Outline

- Arithmetic Functions and Circuits: operate on binary vectors, use the same sub-function in each bit position
  - Adders
  - Multipliers
  - Others
Adder/Subtractor

- Basic building block for Computer-Arithmetic and Digital Signal Processing
- Implementation:
  - Schematic Capture Implementation
  - VHDL Implementation

Adder Design

- Half-adder (HA): a 2-input bitwise addition FB
- Full-adder (FA): a 3-input bit-wise addition FB
- Ripple carry adder: an iterative array to perform binary addition, full adders chained together
- Carry-look-ahead adder: a hierarchical adder to improve performance
  - Propagate and generate logic
Functional Block Implementation

- **Half adder:**
  \[ S = X \oplus Y \]
  \[ C = X \cdot Y \]

- **Full adder:**
  \[ S = X \oplus Y \oplus C_{in} \]
  \[ C = X \cdot Y + (X \oplus Y)C_{in} \]
  \[ = G + P \cdot C_{in} \]
  - \( XY \): carry generate \( G \)
  - \( X \oplus Y \): carry propagate \( P \)

Ripple Carry Adder

- A 4-bit ripple carry adder made from four 1-bit full adder

- Worst case delay - \#bits x full adder delay
  - The longest path is from \( A_0 / B_0 \) through the circuit to \( S_3 \)
  - Or from \( C_0 \) through the circuit to \( C_4 \)
Carry Lookahead Adder

- From the full-adder implementation, two signal conditions: generate \( G \) and propagate \( P \).
  
  \[
  P_i = A_i \oplus B_i \quad S_i = P_i \oplus C_i \quad G_i = A_iB_i \quad C_{i+1} = G_i + P_iC_i
  \]

- In order to reduce the length of the carry chain, \( C_i \) is changed to a more global function spanning multiple cells.
  
  \[
  \begin{align*}
  C_1 &= G_0 + P_0C_0 \\
  C_2 &= G_1 + P_1C_1 = G_1 + P_1(G_0 + P_0C_0) \\
  C_3 &= G_2 + P_2C_2 = G_2 + P_2(G_1 + P_1G_0 + P_1P_0C_0) \\
  &= G_2 + P_2G_1 + P_2P_0C_0 = G_0 + P_0C_0 \\
  C_4 &= G_3 + P_3C_3 = G_3 + P_3(G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0) \\
  &= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0 = G_0 + P_0C_0
  \end{align*}
  \]
VHDL Description of a 4-bit CLA

entity CLA4 is
   port (A, B: in bit_vector (3 downto 0); Ci: in bit;
        S: out bit_vector (3 downto 0); Co, PG, GG: out bit);
end CLA4;

architecture structure of CLA4 is
component CPFullAdder
   port (X, Y, Cin: in bit;
        G, P, Sum: out bit);
end component;
component CLALogic is
   port (G, P: in bit_vector (3 downto 0); Ci: in bit;
        C: out bit_vector (3 downto 1); Co, FG, CG: out bit);
end component;
signal G, P: bit_vector (3 downto 0);
signal C: bit_vector (3 downto 0);
beg
   CarryLogic: CLALogic port map (G, P, Ci, C, Co, PG, GG);
   FA0: CPFullAdder port map (A(0), B(0), Ci, G(0), P(0), S(0));
   FA1: CPFullAdder port map (A(1), B(1), C(1), G(1), P(1), S(1));
   FA2: CPFullAdder port map (A(2), B(2), C(2), G(2), P(2), S(2));
   FA3: CPFullAdder port map (A(3), B(3), C(3), G(3), P(3), S(3));
end structure;

CLALogic

entity CLALogic is
   port (G, P: in bit_vector (3 downto 0); Ci: in bit;
        C: out bit_vector (3 downto 1); Co, PG, GG: out bit);
end CLALogic;
architecture Equations of CLALogic is
signal GG_int, PG_int: bit
begin
   C(1)<=G(0) or (P(0) and Ci);
   C(2)<=G(1) or (P(1) and G(0) or (P(1) and P(0) and Ci);
   C(3)<=G(2) or (P(2) and P(1) and G(0) or (P(2) and P(1) and P(0) and Ci);
   PG_int<=P(3) and P(2) and P(1) and P(0);
   GG_int<=G(3) or (P(3) and G(2)) or (P(3) and P(2) and (G(2)) or (P(3) and P(2) and P(1) and P(0) and G(0));
   Co<=GG_int or (PG_int and Ci);
   PG<=PG_int;
   GG<=GG_int;
end Equations;
**Carry Lookahead Adder Example**

- Extend to 16 bits, to have four 4-bit adders use one of the same carry lookahead circuits
- Delay Specifications
  - NOT=1, XOR=Isolated AND=3, AND-OR=2
- Longest delays:
  - Ripple carry adder: \(=3+15*2+3=36\)
  - CLA=3+3*2+3=12

**Schematic Capture & Macrofunctions**

- **Schematic Capture**
  - Most schematic capture tools include modules as macrofunctions or megafunctions that are useful for implementing some of the highly complex digital hardware such as adders, subtractors, multipliers, and dividers.

