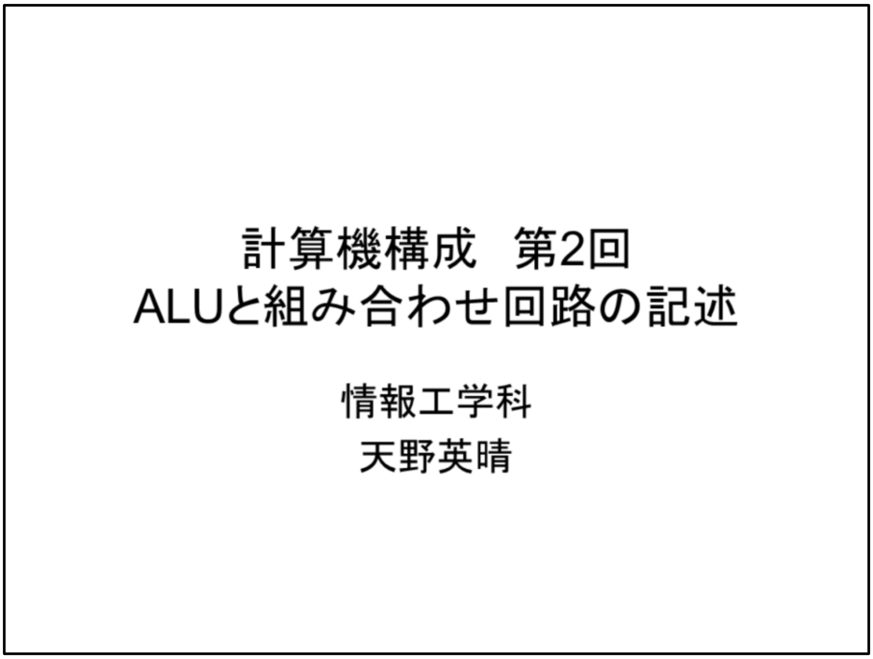
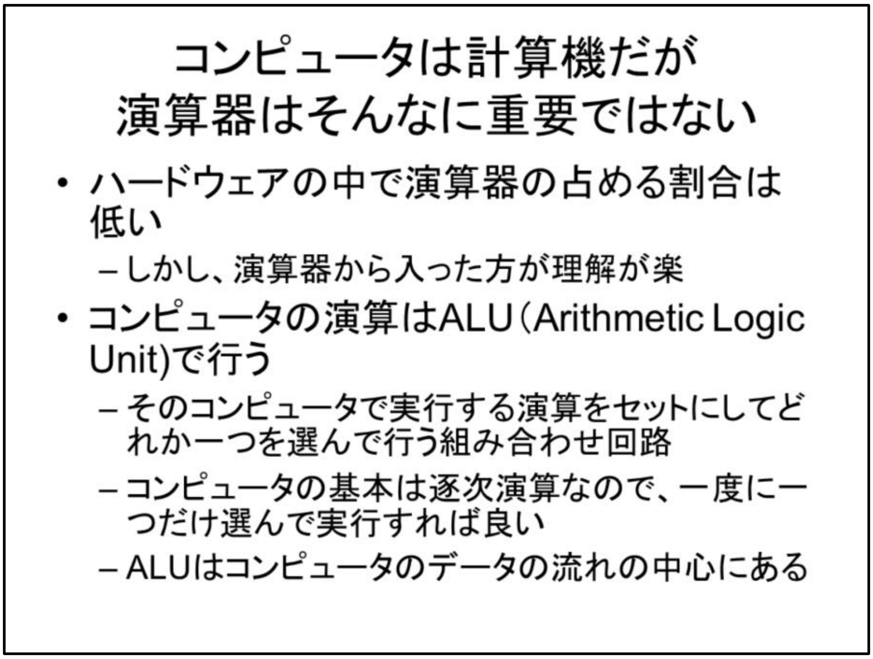


計算機構成　第2回
ALUと組み合わせ回路の記述

情報工学科

天野英晴

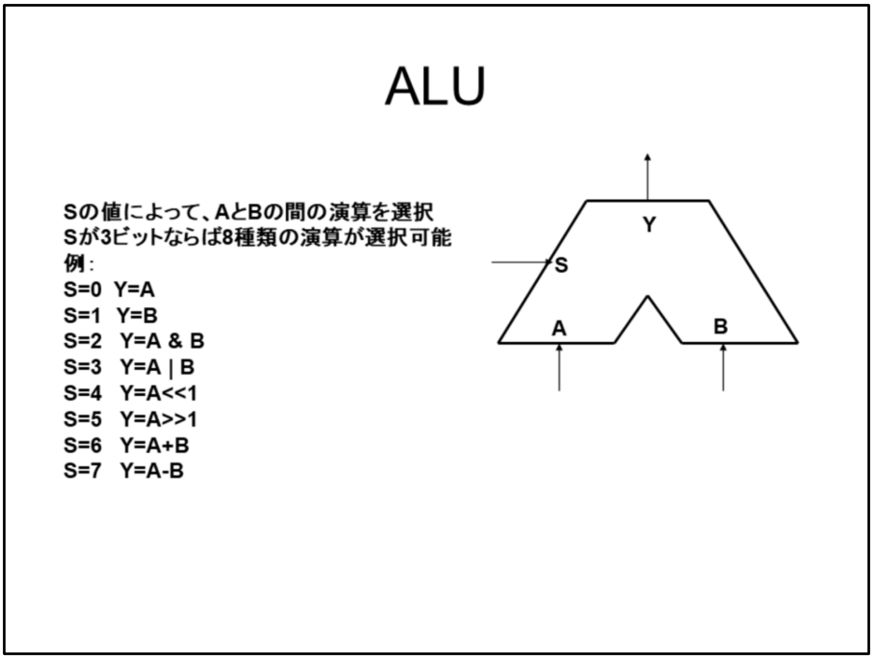
今回はALUと組み合わせ回路の記述を学びます。



コンピュータは計算機だが演算器はそんなに重要ではない

- ハードウェアの中で演算器の占める割合は低い
  - しかし、演算器から入った方が理解が楽
- コンピュータの演算はALU（Arithmetic Logic Unit)で行う
  - そのコンピュータで実行する演算をセットにしてどれか一つを選んで行う組み合わせ回路
  - コンピュータの基本は逐次演算なので、一度に一つだけ選んで実行すれば良い
  - ALUはコンピュータのデータの流れの中心にある

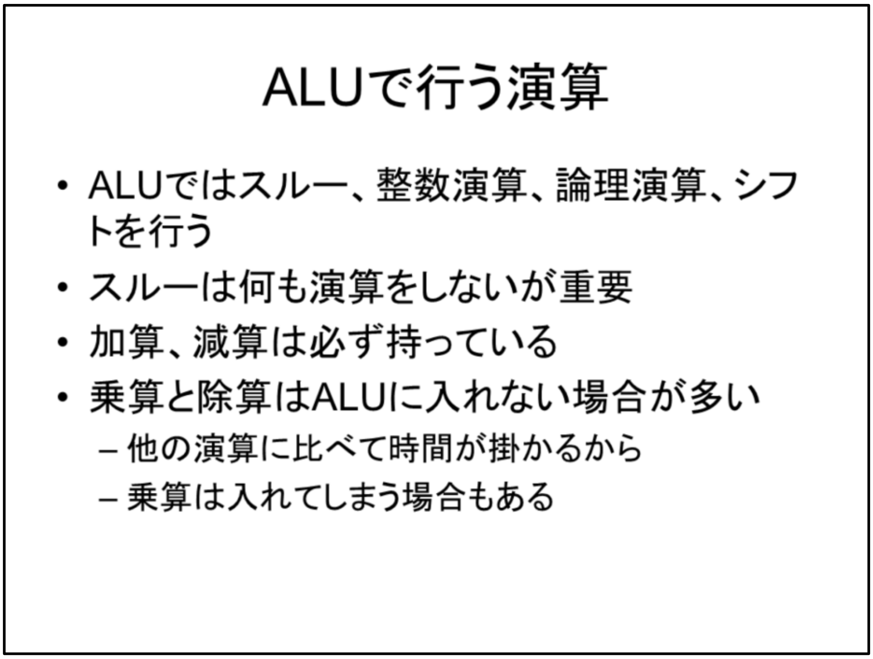
コンピュータは計算機という訳語で呼ばれますが、実際には演算処理ばかりやっているわけではありません。コンピュータのハードウェアの中で演算器の占める割合はそんなに大きくありません。しかし、この授業では演算器から入ります。この方が理解が楽なのです。コンピュータは様々な種類の演算処理を行いますが、これはALU（Arithmetic Logic Unit）というハードウェアで行われます。このALUは、コンピュータで実行する演算をセットにして、それから一つを選んで実行する組み合わせ回路です。なぜ、このような構成にするのでしょう？たくさん演算装置を持っているならば、これを同時に使えばもっと効率が上がるのではないでしょうか？もちろん、こうやって性能を上げる手もあるのですが、本質的にコンピュータの基本は逐次演算、つまり一つずつ命令を実行します。基本は一度に一つの命令を実行するので、一度に一つだけ演算ができれば良く、したがってこのように固めて一つの場所に入れておいて、これから一つを選んで実行するのが合理的です。すべての演算がまとまっているので、演算をしようと思ったらALUを通して行うしかなく、このためALUはコンピュータのデータの流れの中心に置かれます。



ALU

Sの値によって、AとBの間の演算を選択
Sが3ビットならば8種類の演算が選択可能
例：
S=0 Y=A
S=1 Y=B
S=2 Y=A & B
S=3 Y=A | B
S=4 Y=A<<1
S=5 Y=A>>1
S=6 Y=A+B
S=7 Y=A-B

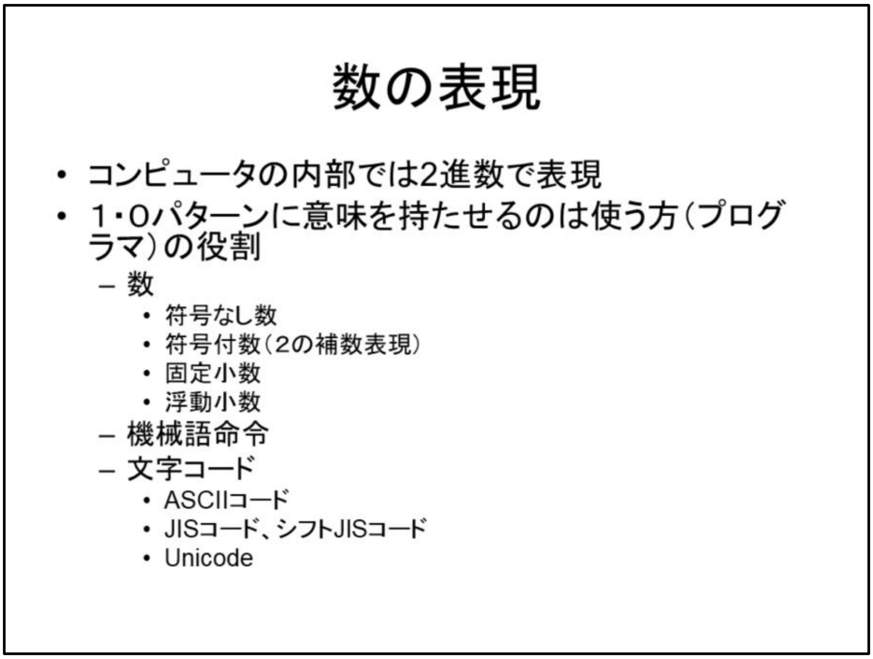
ALUは、この図で示すように、２つの入力A,Bを持ち、これに対して演算を行い、結果をYから出します。どのような演算を行うかを選択入力Sによって選ぶことができます。例えばSが3ビットならば、８種類の演算を選ぶことができます。ここでは大変簡単な演算を８つ選んで行うことができるALUの例を紹介しましょう。



ALUで行う演算

- ALUではスルー、整数演算、論理演算、シフトを行う
- スルーは何も演算をしないが重要
- 加算、減算は必ず持っている
- 乗算と除算はALUに入れない場合が多い
  - 他の演算に比べて時間が掛かるから
  - 乗算は入れてしまう場合もある

