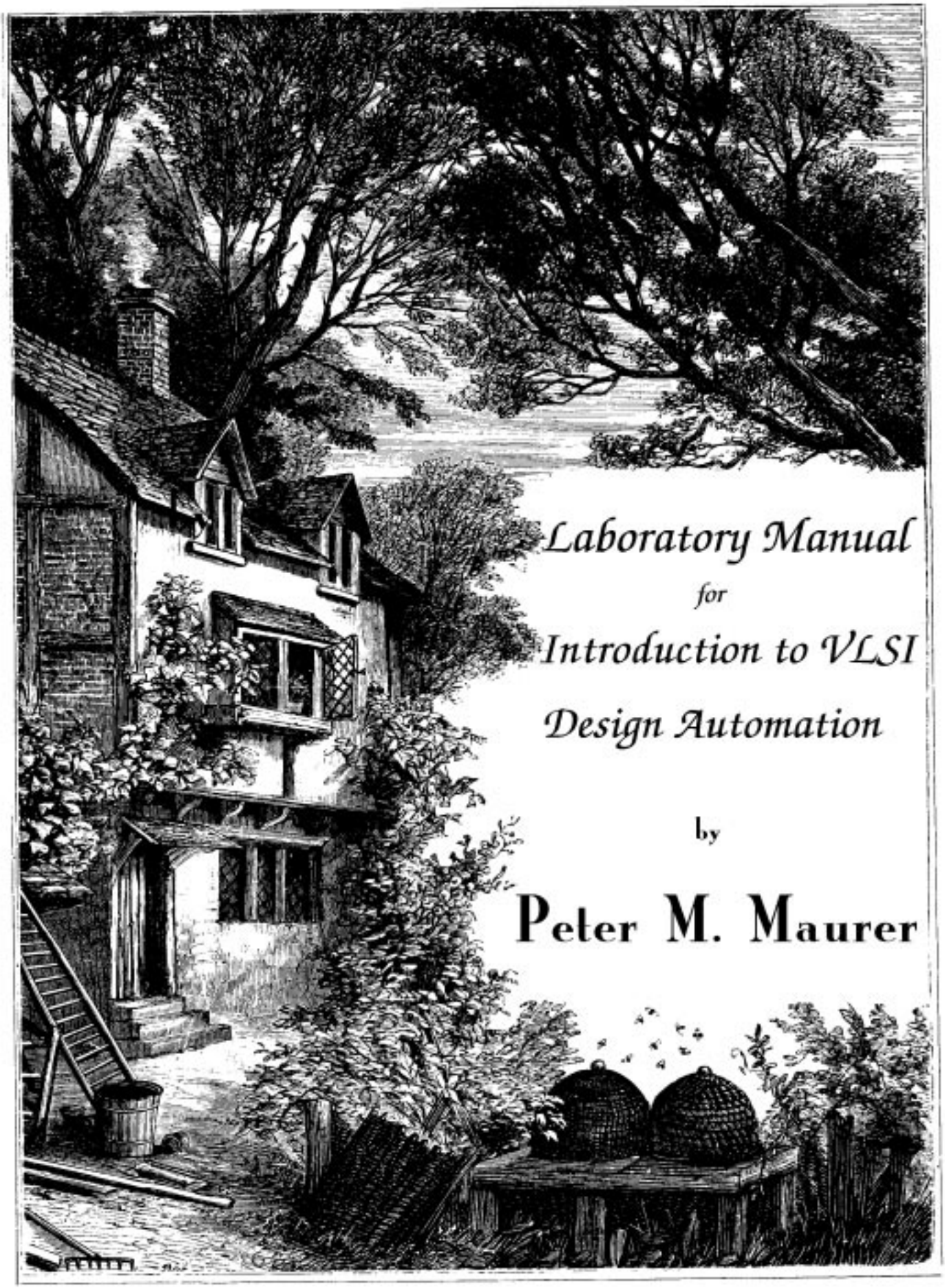


Laboratory Manual

for

Introduction to VLSI Design Automation

by

Peter M. Maurer

Experiment 1
C Programming

Objective: The FHDL system consists of several independent components that are combined using Visual Basic. The components are written in C and are compiled into Windows Dynamic Link Libraries (DLLs). The purpose of this experiment is to familiarize you with the C language, and teach you the principles of creating a DLL.

