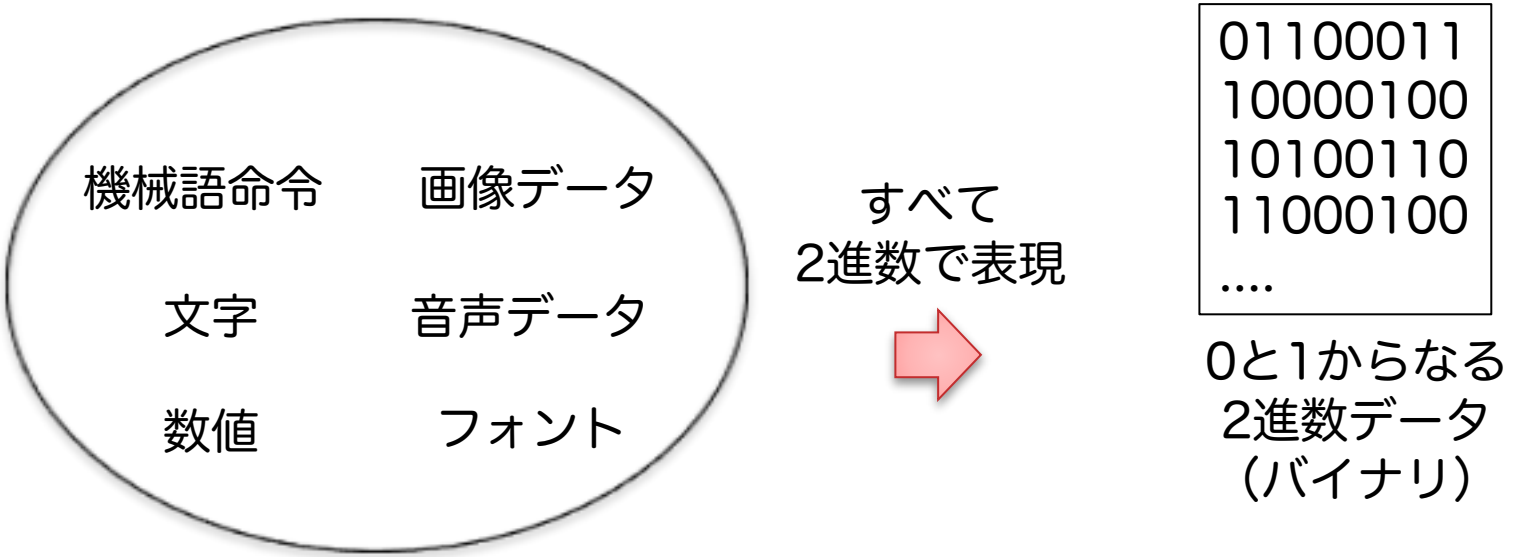
アセンブリ言語

データ表現，2の補数

情報工学系
権藤克彦

コンピュータは0と1だけの世界

- コンピュータ中のデータは，すべて0と1から成る．
  - つまり，2進数（バイナリ）で表現．
  - プログラム（機械語命令）も2進数で表現．

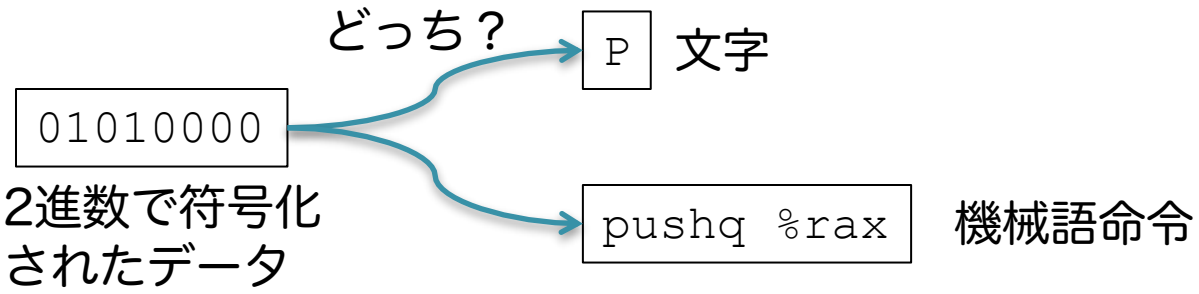


- 2進数表現は長い→人間の読み書きには16進数を使う．
  - 例：2進数の01010000は，16進数では50．
    （10進数と区別するため 0x50，50_{16} などとも表現する）

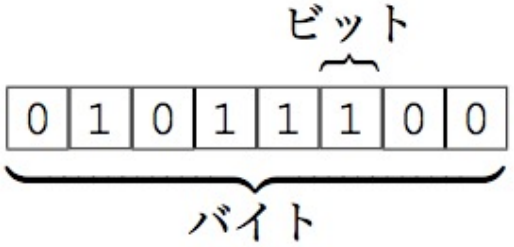
2進数と符号化 (encoding)

- 符号化
  - ある規則に従って，文字や機械語命令などを符号（2進数）に変換すること.
  - 例：文字P を ASCII文字として符号化すると 01010000.
  - 例：pushq %rax をx86-64機械語命令として符号化すると 01010000. ← 同じ

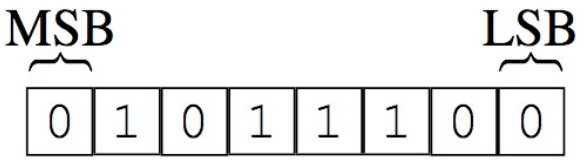
- ある2進数が何を表すかは解釈によって異なる.
  - 解釈の方法が分からなければ，01010000が何を表すのか分からない.



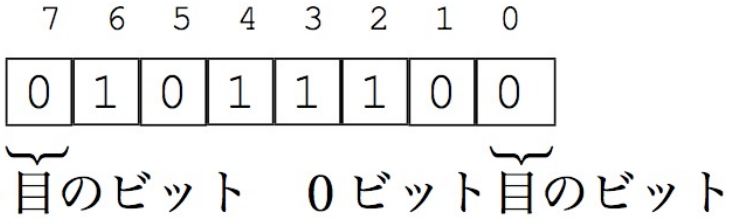
ビット，バイト，MSB，LSB



- ビット(bit)
  - コンピュータが扱うデータ量の最小単位．binary digit の略．
  - 2進数の1桁が1ビット．1ビットで0か1かの2通りの状態を表現．
- バイト(byte)
  - 通常，8ビットのこと．



- MSBとLSB
  - ビット列の最左ビット＝最上位ビット (MSB, most significant bit).
  - ビット列の最右ビット＝最下位ビット (LSB, least significant bit).



- ビットの呼び方
  - LSBから左に，0ビット目，1ビット目…，7ビット目と呼ぶ．
  - 0から始まることがポイント．これを0オリジンという．

補助単位

- 大きなデータを簡潔に表現するのに使う.
  - 例：10,000,000バイトを10メガバイト（10MB）と表現.
- キロ，メガ，ギガ，テラ，ペタを覚える.

