Integer Representation

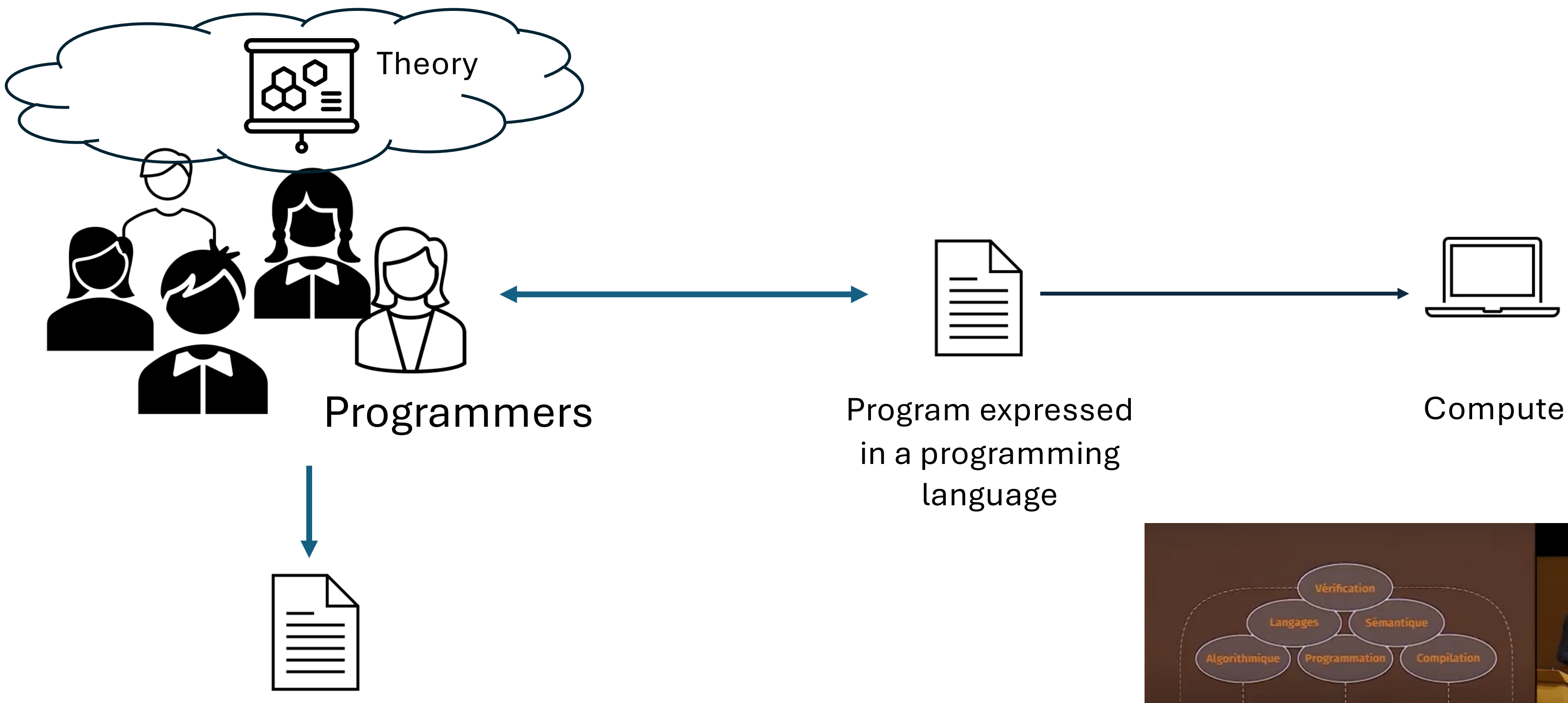
Philippe Bonnet, bonnet@di.ku.dk
HPPS 2025 – 1b

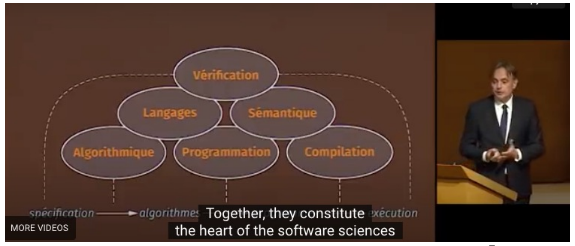
(with some illustrations from Computer Systems a Programmer's Perspective)

Outline

- Back to PoP
- Two’s complement
- Bit manipulation

What is programming?





https://youtu.be/JVq11lV4gRY

A program

A program is built with data and functions.

Data are immutable values that have a type.

