# TC2009B: Digital design
# Multiplication and Division

Isaac Pérez Andrade

ITESM Guadalajara
School of Engineering & Science
Department of Computer Science
August – December 2021

# References

The following material has been adopted and adapted from

Patterson, D. A., Hennessy, J. L., *Computer Organization and design: The hardware/software interface – ARM edition*, Morgan Kaufmann, 2017.

S. L. Harris and D. M. Harris, *Digital design and computer architecture - ARM edition*, Morgan Kaufmann, 2016.

# Arithmetic for Computers

## Operations on integers

- Addition and subtraction
- Multiplication and division
- Dealing with overflow

## Floating-point real numbers

- Representation and operations

# Integer operations

# Two's complement review

## Assume two's complement format

- Q: What's the range (minimum and maximum values that can be represented) of an $N$-bit two's complement number?

  A: $\left[-(2^{(N-1)}), 2^{(N-1)} - 1\right]$

  - For example, an 8-bit two's complement number may represent values in the range

  $[-2^{8-1}, 2^{8-1} - 1] = [-2^7, 2^7 - 1] = [-128,127]$

# Overflow & underflow

- Q: What is **overflow**?

  A: A condition when the result of a calculation **exceeds** the **maximum** value that can be represented in a numeric format.

- Q: What is **underflow**?

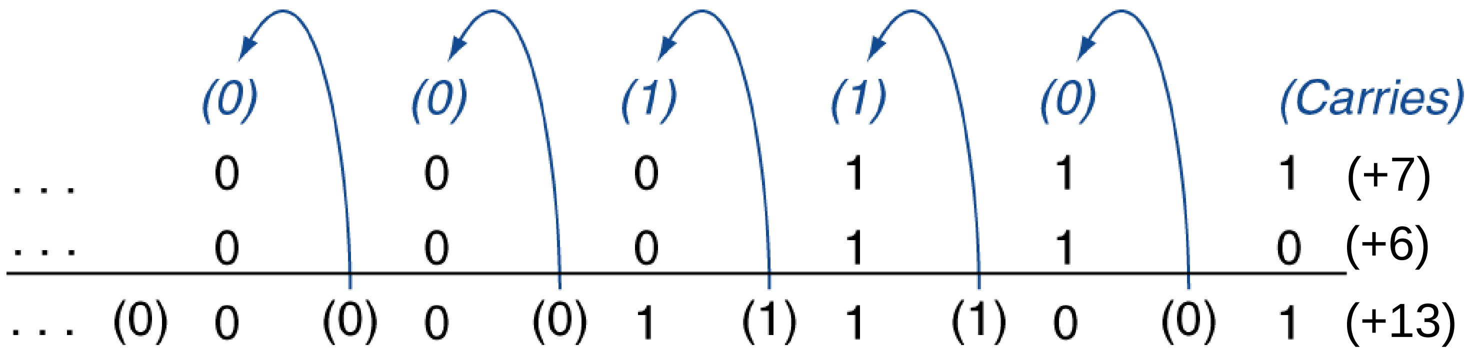
  A: A condition when the result of a calculation is **smaller** than the **minimum** value that can be represented in a numeric format.

- Sometimes, the term overflow is used for describing both conditions.

# Addition

# Integer addition

## Example: $7 + 6$




Overflow if result out of range

- Adding +ve and –ve operands, no overflow
- Adding two +ve operands

  Overflow if result sign is 1
- Adding two –ve operands

  Overflow if result sign is 0

# Integer addition

- Example: Adding two 4-bit two's complement numbers

5 + 1

```
+5:   0101
+1:   0001
+6:   0110
```

3 + 6

```
+3:   0011
+6:   0110
-7:   1001
```

Overflow:+9 and -9 can not be represented in 4-bit two's complement.

−2 + 5

```
-2:   1110
+5:   0101
+3:   0011
```

−7 +(−1)

```
-7:   1001
-1:   1111
-8:   1000
```

−3 +(−6)

```
-3:   1101
-6:   1010
+7:   0111
```

# Subtraction

# Integer subtraction

Addition with negation of second operand

Example: $7 - 6 = 7 + (-6)$

| | |
|---|---|
| +7: | 0000 0000 … 0000 0111 |
| –6: | 1111 1111 … 1111 1010 |
| +1: | 0000 0000 … 0000 0001 |

- Overflow if result out of range
  - Subtracting two +ve or two –ve operands, no overflow
  - Subtracting +ve from –ve operand
    - Overflow if result sign is 0
  - Subtracting –ve from +ve operand
    - Overflow if result sign is 1

# Integer subtraction

- Example: Subtracting two 4-bit two's complement numbers

5 – (+1)

```
+5:   0101
-1:   1111
----------
+4:   0100
```

–2 – (–5)

```
-2:   1110
+5:   0101
----------
+3:   0011
```

–3 – (+6)

```
-3:   1101
-6:   1010
----------
+7:   0111
```

7 –(–1)

```
+7:   0111
+1:   0001
----------
-8:   1000
```

Overflow: -9 and +8 can not be represented in 4-bit two's complement.

# Addition & subtraction overflow summary

- Overflow conditions for additions and subtraction in two's complement.

| Operation | Operand A | Operand B | Result indicating overflow |
|---|---|---|---|
| A + B | $\geq 0$ | $\geq 0$ | $< 0$ |
| A + B | $< 0$ | $< 0$ | $\geq 0$ |
| A – B | $\geq 0$ | $< 0$ | $< 0$ |
| A – B | $< 0$ | $\geq 0$ | $\geq 0$ |

# Multiplication

# Multiplier

- **Partial products** formed by multiplying a single digit of the multiplier with multiplicand
- **Shifted** partial products **summed** to form result

| Decimal | | Binary |
| --- | --- | --- |
| 230 | multiplicand | 0101 |
| x 42 | multiplier | x 0111 |
| 460 | partial products | 0101 |
| + 920 | | 0101 |
| | | 0101 |
| | | + 0000 |
| 9660 | result | 0100011 |

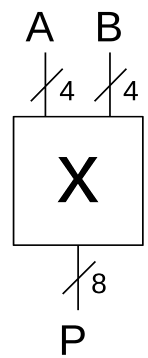
230 x 42 = 9660

5 x 7 = 35

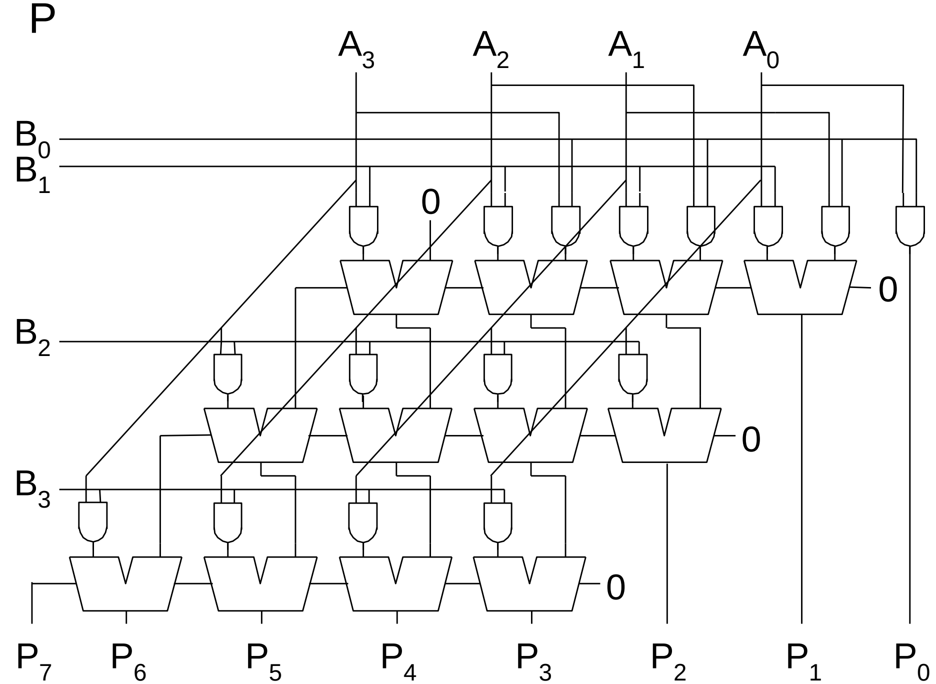
# Parallel multiplication

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | $A_3$ | $A_2$ | $A_1$ | $A_0$ |
| | | | x | $B_3$ | $B_2$ | $B_1$ | $B_0$ |
| | | | | $A_3B_0$ | $A_2B_0$ | $A_1B_0$ | $A_0B_0$ |
| | | | $A_3B_1$ | $A_2B_1$ | $A_1B_1$ | $A_0B_1$ | |
| | | $A_3B_2$ | $A_2B_2$ | $A_1B_2$ | $A_0B_2$ | | |
| + | $A_3B_3$ | $A_2B_3$ | $A_1B_3$ | $A_0B_3$ | | | |
| $P_7$ | $P_6$ | $P_5$ | $P_4$ | $P_3$ | $P_2$ | $P_1$ | $P_0$ |

# Parallel multiplication




| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | $A_3$ | $A_2$ | $A_1$ | $A_0$ |
| | | | x | | $B_3$ | $B_2$ | $B_1$ | $B_0$ |
| | | | | | $A_3B_0$ | $A_2B_0$ | $A_1B_0$ | $A_0B_0$ |
| | | | | $A_3B_1$ | $A_2B_1$ | $A_1B_1$ | $A_0B_1$ | |
| | | | $A_3B_2$ | $A_2B_2$ | $A_1B_2$ | $A_0B_2$ | | |
| + | | $A_3B_3$ | $A_2B_3$ | $A_1B_3$ | $A_0B_3$ | | | |
| | $P_7$ | $P_6$ | $P_5$ | $P_4$ | $P_3$ | $P_2$ | $P_1$ | $P_0$ |



