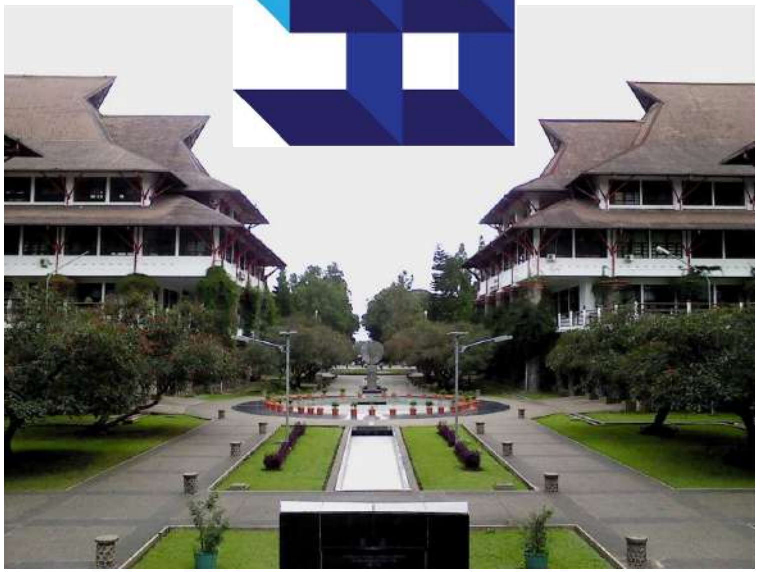


Modul 3

Integer

Representasi dan Operasi

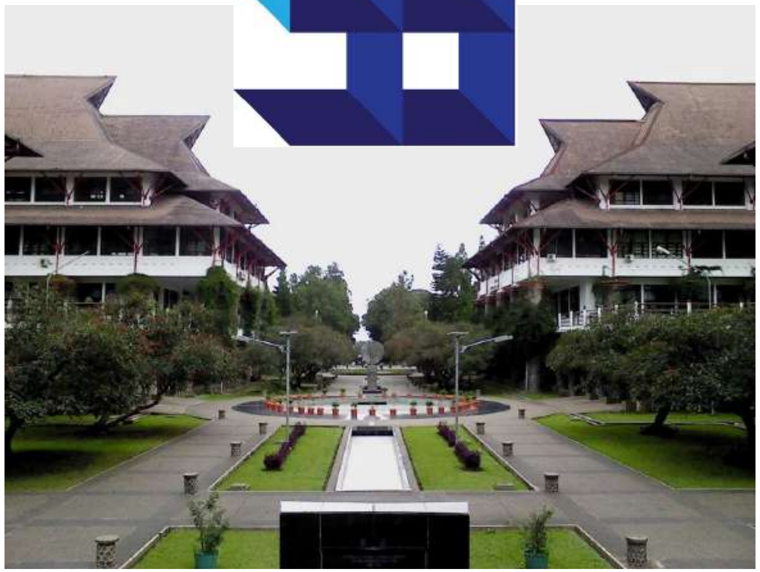
EL3011 Arsitektur Sistem Komputer

STEI - Institut Teknologi Bandung

Contents

1. Integer Representation
2. Conversion and Casting
3. Expanding and Truncating
4. Integer Addition, Negation, Multiplication and Shifting
5. Summary

This module adopted from 15-213 Introduction to Computer Systems Lecture, Carnegie Mellon University, 2020



STEI - Institut Teknologi Bandung

Modul 3. Integer

3.1. Integer Representation

EL3011 Arsitektur Sistem Komputer

Encoding Integers

Unsigned

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

Two's Complement

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

Sign Bit

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

- C does not mandate using two's complement
  - But, most machines do, and we will assume so
- C `short` 2 bytes long

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

- Sign Bit
  - For 2's complement, most significant bit indicates sign
    - 0 for nonnegative
    - 1 for negative

Two-complement: Simple Example

	-16	8	4	2	1	
10 =	**0**	**1**	**0**	**1**	**0**	8+2 = 10

	-16	8	4	2	1	
-10 =	**1**	**0**	**1**	**1**	**0**	-16+4+2 = -10

Two-complement Encoding Example (Cont.)

```
x =      15213: 00111011 01101101
y =     -15213: 11000100 10010011
```

Weight	15213		-15213	
1	1	1	1	1
2	0	0	1	2
4	1	4	0	0
8	1	8	0	0
16	0	0	1	16
32	1	32	0	0
64	1	64	0	0
128	0	0	1	128
256	1	256	0	0
512	1	512	0	0
1024	0	0	1	1024
2048	1	2048	0	0
4096	1	4096	0	0
8192	1	8192	0	0
16384	0	0	1	16384
-32768	0	0	1	-32768
Sum		**15213**		**-15213**

Numeric Ranges

- Unsigned Values
  - *UMin* = 0
    000…0
  - *UMax* = $2^w - 1$
    111…1

- Two's Complement Values
  - *TMin* = -2^{w-1}
    100…0
  - *TMax* = $2^{w-1} - 1$
    011…1
  - Minus 1
    111…1

Values 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

Values for Different Word Sizes

	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

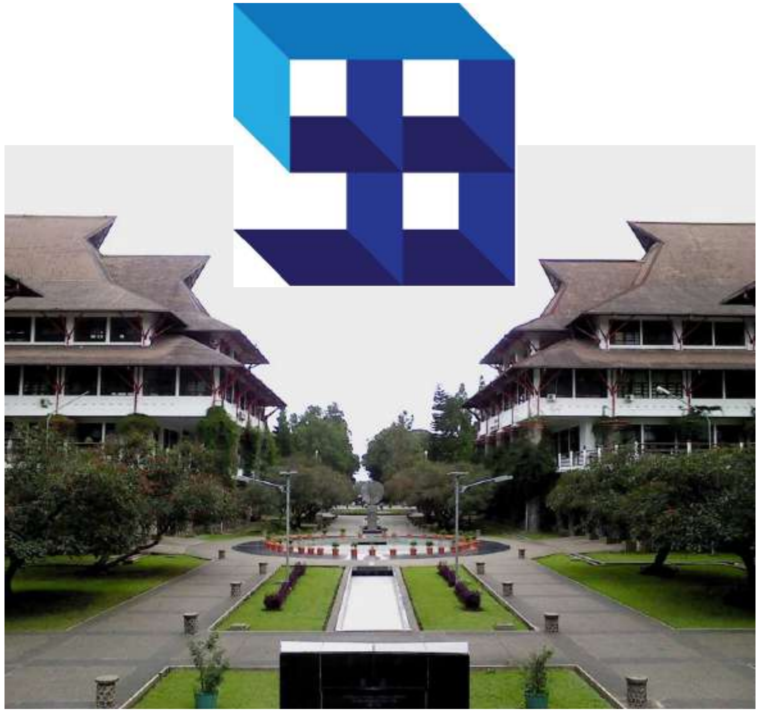
- Observations
  - $|TMin| \quad = \quad TMax + 1$
    - Asymmetric range
  - $UMax \quad = \quad 2 * TMax + 1$
  - Question: abs(TMin)?

