Arithmetic (Multiplication and Division)

Chapter 3

Multiplication

- Simple multiplication routine:
  - adding: a *b = a + a + ... a , b times
- Longhand multiplication

```
 Multiplicand * Multiplier
        6000₁₀
        2002₁₀
    ----------*
       12000
       0000
      0000
    12000
    ----------+
    12012000₁₀
```

- Product has more digits than multiplicand or multiplier -> Overflow

Multiplication

- Basic functions:
  - Multiplication of the multiplicand with the last digit of the multiplier with the next digit,...
  - Adding up the partial products
- Simplification:
  - Multiplication with numbers 0...9
  - Addition
- Further simplification in binary system:
  - Shift
  - Either add multiplier or add zero.

Multiplication

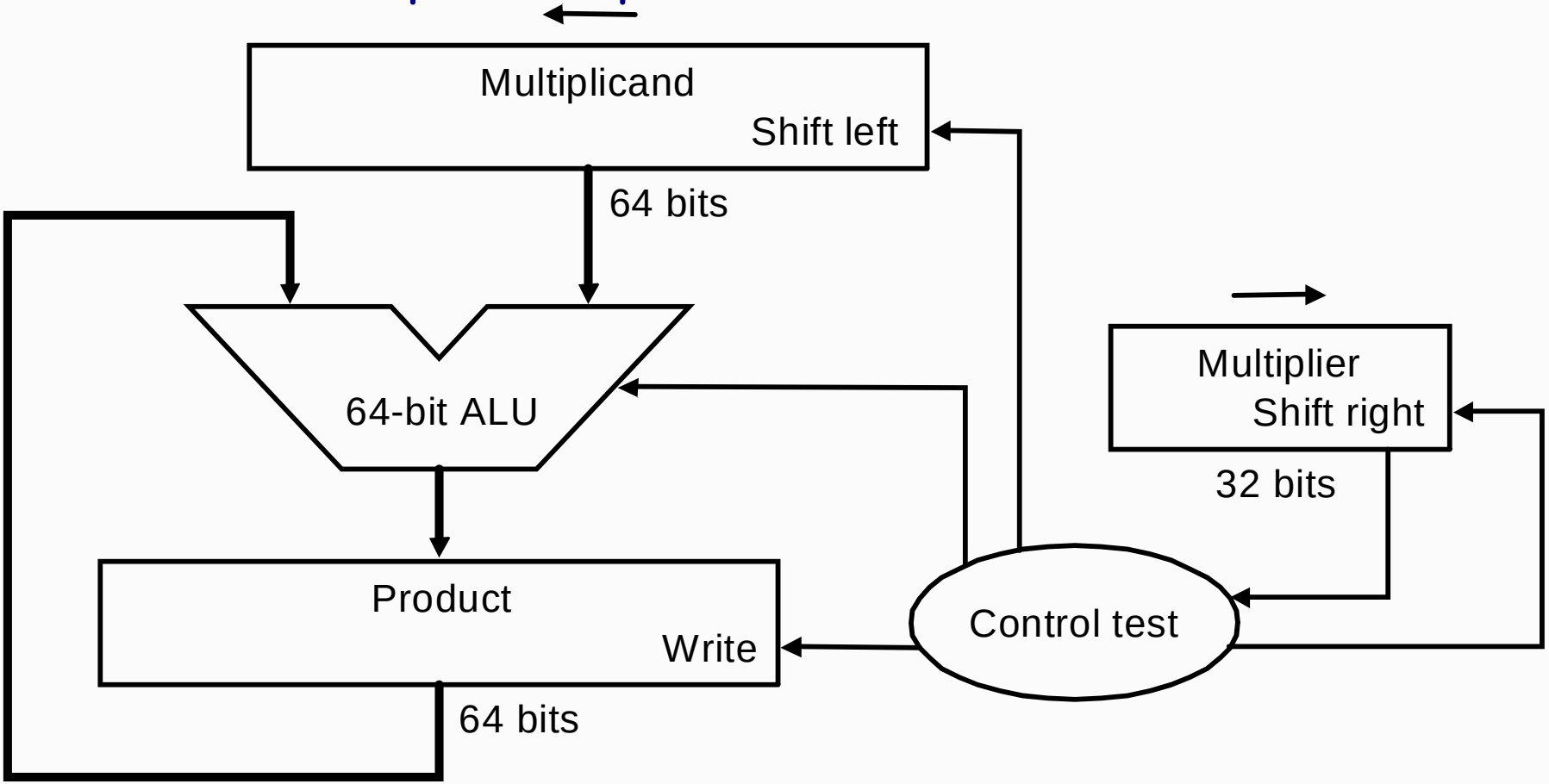
- Binary multiplication

```
Multiplicand * Multiplier
        1000₂
        1001₂
    ----------*
        1000
       0000
      0000
     1000
    ----------+
     1001000₂
```

