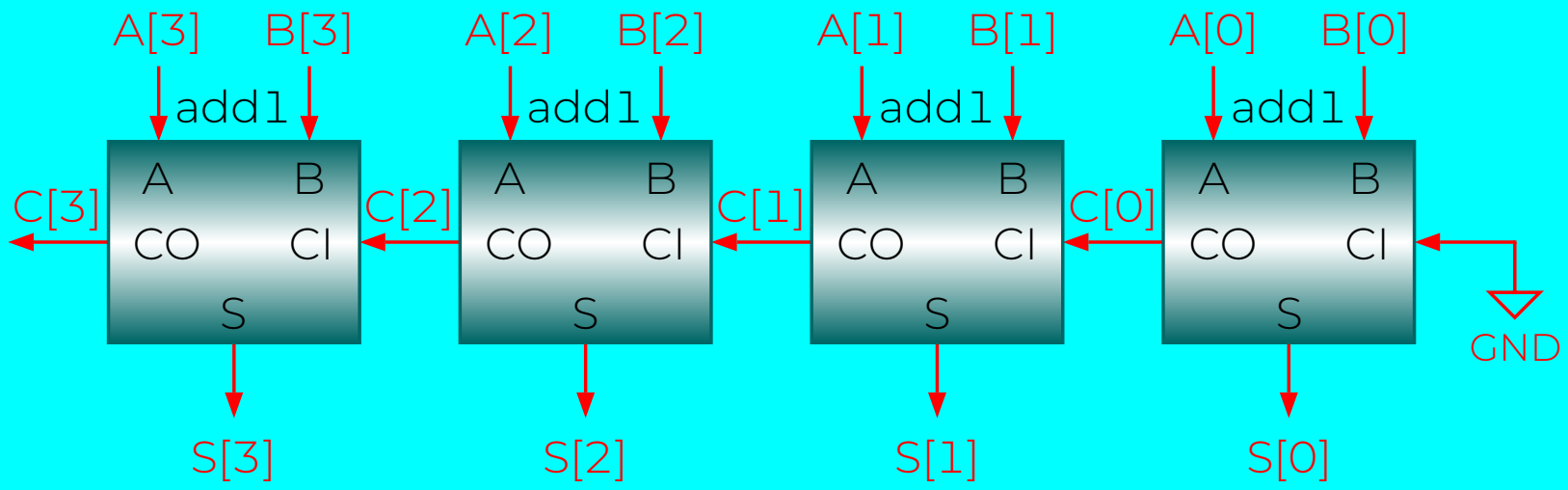
A[3:0] ━━ 4 ビット 入力信号

B[3:0] ━━ 4 ビット 入力信号

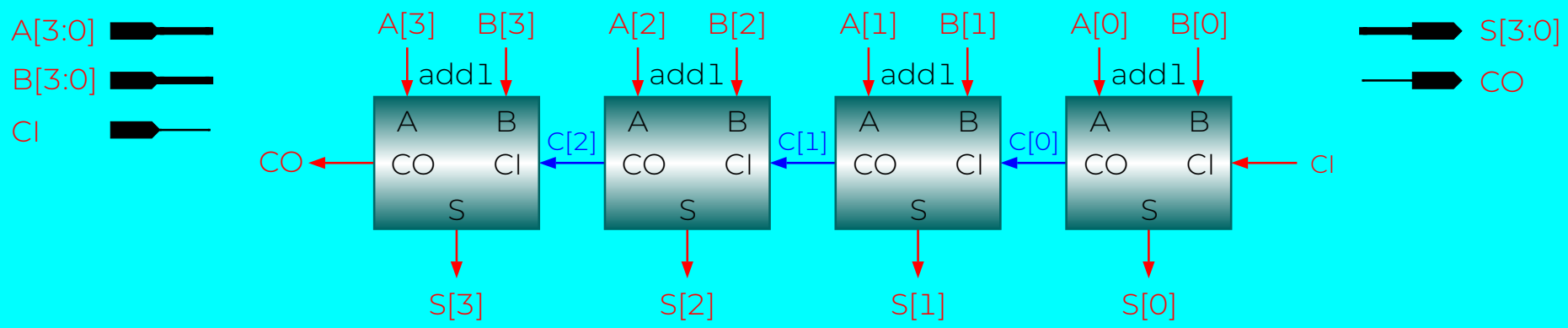
4 ビット 出力信号 ━━ S[3:0]

楽な作り方

[3:0]: 4 ビットのまとめ表現



マルチビット (add4 回路)




```
‘timescale 1ns/1ns
module add4 (A, B, CI, CO, S);
    input        CI;
    input  [3:0] A, B;
    output       CO;
    output [3:0] S;

    wire   [2:0] C;

    // add1 (a,    b,    ci,   co,   s);
    add1 i0 (A[0], B[0], CI,   C[0], S[0]);
    add1 i1 (A[1], B[ ],    ,     , S[ ]);
    add1 i2 (A[2], B[ ],    ,     , S[ ]);
    add1 i3 (A[3], B[ ],    , CO,   S[ ]);
endmodule
```

add4.v

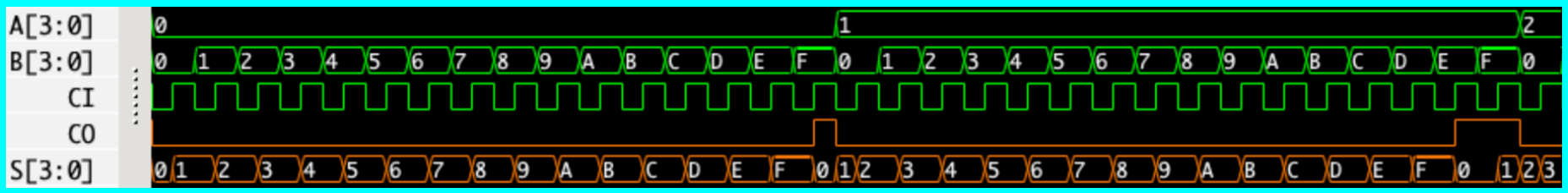
マルチビット (add4 テストベンチ)

```
‘timescale 1ns/1ns
module add4_tb;
    reg        CI;
    reg [3:0]  A, B;
    wire       CO;
    wire [3:0] S;
    add4 i0 (A, B, CI, CO, S);
    integer    i, j, k;
    initial begin
        for (i = 0; i < 16; i = i + 1) begin
            for (j = 0; j < 16; j = j + 1) begin
                for (k = 0; k < 2; k = k + 1) begin
                    A = i; B = j; CI = k;
                    #1 ;
                end
            end
        end
        #1 $finish;
    end
    initial begin
        $dumpfile ("add4.vcd");
        $dumpvars;
    end
endmodule
```