- **Macrofunctions may be technology-dependent or technology–independent**
  - A technology-dependent macrofunction is designed to suit a specific type of chip
  - A technology-independent macrofunction can be implemented in any chip
Xilinx Unisim Library of Primitives and Macros

- Xilinx ISE XST system includes a library of primitives and macros called Unisim
- Many modules in the library are technology dependent.
- Most modules are parameterized, implying that it can be used in a variety of ways.
- More modules are available through CoreGenerator

Xilinx Library

- Spartan-II
- Core Generator
  - IP Center on Xilinx.com
  - Third party IP including
    - Communication blocks - ethernet, ATM, encoders, etc.
    - Arithmetic blocks - multipliers, comparators, adders, etc.
    - Microprocessors - PowerPC, 68000, 8051, UARTs, etc.
    - Bus Interfaces - PCI, HyperChannel, etc.
    - Memories - CAMs, RAM interfaces, FIFOs, etc.
- Opencores.org
  - Free, open source VHDL for many cores
16-bit adder/subtractor

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

ENTITY adderlpm IS
    PORT ( Cin : IN STD_LOGIC;
            X, Y : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
            S : OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
            Cout : OUT STD_LOGIC;
            Overflow : OUT STD_LOGIC);
END adderlpm;

ARCHITECTURE addsoft OF adderlpm IS
    BEGIN
        instance: lpm_add_sub
            GENERIC MAP (LPM_WIDTH => 16)
            PORT MAP (add_sub => "0", cin => Cin, dataa => X,
                        datab => Y, result => S, cout => Cout, overflow=>Overflow);
END addsoft;
```

VHDL for 16-bit adder

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

ENTITY adderpm IS
    PORT ( Cin : IN STD_LOGIC;
            X, Y : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
            S : OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
            Cout : OUT STD_LOGIC;
            Overflow : OUT STD_LOGIC);
END adderpm;

ARCHITECTURE addsoft OF adderpm IS
    BEGIN
        instance: lpm_add_sub
            GENERIC MAP (LPM_WIDTH => 16)
            PORT MAP (add_sub => '0', cin => Cin, dataa => X,
                        datab => Y, result => S, cout => Cout, overflow=>Overflow);
END addsoft;
```
Subtraction

- **Subtraction (A-B)**
  - Unsigned:
    - \( A \geq B \Rightarrow A - B \)
    - \( A < B \Rightarrow \) the difference \( A - B + 2^n \) is subtracted from \( 2^n \), a "-" sign added before the result (\( 2^n - X \) is taking the 2’s complement of \( X \))
  - Signed integer:
    - For binary numbers \( a_n a_{n-1} \ldots a_2 a_1 a_0 \)
      - \( s = 0 \) for positive numbers;
      - \( s = 1 \) for negative numbers
    - Signed-magnitude: the \( n-1 \) digits are a positive magnitude
    - Signed 2’s complement
2’s Complement Adder/Subtractor

- Subtraction can be done by adding 2’s complement

- For \( s=1 \), subtract, the 2’s complement of B is formed by using XORs to form the 1’s complement and adding the 1 applied to \( C_0 \)
- For \( s=0 \), add, B is passed through unchanged

VHDL for adder/subtractor

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

ENTITY addsubtract IS
PORT ( S : IN STD_LOGIC;
       A, B : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
       SO : OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
       Cout : OUT STD_LOGIC);
END adderpm;

ARCHITECTURE behavior OF addsubtract IS
signal Sum : STD_LOGIC_VECTOR (4 downto 0);
BEGIN
with S select
Sum <= A+B when '0'
A-B+"10000" when others;
Cout <= Sum (4);
SO <= Sum (3 downto 0);
END behavior;
```
Multiply

- Multiply requires shifting the multiplicand to the left adding it to the partial sum.
- Requires a shift register as wide as the product and an accumulator for the partial and final product.

\[
\begin{align*}
9 &= \text{1001 (multiplicand)} \\
x\text{13} &= \times \text{1101 (multiplier)} \\
0001 \\
+0000 \\
= 01001 \\
+0001 \\
= 101101 \\
+0001 \\
117 &= 1110101
\end{align*}
\]

Multiplier

- Here is a hardware description of a multiplier

- If B[0] is 1, load (add # to accumulator)
- Shift multiplicand reg left
- Shift multiplier reg right -> B[0]
- Repeat until done
Multiplier

```
library unisim;
use unisim.vcomponents.all;

architecture beh of mult is
component MULT18X18
port(
 A : in std_logic_vector (17 downto 0);
 B : in std_logic_vector (17 downto 0);
 P : out std_logic_vector (35 downto 0)
);
end component;

begin
  U_MULT18X18 : MULT18X18
  port map (A,B,P);
end
```
Division

- Division macros are not part of the Unisim library
- Multiply and Divide by power of 2 can be accomplished with a shifter
Other Arithmetic Functions

- **Overflow detection**: overflow occurs if \( n+1 \) bits are required to contain the results from an \( n \)-bit addition or subtraction
- **Incrementing**: counting up, \( A+1, B+4 \)
- **Decrementing**: counting down, \( A-1, B-4 \)
- **Multiplication by constant**: left shift
- **Division by constant**: right shift
- **Zero fill**: filling zero either at MSB or LSB end
- **Extension**: copy the MSB of the operand into the new positions

Design by Contraction

- Contraction is a technique for simplifying the logic in a functional block to implement a different function
- Example: contraction of a ripple-carry adder to increment for 3 bits (set \( B=001 \))

![Diagram of Design by Contraction](image)