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

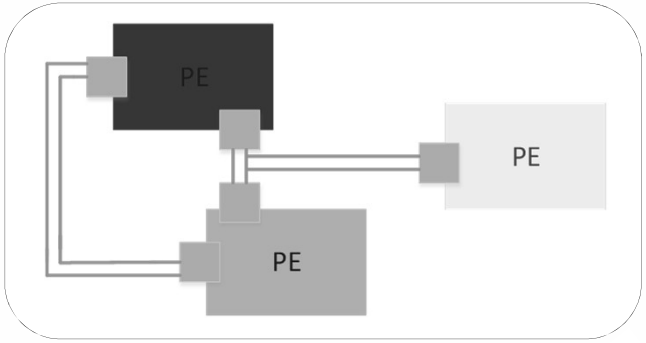
Types and Operators for Logic Modeling

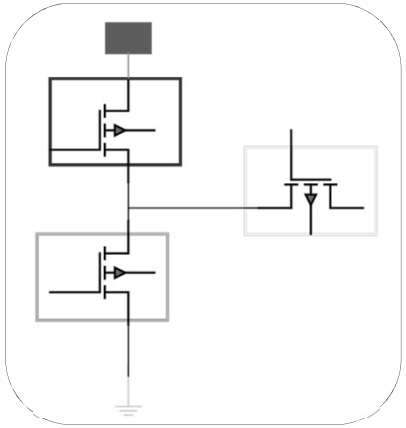
+ Basic Logic Simulation
+ Enhanced Logic Simulation with Timing
+ More Functions for Wires and Gates
+ Inheritance in Logic Structures
+ Hierarchical Modeling of Digital Components

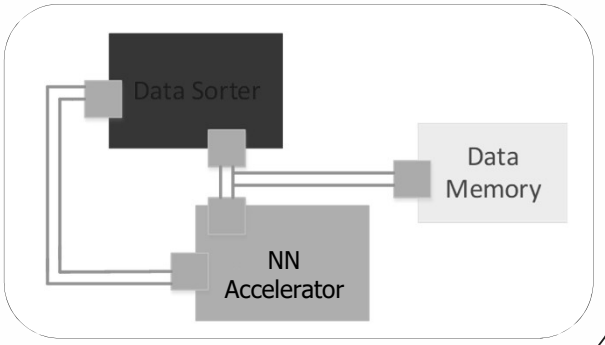
Summary

Procedural languages for Hardware Modeling

- Hardware Description Language
  - Concurrency
  - Timing
- Software Description Languages
  - Procedural

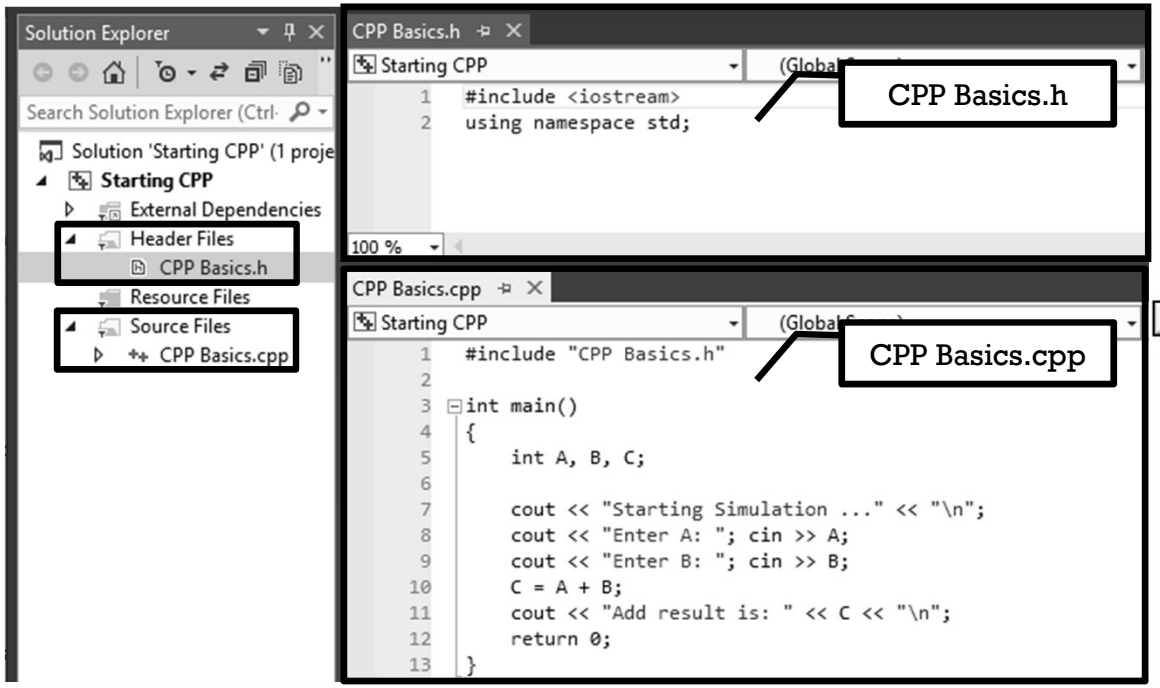




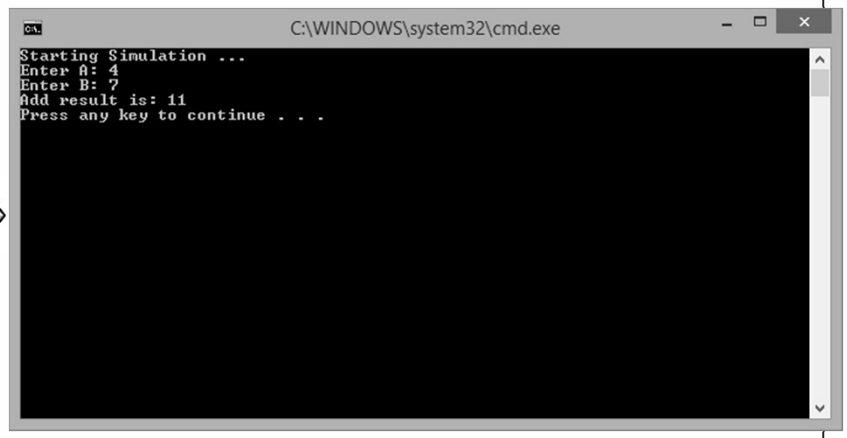


Procedural languages for Hardware Modeling

- C++ Environment



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



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

Ref. [1]: J. Soulie, C++ Language Tutorial, 2007.

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

Boolean Type.h

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

Boolean Type.cpp

```
#include "Boolean Type.h"

int main ()
{
    //bool a = true;
    //bool b = false;
    bool a(0);
    bool b(1);
    bool anding;
    int go;
    cout << "Performing Logic Simulation . . .\n";
    anding = a && b;
    cout << "a:" << a << "; b:" << b << "; anding:" << anding << "\n";
    cout << "Enter 0 to exit:";
    cin >> go;

    return 0;
}
```

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

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

Character Type.cpp

```
#include "Character Type.h"

#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

Creating a new data type using *enum*

Enumerations are type compatible with numeric variables

Constant arrays

Conditional Structure: if - else

Ref. [1] – Enumerations - P.84-P.85

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

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

Four Value System.h

```
Four Value System.h | Enum Type | (Global Scope)
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  enum lv4 {lX, l0,l1, lZ};
6  const lv4 lv4Value [4] = {lX, l0, l1, lZ};
7  const string lv4Image [4] = {"lX", "l0", "l1", "lZ"};
```

Four Value System.cpp

```
Four Value System.cpp | Enum Type | (Global Scope)
1  #include "Four Value System.h"
2
3  lv4 ANDlv4 (lv4 a, lv4 b)
4  {
5      lv4 w;
6      if (a==lX || b==lX || a==lZ || b==lZ) w=lX;
7      else if (a==l1 && b==l1) w=l1;
8      else w=l0;
9      return w;
10 }
11
12 lv4 ORlv4 (lv4 a, lv4 b){ ... }
20
21 lv4 XORlv4 (lv4 a, lv4 b){ ... }
29
30 int main (){ ... }
56
```

Types and Operators for Logic Modeling

- Using Enumeration (cont.)

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      do {
39          cout << "Enter operation (A,O,X), then inputs (0 to 3): ";
40          cin >> op >> Ii1 >> Ii2;
41          i1=lv4Value[Ii1]; i2=lv4Value[Ii2];
42          switch (op) {
43              case 'A': out = ANDlv4 (i1, i2); break;
44              case 'O': out = ORlv4 (i1, i2); break;
45              case 'X': out = XORlv4 (i1, i2); break;
46              default: out = lX;
47          }
48          cout << i1 << " " << op << " " << i2;
49          cout << ", is: " << out << '\n';
50          cout << lv4Image[i1] << " " << op << " " << lv4Image[i2];
51          cout << ", is: " << lv4Image[out] << '\n';
52          cout << "Enter 0 to end:"; cin >> go;
53      } while (go);
54      return 0;
55  }
56
```

100 %

Four Value System.cpp

Convert Integer value to lv4 enumeration type

Conversion to string for printing

Iteration Structure: do-while loop

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

Types and Operators for Logic Modeling

- Using String
  - Waveform Generation

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

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

Ref. [1] – Functions (I) - P. 41-46

String Character.h

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

String Character.cpp

```
#define MIN(a,b)((a<b)?a:b)

int wave (string seq)
{   int i, l;
    l = seq.length();
    for (i=0; i < l; i++)
    {
        if (seq[i]=='0') cout << "__";
        else cout << "--";
    }
    cout << '\n';
    return l;
}

bool char2bool (char c)
{
    if (c=='0') return 0;
    else return 1;
}

bool operation (string fn, bool in1, bool in2)
{
    bool out;
    if (fn == "AND" || fn == "and") out = in1 && in2;
    else if (fn == "OR" || fn == "or") out = in1 || in2;
    else if (fn == "XOR" || fn == "xor") out = in1 != in2;
    else out = 0;
    return out;
    }
```

Types and Operators for Logic Modeling

- Using String (cont.)
  - Waveform Generation

String Character.cpp

```
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  }
58
100 %
```

Calling the functions

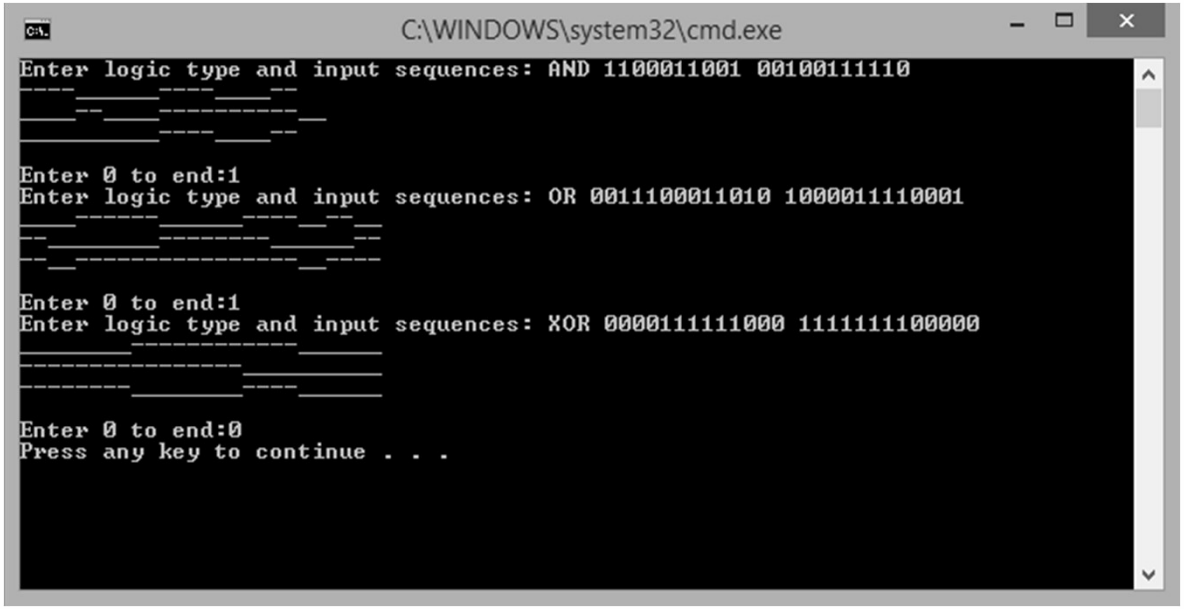
Using the macro:
for calculating the output waveform length

Iteration Structure:
for loop

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

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

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

Logic Simulation | (Global Scope)

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

LogicGates.h

primitives.h

Logic Simulation | (Global Scope)

```
1  bool and (bool a, bool b);
2  bool or (bool a, bool b);
3  bool not (bool a);
4  bool nand (bool a, bool b);
5  bool nor (bool a, bool b);
6  bool xor (bool a, bool b);
7
8  void and (bool a, bool b, bool& w);
9  void or (bool a, bool b, bool& w);
10 void not (bool a, bool& w);
11 void nand (bool a, bool b, bool& w);
12 void nor (bool a, bool b, bool& w);
13 void xor (bool a, bool b, bool& w);
14
15 bool logic (bool a, bool b, void (*f) (bool, bool, bool&));
16
17 bool and5 (bool a=true, bool b=true, // up to 5 inputs
18            bool c=true, bool d=true, bool e=true);
19 bool or5  (bool a=false, bool b=false,
20            bool c=false, bool d=false, bool e=false);
21 bool xor5 (bool a=false, bool b=false,
22            bool c=false, bool d=false, bool e=false);
23
24 void and (bool[], bool[], bool[], const int);
25 void or (bool[], bool[], bool[], const int);
26
27
```

Primitives.h

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

Logic Functions

- Gate Function Prototypes (cont.)