- Look at current bit position
  - If multiplier is 1 then add multiplicand
  - Else add 0
  - shift multiplicand by 1

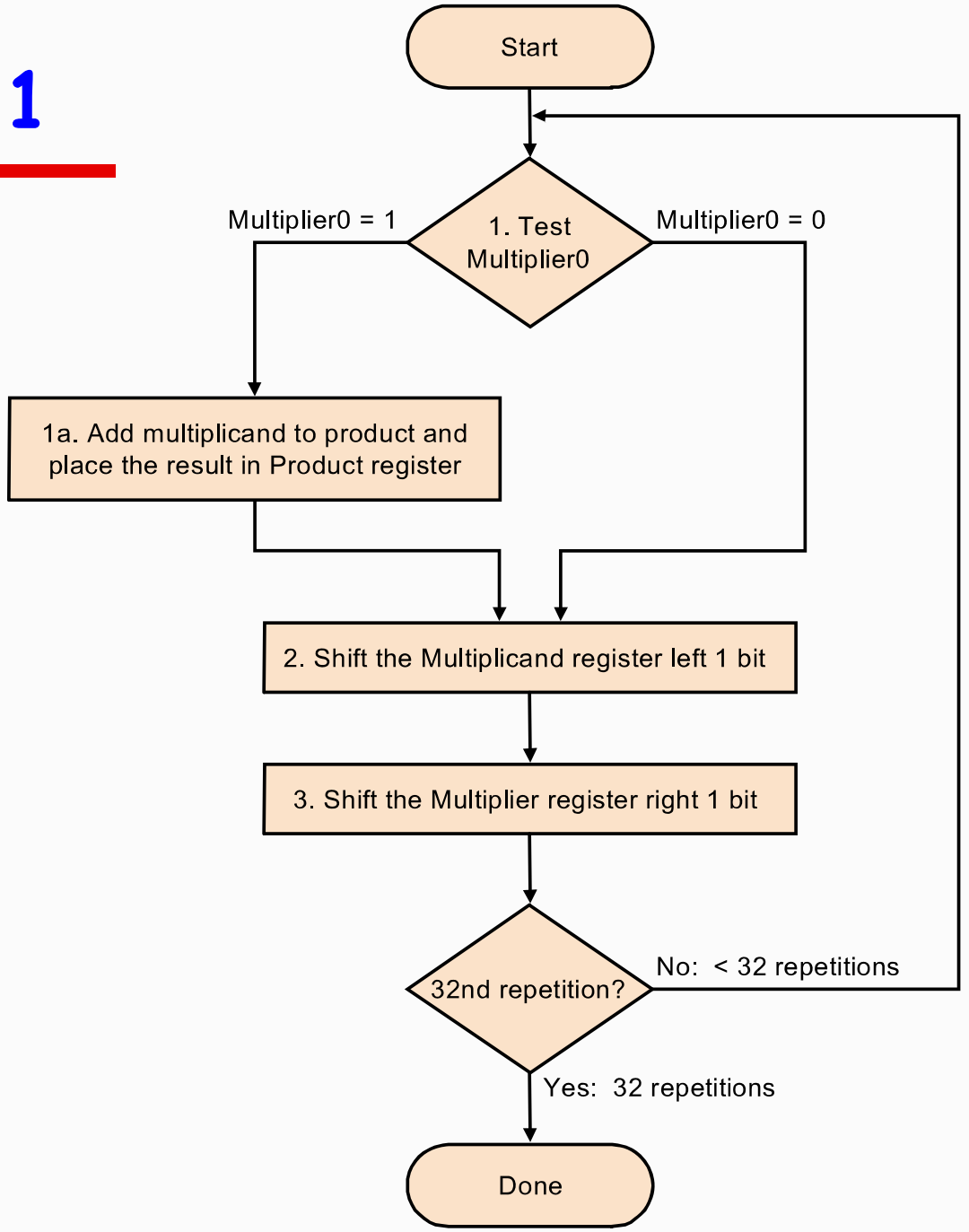
Multiplier Version 1

- 32 bits: multiplier
- 64 bits: multiplicand, product, ALU



Algorithm Version 1

- Basic algorithm
- Requires 32 iterations
  - Addition
  - Shift
  - Comparison
- Almost 100 cycles
- Too slow!