Background: A Dynamic Link Library is a collection of independently compiled functions that can be called by other programs. A DLL can contain two types of functions, Exported Functions, and Internal Functions. Exported Functions must be the target of a **FAR** call, and must use **PASCAL** calling conventions. Exported functions must also have the **_export** keyword. Small DLLs, such as that created in this experiment, usually are coded in a single **.c** file. This **.c** file should begin with the following code.

```
#include <windows.h>

/* The following two functions are required, but do nothing */

/* the following statement is used only with Borland C */
#pragma argsused
int FAR PASCAL _export WEP(int exittype)
{
    return 1;
}

/* the following statement is used only with Borland C */
#pragma argsused
int FAR PASCAL LibMain (HANDLE hInstance, WORD wDataSeg,
                        WORD wHeapSize, LPSTR lpszCmdLine)
{
    return 1 ;
}
```

Instructions: Create an MS Windows dynamic link library that will perform the following functions.

1. **Convert a string of hexadecimal digits into a long integer.**
2. **Convert a long integer into a string of eight hexadecimal digits.**
3. **Given an integer n, compute the sum of integers from 1 to n. Return an integer.**
4. **Compute the square of an integer, and return an integer.**

The first function must be named *HexToLong*, the second must be named *LongToHex*, the third must be named *SumToN*, and the fourth must be named *ISquare*. The name of your library must be *firstone.dll*.

The function declarations must look as follows.

```
long FAR PASCAL _export HexToLong(char *Hex)
void FAR PASCAL _export LongToHex(long Value,char *Hex)
int FAR PASCAL _export SumToN(int N)
int FAR PASCAL _export ISquare(int N)
```

Every Windows DLL must have a **.def** file which is used during the link phase of the compilation. The .def file for this project should look as follows.

```
LIBRARY FIRSTONE

DESCRIPTION 'My First DLL'

EXETYPE WINDOWS

CODE PRELOAD MOVEABLE DISCARDABLE
DATA PRELOAD MOVEABLE SINGLE

HEAPSIZE 4096
```

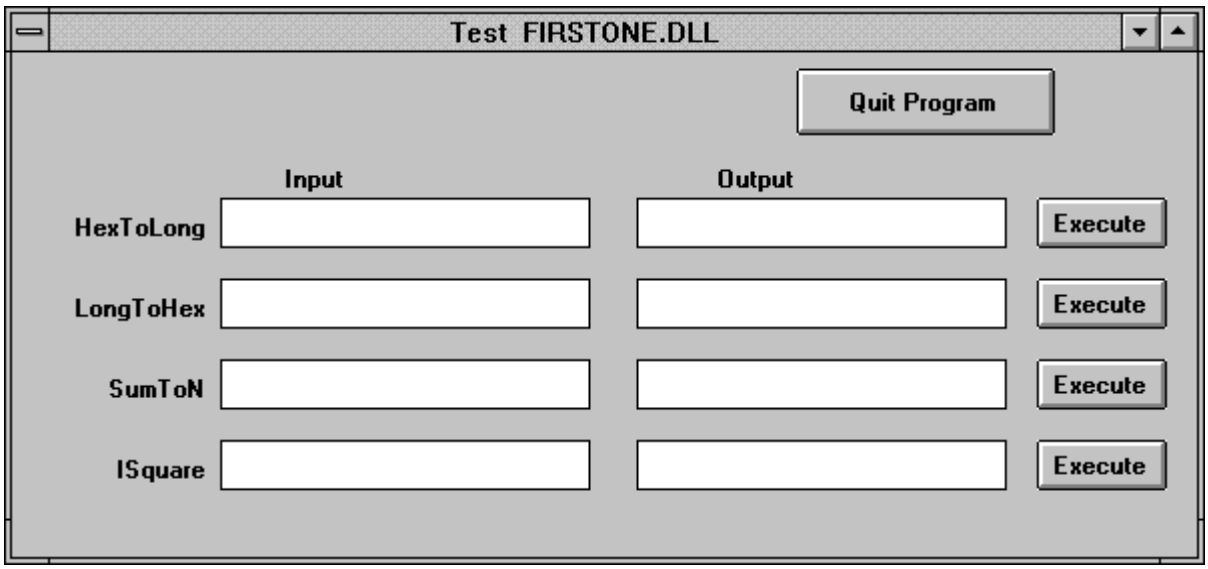
The algorithms for conversion from and to hexadecimal are not necessarily well known. The following algorithms can be used to perform the conversions, *but each algorithm contains one bug*. If you use these algorithms instead of coding your own, you must fix the bugs. *You may not use any standard C functions in your code*.

```
long FAR PASCAL _export HexToLong(char *Hex)
{
        unsigned long rv;
        char *ts;
        for (ts = Hex, rv = 0L; *ts != '\0' ; ts++)
        {
                rv <<= 4;
                if (ts <= '9' && ts >= '0')
                {
                        rv |= *ts - '0';
                }
                else if (*ts <= 'f' && *ts >= 'a')
                {
                        rv |= *ts - 'a' + 10;
                }
                else if (*ts <= 'F' && *ts >= 'A')
                {
                        rv |= *ts - 'A' + 10;
                }
        }
        return (long)rv;
}
```

```
void FAR PASCAL _export LongToHex(long Value,char *Hex)
{
          unsigned long tv;
          char tc;
          int i;
          tv = (unsigned long)Value;
          Hex[8] = '\0';
          for (i=0 ; i<8; i++)
          {
                    tc = (char)(tv & 0xf);
                    if (tc > 9)
                    {
                              tc += 'A';
                    }
                    else
                    {
                              tc += '0';
                    }
                    Hex[7-i] = tc;
                    tv >>= 4;
          }
}
```

