QUANTUM COMPUTING AND QUANTUM INFORMATION

Quantum computing is a generalisation of ordinary (classical) computing exploiting the specific character of quantum mechanical laws. In particular, **bits** are replaced by the so-called **qubits**. These are the states of spin-$\frac{1}{2}$ particles.

Quantum information theory is a closely related subject that deals with the question how information can be represented by qubits. This idea has given rise to new ways of encoding and transmitting information that holds great promise for the future.

This course starts with an introduction to the basics of classical computing. We'll alternate between classical and quantum in order not to spend too much time on classical computing at the start. There will also be a brief reminder of the basics of quantum mechanics, concentrating on the most relevant aspects, which are more conceptual than calculational.

Recommended books:

R. P. Feynman: Lectures on Computation.
Penguin, 1996.

J. Gruska: Quantum Computing.
McGraw-Hill, 1999.

M. A. Nielsen & I. L. Chuang: Quantum Computation and Quantum Information.
Cambridge Univ. Press, 2000.

COMPUTER LANGUAGES

There are many different programming languages, some of which you will probably be familiar with. **Higher-level languages** are for example: BASIC, ALGOL, FORTRAN, PASCAL, C++, ADA, PROLOG, COBOL, etc.

Assembly language is more closely related to the inner workings of the computer. Machine language is the binary language that is in immediate one-to-one correspondence with the CPU operation.

Language	Advantage	Disadvantage
Higher-level Language	Portable Intelligible Versatile Error checking	Abstract Removed from core computer Compiled
Assembly	Concrete Close to CPU op. Fast code	Difficult to program OS specific Error prone
Machine Language	1-1 with CPU No assembly Fast	All in binary Microprocessor specific Inscrutable Laborious to enter.

DOS Assembly Language Commands

ADD *destination, source*

Adds the source to the destination.

Flags affected: AF, CF, OF, PF, SF, ZF.

AND *destination, source*

Bitwise AND of destination and source, result in destination. CF and OF are set to zero.

Flags affected: CF, OF, PF, SF, ZF.

CALL *address* Call a subroutine.

CBW (no operands).

Converts a number in AL to the same number in AX by sign extension.

CLC (no operands). Clears the Carry Flag, CF.

CMC (no operands). Complements the Carry Flag, CF.

CMP *destination, source*

Performs a subtraction of the two operands causing the flags to be affected but without affecting the operands. Source and destination must be of the same type.

Flags affected: AF, CF, OF, PF, SF, ZF.

DEC *destination* Subtracts 1 from the operand.

DIV *source*

Performs an unsigned division of a number with low word in AX and high word in DX by the number in the source operand. It returns a single-length quotient to AX and the remainder to DX. If the quotient is too large, an interrupt type zero is generated.

INC *destination* Adds 1 to the operand.

INT *number*

Interrupt routine. In particular, INT 20 is the program terminate routine, and INT 21 is the DOS interrupt. The particular DOS routine executed depends on the value of AH. Examples are:

AH=1: Keyboard Input Function. Keyboard character is returned in AL.

AH=2: Display Output Function. Put Character to be printed on screen in DL.

AH=5: Printer Output Function. Put character to be printed into DL.

AH=8: No-Echo Input. Same as AH=1, but no output on screen.

AH=9: Print String Function. Address of string into DX. String must end with $.

AH=A: Buffered Keyboard Input Function. Characters typed are stored in a buffer starting at the address in DX.

AH=B: Input Status Function. Check if character has been typed. If yes, AL=FF, else AL=0.

JZ/JE *address*

Jump if zero/equal. Transfers control to the (local) address of operand if ZF=1.

JG *address*

Jump if greater than: transfers control if ZF=0 and SF=OF.

JGE *address* Jump if greater or equal.

JL *address* Jump if less than.

JLE *address* Jump if less than or equal.

JMP *address* Jump to the given address.

JNE *address* Jump if not equal.

LOOP *address* Decrements CX by 1 and transfers control to the operand address until CX=0.

MOV *destination, source*
Copies the source into the destination.

