# Computer Organization and Architecture

## COSC 2425

Lecture – 10

Sept 21st , 2022

Acknowledgement: Slides from Edgar Gabriel & Kevin Long

# Chapter 3

## Arithmetic for Computers

# Questions

1. What operations can hardware perform? How to instruct computer to perform a certain operation? How are negative numbers/exponentials represented?
2. How do we perform addition, multiplication, division?
3. How do we improve the speed of the computer? Can we do things in parallel (compute while loading next data, etc.)
4. Where is data stored? How can we make it efficient?
5. Can we perform computations in parallel to improve performance?
6. How do we define performance?

# Questions

1. What operations can hardware perform? How to instruct computer to perform a certain operation? How are negative numbers/exponentials represented?
2. How do we perform addition, multiplication, division?
3. How do we improve the speed of the computer? Can we do things in parallel (compute while loading next data, etc.)
4. Where is data stored? How can we make it efficient?
5. Can we perform computations in parallel to improve performance?
6. How do we define performance?

# Arithmetic for Computers

- Operations on integers
  - Addition and subtraction
  - Dealing with overflow
  - Multiplication and division

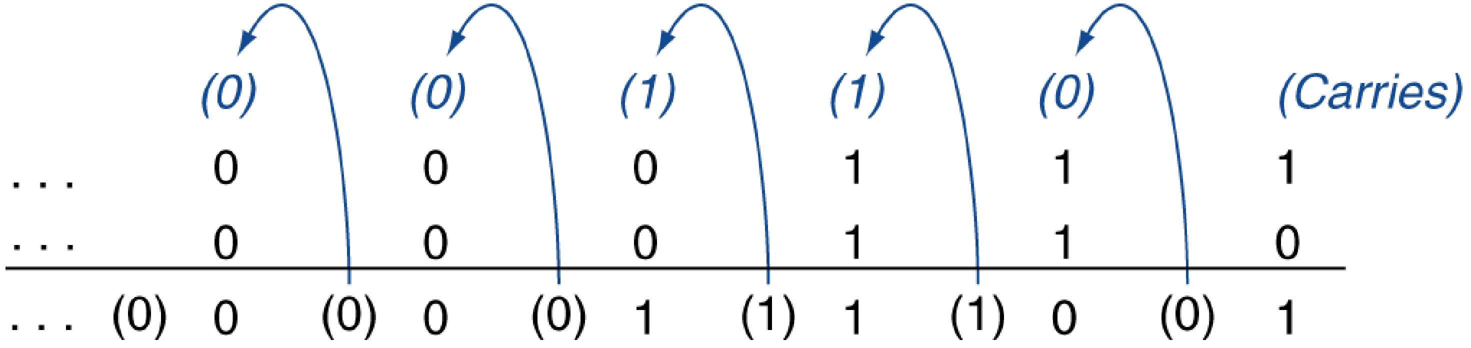
# Addition and Subtraction

- Unsigned numbers:
- Addition in binary is same as addition in decimal system.

```
   1   <----------  Carry
   27
+  59
 ----
   86
```

# Addition and Subtraction

- Addition in binary is same as addition in decimal system.
- Addition of unsigned numbers
- Adding $6_{ten}$ to $7_{ten}$



# Adding 64-bit numbers

$$
\begin{array}{rll}
 & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000111_{two} & = 7_{ten} \\
+ & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000110_{two} & = 6_{ten} \\
\hline
= & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00001101_{two} & = 13_{ten}
\end{array}
$$

# Subtraction

- Subtracting $6_{ten}$ from $7_{ten}$ directly

$$
\begin{array}{rl}
 & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000111_{two} = 7_{ten} \\
 & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000110_{two} = 6_{ten} \\
\hline
= & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{two} = 1_{ten}
\end{array}
$$

# Subtraction

- Subtracting $6_{ten}$ from $7_{ten}$ directly

$$
\begin{array}{rl}
& 0000000\ 0000000\ 0000000\ 0000000\ 0000000\ 0000000\ 0000000\ 0000111_{two} = 7_{ten} \\
& 0000000\ 0000000\ 0000000\ 0000000\ 0000000\ 0000000\ 0000000\ 0000110_{two} = 6_{ten} \\
\hline
= & 0000000\ 0000000\ 0000000\ 0000000\ 0000000\ 0000000\ 0000000\ 0000001_{two} = 1_{ten}
\end{array}
$$

- Subtracting $6_{ten}$ from $7_{ten}$ using two's complement.

$$7 + (-6)$$

$$0000000\ 0000000\ 0000000\ 0000000\ 0000000\ 0000000\ 0000000\ 0000111_{two} = 7_{ten}$$

# Subtraction

- Subtracting $6_{ten}$ from $7_{ten}$ directly

$$
\begin{array}{rl}
 & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000111_{two} = 7_{ten} \\
 & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000110_{two} = 6_{ten} \\
\hline
= & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{two} = 1_{ten}
\end{array}
$$

- Subtracting $6_{ten}$ from $7_{ten}$ using two's complement.

$$7 + (-6)$$

$$
\begin{array}{rl}
 & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000111_{two} = 7_{ten} \\
+ & 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111010_{two} = -6_{ten} \\
\hline
= & 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{two} = 1_{ten}
\end{array}
$$

# Subtraction

Subtraction uses addition, the appropriate operand is negated.

# Overflow

Unsigned Integers

$$
\begin{array}{r}
11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111_{two} \\
+ \\
00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001_{two} \\
\hline
1\ \ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000_{two}
\end{array}
$$

65 bits needed
Results in an **overflow**.

# Overflow

- Signed integers, addition
- When can an overflow occur?

| Operand 1 | Operand 2 | Overflow | Check |
|---|---|---|---|
| +ve | -ve | | |
| -ve | +ve | | |
| | | | |
| | | | |

# Overflow

- Signed integers, addition
- When can an overflow occur?

| Operand 1 | Operand 2 | Overflow | Check |
|---|---|---|---|
| +ve | -ve | No | |
| -ve | +ve | No | |
| | | | |
| | | | |

$$-10 + 4 = -6$$

Sum is never larger than one of the operands, and must fit in 64 bits.

# Overflow

- Signed integers, addition
- When can an overflow occur?

| Operand 1 | Operand 2 | Overflow | Check |
|---|---|---|---|
| +ve | -ve | No | |
| -ve | +ve | No | |
| +ve | +ve | | |
| | | | |

# 2's Complement

- Lets consider a 4-bit representation of numbers, 16 combinations are possible
  - Let us consider 7 + 4

```
  0111
+ 0100
```

| | |
|---|---|
| 0000 | 0 |
| 0001 | 1 |
| 0010 | 2 |
| 0011 | 3 |
| 0100 | 4 |
| 0101 | 5 |
| 0110 | 6 |
| 0111 | 7 |
| 1000 | -8 |
| 1001 | -7 |
| 1010 | -6 |
| 1011 | -5 |
| 1100 | -4 |
| 1101 | -3 |
| 1110 | -2 |
| 1111 | -1 |

# 2's Complement

- Lets consider a 4-bit representation of numbers, 16 combinations are possible
  - Let us consider 7 + 4

```
  0111
+ 0100
  ----
  1011
```

Adding two +ve numbers results in a negative number, then overflow has occurred

| | |
|---|---|
| 0000 | 0 |
| 0001 | 1 |
| 0010 | 2 |
| 0011 | 3 |
| 0100 | 4 |
| 0101 | 5 |
| 0110 | 6 |
| 0111 | 7 |
| 1000 | -8 |
| 1001 | -7 |
| 1010 | -6 |
| 1011 | -5 |
| 1100 | -4 |
| 1101 | -3 |
| 1110 | -2 |
| 1111 | -1 |

# Overflow

- Signed integers, addition
- When can an overflow occur?

| **Operand 1** | **Operand 2** | **Overflow** | **Check** |
|---|---|---|---|
| +ve | -ve | No | |
| -ve | +ve | No | |
| +ve | +ve | Yes | -ve result |
| | | | |

# Overflow

- Signed integers, addition
- When can an overflow occur?

| Operand 1 | Operand 2 | Overflow | Check |
|---|---|---|---|
| +ve | -ve | No | |
| -ve | +ve | No | |
| +ve | +ve | Yes | -ve result |
| -ve | -ve | | |

# 2’s Complement

- Lets consider a 4-bit representation of numbers, 16 combinations are possible
  – Let us consider (-8) + (- 7)

```
    1000
  + 1001
  ------
   10001
```

Over flow bit → first 1 of 10001; Sign bit → second digit of 10001

Adding two -ve numbers results in a positive number, then overflow has occurred

| | |
|---|---|
| 0000 | 0 |
| 0001 | 1 |
| 0010 | 2 |
| 0011 | 3 |
| 0100 | 4 |
| 0101 | 5 |
| 0110 | 6 |
| 0111 | 7 |
| 1000 | -8 |
| 1001 | -7 |
| 1010 | -6 |
| 1011 | -5 |
| 1100 | -4 |
| 1101 | -3 |
| 1110 | -2 |
| 1111 | -1 |

# Overflow

- Signed integers, addition
- When can an overflow occur?

| Operand 1 | Operand 2 | Overflow | Check |
|---|---|---|---|
| +ve | -ve | No | |
| -ve | +ve | No | |
| +ve | +ve | Yes | -ve result |
| -ve | -ve | Yes | +ve result |

# OVERFLOW RULE:

If two numbers are added, and they are both positive or both negative, then overflow occurs if and only if the result has the opposite sign.

No overflow can occur when adding numbers with different signs

# 2's Complement

- Lets consider a 4-bit representation of numbers, 16 combinations are possible
  - Let us consider 6 + (-2)

```
              0110
            + 1110
            ------
Over flow bit → 10100
Sign bit       →  (second digit)
```

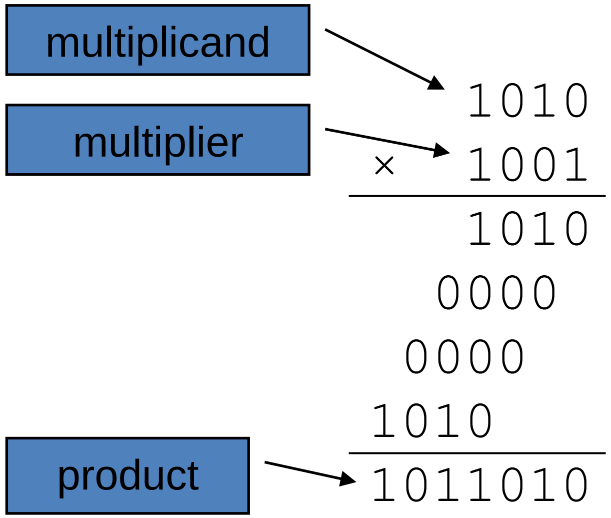
- Overflow can not occur if the two operands have different signs -> ignore overflow bit

| | |
|---|---|
| 0000 | 0 |
| 0001 | 1 |
| 0010 | 2 |
| 0011 | 3 |
| 0100 | 4 |
| 0101 | 5 |
| 0110 | 6 |
| 0111 | 7 |
| 1000 | -8 |
| 1001 | -7 |
| 1010 | -6 |
| 1011 | -5 |
| 1100 | -4 |
| 1101 | -3 |
| 1110 | -2 |
| 1111 | -1 |

# Multiplication

- Start with long-multiplication approach

Multiplication of binary number is similar to decimal system



# Multiplication Hardware

- Start with long-multiplication approach
- Multiplication of binary number is similar to decimal system

```
  1111
× 1111
------
```

# Multiplication Hardware

- Start with long-multiplication approach
- Multiplication of binary number is similar to decimal system

Length of product never greater sum of operand lengths

Operand 1 length $\rightarrow$ 4

Operand 2 length $\rightarrow$ 4

Result length $\leq$ 8

```
      1111
    × 1111
  --------
      1111
     1111
    1111
   1111
  --------
  11100001
```

# Multiplication Hardware

- Start with long-multiplication approach
- Multiplication of binary number is similar to decimal system

1. Length of product never greater sum of operand lengths

```
1010
1001
____
```

# Multiplication Hardware

- Start with long-multiplication approach
- Multiplication of binary number is similar to decimal system

1. Length of product never greater sum of operand lengths

```
1010
1001
```

# Multiplication Hardware

- Start with long-multiplication approach
- Multiplication of binary number is similar to decimal system

1. Length of product never greater sum of operand lengths
2. Multiplication:
   1. If multiplier bit is 1, simply copy the multiplicand

```
 1010
 1001
-----
 1010
```

# Multiplication Hardware

- Start with long-multiplication approach
- Multiplication of binary number is similar to decimal system

1. Length of product never greater sum of operand lengths
2. Multiplication:
   1. If multiplier bit is 1, simply copy the multiplicand
   2. Shift multiplicand by 1 bit to left and fill zeros on right

```
10100
 1001
-----
 1010
```

# Multiplication Hardware

- Start with long-multiplication approach
- Multiplication of binary number is similar to decimal system

1. Length of product never greater sum of operand lengths
2. Multiplication:
    1. If multiplier bit is 1, simply copy the multiplicand
    2. Shift multiplicand by 1 bit to left and fill zeros on right
    3. Shift Multiplier 1 bit to the right

$$\begin{array}{r} 10100 \\ 100 \\ \hline 1010 \end{array}$$

# Multiplication Hardware

- Start with long-multiplication approach
- Multiplication of binary number is similar to decimal system

1. Length of product never greater sum of operand lengths
2. Multiplication:
    1. If multiplier bit is 1, simply copy the multiplicand
    2. Shift multiplicand by 1 bit to left and fill zeros on right
    3. Shift Multiplier 1 bit to the right
    4. Repeat

```
10100
  100
-----
 1010
```

# Multiplication Hardware

- Start with long-multiplication approach
- Multiplication of binary number is similar to decimal system

1. Length of product never greater sum of operand lengths
2. Multiplication:
    1. If multiplier bit is 1, simply copy the multiplicand
        1. **If multiplier bit is 0, fill with zeros**
    2. Shift multiplicand by 1 bit to left and fill zeros on right
    3. Shift Multiplier 1 bit to the right
    4. Repeat

```
 10100
   100
------
  1010
 00000
```

# Multiplication Hardware

- Start with long-multiplication approach
- Multiplication of binary number is similar to decimal system

1. Length of product never greater sum of operand lengths
2. Multiplication:
    1. If multiplier bit is 1, simply copy the multiplicand
        1. If multiplier bit is 0, fill with zeros
    2. **Shift multiplicand by 1 bit to left and fill zeros on right**
    3. Shift Multiplier 1 bit to the right
    4. Repeat

```
101000
   100
------
  1010
 00000
```

# Multiplication Hardware

- Start with long-multiplication approach
- Multiplication of binary number is similar to decimal system

1. Length of product never greater sum of operand lengths
2. Multiplication:
    1. If multiplier bit is 1, simply copy the multiplicand
        1. If multiplier bit is 0, fill with zeros
    2. Shift multiplicand by 1 bit to left and fill zeros on right
    3. **Shift Multiplier 1 bit to the right**
    4. Repeat

```
101000
    10
------
  1010
 00000
```

# Multiplication Hardware

- Start with long-multiplication approach
- Multiplication of binary number is similar to decimal system

1. Length of product never greater sum of operand lengths
2. Multiplication:
    1. If multiplier bit is 1, simply copy the multiplicand
        1. **If multiplier bit is 0, fill with zeros**
    2. Shift multiplicand by 1 bit to left and fill zeros on right
    3. Shift Multiplier 1 bit to the right
    4. Repeat

```
101000
    10
------
  1010
 00000
000000
```

# Multiplication Hardware

- Start with long-multiplication approach
- Multiplication of binary number is similar to decimal system

1. Length of product never greater sum of operand lengths
2. Multiplication:
    1. If multiplier bit is 1, simply copy the multiplicand
        1. If multiplier bit is 0, fill with zeros
    2. **Shift multiplicand by 1 bit to left and fill zeros on right**
    3. Shift Multiplier 1 bit to the right
    4. Repeat

```
1010000
     10
-------
   1010
  00000
 000000
```

# Multiplication Hardware

- Start with long-multiplication approach
- Multiplication of binary number is similar to decimal system

1. Length of product never greater sum of operand lengths
2. Multiplication:
   1. If multiplier bit is 1, simply copy the multiplicand
      1. If multiplier bit is 0, fill with zeros
   2. Shift multiplicand by 1 bit to left and fill zeros on right
   3. **Shift Multiplier 1 bit to the right**
   4. Repeat

```
1010000
      1
-------
   1010
  00000
 000000
```

# Multiplication Hardware

- Start with long-multiplication approach
- Multiplication of binary number is similar to decimal system

1. Length of product never greater sum of operand lengths
2. Multiplication:
   1. **If multiplier bit is 1, simply copy the multiplicand**
      1. If multiplier bit is 0, fill with zeros
   2. Shift multiplicand by 1 bit to left and fill zeros on right
   3. Shift Multiplier 1 bit to the right
   4. Repeat

```
1010000
      1
-------
   1010
  00000
 000000
1010000
```

# Multiplication Hardware

- Start with long-multiplication approach
- Multiplication of binary number is similar to decimal system

1. Length of product never greater sum of operand lengths
2. Multiplication:
    1. **If multiplier bit is 1, simply copy the multiplicand**
        1. If multiplier bit is 0, fill with zeros
    2. Shift multiplicand by 1 bit to left and fill zeros on right
    3. Shift Multiplier 1 bit to the right
    4. Repeat

```
  1010000
        1
---------
     1010
+   00000
+  000000
+ 1010000
---------
  1011010
```

# Multiplication Hardware

- Start with long-multiplication approach
- Multiplication of binary number is similar to decimal system

Length of product never greater sum of operand lengths

If the integers are represented using 4 btis,

To design the multiplication hardware, lets consider the **multiplicand** and the **result** to be **8 bits long**.

# Multiplication Hardware

- Multiplicand: 1010
- Multiplier: 1001

# Multiplication Hardware

- Multiplicand: 1010
- Multiplier: 1001
- Initialize multiplicand register with 8 bits, consisting of leading zeros

Multiplicand register
0000 1010

# Multiplication Hardware

- Multiplicand: 1010
- Multiplier: 1001
- Initialize multiplicand register
- Initialize product register with 8 bits, consisting of all zeros

Multiplicand register
0000 1010

Product register
0000 0000

# Multiplication Hardware

- Multiplicand: 1010
- Multiplier: 1001
- Initialize multiplicand register
- Initialize product register
- Initialize multiplier register, 4 bits

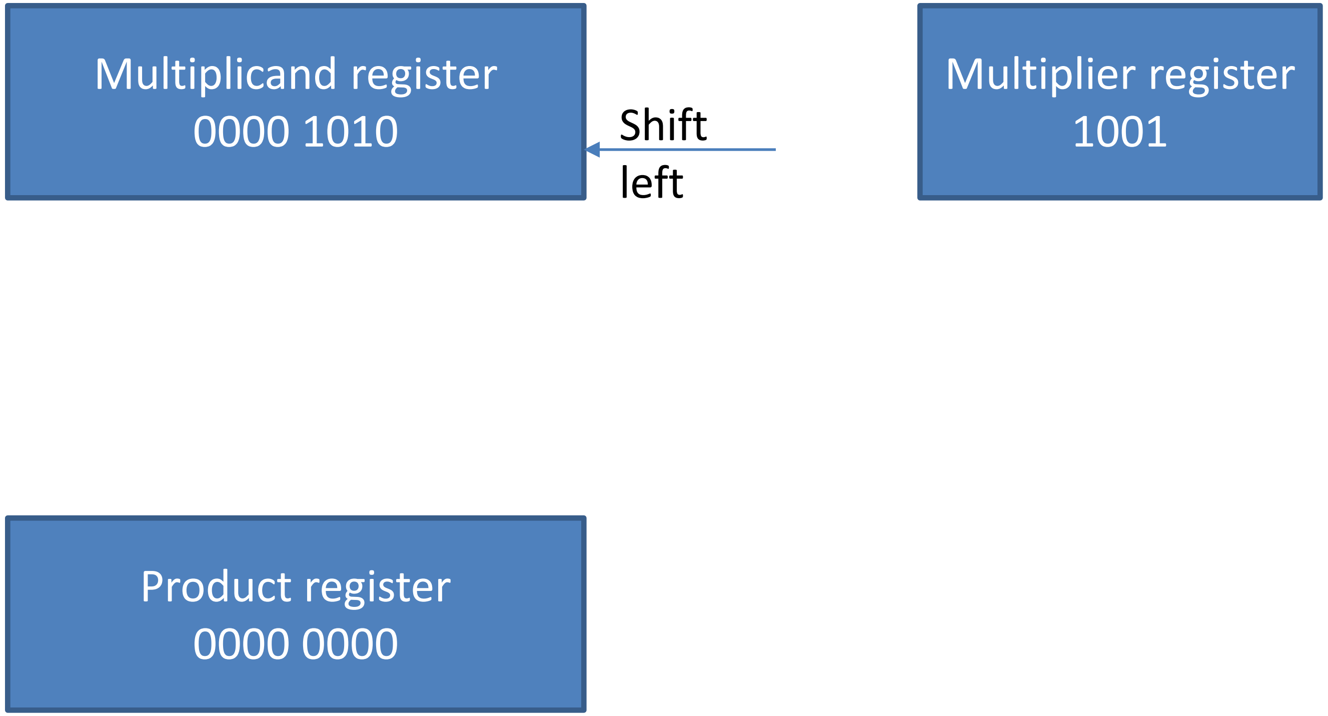
Multiplicand register
0000 1010

Multiplier register
1001

Product register
0000 0000

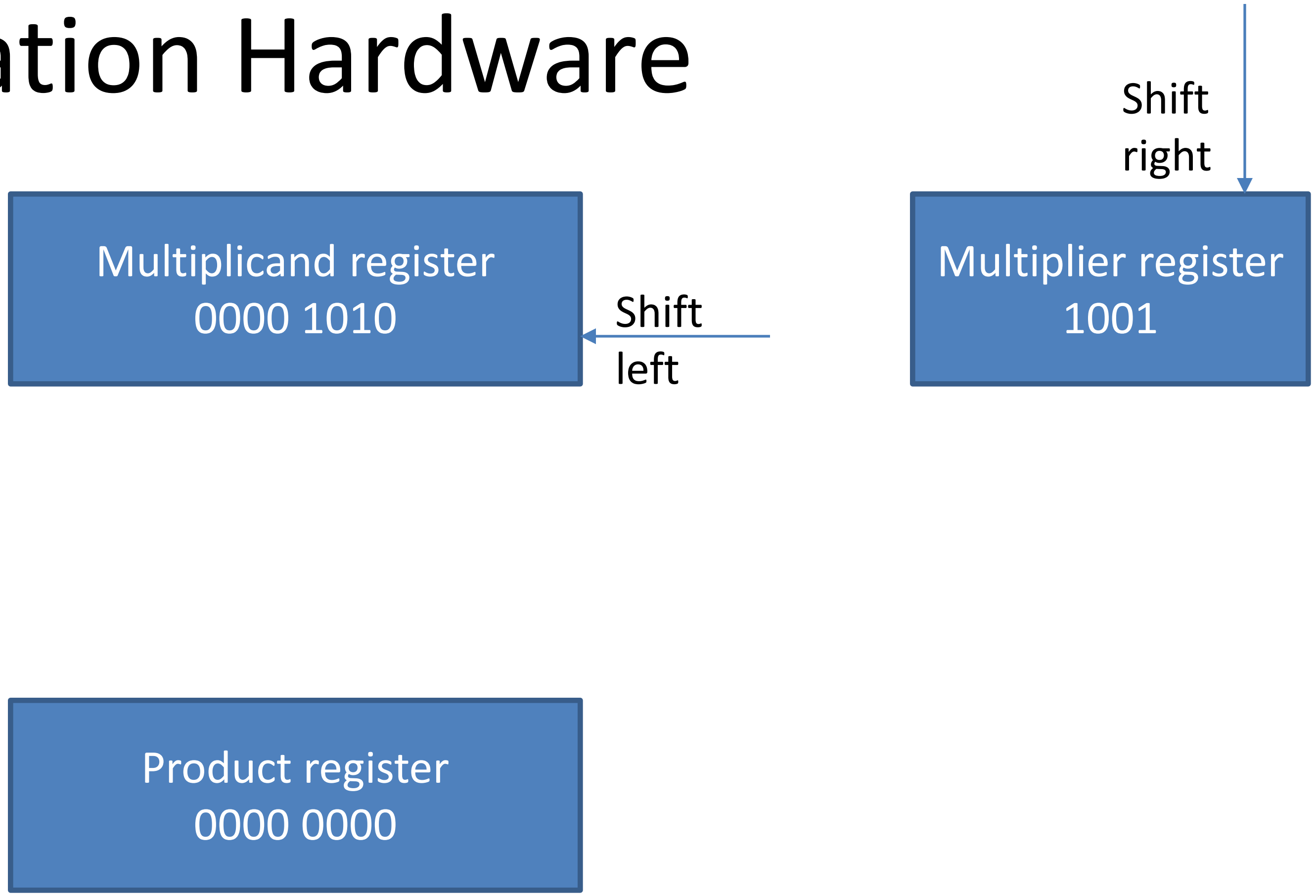
# Multiplication Hardware

- Multiplicand: 1010
- Multiplier: 1001
- Initialize multiplicand register
  - HW Shift left
- Initialize product register
- Initialize multiplier register, 4 bits



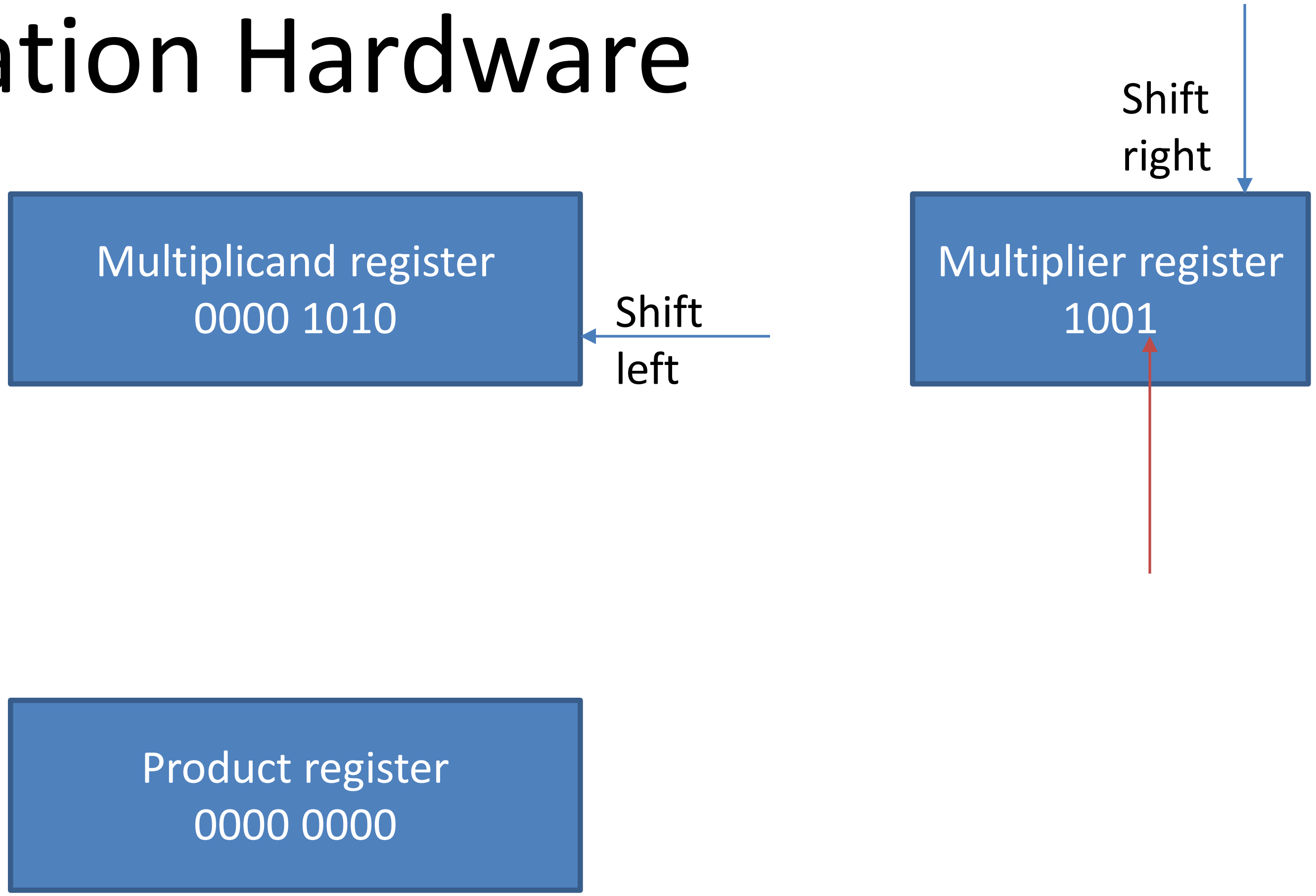
# Multiplication Hardware

- Multiplicand: 1010
- Multiplier: 1001
- Initialize multiplicand register
  - HW Shift left
- Initialize product register
- Initialize multiplier register, 4 bits
  - HW Shift right