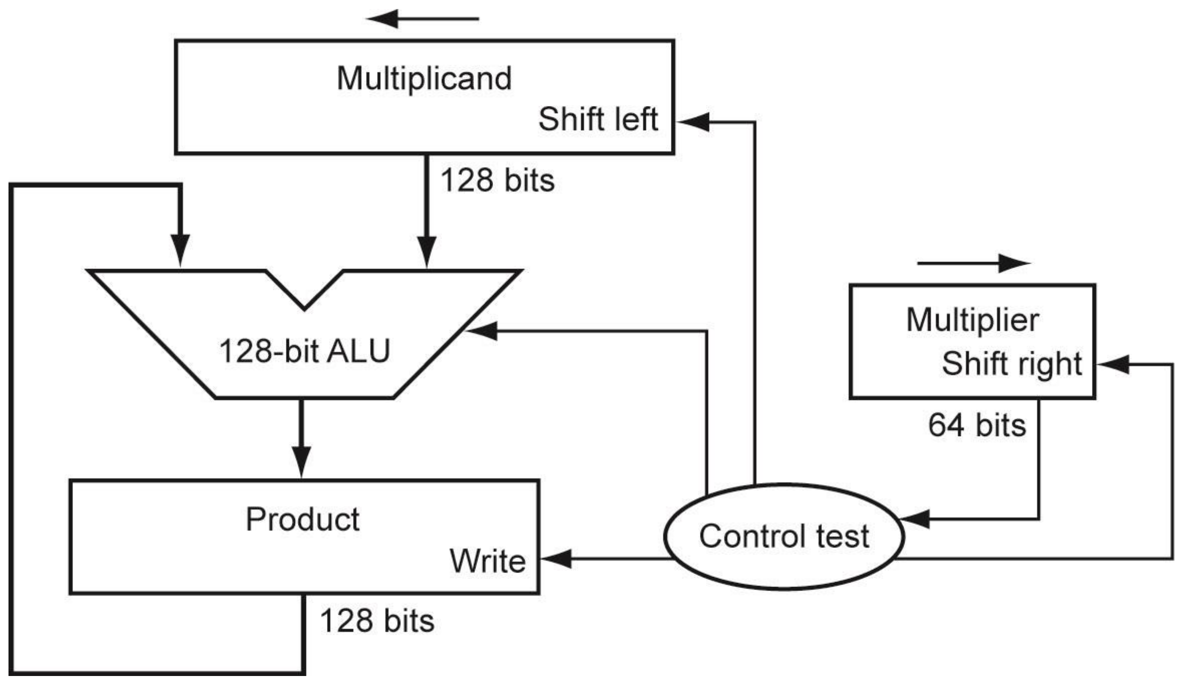
# Sequential multiplication

- Start with long-multiplication approach

Assume we want to multiply two 64-bit numbers

multiplicand
multiplier
product

```
       1000
  ×    1001
  ---------
       1000
      0000
     0000
    1000
  ---------
   01001000
```



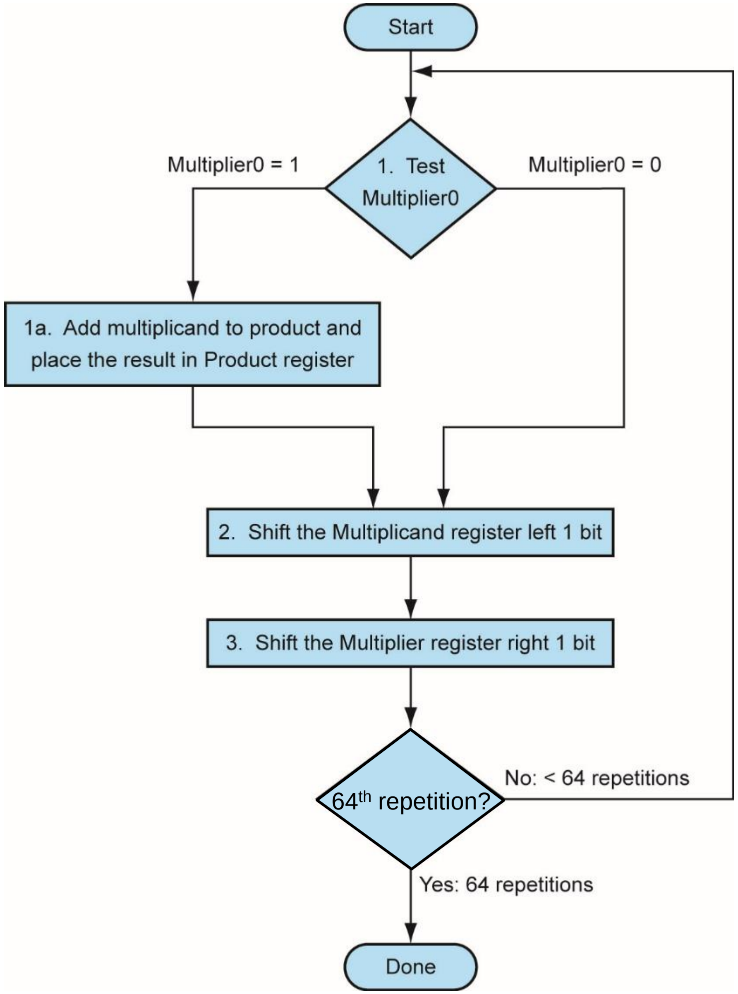

Length of product is the sum of operand lengths.

$$A_{M-bits} \times B_{N-bits} = X_{(M+N)-bits}$$

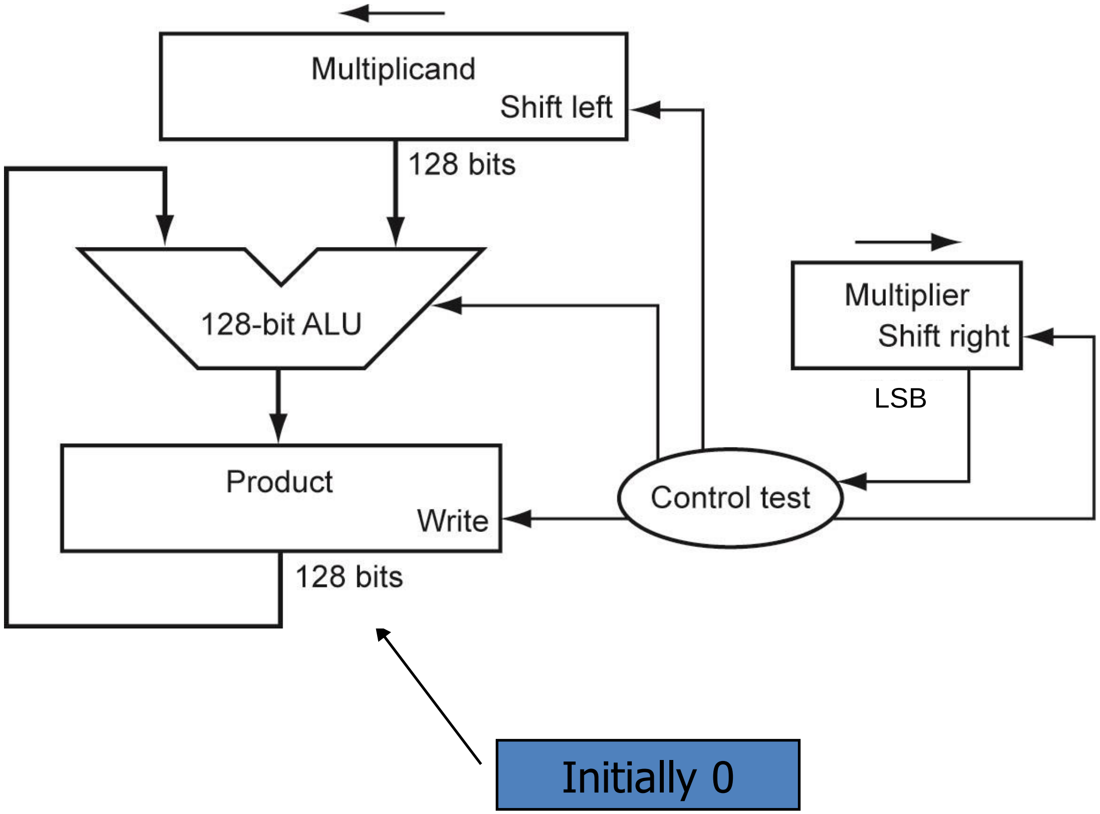
What can be improved in this architecture?

# Multiplication hardware

There's one error in the flow chart. Can you spot it?

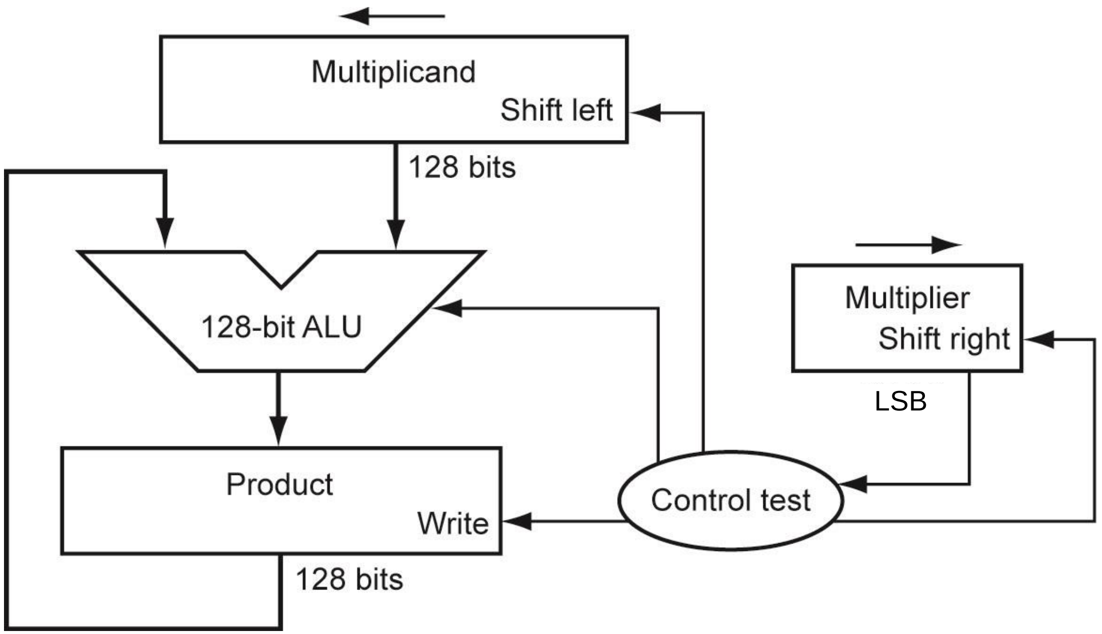



What can be improved in this architecture?



# Multiplication hardware

This architecture has a major flaw.
Can you spot it?



