論理回路入門（７）

組合せ回路２：２進数の減算と加減算器

李　亜民

2024 年 11 月 5 日 (火)

2 進数の減算

ポイント

- 符号なし数
- 負の整数
- 2 の補数で表現する方法
- X から −X を求める
- 2 進数の減算
- 加算器を利用した減算
- 加減算器
- オーバーフロー (符号なしの場合、2 の補数の場合)

負の整数

- C 言語プログラムの中に

```
int i = -1;
```

- 質問: この -1 は、32 ビットの 2 進数でどんなビットパターンになるか？
- ヒント: $(-1)+(+1)=0$、つまり

$$
\begin{array}{rll}
 & ????????????????????????????????? & (-1) \\
+ & 00000000000000000000000000000001 & (+1) \\
\hline
 & 00000000000000000000000000000000 & (\ 0)
\end{array}
$$

負の整数

- 答え:

$$-1_{10} = 11111111111111111111111111111111_2$$

- つまり

$$\begin{array}{rl} & 11111111111111111111111111111111 \quad (-1) \\ + & 00000000000000000000000000000001 \quad (+1) \\ \hline & 00000000000000000000000000000000 \quad (\ 0) \end{array}$$

- ここは 2 の補数で表現する方法を使用している。

2 の補数で表現する方法

- n ビット 2 の補数で表される符号付き数の大きさは

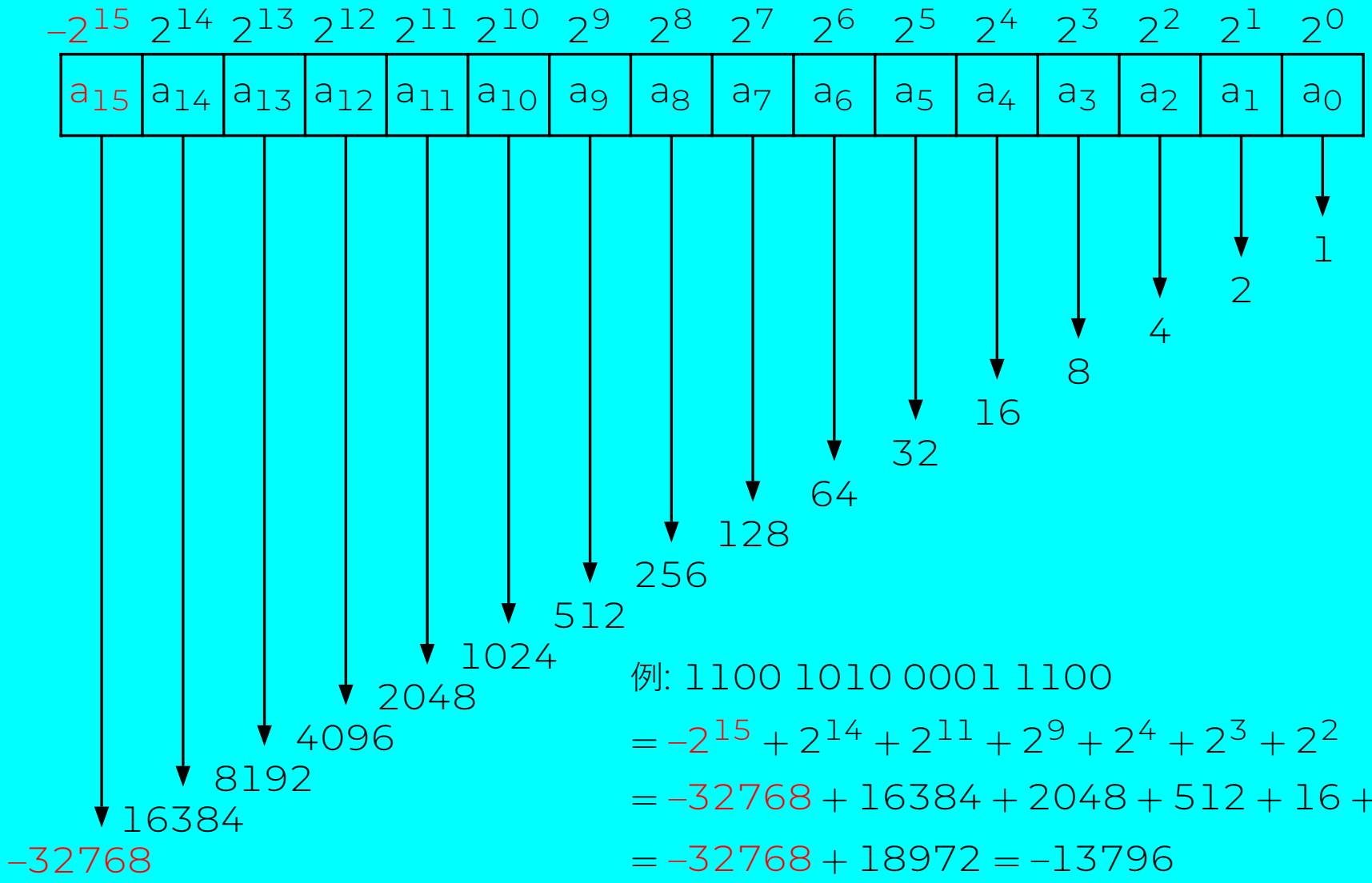
  $a_{n-1}a_{n-2}\cdots a_1a_0 = -a_{n-1} \times 2^{n-1} + a_{n-2} \times 2^{n-2} + \cdots + a_1 \times 2^1 + a_0 \times 2^0$

