High performance parallel systems

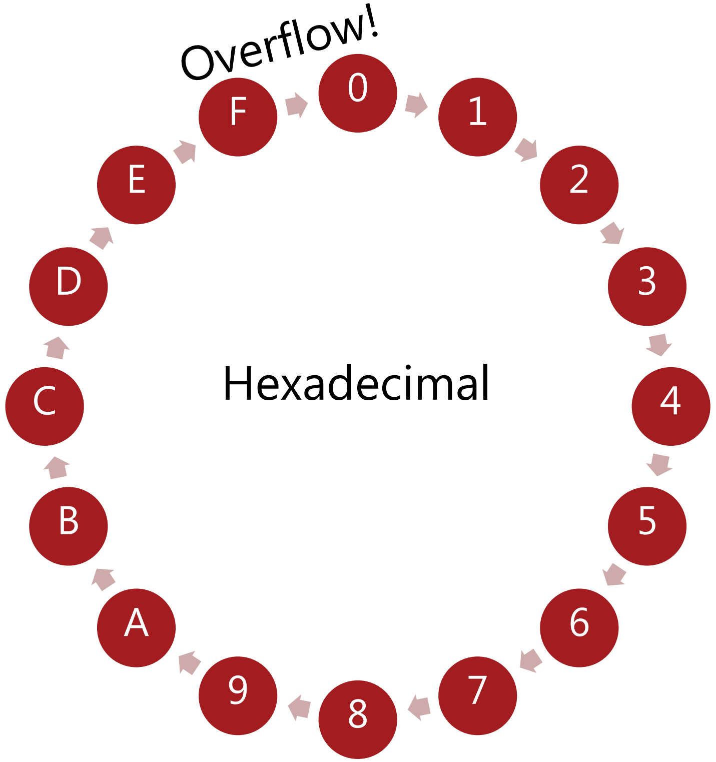
Lecture 2 – Numbers and the C programming language

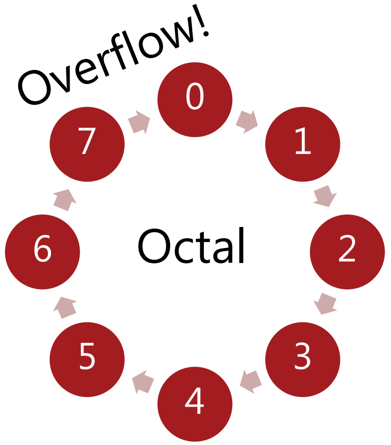
Kenneth Skovhede, NBI, 2020-11-19

UNIVERSITY OF COPENHAGEN

Recap: Numbers in a digital system

Binary (0b)	Decimal	Octal (0c)	Hexadecimal (0x)
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	10	8
1001	9	11	9
1010	10	12	A
1011	11	13	B
1100	12	14	C
1101	13	15	D
1110	14	16	E
1111	15	17	F
10000	16	20	10





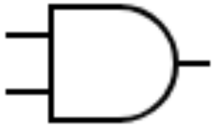
Recap: Addition with binary numbers

$$\begin{array}{r}
1\ 1\ 1 \\
\underline{0111}\ (7) \\
+\underline{0011}\ (3) \\
\hline
1010\ \underline{(10)}
\end{array}$$