Example

- 3 · 2 or 0011 · 0010

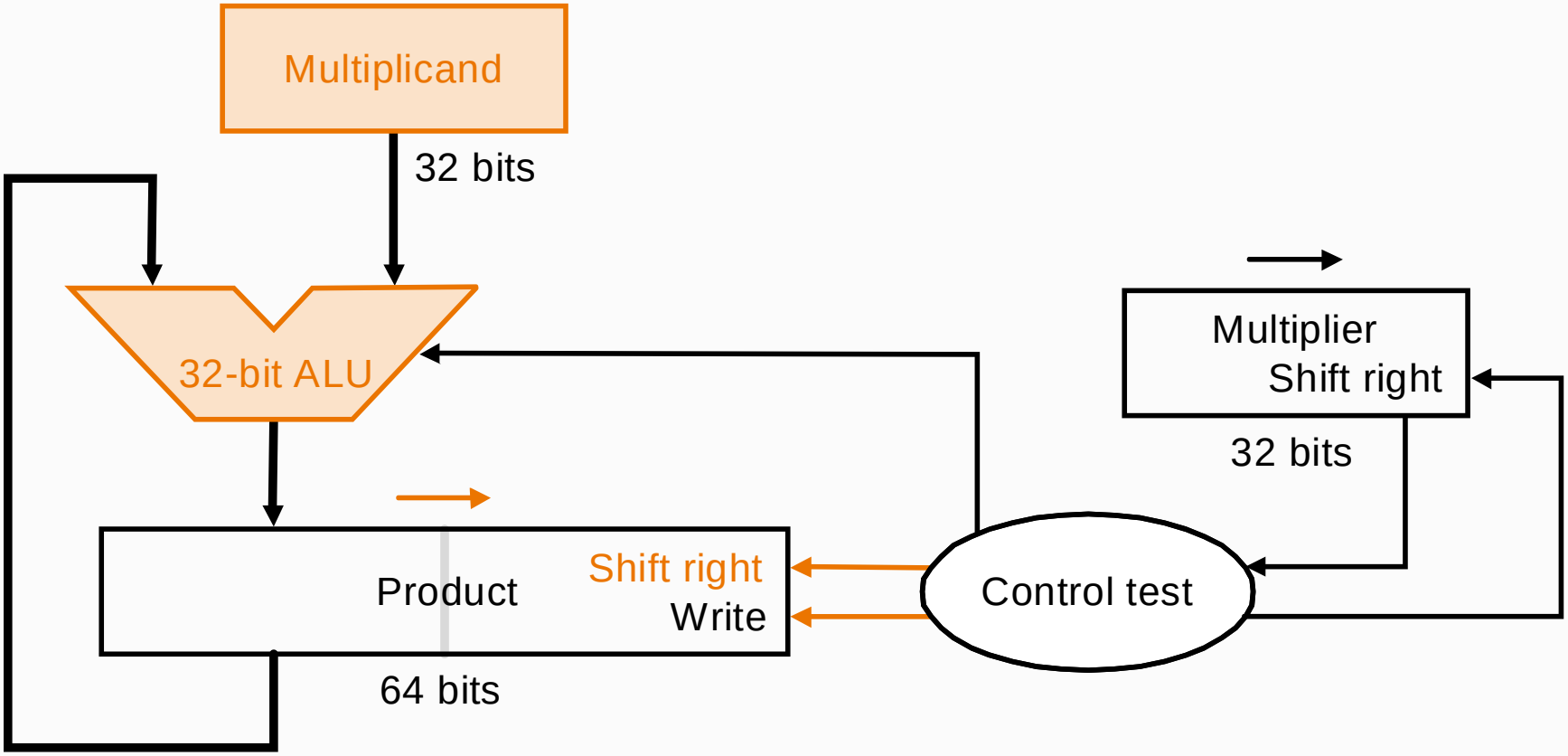
Iteration	Step	Multiplier	Multiplicand	Product
0	Initial Values	0011	0000 0010	0000 0000
	1a: 1⇒ Prod = Prod + Mcand	0011	0000 0010	0000 0010
1	2: Shift Left Multiplicand	0011	0000 0100	0000 0010
	3: Shift Right Multiplier	0001	0000 0100	0000 0010
	1a: 1⇒ Prod = Prod + Mcand	0001	0000 0100	0000 0110
2	2: Shift Left Multiplicand	0001	0000 1000	0000 0110
	3: Shift Right Multiplier	0000	0000 1000	0000 0110
	1a: 0⇒ No Operation	0000	0000 1000	0000 0110
3	2: Shift Left Multiplicand	0000	0001 0000	0000 0110
	3: Shift Right Multiplier	0000	0001 0000	0000 0110
	1a: 0⇒ No Operation	0000	0001 0000	0000 0110
4	2: Shift Left Multiplicand	0000	0010 0000	0000 0110
	3: Shift Right Multiplier	0000	0010 0000	0000 0110