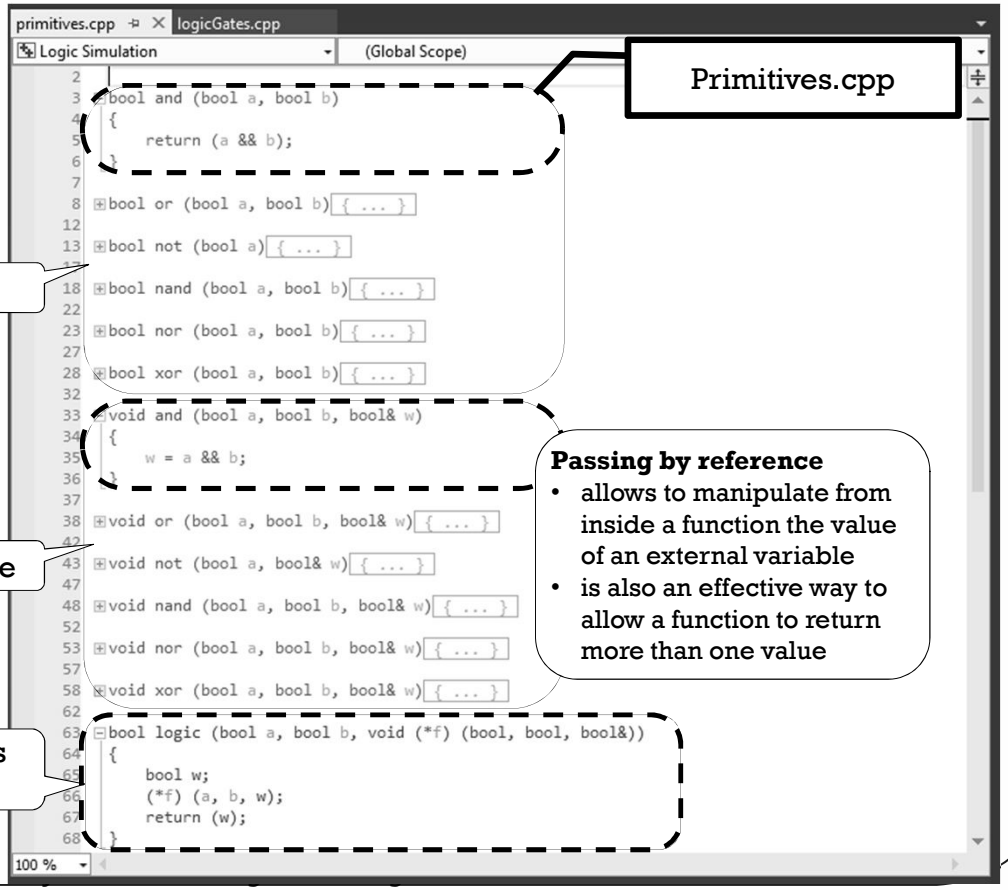
Functions are overloaded for various type of procedure and vector format

Ref. [1] – Overloaded functions - P. 50

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

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*

Logic Simulation | (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 }
```

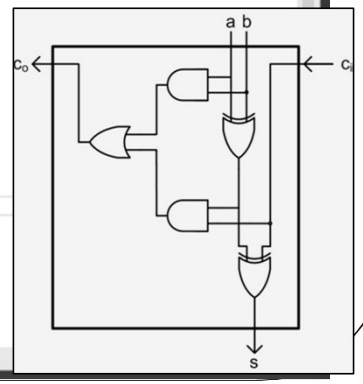
primitives.cpp | logicGates.cpp*

Logic Simulation | (Global Scope)

logicGates.cpp

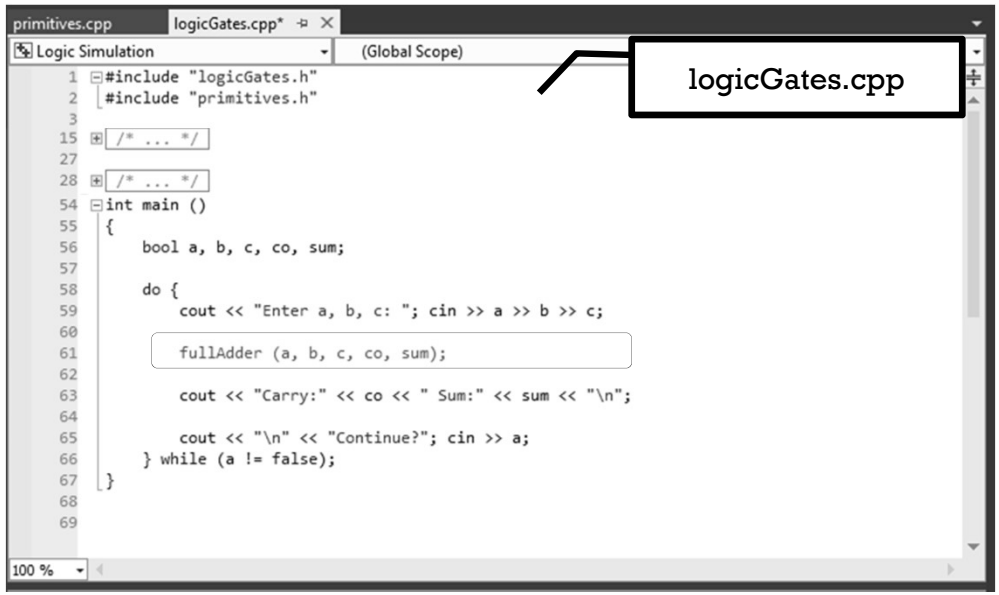
```
1  #include "logicGates.h"
2  #include "primitives.h"
3
4  /* ... */
16
17 /* ... */
29
30 void fullAdder (bool a, bool b, bool ci, bool& co, bool& sum)
31 {
32     bool axb, ab, abc;
33
34     axb = logic (a, b, xor); // uses: void xor (bool, bool, bool&);
35     ab  = logic (a, b, and);
36     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 () { ... }
66
67
```

100 %



Building Higher Level Structures

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



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

Handling 4-value Logic

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

characterFunctions.h

Character Logic | (Global Scope)

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

100 %

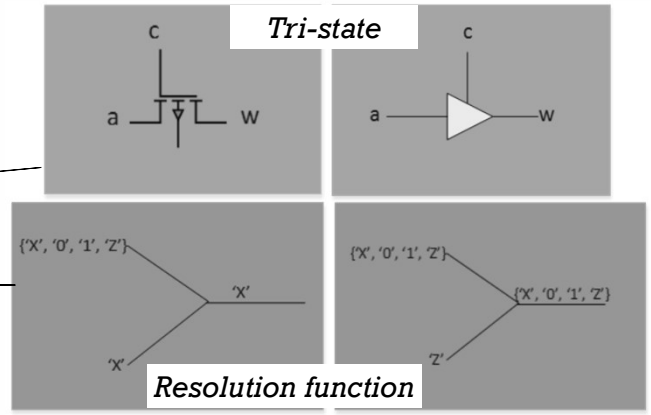
characterPrimitives.h

Character Logic | (Global Scope)

```
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
```

CharacterFunctions.h

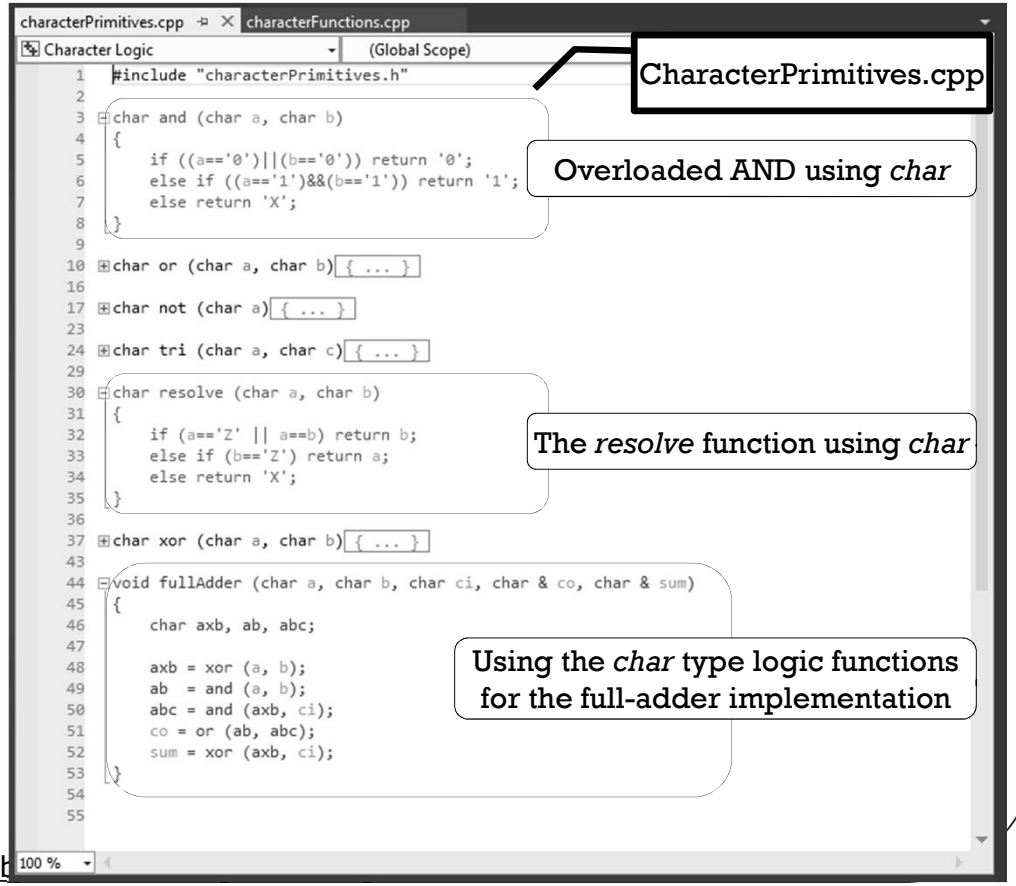
CharacterPrimitives.h



Handling 4-value Logic

- We have to generate our own logical functions
- This happens one 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




```
#include "characterPrimitives.h"

char and (char a, char b)
{
    if ((a=='0')||(b=='0')) return '0';
    else if ((a=='1')&&(b=='1')) return '1';
    else return 'X';
}

char or (char a, char b){ ... }

char not (char a){ ... }

char tri (char a, char c){ ... }

char resolve (char a, char b)
{
    if (a=='Z' || a==b) return b;
    else if (b=='Z') return a;
    else return 'X';
}

char xor (char a, char b){ ... }

void fullAdder (char a, char b, char ci, char & co, char & sum)
{
    char axb, ab, abc;

