Heltal og Two’s Complement

Heltal og Integers

- Repræsentation af positive tal og negative heltal?
  - Udnytte alle bits
  - Nem/hurtig at regne med,
  - Pæne, gennemskuelige matematiske regne-regler
- Fortegns-bit
- Ones' complement
  - +0,-0 ??
  - a+(-a)≠0
- Two's complement
- Forskudt talområde (floats)

```
int x =  15213;
int y = -15213;
```

```
Unsigned int x =  15213;
```

```
01000100001001010101O
????????
```

Se evt. denne video:
https://www.youtube.com/watch?v=Z3mswCN2FJs

Positive heltal (`Unsigned Int`)

- Med *w*-bits kan vi indkode 2^w forskellige værdier:

Unsigned

"Binary-to-unsigned"

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

1 0 1 0

w_3, w_2, w_1, w_0

$$1\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 0\cdot 2^0 =$$
$$8 + 0 + 2 + 0 = 10$$

Integers

- Med *w*-bits kan vi indkode 2^w forskellige værdier

Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two's Complement

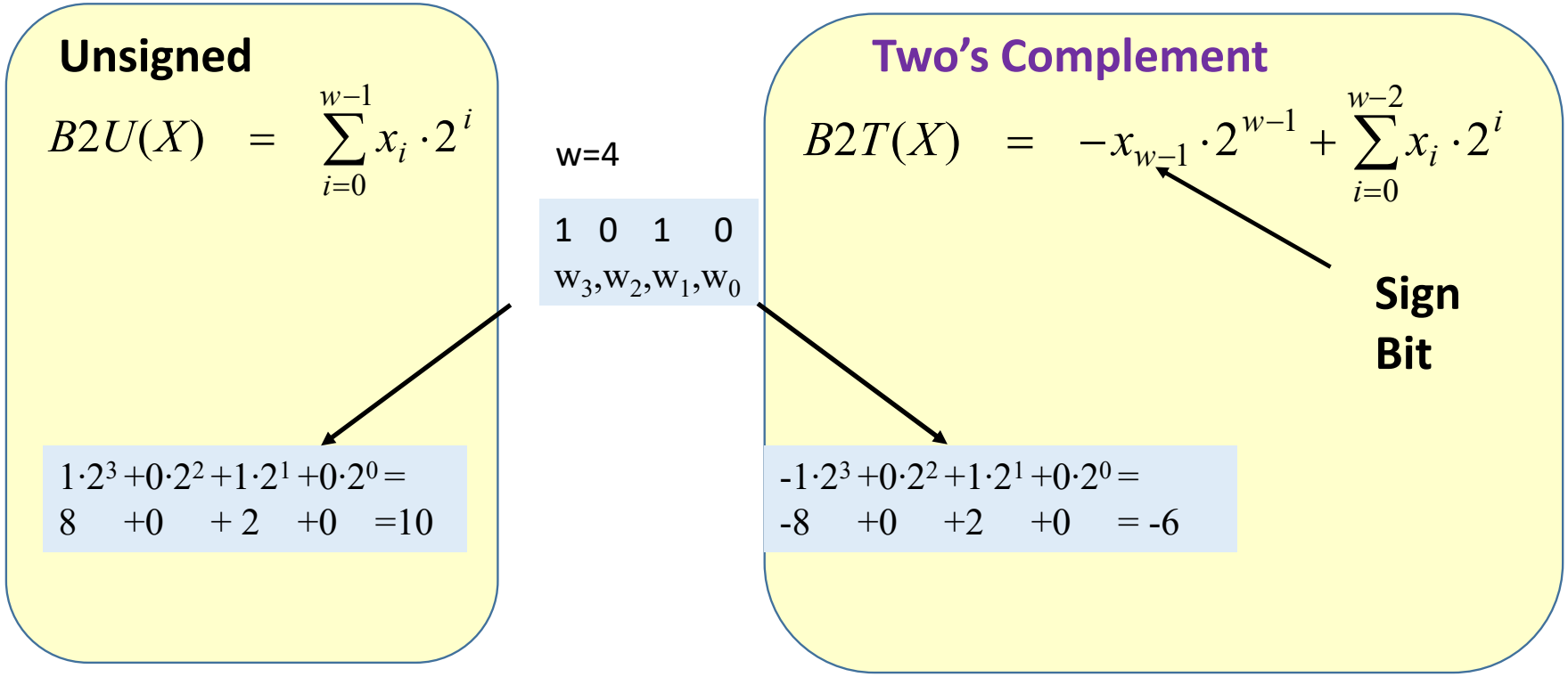
$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

"Binary-to-two's comp"

Sign Bit

Integers (signed)