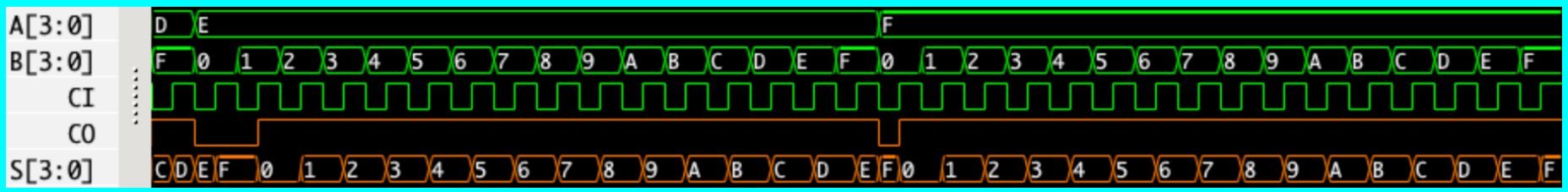
add4_tb.v

マルチビット (add4 波形)

```
% iverilog -Wall -o add4 add4_tb.v \
  add4.v add1.v half_adder.v
% vvp add4
% gtkwave add4.vcd
```



$$\{CO,S[3:0]\} = A[3:0] + B[3:0] + CI$$

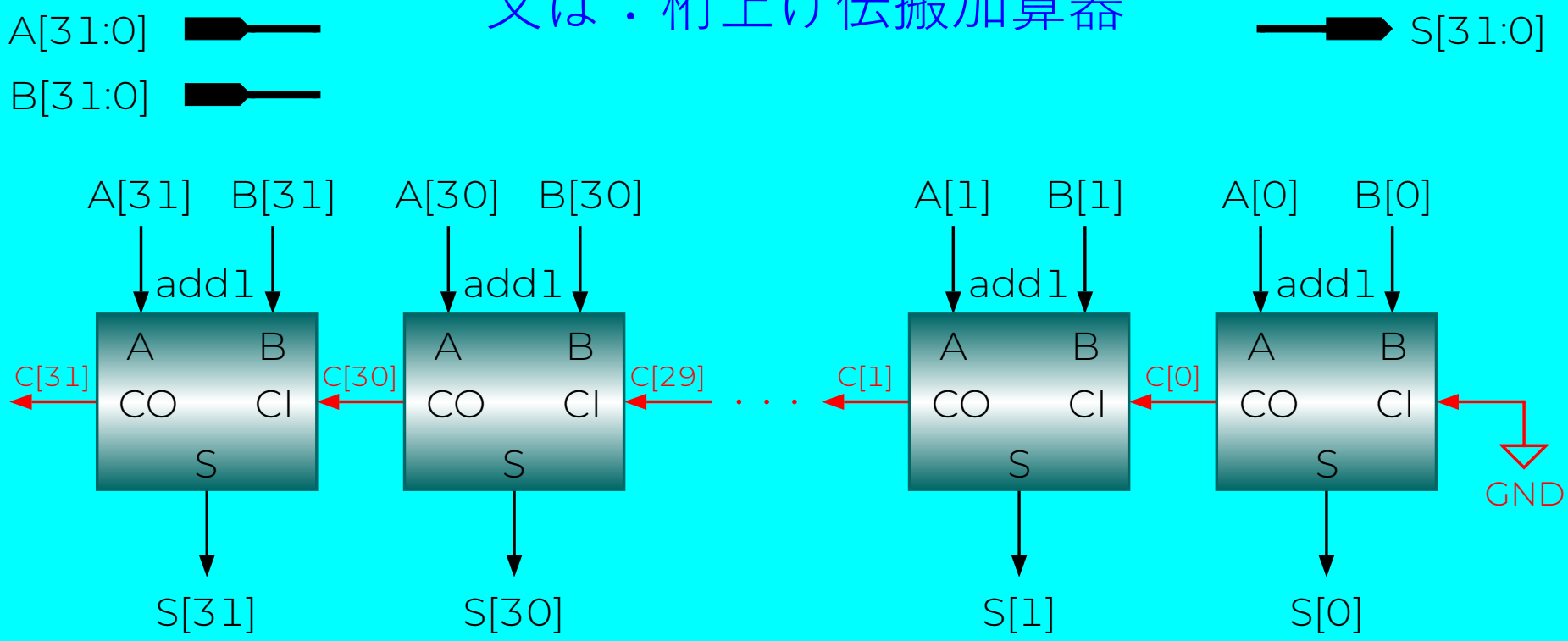


$$\{CO,S[3:0]\} = A[3:0] + B[3:0] + CI$$

リップルキャリー加算器 (RCA)

リップルキャリーアダー (Ripple Carry Adder — RCA)

又は：桁上げ伝搬加算器



キャリー信号が下位の桁 (右) から順番に上がってくる (左へ)。
したがって計算速度が遅い。

桁上げ先見加算器 (CLA)

桁上げ先見加算器 (Carry Lookahead Adder — CLA)

$$
\begin{aligned}
C_i &= A_i \cdot B_i + A_i \cdot C_{i-1} + B_i \cdot C_{i-1} \\
&= A_i \cdot B_i + (A_i + B_i) \cdot C_{i-1} \\
&= G_i + P_i \cdot C_{i-1}
\end{aligned}
$$

$G_i = A_i \cdot B_i$ Carry generator (桁上げ発生器)

$P_i = A_i + B_i$ Carry propagator (桁上げ伝播器)

$$
\begin{aligned}
C_{i-1} &= G_{i-1} + P_{i-1} \cdot C_{i-2} \\
C_{i-2} &= G_{i-2} + P_{i-2} \cdot C_{i-3} \\
&\cdots \\
C_1 &= G_1 + P_1 \cdot C_0 \\
C_0 &= G_0 + P_0 \cdot C_{-1}
\end{aligned}
$$