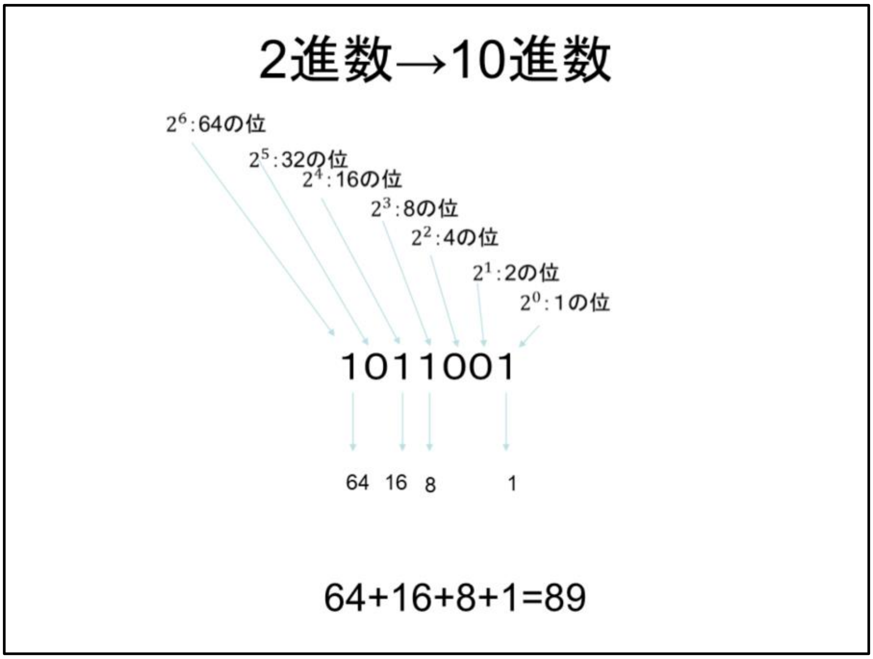
ALUで行う演算は、スルー、整数演算、論理演算、シフト操作に分けられます。スルーは何もしないで入力のA,BをそのままSに出す操作です。例題のALUではS=0,S=1に相当します。何もしないのでバカバカしいようですが、重要で、多くのALUはこの機能を持っています。整数演算のうち加算と減算はほとんどのALUで持っていますが、乗算と除算は持っていない場合が多いです。これは乗算、除算が他の演算に比べて時間が掛かるためです。ALUは全ての演算が短時間でできることが重要です。



数の表現

- コンピュータの内部では2進数で表現
- 1・0パターンに意味を持たせるのは使う方（プログラマ）の役割
  - 数
    - 符号なし数
    - 符号付数（2の補数表現）
    - 固定小数
    - 浮動小数
  - 機械語命令
  - 文字コード
    - ASCIIコード
    - JISコード、シフトJISコード
    - Unicode

以下、ちょっと復習をします。
ご存知と思いますが、コンピュータの内部では、数は1，0から成る2進数で表現します。ここで注意しておきたいのは、1，0のパターンに意味を持たせるのは使う側です。コンピュータ内部には数を識別する機能はないので、同じ1・0のパターンが、解釈の仕様によって、様々な表現の数、機械語命令、文字コードなどを表します。



2進数→10進数

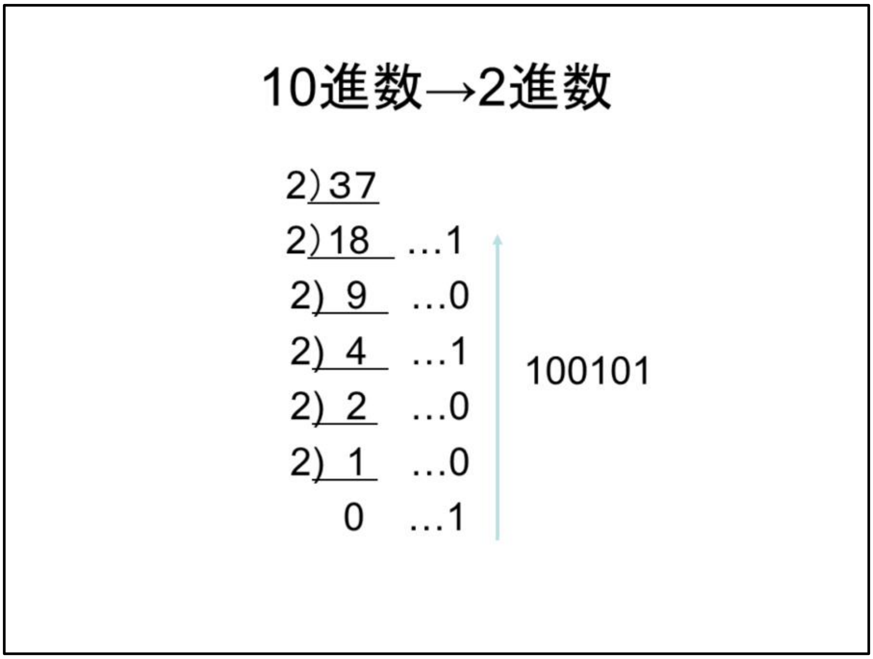
2^6:64の位
2^5:32の位
2^4:16の位
2^3:8の位
2^2:4の位
2^1:2の位
2^0:1の位

1011001

64 16 8 1

64+16+8+1=89

最も簡単なのは、符号なし数です。これは中学で習った2進数そのものです。2進数は、各桁は1か0で、２の0乗の位、2の１乗の位、、、、と順番に桁の重みを表します。
この例では2の0乗の位＝１、２の３乗の位＝８、２の４乗の位＝１６、２の６乗の位＝６４が1なので、全てを足すと８９になります。

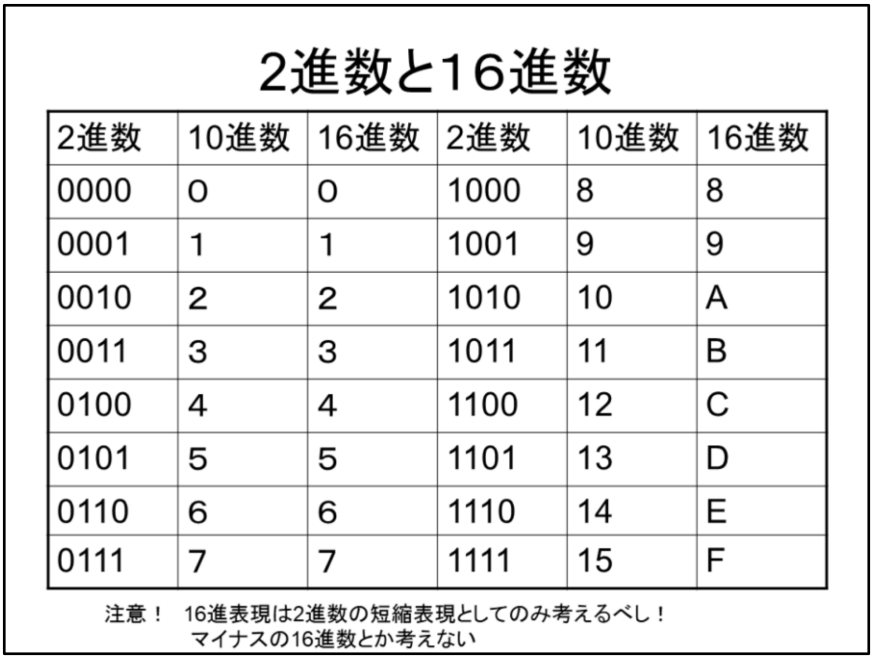


10進数→2進数

2) 37
2) 18 …1
2) 9 …0
2) 4 …1
2) 2 …0
2) 1 …0
0 …1

100101

逆に変換する場合、２で割っていき、余りを記録して行きます。この場合２で6回割ったとき、3回割ったとき、１回目に割ったときが1なので、１００１０１になります。

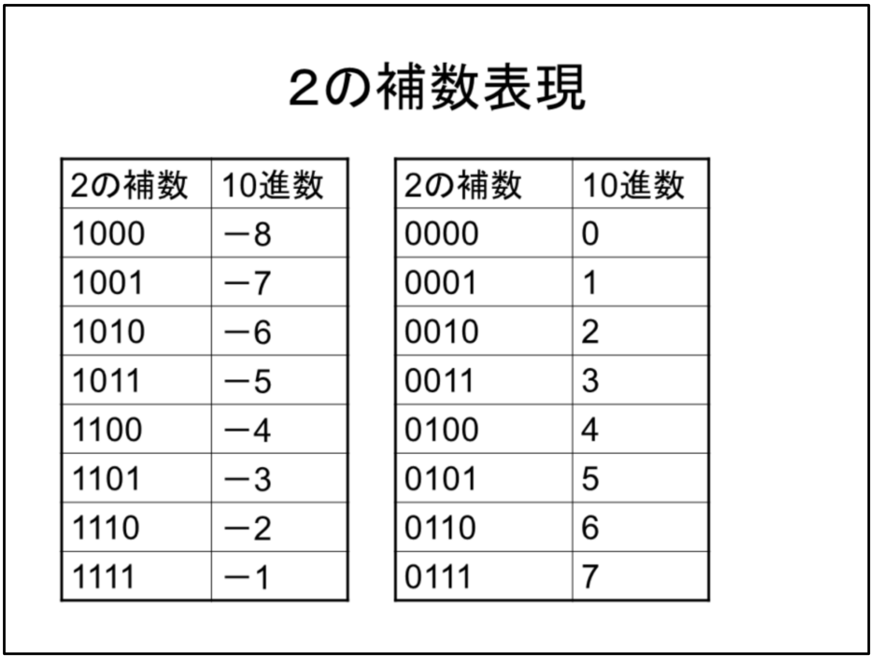


2進数と16進数

2進数	10進数	16進数	2進数	10進数	16進数
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	10	A
0011	3	3	1011	11	B
0100	4	4	1100	12	C
0101	5	5	1101	13	D
0110	6	6	1110	14	E
0111	7	7	1111	15	F

注意！　16進表現は2進数の短縮表現としてのみ考えるべし！
マイナスの16進数とか考えない

２進数は、ちょっとした大きさの数を表すときも、桁数が長くなってしまいます。そこで、これを４桁ずつ区切って読みます。０－９までは普通の１０進数と同じですが、1010は10ではなくてAを使います。以降、B,C,D,E,Fと使います。すなわち16進数は一桁を０－Fで表すのです。ここで注意です。16進数はあくまで2進数の短縮表現として考えましょう。マイナスの16進数というのは定義可能ですが、あまり使いませんし、頭が混乱してしまいます。

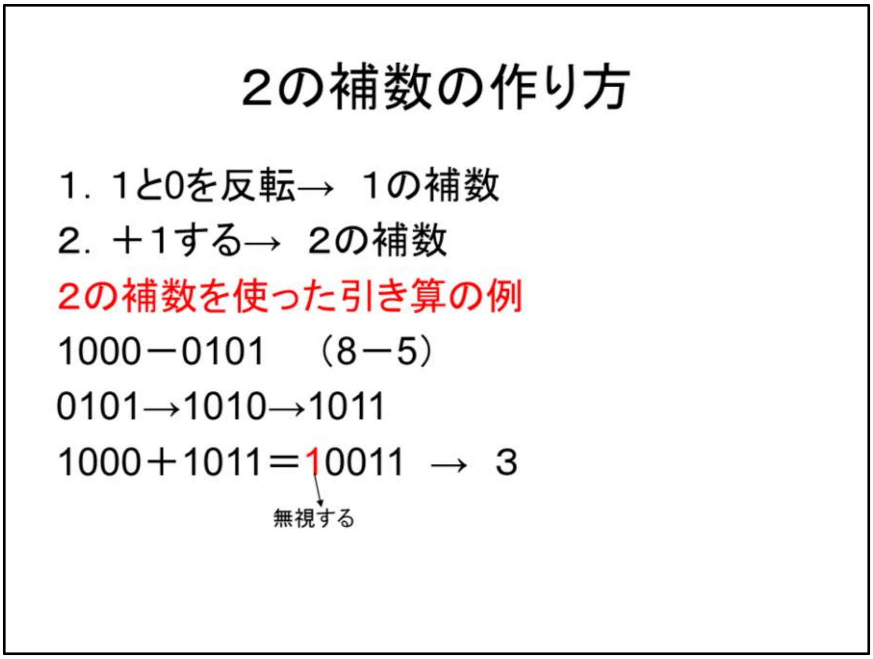


２の補数表現

2の補数	10進数
1000	－8
1001	－7
1010	－6
1011	－5
1100	－4
1101	－3
1110	－2
1111	－1

2の補数	10進数
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

さて、次にマイナスの数を含む符号付き数を考えます。現在のコンピュータでは符号付き数として、2の補数表現を使います。2の補数とは、元の数と足した時に、桁あふれを除いて全て0になる数のことです。例えば、0001（１）の、２の補数は、1111です。1111＋0001＝10000になるからです。0011（３）の、２の補数は、1101です。0011＋1101＝10000になるからです。