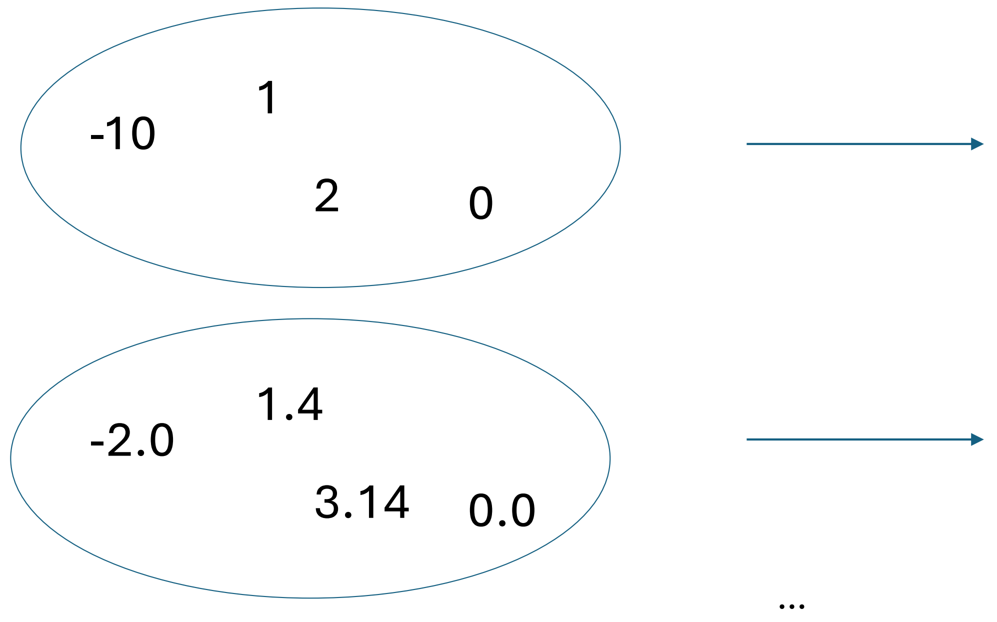
Expressions are evalued to obtain values.

Functions map inputs into an output.

Functions are computations or actions (with side effects).

A program is composed of data, computations and actions.

Values and types



integer
(supports arithmetic operations, ...)

float
(supports arithmetic operations, ...)

A **type** is a grouping of values (with associated operations) with similar digital representation

Ken's method for function design

For each function:

1. Write a brief description of what the function should do
2. Find a name for the function
3. Write down examples
4. Find out the type of inputs and outputs
5. Generate code for the function (and possibly helper functions)
6. Write test cases
7. Write short documentation for the function

Ken Friis Larsen

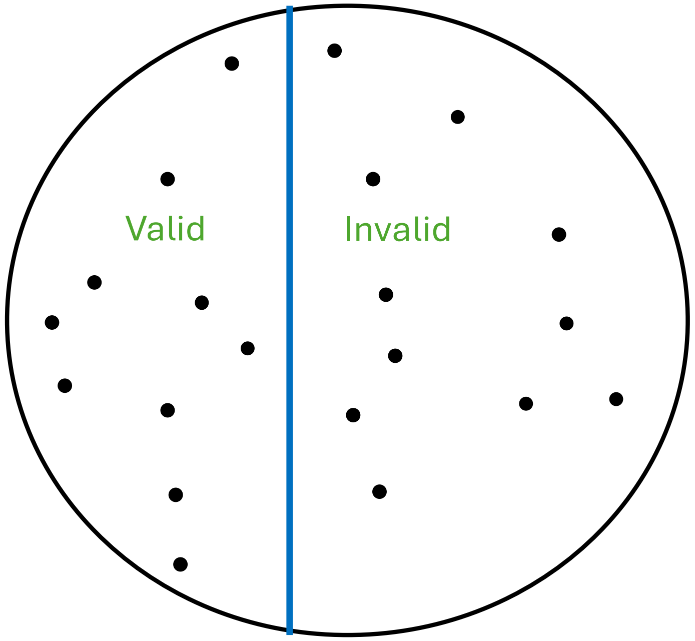
Testing

- We focus on unit testing (testing a single function)
- Tests are a systematic way to exhibit bugs
  - A but is a discrepancy between what the function should do (its **specification**) and what it actually does (the code)
- A test is a collection of test cases
  - Each test case compares what we expect (based on the specification) and what we obtain (based on calling the function) with well defined input(s).

How to choose which input to test a function with?

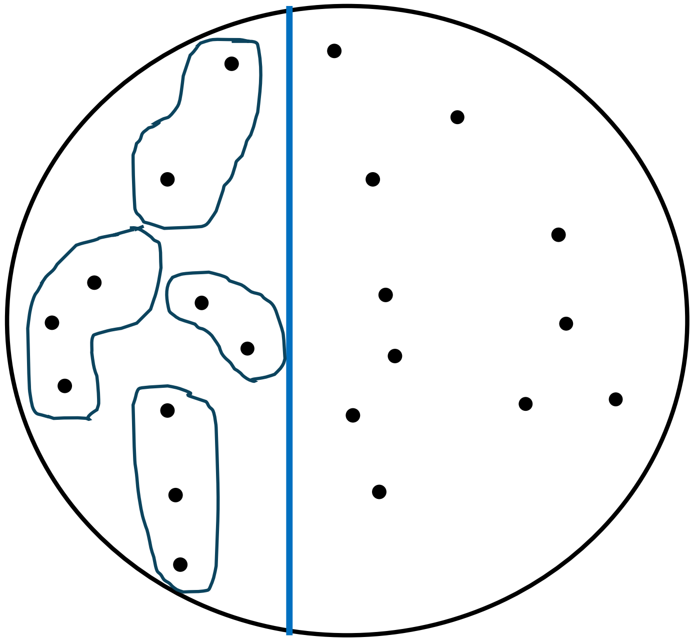
Testing – Input partitioning

- Idea:
    1. Consider valid inputs
    2. Partition valid inputs into a *finite* number of disjoint subsets equally 'likely' to manifest an error.
    3. Test suite: Choose *few* (1 to 3) representative values from *each* equivalence class.



