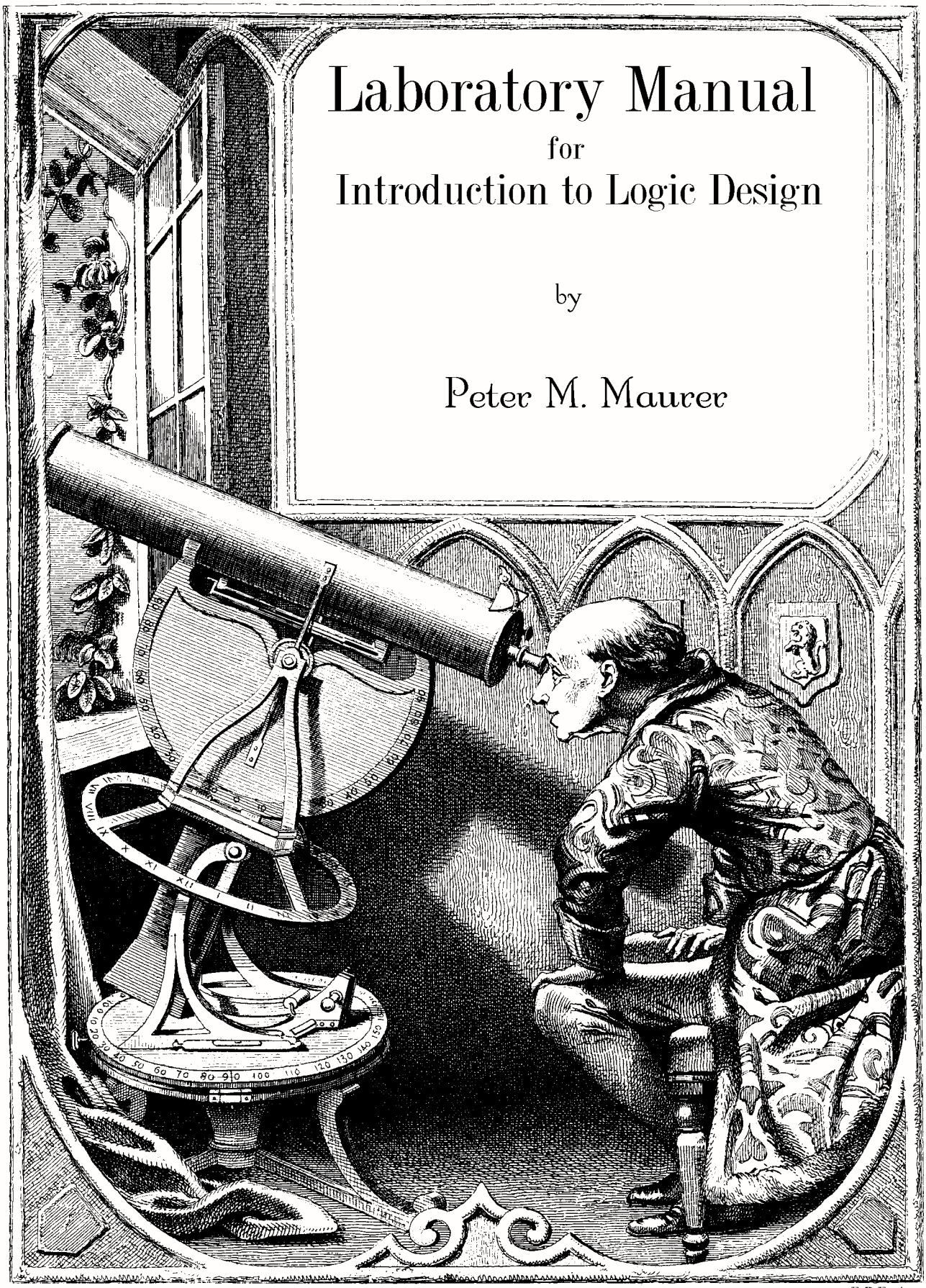
Laboratory Manual

for

Introduction to Logic Design

by

Peter M. Maurer



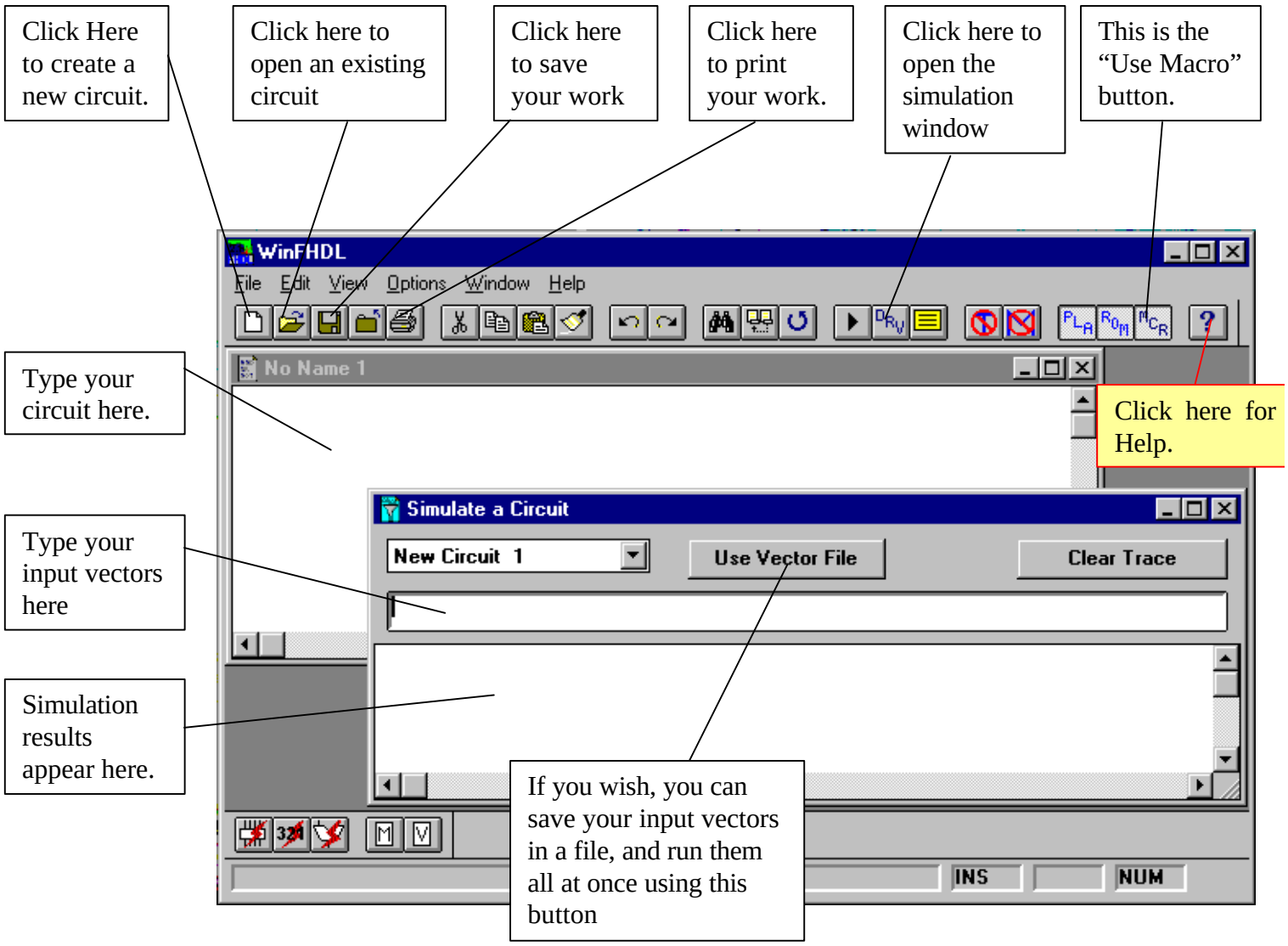
D. Maclise, R.A. F. P. Becker.

Table of Contents

Experiment 1:
Elementary Boolean Functions

Objective: Master the elementary Boolean functions, and learn how to use the basic features of the FHDL Simulator.

Instructions: WinFHDL is designed for MS Windows 3.1 and Windows 95. It is assumed that you already have a working knowledge of MS Windows and its environment before beginning this experiment. The FHDL Simulator program is located in the *FHDL system* program group. Copies of the WinFHDL installation diskettes should be available through your instructor. Alternatively, you may download the installation program from web site http://www.csee.usf.edu/~maurer/fhdl.html. This program will be used for all experiments. When running, the simulator program should look as follows.



To familiarize yourself with the simulator, create a simple circuit consisting of a single AND gate. This circuit is typed as follows.

```
MyCircuit:  circuit
            inputs    a,b
            outputs   q
gate1:      and       (a,b),q
            endcircuit
```

This circuit illustrates the format of FHDL statements. Each statement has a label field, an operation code, and an operand field. The label field may be blank. Labels always end with colons. If a statement has no label, then it must start with a blank. Labels always start in column 1. Gates, such as **and** have two operands, an input list and an output list. The lists must be enclosed in parentheses, however if either list contains only one element, the parentheses may be omitted.

To simulate the circuit, open the simulation window and type the following into the input vector area.

```
0,0
```

The "Simulation Trace" window should now contain the following two lines.

```
I>0,0
2>0
```

Any line beginning with **I>** contains input typed by you. Any line beginning with **n>** contains output produced by the simulator. (The letter *n* represents a number.)

Generate a complete truth table for the and gate by entering the following four additional vectors. Note that several vectors can be typed at once if you separate them with semi-colons.

```
0,1;1,0;1,1
```

Print your results using the Print Both command, which is found in the File menu.

Generate and print truth tables for the following gates.

```
gate2:   or     (a,b),q
gate3:   xor    (a,b),q
gate4:   not    a,q
gate5:   nor    (a,b),q
gate6:   nand   (a,b),q
gate7:   buff   a,q
```

gate8: xnor (a,b),q

Results: Turn in the printed truth tables for each function.

FHDL: This experiment has introduced several new types of FHDL statements. As the sample circuits illustrate, each circuit must begin with the **circuit** statement. The keyword **circuit** must be preceded by a label that names the circuit. As with all labels, the label on the **circuit** statement must not be preceded by any spaces, and must end with a colon(**:**). The colon is not considered to be part of the label. The colon must be followed by at least one space. Every circuit must end with the **endcircuit** statement. The **endcircuit** keyword must be preceded by at least one space.

The **inputs** and **outputs** statements define the primary inputs and outputs of a circuit. The **inputs** and **outputs** keywords must be preceded by at least one space. Following the keyword is a list of primary inputs or outputs of the circuit. The primary inputs are those inputs that receive new values when you type a new input vector, and the primary outputs are those outputs that are displayed in the **Simulation Trace** window. The lists of inputs and outputs are lists of names separated by commas. There should be no spaces in these lists. Each primary input should appear as an input to at least one gate. Each primary output should be the output of a gate.

The gate statement usually begins with a label, followed by a gate type. Strictly speaking, the label is unnecessary. However, if there are errors in the gate, the gate label will appear in the error message. Without this information, the error messages may be hard to diagnose. The gate type must be preceded and followed by at least one space. Following the gate type is a list of inputs enclosed in parentheses. The inputs may be primary inputs or internal connections. (See later experiments for internal connections.) If the list of inputs contains a single item, the parentheses may be omitted.

Following the list of inputs is a comma and a list of outputs contained in parentheses. If the list of outputs contains a single item, as is often the case, the parentheses may be omitted. The items in the input and output lists must be separated by commas. There should be no spaces in input/output specifications. Gate type may be one of the following: **and, or, not, xor, nand, nor, buff, xnor**. New FHDL features will be introduced in later experiments.

Experiment 2
The Properties of Boolean Functions

Objective: Experiment with different configurations of gates to verify some of the elementary laws of Boolean Algebra.

Instructions: The following equations represent the fundamental laws of Boolean Algebra.

Equation	Law
a•1=a	**And** Identity Law
a+0=a	**Or** Identity Law
ab = ba	Commutativity of **and**
a+b = b+a	Commutativity of **or**
(ab)c = a(bc)	Associativity of **and**
(a+b)+c = a+(b+c)	Associativity of **or**
a(b+c) = ab+ac	**And** distributes over **or**
a+bc = (a+b)(a+c)	**Or** distributes over **and**
aa'=0	**And** inverse law
a+a'=1	**Or** inverse law

These laws can be verified experimentally using the FHDL simulator**.** To do this, first start the FHDL simulator, then type the following circuit description, which will verify the AND Identity law.

```
AndIdentity: circuit
        inputs      a
        outputs     q
        one         c1
g1:     and         (a,c1),q
        endcircuit
```

Input the two vectors 0 and 1, and use the simulation trace to verify that the input and output are the same. Now create a new *Circuit Description* window by selecting the *New* command from the *File* menu, and clear the *Simulation Window* by selecting the *Clear Tr*ace command from the *Edit* menu. Then type the following circuit into the new *Circuit Description* window.

```
AndAssoc:  circuit
           inputs      a,b,c
           outputs     left,right
left1:     and         (a,b),x1
left2:     and         (x1,c),left
right1:    and         (b,c),x2
right2:    and         (a,x2),right
           endcircuit
```

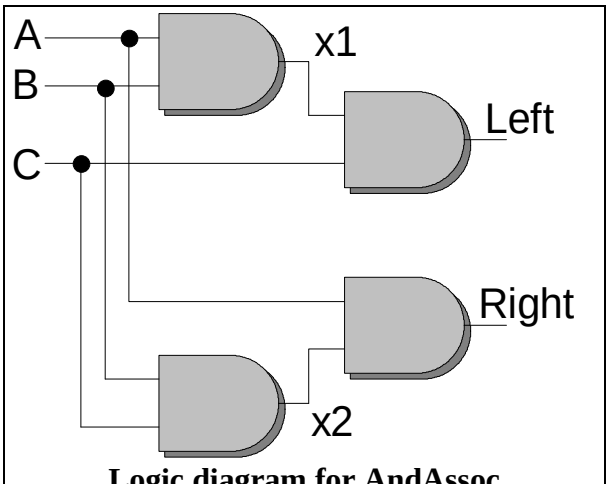
Input the full sequence of eight vectors, as listed below, and print both the circuit and the simulation trace.

```
0,0,0    1,0,0
0,0,1    1,0,1
0,1,0    1,1,0
0,1,1    1,1,1
```

Repeat this experiment with the other five laws listed above.
Verify that exclusive or (XOR) is both commutative and associative. Print the results.
Verify that NAND is commutative but not associative. Print the results.
Verify that NOR is commutative but not associative. Print the results.

Results: Turn in the printed output for each circuit. The circuit and the trace should appear on the same page.

FHDL: To complete this experiment it is necessary to master the art of transforming a Boolean equation into an FHDL description, and by implication into a network of gates. The first example illustrates how to declare constant signals. The **one** statement declares signals to be constant one, while the **zero** statement declares signals to be constant zero. These two statements have the same format as the **inputs** and **outputs** statements. The second example shows how to declare and use internal signals. In this example, the signals **x1** and **x2** are internal signals. An internal signal must appear as the output of one gate, and the input of one or mor other gates. Note that the FHDL equivalent of **a'** is the following statement.



Logic diagram for AndAssoc

```
gaprime:  not   a,aprime
```

Experiment 3
Theorems and Canonical Forms

Objective: This Experiment will allow you to construct circuits to verify some of the basic theorems of Boolean Algebra. It will also give you some experience with the canonical forms for Boolean functions. This experiment introduces FHDL gates with more than two inputs.

Instructions: The following equations represent some of the more important theorems of Boolean Algebra.

Equation	Theorem
(ab)' = a'+b'	Demorgan's Laws
(a+b)' = a'b'	Demorgan's Laws
a+a = a	Idempotency
aa = a	Idempotency
(a')' = a	Involution
a+1 = 1	Absorption
a•0 = 0	Absorption
a+ab = a	Absorption
a(a+b) = a	Absorption

Construct circuits to verify each of these laws. Print the circuit and simulation trace for each.

The following table lists the possible minterms and maxterms for a three-input Boolean function.

Input Combination	*Minterm*	*Maxterm*
0,0,0	**a'b'c'**	**a+b+c**
0,0,1	**a'b'c**	**a+b+c'**
0,1,0	**a'bc'**	**a+b'+c**
0,1,1	**a'bc**	**a+b'+c'**
1,0,0	**ab'c'**	**a'+b+c**
1,0,1	**ab'c**	**a'+b+c'**
1,1,0	**abc'**	**a'+b'+c**
1,1,1	**abc**	**a'+b'+c'**

Construct circuits for the minterms and maxterms that correspond to the input combinations 0,1,0, and 1,1,0. Simulate each of these four circuits using all eight input combinations. Print the circuit and simulation trace for each.

The following table gives the truth table for thee Boolean functions. Construct two circuits for each function, one that is a sum of minterms, and one that is a product of maxterms. Print the circuit and simulation trace for each.

Input Combination	*Function 1*	*Function 2*	*Function 3*
0,0,0	**0**	**0**	**1**
0,0,1	**1**	**0**	**0**
0,1,0	**0**	**0**	**1**
0,1,1	**0**	**1**	**1**
1,0,0	**1**	**0**	**0**
1,0,1	**0**	**1**	**1**
1,1,0	**0**	**1**	**1**
1,1,1	**1**	**1**	**0**

The following functions are more complicated than necessary. Reduce these functions to the minimum number of literals, using the theorems listed above, and the basic laws described in Experiment 2. Once you have reduced these functions, construct two circuits, one corresponding to the original equation, and one corresponding to the reduced equation. Simulate each using all input combinations to verify that they are the same.

```
ab+ab'
(a+b)(a+b')
abc+a'b+abc'
```

Results: Turn in printouts of all circuits, along with a simulation trace of each.

FHDL: To complete this experiment, it is necessary to use FHDL gates with more than two inputs. Gates of type **and**, **or**, **nand**, **nor**, **xor**, and **xnor** can have an arbitrary number of inputs. At least two inputs are required, but there is no upper limit. When implementing gates in hardware, there are limits imposed by the implementation technology, but FHDL imposes none. The following are some examples of multiple input gates.

```
g1:  and   (a,b,c),q1
g2:  or    (a,b,c,d,e,f,g),q2
g3:  nand  (a,b,c,d),q3
g4:  xor   (a,b,c),q4
```

Experiment 4
Two-Level Functions

Objective: This experiment will introduce you to the concept of two-level Boolean functions, and the techniques for minimizing such functions.

Instructions: Create two circuits for the following table, one using minterms, and one that is minimized to the smallest number of prime implicants. Use sum-of-products form. Use a Karnaugh map to minimize the function. Simulate both functions using all eight input combinations.

Input	*Value*	*Input*	*Value*
0,0,0	**0**	**1,0,0**	**0**
0,0,1	**0**	**1,0,1**	**1**
0,1,0	**1**	**1,1,0**	**0**
0,1,1	**1**	**1,1,1**	**1**

Minimize the following function using a Karnaugh map. Implement the function as an FHDL circuit, and simulate it using all input combinations to prove your minimized function is correct.

F(a,b,c,d) = Σ(0,2,3,4,11,12)

Express the following function as a product of maxterms. Implement the function in FHDL and simulate it using all eight input combinations. Minimize the function into product-of-sums form using a Karnaugh map. Implement the minimized function in FHDL and simulate it using all eight input combinations.