(C_{-1}: Carry in CI)

すべての C_i、$i = 0, 1, \cdots, 31$ を並列に (同時に) 計算したい。

桁上げ先見加算器 (CLA)

$$
\begin{aligned}
C_i &= G_i + P_i \cdot C_{i-1} \\
C_{i-1} &= G_{i-1} + P_{i-1} \cdot C_{i-2} \\
C_{i-2} &= G_{i-2} + P_{i-2} \cdot C_{i-3} \\
&\cdots \\
C_1 &= G_1 + P_1 \cdot C_0 \\
C_0 &= G_0 + P_0 \cdot C_{-1}
\end{aligned}
$$

代入

$$
\begin{aligned}
C_i &= G_i \\
&+ P_i \cdot G_{i-1} \\
&+ P_i \cdot P_{i-1} \cdot G_{i-2} \\
&+ \cdots \\
&+ P_i \cdot P_{i-1} \cdots P_1 \cdot G_0 \\
&+ P_i \cdot P_{i-1} \cdots P_1 \cdot P_0 \cdot C_{-1}
\end{aligned}
$$

(C_{-1}: Carry in CI)

C_i の式は複雑すぎる。i が大きくなければ OK である。

桁上げ先見加算器 (CLA4)

$$G_0 = A_0 \cdot B_0 \qquad P_0 = A_0 + B_0$$
$$G_1 = A_1 \cdot B_1 \qquad P_1 = A_1 + B_1$$
$$G_2 = A_2 \cdot B_2 \qquad P_2 = A_2 + B_2$$
$$G_3 = A_3 \cdot B_3 \qquad P_3 = A_3 + B_3$$

$$C_0 = G_0 + P_0 \cdot C_{-1}$$ (C_{-1}: Carry in CI)

$$C_1 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_{-1}$$
$$C_2 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_{-1}$$
$$C_3 = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_{-1}$$

C_0 C_1 C_2 C_3 を並列に計算する。回路は次のページにある。

桁上げ先見加算器の回路

```
‘timescale 1ns/1ns
module cla4 (A, B, CI, CO, S);
    input        CI;
    input  [3:0] A, B;
    output       CO;
    output [3:0] S;

    wire   [3:0] G = A & B;
    wire   [3:0] P = A | B;

    wire   [3:0] C;

    assign C[0] = G[0] | P[0]&CI;
    assign C[1] = G[1] | P[1]&G[0] | P[1]&P[0]&CI;
    assign C[2] = G[2] | P[2]&G[1] | P[2]&P[1]&G[0] | P[2]&P[1]&P[0]&CI;
    assign C[3] = G[3] | P[3]&G[2] | P[3]&P[2]&G[1] | P[3]&P[2]&P[1]&G[0]
                                                    | P[3]&P[2]&P[1]&P[0]&CI;
    assign S[0]  = A[0] ^ B[0] ^ CI;
    assign S[1]  = A[1] ^ B[1] ^ C[0];
    assign S[2]  = A[2] ^ B[2] ^ C[1];
    assign S[3]  = A[3] ^ B[3] ^ C[2];

    assign CO = C[3];
endmodule
```

cla4.v

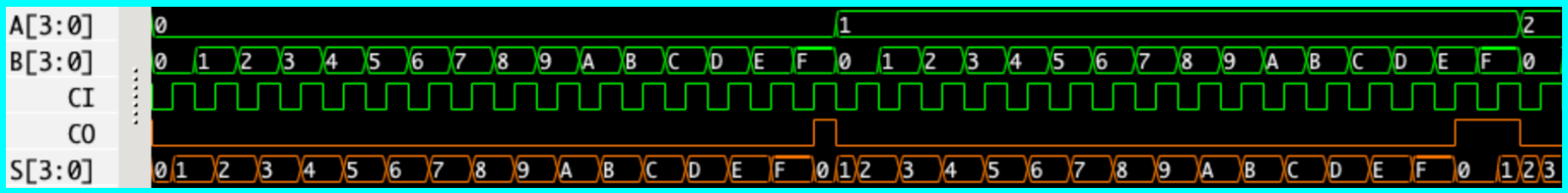
桁上げ先見加算器のテストベンチ

```
‘timescale 1ns/1ns
module cla4_tb;
    reg        CI;
    reg [3:0]  A, B;
    wire       CO;
    wire [3:0] S;
    cla4 i0 (A, B, CI, CO, S);
    integer    i, j, k;
    initial begin
        for (i = 0; i < 16; i = i + 1) begin
            for (j = 0; j < 16; j = j + 1) begin
                for (k = 0; k < 2; k = k + 1) begin
                    A = i; B = j; CI = k;
                    #1 ;
                end
            end
        end
        #1 $finish;
    end
    initial begin
        $dumpfile ("cla4.vcd");
        $dumpvars;
    end
endmodule
```

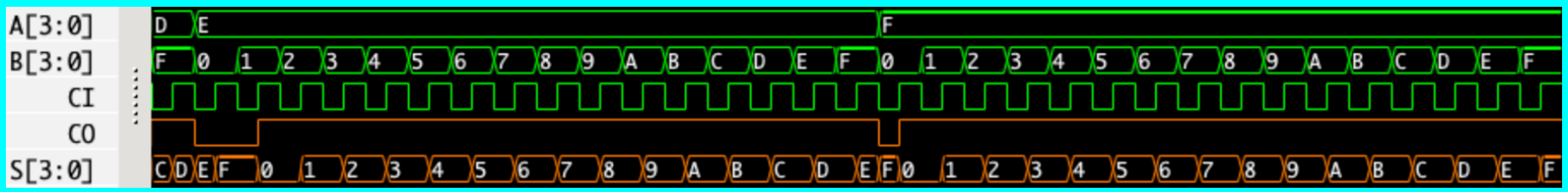
cla4_tb.v

桁上げ先見加算器の波形

```
% iverilog -Wall -o cla4 cla4_tb.v \
  cla4.v add1.v half_adder.v
% vvp cla4
% gtkwave cla4.vcd
```