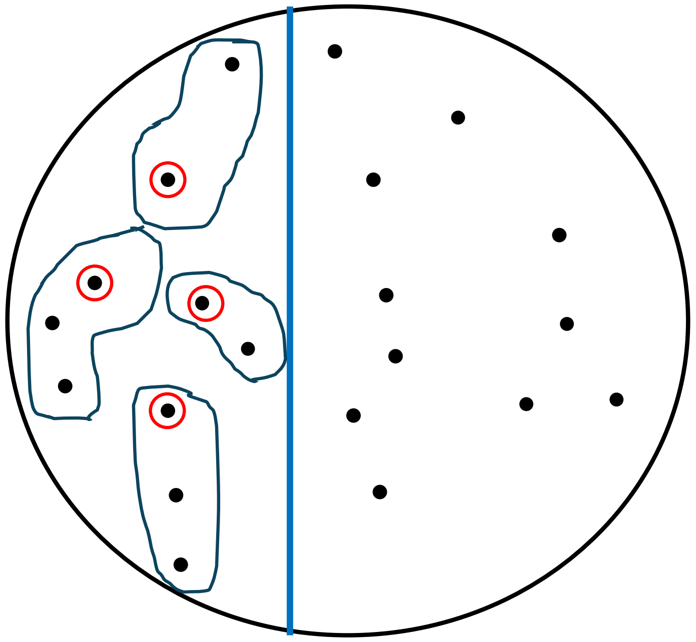
Testing – Input partitioning

- Idea:
  1. Consider valid inputs
  2. Partition valid inputs into a *finite* number of disjoint subsets equally 'likely' to manifest an error.
  3. Test suite: Choose *few* (1 to 3) representative values from *each* equivalence class.



Testing – Input partitioning

- Idea:
  1. Consider valid inputs
  2. Partition valid inputs into a *finite* number of disjoint subsets equally ‘likely’ to manifest an error.
  3. Test suite: Choose *few* (1 to 3) representative values from *each* equivalence class.

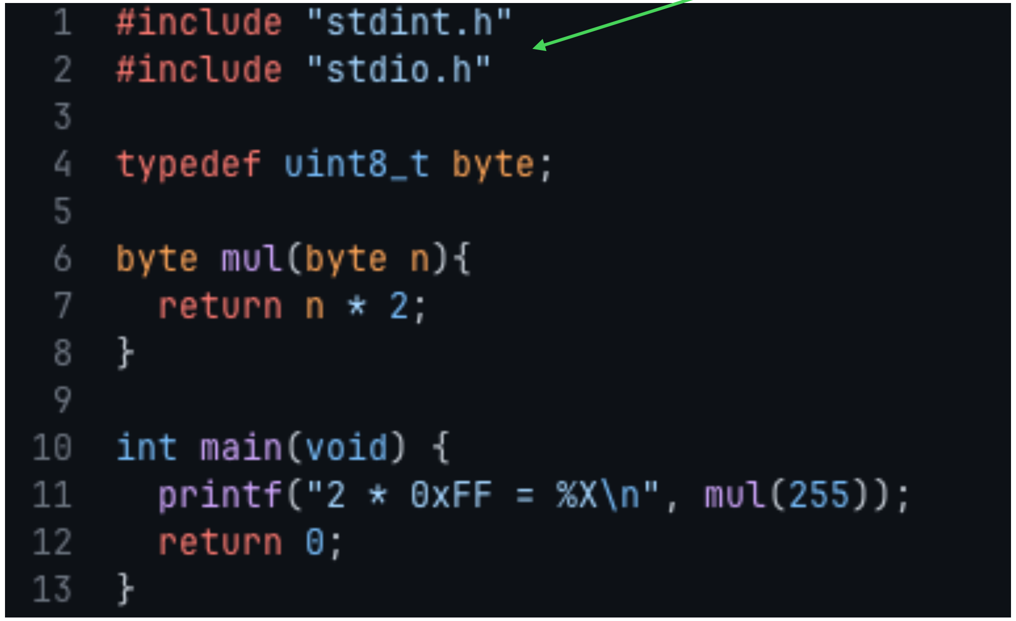


PoP programming language concepts in C

- Mutable bindings with variables as default
- Basic types are a bit of a mess (more in a couple of slides)
  - Arrays are a type in C; Lists are not (linked list are built with pointers)
  - `struct` is a collection of named fields (accessed by name as opposed to tuples in Python where data is accessed by index)
- Statements (terminated by ;) and expressions (within statements)
- Blocks of code within { }
  - Indentation important for reading code, but not as crucial as in Python or F#
- Explicit control flow (conditionals with if, loops with for and while)
  - Should not be used in assignment 1!!
- Explicit types for variables, function parameters and return values
  - C23 introduces type inference (as in F# and Python)
- Return statement in function (as in Python)
- `main` is entry point in a program that is executed
  - More next slide
- No modules, but modularity by convention with:
  - Header files (.h) that contain definitions, and
  - Implementation files (.c) that contain implementation
- Generic types are beyond the scope of HPPS

The main feature of C that was not part of PoP is explicit memory management with **pointers**. More on this in a week.

A simple C program



```c
#include "stdint.h"
#include "stdio.h"

typedef uint8_t byte;

byte mul(byte n){
  return n * 2;
}

int main(void) {
  printf("2 * 0xFF = %X\n", mul(255));
  return 0;
}
```

Equivalent of import module:
We include the definitions from header files, whose implementation is either in the C standard library or defined locally.

Type definition (with typedef keyword).
The type `uint8_t` is defined in stdint.h.