Input	*Value*	*Input*	*Value*
0,0,0	**0**	**1,0,0**	**0**
0,0,1	**0**	**1,0,1**	**1**
0,1,0	**0**	**1,1,0**	**1**
0,1,1	**1**	**1,1,1**	**0**

Minimize the following functions using the don't care conditions. Implement both the original and the minimized functions in FHDL, and simulate them with all input combinations to show they are identical, except for the Don't Cares. Use Karnaugh maps for the minimization.

F(a,b,c,d) = a'bc'd' + a'b'c'd + a'b'cd + a'bcd + abcd " ab'cd
DontCare(a,b,c,d) = a'b'c'd' + a'bc'd + a'bcd'

F(a,b,c) = a'b'c' + ab'c + a'bc
DontCare(a,b,c) = ab'c' + abc

Minimize the following function using a Karnaugh map. Give two different minimal sum-of-products equations for this function. Implement both of these functions in FHDL, and simulate them using all input combinations.

Input	*Value*	*Input*	*Value*	*Input*	*Value*	*Input*	*Value*
0,0,0,1	**0**	**0,1,0,1**	**0**	**1,0,0,1**	**0**	**1,1,0,1**	**0**
0,0,0,0	**0**	**0,1,0,0**	**1**	**1,0,0,0**	**1**	**1,1,0,0**	**1**
0,0,1,1	**0**	**0,1,1,1**	**1**	**1,0,1,1**	**1**	**1,1,1,1**	**1**
0,0,1,0	**0**	**0,1,1,0**	**1**	**1,0,1,0**	**1**	**1,1,1,0**	**0**

What does this last function tell us about minimal sum-of-products forms?

Results: Print both the circuit and the simulation trace for all circuits, and turn in the print-outs. Turn in all Karnaugh maps used for minimization.

FHDL: No new FHDL features are introduced in this experiment.

Experiment 5
Nand/Nor Implementations

Objective: This experiment will familiarize you with Nand-only and Nor-only 2-level implementations of Boolean functions.

Instructions: Recall that a Nand-only implementation starts with the sum-of-products form, while the Nor-only implementation starts with the product-of-sums form.

Implement the following function in four ways, sum-of-products, product-of-sums, Nand-only, and Nor-only. Simulate each circuit using all eight input combinations to prove that each is correct.

Input	*Value*	*Input*	*Value*
0,0,0	**0**	**1,0,0**	**0**
0,0,1	**0**	**1,0,1**	**1**
0,1,0	**1**	**1,1,0**	**0**
0,1,1	**1**	**1,1,1**	**1**

Implement the following function in four ways, sum-of-products, product-of-sums, Nand-only, and Nor-only. Simulate each circuit using all sixteen input combinations to prove that each is correct.

F(a,b,c,d) = Σ(0,2,3,4,11,12)

Minimize the following function using prime-implicant tables, and simulate the result using all 64 input combinations, to prove that your minimization is correct.

F(a,b,c,d,e,f) =Σ(3,11,12,13, 14,15,19,27, 28,29,30,31, 35,43,44,45, 46,47,48,49, 50,51,52,53, 54,55,56,57, 58,59,60,61, 62,63)

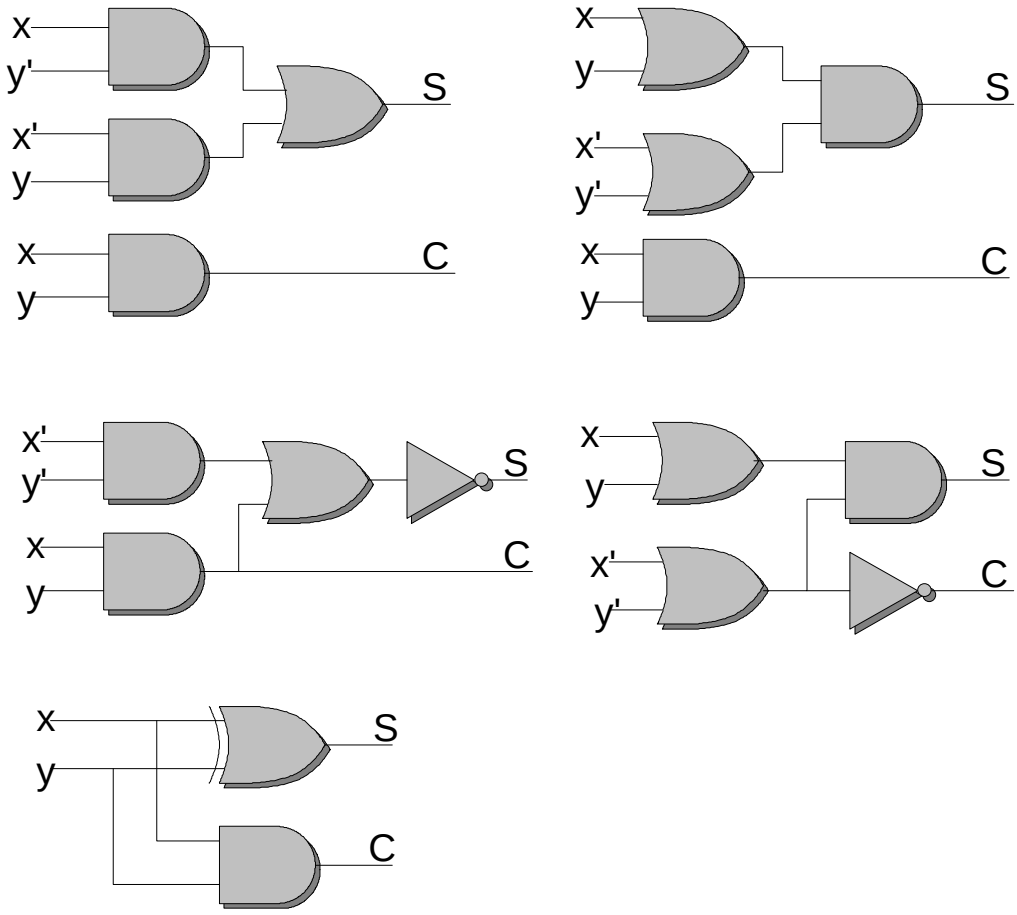
Results: Print both the circuit and the simulation trace for all circuits, and turn in the print-outs. Turn in all Karnaugh maps used for minimization.

FHDL: No new FHDL features are introduced in this experiment.

Experiment 6
Adders and Subtractors

Objective: In this experiment you will learn how to create single-bit adders and subtractors.

Instructions: Implement the following circuits in FHDL, and prove through simulation, that all perform the same function. Use NOT gates to obtain complements.



Implement a full adder as two two-level functions using the following table. Simulate the circuit to prove it is correct.

Inputs	Sum	Carry
0,0,0	**0**	**0**
0,0,1	**1**	**0**
0,1,0	**1**	**0**
0,1,1	**0**	**1**
1,0,0	**0**	**0**
1,0,1	**1**	**1**

1,1,0	**1**	**1**
1,1,1	**0**	**1**

Using your favorite half adder, implement the full adder as a combination of two half adders. Use the half-adder directly in a hierarchical circuit, as illustrated in the following circuit. Simulate the circuit to prove it is correct.

```
FullAdder:  circuit
            inputs      a,b,c
            outputs     s,co
g1:         HalfAdder   (a,b),(s1,co1)
g2:         HalfAdder   (c,s1),(s,co2)
g3:         or          (co1,co2),co
            endcircuit

HalfAdder:  circuit
            inputs      a,b
            outputs     s,c
            …
            endcircuit
```

FHDL Coding Rules for Hierarchical Circuits

1. The first circuit in a file is considered the main circuit, all other circuits in the same file are considered subcircuits.
2. Any subcircuit can be used as if it were a gate. This is known as a *call* to a subcircuit.
3. The call to a subcircuit must have the same number of inputs and outputs as the subcircuit definition.
4. The scope of net and gate names is a circuit, so name reuse is permitted between circuits.
5. Subcircuit names are case-sensitive, so “HalfAdder” and “halfadder” are considered different.
6. One subcircuit can contain a call to another subcircuit. There is no limit on the number of levels of calls.
7. Subcircuit calls *cannot* be recursive.
8. Except for putting the main circuit first, the order of the subcircuits is unimportant.

The following table describes a half-subtractor. A half-subtractor takes two inputs, and produces their difference, and a borrow. Implement the half-subtractor, and simulate it to prove it is correct.

Inputs	Difference	Borrow
0,0	**0**	**0**

0,1	**1**	**1**
1,0	**1**	**0**
1,1	**0**	**0**

The following table describes a full subtractor. (The third input to this circuit is a borrow-in from the previous stage.) Implement the subtractor, and simulate it to prove it is correct.

Inputs	Difference	Borrow
0,0,0	**0**	**0**
0,0,1	**1**	**1**
0,1,0	**1**	**1**
0,1,1	**0**	**1**
1,0,0	**1**	**0**
1,0,1	**0**	**0**
1,1,0	**0**	**0**
1,1,1	**1**	**1**

Results: Print the circuit and the simulation trace for all circuits, and turn in the print-outs. Turn in all Karnaugh maps used for minimization.

FHDL: This experiment introduces hierarchically specified circuits.

Experiment 7
Code Converters

Objective: This experiment introduces some circuits that are used for code conversions. New concepts include BCD digits, XS3 encoding, and Even/Odd parity.

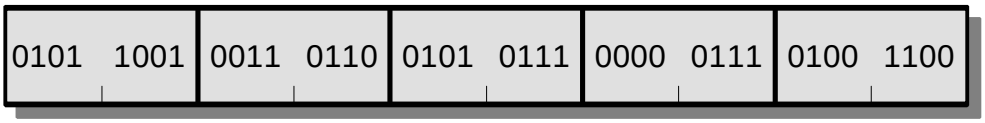
Instructions: The term "BCD" stands for Binary Coded Decimal. Many applications require the use of decimal digits. Financial transactions, for example, are typically done using decimal digits rather than binary. One reason for the use of decimal rather than binary, is that fractions that can be represented exactly in decimal end up as repeating binary expansions. This can introduce errors into computations that must be exact. To illustrate this problem, consider the fraction 1/5. In decimal notation, this fraction can be represented exactly as 0.2, however in binary, this fraction is $0.00110011\overline{0011}$.

In BCD, four bits are used to represent each binary digit as indicated in the following table.

Digit	Code
0	0000
1	0001
2	0010
3	0011
4	0100

Digit	Code
5	0101
6	0110
7	0111
8	1000
9	1001

The remaining six codes, 1010, 1011, 1100, 1101, 1110, and 1111 are considered to be invalid BCD digits. These codes are used to represent the sign of a multi-digit BCD number. The most common format for multi-digit BCD numbers is the *packed decimal* format. In this format, each 8-bit byte contains two decimal digits. The final digit of the final byte is considered to be a sign, and must contain one of the six codes, 1010 through 1111. The codes 1011 and 1101 are considered negative, while the rest are considered positive. The following diagram illustrates this format.



Assume you are given a 4-bit binary number that you want to convert to decimal. Two BCD digits will be required. The high-order digit will be 0000 or 0001. The following table shows the functions that must be computed to perform this conversion.

Input	D1-3	D1-2	D1-1	D1-0	D2-3	D2-2	D2-1	D2-0
0000	0	0	0	0	0	0	0	0
0001	0	0	0	1	0	0	0	0
0010	0	0	1	0	0	0	0	0
0011	0	0	1	1	0	0	0	0
0100	0	1	0	0	0	0	0	0
0101	0	1	0	1	0	0	0	0
0110	0	1	1	0	0	0	0	0
0111	0	1	1	1	0	0	0	0
1000	1	0	0	0	0	0	0	0
1001	1	0	0	1	0	0	0	0
1010	0	0	0	0	0	0	0	1
1011	0	0	0	1	0	0	0	1
1100	0	0	1	0	0	0	0	1
1101	0	0	1	1	0	0	0	1
1110	0	1	0	0	0	0	0	1
1111	0	1	0	1	0	0	0	1

Create a circuit that computes the eight functions given above. The circuit should have four inputs, and two outputs. For readability, the two BCD digits must be output as two 4-bit quantities. This is done using the following statements.

This statement declares a primary output to be 4-bits wide.

```
wire    d1,width=4
```

The next statement combines 4 one-bit signals into a single 4-bit signal.

```
collect    (d13,d12,d11,d10),d1
```

Taken together, these statements would be collected into a circuit description in the following manner.

```
MyCkt:  circuit
        inputs      a3,a2,a1,a0
        outputs     d2,d1
        wire        d1,width=4
        wire        d2,width=4
        collect     (d13,d12,d11,d10),d1
        collect     (d23,d22,d21,d20),d2
        /* logic to compute d10 through d23 */
        …
        endcircuit
```

Coding rules for the WIRE statement.

1. The keyword "wire" is case insensitive, so "wire," "Wire," and "WIRE" are all the same.
2. The keyword "wire" must be preceded by at least one space or tab.
3. The first operand must be the name of a signal used elsewhere in the circuit.
4. The position of the wire statement with respect to signal usage is unimportant. The wire statement can precede or follow the first usage of the signal.
5. The second parameter must be of the form "width=n" where n is a number from 1 to 32.
6. Using multiple wire statements for a single signal is not an error. The last specified wire statement takes precedence.
7. The wire statement takes other parameters, which will be mentioned in later experiments.

As mentioned above, the Collect statement is used to gather single-bit signals into a multi-bit signal. The Distribute statement is used to perform the opposite function. For example, suppose in the above circuit, you wished to specify the input as a single 4-bit signal. You would redefine the input as follows.

```
inputs   a
wire     a,width=4
```

You would then use the distribute statement to create the four inputs, as follows.

```
distribute   a,(a3,a2,a1,a0)
```

To FHDL the collect and distribute statements are gates, so the coding rules for these statements is the same as those for gates.

BCD is not the only scheme for encoding decimal digits. The XS3 code is a technique for encoding decimal digits that has some nicer mathematical properties than BCD. This code is illustrated in the following table.

Digit	Code
0	0011
1	0100
2	0101
3	0110
4	0111

Digit	Code
5	1000
6	1001
7	1010
8	1011
9	1100

XS3 stands for "Excess Three." An excess code is a code in which numbers are represented by binary quantities that are numerically larger than their value. In XS3, each

code is 3 larger than the numeric value of the digit. The codes 0000, 0001, 0010, 1101, 1110, and 1111 are considered invalid XS3 codes.

Create a circuit that transforms BCD into XS3. Both the input and the output must be 4-bit signals. In addition, the circuit must have a single-bit output indicating whether the input is a valid BCD digit. The following table describes the functions computed by this circuit.

Input	XS3-3	XS3-2	XS3-1	XS3-0	Invalid
0000	0	0	1	1	0
0001	0	1	0	0	0
0010	0	1	0	1	0
0011	0	1	1	0	0
0100	0	1	1	1	0
0101	1	0	0	0	0
0110	1	0	0	1	0
0111	1	0	1	0	0
1000	1	0	1	1	0
1001	1	1	0	0	0
1010	X	X	X	X	1
1011	X	X	X	X	1
1100	X	X	X	X	1
1101	X	X	X	X	1
1110	X	X	X	X	1
1111	X	X	X	X	1

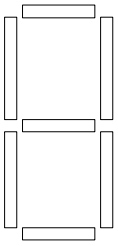
Results: Turn in print-outs of each circuit, along with the simulation results that prove the correctness of the circuits.

FHDL: This experiment introduces multi-bit signals.

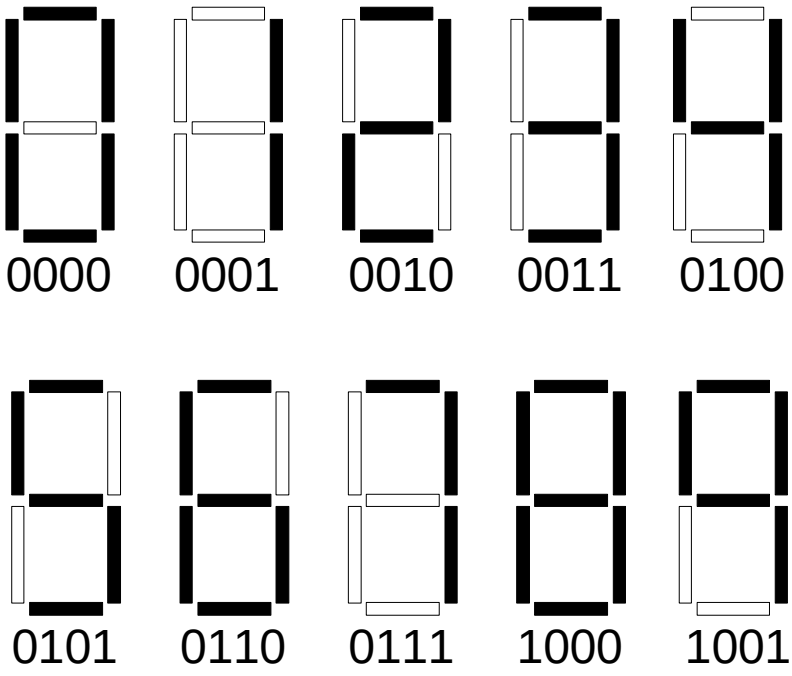
Experiment 8
Seven Segment Displays

Objective: This experiment familiarizes you with the usage of 7-segment displays.

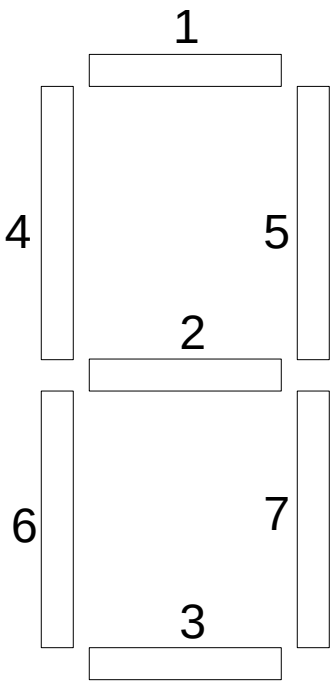
Instructions: BCD digits can be displayed using seven-segment displays. The seven-segment display consists of seven LED's arranged in the following pattern.



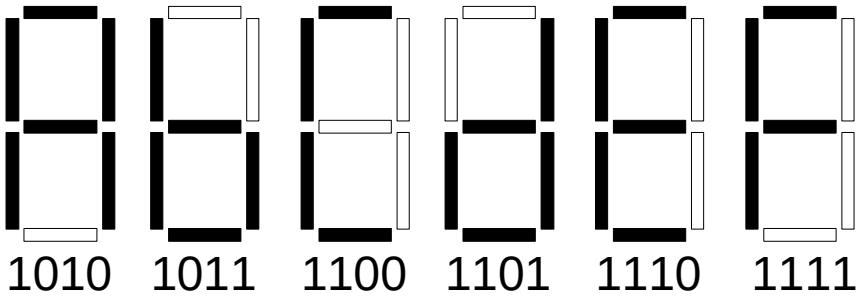
This device can be used to display decimal digits, depending on which segments are lit. The following diagram indicates the segments that must be lit for each BCD code.



For clarity, number the segments as illustrated below. Construct a circuit that accepts 4 single-bit inputs and creates seven single-bit outputs, specified in segment-number order. If the input is an invalid BCD digit, all outputs must be zero. This corresponds to blanking the display. (An output of one indicates that a segment is ON.)



