論理回路入門（6）

マルチビット加算回路

李　亜民

2022 年 11 月 1 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　S

C: Carry out (上位桁への繰り上がり)

S: Sum (和)

復習：半加算器回路設計

真理値表

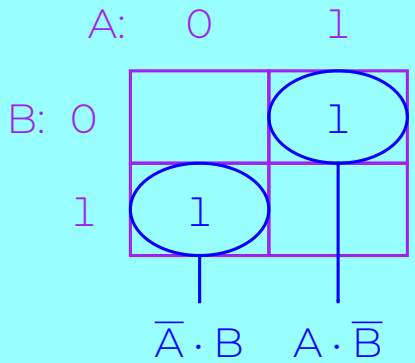
A	B	C	S	コメント
0	0	0	0	$0+0=0\,0$ (加算)
0	1	0	1	$0+1=0\,1$ (加算)
1	0	0	1	$1+0=0\,1$ (加算)
1	1	1	0	$1+1=1\,0$ (加算)

論理式

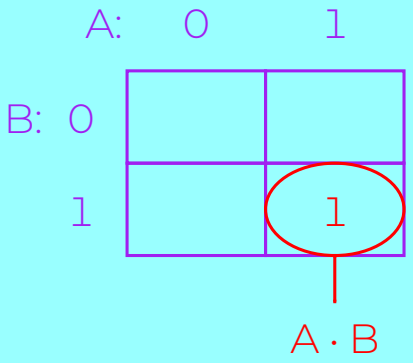
$$S = \overline{A}\cdot B + A\cdot\overline{B} = A \oplus B$$

$$C = A\cdot B$$

S のカルノー図



C のカルノー図

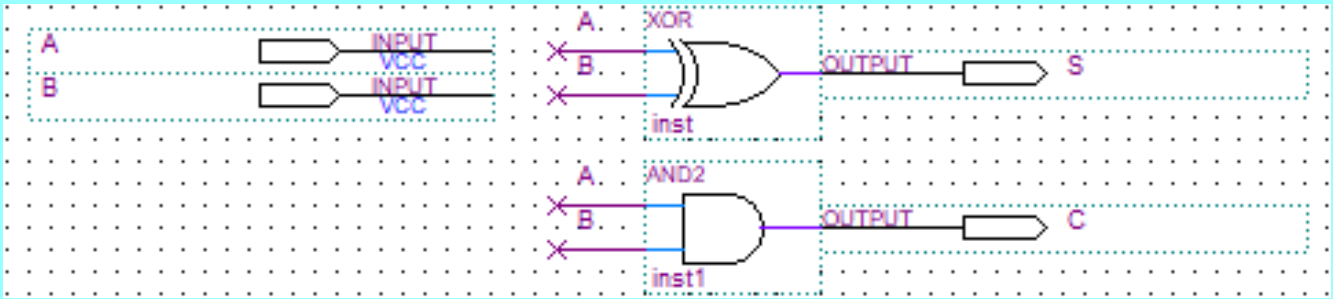


復習：半加算器回路設計

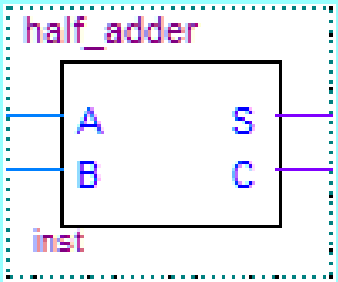
論理式　　$S = \overline{A} \cdot B + A \cdot \overline{B} = A \oplus B$

$C = A \cdot B$

回路図（half_adder.bdf）



シンボル（記号）（half_adder.bsf）



この半加算器シンボルは、

全加算器を作成するときに使用される。

復習：全加算器回路設計

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	$0+0+0=00$ (加算)
0	0	1	0	1	$0+0+1=01$ (加算)
0	1	0	0	1	$0+1+0=01$ (加算)
0	1	1	1	0	$0+1+1=10$ (加算)
1	0	0	0	1	$1+0+0=01$ (加算)
1	0	1	1	0	$1+0+1=10$ (加算)
1	1	0	1	0	$1+1+0=10$ (加算)
1	1	1	1	1	$1+1+1=11$ (加算)

出力の CO、S は A + B + CI の結果 (和) を 2 進数で表した数値。
$00_2 = 0_{10}$　　$01_2 = 1_{10}$　　$10_2 = 2_{10}$　　$11_2 = 3_{10}$

復習：全加算器回路設計

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)

S のカルノー図

CI \ A B	0 0	0 1	1 1	1 0
0		1		1
1	1		1	

CO のカルノー図

CI \ A B	0 0	0 1	1 1	1 0
0			1	
1		1	1	1

B CI　A B　A CI

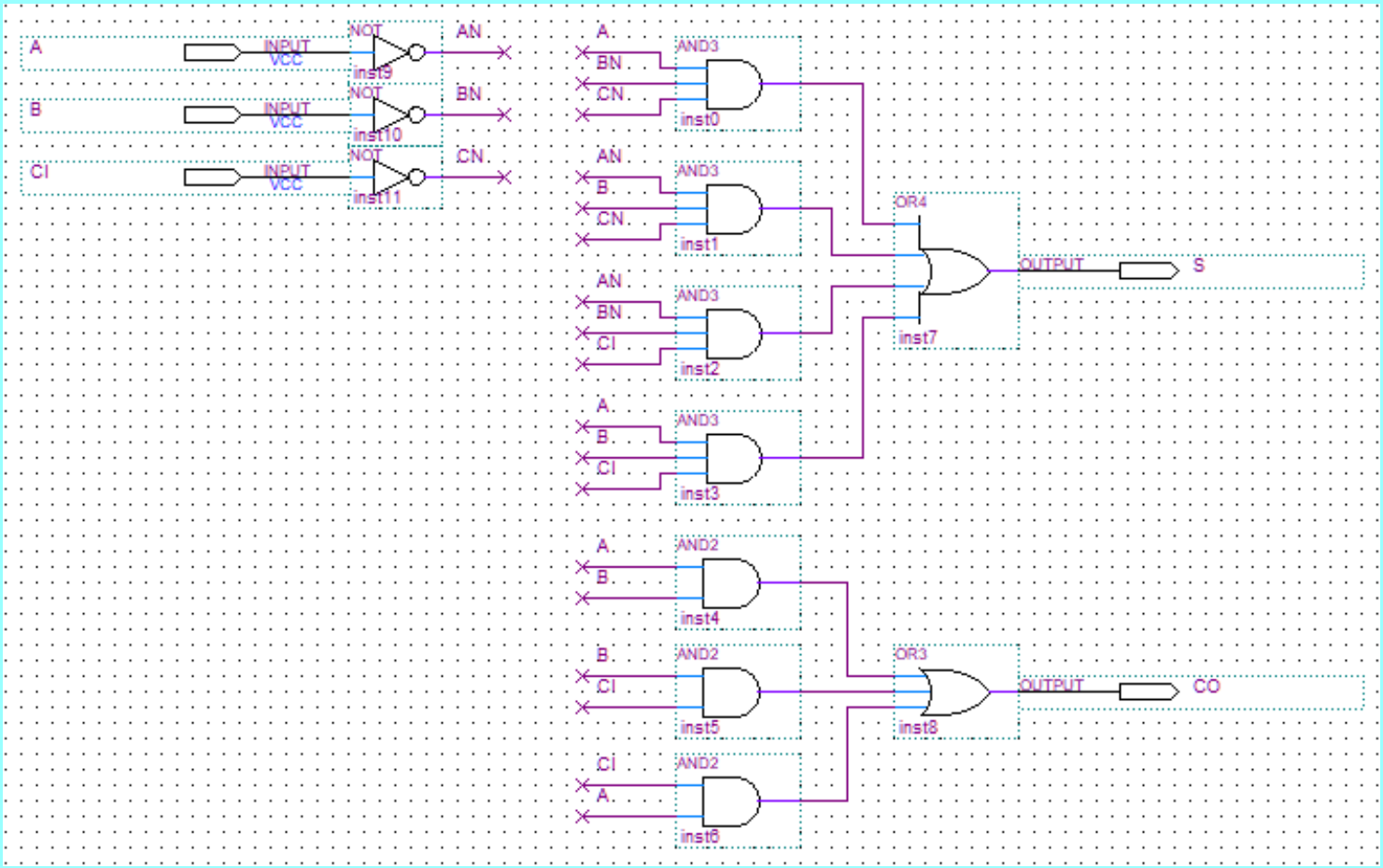
$$S = A\,\overline{B}\,\overline{CI} + \overline{A}\,B\,\overline{CI} + \overline{A}\,\overline{B}\,CI + A\,B\,CI$$

$$CO = A\,B + A\,CI + B\,CI$$ (二つの入力が 11 であれば)

復習：全加算器の回路図

$$S = A\,\overline{B}\,\overline{CI} + \overline{A}\,B\,\overline{CI} + \overline{A}\,\overline{B}\,CI + A\,B\,CI$$

$$CO = A\,B + A\,CI + B\,CI \text{ (二つの入力が 1 1 であれば)}$$

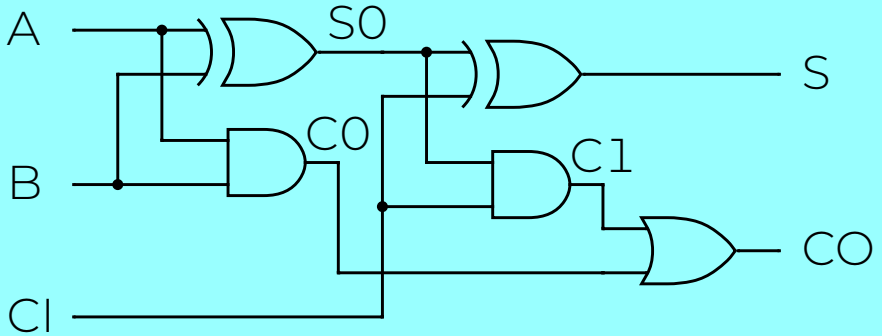


復習：半加算器で全加算器の設計

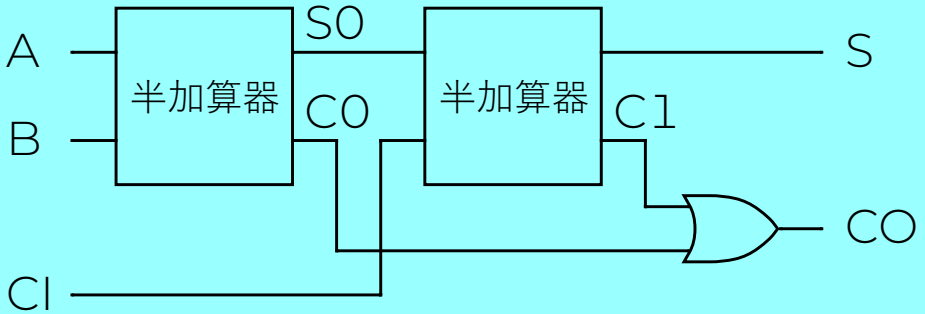
全加算器は、２つの半加算器と１つのOR回路を用いて構成することができる。

$$S = A\,\overline{B}\,\overline{CI} + \overline{A}\,B\,\overline{CI} + \overline{A}\,\overline{B}\,CI + A\,B\,CI = (A \oplus B) \oplus CI$$

$$\begin{aligned} CO &= \overline{A}\,B\,CI + A\,\overline{B}\,CI + A\,B\,\overline{CI} + A\,B\,CI \\ &= (\overline{A}\,B + A\,\overline{B})\,CI + A\,B\,(\overline{CI} + CI) \\ &= (A \oplus B)\,CI + A\,B \cdots \text{(A、B いずれかかつ CI または A かつ B)} \end{aligned}$$



