EE 459/500 – HDL Based Digital Design with Programmable Logic

Lecture 10
Arithmetic Units

Read before class:

First part of Chapter 4 from textbook

Overview

- Adders/Subtractors
- Multipliers
- Xilinx Unisim

Adders/Subtractors - Integers

- Basic building block for Computer-Arithmetic and Digital Signal Processing
- Operate on binary vectors; use the same sub-function in each bit position

Adder Design

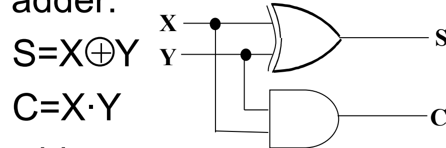
- Functional blocks:
  - Half-Adder (HA): a 2-input bitwise addition
  - Full-Adder (FA): a 3-input bit-wise addition
- Ripple-carry adder: an iterative array to perform binary addition, full adders chained together
- Carry-look-ahead adder: a hierarchical adder to improve performance
  - Propagate and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

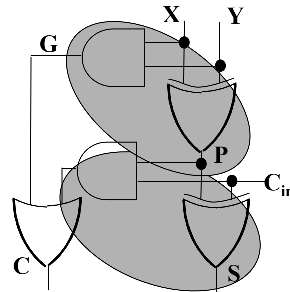
Functional Block Implementation

- Half adder:

  $S = X \oplus Y$

  $C = X \cdot Y$





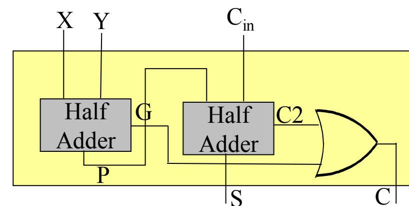
- Full adder:

  $S = X \oplus Y \oplus C_{in}$

  $C = XY + (X \oplus Y)C_{in}$

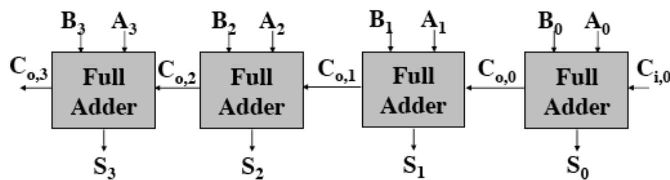
  $= G + P \cdot C_{in}$

  - XY: *carry generate G*
  - $X \oplus Y$: *carry propagate P*

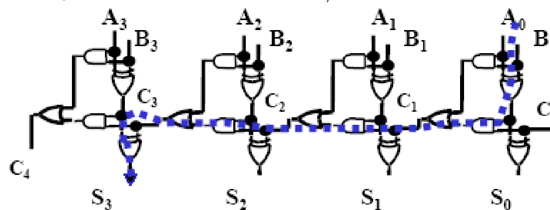


Ripple Carry Adder

- 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.

$$P_i = A_i \oplus B_i \qquad S_i = P_i \oplus C_i$$
$$G_i = A_i B_i \qquad C_{i+1} = G_i + P_i C_i$$

- In order to reduce the length of the carry chain, C_i is changed to a more global function spanning multiple cells