名称	記号	正式	慣用
キロ	k	10^3	2^{10}
メガ	M	10^6	2^{20}
ギガ	G	10^9	2^{30}
テラ	T	10^{12}	2^{40}
ペタ	P	10^{15}	2^{50}

SI単位系の補助単位（一部）

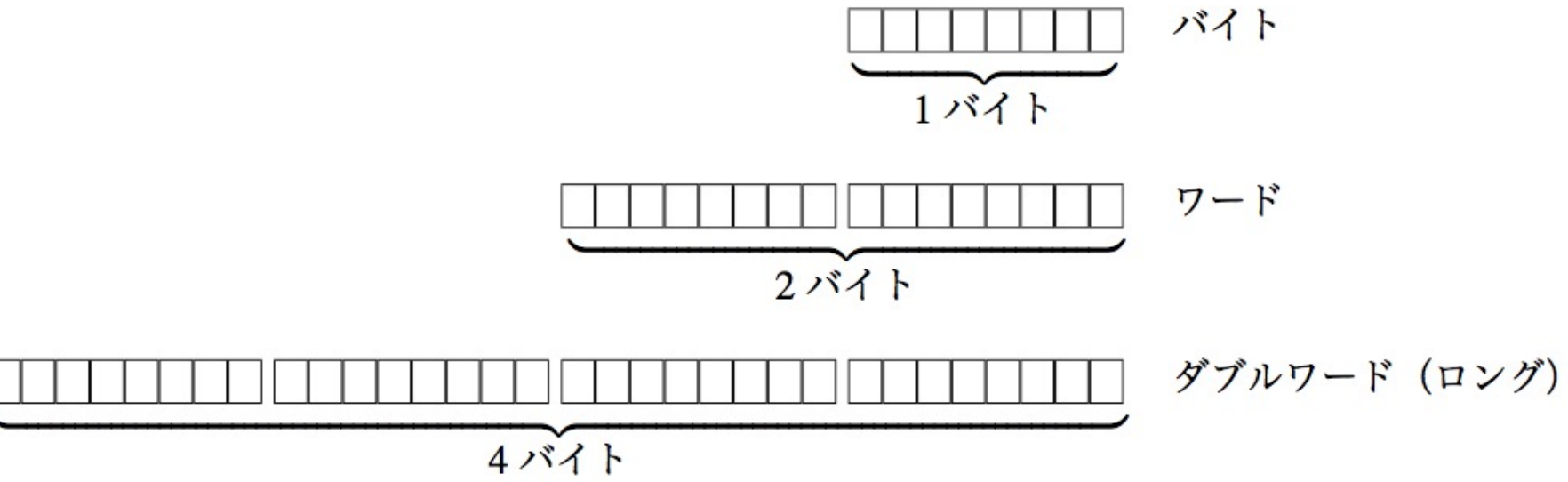
国際単位系（SI単位系）
The international system of units

ワード，ロング，クアッド（1）

- ワード(word)は2バイト.
- ロング(long)は4バイト.
- クアッド(quad)は8バイト.

x86-64の場合

  - ワード等の大きさはプラットフォーム依存.
  - ロングのことをダブルワード(double word)とも呼ぶ.



クアッドの図は省略

ワード，ロング，クアッド（2）

	サイズ		Cのデータ型(ILP32)	Cのデータ型(LP64)	GNUアセンブラの表現	
	バイト数	ビット数			データ型	命令の接尾語
バイト	1	8	char	char	.byte	movb
ワード	2	16	short	short	.word	movw
ロング	4	32	int long ポインタ	int	.long	movl
クアッド	8	64	long long	long ポインタ	.quad	movq

- Cのデータ型
  - short, int, long，ポインタ型のサイズはプラットフォーム依存.
- 機械語命令の接尾語
  - GNUアセンブラでは機械語命令に接尾語がつく.

2進数，8進数，10進数，16進数（1）

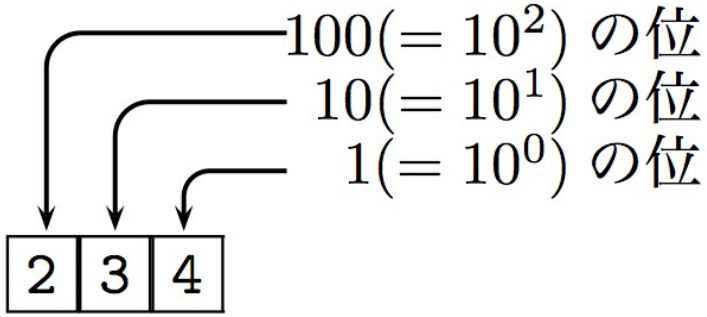
nを底・基数(radix)という

- n進数で $d_m d_{m-1} \cdots d_2 d_1 d_0$ という数の値は,

$$\sum_{i=0}^{m} d_i \times n^i = d_m \times n^m + d_{m-1} \times n^{m-1} + \cdots + d_2 \times n^2 + d_1 \times n^1 + d_0 \times n^0$$

- 10進数(decimal number)で 234 という数の値は,

$$2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0 = 200 + 30 + 4 = 234$$



- 2進数(binary number)で 1101 という数の値は,

$$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 8 + 4 + 1 = 13$$

2進数を10進数に変換

2進数，8進数，10進数，16進数（2）

- 8進数(octal number)は0から7までの8種類の数字を使う.
- 8進数で 234 という数の値は,

$$2 \times 8^2 + 3 \times 8^1 + 4 \times 8^0 = 128 + 24 + 4 = 156$$

- 16進数(hexadecimal number)は0から9，AからFまでの16種類の数字を使う.
  - Aの値は10，Bの値は11，・・・，Fの値は15.
- 16進数で 1F4 という数の値は,

$$1 \times 16^2 + \mathrm{F} \times 16^1 + 4 \times 16^0 = 256 + 240 + 4 = 500$$

対応表

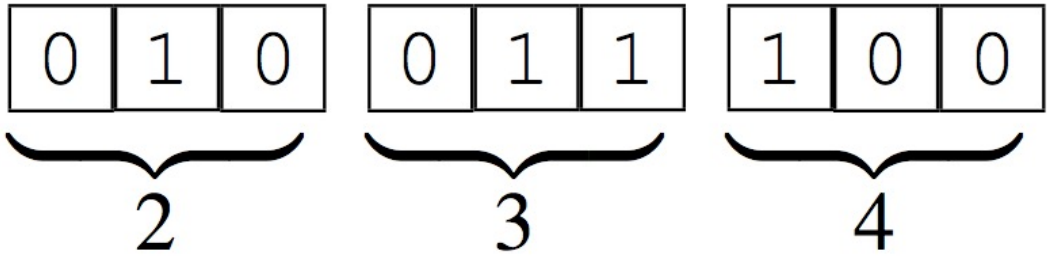
10進数	0	1	2	3	4	5	6	7
2進数	0	1	10	11	100	101	110	111
8進数	0	1	2	3	4	5	6	7
16進数	0	1	2	3	4	5	6	7

10進数	8	9	10	11	12	13	14	15
2進数	1000	1001	1010	1011	1100	1101	1110	1111
8進数	10	11	12	13	14	15	16	17
16進数	8	9	A	B	C	D	E	F

8進数・16進数は2進数をまとめた表現

最下位ビットから

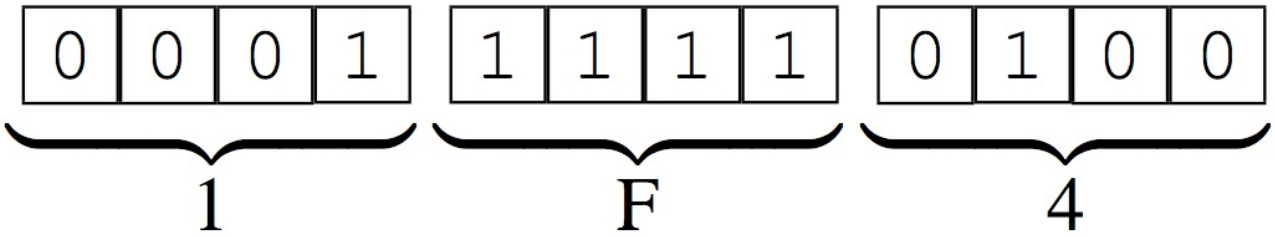
- 8進数は2進数を3桁ずつまとめた表現.