# Multiplication Hardware

- Iteration 1
  - Check if multiplier bit is 0/1

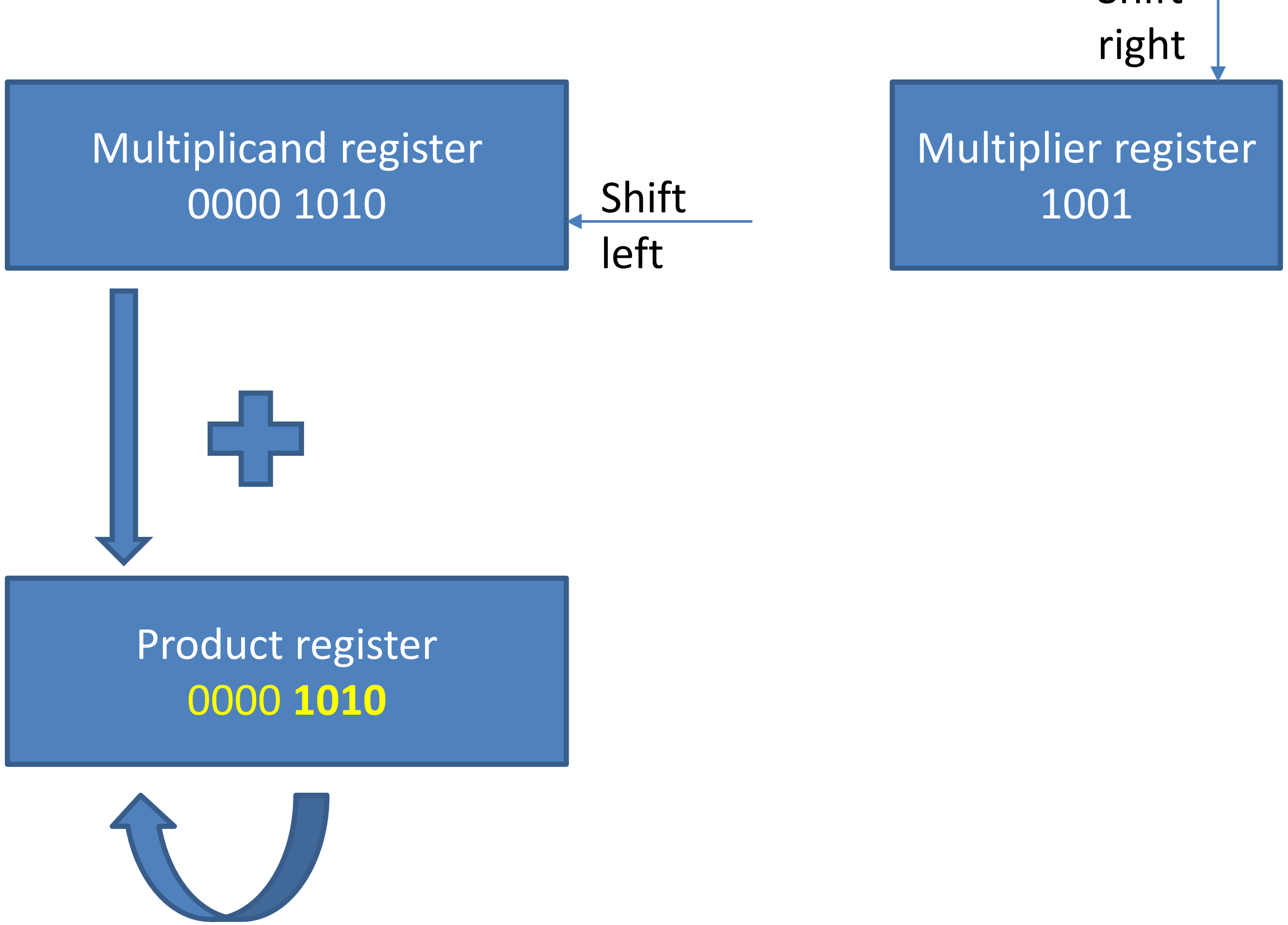



```
       1010
 ×     1001
 ----------
       1010
      0000
     0000
    1010
 ----------
    1011010
```

# Multiplication Hardware

- Iteration 1
  - **Check if multiplier bit is 0/1**
  - **If 1 add multiplicand to product register**
  - Else do nothing

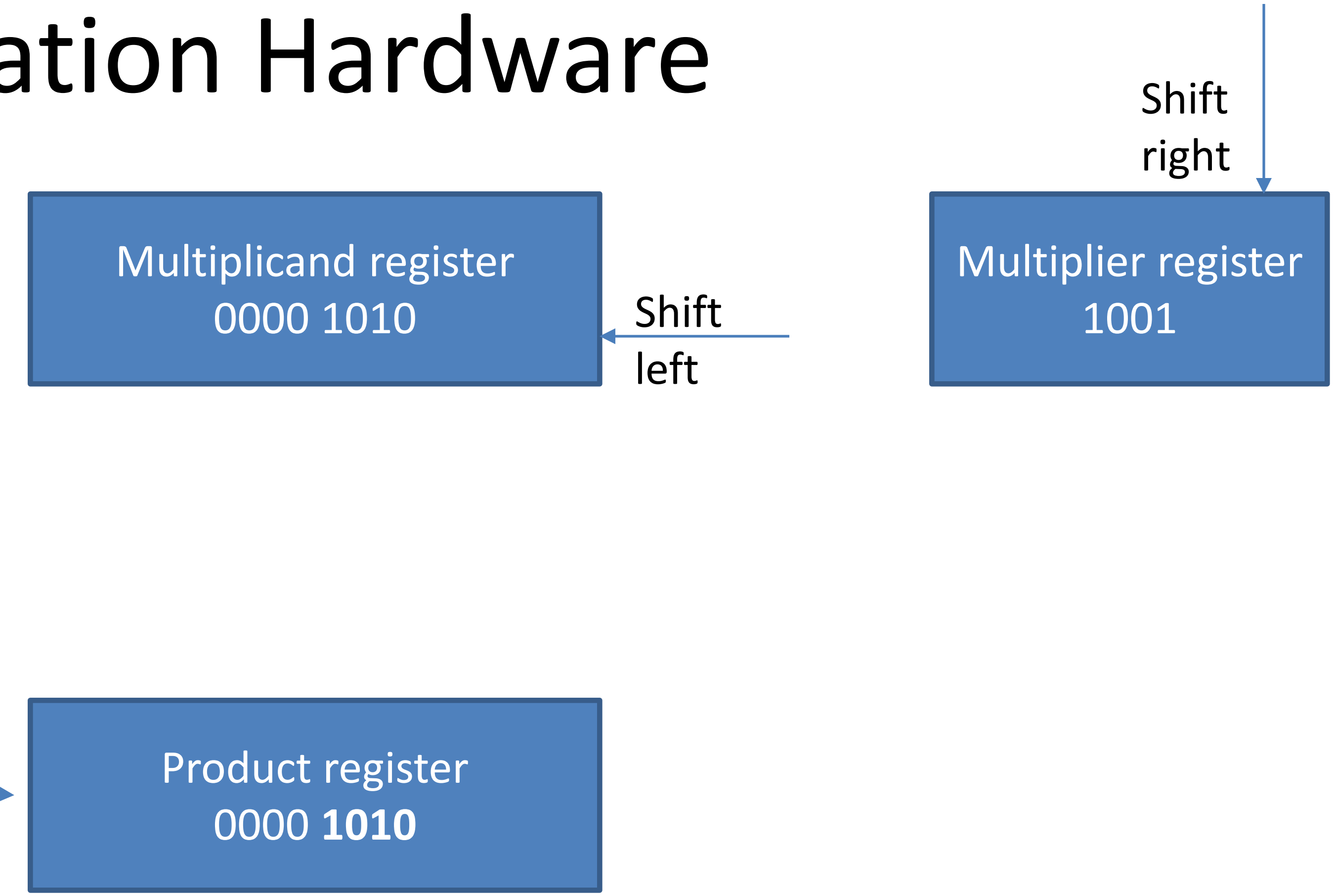
```
      1010
×     1001
----------
      1010
     0000
    0000
   1010
----------
   1011010
```



# Multiplication Hardware

- Iteration 1
  - Check if multiplier bit is 0/1
  - If 1 add multiplicand to product register
  - Else do nothing

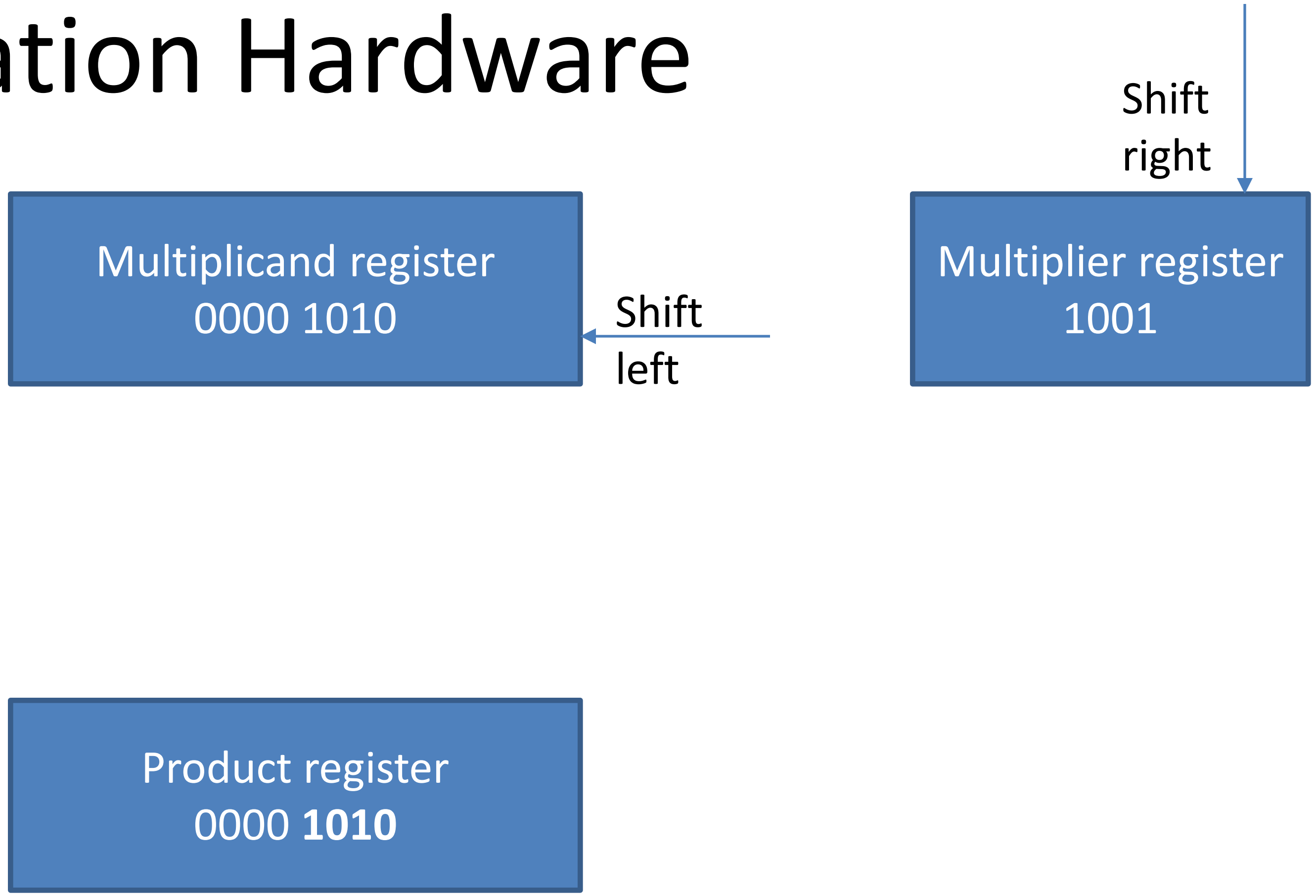
```
     1010
×    1001
---------
     1010
    0000
   0000
  1010
---------
  1011010
```



# Multiplication Hardware

- Iteration 1
  - Check if multiplier bit is 0/1
    - If 1 add multiplicand to product register
    - Else do nothing
  - Shift multiplicand to left by 1 bit

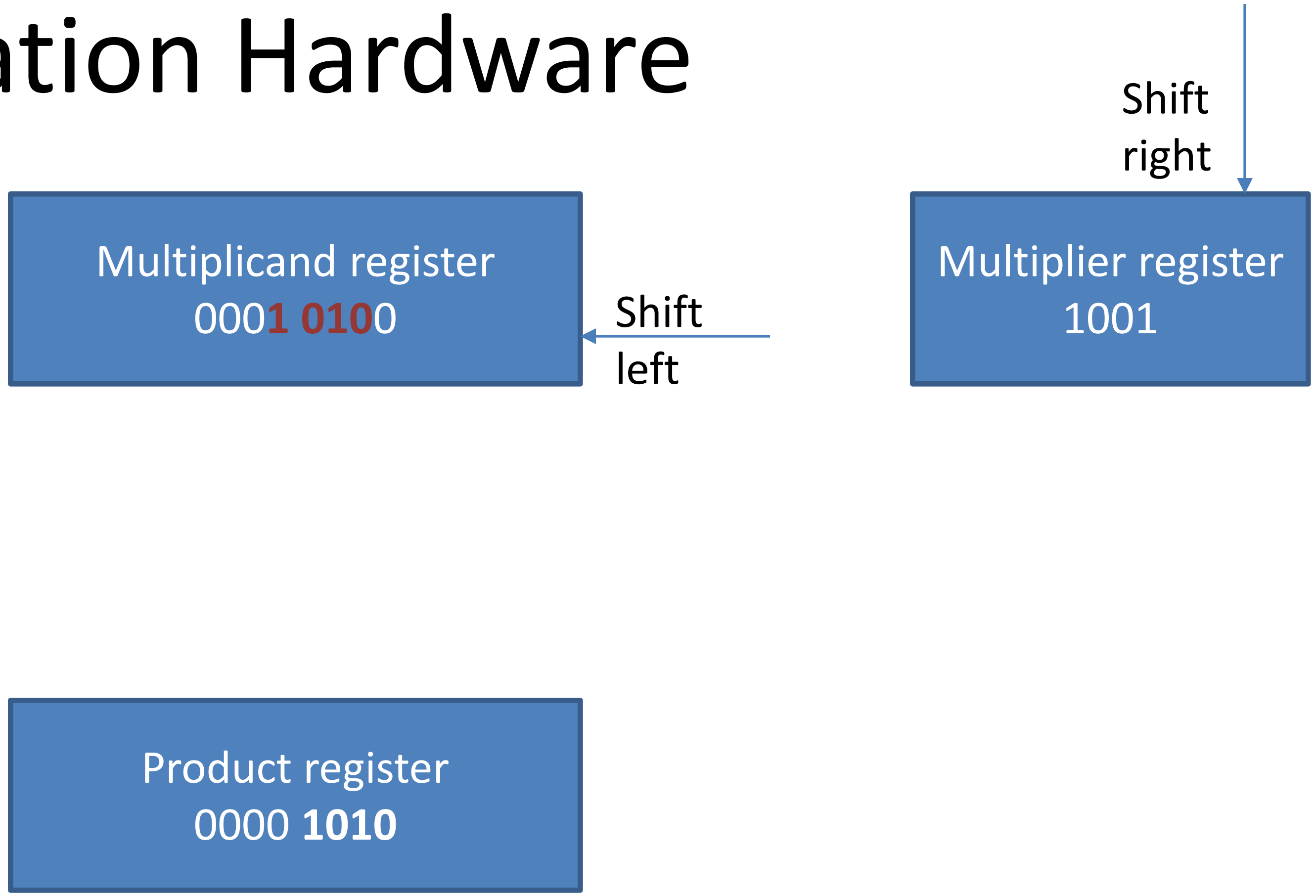
```
      1010
×     1001
----------
      1010
     0000
    0000
   1010
----------
   1011010
```



# Multiplication Hardware

- Iteration 1
  - Check if multiplier bit is 0/1
    - If 1 add multiplicand to product register
    - Else do nothing
  - Shift multiplicand to left by 1 bit

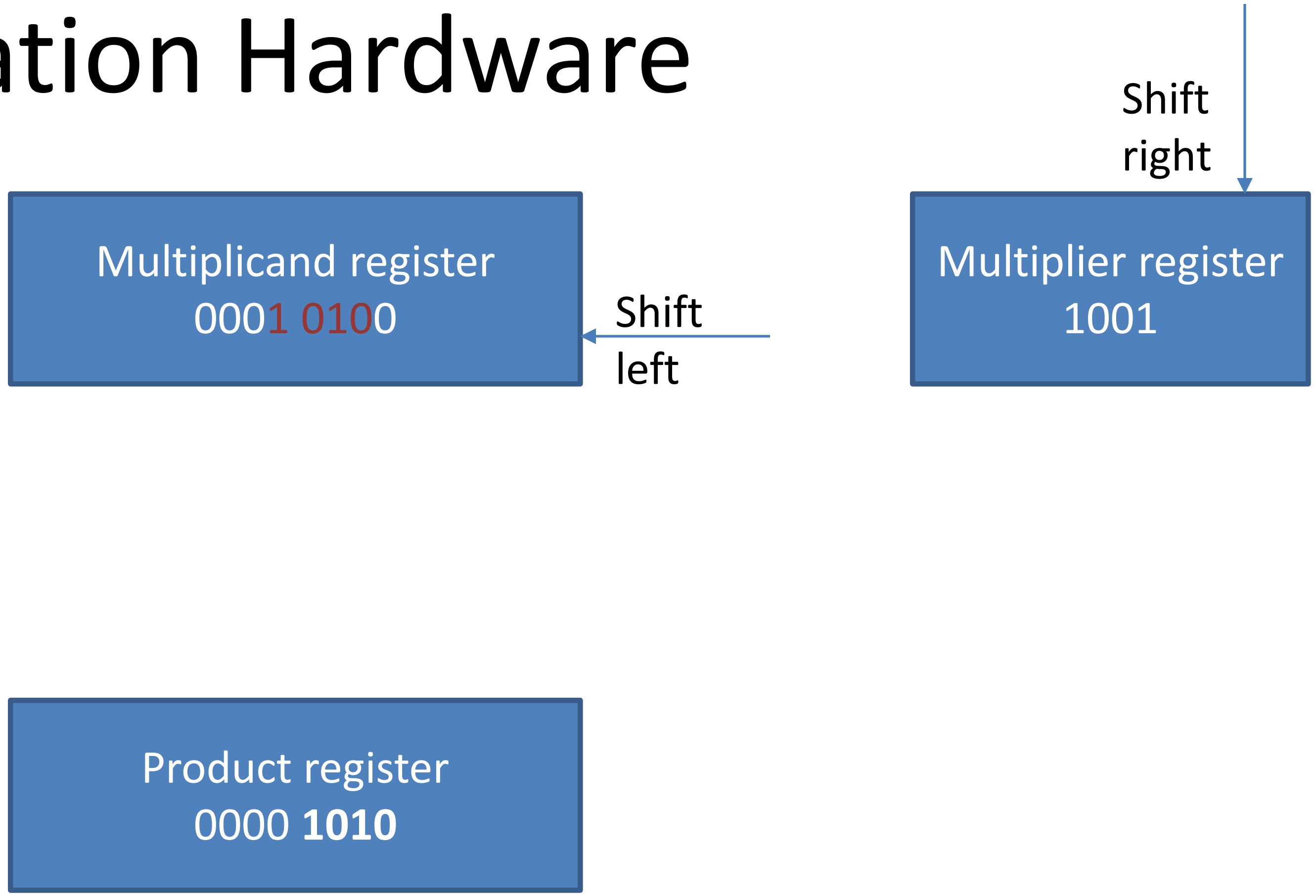
```
       1010
×      1001
-----------
       1010
      0000
     0000
    1010
-----------
    1011010
```



# Multiplication Hardware

- Iteration 1
  - Check if multiplier bit is 0/1
    - If 1 add multiplicand to product register
    - Else do nothing
  - Shift multiplicand to left by 1 bit
  - Shift multiplier 1 bit to the right

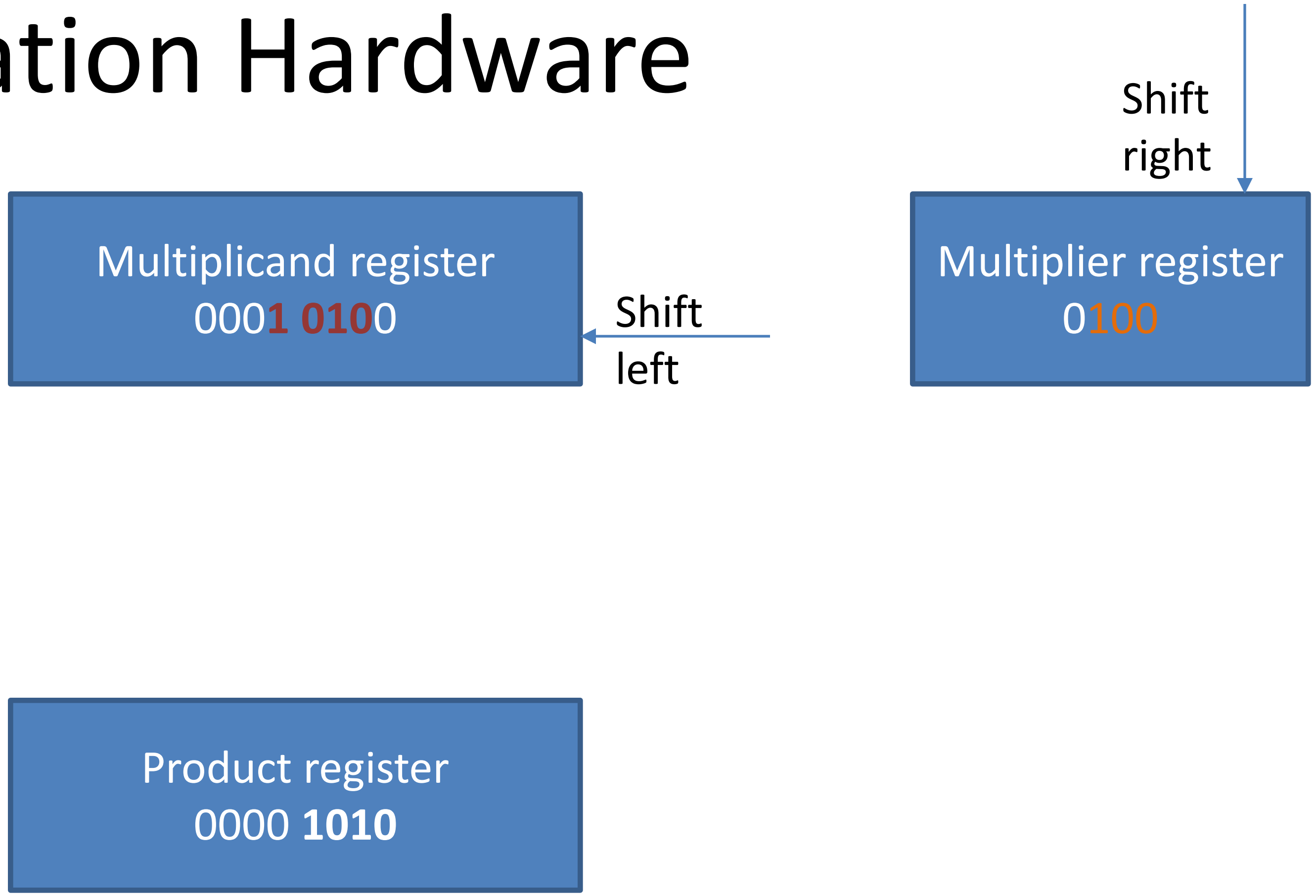
```
       1010
×      1001
-----------
       1010
      0000
     0000
    1010
-----------
    1011010
```



# Multiplication Hardware

- Iteration 1
  - Check if multiplier bit is 0/1
    - If 1 add multiplicand to product register
    - Else do nothing
  - Shift multiplicand to left by 1 bit
  - Shift multiplier 1 bit to the right

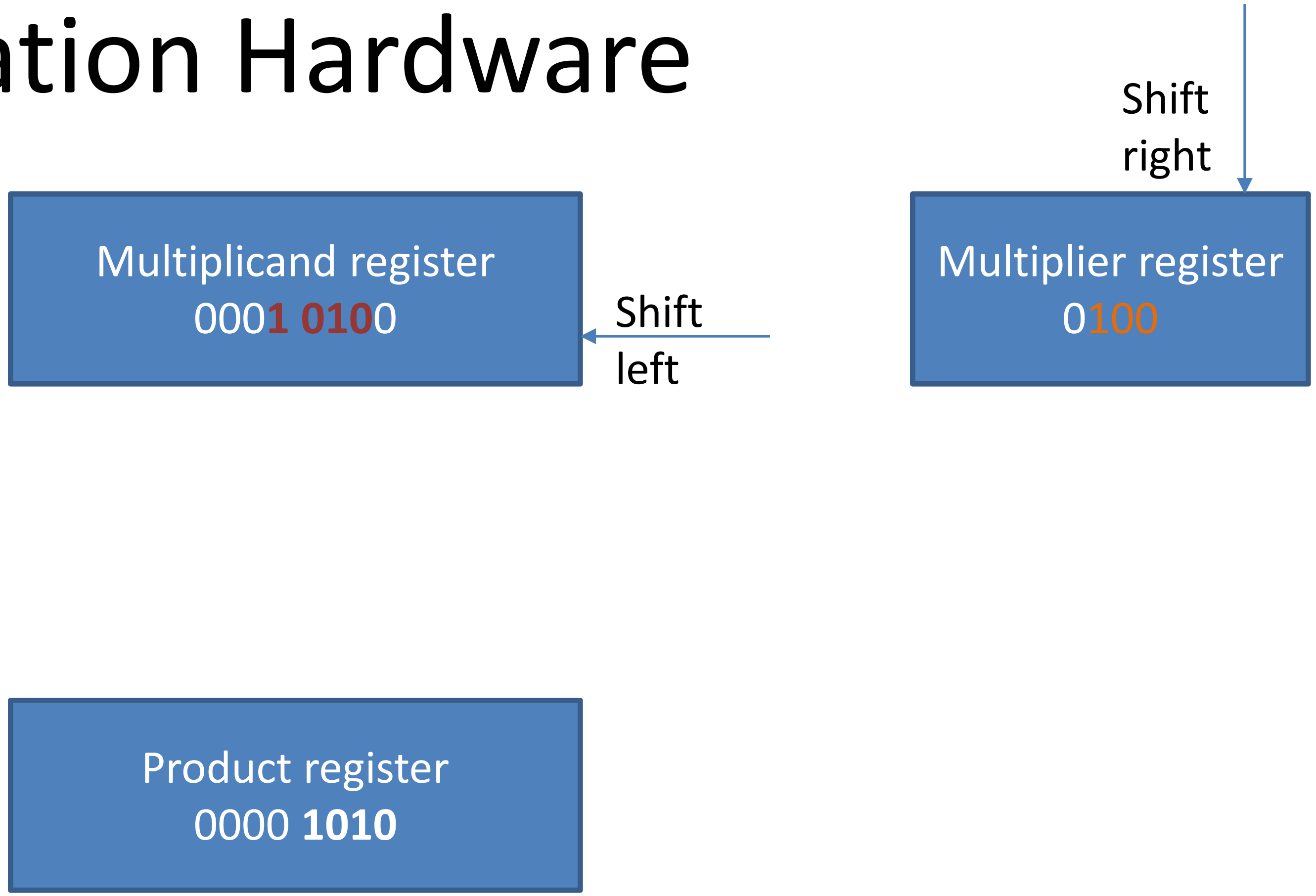
```
       1010
  ×    1001
  ---------
       1010
      0000
     0000
    1010
  ---------
    1011010
```