- 32 ビット 2 の補数で表される符号付き数:

  0000 0000 0000 0000 0000 0000 0000 0000 = 0_{10}
  0000 0000 0000 0000 0000 0000 0000 0001 = $+1_{10}$
  ... ...
  0111 1111 1111 1111 1111 1111 1111 1111 = $+2{,}147{,}483{,}647_{10}$
  1000 0000 0000 0000 0000 0000 0000 0000 = $-2{,}147{,}483{,}648_{10}$
  ... ...
  1111 1111 1111 1111 1111 1111 1111 1111 = -1_{10}

- n ビット 2 の補数で表される符号付き数のうち最小値は -2^{n-1}、最大値は $+(2^{n-1}-1)$ である。
  例：n = 4 のとき、最小値は −8、最大値は +7 である。

2 の補数で表現する方法



例: 1100 1010 0001 1100

$= -2^{15} + 2^{14} + 2^{11} + 2^9 + 2^4 + 2^3 + 2^2$

$= -32768 + 16384 + 2048 + 512 + 16 + 8 + 4$

$= -32768 + 18972 = -13796$

表現範囲

ビット数	10 進数の表現範囲 (覚えてください)					
	符号なし数			2 の補数表現		
1	0	~	1	−1	~	0
2	0	~	3	−2	~	+1
3	0	~	7	−4	~	+3
4	0	~	15	−8	~	+7
5	0	~	31	−16	~	+15
6	0	~	63	−32	~	+31
7	0	~	127	−64	~	+63
8	0	~	255	−128	~	+127
9	0	~	511	−256	~	+255
10	0	~	1023	−512	~	+511
11	0	~	2047	−1024	~	+1023
12	0	~	4095	−2048	~	+2047
13	0	~	8191	−4096	~	+4095
14	0	~	16383	−8192	~	+8191
15	0	~	32767	−16384	~	+16383
16	0	~	65535	−32768	~	+32767

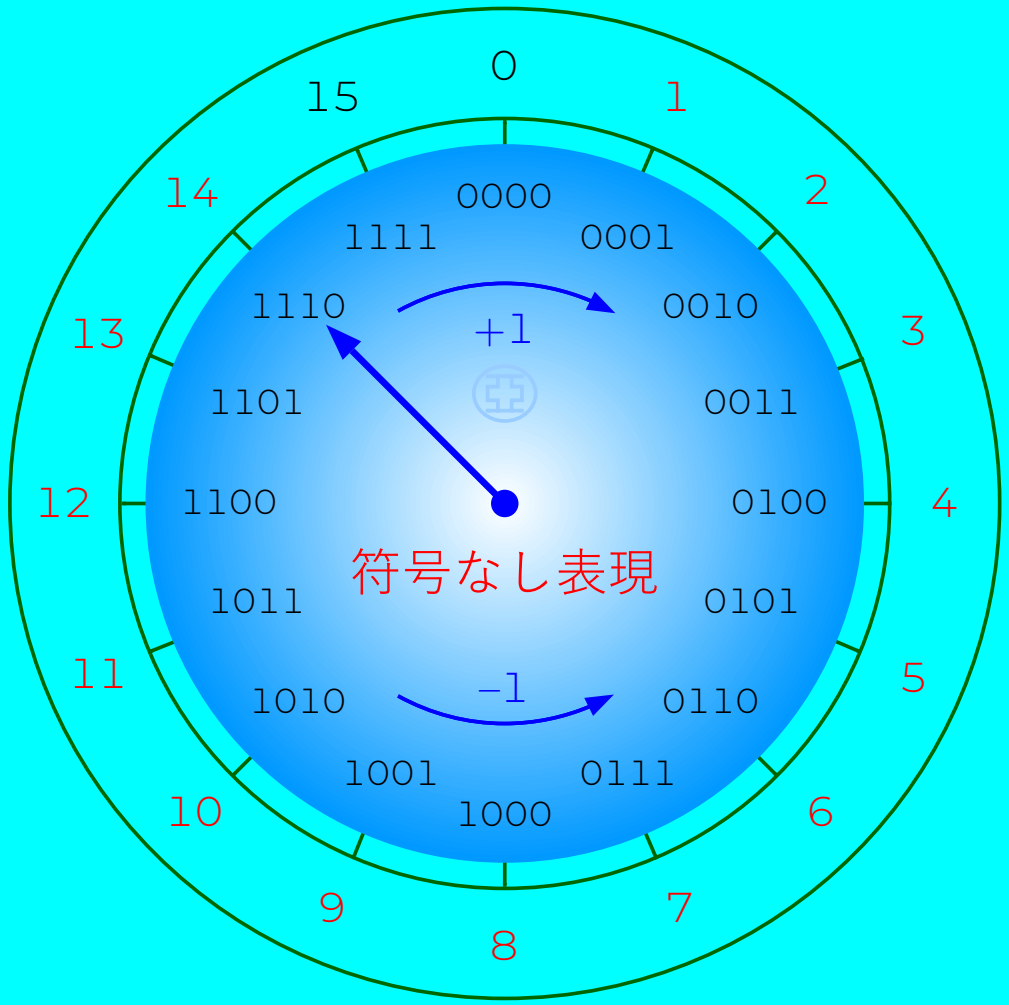
4 ビット 2 進数のまとめ

パターン			意味 (10 進数)	
2 進数	8 進数	16 進数	符号なし数	2 の補数表現
0000	0	0	0	0
0001	1	1	1	+1
0010	2	2	2	+2
0011	3	3	3	+3
0100	4	4	4	+4
0101	5	5	5	+5
0110	6	6	6	+6
0111	7	7	7	+7
1000	10	8	8	−8
1001	11	9	9	−7
1010	12	A	10	−6
1011	13	B	11	−5
1100	14	C	12	−4
1101	15	D	13	−3
1110	16	E	14	−2
1111	17	F	15	−1