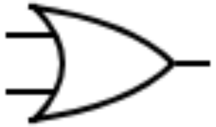
Recap: Basic gates

AND	0	1
0	0	0
1	0	1

AND

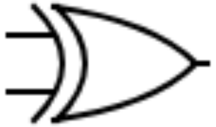


OR

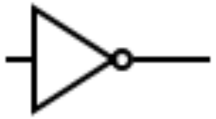


OR	0	1
0	0	1
1	1	1

XOR	0	1
0	0	1
1	1	0



XOR



Not

NOT	0	1
	1	0

Building an adder

A+B = S

0+0 = 00
0+1 = 01
1+0 = 01
1+1 = 10

+	0	1
0	0	1
1	1	0

Same as XOR

Carry	0	1
0	0	0
1	0	1

Same as AND

Building an adder

A+B = S

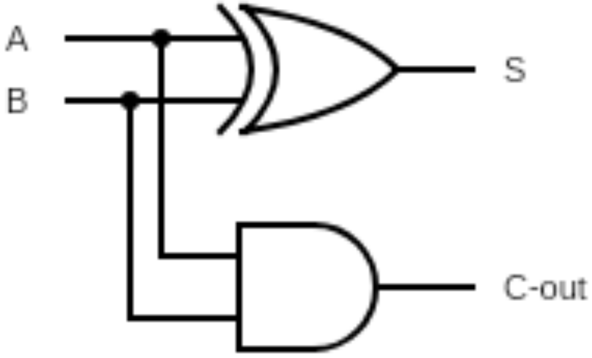
0+0 = 00
0+1 = 01
1+0 = 01
1+1 = 10

+	0	1
0	0	1
1	1	0

Same as XOR

Carry	0	1
0	0	0
1	0	1

Same as AND



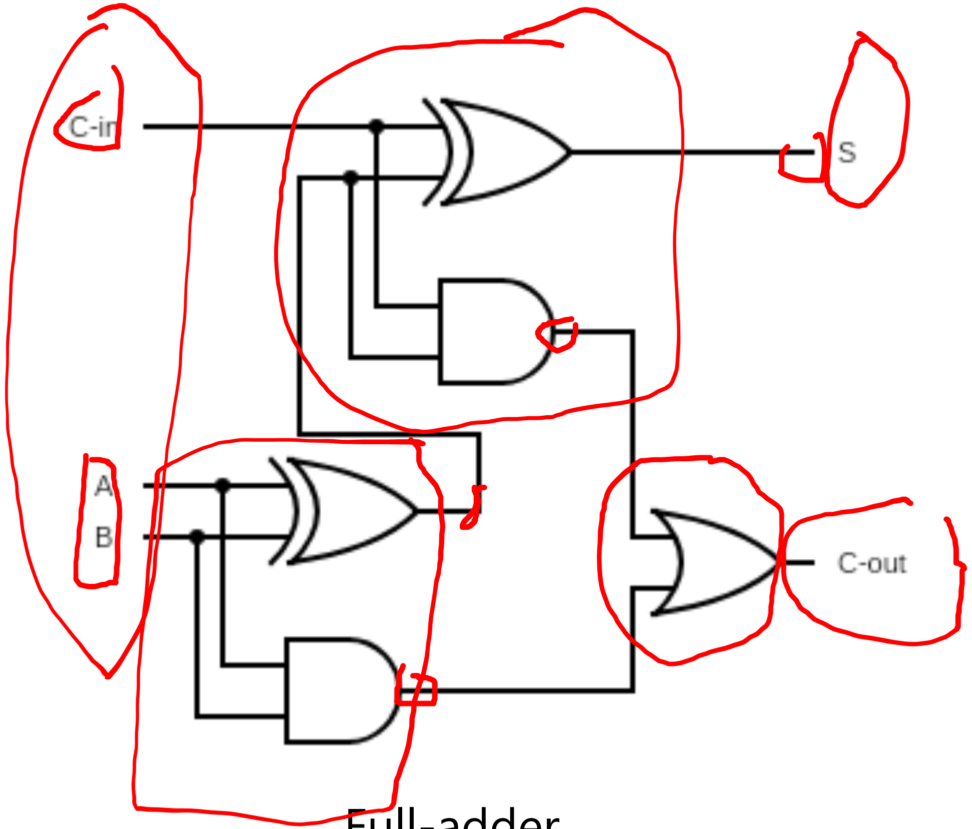
Half-adder

1 1 1
0111 (7)
+0011 (3)
1010 (10)

Building an adder

A+B = S

0+0+0 = 00
0+0+1 = 01
0+1+0 = 01
0+1+1 = 10
1+0+0 = 01
1+0+1 = 10
1+1+0 = 10
1+1+1 = 11

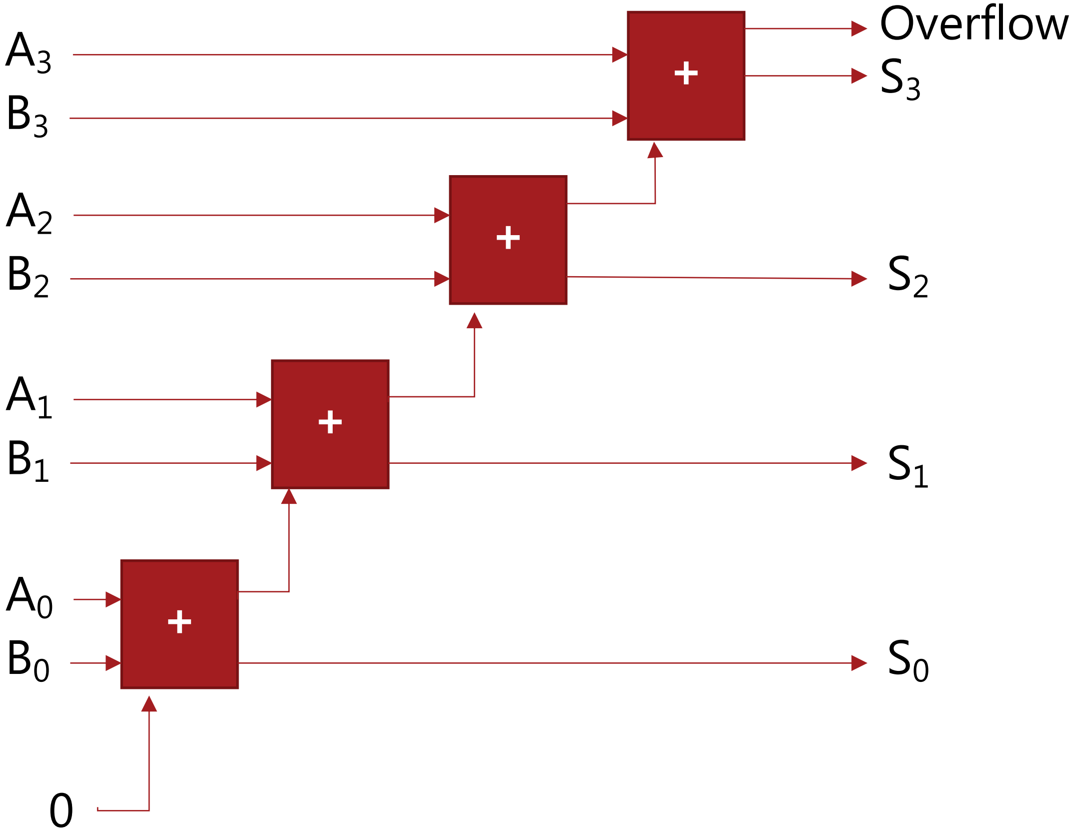


Full-adder

3 inputs
2 outputs

Ripple Carry Adder

```
   1 1 1
   0111 (7)
  +0011 (3)
  --------
   1010 (10)
```