Multiplier Version 2

- Half of the 64 bits of the multiplicand: always zero!
- Real addition is performed only with 32 bits
- Lest significant bits of the product don't change
- Idea:
  - Keep the multiplicand in a register
  - Shift the product
  - Shift the multiplier
- ALU reduced to 32 bits!

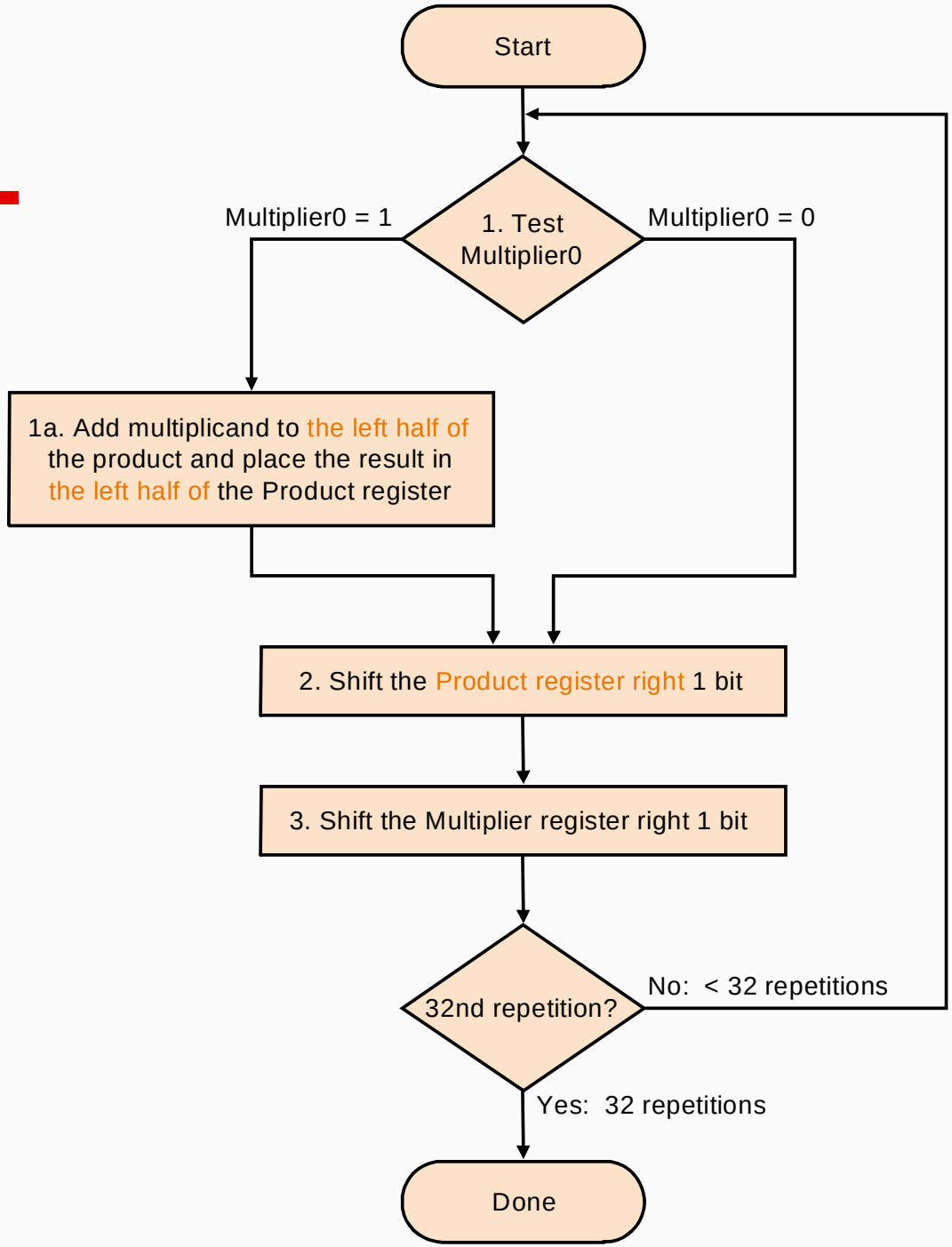
Multiplier Version 2

- Diagram of the V2 multiplier
- Only left half of product register is changed



Algorithm Ver 2

- Addition performed only on left half of product register
- Shift of product register



