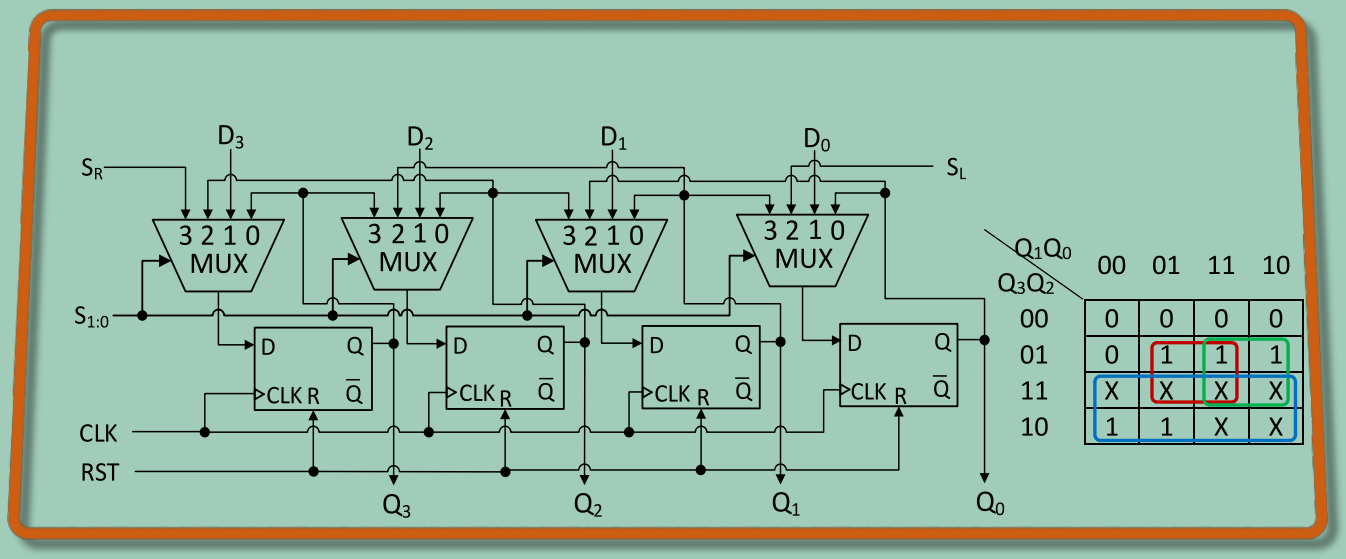
Cristian-Cosmin VANCEA

LOGIC DESIGN

Laboratory Works



U.T.PRESS
Cluj-Napoca, 2026
ISBN 978-606-737-847-4

Cristian-Cosmin VANCEA

LOGIC DESIGN

Laboratory Works

U.T.PRESS
Cluj-Napoca, 2026
ISBN 978-606-737-847-4

Editura U.T.PRESS
Str. Observatorului nr. 34
400775 Cluj-Napoca
Tel.:0264-401.999
e-mail: utpress@biblio.utcluj.ro
https://biblioteca.utcluj.ro/editura

Recenzia: Prof.dr.ing. Radu Gabriel Dănescu
Prof.dr.ing. Florin Ioan Oniga

Pregătire format electronic on-line: Gabriela Groza

ISBN 978-606-737-847-4

Foreword

The Laboratory Works are addressed to students from the Faculty of Automation and Computer Science, who study the domain of Logic Design in the undergraduate cycle. They can also be useful to anyone who wants to study the fundamental techniques of logic design, as the basis for the implementation of digital computing systems. The content of this book represents the foundation for the preparation of practical applications within the domain, and an important means for verifying the acquired knowledge, by getting familiar with a varied range of solutions specific to practical tasks.

The chapters follow a didactic organization focused on a progressive degree of difficulty. It is recommended to approach their study in the order of appearance within the book. The objectives are detailed using a theoretical formalism, with the role of substantiating knowledge, but considering a complete experience based on practical implementation and testing of the circuits in the operative state. As the subject covers a vast area, the objectives aim particularly the deepening of the knowledge about the functioning of essential components within the digital systems, in view of the subsequent approach of broader subjects, such as the implementation of automata and processor microarchitectures.

Regarding the circuits studied, both individual operating concepts and ways of integration into more complex architectures are presented, thus providing a higher perspective, which combines the understanding of hardware components with the formation of creative skills, aimed to designing circuitry with more advanced computing capabilities. To facilitate implementation and testing, a framework was developed for the ISE Project Navigator application, and a simulation library was developed for the Logisim application. Together, they complete the reader's experience both at a practical level, on FPGAs, and within a simulator.

Within the struture of the laboratory works, the information follows a constructive approach. Each chapter presents the objectives, followed by the theoretical concepts necessary for the implementation. The practical activities proposed at the end of each chapter mittigate the development of hardware design skills through concrete implementation and testing sessions. When several solutions are available, comparative analysis can be emerged, similar to research activities. To facilitate the comprehension of the hardware design principles, it is recommended to carefully read each chapter before the practical assignments. The appendices provide solutions to common hardware applications, by using the circuits studied throughout the book.

The author wishes you a pleasant and enriching reading experience!

CONTENTS

1 Design and implementation using didactic boards

1.1 Objectives

This work presents the resources necessary for laboratory activity. The basic notions used in modelling the behavior of digital circuits are studied. A step-by-step guide presents the development methodology with Xilinx ISE suite and the implementation on FPGA (Field Programmable Gate Array) boards. The activity will be carried out into an already existing project that integrates the TTL (Transistor-Transistor Logic) circuits. The project is described and analyzed in detail. Several circuits are implemented using basic logic gates and their functionality is tested according to their function (truth) table.

1.2 Resources necessary

The following resources are required for the implementation and testing of the circuits:

- Nexys A7 development board, The reference manual [1].
- Xilinx ISE WebPACK 14.7 suite [2]; The lab works will be carried out in an already existing project, which contains the TTL circuits.

1.3 Fundamentals on modelling digital circuits behavior

The mathematical background behind modelling the behavior of digital circuits uses concepts from *boolean algebra* (conceived by mathematician George Boole). It contains a set of logical operations, which are commonly used to describe their functionality. A digital circuit has several inputs and outputs, called *terminals* (or pins). Any output can be modeled as a function of the inputs, which means the inputs are the variables of the function. In boolean algebra the result of the function (output) and its variables (inputs) can have two values, either 1 (true) or 0 (false). These functions are also known as *boolean functions*. The values are associated with electrical properties, such as voltage. For transistor-based circuits of the TTL family [3], value 0 is associated with low voltage and 1 is associated with high voltage. The representation is called *positive logic*. When the polarity is reversed (0 means high voltage and 1 means low voltage) the representation is in *negative logic*.

Any boolean function can be described by its *truth table* (also called *function table*). The left side of the table contains all possible combinations of input values. The function value (response) for each combination is placed on the right side. An example of truth table for a 2-variable function is presented in Table 1. 1, left side. According to the table, the associated circuit will output 1 when x_1=1 and x_2=1, hence its name, the AND function. Considering its simplicity, the circuit is also called a *basic logic gate*. In boolean algebra AND uses the · (dot) operator (not to be confused with multiplication). The corresponding graphic symbol used within a circuit is  , with inputs on the left and output on the right. There are AND gates with higher number of inputs, which implement

x_{n-1}·…·x_0. They respect the same behavior: the output is 1 only when all inputs are 1. Since AND is associative, the extension to a higher number of inputs can be implemented using AND gates with less inputs. For instance, based on the property that $x_2 \cdot x_1 \cdot x_0 = x_2 \cdot (x_1 \cdot x_0)$, a 3-input AND gate can be implemented with 2-input AND gates: 

.

Table 1. 1 Truth-tables for AND (left), OR (middle) and NOT (right) gates

x_1	x_0	AND
0	0	0
0	1	0
1	0	0
1	1	1

x_1	x_0	OR
0	0	0
0	1	1
1	0	1
1	1	1

x	NOT
0	1
1	0

There are other types of basic logic gates, such as OR and NOT (also called INV). The operator for OR is represented by a + (attention! it doesn't represent summation). The expression x_1+x_0 will be 1, only if x_1=1 or x_0=1 (see Table 1. 1, middle). The graphic symbol for OR is  . NOT is a 1-input gate and inverts its value. The representation in boolean algebra is NOT = $\bar{x}$. Its symbol is  . AND, OR and NOT gates implement the basic operations in boolean algebra. By repeated interconnections, an unlimited number of circuits can be implemented, representing boolean functions with any number of input variables. For instance, the circuit implementing $x_2 + (\bar{x_1} \cdot x_0)$ is 

.

The priority of the operators inside a boolean expression is: 1) brackets; 2) NOT; 3) AND; 4) OR. The operations inside the circuit follow the rules from the truth tables, in left to right order, from inputs to outputs. Considering x_2=0, x_1=0 and x_0=1, and replacing their values inside the expression, the result is $x_2 + (\bar{x_1} \cdot x_0) = 0 + (\bar{0} \cdot 1) = 0 + (1 \cdot 1) = 0 + 1 = \mathbf{1}$. Similarly, the expression can be computed for any combination of input values, and the truth table can be generated. Several pairs of input values are presented as follows, with the corresponding computation of the expected values:

- x_2=1, x_1=0, x_0=0 → $x_2 + (\bar{x_1} \cdot x_0) = 1 + (\bar{0} \cdot 0) = 1 + (1 \cdot 0) = 1 + 0 = \mathbf{1}$;
- x_2=0, x_1=1, x_0=1 → $x_2 + (\bar{x_1} \cdot x_0) = 0 + (\bar{1} \cdot 1) = 0 + (0 \cdot 1) = 0 + 0 = \mathbf{0}$;
- x_2=0, x_1=0, x_0=0 → $x_2 + (\bar{x_1} \cdot x_0) = 0 + (\bar{0} \cdot 0) = 0 + (1 \cdot 0) = 0 + 0 = \mathbf{0}$.

1.4 Xilinx ISE Project Navigator – a step-by-step guide

The Project Navigator application is part of Xilinx ISE suite and offers support for designing digital circuits using a schematic editor.

1.4.1 Loading the *ttl_env* project

After launching Project Navigator, the **ttl_env** [4] project must be loaded from File > Open Project… in the menu and selecting ttl_env.xise project file. Next, the application will display several working panels, as in Figure 1. 1. Some panels on the lateral side are dedicated to project management and those at the bottom offer information related to

logic diagram interpretation and processing [**Note**: If the graphical user interface differs it can be reset to default from Layout > Load Default Layout.]:

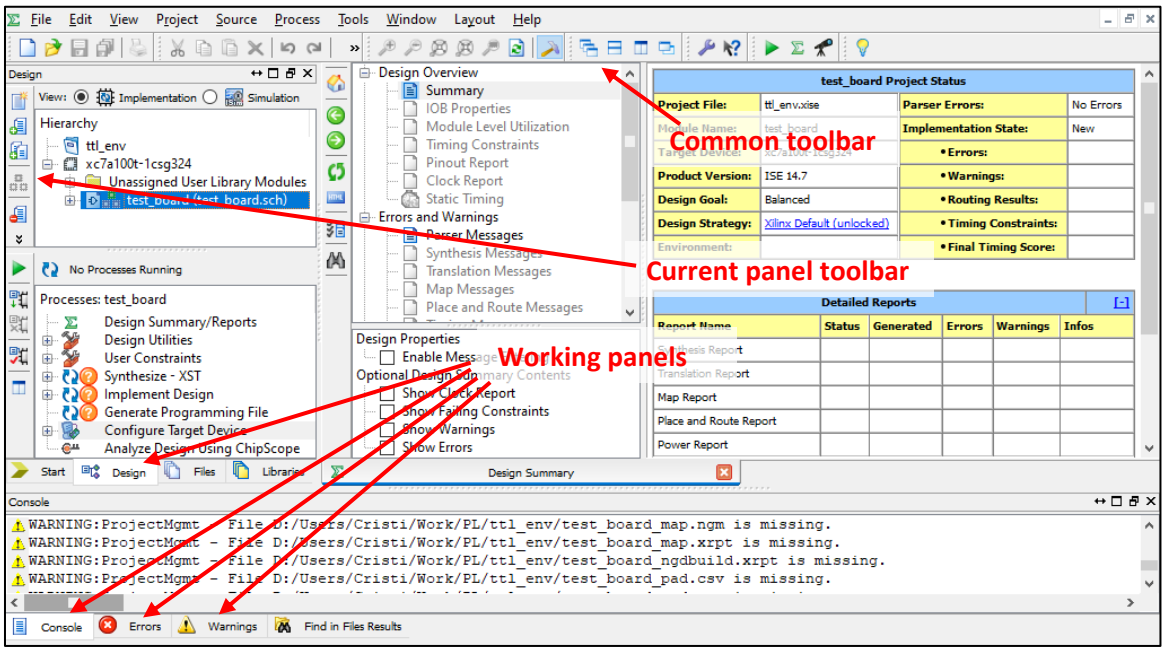


Figure 1. 1 Project Navigator interface

- **Design** panel (Figure 1. 1) – highlights the project structure organized as a set of interconnected circuits. Initially, the project contains the test circuit called test_board. With double-click on the name of the circuit the schematic editor will appear, and the Symbols panel will open.
- **Symbols** panel (Figure 1. 2) – contains the list of components that can be integrated into the circuit diagram, represented by their symbols. It can be activated by clicking on Add Symbol in the toolbar of the schematic editor. In this panel the symbols are organized into several categories. Selecting a category in the upper layer, you will notice the list of corresponding circuits bellow. When selecting <--All Symbols--> at the top, the list of all available components will appear. A component can be searched into the current category by introducing its name in the Symbol Name Filter box. The most relevant categories are *Arithmetic*, *Buffer*, *General*, *Logic*, *Mux* and *Decoder*. The category next to <--All Symbols--> represents the local library attached to the project and contains the circuits from the TTL family. **Note**: The name of these circuits starts with **TTL_** followed by the code of the circuit.
- **Console**, **Errors**, **Warnings** (Figure 1. 1) – these panels display relevant information when the diagram is processed. It is useful for debugging features. The **Console** displays all messages generated in the process, while **Errors** and **Warnings** are related to specific messages. A typical warning appears when some inputs are not connected. In such a case the inputs will be automatically connected to 0 (Ground – GND). For connectionless outputs, Project Navigator warns about pruning the circuitry. Unlike errors, the warnings do not negatively affect the implementation process.

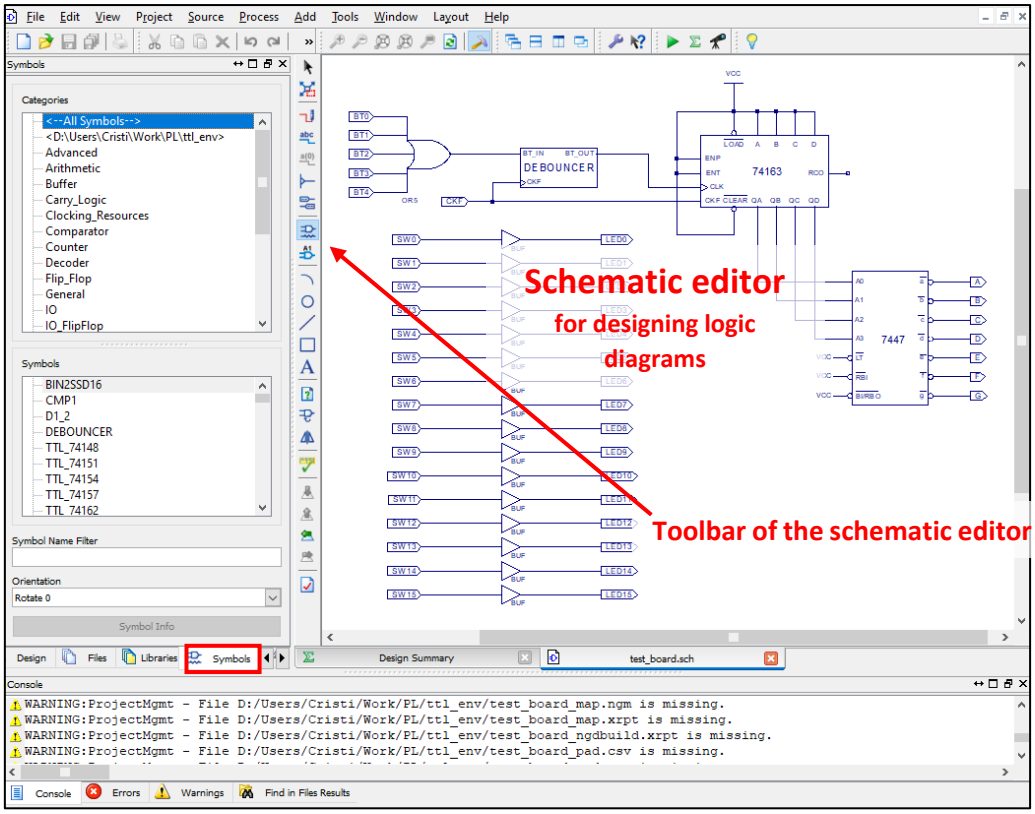


Figure 1. 2 Symbols Panel and the Schematic Editor

1.4.2 Creating a new diagram

A blank diagram can be added to the project by pressing New Source button in the toolbar of the Desing panel. In the next step, the type of source will be set to Schematic, and the name of the diagram will be chosen (without special characters or spaces): e.g., **lab01_01** (Figure 1. 3). **Note**: Verify that Add to project is checked. Then press Next and Finish. The schematic editor will highlight the empty diagram, which can be added new components and connections. Zooming is possible using Zoom In and Zoom Out in the common toolbar at the top.

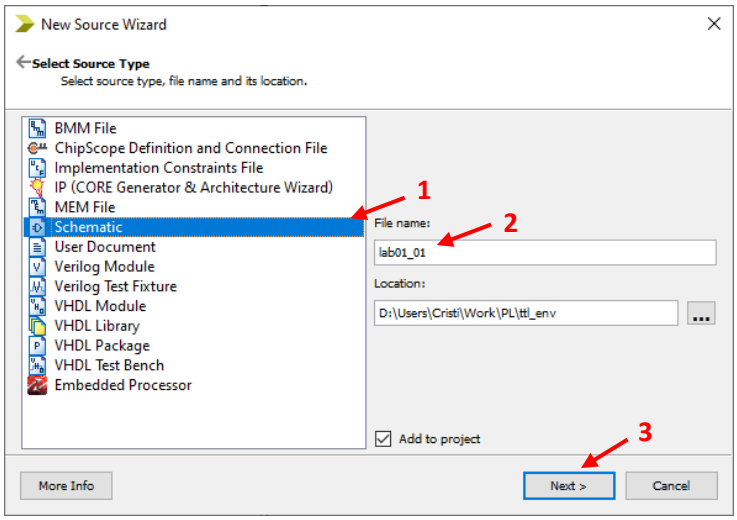


Figure 1. 3 Setting up a new diagram

When the project has several diagrams attached, only one can be set active. Setting **lab01_01** active is possible by right-clicking its name in the Desing panel and choosing Set as Top Module from the list of options. The active diagram is marked by the symbol in the project hierarchy. The active diagram can be changed at any moment.

A diagram can be removed by right-clicking its name and choosing Remove from the list of commands.

Inside the diagram you will introduce an AND component with two inputs. In the Symbols panel select <--All symbols--> at the top. The search will include the entire list of components. Then type *and2* (2 indicates the number of inputs) in the Symbols Name Filter (Figure 1. 4). Click on **and2** in the filtered list and finally, click on the diagram to place the gate. You can add more gates by clicking several times. (**Note**: You can use the Orientation setting to change the symbol appearance before being placed on the diagram.)

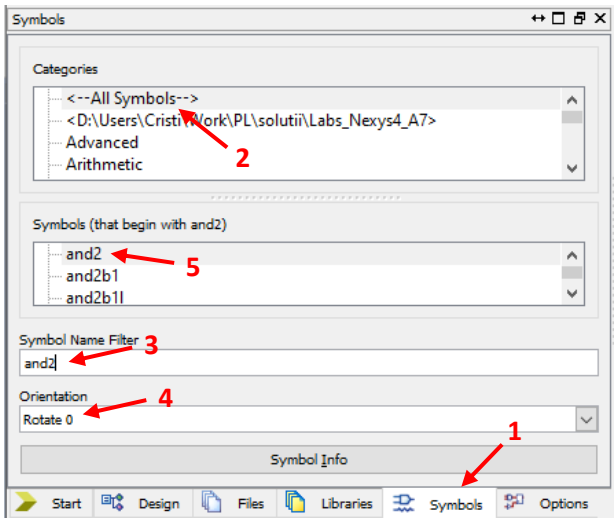


Figure 1. 4 Steps required for selecting **and2** in the list of symbols

The selection of a component within the diagram is possible by enabling the selection mode with a click on Select button. Once selected, an element can be drag-and-dropped and can be removed with right-click and Delete (or simply press the *Del* key).

Next, you will introduce wires on the inputs and outputs of AND2 gate. Click on Add Wire in the toolbar to enter the wiring mode. The wire can be placed by drawing it between the interconnected points. You should notice a specific mouse hover near connection points, meaning a click will generate a connection in place. A wire is finished when both ends are connected. An unconnected end is marked by a red rectangle.

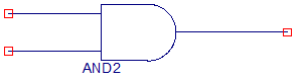


Figure 1. 5 Wires attached to AND2 having unconnected end points

At unconnected ends you will place input and output terminals. The input terminals are required to introduce signal values (0 or 1) in the circuit. The output

terminals carry the results outside the circuit. Click on the Add I/O Marker button and place terminals near the end points of the wires to mark connections. (**Note**: Terminals can be attached directly to the pins of a symbol without wires.) The proper type of terminal (input or output) is automatically chosen by Project Navigator. The name of the terminal is automatically generated, nevertheless it is possible to change it by right-clicking the terminal and choosing Rename Port. The name should not contain special characters or spaces. Rename the input terminals to A and B and the output terminal to F1 (Figure 1. 6). **Note**: Wires connected to terminals carry their names, automatically. **Important**: Wires with a similar name are considered connected (a *logical connection*).

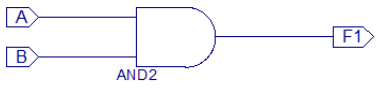


Figure 1. 6 The AND2 gate connected to input and output terminals

A new output function F2 will be added to the diagram. This function will extend the AND operation to 3 inputs using the already existing AND gate. Consequently, a new AND gate will be added to the diagram. The two inputs of this gate will be connected, one to the output of the first AND, and one to a third input terminal C. Its output will be connected to a second output terminal F2. The steps required are as follows:

1. Add a new AND2 gate to the diagram from the Symbols panel.
2. Connect the output of the previous AND2 to one of the inputs of the new gate. Doing so, will generate a new branch on the wire connected to F1, marked by a rectangle. **Note**: All branches represent the same wire and always carry the same value.
3. Connect wires to the free pins of AND2.
4. Link a C terminal on the unconnected input wire and an F2 terminal on the unconnected output wire. The result should be as in Figure 1. 7.
5. Save the diagram using Ctrl+S or File > Save from the menu.

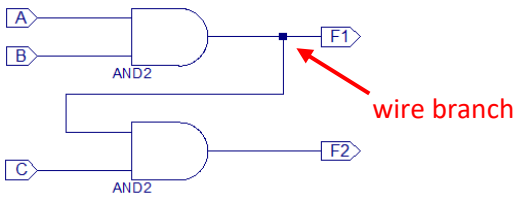


Figure 1. 7 Circuit with two output functions: F1=A · B and F2=(A · B) · C

Next, connect the input terminal C to a new output terminal named F3. **Note**: A direct connection from an input terminal to an output terminal is only allowed though a buffer called BUF. BUF can be found in the category *General*. After adding a BUF, the circuit should be as highlighted in Figure 1. 8.

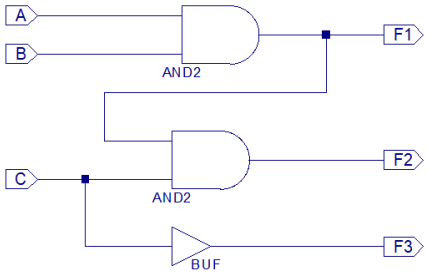


Figure 1. 8 Circuit with three output functions: F1=A · B, F2=(A · B) · C and F3=C

1.4.3 Defining design constraints

The circuits designed using the schematic editor are tested on the board shown in Figure 1. 9. It has **16 switches** and **5 push buttons** for generating logical values on the input terminals. The results registered on output terminals can be visualized on **16 leds**.

When pushed, a **button** generates 1, otherwise it always generates 0. A **switch** generates 1 when turned towards the leds, and 0 on the other side. A **led** is off when receiving 0 and lights up when receiving 1. The switches and the leds are indexed from 0 to 15, in right to left order.

The input and output terminals in a diagram are associated (constrained) with buttons, switches and leds on the board. The correspondence is defined in a text file having the .ucf (User Constraints File) extension. Every diagram must be assigned a constraints file. The file is not case sensitive. To ease the task of elaborating the constraints file from scratch there is a *_template.ucf* file within the project folder, which entails all possible resources, in comments. A comment starts with a # character. Your task will be to uncomment only those resources necessary for the current diagram.

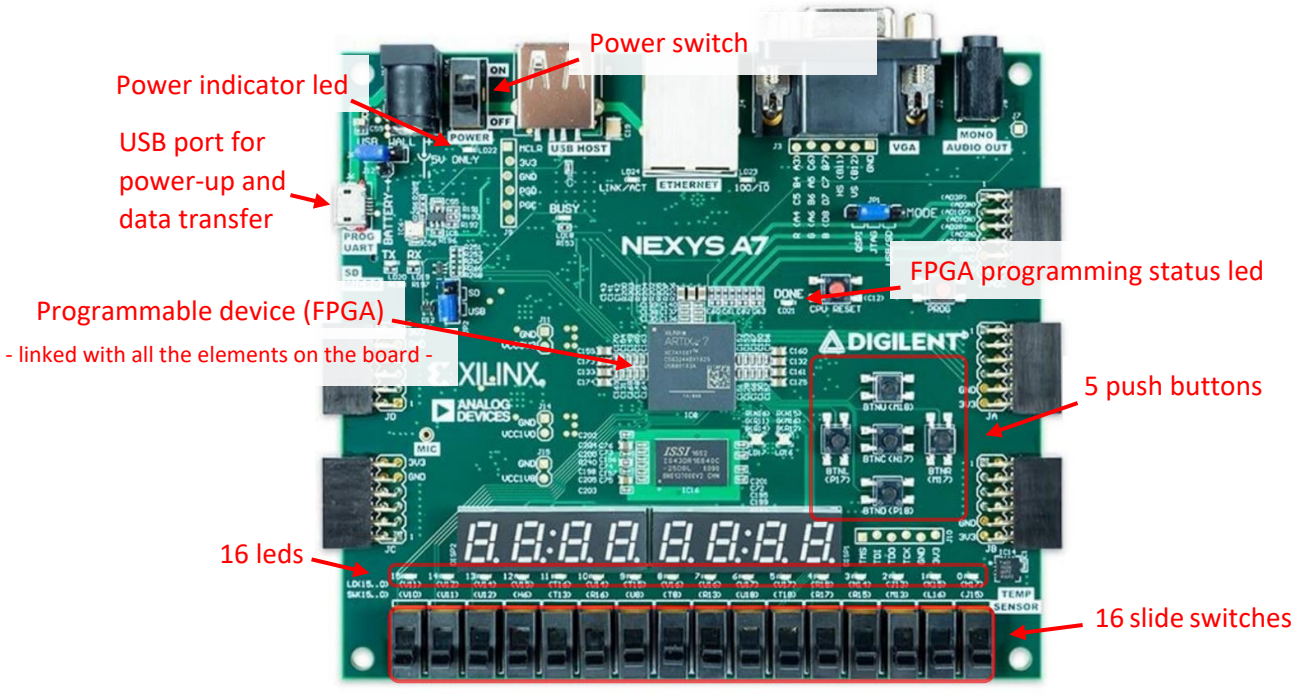
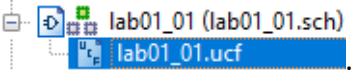


Figure 1. 9 Testing board

Before assigning a constraints file, a copy of *_template.ucf* should be renamed as the diagram, still keeping the .ucf extension. Hence, for diagram lab01_01 a copy of *_template.ucf* should be renamed to *lab01_01.ucf*. The assignment can be carried out if lab01_01 is the active diagram of the project (the top module). This is indicated by the symbol in the Design panel. Check this out and then press the Add Source . Select *lab01_01.ucf* from the disk and finish the process. The constraints file will appear under the diagram name in the project hierarchy 

. Double-click the constraints file in the hierarchy to open for edit.

Inside the constraints file you will notice all available resources carrying predefined names in double quotes, immediately after the keyword NET. The lines corresponding to leds, and switches necessary for the current diagram, will be uncommented by removing the preceding # character. Also, their names will be adjusted to match the names of the terminals within the diagram. Considering that input terminals A, B, C will be associated with switches 0, 1 and 2 respectively, you will have to uncomment lines 13-15 and modify them as follows (follow the changes in red square):

```
## Switches
NET "A" LOC=J15 | IOSTANDARD=LVCMOS33 | CLOCK_DEDICATED_ROUTE=FALSE; # sw0
NET "B" LOC=L16 | IOSTANDARD=LVCMOS33 | CLOCK_DEDICATED_ROUTE=FALSE; # sw1
NET "C" LOC=M13 | IOSTANDARD=LVCMOS33 |CLOCK_DEDICATED_ROUTE=FALSE; # sw2
```

Note: The values of LOC attribute (LOC is short for location) are identical with the codes mentioned on the board near the switches. The situation is similar for push buttons and leds.

Considering F1, F2 and F3 will be connected to leds 0, 1 and 2, lines 40-42 will be uncommented and changed accordingly:

```
## LEDs
NET "F1" LOC=H17 | IOSTANDARD=LVCMOS33; # led0
NET "F2" LOC=K15 | IOSTANDARD=LVCMOS33; # led1
NET "F3" LOC=J13 | IOSTANDARD=LVCMOS33; # led2
```

Save the constraints file with *Ctrl+S* or File > Save from the menu.

1.4.4 Generating the programming file

The programming file can be generated from **Generate Programming File** at the bottom of Design panel (Figure 1. 10), by a right-click and choosing one of the commands available: Run, ReRun or Rerun All. Various messages will appear on the console. In case of errors, depending on their gravity, the entire process might stop. The issues reported must be solved accordingly, and the entire process relaunched with Rerun All.

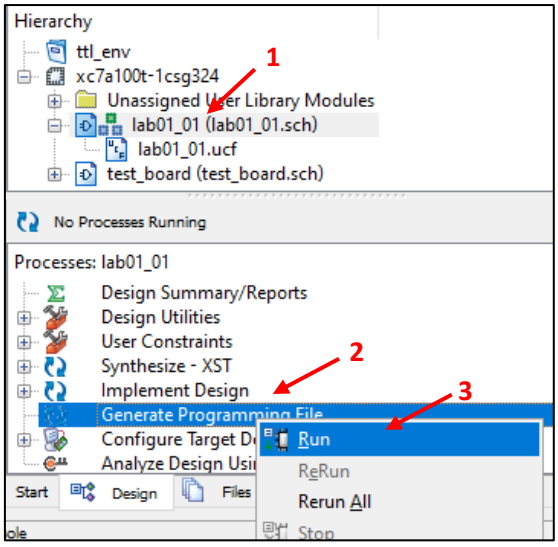


Figure 1. 10 Generating the programming file

If the programming file is successfully generated, the file lab01_01.bit will appear in the project folder along with a green icon: Generate Programming File.

1.4.5 Programming the FPGA and testing on the board

Important: Before any programming session, the board (Figure 1. 9) must be connected to the working station using a USB cable and must be powered-up from the power switch. The power indicator led will light up.

You will launch the ISE iMPACT tool by right-clicking on **Configure Target Device** and selecting Run from the menu (Figure 1. 11).

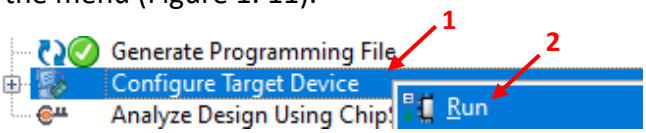


Figure 1. 11 Launching the ISE iMPACT tool

A new working session ca be created inside ISE iMPACT by double-clicking on Boundary Scan (Figure 1. 12).

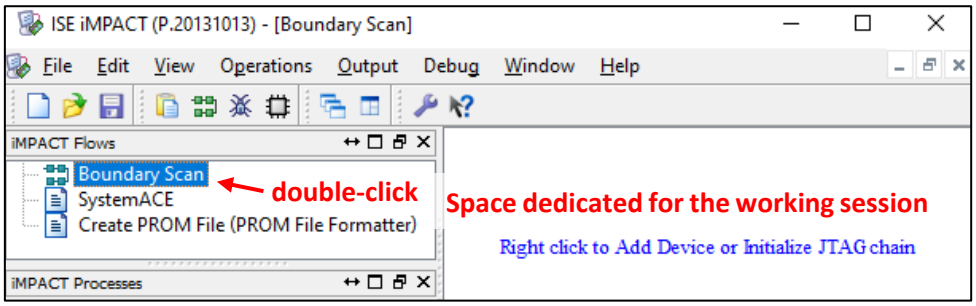
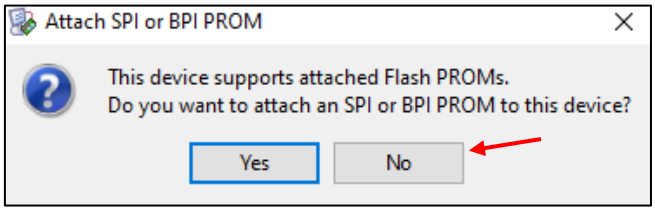


Figure 1. 12 Creating a working session in ISE iMPACT

Steps required to initialize the connection with the board:

1. Right-click in the working session and choose Initialize Chain.

2. Confirm the association with the programming file and select lab01_01.bit from the project folder.
3. **Important**: Always refuse the attachment of SPI or BPI PROM when asked.



4. Press OK in the next window showing programming settings.

A successful connection will highlight the FPGA symbol in the working space, its model, and the name of the current configuration file, as in Figure 1. 13.

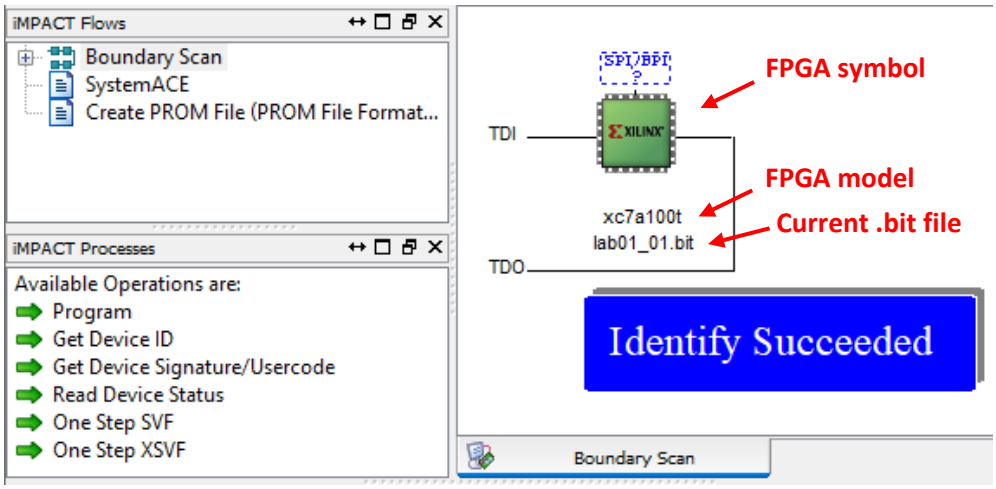


Figure 1. 13 ISE iMPACT has successfully connected to the board

The final step is the physical configuration of the FPGA board with the programming file. Right-click on the circuit symbol and select Program from the list (Figure 1. 14). Immediately after this step you will notice the programming status indicator lights up. From this moment the board behaves according to the circuitry on the diagram. You can use the switches to send 0 and 1 input values, and you will notice the results on the leds: F1=1, if A=B=1, F2=1, if A=B=C=1 and F3=C.

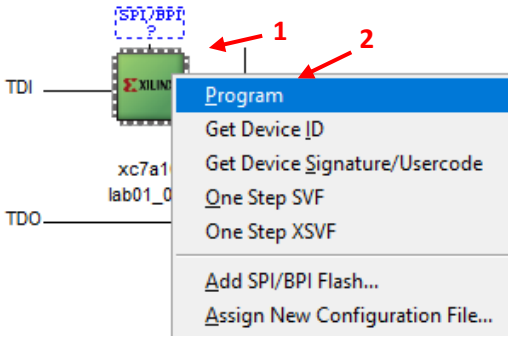


Figure 1. 14 Launch the programming session after attaching the .bit file

The current configuration is maintained for as long as the board is powered and will be lost when switching off. The ISE iMPACT session can be maintained even when new

changes are made to the current diagram in Project Navigator. To save time, you should not close ISE iMPACT after each tested circuit. If you apply changes to the diagram and generate a new programming file with the same name, you may return to ISE iMPACT and Program the board without having to pass through all the steps required for connection.

If the name of the programming file changes, supposing you want to test another diagram, you should choose Assign New Configuration File… first (Figure 1. 14). You will have to select the new .bit file from the disk and then you can Program the board. Again, passing through all the steps required for connection is not necessary.

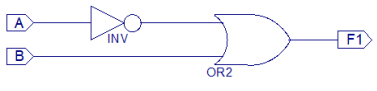
Note: When closing the ISE iMPACT (at the end of the class), you will be asked to save the current project. It is not recommended to save the current project in ISE iMPACT.

Important: Project *ttl_env* can be cleaned by files generated throughout time, by executing the script clean_win.bat in Windows or clean_lnx.sh in Linux. Both scripts can be found in the project folder. They will preserve the .sch diagrams and the .ucf files.

1.5 Assignments

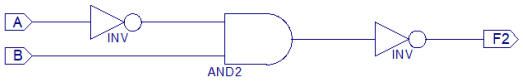
1. Add the following circuits to the *ttl_env* project (use a separate diagram for each circuit) and test them on the board using the truth tables. Name the diagrams and use switches and leds of your choice.

a) F1 = $\overline{A} + B$



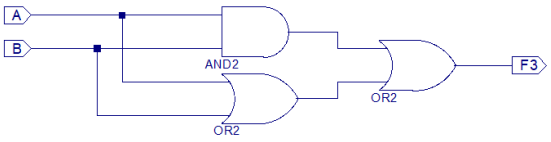
A	B	F1
0	0	1
0	1	1
1	0	0
1	1	1

b) F2 = $\overline{\overline{A} \cdot B}$



A	B	F2
0	0	1
0	1	0
1	0	1
1	1	1

c) F3 = $(A \cdot B) + (A + B)$



A	B	F3
0	0	0
0	1	1
1	0	1
1	1	1

1.6 Bibliography

[1] Digilent, “Nexys A7 Reference Manual”, Available online: https://digilent.com/reference/programmable-logic/nexys-a7/reference-manual

[2] AMD Xilinx, “ISE WebPACK Design Software”, Available online: https://wiki.archlinux.org/title/Xilinx_ISE_WebPACK

[3] Wikipedia, “Transistor–transistor logic”, Available online: https://en.wikipedia.org/wiki/Transistor-transistor_logic

[4] Cristian-Cosmin Vancea, “TTL_Env Project for Project Navigator”, Available online: https://drive.google.com/uc?export=download&id=1800AdR6Vdo8PDd1tB9n5LCQZNHAcgh9K

2 Basic logic gates

2.1 Objectives

The fundamental principles and properties of digital circuits are presented from a technological perspective. The basic logic gates are introduced and analyzed using the operations from boolean algebra. Given the priority of the boolean operators, the computation of the truth table for a boolean expression is detailed. The most common properties of boolean algebra are listed and several operational conversions are explained at gate level as well as implementing new functionalities.

2.2 Fundamental technological aspects

The basic logic gates are the primary circuits used in logic design. They have simple functionality and contain a small number of transistors or semiconductors. They implement the basic operations in boolean algebra: NOT (INV), OR, AND. Gates implementing combined functionality such as NOR (NOT OR), NAND (NOT AND), XOR (eXclusive OR) and XNOR (eXclusive NOR) are also considered basic gates due to the simplicity of their electronic compound.

From a technological perspective there are two common categories of gates, the TTL (Transistor-Transistor Logic) and the CMOS (Complementary Metal-Oxide Semiconductor). The most important parameters defining the two families are related to tolerance against variations, switching speed and power consumption.

- *tolerance against variations* – according to values in Figure 2. 1 the CMOS circuits have an extended voltage interval for state 0, which means a better tolerance against oscillations around small values and a smaller chance for an uncontrolled exit outside this state, as compared to TTL circuits.

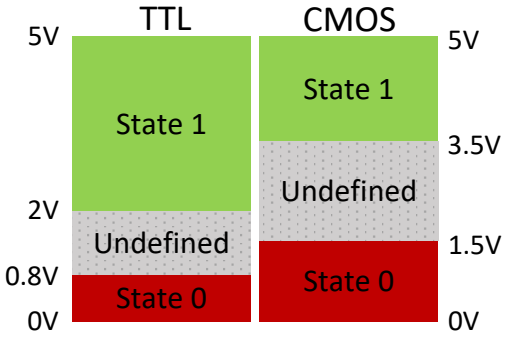


Figure 2. 1 Voltage level association with logic states

- *switching speed* – this parameter represents the reaction time when a state transition should occur on output due to changes of values on one or more inputs. Usually, the CMOS gates are slower than their counterpart.

- *power consumption* – the TTL circuits are power hungry, hence mobile devices running on batteries or accumulators are equipped with CMOS gates, for a longer life period.

Circuits from the TTL family will be studied in detail. In dedicated electronic catalogues they can be identified by their code composed of more digits. The initial two digits are either 74 or 54. Considering the properties mentioned earlier several categories of TTLs can be identified:

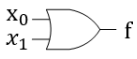
- standard.
- low power.
- high-speed.
- Schottky (the fastest).
- other combinations (e.g., Low-Power Schottky).

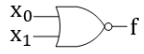
2.3 The basic logic gates

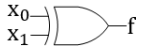
The basic logic gates implement the functionality of boolean operators with one or two inputs (variables) and can be extended to more inputs. Their expression, the truth tables and the associated graphic symbol for logic diagrams are listed in Table 2. 1.

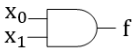
Table 2. 1 The basic logic gates

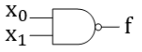
<table>
<tr>
<td colspan="3">NOT (INVerter)
$f = \overline{x_0}$

<table>
<tr><th>x_0</th><th>f</th></tr>
<tr><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td></tr>
</table>
</td></tr>
<tr>
<td>OR
$f = x_1 + x_0$
 

<table>
<tr><th>x_1</th><th>x_0</th><th>f</th></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
</table>
</td>
<td>NOR (NOT OR)
$f = \overline{x_1 + x_0}$
 

<table>
<tr><th>x_1</th><th>x_0</th><th>f</th></tr>
<tr><td>0</td><td>0</td><td>1</td></tr>
<tr><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>0</td></tr>
</table>
</td>
<td>XOR (eXclusive OR)
$f = x_1 \oplus x_0$
 

<table>
<tr><th>x_1</th><th>x_0</th><th>f</th></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>0</td></tr>
</table>
</td>
</tr>
<tr>
<td>AND
$f = x_1 \cdot x_0$
 

<table>
<tr><th>x_1</th><th>x_0</th><th>f</th></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
</table>
</td>
<td>NAND (NOT AND)
$f = \overline{x_1 \cdot x_0}$
 

<table>
<tr><th>x_1</th><th>x_0</th><th>f</th></tr>
<tr><td>0</td><td>0</td><td>1</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>0</td></tr>
</table>
</td>
<td>XNOR (eXclusive NOR)
$f = \overline{x_1 \oplus x_0} = x_1 \odot x_0$
 

<table>
<tr><th>x_1</th><th>x_0</th><th>f</th></tr>
<tr><td>0</td><td>0</td><td>1</td></tr>
<tr><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
</table>
</td>
</tr>
</table>

An analysis of the symbols shows that logic inversion is usually represented by a small circle and the mathematical operator is represented by a bar on top of the inverted boolean expression. Also, you should not confuse the + operator with addition, nor the · operator with multiplication from arithmetic.

From the truth tables, the effect of the logic gates can be summarized as follows:

- the NOT gate inverts the input.
- the result of OR is 1 if at least one input is 1 – the equivalent for max function.
- the result of AND is 1 if all inputs are 1 – the equivalent for min function.
- NOR is the inversion of OR.

- NAND is the inversion of AND.
- the result of XOR is 1 when both inputs are not equal.
- the result of XNOR is 1 when both inputs are identical.
- XNOR is the inversion of XOR.

The logic gates are associated with unique codes (Table 2. 2), which identify the boolean operation.

Table 2. 2 Codes associated with TTL basic logic gates

Gate	Code
NAND	7400
NOR	7402
NOT	7404
AND	7408
OR	7432
XOR	7486
XNOR	74266

Note: In Project Navigator the basic logic gates are grouped in the *Logic* category and can be searched by their name (AND, OR, INV etc.). INV replaces NOT.

Observation: An alternative implementation for NOR is to connect the output of OR to the input of NOT. This solution is less efficient since it uses two gates, therefore doubles the resources used. Also, the speed decreases because the propagation path covers both gates, and the power consumption increases. Nevertheless, the preferred solution is the use of dedicated logic gates implementing the behavior described by the truth table of NOR. The situation is similar for NAND, XOR and XNOR.

2.4 Generating the truth table for a boolean expression

The computation of an expression is based on the priority of operators presented in Table 2. 3. The results are computed for all combinations of values associated to the variables. For *n* variables there are 2^n combinations.

Considering the expression $\overline{a \cdot b} + c$ the first operation will be the NAND over a and b, followed by the operation OR over the previous result and the variable c. The computations will use the specific truth tables of the operations. According to this rule the intermediary values and results are presented in Table 2. 4.

Table 2. 3 Priority of operators in boolean algebra

Operator	Priority
()	high
$\overline{}$	
$\bullet$	
$\oplus, \odot$	
+	low

Table 2. 4 Computing the truth table for $\overline{a \cdot b} + c$

a b c	$\overline{a \cdot b} + c$
0 0 0 :	$\overline{0 \cdot 0} + 0 = 1 + 0 = \mathbf{1}$
0 0 1 :	$\overline{0 \cdot 0} + 1 = 1 + 1 = \mathbf{1}$
0 1 0 :	$\overline{0 \cdot 1} + 0 = 1 + 0 = \mathbf{1}$
0 1 1 :	$\overline{0 \cdot 1} + 1 = 1 + 1 = \mathbf{1}$
1 0 0 :	$\overline{1 \cdot 0} + 0 = 1 + 0 = \mathbf{1}$
1 0 1 :	$\overline{1 \cdot 0} + 1 = 1 + 1 = \mathbf{1}$
1 1 0 :	$\overline{1 \cdot 1} + 0 = 0 + 0 = \mathbf{0}$
1 1 1 :	$\overline{1 \cdot 1} + 1 = 0 + 1 = \mathbf{1}$

a b c	$\overline{a \cdot b} + c$
0 0 0	1
0 0 1	1
0 1 0	1
0 1 1	1
1 0 0	1
1 0 1	1
1 1 0	0
1 1 1	1

2.5 Designing the logic diagram for an expression

The connection of gates in a logic diagram follows the operator priorities. Applying the priorities from Table 2. 3 over the expression $f = b + a \cdot \overline{a + (b + a)}$, the result can be calculated using the following steps:

1. P1 = $b + a$ (the OR inside brackets)
2. P2 = $\overline{a + P1}$ (NOR operation)
3. P3 = $a \cdot P2$ (AND operation)
4. f = $b + P3$ (OR operation)

Considering the gates propagate the values from inputs on the left side to the output on the right side, the resulting logic diagram is presented in Figure 2. 2.

