

Digital Logic

Aims	2
Equipment	2
Environment.....	2
Transferring files from one lab partner to another	2
1.0 Boolean algebra.....	2
1.1 Boolean variables and expressions	2
1.2 Boolean operators.....	2
1.3 Truth tables.....	3
1.4 Associative and distributive laws	3
1.5 Basic logic identities	3
2.0 Electronic gates	4
2.1 CMOS logic.....	4
2.2 Redundancy of logical operators	5
2.3 The NAND gate	5
3.0 Basic Windows.....	5
4.0 Learning to use Quartus II	6
5.0 Gate Notation	9
5.1 Other Types of Gates.....	10
6.0 Combinatorial logic.....	10
6.1 Logic and arithmetic	10
6.2 Expression abstraction	11
6.3 Implementation of circuits in sum-of-product form.....	11
6.4 A full adder.....	12
6.5 A 4-bit ripple-through adder	13
6.6 Using Bus lines in Quartus II.....	13
6.7 The Hierarchy Display	14
7.0 The two's complement representation	15
8.0 A simple arithmetic logic unit.....	16
8.1 An improved ALU.....	16
9.0 Gating and multiplexing.....	16
9.1 Extending the range of a multiplexer.....	17
10.0 Extending the ALU.....	17
11.0 Sequential Logic.....	18
11.1 A testing system for the ALU	18
11.2 Adding feedback and memory	19
11.3 The D flip-flop	19
Suggestions for further reading	20

Aims

In this experiment, you will learn the basis of **Boolean algebra** and the rules for the manipulation of logical expressions. You will use this algebra to solve problems of combinatorial and sequential logic design. Your solutions will be implemented electronically as arrangements of gates based on CMOS components. However, you will test all your solutions *'virtually'*, using a software simulation package called **Quartus II**¹.

Equipment

This is a software – only experiment: use a networked lab XP PC

Environment

You will be using a personal computer (PC) fitted with a pentium processor chip, internal hard disc, keyboard and mouse. All the application software you need are stored locally on drive c: which is read-only. All your personal files are stored on a network disk on level 10. You will be working entirely in the Microsoft Windows XP environment. This will allow you to select and manipulate applications and tools using graphical symbols (**icons**) instead of typing commands through the keyboard.

Transferring files from one lab partner to another

You will work in pairs on this experiment, and so at the end of each session, you should exchange any files that you have made, so that both partners have a complete set of data. So long as you have genuinely worked together, this does not constitute plagiarism. Because of the way that Windows XP is set up, it is difficult to map drives between students, so the easiest way to transfer data is by floppy disc or memory stick. Remember to retrieve any such device once you have used it, and not leave it in the PC!

1.0 Boolean algebra

A **symbolic algebra** consists of a set of operators that act on a corresponding set of variables according to predefined rules. The resulting expressions can then describe relationships between the variables. While such algebras can take many forms, the version used to describe logical relations (known as **Boolean algebra**, after its originator, George Boole, 1815-1864) has been overwhelmingly successful. As originally conceived, Boolean algebra was intended simply to decide the truth or falsehood of logical propositions (it is therefore sometimes known as **propositional calculus**). However, since it is essentially a binary system, it can also be used as a basis for binary arithmetic. Furthermore, since binary states may be conveniently described by the condition of two-way switches, it is a natural candidate for electronic implementation. All digital computers (i.e. all practical electronic machines that carry out both logical and arithmetic operations) are therefore based on Boolean algebra at their most fundamental level of design.

1.1 Boolean variables and expressions

Boolean expressions are normally two-valued, being either **true** or **false**. For example, the proposition 'has wings' would be *true* for a glider; however, the proposition 'has engine' would be *false*. To simplify the notation, propositions are described symbolically. The proposition 'has wings' might therefore be described as 'proposition A', or more simply as the **Boolean variable** 'A', while 'has engine' might be defined as the variable 'B'. The two possible states of each variable may then be written more simply as '1' (standing for 'true') and '0' (for 'false').

Boolean expressions are constructed by combining variables with operators. In the above example, the proposition 'has wings and an engine' is an expression, since it establishes a particular relationship between the variables 'has wings' and 'has engine'. In this case, the link is provided by the word 'and', which is itself a **Boolean operator**. Boolean expressions are also two-valued - 'has wings and an engine' would be *true* for an airliner, but *false* for a glider.

1.2 Boolean operators

Conventionally, Boolean algebra is based on the use of three operators: **AND**, **OR** and **NOT**. The first is used to establish a relationship of the form 'A and B', while the second is used to describe 'either A or

¹ <http://www.altera.com/products/software/products/quartus2/qts-index.html>. Quartus II Web edition can be downloaded free from this site.

B'. The last acts on a single variable to produce a complement or inverse. For example, if the variable A describes the proposition 'has wings', NOT A represents 'does not have wings'. Furthermore, if A is true, NOT A must be false. The notation used in this experiment to describe the use of these operators is shown below:

Operator	Symbol	Example
AND	•	$A \bullet B$
OR	+	$A + B$
NOT	-	\overline{A}

1.3 Truth tables

Simplification of a logical expression can be assisted by considering its **truth table**. This details the value of the expression (the **output**) obtained for every combination of the variables in the expression (the **inputs**). For example, Figure 3 shows the truth tables for AND, OR and NOT. Notice that since there are two inputs in the left-hand table, each row of inputs can be considered as two-bit **binary number**, ascending in order from 00 (decimal zero) at the top to 11 (three) at the bottom. This arrangement ensures the inclusion of all possible combinations. Here, there are 2^2 rows; for N variables, there will be 2^N rows.

Inputs		Outputs	
A	B	$A \bullet B$	$A + B$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Input	Output
A	\overline{A}
0	1
1	0

Exercise 1. ✎ Gliders (X) and airliners (Y) may have as component parts wings (A) and engines (B). Construct logical expressions for X and Y in terms of A and B, and draw up appropriate truth tables for each.

1.4 Associative and distributive laws

Boolean expressions obey some standard laws of linear algebra. In particular, the **associative** and **distributive laws** can be used to group and ungroup expressions during simplification. Examples of their use are given below.

Associative law: $A \bullet B \bullet C = (A \bullet B) \bullet C = A \bullet (B \bullet C)$
Distributive law: $A \bullet (B + C) = (A \bullet B) + (A \bullet C)$

1.5 Basic logic identities

A number of other rules exist which are useful in simplifying expressions. These are summarised below.

- i) $A \bullet \overline{A} = 0$ $A + \overline{A} = 1$
- ii) $A \bullet 1 = A$ $A \bullet 0 = 0$ $A \bullet A = A$
- iii) $A + 1 = 1$ $A + 0 = A$ $A + A = A$
- iv) $\overline{(A + B)} = \overline{A} \bullet \overline{B}$ $\overline{(A \bullet B)} = \overline{A} + \overline{B}$

Equations iv) (known as **de Morgan's theorem**) suggest that a logical function may be replaced by an alternative (the **inverse function**) by a simple procedure: a) complementing all the variables in the function, b) replacing all ANDs by ORs, and vice versa, and c) complementing the result. This procedure can be extended to functions of any number of variables.

Exercise 2. ✎ Verify Equations iv) above using truth tables. Derive one from the other.