# Multiplication Hardware

- Iteration 1
  - Check if multiplier bit is 0/1
    - If 1 add multiplicand to product register
    - Else do nothing
  - Shift multiplicand to left by 1 bit
  - Shift multiplier 1 bit to the right
  - Repeat 4 times (total)

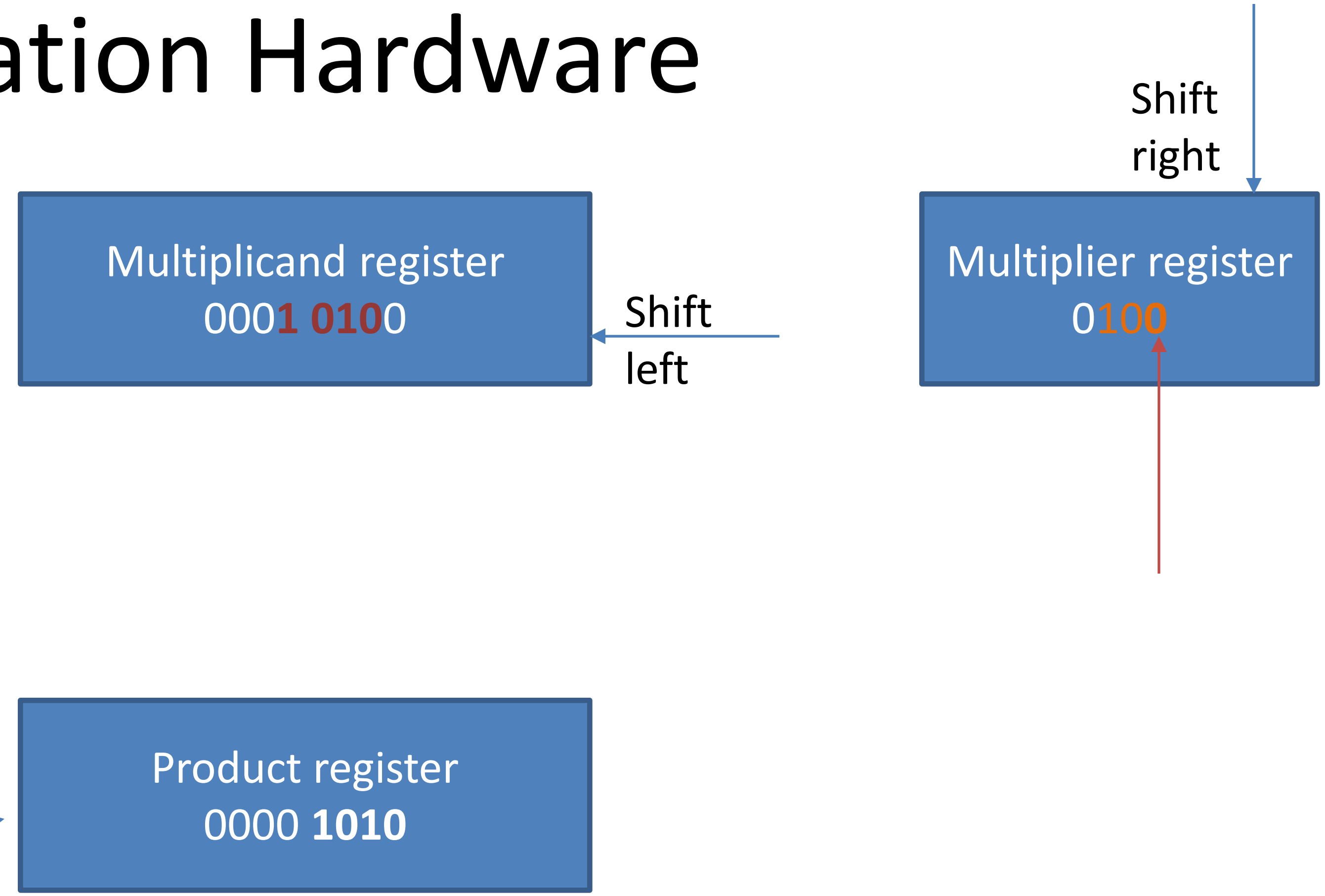
```
       1010
×      1001
-----------
       1010
      0000
     0000
    1010
-----------
    1011010
```



# Multiplication Hardware

- Iteration **2**
  - **Check if multiplier bit is 0/1**
    - If 1 add multiplicand to product register
    - **Else do nothing**
  - Shift multiplicand to left by 1 bit
  - Shift multiplier 1 bit to the right
  - Repeat 4 times (total)

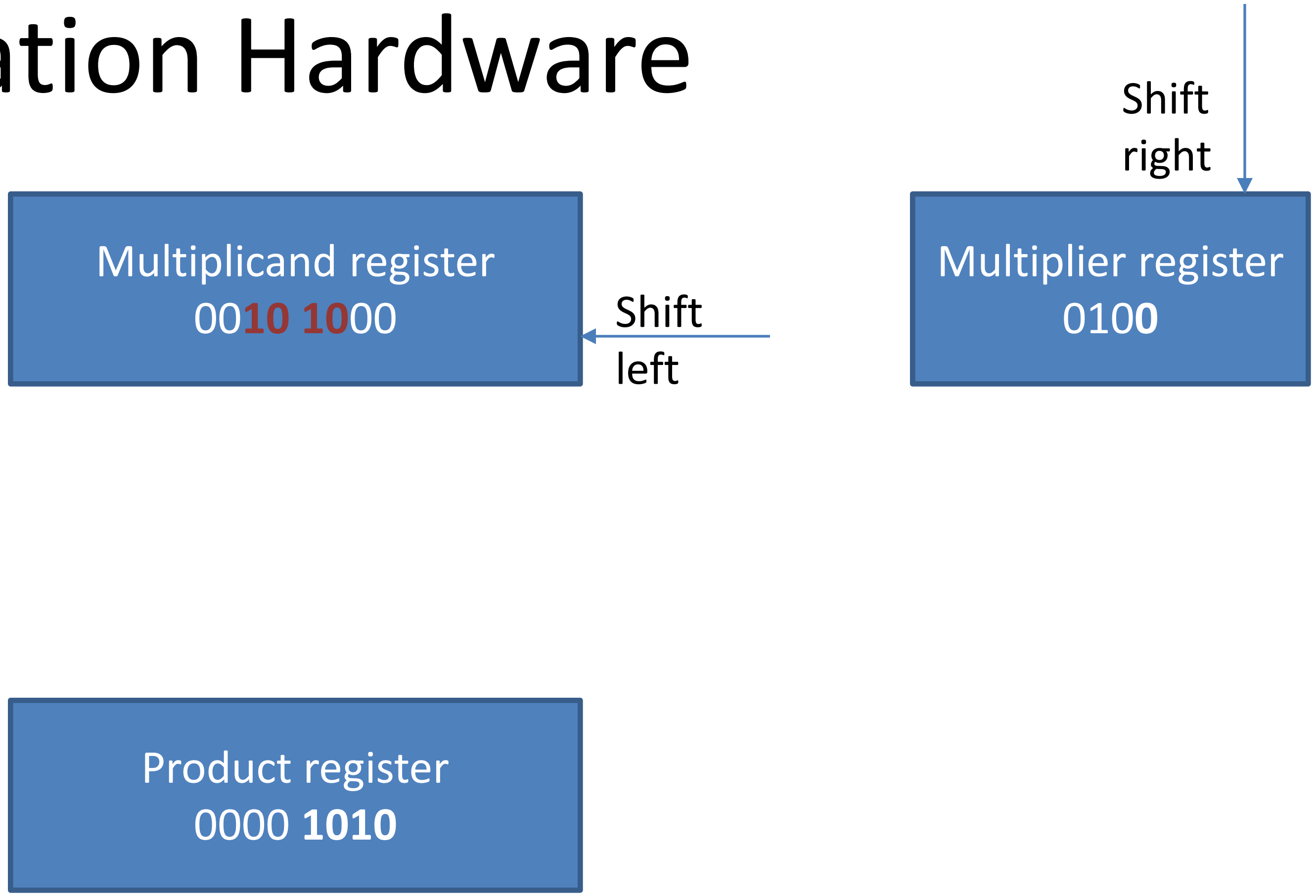
```
       1010
×      1001
-----------
       1010
      0000
     0000
    1010
-----------
    1011010
```



# Multiplication Hardware

- Iteration 2
  - Check if multiplier bit is 0/1
    - If 1 add multiplicand to product register
    - Else do nothing
  - **Shift multiplicand to left by 1 bit**
  - Shift multiplier 1 bit to the right
  - Repeat 4 times (total)

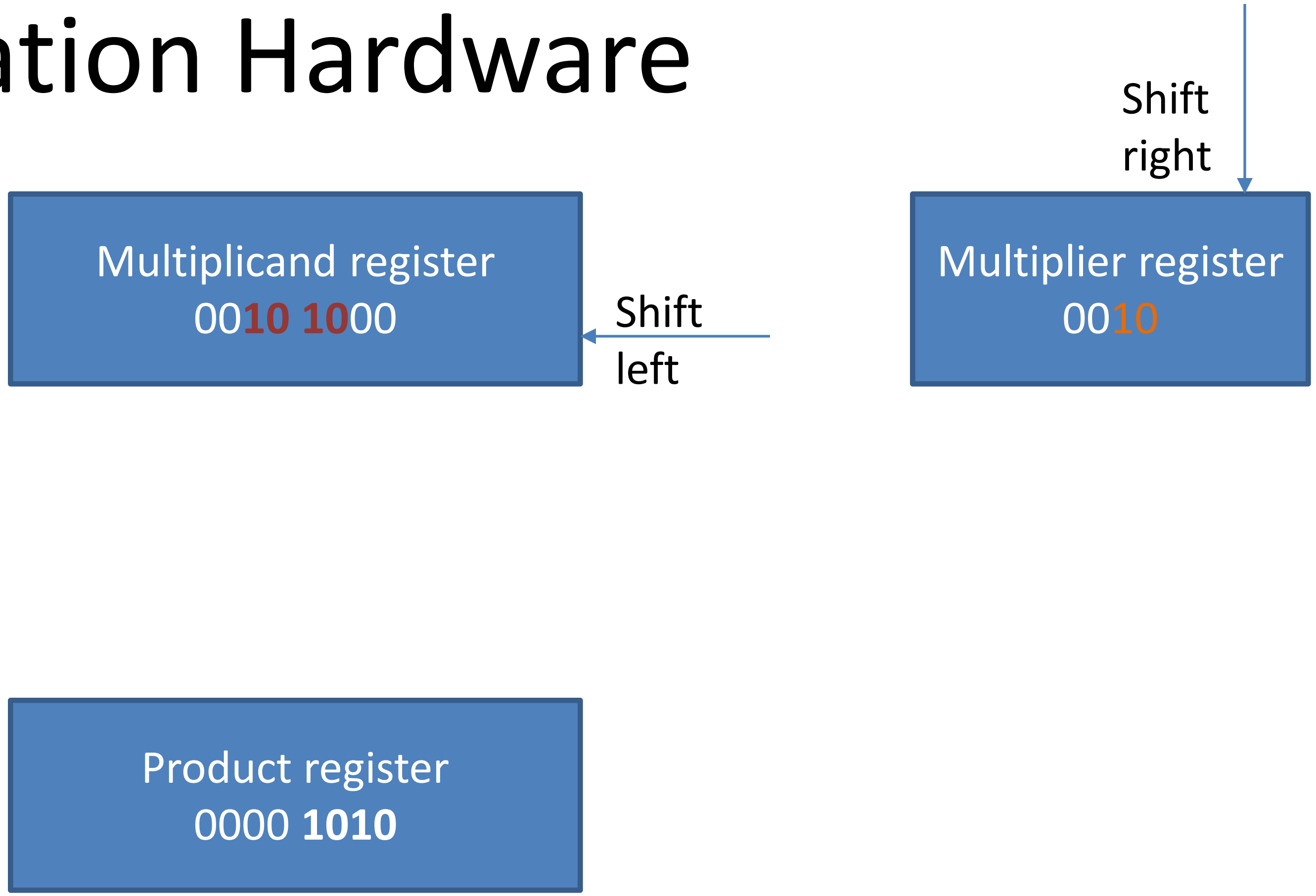
```
      1010
×     1001
----------
      1010
     0000
    0000
   1010
----------
   1011010
```



# Multiplication Hardware

- Iteration 2
  - Check if multiplier bit is 0/1
    - If 1 add multiplicand to product register
    - Else do nothing
  - Shift multiplicand to left by 1 bit
  - **Shift multiplier 1 bit to the right**
  - Repeat 4 times (total)

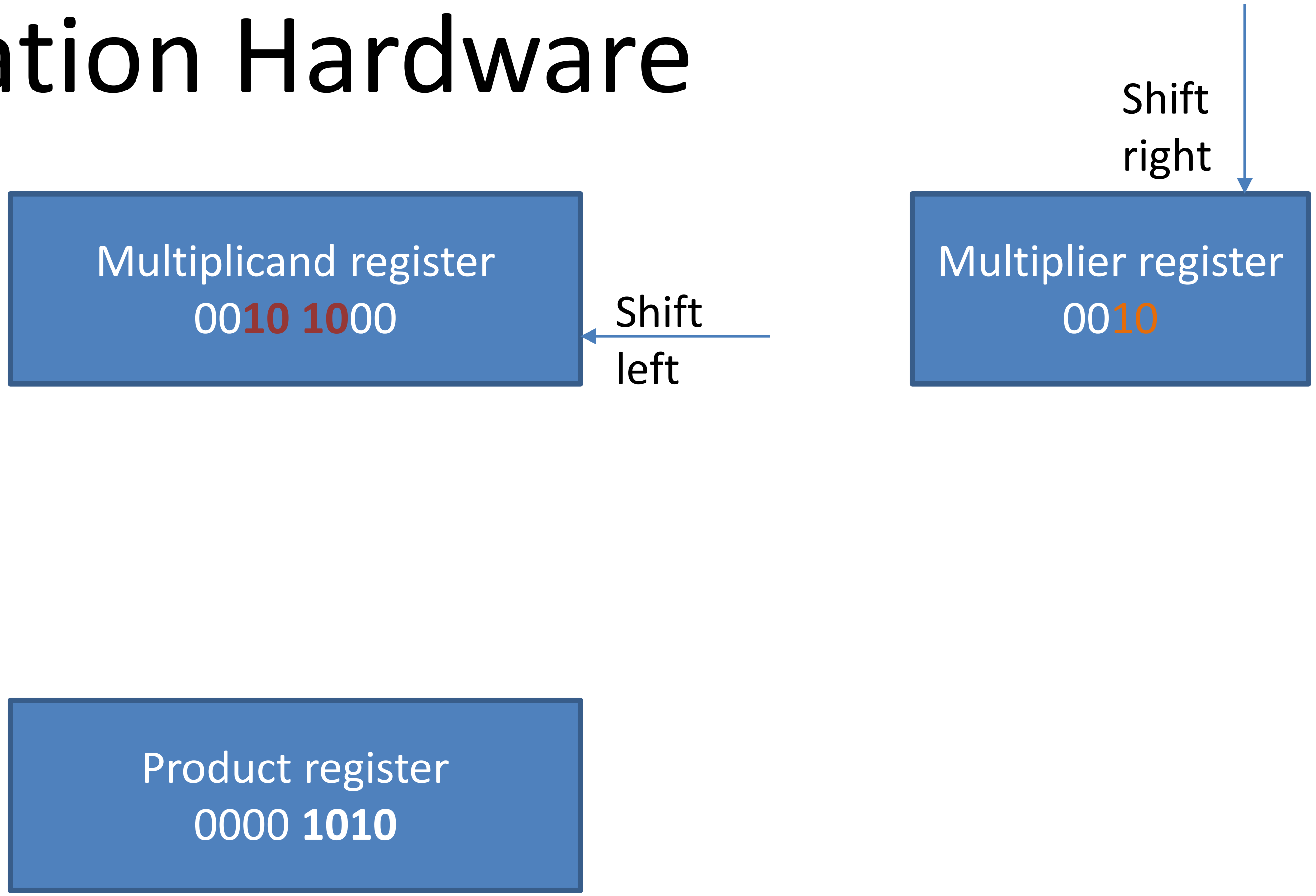
```
      1010
×     1001
----------
      1010
     0000
    0000
   1010
----------
   1011010
```



# Multiplication Hardware

- Iteration 2
  - Check if multiplier bit is 0/1
    - If 1 add multiplicand to product register
    - Else do nothing
  - Shift multiplicand to left by 1 bit
  - Shift multiplier 1 bit to the right
  - Repeat 4 times (total)

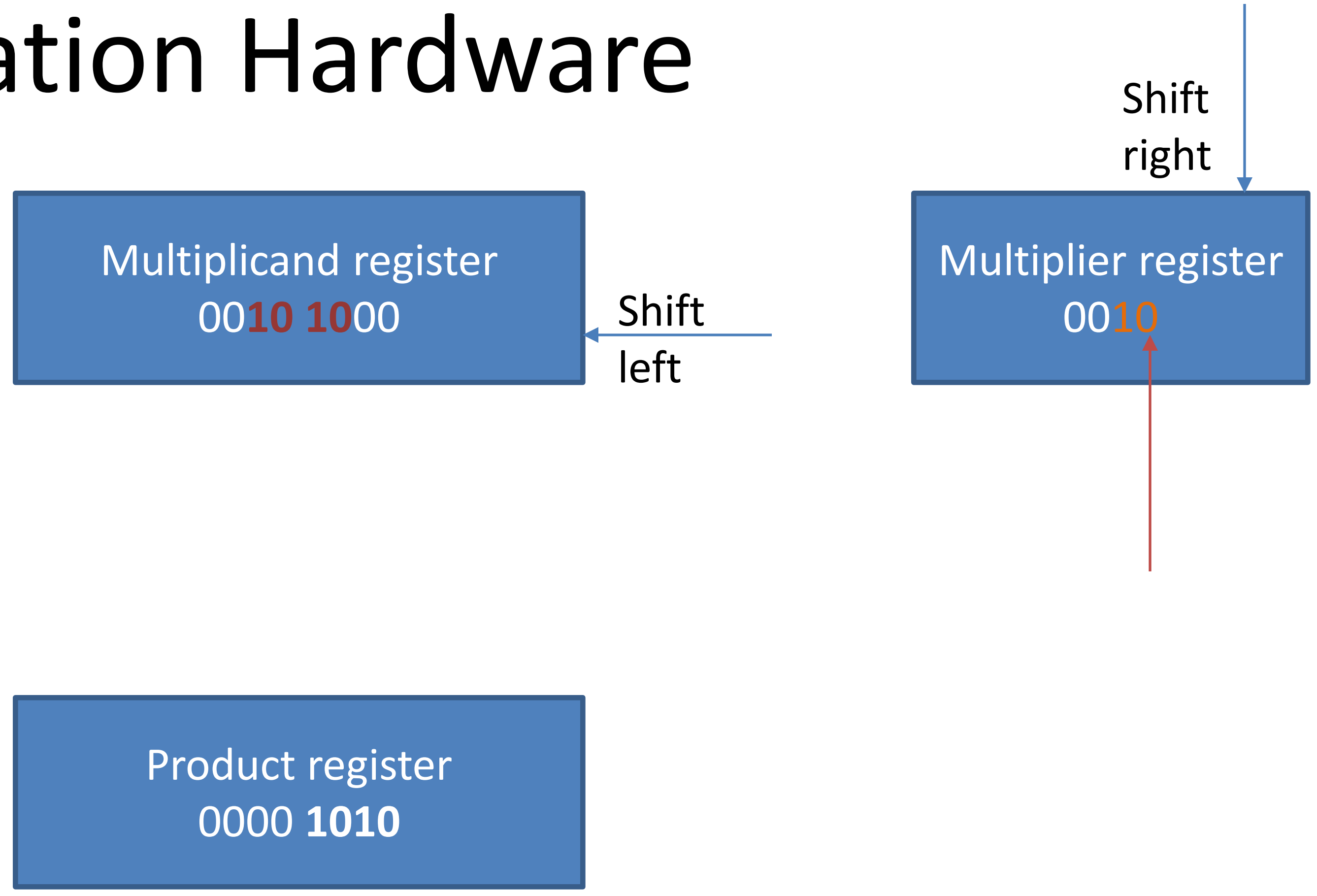
```
      1010
×     1001
----------
      1010
     0000
    0000
   1010
----------
   1011010
```



# Multiplication Hardware

- Iteration 3
  - **Check if multiplier bit is 0/1**
    - If 1 add multiplicand to product register
    - **Else do nothing**
  - Shift multiplicand to left by 1 bit
  - Shift multiplier 1 bit to the right
  - Repeat 4 times (total)

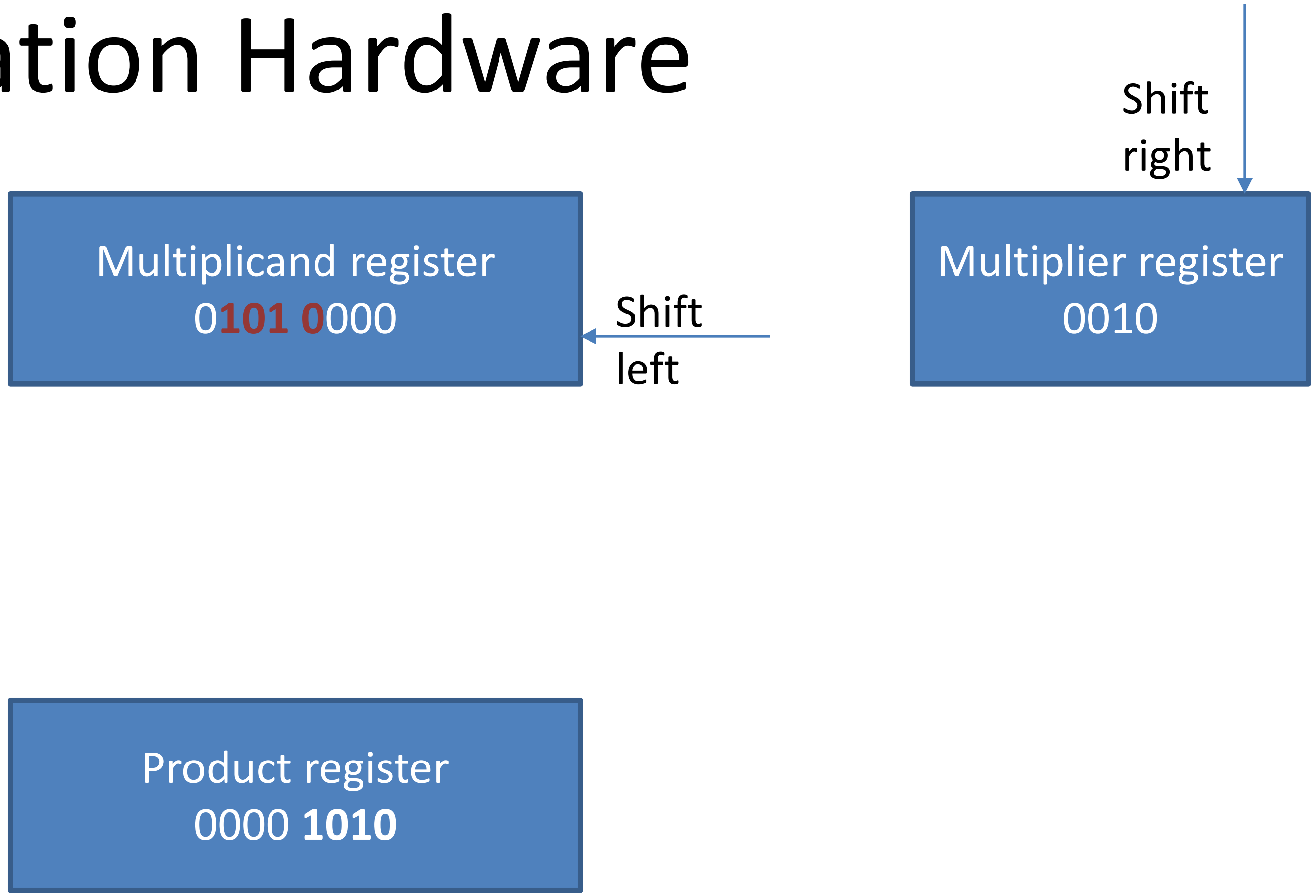
```
      1010
 ×    1001
 ---------
      1010
     0000
    0000
   1010
 ---------
   1011010
```



# Multiplication Hardware

- Iteration 3
  - Check if multiplier bit is 0/1
    - If 1 add multiplicand to product register
    - Else do nothing
  - **Shift multiplicand to left by 1 bit**
  - Shift multiplier 1 bit to the right
  - Repeat 4 times (total)