全加算器



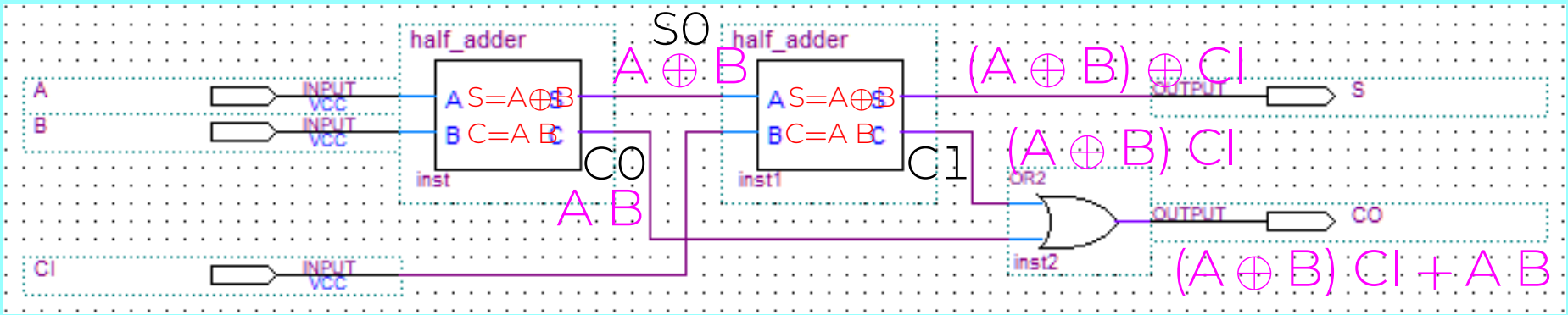
全加算器

復習：半加算器で全加算器の設計

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。

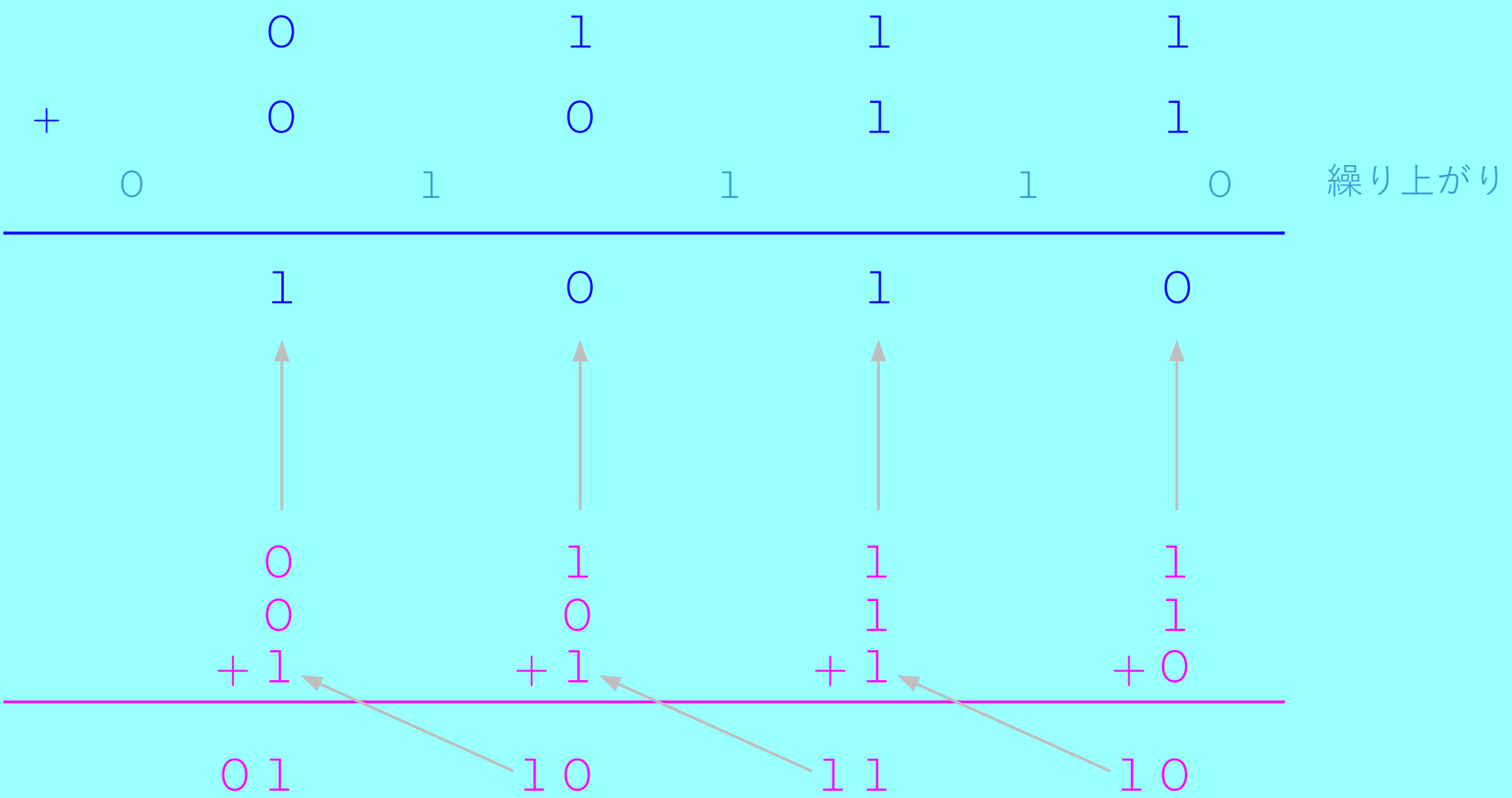
$$S = A\,\overline{B}\,\overline{CI} + \overline{A}\,B\,\overline{CI} + \overline{A}\,\overline{B}\,CI + A\,B\,CI = (A \oplus B) \oplus CI$$

$$\begin{aligned} CO &= \overline{A}\,B\,CI + A\,\overline{B}\,CI + A\,B\,\overline{CI} + A\,B\,CI \\ &= (\overline{A}\,B + A\,\overline{B})\,CI + A\,B\,(\overline{CI} + CI) \\ &= (A \oplus B)\,CI + A\,B \cdots \text{(A、B いずれかかつ CI または A かつ B)} \end{aligned}$$



