Processors Field Guide

Lucas Cabooter
Radboud University
Computer Science

March 20, 2023

Contents

Preface

This document will contain the summary for the course *Processors* (NWI-IPC006). This document will contain a *detailed* summary of everything we have to know for the final exam. The summary is based on the slides given in the lecture, and the book "Structured Computer Organisation", sixth edition, Pearson, 2013, or fifth edition, Pearson, 2006 by Andrew S. Tanenbaum.

One note is that this summary will be more detailed than what is presented in the slides and during the lectures. I therefore present extra examples and exercises to practice with whenever it is most applicable.
These exercises and their corresponding answers can be found at the end of each's week summary. Any extra resources for practice and information will also be there at the end of each week. It could be possible that there are some mistakes, but I will always try and verify every practice question to make sure anyone reading this won't make the same mistake.

As it is with many courses in Computer Science, practice makes perfect. Many times you will have to see a certain pattern, or understand what the underlying thought is behind a question. Therefore I will present as many different situations of questions they can ask at the exam, since I myself don't know what they are going to ask either.

Week 1: Boolean Algebra

Boolean Values and Operators

In the lecture the following Boolean values and operators get mentioned:

Boolean		Logic	
$\bar{x}$	complement (NOT)	$\neg p$	negation
$x \cdot y$	product (AND)	$p \wedge q$	conjunction
$x + y$	sum (OR)	$p \vee q$	disjunction
$\overline{xy}$	NAND		
$x \bar{+} y$	NOR		
$x \bar{y}$	XOR		

x	$\bar{x}$
0	1
1	0

x	y	xy
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$x+y$
0	0	0
0	1	1
1	0	1
1	1	1

x	y	$\overline{xy}$	$x+y$	$x \oplus y$
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	0	0	0

These operators will be discussed more thoroughly later on in this week's summary and I will provide some examples as well (that could be asked on the exam).

Boolean Functions and Expressions

Boolean expressions $f : B^n \to B$ represent Boolean functions. There exist 2^n (i.e., 2^{2^n}) distinct functions of degree n.

Deriving a Boolean expression for a function:

- *Literal:* x_i or $\overline{x_i}$
- *Minterm:* product of literals where each variable occurs exactly once
- *Expression for* f: sum of the minterms where $f = 1$ (full) Disjunctive Normal Form (DNF), or the '*sum of products*' representation
- *Equivalent*: two Boolean expressions that represent the same Boolean function are called equivalent.

Example:

x	y	$f(x,y)$	
0	0	1	$\bar{x}\,\bar{y}$
0	1	0	
1	0	1	$x\,\bar{y}$
1	1	1	$x\,y$

This gives us $f(x,y) = \bar{x}\,\bar{y} + x\bar{y} + xy$

You only look at the minterms where the function is a '1' (so 00, 10 and 11 in the example above).
If the variable/atomic proposition (x, y, z, etc.) is equivalent to '0', it will be written down as a complement in the Boolean expression.
If the variable/atomic proposition is equivalent to '1', it will be written down as a normal Boolean expression.
The complete function, so $f(x, y) = \bar{x} \cdot \bar{y} + x\bar{y} + xy$ is in this form, in (full) Disjunctive Normal Form (DNF).

Two Boolean expressions that represent the same Boolean function are called equivalent.
To show that two functions are actually equivalent you can use the laws of Boolean logic:

Name	Multiplicative form	Additive form
Complement	$\overline{\overline{x}} = x$	
Identity	$x \cdot 1 = x$	$x + 0 = x$
Null	$x \cdot 0 = 0$	$x + 1 = 1$
Idempotent	$x\,x = x$	$x + x = x$
Inverse	$x\,\overline{x} = 0$	$x + \overline{x} = 1$
Commutative	$xy = yx$	$x + y = y + x$
Associative	$x(yz) = (xy)z$	$x + (y + z) = (x + y) + z$
Distributive	$x(y + z) = xy + xz$	$x + yz = (x + y)(x + z)$
Absorption	$x(x + y) = x$	$x + xy = x$
De Morgan	$\overline{xy} = \overline{x} + \overline{y}$	$\overline{x + y} = \overline{x}\,\overline{y}$

There are more laws than these, but these are the most important/used ones.
They should be provided on the exam and an example of using these laws looks something like this:

$$
\begin{aligned}
(x + z)(\overline{x + \overline{y}}) &= (x + z)(\overline{x} \cdot \overline{\overline{y}}) && \text{(de Morgan)} \\
&= x\overline{x}y + z\overline{x}y && \text{(distributivity, double complement)} \\
&= 0 + z\overline{x}y && \text{(inverse, null)} \\
&= \overline{x}yz && \text{(identity, commutativity)}
\end{aligned}
$$