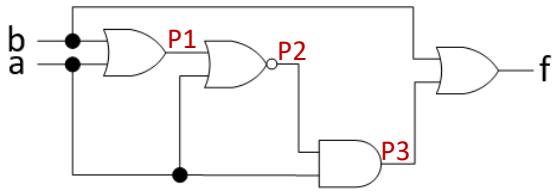


Figure 2. 2 The logic diagram for $f = b + a \cdot \overline{a + (b + a)}$

2.6 Properties of boolean algebra

Two boolean functions are considered identical if they return the same result for all possible combination of input values.

Using the tables for the basic logic gates, the following properties can be demonstrated true:

- Commutative law: $a + b = b + a$; $a \cdot b = b \cdot a$
- Associative law: $(a + b) + c = a + (b + c)$; $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
- Distributive law: $a + (b \cdot c) = (a + b) \cdot (a + c)$; $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$
- Identity element: $a \cdot 1 = a$; $a + 0 = a$
    - Consequences of the identity element: $a + 1 = 1$; $a \cdot 0 = 0$

- Idempotent law: $a + a = a$; $a \cdot a = a$
- Double negation law: $\bar{\bar{a}} = a$
- XOR operation: $a \oplus b = \bar{a} \cdot b + a \cdot \bar{b}$
- XNOR operation: $a \odot b = a \cdot b + \bar{a} \cdot \bar{b}$
- De Morgan: $\overline{a + b} = \bar{a} \cdot \bar{b}$; $\overline{a \cdot b} = \bar{a} + \bar{b}$

2.7 Transforming logic gates and generating new behaviors

It is possible to generate various functionalities using the logic gates and the properties of boolean algebra. A common example is the reduction of inputs for AND, NAND, OR and NOR gates. A 4-input AND implementing expression $a \cdot b \cdot c \cdot d$ can be redesigned to implement $a \cdot b$ using any of the following properties: $a \cdot b = a \cdot 1 \cdot b \cdot 1 = a \cdot b \cdot a \cdot b = a \cdot b \cdot b \cdot b = a \cdot a \cdot a \cdot b = \cdots$, etc. In these expressions, the value 1 represents the power source (5V) and is called VCC (Voltage Common Collector). It is available on Project Navigator, in the *General* category. Several solutions implementing a 2-input AND using a 4-input AND are highlighted in Figure 2. 3. The solutions are similar for AND, and NAND.

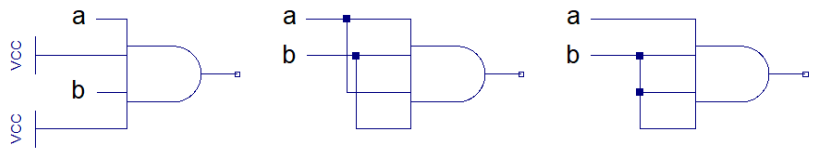


Figure 2. 3 Solutions for implementing a 2-input AND using a 4-input gate

The equivalent properties of OR operation use the constant 0, which is the ground GND (0V) of the circuit: $a + b = a + 0 + b + 0 = a + b + a + b = a + b + b + b = a + a + a + b = \cdots$, etc. Several solutions for input reduction are showcased in Figure 2. 4. They are similar in the case of the NOR gate. GND is also available in the *General* category.

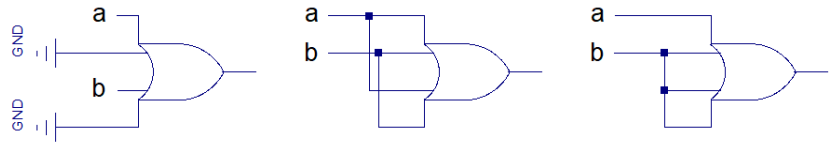


Figure 2. 4 Solutions for implementing a 2-input OR using a 4-input gate

An inverter (NOT) can be implemented from a 2-input NAND or NOR by the following expressions: $\bar{a} = \overline{a \cdot 1} = \overline{a \cdot a} = \overline{a + a} = \overline{a + 0}$ (Figure 2. 5).

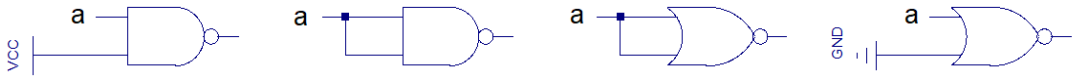


Figure 2. 5 Implementing NOT from other gates

Based on De Morgan laws, an OR can be implemented from a NAND and two inverters. An AND can be implemented from a NOR and two inverters. They respect the following properties: $a \cdot b = \overline{\bar{a} + \bar{b}}$, $a + b = \overline{\bar{a} \cdot \bar{b}}$ (Figure 2. 6).

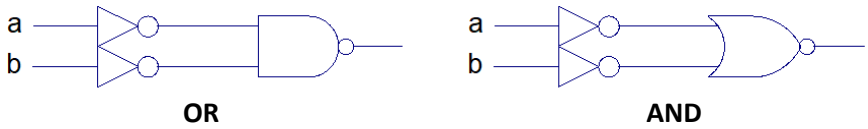
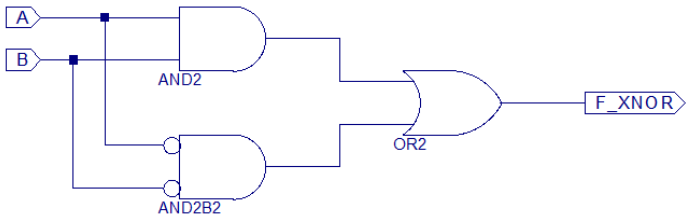


Figure 2. 6 Alternative solutions for implementing OR (left) and AND (right)

2.8 Assignments

1. Add one diagram to the project *ttl_env*, which implements the basic logic gates NOT, NOR and XOR with two inputs. Use the board and the corresponding tables from Table 2. 1 to test their functionality.
2. Add the following logic diagram to the project. The diagram implements the expression $a \cdot b + \bar{a} \cdot \bar{b}$. Use the board and the table of XNOR to demonstrate the equality $a \odot b = a \cdot b + \bar{a} \cdot \bar{b}$.



3. For each expression below calculate the truth table, design the corresponding logic diagram, and test its behavior on the board against the truth table. Do not apply any changes to the expressions. Finalize each point before advancing to the next.

a. $\bar{a} \cdot b + \bar{c}$
b. $a \cdot c + \overline{b \cdot c}$
c. $a + \overline{b \cdot c}$
d. $\overline{a + \bar{b}} \cdot c$
e. $\overline{(a + b) \cdot \overline{a + c}}$
f. $a + \overline{b \cdot \overline{a + c}}$

3 Design and simulation of digital circuits using computer aided design software (I)

3.1 Objectives

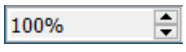
Logisim offers a dedicated environment for design and simulation of digital circuits. Its structural elements are presented, and the handling of active controls as part of the simulation environment. The design steps for several circuits and their simulation sessions are detailed as guide for using this tool and its facilities. This work is mainly focused on schematic editor and symbol editor as well as the management of the circuits in the default library attached to the project.

3.2 The Logisim application

Logisim (v. 2.7.1) offers support for logic design of digital circuits and functional simulation with values applied on input terminals, while highlighting the results on output terminals. The design of the circuit has a graphical representation called *logic diagram*, which contains the symbols of the inner circuits and their interconnections. Logisim contains a set of basic logic gates and several other circuits. Nevertheless, the user has the possibility to extend them with more complex circuits by means of hierarchy design and encapsulation in reusable libraries. The design phase is twofold:

1. The implementation of logic diagram in the Canvas of the schematic editor by defining the structure of the circuit. In this phase, Logisim allows the enabling/disabling of the simulation engine.
2. The design of the circuit symbol, which enables its integration inside other circuits. This is representative of the hierarchy design using abstraction levels of progressive complexity.

3.2.1 The schematic editor

The schematic editor is available immediately after launching Logisim. A logic diagram can be implemented in the Canvas (Figure 3. 1). The zoom in/zoom out control 

in the lower-left corner can be used to change the size of the elements in the diagram.

3.2.1.1 Elements of the logic diagram and their packaging into libraries

The Explorer Pane (also called Toolbox) in the lateral side of the Canvas shows the elements (tools/circuitry) that can be placed on the logic diagram. The elements are packed into libraries. The topmost library is called *Untitled*, same as project name. Initially, it contains the circuit called *main*, which is currently under development. Pressing ⊞ on the left side of a library name, will highlight the list of tools and circuits inside. For instance, the *Gates* library contains the basic logic gates. The other libraries contain more complex circuitry or useful tools required for the development of logic diagrams.

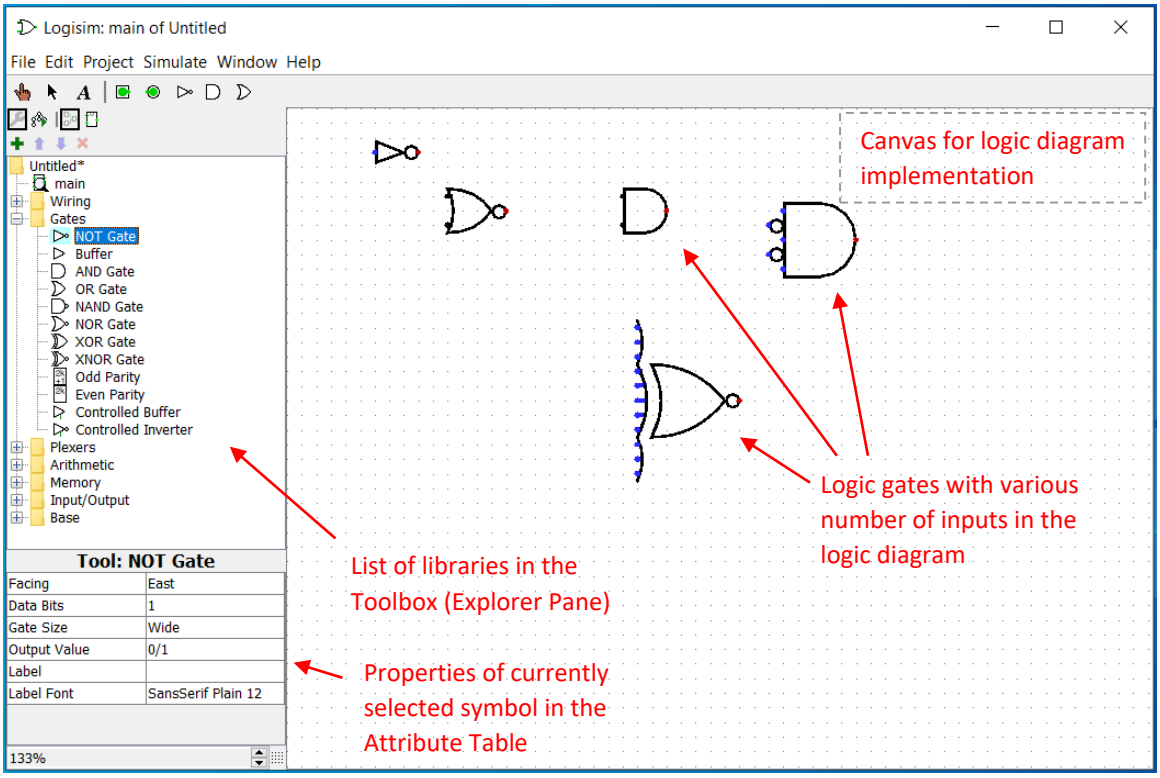


Figure 3. 1 The schematic editor in Logisim

3.2.1.2 Symbols and their properties

The symbols are graphical representations of the available circuits and tools. They can be inserted in the logic diagram by selection, followed by a click in the Canvas. Also, they can be drag-and-dropped to other positions. A symbol in the diagram can be multiplied with Copy, Paste actions available in the Edit section of the menu bar. The keyboard shortcuts for Copy-Paste are *Ctrl+C* followed by *Ctrl+V*, or simply *Ctrl+Insert*. When a symbol is selected, either in the Toolbar or the Canvas, the Explorer Pane will display its properties in the Attributes Table below the library list. For instance, by selecting the inverter gate (NOT) the list of properties will contain the following set of attributes (Figure 3. 1):

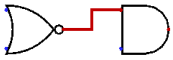
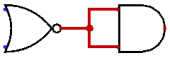
- `Facing` – orientation of the symbol (used to rotate the symbol);
- `Data Bits` – number of bits for the logic gate;
- `Gate Size` – symbol dimension in the diagram;
- `Output Value` – the output behavior: "0/1" is most common;
- `Label` – an optional text used as ID in the diagram;
- `Label Font` – font size for label.

The list of properties may change by the symbol type, but there is a common list of properties for most of them. The attribute values can be modified, which may affect the functionality of the circuit, sometimes reflected by minor changes of the symbol's appearance in the Canvas. For instance, a NAND gate with attribute `Number Of Inputs` = 2 has the symbol  ; with 5 inputs, the symbol changes its shape to  ; if inputs

2 and 4 are inverted, with attribute values `Negate 2 = Yes` and `Negate 4 = Yes`, a small circle appears nearby:  .

3.2.1.3 Connections between elements on the diagram

When placed in the diagram, inputs and outputs of the elements are represented by dots visible on the boundaries of the symbol with a different color. When the mouse is hover around an input or output pin a green circle highlights its presence  .

To connect inputs and outputs, the user should move the mouse from one pin to another, while keeping the left button pressed; a connection appears like this:  . To create a branch in a certain position of the wire the user can start a new wire from that point to an unconnected input/output:  . The starting point of a branch takes the form of a visible dot on the wire. There are two types of connections allowed on the diagram:

- direct connection from output to input;
- connection from output to wire or from wire to input. As a side note, two connected wires always fuse to one wire.

A cable with more wires can be created with the Splitter element available in the *Wiring* library. Attributes `Fan Out` and `Bit Width In` must be set to the number of wires. **Note**: Any element on the diagram (symbol or wire segment) can be removed by right-clicking it and choosing Delete in the context menu or by pressing *Del* key after selection.

3.2.1.4 Inputs with fixed value

The *Wiring* library contains the Power and Ground elements. Power generates a “1” (VCC) and Ground represents the mass “0” (GND). Figure 3. 2 highlights the usage of such inputs. Because these values propagate through the circuitry, they produce visible effects on outputs and wires: the wires carrying “1” are colored in light-green, while those carrying “0” are in dark-green.

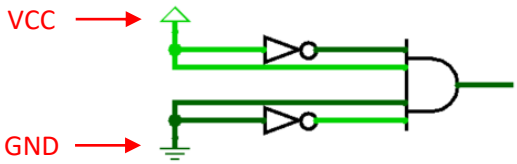


Figure 3. 2 Circuit with inputs connected to VCC and GND

3.2.1.5 The text tool

The text tool Text Tool can be accessed from the *Base* library or from the Toolbar in the upper side  . It can be used to insert a block of text in the

diagram, without any functional effect. The text has a descriptive role or can be used to highlight relevant labels.

3.2.1.6 Input and output terminals

Any logic diagram requires input and output terminals. Input terminals are required to send signals to the elements of the circuit. Output terminals capture the results. The number of input and output terminals may vary with the structure of the circuit. During simulation, the terminals highlight their values. In Logisim, the input terminal Pin is available in the *Wiring* library. In case the `Output` attribute is changed to False the terminal type switches to output, and its shape turns circular , as in Figure 3. 3. A terminal can have one or more bits defined by `Data Bits` attribute. For 1-bit terminals `Data Bits` should be set to 1. The terminal can be named using attribute `Label`.

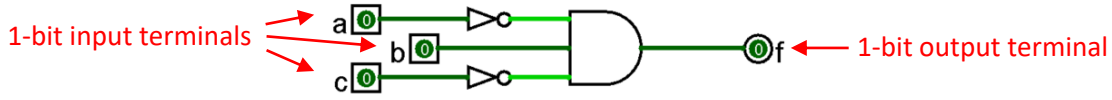


Figure 3. 3 A diagram with input and output terminals highlighting their values

3.2.1.7 Simulation in Logisim

When placing input terminals, their values propagate throughout the circuitry and generate corresponding effects. Since edit mode is enabled at startup, it is not possible to change their values. The simulation mode can be enabled by using the Poke Tool from the *Base* library or from the Toolbar . Afterwards, any click on the input terminals will toggle their values between "0" and "1". The return to edit mode is possible using either Select Tool or Edit Tool from the *Base* library or from the Toolbar: .

When the simulation engine detects errors or incomplete connections, the output terminal will signal unknown or erroneous values. All options linked to simulation can be found in the Simulate menu. Simulation Enabled or *Ctrl+E* will toggle between starting/stopping the simulation. Reset Simulation or *Ctrl+R* applies a reset on the simulation and clears all values to "0".

3.2.1.8 The Toolbar of the schematic editor

The Toolbar situated beneath the menu bar offers quick access to a subset of elements that can be placed in the diagram. The Toolbar has the following structure , but it is possible to bring appropriate changes by accessing Options in the Project menu. A new window will appear, where the Toolbar tab allows access to configuring Toolbar components.

3.2.2 The symbol editor

The circuit from the logic diagram can be encapsulated in a symbolic representation having a rectangular shape with input and output pins (representing

terminals). The symbol is saved in the project library; hence it is possible to be used inside other diagrams, with the functionality of the circuit it represents. Any communication with the circuit is possible solely through the pins of the symbol. The switch from schematic editor to symbol editor, and back, is possible using the commands Edit Circuit Appearance and Edit Circuit Layout from the Project menu. A quick switch is possible using the shortcut buttons from the Toolbar:  . For instance, the symbol generated automatically for the diagram in Figure 3. 3 is highlighted in Figure 3. 4.

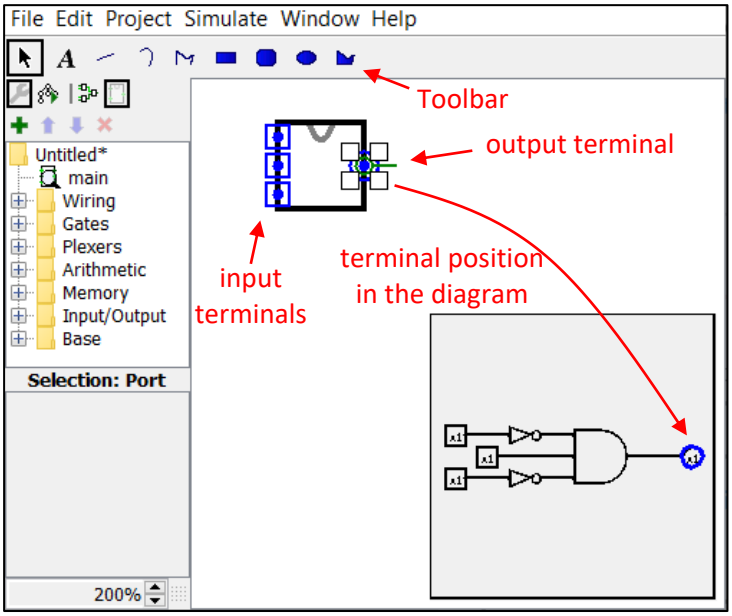


Figure 3. 4 The symbol editor associated with current diagram

The symbol and its graphical elements can be modified. The input terminals on the symbol have a rectangular blue shape, while the outputs are circular. Selecting a terminal pin from the symbol will highlight its position in the logic diagram. Optionally, the label of the terminals or the name of the diagram can be placed on the symbol using the text tool from the Toolbar of the symbol editor:  .

3.2.3 Managing the circuits in the project library

The project library can support one or several circuits. The Add circuit  command on top of the list of libraries can be used to add a new circuit. Logisim will ask its name and will generate an empty diagram. The up-down buttons  can be used to move the current circuit into another position within the library. A circuit can be deleted using the remove button  .

It is possible to switch between the circuits within a library by double-clicking their symbol in the Toolbox. The schematic editor will load the corresponding diagram in edit mode.

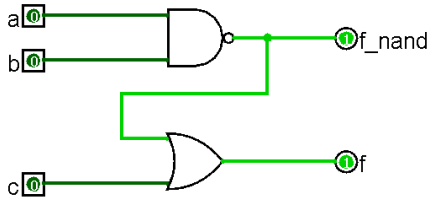
The File menu contains options to Save the project, Open a project or create a New project. A saved project will save its library in the project file.

Note: It is important to save the project as often as possible to keep the work safe.

Selecting a symbol in the current library will display its properties. Changing the attribute `Circuit Name` will rename the circuit and its symbol within the library.

3.3 Assignments

1. Create a new project in Logisim and design a circuit that contains the NOT, AND, XOR and OR gates from the *Gates* library. Excepting NOT, the gates will be configured to accept two 1-bit inputs. Add two input terminals linked to all gates and four output terminals, one for each gate. Test the logic gates using the simulator and their truth tables.
2. Add a new circuit to the project and design the logic diagram in the figure below. Calculate the truth table. Apply all possible combinations of values on a, b, c and compare the results with the truth table. Switch to symbol editor and alter the symbol generated automatically by adding the names of the terminals using the text tool from the Toolbar.



3. Add a new circuit for each of the following logic functions. Test the circuits using the simulator, by confronting the results with the corresponding truth table for all possible combinations of input values:

a) $f_1 = a + \bar{b} + c$
b) $f_2 = (a + b) \cdot (a + c)$
c) $f_3 = a + \overline{b \cdot \bar{c}}$
d) $f_4 = b \cdot (\overline{a + c} + b \cdot \bar{c})$
e) $f_5 = \overline{a \cdot \overline{\overline{b \cdot c}}}$
f) $f_6 = \bar{a} + \bar{b} + \overline{a + c}$
g) $f_7 = \overline{a \cdot b} + b \cdot \bar{c}$

4 Design and simulation of digital circuits using computer aided design software (II)

4.1 Objectives

The management of the libraries is described as means to hierarchy design in Logisim. An example of hierarchy design is detailed. The facilities offered by Logisim for the combinational analysis of a circuit are enumerated.

4.2 Library management in Logisim

An implicit library is attached by default to a Logisim project. The library may contain several circuits represented by their logic diagram. The file created by Logisim – when saving the project – contains the project library with its logic diagrams, as well as the project dependencies. This enables the possibility to load the project library into another project; its circuits will be available for reuse. The integration of circuits from current library or from other libraries is called *hierarchy design*. A library can be loaded with command Load Library > Logisim-library… from the Project menu. Logisim will request the associated project file, as in Figure 4. 1.

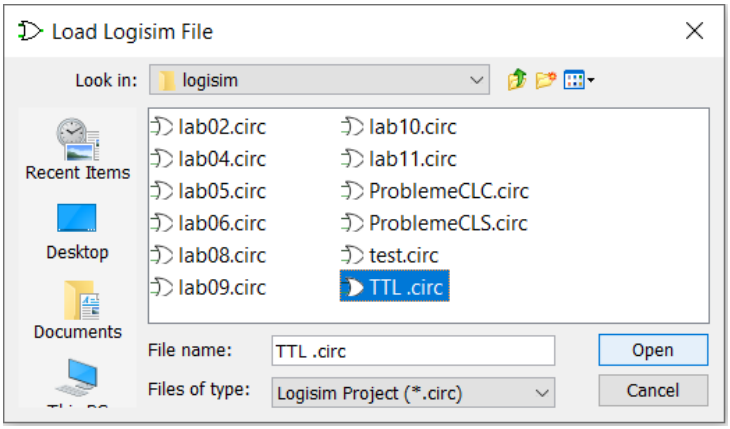


Figure 4. 1 When loading a library, Logisim asks the location of the project file implementing the library

The libraries loaded into the project appear in the Toolbox area, and their circuits – represented by symbols – are available for reuse. Figure 4. 2 highlights the access to circuits from TTL library [1]. A project can load as many libraries as necessary, which allows the possibility to integrate components from multiple sources.

A library can be removed from the project by right-clicking its name and selecting Unload Library from the context menu. Selecting Reload Library will update the library.

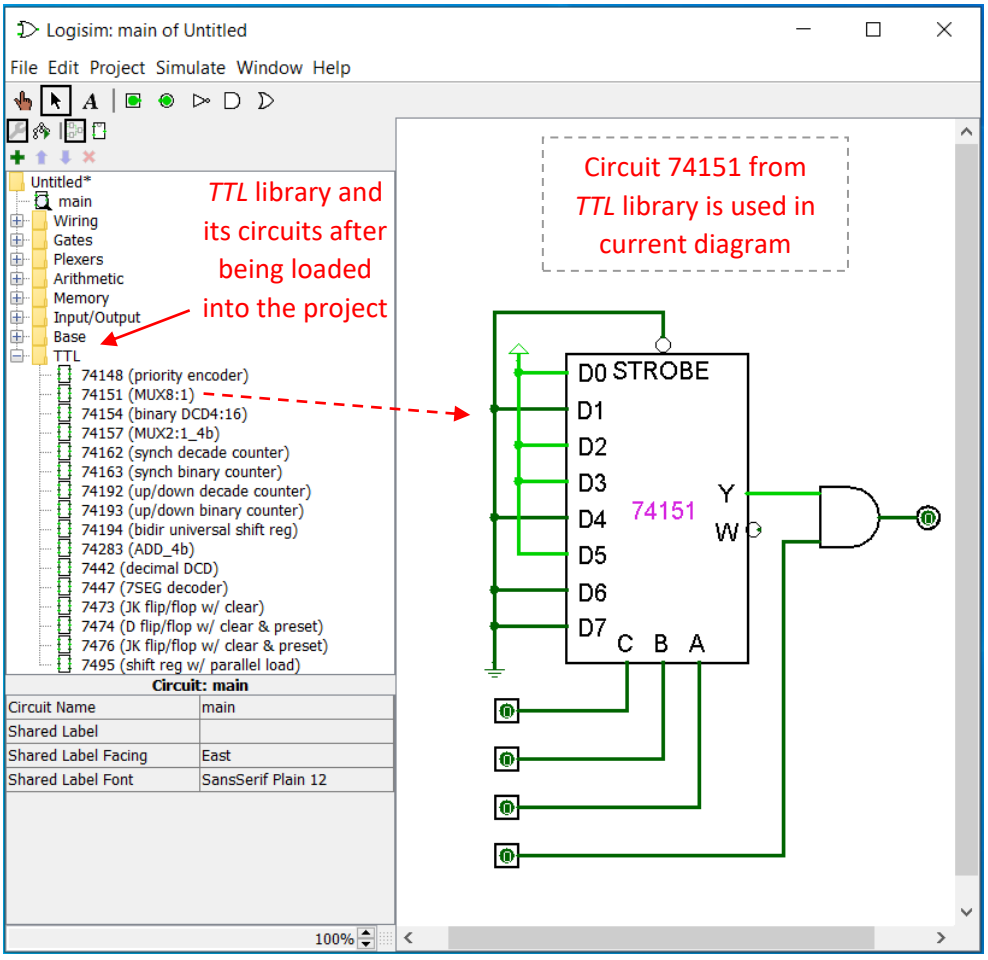


Figure 4. 2 *TTL* library appears in the Toolbox after being loaded. Its circuits can be accessed via their symbols

4.3 Circuit analysis

A logic diagram supports the following types of analysis:

- Circuit analysis;
- Circuit statistics.

Circuit analysis is available with Analyze Circuit command from Project menu. The Combinational Analysis window that appears when launching this utility is organized on several Tabs as follows (Figure 4. 3):

- Inputs – contains the list of input terminals detected on the diagram. They are identified by their label or receive one automatically, if the label is missing.
- Outputs – contains the list of detected output terminals.
- Table – highlights the truth table of the circuit outputs. A click in the cells of the output columns will toggle the values between 0, 1, and x.
- Expression – lists the boolean expressions of the outputs in Minimal Disjunctive Normal Form or in Minimal Conjunctive Normal Form.
- Minimized – lists the functions of the outputs in a Minimal Normal Form – sum of products (disjunctive form) or product of sums (conjunctive form) – as set

by the Format choice. If the diagram contains up to 4 inputs, this section will highlight the corresponding Karnaugh map. The map has an interactive behavior: any click inside the grid will toggle the value in the corresponding cell between 0, 1 and x. Consequently, the minimization adjusts automatically, with effects on the grouped cells, as well as on the resulting minimal expression. If enabled, the Set As Expression button should be pressed, to set the minimized form as primary expression of the Combinational Analysis tool.

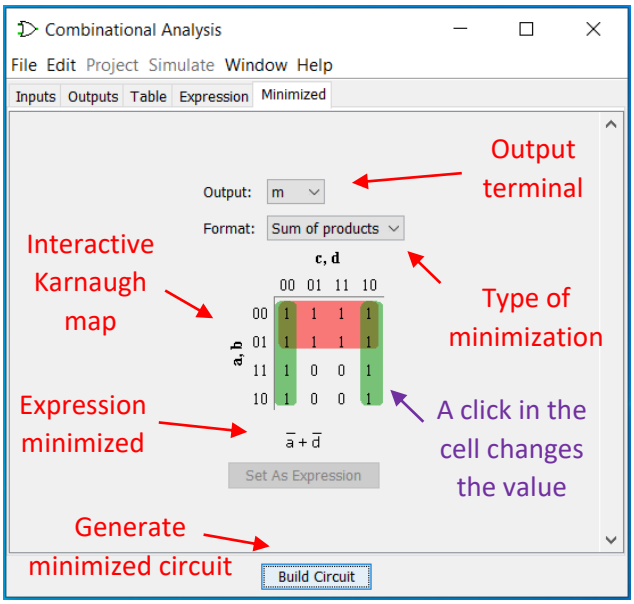


Figure 4. 3 Combinational analysis of the circuit from Figure 4. 4

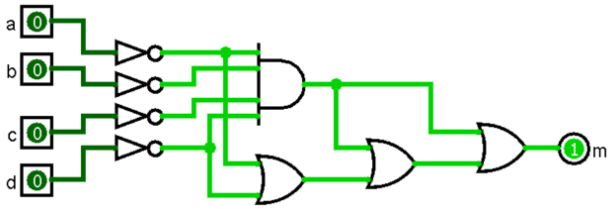


Figure 4. 4 A logic diagram with complex structure that can be simplified using the Combinational Analysis tool

Note: Circuit analysis works only for combinational circuits: which have their outputs as a function of the inputs.

The Build Circuit button is available on all tabs of the Combinational Analysis window. Pressing the button will generate a new circuit that implements the current expressions in the Combinational Analysis tool. The tool will require the name of the new circuit. The circuit will be attached to the project library. For example, the minimized version of the circuit in Figure 4. 4 is highlighted in Figure 4. 5.

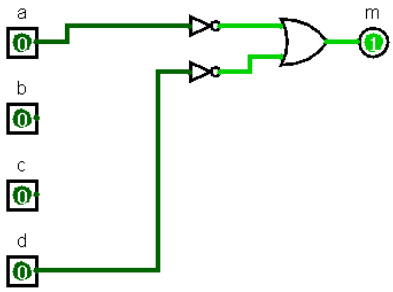


Figure 4. 5 Logic diagram after minimizing the circuit from Figure 4. 4

The Combinational Analysis tool may also be used to generate a new circuit from its analytical expression, truth table or Karnaugh map. When the diagram is empty the tool can be launched with the command Combinational Analysis from the Window menu. The input terminals may be added in the Inputs tab, and the output terminals in the Outputs tab. The logic functions corresponding to outputs can be defined analytically in the Expression tab, by truth tables in the Table tab or through the Karnaugh maps in the Minimized tab. **Note**: Karnaugh maps are available if the number of inputs is less than 5. After the output functions are defined, the circuit can be generated by pressing the Build Circuit button.

Note: The analytical expression defined in the Expression tab accepts the following operators: NOT, AND, OR and XOR. Operator NOT is represented by character ~, AND is represented by a space, OR can be defined with + and XOR with ^. Brackets may also be used if necessary. After finalizing the expression, the Enter button should be pressed, if available (Figure 4. 6).

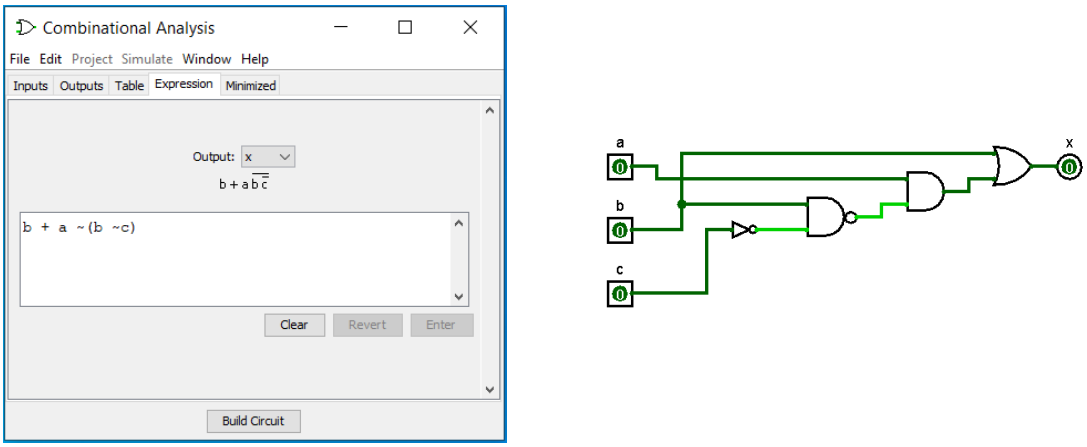
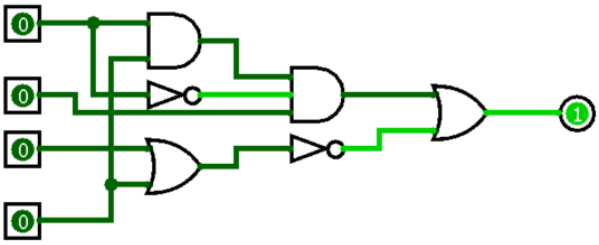


Figure 4. 6 Boolean expression of the output function defined in the Combinational Analysis tool (left). The corresponding circuit generated for this expression (right)

Circuit statistics can be visualized using the command Get Circuit Statistics from the Project menu. A new window will appear containing statistical data related to elements found in the current logic diagram.

4.4 Assignments

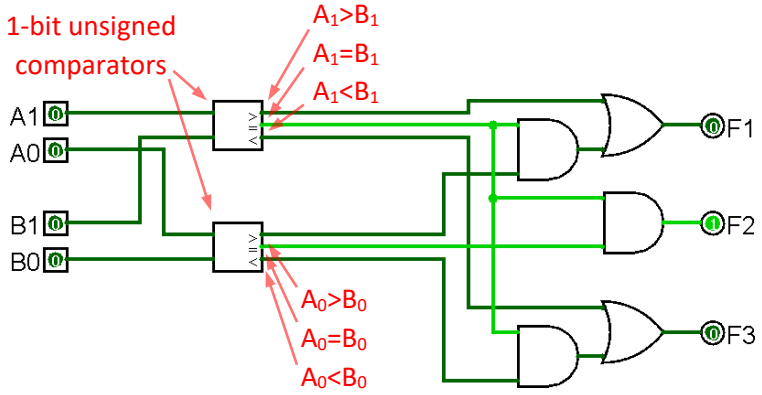
1. Create a new project in Logisim and load the *TTL* library. Analyze the list of circuits inside the library.
2. Create a new diagram for the circuit in the next figure and use the Combinational Analysis tool to generate the equivalent circuit obtained after minimization using the Minimized tab.



3. Use tabs Inputs, Outputs and Expression from the Combinational Analysis tool to implement the following logic functions in a single logic diagram.
f1 = ~a + b + ~c f2 = ~(a + b) (a + c) f3 = ~a + ~(~b c)
4. Implement the logic diagram in the next figure, which represents the design of an unsigned 2-bit comparator unit, using 1-bit unsigned comparators (with attribute `Numeric Type=Unsigned`) from the *Arithmetic* library, and additional logic gates. Test the 2-bit comparator in the simulator for all input combinations.

The diagram explained: The 2-bit numbers which are compared are A_1A_0 and B_1B_0. The diagram implements the next rules for the 3 cases possible:

- $F_1 = A_1A_0 > B_1B_0$: F_1=1 if $A_1 > B_1$ **OR** ($A_1 = B_1$ **AND** $A_0 > B_0$);
- $F_2 = A_1A_0 = B_1B_0$: F_2=1 if $A_1 = B_1$ **AND** $A_0 = B_0$;
- $F_3 = A_1A_0 < B_1B_0$: F_3=1 if $A_1 < B_1$ **OR** ($A_1 = B_1$ **AND** $A_0 < B_0$).



4.5 Bibliography

[1] Cristian-Cosmin Vancea, "TTL library for Logisim", Available online: https://drive.google.com/uc?export=download&id=1j4kRe9JXdQi6MqnqsB5nrXfX1uwoVc43

5 Combinational logic circuits – optimization and synthesis

5.1 Objectives

The two classes, characterizing completely and incompletely specified functions, are studied. Several strategies for function optimization are enumerated. Next, commonly used functions are exemplified, optimized and synthesized using logic gates. The following combinational circuits are implemented: the BCD–Excess 3 converter, the 2-bit arithmetic comparator and the summation units.

5.2 Theoretical considerations

According to their structure, circuits can be *combinational* or *sequential*. The outputs of the *combinational* circuits represent functions which are strictly dependent on input values. When implemented with logic gates, they can be recognized for their lack of backward connectivity, from output pins back to input pins. The functions implemented by the outputs can be *completely specified* or *incompletely specified*.

A function is *incompletely specified* when the output is unknown for a subset of input combinations of values. In such cases, the common notation used for the output is the symbol X or Ø. For instance, a circuit expecting 4-bit decimal digits on its input will have an undefined behavior for binary values in the range 1010-1111, because they do not represent decimal digits. This is the case for the BCD–Excess 3 converter with the behavior defined in Table 2. 3.

Table 5. 1 Truth table of the BCD–Excess 3 converter

A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0
1	0	1	0	X	X	X	X
1	0	1	1	X	X	X	X
1	1	0	0	X	X	X	X
1	1	0	1	X	X	X	X
1	1	1	0	X	X	X	X
1	1	1	1	X	X	X	X

5.2.1 Techniques for optimizing logic circuits

The optimization process aims to generate a new circuit with similar functionality, but with a simpler structure. The goal is to decrease the number of operations and the number of variables, which will decrease the necessary logic gates, their interconnections and the length of the wires. Next, we enumerate several optimization strategies commonly used when synthesizing circuits:

a) **The use of additional variables**: For instance, the set of boolean expressions $\begin{cases} a = (c \odot d) \cdot (e \oplus f) \\ b = \overline{c \odot d} + (e \oplus f) \end{cases}$ can be rewritten using the auxiliary variables $m = c \odot d$ and $n = e \oplus f$. The result will be a new set of expressions: $\begin{cases} a = m \cdot n \\ b = \overline{m} + n \end{cases}$.

b) **Applying the laws from boolean algebra**: The laws of boolean algebra are largely used in circuit optimization. Some of the most common are: $\overline{\overline{x}} = x$, $x + 1 = 1$, $x \cdot 1 = x$, $x + \overline{x} = 1$ or $x \cdot \overline{x} = 0$, etc. Considering the expression $x \cdot y \cdot z + x \cdot y \cdot \overline{z}$ and applying the distributive law we obtain the following equivalent formulations: $x \cdot y \cdot z + x \cdot y \cdot \overline{z} = x \cdot y \cdot (z + \overline{z}) = x \cdot y \cdot 1 = x \cdot y$.

c) **Adopting general scale techniques**, such as minimization based on Karnaugh maps or the Quine-McCluskey algorithm.

5.2.2 Boolean functions synthesis

The synthesis starts with a graphical representation of the implemented functions using the truth tables or the Karnaugh maps. The representation is converted to optimized boolean expressions, which are implemented using physical circuits. The following sections will describe several synthesis examples.

5.2.2.1 The BCD–Excess 3 converter

The set of outputs W, X, Y, Z in the truth Table 2. 3 describe the conversion from BCD (Binary Coded Decimal) code to Excess 3 code. Their corresponding Karnaugh maps are presented in Figure 5. 1.

Function optimization to Minimal Disjunctive Normal Form (MDNF): Looking at the Karnaugh map representation the goal is to identify groups of neighboring 1s and X cells. The groups must have rectangular shapes, and their size (number of cells) must be a power of 2. It should be considered that, for a Karnaugh map, the first and last rows as well as the first and last columns are adjacent (neighbors). While all 1s should be part of a group, the size of the identified groups must be maximized, and their number minimized. Groups solely based on X cells must be avoided. A 1 or an X cell can be part of several groups, if needed to increase their size.

A term will result for each group by applying the operator AND over the variables which are constant within the group. The variables will be inverted if their value is 0, or not, otherwise. The Minimal Disjunctive Normal Form is obtained by ORing the terms.

Function optimization to Minimal Conjunctive Normal Form (MCNF): The technique is symmetrical to Minimal Disjunctive Normal Form, because it aims to identify

groups of 0s, based on the same principles. All 0s must be grouped. The groups can contain X cells, as well. The term resulted from a group apply an OR operation solely on the constant variables within the group. The variables are inverted if their value is 1, and not inverted, otherwise. The Minimal Conjunctive Normal Form is obtained by applying AND over these terms.

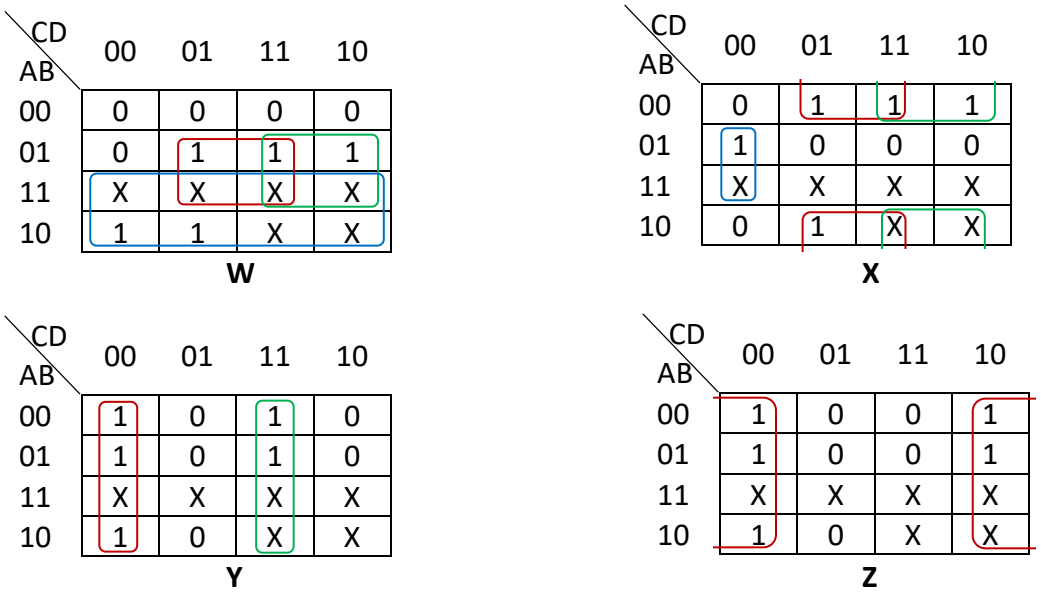


Figure 5. 1 Karnaugh maps for the BCD–Excess 3 converter outputs

The Minimal Disjunctive Normal Forms resulted from groups identified in Figure 5. 1 will reveal the following expressions:

$$\begin{aligned} W &= B \cdot D + B \cdot C + A \\ X &= \overline{B} \cdot D + \overline{B} \cdot C + B \cdot \overline{C} \cdot \overline{D} \\ Y &= \overline{C} \cdot \overline{D} + C \cdot D \\ Z &= \overline{D} \end{aligned} \quad (5.\ 1)$$

5.2.2.2 The 2-bit unsigned comparator

The 2-bit unsigned comparator have the inputs $N_1=A_1A_0$ and $N_2=B_1B_0$. The outputs are F_1, F_2 and F_3. Their functionality respects the following rules:

- F_1=1, if $N_1>N_2$ and F_1=0, otherwise;
- F_2=1, if $N_1=N_2$ and F_2=0, otherwise;
- F_3=1, if $N_1<N_2$ and F_3=0, otherwise.

The block diagram of the 2-bit unsigned comparator is presented in the following figure.