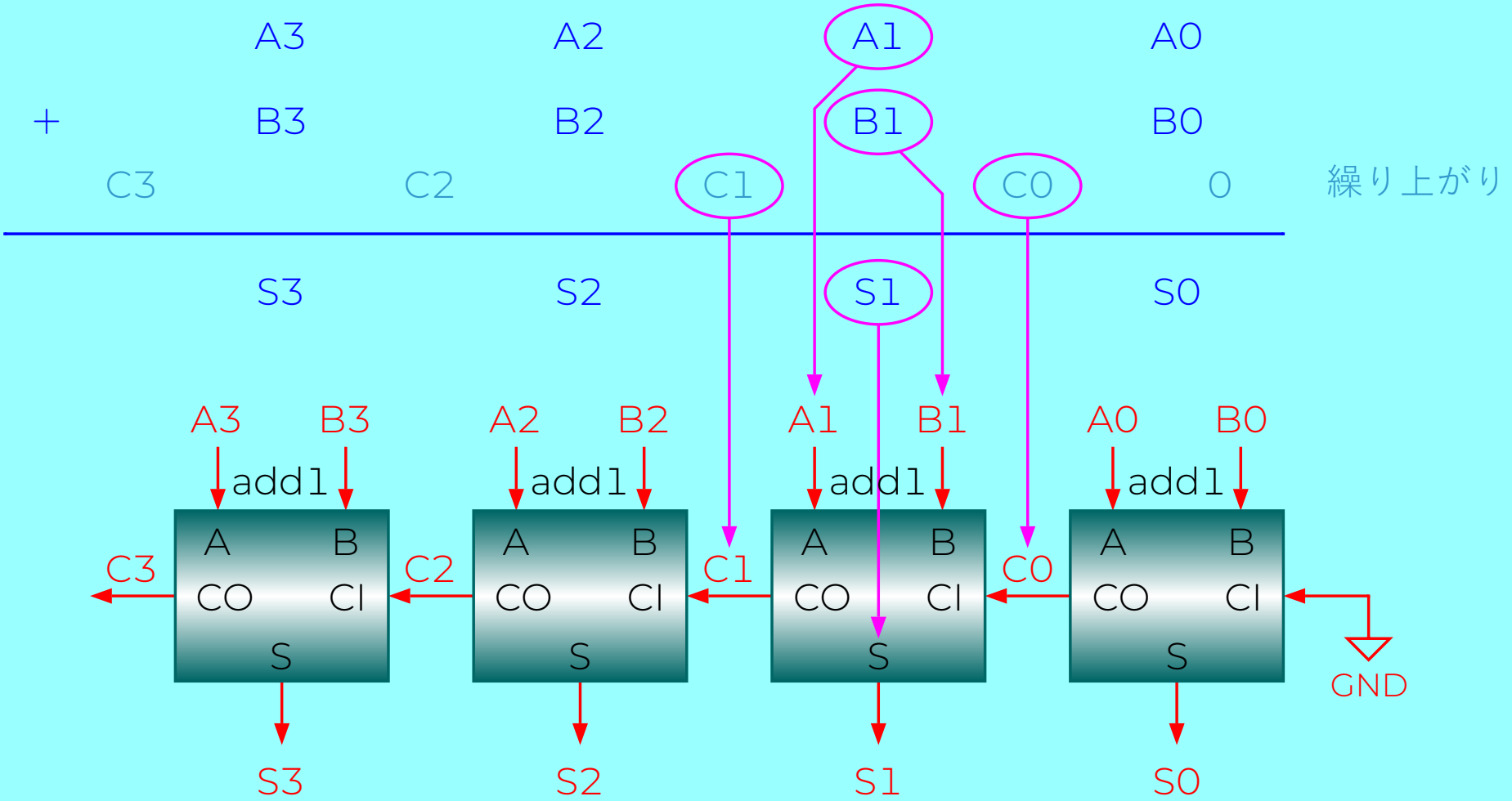
マルチビット

4 ビットの加算



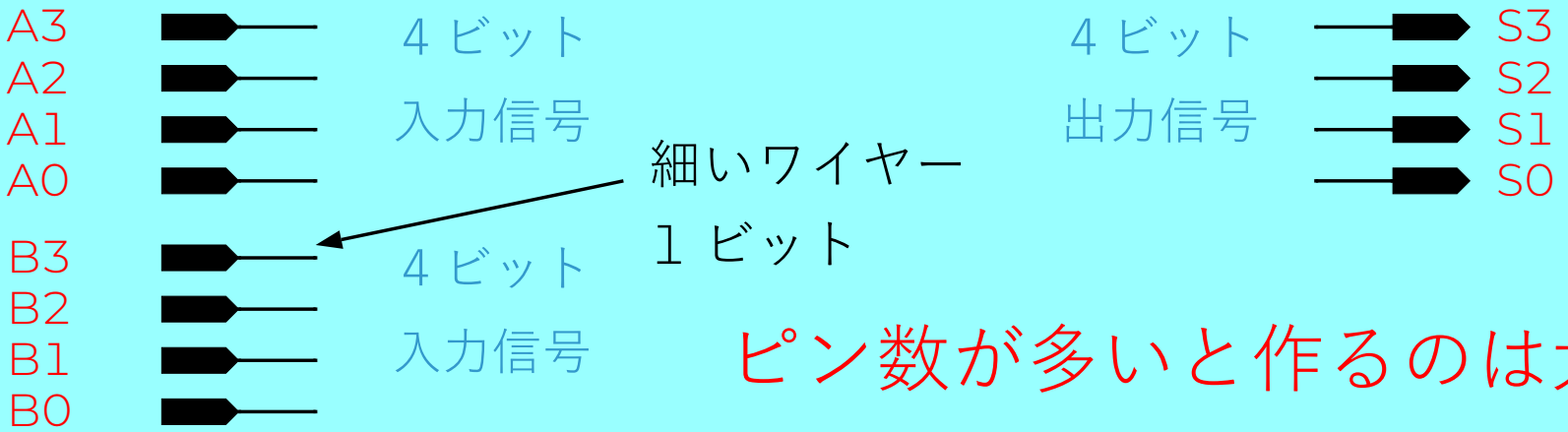
マルチビット

4 ビットの加算（信号の名前と回路）

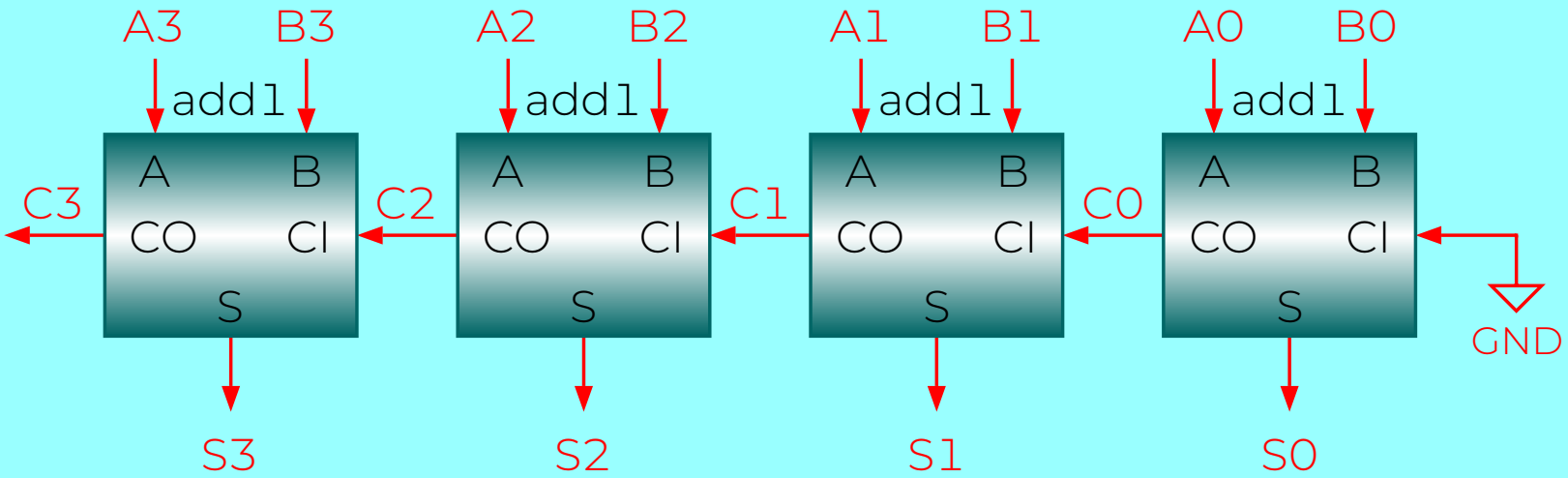


マルチビット

４ビットの加算（入力信号８本、出力信号４本）

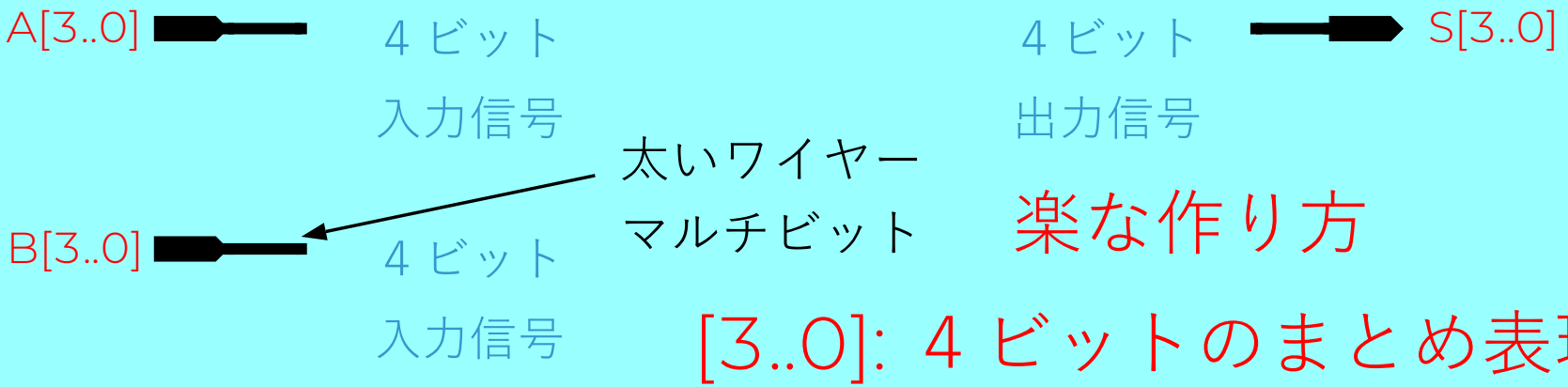


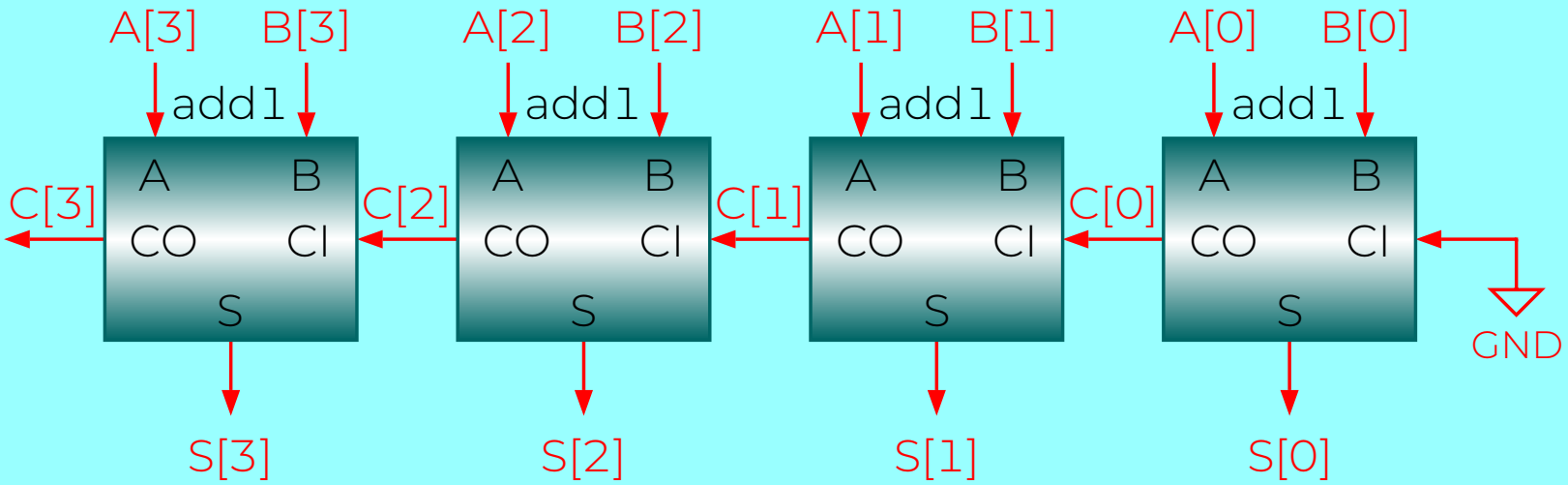
ピン数が多いと作るのは大変



マルチビット

4 ビットの加算（入力信号 8 本、出力信号 4 本）

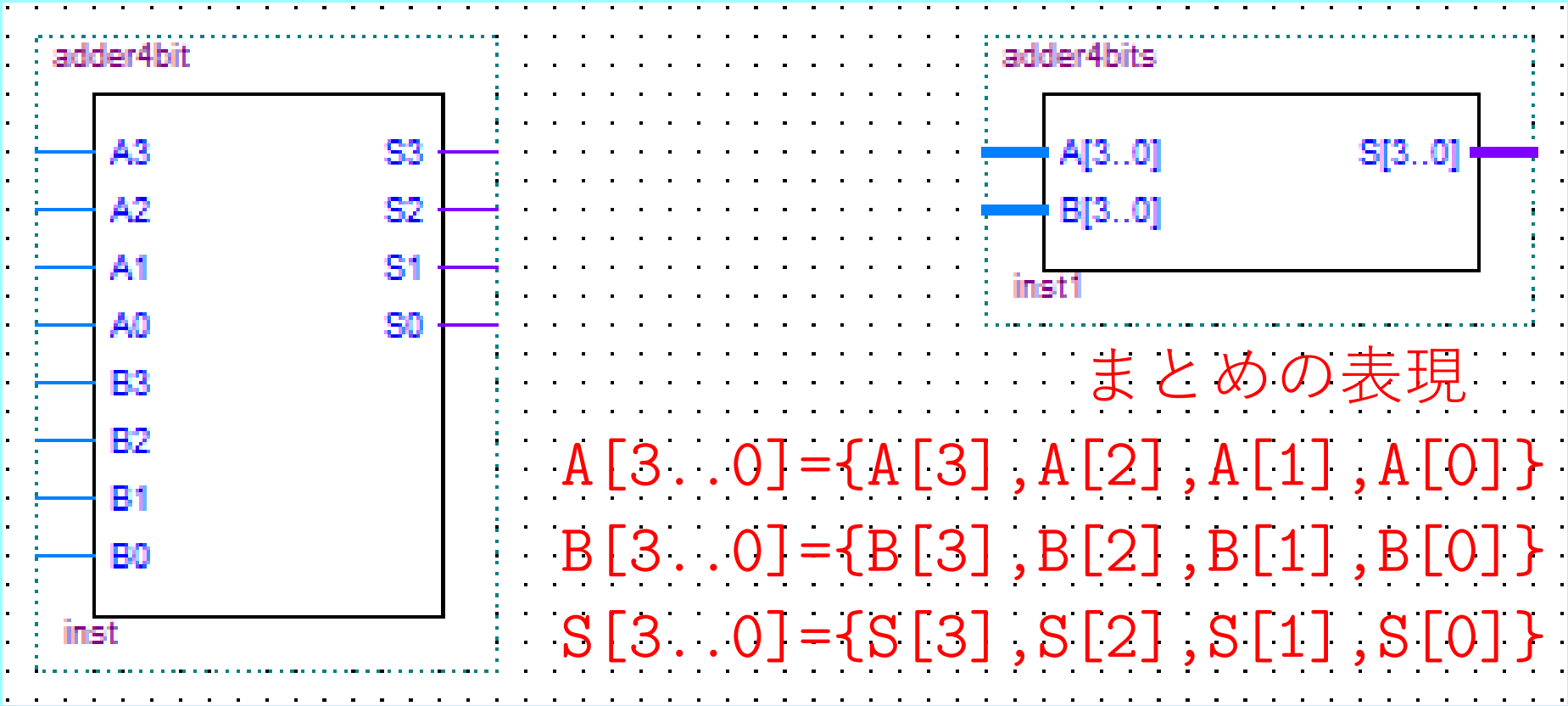




マルチビット

４ビットの加算（入力信号８本、出力信号４本）

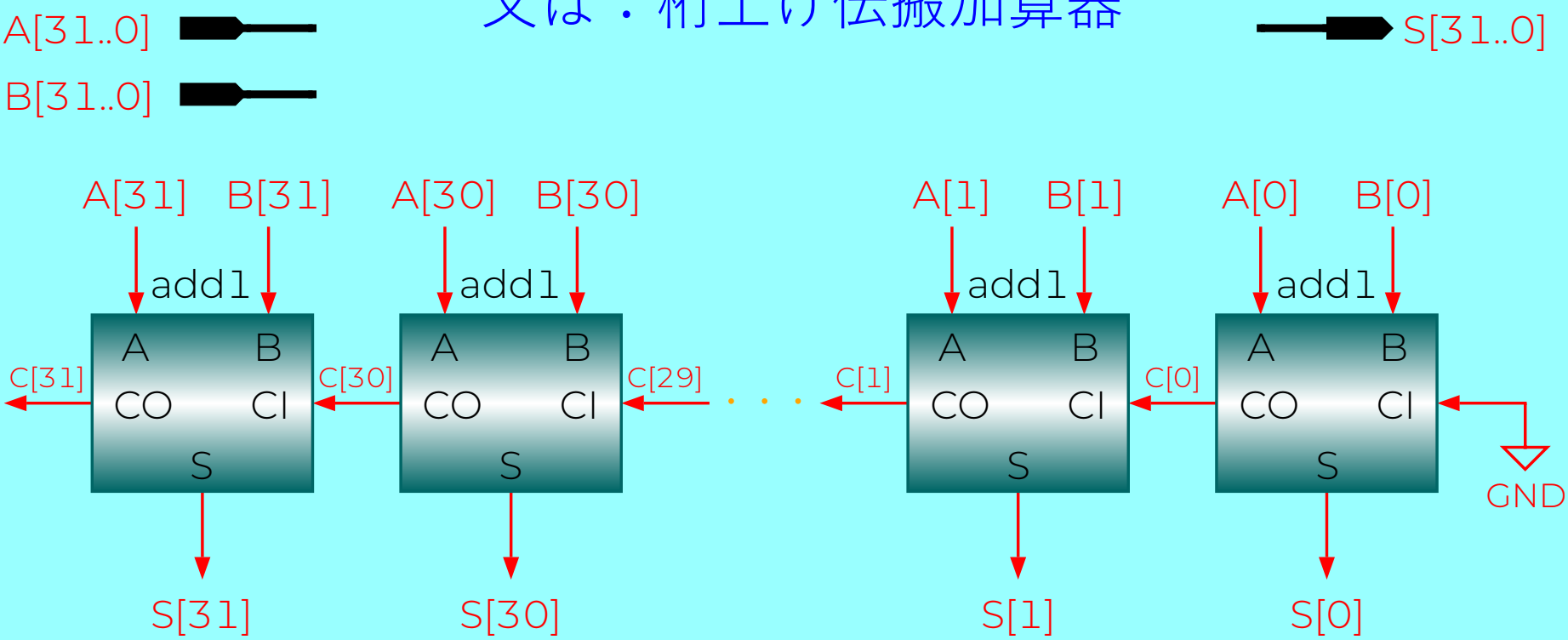
二つの回路のシンボル（右はまとめの表現）：



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)