Negative numbers

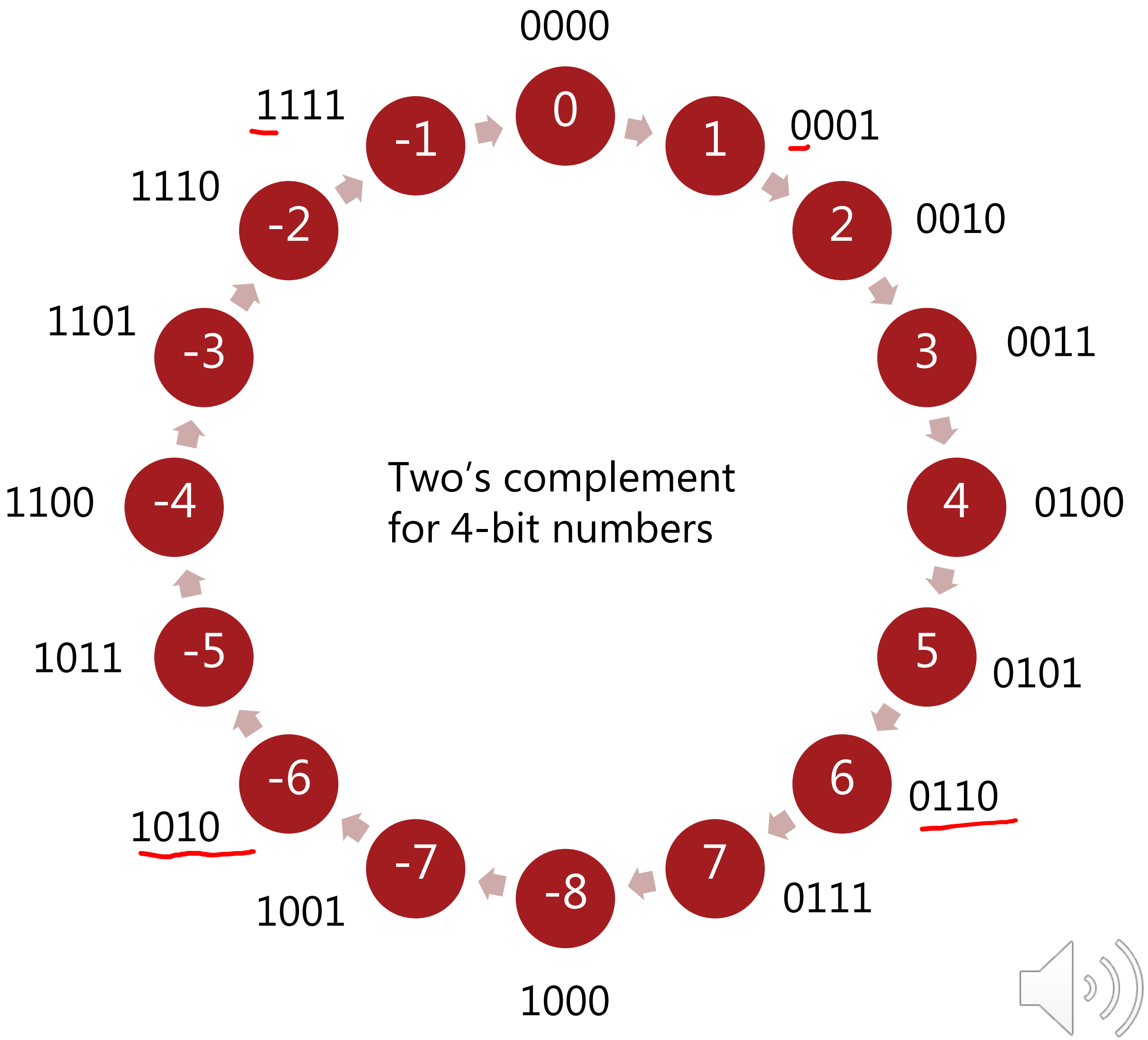
Bits	Min	Max
4	-8	7
8	-128	127
16	-16384	16383
32	-2147483648	2147483647

Negation is bitwise invert + 1

-6 => 0b1010
~0b1010 + 0b0001 =
0b0101 + 0b0001 = 0b0110 = 6

Odd behavior due to non-symmetry

- -(-8) = -8
- Abs(-8) = -8
- -8 * -1 = -8



Try it yourself: Addition with Two's complement

Try to compute these basic math problems in binary.
Remember that a negating a number in two's complement
is equivalent to a bitwise inversion + 1

Binary numbers:	Result (in binary)	Decimal
0b0010 + 0b1010 =		2 + -6 = -4
0b0101 - 0b1010 =		
0b0111 - 0b1000 =		

Try it yourself: Addition with Two’s complement

Try to compute these basic math problems in binary.
Remember that a negating a number in two’s complement is equivalent to a bitwise inversion + 1

Binary numbers:	Result (in binary)	Decimal
0b0010 + 0b1010 =	0b1100	2 + -6 = -4
0b0101 - 0b1010 =	0b1011	5 - -6 = -5 (11)
0b0111 - 0b1000 =	0b1111	7 - -8 = -8 (15)