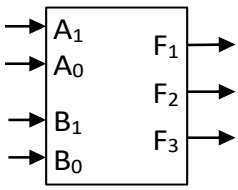


Figure 5. 2 Block diagram of the 2-bit unsigned comparator

As noticed in the previous lab work, the 2-bit unsigned comparator can be implemented using 1-bit unsigned comparators and additional logic gates. An alternative solution is fully based on logic gates. It uses the Minimal Disjunctive Normal Forms of the outputs, as functions of the 4 variables: A_1, A_0, and B_1, B_0, respectively. The Karnaugh maps can be filled according to simplified rules. For F_1, the 1s in the Karnaugh map are matched where $A_1A_0>B_1B_0$. Similarly, in the Karnaugh map for F_3, the 1s indicate $A_1A_0<B_1B_0$, while for F_2, they indicate $A_1A_0=B_1B_0$. The rest of cells are 0ed. The diagrams are highlighted below:

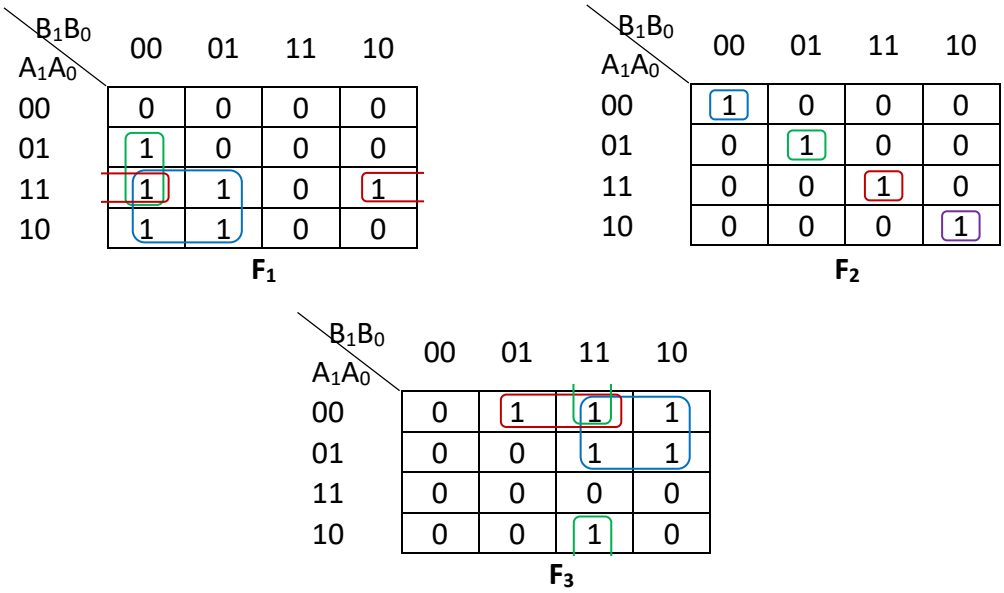


Figure 5. 3 Karnaugh maps for the 2-bit unsigned comparator outputs

Based on groups identified in the Karnaugh maps, the resulting Minimal Disjunctive Normal Forms for F_1, F_2 and F_3 are:

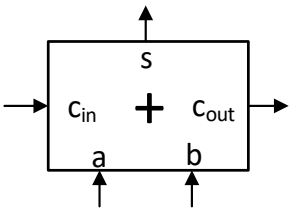
$$\begin{aligned} F_1 &= A_1 \cdot \overline{B_1} + A_0 \cdot \overline{B_1} \cdot \overline{B_0} + A_1 \cdot A_0 \cdot \overline{B_0} \\ F_2 &= \overline{A_1} \cdot \overline{A_0} \cdot \overline{B_1} \cdot \overline{B_0} + \overline{A_1} \cdot A_0 \cdot \overline{B_1} \cdot B_0 + A_1 \cdot A_0 \cdot B_1 \cdot B_0 + A_1 \cdot \overline{A_0} \cdot B_1 \cdot \overline{B_0} \\ F_3 &= \overline{A_1} \cdot B_1 + \overline{A_0} \cdot B_1 \cdot B_0 + \overline{A_1} \cdot \overline{A_0} \cdot B_0 \end{aligned} \quad (5.\ 2)$$

Using the boolean algebra laws, F_2 can be further simplified as:

$$\begin{aligned} F_2 &= \overline{A_1} \cdot \overline{B_1} \cdot (\overline{A_0} \cdot \overline{B_0} + A_0 \cdot B_0) + A_1 \cdot B_1 \cdot (\overline{A_0} \cdot \overline{B_0} + A_0 \cdot B_0) = \\ &= (\overline{A_1} \cdot \overline{B_1} + A_1 \cdot B_1) \cdot (A_0 \odot B_0) = (\mathbf{A_1} \odot \mathbf{B_1}) \cdot (\mathbf{A_0} \odot \mathbf{B_0}) \end{aligned} \quad (5.\ 3)$$

5.2.2.3 The 1-bit full adder

The 1-bit full adder is the basic element when implementing summation on inputs with *n* bits. For an *n*-bit adder, the 1-bit full adder is cascaded *n* times. The 1-bit full adder unit implements the summations of 2 input bits plus a carry input bit c_{in} (Carry In), that can be generated from inferior ranks. On the output, the 1-bit full adder unit generates the result bit s (Sum) and the carry bit c_{out} (Carry Out) that remains to be forwarded at the superior rank. The block diagram and the truth tables are highlighted in Figure 5. 4.



a	b	c_{in}	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Figure 5. 4 Block diagram and the truth table for a 1-bit full adder

From the truth table above the Karnaugh maps of the outputs can be generated:

Figure 5. 5 Karnaugh maps for the 1-bit adder outputs

The resulting Minimal Disjunctive Normal Forms for c_{out} and s are:

$$c_{out} = a \cdot c_{in} + b \cdot c_{in} + a \cdot b$$
$$s = a \cdot \bar{b} \cdot \overline{c_{in}} + \bar{a} \cdot \bar{b} \cdot c_{in} + a \cdot b \cdot c_{in} + \bar{a} \cdot b \cdot \overline{c_{in}} = \ldots = a \oplus b \oplus c_{in} \quad (5.4)$$

5.2.2.4 The 4-bit (half-byte) adder designed by cascading 1-bit full adders

A 4-bit adder, for inputs $A_{3:0}$ and $B_{3:0}$, can be implemented with four 1-bit full adders connected to allow the carry bit propagation from lower ranks to higher ranks (cascading). The carry input at the lowest rank will be connected to 0 (GND) – the result will be a 4-bit half-adder. The highest carry output bit will indicate an overflow. The logic diagram is presented next:

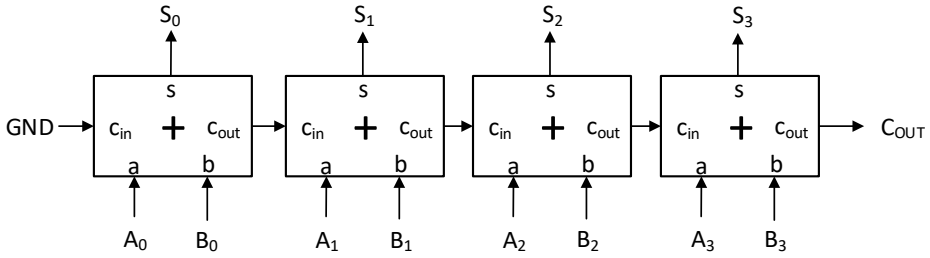


Figure 5. 6 Design of a 4-bit half-adder by cascading 1-bit full adders

Note: Similarly, 4-bit adders can be cascaded to implement summation units on multiple of 4 number of inputs bits: the carry input at the lowest rank will be connected to 0, while the rest of the carry lines will be interconnected to forward the carry bits to superior ranks.

5.2.2.5 Designing adders using minimization techniques

An alternative solution is to express the outputs of the adder as functions of input bits. From the truth tables, it is possible to generate the Karnaugh maps and extract the Minimal Disjunctive Normal Forms, which can be synthesized using logic gates. For instance, a 2-bit adder implementing the sum $A_1A_0+B_1B_0$, will output the sum on S_1S_0, plus a carry out C_{OUT} bit. Its functionality is described in Table 5. 2. Functions S_1, S_0, and C_{OUT} can be minimized using the Karnaugh maps and the boolean algebra laws, leading to the following expressions:

Table 5. 2 A 2-bit half-adder

A_1	A_0	B_1	B_0	C_{OUT}	S_1	S_0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0

$$C_{OUT} = A_1 \cdot B_1 + A_0 \cdot B_1 \cdot B_0 + A_1 \cdot A_0 \cdot B_0$$
$$S_1 = A_1 \oplus B_1 \oplus (A_0 \cdot B_0) \quad (5.5)$$
$$S_0 = A_0 \oplus B_0$$

Note: The pros with minimization techniques (vs. cascading) is the reduced number of logic gates. The cons is the exponential number of combinations involved in the minimization process, as the number of variables increases. Since the minimization based on Karnaugh maps is limited to 4 inputs, alternative solutions should be adopted, such as the Quine-McCluskey algorithm.

5.3 Assignments

1. Calculate the Minimal Disjunctive Normal Form of $f = \sum(0,4,7,10,12,13) + \sum_{\Phi}(1,8,15)$.
2. Implement and test on the board the output W of the BCD–Excess 3 converter.
3. Implement and test on the board the output F_2 of the 2-bit unsigned comparator, using XNOR gates.
4. Implement and test on the board the 1-bit full adder. For the s (Sum) output use XOR gates.
5. Implement and test in Logisim a 4-bit half-adder using 1-bit adders (with attribute `Data Bits = 1`) from the *Arithmetic* library.
6. Implement and test in Logisim an 8421(BCD)–2421(Aiken, https://en.wikipedia.org/wiki/Aiken_code) converter. For implementation, design the truth table and generate the Minimal Disjunctive Normal Forms.

6 Basic MSI combinational circuits

6.1 Objectives

The most common circuits from the MSI (Medium Scale Integration) family are analyzed and their functionality is tested. We enumerate the demultiplexers, the multiplexers and the decoders. From the same family, we also study the priority encoder and the binary-Gray converter.

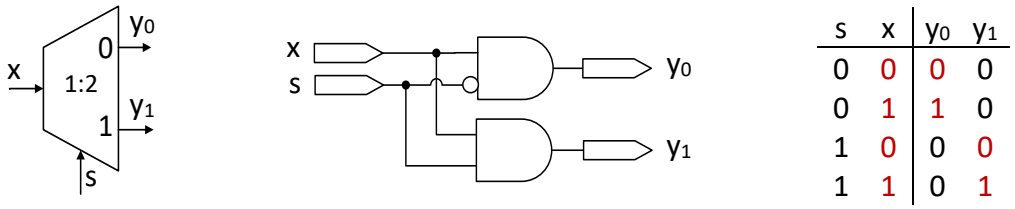
6.2 Theoretical considerations

The TTL circuits in the MSI family encompass around 50-500 transistors. Considering their complex functionality, the MSI family is often preferred when implementing digital circuits, mostly because they reduce the size of the circuit, while replacing a significant number of basic logic gates and their interconnections. In practice, a hybrid approach is preferred, which combines MSI circuits with basic logic gates. Next, we will study several MSI circuits, with simple structures, which are commonly used in hardware synthesis.

6.2.1 Demultiplexers

A 1:*n* demultiplexer (DMUX 1:*n*) has an input **x**, *n* outputs $\mathbf{y}_i$, and *m* selection signals $\mathbf{s}_k$, where $n=2^m$. The output indicated by the value on selection signals will take the value of the input, while the rest of the outputs will be 0ed. The effect can be described as: $y_i = x$, if $i = \sum_{k=0}^{m-1} s_k \times 2^k$, otherwise $y_i = 0$.

The DMUX 1:2 implementation, its symbol and the function table are highlighted in Figure 6. 1. Calculating the Minimal Disjunctive Normal Form will conclude the following expressions for the outputs: $y_0 = x \cdot \bar{s}$, $y_1 = x \cdot s$.



s	x	y_0	y_1
0	0	0	0
0	1	1	0
1	0	0	0
1	1	0	1

Figure 6. 1 DMUX 1:2 symbol (left), its implementation (middle) and the function table (right)

Demultiplexers with a higher number of inputs can be synthesized using basic logic gates or by cascading smaller demultiplexers. For instance, a DMUX 1:4 can be implemented with three DMUX 1:2 units, cascaded on two logic levels:

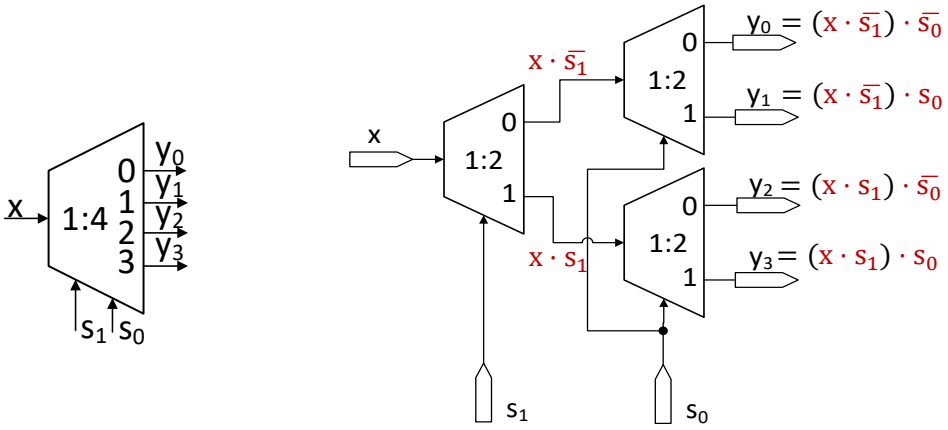


Figure 6. 2 DMUX 1:4 symbol (left) and its implementation with DMUX 1:2 units (right)

Note: When input x = 1, the demultiplexer has the functionality of a decoder, because it enables the output indicated on the selection bits.

There are demultiplexers with a higher number of inputs such as DMUX 1:8, 1:16, 1:32, etc.

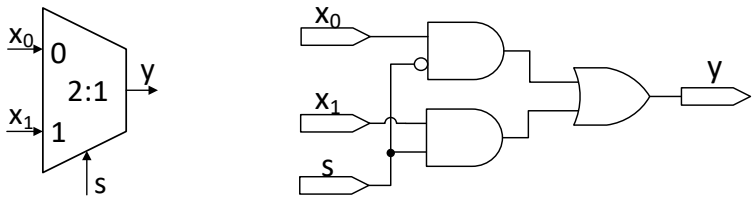
In Logisim, the demultiplexers are available in the *Plexers* library. Their size can be set using the `Select Bits` attribute, which refers to the number of selection inputs. The width of data input and data output signals can be set to 1 bit by having attribute `Data Bits = 1`. Optionally, the demultiplexer can have an additional enable input, which can be removed by setting `Include Enable = No`.

In Project Navigator, 1:4, 1:8 and 1:16 demultiplexers can be implemented using the decoders D2_4E, D3_8E, and D4_16E respectively, which are available in the *Decoder* category. The E (Enable) input of the decoder can be used as data input of the demultiplexer. The data inputs A_i can be used as selection bits. A 1:2 demultiplexer is available in the local library of the *ttl_env* project. Its name is D1_2.

6.2.2 Multiplexers

A *n*:1 multiplexer (MUX *n*:1) has *n* inputs $\mathbf{x}_i$, an output **y**, and *m* selection signals $\mathbf{s}_k$, where $n=2^m$. The output will get the value of the input indicated by the selection bits. Multiplexers are useful when one signal should be selected out of several inputs. Its effect can be expressed as: $y = x_i$, where $i = \sum_{k=0}^{m-1} s_k \times 2^k$.

The 2:1 multiplexer highlighted in Figure 6. 3 selects the input indicated by the signal s. Using the function table and calculating the Minimal Disjunctive Normal Form, the resulting expression for the output signal becomes: $y = x_0 \cdot \bar{s} + x_1 \cdot s$.



s	x_0	x_1	y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Figure 6. 3 The symbol of MUX 2:1 (left), its implementation (middle) and the function table (right)

Designing multiplexers with a higher number of inputs is possible through cascading smaller multiplexers, or by using basic logic gates. For instance, a MUX 4:1 can be implemented with three multiplexers MUX 2:1, cascaded over two logic levels:

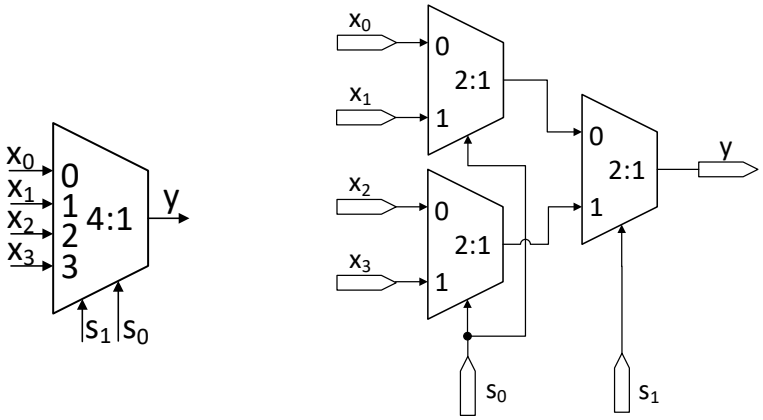


Figure 6. 4 The MUX 4:1 symbol (left) and its implementation with cascaded units (right)

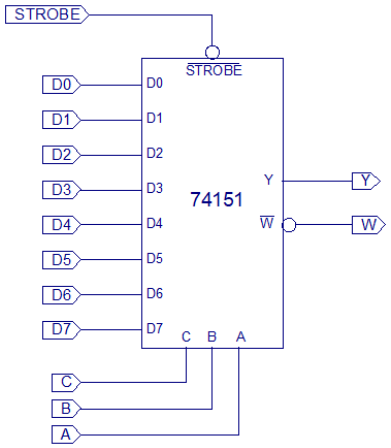
The multiplexers can be extended to a power of 2 inputs: MUX 8:1, 16:1, 32:1, etc.

In Logisim, multiplexers are available in the *Plexers* library. Attribute `Select Bits` controls the number of selection bits; `Data Bits` controls the number of bits on data inputs and output. The enable can be removed by setting `Include Enable = No`.

In Project Navigator, the multiplexers are available in the *MUX* category. Their names are M2_1, M2_1E, M4_1E and M8_1E, respectively. The suffix E indicates the presence of the enable. It must be connected to VCC, otherwise the output will be 0.

The TTL circuit 74151 in Figure 6. 5 implements a MUX 8:1 unit with inputs D_0, …, D_7, having selections A, B, C and two complementary outputs: Y and W, respectively. Output Y is active high and output W is active low. They are related by the following expression: $W = \overline{Y}$. The selection bits generate the binary number CBA_2. There is an additional enable input called STROBE (S), which is active low. If STROBE = 0 the circuit functions normally, otherwise Y = 0 and $W = \overline{Y} = 1$.

In Logisim, the TTL circuits are available inside the *TTL* library. Inside the project *ttl_env*, the TTL circuits are available in the local library. Their name is prefixed by **TTL_**, followed by their code.



STROBE	C	B	A	Y	W
1	X	X	X	0	1
0	0	0	0	D_0	$\overline{D_0}$
0	0	0	1	D_1	$\overline{D_1}$
0	0	1	0	D_2	$\overline{D_2}$
0	0	1	1	D_3	$\overline{D_3}$
0	1	0	0	D_4	$\overline{D_4}$
0	1	0	1	D_5	$\overline{D_5}$
0	1	1	0	D_6	$\overline{D_6}$
0	1	1	1	D_7	$\overline{D_7}$

Figure 6. 5 The TTL circuit 74151 implements a MUX 8:1 unit

6.2.3 Decoders

The decoder has *n* inputs and maximum 2^n outputs. The decoder activates the output indicated by the input bits. The rest of outputs are disabled. The decoders can have active high or active low outputs.

The TTL 7442 circuit in Figure 6. 6 (left) is a BCD-decimal decoder with 4 inputs (the BCD code represented by $DCBA_2$), and 10 active low outputs, counted from 0 to 9. The function table is highlighted in Table 6. 1. For input values in range 1010_2-1111_2 the outputs are disabled (set to 1).

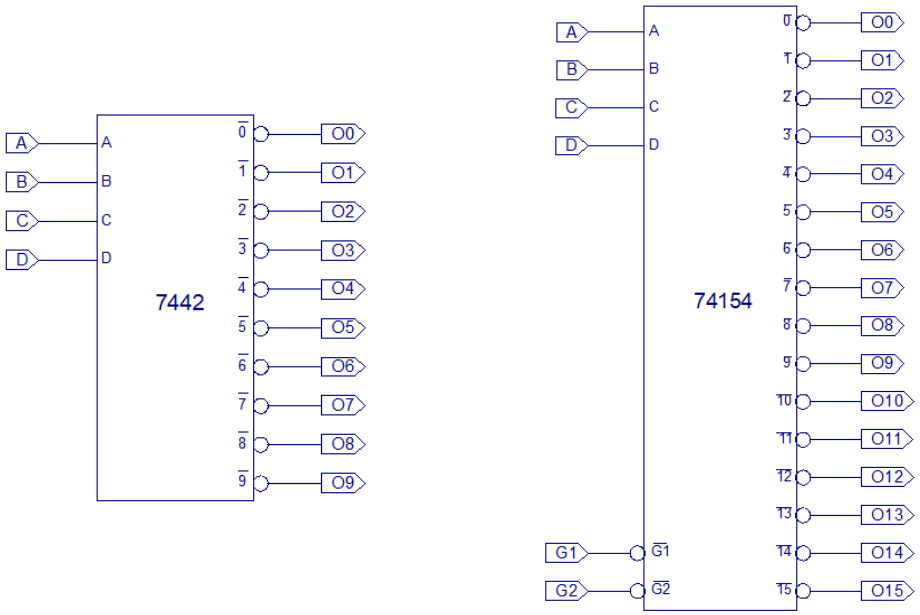


Figure 6. 6 The decimal decoder 7442 and the hexadecimal decoder 74154

The 16-bit output version is implemented by TTL 74154. This is a hexadecimal decoder DCD 4:16. Any input value in range 0-15 will enable a corresponding output (with value 0). This circuit has two additional active low enable inputs G_1 and G_2, respectively. To enable the circuit, G_1 and G_2 should be set to 0 (GND), otherwise the outputs will be disabled (value 1).

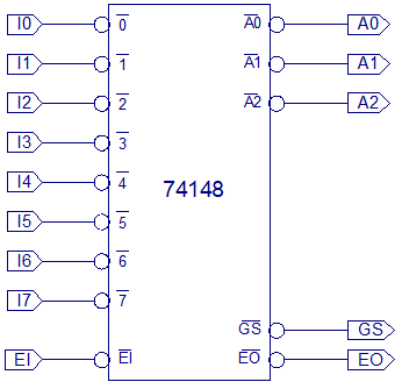
Table 6. 1 Function table of the decimal decoder 7442

BCD				Decimal									
D	C	B	A	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	1	1	1	1	1	1	1	1	1
0	0	0	1	1	0	1	1	1	1	1	1	1	1
0	0	1	0	1	1	0	1	1	1	1	1	1	1
0	0	1	1	1	1	1	0	1	1	1	1	1	1
0	1	0	0	1	1	1	1	0	1	1	1	1	1
0	1	0	1	1	1	1	1	1	0	1	1	1	1
0	1	1	0	1	1	1	1	1	1	0	1	1	1
0	1	1	1	1	1	1	1	1	1	1	0	1	1
1	0	0	0	1	1	1	1	1	1	1	1	0	1
1	0	0	1	1	1	1	1	1	1	1	1	1	0
1	0	1	0	1	1	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1	1	1	1	1
1	1	0	0	1	1	1	1	1	1	1	1	1	1
1	1	0	1	1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1

6.2.4 Encoders – the priority encoder

The priority encoder has 2^n inputs and n outputs. The code generated on the outputs indicates the active input with the highest priority. The priority increases with the index.

The circuit 74178 is a priority encoder with 8 inputs and 3 outputs. All terminals are active low (Figure 6. 7). Additionally, there is an active low enable input EI (0 – enable, 1 – disable) and two outputs, GS and EO, respectively. GS is enabled (GS = 0) when the output code on $A_{2:0}$ is valid. A code is invalid when the circuit is either disabled (EI = 1) or there is no active input. The EO output signals an error. It is activated (EO = 0) if the circuit is enabled and all inputs are disabled. The function table is highlighted below:



	Inputs								Outputs				
EI	0	1	2	3	4	5	6	7	A_2	A_1	A_0	GS	EO
1	X	X	X	X	X	X	X	X	1	1	1	1	1
0	1	1	1	1	1	1	1	1	1	1	1	1	0
0	X	X	X	X	X	X	X	0	0	0	0	0	1
0	X	X	X	X	X	X	0	1	0	0	1	0	1
0	X	X	X	X	X	0	1	1	0	1	0	0	1
0	X	X	X	X	0	1	1	1	0	1	1	0	1
0	X	X	X	0	1	1	1	1	1	0	0	0	1
0	X	X	0	1	1	1	1	1	1	0	1	0	1
0	X	0	1	1	1	1	1	1	1	1	0	0	1
0	0	1	1	1	1	1	1	1	1	1	1	0	1

Figure 6. 7 The priority encoder 74148 (left) and its function table (right)

6.2.5 Conversion units – the 4-bit binary-Gray converter

The converters perform the transition from one code to another. Usually, they are used for communication purposes, linking systems using different encoding schemes.

The 4-bit binary-Gray converter has 4 inputs and 4 outputs. A binary code on the inputs will generate a Gray code on the outputs (the inverse conversion is also possible). The function table is presented next:

B_3	B_2	B_1	B_0	G_3	G_2	G_1	G_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

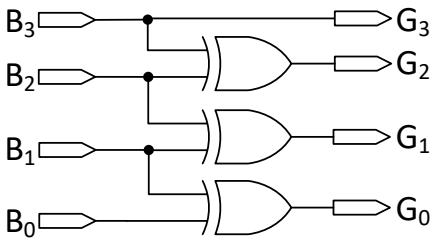


Figure 6. 8 The 4-bit binary-Gray conversion (left) and the design of the circuit (right)

Each output G_i is a separate boolean function of 4 variables (the input bits). Using minimization and boolean algebra laws, the output expressions can be reduced to: $G_3 = B_3, G_2 = B_2 \oplus B_3, G_1 = B_1 \oplus B_2, G_0 = B_0 \oplus B_1$.

6.3 Assignments

1. Implement and test on the board the 74151 multiplexer.
2. Implement and test on the board the 7442 decimal decoder.
3. Implement and test on the board the 74148 priority encoder.
4. Implement and test in Logisim a DMUX 1:2 unit, using basic logic gates.
5. Implement and test in Logisim a DMUX 1:4 unit, by cascading DMUX 1:2 units.
6. Implement and test in Logisim a MUX 4:1 unit, by cascading MUX 2:1 units.
7. Implement and test in Logisim the hexadecimal decoder 74154.
8. Implement and test in Logisim the 4-bit binary-Gray converter.
9. Implement function $f(A, B, C, D, E) = \overline{B} \cdot \overline{C} \cdot D + C \cdot \overline{D} \cdot E + B \cdot \overline{C} \cdot \overline{E} + A \cdot \overline{B} \cdot \overline{D} + A \cdot B \cdot C \cdot D$ using one 16:1 multiplexer and constants 0, 1. The variables cannot be inverted.

7 Complex MSI combinational circuits

7.1 Objectives

The structure of several complex MSI combinational circuits is studied, such as: the multiplexer with multiple data bits on datapath, the adder, the arithmetic-logic unit and the BCD-7 segment decoder. An adder-subtractor unit is implemented using only adder units, and its functionality is analyzed. Cascading is used as strategy to extend the bit width.

7.2 Theoretical considerations

Complex MSI combinational circuits are widespread in applications based on digital circuits. For instance, the most frequent operations in computational devices are arithmetic and logic, therefore arithmetic-logic units are ubiquitous. Extending the number of bits of the datapath is an implicit consequence of having most computations performed on operands with a multiple of 8 bits (1 byte). In most cases the results are displayed in base 10. The BCD-7 segment decoders represent a common solution to converting binary values to the format specific to 7-segment display units.

7.2.1 Multiplexers and demultiplexers with multiple bits on the datapath

The multiplexers forward one input from a list to a single output. The demultiplexers forward a unique input value to one of several outputs. The selection is based on the value set on the selection bits. Both multiplexers and demultiplexers can have multiple-bit datapath. For instance, the data inputs and output of a MUX 4:1 unit with 3-bit datapath consist in 3-bit data lines (buses). This can be interpreted as having three 1-bit MUX 4:1 units, sharing 2 selection bits, as in Figure 7. 1.

Figure 7. 1 The MUX 4:1 with 3-bit datapath – implementation with three 1-bit MUX 4:1 units (left) and the symbol (right)

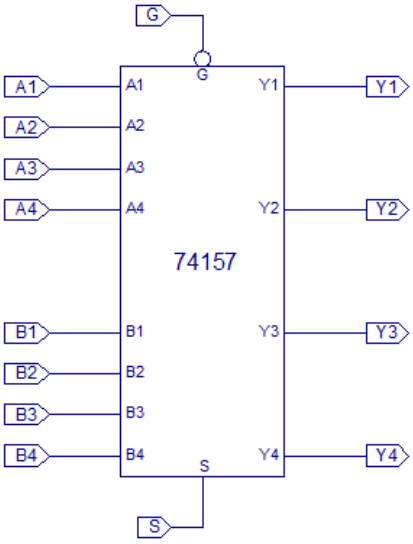
Similarly, a DMUX 1:4 with 3-bit datapath can be implemented with three 1-bit DMUX 1:4 units, sharing the selection bits, as in Figure 7. 2.

Figure 7. 2 The DMUX 1:4 unit with 3-bit datapath – implementation with three 1-bit DMUX 1:4 units (left) and the symbol (right)

The TTL 74157 implements a 4-bit MUX 2:1 unit. The symbol of the circuit is highlighted in Figure 7. 3. Data inputs $A_{4:1}$, $B_{4:1}$ and the output $Y_{4:1}$ are 4-bit wide. The selection bit S decides the input forwarded to the output:

$$Y_{4:1} = \begin{cases} A_{4:1}, \text{if } S = 0 \\ B_{4:1}, \text{if } S = 1 \end{cases} \quad (7.\ 1)$$

The G input (Strobe) enables the circuit, when G=0. If G=1, the outputs are 0ed.



Inputs				Output
G (Strobe)	S (Select)	A_i	B_i	Y_i
1	X	X	X	0
0	0	0	X	0
0	0	1	X	1
0	1	X	0	0
0	1	X	1	1

Figure 7. 3 Multiplexer 74157: the symbol (left) and the function table (right)

7.2.2 Circuits implementing arithmetic and logic operations

The basic arithmetic operations are summation and subtraction. The TTL 74283 circuit (Figure 7. 4) implements 4-bit binary summation of operands $A_{4:1}$ and $B_{4:1}$. The result is calculated on output $S_{4:1}$. The circuit has a carry input C_0 of rank 0, and a carry output C_4, of rank 4. The C_4 line signals the overflow. The carry lines can be used to extend the number of bits by cascading several 74283 units. **Note**: Setting C_0=1 is equivalent to incrementing the result by +1. The arithmetic expression implemented by 74283 is:

$$(C_4S_4S_3S_2S_1)_2 = (A_4A_3A_2A_1)_2 + (B_4B_3B_2B_1)_2 + (000C_0)_2 \quad (7.\ 2)$$

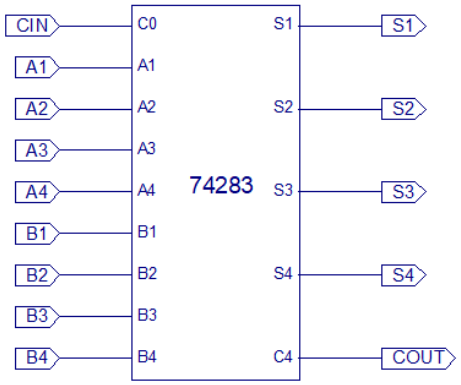


Figure 7. 4 The 4-bit adder 74283

7.2.2.1 Implementing the 4-bit adder-subtractor

In 2's Complement representation, the subtraction can be implemented by adding the first term with the 2's complement of the second term. The 2's complement is the 1's complement plus 1, meaning that: $A - B = A + \overline{\overline{B}} = A + \overline{B} + 1$. On 4-bit operands the expression becomes:

$$(A_4A_3A_2A_1)_2 - (B_4B_3B_2B_1)_2 = (A_4A_3A_2A_1)_2 + (\overline{B}_4\overline{B}_3\overline{B}_2\overline{B}_1)_2 + (0001)_2 \quad (7.3)$$

In boolean algebra, an expression ɸ follows the rules: ɸ ⊕ 0 = ɸ and ɸ ⊕ 1 = $\overline{ɸ}$. Consequently, addition and subtraction admit the next XOR-based formulation:

$$\begin{cases}(A_4A_3A_2A_1)_2 + (B_4B_3B_2B_1)_2 = (A_4A_3A_2A_1)_2 + (B_4B_3B_2B_1 \oplus 0000)_2 + (0000)_2 \\ (A_4A_3A_2A_1)_2 - (B_4B_3B_2B_1)_2 = (A_4A_3A_2A_1)_2 + (B_4B_3B_2B_1 \oplus 1111)_2 + (0001)_2\end{cases}$$

Both operations can be unified as $(C_4S_4S_3S_2S_1)_2 = (A_4A_3A_2A_1)_2 + (B_4B_3B_2B_1 \oplus \text{Sel Sel Sel Sel})_2 + (0\ 0\ 0\ \text{Sel})_2$, where **Sel=0 for addition and Sel=1 for subtraction**. It is possible to implement the expression with a 74283 unit and 4 XOR gates, as follows:

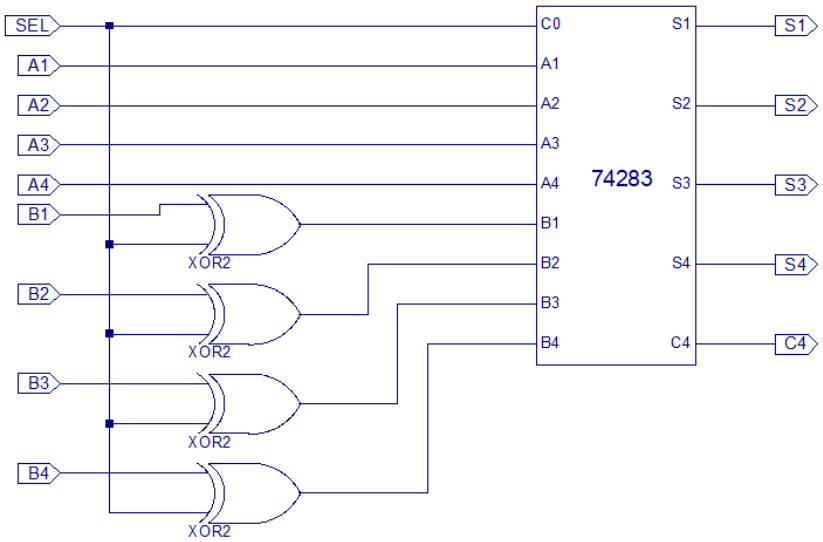


Figure 7. 5 The 4-bit adder-subtracter unit implemented with 74283

7.2.2.2 Implementing the 8-bit adder-subtractor

When extended to 8 bits the adder-subtractor expression has the following formulation:

$$(C_8S_8S_7S_6S_5S_4S_3S_2S_1)_2 = (A_8A_7A_6A_5A_4A_3A_2A_1)_2 + \\ + (B_8B_7B_6B_5B_4B_3B_2B_1 \oplus \text{Sel Sel Sel Sel Sel Sel Sel Sel})_2 + (0\ 0\ 0\ 0\ 0\ 0\ 0\ \text{Sel})_2 \quad (7.4)$$

When implemented in hardware, the expression uses two cascaded 74283 adders and 8 XOR gates. One adder will perform summation on bits ranked 1-4, having the Sel signal connected to its C_0 input. The other adder will perform summation on bits ranked 5-8, with C_0 connected to the carry output C_4 of the other unit:

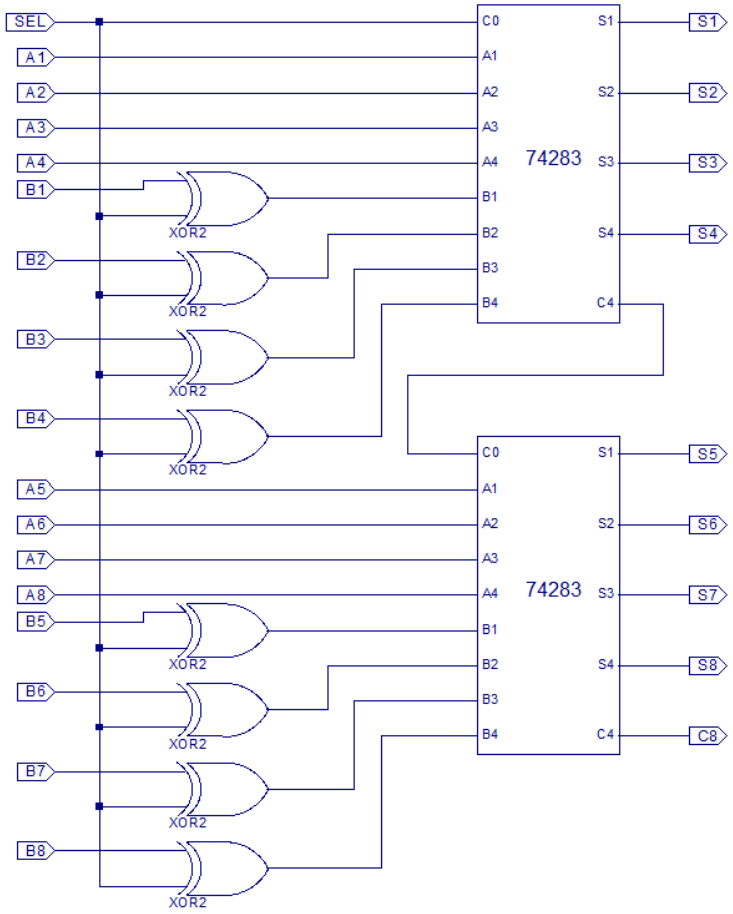
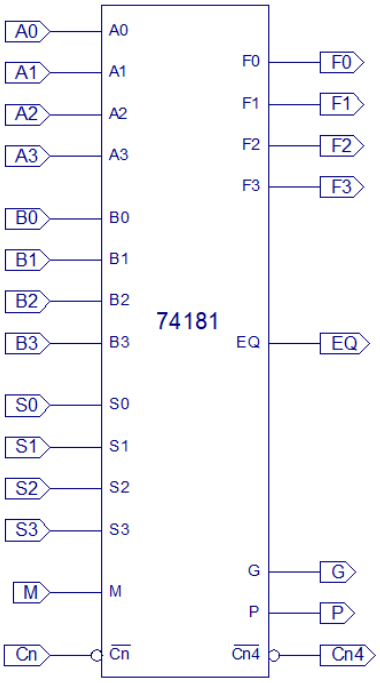


Figure 7. 6 The 8-bit adder-subtracter implemented by cascading two 74283 units

7.2.2.3 Arithmetic-logic units

The 74181 circuit (Figure 7. 7) performs 2's complement arithmetic and logic operations on 4-bit operands $A_{3:0}$ and $B_{3:0}$. The current operation is defined by the binary code set on the selection inputs $S_{3:0}$. Each code represents two operations: a bitwise logic operation is performed when input M=1, and an arithmetic operation is performed when M=0. The result is output on $F_{3:0}$. The carry input (C_n) and carry output (C_{n4}) signals are active low. They are useful when cascading 74181 units to extend the bit width. The output EQ (or A=B) tests the equality between $A_{3:0}$ and $B_{3:0}$, when performing A minus B minus 1 (when $S_{3:0}$=0110). The table below highlights the complete set of operations:



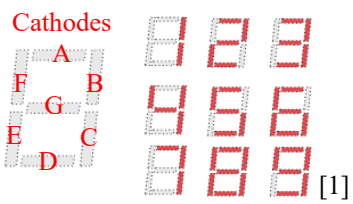
Selection inputs				Operations	
S_3	S_2	S_1	S_0	Logic M = 1	Arithmetic M = 0, C_n = 1
0	0	0	0	$\overline{A}$	A
0	0	0	1	$\overline{A+B}$	$A+B$
0	0	1	0	$\overline{A}\cdot B$	$A+\overline{B}$
0	0	1	1	0	minus 1
0	1	0	0	$\overline{A\cdot B}$	A plus $(A\cdot\overline{B})$
0	1	0	1	$\overline{B}$	$(A+B)$ plus $(A\cdot\overline{B})$
0	1	1	0	$A\oplus B$	A minus B minus 1
0	1	1	1	$A\cdot\overline{B}$	$(A\cdot B)$ minus 1
1	0	0	0	$\overline{A}+B$	A plus $(A\cdot B)$
1	0	0	1	$\overline{A\oplus B}$	A plus B
1	0	1	0	B	$(A+\overline{B})$ plus $(A\cdot B)$
1	0	1	1	$A\cdot B$	$(A\cdot B)$ minus 1
1	1	0	0	1	A plus A
1	1	0	1	$A+\overline{B}$	$(A+B)$ plus A
1	1	1	0	$A+B$	$(A+\overline{B})$ plus A
1	1	1	1	A	A minus 1

Figure 7. 7 Circuit 74181: the symbol (left) and its operations (right)
(+ = OR, · = AND, $\oplus$ = XOR, plus = summation, minus = subtraction)

7.2.3 Displaying decimal digits

The testing board has eight 7-segment display units used for displaying decimal digits (Figure 7. 8) [1]. A decimal digit appears on each display by lighting up a subset of the 7 segments (Figure 7. 9). The status of the segments is controlled by 7 signals, called *cathodes*, indexed from A to G. The cathodes are shared by all displays. Each display has an individual enable signal, called *anode*. The anodes of the displays are indexed 0-7, in right to left order. **Note**: Because cathodes are shared, all displays will highlight the same decimal digit, however there are techniques to display different digits, for decimal values with more digits. The anodes and cathodes are active low. By default, the cathodes have a value of 1 and the anodes have a value of 0, meaning that displays are active, but their segments are inactive (by default, the displays are blank).

Figure 7. 8 The 7-segment display units on the testing board



[1]

Figure 7. 9 The segments on the 7-segment display and the decimal digits configuration

Circuit 7447 is a BCD-7 segment decoder, which converts 4-bit binary codes to the cathodes' configuration associated with the corresponding decimal digit on the 7-segment display. The complete set of associations is highlighted in Figure 7. 10.

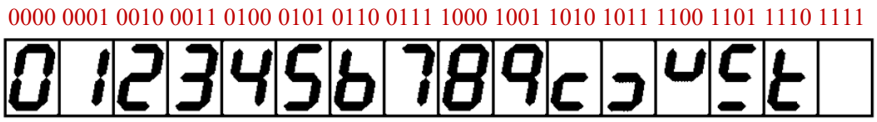


Figure 7. 10 Symbols on the 7-segment display associated to binary codes

The outputs of the decoder representing the A-G cathodes are active low (Figure 7. 11). The inputs $\overline{LT}$, $\overline{RBI}$, $\overline{BI}/\overline{RBO}$ are reserved for testing, therefore they are disabled in a normal regime, by connecting to VCC.

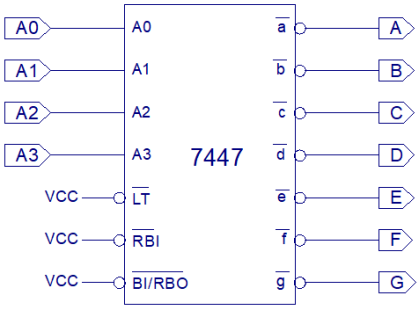


Figure 7. 11 The 7447 BCD-7 segments decoder

In Logisim, the 7-segment display is available in *Input/Output* library and has active high inputs, therefore the outputs of the 7447 decoder must be inverted with NOT gates.

In Project Navigator, the cathodes and the anodes are found in a separate section of the .ucf file:

```
## 7 segment display
NET "A" LOC=T10 | IOSTANDARD=LVCMOS33; # cat a
NET "B" LOC=R10 | IOSTANDARD=LVCMOS33; # cat b
. . .
```

7.3 Assignments

1. Implement and test on the board the 74157 multiplexer with 4-bit datapath.
2. Implement and test on the board the 74283 4-bit adder.
3. Implement and test on the board the 4-bit adder-subtractor using the 74283 unit.
4. Implement and test on the board the 7447 BCD-7 segment decoder.
5. Implement and test in Logisim the 8-bit adder using 74283 units.
6. Implement and test in Logisim the 8-bit adder-subtractor using 74283 units.
7. Implement and test in Logisim the 4-bit arithmetic-logic unit 74181.

7.4 Bibliography

[1] Digilent, "Nexys A7 Reference Manual", Available online: https://digilent.com/reference/programmable-logic/nexys-a7/reference-manual

8 Sequential logic circuits – bistable elements

8.1 Objectives

The synthesis and functionality of common bistable elements is studied. For the synthesis part, the implementation is strictly NAND-based. The functionality highlights their specific behavior according to input commands, which can be asynchronous or synchronous. The advantages and disadvantages concerning different types of clock signal synchronization (e.g. level based, or edge based) are detailed. Finally, a method for implementing a bistable with other types of bistables, and logic gates, is presented.

8.2 Theoretical considerations

The sequential logic circuits are level 1 automata characterized by internal state and outputs, which can be controlled using input commands. The simplest sequential circuits are the bistable elements, also known as *bistables*. They have two distinct states coded as 0 and 1, meaning they can store one bit of data. When powered, the state of the circuit can be maintained indefinitely or can be changed. The bistables have two complementary outputs, representing the internal state and its inverted value, respectively.

Considering the reaction from the input commands the bistables are classified as a*synchronous* and *synchronous*. The asynchronous bistables have an immediate reaction, when triggered. The synchronous bistables react depending on the *clock signal* state. The clock is a signal oscillating between 0 and 1, with a period T and a frequency *f*=1/T. The synchronous bistables can have asynchronous inputs with immediate effect on their state, when activated. These inputs have priority over synchronous commands. Usually, the asynchronous inputs are used to initialize the state of the bistable to 0 or 1.

8.2.1 The asynchronous SR latch

The asynchronous SR latch has two asynchronous inputs, one for set (S) and another for reset (R). The function table is highlighted in Table 8. 1.

Table 8. 1 Function table of the asynchronous SR latch (Q^t = current state, Q^{t+1} = next state)

S	R	Q^{t+1}
0	0	Q^t
0	1	0
1	0	1
1	1	*

The synthesis with NAND gates can be obtained from the Minimal Disjunctive Normal Form (MDNF) by inverting twice and applying De Morgan law (Figure 8. 1). **Note**: Inputs Sn and Rn are active low.

Q^t	S	R	Q^{t+1}	$\overline{Q^{t+1}}$
0	0	0	0	1
0	0	1	0	1
0	1	0	1	0
0	1	1	X	X
1	0	0	1	0
1	0	1	0	1
1	1	0	1	0
1	1	1	X	X

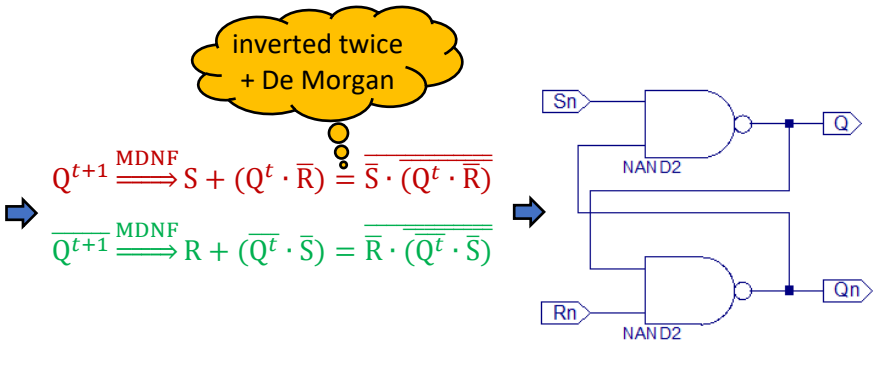
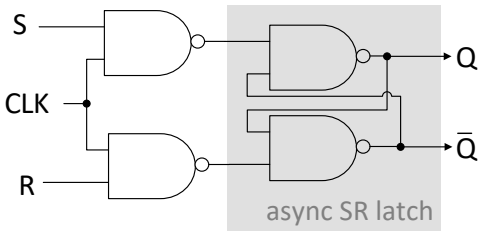


Figure 8. 1 Synthesis of the asynchronous SR latch using NAND gates (Sn=$\overline{S}$, Rn=$\overline{R}$, Qn=$\overline{Q}$)

8.2.2 The synchronous SR gated-latch

The SR gated-latch has a clock signal CLK, and two active high inputs S and R, implementing the functionality in Table 8. 1. Signals R and S are synchronized with the clock signal, meaning they can only affect the bistable state when CLK=1. This type of circuit can be implemented using NAND gates, by extending the implementation of the asynchronous SR latch according to Figure 8. 2.



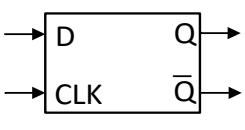
Q^t	Q^{t+1}	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

Figure 8. 2 Synthesis of the synchronous SR gated-latch using NAND gates (left) and its excitation table (right)

The *excitation table* in Figure 8. 2 can be extracted from Table 8. 1. It highlights the values on the input commands S, R, necessary for a certain behavior on the bistable state. These values are defined for all combinations of current and future states. For instance, the top row highlights that keeping the state to 0 (Q^t=0, Q^{t+1}=0) is possible when S=0 and R=X (R can be 0 or 1). For the second row, changing the state from 0 to 1 (Q^t=0, Q^{t+1}=1), requires S=0 and R=1 on inputs.