Function signature

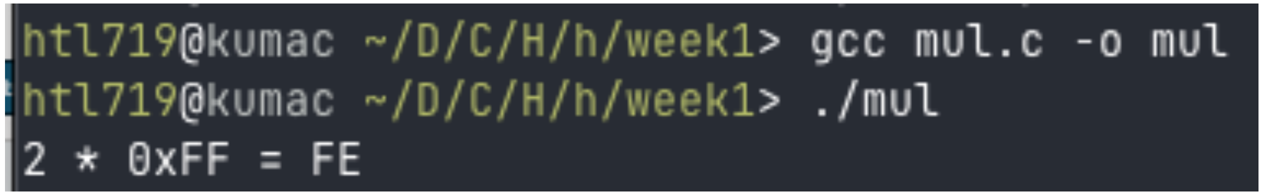
Function body (as code block within `{ }`)

Main function, entry point in the executable program. No argument (`void` type).

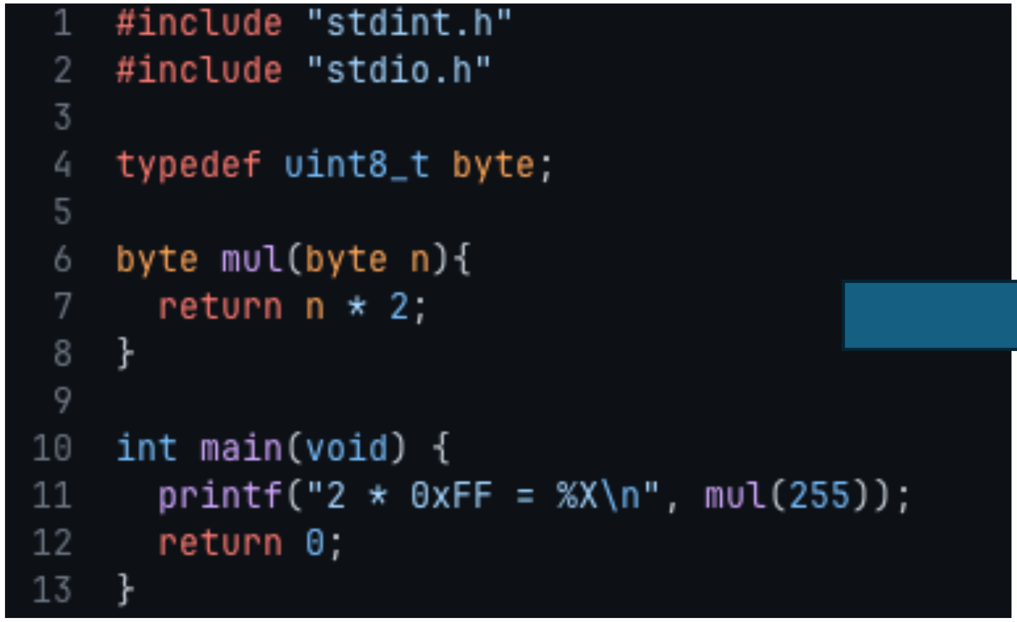
Print statement, `printf` is defined in stdio.h

return statement: 0 all good, anything else indicates an error.

Compilation



```
htl719@kumac ~/D/C/H/h/week1> gcc mul.c -o mul
htl719@kumac ~/D/C/H/h/week1> ./mul
2 * 0xFF = FE
```

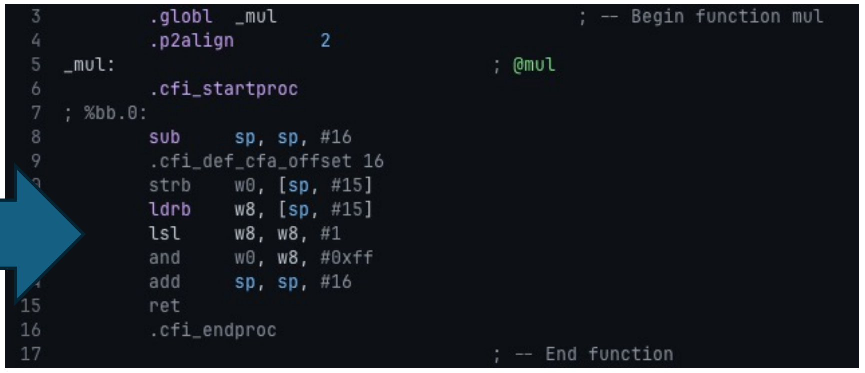


```c
#include "stdint.h"
#include "stdio.h"

typedef uint8_t byte;

byte mul(byte n){
  return n * 2;
}

int main(void) {
  printf("2 * 0xFF = %X\n", mul(255));
  return 0;
}
```



```
        .globl  _mul                                    ; -- Begin function mul
        .p2align        2
_mul:                                           ; @mul
        .cfi_startproc
; %bb.0:
        sub     sp, sp, #16
        .cfi_def_cfa_offset 16
        strb    w0, [sp, #15]
        ldrb    w8, [sp, #15]
        lsl     w8, w8, #1
        and     w0, w8, #0xff
        add     sp, sp, #16
        ret
        .cfi_endproc
                                                ; -- End function
```

```
$ gcc -S mul.c
```

Arithmetic interpretation of bit vectors

1. Finite representation
   - There is a limit to the number of integers that can be represented on a fixed number of bytes => min and max values.
2. Representing positive and negative integers
   - Sign and values must be represented

Bits are just bits. We interpret them in different ways!

Finite size for digital representation

Integer types defined over 8, 16, 32 or 64 bits, i.e., 1, 2, 4 or 8 bytes.

The size of original C types is platform dependent: char, short, int, long.