２の補数の作り方

１．１と0を反転→　１の補数

２．＋１する→　２の補数

２の補数を使った引き算の例

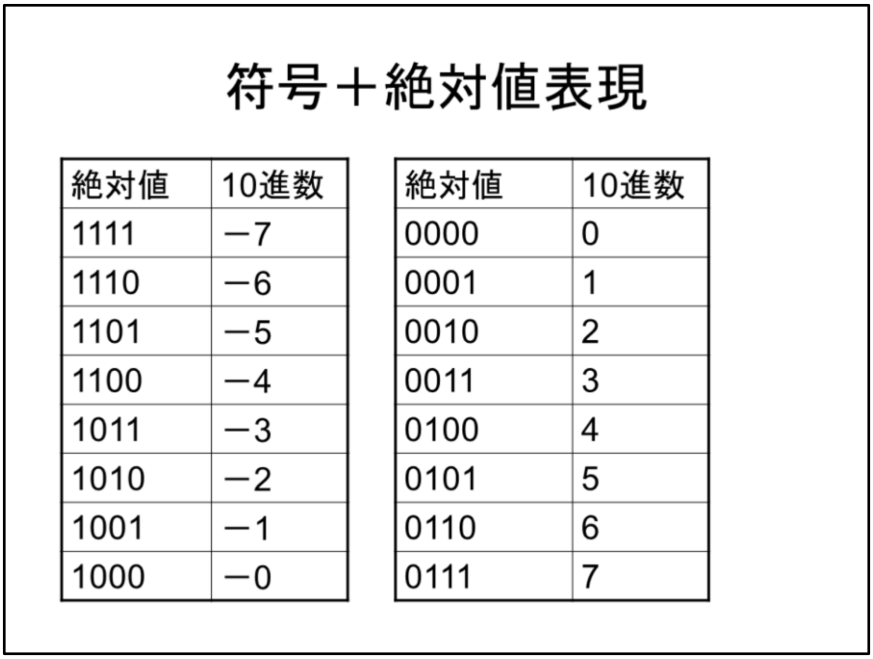
1000－0101　（8－5）

0101→1010→1011

1000＋1011＝10011　→　３

無視する

2の補数の作りかたは簡単です。まず1と0を反転します。これで足して全ての桁が1になる1の補数ができます。これに1を足すと、桁溢れを除いて全ての桁が0になる2の補数になります。２の補数を使うと引き算が簡単にできます。引く数の２の補数を足してやれば良いのです。この例では８－５を計算します。５の「２の補数」は0101→1010→1011です。これを1000（８）に足して桁あふれを無視すれば、0011（３）を求めることができます。

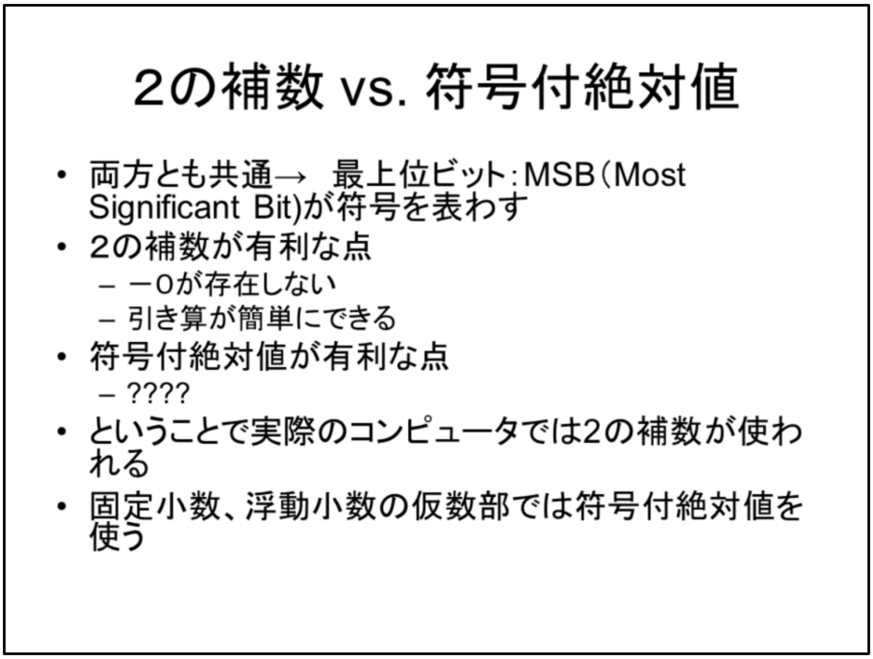


符号＋絶対値表現

絶対値	10進数
1111	－7
1110	－6
1101	－5
1100	－4
1011	－3
1010	－2
1001	－1
1000	－0

絶対値	10進数
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

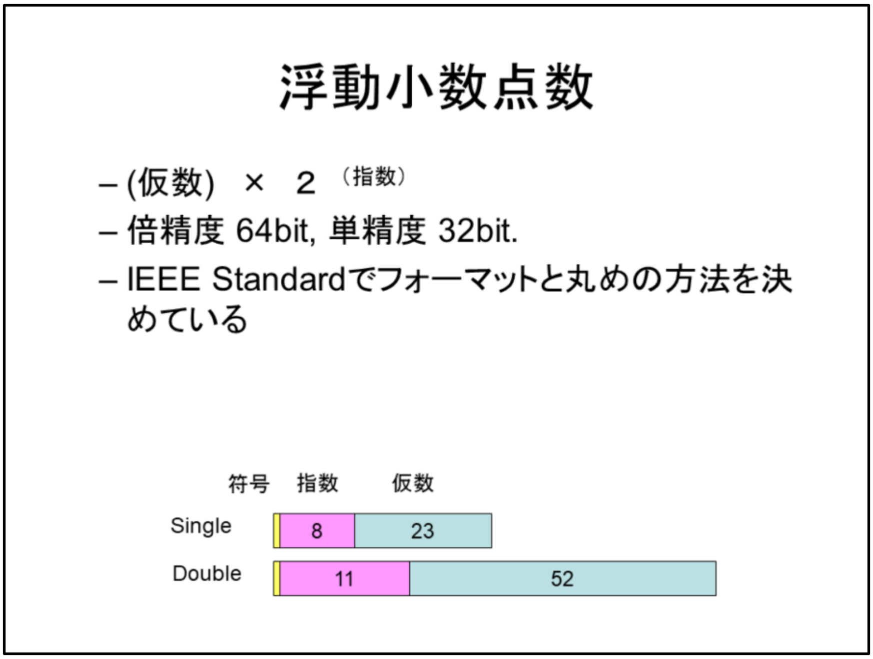
２の補数表現を符号＋絶対値表現と勘違いしている人も居ます。符号＋絶対値表現も浮動小数点表記の一部に使われることもあるのですが、普段使うのは圧倒的に2の補数表現です。



2の補数 vs. 符号付絶対値

- 両方とも共通→　最上位ビット：MSB（Most Significant Bit)が符号を表わす
- 2の補数が有利な点
  - －0が存在しない
  - 引き算が簡単にできる
- 符号付絶対値が有利な点
  - ????
- ということで実際のコンピュータでは2の補数が使われる
- 固定小数、浮動小数の仮数部では符号付絶対値を使う

両者とも、最上位ビット（MSB:Most Significant Bit)が、符号を表し、これが1の場合、負の数になります。しかし、２の補数は－0が存在しない、引き算が簡単にできるという利点があるのに対して、符号付き絶対値表現は特に有利な点が見当たりません。というわけで、実際のコンピュータでは２の補数が使われますが、例えば浮動小数の仮数の表現には符号付き絶対値を使います。

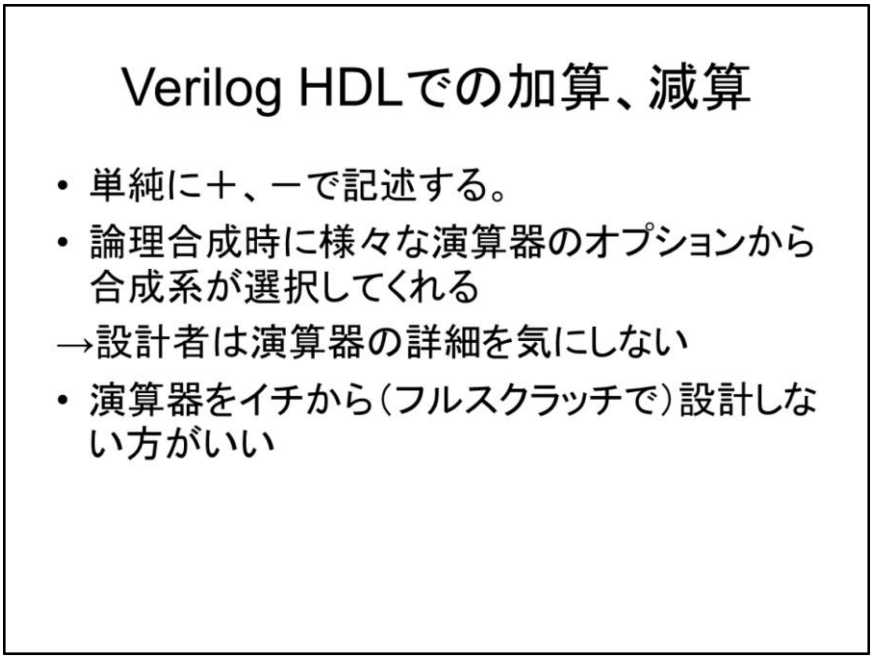


浮動小数点数

- (仮数) × $2^{（指数）}$
- 倍精度 64bit, 単精度 32bit.
- IEEE Standardでフォーマットと丸めの方法を決めている

	符号	指数	仮数
Single		8	23
Double		11	52

浮動小数点数は、表現できる範囲を広げるため、仮数×２の指数乗で表します。これは10進数の指数表現と同じです。この表現は、コンピュータ毎に別なものを使うとデータの交換ができなくなるため、IEEE Standardという国際標準に基づきます。単精度（C言語だとfloat)は32ビット、倍精度（C言語だとdouble)は64ビットで表します。このスライドで示すように指数部と仮数部を分割しています。浮動小数点表記では符号を独立させているため、仮数部は絶対値表現を使いますが、一番上の桁の1を省略します。また指数部は下駄履き(バイアス）表現を使います。また、浮動小数は、答えがぴったり仮数部の桁に収まらない場合にどのようにするか（丸め）という問題があって結構面倒です。ここでは深く突っ込まないことにします。