- **C Programming**
  - #include <limits.h>
  - Declares constants, e.g.,
    - ULONG_MAX
    - LONG_MAX
    - LONG_MIN
  - Values platform specific

Unsigned & Signed Numeric Values

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

- Equivalence
  - Same encodings for nonnegative values
- Uniqueness
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding
- $\Rightarrow$ Can Invert Mappings
  - U2B(x) = B2U^{-1}(x)
    - Bit pattern for unsigned integer
  - T2B(x) = B2T^{-1}(x)
    - Bit pattern for two's comp integer



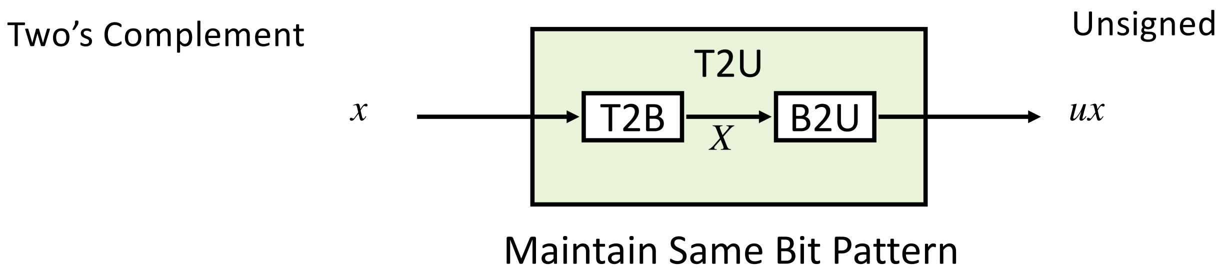
STEI - Institut Teknologi Bandung

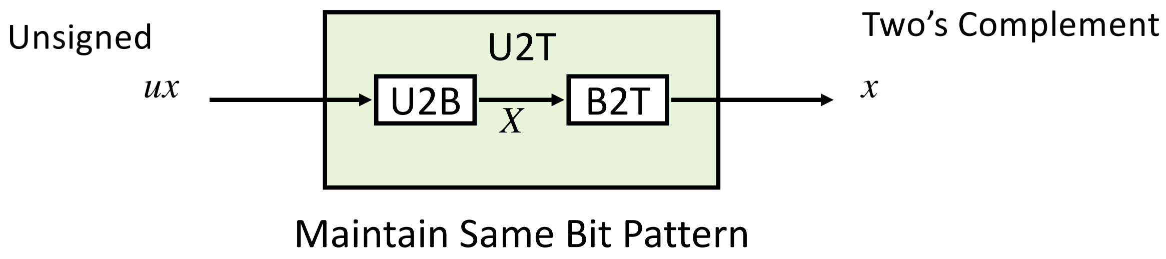
Modul 3. Integer

3.2. Conversion and Casting

EL3011 Arsitektur Sistem Komputer

Mapping Between Signed & Unsigned





- Mappings between unsigned and two’s complement numbers: Keep bit representations and reinterpret

Mapping 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

→ T2U →

← U2T ←

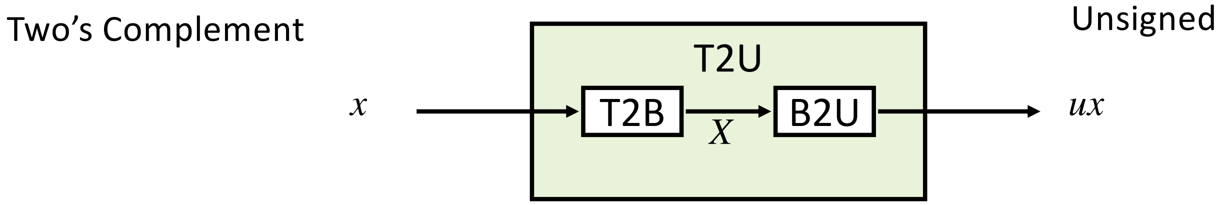
Mapping Signed ↔ Unsigned

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

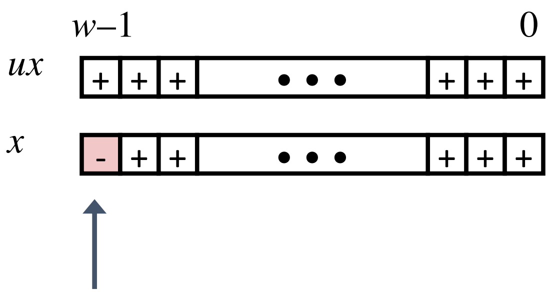
=

+/- 16

Relation between Signed & Unsigned



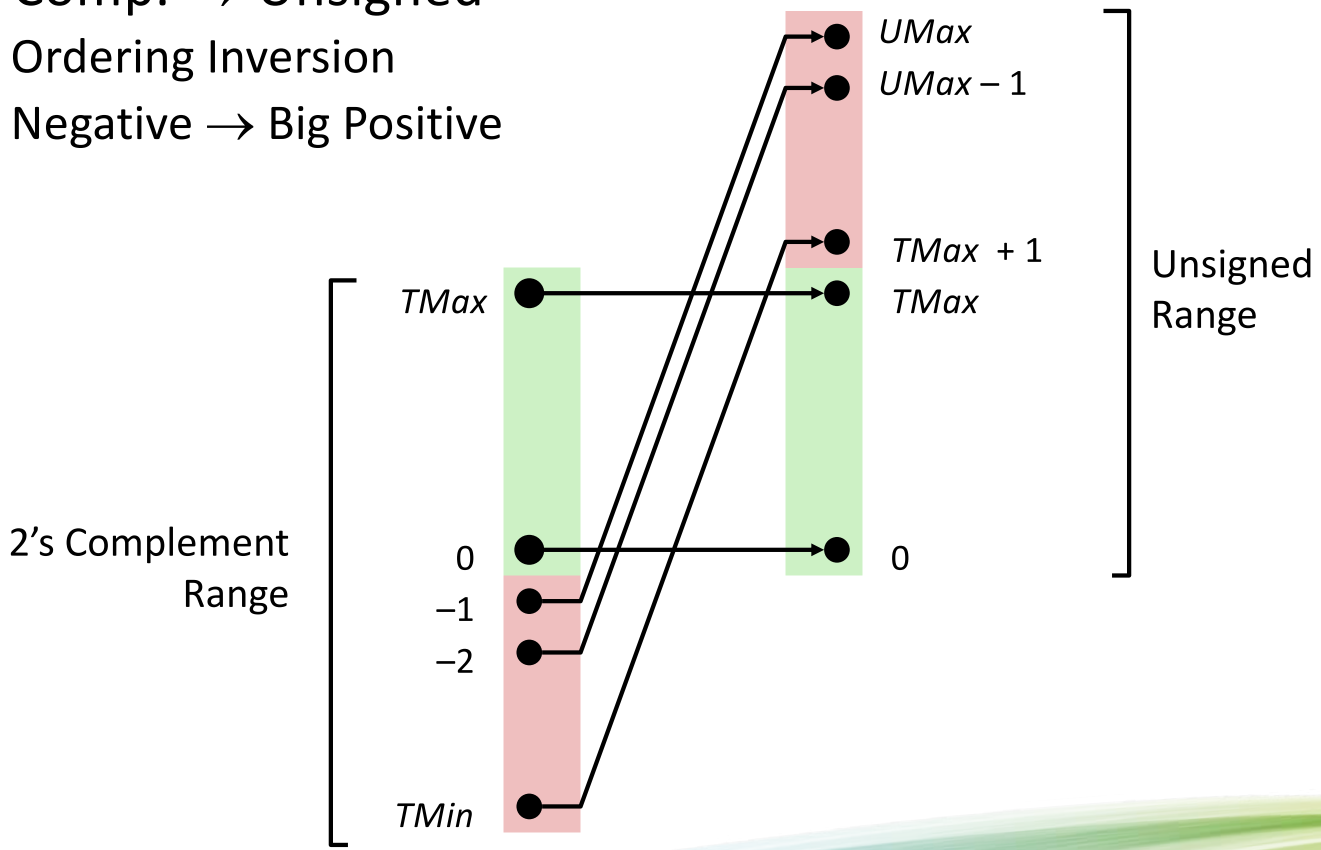
Maintain Same Bit Pattern



Large negative weight
becomes
Large positive weight

Conversion Visualized

- 2's Comp. → Unsigned
  - Ordering Inversion
  - Negative → Big Positive



Signed vs. Unsigned in C

- Constants
  - By default are considered to be signed integers
  - Unsigned if have "U" as suffix

