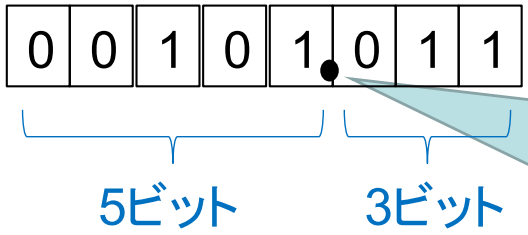
実際のコンピュータ内部での表現は？

- まず、表現できる桁数は固定（有限）、つまり制限があります。
  - メモリ素子数や配線数を、コンピュータハードウェアの製造後に変更することはできません。
- 小数点をどう扱うか、決めておく必要があります。
- 負の数をどのように表現するのかも、決めておく必要があります。

小数点の扱い

- 固定小数点数では、小数点の位置は、決まった位置に定めます。
  - 例えば、8ビットで数を表現する場合、整数部を5ビット、小数部を3ビットと定めることにします。
    - 00101.011 は以下のように表現します。



小数点はここに「あるもの」として扱います。小数点自体をデータの一部として表現・記憶しているのではないことに注意してください（つまり、小数点は見えません）。

– さて、ここで固定小数点数の加減算について考えてみます。

例えば、

- 00101.011 + 00011.010 = 01000.101
- 0010.1011 + 0001.1010 = 0100.0101
- 00101011. + 00011010. = 01000101.

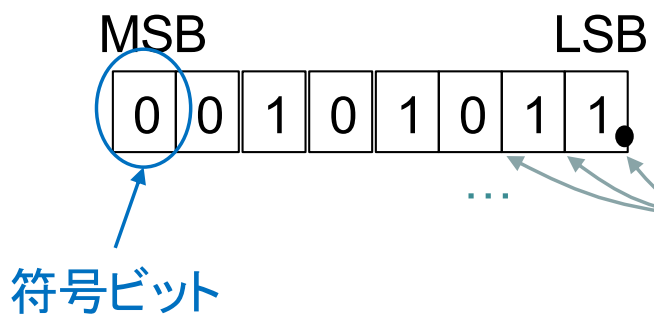
これらは、いずれも、0/1のパターンとしては同じ計算です。

つまり、8ビットの加算回路（加算器）を論理設計する際には、小数点の位置など考慮する必要はないのです。

- だから、小数点の位置は、使う側（プログラマ）が勝手に決めて良い、ということになります。
- つまり、整数部5ビット・小数部3ビットの実数同士の加算も、整数部4ビット・小数部4ビットの実数同士の加算も、8ビットの整数同士の加算として計算して何も不都合は生じません。
  - このことから、通常は、固定小数点数を整数として扱っている場合が多いのです。
  - 乗除算の場合は、整数として計算すると小数点の位置がズレますので、使う側が注意する（つまりは修正する）必要が生じます。

負の数の表現

- まず、最上位ビットを符号ビットとして、それ以外のビットで数の大きさを表します。
  - 符号ビットは、0なら正、1なら負と解釈します。



小数点をどこに置くかはプログラマが勝手に決めれば良いのですが、特に必要がなければ右端にあるもの（つまり、整数）として扱います。

最上位ビットをMSB(Most Significant Bit)、最下位ビットとLSB(Least Significant Bit)と呼びます。最後のBは、bitではなくbyteを指すこともあるので注意が必要です。

- 負の数の表現方法として、以下の3種類が考えられます。
  - 符号と絶対値による表現
  - 1の補数による表現
  - 2の補数による表現

次ページから、上記の表現について、順に見ていきます。

- 数の表現例としては、4ビットの固定小数点数の場合で例示することにします。

符号と絶対値による表現

- 符号ビットのみで正負を表現します。
  - 例） 0101 (= 5), 1101 (= −5)
- nビットの整数m の表現範囲
  - $-(2^{n-1}-1) \leqq m \leqq 2^{n-1}-1$
    - 例） 1111 (= −7) $\leqq m \leqq$ 0111 (= 7)
- 特徴:
  - 人間にとって理解しやすい
  - 0に対して+0と−0の2通りの表現

1（*r*–1）の補数表現

- 正の数に対する負の数の作り方
  - 各桁を1（*r*–1）から引く
    - *r*=2の場合は各桁を反転(NOT)したものと同じ
  - 例） 0101 (= 5), 1010 (= −5)

  - 負の数から正の数をつくるときも同じ操作

- nビットの整数m の表現範囲
  - $-(2^{n-1}-1) \leqq m \leqq 2^{n-1}-1$
    - 例） 1000 (= −7) ≦ m ≦ 0111 (= 7)

