Chapter 3

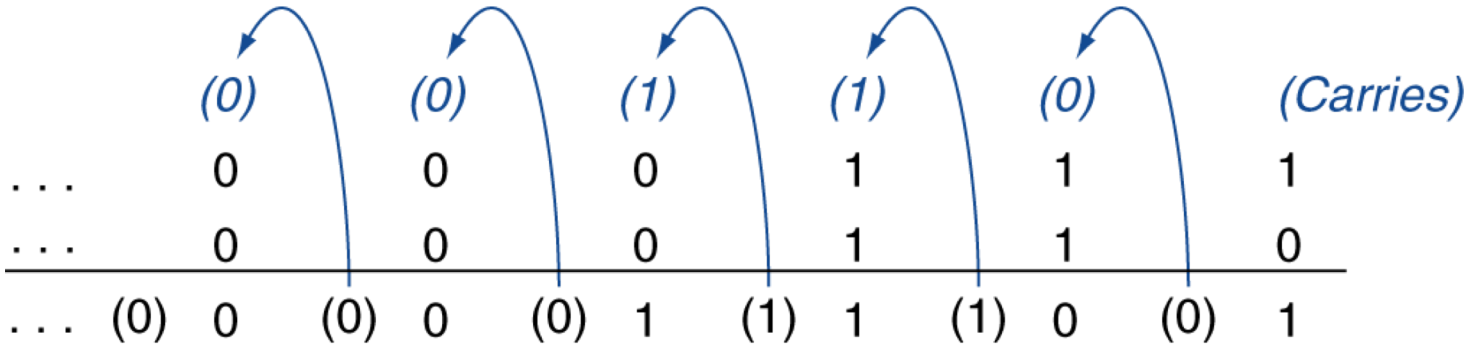
Arithmetic for Computers

Arithmetic for Computers

- Operations on integers
  - Addition and subtraction
  - Multiplication and division
- Floating-point real numbers
  - Representation and operations

Integer Addition/Subtraction

- 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/Subtraction

- For subtraction, add negation of second operand
- Example: 7 – 6 = 7 + (–6)

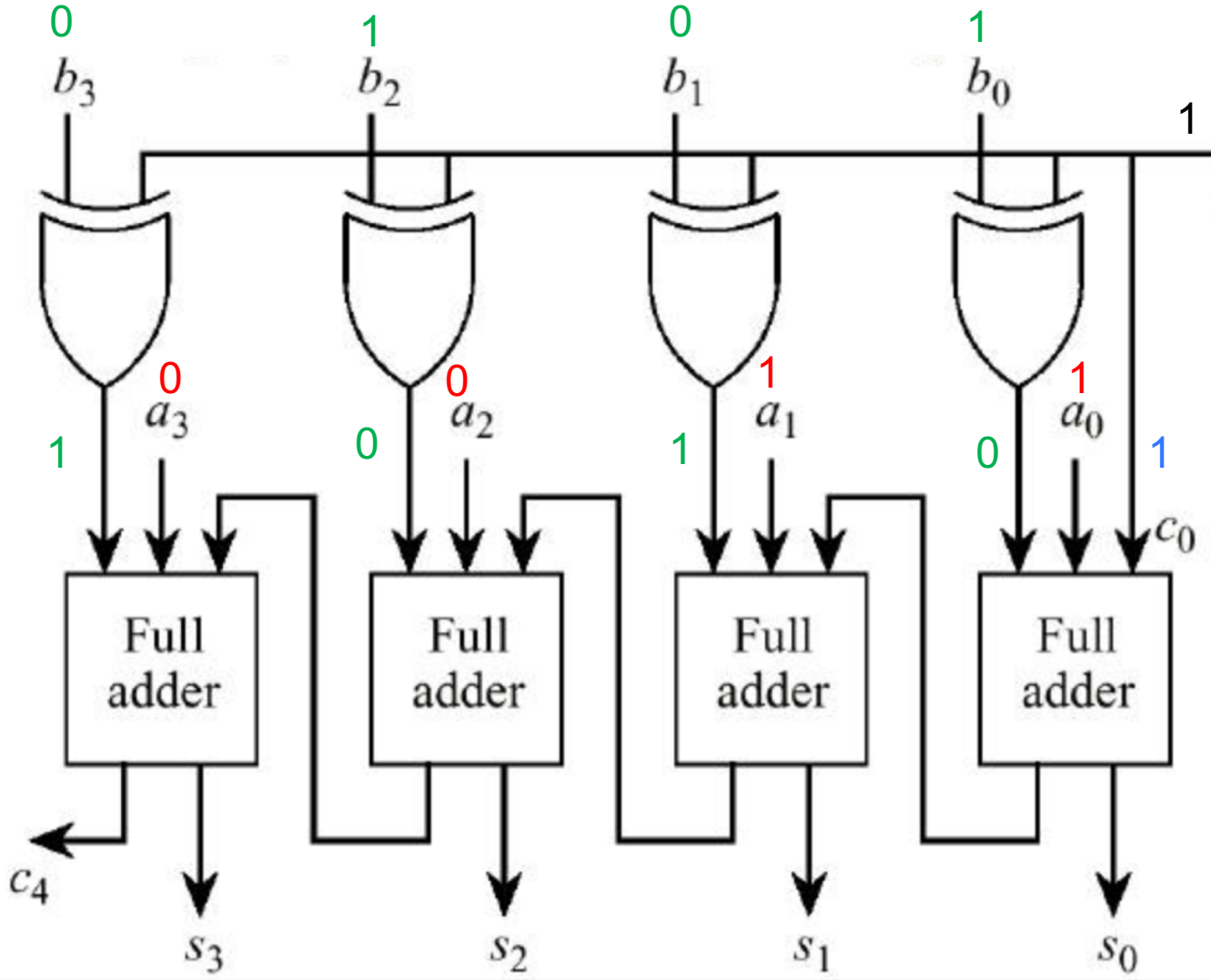
+7:	0000 0000 … 0000 0111
–6:	1111 1111 … 1111 1010
+1:	0000 0000 … 0000 0001

- 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

Overflow Conditions for Addition and Subtraction

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

Integer Add/Sub Circuit



0011 (+3) +
0101 (+5)

0011 (+3) -
0101 (+5)

0011 (+3) +
1011 (-5)

1110 (-2)

Changing the Sign in 2's Complement

Sign+Magnitude:

+5 = 0101

Change 1 bit

-5 = 1101

2's Complement:

+5 = 0101

Invert

1010

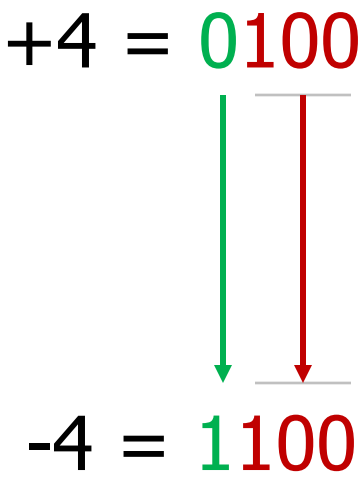
Increment

+1

-5 = 1011

Easier Hand Method (example 1)

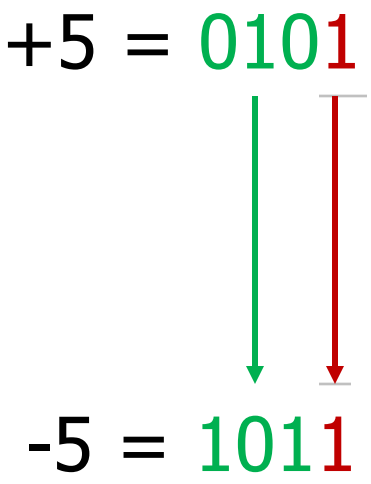
Step 2: Copy the inverse of the remaining bits.



Step 1: Copy the bits from right to left, through and including the first 1.

Easier Hand Method (example 2)

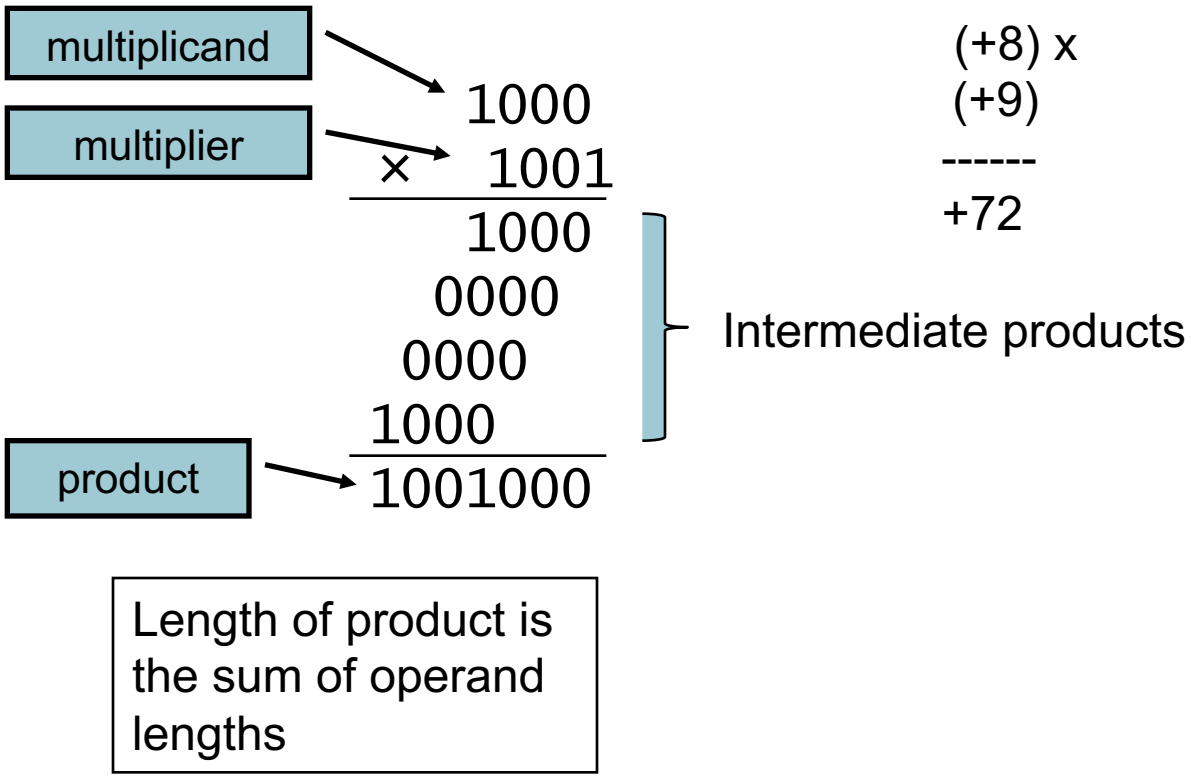
Step 2: Copy the inverse of the remaining bits.



Step 1: Copy the bits from right to left, through and including the first 1.