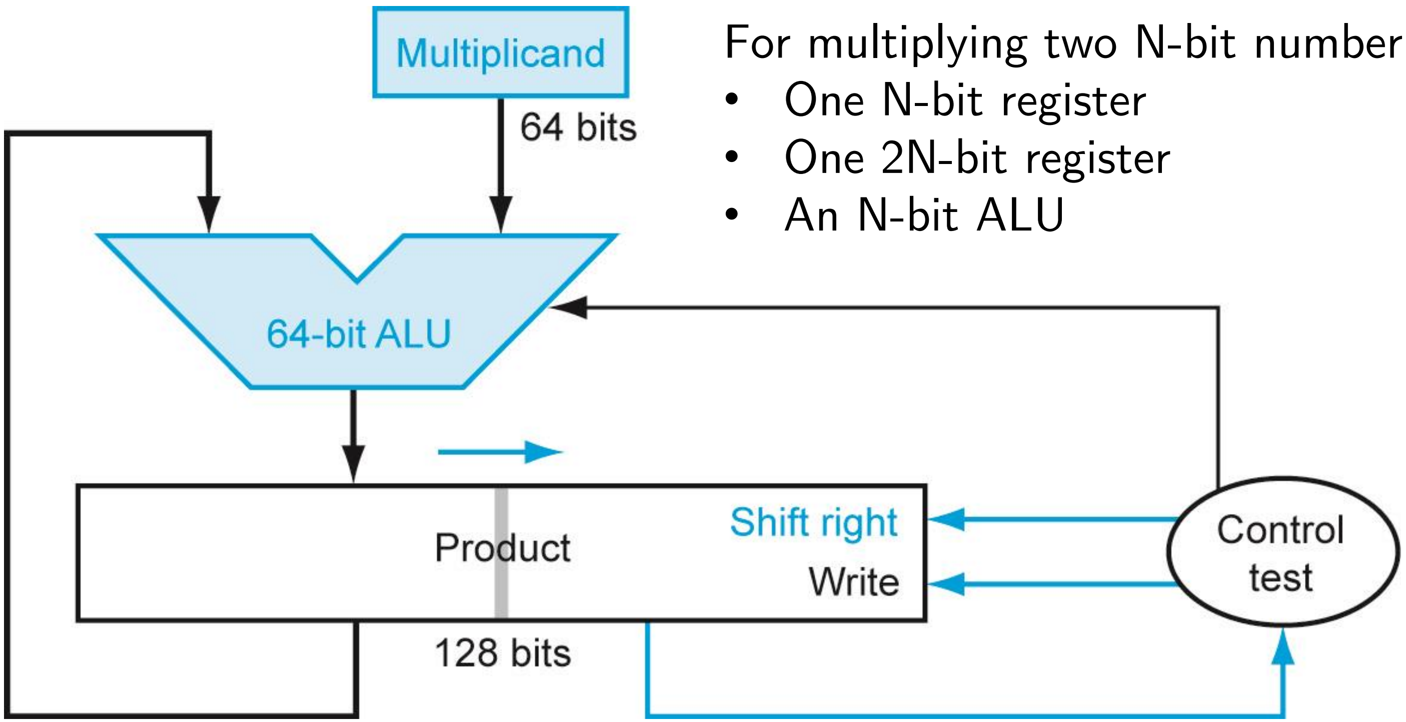

For multiplying two N-bit number, it requires:

- Two 2N-bit registers
- One N-bit register
- A 2N-bit ALU

This is a waste of resources!

# Optimised multiplier

- Perform steps in parallel: add/shift

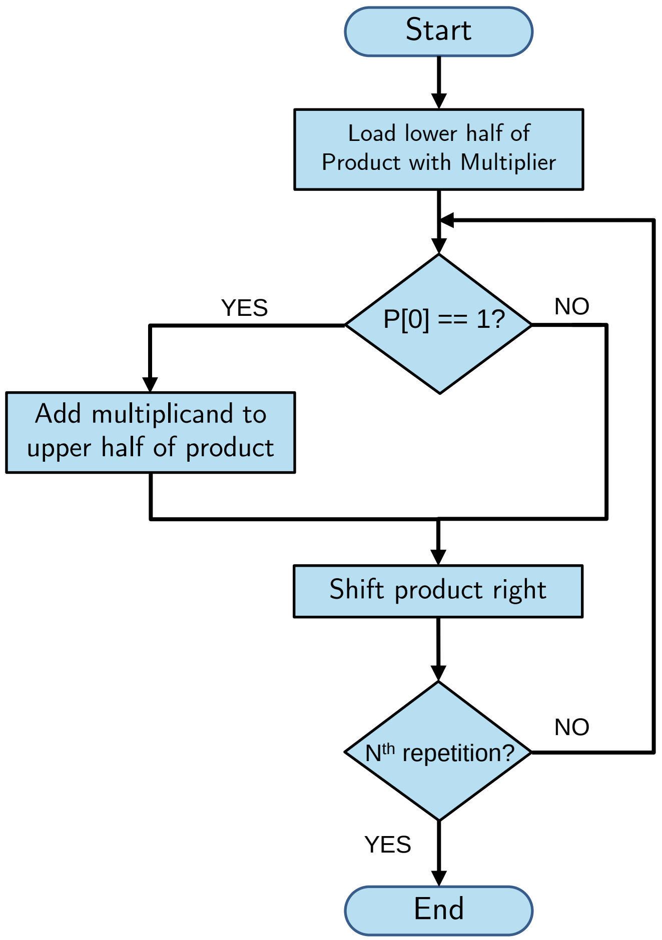


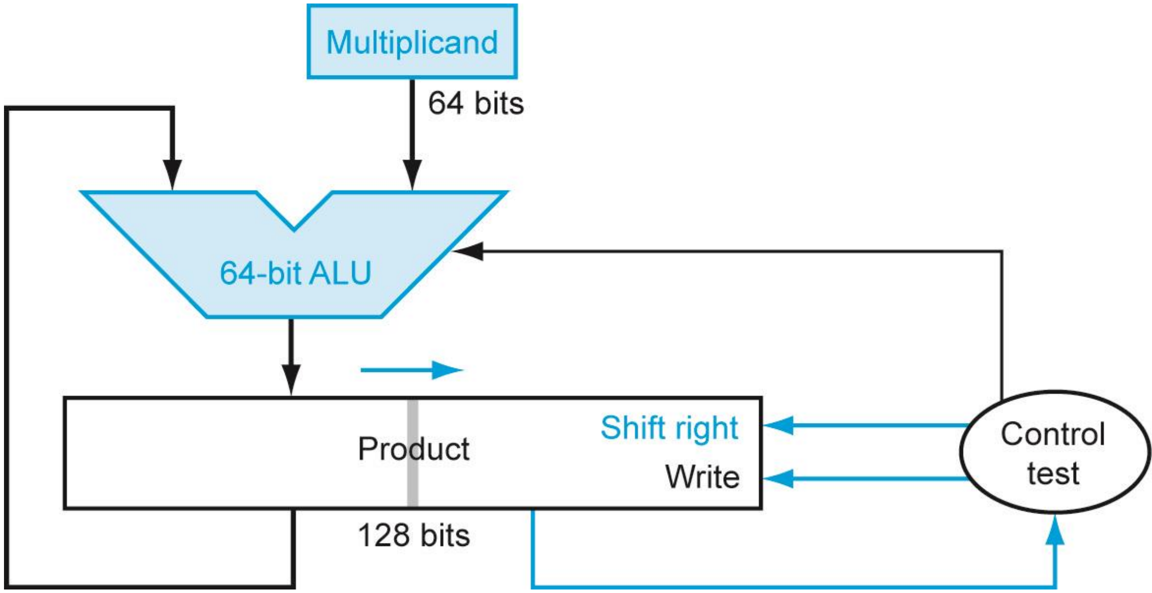

For multiplying two N-bit number, it requires:
- One N-bit register
- One 2N-bit register
- An N-bit ALU