Multiplication with powers of two

Number	*2 or << 1	*4 or << 2	*8 or << 3
0000 0001 (1)	0000 0010 (2)	0000 0100 (4)	0000 1000 (8)
0000 1001 (9)	0001 0010 (18)	0010 0100 (36)	0100 1000 (72)
0001 0110 (22)	0010 1100 (44)	0101 1000 (88)	1011 0000 (176)
1111 0001 (241)	1110 0010 (226)	1100 0100 (196)	1000 1000 (136)
1000 1000 (136)	0001 0000 (16)	0010 0000 (32)	0100 0000 (64)

In binary mode, shifting n bits left is equivalent to multiplying by 2^n

Shifting right is equivalent to dividing by a power of 2

Special instructions are required for negative numbers where the sign bit must be preserved

Real numbers

Fixed point numbers

$$\begin{array}{r} {\scriptstyle 1\;\;1\;\;\;} \\ 123.45 \\ +235.56 \\ \hline 359.01 \end{array} \quad \longleftrightarrow \quad \begin{array}{r} {\scriptstyle 1\;\;1\;\;\;} \\ 12345 \\ +23556 \\ \hline 35901 \end{array}$$

Currency is often required to use fixed point

Cannot handle ranges, like 1km + 1nm

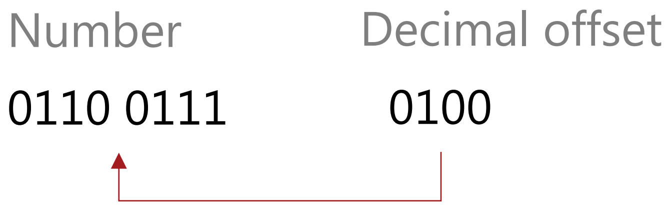
Floating point numbers

0.0000012345

123450000.0

Despite a large numeric range, the precision is limited due to the limited number of bits

One suggestion is to use two values



Number	Decimal offset
0110 0111	0100

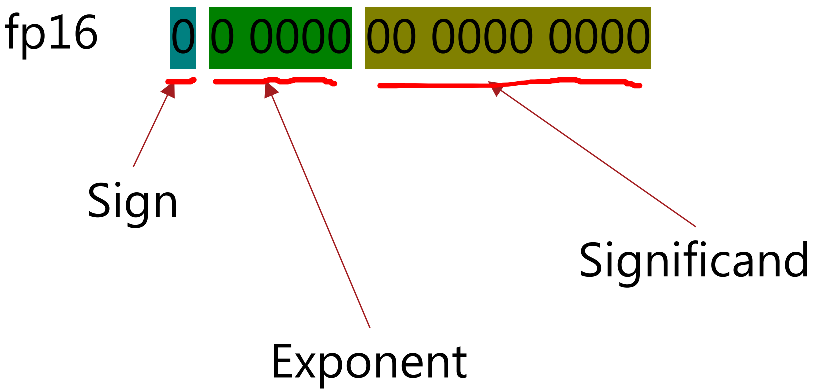
Simpler if we base it on scientific notation

$\underline{n * 10^{e}}$

But binary: $\underline{n * 2^{e}}$

Floating point - IEEE-754

Name	Bits	Exponent bits	Significand bits	Decimal
Half precision	16	5 (-14 to +15)	10(+1) (0 to 1023)	3.31
Single precision	32	8 (-126 to +127)	23(+1)	7.22
Double precision	64	11 (-1022 to +1023)	52(+1)	15.95



fp16 0 0 0000 00 0000 0000

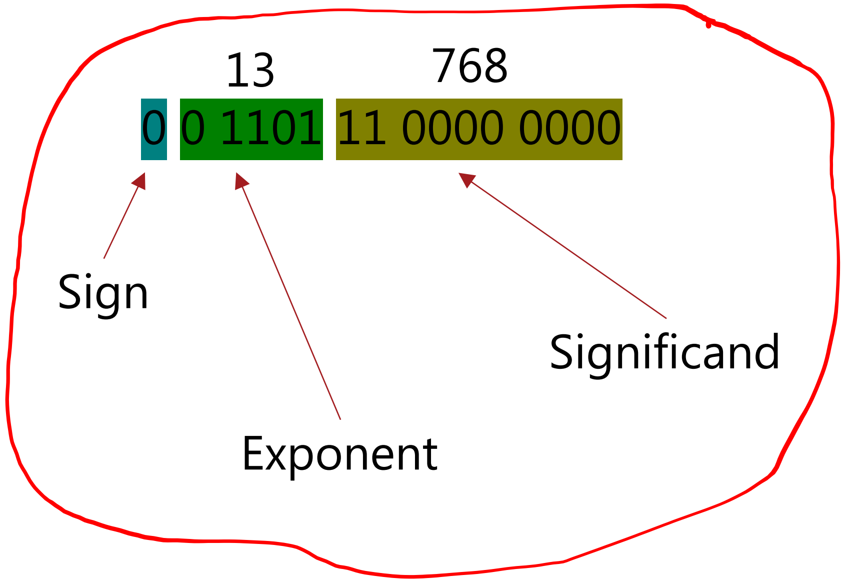
Sign

Exponent

Significand

For fp32, visit: https://www.h-schmidt.net/FloatConverter/IEEE754.ht

Floating point - IEEE-754



Actual CPU implementation can differ,
but any stored value must follow this format

The 11th "bit"