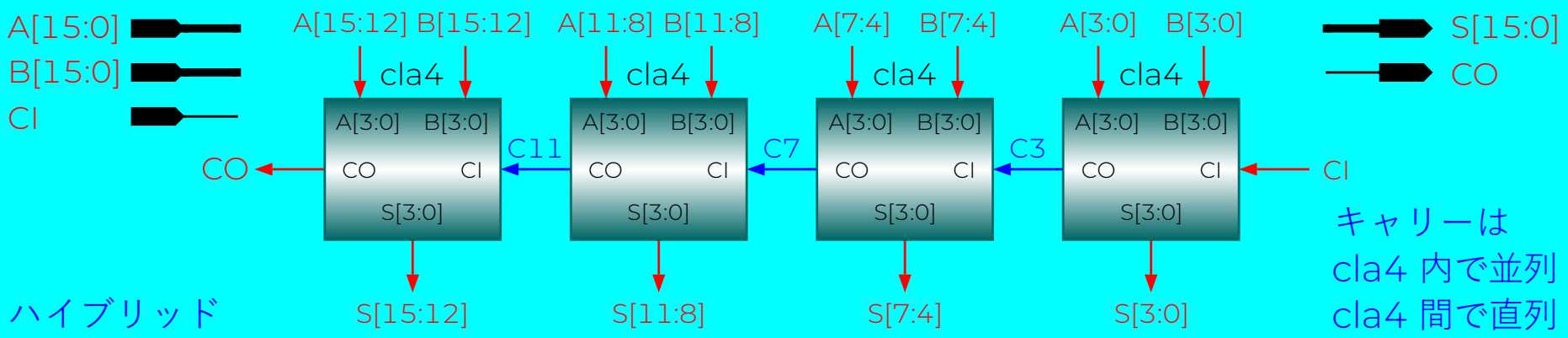


$$\{CO,S[3:0]\} = A[3:0] + B[3:0] + CI$$



$$\{CO,S[3:0]\} = A[3:0] + B[3:0] + CI$$

CLA4 を使用する 16 ビット加算器




```
‘timescale 1ns/1ns
module add16 (A, B, CI, CO, S);
    input         CI;
    input  [15:0] A, B;
    output        CO;
    output [15:0] S;
    wire          C3, C7, C11;
    // cla4 (A,         B,         CI,  CO,  S);
    cla4 i0 (A[3:0],   B[3:0],   CI,  C3,  S[3:0]);
    cla4 i1 (A[7:4],   B[7:4],   C3,  C7,  S[7:4]);
    cla4 i2 (A[11:8],  B[11:8],  C7,  C11, S[11:8]);
    cla4 i3 (A[15:12], B[15:12], C11, CO,  S[15:12]);
endmodule
```

add16.v

CLA4 を使用する 16 ビット加算器

```
‘timescale 1ns/1ns
module add16_tb;
    reg         CI;
    reg  [15:0] A, B;
    wire        CO;
    wire [15:0] S;
    add16 i0 (A, B, CI, CO, S);
    initial begin
        #0 A  = 16’h5555; B  = 16’haaaa; CI = 0;
        #1 CI = 1;
        #1 B  = 16’haaab; CI = 0;
        #1 CI = 1;
        #1 $finish;
    end

    initial begin
        $dumpfile ("add16.vcd");
        $dumpvars;
    end
endmodule
```

add16_tb.v

CLA4 を使用する 16 ビット加算器

```
% iverilog -Wall -o add16 add16_tb.v \
  add16.v add1.v half_adder.v
% vvp add16
% gtkwave add16.vcd
```

２進　　A、B、S 各 16 ビット

```
   A[15:0] 0101010101010101        0101010101010101
   B[15:0] 1010101010101010        1010101010101010
 +      CI                0      +               1
-------------------------------------------------------
   S[15:0] 1111111111111111        0000000000000000
```

CLA4 を使用する 16 ビット加算器

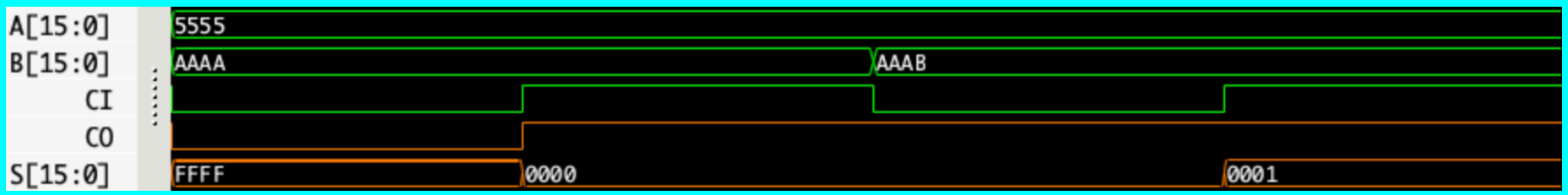
```
% iverilog -Wall -o add16 add16_tb.v \
  add16.v add1.v half_adder.v
% vvp add16
% gtkwave add16.vcd
```

２進　　A、B、S 各 16 ビット

```
   A[15:0] 0101010101010101      0101010101010101
   B[15:0] 1010101010101011      1010101010101011
 +      CI                0    +               1
-------------------------------------------------
   S[15:0] 0000000000000000      0000000000000001
```

CLA4 を使用する 16 ビット加算器

```
% iverilog -Wall -o add16 add16_tb.v \
  add16.v add1.v half_adder.v
% vvp add16
% gtkwave add16.vcd
```

