Adder Design

- Basic building block is a full adder
- Chained together as a ripple carry adder
- Carry lookahead adder is an other option
  - Propagate and generate logic
Ripple Carry Adder

- Worst case delay - \( \text{#bits} \times \text{full adder delay} \)
Generate Statements

- There are two variants of the generate statement:
  - **FOR GENERATE** statement
    - Seldom used.
  - **IF GENERATE** statement
    - Frequently used.
- **FOR GENERATE** statement provides a convenient way of repeating either a logic equation or a component instantiation.
Generate Statements Formats

- The syntax for the `GENERATE` statement is:

  \[
  \text{Label : generation\_scheme GENERATE} \\
  \text{[concurrent\_statements]} \\
  \text{END GENERATE label;}
  \]

- `Generation\_scheme ::=`:
  - `FOR generate\_specification`
  - Or
  - `IF condition`

The beginning delimiter for the `generate` statement is denoted by the keyword `GENERATE`. The ending delimiter is marked by the keywords `END GENERATE`.

A label is required for the `generate statement` and is optional when used with the `END GENERATE` statement. If the label is used with the `END GENERATE` statement it must match generate label.
Example

A 16-bit wide bus is to be connected to a 16-bit register. Create such a bus from 16 string of wires using the GENERATE VHDL construct?

```
LIBRARY ieee;
USE WORK.ECE252_package.all;

ENTITY reg16 IS
  PORT (input : IN STD_LOGIC_VECTOR (0 to 15);
        clock : IN STD_LOGIC;
        output : OUT STD_LOGIC_VECTOR (0 to 15);
  END reg16;
```
ARCHITECTURE bus16_wide OF reg16 IS
 COMPONENT dff
   PORT ( d, clk : IN STD_LOGIC,
          q      : OUT STD_LOGIC);
 END COMPONENT;
BEGIN

G1 : FOR i IN 0 to 15 GENERATE
    dff1: dff PORT MAP (input (i), clock, output(i));
END GENERATE G1;
END bus16_wide;

i - is the counter and does not need to be declared. It will automatically increase by 1 for each loop through the generate statement.

When 16 loops have been completed, generation will stop.
Example

entity adder is
  port( a, b : in bit_vector;
       cin : in bit;
       s : out bit_vector;
       cout : out bit );
end entity adder;
Example

architecture behavioral of adder is
begin
array : for i in 0 to 15 generate
    signal c : bit_vector;
begin
first : if i=0 generate
begin
    cell : component full_adder
        port map(a(i), b(i), cin, s(i), c(i));
end generate first;
other : if i/=0 generate
begin
    cell : component full_adder
        port_map(a(i), b(i), c(i-1), s(i), c(i));
end generate other;
end generate array;
end architecture behavioral;
Macrofunctions

- 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
VHDL for 16-bit adder

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);
END adderlpm;

ARCHITECTURE addsoft OF adderlpm IS
BEGIN
  instance: lpm_add_sub
    GENERIC MAP (LPM_WIDTH => 16)
    PORT MAP (cin => Cin, dataa => X, datab => Y, result => S, cout => Cout);
END addsoft;
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.
- Not all modules are available on all technologies
- Some modules are inferred and some are instantiated
- 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
Multiply

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

\[ 9 = \begin{array}{c} 1001 \\ \times 13 = x \begin{array}{c} 1101 \\ 1001 \end{array} \\ +0000 \\ = 01001 \\ +1001 \\ = 101101 \\ +1001 \\ = 1110101 \end{array} \]

\[ 117 = 1110101 \]
Multiplier

Here is a hardware description of a multiplier

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

- **Multiplicand Reg**
- **Multiplier Reg**
- **Accumulator**
Multiplier
Multiply VHDL

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
Multiply

• Division macros are not part of the Unisim library

• Multiply and Divide by power of 2 can be accomplished with a shifter
VHDL Functions

- Useful when you want to modularize code

Function inc(a : std_logic_vector(2 downto 0)) return std_logic_vector(2 downto 0) is

Begin
    return a+1;
end
Lab 2

- Design a simple calculator with addition, subtraction, and multiplication
Debouncing switches

• Switches are mechanical and cause glitches on the input
  – Instead of seeing a clean ‘1’ or ‘0’ you may see the signal bounce between ‘1’ and ‘0’ before settling on a value.
  – Can be problematic in certain digital circuits
Asynchronous Inputs

Very simple synchronizer circuit:

This works most of the time (failure rate proportional to system clock).
Asynchronous Inputs: Multiple Synchronizers

Sample an asynchronous signal at one place in your circuit. Otherwise, system might see inconsistent values of input.