8.2.3 The D gated-latch and the D flip-flop (D ≈ data/delay)

The D gated-latch has one synchronous input called D. When CLK=1 the value on D is transferred to the state of the bistable. The symbol of the circuit and its synthesis with NAND gates are presented in Figure 8. 3.



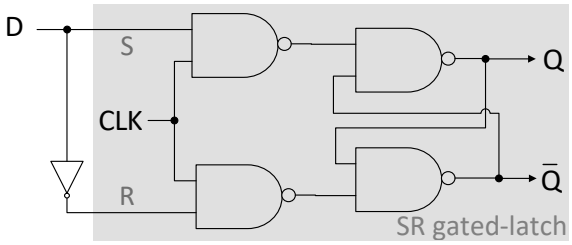


Figure 8. 3 The symbol of the D gated-latch (left) and its NAND-based synthesis (right)

Note: A D-latch can be implemented using an SR bistable with S=D and R=$\overline{D}$.

Both the function table and the excitation table (Table 8. 2) expose the expression $Q^{t+1} = D$, which describe the future state as a function of the input D. The expression is also called the *characteristic equation* of the D bistable.

Table 8. 2 The function table (left) and the excitation table (right) of the D bistable

CLK	D	Q^{t+1}
0	X	Q^t
1	0	0
1	1	1

Q^t	Q^{t+1}	D
0	0	0
0	1	1
1	0	0
1	1	1

The behavior of the bistable is highlighted in the next timing diagram. The input D affects the bistable state when CLK=1 – the trigger is on the *positive level* – and it is blocked when CLK=0. Both intervals are marked with green and red in the diagram.

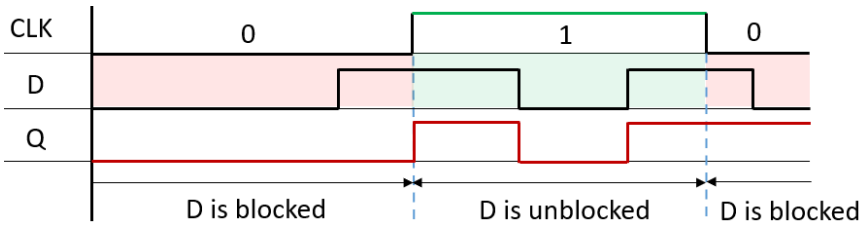
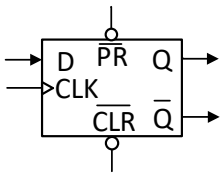


Figure 8. 4 The functionality of the D gated-latch triggered on the positive level

Note: There are D latches triggered on the *negative level*, meaning input D is effective when CLK=0.

The TTL 7474 implements the D bistable with two additional asynchronous inputs for set ($\overline{PR}$) and reset ($\overline{CLR}$). The $\overline{PR}$ (preset) and $\overline{CLR}$ (clear) are active low. Their effect is immediate, and they have priority over the input D. When $\overline{PR} = \overline{CLR} = 1$ (disabled) , D is triggered on the rising edge of the clock. The rising edge takes place when CLK shifts from 0 to 1. Bistables with edge-based trigger are also called *flip-flops*. Level-based trigger bistables are called *latches*. The behavior of the flip-flop is presented in the timing diagram from Figure 8. 5. You can notice the effect of input D, only on the rising edge. The bistable reacts immediately after the edge.



$\overline{PR}$	$\overline{CLR}$	CLK	D	Q^{t+1}
0	0	X	X	*
0	1	X	X	1
1	0	X	X	0
1	1	0	X	Q^t
1	1	↑	D	D

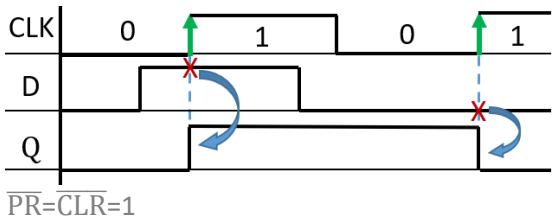


Figure 8. 5 The 7474 D flip-flop (left), the function table (middle) and its rising edge-based behavior (right)

Note: In Project Navigator, all TTL-based sequential logic circuits in the *ttl_env* project have an additional input, called CKF (ClocK-Fpga), which must be connected to the clock signal of the board, with a similar name. The clock signal of the board must be declared in the .ucf file, in the clock section, as follows:

```
## Clock signal
NET "CKF" LOC = "E3" | IOSTANDARD = "LVCMOS33";
NET "CKF" TNM_NET = sys_clk_pin;
TIMESPEC TS_sys_clk_pin = PERIOD sys_clk_pin 100 MHz HIGH 50%;
```

Important: The user clock on the test board will be generated using any of the 5 buttons. For each push, a button may generate several clock pulses due to mechanical deterioration (noise). The DECOUNCER unit (Figure 8. 6) available in the library of the *ttl_env* project, can be used to filter out false pulses. Its BT_IN input will be connected to the button terminal. The filtered signal will be available on BT_OUT. The CKF input will be connected to the clock signal of the board. The test circuit for TTL 7474 is highlighted, as follows:

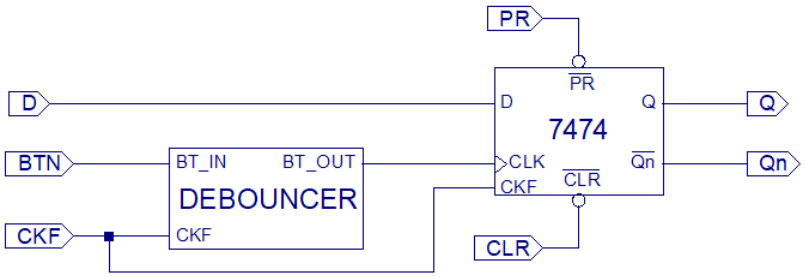


Figure 8. 6 The circuit for testing 7474 D flip-flop in Project Navigator

Note: The .ucf file contains a special section for declaring the buttons in use:

```
## Buttons
NET "BTN" LOC=N17 | IOSTANDARD=LVCMOS33; # center
```

In Logisim, the signal CKF is missing, because it is specific only to FPGA boards. The DEBOUNCER unit is unnecessary because the simulator is noise-free.

Note: In practice, there are also D flip-flops triggered on the *falling edge* (on the clock shift from 1 to 0). Several solutions exist for implementing edge-based triggered bistables from level-based triggered bistables. One solution is to connect two SR latches in a *master-slave* configuration. The D input is connected to the master, and the Q output is generated by the slave. The implementation with NAND gates is highlighted in Figure 8. 7. The master has direct connection to the clock signal, while it is inverted for the slave, leading to an effect of alternate triggering for both latches.

Functionality: A command on input D affects the master while CLK=1, but its output cannot propagate to the slave, until CLK=0. The values for the signals are highlighted in Figure 8. 8. It can be noticed how blocking intervals alternate based on the value of the CLK, leading to the edge-based effect: only the most recent command before the falling edge will trigger the output, with results visible immediately after the edge.

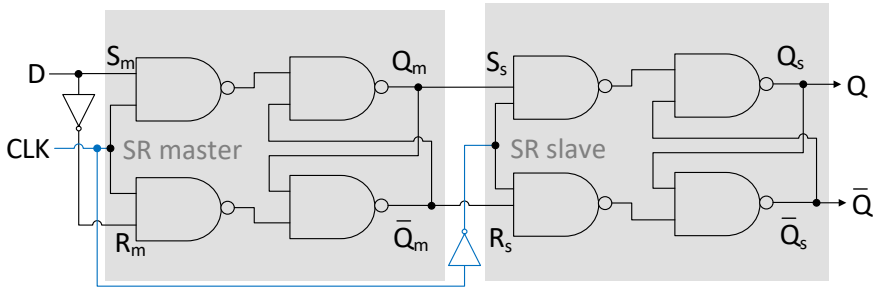


Figure 8. 7 Implementation of the master-slave D flip-flop using NAND gates

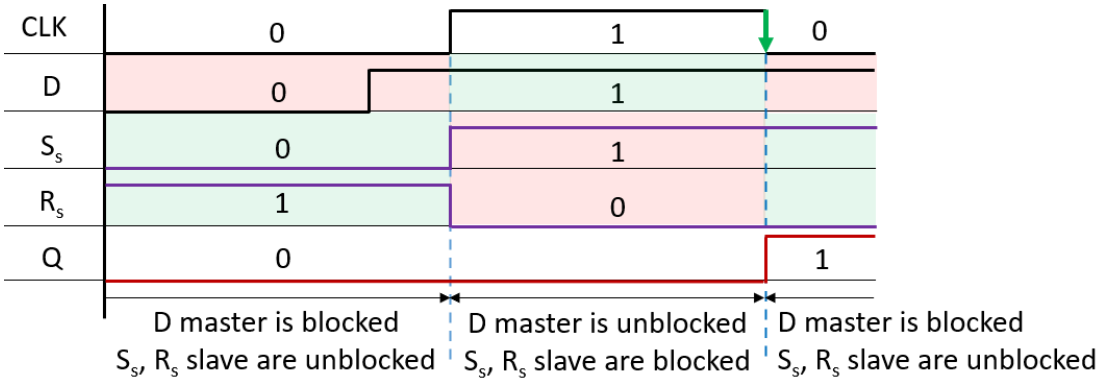


Figure 8. 8 The behavior of the signals from a master-slave D flip-flop

8.2.4 The JK gated-latch and the JK flip-flop

The JK gated-latch has two synchronous inputs J and K, replacing the S and R inputs in a SR gated-latch. The functionality is almost like an SR circuit, except when J=K=1, the bistable state inverts itself. The function and the excitation tables are presented next:

Table 8. 3 The function table (left) and the excitation table (right) of the JK gated-latch

CLK	J	K	Q^{t+1}
0	X	X	Q^t
1	0	0	Q^t
1	0	1	0
1	1	0	1
1	1	1	$\overline{Q^t}$

Q^t	Q^{t+1}	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

The NAND-based implementation extends the structure of an SR gated-latch with the following expressions for the inputs: $S = \overline{Q^t} \cdot J$, $R = Q^t \cdot K$ (Figure 8. 9 – left). Merging AND gates with NAND gates generates the 2-level logic circuit in Figure 8. 9 – right.

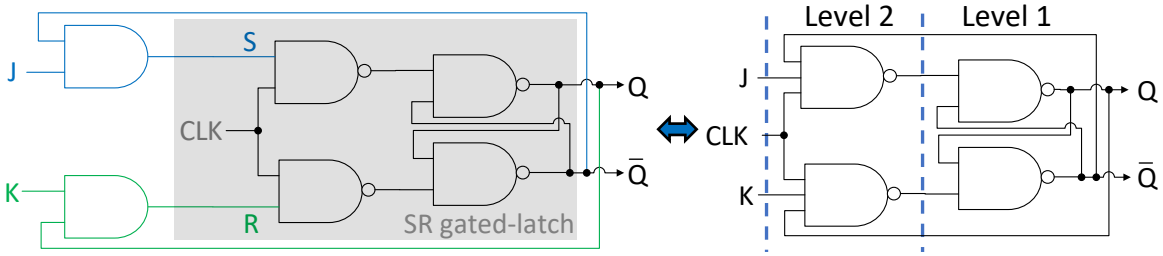


Figure 8. 9 The JK gated-latch implemented using basic logic gates

The master-slave version of the JK flip-flop below is triggered on the falling edge. The reactions are connected from the output of the slave to the input of the master:

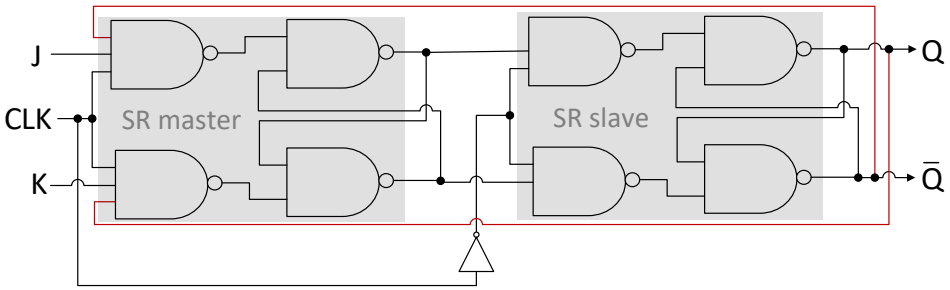


Figure 8. 10 The master-slave JK flip-flop implemented using NAND gates

The edge-based triggering advantage over level-based triggering is outlined when J=K=1, because the bistable state suffers only one change during a clock cycle (Figure 8. 11). For a level-based triggered JK latch, multiple changes are possible, depending on the duration of the clock period, which can lead to uncertain behavior.

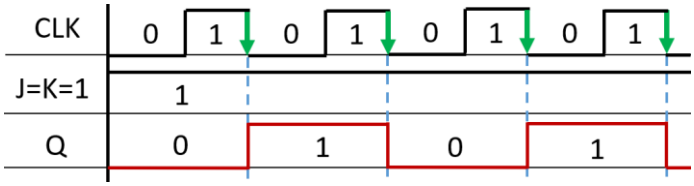


Figure 8. 11 The master-slave JK flip-flop state changes once per clock cycle

Circuits TTL 7473 and TTL 7476 implement master-slave JK flip-flops triggered on the falling edge. The 7473 has an asynchronous input $\overline{\mathrm{CLR}}$ (clear) for reset. The 7476 has both an asynchronous $\overline{\mathrm{CLR}}$ and an additional asynchronous set, called $\overline{\mathrm{PR}}$ (preset). The asynchronous inputs are active low and have priority over the synchronous inputs. The circuits are highlighted in Figure 8. 12.

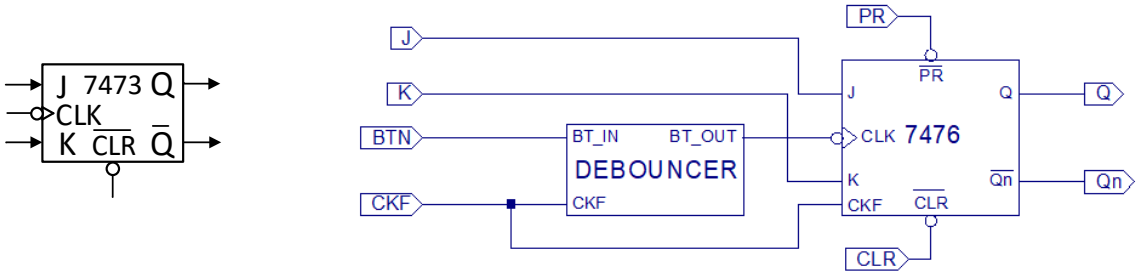


Figure 8. 12 The 7473 JK flip-flop (left) and the testing circuit for 7476 JK flip-flop (right)

8.2.5 The T gated-latch and the T flip-flop (T ≈ toggle)

The T gated-latch has one synchronous input, called T. When T=0, the state is unchanged. The state toggles when T=1. There are also flip-flop variants of the T bistable. Its falling-edge triggered variant is presented in Figure 8. 13. Considering the function table, the *characteristic equation* is: $\mathrm{Q}^{t+1} = \mathrm{T} \oplus \mathrm{Q}^{t}$. **Note**: The behavior is identical to a JK bistable having J and K connected to T. The right side of Figure 8. 13 highlights the implementation of a T flip-flop using the 7476 JK flip-flop.

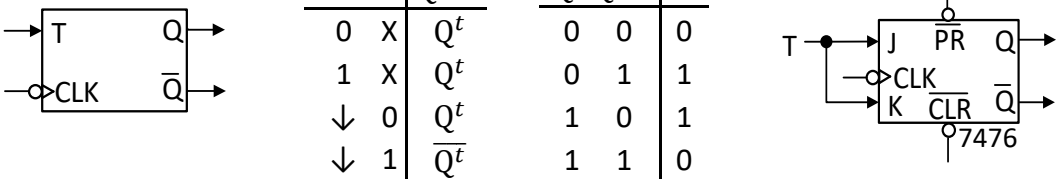


Figure 8. 13 The T flip-flop triggered on the falling-edge (left), the function and excitation tables (middle) and its implementation using the 7476 JK flip-flop (right)

8.2.6 Implementing a bistable with other types of bistables

Implementing a bistable of type A with a bistable of type B is based on both excitation tables, as follows:

1. The pairs in both tables are associated based on similar values for the current and next states (Q^t, Q^{t+1}). A new truth table is generated, which places the current state Q^t and the type A bistable inputs on the left side, while having the type B bistable inputs on the right side.
2. The expressions of the type B bistable inputs are extracted using the values from the truth table.

Several examples of bistable "conversion" are presented next:

a) Implementing D using JK

When implementing a D using a JK we use their excitation tables:

Q^t	Q^{t+1}	D
0	0	0
0	1	1
1	0	0
1	1	1

Q^t	Q^{t+1}	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

Pairing common values for (Q^t, Q^{t+1}) in both tables we obtain the truth table below, which expresses the J, K inputs as functions of current state Q^t and input D. The expressions for J and K can be extracted using the Minimal Disjunctive Normal Form from the Karnaugh maps: $J = D$ and $K = \overline{D}$. We obtain the following implementation:

Q^t	D	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

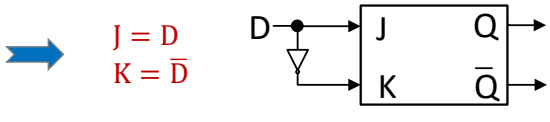


Figure 8. 14 Truth table for inputs J, K and the implementation using a bistable of type D

b) Implementing D using T

We use the excitation tables for D and T, and generate the truth table expressing T. By extracting the expression from the table we end up with the following solution:

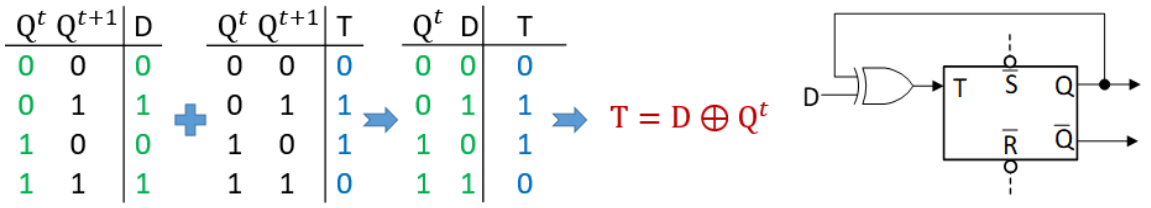


Figure 8. 15 The steps for implementing a D bistable using a T bistable

c) Implementing T using JK

Using the excitation tables for T and JK we generate the truth table expressing the inputs J and K. By extracting the expressions from the table we obtain the following circuit:

Figure 8. 16 The steps for implementing a T bistable using a JK bistable

8.3 Assignments

1. Implement and test on the board the asynchronous SR latch using NAND gates.
2. Implement on the board the 7474 D flip-flop. Test the asynchronous and synchronous commands.
3. Implement on the board the 7476 JK flip-flop. Test the asynchronous and synchronous commands.
4. Implement and test in Logisim the synchronous SR gated-latch using NAND gates.
5. Implement in Logisim a T flip-flop using the 7476 JK flip-flop. Test the asynchronous and synchronous commands.
6. Implement in Logisim the 7473 JK flip-flop. Test the asynchronous and synchronous commands.
7. Implement in Logisim a JK flip-flop using the 7474 D flip-flop. Disable the asynchronous commands, by connecting to VCC, and test the synchronous commands.

9 Sequential logic circuits – counters

9.1 Objectives

The function of the counters is defined, and their general properties are studied. The advantages and disadvantages of implementing asynchronous and synchronous counters with JK flip-flops running in *toggle* configuration are analyzed. Several integrated circuits implementing 4-bit count-up and up-down counters are presented, while focusing on their behavior and specific details.

9.2 Theoretical considerations

The counters are sequential logic circuits, which count the number of clock cycles registered on their clock input. They are implemented with flip-flop bistables. The number of flip-flops sets the number of bits used for counting. The number of states defines the *counter capacity*. In an *asynchronous* counter the flip-flops trigger at different moments of time. For *synchronous* counters, the flip-flops trigger simultaneously at moments defined by the clock signal state. Considering the state succession, counters can be *direct*, *inverse* or *reversible*. In general, direct counters count upwards, inverse counters count downwards, and reversible counters are capable of counting both ways. The counting can be *binary*, *decadic* or *modulo p*. The sequence counted by a *modulo p* counter contains *p* ($p<2^n$) *n*-bit values.

9.2.1 The asynchronous count-up binary counter

The asynchronous count-up counter has the simplest structure, being based solely on JK flip-flops running in *toggle* configuration (J=K=1). No additional circuitry is required. The counter is asynchronous because, excepting the rank 0 flip-flop, which is connected to the CLK signal, the clock input for any higher rank flip-flop is connected to the output of the inferior rank flip-flop. Figure 9. 1 highlights the implementation with JK flip-flops of a 3-bit asynchronous count-up binary counter. The counter size can be extended by adding more flip-flops. **Note**: The active-low $\overline{RST}$ signal resets the counter to 0, asynchronously (the effect takes place immediately).

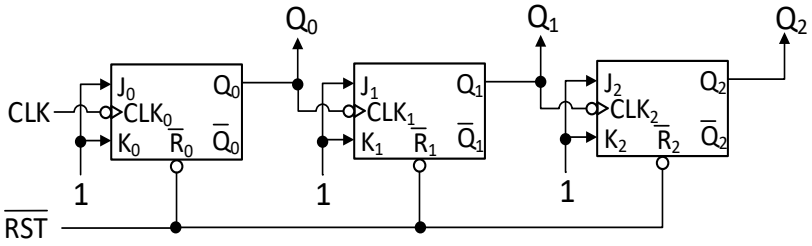


Figure 9. 1 3-bit asynchronous count-up binary counter implemented using JK flip-flops

The timing diagram in Figure 9. 2 shows the various states registered on the outputs of the asynchronous count-up binary counter, implemented with 3 JK flip-flops. The counting loops is 0-7. Each falling edge of the CLK advances the counting by +1. Also,

a falling edge on the output state of a flip-flop triggers the toggling of the superior rank flip-flop. The transition from 111_2 to 000_2 is the slowest, because all flip-flops must toggle in sequential order. The drawback: considering that CLK cycle period must be higher than the slowest transition, increasing the size of the counter limits the working frequency.

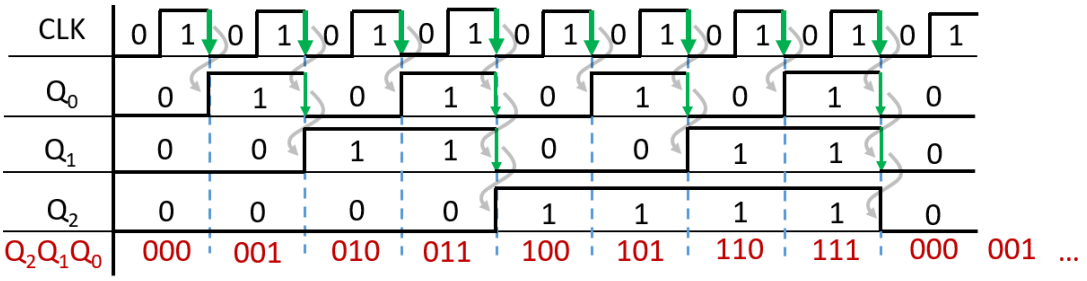


Figure 9. 2 The timing diagram for a 3-bit asynchronous count-up binary counter

Analyzing the waveforms for outputs Q_i in Figure 9. 2, it can be noticed that each flip-flop generates a clock period, double the amount of clock period generated by the inferior rank. Consequently, the JK flip-flops running in *toggle* configuration are *frequency dividers* with a factor of 2. Compared to CLK signal, Q_0, Q_1 and Q_2 apply a division factor of 2, 4, and 8, respectively.

9.2.2 The synchronous count-up serial binary counter

The frequency limitation problem encountered for asynchronous counters is partially solved by synchronous serial counters. In this case, the JK flip-flops trigger synchronously, being connected to the same CLK signal, as in Figure 9. 3. The J and K inputs are connected, however the boolean function controlling their values can be extracted by analyzing the behavior of each bit during the counting loop. Table 9. 1 highlights the values registered on the outputs for the loop 0-15.

Table 9. 1 Outputs of a 4-bit counter

Nr.	Q_3	Q_2	Q_1	Q_0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1
0	0	0	0	0

Analyzing each column Q_i, the following associations can be noticed: Q_0 toggles each clock cycle; Q_1 toggles when Q_0=1; Q_2 toggles when Q_1=1 and Q_0=1; Q_3 toggles when Q_2=1 and Q_1=1 and Q_0=1. By analogy, a flip-flop should toggle when all flip-flops of inferior rank have a value of 1. Consequently, the following boolean expression are true for inputs J_i and K_i:

$$\begin{aligned} J_0 = K_0 = 1 \\ J_1 = K_1 = Q_0 \\ J_2 = K_2 = Q_1 \cdot Q_0 \\ J_3 = K_3 = Q_2 \cdot Q_1 \cdot Q_0 \end{aligned} \quad (9.\ 1)$$

In a serial configuration, 2-input AND gates are connected serially, to implement the operations required, because they have a simple structure. The drawback with the serial connection is the increase of propagation delay with the increase of flip-flop number, when extending the counter size. The consequence is a partial limitation of the clock frequency.

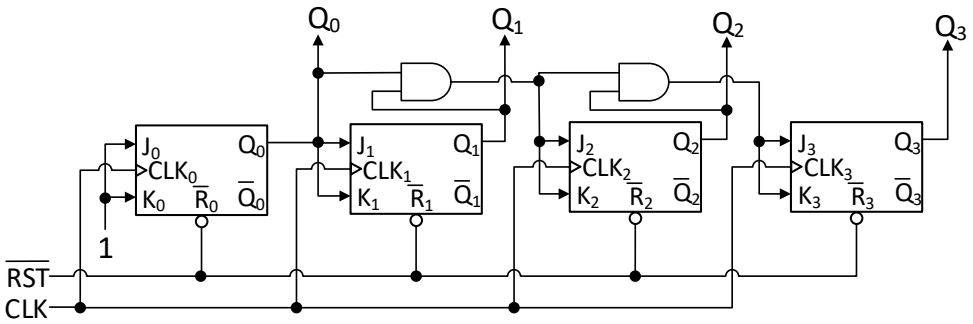


Figure 9. 3 4-bit synchronous count-up serial counter implemented using JK flip-flops

9.2.3 The synchronous count-up parallel binary counter

To remove the limitations imposed by the serial connection, a solution would be to use multiple-input AND gates, which implement the expected boolean expressions simultaneously, as in Figure 9. 4. The propagation delay is dictated by the slowest AND gate. In matters of complexity, the slowest gate has the highest number of inputs, however its time delay is significantly lower than 2-input AND gates connected serially. Consequently, the working frequency is less affected when increasing the counter size.

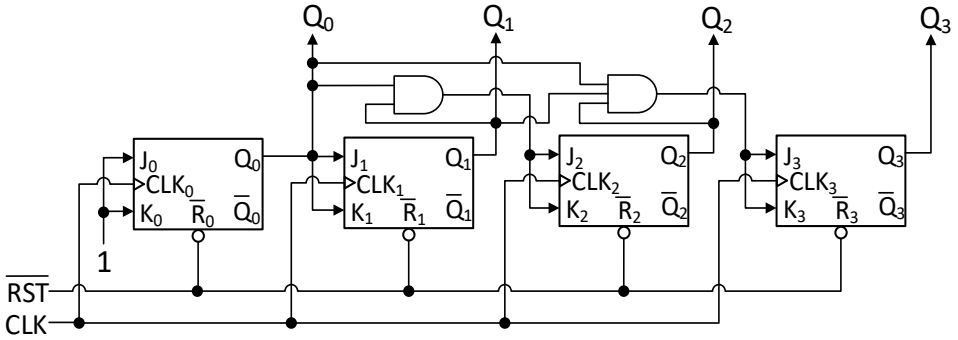
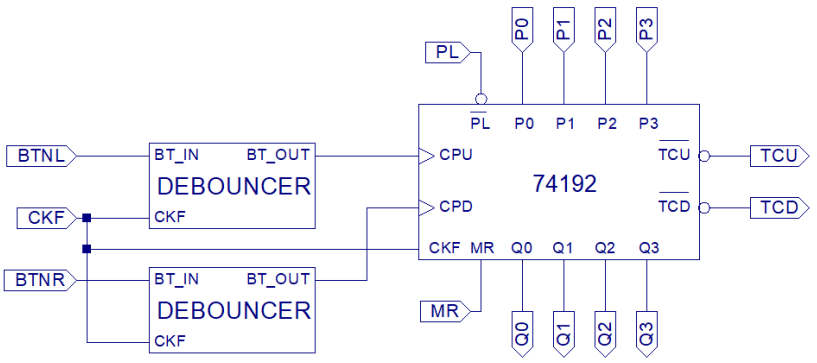


Figure 9. 4 4-bit synchronous count-up parallel counter implemented using JK flip-flops

9.2.4 The synchronous up-down decade counter 74192

Circuit TTL 74192 in Figure 9. 5 is a 4-bit up-down counter, running the counting loop 0-9, associated with the decimal digits. The counter has an asynchronous reset command MR (Master Reset). The reset sets the counter to 0. A second asynchronous command $\overline{PL}$ (Parallel Load) sets the counter to the 4-bit value registered on inputs $P_{0:3}$. The reset has priority over the load. The load command $\overline{PL}$ is active low. The current value of the counter can be read on outputs $Q_{3:0}$. The effect of the signals is presented in the function table from Figure 9. 5.

To enable the count-up, the clock signal must be connected to CP_U (Count Up), and CP_D (Count Down) must be connected to 1. To enable the count-down, the clock signal must be connected to CP_D, and CP_U must be connected to 1.



MR	$\overline{PL}$	CP_U	CP_D	$Q_{3:0}^{t+1}$
1	X	X	X	$(0000)_2$
0	0	X	X	$P_{3:0}$
0	1	↑	1	$Q_{3:0}^{t}+1$
0	1	1	↑	$Q_{3:0}^{t}-1$
0	1	1	1	$Q_{3:0}^{t}$

Figure 9. 5 Testing circuit for counter 74192 (left) and the function table (right)

At count-up, the active low output flag $\overline{TC}_U$ (Terminal Count Up) signals the upper bound (by having $\overline{TC}_U = 0$), when reaching value $Q_{3:0}$=1001. Similarly, at count-down, the active low output flag $\overline{TC}_D$ (Terminal Count Down) signals the lower bound (by having $\overline{TC}_D = 0$), when reaching value $Q_{3:0}$=0000. **Note**: The signaling takes place in the second part of the clock cycle, as shown in Figure 9. 6. The expressions for $\overline{TC}_U$ and $\overline{TC}_D$ are:

$$\overline{TC}_U = \overline{Q_3 \cdot Q_0 \cdot \overline{CP}_U}$$
$$\overline{TC}_D = \overline{\overline{Q_3} \cdot \overline{Q_2} \cdot \overline{Q_1} \cdot \overline{Q_0} \cdot \overline{CP}_D} \quad (9.\ 2)$$

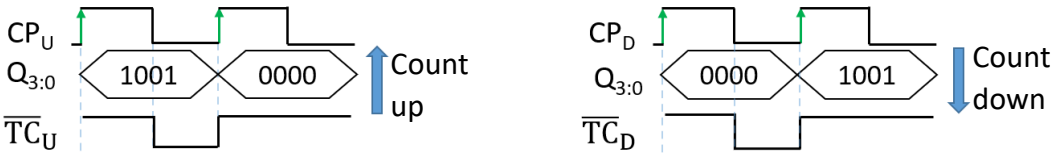
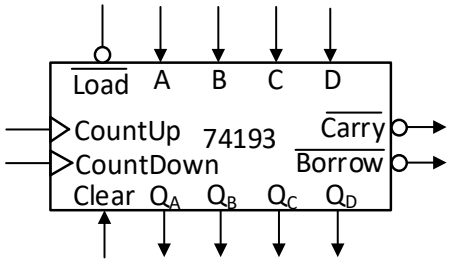


Figure 9. 6 Signaling the upper and lower bounds for 74192

The 74192 counter has *synchronous self-correction* mechanism: in case the current value is outside 0-9, the state will return to the counting loop after several clock cycles.

9.2.5 The synchronous up-down binary counter 74193

Circuit TTL 74193 in Figure 9. 7 is the binary version of counter 74192. It counts the complete 4-bit counting loop in range 0-15. The inputs and outputs have different names, but their functionality is the same as for 74192.



Clear	$\overline{Load}$	CountUp	CountDown	$Q_{D:A}^{t+1}$
1	X	X	X	$(0000)_2$
0	0	X	X	$(DCBA)_2$
0	1	↑	1	$Q_{D:A}^{t}+1$
0	1	1	↑	$Q_{D:A}^{t}-1$
0	1	1	1	$Q_{D:A}^{t}$

Figure 9. 7 The symbol for counter 74193 (left) and the function table (right)

The signals Clear and $\overline{Load}$ represent the asynchronous reset and load with the binary value $DCBA_2$, respectively. The inputs CountUp and CountDown are the clock signal inputs. When connecting the clock to one of them, the other input must be connected to

1. The current value of the counter can be read on outputs $Q_{D:A}$. According to the timing diagrams in Figure 9. 8, the output $\overline{\text{Carry}}$ signals the maximum value 15 when counting up, and the output $\overline{\text{Borrow}}$ signals the minimum value 0 when counting down. The signaling takes place only when the clock signal is 0. Accordingly, the boolean expressions for $\overline{\text{Carry}}$ and $\overline{\text{Borrow}}$ are:

$$\begin{aligned}\overline{\text{Carry}} &= \overline{Q_3 \cdot Q_2 \cdot Q_1 \cdot Q_0 \cdot \overline{\text{CountUp}}} \\ \overline{\text{Borrow}} &= \overline{\overline{Q_3} \cdot \overline{Q_2} \cdot \overline{Q_1} \cdot \overline{Q_0} \cdot \overline{\text{CountDown}}}\end{aligned} \tag{9. 3}$$

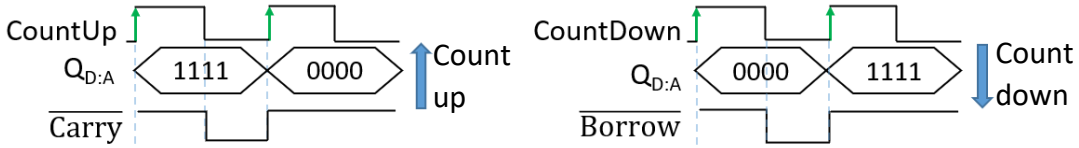
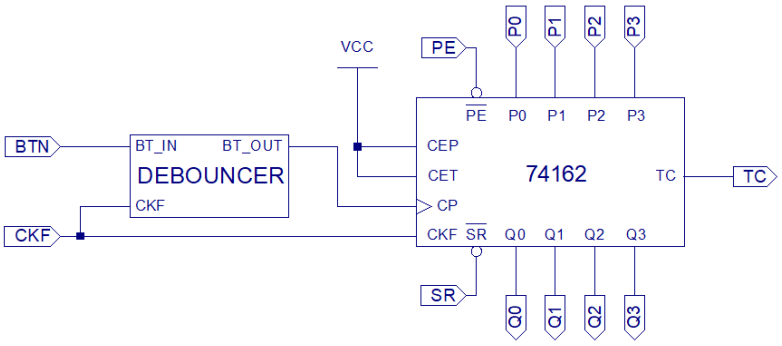


Figure 9. 8 Signaling the upper and lower bounds for 74193

9.2.6 The synchronous count-up decade counter 74162

Counter 74162 (Figure 9. 9) counts the 4-bit loop 0-9. The current value of the counter can be read on outputs $Q_{3:0}$. The clock signal is applied on input CP (Clock Pulse). The active low command $\overline{\text{SR}}$ (Synchronous Reset) resets the counter synchronously (on the rising edge of the clock), setting its value to 0. The active low command $\overline{\text{PE}}$ (Parallel Enable) loads the counter synchronously with the 4-bit value on inputs $P_{3:0}$. According to function table in Figure 9. 9, the command $\overline{\text{SR}}$ has priority over $\overline{\text{PE}}$. Both have priority over counting.



$\overline{\text{SR}}$	$\overline{\text{PE}}$	CET	CEP	CP	$Q_{3:0}^{t+1}$
0	X	X	X	↑	$(0000)_2$
1	0	X	X	↑	$P_{3:0}$
1	1	1	1	↑	$Q_{3:0}^{t}+1$
1	1	0	X	X	$Q_{3:0}^{t}$
1	1	X	0	X	$Q_{3:0}^{t}$

Figure 9. 9 Testing circuit for counter 74162 (left) and the function table (right)

To activate the counting, signals CEP and CET must be set (CEP=CET=1), and synchronous commands $\overline{\text{SR}}$, $\overline{\text{PE}}$ must be disabled ($\overline{\text{SR}} = \overline{\text{PE}} = 1$). The output TC (Terminal Count) signals when reaching the upper bound 1001_2, by enabling TC=1 (Figure 9. 10). The output TC is functional only when CET=1, according to the expression:

$$\text{TC} = Q_3 \cdot \overline{Q_2} \cdot \overline{Q_1} \cdot Q_0 \cdot \text{CET} \tag{9. 4}$$

Considering that values outside the counting loop can be loaded, the counter has *synchronous self-correction* mechanism, which brings the counter state back to the loop, after maximum 2 clock cycles.

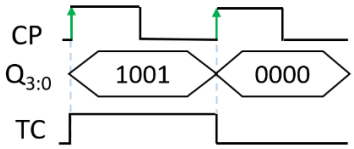
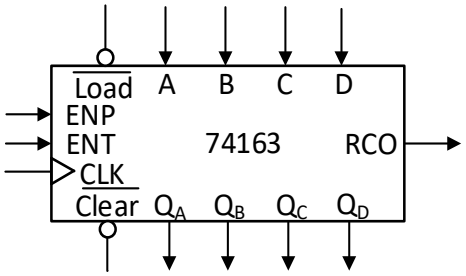


Figure 9. 10 The output TC signals the presence of the upper bound for 74162

9.2.7 The synchronous count-up binary counter 74163

The counter 74163 is similar to the decade counter 74162 presented in the previous section, except that the counting loop is 0-15. According to its symbol, highlighted in Figure 9. 11, the terminals of the counter have different names, but their functionality is preserved, as described in the table below.



$\overline{\text{Clear}}$	$\overline{\text{Load}}$	ENP	ENT	CLK	$Q_{D:A}^{t+1}$
0	X	X	X	↑	$(0000)_2$
1	0	X	X	↑	$(DCBA)_2$
1	1	1	1	↑	$Q_{D:A}^{t}+1$
1	1	0	X	X	$Q_{D:A}^{t}$
1	1	X	0	X	$Q_{D:A}^{t}$

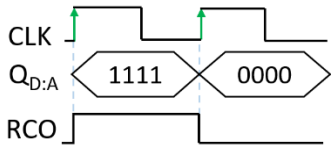


Figure 9. 11 Counter 74163, the function table and the signaling of the upper bound

The active low commands $\overline{\text{Clear}}$ and $\overline{\text{Load}}$, reset and load the binary value $DCBA_2$, respectively. Both have priority over counting. To enable the counting, ENP and ENT must be activated (ENP=ENT=1). The current value of the counter can be read on outputs $Q_{D:A}$. The output RCO (Ripple Carry Output) signals when reaching 15, the maximum value. The behavior is described in the timing diagram from Figure 9. 11. The expression for RCO is:

$$\text{RCO} = Q_D \cdot Q_C \cdot Q_B \cdot Q_A \cdot \text{ENT} \tag{9. 5}$$

9.3 Assignments

1. Implement on the board counter 74192. Test the reset and load commands, the up-down counting, the self-correction, and the signaling of the boundaries.
2. Implement on the board counter 74162. Test the reset and load commands, the counting, the self-correction, and the signaling of the upper bound.
3. Implement in Logisim counter 74193. Test the reset and load commands, the up-down counting, and the signaling of the boundaries.
4. Implement in Logisim counter 74163. Test the reset and load commands, the counting, and the signaling of the upper bound.
5. Implement in Logisim a 3-bit asynchronous count-up binary counter, using 7473 JK flip-flops. Test the counting loop.
6. Implement in Logisim a 4-bit synchronous count-up serial binary counter, using 7473 JK flip-flops. Test the counting loop.

10 Sequential logic circuits – applications based on counters

10.1 Objectives

Several methods are presented for extending the counting loop, by cascading smaller counters with reduced number of bits. The differences between binary and decade counters are analyzed, when implemented with multiple counters. The implementation of *modulo p* counters is studied, while highlighting the particularities specific to counters used.

10.2 Theoretical considerations

Digital systems with larger number of bits incorporate counters of higher capacity. The size of the counters can be extended by cascading smaller counters. For instance, using the 4-bit counters studied in the previous work, the size can be extended to any multiple of 4 bits by connecting several such units. Consequently, the counting loop is extended from 2^4 values to 2^{nx4}, where *n* represents the number of cascaded counters.

Another situation frequently met in digital systems is the reduction of the counting loop to a subset of values from the total of 2^n *n*-bit values. A counter of capacity *p*, where $p<2^n$, is called a *modulo p* counter. The *p* values can be randomly chosen, meaning there can be several counting loops defined for the same *p*. For instance, the decade counter representing the decimal digits (with the counting loop 0-9) is *modulo 10*.

10.2.1 Extending the counting loop

The increase of the counting loop, using cascaded counters, respects the following principle: the counting for the lowest ranking counter is enabled every clock cycle; any other counter from the list of cascaded counters will activate its counting once, for each end of the inferior counter counting loop.

When cascading 74193 up-down binary counters, the *carry* and *borrow* flags must be connected to the clock signals of the superior rank counter (if any), as highlighted in Figure 10. 1:

- At count up, the counter on $Q_{3:0}$ has its CountUp input connected to the clock signal, while the CountDown input is connected to 1. Consequently, the $\overline{\text{Borrow}}$ output flag will be kept inactive to 1, meaning the $Q_{7:4}$ counter will increment once for every rising edge registered on the $\overline{\text{Carry}}$ flag, specifically on the transition from 15 to 0 of the $Q_{3:0}$ counter (Figure 10. 2 – left).
- At count down, the $Q_{3:0}$ counter connects its CountDown input to the clock signal, and CountUp=1. Hence, the $\overline{\text{Carry}}$ flag will be 1 (inactive), and the rising edge on the $\overline{\text{Borrow}}$ flag, at the 0 to 15 transition (Figure 10. 2 – right), will decrement the $Q_{7:4}$ counter.

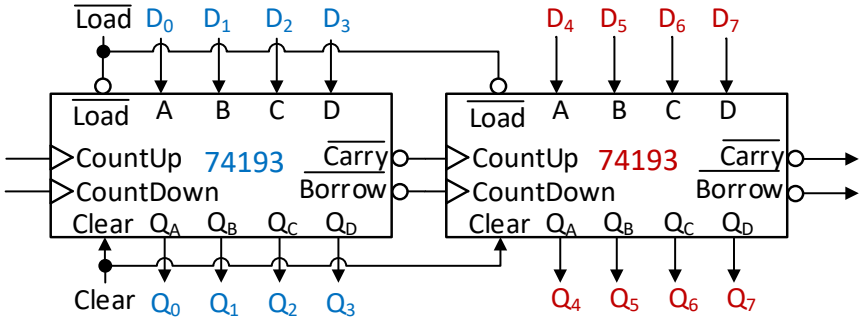


Figure 10. 1 Cascading two 74193 counters

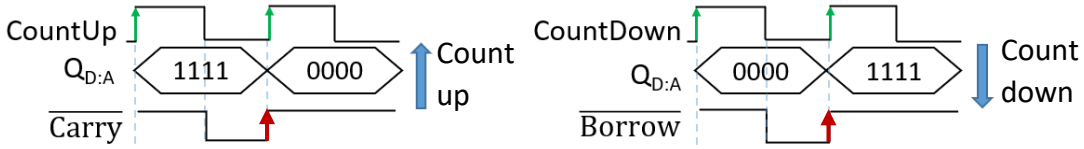


Figure 10. 2 The counter 74193 generates a rising edge on $\overline{\text{Carry}}$ or $\overline{\text{Borrow}}$, at the end of the loop

Note: The 8-bit loop on the outputs has 256 values, in range 0-255 (Figure 10. 4 – left).

When cascading up-down decade counters 74192, their interconnection follow similar rules: the output flags $\overline{\text{TC}}_\text{U}$ and $\overline{\text{TC}}_\text{D}$ are connected to clock inputs CP_U and CP_D, respectively, as in Figure 10. 3. The output values must be interpreted in base 10, after converting 4-bit pairs to corresponding decimal digits. At runtime, each 10 consecutive values on $Q_{3:0}$ correspond to a change on $Q_{7:4}$. The values registered on outputs $Q_{7:0}$ generate the loop 00-99: 00, 01, 02, … 09, 10, 11, 12, … 19, 20, 21, 22, … 99, 00, 01, 02, … (Figure 10. 4 – right).

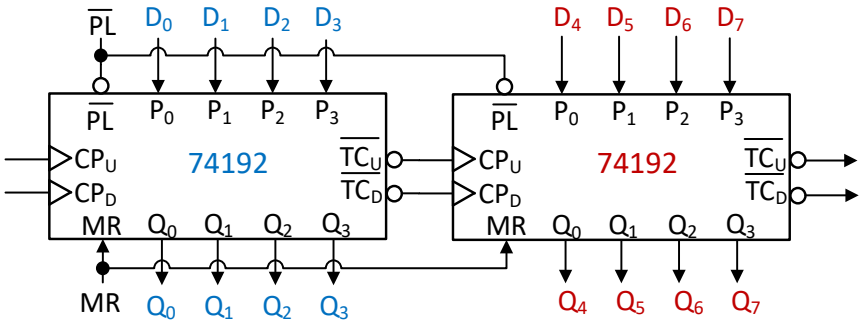


Figure 10. 3 Cascading two 74192 counters

Note: The asynchronous reset and the asynchronous load with $D_{7:0}$ are shared by all counters. Their effect is simultaneous.

