

論理回路入門（7）

2進数の減算と加減算器の設計

李 亜民

2022年11月8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)

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}

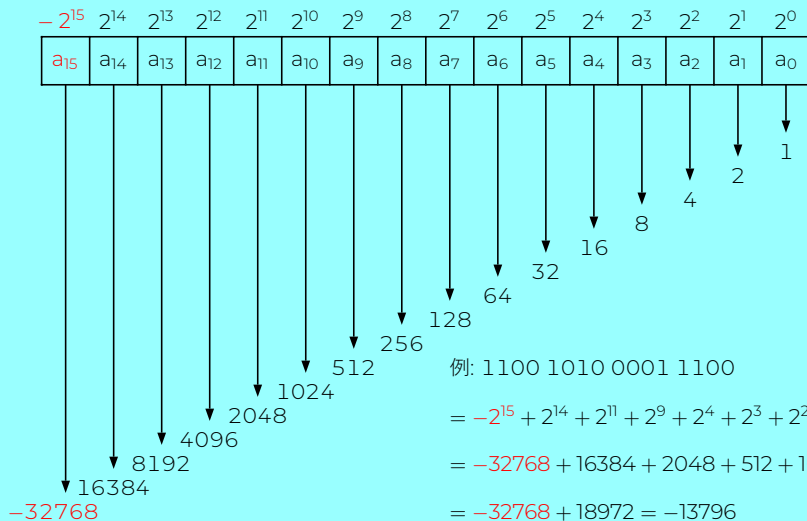
- n ビット 2 の補数で表される符号付き数のうち最小値は -2^{n-1} 、最大値は $+(2^{n-1} - 1)$ である。

例： $n = 4$ のとき、最小値は -8 、最大値は $+7$ である。

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

2の補数で表現する方法

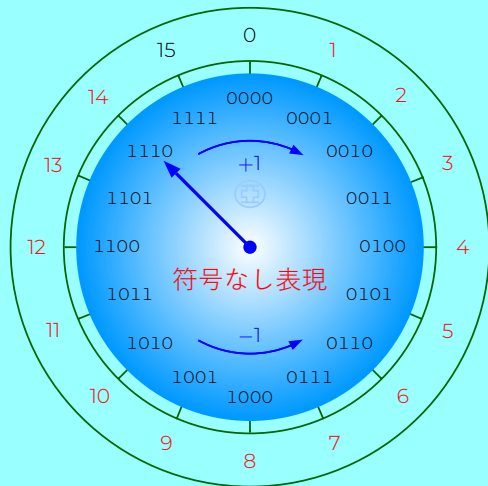


4ビット2進数のまとめ

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

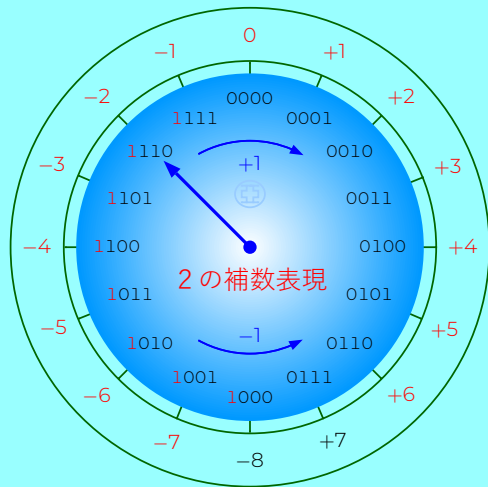
1111 = -8 + (+7) = -1 (2の補数表現)

符号なしでの表現 (4ビットの例)



表現範囲：0 ~ 15 最大値 $15 + 1 = 0$ (最小値)

2の補数での表現 (4ビットの例)



表現範囲： $-8 \sim +7 \dots\dots\dots$ 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、

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

- X から $-X$ を求める例 1

- ▶ $X = 0011 = +3_{10}$
- ▶ $-X = \bar{X} + 1 = 1100 + 1 = 1101 = -3_{10}$

- X から $-X$ を求める例 2

- ▶ $X = 1101 = -3_{10}$
- ▶ $-X = \bar{X} + 1 = 0010 + 1 = 0011 = +3_{10}$

2進数の減算

- 2の補数では、 $-X = \bar{X} + 1$
- 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。

