第１回 ハードウェア記述言語
～回路のゲートレベル記述、シミュレーションの方法～

中野 秀洋

ノートPC、電源、LANケーブルを毎回持参

https://webclass.tcu.ac.jp/

参考書：

HDLによるVLSI設計

～VerilogHDLとVHDLによるCPU設計～

深山・北川・秋田・鈴木　共著、共立出版

はじめに

- LSI (Large Scale Integration)の大規模化・複雑化
  - 回路設計困難
- HDL (Hardware Description Language)
  - 設計が容易

LSI 設計段階と表現

抽象度　記述量

高い　少ない

低い　多い

仕様設計

RTL設計
（機能設計）

ゲートレベル設計
（論理設計）

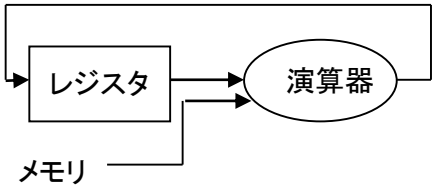
トランジスタレベル設計
（回路設計）

レイアウト設計

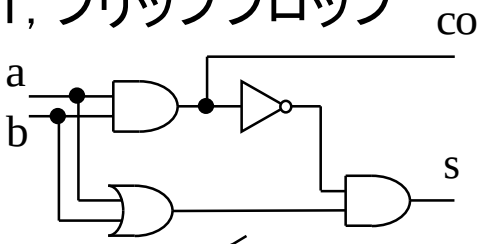
命令セット，レジスタセット
データ、アドレス、制御信号
パイプライン、割込み、キャッシュ

```
LD  GR1, 100, GR7
ADD GR1, 200, GR7
```

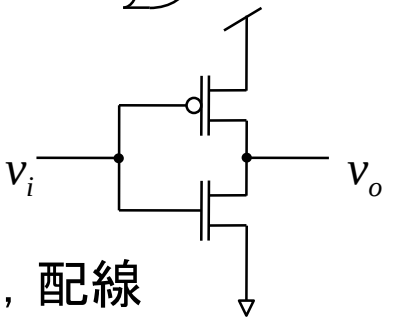
組合せ回路とレジスタ



AND, OR, NOT, フリップフロップ



トランジスタ



チップに配置，配線

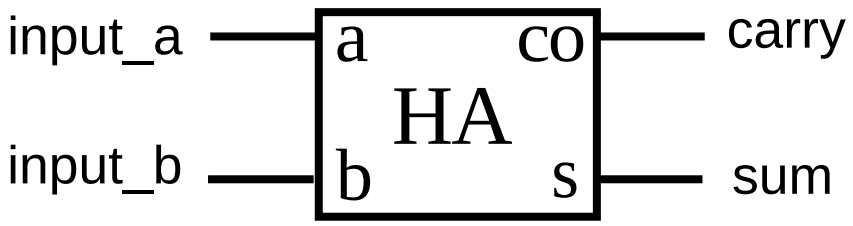
HDL設計の利点

- 回路記述の簡単化
  - 例）加算器: z = a + b
- シミュレーション早期開始が可能
  - RTL設計のままシミュレーションが可能
    (RTL; Register Transfer Level)
- ゲートレベル設計の自動化
  - 論理合成ツールが論理回路をFPGAなどに自動的に合成
    (FPGA; Field Programmable Gate Array)

本講義で学習するHDL:　　Verilog HDL

Half Adder（半加算器）

a	b	co	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



入力 a, b を加算

加算結果を s に出力

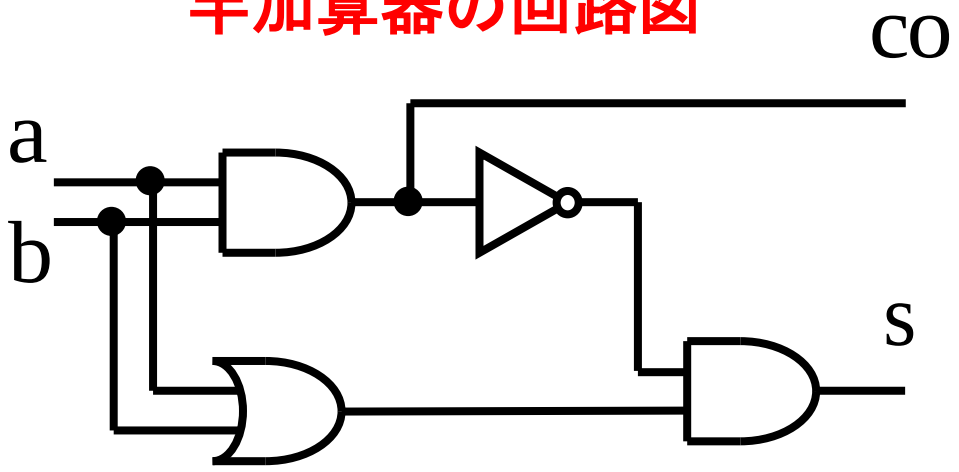
桁上げ（キャリー）を co に出力

半加算器の論理式

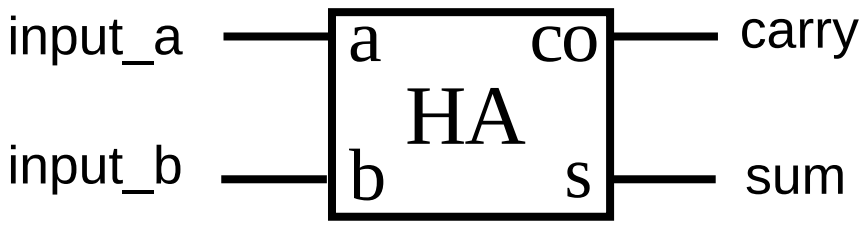
$$s = \overline{(a \cdot b)} \cdot (a + b)$$

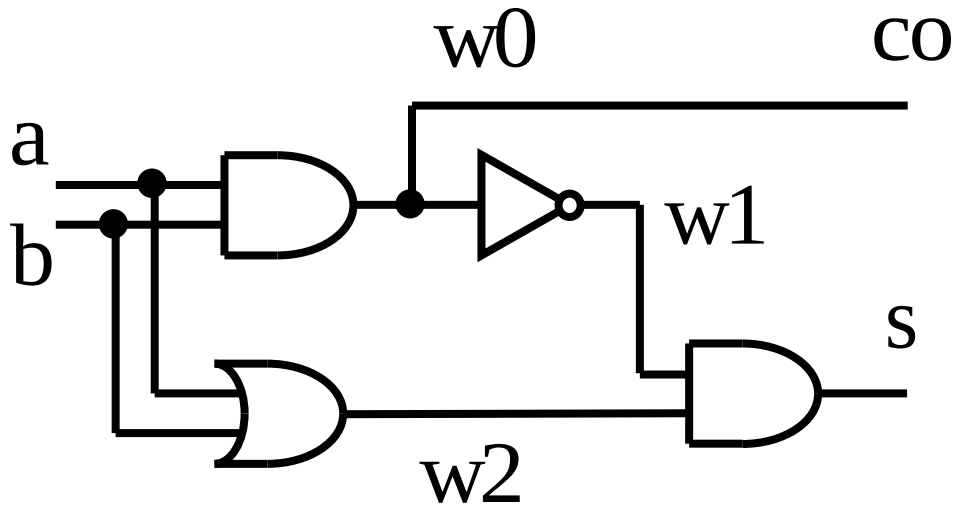
$$co = a \cdot b$$

半加算器の回路図



Half Adder（半加算器）






```
/* 半加算器のmoduleの定義 */
module half_add(a, b, co, s);
    /* input port */
    input a, b;
    /* output port */
    output co, s;
    /* wire */
    wire w0, w1, w2;

    /* assign文 */
    assign w0 = a & b;
    assign w1 = ~w0;
    assign w2 = a | b;
    assign s = w1 & w2;
    assign co = w0;
endmodule
```

assign文で使用できる論理演算子

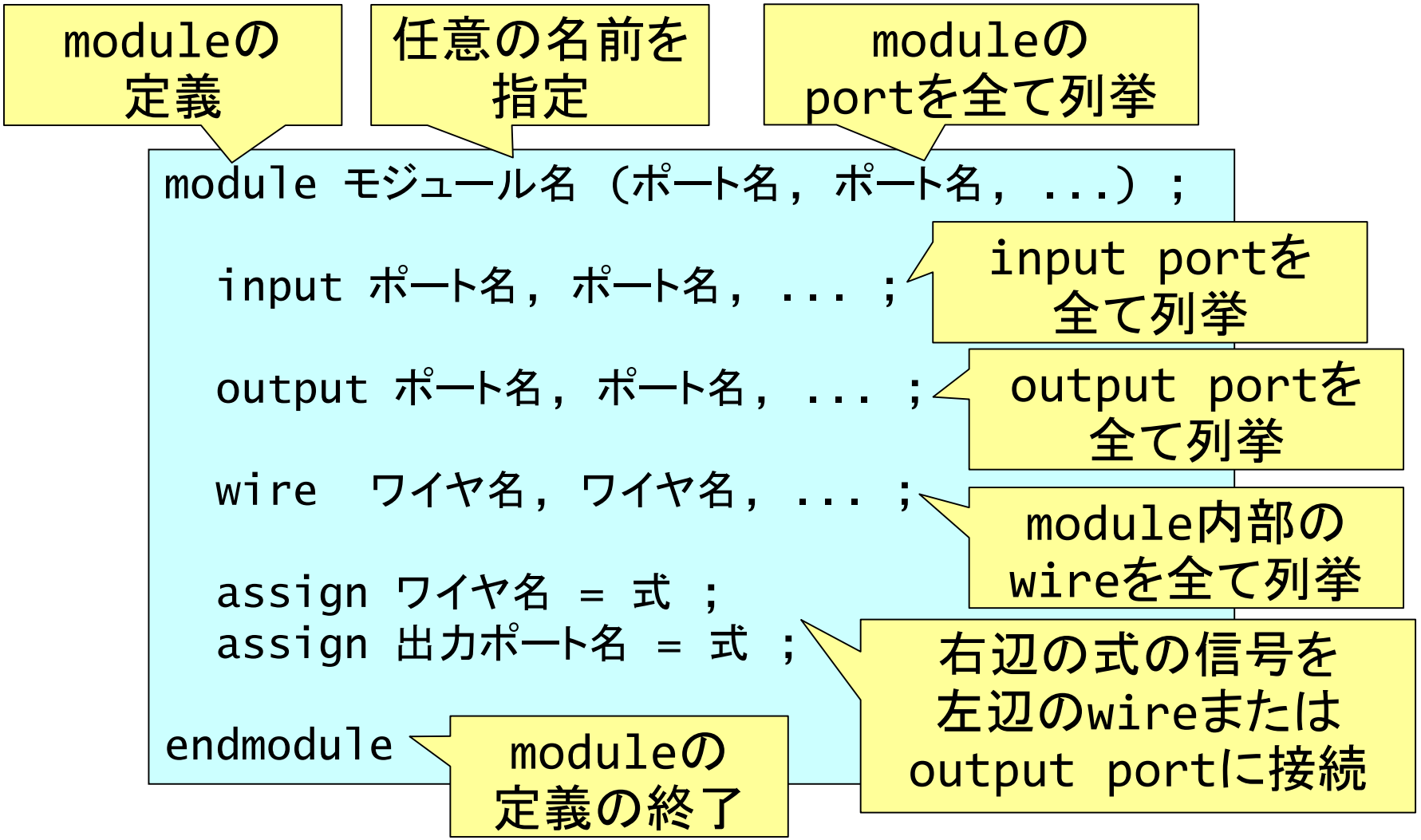
a | b	aとbのOR
a & b	aとbのAND
a ^ b	aとbのXOR
~a	aのNOT