+1 (符号なし数)　　+1 (2 の補数表現)

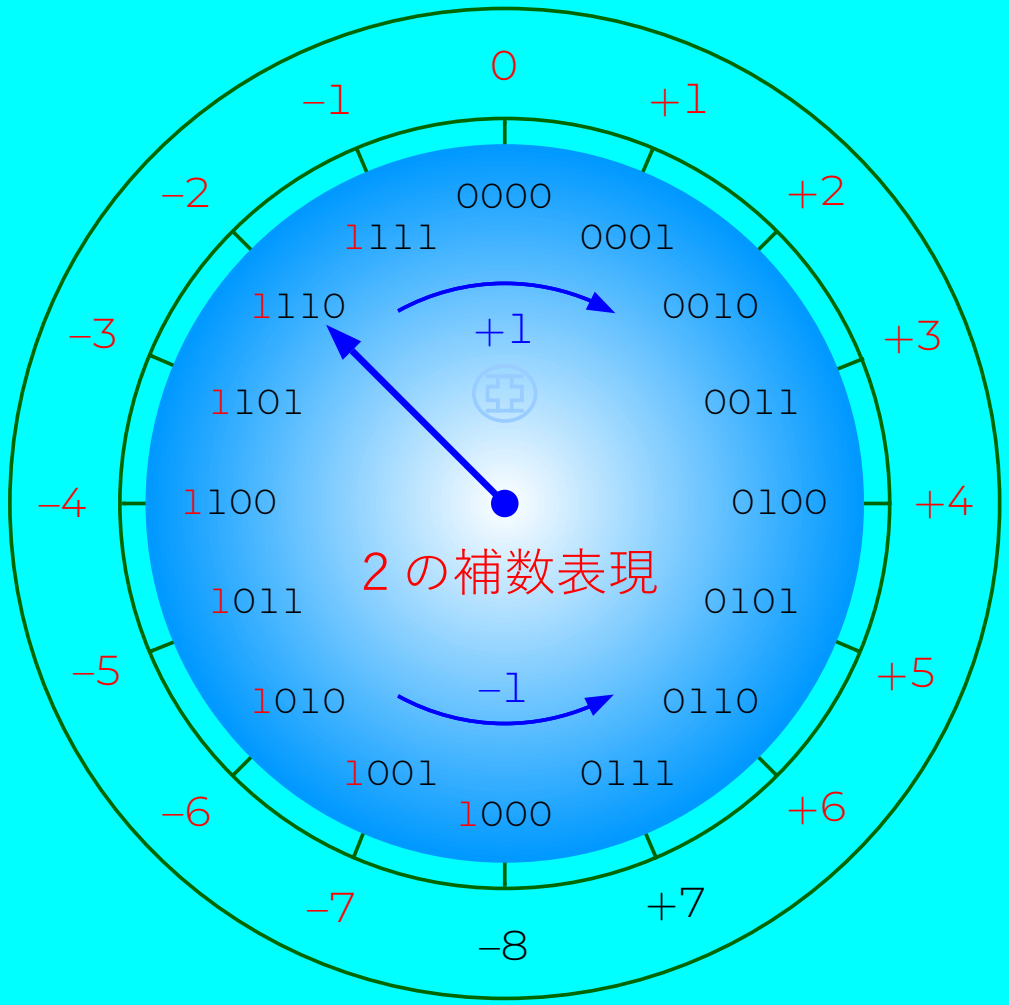
1111 = (−8) + (+7) = −1 (2 の補数表現)

符号なしによる表現（4ビットの例）



表現範囲：0 ～ 15 ・・・・・・・・・・・・・・・・・・・・最大値 15 + 1 = 0 (最小値)

2 の補数による表現（4 ビットの例）



表現範囲： −8 ～ +7 ・・・・・・・・・・・・・・・・最大値 +7 + 1 = −8 (最小値)

X から −X を求める

- X から −X を求める手順
  1. X のビットパターンを反転する
  2. 反転されたビットパターンに 1 を加算する。即ち、

$$-X = \overline{X} + 1$$

- X から −X を求める例 1
  - $X = 0011 = +3_{10}$
  - $-X = \overline{X} + 1 = 1100 + 1 = 1101 = -3_{10}$
- X から −X を求める例 2
  - $X = 1101 = -3_{10}$
  - $-X = \overline{X} + 1 = 0010 + 1 = 0011 = +3_{10}$

