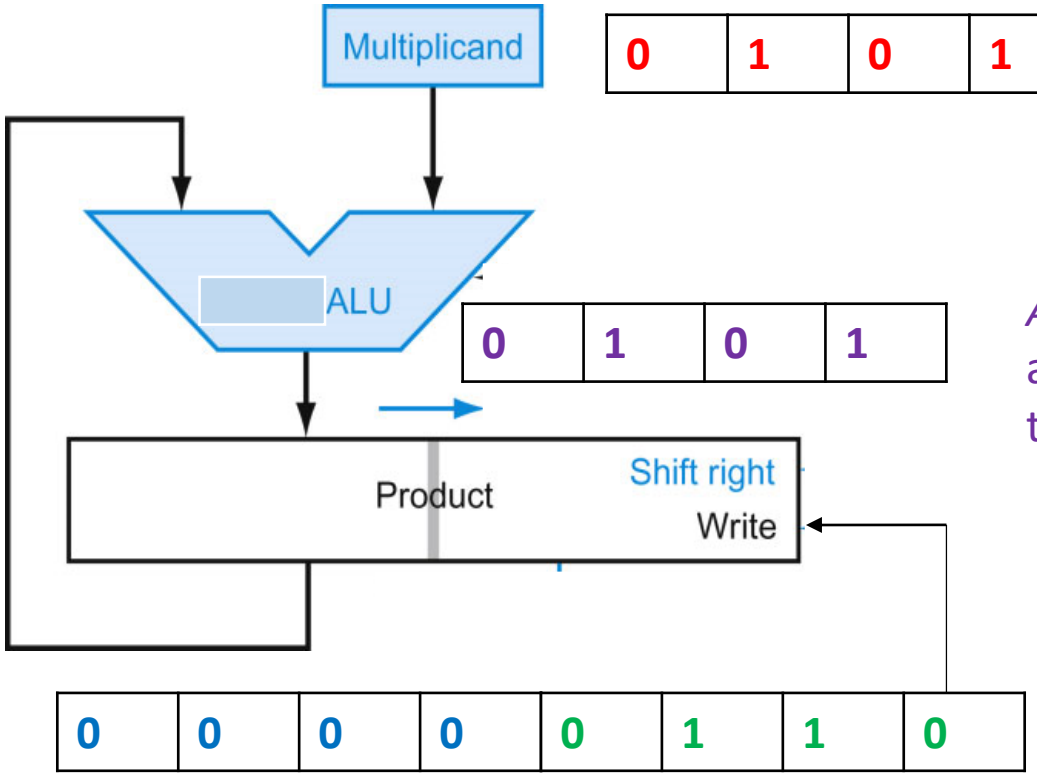
Four-bit multiplier producing 8-bit result – efficient (2nd implementation)

4-bit Multiplicand register, 4-bit ALU (adder), 8-bit Product Register, 5 times 6 = 30

Initialization:



Multiplicand initialized and then never changed

ALU output is the sum of *Multiplicand* and most-significant half of *Product* at all times