a ~| b	aとbのNOR
a ~& b	aとbのNAND
a ~^ b	aとbのXNOR

$$s = \overline{(a \cdot b)} \cdot (a + b)$$

$$co = a \cdot b$$

moduleの定義



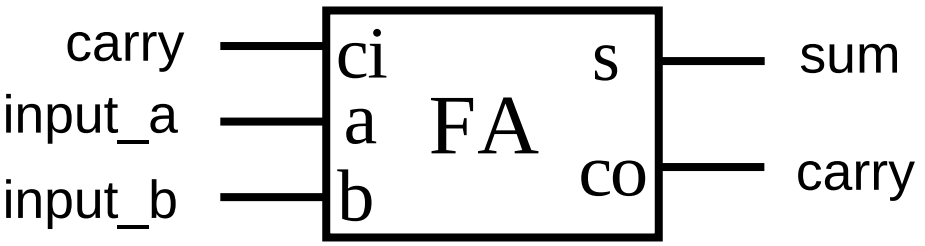
モジュール名、ポート名、ワイヤ名として以下は使用できない

数字から始まる文字列、文法で定義済みの文字列

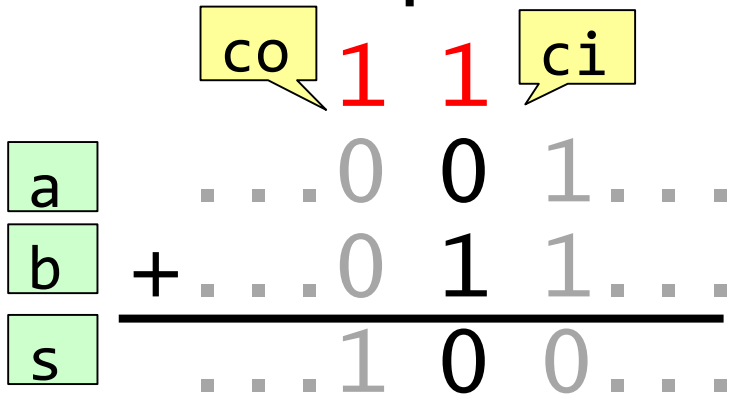
演算記号などを含む文字列

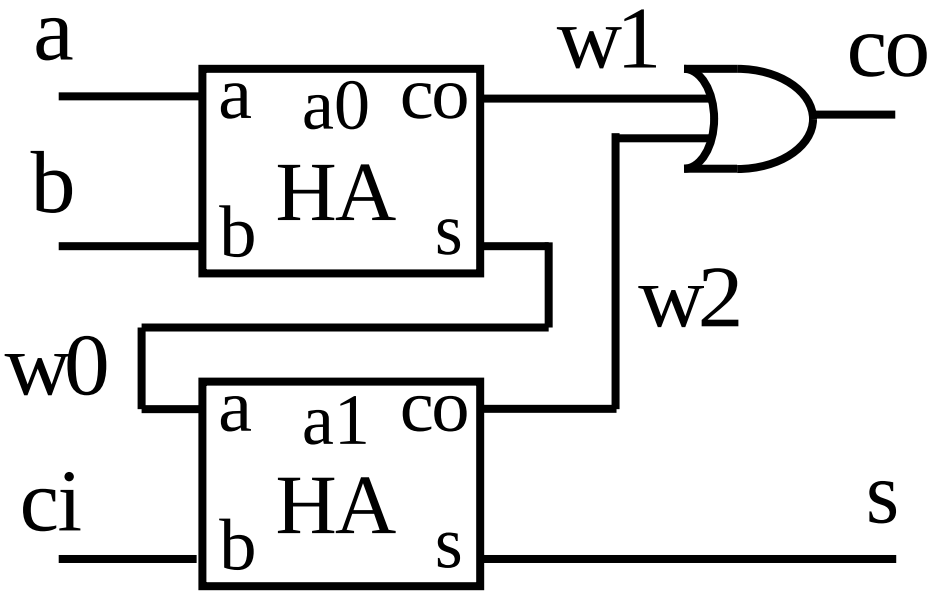
Full Adder（全加算器）

a	b	ci	co	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

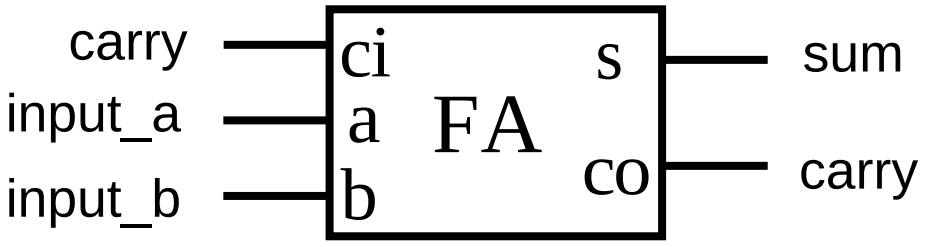


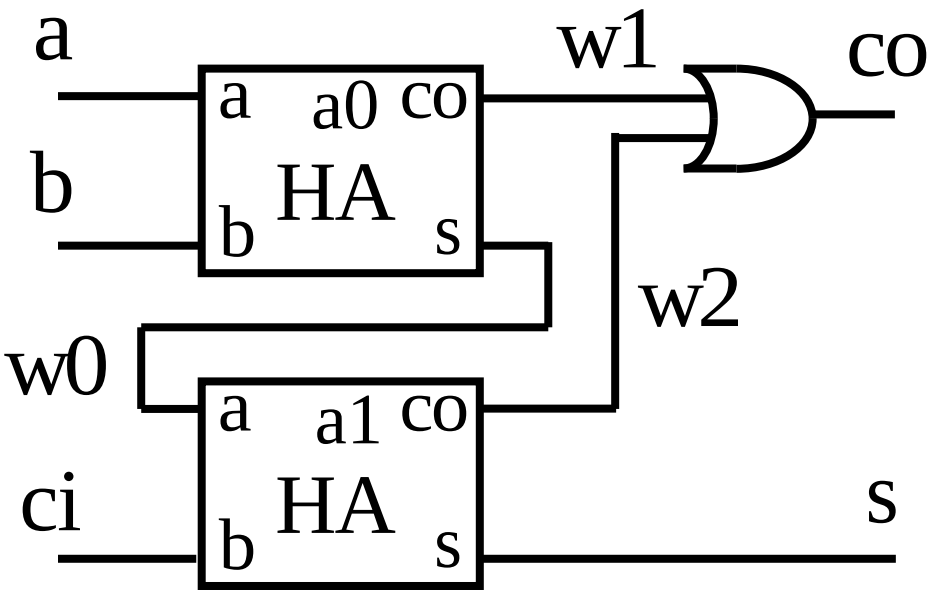
2つのHAとORゲート:
入力a, bを加算して
下位からのキャリーciを加算
どちらかの加算で桁上げしたら
上位へのキャリーcoを出力





Full Adder（全加算器）






```
/* 全加算器のmoduleの定義 */
module full_add(a, b, ci, co, s);
    /* input port */
    input a, b, ci;
    /* output port */
    output co, s;
    /* wire */
    wire w0, w1, w2;

    /* instance文 */
    half_add a0(.a(a),   .b(b),
                .co(w1), .s(w0));
                          w0を介して接続
    half_add a1(.a(w0),  .b(ci),
                .co(w2), .s(s));
    /* assign文 */
    assign co = w1 | w2;
endmodule
```

instance文（下位moduleの使用）

```
module モジュール名 (ポート名, ポート名, ...) ;

  input ポート名, ポート名, ... ;
  output ポート名, ポート名, ... ;
  wire  ワイヤ名, ワイヤ名, ... ;

  assign ワイヤ名 = 式 ;
  assign 出力ポート名 = 式 ;

  モジュール名 インスタンス名( .ポート名( ワイヤ名 ), ... ) ;

endmodule
```