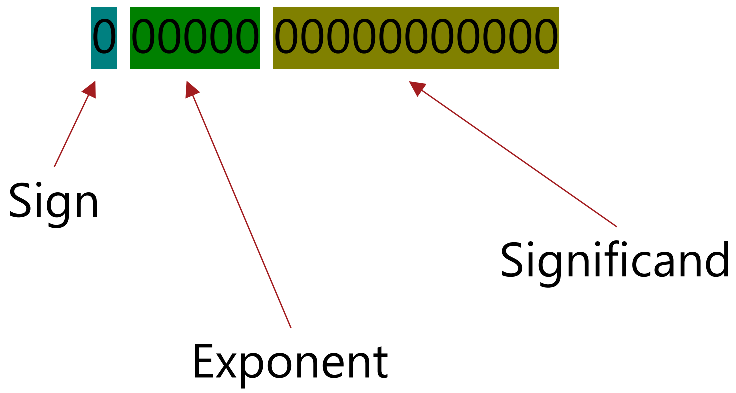
$$\text{sign} = 1$$

$$\text{exponent} = \underline{13-\boxed{(2^4-1)}} = \underline{-2}$$

$$\text{value} = \underline{\text{sign} * 2^{\text{exponent}} * (1 + \text{significand} / 1024)} = \boxed{1} * \underline{2^{-2}} * (\boxed{1} + \underline{(768/1024)}) = \underline{0.4375}$$

IEEE – Special numbers

Exponent	Significand	Notes
Not all zero and not all ones	Implicit 1.xxx	Normalized form, most common
All zero	Implicit 0.xxx	Denormalized form, more accuracy for numbers -1 < n < 1
All ones	All zero	Infinity, either +inf or -inf
All ones	Non-zero	Not-a-number, NaN



0 00000 00000000000

Sign

Significand

Exponent

IEEE – Floating point math

Significand is 2^{10}, meaning [0:1023]

```
    13        845
  0 0 1101 11 0100 1101
+ 0 0 1101 10 0000 1000
    13        520
```

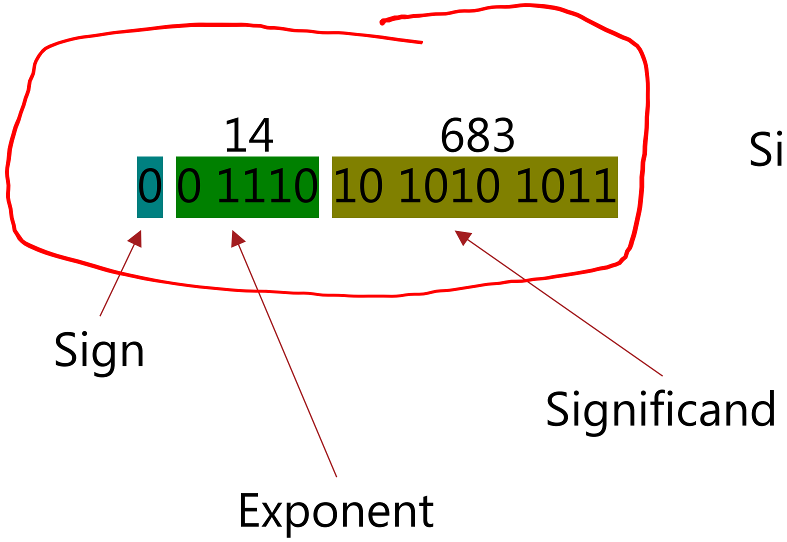
$= 1 * 2^{(13-15)} * (1+ (845/1024)) = 0.4562988281$

$= 1 * 2^{(13-15)} * (1+ (520/1024)) = 0.376953125$

1365 /1024 **0.833**2519531

0.8332519531 / 2^{-2} = 3.3330078124

0.8332519531 / 2^{-1} = 1.6665039062



14 683

0 0 1110 10 1010 1011

Sign

Exponent

Significand

Significand = (1 - 1.6665039062) * 1024 = 683

$= 1 * 2^{-1} * (1+ (683/1024)) =$ **0.833**4960938

This means that we have an error of 0.000244140625

IEEE – Floating point - denormalization

8 519

0 | 0 1101 | 10 0000 0111 $= 1 * 2^{(8-15)} * (1 + (519/1024)) = 0.01177215576$

- 0 | 0 1101 | 10 0000 1000 $= 1 * 2^{(8-15)} * (1 + (520/1024)) = 0.01177978516$

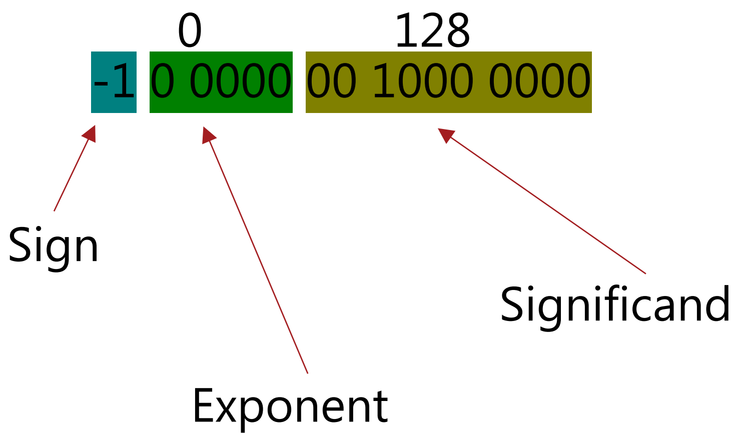
8 520

-0.000007629394531

$-0.000007629394531 / 2^{-7} = -0.00003051757812$

$-0.000007629394531 / 2^{-17} = -1.0$

Exponent range is [-14;15]



Denormalized we can describe it as:

$-1 * 2^{-15} * (0. + 128/1024)$

Try it yourself: Floating point

20 320
1 1 0100 01 0100 0000 = ?