The seven-segment display can also be used to display the invalid BCD codes as the letters A, B, C, D, E, and F. (The B is difficult to distinguish from the 6.) The following diagram illustrates how this is done.



Redesign your circuit to display these six additional digits. Include a fifth input, which will be used to blank the display. When this signal is equal to 1, all outputs must be zero.

Results: Both circuits must be tested with all possible input configurations to verify correctness. Turn in print-outs of the circuits, and the simulations.

FHDL: No new FHDL is introduced in this circuit.

Experiment 9
Multi-Level NAND/NOR Circuits

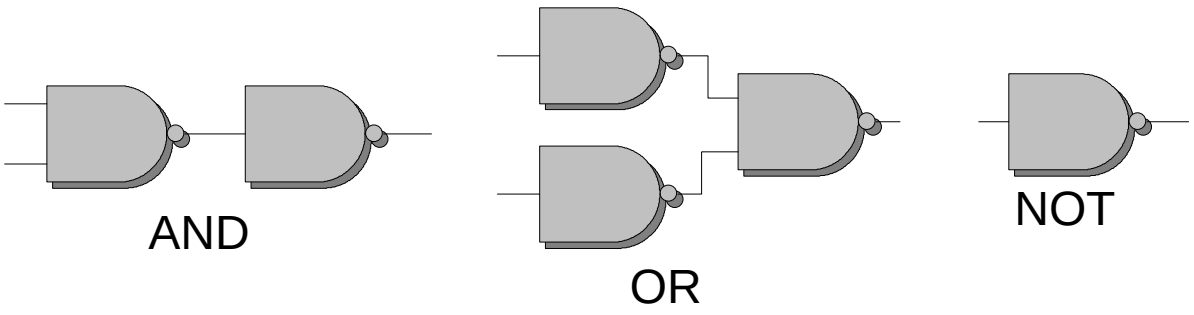
Objective: This experiment shows how to convert multi-level circuits into NAND-Only or NOR-Only implementations.

Instructions: In many circuit technologies, AND and OR gates are more difficult to construct than NAND and NOR gates. In fact, there are some design methodologies where the only available gates are NANDs or NORs. (One or the other, never both.)

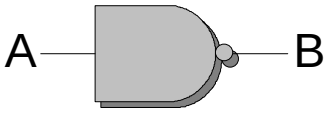
As noted in Experiment 5, two-level NAND implementations can be created from two-level sum-of-products implementations by replacing all gates with NAND gates. Two-level NOR implementations can be created just as easily. For multi-level circuits, the procedure is more complex.

It is well known that both the NAND gate and the NOR gate are universal. That is, any circuit can be constructed using only NAND gates or only NOR gates. In particular AND, NOT and OR functions can be implemented as NAND-Only or NOR-Only circuits. The method for converting multi-level circuits is to create NAND-Only or NOR-Only implementations of the basic gates, and do a direct substitution into the original circuit.

The NAND-Only implementation of the AND, OR and NOT functions are illustrated below.

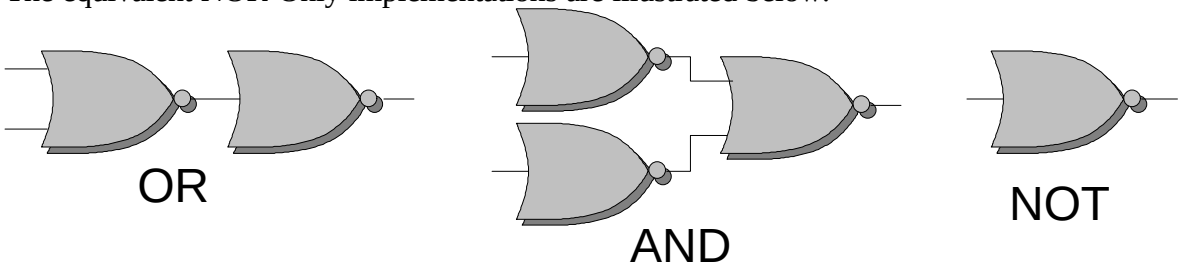


Note that in FHDL a single-input NAND gate is not allowed, so the single-input NAND must be coded as illustrated below.

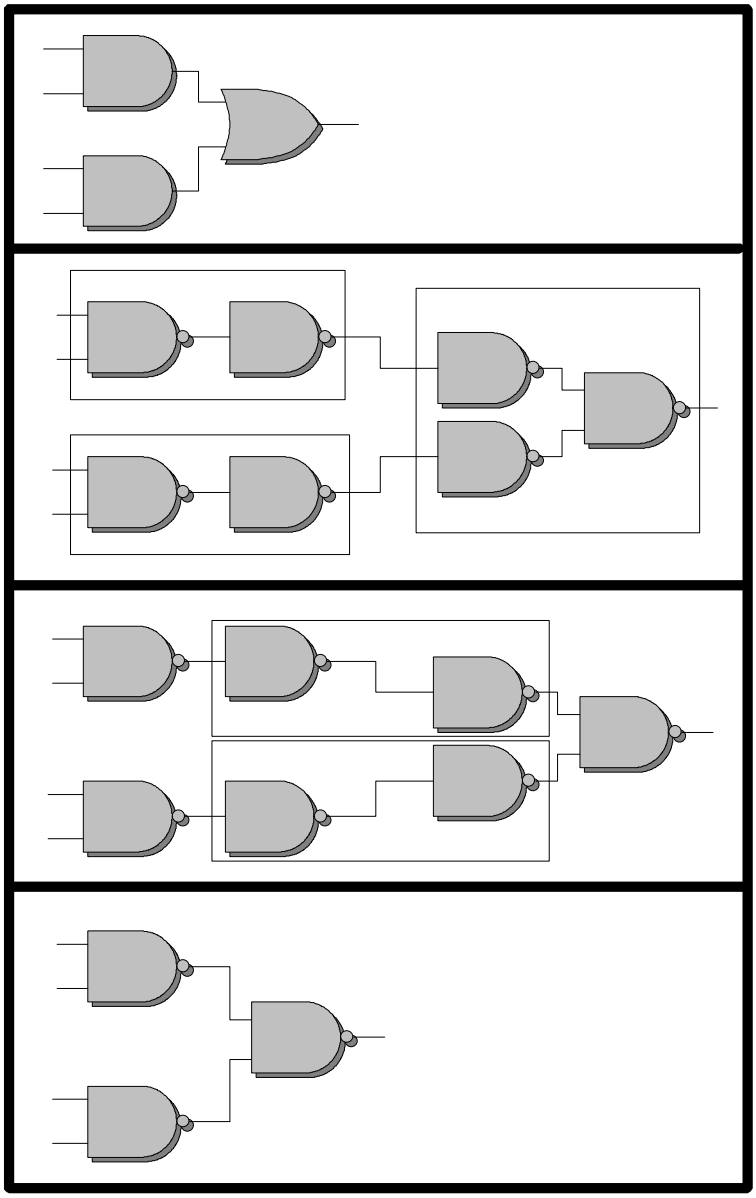



```
G1:     NAND     (A,A),B
```

The equivalent NOR-Only implementations are illustrated below.

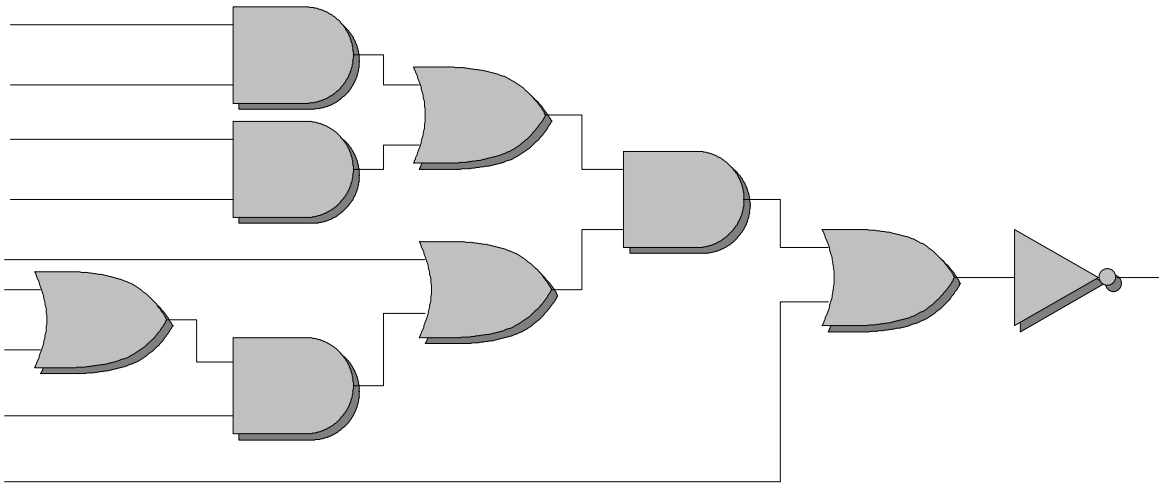


Doing the direct substitution will generally result in a large number of consecutive NOT gates. Pairs of consecutive NOT gates must be eliminated to simplify the circuit. This process is illustrated below.



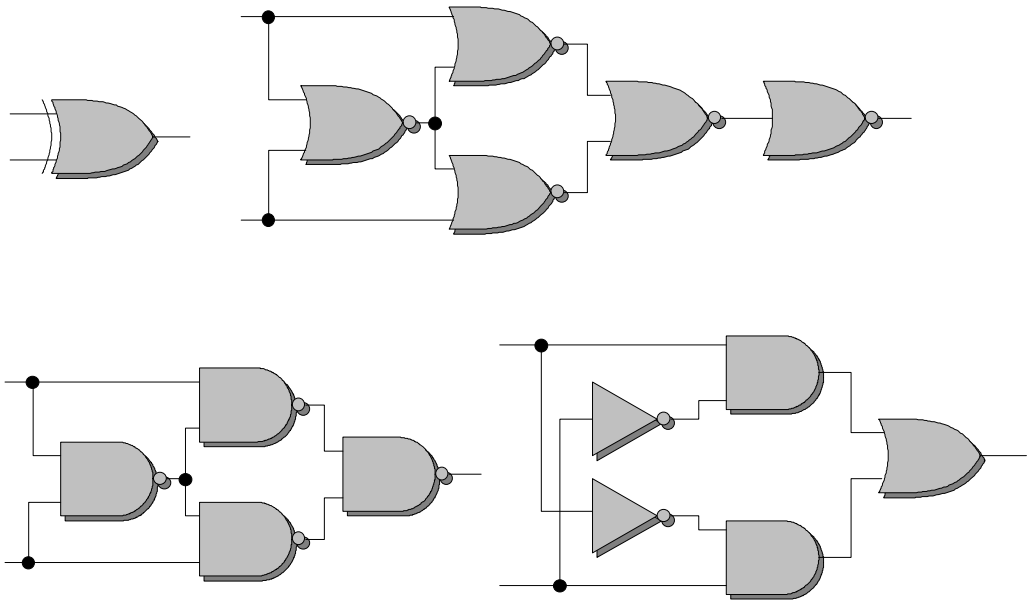
Implement the following circuit in the following 5 ways.

1. Implement it exactly as shown.
2. Do a direct substitution of elementary functions to create a NAND-Only implementation.
3. Optimize the NAND-Only implementation by eliminating consecutive NOT functions.
4. Do a direct substitution of elementary functions into the original diagram to create a NOR-Only implementation.
5. Optimize the NOR-Only implementation by eliminating consecutive NOT functions.



Simulate the circuit using several different input combinations to prove that all implementations are the same.

The NAND/NOR implementation of the exclusive OR function has not yet been mentioned. Direct implementations of this function are comparatively rare, although some efficient implementations are available if the complements of the inputs are readily available. In many cases, this function is implemented as a combination of AND and OR gates. The following a list of different XOR implementations. Code these circuits in FHDL and simulate them to prove they are identical.



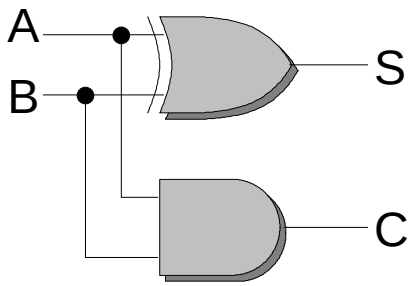
Results: Turn in print-outs of all circuits, as well as print-outs of all simulation runs.

FHDL: No new FHDL features are introduced in this experiment.

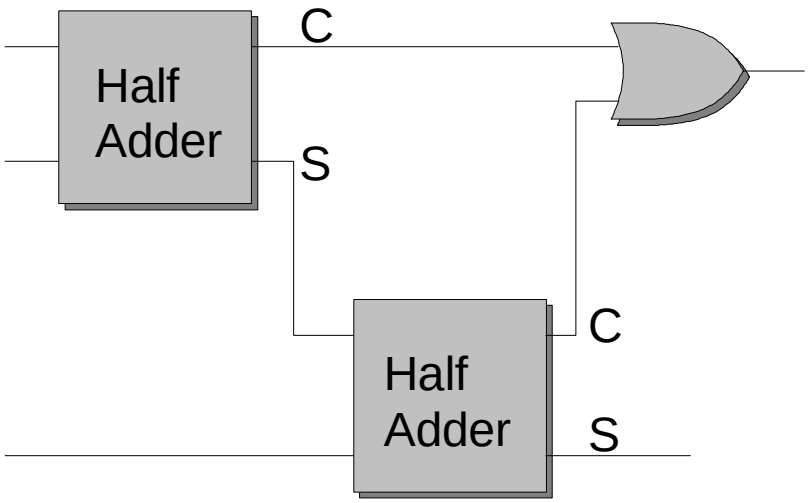
Experiment 10
Ripple Carry Adders

Objective: This experiment will teach you how to construct multiple-bit adders and subtractors.

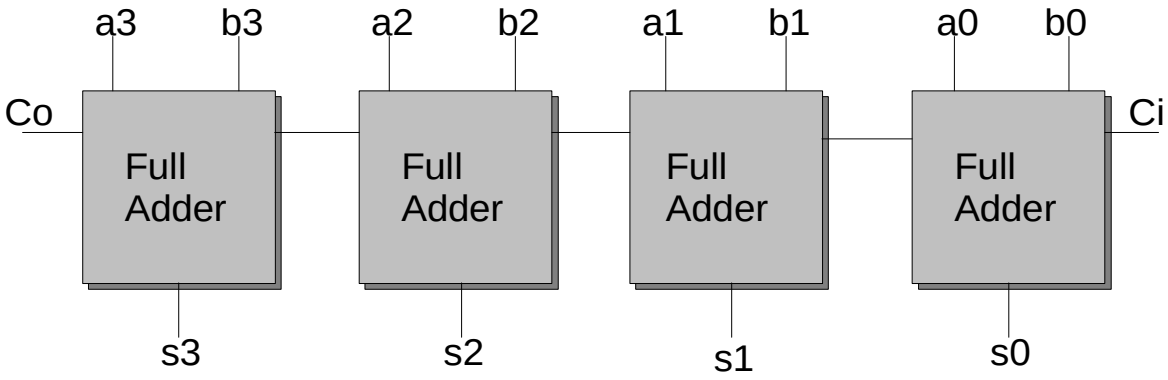
Instructions: Begin by constructing a half adder according to the following diagram.

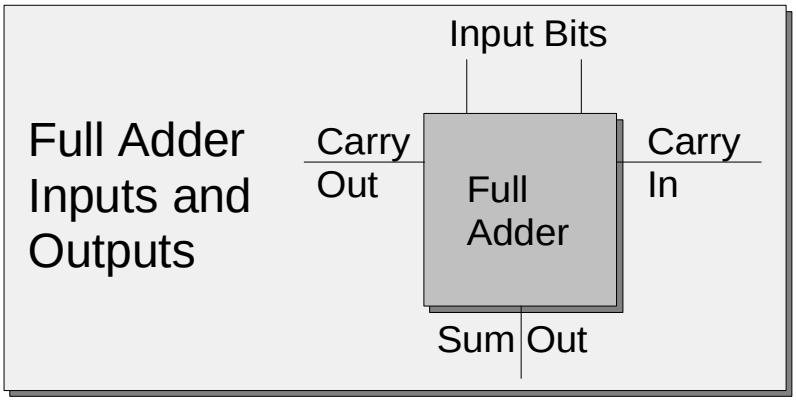


Use this half-adder as a gate to construct a full adder, according to the following diagram.



Use this full adder as a gate to construct the following 4-bit adder.





Use the 4-bit adder hierarchically to create a 16-bit adder.

Test each circuit in the hierarchy to make sure it is correct before advancing to the next step. Test the 16-bit adder with several input combinations to make sure it is correct.

Now, using the full subtractor circuit constructed in Experiment 6, construct a 4-bit subtractor, test it, and use it as a gate to construct a 16-bit subtractor. Test the 16-bit subtractor with several input combinations to verify that it is correct.

Results: Turn in printouts of all circuits and all simulations.

FHDL: This experiment provides a refresher course in hierarchical design.

Experiment 11
Carry Lookahead

Objective: This experiment will familiarize you with a faster method of performing addition than that given in Experiment 10.

Instructions: The 16-bit adder you constructed in Experiment 10 will be extremely slow, because it is possible for a carry-in to carry from the low-order stage through the high-order stage. The enormous number of gate delays in the carry chain will cause the circuit to perform slowly. It is possible to save time by computing the carries independently, using as few logic levels as possible. The first step in this computation is to determine whether a particular stage can generate a carry. The Generate function for two inputs is given by the following table.

Inputs	Generate
00	0
01	0
10	0
11	1

As can be seen from this table, a stage will generate a carry only if both inputs are equal to 1. The next step is to determine whether a stage will propagate a carry. In other words, if there is a carry in, will that carry be propagated through the stage to the carry-out. The following table describes this function.

Inputs	Propagate
00	0
01	1
10	1
11	Don't care

If both inputs are one, then the stage will generate a carry, so the carry-out will equal one regardless of the value of the carry-in.

The Generate function can be computed using an AND gate, and the Propagate function can be computed using an OR gate. When all propagate and generate functions have been computed, the carries can be computed using two-level sum-of-products equations.

For simplicity, assume that the stages are numbered from zero, starting with the low-order bit. The stage inputs are A_0-A_n, and B_0-B_n. The A and B inputs are used to compute the propagate and generate functions G_0-G_n, and P_0-P_n. The carry inputs of the stages are designated C_0-C_n. The carry-in to the whole adder is C_0, and the carry-out of the whole adder is C_{n+1}.

C_1 will be equal to 1 if G_0 is equal to 1, or if C_0 is equal to 1 and P_0 is equal to 1. This equation is written as follows.

$C_1 = G_0 + P_0C_0$

C_2 will be equal to 1, if stage 1 generates a carry or if it propagates a carry generated by stage 0 or if both stage 1 and stage 0 propagate a carry from C_0. This equation is written as follows.

$C_2 = G_1 + P_1G_0 + P_1P_0C_0$

By now the pattern should be obvious, so the equation for C_3 is as follows.

$C_3 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$

Write your own equations for C_4, C_5, and C_6.

Once the carries have been computed, it is possible to compute the sum bits using a 3-input exclusive-or. However, this can be slow. Another technique is to use the propagate and generate functions to compute an intermediate sum. The following equation shows how this is done.

$IS_1 = P_1G_1'$ (Propagate and not Generate).