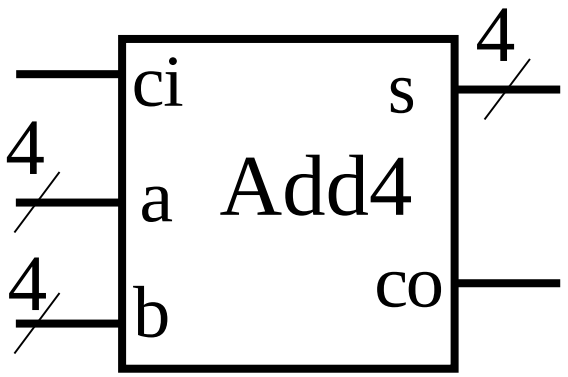
使用する下位moduleの
instanceの名前

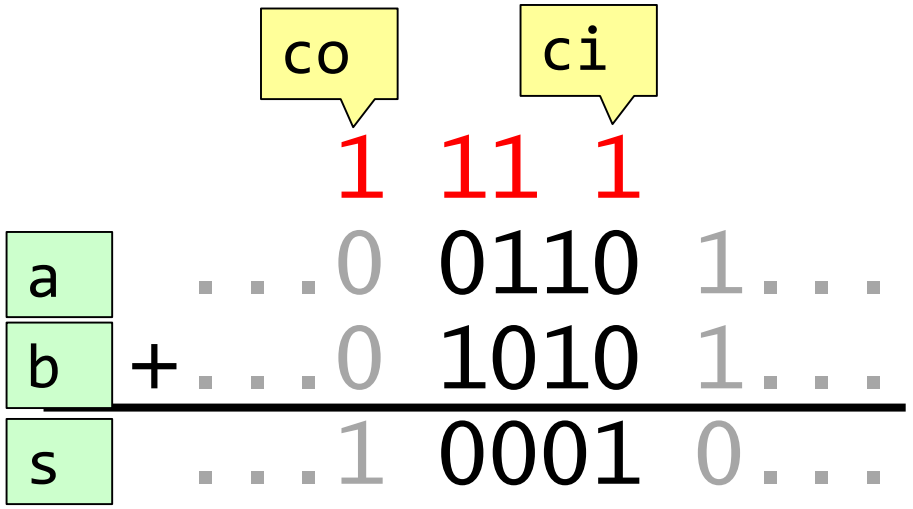
下位moduleで
宣言したportの名前

定義済みの
下位moduleの名前

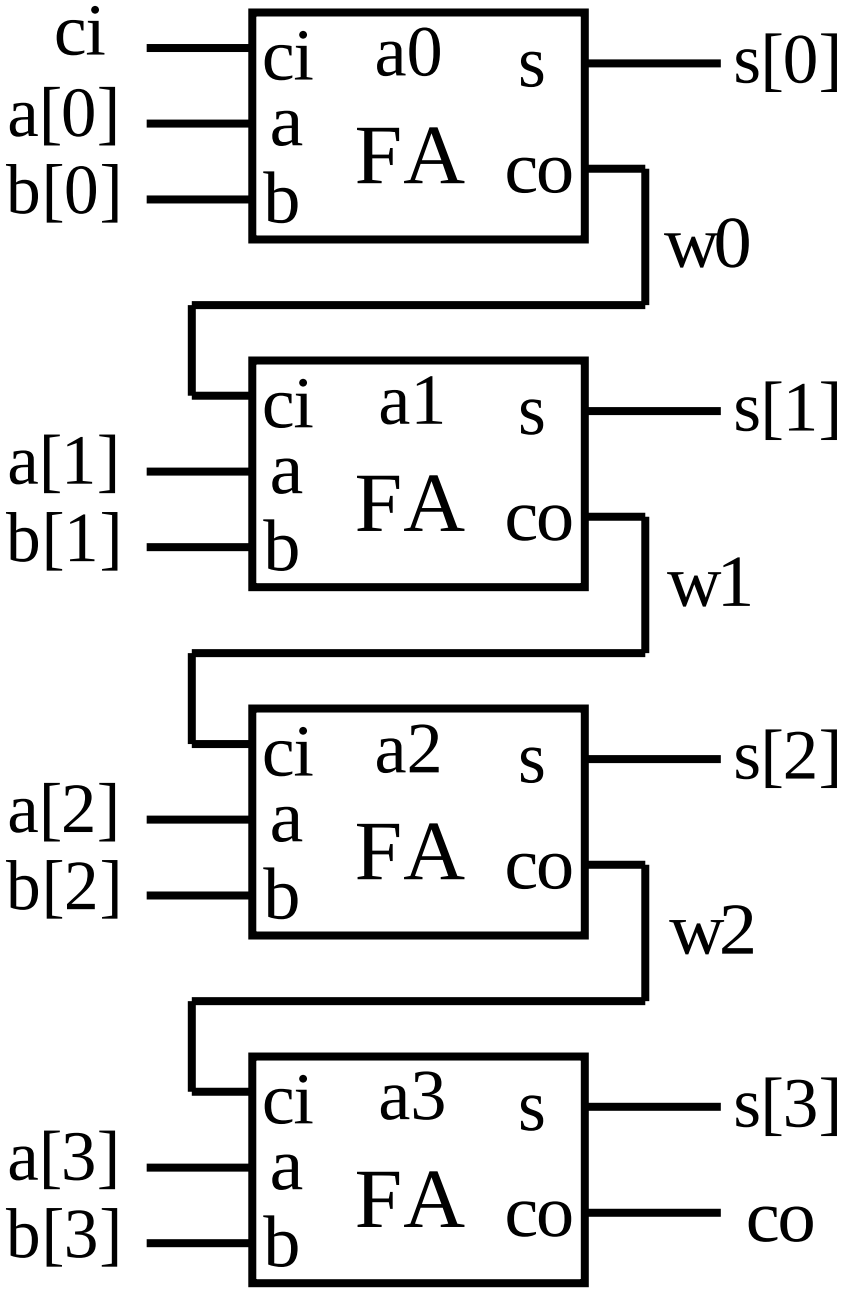
上位moduleで
宣言したwireの名前
またはportの名前

4bit Adder（加算器）

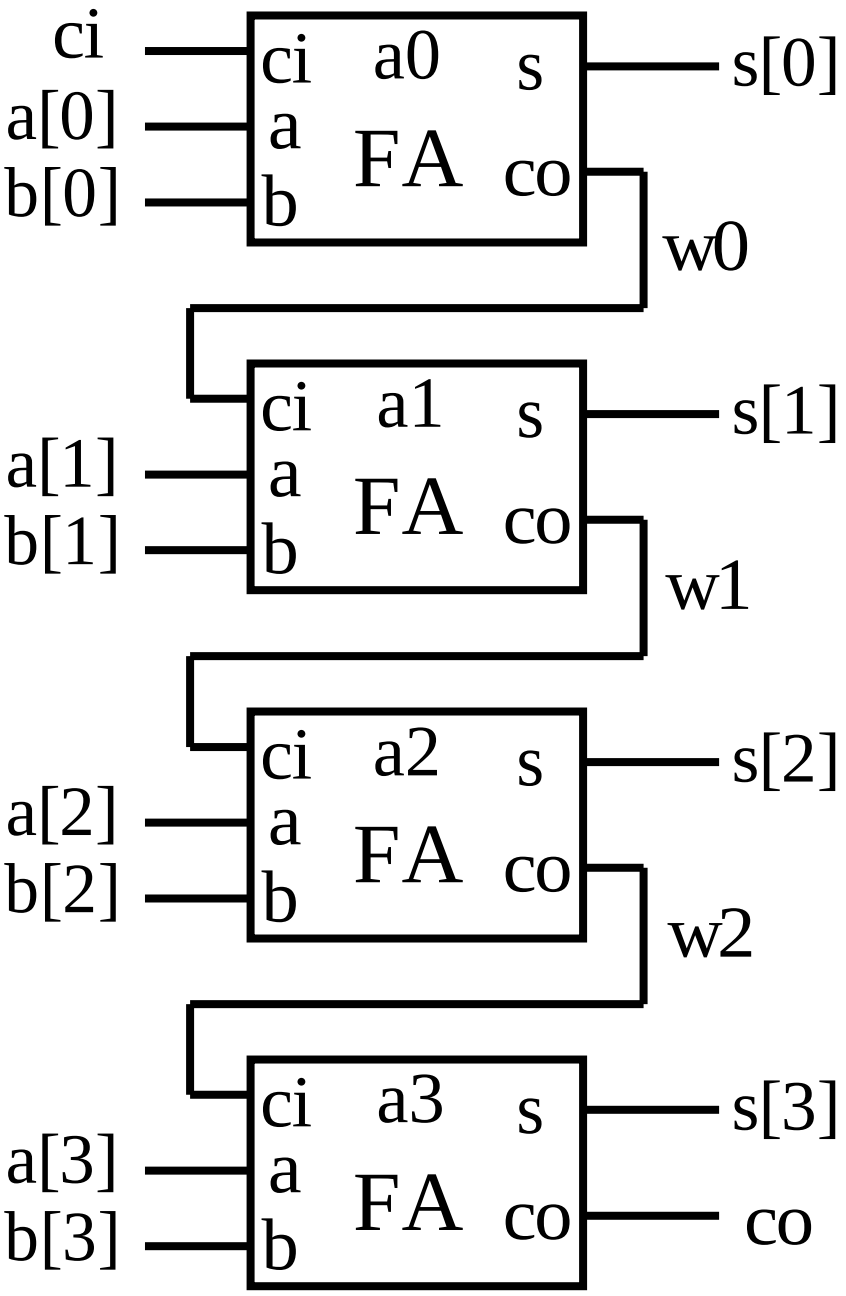




4つの全加算器:
下位桁のFAのキャリーcoと
上位桁のFAのキャリーciを接続



4bit Adder（加算器）

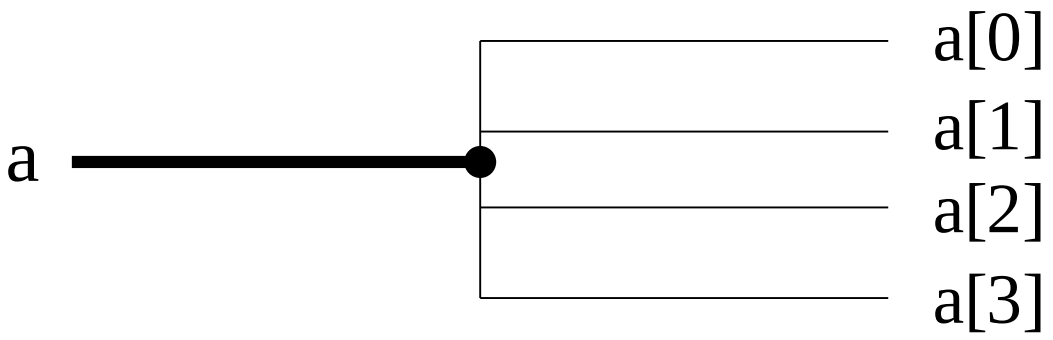



```
/* 4ビット加算器のmoduleの定義 */
module add4(a, b, ci, co, s);
    /* input port */
    input  [3:0] a, b;
    input        ci;
    /* output port */
    output       co;
    output [3:0] s;
    /* wire */
    wire w0, w1, w2;
    /* instance文 */
    full_add a0(.a(a[0]), .b(b[0]),
        .ci(ci), .co(w0), .s(s[0]));

    full_add a1(.a(a[1]), .b(b[1]),
        .ci(w0), .co(w1), .s(s[1]));

    full_add a2(.a(a[2]), .b(b[2]),
        .ci(w1), .co(w2), .s(s[2]));

    full_add a3(.a(a[3]), .b(b[3]),
        .ci(w2), .co(co), .s(s[3]));
endmodule
```

複数ビットの信号（1）

- 宣言方法　［MSB：LSB］ワイヤ名またはポート名
  - 4bitのwireを宣言する例： wire [3:0] a;

例)



- 信号の分離

例)

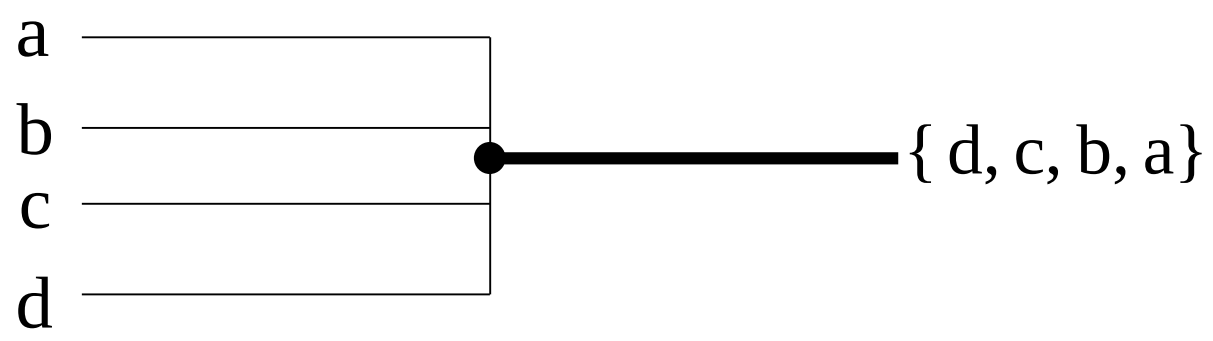
portやwireの宣言でビット幅を省略したときのビット幅は1

assign文やinstance文で省略した場合は宣言時のビット幅

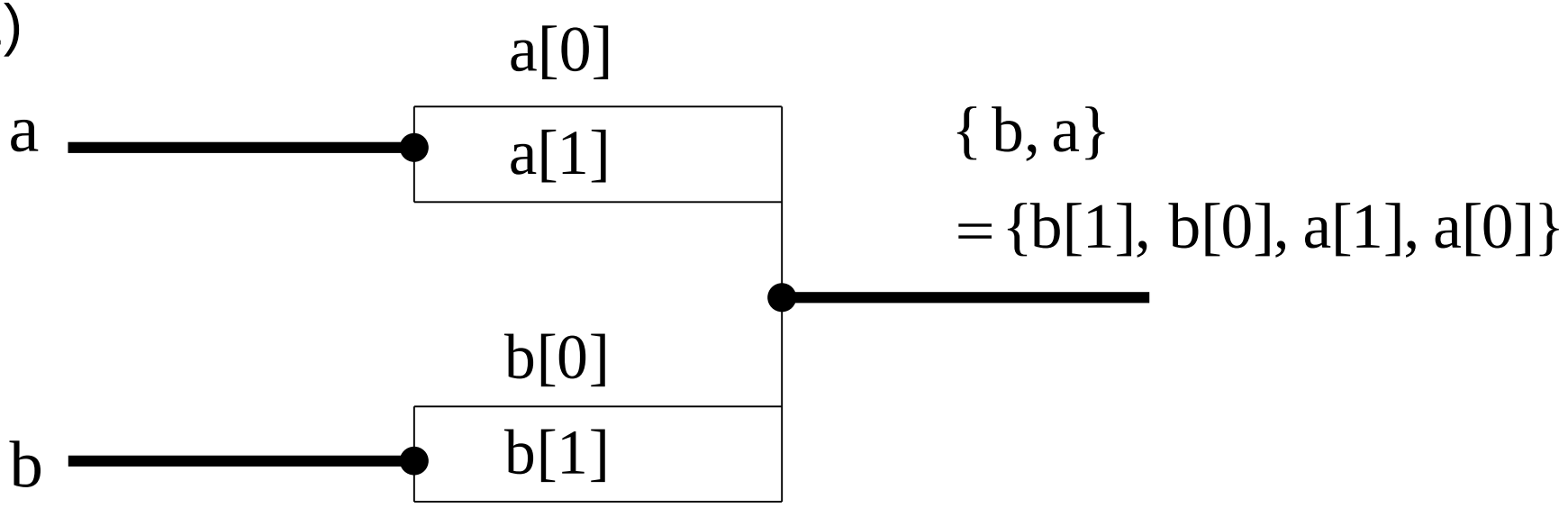
複数ビットの信号（2）

- 信号の連接

例1)

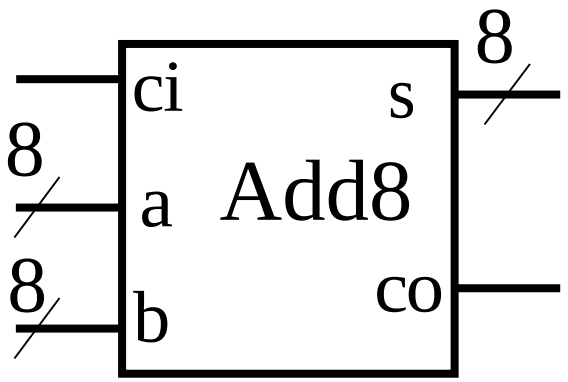


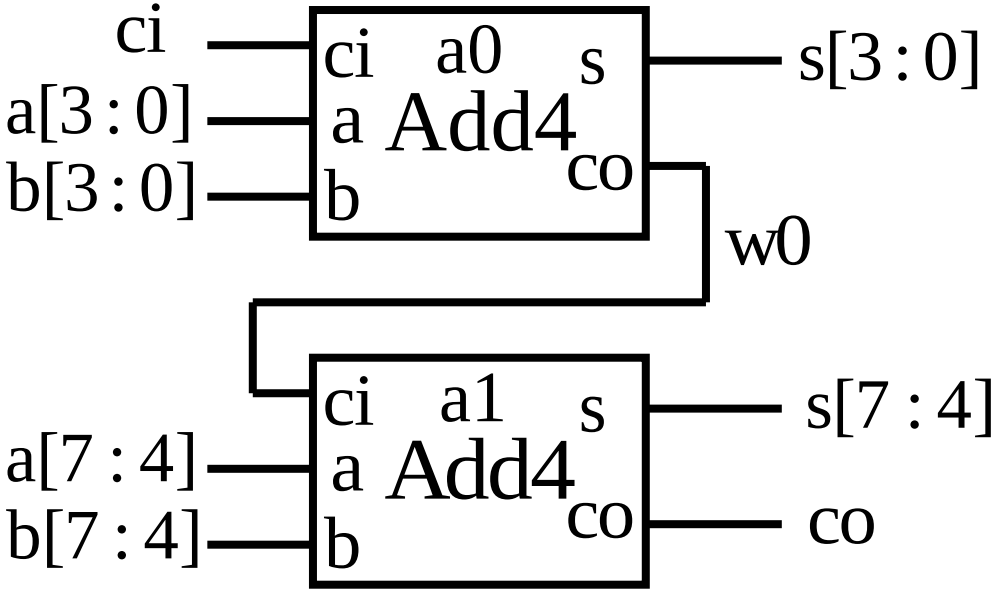
例2)