Results: You should test your DLL and turn in a printout of the code. Since a DLL (usually) has no code for interfacing with the Windows environment, it is necessary to have an independent program to test your DLL. Fortunately, this has already been done for you. The Windows program "Test FirstOne.dll" is available from your instructor. Place your

DLL either in the same directory as the test program, and run the test program in the usual way. This program will display the following window.

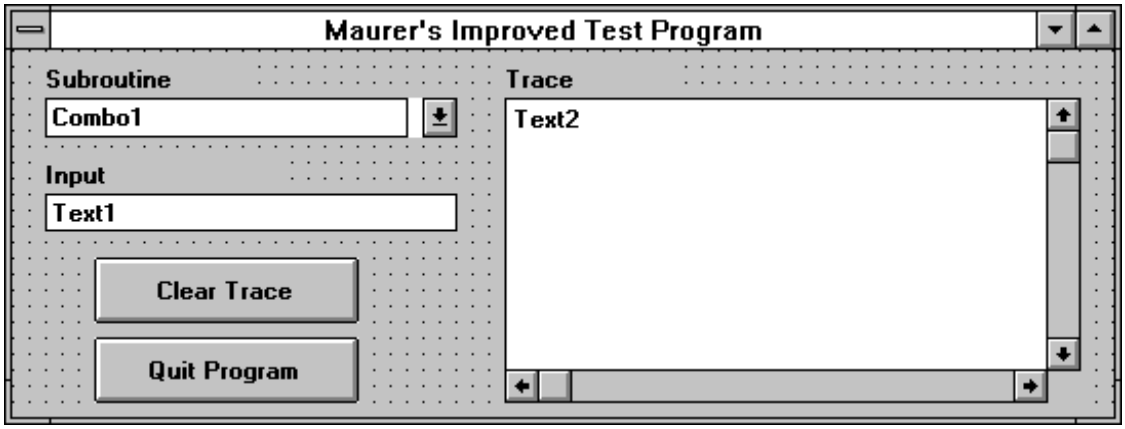


For each of your functions, enter the input value on the left, and click the execute button for the row. The output value will appear on the right. Click Quit Program before trying to recompile your code. The following is the icon for this program.

Experiment 2
Visual Basic Programming

Objective: The objective of this experiment is to familiarize you with programming in Visual Basic. The principles governing visual programming are roughly the same, regardless of which language is used.

Instructions: Create a simplified replacement for the test program used in the previous experiment. The screen should look as follows. (This is a design-time view.)



First start Visual Basic, and create a new project. Set the **Background** color of the form to gray and place the following controls on the form. One combo box, two text boxes, two command buttons, and three labels. Arrange them as shown above. For the second text box, set the **MultiLine** Property to **True**, and set the **ScrollBar** property to **Both**. Change the **Caption** property of the form to something that includes your name. Change the **Caption** properties of the three labels to **Subroutine**, **Input**, and **Trace**. Change the **Caption** properties of the buttons to **Clear Trace** and **Quit Program**. Adjust the size of the form and the controls to suit your taste.