Integer Multiplication

- Start with long-multiplication approach



Integer Multiplication

- Start with long-multiplication approach

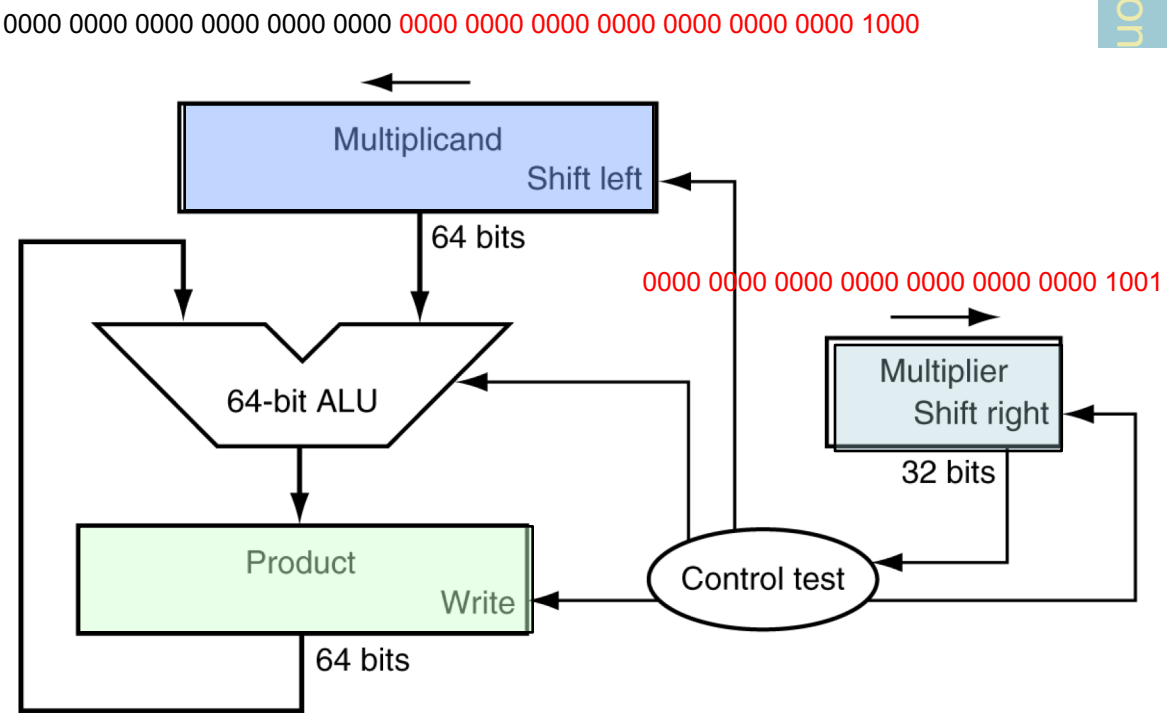
multiplicand

multiplier

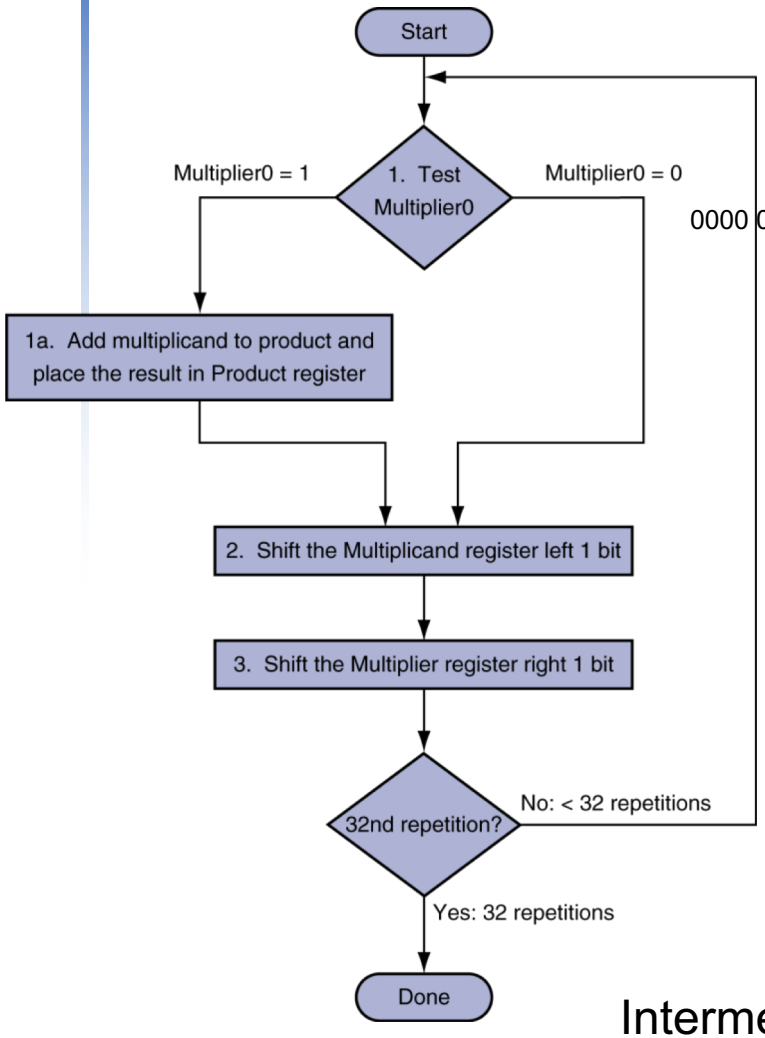
product

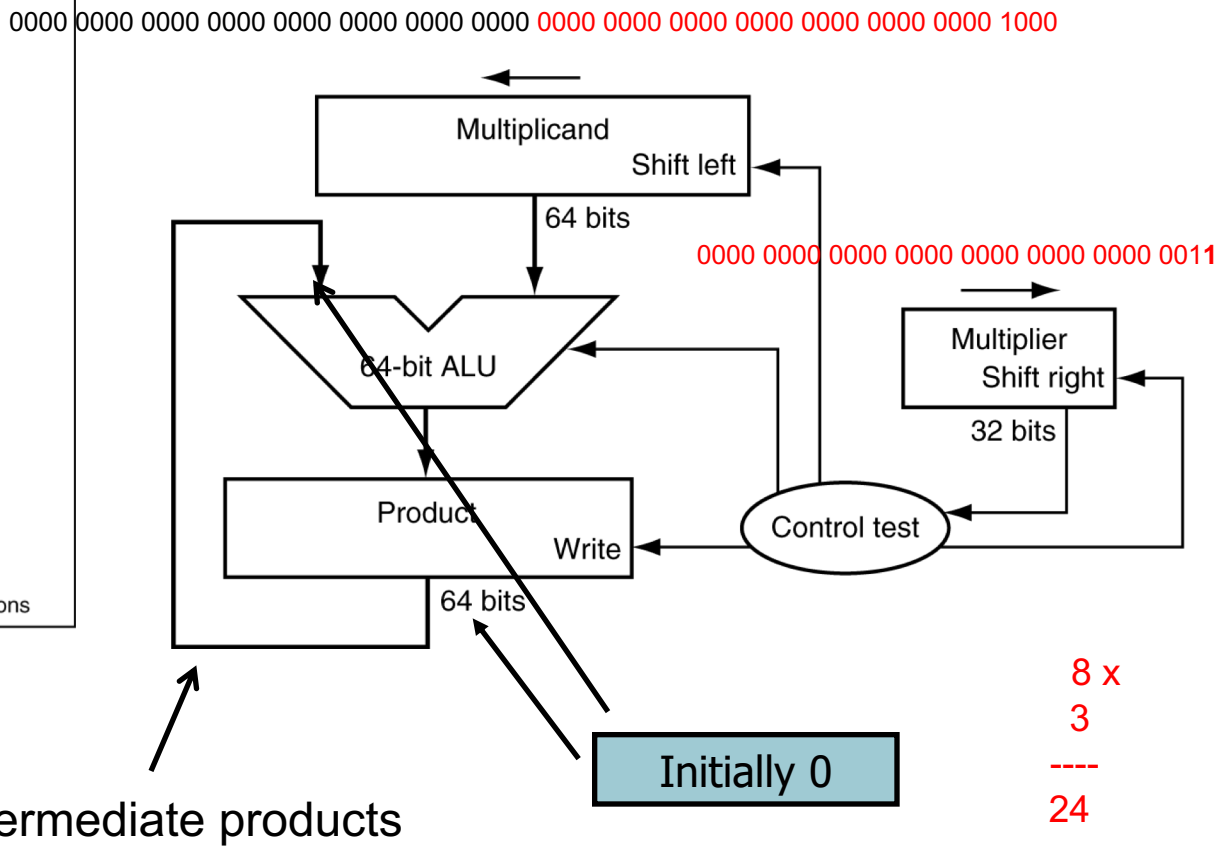
```
  1000
×  1001
------
  1000
 0000
0000
1000
-------
1001000
```