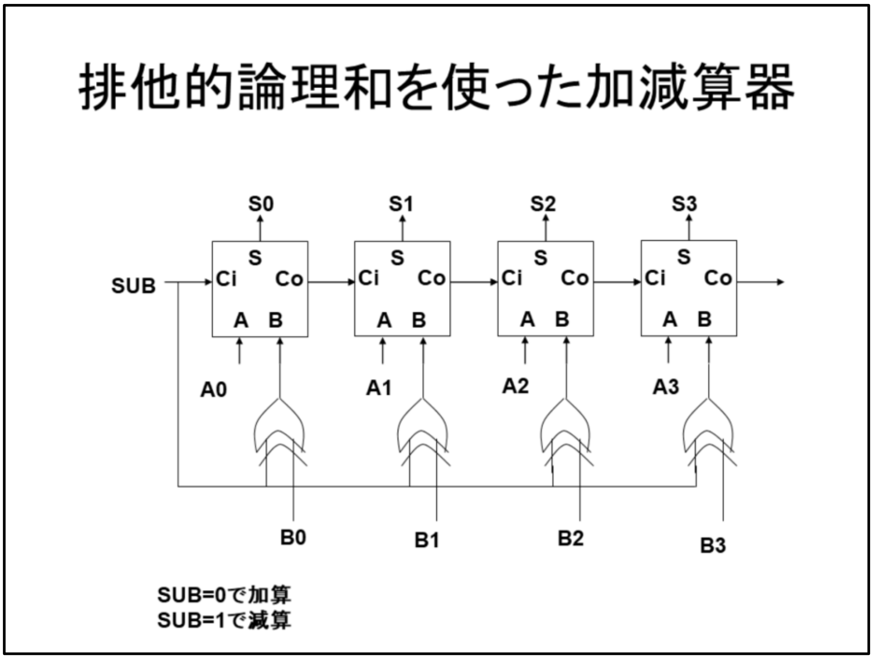
Verilog HDLでの加算、減算

- 単純に＋、－で記述する。
- 論理合成時に様々な演算器のオプションから合成系が選択してくれる

→設計者は演算器の詳細を気にしない

- 演算器をイチから（フルスクラッチで）設計しない方がいい

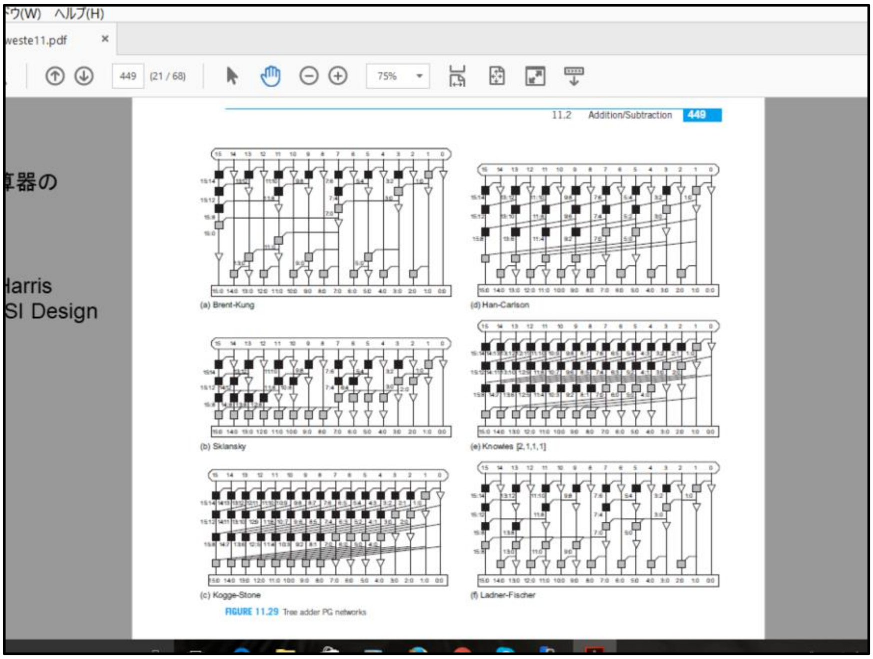
加算、減算はVeirlog上では単純に＋、－と書きます。最近の論理合成用のCADは様々な加算器を持っていて、要求された性能とハードウェア量を考えて、適切なものを選んで使ってくれます。このため、設計者は演算器の詳細について知る必要がありません。演算器は検証が大変なので、一から作ることはしないのがふつうです。



排他的論理和を使った加減算器

SUB=0で加算
SUB=1で減算

Verilogでは＋と書くと加算器、－と書くと減算器がハードウェアモジュールとして想定されます。この辺がソフトウェアとの違いです。単純に＋、－と書いただけで、実際には大きなハードウェアができてしまう場合があります。この図はリプルキャリアダーといって非常に簡単な加算器ですが、加算器にはたくさんの種類があり、速度とハードウェアの大きさが違います。これはVerilogで書いた記述を論理合成するときにCADが判断して決めてくれます。



11.2 Addition/Subtraction 449

(a) Brent-Kung

(b) Sklansky

(c) Kogge-Stone

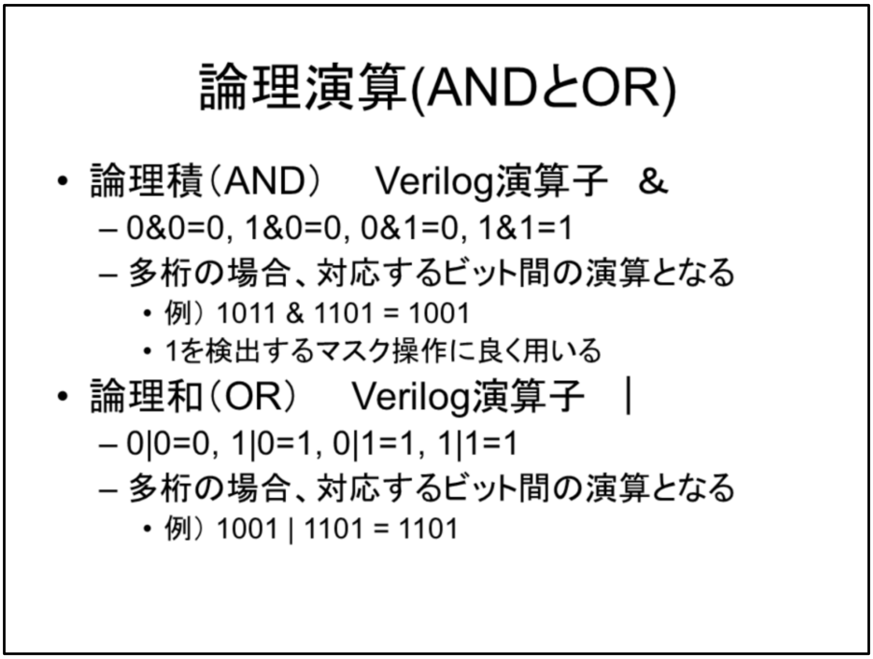
(d) Han-Carlson

(e) Knowles [2,1,1,1]

(f) Ladner-Fischer

FIGURE 11.29 Tree adder PG networks

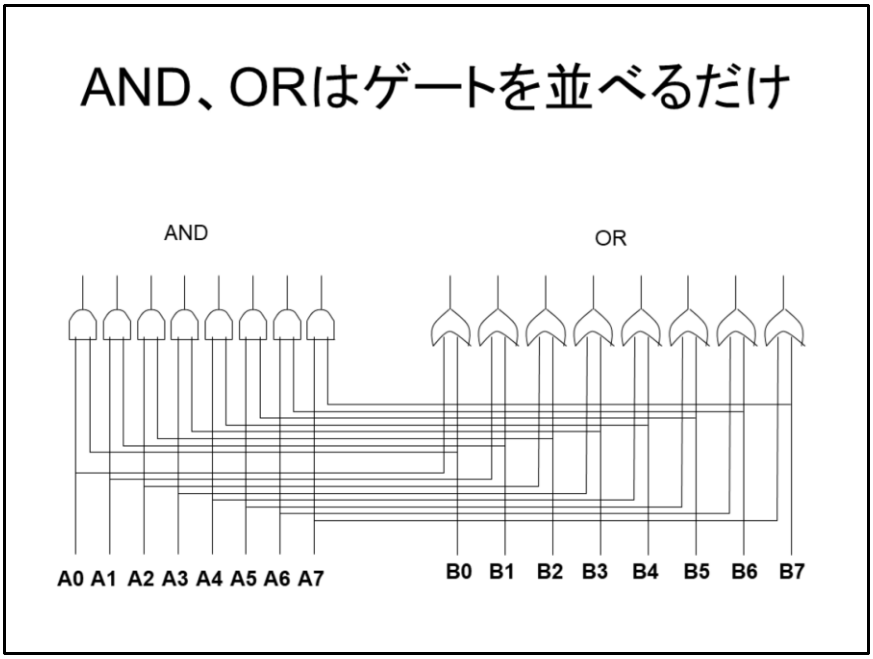
これは様々な配列型加算器の例です。遅延時間の他にも出力数、消費電力などにより様々な性質を持っています。



論理演算(ANDとOR)

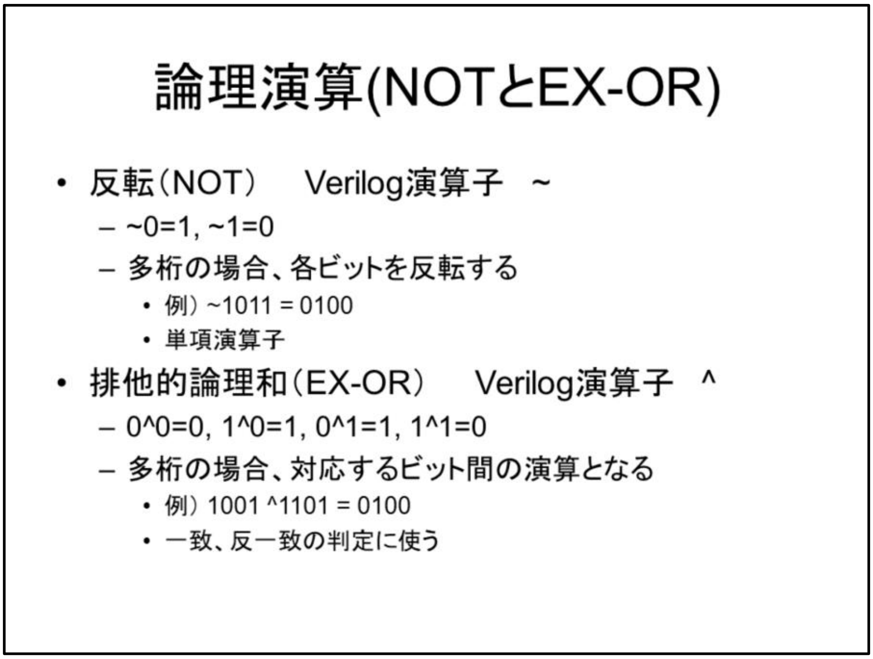
- 論理積（AND）　Verilog演算子　&
  - 0&0=0, 1&0=0, 0&1=0, 1&1=1
  - 多桁の場合、対応するビット間の演算となる
    - 例）1011 & 1101 = 1001
    - 1を検出するマスク操作に良く用いる
- 論理和（OR）　Verilog演算子　|
  - 0|0=0, 1|0=1, 0|1=1, 1|1=1
  - 多桁の場合、対応するビット間の演算となる
    - 例）1001 | 1101 = 1101

では他にはコンピュータではどのような演算を行うのでしょうか？まず論理演算があります。論理積ANDはVerilogの演算子では＆で表し、二つの入力が共に1の時だけ出力が1になります。多桁の2進数の場合、それぞれの桁の論理積をとります。この操作は、マスク操作といってある特定の桁が1かどうか判断するのに使います。これに対して論理和ORはVerilog演算子では｜で表し、どちらか片方の入力が1の時に出力が1になります。ANDと同様、多桁の場合、対応するビットの間のORになります。この二つの演算子はVerilogとCで同じです。



AND、ORはゲートを並べるだけ

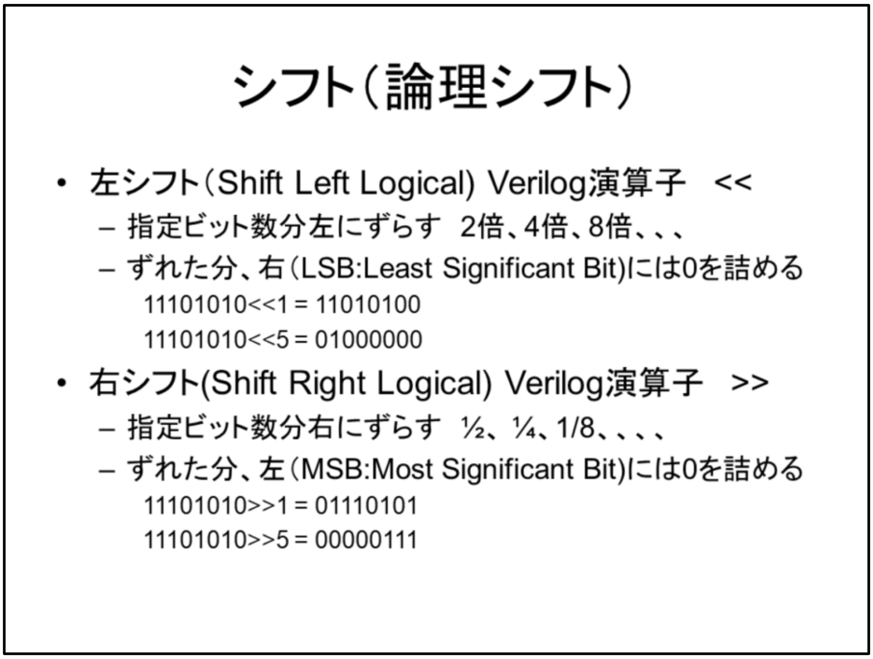
AND、ORは各桁の論理積、論理和なので、単にゲートを並べれば良いです。ハードウェアのコストは高くありません。