Exercise 3. ✎ Simplify the following boolean expression as much as possible:

$$\overline{(\overline{V} + X)} \bullet (W \bullet (\overline{Y} + Z)) + \overline{(\overline{V} + X)} \bullet (W \bullet (\overline{Y} + Z))$$

2.0 Electronic gates

Logical expressions may be evaluated electronically using circuits called **gates**. Figure 1 shows an example. The input is a voltage, which can take two possible states, “High” ($= V_{CC}$), or “Low” ($= 0V$). We are interested in the output voltage. In Figure 4a & 4b, two switches link a voltage supply rail V_{CC} with a zero volt rail. Assume that the switches are electrically – activated (like relays, or transistors). Each switch is controlled by the same input; however, they are of different types. The *upper* one is arranged to be open when the input voltage is high, and closed when it is low. The *lower* one has the opposite property; it is closed when the input is high, and open when it is low. As a result, the switches can never be open together. When the input is high, the lower switch ties the output to zero, forcing it low. On the other hand, when the input is low, the output is tied to V_{CC} , forcing it high. Assuming that low and high voltages correspond to logic states 0 and 1, the output is the complement of the input and the circuit is an **inverter**. Figure 1c shows how this could be implemented using CMOS transistors.

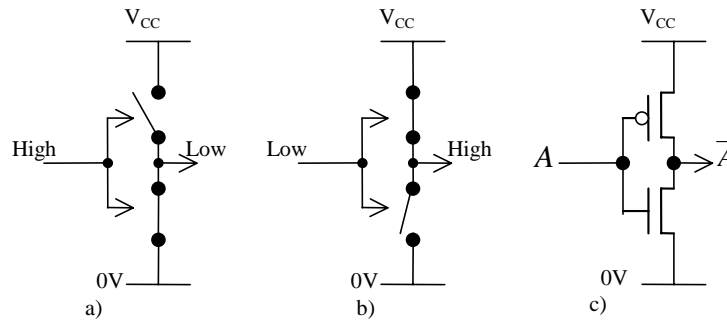


Figure 1. a) and b) switch connections implementing an inversion; c) CMOS inverter

2.1 CMOS logic

The operation of a **CMOS (complementary metal oxide semiconductor)** FET is shown in Figure 2a. Current flows between two contact electrodes, the **source** and the **drain**, via a pathway known as a **channel**. Above the channel is a further electrode, the **gate**. Using some clever semiconductor engineering, a field applied across an insulating layer between the gate and the substrate can be made to control the **resistivity** of the channel. As a result, the source-drain current may be passed or blocked at will, so the device acts as a switch. Figure 2b shows a simple CMOS process. Here (by a further clever trick, this time one of fabrication) the use of polysilicon allows the gate to be exactly aligned over the channel, without the need for alignment tolerances. The channel length can therefore be minimised, thus maximising the device speed.

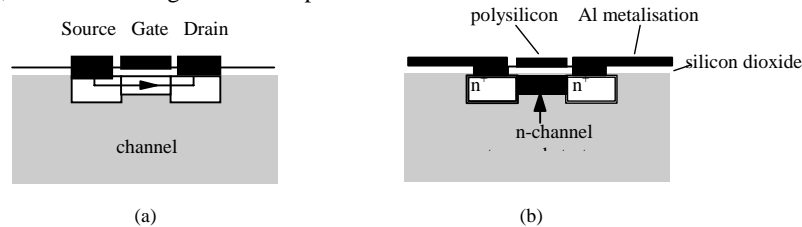


Figure 2. a) Schematic of a field effect transistor, and b) realisation of n-channel FET in CMOS technology

Different CMOS transistor variants exist. **N-channel** FETs operate via the flow of electrons (n-type carriers), while **p-channel** FETs use holes (p-type carriers). Conveniently, unblocking the channel requires a different control voltage in each case. With further adjustment, the n-channel transistor can be made to turn on if a high voltage (logic 1) is applied to its gate, and off if a low voltage (logic 0) is applied. The p-channel transistor is exactly the reverse; it is off if a logic 1 is applied, and on if a logic zero is applied. Figure 3 shows the symbols used for the different types. Figure 1c shows how the inverter circuit previously described can be built up using two CMOS transistors. Make sure that you understand the way it operates. Would it still work if the transistors were exchanged?

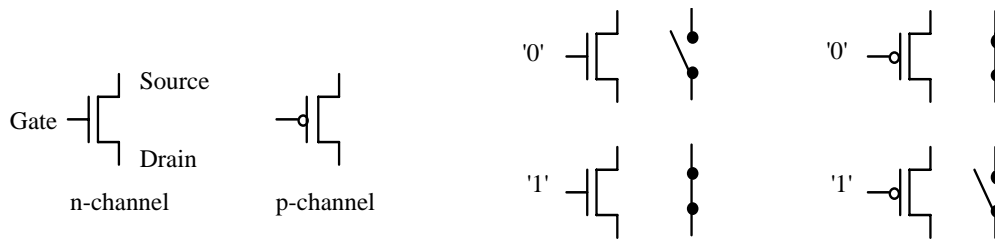


Figure 3. Symbols for n- and p-channel CMOS transistors

2.2 Redundancy of logical operators

From de Morgan's theorem, it can be seen that of the three basic logical operators, either the OR or the AND operator is unnecessary. For example, since $A + B = \overline{(\overline{A} \bullet \overline{B})}$, all occurrences of OR can be replaced by combinations of AND and NOT. Similarly, since $A \bullet B = \overline{(\overline{A + B})}$, AND can be replaced by combinations of OR and NOT. The first such combination - AND and NOT - is used so often that it is given a special name, **NAND**. Similarly, the second combination - OR and NOT - is referred to as **NOR**. The truth tables for these new operators are

A	B	$\overline{(A \bullet B)}$	$\overline{(A + B)}$
0	0	1	1
0	1	1	0
1	0	1	0
1	1	0	0

This process of eliminating redundant operators may be continued. For example, since $\overline{(\overline{A} \bullet A)} = \overline{A}$, a NOT operation may be performed using a NAND. Finally, since $\overline{\overline{(A \bullet B)}} = A \bullet B$, an AND operation can be realised using NAND and NOT operators, and hence by a combination of NANDs.

2.3 The NAND gate

While the procedures above appear tortuous, they allow a very powerful conclusion to be reached: any logical expression may be realised using NANDs alone, so the NAND operator is **functionally complete**. The importance of this result becomes clear when we consider the electronic realisation of logical operators: it reduces the number of circuit types required to just one, the NAND gate! This is shown in Figure 4; check that you understand its operation.

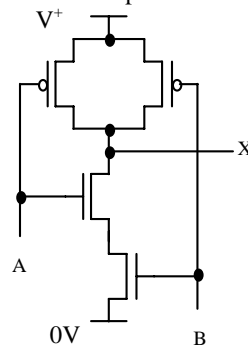





Figure 4. CMOS NAND gate

Exercise 4. Figure 4 shows a CMOS NAND gate. Design a similar NOR gate using CMOS transistors as switches.



3.0 Basic Windows

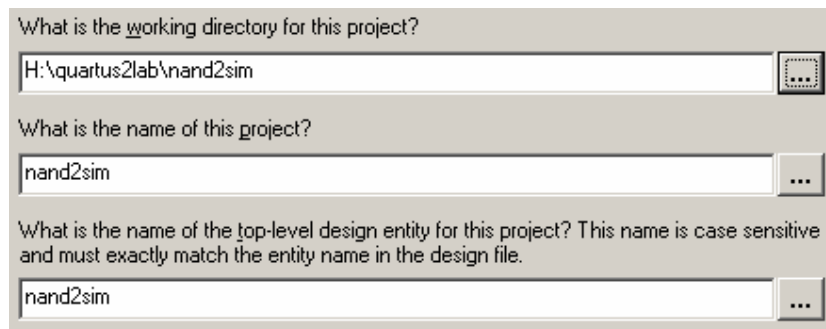
- Log onto the XP system
You need to make a folder on your **h:** drive, to store your work.

- LH mouse click on  My Computer
- Select drive **h:** by clicking on the drive ICON 
- Use the RH mouse button to access the pull-down menu. Move down to *NEW*, then across to *FOLDER*. Name your new folder *quartus2lab*.
- Close the windows by LH mouse clicking on the  at the top of each window.