2進数

8進数

- 16進数は2進数を4桁ずつまとめた表現.



2進数

16進数

2進数⇔8進数・16進数の変換は簡単

10進数→2進数の変換

- 0になるまで繰り返し2で割り，余りを逆順に並べる.
- 例：13 を2進数に変換.

$$
\left.\begin{array}{l}
13 \div 2 = 6 \text{ 余り } 1 \\
6 \div 2 = 3 \text{ 余り } 0 \\
3 \div 2 = 1 \text{ 余り } 1 \\
1 \div 2 = 0 \text{ 余り } 1
\end{array}\right\} \text{余りを下から上に並べて } 1101
$$

8進数や16進数への変換も同様にできる.

bcコマンドで変換

```
% bc
1+2*3+10/5
9
^D
%
```

- bcコマンドはいわゆる「電卓」.
  - 例：簡単な四則演算を計算.
  - 最後は^D（コントロールを押しながらD）を入力して終了.

- 入力の底(ibase)や出力の底(obase)を変更すれば,任意のn進数の変換ができる.
  - 例：10進数の500を2進数の111110100に変換.
  - obaseを先に変更するのがコツ.

```
% bc
obase=2
ibase=10
500
111110100
^D
%
```

文字コード

- 各文字を区別するために，重複無く割り振った番号.
  - 例：ASCIIコードで文字Aの文字コードは（1バイトの）65.
  - 例：ASCIIコードで文字9の文字コードは（1バイトの）57.
- 文字コードはいろいろある.
  - ASCII，ISO-8859-1，ISO-2022-JP（JISコード），EUC-JP，Shift_JIS，Unicode，UTF-8，・・・

ここでは簡単のため**文字集合**と
文字符号化方式を区別していない.

- 「フォントや大きさの違い」の情報を含まない.

Times

Matura MT Script Capitals

どちらも（ASCIIコードでは）
文字コードは同じ65になる.

- この講義ではASCIIコードだけを扱う.

ASCIIコード

- 128個の文字を扱う1バイトのコード体系.
  - MSBは常に0.
- アルファベット，数字，記号，制御文字を含む.

番号	文字	番号	文字	番号	文字	番号	文字	番号	文字	番号	文字	番号	文字	番号	文字
0	^@	16	^P	32	␣	48	0	64	@	80	P	96	`	112	p
1	^A	17	^Q	33	!	49	1	65	A	81	Q	97	a	113	q
2	^B	18	^R	34	"	50	2	66	B	82	R	98	b	114	r
3	^C	19	^S	35	#	51	3	67	C	83	S	99	c	115	s
4	^D	20	^T	36	$	52	4	68	D	84	T	100	d	116	t
5	^E	21	^U	37	%	53	5	69	E	85	U	101	e	117	u
6	^F	22	^V	38	&	54	6	70	F	86	V	102	f	118	v
7	^G	23	^W	39	'	55	7	71	G	87	W	103	g	119	w
8	^H	24	^X	40	(	56	8	72	H	88	X	104	h	120	x
9	^I	25	^Y	41	)	57	9	73	I	89	Y	105	i	121	y
10	^J	26	^Z	42	*	58	:	74	J	90	Z	106	j	122	z
11	^K	27	^[	43	+	59	;	75	K	91	[	107	k	123	{
12	^L	28	^\	44	,	60	<	76	L	92	\	108	l	124	|
13	^M	29	^]	45	−	61	=	77	M	93	]	109	m	125	}
14	^N	30	^^	46	.	62	>	78	N	94	^	110	n	126	~
15	^O	31	^_	47	/	63	?	79	O	95	_	111	o	127	^?

ASCIIコードの制御文字

- 表示できる文字が図形文字，それ以外は制御文字.
- 制御文字は出力装置に動作を要求する.
  - 例：改行文字（^J，¥n）は端末ディスプレイに改行を要求する.
  - 例：エスケープ文字で始まるエスケープシーケンス ^[$B は JIS (iso-2022-jp)による日本語表示を要求する.

文字	意味	char	文字	意味	char	文字	意味	char
^@	Null（ヌル文字）	'\0'	^K	Vertical Tab （垂直タブ）	'\v'	^V	Synchronous Idle	
^A	Start of Header		^L	Form Feed （改ページ）	'\f'	^W	End of Transmission Block	
^B	Start of Text		^M	Carriage Return （復帰）	'\r'	^X	Cancel	
^C	End of Text		^N	Shift Out		^Y	End of Medium	
^D	End of Transmission (EOF)		^O	Shift In		^Z	Substitute	
^E	Enquire		^P	Data Link Escape		^[	Escape（エスケープ）	
^F	Acknowledge		^Q	Device control 1		^\	File Separator	
^G	Bell	'\a'	^R	Device control 2		^]	Group Separator	
^H	Back Space （後退）	'\b'	^S	Device control 3		^^	Record Separator	
^I	Horizontal Tab（水平タブ）	'\t'	^T	Device control 4		^_	Unit Separator	
^J	Line Feed（改行）	'\n'	^U	Negative Acknowledge		^?	Delete（削除）	

で囲った制御文字だけ覚えれば十分

Cヘッダファイル <stdint.h>

後で出てきます

- C言語の `int`型が何バイトかはプラットフォーム依存.
  - `short`, `long`, `long long` 型も同じ.
  - バイト数を意識するアセンブリ言語では不便.
- <stdint.h> で固定長の整数型を使う.
  - <stdint.h>はC言語規格C99以降の標準ヘッダファイル.
  - 例：`uint32_t` は「符号なし32ビット整数」.

型名	説明	型名	説明
int8_t	符号あり 8 ビット	uint8_t	符号なし 8 ビット
int16_t	符号あり 16 ビット	uint16_t	符号なし 16 ビット
int32_t	符号あり 32 ビット	uint32_t	符号なし 32 ビット
int64_t	符号あり 64 ビット	uint64_t	符号なし 64 ビット
intptr_t	符号ありポインタ用	uintptr_t	符号なしポインタ用
intmax_t	符号あり最大幅	uintmax_t	符号なし最大幅
(int64_t など 64 ビットの型は 64 ビット幅の整数型がある場合のみ)			

符号なし整数(unsigned integer)のビット表現

- 2進数の各桁をそのままビットで表現する.
- 例：2を8ビットの符号なし整数で表現すると,
  $2 = 10_2$ なので，00000010になる.
  - 余った上位ビットに0を入れることに注意.