左に記述した信号が上位ビット、右に記述した信号が下位ビットになる

8bit Adder（加算器）

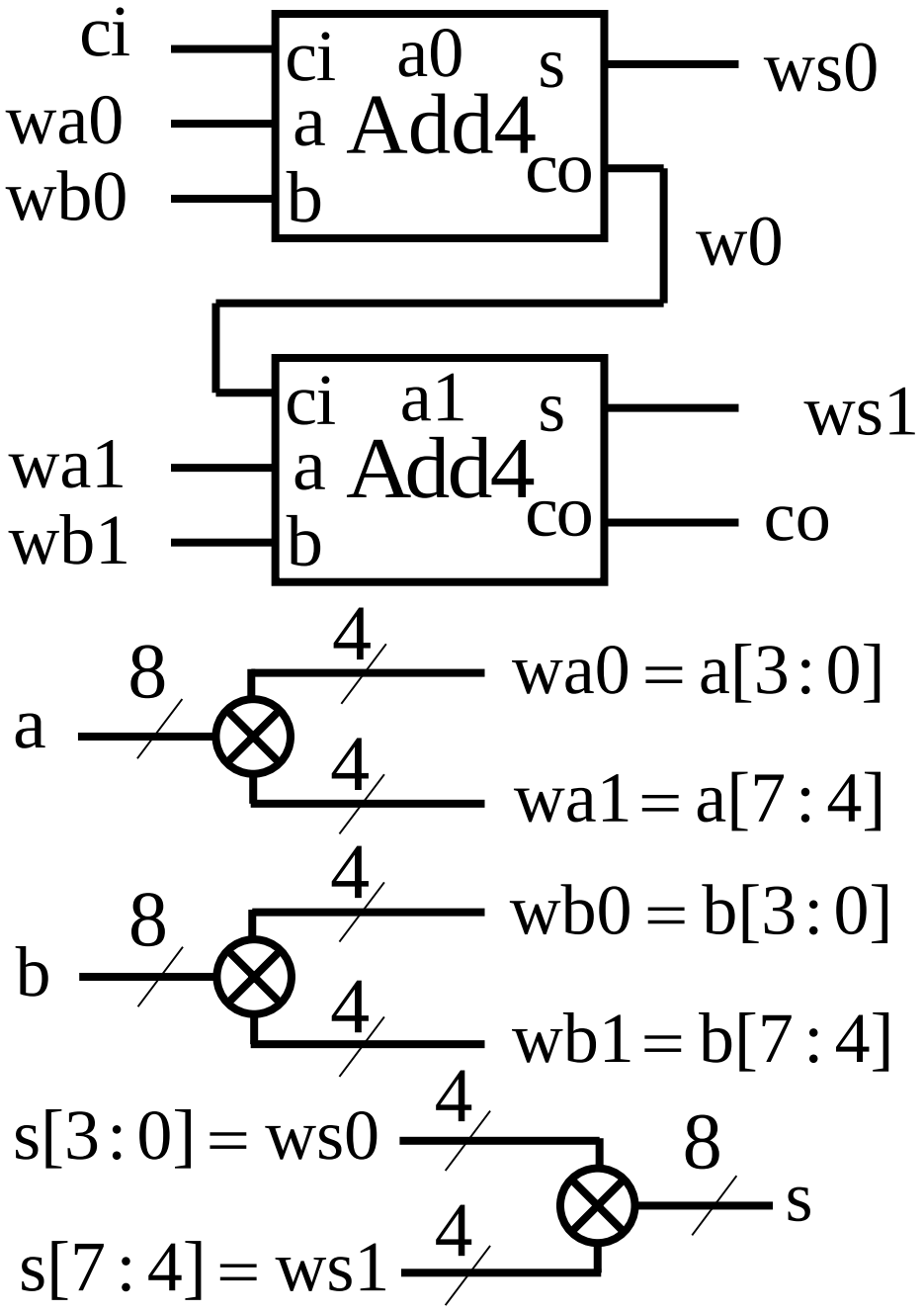





```
 1  11  1
    0110  1010
+   1010  1100
-------------
    0001  0110
```

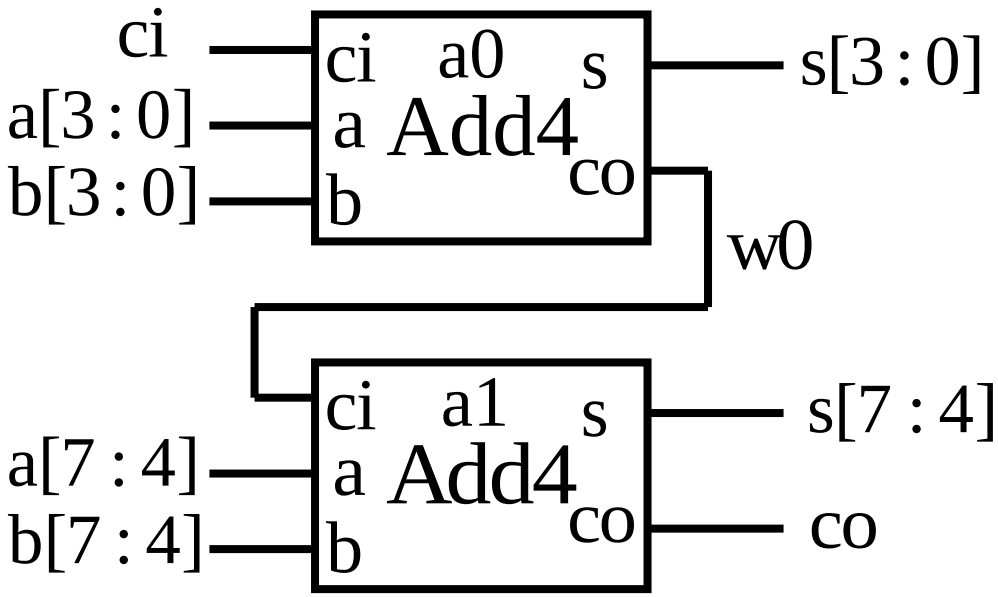
2つのAdd4:
下位桁のAdd4のキャリーcoと
上位桁のAdd4のキャリーciを接続

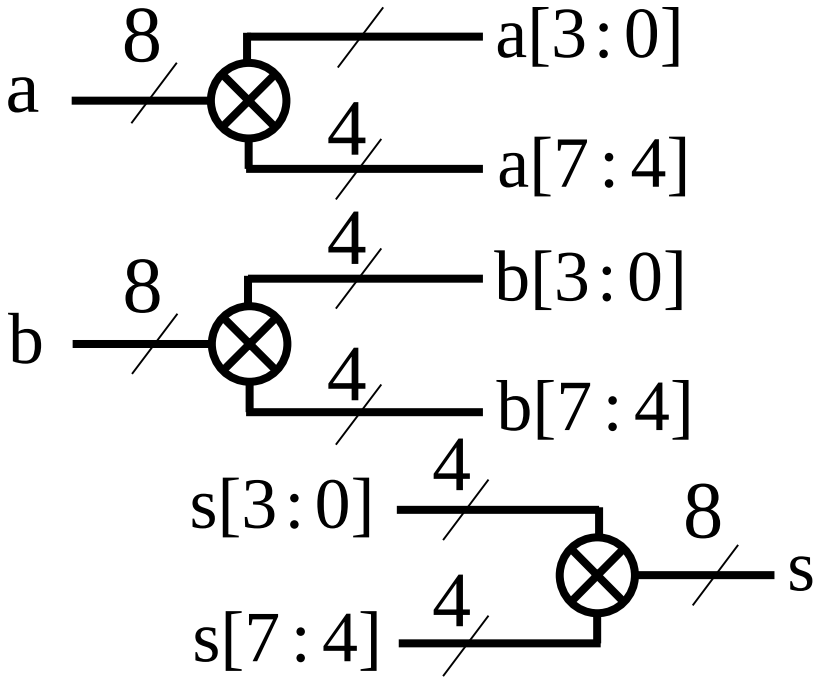
8bit Adder（加算器）




```
/* 8ビット加算器のmoduleの定義 */
module add8(a, b, ci, co, s);
    /* input port */
    input  [7:0] a, b;
    input        ci;
    /* output port */
    output       co;
    output [7:0] s;
    /* wire */
    wire w0;
    wire [3:0] wa0, wb0, ws0;
    wire [3:0] wb1, wb1, ws1;
    /* assign文、instance文 */
    assign wa0 = a[3:0];
    assign wa1 = a[7:4];
    assign wb0 = b[3:0];
    assign wb1 = b[7:4];
    add4 a0(.a(wa0), .b(wb0),
     .ci(ci), .co(w0), .s(ws0));
    add4 a1(.a(wa1), .b(wb1),
     .ci(w0), .co(co), .s(ws1));
    assign s = { ws1, ws0 };
endmodule
```

8bit Adder（別の記述）





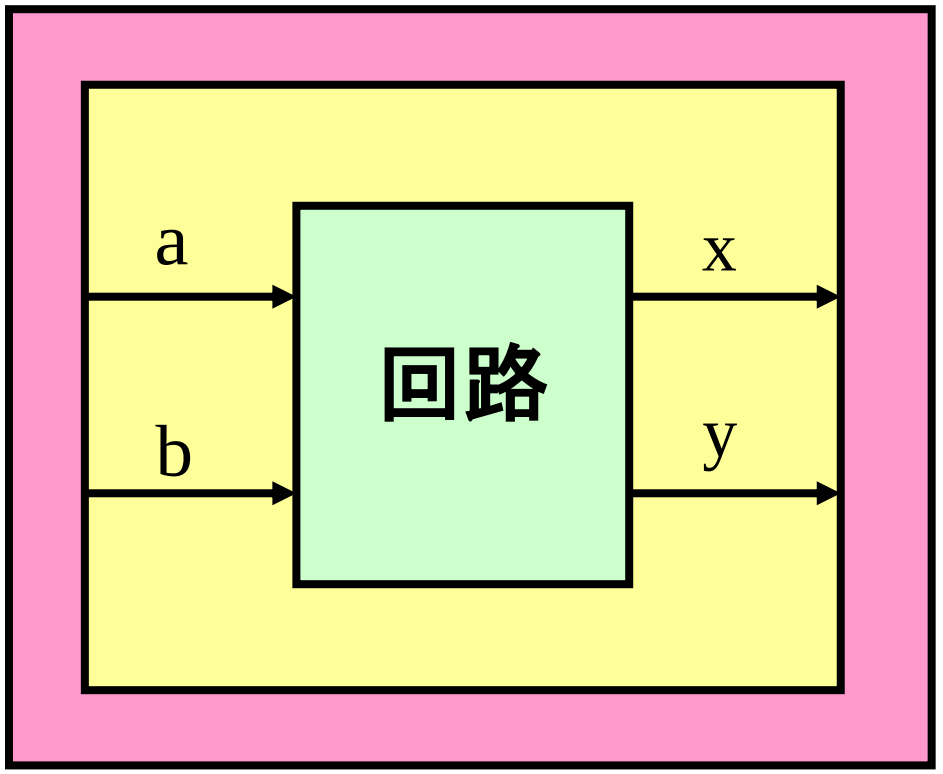

```
module add8(a, b, ci, co, s);
   input  [7:0] a, b;
   input        ci;
   output       co;
   output [7:0] s;
   wire w0;

   add4 a0(.a(a[3:0]), .b(b[3:0]), .ci(ci), .co(w0), .s(s[3:0]));
   add4 a1(.a(a[7:4]), .b(b[7:4]), .ci(w0), .co(co), .s(s[7:4]));
endmodule
```

