CPEG 422/622
EMBEDDED SYSTEMS DESIGN

Chengmo Yang

chengmo@udel.edu

Evans 201C

LECTURE 8
MULTIPLICATION

REVIEW

- Last Lecture:
  - Register and FSM

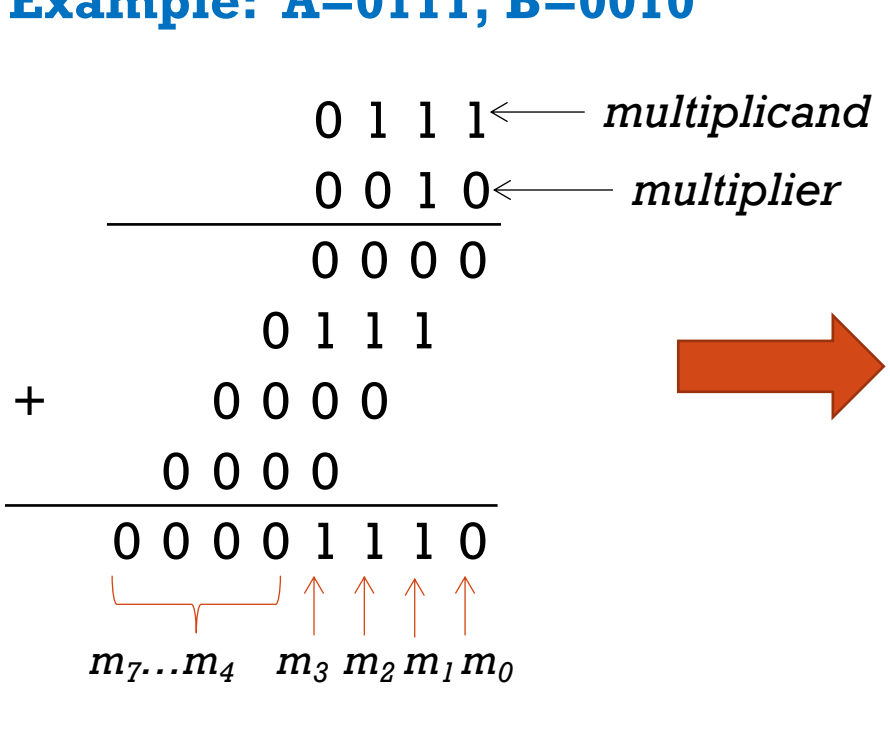
- This lecture:
  - Basic multiplication
  - Booth's multiplication
  - Multiplier design in VHDL

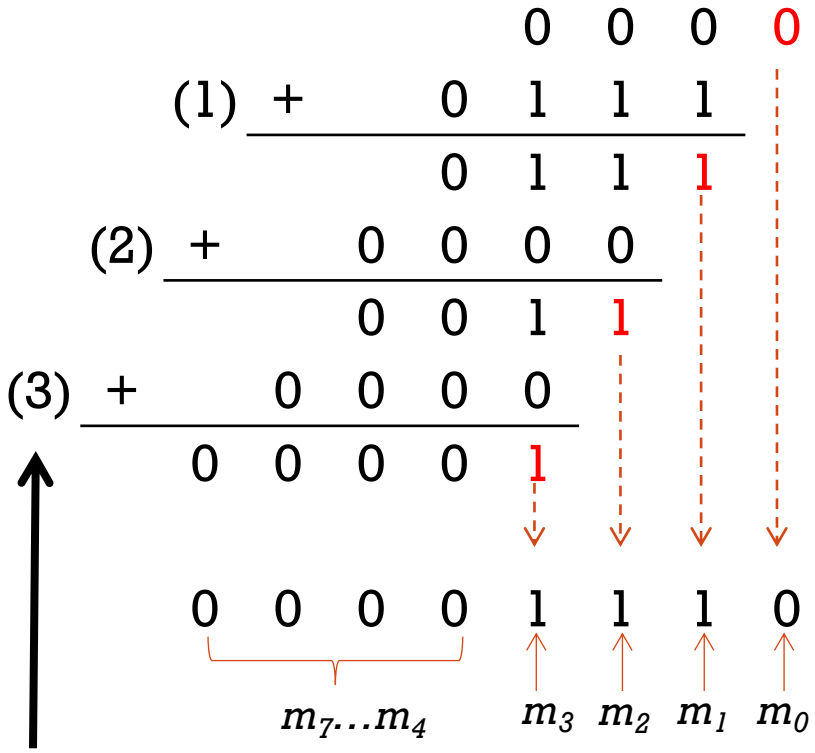
BINARY MULTIPLICATION

Combinational vs. Sequential logic

A binary multiplier with combinational logic → use repeated sums

Example: A=0111, B=0010





Need three 4-bit adders + bunch of gates

BINARY MULTIPLICATION

Q: How to use sequential logic for the 4-bit multiplier?