$C_1 = G_0 + P_0 C_0$

$C_2 = G_1 + P_1 C_1 = G_1 + P_1(G_0 + P_0 C_0)$

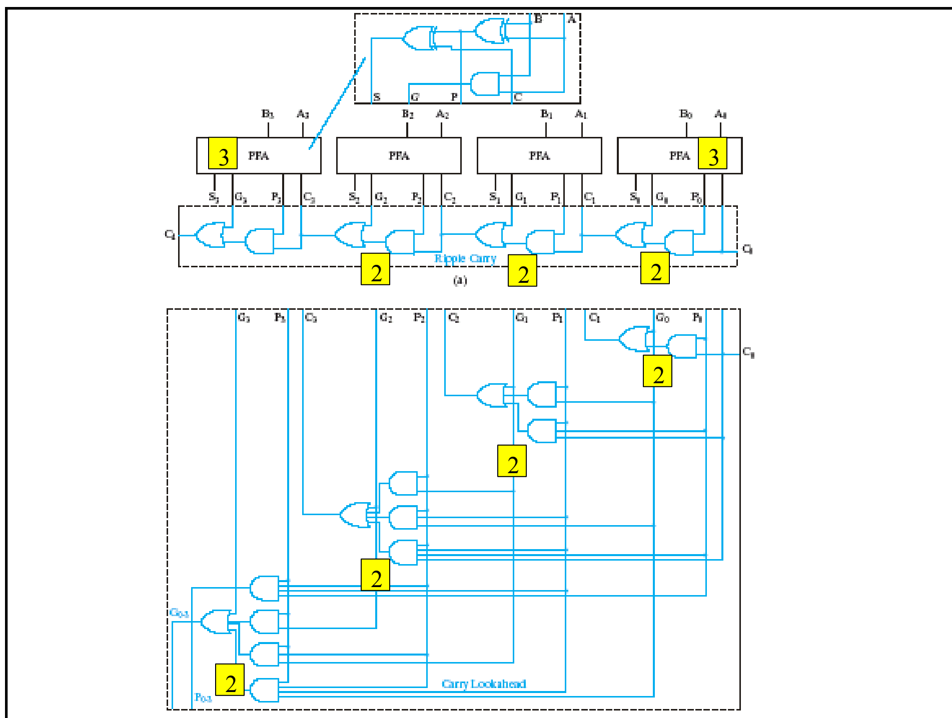
$\quad = G_1 + P_1 G_0 + P_1 P_0 C_0 = G_{0-2} + P_{0-2} C_0$

$C_3 = G_2 + P_2 C_2 = G_2 + P_2(G_1 + P_1 G_0 + P_1 P_0 C_0)$

$\quad = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0 = G_{0-3} + P_{0-3} C_0$

$C4 = G_3 + P_3 C_3 = G_3 + P_3(G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0)$

$\quad = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0 = G_{0-4} + P_{0-4} C_0$



VHDL Description of a 4-bit CLA

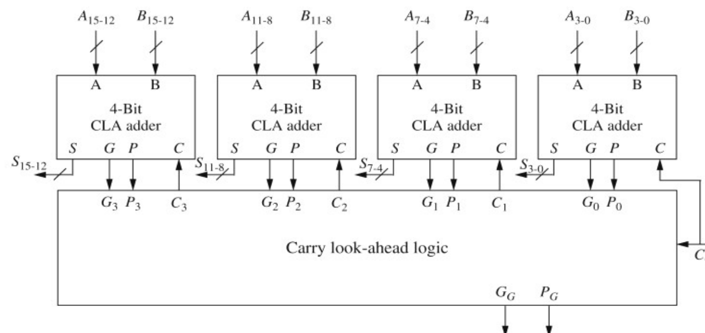
```
entity CLA4 is
   port (A, B: in bit_vector (3 downto 0); Ci: in bit;
         S: out bt_vector (3 downto 0); Co, PG, GG: out bit);
end CLA4;

architecture structure of CLA4 is
 component GPFullAdder
   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: int 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);
begin
   CarryLogic: CLALogic port map (G, P, Ci, C, Co, PG, GG);
   FA0: GPFullAdder port map (A(0), B(0), Ci, G(0), P(0), S(0));
   FA1: GPFullAdder port map (A(1), B(1), C(1), G(1), P(1), S(1));
   FA2: GPFullAdder port map (A(2), B(2), C(2), G(2), P(2), S(2));
   FA1: GPFullAdder 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(1)) 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;
```

16-bit Carry Lookahead Adder

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



Subtraction

- Subtraction (A-B)
  - Unsigned:
    - A≥B => A-B
    - A<B => 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

    s $a_{n-2} \dots a_2 a_1 a_0$

    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

B_3 A_3 B_2 A_2 B_1 A_1 B_0 A_0 S

FA FA FA FA

C_3 C_2 C_1 C_0

C_4 S_3 S_2 S_1 S_0

- 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 code for adder/subtractor

```
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);
        Sout    : OUT  STD_LOGIC_VECTOR (3 DOWNTO 0);
        Cout    : OUT  STD_LOGIC);
END adderlpm;

ARCHITECTURE structural OF addsubtract IS

COMPONENT full_add
    PORT( a, b, c_in : IN STD_LOGIC;
          c_out : OUT STD_LOGIC);
END COMPONENT;

-- Define a signal for internal carry bits
SIGNAL C : STD_LOGIC_VECTOR (4 downto 0);
SIGNAL B_comp : STD_LOGIC_VECTOR (4 downto 1);
```