テストベンチ

回路は入力が無ければ動かない！

例えば、電卓や時計は、ボタンを押してみないと

正しく動作するかどうかがわからない



回路に信号を入力して出力を観測

a	b	x	y
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

テストベンチ

(例) 半加算器のテストベンチ(1)

```
`timescale 1ns/1ns

module test_half_add();
    reg  a, b;
    wire co, s;

    half_add a0(.a(a), .b(b), .co(co), .s(s));

    initial begin
        a = 1'b0; b = 1'b0;
        #10;
        a = 1'b0; b = 1'b1;
        #10;
        a = 1'b1; b = 1'b0;
        #10;
        a = 1'b1; b = 1'b1;
        #10;
        $finish();
    end

    ...

endmodule
```

単位時間と精度

テストベンチのmodule定義

テストするmoduleへの入力は reg
moduleからの出力はwire で宣言

テストするmoduleの接続

入力信号を定数で順に記述
先頭の数値は信号のビット数
b:2進数、d:10進数、h:16進数

指定した時間が
経過したら
次の入力信号へ

a	b	co	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(例) 半加算器のテストベンチ(2)

```
`timescale 1ns/1ns
module test_half_add();

    ...

    initial begin
        $monitor($time,,"a=%b b=%b co=%b s=%b",
                 a, b, co, s);
    end


    initial begin
        $dumpfile("half_add.vcd");
        $dumpvars(0, test_half_add);
    end
endmodule
```

時間を表示

表示のフォーマット指定
%b:2進数、%d:10進数、%h:16進数

端末への表示

表示する信号を順に記述

波形表示用ファイル名の指定
拡張子はvcd

波形表示の開始時刻(通常は0)

テストベンチのmoduleの名前

コンパイルと実行

```
iverilog src1.v src2.v ... -o out
```

実行ファイル名の指定
省略した場合の
実行ファイル名は「a.out」

ソースファイル名
実行に必要なmoduleが記述されたファイルを（テストベンチも含めて）全て指定
正規表現で指定することも可能

```
./out
```

シミュレーションの実行

```
./out > file.txt
```

リダイレクトでテキストファイルを指定

半加算器の場合

正規表現：カレントディレクトリに
half_add.v と test_half_add.v のみがある場合

```
$ cd half_add
$ iverilog *.v -o half_add
$ ./half_add
VCD info: dumpfile half_add.vcd opened for output.
                   0 a=0 b=0 co=0 s=0
                  30 a=0 b=1 co=0 s=1
                  30 a=1 b=0 co=0 s=1
                  30 a=1 b=1 co=1 s=0
                  40 a=x b=x co=x s=x
$ ./half_add > half_add.txt
$
```

波形表示用ファイルは実行時に生成

実行時にリダイレクトを使用した場合は
実行されるが画面上には何も表示されない

波形の表示

```
gtkwave file.vcd &
```

テストベンチに記述した波形表示用ファイル名を指定
（シミュレーション実行時に生成される）

8bit加算器の場合

```
$ cd add8
$ iverilog *.v -o add8
$ ./add8
$ gtkwave add8.vcd &
$
```

```
gtkwave file.vcd file.sav &
```

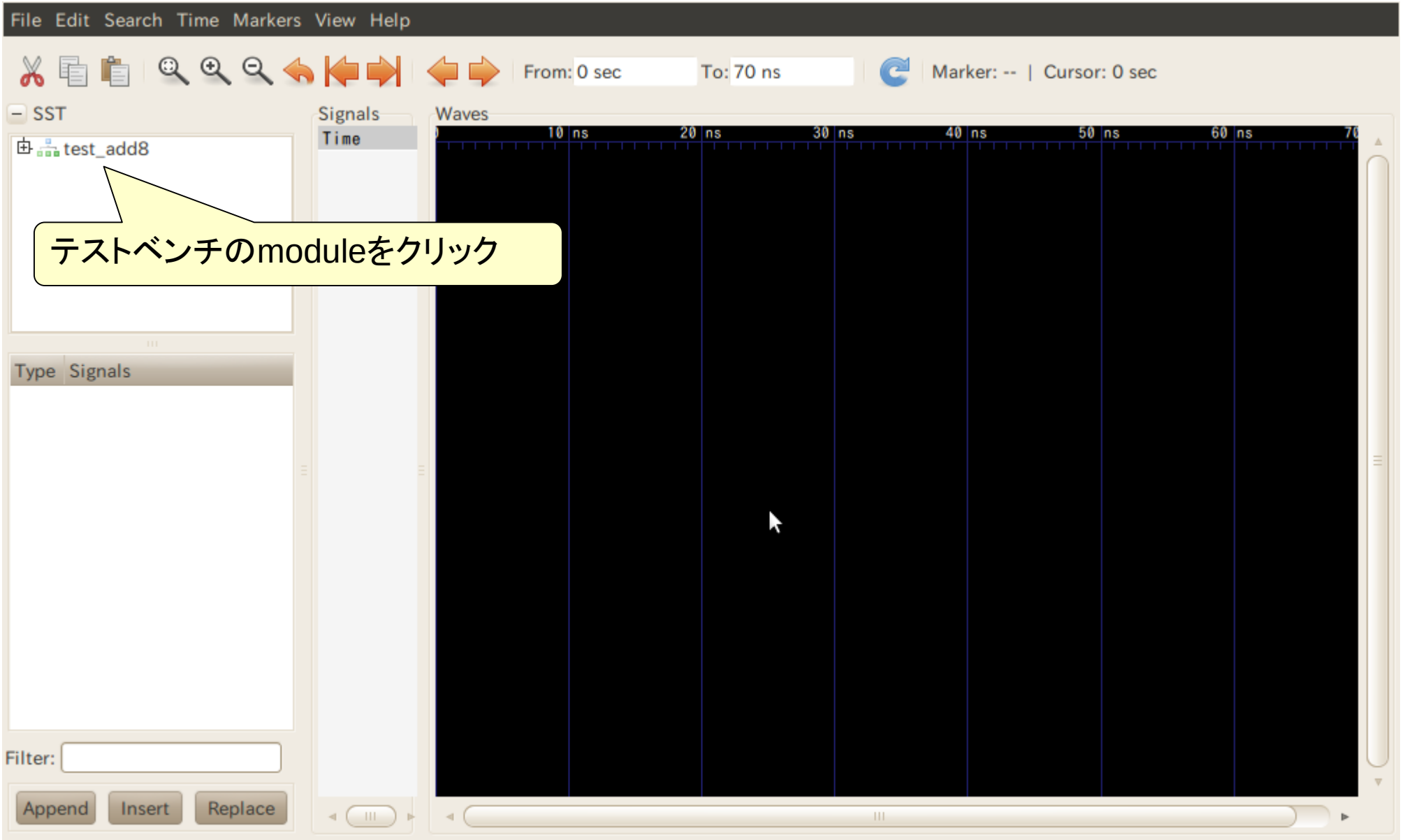
gtkwaveの実行時に保存したファイル名を
起動時に指定すれば保存した状態から再開できる
（表示した信号と順番、表示フォーマットなど）

8bit加算器の場合

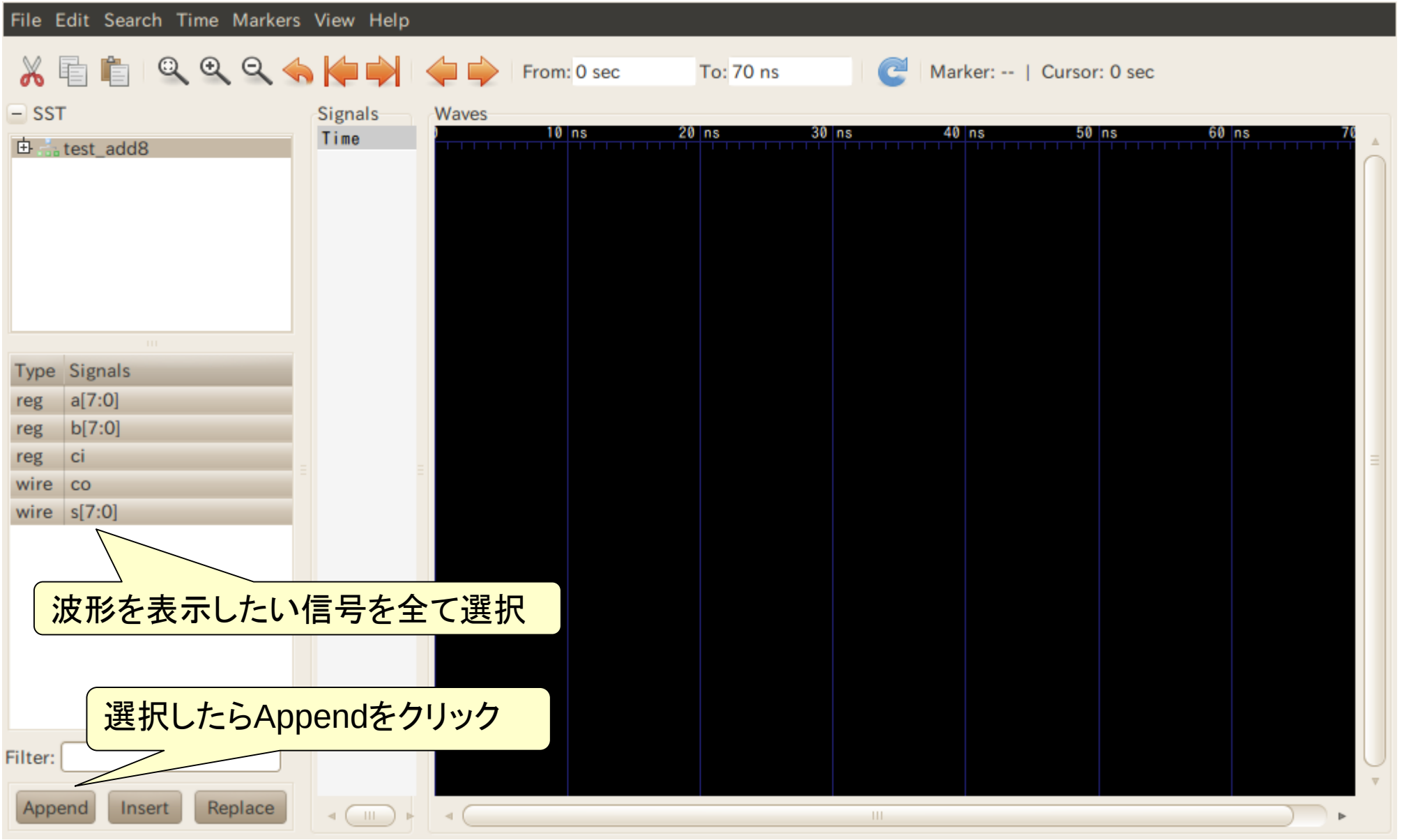
gtkwaveでadd8.savを保存した後

```
$ gtkwave add8.vcd add8.sav &
$
```