Name	Multiplicative form	Additive form
Null	$x \cdot 0 = 0$	$x + 1 = 1$
Inverse	$x\,\overline{x} = 0$	$x + \overline{x} = 1$
Distributive	$x(y + z) = xy + xz$	$x + yz = (x + y)(x + z)$
De Morgan	$\overline{xy} = \overline{x} + \overline{y}$	$\overline{x + y} = \overline{x}\,\overline{y}$

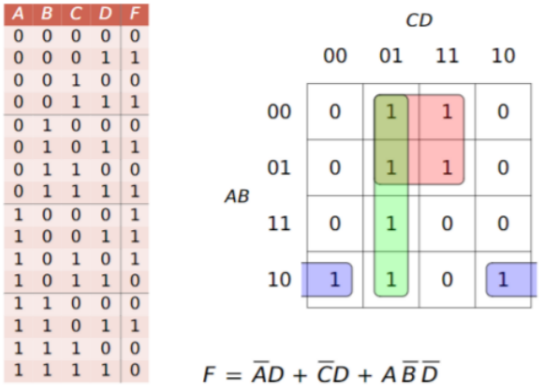
More examples using these rules will be at the end of this week's summary. In the 2022 exam, they for instance asked to find the simplest Boolean expression which was equivalent to an expression containing the XOR operator.

Karnaugh Maps

Karnaugh Maps is a graphical method for simplifying a Boolean expression.
The basic idea is that two minterms that differ in one literal can be combined into a simpler term.

Rules of Karnaugh Maps:

- Blocks are allowed to overlap.
- Blocks can be rectangular, have lengths of 1, 2, or 4, and be as large as possible.
- Use as few blocks as possible.
- Wrap around: top-bottom, left-right, corners.
- The blocks consist of only 1's, no 0's!



A	B	C	D	F
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

The guide to find the Boolean expression using a Karnaugh Map:

1. The first thing you do is write the 0's and 1's on top of the map and on the left side. They **must** be in the order 00, 01, 11, 10, otherwise, the map will give other results which won't give us an equivalent Boolean function (known as the grey code).
2. The next step is to fill in all the 0's and 1's which are inside op the map. You do this by looking at the values inside of the truth table. For instance the top left square in the map is 0, because $A = 0, B = 0, C = 0, D = 0$ gives $F = 0$, meaning 0 will be filled in on that spot in the map. 1010 gives back a 1, meaning this will be filled in on that square in the map.
3. Then make the squares following the rules which were mentioned above. Note that every single 1 in the map has to be assigned to a square!
4. To find the Boolean expression F we will do the following:
   - For the blue square we look at the differences of the bits for each of the variables.
     - For the blue part on the left of the map. we can see that we have $A = 1, B = 0, C = 0$ and $D = 0$.
     - For the blue part on the right of the map, we can see that we have $A = 1, B = 0, C = 1$ and $D = 0$.
       * For the Boolean expression, we only write down the variables which don't change, in this case A, B and D. 0 is equal to the complement, thus we get $A\bar{B}\bar{D}$ (which we can also see in the picture above).

Functional Completeness

- Every Boolean function can be expressed as a sum of products of literals (DNF) $\rightarrow$ the set of operators is functionally complete.
- The sets $\{\text{com}, \cdot\}$ and $\{\text{com}, +\}$ are also functionally complete.
- The operators NAND and NOR are each functionally complete on their own.

Extra Exercises

Simplifying expressions
There is a high chance there will be a question in the exam where you have to simplify a given Boolean expression. Therefore, I will here give some of the most important rules to remember when simplifying expressions.

Boolean properties of each operator (which correspond to their truth table)

- AND $(\cdot) \rightarrow A$ AND $B = AB$
- OR $(+) \rightarrow A$ OR $B = AB + \bar{A}B + A\bar{B}$
- XOR $(\oplus) \rightarrow A$ XOR $B = A\bar{B} + \bar{A}B$

and their negations

- NAND $\rightarrow A$ NAND $B = \bar{A}B + A\bar{B} + \overline{AB}$
- NOR $\rightarrow A$ NOR $B = \overline{AB}$
- NEXOR $\rightarrow A$ NEXOR $B = AB + \overline{AB}$

Now as practice I will present some extra exercises to try and of which the answers will be on the next page.