- 特徴:
  - 0に対して+0と−0の2通りの表現
  - ハードウェアで補数が作りやすい
    - 各桁（ビット）毎にNOT演算を行うだけ

2(*r*)の補数表現

- 正の数に対する負の数の作り方
  - *r*=2の場合は各桁を反転(NOT)したものの最下位に1を加算する
    - つまり、1の補数の最下位に1を加算する
  - 例） 0101 (= 5), 1011 (= −5)

各桁を反転して＋1

- 0101→(反転)→1010→(+1)→1011
- 1011→(反転)→0100→(+1)→0101

  - 負の数から正の数をつくるときも同じ操作

- nビットの整数m の表現範囲

  $-2^{n-1} \leqq m \leqq 2^{n-1}-1$

  - 例） 1000 (= −8) $\leqq m \leqq$ 0111 (= 7)

- 特徴:
  - 0の表現は一意
  - 補数の生成に加算が必要

さて、どの表現を用いるのが都合が良いのでしょうか？

そして、補数ってどういう意味があるのでしょうか？　なぜ、負の数を補数で表現しようなどという考えが浮かんだのでしょうか？

次ページ以降で、これらについて考えていきます。

- まず、符号と絶対値による表現について考えます。
  - これが使えれば、わかりやすくて好ましいのですが、実は、そう上手くはいきません。

  - ところで皆さんは、「2−5=？」の計算はできますよね？　でも、「2から、直接、一度に5を引く」ことはできますか？　“直接、一度に”ですよ。
  - 恐らく、できないと思います。
  - 皆さんの頭の中では、5から2を引いて、その結果の3にマイナスを付けて答えにするはずです。

- つまり、皆さんは、2と5のどちらが大きかを最初からわかっているのです。
  - だから、大きい方から小さい方を引こうとするのです。
- しかし、これが機械（論理回路）に“最初から”わかるでしょうか？
  - どちらが大きいかは、引いてみて、結果が正か負かでようやく判断できるのです。
- したがって、「引き算をするためには、すでに引き算ができる状態でなければならない」ということです。
  - これでは「引き算のしかた」を導き出すことはできません。

- 以上より、結論として、「符号と絶対値による表現では使い物にならない」ということになります。

- そこで、補数を用いる方法に期待が寄せられたわけです。
  - 補数の数学的な定義は以下の通りです。
    - ある定数Cに対して、$b=C-a$を、aの補数といいます。
      - したがって、aはbの補数でもあります。
  - nビットの固定少数点数において
    - $C=2^n$ としたものが “2の補数” です。
    - $C=2^n-1$ としたものが “1の補数” です。

- そして、C(≠0)を0とみなすことにしよう、と決めるのです。
  - *n*=3の場合で考えると、

この素晴らしいアイデアには、全く、脱帽ですね。

10進数	1の補数		2の補数	
3	011		011	+1
2	010		010	+1
1	001		001	+1
0	000	= 111	000	= 1000
-1		110	111	-1
-2		101	110	-1
-3		100	101	-1
-4			100	-1

この1は3ビットを超えるので、実際には捨てられている(表現されない)。

符号ビットで正/負を分ける。

- このように、負の数を補数によって表現することで、引き算を足し算で行うことができるようになります。
  - “*a*−*b*” は、“*a*+（*b*の補数）” で計算できます。
  - つまり、コンピュータのハードウェア設計（論理設計）において、わざわざ減算回路を設けなくても、加算回路だけで加算も減算も計算できるのです。
- そして、加算は、符号ビットも含めて普通に2進数として計算することができます。
  - その様子を次ページから見ていきます。

2の補数表現における加算

- 例として、4ビットの固定小数点数で考えます。
- 正の数同士の加算
  - 0101(5)+0001(1)=0110(6)　→OK
  - 0110(6)+0011(3)=1001(-7)　→NG
    - 4ビットの符号無し整数（非負の整数）として見たときは、6+3=9で正解です。
    - しかし、4ビットの2の補数表現では、結果が負になってしまいました。これは表現できる最大数の7を超えてしまったからです。

- 演算結果が表現できる範囲を超えてしまった場合のエラーをオーバフロー（overflow、桁あふれ）といいます。
- このようなエラーは、演算結果とは別に、検出しなければなりません。
  - なお、符号無し整数のつもりで計算している場合は、この結果はエラーではないことに注意してください。