    axb = xor (a, b);
    ab  = and (a, b);
    abc = and (axb, ci);
    co = or (ab, abc);
    sum = xor (axb, ci);
}
```

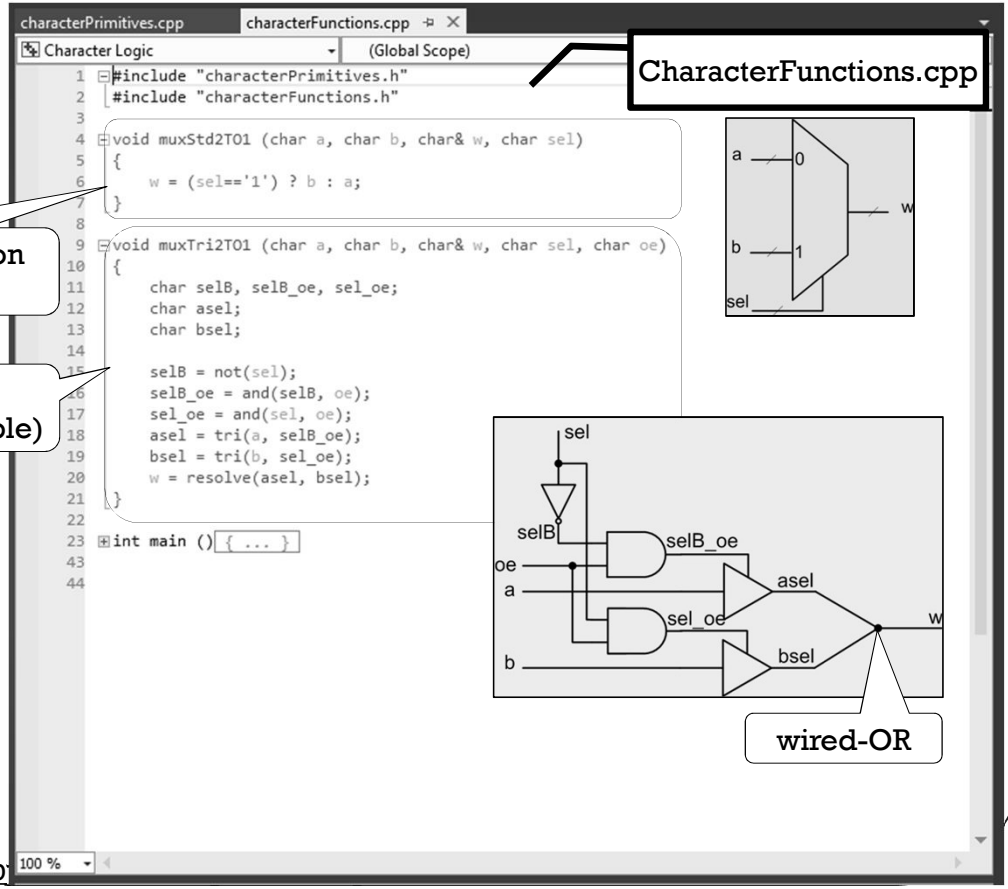
Handling 4-value Logic

- Using *char* for the multiplexer implementation

Behavioral implementation of Multiplexer

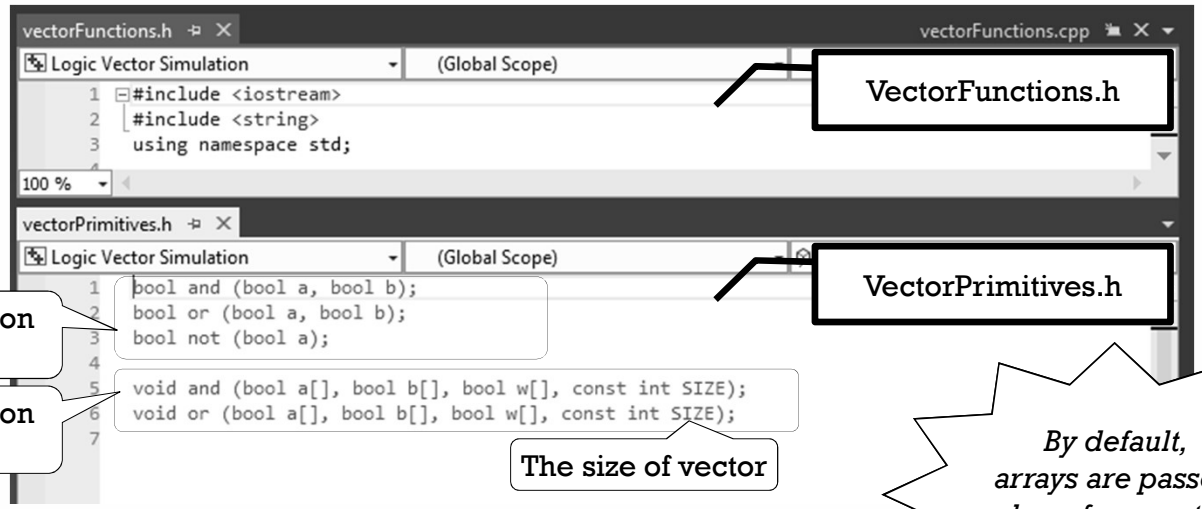
Structural implementation of Multiplexer with OE (Output Enable)

CharacterFunctions.cpp



Logic Vector

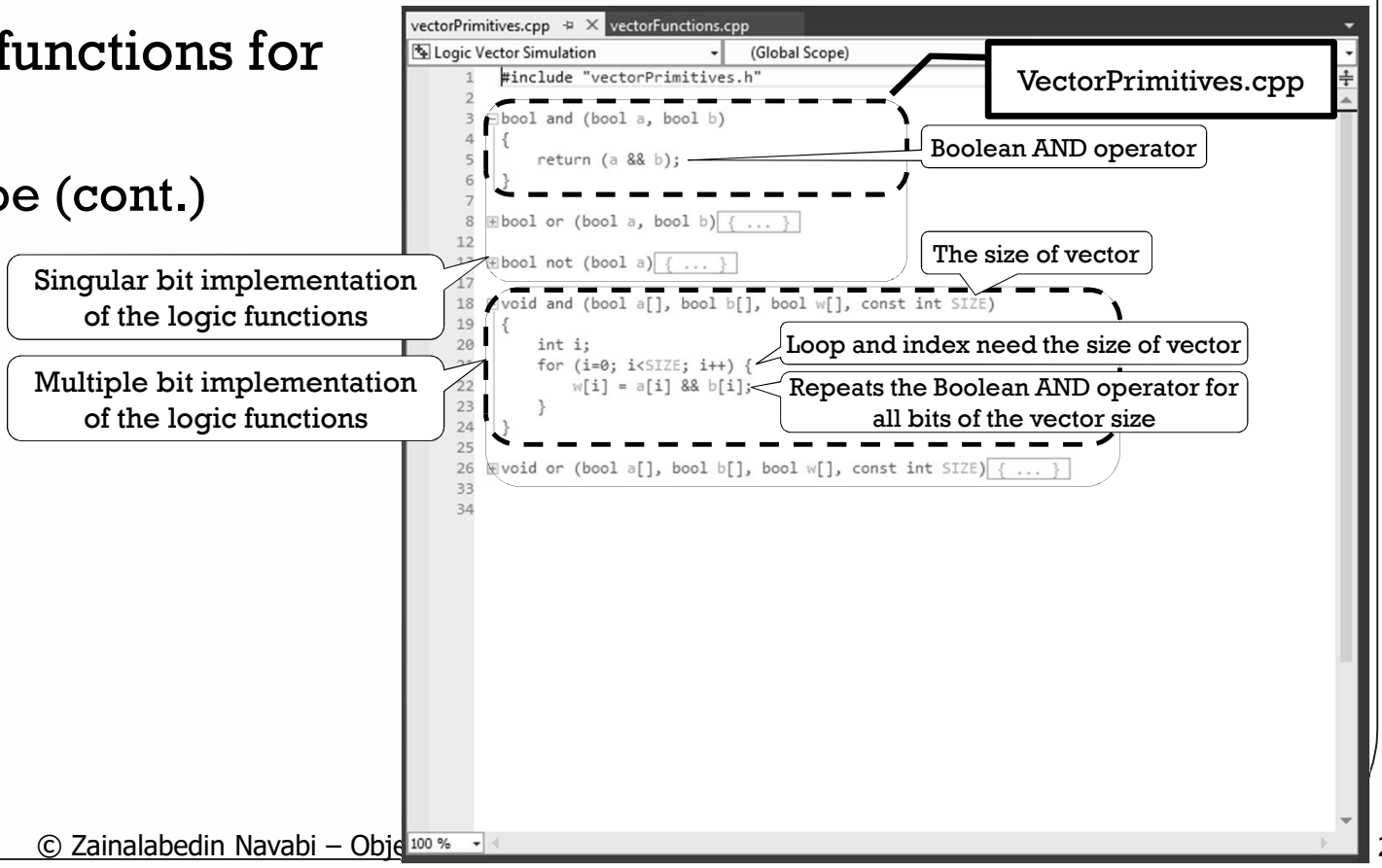
- Overloaded logic functions for vector format
  - Using Boolean Type



Ref. [1] – Arrays - P. 54-59

Logic Vector

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



Logic Vector

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

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

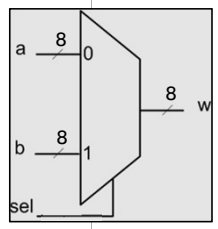
Implementing 8-bit *bool* vector based Multiplexer

Using the singular bit implementation of AND function

Using the multiple bit implementation of OR function

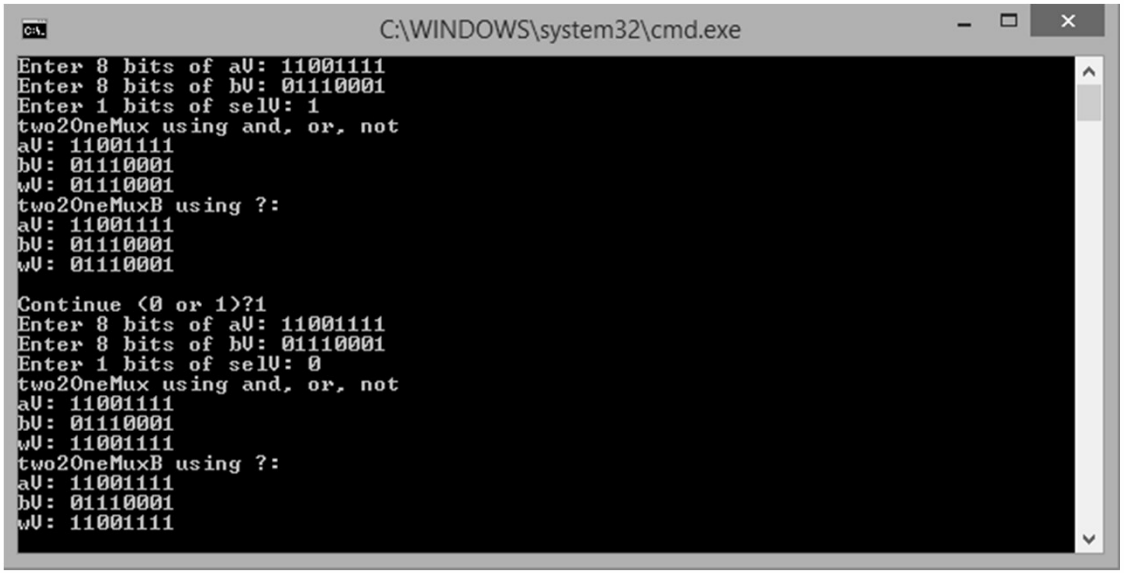
VectorFunctions.cpp

```
vectorPrimitives.cpp | vectorFunctions.cpp
Logic Vector Simulation | (Global Scope)
#include "vectorPrimitives.h"
#include "vectorFunctions.h"

void getBits (string vectorName, int numBits, bool values[])
{
    string valuesS;
    int i;
    cout << "Enter " << numBits << " bits of " << vectorName << ": ";
    cin >> valuesS;
    for (i=0; i<numBits; i++){
        if (valuesS[i] == '1') values[i] = true;
        else values[i] = false;
    }
}

void putBits (string vectorName, int numBits, bool values[]) { ... }
void two2OneMux (bool a[], bool b[], bool w[], bool sel, int SIZE=8)
{
    bool as [8];
    bool bs [8];

    int i;
    for (i=0; i<SIZE; i++) {
        as[i] = and (a[i], not(sel));
    }
    for (i=0; i<SIZE; i++) {
        bs[i] = and (b[i], sel);
    }

    or (as, bs, w, SIZE);
}

void two2OneMuxB (bool a[], bool b[], bool w[], bool sel, int SIZE=8) { ... }

int main () { ... }

100 %
```



Logic Vector

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



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

characterVectorFunctions.cpp | characterVectorFunctions.h | characterVectorPrimitives.h

Character Vector Logic | (Global Scope)

characterVectorPrimitives.h

```
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);
18
```

Singular bit implementation of the logic functions

Multiple bit implementation of the logic functions

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

100 %

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

characterVectorPrimitives.cpp

Using the singular bit implementation of AND function

Puts the null character on the *i*th (last) position of the output vector

characterVectorFunctions.cpp | characterVectorPrimitives.h | characterVectorPrimitives.cpp

Character Vector Logic | (Global Scope)

```
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]);
            i++;
44      };
45      w[i] = '\0';
46  }
47
48  void or (char a[], char b[], char w[]) { ... }
57
58  void tri (char a[], char c, char w[]) { ... }
67
68  void resolve (char a[], char b[], char w[])
69  {
70      int i=0;
71      while (a[i] != '\0') {
72          w[i] = resolve (a[i], b[i]);
73          i++;
74      };
75      w[i] = '\0';
76  }
77
78  char xor (char a, char b) { ... }
84
85  void fullAdder (char a, char b, char ci, char & co, char & sum)
86  {
87      char axb, ab, abc;
88
89      axb = xor (a, b);
90      ab  = and (a, b);
91      abc = and (axb, ci);
92      co = or (ab, abc);
93      sum = xor (axb, ci);
94  }
95
```

100 %

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)

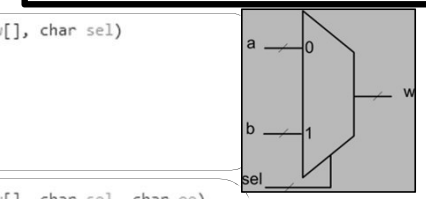
Using the singular bit implementation of logic function

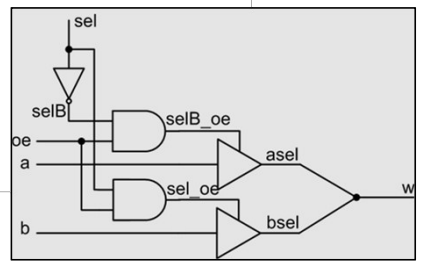
Using the multiple bit implementation of logic function

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

characterVectorFunctions.cpp

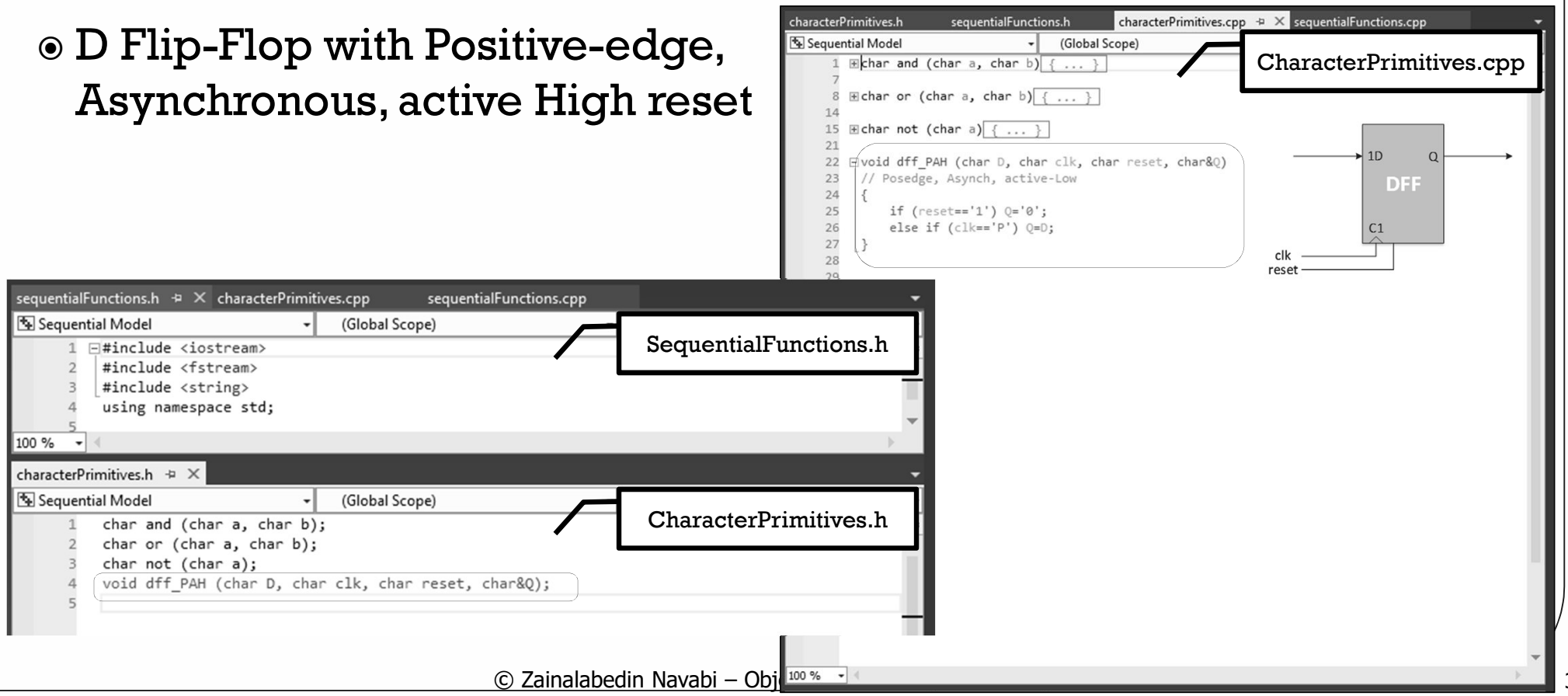
```
#include "characterVectorPrimitives.h"
#include "characterVectorFunctions.h"

void mux8Std2TO1 (char a[], char b[], char w[], char sel)
{
    int i=0;
    do {
        w[i] = (sel=='1') ? b[i] : a[i];
    } while (a[i++] != '\0');
}

void mux8Tri2TO1 (char a[], char b[], char w[], char sel, char oe)
{
    char selB, selB_oe, sel_oe;
    char asel [9];
    char bsel [9];

    selB = not(sel);
    selB_oe = and(selB, oe);
    sel_oe = and(sel, oe);
    tri(a, selB_oe, asel);
    tri(b, sel_oe, bsel);
    resolve(asel, bsel, w);
}

int main ()
{
    char aCV [9], bCV [9];
    char sel, oe;
    char wCV [9];
    int ai;
    do {
        cout << "Enter eight bits of aCV <space> bCV: "; cin >> aCV >> bCV;
        cout << "Enter sel <space> oe: "; cin >> sel >> oe;

        mux8Std2TO1 (aCV, bCV, wCV, sel);
        cout << "The " << strlen(wCV) << " bits of wC become as follows: \n";
```





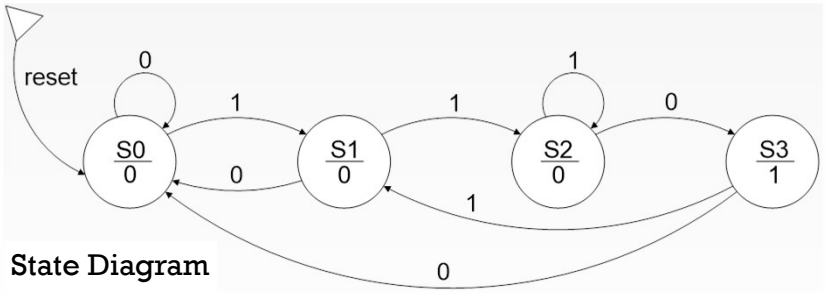
Sequential Circuit Modeling

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

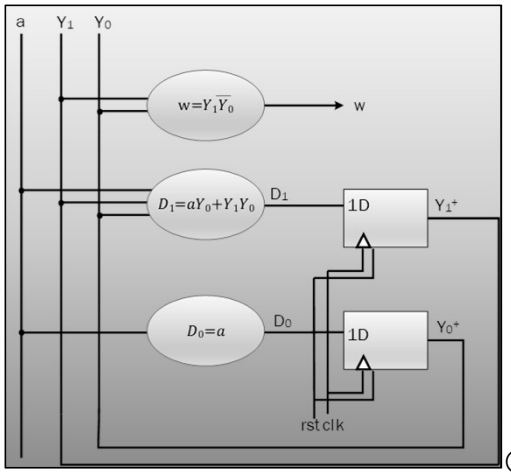


Sequential Circuit Modeling

- Moore 110 sequence detector



State Diagram



State Table

	State	a = 0	a = 1	w
00	S0	S0	S1	0
01	S1	S0	S2	0
11	S2	S3	S2	0
10	S3	S0	S1	1
		State+		

Transition Table

Y_1Y_0	a = 0	a = 1	w
00	00	01	0
01	00	11	0
11	10	11	0
10	00	01	1
	$Y_1^+Y_0^+$		

Excitation Table

Y_1Y_0	a = 0	a = 1	w
00	00	01	0
01	00	11	0
11	10	11	0
10	00	01	1
	D_1D_0		

Y_1Y_0 \ a	0	1
00	0	0
01	0	1
11	1	1
10	0	0

D_1

$D_1 = aY_0 + Y_1Y_0$

Y_1Y_0 \ a	0	1
00	0	1
01	0	1
11	0	1
10	0	1

D_0

$D_0 = a$

Y_0 \ Y_1	0	1
0	0	1
1	0	0

w

$w = Y_1\overline{Y_0}$

Sequential Circuit Modeling

- Moore 110 sequence detector (cont.)

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

Input file

Output file

indata.tst