Double Click on the **Quit Program** button to obtain the code-window for the button. The default window is for the "click" event, which is executed when the button is clicked with the mouse. This button should terminate the program, which is done by executing the **End** Statement. Close the window and double-click on the **Clear Trace** button. This button must clear the **Text2** edit control, which is done by executing the following statement.

```
Text2.Text = ""
```

This statement shows how to access control properties at run-time. The **Text** property is the same as that which appears in the property window at design time.

It is possible to test your program after each change. Run the program, test for errors, and go on to the next step.

Now it is necessary to create the initialization code for your program. Two things are required for initialization. The combo box **Combo1** must be initialized with the names of the four functions, to allow the user to select the name of the function to be tested. It is also necessary to clear both the Text1 and Text2 controls. (This could be done at design time, but it is easier to remember the name of the control if we do the initialization at run time.) Initialization code is placed in the **form Load** event. The window for this event can be opened by double clicking on the background of the form.

The **AddNew** method is used to add items to a combo box. The only difference between a method and a function is that the name of the method follows the control name instead of preceding it. The following statement adds the first function name to the combo box. Note the period between **Combo1** and **AddNew**.

```
Combo1.AddNew "HexToLong"
```

Add this and the other three **AddNew** statements to the **form Load** event. Now that the combo box has four strings, it is necessary to tell the box which string to display. This is done by setting the **ListIndex** property of the combo box. This property has no default value, so if you don't set it, the box will display garbage. The first string has **ListIndex 0**, the second has **ListIndex 1**, and so forth. Add the following statement after the **AddNew** statements.

```
Combo1.ListIndex = 0
```

Add the two statements to clear **Text1** and **Text2**.

We will use the following convention for testing the DLL functions. To test a function, the user must select the function using the combo box, and then must type the input value into **Text1**. After the input value has been typed, the user must press **Enter**. When the enter key is pressed, the name of the function, the input value and the output value will be displayed at the end of **Text2**.

Typing is handled automatically by the default text-box behavior. To intercept the **Enter** key, it is necessary to add code to the **KeyPress** event of **Text1**. To do this, double-click on **Text1**. The default event is not **KeyPress**, so you must go to the combo box at the upper right of the code window, and select **KeyPress** before typing any code. This event has one parameter, **KeyAscii**. The enter key produces a **KeyAscii** code of **13**, so the following code will intercept the enter key.

```
If KeyAscii = 13 Then
    DoFunctions
EndIf
```

This code can be improved slightly. Every time the user presses the enter key in the Text1 box, the system will beep. To turn off the beep, add the following statement *between the **If** and **EndIf** statements*.

```
KeyAscii = 0
```

*Don't put this outside the **If** statement*. If you do, **Text1** will cease to function.

Before you can call the DLL functions, you must declare them to Visual Basic. To do this, double-click on the background of the form. Then, using the left-hand combo box at the top of the code window, select the (**declarations**) section. This will be the first item on the list. You will need to use the scroll-bar to find it. If this section does not already begin with the following line, add it immediately. This will make debugging much easier.

```
Option Explicit
```

Type the declarations following the **Option** line. The following is the declaration of the **HexToLong** function.

```
Declare Function HexToLong Lib "FirstOne.dll" (ByVal InString As String) As Long
```

Each declaration must begin with the keyword **Declare**. Next comes either **Sub** or **Function**, depending on whether the function returns a value. (In C, a **Sub** must be declared as **void**.) The name of the function follows the **Function** or **Sub** keyword. This name must match the name exactly as it appears in the C code for the DLL. The **Lib** keyword identifies the name of the DLL file, which must be enclosed in quotes. Next comes the list of arguments enclosed in parentheses. Each argument declaration begins with the keyword **ByVal**. (This keyword may be omitted for some DLLs, but it is required for all the arguments in this exeperiment.) The name of the argument follows **ByVal**. Since the name of the argument is never used, any name may be specified. Following the name is the keyword **As**, followed by a Visual Basic type, in this case, **String**. Following the list of arguments is the declaration of the type of value returned by the function. As in all type specifications, the keyword **As** is used, followed by the Visual Basic type, **Long**.

