Computer Arkitektur
Bits, Bytes, og Integers

Forelæsning 1
Brian Nielsen

Credits to
Randy Bryant & Dave O'Hallaron (CMU)

Kursusgang 0-2: Tal repræsentation

Introduktion

- Computer Arkitektur?
- Realiteterne
- Kursus mål
- Kursus format

Binær repræsentation

- Eksempel-system
  - X86-64, Linux, gcc
- Binær, Hex, Dec
- Model for hukk.
  - Words
  - Adresser
  - Endian-ness
- C sproget
- Typer+størrelser
  - Char, Short, Ints, Long, Pointers (8,16,32,64,) …
- Boolsk Algebra
  - True, False
  - $\wedge$ AND, $\vee$ OR, $\neg$ NOT, $\oplus$ XOr
- Logiske Operationer
  - True, False, &&, ||, !
- Bit-vektor Operationer
  - 0,1, &, |, ~, ^

Heltal og Floats

- Unsignede/signed
  - Two's complement
- Operationer
  - Addition
  - Multiplikation
  - Skiftning
  - Konvertering
  - Ekspandering
  - Trunkering
- Floating Point
  - Binære brøker
  - IEEE 754
  - Præcision
  - Afrunding

- **Hvordan organiseres hukommelsen**
- **Hvordan repræsenteres og tal og hvad er de grundlæggende operationer derpå?**
- **Vigtige begrænsninger ved tal-repræsentationerne**

Tal-systemer og notation

Decimal tal

- Base 10
  - Streng af cifrene 0,1,2,3,4,5,6,7,8,9
  - $123_{10} = 1*10^2 + 2*10^1 + 3*10^0$
  - Mest betydende ciffer: tallet med størst vægt (til venstre)
  - Mindst betydende ciffer: tallet med mindst vægt

Binære tal

- Base 2
  - Streng af cifrene 0,1
  - $101_2 = 1*2^2 + 0*2^1 + 1*2^0$

  - Mest betydende ciffer: tallet med størst vægt (til venstre)
    - **Mest betydende bit**
  - Mindst betydende ciffer: tallet med mindst vægt
    - **Mindst betydende bit**

Hexa-decimale tal

- Base 16
  - Streng af cifrene ‘0’ til ‘9’ og ‘A’ til ‘F’
  - $1AF_{16} = 1*16^2 + A*16^1 + F*16^0$
    $=1*16^2 + 10*16^1 + 15*16^0 = 431_{10}$

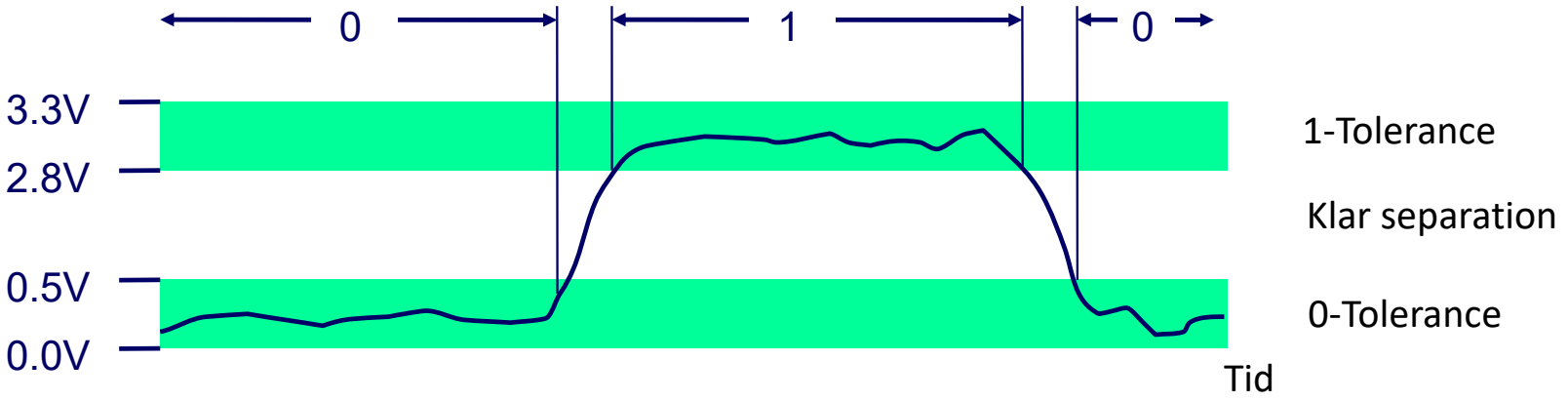
- C-notation: hex tal præfixes med ”0x”
  - $FA1D37B_{16}$ skrives som
    - `0xFA1D37B` , eller `0xfa1d37b`

- Konvertering til binært er NEMT
  - Erstat hvert hex-tal med dets binære værdi (4 bits)
  - 2BA → 0010 1011 1010
- Konvertering fra binært er NEMT
  - Opdel bitstreng i grupper af 4 (højre mod venstre)
  - 1010111010 → 10 1011 1010 → 2BA

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

- Investér tid nu til at blive fortrolig med notationen og konvertering.
- NEM konvertering HEX-BIN

Binær Repræsentation



- Mennesker: 10 fingers, base 10 naturligt.
- Computere: base 2 mere naturlig og pålideligt
- Base16 (Hexa-decimal): mere kompakt og nemt at oversætte til/fra binær

32 bit tal: $FDECBA98_{16}=11111101111011001011101010011000_2$

$FEDCBA9876543210_{16=}1111111011011100101110101001100001110110010101000011001000010000_2$

Notation for Byte værdier

PP2.1
PP2.3

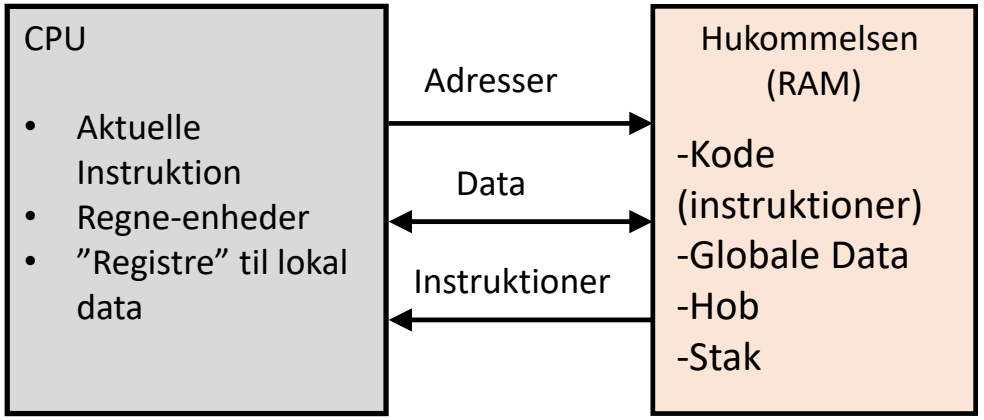
- Byte = 8 bits
- Binært: 00000000_2 to 11111111_2
  - $11000011_2 = 1*2^7+1*2^6+0*2^5+...+1*2^1+1*2^0 = 195_{10}$
  - Most significant bit, least significant bit
- Decimal: 0_{10} to 255_{10}
- Hexa-decimal: 00_{16} to FF_{16}
- Den mindste datatype processoren gemmer i hukommelsen