Some important lessons from the comb. logic version:

0 1 1 1 ← *multiplicand*

0 0 1 0 ← *multiplier*

① The 4-bit multiplier requires 3 sums

② The "new sum" to add is either **0000** or **0111** (*the multiplicand*)

③ The LSB(red bit) of a sum is NOT used in the next sum (but is part of final answer)

```
                    0 0 0 0
              +     0 1 1 1
                    0 1 1 1
          +   0 0 0 0
                    0 0 1 1
      +   0 0 0 0
                    0 0 0 1
  ---------------------------
  0 0 0 0 1 1 1 0
  m7...m4 m3 m2 m1 m0
```

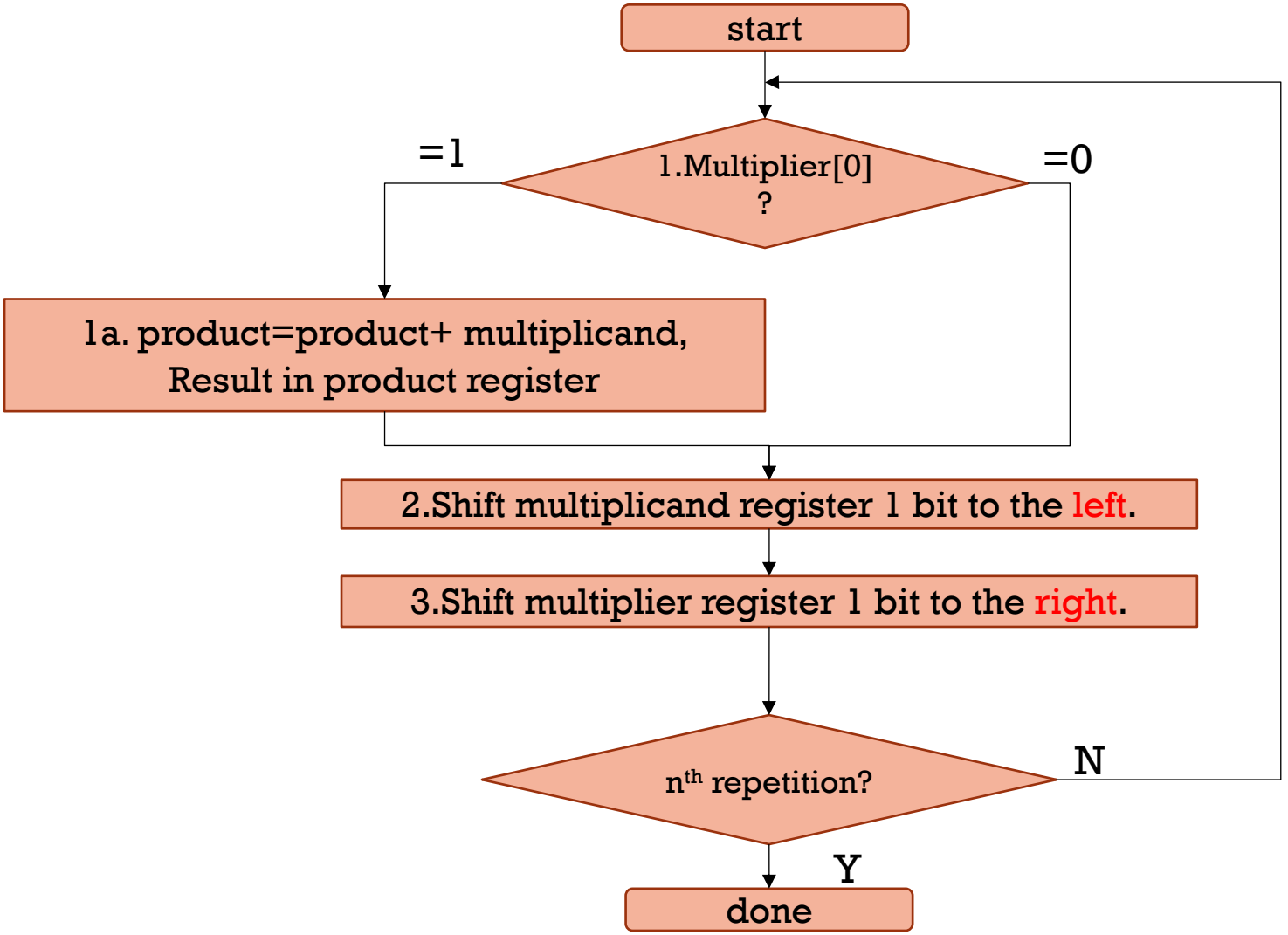
So, the sequential version requires:

① One 4-bit adder used 3 times

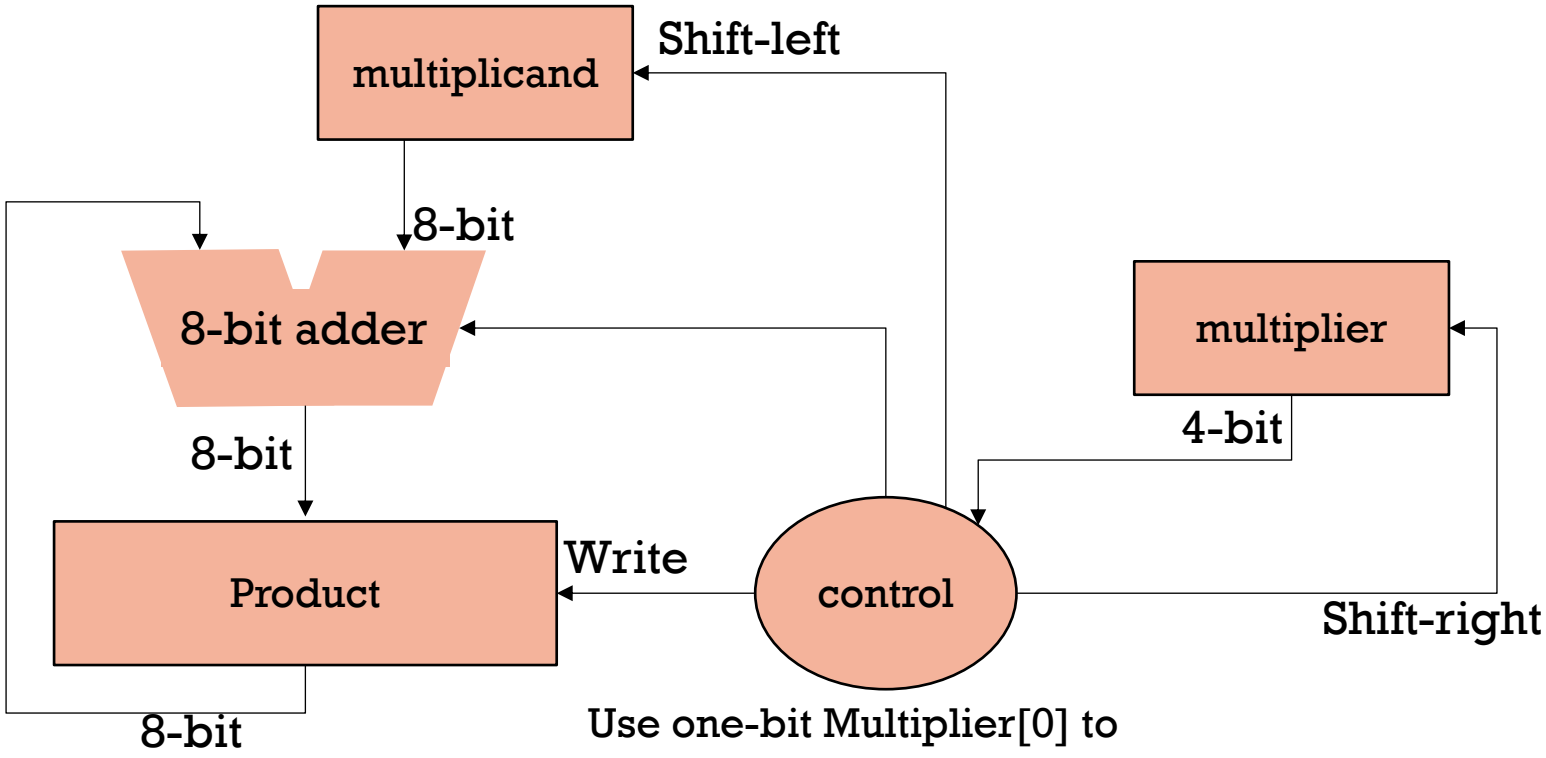
② Store **0111** for use in later partial sums

③ Can fill a shift register with the LSB's

BASIC MULTIPLICATION ALGORITHM



A BASIC MULTIPLIER DESIGN



Hardware cost:

- An 8-bit adder
- An 8-bit shift register for multiplicand
- A 4-bit shift register for multiplier
- An 8-bit register for product

IMPROVE THE DESIGN?