The intermediate sum can be computed simultaneously with the carry bits. The true sum is the exclusive-or of the intermediate sum and the carry bit, as in the following equation.

$S_1 = IS_1 \oplus C_1$

Using these equations, create a 6-bit carry-lookahead adder. Test this adder using several input combinations to guarantee it is correct.

Use these equations to create a 4-bit carry-lookahead adder. Test this adder using several input combinations to guarantee it is correct. Use this adder as a gate to construct a 16-bit carry-lookahead adder. Test the 16-bit adder using several input combinations to guarantee it is correct.

Results: Turn in printouts of all circuits and simulations.

FHDL: There is no new FHDL introduced in this experiment.

Experiment 12
Adder/Subtractors

Objective: This experiment will show you how to construct a circuit that is capable of both addition and subtraction.

Instructions: Before dealing with the problem of creating an Adder/Subtractor, it is necessary to first deal with the problem of representing negative numbers. No arithmetic system would be complete without some representation of negative numbers, yet the circuits that we have designed so far have assumed that all of their operands are positive. At the same time, it is also necessary to deal with the problem of overflow. In other words, we must design our circuits in such a way that there is some indication when the result is too large to be represented in a given number of bits.

For unsigned numbers, the carry-out of the high-order stage can be used to test for overflow. If the carry-out is equal to one, then overflow has occurred, otherwise it has not. As we will see, the problem is more complex when dealing with negative numbers.

Conceptually, the simplest way to represent negative numbers is in *signed-magnitude* form. In this form, an extra bit is added to represent the sign. It is customary to use 1 for the negative sign and 0 for the positive sign. (This will be true regardless of how we represent negative numbers.) The value, or magnitude, of the number is represented as an unsigned binary quantity. Using 4-bit magnitudes, a positive 3 would be represented as 00011, and a negative 3 would be represented as 10011. Unfortunately, performing arithmetic with signed-magnitude numbers is quite complicated. The following procedure is used to add two numbers, A and B.

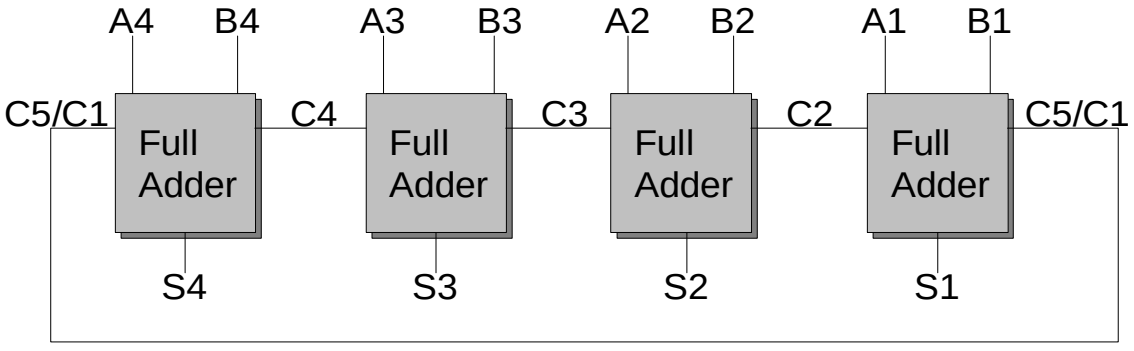
```
If A.Sign = B.Sign Then
        R.Sign = A.Sign
        R.Magnitude = A.Magnitude + B.Magnitude
        Set Overflow indicator to carry-out of high order bit
Else If A.Magnitude = B.Magnitude Then
        R.Sign = 0
        R.Magnitude = 0
        Set Overflow indicator to 0
Else If A.Magnitude > B.Magnitude Then
        R.Sign = A.Sign
        R.Magnitude = A.Magnitude – B.Magnitude
        Set Overflow indicator to 0
Else
        R.Sign = B.Sign
        R.Magnitude = B.Magnitude – A.Magnitude
        Set Overflow indicator to 0
```

EndIf

This level of complexity is beyond our skills at this time. Fortunately, there are ways to represent negative numbers that can simplify the addition algorithm. One such technique is to represent the magnitude of negative numbers in ones-complement form. To represent a number in ones-complement form, simply invert every bit of the magnitude. The numbers +3 and –3 would be represented as 00011 and 11100 in ones-complement form. There are two representations of zero, 00000 and 11111. To negate a number, invert every bit, including the sign. Using an ordinary 5-bit adder, adding +3 and –3 would yield 11111, one representation of zero.

There are some problems with one's complement representation, however. For example, consider adding two negative numbers –3 and –1. These numbers would be represented as 11100 and 11110. When adding these numbers with an ordinary adder, the result is 11010. This is –5, not –4, the desired result. A similar problem occurs when adding a positive number to a negative number, and obtaining a positive result. Suppose 3 is added to –1. These numbers are represented as 00011 and 11110. When added with a conventional adder, the result is 00001, which is +1, not +2, the desired result.

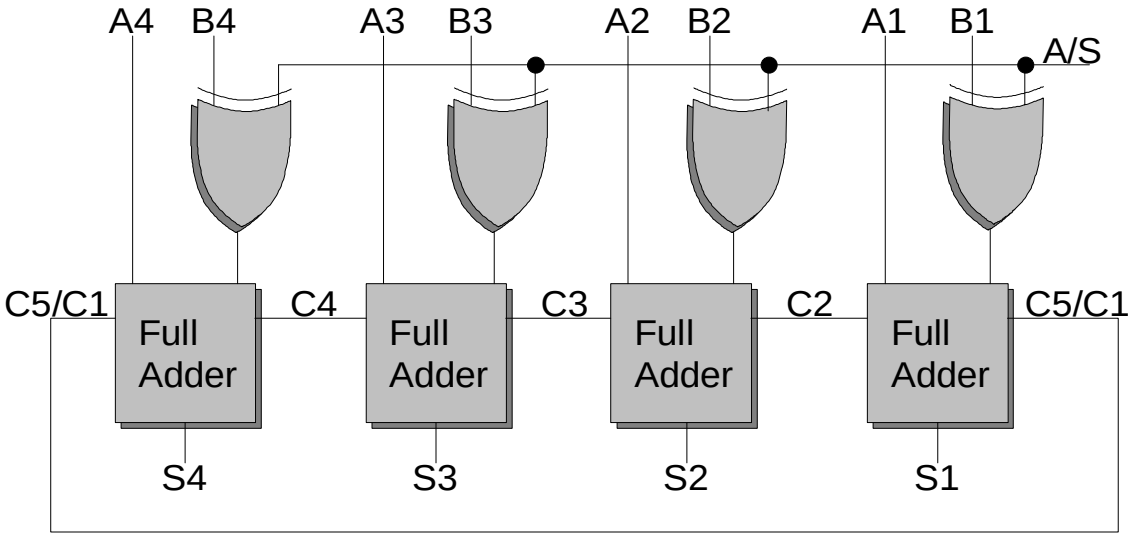
A careful examination of simulation results will show that the result is incorrect whenever there is a carry out of the sign position, and no overflow occurs. When the result is incorrect, it can be corrected by adding 1 to the result. Thus, the conventional adder can be "fixed" by feeding the carry-out of the sign position into the carry-in of the low-order bit as illustrated below. This creates a circular dependency in the logic, which may cause problems with some simulators. (WinFhdl will have no problem with this circuit.) Despite the circular dependency, the circuit is not sequential, because the carry into the low-order bit can never be propagated out of the sign position. Overflow is detected by comparing the signs of the input operands to the sign of the result. If the two input operands have the same sign, and the sign of the result is different from the sign of the operands, then overflow has occurred. No overflow can occur for operands of opposite sign.



Construct a 5-bit ones complement adder, and test it with several ones-complement numbers. Use the ripple-carry adder of Experiment 10.

The adder can be converted into an adder-subtractor by providing logic to selectively complement all bits of one input. A 2-input XOR gate can be used for this purpose. If we

consider one input to the XOR to be a control input, and the other to be a data input, the XOR will pass the data input through unchanged when the control input is zero, and will complement the data input when the control input is one. The following diagram illustrates how to convert a 4-bit ones-complement adder into a 4-bit adder/subtractor.



Create a 5-bit ones-complement adder/subtractor, and test it using several input combinations.

Despite its advantages, ones-complement notation has several undesirable properties. The first is the end-around carry necessary to correct results. Even if carry-lookahead is used, the low-order carries cannot be accurately computed until the high-order carry is available. Another undesirable property is the dual representation of zero. This will complicate comparisons, since any comparator must take the two representations into account. Finally, detection of overflow is complicated. The ones-complement notation does have the advantage that conventional adders can be used for addition with only minor modifications.

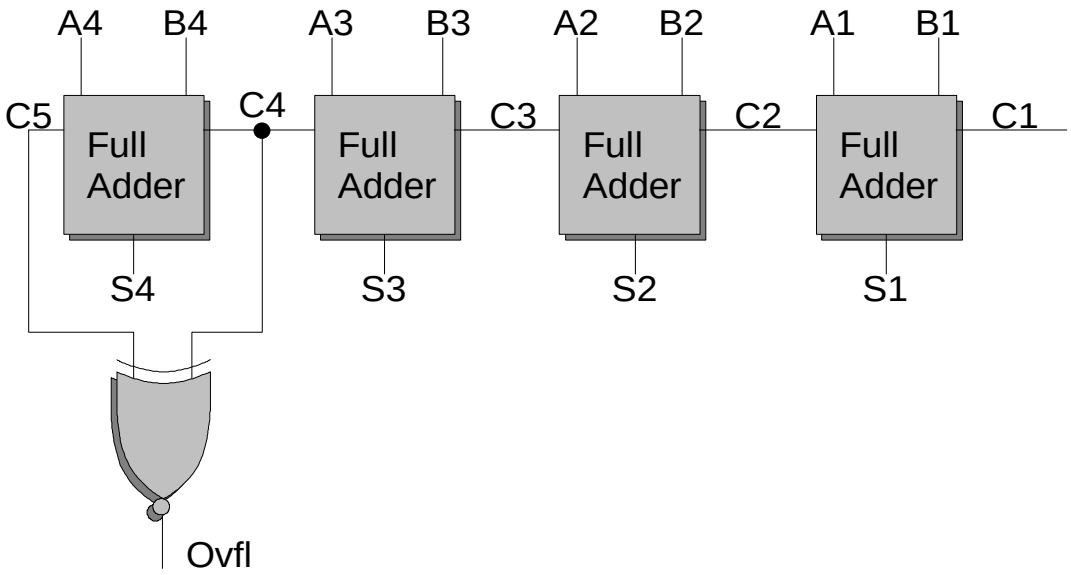
Representing numbers in twos-complement notation allows one to retain the desirable properties of the ones-complement notation, and at the same time eliminates most of the problems with ones-complement notation. To obtain the twos-complement of a number, first form the ones complement by inverting all bits of the number, then add 1 and ignore the carry-out. An equivalent procedure is to locate the rightmost one, and invert all bits to the left of it. The rightmost one, and the zeros to the right of it are left unchanged. The following table gives the twos-complement of several 5-bit numbers. Performing the twos-complement operation twice gives the original number

Number	Complement
00011	11101
01111	10001
10000	10000

Number	Complement
01000	11000
01010	10110
01100	10100

00000	00000	00010	11101

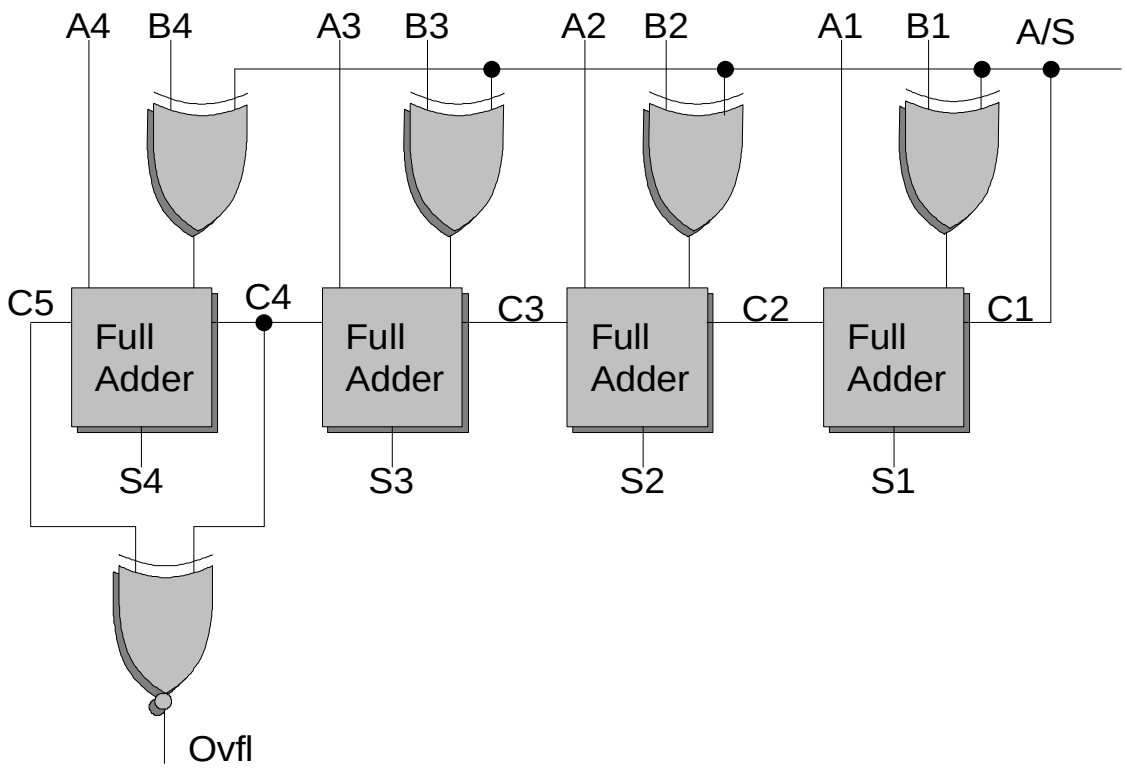
The only difference between a twos-complement adder and an ordinary adder is the logic for detecting overflow. Carry-lookahead logic can be used without modification. Overflow is detected by comparing the carry into and out of the sign position. If the two carries are different, then an overflow has occurred. Otherwise, there is no overflow. The following diagram shows a 4-bit twos-complement adder.



Create a 5-bit twos-complement adder, and test it using several input combinations.

Surprisingly, creating a twos-complement adder/subtractor is similar to creating a ones-complement adder/subtractor. XOR gates are used to complement one input. (This accomplishes the first step of the twos-complement operation.) The control signal is also fed into the carry-in to the first stage, adding 1 to the result when a subtraction is being done. (This completes the second step of the twos-complement operation.) The following diagram illustrates a 4-bit twos-complement adder/subtractor.

Create a 5-bit Adder/Subtractor and test it using several input combinations.



Results: Turn in printouts of all circuits and printouts of all simulations.

FHDL: No new FHDL features are introduced in this experiment.

Experiment 13
BCD Adders

Objective: This experiment shows how to construct a single-stage BCD adder, and how to connect them into a multi-stage BCD adder.

Instructions: The following table shows the BCD encoding of decimal digits. Each decimal digit is represented by a 4-bit combination. The remaining combinations are considered invalid.

Digit	BCD Code
0	0000
1	0001
2	0010
3	0011
4	0100

Digit	BCD Code
5	0101
6	0110
7	0111
8	1000
9	1001

The following table shows how to compute the 4-digit sum output of the BCD adder, when the carry-in is zero.

	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
0000	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
0001	0001	0010	0011	0100	0101	0110	0111	1000	1001	0000
0010	0010	0011	0100	0101	0110	0111	1000	1001	0000	0001
0011	0011	0100	0101	0110	0111	1000	1001	0000	0001	0010
0100	0100	0101	0110	0111	1000	1001	0000	0001	0010	0011
0101	0101	0110	0111	1000	1001	0000	0001	0010	0011	0100
0110	0110	0111	1000	1001	0000	0001	0010	0011	0100	0101
0111	0111	1000	1001	0000	0001	0010	0011	0100	0101	0110
1000	1000	1001	0000	0001	0010	0011	0100	0101	0110	0111
1001	1001	0000	0001	0010	0011	0100	0101	0110	0111	1000

The following table shows how to compute the 4-digit sum output of the BCD adder, when the carry-in is one.

	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	0000
0001	0010	0011	0100	0101	0110	0111	1000	1001	0000	0001
0010	0011	0100	0101	0110	0111	1000	1001	0000	0001	0010
0011	0100	0101	0110	0111	1000	1001	0000	0001	0010	0011
0100	0101	0110	0111	1000	1001	0000	0001	0010	0011	0100
0101	0110	0111	1000	1001	0000	0001	0010	0011	0100	0101
0110	0111	1000	1001	0000	0001	0010	0011	0100	0101	0110
0111	1000	1001	0000	0001	0010	0011	0100	0101	0110	0111
1000	1001	0000	0001	0010	0011	0100	0101	0110	0111	1000

1001	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

The following table shows how to compute the 1-bit carryout of the BCD adder when the carry-in is equal to zero.

	0000	**0001**	**0010**	**0011**	**0100**	**0101**	**0110**	**0111**	**1000**	**1001**
0000	0	0	0	0	0	0	0	0	0	0
0001	0	0	0	0	0	0	0	0	0	1
0010	0	0	0	0	0	0	0	0	1	1
0011	0	0	0	0	0	0	0	1	1	1
0100	0	0	0	0	0	0	1	1	1	1
0101	0	0	0	0	0	1	1	1	1	1
0110	0	0	0	0	1	1	1	1	1	1
0111	0	0	0	1	1	1	1	1	1	1
1000	0	0	1	1	1	1	1	1	1	1
1001	0	1	1	1	1	1	1	1	1	1

The following table shows how to compute the 1-bit carryout of the BCD adder when the carry-in is equal to one.

	0000	**0001**	**0010**	**0011**	**0100**	**0101**	**0110**	**0111**	**1000**	**1001**
0000	0	0	0	0	0	0	0	0	0	1
0001	0	0	0	0	0	0	0	0	1	1
0010	0	0	0	0	0	0	0	1	1	1
0011	0	0	0	0	0	0	1	1	1	1
0100	0	0	0	0	0	1	1	1	1	1
0101	0	0	0	0	1	1	1	1	1	1
0110	0	0	0	1	1	1	1	1	1	1
0111	0	0	1	1	1	1	1	1	1	1
1000	0	1	1	1	1	1	1	1	1	1
1001	1	1	1	1	1	1	1	1	1	1

Create a BCD adder using these tables, and test it using several inputs.

An alternative procedure is to create a BCD adder from two 4-bit binary adders. First compute the binary sum of the two BCD digits. A BCD carryout is generated if the carry-out of the 4-bit adder is equal to one, or if sum-bit 4 is equal to one and either sum-bit 3 or sum bit 2 is equal to 1. The logic equation for the BCD carry is as follows.

$$BC = C_5 + S_4S_3 + S_4S_2$$

If the BCD carry is equal to 1, then it is necessary to add 6 (0110) to the intermediate sum. Assuming that the inputs to the second adder are A4, A3, A2, A1, B4, B3, B2, B1,

these inputs must be assigned as follows. A4=0, A3=BC, A2=BC, A1=0, B4=S4, B3=S3, B2=S2, B1=S1. The carry-in of this adder must be set to zero.