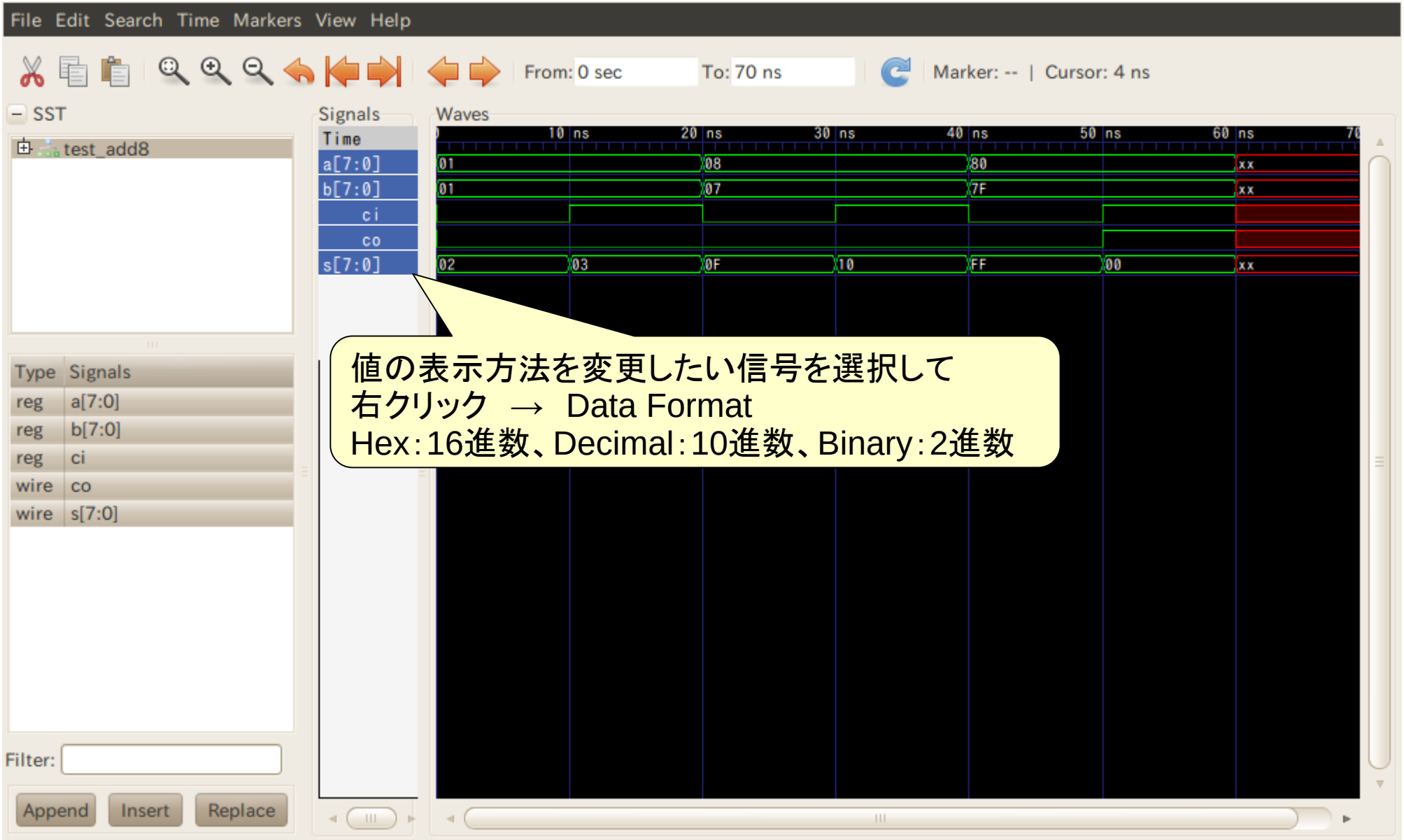
波形の表示


File Edit Search Time Markers View Help
From: 0 sec
To: 70 ns
Marker: -- | Cursor: 0 sec
SST
test_add8
テストベンチのmoduleをクリック
Signals
Time
Waves
10 ns
20 ns
30 ns
40 ns
50 ns
60 ns
Type
Signals
Filter:
Append
Insert
Replace

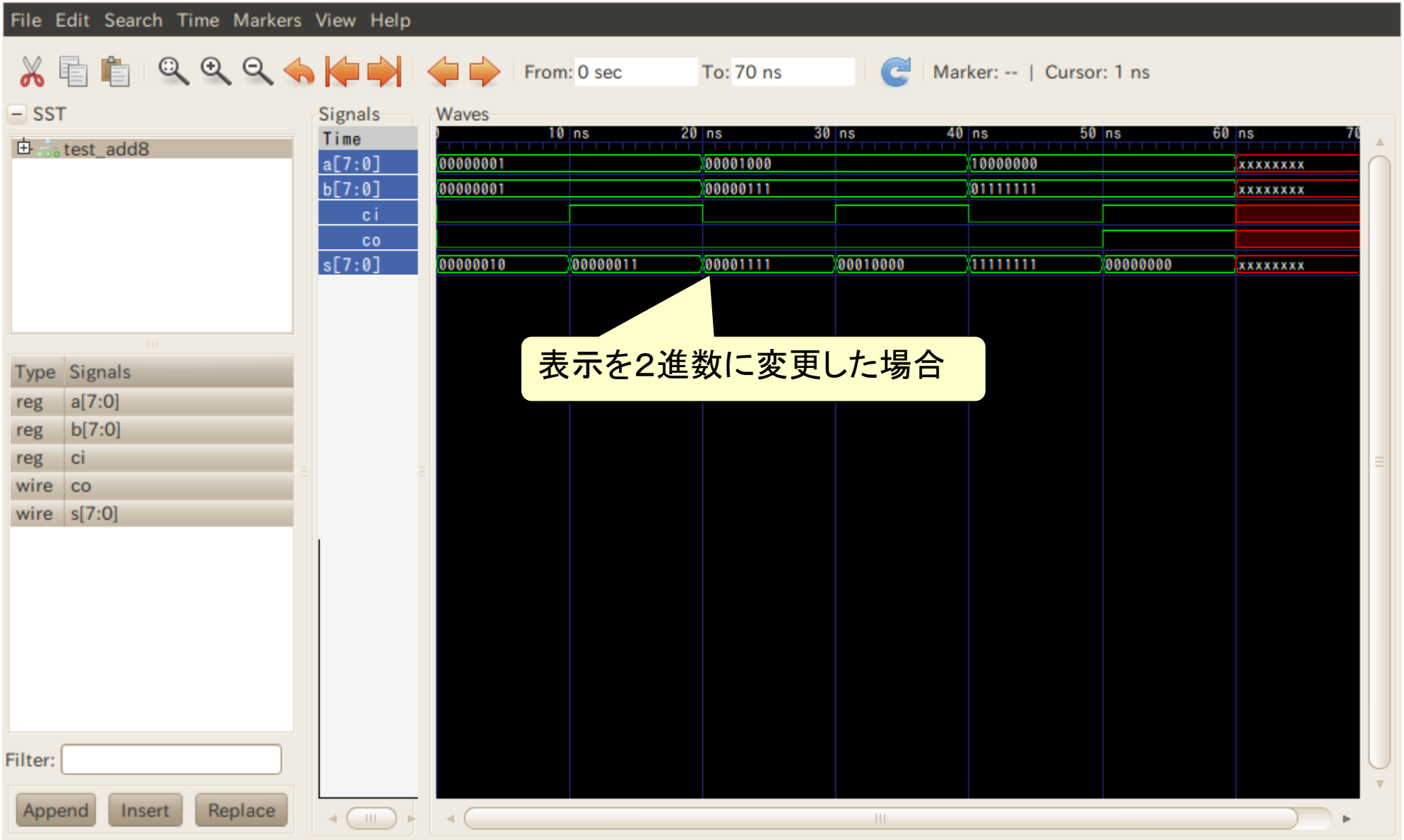
波形の表示


File Edit Search Time Markers View Help
From: 0 sec
To: 70 ns
Marker: -- | Cursor: 0 sec
SST
test_add8
Signals
Time
Waves
10 ns
20 ns
30 ns
40 ns
50 ns
60 ns
Type Signals
reg a[7:0]
reg b[7:0]
reg ci
wire co
wire s[7:0]
波形を表示したい信号を全て選択
選択したらAppendをクリック
Filter:
Append
Insert
Replace

波形の表示


File Edit Search Time Markers View Help
From: 0 sec
To: 70 ns
Marker: -- | Cursor: 4 ns
SST
test_add8
Type Signals
reg a[7:0]
reg b[7:0]
reg ci
wire co
wire s[7:0]
Filter:
Append Insert Replace
Signals
Waves
Time
a[7:0]
b[7:0]
ci
co
s[7:0]
10 ns 20 ns 30 ns 40 ns 50 ns 60 ns
01 08 80 xx
01 07 7F xx
02 03 0F 10 FF 00 xx
値の表示方法を変更したい信号を選択して
右クリック → Data Format
Hex：16進数、Decimal：10進数、Binary：2進数

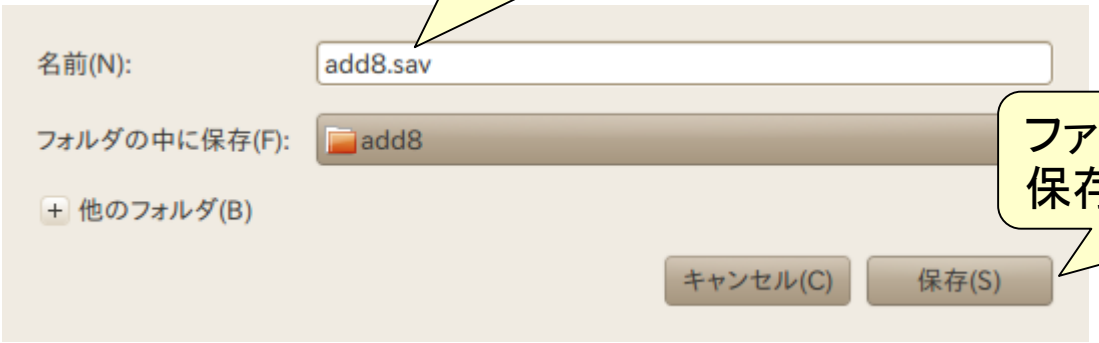
波形の表示


File Edit Search Time Markers View Help
From: 0 sec
To: 70 ns
Marker: -- | Cursor: 1 ns
SST
test_add8
Type Signals
reg a[7:0]
reg b[7:0]
reg ci
wire co
wire s[7:0]
Filter:
Append
Insert
Replace
Signals
Time
a[7:0]
b[7:0]
ci
co
s[7:0]
Waves
10 ns
20 ns
30 ns
40 ns
50 ns
60 ns
00000001
00001000
10000000
xxxxxxxx
00000001
00000111
01111111
xxxxxxxx
00000010
00000011
00001111
00010000
11111111
00000000
xxxxxxxx
表示を2進数に変更した場合

表示結果の保存

File → Write Save File As

ファイル名を指定
拡張子はsav



ファイル名を指定したら
保存をクリック

```
gtkwave file.vcd file.sav &
```

起動時にファイル名を指定すれば
保存した状態から再開できる

```
$ gtkwave add8.vcd add8.sav &
$
```

画像ファイルの作成

File → Print To File

用紙サイズはA4を指定

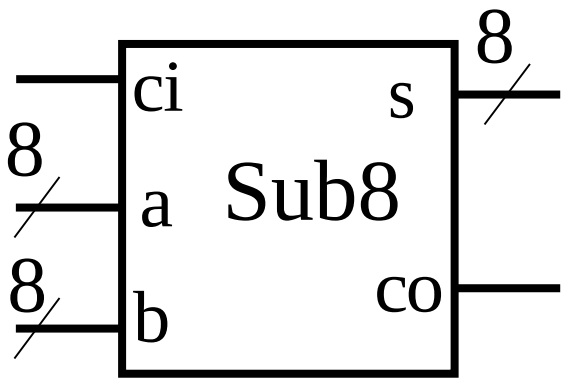
画像形式を指定
PDFやPSなど

指定が終わったらSelect Output Fileをクリック

ファイル名を指定

ファイル名を指定したら
保存をクリック

Subtractor（減算器）



2の補数
d を加算して **0** になる値
→ 負の数 **–d**

正の数	1	2	3	4
d	00000001	00000010	00000011	00000100
$\sim d$	11111110	11111101	11111100	11111011
$-d$	11111111	11111110	11111101	11111100
負の数	–1	–2	–3	–4

減算 = 負の数の加算

$$a - b = a + (-b)$$
$$= a + (\overline{b} + 1)$$

```
0     1
  0111 0110
- 0010 1100
-----------
  0100 1010

1 111   1   1
  0111 0110
+ 1101 0011
-----------
  0100 1010
```