Length of product is the sum of operand lengths

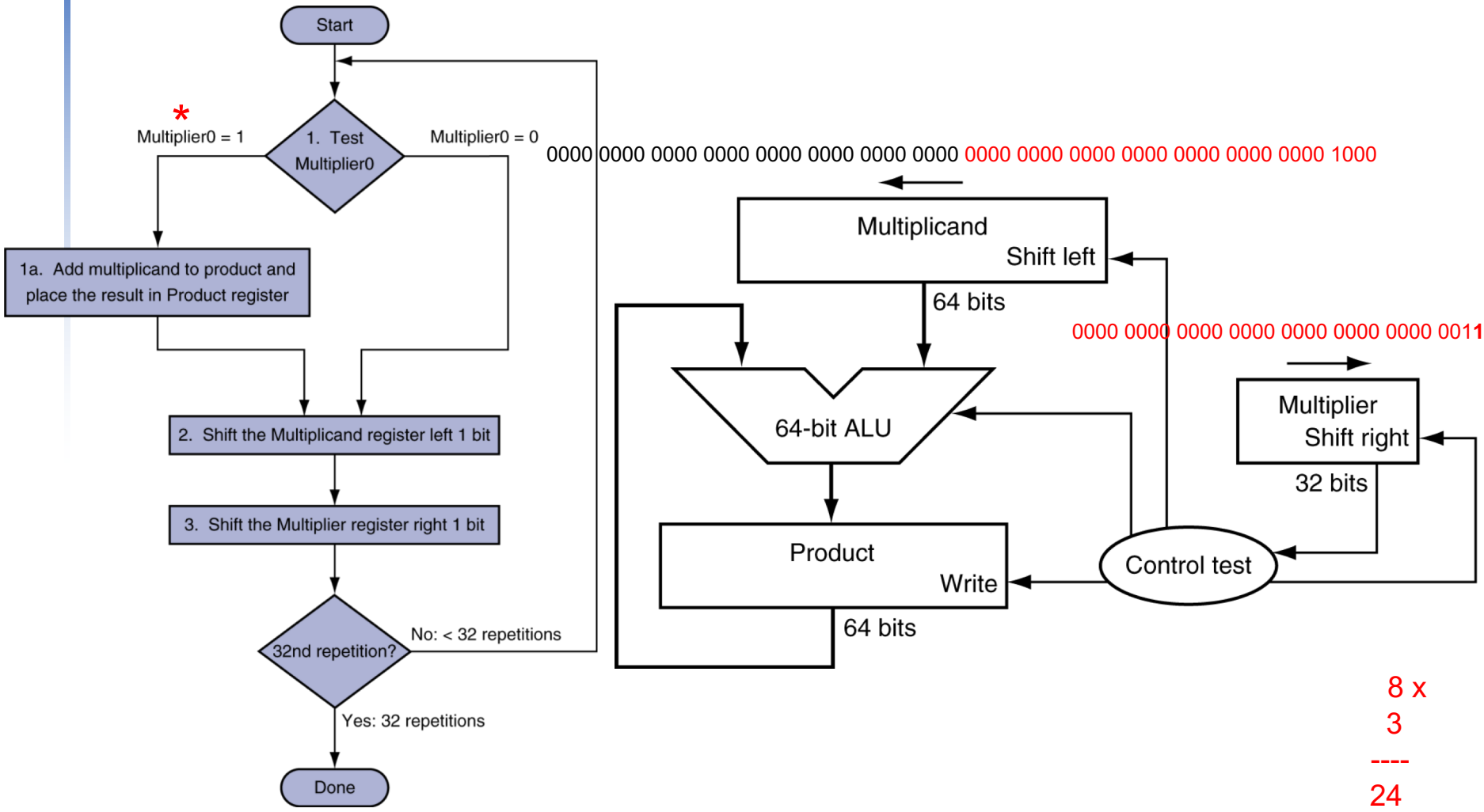


Multiplication Hardware

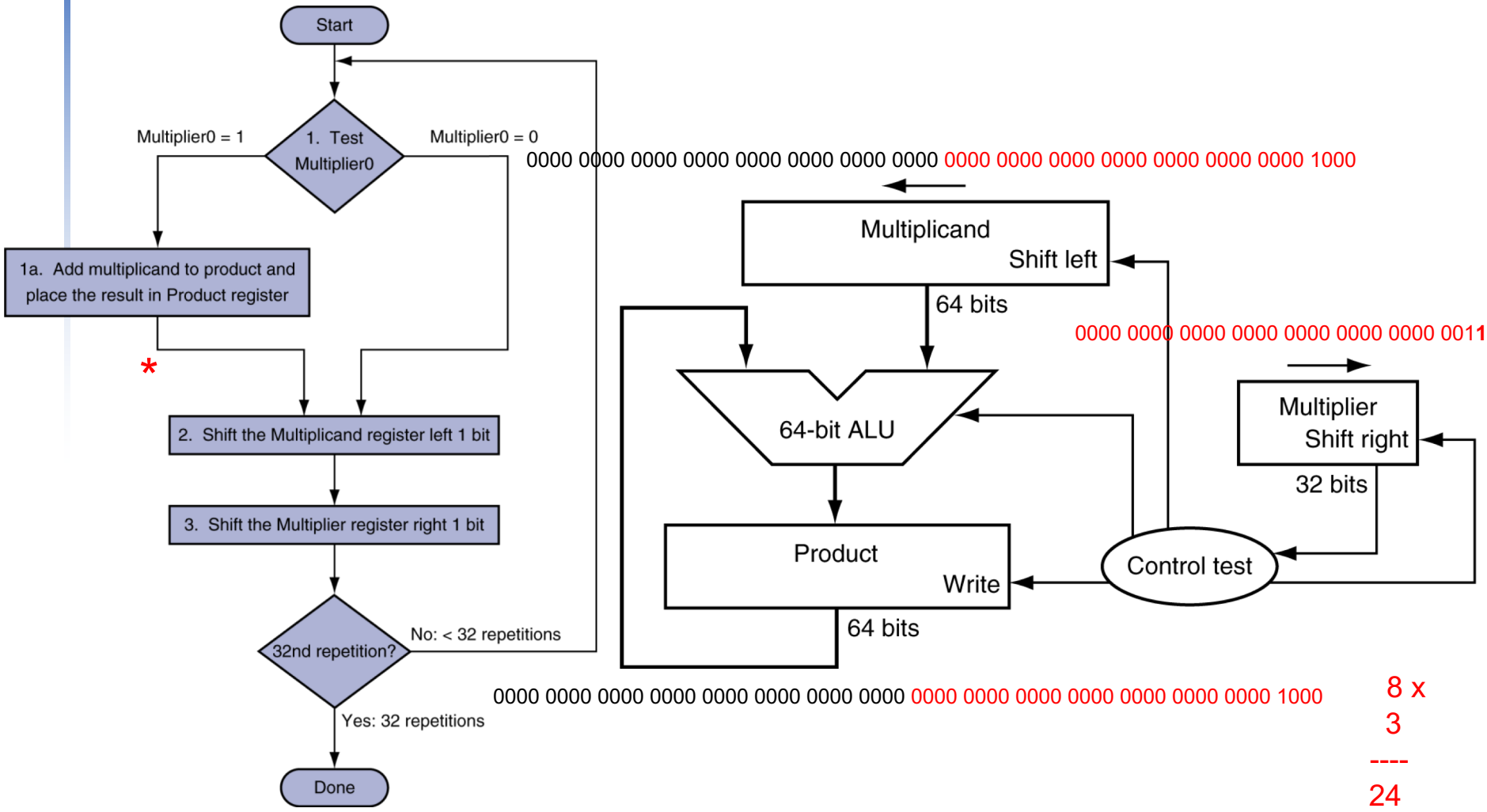




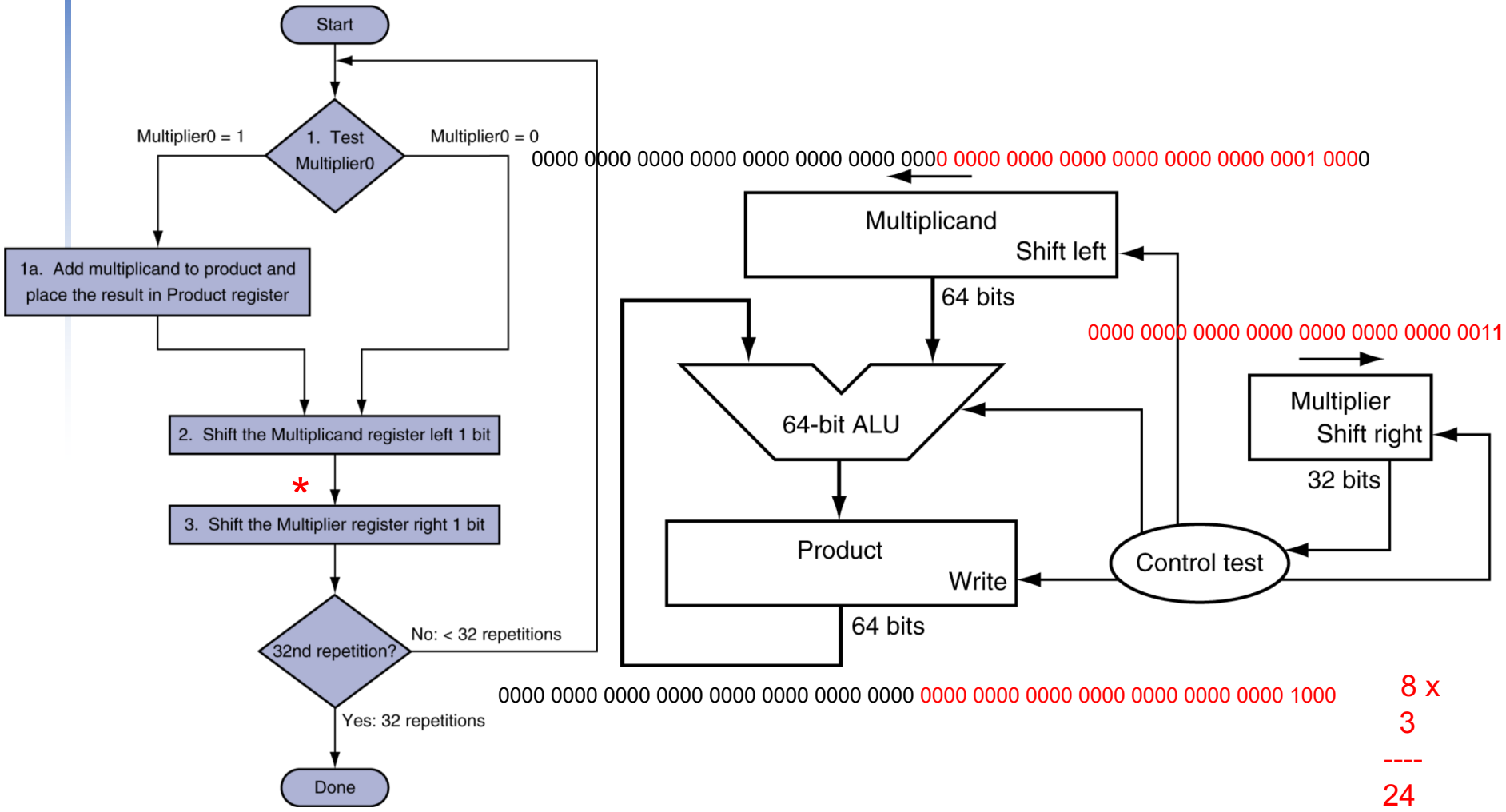
Multiplication Hardware



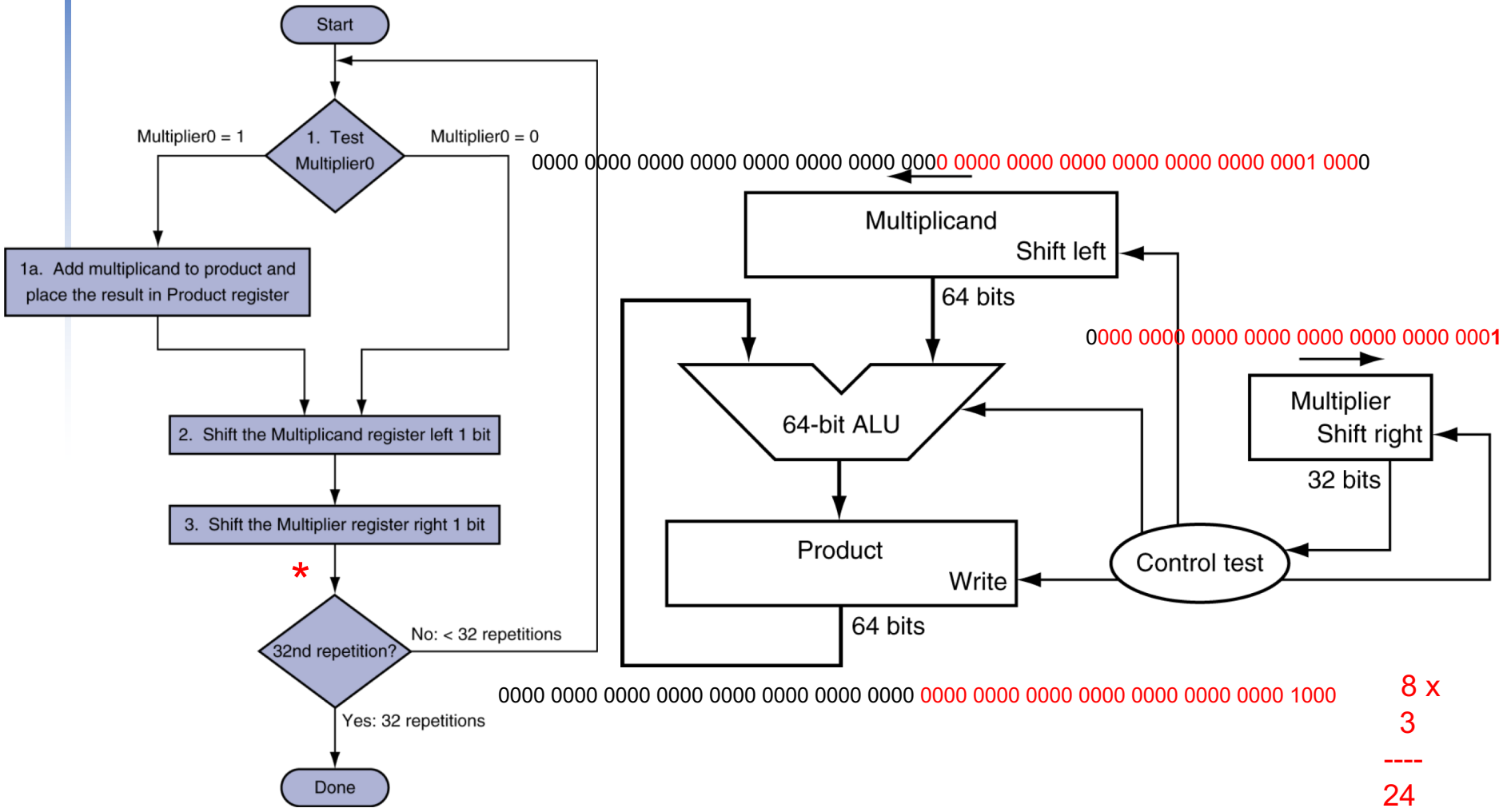
Multiplication Hardware



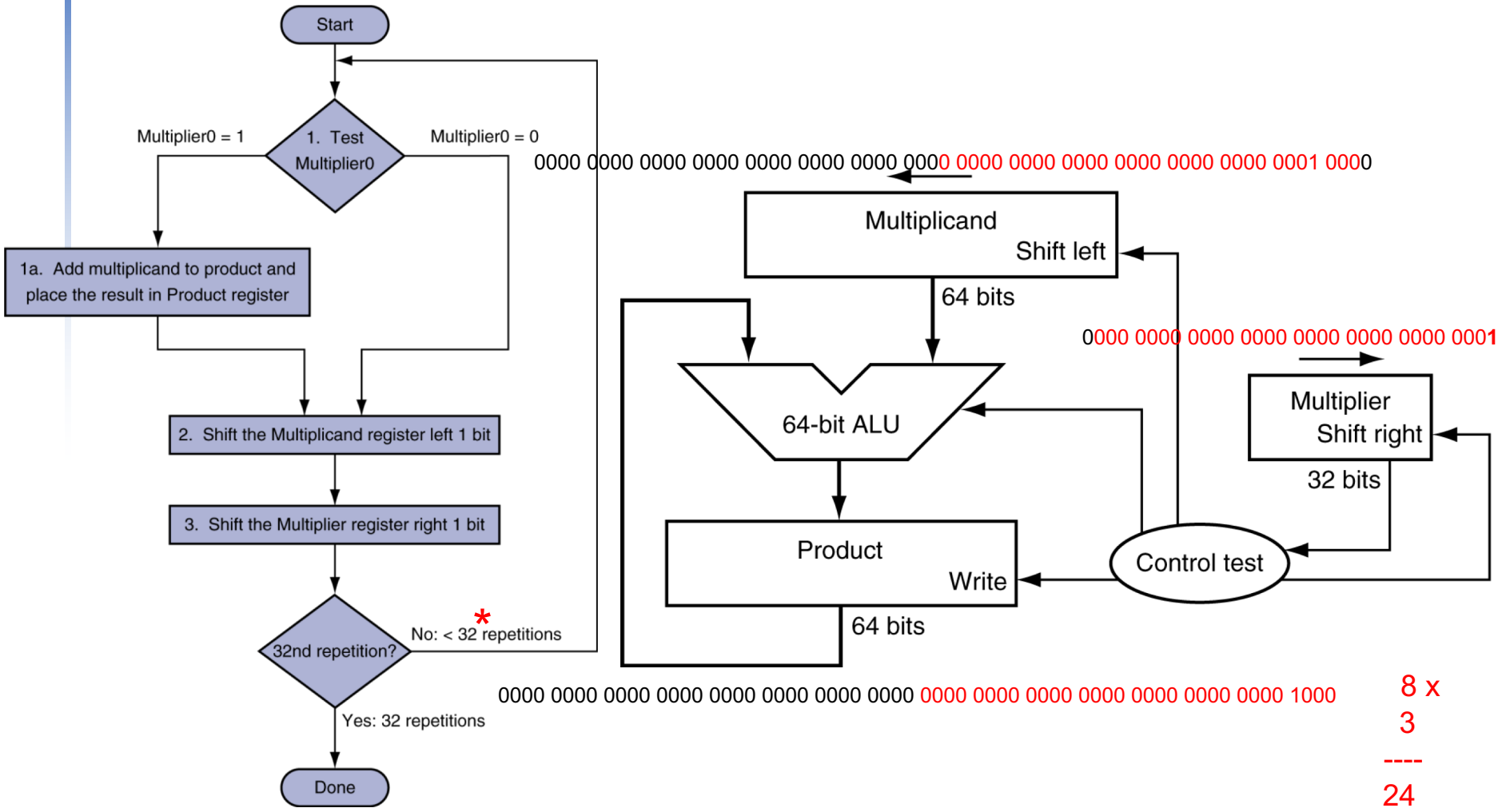
Multiplication Hardware



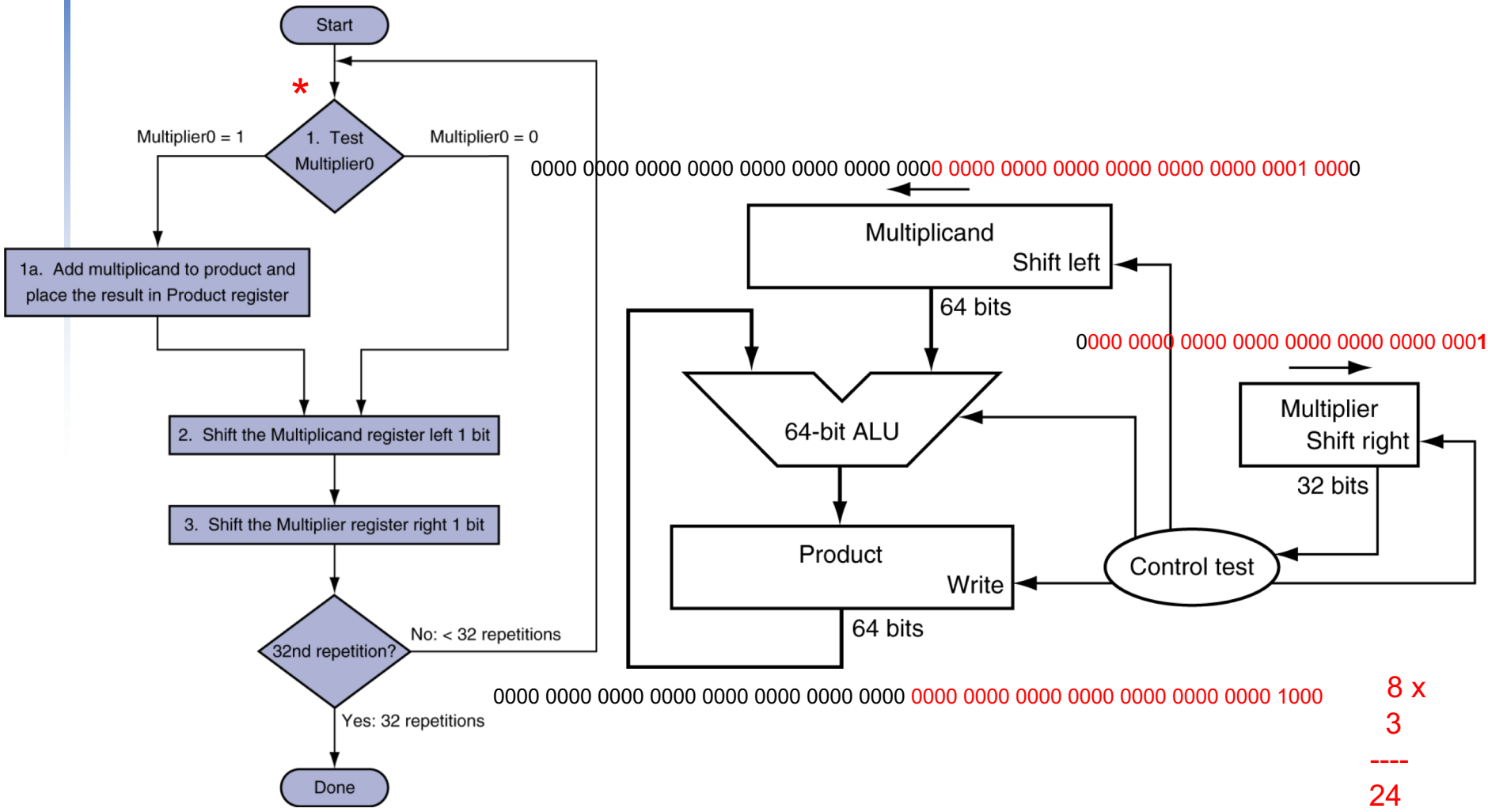
Multiplication Hardware



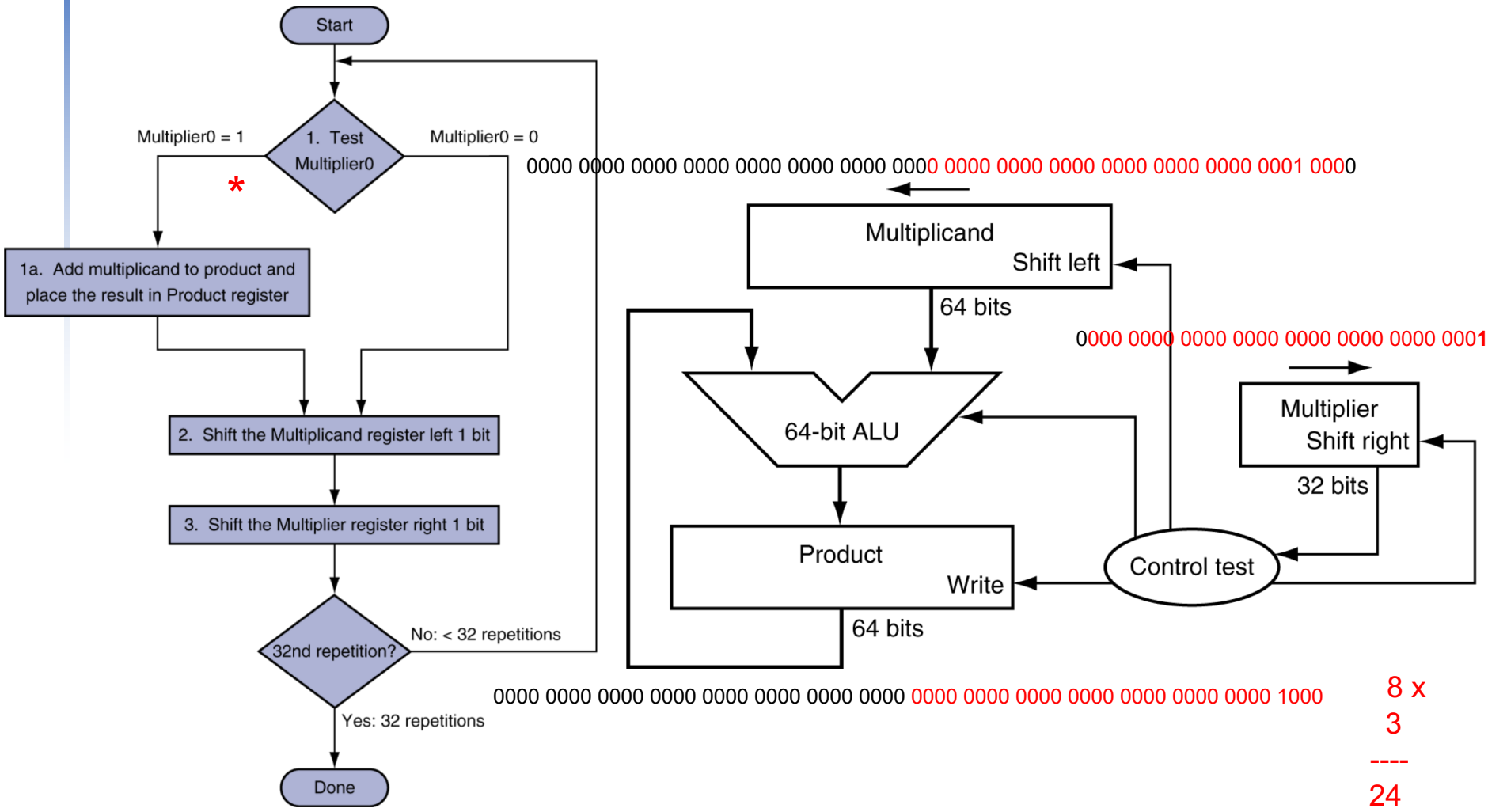