16 584
0 1 0000 10 0100 1000 = ?

? ?
1/3 = ? ? ???? ?? ???? ????

? ?
12345 = ? ? ???? ?? ???? ????

sign = +1 or -1

exponent = exp-(2^4-1)

value = sign * 2^{exponent} * (1 + significand / 1024)

Try it yourself: Floating point

20 320

1 1 0100 01 0100 0000 $= -1 * 2^{(20-15)} * (1 + 320/1024) = -42$

16 584

0 1 0000 10 0100 1000 $= 1 * 2^{(16-15)} * (1 + 584/1024) = 3.140625$

Best representation of π with fp16

13 341

1/3 = 0 0 1101 01 0101 0101 $= 1 * 2^{(13-15)} * (1 + 341/1024) = 0.3332519531$

28 519

12345 = 0 1 1100 10 0000 0111 $= 1 * 2^{(28-15)} * (1 + 519/1024) = 12344$

sign = +1 or -1

exponent = $\exp-(2^4-1)$

value = sign * 2^{exponent} * (1 + significand / 1024)

Other ways to compute with real numbers

- Fractions: 1/3 + 1/3 = 2/3
- Decimal strings “123.7890” + “0.3333333333”
- Symbolic: ½ * π * 4 = 2π

The C programming language

The basics of a C program

```
#include <stdio.h>

/* Entry function is called main() */
int main() {
  // printf() writes to stdout
  printf("Hello, World!");
  // Zero means success
  return 0;
}
```

`#include` will copy the file into this file
We need stdio.h to get `printf()`

`/* */` block comments can span multiple lines

Starts the function `main()` returning an `int`eger

Single line comments end with the line

`printf()` is the primary output/debug method

`return` exits the function giving the result back

Blocks in C are written inside curly brackets `{ … }`

```
#include <stdio.h>
int main(){printf("Hello, World!");return 0;}
```

Statements end with semicolon `;`
Whitespace is ignored (generally)

C – easy to mess up

```
#include <stdio.h>
int main() {
  int b = 2;
  /* some code here */
  if (b < 2)
    //printf(“b = %d\n”, b);
  b = b * 2;
  return b;
}
```

What is the return value?

C – declaration and definition

```
#include <stdio.h>



int main() {
  int a = 1;
  int b = 2;
  int c = add(a, b);
  printf(“%d + %d = %d”, a, b, c);
}

int add(int a, int b) {
  return a + b;
}
```

Error: `add(int, int)` not declared

The method definition

C – declaration and definition

```
#include <stdio.h>
int add(int a, int b);

int main() {
  int a = 1;
  int b = 2;
  int c = add(a, b);
  printf(“%d + %d = %d”, a, b, c);
}

int add(int a, int b) {
  return a + b;
}
```

Declare the method

The method is declared, no problems

The method definition

Basic operators in C

Symbol	Description	Example
&	Bitwise AND	1 & 3 = 1
|	Bitwise OR	1 | 2 = 3
^	Bitwise XOR	1 ^ 2 = 3
< <	Left shift	1 << 2 = 4
> >	Right shift	4 >> 2 = 1
~	Bitwise NOT	~0b0011 = 0b1100

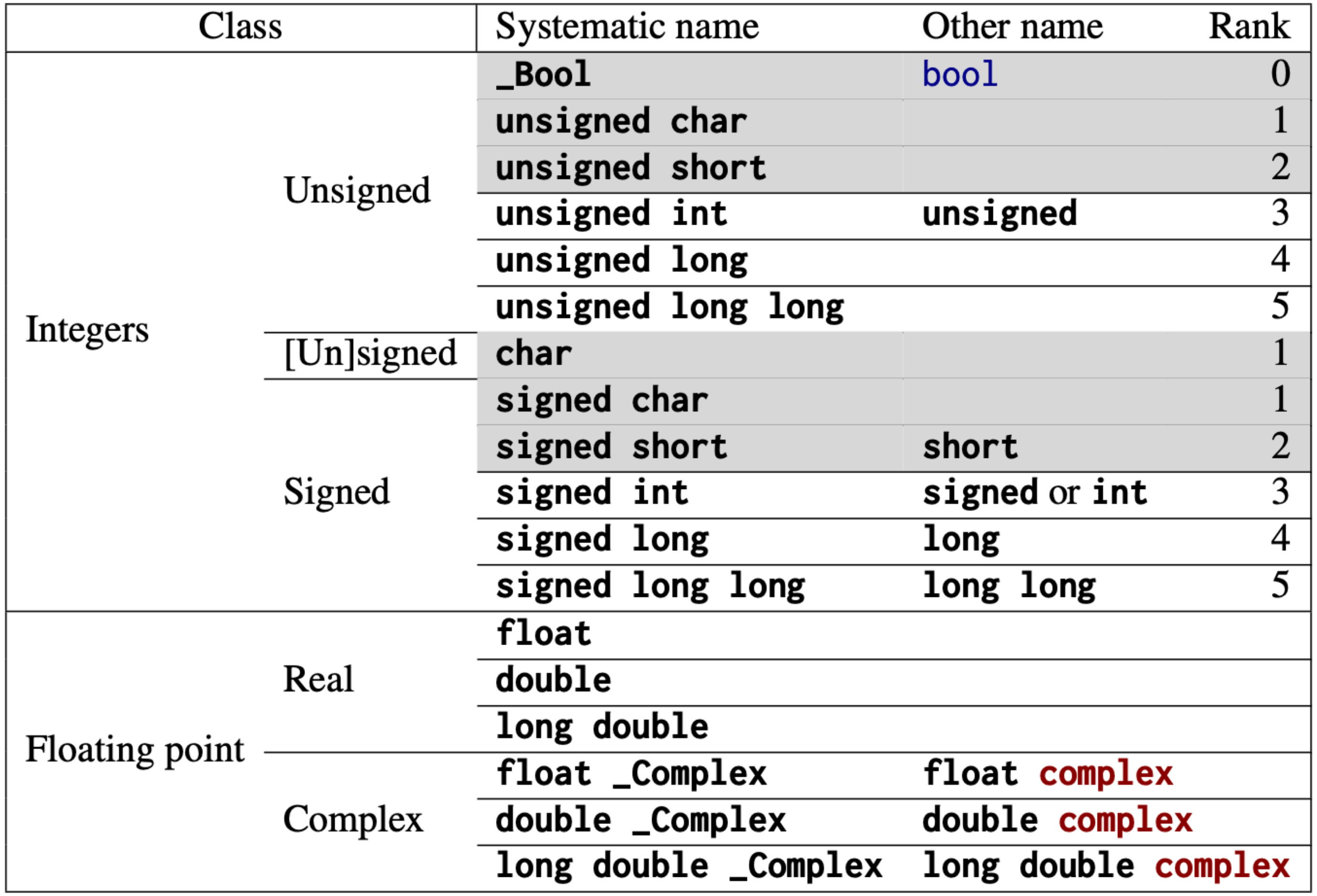
Symbol	Description	Example
&&	Logical AND	True && False = False
||	Logical OR	True || False = True
!=	Not Equal	True != False = True
!	Logical NOT	!True = False