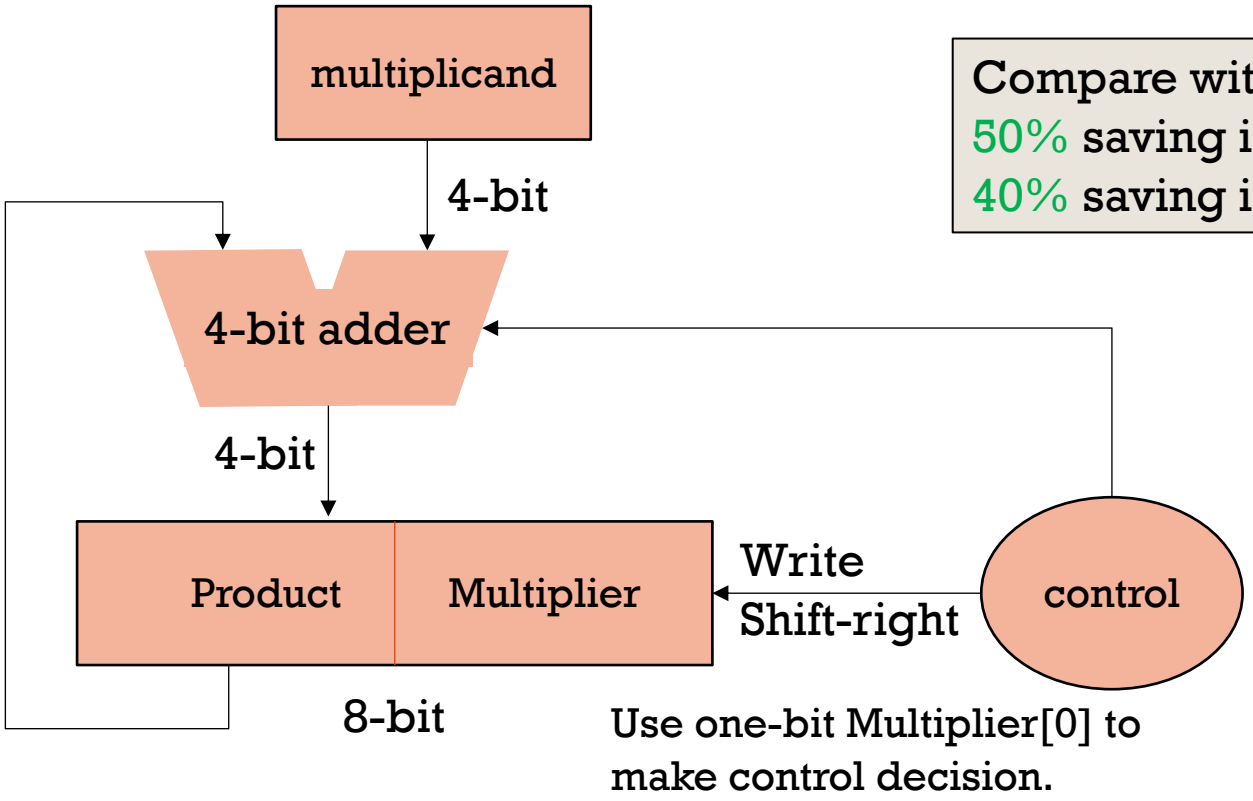
- Every step, the multiplicand is shift left then added to the product.

This is equivalent to

- Shift the product right then add the multiplicand to its higher bits.
- At every step,
  - the LSB of the multiplier is consumed and will not be used again
  - the LSB of the product is produced and will not be needed again

Can use the upper part of the multiplier register to store the lower part of the product

AN IMPROVED MULTIPLIER DESIGN



Compare with before we have
50% saving in adder hardware
40% saving in register hardware

Use one-bit Multiplier[0] to make control decision.

Hardware cost:

- A 4-bit adder
- A 4-bit register for multiplicand
- A 4-bit shift register for multiplier
- A 4-bit shift register for product

AN IMPROVED MULTIPLICATION ALGORITHM

- In the binary numbering system, the sum of a consecutive sequence of 1's can be easily calculated as:

$$2^{i+k} + \cdots + 2^{i+1} + 2^i = 2^{i+k+1} - 2^i$$

- **Example:**

$$0111\ 1110 = 1000\ 000 - 0000\ 0010$$

If 0111 1110 is the multiplier, the basic algorithm requires 6 add operations. But they can be replaced with multiplicand * (1000 000 – 0000 0010) -- one addition and one subtraction!