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

$$C_0 = G_0 + P_0 \cdot C_{-1}$$
$$C_1 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_{-1}$$
$$C_2 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_{-1}$$
$$C_3 = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_{-1}$$

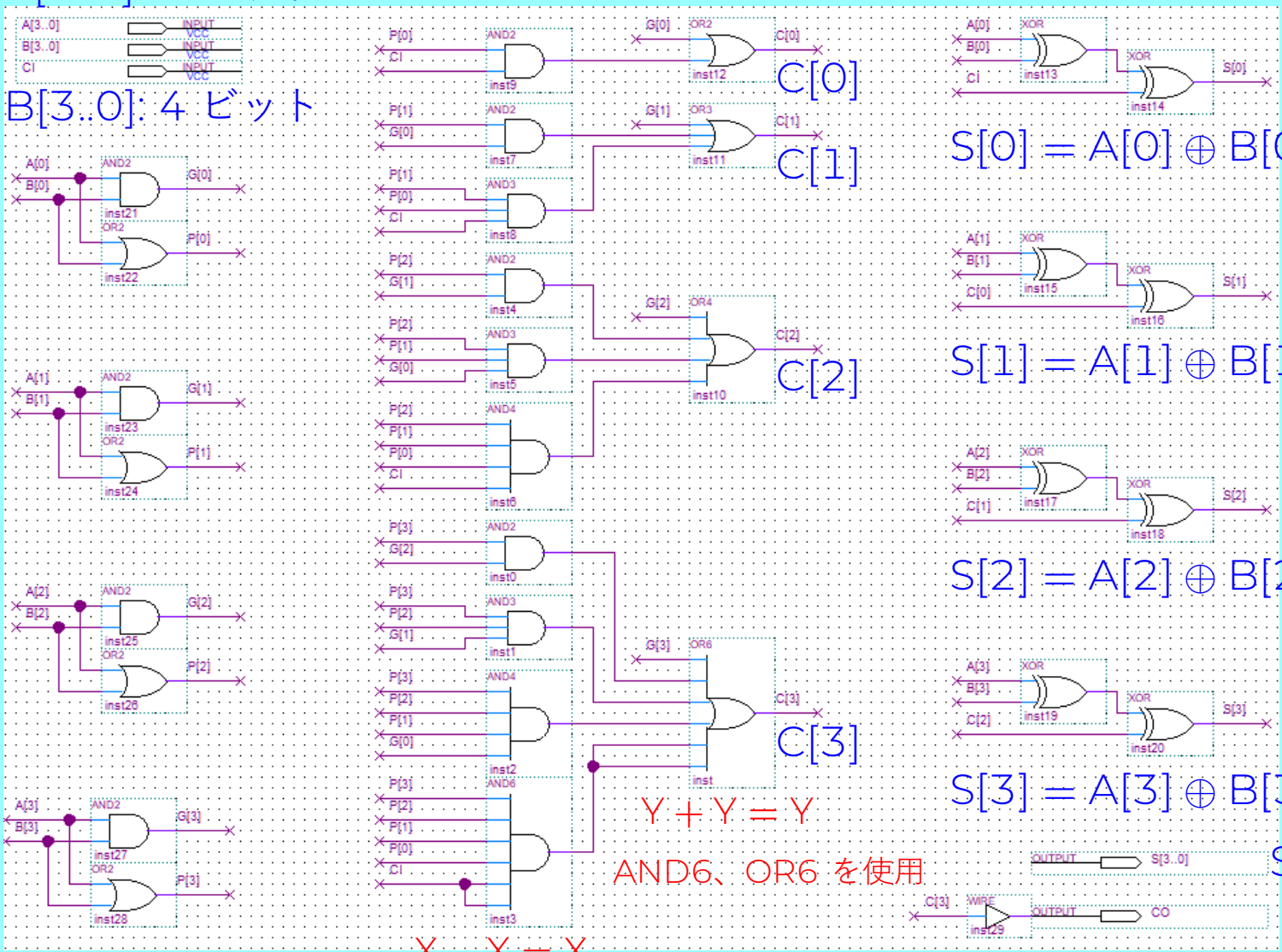
(C_{-1}: Carry in CI)

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。

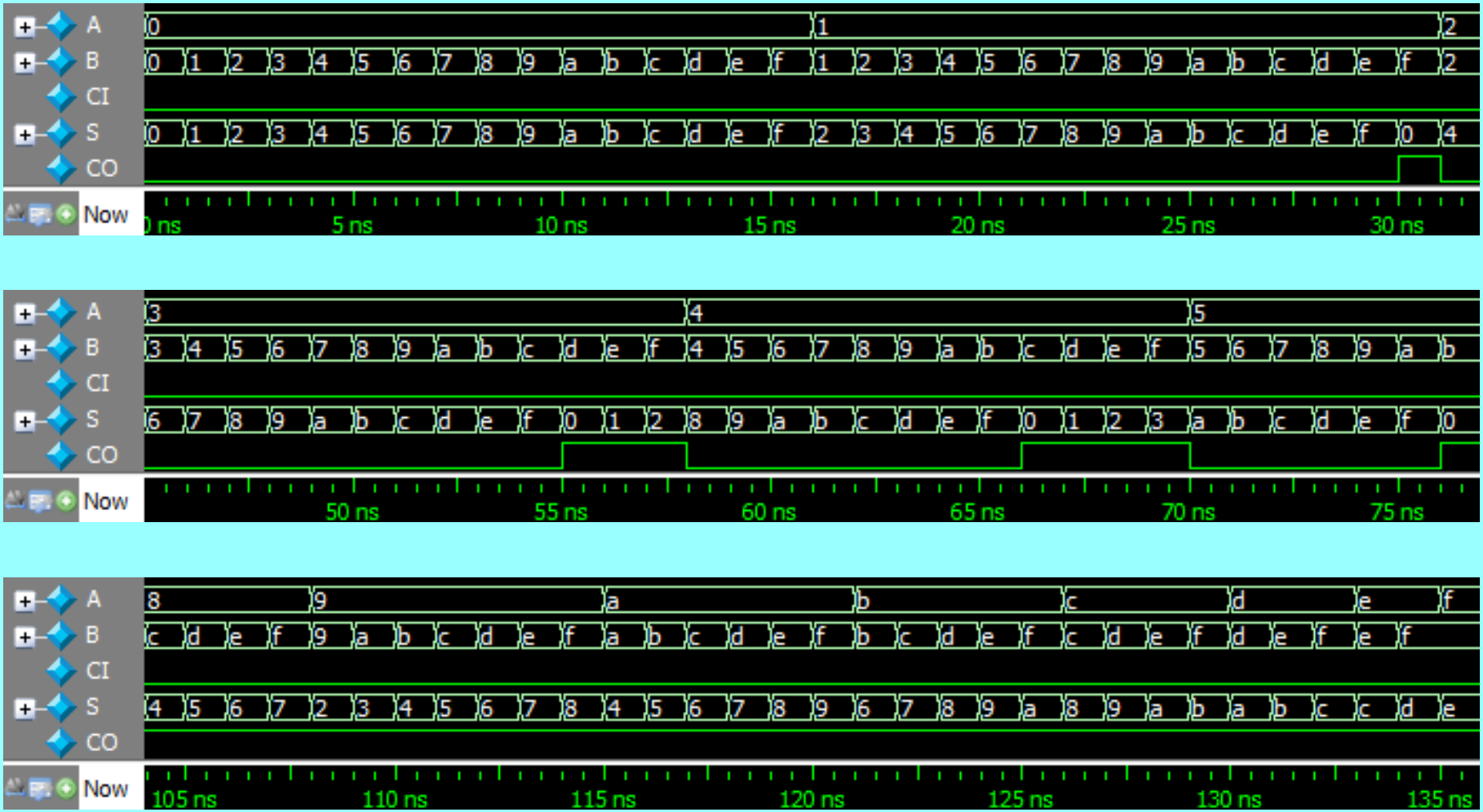
桁上げ先見加算器 (CLA4) の回路

A[3..0]: 4 ビット

B[3..0]: 4 ビット

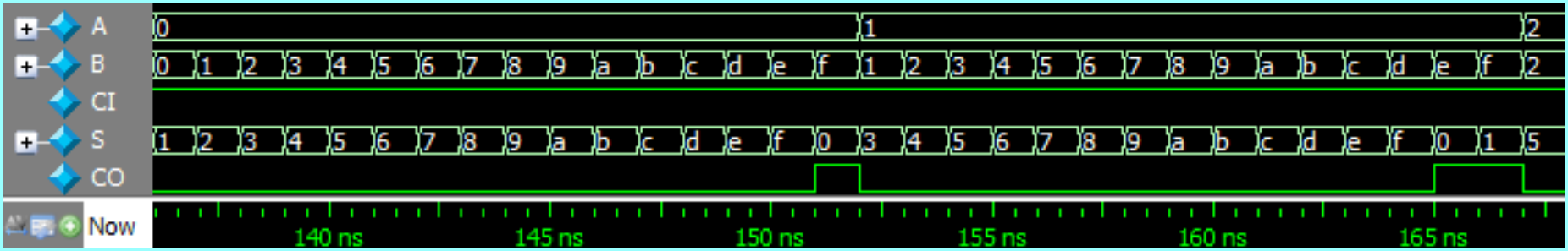


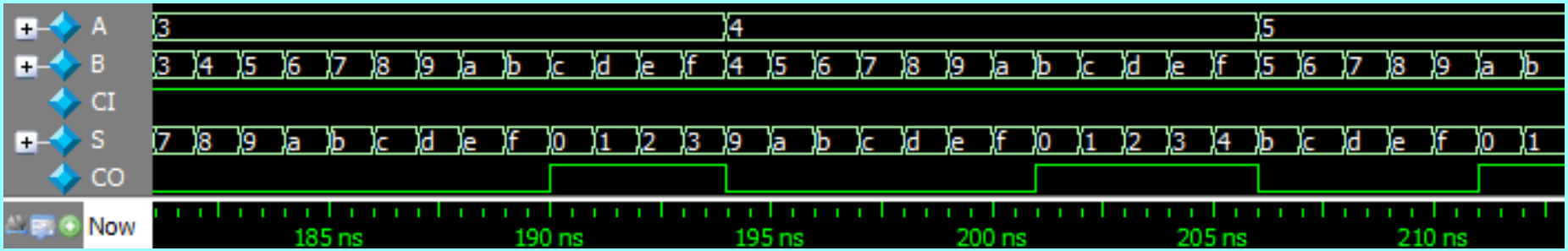
桁上げ先見加算器 (CLA4) の波形

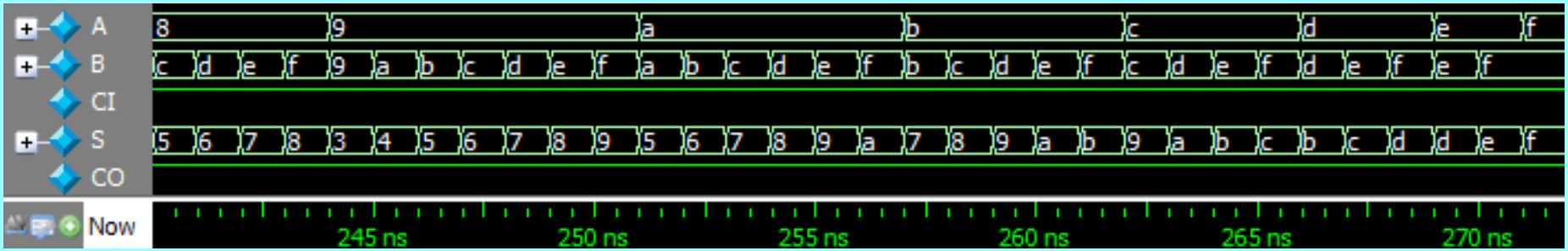


Radix: Hexadecimal (16 進)