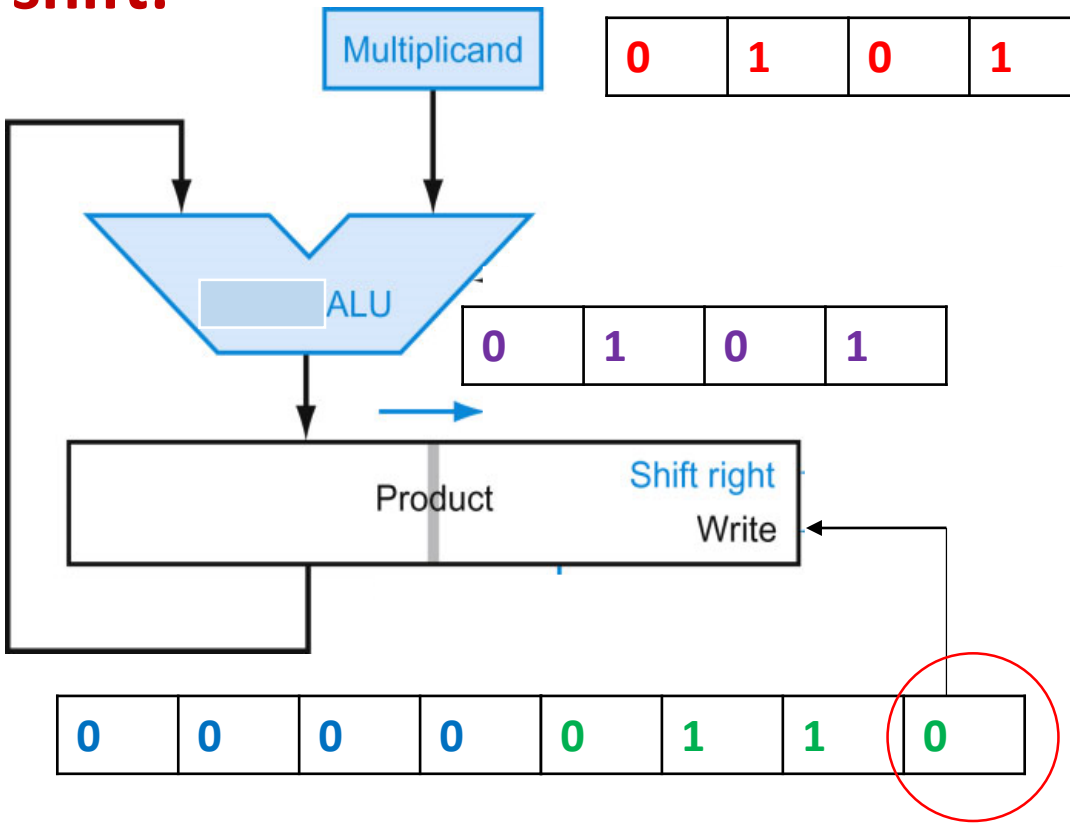
Most-significant half of *Product* initialized to all zeros

Least-significant half of *Product* initialized to multiplier value

Four-bit multiplier producing 8-bit result – efficient (2nd implementation)

4-bit Multiplicand register, 4-bit ALU (adder), 8-bit Product Register, 5 times 6 = 30

1st cycle before shift:



0 1 0 1
x 0 1 1 0
0 0 0 0

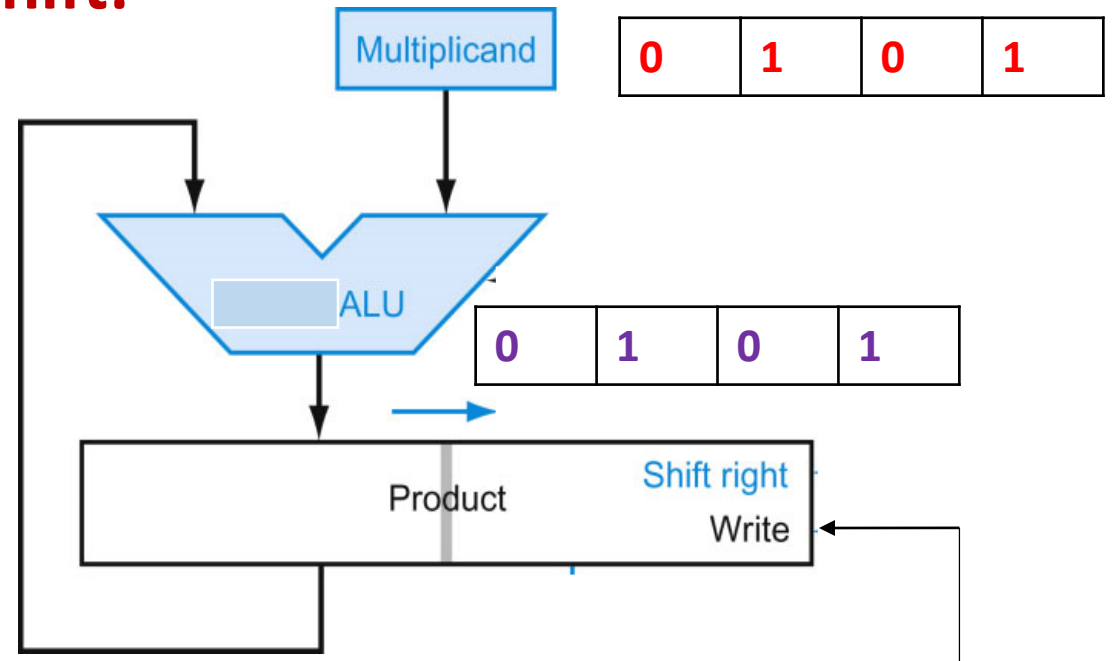
+
0 0 0 0

Most-significant half of *Product* unchanged because LSb of *Product* is 0

Four-bit multiplier producing 8-bit result – efficient (2nd implementation)