- Med *w*-bits kan vi indkode 2^w forskellige værdier



Integers

```
short int x =  15213;
short int y = -15213;
```

Two's Complement

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Sign Bit

- Eksempel: (C short, 2 bytes)

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
y	-15213	C4 93	11000100 10010011

$-1\cdot2^{15}+1\cdot2^{14}+1\cdot2^{10}+1\cdot2^{7}+1\cdot2^{4}+1\cdot2^{1}+1\cdot2^{0} =$
$-32768+16384+128+16+2+1 = -15213$

- Fortegns bit (Sign bit)
  - I Two's complement, angiver mest betydende bit fortegnet
    - 0 for ikke-negative
    - 1 for negative
    - Bidrager med stor negativ vægt.

(negation (-x) kan fås fra x ved at beregne komplementet til x og addere 1).

Numeriske intervaller (Ranges)

Unsigned Værdier

- *UMin*=0
  000…0
- *UMax*= $2^{w-1}+\ldots+2^1+2^0 = 2^w - 1$
  111…1

Værdier i Two's Complement

- *TMin* = -2^{w-1}
  100…0
- *TMax* = $2^{w-1} - 1$
  011…1
- Minus 1
  111…1

Værdier for *W* = 16

	Decimal	Hex	Binary
UMax	65535	FF FF	11111111 11111111
TMax	32767	7F FF	01111111 11111111
TMin	-32768	80 00	10000000 00000000
-1	-1	FF FF	11111111 11111111
0	0	00 00	00000000 00000000

Intervaller for andre ord-størrelser

	W			
	8	**16**	**32**	**64**
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

Observationer

- $|TMin| = TMax + 1$
  - Asymmetrisk interval
  - $UMax = 2 * TMax + 1$

C Programmering

- #include <limits.h>
- Erklærer konstanterne, fx.,
  - ULONG_MAX
  - LONG_MAX
  - LONG_MIN
- Værdierne er platforms specifikke

C#/JAVA har tilsvarende konstanter.

Unsigned & Signed Værdier

X	B2U(*X*)	B2T(*X*)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

- Ækvivalens
  - Samme indkodning for positive værdier
- Unikke
  - Hver bit mønster repræsenterer éen unik integer værdi
  - Hvert representabel integer har en unik bit indkodning
- $\Rightarrow$ Vi kan invertere afbildingen
  - $U2B(x) = B2U^{-1}(x)$
    - Find bit mønster, som giver x i unsigned int
  - $T2B(x) = B2T^{-1}(x)$
    - Find bit mønster, som giver x i two's comp.