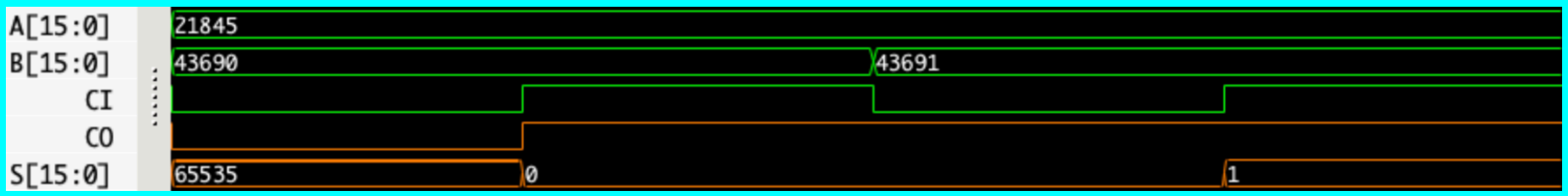


16 進　　A、B、S 各 16 ビット

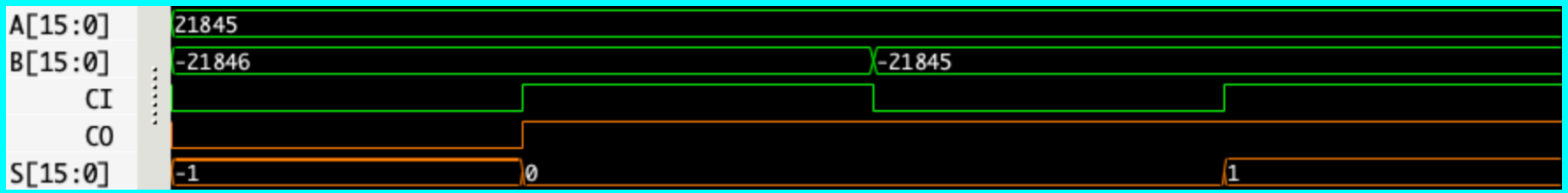
A[15:0]	5555	5555	5555	5555
B[15:0]	AAAA	AAAA	AAAB	AAAB
+ CI	0	+ 1	+ 0	+ 1
S[15:0]	FFFF	0000	0000	0001

CLA4 を使用する 16 ビット加算器

```
% iverilog -Wall -o add16 add16_tb.v \
  add16.v add1.v half_adder.v
% vvp add16
% gtkwave add16.vcd
```



10 進 (絶対値)　　A、B、S 各 16 ビット



10 進 (整数)　　A、B、S 各 16 ビット

Radix の例

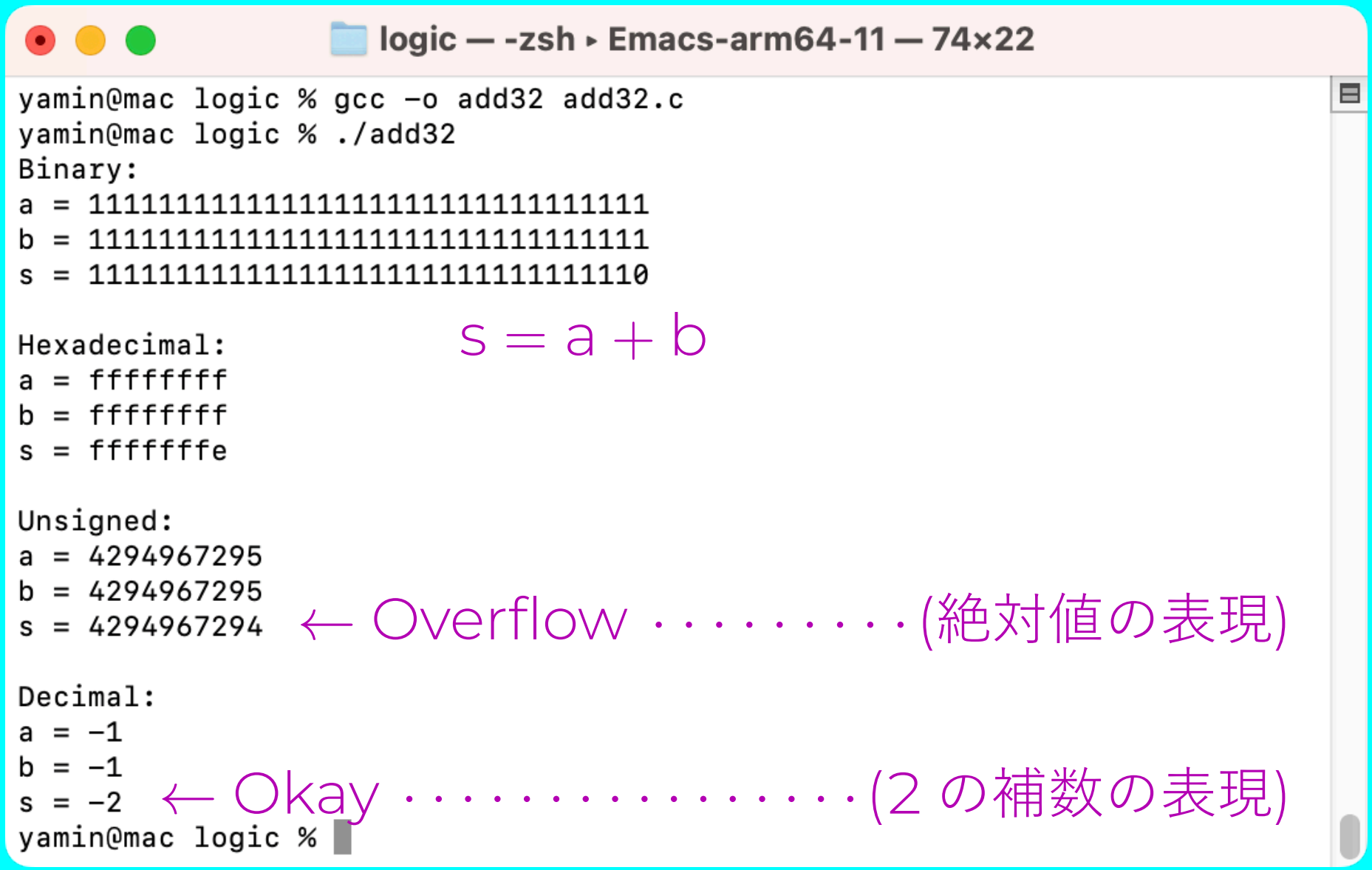
add32.c

```c
#include<stdio.h>
void print_binary(unsigned int number) {
    int i;
    unsigned int one_bit;
    for (i = 31; i >= 0; i--) {
        one_bit = (number >> i) & 1;
        putc((one_bit == 0) ? '0' : '1', stdout);
    }
}
int main() {
    unsigned int a = 0xffffffff;
    unsigned int b = 0xffffffff;
    unsigned int s = a + b;
    printf("Binary:\n"); // print a, b, s in binary
    printf("a = "); print_binary(a); printf("\n");
    printf("b = "); print_binary(b); printf("\n");
    printf("s = "); print_binary(s); printf("\n\n");
    printf("Hexadecimal:\n"); // print a, b, s in hexadecimal
    printf("a = %x\n", a);
    printf("b = %x\n", b);
    printf("s = %x\n\n", s);
    printf("Unsigned:\n"); // print a, b, s in unsigned
    printf("a = %u\n", a);
    printf("b = %u\n", b);
    printf("s = %u\n\n", s);
    printf("Decimal:\n"); // print a, b, s in decimal
    printf("a = %d\n", a);
    printf("b = %d\n", b);
    printf("s = %d\n", s);
    return 0;
}
```

