

# ECE385 DIGITAL SYSTEMS LABORATORY

## Experiment 8 An 8-Bit Multiplier

© Janak H. Patel  
Department of Electrical and Computer Engineering  
University of Illinois at Urbana-Champaign

## Today's Topics

- **Experiment 8 (due next week)**
  - 2's Complement Multiplier
  - "shift-and-add" algorithm
  - Design of an Arithmetic Unit in VHDL
  - State Controller in Symbolic VHDL
  - Debouncing a switch input

2

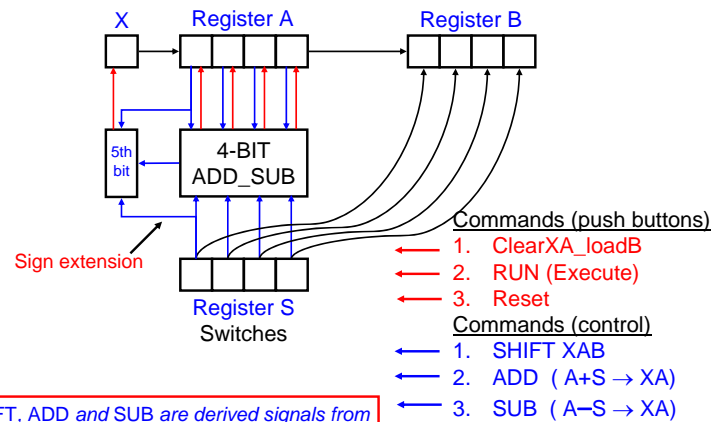
## Pencil-and-Paper Multiplication

```

00000111          7 (multiplicand)
x 11000101      x (-)59 (multiplier)
00000111
+00000000x
+00000111xx
+00000000xxx
+00000000xxxx
+00000000xxxxx
+00000111xxxxxx
-00000111xxxxxxxx Subtract
1111111001100011
(2's comp of result=0000000110011101=413)
    
```

3

## 4-Bit Multiplier



SHIFT, ADD and SUB are derived signals from State Controller. They last only one clock cycle

4

## Experiment 8 Tasks

- **Prelab**
  - Design a 2's Complement Multiplier unit in VHDL using logic operations. Do not use VHDL Arithmetic Operations.
    - ◆ i.e. Do not use  $A \leftarrow A + B$ ; or  $A \leftarrow A - B$ ;
  - Create the following entities and their architecture
    - ◆ 8-bit Registers A and B with parallel load and shift right capability (can use one from Expt. 7)
    - ◆ Extend bit X, a D-flip flop or 1-bit Register
    - ◆ ADD\_SUB9, a 9-bit unit with function control bit **fn**, **fn=0**: Add and **fn=1**: Subtract
    - ◆ A controller implemented using Behavioral Symbolic State Machine description
  - Simulation printout with some test inputs and a floppy disk with your source VHDL files

5

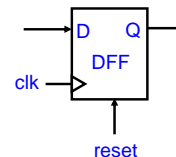
## A D-flip flop

```

entity DFF is
port(D, clk, reset: in std_logic;
      Q: out std_logic);
end DFF;
    
```

```

architecture behavior of DFF is
begin
process(reset, clk);
begin
if reset = '1' then
Q <= '0';
elsif (rising_edge(clk)) then
Q <= D;
end if;
end process;
end behavior;
    
```

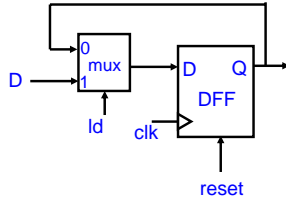


A D-flip flop always loads its input on a clock edge. To hold data in a D-flip flop, we must provide an additional control signal, such as Load/Hold

6

## A D-flip flop with load/hold

```
entity Dreg is
port(D, clk, reset, ld: in std_logic;
      Q : out std_logic);
end Dreg;
```