    ```
    0U, 4294967259U
    ```
- Casting
  - Explicit casting between signed & unsigned same as U2T and T2U

    ```
    int tx, ty;
    unsigned ux, uy;
    tx = (int) ux;
    uy = (unsigned) ty;
    ```
  - Implicit casting also occurs via assignments and procedure calls

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

    ```
    int fun(unsigned u);
    uy = fun(tx);
    ```

Casting Surprises

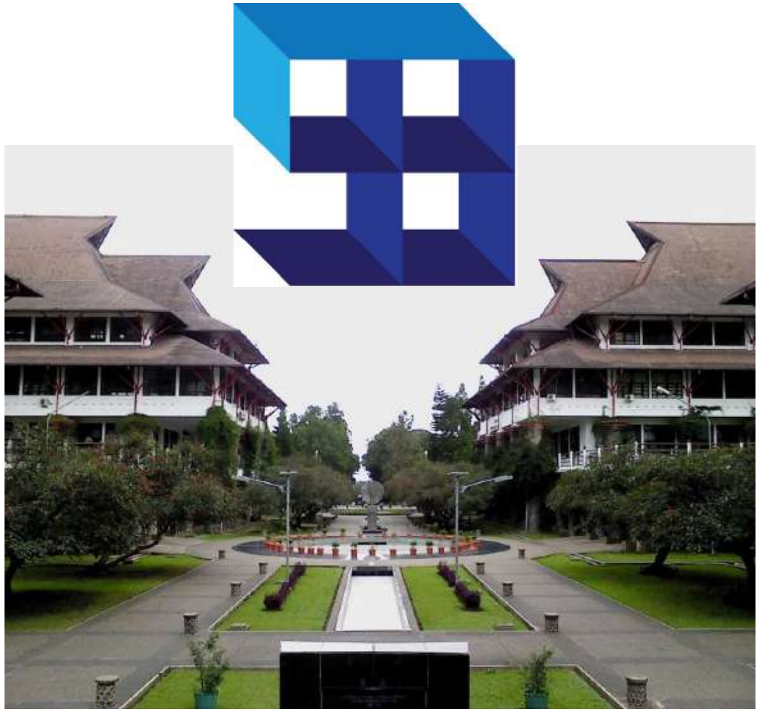
- Expression Evaluation
  - If there is a mix of unsigned and signed in single expression, ***signed values implicitly cast to unsigned***
  - Including comparison operations **<, >, ==, <=, >=**
  - Examples for $W = 32$: **TMIN = -2,147,483,648 ,** **TMAX = 2,147,483,647**

Constant$_1$	Constant$_2$	Relation	Evaluation
0	0U	==	unsigned
-1	0	<	signed
-1	0U	>	unsigned
2147483647	-2147483647-1	>	signed
2147483647U	-2147483647-1	<	unsigned
-1	-2	>	signed
(unsigned)-1	-2	>	unsigned
2147483647	2147483648U	<	unsigned
2147483647	(int) 2147483648U	>	signed

Summary
Casting Signed ↔ Unsigned: Basic Rules

- Bit pattern is maintained
- But reinterpreted
- Can have unexpected effects: adding or subtracting 2^w

- Expression containing signed and unsigned int
  - `int` is cast to `unsigned`!!



Modul 3. Integer

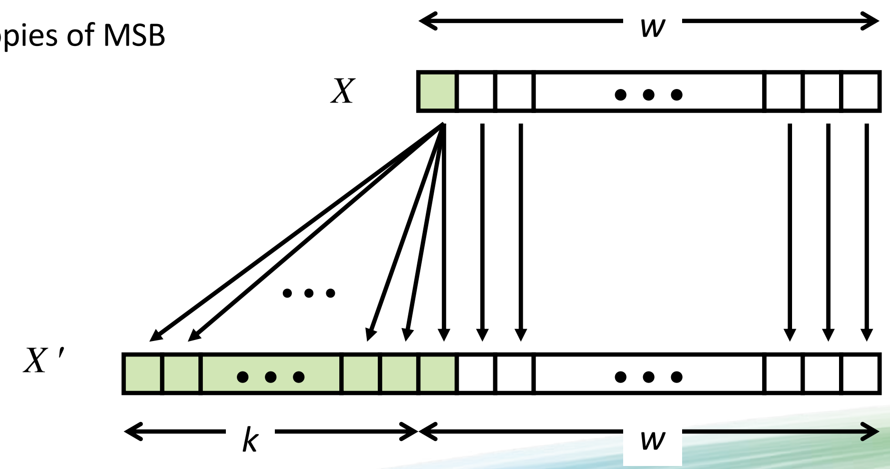
3.3. Expanding and Truncating

EL3011 Arsitektur Sistem Komputer

STEI - Institut Teknologi Bandung

Sign Extension

- Task:
  - Given *w*-bit signed integer *x*
  - Convert it to *w*+*k*-bit integer with same value
- Rule:
  - Make *k* copies of sign bit:
  - $X' = \underbrace{x_{w-1},\ldots,x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2},\ldots,x_0$



Sign Extension: Simple Example

Positive number

	-16	8	4	2	1
10 =	0	1	0	1	0

	-32	16	8	4	2	1
10 =	0	0	1	0	1	0

Negative number

	-16	8	4	2	1
-10 =	1	0	1	1	0

	-32	16	8	4	2	1
-10 =	1	1	0	1	1	0

Larger Sign Extension Example

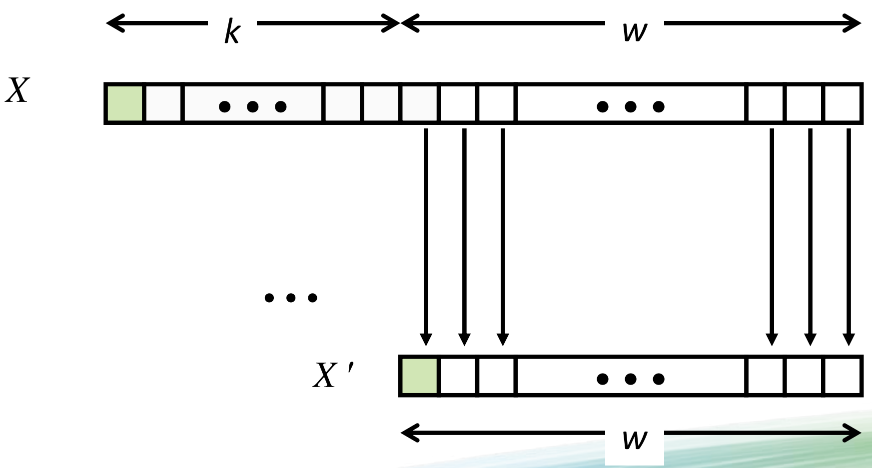
```
short int x =  15213;
int       ix = (int) x;
short int y = -15213;
int       iy = (int) y;
```

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
ix	15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

- Converting from smaller to larger integer data type
- C automatically performs sign extension