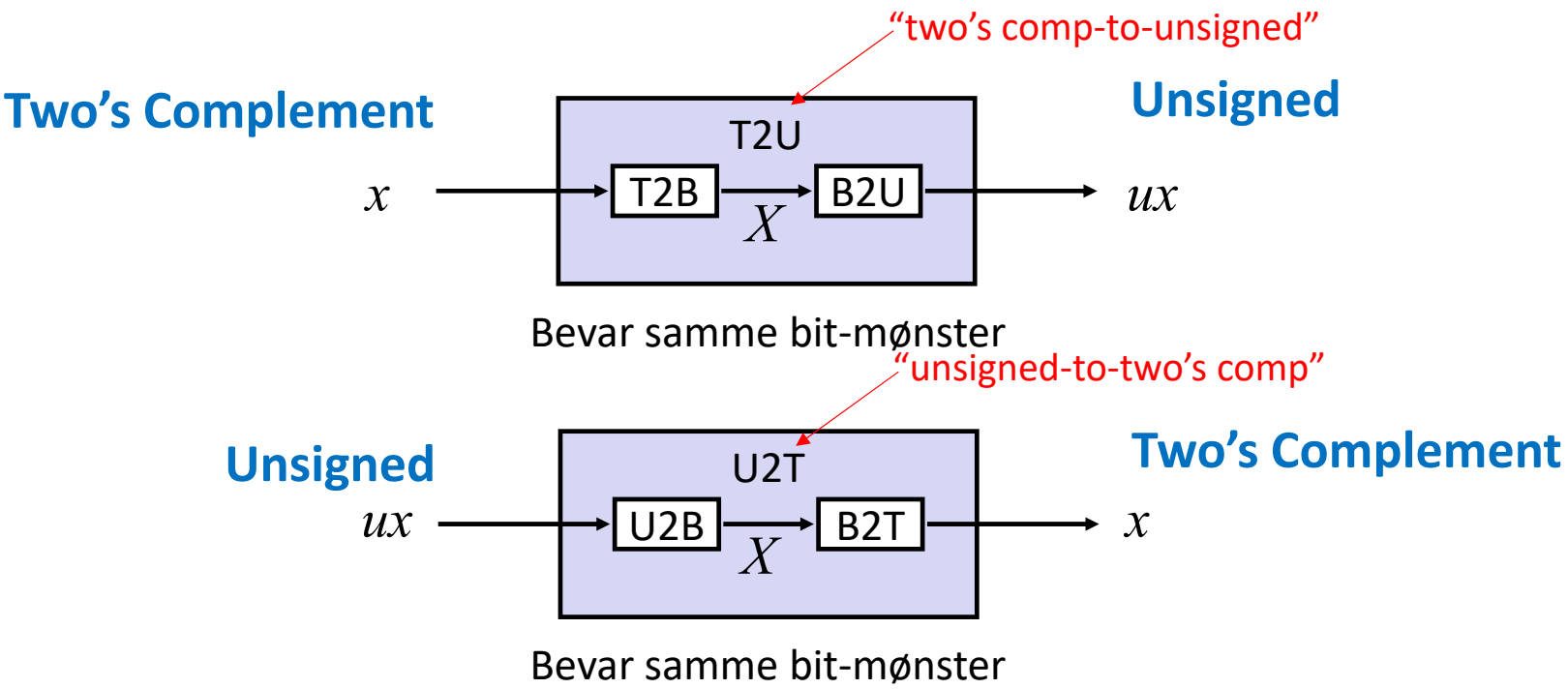
Bemærking: Præcision

- En maskine med max ordlængde w kan godt operere på større tal end ordlængden direkte tillader
  - Fx 8-bit micro-controller kan godt addere int_32 (men ikke i eét hug)
  - Compiler genererer den nødvendige serie af instruktioner

- Der findes program biblioteker, der understøtter beregninger med vilkårlig præcision
  - Begrænset af RAM.
  - Sprog: Ruby/Python integers
    - Prøv det.
  - IKKE indbygget i maskinen!
    - Væsentligt langsommere

Kontertering mellem
Signed and unsigned

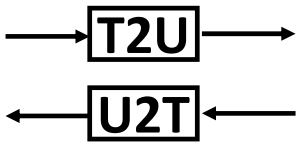
Konvertering imellem Signed & Unsigned



- Mapping imellem unsigned og two's complement værdier:
  - bevar bit repræsentation og genfortolk
  - $x \neq ux$

Konvertering Signed ↔ Unsigned

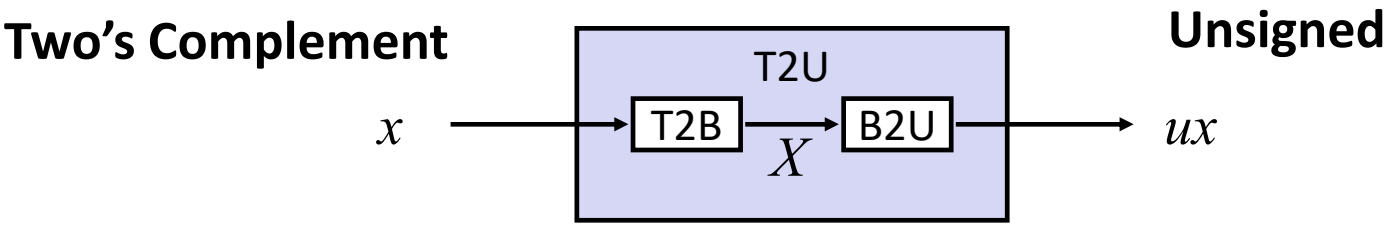
Bits	Signed	Unsigned
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	-8	8
1001	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15



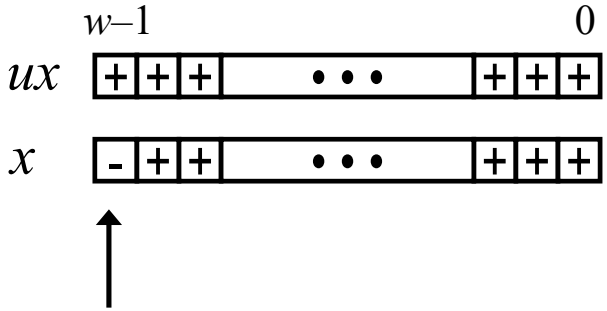
Konvertering Signed ↔ Unsigned

Bits	Signed		Unsigned
0000	0		0
0001	1		1
0010	2		2
0011	3	=	3
0100	4		4
0101	5		5
0110	6		6
0111	7		7
1000	-8		8
1001	-7		9
1010	-6		10
1011	-5	+/- 16	11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15

