

Chapter 2

Object Oriented Logic Modeling

Zainalabedin Navabi

Object Oriented Logic Modeling

Procedural Languages for Hardware Modeling

Types and Operators for Logic Modeling

- + Basic Logic Simulation
- + Enhanced Logic Simulation with Timing
- + More Functions for Wires and Gates
- + Inheritance in Logic Structures
- + Hierarchical Modeling of Digital Components

Summary

Object Oriented Logic Modeling

Procedural Languages for Hardware Modeling

Types and Operators for Logic Modeling

- Basic Logic Simulation

- + Logic functions

- Building higher level structures

- Handling 4-value logic

- Logic vector

- Sequential circuit modeling

- Using pointers for logic vectors

- Enhanced Logic Simulation with Timing

- Using *struct* for timing and logic

- Gates that handle timing

Utility functions

Timing in logic structures

Overloading logical operators

Using Boolean expressions

- More Functions for Wires and Gates

- Gate classes

- Carrier generic modeling

- Pointer-based logic classes

- Gate classes with power and timing calculation

- Wire and gate vectors

Object Oriented Logic Modeling

- Inheritance in Logic Structures

- A generic gate definition

- Gates to include timing

- Building structures from objects

- Hierarchical Modeling of Digital Components

- Wire functionalities

- Gate functionalities

- Polymorphic gate base

- Flip flop description hierarchies

Object Oriented Logic Modeling

Procedural Languages for Hardware Modeling

Types and Operators for Logic Modeling

- + Basic Logic Simulation
- + Enhanced Logic Simulation with Timing
- + More Functions for Wires and Gates
- + Inheritance in Logic Structures
- + Hierarchical Modeling of Digital Components

Summary

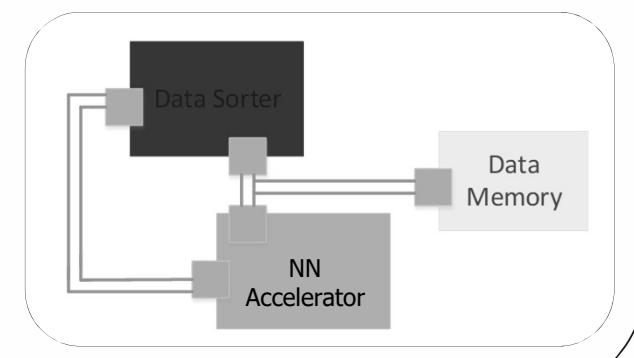
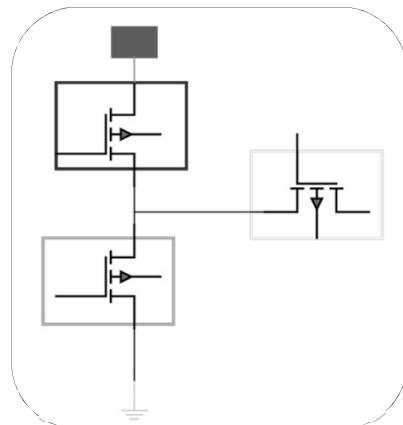
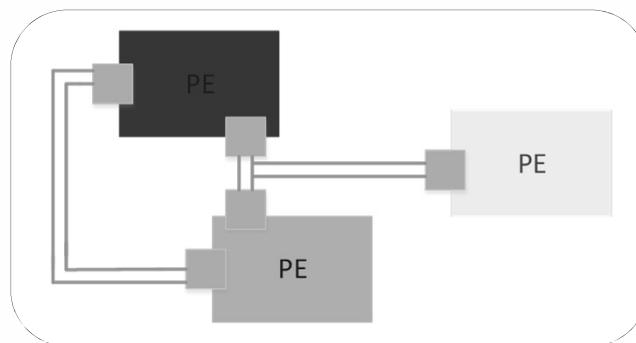
Procedural languages for Hardware Modeling

- **Hardware Description Language**

- Concurrency
- Timing

- **Software Description Languages**

- Procedural



Procedural languages for Hardware Modeling

◎ C++ Environment

The screenshot shows a C++ development environment with two open files:

- CPP Basics.h**: Contains the following code:

```
1 #include <iostream>
2 using namespace std;
```
- CPP Basics.cpp**: Contains the following code:

```
1 #include "CPP Basics.h"
2
3 int main()
4 {
5     int A, B, C;
6
7     cout << "Starting Simulation ..." << "\n";
8     cout << "Enter A: "; cin >> A;
9     cout << "Enter B: "; cin >> B;
10    C = A + B;
11    cout << "Add result is: " << C << "\n";
12    return 0;
13 }
```

Ref. [1] - Structure of a program - P. 7-10

The terminal window displays the output of the program:

```
C:\WINDOWS\system32\cmd.exe
Starting Simulation ...
Enter A: 4
Enter B: 7
Add result is: 11
Press any key to continue . . .
```

Ref. [1] - Basic Input/Output - P. 29-31

Object Oriented Logic Modeling

Procedural Languages for Hardware Modeling

Types and Operators for Logic Modeling

- + Basic Logic Simulation
- + Enhanced Logic Simulation with Timing
- + More Functions for Wires and Gates
- + Inheritance in Logic Structures
- + Hierarchical Modeling of Digital Components

Summary

Types and Operators for Logic Modeling

Group	Type names	Note on size/Precision
Character Types	Char	Exactly one byte in size. At least 8 bits
Integer Types (signed)	Signed Char	Same size as char. At least 8 bits
	Signed Int	At least 16 bits
Integer Types (unsigned)	Unsigned Char	Same size as char. At least 8 bits
	Unsigned Int	At least 16 bits
Floating-point Type	Float	
	Double	Precision not less than float
	Long Double	Precision not less than float
Boolean Type	Bool	
Void Type	Void	No storage

Types and Operators for Logic Modeling

Using Boolean Type

Ref. [1] –

Declaration of variables - P. 12

Scope of variables - P. 14

Initialization of variables - P.15

known as C-like
initialization

known as constructor
initialization

Boolean AND Operator

Ref. [1] – Operators
- P. 21-28

The screenshot shows a code editor with two tabs: Boolean Type.h and Boolean Type.cpp. The Boolean Type.h tab contains the declaration of a Boolean type using namespaces and includes. The Boolean Type.cpp tab contains the implementation of a main function that performs logic simulation and uses the Boolean type.

```
Boolean Type.h // X
Boolean Type (Global Scope)
1 #include <iostream>
2 using namespace std;
3

Boolean Type.cpp // X
Boolean Type (Global Scope)
1 #include "Boolean Type.h"
2
3 int main ()
4 {
5     //bool a = true;
6     //bool b = false;
7     bool a(0);
8     bool b(1);
9     bool anding;
10    int go;
11    cout << "Performing Logic Simulation . . .\n";
12    anding = a && b;
13    cout << "a:" << a << " b:" << b << " anding:" << anding << "\n";
14    cout << "Enter 0 to exit:";
15    cin >> go;
16
17    return 0;
18 }
```

Types and Operators for Logic Modeling

Using Char Type

Ref. [1] – Preprocessor
directives - P.133-P.134

Macro Declaration:
Converts '0' and '1' to 0 and
1 for Boolean operations

Iteration Structure:
while loop

Selective Structure:
switch case

Ref. [1] – Control
Structures - P. 34-40

```
Character Type.h
Character Type
#include <iostream>
using namespace std;

Character Type.cpp*
Character Type
#define BIT(c)(c=='0'?0:1)

int main ()
{
    char i1 = '0';
    char i2 = '0';
    char op;
    bool go(1);
    while (go) {
        cout << "Enter Operation (A, O, X) followed by input values: ";
        cin >> op >> i1 >> i2;
        switch (op) {
            case 'A': case 'a':
                cout << i1 << " AND " << i2 << " is: " << (BIT(i1) && BIT(i2)) << '\n';
                break;
            case 'O': case 'o':
                cout << i1 << " OR " << i2 << " is: " << (BIT(i1) || BIT(i2)) << '\n';
                break;
            case 'X': case 'x':
                cout << i1 << " XOR " << i2 << " is: " << (BIT(i1) != BIT(i2)) << '\n';
                break;
            default:
                cout << "Wrong operation \n";
        }
        cout << "Enter 0 to end:"; cin >> go;
    }
    return 0;
}
```

Types and Operators for Logic Modeling

Using Enumeration

Ref. [1] – Enumerations
- P.84-P.85

Creating a new data type using enum

Enumerations are type compatible with numeric variables

Ref. [1] –
Constants - P. 17-20
Arrays - P. 54-59

Constant arrays

Conditional Structure:
if - else

Ref. [1] – Control Structures - P. 34-40

Four Value System.h

```
#include <iostream>
#include <string>
using namespace std;

enum lv4 {lX, l0, l1, l2};
const lv4 lv4Value [4] = {lX, l0, l1, l2};
const string lv4Image [4] = {"lX", "l0", "l1", "l2"};
```

Four Value System.cpp

```
#include "Four Value System.h"

lv4 ANDlv4 (lv4 a, lv4 b)
{
    lv4 w;
    if (a==lX || b==lX || a==lZ || b==lZ) w=lX;
    else if (a==l1 & b==l1) w=l1;
    else w=l0;
    return w;
}

lv4 ORlv4 (lv4 a, lv4 b){ ... }

lv4 XORlv4 (lv4 a, lv4 b){ ... }

int main (){ ... }
```

Types and Operators for Logic Modeling

◎ Using Enumeration (cont.)

Ref. [1] – Control
Structures - P. 34-40

Convert Integer value to
lv4 enumeration type

Conversion to string
for printing

Iteration Structure:
do-while loop

```
Four Value System.cpp  Four Value System.h
Enum Type (Global Scope)
1 #include "Four Value System.h"
2
3 #lv4 ANDlv4 (lv4 a, lv4 b){ ... }
11
12 #lv4 ORlv4 (lv4 a, lv4 b){ ... }
20
21 #lv4 XORlv4 (lv4 a, lv4 b){ ... }
29
30 int main ()
31 {
32     lv4 i1 = lx;
33     lv4 i2 = lx;
34     lv4 out = lx;
35     int Ii1, Ii2, Iout;
36     char op;
37     bool go;
38
39     do {
40         cout << "Enter operation (A,O,X), then inputs (0 to 3): ";
41         cin >> op >> Ii1 >> Ii2;
42         i1=lv4Value[Ii1]; i2=lv4Value[Ii2];
43         switch (op) {
44             case 'A': out = ANDlv4 (i1, i2); break;
45             case 'O': out = ORlv4 (i1, i2); break;
46             case 'X': out = XORlv4 (i1, i2); break;
47             default: out = lx;
48         }
49         cout << i1 << " " << op << " " << i2;
50         cout << ", is: " << out << '\n';
51         cout << lv4Image[i1] << " " << op << " " << lv4Image[i2];
52         cout << ", is: " << lv4Image[out] << '\n';
53         cout << "Enter 0 to end:"; cin >> go;
54
55     } while (go);
56
57     return 0;
58 }
```

Types and Operators for Logic Modeling

◎ Using String

▪ Waveform Generation

Ref. [1] –
Introduction to
strings - P. 15-16

Ref. [1] –
Functions (I) -
P. 41-46

String is not a fundamental type, but it behaves in a similar way

MIN Macro Declaration

The wave function gets a sequence of 1s and 0s and turns into waveform

This function is equivalent to BIT macro

The operation function:
Bool is good for logical operations

```
String Character.h
String Characters (Global Scope)
1 #include <iostream>
2 #include <string>
using namespace std;

String Character.cpp
String Characters (Global Scope)
3 #define MIN(a,b)((a<b)?a:b)
4
5 int wave (string seq)
6 {
7     int i, l;
8     l = seq.length();
9     for (i=0; i < l; i++)
10    {
11        if (seq[i]=='0') cout << "_";
12        else cout << "-";
13    }
14    cout << '\n';
15    return 1;
16}
17
18 bool char2bool (char c)
19 {
20     if (c=='0') return 0;
21     else return 1;
22}
23
24 bool operation (string fn, bool in1, bool in2)
25 {
26     bool out;
27     if (fn == "AND" || fn == "and") out = in1 && in2;
28     else if (fn == "OR" || fn == "or") out = in1 || in2;
29     else if (fn == "XOR" || fn == "xor") out = in1 != in2;
30     else out = 0;
31     return out;
32}
```

Types and Operators for Logic Modeling

◎ Using String (cont.) ▪ Waveform Generation

Ref. [1] – Control
Structures - P. 34-40

Calling the functions

Using the macro:
for calculating the output
waveform length

Iteration Structure:
for loop

The screenshot shows a code editor window titled "String Character.cpp". The code implements logic operations on character strings. Annotations highlight specific parts of the code:

- A callout points to the line "cout << "Enter logic type and input sequences: "; cin >> logic >> i1Seq >> i2Seq;" with the text "Calling the functions".
- A callout points to the line "#include <iomanip>" with the text "Using the macro: for calculating the output waveform length".
- A callout points to the line "for (i=0; i<outLen; i++) {" with the text "Iteration Structure: for loop".

```
String Character.cpp  # X
String Characters  (Global Scope)
17 bool char2bool (char c) { ... }
22
23 bool operation (string fn, bool in1, bool in2) { ... }
32
33 int main ()
34 {
35     string i1Seq, i2Seq;
36     string logic;
37     int i, i1Len, i2Len, outLen;
38     bool i1=0, i2=0, out=0;
39     bool go(1);
40     while (go) {
41         cout << "Enter logic type and input sequences: ";
42         cin >> logic >> i1Seq >> i2Seq;
43         i1Len=wave (i1Seq);
44         i2Len=wave (i2Seq);
45         outLen = MIN (i1Len, i2Len);
46         string outSeq (outLen, '0');
47         for (i=0; i<outLen; i++) {
48             i1 = char2bool (i1Seq[i]);
49             i2 = char2bool (i2Seq[i]);
50             out=operation(logic, i1,i2);
51             outSeq[i] = out ? '1' : '0';
52         }
53         outLen=wave (outSeq); cout << '\n';
54         cout << "Enter 0 to end: "; cin >> go;
55     }
56     return 0;
57 }
```

Types and Operators for Logic Modeling

⦿ Using String (cont.)