$s = a + b$

-:--- **add32.c** All L32 (C/*l Abbrev)

Radix の例

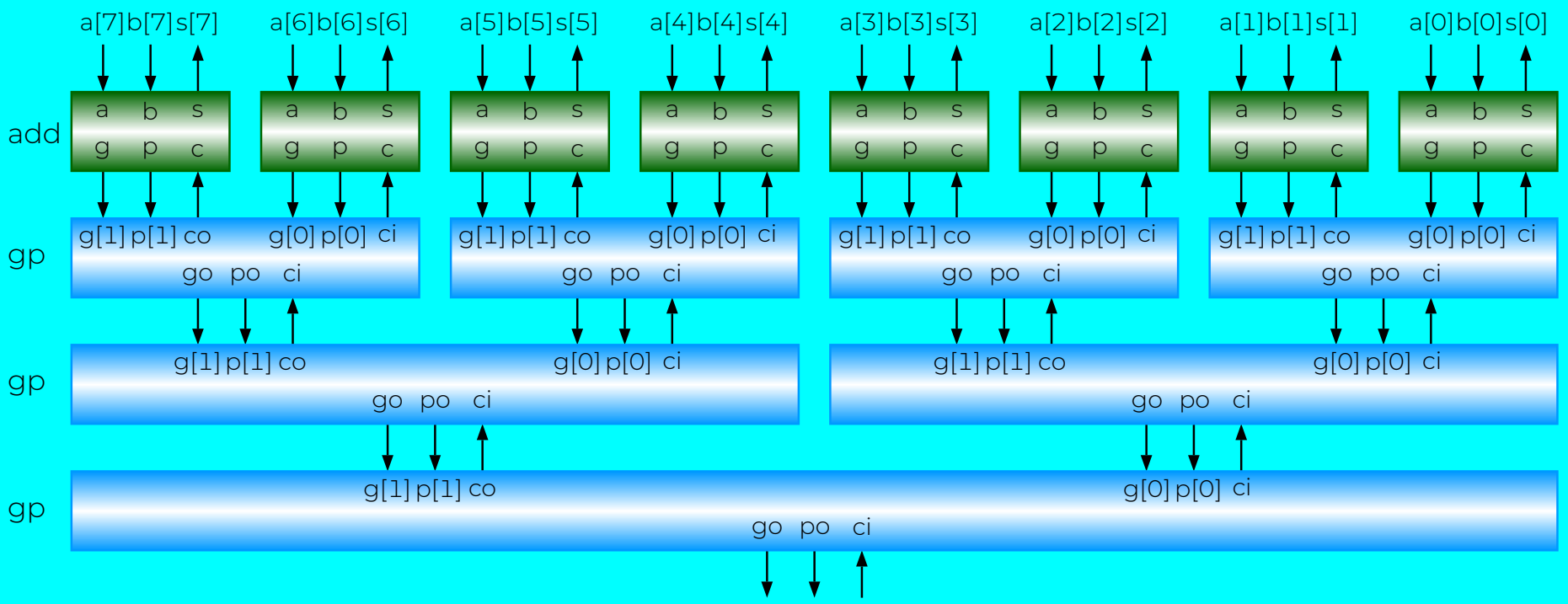


$s = a + b$

← Overflow ・・・・・・・・・(絶対値の表現)

← Okay ・・・・・・・・・・・・・・・・(2 の補数の表現)