Multiplication Hardware



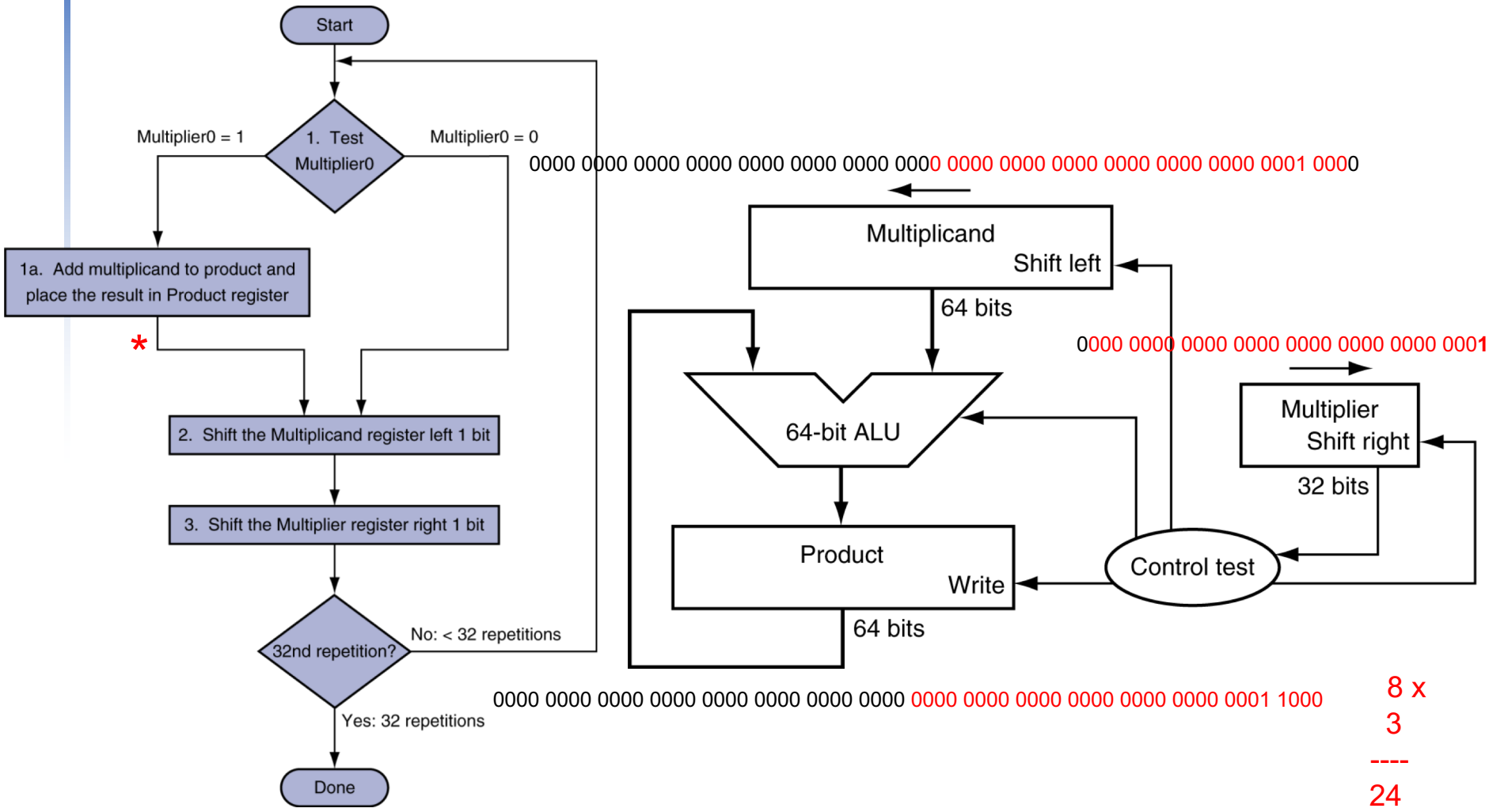
Multiplication Hardware



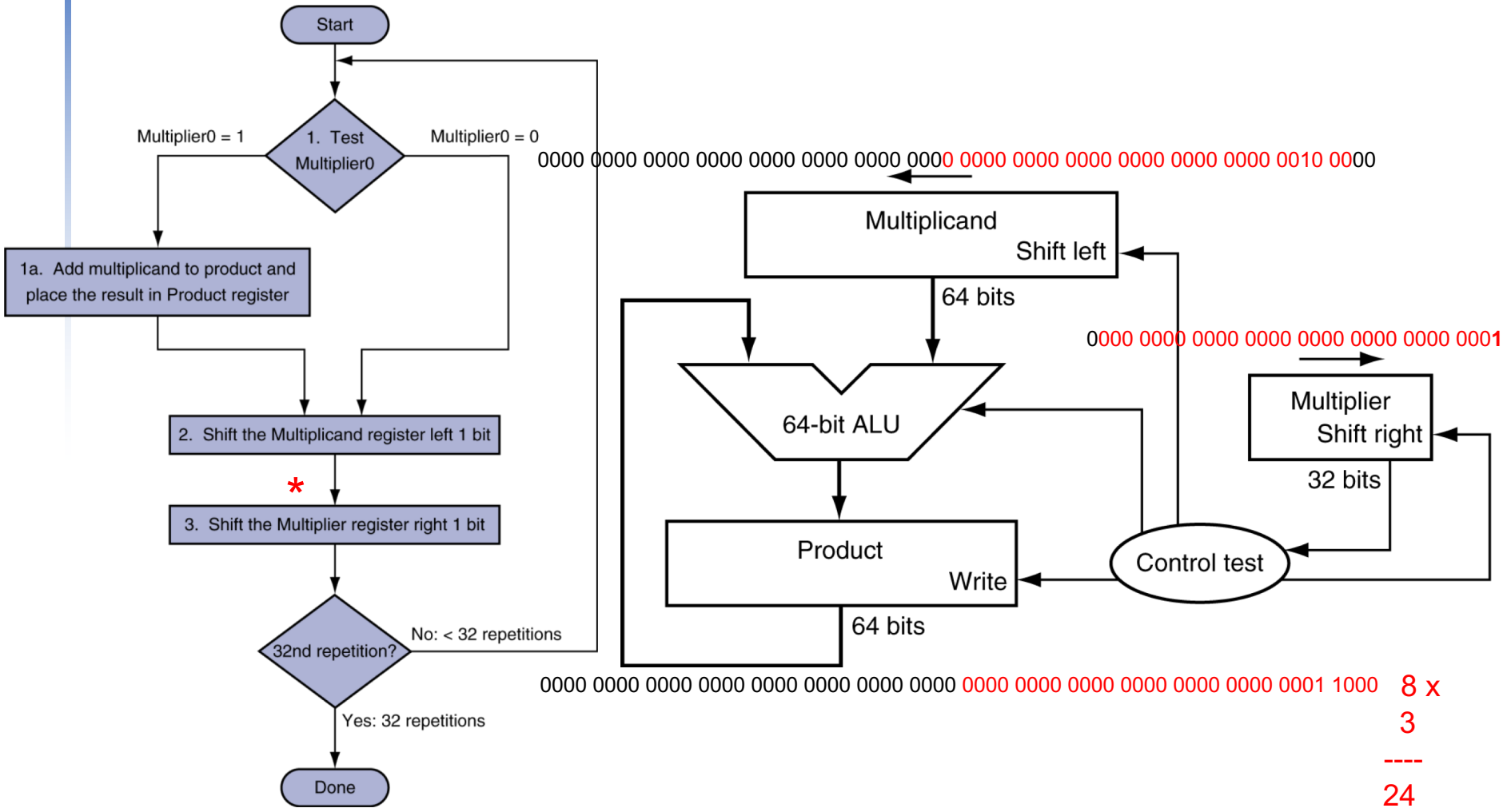
Multiplication Hardware



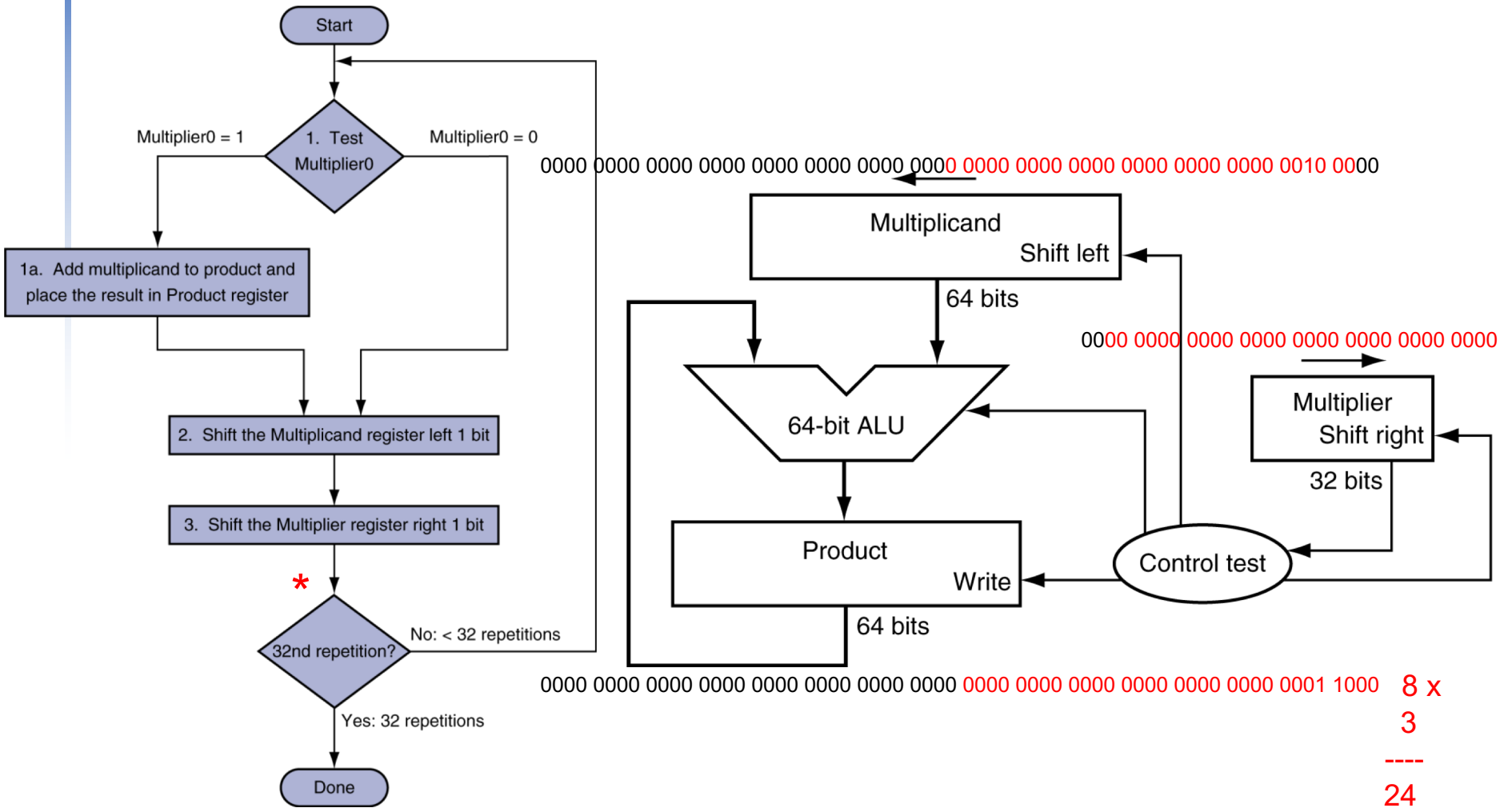
Multiplication Hardware



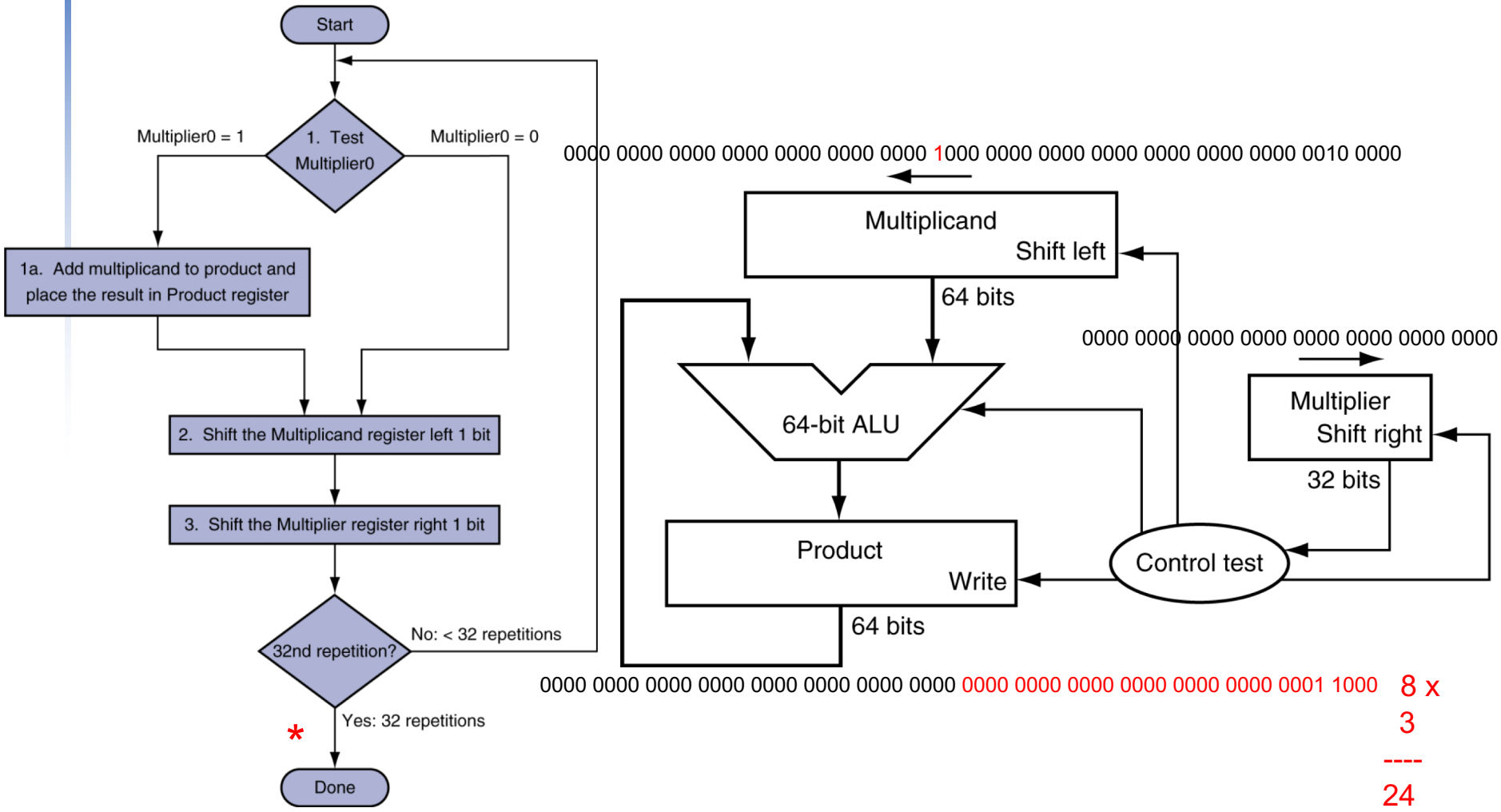
Multiplication Hardware



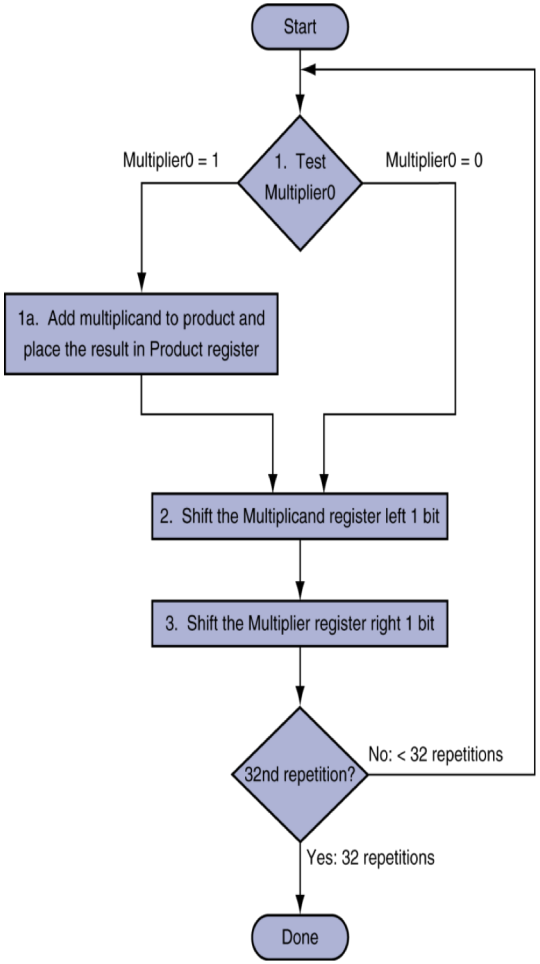
Multiplication Hardware

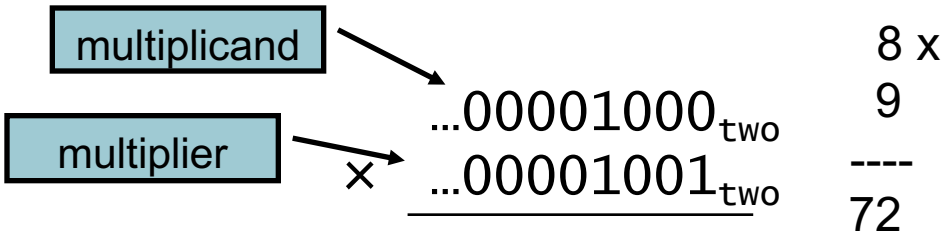


Multiplication Hardware



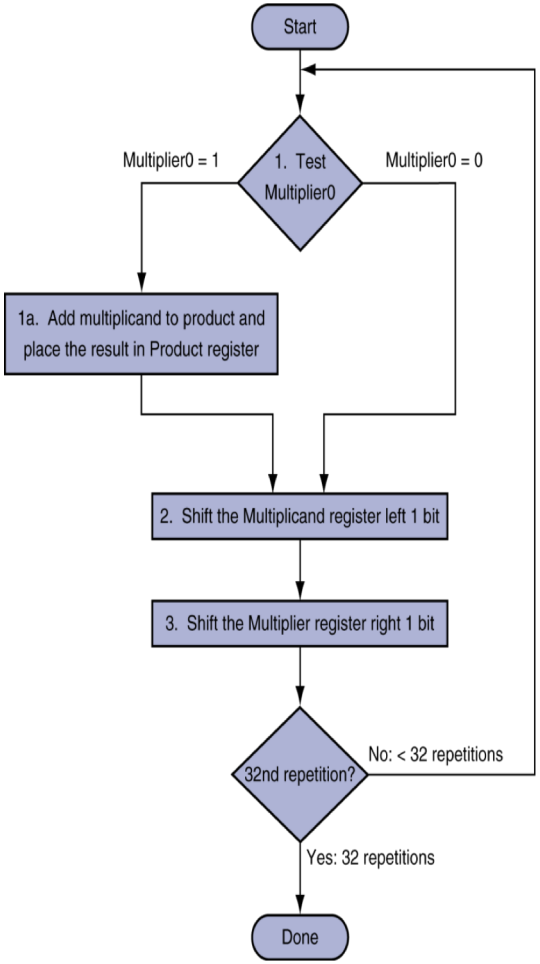