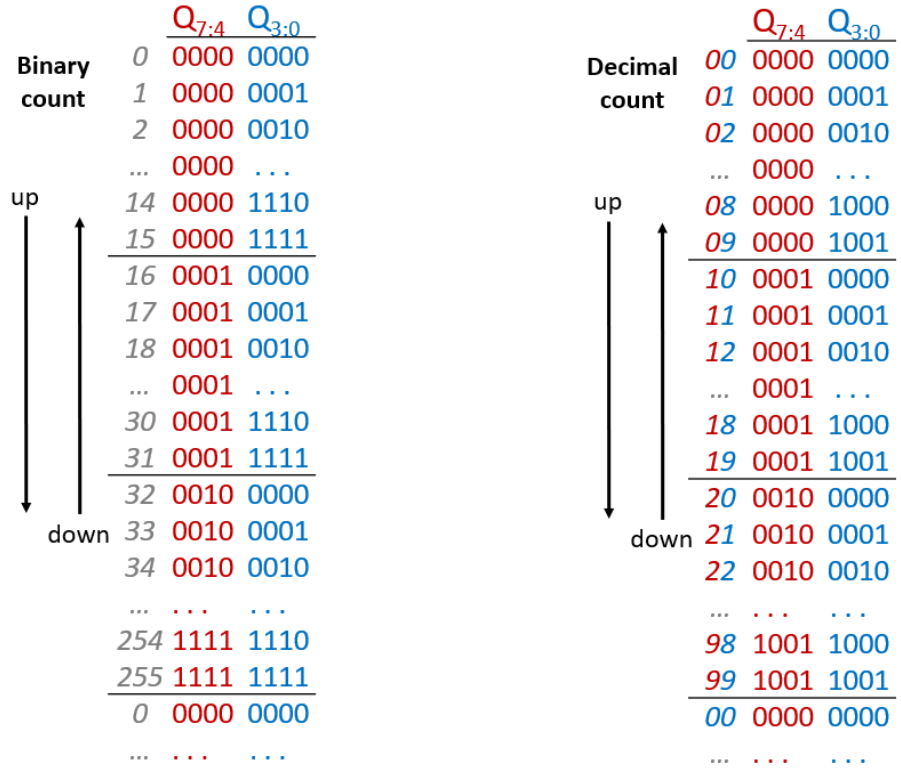


Figure 10. 4 The 8-bit counting loops: binary (left) and decimal (right)

It is also possible to cascade binary and decade counters. If the objective is a binary counter, the less significant bits $Q_{3:0}$ should be the outputs of 74193, and the rest of bits $Q_{7:0}$, should be the outputs of 74192 (Figure 10. 5). The ensemble will generate a counting loop in range 0-159 (159_{10}=10011111_2). Otherwise, if the decimal counter 74192 counts on $Q_{3:0}$ the loop will be discontinuous, because after 10 consecutive values, the following 7 values will be skipped: 0, 1, … 9, 16, 17, 18, … 25, 32 … .

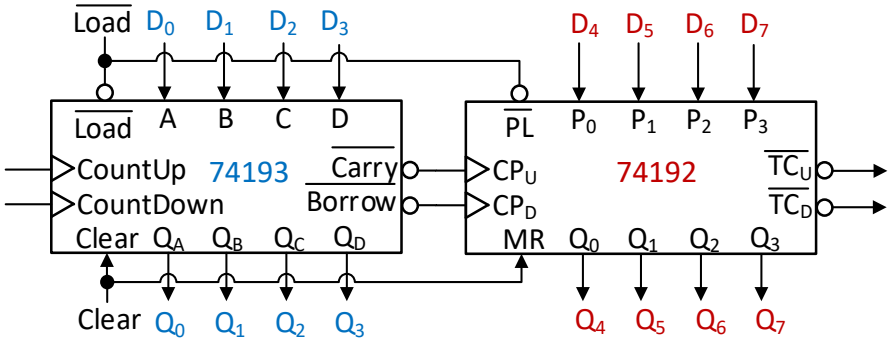


Figure 10. 5 Cascading counters 74192 and 74193 to generate a binary counter

When cascading 74163 count-up binary counters (Figure 10. 6 – left), they must share a common clock signal. The counter on $Q_{3:0}$ should always be active, by enabling inputs ENP=ENT=1. The rest of counters should have ENP=1, and ENT should be connected to the RCO output flag of the inferior counter. Since RCO activates when $Q_{3:0}$=1111, the higher rank counter increments on $Q_{3:0}$ transition to 0000, as shown in the timing diagram from Figure 10. 6. Afterwards, the RCO flag is disabled for the next 15 cycles. The resulting 8-bit binary loop is 0-255.

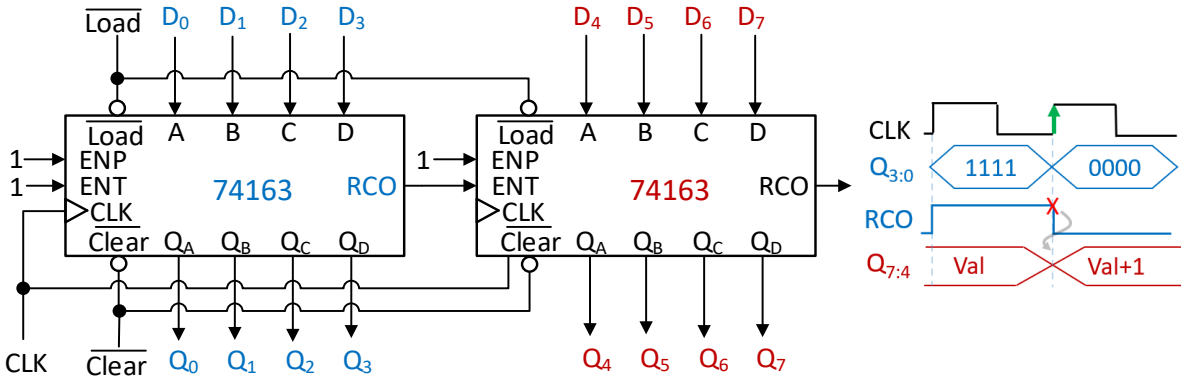


Figure 10. 6 Cascading two 74163 counters (left); the counting for the superior rank counter is enabled when the inferior rank counter ends its loop (right)

Cascading decade count-up counters 74162 (Figure 10. 7) follows the same principles as for 74163. The resulting 8-bit counting loop has 2 decimal digits, in range 00-99.

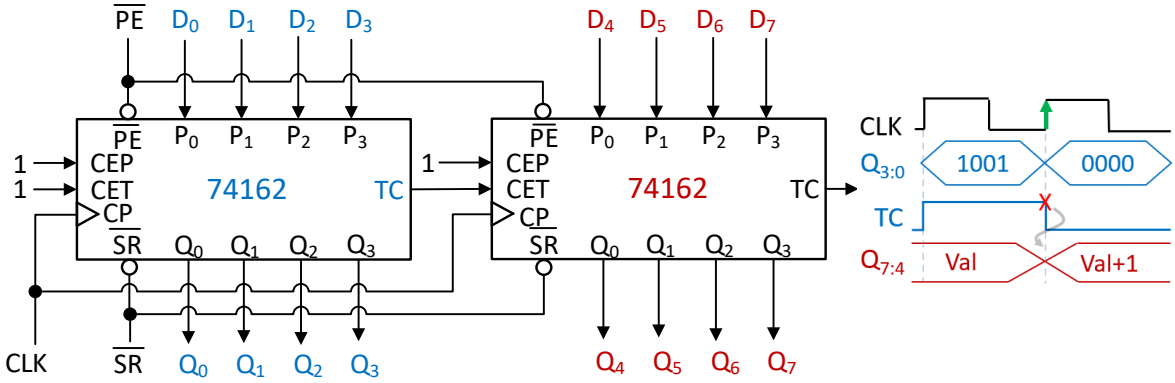


Figure 10. 7 Cascading two 74162 counters (left); the counting for the superior rank counter is enabled when the inferior rank counter ends its loop (right)

Figure 10. 8 highlights an 8-bit counter implemented using a decade counter 74162, and a binary counter 74163. To avoid loop discontinuities, the binary counter must be placed on outputs $Q_{3:0}$.

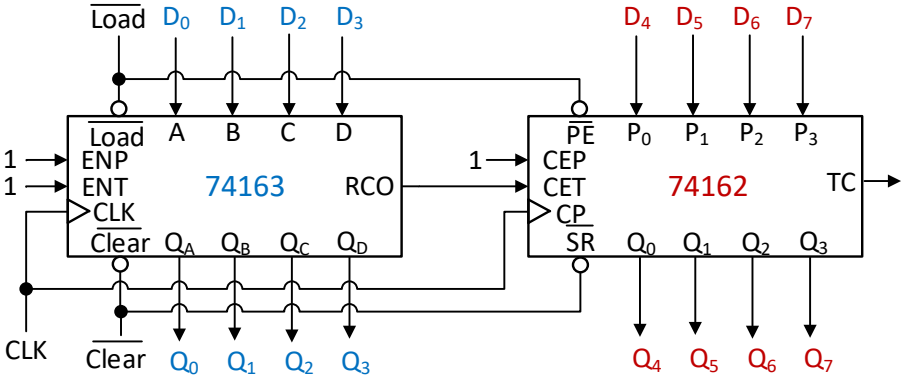


Figure 10. 8 Cascading counters 74162 and 74163 to generate a binary counter

The binary loop of the 8-bit counter in Figure 10. 8 is in range 0-159 (159_{10}=$1001\ 1111_2$).

10.2.2 *Modulo p* counters with continuous loop

The loop of a *modulo p* counter contains *p* *n*-bit values, out of the 2^n possible values. Continuous loops contain consecutive values, but the start value, the end value and the counting direction can be customized. Considering the output bits of the counter are linked to other synchronous units, only values registered (visible) at the rising edge of the clock signal are important. Other intermediate values are ignored; hence they are not considered part of the counting loop.

The strategy behind implementing *modulo p* counters using 74192, 74193, 74162 and 74163 units, is to load the start value after reaching the end value. If the start value is 0, the load can be replaced by reset. The end value can be decoded with additional logic, or using the *carry*/*borrow* flags, when possible. **Note**: The counters used when implementing *modulo p* counters should be able to count all the values in the loop. For instance, when using a decade counter, it is possible to implement only *modulo p* counters with values in range 0-9.

Note: If the counter is 0ed at startup, and this value is outside the *modulo p* loop, several clock cycles will be required before entering the counting loop.

When using 74192 and 74193 counters, the reset and load commands are asynchronous. If *Val* represents the end of loop, a reset or load will overwrite *Val* immediately, before the next rising edge. Consequently, *Val* is removed from the loop. To avoid such effect, the solution is to delay the load for one clock cycle, therefore, to detect *Val*+1 at count up or *Val*-1 at count down. Considering the start of loop is $D_{3:0}$, the implementation is highlighted in Figure 10. 9. Inputs A, B, C, D will be connected to 0 (GND) or 1 (VCC) as indicated by $D_{3:0}$. If $D_{3:0}$=7 (7_{10}=0111_2), then A=1, B=1, C=1 and D=0, respectively. **Note**: When not used, Clear must be disabled. The values of the signals when the loop is restarted, are highlighted in Figure 10. 10.

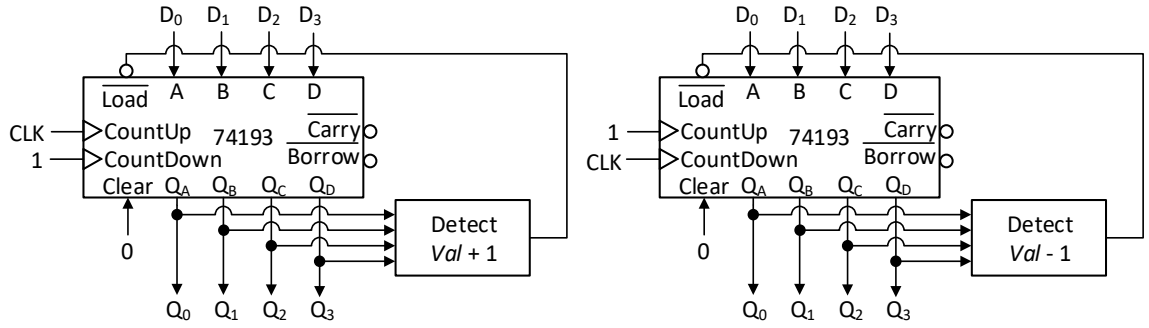


Figure 10. 9 Implementing loop $D_{3:0}$-*Val* using counter 74193; count up loop (left) and count down loop (right)

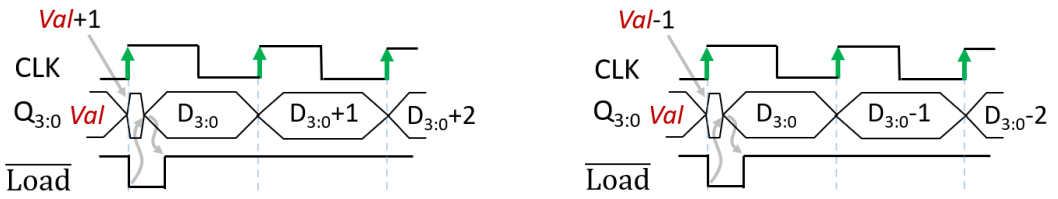


Figure 10. 10 Signal values at count up (left) and count down (right) when restarting the loop $D_{3:0}$-*Val*, implemented with 74193

The presence of a certain value on the outputs of the counter can be detected by implementing the corresponding minterm, using an AND gate, but knowing that $\overline{\text{Load}}$ is active low, the NAND gate is more appropriate. If the decoded value is 15 or 0, the $\overline{\text{Carry}}$ and $\overline{\text{Borrow}}$ flags can be used instead. For instance, when implementing a 0-14 loop counter, value 15 can be decoded using the $\overline{\text{Carry}}$ flag. Loading the start value 0 can be issued by enabling Clear and keeping $\overline{\text{Load}}$ inactive (Figure 10. 11 – left). When implementing loop 7-15, the start value 7 should be loaded when detecting 0 (0=15+1), as in Figure 10. 11 – right.

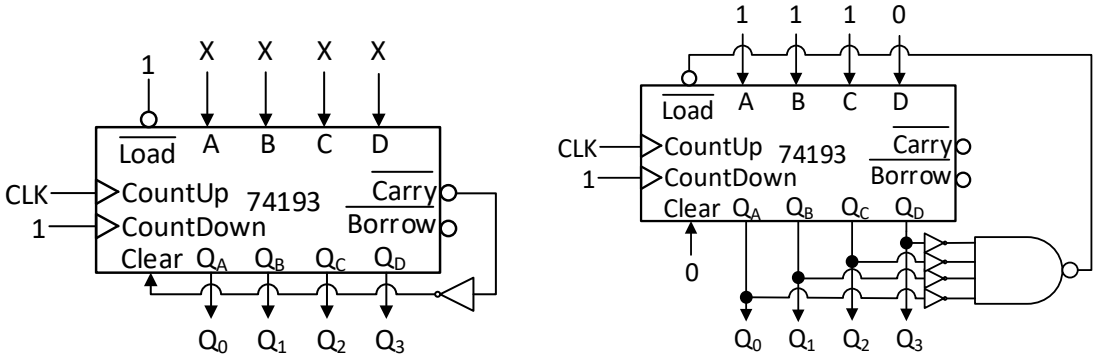


Figure 10. 11 Loops 0-14 (left) and 7-15 (right), implemented using counter 74193

The implementation using the decade counter 74192 is similar, excepting the counting loop, which must be in range 0-9. Figure 10. 12 highlights the implementation of loops 6-0 (6 is loaded when 9 is detected) and 8-1 (8 is loaded when 0 is detected).

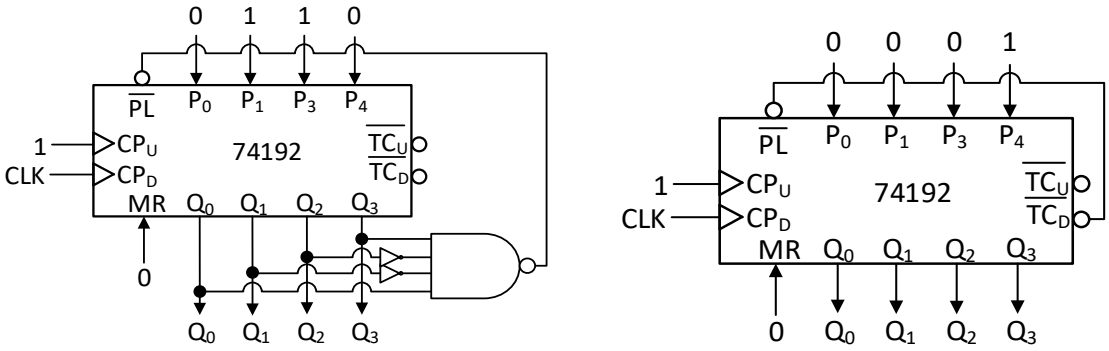


Figure 10. 12 Loops 6-0 (left) and 8-1 (right), implemented using counter 74192

When using 74162 and 74163 counters, only count up loops are possible, in range 0-9 or 0-15, respectively. The end of loop is detected, followed by a reset or load with the start of loop. Since reset and load are synchronous, their effect is only after the next rising

edge of the clock, thus avoiding the overwrite of the end of loop. The implementation of the loop $D_{3:0}$-*Val*, using the counter 74163, is presented in Figure 10. 13, along with the actual signal values, when the loop is restarted.

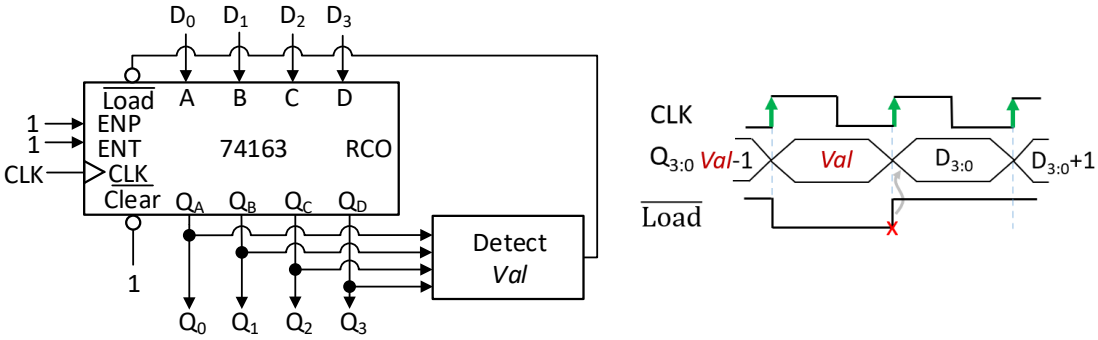


Figure 10. 13 Implementing loop $D_{3:0}$-*Val* using counter 74163 (left), and the signal values, when the loop is restarted (right)

To implement the loop 10-12, using counter 74163, the value 10 is loaded when 12 is detected on output. The test circuit is highlighted in Figure 10. 14.

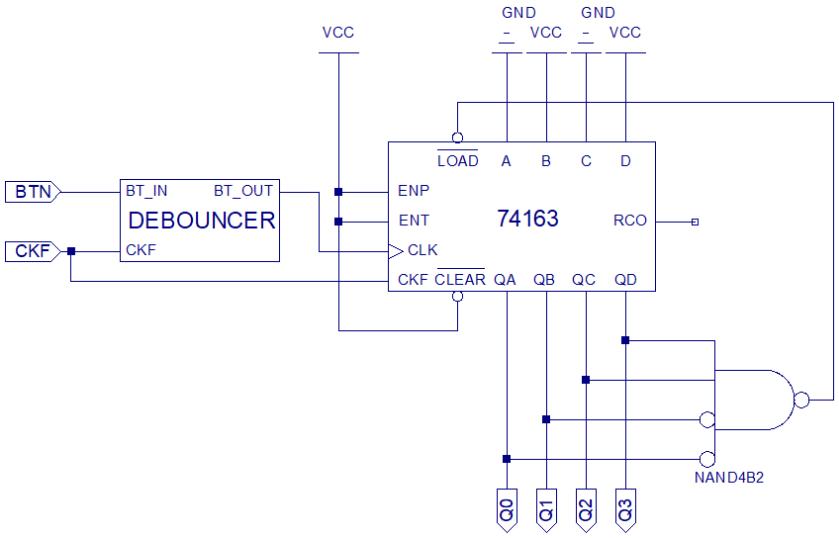


Figure 10. 14 Test circuit for the counter with loop 10-12, implemented using 74163

The following circuits implement the count up loops 4-9 and 0-8, using the decimal counter 74162:

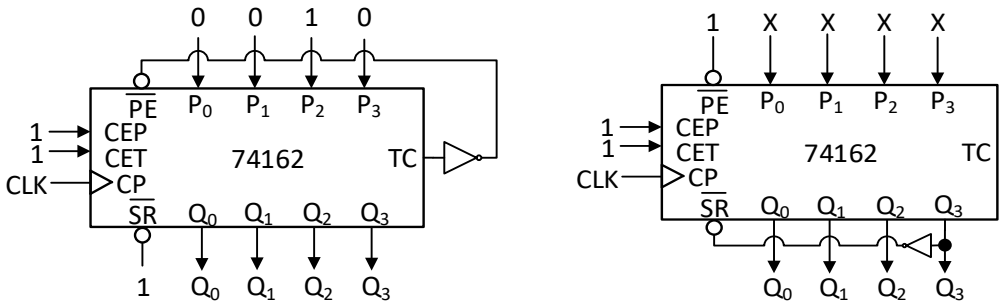


Figure 10. 15 Counters with loops 4-9 (left) and 0-8 (right), implemented with 74162

Note: In the first case, the value 9 is detected using the TC flag. In the second case, value 8 is detected using only Q_3, because it is activated solely when reaching 8 within the loop. For the rest of the loop Q_3=0.

When implementing extended loops, the counters can be cascaded onto a larger number of bits. For instance, loop 37-74 can be implemented using two cascaded 74193 counters, by loading value 37 (00100101_2) when detecting value 75 (01001011_2):

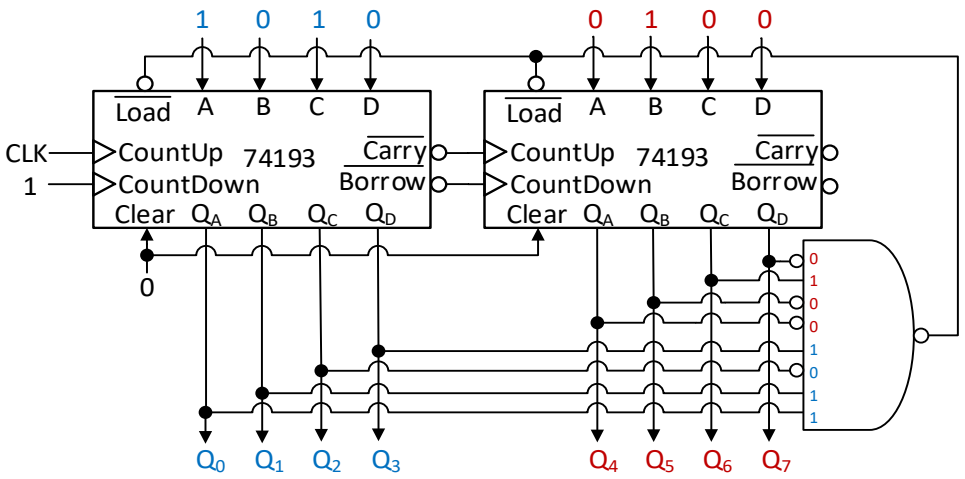
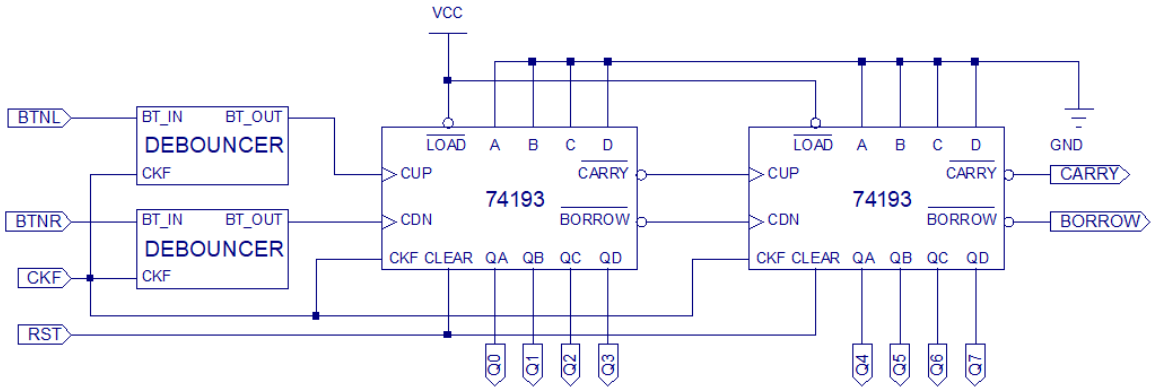


Figure 10. 16 Counter with loop 37-74, implemented using cascaded 74193 units

10.3 Assignments

1. Implement on the board the counter below, using two 74193 cascaded units. Test the asynchronous reset command, and the up-down counting loop 0-255.



2. Implement on the board the 4-bit *modulo 3* counter, with ascending counting loop 10-12, implemented using the 74163 unit. Test the counting loop.
3. Implement in Logisim an 8-bit counter, by cascading 74162 and 74163 units. Test the synchronous reset and load, and the ascending counting loop 0-159.
4. Implement in Logisim the 4-bit *modulo 7* counter, with descending counting loop 6-0, implemented using the 74192 unit. Test the counting loop.
5. Implement in Logisim the 4-bit *modulo 6* counter, with ascending counting loop 4-9, implemented using the 74162 unit. Test the counting loop.
6. Implement in Logisim the 8-bit *modulo 38* counter, with ascending counting loop 37-74, by cascading two 74193 units. Test the counting loop.

11 Sequential logic circuits – registers

11.1 Objectives

The registers are defined and classified based on their functionality. The implementation of the universal shift register, based on flip-flops and multiplexers, is studied. Two representative integrated circuits having register functionality are presented. Several numerical sequence generators are implemented using the analyzed registers.

11.2 Theoretical considerations

The registers are sequential logic circuits capable of storing or shifting binary data. They are implemented using as many bistables as the number of bits. For simplicity, the registers are often implemented with D flip-flops. The number of stored bits represent the *register capacity*. According to their functionality, the registers can be classified as:

- storage registers – can load and store data every clock cycle;
- shift registers – can shift data one position every clock cycle, while serially loading one bit of data: SerialBit→Q_A→Q_B→Q_C→ …;
- combined storage and shift registers – combines the functionality of the previous categories;
- universal shift registers – adds bidirectional shift to a combined storage and shift register.

11.2.1 The universal shift register

Figure 11. 1 highlights the implementation of a 4-bit universal shift register using D flip-flops and multiplexers. Each output bit is associated with a flip-flop and a multiplexer defining the various functionalities. The flip-flops can be reset asynchronously.

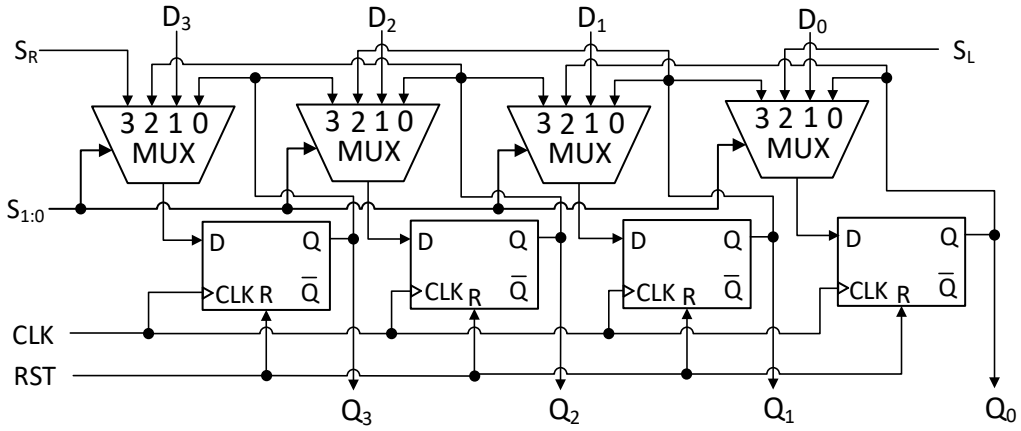


Figure 11. 1 Implementation of a 4-bit universal shift register using D flip-flops

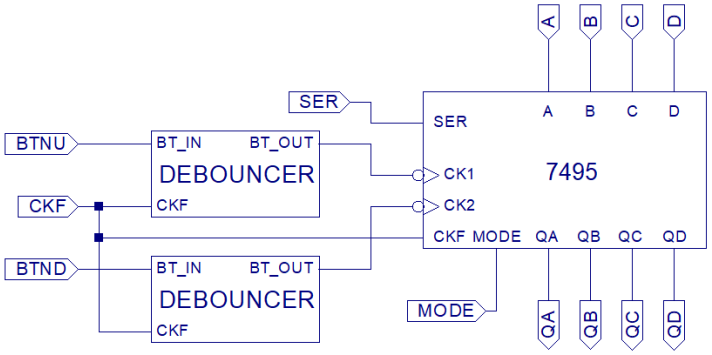
The selections S_1, S_0 define the functionality as follows:

- $S_{1:0}$=00 – storage: $Q_3Q_2Q_1Q_0{}^{t+1} = Q_3Q_2Q_1Q_0{}^{t}$;
- $S_{1:0}$=01 – parallel load: $Q_3Q_2Q_1Q_0{}^{t+1} = D_3D_2D_1D_0$;
- $S_{1:0}$=10 – serial load on S_L (Serial Left) with left shift: $Q_3Q_2Q_1Q_0{}^{t+1} = Q_2Q_1Q_0S_L{}^{t}$;
- $S_{1:0}$=11 – serial load on S_R (Serial Right) with right shift: $Q_3Q_2Q_1Q_0{}^{t+1} = S_RQ_3Q_2Q_1{}^{t}$.

Note: Serial load refers to filling one bit of data each clock cycle. Filling a 4-bit register entirely requires 4 consecutive clock cycles. Alternatively, a parallel load fills the register in 1 clock cycle.

11.2.2 The shift register 7495

The 4-bit shift register 7495 has the following functions: load, storage and data shift. The testing circuit and the function table are highlighted in Figure 11. 2. The register has two functions: if MODE=0, every falling edge of the clock signal CK_1, the data is shifted with serial load on input SER (shift direction: SER→Q_A→Q_B→Q_C→Q_D); if MODE=1, the binary value $(ABCD)_2$ is loaded synchronously on the CK_2 falling edge.



MODE	CK_1	CK_2	SER	$Q_{A:D}^{t+1}$
0	↓	X	0	$0\ Q_A^t\ Q_B^t\ Q_C^t$
0	↓	X	1	$1\ Q_A^t\ Q_B^t\ Q_C^t$
1	X	↓	X	A B C D

Figure 11. 2 Testing circuit for register 7495 (left) and the function table (right)

When cascading several 7495 registers, the same functionality is extended to a multiple of 4 bits. An 8-bit register is implemented from two 4-bit registers, as shown in Figure 11. 3.

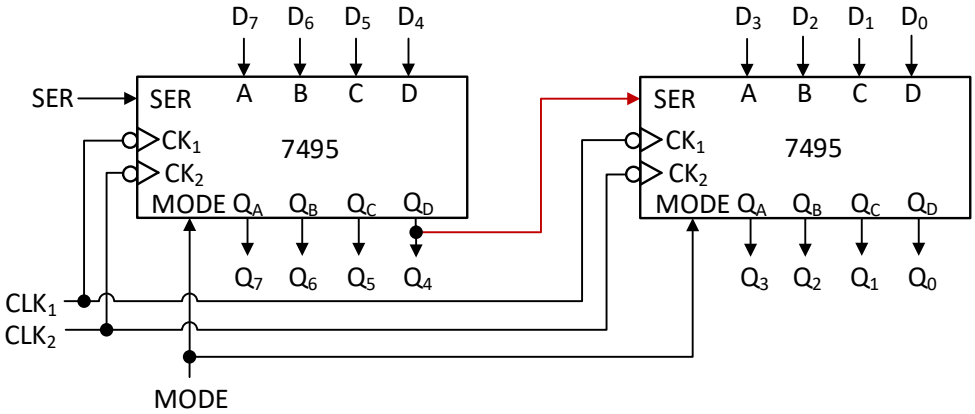


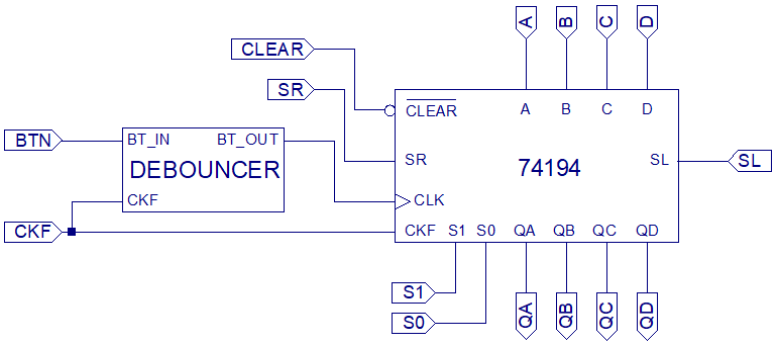
Figure 11. 3 Cascading two 7495 registers

The clock and MODE signals are shared by all registers. To enable the shift of data bits from one register to its right neighbor, the output Q_D is connected to the serial input SER of the neighbor, thus implementing the shift from Q_4 to Q_3.

11.2.3 The universal shift register 74194

Circuit 74194 is a 4-bit universal shift register with synchronous operations on the clock rising edge. The active low input $\overline{CLEAR}$ performs asynchronous reset and has priority over the other operations. Selection signals S_1 and S_0, define 4 synchronous operations, highlighted in the function table from Figure 11. 4:

- $S_{1:0}$=00 – for storage;
- $S_{1:0}$=01 – for right shift with serial load on S_R;
- $S_{1:0}$=10 – for left shift with serial load on S_L;
- $S_{1:0}$=11 – for load with binary value $(ABCD)_2$.



$\overline{CLR}$	S_1	S_0	CLK	$Q^{t+1}_{A:D}$			
0	X	X	X	0	0	0	0
1	0	0	↑	Q^t_A	Q^t_B	Q^t_C	Q^t_D
1	0	1	↑	S_R	Q^t_A	Q^t_B	Q^t_C
1	1	0	↑	Q^t_B	Q^t_C	Q^t_D	S_L
1	1	1	↑	A	B	C	D

Figure 11. 4 Testing circuit for register 74194 (left) and the function table (right)

To extend the capacity, several 74194 registers can be cascaded. Figure 11. 5 highlights two cascaded registers implementing an 8-bit universal shift register. Both 74914 units share the same clock signal, the asynchronous command $\overline{CLEAR}$, and the selection pair S_1 and S_0. Consequently, the registers have the same function regime. To shift the data bits between registers, the output Q_D is connected to the serial input S_R (Q_4→Q_3), and the output Q_A is connected to the serial input S_L (Q_3→Q_4).

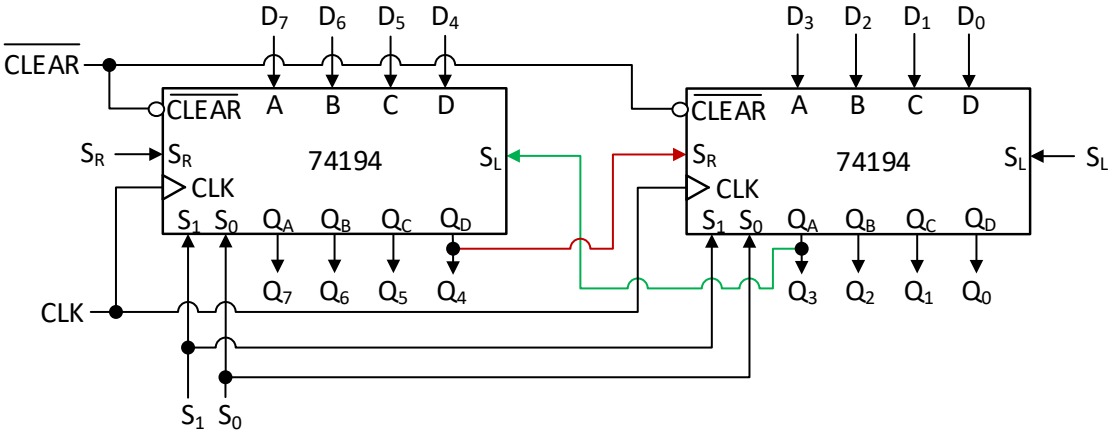


Figure 11. 5 Cascading two 74194 registers

11.2.4 Numerical sequence generators

Numerical sequences are arrays of values, which repeat in the same order. Such sequences can be easily generated using shift registers and additional combinational logic. A sequence containing 2^n-1 non-zero random *n*-bit values is called a *pseudo-random sequence*. Such a sequence can be generated using a register, initially loaded with any non-zero value. The register must perform data shift, while serially loading the XOR result between the most significant output bit and the least significant output bit. Figure 11. 6 shows two versions for implementing a 4-bit pseudo-random sequence generator using register 74194. The Initial value 0001_2 is loaded at startup, by setting $S_{1:0}$=11, for at least 1 clock cycle.

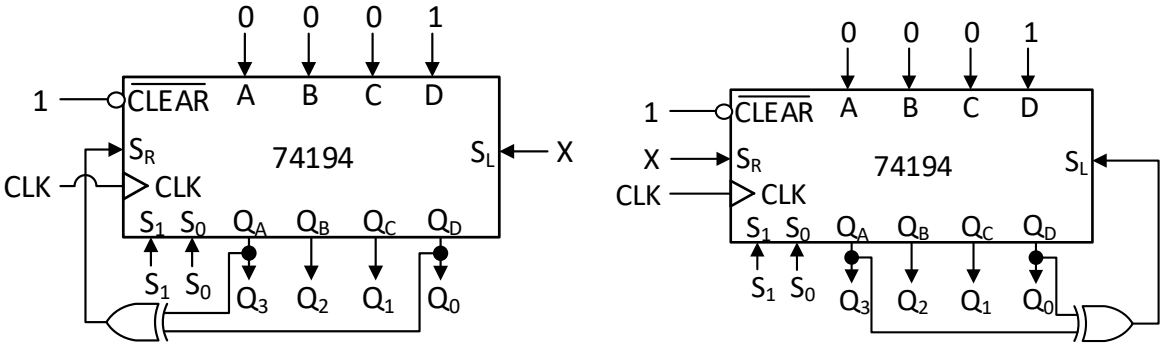


Figure 11. 6 The 4-bit pseudo-random sequence generator, implemented using register 74194, with serial load on S_R (left) and with serial load on S_L (right)

In the first case (Figure 11. 6 – left), after loading the initial value, the functionality is changed to right shifting with serial load on S_R, by setting $S_{1:0}$=01. The sequence generated on $Q_{3:0}$ contains the following 15 values: 0001 → 1000 → 1100 → 1110 → 1111 → 0111 → 1011 → 0101 → 1010 → 1101 → 0110 → 0011 → 1001 → 0100 → 0010. After the last value the sequence restarts, while keeping the same order.

In the second case (Figure 11. 6 – right), turning $S_{1:0}$=10 after the initial load, enables the left shift regime with serial load on S_L. The recurrent sequence generated on outputs becomes: 0001 → 0011 → 0111 → 1111 → 1110 → 1101 → 1010 → 0101 → 1011 → 0110 → 1100 → 1001 → 0010 → 0100 → 1000.

The implementation of the first sequence using register 7495 is highlighted in Figure 11. 7.

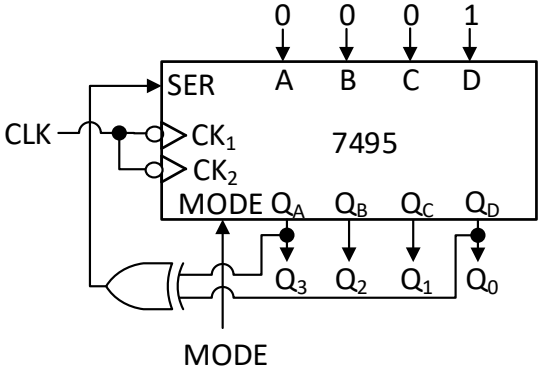


Figure 11. 7 The 4-bit pseudo-random sequence generator, implemented using 7495

The command MODE is set to MODE=1, for at least one clock cycle, and value $(0001)_2$ gets loaded into the register. Afterwards, MODE is set to MODE=0, and the rest of sequence is generated. Every transition between values takes place on the clock rising edge.

Note: A zero value should never be loaded, because the output will not change afterwards.

To generate pseudo-random sequences on a larger number of bits, the registers must be cascaded, as in Figure 11. 8. After loading the non-zero value, the ensemble transitions to data shifting regime. To implement a 6-bit sequence, 4 bits can be generated by one register, and 2 bits by the second register.

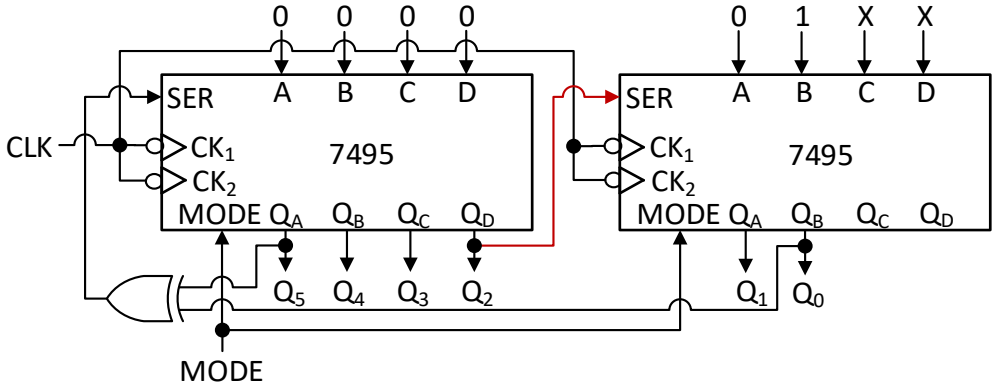


Figure 11. 8 A 6-bit pseudo-random sequence generator, implemented using cascaded 7495 registers

11.3 Assignments

1. Implement on the board the 7495 register. Test the commands synchronized with the clock falling edge. Considering Q_A is the most significant bit, load the following values, using parallel load and serial load: 13, 15, 17.
2. Implement on the board the 74194 register. Test the asynchronous reset, and the commands synchronized with the clock rising edge. Considering Q_A is the most significant bit, after an asynchronous reset, load the following values, using parallel load and serial load (in both directions): 10, 12, 6.
3. Implement and test in Logisim a 4-bit pseudo-random sequence generator starting with value 0001_2, loaded at startup. Implement using the 7495 register.
4. Implement and test in Logisim a 6-bit pseudo-random sequence generator starting with value 000001_2, loaded at startup. Implement using 7495 registers.
5. Implement in Logisim an 8-bit universal shift register, by cascading two 74194 registers. Test the asynchronous reset, and the commands synchronized with the clock rising edge. After an asynchronous reset, load the following values, using parallel load and serial load (in both directions): 17, 31, 67.

12 Logic design using programmable circuits of type FPGA

12.1 Objectives

The architecture of the FPGA (Field Programmable Gate Array) is presented. The structure of the configurable logic block is detailed and the design principles for using these resources. The automatic symbol creation of a circuit is described. Simulation basics with ISim tool are studied and the configuration steps required are mentioned. Simulation details are explained through practical examples.

12.2 Theoretical considerations

The development board used for testing is equipped with an FPGA chip, that contains a grid-like structure of Configurable Logic Blocks (CLBs), which are interconnected as in Figure 12. 1 – left. A CLB is composed of 2 or more *slices*. A *slice* contains *Look-Up Table* (LUT) memories (RAM units), *carry* logic for arithmetic operations and bistables. The LUT memories can be configured to implement any boolean expression of *n* variables, where *n* is the number of address bits. The *carry* logic enables fast carry propagation through bits from the lowest to the highest rank. The bistables are required to implement any type of sequential logic. The board is equipped with an Artix-7 FPGA model, whose *slice* has 4 LUTs and 8 bistables (Figure 12. 1 – right). The LUT response can be driven outside the *slice*, directly or through a bistable. Using the interconnection network available in the FPGA, the *slices* can be connected to themselves or to other *slices* from similar or different CLBs.

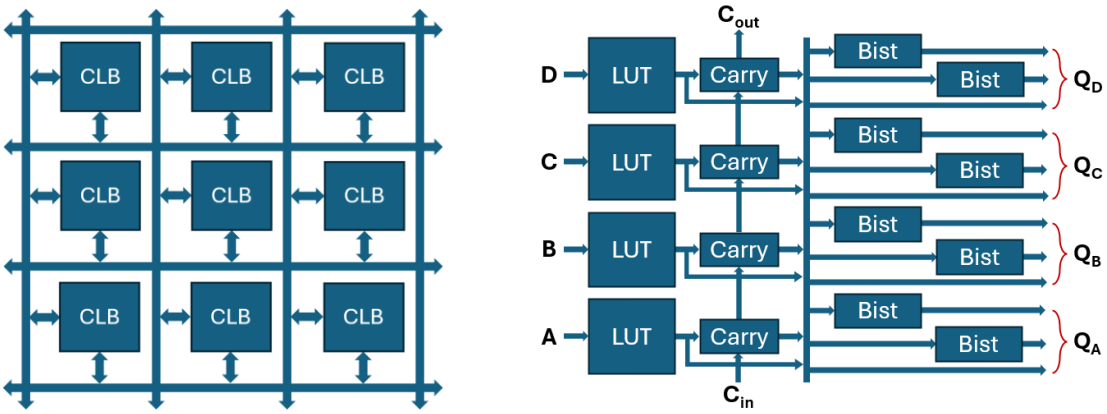


Figure 12. 1 The grid-like structure of the FPGA (left) and the composition of a *slice* (right)

The interconnection wires are distributed around the CLBs. They intersect through Programmable Interconnection Points (PIPs) controlled by the cells of a special RAM unit located inside the FPGA. This memory is separated from the CLB grid, and defines the active connections inside the FPGA, according to the architecture being implemented. Its content is written when the FPGA is programmed by the user.

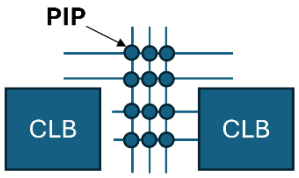
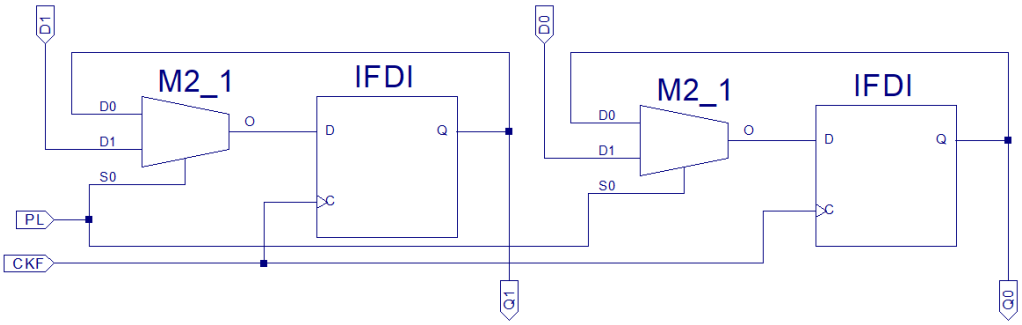


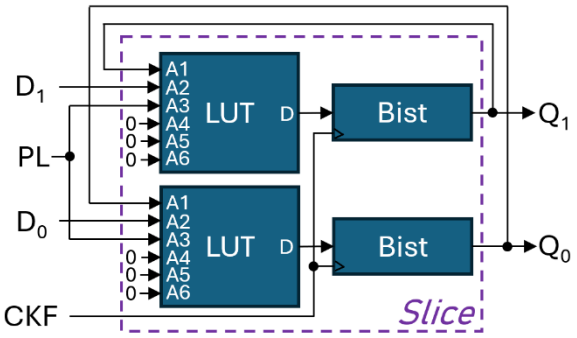
Figure 12. 2 Programmable interconnections spanning the grid of CLBs in FPGA

12.2.1 Implementing a circuit using FPGA resources

When implementing circuits using FPGA devices, the original architecture is redesigned based on the available resources inside the CLBs. The process is carefully conducted, to avoid altering the initial functionality. The synthesis and implementation tools analyze the proposed architecture and automatically perform the changes necessary for a CLB-based structure. They also allocate the CLBs from the grid, decide the internal configuration of the *slices* and the necessary interconnections. For instance, the 2-bit register in Figure 12. 3 (a) can be implemented in a single *slice*, using two LUTs and two D flip-flops, connected as in Figure 12. 3 (b). The LUTs can be configured to implement MUX 2:1 units, if written with the output column from Figure 12. 3 (c). Consequently, the LUTs will output the value of bit address A_1 or A_2, based on the selection connected to bit address A_3. The rest of address bits A_4, A_5, A_6 will be connected to 0, inside the slice, without requiring any wires from the interconnection grid.