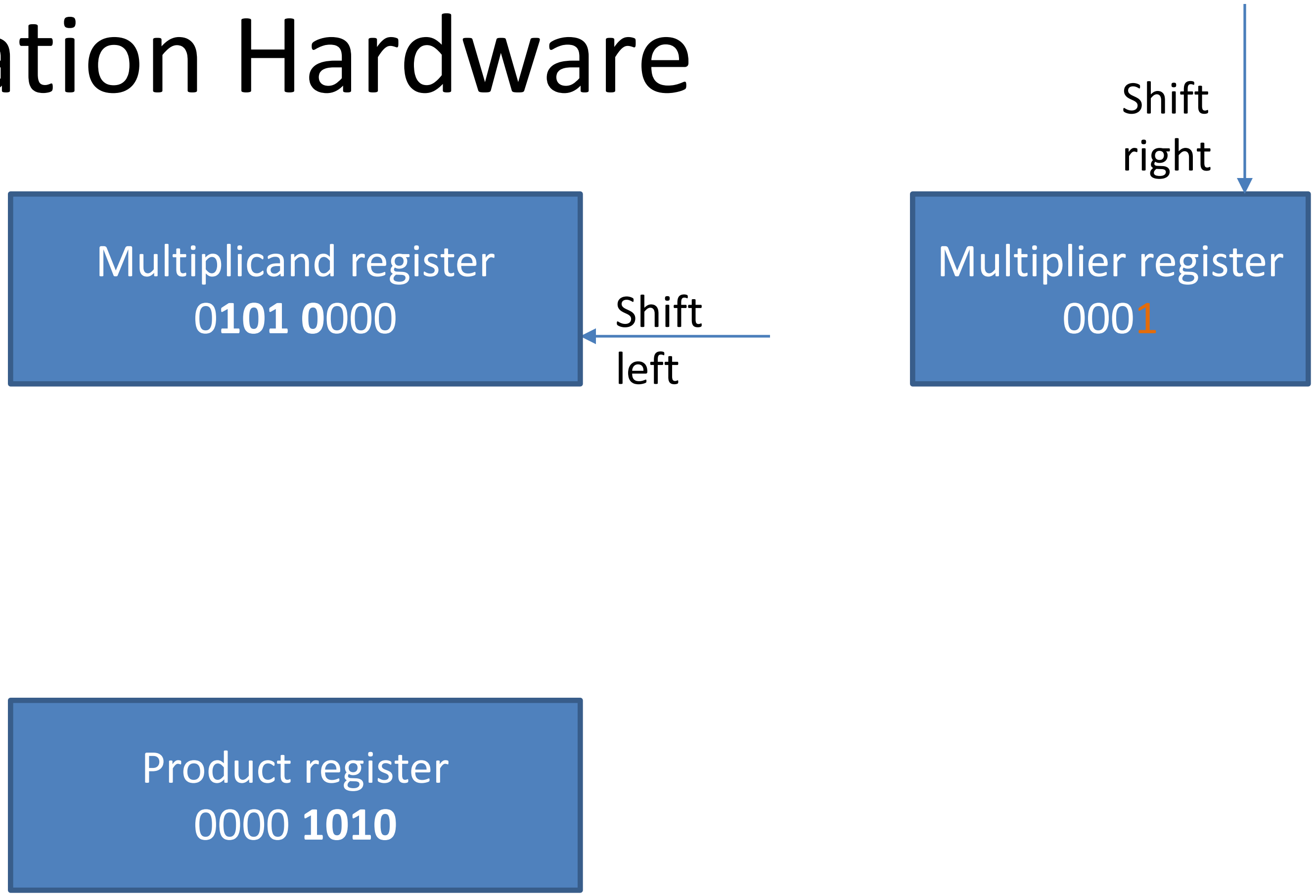
```
       1010
  ×    1001
  ---------
       1010
      0000
     0000
    1010
  ---------
    1011010
```



# Multiplication Hardware

- Iteration 3
  - Check if multiplier bit is 0/1
    - If 1 add multiplicand to product register
    - Else do nothing
  - Shift multiplicand to left by 1 bit
  - **Shift multiplier 1 bit to the right**
  - Repeat 4 times (total)

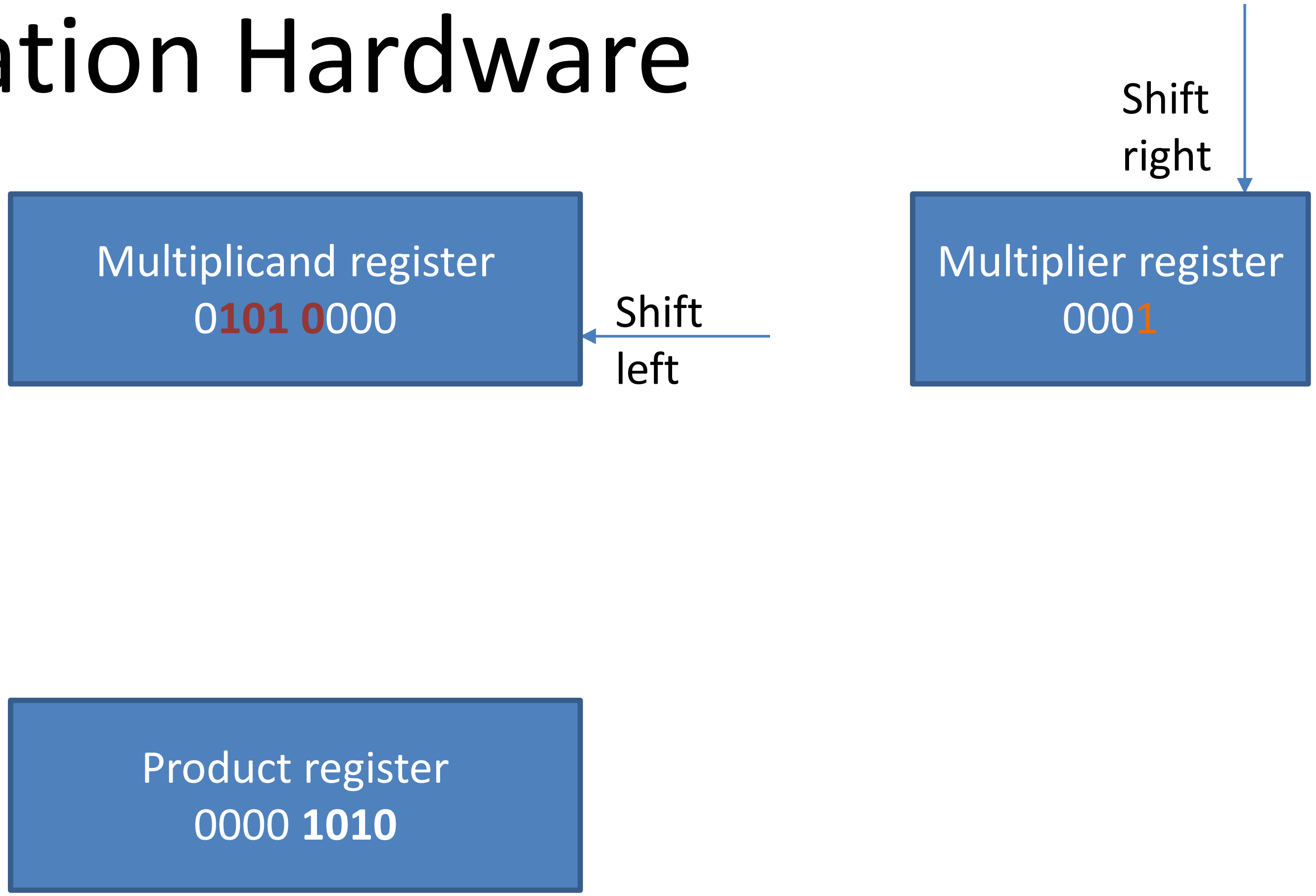
```
     1010
×    1001
---------
     1010
    0000
   0000
  1010
---------
  1011010
```



# Multiplication Hardware

- Iteration 3
  - Check if multiplier bit is 0/1
    - If 1 add multiplicand to product register
    - Else do nothing
  - Shift multiplicand to left by 1 bit
  - Shift multiplier 1 bit to the right
  - Repeat 4 times (total)

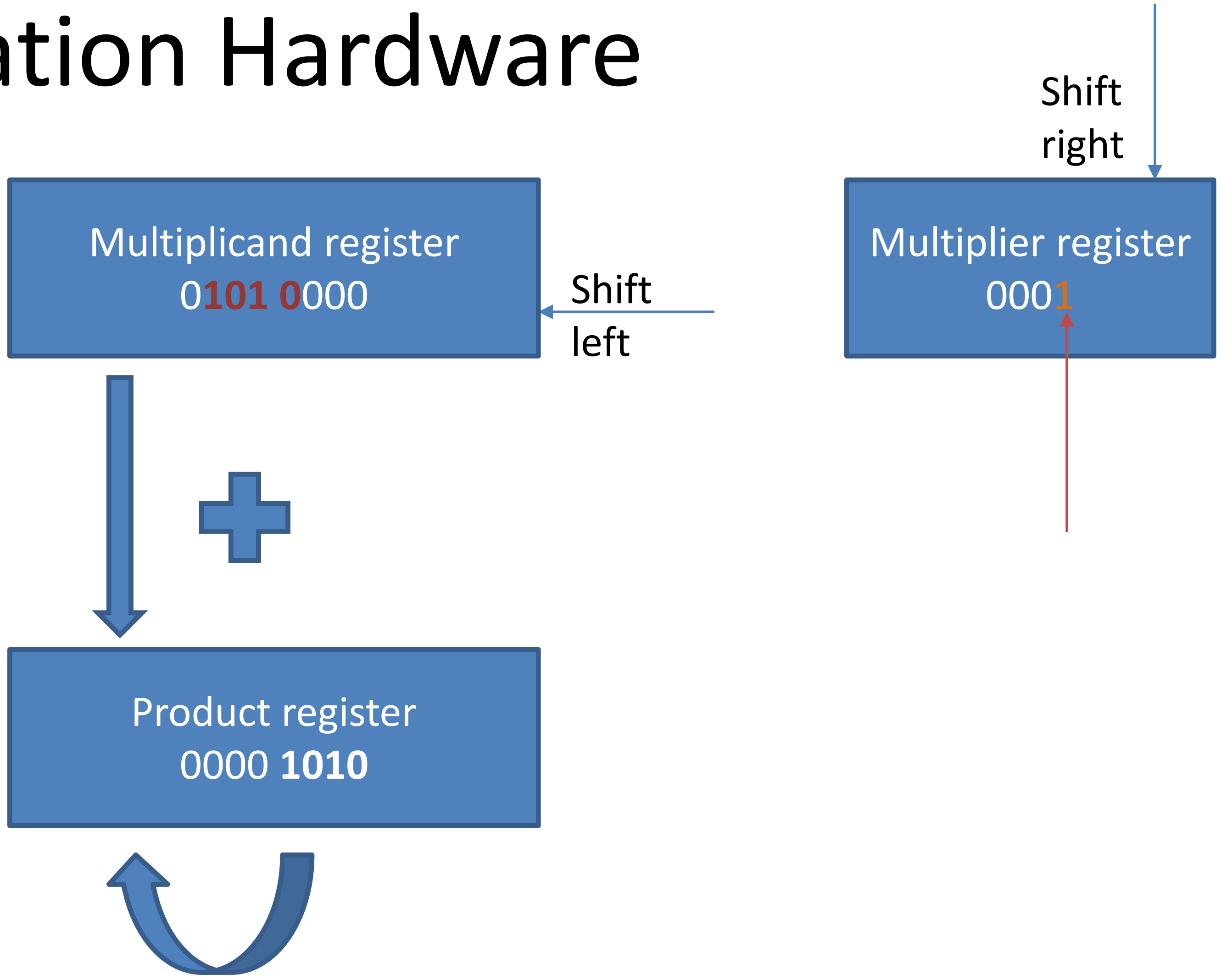
```
      1010
×     1001
----------
      1010
     0000
    0000
   1010
----------
   1011010
```



# Multiplication Hardware

- Iteration 4
  - **Check if multiplier bit is 0/1**
    - **If 1 add multiplicand to product register**
    - Else do nothing
  - Shift multiplicand to left by 1 bit
  - Shift multiplier 1 bit to the right
  - Repeat 4 times (total)

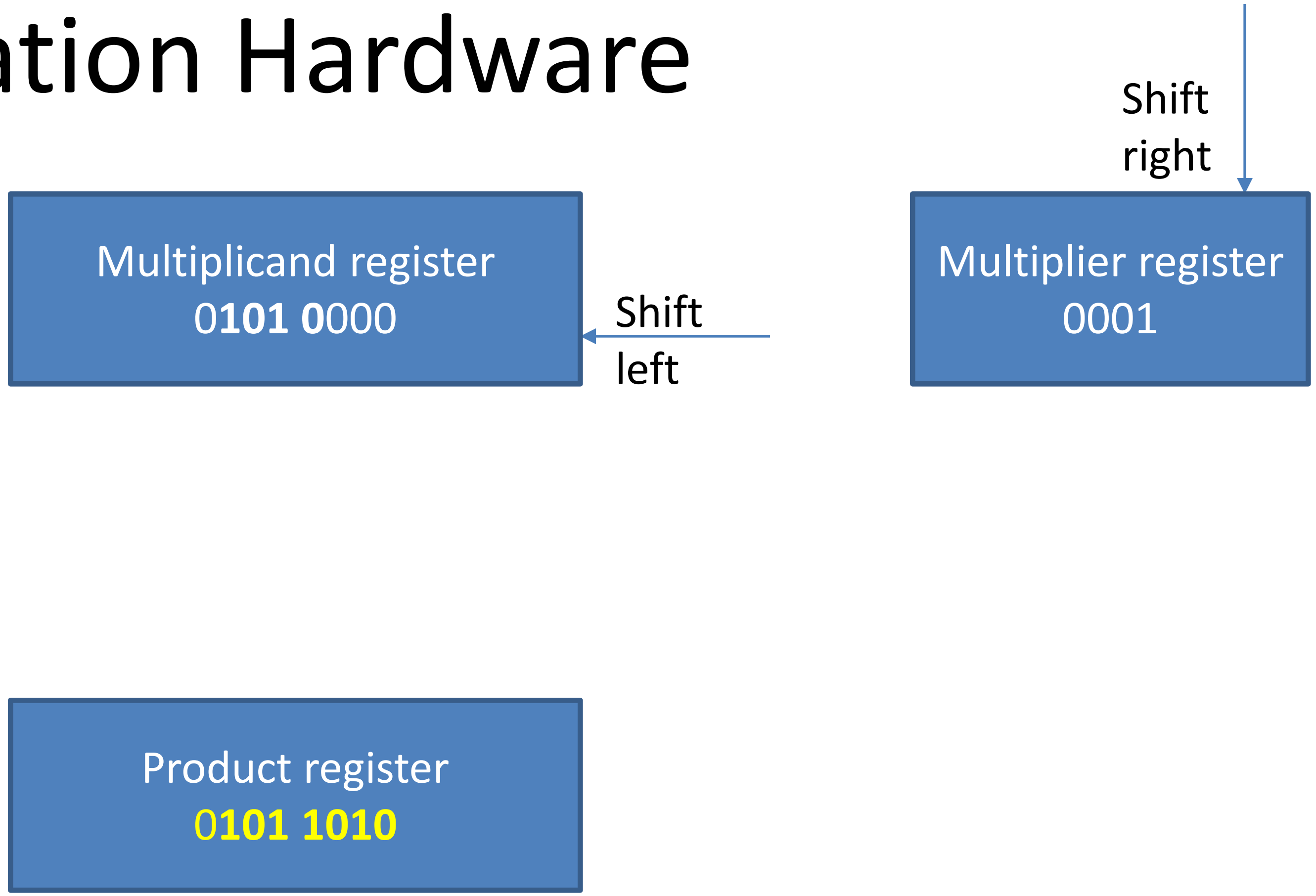
```
       1010
×      1001
-----------
       1010
      0000
     0000
    1010
-----------
    1011010
```



# Multiplication Hardware

- Iteration 4
  - Check if multiplier bit is 0/1
    - If 1 add multiplicand to product register
    - Else do nothing
  - Shift multiplicand to left by 1 bit
  - Shift multiplier 1 bit to the right
  - Repeat 4 times (total)

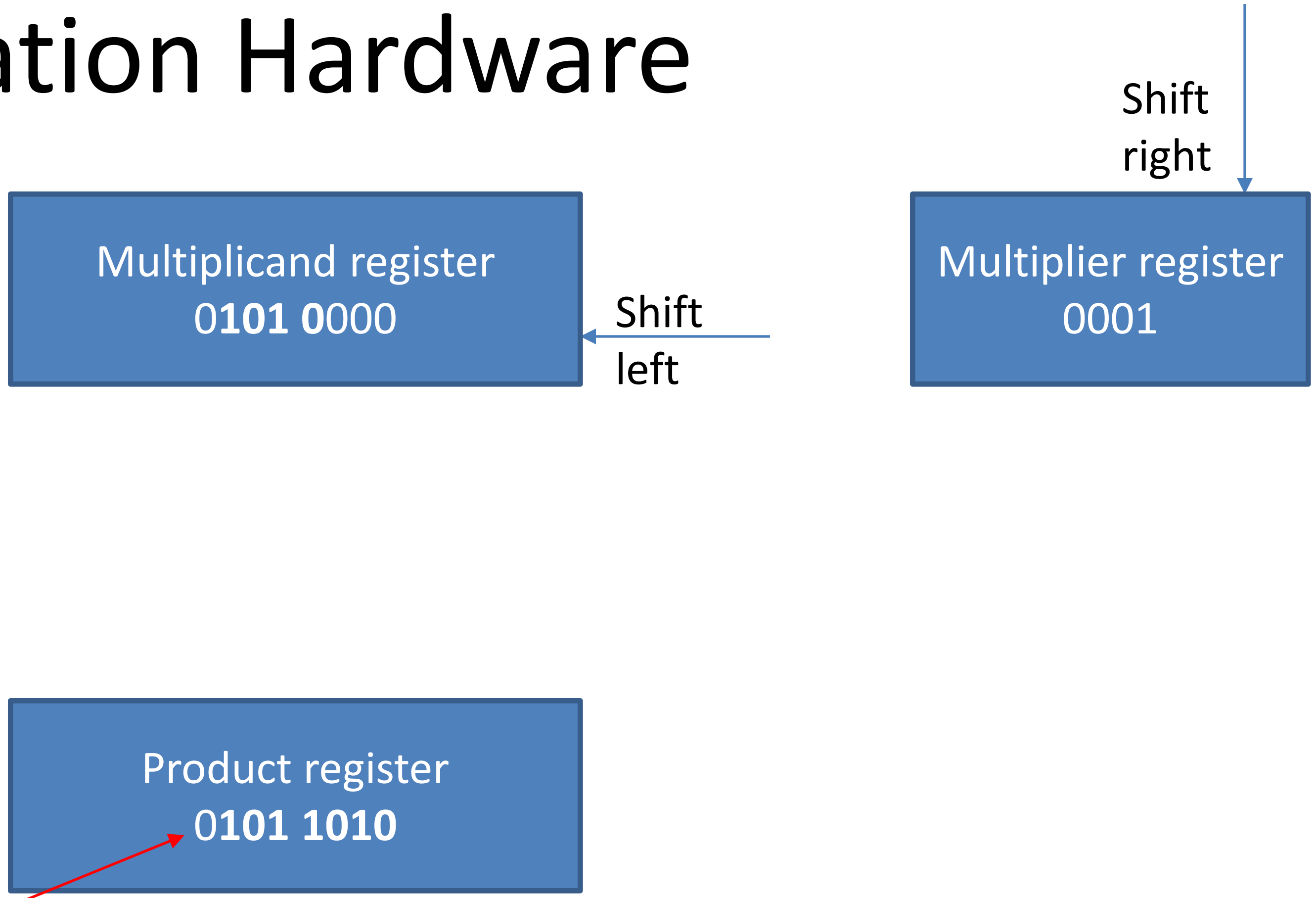
```
       1010
×      1001
-----------
       1010
      0000
     0000
    1010
-----------
    1011010
```



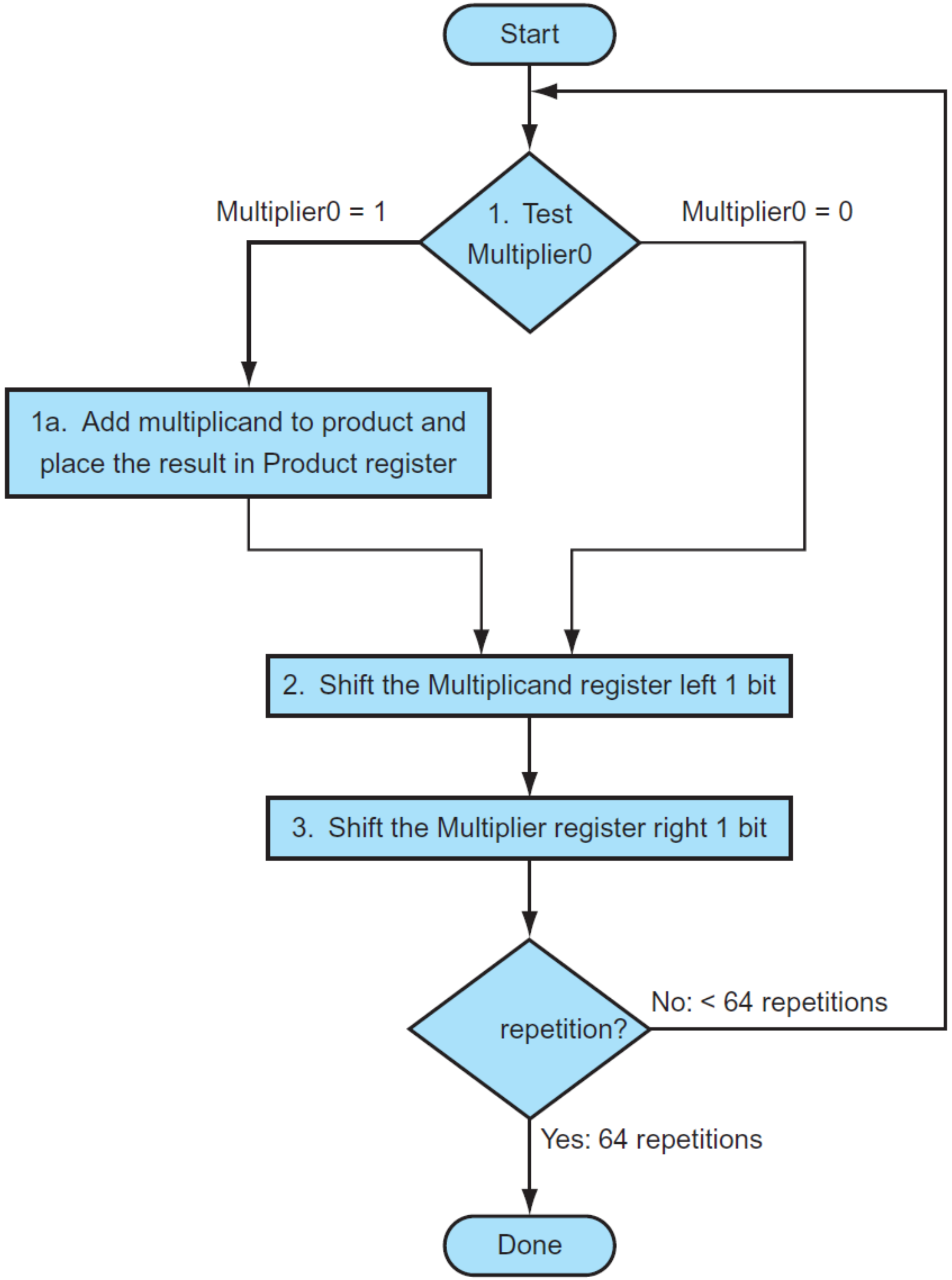
# Multiplication Hardware

- Iteration **4**
  - Check if multiplier bit is 0/1
    - If 1 add multiplicand to product register
    - Else do nothing
  - Shift multiplicand to left by 1 bit
  - Shift multiplier 1 bit to the right
  - Repeat **4 times (total)**
  - Exit

```
       1010
×      1001
-----------
       1010
      0000
     0000
    1010
-----------
    1011010
```