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

- 2の補数を使うと、 $X + (-X) \equiv 0$
 - ▶ 理由は
 - ★ $X + \bar{X} = 11 \dots 11_2 = -1_{10}$ 、左右に1を加算すると
 - ★ $X + \bar{X} + 1 = -1 + 1$ 、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、 $X - Y = X + (-Y) = X + \bar{Y} + 1$
- 問題: $Z = X - Y = 10001100 - 11010111$ を計算せよ。
- 解答: $-Y = \bar{Y} + 1 = 00101000 + 1 = 00101001$

$X + (-Y)$:

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

10進数での確認:

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

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

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

$$\text{即ち、} Z = X - Y = (-116) - (-41) = -75$$

2進数の減算

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

$X + (-Y)$:

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

10進数での確認:

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

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

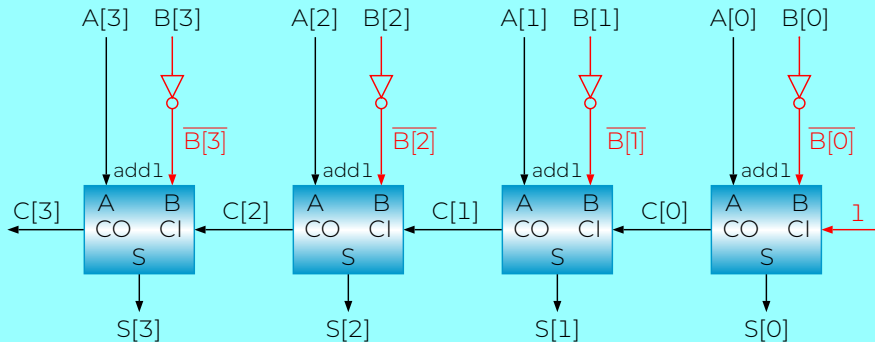
$$Z = 10100101_2 = -128 + 32 + 4 + 1 = -91$$

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

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

2進数減算器

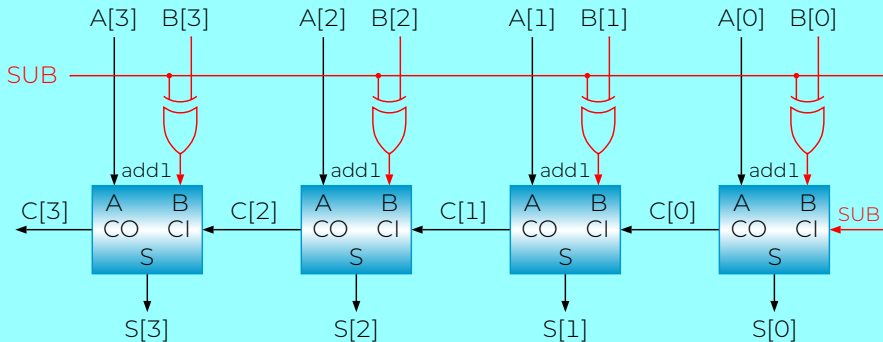
$-X = \bar{X} + 1$ なので、 $A - B = A + (-B) = A + \bar{B} + 1$



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




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

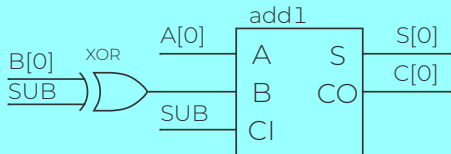


もし $SUB = 0$ 、 $S = A + B + 0 = A + B$ ($B \oplus 0000 = B$)

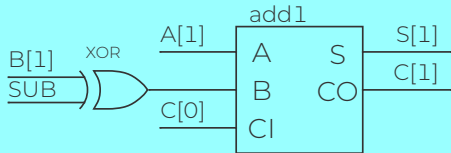
もし $SUB = 1$ 、 $S = A + \bar{B} + 1 = A - B$ ($B \oplus 1111 = \bar{B}$)

2 進数加減算器

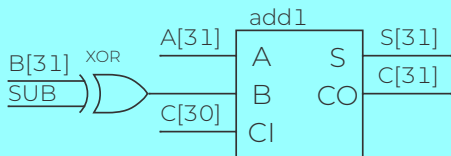
A[31..0] 
B[31..0] 
SUB 



 S[31..0]



...