2 進数の減算

- 2 の補数では、$-X = \overline{X} + 1$
- 2 の補数を使うメリットは減算が加算器で可能なことである。

$$X - Y = X + (-Y) = X + \overline{Y} + 1$$

- 2 の補数を使うと、$X + (-X) \equiv 0$
  - 理由は
    - ★ $X + \overline{X} = 11 \cdots 11_2 = -1_{10}$、左右に 1 を加算すると
    - ★ $X + \overline{X} + 1 = -1 + 1$、すなわち
    - ★ $X + (-X) = 0$
- 演算の結果が表現範囲を超える場合には、オーバーフロー (Overflow) が発生する。

2 進数の減算

- 2 の補数では、$X - Y = X + (-Y) = X + \overline{Y} + 1$
- 問題: $Z = X - Y = 10001100 - 11010111$ を計算せよ。
- 解答: $-Y = \overline{Y} + 1 = 00101000 + 1 = 00101001$

$X + (-Y)$:

$$\begin{array}{r} 10001100 \\ +\,00101001 \\ \hline 10110101 \end{array}$$

10 進数による確認：

$X = 10001100_2 = -128 + 8 + 4 = -116$

$Y = 11010111_2 = -1 - 32 - 8 = -41$

$Z = 10110101_2 = -1 - 64 - 8 - 2 = -75$

即ち、$Z = X - Y = (-116) - (-41) = -75$

2 進数の減算

- 2 の補数では、$X - Y = X + (-Y) = X + \overline{Y} + 1$
- 問題: $Z = X - Y = 01111100 - 11010111$ を計算せよ。
- 解答: $-Y = \overline{Y} + 1 = 00101000 + 1 = 00101001$

$X + (-Y)$:

$$\begin{array}{r} 01111100 \\ +\,00101001 \\ \hline 10100101 \end{array}$$

10 進数による確認：

$X = 01111100_2 = +127 - 2 - 1 = +124$

$Y = 11010111_2 = -1 - 32 - 8 = -41$

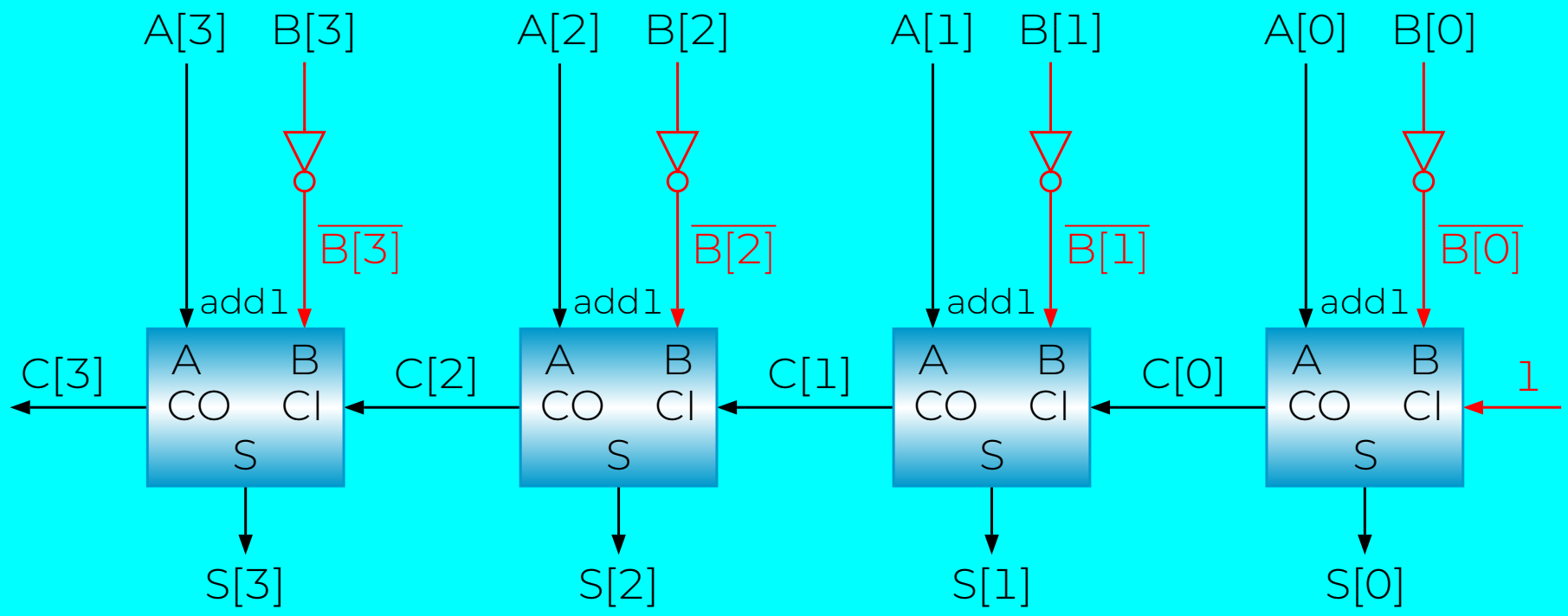
$Z = 10100101_2 = -128 + 32 + 4 + 1 = -91$

