

## ECE380 Digital Logic

### Number Representation and Arithmetic Circuits: Design of Arithmetic Circuits Using CAD Tools

## Design using schematic capture

- One way to design an arithmetic circuit is via schematic capture, drawing all necessary logic gates
- To create an  $n$ -bit adder
  - Start with a single full adder
  - Chain together  $n$  instances of this to produce the  $n$ -bit adder
  - If a CLA adder is desired, add carry lookahead logic
- Design process becomes complex rapidly
- A better approach is to use predefined subcircuits
  - CAD tools provide a library of basic logic gates
  - Most CAD tools also provide a library of commonly used circuits, such as adders
  - Each subcircuit is provided as a module that can be imported into a schematic and used as a part of a larger circuit

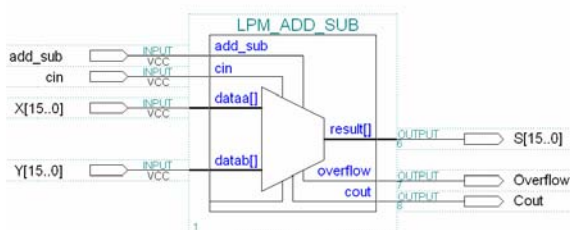
## Macro- and megafunctions

- In some CAD systems (such as Altera's MAX+PLUS2), these library functions are called **macrofunctions** or **megafunctions**
- Two primary types of macrofunctions:
  - Technology-dependent: designed to suit a specific type of chip (such as a particular FPGA)
  - Technology-independent: implemented in any type of chip, with different circuits for different types of chips
- A good example of a library of macrofunctions is the **Library of Parameterized Modules (LPM)** as a part of the MAX+PLUS2 system
  - Each module is technology independent
  - Each module is parameterized: it can be used in a variety of ways

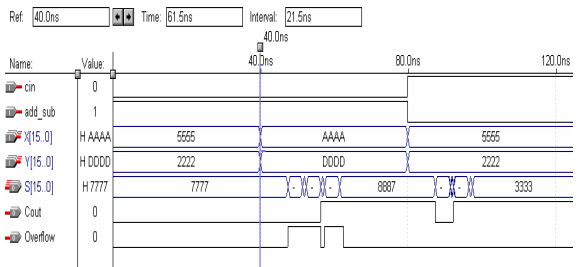
## LPM\_ADD\_SUB

- The LPM library includes an  $n$ -bit adder named **LPM\_ADD\_SUB**
  - Implements a basic add/subtract circuit
  - The number of bits,  $n$ , is set by a parameter **LPM\_WIDTH**
  - Another parameter **LPM\_REPRESENTATION**, determines whether the numbers are treated as unsigned or signed

## An adder using LPM\_ADD\_SUB



## Adder simulation



## Design using VHDL

- We can use a hierarchical approach in designing VHDL code
  - First construct a VHDL entity for a full adder
  - Use multiple **instances** to create a multi-bit adder
- In VHDL, a logic signal is represented as a data object
  - We used a **BIT** data type before that could only take on the values 0 and 1
  - Another data type, **STD\_LOGIC**, is actually preferable because it can assume several different values [0, 1, Z (high impedance), - (don't care)]
- We must declare the library where the data type exists, and declare that we will use the data type

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
```

## VHDL full adder

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY fulladd IS
    PORT ( Cin, x, y : IN  STD_LOGIC ;
          s, Cout   : OUT STD_LOGIC ) ;
END fulladd ;

ARCHITECTURE LogicFunc OF fulladd IS
BEGIN
    s <= x XOR y XOR Cin ;
    Cout <= (x AND y) OR (Cin AND x) OR (Cin AND y) ;
END LogicFunc ;
```

## VHDL 4-bit ripple carry adder

### ENTITY construct

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY adder4 IS
    PORT ( Cin           : IN  STD_LOGIC ;
          x3, x2, x1, x0 : IN  STD_LOGIC ;
          y3, y2, y1, y0 : IN  STD_LOGIC ;
          s3, s2, s1, s0 : OUT STD_LOGIC ;
          Cout           : OUT STD_LOGIC ) ;
END adder4 ;
```

## VHDL 4-bit ripple carry adder

### ARCHITECTURE construct