論理演算(NOTとEX-OR)

- 反転（NOT）　Verilog演算子　~
  - ~0=1, ~1=0
  - 多桁の場合、各ビットを反転する
    - 例）~1011 = 0100
    - 単項演算子
- 排他的論理和（EX-OR）　Verilog演算子　^
  - 0^0=0, 1^0=1, 0^1=1, 1^1=0
  - 多桁の場合、対応するビット間の演算となる
    - 例）1001 ^1101 = 0100
    - 一致、反一致の判定に使う

反転（NOT)は、入力の1・0をひっくり返して出力します。これは単項演算子といって入力をひとつだけ取ります。Verilog演算子では～で表します。排他的論理和（ExclusiveOR：Ex-OR)は、2つの入力が同じの時には0、違っている場合は1を出力します。Verilog演算子では^です。NOTとEx-ORは、Verilogの演算子がC言語と違っているので注意しましょう。



シフト（論理シフト）

- 左シフト（Shift Left Logical) Verilog演算子　<<
  - 指定ビット数分左にずらす　2倍、4倍、8倍、、、
  - ずれた分、右（LSB:Least Significant Bit)には0を詰める

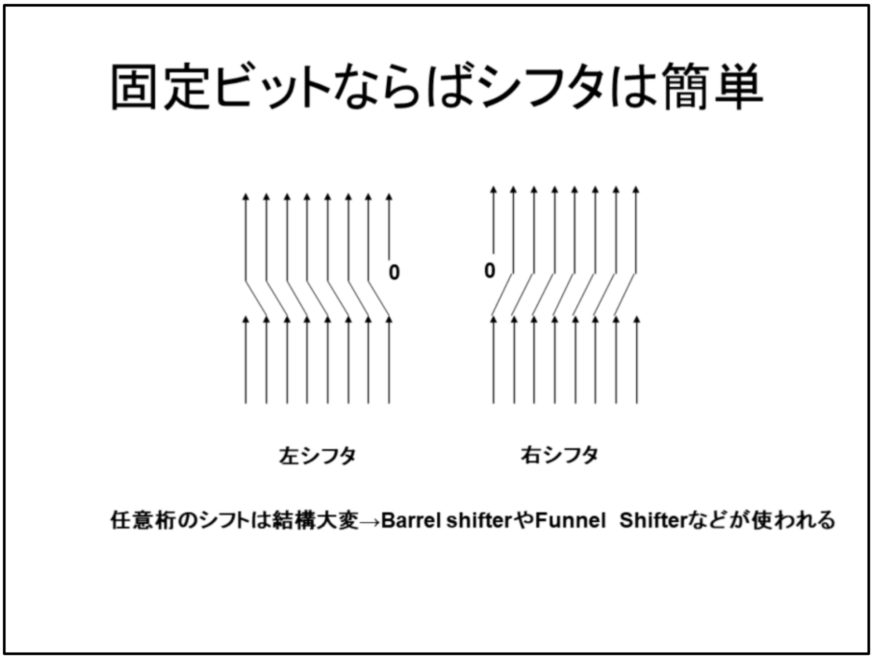
    11101010<<1 = 11010100

    11101010<<5 = 01000000
- 右シフト(Shift Right Logical) Verilog演算子　>>
  - 指定ビット数分右にずらす　½、¼、1/8、、、、
  - ずれた分、左（MSB:Most Significant Bit)には0を詰める

    11101010>>1 = 01110101

    11101010>>5 = 00000111

シフト操作はビットを左右にずらす操作です。左方向にずらす左シフト（Shift Left)は、VerilogでもC言語と同じく＜＜という演算子で表し、ずらす桁数を＜＜の後に記して表します。最も下の桁（Least Significant Bit:LSB)のずらした分には0を詰めるのが普通で、これを論理シフトと呼びます。論理左シフトは、数を2倍、4倍、8倍にすることに相当します。
一方、右方向にずらす右シフト（Shift Right)は、Verilogでは＞＞という演算子で表し、ずらす桁数を同様に後ろに書いて示します。最も上の桁（Most Significant Bit:MSB)のずれた分には0を詰めるので、元の数を１／２、１／４、1／８…にすることに相当します。

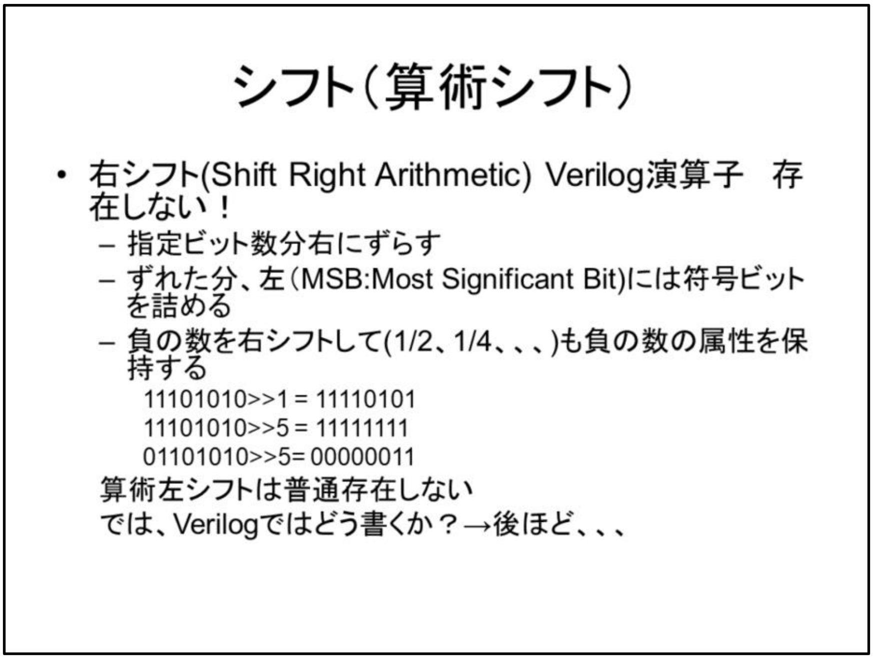


固定ビットならばシフタは簡単

左シフタ　右シフタ

任意桁のシフトは結構大変→Barrel shifterやFunnel Shifterなどが使われる

シフトを行うハードウェアは、固定ビットのシフトならば線の繋ぎ変えで済むので簡単です。しかし任意のビット数のシフトは結構大変で、Barrel ShifterやFunnel Shifterなど専用のハードウェアが考案されています。



シフト（算術シフト）

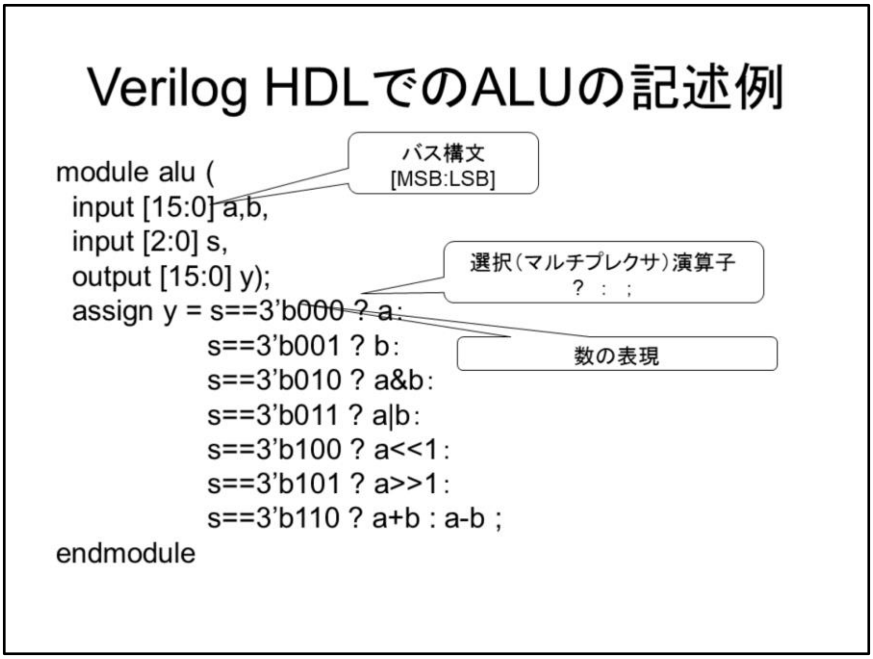
- 右シフト(Shift Right Arithmetic) Verilog演算子　存在しない！
  - 指定ビット数分右にずらす
  - ずれた分、左（MSB:Most Significant Bit)には符号ビットを詰める
  - 負の数を右シフトして(1/2、1/4、、、)も負の数の属性を保持する

    11101010>>1 = 11110101
    11101010>>5 = 11111111
    01101010>>5= 00000011

  算術左シフトは普通存在しない

  では、Verilogではどう書くか？→後ほど、、、

シフト操作の中で論理右シフトは、ずらした隙間に0を詰めるため、ずらしたことにより符号ビットが0になってしまいます。負の数を右シフトさせても負の数の属性を維持するためには、ずらした隙間には符号ビットが１の時は１、0の時は0を詰める必要があります。これを行うのが算術シフトです。この記号はVerilog演算子には存在せず、論理シフトの書き方を元に工夫して書きます。これは後に紹介します。これでALUで行う演算を概ね紹介しました。ではALUをVerilogで書いてみましょう。



Verilog HDLでのALUの記述例

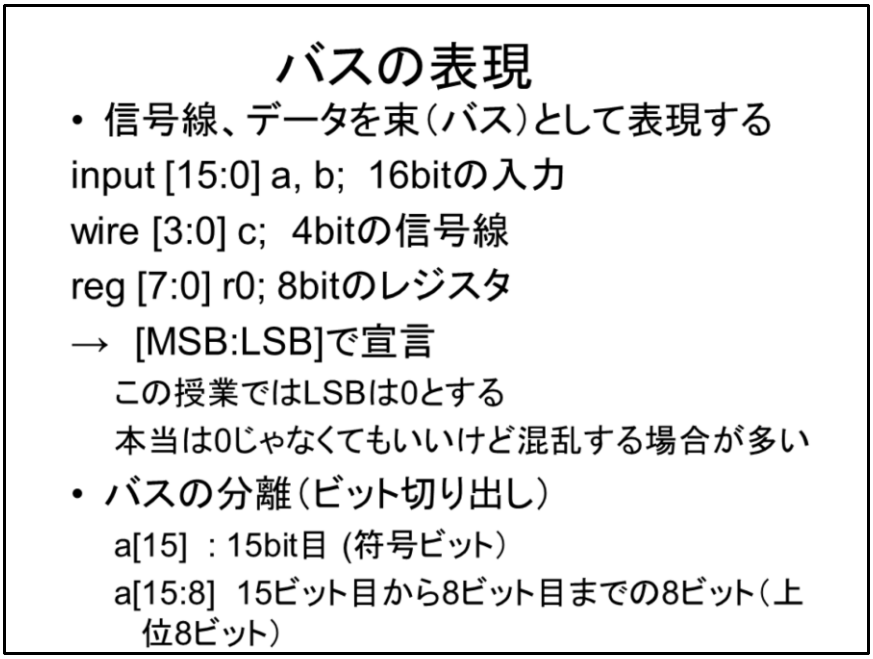
```
module alu (
  input [15:0] a,b,
  input [2:0] s,
  output [15:0] y);
  assign y = s==3’b000 ? a：
                 s==3’b001 ? b：
                 s==3’b010 ? a&b：
                 s==3’b011 ? a|b：
                 s==3’b100 ? a<<1：
                 s==3’b101 ? a>>1：
                 s==3’b110 ? a+b : a-b ;
endmodule
```