Simplify the following expressions using the laws of Boolean logic:

1. $A \cdot (A + B)$
2. $(A + B) \cdot (A + C)$
3. $AB \cdot (\bar{B}C + AC)$

Answers and Resources

Answers to the questions (note that there are often multiple ways to solve them)

1. $A \cdot (A + B)$: Distributive Law
   $= AA + AB$: Idempotent Law
   $= A + AB$: Distributive Law
   $= A(B + 1)$: Null
   $= A(1)$: Identity
   $= A$

2. $(A + B) \cdot (A + C)$: Distributive Law
   $= AA + AB + AC + BC$: Idempotent Law
   $= A + AB + AC + BC$: Distributive
   $= A(B + C + 1) + BC$: Null
   $= A(B + 1) + BC$: Null
   $= A(1) + BC$: Identity
   $= A + BC$

3. $AB \cdot (\bar{B}C + AC)$: Distributive Law
   $= AB\bar{B}C + ABAC$: Idempotent Law
   $= AB\bar{B}C + ABC$: Inverse
   $= A0C + ABC$: Null
   $= 0C + ABC$: Null
   $= 0 + ABC$: Identity
   $= ABC$

Resources

- Extra exercises:
  https://www.electronics-tutorials.ws/boolean/boolean-algebra-simplification.html
- Video about the simplification of the XOR operator:
  https://www.youtube.com/watch?v=xD8eVy021Dw&list=PL4HCKYuyhl5gjL6dQ-9VL1G_i8EYjpEcN&index=1&t=126s
- Video about a wrapping Karnaugh map:
  https://www.youtube.com/watch?v=KkJfxlZtcKI
- Worksheet with a variety of exercises (Question 25 onwards)
  https://www.ibiblio.org/kuphaldt/socratic/output/boolean.pdf

Week 2: Gates, Circuits and Number Representation

In digital electronics, *gates* are electronic devices that perform logical operations on one or more binary inputs and produce a single binary output. The output of a gate is determined by its logical function, which can be described by a truth table that shows the output for all possible combinations of inputs.

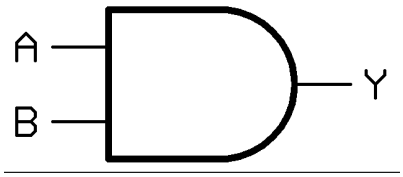
Types of Gates

In the course we handle the AND, OR, XOR, NOT, NAND, and NOR gates, and therefore it is important to know what they look like and how they behave. Here are the gates alongside their truth tables:

1. AND Gate: An AND gate produces a high output (1) only when all of its inputs are high (1).
   The truth table for an AND gate is:

A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

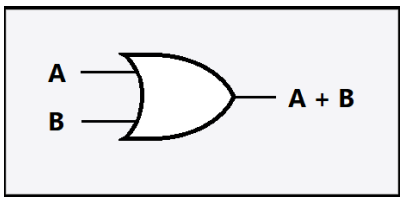
   AND gates are used in digital circuits to implement logical operations such as multiplication and comparison.



2. OR Gate: An OR gate produces a high output (1) when any of its inputs are high (1).
   The truth table for an OR gate is:

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

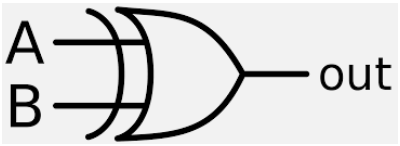
   OR gates are used in digital circuits to implement logical operations such as addition and comparison.



3. XOR Gate: An XOR gate produces a high output (1) when its inputs are different, and a low output (0) when its inputs are the same.
   The truth table for an XOR gate is:

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

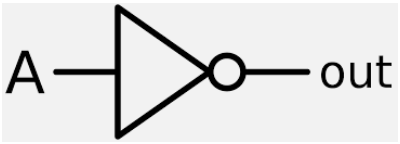
   XOR gates are used in digital circuits to implement logical operations such as exclusive OR and comparison.



4. NOT Gate: A NOT gate, also called an inverter, produces an output that is the complement of its input. That is, it produces a high output (1) when its input is low (0), and vice versa.
   The truth table for a NOT gate is:

A	Output
0	1
1	0

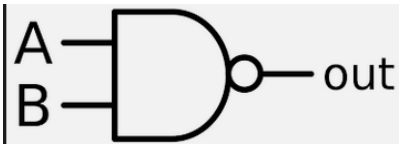
   NOT gates are used in digital circuits to invert or complement a binary signal.