MUL *source*
Performs an unsigned multiplication of AX (or AL) and the source operand. The result is in AX (low word) and DX (high word), resp. AX.
Flags affected: CF, OF.

NEG *destination* Negate, i.e. subtract from zero.
Flags affected: AF, CF, OF, PF, SF, ZF.

NOP No operation.

NOT *destination* Bitwise logical NOT.

OR *destination, source*
Bitwise OR, result in destination. Flags affected: CF, OF, PF, SF, ZF.

POP *destination* Pops destination off the stack.

POPF Pops flags off the stack.

PUSH *source* Push source on the stack.

PUSHF Push flags on the stack.

RET Return from procedure.

ROL *destination, count* Rotate destination by *count* bits to the left. Count must be CL or 1.

ROR *destination, count* Rotate destination by *count* bits to the right. Count must be CL or 1.

SHR *destination, count* Shift destination by *count* bits, replacing the high-order bits by zeros; *count must be 1 or CL.*
Flags affected: CF, OF, PF, SF, ZF.

STC Set Carry Flag.

SUB *destination, source* Subtract source from destination.

XCHG *destination, source* Exchanges the contents of the destination and source registers.

XOR *destination, source* Bitwise exclusive OR.

Example Assembly Programs

Printing a sentence:

```
100    mov    dx,109
103    mov    ah,9
105    int    21
107    int    20
109    db     'Quantum Computing...'
11D    db     0D, 0A, '...A pipe dream ?$'
```

Alternative method:

```
100    mov    cx,23
103    mov    bx,115
106    mov    dl,[bx]
108    mov    ah,2
10A    int    21
10C    inc    bx
10D    loop   106
10F    mov    ah,8
111    int    21
113    int    20
115    db     'Quantum Computing...'
129    db     'A pipe dream ?$'
```

BINIHEX

```
100    mov    ch,4
102    mov    cl,4
104    rol    bx,cl
106    mov    al,bl
108    and    al,0F
10A    add    al,30
10C    cmp    al,3A
10E    jl     112
110    add    al,7
112    mov    dl,al
114    mov    ah,2
116    int    21
118    dec    ch
11A    jnz    102
11C    int    20
```

Exercise 1: Use this as a procedure in a program to convert a decimal number to hexadecimal. You need the *multiply instruction* MUL CX, and the Keyboard Input Routine INT 21 with AH=1 and resulting character in AL.

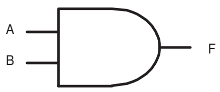
Exercise 2: Write a program to print the first 128 ASCII characters (on the screen). Then analyse the more sophisticated ASCII2 program (handed out).

Boolean Logic

Basic logic elements:

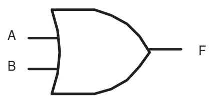
Logic Truth Tables

AND Gate



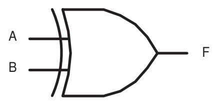
INPUT		OUTPUT
A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

OR Gate



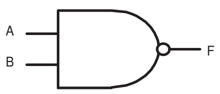
INPUT		OUTPUT
A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

XOR Gate



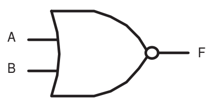
INPUT		OUTPUT
A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

NAND Gate



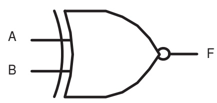
INPUT		OUTPUT
A	B	F
0	0	1
0	1	1
1	0	1
1	1	0