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

Hukommelses-model

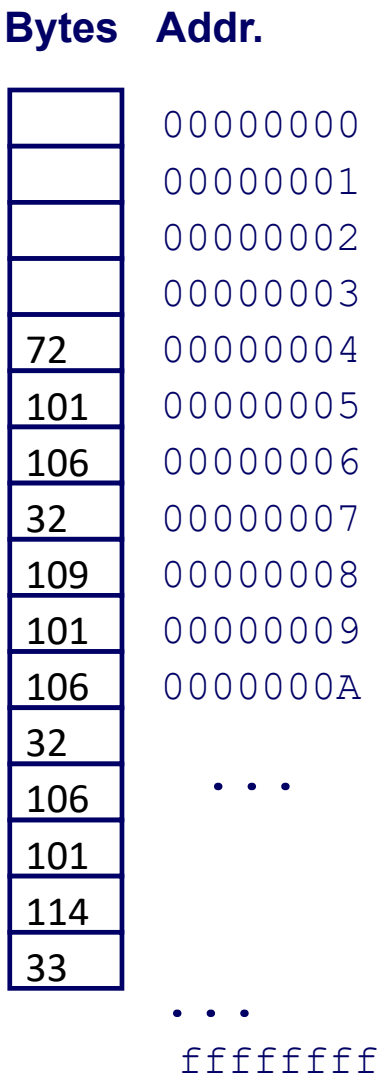
En simpel maskine-model

- Processor (central processing unit)
  - Udfører maskin-instruktioner (fx ”add”)
- Hukommelsen
  - RAM aka. Primær Hukommelsen aka. Arbejdshukommelsen
  - Lagrer instruktionerne for kørende programmer
  - Lagrer kørende programmer’s data
    - Variable, temps, objekter, stak, …
  - Processoren arbejder hele tiden med hukommelsen
    - Læser instruktioner
    - Læser og skriver variable
  - Flygtig



Byte-baseret hukommelsesmodel

- Model: Tænk på hukommelsen som et meget stort byte-array!
  - Index kaldes adresse
  - Hvert element kan indeholde 1 byte
- EX
  - Et ascii tegn fylder 1 byte
    - ”Hej med jer!”
    - 72,101,106, 32,109,101,100,32, 106, 101, 114, 33, 0



Pop Quizz1

I hvilken adresse ligger ”!” placeret?

- Som decimal adresse?
- Som hexa-decimal adresse ?

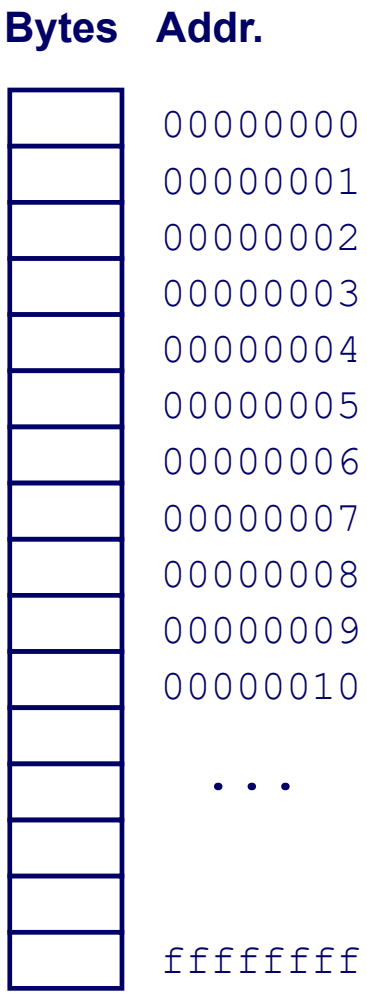
Pop Quizz2

Antag 2^{32} bytes stor hukommelse

- Hvad hedder den første adresse?
- Hvad hedder den største (i dec og hex)?

Byte-baseret hukommelsesmodel

- Processoren udsteder virtuelle adresser
  - Abstraktion: et meget stort byte-array!
  - I virkeligheden flere RAM moduler og et hierarki af forskellige hukommelsestyper
  - Operativ System giver et privat adresserum til hver “proces”
    - Proces= kørende program
    - Program må overskrive egne data, ikke andres
    - OS/HW oversætter virtuelle adresser til fysisk adresse
- Compiler + køretidsomgivelse styrer allokering
  - Bestemmer hvor instruktioner, globale data, stak, hob, mv. gemmes
  - inden for en proces’ eget virtuelle adresserum



Maskin “Ord”/Words

- Word / ord =
  - En samling a bits, som computeren kan arbejde med som en naturlig enhed
  - Typeløs værdi
  - Datatype afhænger af den fortolkning vi lægger ned over ordet: (signed/unsigned) heltal, float, instruktion, pointer,…
  - Længde: typisk 8, 16, 32, 64 bits (potens af 2)

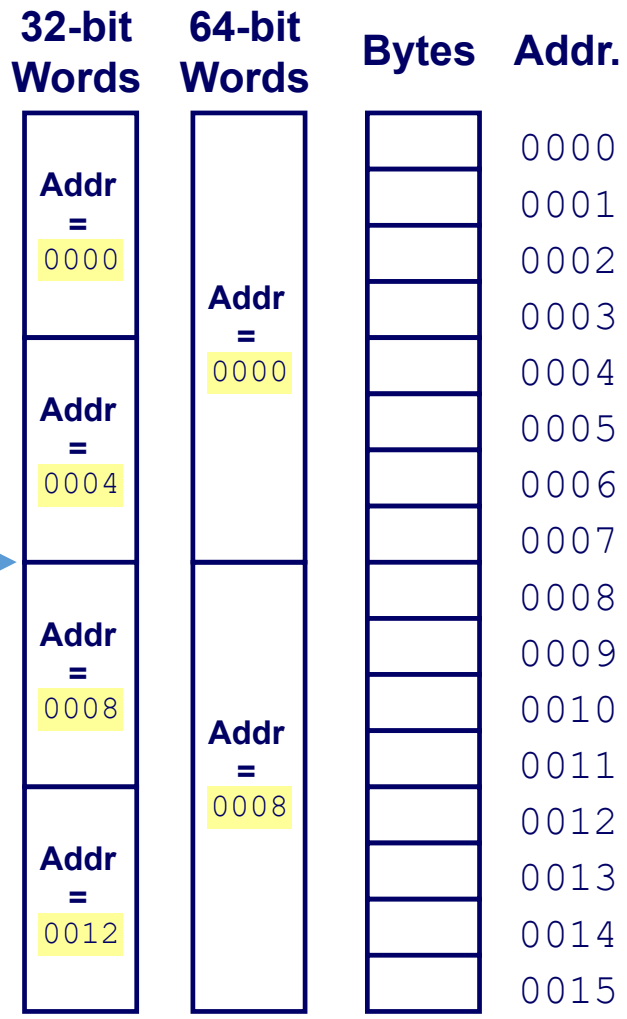
- En arkitekturs ordstørrelse (“**Word** Size”): det største antal bits maskinen normalt kan håndtere i éen operation
  - Læse, skrivning i hukommelsen
  - Maks antal bits i register
  - Største heltal
  - Størrelse på adresser
- Tidligere(PC klasse og op): 32 bits (4 bytes) words
  - Adresser begrænset til 4GB: Blev for lidt til moderne data intensive applikationer
- Nuværende (PC klasse og op): 64 bits (8 bytes) words
  - Potential adresserum $=2^{64} \approx 1.8 \text{ X } 10^{19}$ bytes
  - Nuværende x86-64 processorer sparer og bruger ”kun” 48-bit adresser: 256 Terabytes
- Maskinen understøtter flere (mindre) data-enheder
  - Størrelsen går op i eller er deleligt med ordstørrelsen

Ord-baseret hukommelses organisering

- Adresse angiver placering
  - Adresse på første byte i word
  - Adresse på følgende ord ændres med 4 (32-bit) or 8 (64-bit)

- Pointer= adresse på ordets (start) placering i hukommelsen

Relateret til pointer aritmetik.