The **LongToHex** function is declared as follows.

```
Declare Sub LongToHex Lib "FirstOne.dll" (ByVal InLong As Long, ByVal OutStr As String)
```

This declaration is similar to the last. Since the **LibToHex** function does not return a value it is declared as a **Sub**. It also has two argument declarations separated by a comma. The two remaining declarations are similar.

```
Declare Function SumToN Lib "FirstOne.dll" (ByVal InVal As Integer) As Integer
Declare Function ISquare Lib "FirstOne.dll" (ByVal InVal As Integer) As Integer
```

Above you were instructed to place the following line into the code for **Text1**.

```
DoFunctions
```

This is actually a function call to a subroutine called **DoFunctions**. The code for this subroutine should be typed immediately after the last **Declare** statement. Start this code by typing the following line and pressing **Enter**.

```
Sub  DoFunctions
```

Visual Basic will jump to a new window, and add some extra stuff to this declaration, including an **End Sub** statement. Type the code between the **Sub** and **End Sub** statements. When this function is created, it becomes part of the (**declarations**) section of the form. To locate the code for this or any other independent function you've written, double-click on the form background, and use the left combo box at the top of the code window to go to the (**declarations**) section. Then, use the combo box at the right to locate the function you wish to edit. Initially this box contains the text (**general**) to indicate that the declarations in this section are global for all functions and subroutines in the form.

The first step in creating **DoFunctions** is to determine which function is selected in the combo box. The following If statement will do this.

```
If Combo1.ListIndex = 0 Then
‘ Code for LongToHex
ElseIf Combo1.ListIndex = 1 Then
‘ Code for HexToLong
ElseIf Combo1.ListIndex = 2 Then
‘Code for SumToN
ElseIf Combo1.ListIndex = 3 Then
‘ Code for ISquare
EndIf
```

The code for **LongToHex** looks as follows.

```
TempStr = Text1.Text
TempStr = TempStr & Chr(0)
TempLong = HexToLong(TempStr)
DoOutput "HexToLong", Text1.Text & "", Str$(TempLong)
```

The first statement copies the contents of **Text1** into a temporary string, so that a null character can be tacked on the end. When **HexToLong** is called, the address of the first

byte of the string will be passed to the function, but no zero will automatically be added to the end. The second line adds the null character to the end of the string, as required by the C language. The third statement calls the function and assigns the result to a long variable. The fourth statement is a call to a subroutine called **DoOutput**. **DoOutput** handles the display of data in **Text2**, and is declared as a **Sub**. When a **Sub** is called, the arguments must be listed without parentheses. This function requires three arguments, the first is the name of the function being tested, the second is the input value, and the third is the output value. Since all three arguments must be strings, the **Str$** function is used to convert **TempLong** from a **Long** to a **String**. The second argument is specified as **Text1.Text&""** because Text1.Text is not considered to be a valid string argument in function calls, but can be used as a string in expressions. **Text1.Text&""** concatenates **Text1.Text** with the null string and creates a new temporary string to be used as the argument.

The code for **LongToHex** looks as follows.

```
TempStr = Text1.Text
TempLong = Val(TempStr)
TempStr = "123456789"
LongToHex TempLong, TempStr
DoOutput "LongToHex", Text1.Text & "", Left$(TempStr, 8)
```

The first line extracts the contents of Text1 into a string. The second line converts the string to a long integer. The third statement guarantees that TempStr is long enough to hold the output data. LongToHex requires TempStr to be at least nine bytes long. The first 8 bytes contain the data, while the ninth contains a null character. Note that the call to LongToHex specifies arguments without parentheses. Note also that the third argument for DoOutput extracts only the leftmost eight characters of TempStr, eliminating the terminating null character. Attempting to output a string containing a null character will normally cause output to be truncated at the location of the null.

The code for the other two functions is virtually identical, and looks as follows.

```
TempIn = Val(Text1.Text)
TempOut = SumToN(TempIn)
DoOutput "SumToN", Str$(TempIn), Str$(TempOut)
```

These code segments require four local variables to be declared in the DoFunctions procedure. To do this add the following four lines immediately following the **Sub DoFunctions ( )** line.

```
Dim TempStr As String
Dim TempLong As Long
Dim TempIn As Integer
Dim TempOut As Integer
```

Finally, it is necessary to create the **DoOutput** subroutine. To do this, type the following line after the **End Sub** statement for the **DoFunctions** procedure, and press **Enter**.

```
Sub DoOutput
```

This will immediately change to the following.

```
Sub DoOutput ( )
```

Go back up to the altered line and put the following declarations in between the parentheses.

```
Fname As String, InVal As String, OutVal As String
```