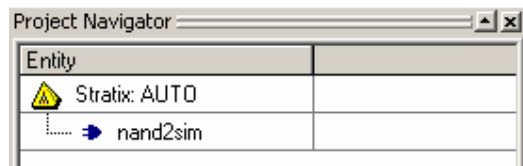
4.0 Learning to use Quartus II


In this part of the experiment, you will create a simple NAND gate schematic and simulate it under the **Quartus II** environment. To launch the program:

1. Start **Quartus II** (Select **Start>Programs>Altera>Quartus II**). LH mouse click on *Start* , and slide up to *Programs*, across and up to *Altera*, and across to *Quartus II*. When you release the mouse button, the program will start.²
2. Start a new project  (Select **File>New Project Wizard**)
3. Select your **working directory**, choose the folder *quartus2lab* which you created before and create a folder called *nand2sim*. Note that you may create all projects under *quartus2lab* however it is recommended that you create a new folder for each exercise. You should now see the following dialogue box:



4. Click **Finish**³. If you have not create the directory *nand2sim*, click yes when the program ask if you want to create the directory. You should now see the following in your Project Navigator window:



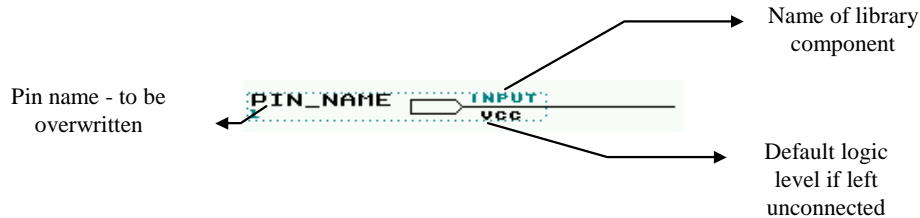
5. Next, open a new schematic sheet with the menu command: **File>New** or click on  and select **Block Diagram/Schematic File**.
6. **Saving your Schematic** - Save this Schematic File with the menu command: **File>Save As**, the file name should be the same as your project name (*nand2sim*) and click save.

² It will save time if you make a new shortcut on your desktop. Use the RH (right-hand) mouse button.

³ The next few steps specify other details of the project including the target device, as you are only simulating your design these steps are not necessary.

Library Components

- Obtain a NAND gate from the component library by double-clicking the Left Mouse Button (LMB) on the blank part of the sheet. For this experiment you will be using components from the **primitive library** (*prim*), the **macrofunction library** (*mf*), and components from your design's home directory.
- You will find the primary library in the path c:\altera\quartus50\libraries\primitives. Expand the list and locate the folder logic.
- Pick up a 2-input NAND gate from the logic folder (*nand2*). Place it on the schematic.
- Pick up and place both an input port symbol (*input*) and an output port symbol (*output*) from the primitive library under **pin**. In order to simulate any design, all input and output pins must be connected to the port symbols.
- Note that a input port symbol consists of a number of elements as shown below:



- Make a duplicate copy of the input port symbol using menu command: **Edit > Copy**, and then **Edit > Paste**. (A shortcut is to hold the *Ctrl*-key down and click the LMB on the symbol and drag a copy elsewhere on the sheet.)

Wiring-up a Circuit

- Next we must wire up the various symbols. Move the cursor to a terminal of the NAND gate, press, and keep pressing, the LMB and drag the mouse to the destination terminal. You should see a right-angle connection being drawn on the schematic. You may also label any wire by selecting the connection with LMB (it turns BLUE), and simply type the name of the wire.
- Wire-up your circuit according to figure 6.

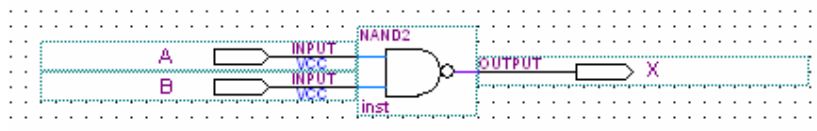


Figure 6 A Simple NAND gate circuit

Changing pin names of input/output port symbols

- Next label all the input and output ports correctly. This can be done by a RMB click on the pin and then select **Properties**. Replace the original pin_name with the new name (such as A) on the keyboard and click OK.

Compiling your Design (button:)

- Either use the pull-down menu: **Processing>Start Compilation...**, or click on the above button, make sure that the schematic diagram does not contain obvious errors. Compilation will synthesize your design and creates all the necessary files for simulation.
- Note that during compilation, two files (**.sof** and **.pof**) will be created for loading the design to the targeted hardware. However these files are not needed for simulation and can be deleted to save space on your hard disk.

Creating Signals to Test the Design

- Use the menu command: **File>New**, and select **Vector Waveform File** under **Other Files** tab.
- Specify the input and output nodes from your design using the menu command: **Edit>Insert Node or Bus**.
- Click the **Node Finder** button on the dialogue box. Choose **Pins: all** under **Filter** and click the **list** button. You should see the following dialogue box:

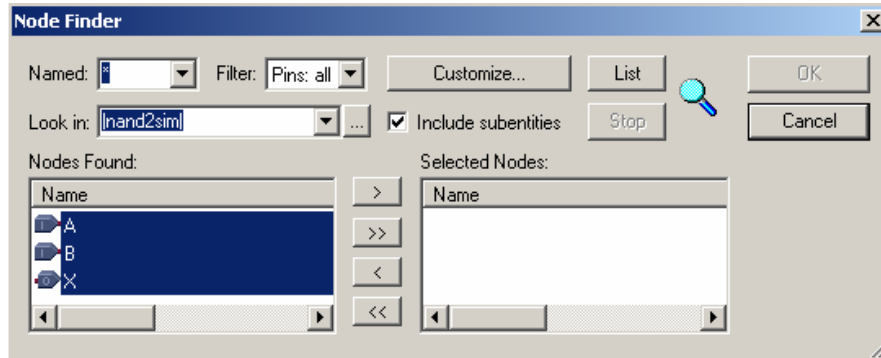


Fig 7 Node Finder dialogue box

- Select the nodes A, B and X and click the **>>** button, followed by the **OK** button.
- You should see these three signals are now included in the Waveform display. Next, we must define the timing resolution with the menu command: **Edit>Grid Size...** Enter a grid size of 100ns.
- Next, specify how long you want to simulate for. Use the menu command: **Edit>End Time**, and enter 1us. Hold **Ctrl** and click **W** to zoom so that it show the full 1us long timeline in 100ns step.
- You are now ready to create waveforms for inputs A and B. To do this, first select the signal A by clicking on the pin symbol **A**. Then use the menu command: **Edit>Value>Count Value...** You should see a dialogue box as shown here:

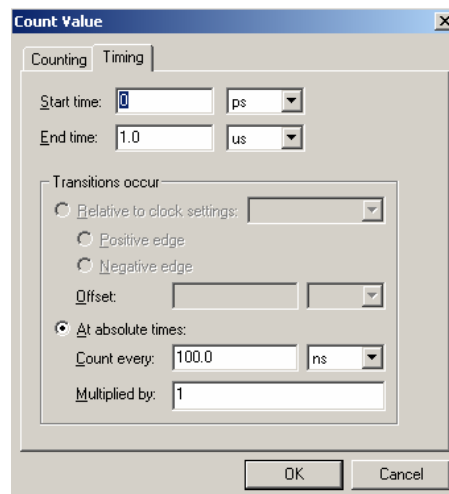


Fig 8 Count value dialogue box

- With the value entered as shown, the signal A will toggle between 0 and 1 at a duration of 100ns.
- Do the same for signal B, but enter a value of 2 in the **Multiplied By** box.
- Save this waveform signal file (**nand2sim.vwf**). You should see a waveform file similar to the following:

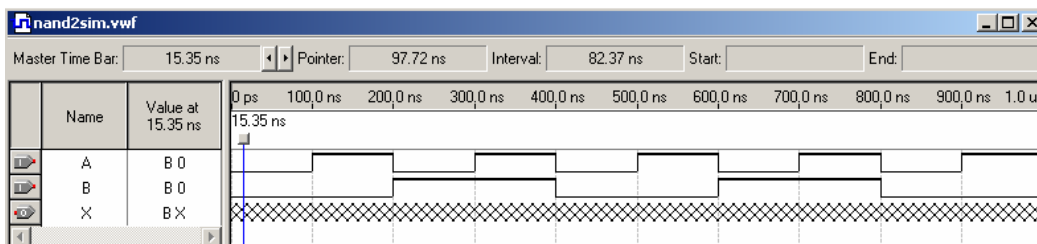


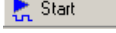
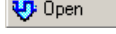


Fig 9 the waveform editor

Remember to save your work, and simulate the circuit, *Using the Simulator* (button: )

- Invoke the menu command: **Tools>Simulator Tool**. A dialogue box will appear. Click on the  button and select nand2sim.vwf as the input file. Followed by clicking the **Overwrite simulation input file with simulation results check box** and click the  Start button. Open the file by clicking the  Open button and verify that the output signal X is as expected.

5.0 Gate Notation

The most common gates are NOT, AND, OR, NAND and NOR; however others (e.g. **XOR**, standing for **exclusive OR**) also exist and may occasionally be encountered. There are two standard symbol sets for representing gates. Figure 10 shows the old **US Military Standard**. This will be used here for compatibility with **Quartus II**; however, you should be aware that it has been superseded by the more modern ANSI/IEEE Standard.

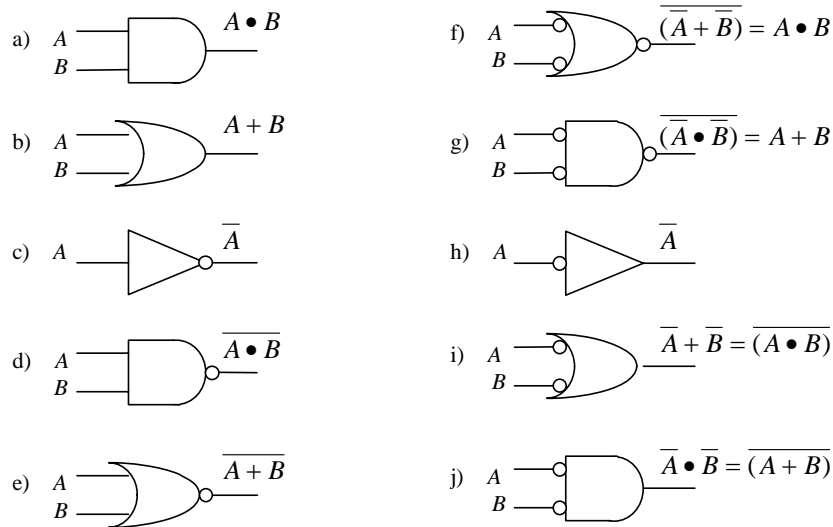


Figure 10 a) - e) Logical symbols for AND, OR, NOT, NAND and NOR gates; f) - j) mixed logic representations

In the Military standard, functions are represented using symbols of different shapes. For example, Figures 10a) and 10b) show AND and OR gates. Inversion of the output is indicated by a circle, so that NOT, NAND and NOR gates are drawn as shown in Figures 10c) - 10e). It is also possible to use an alternative scheme known as **mixed logic representation**, in which the inputs are inverted in addition to the outputs. Figures 10f) - 10j) show mixed logic equivalents for the gates in Figs 10a) - 10e).


Exercise 5.  Figures 11a) and b) use NAND gates to implement logic functions. Determine the function in each case. Using similar principles (i.e., using only NAND gates), design an OR gate.



Figure 11 Different logic functions based on NAND gates

5.1 Other Types of Gates

Most logic families contain a large variety of gates. Quartus II system offers a full and comprehensive library of such gating functions either in the primitive function library or the macrofunction library. You can explore this using the help menu: **HELP>Macrofunctions/LPM**. You should see a help screen similar to the ones shown in figure 12. Click on any of these to get a description of what it does.

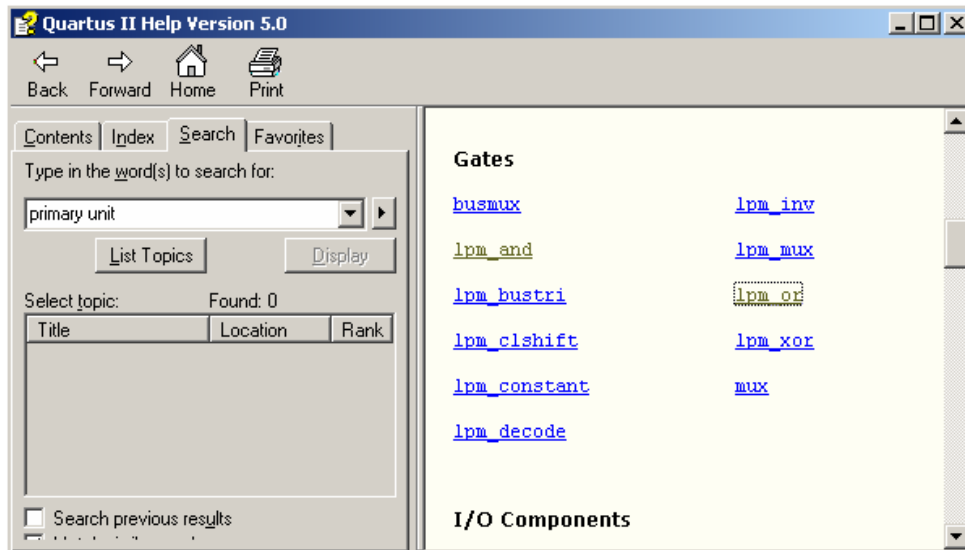


Figure 12 Help Pages for Macrofunctions/LPM

6.0 Combinatorial logic

In many logic problems, the outputs are simple functions of the inputs. Thus, a pair of outputs X and Y may be related to inputs A , and B by functions f_1 and f_2 , such that $X = f_1(A, B)$ and $Y = f_2(A, B)$. Problems of this type are referred to as **combinatorial problems**, and part of this experiment is concerned with their solution and implementation using gates. Later on, you will encounter problems in which the outputs are also functions of their own previous values, through a form of feedback; such problems are referred to as **sequential**.

Combinatorial problems are generally tackled in a systematic way. Firstly, a truth table is drawn up for the problem. Secondly, logical expressions for the outputs are extracted from the truth table as a sum of canonical products. Various procedures are then used to simplify the sum-of-product expression (you will encounter some of these in the 1st-year digital electronics course). Finally, the circuit is constructed according using a standard technique. We shall consider each of these steps in turn, using the example of a circuit for **binary addition**.

6.1 Logic and arithmetic

Consider the addition of two 1-bit binary variables A and B . The result of the addition is clearly decimal zero (binary 00 in 2-bit notation) when A and B are both zero, decimal 1 (binary 01) when A or B (but not both) are 1, and decimal 2 (binary 10) when A and B are both 1. These results are shown as a lookup table in Figure 13.

A	B	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Figure 13 Lookup table for 1-bit arithmetic

The close resemblance of Figure 13 to a truth table suggests that arithmetic may be performed electronically by using gates to generate solutions from a binary lookup table, instead of executing a more conventional decimal algorithm.

6.2 Expression abstraction

Given a truth table, expressions describing relationships between inputs and outputs may be abstracted very simply.