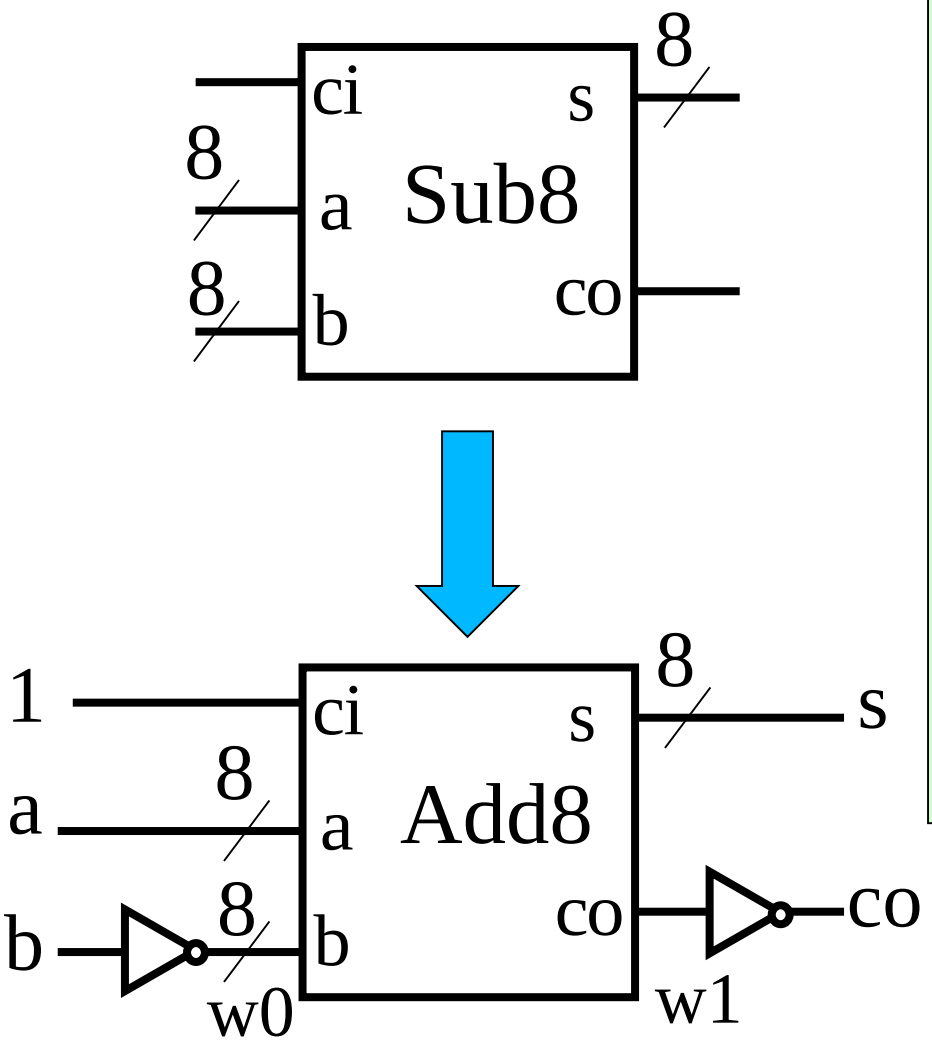
```
1     1
  0111 0110
- 1010 1100
-----------
  1100 1010

0 111   1   1
  0111 0110
+ 0101 0011
-----------
  1100 1010
```

符号の反転
d のビット反転 + **1**

加算器の桁上げ（キャリー）入力 = **1**

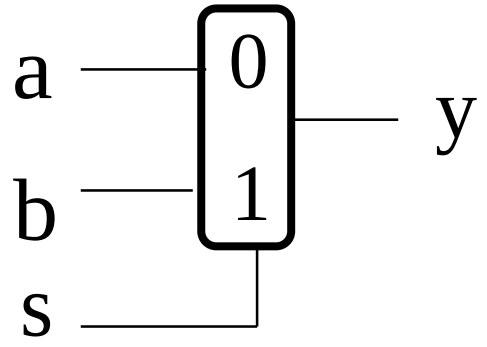
加算器のキャリー出力の**反転** = **減算器の桁借り（ボロー）出力**

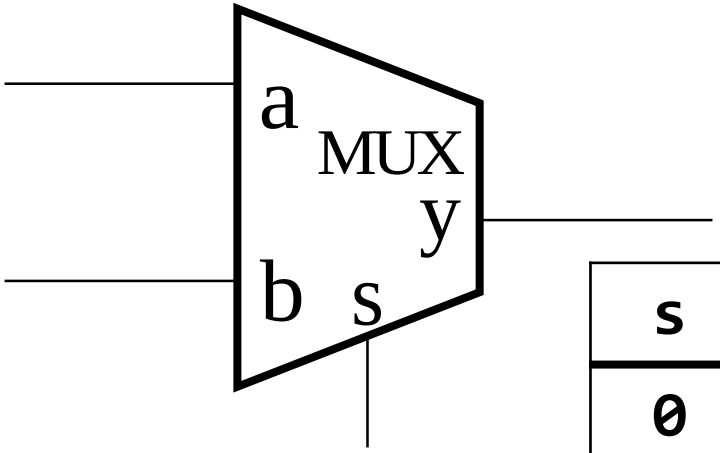
8bit Subtractor（減算器）




```
/* 8ビット減算器のmoduleの定義 */
module sub8(a, b, ci, co, s);
    /* input port */
    input  [7:0] a, b;
    input        ci;
    /* output port */
    output       co;
    output [7:0] s;
    /* wire */
    wire         w0;

    add8 a0(.a(a), .b(~b),
        .ci(ci), .co(w1), .s(s));

    assign co = ~w1;
endmodule
```

1bit 2入力
マルチプレクサ





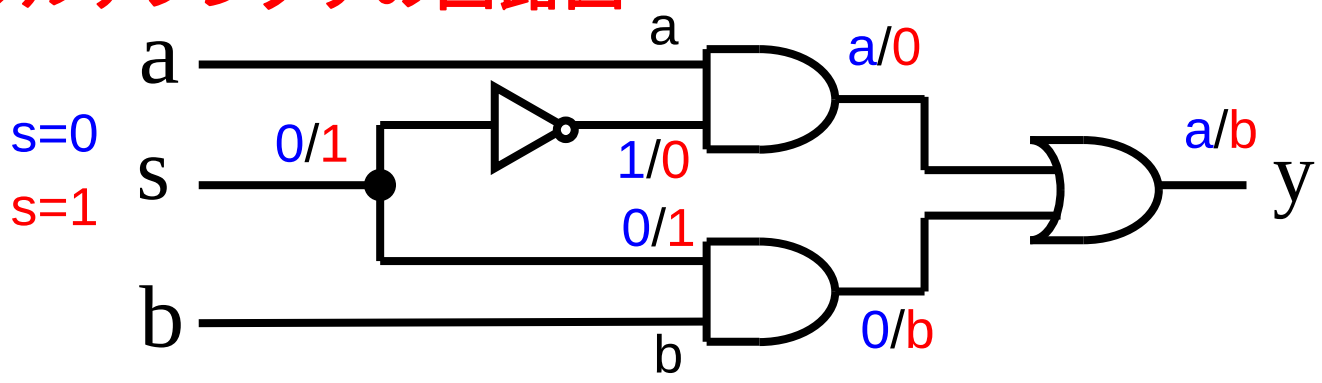
s	y
0	a
1	b

2つの1bitの入力の中から

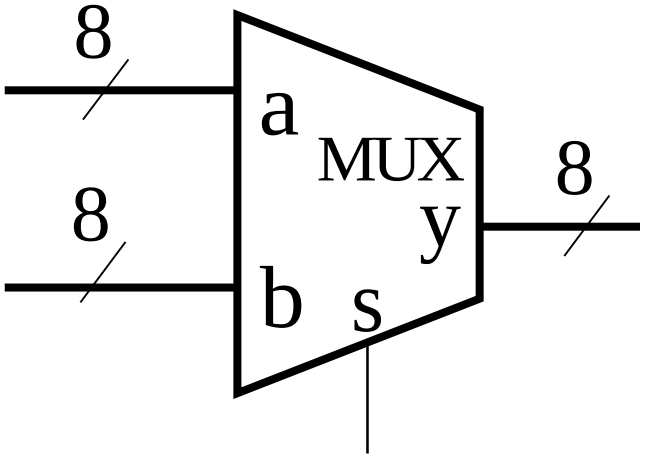
1つを選択して出力

```
module mux1(a, b, s, y);
  input  a, b;
  input  s;
  output y;

  assign y = (a & ~s) | (b & s);
endmodule
```

1bitマルチプレクサの回路図



8bit 2入力マルチプレクサ

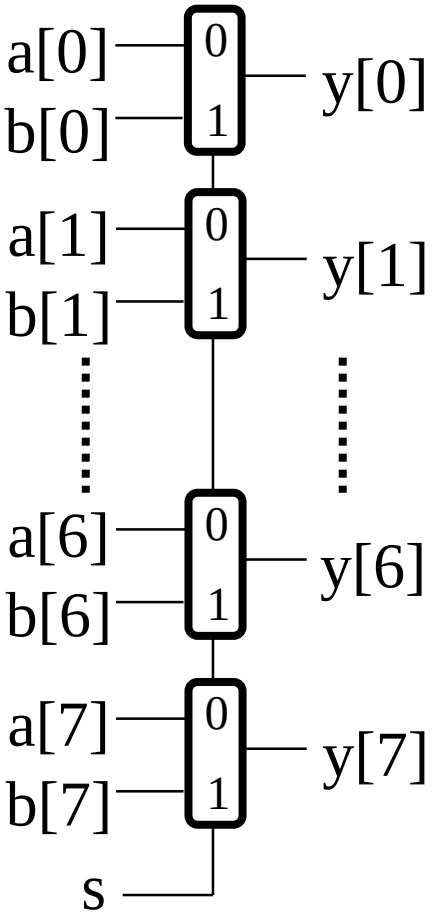


s	y
0	a
1	b

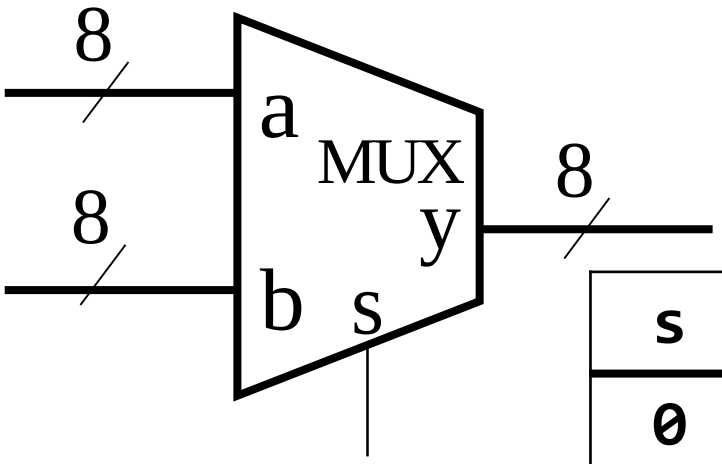
2つの8bitの入力の中から1つを選択して出力

```
module mux8(a, b, s, y);
  input  [7:0] a, b;
  input        s;
  output [7:0] y;

  mux1 m0(.a(a[0]), .b(b[0]), .s(s), .y(y[0]));
  mux1 m1(.a(a[1]), .b(b[1]), .s(s), .y(y[1]));
  mux1 m2(.a(a[2]), .b(b[2]), .s(s), .y(y[2]));
  mux1 m3(.a(a[3]), .b(b[3]), .s(s), .y(y[3]));
  mux1 m4(.a(a[4]), .b(b[4]), .s(s), .y(y[4]));
  mux1 m5(.a(a[5]), .b(b[5]), .s(s), .y(y[5]));
  mux1 m6(.a(a[6]), .b(b[6]), .s(s), .y(y[6]));
  mux1 m7(.a(a[7]), .b(b[7]), .s(s), .y(y[7]));
endmodule
```

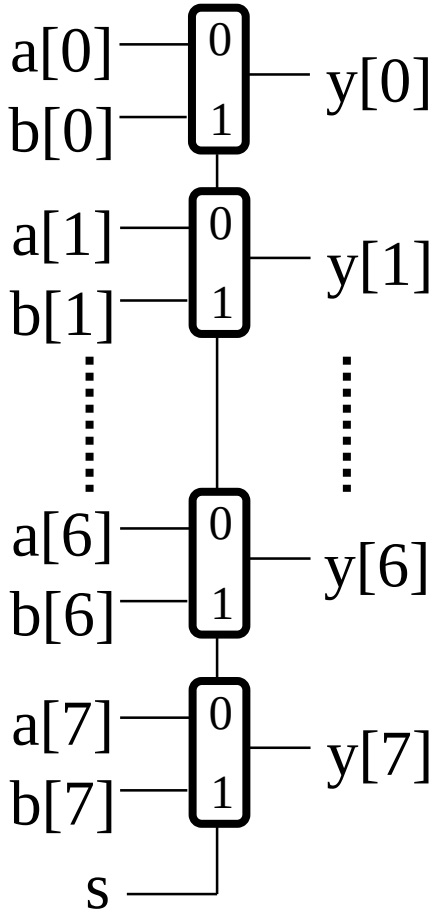
1bitマルチプレクサを8個接続
選択信号sは共通



8bit 2入力 マルチプレクサ（別の記述）



s	y
0	a
1	b



N{w}
信号wをN本コピー

```
module mux8(a, b, s, y);
  input  [7:0] a, b;
  input        s;
  output [7:0] y;

  assign y = (a & { 8{~s} } ) | ( b & { 8{s} } );
endmodule
```