バス構文
[MSB:LSB]

選択（マルチプレクサ）演算子
？ ： ；

数の表現

前回同様の操作で2kai.tarをダウンロードし、tar xvfで解凍します。ここで、alu.vがALUの記述です。module文は前回同様ですが、このALUの場合、入力と出力を16ビット、選択入力sは3ビットです。これを表すのに多桁のバス表示を使います。次にsの値に応じてyにaとbの様々な演算結果を出力するために、選択演算子を使います。また多桁の数を表現する方法も使っています。順に説明しましょう。



バスの表現

- 信号線、データを束（バス）として表現する

input [15:0] a, b; 16bitの入力

wire [3:0] c; 4bitの信号線

reg [7:0] r0; 8bitのレジスタ

→ [MSB:LSB]で宣言

この授業ではLSBは0とする

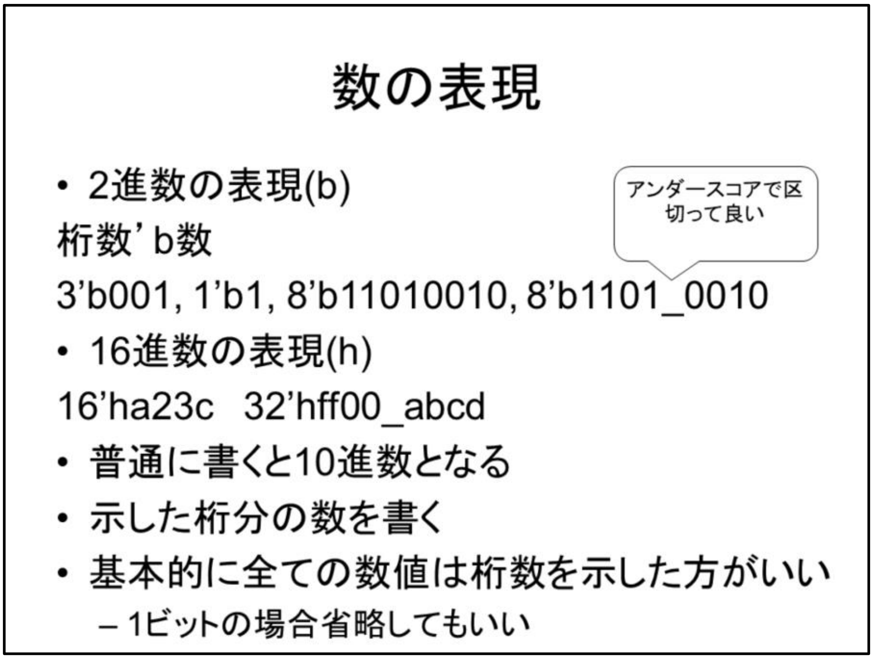
本当は0じゃなくてもいいけど混乱する場合が多い

- バスの分離（ビット切り出し）

a[15] : 15bit目 (符号ビット）

a[15:8] 15ビット目から8ビット目までの8ビット（上位8ビット）

信号線、データは、多数のビットの束として表す方が便利が場合があります。このような信号をバスと呼びます。信号線をバスとして定義するためには、信号名の後に大括弧でMSB(一番上の桁）とLSB（一番下の桁）を指定します。例えばinput a [15:0]は16ビットの入力を表します。LSBは0である必要はないのですが、この授業では混乱を避けるために0として使います。この書き方を使って、バスの一部の信号を表すことができます。例えばa[15]はバスの15ビット目を表します。16ビットのバスの場合、これは符号ビットに相当します。a[15:8]と書いた場合、バスの15ビット目から8ビット目、つまり上位8ビットを表します。



数の表現

- 2進数の表現(b)

桁数’b数

3’b001, 1’b1, 8’b11010010, 8’b1101_0010

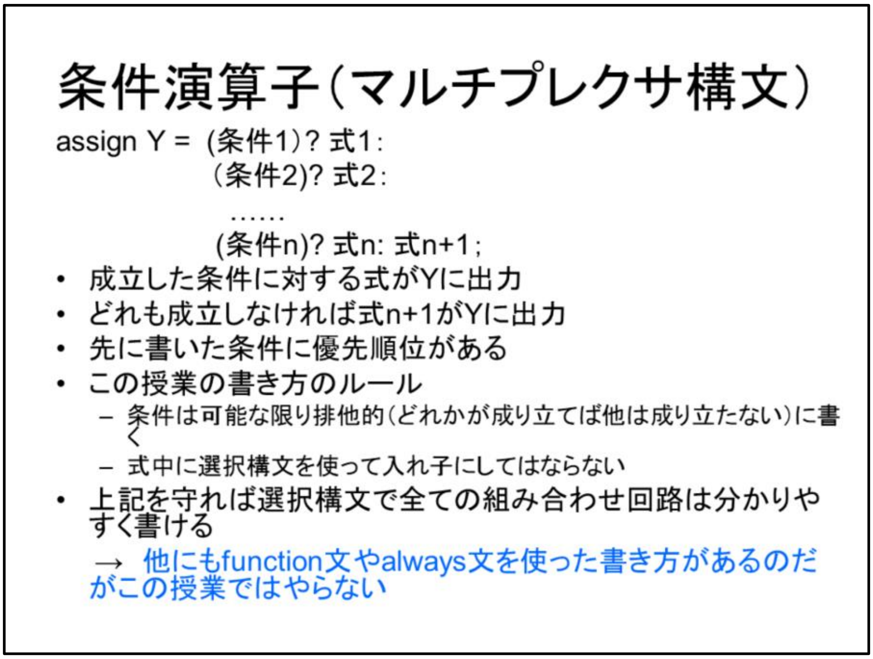
アンダースコアで区切って良い

- 16進数の表現(h)

16’ha23c　32’hff00_abcd

- 普通に書くと10進数となる
- 示した桁分の数を書く
- 基本的に全ての数値は桁数を示した方がいい
  - 1ビットの場合省略してもいい

Verilogでは、C言語同様、何も指定しないと10進数を表します。しかし、ハードウェアの設計では2進数を扱うことが多いので、2進数、16進数の表現方法を持っています。2進数を表現するには「桁数 ‘b数」で示します。例えば3’b001は3ビットの001を示します。桁が長いと読みにくいのでアンダースコアで区切ってもいいです。これはあってもないのと同じに解釈されます。16進数ではbの代わりにhを使います。16進数で表す場合、桁数が４の倍数でなくても良いですが、この場合、一番上の桁の数字に制限が加わります。桁数を示したら、その桁分の数字を書くことをお勧めします。３‘ｂ１は３’ｂ００１と同じですが、処理系によってはエラーになります。基本的にVerilogの記述では、コード中に出てくる全ての数値は、桁数を示して表記することをお勧めします。1ビットの場合、めんどくさいので省略する場合が多いですが、これはOKでしょう（僕は省略しちゃっています。）



条件演算子（マルチプレクサ構文）

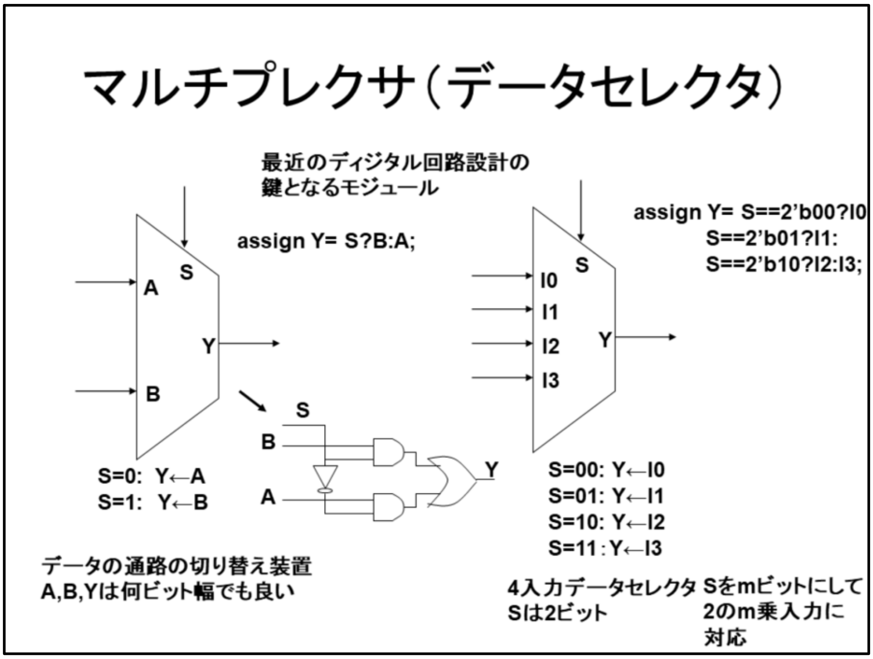
```
assign Y =  (条件1)? 式1:
            (条件2)? 式2:
             ……
            (条件n)? 式n: 式n+1;
```

- 成立した条件に対する式がYに出力
- どれも成立しなければ式n+1がYに出力
- 先に書いた条件に優先順位がある
- この授業の書き方のルール
  - 条件は可能な限り排他的（どれかが成り立てば他は成り立たない）に書く
  - 式中に選択構文を使って入れ子にしてはならない
- 上記を守れば選択構文で全ての組み合わせ回路は分かりやすく書ける

→ 他にもfunction文やalways文を使った書き方があるのだがこの授業ではやらない

今回、最も重要な構文は条件演算子です。これはassign文の右辺の書き方で、条件１？式１：条件２？式２：…：式n+1;の形で、成立した条件に想定した式がYに出力されます。この構文はハードウェアとしてはマルチプレクサを生成するため、マルチプレクサ構文と呼ばれる場合もあります。C言語に慣れた方はswitch文に相当することがご理解できると思います。条件が複数成立した場合、先に書いた方の条件が優先されます。しかし、できる限り条件は排他的に書くことをお勧めします。排他的とは、ある条件が成立したら、他の条件は成り立たない、つまりただひとつだけ条件が成立するという意味です。また、式n+1はデフォルト、つまりどの条件も成り立たない場合に出力されます。条件演算子は式n+1を書かないとエラーになります。

もう一つ注意したいのは、式の書き方です。式の中にさらに条件演算子を使うこともできるのですが、これは絶対に止めてください。条件演算子の入れ子を使うと非常に読みにくくなります。このような場合、信号線を新たに定義して、別の条件演算子を使ってください。



マルチプレクサ（データセレクタ）

最近のディジタル回路設計の
鍵となるモジュール

```
assign Y= S?B:A;
```

```
assign Y= S==2'b00?I0
           S==2'b01?I1:
           S==2'b10?I2:I3;
```

S=0: Y←A
S=1: Y←B

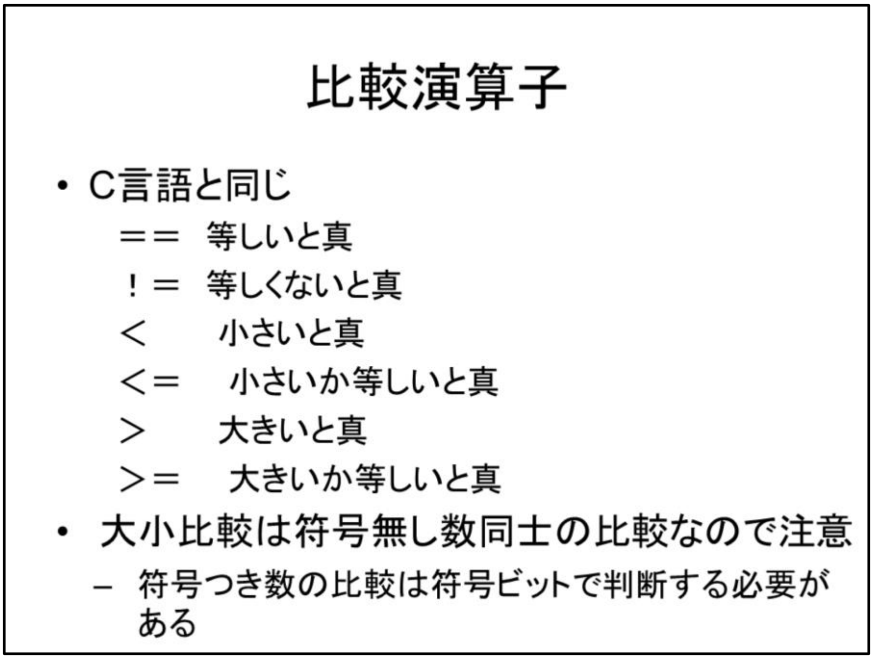
S=00: Y←I0
S=01: Y←I1
S=10: Y←I2
S=11 : Y←I3