ツリー型桁上げ先見加算器の回路



add: $s = a \oplus b \oplus c$

$g = a \cdot b$

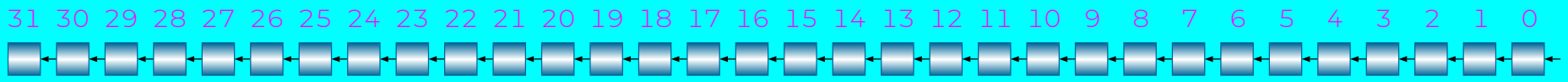
$p = a + b$

gp: $co = g[0] + p[0] \cdot ci$

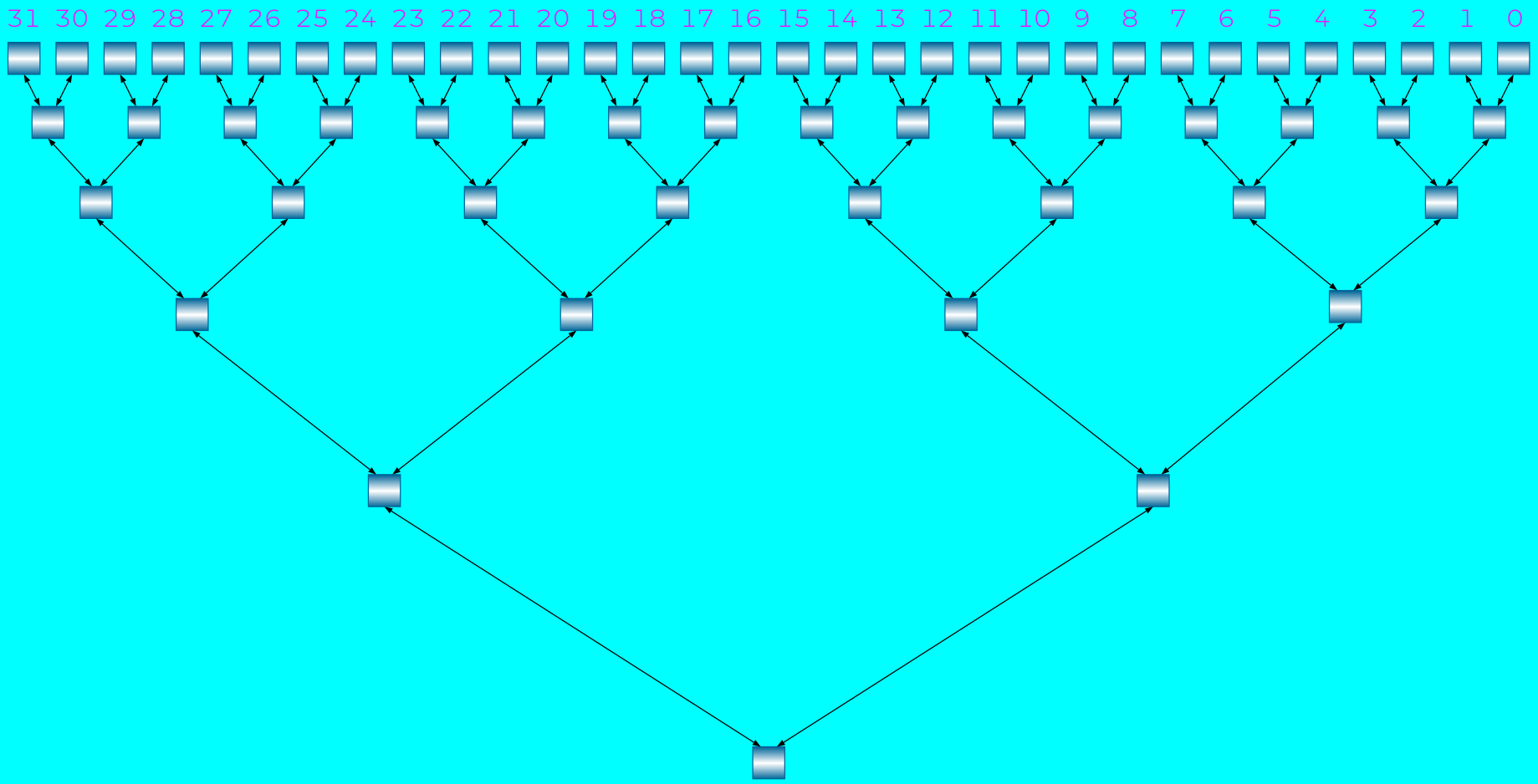
$go = g[1] + p[1] \cdot g[0]$

$po = p[1] \cdot p[0]$

加算器の遅延時間

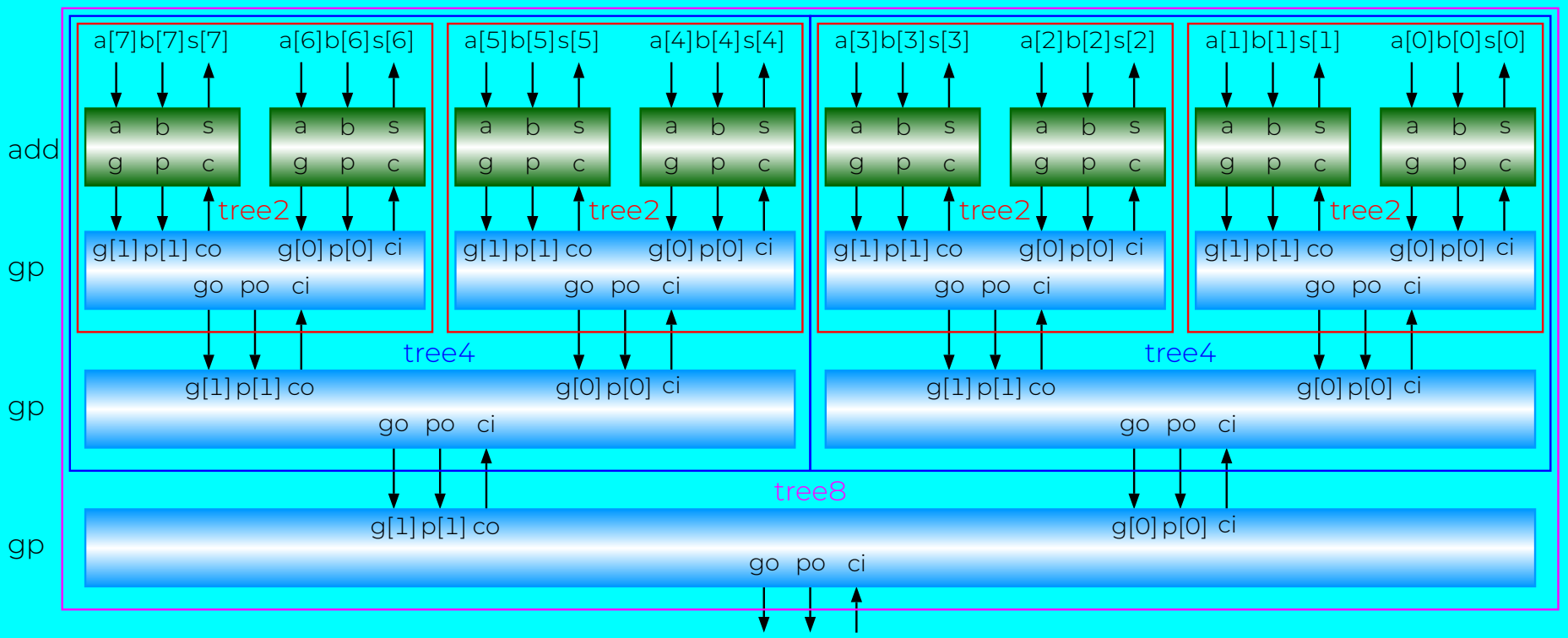


リップルキャリー加算器の遅延時間 $O(n)$



ツリー型桁上げ先見加算器の遅延時間 $O(\log_2 n)$

ツリー型桁上げ先見加算器の回路



この回路は階層的に設計できる。設計順番:

tree2 (2 ビット加算器、二つの add と一つの gp を使用) ⇒
tree4 (4 ビット加算器、二つの tree2 と一つの gp を使用) ⇒
tree8 (8 ビット加算器、二つの tree4 と一つの gp を使用)