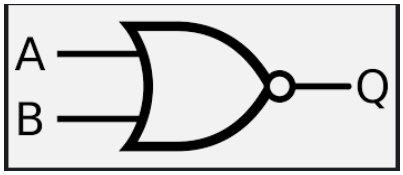
5. NAND Gate: A NAND gate is a combination of an AND gate followed by a NOT gate. It produces a low output (0) only when all of its inputs are high (1).
   The truth table for a NAND gate is:

A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

   NAND gates are used in digital circuits to implement logical operations such as negation and Boolean algebra.



6. NOR Gate: A NOR gate is a combination of an OR gate followed by a NOT gate. It produces a high output (1) only when all of its inputs are low (0). The truth table for a NOR gate is:



Input1	Input2	Output
0	0	1
0	1	0
1	0	0
1	1	0

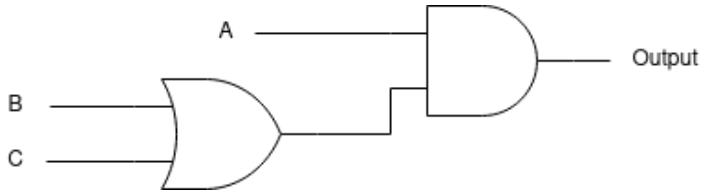
NOR gates are used in digital circuits to implement logical operations such as negation and Boolean algebra. They are also used as universal gates, which means that any other logic gate can be constructed using only NOR gates.

Combination of Gates

In digital electronics, complex logic circuits can be constructed by combining multiple gates together. The output of one gate can be used as the input to another gate, allowing for the creation of more sophisticated logical functions. In the course you will see that it is necessary to create more complex circuits and in real life they will be even more complex and sophisticated.

For example, a circuit that implements the logical function A AND (B OR C) can be constructed by combining an AND gate and an OR gate. The AND gate takes A and the output of the OR gate as inputs, while the OR gate takes B and C as inputs. The output of the AND gate and the output of the OR gate are then combined using a wire or other means to produce the final output. Depending on the input values, the output will differ.

A	B	C	Output
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

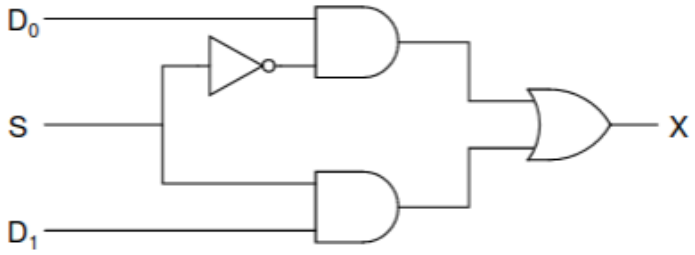


Multiplexer

A *multiplexer*, or MUX for short, is a digital circuit that selects one of several input signals and outputs a single signal based on a set of control signals. A multiplexer has multiple input lines, one output line, and a set of control lines that determine which input line is selected. The number of input lines and control lines is usually specified by a notation such as 2^k, where k is the number of control lines.

Multiplexers are commonly used in digital circuits to select between multiple sources of data, such as memory or input/output devices. They are also used in communications systems to combine multiple signals onto a single channel.

Here's an example of a 2:1 multiplexer which we saw in the lecture:



In this circuit, the two input signals are labeled D_0 and D_1, and the control signal is labeled S. When $S = 0$, the output is equal to D_0, and when $S = 1$, the output is equal to D_1. This is because we choose D_0 or D_1 based on whether the value of S is 1 when it arrives at the AND gate.
The circuit can be implemented using an OR gate, two AND gates, and an inverter, as shown in the figure. The control signal is used to select which input signal is passed through to the output (as described before).

General multiplexer:
A multiplexer with n control inputs selects one of the input signals $D_0, ...D_{2^n-1}$ and passes this to the output.

Decoder:
The binary number on the n inputs determines which of the 2^n outputs will become active (output a '1').

Arithmetic operations

For all arithmetic operations we use the two's complement representation, since this is used in the slides and the course.
Two's complement: based on the idea of all calculations taking place *modulo* 2^n.
A negative number x is represented as $x + 2^n$ (you flip all the bits and add 1 to the LSB).

In this summary I will discuss the addition and subtraction, of binary numbers. If they were to ask a question on the exam it is most likely about the addition of two binary numbers, but in case you're interested you can read the rest as well (you never know what they might ask).

Addition of two binary numbers

Addition of two numbers is probably the easiest to compute and most important to know.

1. Convert the number you have into binary representation (possible a negative number).
2. Then perform addition using a method shown below.
3. Depending on which two numbers are added up, certain flags have to be set (see '*Setting the flags of an operation*').