データの通路の切り替え装置
A,B,Yは何ビット幅でも良い

4入力データセレクタ
Sは2ビット

Sをmビットにして
2のm乗入力に
対応

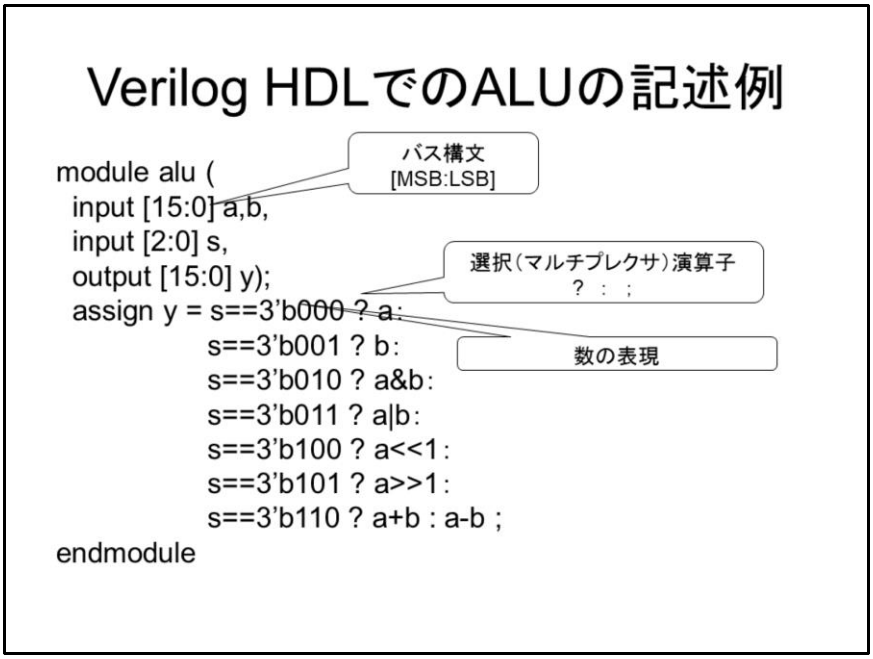
マルチプレクサは電子回路基礎でも勉強しましたが複数の入力から一つの出力を選ぶハードウェアモジュールです。Verilogの条件選択文はこのマルチプレクサを生成します。



比較演算子

- C言語と同じ
  - ＝＝　等しいと真
  - ！＝　等しくないと真
  - ＜　小さいと真
  - ＜＝　小さいか等しいと真
  - ＞　大きいと真
  - ＞＝　大きいか等しいと真
- 大小比較は符号無し数同士の比較なので注意
  - 符号つき数の比較は符号ビットで判断する必要がある

条件を記述するのに比較演算子を使います。これはC言語の比較とほとんど同じなのであまり問題はないでしょう。ただし、大小比較については、全て符号無しの数を想定して比較が行われますので、この点を注意してください。符号付数同士の比較は符号ビットを判断して判断しなければなりません。（この記述は結構めんどくさい）



Verilog HDLでのALUの記述例

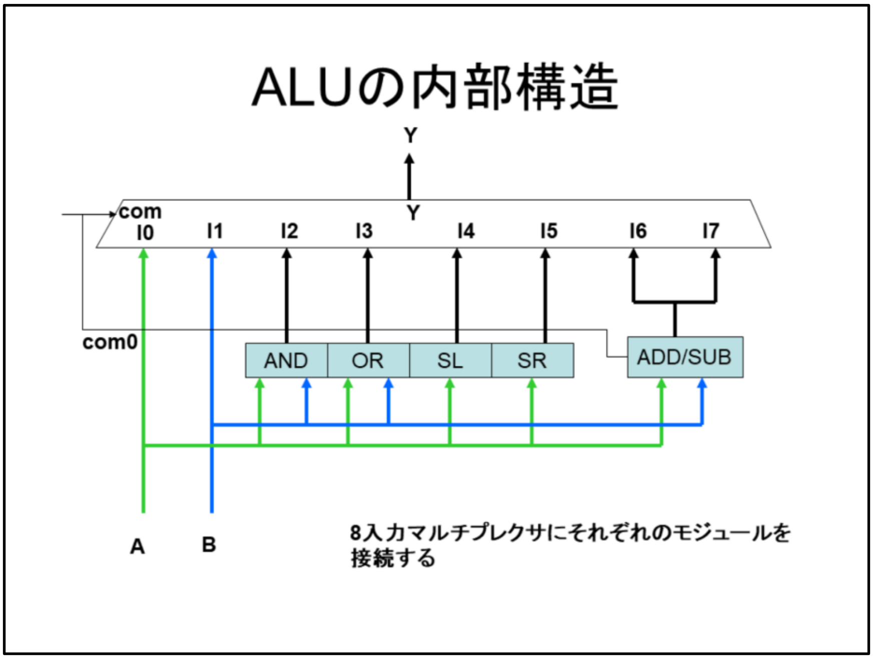
```
module alu (
  input [15:0] a,b,
  input [2:0] s,
  output [15:0] y);
  assign y = s==3'b000 ? a：
             s==3'b001 ? b：
             s==3'b010 ? a&b：
             s==3'b011 ? a|b：
             s==3'b100 ? a<<1：
             s==3'b101 ? a>>1：
             s==3'b110 ? a+b : a-b ;
endmodule
```

バス構文
[MSB:LSB]

選択（マルチプレクサ）演算子
？ ： ；

数の表現

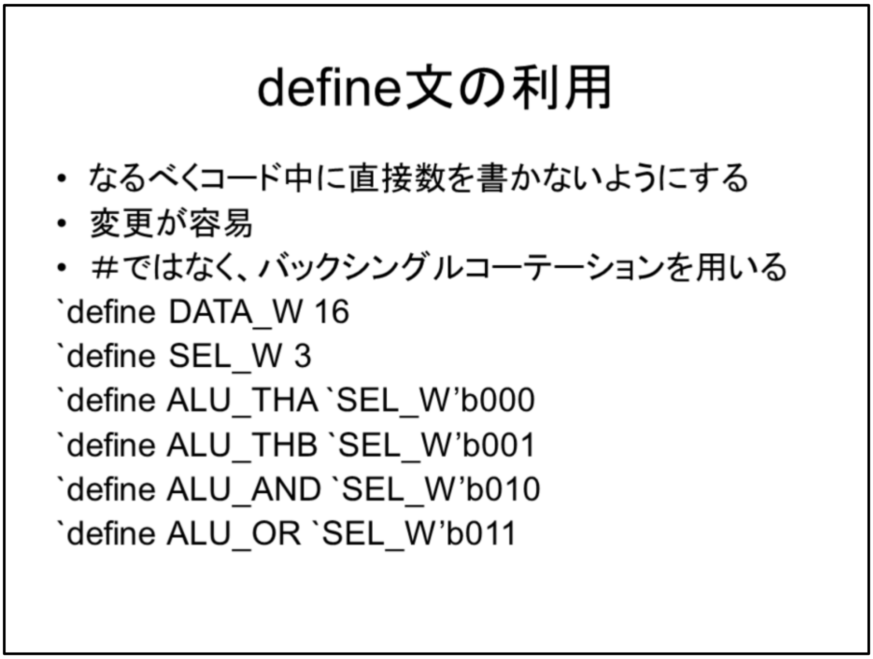
今までの知識でALUの記述を見てみましょう。8種類の演算が選ばれていることがわかります。この選択のパターンは最初に示した例と同じになっています。



ALUの内部構造

8入力マルチプレクサにそれぞれのモジュールを接続する

この記述に対応するハードウェアのイメージを示します。8入力のマルチプレクサを用いて、それぞれのハードウェアからの出力をまとめてYに出力します。本当に論理合成されるのがこの回路になるとは限りませんが、大体これに近い形になります。

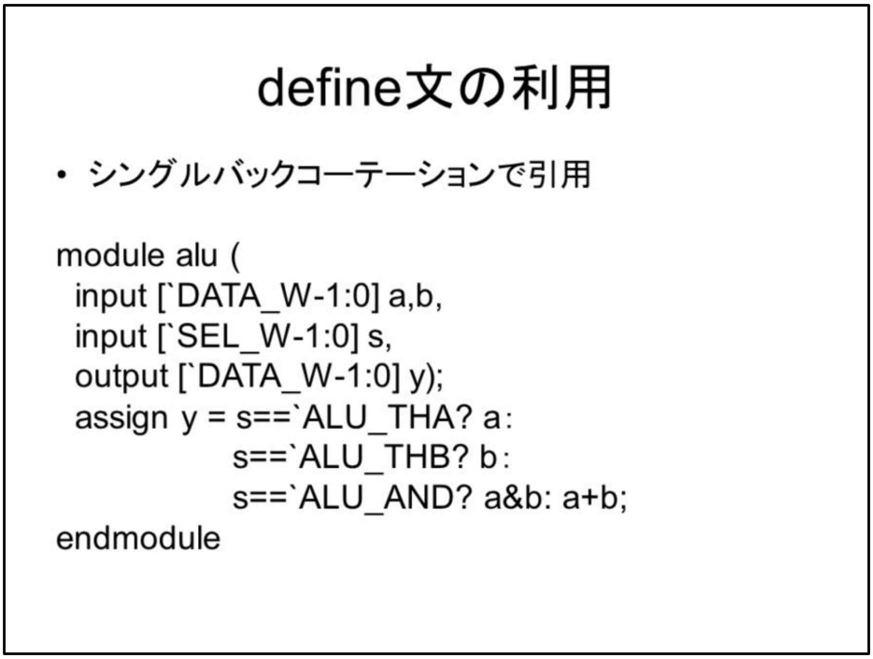


define文の利用

- なるべくコード中に直接数を書かないようにする
- 変更が容易
- ＃ではなく、バックシングルコーテーションを用いる

```
`define DATA_W 16
`define SEL_W 3
`define ALU_THA `SEL_W’b000
`define ALU_THB `SEL_W’b001
`define ALU_AND `SEL_W’b010
`define ALU_OR `SEL_W’b011
```

では次に、Verilogコードの書き方について新しい方法を紹介します。今回様々な値をコード中に記述しましたが、実はなるべくこのような値を直接コードに書くことは避けた方がいいのです。これは、意味が分かり難いことと、後で修正する時に一括でできないことによります。これはどのプログラミング言語でも同じです。そこで、VerilogではC言語同様define文で定義することができるようになっています。C言語との違いは＃ではなくて、バックシングルコーテーションを用いることと、定義するときだけでなく、この定義をコード中で引用するときもバックシングルコーテーションを付けること、の2点です。このバックシングルコーテーションはシングルコーテーション（数字の桁を指定するときに使う）と紛らわしいし、キーボードによって位置がはげしく異なるため、困ったもんです。このためこれを嫌ってparameter文を使う人も居ます。