Stdint.h introduces types with explicit, platform independent sizes:

```
int8_t   uint8_t
int16_t  uint16_t
int32_t  uint32_t
int64_t  uint64_t
```

Integer values

- Decimal, binary (0b), hexadecimal (0x)

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

Unsigned integers – B2U

- X is a bit vector of size w

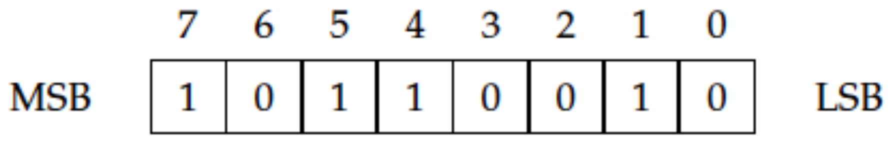
$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

- Let us take w = 8
  - X = 0b00000000 = 0x00 => B2U(0x00) = $0*2^7+ 0*2^6+ 0*2^5+ 0*2^4+ 0*2^3+ 0*2^2+ 0*2^1+ 0*2^0$ = 0
  - X = 0b11111111 = 0xFF => B2U(0xFF) = $1*2^7+ 1*2^6+ 1*2^5+ 1*2^4+ 1*2^3+ 1*2^2+ 1*2^1+ 1*2^0$ = 255
  - X = 0b10010001 = 0x91 => B2U(0x91) = $1*2^7+ 0*2^6+ 0*2^5+ 1*2^4+ 0*2^3+ 0*2^2+ 0*2^1+ 1*2^0$ = 145

$2^0 = 1$
$2^1 = 2$
$2^2 = 4$
$2^3 = 8$
$2^4 = 16$
$2^5 = 32$
$2^6 = 64$
$2^7 = 128$

Min: 0
Max: 255

Signed integers – Naïve interpretation



- One option is to take the Most Significant Bit as a sign bit and then B2U interpretation for the rest of the bits
  - The bit vector above would be – (because of bit 7), then B2U(0b0110010) = 32+16+2 = 50 => -50
  - Min is -127, max is 127

Signed integers – Two's complement B2T

- X is a bit vector of size w

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

- Let us take w = 8
  - X = 0b00000000 = 0x00 => B2U(0x00) = 0*2^7+ 0*2^6+ 0*2^5+ 0*2^4+ 0*2^3+ 0*2^2+ 0*2^1+ 0*2^0 = 0
  - X = 0b10000000 = 0x7F => B2U(0x91) = 1*2^7+ 0*2^6+ 0*2^5+ 0*2^4+ 0*2^3+ 0*2^2+ 0*2^1+ 0*2^0 = 127
  - X = 0b11111111 = 0xFF => B2U(0xFF) = -1*2^7+ 1*2^6+ 1*2^5+ 1*2^4+ 1*2^3+ 1*2^2+ 1*2^1+ 1*2^0 = -128+64+32+16+8+4+2+1 **= -1**
  - X = 0b10000000 = 0x80 => B2U(0x91) = 1*2^7+ 0*2^6+ 0*2^5+ 0*2^4+ 0*2^3+ 0*2^2+ 0*2^1+ 0*2^0 = -128

$2^0 = 1$
$2^1 = 2$
$2^2 = 4$
$2^3 = 8$
$2^4 = 16$
$2^5 = 32$
$2^6 = 64$
$2^7 = 128$

Min: -128
Max: 127

Two complement’s method

How to get the binary representation of a negative number with Two’s complement?

1. Binary representation of corresponding positive value on w bits
2. Invert all digits (0 becomes 1; 1 becomes 0)
3. Add one

Example:

How to represent -17 on 8 bits?

1. 17 is represented is 16+1
   0x11 = 0b00010001
   00010001
2. 11101110
3. 11101111

B2T(11101111) = -17

Two complement’s method

How to get the binary representation of a negative number with Two’s complement?

1. Binary representation of corresponding positive value on w bits
2. Invert all digits (0 becomes 1;

3

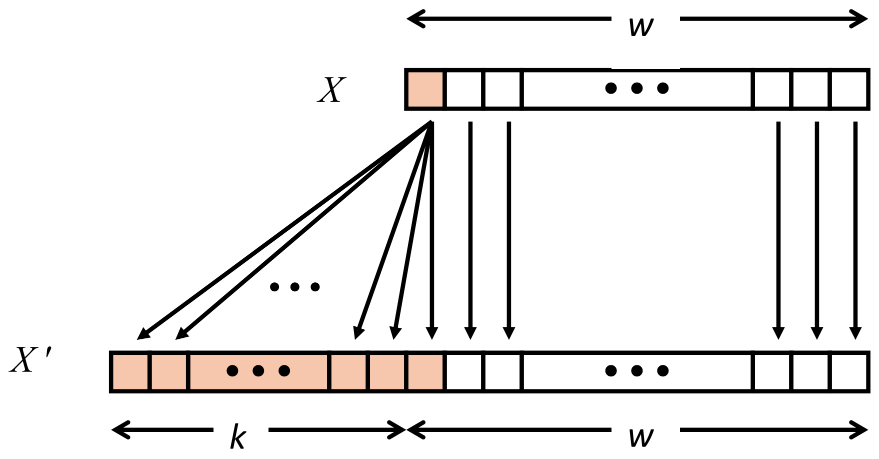
Example:

How to represent -17 on 8 bits?

1. 17 is represented is 16+1
   0x11 = 0b00010001
   00010001
2. 11101110
3. 11101111

To understand why it works, we need to learn about bitwise operations. Before that: arithmetic operations.

Sign extension



- Expanding (e.g., from 8 to 16 bits)
  - Unsigned: zeros added (on left)
  - Signed: sign extension)
  - Both yield expected result

- Truncating (e.g., from 16 to 8 bits)
  - Unsigned/signed: bits are truncated
  - Result reinterpreted
  - Unsigned => modulo operation
  - Signed: depends on bit pattern (large negative number might be truncated to positive number)

Integer arithmetic

- Adding two integers encoded on w bytes should take w+1 bytes
- How to encode the sum on w bytes?