Example 1:

$235_{dec} + 187_{dec}$

$235_{dec} = 11101011_{bin}$
$187_{dec} = 10111011_{bin}$

$$\begin{array}{rcccccccccccc} & & & & 1 & 1 & 1 & 1 & 1 & & 1 & 1 & \\ & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & \textit{Addend} \\ + & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & \textit{Augend} \\ \hline & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & \textit{Sum} \end{array} \tag{1}$$

So, 125 + 187 = 10100110 = 166.

The red 1 is identified as the 'Carry' in this operation (setting the carry flag to 1).
This bit does not get counted into the final result, meaning the result of this operation is 166 and not 422!
Note that the carry flag only gets set if we treat the result as an unsigned integer!

When adding two signed integers, the Carry flag is not used, as it is only applicable to unsigned arithmetic. In signed arithmetic, the carry-out from the MSB is interpreted as a sign bit, and the result is interpreted in two's complement representation.

If we were to treat the result as a signed integer, then the Overflow flag and the Negative flag should be set.

Example 2:
$-41_{dec} + -99_{dec}$
$41_{dec} = 00101001$
$-41_{dec} = 11010110 + 1 = 11010111$
$99_{dec} = 01100011$
$-99_{dec} = 10011100 + 1 = 10011101$
We therefore get the following:

$$\begin{array}{rcccccccccc} & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ + & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ \hline & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \end{array} \tag{2}$$

Here again, the red 1 is identified as the 'Carry' in this operation (setting the carry flag to 1). This bit does not get counted into the final result, meaning the result of this operation is 116 and not 372! Note that the carry flag only gets set if we treat the result as an unsigned integer!

In this case, the negative flag does not get set since the MSB is 0.

Subtraction of two binary numbers

Subtraction is something which cannot be done in the way we are used to in normal arithmetic. Instead of $a - b$ (1), we calculate it as $a + (-b)$ (2), which ultimately gives us the same answer. To subtract, we have to find the negative number m:

- Find the unsigned representation of $|m|$.
- Invert all the bits.
- Add 1 (00000001 for 8 bits, and possibly apply the carries).

Example:

$-57_{dec} = ??_{bin}$
$57_{dec} = 00111001_{bin}$
$-57_{dec} = 11000110_{bin} + 1 = 11000111_{bin} = 199 = 256 - 57$

What can also happen, is that a negative number becomes a positive number due to rule (2). So for instance 7 - -13 $= a - b$ where $a = 7$ and $b = -13$. If we apply rule (2), we get $7 + (- - 13)$ and we know that $- = +$, but we can't assume this is the case, we do have to show it step by step. The subtraction operation is the reverse order of the addition operation.

Example 1:

For this example, the above mentioned rule of a negative number that becomes a positive number will be applied.
$7_{dec} - -13_{dec}$
$7_{dec} = 00000111$
$13_{dec} = 00001101$
$-13_{dec} = 11110010 + 1 = 11110011$

Applying rule (2), we get 7 + (–13).
$- - 13 = 11110011 + 1 = 11110100$.
We know we want to go from $- - 13$ to -13, so instead of adding a minus, we remove one. This is done by subtracting 1 from 11110100 (–13).
If we do this, the result is 11110011, or otherwise known as -13.
We then subtract 1 again, and flip all the bits to go from -13 to 13.

So we have now concluded that 7 - -13 = 7 + 13 and thus we only have to do a simple addition operation:

$$\begin{array}{cccccccccc} & & & & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & \\ & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & \\ + & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & \\ \hline & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & \end{array} \tag{3}$$

So, the result is 20.

Example 2: ($a < b$)
This example shows the 'original type of subtraction'.
$41_{dec} - 123_{dec}$

$41_{dec} = 00101001$
$123_{dec} = 01111011$
$-123_{dec} = 10000100$
In this example we apply rule (2), meaning the operation becomes 41 + (-123).
This is yet again a simple addition:

$$\begin{array}{cccccccccc} & & & & & & & & {\scriptstyle 1} & \\ & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & \\ + & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & \\ \hline & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & \end{array} \tag{4}$$

This results in 174, but we know that 41 - 123 cannot be 174, so what we have to do is to subtract 256 (to go over the cycle) from 174, meaning we get -82, which is actually 41 - 123.