For example, the fourth line in Figure 13 suggests that $\text{Carry} = A \bullet B$, since Carry is only 1 when A and B are both 1. Expressions of this type are known as products, since they have the appearance of a simple arithmetic product. Two combinations can be found that give 1 for Sum, either $\overline{A} \bullet B$ or $A \bullet \overline{B}$. Thus, the expression for Sum is a sum of products, written as $\text{Sum} = (\overline{A} \bullet B) + (A \bullet \overline{B})$.

Sum-of-product expressions are also known as canonical expressions when every variable appears in every term.

Each product in a canonical expression is known as a minterm. Each minterm can be considered as a number for the purposes of identification. The numbers are obtained by considering the variables as binary 1's and their complements as 0's. Thus, the first minterm in Sum would be identified by the decimal number 1 (binary 01) and the second by decimal 2 (binary 10).

6.3 Implementation of circuits in sum-of-product form

Three stages are required in the construction of a sum-of-product expression. In the first, some or all of the inputs are inverted. In the second, different combinations of the inputs (and/or their complements) are ANDed together to give the individual product terms. Finally, the terms are ORed together to give the sum.

These three stages can still be implemented in a circuit constructed from NAND gates alone. Single NAND gates can obviously replace the inverters required in the first stage. NANDs can also replace the ANDs in the second stage, provided we take note of the additional inversion involved. As it turns out, De Morgan's theorem suggests that a single NAND can then implement the final OR operation in a particularly elegant way.

De Morgan's theorem was previously given as $\overline{(A \bullet B)} = \overline{A} + \overline{B}$. However, the alternative

$\overline{(\overline{A} \bullet \overline{B})} = A + B$ is equally valid. This implies that the final OR can be performed as a NAND, if the terms involved have previously been inverted. As just described, this inversion occurs automatically when NANDs are used in the second stage. Therefore, in the all-NAND approach, all that is required is to replace all the second- and third-stage AND and OR gates by NANDs.

To illustrate this principle, consider the expression for Sum found above. Using De Morgan's theorem,

this can be written as $\text{Sum} = \overline{(\overline{A \bullet B}) \bullet (\overline{A \bullet B})}$. To generate this expression, five NANDs are required: two first-stage NANDs to provide the complements \overline{A} and \overline{B} , two second-stage NANDs to generate the products $\overline{(\overline{A} \bullet \overline{B})}$ and $\overline{(A \bullet B)}$, and a third-stage NAND to provide the sum $\overline{(\overline{(\overline{A} \bullet \overline{B})}) \bullet (\overline{(A \bullet B)})}$. Figure 14 shows the circuit.

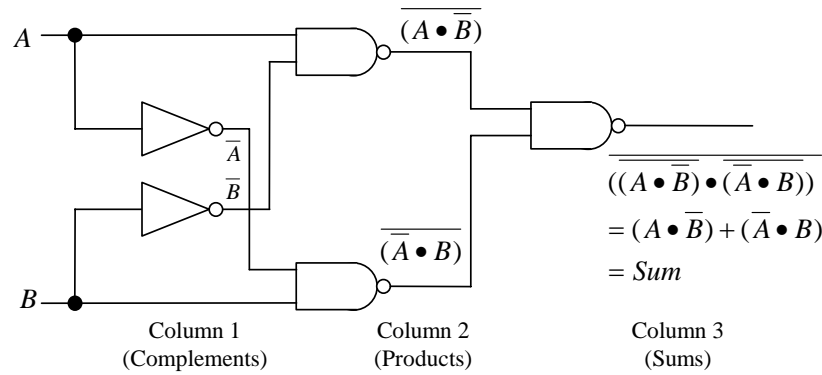


Figure 14 Sum-of-product circuit for the Sum output

Exercise 6. Design a circuit to generate the Carry output for 1-bit arithmetic. Using **Quartus II** construct a combined Sum and Carry circuit from **nand** gates only, and save it under a **new project name halfadd** (H:\quartus2lab\halfadd). Verify its operation using the simulator. Make sure that you create a default symbol for this using the menu command: **File>Create/Update>Create Symbol files for Current File**. This will make a symbol similar to that shown in figure 15 a) automatically.

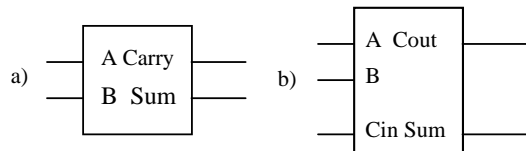


Figure 15 a) half_adder and b) full_adder components

NOTE: You should create a new folder/directory for each project (design) as this will greatly help the structuring of your work especially when your design becomes large.

6.4 A full adder

One-bit addition circuits can be modified to add together two binary digits A and B, together with the carry from a previous addition. The resulting **full adder** can then be used as the building block of a more complicated circuit that can add together two N-bit numbers (and hence perform useful arithmetic). Figure 16 shows the truth table for a full adder. Check that you understand the way that it is constructed.

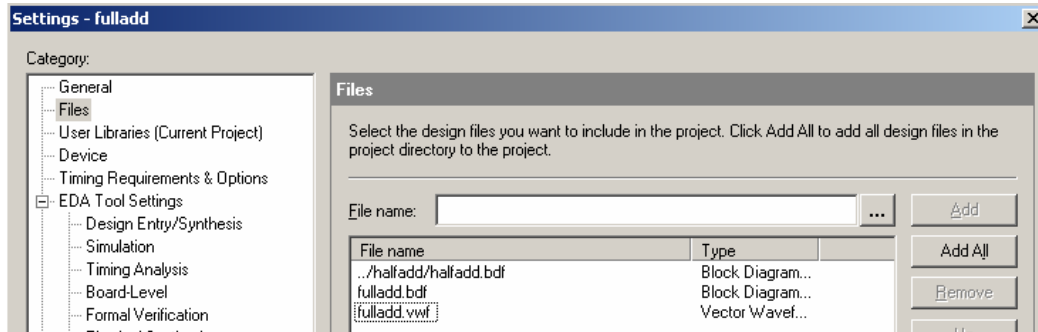
A	B	Carry_in	Carry_out	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1


Figure 16 Full adder truth table

Exercise 7. Using Figure 16, construct sum-of-product expressions for Carry_out and Sum. Design a circuit to generate Carry_out and Sum using two half adders and an OR gate. Using your component **halfadd** and an **OR gate**, construct the circuit and save it under the project name **fulladd**. Verify that it generates the truth table of Figure 16 using the simulator. Create a default symbol for this and you should have a default symbol similar to that shown in Figure 15(b).

Adding symbol to your design

- Assuming you have created a project in different folder for each exercise, you will need to add the **.bdf Block Diagram** file to your design. Use the menu command: **Assignments>Settings** and choose **Files** under **Category**, locate and add the .bdf file to your project. You should see a dialogue box similar to the one shown below.



- To add this symbol to the schematic file, double click on the blank part of the sheet and click on the  button. Locate your symbol (e.g. H:\quartus2lab\halfadd\halfadd.bsf) and click OK.

6.5 A 4-bit ripple-through adder

Two multi-bit binary numbers (or **words**) can be summed by using full adders to emulate the process of manual addition. Figure 17 shows a circuit that can add together two 4-bit numbers $A = A_3 \dots A_0$ and $B = B_3 \dots B_0$. Four full adders are used, in a chain. The N^{th} stage adds together the corresponding bits A_N and B_N from each word (together with any carry from the $N-1^{\text{st}}$ stage) and produces the N^{th} bit in the sum. There is no need for a carry into the 0^{th} stage, but there may be a carry out of the 3^{rd} stage if the sum exceeds 1111. Note that the N^{th} stage addition will only generate the correct sum when the carry from the $N-1^{\text{st}}$ stage is ready. This is a **serial adder**, also known as a **ripple-through adder**, as the final value of the sum will keep changing while the carries propagate through the chain from the right.