Truncation

- Task:
  - Given k+*w*-bit signed or unsigned integer *X*
  - Convert it to *w*-bit integer X’ with same value for “small enough” X
- Rule:
  - Drop top *k* bits:
  - $X' = x_{w-1}, x_{w-2}, \ldots, x_0$



Truncation: Simple Example

No sign change

	-16	8	4	2	1
2 =	0	0	0	1	0

	-8	4	2	1
2 =	0	0	1	0

2 mod 16 = 2

	-16	8	4	2	1
-6 =	1	1	0	1	0

	-8	4	2	1
-6 =	1	0	1	0

-6 mod 16 = 26U mod 16 = 10U = -6

Sign change

	-16	8	4	2	1
10 =	0	1	0	1	0

	-8	4	2	1
-6 =	1	0	1	0

10 mod 16 = 10U mod 16 = 10U = -6

	-16	8	4	2	1
-10 =	1	0	1	1	0

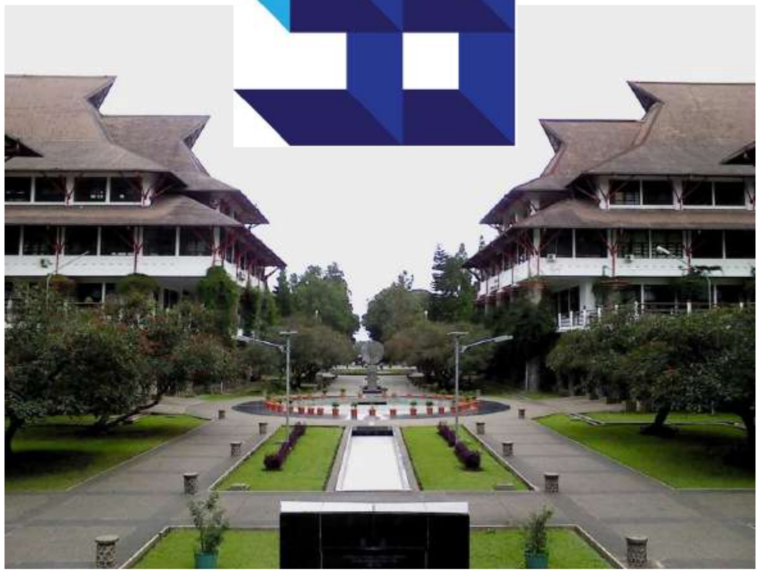
	-8	4	2	1
6 =	0	1	1	0

-10 mod 16 = 22U mod 16 = 6U = 6

Summary:
Expanding, Truncating: Basic Rules

- Expanding (e.g., short int to int)
  - Unsigned: zeros added
  - Signed: sign extension
  - Both yield expected result

- Truncating (e.g., unsigned to unsigned short)
  - Unsigned/signed: bits are truncated
  - Result reinterpreted
  - Unsigned: mod operation
  - Signed: similar to mod
  - For small (in magnitude) numbers yields expected behavior



Modul 3. Integer

3.4. Addition, Negation, Multiplication, Shifting

EL3011 Arsitektur Sistem Komputer

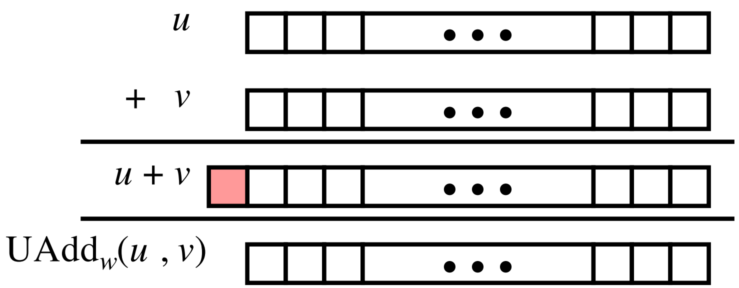
STEI - Institut Teknologi Bandung

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



- Standard Addition Function
  - Ignores carry output
- Implements Modular Arithmetic

  $s \;=\; \text{UAdd}_w(u\ ,\ v) \;=\; u + v \mod 2^w$

```
unsigned char     1110 1001      E9      223
                + 1101 0101    + D5    + 213
                  _________    ____    _____

                  _________    ____    _____
```

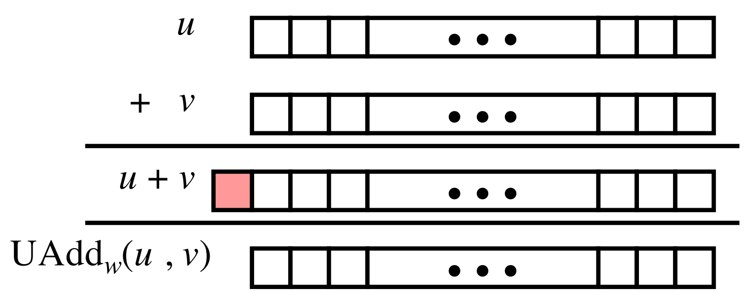
Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



- Standard Addition Function
  - Ignores carry output
- Implements Modular Arithmetic

$$s \quad = \quad \mathrm{UAdd}_w(u\,,v) \quad = \quad u + v \mod 2^w$$

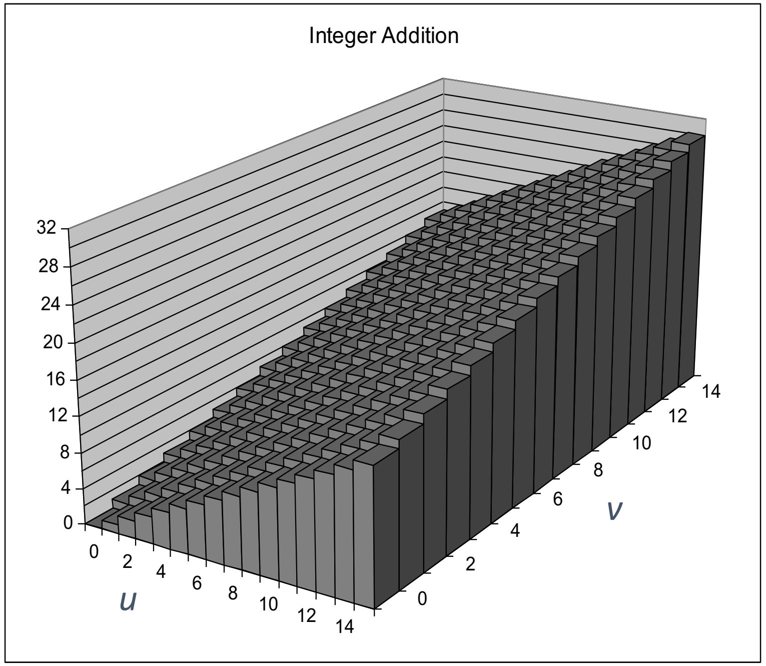
```
unsigned char      1110 1001      E9      223
               +   1101 0101    + D5    + 213
                 1 1011 1110     1BE      446
                   1011 1110      BE      190
```

Hex	Decimal	Binary
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

Visualizing (Mathematical) Integer Addition

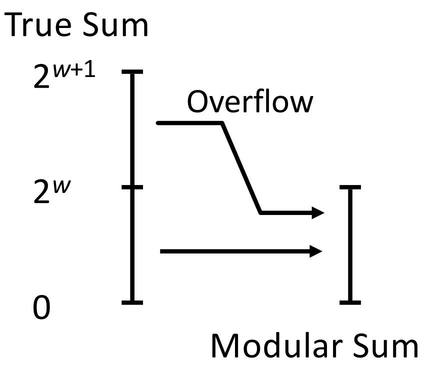
- Integer Addition
  - 4-bit integers *u*, *v*
  - Compute true sum $Add_4(u, v)$
  - Values increase linearly with *u* and *v*
  - Forms planar surface