No magic!! The sum **will overflow**

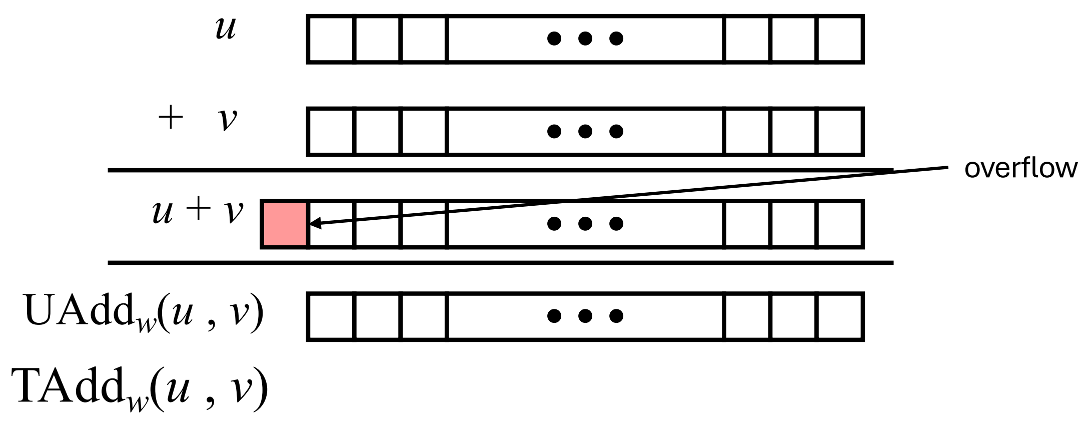
- Multiplying two integers encoded on w bytes should take 2.w bytes
- How to encode the product on w bytes?

No magic!! The product **will overflow**

Addition

Operands: *w* bits

True Sum: *w*+1 bits

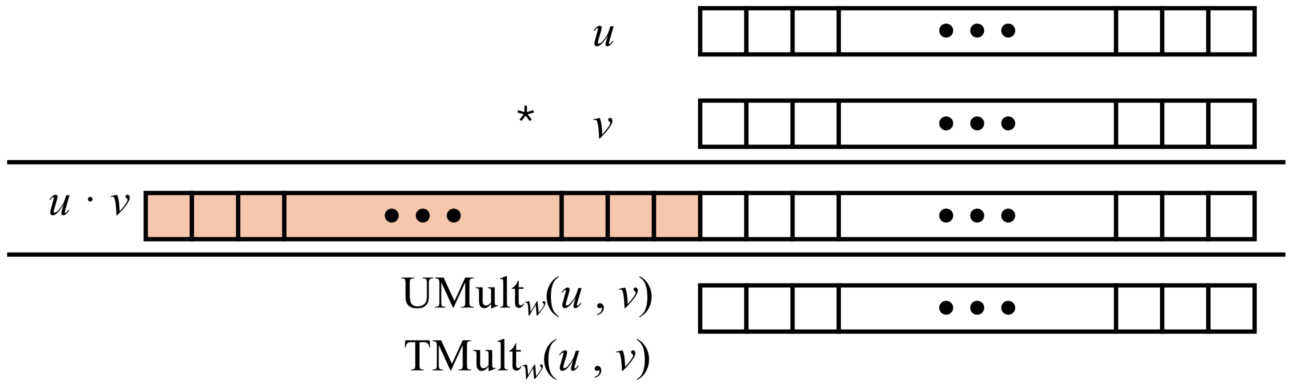


TAdd and UAdd have Identical **Bit-Level** Behavior.
$\mathrm{UAdd}_w(u\ ,\ v)$ is $u+v \bmod 2^w$
$\mathrm{TADD}_w(u,v)$ might lead to positive or negative number!

Multiplication

Operands: *w* bits

True Product: 2**w* bits



Ignores high order *w* bits

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

$\mathrm{TMult}_w(u, v) = \mathrm{U2T}(u.\ v \pmod{2^w})$, where $\mathrm{U2T}(u) = u_{u-1}.2^w + u$

Bitwise operations

- Bitwise operations apply to **bit vectors** (of size 8, 16, 32 or 64)
- Bitwise operations in any programming language (C notation here)
  - `&` : bitwise and
  - `|` : bitwise or
  - ^: bitwise xor
  - ~: bitwise negation
  - >>: rightshift
  - <<: leftshift

```
   00001101
&  00010000
   ------------
   00000000
```

Is bit 4 set?

& is used for **masks**, i.e., focusing on a bit pattern.

