LECTURE 1 – BOOLEAN LOGIC AND INTEGERS

BOOLEAN LOGIC

Boolean values

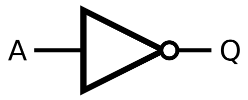
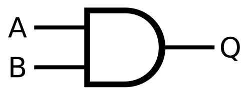
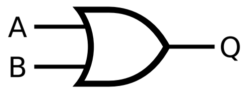
- False = 0
- True = 1

Boolean variables:

$$x \in \{0, 1\}$$

Boolean expressions

Boolean operators:

operator	math	pseudocode	C code	logic gate
negation	¬	not	!	
conjunction	∧, ×	and	&&, &	
disjunction	∨, +	or	||, |	

Example expression:

```
(a and b) or (not c)
```

Example function:

```
f(a, b, c) := (a and b) or c
```

NOT operator

Truth table:

x	**not x**
0	1
1	0

Example assignment:

```
w := not a
```

AND operator

Truth table:

x	y	x and y
0	0	0
0	1	0
1	0	0
1	1	1

Example assignment:

```
z := a and (not b)
```

OR operator

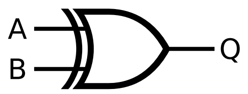
Truth table:

x	**y**	**x or y**
0	0	0
0	1	1
1	0	1
1	1	1

Example assignment:

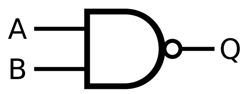
```
z := (not a) or (b and c)
```

More operators!



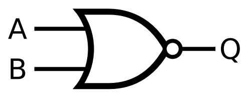
XOR

x	y	x xor y
0	0	0
0	1	1
1	0	1
1	1	0



NAND

x	y	x nand y
0	0	1
0	1	1
1	0	1
1	1	0



NOR

x	y	x nor y
0	0	1
0	1	0
1	0	0
1	1	0

Q: How many distinct **unary** Boolean operators?

A: one? (NOT)

Actually, we have 4 deterministic unary operators in total (counting 3 trivial unary operators):

always false

`x`	`0`
0	0
1	0

always true

`x`	`1`
0	1
1	1

identity

`x`	`x`
0	0
1	1

NOT

`x`	`not x`
0	1
1	0

Q: How many distinct binary operators?

A: As many as there are corresponding truth tables.

Q: How many distinct truth tables for two Boolean inputs and one Boolean output?

`x`	`y`	`op(x, y)`
0	0	?
0	1	?
1	0	?
1	1	?

Q: Why do we usually use NOT, AND, OR only?

A: Because

- they are the most intuitive
- all nontrivial operators can be represented with NOT, AND and OR

Examples:

```
x nand y = not (x and y)
x xor y = (x or y) and (not (x and y))
```

Note:

NAND and NOR are called *universal* logic gates:

every nontrivial operator can be represented with each *alone*

Q: How do we prove this?

```
x xor y = (x or y) and (not (x and y))
```

A:

x	**y**	**x xor y**	**(x or y) and (not (x and y))**
0	0	0	0
0	1	1	1
1	0	0	0
1	1	1	1

The identity is correct iff the truth tables match.

Boolean identities I

- `x and 0 = 0`
- `x or 1 = 1`
- `x and 1 = x`
- `x or 0 = x`
- `x or x = x`
- `x and x = x`

Boolean identities II

- AND is commutative:

```
x and y = y and x
```

- AND is associative:

```
x and (y and z) = (x and y) and z
```

- OR is commutative:

```
x or y = y or x
```

- OR is associative:

```
x or (y or z) = (x or y) or z
```

Boolean identities III

- Distributivity (AND over OR):

```
x and (y or z) = (x and y) or (x and z)
```

- Distributivity (OR over AND):

```
x or (y and z) = (x or y) and (x or z)
```

- De Morgan's law (1):

```
(not x) and (not y) = not (x or y)
```

- De Morgan's law (2):

```
(not x) or (not y) = not (x and y)
```

Satisfiability problem

Given a Boolean expression, find a value for each variable such that the expression is true.

Equivalently: Find a 1 in the truth table.