{8{s}} は以下と同じ
{s,s,s,s,s,s,s,s}

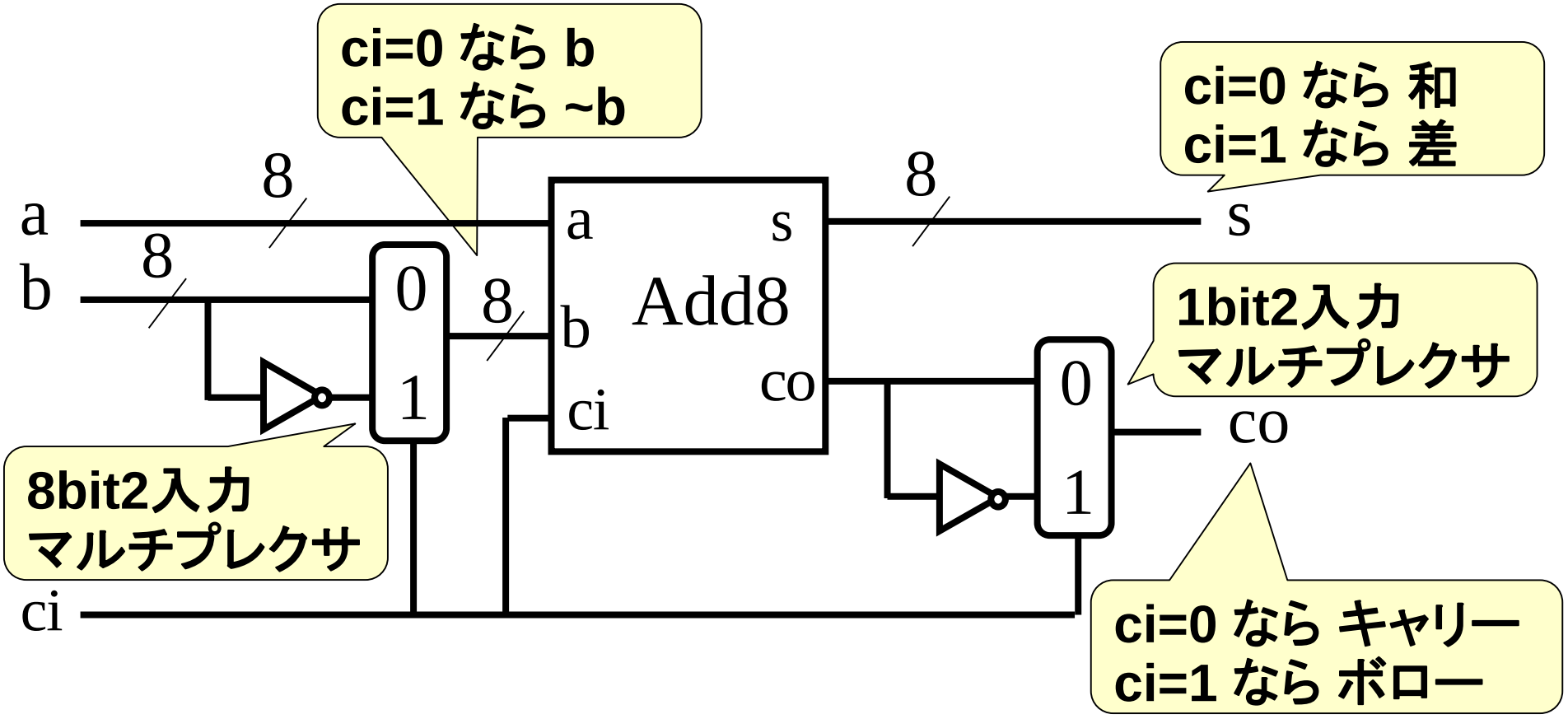
ビットごとの論理和

ビットごとの論理積

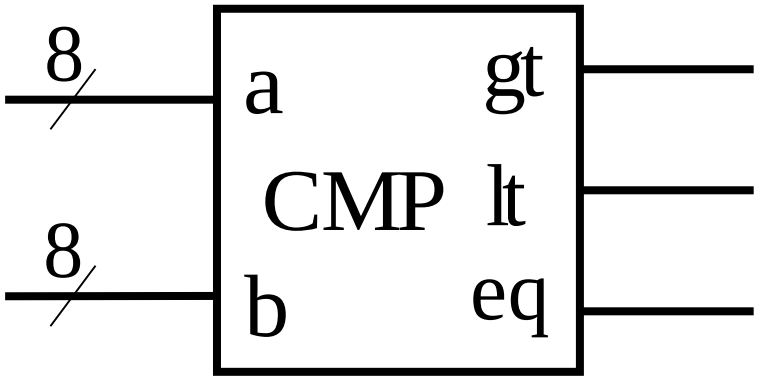
課題(1)

加算器とマルチプレクサを用いて

8bit加減算器を設計せよ



Comparator（比較器）



入力された値の
大小を比較して結果を出力

	gt	lt	eq
1	a > b (a - b > 0)	a < b (a - b < 0)	a == b (a - b == 0)
0	a <= b (a - b <= 0)	a >= b (a - b >= 0)	a != b (a - b != 0)

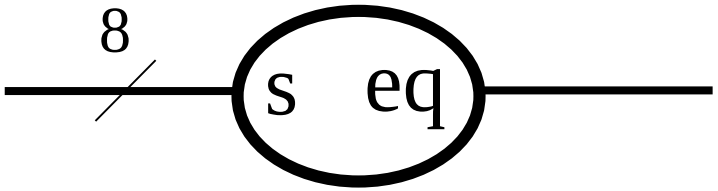
減算結果が零 ➔ a==b ➔ eq=1

減算結果が負 ➔ a<b ➔ lt=1

上記以外の時 ➔ a>b ➔ gt=1

減算結果を用いて
値の大小を比較できる

零判定回路

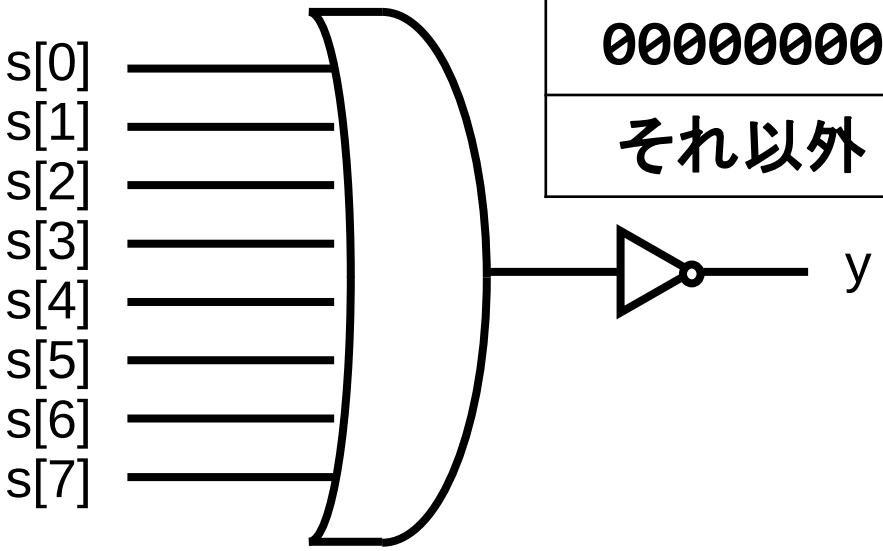



```
...
assign eq = ~|s;
...
```

~|s は以下と同じ
~(s[0]|s[1]|s[2]|...|s[6]|s[7])

8bitの入力(減算結果)が
全て 0 なら 1 を出力
いずれかが 1 なら 0 を出力

s	eq
00000000	1
それ以外	0



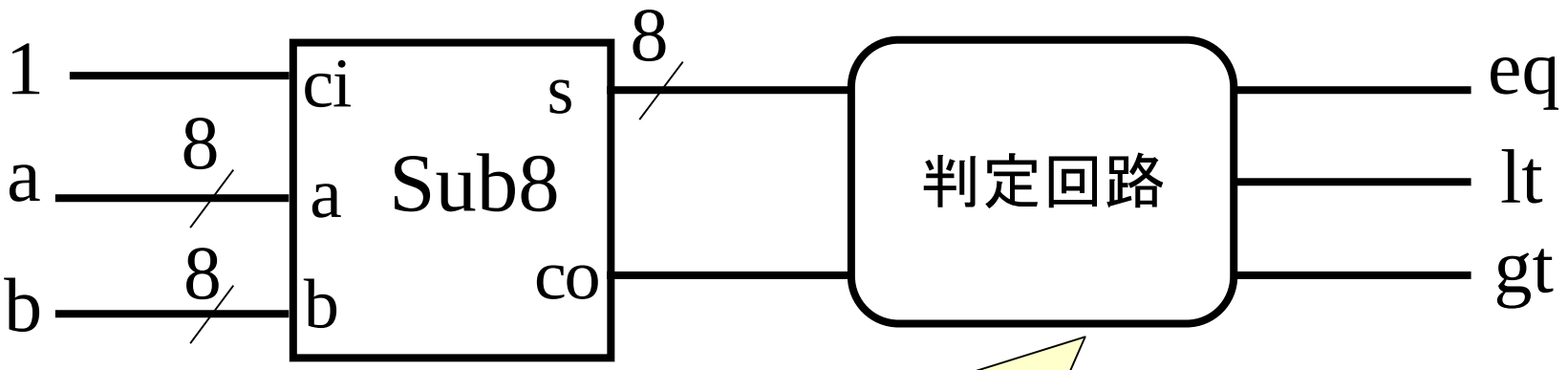
リダクション演算
信号の各ビットに対して
同じ論理演算を実行

&s	どれか0なら0,	全部1なら1
|s	どれか1なら1,	全部0なら0
^s	1が偶数個なら0,	奇数個なら1
~&s	どれか0なら1,	全部1なら0
~|s	どれか1なら0,	全部0なら1
~^s	1が偶数個なら1,	奇数個なら0

課題(2)

減算器に判定回路を追加して

8bit比較器を設計せよ



減算結果が0 ➔ eq=1

ボローが1 ➔ lt=1

上記以外の時 ➔ gt=1

符号なし数の比較

```
1000 0000 (128) = 1000 0000 (128)
0000 0001 (  1) < 1000 0000 (128)
1111 1111 (255) > 0111 1111 (127)
```

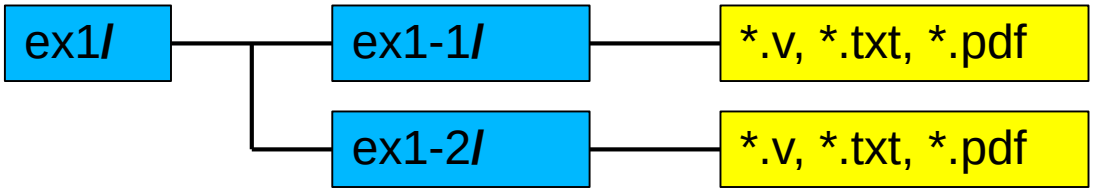
課題

以下の回路のverilogソースファイル（テストベンチを含む）を作成し、シミュレーションを実行するとともに波形を表示せよ。

（1）8bit加減算器　（提出ファイル：ex1-1.tar.gz）

（2）8bit比較器　　（提出ファイル：ex1-2.tar.gz）

各課題について提出すべきファイルを
tar.gz形式の圧縮アーカイブファイルにまとめて
それぞれWeb上から提出せよ




```
cd ex1
tar zcvf ex1-1.tar.gz ex1-1/
tar zcvf ex1-2.tar.gz ex1-2/
```

提出すべきファイル

- **ex1-1/**
  - addsub8.v （加減算器のソースファイル）
    - サブモジュールを使う場合はそのファイルも含めて提出
  - test_addsub8.v （テストベンチのソースファイル）
  - addsub8.txt （iverilog の実行結果をリダイレクトして作成）
  - addsub8.pdf （gtkwaveから作成）
- **ex1-2/**
  - cmp8.v （比較器のソースファイル）
    - サブモジュールを使う場合はそのファイルも含めて提出
  - test_cmp8.v （テストベンチのソースファイル）
  - cmp8.txt （iverilog の実行結果をリダイレクトして作成）
  - cmp8.pdf （gtkwaveから作成）