8ビット符号なし整数のビット表現

ビット表現	10 進値	16 進値	ビット表現	10 進値	16 進値
00000000	0	0_{16}	10000000	128	80_{16}
00000001	1	1_{16}	10000001	129	81_{16}
00000010	2	2_{16}	10000010	130	82_{16}
⋮	⋮	⋮	⋮	⋮	⋮
01111101	125	$7D_{16}$	11111101	253	FD_{16}
01111110	126	$7E_{16}$	11111110	254	FE_{16}
01111111	127	$7F_{16}$	11111111	255	FF_{16}

符号なし整数の範囲

- 固定長の整数の範囲は有限.
  - 例：8ビット符号なし整数が表現できる範囲は0～255.
  - 一般にnビット符号なし整数の範囲は0～$(2^n - 1)$
  - （当たり前だが）負の数は表現できない.

	ビット表現				10 進値	16 進値
8 ビットの最小値				00000000	0	0_{16}
8 ビットの最大値				11111111	$255 = 2^8 - 1$	FF_{16}
16 ビットの最小値			00000000	00000000	0	0_{16}
16 ビットの最大値			11111111	11111111	$65535 = 2^{16} - 1$	$FFFF_{16}$
32 ビットの最小値	00000000	00000000	00000000	00000000	0	0_{16}
32 ビットの最大値	11111111	11111111	11111111	11111111	$4294967295 = 2^{32} - 1$	$FFFFFFFF_{16}$

オーバーフローとアンダーフロー

- オーバーフロー(overflow)，上位桁あふれ
  - 「表現できる範囲の上限」を上回ること．
- アンダーフロー(underflow)，下位桁あふれ
  - 「表現できる範囲の下限」を下回ること．

注：「整数のアンダーフロー」という言葉は間違いで，
「負のオーバーフロー」と言うべき，という人達もいる

符号なし整数のオーバーフローの例

- 8ビット符号なし整数の255に1を足すと，結果は0.

```
#include <stdio.h>
#include <stdint.h>
int main (void)
{
    uint8_t x = 255;
    x++;
    printf ("%d¥n", x);
}
```

```
% gcc overflow.c
% ./a.out
0
```

符号なし整数のアンダーフローの例

- 8ビット符号なし整数の0から1を引くと，結果は255.

```
#include <stdio.h>
#include <stdint.h>
int main (void)
{
    uint8_t x = 0;
    x--;
    printf ("%d¥n", x);
}
```

```
% gcc underflow.c
% ./a.out
255
```

符号なし整数のオーバーフロー・アンダーフローの計算結果

- 8ビット符号なし整数の場合：
  - 0～255の範囲に収まるように256で割った余りを計算結果とする（オーバーフロー・アンダーフローを単純に無視する）
- nビット符号なし整数の場合：
  - 0～ $(2^n - 1)$ の範囲に収まるように, 2^n で割った余りを計算結果とする.

- 「2^n で割った余りを計算結果とする」の別の言い方.
  - 「モジュロ (modulo) 2^n を取る」
  - 「2^n を法とする」

キャリーとボロー

- キャリー(carry)＝繰り上げ
- ボロー(borrow)＝繰り下げ

例：8ビット符号なし整数で 255+1 を計算すると，キャリーが発生.

例：8ビット符号なし整数で 0-1 を計算すると，ボローが発生.

- 符号なし整数のオーバーフローやアンダーフローは，x86-64ではキャリーフラグ(CF)で検出可能.
  - cf. 符号ありの場合はオーバーフローフラグ(OF)で検出

先取り情報

符号あり整数(signed integer)のビット表現

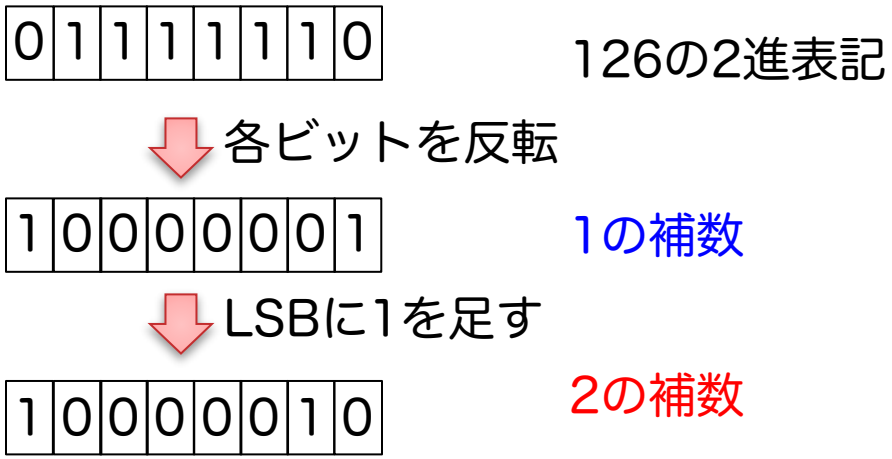
- 負の数は2の補数をビット表現とする.
  - MSBは符号ビットになる．0なら正に，1なら負になる.
- 例：8ビット符号あり整数の場合,
  - 前半の0～127を正の数，後半の128～255を負の数にする.
  - 126に対する2の補数は130 (=256-126)なので,
    130の2進表記 10000010 を -126 のビット表現とする.

ビット表現	10 進値	16 進値	ビット表現	10 進値	16 進値
00000000	0	0_{16}	10000000	-128	-80_{16}
00000001	1	1_{16}	10000001	-127	$-7F_{16}$
00000010	2	2_{16}	10000010	-126	$-7E_{16}$
⋮	⋮	⋮	⋮	⋮	⋮
01111101	125	$7D_{16}$	11111101	-3	-3_{16}
01111110	126	$7E_{16}$	11111110	-2	-2_{16}
01111111	127	$7F_{16}$	11111111	-1	-1_{16}

8ビット符号あり整数のビット表現

2の補数 (2's complement)

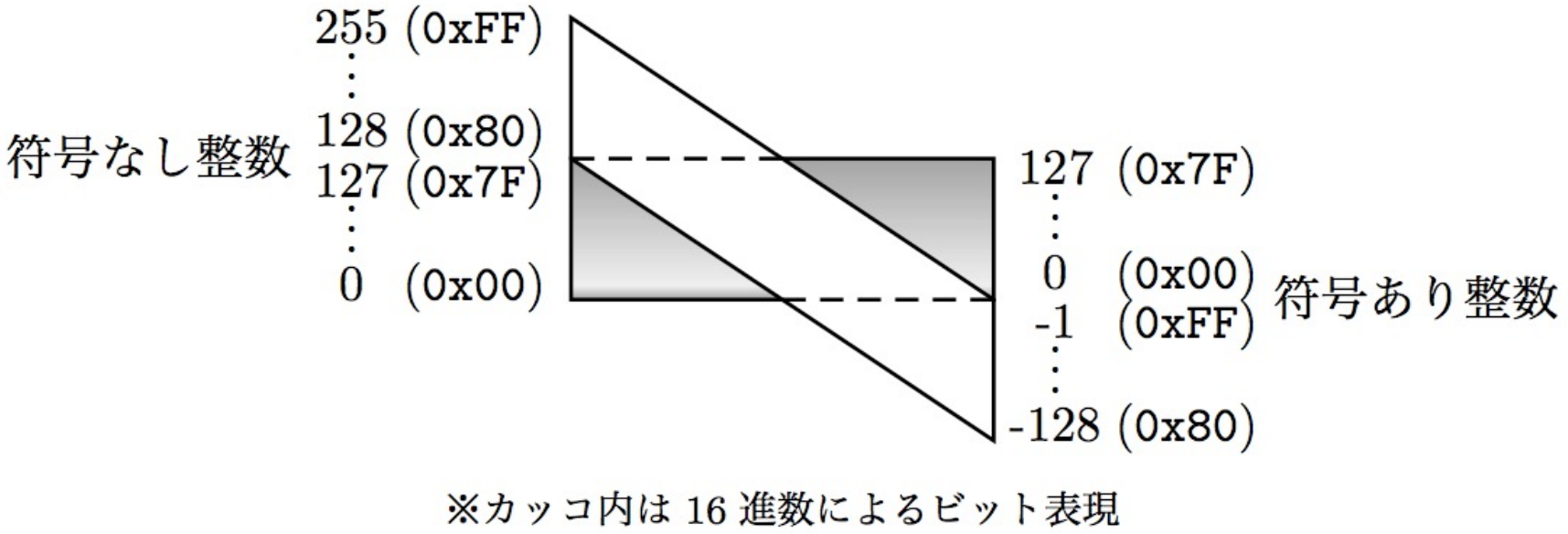
- 2の補数は「各ビットを反転して1を足す」で得られる.
- 例：126に対する2の補数を計算：