4-bit Multiplicand register, 4-bit ALU (adder), 8-bit Product Register, 5 times 6 = 30

1st cycle after shift:



0 1 0 1
x 0 1 1
0 0 0 0

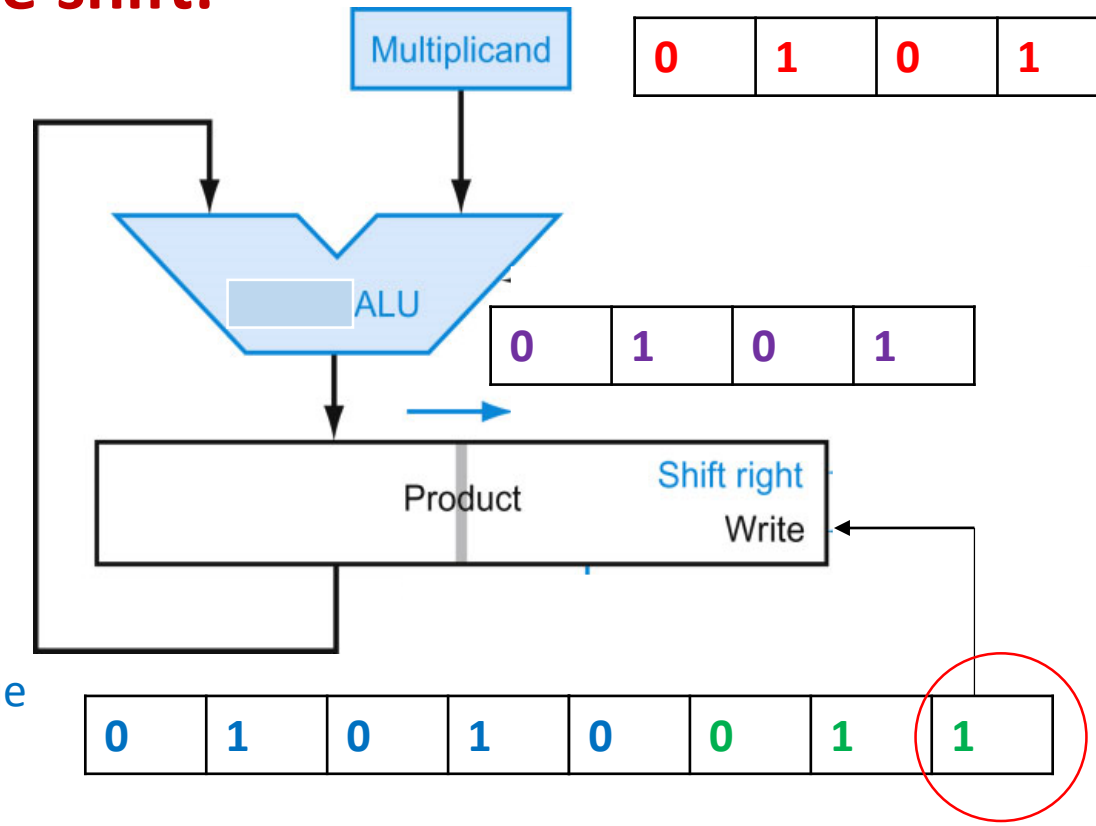
+
0 0 0 0 0

0 shifted into *Product* from left (always)

Four-bit multiplier producing 8-bit result – efficient (2nd implementation)

4-bit Multiplicand register, 4-bit ALU (adder), 8-bit Product Register, 5 times 6 = 30

2nd cycle before shift:



Most-significant half of *Product* gets ALU value because LSb of *Product* is 1

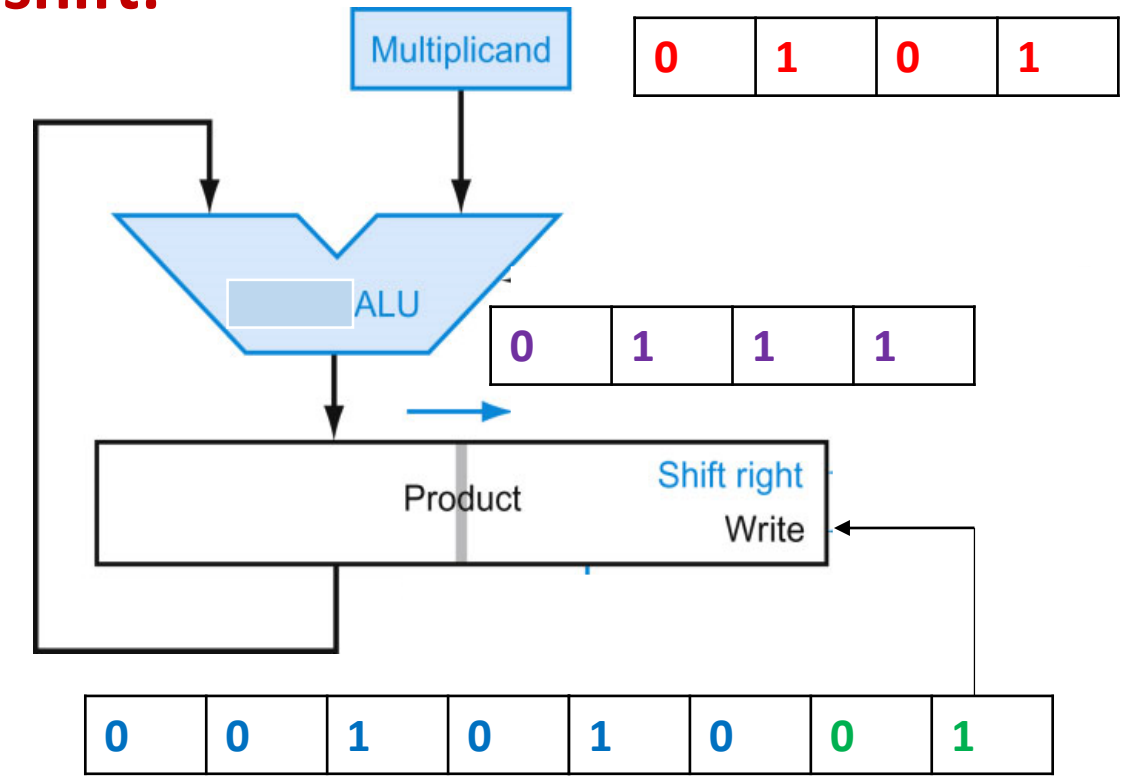
```
  0 1 0 1
x 0 1 1
  0 0 0 0
0 1 0 1

+
 0 1 0 1 0
```

Four-bit multiplier producing 8-bit result – efficient (2nd implementation)

4-bit Multiplicand register, 4-bit ALU (adder), 8-bit Product Register, 5 times 6 = 30

2nd cycle after shift:

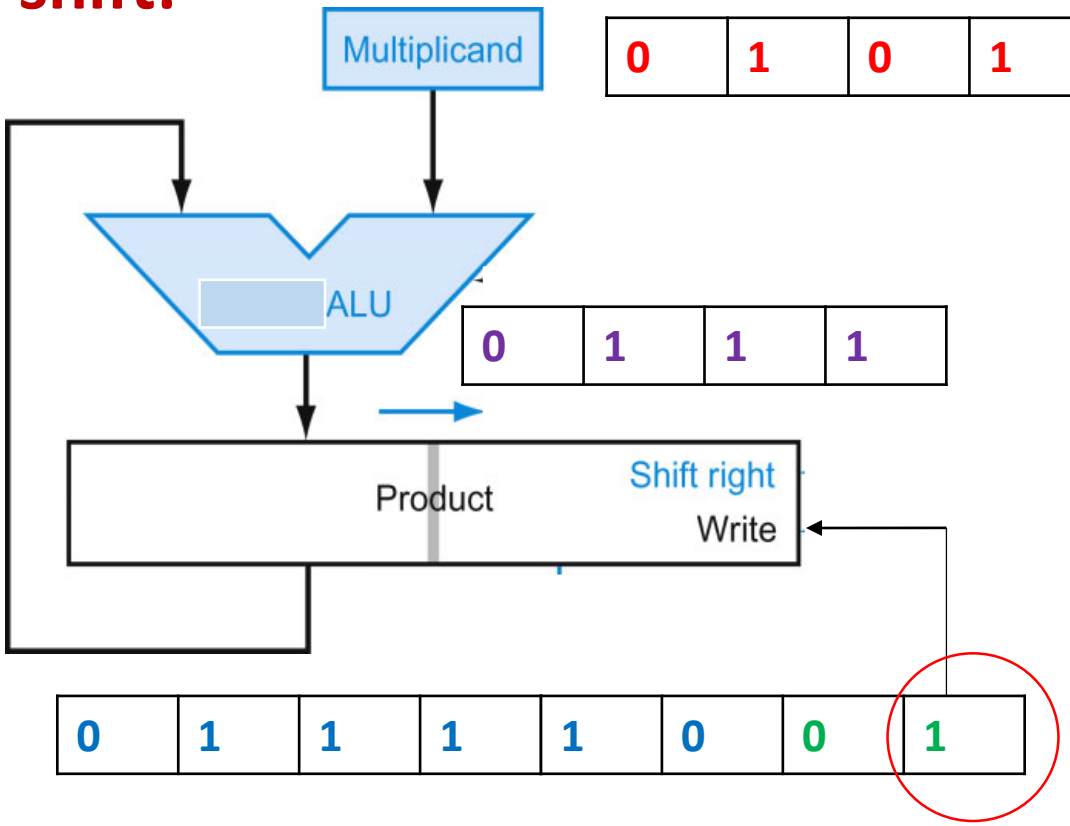


0 1 0 1
x 0 1
0 0 0 0
0 1 0 1
+
0 0 1 0 1 0

Four-bit multiplier producing 8-bit result – efficient (2nd implementation)

4-bit Multiplicand register, 4-bit ALU (adder), 8-bit Product Register, 5 times 6 = 30

3rd cycle before shift:



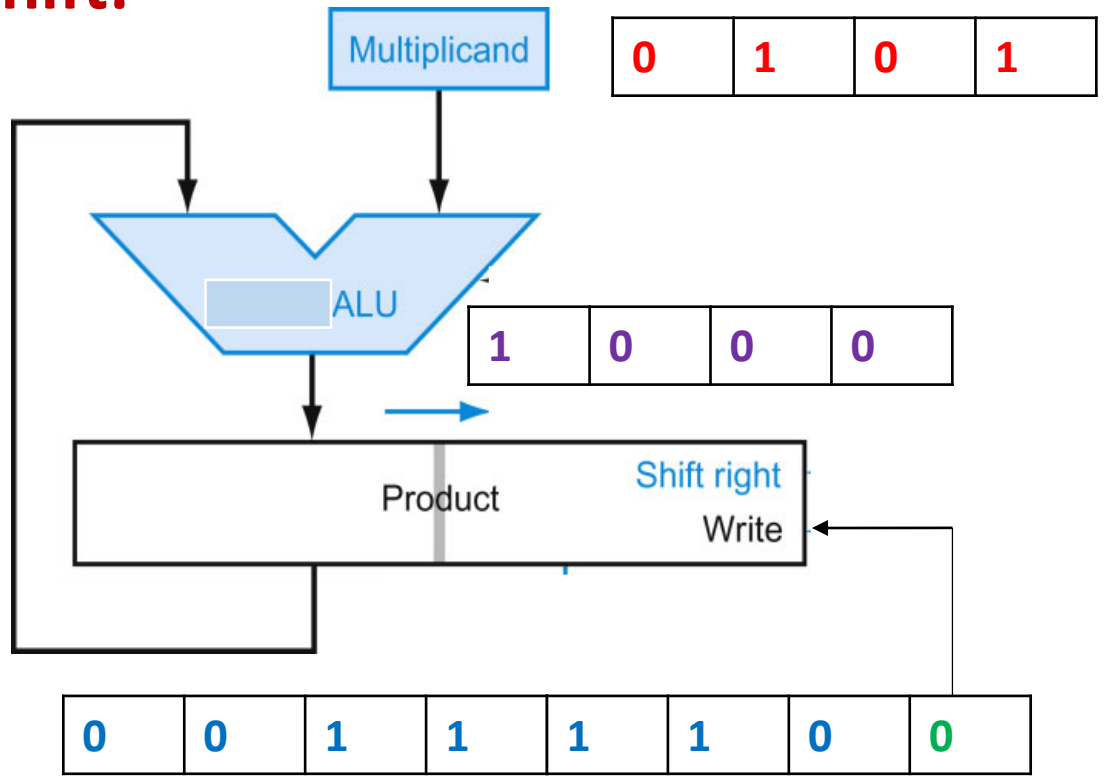
Most-significant half of *Product* gets ALU value because LSb of *Product* is 1

```
    0 1 0 1
x   0 1
  ---------
    0 0 0 0
  0 1 0 1
0 1 0 1
+ ---------
0 1 1 1 1 0
```