Pop Quizz3

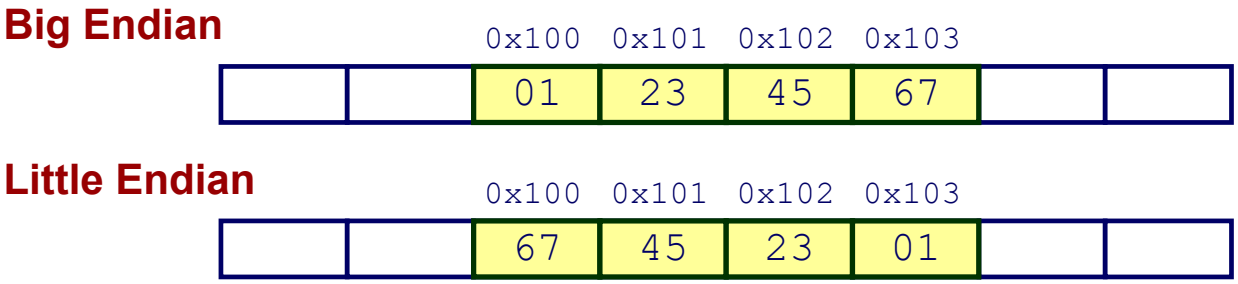
Tallet 4294967234_{10} skal gemmes i hukommelsen som et 32 bit ord, startende i adresse 0.

- I hvilken adresse slutter det?

Byte Ordning

- I hvilken rækkefølge er et ords bytes lagret?
- Konventioner
  - **Big End-ian**: Sun, Motorola Power PC, Internet
    - Mest betydende byte først; mindst betydende byte har højeste adresse
  - **Little End-ian**: x86
    - Mindst betydende byte først; mindst betydende byte has laveste adresse

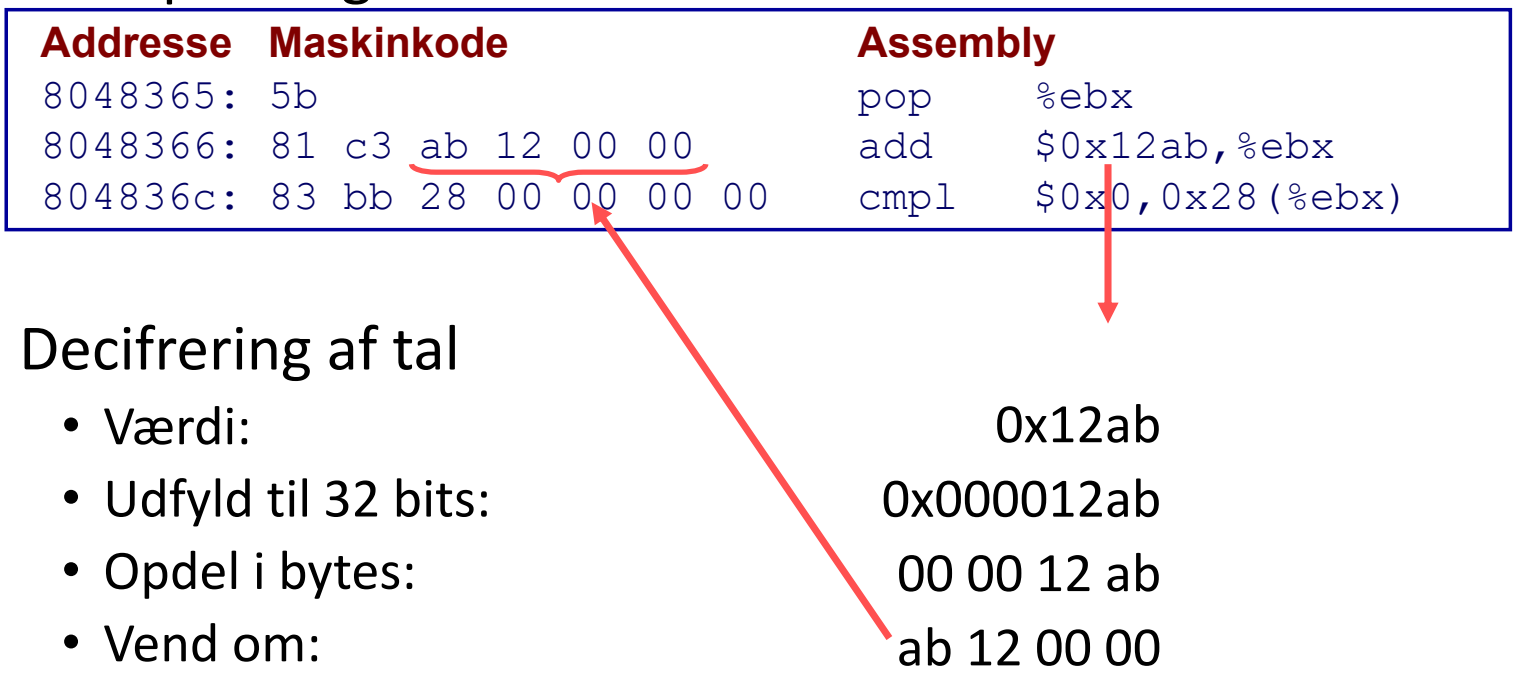
- Ex. Et 32-bit ord (4 bytes) gemmer et heltals-variabel *x* med værdien 0x01234567 (1193046_{10}), lagret med start adresse 0x100:



Læsning af byte-omvendt kode

- Disassemblering
  - Text repræssentation af binær maskinkode
  - Generated af disassembler-program med maskinkode input
- Example Fragment

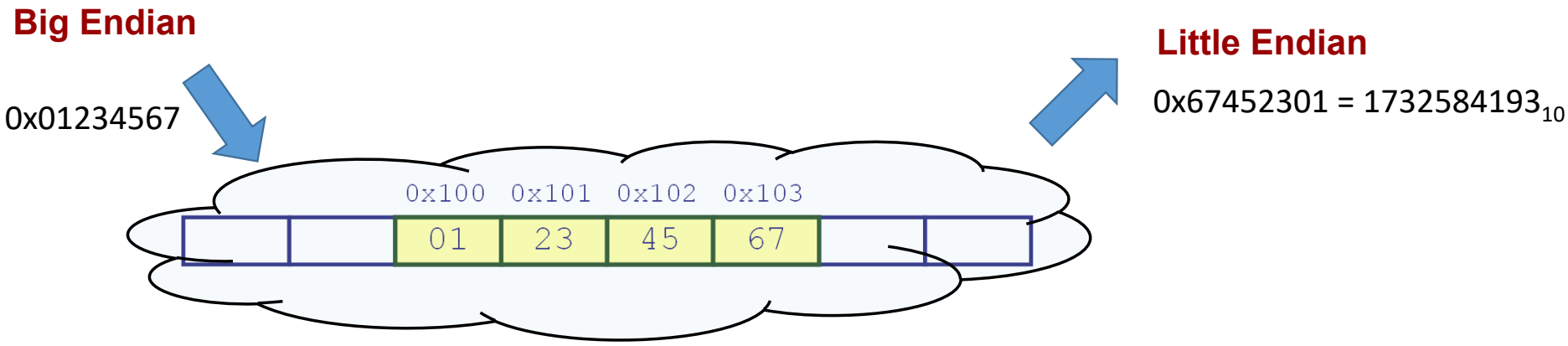
Addresse	Maskinkode	Assembly	
8048365:	5b	pop	%ebx
8048366:	81 c3 ab 12 00 00	add	$0x12ab,%ebx
804836c:	83 bb 28 00 00 00 00	cmpl	$0x0,0x28(%ebx)



- Decifrering af tal
  - Værdi: 0x12ab
  - Udfyld til 32 bits: 0x000012ab
  - Opdel i bytes: 00 00 12 ab
  - Vend om: ab 12 00 00

Kommunikation af binær data

- En big-endian maskine sender værdien 0x01234567 (1193046_{10}) til en little endian maskine.