Example 3: ($a > b$)
$73_{dec} - 72_{dec}$

$73_{dec} = 01001001$
$72_{dec} = 01001000$
$-72_{dec} = 10111000$
Here we again apply rule (2) which gives us the following adding operation:

$$\begin{array}{cccccccccc} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & & & \\ & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ + & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ \hline & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array} \tag{5}$$

I think we can then all see that the result is 1, which is what we wanted the result to be.

NOTE!
The difference between examples 2 and 3 is that in example 2, the a value was smaller than the b value ($41 < 123$).
If that is the case, then you have to subtract 256 from the result you got from the computation $a + (-b)$.
If the a value is bigger than the b value, then you just do a normal addition with the two given binary values and you don't subtract anything (like in example 3).

Logical operations on two binary numbers

Logical operations can also be done on binary numbers, so for instance an OR, XOR or AND operation.
To find the result of the operation, use the truth table of the logical operator to decide whether a 0 or 1 has to be chosen.

Example 1:
150_{dec} OR 77_{dec} = 10010110 OR 01001101.
For the OR operator, we know it is only 0 if both the first number is the second number is 0 (both are False, $0 \vee 0 = 0$).
We therefore get the following:

$$\begin{array}{rcccccccc} & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ OR & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ \hline & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \end{array} \tag{6}$$

So, 150 OR 77 = 11011111 = 223

Example 2:
42_{dec} XOR 69_{dec}
For the XOR operator, we know that it can only be true if you have a either a 1 XOR 0 or a 0 XOR 1.
We therefore get the following:

$$\begin{array}{rcccccccc} & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ XOR & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ \hline & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \end{array} \tag{7}$$

So, 42 XOR 69 = 01101111 = 111.

Setting the flags of an operation

- The CPU doesn't know if it's performing signed or unsigned computation.
- CPU contains flags that are set or cleared according to the outcome of the last arithmetic operation.

We have four types of flags:

- **The Carry flag**
  The *carry* flag is set for an operation when you 'carry' a 1 at the end of an operation.
  The Carry flag is only relevant for unsigned computations.
- **The Overflow flag**
  The *overflow* flag is set when the result of a signed arithmetic operation is too large or too small to be represented in the available number of bits. This can occur when the operands have the same sign, and the sign of the result is different, or when the operands have opposite signs, and the sign of the result is the same as the sign of one of the operands.
  So if we calculate 127 + 7, we have two positive operands (+), but our result involves a MSB which is 1 (negative flag), it can't be a valid result and thus the overflow flag is set.
  The Overflow flag is only relevant for signed computations.
- **The Negative flag**
  The *negative* flag is set to the value of the MSB of the result.
  If the MSB is 1, the negative flag is set, otherwise it is 0. The Negative flag is only relevant for signed computations.
- **The Zero flag**
  The *zero* flag is set whenever the operation ends up giving '0' as a result.

Extra Exercises 1

For each of the following questions, give the computation as its done in the examples above and denote which flags are set (C, O, N, Z), if any.
We interpret the results as signed, not unsigned!

1. $9c_{hex} + 234_{oct}$
2. $15_{dec} - 23_{dec}$
3. $33_{dec} - 18_{dec}$
4. 99_{dec} AND 171_{dec}

Extra Exercise 2

1. Implement an AND gate using NAND gates only.

Answers and Resources 1

1. Whenever you calculate a hex value, split the 8 bits into 2 parts of 4 bits.
   Whenever you calculate a octagonal value, split the amount of bits into parts of 3 bits (it can occur that the final binary value consists of 9 bits, for instance the value 432).

   $9c_{hex} = 1001$ (for 9) concatenated with 1100 (for c) gives 10011100.
   $234_{oct} = (2_{oct} = 010)(3_{oct} = 011)(4_{oct} = 100)$ gives 010011100, including the most significant bit 0.
   We then again perform the standard addition operation:

$$\begin{array}{rcccccccccc} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & & & & {\scriptstyle 1} & {\scriptstyle 1} & \\ & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & \\ + & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & \\ \hline & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & \end{array} \tag{8}$$

   So, $9c_{hex} + 234_{oct} = 100111000 = 312$.

   The flags we set are the Negative flag, since the MSB is 1, and the Overflow flag.
   In this case, the result of the computation, 100111000 in binary, is outside the range of values that can be represented in 9-bit two's complement representation. Therefore, an overflow has occurred, and the Overflow flag should be set to indicate this.

2. $15_{dec} = 00001111$
   $23_{dec} = 00010111$
   $-23_{dec} = 11101001$

$$\begin{array}{rcccccccc} & & & & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & \\ & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ + & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ \hline & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \end{array} \tag{9}$$

   This results in 1111000, which is equal to 248.
   We all know that 15 - 23 is not 248, meaning we have to think of the fact that we have a situation where $a < b$, meaning we have to subtract 256 from the answer we got, which gives us 248 - 256 = -8.