■ Waveform Generation

Object Oriented Logic Modeling

Procedural Lang. for Hardware Modeling
Types and Operators for Logic Modeling

- Basic Logic Simulation

- Logic functions
 - Arguments passed by value
 - Arguments passed by reference
 - Functions with no type
 - Using default values
 - Function overloading

Building higher level structures

Handling 4-value logic

Logic vector

Sequential circuit modeling

Using pointers for logic vectors

- + Enhanced Logic Simulation with Timing
 - + More Functions for Wires and Gates
 - + Inheritance in Logic Structures
 - + Hierarchical Modeling of Digital Components
- Summary

- Basic Logic Simulation

Logic Functions

◎ Gate Function Prototypes

Ref. [1] – Functions (II)
- P. 47-49

Passed by value

Passed by reference

Passing function as
an argument

Using default values

Logic vector modeling
using arrays

LogicGates.h

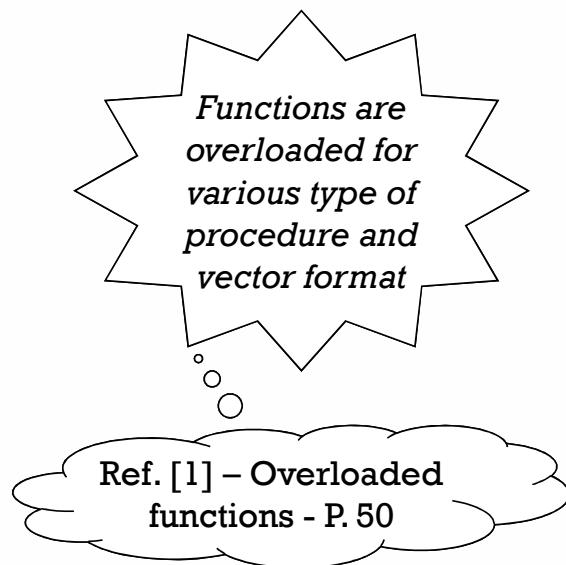
Primitives.h

To use this function
with fewer arguments,
all arguments must
have default values

- Basic Logic Simulation

Logic Functions

⦿ Gate Function Prototypes (cont.)



```
primitives.cpp logicGates.cpp
Logic Simulation (Global Scope)
2 bool and (bool a, bool b)
3 {
4     return (a && b);
5 }
6
8 bool or (bool a, bool b){ ... }
12
13 bool not (bool a){ ... }
18
22
23 bool nor (bool a, bool b){ ... }
27
28 bool xor (bool a, bool b){ ... }
32
33 void and (bool a, bool b, bool& w)
34 {
35     w = a && b;
36 }
38
42
43 void or (bool a, bool b, bool& w){ ... }
47
48 void not (bool a, bool& w){ ... }
52
53 void nand (bool a, bool b, bool& w){ ... }
57
58 void nor (bool a, bool b, bool& w){ ... }
59
60 void xor (bool a, bool b, bool& w){ ... }
62
63
64 void logic (bool a, bool b, void (*f) (bool, bool, bool&))
65 {
66     bool w;
67     (*f) (a, b, w);
68     return (w);
69 }
```

Primitives.cpp

Passed by value

Passed by reference

Passing function as an argument

Passing by reference

- allows to manipulate from inside a function the value of an external variable
- is also an effective way to allow a function to return more than one value

- Basic Logic Simulation

Building Higher Level Structures

- Calling three groups of the logic functions for three different implementations of full-adder

Order matters in procedural languages

```
primitives.cpp logicGates.cpp* (Global Scope)
1 #include "logicGates.h"
2 #include "primitives.h"
3
4 void fullAdder (bool a, bool b, bool ci, bool& co, bool& sum)
5 {
6     bool axb, ab, abc;
7
8     axb = xor (a, b);
9     ab = and (a, b);
10    abc = and (axb, ci);
11    co = or (ab, abc);
12    sum = xor (axb, ci);
13}
```

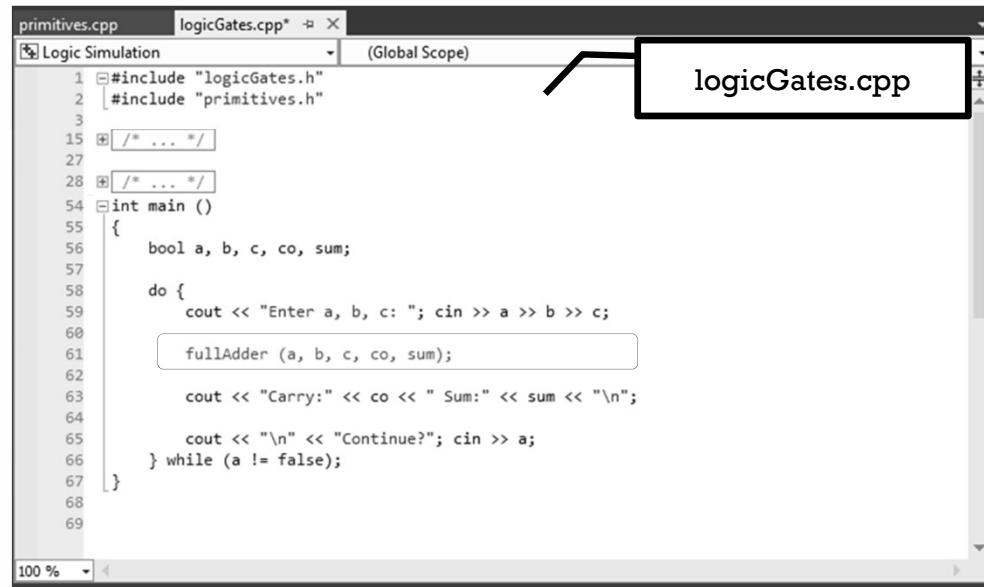
logicGates.cpp

```
primitives.cpp logicGates.cpp* (Global Scope)
1 #include "logicGates.h"
2 #include "primitives.h"
3
4 /* ... */
16
17 /* ... */
29
30 void fullAdder (bool a, bool b, bool ci, bool& co, bool& sum)
31 {
32     bool axb, ab, abc;
33
34     axb = logic (a, b, xor); // uses: void xor (bool, bool, bool&)
35     ab = logic (a, b, and);
36     abc = logic (axb, ci, and);
37     co = logic (ab, abc, or);
38     sum = logic (axb, ci, xor);
39 }
40
41 void fullAdder (bool a, bool b, bool ci, bool& co, bool& sum)
42 {
43     bool ab, bc, ac;
44
45     ab = and5 (a, b);
46     bc = and5 (b, ci);
47     ac = and5 (a, ci);
48     co = or5 (ab, bc, ac);
49     sum = xor5 (a, b, ci);
50 }
51
52 int main () { ... }
```

- Basic Logic Simulation

Building Higher Level Structures

- Calling the full-adder in *main* as a testbench



A screenshot of a code editor window titled "logicGates.cpp". The code is as follows:

```
primitives.cpp logicGates.cpp > X
Logic Simulation (Global Scope)
1 #include "logicGates.h"
2 #include "primitives.h"
3
15 /* ... */
27
28 /* ... */
54 int main ()
55 {
56     bool a, b, c, co, sum;
57
58     do {
59         cout << "Enter a, b, c: "; cin >> a >> b >> c;
60
61         fullAdder (a, b, c, co, sum);
62
63         cout << "Carry:" << co << " Sum:" << sum << "\n";
64
65         cout << "\n" << "Continue?"; cin >> a;
66     } while (a != false);
67 }
68
69
```

The line "fullAdder (a, b, c, co, sum);" is highlighted with a red rectangle. A black bracket points from the word "fullAdder" to a callout box labeled "logicGates.cpp".

- Basic Logic Simulation

Handling 4-value Logic

◎ Four-Value Logic System

Value	Description
0	Forcing 0 or Pulled 0
1	Forcing 1 or Pulled 1
Z	Float or High Impedance
X	Uninitialized or Unknown

Z is the weakest logic value

X is the strongest logic value

- Basic Logic Simulation

Handling 4-value Logic

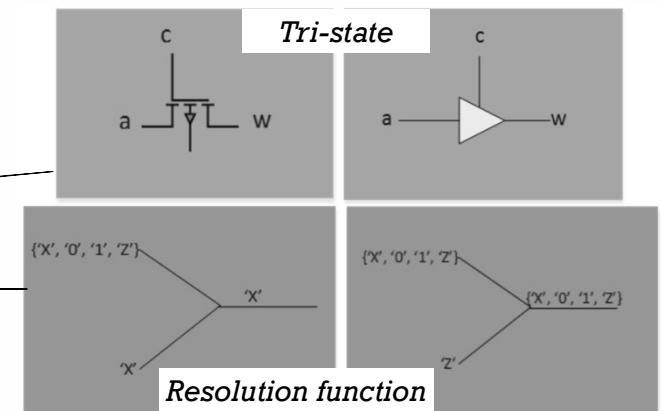
- Using *char* for handling 4-value logic due to easier and more expressive *in* and *out*

```
characterFunctions.h
1 #include <iostream>
2 using namespace std;
3

characterPrimitives.h
1 char and (char a, char b);
2 char or (char a, char b);
3 char not (char a);
4 char tri (char a, char c);
5 char resolve (char a, char c);
6 char xor (char a, char b);

7

8 void fullAdder (char a, char b, char ci, char & co, char & sum);
9
```



- Basic Logic Simulation

Handling 4-value Logic

- ⦿ We have to generate our own logical functions
- ⦿ This happens once and can easily be reused

		Resolution function		
b \ a	x	0	1	z
x	x	x	x	x
0	x	0	x	0
1	x	x	1	1
z	x	0	1	z

```

characterPrimitives.cpp + X characterFunctions.cpp
Character Logic (Global Scope)
1 #include "characterPrimitives.h"
2
3 char and (char a, char b)
4 {
5     if ((a=='0')||(b=='0')) return '0';
6     else if ((a=='1')&&(b=='1')) return '1';
7     else return 'X';
8 }
9
10 char or (char a, char b){ ... }
11
12 char not (char a){ ... }
13
14 char tri (char a, char c){ ... }
15
16 char resolve (char a, char b)
17 {
18     if (a=='Z' || a==b) return b;
19     else if (b=='Z') return a;
20     else return 'X';
21 }
22
23 char xor (char a, char b){ ... }
24
25 void fullAdder (char a, char b, char ci, char & co, char & sum)
26 {
27     char axb, ab, abc;
28
29     axb = xor (a, b);
30     ab = and (a, b);
31     abc = and (axb, ci);
32     co = or (ab, abc);
33     sum = xor (axb, ci);
34 }
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55

```

Overloaded AND using char

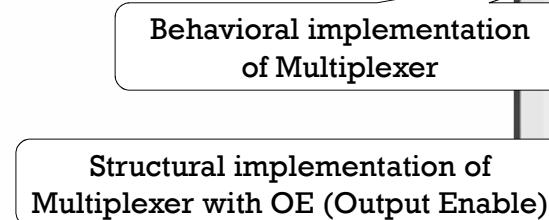
The resolve function using char

Using the char type logic functions for the full-adder implementation

- Basic Logic Simulation

Handling 4-value Logic

- ◎ Using *char* for the multiplexer implementation



```
characterPrimitives.cpp characterFunctions.cpp > X
Character Logic (Global Scope)
1 #include "characterPrimitives.h"
2 #include "characterFunctions.h"
3
4 void muxStd2T01 (char a, char b, char& w, char sel)
5 {
6     w = (sel=='1') ? b : a;
7 }
8
9 void muxTri2T01 (char a, char b, char& w, char sel, char oe)
10 {
11     char selB, selB_oe, sel_oe;
12     char asel;
13     char bsel;
14
15     selB = not(sel);
16     selB_oe = and(selB, oe);
17     sel_oe = and(sel, oe);
18     asel = tri(a, selB_oe);
19     bsel = tri(b, sel_oe);
20     w = resolve(asel, bsel);
21
22
23 int main () { ... }
```

CharacterFunctions.cpp

- Basic Logic Simulation

Logic Vector

⦿ Overloaded logic functions for vector format

- Using Boolean Type

The screenshot shows a code editor with three files:

- VectorFunctions.h**: Contains the definition of overloaded logic functions for singular bits:

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
```
- VectorPrimitives.h**: Contains the definition of overloaded logic functions for multiple bits:

```
1 bool and (bool a, bool b);
2 bool or (bool a, bool b);
3 bool not (bool a);
```

```
5 void and (bool a[], bool b[], bool w[], const int SIZE);
6 void or (bool a[], bool b[], bool w[], const int SIZE);
```
- vectorFunctions.cpp**: (Partially visible)

Annotations explain the code:

- Singular bit implementation of the logic functions** points to the first three lines of **VectorFunctions.h**.
- Multiple bit implementation of the logic functions** points to the first three lines of **VectorPrimitives.h**.
- The size of vector** points to the parameter `SIZE` in the `void and` and `void or` functions.
- By default, arrays are passed by reference to first location** is a general note about array passing in C++.
- Ref. [1] – Arrays - P. 54-59** is a reference note.

- Basic Logic Simulation

Logic Vector

- Overloaded logic functions for vector format
 - Using Boolean Type (cont.)

Singular bit implementation
of the logic functions

Multiple bit implementation
of the logic functions

```
vectorPrimitives.cpp  X  vectorFunctions.cpp
Logic Vector Simulation  (Global Scope)
1 #include "vectorPrimitives.h"
2
3 bool and (bool a, bool b)
4 {
5     return (a && b);
6 }
7
8 bool or (bool a, bool b) { ... }
9
10 bool not (bool a) { ... }
11
12 void and (bool a[], bool b[], bool w[], const int SIZE)
13 {
14     int i;
15     for (i=0; i<SIZE; i++) {
16         w[i] = a[i] && b[i];
17     }
18 }
19
20 void or (bool a[], bool b[], bool w[], const int SIZE) { ... }
```

VectorPrimitives.cpp

Boolean AND operator

The size of vector

Loop and index need the size of vector

Repeats the Boolean AND operator for all bits of the vector size

– Basic Logic Simulation

Logic Vector

- ⦿ Overloaded logic functions for vector format

▪ Using Boolean Type (cont.)

Read string and turns it
into an array of *bool*

Implementing 8-bit *bool* vector based Multiplexer

Using the singular bit implementation of AND function

Using the multiple bit implementation of OR function

vectorPrimitives.cpp | vectorFunctions.cpp

Logic Vector Simulation (Global Scope)

VectorFunctions.cpp

```
1 #include "vectorPrimitives.h"
2 #include "vectorFunctions.h"
3
4 void getBits (string vectorName, int numBits, bool values[])
5 {
6     string valuesS;
7     int i;
8     cout << "Enter " << numBits << " bits of " << vectorName << ": ";
9     cin >> valuesS;
10    for (i=0; i<numBits; i++){
11        if (valuesS[i] == '1') values[i] = true;
12        else values[i] = false;
13    }
14 }
15
16 void putBits (string vectorName, int numBits, bool values[]) { ... }
17 void two2OneMux (bool a[], bool b[], bool w[], bool sel, int SIZE=8)
18 {
19     bool as [8];
20     bool bs [8];
21
22     int i;
23     for (i=0; i<SIZE; i++) {
24         as[i] = and (a[i], not(sel));
25     }
26     for (i=0; i<SIZE; i++) {
27         bs[i] = and (b[i], sel);
28     }
29
30     or (as, bs, w, SIZE);
31 }
32
33 void two2OneMuxB (bool a[], bool b[], bool w[], bool sel, int SIZE=8) { ... }
34
35 int main () { ... }
```

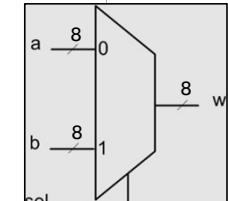
it

ool
ker

ion

on

100 %

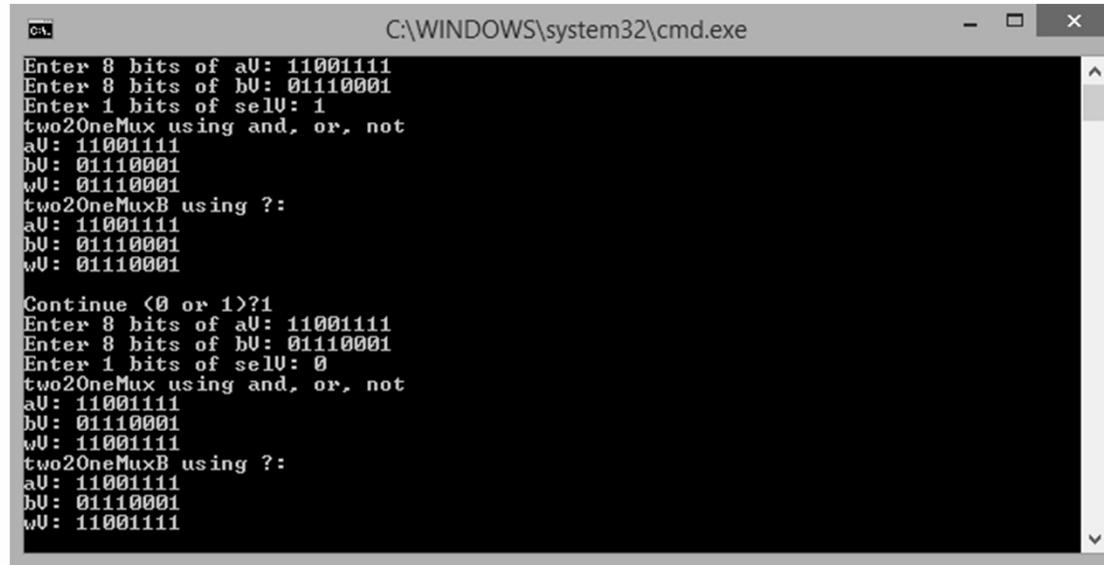


- Basic Logic Simulation

Logic Vector

- Overloaded logic functions for vector format

- Using Boolean Type (cont.)



The screenshot shows a Windows command prompt window titled 'cmd' with the path 'C:\WINDOWS\system32\cmd.exe'. The window displays the following text:

```
Enter 8 bits of a0: 11001111
Enter 8 bits of b0: 01110001
Enter 1 bits of sel0: 1
two2OneMux using and, or, not
a0: 11001111
b0: 01110001
w0: 01110001
two2OneMuxB using ?:
a0: 11001111
b0: 01110001
w0: 01110001

Continue <0 or 1>?1
Enter 8 bits of a0: 11001111
Enter 8 bits of b0: 01110001
Enter 1 bits of sel0: 0
two2OneMux using and, or, not
a0: 11001111
b0: 01110001
w0: 11001111
two2OneMuxB using ?:
a0: 11001111
b0: 01110001
w0: 11001111
```

- Basic Logic Simulation

Logic Vector

⦿ Overloaded logic functions for vector format

▪ Using Char Type

- Removes the complications of using Boolean type including
 - a) Entering the size of vector
 - b) Requiring the conversion functions (*getBits* or *putBits*)

By default, arrays are passed by reference to first location

In Char type, the null character ('\0') marks the end of the vector

© Zainalabedin Navabi – Ob

```
characterVectorFunctions.cpp      characterVectorFunctions.h      characterVectorPrimitives.h + X
Character Vector Logic          (Global Scope)
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 char and (char a, char b);
6 char or (char a, char b);
7 char not (char a);
8 char tri (char a, char c);
9 char resolve (char a, char c);
10
11 void and (char a[], char b[], char w[]);
12 void or (char a[], char b[], char w[]);
13 void tri (char a[], char c, char w[]);
14 void resolve (char a[], char b[], char w[]);
15
16 char xor (char a, char b);
17 void fullAdder (char a, char b, char ci, char & co, char & sum);
```

Singular bit implementation of the logic functions

Multiple bit implementation of the logic functions

Ref. [1] – Character Sequences - P. 60-62

- Basic Logic Simulation

Logic Vector

- Overloaded logic functions for vector format
 - Using Char Type (cont.)

This loop is repeated for as long as it has not reached the end marker of the vector

Using the singular bit implementation of *resolve* function

The screenshot shows a code editor with three tabs: characterVectorFunctions.cpp, characterVectorPrimitives.h, and characterVectorPrimitives.cpp. The characterVectorPrimitives.h tab is active, displaying the header file for logic functions. The characterVectorPrimitives.cpp tab shows the implementation of these functions. The code includes functions for AND, OR, TRI, and resolve operations on character arrays. A callout box points to the AND function implementation, explaining its use of a singular bit and how it handles the null character at the end of the vector. Another callout box points to the resolve function implementation, also using a singular bit approach.

```
characterVectorFunctions.cpp characterVectorPrimitives.h characterVectorPrimitives.cpp
38 void and (char a[], char b[], char w[])
39 {
40     int i=0;
41     while (a[i] != '\0') {
42         w[i] = and (a[i], b[i]);
43         i++;
44     }
45     w[i] = '\0';
46 }
47
48 void or (char a[], char b[], char w[])
49
50 void tri (char a[], char c, char w[])
51
52 void resolve (char a[], char b[], char w[])
53 {
54     int i=0;
55     while (a[i] != '\0') {
56         w[i] = resolve (a[i], b[i]);
57         i++;
58     }
59     w[i] = '\0';
60 }
61
62 char xor (char a, char b)
63
64 void fullAdder (char a, char b, char ci, char & co, char & sum)
65 {
66     char axb, ab, abc;
67
68     axb = xor (a, b);
69     ab = and (a, b);
70     abc = and (axb, ci);
71     co = or (ab, abc);
72     sum = xor (axb, ci);
73 }
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
```

- Basic Logic Simulation

Logic Vector

- ⦿ Overloaded logic functions for vector format
 - Using Char Type (cont.)

Behavioral implementation of char vector based Multiplexer

Structural implementation of char vector based Multiplexer with OE (Output Enable)

Using the singular bit implementation of logic function

Using the multiple bit implementation of logic function

If fewer than 8-bits are entered, using *cin* automatically puts "\0" at the end of string

```

characterVectorFunctions.cpp  characterVectorPrimitives.h  characterVectorFunctions.cpp
Character Vector Logic  (Global Scope)  characterVectorFunctions.cpp
1 #include "characterVectorPrimitives.h"
2 #include "characterVectorFunctions.h"
3
4 void mux8Std2T01 (char a[], char b[], char w[], char sel)
5 {
6     int i=0;
7     do {
8         w[i] = (sel=='1') ? b[i] : a[i];
9     } while (a[i++] != '\0');
11
12 void mux8Tri2T01 (char a[], char b[], char w[], char sel, char oe)
13 {
14     char selB, selB_oe, sel_oe;
15     char asel [9];
16     char bsel [9];
17
18     selB = not(sel);
19     selB_oe = and(selB, oe);
20     sel_oe = and(sel, oe);
21
22     tri(a, selB_oe, asel);
23     tri(b, sel_oe, bsel);
24     resolve(asel, bsel, w);
25
26 int main ()
27 {
28     char aCV [9], bCV [9];
29     char sel, oe;
30     char wCV [9];
31     int ai;
32     do {
33         cout << "Enter eight bits of aCV <space> bCV: "; cin >> aCV >> bCV;
34         cout << "Enter sel <space> oe: "; cin >> sel >> oe;
35
36         mux8Std2T01 (aCV, bCV, wCV, sel);
37         cout << "The " << strlen(wCV) << " bits of wC become as follows: \n";
}

```

– Basic Logic Simulation

Sequential Circuit Modeling

- ⦿ D Flip-Flop with Positive-edge, Asynchronous, active High reset

The screenshot shows a code editor with two tabs open:

- SequentialFunctions.h**: Contains C++ code including #include <iostream>, #include <fstream>, #include <string>, and using namespace std;.
- CharacterPrimitives.h**: Contains declarations for logical operations: char and (char a, char b);, char or (char a, char b);, char not (char a);, and void dff_PAH (char D, char clk, char reset, char&Q);.

characterPrimitives.h sequentialFunctions.h characterPrimitives.cpp sequentialFunctions.cpp

Sequential Model (Global Scope)

```
1    char and (char a, char b) { ... }
2
3    char or (char a, char b) { ... }
4
5    char not (char a) { ... }
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22 void dff_PAH (char D, char clk, char reset, char&Q)
23 // Posedge, Asynch, active-Low
24 {
25     if (reset=='1') Q='0';
26     else if (clk=='P') Q=D;
27 }
```

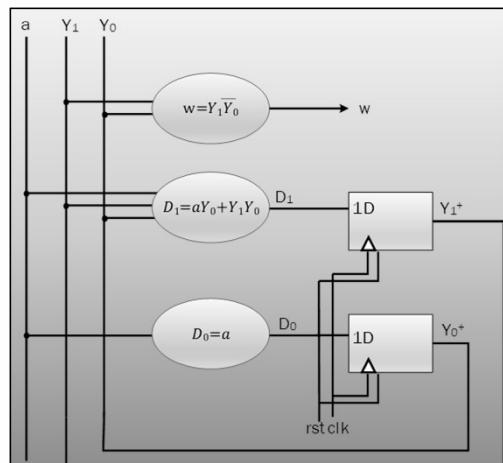
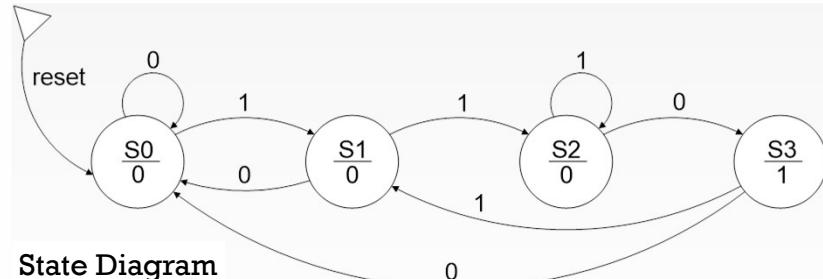
CharacterPrimitives.cpp



- Basic Logic Simulation

Sequential Circuit Modeling

Moore 110 sequence detector



State	State Table			Transition Table			Excitation Table							
	0	a	1	w	$Y_1 Y_0$	0	a	1	w	$Y_1 Y_0$	0	a	1	w
00	S0	S0	S1	0	00	00	01		0	00	00	01		0
01	S1	S0	S2	0	01	00	11		0	01	00	11		0
11	S2	S3	S2	0	11	10	11		0	11	10	11		0
10	S3	S0	S1	1	10	00	01	1	1	10	00	01	1	1
					$Y_1^+ Y_0^+$					D1D0				

$Y_1 Y_0$	0	1
00	0	0
01	0	1
11	1	1
10	0	0

$$D_1 = aY_0 + Y_1Y_0$$

$Y_1 Y_0$	0	1
00	0	1
01	0	1
11	0	1
10	0	1

$$D_0 = a$$

Y_0	0	1
0	0	1
1	0	0

$$w = Y_1 \bar{Y}_0$$

© Zainalabedin Navabi – Object Oriented Logic Modeling

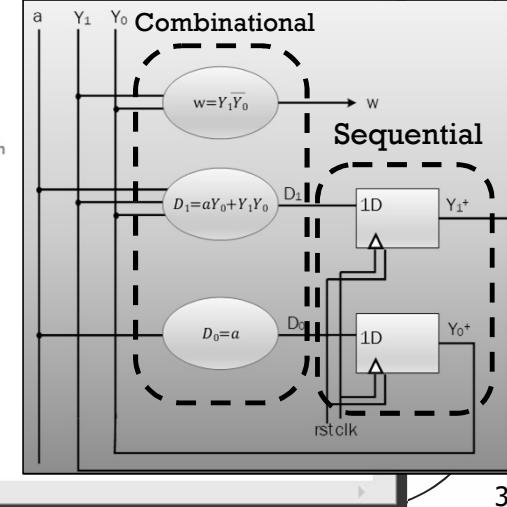
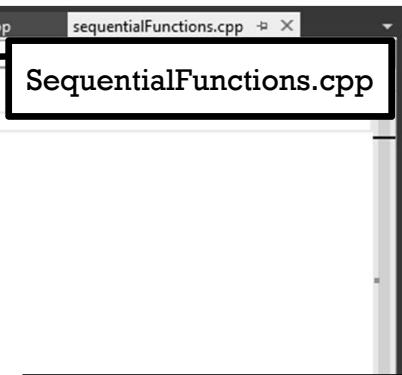
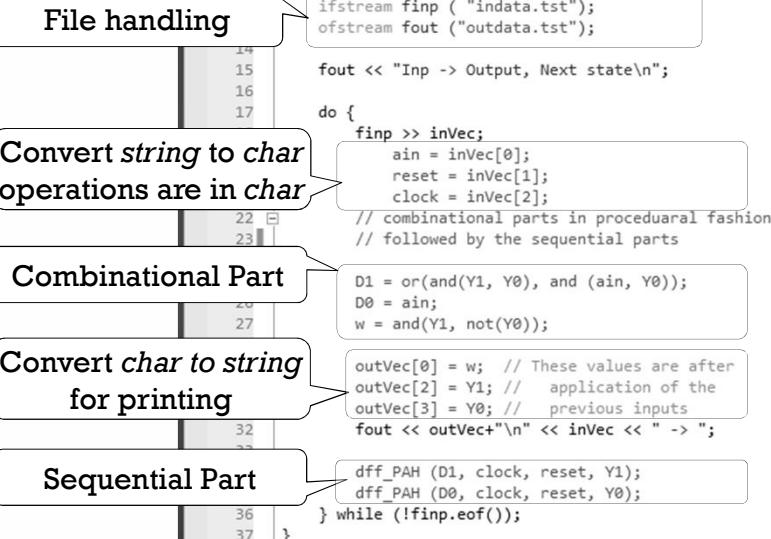
– Basic Logic Simulation

Sequential Circuit Modeling

- ⦿ Moore 110 sequence detector
(cont.)

Ref. [1] – Input/Output with files - P. 138-144

Input file		Output file	
	inidata.tst		outdata.tst
000		Inp	→ Output, Next state
000		X,XX	
000		000 → X,XX	
100		000 → X,XX	
10P		000 → X,XX	
010		100 → X,XX	
00P		10P → 0,X1	
10P		010 → 0,00	
10P		00P → 0,00	
00P		10P → 0,01	
00P		10P → 0,11	
10P		00P → 1,10	
10P		00P → 0,00	
10P		10P → 0,01	
00P		10P → 0,11	
10P		10P → 0,11	
10P		00P → 1,10	
00P		10P → 0,01	
10P		10P → 0,11	
00P		00P → 1,10	
10P		10P → 0,01	
10P		10P → 0,11	
reset, clock		w and 2-bits	



- Basic Logic Simulation

Using Pointers for Logic Vectors

- Overloaded logic functions for vector format using pointers instead of arrays

The screenshot shows a code editor with four tabs: `pointerFunctionsFileData.h`, `pointerPrimitives.h`, `pointerPrimitives.cpp`, and `pointerFunctionsFileData.cpp`. The `pointerPrimitives.h` tab is active, displaying the following C++ code:

```
1 void and (char a, char b, char & w);
2 void or (char a, char b, char & w);
3 void not (char a, char & w);
4 void tri (char a, char c, char & w);
5 void resolve (char a, char c, char & w);
6
7 void and (char* a, char* b, char* w);
8 void or (char *a, char *b, char *w);
9 void not (char *a, char *w);
10 void tri (char *a, char *c, char *w);
11 void resolve (char *a, char *b, char *w);
12
13 void mux8Std2T01 (char*, char*, char*, char);
14 void mux8Tri2T01 (char*, char*, char*, char, char);
```

Annotations on the left side of the code editor:

- A callout bubble points to the first five functions (single-bit implementation): "Singular bit implementation of the logic functions".
- A callout bubble points to the last six functions (multiple-bit implementation with pointer arguments): "Multiple bit implementation of the logic functions with pointer arguments".
- A callout bubble points to the bottom of the code editor window: "Ref. [1] – Pointers - P. 63-73".

An annotation on the right side of the code editor:

- A callout bubble points to the `pointerPrimitives.h` tab: "pointerPrimitives.h".
- A large starburst callout points to the `w` variable in the function signatures: "Using a pointer we can directly access the value stored in the variable which it points to".

- Basic Logic Simulation

Using Pointers for Logic Vectors

- Overloaded logic functions for vector format using pointers instead of arrays (cont.)

The screenshot shows a code editor window with several files open. The main file is `pointerPrimitives.cpp`, which contains implementations for various logic functions using pointers. Three specific sections of the code are highlighted with callout boxes:

- Singular bit implementation of the AND functions**: This box highlights the implementation of the `and` function, which takes three `char` arguments and returns a single `char` result based on the rules of Boolean AND.
- Multiple bit implementation of the AND functions with pointer arguments**: This box highlights the implementation of the `and` function, which takes three `char*` arguments and performs a bit-by-bit AND operation across multiple bits.
- Using the singular bit implementation of AND function**: This box highlights the implementation of the `and` function, which takes three `char` arguments and returns a single `char` result based on the rules of Boolean AND.