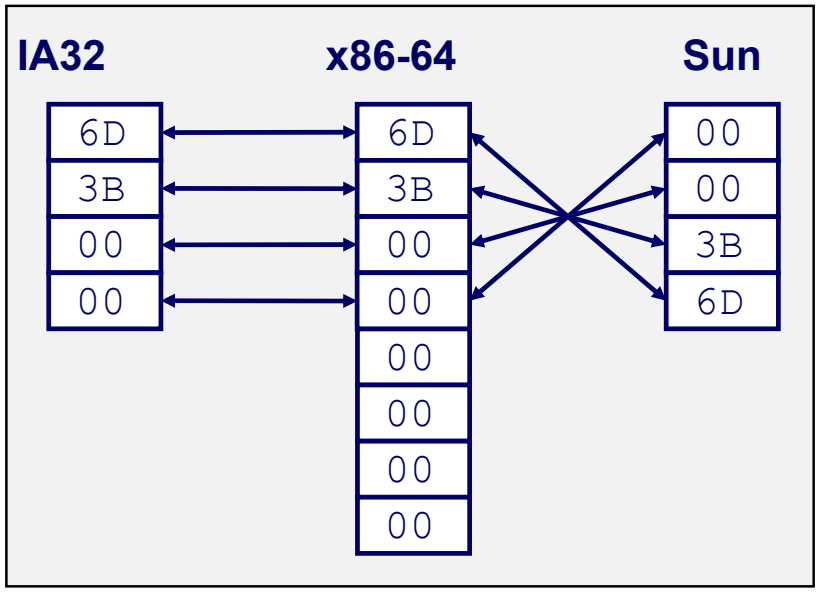
- Forskellige maskiner kan have forskellige ord-størrelse og ”end-ian-ness”.
- Internet-protokollerne bruger ”big-endian” byte ordning.
- Ved kommunikation af binær data, fx over netværk og via filer, skal maskinerne blive ”enige” om data repræsentation, og konvertere ved sender/eller modtager, eller begge steder

Repræsentation af Basale Data typer

Repræsentation af Integers

Decimal: 15213
Binær: 0011 1011 0110 1101
Hex: 3 B 6 D

long int C = 15213;



- Et heltal repræsenteres i dets binære form og lagres som et ”ord”
- Efter maskinens end-ianess
- Nødvendige antal bytes afhængigt af datatypen og maskinens ordstørrelse

Hvad fylder basale C-typer (i bytes)?

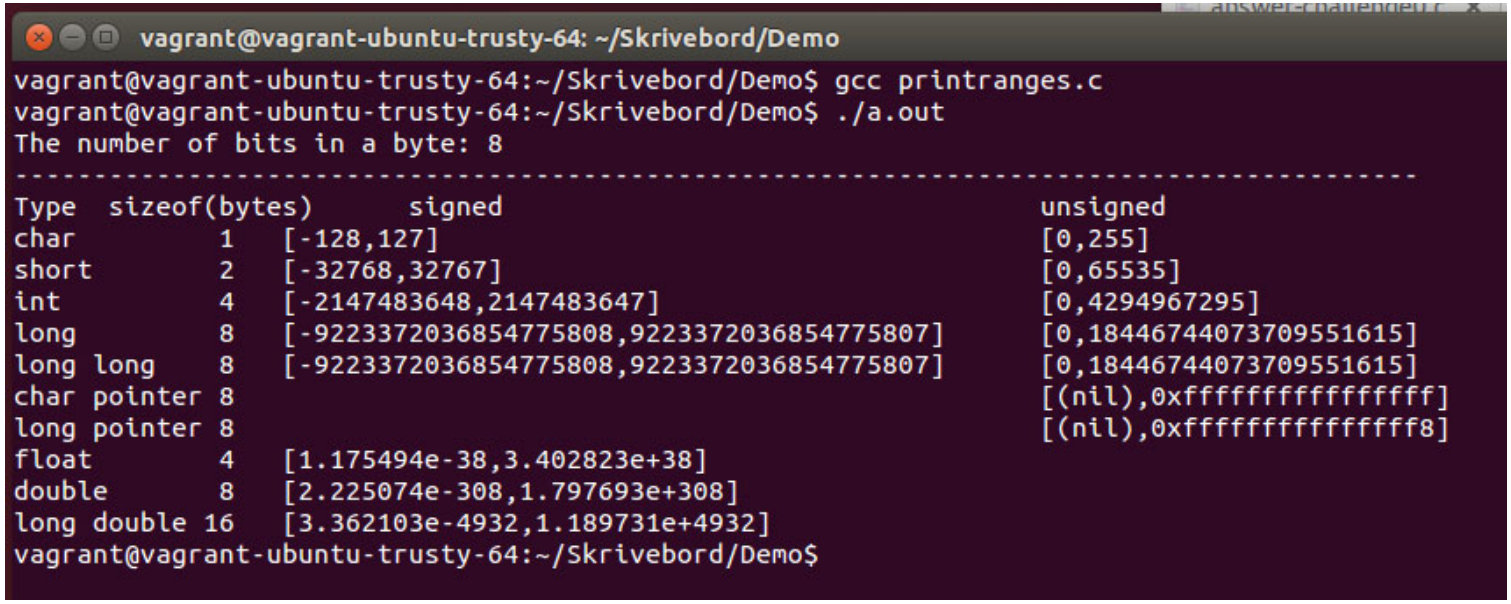
C Data Type	Typical 32-bit	Intel IA32	x86-64	Arduinos		
				ESP8266 (32bit)	Nano-ATMega328p (8-bit)	ATTiny85 (8-bit)
char	1	1	1	1	1	1
short	2	2	2	2	2	2
int	4	4	4	4	2	2
long	4	4	8	4	4	4
long long	8	8	8	8	8	8
float	4	4	4	4	4	4
double	8	8	8	8	4	4
long double	8	10/12 *)	10/16 *)	8	4	4
pointer	4	4	8	4	2	2

NB! Garanterede numeriske intervaller:
https://en.wikipedia.org/wiki/C_data_types

*) 80-bit extended precision type understøttet af x86 hardware

Demo

- **`printranges.c`**



Udskrivning af data repræsentation

Prøv det!

- Casting pointer til unsigned char * giver et byte array

```
typedef unsigned char *pointer;

void show_bytes(pointer start, int len){
  int i;
  for (i = 0; i < len; i++)
    printf("%p\t0x%.2x\n",start+i, start[i]);
  printf("\n");
}
```

Printf directives:
%p: Print pointer
%x: Print Hexadecimal

```
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

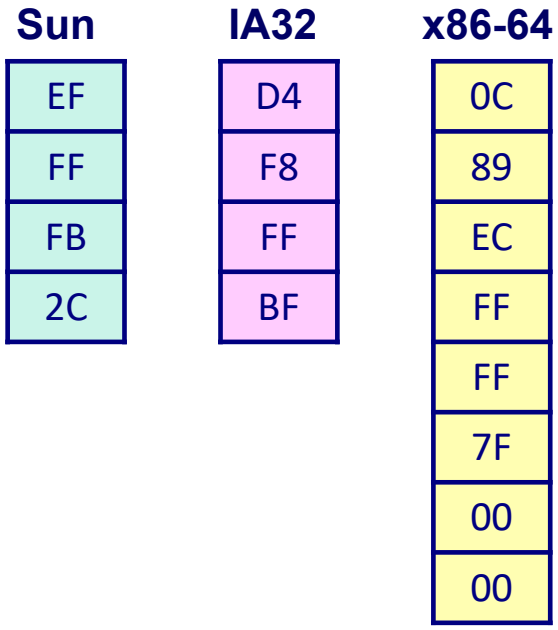
Resultat (Linux, X86-64):