Bitwise operations

- Bitwise operations apply to **bit vectors** (of size 8, 16, 32 or 64)
- Bitwise operations in any programming language (C notation here)
  - `&` : bitwise and
  - `|` : bitwise or
  - ^: bitwise xor
  - ~: bitwise negation
  - >>: rightshift
  - <<: leftshift

```
   00001101
|  11011101
   ------------
   11011101
```

Bitwise operations

- Bitwise operations apply to **bit vectors** (of size 8, 16, 32 or 64)
- Bitwise operations in any programming language (C notation here)
  - `&` : bitwise and
  - `|` : bitwise or
  - `^`: bitwise xor
  - `~`: bitwise negation
  - `>>`: rightshift
  - `<<`: leftshift

```
    00001101
^   11011101
   ------------
   11010000
```

Bitwise operations

- Bitwise operations apply to **bit vectors** (of size 8, 16, 32 or 64)
- Bitwise operations in any programming language (C notation here)
  - `&` : bitwise and
  - `|` : bitwise or
  - `^`: bitwise xor
  - `~`: bitwise negation
  - `>>`: rightshift
  - `<<`: leftshift

```
~   00001101
    ------------
    11110010
```

Bitwise operations

- Bitwise operations apply to **bit vectors** (of size 8, 16, 32 or 64)
- Bitwise operations in any programming language (C notation here)
  - `&` : bitwise and
  - `|` : bitwise or
  - `^`: bitwise xor
  - `~`: bitwise negation
  - `>>`: logical rightshift (unsigned)
  - `<<`: logical leftshift (unsigned)

```
00001101 >> 1
------------
00000110
```

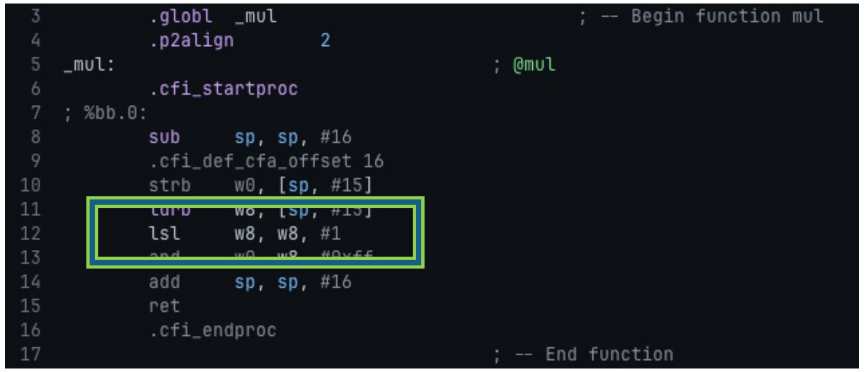
Bitwise operations

- Bitwise operations apply to **bit vectors** (of size 8, 16, 32 or 64)
- Bitwise operations in any programming language (C notation here)
  - `&` : bitwise and
  - `|` : bitwise or
  - `^`: bitwise xor
  - `~`: bitwise negation
  - `>>`: logical rightshift (unsigned)
  - `<<`: logical leftshift (unsigned)

```
00001101 << 1
------------
00011010
```

Bitwise operations

- Bitwise operations apply to **bit vectors** (of size 8, 16, 32 or 64)
- Bitwise operations in any programming language (C notation here)
    - `&` : bitwise and
    - `|` : bitwise or
    - `^`: bitwise xor
    - `~`: bitwise negation
    - `>>`: logical rightshift (unsigned)
    - `<<`: logical leftshift (unsigned)



```
        .globl  _mul                            ; -- Begin function mul
        .p2align        2
_mul:                                   ; @mul
        .cfi_startproc
; %bb.0:
        sub     sp, sp, #16
        .cfi_def_cfa_offset 16
        strb    w0, [sp, #15]
        ldrb    w8, [sp, #15]
        lsl     w8, w8, #1
        and     w0, w8, #0xff
        add     sp, sp, #16
        ret
        .cfi_endproc
                                        ; -- End function
```

Logical Left Shift (LSL) by 1 is multiplication by 2!
See slide on “Power-of-2 multiply with shift”

Bitwise operations

- Bitwise operations apply to **bit vectors** (of size 8, 16, 32 or 64)
- Bitwise operations in any programming language (C notation here)
  - `&` : bitwise and
  - `|` : bitwise or
  - `^`: bitwise xor
  - `~`: bitwise negation
  - `>>`: rightshift
  - `<<`: leftshift

Bitwise operations applied on all bits in parallel!

Logical interpretation of bit vectors

- 00000000 is true; any sequence containing a 1 is false.
- Boolean operators && (and), (||) or and ! (negation) apply on Boolean interpretation of
- Consider X a bit vector
  - X interpreted as a Boolean expression is true if X != 0
  - !X is true if X == 0
  - !!X transforms any sequence of bit vector that contains a 1 into all 1s, while leaving 0 all 0s.

Bitwise tricks (with signed integers)

~x + x = ~x & x = 11111111_2 = -1

=> -x = ~x + 1 (our method to find the representation of a negative number)

x & -x = x & (~x + 1)