- 一般に，nビットの場合，k （ただし$0 < k < 2^n$ ）に対する2の補数は$2^n - k$.
  - 例：8ビットの126に対する2の補数は $2^8 - 126 = 130$
  - 「足して 2^n になる数」が2の補数（nビットの場合）.

符号なし整数と符号あり整数の関係

- 8ビットの場合,
  - 符号なし整数の128～255の範囲のビット表現を,
    符号あり整数の-128～-1の範囲にシフトしたことになる.



※カッコ内は 16 進数によるビット表現

符号あり整数の範囲

-128～128でないのは0があるから

- 固定長の整数の範囲は有限.
  - 例：8ビット符号あり整数が表現できる範囲は-128～127.
  - 一般にnビット符号あり整数の範囲は $(-2^n/2) \sim (2^n/2-1)$.
    - $(-2^{n-1}) \sim (2^{n-1}-1)$ と書いても同じ.
    - $(-2^n/2) \sim (2^n/2-1)$ の方が, 2^n 通りの数を半分ずつ正負で使っていることが分かりやすい.

	ビット表現				10 進値	16 進値
8 ビットの最小値				10000000	$-128 = -2^8/2$	-80_{16}
8 ビットの最大値				01111111	$127 = 2^8/2-1$	$7F_{16}$
16 ビットの最小値			10000000	00000000	$-32768 = -2^{16}/2$	-8000_{16}
16 ビットの最大値			01111111	11111111	$32767 = 2^{16}/2-1$	$7FFF_{16}$
32 ビットの最小値	10000000	00000000	00000000	00000000	$-2147483648 = -2^{32}/2$	-80000000_{16}
32 ビットの最大値	01111111	11111111	11111111	11111111	$2147483647 = 2^{32}/2-1$	$7FFFFFFF_{16}$

2の補数の利点

- 2の補数を使うと加算器で減算できる.
- 例：
  - 31-126は，31と-126のビット表現の加算で計算できる.
    つまり 00011111+10000010=10100001=-95.
  - 31＋130の計算結果 161=10100001 と一致することに注意.
    - 130は（8ビットの場合の）126に対する2の補数.
- 実際，x86-64は整数が符号ありか符号なしかを区別せず加算している．計算結果はどちらに対しても正しい.

符号なし整数の加算	ビット表現	符号あり整数の加算
31	00011111	31
+130	+ 10000010	−126
161	10100001	−95

符号あり整数のオーバーフローの例

- 8ビット符号あり整数の64に64を足すと，
  結果は-128.

```
#include <stdio.h>
#include <stdint.h>
int main (void)
{
    int8_t x = 64;
    x += 64;
    printf ("%d¥n", x);
}
```

```
% gcc overflow2.c
% ./a.out
-128
```

C言語規格上，符号あり整数のオーバーフローは未定義動作．
つまり，符号あり整数をオーバーフローさせてはいけない．

キャリーとボロー（再び）

先取り情報

- 符号あり整数のオーバーフローやアンダーフローは，x86-64ではオーバーフローフラグ(OF)で検出．
  - cf. 符号なしの場合はキャリーフラグ(CF)で検出可能

```
  01000000
+ 01000000
----------
0 10000000
```

64+64はオーバーフローする．
しかし，キャリーはなし．

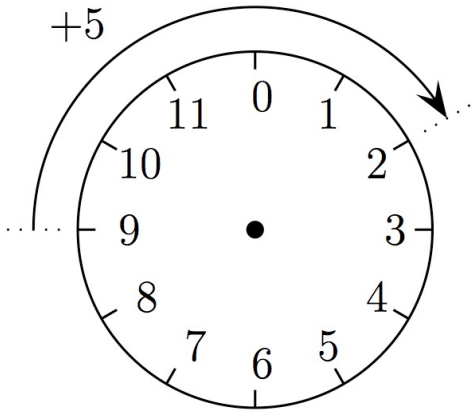
```
  11111111
+ 11111111
----------
1 11111110
```

-1-1はオーバーフローしない．
しかし，キャリーはあり．

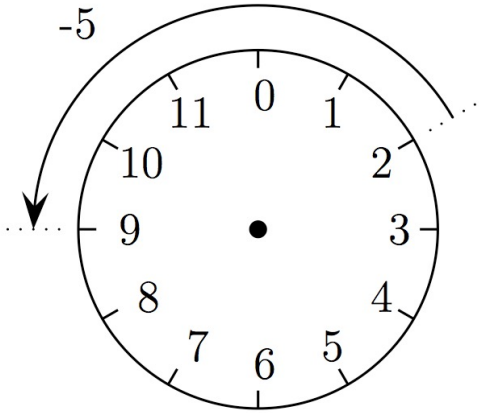
つまり，符号あり整数ではオーバーフローとキャリーは関係ない．

2の補数による負数表現を時計で説明（1）

- オーバーフローとアンダーフロー
  - 時計は0時から11時までしか表記できない（12時を0時として）
  - それ以外の時間は0時～11時の範囲に収まるように，
    12で割った余りとする．



9時に5時間を足すと14時だが
オーバーフローなので12で割った
余りである2時とする．
14時と2時は同じ．

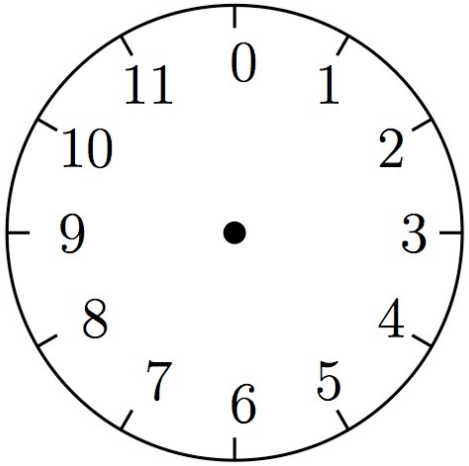


2時から5時間を引くと-3時だが
アンダーフローなので12で割った
余りである9時とする．
-3時と9時は同じ．

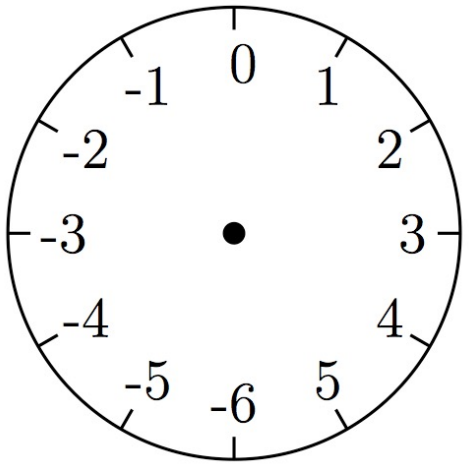
2の補数による負数表現を時計で説明（2）

- 2の補数で，なぜ負の整数を表現できるのか．
  - 時計の世界では「足して12になる数」が2の補数．
    - 「足し算すると表現できる数の上限を1つ超える数」が2の補数．
  - 例：3に対する2の補数は9．（つまり9で-3を表す）
    ここで，（12で割った余りの下では）9時=-3時になるから．
    9時は0時の「3時間前」なので，この表現は自然．