```
int a = 15213;
0x11ffffcb8 0x6d
0x11ffffcb9 0x3b
0x11ffffcba 0x00
0x11ffffcbb 0x00
```

Repræsentation af Pointers

```
int B = -15213;
int *P = &B;
```

- Pointer= adresse på objektets placering i hukommelsen
- Størrelse af pointers svarer normalt til ordstørrelsen
  (men også afhængigt af understøttet hukommelsesmængde, jfv. 8-bit micro controller.)



Forskellige compilere og maskiner placerer objekter forskelligt!

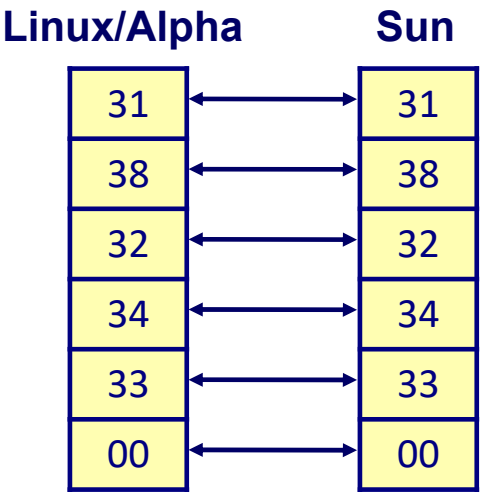
- Data-layout og ”alignment”

Repræsentation of Strenge

PP2.7

```
char S[6] = "18243";
```

- Strenge i C
  - Repræsenteret ved array af karakterer (chars)
  - Karkterer kodes i ASCII format
    - Standard 7-bit indkodning
    - Karakteren “0” has kode 0x30
      - Cifferet *i* har værdi 0x30+*i*
  - Strenge skal nul-termineres
  - Sidste karakter = 0
- Kompatibilitet
  - Byte ordning uproblematisk



Boolsk algebra
Bit niveau operationer
Logiske udtryk

Boolsk Algebra

- Udviklet af George Boole ca. år 1850
  - Algebra over Sand-Falsk værdier
  - Konvention: “True” kodes som 1, og “False” som 0

True, False
$\wedge$ AND
$\vee$ OR
$\neg$ NOT
$\oplus$ XOr

And
A&B = 1 når både A=1 og B=1

&		B	
		0	1
A	0	0	0
	1	0	1

Or
A|B = 1 når A=1 eller B=1

|		B	
		0	1
A	0	0	1
	1	1	1

Exclusive-Or (Xor: ”enten-eller”)
A^B = 1 når netop ét af: A=1 eller B=1,

^		B	
		0	1
A	0	0	1
	1	1	0

Not
~A = negeres

~		
A	0	1
	1	0

General Boolsk Algebra

PP2.8

- Opererer med Bit Vektorer
- Operationer anvendes bit-vist

```
  01101001     01101001     01101001
& 01010101   | 01010101   ^ 01010101   ~ 01010101
  01000001     01111101     00111100     10101010
```

- Regneregler for Boolsk Algebra (også for bit-vektorer)

Commutative:
- `A & B = B & A`
- `A | B = B | A`

Associative:
- `(A & B) & C = A & (B & C)`
- `(A | B) | C = A | (B | C))`

Distributive:
- `(A & (B | C) = (A & B) | (A & C)`
- `(A | (B & C) = (A | B) & (A | C)`

Identity:
- `A & 1 = A`
- `A | 0 = A`

Complement:
- `(A & (~A) = 0`
- `A | (~A ) = 1`

DeMorgan's Law:
- `~( A & B) = (~A) | (~B)`
- `~( A | B) = (~A) & (~B)`

Bit-niveau Operationer i C

PP2.10
PP2.11

- Bit-vise operationer &, |, ~, ^ findes i C
  - Kan anvendes på alle "hel-tallige" data typer: long, int, short, char, unsigned varianter
  - Operander fortolkes som bit vektorer

```
char b;
char a=0x41;
b=~a;
c=a&b
```

- Ex. Char data type:

```
~0x0C → 0xF3
    ~0000 1100₂ → 1111 0011₂
~0x00 → 0xFF
    ~0000 0000₂ → 1111 1111₂
```

```
0x69 | 0x0C → 0x6D (set bit 3-4)
      0110 1001₂
    | 0000 1100₂
    = 0110 1101₂
```

```
0x69 & 0x0C → 0x08 (udlæse bit 3-4)
      0110 1001₂
    & 0000 1100₂
    = 0000 1000₂
```

```
0x69 & ~0x0C → 0x6D (slet bit 3-4)
      0110 1001₂
    & 1111 0011₂
    = 0110 0001₂
```

Eksempel på anvendelse: Repræsentation af (små) tal-mængder

Nyttigt, simpelt, effektivt for "små" integers.

- Repræsentation
  - bit vektor *a* med længde w kan repræsentere en delmængde af tallene : $A \subseteq \{0, \ldots, w-1\}$
  - Bit $a_j = 1$, hvis $j \in A$

    A={ 0, 3, 5, 6 } 0110 1001
    7654 3210
  - Fx

    B={ 0, 2, 4, 6 } 0101 0101
    7654 3210
- Operationer
  - & fællesmængde $A \cap B = \{ 0, 6 \}$ 01000001
  - | foreningsmængde $A \cup B = \{ 0, 2, 3, 4, 5, 6 \}$ 01111101
  - ^ Symmetrisk difference $A \ominus B = \{ 2, 3, 4, 5 \}$ 00111100
  - ~ Komplementærmængde $B^C = \{ 1, 3, 5, 7 \}$ 10101010

Modsætning: Logiske udtryk i C

PP2.13
PP2.14

- ”Logiske” Operatorer i **betingede udtryk**:
  - &&, ||, !
  - Betragter 0 som “False”
  - Alle ikke-nul værdier er “True”
  - Returnerer altid **0 or 1**
  - Tidlig terminering (Short-circuit evaluering):