Four-bit multiplier producing 8-bit result – efficient (2nd implementation)

4-bit Multiplicand register, 4-bit ALU (adder), 8-bit Product Register, 5 times 6 = 30

3rd cycle after shift:

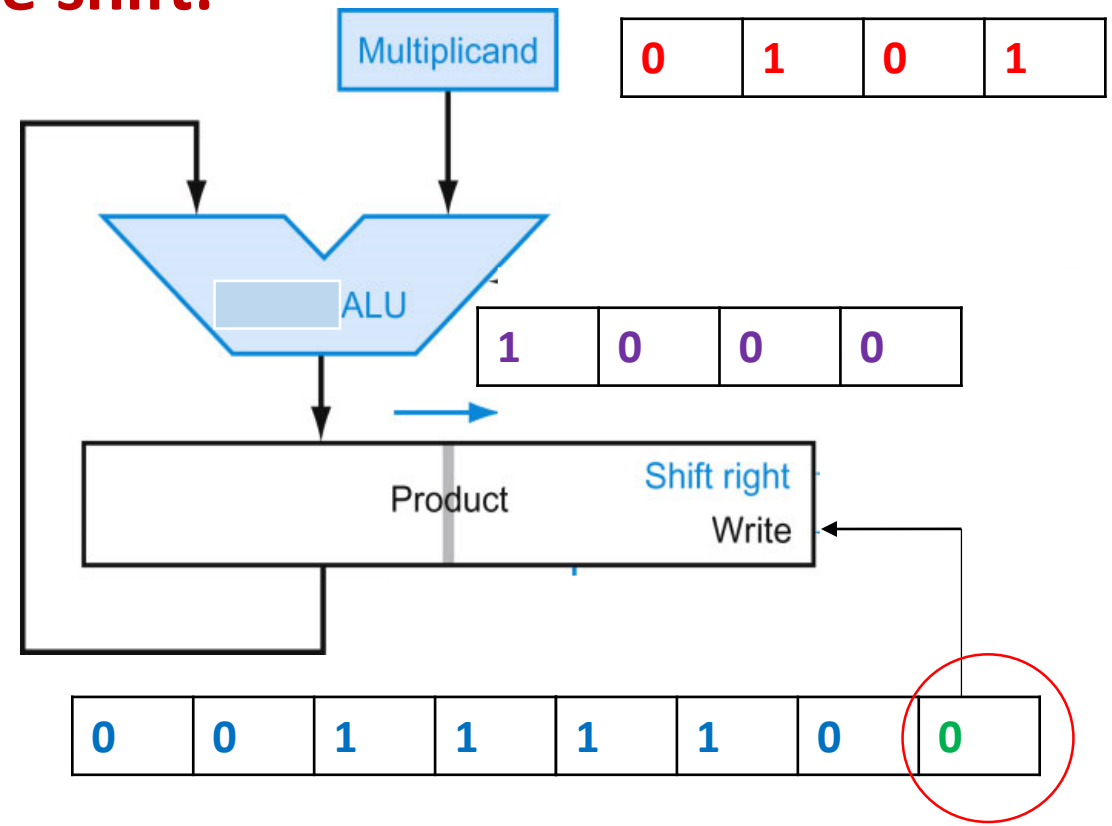



```
      0 1 0 1
x     0
_____________
      0 0 0 0
    0 1 0 1
  0 1 0 1
+
_____________
0 0 1 1 1 1 0
```