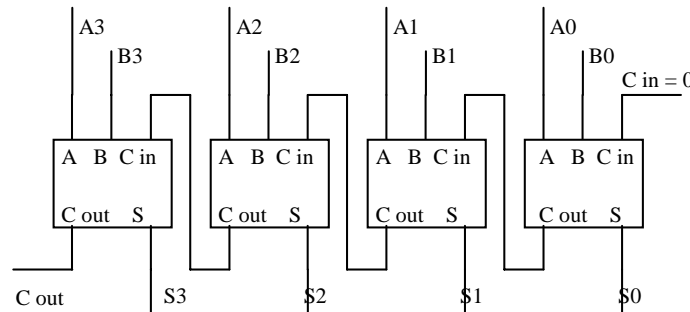


Figure 17 4-bit ripple-through adder circuit

6.6 Using Bus lines in Quartus II

As you can see from Figure 17, the inputs A & B and output S of the 4-bit adder may be simplified if, instead of dealing with sets of four independent bit lines, we deal with two 4-bit input **bus lines** and one 4-bit output **bus line** (we also have, of course, a bit input line for Cin and a bit output line for Cout). **Quartus II** lets you define such buses and this makes subsequent interfacing of components considerably easier as you only have to connect the bus lines and not each and every bit line that makes them up. Figure 18 shows detail from the inputs to a possible **4bitadd** design which makes use of bus lines.

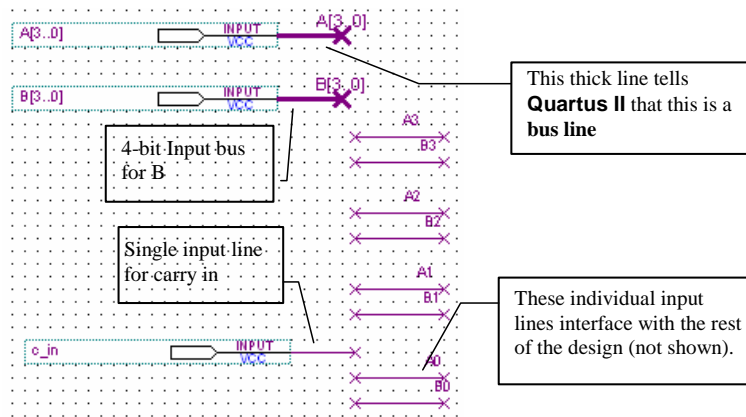


Figure 18 Input bus lines to 4bitadd

As you can see, there are two main stages to making a bus line in **Quartus II**. Firstly the inputs, for example for A, must be labelled in a sequential method **starting at 0**, i.e. A0,A1,A2,A3 if A is 4-bit. Secondly, input or output bus lines must be made and labelled. You will note that these inputs/outputs use the same input symbol as a single line input/output. The labelling, however, must indicate a bus. For example, if A is a 4-bit bus then the input line for A is labelled **A[3..0]** (i.e. A ranges from low bit 0 to high bit 3). A bus line must also be attached to the input/output. This is a *thick line* connection, and is obtained as shown in Figure 19 below.

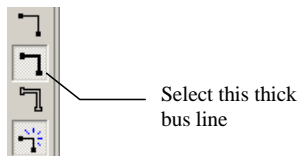


Figure 19 Creating a bus connection

Exercise 8. Using your **fulladd** component and bus lines, construct a 4-bit ripple-through adder circuit, and save it under the name **4bitadd**. Verify that it generates the correct answer to the additions 0111 + 0001, 1101 + 0010 and 1111 + 0001. Once again, remember to create a default symbol.

6.7 The Hierarchy Display

The last design you have constructed is complex enough to introduce the powerful *hierarchy* utility of **Quartus II**. The hierarchy display utility is located at the Project Navigator window similar to that shown in Figure 20.

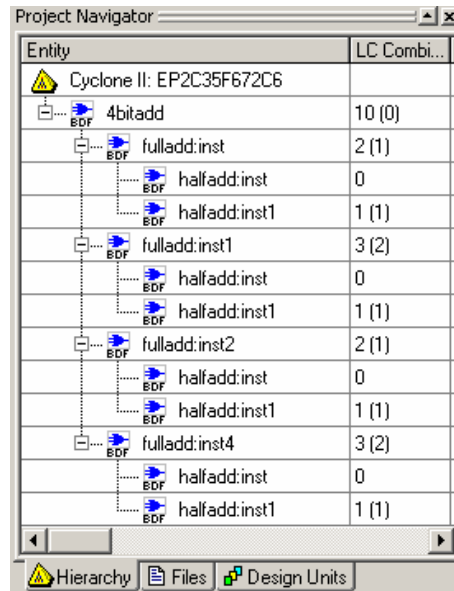


Figure 20 The Hierarchy display

You can clearly see the dependency of **4bitadd** on four **fulladd** circuits, each of which is dependent upon two half adders. With the LMB double click (select) one of the **fulladd** components. You will invoke another window which shows the **fulladd** circuit. You can also click on the Files tab to view all the files used in your design.

7.0 The two's complement representation

A four-bit number can be used to represent the positive integers from $0000 = 0$ to $1111 = 15$. This is known as **unsigned binary** notation. The weighting of each bit is as in table (a) below.

bit	weight
b0	2^0
b1	2^1
b2	2^2
b3	2^3
Unsigned number	

(a)

bit	weight
b0	2^0
b1	2^1
b2	2^2
b3	-2^3
2's complement number	

(b)

However, in most calculations, negative numbers will be required. A different convention known as the **two's complement** provides a way to represent negative numbers. Here the weighting of the most significant bit (MSB) is **negative** instead of positive, as shown in table (b) above. In this scheme, numbers in the range 0000 to 0111 (0 to 7) are chosen to be **positive**. These represent the numbers 0 to +7 (in that order). Numbers in the range 1000 to 1111 (8 to 15) are taken to be negative, and represent -8 to -1 (again, in that order). The name two's complement arises because the negative of a number can be found by changing all the 1's in its binary description to 0's (and vice versa) and adding 1 to the result.

Exercise 9. ✍ Write down the binary equivalent of -3 and -5. Perform the additions $1 + (-1)$ and $3 + (-7)$ in binary.

Exercise 10. ✍ Since $A - B = A + (-B)$, A minus B can be evaluated by adding A to minus B, where minus B is specified according to the two's complement scheme. Design a circuit to perform 4-bit subtraction, based on your **4bitadd** component. (N.B. There is no need to enter this design into **Quartus II**).

8.0 A simple arithmetic logic unit

The computational heart of a microprocessor is an **arithmetic logic unit** (or **ALU**). As its name suggests, this circuit can perform a variety of arithmetic and logical operations. Some operations (e.g. addition) act on two binary inputs A and B to generate a binary result; others (e.g. incrementing or decrementing) act on a single input. In general, an ALU therefore has two binary inputs and one binary output to handle data. It also has a binary control input to select the operation required. You can construct a simple ALU from your component **4bitadd** by adding a single control line to make it add and subtract as required.

Exercise 11. Your solution to Exercise 10 (designing a 4-bit subtractor) should have involved inverting the B inputs to the 4-bit adder, and altering the value of Cin. Design and construct a circuit that uses **XOR** gates to implement these changes under the control of a signal called **add_sub**. Save the circuit under the name **addsub**. Save, check and compile the design and test its operation. Remember to add the required files before compiling the project and create a default symbol for it as shown in Figure 21.