Multiplication Hardware

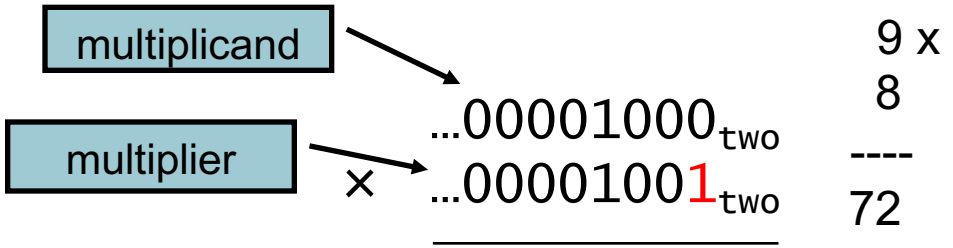




Multiplicand (64 bits)	Product (64 bits)
…0000000000001000	…0000000000000000

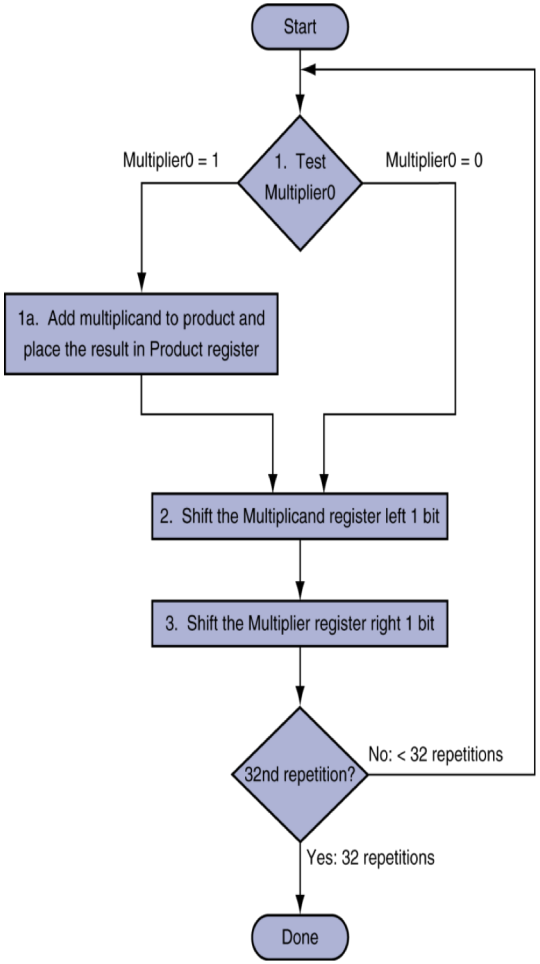
Multiplication Hardware

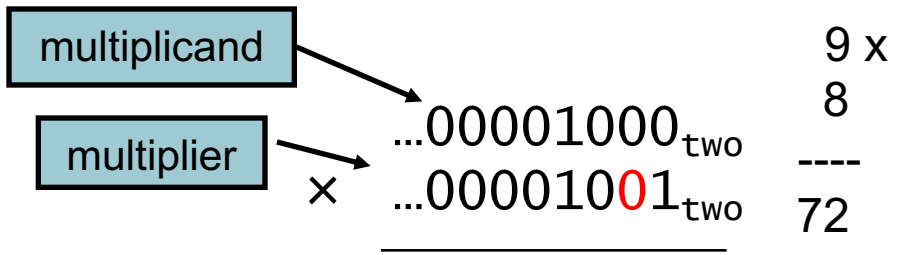




Multiplicand (64 bits)	+	Product (64 bits)
…000000000001000	→	…0000000000000000
…000000000010000		…0000000000001000

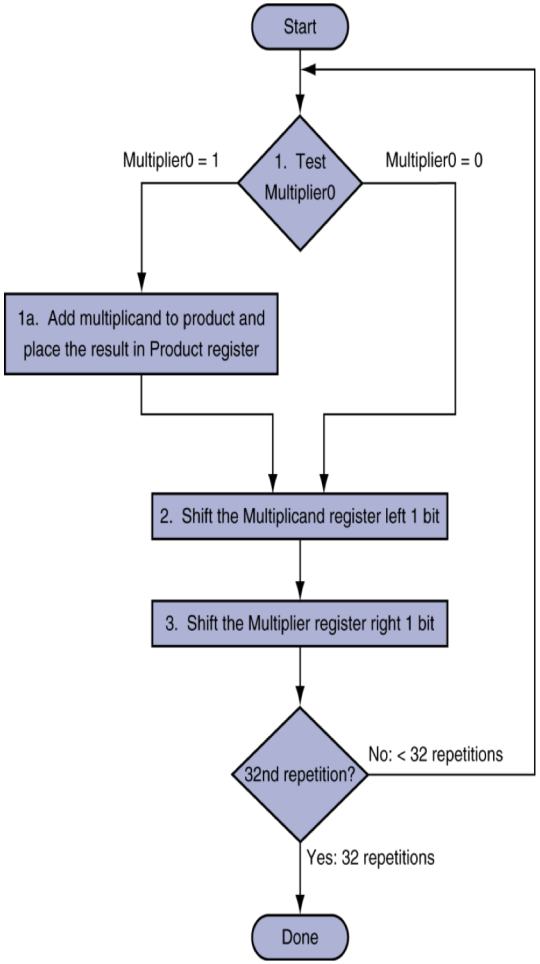
Multiplication Hardware