VHDL code for adder/subtractor

```
-- add/subtract select to carry input (S = 1 for subtract)
C(0) <= S;

adders:
FOR i IN 1 to 4 GENERATE
    --invert B for subtract function (B(i) xor 1,)
    --do not invert B for add function (B(i) xor 0)
    B_comp(i) <= B(i) xor S;
    adder: full_add PORT MAP (A(i),B_comp(i),C(I -1),C(i),Sout(i));
END GENERATE;

Cout <= C(4);

END structural;
```

VHDL code for adder/subtractor

```
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);
       Sout    : OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
       Cout    : OUT STD_LOGIC);
END adderlpm;

ARCHITECTURE behavioral 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);
Sout <= Sum(3 downto 0);
END behavioral;
```

Overview

- Adders/Subtractors
- Multipliers
- Xilinx Unisim

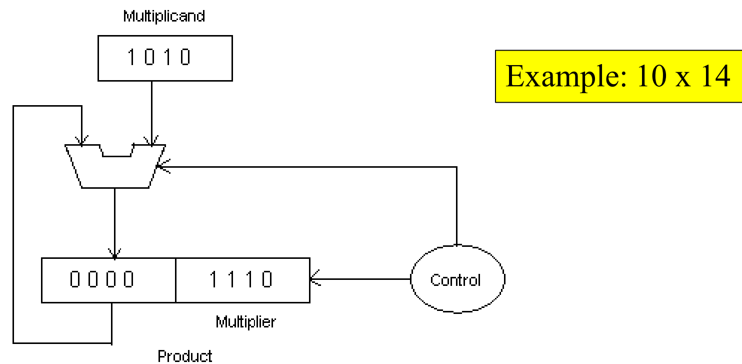
Multiplication

- 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.

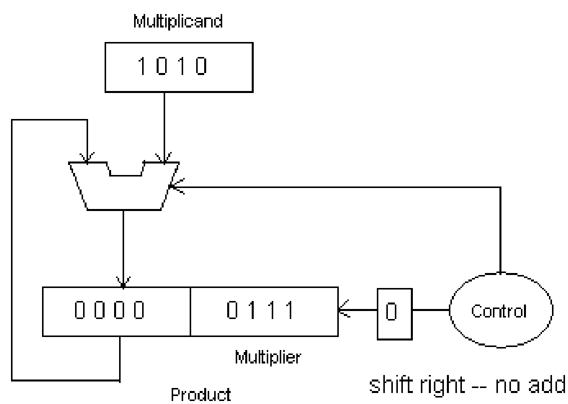
```
Multiplicand ------->  1 1 0 1  (13)
Mutliplier   ------->  1 0 1 1  (11)
                      ---------
                       1 1 0 1
                     1 1 0 1
                    -----------
Partial            1 0 0 1 1 1
products         0 0 0 0
                 -------------
                   1 0 0 1 1 1
                 1 1 0 1
               ---------------
               1 0 0 0 1 1 1 1  (143)
```

Add-and-Shift Multiplier

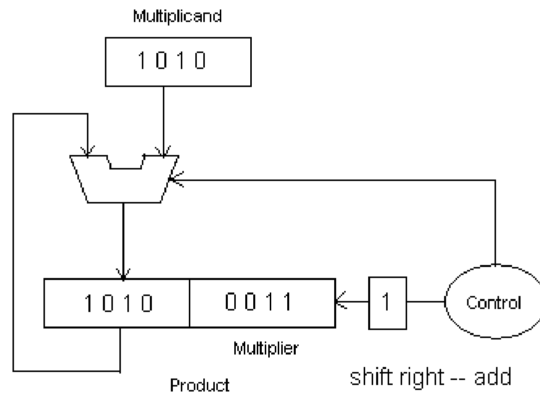
- Place the multiplier in the rightmost 4 bits of the 8-bit product register



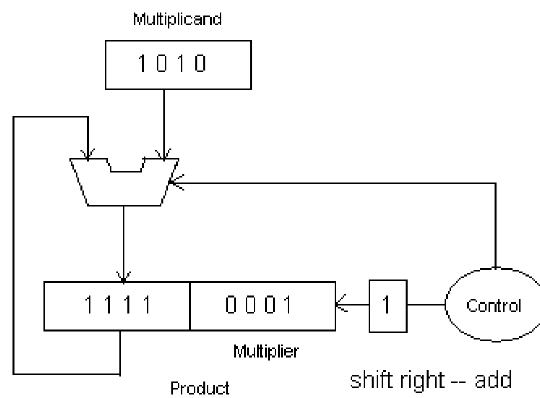
- Shift 1 -- The bit shifted out of the product register is 0. No add is performed.