One cycle per partial-product addition

- That's ok, if frequency of multiplications is low

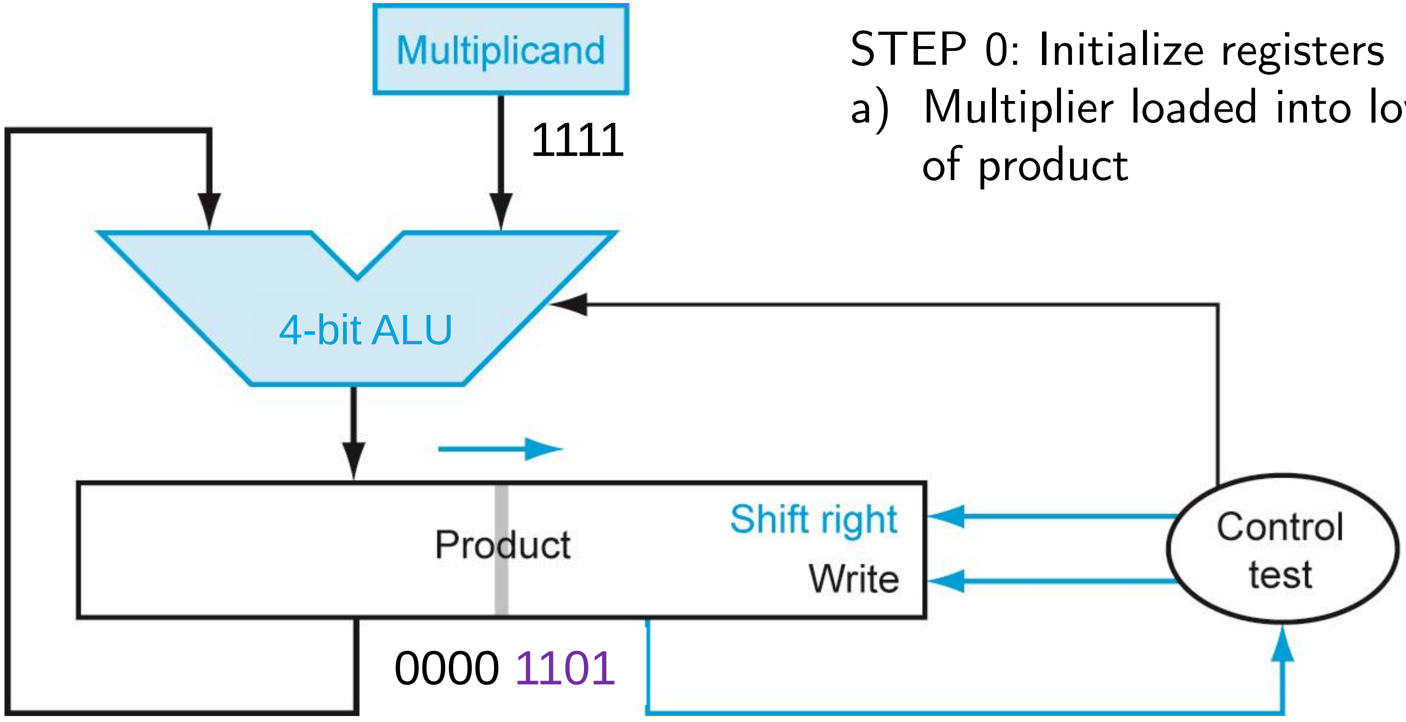
# Optimised multiplier



# Optimised multiplier - example

- Multiplicand $= 15_{10}$: 1111
- Multiplier $= 13_{10}$: 1101




STEP 0: Initialize registers

a) Multiplier loaded into lower half of product

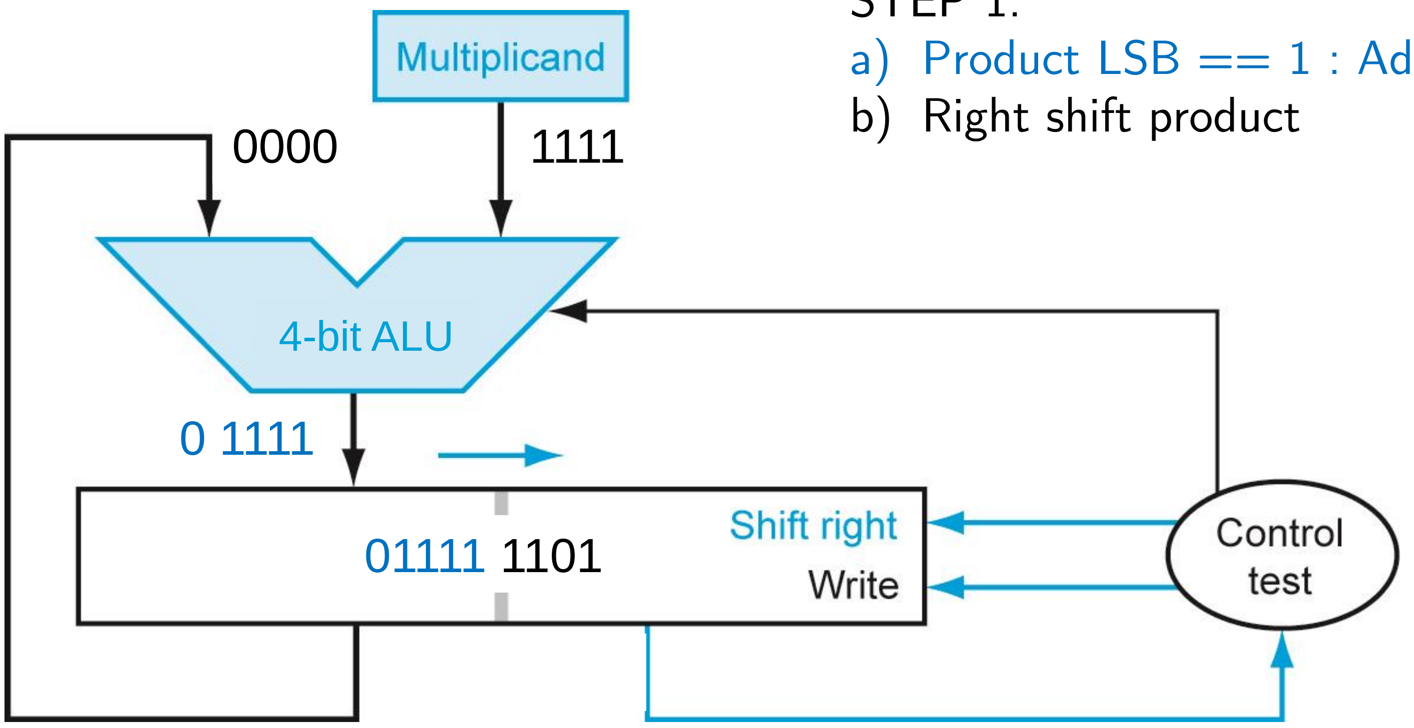
# Optimised multiplier - example

- Multiplicand $= 15_{10}$: 1111
- Multiplier $= 13_{10}$: 1101

STEP 1:

a) Product LSB == 1 : Add

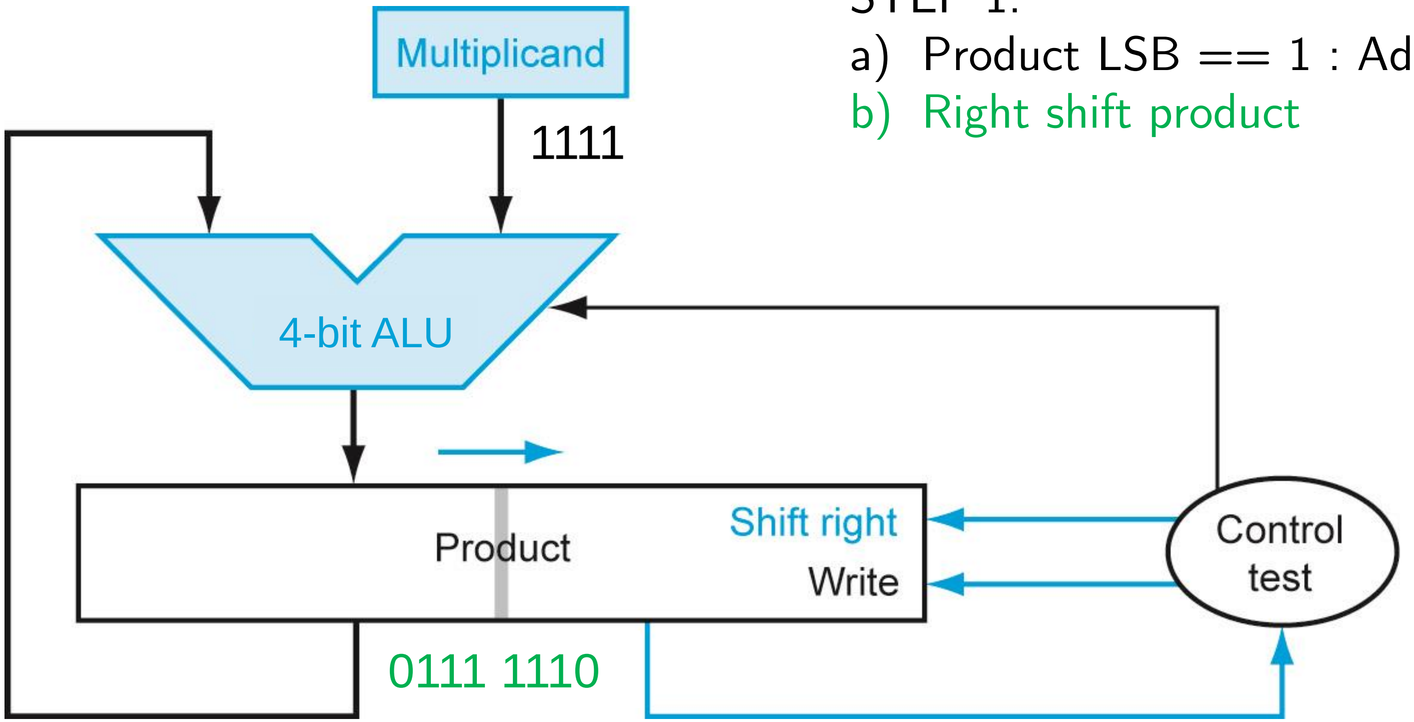
b) Right shift product



# Optimised multiplier - example

- Multiplicand $= \mathbf{15_{10}}$: 1111
- Multiplier $= \mathbf{13_{10}}$: 1101

STEP 1:
a) Product LSB == 1 : Add
b) Right shift product



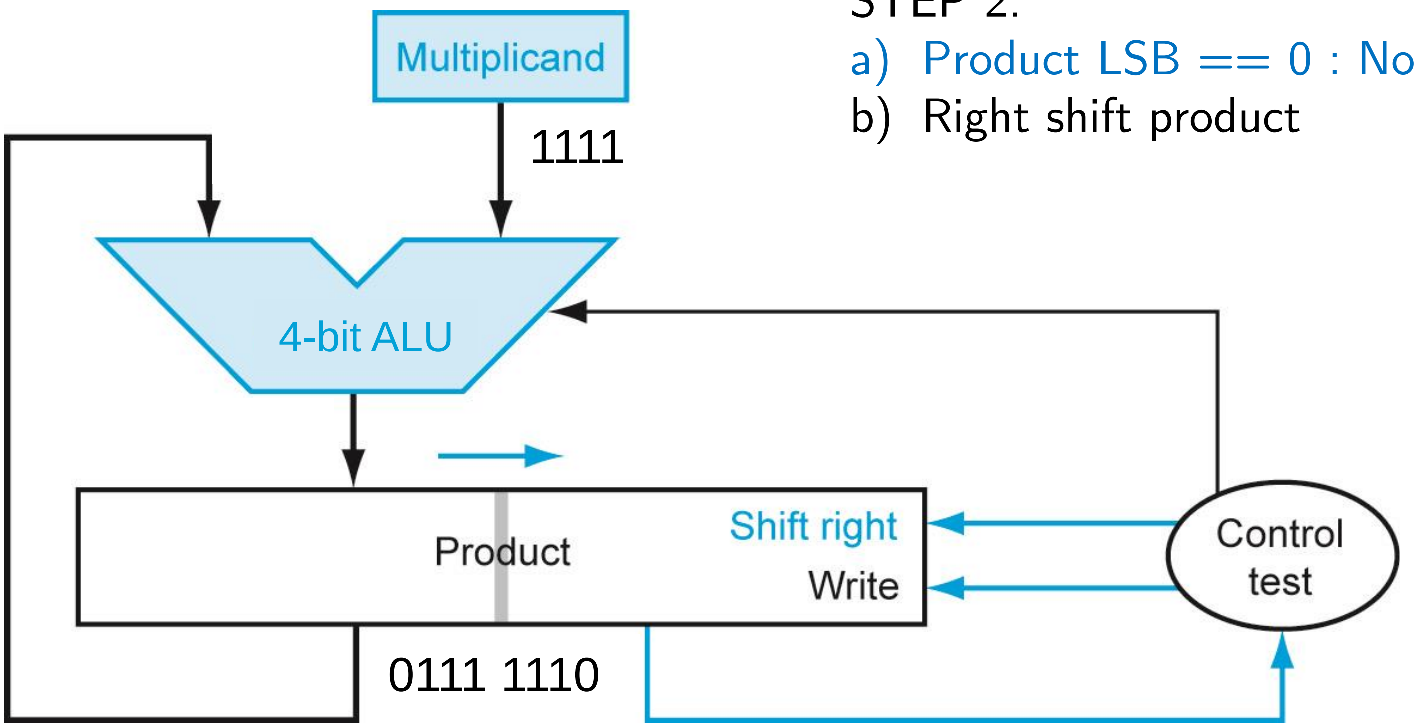
# Optimised multiplier - example

- Multiplicand $= \mathbf{15_{10}}$: 1111
- Multiplier $= \mathbf{13_{10}}$: 1101