```
#include <iostream>
using namespace std;

void and (char a, char b, char & w)
{
    w = ((a=='0')||(b=='0')) ? '0':
        ((a=='1')&&(b=='1')) ? '1':
        'X';
}

void or (char a, char b, char & w) { ... }

void not (char a, char & w) { ... }

void tri (char a, char c, char & w) { ... }

void resolve (char a, char b, char & w) { ... }

void and (char* a, char* b, char* w)
{
    int i=0;
    do {
        and (*(a+i), *(b+i), *(w+i));
        i++;
    } while (*(a+i) != '\0');
    *(w+i) = '\0';
}

void or (char *a, char *b, char *w) { ... }

void not (char *a, char *w) { ... }

void tri (char *a, char *c, char *w) { ... }

void resolve (char *a, char *b, char *w) { ... }

void mux8Std2TO1 (char *a, char *b, char *w, char sel)
```

- Basic Logic Simulation

Using Pointers for Logic Vectors

- Overloaded logic functions for vector format using pointers instead of arrays (cont.)

pointerPrimitives.cpp

```
pointerFunctionsFileData.h pointerPrimitives.h pointerPrimitives.cpp* pointerPrimitives.h
Pointer Logic File Data (Global Scope) resolve(0)
84 void mux8Std2T01 (char *a, char *b, char *w, char sel)
85 {
86     int i=0;
87     do {
88         *(w+i) = (sel=='1') ? *(b+i) : *(a+i);
89         i++;
90     } while (*(a+i) != '\0');
91     *(w+i) = '\0';
92 }
93
94 void mux8Tri2T01 (char *a, char *b, char *w, char sel, char oe)
95 {
96     int i=0;
97     do {
98         if (oe == '1') *(w+i) = (sel=='1') ? *(b+i) : *(a+i);
99         else *(w+i) = 'Z';
100        i++;
101    } while (*(a+i) != '\0');
102    *(w+i) = '\0';
103 }
```

Behavioral implementation of standard Multiplexer

Behavioral implementation of Tri-state Multiplexer with OE

100 %

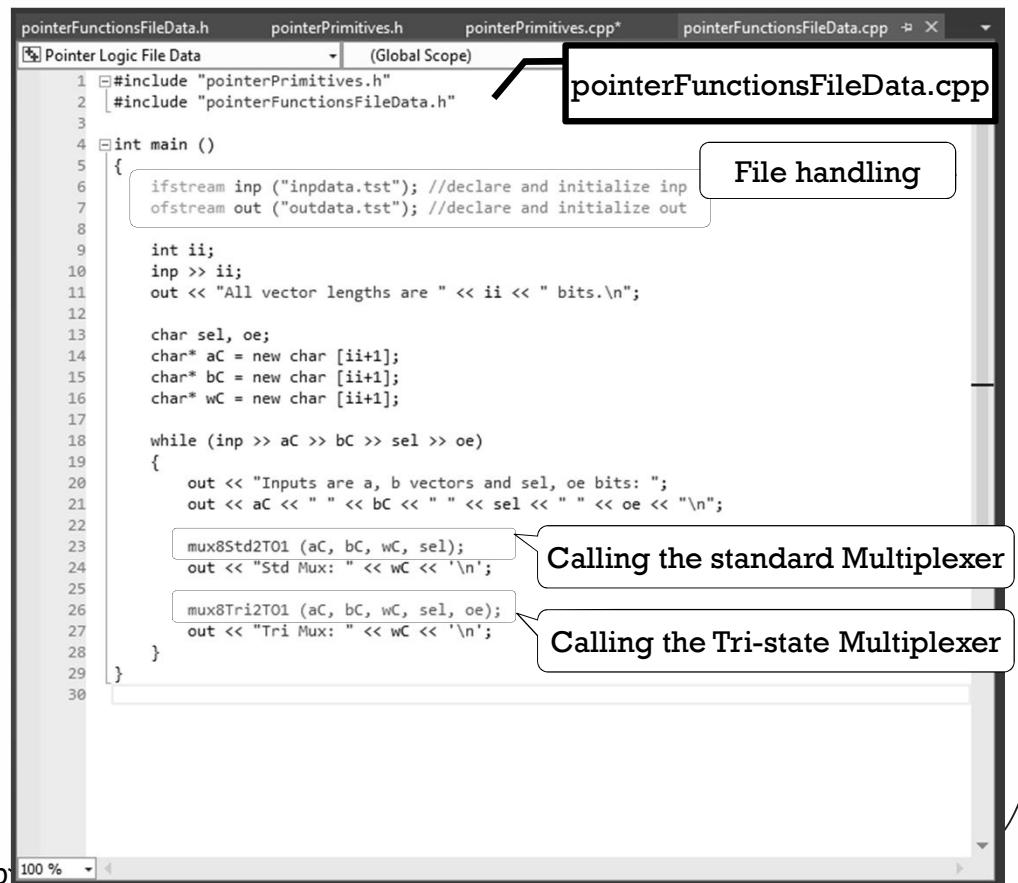
- Basic Logic Simulation

Using Pointers for Logic Vectors

- Overloaded logic functions for vector format using pointers instead of arrays (cont.)
 - Testing the multiplexers

Output file

```
outdata.tst <
All vector lengths are 8 bits.
Inputs are a, b vectors and sel, oe bits: 11001111 11110001 0 0
Std Mux: 11001111
Tri Mux: ZZZZZZZZ
Inputs are a, b vectors and sel, oe bits: 11110001 00010101 0 1
Std Mux: 11110001
Tri Mux: 11110001
Inputs are a, b vectors and sel, oe bits: 10101011 11110000 1 0
Std Mux: 11110000
Tri Mux: ZZZZZZZZ
Inputs are a, b vectors and sel, oe bits: 11001111 11001100 1 1
Std Mux: 11001100
Tri Mux: 11001100
Inputs are a, b vectors and sel, oe bits: 11110000 11101010 1 1
Std Mux: 11101010
Tri Mux: 11101010
```



```
pointerFunctionsFileData.h pointerPrimitives.h pointerPrimitives.cpp* pointerFunctionsFileData.cpp < X
Pointer Logic File Data (Global Scope)
1 #include "pointerPrimitives.h"
2 #include "pointerFunctionsFileData.h"
3
4 int main ()
5 {
6     ifstream inp ("inpdata.tst"); //declare and initialize inp
7     ofstream out ("outdata.tst"); //declare and initialize out
8
9     int ii;
10    inp >> ii;
11    out << "All vector lengths are " << ii << " bits.\n";
12
13    char sel, oe;
14    char* aC = new char [ii+1];
15    char* bC = new char [ii+1];
16    char* wC = new char [ii+1];
17
18    while (inp >> aC >> bC >> sel >> oe)
19    {
20        out << "Inputs are a, b vectors and sel, oe bits: ";
21        out << aC << " " << bC << " " << sel << " " << oe << "\n";
22
23        mux8Std2T01 (aC, bC, wC, sel);
24        out << "Std Mux: " << wC << '\n';
25
26        mux8Tri2T01 (aC, bC, wC, sel, oe);
27        out << "Tri Mux: " << wC << '\n';
28    }
29
30 }
```

pointerFunctionsFileData.cpp

File handling

Calling the standard Multiplexer

Calling the Tri-state Multiplexer

Object Oriented Logic Modeling

Procedural Lang. for Hardware Modeling

Types and Operators for Logic Modeling

+ Basic Logic Simulation

- Enhanced Logic Simulation with Timing

Using *struct* for timing and logic

Gates that handle timing

Utility functions

Timing in logic structures

Overloading logical operators

Using Boolean expressions

+ More Functions for Wires and Gates

+ Inheritance in Logic Structures

+ Hierarchical Modeling of Digital Components
Summary

- Enhanced Logic Simulation with Timing

Using *struct* for Timing and Logic

○ Data structure

- is a group of data elements (known as members) grouped together under one name

The screenshot shows a code editor with two tabs: "timedFunctions.cpp" and "timedPrimitives.h". The "timedPrimitives.h" tab is active, displaying the following code:

```
Timed Logic Structs (Global Scope)
1 struct tlogic {
2     char logic;
3     int time;
4 };
5
6 tlogic and (tlogic a, tlogic b, int delay);
7 tlogic or (tlogic a, tlogic b, int delay);
8 tlogic not (tlogic a, int delay);
9 tlogic xor (tlogic a, tlogic b, int delay);
```

A callout bubble points to the first few lines of code with the text: "Structure to accommodate time as well as logic". Another callout bubble at the bottom left points to the "Ref. [1] - Data structures - P. 77-81" text.

Ref. [1] – Data structures - P. 77-81

- Enhanced Logic Simulation with Timing

Gates that Handle Timing

- ◎ A more accurate delay propagation requires the gate function to be aware of its previous output value

AND logic function
with timing

```
timedFunctions.cpp      timedPrimitives.h      timedPrimitives.cpp      timedFunctions.h
Timed Logic Structs      (Global Scope)
1 #include "timedPrimitives.h"
2
3 #tlogic and (tlogic a, tlogic b, int delay)
4 {
5     tlogic tl;
6     if ((a.logic=='0')||(b.logic=='0')) {
7         tl.logic = '0';
8         if (a.logic=='0') tl.time = a.time + delay;
9         else tl.time = b.time + delay;
10    }
11   else if ((a.logic=='1')&&(b.logic=='1')) {
12       tl.logic = '1';
13       if (a.time > b.time) tl.time = a.time + delay;
14       else tl.time = b.time + delay;
15   }
16   else {
17       tl.logic = 'X';
18       if (a.logic != '1') tl.time = a.time + delay;
19       else tl.time = b.time + delay;
20   };
21 }
22
23
24 #tlogic or (tlogic a, tlogic b, int delay){ ... }
25
26 #tlogic not (tlogic a, int delay){ ... }
27
28 #tlogic xor (tlogic a, tlogic b, int delay)
29 {
30     tlogic tl;
31     if (a.logic==b.logic) tl.logic = '0';
32     else tl.logic = '1';
33     if (a.time > b.time) tl.time = a.time + delay;
34     else tl.time = b.time + delay;
35     return tl;
36 }
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
```

- Enhanced Logic Simulation with Timing

Utility Functions

- The following table summarizes possible combinations of pointers and structure members

Expression	What is evaluated	Equivalent
a.b	Member b of object a	
a->b	Member b of object pointed by a	(*a).b
*a.b	Value pointed by member b of object a	*(a.b)

```

timedFunctions.cpp  X timedFunctions.h
(Global Scope)
1 #include "timedPrimitives.h"
2 #include "timedFunctions.h"
3 #include "characterPrimitives.h"
4
5 #define MAX(a,b) (a>b?a:b);
6 #define MIN(a,b) (a<b?a:b);
7
8 void getVect (string vectorName, int numBits, tlogic values[])
9 { //order according to bit significance
10    string valuesS;
11    int i, bits, delay;
12    cout << "Enter " << numBits << " bits of " << vectorName << ": ";
13    cin >> valuesS;
14    bits = MIN (valuesS.length(), numBits); // if fewer are entered
15    cout << "Enter vector delay: "; cin >> delay;
16    for (i=bits-1; i>=0; i--) {
17        values[i].logic = char(valuesS[bits-1-i]); // reverse bits
18        // values[i].time = delay; // This or two below are good
19        // ("values+i)).time = delay;
20        (values+i)->time = delay;
21    }
22
23
24 void putVect (string vectorName, int numBits, tlogic values[])
25 {
26    int i, delay;
27    delay = 0;
28    cout << vectorName << ": ";
29    for (i=numBits-1; i>=0; i--) {
30        cout << values[i].logic;
31        if (values[i].time > delay) delay=values[i].time;
32    }
33    cout << " AT " << delay << "\n";
34
35
36 void fullAdder (tlogic a, tlogic b, tlogic ci, tlogic& co, tlogic& sum){ ... }
37
38 void nBitAdder (tlogic a[], tlogic b[], tlogic c[], tlogic sum[], int bits){ ... }
39
40 void nBitAdder (tlogic* a, tlogic* b, tlogic* c, tlogic* sum, int bits, int worstDelay){ ... }

```

timedFunctions.cpp

The function gets a vector and its delay

Entered: 1011
valuesS: 1011
values: 1101

The arrow operator (->) is a dereference operator that is used with pointers to objects with members

The function puts a vector and its delay

- Enhanced Logic Simulation with Timing

Timing in Logic Structures

- Implementation of 1-bit full adder
- Implementation of n-bit ripple carry adder

Singular bit implementation of the FA function with timing

```
timedFunctions.cpp
35 void fullAdder (tlogic a, tlogic b, tlogic ci, tlogic& co, tlogic& sum)
36 {
37     tlogic axb, ab, abc;
38
39     axb = xor (a, b, 5);
40     ab = and (a, b, 3);
41     abc = and (axb, ci, 3);
42     co = or (ab, abc, 4);
43     sum = xor (axb, ci, 5);
44 }
45 
```

Using the timed logic functions

Multiple bit implementation of the FA function with timing

```
void nBitAdder (tlogic a[], tlogic b[], tlogic ci[], tlogic co[], tlogic sum[], int bits)
47 {
48     // assumes 0 is LSB
49     int i;
50     tlogic* c = new tlogic[bits+1];
51     c[0] = ci[0];
52     for (i = 0; i<bits; i++)
53     {
54         fullAdder(a[i], b[i], c[i], c[i+1], sum[i]);
55     }
56     co[0] = c[bits];
57 }
58 
```

The operator *new* is used to allocate dynamic memory

Ref. [1] – Dynamic Memory - P. 74-76

Using the timed singular bit FA

- Enhanced Logic Simulation with Timing

Timing in Logic Structures (cont.)

- Calling n-bit ripple carry adder function in *main* as a testbench

```
C:\WINDOWS\system32\cmd.exe
Enter number of bits of operations: 8
Enter 8 bits of aV: 10010011
Enter vector delay: 3
aV: 10010011 AT 3
Enter 8 bits of bV: 11110110
Enter vector delay: 5
bV: 11110110 AT 5
Enter 1 bits of ci: 1
Enter vector delay: 7
ci: 1 AT 7
aV: 10010011 AT 3
bV: 11110110 AT 5
ci: 1 AT 7
sumV: 10001010 AT 31
co: 1 AT 12
Enter 0 to exit: 0
Press any key to continue . . .
```

8-bit aV input
8-bit bV input
carry-in input
sumV and carry-out outputs

```
timedFunctions.cpp  X timedPrimitives.h      timedPrimitives.cpp      timedFunctions.h
Timed Logic Structs          (Global Scope)          timedFunctions.cpp

58
59 int main ()
60 {
61     tlogic *aV, *bV, *ci, *co, *sumV;
62
63     int bits, go(1);
64
65     while (go)
66     {
67         cout << "Enter number of bits of operations: "; cin >> bits;
68         aV = new tlogic[bits];
69         bV = new tlogic[bits];
70         ci = new tlogic[1];
71         co = new tlogic[1];
72         sumV = new tlogic[bits];
73
74         getVect ("aV", bits, aV); putVect ("aV", bits, aV);
75         getVect ("bV", bits, bV); putVect ("bV", bits, bV);
76         getVect ("ci", 1, ci); putVect ("ci", 1, ci);
77         cout << "\n";
78
79         nBitAdder (aV, bV, ci, co, sumV, bits); /* */
80
81         putVect (" aV", bits, aV); putVect (" bV", bits, bV);
82         putVect (" ci", 1, ci);
83         putVect ("sumV", bits, sumV); putVect (" co", 1, co);
84
85         delete [] aV;
86         delete [] bV;
87         delete [] ci;
88         delete [] co;
89         delete [] sumV;
90
91         cout << "\nEnter 0 to exit: "; cin >> go;
92
93     }
94 }
```

The operator *new* is used to allocate dynamic memory

Multiple bit char based carry ripple adder calculates all propagations

The operator *delete* is used to delete the memory allocated

- Enhanced Logic Simulation with Timing

Overloading Logical Operators

- Overloaded operators for the *tlogic struct* type

- Operation itself does not have a delay
- Operands have the delays and can be carried over to the output

timedFunctions.cpp timedFunctions.h timedOperators.h (Global Scope) timedOperators.h

```

1 struct tlogic {
2     char logic;
3     int time;
4 };
5
6 tlogic operator& (tlogic a, tlogic b);
7 tlogic operator| (tlogic a, tlogic b);
8 tlogic operator~ (tlogic a);
9 tlogic operator^ (tlogic a, tlogic b);
10
11

```

Structure to accommodate time as well as logic

Ref. [1] – Overloading operators - P. 95-97

timedFunctions.cpp timedFunctions.h timedOperators.h* (Global Scope) timedOperators.cpp

```

1 #include "timedOperators.h"
2
3 tlogic operator& (tlogic a, tlogic b){ ... }
4
5 tlogic operator| (tlogic a, tlogic b){ ... }
6
7 tlogic operator~ (tlogic a)
8 {
9     tlogic tl;
10    if (a.logic=='1') tl.logic = '0';
11    else if (a.logic=='0') tl.logic = '1';
12    else tl.logic=='X';
13    tl.time = a.time;
14    return tl;
15 }
16
17 tlogic operator^ (tlogic a, tlogic b)
18 {
19     tlogic tl;
20     if (a.logic==b.logic) tl.logic = '0';
21     else tl.logic = '1';
22     if (a.time > b.time) tl.time = a.time;
23     else tl.time = b.time;
24     return tl;
25 }
26
27

```

Consistent with the HDLs

- Enhanced Logic Simulation with Timing

Using Boolean Expressions

- Implementation of 1-bit full adder using Boolean expression
 - There are no inside wires to propagate delay values