```
ARCHITECTURE Structure OF adder4 IS
    SIGNAL c1, c2, c3 : STD_LOGIC ;
    COMPONENT fulladd
        PORT ( Cin, x, y : IN  STD_LOGIC ;
              s, Cout   : OUT STD_LOGIC ) ;
    END COMPONENT ;
BEGIN
    stage0: fulladd PORT MAP ( Cin, x0, y0, s0, c1 ) ;
    stage1: fulladd PORT MAP ( c1, x1, y1, s1, c2 ) ;
    stage2: fulladd PORT MAP ( c2, x2, y2, s2, c3 ) ;
    stage3: fulladd PORT MAP (
        Cin => c3, Cout => Cout, x => x3, y => y3, s => s3 ) ;
END Structure ;
```

## New VHDL syntax

- There are several new constructs in the previous VHDL code
- **SIGNAL c1, c2, c3 : STD\_LOGIC ;**
  - Appears in the ARCHITECTURE construct
  - Basically defines signals that will be used internal to the design (i.e. not specifically an IN or an OUT signal as appears in the PORT statement)
- **COMPONENT fulladd**
  - Appears in the ARCHITECTURE construct
  - Defines the PORT for a subcircuit (component) that is defined in another file (fulladd.vhd in this example)
  - The VHDL file (fulladd.vhd) should normally be in the same directory as the file adder4.vhd

## New VHDL syntax

- **stage0: fulladd PORT MAP ( Cin, x0, y0, s0, c1 ) ;**
  - Defines an instance of the component fulladd named stage0
  - Uses **positional association** because the inputs and outputs listed in the PORT MAP appear in the exact same order as in the COMPONENT statement
- **stage3: fulladd PORT MAP ( Cin => c3, Cout => Cout, x => x3, y => y3, s => s3 ) ;**
  - Defines an instance of the component fulladd named stage3
  - Uses **named association** because each input and output listed in the PORT MAP is associated with a specific named signal in the COMPONENT statement

## VHDL packages

- A VHDL package can be created for a component (subcircuit) such that the COMPONENT statement is not explicitly required when creating instances of the component in another file

## VHDL packages

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

PACKAGE fulladd_package IS
  COMPONENT fulladd
    PORT ( Cin, x, y : IN STD_LOGIC ;
          s, Cout : OUT STD_LOGIC ) ;
  END COMPONENT ;
END fulladd_package ;
```

Usually compiled as a separate file in the same directory as fulladd.vhd

## VHDL packages

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.fulladd_package.all ;
```

```
ENTITY adder4 IS
  PORT ( Cin : IN STD_LOGIC ;
        x3, x2, x1, x0 : IN STD_LOGIC ;
        y3, y2, y1, y0 : IN STD_LOGIC ;
        s3, s2, s1, s0 : OUT STD_LOGIC ;
        Cout : OUT STD_LOGIC ) ;
END adder4 ;
```

```
ARCHITECTURE Structure OF adder4 IS
  SIGNAL c1, c2, c3 : STD_LOGIC ;
BEGIN
  stage0: fulladd PORT MAP ( Cin, x0, y0, s0, c1 ) ;
  stage1: fulladd PORT MAP ( c1, x1, y1, s1, c2 ) ;
  stage2: fulladd PORT MAP ( c2, x2, y2, s2, c3 ) ;
  stage3: fulladd PORT MAP (
    Cin => c3, Cout => Cout, x => x3, y => y3, s => s3 ) ;
END Structure ;
```

## Numbers in VHDL

- A number in VHDL is a multibit SIGNAL data object
- **SIGNAL C: STD\_LOGIC\_VECTOR(1 TO 3)**
  - C is a 3-bit STD\_LOGIC signal
    - C - a 3-bit quantity
      - C <= "100";
      - C(1) - a 1-bit quantity (the most significant bit)
      - C(2) - a 1-bit quantity
      - C(3) - a 1-bit quantity (the least significant bit)
- The ordering of the bits can be reversed
- **SIGNAL X: STD\_LOGIC\_VECTOR(3 DOWNTO 0)**
  - X is a 4-bit STD\_LOGIC signal
    - X(3) is the most significant bit
    - X(0) is the least significant bit

## Numbers in VHDL

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE work.fulladd_package.all ;
ENTITY adder4 IS
  PORT ( Cin : IN STD_LOGIC ;
        X, Y : IN STD_LOGIC_VECTOR(3 DOWNTO 0) ;
        S : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) ;
        Cout : OUT STD_LOGIC ) ;
END adder4 ;
```

```
ARCHITECTURE Structure OF adder4 IS
  SIGNAL C : STD_LOGIC_VECTOR(1 TO 3) ;
BEGIN
  stage0: fulladd PORT MAP ( Cin, X(0), Y(0), S(0), C(1) ) ;
  stage1: fulladd PORT MAP ( C(1), X(1), Y(1), S(1), C(2) ) ;
  stage2: fulladd PORT MAP ( C(2), X(2), Y(2), S(2), C(3) ) ;
  stage3: fulladd PORT MAP ( C(3), X(3), Y(3), S(3), Cout ) ;
END Structure ;
```

## Behavioral VHDL descriptions

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;
```

Allows use of STD\_LOGIC signals as signed values

```
ENTITY adder16 IS
  PORT ( X, Y : IN STD_LOGIC_VECTOR(15 DOWNTO 0) ;
        S : OUT STD_LOGIC_VECTOR(15 DOWNTO 0) ) ;
END adder16 ;
```

```
ARCHITECTURE Behavior OF adder16 IS
  BEGIN
    S <= X + Y ;
  END Behavior ;
```

We are really describing the behavior of the circuit

## The VHDL arithmetic package

---

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_arith.all ;
ENTITY adder16 IS
    PORT (Cin           : IN  STD_LOGIC ;
          X, Y          : IN  SIGNED(15 DOWNTO 0) ;
          S             : OUT SIGNED(15 DOWNTO 0) ;
          Cout, Overflow : OUT STD_LOGIC ) ;
END adder16 ;

ARCHITECTURE Behavior OF adder16 IS
    SIGNAL Sum : SIGNED(16 DOWNTO 0) ;
BEGIN
    Sum <= ('0' & X) + Y + Cin ;
    S <= Sum(15 DOWNTO 0) ;
    Cout <= Sum(16) ;
    Overflow <= Sum(16) XOR X(15) XOR Y(15) XOR Sum(15) ;
END Behavior ;
```