桁上げ先見加算器 (CLA4) の波形

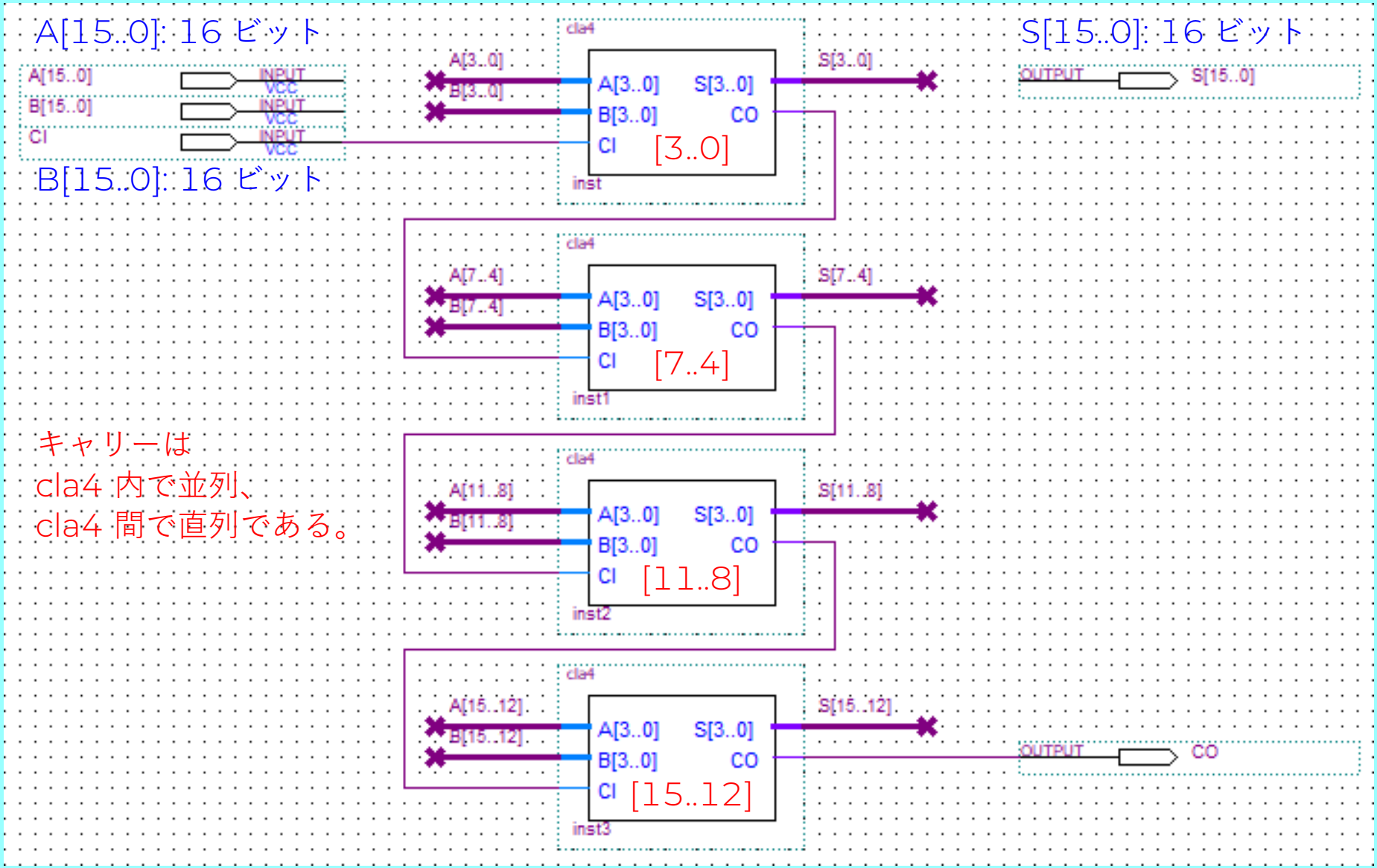






Radix: Hexadecimal (16 進)

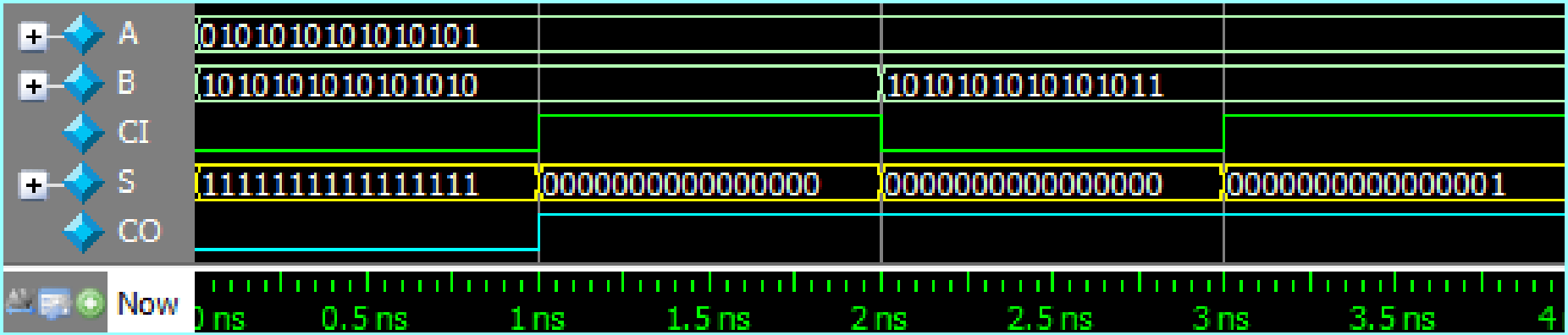
CLA4 を使用する 16 ビット加算器



CLA4 を使用する 16 ビット加算器

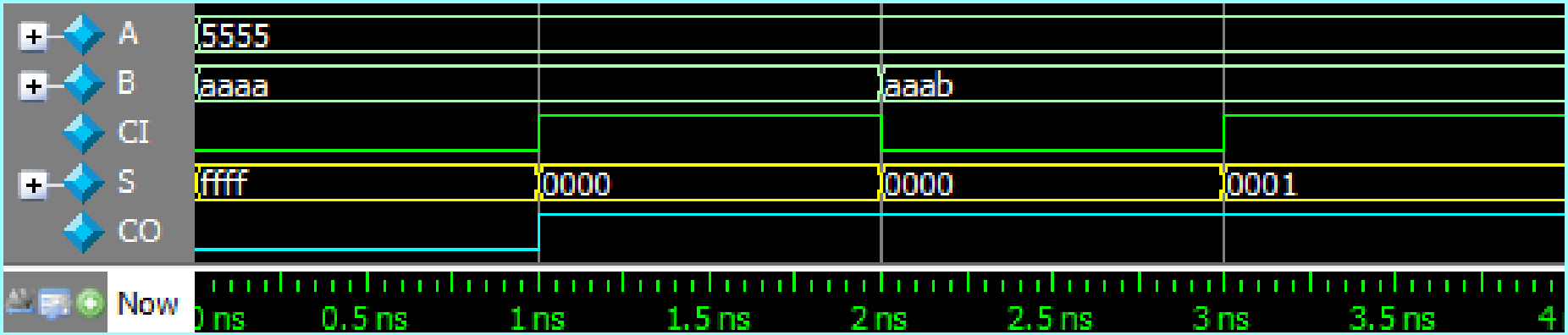
```
‘timescale 1ns/1ps // unit = 1 ns; accuracy = 1 ps
module add16cla4_tb;
    reg  [15:0] A; // 16 bits
    reg  [15:0] B; // 16 bits
    reg         CI;
    wire [15:0] S; // 16 bits
    wire        CO;
    add16cla4 i1 (.A(A), .B(B), .CI(CI), .S(S), .CO(CO));
    initial begin
        #0 A  = 16’h5555; B  = 16’haaaa; CI = 0;
        #1 CI = 1;
        #1 B  = 16’haaab; CI = 0;
        #1 CI = 1;
        #1 $stop;
    end
endmodule
```

CLA4 を使用する 16 ビット加算器



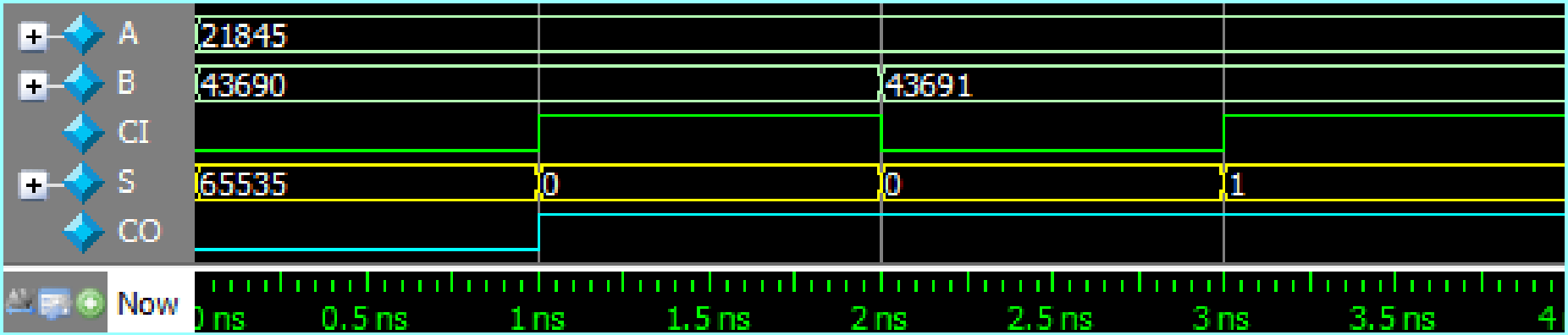
Radix: Binary (2 進) ・・・・・・・・・・・・A, B, S: それぞれ 16 ビット