```
C:\WINDOWS\system32\cmd.exe -> Enter number of bits of operations: 8
Enter 8 bits of aV: 10010011
Enter vector delay: 3
aV: 10010011 AT 3
Enter 8 bits of bV: 11110110
Enter vector delay: 5
bV: 11110110 AT 5
Enter 1 bits of ci: 1
Enter vector delay: 7
ci: 1 AT 7

aV: 10010011 AT 3
bV: 11110110 AT 5
ci: 1 AT 7
sumV: 10001010 AT 7
co: 1 AT 5

Enter 0 to exit: 0
Press any key to continue . . .
```

8-bit aV input
8-bit bV input
carry-in input
sumV and carry-out outputs (worst-case delay)

```
timedFunctions.cpp  X  timedFunctions.h  timedOperators.h*  timedOperators.cpp*
Timed Logic Overloading (Global Scope)
1 #include "timedOperators.h"
2 #include "timedFunctions.h"
3
4 #define MAX(a,b) (a>b?a:b);
5 #define MIN(a,b) (a<b?a:b);
6
7 void getVect (string vectorName, int numBits, tlogic values[])
22
23 void putVect (string vectorName, int numBits, tlogic values[])
34
35 void fullAdder(tlogic a, tlogic b, tlogic ci, tlogic& co, tlogic& sum)
36 {
37     co = (a & b) | (a & ci) | (b & ci);
38     sum = a ^ b ^ ci;
39 }
40
41 void nBitAdder(tlogic a[], tlogic b[], tlogic ci[], tlogic co[],
52 bits){ ... }
53
54 int main(){ ... }
```

The function gets/puts a vector and its delay

Object Oriented Logic Modeling

Procedural Lang. for Hardware Modeling

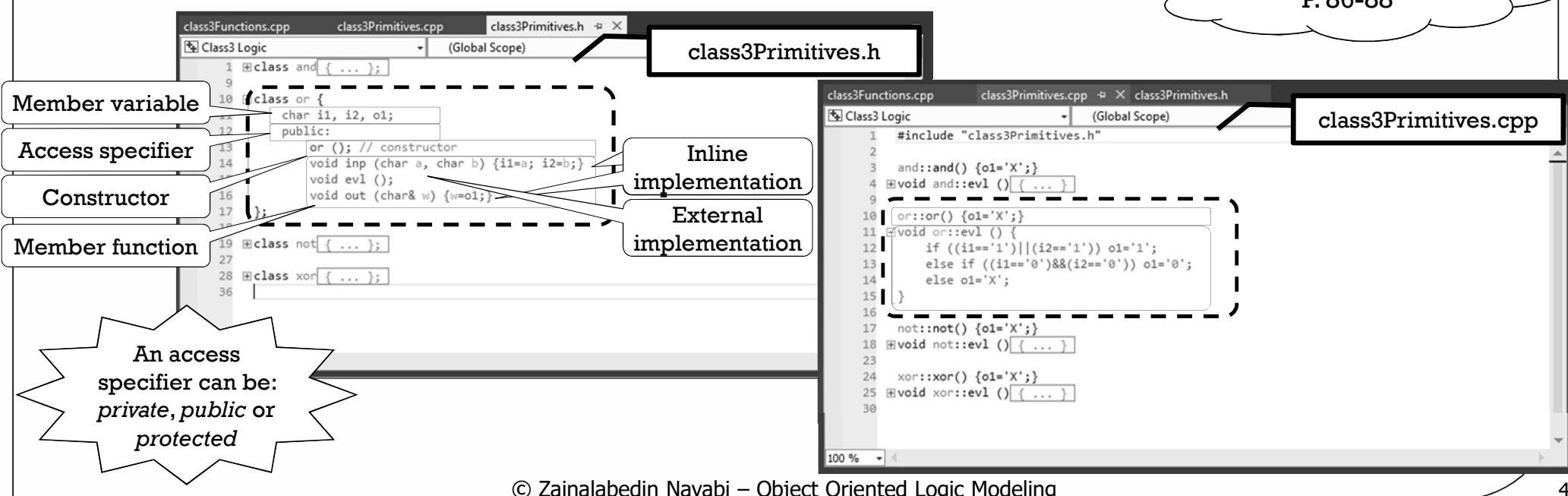
Types and Operators for Logic Modeling

- + Basic Logic Simulation
- + Enhanced Logic Simulation with Timing
- More Functions for Wires and Gates
 - Gate classes
 - Carrier generic modeling
 - Pointer-based logic classes
 - Gate classes with power and timing calculation
 - Wire and gate vectors
- + Inheritance in Logic Structures
- + Hierarchical Modeling of Digital Components
- Summary

– More Functions for Wires and Gates

Gate Classes

- A class is an expanded concept of a data structure: instead of holding only data, it can hold both data and functions
 - An object is an instantiation of a class



- More Functions for Wires and Gates

Gate Classes (cont.)

- Implementation of 1-bit full adder using gate classes

The gate classes

```

class3Functions.cpp  X class3Primitives.cpp  class3Primitives.h
Class3 Logic  (Global Scope)  class3Functions.cpp
1 #include "class3Primitives.h"
2 [ #include "class3Functions.h"
3
4 void fullAdder (char a, char b, char ci, char & co, char & sum)
{
    char axb, ab, abc;
    xor xor1, xor2;
    and and1, and2;
    or or1;

    xor1.inp(a, b);
    xor1.evl();
    xor1.out(axb);
    and1.inp(a, b);
    and1.evl();
    and1.out(ab);
    and2.inp(axb, ci);
    and2.evl();
    and2.out(abc);
    or1.inp(ab, abc);
    or1.evl();
    or1.out(co);
    xor2.inp(axb, ci);
    xor2.evl();
    xor2.out(sum);
}

```

class3Functions.cpp

```

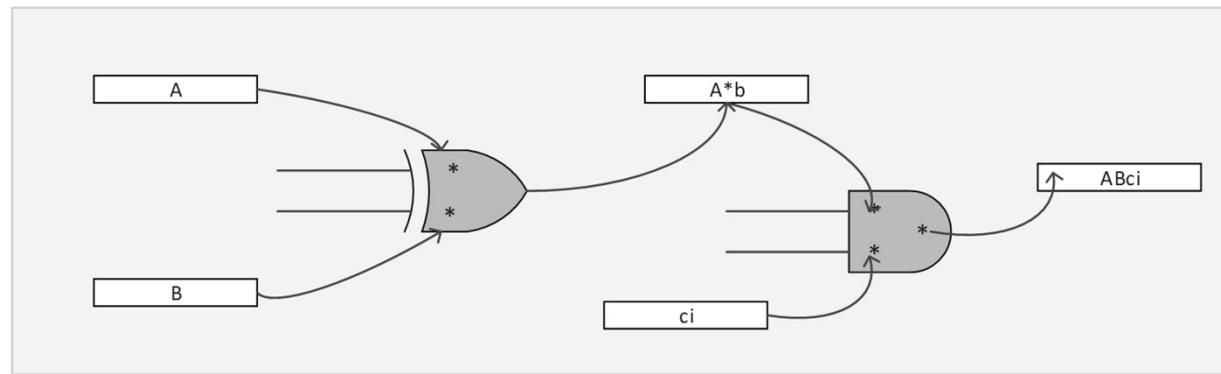
class3Functions.cpp  X class3Primitives.cpp  class3Primitives.h
Class3 Logic  (Global Scope)  class3Functions.cpp
1 // Class3 Logic
2 int main ()
3 {
4     char aC;
5     char bC;
6     char ciC;
7     char coC;
8     char sumC;
9
10    int ai;
11
12    do {
13
14        cout << "Enter a: ";
15        cin >> aC; cout << aC << '\n';
16
17        cout << "Enter b: ";
18        cin >> bC; cout << bC << '\n';
19
20        cout << "Enter ci: ";
21        cin >> ciC; cout << ciC << '\n';
22
23        // and (aC, bC, wC);
24        // cout << "and:" << wC << '\n';
25
26        // or (aC, bC, wC);
27        // cout << "or:" << wC << '\n';
28
29        fullAdder (aC, bC, ciC, coC, sumC);
30
31        cout << "Carry: " << coC << '\n';
32        cout << " Sum: " << sumC << '\n';
33
34        cout << "\n" << "Continue?"; cin >> ai;
35
36    } while (ai>0);
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

```

Calling the 1-bit full adder function in *main* as a testbench

- More Functions for Wires and Gates

Carrier Generic Modeling



– More Functions for Wires and Gates

Pointer Based Logic Classes

- *char* type pointer-based gate classes

- AND gate

The class do not hold values. Since the lines are just pointers, someone else has to declare them and allocate them.

- ✓ Constructor is to initialize variables or assign dynamic memory
 - ✓ Constructor function must have the same name as the class, and cannot have any return type

Destructor fulfills the opposite functionality. It is suitable to release the memory that the object was allocated

Ref. [1] –
Constructors and
destructors - P. 88-90

evl and *out* are combined and *evl* does both. Actually, since the outputs are pointers they will just be updated by *evl*. Every invocation of *evl* puts the internal output values on the *evl* return value

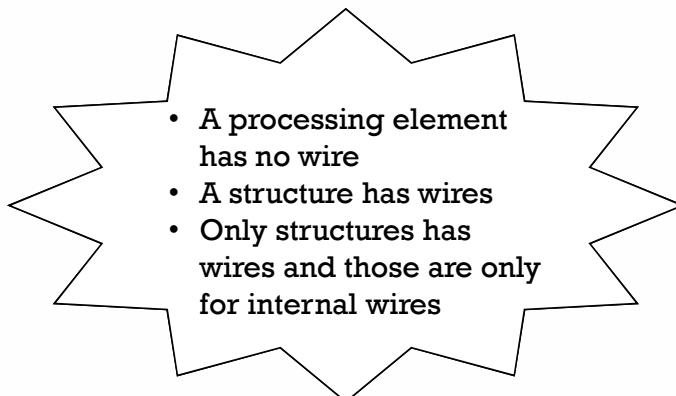
variables are of *char* type

- * Gates only process and points to wires
 - * Wires as holders of values and transmitters

- More Functions for Wires and Gates

Pointer Based Logic Classes

- *char* type pointer-based gate classes (cont.)
 - AND gate



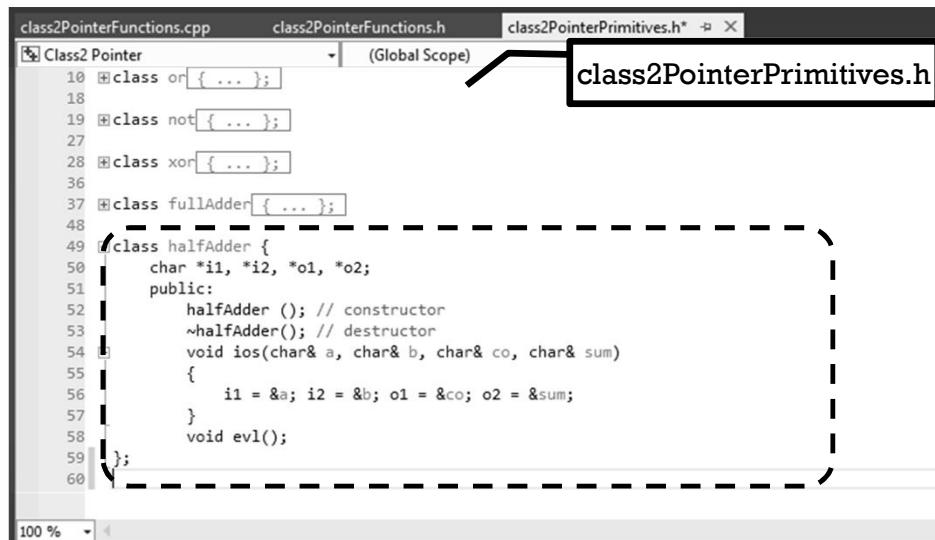
```
class2PointerFunctions.h      class2PointerPrimitives.h      class2PointerPrimitives.cpp
Class2 Pointer                (Global Scope)                  class2PointerPrimitives.cpp
1 #include "class2PointerPrimitives.h"
2 #include "class2Pointerfunctions.h"
3
4 and::and() {}
5 void and::evl () {
6     if ((*i1=='0')||(*i2=='0')) *o1='0';
7     else if ((*i1=='1')&&(*i2=='1')) *o1='1';
8     else *o1='X';
9 }
10
11 or::or() {}
12 void or::evl () { ... }
13
14 not::not() {}
15 void not::evl () { ... }
16
17 xor::xor() {}
18 void xor::evl () { ... }
19
20 fullAdder::fullAdder() {}
21 void fullAdder::evl () { ... }
22
23 halfAdder::halfAdder() {}
24 void halfAdder::evl () { ... }
```

- More Functions for Wires and Gates

Pointer Based Logic Classes

- *char* type pointer-based gate classes (cont.)

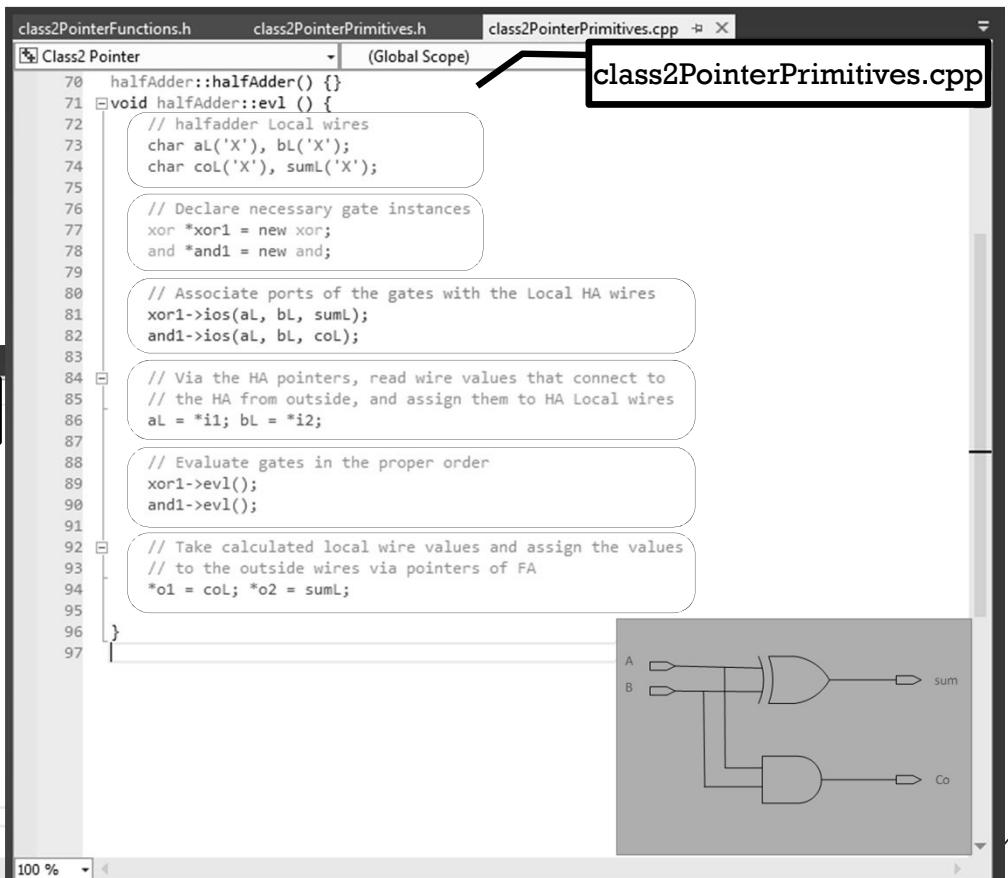
- Half adder
 - Higher-level structures also have gates that have no internal wires