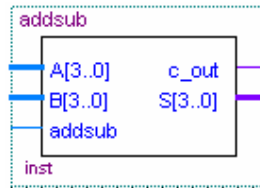


Figure 21 Example symbol - addsub component

8.1 An improved ALU

Additional functions can be added to extend the capabilities of the ALU. At the moment it can perform the operations **ADD** ($S = A + B$) and **SUB** ($S = A - B$). Two further functions **INC** ($S = A + 1$) and **DEC** ($S = A - 1$) are often needed by microprocessors to act as loop counters. In order to add additional circuitry to your **addsub** component, to allow the B input to be switched between a B data word and the binary number 0001 under the control of a further select line, we need to understand the operation of and make use of **multiplexers**.

9.0 Gating and multiplexing

In addition to performing logical and arithmetic operations, logic gates can also be used to control and select signals. For example, Figure 22(a) shows an AND gate being used to enable or disable an output. A glance at the AND truth table (Figure 3) should convince you that when Enable = 1, Out = In; similarly, when Enable = 0, Out = 0 (independent of the value of In). As a result, Enable can be said to control or **gate** the value of In that is fed to Out.

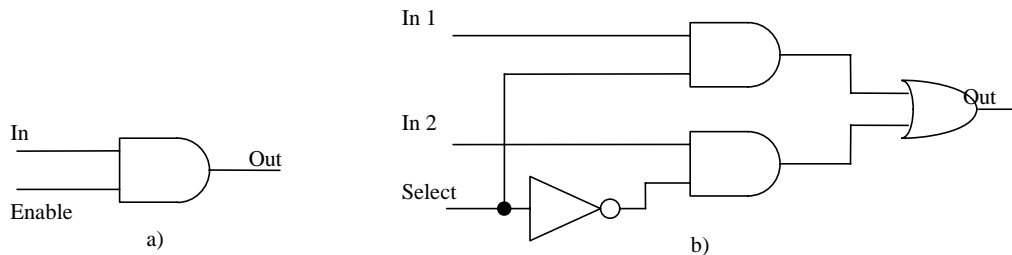


Figure 22 a) Signal gate; b) 2×1 multiplexer

When combined with additional logic, control gates can be used to select which of a number of inputs is fed to an output. For example, Figure 22(b) shows a **2×1 multiplexer**. Here two inputs In0 and In1 are gated by signals Select and $\overline{\text{Select}}$, respectively. The gated outputs are combined by an OR gate into the final output Out. When Select = 0, Out = In0, and when Select = 1, Out = In1. Note that an OR gate suffices as the final combiner, even though it can generate a high output when both of its inputs

are high. No confusion arises, because the select lines for In_0 and In_1 are inverses; as a result, only one input to the OR gate can be high at any one time.

9.1 Extending the range of a multiplexer

More complex devices can be constructed by combining multiplexers. For example, Figure 23 shows a **4 × 1 multiplexer** constructed from three 2 × 1 multiplexers. There are four data inputs I3 .. I0 and two select inputs S1 and S0. When considered together, the select lines act as a binary code identifying the data selected. For example, decimal 3 = binary 11; in the circuit, I3 is passed to the output when S1 = 1 and S0 = 1. Similarly, I2 is passed to the output when S1 = 1 and S0 = 0, and so on. S1 therefore acts as the high bit in the select code, and S0 as the low bit.

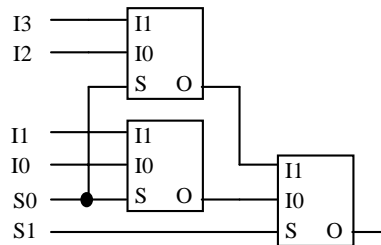


Figure 23 4x1 multiplexer

Exercise 12. Design an 8 x 1 multiplexer based on the 4×1 and the 2×1 multiplexer symbols shown in Figure 24 (a) and (b). (N.B. There is no need to enter this design into **Quartus II**).

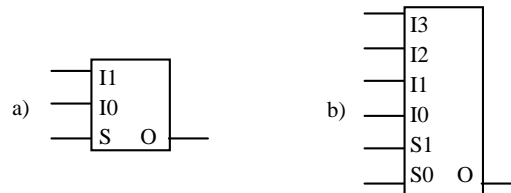


Figure 24 a) 2×1 and b) 4×1 symbols


10.0 Extending the ALU

Exercise 13. Figure 25 shows the truth table for the instructions ADD, SUB, INC and DEC in terms of the control lines **Set_B=1** and **Subtract**. Each of these control lines may be regarded as a bit in an **instruction code** with bit elements Instruction0 and Instruction1.

Instruction 0 Set_B=1	Instruction 1 Subtract	Operation
0	0	Add
0	1	SUB
1	0	INC
1	1	DEC

Figure 25 Truth table for the ALU instruction set, in terms of control lines and instructions

Your design for **addsub** already performs *addition* and *subtraction*. The INC and DEC instructions may be simply performed by setting B = 1 and then performing an addition or subtraction. Verify the truth table shown in Figure 25 and design and construct the 4-operation ALU (use a multiplexer from the **others>maxplus2**, for example, the 74157 which you will find in the mf library, or make your own!). Using the simulator, test its operation. By now the circuit is getting quite complex, and you may need to extend the simulation time – with the waveform editor open, use Edit>End Time.

Exercise 14.  Modify the ALU and its instruction decoder to allow an additional instruction **ASL**, which shifts every bit in the A input to the left by one and places the result in S.

11.0 Sequential Logic

Up till now you have been involved in the design, construction and simulation of *combinatorial, non-sequential* logic circuits i.e. the outputs at any instant in time are entirely dependent upon the inputs present at that time. A *sequential* circuit is one in which the current outputs are also dependent upon *previous* inputs and/or outputs. There are two main types of sequential circuit, and their classification depends upon the *timing* of their signals. For a *synchronous* circuit, the behaviour may be evaluated from a knowledge of its signals at discrete instants in time (related to a *clock* signal). The behaviour of an *asynchronous* circuit, however, depends upon the *order* in which its input signals change and these may not be directly related to a clock signal. In the following exercises we will look only at the first type of sequential circuit, whose operation is governed by means of a clock signal. For this reason this type of circuit is often referred to as a *clocked sequential circuit*. A typical sequential logic circuit is shown in Figure 26.

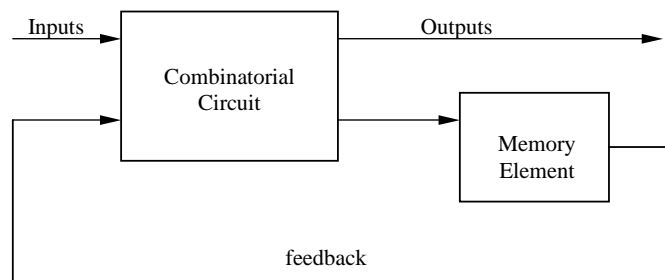


Figure 26 Block diagram of typical sequential circuit

11.1 A testing system for the ALU

When you were testing the operation of your ALU, you spent some time setting up the input waveforms so as to test its operation. We will now use a simple *counter* and clock circuit to generate the relevant numbers. This forms a test circuit for the ALU, to which it may be added. A counter is a simple device to understand in operation. It has (in the basic form) a single input and a single output bus, as shown in Figure 27.

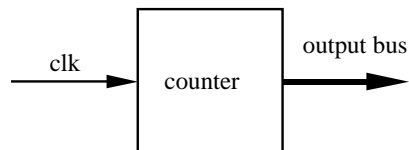



Figure 27 A counter

Upon the arrival of each clock pulse to the counter, the output bus line increments its value by one. The output for a two-bit counter hence follows the sequence {00, 01, 10, 11, 00,}.