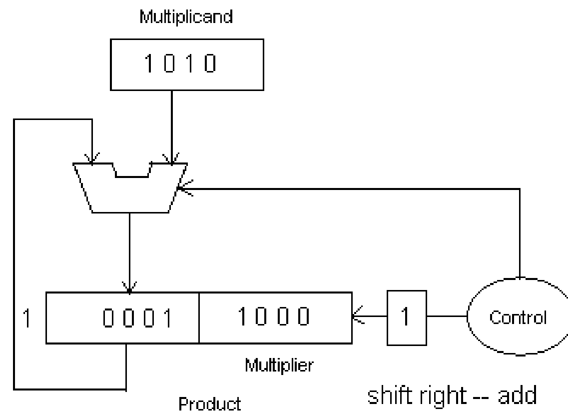
- Shift 2 -- The bit shifted out of the product register is 1. Add the multiplicand to the first 4 bits of the product register.



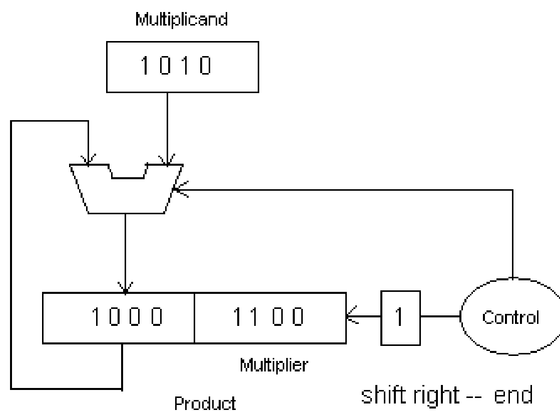
- Shift 3 -- Again add the multiplicand to the leftmost 4 bits of the product register.



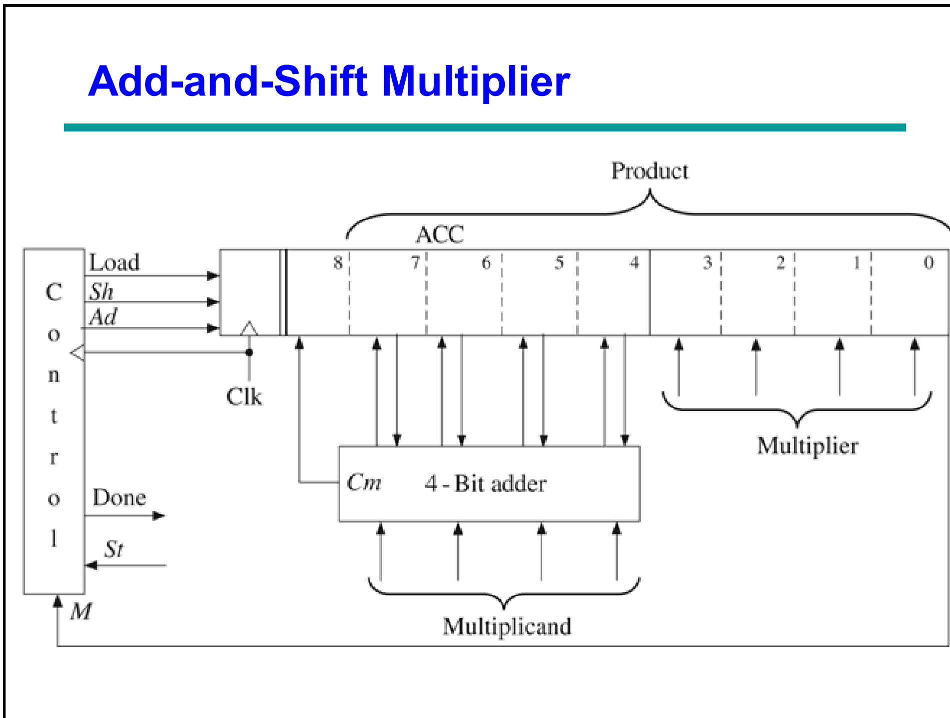
- Shift 4 -- Shift then add.