```

class2PointerFunctions.cpp  class2PointerFunctions.h  class2PointerPrimitives.h* ✘
Class2 Pointer
10 #class or{ ... };
18
19 #class not{ ... };
27
28 #class xor{ ... };
36
37 #class fullAdder{ ... };
48
49 #class halfAdder {
50     char *i1, *i2, *o1, *o2;
51     public:
52         halfAdder(); // constructor
53         ~halfAdder(); // destructor
54         void ios(char& a, char& b, char& co, char& sum)
55         {
56             i1 = &a; i2 = &b; o1 = &co; o2 = &sum;
57         }
58         void evl();
59     };
60

```



```

class2PointerFunctions.h  class2PointerPrimitives.h  class2PointerPrimitives.cpp ✘
Class2 Pointer
70     halfAdder::halfAdder() {}
71     void halfAdder::evl () {
72         // halfadder Local wires
73         char al('X'), bl('X');
74         char col('X'), sumL('X');
75
76         // Declare necessary gate instances
77         xor *xor1 = new xor;
78         and *and1 = new and;
79
80         // Associate ports of the gates with the Local HA wires
81         xor1->ios(al, bl, sumL);
82         and1->ios(al, bl, col);
83
84         // Via the HA pointers, read wire values that connect to
85         // the HA from outside, and assign them to HA Local wires
86         al = *i1; bl = *i2;
87
88         // Evaluate gates in the proper order
89         xor1->evl();
90         and1->evl();
91
92         // Take calculated local wire values and assign the values
93         // to the outside wires via pointers of FA
94         *o1 = col; *o2 = sumL;
95     }
96 }

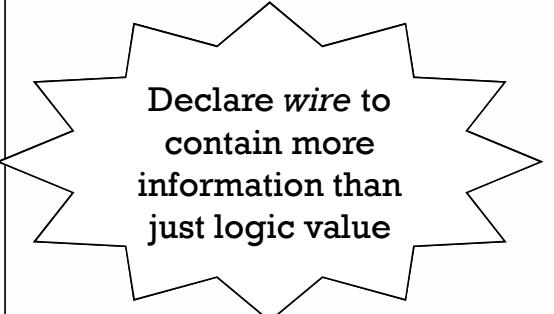
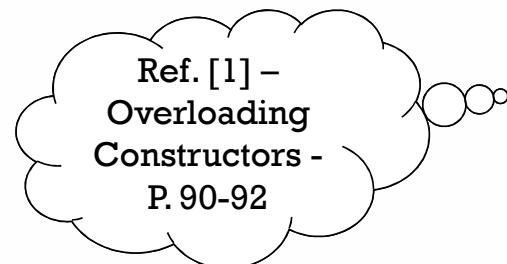
```

A logic diagram of a half adder is shown at the bottom right, consisting of two inputs (A and B) entering an XOR gate, whose output enters an AND gate along with input B. The outputs of the XOR and AND gates are summed to produce the sum and carry outputs respectively.

- More Functions for Wires and Gates

Gate Classes with Power and Timing Calculation

- wire type pointer-based logic classes
 - AND gate



Member variables:

- *EventTime* to propagate delay
- *ActivityCount* to carry power consumption

wire class has two constructors
One is for *value* and *eventTime*

Inline member functions:

- They have *put* and *get* for accessing their *value* and *eventTime*
- Wires have access function to *activityCount*

wire pointers

AND constructor just ties port pointers to wires

```

timedLogicFunctions.cpp      timedLogicUtilities.cpp      timedLogicPrimitives.h
Timed Logic Classes          (Global Scope)
1  int calculateEventTime(char lastValue, char newValue,
2                           int in1LastEvent, int in2LastEvent, int gateDelay, int lastEvent);
3
4  class wire {
5   public:
6     char value;
7     int eventTime;
8     int activityCount=0;
9   public:
10    wire(char c, int d) : value(c), eventTime(d) {}
11    wire();
12    void put(char a, int d) { value = a; eventTime = d; }
13    void get(char& a, int& d) { a = value; d = eventTime; }
14    int activity() { return activityCount; }
15  };
16
17  class and {
18   wire *i1, *i2, *o1;
19   int gateDelay, lastEvent;
20   char lastValue;
21   public:
22    and(wire& a, wire& b, wire& w, int d) :
23        i1(&a), i2(&b), o1(&w), gateDelay(d) {};
24    ~and();
25    void eval();
26  };
27
28  class or { ... };
29
30  class not { ... };
31
32  class xor { ... };
33
34  class dff_ar {
35   wire *D, *clk, *R, *Q;
36   int clkQDelay, rstQDelay;
37   int lastEvent; // last time output changed
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

- More Functions for Wires and Gates

Gate Classes with Power and Timing Calculation

- wire type pointer-based logic classes (cont.)
 - AND gate

If output has changed, the last event time on output is the larger of the inputs plus gate delay

Overloaded function

Logic part

Event part (timing)

Activity part (power)

Retain last event and last value

```

timedLogicUtilities.cpp      timedLogicUtilities.h      timedLogicFunctions.h
Timed Logic Classes          fullAdder               timedLogicPrimitives.h
timedLogicPrimitives.cpp

1 #include "timedLogicPrimitives.h"
2 #include "timedLogicFunctions.h"
3
4 #define MAX(a,b) ((a>b)?a:b)
5
6 int calculateEventTime(char lastValue, char newValue,
7   int in1LastEvent, int in2LastEvent, int gateDelay, int lastEvent){
8
9   if (lastValue == newValue)
10    return lastEvent;
11   else
12    return gateDelay + MAX (in1LastEvent, in2LastEvent);
13 }
14
15 int calculateEventTime(char lastValue, char newValue,
16   int in1LastEvent, int gateDelay, int lastEvent){
17
18   if (lastValue == newValue)
19    return lastEvent;
20   else
21    return gateDelay + in1LastEvent;
22 }
23
24 void and::evl () {
25
26   if ((i1->value == '0') || (i2->value == '0'))
27     o1->value = '0';
28   else if ((i1->value == '1') && (i2->value == '1'))
29     o1->value = '1';
30   else
31     o1->value='X';
32
33   o1->eventTime = calculateEventTime(lastValue, o1->value,
34                                     i1->eventTime, i2->eventTime, gateDelay, lastEvent);
35
36   o1->activityCount = i1->activityCount + i2->activityCount +
37   ((lastValue == o1->value) ? 0 : 1);
38
39   lastEvent = o1->eventTime;
40   lastValue = o1->value;
41
42   ...
43
44   ...
45
46   ...
47
48   ...
49
49
50
51
52
53
54
55
56
57
58
59
59
60
61
62
63
64
65
66
67
68
69
69
70
71
72
73
74
75
76
77
78
79
79
80
81
82
83
84
85
86
87
88
89
89
90
91
92
93
94
95
96
97
98
99
100 %

```

- More Functions for Wires and Gates

Gate Classes with Power and Timing Calculation

- wire type pointer-based logic classes (cont.)

- D-Flip Flop with Asynchronous Reset

The screenshot shows a code editor with two files open:

- timedLogicPrimitives.h**: Contains the declaration of a class `dff_ar` which represents a D-Flip Flop. It has private members `D`, `clk`, `R`, and `Q`, and public methods `dff_ar`, `~dff_ar`, and `evl`.
- timedLogicPrimitives.cpp**: Contains the implementation of the `dff_ar::evl` method. This method updates the state based on the inputs `D`, `clk`, and `R`, and calculates timing information using `calculateEventTime`. It also increments the activity count for the output `Q`.

A D-Flip Flop block diagram is overlaid on the code. The diagram shows a grey rectangle labeled "DFF" with three inputs: `D`, `clk`, and `reset`. The output is `Q`. A callout box labeled "Logic & event (timing) parts" points to the timing calculation code in the `evl` method. Another callout box labeled "Activity part (power)" points to the line where the activity count is updated.

```

timedLogicFunctions.cpp      timedLogicUtilities.cpp      timedLogicFunctions.h
Timed Logic Classes          Timed Logic Classes          Timed Logic Classes
( Global Scope )             ( Global Scope )             ( Global Scope )
58 class dff_ar {
59     wire *D, *clk, *R, *Q;
60     int clkQDelay, rstQDelay;
61     int lastEvent; // last time output changed
62     char lastValue;
63
64     public:
65         dff_ar(wire& d, wire& c, wire& r, wire& q, int dC, int dR) :
66             D(&d), clk(&c), R(&r), Q(&q), clkQDelay(dC), rstQDelay(dR) {};
67         ~dff_ar();
68         void evl();
69    };
100
101 void dff_ar::evl() {
102
103     if (R->value == '1') {
104         Q->value = '0';
105         Q->eventTime = calculateEventTime(lastValue, Q->value,
106                                         R->eventTime, rstQDelay, lastEvent);
107     }
108     else if (clk->value == 'P') {
109         Q->value = D->value;
110         Q->eventTime = calculateEventTime(lastValue, Q->value,
111                                         clk->eventTime, clkQDelay, lastEvent);
112     }
113
114     Q->activityCount = D->activityCount + 2 +
115                         ((lastValue == Q->value) ? 0 : 3);
116
117 }

```

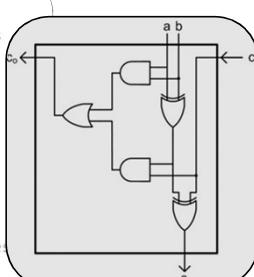
- More Functions for Wires and Gates

Gate Classes with Power and Timing Calculation

- wire type pointer-based logic classes (cont.)

- 1-bit full adder

Ref. [1] – Pointers to classes - P. 92-94



```
timedLogicFunctions.h      timedLogicPrimitives.cpp
Timed Logic Classes        Timed Logic Classes
120 void fullAdder::evl () {
121     // Via the FA pointers, read wire values that connect to
122     // the FA from outside, and assign them to FA Local wires
123     aL = *i1; bL = *i2; cL = *i3;
124
125     // Evaluate gates in the proper order
126     xor1->evl();
127     and1->evl();
128     and2->evl();
129     or1->evl();
130     xor2->evl();
131
132     // Take calculated local wire values and assign the values
133     // to the outside wires via pointers of FA
134     *o1 = oL; *o2 = sumL;
135
136 }
```

timedLogicFunctions.h timedLogicUtilities.cpp timedLogicPrimitives.h

Timed Logic Classes Timed Logic Classes Timed Logic Classes

```
71 // Structures based on above primitives begin here
72
73 class fullAdder {
74     wire *i1, *i2, *i3, *o1, *o2;
75
76     // Declare necessary gate instances
77     xor *xor1;
78     xor *xor2;
79     and *and1;
80     and *and2;
81     or *or1;
82
83     // fulladder Local wires
84     wire aL, bL, cL;
85     wire oL, sumL;
86     wire axbL, abL, abcL;
87
88 public:
89     fullAdder(wire& a, wire& b, wire& ci, wire& co, wire& sum) :
90         i1(&a), i2(&b), i3(&ci), o1(&co), o2(&sum),
91         aL('X', 0), bL('X', 0), cL('X', 0),
92         oL('X', 0), sumL('X', 0),
93         axbL('X', 0), abL('X', 0), abcL('X', 0) {
94
95         // Associate ports of the gates with the Local FA wires
96         xor1 = new xor(aL, bL, axbL, 5); // 5 is gate delay
97         xor2 = new xor(axbL, cL, sumL, 5);
98         and1 = new and(aL, bL, abL, 3);
99         and2 = new and(axbL, cL, abcL, 3);
100        or1 = new or(abL, abcL, oL, 3);
101    };
102    ~fullAdder();
103    void evl();
104
105 };
```

Full adder class definition declares gates and internal wires

Full adder constructor ties ports of the full adder to external wires, initialize internal wires, and then associate the ports of gates with the local wires

Destructors

It has evl() function that call gate classes in proper order

- More Functions for Wires and Gates

Gate Classes with Power and Timing Calculation

- *wire* type pointer-based logic classes (cont.)
 - Utility functions
 - For implementing this we need several utility functions for *inbit* and *outbit* to get time and value for wires

```
timedLogicUtilities.h  ✘ X  timedLogicFunctions.cpp      timedLogicPrimitives.h
Timed Logic Classes   (Global Scope)
1 #include "timedLogicPrimitives.h"
2 #include "timedLogicFunctions.h"
3
4 void inpBit(string, wire&);
5 void outBit(string, wire);

timedLogicUtilities.cpp  ✘ X
Timed Logic Classes   (Global Scope)
1 #include "timedLogicUtilities.h"
2
3 void inpBit(string wireName, wire& valtim) {
4     char value;
5     int time;
6     cout << "Enter value followed by @ time for " << wireName << ": ";
7     cin >> value; cin >> time;
8     valtim.put(value, time);
9 }
10
11 void outBit(string wireName, wire valtim) {
12     char value;
13     int time;
14     valtim.get(value, time);
15     cout << wireName << ":" << value << "@" << time << "\n";
16 }
```

- More Functions for Wires and Gates

Gate Classes with Power and Timing Calculation

- wire type pointer-based logic classes (cont.)
 - Serial adder

```
C:\WINDOWS\system32\cmd.exe
Enter value followed by @ time for FF Async Reset: 0 50
Carry output : 0 @ 761
Serial output: 1 @ 760
Feedback: 0 @ 1004

Continue? 1
Enter value followed by @ time for Serial input A: 0 550
Enter value followed by @ time for Serial input B: 0 1100
Enter value followed by @ time for FF Clock input: 0 1100
Enter value followed by @ time for FF Async Reset: 0 50
Carry output : 0 @ 761
Serial output: 0 @ 1110
Feedback: 0 @ 1004

Continue? 1
Enter value followed by @ time for Serial input A: 0 550
Enter value followed by @ time for Serial input B: 0 1100
Enter value followed by @ time for FF Clock input: P 1200
Enter value followed by @ time for FF Async Reset: 0 50
Carry output : 0 @ 761
Serial output: 0 @ 1110
Feedback: 0 @ 1204

Continue? 0
Activities: Sum: 101; Carry: 101; Feedback: 106
```

timedLogicUtilities.cpp timedLogicUtilities.h timedLogicFunctions.h
 Timed Logic Classes (Global Scope) timedLogicFunctions.cpp

```
30
31 int main ()
32 {
33     wire A, B;
34     wire clk, rst, fb('X',0);
35     wire sum, carry;
36
37     fullAdder *FA = new fullAdder(A, B, fb, carry, sum);
38     dff_ar *FF = new dff_ar(carry, clk, rst, fb, 4, 6);
39
40     int ai=1;
41
42     do {
43         inpBit("Serial input A", A);
44         inpBit("Serial input B", B);
45         inpBit("FF Clock input", clk);
46         inpBit("FF Async Reset", rst);
47
48         FA->evl();
49
50         outBit("Carry output ", carry);
51         outBit("Serial output", sum);
52
53         FF->evl();
54
55         outBit("Feedback", fb);
56
57         cout << "\n" << "Continue? "; cin >> ai;
58     } while (ai>0);
59
60     cout << "Activities: Sum: " << sum.activity()
61             << "; Carry: " << carry.activity()
62             << "; Feedback: " << fb.activity() << '\n';
63
64
65 }
```

– More Functions for Wires and Gates

Wire and Gate Vectors

- *wireV* type pointer-based logic classes

- AND gate

timedVectorLogicFunctions.cpp timedVectorLogicUtilities.h

Timed Vector Logic Classes

```
133
134 class wireV {
135 public:
136     char* value;
137     int n; //7/8bits
138     int eventTime;
139     int activityCount = 0;
140 public:
141     wireV(string v, int d, int size);
142     wireV(){};
143     ~wireV(){};
144     void put(string a, int d);
145     void get(string& a, int& d);
146     int activity() { return activityCount; }
147 };
148
149 class andV {
150     wireV *i1, *i2, *o1;
151     int gateDelay, lastEvent;
152     char* lastValue;
153 public:
154     andV(wireV& a, wireV& b, wireV& w, int d) :
155         i1(&a), i2(&b), o1(&w), gateDelay(d) {
156         lastValue = new char[w.n+1];
157     };
158     ~andV(){};
159     void evl();
160 };
161 
```