Example: `x1 and ((not x2) or x3) and (not x3)`

x1	x2	x3	x1 and ((not x2) or x3) and (not x3)
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Solution: `x1 = 1,  x2 = 0,  x3 = 0`

Definitions

- **Variable:** $x_j,$ for some $j \in J \subseteq \mathbb{N}$

  Ex.:

```
x1
x5
```

- **Literal:** either x_j or $\neg x_j,$ for some $j \in J$

  Ex.:

```
x3
(not x8)
```

- **Disjunctive clause:** $\bigvee_{j \in J^0} \neg x_j \vee \bigvee_{j \in J^1} x_j$ for some $J^0, J^1 \subseteq J$

  Ex.:

```
x2 or (not x4) or (not x6)
(not x1) or x5 or x6 or x7 or x9
```

- **Conjunctive clause:** $\bigwedge_{j \in J^0} \neg x_j \wedge \bigwedge_{j \in J^1} x_j$ for some $J^0, J^1 \subseteq J$

  Ex.:

```
x2 and (not x4) and (not x6)
(not x1) and x5 and x6 and x7 and x9
```

Conjunctive normal form

The conjunctive normal form (CNF) is a conjunction of disjunctive clauses:

$$\bigwedge_{i \in I} \left(\bigvee_{j \in J^{i,0}} \neg x_j \vee \bigvee_{j \in J^{i,1}} x_j \right), \qquad \text{where } J^{i,0}, J^{i,1} \subseteq J \subseteq \mathbb{N},\ \forall i \in I \subseteq \mathbb{N}$$

Examples:

```
((x1 or x2) and (x3 or x4) and (x5 or x6))

((x1 or (not x2)) and (x3 or (not x4)))

    (x2 or (not x4) or (not x6))
and ((not x1) or x5 or x6 or x7 or x9)
and ((not x1) or (not x2) or (not x3))
and (x4 or x5 or x6)
```

Disjunctive normal form

The disjunctive normal form (DNF) is a disjunction of conjunctive clauses:

$$\bigvee_{i \in I} \left(\bigwedge_{j \in J^{i,0}} \neg x_j \wedge \bigwedge_{j \in J^{i,1}} x_j \right), \qquad \text{where } J^{i,0}, J^{i,1} \subseteq J \subseteq \mathbb{N}, \ \forall i \in I \subseteq \mathbb{N}$$

Examples:

```
((x1 and x2) or (x3 and x4) or (x5 and x6))

((x1 and (not x2)) or (x3 and (not x4)))

    (x2 and (not x4) and (not x6))
or ((not x1) and x5 and x6 and x7 and x9)
or ((not x1) and (not x2) and (not x3))
or (x4 and x5 and x6)
```

Theorems

- Every Boolean expression can be put into CNF
  - For every Boolean expression with n variables and k literals using operators { NOT, AND, OR }, there exists an equivalent CNF with $n + k$ variables $3k$ clauses and $7k$ literals at most.
  - Satisfiability for a CNF ("SAT") is hard.
- Every Boolean expression can be put in DNF
  - For every Boolean expression with n variables and k literals using operators { NOT, AND, OR }, there exists an equivalent DNF with n variables and $n \times 2^n$ literals at most
  - Satisfiability for a DNF is trivial.

Example:

```
    (x2 and (not x4) and (not x6))
or ((not x1) and x5 and x6 and x7 and x9)
or ((not x1) and (not x2) and (not x3))
or (x4 and x5 and x6)
```

1. Take any clause, e.g. `(x2 and (not x4) and (not x6))`.
2. Set `x2 = 1,  x4 = 0,  x6 = 0`.
3. Done

INTEGER ARITHMETIC

- Computers are made out of Boolean gates
- But we want to represent numbers other than 0 and 1
- How do we proceed?

- Consider Booleans as binary **digits** (*bits*)
- Group them together to form numbers in base 2

Base-10 numbers

In base 10 (decimal), we have 10 digits: { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }

Using one digit, we can count to 9:

```
0    1    2    3    4    5    6    7    8    9
```

Then we need more digits:

```
10   11   12   13   14   15   16   17   18   19
20   21   22   23   ...
```

If we wanted to count from 0 to 9999 (say, to represent a date), we may decide to use 4 digits:

```
0000 0001 0002 0003 0004 0005 0006 0007 0008 0009
0010 0011 0012 0013 ...
```

Base-10 numbers

1984 = ?

$$\begin{array}{ccccccccc} & 1 & & 9 & & 8 & & 4 \\ \hline = & 1 \times 1000 & + & 9 \times 100 & + & 8 \times 10 & + & 4 \\ \hline = & 1 \times 10^3 & + & 9 \times 10^2 & + & 8 \times 10^1 & + & 4 \times 10^0 \end{array}$$

Base-2 numbers

In base 2 (binary), we have 2 digits: { 0, 1 }

Using one digit, we can count to 1:

```
0    1
```

Then we need more digits:

```
10   11  100  101  110  111 1000 1001 ...
```

If we wanted to count from 0 to 15, we may decide to use 4 digits:

```
0000 0001 0010 0011 0100 0101 0110 0111
1000 1001 1010 1011 1100 1101 1110 1111
```

Base-2 numbers

1001b = ?