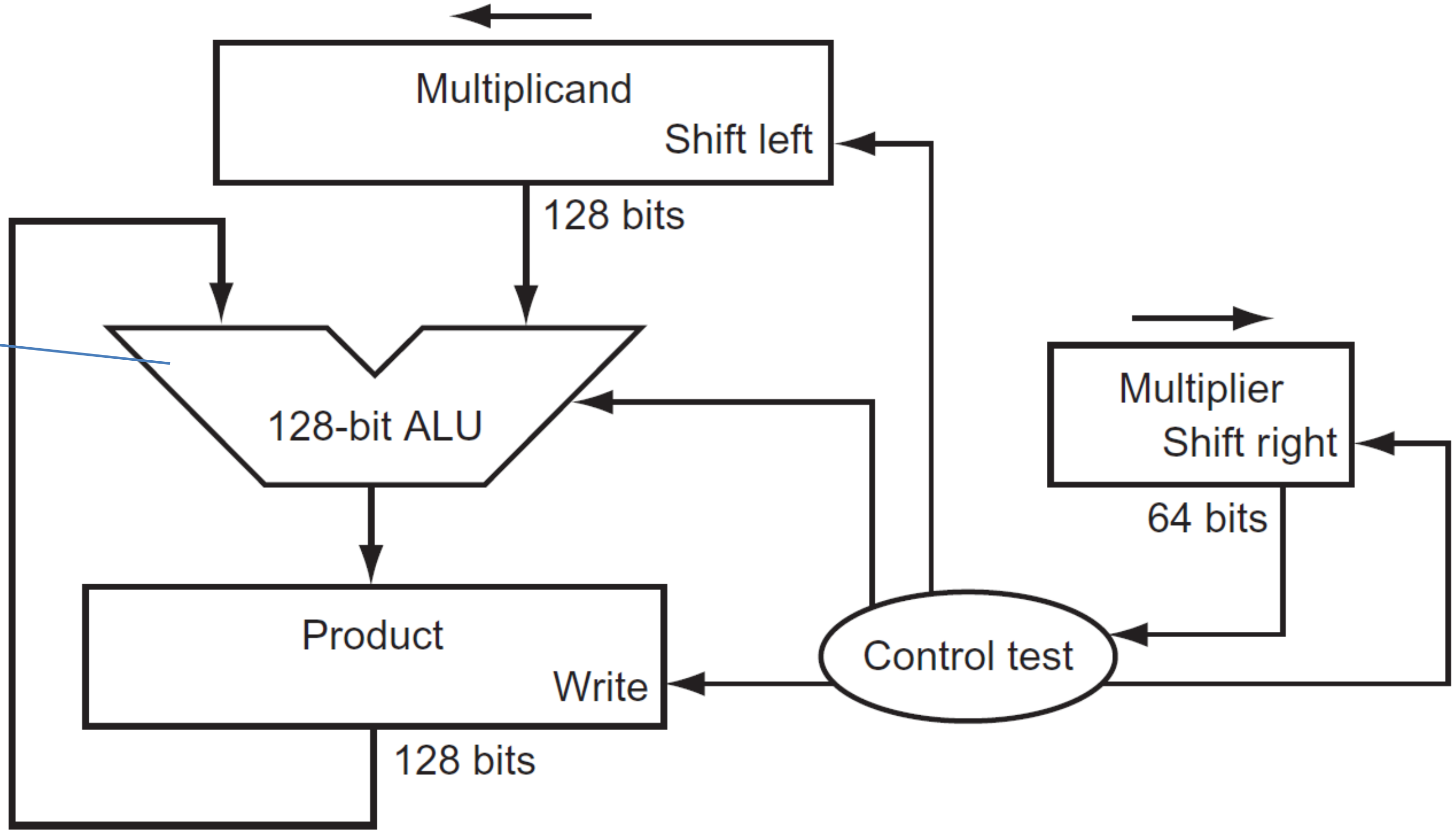


# Multiplication Flowchart



# Multiplication Hardware

Think of this as an Adder



# Multiplication Hardware

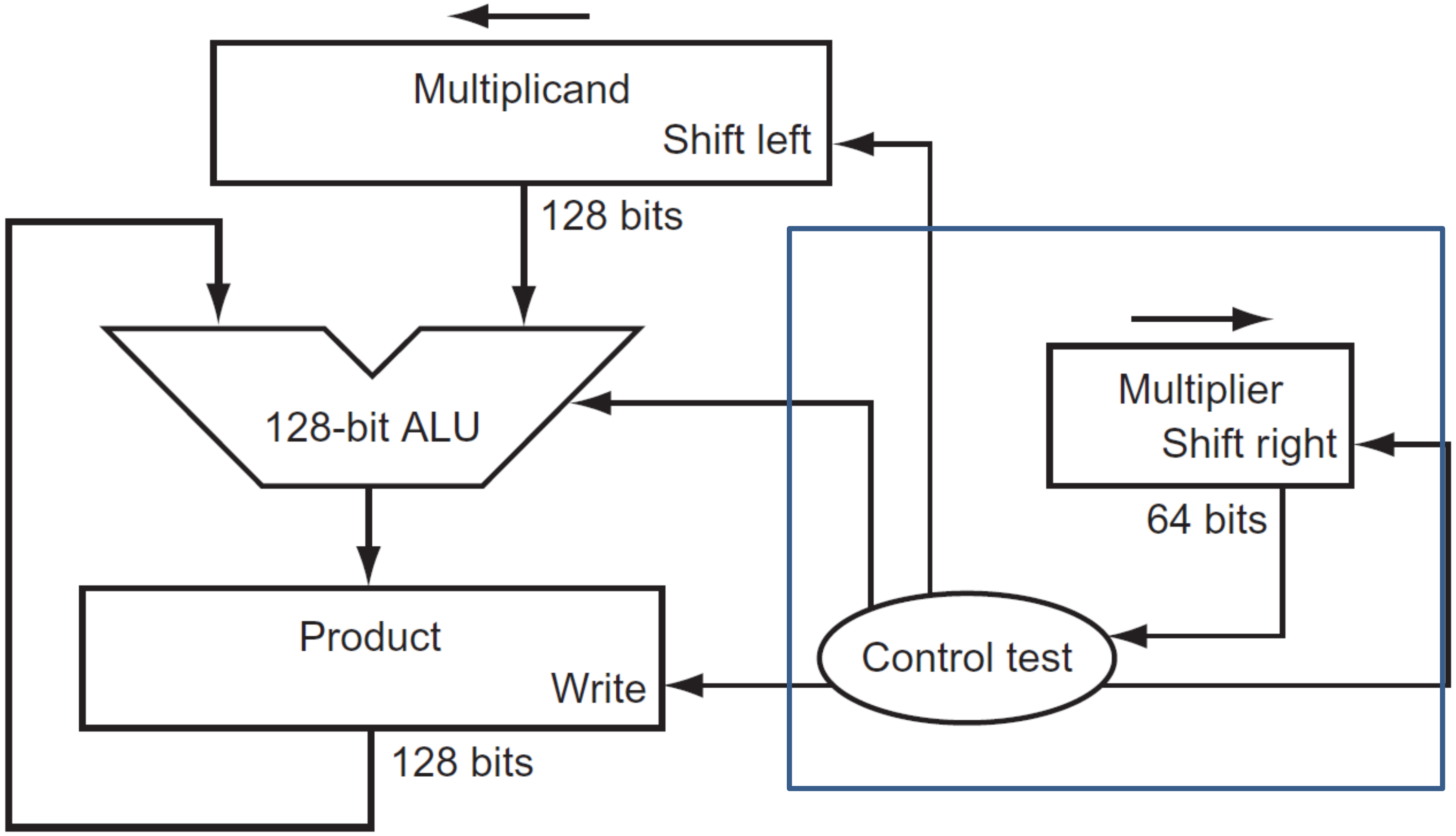
**Check if multiplier bit is 0/1**

- If 1 add multiplicand to product register
- Else do nothing

Shift multiplicand 1 bit to left

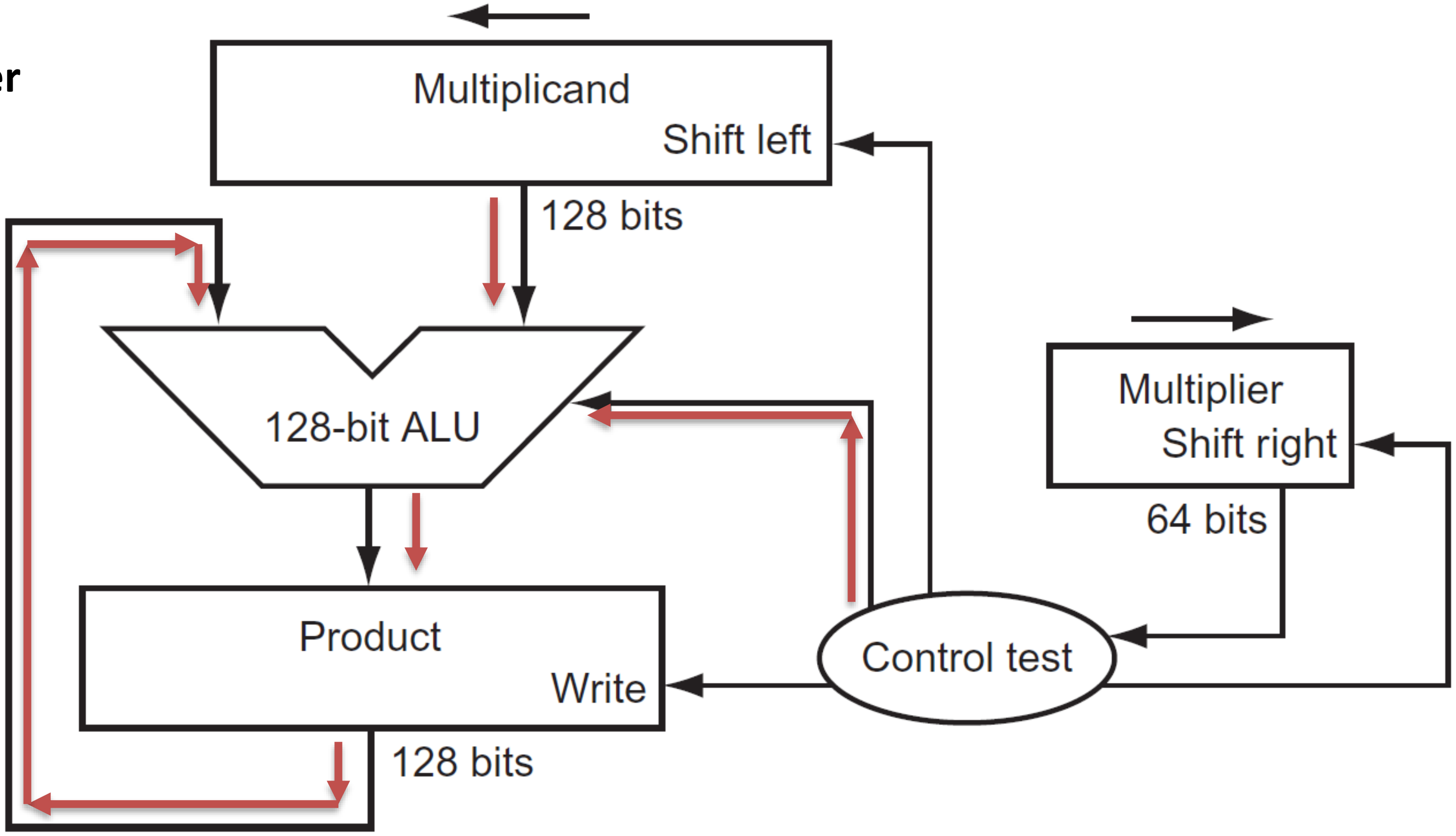
Shift multiplier 1 bit to the right

Repeat 32 times (total)



# Multiplication Hardware

- **Check if multiplier bit is 0/1**
  - **If 1 add multiplicand to product register**
  - Else do nothing
- Shift multiplicand 1 bit to left
- Shift multiplier 1 bit to the right
- Repeat 32 times (total)



# Multiplication Hardware

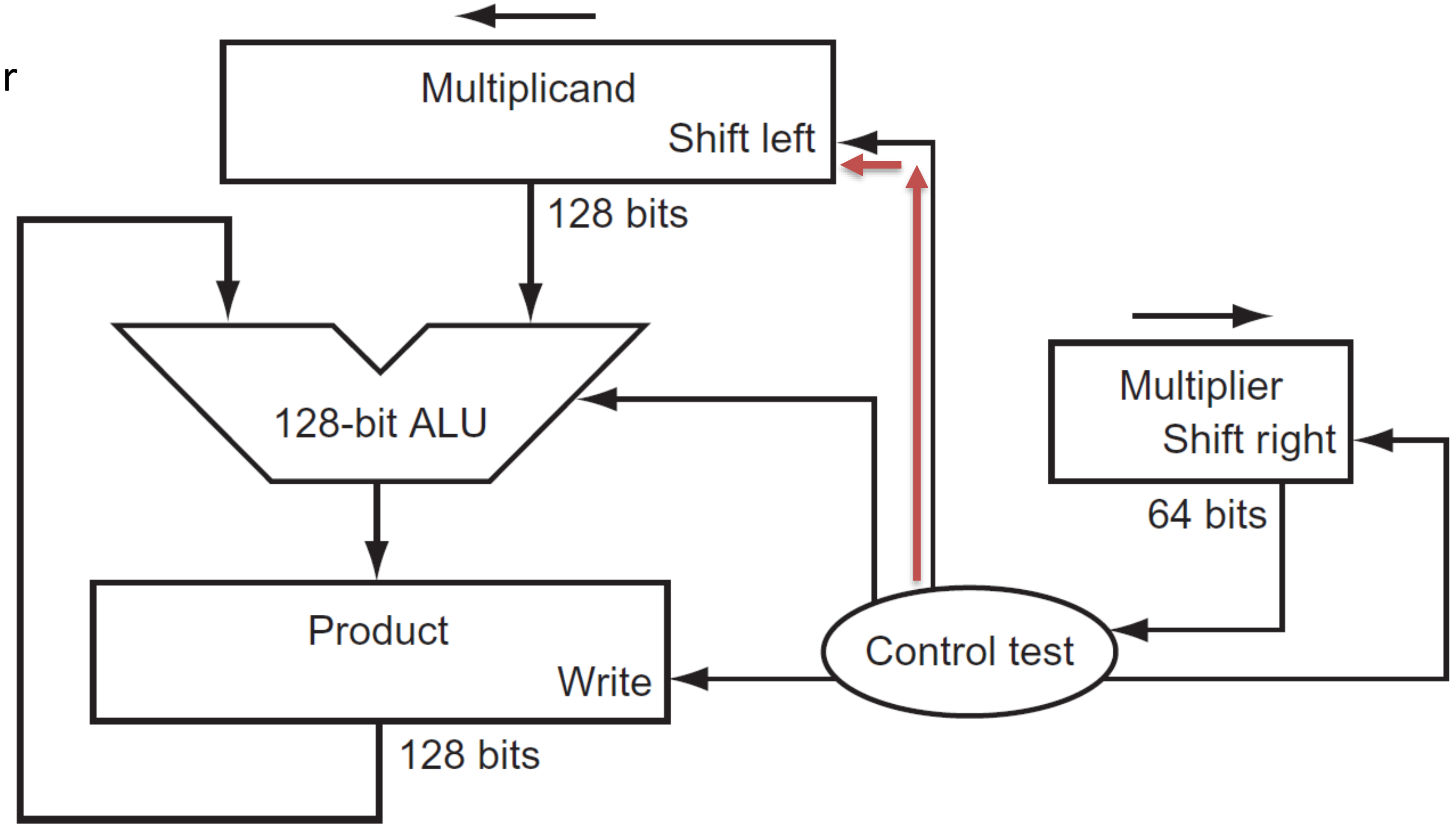
Check if multiplier bit is 0/1

- If 1 add multiplicand to product register
- Else do nothing

**Shift multiplicand 1 bit to left**

Shift multiplier 1 bit to the right

Repeat 32 times (total)



# Multiplication Hardware

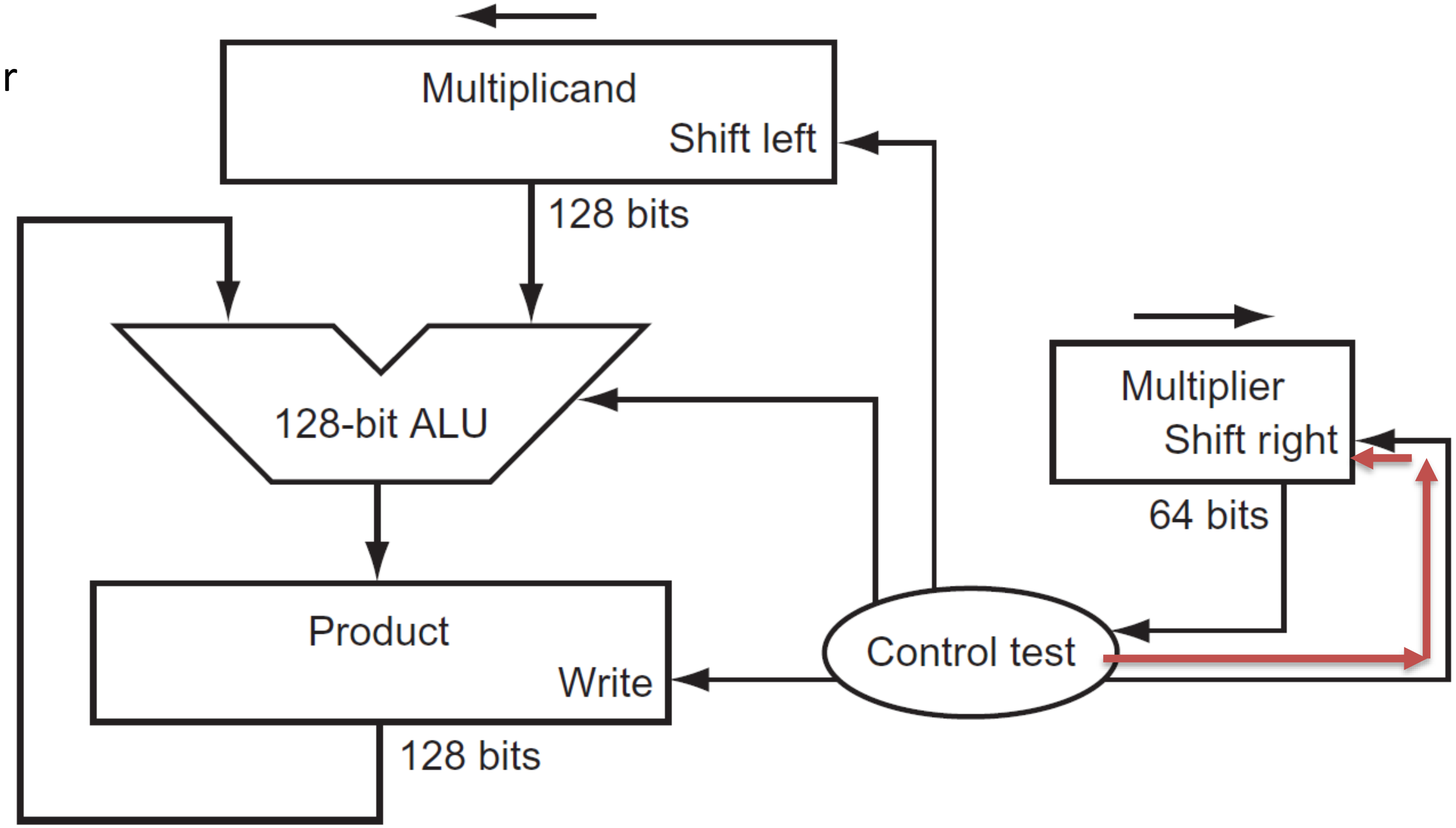
Check if multiplier bit is 0/1

- If 1 add multiplicand to product register
- Else do nothing

Shift multiplicand 1 bit to left

**Shift multiplier 1 bit to the right**

Repeat 32 times (total)



# Signed Multiplication

# Signed Multiplication

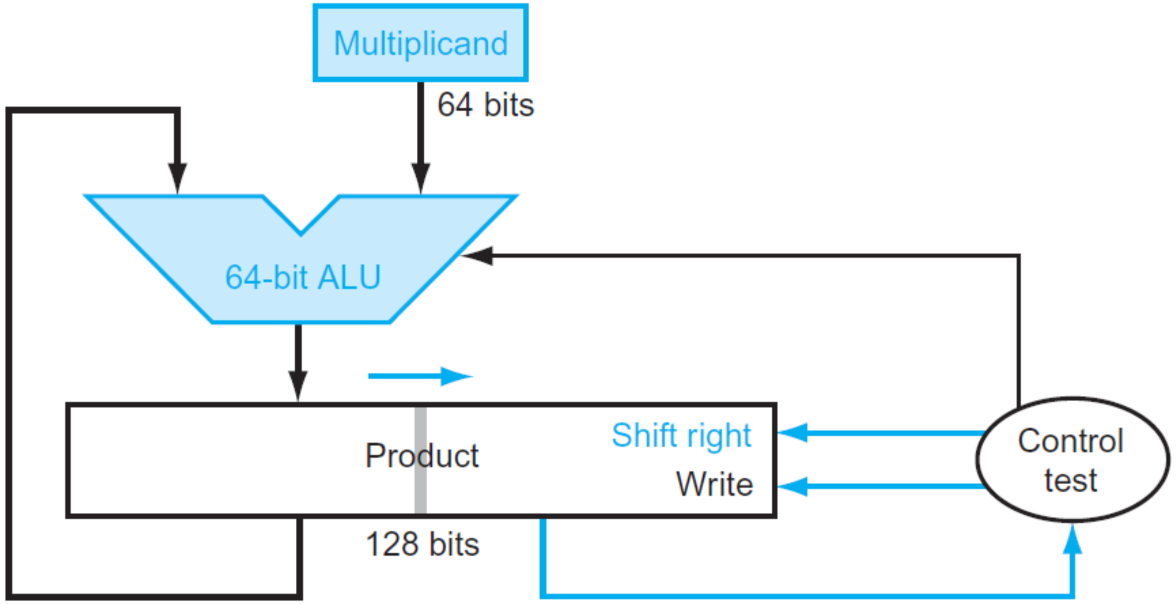
- Convert to Multiplicand and Multiplier to positive and remember the sign

| Multiplicand | Multiplier | Result |
|---|---|---|
| -ve | +ve | -ve |
| +ve | -ve | -ve |
| +ve | +ve | +ve |
| -ve | -ve | +ve |

If multiplicand and multiplier signs disagree, then the result is negative.

# Optimized Multiplier

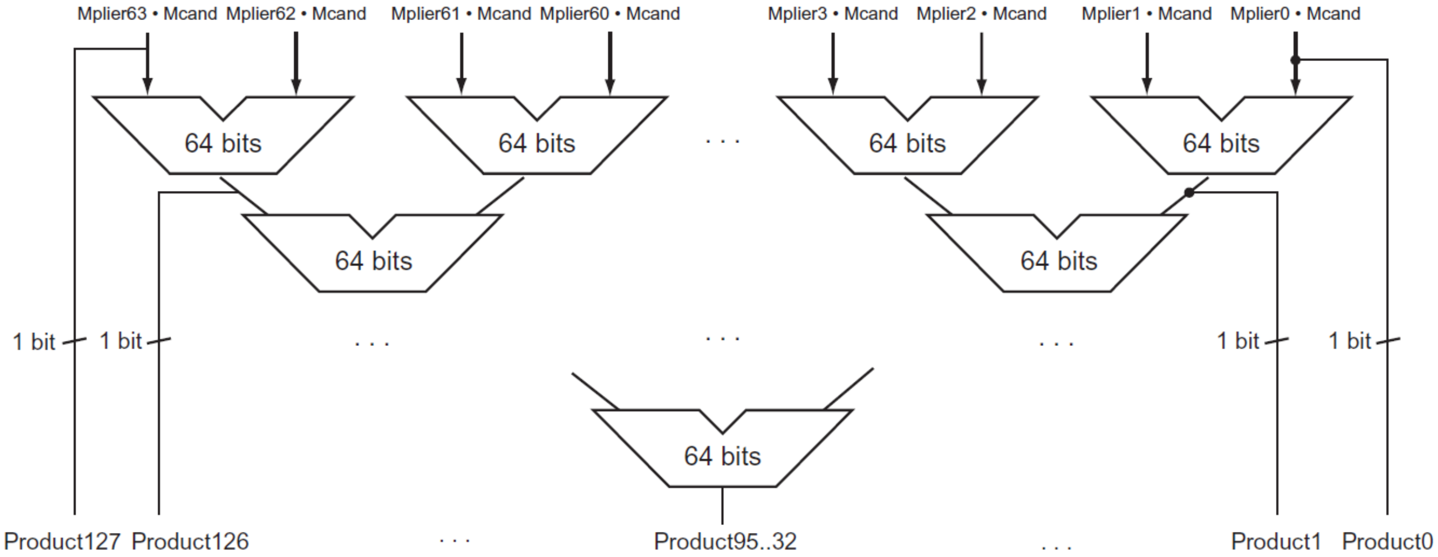
- Perform steps in parallel: add/shift




- One cycle per partial-product addition
  - That's ok, if frequency of multiplications is low

# Faster Multiplier

- Uses multiple adders
  - Cost/performance tradeoff




- Can be pipelined
  - Several multiplication performed in parallel

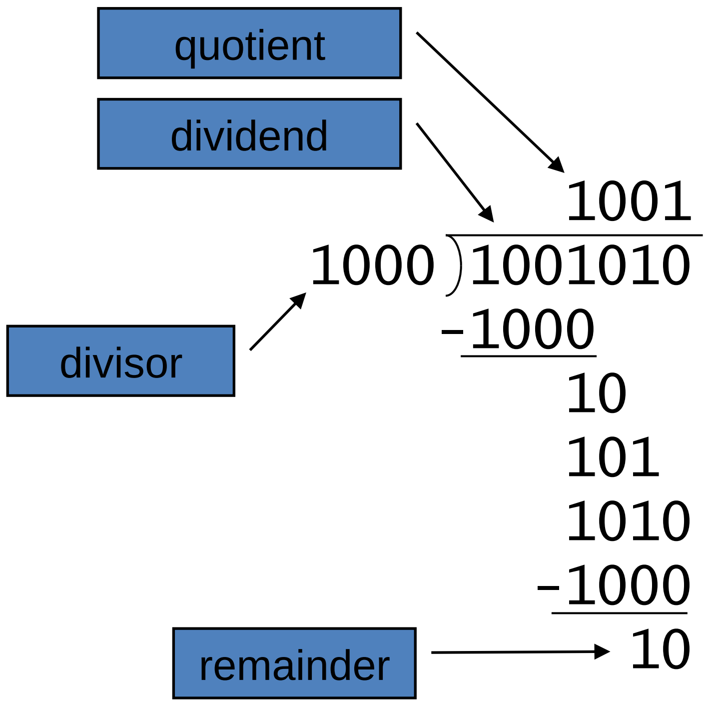
# LEGv8 Multiplication

- Three multiply instructions:
  - MUL: multiply
    - Gives the lower 64 bits of the product
  - SMULH: signed multiply high
    - Gives the upper 64 bits of the product, assuming the operands are signed
  - UMULH: unsigned multiply high
    - Gives the upper 64 bits of the product, assuming the operands are unsigned

# Instructions

| Type | Name |
|---|---|
| Arithmetic | ADD, SUB, MUL |
| Data transfer | LDUR, STUR |
| Arithmetic Immediate | ADDI, SUBI, ORRI, ANDI, EORI, **MUL, SMULH, UMULH** |
| Logical Operations | LSL, LSR, AND, ORR, EOR |
| Branches | B, CBZ, CBNZ, B.Cond |
| Set Condition Flag | ADDS, ADDIS, SUBS, SUBIS, ANDS, ANDIS |

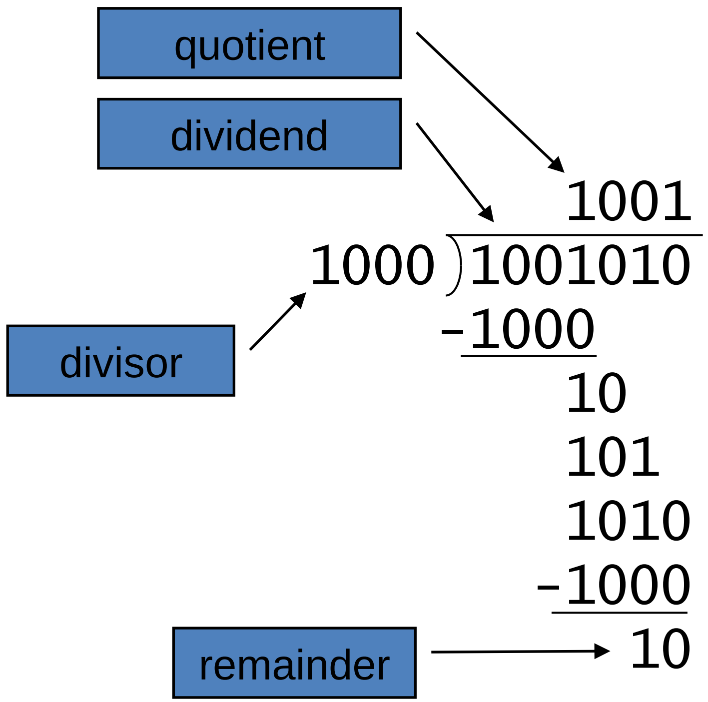
# Division




*n*-bit operands yield *n*-bit quotient and remainder

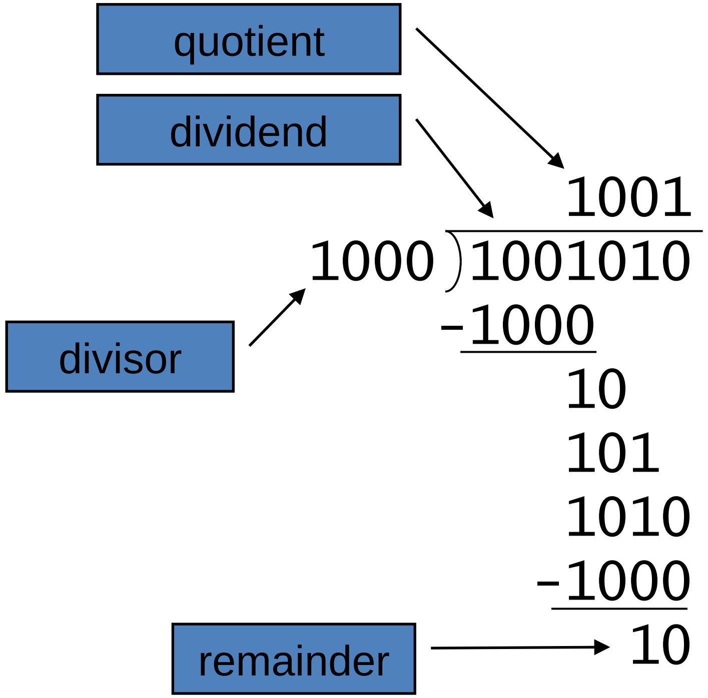
# Division

*1. n*-bit operands yield *n*-bit quotient and remainder



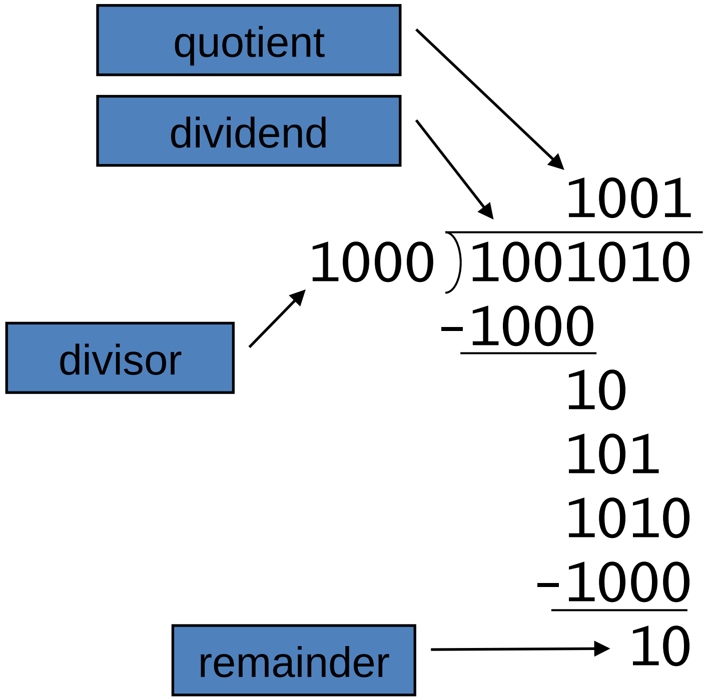
# Division

1. *n*-bit operands yield *n*-bit quotient and remainder
2. Divisor goes in dividend either 0 times or 1 times.