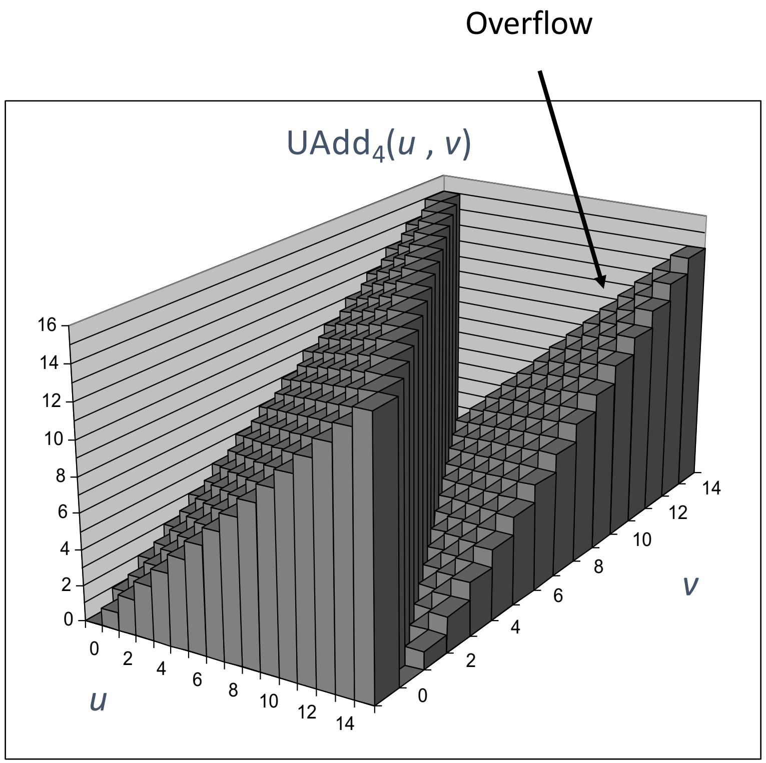
$Add_4(u, v)$



Visualizing Unsigned Addition

- Wraps Around
  - If true sum ≥ 2^w
  - At most once



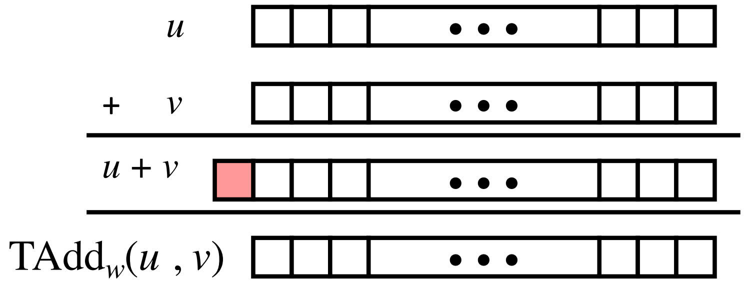


Two's Complement Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



- TAdd and UAdd have Identical Bit-Level Behavior
  - Signed vs. unsigned addition in C:

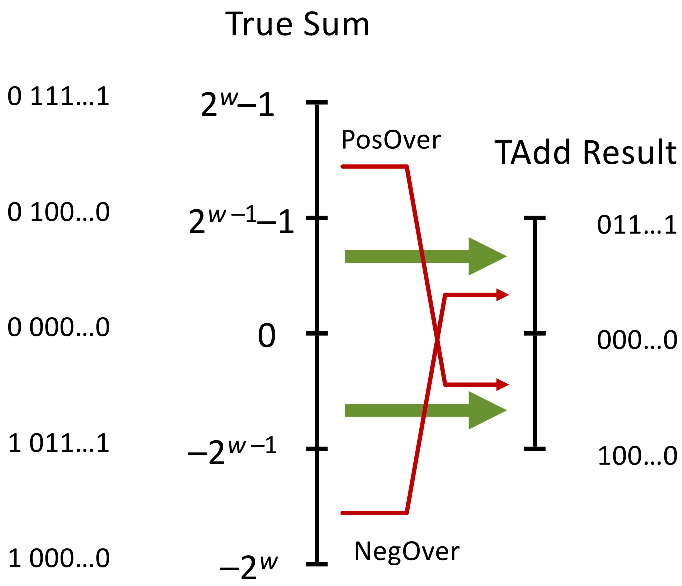
```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v
```

  - Will give `s == t`

1110 1001	E9	-23
+ 1101 0101	+ D5	+ -43
1 1011 1110	1BE	-66
1011 1110	BE	-66

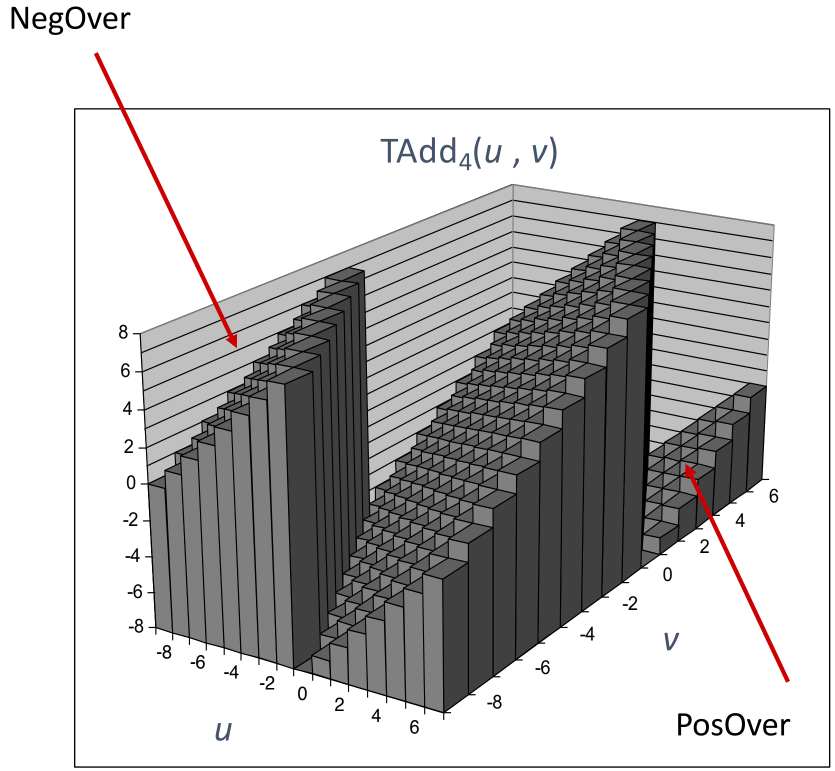
TAdd Overflow

- Functionality
  - True sum requires w+1 bits
  - Drop off MSB
  - Treat remaining bits as 2's comp. integer



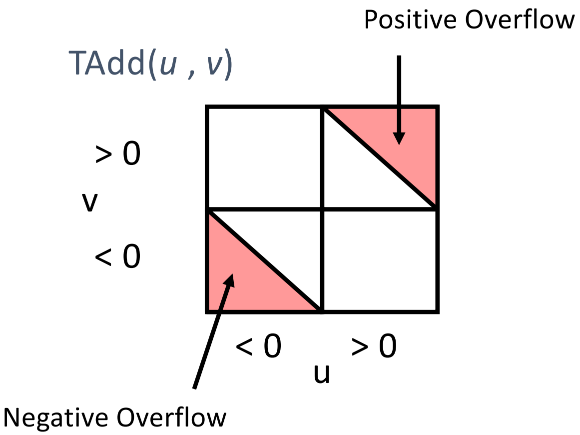
Visualizing 2's Complement Addition

- Values
  - 4-bit two's comp.
  - Range from -8 to +7
- Wraps Around
  - If sum $\geq 2^{w-1}$
    - Becomes negative
    - At most once
  - If sum $< -2^{w-1}$
    - Becomes positive
    - At most once