$X - Y = 124 - (-41) = 165$ のはず! でも $Z = -91$

オーバーフロー: $-128 \leq$ 8 ビット 2 の補数 $\leq +127$

2 進数減算器

$-X = \overline{X} + 1$ なので、$A - B = A + (-B) = A + \overline{B} + 1$

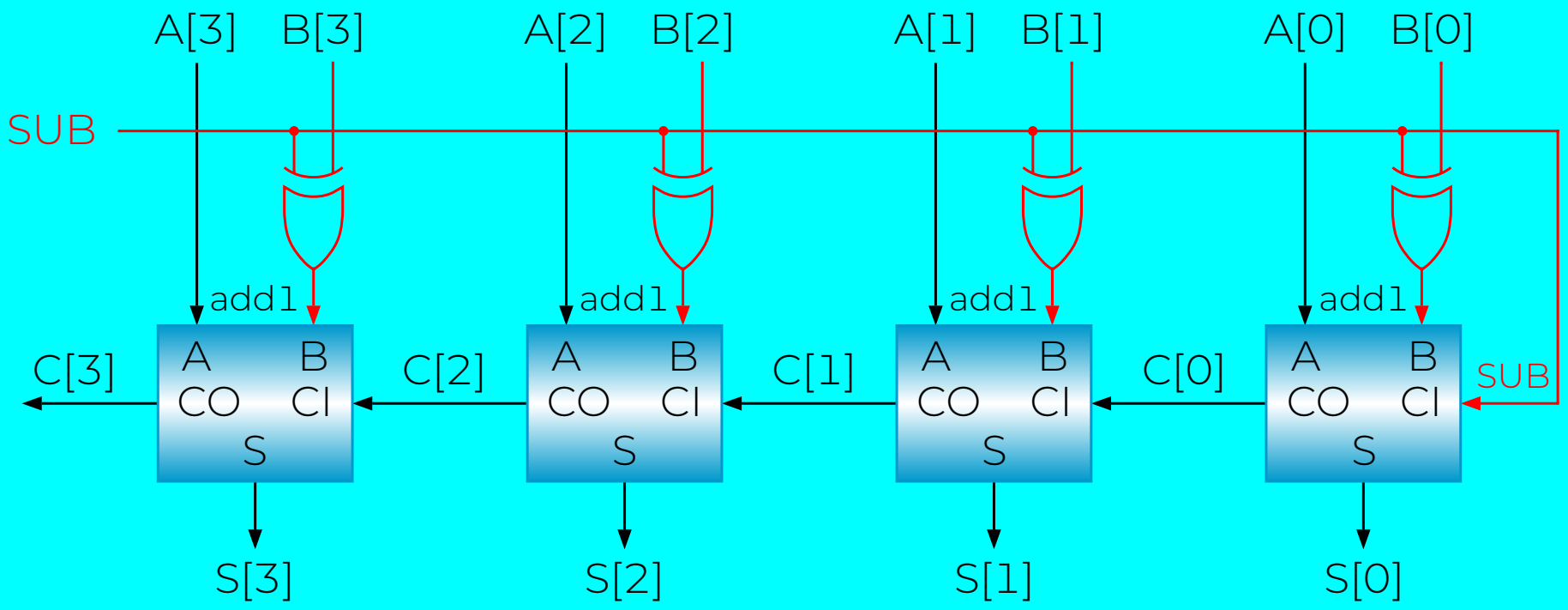


加算器を使用して減算を実現する（$A - B = A + \overline{B} + 1$）

次に、足し算と引き算両方ができる加減算器を実装する

2 進数加減算器

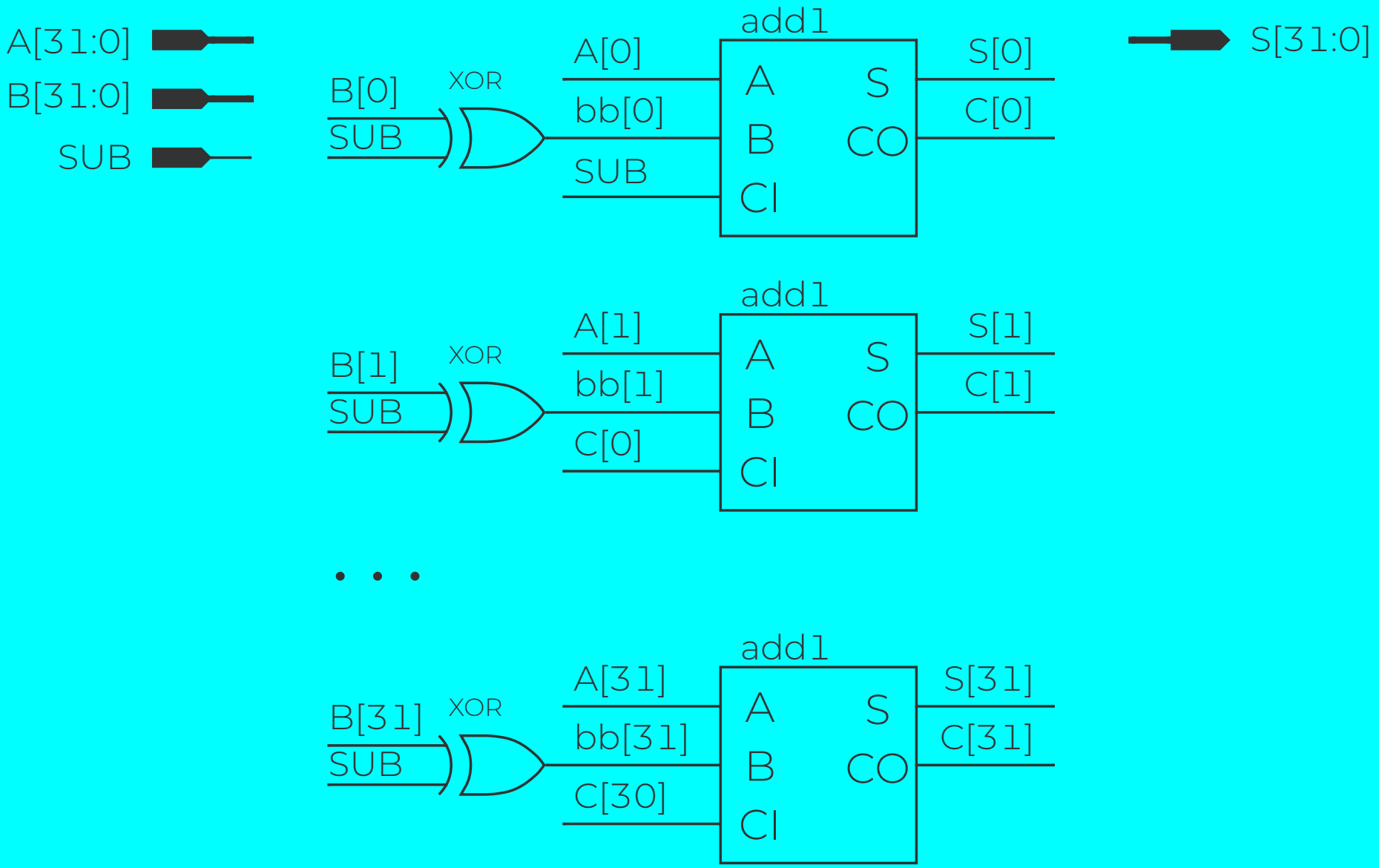
$-X = \overline{X} + 1$ なので、$A - B = A + (-B) = A + \overline{B} + 1$