# Division

1. *n*-bit operands yield *n*-bit quotient and remainder
2. Divisor goes in dividend either 0 times or 1 times.
    1. If dividend-divisor ≥ 0, then divisor goes 1 time
    2. Else divisor goes 0 times.




To design the division hardware, lets consider the **dividend** and the **quotient** to be **8 bits long, the divisor is 4 bits long.**

# Division Hardware

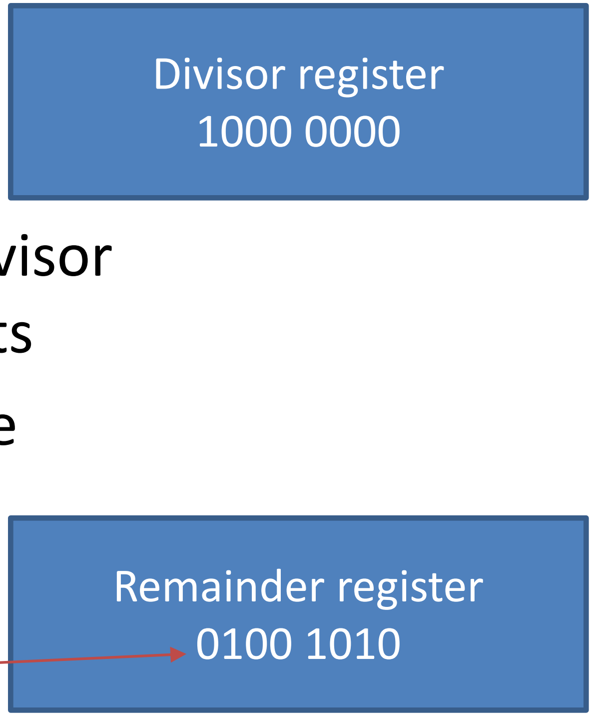
- Dividend: 1001010
- Divisor: 1000
- Initialize Divisor register, with the divisor value in the left most significant bits

Divisor register
1000 0000

```
         1001
1000 )1001010
     -1000
         10
         101
         1010
        -1000
           10
```

# Division Hardware

- Dividend: 1001010
- Divisor: 1000
- Initialize Divisor register, with the divisor value in the right most significant bits
- Initialize remainder register, with the value of the dividend




```
          1001
1000 ) 1001010
      -1000
          10
          101
          1010
         -1000
            10
```

# Division Hardware

- Dividend: 1001010
- Divisor: 1000
- Initialize Divisor register, with the divisor value in the right most significant bits
- Initialize remainder register, with the value of the dividend
- Initialize Quotient register, 4 bits, with zeros

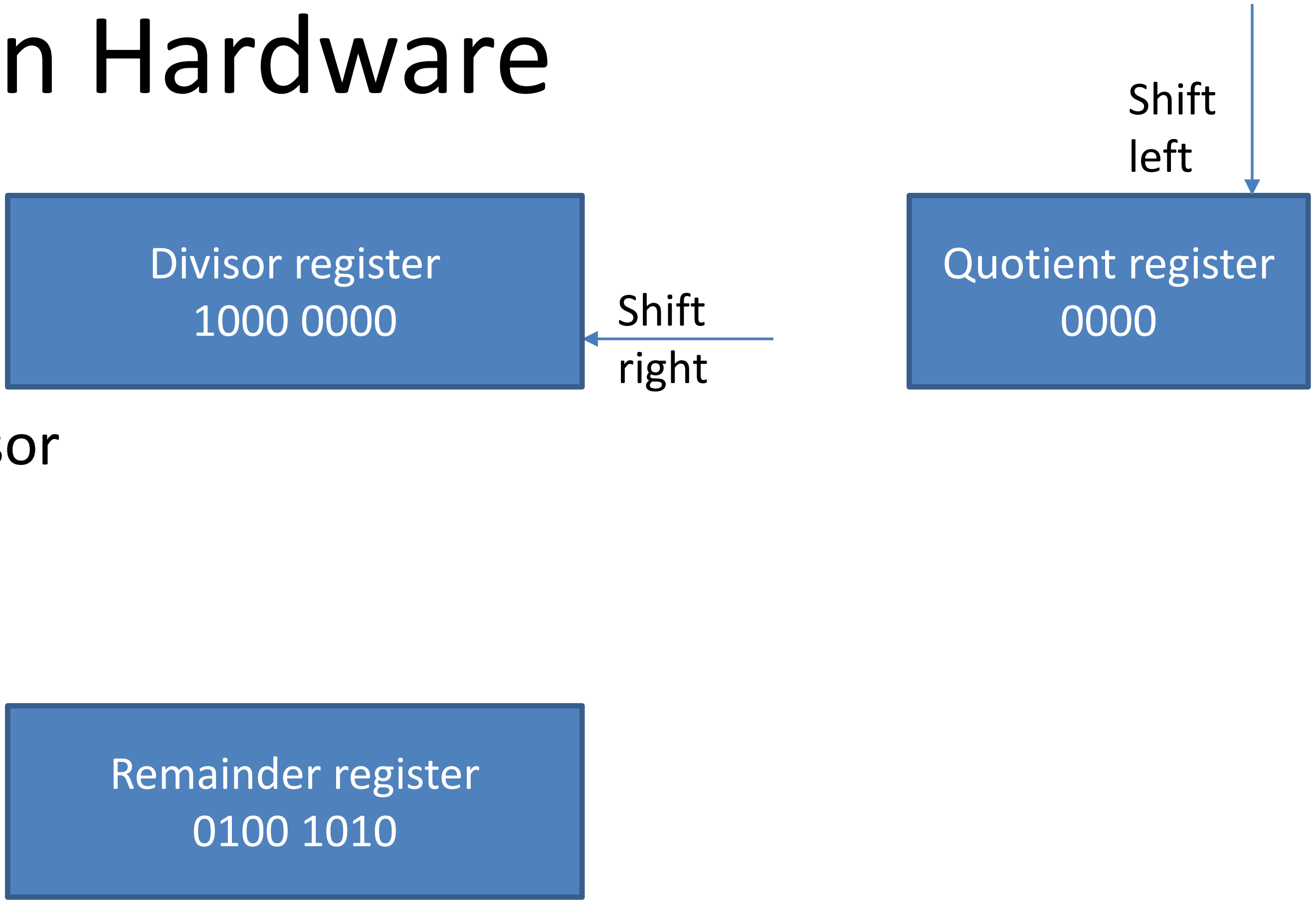
Divisor register
1000 0000

Quotient register
0000

Remainder register
0100 1010

# Division Hardware

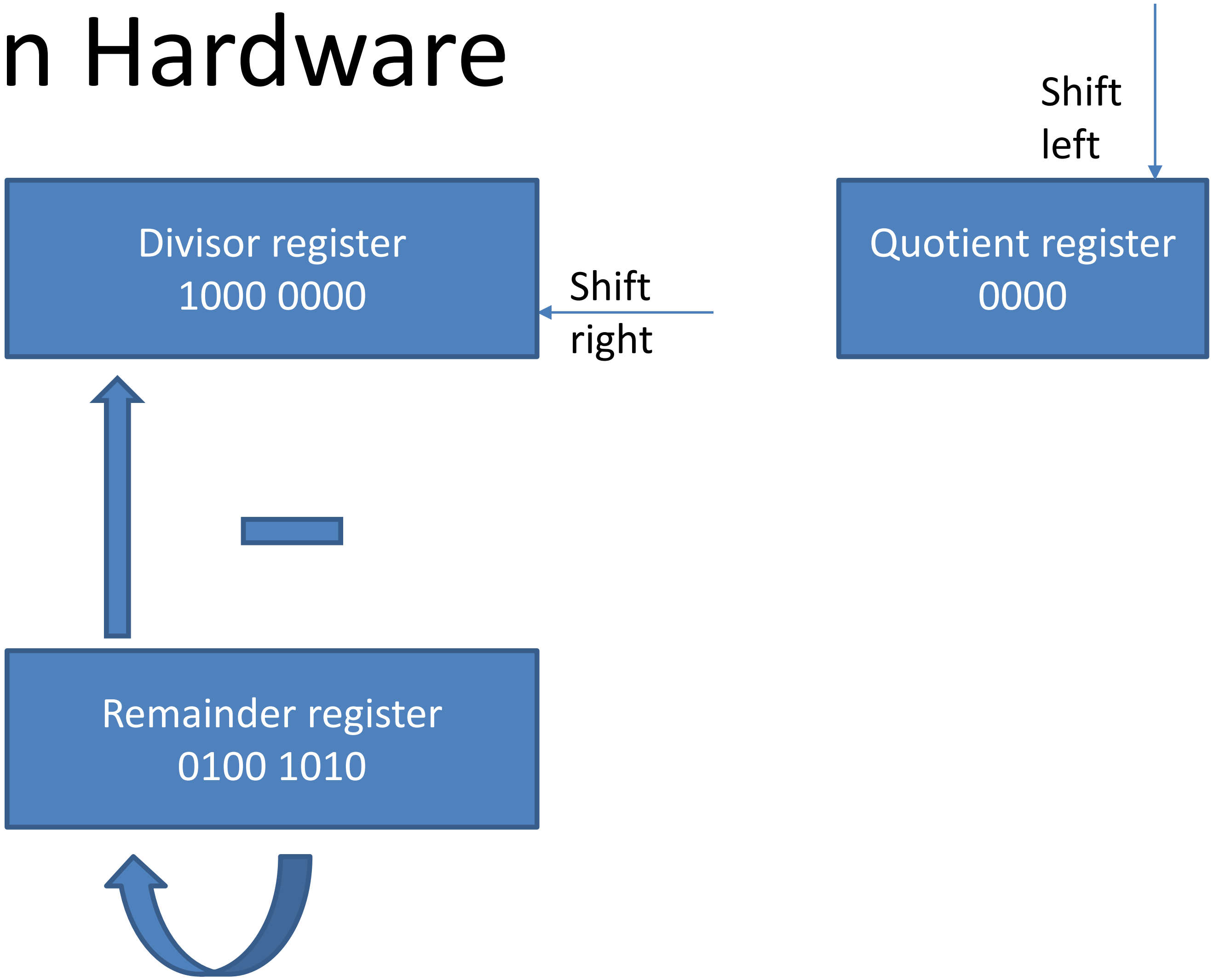
- Dividend: 1001010
- Divisor: 1000
- Initialize Divisor register, with the divisor value in the right most significant bits
  - HW to shift right
- Initialize remainder register, with the value of the dividend
- Initialize Quotient register, 4 bits, with zeros
  - Shift left



# Division Hardware

Iteration 1

1. Remainder = Remainder (Dividend) - Divisor
    1. If remainder < 0

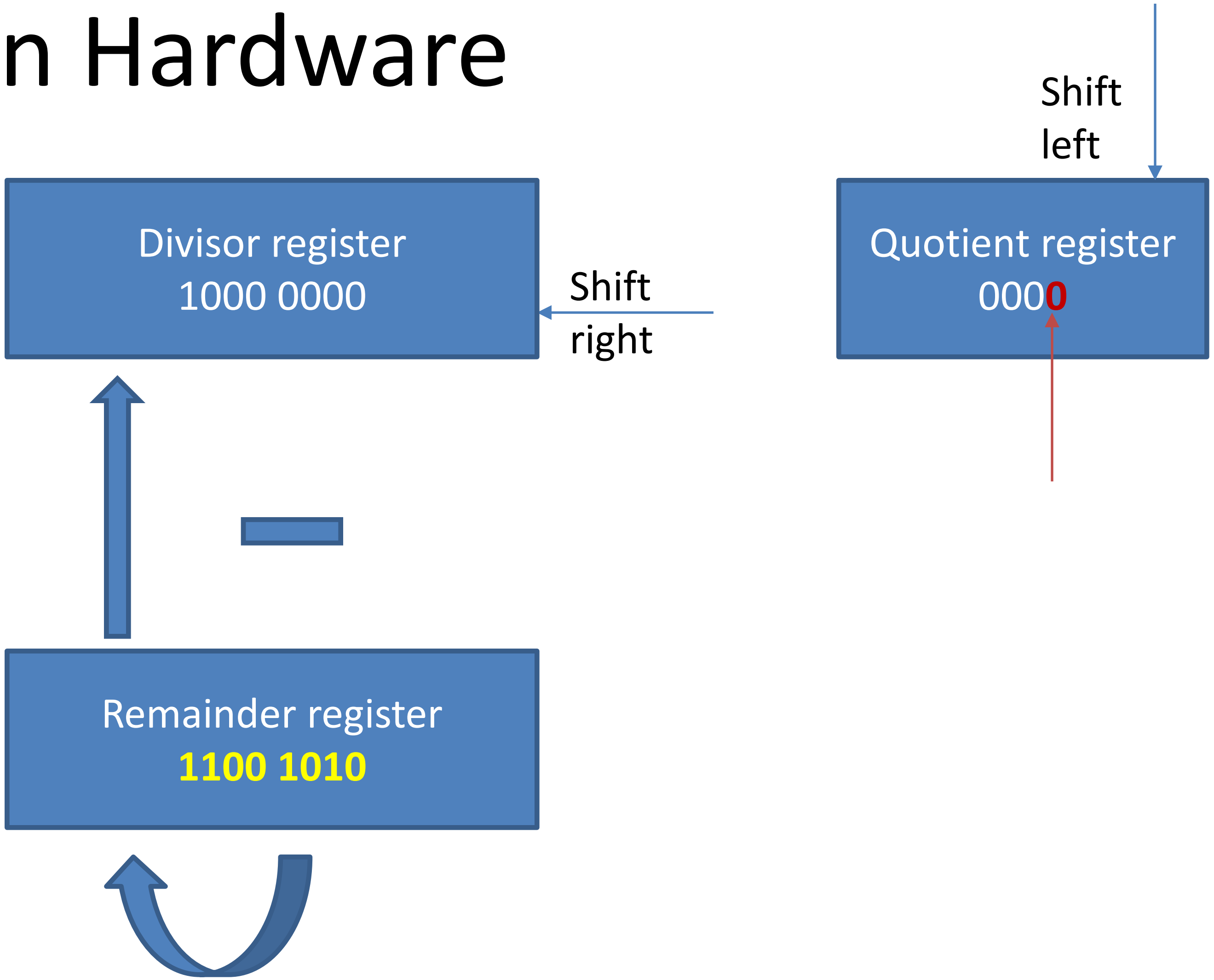



```
          1001
1000 )1001010
     -1000
         10
         101
         1010
        -1000
           10
```

# Division Hardware

Iteration 1

1. Remainder = Remainder - Divisor
   1. If remainder < 0,
      1. Shift quotient to left, and add 0 to end

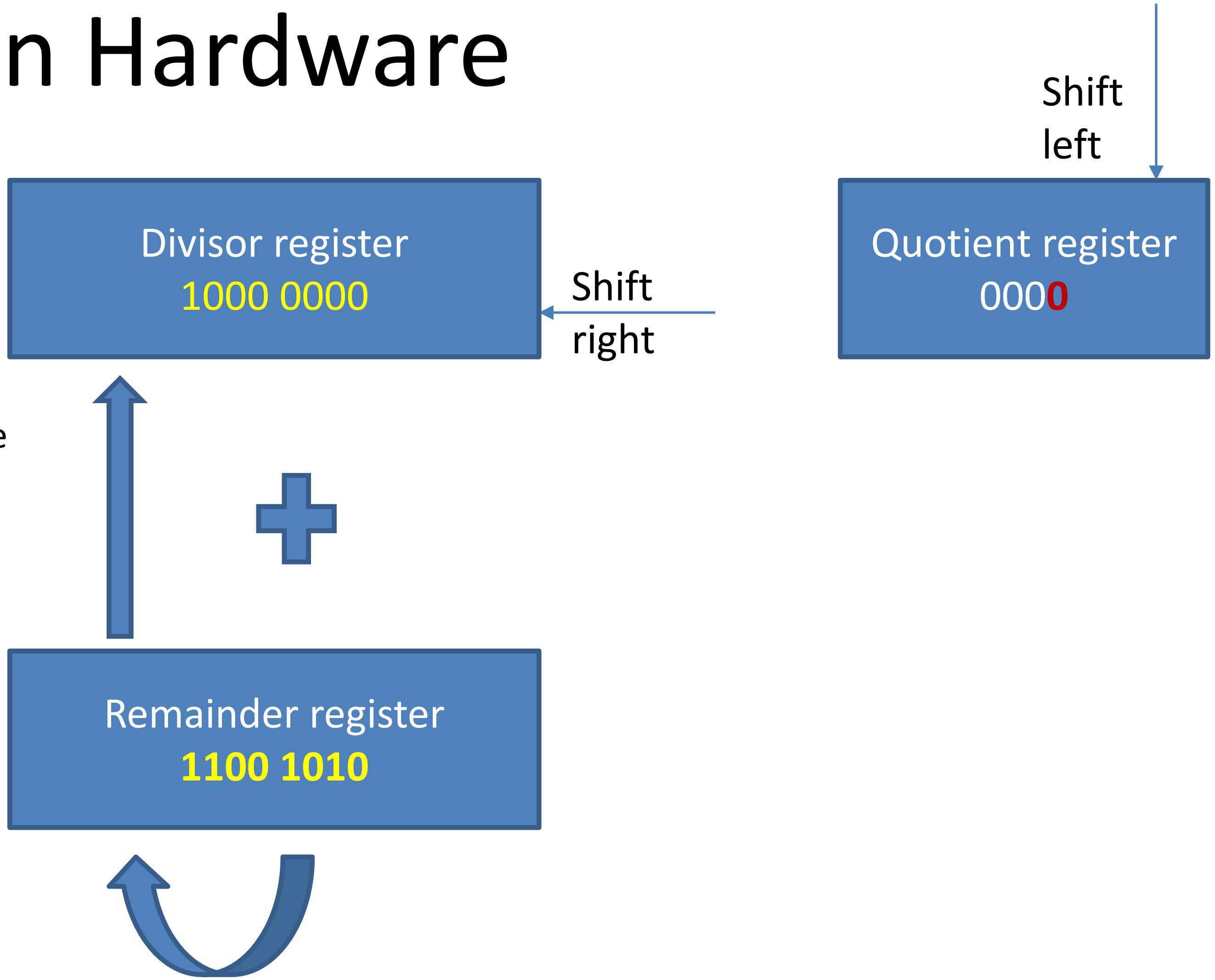



```
          1001
 1000 ) 1001010
       -1000
          10
          101
          1010
         -1000
            10
```

# Division Hardware

Iteration 1

1. Remainder = Remainder - Divisor
   1. If remainder < 0,
      1. Shift quotient to left, and add 0 to end
      2. Add the remainder back to divisor, and restore value



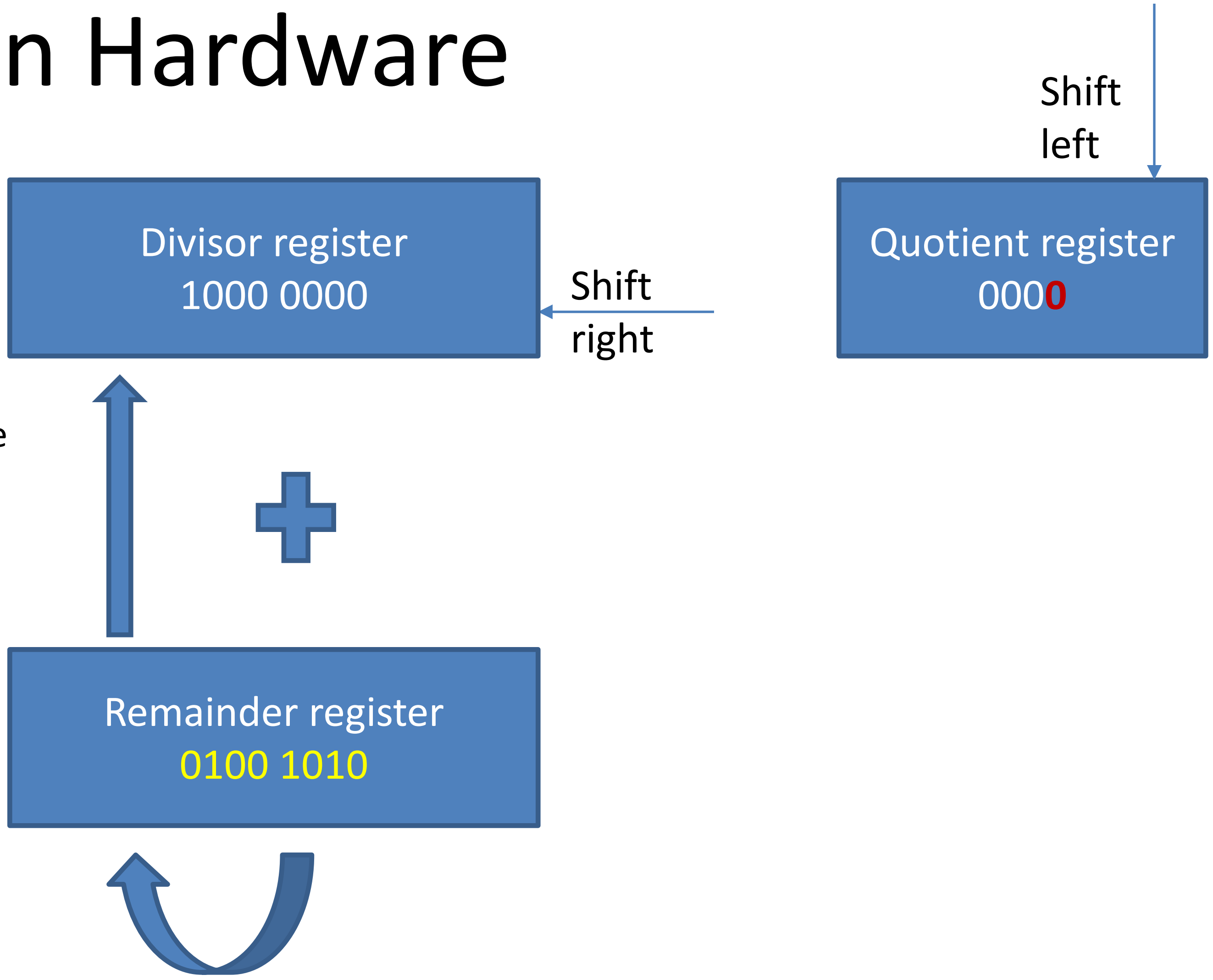

```
          1001
1000 ) 1001010
      -1000
          10
          101
          1010
         -1000
            10
```

# Division Hardware

Iteration 1

1. Remainder = Remainder - Divisor
    1. If remainder < 0,
        1. Shift quotient to left, and add 0 to end
        2. Add the remainder back to divisor, and restore value
2. Shift Divisor to the right by 1 bit

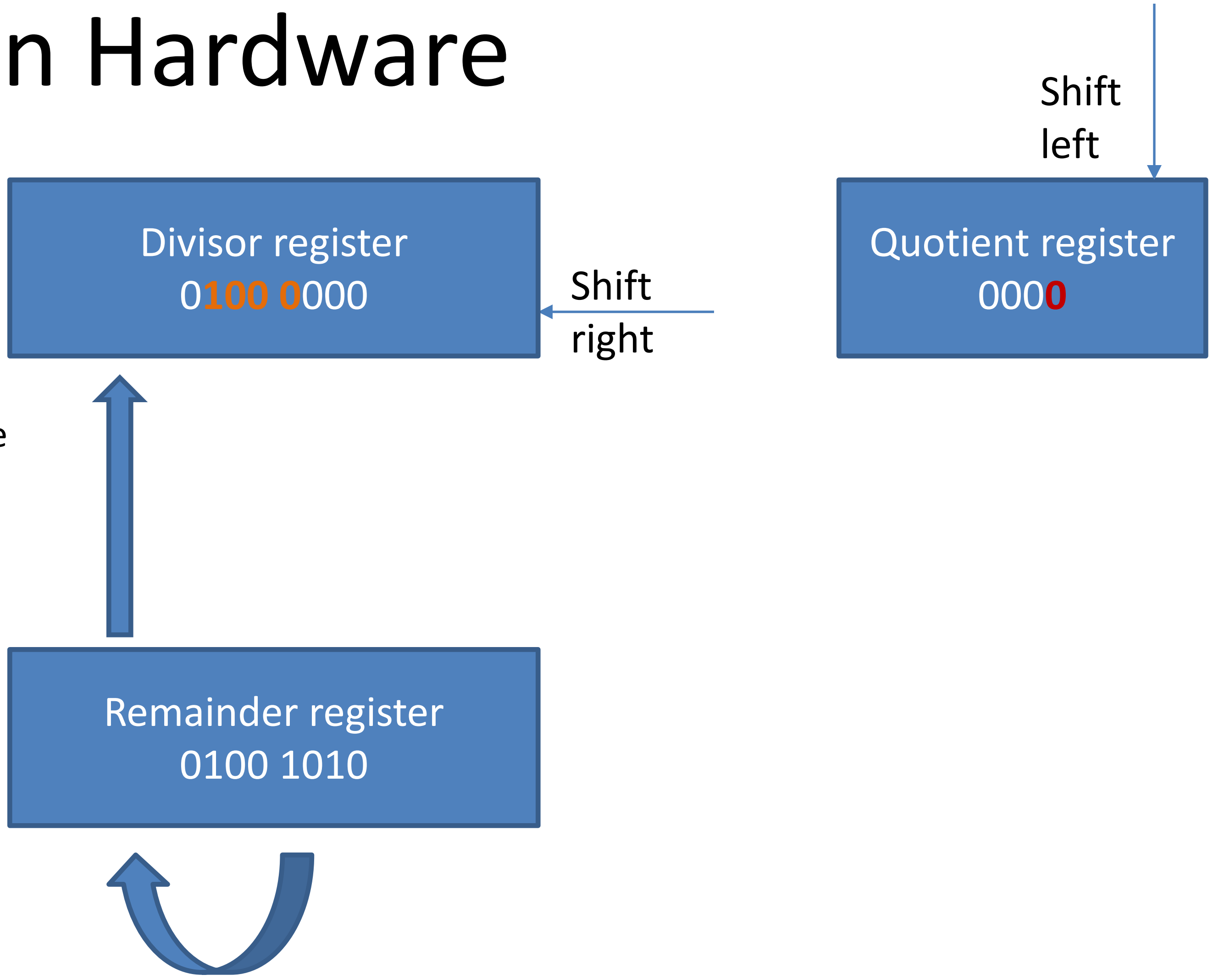
```
          1001
1000 ) 1001010
      -1000
          10
          101
          1010
         -1000
            10
```



# Division Hardware

Iteration 1

1. Remainder = Remainder - Divisor
    1. If remainder < 0,
        1. Shift quotient to left, and add 0 to end
        2. Add the remainder back to divisor, and restore value
2. Shift Divisor to the right by 1 bit