Create a second BCD adder from two 4-bit adders and additional logic gates. Create one 4-bit adder and use it twice as a gate.

Results: Turn in printouts of all circuits and printouts of all simulations.

FHDL: No new FHDL is introduced in this experiment.

Experiment 14
Comparators

Objective: This experiment shows how to construct single-stage and multi-stage magnitude comparators

Instructions: A magnitude comparator is a circuit that takes two unsigned n-bit numbers, A and B, as inputs, and produces three single bit outputs, A<B, A=B, and A>B. For a 4-bit magnitude comparator the input would be two numbers, A and B, whose individual bits are designated A_4,A_3,A_2,A_1 and B_4,B_3,B_2,B_1. (B_1 is the low order bit.) The first step in creating the comparator is to compute greater than, less than, and equal functions for each bit. For bit 1, functions are given by the following equations.

$AG_1 = A_1B_1'$
$AL_1 = A_1'B_1$
$AE_1 = (AL_1 + AG_1)'$

These equations can be entered directly into FHDL, using the following syntax. NOTE: the USE MACRO PROCESSOR button must be pressed to use this syntax!

```
Comp: circuit
        inputs  A4,A3,A2,A1,B4,B3,B2,B1
        outputs AL,AE,AG

        equation        AG1=A1&~B1
        equation        AL1=~A1&B1
        equation        AE1=~(AL1|AG1)
        …
        endcircuit
```

Once the functions are computed for the individual bits, the 4-bit functions are computed using the following equations. AL indicates A<B, AG indicates A>B, and AE indicates A=B.

$AE = AE_1AE_2AE_3AE_4$
$AG = AG_4 + AE_4AG_3 + AE_4AE_3AG_2 + AE_4AE_3AE_2AG_1$
$AL = AL_4 + AE_4AL_3 + AE_4AE_3AL_2 + AE_4AE_3AE_2AL_1$

Build a circuit using these equations directly, and test it using several input combinations.

This four-bit comparator can be modified to be used as one four-bit stage in a multi-bit comparator. The existing outputs, AL, AE, and AG, can be combined with three inputs from the next lower stage. If AL or AG is equal to 1, then the result of the comparison is

determined from the current stage. However if AE is equal to 1, then the result is determined by the next lower stage. Let the inputs from the next lowest stage be denoted by ALI, AGI, and AEI, and let the new outputs be denoted as ALO, AGO, and AEO. The new outputs are computed using the following FHDL-encoded equations.

```
equation    AEO = AEI & AE
equation    AGO = AG | AE & AGI
equation    ALO = AL | AE & ALI
```

When this circuit is used to create a multi-stage comparator, ALI, AGI, and AEI inputs of the lowest order stage must be set to constant zero, constant zero, and constant one, respectively.

Convert your 4-bit comparator in the manner described above, and use it as a gate to construct a 16-bit comparator. Test the circuit using several input combinations.

Results: Turn in a print-out of all circuits, and all simulations.

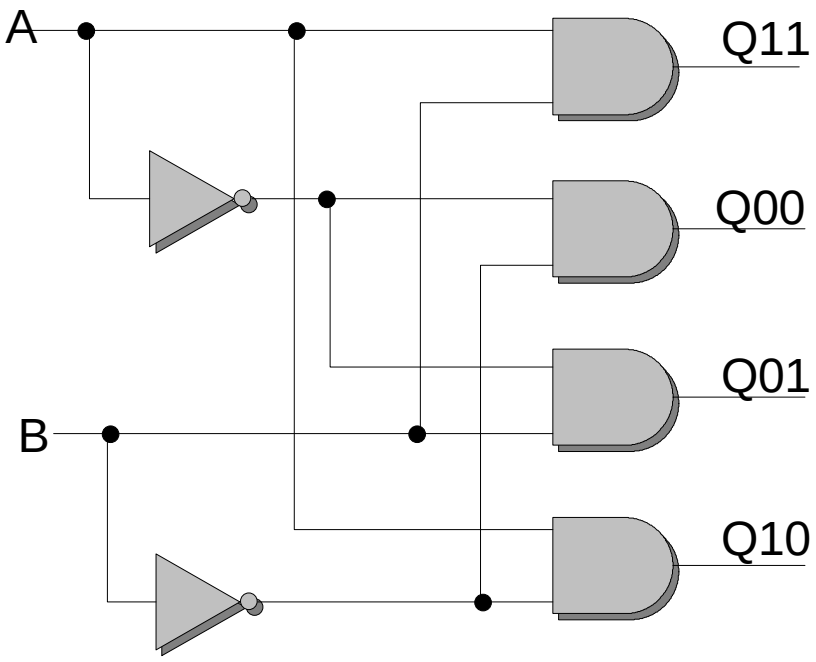
FHDL: This Experiment introduces the "equation" macro, which is part of the standard FHDL package.

Experiment 15
Decoders and Demultiplexers

Objective: This experiment shows how to construct and use decoders and demultiplexers.

Instructions: A decoder is a circuit which takes n inputs and produces 2^n outputs. For each input combination, exactly one of the 2^n outputs will be equal to 1. The rest will be equal to zero. Each of the 2^n input combinations is assigned to a particular output.

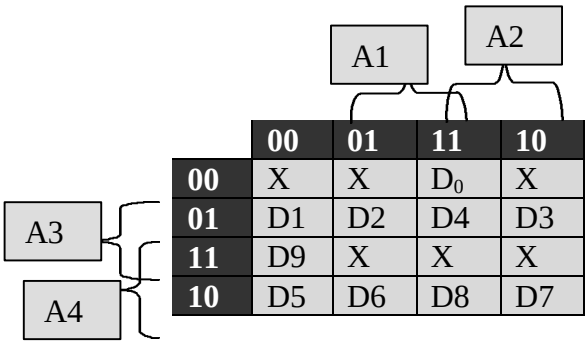
The following circuit is an example of a 2-input decoder.



A 3-input decoder would require 3 NOT gates and eight 3-input AND gates.

Create a 4-input decoder, and test it using various input combinations.

If certain input combinations can never occur, this creates *don't care* conditions in the decoder truth table. The presence of these don't care conditions will not only reduce the number of required AND gates, but may also allow one to use AND gates with fewer inputs. If the truth table is encoded as a Karnaugh map, it is easy to see why this is so. Suppose, for example, that one wishes to construct a decoder for the XS3 code. The invalid conditions 0000, 0001, 0010, 1101, 1110, and 1111 can never occur, so the Karnaugh map will look as follows



	00	01	11	10
00	X	X	D_0	X
01	D1	D2	D4	D3
11	D9	X	X	X
10	D5	D6	D8	D7

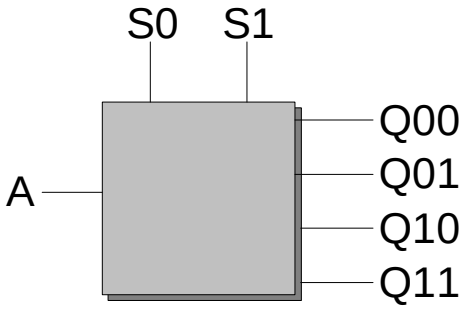
As this Karnaugh map illustrates, D9 can be computed using the following simplified equation.

equation D9=A3&A4

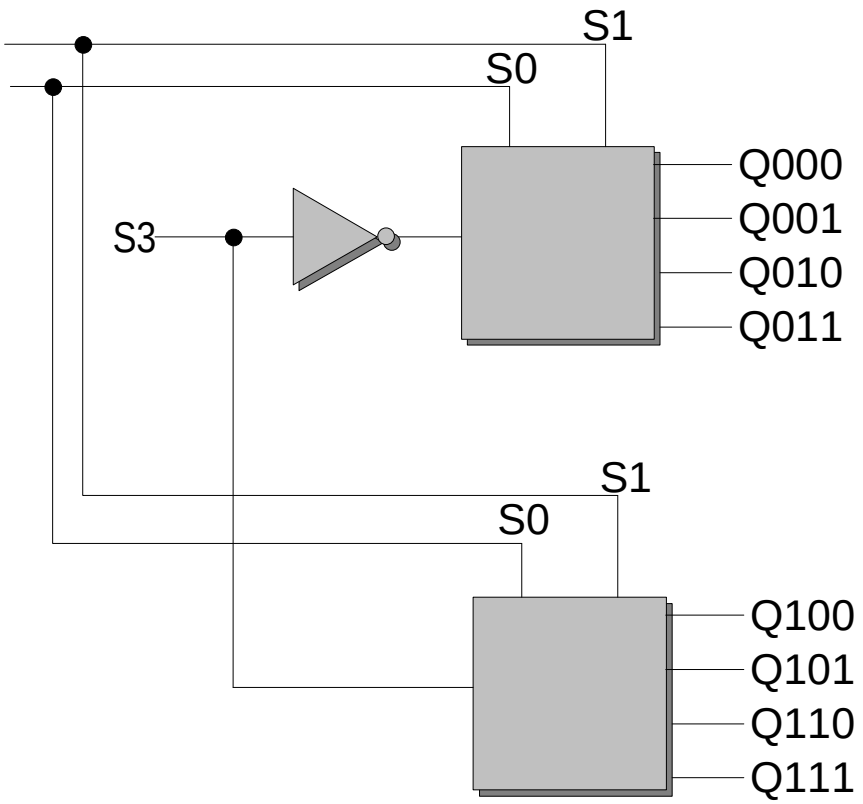
If it is desirable to have all outputs suppressed when a don't care condition occurs, then it is necessary to use 4-input AND gates, but only 10 AND gates will be required.

Create two XS3 decoders, one making full use of the don't care conditions, and one which suppresses all outputs when an invalid input combination is used. Test your circuits using several input combinations.

A Demultiplexer is a circuit that receives a binary input on one input and transmits that input on exactly one of 2^n outputs. There are *n* select inputs that allow one to choose which output will receive the output. Unused outputs will be set to zero. The following is the block diagram of a demultiplexer.



In another sense, this circuit can be viewed as a 2-input decoder with an enabling signal. If A is equal to zero, then all outputs will be equal to zero. If A is equal to 1, then the output selected by S0 and S1 will be equal to 1 and all other outputs will be zero. If A is one, then the circuit acts as a decoder. If A is equal to zero, then all outputs will be zero, regardless of the value of S0 and S1. The circuit above can be used to create a 3-input decoder as illustrated in the following diagram.



Convert your 4-input decoder into a 4-input demultiplexer. Test the circuit using several input combinations. Use the 4-input demultiplexer as a gate to create a 5-input decoder.

Because a decoder has one active line for each minterm, it can be used to compute logic functions. This is especially useful when several functions must be computed using the same set of inputs. Each function requires one OR gate which ors the appropriate minterm outputs together.

Use your 4-input decoder to create a circuit that computes the following three functions.

Input	F1	F2	F3
0000	0	1	0
0001	0	1	0
0010	0	0	0
0011	1	0	0
0100	0	0	1
0101	0	0	0
0110	1	1	0
0111	0	0	0

Input	F1	F2	F3
1000	0	0	0
1001	1	0	0
1010	0	0	0
1011	0	0	1
1100	1	0	0
1101	0	0	0
1110	0	1	0
1111	0	0	0

Results: Turn in print-outs of all circuits and all simulations.

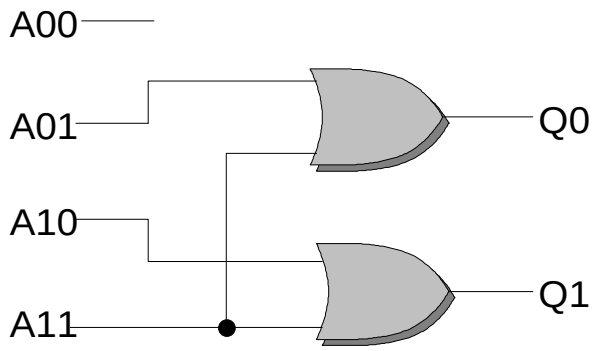
FHDL: No new FHDL features are presented in this experiment.

Experiment 16
Encoders and Multiplexers

Objective: This experiment shows how to construct and use encoders and multiplexers.

Instructions: The encoder performs the reverse function of the decoder. An encoder has 2^n inputs and n outputs. The outputs form an n-bit code identifying which input is set to 1. To function correctly, no more than one input can be set to one at any one time.

As the following diagram shows, an encoder consists of a set of OR gates.



This circuit is coded in FHDL in the following way.

```
Enc1:  circuit
       inputs    A00,A01,A10,A11
       outputs   Q0,Q1

       wire      A00,type=no_connect
       or        (A01,A11),Q0
       or        (A10,A11),Q1
       endcircuit
```

Under normal conditions, unused inputs and outputs are treated as errors. You can suppress this error by declaring the net to be of type “no_connect”.

Construct an 8-input, 3-output encoder, and test it with a number of different input combinations.

Because of the severe restrictions on the input configurations, a simple encoder is not particularly useful. However, another type of encoder, called a priority encoder, is a commonly used circuit. A priority encoder has 2^n inputs and n outputs. All input combinations are permitted. When more than one input is equal to one, the outputs will contain the code of the lowest-numbered input containing a 1. So if bit 0 contains a 1,

then all outputs will be equal to zero, regardless of the value of the other inputs. An additional output can be used to indicate that the input is all zero.

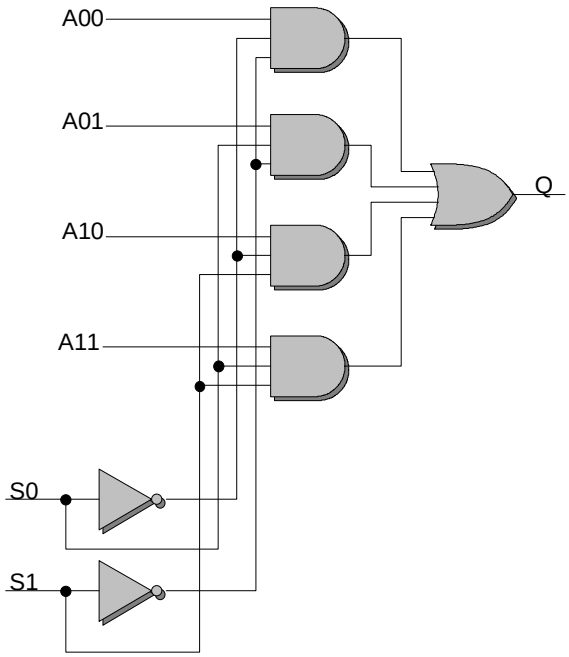
The following is the truth table for an 8-input, 3-output priority encoder,

A000	A001	A010	A011	A100	A101	A110	A111	Q0	Q1	Q2	Z
0	0	0	0	0	0	0	0	X	X	X	1
1	X	X	X	X	X	X	X	0	0	0	0
0	1	X	X	X	X	X	X	1	0	0	0
0	0	1	X	X	X	X	X	0	1	0	0
0	0	0	1	X	X	X	X	1	1	0	0
0	0	0	0	1	X	X	X	0	0	1	0
0	0	0	0	0	1	X	X	1	0	1	0
0	0	0	0	0	0	1	X	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1	0

Construct a priority encoder from this table, and test it using several input combinations.

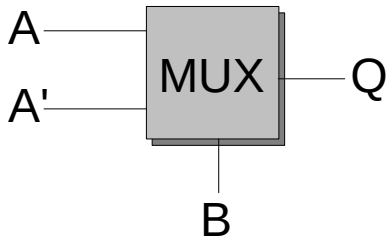
A multiplexer allows several inputs to be selectively switched to a single output. A multiplexer has 2^n inputs, *n* select lines, and 1 output. Each input combination on the *n* select lines corresponds to one of the 2^n inputs. When the combination corresponding to input *k* is placed on the *n* select lines, the value of input *k* is copied to the output.

The following is the logic diagram for a 4-input multiplexer.



Using this diagram as a pattern, create an 8-input multiplexer. There will be a single output and three control lines.

Multiplexers can be used to implement certain logic functions. One common technique for implementing the XOR function is to use a 2-input multiplexer. The complement of at least one of the inputs must be available. Note that a 2-input multiplexer has a single control line. The following logic diagram shows how this circuit is implemented.



Implement an XOR gate as illustrated above, and test it with all input combinations.

Results: Turn in printouts of all circuits, and all simulations.

FHDL: No new FHDL features are introduced in this experiment.

Experiment 17
Hamming Codes

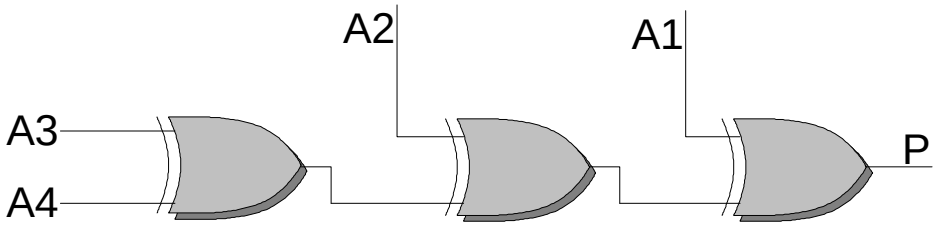
Objective: This experiment shows how to implement a simple error-correcting code.

Instructions: As you probably know, a parity bit is sometimes added to data stored in memory, for the purpose of detecting errors that may occur in the memory. There are two types of parity, known as even parity and odd parity. When even parity is used, an extra bit is added to each word to make the total number of one bits even. For odd parity, the total number of one bits must be odd. The following is a table of 4-bit quantities with even and odd parity bits added.

Original	Even	Odd
0000	00000	00001
0001	00011	00010
0010	00101	00100
0011	00110	00111
0100	01001	01000
0101	01010	01011
0110	01100	01101
0111	01111	01110

Original	Even	Odd
1000	10001	10000
1001	10010	10011
1010	10100	10101
1011	10111	10110
1100	11000	11001
1101	11011	11010
1110	11101	11100
1111	11110	11111

Even parity can be calculated using a single XOR gate, but building XOR gates with more than two inputs is a practical impossibility. (Odd parity is simply the inverse of even parity.) Generally parity is calculated using a chain of 2-input XOR gates, as illustrated below.



If the multi-bit value is stored in a register, the complements of all the inputs will be readily available, making it possible to use the multiplexer circuit of Experiment 16 to implement the XOR functions.

Implement the above circuit, and simulate it using all 16 input combinations to prove that it calculates even parity correctly.

A parity bit can be used to detect single-bit errors, and can also detect any odd number of single-bit errors in the same word. Hamming codes go one step further, by allowing one

to correct single-bit errors. The number of extra bits required to do this is not fixed, but depends on the number of bits in the word. If *E* is the number of extra bits, and *W* is the number of bits in the word, *E* and *W* must have the following relationship.

$$2^E \geq E + W + 1$$

The rationale behind this formula is that the extra bits will have 2^E different combinations. If any bit is incorrect, including both the original data bits and the new extra bits, there must be a unique combination that indicates which bit is wrong. Otherwise the incorrect bit could not be corrected. There must also be an indication that nothing is wrong, so this means there must be at least *E*+*W*+1 different combinations. When adding extra bits, E should be set to the smallest value possible. This is easy to do by trial and error. The following table gives the extra bits required for various word sizes.