もし SUB = 0、 $S = A + B + 0 = A + B$ $(B \oplus 0000 = B)$

もし SUB = 1、 $S = A + \overline{B} + 1 = A - B$ $(B \oplus 1111 = \overline{B})$

2 進数加減算器


A[31:0]
B[31:0]
SUB
S[31:0]
add1
A[0]
B[0]
XOR
SUB
bb[0]
SUB
A
B
CI
S
CO
S[0]
C[0]
add1
A[1]
B[1]
XOR
SUB
bb[1]
C[0]
S[1]
C[1]
add1
A[31]
B[31]
XOR
SUB
bb[31]
C[30]
S[31]
C[31]

加減算器addsub4 回路

```
‘timescale 1ns/1ns
module addsub4 (A, B, SUB, S);
    input  [3:0] A, B;
    input        SUB;
    output [3:0] S;

    wire [3:0]   bb = B ^ {4{SUB}}; // XOR
    wire [3:0]   c;

    // add1 (a,    b,     ci,   co,   s);
    add1 i0 (A[0], bb[0], SUB,  c[0], S[0]);
    add1 i1 (    ,      ,     ,     ,     );
    add1 i2 (    ,      ,     ,     ,     );
    add1 i3 (    ,      ,     ,     ,     );
endmodule
```

addsub4.v

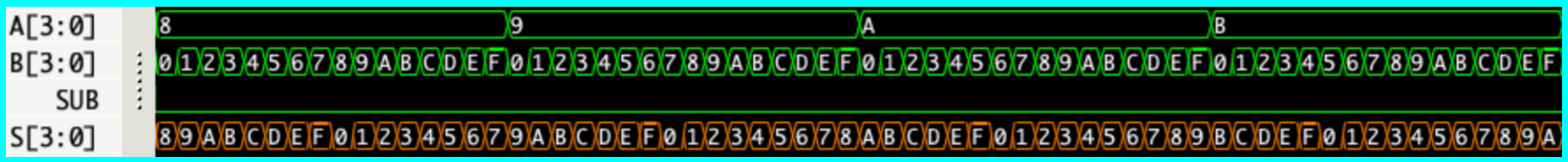
加減算器addsub4 テストベンチ

```
‘timescale 1ns/1ns
module addsub4_tb;
    reg  [3:0] A, B;
    reg        SUB;
    wire [3:0] S;
    addsub4 i0 (A, B, SUB, S);
    integer    i, j;
    initial begin
        #0 SUB = 0; // add
        for (i = 0; i < 16; i = i + 1) begin
            for (j = 0; j < 16; j = j + 1) begin
                A = i; B = j;
                #1 ;
            end
        end
        #0 SUB = 1; // sub
        for (i = 0; i < 16; i = i + 1) begin
            for (j = 0; j < 16; j = j + 1) begin
                A = i; B = j;
                #1 ;
            end
        end
        #1 $finish;
    end
    initial begin
        $dumpfile ("addsub4.vcd");
        $dumpvars;
    end
endmodule
```