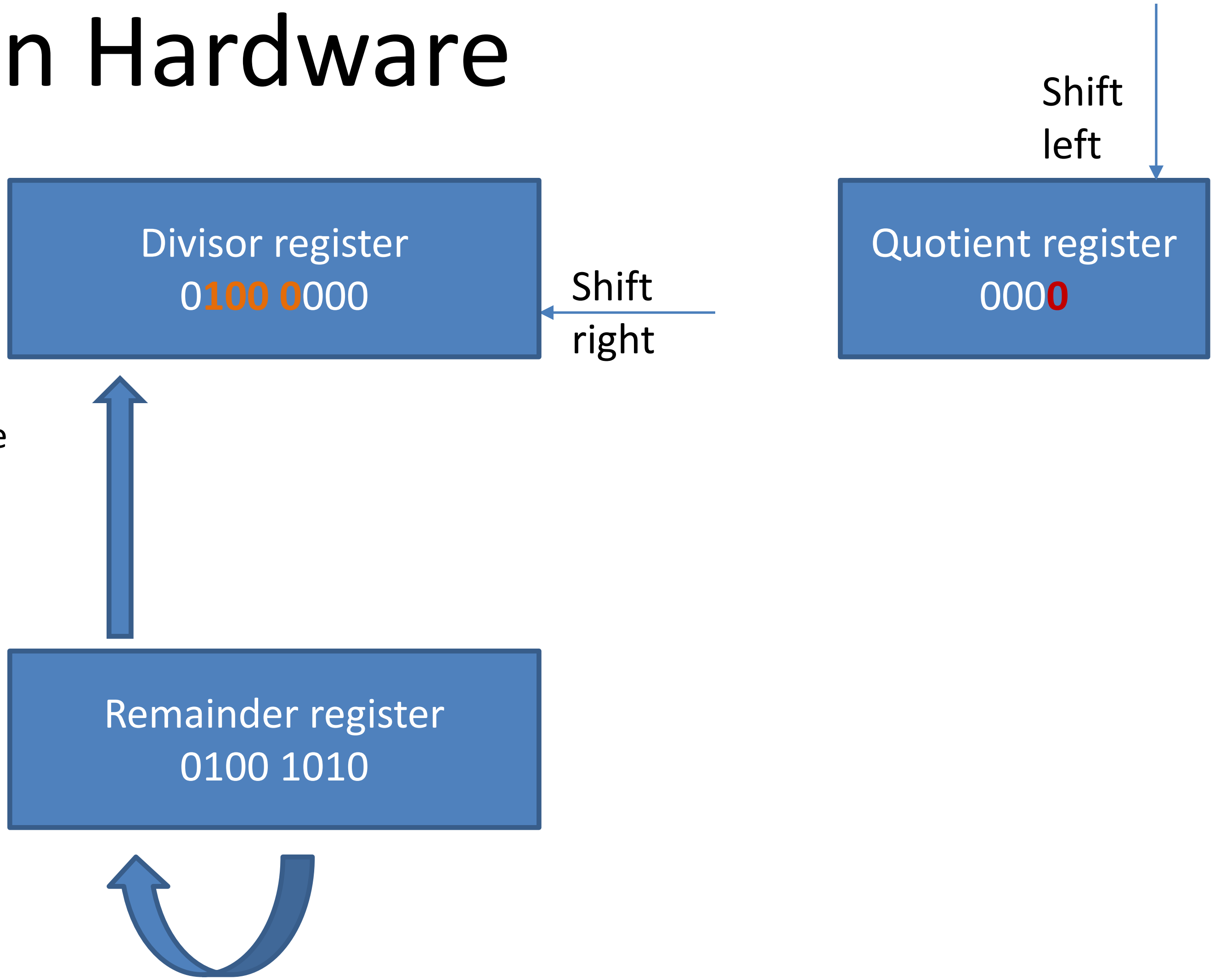

```
         1001
1000 ) 1001010
      -1000
          10
          101
          1010
         -1000
            10
```

# Division Hardware

Iteration 1

1. Remainder = Remainder - Divisor
   1. If remainder < 0,
      1. Shift quotient to left, and add 0 to end
      2. Add the remainder back to divisor, and restore value
2. Shift Divisor to the right by 1 bit
3. Repeat 5 times total

```
           1001
1000 ) 1001010
      -1000
           10
           101
           1010
          -1000
             10
```

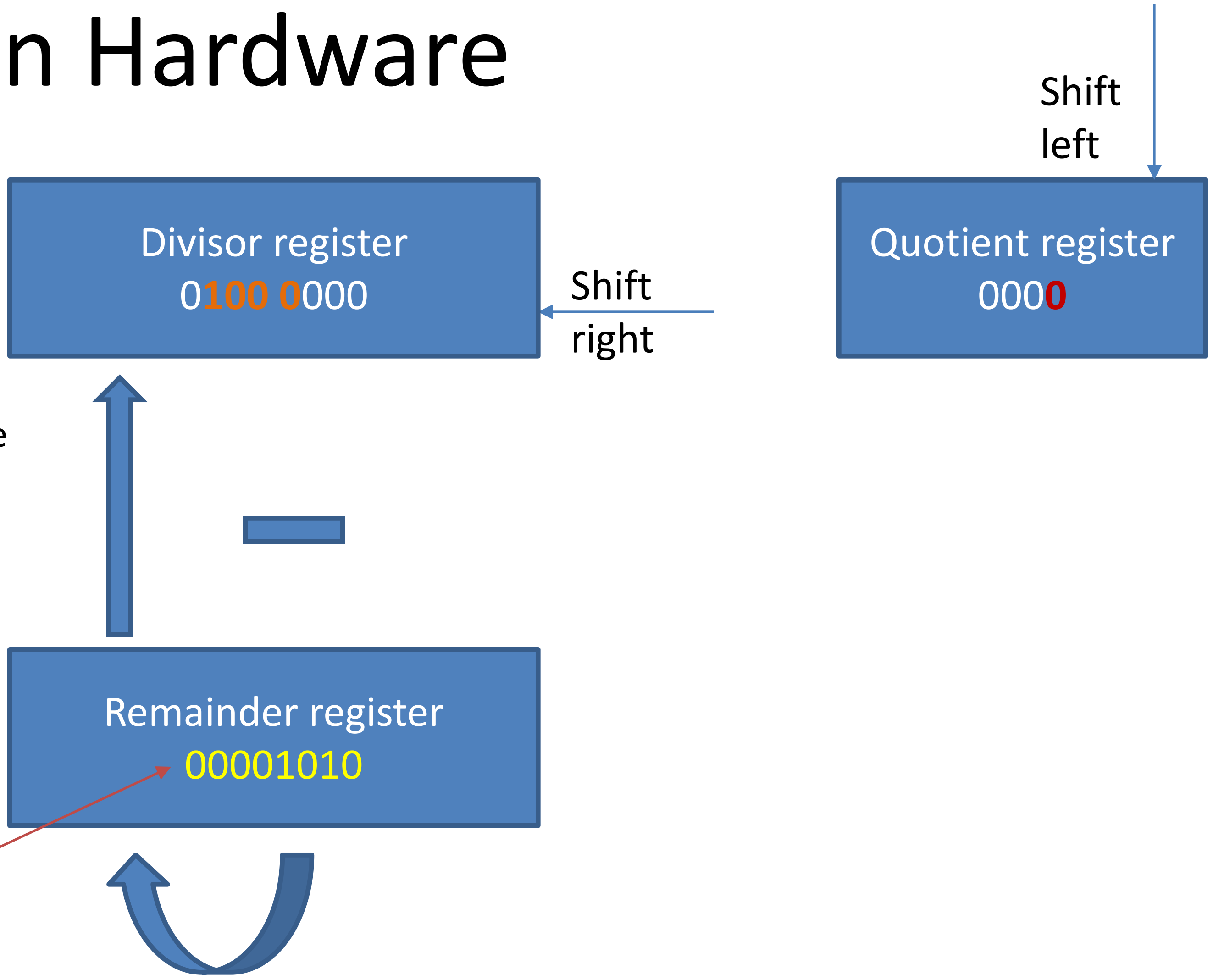


# Division Hardware

Iteration 2

1. **Remainder = Remainder - Divisor**
    1. If remainder < 0,
        1. Shift quotient to left, and add 0 to end
        2. Add the remainder back to divisor, and restore value
2. Shift Divisor to the right by 1 bit
3. Repeat 5 times total

```
           1001
1000 ) 1001010
      -1000
         10
         101
         1010
        -1000
           10
```

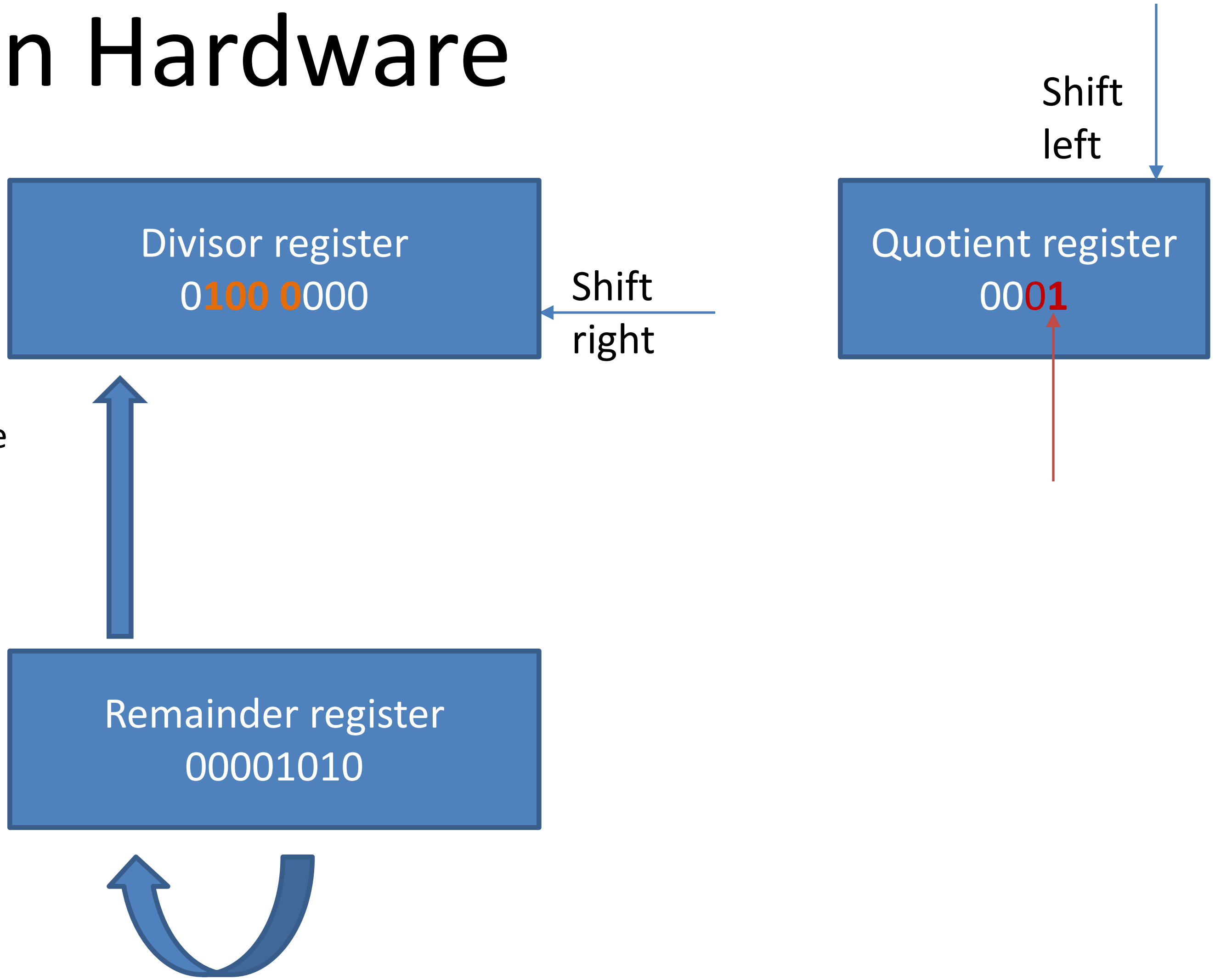


# Division Hardware

Iteration 2

1. Remainder = Remainder - Divisor
    1. If remainder < 0,
        1. Shift quotient to left, and add 0 to end
        2. Add the remainder back to divisor, and restore value
    2. **If remainder > 0,**
        1. **Shift quotient to left, and add 1 to end**
2. Shift Divisor to the right by 1 bit
3. Repeat 5 times

```
          1001
1000 )1001010
      -1000
          10
          101
          1010
         -1000
            10
```

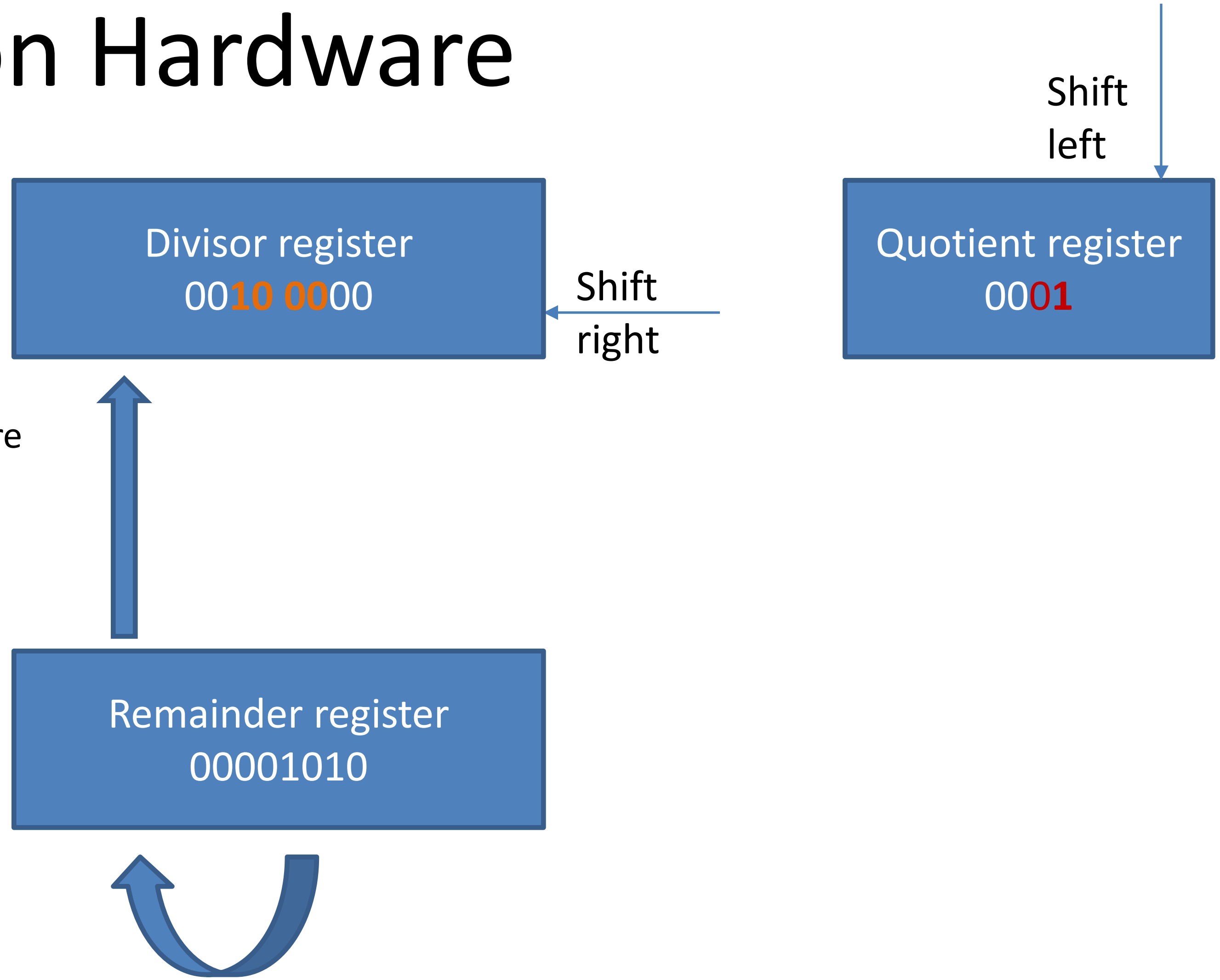


# Division Hardware

Iteration 2

1. Remainder = Remainder - Divisor
   1. If remainder < 0,
      1. Shift quotient to left, and add 0 to end
      2. Add the remainder back to devisor, and restore value
   2. If remainder > 0,
      1. Shift quotient to left, and add 1 to end
2. **Shift Divisor to the right by 1 bit**
3. Repeat 5 times

```
         1001
1000 )1001010
     -1000
         10
         101
         1010
        -1000
           10
```

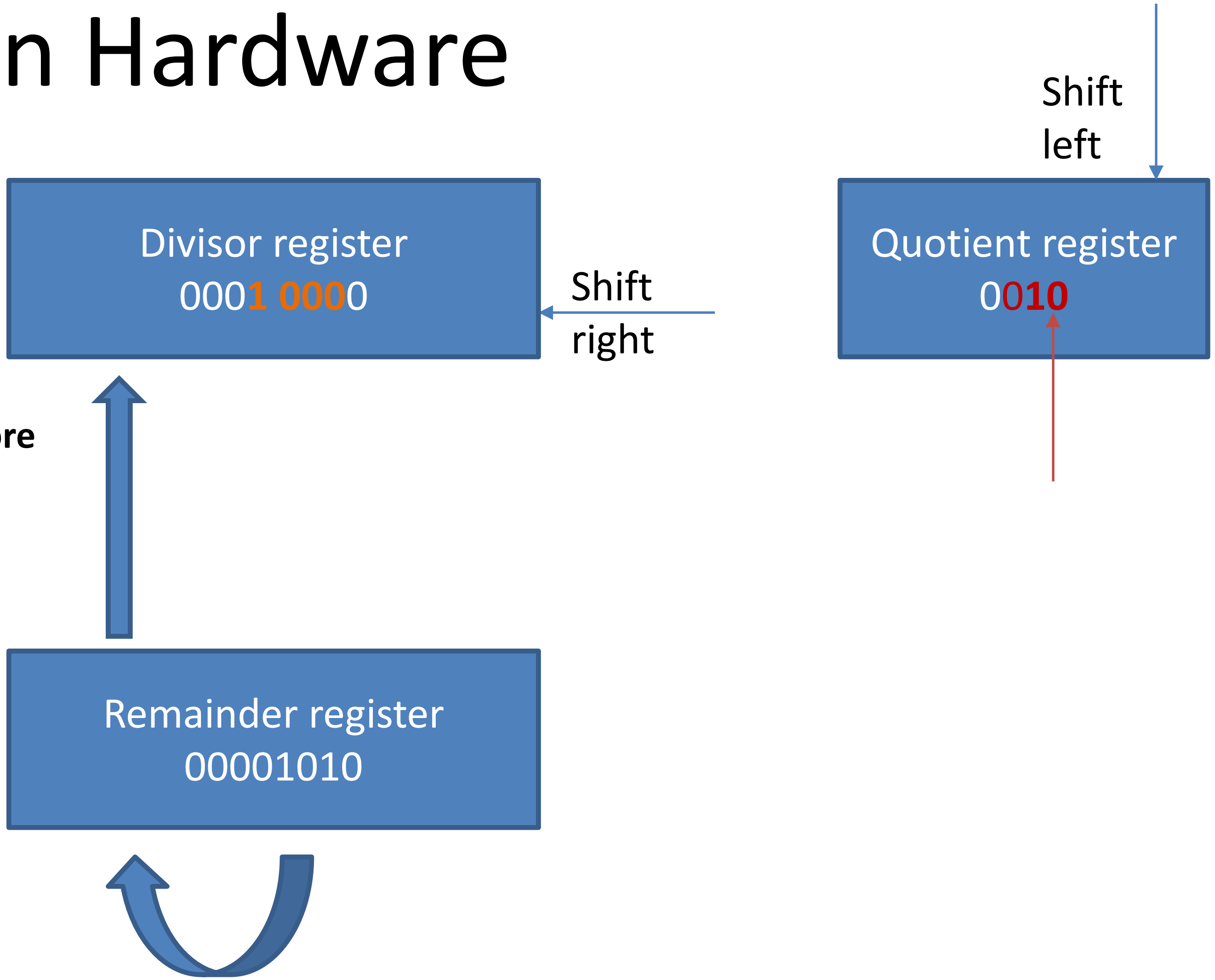


# Division Hardware

Iteration 3

1. **Remainder = Remainder - Divisor**
   1. **If remainder < 0,**
      1. **Shift quotient to left, and add 0 to end**
      2. **Add the remainder back to devisor, and restore value**
   2. If remainder > 0,
      1. Shift quotient to left, and add 1 to end
2. **Shift Divisor to the right by 1 bit**
3. Repeat 5 times

```
        1001
1000 )1001010
     -1000
         10
         101
         1010
        -1000
           10
```

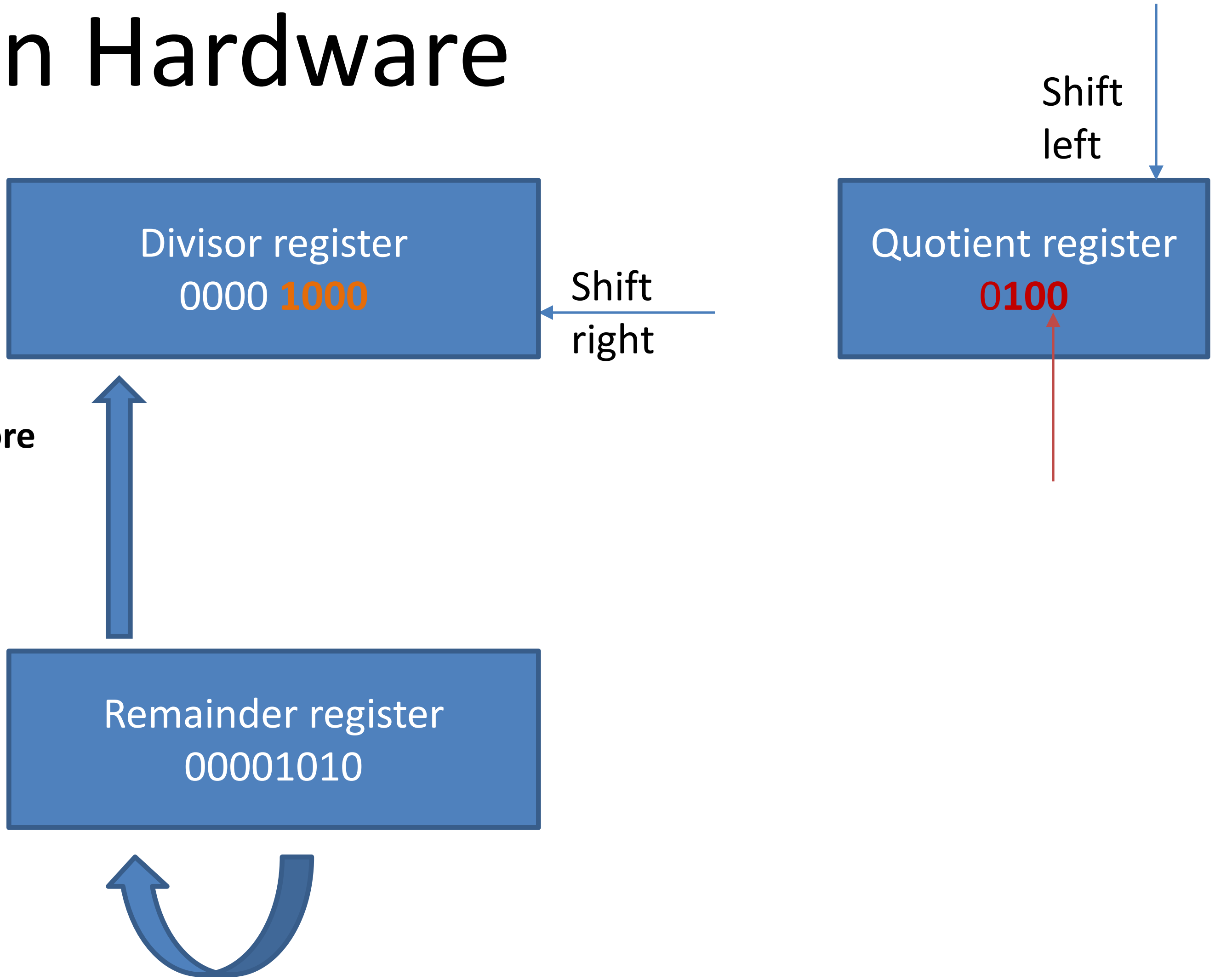


# Division Hardware

Iteration 4

1. **Remainder = Remainder - Divisor**
   1. **If remainder < 0,**
      1. **Shift quotient to left, and add 0 to end**
      2. **Add the remainder back to devisor, and restore value**
   2. If remainder > 0,
      1. Shift quotient to left, and add 1 to end
2. **Shift Divisor to the right by 1 bit**
3. Repeat 5 times

```
        1001
1000 )1001010
     -1000
         10
         101
         1010
        -1000
           10
```

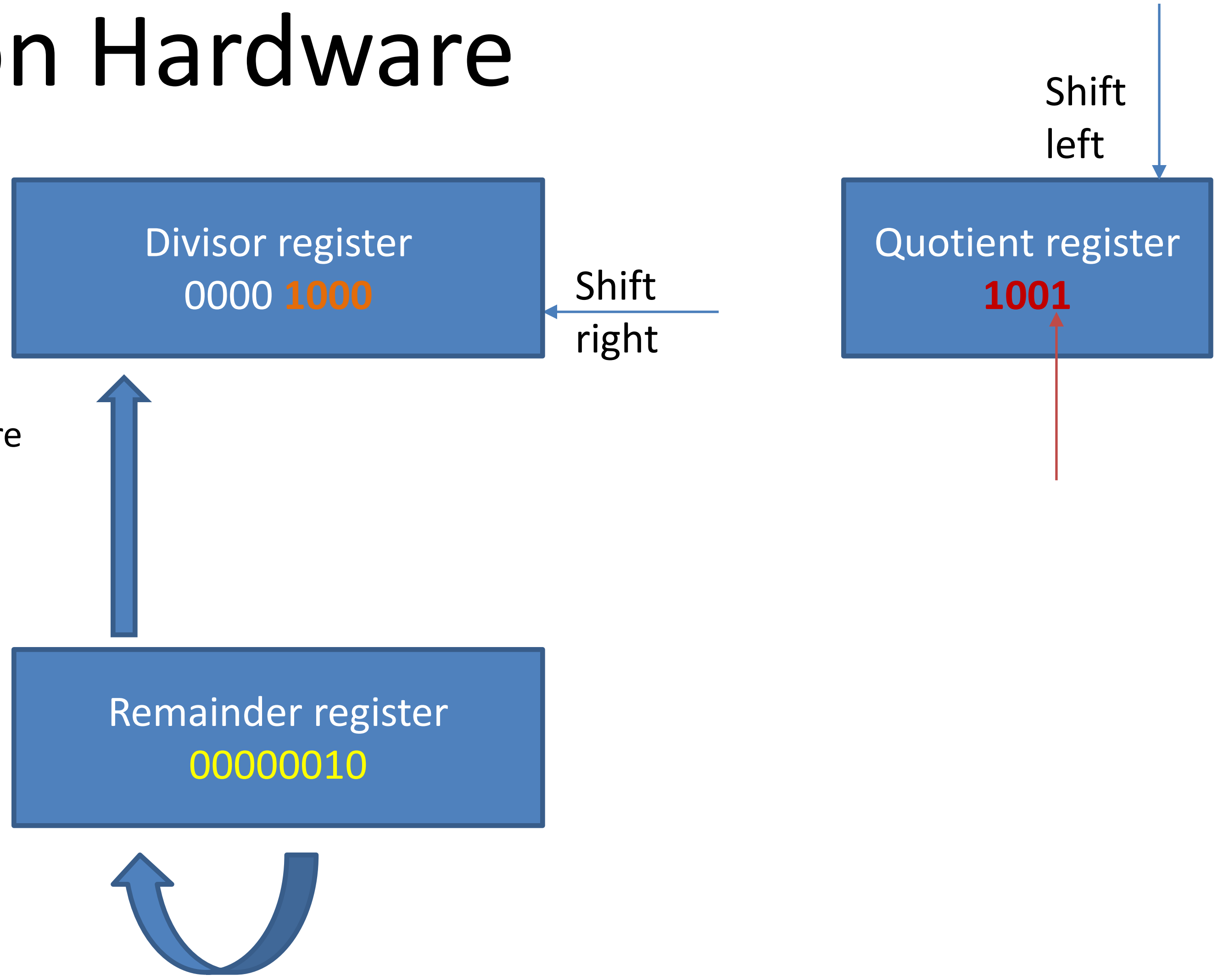


# Division Hardware

Iteration 5

1. **Remainder = Remainder - Divisor**
   1. If remainder < 0,
      1. Shift quotient to left, and add 0 to end
      2. Add the remainder back to devisor, and restore value
   2. **If remainder > 0,**
      1. **Shift quotient to left, and add 1 to end**
2. **Shift Divisor to the right by 1 bit**
3. Repeat 5 times