- Booth algorithm replaces each sequence of consecutive 1's with one addition and one subtraction.
- **Booth encoding of the multiplier:**

$$0111\ 1110 = 1000\ 00\bar{1}0$$

add subtract

BOOTH MULTIPLICATION: ALGORITHM

- Multiply two 2's compliment binary numbers.
- Basic idea:
  1. Look at TWO bits to decide operations.
  2. ADD/SUB +SHIFT to implement multiplication.

Multiplier[0] Initially 0, then get Multiplier[0] in previous iteration.

Operation Code Table

OP_1	OP_0	Operation	Add/sub
0	0	**Shift right** (Product and multiplier)	No-op
0	1	**Product = Product + multiplicand** **Shift right**(Product and multiplier)	Add
1	0	**Product = Product - multiplicand** **Shift right**(Product and multiplier)	Sub
1	1	**Shift right** (Product and multiplier)	No-op

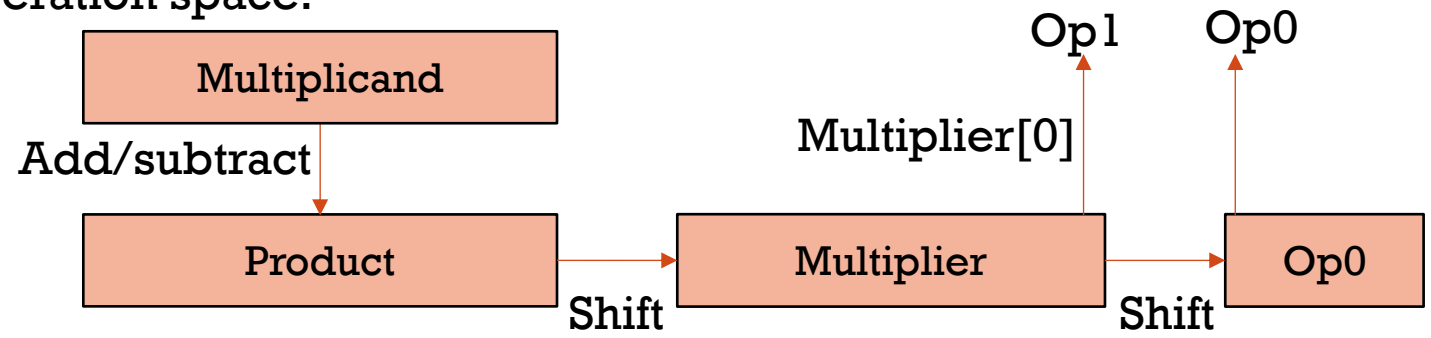
BOOTH MULTIPLICATION: ALGORITHM

- Multiply two 2's compliment binary numbers.

- Basic idea:
  1. Look at TWO bits to decide operations.
  2. ADD/SUB +SHIFT to implement multiplication.

BOOTH MULTIPLICATION

Basic operation space:

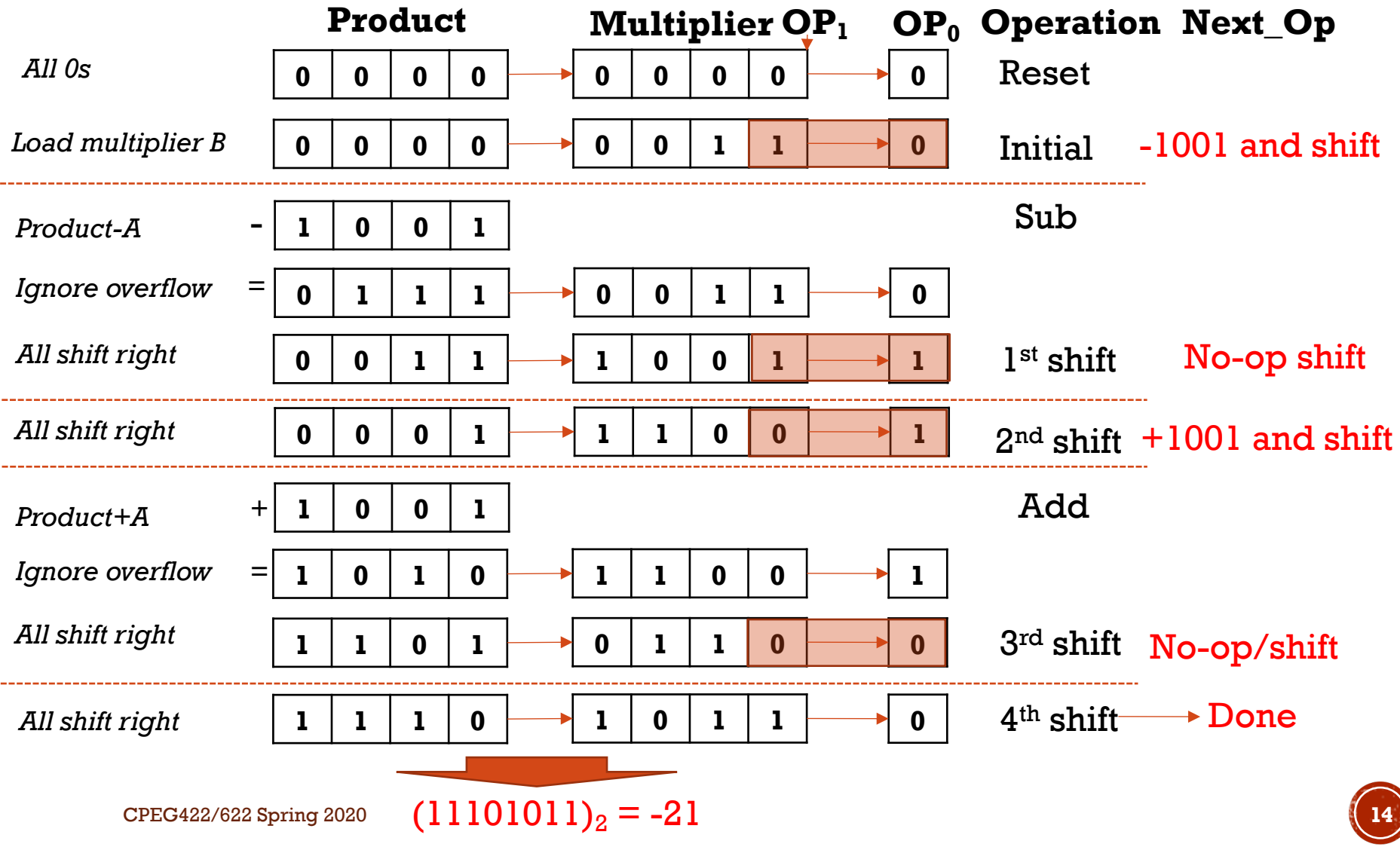


Notes:

1. No need to worry about overflow since addition and subtraction are always performed alternately.
2. Shift of product is **arithmetic shift (fill sign bit at MSB)**.
3. Shift in of multiplier comes from shift out of product, and shift out of multiplier goes to OP_0.