Revised 4-bit example

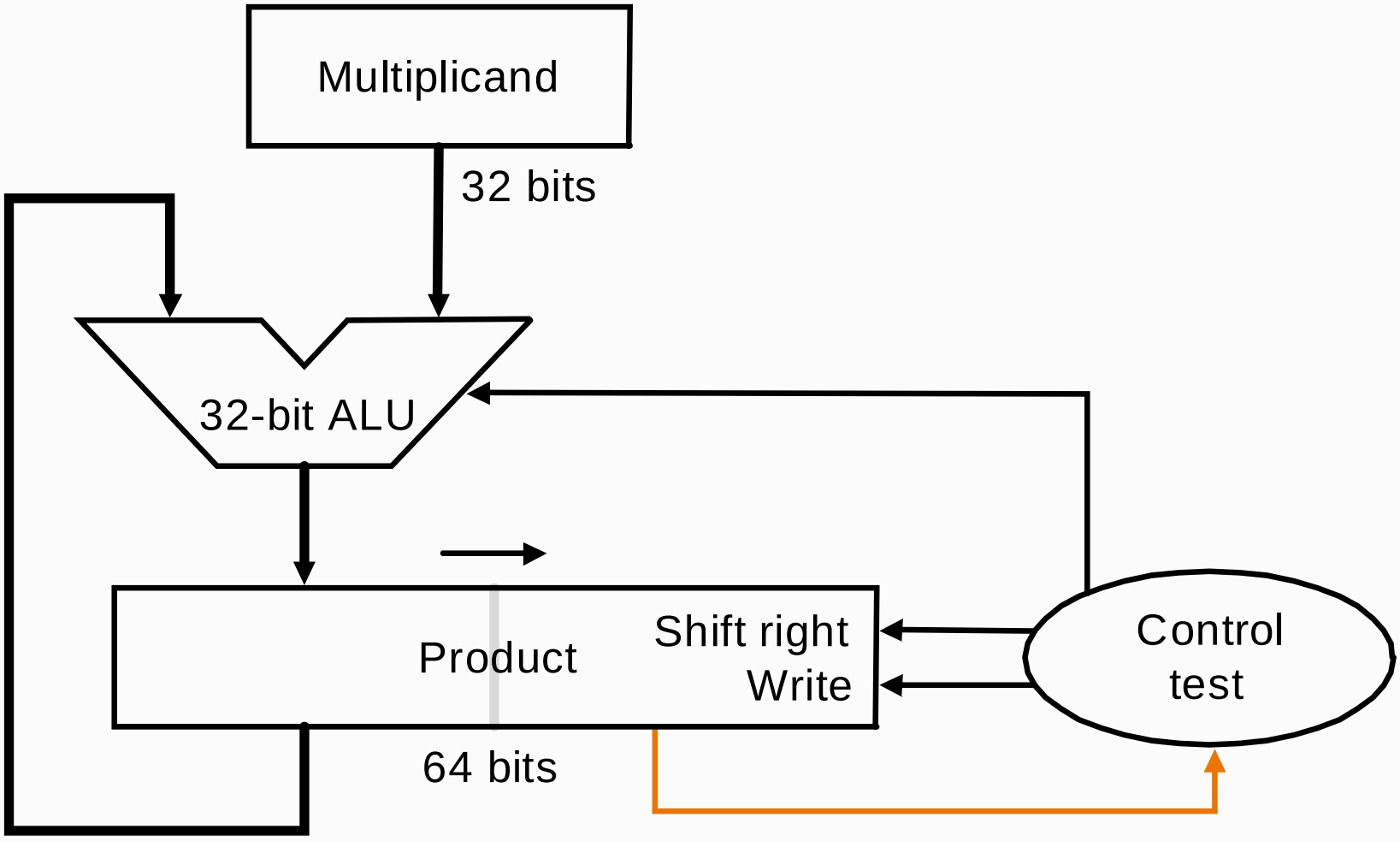
- 3 · 2 or 0011 · 0010

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial Values	0011	0010	0000 0000
	1a: 1⇒ Prod = Prod + Mcand	0011	0010	0010 0000
1	2: Shift Right Product	0011	0010	0001 0000
	3: Shift Right Multiplier	0001	0010	0001 0000
	1a: 1⇒ Prod = Prod + Mcand	0001	0010	0011 0000
2	2: Shift Right Product	0001	0010	0001 1000
	3: Shift Right Multiplier	0000	0010	0001 1000
	1a: 0⇒ No Operation	0000	0010	0001 1000
3	2: Shift Right Product	0000	0010	0000 1100