正確には
12の補数



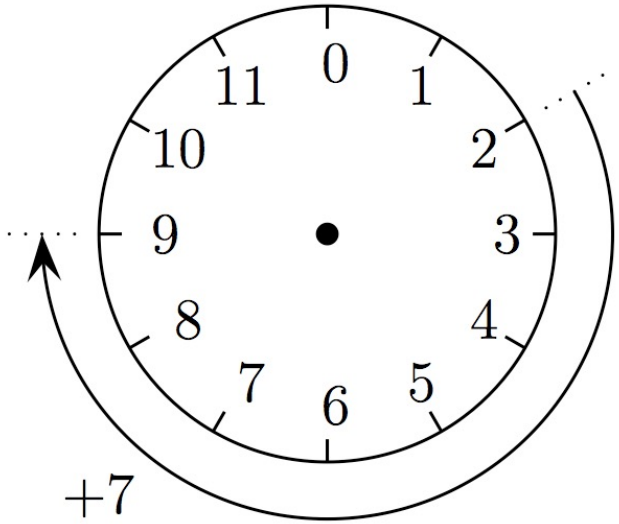
符号なし整数で時計を表現



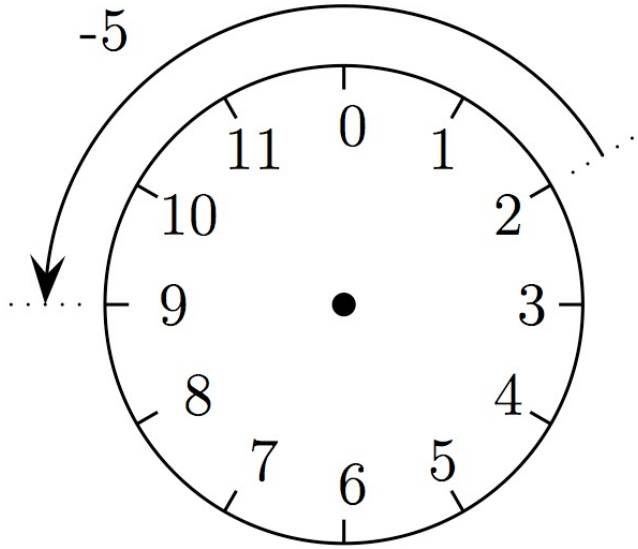
符号あり整数で時計を表現

2の補数による負数表現を時計で説明（3）

- 2の補数を使うと，なぜ足し算で引き算ができるのか.
  - 時計の世界では「7時間足すこと」＝「5時間引くこと」だから.



2時に7時間足すと9時になる



2時から5時間を引くと9時になる

整数の計算に注意（1）

- 正の整数同士の和が負になる

```
#include <stdio.h>
#include <stdint.h>
int main ()
{
    int32_t i1 = 10*10000*10000; // 10億
    int32_t i2 = 15*10000*10000; // 15億
    int32_t i3 = i1 + i2; // オーバーフロー発生
    printf ("%d¥n", i3);  // -1794967296
}
```

```
% gcc int-anomaly1.c
% ./a.out
-1794967296
```

C言語規格上の未定義動作

整数の計算に注意（2）

符号あり整数で
表現可能な最小値.

- 絶対値を計算できない整数がある.

```
#include <stdio.h>
#include <stdint.h>
int main ()
{
    int8_t  i1 = -128;
    int16_t i2 = -32768;
    int32_t i3 = -2147483648;
    i1 = -i1; i2 = -i2; i3 = -i3;
    printf ("%d, %d, %d¥n", i1, i2, i3);
}
```

```
% gcc int-anomaly2.c
% ./a.out
-128, -32768, -2147483648
```

整数の計算に注意（3）

- signedとunsignedを混ぜると直感に反する結果になる.
  - 0UのUは「符号なし」を示す，定数につける接尾語(suffix).
  - この場合，-1を符号なしにまず変換する．4294967295になる.
    4294967295<0U の比較をして偽(0)になる.
    - 違う型を持つ値の演算では，まず型を同じにするため（通常の算術型変換）.

```
#include <stdio.h>
#include <stdint.h>
int main ()
{
    int32_t  i1 = -1;
    uint32_t i2 = 0U;
    printf ("%d¥n", i1 < i2);
}
```

```
% gcc int-anomaly3.c
% ./a.out
0
```

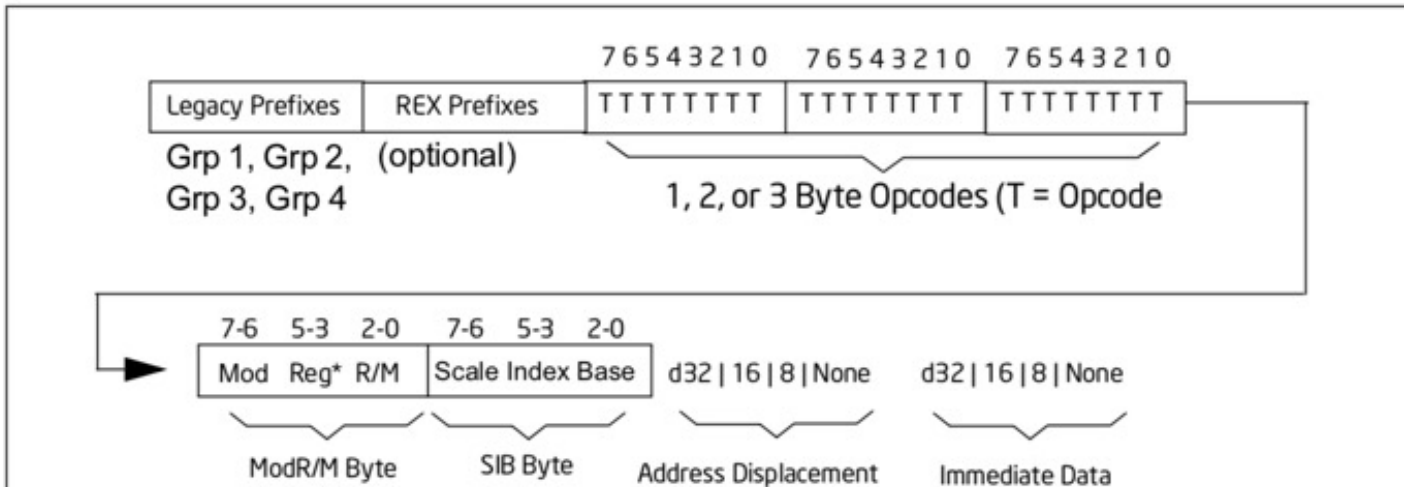
機械語命令の符号化

インテル64およびIA-32 インテルアーキテクチャー・ソフトウェア・デベロッパーズ・マニュアル

- 機械語命令も2進数で符号化される.
  - ただし，符号化方法はCPUの種類ごとに異なる.
  - x86-64の符号化方法はマニュアルに記載あり.
    - 具体的な符号化方法は覚えなくてＯＫ.
- x86-64の機械語命令の長さは可変長.
  - 最小で1バイト，最大で15バイト．CISCの特徴
  - cf. RISCのCPUの多くは固定長.