- Finally, shift right and end. The product is found in the 8-bit product register (140)



10 x 14 = 140



Add-and-Shift Multiplier

initial contents of product register	0 0 0 0 0 ¦ 1 0 1 1	← M (11)
(add multiplicand since $M = 1$)	1 1 0 1	(13)
after addition	0 1 1 0 1 ¦ 1 0 1 1	
after shift	0 0 1 1 0 1 ¦ 1 0 1	← M
(add multiplicand since $M = 1$)	1 1 0 1	
after addition	1 0 0 1 1 1 ¦ 1 0 1	
after shift	0 1 0 0 1 1 1 ¦ 1 0	← M
(skip addition since $M = 0$)		
after shift	0 0 1 0 0 1 1 1 ¦ 1	← M
(add multiplicand since $M = 1$)	1 1 0 1	
after addition	1 0 0 0 1 1 1 1 ¦ 1	
after shift (final answer)	0 1 0 0 0 1 1 1 1 ¦	(143)

dividing line between product and multiplier

State Graph

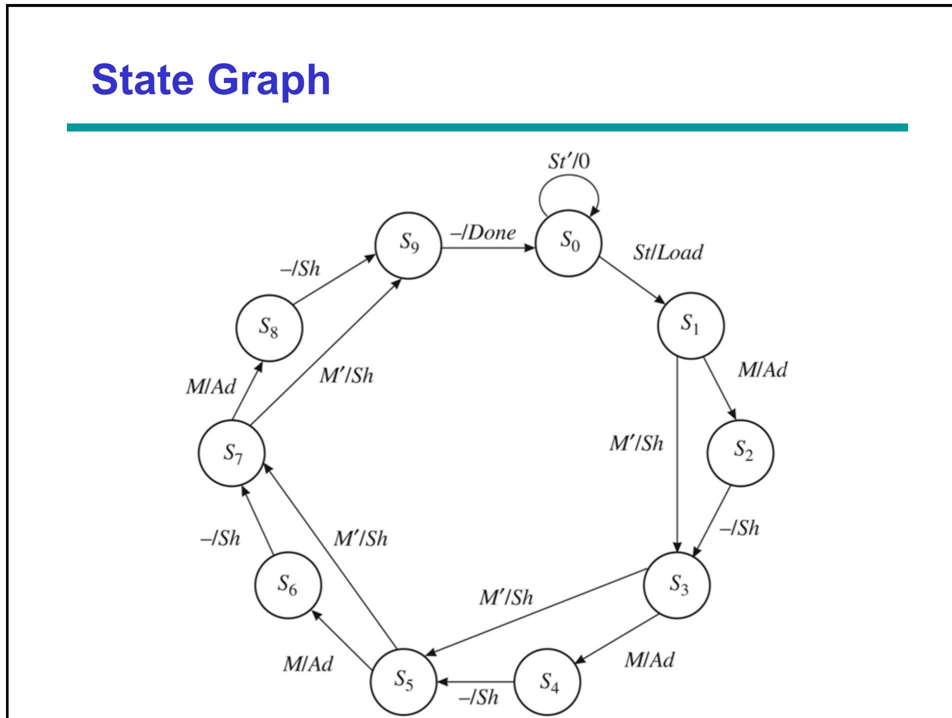


FIGURE 4-27: **Behavioral Model for 4 × 4 Binary Multiplier**

VHDL code for 4-bit binary multiplier

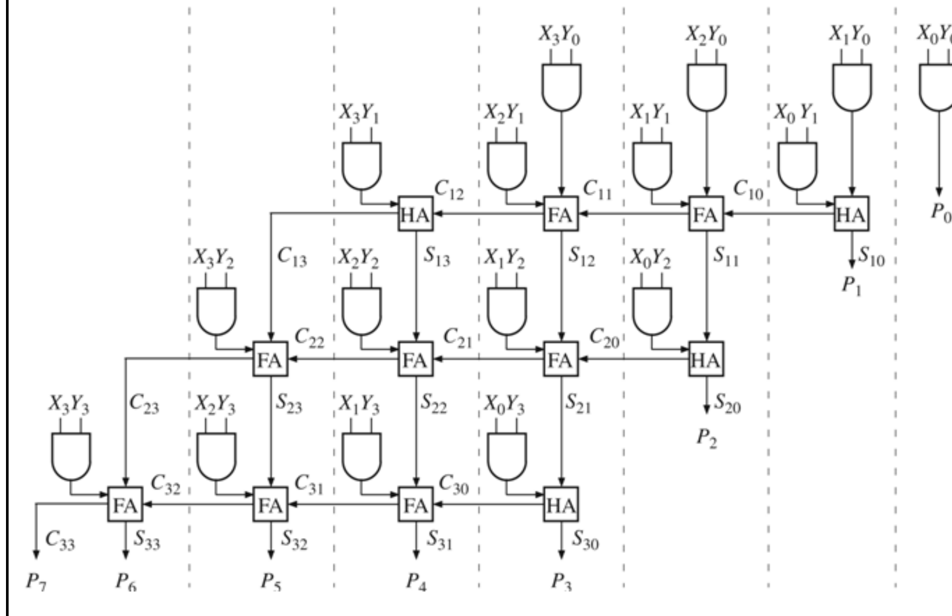
```
-- This is a behavioral model of a multiplier for unsigned
-- binary numbers. It multiplies a 4-bit multiplicand
-- by a 4-bit multiplier to give an 8-bit product.

-- The maximum number of clock cycles needed for a
-- multiply is 10.

library IEEE;
use IEEE.numeric_bit.all;

entity mult4X4 is
  port(Clk, St: in bit;
       Mplier, Mcand: in unsigned(3 downto 0);
       Done: out bit);
end mult4X4;

architecture behave1 of mult4X4 is
signal State: integer range 0 to 9;
signal ACC: unsigned(8 downto 0); -- accumulator
alias M: bit is ACC(0);       -- M is bit 0 of ACC
begin
  process(Clk)
  begin
    if Clk'event and Clk = '1' then  -- executes on rising edge of clock
      case State is