   In the calculation of 15 - 23 using signed binary arithmetic, the result 248 (11111000 in binary) is obtained, which is outside of the range of an 8-bit representation. Therefore, the Overflow flag should be set to indicate that the result is invalid.

   The MSB in the final result is 1, meaning that the Negative flag also gets set as a result of this operation.

3. $33_{dec} = 00100001$
   $18_{dec} = 00010010$
   $-18_{dec} = 11101110$

$$\begin{array}{cccccccccc} & & & & {\scriptstyle 1} & & & & & \\ & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ + & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ \hline & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{array} \tag{10}$$

   So, 33 - 18 = 15, with a carry at the end. In this case, since we work with signed integers, no flags will be set.

4. $99_{dec} = 01100011$
   $171_{dec} = 10101011$
   We get the following:

$$\begin{array}{cccccccc} & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ AND & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ \hline & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \end{array} \tag{11}$$

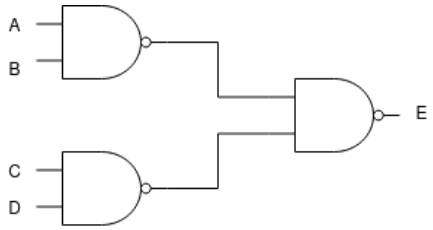
   So, 99 AND 171 = 00100011 = 35.
   There are no flags set in this operation.

Resources:

- Multiplication and Division of numbers:
  https://study.com/academy/lesson/binary-division-multiplication-rules-examples.html#:~:text=For%20binary%20multiplication%2C%20we%20follow,one%20digit%20to%20the%20left.
- More information about the flags can be found using this link:
  https://sites.google.com/rgc.aberdeen.sch.uk/rgcahcomputingrevision/computer-systems/computer-structure/alu-operations

Answer 2

1. To implement an AND gate using NAND gates only, we are going to need 3 NAND gates in total.
   We have the inputs A, B, C and D. A and B are the inputs of the top NAND gate and C and D are the inputs of the bottom NAND gate. The outputs of both gates are then the input of the third NAND gate which produces the output E.
   Visualized this looks like the following:



   Alongside the figure, we can also create a truth table which shows the results:

A	B	C	D	AB	CD	$ABCD$	$\overline{AB}$	$\overline{CD}$	$\overline{ABCD}$	$\overline{\overline{ABCD}}$
0	0	0	0	0	0	0	1	1	1	0
0	1	0	1	0	0	0	1	1	1	0
1	0	1	0	0	0	0	1	1	1	0
1	1	1	1	1	1	1	0	0	0	1

Week 3: Memory and Sequential Logic

This week focuses on extending the digital circuits discussed in Week 2, to account for memory. We so far had seen a **Combinational circuit**:

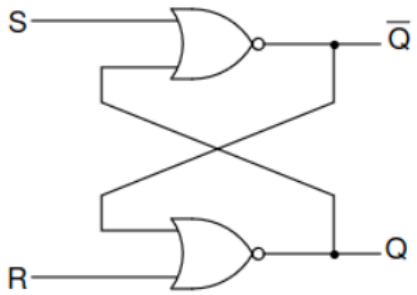
- Output depends only on input at the current time
- Implements a Boolean function
- Examples: acyclic circuit, basic gates, multiplexer, adder, etc.

New this week is the **Sequential circuit**:

- Output of circuit depends on current input and history of previous inputs
- Implements a finite state automaton
- 'Circuit with memory'
- With combinational components using feedback (cyclic circuit)

SR latch

An **SR latch**, also known as a *Set-Reset latch*, is a digital circuit made up of two cross-coupled NOR gates that can store a single bit of information. It has two inputs, S (set) and R (reset), and two outputs, Q and $\overline{Q}$. What makes it special is that a new output is based on the input of a previous output, or **feedback**, as it is called.



For the inputs S and R we have four possible cases:

- **Case 1**

  $$S = R = 0$$

  The states of Q and $\overline{Q}$ are not dependent on the inputs, but on the output (so after the NOR gates), since they can have two different states:
  - $Q = 1$, $\overline{Q} = 0$
  - $Q = 0$, $\overline{Q} = 1$

  If the output of the bottom gate is 1, then we get that $Q = 1$ and $\overline{Q} = 0$.
  If the output of the bottom gate is 0, then we get that $Q = 0$ and $\overline{Q} = 1$.
  We therefore say that the state of Q is based on the previous state of Q (Q_{prev}) and the state of $\overline{Q}$ is based on the previous state of $\overline{Q}$ ($\overline{Q}_{prev}$).