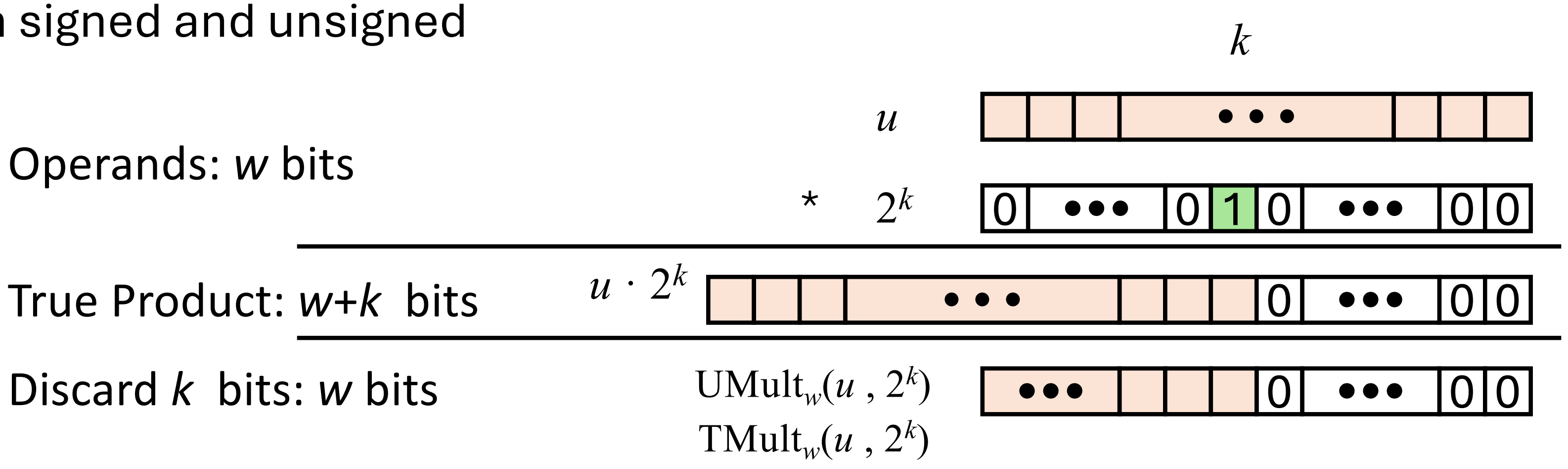
=> a mask isolating the right most 1 bit in x : 01001000 => 00001000

```
x =  01001000 ; ~x = 10110111              10111000
                   + 00000001            & 01001000
                     ------------          ------------
                      10111000               00001000
```

Power-of-2 Multiply with Shift

`u << k` gives **`u * `** ***2^k^***

Both signed and unsigned



Operands: *w* bits
True Product: *w*+*k* bits
Discard *k* bits: *w* bits
k
u
$*$ 2^k
0 ••• 0 1 0 ••• 0 0
$u \cdot 2^k$
••• 0 ••• 0 0
$\mathrm{UMult}_w(u\ ,\ 2^k)$
$\mathrm{TMult}_w(u\ ,\ 2^k)$
••• 0 ••• 0 0

Examples

```
u << 3             == u * 8
u << 5 - u << 3    == u * 24
```

Take-aways

1. Bits are just bits. Fixed size of bit vector defined by type.
2. Unsigned and signed integer interpretations.
3. No magic. Additions and multiplications are truncated
4. Bitwise operations are parallel operations on bits!

Quizz

- How do you test whether bit 1 is set?
- How do you test whether any bit is set to 1?
- How do you test if a single bit is set to 1?
- How do you count the number of bits set to 1?

Quizz answers

- How do you test whether the Ack flag (0x10) is set?

  (X & 0x10)

  => interpreted as false if flag not set, true otherwise
- How do you test whether any flag is set?

  (X)
  => interpreted as false if no flag is set , true otherwise
- How do you test if a single flag is set?

  X && ((X & - X) == X)

  => X makes sure that X is not 0
  (X & -X) isolates the right most bit
  If this is equal to X then a single bit is set.

Quizz answers

- How do you count the number of bits set?
- Trivial for 2x1 bits with addition

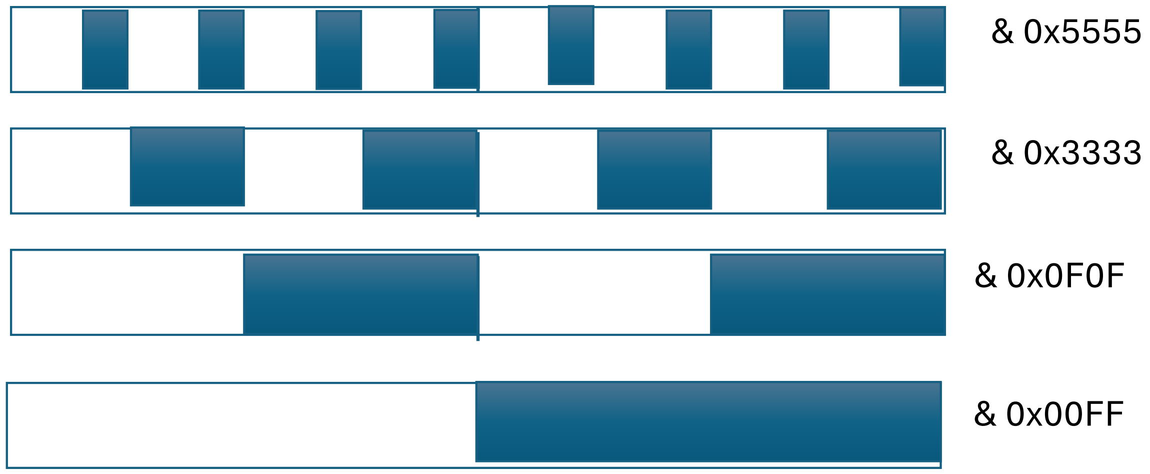
• 0+0 -> 00	0 bit set	
• 0+1 -> 01	1 bit set	Result fits in 2 bits
• 1+0 -> 01	1 bit set	
• 1+1 -> 10	2 bits set	

- Divide and conquer
  - Consider 2 bytes, i.e., 16 bits
  - Count bits 1+1 (results fits in 2 bits), then 2+2 (result fits in 4 bits), then 4+4 (result fits in 8 bits), then 8+8 (result fits in 16 bits).

Quizz answers

x is 2B

Masks



Bitwise operations in parallel

```
x = (x & 0x5555) + ((x >> 1) & 0x5555)
x = (x & 0x3333) + ((x >> 2) & 0x3333)
x = (x & 0x0F0F) + ((x >> 4) & 0x0F0F)
x = (x & 0x00FF) + ((x >> 8) & 0x00FF)
```