Characterizing TAdd

- Functionality
  - True sum requires *w*+1 bits
  - Drop off MSB
  - Treat remaining bits as 2's comp. integer



$$TAdd_w(u,v) = \begin{cases} u+v+2^w & u+v < TMin_w \quad \text{(NegOver)} \\ u+v & TMin_w \le u+v \le TMax_w \\ u+v-2^w & TMax_w < u+v \quad \text{(PosOver)} \end{cases}$$

Multiplication

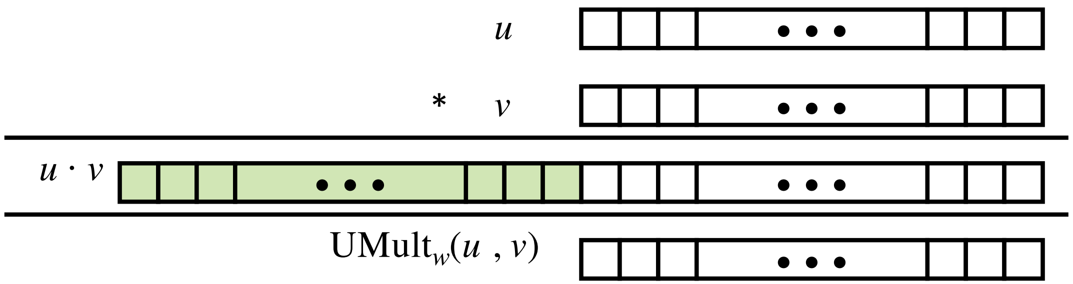
- Goal: Computing Product of *w*-bit numbers *x*, *y*
  - Either signed or unsigned
- But, exact results can be bigger than *w* bits
  - Unsigned: up to 2*w* bits
    - Result range: $0 \le x * y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Two's complement min (negative): Up to 2*w*-1 bits
    - Result range: $x * y \ge (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
  - Two's complement max (positive): Up to 2*w* bits, but only for $(TMin_w)^2$
    - Result range: $x * y \le (-2^{w-1})^2 = 2^{2w-2}$
- So, maintaining exact results…
  - would need to keep expanding word size with each product computed
  - is done in software, if needed
    - e.g., by "arbitrary precision" arithmetic packages

Unsigned Multiplication in C

Operands: *w* bits

True Product: 2**w* bits

Discard *w* bits: *w* bits



- Standard Multiplication Function
  - Ignores high order *w* bits
- Implements Modular Arithmetic

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

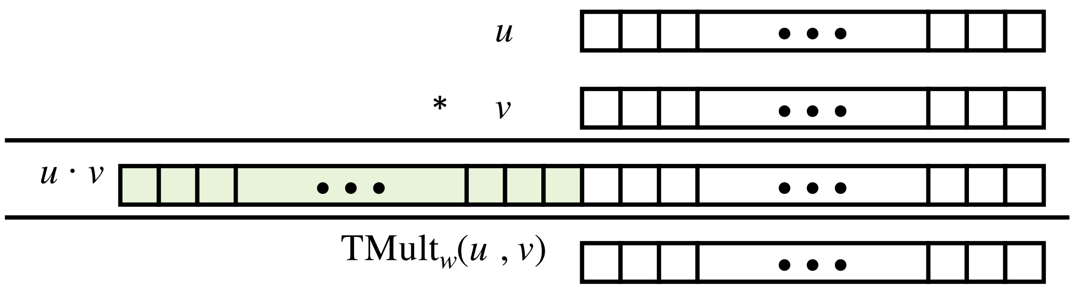
```
               1110 1001         E9          223
*              1101 0101     *   D5     *    213
  1100 0001 1101 1101          C1DD        47499
               1101 1101         DD          221
```

Signed Multiplication in C

Operands: *w* bits

True Product: 2**w* bits

Discard *w* bits: *w* bits

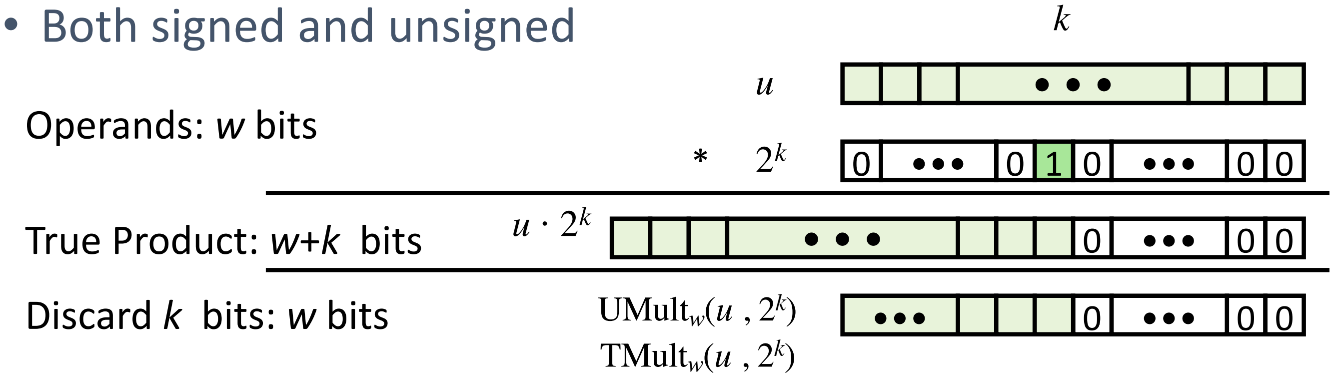


- Standard Multiplication Function
  - Ignores high order *w* bits
  - Some of which are different for signed vs. unsigned multiplication
  - Lower bits are the same

```
              1110 1001        E9          -23
*             1101 0101    *   D5     *    -43
 0000 0011 1101 1101         03DD          989
              1101 1101        DD          -35
```

Power-of-2 Multiply with Shift

- Operation
  - **`u << k`** gives **`u * `** ***2^k***
  - Both signed and unsigned



- Examples
  - `u << 3          ==  u * 8`
  - `(u << 5) - (u << 3)  ==   u * 24`
  - Most machines shift and add faster than multiply
    - Compiler generates this code automatically

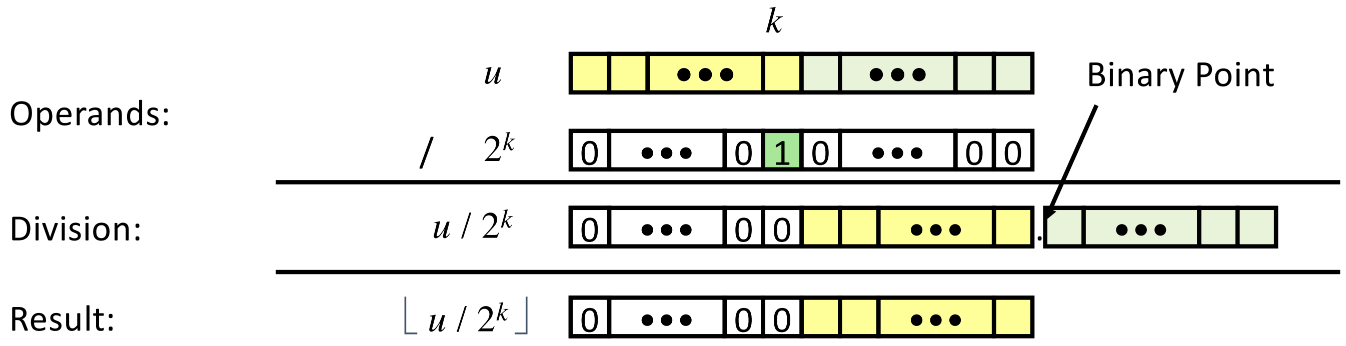
Important Lession:
Trust Your Compiler!