ツリー型桁上げ先見加算器の回路

1 ビットツリー型桁上げ先見加算器の回路

```
‘timescale 1ns/1ns
module add (a, b, c, g, p, s);
    input  a, b, c;
    output g, p, s;
    assign s = a ^ b ^ c;
    assign g = a & b;
    assign p = a | b;
endmodule
```

add.v

ツリー型桁上げ先見加算器の回路

ツリー型桁上げ先見加算器の GP の回路

```
‘timescale 1ns/1ns
module g_p (g, p, c_in, g_out, p_out, c_out);
    input [1:0] g, p;
    input       c_in;
    output      g_out, p_out, c_out;
    assign      c_out = g[0] | p[0] & c_in;
    assign      g_out = g[1] | p[1] & g[0];
    assign      p_out = p[1] & p[0];
endmodule
```

g_p.v

ツリー型桁上げ先見加算器の回路

2 ビットツリー型桁上げ先見加算器の回路

```
‘timescale 1ns/1ns
module tree_2 (a, b, c_in, g_out, p_out, s);
    input  [1:0] a, b;
    input        c_in;
    output       g_out, p_out;
    output [1:0] s;
    wire   [1:0] g, p;
    wire         c_out;
    add a0 (a[0], b[0], c_in,  g[0], p[0], s[0]);
    add a1 (a[1], b[1], c_out, g[1], p[1], s[1]);
    g_p gp (g, p, c_in, g_out, p_out, c_out);
endmodule
```

tree_2.v

ツリー型桁上げ先見加算器の回路

4 ビットツリー型桁上げ先見加算器の回路

```
‘timescale 1ns/1ns
module tree_4 (a, b, c_in, g_out, p_out, s);
    input  [3:0] a, b;
    input        c_in;
    output       g_out, p_out;
    output [3:0] s;
    wire   [1:0] g, p;
    wire         c_out;
    tree_2 a0 (a[1:0], b[1:0], c_in,  g[0], p[0], s[1:0]);
    tree_2 a1 (a[3:2], b[3:2], c_out, g[1], p[1], s[3:2]);
    g_p    gp (g, p, c_in, g_out, p_out, c_out);
endmodule
```

tree_4.v

ツリー型桁上げ先見加算器の回路

8ビットツリー型桁上げ先見加算器の回路

```
‘timescale 1ns/1ns
module tree_8 (a, b, c_in, g_out, p_out, s);
    input  [7:0] a, b;
    input        c_in;
    output       g_out, p_out;
    output [7:0] s;
    wire   [1:0] g, p;
    wire         c_out;
    tree_4 a0 (a[   ], b[   ], c_in,  g[0], p[0], s[   ]);
    tree_4 a1 (a[   ], b[   ], c_out, g[1], p[1], s[   ]);
    g_p    gp (g, p, c_in, g_out, p_out, c_out);
endmodule
```

tree_8.v

ツリー型桁上げ先見加算器の回路

16 ビットツリー型桁上げ先見加算器の回路

```
‘timescale 1ns/1ns
module tree_16 (a, b, c_in, g_out, p_out, s);
    input  [15:0] a, b;
    input         c_in;
    output        g_out, p_out;
    output [15:0] s;
    wire    [1:0] g, p;
    wire          c_out;
    tree_8 a0 (a[7:0], b[7:0], c_in,  g[0], p[0], s[7:0]);
    tree_8 a1 (a[15:8], b[15:8], c_out, g[1], p[1], s[15:8]);
    g_p    gp (g, p, c_in, g_out, p_out, c_out);
endmodule
```

tree_16.v

ツリー型桁上げ先見加算器の回路

32 ビットツリー型桁上げ先見加算器の回路

```
‘timescale 1ns/1ns
module tree_32 (a, b, c_in, g_out, p_out, s);
    input  [31:0] a, b;
    input         c_in;
    output        g_out, p_out;
    output [31:0] s;
    wire    [1:0] g, p;
    wire          c_out;
    tree_16 a0 (a[15:0], b[15:0], c_in,  g[0], p[0], s[15:0]);
    tree_16 a1 (a[31:16], b[31:16], c_out, g[1], p[1], s[31:16]);
    g_p     gp (g, p, c_in, g_out, p_out, c_out);
endmodule
```

tree_32.v

ツリー型桁上げ先見加算器テストベンチ

```
‘timescale 1ns/1ns
module tree_32_tb;
    reg  [31:0] a,b;
    reg         ci;
    wire        g,p;
    wire [31:0] s;
    tree_32 adder_tree (a,b,ci,g,p,s);
    initial begin
            a  = 32’h77777777;
            b  = 32’hffffffff;
            ci = 0;
        #10 a  = 32’haaaaaaaa;
            b  = 32’h55555555;
        #10 a  = 32’hcccccccc;
            b  = 32’hcccccccc;
        #10 a  = 32’h00000000;
            b  = 32’h00000000;
        #10 $finish;
    end
    always #5 ci = ~ci;

    initial begin
        $dumpfile ("tree_32.vcd");
        $dumpvars;
    end
endmodule
```

tree_32_tb.v

ツリー型桁上げ先見加算器の波形

```
% iverilog -Wall -o tree_32 tree_32_tb.v tree_32.v \
  tree_16.v tree_8.v tree_4.v tree_2.v add.v g_p.v
% vvp tree_32
% gtkwave tree_32.vcd
```

3 グループ信号：16 進、10 進 (絶対値)、10 進 (整数)

マルチビット加算回路

まとめ

- マルチビット
- マルチビットの加算
- リップルキャリー加算器 (RCA)

  RCA: Ripple-Carry Adder
- 桁上げ先見加算器 (CLA)

  CLA: Carry-Look-ahead Adder
- ツリー型桁上げ先見加算器の回路

課題 VI (100 点 + 100 点)

問題：P21-P23 を参照し、全加算器 add1 を使って、6 ビットリップルキャリーアダー add6 を設計し動作検証シミュレーションして下さい。

入力信号: A[5:0] . . . . . 6-bit data
入力信号: B[5:0] . . . . . 6-bit data
入力信号: CI . . . . . 1-bit carry in
出力信号: S[5:0] . . . . . 6-bit result
出力信号: CO . . . . . 1-bit carry out

モジュール名は add6 にすること。
テストベンチ add6_tb.v を使って下さい。

課題 VI (100 点 + 100 点)

オプション (+100 点) : P39-P50 を参照し、8 ビットツリー型桁上げ先見加算器の回路を設計し動作検証シミュレーションして下さい。

入力信号: a[7:0] . . . . . 8-bit data
入力信号: b[7:0] . . . . . 8-bit data
入力信号: c_in . . . . . 1-bit carry in
出力信号: s[7:0] . . . . . 8-bit result
出力信号: g_out . . . . . 1-bit carry generator
出力信号: p_out . . . . . 1-bit carry propagator

モジュール名は tree_8 にすること。
テストベンチ tree_8_tb.v を使って下さい。