Main difference with wire

wireV has an eventTime and an activityCount for a group of wires. This model is not accurate since all individual wires are treated the same

timedVectorLogicUtilities.cpp timedVectorLogicUtilities.h

Timed Vector Logic Classes orV

```
156 E wireV::wireV(string v, int d, int size) : eventTime(d), n(size) {  
157     int i;  
158     value = new char[n + 1];  
159     v.resize(n, 'X');  
160     for (i = 0; i < n; i++){ *(i + value) = v.at(i); };  
161     *(n + value) = '\0';  
162 }  
163 void wireV::put(string a, int d){  
164     int i;  
165     eventTime = d;  
166     a.resize(n, '0');  
167     for (i = 0; i < n; i++){ *(i + value) = a.at(i); };  
168 }  
169 void wireV::get(string& a, int& d){  
170     int i;  
171     d = eventTime;  
172     a.resize(n, '0');  
173     for (i = 0; i < n; i++){ a.at(i) = *(i + value); };  
174 }  
175  
176 void andV::eval() {  
177     int i = 0;  
178  
179     while (i1->value[i] != '\0'){  
180         if ((i1->value[i]) == '0') || ((i2->value[i]) == '0')  
181             o1->value[i] = '0';  
182         else if ((i1->value[i] == '1') && (i2->value[i] == '1'))  
183             o1->value[i] = '1';  
184         else  
185             o1->value[i] = 'X';  
186         i++;  
187     };  
188 }  
189  
100 %
```

Adding '\0' to make it compatible with the c++ predefined string class

Since they are clusters, individual delay and power do not apply

- More Functions for Wires and Gates

Wire and Gate Vectors

- *wireV* type pointer-based logic classes (cont.)
 - Utility functions

Utility for individual wires

Utility for vector wires

The screenshot shows a code editor with two tabs: "timedVectorLogicUtilities.cpp" and "timedVectorLogicPrimitives.h". The "timedVectorLogicPrimitives.h" tab is active and contains the following C++ code:

```
#include "timedVectorLogicUtilities.h"

void inpBit(string wireName, wire& valtim) {
    char value;
    int time;
    cout << "Enter value followed by @ time for " << wireName << ": ";
    cin >> value; cin >> time;
    valtim.put(value, time);
}

void outBit(string wireName, wire valtim) {
    char value;
    int time;
    valtim.get(value, time);
    cout << wireName << ":" << value << "@" << time << "\n";
}

void inpBit(string wireName, wireV& valtim) {
    string values;
    int time;
    cout << "Enter value followed by @ time for " << wireName << ": ";
    cin >> values; cin >> time;
    valtim.put(values, time);
}

void outBit(string wireName, wireV valtim) {
    string value;
    int time;
    valtim.get(value, time);
    cout << wireName << ":" << value << "@" << time << "\n";
}
```

- More Functions for Wires and Gates

Wire and Gate Vectors

- *wireV* type pointer-based logic classes (cont.)
 - Calling the gate functions in *main* as a testbench

```
C:\WINDOWS\system32\cmd.exe
Enter value followed by @ time for Wire a: 11001101 3
Enter value followed by @ time for Wire b: 10011100 5
Wire w AND result: 10001100 @ 0
Wire w OR result: 11011111 @ 0
Continue?
```

```
timedVectorLogicUtilities.cpp          timedVectorLogicPrimitives.cpp
Timed Vector Logic Classes             (Global Scope)
timedVectorLogicFunctions.cpp

64
65 int main ()
66 {
67     wireV aWV("10101111", 0, 8), bWV("00110000", 0, 8), cWV("00001111", 0, 8),
68     wWV("00001111", 0, 8), yWV("XXXX0000", 0, 8);
69
70     andV *AND = new andV(aWV, bWV, wWV, 0);
71     orV *OR = new orV(aWV, bWV, yWV, 0);
72
73     int ai;
74
75     do {
76         inpBit("Wire a", aWV);
77         inpBit("Wire b", bWV);
78
79         AND->evl();
80         OR->evl();
81
82         outBit("Wire w AND result", wWV);
83         outBit("Wire w OR result", yWV);
84
85         cout << "\n" << "Continue? "; cin >> ai;
86
87     } while (ai>0);
88
89
90 }
```

Object Oriented Logic Modeling

Procedural Lang. for Hardware Modeling

Types and Operators for Logic Modeling

- + Basic Logic Simulation
- + Enhanced Logic Simulation with Timing
- + More Functions for Wires and Gates
- Inheritance in Logic Structures
 - A generic gate definition
 - Gates to include timing
 - Building structures from objects
- + Hierarchical Modeling of Digital Components
- Summary

– Inheritance in Logic Structures

A generic gate definition

○ Inheritance-based logic classes

Ref. [1] – Inheritance between classes - P. 101-106

Accessible by gate classes that
are inherited from gates

Different constructors for 2-input and 1-input gates and no initialization

External member functions:

- `evl()` is needed for each gate.
Each gate instance uses its own
`evl()` function. Wires have access
function to `activityCount`
 - Timing activity functions for 2
and 1 input gates

Inheritance allows to create classes which are derived from other classes, so that they include some of its "parent's" members, plus its own

inheritedLogicClassesFunctions.cpp inheritedLogicClassPrimitives.h

Logic Class Inheritance

```
16 class gates {
17     protected:
18     wire *i1, *i2, *o1;
19     int gateDelay, lastEvent;
20     char lastValue;
21 public:
22     gates(wire& a, wire& w, int d) :
23         i1(&a), o1(&w), gateDelay(d) {}
24     gates(wire& a, wire& b, wire& w, int d) :
25         i1(&a), i2(&b), o1(&w), gateDelay(d) {}
26     gates();
27     ~gates();
28     void evl();
29     void timingActivity2();
30     void timingActivity1();
31 };
32
33 class and: public gates {
34 public:
35     and(wire& a, wire& b, wire& w, int d) : gates(a, b, w, d) {}
36     ~and();
37     void evl();
38 };
39
40 class or: public gates {
41 public:
42     or(wire& a, wire& b, wire& w, int d) : gates(a, b, w, d) {}
43     ~or();
44     void evl();
45 };
46
47 class not: public gates {
48 public:
49     not(wire& a, wire& w, int d) : gates(a, w, d) {}
50     ~not();
51     // void evl() does not exist, will use gates::evl()
52 };
```

InheritedLogicClassPrimitives.h

All gates are inherited from the *gates* class

An inherited class that does not have its own *evl()* can depend on the base class

All gates are
inherited from
the *gates* class

An inherited class that does not have its own eval() can depend on the base class

- Inheritance in Logic Structures

A generic gate definition

- Inheritance-based logic classes
(cont.)

The screenshot shows a code editor with two tabs: `inheritedLogicClassesFunctions.cpp` and `InheritedLogicClassPrimitives.cpp`. The `InheritedLogicClassPrimitives.cpp` tab is active, showing the following code:

```
inheritedLogicClassesFunctions.cpp      InheritedLogicClassPrimitives.cpp
Logic Class Inheritance
xor
4 #define MAX(a,b) ((a>b)?a:b)
5
6 int calculateEventTime(char lastValue, char newValue,
7     int in1LastEvent, int in2LastEvent, int gateDelay, int lastEvent){
8
9     if (lastValue == newValue)
10        return lastEvent;
11    else
12        return gateDelay + MAX (in1LastEvent, in2LastEvent);
13 }
14
15 int calculateEventTime(char lastValue, char newValue,
16     int in1LastEvent, int gateDelay, int lastEvent){ ... }
17
18 void gates::evl() { // inverts its input 1
19
20     if (i1->value == '0')
21         o1->value = '1';
22     else if (i1->value == '1')
23         o1->value = '0';
24     else
25         o1->value = 'X';
26
27     gates::timingActivity1();
28 }
29
30 void gates::timingActivity2() {
31
32     o1->eventTime = calculateEventTime(lastValue, o1->value,
33                                         i1->eventTime, i2->eventTime, gateDelay, lastEvent);
34
35     o1->activityCount = i1->activityCount + i2->activityCount +
36     ((lastValue == o1->value) ? 0 : 1);
37
38     lastEvent = o1->eventTime;
39     lastValue = o1->value;
40
41     gates::timingActivity1(); ...
42 }
```

A dashed rectangular box highlights the timing activity code starting from line 29. An arrow points from the word "xor" in the first tab to the start of the code in the second tab.

- Inheritance in Logic Structures

Gates to include timing

○ Inheritance-based logic classes (cont.)

- AND gate
- OR gate
- NOT gate
- XOR gate

The screenshot shows a code editor window with two tabs: "inheritedLogicClassesFunctions.cpp" and "InheritedLogicClassPrimitives.cpp". The "inheritedLogicClassesFunctions.cpp" tab is active, displaying C++ code for logic gates. The "InheritedLogicClassPrimitives.cpp" tab is visible in the background.

```
inheritedLogicClassesFunctions.cpp
inheritedLogicClasses
Logic Class Inheritance -> dff_ar
InheritedLogicClassPrimitives.cpp

57 void and::evl() {
58
59     if ((i1->value == '0') || (i2->value == '0'))
60         o1->value = '0';
61     else if ((i1->value == '1') && (i2->value == '1'))
62         o1->value = '1';
63     else
64         o1->value = 'X';
65
66     gates::timingActivity2();
67 }
68
69 void or::evl() { ... }
70
71 /*void not::evl () { // uses gates::evl(); }*/
72
73 void xor::evl () {
74
75     if ((i1->value == 'X') || (i2->value == 'X') ||
76         (i1->value == 'Z') || (i2->value == 'Z'))
77         o1->value = 'X';
78     else if (i1->value==i2->value)
79         o1->value='0';
80     else
81         o1->value='1';
82
83     gates::timingActivity2();
84 }
```

Annotations on the right side of the code editor:

- A callout bubble points to the first three methods (and::evl, or::evl, xor::evl) with the text: "Calculate output value and call timing activity at gates".
- A callout bubble points to the commented-out not::evl method with the text: "No evl() for not to use that of gates".

- Inheritance in Logic Structures

Building Structures from Objects

• 1-bit full adder

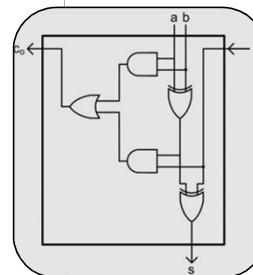
inheritedLogicClassesFunctions.cpp

```

116 void fullAdder::evl () {
117     // Via the FA pointers, read wire values that connect to
118     // the FA from outside, and assign them to FA Local wires
119     al = *i1; bl = *i2; cil = *i3;
120     and1->timingActivity1();
121     // Evaluate gates in the proper order
122     xor1->evl();
123     and1->evl();
124     and2->evl();
125     or1->evl();
126     xor2->evl();
127
128     // Take calculated local wire values and assign the values
129     // to the outside wires via pointers of FA
130     *o1 = col; *o2 = sumL;
131
132
133
134

```

100 %



inheritedLogicClassPrimitives.cpp

inheritedLogicClassesFunctions.cpp

inheritedLogicClassPrimitives.h

```

76 class fullAdder {
77     wire *i1, *i2, *i3, *o1, *o2;
78
79     // Declare necessary gate instances
80     xor *xor1;
81     xor *xor2;
82     and *and1;
83     and *and2;
84     or *or1;
85
86     // fulladder Local wires
87     wire aL, bL, ciL;
88     wire col, sumL;
89     wire axbL, abL, abcL;
90
91 public:
92     fullAdder(wire& a, wire& b, wire& ci, wire& co, wire& sum) :
93         i1(&a), i2(&b), i3(&ci), o1(&co), o2(&sum),
94         aL('X', 0), bL('X', 0), ciL('X', 0),
95         col('X', 0), sumL('X', 0),
96         axbL('X', 0), abL('X', 0), abcL('X', 0) {
97
98     // Associate ports of the gates with the Local FA wires
99     xor1 = new xor(aL, bL, axbL, 5); // 5 is gate delay
100    xor2 = new xor(axbL, ciL, sumL, 5);
101    and1 = new and(aL, bL, abL, 3);
102    and2 = new and(axbL, ciL, abcL, 3);
103    or1 = new or(abL, abcL, col, 3);
104 }
105 ~fullAdder();
106 void evl();
107 };

```

inheritedLogicClassPrimitives.h

Full adder class definition declares gates and internal wires

Full adder constructor ties ports of the full adder to external wires, initialize internal wires, and then associate the ports of gates with the local wires

Full adder uses inherited gates

Destructors

It has evl() function that call gate classes in proper order

Object Oriented Logic Modeling

Procedural Lang. for Hardware Modeling

Types and Operators for Logic Modeling

- + Basic Logic Simulation
- + Enhanced Logic Simulation with Timing
- + More Functions for Wires and Gates
- + Inheritance in Logic Structures
- Hierarchical Modeling of Digital Components

Wire functionalities

Gate functionalities

Polymorphic gate base

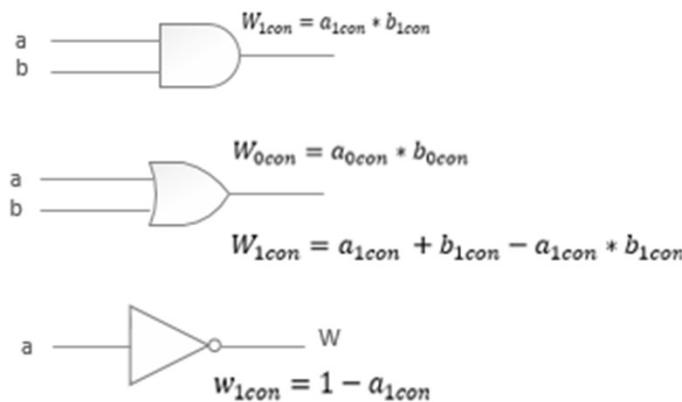
Flip flop description hierarchies

Summary

- Hierarchical Modeling of Digital Components

Wire Functionality

- Logic Testability Analysis



- Hierarchical Modeling of Digital Components

Wire Functionality

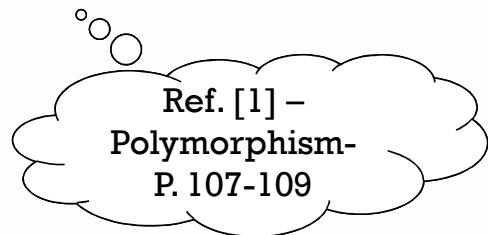
- wire class has wire identifier and static number of wires

```
polymorphismLogicClassesFunctions.h
Logic Class Polymorphism (Glob)
2
3 class wire {
4 protected:
5     static int numberOfWires;
6 public:
7     char value;
8     int eventTime;
9     int activityCount = 0;
10    float controlability = 0.5;
11 public:
12     int wireIdentifier;
13     wire(char c, int d) : value(c), eventTime(d) {
14         wireIdentifier = numberOfWires;
15         numberOfWires++;
16     }
17     wire();
18     void put(char a, int d) { value = a; eventTime = d; }
19     void get(char& a, int& d) { a = value; d = eventTime; }
20     int activity() { return activityCount; }
21 };
22
100 %
```

- Hierarchical Modeling of Digital Components

Gate Functionality

- A class that declares or inherits a virtual function is called a polymorphic class



Gates constructor assigns an id and increments the gate count

Overloading Constructor

Virtual member functions

PolymorphismLogicClassesPrimitives.h

```

polymorphismLogicClassesFunctions.h
Logic Class Polymorphism
22
23 class gates {
24 protected:
25     wire *i1, *i2, *o1;
26     int gateDelay, lastEvent;
27     char lastValue;
28
29     void timingActivity2();
30     void timingActivity1();
31     static int numberOFGates;
32 public:
33     int gateIdentifier;
34     float outputControlability = 1.0;
35     gates(wire& a, wire& w, int d) :
36         i1(&a), i2(&w), o1(&w), gateDelay(d) {
37         gateIdentifier = numberOFGates;
38         numberOFGates++;
39     }
40     gates(wire& a, wire& b, wire& w, int d) :
41         i1(&a), i2(&b), o1(&w), gateDelay(d) {
42         gateIdentifier = numberOFGates;
43         numberOFGates++;
44     }
45     gates(){}
46     ~gates(){}
47     virtual void eval();
48     virtual void prob(){}
49 };
50