NOR Gate



INPUT		OUTPUT
A	B	F
0	0	1
0	1	0
1	0	0
1	1	0

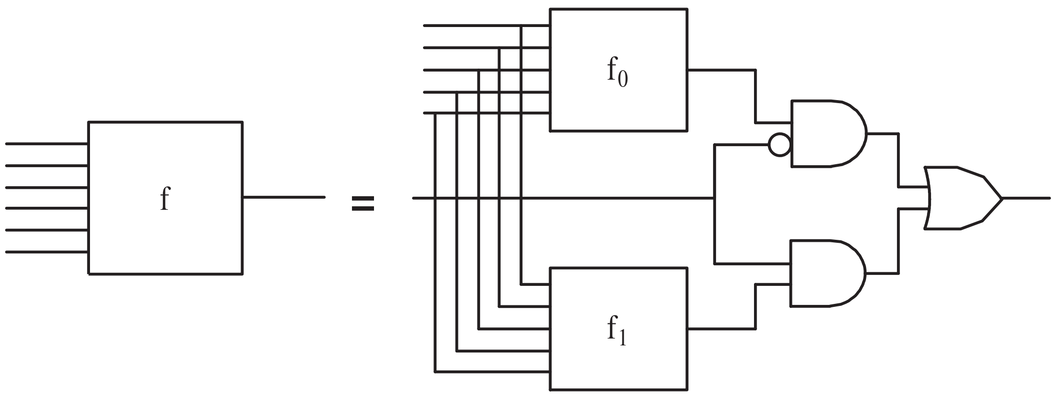
XNOR Gate



INPUT		OUTPUT
A	B	F
0	0	1
0	1	0
1	0	0
1	1	1

Theorem. *Every logical operation $f : \{0,1\}^n \to \{0,1\}^m$ can be expressed in terms of AND, OR and NOT.*

The proof is by induction, and uses the following diagram in the induction step:



Exercise: Construct a circuit diagram for the logical function $f : \{0,1\}^3 \to \{0,1\}$ given by $f(x_1, x_2, x_3) = (x_1 + x_2 - 2x_1x_2)x_3 + x_1x_2$. (In particular, show that f is well-defined.)

Definition. *A* **ring** *is a set $\mathcal{R}$ with operations $\oplus$ and $\cdot$ such that*

1. $\mathcal{R}$ with $\oplus$ is an abelian group;

2. $\cdot$ is associative; and

3. (Distributive law) For all $a, b, c \in \mathcal{R}$,
$a \cdot (b \oplus c) = a \cdot b \oplus a \cdot c$ *and*
$(a \oplus b) \cdot c = a \cdot c \oplus b \cdot c$.

Definition. *A* **Boolean ring** *is a ring $\mathcal{R}$ with unit $\mathbf{1}$ such that for all $a \in \mathcal{R}$, $a \cdot a = a$.*

Theorem. *A Boolean ring $\mathcal{R}$ is commutative, and satisfies: $a \oplus a = 0$ for all $a \in \mathcal{R}$.*

Given a Boolean ring, one can define logic operations through:

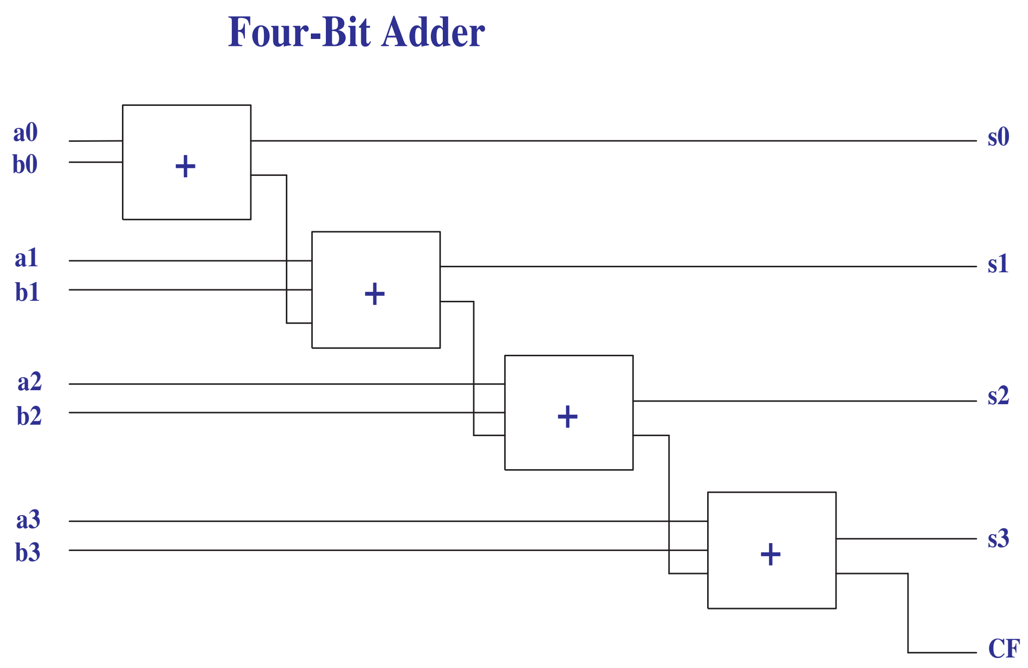
$$\neg a = \bar{a} = \mathbf{1} \oplus a;$$
$$a \vee b = a + b = a \oplus b \oplus a \cdot b;$$
$$a \wedge b = a \cdot b.$$