The remainder of the code for **DoOutput** is fairly simple.

```
Dim TmpS As String

TmpS = Fname
TmpS = TmpS & ":  Input=" & InVal
TmpS = TmpS & ",  Output=" & OutVal
TmpS = TmpS & Chr(13) & Chr(10)
Text2.SelStart = Len(Text2.Text)
Text2.SelLength = 0
Text2.SelText = TmpS
```

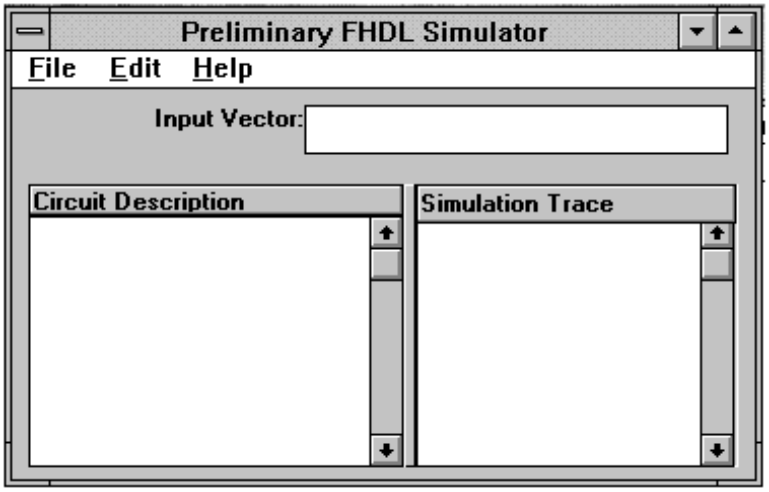
The first four lines create a line of text. The **&** operator performs string concatenation. The fourth line adds the termination characters (**Return** and **NewLine**) to the line. They must both be specified, and they must be specified in the give order. The final three lines concatenate the newly created line to whatever is currently being displayed in the **Text2** window. Setting the **SelStart** property as indicated puts the text cursor past the last character currently displayed in **Text2**, while setting the **SelLength** property to zero indicates that there is no current selection. Assigning the newly created line to the **SelText** property causes the null string at the text cursor position to be replaced with the newly created line.

Results: Demonstrate your running program. Save the form (**.frm**) file as text and turn in a printed copy.

Experiment 3
Practice With FHDL

Objective: The objective of this experiment is to familiarize you with the FHDL system, with the eventual objective of learning how such tools are created.

Instructions: The following is a snapshot of the FHDL student simulator.



The window labeled "**Circuit Description**" is used to type an FHDL circuit description, while the window labeled "Simulation Trace" is used to display the progress of the simulation. Begin by entering the following circuit, which is a simple two-input **and** gate.

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

To simulate the circuit, go to the small upper window labeled "**Input Vector**," and type the following.

```
0,0
```

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

```
I>0,0
->0
```

Any line beginning with **I>** contains input typed by you. Any line beginning with **->** contains output produced by the simulator. Generate a complete truth table for the **and** gate by entering the following vectors.

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

Print your results using the **Print Both** command, which is found in the **File** menu. Now add the following seven gates to the circuit between the and gate and the endcircuit statement. Simulate this new circuit using all four input combinations. For this, and all other circuits in this experiment, print the results using the **Print Both** command.

FHDL Syntax: As the sample circuit illustrates, 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**.

Instructions: Create the following circuit, which is a half adder, and simulate it using all four input combinations.

```
HalfAdder:  circuit
            inputs      a,b
            outputs     carry,sum
gsum:       xor         (a,b),sum
gcarry:     and         (a,b),carry
            endcircuit
```

Now use the half-adder to create a full adder. To do this, add the following circuit definition, in front of the half-adder. To repeat, the **FullAdder** must come first, followed by the **HalfAdder**.

```
FullAdder:  circuit

            inputs      carryin,a,b
            outputs     carryout,sum

gis1:       HalfAdder   (a,b),(ic1,is)
gsum:       HalfAdder   (carryin,is),(ic2,sum)
gcarryout:  or          (ic1,ic2),carryout

            endcircuit
```

Test this circuit using all eight input combinations. Now use the full adder and half adder to create a four-bit ripple-carry adder. To do this add the following circuit ahead of the **FullAdder** definition.

```
FourBit:    circuit

            inputs      ci,a4,a3,a2,a1,b4,b3,b2,b1
            outputs     co,s4,s3,s2,s1

gs1:        FullAdder   (ci,a1,b1),(c1,s1)
gs2:        FullAdder   (c1,a2,b2),(c2,s2)
gs3:        FullAdder   (c2,a3,b3),(c3,s3)
gs4:        FullAdder   (c3,a4,b4),(co,s4)

            endcircuit
```

Test this circuit with a number of different input combinations, to prove that it works.