同じ波形、基数 (Radix) のみが異なる



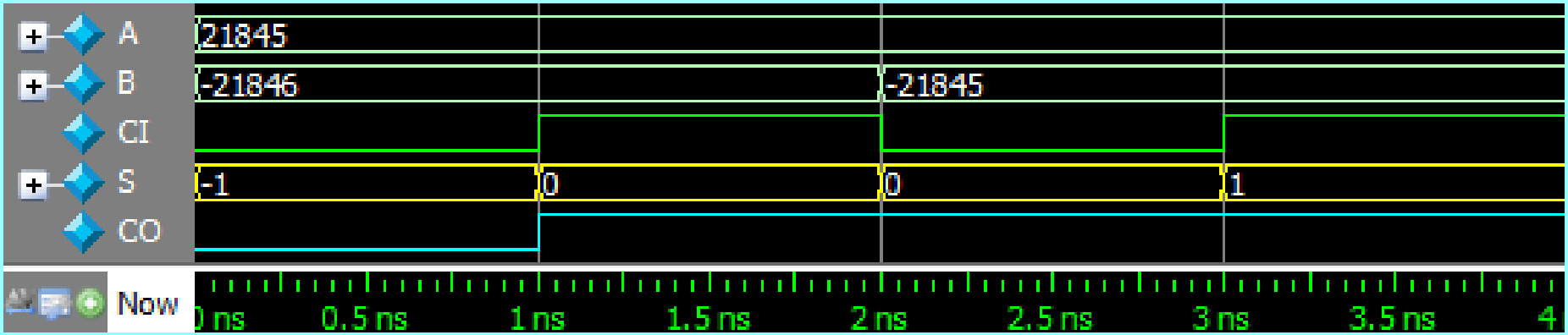
Radix: Hexadecimal (16 進) ・・・・A, B, S: それぞれ 16 ビット

CLA4 を使用する 16 ビット加算器



Radix: Unsigned (10 進絶対値) ‥A, B, S: それぞれ 16 ビット

同じ波形、基数 (Radix) のみが異なる



Radix: Decimal (10 進整数) ・・・・・A, B, S: それぞれ 16 ビット

Radix の例

emacs@localhost

File Edit Options Buffers Tools C Help

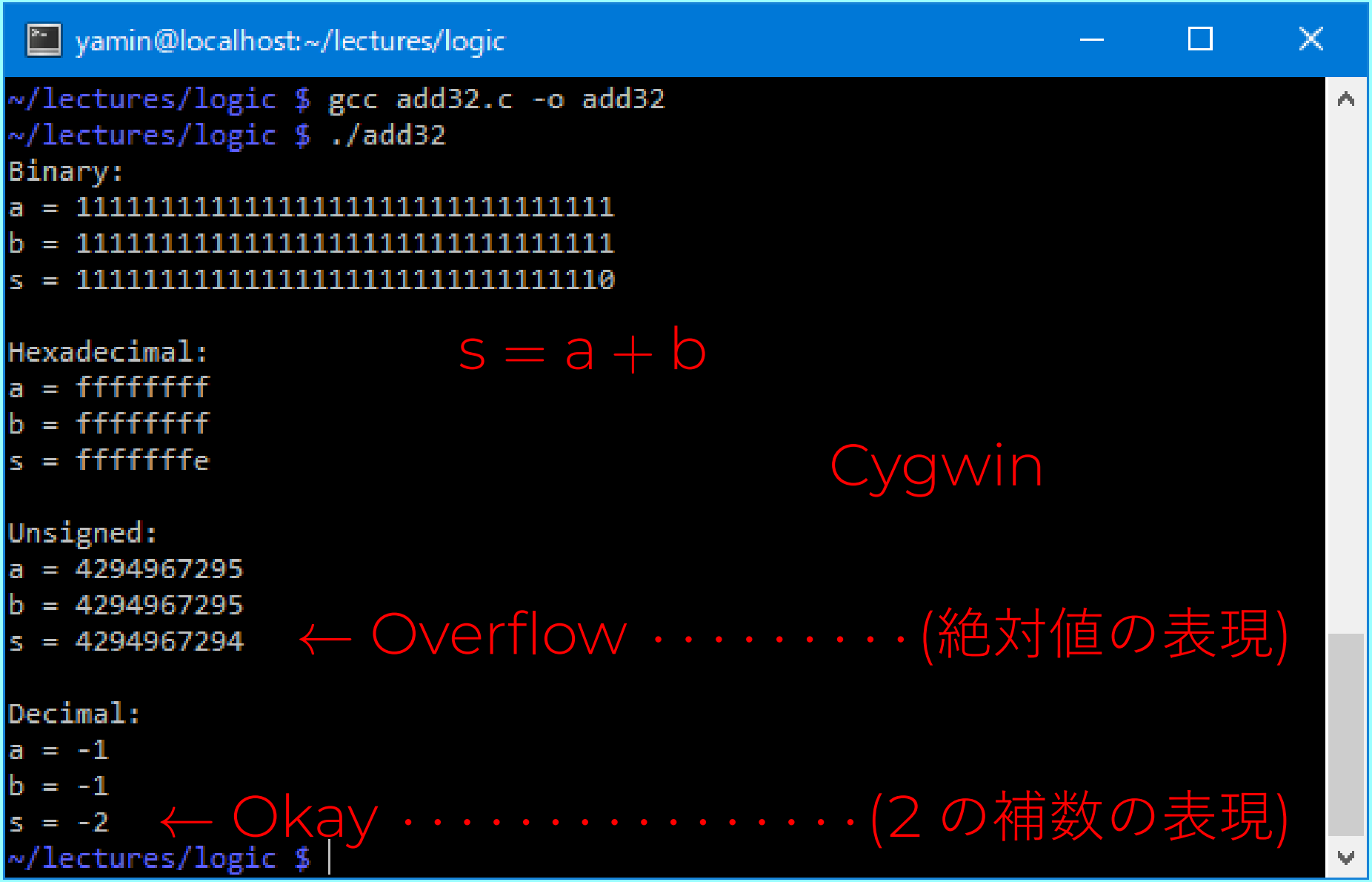
```c
#include<stdio.h>
void print_binary(unsigned int number) {
    int i;
    unsigned int one_bit;
    for (i = 31; i >= 0; i--) {
        one_bit = (number >> i) & 1;
        putc((one_bit == 0) ? '0' : '1', stdout);
    }
}
int main() {
    unsigned int a = 0xffffffff;
    unsigned int b = 0xffffffff;
    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;
}
```

U:--- **add32.c** All L31 (C/l Abbrev)

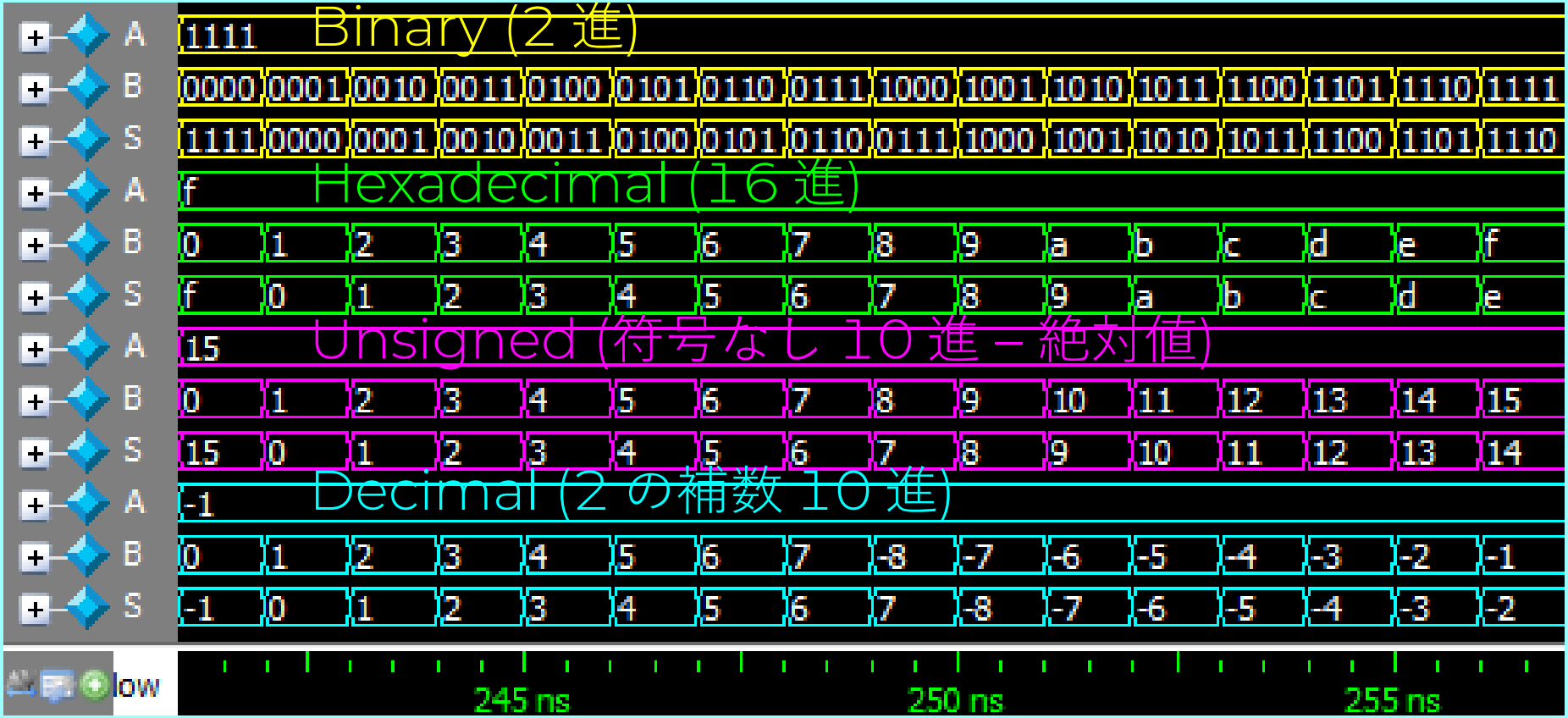
$s = a + b$

add32.c

Radix の例

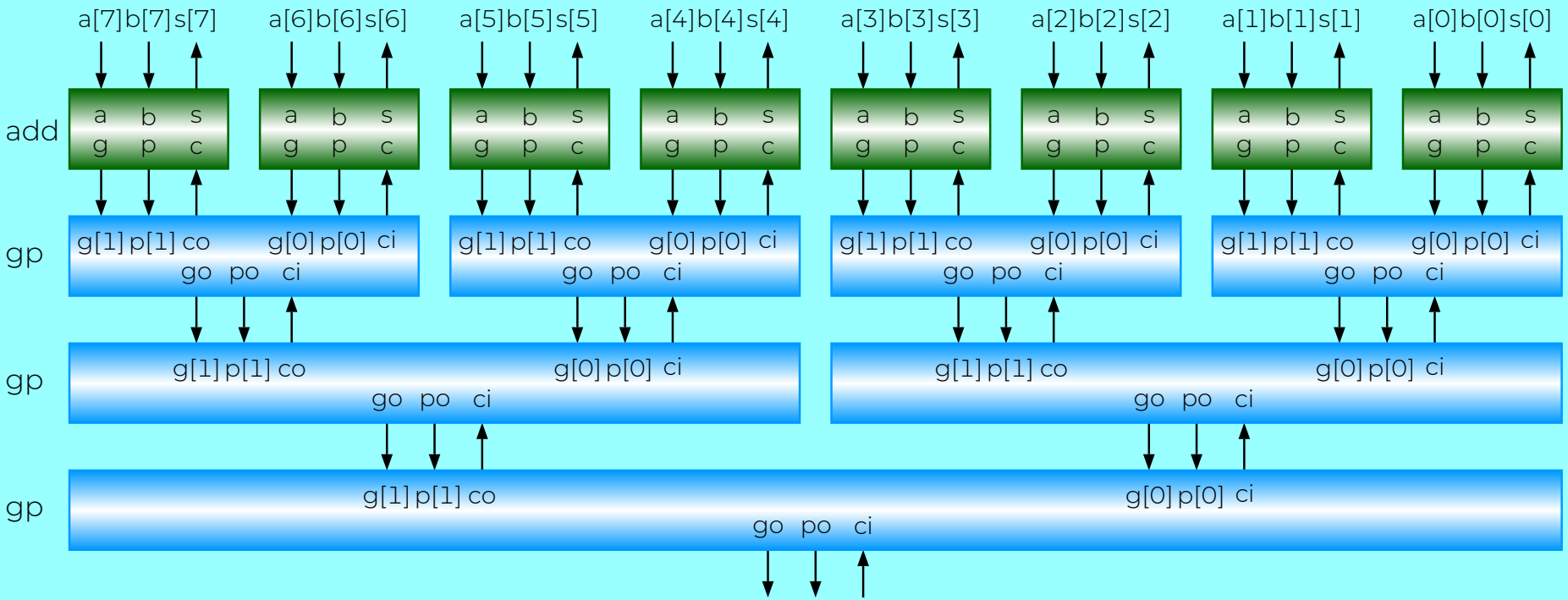

yamin@localhost:~/lectures/logic
~/lectures/logic $ gcc add32.c -o add32
~/lectures/logic $ ./add32
Binary:
a = 11111111111111111111111111111111
b = 11111111111111111111111111111111
s = 11111111111111111111111111111110
Hexadecimal:
a = ffffffff
b = ffffffff
s = fffffffe
Unsigned:
a = 4294967295
b = 4294967295
s = 4294967294
Decimal:
a = -1
b = -1
s = -2
~/lectures/logic $
s = a + b
Cygwin
← Overflow ・・・・・・・・・(絶対値の表現)
← Okay ・・・・・・・・・・・・・・・(2 の補数の表現)