- 負の数同士の加算
  - 1100(-4)+1110(-2)=1010(-6)　→OK
    - 計算結果は、本当は5ビット（11010）になります。
    - 最上位桁からの桁上げ（キャリー; carry）は、符号無し整数の場合は、通常、エラーとして検出しなければなりません。
    - 補数表現の場合は、桁上げは無視して構いません。

– 1010(-6)+1011(-5)=0101(5)　→NG

- 負の数の加算結果が正になっています。$(-11)_{10}$は4ビットでは表現できないからです。この場合も、オーバフローとして検出しなければなりません。
- 符号無し整数として加算した場合も、最上位ビットから桁上げが生じていますから、これを検出しなければなりません。
  - 結局のところ、コンピュータハードウェアにとっては、符号無しと符号付(補数表現)のどちらの計算かは判別できません。つまり、1010+1011が、10進数で10+11の意味なのか(-6)+(-5)の意味なのか判別できませんが、どちらであっても1010+1011の加算処理自体は同じです。
  - そこで、ハードウェアでオーバフローと桁上げをそれぞれ個別に(両方とも)検出し、プログラム中でそのどちらかの検出結果を調べて、エラー処理を行なえるようにします(しています)。

- 正の数と負の数の加算
  - この場合はオーバフローは起こりません。

  - 0011(3)+1011(-5)=1110(-2)　→OK
  - 1101(-3)+0101(5)=0010(2)　→OK
    - この例のように、桁上げが発生することはあります。もちろん、補数表現の場合は、桁上げを無視します。

1の補数表現における加算

- 基本的に、2の補数の場合と同様に計算できますが、ちょっとだけ厄介なことが起こります。
- 例えば、
  - 1100(-3)+0101(5)=0001(1)　→NG？
    - オーバフローは起こっていませんが、結果が変です。
    - ただし、この計算では、桁上げが発生しています。

- 結局、1の補数の場合は次のように計算します
  - 符号ビットも含めて、普通に2進数として加算します。
  - 最上位ビットからの桁上げを、求めた和の最下位ビットに加えて補正します。
    - これを循環桁上げ(end-around carry)と言います。
    - 結局、加算を2回行わなければなりません。

```
    1100   (-3)
+)  0101   (5)
-----------
   10001
       1
-----------
   0010   (2)
```

```
    0011   (3)
+)  1011   (-5)
-----------
   01110
       0
-----------
   1110   (-2)
```

- ちょっと待ってください！　おかしいです！
  - 2^nの位の値を2^0の位に加算するなんて、無茶苦茶ですねぇ。。。
  - でも、これは、数学として考えたことによる結論ではなく、論理回路として考えた結論なのです。
  - どういうことか、見ていきましょう。

- 再び (−3)+5 の計算をします。
  - 今度は (−3) に +1 を5回施します。
    - » 桁上げ分を含めて5ビットで表現することにします
  - (0)1100→(0)1101→(0)1110→(0)1111→(1)0000→(1)0001
    ちゃんと、5回、1ずつカウントアップしました。
  - しかし、1の補数表現では、1111 も 0000 も 0 を表します。つまり、1111 から 0000 にカウントアップしても、1増えたことにはならないのです。
    - これが、普通に計算したときに、和が1少ない値として求まる原因です。
      - この例では、答えが2になるべきところ、1になっています。

– したがって、1111 から 0000 へカウントアップした場合は、結果に1を加えて補正するのです。
– そして、1111 から 0000 へカウントアップしたかどうかは、最上位ビットから桁上げが生じたかどうかで検出することができます。
  • 実際には1ずつ増やして数えるのではないことに注意してください。
– したがって、結論としては、
  • 最上位ビットから桁上げが生じれば、加算結果に1を加える。
  • 桁上げが生じなければ、何もしない（0を加える）。

という処理になります。

– そして、1を加えるか否かは、桁上げが1か0かで決まるのですから、桁上げの値をそのまま最下位に加算してしまえばよい、ということになります。
  - 真理値表を書くまでもない簡単な話です。