Exercise 15.  Select the macrofunctions option from **help**, and then the **Counters** option. In order to test all the combinations of two 4-bit numbers as input to our ALU, we will need **an 8-bit counter**. Create a new **bdf** file and get the component **8 count** from the macrofunction library (**megafuncions>arithmetic>lpm_counter**). When you click ok a MegaWizard Manager will pop up, choose VHDL for output file type and a directory for the output file (e.g. H:\quartus2lab\test\count8). You should now see a dialogue box similar to the one shown in Figure 28.

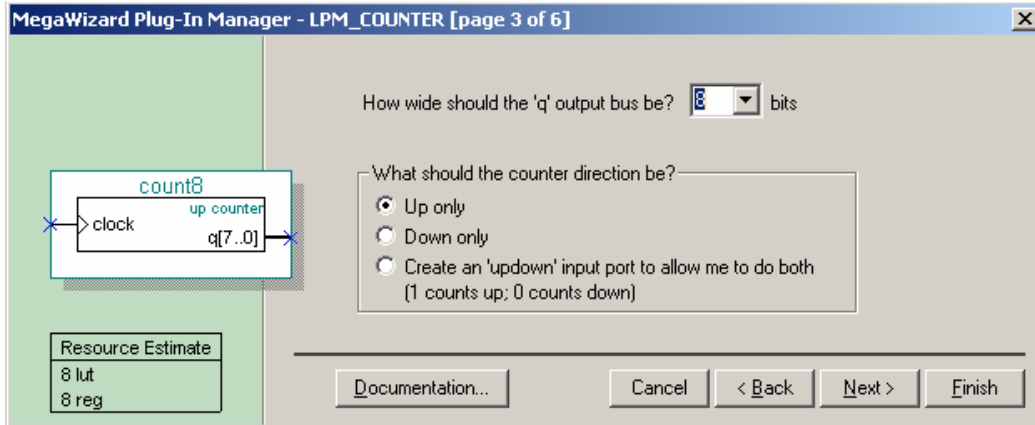


Figure 28 MegaWizard counter dialogue box

Configure the counter settings as shown in Figure 28 and click the **Finish** button. Add this counter symbol to your design.

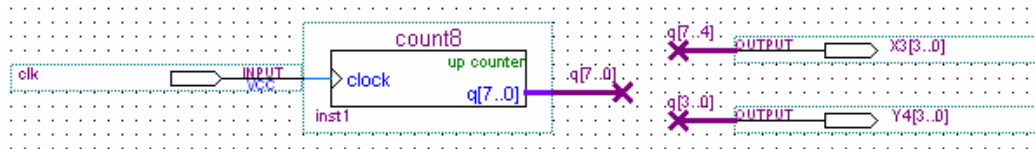


Figure 29 8-bit counter

You will note that the outputs **X** and **Y** correspond to the low and high words (4-bits) of the full 8-bit output from the counter. This means that for every value of **Y** (0000 to 1111), **X** will count from 0000 to 1111. This is precisely what we desire as our test outputs. Save the above design as **test** and then use it to construct, along with your **alu** design, a circuit called **alutest** which tests all combinations of 4-bit addition, subtraction, INC and DEC. Save and compile this design and make sure that the simulated results are what you would expect.

11.2 Adding feedback and memory

If we look at the diagram in Figure 30, we see that the sequential logic part of the ALU test circuit is hidden within the workings of the 8-bit counter, and that the rest of the circuit is combinatorial. We will now look at a simple circuit which follows the design of Figure 26. We need, however, to introduce a simple 'memory' element.

11.3 The D flip-flop

The memory elements of clocked sequential circuits are called **flip-flops**. These circuit elements are binary cells capable of storing 1 bit of information which they can maintain indefinitely (so long as power is supplied). Binary information can enter and exit a flip-flop in a variety of ways, which gives rise to several different types. In this experiment we will make use of a flip-flop type known as a **D flip-flop**. The D flip-flop is so called because of its ability to hold **data** and is sometimes referred to as a **gated D latch**. It is represented symbolically as shown in Figure 30(a) and may be constructed from 5 NAND gates as shown in Figure 30(b). **N.B.** There is no need to create a flip-flop from NAND gates though.

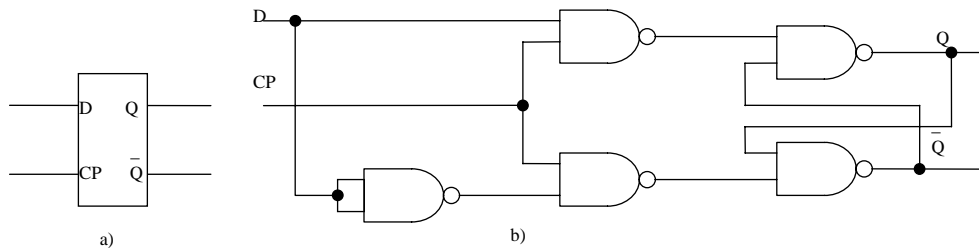


Figure 30 D flip-flop (a) symbol, (b) NAND level circuit

Figure 30 shows the characteristic table for the D flip-flop :

Q(t)	D	Q(t+1)
0	0	0
0	1	1
1	0	0
1	1	1

Figure 31 Characteristic table for the D flip-flop

As you can see, the output at the next clock pulse $Q(t+1)$ is equal to the **current** data input D and is **independent** of $Q(t)$, the current output. This device, therefore, operates as a one-cycle delay.

Exercise 16. Describe a relationship between the current output $S(t)$, the previous output $S(t-1)$ and the input $A(t)$ for the circuit shown below in Figure 32 (assume that all numbers are < binary 1111 so that carry bits may be ignored).

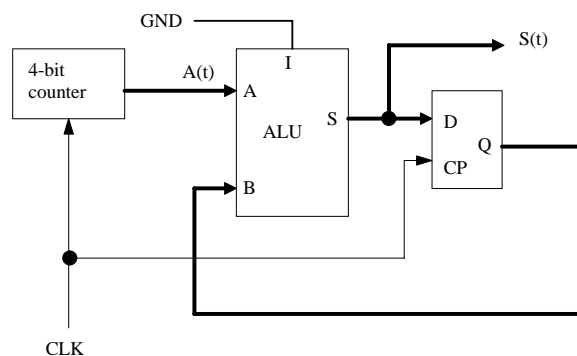



Figure 32 Clocked sequential circuit using the ALU

If A starts at binary 0000, what are the outputs for $A = 0000$ through to 0101? Using the **help** command of **Quartus II**, select **Registers Declaration**, general description. **Quartus II** has in the **others>maxplus2** library a 4-bit D flip-flop, the 74175. You may now begin entering a new design and select this register from **others>maxplus2** library, find out what signal you need to apply to the CLRn input. Together with your circuit **alutest**, create the circuit shown in Figure 32. Save the design and compile it. Using the simulator, test its behaviour. Check that it gives the correct outputs for $A =$ binary 0000 to 0101. What happens after this?

Exercise 17. (optional)

Some of you may have noticed that this circuit outputs the sequence of triangular numbers, so called

because they stack into triangles, i.e. 6 may be represented as  There is a simple relationship between the triangular numbers and the sequence of square numbers 0,1,4,9,16 etc. What is this relationship? Using your last circuit and another ALU circuit, design a circuit whose output is the sequence of squares (for a 4-bit system overflow will, of course, occur early in the sequence though).

Suggestions for further reading

- “[Digital Systems – Principles and Applications](#)”, 9th Ed, R. J. Tocci and N. S. Widmer, Prentice Hall, ISBN: 0131219316, 2004 (£45)
- “[Digital Fundamentals](#)”, T.L. Floyd, Prentice Hall, ISBN: 0-13-197255-3, June 2005 (£43)
- Digital Electronics I Course webpage:
http://www.ee.ic.ac.uk/pcheung/teaching/ee1_digital/index.html

VCFL/SJR/PYKC 2006.