Radix の例 (S = A + B)



4 つの Radix 表示：1. Binary (2 進)
2. Hexadecimal (16 進)
A = 1111 3. Unsigned (符号なし 10 進 – 絶対値)
B = 0000 ～ 1111 4. Decimal (2 の補数 10 進)

ツリー型桁上げ先見加算器の回路



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

$g = a \cdot b$

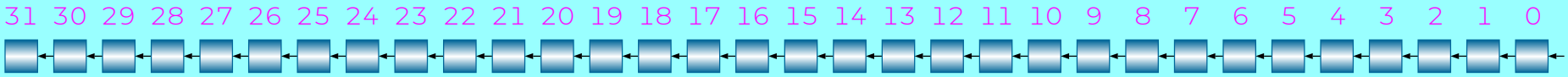
$p = a + b$

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

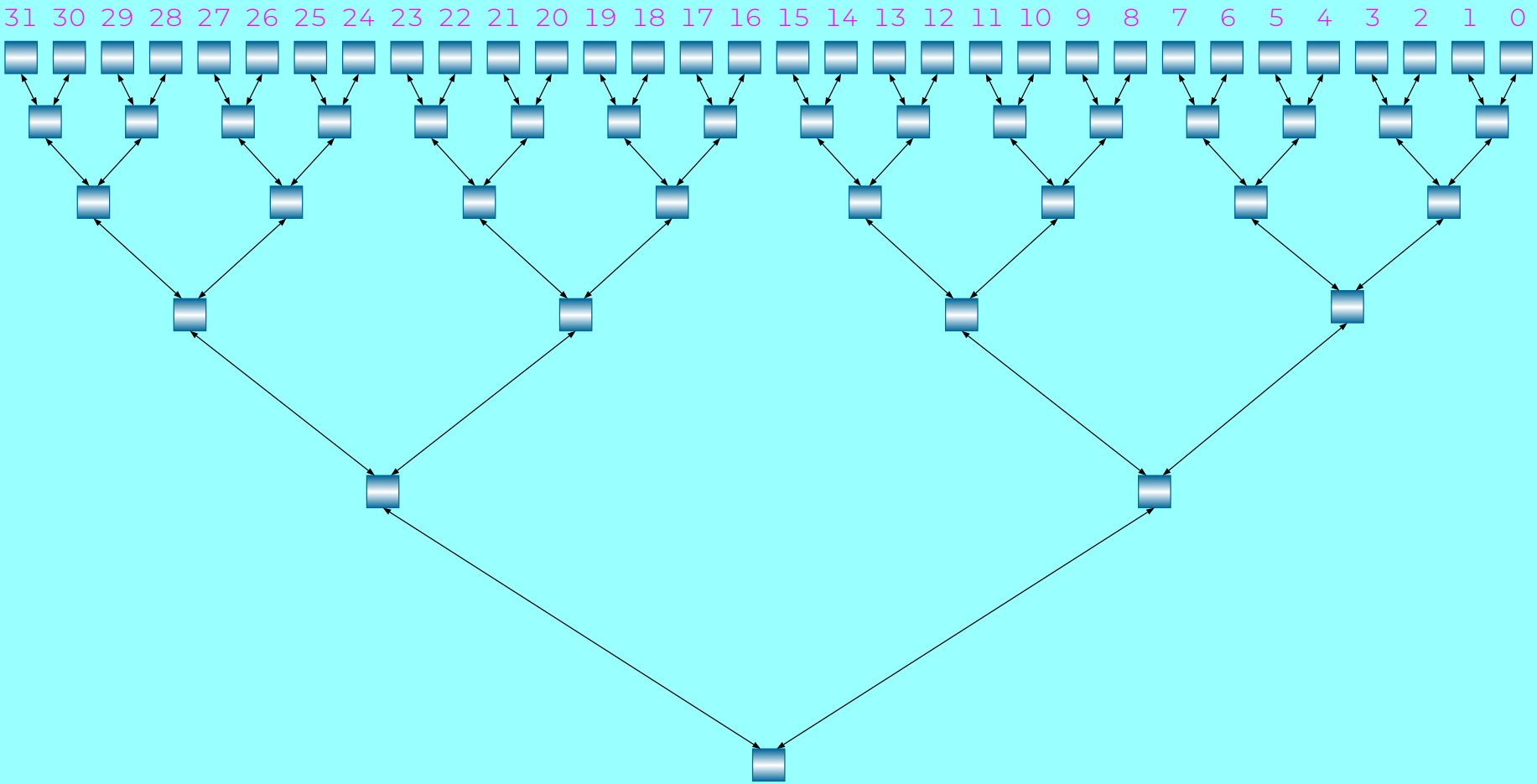
$go = g[1] + p[1] \cdot g[0]$

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

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 $O(\log_2 n)$

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)

14 ページを参照し、全加算器 add1 を使って、 4 ビットリップルキャリーアダー add4 を設計し動作検証シミュレーションして下さい。

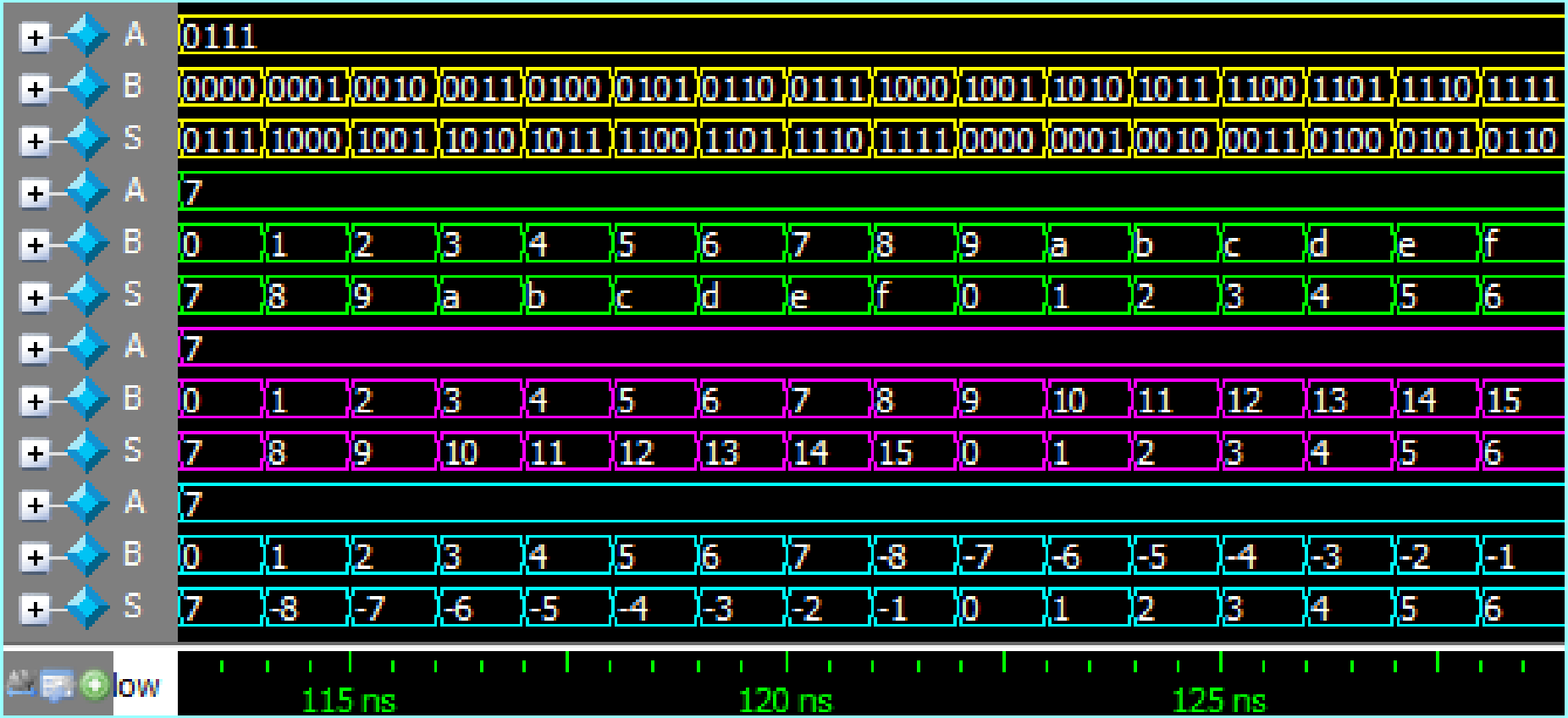
入力信号: A[3..0] . . . . . 4-bit data
入力信号: B[3..0] . . . . . 4-bit data
出力信号: S[3..0] . . . . . 4-bit result

プロジェクト名は add4 にすること。
テストベンチ add4_tb.v を使って下さい。

課題 VI (100 点 + 100 点)

4 つの Radix 表示：1. Binary (2 進)
2. Hexadecimal (16 進)
A = 0001 3. Unsigned (符号なし 10 進 – 絶対値)
B = 0000 ～ 1111 4. Decimal (2 の補数 10 進)

課題 VI (100 点 + 100 点)



4 つの Radix 表示：1. Binary (2 進)
2. Hexadecimal (16 進)
A = 0111 3. Unsigned (符号なし 10 進 – 絶対値)
B = 0000 ～ 1111 4. Decimal (2 の補数 10 進)

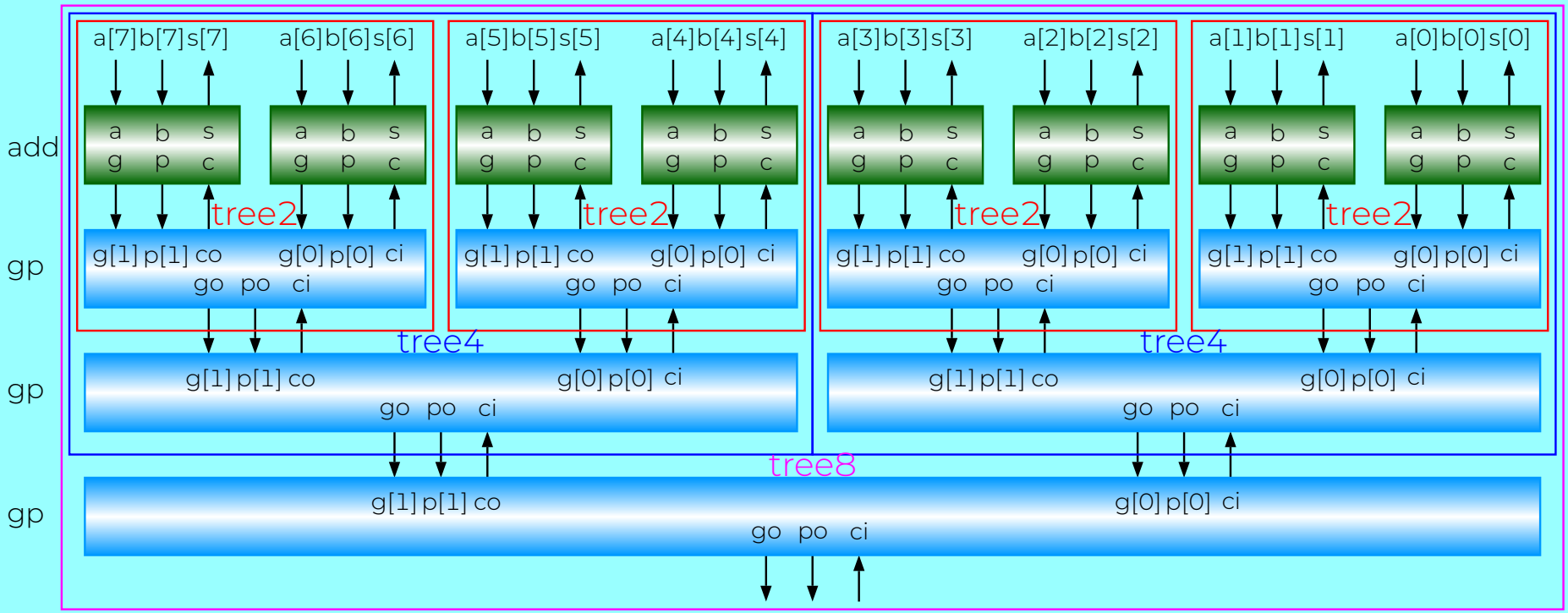
課題 VI (100 点 + 100 点)