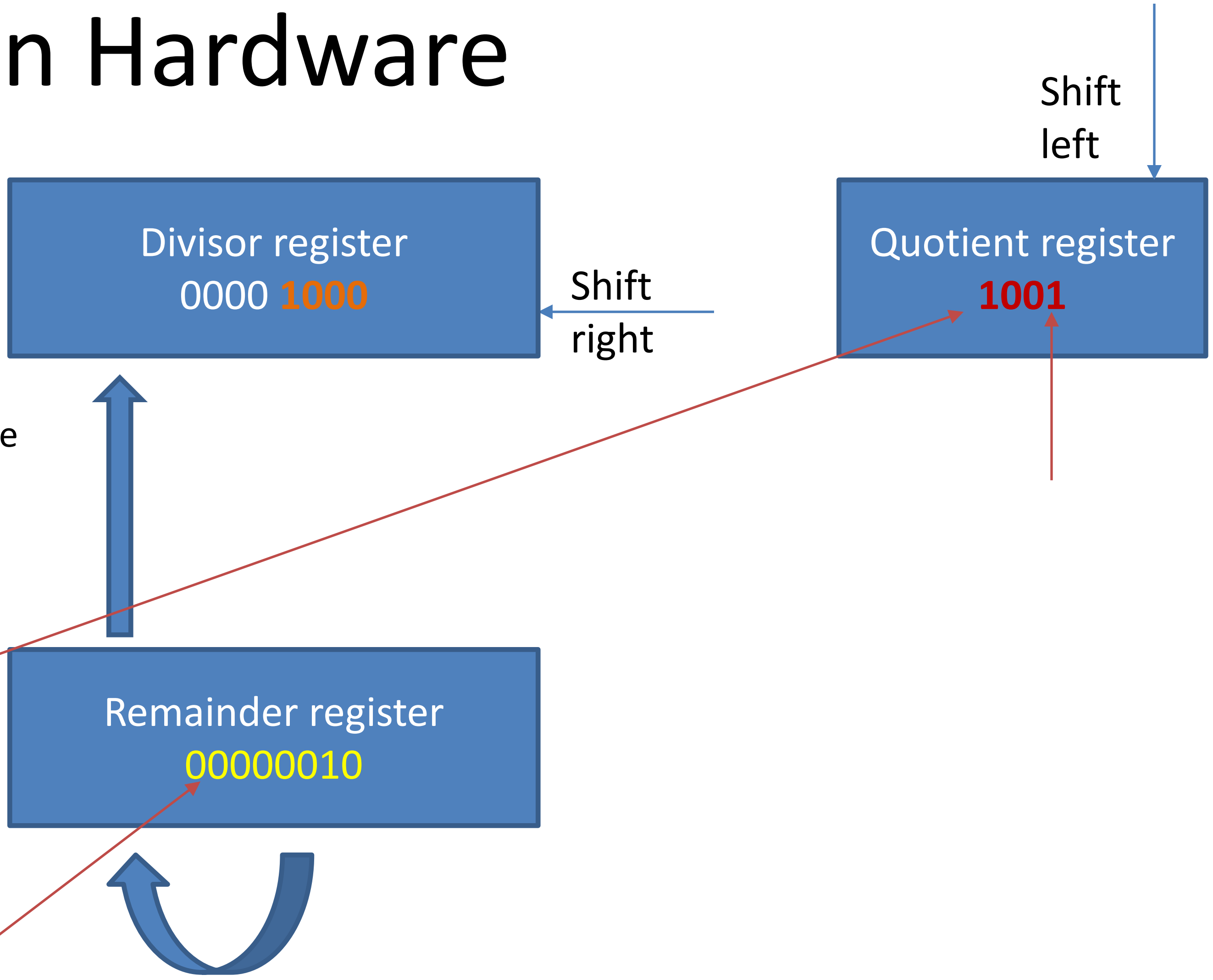
```
        1001
1000 )1001010
     -1000
         10
         101
         1010
        -1000
           10
```



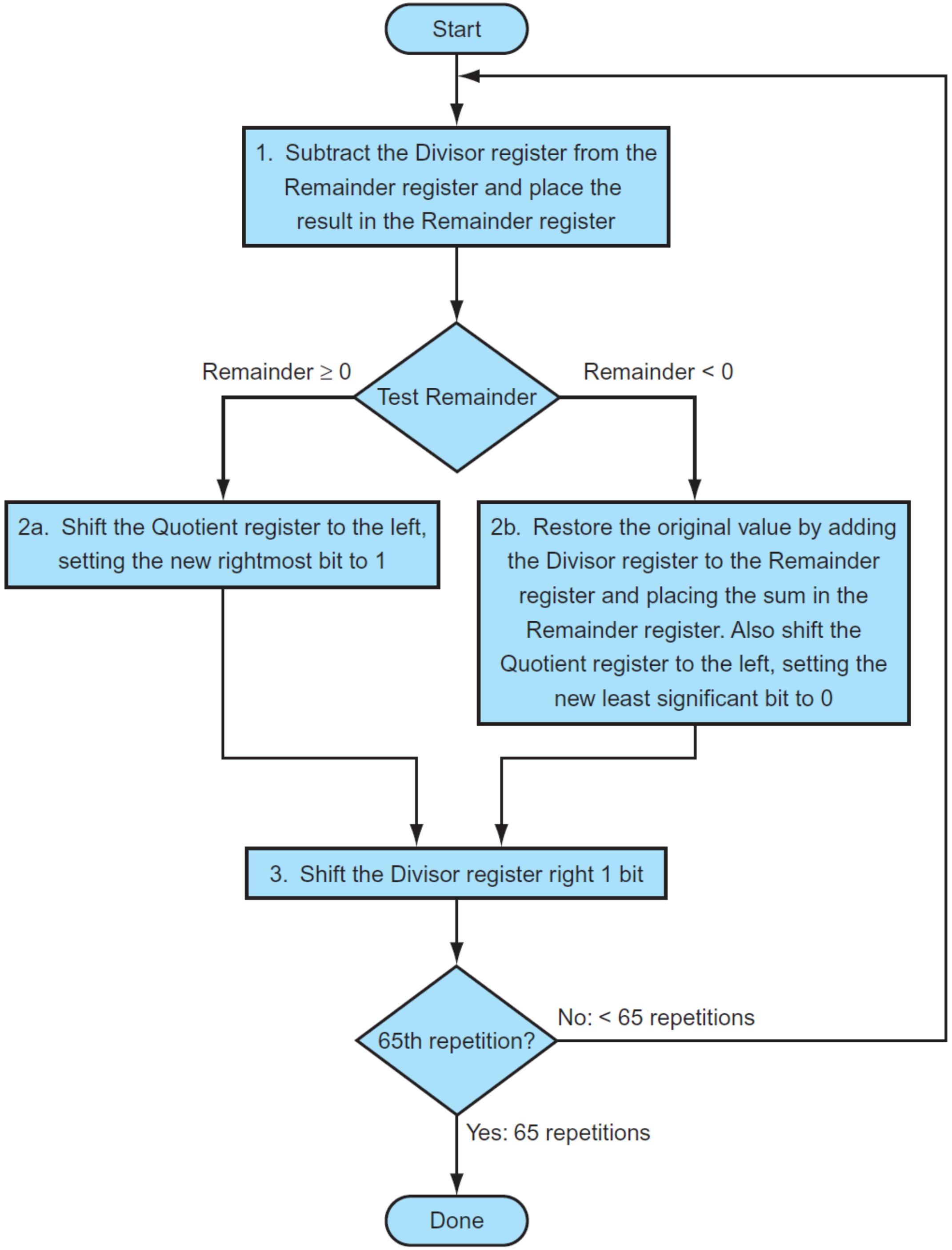
# Division Hardware

1. **Remainder = Remainder - Divisor**
   1. If remainder < 0,
      1. Shift quotient to left, and add 0 to end
      2. Add the remainder back to devisor, and restore value
   2. **If remainder > 0,**
      1. **Shift quotient to left, and add 1 to end**
2. **Shift Divisor to the right by 1 bit**
3. Repeat 5 times

```
          1001
1000 )1001010
       -1000
           10
           101
           1010
          -1000
             10
```

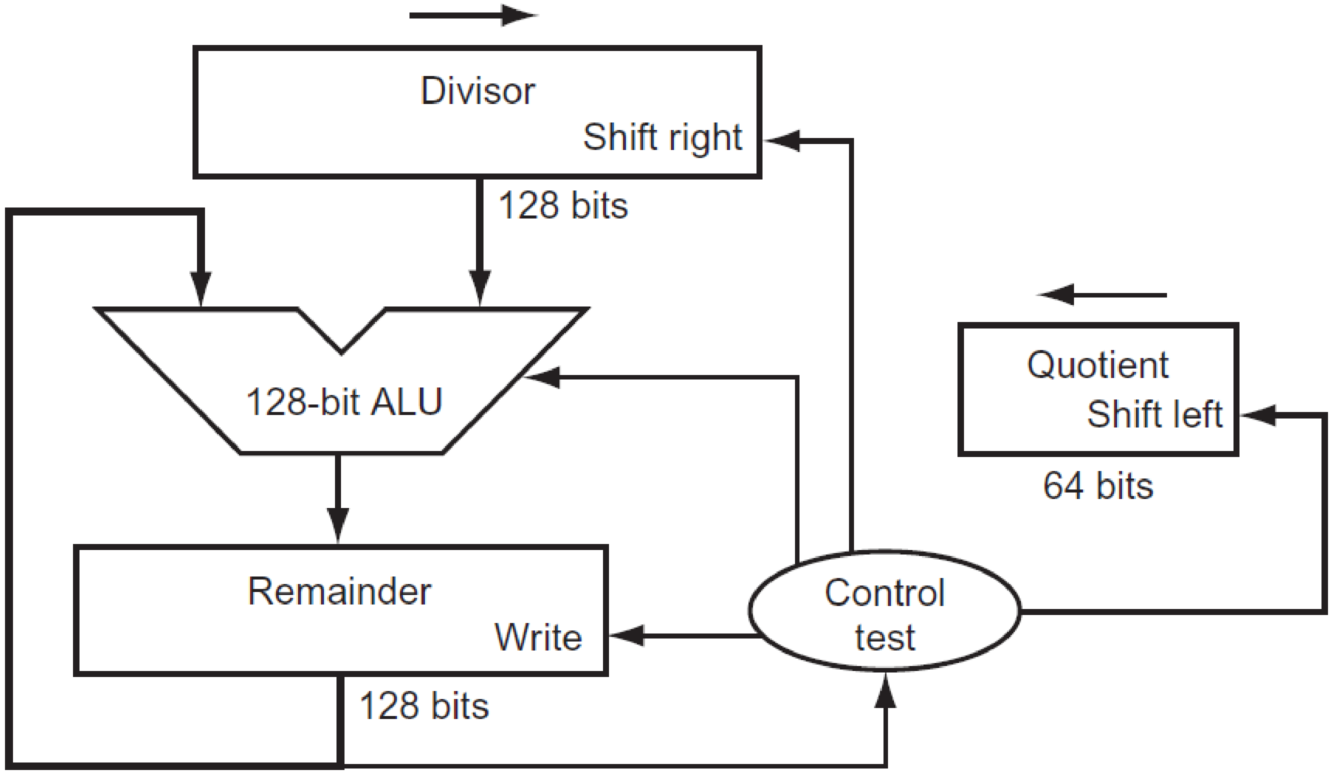


# Division Flowchart



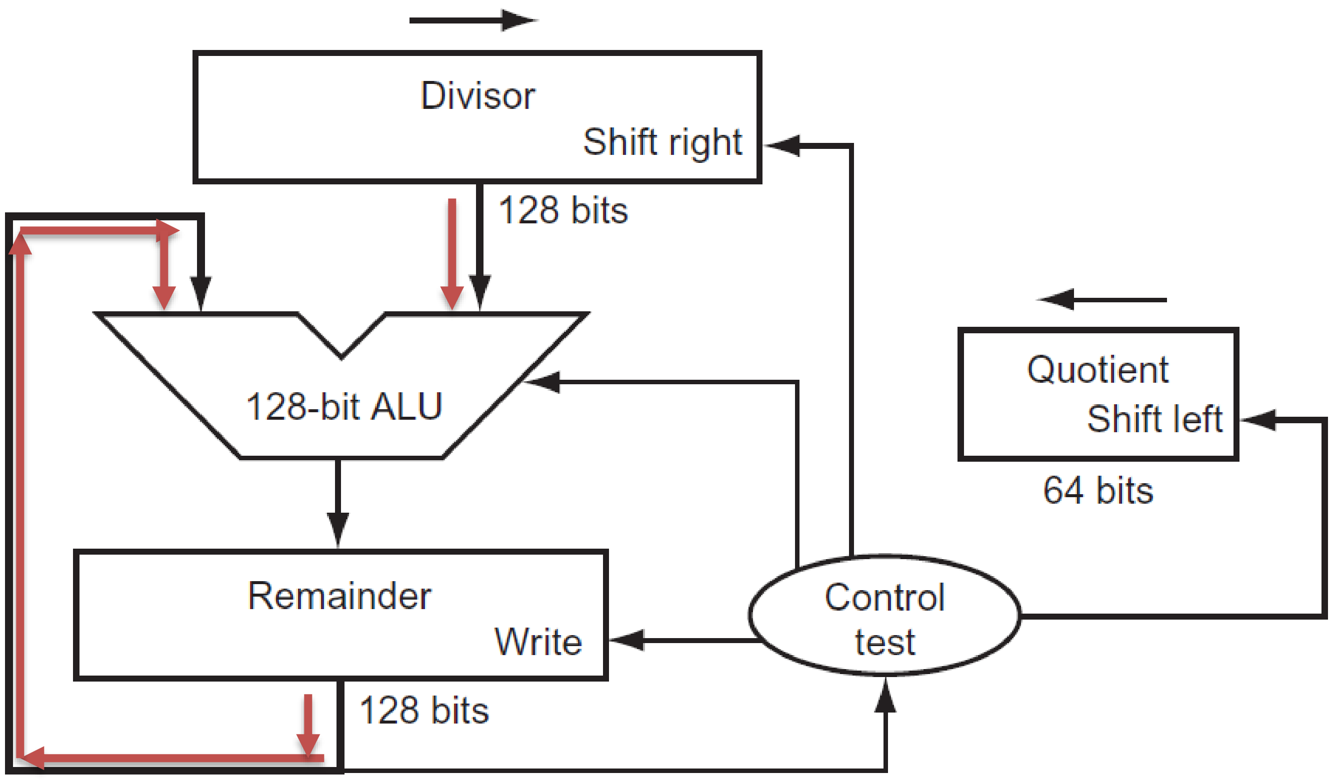
# Division Hardware

1. Remainder = Remainder - Divisor
   1. If remainder < 0,
      1. Shift quotient to left, and add 0 to end
      2. Add the remainder back to devisor, and restore value
   2. If remainder > 0,
      1. Shift quotient to left, and add 1 to end
2. Shift Divisor to the right by 1 bit
3. Repeat 5 times



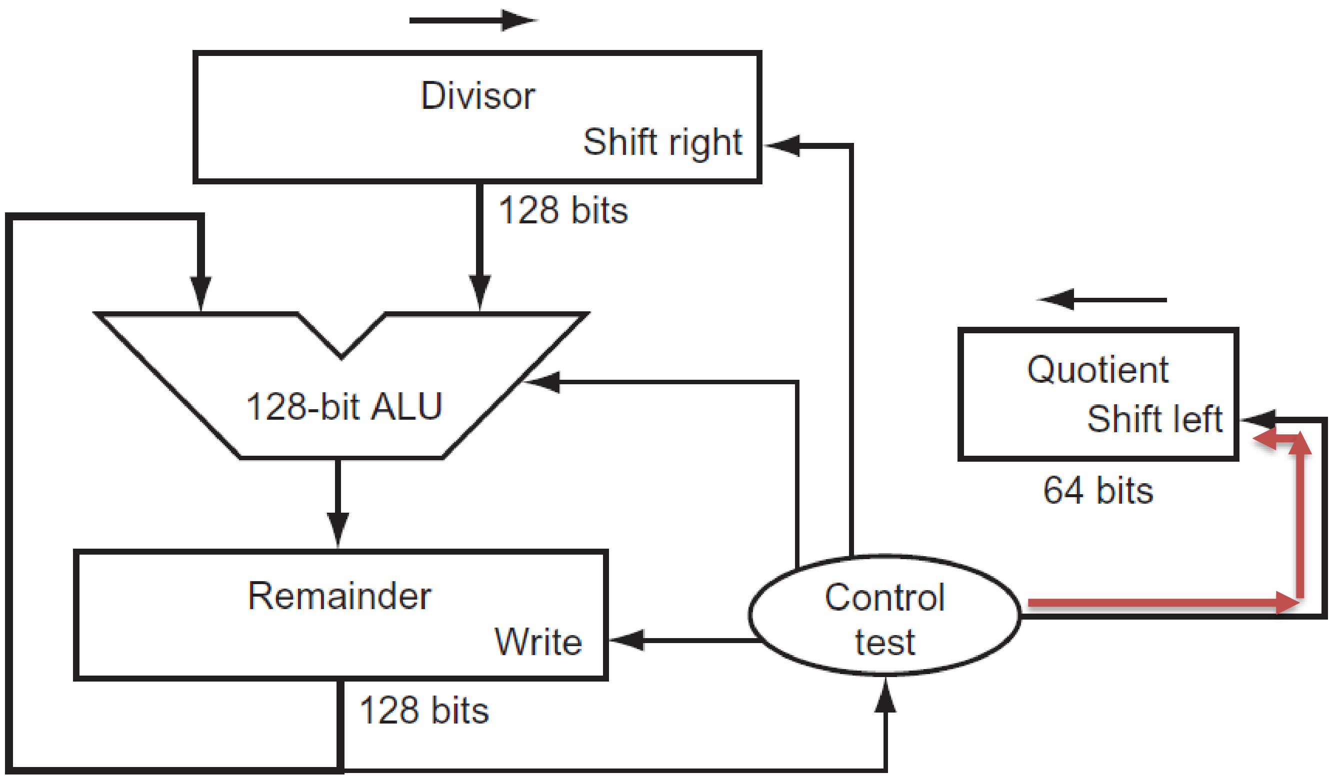
# Division Hardware

1. **Remainder = Remainder - Divisor**
   1. If remainder < 0,
      1. Shift quotient to left, and add 0 to end
      2. Add the remainder back to devisor, and restore value
   2. If remainder > 0,
      1. Shift quotient to left, and add 1 to end
2. Shift Divisor to the right by 1 bit
3. Repeat 5 times



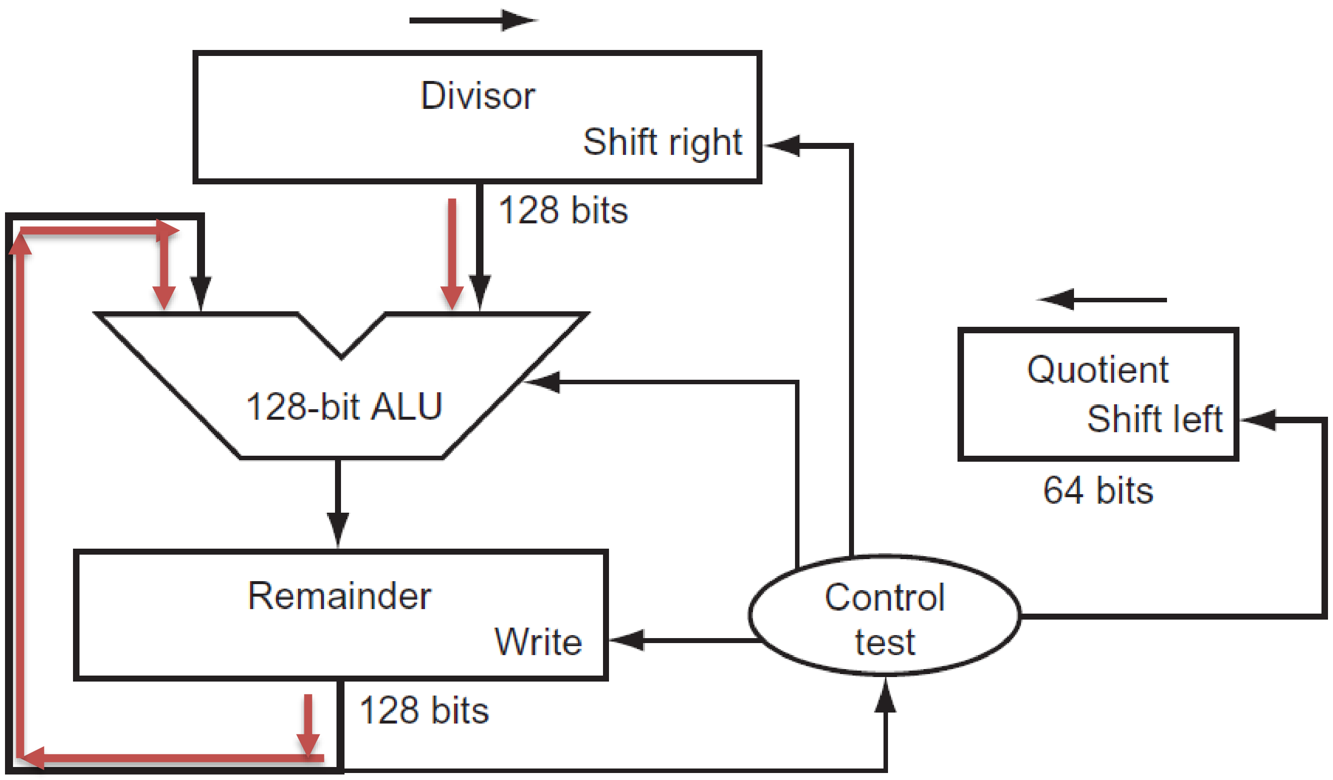
# Division Hardware

1. Remainder = Remainder - Divisor
   1. **If remainder < 0,**
      1. **Shift quotient to left, and add 0 to end**
      2. Add the remainder back to devisor, and restore value
   2. **If remainder > 0,**
      1. **Shift quotient to left, and add 1 to end**
2. Shift Divisor to the right by 1 bit
3. Repeat 5 times



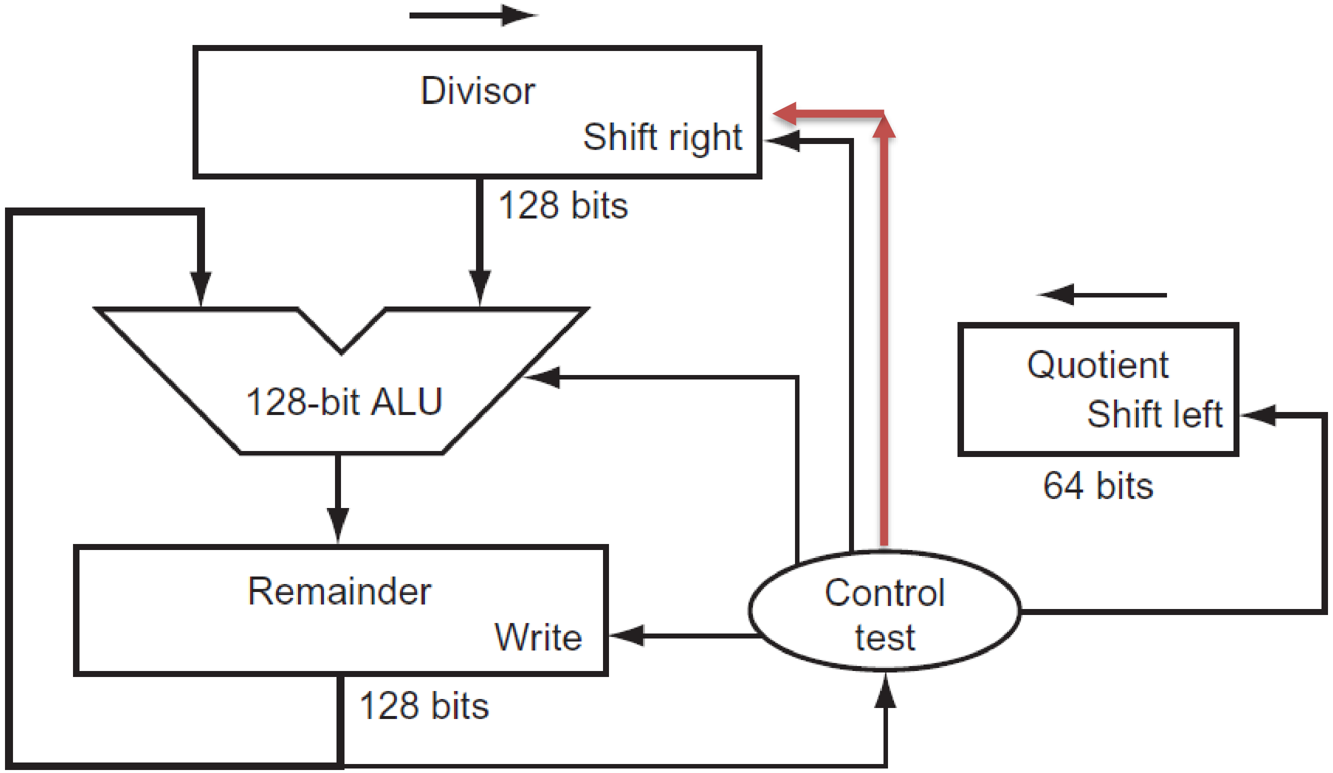
# Division Hardware

1. Remainder = Remainder - Divisor
    1. If remainder < 0,
        1. Shift quotient to left, and add 0 to end
        2. **Add the remainder back to devisor, and restore value**
    2. If remainder > 0,
        1. Shift quotient to left, and add 1 to end
2. Shift Divisor to the right by 1 bit
3. Repeat 5 times



# Division Hardware

1. Remainder = Remainder - Divisor
   1. If remainder < 0,
      1. Shift quotient to left, and add 0 to end
      2. Add the remainder back to devisor, and restore value
   2. If remainder > 0,
      1. Shift quotient to left, and add 1 to end
2. **Shift Divisor to the right by 1 bit**
3. Repeat 5 times



# Signed Division

- Convert to Dividend and Divisor to positive and remember the sign

| Dividend | Divisor | Quotient |
|---|---|---|
| -ve | +ve | -ve |
| +ve | -ve | -ve |
| +ve | +ve | +ve |
| -ve | -ve | +ve |

If Dividend and Divisor signs disagree, then the quotient is negative.

Remainder has the same sign as dividend

# Faster Division

- Can't use parallel hardware as in multiplier
  - Subtraction is conditional on sign of remainder
- Faster dividers (e.g. SRT division) generate multiple quotient bits per step
  - Still require multiple steps

# LEGv8 Division

- Two divide instructions:
  - SDIV: Signed divide
  - UDIV: unsigned divide

# Instructions

| Type | Name |
| --- | --- |
| Arithmetic | ADD, SUB, MUL |
| Data transfer | LDUR, STUR |
| Arithmetic Immediate | ADDI, SUBI, ORRI, ANDI, EORI, MUL, SMULH, UMULH, **SDIV, UDIV** |
| Logical Operations | LSL, LSR, AND, ORR, EOR |
| Branches | B, CBZ, CBNZ, B.Cond |
| Set Condition Flag | ADDS, ADDIS, SUBS, SUBIS, ANDS, ANDIS |

# Floating Point numbers