W	E	2^E	E+W+1
4	3	8	8
8	4	16	13
12	5	32	18
16	5	32	22
24	5	32	30

W	E	2^E	E+W+1
30	6	64	37
32	6	64	39
64	7	128	72
128	8	256	137
256	9	512	266

The first step in creating a Hamming code is to number the E+W bit positions from right to left in binary, starting with 1. For a 4-bit quantity, the bit positions would be numbered as follows.

111	110	101	100	11	10	1

The extra bits are not clustered at the right, as one might suppose, but distributed through the E+W bit positions as follows.

111	110	101	100	11	10	1

The extra bits will be placed in bit positions whose index is a power of two. If five extra bits are required, then bit positions 1, 10, 100, 1000, and 10000 will contain the extra bits.

The values of the extra bits is computed by doing an even parity computation on selected bit positions of the E+W bits. The first parity computation is done on all positions that have a 1 in the low-order bit of their index. The result of this computation is placed in position 1. The second parity computation is done on all positions that have a 1 in the second-lowest bit of their index. The result of the second computation is placed in bit position 10. This procedure continues until the values of all extra bits have been completed. For the fifth extra bit (if present) the parity computation will select all

positions that have a 1 in the fifth bit position of the index, and the result will be placed in position 10000. The following table gives the details for four data bits.

Bit Positions Selected	Result Location
11, 101, 111	1
11, 110, 111	10
101, 110, 111	100

Create a circuit that will generate the Hamming bits for a 4-bit input. Test this circuit using all 16 input combinations.

Error correction is done by repeating the parity computations that generated the values for the extra bits, and including the extra bit in the computation. If the result is all zero, then there is no error. Otherwise, the parity checks give the binary value of the bit position that is in error. The following table shows the bit positions that must be selected for each parity check when there are 4 data bits.

Bit Positions Selected
1,11, 101, 111
10,11, 110, 111
100,101, 110, 111

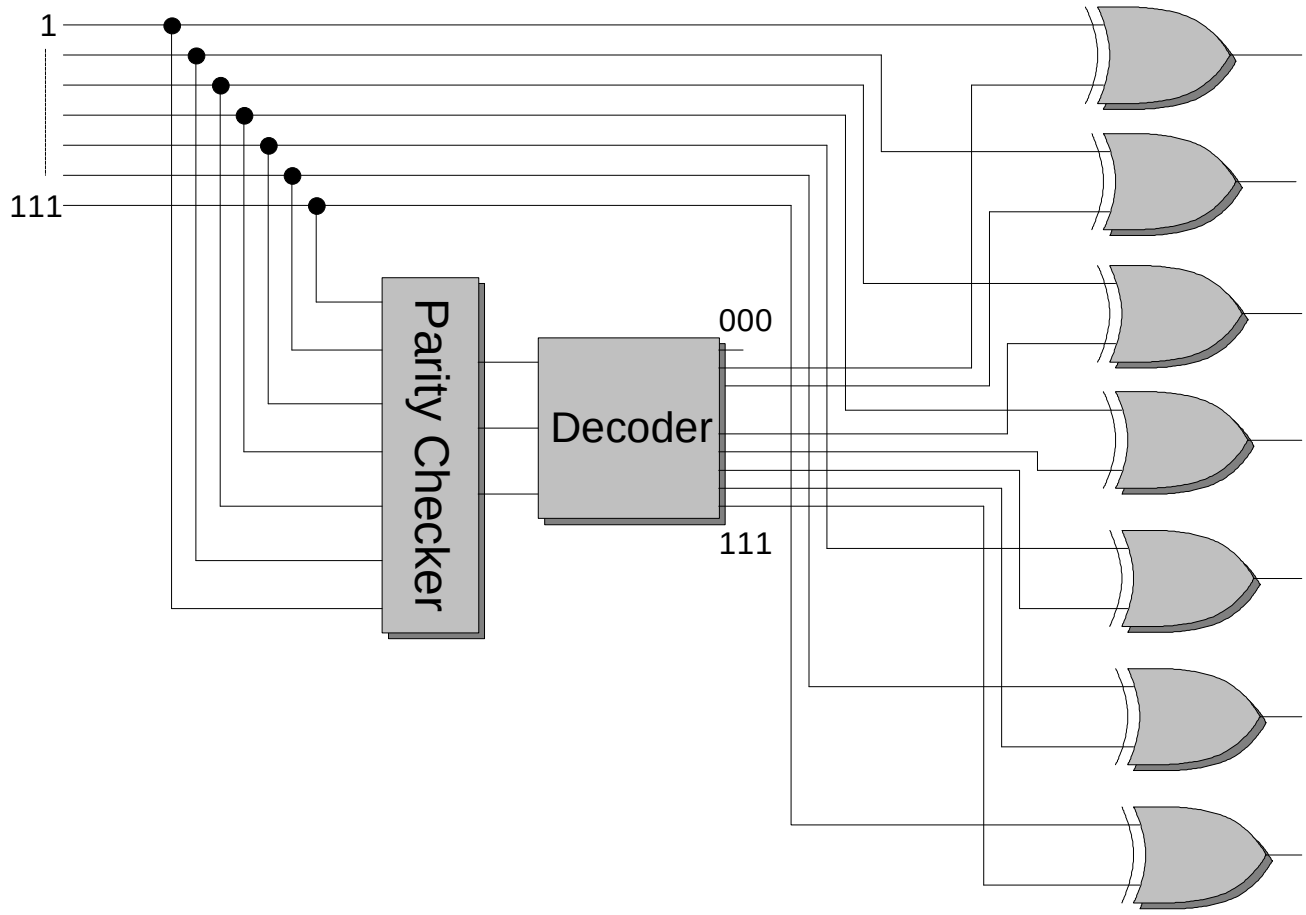
Create a circuit that will perform the parity checks for a 7-bit input, and output the result as a 3-bit binary number. Test this circuit using at least 32 input combinations.

Once the erroneous bit position has been detected, it is possible to correct it using a decoder and a set of XOR gates, as illustrated below. In this diagram, both the input bits and the decoder outputs are numbered in ascending order from top to bottom.

Implement this error-correction circuit, and test it using all 128 input combinations to verify that it does indeed correct single-bit errors. Use the DECODER statement to create the decoder. Note that the 000 output of the decoder is not used. The DECODER statement is written as follows.

dec1: decoder (i2,i1,i0),(q111,q110,q101,q100,q011,q010,q001,q000)

The q000 output must be declared as “no_connect.”



Results: Turn in printouts of all circuits and all simulation runs.

FHDL: This experiment introduces the FHDL decoder statement..

Experiment 18
ROMs and PLAs

Objective: This experiment

Instructions: In its simplest form, a ROM contains *n* address inputs, and a single output. Such a ROM can be constructed from a decoder and an OR gate. The outputs of the decoder correspond to the single-bit words of the ROM. Any word that contains the value 1 has its corresponding decoder output attached to an input of the OR gate. Any word that contains the value 0 has its decoder output declared as "no_connect."

The following table gives the contents of a single-bit ROM. Implement this ROM using a decoder and an OR gate. Use the FHDL decoder statement, and remember to add the required "no_connect" declarations.

Address	Value
000	1
001	0
010	0
011	1
100	1
101	1
110	0
111	1

A Multi-bit ROM can be constructed from a single decoder and a collection of OR gates. There must be one OR gate for each output bit. The inputs of the OR gate are connected to the decoder outputs that should have one bits in that position. In programmable ROMs, it is possible to selectively connect any OR gate to any of the decoder outputs. The method for doing this depends on the type of Programmable ROM.

Implement the following ROM using a decoder and several OR gates. "no_connect" declarations are required only if a word has a value of all zeros.

Address	Value
000	1011
001	0010
010	0001
011	1010
100	0000
101	1110
110	0111

111	1101

A PLA is essentially the same thing as a ROM, but the decoder is deliberately left incomplete. In a ROM, each word has an address consisting of a string of ones and zeros. In a PLA each word (or wordline, to use the proper term) has an address consisting of ones, zeros, and don't cares. This has two consequences. First, a wordline can have more than one address. For example, if a wordline has an address of x000, the two addresses 0000 and 1000 will cause the wordline to be selected. The second consequence is that a single address can select more than one wordline. Given two wordlines with addresses xx11 and 11xx, the address 1111 will select both wordlines. When more than one wordline is selected, the output is either the logical AND or logical OR of the outputs, depending on how the PLA is implemented. In a PLA, the partial decoder is known as the AND-plane. The OR gates and their connections to the AND-plane outputs are known collectively as the OR plane.

Using the principles of partial decoders given in Experiment 15, construct a PLA according to the following table. If more than one wordline is selected, the output should be the logical OR of the selected wordlines.

Address	Value
x100	0110
11x1	1110
1x00	1010
0001	1100
1010	0101
1x0x	0001
xxx1	0011
01xx	1011

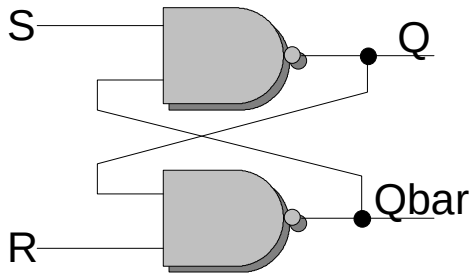
Results: Simulate all circuits using several input combinations. Turn in printouts of all circuits and all simulations.

FHDL: No new FHDL features are introduced in this document.

Experiment 19
Flip-Flops

Objective: This experiment shows the internal implementation of several different types of flip-flops.

Instructions: A basic RS flip-flop is illustrated below.



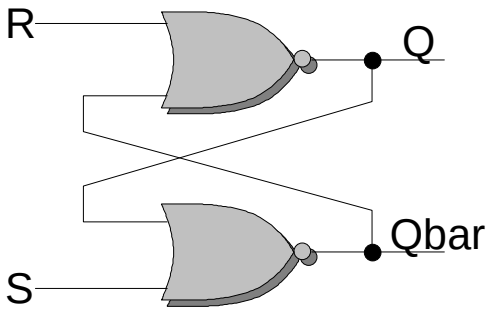
The following table gives the behavior of this flip-flop.

Inputs	Result
S=0, R=0	Q=1, Qbar=1, Potential Instability
S=0, R=1	Q=1, Qbar=0
S=1, R=0	Q=0, Qbar=1
S=1, R=1	Retain Previous State

In this flip-flop, if S and R are both set to 0, and then simultaneously set back to 1, the circuit will go into oscillation. In a real circuit, the final state will depend on the relative delays of the two NAND gates. In WinFHDL, the gate delays are identical, so the circuit will enter a state of permanent oscillation. This oscillation will be detected by the simulator, and reported as an error condition.

Implement this flip-flop in FHDL, and simulate it using various input combinations. You should explicitly test the 00->11 transition.

An RS flip-flop can also be implemented using NOR gates as illustrated below.



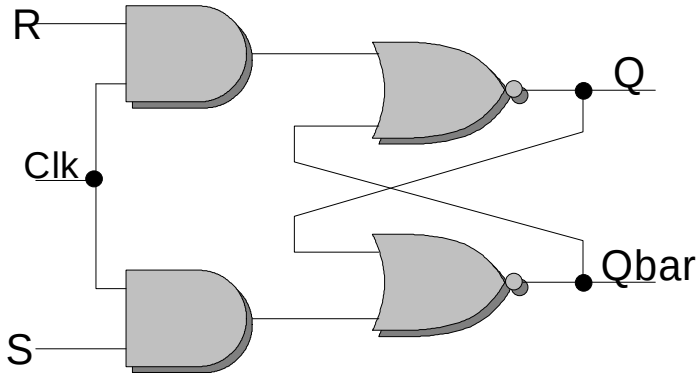
With this implementation of an RS flip-flop, the S and R values are inverted, as illustrated in the following table.

Inputs	Result
S=1, R=1	Q=1, Qbar=1, Potential Instability
S=1, R=0	Q=1, Qbar=0
S=0, R=1	Q=0, Qbar=1
S=0, R=0	Retain Previous State

The transition 11->00 will cause this flip-flop to oscillate.

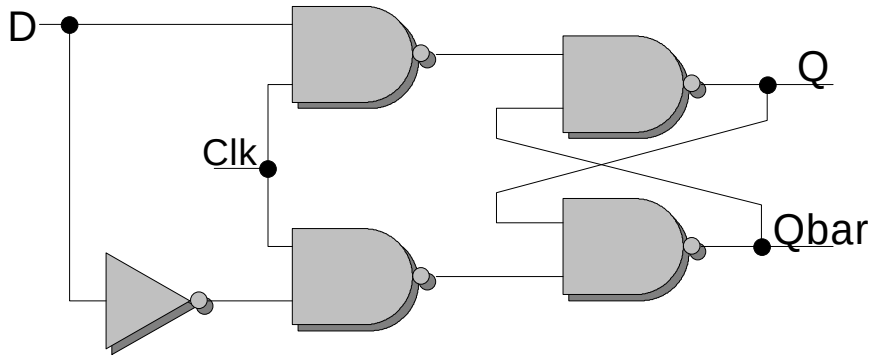
Implement an RS flip-flop using NOR gates and test it using several input combinations. Explicitly test the 11->00 transition.

It is possible to synchronize transitions of this flip-flop with a clock signal by using the following logic.



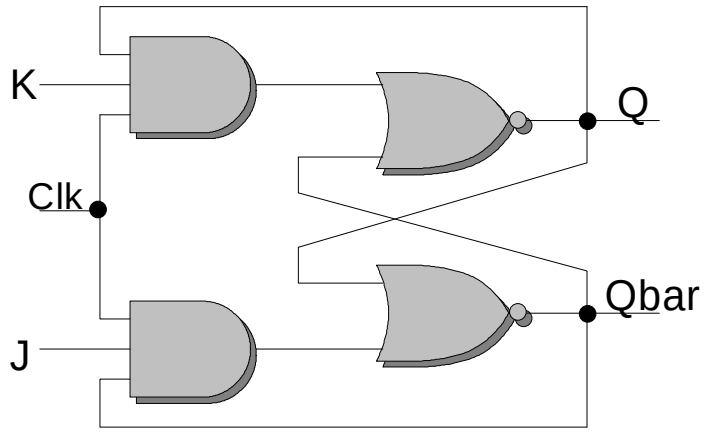
Implement this circuit and test it with several input combinations.

Another commonly used flip-flop is the D flip-flop. This flip-flop copies its input signal, D, to its output Q when the clock is active. Although this flip-flop eliminates the instability of the 00->11 transition, nearly simultaneous changes in the D and Clk inputs can cause instability in the circuit. It is impossible to simulate this instability in WinFHDL. The logic of the D flip-flop is as follows.

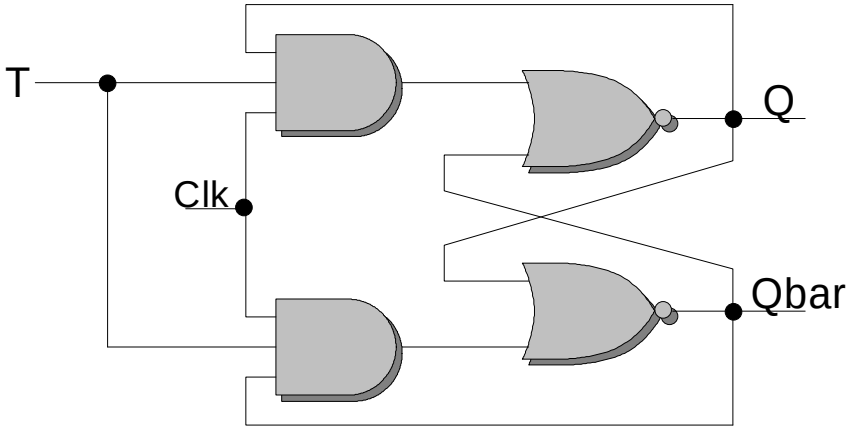


Implement the D flip-flop and test it with several input combinations.

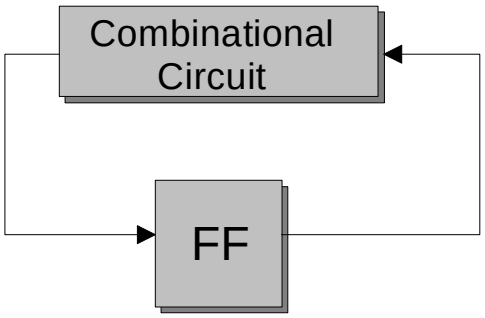
The JK flip-flop is an improvement over the RS flip-flop. The J and K inputs of the JK flip-flop act like the S and R inputs of the NOR-Gate RS flip-flop. However, when J=1, and K=1, the output of the flip-flop inverts. The 11->00 transition will not cause an oscillation to occur. The logic diagram for the JK flip-flop is illustrated below. Implement this flip-flop in FHDL, and test it using several input combinations.



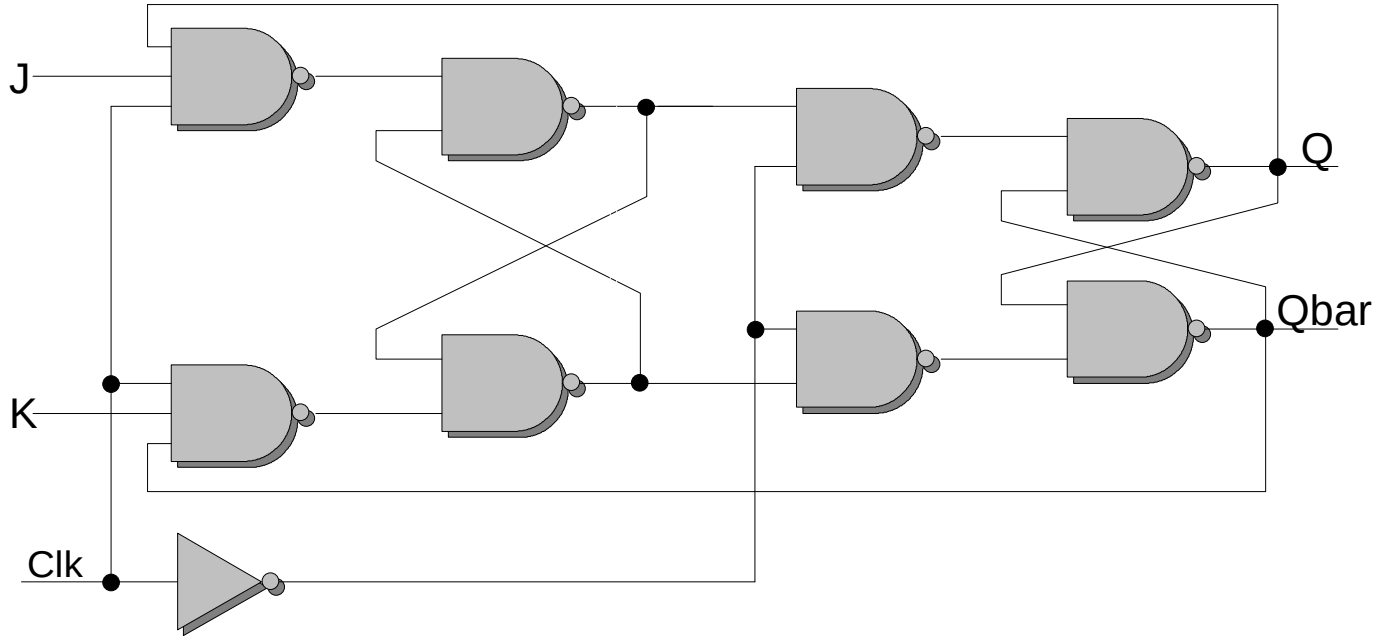
The T flip-flop is a variation of the JK flip-flop that makes use of the output-inversion property. When both the clock and the T input of the T flip-flop are equal to 1, the output will toggle between 1 and 0. The following is the logic diagram for the T flip-flop. Implement this flip-flop in FHDL and test it using various input combinations.