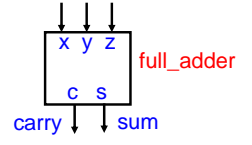


```
architecture behavior of Dreg is
begin
process(reset, clk);
begin
if reset = '1' then
Q <= '0';
elsif (rising_edge(clk)) then
if ld = '1' then
Q <= D; -- else Q is unchanged
end if;
end if;
end process;
end behavior;
```

7

## Adder-Subtractor

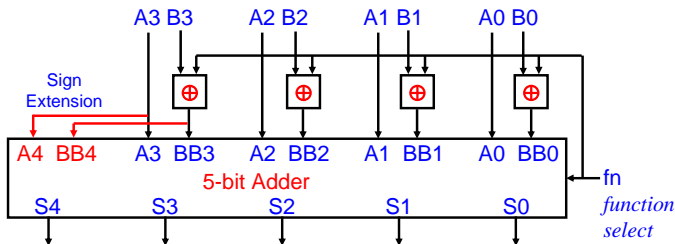
```
entity full_adder is
port(x, y, z : in std_logic;
      s, c : out std_logic);
end entity;
-- we will use the component full_adder to
-- build a 4-bit Adder-Subtractor
```



```
architecture structural of full_adder is
begin
s <= x xor y xor z;
c <= (x and y) or (x and z) or (y and z);
end architecture structural;
-- or end structural; (in older VHDL)
```

8

## 4-bit Adder-Subtractor



```
entity ADD_SUB5 is
port(A,B : in std_logic_vector (3 downto 0);
      S : out std_logic_vector (4 downto 0);
      fn : in std_logic);
end entity;
```

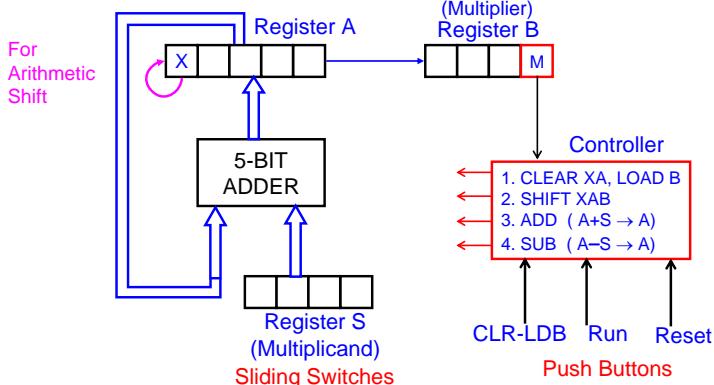
9

## Structural of ADD\_SUB5

```
architecture structural of ADD_SUB5 is
component full_adder is -- omit "is" for older simulators
port(x,y,z: in std_logic; s,c: out std_logic); -- reproduce the entity description
end component full_adder; -- omit name "full_adder" for older simulators
signal c0,c1,c2, c3: std_logic ; -- internal carries in the 4-bit adder
signal BB: std_logic_vector (3 downto 0); -- internal B or NOT(B)
signal A4, BB4: std_logic; -- internal sign extension bits
begin
BB <= B xor (fn&fn&fn&fn); -- when fn=1, complement B
A4 <= A(3); BB4 <= BB(3); -- Sign extension bits copied from sign-bits
FA0: full_adder port map(x =>A(0), y =>BB(0), z =>fn, s =>S(0), c =>c0);
FA1: full_adder port map(x =>A(1), y =>BB(1), z =>c0, s =>S(1), c =>c1);
FA2: full_adder port map(x =>A(2), y =>BB(2), z =>c1, s =>S(2), c =>c2);
FA3: full_adder port map(x =>A(3), y =>BB(3), z =>c2, s =>S(3), c =>c3);
FA4: full_adder port map(x =>A4, y =>BB4, z =>c3, s =>S(4));
-- this should work as well (x =>A(3), y =>BB(3), z =>c3, s =>S(4))
end structural;
```