(a)



(b)

A_6	A_5	A_4	A_3	A_2	A_1	D
0	0	0	0	0	0	0
0	0	0	0	0	1	1
0	0	0	0	1	0	0
0	0	0	0	1	1	1
0	0	0	1	0	0	0
0	0	0	1	0	1	0
0	0	0	1	1	0	1
0	0	0	1	1	1	1
0	0	1	X	X	X	X
0	1	X	X	X	X	X
1	X	X	X	X	X	X

(c)

Figure 12. 3 The 2-bit register (a) is implemented using one *slice* inside the FPGA (b) and the 64x1 LUTs are configured with the bitmap table (c) to act like 2:1 multiplexers

In Project Navigator, the synthesis and implementation phases precede the programming .bit file generation. The succession is highlighted in Figure 12. 4. Besides other tasks, the programming step with the ISE iMPACT tool writes the RAM which controls the programmable connections, and the LUTs inside the *slices* that were allocated.

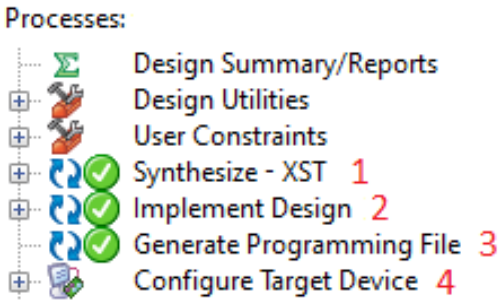


Figure 12. 4 Steps for circuit implementation in Project Navigator

12.2.2 Generating the symbol of a circuit

A circuit can be encapsulated in a representative symbol, which can be later integrated into other designs. To generate a symbol in Project Navigator, the circuit must be selected in the Design panel. Extend Design Utilities section at the bottom, click on Create Schematic Symbol and chose Run. These steps are highlighted in Figure 12. 5 – left. The symbol generated will be available in the project library under the name of the circuit. Figure 12. 5 – right shows the symbol for the 2-bit register in Figure 12. 3 (a). The pins of the symbol represent the inputs and outputs of the register.

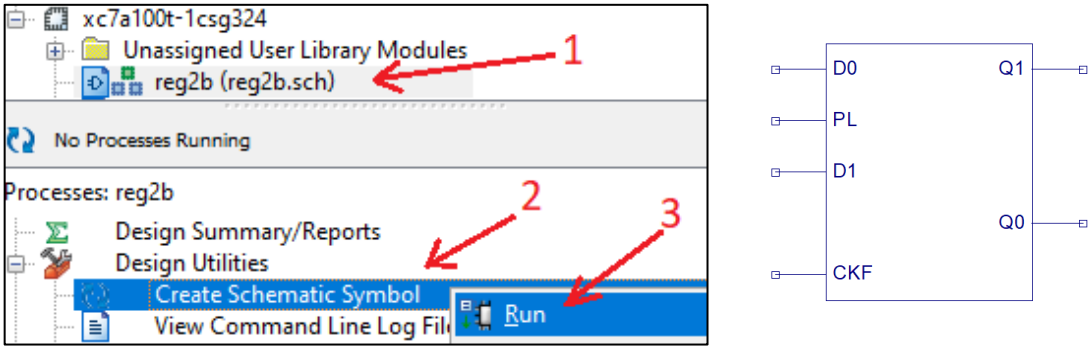


Figure 12. 5 Steps for generating the symbol of a circuit in Project Navigator (left); the symbol for the 2-bit register from previous section (right)

12.2.3 Simulating the functionality of a circuit

The simulation module is called ISim and can be launched from Project Navigator, for any circuit under development. The launching steps are: select Simulation at the top of Design panel, click on the diagram, extend ISim Simulator at the bottom, right click on Simulate Behavioral Model and select Run. The steps are highlighted in Figure 12. 6. ISim will display the waveforms generated for the signals in a dedicated area.

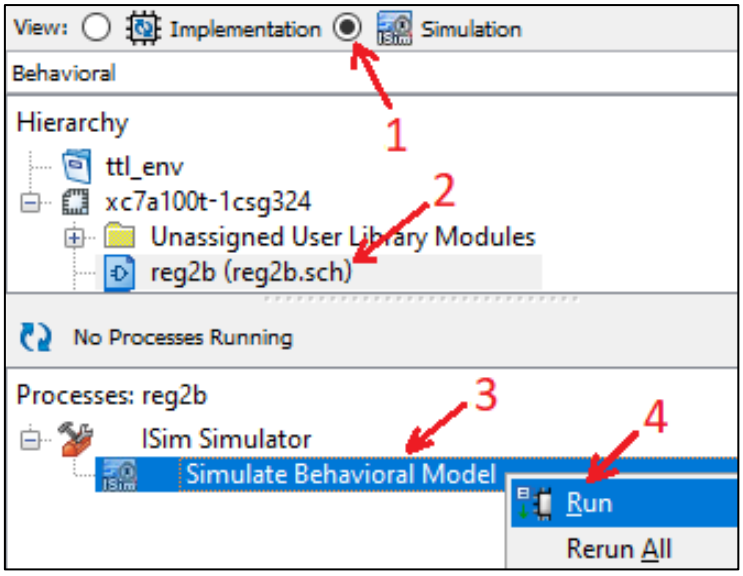


Figure 12. 6 Launching the simulation tool ISim from Project Navigator

Regarding the register in Figure 12. 3 (a), the signals listed for simulation represent the inputs, the outputs and other internal signals (Figure 12. 7). Initially, their values are set to unknown (U).

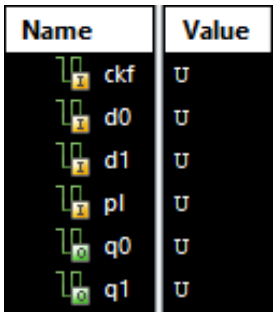


Figure 12. 7 The list of active signals in the simulation window

The simulation can be set at time 0 by pressing Restart in the toolbar at the top: 

. Each signal can be assigned different values by right click on its name and choosing Force Constant …. In the next step, set the value for attribute Force to Value, as shown in Figure 12. 8.

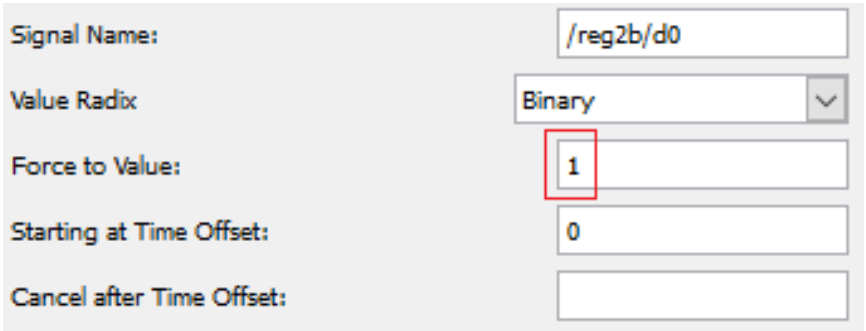


Figure 12. 8 Setting a signal in the simulator to 1

The clock signal can be set by choosing Force Clock… instead of Force Constant…. In the next step (Figure 12. 9), set attributes Leading Edge Value = 0, Trailing Edge Value = 1 and Period = 10ns (the equivalent of 100MHz). **Note**: The clock Period can be assigned any value.

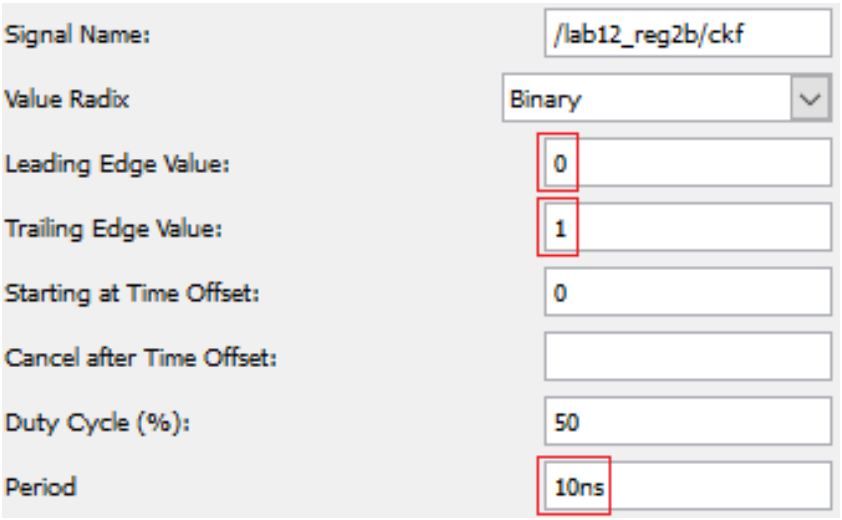


Figure 12. 9 The clock signal period is set to 10ns in the simulator

During simulation, the execution is partitioned into simulation steps. The time interval for the simulation step can be set in the toolbar: 

. For simplicity, a good choice is to set to 10ns, the equivalent of the clock period. A simulation step is executed when pressing the button  . The simulator displays the waveforms of the signals for 10ns. The values of the inputs can be modified before each step, using option Force Constant…. Figure 12. 10 shows a 4-step simulation, the equivalent of 40ns, for the 2-bit register. During the first cycle, value 2 is written on the rising edge of CKF by setting PL=1 and D_1D_0=10. During the next cycles, by disabling PL (PL=0), the state of the register remains unaltered, despite the values set on D_1D_0. **Note:** On a visual scale, the waveforms can be changed by using Zoom In  , Zoom Out  and Zoom to Full View  , from the toolbar.

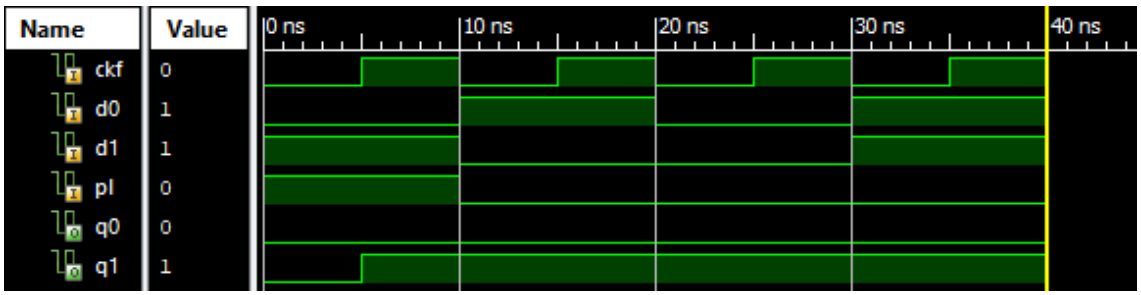


Figure 12. 10 Testing a 2-bit register using the ISim simulation tool

Inside the simulation panel, the signals can be added by *drag&drop* from the Objects window (Figure 12. 11 – left). Also, the signals can be removed by right click and Delete (Figure 12. 11 – right).

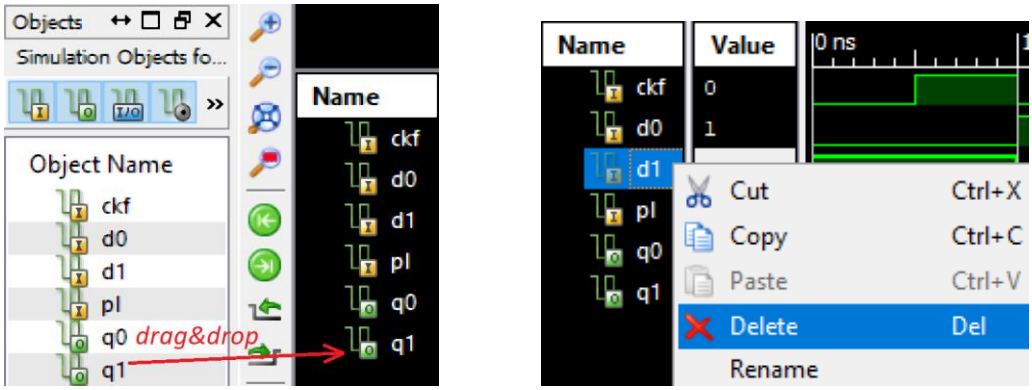
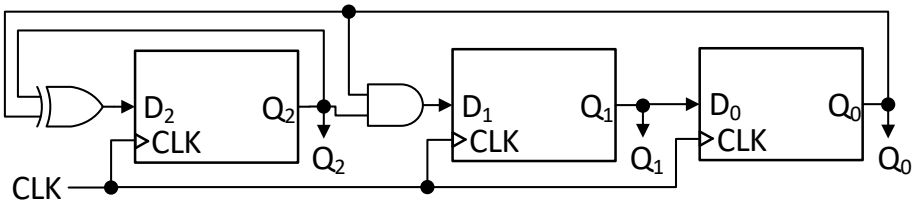


Figure 12. 11 Adding signals to the waveform panel (left) and elliminating the signals from the panel (right)

12.3 Assignments

1. Implement the next circuit with *slices* and define the bitmaps for the LUT memories required:



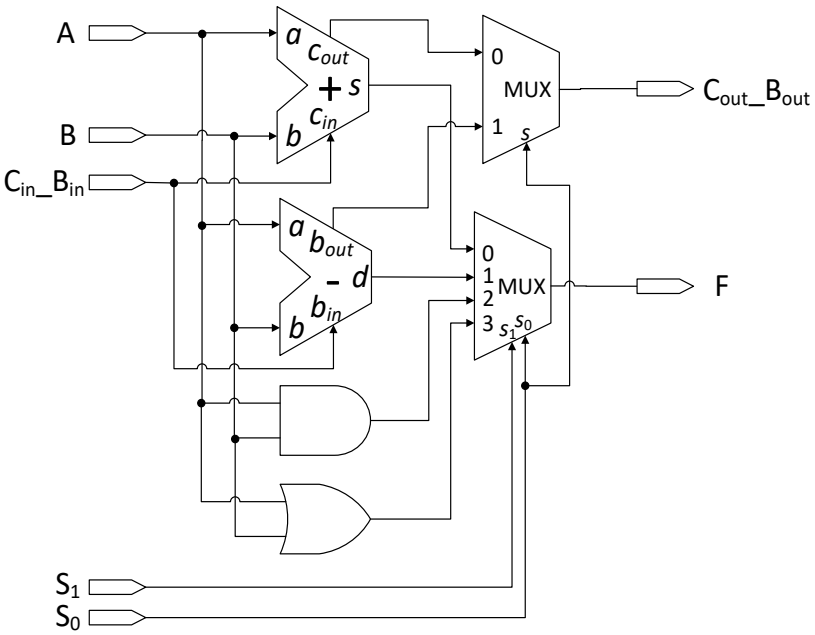
2. Implement in Project Navigator the 2-bit register from Figure 12. 3 (a), generate its symbol and simulate its functionality using ISim.

3. Implement in Project Navigator a 1-bit full adder using basic logic gates and generate its symbol. In a new diagram, implement a 3-bit full adder, by cascading the 1-bit full adder. Test the functionality of new circuit using ISim.

A. Annex 1 – Problems with combinational logic circuits

1. Implement a 1-bit ALU (Arithmetic-Logic Unit) for two operands A, B capable to perform the following operations given the selection signals S_1S_0: 00 -> unsigned summation; 01 -> unsigned subtraction A-B; 10 -> bitwise logical AND; 11 -> bitwise logical OR. The unit shall have 2-role Carry/Borrow signals on input and output.

Solution: The ALU (Arithmetic Logic Unit) can be found in most microprocessors and is capable of various arithmetic and logic operations. The operations are performed in parallel. The output F takes the result indicated by selection inputs. As there are 4 operations, the selection will be implemented using a MUX4:1 unit. The 2-role Carry/Borrow signal appears on both input and output. Using a MUX2:1 unit, the signal S_0 sets the role of Carry or Borrow on output. It can be noticed that for $S_1 = 1$ the output $C_{out}\_B_{out}$ has no meaning, because the operation selected is either AND or OR. Hence, $C_{out}\_B_{out}$ has meaning when $S_1 = 0$ and should be ignored when $S_1 = 1$.



Implementation of 1-bit **adder** unit:

$c_{out} = (b \cdot c_{in}) + (a \cdot c_{in}) + (a \cdot b)$

$s = a \oplus b \oplus c_{in}$

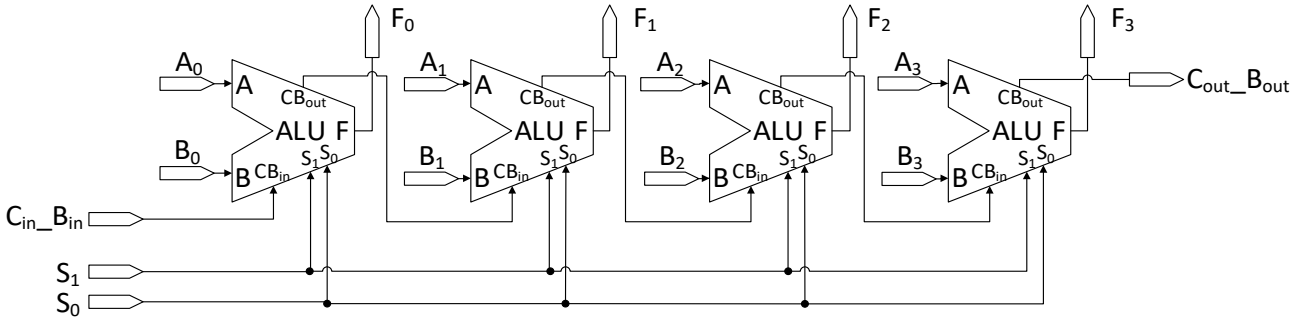
Implementation of 1-bit **subtractor** unit:

$b_{out} = (\bar{a} \cdot b_{in}) + (\bar{a} \cdot b) + (b \cdot b_{in})$

$d = a \oplus b \oplus b_{in}$

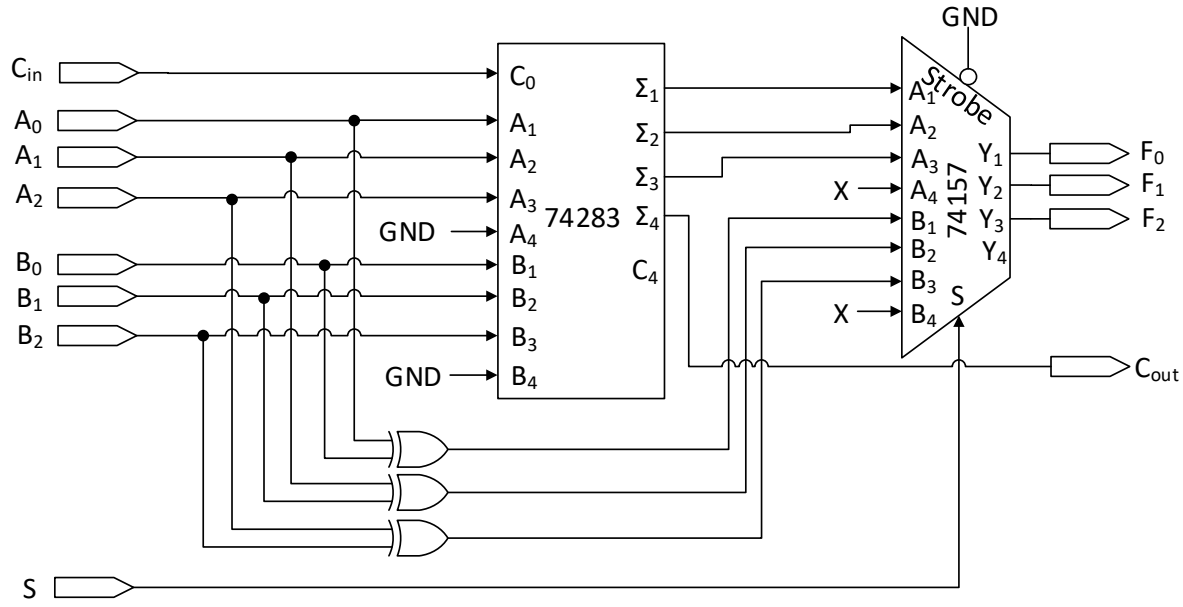
2. Using 1-bit ALUs (Arithmetic-Logic Units) implement a 4-bit ALU for two operands A, B capable to perform the following operations given the selection signals S_1S_0: 00 -> unsigned summation; 01 -> unsigned subtraction A-B; 10 -> bitwise logical AND; 11 -> bitwise logical OR. The unit shall have 2-role Carry/Borrow signals on input and output.

Solution: A multi-bit ALU can be implemented by cascading several 1-bit ALUs. The selection signals S_1S_0 are commonly used by all 1-bit ALUs. The input Carry/Borrow signal is connected to the less significant 1-bit ALU. The output Carry/Borrow signal is generated by the most significant 1-bit ALU. The intermediary Carry/Borrow signals are propagated though the rest of ALUs from inferior ranks to higher ranks. It can be noticed that even if the operations are performed in parallel the Carry/Borrow information travels from one end to the other. Hence, when changing the input operands the result stabilizes after the end of the propagation.



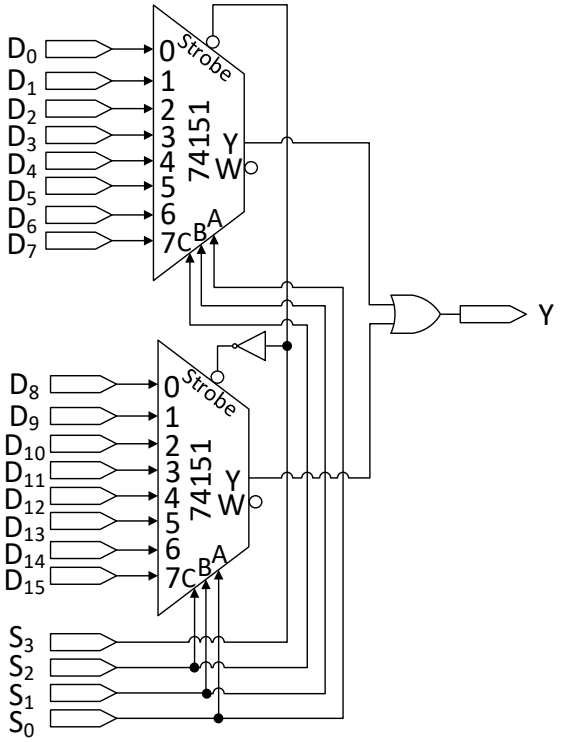
3. Implement a 3-bit ALU for two operands A, B capable to perform the following operations given the selection signal S: 0 -> unsigned summation; 1 -> bitwise $A \oplus B$. The unit shall have Carry signals on input and output. Use the 74283 adder and the 74157 multiplexer.

Solution: As there are 2 operations, the selection of the result on output F will be implemented using a MUX2:1 unit (74157). The output takes the appropriate value based on the selection S. The summation can be implemented using the 4-bit adder 74283. Since only 3 bits are required, they will connect to the less significant inputs of 74283, and the rest will be zeroed (GND), to leave the summation unaffected. The A_4 and B_4 inputs of the 74157 multiplexer are irrelevant (X means they are connected to either VCC or GND).



4. Implement a MUX 16:1 unit using two 74151 multiplexers and two additional logic gates.

Solution: The 74151 multiplexers have an *Strobe* signal, which can be used to alternate the enabling of the two units (one active, one inactive). The most significant S_3 selection will decide the active unit by connecting it to the *Strobe* for one multiplexer, and inverted for the other. The other selections S_2, S_1, S_0 will be common for both multiplexers. The output of both 74151 units will be ORed to generate the final result. This way a selection value $S_{3:0}$ in range 0-7 will enable the superior multiplexer and a selection value in range 8-15 will enable the inferior multiplexer. At any time, the disabled multiplexer will ouput 0, thus keeping the final OR result unaffected.



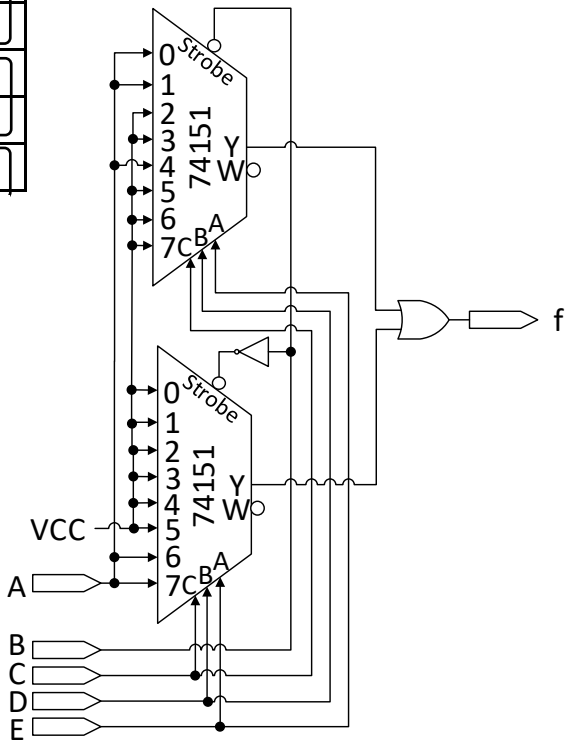
5. Implement function $f(A, B, C, D, E) = A + \overline{C} \cdot D + B \cdot \overline{D} + \overline{B} \cdot D + \overline{B} \cdot C \cdot E$ using 74151 multiplexers. The available inputs are 0 (GND), 1 (VCC) and the variables in direct form, not inverted.

Solution: Because A is not inverted in the expression, the variables B, C, D, E are connected to the selections of a MUX 16:1. Therefore, the cells in the Karnaugh map are grouped based on identical combinations for B, C, D, E. If a group has similar value, that value is applied to the multiplexer input selected by the code on B, C, D, E. If a group has different values the corresponding input is decided based on the value of A for the cell containing 1:

- If A=0, then $\overline{A}$ is applied.
- If A=1, then A is applied.

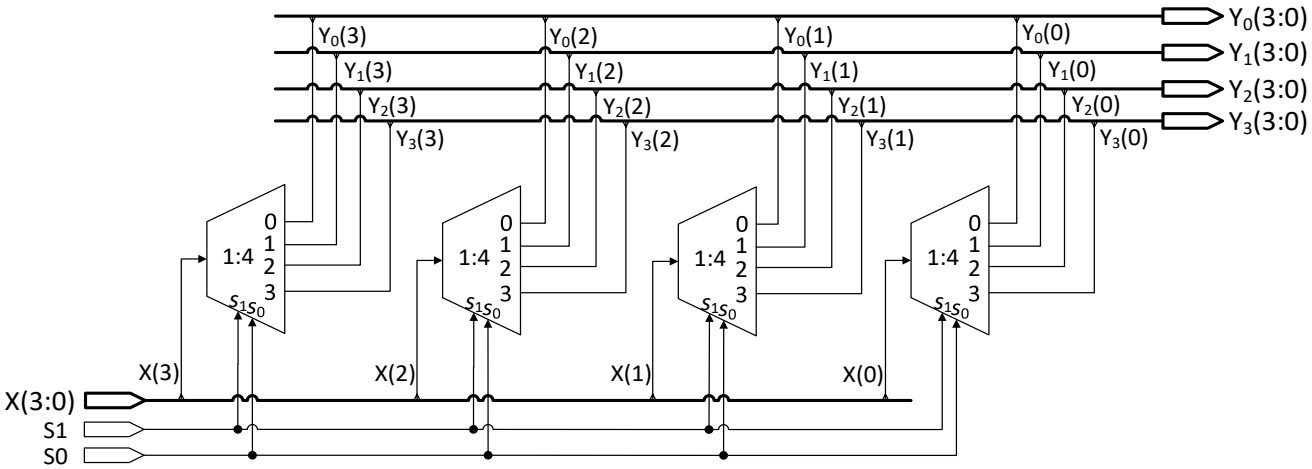
Note: The implementation of a MUX 16:1 unit using two 74151 units has been presented in the previous exercise.

AB \ CDE	000	001	011	010	110	111	101	100
00	0	0	1	1	1	1	1	0
01	1	1	1	1	0	0	1	1
11	1	1	1	1	1	1	1	1
10	1	1	1	1	1	1	1	1



6. Implement a 4-bit DMUX 1:4 unit using 1-bit DMUX 1:4 units.

Solution: The implementation of a 4-bit DMUX 1:4 unit requires four 1-bit DMUX 1:4 units. The DMUX units will share both selection signals S_1 and S_0. Each DMUX will generate a certain output bit from the output buses. A DMUX will decide for bit 0, the next DMUX will decide for bit 1, etc. The data inputs of the DMUX units will be connected to a unique bit from the input data bus.



7. Implement a 4:2 priority encoder using logic gates. The highest priority input has index 3. Inputs and outputs are active-high. When all inputs are inactive the output code is 00.

Solution: The truth table is designed and the outputs are minimized using Karnaugh maps. X values in the truth table are replaced by all possible combinations of 0 and 1, thus filling more cells in the Karnaugh maps – one cell for each combination.

Truth table

X_3	X_2	X_1	X_0	Y_1	Y_0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	X	0	1
0	1	X	X	1	0
1	X	X	X	1	1

Y_1:

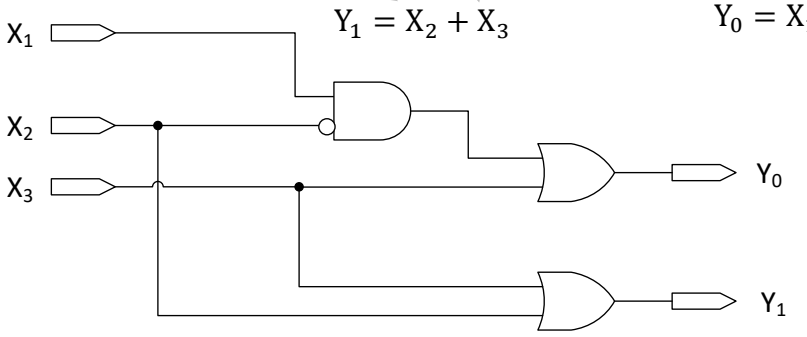
X_3X_2 \ X_1X_0	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	1	1	1	1
10	1	1	1	1

Y_0:

X_3X_2 \ X_1X_0	00	01	11	10
00	0	0	1	1
01	0	0	0	0
11	1	1	1	1
10	1	1	1	1

$Y_1 = X_2 + X_3$

$Y_0 = X_1 \cdot \overline{X_2} + X_3$



8. Implement $f = \sum(0,1,2,4,6)$ using 3 possible implementation strategies.

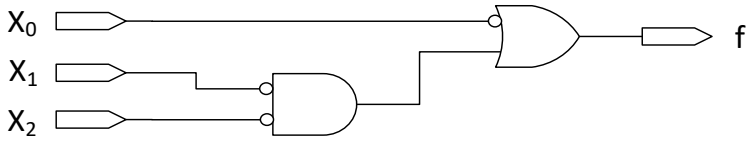
Solution: Because the highest minterm is P_6 the function requires 3 input variables to encode it.

Version 1

The implementation using logic gates requires the minimal disjunctive normal form (MDNF) obtained from the Karnaugh map.

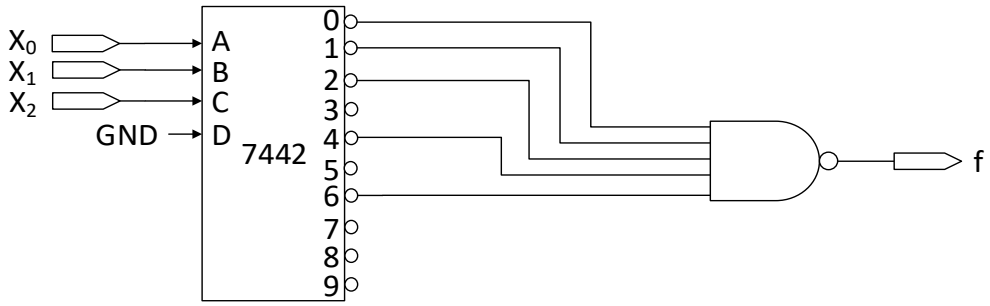
X_2 \ X_1X_0	00	01	11	10
0	1	1	0	1
1	1	0	0	1

$f = \overline{X_0} + \overline{X_2} \cdot \overline{X_1}$



Version 2

The implementation using a decoder requires the canonical disjunctive normal form (CDNF). The 7442 BCD-to-Decimal decoder implements the inverted minterms on its outputs, therefore a NAND on the right outputs should generate a disjunctive OR (acc. De Morgan law), hence implementing the canonical disjunctive normal form of the function. The 3 variables are connected to the less significant inputs of 7442 and the 4th input is zeroed (GND).

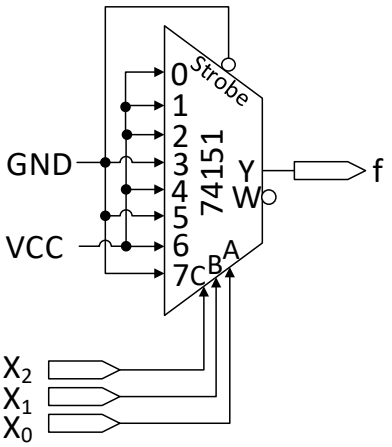


Version 3

When using MUX 8:1 (74151) the input variables will be connected to selections according to their rank. Therefore, the MUX will implement the canonical disjunctive normal form (CDNF) by ORing the minterms from the truth table. In this regard, the data inputs corresponding to function minterms will be connected to 1 (VCC) and the rest will be connected to 0 (GND), which means the output column of the truth table can be applied to MUX data inputs.

Truth table

	x_2	x_1	x_0	f_2
0	0	0	0	1
1	0	0	1	1
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	0
6	1	1	0	1
7	1	1	1	0



9. Implement $f = (\overline{A + B} \oplus B) \cdot C$ using NAND gates.

Solution: To implement with NAND gates, the function is minimized to minimal disjunctive normal form (MDNF) using the Karnaugh map. The MDNF is transformed using double negation and De Morgan law. Finally, the result becomes a NAND-based expression. Any inverted variable in the resulting expression will be implemented with a NAND gate using one of these equations: $\overline{X} = \overline{X \cdot X} = \overline{X \cdot 1}$.

Truth table

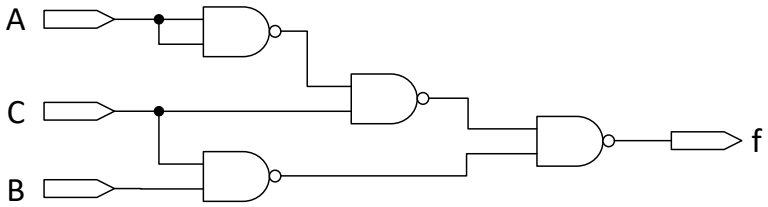
A	B	C	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

A \ BC	00	01	11	10
0	0	1	1	1
1	0	0	1	1

$$f \xrightarrow{\text{MDNF}} \overline{A} \cdot C + B \cdot C = \overline{\overline{\overline{A} \cdot C + B \cdot C}} = \overline{\overline{\overline{A} \cdot C} \cdot \overline{B \cdot C}} = \overline{\overline{\overline{A} \cdot \overline{A} \cdot C} \cdot \overline{B \cdot C}}$$

Double negation

De Morgan



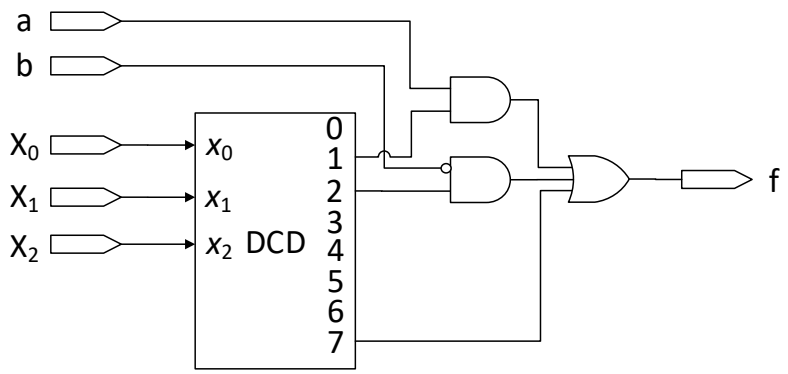
10. Using a decoder with active-high outputs implement the function f corresponding to the Karnaugh map below.

X_2 \ X_1X_0	00	01	11	10
0	0	a	0	$\overline{b}$
1	X	X	1	0

Solution: The implementation using a decoder requires the expression in canonical disjunctive normal form (CDNF). The Karnaugh map represents a disjunction (OR) over the minterms corresponding to cells of 1. Additionally, cells with embedded expression generate a conjunction (AND) between the expression and the corresponding minterm. The X values will be replaced with 0 to reduce the number of minterms. The function can be written as: $f = a \cdot P_1 + \overline{b} \cdot P_2 + 1 \cdot P_7$, where $P_1 = \overline{X_2} \cdot \overline{X_1} \cdot X_0$, $P_2 = \overline{X_2} \cdot X_1 \cdot \overline{X_0}$ and $P_7 = X_2 \cdot X_1 \cdot X_0$ are minterms of variables X_2, X_1, X_0.

Variables X_2, X_1, X_0 will be connected to the decoder inputs, hence the minterms will be available on the outputs. Using the outputs, the CDNF can be implemented accordingly with additional logic gates.

Note: The implementation may follow the values from the Karnaugh map without necesarilly rewritting the CDNF.



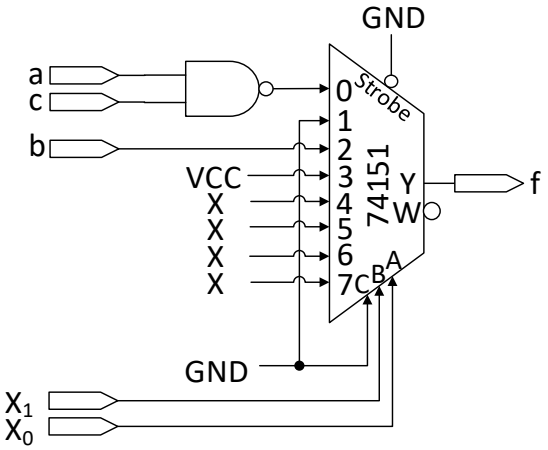
11. Implement the function f corresponding to the Karnaugh map below using a 74151 multiplexer.

X_1 X_0	0	1
0	$\overline{a \cdot c}$	X
1	b	1

Solution: The implementation using a multiplexer requires the expression in canonical disjunctive normal form (CDNF). The Karnaugh map represents a disjunction (OR) over the minterms corresponding to cells of 1. Additionally, cells with embedded expression generate a conjunction (AND) between the expression and the corresponding minterm. The X values will be replaced with 0 to reduce the number of minterms. The function can be written as: $f = \overline{a \cdot c} \cdot P_0 + b \cdot P_2 + 1 \cdot P_3$, where $P_0 = \overline{X_1} \cdot \overline{X_0}$, $P_2 = X_1 \cdot \overline{X_0}$, $P_3 = X_1 \cdot X_0$ are minterms of variables X_1, X_0.

As multiplexer 74151 has 3 selection inputs, both variables X_1, X_0 will be connected to the less significant selections. The remaining selection will be 0ed (GND). Therefore, the values from the Karnaugh map can be connected to data inputs 0 to 3. Since selection C = 0, inputs 4 to 7 are irrelevant (X means they are connected to either VCC or GND).

Note: The implementation may follow the values from the Karnaugh map without necesarilly rewritting the CDNF.



12. Implement $f(A, B, C, D) = \bar{b} \cdot P_2 + P_7 + (a \oplus c) \cdot P_{13} + P_{14} + P_{15} + \sum_{\Phi}(1,3,4,5,6,8,9,10,11)$ with logic gates, where P_i are the minterms of variables A, B, C, D (in order, A is most significant and D is least significant).

Solution:

The implementation using logic gates requires the minimal disjunctive normal form (MDNF) obtained from the Karnaugh map. The embedded expressions are placed in those cells corresponding to their conjuction minterms.

AB \ CD	00	01	11	10
00	0	X	X	$\bar{b}$
01	X	X	1	X
11	0	$a \oplus c$	1	1
10	X	X	X	X

Minimization

Step₁: Replace embedded expressions with 0 and generate a minimal number of maximal rectangular groups containing 1s and Xs, such that all 1s are covered.

Step₂: Replace 1s with Xs and generate a minimal number of maximal rectangular groups covering all embedded expressions. A group cannot contain two different expressions. For each group generate an AND between the embedded expression and the corresponding term of the group.

Step₃: Apply OR over the results from previous steps.

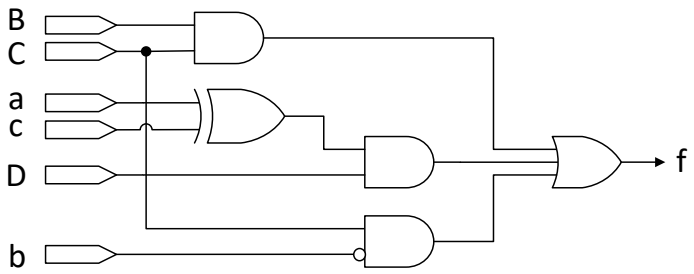
Step₁

AB \ CD	00	01	11	10
00	0	X	X	0
01	X	X	1	X
11	0	0	1	1
10	X	X	X	X

Step₂

AB \ CD	00	01	11	10
00	0	X	X	$\bar{b}$
01	X	X	X	X
11	0	$a \oplus c$	X	X
10	X	X	X	X

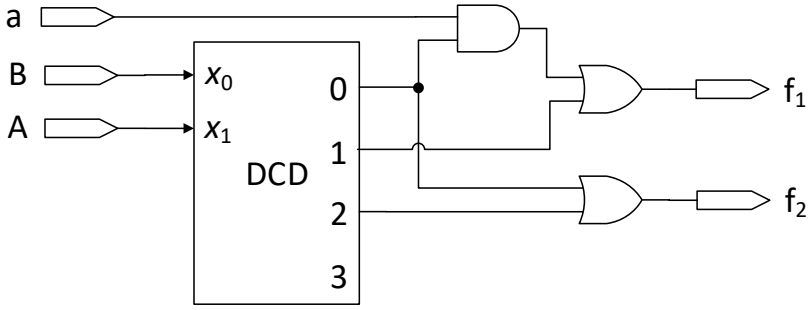
Step₃ $f = B \cdot C + (a \oplus c) \cdot D + \bar{b} \cdot C$



13. Use a decoder with active-high outputs to implement $f_1(A, B) = a \cdot P_0 + P_1$ and $f_2(A, B) = \sum(0,2)$, where P_i are the minterms of variabiles A, B (in order, A is the most significant and B is the least significant).

Solution:

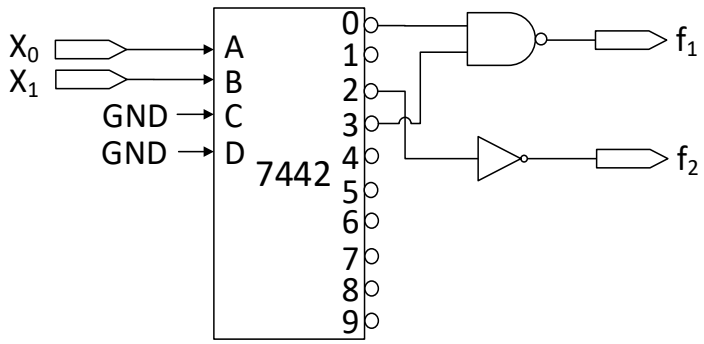
The implementation using a decoder requires the canonical disjunctive normal form (CDNF). Both functions are expressed in CDNF. Since they use the same set of variables they can be implemented with the same decoder. Variables A and B will be connected to the decoder inputs, hence the minterms will be available on the outputs. The embedded expressions will AND with the outputs correponding to minterms. Finally, OR gates will generate the functions' expressions.



14. Implement functions $f_1 = \sum(0,3)$ and $f_2 = \sum(2)$ using the 7442 decoder.

Solution:

The implementation using a decoder requires the canonical disjunctive normal form (CDNF). As noticed, both functions are expressed in CDNF. Since they use the same set of variables they can be implemented with the same decoder. The 7442 BCD-to-Decimal decoder implements the inverted minterms on its outputs, therefore a NAND over the right outputs should generate a disjunctive OR (acc. De Morgan law), hence implementing a canonical disjunctive normal form. The two variables are connected to the less significant inputs of 7442. The 3rd and 4th inputs are zeroed (GND). Relevant minterms will be found on outputs 0 to 3. f_1 has 2 minterms and requires a 2-input NAND. f_2 has 1 minterm and requires an inverter.



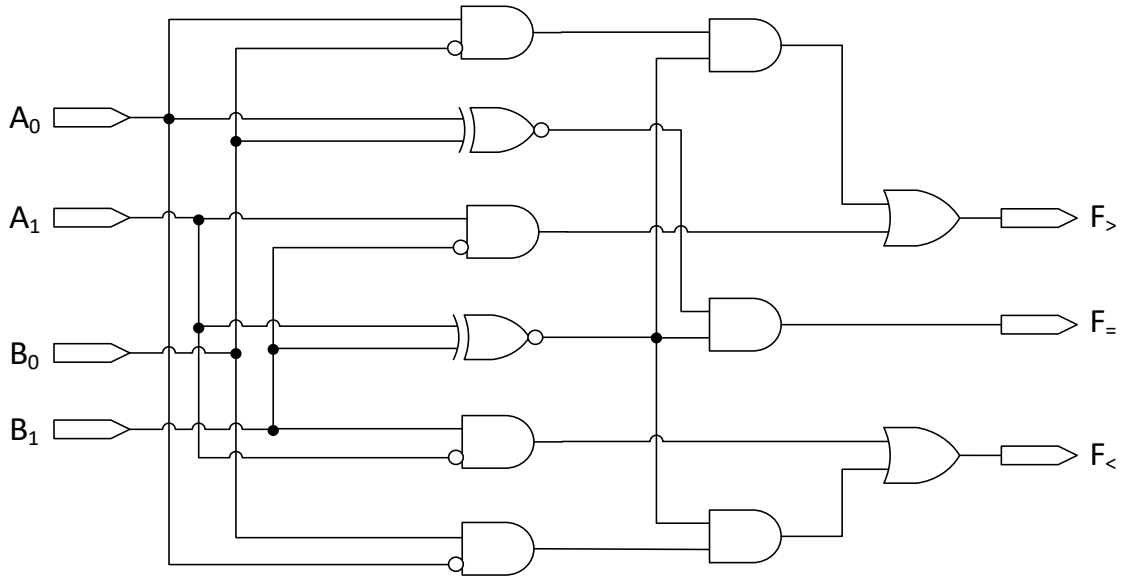
15. Implement – using logic gates – a comparator which performs >, <, = between 2 unsigned 2-bit operands.