それなりに複雑.
覚える必要なし.

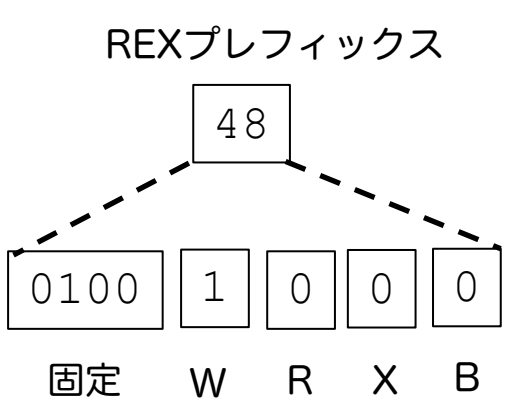
x86-64の機械語命令フォーマット

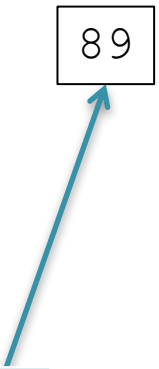


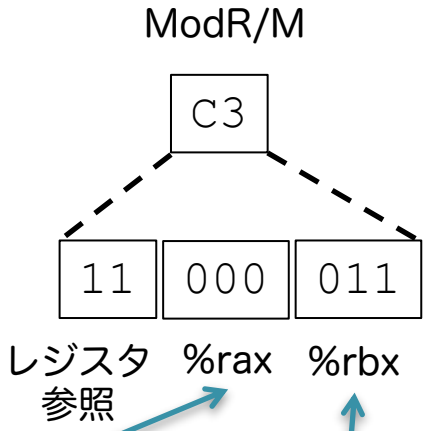
機械語命令の符号化（例）

```
.text
movq %rax, %rbx
```

```
% gcc -c movq.s
% objdump -d movq.o （表示を一部略）
Disassembly of section .text:
 0:  48 89 c3      mov %rax,%rbx
```







ニモニック	オペコード
movq *r64, r/m64*	REX.W 89
movq *r/m64, r64*	REX.W 8B
movl *imm32, r32*	B8 +rd
movq (movabsq) *imm64, r64*	REX.W B8 +rd
movq *imm32, r/m64*	REX.W C7 /0

レジスタ	%rax	%rcx	%rdx	%rbx
符号	000	001	010	011
レジスタ	%rsp	%rbp	%rsi	%rdi
符号	100	101	110	111

ゼロ拡張

- ゼロ拡張 (zero extension)
  - 上位ビットを0で埋めてビット列を大きくする変換.
- 符号なし整数をゼロ拡張すると，値は変化しない.
  - 例：2バイトの符号なし整数65535をゼロ拡張で4バイトに変換しても，値は変化しない.

データサイズ	ビット表現				値
2バイト			11111111	11111111	65535
4バイト	00000000	00000000	11111111	11111111	65535

常に0を埋める

符号拡張（1）

- 符号拡張 (sign extension)
  - 上位ビットを元データのMSBで埋めてビット列を大きくする変換.
  - つまり，正の場合は0を，負の場合は1を上位ビットに埋める.
- 符号あり整数を符号拡張すると，値は変化しない.

データサイズ	ビット表現	値
2バイト	01111111 11111111	32767
4バイト	00000000 00000000 01111111 11111111	32767

正の場合は0を埋める

データサイズ	ビット表現	値
2バイト	11111111 11111111	−1
4バイト	11111111 11111111 11111111 11111111	−1

負の場合は1を埋める

符号拡張（2）

- 符号あり整数をゼロ拡張すると，値が変化する．

データサイズ	ビット表現				値
2バイト			11111111	11111111	−1
4バイト	00000000	00000000	11111111	11111111	65535

ゼロ拡張で0を埋める

movzlq命令とmovslq命令

先取り情報

- movzlq命令
  - ロングをクアッドにゼロ拡張して，データをコピーする命令.
  - move with zero extension from long to quad
- movslq命令
  - ロングをクアッドに符号拡張して，データをコピーする命令.
  - move with sign extension from long to quad

- 通常，符号なし整数にはmovzlqを使い，
  符号あり整数にはmovslqを使う.

注意：movzlqとmovslqはAT&Tスタイルのニモニック.
Intelスタイルではmovsxdとmovzxd.

切り詰め（切り捨て）（1）

- 切り詰め (truncation)
  - 上位ビットを捨てて，ビット列を小さくする変換.
- 符号の変化
  - 符号あり整数を切り詰めると，正負が変わることがある.
    - （当たり前ですが）符号なし整数では正負は変わらない.

	ビット表現				値
	00000000	00000001	10000110	10100000	100000
上位2バイトを切り詰め			10000110	10100000	-31072

切り詰め（切り捨て）（2）

```
#include <stdio.h>
#include <stdint.h>
int main (void)
{
    int32_t i = 100000;
    int16_t s = i;
    printf ("%d¥n", s);
}
```

```
% gcc trunc.c
% ./a.out
-31072
```

符号あり整数を切り詰めて，
正負が変わった例．

アラインメント

- アラインメント（境界調整） (alignment)
  - 特定のアドレス（例：4の倍数のアドレス）にデータを配置すること．CPUの都合で必要となる．
  - アラインメント制約はABIの一部．
- アラインメントしないと
  - x86-64では実行速度が遅くなる．
  - 他のCPUではバスエラーなどの例外（実行時エラー）が生じることも．

先取り情報

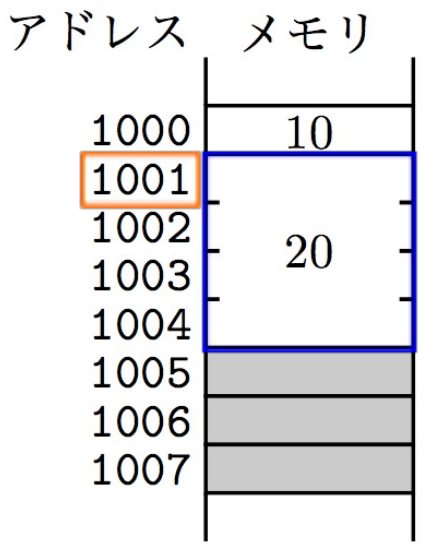
- アセンブラ命令 `.align` で実現できる．
  - `.align 4` は次のデータを「4の倍数のアドレス」に配置する．
  - `.p2align 4` は「2^4=16の倍数のアドレス」に配置する．

アラインメント（例）

多バイト長データのアドレスは
先頭バイトのアドレス.