```
if(a && b ){}        if (a||b) {}
```

- Eksempler (char datatype)
  - !0x41 → 0x00
  - !0x00 → 0x01
  - !!0x41 → 0x01
  - 0x69 && 0x55 → 0x01
  - 0x69 || 0x55 → 0x01
  - p && *p (undgå de-referering af null-pointer)

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

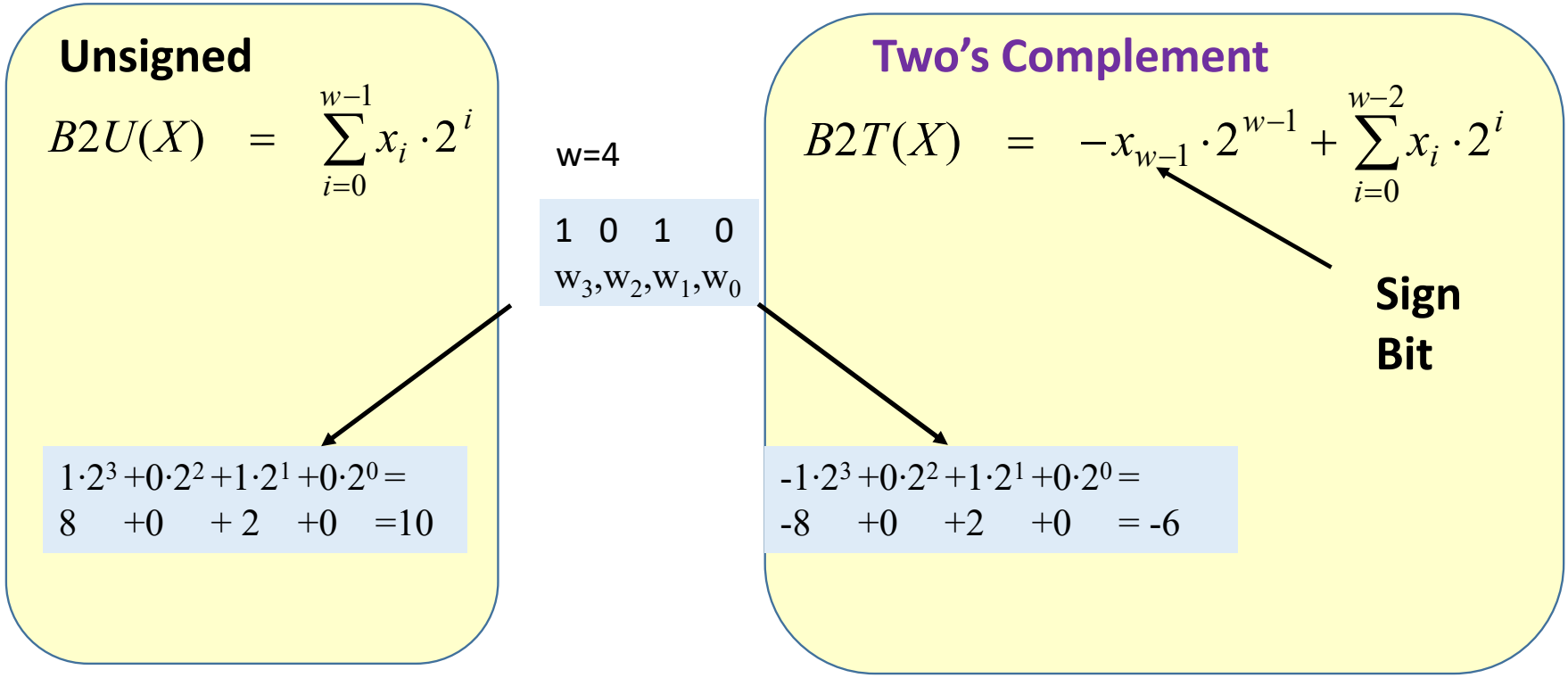
$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

"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

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

- 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.

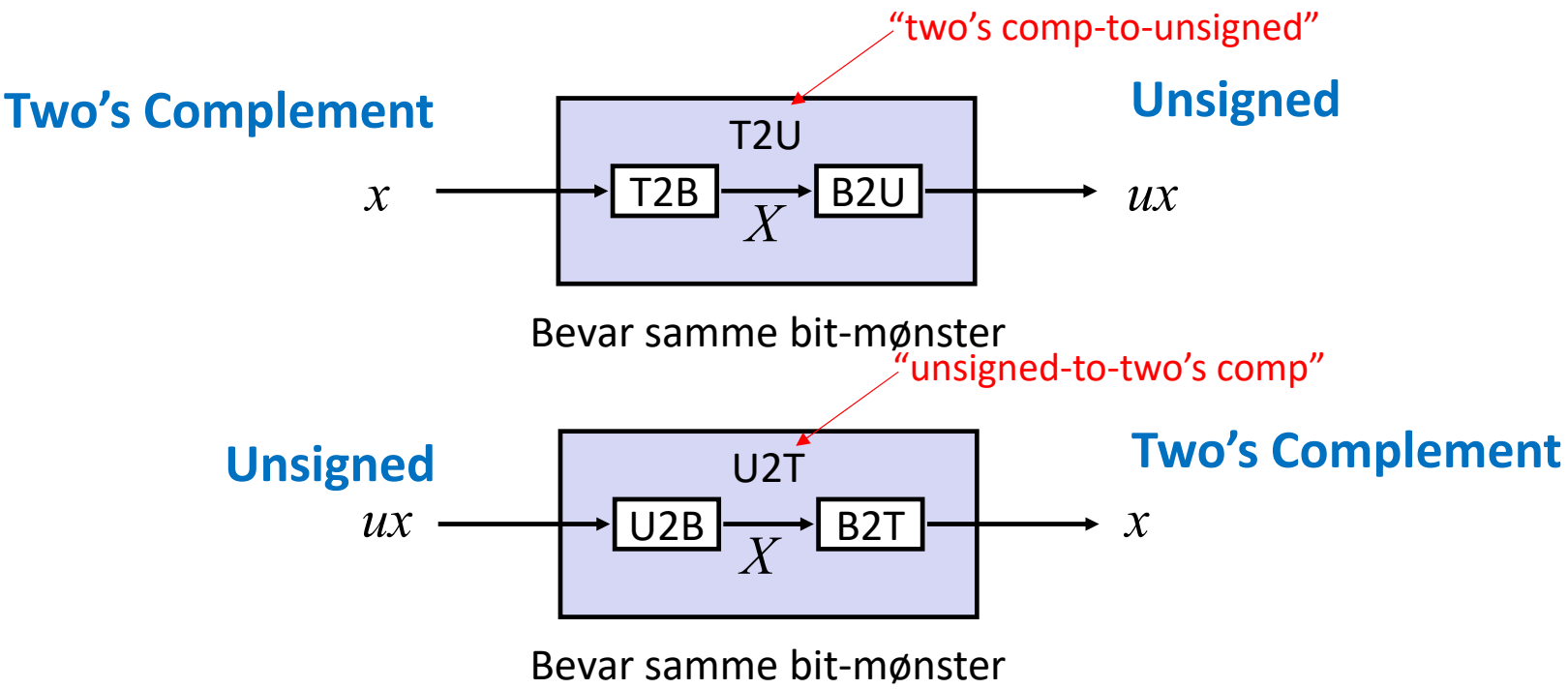
Bemærking: Præcision

- 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
  - $x \neq ux$

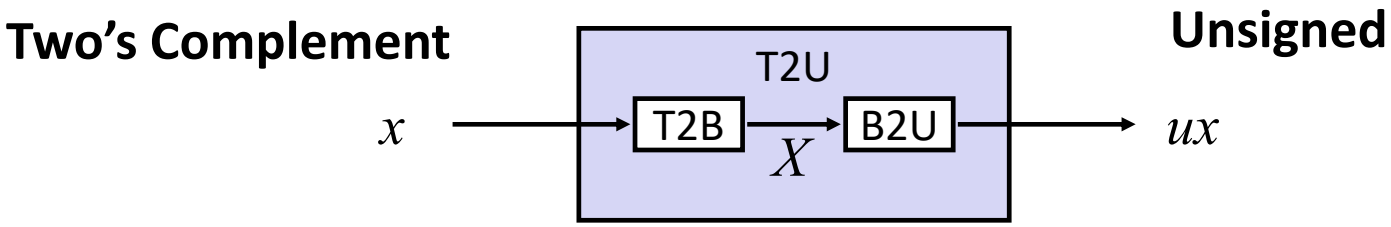
Konvertering Signed ↔ Unsigned

Bits	Signed		Unsigned
0000	0		0
0001	1		1
0010	2		2
0011	3		3
0100	4		4
0101	5	T2U →	5