$$
\begin{array}{c|c|c|c|c|c|c|c}
 & 1 & & 0 & & 0 & & 1 \\
\hline
= & 1 \times 8 & + & 0 \times 4 & + & 0 \times 2 & + & 1 \\
\hline
= & 1 \times 2^3 & + & 0 \times 2^2 & + & 0 \times 2^1 & + & 1 \times 2^0
\end{array}
$$

$$= 9$$

Note:

- rightmost / least-significant bit is called bit 0
- leftmost / most-significant bit is called bit $n - 1$

Fixed bit width

- For any integer, we must always know how many digits (bits) it has.
- Typically, this number of bits is fixed in our code.

bits	a.k.a.	C type	other C type
8	byte†	`uint8_t`	`unsigned char`†
32		`uint32_t`	`unsigned int` (Windows, Linux, BSD, macOS)
64		`uint64_t`	`unsigned long` (Linux, BSD, macOS)
			`unsigned long long` (Windows)

† = on almost all contemporary platforms as of 2023

Integers in hardware and in programming languages

- Most computers† support 8, 16, 32 and 64-bit arithmetic natively (i.e., operations are fast)
- Arithmetic can be performed with arbitrary-sized integers by implementing the operations in software (hence much slower).
- In C, every integer type has a specific size.
- In C, arbitrary-sized integers are not supported by the language (they require using specific libraries).
- In Python, all integers can have arbitrary sizes (with a large performance penalty, especially when exceeding 32 bits)

bits	largest integer $= 2^{\text{bits}} - 1$	(approx.)
8	255	
16	65,535	
32	4,294,967,295	4 billions
64	18,446,744,073,709,551,615	2.10^{19}
128	340,282,366,920,938,463,463,374,607,431,768,211,455	3.10^{38}

$$1 \text{ decimal digit} = \log_2 10 \text{ bits} \simeq 3.3219 \text{ bits}$$

Operations with integers

Essentially the same a schoolbook operations:

	0	1	0	1	0	0	1	1
+	0	1	1	0	0	0	0	1
=	1	0	1	1	0	1	0	0

Just like in school:

- addition and subtraction are straightforward
- multiplication is more complex
- division is much more complex

Signed integers

- How do we represent negative numbers?
- Impossible with previous approach.
- Solution 1:
  - “sign-magnitude”: sacrifice one bit, which we reserve to store the sign.
  - Drawback: zero has two representations (+0 and -0)
  - Drawback: Boolean logic for + and - must handle many cases
- Solution 2:
  - “one’s complement”: reserve top bit for the sign, must be zero for a positive number
  - when a number is negative, takes its (positive) opposite and flip all bits
  - Drawback: zero has two representations (+0 and -0)
  - Drawback: Boolean logic for + and - is simpler but still affected

Signed integers: two’s complement

- Solution 3 (all current computers†):
  - “two’s complement”: when a n-bit number x is negative, represent it the same as the unsigned number $2^n - x$.
  - The top bit is 1 for negative numbers.
  - Drawback: Flipping sign slightly more complex (flip all non-sign bits then add one).
  - Advantage: zero has a single representation
  - Advantage: Boolean logic for + and - is **the same** as for unsigned integers

4-bit signed integers (two's complement)

b3	b2	b1	b0	unsigned	signed
0	0	0	0	0	0
0	0	0	1	1	1
0	0	1	0	2	2
0	0	1	1	3	3
0	1	0	0	4	4
0	1	0	1	5	5
0	1	1	0	6	6
0	1	1	1	7	7
1	0	0	0	8	-8
1	0	0	1	9	-7
1	0	1	0	10	-6
1	0	1	1	11	-5
1	1	0	0	12	-4
1	1	0	1	13	-3
1	1	1	0	14	-2
1	1	1	1	15	-1

Example:

signedness	decimal	binary
unsigned	`2 + 11 = 13`	`0010b + 1011b = 1101b`
signed	`2 + -5 = -3`	`0010b + 1011b = 1101b`

bits	$-2^{\text{bits}-1}$ (min)	$2^{\text{bits}-1} - 1$ (max)
8	-128	127
16	-32768	32767
32	-2,147,483,648	2,147,483,647
64	$\simeq -9.10^{18}$	$\simeq 9.10^{18}$
128	$\simeq -2.10^{38}$	$\simeq 2.10^{38}$

Q: What happens if we run this?

```
unsigned char a = 255;
unsigned char b = 1;
unsigned char x = a + b;
```

```
signed char a = 127;
signed char b = 1;
signed char x = a + b;
```

```
unsigned char a = 1;
unsigned char b = 2;
unsigned char x = a - b;
```

```
signed char a = -128;
signed char b = 1;
signed char x = a - b;
```

A: It’s complicated!

We will dedicate an entire chapter to this.