4 つの Radix 表示：1. Binary (2 進)
2. Hexadecimal (16 進)
A = 1111 3. Unsigned (符号なし 10 進 – 絶対値)
B = 0000 ～ 1111 4. Decimal (2 の補数 10 進)

課題 VI (100 点 + 100 点)

オプション (+100 点) : 30 ページを参照し、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。テストベンチを自分で作って下さい。

課題 VI (100 点 + 100 点)



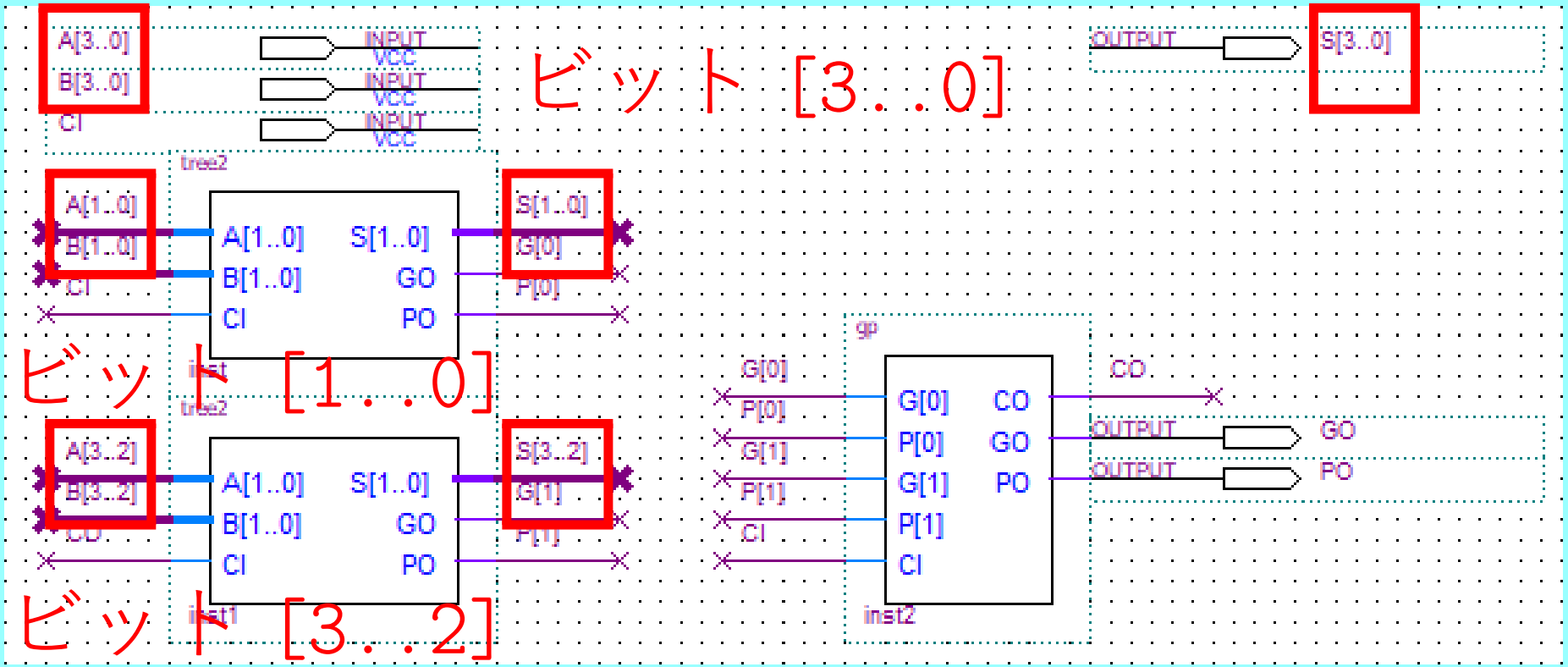
この回路は階層的に設計できる。設計順番:

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)

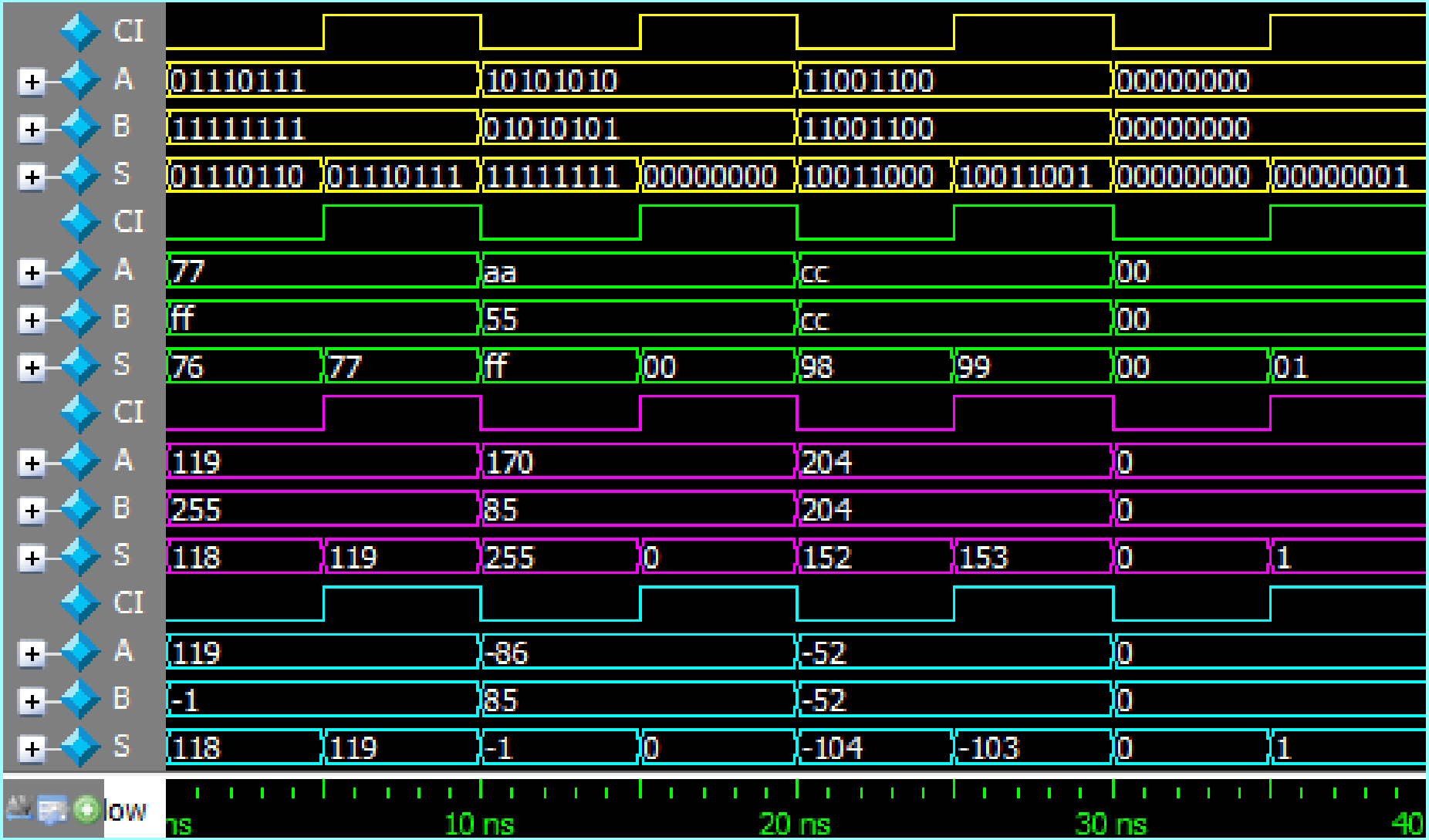
課題 VI (100 点 + 100 点)



tree4 (4 ビット加算器、二つの tree2 と一つの gp を使用) の例

マルチビット番号に注意してください

課題 VI (100 点 + 100 点)



tree8 のシミュレーションの例

発展：自由練習

Verilog HDL による課題の実装

全加算器の論理式

$S = A \oplus B \oplus CI$

$CO = A \cdot B + B \cdot CI + CI \cdot A$

```
module add1b (A, B, CI, CO, S);
    input  A, B, CI;
    output CO, S;
    assign S  = A ^ B ^ CI;
    assign CO =           ;
endmodule
```

上記コードを完成しシミュレーションして下さい。

発展：自由練習

Verilog HDL による課題の実装

4 ビットリップルキャリーアダーの設計（P14 を参照）

```
module add4b (A, B, CI, CO, S);
    input  [3:0] A, B;
    input        CI;
    output [3:0] S;
    output       CO;
    wire   [2:0] c; // internal wires
    add1b i0 (A[0], B[0],   CI, c[0], S[0]); // full adder 0
    add1b i1 (A[1], B[1],     ,    , S[1]); // full adder 1
    add1b i2 (A[2], B[2],     ,    , S[2]); // full adder 2
    add1b i3 (A[3], B[3],     ,   CO, S[3]); // full adder 3
endmodule
```

上記コードを完成しシミュレーションして下さい。

発展：自由練習

Verilog HDL による課題の実装

１ビットツリー型桁上げ先見加算器の回路（P30 を参照）

```
module tree_1 (a, b, c, g, p, s);
    input  a, b, c;
    output g, p, s;
    assign s = a ^ b ^ c;
    assign g =      ;
    assign p =      ;
endmodule
```

上記コードを完成しシミュレーションして下さい。

発展：自由練習

Verilog HDL による課題の実装

ツリー型桁上げ先見加算器の GP の回路（P30 を参照）

```
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[ ] | p[ ] & g[ ];
    assign      p_out = p[ ] & p[ ];
endmodule
```

上記コードを完成しシミュレーションして下さい。

発展：自由練習

Verilog HDL による課題の実装

2 ビットツリー型桁上げ先見加算器の回路（P30 を参照）

```
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;
    tree_1 a0 (a[ ], b[ ], c_in,  g[ ], p[ ], s[ ]);
    tree_1 a1 (a[ ], b[ ], c_out, g[ ], p[ ], s[ ]);
    g_p    gp (g, p, c_in, g_out, p_out, c_out);
endmodule
```

上記コードを完成しシミュレーションして下さい。

発展：自由練習

Verilog HDL による課題の実装

4 ビットツリー型桁上げ先見加算器の回路（P30 を参照）

```
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[   ], b[   ], c_in,  g[ ], p[ ], s[   ]);
    tree_2 a1 (a[   ], b[   ], c_out, g[ ], p[ ], s[   ]);
    g_p    gp (g, p, c_in, g_out, p_out, c_out);
endmodule
```

上記コードを完成しシミュレーションして下さい。

発展：自由練習

Verilog HDL による課題の実装

8 ビットツリー型桁上げ先見加算器の回路（P30 を参照）

```
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[ ], p[ ], s[   ]);
    tree_4 a1 (a[   ], b[   ], c_out, g[ ], p[ ], s[   ]);
    g_p    gp (g, p, c_in, g_out, p_out, c_out);
endmodule
```

上記コードを完成しシミュレーションして下さい。