10

## 4-bit Multiplier Block Diagram



11

## Add-Shift Algorithm illustrated

Multiply 0000 0111 x 1100 0101					
Function	X	A <sub>7:0</sub>	B <sub>7:0</sub>	M	Next Step in words
ClearA, LoadB	0	0000 0000	11000101	B <sub>0</sub> 1	Since M = 1, multiplicand (available from switches S) will be added to A.
ADD	0	0000 0111	11000101	1	Shift XAB by one bit after ADD complete
SHIFT	0	0000 0011	1 1100010	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	0	0000 0001	11 110001	1	Add S to A since M = 1.
ADD	0	0000 1000	11 110001	1	Shift XAB by one bit after ADD complete
SHIFT	0	0000 0100	011 11000	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	0	0000 0010	0011 1100	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	0	0000 0001	00011 110	0	Do not add S to A since M = 0. Shift XAB.
SHIFT	0	0000 0000	100011 11	1	Add S to A since M = 1
ADD	0	0000 0111	100011 11	1	Shift XAB by one bit after ADD complete
SHIFT	0	0000 0011	1100011 1	1	Subtract S from A since 8 <sup>th</sup> bit M = 1.
SUB	1	1111 1100	1100011 1	1	Shift XAB after SUB complete
SHIFT	1	1111 1110	01100011 1	1	8 <sup>th</sup> shift done. Stop. 16-bit Product in AB.

12

## Binary Multiplication

- Consider how you would load the multiplier and the multiplicand and command your controller to perform one multiplication
  - Set Reset to 1 and then 0
  - Set switches  $S \leftarrow$  Multiplier
  - Clear XA and Load B from S
  - Set switches  $S \leftarrow$  Multiplicand
  - RUN the Multiplication  $AB \leftarrow S \times B$ 
    - ◆ Multiplication of two 8-bit numbers S and B results in a 16-bit product in register AB

13

## Multiplication Algorithm

Assuming two 4-bit 2's Complement numbers the following RTL Program implements the multiplication.

state:	Action	
S0:	$XA \leftarrow A + M \cdot S$	<start when RUN switch is ON> <if M=1 then Add S to the partial product>
S1:	SHIFT XAB	<arithmetic right shift >‡
S2:	$XA \leftarrow A + M \cdot S$	[fn<='0'; if M=1 then loadX <='1'; loadA <='1'];
S3:	SHIFT XAB	
S4:	$XA \leftarrow A + M \cdot S$	
S5:	SHIFT XAB	
S6:	$XA \leftarrow A - M \cdot S$	<if M=1 then Subtract S from partial product>
S7:	SHIFT XAB	<final product in Register AB>
S8:	Wait Until RUN switch returns to 0	

‡ Any shift is permissible in our multiplier, if X is always overwritten. However, when adding a Zero, if you skip an ADD, you must use Arithmetic Shift

14

## A word on Input/Output

- Use Push Buttons for all command inputs
  - You may have to design a debouncing circuit for the RUN button otherwise it may execute the multiply more than once!
    - ◆ A simple Debouncing circuit uses a counter of say 20-bits to count out a million consecutive 1's or 0's
    - ◆ When you see RUN=1 the first time, start the multiplication and reset the counter to 0. At the end of multiply operation, use the counter to find consecutives  $2^{20}$  0's on RUN button. Counter must be reset every time you see RUN=1.
  - Reset, Load, Clear etc need not be debounced
- Use sliding switches for data input
- Use Hex display for result

15

## Remarks

- Follow Experiment-8 description in Lab Notes
  - Bring your code on a floppy disk or FTP it
  - Bring a detailed block diagram of your design
    - ◆ Components, ports and interconnections labeled
- A Note on Cooperation with others
  - You are encouraged to cooperate with others in the class. However, each team must write its own VHDL source and Lab Report.
  - You are permitted to exchange ideas, algorithms, very high level flow charts and so on – **BUT NO Entities, Components, Architectures, or Files of any kind.**

16