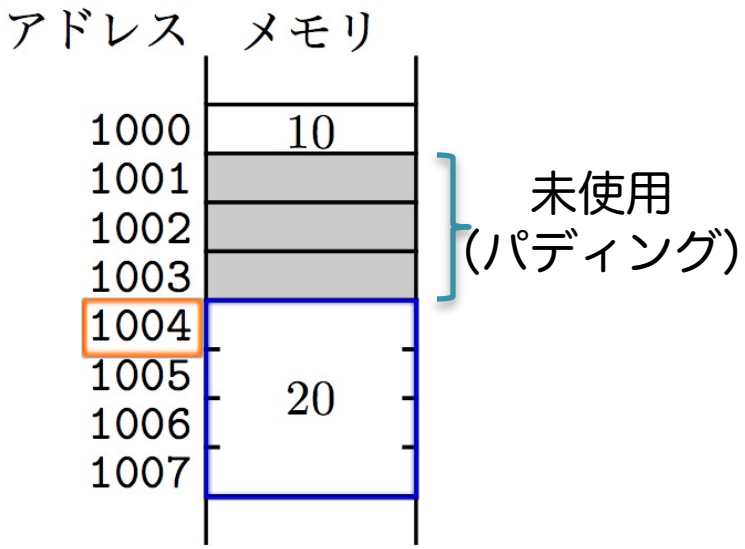
- 「4バイトのデータは4の倍数のアドレスに置く」というアラインメント制約がある場合：

アラインメント制約を
満たしていない例



× 1001 番地は 4 の倍数ではない

アラインメント制約を
満たしている例



○ 1004 番地は 4 の倍数

Mac OS X(x86-64)のアラインメント制約

型	C言語の型	サイズ	アラインメント制約
整数	char	1バイト	1の倍数のアドレス
	short	2バイト	2の倍数のアドレス
	int	4バイト	4の倍数のアドレス
	long	8バイト	8の倍数のアドレス
	long long	8バイト	
ポインタ	ポインタ型	8バイト	
浮動小数点数	float	4バイト	4の倍数のアドレス
	double	8バイト	8の倍数のアドレス
	long double	16バイト	16の倍数のアドレス

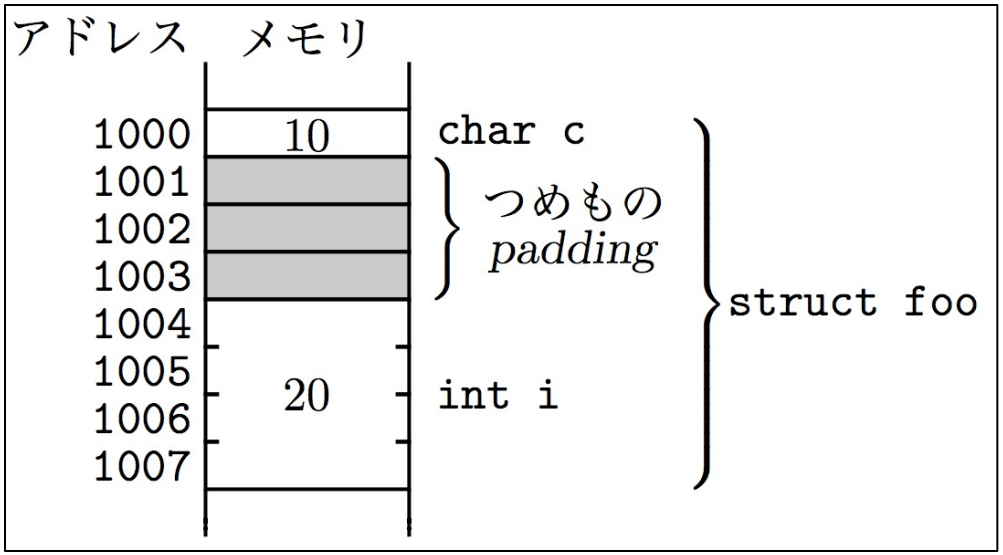
出典：System V Application Binary Interface
AMD64 Architecture Processor Supplement

パディング（1）

- パディング (padding)
  - アラインメント制約を満たすために生じる「すき間」のこと.
  - 例：構造体のメンバー間のパディング.
  - アセンブラ命令.space（.skipでも同じ）や.alignで実現.
    - .space 3 は，3バイト空けて，次のデータを配置する.

```
struct foo {
    char c;
    int i;
};
struct foo x = {10, 20};
```

```
.globl _x
  .data
  .p2align 2
_x:
  .byte  10
  .space 3
  .long  20
```



3バイトのパディングを配置.

パディング（2）

先頭に入ることはない

- 構造体の最後にパディングが入ることもある.
  - C言語の配列はメモリ上で連続する必要がある.
  - 構造体の配列が，連続性とアラインメント制約を満たすため.

```
struct foo {
    int i;
    char c;
};
struct foo x = {10, 20};
```

```
sizeof(struct foo)
```

は8となる.

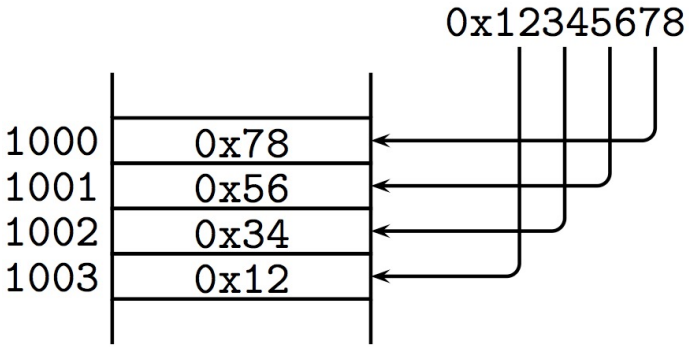
```
.globl _x
  .data
  .p2align 2
_x:
  .long  10
  .byte  20
  .space 3
```

構造体の最後に
3バイトのパディング.

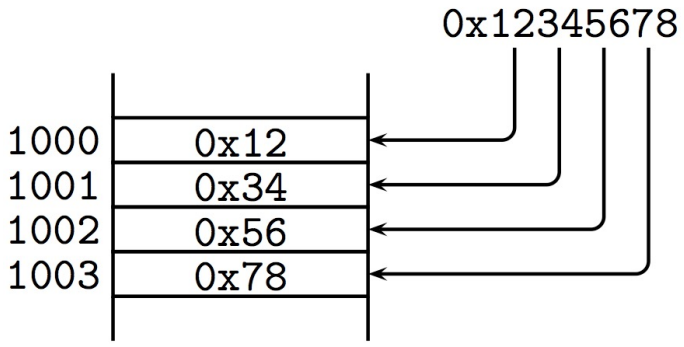
先取り
情報

バイトオーダとエンディアン

- バイトオーダ (byte order)
  - 多バイト長のデータをバイト単位に格納する順序.
- 代表的なバイトオーダ
  - リトルエンディアンとビッグエンディアン.
  - x86-64はリトルエンディアン.



リトルエンディアン.
最下位バイトから格納.



ビッグエンディアン.
最上位バイトから格納.

リトルエンディアンの注意

- リトルエンディアンでは，16進ダンプすると，多バイト長データは逆順に表示される．

```
int main (void)
{
    return 0x12345678;
}
```

```
% gcc -c little-rev.c
% od -t x1 little-rev.o
0000000  ce fa ed fe 07 00 00 00 03 00 00 00 01 00 00 00
 (中略)
0000400  55 89 e5 83 ec 08 b8 78 56 34 12 c9 c3 00 00 00
```

0x12345678が
逆順に表示されている