        when 0 =>                -- initial State
          if St = '1' then
            ACC(8 downto 4) <= "00000"; -- begin cycle
            ACC(3 downto 0) <= Mplier;  -- load the multiplier
            State <= 1;
          end if;
```

```
        when 1 | 3 | 5 | 7 =>     -- "add/shift" State
          if M = '1' then         -- add multiplicand
            ACC(8 downto 4) <= '0' & ACC(7 downto 4) + Mcand;
            State <= State + 1;
          else
            ACC <= '0' & ACC(8 downto 1);      -- shift accumulator right
            State <= State + 2;
          end if;
        when 2 | 4 | 6 | 8 =>                  -- "shift" State
          ACC <= '0' & ACC(8 downto 1);        -- right shift
          State <= State + 1;
        when 9 =>                              -- end of cycle
          State <= 0;
      end case;
    end if;
  end process;
  Done <= '1' when State = 9 else '0';
end behave1;
```

Array Multiplier

				X_3	X_2	X_1	X_0	Multiplicand
				Y_3	Y_2	Y_1	Y_0	Multiplier
				X_3Y_0	X_2Y_0	X_1Y_0	X_0Y_0	Partial product 0
			X_3Y_1	X_2Y_1	X_1Y_1	X_0Y_1		Partial product 1
			C_{12}	C_{11}	C_{10}			First row carries
		C_{13}	S_{13}	S_{12}	S_{11}	S_{10}		First row sums
		X_3Y_2	X_2Y_2	X_1Y_2	X_0Y_2			Partial product 2
		C_{22}	C_{21}	C_{20}				Second row carries
	C_{23}	S_{23}	S_{22}	S_{21}	S_{20}			Second row sums
	X_3Y_3	X_2Y_3	X_1Y_3	X_0Y_3				Partial product 3
	C_{32}	C_{31}	C_{30}					Third row carries
C_{33}	S_{33}	S_{32}	S_{31}	S_{30}				Third row sums
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0	Final product