Konvertering mellem Signed og Unsigned



Bevar samme bit-mønster



Stor negativ vægt: -2^{w-1}

bliver

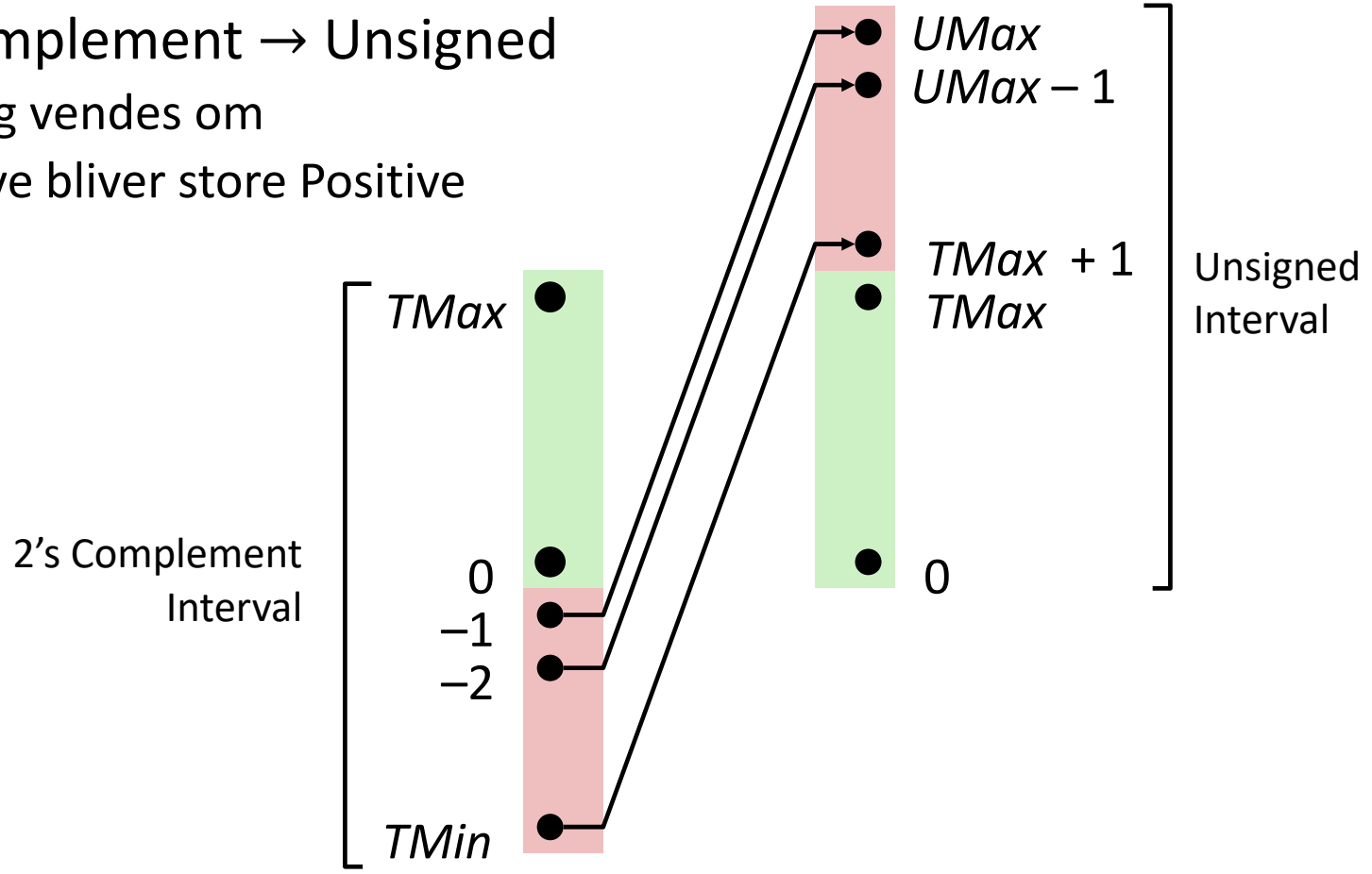
Stor positiv vægt: 2^{w-1}

$$ux = \begin{cases} x & x \geq 0 \\ x + 2^{w} & x < 0 \end{cases}$$

Difference: $2*2^{w-1}=2^{w}$

Visualisering af konvertering

- Two's Complement → Unsigned
  - Ordning vendes om
  - Negative bliver store Positive



Type konvertering i ”C”