Exercise 3: Given a Boolean ring $\mathcal{R}$, prove De Morgan's laws: $\overline{a+b} = \bar{a}\cdot\bar{b}$ and $\overline{a\cdot b} = \bar{a}+\bar{b}$, as well as the relations: $a(b+c) = ab+ac$, $a+a=a$, $a+\bar{a} = \mathbf{1}$ and $a\cdot\bar{a} = 0$.

Exercise 4: Construct a circuit diagram for the **full binary adder** given by the truth table:

A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Here is a diagram for a 4-bit adder, using the full binary adder:

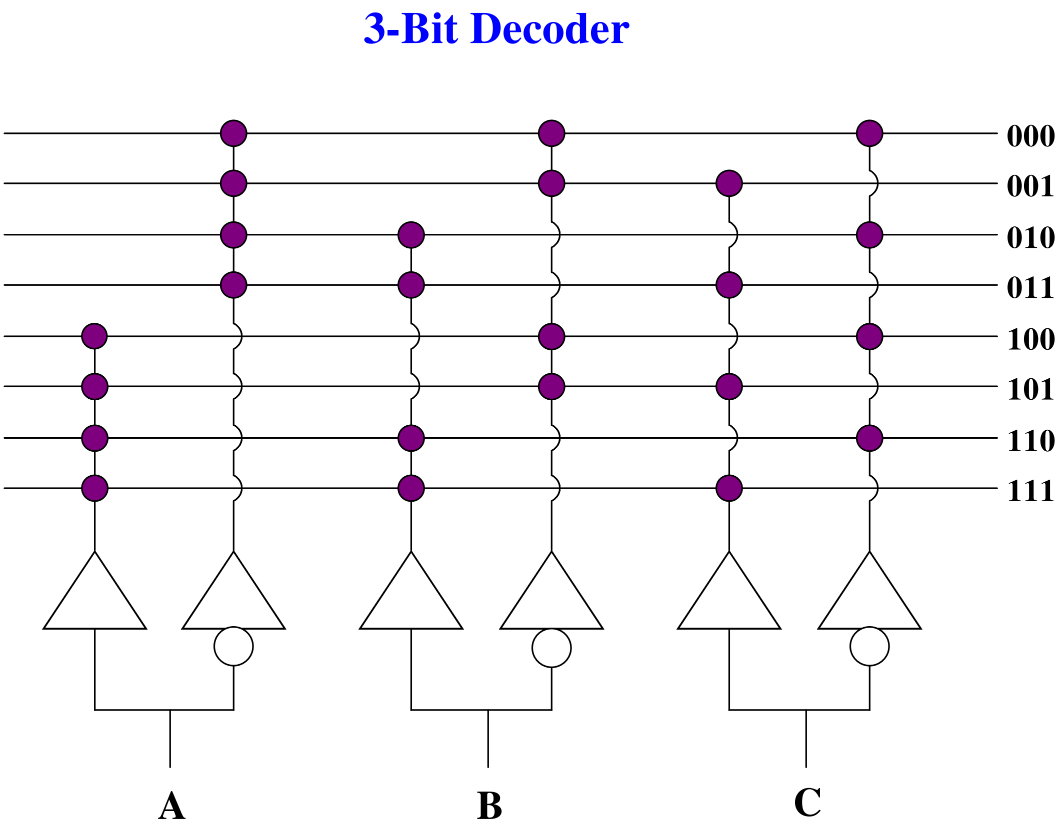


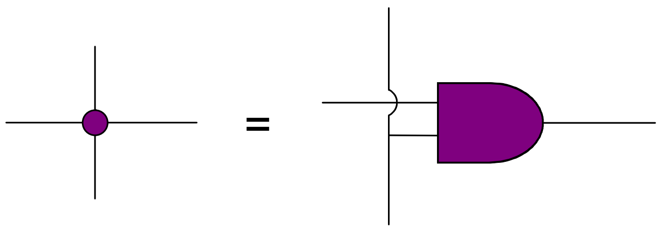
Exercise 5: Design a 4-bit **comparator** with 2×4 input lines and 3 outputs: one giving 1 if the two 4-bit inputs are equal and 0 otherwise; one giving 1 if the first is greater than the last and 0 otherwise; and one giving 1 if the last is greater than the first and 0 otherwise.