Array Multiplier



VHDL code for 4-bit array multiplier

```
entity Array_Mult is
  port(X, Y: in bit_vector(3 downto 0);
       P: out bit_vector(7 downto 0));
end Array_Mult;

architecture Behavioral of Array_Mult is
signal C1, C2, C3: bit_vector(3 downto 0);
signal S1, S2, S3: bit_vector(3 downto 0);
signal XY0, XY1, XY2, XY3: bit_vector(3 downto 0);
component FullAdder
  port(X, Y, Cin: in bit;
       Cout, Sum: out bit);
end component;
component HalfAdder
  port(X, Y: in bit;
       Cout, Sum: out bit);
end component;
begin
  XY0(0) <= X(0) and Y(0); XY1(0) <= X(0) and Y(1);
  XY0(1) <= X(1) and Y(0); XY1(1) <= X(1) and Y(1);
  XY0(2) <= X(2) and Y(0); XY1(2) <= X(2) and Y(1);
  XY0(3) <= X(3) and Y(0); XY1(3) <= X(3) and Y(1);

  XY2(0) <= X(0) and Y(2); XY3(0) <= X(0) and Y(3);
  XY2(1) <= X(1) and Y(2); XY3(1) <= X(1) and Y(3);
  XY2(2) <= X(2) and Y(2); XY3(2) <= X(2) and Y(3);
  XY2(3) <= X(3) and Y(2); XY3(3) <= X(3) and Y(3);

  FA1: FullAdder port map (XY0(2), XY1(1), C1(0), C1(1), S1(1));
  FA2: FullAdder port map (XY0(3), XY1(2), C1(1), C1(2), S1(2));
  FA3: FullAdder port map (S1(2), XY2(1), C2(0), C2(1), S2(1));
  FA4: FullAdder port map (S1(3), XY2(2), C2(1), C2(2), S2(2));
  FA5: FullAdder port map (C1(3), XY2(3), C2(2), C2(3), S2(3));
  FA6: FullAdder port map (S2(2), XY3(1), C3(0), C3(1), S3(1));
  FA7: FullAdder port map (S2(3), XY3(2), C3(1), C3(2), S3(2));
  FA8: FullAdder port map (C2(3), XY3(3), C3(2), C3(3), S3(3));
  HA1: HalfAdder port map (XY0(1), XY1(0), C1(0), S1(0));
  HA2: HalfAdder port map (XY1(3), C1(2), C1(3), S1(3));
  HA3: HalfAdder port map (S1(1), XY2(0), C2(0), S2(0));
  HA4: HalfAdder port map (S2(1), XY3(0), C3(0), S3(0));

  P(0) <= XY0(0); P(1) <= S1(0); P(2) <= S2(0);
  P(3) <= S3(0); P(4) <= S3(1); P(5) <= S3(2);
  P(6) <= S3(3); P(7) <= C3(3);
end Behavioral;

-- Full Adder and half adder entity and architecture descriptions
-- should be in the project
```

VHDL code for 4-bit array multiplier

```
entity FullAdder is
  port(X, Y, Cin: in bit;
       Cout, Sum: out bit);
end FullAdder;

architecture equations of FullAdder is
begin
  Sum <= X xor Y xor Cin;
  Cout <= (X and Y) or (X and Cin) or (Y and Cin);
end equations;

entity HalfAdder is
  port(X, Y: in bit;
       Cout, Sum: out bit);
end HalfAdder;

architecture equations of HalfAdder is
begin
  Sum <= X xor Y;
  Cout <= X and Y;
end equations;
```

Overview

- Adders/Subtractors
- Multipliers
- Xilinx Unisim

Xilinx simulation libraries

- Xilinx provides the following simulation libraries for simulating primitives and cores (http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ise_c_simulation_libraries.htm):
  - **UNISIM library** for functional simulation of Xilinx primitives
  - UniMacro library for functional simulation of Xilinx macros
  - XilinxCoreLib library for functional simulation of Xilinx cores
  - Xilinx EDK library for behavioral simulation of Xilinx Embedded Development Kit (EDK) IP components
  - SIMPRIM library for timing simulation of Xilinx primitives
  - SmartModel/SecureIP simulation library for both functional and timing simulation of Xilinx Hard-IP, such as PPC, PCIe, GT, and TEMAC IP.