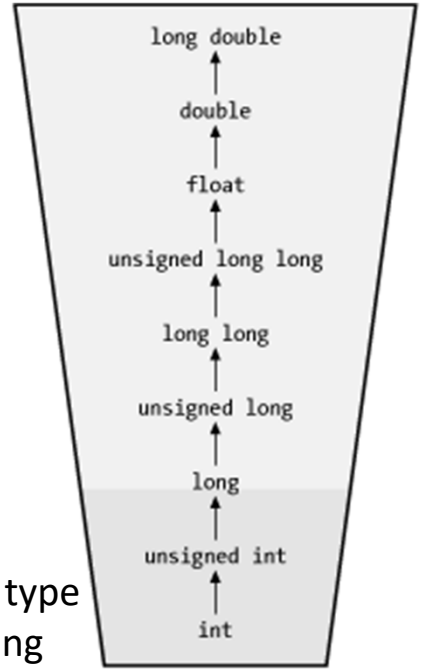
Onde men vigtige teknikaliteter.

- Konstanter
  - Er som udgangspunkt signed integers
  - Unsigned, gennemtvinges med endelsen “U”
  - 0U, 4294967259U
- Casting
  - Type konvertering (casting) mellem signed and unsigned gør det same som U2T og T2U

```
int tx, ty;
unsigned ux, uy;
tx = (int) ux;
uy = (unsigned) ty;
```

- Implicit typekonvertering (casting) forekommer
  - i udtryk, især blandede
  - ved assignment, og
  - procedure kald

```
tx = ux;
uy = ty;
```



Jfv C sprogets regelsæt for type konvertering og promovering

Nogle sprog, fx Java, har kun ints.

Overraskelser ved kontertering

Konklusion: Do not mix!

PP2.21

- Evaluering af udtryk
  - I et udtryk med en blanding af unsigned og signed ints, bliver ***signed værdier implict konverteret til unsigned!!!!!!!***
  - Det inkluderer sammenligninger: **<, >, ==, <=, >=**
  - Eksempel med *W* = 32: **TMIN = -2,147,483,648 , TMAX = 2,147,483,647**

Konstant$_1$	**Konstant$_2$**	**Relation**	**Evaluering**
0	0U	==	unsigned
-1	0	<	signed
-1	0U	>	unsigned!
2147483647	-2147483648	>	signed
2147483647U	-2147483648	<	unsigned!
-1	-2	>	signed
(unsigned) -1	-2	>	unsigned
2147483647	2147483648U	<	unsigned
2147483647	(int) 2147483648U	>	signed!

Eksempel på sikkerhedshul

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

- Svarende til kode fundet i FreeBSD's implementation af **`getpeername`**
- Der er hære af "smarte" mennesker, som søger sådanne sikkerhedshuller

Typisk Brug

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, MSIZE);
    printf("%s\n", mybuf);
}
```

Ondsindet brug

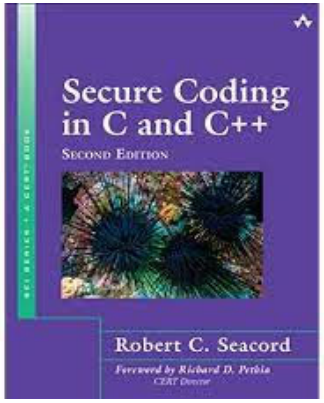
```
/* Declaration of library function memcpy */
void *memcpy(void *dest, void *src, size_t n);
/* stddef.h:size_t -> unsigned int */
```

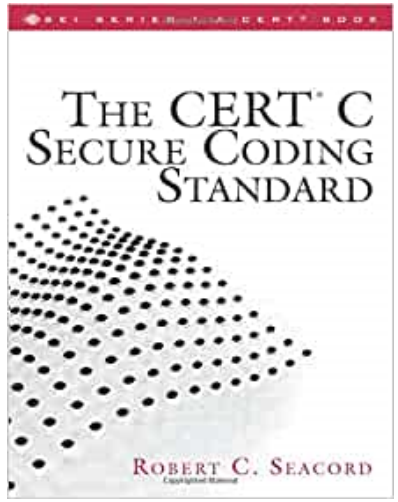
```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[1000*MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    . . .
}
```

copy_from_kernel(mybuf,-528);
int len=-528; //=4294966768 unsigned
memcpy(mybuf,kbuf, 4294966768);

Programmering af sikre systemer:





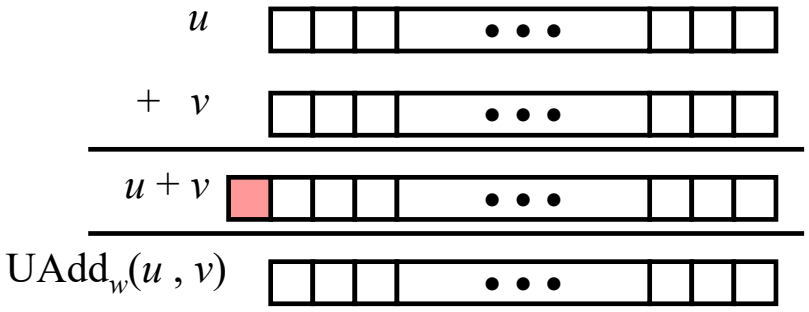
Aritmetik og over-/under-flow

Unsigned Addition

Operander: *w* bits

Sande Sum: *w*+1 bits

Menten (Carry) kasséres: *w* bits



- Overløb, men veldefineret
  - Implementerer Modular Aritmetik
    - $UAdd_w(u , v) = (u + v) \mod 2^w$
  - Bevarer normale regne-regler for addition af heltal (”Abelsk gruppe”)!

Tallet bliver for stort til repræsentation med det givne antal bits!