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 $0 \sim 15$ 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 $-8 \sim +7$ 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		2の補数表現
	0110 (6)	同じパターン	0110 (+6)
	+ 0111 (7)	⇔	+ 0111 (+7)
	<u> </u>		<u> </u>
Carry →	01101 (13) OK!		01101 (-3) Overflow!
	符号無し表現		2の補数表現
	1100 (12)	同じパターン	1100 (-4)
	+ 0111 (7)	⇔	+ 0111 (+7)
	<u> </u>		<u> </u>
Carry →	10011 (3) Overflow!		10011 (+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

```
emacs@localhost
File Edit Options Buffers Tools C Help
#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 L31 (C/*\ Abbrev)
```

↓ 注目

$s = a + b$

add_32.c

オーバーフロー：32 ビットの例2

```
emacs@localhost
File Edit Options Buffers Tools C Help
#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;
}
--:--- add32.c All L31 (C/*\ Abbrev)
```

↓ 注目

$s = a + b$

add32.c

オーバーフロー：32 ビットの例 1

```
yamin@localhost:/home/yamin/lectures/logic
~/lectures/logic $ gcc add_32.c -o add_32
~/lectures/logic $ ./add_32
Binary:
a = 01111111111111111111111111111111
b = 01111111111111111111111111111111
s = 11111111111111111111111111111110 ← ..... (2 進数の表現)
Hexadecimal:
a = 7fffffff
b = 7fffffff
s = ffffffff ← ..... (16 進数の表現)
Unsigned:
a = 2147483647
b = 2147483647
s = 4294967294 ← Okay ..... (絶対値の表現)
Decimal:
a = 2147483647
b = 2147483647
s = -2 ← Overflow ..... (2 の補数の表現)
~/lectures/logic $
```

オーバーフロー：32 ビットの例 2

```
yamin@localhost:/home/yamin/lectures/logic
~/lectures/logic $ gcc add32.c -o add32
~/lectures/logic $ ./add32
Binary:
a = 11111111111111111111111111111111
b = 11111111111111111111111111111111
s = 11111111111111111111111111111110 ← ..... (2 進数の表現)
Hexadecimal:
a = ffffffff
b = ffffffff
s = fffffffe ← ..... (16 進数の表現)
Unsigned:
a = 4294967295
b = 4294967295
s = 4294967294 ← Overflow ..... (絶対値の表現)
Decimal:
a = -1
b = -1
s = -2 ← Okay ..... (2 の補数の表現)
~/lectures/logic $
```

オーバーフローのまとめ

① 符号付き数の計算 $S = A \pm B$

	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 $S = A \pm B$

	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)

全加算器を使って、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

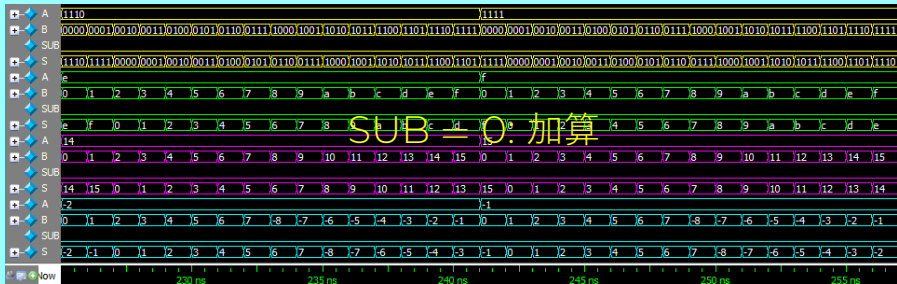
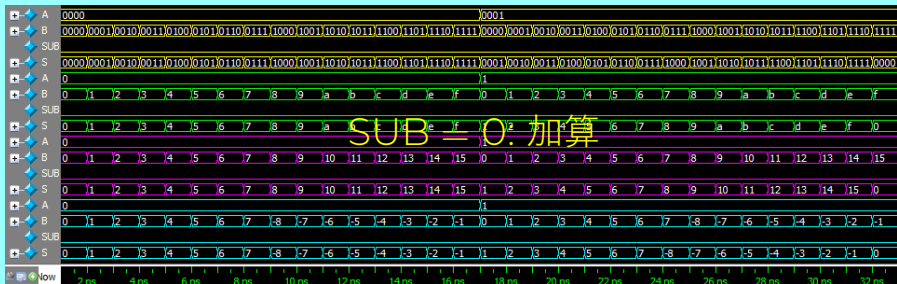
入力信号: SUB 1-bit. 0: add; 1: subtract