STEP 2:

a) Product LSB == 0 : No Add

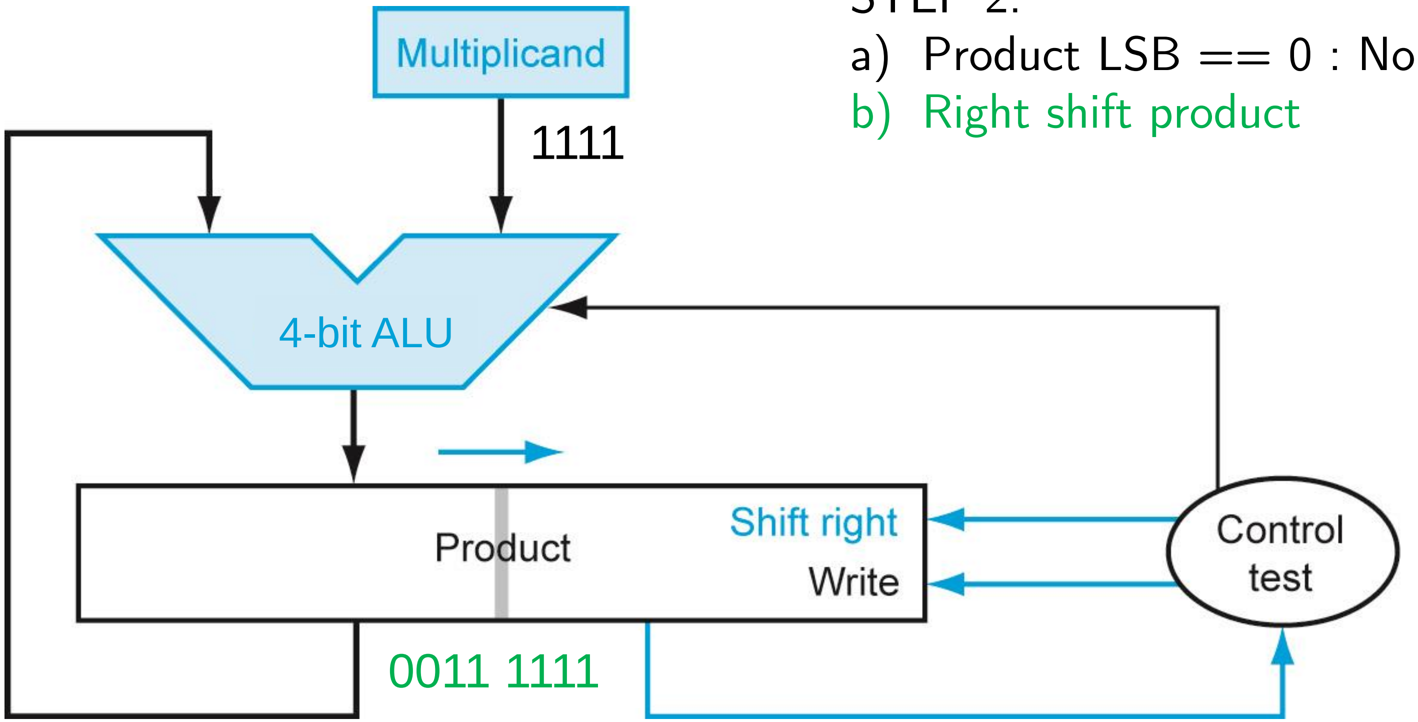
b) Right shift product



# Optimised multiplier - example

- Multiplicand $= \mathbf{15_{10}}$: 1111
- Multiplier $= \mathbf{13_{10}}$: 1101

STEP 2:
a) Product LSB == 0 : No Add
b) Right shift product



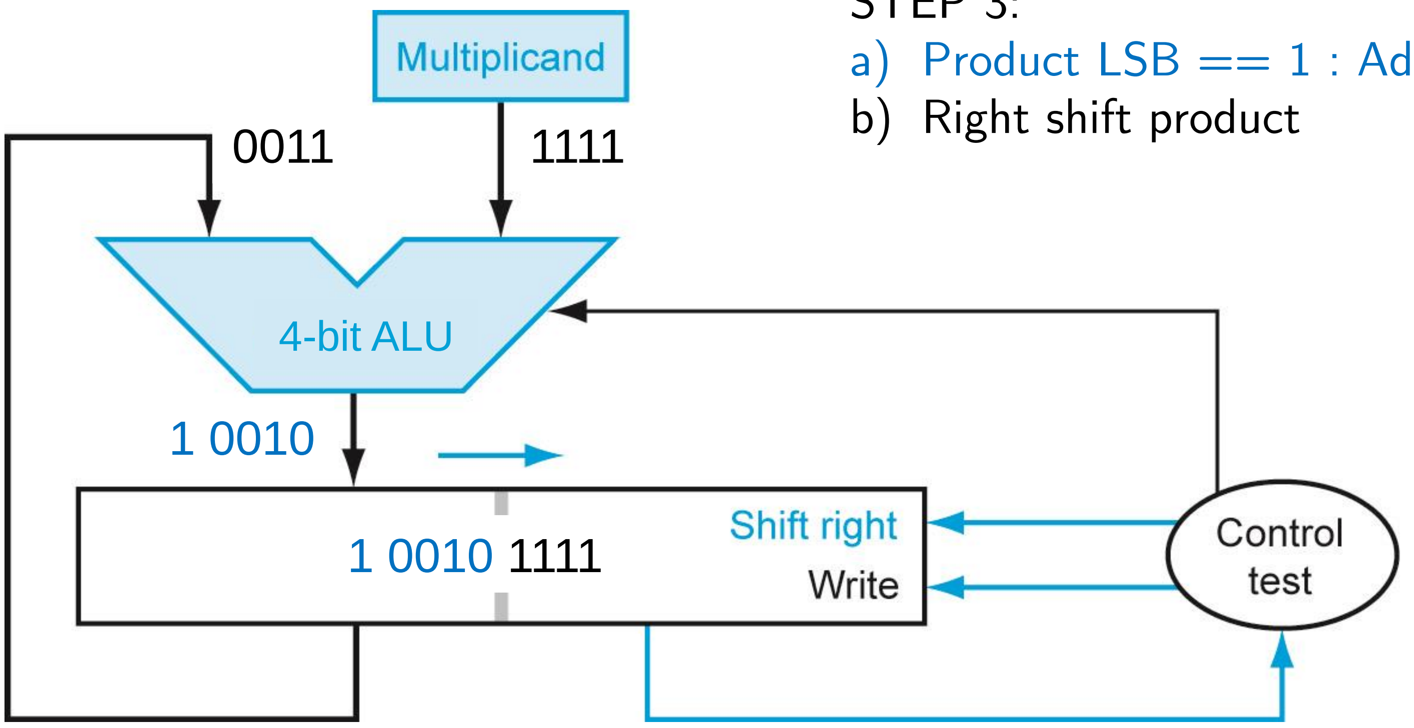
# Optimised multiplier - example

- Multiplicand $= 15_{10}$: 1111
- Multiplier $= 13_{10}$: 1101

STEP 3:

a) Product LSB == 1 : Add

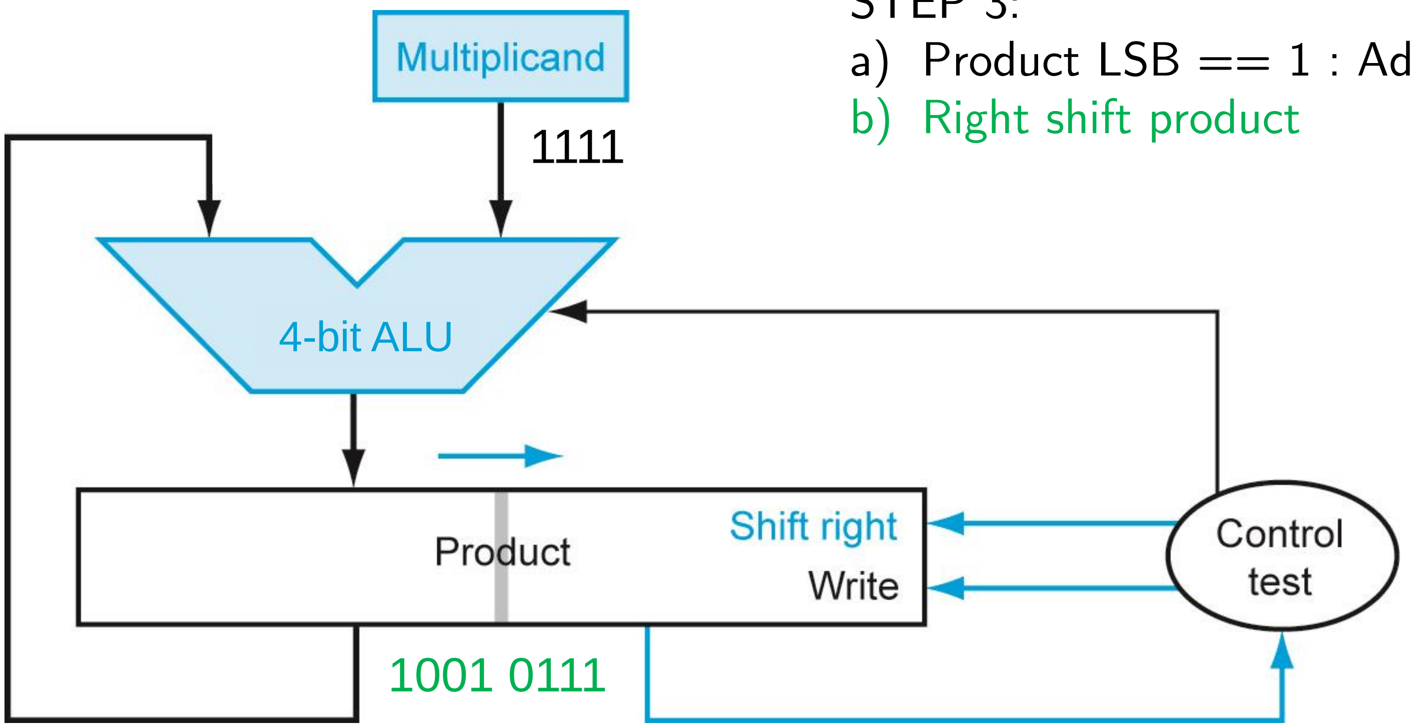
b) Right shift product



# Optimised multiplier - example

- Multiplicand $= \mathbf{15_{10}}$: 1111
- Multiplier $= \mathbf{13_{10}}$: 1101