Results: Turn in the printouts for all circuits.

FHDL: This last set of circuits illustrates the procedure for creating hierarchical circuits. You should consult the FHDL help file for information on how to declare constant one and zero signals, bus declarations, and additional gate types.

Experiment 4
FHDL Internals I

Objective: The objective of this experiment is to familiarize you with the data structures and components used by the FHDL simulator.

Background: The VBFHDL.VBX component is used to parse FHDL. Components of this nature are known as custom controls. Custom controls are contained in .VBX files, which are .DLL files with certain standard functions and a .VBX suffix. The .VBX files are generally stored in the Windows System directory. To use the control (or controls) contained in a VBX file, you first must add the file to your project using the Add File command in the Visual Basic File menu. Some VBX files may be added to your project by default. (The project file AUTOLOAD.MAK, located in the VB main directory, contains the list of VBX files that are loaded by default into each project.)

The VBFHDL.VBX contains one control of type FHDL. This control has the following properties.

VBFHDL.VBX Properties			
Name	***Run***	***Desg.***	***Description***
Action	W		Assigning a value to this property causes various different functions to be performed. See below for the various action codes and their meanings.
Enabled	*	*	VB Standard
ErrorString	R		This string contains a list of error messages. Each message is on a separate line. *Currently, this property has not been fully implemented.*
FileName	W		In addition to parsing FHDL files, VBFHDL.VBX contains fast Open and Save routines. This is the file name that is used for these functions.
Height	*	*	VB Standard
HWnd	R		VB Standard
Index	R	*	VB Standard
InputString	*		This string contains the circuit to be parsed, or the result of an Open operation, and the source for a Save operation.
Left	*	*	VB Standard
Name	R	*	VB Standard
OutputHandle	R		This contains the result of a parsing operation. Although it is formally a long integer, it is actually the address of a data structure containing the parsed description of the circuit. This address cannot be used by visual basic. It must be passed to other vbx's or dll's.

VBFHDL.VBX Properties			
Name	***Run***	***Desg.***	***Description***
Parent	R	*	VB Standard
Result	R		This will contain False if the circuit did not parse correctly, and True if the circuit was OK.
Tag	*	*	VB Standard
Top	*	*	VB Standard
Width	*	*	VB Standard

The following are the action codes that can be used to perform functions using VBFHDL.VBX. To cause any one of the following actions to occur, assign the numeric code to the **Action** property.

VBFHDL.VBX Action Codes		
Value	***Action***	***Description***
1	Parse FHDL Program	Parse the circuit description contained in the property **InputString**, and set the values of **OutputHandle**, **Result**, and **ErrorString**
2	Reserved	
3	Release Error String	Destroys the string contained in property **ErrorString**, and releases the storage assigned to it.
4	Open File	Opens the file whose name is in the property **FileName** reads the file into a Visual Basic string, and places the result into **InputString**. This action can read either DOS or UNIX files.
5	Release Circuit Storage	Destroys the parsed circuit description pointed to by **OutputHandle**, and frees the storage containing it.
6	Save File	Writes the contents of the property **InputString** to the file named by property **FileName**.

The following is the data structure returned by VBFHDL.VBX.

```
typedef struct cktdef
{
      STORPOOL   Pool;
      NETWORK   *CircuitHead;
      NETWORK   *MemoryHead;
      NETWORK   *ASMHead;
      IDHOLD   *SubnetList;
      int   result;
      DICTE   **Dictionary;
}
CKTDEF;
```

The only element of this structure which is currently of interest to us are elements, **CircuitHead** and **result**. If the element **result** contains the value 0, then the remainder of the datastructure is invalid, and should not be used. If the element **result** is non-zero then the rest of the structure can be assumed to be valid. The **CircuitHead** element points to a parsed circuit definition. **CircuitHead** contains a pointer to a structure of the following type.

```
typedef struct network
{
    struct  network  *next;
    DICTE  *id;
    short  type;
    long  gate_count;
    long  net_count;
    short  input_count;
    short  output_count;
    short  io_count;
    short  attr_count;
    struct  attrhold  *attr;
    long  fault_count;
    struct  fault  *faults;
    long  user;
    long  ref;
    char  *gname;
    NET  **netlist;
    GATE  **gatelist;
    NET  **inputs;
    NET  **outputs;
    NET  **ioputs;
}
NETWORK;
```