```
000
000
000
100
10P
010
00P
10P
10P
00P
00P
10P
10P
10P
00P
10P
10P
00P
10P
00P
10P
10P
```

outdata.tst

```
Inp -> Output, Next state
X,XX
000 -> X,XX
000 -> X,XX
000 -> X,XX
100 -> X,XX
10P -> 0,X1
010 -> 0,00
00P -> 0,00
10P -> 0,01
10P -> 0,11
00P -> 1,10
00P -> 0,00
10P -> 0,01
10P -> 0,11
00P -> 1,10
10P -> 0,11
10P -> 0,11
00P -> 1,10
10P -> 0,01
10P -> 0,11
00P -> 1,10
10P -> 0,01
10P -> 0,11
10P ->
```

clock by clock output

ain, reset, clock

w and 2-bits of states

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

Sequential Model | (Global Scope)

SequentialFunctions.cpp

```
#include "characterPrimitives.h"
#include "sequentialFunctions.h"

int main ()
{
    string inVec;
    string outVec = ",,,,";
    char ain('0'), reset, clock;

    char Y1('X'), Y0('X'), D1, D0, w;

    ifstream finp ( "indata.tst");
    ofstream fout ("outdata.tst");

    fout << "Inp -> Output, Next state\n";

    do {
        finp >> inVec;
        ain = inVec[0];
        reset = inVec[1];
        clock = inVec[2];
      // combinational parts in proceduaral fashion
      // followed by the sequential parts

      D1 = or(and(Y1, Y0), and (ain, Y0));
      D0 = ain;
      w = and(Y1, not(Y0));

      outVec[0] = w;  // These values are after
      outVec[2] = Y1; //   application of the
      outVec[3] = Y0; //   previous inputs
      fout << outVec+"\n" << inVec << " -> ";

      dff_PAH (D1, clock, reset, Y1);
      dff_PAH (D0, clock, reset, Y0);
    } while (!finp.eof());
}
```

100 %

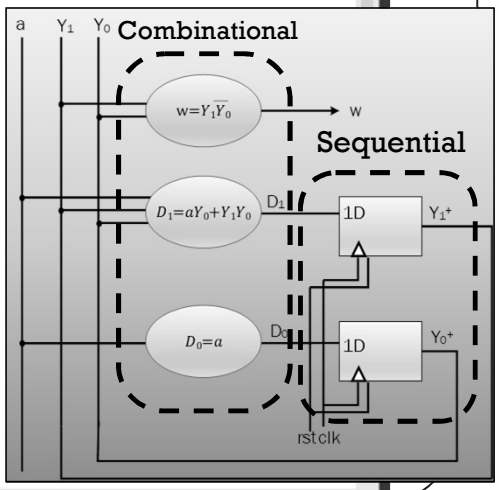
File handling

Convert *string* to *char* operations are in *char*

Combinational Part

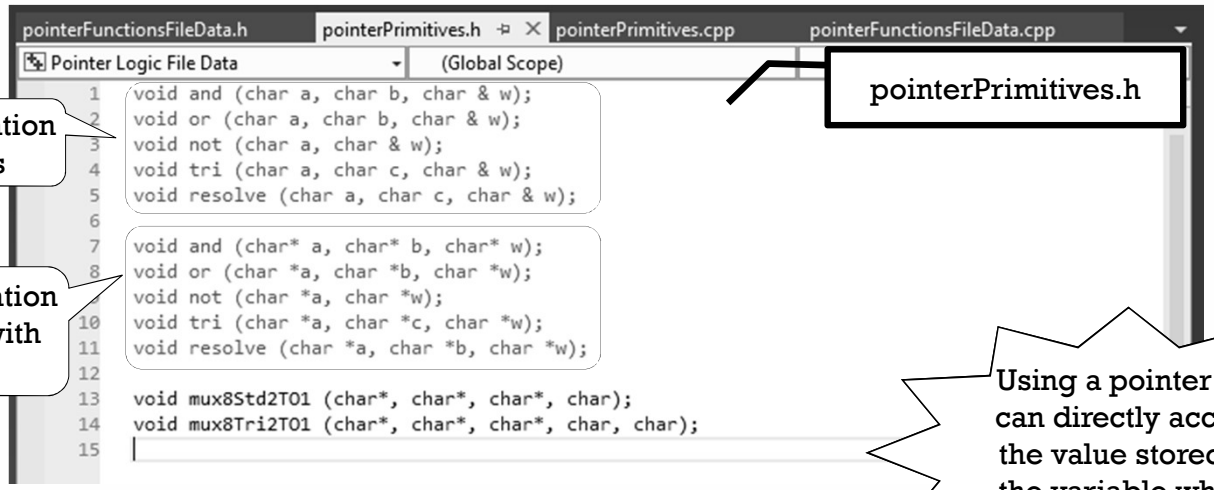
Convert *char to string* for printing

Sequential Part



Using Pointers for Logic Vectors

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



Singular bit implementation of the logic functions

Multiple bit implementation of the logic functions with pointer arguments

Using a pointer we can directly access the value stored in the variable which it points to

Ref. [1] – Pointers - P. 63-73

Using Pointers for Logic Vectors

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

Singular bit implementation of the AND functions

Multiple bit implementation of the AND functions with pointer arguments

Using the singular bit implementation of AND function

pointerPrimitives.cpp

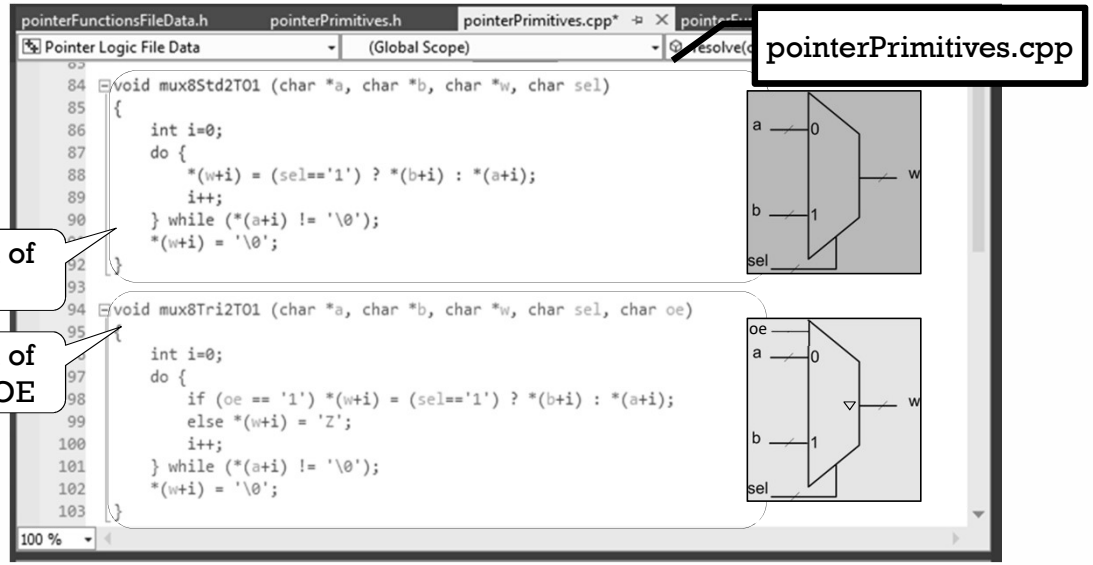
```
pointerFunctionsFileData.h | pointerPrimitives.h | pointerPrimitives.cpp | pointerFunctionsFileData.cpp
Pointer Logic File Data | (Global Scope)
 1  #include <iostream>
 2  using namespace std;
 3
 4  void and (char a, char b, char & w)
 5  {
 6      w = ((a=='0')||(b=='0')) ? '0':
 7          ((a=='1')&&(b=='1')) ? '1':
 8                                 'X';
 9  }
11  void or (char a, char b, char  & w){ ... }
17
18  void not (char a, char & w){ ... }
24
25  void tri (char a, char c, char & w){ ... }
29
30  void resolve (char a, char b, char & w){ ... }
36
37  void and (char* a, char* b, char* w)
38  {
39      int i=0;
40      do {
41          and (*(a+i), *(b+i), *(w+i));
42          i++;
43      } while (*(a+i) != '\0');
44      *(w+i) = '\0';
45  }
46
47  void or (char *a, char *b, char *w){ ... }
56
57  void not (char *a, char *w){ ... }
66
67  void tri (char *a, char *c, char *w){ ... }
76
77  void resolve (char *a, char *b, char *w){ ... }
86
87  void mux8Std2T01 (char *a, char *b, char *w, char sel)
100 %
```

Using Pointers for Logic Vectors

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

Behavioral implementation of standard Multiplexer

Behavioral implementation of Tri-state Multiplexer with OE



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

Pointer Logic File Data | (Global Scope)

pointerFunctionsFileData.cpp

```
#include "pointerPrimitives.h"
#include "pointerFunctionsFileData.h"

int main ()
{
    ifstream inp ("inpdata.tst"); //declare and initialize inp
    ofstream out ("outdata.tst"); //declare and initialize out

    int ii;
    inp >> ii;
    out << "All vector lengths are " << ii << " bits.\n";

    char sel, oe;
    char* aC = new char [ii+1];
    char* bC = new char [ii+1];
    char* wC = new char [ii+1];

    while (inp >> aC >> bC >> sel >> oe)
    {
        out << "Inputs are a, b vectors and sel, oe bits: ";
        out << aC << " " << bC << " " << sel << " " << oe << "\n";

        mux8Std2TO1 (aC, bC, wC, sel);
        out << "Std Mux: " << wC << '\n';

        mux8Tri2TO1 (aC, bC, wC, sel, oe);
        out << "Tri Mux: " << wC << '\n';
    }
}
```

File handling

Calling the standard Multiplexer

Calling the Tri-state Multiplexer

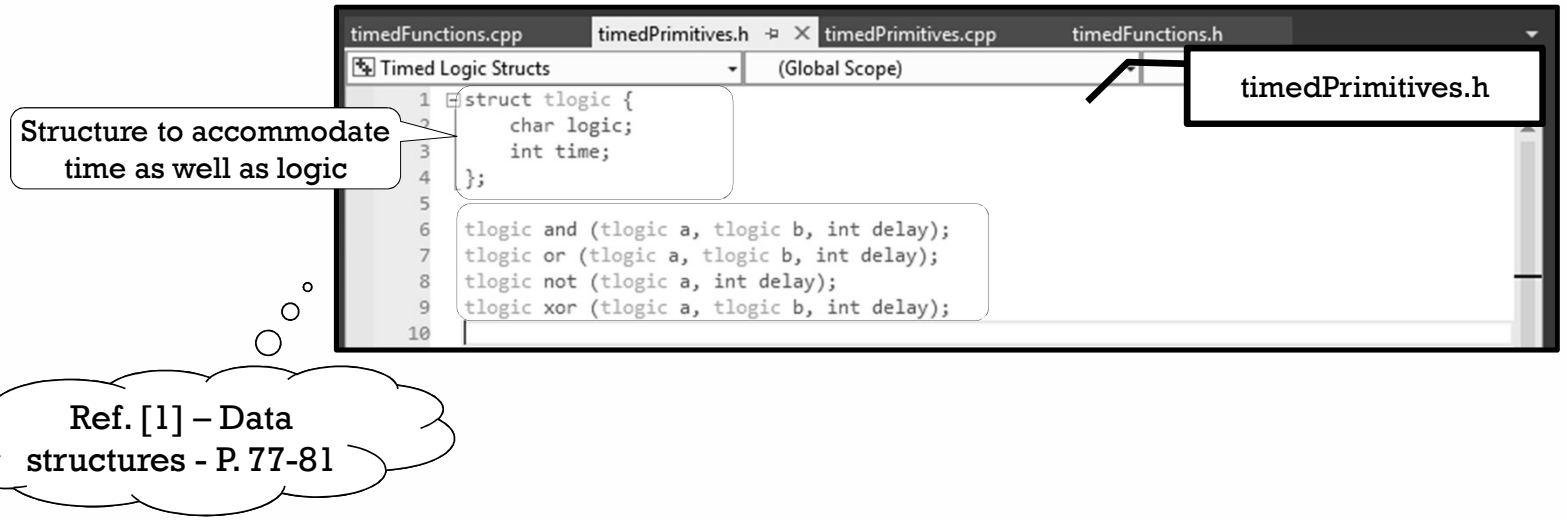
100 %

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

Using *struct* for Timing and Logic

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



Gates that Handle Timing

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

timedPrimitives.cpp

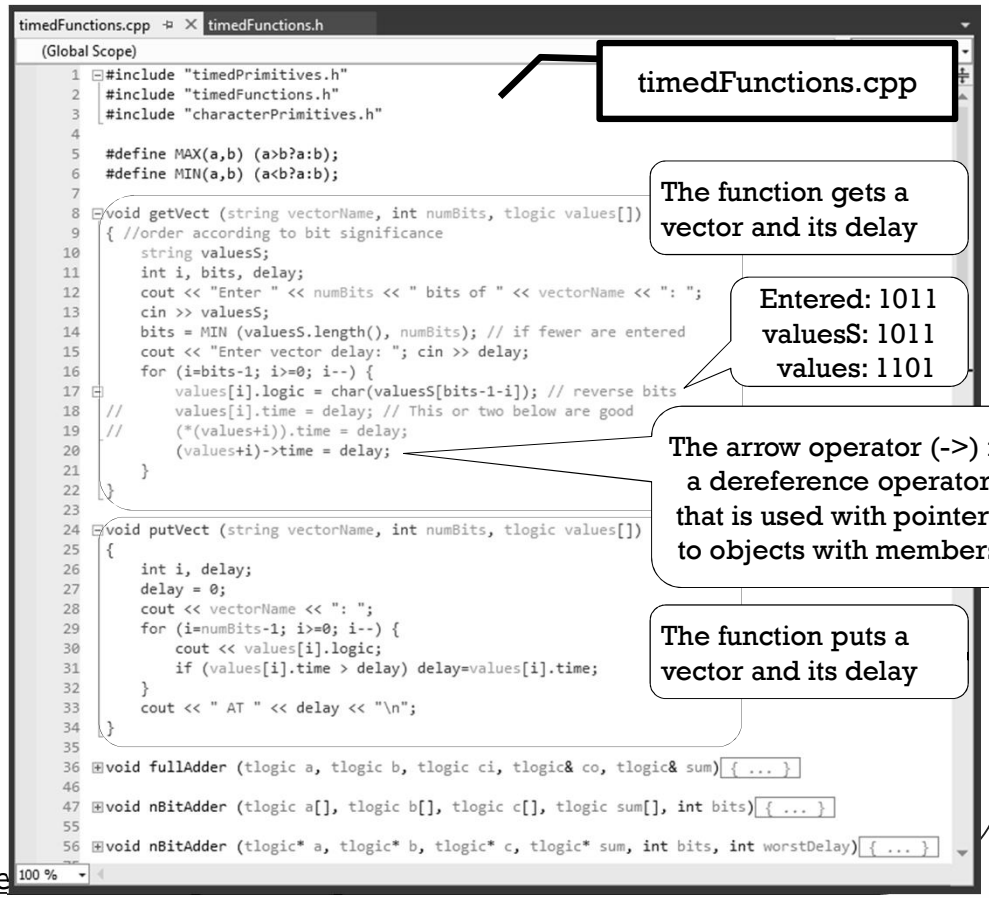
AND logic function with timing