Four-bit multiplier producing 8-bit result – efficient (2nd implementation)

4-bit Multiplicand register, 4-bit ALU (adder), 8-bit Product Register, 5 times 6 = 30

4th cycle before shift:



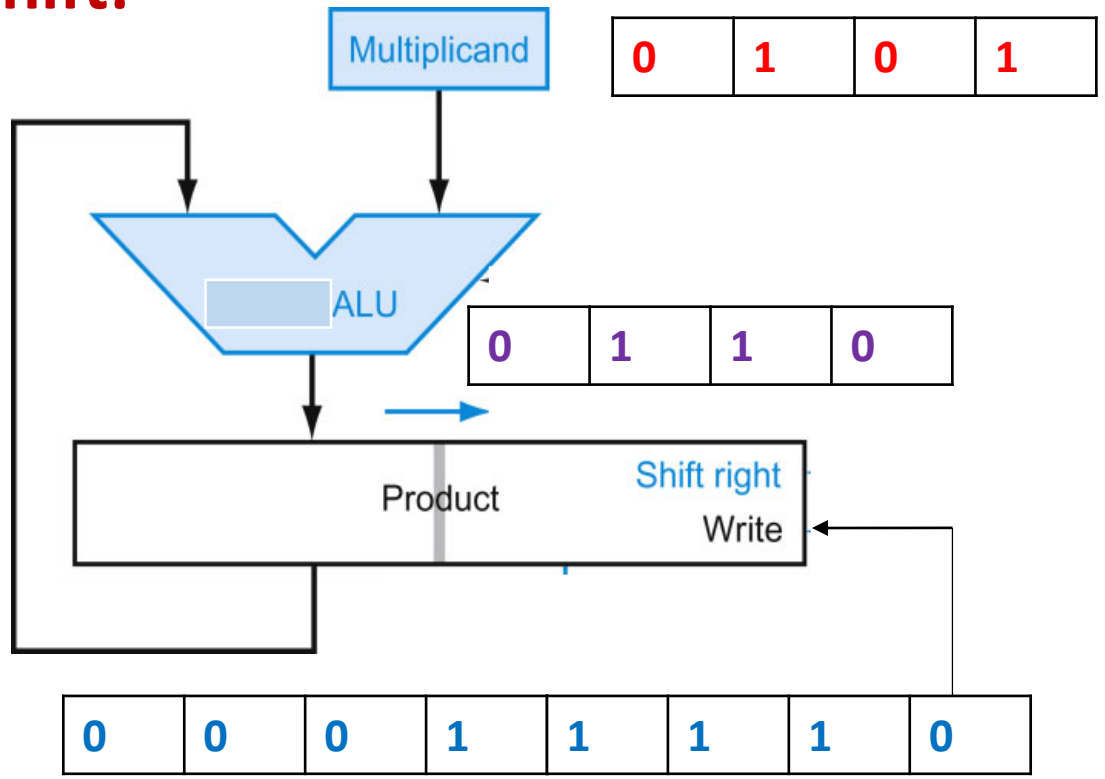

```
      0 1 0 1
x     0
-------------
      0 0 0 0
    0 1 0 1
  0 1 0 1
+ 0 0 0 0
-------------
  0 0 1 1 1 1 0
```

Most-significant half of *Product* unchanged because LSb of *Product* is 0

Four-bit multiplier producing 8-bit result – efficient (2nd implementation)

4-bit Multiplicand register, 4-bit ALU (adder), 8-bit Product Register, 5 times 6 = 30

4th cycle after shift:




```
      0 1 0 1
x
        0 0 0 0
      0 1 0 1
    0 1 0 1
+   0 0 0 0
  0 0 0 1 1 1 1 0
```

Result in *Product* is 0001 1110 binary or 30 decimal – note that MSb of *Product* will always be 0