– ということで、1の補数表現の場合は、前述のように、循環桁上げを用いた補正を行います。

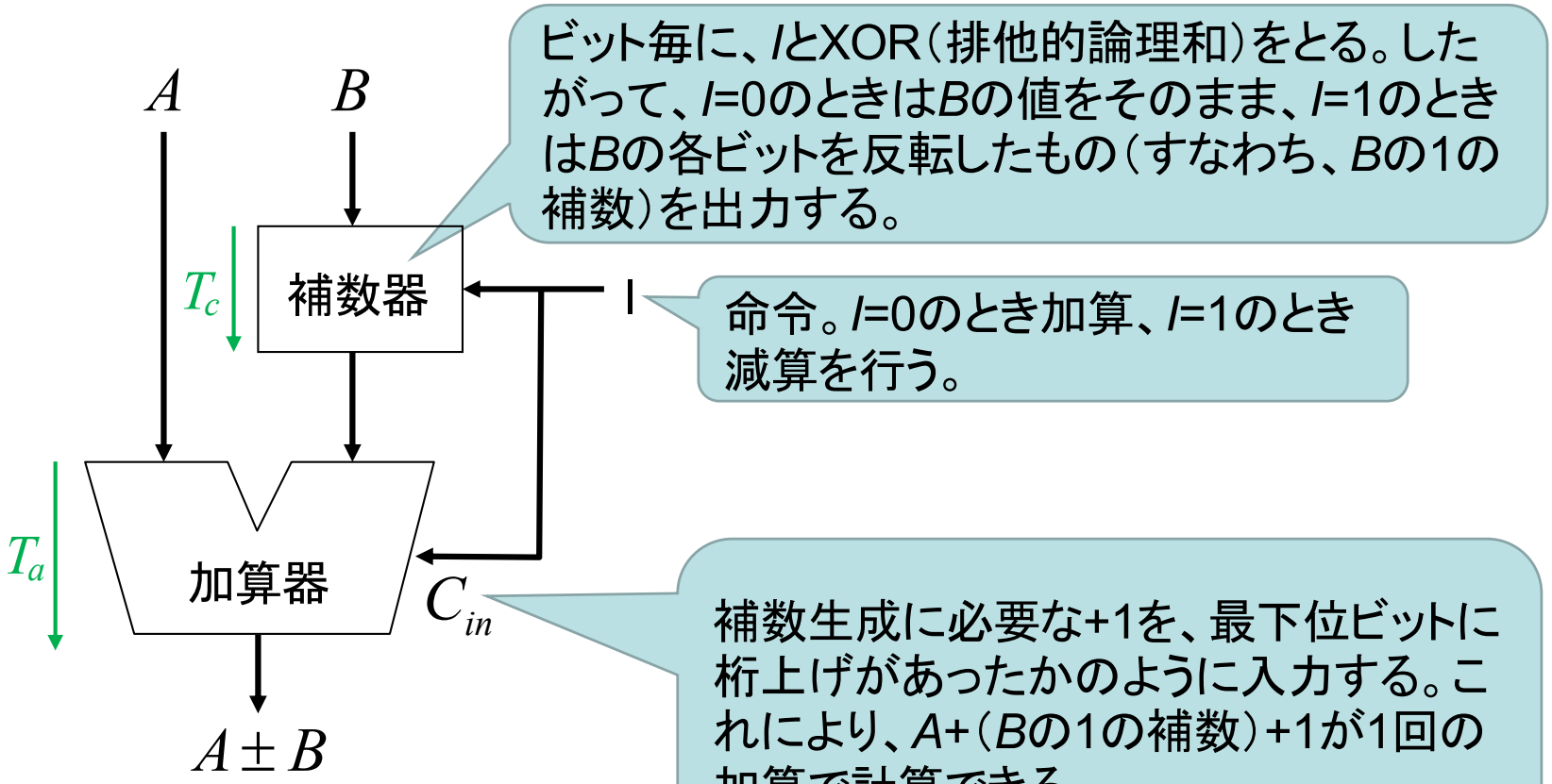
1の補数 vs 2の補数

- では、1の補数と2の補数のどちらで表現するのがよいのでしょうか？

- これを検討するために、加減算を行う場合を対象にして、両者の特徴を比較します。

- 1の補数
  - 補数生成は、各ビットを反転するだけ。
  - 加算結果に補正（循環桁上げ）が必要。つまり、加算を2回行う必要がある。
- 2の補数
  - 補数生成に+1の加算が必要。
  - 加算時に補正は不要。つまり、加算は1回だけ。
- 結局、どちらも、合計で2回加算しなければなりません。
  - これでは、どちらがよいか決着がつきません。しかし。。。

- 2の補数の場合は、以下の加算回路によって、1回の加算で計算できます。



- 補数器の遅延時間をT_c、加算器の遅延時間をT_aとすると、加減算に要する時間はT_a+T_cとなります。

- これに対して、1の補数の場合は、
  - 加算結果に対して、さらに補正の加算が必要になるので（加算を行う前に補正が必要かどうかはわからない）、加算回数を1回に減らすのは無理。
  - したがって、加減算に要する時間は$2T_a+T_c$です。

- よって、2の補数を用いた方が、高速に加減算が行えます。
  - したがって、一般には、2の補数を用いて負の数を表現します。