Solution: One possible solution would be to define the truth table having the 2-bit operands on inputs and the results on three output columns for <, > and =, respectively. Each output function can be minimized using the Karnaugh maps.

Another solution would be to implement the following arithmetic rules for unsigned numbers:

- $F_> = 1$, if A > B, meaning $A_1 > B_1$ or ($A_1 = B_1$ and $A_0 > B_0$) $\Leftrightarrow F_> = A_1 \cdot \overline{B_1} + (A_1 \odot B_1) \cdot (A_0 \cdot \overline{B_0})$
- $F_< = 1$, if A < B, meaning $A_1 < B_1$ or ($A_1 = B_1$ and $A_0 < B_0$) $\Leftrightarrow F_< = \overline{A_1} \cdot B_1 + (A_1 \odot B_1) \cdot (\overline{A_0} \cdot B_0)$
- $F_= = 1$, if A = B, meaning $A_1 = B_1$ and $A_0 = B_0$ $\Leftrightarrow F_= = (A_1 \odot B_1) \cdot (A_0 \odot B_0)$

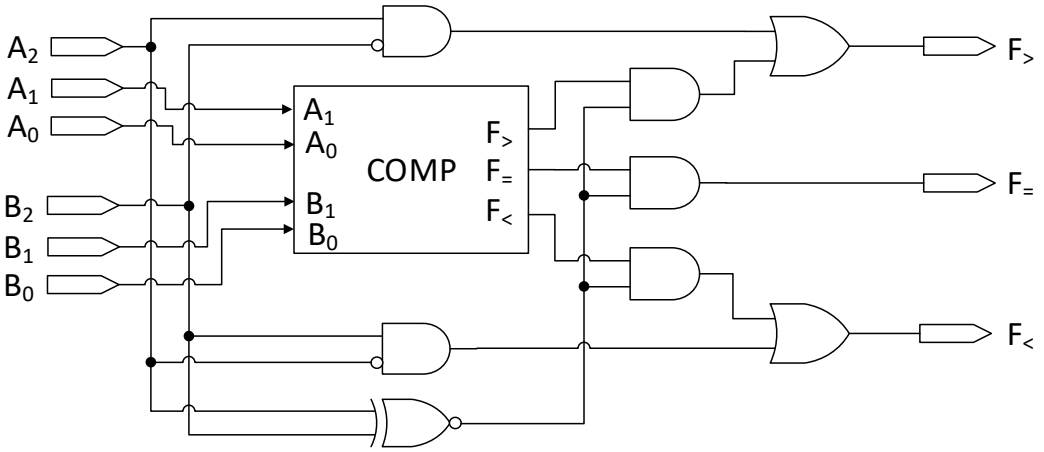
Note: Even if expressions obtained though minimization are less expensive, the minimization itself becomes exhaustive when increasing the number of input bits. Therefore the 2nd solution based on arithmetic properties is more convenient.



16. Implement – using logic gates and 2-bit unsigned comparators – a comparator, which performs >, <, = between 2 unsigned 3-bit operands.

Solution: The following arithmetic rules for unsigned numbers will be implemented:

- $F_> = 1$, if A > B, meaning $A_2 > B_2$ or ($A_2 = B_2$ and $A_{1:0} > B_{1:0}$) $\Leftrightarrow F_> = A_2 \cdot \overline{B_2} + (A_2 \odot B_2) \cdot F_{>_{1:0}}$
- $F_< = 1$, if A < B, meaning $A_2 < B_2$ or ($A_2 = B_2$ and $A_{1:0} < B_{1:0}$) $\Leftrightarrow F_< = \overline{A_2} \cdot B_2 + (A_2 \odot B_2) \cdot F_{<_{1:0}}$
- $F_= = 1$, if A = B, meaning $A_2 = B_2$ and $A_{1:0} = B_{1:0}$ $\Leftrightarrow F_= = (A_2 \odot B_2) \cdot F_{=_{1:0}}$



17. Implement – using logic gates and 2-bit unsigned comparators – a comparator, which performs the relation ≥ between 2 unsigned 3-bit operands.

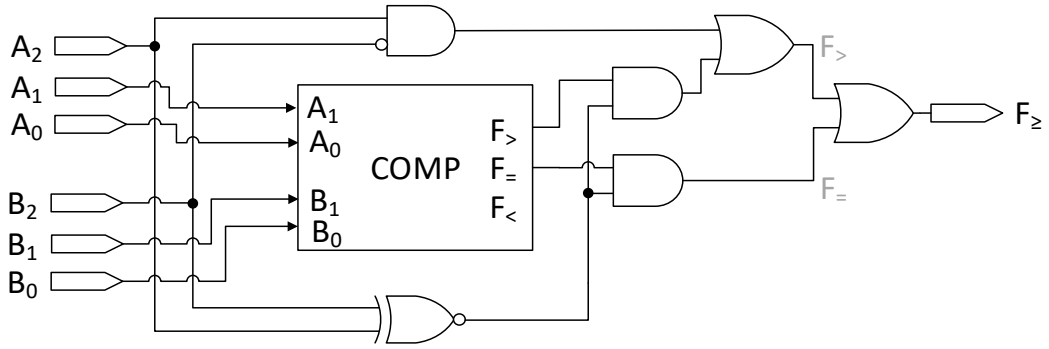
Solution:

Version 1

$F_{\geq}$ = 1, if A ≥ B, meaning A=B or A>B ⇔ $F_{\geq} = F_{>} + F_{=}$

The following arithmetic rules for unsigned numbers will be implemented:

- $F_{>}$ = 1, if A > B, meaning $A_2 > B_2$ or ($A_2 = B_2$ and $A_{1:0} > B_{1:0}$) ⇔ $F_{>} = A_2 \cdot \overline{B_2} + (A_2 \odot B_2) \cdot F_{>_{1:0}}$
- $F_{<}$ = 1, if A < B, meaning $A_2 < B_2$ or ($A_2 = B_2$ and $A_{1:0} < B_{1:0}$) ⇔ $F_{<} = \overline{A_2} \cdot B_2 + (A_2 \odot B_2) \cdot F_{<_{1:0}}$

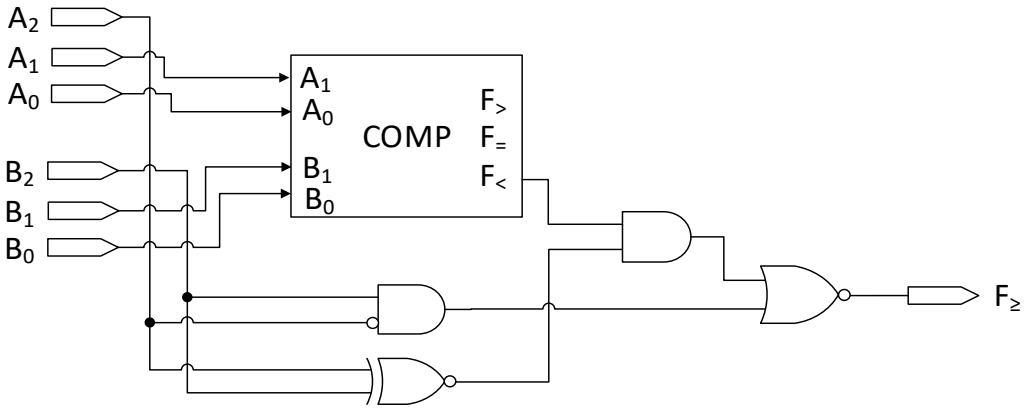


Version 2

$F_{\geq}$ = 1, if A ≥ B, meaning not(A<B) ⇔ $F_{\geq} = \overline{F_{<}}$

The following arithmetic rules for unsigned numbers will be implemented:

- $F_{<}$ = 1, if A < B, meaning $A_2 < B_2$ or ($A_2 = B_2$ and $A_{1:0} < B_{1:0}$) ⇔ $F_{<} = \overline{A_2} \cdot B_2 + (A_2 \odot B_2) \cdot F_{<_{1:0}}$

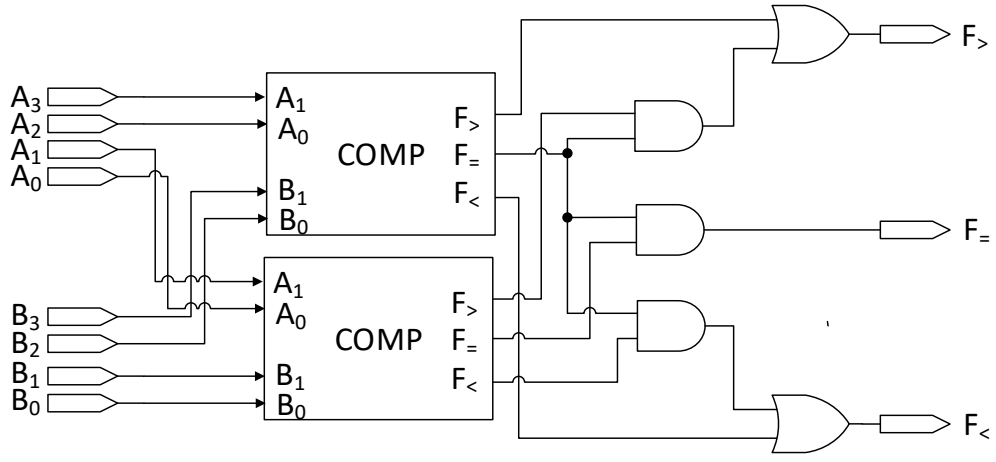


18. Implement – using logic gates and 2-bit unsigned comparators – a comparator, which performs >, <, = between 2 unsigned 4-bit operands.

Solution: The following arithmetic rules for unsigned numbers will be implemented:

- $F_> = 1$, if $A > B$, meaning $A_{3:2}>B_{3:2}$ or ($A_{3:2}=B_{3:2}$ and $A_{1:0}>B_{1:0}$) $\Leftrightarrow F_> = F_{>_{3:2}} + F_{=_{3:2}} \cdot F_{>_{1:0}}$
- $F_< = 1$, if $A < B$, meaning $A_{3:2}<B_{3:2}$ or ($A_{3:2}=B_{3:2}$ and $A_{1:0}<B_{1:0}$) $\Leftrightarrow F_< = F_{<_{3:2}} + F_{=_{3:2}} \cdot F_{<_{1:0}}$
- $F_= = 1$, if $A = B$, meaning $A_{3:2}=B_{3:2}$ and $A_{1:0}=B_{1:0}$ $\Leftrightarrow F_= = F_{=_{3:2}} \cdot F_{=_{1:0}}$

Note: One 2-bit comparator will compare the bits in range 3:2 and another 2-bit comparator will compare the bits in range 1:0.

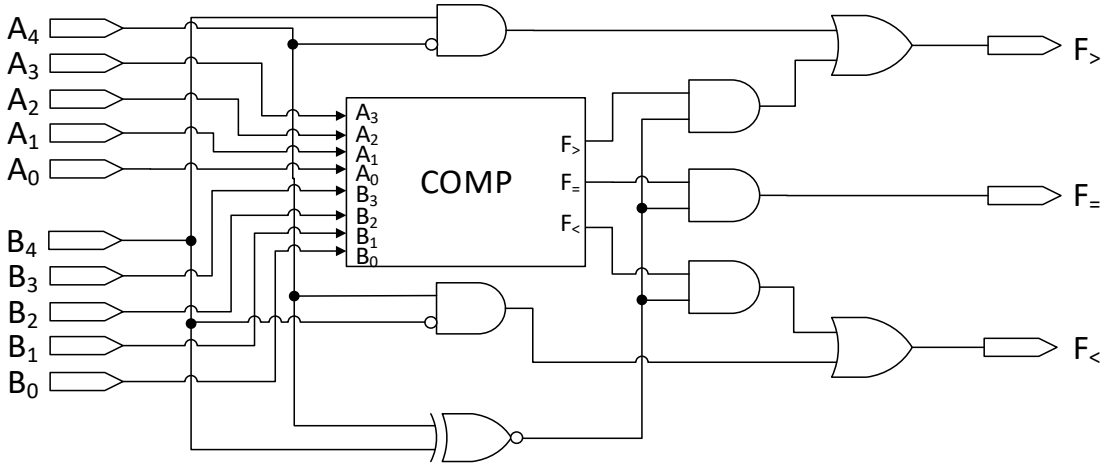


19. Implement – using logic gates and 4-bit unsigned comparators – a comparator, which performs >, <, = between 2 signed 5-bit operands in 2's Complement representation.

Solution: The following arithmetic rules for signed numbers in 2's Complement representation will be implemented:

- $F_> = 1$, if $A > B$, meaning ($A_4=0$ and $B_4=1$) or ($A_4=B_4$ and $A_{3:0}>B_{3:0}$) $\Leftrightarrow F_> = \overline{A_4} \cdot B_4 + (A_4 \odot B_4) \cdot F_{>_{3:0}}$

- $F_< = 1$, if A < B, meaning (A_4=1 and B_4=0) or (A_4=B_4 and $A_{3:0}$<$B_{3:0}$) $\Leftrightarrow F_< = A_4 \cdot \overline{B_4} + (A_4 \odot B_4) \cdot F_{<_{3:0}}$
- $F_= = 1$, if A = B, meaning A_4=B_4 and $A_{3:0}$=$B_{3:0}$ $\Leftrightarrow F_= = (A_4 \odot B_4) \cdot F_{=_{3:0}}$

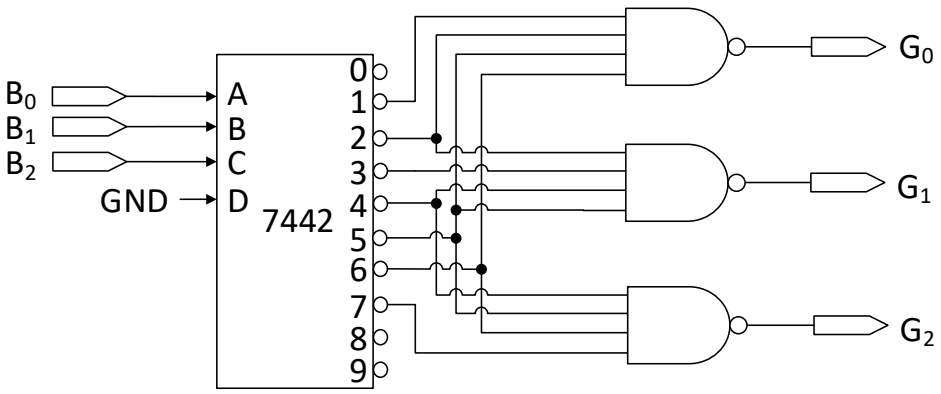


20. Implement a 3-bit BCD-to-Gray converter using the 7442 decoder and logic gates.

Solution: The implementation using a decoder requires the canonical disjunctive normal form (CDNF) or the truth table. As the output functions use the same set of variables they can be implemented with the same decoder. The 7442 BCD-to-Decimal decoder implements the inverted minterms on its outputs, therefore a NAND over the right outputs should generate a disjunctive OR (acc. De Morgan law), hence implementing a canonical disjunctive normal form. The three variables of the functions are connected to the less significant inputs of 7442. The 4th input is zeroed (GND). Relevant minterms will be found on outputs 0 to 7. All functions contain 4 minterms, consequently three additional 4-input NAND gates shall be used.

Truth table

	B_2	B_1	B_0	G_2	G_1	G_0
0	0	0	0	0	0	0
1	0	0	1	0	0	1
2	0	1	0	0	1	1
3	0	1	1	0	1	0
4	1	0	0	1	1	0
5	1	0	1	1	1	1
6	1	1	0	1	0	1
7	1	1	1	1	0	0

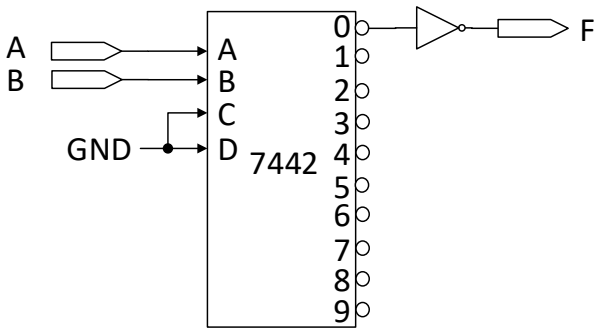


21. Implement the 2-input NOR gate using the 7442 decoder and logic gates.

Solution: The implementation using a decoder requires the canonical disjunctive normal form (CDNF) or the truth table. The 7442 BCD-to-Decimal decoder implements the inverted minterms on its outputs. Inverting the 0^{th} output would return the P_0 minterm, which is also the NOR function (acc. Truth table). Both function variables are connected to the less significant inputs of 7442. The higher ranking inputs are zeroed (GND).

Truth table

	B	A	F
0	0	0	1
1	0	1	0
2	1	0	0
3	1	1	0

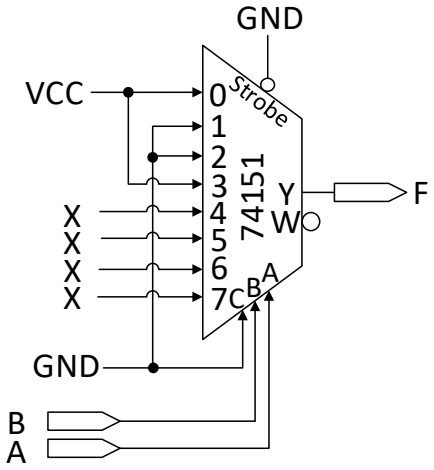


22. Implement the 2-input XNOR gate using the 74151 multiplexer.

Solution: The implementation using a multiplexer requires the expression in canonical disjunctive normal form (CDNF) or the truth table, which highlights the minterms. Both variables A, B can be connected to the less significant selections A, B of the multiplexer 74151. The remaining selection will be zeroed (GND). Therefore, the values from the Truth table can be connected to data inputs 0 to 3. As selection C = 0, inputs 4 to 7 are irrelevant (X means they are connected to either VCC or GND).

Truth table

	B	A	F
0	0	0	1
1	0	1	0
2	1	0	0
3	1	1	1



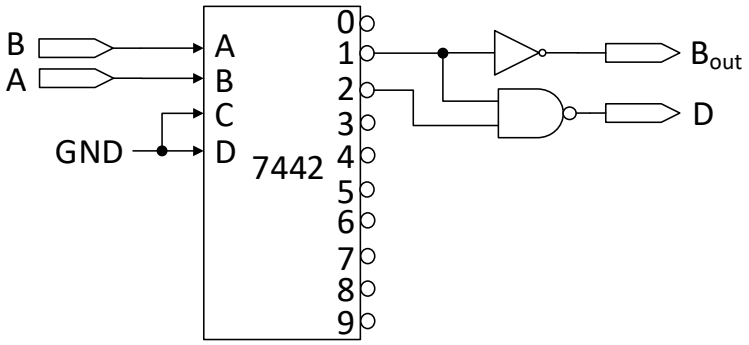
23. Implement a 1-bit half subtractor unit using the 7442 decoder and logic gates.

Solution: The implementation using a decoder requires the canonical disjunctive normal form (CDNF) or the truth table. The 7442 BCD-to-Decimal decoder implements the inverted minterms on its outputs, therefore a NAND over outputs 1 and 2 should generate a disjunctive OR (acc. De Morgan law), hence implementing the CDNF of D. Note that B_{out}

requires 1 minterm, meaning the corresponding decoder output must be inverted. The two variables of the functions are connected to the less significant inputs of 7442. The higher ranking inputs are zeroed (GND).

Truth table

	A	B	B_{out}	D
0	0	0	0	0
1	0	1	1	1
2	1	0	0	1
3	1	1	0	0

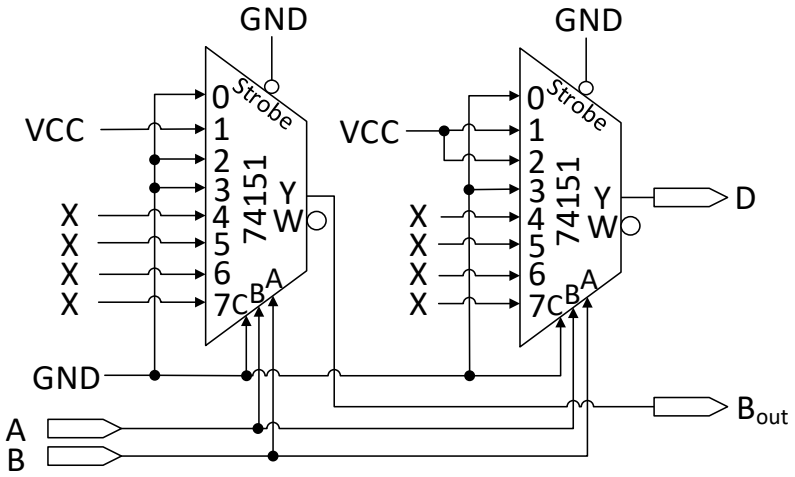


24. Implement a 1-bit half subtractor unit using two 74151 multiplexers.

Solution: The implementation using a MUX requires the expression in canonical disjunctive normal form (CDNF) or the truth table, which highlights the minterms. Both variables A, B can be connected to the less significant selections A, B of the multiplexer 74151. The remaining selection will be zeroed (GND). Therefore, the values from the Truth table can be connected to data inputs 0 to 3. As selection C = 0, inputs 4 to 7 are irrelevant (X means they are connected to either VCC or GND). The implementation of each output function requires a separate 74151 multiplexer.

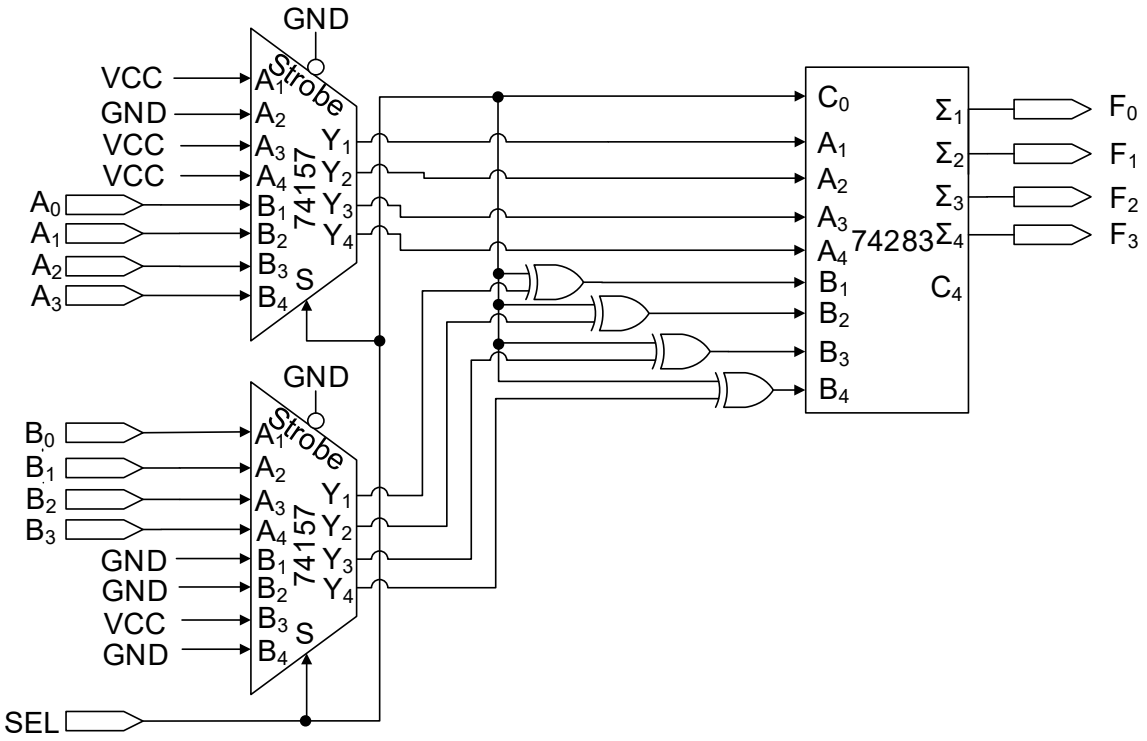
Truth table

	A	B	B_{out}	D
0	0	0	0	0
1	0	1	1	1
2	1	0	0	1
3	1	1	0	0



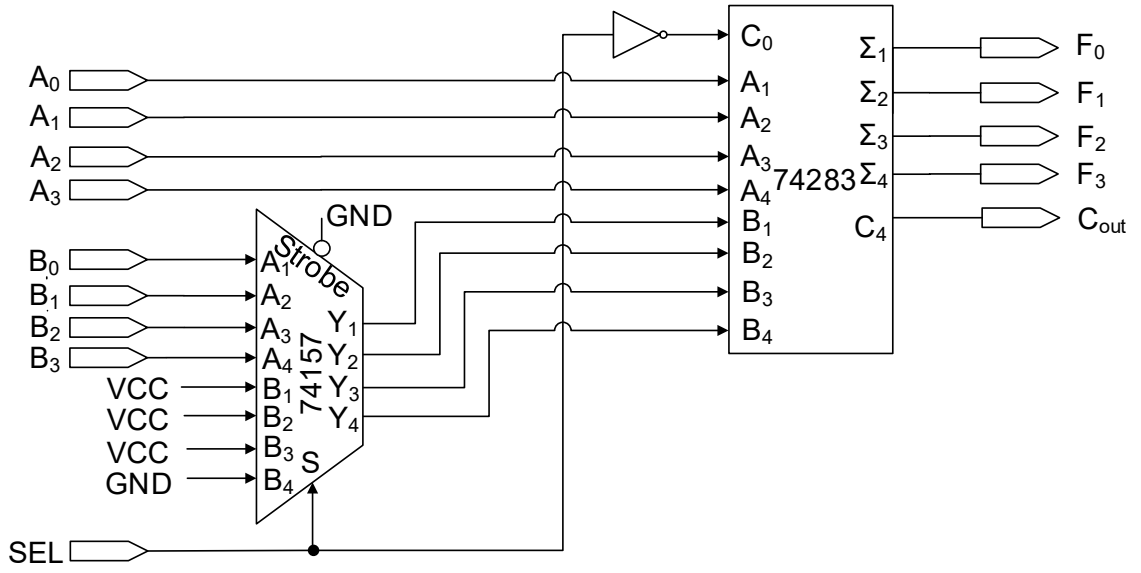
25. Implement a 4-bit ALU (Arithmetic-Logic Unit) for two operands A, B capable to perform the following operations given the selection signal SEL: 0 -> 13+B; 1 -> A-4 in 2's Complement representation. Use the 74283 adder and 74157 multiplexers.

Solution: It may be noticed both operations can be implemented with an adder-subtractor unit (circuit 74283 + XOR gates) having the inputs indicated by signal SEL. Two 74157 units (4-bit MUX 2:1) can be used to select the proper inputs of the adder-subtractor, based on the value of SEL.



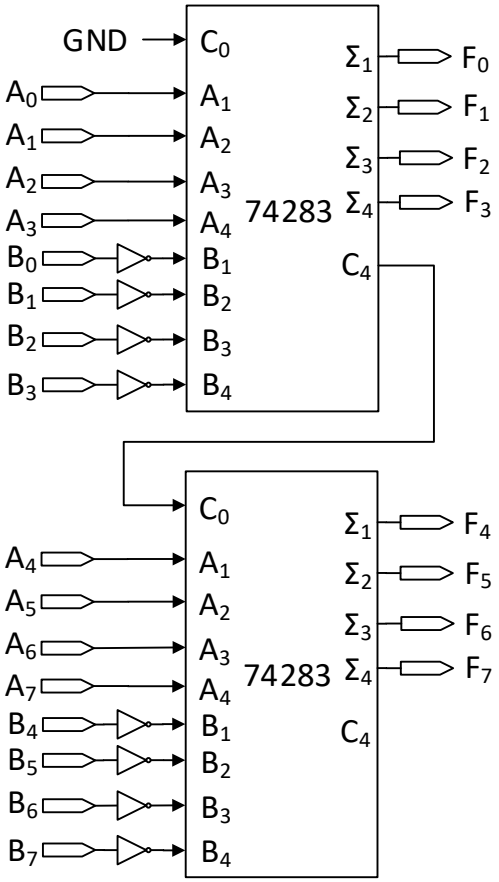
26. Implement a 4-bit ALU (Arithmetic-Logic Unit) for two operands A, B capable to perform the following operations given the selection signal SEL: 0 -> A+B+1; 1 -> A+7. Use the 74283 adder and the 74157 multiplexer.

Solution: It may be noticed both operations can be implemented with a 74283 adder having the 2nd input term indicated by SEL, as follows: it is B when SEL=0, and it is 0111_2 when SEL=1. A 74157 unit (4-bit MUX 2:1) can be used to select the proper 2nd term. The +1 increment from A + B + 1 can be added by connecting $\overline{SEL}$ to C_0. Consequently, when SEL=0, C_0 becomes 1 and the increment (+1) is enabled. When SEL = 1, C_0 becomes 0, maintaining the result unaffected.



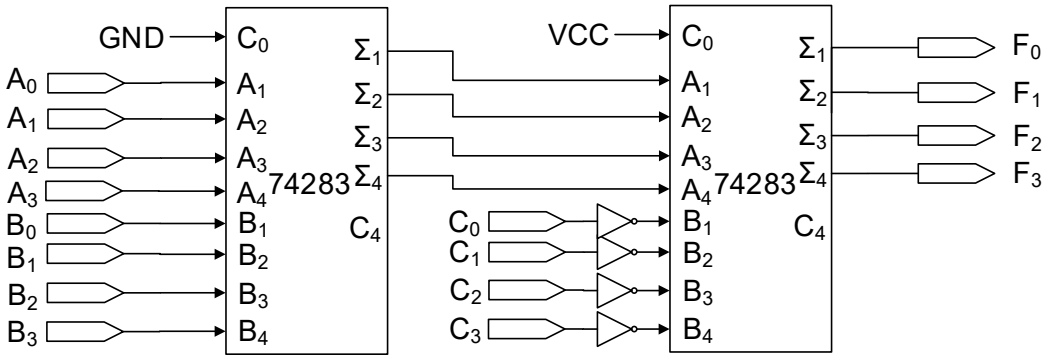
27. Implement an 8-bit arithmetic unit for two operands in 2's Complement representation, which calculates A - B - 1 using two 74283 adders and basic logic gates.

Solution: In 2's Complement representation A - B - 1 may be rewritten as: A - B - 1 = A + $\overline{B}$ + 1 - 1 = A + $\overline{B}$. The expression can be implemented using two 74283 adders and several inverters. C_0 will be connected to 0 (GND).



28. Implement a 4-bit arithmetic unit for two operands in 2's Complement representation, which calculates A + B - C using two 74283 adders and basic logic gates.

Solution: In 2's Complement representation A + B - C may be rewritten as: A + B - C = A + B + $\overline{C}$ + 1. The expression can be implemented using two 74283 adders and several inverters. The first C_0 will be connected to 0 (GND) and the second C_0 will be connected to 1 (VCC), to generate the final increment.



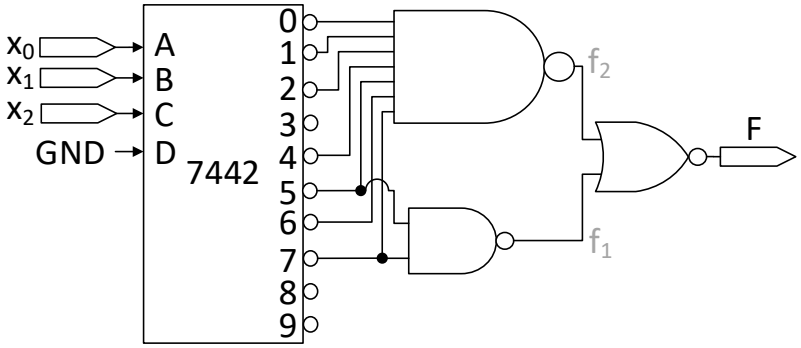
29. Design a circuit which implements $F(x_2, x_1, x_0) = \overline{f_1(x_2, x_1, x_0) + f_2(x_2, x_1, x_0)}$, where $f_1(x_2, x_1, x_0)=\sum(5, 7)$ and $f_2(x_2, x_1, x_0) = \overline{\overline{x_1} \cdot \overline{x_2}} + \overline{x_0 \cdot x_1}$. Use a 7442 decoder and logic gates to implement f_1 and f_2.

Solution:

The implementation using a decoder requires the canonical disjunctive normal forms (CDNF) or the truth tables. The minterms of f_1 are known, consequently its CDNF is straightforward. The truth table shall be calculated for f_2 to identify its minterms. The 7442 BCD-to-Decimal decoder implements the inverted minterms on its outputs, therefore a NAND over the right outputs should generate a disjunctive OR (acc. De Morgan law), hence implementing the CDNF. The three variables of the functions are connected to the less significant inputs of 7442. The higher ranking input is zeroed (GND). Each function is implemented using separate NAND gate. Finally, a NOR gate generates F.

Truth table of f_2

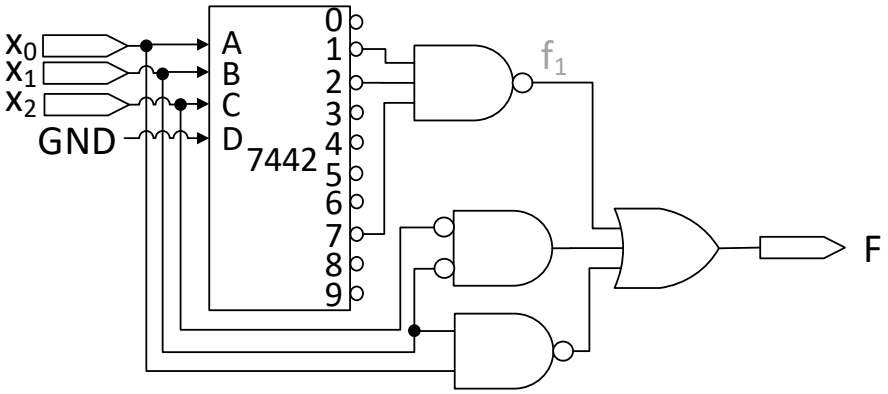
	x_2	x_1	x_0	f_2
0	0	0	0	1
1	0	0	1	1
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	0	1	1
7	1	1	1	1



30. Design a circuit which implements $F(x_2, x_1, x_0) = f_1(x_2, x_1, x_0) + \overline{x_1} \cdot \overline{x_2} + \overline{x_0 \cdot x_1}$, where $f_1(x_2, x_1, x_0)=\sum(1, 2, 7)$. Use a 7442 decoder and logic gates to implement f_1.

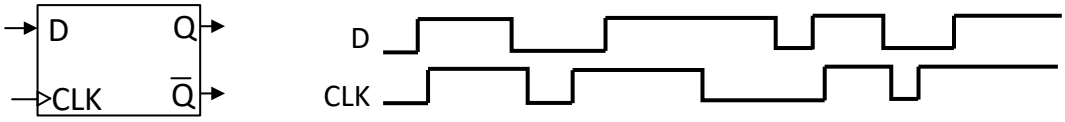
Solution:

The implementation using a decoder requires the canonical disjunctive normal form (CDNF) or the truth table. The minterms of f_1 are known, consequently its CDNF is straightforward. The 7442 BCD-to-Decimal decoder implements the inverted minterms on its outputs, therefore a NAND over the right outputs should generate a disjunctive OR (acc. De Morgan law), hence implementing the CDNF of f_1. The rest of F can be implemented using basic logic gates. The three variables of the functions are connected to the less significant inputs of 7442. The higher ranking input is zeroed (GND).


x_0
x_1
x_2
GND
A
B
C
D
7442
0
1
2
3
4
5
6
7
8
9
f_1
F

B. Annex 2 – Problems with sequential logic circuits

1. Highlight the outputs of the D flip-flop having the input sequence below.

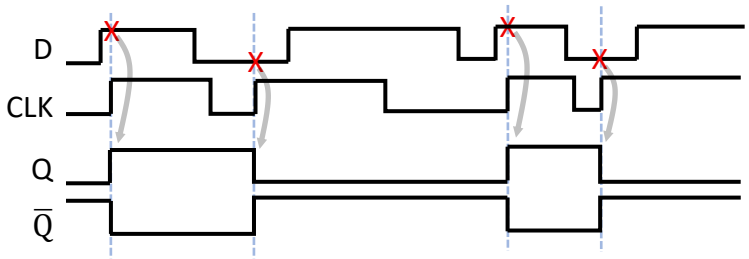


Solution:

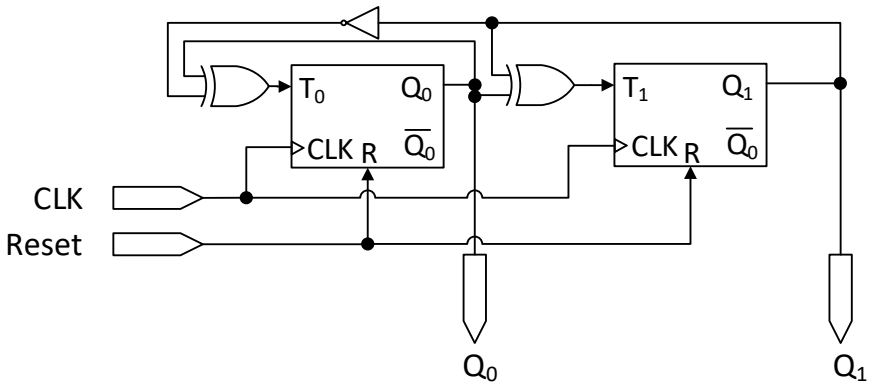
The D flip-flop in the figure triggers on the rising edge. At the time of the rising edge the flip-flop copies the input value D on its output Q. The output $\overline{Q}$ carries the inverted value of Q.

Truth table

D	Q^{t+1}
0	0
1	1

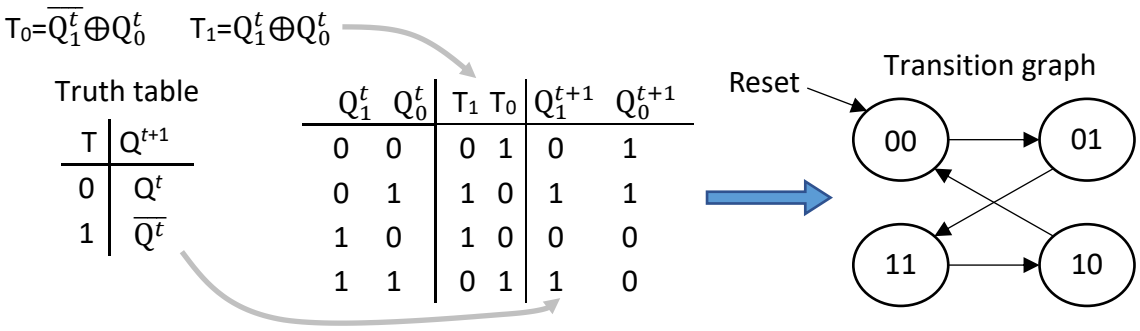


2. Highlight the transition graph for the circuit in the figure.

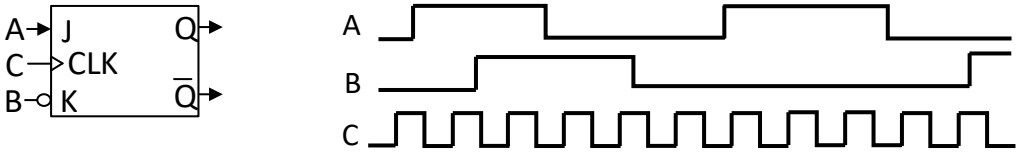


Solution:

The boolean expressions for the flip-flop inputs T_1, T_0 must be determined from the logic diagram and then a truth table can be generated containing pairs of all current states $Q_1^t Q_0^t$ and corresponding values resulted on T_1, T_0. Consequently, the next state $Q_1^{t+1} Q_0^{t+1}$ is the result of matching the T flip-flop truth table with the pairs (Q_1^t, T_1), (Q_0^t, T_0) in each row. The resulted pairs of current state and next state ($Q_1^t Q_0^t$, $Q_1^{t+1} Q_0^{t+1}$) represent arcs in a transition graph whose nodes are all possible state values.



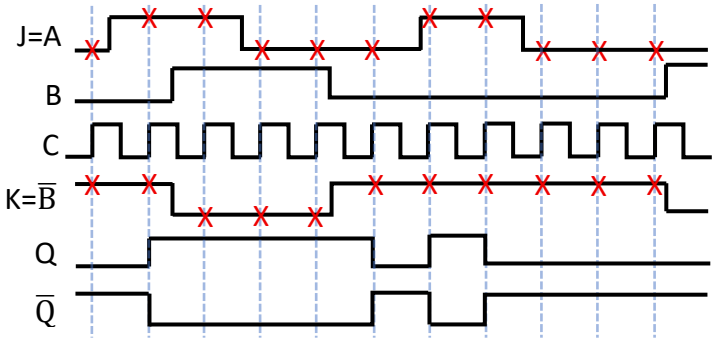
3. Highlight the outputs of the JK flip-flop having the input sequence below.



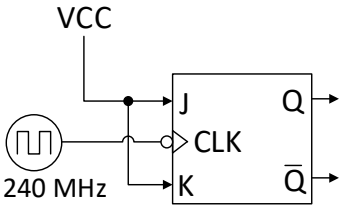
Solution: The JK flip-flop in the figure triggers on the rising edge of C and changes its state according to truth table below. The output $\overline{Q}$ carries the inverted value of Q. The values of A and B (marked with X) have effect on the flip-flop state at the rising edge.

Truth table

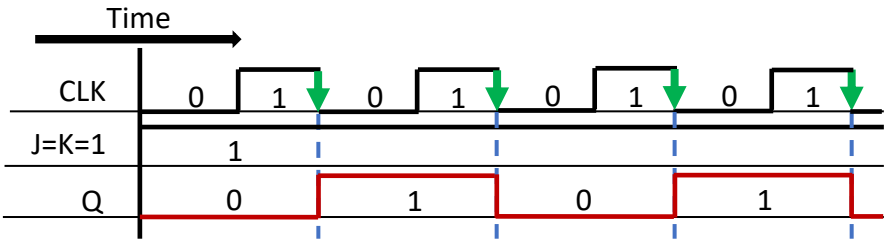
J K	Q^{t+1}
0 0	Q^t
0 1	0
1 0	1
1 1	$\overline{Q^t}$



4. Considering the frequency of the input clock signal below is 240Hz, what would be the frequency of the signal generated on the output Q of the JK flip-flop in the figure?



Solution: The JK flip-flop with inputs connected to 1 (VCC) toggles its value every falling edge of the clock signal, as highlighted in the timing diagram below. The resulting output of Q takes the form of a clock signal with double period than C and half frequency (works as a frequency divider with a factor of 2). Therefore, the frequency on the output Q is 240/2 = 120Hz.



5. Implement a JK flip-flop using a D flip-flop, a MUX 2:1 and a NOT gate.

Solution: Identical pairs of current and next state are matched in the excitation tables for JK and D flip-flops, respectively. A truth table results containing current state Q^t and J, K values on the left side, and D values on the right side. Based on the values from the truth table, the expression of D can be minimized using the Karnaugh map.

Excitation table for JK

Q^t	Q^{t+1}	J K
0	0	0 X
0	1	1 X
1	0	X 1
1	1	X 0

Excitation table for D

Q^t	Q^{t+1}	D
0	0	0
0	1	1
1	0	0
1	1	1

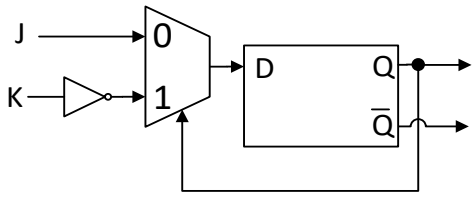
Q^t	J	K	D
0	0	X	0
0	1	X	1
1	X	1	0
1	X	0	1

D:

Q^t \ J K	00	01	11	10
0	0	0	1	1
1	1	0	0	1

$$D = (\overline{Q^t} \cdot J) + (Q^t \cdot \overline{K})$$

The expression of D can be implemented using a MUX 2:1 having Q^t connected to the selection input and J, $\overline{K}$ connected to the data inputs:



6. A PN flip-flop has 4 operations for any combination of its P, N inputs: 00 → reset, 01 → hold, 10 → toggle; 11 → set. Highlight the following properties of a PN flip-flop: a) The truth table; b) The characteristic equation (expression of Q^{t+1}); c) The excitation table; d) Generate a PN flip-flop using a D flip-flop and logic gates.

Solution:

a) Truth table

Q^t	P	N	Q^{t+1}
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

b) Characteristic equation

Q^{t+1}:

Q^t \ P N	00	01	11	10
0	0	0	1	1
1	0	1	1	0

$$D = (\overline{Q^t} \cdot P) + (Q^t \cdot N)$$

c) Excitation table

Q^t	Q^{t+1}	P N
0	0	0 X
0	1	1 X
1	0	X 0
1	1	X 1

d) Identical pairs of current and next state are matched in the excitation tables for PN and D flip-flops, respectively. A truth table results containing current state Q^t and P, N values on the left side, and D values on the right side. Based on the values from the truth table, the expression of D can be minimized using the Karnaugh map.

Excitation table for PN

Q^t	Q^{t+1}	P N
0	0	0 X
0	1	1 X
1	0	X 0
1	1	X 1

Excitation table for D

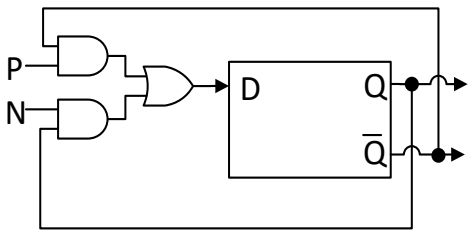
Q^t	Q^{t+1}	D
0	0	0
0	1	1
1	0	0
1	1	1

Q^t	J	K	D
0	0	X	0
0	1	X	1
1	X	0	0
1	X	1	1

D:

Q^t \ P N	00	01	11	10
0	0	0	1	1
1	0	1	1	

$$D = (\overline{Q^t} \cdot P) + (Q^t \cdot N)$$

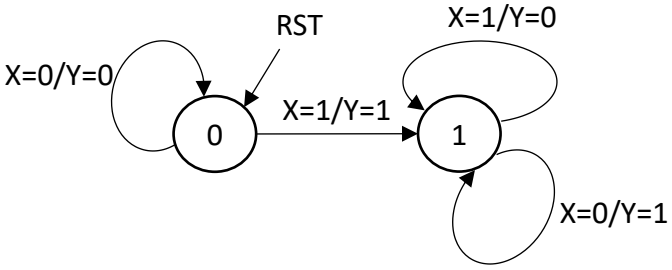


7. Implement a circuit which generates the 2's Complement of a binary number. The circuit must have a serial input X, an asynchronous reset, a clock and a serial output Y. The number is received on X and the 2's Complement is generated on Y, one bit per clock

cycle. The input sequence starts with the least significant bit and ends with the most significant bit when enabling the reset. A new number starts after disabling the reset.