  For instance, if $Q = 0$, and $S = 0$, then the upper NOR gate returns 1, which is then the value of $\overline{Q}$, but it is also the input together with R in the bottom NOR gate, which with a 1 and a 0 returns 0, which is the original value we had for Q. So, $Q = Q_{prev}$.

- **Case 2**

  $$S = 0,\ R = 1$$

  In this case we must have $Q = 0$ and $\overline{Q} = 1$.
  This state is also known as the *Reset* state (since $R = 1$).

- **Case 3**

  $$S = 1,\ R = 0$$

  In this case we must have $Q = 1$ and $\overline{Q} = 0$.
  This state is also knows as the *Set* state (since $S = 1$).

- **Case 4**

  $$S = 1,\ R = 1$$

  In this case we must have $Q = 0 = \overline{Q} = 0$.
  When both inputs of are set to 1 simultaneously, it can cause a race condition, also known as a *metastable state*. In this condition, the outputs of the latch may oscillate or toggle randomly between 0 and 1, making it impossible to determine the correct state of the latch (similar to the superposition of a qubit).

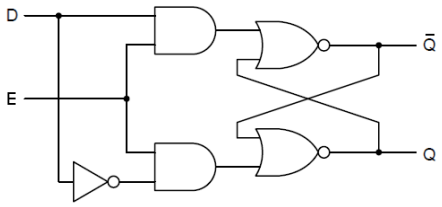
  This is because when both S and R inputs are set to 1, both outputs Q and $\overline{Q}$ are initially set to 0.
  But, the feedback between the two gates can cause a momentary imbalance in the circuit, resulting in unpredictable output values. As for this course, it is just important to know that this state is the 'forbidden state' and must be avoided at all times.

It can be nicely summarized in the following picture:

S	R	Q	$\overline{Q}$	
0	0	Q_{prev}	$\overline{Q}_{prev}$	Hold
1	0	1	0	Set
0	1	0	1	Reset
1	1	0	0	Forbidden

D latch

A **D latch**, also known as a *Data latch* or *Delay latch*, is a digital circuit that can store a single bit of information. What makes this circuit special is that it takes care of the 'forbidden state' we saw with the SR latch, meaning that we cannot end up in the case where $S = R = 1$.



The D latch is similar to the SR latch, but instead of having separate set and reset inputs, it has a single data input that determines the state of the latch.
Here again we have a few different cases:

- **Case 1**
  If $E = 0$, the latch holds its current state regardless of any changes to the input data.
- **Case 2**
  If $E = 1$, then the latch can change its state according to the input data.
  - then if $D = 0$, the latch will perform the 'reset' action
  - then if $D = 1$, the latch will perform the 'set' action

The reason why the state $S = R = 1$ can never be reached is because of the NOT gate that has the D value as input.
If E is 1, then this value will be AND'ed with the negated D value, which is either 0 or 1.
- If $D = 0$, we get that the top AND gate has output 0 and the bottom AND gate has as output 1 (meaning we perform the reset action).
- If $D = 1$, we get that the top AND gate has output 1 and the bottom AND gate has as output 0 (meaning we perform the set action).

So, there is no possible way for the output of both S and R to be 1.
We can also visualize this in the following table:

$D(ata)$	$E(nable)$	Q	$\overline{Q}$
0	0	Q_{prev}	$\overline{Q}_{prev}$
0	1	0	1
1	0	Q_{prev}	$\overline{Q}_{prev}$
1	1	1	0

Synchronization

Sequential logic can be **asynchronous** or **synchronous**.

Asynchronous circuit:

- State changes can happen at any time
- Must take into account signal propagation time
- Difficult to coordinate between different parts of circuit

Synchronous circuit:

- Clock signal
- State changes are synchronized to clock
- **Level triggered**: on constant value of the clock "Clocked D latch" (clock attached to E-input)
- **Edge triggered**: on clock transition $0 \rightarrow 1$ (or $1 \rightarrow 0$) "Flip-flop". For the rest of the course we use this type of trigger (so activate when we reach the high-edge and stop when we reach the lower-edge)

Edge triggered logic

D Flip-Flop

Registers

Week 4: Machine Code, Execution Cycles and Data Paths

Week 5: More Machine Code and Assembly

Week 6: More Assembly

Week 7: Assembly Function Calls and Pipelines