出力信号: S[3..0] 4-bit result

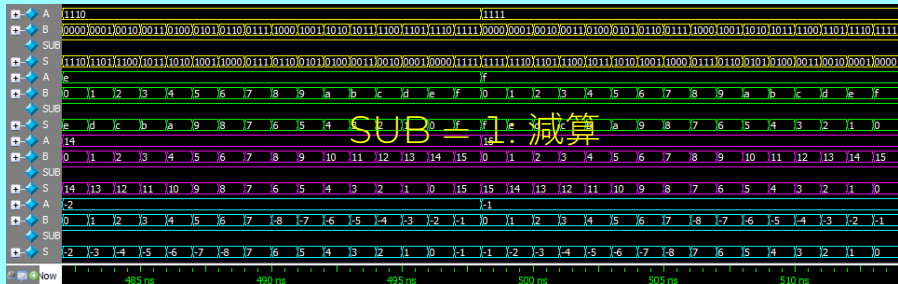
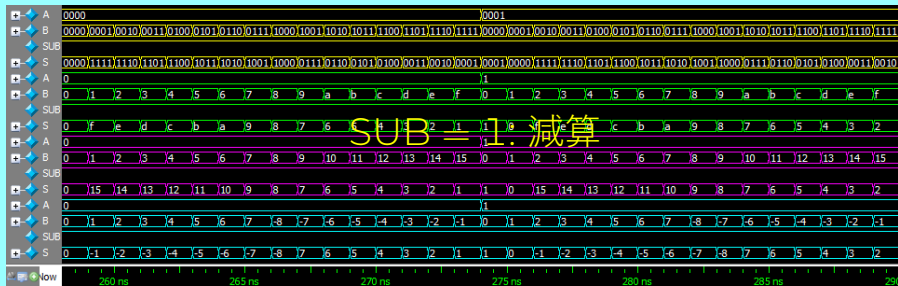
プロジェクト名は `addsub4` にすること。

テストベンチ [addsub4_tb.v](#) を使って下さい。

課題 VII (100 点 + 100 点)



課題 VII (100 点 + 100 点)



課題 VII (100 点 + 100 点)

オプション (+100 点): P24 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

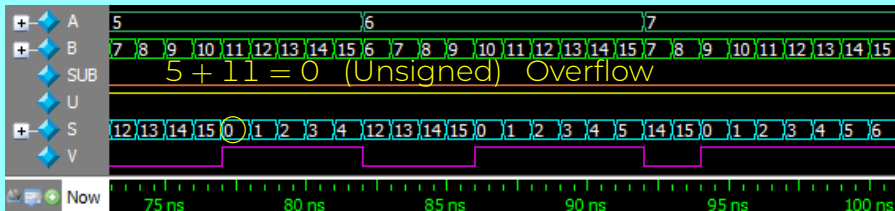
出力信号: V 1-bit. 0: no overflow; 1: overflow

プロジェクト名は `addsub4v` にすること。

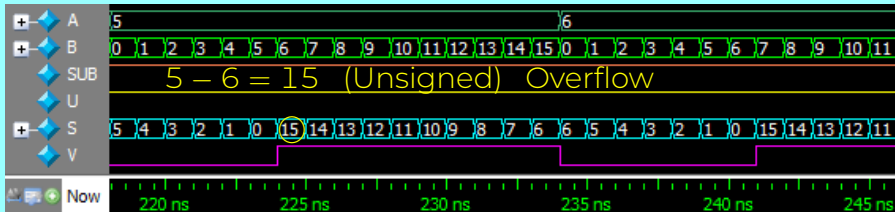
テストベンチ [addsub4v_tb.v](#) を使って下さい。

課題 VII (100 点 + 100 点)

UNSIGNED ADD:

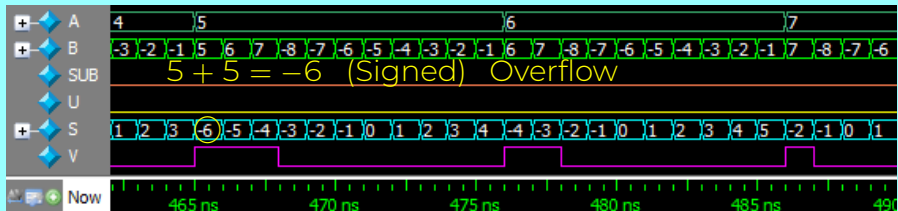


UNSIGNED SUB:

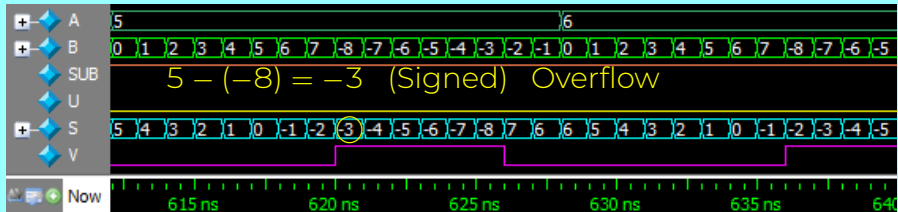


課題 VII (100 点 + 100 点)

SIGNED ADD:

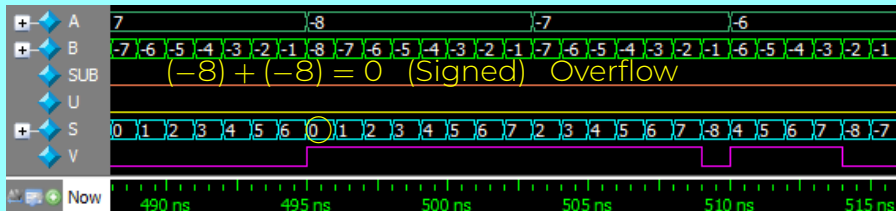


SIGNED SUB:

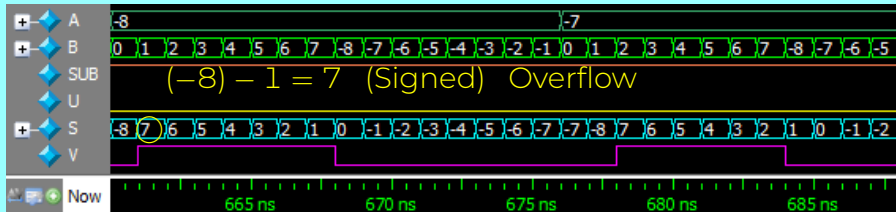


課題 VII (100 点 + 100 点)

SIGNED ADD:



SIGNED SUB:



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ビット加減算器の設計 (P14、P15、P16 を参照)

```
module addsub4b (A, B, SUB, S);
    input  [3:0] A, B;
    input          SUB;
    output [3:0] S;

    wire  [3:0] bs = B ^ {4{SUB}}; // internal wires
    wire  [3:0] c; // internal wires
    add1b i0 (A[0], bs[0], SUB, c[0], S[0]); // full adder 0
    add1b i1 (A[1], bs[1], , , S[1]); // full adder 1
    add1b i2 (A[2], bs[2], , , S[2]); // full adder 2
    add1b i3 (A[3], bs[3], , c[3], S[3]); // full adder 3

endmodule
```

発展：自由練習

Verilog HDL による課題の実装

4ビット加減算器 (オーバーフロー) の設計 (P23 を参照)

```
module addsub4bv (A, B, SUB, U, S, V);
    input  [3:0] A, B;
    input          SUB, U; // U: unsigned
    output [3:0] S;
    output          V; // V: overflow
    wire  [3:0] bs = B ^ {4{SUB}}; // internal wires
    wire  [3:0] c; // internal wires
    add1b i0 (A[0], bs[0], SUB, c[0], S[0]); // full adder 0
    add1b i1 (A[1], bs[1], , S[1]); // full adder 1
    add1b i2 (A[2], bs[2], , S[2]); // full adder 2
    add1b i3 (A[3], bs[3], c[3], S[3]); // full adder 3
    assign V = ; // overflow
endmodule
```