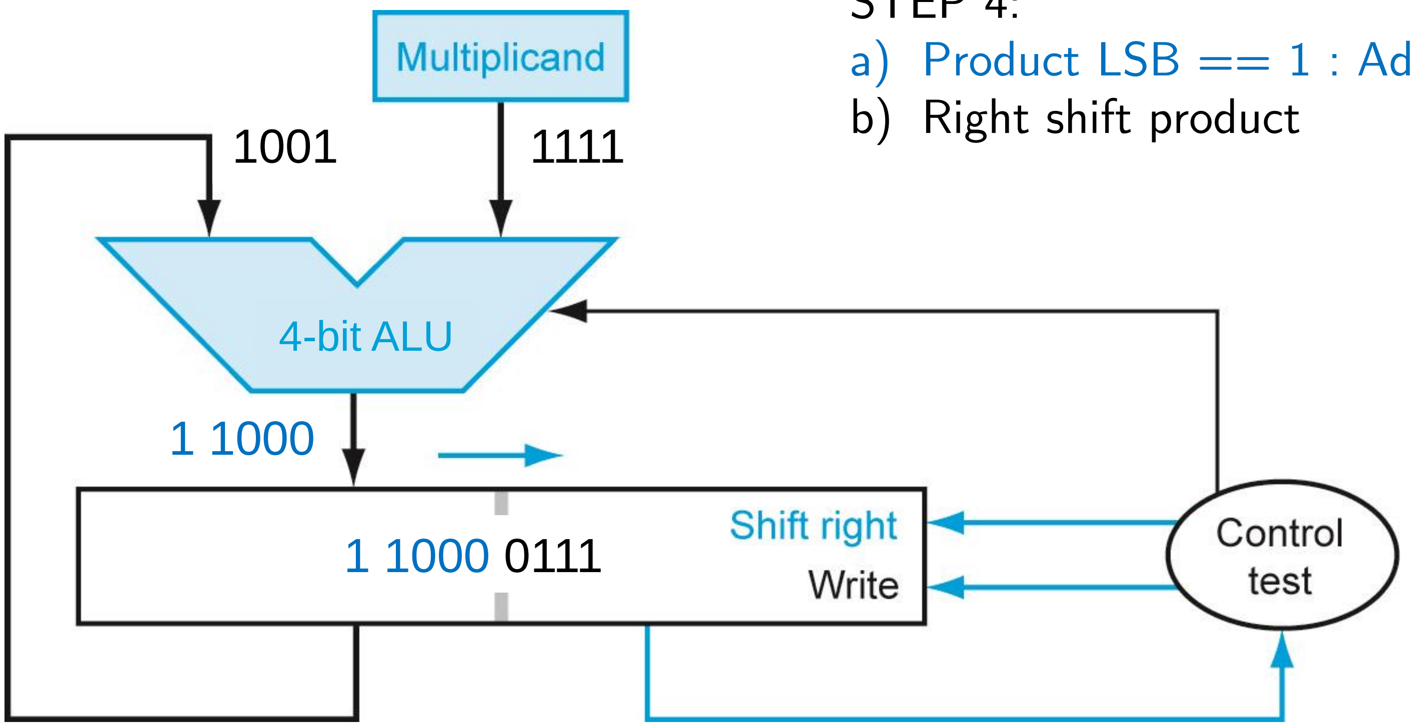
STEP 3:
a) Product LSB == 1 : Add
b) Right shift product



# Optimised multiplier - example

- Multiplicand $= \mathbf{15_{10}}$: 1111
- Multiplier $= \mathbf{13_{10}}$: 1101

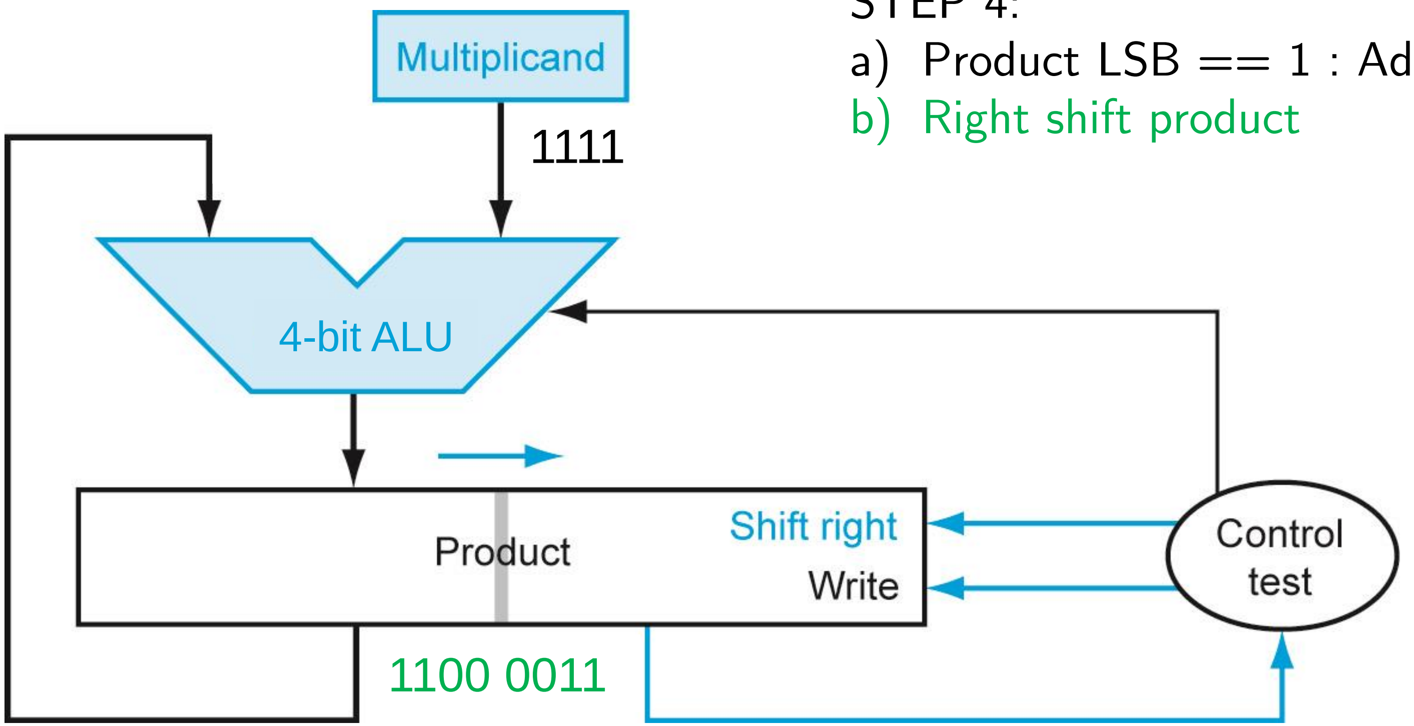
STEP 4:
a) Product LSB == 1 : Add
b) Right shift product



# Optimised multiplier - example

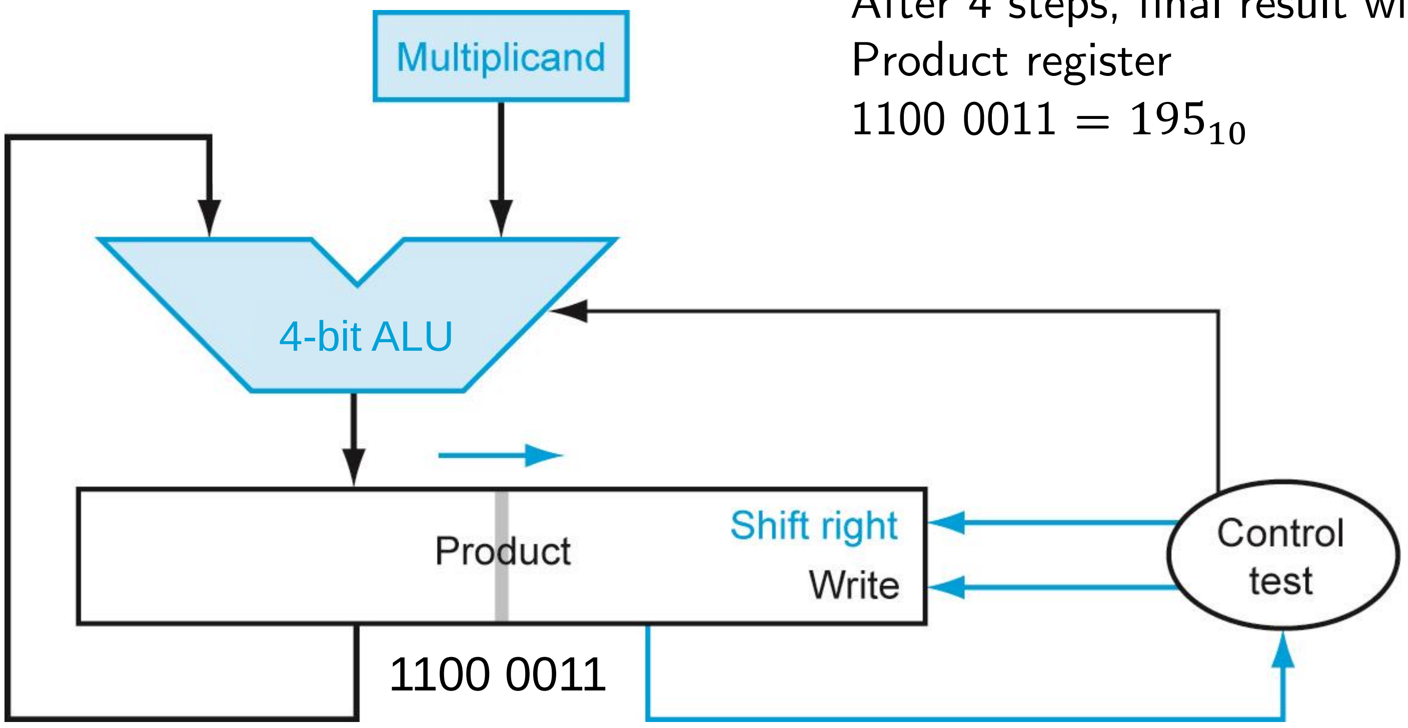
- Multiplicand = $\mathbf{15_{10}}$: 1111
- Multiplier = $\mathbf{13_{10}}$: 1101

