Digital computer

- **Two voltage levels of interest**
  - **High level**
    - Logical one (signal *high/true/asserted*)
  - **Low level**
    - Logical zero (signal *low/false/deasserted*)
  - Logical values are complementary and inverse of each other
    - Unlike the voltage levels representing logical ones and zeros
Logic blocks

- **Combinational**
  - No memory $\rightarrow$ no internal state
  - Output depends only on current input
  - Represents logical functions

- **Sequential**
  - Has memory $\rightarrow$ has internal state
  - Output depends on input and internal state
  - Captures sequence of steps
Logic functions and truth tables

- **Logic function (also Boolean function)**
  - Output value is a function of input values
  - $f : \mathcal{B}^k \to \mathcal{B}$, where $\mathcal{B} = \{ 0, 1 \}$ and $k \in \mathbb{N}$ is arity

- **Truth table**
  - Function defined by enumerating the output for each combination of inputs (a table $2^k$ rows for $k$ inputs)

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Output $f(a, b)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$b$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Boolean algebra

- Logic functions expressed as equations
  - Variables hold values from $B = \{0, 1\}$
  - Basic operators – primitive logic functions
    - **Logical inversion** (NOT): $\bar{x}$, $\neg x$, $!x$
    - **Logical product, conjunction** (AND): $x \cdot y$, $x \land y$, $x \&\& y$
    - **Logical sum, disjunction** (OR): $x + y$, $x \lor y$, $x |\| y$
  - Additional operators (16 for 2 variables)
    - NAND, NOR, XOR etc.
## Logic operators

<table>
<thead>
<tr>
<th>Inputs ( a ) ( b )</th>
<th>Basic operators</th>
<th>Universal operators</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NOT ( a )</td>
<td>( a ) AND ( b )</td>
</tr>
<tr>
<td>( 0 ) ( 0 )</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( 0 ) ( 1 )</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( 1 ) ( 0 )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( 1 ) ( 1 )</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inputs ( a ) ( b )</th>
<th>Other operators</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( a ) XOR ( b )</td>
<td>( a ) XNOR ( b )</td>
</tr>
<tr>
<td>( 0 ) ( 0 )</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( 0 ) ( 1 )</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( 1 ) ( 0 )</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( 1 ) ( 1 )</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Boolean algebra laws

- **Idempotency:** $a + a = a$, $a \cdot a = a$
- **Computativity:** $a + b = b + a$, $a \cdot b = b \cdot a$
- **Associativity:** $a + (b + c) = (a + b) + c$, $a \cdot (b \cdot c) = (a \cdot b) \cdot c$
- **Absorption:** $a \cdot (a + b) = a$, $a + (a \cdot b) = a$
- **Distributivity:** $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$, $a + (b \cdot c) = (a + b) \cdot (a + c)$
- **Neutrality of 0 and 1:** $a + 0 = a$, $a \cdot 1 = a$
- **Aggressivity of 0 and 1:** $a + 1 = 1$, $a \cdot 0 = 0$
- **Complementarity:** $a + \neg a = 1$, $a \cdot \neg a = 0$
- **Absorption of negation:** $a \cdot (\neg a + b) = a \cdot b$, $a + (\neg a \cdot b) = a + b$
- **De Morgan’s laws:** $\neg(a + b) = \neg a \cdot \neg b$, $\neg(a \cdot b) = \neg a + \neg b$
- **Double negation:** $\neg(\neg a) = a$
Logic functions extended to operate on (finite) sequences of bits

- **Word** = finite sequence of bits
- **Word length** = number of bits in the sequence
- Output of a logic operation \( f \) is function of input values
  - \( f: (\mathbb{B}^n)^k \rightarrow \mathbb{B}^n \), where \( \mathbb{B} = \{0, 1\} \), \( k \in \mathbb{N} \) is arity and \( n \in \mathbb{N} \) is word length
Intermezzo: CPU logical operations (2)

- (Bitwise) logical product/sum/inversion
  - Operators &, |, ~ etc. in C-like languages
  - Primitive logic function applied to individual bits of the input words, result stored to individual bits of the output word
  - Allow isolating (AND), zeroing (AND, NOR), setting (OR), inverting (XOR) selected bits, or inverting all bits (NOT), of the input word
Intermezzo: CPU logical operations (3)

- **Logical shifts (left and right)**
  - Operators `<<` and `>>` in C-like languages
  - Shifts bits in a word `i` positions to the left or right
    - “Vacated” bits are replaced with 0
  - For binary natural numbers
    - Shift by `i` bits to the left $\rightarrow$ multiplying by $2^i$
    - Shift by `i` bits to the right $\rightarrow$ dividing by $2^i$


**Logic gates (1)**

- **Physical implementation basic logic functions**
  - Basic gates: NOT, OR, AND

```
Logic gates

- NOT: \( \neg a \)
- OR: \( a + b \)
- AND: \( a \cdot b \)
```

![Logic gate diagrams](image-url)
Logic gates (2)

Physical implementation of logic operators

- Inverting gates: NAND, NOR
- Less common gates: XOR

\[
\begin{align*}
\text{NAND: } & \quad \overline{a \cdot b} \\
\text{NOR: } & \quad \overline{a + b} \\
\text{XOR: } & \quad a \oplus b
\end{align*}
\]
Combinational logic circuits

- Implementation of more complex logic functions
  - Combines multiple logic operators
    - Logic signals correspond to variables
    - Logic gates correspond to primitive operators
  - Most commonly NAND or NOR gates
    - Sufficient for expressing any logic function

- Logic block
  - Abstracts away from internal structure of a circuit
  - Provides functional building blocks
Logic blocks: binary (half) adder

- Adds two 1-bit numbers
  - The simplest case
  - **Input:**
    - operand \( a \)
    - operand \( b \)
  - **Output:**
    - sum \( s \)
    - carry \( c \)
  - **Function:**
    - \( s = a \cdot \neg b + \neg a \cdot b = a \text{ XOR } b \)
    - \( c = a \cdot b = a \text{ AND } b \)
Logic blocks: binary (half) adder (2)
Logic blocks: binary adder (3)

- Adding $n$-bit numbers
  - Merge $n$ $\frac{1}{2}$-adders for individual bits?