```
Fx, w=4, 8+11=19
  1000
+ 1011
------
 10011 //19 mod 2^4 = 3
```

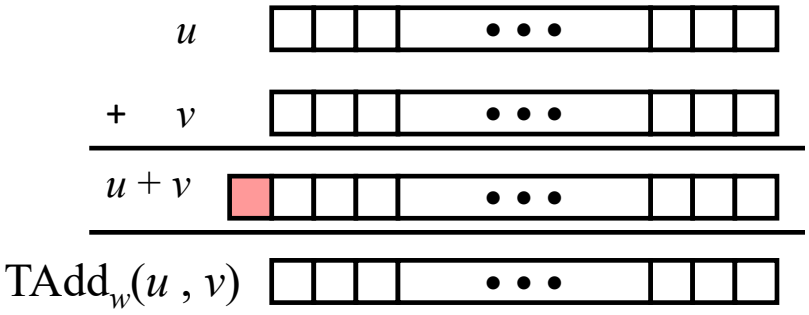
I DSP bruges også ”saturated” aritemtik:
- bevarer største værdi: 1000+1011=1111
- Resultat bliver numerisk tættest på sande sum
- Distributive og associative love gælder ej

Two's Complement Addition

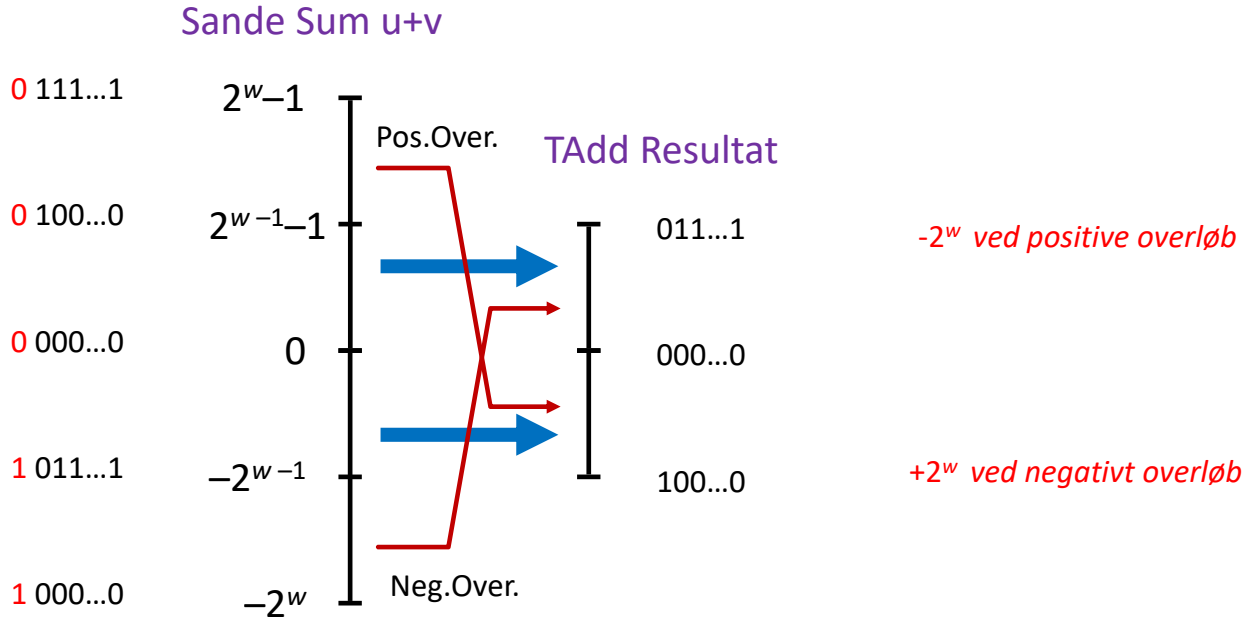
Operander: *w* bits

Sande sum: *w*+1 bits

Menten (Carry) kasséres: *w* bits



- Overløb (Overflow)
  - Sande sum kræver *w*+1 bits
  - Msb mistes
  - Resterende bits behandles som Two's comp. Integer
  - "Abelsk gruppe"

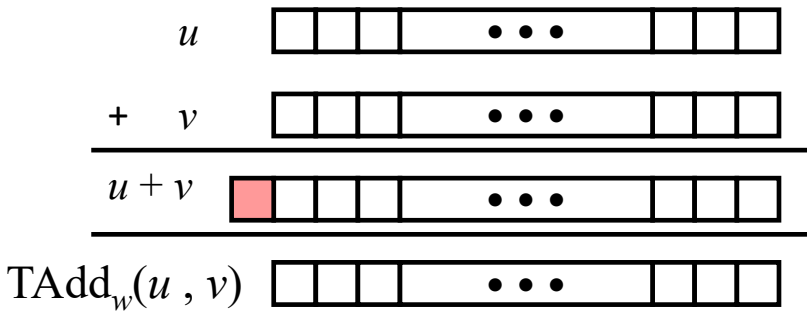


Two's Complement Addition

Operander: *w* bits

Sande sum: *w*+1 bits

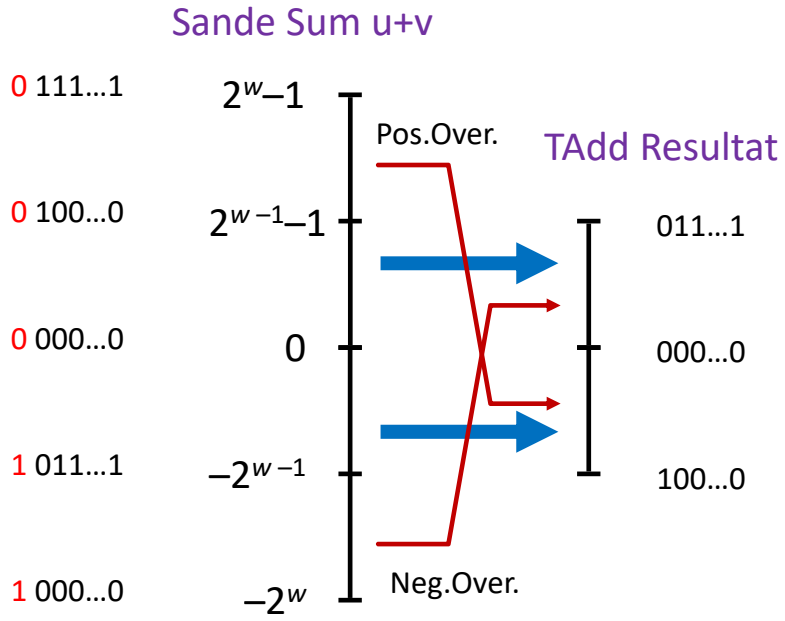
Menten (Carry) kasséres: *w* bits



SIGNED Positivt overløb