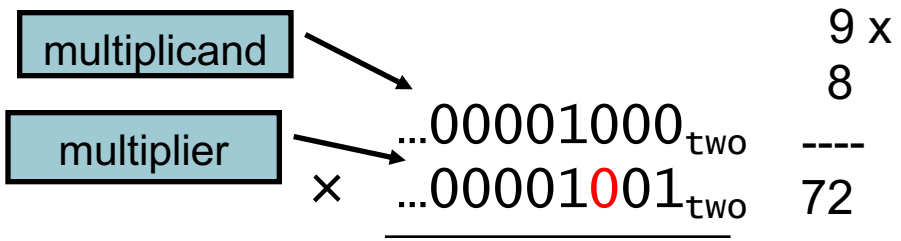




Multiplicand (64 bits)	Product (64 bits)
…000000000001000	…000000000000000
…000000000010000	…000000000001000
…000000000100000	…000000000001000

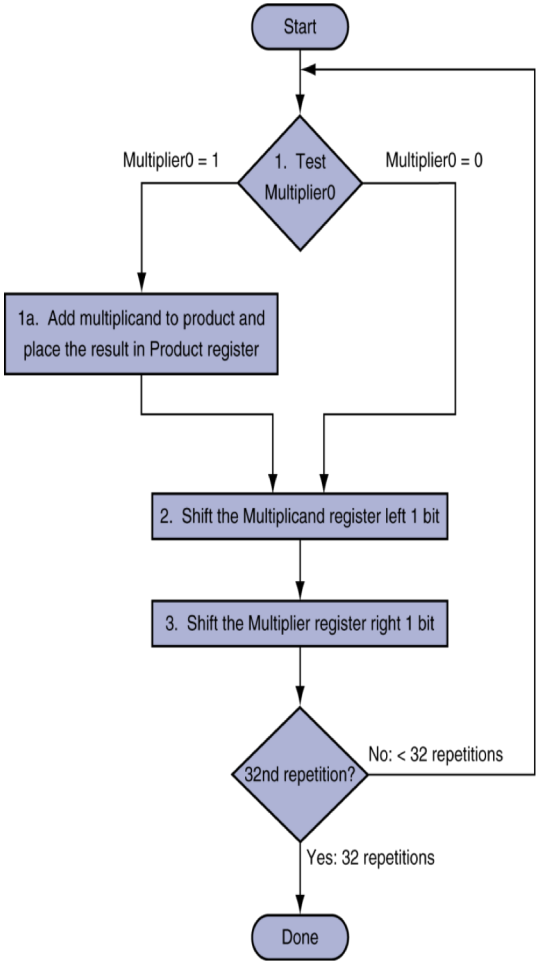
Multiplication Hardware

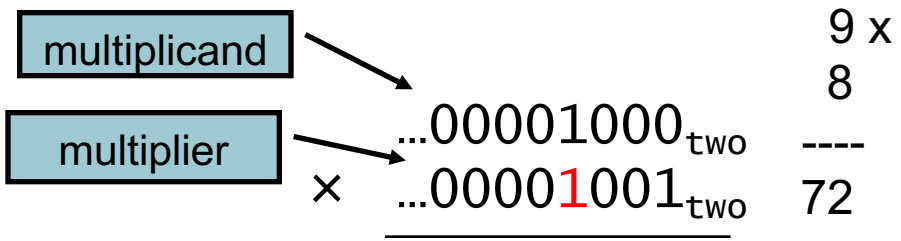




Multiplicand (64 bits)	Product (64 bits)
…0000000000001000	…0000000000000000
…0000000000010000	…0000000000001000
…0000000000100000	…0000000000001000
…0000000001000000	…0000000000001000