```
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      return tl;
22  }
23
24  tlogic or (tlogic a, tlogic b, int delay) { ... }
44
45  tlogic not (tlogic a, int  delay) { ... }
54
55  tlogic xor (tlogic a, tlogic b, int delay)
56  {
57      tlogic tl;
58      if (a.logic==b.logic) tl.logic = '0';
59      else tl.logic = '1';
60      if (a.time > b.time) tl.time = a.time + delay;
61      else tl.time = b.time + delay;
62      return tl;
63  }
64
100 %
```

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)



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

Using the timed logic functions

Multiple bit implementation of the FA function with timing

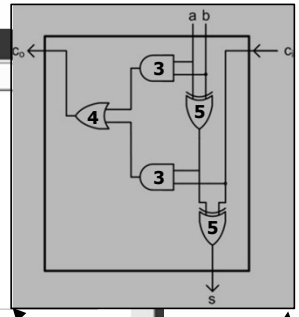
The operator *new* is used to allocate dynamic memory

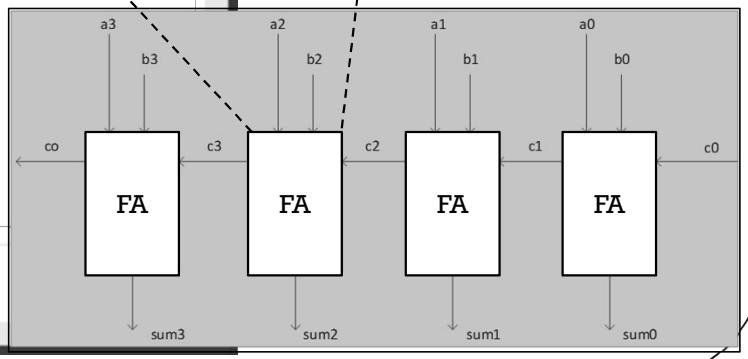
Ref. [1] – Dynamic Memory - P. 74-76

Using the timed singular bit FA

timedFunctions.cpp

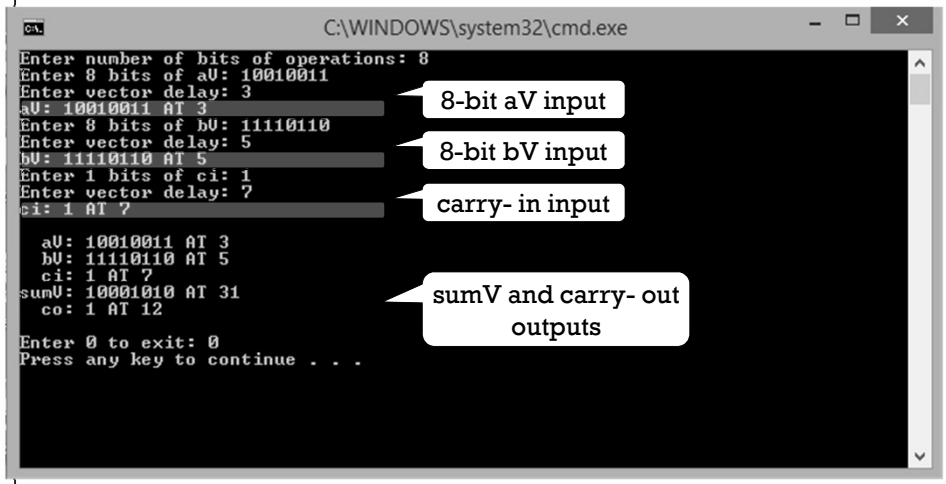
```
void fullAdder (tlogic a, tlogic b, tlogic ci, tlogic& co, tlogic& sum)
{
    tlogic axb, ab, abc;

    axb = xor (a, b, 5);
    ab  = and (a, b, 3);
    abc = and (axb, ci, 3);
    co = or (ab, abc, 4);
    sum = xor (axb, ci, 5);
}

void nBitAdder (tlogic a[], tlogic b[], tlogic ci[], tlogic co[], tlogic sum[], int bits)
{
    // assumes 0 is LSB
    int i;
    tlogic* c = new tlogic[bits+1];
    c[0] = ci[0];
    for (i = 0; i<bits; i++)
    {
        fullAdder(a[i], b[i], c[i], c[i+1], sum[i]);
    }
    co[0] = c[bits];
}

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

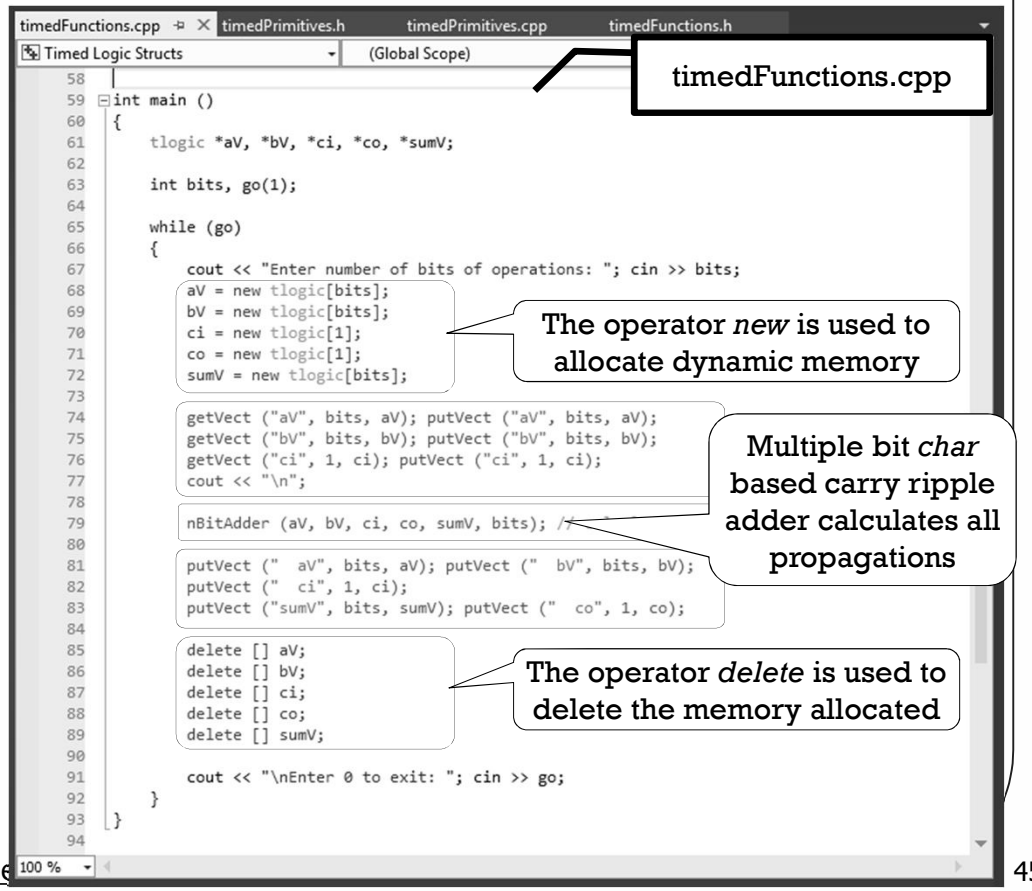




Timing in Logic Structures (cont.)

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





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

timedOperators.cpp

```
#include "timedOperators.h"

tlogic operator& (tlogic a, tlogic b) { ... }

tlogic operator| (tlogic a, tlogic b) { ... }

tlogic operator~ (tlogic a)
{
    tlogic tl;
    if (a.logic=='1') tl.logic = '0';
    else if (a.logic=='0') tl.logic = '1';
    else tl.logic=='X';
    tl.time = a.time;
    return tl;
}

tlogic operator^ (tlogic a, tlogic b)
{
    tlogic tl;
    if (a.logic==b.logic) tl.logic = '0';
    else tl.logic = '1';
    if (a.time > b.time) tl.time = a.time;
    else tl.time = b.time;
    return tl;
}
```

timedOperators.h

```
struct tlogic {
    char logic;
    int time;
};

tlogic operator& (tlogic a, tlogic b);
tlogic operator| (tlogic a, tlogic b);
tlogic operator~ (tlogic a);
tlogic operator^ (tlogic a, tlogic b);
```

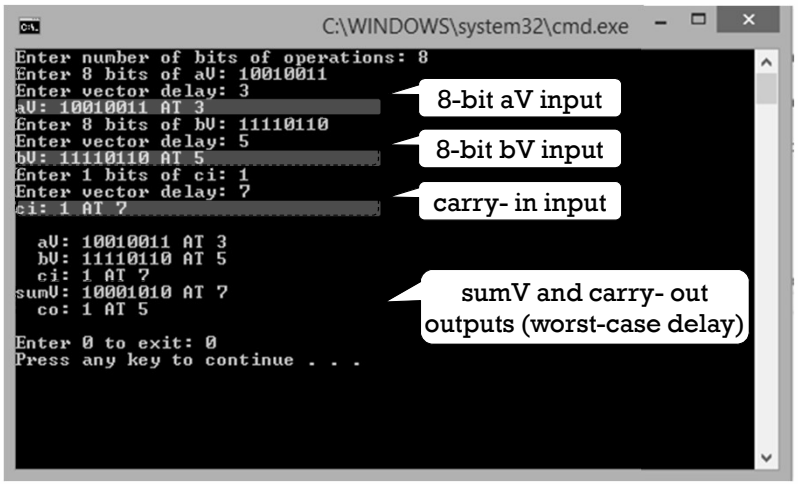
Structure to accommodate time as well as logic

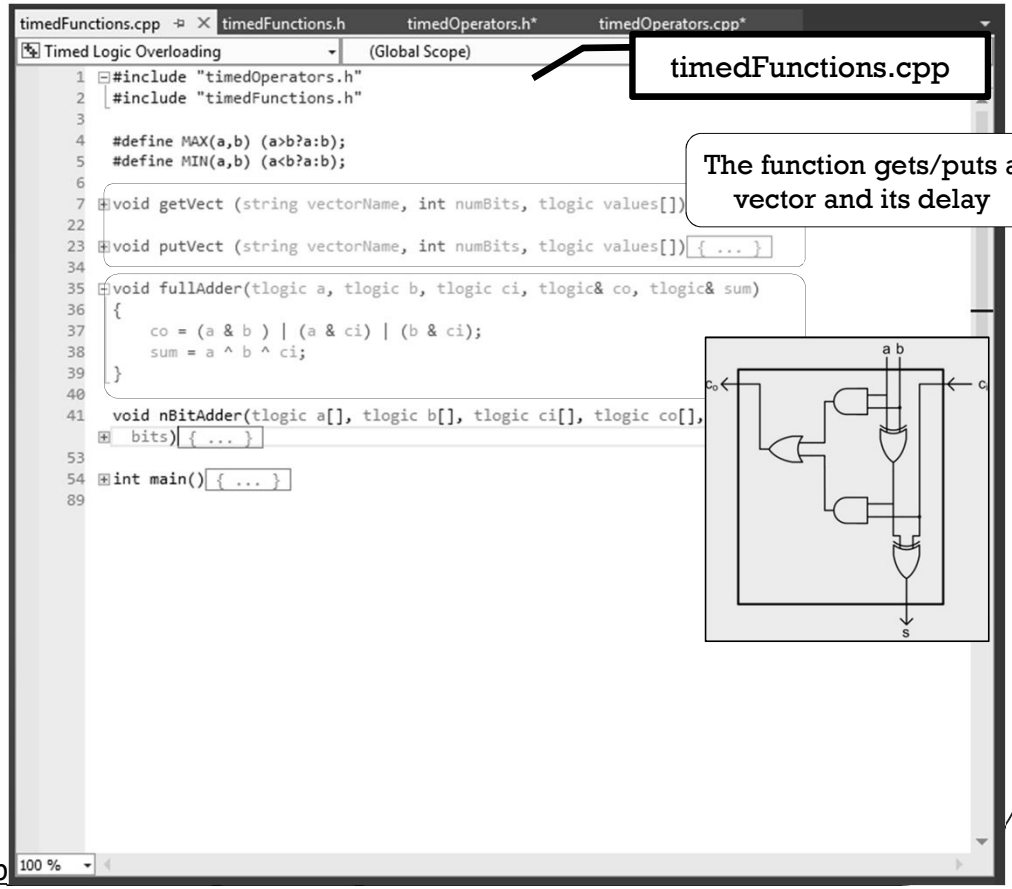
Ref. [1] – Overloading operators - P. 95-97