0110	6	← U2T	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

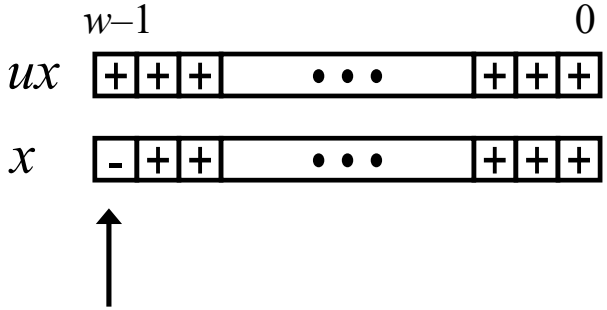
Konvertering Signed ↔ Unsigned

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



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

Stor negativ vægt: -2^{w-1}

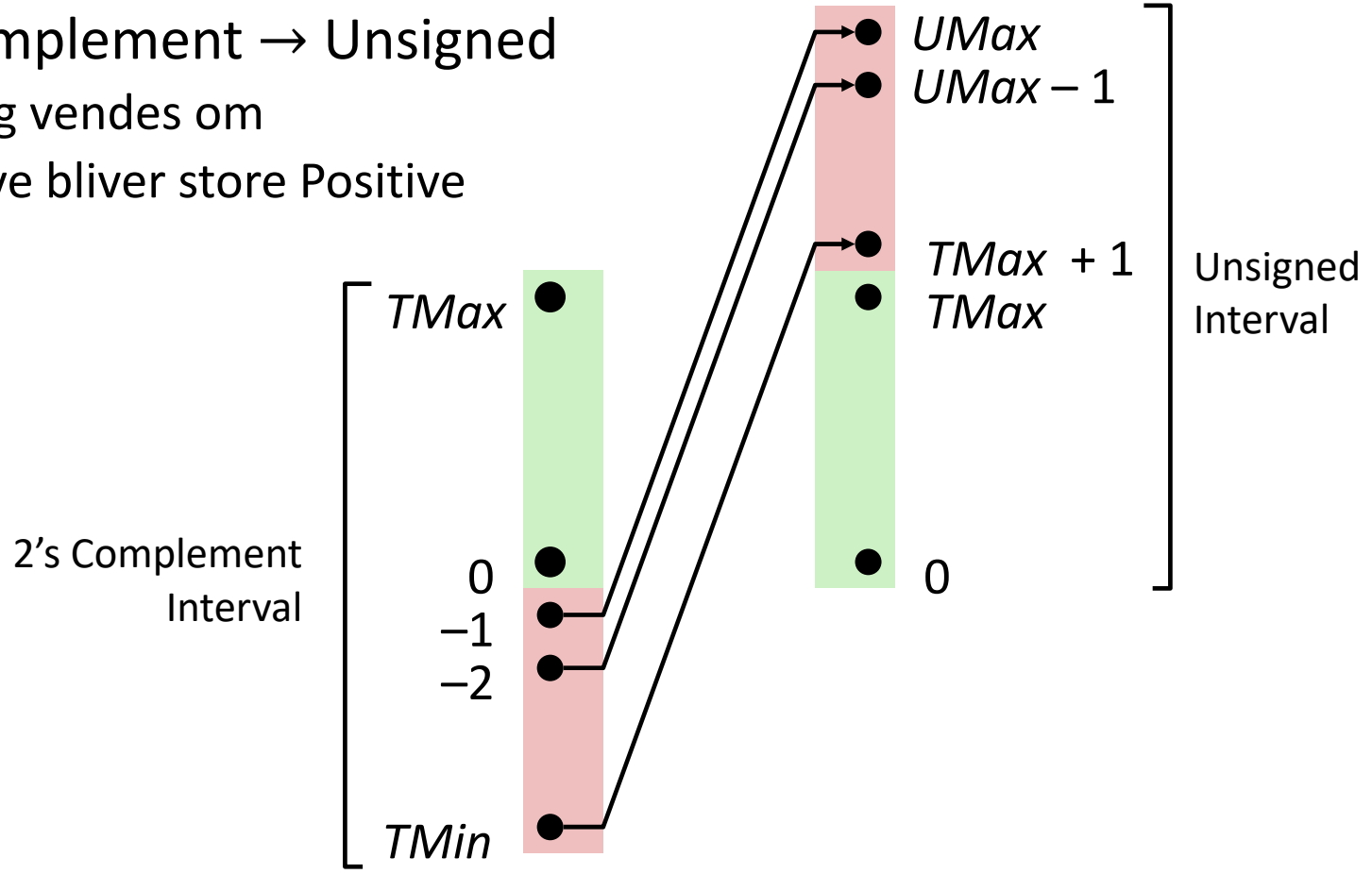
bliver

Stor positiv vægt: 2^{w-1}

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”

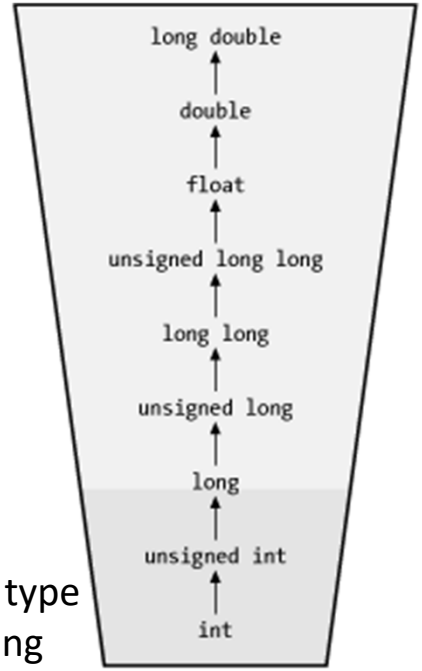
Onde men vigtige teknikaliteter.

- 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

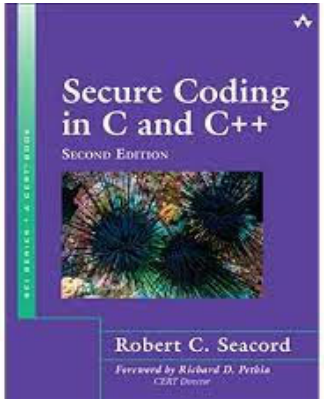
```
/* Declaration of library function memcpy */
void *memcpy(void *dest, void *src, size_t n);
/* stddef.h:size_t -> unsigned int */
```

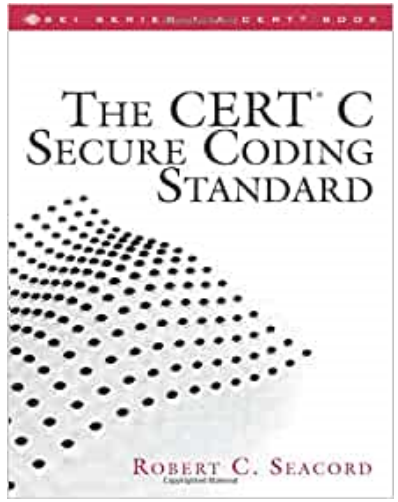
```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* 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:





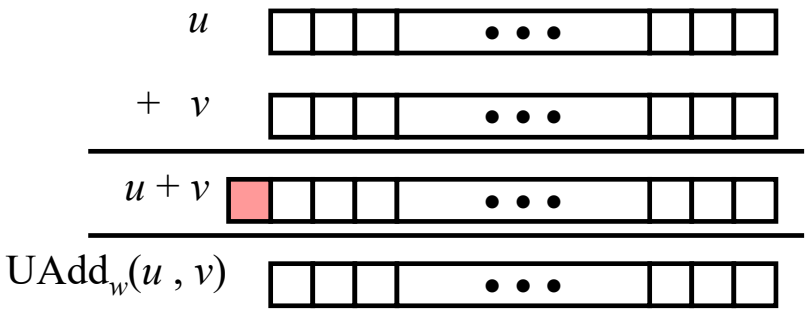
Aritmetik og over-/under-flow

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
```

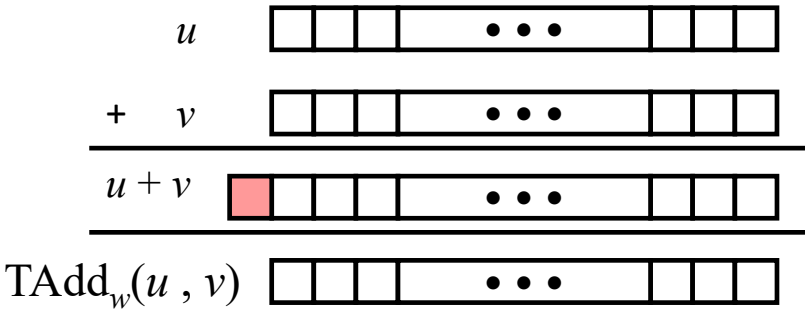
I DSP bruges også ”saturated” aritemtik:
- bevarer største værdi: 1000+1011=1111
- Resultat bliver numerisk tættest på sande sum
- Distributive og associative love gælder ej

Two's Complement Addition

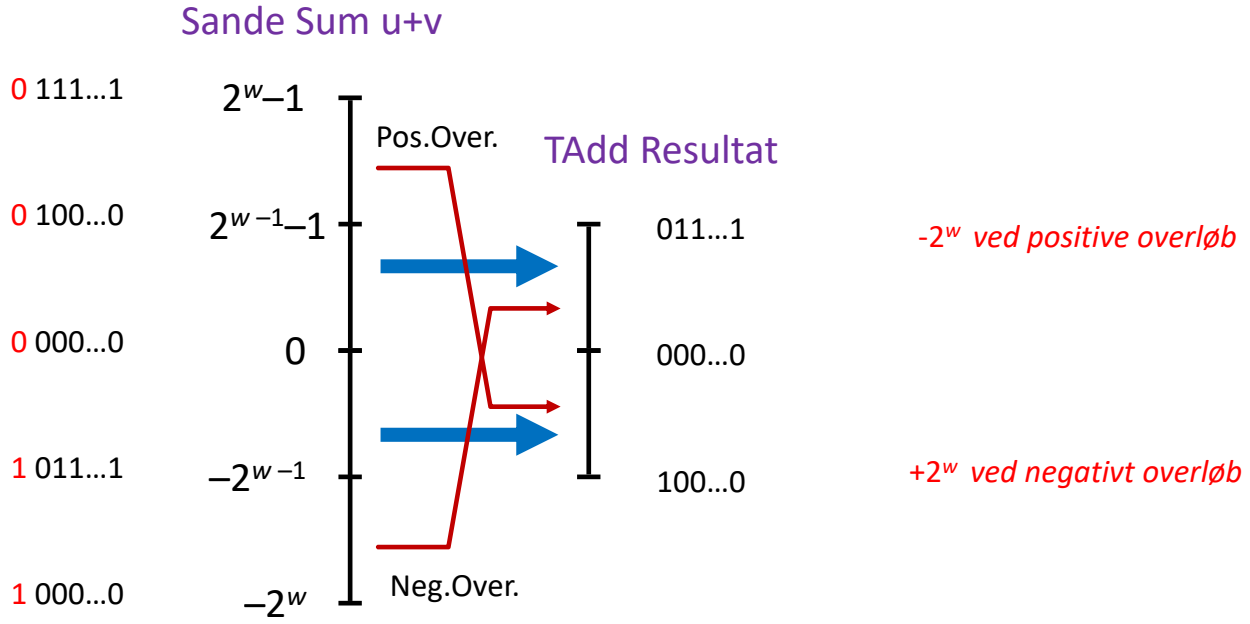
Operander: *w* bits

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"

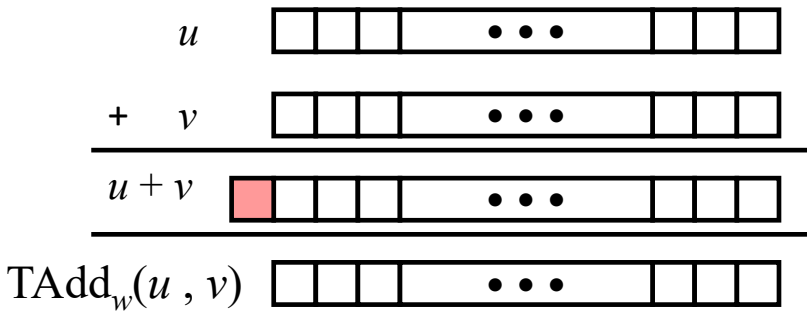


Two's Complement Addition

Operander: *w* bits

Sande sum: *w*+1 bits

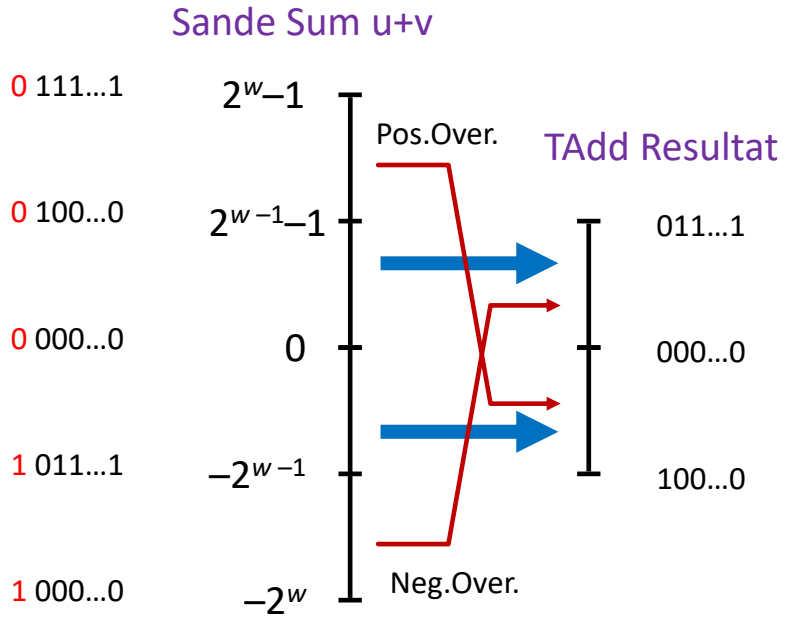
Menten (Carry) kasséres: *w* bits



SIGNED Positivt overløb

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

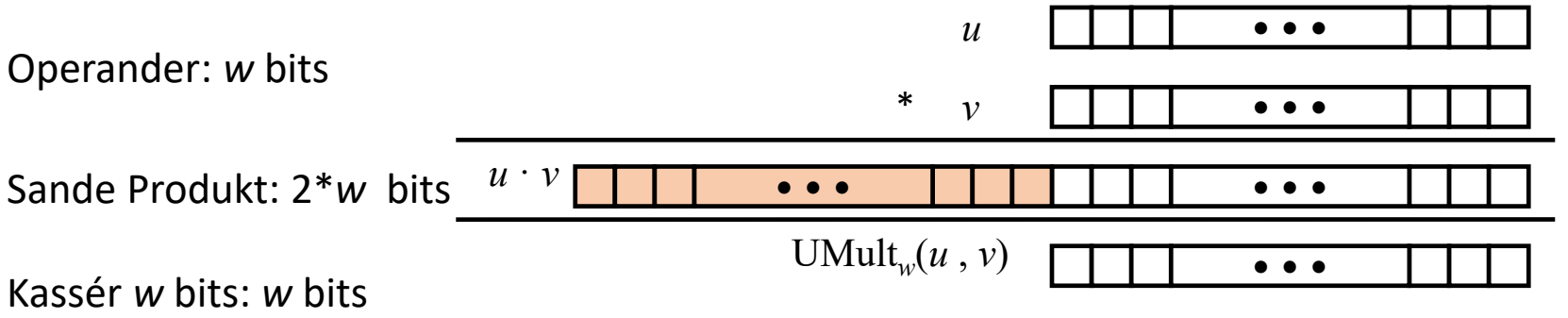
- Overløb (Overflow)
  - Sande sum kræver *w*+1 bits
  - Msb mistes
  - Resterende bits behandles som Two's comp. Integer
  - "Abelsk gruppe"



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 $\lfloor u / 2^k \rfloor$
  - 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 $\lfloor u / 2^k \rfloor$
    - 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`**!!