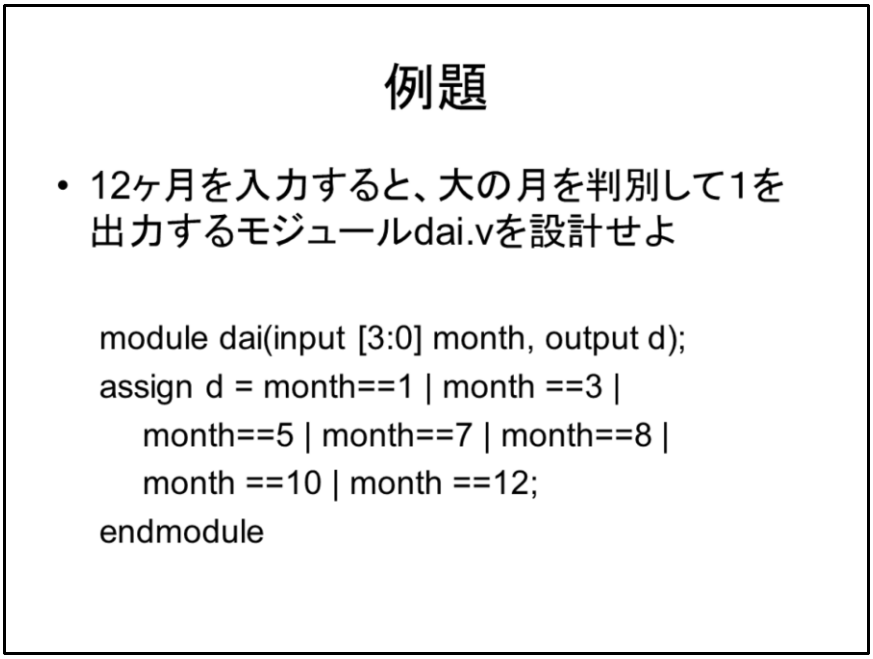


define文の利用

- シングルバックコーテーションで引用

```
module alu (
  input [`DATA_W-1:0] a,b,
  input [`SEL_W-1:0] s,
  output [`DATA_W-1:0] y);
  assign y = s==`ALU_THA? a：
                s==`ALU_THB? b：
                s==`ALU_AND? a&b: a+b;
endmodule
```

シングルバックコーテーションで引用したALUの例を示します。これで少し見やすくなったと思います。



例題

- 12ヶ月を入力すると、大の月を判別して1を出力するモジュールdai.vを設計せよ

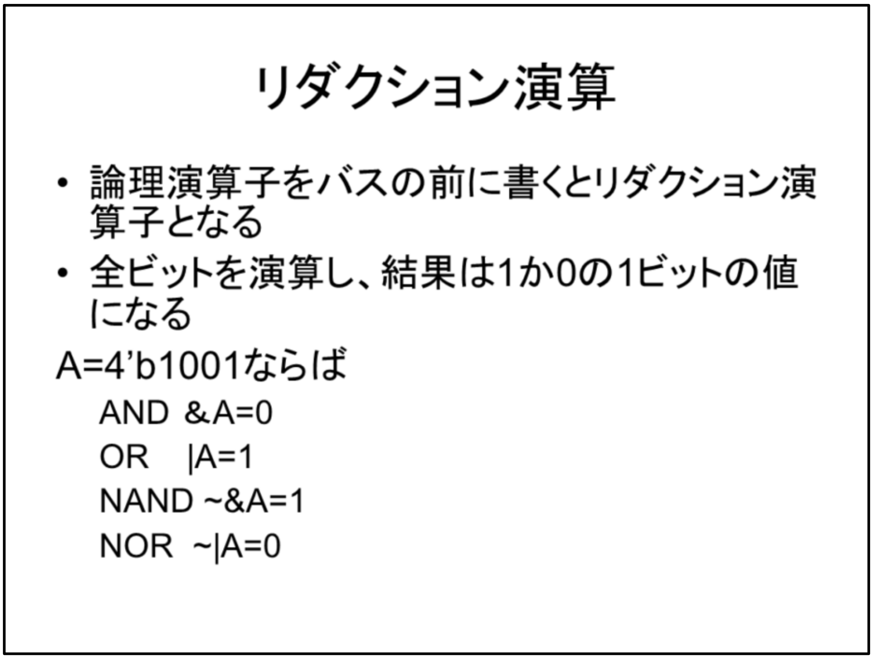
```
module dai(input [3:0] month, output d);
assign d = month==1 | month ==3 |
    month==5 | month==7 | month==8 |
    month ==10 | month ==12;
endmodule
```

比較演算子、バスを使う例題を一つやってみましょう。

別の書き方

```
assign d = month<8 ?  month[0] :
                month<13 ? ~month[0] : 1’b0;
```

これには別の書き方もあります。



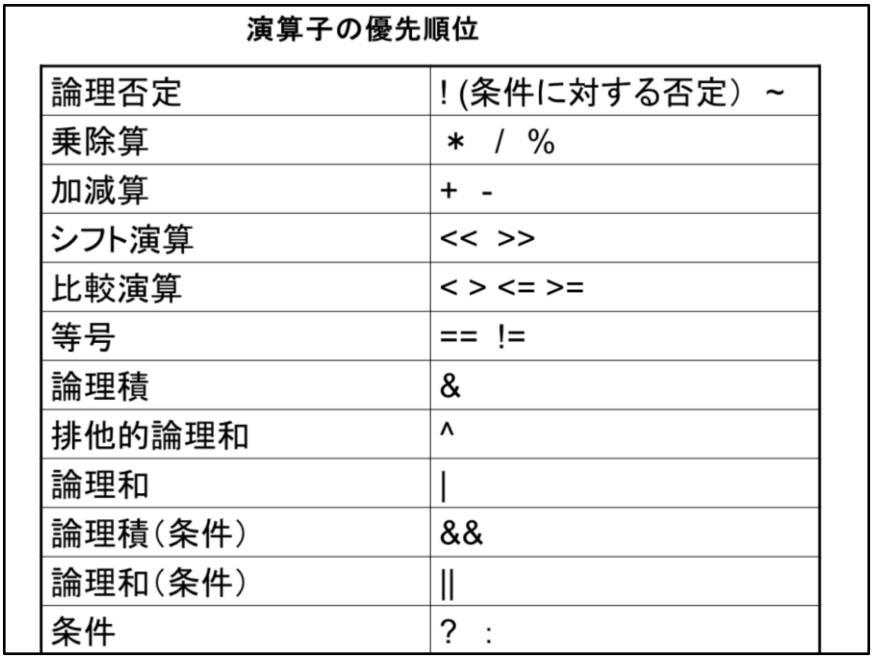
リダクション演算

- 論理演算子をバスの前に書くとリダクション演算子となる
- 全ビットを演算し、結果は1か0の1ビットの値になる

A=4’b1001ならば

```
AND  &A=0
OR    |A=1
NAND ~&A=1
NOR  ~|A=0
```

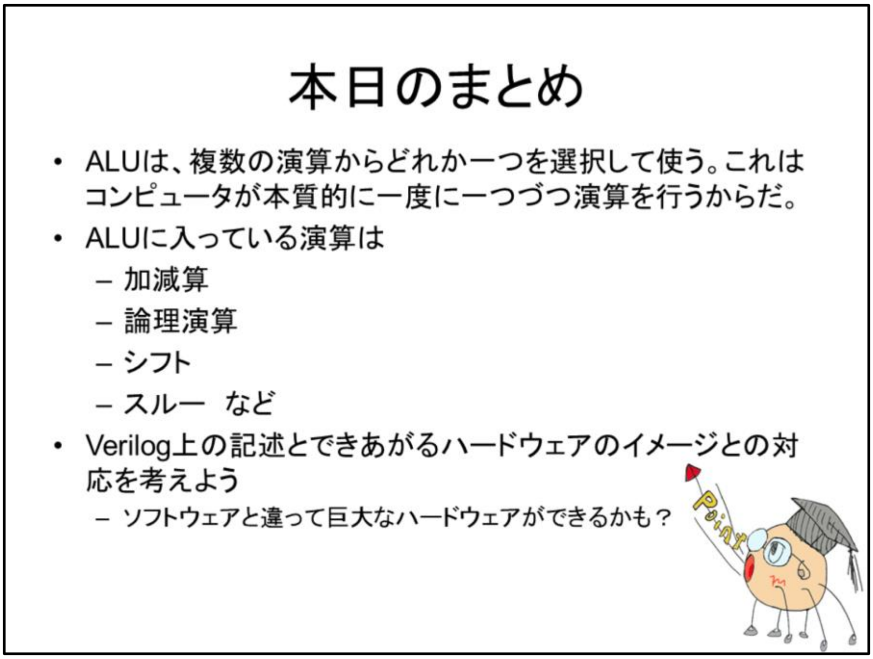
最後に今回は使わないと思うのですが、リダクション演算子について触れて置きます。これは、バスで定義された信号について、全てのビットに対して同一の演算を行って一つの結果を得る演算子です。例えばA=4’b1001で、＆Aと書くと、全ビットのAND、すなわち１＆０＆０＆１が演算され、答えは0となります。A[3]&A[2]&A[1]&A[0]と同じなのですが、信号の名前の前に記号を書けばよいので、スマートに記述ができます。同じ方法でORやNOTを付けることもできます。Aが0かどうかを判別するのに、A==4 ‘b0000とやる代わりに、～｜Aとやってもいいのです。（でも格好だけで同じですが、、）



演算子の優先順位

論理否定	! (条件に対する否定） ~
乗除算	＊ / %
加減算	+ -
シフト演算	<< >>
比較演算	< > <= >=
等号	== !=
論理積	&
排他的論理和	^
論理和	|
論理積（条件）	&&
論理和（条件）	||
条件	? :

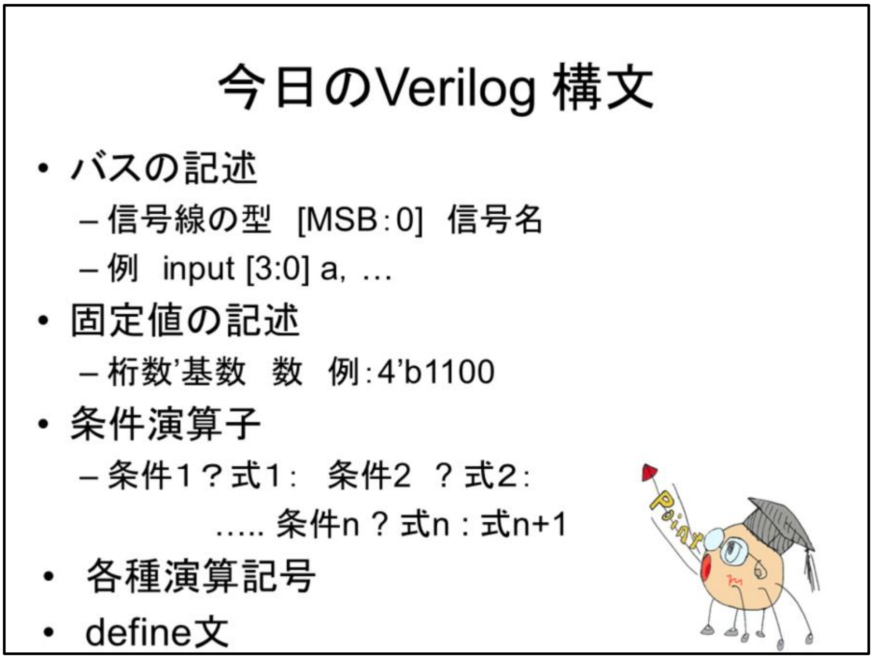
最後にここで使う演算子の優先順位を示します。比較的常識的な優先順位ですので、自然に使えば大丈夫です。



本日のまとめ

- ALUは、複数の演算からどれか一つを選択して使う。これはコンピュータが本質的に一度に一つづつ演算を行うからだ。
- ALUに入っている演算は
  - 加減算
  - 論理演算
  - シフト
  - スルー　など
- Verilog上の記述とできあがるハードウェアのイメージとの対応を考えよう
  - ソフトウェアと違って巨大なハードウェアができるかも？

インフォ丸が教えてくれる今日のまとめです。



今日のVerilog 構文

- バスの記述
  - 信号線の型 [MSB:0] 信号名
  - 例 input [3:0] a, …
- 固定値の記述
  - 桁数’基数 数 例:4’b1100
- 条件演算子
  - 条件1？式1: 条件2 ？式2:
    ….. 条件n ? 式n : 式n+1
- 各種演算記号
- define文

今回は新しい構文をたくさん紹介しました。どれも重要ですが、そんなに難しいのはないと思います。

演習

演習2－1　2kai中のALU(alu.v)をシフトの代わりに論理反転と排他的論理和を入れるようにせよ。

演習2－2　4ビットの正の数のうち、素数が入力された際に1を出力するモジュールsosu.vを設計せよ。

では演習課題をやってみましょう。

演習問題2ヒント

```
module sosu (input [3:0] a, output y);
assign y=….

endmodule
```

2進数4ビットの素数は2，3，5，7，11，13

素数を数えて気を落ち着けて、さあ中身を考えよう！