	3: Shift Right Multiplier	0000	0010	0000 1100
	1a: 0⇒ No Operation	0000	0010	0000 1100
4	2: Shift Right Product	0000	0010	0000 0110
	3: Shift Right Multiplier	0000	0010	0000 0110

Multiplier Ver 3

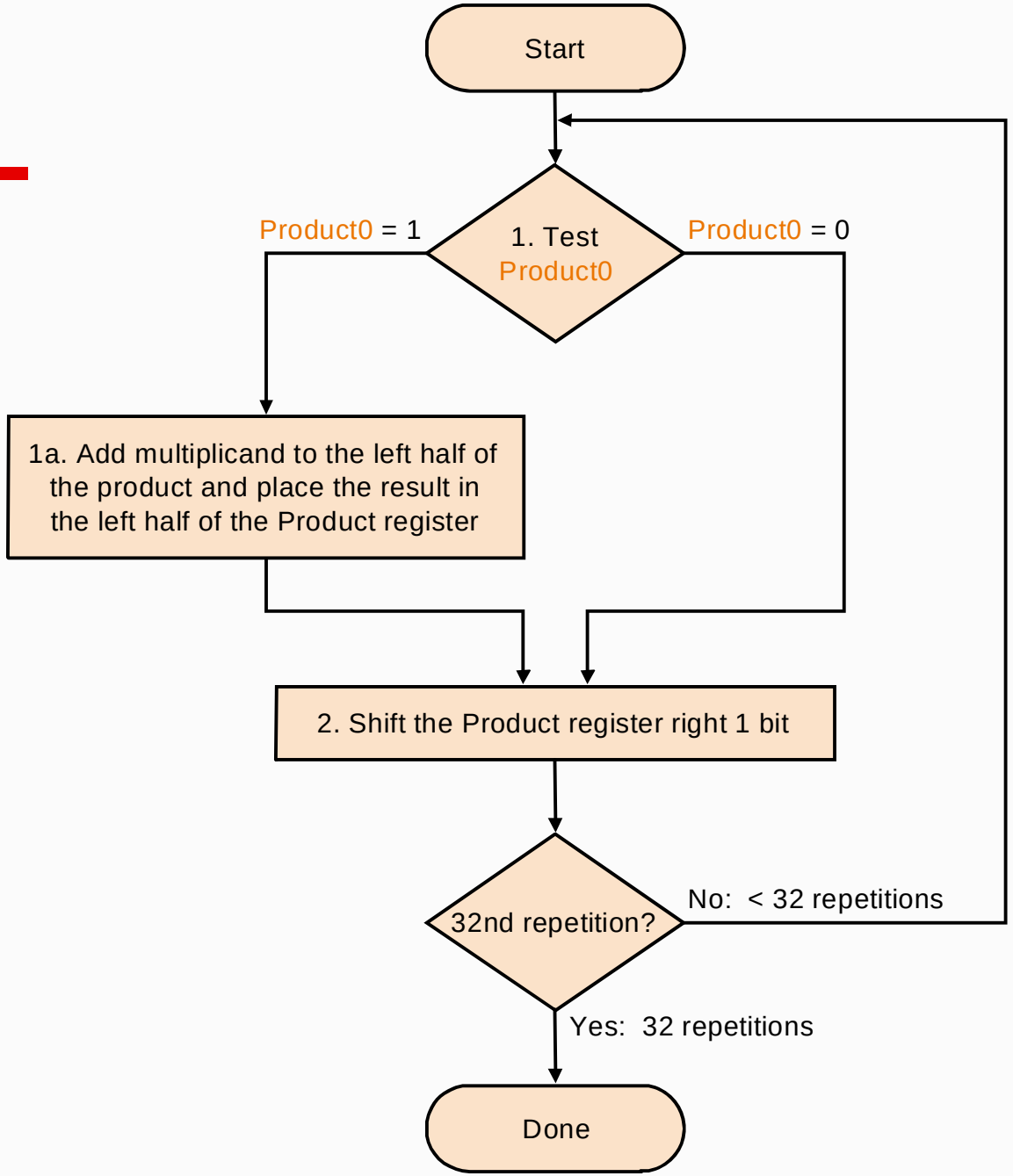
- Further optimization
- At the initial state the product register contains only '0'
- The lower 32 bits are simply shifted out
- Idea:
  - use these 32 bits for the multiplier and check the lowest bit of the product register
  - if add & shift or shift only