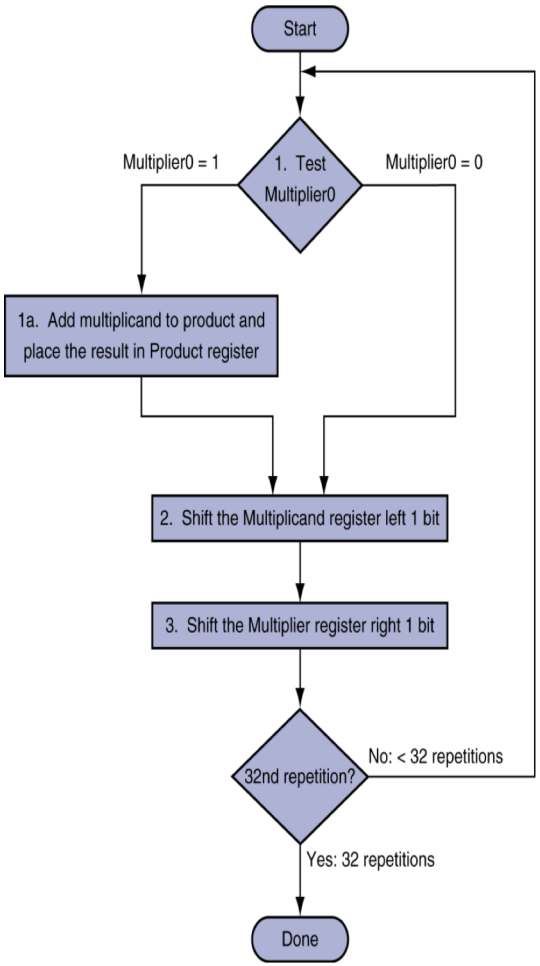
Multiplication Hardware





Multiplicand (64 bits)	Product (64 bits)
…0000000000001000	…0000000000000000
…0000000000010000	…0000000000001000
…0000000000100000	…0000000000001000
…0000000001000000	…0000000000001000

Multiplication Hardware



multiplicand

multiplier

$\ldots00001000_{two}$

$\times \ldots00001001_{two}$

9 x 8 = 72

Multiplicand (64 bits)	Product (64 bits)
…000000000001000	…000000000000000
…000000000010000	…000000000001000
…000000000100000	…000000000001000
…000000001000000	+ …000000000001000
	…000000001001000

LEGv8 Multiplication

- Three multiply instructions:
  - MUL: multiply
    - Two 64-bit registers can be multiplied to produce a 64-bit result: Gives the lower 64 bits of the product
  - SMULH: signed multiply returning high half
    - Gives the upper 64 bits of the 128-bit product, assuming the operands are signed
  - UMULH: unsigned multiply returning high half
    - Gives the upper 64 bits of the 128-bit product, assuming the operands are unsigned

LEGv8 Multiplication

```
MUL Wd, Wn, Wm ; 32-bit general registers

MUL Xd, Xn, Xm ; 64-bit general registers
```

Where:

`Wd`

Is the 32-bit name of the general-purpose destination register.

`Wn`

Is the 32-bit name of the first general-purpose source register holding the multiplicand.

`Wm`

Is the 32-bit name of the second general-purpose source register holding the multiplier.

`Xd`

Is the 64-bit name of the general-purpose destination register.

`Xn`

Is the 64-bit name of the first general-purpose source register holding the multiplicand.

`Xm`

Is the 64-bit name of the second general-purpose source register holding the multiplier.

Operation

`Rd = Rn * Rm`, where `R` is either `W` or `X`.

LEGv8 Multiplication

SMULH

Signed Multiply High.

Syntax

`SMULH Xd, Xn, Xm`

Where:

Xd

Is the 64-bit name of the general-purpose destination register.

Xn

Is the 64-bit name of the first general-purpose source register holding the multiplicand.

Xm

Is the 64-bit name of the second general-purpose source register holding the multiplier.

Operation

Signed Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.

`Xd = bits<127:64> of Xn * Xm.`

LEGv8 Multiplication

UMULH

Unsigned Multiply High.

Syntax

`UMULH Xd, Xn, Xm`

Where:

Xd

Is the 64-bit name of the general-purpose destination register.

Xn

Is the 64-bit name of the first general-purpose source register holding the multiplicand.

Xm

Is the 64-bit name of the second general-purpose source register holding the multiplier.

Operation

Unsigned Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.

```
Xd = bits<127:64> of Xn * Xm.
```

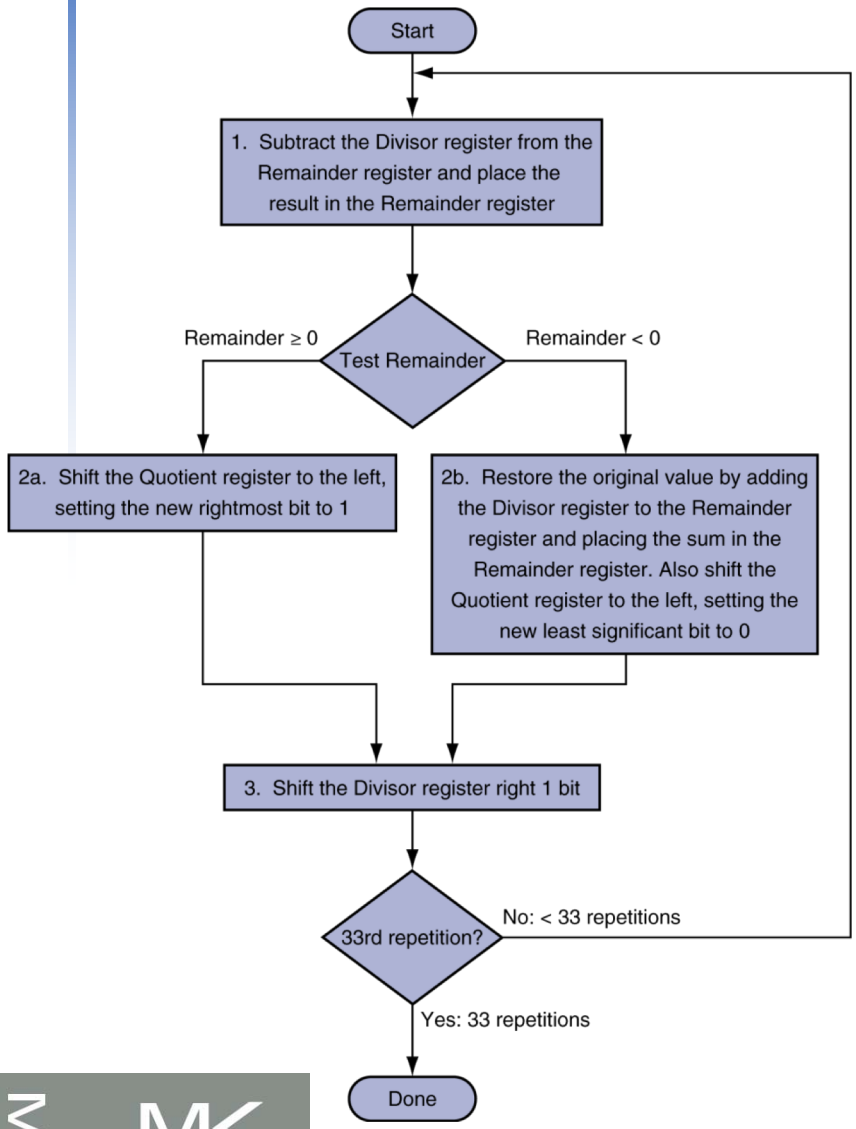
LEGv8 Multiplication

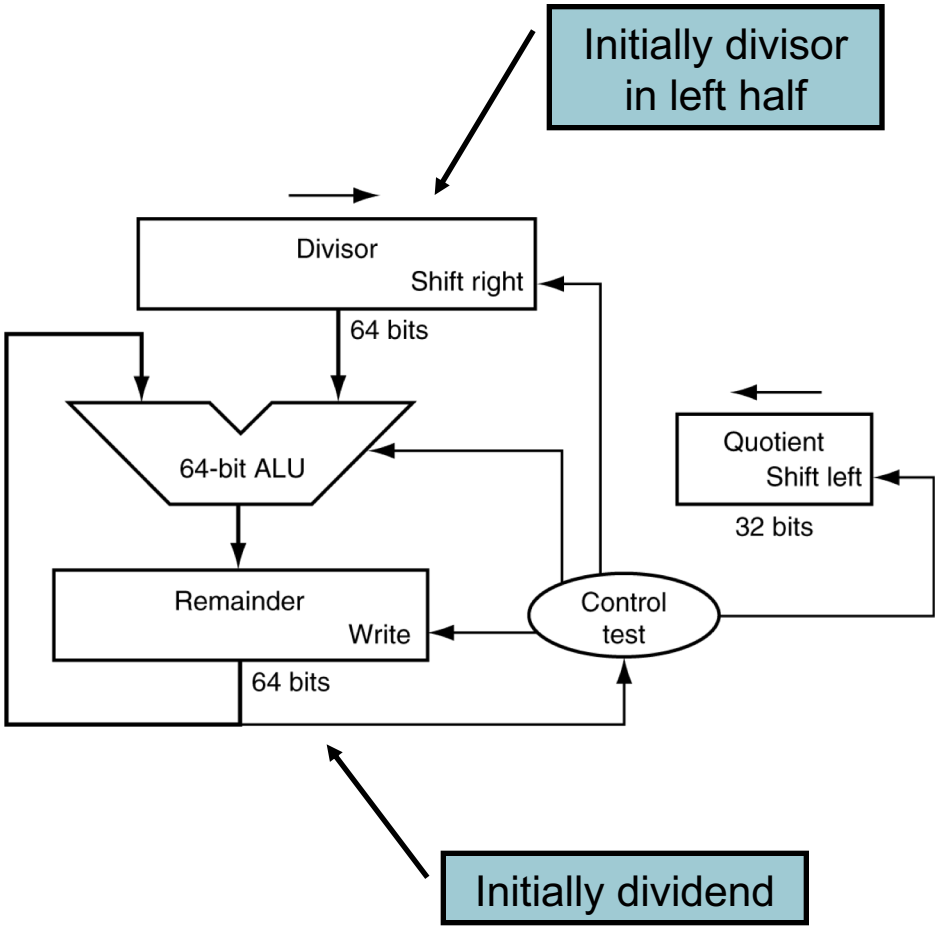
Example:
Real result of the two unsigned numbers
0xFFFF_FFFF_FFFF_FFFF (2^{64} - 1)
multiplied by
0x2 (2)
Result can not be represented by 64 bits and
will overflows to the higher 64 half
Result =
0x0000_0000_0000_0010_0000_0000_99D5_5C00

Assumptions:
X0 = #0xFFFFFFFFFFFFFFFF
X1 = #0x2

```
MUL    X2, X0, X1   // X2 = 0x0000000099D55C00
UMULH  X3, X0, X1   // X3 = 0x0000000000000010
```

Division Hardware





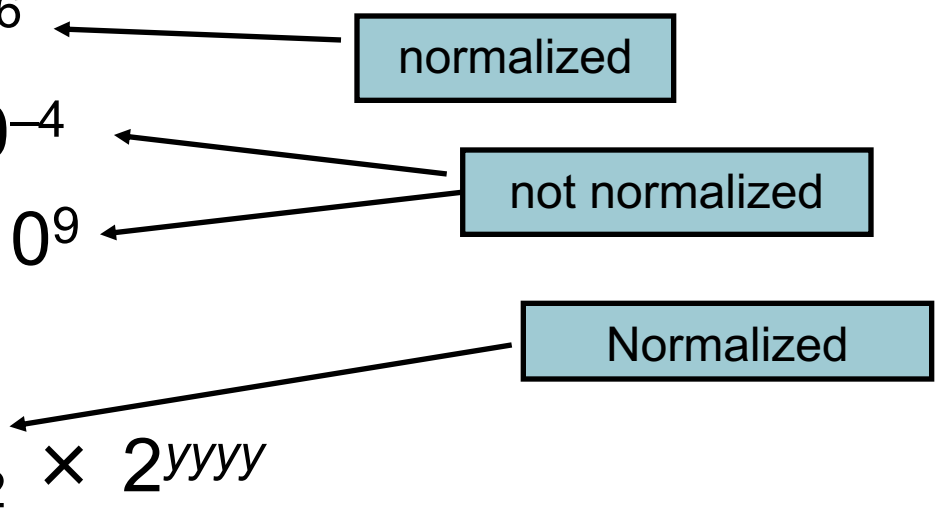
LEGv8 Division

- Two instructions:
  - SDIV (signed divide)
  - UDIV (unsigned divide)