For our purposes, the important elements of this structure are **id**, **gate_count**, **net_count**, **gatelist**, and **netlist**. The **id** element is used to find the name of a circuit. This element points to a structure of the following type.

```
typedef struct dicte
{
      struct dicte *next;
      unsigned char *name;
      short type;
      unsigned char hashp;
      ELEM elem;
}
DICTE;
```

For ths experiment, the only element of the **DICTE** structure that is important is the element **name**. If the variable **NetHead** contains a pointer to a **NETWORK** structure, then the name of the circuit can be accessed using the expression **NetHead->id->name**.

Returning to the **NETWORK** structure, the element **net_count** gives the number of nets in the circuit, while **gate_count** gives the number of gates. The elements **netlist** and **gatelist** are used as arrays to access the data structures for the nets and gates of the circuit. For example, **netlist[2]** is a pointer to the third net of the circuit, while **gatelist[5]** is a pointer to the sixth gate. The first gate is accessed using the expression **gatelist[0]**.

The netlist and gatelist pointers point to elements of the following types.

```
typedef struct net
{
    struct net *next;
    DICTE *id;
    short type;
    unsigned short width;
    long index;
    long scc_index;
    unsigned short input_count;
    unsigned short output_count;
    long attr_count;
    struct attrhold *attr;
    long value;
    long user;
    char *gname;
    struct gate *gates[1];
}
NET;
```

```
typedef struct gate
{
    struct gate *next;
    DICTE *id;
    char *ctype;
    short type;
    long type2;
    long index;
    long scc_index;
    unsigned short input_count;
    unsigned short output_count;
    long attr_count;
    struct attrhold *attr;
    struct fault *faults;
    long user;
    char *gname;
    struct net *nets[1];
}
GATE;
```

For the purposes of this experiment, we will be concerned only with the **id** elements of these structures.

Instructions: Create a Windows DLL that implements the functions outlined in the following table.

Function	*Purpose*
RegisterCircuit	Gives your DLL the address of a circuit
CircuitName	Returns the name of the registered circuit
FirstGate	Returns the name of the first gate in the circuit
NextGate	When called repeatedly, returns all gate names in the circuit.
FirstNet	Returns the name of the first net in the circuit
NextNet	When called repeatedly, returns all net names in the circuit.

The program should contain three global variables, one for the address of registered circuits, one for the current position of **NextGate**, and one for the current position of **NextNet**. (Global variables are variables defined outside of any function.)

To illustrate, the following is an example of how RegisterCircuit might be written.

```
CKTDEF *RegCkt = NULL;
int NetPos = 0, GatePos = 0;

void FAR PASCAL _export RegisterCircuit(CKTDEF *inckt)
{
    RegCkt = inckt;
    NetPos = 0;
    GatePos = 0;
}
```

The functions FirstGate and NextGate might be written as follows.

```
int FAR PASCAL _export FirstGate(char *Ostr)
{
    if (RegCkt == 0)
    {
        return 0;
    }
    NetPos = 0;
    if (RegCkt->CircuitHead->gate_count < 1)
    {
        return 0;
    }
    else
    {
        strcpy(Ostr,RegCkt->CircuitHead->gatelist[0]->id->name);
        return -1;
    }
}
```

```
int FAR PASCAL _export NextGate(char *Ostr)
{
    if (RegCkt == 0)
    {
        return 0;
    }
    NetPos++;
    if (NetPos >= RegCkt->CircuitHead->gate_count)
    {
        return 0;
    }
    else
    {
        strcpy(Ostr,RegCkt->CircuitHead->gatelist[NetPos]->id->name);
        return -1;
    }
}
```

The functions **NextGate** and **NextNet** must return 0 if there are no more names, and -1 otherwise. The functions FirstNet and FirstGate must check for zero nets or zero gates, and return 0 if there aren't any. They should return -1 otherwise. **FirstNet** and **FirstGate** must also reinitialize the **NextNet** and **NextGate** functions.

Results: Create a Visual Basic program to test your DLL. Demonstrate that your program works. Turn in printouts of the DLL source code and the Visual Basic program.