STEP 4:
a) Product LSB == 1 : Add
b) Right shift product



# Optimised multiplier - example

- Multiplicand $= 15_{10}$: 1111
- Multiplier $= 13_{10}$: 1101



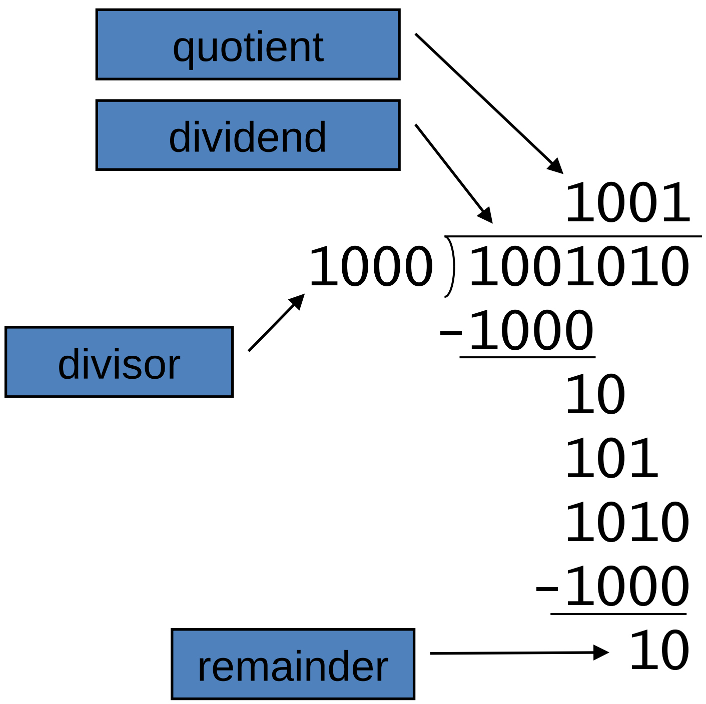

After 4 steps, final result will be in Product register
$1100\ 0011 = 195_{10}$

# Signed multiplication

- So far, we've only dealt with unsinged operands.
- What happens in signed multiplication?
- For adding two signed N-bit numbers:
  1. Convert both multiplicand and multiplier to positive numbers and keep track of their respective sign.
  2. Apply multiplication algorithm N-1 times.
  3. Negate product if signs are not the same.
- Alternatively, previous algorithm works for signed numbers if shifts are performed using sign extension.

# Division

# Division




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

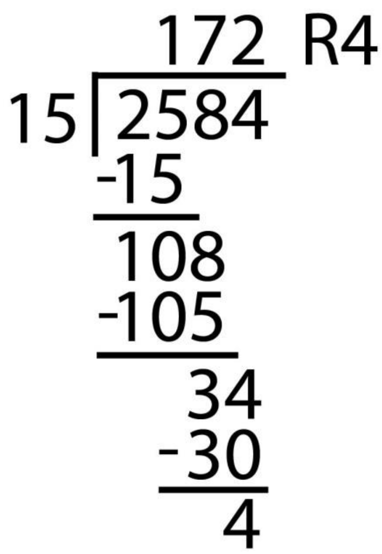
- Check for 0 divisor
- Long division approach
  - If divisor $\leq$ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division
  - Do the subtract, and if remainder goes $< 0$, add divisor back
- Signed division
  - Divide using absolute values
  - Adjust sign of quotient and remainder as required

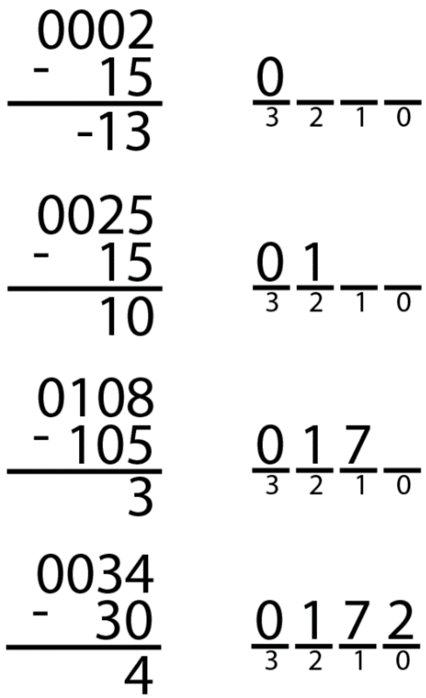
# Divider

$$\mathbf{A/B = Q + R}$$

**Decimal Example:** $\mathbf{2584/15 = 172\ R4}$

**Long-Hand:**



# Divider

$$\mathbf{A/B = Q + R}$$

**Decimal:**

$$\mathbf{2584/15 = 172\ R4}$$

$$\begin{array}{r} 0002 \\ -\ \ 15 \\ \hline -13 \end{array} \qquad \underset{3}{\underline{0}}\ \underset{2}{\underline{\ \ }}\ \underset{1}{\underline{\ \ }}\ \underset{0}{\underline{\ \ }}$$

$$\begin{array}{r} 0025 \\ -\ \ 15 \\ \hline 10 \end{array} \qquad \underset{3}{\underline{0}}\ \underset{2}{\underline{1}}\ \underset{1}{\underline{\ \ }}\ \underset{0}{\underline{\ \ }}$$

$$\begin{array}{r} 0108 \\ -\ 105 \\ \hline 3 \end{array} \qquad \underset{3}{\underline{0}}\ \underset{2}{\underline{1}}\ \underset{1}{\underline{7}}\ \underset{0}{\underline{\ \ }}$$

$$\begin{array}{r} 0034 \\ -\ \ 30 \\ \hline 4 \end{array} \qquad \underset{3}{\underline{0}}\ \underset{2}{\underline{1}}\ \underset{1}{\underline{7}}\ \underset{0}{\underline{2}}$$

**Binary:**

$$\mathbf{1101/10 = 0110\ R1}$$

$$\begin{array}{r} 0001 \\ -0010 \\ \hline 1111 \end{array} \qquad \underset{3}{\underline{0}}\ \underset{2}{\underline{\ \ }}\ \underset{1}{\underline{\ \ }}\ \underset{0}{\underline{\ \ }}$$

$$\begin{array}{r} 0011 \\ -0010 \\ \hline 0001 \end{array} \qquad \underset{3}{\underline{0}}\ \underset{2}{\underline{1}}\ \underset{1}{\underline{\ \ }}\ \underset{0}{\underline{\ \ }}$$

$$\begin{array}{r} 0010 \\ -0010 \\ \hline 0000 \end{array} \qquad \underset{3}{\underline{0}}\ \underset{2}{\underline{1}}\ \underset{1}{\underline{1}}\ \underset{0}{\underline{\ \ }}$$

$$\begin{array}{r} 0001 \\ -0010 \\ \hline 1111 \end{array} \qquad \underset{3}{\underline{0}}\ \underset{2}{\underline{1}}\ \underset{1}{\underline{1}}\ \underset{0}{\underline{0}}\ \ \mathrm{R1}$$

# Divider

**A/B = Q + R/B**

$R' = 0$

for $i$ = N-1 to 0

  $R = \{R' << 1, A_i\}$

  $D = R - B$

  if $D < 0$, $Q_i = 0$; $R' = R$

  else    $Q_i = 1$; $R' = D$

$R' = R$

**Binary:** **1101/10 = 0110 R1**

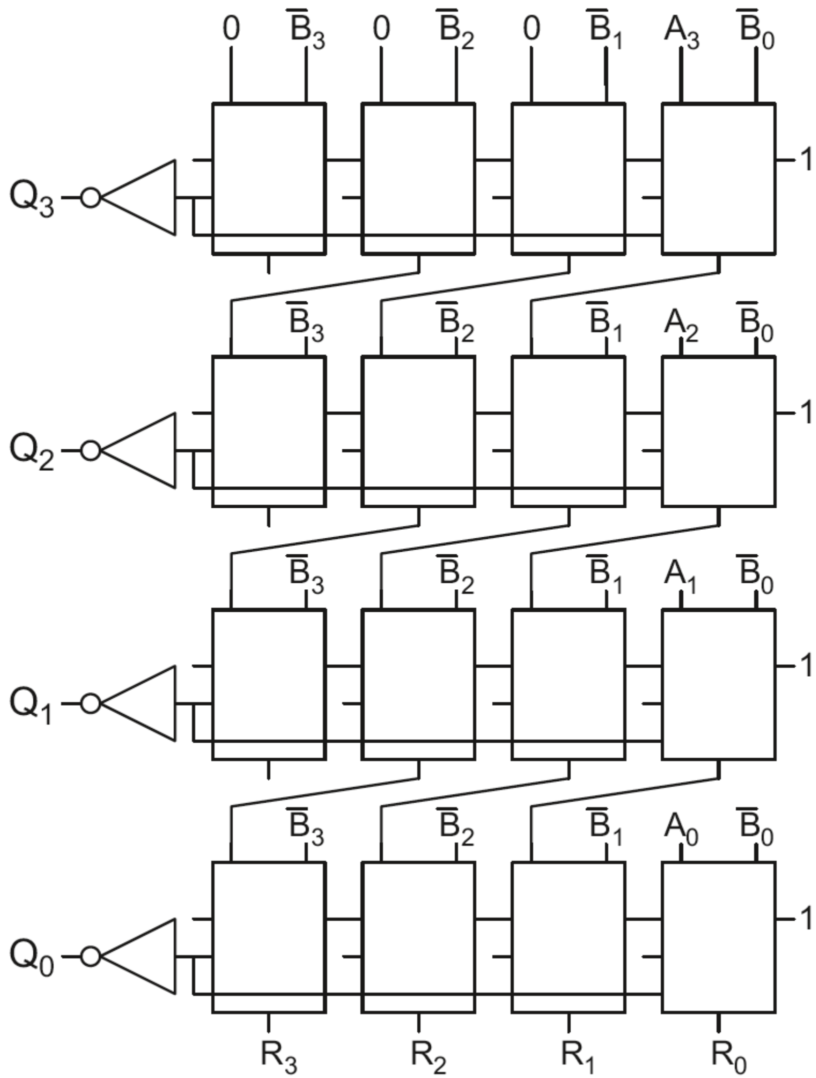
$$\begin{array}{r} 0001 \\ -\,0010 \\ \hline 1111 \end{array} \qquad \underset{3}{\underline{0}}\ \underset{2}{\underline{\ \ }}\ \underset{1}{\underline{\ \ }}\ \underset{0}{\underline{\ \ }}$$

$$\begin{array}{r} 0011 \\ -\,0010 \\ \hline 0001 \end{array} \qquad \underset{3}{\underline{0}}\ \underset{2}{\underline{1}}\ \underset{1}{\underline{\ \ }}\ \underset{0}{\underline{\ \ }}$$