Multiplier Ver 3



Algorithm Ver 3

- Set product register to '0'
- Load lower bits of product register with multiplier
- Test least significant bit of product register



Example

- 2 * 3 or 0010 * 0011

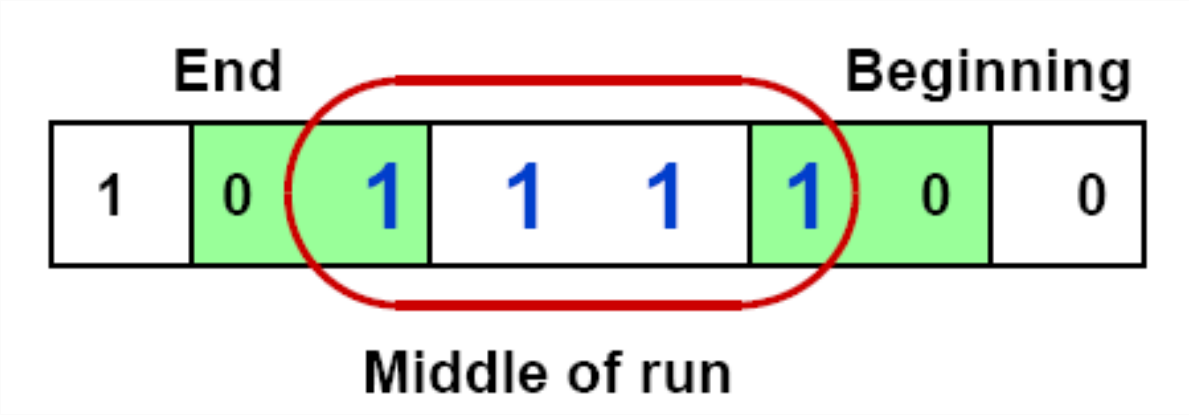
Iteration	Step	Multiplicand	Product
0	Initial Values	0010	0000 0011
1	1a: 1⇒ Prod = Prod + Mcand	0010	0010 0011
	2: Shift Right Product	0010	0001 0001
2	1a: 1⇒ Prod = Prod + Mcand	0010	0011 0001
	2: Shift Right Product	0010	0001 1000
3	1a: 0⇒ No Operation	0010	0001 1000
	2: Shift Right Product	0010	0000 1100
4	1a: 0⇒ No Operation	0010	0000 1100
	2: Shift Right Product	0010	0000 0110

Signed multiplication

- Basic approach:
  - Store the signs of the operands
  - Convert signed numbers to unsigned numbers (most significant bit (MSB) = 0)
  - Perform multiplication
  - If sign bits of operands as equal
    - sign bit = 0, else
    - sign bit = 1
- Improved method: Booth's Algorithm
- Assumption: addition and subtraction are available