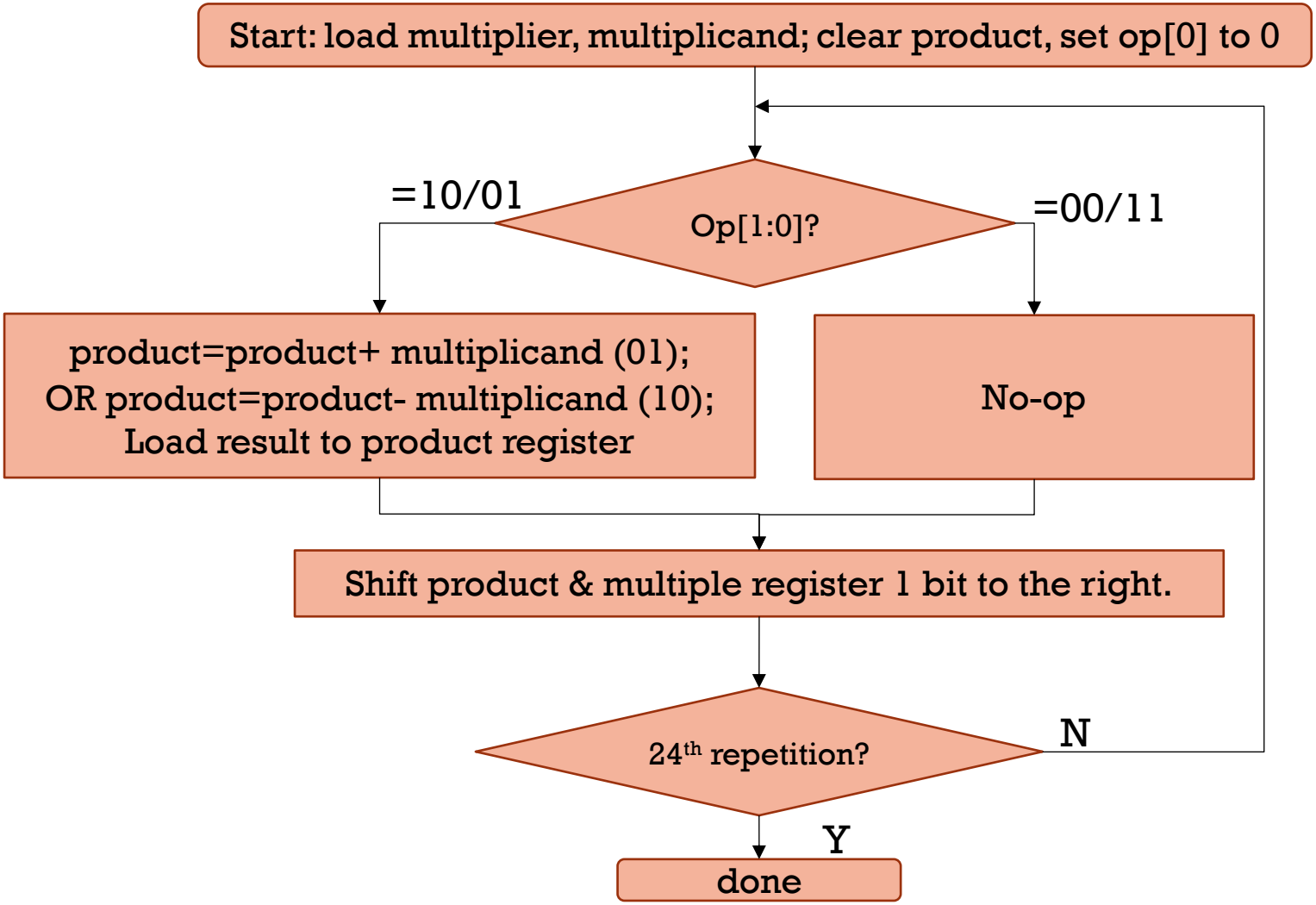
BOOTH MULTIPLICATION

Example: A=1001 (-7), B=0011 (3)