- $n$-bit $\sum$

- $\frac{1}{2}$-adder cannot propagate carry from previous additions (not enough inputs)
Logic blocks: binary adder (4)

- **Full adder**
  - Adds two 1-bit numbers taking into account carry from previous addition

- **Input:**
  - operand $a$
  - operand $b$
  - carry $c_0$

- **Output:**
  - sum $s$
  - carry $c$
logic blocks: binary adder (5)

full adder

- Adds two 1-bit numbers taking into account carry from previous addition
- inputs: operand $a$, operand $b$, carry $c_0$
- outputs: sum $s$, carry $c$

- $s = \overline{c_0} \cdot (a \cdot \overline{b} + \overline{a} \cdot b) + c_0 \cdot (a \cdot b + \overline{a} \cdot \overline{b})$
  
  $s = \ldots$
  
  $s = c_0 \text{ XOR } (a \text{ XOR } b)$

- $c = a \cdot b + c_0 \cdot (a \cdot \overline{b} + \overline{a} \cdot b)$
  
  $c = (a \text{ AND } b) \text{ OR } (c_0 \text{ AND } (a \text{ XOR } b))$

<table>
<thead>
<tr>
<th>$c_0$</th>
<th>$a$</th>
<th>$b$</th>
<th>$c_0$</th>
<th>$s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Logic blocks: binary adder (6)
Logic blocks: binary adder (7)

Abstraction:
\[ a + b + c_0 = (a + b) + c_0 \]
Logic blocks: binary adder (8)

n-bit $\sum$

Function block:

\[
\begin{align*}
&\text{add} \\
&\downarrow c \\
&\begin{array}{c}
& a \\
& \downarrow \ \\
& b \\
& \downarrow \ \\
& s
\end{array}
\end{align*}
\]
Logic block for subtraction

- Taking advantage of 2’s complement
  - Basic building block: adder
  - Use XOR gate as a controlled inverter
  - **Example:** 2-bit ALU supporting addition and subtraction
    - **Data input:** operand bits \(a_1a_0\), operand bits \(b_1b_0\)
    - **Control input:** signal SUB to determine operation
      - \(SUB = 0 \rightarrow\) addition
      - \(SUB = 1 \rightarrow\) subtraction
    - **Output:** sum/difference bits \(s_1s_0\), carry \(c\)
2-bit ALU for adding/subtracting

$SUB = 1$ inverts bits of the second operand and adds 1 (negates number)
Sequential logic

- **Combinational logic + memory elements**
  - Memory elements keep internal state
  - Inputs and the contents of memory (internal state) determines outputs and next internal state

- **Synchronous vs. asynchronous sequential circuits**
  - Determines how and when state changes
  - Need to ensure stable inputs (inputs don’t change)
Clock signal to synchronize state changes

- Change state during one clock cycle
  - Inputs of combinational logic does not change while it is being read
  - Writing of values from outputs to memory elements happens with rising/falling edge of the clock signal
Memory elements

- Pair of inverters in a feedback loop
  - Asynchronous circuit with two stable states
    - Allows “storing” 1 bit of information
    - Need to be able to control the state...
      - We need a gate that can pass the signal unchanged, but allows forcing an output value when required
  - Basic building block for memory elements

![Diagram of memory elements with a pair of inverters in a feedback loop.](image-url)
Set-Reset (R-S) latch, NOR-based

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$q_n$</td>
</tr>
<tr>
<td>$r$</td>
<td>$s$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Diagram of the Set-Reset (R-S) latch with NOR-based logic.
Set-Reset (R-S) latch, NAND-based

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\bar{r})</td>
<td>(\bar{s})</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Other flip-flops

- **Derived from R-S**
  - *Clocked R-S latch*
    - Synchronous R-S latch variant
    - Reacts to \( R \) or \( S \) signals while the clock signal is high
  - *R-S master/slave (R-S flip-flop)*
    - Two clocked R-S latches (in series) with complementary clock signal
    - Reacts to \( R \) or \( S \) signals only on rising/falling edge of the clock signal

\[
\begin{array}{c}
\text{S} \\
\downarrow \\
\text{R} \\
\end{array} \quad \begin{array}{c}
\text{Q} \\
\downarrow \\
\neg Q \\
\end{array}
\]
Other flip-flops (2)

- Derived from R-S
  - J-K master/slave (J-K flip-flop)
    - Extends R-S (J = S, K = R), inverts state when J = K = 1
  - Clocked D latch, D flip-flop
    - Value determines by single input
  - T flip-flop
    - Allows dividing clock signal frequency
Data register made of flip-flops

\[ DQ \quad q_{n-1} \quad q_1 \quad q_0 \]

\[ d_{n-1} \quad d_1 \quad d_0 \]

\[ CLOCK \]

\[ DATA \]
Shift register made of flip-flops

![Shift register diagram](image)

- **DATA**
- **CLOCK**

- **D**
- **Q**

- $q_{n-1}$
- $\ldots$
- $q_1$
- $q_0$
32-bit sequential multiplier

32-bit ∑

A

32-bit ∑

Shift Arith. Right
Write

64-bit A×B

Control
Test
32-bit sequential divider

- 32-bit ALU
- 64-bit
- Shift Right
- Shift Left
- Write
- A mod B
- A div B
- Control
- Test