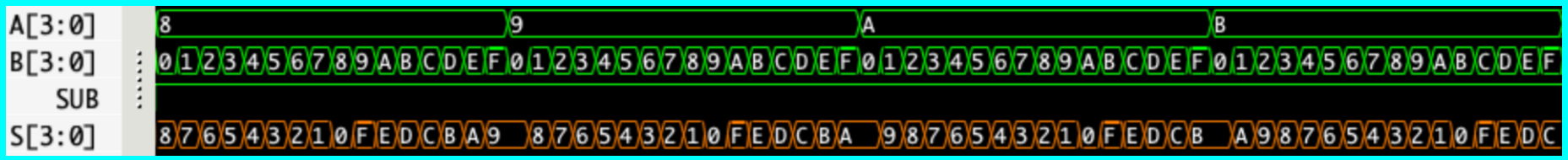
addsub4_tb.v

加減算器addsub4 波形

```
% iverilog -Wall -o addsub4 addsub4_tb.v \
  addsub4.v add1.v half_adder.v
% vvp addsub4
% gtkwave addsub4.vcd
```



S = A + B　　SUB = 0



S = A − B　　SUB = 1

オーバーフローとは

- n ビット場合には、符号無し表現で、表せる範囲が $0 \sim 2^n - 1$ である。
  - 例：n = 4、表せる範囲が 0 ~ 15 である。
- n ビット場合には、2 の補数表現で、表せる範囲が $-2^{n-1} \sim +2^{n-1} - 1$ である。
  - 例：n = 4、表せる範囲が −8 ~ +7 である。
- オーバーフロー (Overflow) は、演算結果が表せる範囲を超えてしまうこと。「溢れ」とも言う。

オーバーフロー：4 ビットの例

- 4 ビットの例 (結果も 4 ビット):

符号無し表現

```
  0 1 1 0   (6)
+ 0 1 1 1   (7)
-----------
0 1 1 0 1  (13)  OK!
```

Carry

同じパターン ⟺

2の補数表現

```
  0 1 1 0  (+6)
+ 0 1 1 1  (+7)
-----------
0 1 1 0 1  (–3)  Overflow!
```

Carry

符号無し表現

```
  1 1 0 0  (12)
+ 0 1 1 1   (7)
-----------
1 0 0 1 1   (3)  Overflow!
```

Carry

同じパターン ⟺

2の補数表現

```
  1 1 0 0  (–4)
+ 0 1 1 1  (+7)
-----------
1 0 0 1 1  (+3)  OK!
```

Carry = 1、オーバーフローしない

- 注意: Carry ≠ Overflow

オーバーフロー：32 ビットの例 1

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)

オーバーフロー：32 ビットの例2

add_32.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 = 0x7fffffff;
    unsigned int b = 0x7fffffff;
    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;
}
```

-:--- add_32.c All L32 (C/*l Abbrev)

⇓ 注目

$s = a + b$

オーバーフロー：32 ビットの例 1

logic — -zsh ▸ Emacs-arm64-11 — 74×22

```
yamin@mac logic % gcc -o add32 add32.c
yamin@mac 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
yamin@mac logic %
```

$s = a + b$

← Overflow ・・・・・・・・・(絶対値の表現)

← Okay ・・・・・・・・・・・・・・・(2 の補数の表現)

オーバーフロー：32 ビットの例2

logic — -zsh ▸ Emacs-arm64-11 — 74×22

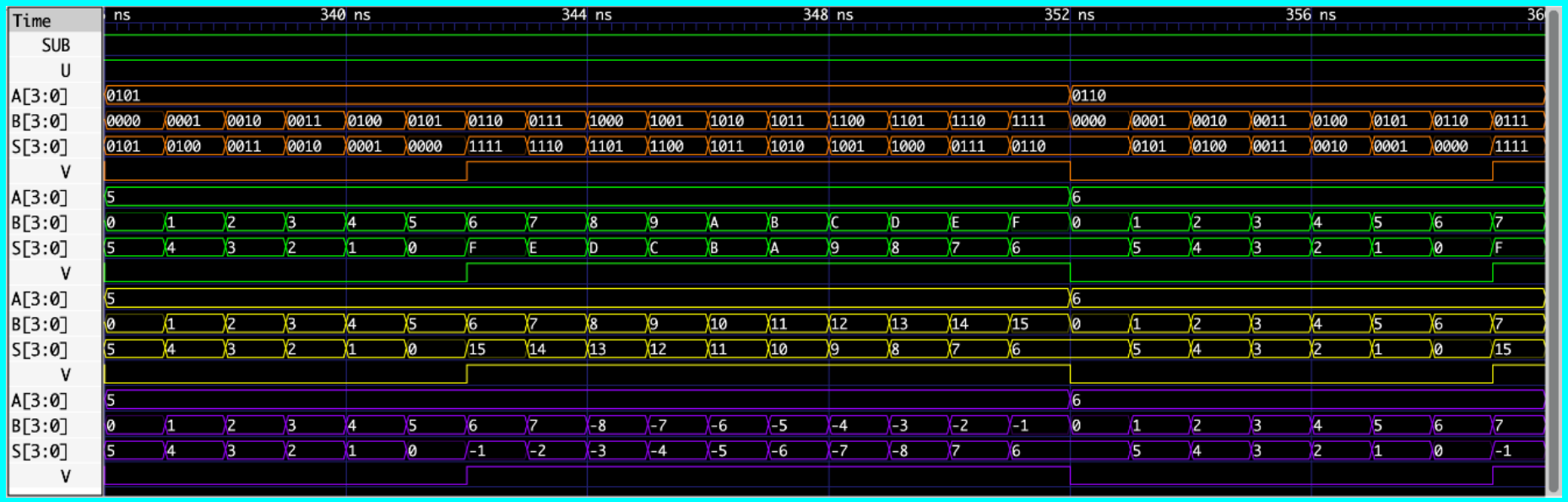
```
[yamin@mac logic % gcc -o add_32 add_32.c
[yamin@mac logic % ./add_32
Binary:
a = 01111111111111111111111111111111
b = 01111111111111111111111111111111
s = 11111111111111111111111111111110