- Examples:
  - UDIV W0, W1, W2 // W0 = W1 / W2 (unsigned, 32-bit divide)
  - SDIV X0, X1, X2 // X0 = X1 / X2 (signed, 64-bit divide)

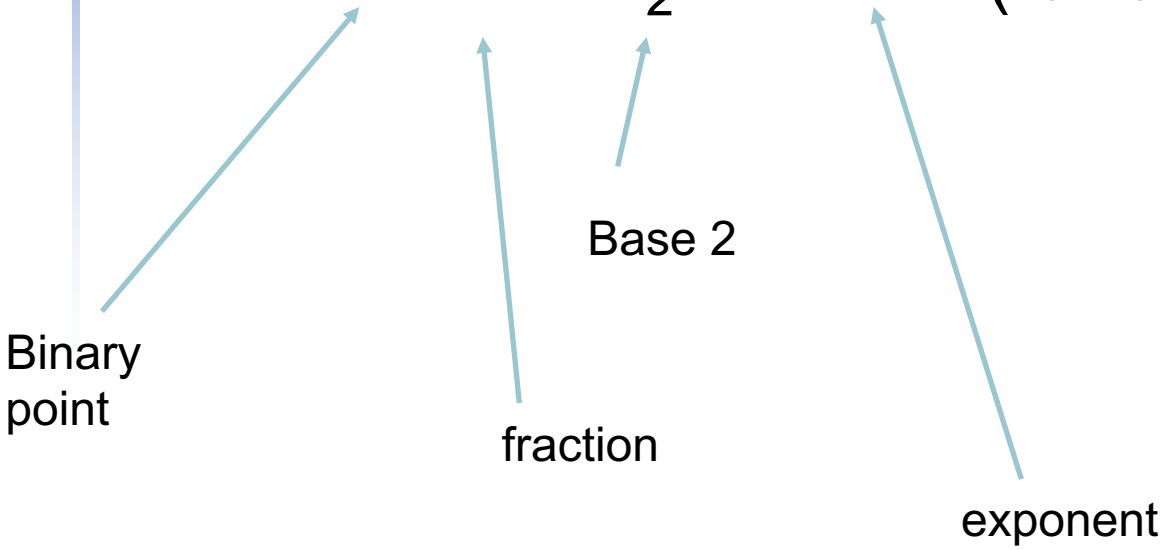
Floating Point (recall)

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - -2.34×10^{56}
  - $+0.002 \times 10^{-4}$
  - $+987.02 \times 10^{9}$
- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types in C: `float` (single-precision FP) and `double` (double-precision FP)



Floating Point (recall)

- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$ (for example, $1.0_2 \times 2^{-1}$)



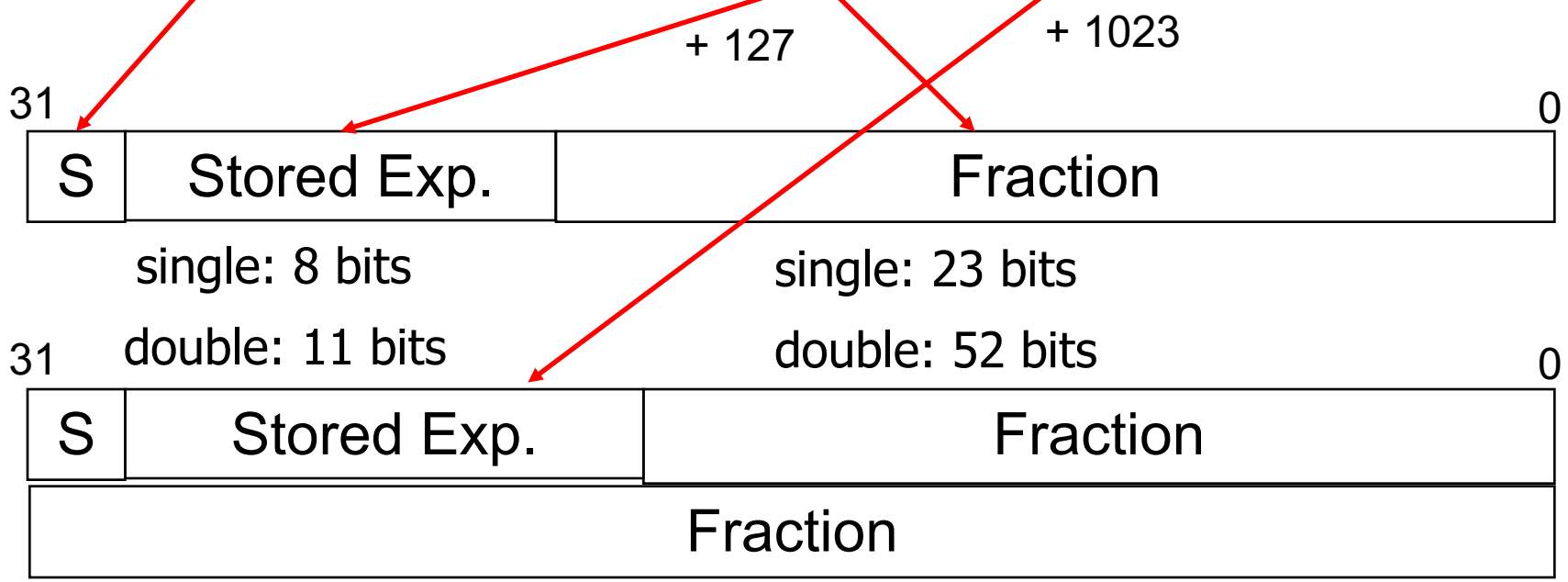
A tradeoff between precision and range.

Floating Point Standard (Recall)

- Defined by **IEEE Standard 754** -1985
- Developed in response to divergence of representations
  - Portability issues for scientific code
- Now almost universally adopted
- Two representations
  - Single precision (32-bit representation)
  - Double precision (64-bit representation)

IEEE Floating-Point Format (recall)

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Actual Exponent})}$$



Floating-Point Example

What is the IEEE 754 single precision representation of the binary number 111111.01 x 2^0?

Normalize the binary number first:
=> move binary point 5 to the left 1.1111101 x 2^5 (sign positive => S = 0)
Exponent (stored) = Actual Exponent (5) + bias
=> Stored Exp. = 5 + 127 = 132

Final bit pattern:
0 1000 0100 1111 1010 0000 0000 0000 000

Floating-Point Example

What is the IEEE 754 single precision representation of the decimal number +97.25?

Recall: Decimal to Binary Conversion (Whole Part: Repeated Division)

Convert 97_{10} to base 2

97 ÷ 2 ➔	quotient = 48,	remainder = 1 (LSB)
48 ÷ 2 ➔	quotient = 24,	remainder = 0.
24 ÷ 2 ➔	quotient = 12,	remainder = 0.
12 ÷ 2 ➔	quotient = 6,	remainder = 0.
6 ÷ 2 ➔	quotient = 3,	remainder = 0.
3 ÷ 2 ➔	quotient = 1,	remainder = 1.
1 ÷ 2 ➔	quotient = 0 (Stop)	remainder = 1 (MSB)

Result = $_{2}$

Recall: Decimal to Binary Conversion
(Fractional Part: Repeated Multiplication)

Example: 0.25_{10}

$.25 \times 2 \rightarrow 0.5$ (fractional part = .5, whole part = 0)

$.5 \times 2 \rightarrow 1.0$ (fractional part = .0 (stop), whole part = 1)

.0

.01

Result = $.01_2$

Floating-Point Example

What is the IEEE 754 single precision representation of the decimal number 97.25?

01100001.01_2

$01100001.01_2 \times 2^0$

$01.10000101_2 \times 2^6$

Final bit pattern:

0 1000 0101 1000 0101 0000 0000 0000 000

More Floating-Point Examples

	S(1)	Exp+127(8)	(implied 1).Significand(23)
2.000	0	10000000	(1).00000000000000000000000
1.000	0	01111111	(1).00000000000000000000000
0.750	0	01111110	(1).10000000000000000000000
0.500	0	01111110	(1).00000000000000000000000
0.000	0	00000000	(0).00000000000000000000000
-0.500	1	01111110	(1).00000000000000000000000
-0.750	1	01111110	(1).10000000000000000000000
-1.000	1	01111111	(1).00000000000000000000000
-2.000	1	10000000	(1).00000000000000000000000

More Floating-Point Examples

Representing decimal 2.0 in IEEE 754 single-precision format:

```
S(1)   Exp+127(8u)(implied 1).Significand(23)

0      10000000      (1).00000000000000000000000
```

2.0_{10} => 10_2 (a +ve number so S = 0)
Then normalize it ➔ 1.0×2^1
Actual exponent = 1 so the stored exponent in single precision is calculated as:
$1+127 = 128_{10}$ or 10000000_2

More Floating-Point Examples

Representing decimal 0.75 in IEEE 754 single-precision format:

```
      S(1)   Exp+127(8u)(implied 1).Significand(23)

        0     01111110     (1).10000000000000000000000
```

0.75 => 0.11_2 (a +ve number so S = 0)
Normalize it ➔ 1.1×2^{-1}
Actual exponent = -1 so the stored exponent in single precision is calculated as:
$-1+127 = 126_{10}$ or 01111110_2

More Floating-Point Examples

```
       S(1)   Exp+127(8u)(implied 1).Significand(23)

-0.500 1      01111110     (1).00000000000000000000000
```

-0.50_{10} -> -0.1_2 (a -ve number so S = 1)
Normalize it ➔ -1.0×2^{-1}
Actual exponent = -1 so the stored exponent in single precision is calculated as:
$-1+127 = 126_{10}$ or 01111110_2

Single-precision Floating-point Representation of Zero and One

```
      S(1)   Exp+127(8)   (implied 1).Significand(23)

 0.000 0     00000000     (1).00000000000000000000000
```

```
Stored exponent = 127 (bias) + Actual exponent
0 = 127 + Actual Exponent
Actual Exponent = -127
```

1.0×2^{-127} does not represent 1.

Another question that comes to mind is this:

If we always have an implied 1 how do we represent a 0 in single-precision IEEE 754 format?

Single-precision Floating-point Representation of Zero and One

```
      S(1)   Exp+127(8)   (implied 0).Significand(23)

0.000 0      00000000     (0).00000000000000000000000
```

Implied 0, if and only if, both stored exponent and the fractional part are zero.

```
      S(1)   Exp+127(8)   (implied 1).Significand(23)

0.000 0      01111111     (1).00000000000000000000000
```

1.000_{10} -> 0001_2 then normalize it as 1.0×2^0
Actual exponent = 0 so the stored storage in single precision becomes 0+127 = 127 or 01111111

FP Instructions in LEGv8

- Separate FP registers
  - 32 single-precision: S0, …, S31
  - 32 double-precision: D0, …, D31
- FP instructions operate only on FP registers
  - Programs generally don’t do integer ops on FP data, or vice versa
- FP load and store instructions
  - `LDURS, LDURD`
  - `STURS, STURD`

FP Instructions in LEGv8

- Single-precision floating-point arithmetic
  - FADDS, FSUBS, FMULS, FDIVS
    - e.g., `FADDS S2, S4, S6`
- Double-precision arithmetic
  - FADDD, FSUBD, FMULD, FDIVD
    - e.g., `FADDD D2, D4, D6`
- Single- and double-precision comparison
  - FCMPS, FCMPD
  - Sets or clears FP condition-code bits
- Branch on FP condition code true or false
  - `B.cond`

FP Example: °F to °C

- C code:

```
float f2c (float fahr) {
  return ((5.0/9.0)*(fahr - 32.0));
}
```

  - fahr in S0, result in S0, and the starting address of constants in memory is in X27

- Compiled LEGv8 code:

```
f2c:
  LDURS S16, [X27,const5]   // S16 = 5.0 (5.0 in memory)
  LDURS S18, [X27,const9]   // S18 = 9.0 (9.0 in memory)
  FDIVS S16, S16, S18       // S16 = 5.0 / 9.0
  LDURS S18, [X27,const32]  // S18 = 32.0
  FSUBS S18, S0, S18        // S18 = fahr - 32.0
  FMULS S0, S16, S18        // S0 = (5/9)*(fahr - 32.0)
  BR LR                     // return
```

Concluding Remarks

- Bits have no inherent meaning
  - Interpretation depends on the instructions applied

- Computer representations of numbers
  - Finite range and precision
  - Need to account for this in programs

Concluding Remarks

- ISAs support arithmetic
  - Signed and unsigned integers
  - Floating-point approximation to reals
- Bounded range and precision
  - Operations can overflow