Beware of logic and bitwise in C

```
                 01
int a = 0b0010;
if (a & 1) // Bitwise
{
    // Will not print as 0b0000 is treated as false
    printf("a & 1 = false\n");
}

if (a && 1) // Logical
{
    // Will print as 0b0010 AND 0b0001 are both treated as true
    printf("a && 1 = true\n");
}
```

C – types



Class		Systematic name	Other name	Rank
Integers	Unsigned	`_Bool`	`bool`	0
		`unsigned char`		1
		`unsigned short`		2
		`unsigned int`	`unsigned`	3
		`unsigned long`		4
		`unsigned long long`		5
	[Un]signed	`char`		1
	Signed	`signed char`		1
		`signed short`	`short`	2
		`signed int`	`signed` or `int`	3
		`signed long`	`long`	4
		`signed long long`	`long long`	5
Floating point	Real	`float`		
		`double`		
		`long double`		
	Complex	`float _Complex`	`float complex`	
		`double _Complex`	`double complex`	
		`long double _Complex`	`long double complex`	

Table from “Modern C” by Jens Gustedt

C – types with forced sizes

Signed name	Unsigned name	Bits
int8_t	uint8_t	8
int16_t	uint16_t	16
int32_t	uint32_t	32
int64_t	uint64_t	64

C – platform independence

```
int doubleup(int x) {
    if (x <= 0)
        x = 1;
    else
        x *= 2;
    return x;
}
```

```
doubleup:
    pushq %rbx          # Save caller registers
    subq $0x18, %rsp    # Allocate stack space

    movq $0, %rbx
    cmp %rax, %rbx      # Check input argument
    jg doubleup_nonzero

    movq $1, %rax        # Set to one
    jmp doubleup_done

doubleup_nonzero:
    movq $2, %rbx        # Multiply by 2
    imulq %rbx
doubleup_done:

    addq $0x18, %rsp    # Deallocate stack space
    popq %rbx           # Restore registers
    ret                 # Pop return address and
                        # returnto caller
```

x64 assembly

Compiling a program

```
#include <stdio.h>
int main() {
  printf("Hello, World!");
  return 0;
}
```