Multiplication

- Goal: Computing Product of *w*-bit numbers *x*, *y*
  - Either signed or unsigned
- But, exact results can be bigger than *w* bits
  - Unsigned: up to 2*w* bits
    - Result range: $0 \leq x * y \leq (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Two's complement min (negative): Up to 2*w*-1 bits
    - Result range: $x * y \geq (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
  - Two's complement max (positive): Up to 2*w* bits, but only for $(TMin_w)^2$
    - Result range: $x * y \leq (-2^{w-1})^2 = 2^{2w-2}$
- So, maintaining exact results…
  - would need to keep expanding word size with each product computed
  - is done in software, if needed
    - e.g., by "arbitrary precision" arithmetic packages

Unsigned Power-of-2 Divide with Shift

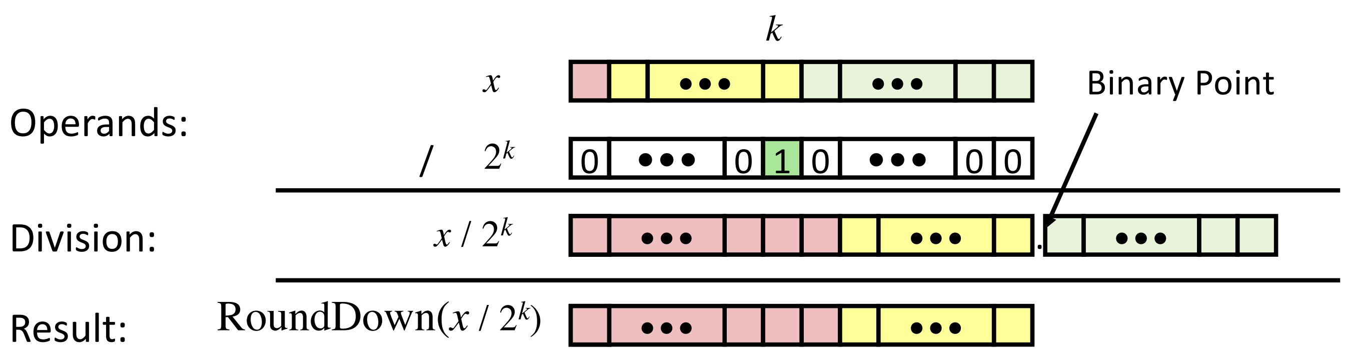
- Quotient of Unsigned by Power of 2
  - **`u >> k`** gives $\lfloor$ **`u / 2`**^{k} $\rfloor$
  - Uses logical shift



	Division	Computed	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`

Signed Power-of-2 Divide with Shift

- Quotient of Signed by Power of 2
  - **`x >> k`** gives $\lfloor$ **x / 2^k** $\rfloor$
  - Uses arithmetic shift
  - Rounds wrong direction when **`u < 0`**

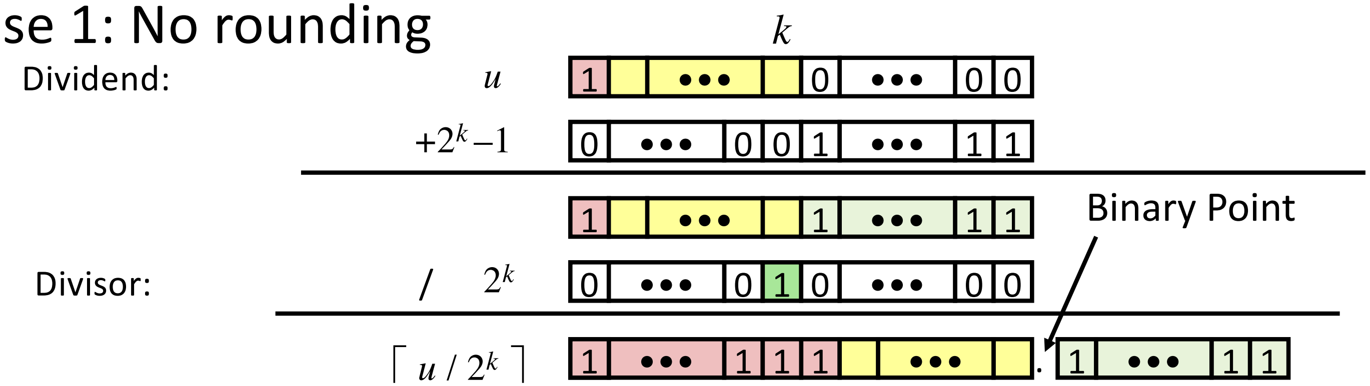


	Division	Computed	Hex	Binary
`y`	-15213	-15213	`C4 93`	`11000100 10010011`
`y >> 1`	-7606.5	-7607	`E2 49`	`11100010 01001001`
`y >> 4`	-950.8125	-951	`FC 49`	`11111100 01001001`
`y >> 8`	-59.4257813	-60	`FF C4`	`11111111 11000100`

Correct Power-of-2 Divide

- Quotient of Negative Number by Power of 2
  - Want $\lceil$ **`x / 2`**^{k} $\rceil$ (Round Toward 0)
  - Compute as $\lfloor$ **`(x+2`**^{k}**`-1) / 2`**^{k} $\rfloor$
    - In C: **`(x + (1<<k)-1) >> k`**
    - Biases dividend toward 0