Solution:

The 2's Complement algorithm starts from the least significant bit until the first bit of 1. While these bits remain unchanged, the following bits are inverted. Therefore, the resulting circuit shall have 2 states: one state will output the input value (Y = X) and the other state will invert the input value (Y = $\overline{X}$). A flip-flop unit is sufficient to store the current state. A state of 0 means output Y = X and a state of 1 means output Y = $\overline{X}$. A *reset* sets the state to 0, and a transition from 0 to 1 takes place when X = 1. Using these rules, it is possible to implement the *transition graph*. The truth table and the boolean expressions for flip-flop's next state and for the output Y can be generated using the transition graph.



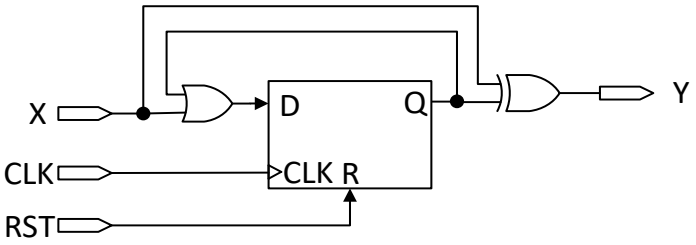
Q^t	X	Q^{t+1}
0	0	0
0	1	1
1	0	1
1	1	1

$$Q^{t+1} = Q^t + X$$

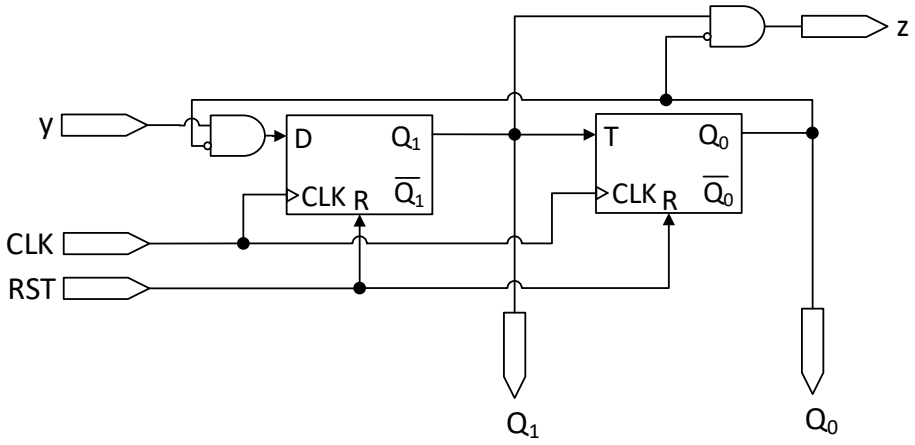
Q^t	X	Y
0	0	0
0	1	1
1	0	1
1	1	0

$$Y = Q^t \oplus X$$

The D flip-flop was chosen to store the state. Knowing the characteristic equation for D flip-flop is $Q^{t+1} = D$ it means $D = Q^t + X$, hence the logic diagram for the resulting circuit will be as follows:



8. Highlight the transition graph for the circuit below having an input y and an output z.



Solution: The boolean expressions for the D and T flip-flop inputs must be determined from the logic diagram. Then a truth table can be generated containing all pairs of input value and current state (y Q_1^t Q_0^t), and the corresponding effect on commands D and T. Consequently, the next state ($Q_1^{t+1}Q_0^{t+1}$) can be calculated using the truth tables of D and T flip-flops and the pairs (Q_1^t, D), (Q_0^t, T), respectively. The output z can be calculated from the current state (Q_1^t, Q_0^t). The next state and the output are added as columns to the table. The resulted pairs of current state and next state ($Q_1^tQ_0^t$, $Q_1^{t+1}Q_0^{t+1}$) represent arcs oriented from current state to the next, while the nodes represent all possible states. The transitions are conditioned by the value of y. The output z may appear attached to arcs or attached to nodes (states). The outputs are highlighted only when active (when z=1).

$D=\overline{Q_0^t}\cdot y$ $T=Q_1^t$

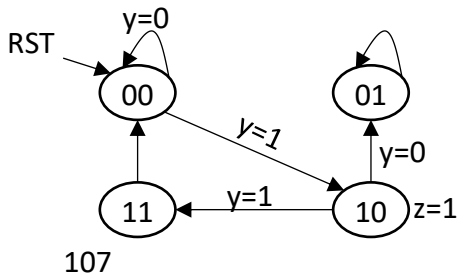
Truth tables

D	Q^{t+1}
0	0
1	1

T	Q^{t+1}
0	Q^t
1	$\overline{Q^t}$

y	Q_1^t	Q_0^t	D	T	Q_1^{t+1}	Q_0^t	z
0	0	0	0	0	0	0	0
1	0	0	1	0	1	0	0
0	0	1	0	0	0	1	0
1	0	1	0	0	0	1	0
0	1	0	0	1	0	1	1
1	1	0	1	1	1	1	1
0	1	1	0	1	0	0	0
1	1	1	0	1	0	0	0

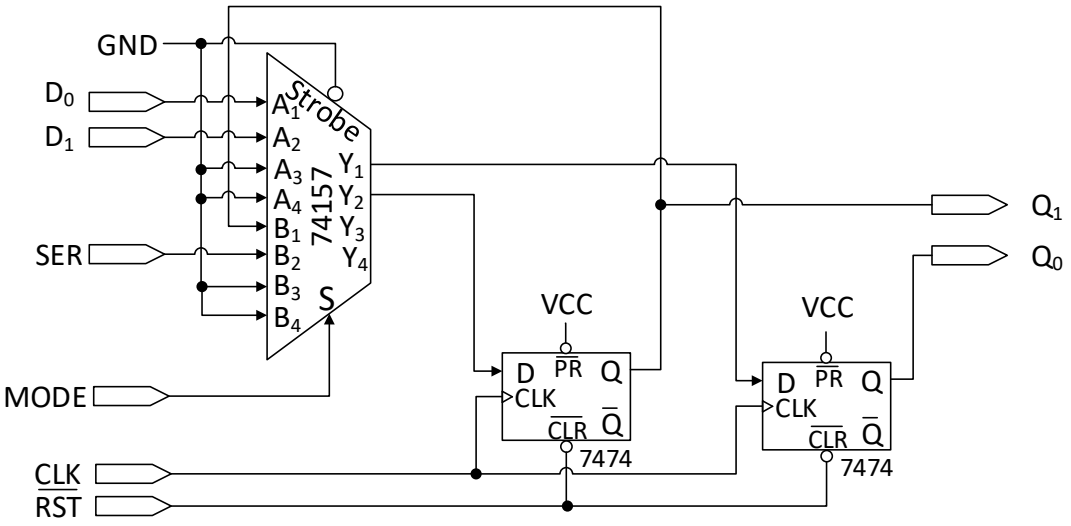
Transition graph



9. Using 7474 D flip-flops and the 74157 multiplexer, implement a 2-bit shift register with the following operations: if input signal MODE = 0 the register performs a parallel load from inputs D_1D_0; if MODE = 1 the register performs right shifting with serial load from input SER. The register is synchronous on the rising edge.

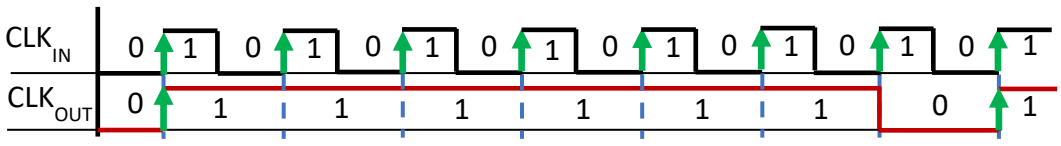
Solution:

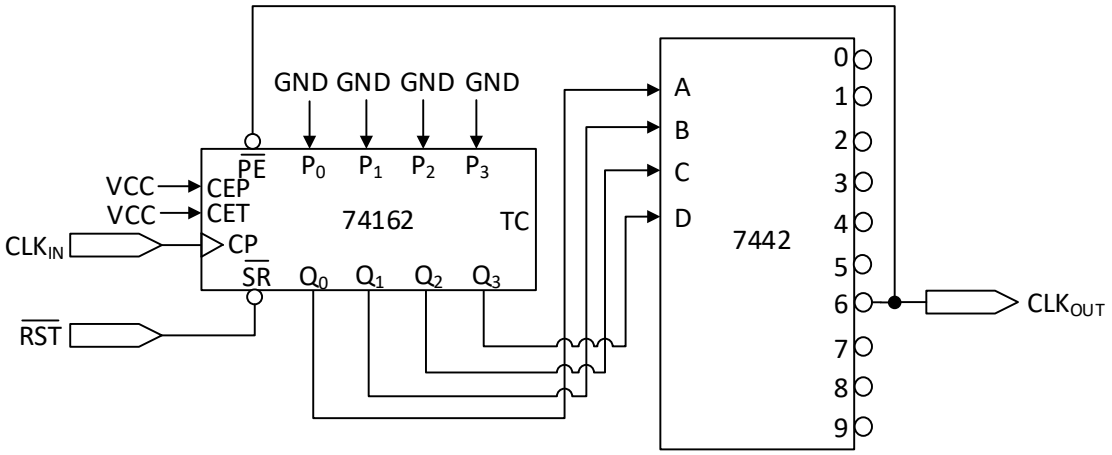
Circuit 7474 contains two D flip-flops. The operations of the register can be implemented using a 2-bit MUX 2:1. Circuit 74157 contains a 4-bit MUX 2:1, hence only 2 bits (the least significant) will be used. The rest of the data bits are zeroed (GND).



10. Implement a frequency divider with a factor of 7 using the 74162 counter and the 7442 decoder. The circuit should have a reset command.

Solution: The circuit should generate an output clock signal CLK_{OUT} with a frequency 7 times less than the frequency of input clock signal CLK_{IN}, meaning the period should be 7 times higher. As the duty cycle of CLK_{OUT} is unconstrained it can follow any ratio. For instance, CLK_{OUT} can be 0 once in seven CLK_{IN} cycles. Such a waveform can be generated using a direct counter with ascending counting loop from 0 to 6. The CLK_{OUT} will be 0 when the counter value is 6, and will be 1 otherwise. The BCD to Decimal decoder 7442 can be used to decode the counter value. The counting is implemented using the decade counter 74162.





11. Implement a 3-bit right-shift register using MUX 4:1 units and T flip-flops. The register is synchronous on the rising edge and has a serial input SER.

Solution:

A regular implementation of the shift register uses D flip-flops. Therefore, D flip-flops must be implemented with T flip-flops and logic gates. To such purpose, identical pairs of current and next state are matched in the excitation tables for D and T flip-flops, respectively. The result is a truth table containing current state Q^t and D values on the left side, and T values on the right side. The expression of T can be implemented using a MUX 4:1 having Q^t and D connected to the selection inputs, and having the data inputs connected to values from the T column of the truth table.

Excitation table for D

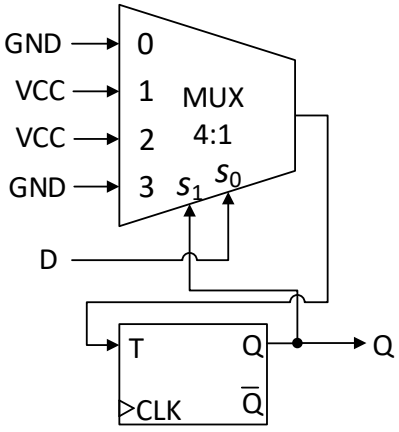
Q^t	Q^{t+1}	D
0	0	0
0	1	1
1	0	0
1	1	1

Excitation table for T

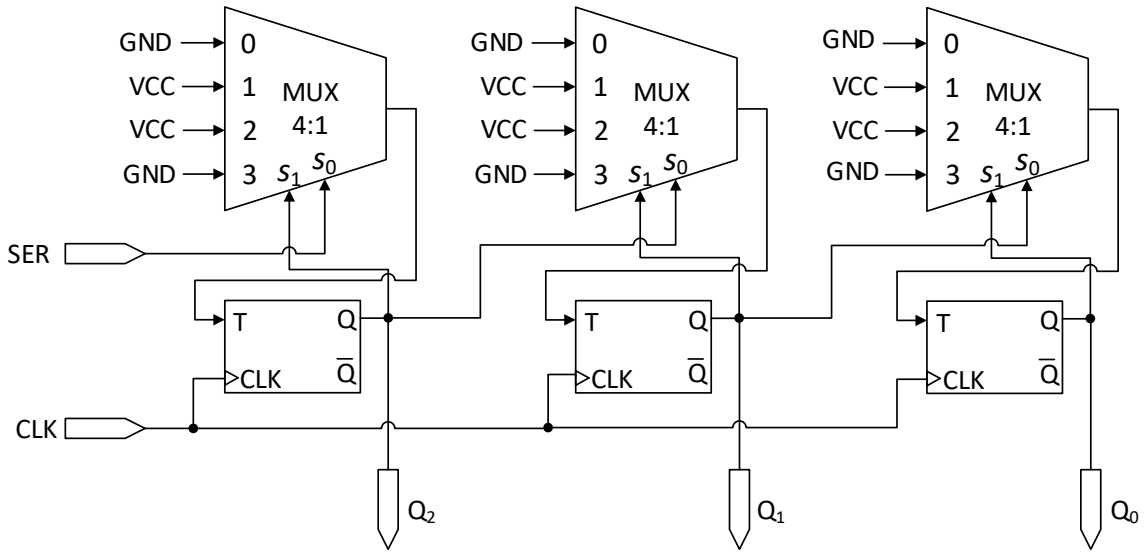
Q^t	Q^{t+1}	T
0	0	0
0	1	1
1	0	1
1	1	0

Q^t	D	T
0	0	0
0	1	1
1	0	1
1	1	0

D flip-flop implemented using T flip-flop and MUX 4:1



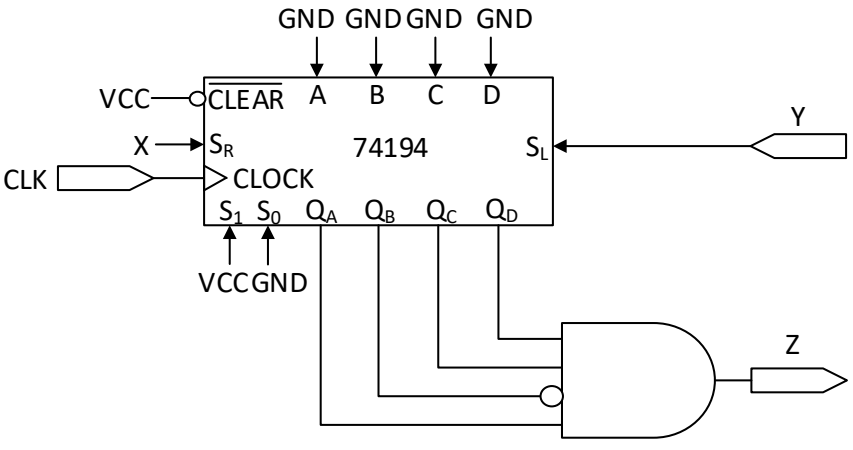
Having a T-based D flip-flop, the implementation of a 3-bit right-shift register is straightforward:



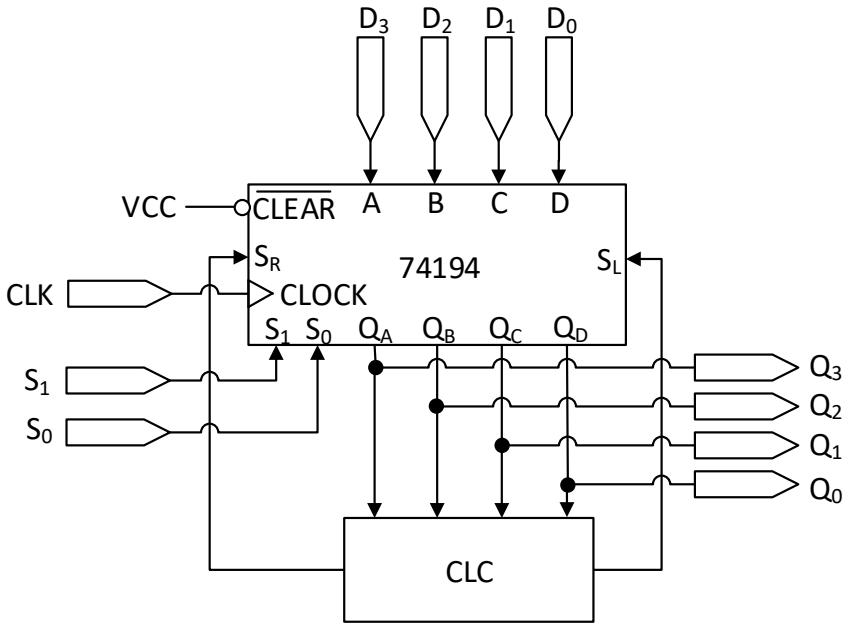
12. Implement a sequence detector using the 74194 bidirectional universal shift-register. The sequence is serially loaded on input Y (1 bit per clock impulse) starting with its most significant bit (left side). The output Z of the circuit will be set to 1 when the last 4 serially loaded input bits generate the sequence "1011", and will be set to 0 otherwise.

Solution:

The register can be implemented using circuit 74194 configured for left-shifting with serial load on S_L. The presence of the sequence "1011" inside the register can be decoded and signaled on output Z using a 4-input AND gate.



13. Determine the output sequence on $Q_{3:0}$ generated by the sequence generator below, considering that first clock impulse loads the register with the binary value $D_{3:0}$ = "1011" and left shift operation is enabled afterwards. $S_L=\overline{Q_A} \cdot Q_D$ and S_R=0.



(CLC – Combinational Logic Circuit)

Solution:

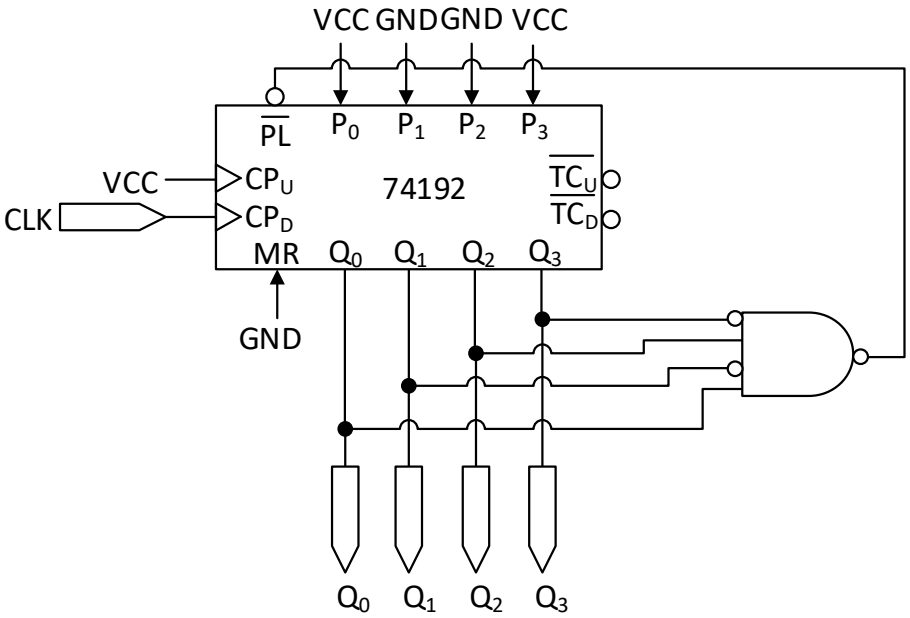
For each clock impulse, the 3 least significant bits (right side) move one position to the left. The new value of the least significant bit will be the S_L function, which depends on the current state bits.

The sequence generated will be: 1011 (initialization), 0110, 1100, 1000, 0000, 0000, … (value 0000 will repeat indefinitely).

14. Implement a counter with decreasing counting loop, from 9 to 6, using the circuit 74192.

Solution:

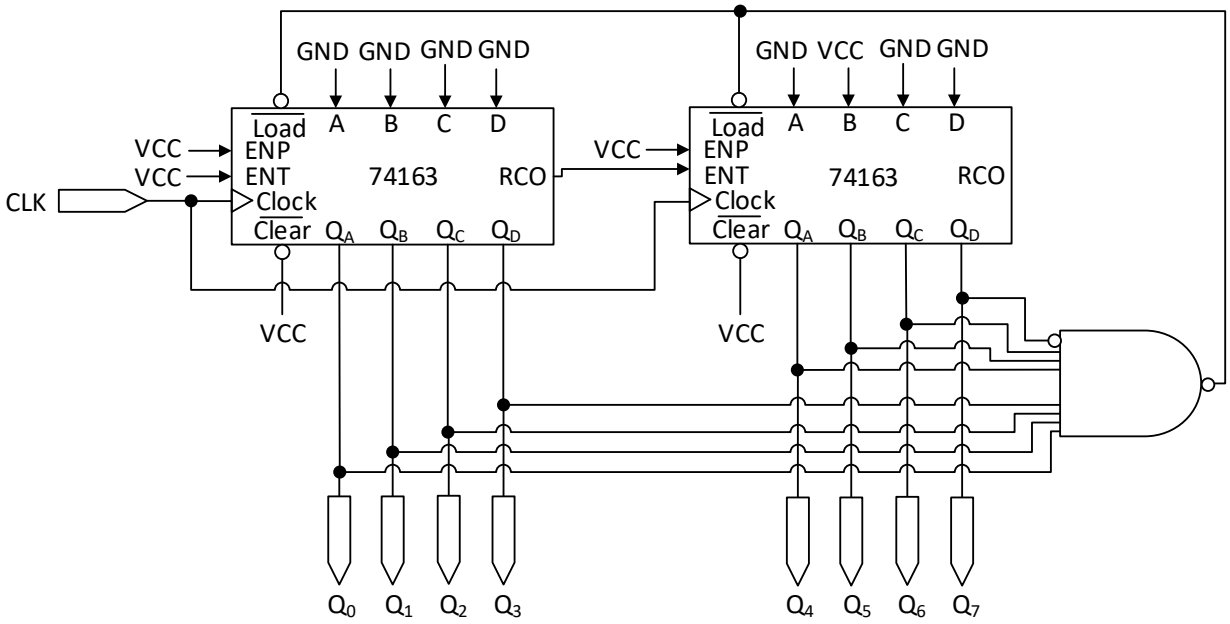
Circuit 74192 is a 4-bit synchronous reversible decade counter with asynchronous parallel load command. A value of 9 (in binary = 1001) will be loaded on the parallel input data lines, because it is the first value in the counting loop. The load command must be delayed for 1 clock impulse after value 6 is reached, because the parallel load command is asynchronous and overwrites the current state immediately. Therefore, the load command is initiated when the counter value reaches 5. For count-down configuration, the count up clock input CP_U is connected to 1 (VCC) and the reset is deactivated with 0 (GND).



15. Implement a synchronous binary counter with ascending counting loop, from 32 to 127, using two cascaded 74163 counters.

Solution:

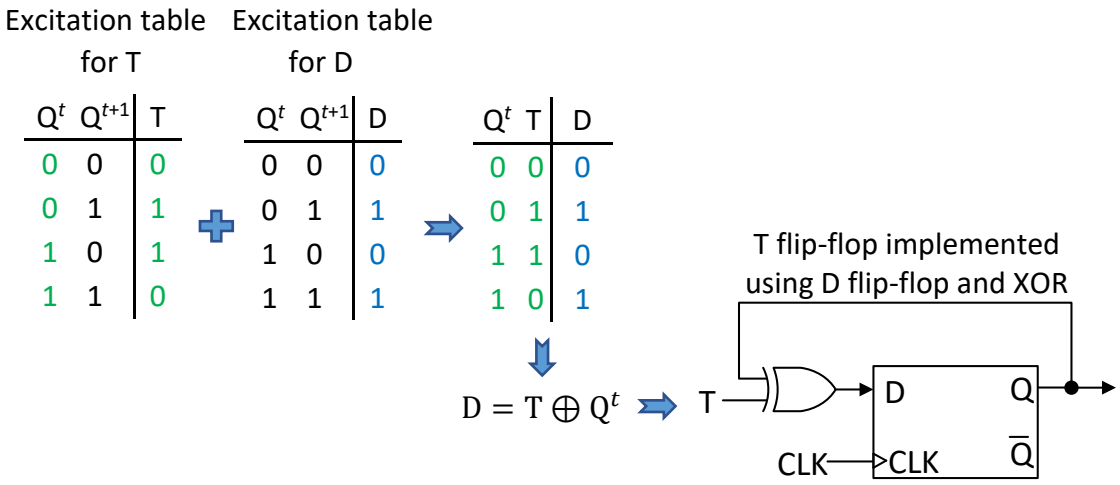
Circuit 74163 is a 4-bit synchronous binary counter with synchronous load command. Two such circuits must be cascaded to obtain an 8-bit counter with extended counting range. A value of 32 (in binary = 0010 0000) will be loaded on the parallel input data lines, because it is the first value in the counting loop. The load command is initiated when the counter value reaches 127. The value 127 (in binary = 0111 1111) is decoded using an 8-input NAND gate (AND + NOT for active low). The reset of the counters is deactivated with 1 (VCC).



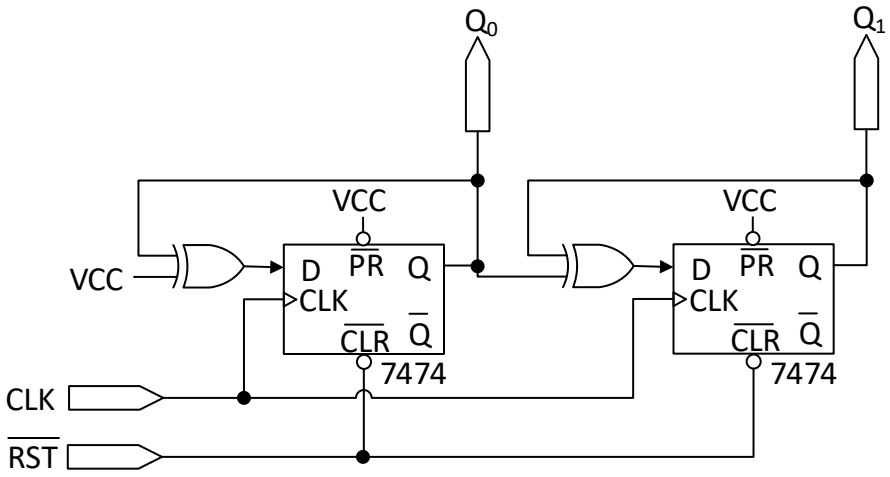
16. Implement a 2-bit count-up synchronous binary counter with asynchronous reset, using 7474 D flip-flops. The counter is triggered on the rising edge of the clock signal.

Solution:

A regular implementation of the shift register uses T flip-flops. Therefore, T flip-flops must be implemented with D flip-flops and logic gates. Identical pairs of current and next state are matched in the excitation tables for T and D flip-flops, respectively. A truth table results containing current state Q^t and T values on the left side, and D values on the right side. Based on the values from the truth table, the expression of D is the XOR operation.



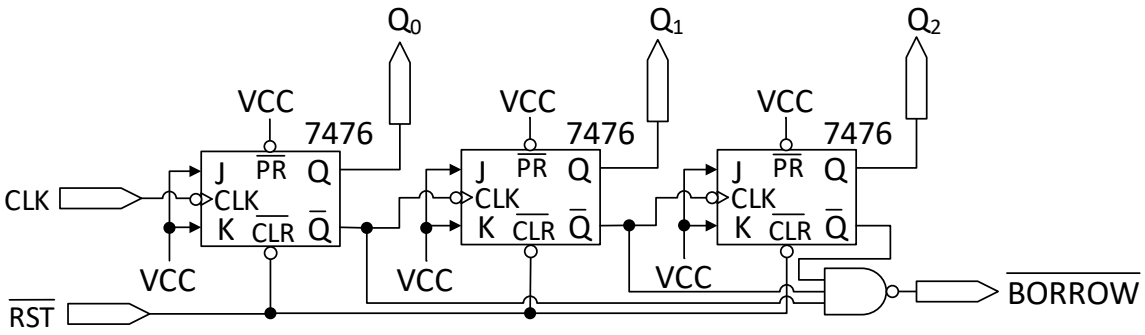
Having a D-based T flip-flop, the implementation with 7474 of a 2-bit synchronous counter is straightforward:



17. Implement a 3-bit count-down asynchronous counter with asynchronous reset, using 7476 JK flip-flops. The counter is triggered on the falling edge of the clock signal and must have an active-low $\overline{\text{BORROW}}$ signal.

Solution:

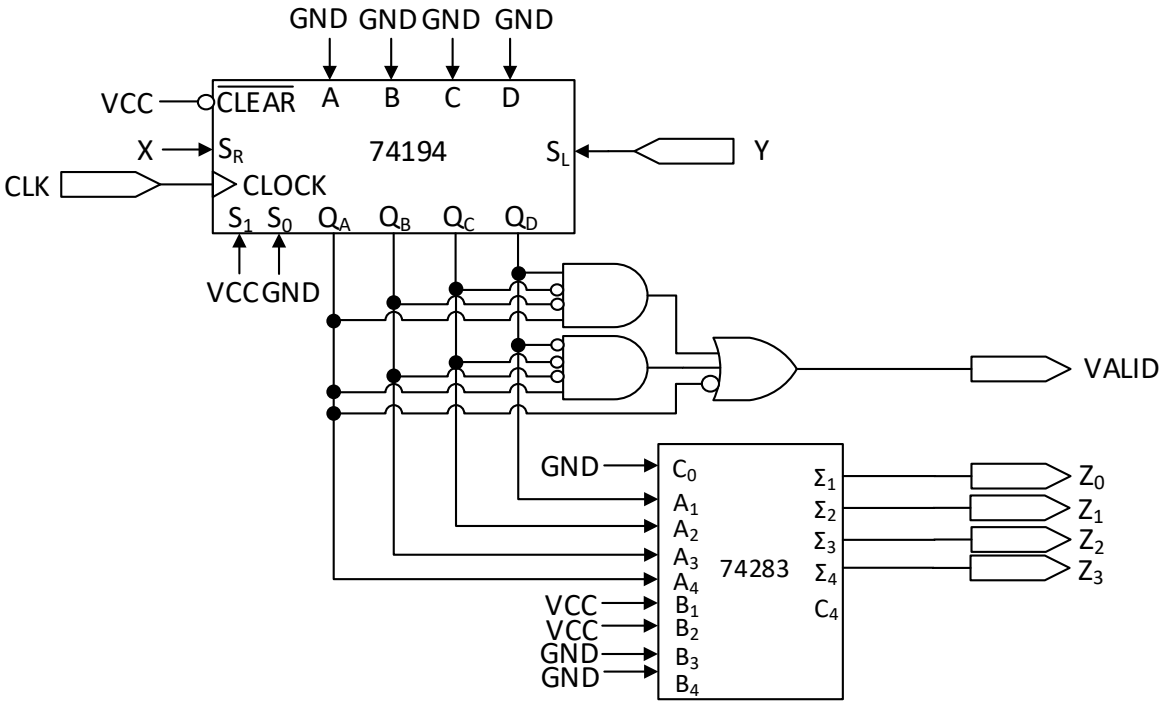
A regular implementation of the counter will use toggling JK flip-flops (J=K=1). The clock signal is connected to the less significant flip-flop, and the other flip-flops will have their clock signals connected to the inverted output of the immediately less significant flip-flop. Hence the least significant flip-flop will toggle every clock impulse, and the rest will toggle when the previous flip-flop transitions from 1 to 0. Finally, a NAND over the flip-flop outputs will implement the active-low $\overline{\text{BORROW}}$ signal, which will detect the presence of 000. $\overline{\text{BORROW}} = \overline{\overline{Q_2} \cdot \overline{Q_1} \cdot \overline{Q_0}}$.



18. Using the 74194 shift-register, the 74283 adder and basic logic gates, implement a circuit which receives a sequence of bits. The sequence is serially loaded on input Y (1 bit per clock impulse). The circuit will output on $Z_{3:0}$ the Excess 3 equivalent of the binary value present on the most recently loaded 4 bits. An additional VALID output will be set to 1 if the value from the register is lower than 10_{10}, otherwise it will be set to 0. This output will validate the Excess 3 code, which is relevant for values between 0 and 9.

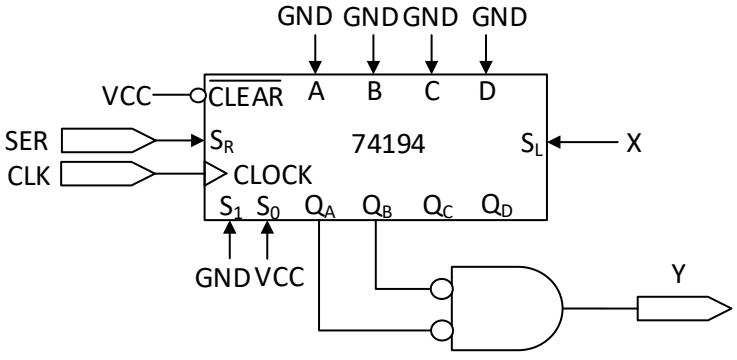
Solution:

The register can be implemented using circuit 74194 configured for left-shifting with serial load on S_L. The Excess 3 code can be calculated using the 74283 adder, which will add the value from the register with 0011_2 (3 in decimal). The value from the register is below 10_{10} if it is 9, or 8 or if the most significant bit is 0 (for numbers less than 8). Therefore, VALID = $Q_A \cdot \overline{Q_B} \cdot \overline{Q_C} \cdot Q_D + Q_A \cdot \overline{Q_B} \cdot \overline{Q_C} \cdot \overline{Q_D} + \overline{Q_A}$.



19. Using the 74194 shift-register and basic logic gates, implement a circuit which sets its output Y=1 when the most recently loaded two bits are equal to 0. The register must be set on right shifting mode with serial load (1 bit per clock impulse) from an input called SER.

Solution: The register can be implemented with circuit 74194 configured for right-shifting with serial load on S_R. In this case the most recently loaded bits are present on outputs Q_A and Q_B. A 2-input AND gate can be used to check whether these outputs are 00_2. Consequently, the expression for Y is $Y = \overline{Q_A} \cdot \overline{Q_B}$ (the P_0 minterm).



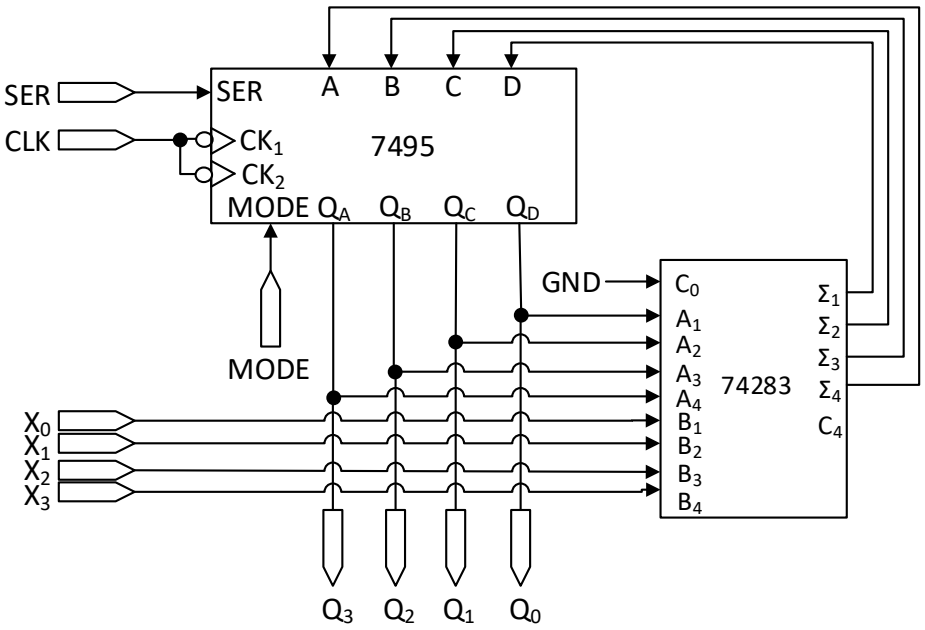
20. Using shift register 7495, implement a circuit having two operations with respect to the value set on MODE:

- MODE = 0 – the register performs right shift with serial load from an input named SER;
- MODE = 1 – the register performs parallel load with the sum between the register value and $X_{3:0}$, calculated using adder 74283 (with carry-out output ignored).

The outputs of the register are linked to output terminals $Q_{3:0}$.

Solution:

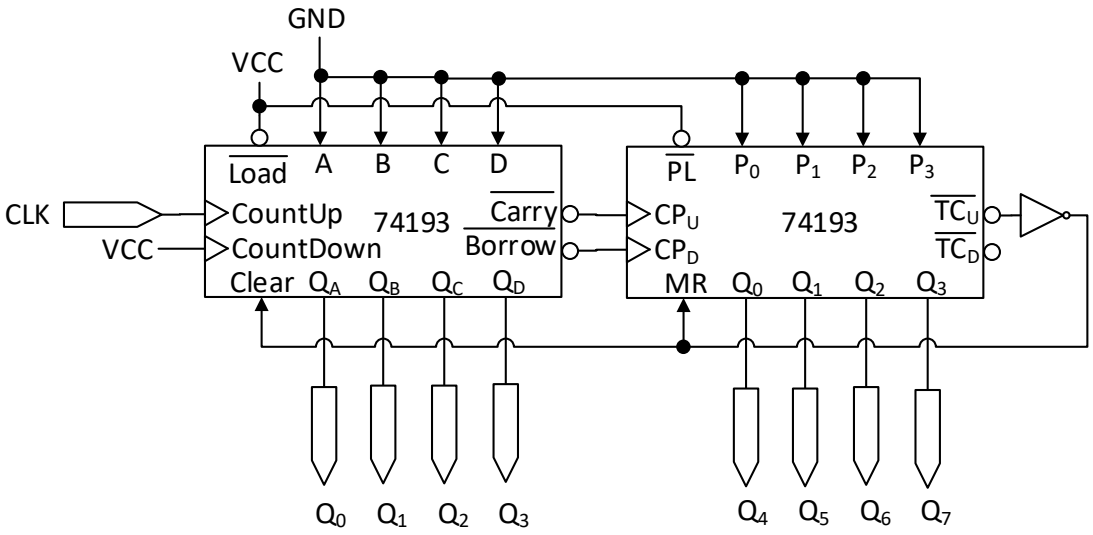
The register can be implemented using circuit 7495 having the clock inputs connected to a common clock signal. The two inputs of the 74283 adder will be connected to register outputs and inputs $X_{3:0}$, respectively. C_0 will be connected to 0 (GND). The result of the summation unit will be forwarded to the data inputs of the register for parallel load.



21. Implement a synchronous binary counter, with ascending counting loop from 0 to 158, using counters 74193 and 74192.

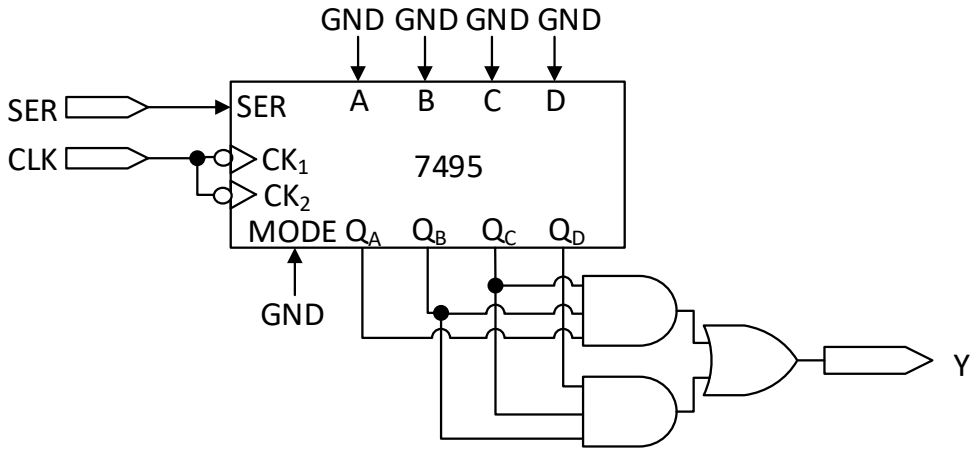
Solution:

Counters 74193 and 74192 are synchronous 4-bit bidirectional counters. For an 8-bit binary counter, the 74193 binary counter will count on the least significant bits $Q_{3:0}$ and the 74192 decimal counter will count on the most significant bits $Q_{7:4}$. Because the first value of the loop is 0, it can be loaded by resetting the counters. The reset command must be delayed for 1 clock impulse after value 158 is reached, because the reset command is asynchronous and overwrites the current state immediately. Therefore, the reset is initiated when the counter value reaches 159. As 159 (in binary = 1001 1111) is the maximum possible value, it can be detected using the TC_U flag (Terminal Count Up) of the most significant counter, the 74192 circuit. The TC_U must be inverted because it is active-low and the reset command is active-high. The parallel load of the counters is deactivated by connecting to 1 (VCC).



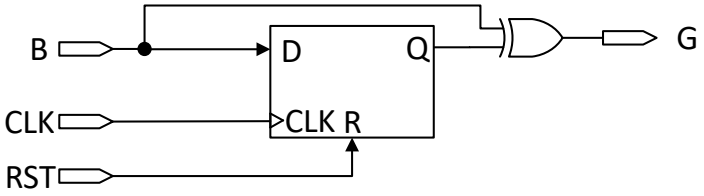
22. Using the 7495 shift-register and basic logic gates, implement a circuit which serially loads a binary sequence (1 bit per clock impulse) from the input SER. The circuit must activate the output Y=1 when there are three consecutive 1s between the most recently loaded 4 bits, otherwise Y=0.

Solution: The register can be implemented using circuit 7495 having the clock inputs connected to a common clock signal. The 3 consecutive 1s can be found on Q_A, Q_B, Q_C or on Q_B, Q_C, Q_D. Consequently, the detection can be implemented with the boolean expression $Y = Q_A \cdot Q_B \cdot Q_C + Q_B \cdot Q_C \cdot Q_D$. For serial load activation, the input MODE will be connected to 0 (GND).



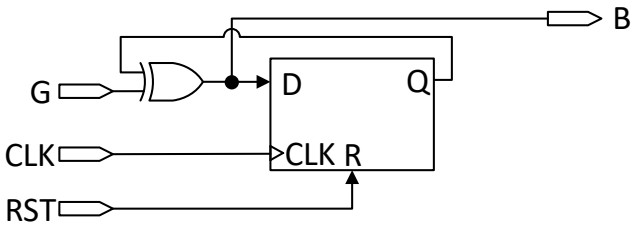
23. Implement a serial BCD->Gray converter with a B input, a reset, a clock signal and a serial output G. The binary number is serially received on B and the correponding Gray code is generated on G, one bit per clock cycle. The input sequence starts with the most significant bit, and ends with the least significant bit when reset is enabled asynchronously. After disabling the reset marks the beginning of a new number.

Solution: The expression for BCD->Gray conversion at rank *i* is: $G_i = B_i \oplus B_{i+1}$. The current bit B_i will be saved in a D flip-flop and will be available on the output Q, in the next clock cycle, when B_{i-1} is available on the input. Therefore, $B_{i-1} \oplus Q$ will generate the next Gray code bit G_{i-1}, as follows: $G_{i-1} = B_{i-1} \oplus Q = B_{i-1} \oplus B_i$. At startup, the flip-flop is initialized to 0, meaning the highest rank bit G_n will be $G_n = B_n \oplus 0 = B_n$, and B_n is saved. At next cycle, $G_{n-1} = B_{n-1} \oplus Q = B_{n-1} \oplus B_n$, and B_{n-1} is saved. Then follows $G_{n-2} = B_{n-2} \oplus Q = B_{n-2} \oplus B_{n-1}$, and B_{n-2} is saved, etc.



24. Implement a serial Gray->BCD converter with a G input, a reset, a clock signal and a serial output B. The Gray code is serially received on G and the correponding binary code is generated on B, one bit per clock cycle. The input sequence starts with the most significant bit, and ends with the least significant bit when reset is enabled asynchronously. After disabling the reset marks the beginning of a new number.

Solution: The expression for Gray->BCD conversion at rank *i* is: $B_i = G_n \oplus G_{n-1} \oplus \ldots \oplus G_i$. The result B_i will be saved in a D flip-flop and will be available on output Q, in the next clock cycle, when G_{i-1} is available on input. Therefore, $Q \oplus G_{i-1}$ will generate the next binary bit B_{i-1}, as follows: $B_{i-1} = Q \oplus G_{i-1} = G_n \oplus G_{n-1} \oplus \ldots \oplus G_i \oplus G_{i-1}$. At startup, the flip-flop is initialized to 0, meaning the highest rank bit B_n will be $B_n = 0 \oplus G_n = G_n$, and G_n is saved. At next cycle, $B_{n-1} = Q \oplus G_{n-1} = G_n \oplus G_{n-1}$, and $G_n \oplus G_{n-1}$ is saved. Then follows $B_{n-2} = Q \oplus G_{n-2} = G_n \oplus G_{n-1} \oplus G_{n-2}$, and $G_n \oplus G_{n-1} \oplus G_{n-2}$ is saved, etc.



Bibliography

1. Lucia Văcariu, Octavian Creț, “Analiza și Sinteza Dispozitivelor Numerice – Îndrumător de laborator”, Ediția a 3-a, U.T. Press, 2009.

2. Lucia Văcariu, Octavian Creț, “Probleme de Proiectare Logică a Sistemelor Numerice / Logic Design Problems for Digital Systems”, Ediția a 2-a, U.T. Press, 2013.

3. Randy H. Katz, Gaetano Borriello, “Contemporary Logic Design”, 2nd Edition, Pearson, 2004.

4. John Francis Wakerly, “Digital Design Principles and Practices”, 5th Edition, Pearson, 2018.

5. Charles H. Roth, Larry L. Kinney, “Fundamentals of Logic Design”, Enhanced 7th Edition, Cengage Learning, 2020.

6. Wayne Wolf, “FPGA-based System Design”, Prentice Hall, 2004.

7. Sarah L. Harris, David Harris, “Digital Design and Computer Architecture”, RISC-V Edition, Morgan–Kaufmann, 2021.

8. M. Morris Mano, Michael D. Ciletti, “Digital Design with an introduction to the Verilog, VHDL, HDL, and SystemVerilog”, 6th Edition, global edition, Pearson, 2018.

9. R. P. Jain, Kishor Sarawadekar, “Modern Digital Electronics”, 5th Edition, Mc Graw Hill India, 2022.

10. Thomas Lawrence Floyd, “Digital Fundamentals”, 11th Edition, Global Edition, Pearson, 2015.

11. Cristian-Cosmin Vancea, “TTL_Env Project for Project Navigator”, Available online: https://drive.google.com/uc?export=download&id=1800AdR6Vdo8PDd1tB9n5LCQZNHAcgh9K

12. Cristian-Cosmin Vancea, “TTL library for Logisim”, Available online: https://drive.google.com/uc?export=download&id=1j4kRe9JXdQi6MqnqsB5nrXfX1uwoVc43