Consistent with the HDLs

Using Boolean Expressions

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





Object Oriented Logic Modeling

Procedural Lang. for Hardware Modeling

Types and Operators for Logic Modeling

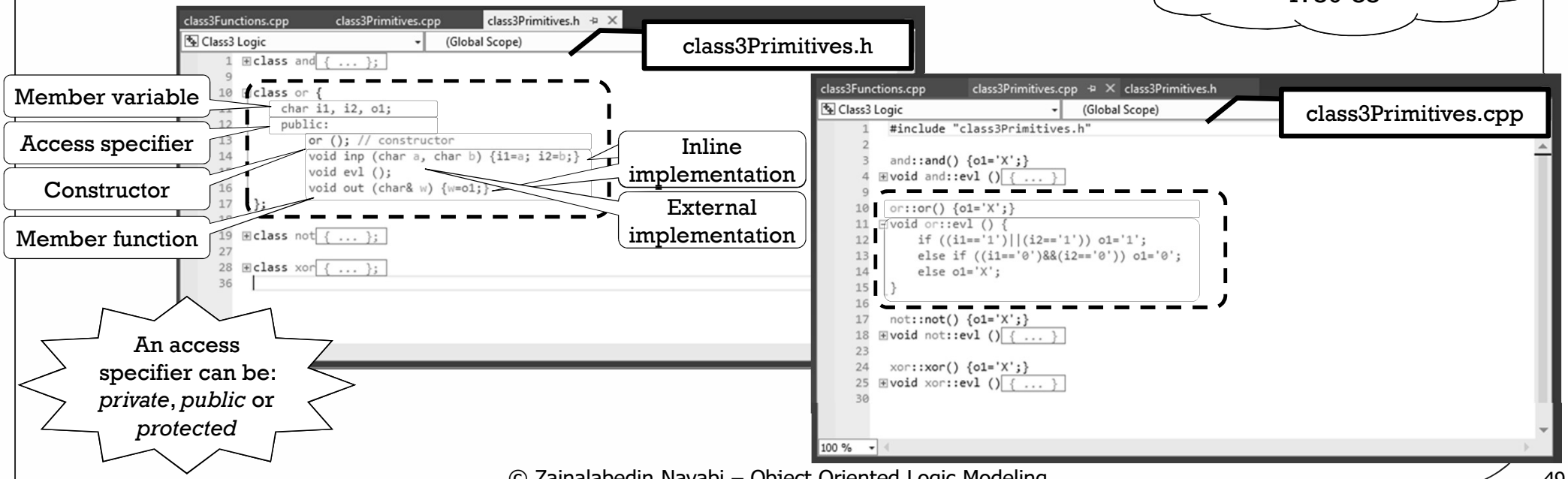
+ Basic Logic Simulation
+ Enhanced Logic Simulation with Timing
– More Functions for Wires and Gates
  - Gate classes
  - Carrier generic modeling
  - Pointer-based logic classes
  - Gate classes with power and timing calculation
  - Wire and gate vectors
+ Inheritance in Logic Structures
+ Hierarchical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

Summary

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

Ref. [1] – Classes (I) - P. 86-88



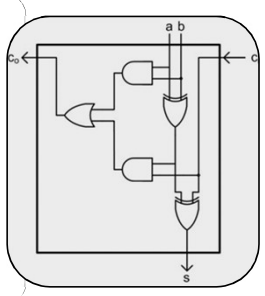
Gate Classes (cont.)

- Implementation of 1-bit full adder using gate classes

class3Functions.cpp

```
#include "class3Primitives.h"
#include "class3Functions.h"

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);
}


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

The gate classes

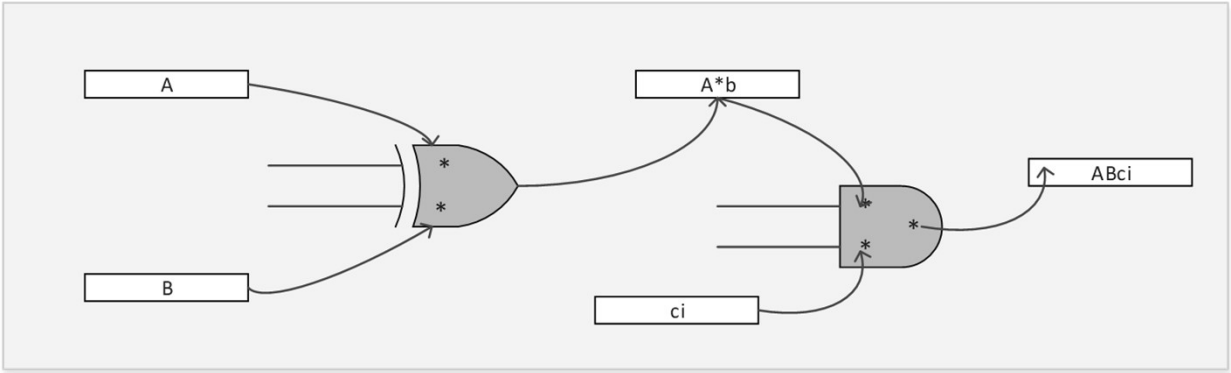


class3Functions.cpp

```
int main ()
{
    char aC;
    char bC;
    char ciC;
    char coC;
    char sumC;

    int ai;

    do {

        cout << "Enter a: ";
        cin >> aC; cout << aC << '\n';

        cout << "Enter b: ";
        cin >> bC; cout << bC << '\n';

        cout << "Enter ci: ";
        cin >> ciC; cout << ciC << '\n';

//      and (aC, bC, wC);
//      cout << "and:" << wC << '\n';

//      or (aC, bC, wC);
//      cout << "or: " << wC << '\n';

        fullAdder (aC, bC, ciC, coC, sumC);

        cout << "Carry: " << coC << '\n';
        cout << "  Sum: " << sumC << '\n';

        cout << "\n" << "Continue?"; cin >> ai;

    } while (ai>0);
}
```

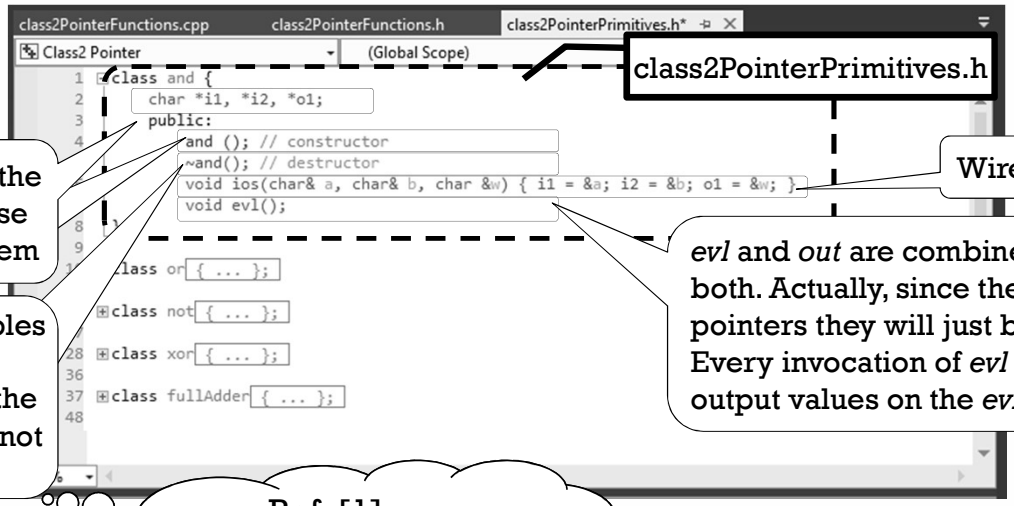
Calling the 1-bit full adder function in *main* as a testbench

Carrier Generic Modeling



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

Wires are of *char* type

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

- ✓ 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

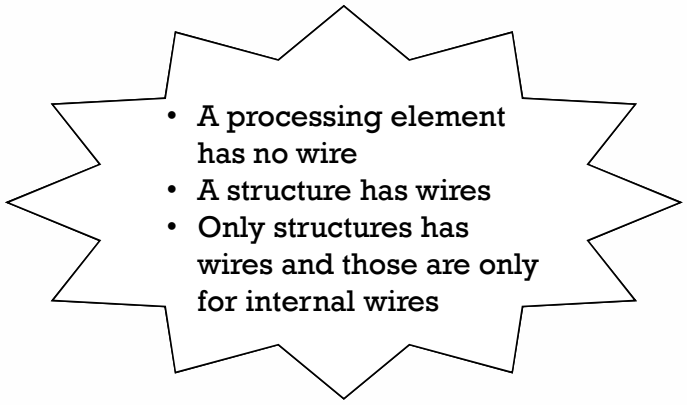
Destructor fulfills the opposite functionality. It is suitable to release the memory that the object was allocated

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

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

Pointer Based Logic Classes

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



class2PointerPrimitives.cpp

```
class2PointerFunctions.h   class2PointerPrimitives.h   class2PointerPrimitives.cpp
Class2 Pointer                (Global Scope)
 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 () { ... }
17
18  not::not() {}
19  void not::evl () { ... }
24
25  xor::xor() {}
26  void xor::evl () { ... }
31
32  fullAdder::fullAdder() {}
33  void fullAdder::evl () { ... }
69
70  halfAdder::halfAdder() {}
71  void halfAdder::evl () { ... }
97
100 %
```

Pointer Based Logic Classes

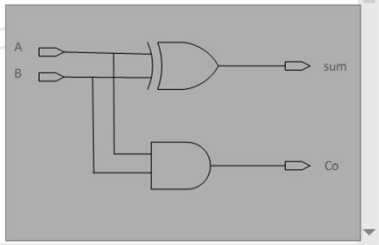
- *char* type pointer-based gate classes (cont.)
  - Half adder
    - Higher-level structures also have gates that have no internal wires

class2PointerPrimitives.h

```
class or { ... };
class not { ... };
class xor { ... };
class fullAdder { ... };
class halfAdder {
    char *i1, *i2, *o1, *o2;
    public:
        halfAdder (); // constructor
        ~halfAdder(); // destructor
        void ios(char& a, char& b, char& co, char& sum)
        {
            i1 = &a; i2 = &b; o1 = &co; o2 = &sum;
        }
        void evl();
};
```

class2PointerPrimitives.cpp

```
halfAdder::halfAdder() {}
void halfAdder::evl () {
    // halfadder Local wires
    char aL('X'), bL('X');
    char coL('X'), sumL('X');

    // Declare necessary gate instances
    xor *xor1 = new xor;
    and *and1 = new and;

    // Associate ports of the gates with the Local HA wires
    xor1->ios(aL, bL, sumL);
    and1->ios(aL, bL, coL);

    // Via the HA pointers, read wire values that connect to
    // the HA from outside, and assign them to HA Local wires
    aL = *i1; bL = *i2;

    // Evaluate gates in the proper order
    xor1->evl();
    and1->evl();

    // Take calculated local wire values and assign the values
    // to the outside wires via pointers of FA
    *o1 = coL; *o2 = sumL;
}
```



Gate Classes with Power and Timing Calculation

- *wire* type pointer-based logic classes
  - AND gate

Ref. [1] – Overloading Constructors - P. 90-92

Declare *wire* to contain more information than just logic value

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

timedLogicPrimitives.h

```
timedLogicFunctions.cpp   timedLogicUtilities.cpp   timedLogic...
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 ev1();
26  };
27
28  class or { ... };
37
38  class not { ... };
47
48  class xor { ... };
57
58  class dff_ar {
59      wire *D, *clk, *R, *Q;
60      int clkQDelay, rstQDelay;
61      int lastEvent; // last time output changed
100 %
```

Gate Classes with Power and Timing Calculation

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

timedLogicPrimitives.cpp

```
timedLogicUtilities.cpp   timedLogicUtilities.h   timedLogicFunctions.h
Timed Logic Classes        fullAdder
 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);
96
97      lastEvent = o1->eventTime;
98      lastValue = o1->value;
99  }
100 %
```

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

Overloaded function

Logic part

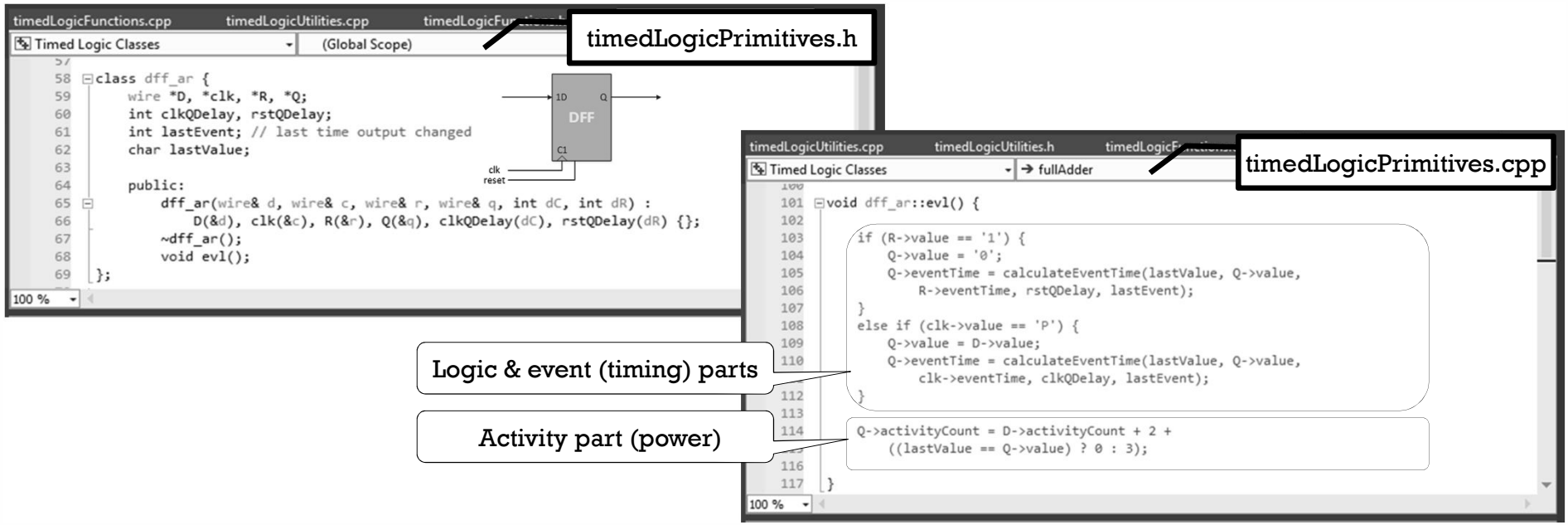
Event part (timing)

Activity part (power)

Retain last event and last value

Gate Classes with Power and Timing Calculation

- *wire* type pointer-based logic classes (cont.)
  - D-Flip Flop with Asynchronous Reset



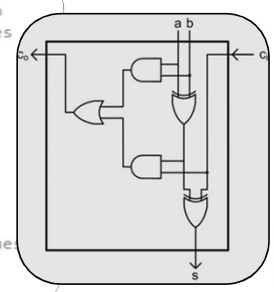
Gate Classes with Power and Timing Calculation

- *wire* type pointer-based logic classes (cont.)
  - 1-bit full adder

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

timedLogicPrimitives.cpp