```
Fx, w=4,7+5=12
  0111
+ 0101
------
  1100 =-4
NB:12-2^4 = -4 )
```

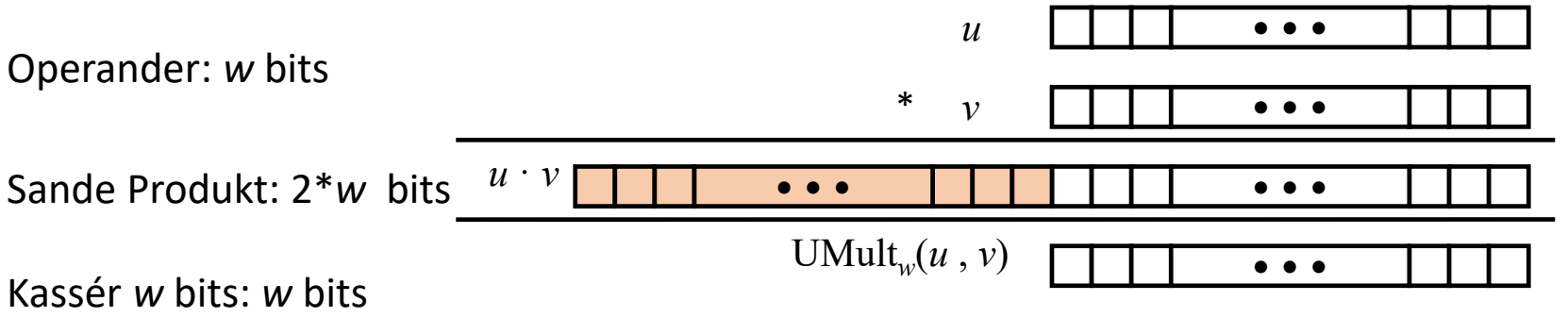
- Overløb (Overflow)
  - Sande sum kræver *w*+1 bits
  - Msb mistes
  - Resterende bits behandles som Two's comp. Integer
  - "Abelsk gruppe"



SIGNED Negativ overløb