Theorem. *Every logical expression can be written in SOP form, i.e. in the form* $f_1 + \ldots + f_n$*, where each* f_i *is an AND-clause in (some of) the original variables or their negations. Similarly, it can also be written in POS form.*

This theorem is the basis for the design of **Programmable Logic Devices** which consist of a grid of wires with fusible links at each cross point. The horizontal wires are connected to a common OR-gate.

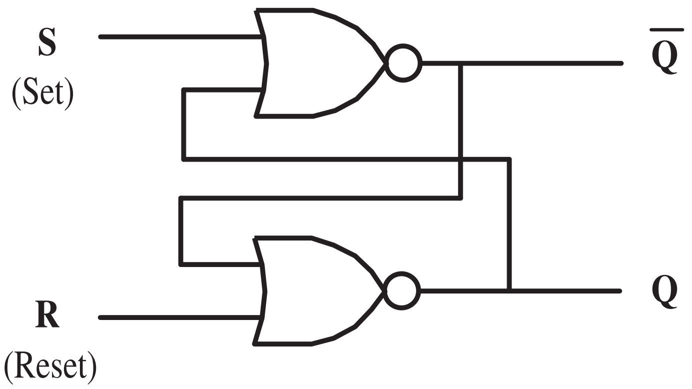
An important circuit is also the **binary decoder**:



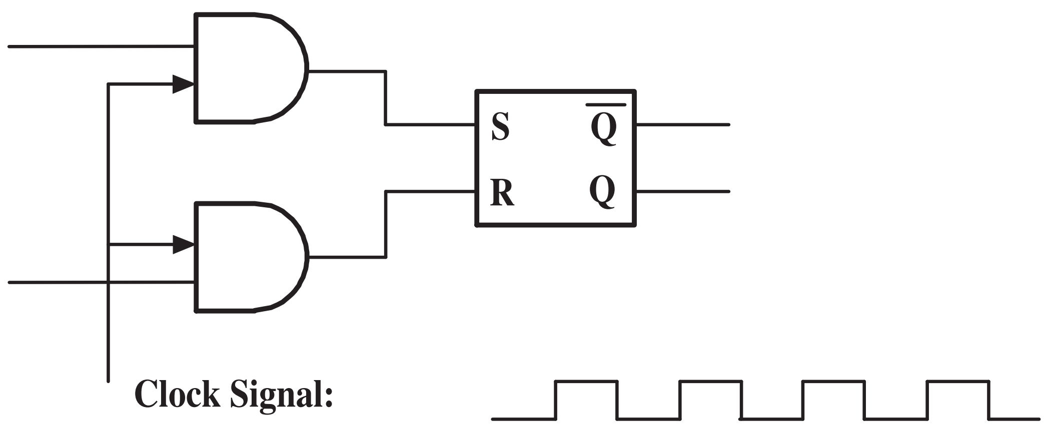


This device can serve as a switch for selecting a particular memory or port address, or, in combination with an OR gate, as a **multiplexer**.

Flip-Flop



Clock operation:



As shown, flip-flop circuits need a clock signal to operate properly. A combination of flip-flops can act as a *shift register*.