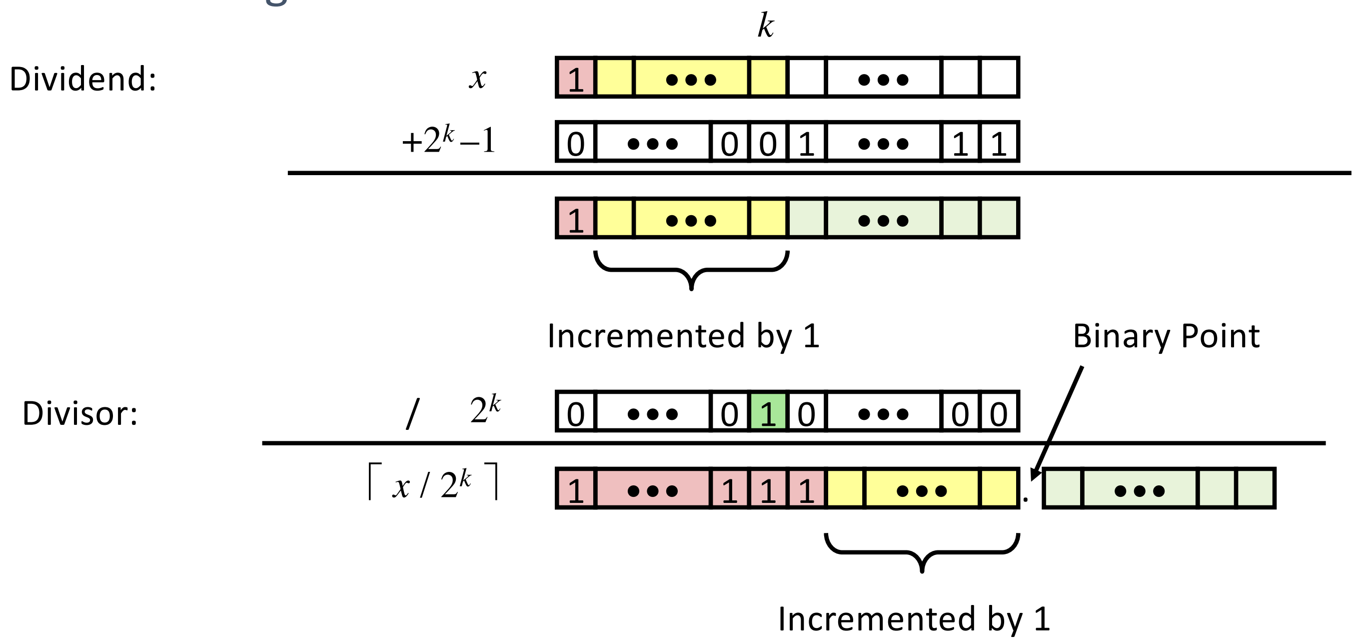
Case 1: No rounding



Biasing has no effect

Correct Power-of-2 Divide (Cont.)

Case 2: Rounding



Biasing adds 1 to final result

Negation: Complement & Increment

- Negate through complement and increase
  `~x + 1 == -x`
- Example
  - Observation: `~x + x == 1111…111 == -1`

x	1 0 0 1 1 1 0 1
+ ~x	0 1 1 0 0 0 1 0
-1	1 1 1 1 1 1 1 1

x = 15213

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
~x	-15214	C4 92	11000100 10010010
~x+1	-15213	C4 93	11000100 10010011
y	-15213	C4 93	11000100 10010011

Complement & Increment Examples

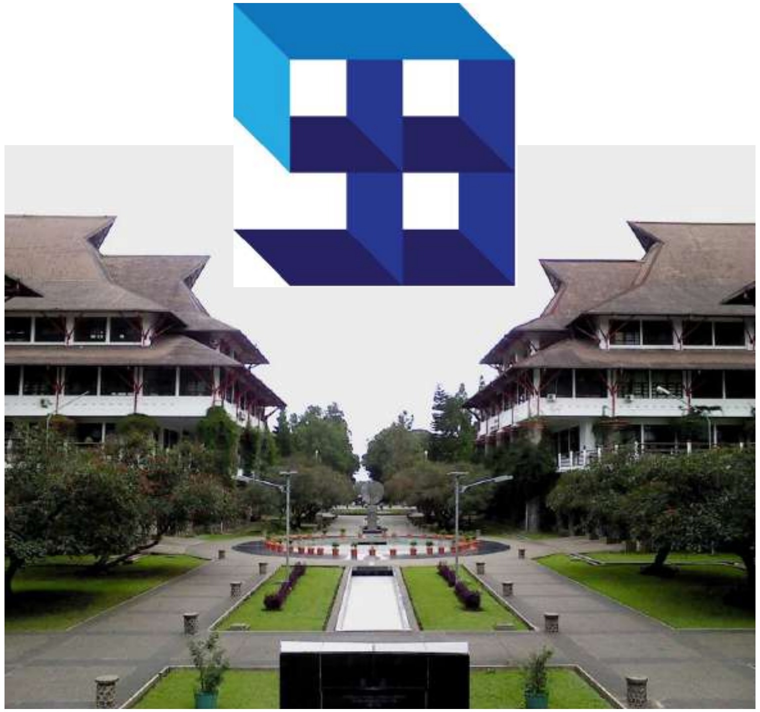
x = 0

	Decimal	Hex	Binary
0	0	00 00	00000000 00000000
~0	-1	FF FF	11111111 11111111
~0+1	0	00 00	00000000 00000000

x = TMin

	Decimal	Hex	Binary
x	-32768	80 00	10000000 00000000
~x	32767	7F FF	01111111 11111111
~x+1	-32768	80 00	10000000 00000000

Canonical counter example



STEI - Institut Teknologi Bandung

Modul 3. Integer

3.5. Integer Summary

EL3011 Arsitektur Sistem Komputer

Arithmetic: Basic Rules

- Addition:
  - Unsigned/signed: Normal addition followed by truncate, same operation on bit level
  - Unsigned: addition mod 2^w
    - Mathematical addition + possible subtraction of 2^w
  - Signed: modified addition mod 2^w (result in proper range)
    - Mathematical addition + possible addition or subtraction of 2^w

- Multiplication:
  - Unsigned/signed: Normal multiplication followed by truncate, same operation on bit level
  - Unsigned: multiplication mod 2^w
  - Signed: modified multiplication mod 2^w (result in proper range)

Why Should I Use Unsigned?

- *Don't* use without understanding implications
  - Easy to make mistakes

```
unsigned i;
for (i = cnt-2; i >= 0; i--)
  a[i] += a[i+1];
```

  - Can be very subtle

```
#define DELTA sizeof(int)
int i;
for (i = CNT; i-DELTA >= 0; i-= DELTA)
  . . .
```

Counting Down with Unsigned

- Proper way to use unsigned as loop index

```
unsigned i;
for (i = cnt-2; i < cnt; i--)
  a[i] += a[i+1];
```

- See Robert Seacord, *Secure Coding in C and C++*
  - C Standard guarantees that unsigned addition will behave like modular arithmetic
    - 0 – 1 → *UMax*
- Even better

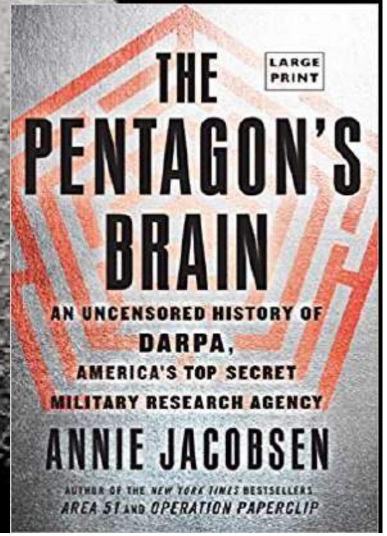
```
size_t i;
for (i = cnt-2; i < cnt; i--)
  a[i] += a[i+1];
```

  - Data type **size_t** defined as unsigned value with length = word size

Why Should I Use Unsigned? (cont.)

- *Do* Use When Performing Modular Arithmetic
  - Multiprecision arithmetic
- *Do* Use When Using Bits to Represent Sets
  - Logical right shift, no sign extension
- *Do* Use In System Programming
  - Bit masks, device commands,…

- Misunderstanding integers can lead to the end of the world as we know it!
- Thule (Qaanaaq), Greenland
- US DoD "Site J" Ballistic Missile Early Warning System (BMEWS)
- 10/5/60: world nearly ends
- Missile radar echo: 1/8s
- BMEWS reports: 75s echo(!)
- 1000s of objects reported
- NORAD alert level 5:
  - Immediate incoming nuclear attack!!!!



- Kruschev was in NYC 10/5/60 (weird time to attack)
  - someone in Qaanaaq said “why not go check outside?”
- “Missiles” were actually THE MOON RISING OVER NORWAY
- Expected max distance: 3000 mi; Moon distance: .25M miles!
- .25M miles % sizeof(distance) = 2200mi.
- Overflow of distance nearly caused nuclear apocalypse!!