```

Virtual can be overwritten by classes that inherit from it. If not overwritten, the same eval() of gates will be used for an inherited class

- Hierarchical Modeling of Digital Components

Gate Functionality (cont.)

```
polymorphismLogicClassesPrimitives.h
logic Class Polymorphism
{
    int wire::numberOfWires = 1;
    void gates::evl() { // puts input 1 on output
        o1->value = i1->value;
        gates::timingActivity1();
    }
    void gates::timingActivity2() {
        o1->eventTime = calculateEventTime(lastValue, o1->value,
                                             i1->eventTime, i2->eventTime, gateDelay, lastEvent);
        o1->activityCount = i1->activityCount + i2->activityCount +
        ((lastValue == o1->value) ? 0 : 1);
        lastEvent = o1->eventTime;
        lastValue = o1->value;
    }
    void gates::timingActivity1() {
        o1->eventTime = calculateEventTime(lastValue, o1->value,
                                             i1->eventTime, gateDelay, lastEvent);
        o1->activityCount = i1->activityCount + ((lastValue == o1->value)?0:1);
        lastEvent = o1->eventTime;
        lastValue = o1->value;
    }
    int gates::numberOfGates=1;
    float getProb(GATES* GATE){
        return GATE->outputControlability;
    }
    void and::evl() {
        if ((i1->value == '0') || (i2->value == '0'))
    }
}
```

PolymorphismLogicClassesPrimitives.cpp

Static initialization must be done

Like a one input buffer

Timing activity functions for 2 and 1 input gates

Static initialization must be done

© Zainalabedin Navabi –

- Hierarchical Modeling of Digital Components

Polymorphic Gate Base

- The logic components are inherited from the *gates* class

PolymorphismLogicClassesPrimitives.h

```

53 class and: public gates {
54     public:
55         and(wire& a, wire& b, wire& w, int d) : gates(a, b, w, d) {}
56         ~and();
57         void evl();
58         void prob();
59     };
60
61 class or: public gates {
62     public:
63         or(wire& a, wire& b, wire& w, int d) : gates(a, b, w, d) {}
64         ~or();
65         void evl();
66         void prob();
67     };
68
69 class not: public gates {
70     public:
71         not(wire& a, wire& w, int d) : gates(a, w, d) {}
72         ~not();
73         void evl();
74         void prob();
75     };
76
77 class xor: public gates {
78     public:
79         xor(wire& a, wire& b, wire& w, int d) : gates(a, b, w, d) {}
80         ~xor();
81         void evl();
82         void prob();
83     };

```

PolymorphismLogicClassesPrimitives.cpp

```

56 void and::evl() {
57
58     if ((i1->value == '0') || (i2->value == '0'))
59         o1->value = '0';
60     else if ((i1->value == '1') && (i2->value == '1'))
61         o1->value = '1';
62     else
63         o1->value = 'X';
64
65     gates::timingActivity2();
66 }
67
68 void and::prob() {
69     outputControlability = i1->controlability * i2->controlability;
70     o1->controlability = outputControlability;
71 }
72
73 void or::evl() { ... }
74 void or::prob() { ... }
75
76 void not::evl() { ... }
77 void not::prob() { ... }
78
79 void xor::evl() { ... }
80 void xor::prob() { ... }
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124

```

uses the constructor of the *gates* class

declares member functions to overwrite *evl()* and *prob()* of the *gates* class

Overwriting the virtual functions of the *gates* class

- Hierarchical Modeling of Digital Components

Polymorphic Gate Base (cont.)

- Overloading `evl()` function

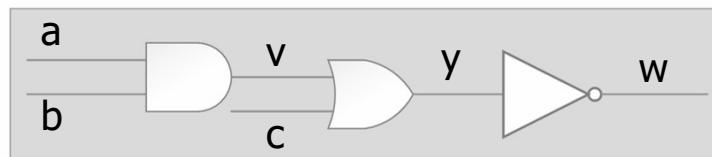
```
57 float evl(gates* GATE){  
58     GATE->evl();  
59     return GATE->outputControlability;  
60 }
```

PolymorphismLogicClassesPrimitives.cpp

- Hierarchical Modeling of Digital Components

Polymorphic Gate Base (cont.)

- Calling inheritance-based logic functions in *main* as a testbench



Pointers to base class (gates)

Calling the overloaded
evl() function

```

polymorphismLogic...ssesPrimitives.cpp
Logic Class Polymorphism
int main()
{
    wire a('0', 3), b('1', 5), c('X', 0);
    wire v('0', 3), w('0', 3), y('1', 5);

    gates *NOT = new not(y, w, 5);
    gates *AND = new and(a, b, v, 7);
    gates *OR1 = new or(v, c, y, 6);

    AND->prob();
    OR1->prob();
    NOT->prob();

    int ai; int time = 0;

    do {
        inpBit("Wire a", a, time);
        inpBit("Wire b", b, time);
        inpBit("Wire c", c, time);

        cout << evl(AND) << " :AND\n";
        cout << evl(OR1) << " :OR1\n";
        cout << evl(NOT) << " :NOT\n";

        outBit("AOI output: ", w);
        cout << "AOI output activity count: " << w.activity() << '\n';

        time += 17;
        cout << "\n" << "Continue? "; cin >> ai; cout << "\n";
    } while (ai>0);
}

/*
int main()
{
}

```

- Hierarchical Modeling of Digital Components

Flip Flop Description Hierarchies

- A pure virtual function is a virtual function for which we don't have implementation, we only declare it. It is declared by assigning 0 in the declaration
- An abstract class is a class which have at least one pure virtual function

○
○
○

Ref. [1] – Abstract
base classes -
P. 109-112

```
polymorphismLogicClassesPrimitives.h*  Logic Class Polymorphism
84
85 class flipflop {
protected:
    wire *D, *clk, *rst, *cen, *Q;
    int clkQDelay;
    int rstQDelay;
    int lastEvent; // last time output changed
    char lastValue;
    bool containsReset = false;
    float clockControllability = 0.5;
    static int numberOfflips;
public:
    int flipflopIdentifier;
    float outputControllability = 1.0;
    flipflop(wire& d, wire& c, wire& q, int dc) :
        D(&d), clk(&c), Q(&q), clkQDelay(dc) {
        flipflopIdentifier = numberOfflips;
        numberOfflips++;
    };
    ~flipflop(){}
    virtual void evl() = 0;
    virtual void prob() = 0;
    virtual void init(float, char) = 0;
};
```

- Hierarchical Modeling of Digital Components

Flip Flop Description Hierarchies

- DFF class

- is inherited from *flipflop* class

polymorphismLogicClassesPrimitives.h

Logic Class Polymorphism

```
109 class DFF : public flipflop {
110 public:
111     DFF(wire& d, wire& c, wire& q, int dc) : flipflop(d, c, q, dc)
112     { containsReset = false; }
113     ~DFF(){}
114     virtual void evl();
115     virtual void prob();
116     virtual void init(float, char);
117 };
```

First level derivation

polymorphismLogicClassesPrimitives.cpp

Logic Class Polymorphism

```
125     int flipflop::numberOfFlipflops = 1;
126
127     void DFF::evl() {
128         char valueToLoad = '0';
129
130         if (!containsReset) valueToLoad = D->value;
131         else valueToLoad = (rst->value == '1') ? '0' : D->value;
132
133         if (clk->value == 'P') {
134             Q->value = valueToLoad;
135             Q->eventTime = calculateEventTime(lastValue, Q->value,
136                                              clk->eventTime, clkQDelay, lastEvent);
137         }
138
139         Q->eventTime = calculateEventTime(lastValue, Q->value,
140                                           clk->eventTime, clkQDelay, lastEvent);
141
142         Q->activityCount = (D->activityCount + clk->activityCount) * 2 +
143                             ((lastValue == Q->value) ? 0 : 3);
144
145         lastEvent = Q->eventTime;
146         lastValue = Q->value;
147     }
148
149     void DFF::prob(){
150         outputControlability = D->controlability * clockControlability;
151         Q->controlability = outputControlability;
152     }
153     void DFF::init(float clkCon, char iniOut) {
154         clockControlability = clkCon; Q->value = iniOut;
155     }
156 }
```

- Hierarchical Modeling of Digital Components

Flip Flop Description Hierarchies

- DFF with Synchronous Reset (*DFFsR*) class
 - is inherited from *DFF* class
- DFF with Synchronous Reset and Enable (*DFFsRE*) class
 - is inherited from *DFFsR* class

The diagram illustrates the inheritance hierarchy between two files: **PolymorphismLogicClassesPrimitives.h** and **PolymorphismLogicClassesPrimitives.cpp**.

PolymorphismLogicClassesPrimitives.h (Left):

- Second level derivation:** A callout points to the `DFFsR` class definition.
- No evl(), so uses the one of DFF:** A callout points to the `virtual void evl();` declaration.
- Third level derivation:** A callout points to the `DFFsRE` class definition.
- Extra arguments will be handled by flip flop:** A callout points to the constructor arguments of the `DFFsRE` class.

PolymorphismLogicClassesPrimitives.cpp (Right):

- DFFsRE calls DFFsR when value is one:** A callout points to the `void DFFsRE::evl()` method, which calls `DFFsR::evl()` if the enable value is '1'.

```

polymorphismLogic...ssesPrimitives.cpp
Logic Class Polymorphism
120 class DFFsR : public DFF {
121 public:
122     DFFsR(wire& d, wire& c, wire& r, wire& q, int dc, int dr) : DFF(d, c, q, dc) {
123         containsReset = true;
124         rst = &r;
125         rstQDelay = dr;
126     };
127     ~DFFsR(){}
128     virtual void prob();
129 };
130
131 class DFFsRE : public DFFsR {
132 public:
133     DFFsRE(wire& d, wire& c, wire& r, wire& e,
134             wire& q, int dc, int dr) : DFFsR(d, c, r, q, dc, dr) {
135         cen = &e;
136     };
137     ~DFFsRE(){}
138     virtual void evl();
139 };
140
100 %
  
```

```

PolymorphismLogicClassesPrimitives.h*
Logic Class Polymorphism
155
156     void DFFsR::prob(){
157         outputControlability = (D->controlability + rst->controlability -
158             D->controlability * rst->controlability ) *
159             clockControlability;
160         Q->controlability = outputControlability;
161     }
162
163     void DFFsRE::evl() {
164         if (en->value == '1') DFFsR::evl();
165     }
  
```

- Hierarchical Modeling of Digital Components

Flip Flop Description Hierarchies

⦿ Utility Functions

The screenshot shows a code editor window with the title "PolymorphismLogicClassesPrimitives.cpp". The code is written in C++ and defines several utility functions for handling bit values and logic values over time.

```
polymorphismLogic...ssesPrimitives.cpp [polymorphismLogic...ssesPrimitives.cpp]
Logic Class Polymorphism (G)
PolymorphismLogicClassesPrimitives.cpp

3 void inpBit(string wireName, wire& valtim) {
4     char value;
5     int time;
6     cout << "Enter value followed by @ time for " << wireName << ": ";
7     cin >> value; cin >> time;
8     valtim.put(value, time);
9 }
10
11 void inpBit(string wireName, wire& valtim, int time) {
12     char value;
13     cout << "For @ time " << time << ", enter logic value for " << wireName << ": ";
14     cin >> value;
15     valtim.put(value, time);
16 }
17
18 void outBit(string wireName, wire valtim) {
19     char value;
20     int time;
21     valtim.get(value, time);
22     cout << wireName << ":" << value << "@" << time << "\n";
23 }
24
25 void inpBit(string wireName, wireV& valtim) {
26     string value;
27     int time;
28     cout << "Enter value followed by @ time for " << wireName << ": ";
29     cin >> value; cin >> time;
30     valtim.put(value, time);
31 }
32
33 void outBit(string wireName, wireV valtim) {
34     string value;
35     int time;
36     valtim.get(value, time);
37     cout << wireName << ":" << value << "@" << time << "\n";
38 }
39 }
```

- Hierarchical Modeling of Digital Components

Flip Flop Description Hierarchies

PolymorphismLogicClassesFunctions.cpp

```

polymorphismLogicClassesUtilities.cpp
Logic Class Polymorphism

3 int main()
4 {
5     wire a('0', 3), b('1', 5), c('X', 0), clk('X', 0), rst('X', 0),
6         en('X', 0),
7         Q1('X', 0), Q2('X', 0), Q3('X', 0);
8     wire v('0', 3), w('0', 3), y('1', 5);
9
10    flipflop *FF1 = new DFF(a, clk, Q1, 401);
11    flipflop *FF2 = new DFFsR(a, clk, rst, Q2, 502, 6);
12    flipflop *FF3 = new DFFsRE(a, clk, rst, en, Q3, 603, 7);
13    FF1->init(float(0.37), '1');
14    FF2->init(float(0.37), '1');
15    FF3->init(float(0.37), '1');
16
17    gates *NOT = new not(y, w, 5);
18    gates *AND = new and(a, b, v, 7);
19    gates *OR1 = new or(v, c, y, 6);
20
21    AND->prob();
22    OR1->prob();
23    NOT->prob();
24    FF1->prob();
25    FF2->prob();
26    FF3->prob();
27
28    cout << "AND gate Id: " << AND->gateIdentifier << '\n';
29    cout << "OR1 gate Id: " << OR1->gateIdentifier << '\n';
30    cout << "NOT gate Id: " << NOT->gateIdentifier << "\n\n";
31
32    cout << "DFF2 output 1-probability: " << FF2->outputControlability << '\n';
33    cout << "DFF3 output 1-probability: " << FF3->outputControlability << "\n\n";
34
35    cout << "AOI output 1-probability: " << getProb(NOT) << '\n';
36    cout << "DFF1 output 1-probability: " << FF1->outputControlability << '\n';
37    cout << "DFF2 output 1-probability: " << FF2->outputControlability << '\n';
38    cout << "DFF3 output 1-probability: " << FF3->outputControlability << "\n\n";
39

```

Logic Class Polymorphism

```

polymorphismLogicClassesUtilities.cpp
Logic Class Polymorphism

39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
100 %

```

```

int ai; int time = 0;
do {
    inpBit("Wire a", a, time);
    inpBit("Wire b", b, time);
    inpBit("Wire c", c, time);
    inpBit("Clock input", clk, time);
    inpBit("Reset input", rst, time);
    inpBit("Enable input", en, time);

    AND->evl();
    OR1->evl();
    NOT->evl();
    FF1->evl();
    FF2->evl();
    FF3->evl();

    outBit("AOI output: ", w);
    outBit("DFF1 output: ", Q1);
    outBit("DFF2 output: ", Q2);
    outBit("DFF3 output: ", Q3);

    cout << "AOI output activity count: " << w.activity() << '\n';
    cout << "DFF1 output activity count: " << Q1.activity() << '\n';
    cout << "DFF2 output activity count: " << Q2.activity() << '\n';
    cout << "DFF3 output activity count: " << Q3.activity() << "\n\n";

    time += 17;
    cout << "\n" << "Continue? "; cin >> ai; cout << "\n";
} while (ai>0);
/*
int main()
{
    wire aW('0', 3), bW('1', 5), ciW('X', 0), coWF('X', 0);
}

```

main as a testbench

Summary

- Procedural Languages for Hardware Modeling
- Types and Operators for Logic Modeling
- Basic Logic Simulation
- Enhanced logic simulation with timing
- More Functions for Wires and Gates
- Inheritance in Logic Structures
- Hierarchical Modeling of Digital Components

References

- [1] J. Soulie, C++ Language Tutorial, 2007.
- [2] B. W. Kernighan, D. Ritchie, and D. M. Ritchie, The C Programming Language, 2 Edition, Prentice Hall PTR, 1988.
- [3] B.Stroustrup, The C++ Programming Language, 3 Edition, Addison-Wesley, 1997.

Copyright and Acknowledgment

- © 2015, Zainalabedin Navabi, System-Level Design and Modeling:
ESL Using C/C++, SystemC and TLM-2.0, ISBN-13: 978-1441986740,
ISBN-10: 144198674X
- Slides prepared by Hanieh Hashemi, ECE graduate student
- Slides updated by Nooshin Nosrati, ECE Ph.D. student