```
timedLogicFunctions.h | timedLogicPrimitives.cpp
Timed Logic Classes | (Global Scope)
120
121 void fullAdder::evl () {
122
123     // Via the FA pointers, read wire values that connect to
124     // the FA from outside, and assign them to FA Local wires
125     aL = *i1; bL = *i2; ciL = *i3;
126
127     // Evaluate gates in the proper order
128     xor1->evl();
129     and1->evl();
130     and2->evl();
131     or1->evl();
132     xor2->evl();
133
134     // Take calculated local wire values and assign the value
135     // to the outside wires via pointers of FA
136     *o1 = coL; *o2 = sumL;
137 }
138
100 %
```



timedLogicPrimitives.h

```
timedLogicFunctions.cpp | timedLogicUtilities.cpp
Timed Logic Classes | (Global Scope)
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, ciL;
85      wire coL, 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), ciL('X', 0),
92              coL('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, ciL, sumL, 5);
98          and1 = new and(aL, bL, abL, 3);
99          and2 = new and(axbL, ciL, abcL, 3);
100         or1 = new or(abL, abcL, coL, 3);
101         };
102         ~fullAdder();
103         void evl();
104 };
105
100 %
```

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

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

```
#include "timedLogicPrimitives.h"
#include "timedLogicFunctions.h"

void inpBit(string, wire&);
void outBit(string, wire);
```

timedLogicUtilities.cpp

```
#include "timedLogicUtilities.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";
}
```

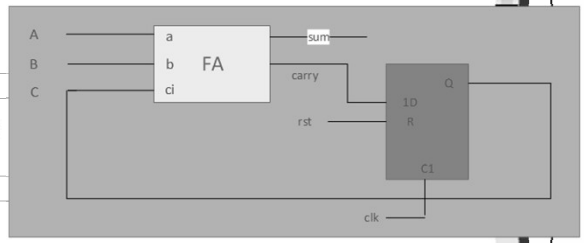
Gate Classes with Power and Timing Calculation

- *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
```

timedLogicFunctions.cpp

```
Timed Logic Classes | (Global Scope)
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
```



Wire and Gate Vectors

- *wireV* type pointer-based logic classes
  - AND gate

timedVectorLogicPrimitives.h

Main difference with *wire*

```
timedVectorLogicFunctions.cpp
Timed Vector Logic Classes | halfAdder
133
134 class wireV {
135 public:
136     char* value;
137     int n; //bits
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
100 %
```

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

timedVectorLogicPrimitives.cpp

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

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

```
timedVectorLogicUtilities.cpp
Timed Vector Logic Classes | orV
156 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::evl() {
177     int i = 0;
178
179     while (i1->value[i] != '\0'){
        if (((i1->value[i]) == '0') || ((i2->value[i]) == '0'))
            o1->value[i] = '0';
        else if ((i1->value[i] == '1') && (i2->value[i] == '1'))
            o1->value[i] = '1';
        else
            o1->value[i] = 'X';
        i++;
    };
187 }
188
100 %
```

Wire and Gate Vectors

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

Utility for individual wires

Utility for vector wires

timedVectorLogicPrimitives.h

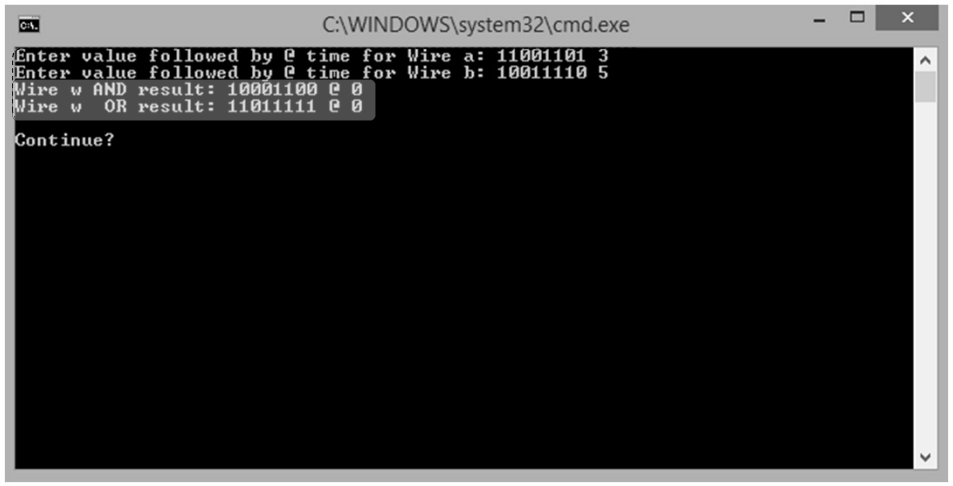
timedVectorLogicUtilities.cpp | timedVectorLogicPrimitives.h

Timed Vector Logic Classes | (Global Scope)

```
1  #include "timedVectorLogicUtilities.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 }
17
18 void inpBit(string wireName, wireV& valtim) {
19    string value;
20    int time;
21    cout << "Enter value followed by @ time for " << wireName << ": ";
22    cin >> value; cin >> time;
23    valtim.put(value, time);
24 }
25
26 void outBit(string wireName, wireV valtim) {
27    string value;
28    int time;
29    valtim.get(value, time);
30    cout << wireName << ": " << value << " @ " << time << "\n";
31 }
```

Wire and Gate Vectors

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



timedVectorLogicFunctions.cpp

```
timedVectorLogicUtilities.cpp   timedVectorLogicPrimi...
Timed Vector Logic Classes      (Global Scope)
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
100 %
```

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

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 use 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

InheritedLogicClassPrimitives.h

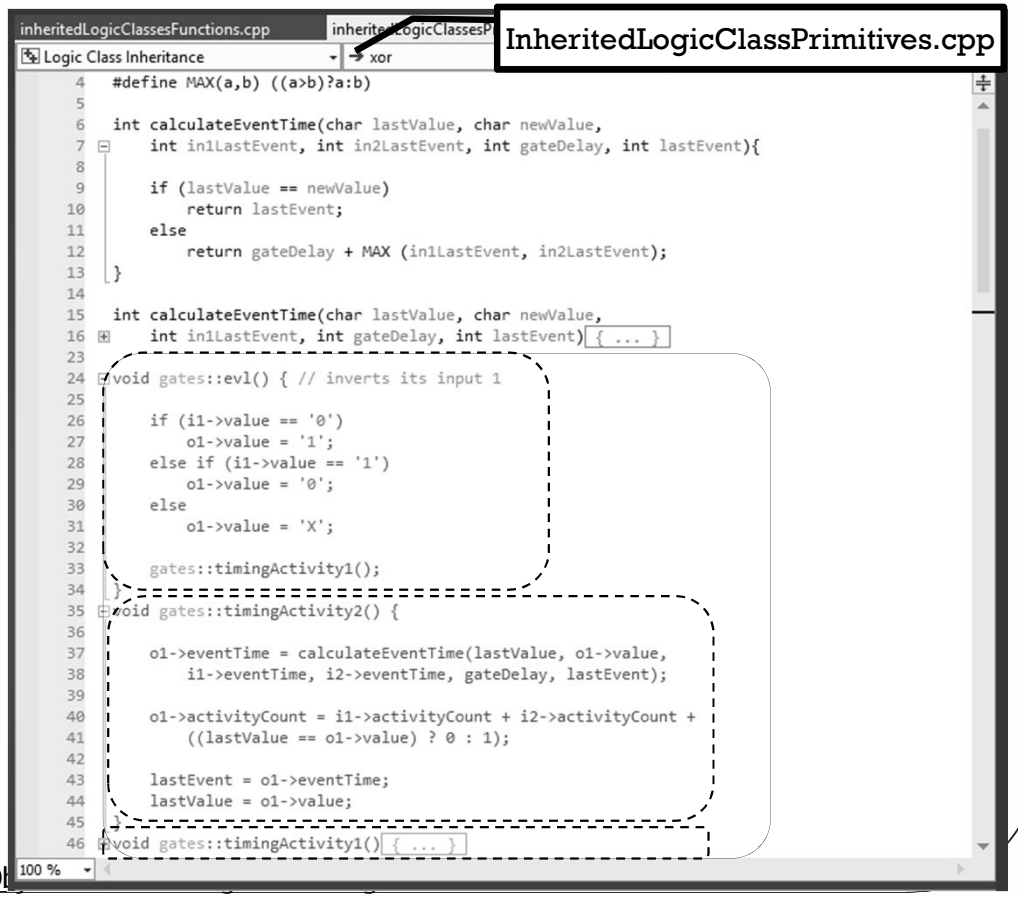
```
inheritedLogicClassesFunctions.cpp
Logic Class Inheritance
16  class gates {
17  protected:
18      wire *i1, *i2, *o1;
19      int gateDelay, lastEvent;
20      char lastValue;
21  public:
22      gates(wire& a, wire& w, int d) :
23          i1(&a), o1(&w), gateDelay(d) {}
24      gates(wire& a, 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  };
100 %
```

All gates are inherited from the *gates* class

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

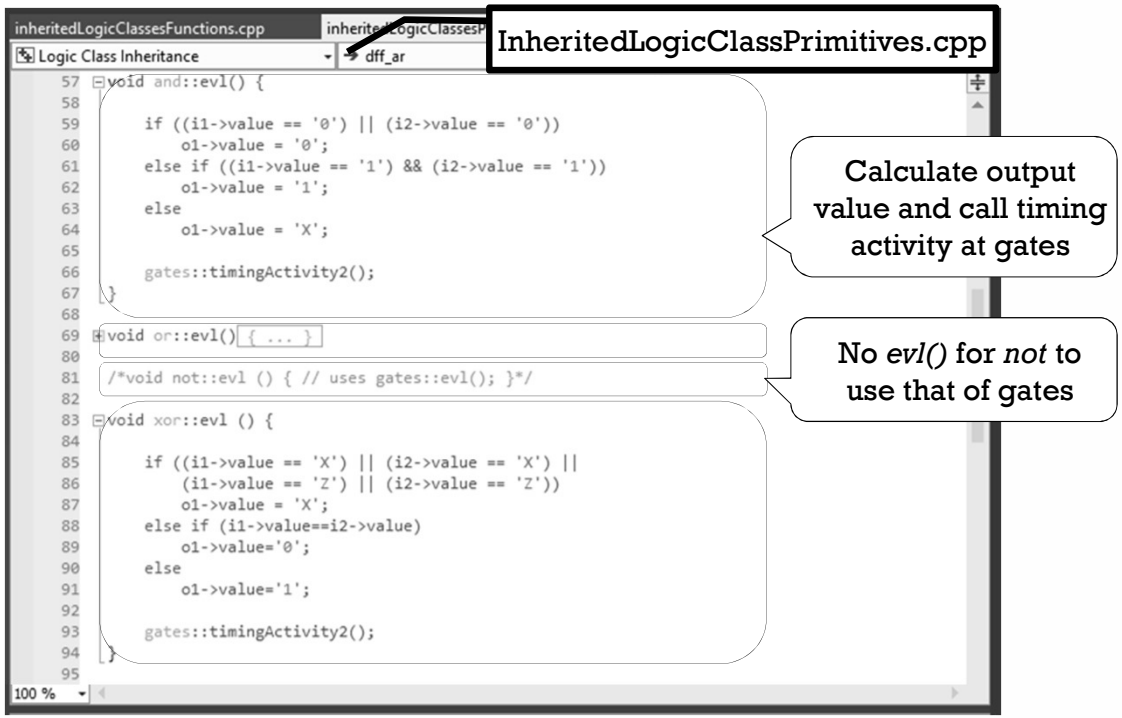
A generic gate definition

- Inheritance-based logic classes (cont.)



Gates to include timing

- Inheritance-based logic classes (cont.)
  - AND gate
  - OR gate
  - NOT gate
  - XOR gate

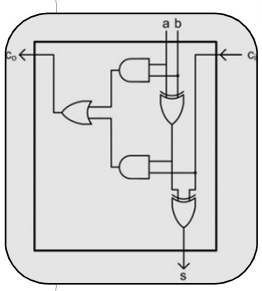


Building Structures from Objects

- 1-bit full adder

inheritedLogicClassPrimitives.cpp

```
inheritedLogicClassesFunctions.cpp
Logic Class Inheritance
→ xor
116
117 void fullAdder::evl () {
118
119     // Via the FA pointers, read wire values that connect to
120     // the FA from outside, and assign them to FA Local wires
121     aL = *i1; bL = *i2; ciL = *i3;
122     and1->timingActivity1();
123     // Evaluate gates in the proper order
124     xor1->evl();
125     and1->evl();
126     and2->evl();
127     or1->evl();
128     xor2->evl();
129
130     // Take calculated local wire values and assign the values
131     // to the outside wires via pointers of FA
132     *o1 = coL; *o2 = sumL;
133 }
100 %
```



inheritedLogicClassPrimitives.h

```
inheritedLogicClassesFunctions.cpp
Logic Class Inheritance
(Global Scope)
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 };
100 %
```

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

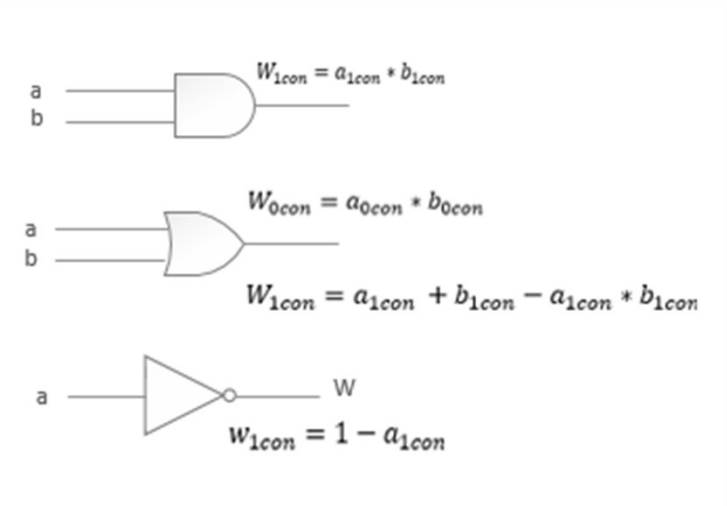
Types and Operators for Logic Modeling

+ 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

Wire Functionality

- Logic Testability Analysis



Wire Functionality

- *wire* class has wire identifier and static number of wires

Ref. [1] – Static members - P. 98-99

Only a copy of it is generated for every instance of wire

Any new wire increments number of wires

PolymorphismLogicClassesPrimitives.h

```
polymorphismLogicClassesFunctions.h
Logic Class Polymorphism
 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 %
```