Contents of file `hello.c`

```
> c99 hello.c
```

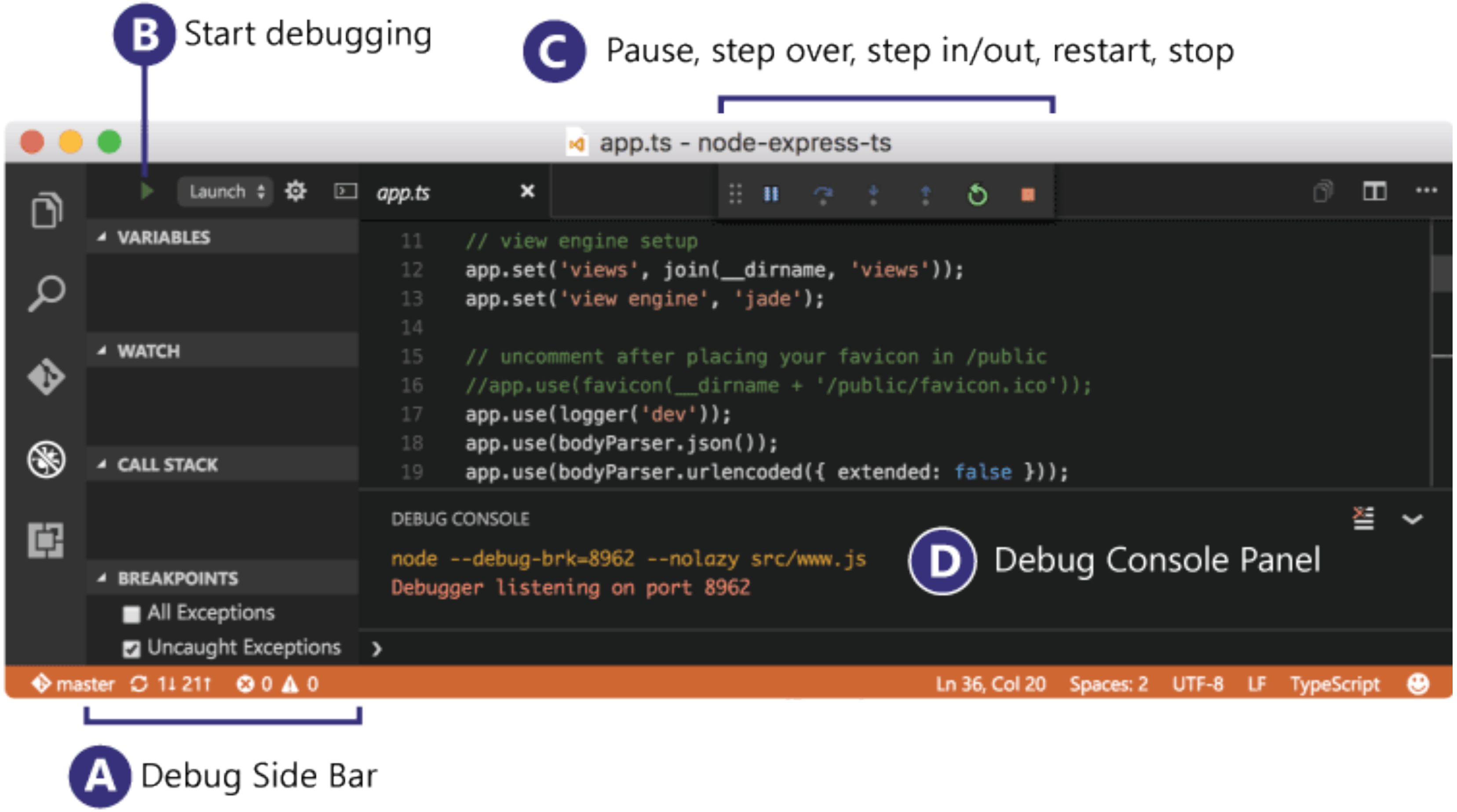
Produces a file named `a.out` ⋯ why?

```
> c99 -o hello hello.c
```

Using `-o` lets you pick the filename

Using an IDE to debug

I suggest trying out VS Code (*not* the same as Visual Studio...)
You need to install the C/C++ extension from Microsoft (`ms-vscode.cpptools`)



B Start debugging

C Pause, step over, step in/out, restart, stop

D Debug Console Panel

A Debug Side Bar

```
// view engine setup
app.set('views', join(__dirname, 'views'));
app.set('view engine', 'jade');

// uncomment after placing your favicon in /public
//app.use(favicon(__dirname + '/public/favicon.ico'));
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
```

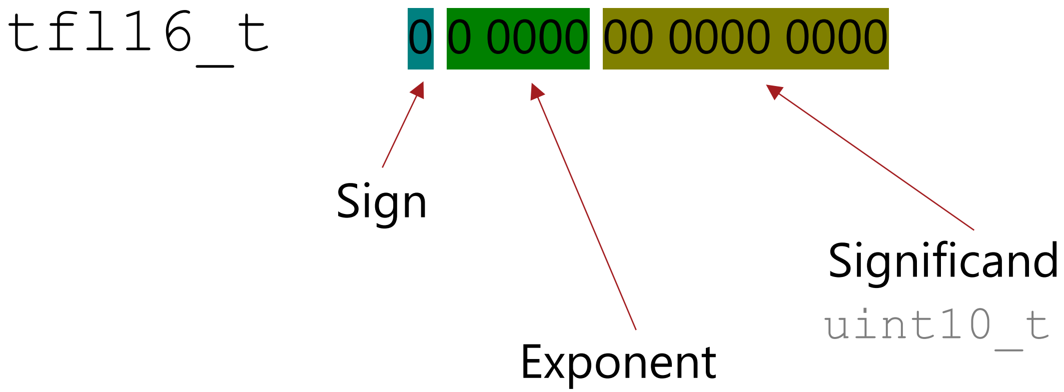
```
node --debug-brk=8962 --nolazy src/www.js
Debugger listening on port 8962
```

Image from https://code.visualstudio.com/docs/editor/debugging

Assignment #1

A floating-point library

aka soft-float



`tfl16_t` 0 0 0000 00 0000 0000

Sign

Exponent

Unbiased, `int5_t`
No special values
No denormalization support

Significand

`uint10_t`

Implement the methods

```
#include <stdlib.h>
typedef uint16_t tfl16_t;

tfl16_t  tfl_sign(tfl16_t value);
int8_t   tfl_exponent(tfl16_t value);
uint16_t tfl_significand(tfl16_t value);
uint8_t  tfl_equals(tfl16_t a, tfl16_t b);
uint8_t  tfl_greaterthan(tfl16_t a, tfl16_t b);
tfl16_t  tfl_normalize(uint8_t sign, int8_t exponent, uint16_t significand);
tfl16_t  tfl_add(tfl16_t a, tfl16_t b);
tfl16_t  tfl_mul(tfl16_t a, tfl16_t b);
```

Wrapping up