```
Fx, w=4,-7+-5=-12
  1001
+ 1011
------
 10100 =4
//NB: -12+2^4 = 4
```

Unsigned Multiplikation i C



- Standard Multiplikations-operation
  - Ignorér de *w* mest betydende bits
- Effekt: Modular Aritmetik

  $\text{UMult}_w(u , v) = u \cdot v \mod 2^w$

Shifting

“Høk æ Hak” operationer (Shift-operations)

høk æ hak' er vendelbomål for at flytte sig en smule

PP2.16

- Venstre skifte: x << y
  - bit-vektoren x forskydes y positioner mod venstre
    - Ekstra bits til venstre smides væk
    - Fyldes med 0 til højre
- Højre skifte: x >> y
  - bit-vektoren x forskydes y positioner mod højre
    - Ekstra bits til højre smides væk
  - Logisk skift
    - Fyldes med 0 til venstre
  - Aritmetisk skift
    - Gentag msb til højre

w=8

Argument x	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	00011000

Argument x	10100010
<< 3	00010000
Log. >> 2	00101000
Arith. >> 1	11010001
Arith. >> 2	11101000

Bemærkning:

- Der er kun det antal pladser som der er!!
- Præcedens!
  - Advarsel! De binære operatorer har lav præcedens
    - a << 3 + b << 2 samme som a << (3 + b) << 2
    - a & 3 != 0 samme som a & (3 != 0)
  - Sæt PARANTESER!
- Skifte er udefineret er udefineret når antallet af pladser, der skal skiftes, overstiger antal bit i data-typen.
  - `Int32_t x; x>>33;` er udefineret!
  - x >> m implementeres typisk som x >> (m % word_size)
  - Defineret sådan i Java.
- En C-compiler anvender "typisk" aritmetisk shift ved signed værdi og logisk ved unsigned; pas på promoveringsregler!
- I Java findes operatoren '>>>' til logisk/unsigned right shift

Fra Eksamen F17:

Antag w=8 bit: beregn 222_{10}<<3

222= 11011110 <<3

~~110~~ 11110000 = **F0**

IKKE 6F0

Potens-af-2 Multiplikation vha. skiftning (shift)

- Multiplikation med operand, som er en potens af 2:
  - **`u << k`** giver **`u * `** 2^k
  - Både signed og unsigned
- FX.: k=1, u=4
  - $4*2^1$ unsigned: `0100 <<1 = 1000 =8`
  - $-4*2^1$ signed: `1100 <<1 = 1000 =-8`
- Eksempler
  - **`u << 3 == u * 8`**
  - **`(u << 5) - (u << 3) == (u*32)-(u*8) == u * 24`**
- De fleste maskiner udfører shift og add hurtigere end multiplikation
  - Compiler genererer selv denne form for kode automatisk

Potens-af-2 division vha. skiftning (Shift)

- Beregning af kvotient, hvor divisor er potents-af-2
- Unsigned
  - **u >> k** giver $\lfloor \mathbf{u} \ / \ 2^k \rfloor$
  - Bruger logisk skifte

	Matematisk Division	Beregnet	Hex	Binary
x	15213	15213	3B 6D	00111011 01101101
x >> 1	7606.5	7606	1D B6	00011101 10110110
x >> 4	950.8125	950	03 B6	00000011 10110110
x >> 8	59.4257813	59	00 3B	00000000 00111011

```
printf("%u\n", 15213/16);
-> 950
```

  - Signed
    - C udtrykket **u >> k** giver $\lfloor \mathbf{u} \ / \ 2^k \rfloor$
    - Bruger aritmetisk shift
      - `1000 >> 1 = 1100` // -8/2=-4
    - Afrundingsproblem: -3/2 burde give -1, ikke -2,
      - `1101 >>1 = 1110`=-2
    - Fix: se lærerbogen s. 142

```
printf("%d\n", -3/2);
-> -1
```

Resumé

- Two's complement
- Overløb
- Grundlæggende regler ved konvertering mellem Signed ↔ Unsigned
  - Bit mønstret bevares
  - Men genfortolkes som den nye type
  - Kan have overraskende effekter: addition or subtraktion af 2^w

- Hvis udtryk indeholder både signed og unsigned integers
  - **`int`** konverteres til **`unsigned`**!!