Gate Functionality

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

Ref. [1] – Polymorphism- P. 107-109

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), 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 evl();
48      virtual void prob(){};
49  };
50
100 %
```

Gates constructor assigns an id and increments the gate count

Overloading Constructor

Virtual member functions

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

Gate Functionality (cont.)

Static initialization must be done

Like a one input buffer

Timing activity functions for 2 and 1 input gates

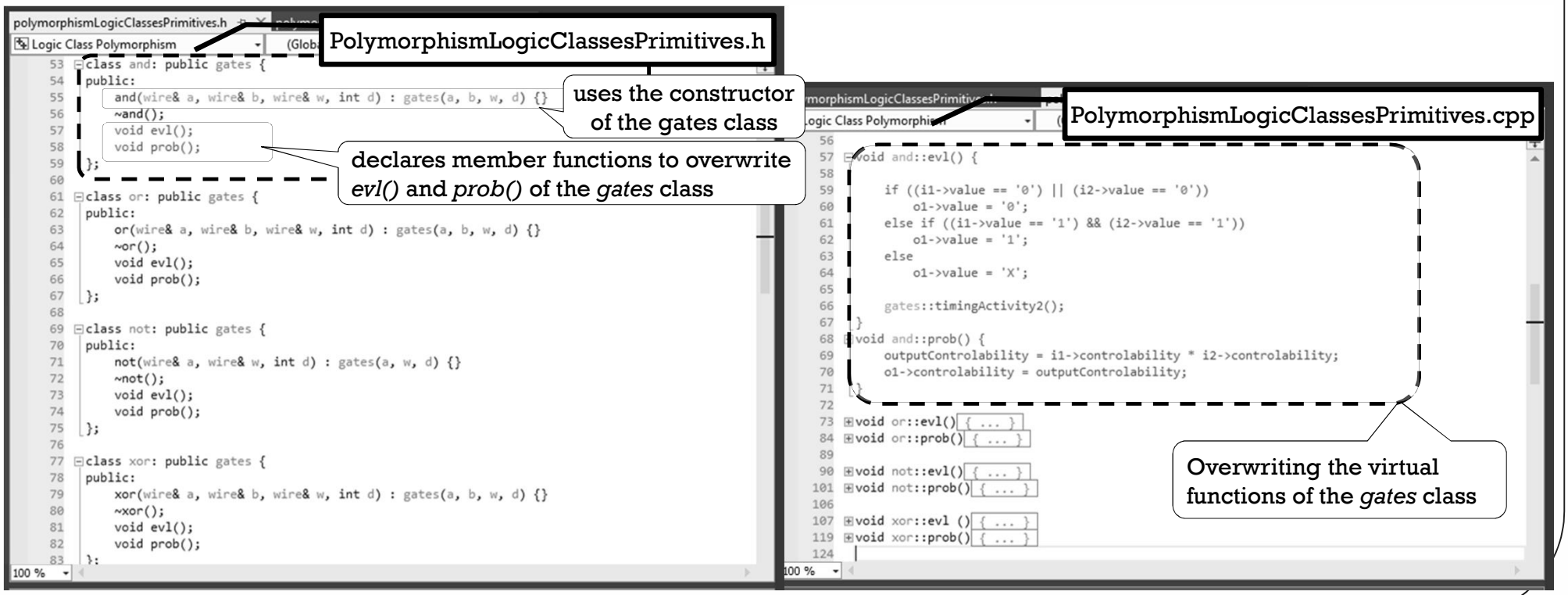
Static initialization must be done

PolymorphismLogicClassesPrimitives.cpp

```
23
24  int wire::numberOfWires = 1;
25
26  void gates::evl() { // puts input 1 on output
27      o1->value = i1->value;
28      gates::timingActivity1();
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  void gates::timingActivity1() {
42
43      o1->eventTime = calculateEventTime(lastValue, o1->value,
44          i1->eventTime, gateDelay, lastEvent);
45
46      o1->activityCount = i1->activityCount + ((lastValue == o1->value)?0:1);
47
48      lastEvent = o1->eventTime;
49      lastValue = o1->value;
50  }
51  int gates::numberOfGates=1;
52
53  float getProb(gates* GATE){
54      return GATE->outputControlability;
55  }
56
57  void and::evl() {
58
59      if ((i1->value == '0') || (i2->value == '0'))
```

Polymorphic Gate Base

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



Polymorphic Gate Base (cont.)

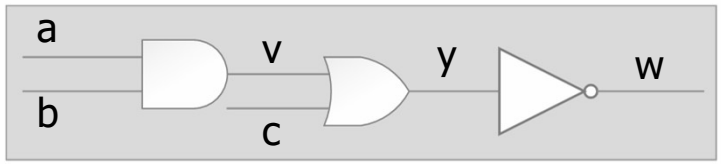
- Overloading *evl()* function

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

PolymorphismLogicClassesPrimitives.cpp

Polymorphic Gate Base (cont.)

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



Pointers to base class (gates)

Calling the overloaded *evl()* function

PolymorphismLogicClassesFunctions.cpp

```
Logic Class Polymorphism
73
74  int main()
75  {
76      wire a('0', 3), b('1', 5), c('X', 0);
77      wire v('0', 3), w('0', 3), y('1', 5);
78
79      gates *NOT = new not(y, w, 5);
80      gates *AND = new and(a, b, v, 7);
81      gates *OR1 = new or(v, c, y, 6);
82
83      AND->prob();
84      OR1->prob();
85      NOT->prob();
86
87      int ai; int time = 0;
88
89      do {
90          inpBit("Wire a", a, time);
91          inpBit("Wire b", b, time);
92          inpBit("Wire c", c, time);
93
94          cout << evl(AND) << " :AND\n";
95          cout << evl(OR1) << " :OR1\n";
96          cout << evl(NOT) << " :NOT\n";
97
98          outBit("AOI  output: ", w);
99          cout << "AOI  output activity count: " << w.activity() << '\n';
100
101         time += 17;
102         cout << "\n" << "Continue? "; cin >> ai; cout << "\n";
103     } while (ai>0);
104 }
105
106
107 /*
108 int main()
109 {
100 %
```

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

Abstract class

PolymorphismLogicClassesPrimitives.h

```
84
85  class flipflop {
    protected:
        wire *D, *clk, *rst, *cen, *Q;
        int clkQDelay;
        int rstQDelay;
        int lastEvent; // last time output changed
91      char lastValue;
92      bool containsReset = false;
93      float clockControlability = 0.5;
94      static int numberOfFlipflops;
95  public:
96      int flipflopIdentifier;
97      float outputControlability = 1.0;
98      flipflop(wire& d, wire& c, wire& q, int dC) :
99          D(&d), clk(&c), Q(&q), clkQDelay(dC) {
100         flipflopIdentifier = numberOfFlipflops;
101         numberOfFlipflops++;
102     };
103     ~flipflop(){};
104     virtual void evl() = 0;
105     virtual void prob() = 0;
106     virtual void init(float, char) = 0;
107 };
108
```

Pure virtual functions

Flip Flop Description Hierarchies

- DFF class
  - is inherited from *flipflop* class

First level derivation

PolymorphismLogicClassesPrimitives.h

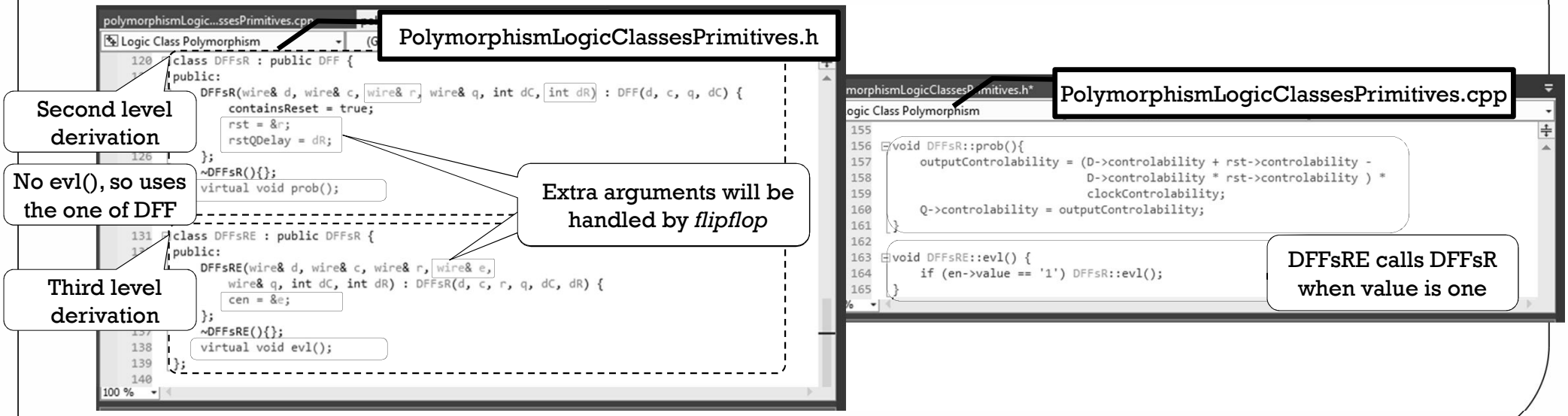
```
polymorphismLogicClassesPrimitives
Logic Class Polymorphism
109  class DFF : public flipflop {
     public:
         DFF(wire& d, wire& c, wire& q, int dC) : flipflop(d, c, q, dC)
         { containsReset = false; };
         ~DFF(){};
         virtual void evl();
         virtual void prob();
         virtual void init(float, char);
116  };
117
100 %
```

PolymorphismLogicClassesPrimitives.cpp

```
polymorphismLogicClassesPrimitives.h*
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  void DFF::prob(){
149      outputControlability = D->controlability * clockControlability;
150      Q->controlability = outputControlability;
151  }
152  void DFF::init(float clkCon, char iniOut) {
153      clockControlability = clkCon; Q->value = iniOut;
154  }
155
100 %
```

Flip Flop Description Hierarchies

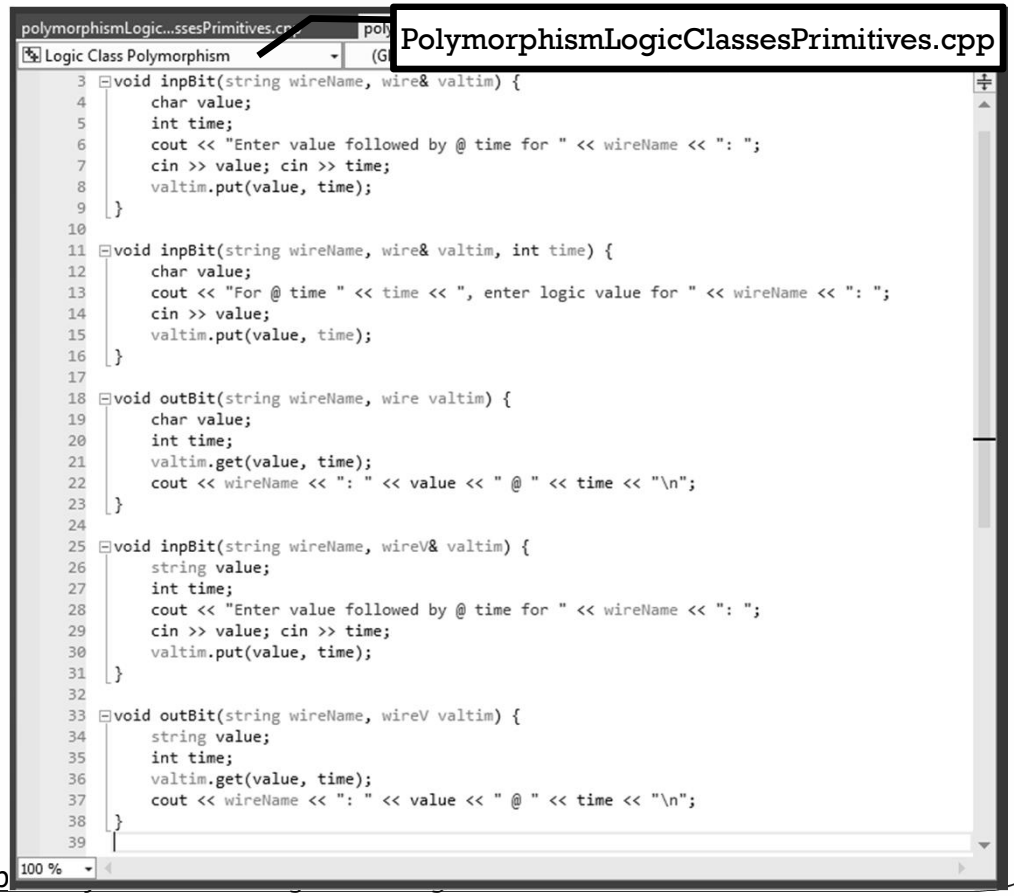
- DFF with Synchronous Reset (*DFFsR*) class
  - is inherited from *DFF* class
- DFF with Synchronous Reset and Enable (*DFFsRE*) class
  - is inherited from *DFFsR* class



Flip Flop Description Hierarchies

- Utility Functions

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
```

Flip Flop Description Hierarchies

PolymorphismLogicClassesFunctions.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
100 %
```

PolymorphismLogicClassesFunctions.cpp

```
Logic Class Polymorphism
39
40     int ai; int time = 0;
41
42     do {
43         inpBit("Wire a", a, time);
44         inpBit("Wire b", b, time);
45         inpBit("Wire c", c, time);
46         inpBit("Clock input", clk, time);
47         inpBit("Reset input", rst, time);
48         inpBit("Enable input", en, time);
49
50         AND->evl();
51         OR1->evl();
52         NOT->evl();
53         FF1->evl();
54         FF2->evl();
55         FF3->evl();
56
57         outBit("AOI  output: ", w);
58         outBit("DFF1 output: ", Q1);
59         outBit("DFF2 output: ", Q2);
60         outBit("DFF3 output: ", Q3);
61
62         cout << "AOI  output activity count: " << w.activity() << '\n';
63         cout << "DFF1 output activity count: " << Q1.activity() << '\n';
64         cout << "DFF2 output activity count: " << Q2.activity() << '\n';
65         cout << "DFF3 output activity count: " << Q3.activity() << "\n\n";
66
67         time += 17;
68         cout << "\n" << "Continue? "; cin >> ai; cout << "\n";
69
70     } while (ai>0);
71 }
72 /*
73 int main()
74 {
75     wire aW('0', 3), bW('1', 5), ciW('X', 0), coWF('X', 0),
100 %
```

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
- Hierarchal 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

- Slides prepared by Hanieh Hashemi, ECE graduate student
- Slides updated by Nooshin Nosrati, ECE Ph.D. student