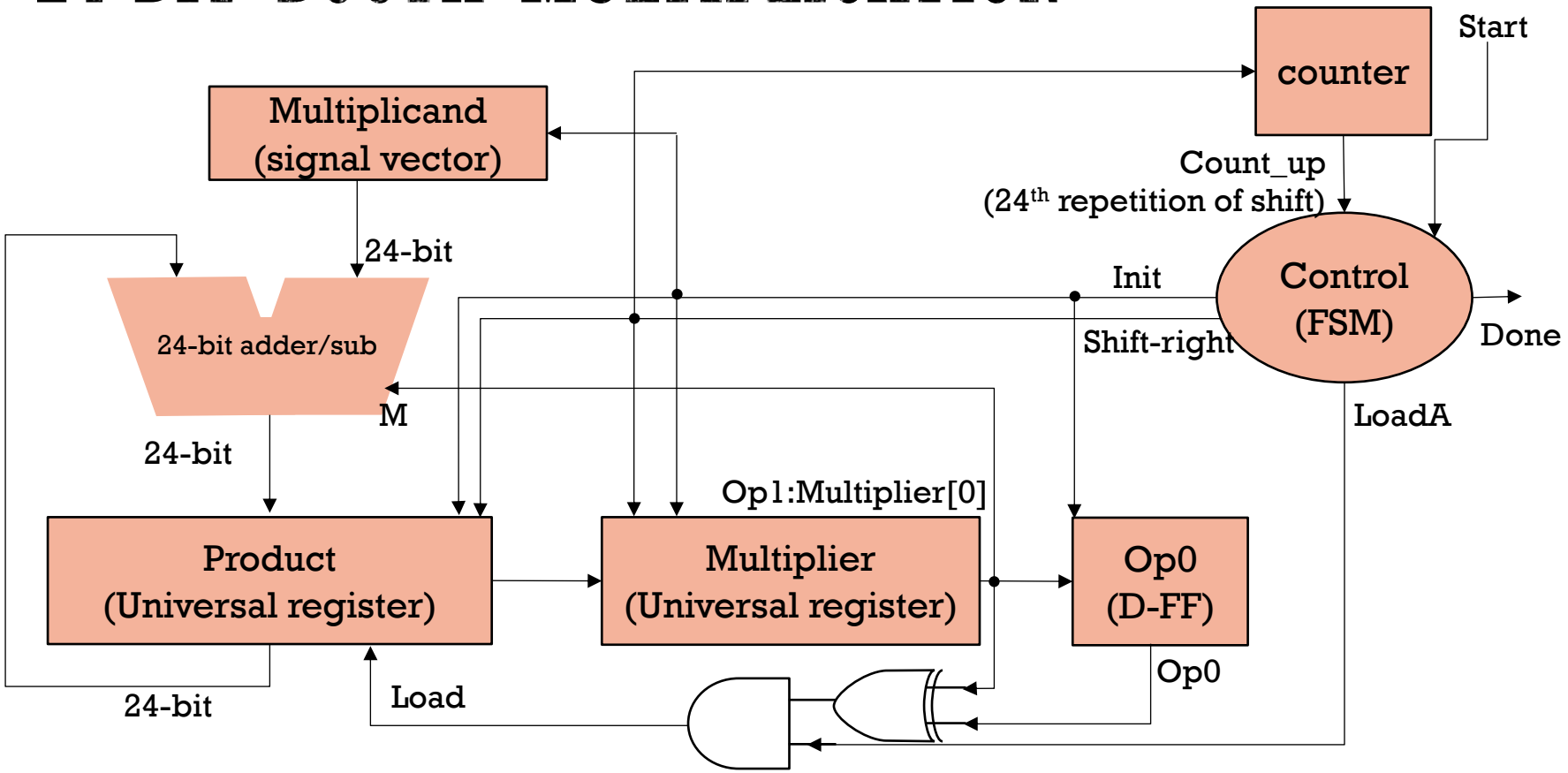


$(11101011)_2 = -21$

24-BIT BOOTH MULTIPLICATION



24-BIT BOOTH MULTIPLICATION

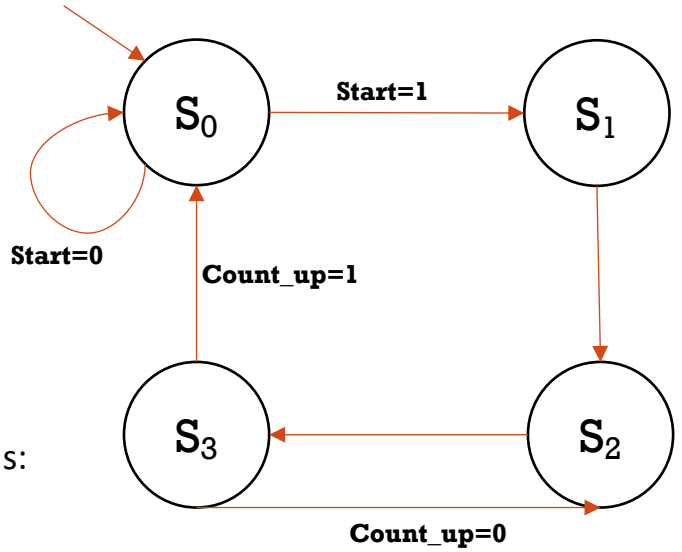


Except for Add/subtractor, all components have **clock** and **reset** input.

Final multiplication result comes from **product** register and **multiplier** register.

BOOTH MULTIPLIER CONTROL FSM

This controller takes 2 states to process each bit.



S_0 : Idle, circuit is ready.
S_1 : Init, initialize components.
S_2 : LoadA, selectively load result from add/sub to product register.
S_3 : Shift, shift product and multiplier registers 1-bit to the right.

a) State transitions caused by input (synchronized by clock rising edge)as:
   In S_0, if start=1, $S_0 \rightarrow S_1$; otherwise remains S_0.
   In S_1, S_1 always goes to S_2 regardless of the input value.
   In S_2, S_2 always goes to S_3 regardless of the input value.
   In S_3, if count_up=0, $S_3 \rightarrow S_2$; otherwise $S_3 \rightarrow S_0$.

b) Outputs in each state:
   S_0 : done =1; Init=0; LoadA=0; Shift=0;
   S_1 : done =0; Init=1; LoadA=0; Shift=0;
   S_2 : done =0; Init=0; LoadA=1; Shift=0;
   S_3 : done =0; Init=0; LoadA=0; Shift=1;

Question for you:
The controller always has LoadA = 1 at state S_2. How to selectively load the product?

VHDL DESIGN

Used components:

- 24-bit adder/subtractor
- 24-bit universal shift register for product, multiplier, multiplicand
- 5-bit counter (count 24 times of shifting)
- Multiplication control FSM

```
entity Booth_top is
Port (
    Clock:      in STD_LOGIC;
    Reset:      in STD_LOGIC;
    Start:      in STD_LOGIC;
    A:          in STD_LOGIC_VECTOR (23 downto 0);
    B:          in STD_LOGIC_VECTOR (23 downto 0);
    Product: out STD_LOGIC_VECTOR (47 downto 0);
    Done:       out STD_LOGIC
   );
end Booth_top;
```

```
entity booth_control is
Port {
        clock:      in STD_LOGIC;
        reset:      in STD_LOGIC;
        start:      in STD_LOGIC;
        count_up:   in STD_LOGIC;
        init:       out STD_LOGIC;
        loadA:      out STD_LOGIC;
        shift:      out STD_LOGIC;
        done:       out STD_LOGIC
};
end booth_control;
```

NEXT LECTURE

- More on VHDL
- More on sequential circuit testing