Xilinx Unisim Library of Primitives

- 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
- More modules are available through CoreGenerator (see also Lab #5 of this course)
- Spartan-6 Libraries Guide for HDL Designs: http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/spartan6_hdl.pdf

Example: VHDL Instantiation Template

```
library UNIMACRO;
use unimacro.Vcomponents.all;


-- ADDMACC_MACRO: Add and Multiple Accumulate Function implemented in a DSP48E
--                Virtex-6, Spartan-6
-- Xilinx HDL Libraries Guide, version 11.2

ADDMACC_MACRO_inst : ADDMACC_MACRO
generic map (
   DEVICE => "VIRTEX6",    -- Target Device: "VIRTEX6", "SPARTAN6"
   LATENCY => 3,           -- Desired clock cycle latency, 1-4
   WIDTH_PREADD => 18,     -- Pre-Adder input bus width, 1-25
   WIDTH_MULTIPLIER => 18, -- Multiplier input bus width, 1-18
   WIDTH_PRODUCT => 48)    -- Product output bus width, 1-48
port map (
   PRODUCT => PRODUCT,        -- ADDMACC ouput bus, width determined by WIDTH_PRODUCT generic
   MULTIPLIER => MULTIPLIER, -- MULTIPLIER input bus, width determined by WIDTH_MULTIPLIER generic
   PREADDER1 => PREADDER1,   -- 1st Pre-Adder input bus, width determined by WIDTH_PREADDER generic
   PREADDER2 => PREADDER2,   -- 2nd Pre-Adder input bus, width determined by WIDTH_PREADDER generic
   CARRYIN => CARRYIN, -- 1-bit carry-in input to accumulator
   CE => CE,        -- 1-bit active high input clock enable
   CLK => CLK,      -- 1-bit positive edge clock input
   LOAD => LOAD, -- 1-bit active high input load accumulator enable
   LOAD_DATA => LOAD_DATA, -- Load accumulator input data,
                           -- width determined by WIDTH_PRODUCT generic
   RST => RST    -- 1-bit input active high reset
);
-- End of ADDMACC_MACRO_inst instantiation
```

Always check documentation for your FPGA family to see what's available!

Example: Spartan-3E FPGA

- Use primitive named "RAMB16_S2" (an 8k x 2 Single-Port RAM for Spartan-3E FPGA) to create an 8k x 4 single port RAM

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

library unisim;
use unisim.vcomponents.all;

entity ram_test is
end ram_test;
```

```vhdl
architecture Behavioral of ram_test is

--signal declarations.
signal clk, en, ssr, we : std_logic:='0';
signal Dout, Din : std_logic_vector(3 downto 0):="0000";
signal addr : std_logic_vector(12 downto 0):=(others => '0');

begin

--RAMB16_S2 is 8k x 2 Single-for Spartan-3E.
--We use this to create 8k x 4 Single-Port RAMPort RAM.
--Initialize RAM which carries LSB 2 bits of the data.
RAM1  : RAMB16_S2 port map (
      DO => Dout(1 downto 0),       -- 2-bit Data Output
      ADDR => ADDR,                 -- 13-bit Address Input
      CLK => CLK,                   -- Clock
      DI => Din(1 downto 0),        -- 2-bit Data Input
      EN => EN,                     -- RAM Enable Input
      SSR => SSR,                   -- Synchronous Set/Reset Input
      WE => WE                      -- Write Enable Input
   )

--Initialize RAM which carries MSB 2 bits of the data.
RAM2  : RAMB16_S2 port map (
      DO => Dout(3 downto 2),       -- 2-bit Data Output
      ADDR => ADDR,                 -- 13-bit Address Input
      CLK => CLK,
      DI => Din(3 downto 2),
      EN => EN,
      SSR => SSR,
      WE => WE
   );
```

```vhdl
--100 MHz clock generation for testing process.
clk_process : process
begin
    wait for 5 ns;
    clk <= not clk;
end process;

--Write and Read.
--RAM has a depth of 13 bits and has a width of 4 bits.
simulate : process
begin
        en <='1';
        we <= '1';
        --Write the value "i" at the address "i" for 10 clock cycles.
        for i in 0 to 10 loop
                addr <= conv_std_logic_vector(i,13);
                din <= conv_std_logic_vector(i,4);
                wait for 10 ns;
        end loop;
        we<= '0';
        --Read the RAM for addresses from 0 to 20.
        for i in 0 to 20 loop
                addr <= conv_std_logic_vector(i,13);
                wait for 10 ns;
        end loop;

end process;

end Behavioral;
```

Summary

- Adders/subtractors are very important arithmetic units utilized in a variety of applications (processors, DSPs, etc.)
- More ways to design them; tradeoffs between area and performance
- Always check documentation for your FPGA family

Appendix A: 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