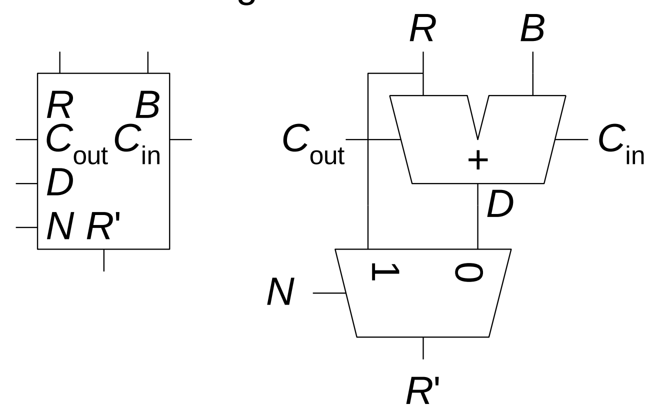
$$\begin{array}{r} 0010 \\ -\,0010 \\ \hline 0000 \end{array} \qquad \underset{3}{\underline{0}}\ \underset{2}{\underline{1}}\ \underset{1}{\underline{1}}\ \underset{0}{\underline{\ \ }}$$

$$\begin{array}{r} 0001 \\ -\,0010 \\ \hline 1111 \end{array} \qquad \underset{3}{\underline{0}}\ \underset{2}{\underline{1}}\ \underset{1}{\underline{1}}\ \underset{0}{\underline{0}}\ \ \mathrm{R1}$$

# 4x4 Divider




Legend




**Division:** $A/B = Q + R/B$

$R' = 0$

for $i$ = N-1 to 0

  $R = \{R' << 1, A_i\}$

  $D = R - B$

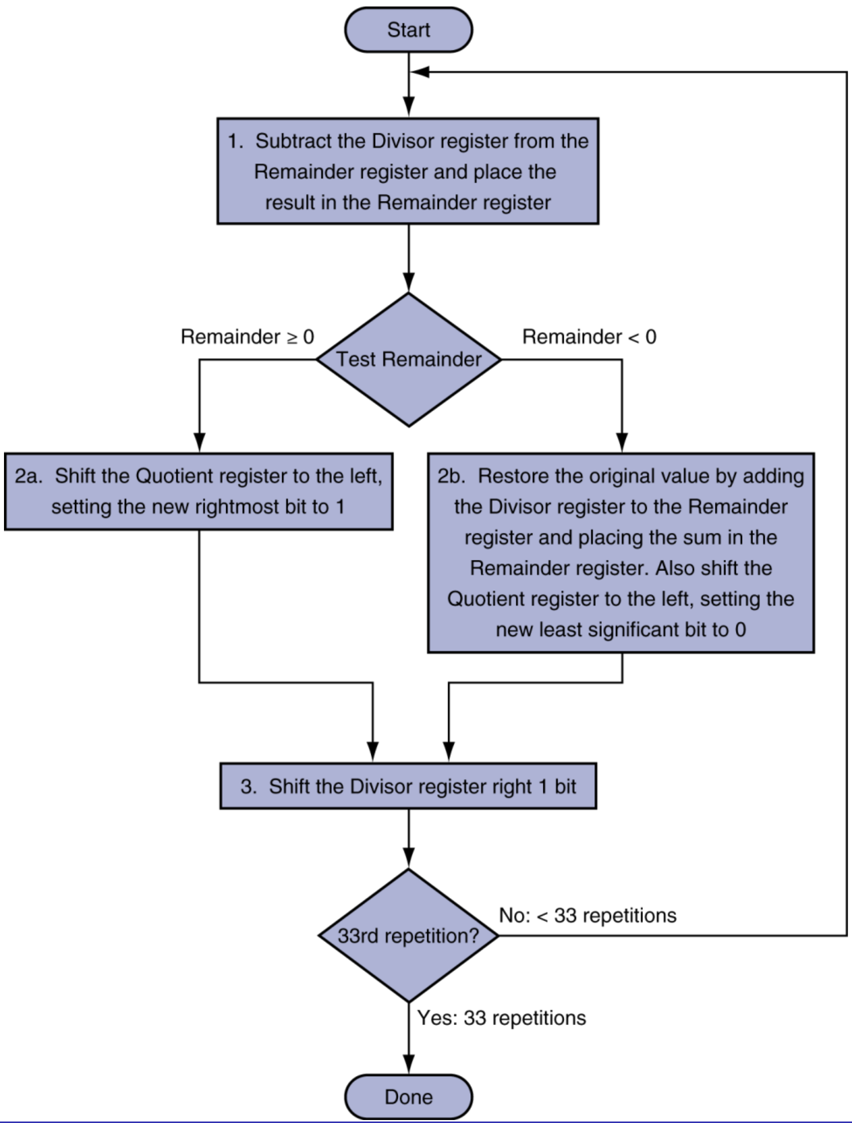
  if $D < 0$, $Q_i=0$, $R'=R$

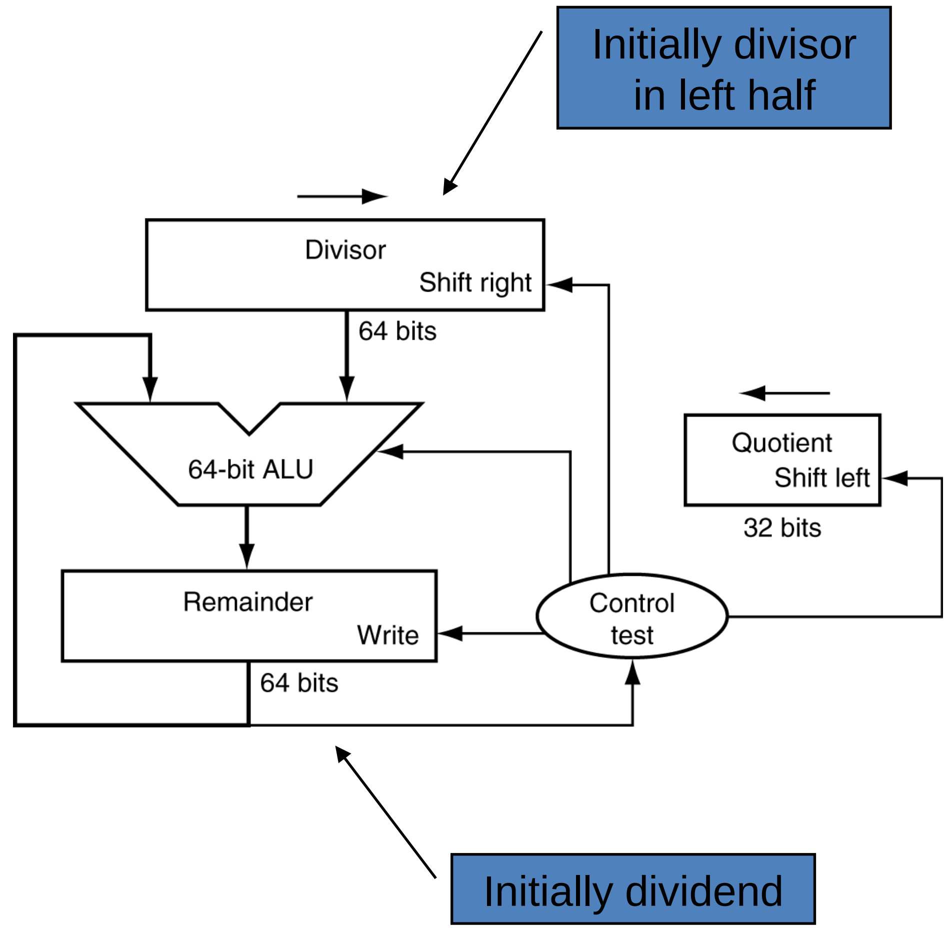
  else  $Q_i=1$, $R'=D$

$R'=R$

**Each row computes one iteration of the division algorithm.**

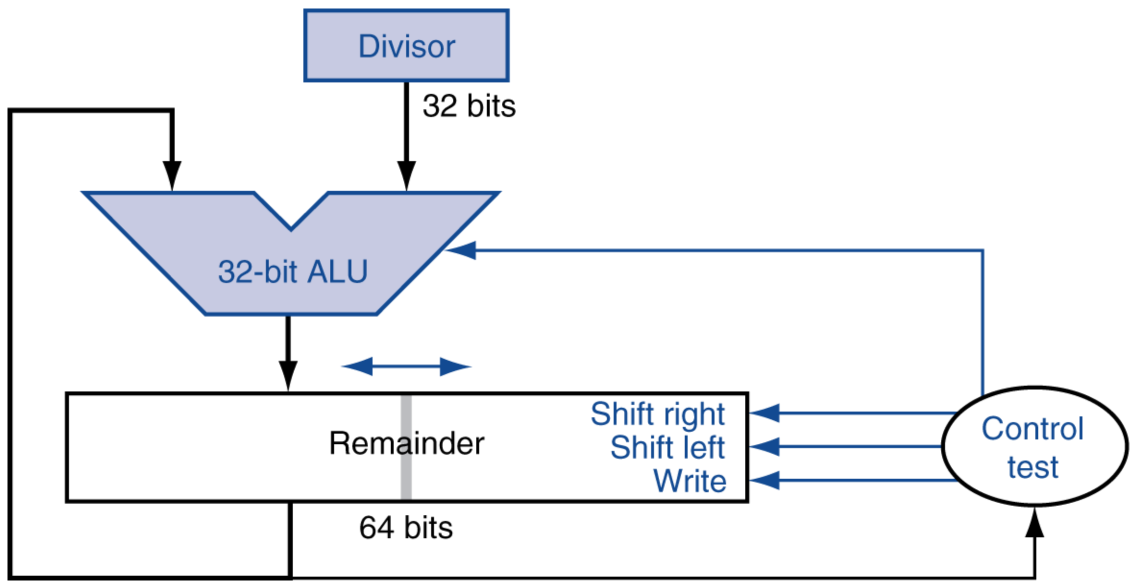
# Sequential division



# Optimized divider

- One cycle per partial-remainder subtraction
- Looks a lot like a multiplier!
  - Same hardware can be used for both



# Faster Division

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