Hexadecimal:
a = 7fffffff
b = 7fffffff
s = fffffffe

Unsigned:
a = 2147483647
b = 2147483647
s = 4294967294

Decimal:
a = 2147483647
b = 2147483647
s = -2
yamin@mac logic %
```

$s = a + b$

← Okay ・・・・・・・・・・・・・(絶対値の表現)

← Overflow ・・・・・・・・・・・・(2 の補数の表現)

gtkwave 信号 Radix の選択



On the gtkwave window, right click a signal, select “Data Format”, and then select one from

“Binary” . . . . . 二進数
“Hex” . . . . . 十六進数
“Decimal” . . . . . 符号なし十進数
“Signed Decimal” . . . . . 符号付き十進数

オーバーフローのまとめ

1. 符号付き数の計算 S = A ± B

	A の MSB	B の MSB	S の MSB	V	コメント
加算	0	0	1	1	正 + 正 = 負
加算	1	1	0	1	負 + 負 = 正
減算	0	1	1	1	正 − 負 = 負
減算	1	0	0	1	負 − 正 = 正

MSB: 最上位ビット; V: オーバーフロー

2. 符号なし数の計算 S = A ± B

	C	V	コメント
加算	1	1	繰り上がり = 1
減算	0	1	繰り上がり = 0

C: 繰り上がり; V: オーバーフロー

2 進数の減算

まとめ

- 符号なし数
- 負の整数
- 2 の補数で表現する方法
- X から −X を求める
- 2 進数の減算
- 加算器を利用した減算
- 加減算器
- オーバーフロー (符号なしの場合、2 の補数の場合)

課題 VII (100 点 + 100 点)

P15-P20 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加減算器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
入力信号: SUB . . . . . 1-bit. 0: add; 1: subtract

出力信号: S[3:0] . . . . . 4-bit result

モジュール名は addsub4 にすること。
テストベンチ addsub4_tb.v を使って下さい。

課題 VII (100 点 + 100 点)

オプション (+100 点): P28 を参照し、4 ビットリップルキャリー加減算器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
入力信号: SUB . . . . . 1-bit. 0: add; 1: subtract
入力信号: U . . . . . 1-bit. 0: signed; 1: unsigned

出力信号: S[3:0] . . . . . 4-bit result
出力信号: V . . . . 1-bit. 0: no overflow; 1: overflow

モジュール名は addsub4v にすること。
テストベンチ addsub4v_tb.v を使って下さい。

課題 VII (100 点 + 100 点)

```
‘timescale 1ns/1ns
module addsub4v (A, B, SUB, U, V, S);
    input  [3:0] A, B;
    input        SUB, U;
    output [3:0] S;
    output       V;

    wire [3:0]   bb = B ^ {4{SUB}}; // XOR
    wire [3:0]   c;

    // add1 (a,    b,     ci,   co,   s);
    add1 i0 (A[0], bb[0], SUB,  c[0], S[0]);
    add1 i1 (    ,      ,     ,     );
    add1 i2 (    ,      ,     ,     );
    add1 i3 (    ,      ,     ,     );
    assign V =                      ;
endmodule
```

addsub4v.v

課題 VII (100 点 + 100 点)

UNSIGNED ADD:

⇑

5 + 11 = 0 Overflow

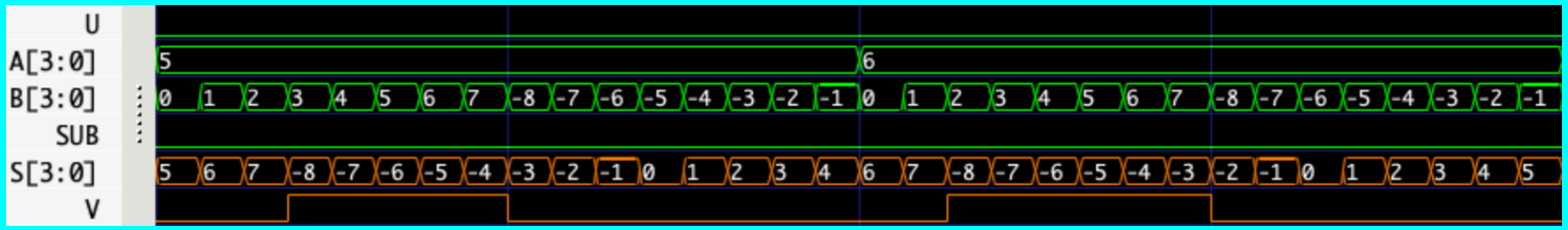
UNSIGNED SUB:

⇑

5 − 6 = 15 Overflow

課題 VII (100 点 + 100 点)

SIGNED ADD:



$\Uparrow$

$5 + 5 = -6$ Overflow

SIGNED SUB:

$\Uparrow$

$5 - (-8) = -3$ Overflow

課題 VII (100 点 + 100 点)

SIGNED ADD:

$$(-8)+(-8)=0 \quad \text{Overflow}$$

SIGNED SUB:

$$(-8)-1=7 \quad \text{Overflow}$$