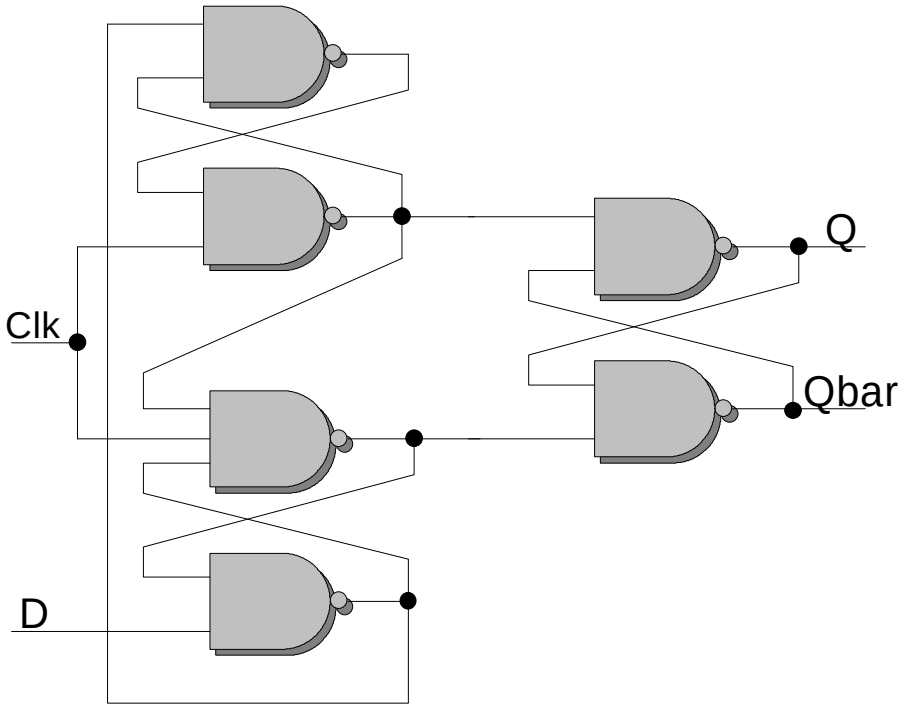
Flip-flops are the basis for various types of sequential circuits. In many of these circuits, the flip-flops are used to store an internal state. A new state is computed from the circuit inputs and from the previous state. This new state is then stored in the flip-flops. The structure of the circuit is illustrated in the following diagram.



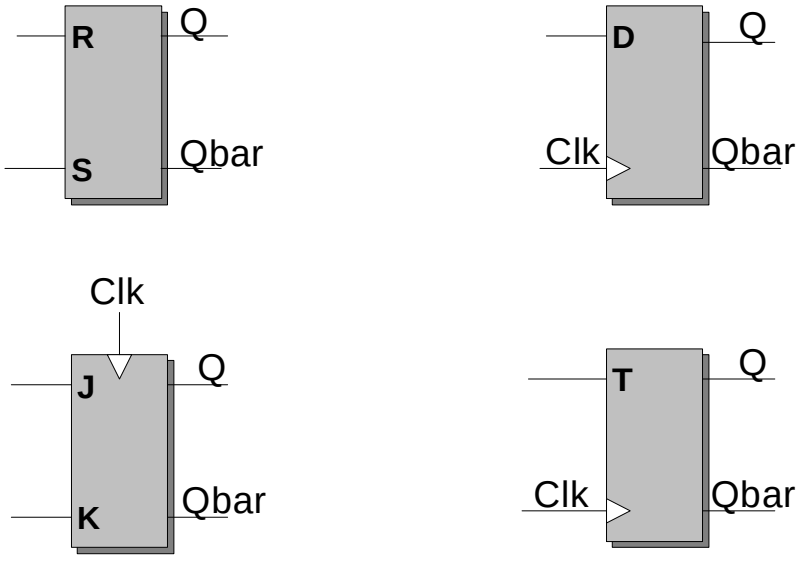
The main problem with circuits such as that illustrated above, is that the combinational circuit may be able to compute several states during one clock period. If more than one flip-flop is used, the state may become unpredictable if the various parts of the combinational circuit do not run at the same speed. The master slave flip-flop is used to control state-transitions in a sequential circuit. There are many varieties of master-slave flip-flops, but all are constructed in more or less the same manner. Two flip-flops are connected in series. The first flip-flop is controlled by the true clock, while the second is controlled by the inverted clock. In the diagram below, when the clock is set to 1, the J and K inputs are processed, and if they cause any changes, these changes are stored on the internal connections between the two flip-flops. When the clock is set to zero, the internal signals are copied to the Q and Qbar outputs.



Edge-triggering is another method of controlling flip-flop transitions. The transitions of an edge-triggered flip-flops are synchronized with *changes* in the clock, rather than being associated with the value of the clock. The 0->1 transition is called the positive-edge of the clock, while the 1->0 transition is called the negative-edge. The following is a positive edge-triggered D flip-flop. Implement this flip-flop in FHDL and simulate it using several input combinations.



Flip-flops are generally specified as a single logic symbol. Various logic symbols are illustrated below.



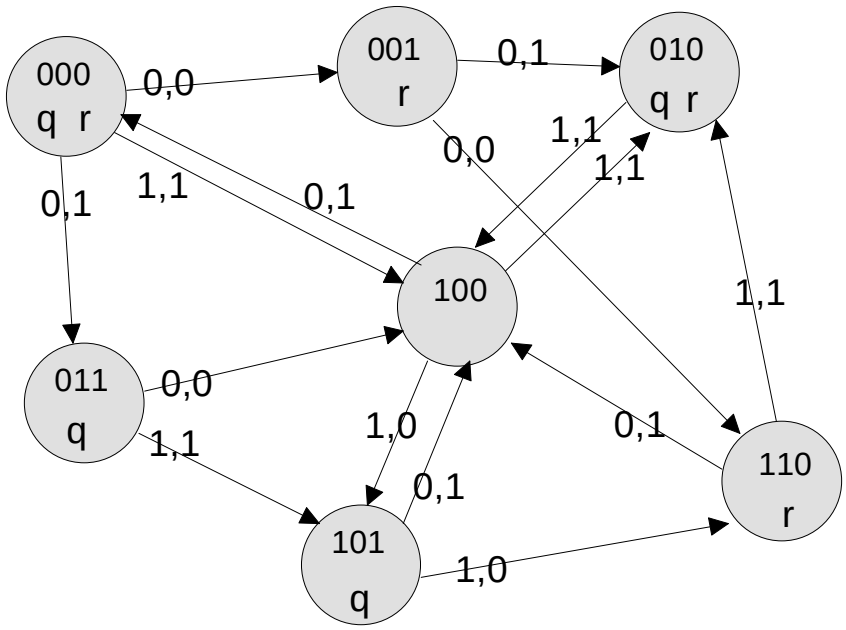
Results: Turn in print-outs of all circuits as well as print-outs of all simulation runs.

FHDL: No new FHDL is introduced in this experiment.

Experiment 20
Sequential Circuits

Objective: This experiment shows how to construct elementary sequential circuits.

Instructions: In most cases, a sequential circuit can be described in terms of a state machine. One classic example is that of a traffic light, which changes from green to yellow to red based on signals from a timer. The process of constructing and optimizing state machines is beyond the scope of this experiment. However, when one is given a state machine, the process of implementing it in hardware is comparatively simple. Suppose we are given the state machine illustrated below, which has two input signals, two output signals, and seven states.



The state diagram of this state machine has been simplified in several ways. For each state, there must be four possible transitions. Any transition that causes the state machine to stay in the same state has been omitted. The names of the inputs have been omitted. The inputs are assumed to be named A and B. The pairs 0,0; 0,1; 1,0; and 1,1 give the values for A and B respectively. Finally, output values have been omitted. The outputs are named q and r. If an output appears in a state, this implies that the output has the value 1 in that state. If the output does not appear in a state, this implies that the output has the value 0 in that state. All of these simplifications must be taken into account when constructing the state transition table for the state machine. This table is given below.

State	q	r	A	B	New
000	1	1	0	0	001
			0	1	011
			1	0	000
			1	1	100
001	0	1	0	0	110
			0	1	010
			1	0	001
			1	1	001
010	1	1	0	0	010
			0	1	010
			1	0	010
			1	1	100
011	1	0	0	0	100
			0	1	011

State	q	r	A	B	New
011	1	0	1	0	011
			1	1	101
100	0	0	0	0	100
			0	1	000
			1	0	101
			1	1	010
101	1	0	0	0	101
			0	1	100
			1	0	110
			1	1	101
110	0	1	0	0	110
			0	1	100
			1	0	110
			1	1	010

This state machine will require 3 flip-flops to maintain the state information. To control state changes, the flip-flops must be clocked, and should be either edge-triggered or master-slave flip-flops. Many textbooks recommend the use of JK flip-flops, because edge-triggered JK flip-flops are readily available in IC packages. However, we will flout convention by using master-slave D flip-flops instead. Assume that the circuit has three inputs, A, B, and Clk. The clock will be distributed through the following gates.

```
buff    Clk,HClk
not     Clk,LClk
```

The use of these two gates to distribute the clock illustrates an important principle. Namely, that the delays in the clock signal should be the same for all parts of the circuit.

The D flip-flops will be declared directly, using the *dff* gate type, as illustrated below.

```
dff     (NS1,HClk),IS1
dff     (IS1,LClk),S1
```

Although the circuit has only two outputs, Q and R, the state outputs S1 S2, and S3 should also be declared as primary outputs to simplify debugging. These primary outputs can be removed once the circuit is functioning correctly.

To complete the implementation of this state machine, it is necessary to derive the logic equations for NS1, NS2, NS3, Q, and R. The equations for Q and R will be derived from the variables S1, S2, and S3, while the equations for NS1, NS2,and NS3 will be derived from S1, S2, S3, A, and B. In both cases, the combination S1=1,S2=1,S3=1 is a don't care condition.

The equation for Q is given below.

$$Q = (S1' \ \& \ S3') \mid (S2' \ \& \ S3') \mid (S2 \ \& \ S3)$$

Derive the equation for R.

The equation for NS1 is given below.

NS1=(S1&B')|(S3&A'&B')|(S1&S3)|(S1&S2&A')|(S1'&S3'&A'&B)|(S2&S3&A&B)

Derive the equations for NS2 and NS3. Note that these equations are intended to be complex. This tends to reflect situations that arise in real-world designs.

Once the equations for Q, R, NS1, NS2, and NS3 have been derived, they can either be specified directly in FHDL or can be manually translated into logic. In either case, these equations complete the implementation of the state machine. Implement these equations using one of these techniques, and simulate the state machine using several input combinations.

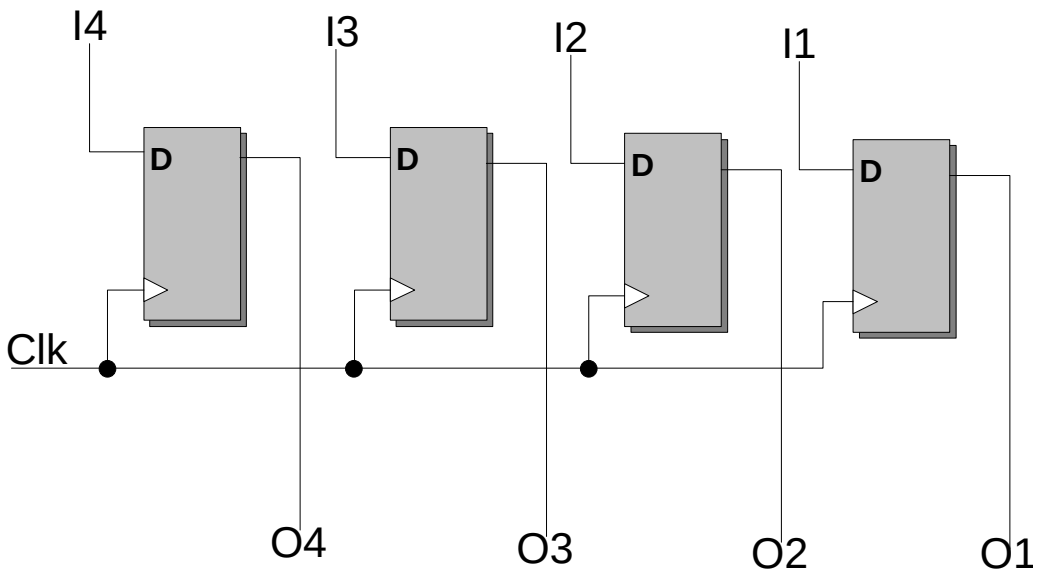
Results: Turn in your hand-written equations, printouts of your circuits, and printouts of all your simulations.

FHDL: The dff gate type is introduced in this experiment.

Experiment 21
Registers

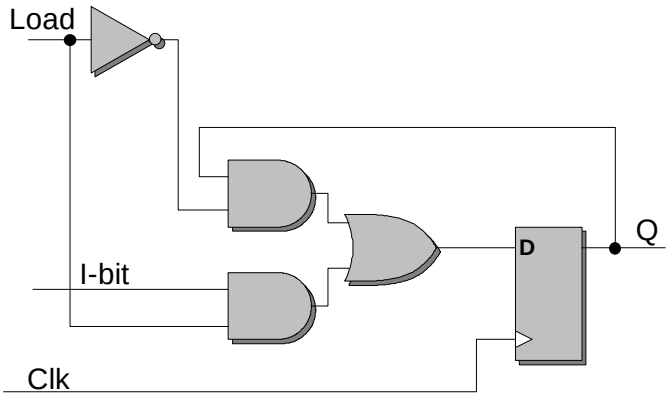
Objective: This experiment will show you how to construct various types of registers.

Instructions: In its simplest form, a register is simply a collection of flip-flops, all driven by the same clock. Construct a four bit register as illustrated below, and test it with several input combinations.



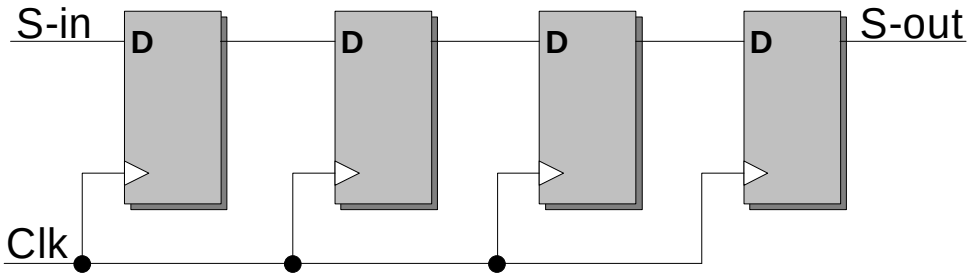
It is important to note that this register is not edge-triggered. If I4 (for example) changes state while the clock is active, O4 will also change state. For correct functioning, it is necessary to guarantee that the inputs are stable while the clock is active.

In many instances, the clock signal can be used as a load signal for the register. If it is desired to have a synchronous load signal in addition to the clock, the following circuit can be used for each bit of the register.



Construct the circuit pictured above, and simulate it using several input combinations. Then construct a 4-bit register, using the circuit as a gate. Test the 4-bit register using several input combinations.

A shift register is used for serial transmission of data. The following is an example of a shift register that shifts its contents to the right on each clock pulse. As with ordinary registers, the clock signal can also be used as a shift control.



Construct the shift-register pictured above, and test it using several input combinations.

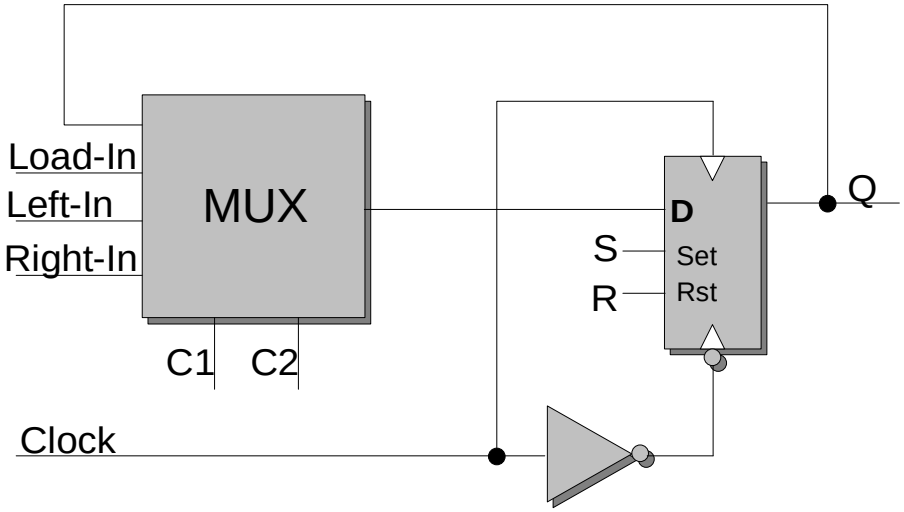
Designing registers with special features is a common problem in modern circuit design. It is typical to design a register with a number of different features. As a final problem we will construct a register that permits parallel loads, left-shifts, right-shifts, asynchronous clear and asynchronous set. The basic flip-flop is illustrated below.

This is a CMOS D-flip flop with asynchronous set and clear. CMOS flip flops require two clock inputs, both the true clock and the inverted clock. In FHDL this flip flop is specified as follows.

G1: dff4 (D,Clock,InvClock,Set,Reset),(Q,Qbar)

The Qbar output may be omitted. When this is done, it is also possible to omit the final set of parenthesis. The **Set** input is active-high, while the **Reset** input is active-low. To inactivate both inputs, **Set** must be zero and **Reset** must be one.

The Set and Clear functions can be implemented using the native flip-flop inputs. The other functions will be implemented using a MUX on the flip-flop D input. The MUX will have four inputs, which are designated as follows: LeftIn, RightIn, LoadIn, and Stable. The resulting circuit looks as follows.



Create the circuit illustrated above, with the illustrated inputs and outputs. Test this circuit using several input combinations. Use this circuit as a gate to create a 4-bit register. Note that the Left-In and Right-In inputs must be wired to the outputs of the neighboring flip-flops. The Left-In of the leftmost flip-flop is the left serial input of the register, and the Right-In of the rightmost flip-flop is the right serial input of the register. The register should have three one-bit control inputs: **Load**, **Left-Shift**, and **Right-Shift**. Use a priority encoder to encode these inputs into the required C1 and C2 control signals. Load should be first priority, but the order of the other two is up to you. In FHDL the priority encoder is specified as follows.

G1: encoder (In00,In01,In10,In11),(C1,C2)

Test your register using several input combinations. Make sure to illustrate correct functioning of all operations.

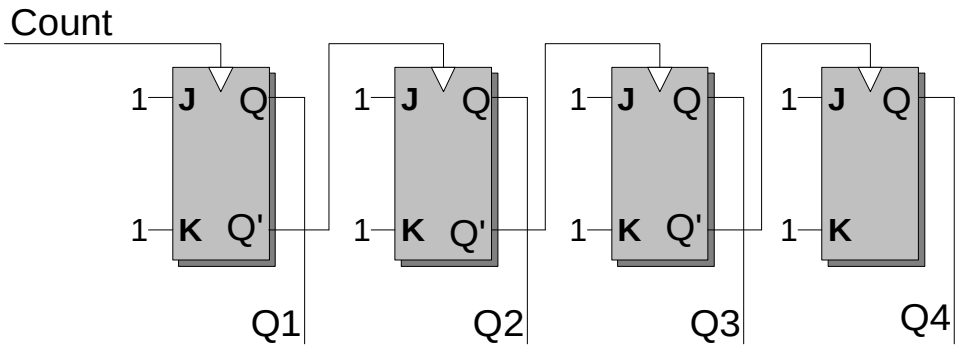
Results: Turn in printouts of all circuits and all simulation runs.

FHDL: This experiment introduces the CMOS flip-flop and the priority encoder.

Experiment 22
Counters

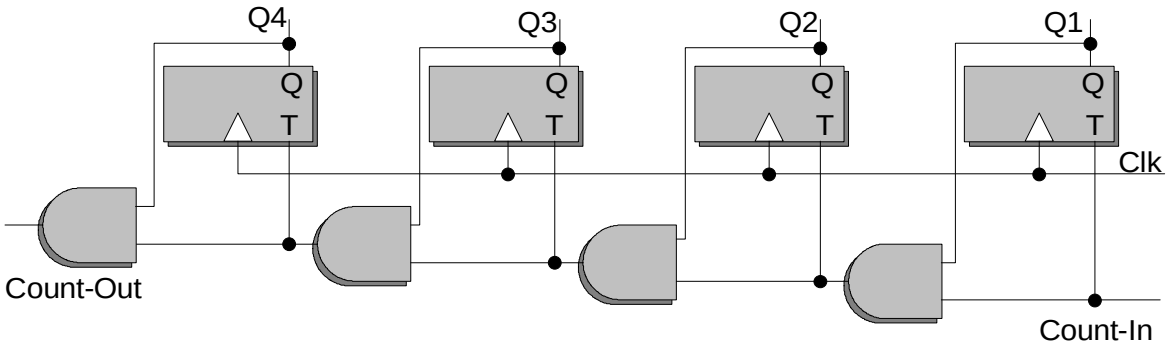
Objective: This experiment shows you how to construct synchronous and asynchronous counters.

Instructions: A counter is a register that contains a binary value, which is incremented or decremented in response to a control signal. The control signal is often called the clock, whether or not the clock is actually used for this purpose.



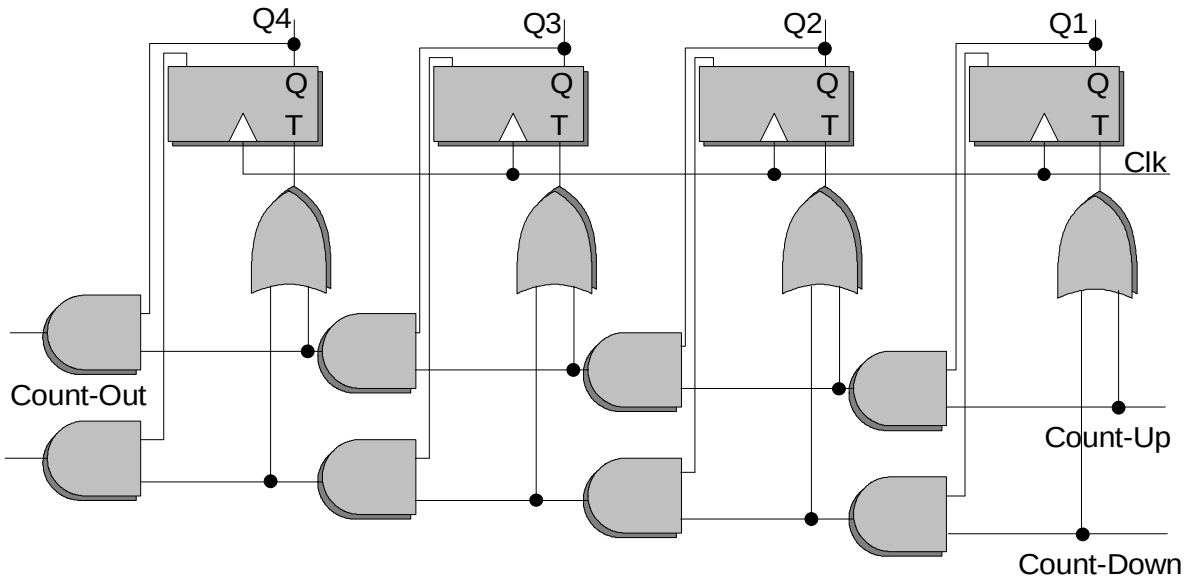
Implement the above circuit in FHDL, and test it using several input combinations. Use the "Show Intermediate" command to demonstrate the asynchronous behavior of the circuit.

The main drawback of the asynchronous counter is that not all outputs change simultaneously. This implies that an instantaneous view of the outputs Q1-Q4 may have intermediate values, which appear to be incorrect. The logic attached to these outputs must take this behavior into account. The following is a synchronous counter, which guarantees that the changes to the outputs will be synchronized with the clock. The count-in signal must be stable when the clock is active. The count-out output can be used to construct a larger counter from a number of four-bit stages.



Implement this circuit in FHDL and test it using several different input combinations.

The following circuit is an extension of the previous circuit, which allows counting in both the up and down directions.



Implement this circuit in FHDL, and test it using various input combinations.

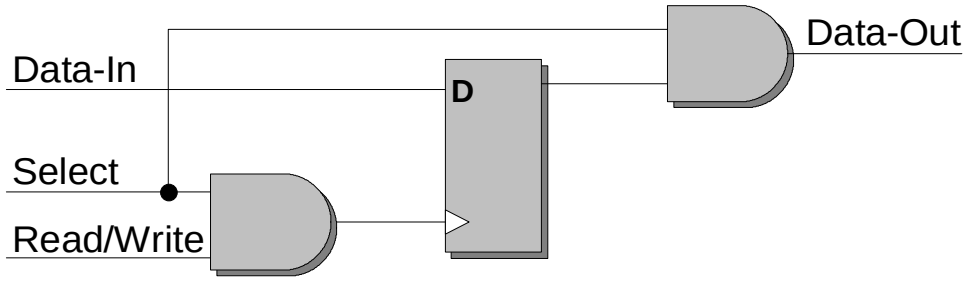
Results: Turn in printouts of all circuits, and printouts of all simulations.

FHDL: No new FHDL features are introduced in this circuit.

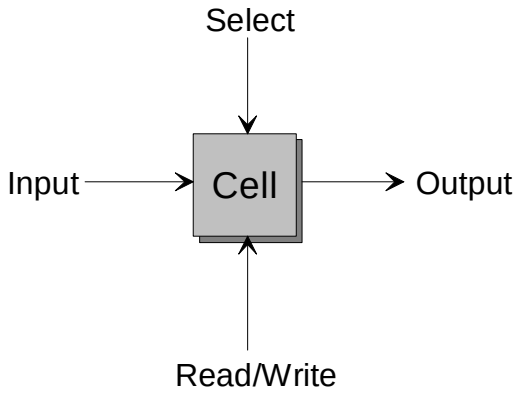
Experiment 23
Random Access Memories

Objective: This experiment shows how to simulate random access memories in FHDL.

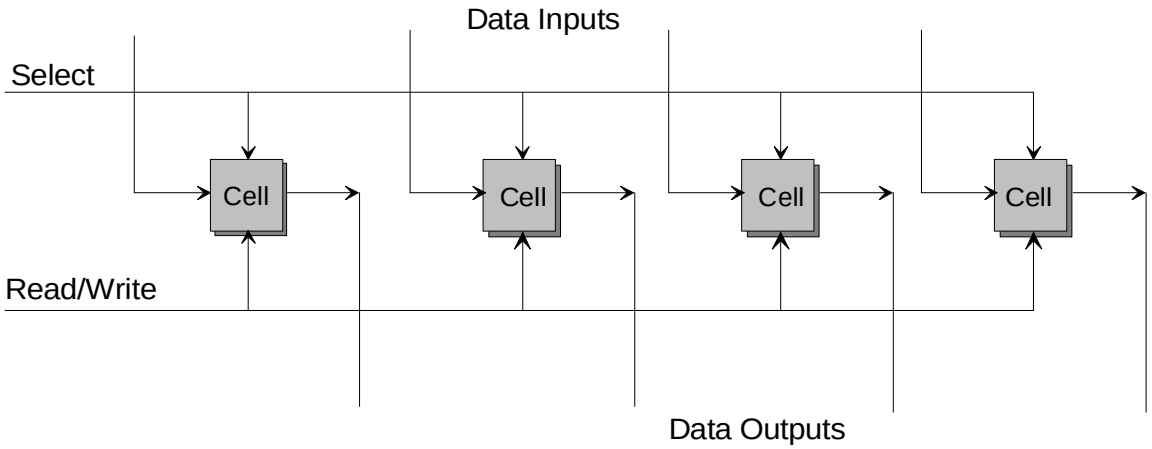
Instructions: The following diagram represents a single memory cell. Implement this memory cell, and test it using several input patterns.



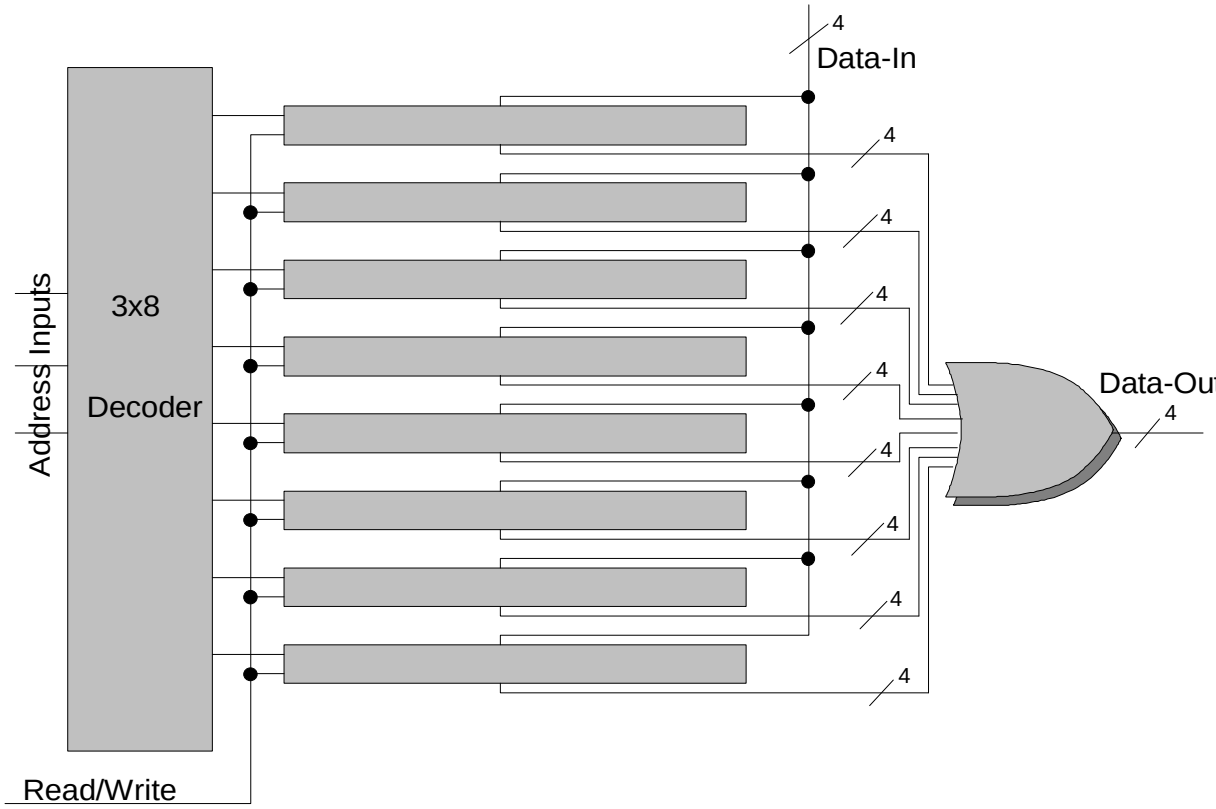
This cell can be represented by the following symbol.



This cell can be used as a gate to construct a memory word as illustrated in the following diagram.



Implement this four-bit word in FHDL and test it using several input combinations. This 4-bit word can be used to construct an 8-word memory as illustrated in the following diagram.

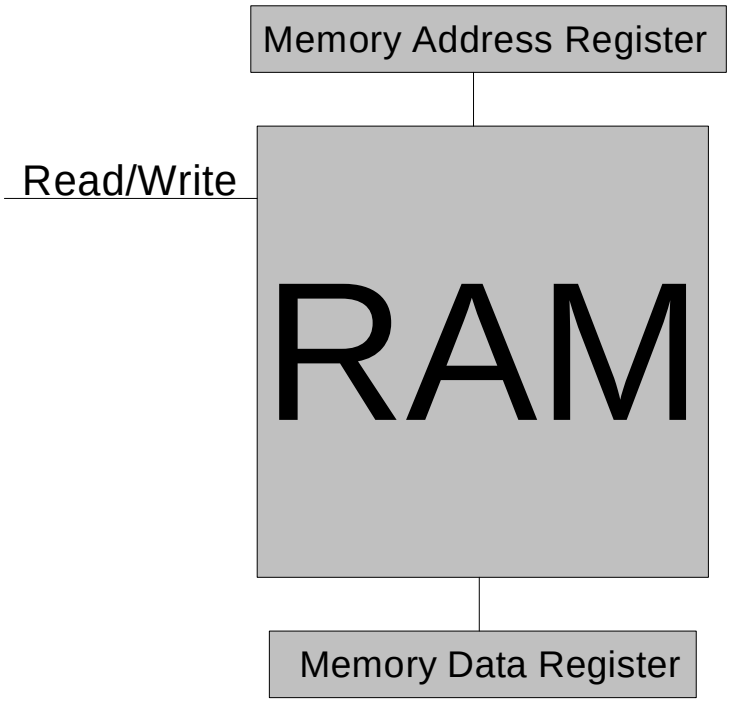


Implement this memory in FHDL, and test it using several input combinations.

When using RAM memories in a large circuit, the FHDL RAM specification can be used to implement the ram. The ram statement is coded as follows.

```
MyRam:      ram     (AddressIn,DataIn,ReadWrite),DataOut
```

The signals AddressIn, DataIn, and DataOut must be multi-bit busses of width greater than one. The size of the RAM depends on the width of the address. DataIn and DataOut must have the same width. Use the RAM statement and the REGISTER statement to create the following circuit. Test this circuit using several input combinations.



Results: Turn in printouts of all circuits and all simulations.

FHDL: This experiment introduces the RAM statement.

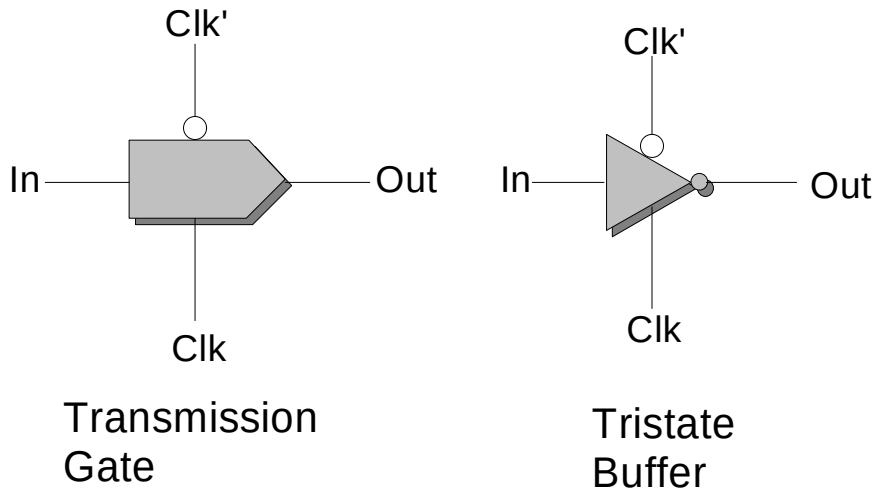
Experiment 24
Tristate Logic

Objective: This experiment

Instructions: The outputs of most gates are binary-valued. That is, they can be set to either zero or one. Physically, a gate output is set to one by connecting it to the circuit's power source, while it is set to zero by connecting it to ground. Tristate gates have a third state in which the gate output is disconnected from both power and ground. (This is sometimes called a floating output.) When in the third state, which is called being tri-stated, the output of the gate will retain its previous value for a short period of time, and will drift toward zero. Tristate gates can be used to create wired-or connections, and to construct multiplexors.

Tristate gates come in two varieties, transmission gates, which have no amplifying power, and tristate buffers, which are essentially NOT gates with tristate control inputs.

The symbols for transmission gates and tristate buffers are illustrated below. These gates are those for CMOS technology. Clk and Clk' must be the logical inverse of one another. These inputs are often referred to as "clock inputs" but are seldom connected to a real clock.

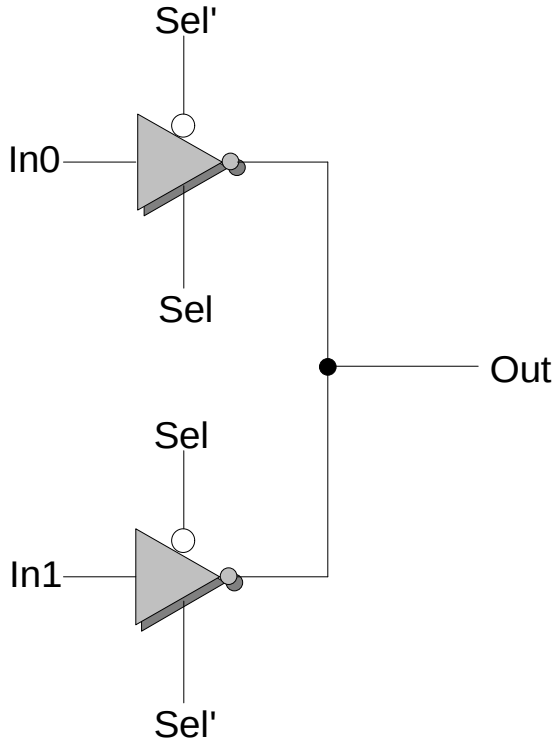


In FHDL, the transmission gate and tristate buffer are specified using the following statements.

```
Tg1:    tgate    (In,Clk,ClkBar),Out

Tb1:    tbufi    (In,Clk,ClkBar),Out
```

As suggested above, the tristate buffer can be used to create a multiplexor. The following illustrates a 2x1 multiplexor constructed with two transmission gates.



Implement this circuit in FHDL and test it using several input combinations. Note that when a circuit of this nature is used, it is not permissible for more than one gate to be un-tristated. If two or more gates are un-tristated, a low-resistance path between power and ground can be created, destroying the circuit.

Results: Turn in printouts of all circuits and all simulation runs.

FHDL: This experiment introduces tristate buffers and transmission gates.