Booth's Algorithm

- Idea: If you have a sequence of '1's
  - subtract at first '1' in multiplier
  - shift for the sequence of '1's
  - add where prior step had last '1'



- Result:
  - Possibly less additions and more shifts
  - Faster, if shifts are faster than additions

Booth's Algorithm

- Lets assume a is the multiplier and b is the multiplicand

a_i	a_{i-1}	Operation
0	0	Do nothing
0	1	Add *b*
1	0	Subtract *b*
1	1	Do nothing

$$(a_{i-1} - a_i)$$

- Result = 0 : do nothing
- Result = -1 : subtract b
- Result = +1 : add b

Shifting the multiplicand left is equivalent to multiplying by power of 2

Booth's Algorithm

- Rewritten

$$
\begin{aligned}
&(a_{-1} - a_0) \times b \times 2^0 \\
&+ (a_0 - a_1) \times b \times 2^1 \\
&+ (a_1 - a_2) \times b \times 2^2 \\
&\ldots \\
&+ (a_{29} - a_{30}) \times b \times 2^{30} \\
&+ (a_{30} - a_{31}) \times b \times 2^{31}
\end{aligned}
$$

$$-a_i \times 2^i + a_i \times 2^{i+1} = (-a_i + 2a_i) \times 2^i = a_i \times 2^i$$

$$a_{-1} = 0$$

$$b \times \left(\left(a_{31} \times -2^{31} \right) + \left(a_{30} \times 2^{30} \right) + \ldots + \left(a_1 \times 2^1 \right) + \left(a_0 \times 2^0 \right) \right) = b \times a$$

Booth's Algorithm

- Example

```
   Straight              Booth
     0010                0010
     0110                0110
     ----*               ----*
     0000 shift          0000 shift
    0010  add           0010  sub
   0010   add          0000   shift
  0000    shift       0010    add
 00001100            00001100
```

Logic required identifying the run

Booth's Algorithm

- Analysis of two consecutive bits

Current Bit	Bit to the right	Explanation	Example
1	0	Beginning of a run of 1's	000011110 00
1	1	Middle of a run of 1's	000011110 00
0	1	End of run of 1's	000011110 00
0	0	Middle of a run of 0's	000011110 00

- Action

1a	0	0	No arithmetic operation
1b	0	1	Add multiplicand to left half
1c	1	0	Subtract multiplicand from left
1d	1	1	No arithmetic operation

Comparison

- Bit -1 = '0'
- Arithmetic shift right:
  - keeps the leftmost bit constant
  - no change of sign bit !

Iteration	Multiplicand	Original Algorithm		Booth's Algorithm	
		Step	Product	Step	Product
0	0010	Initial Value	0000 0110	Initial Value	0000 0110 0
1	0010	1:0⇒no operation	0000 0110	1a:00⇒no operation	0000 0110 0
	0010	2: Shift right Product	0000 0011	2: Shift right Product	0000 0011 0
2	0010	1a: 1⇒ Prod = Prod + Mcand	0010 0011	1c:10⇒Prod = Prod - Mcand	1110 0011 0
	0010	2: Shift Right Product	0001 0001	2: Shift right Product	1111 0001 1
3	0010	1a: 1⇒ Prod = Prod + Mcand	0011 0001	1d:11⇒no operation	1111 0001 1
	0010	2: Shift Right Product	0001 1000	2: Shift right Product	1111 1000 1
4	0010	1:0⇒no operation	0001 1000	1b:10⇒Prod = Prod + Mcand	0001 1000 1
	0010	2: Shift right Product	0000 1100	2: Shift right Product	0000 1100 0

Example with negative numbers

- 2 * -3 = - 6
- 0010 * 1101 = 1111 1010

Iteration	Step	Multiplicand	Product
0	Initial Value	0010	0000 1101 0
1	1c:10⇒Prod = Prod - Mcand	0010	1110 1101 0
	2: Shift right Product	0010	1111 0110 1
2	1b:01⇒Prod = Prod + Mcand	0010	0001 0110 1
	2: Shift right Product	0010	0000 1011 0
3	1c:10⇒Prod = Prod - Mcand	0010	1110 1011 0
	2: Shift right Product	0010	1111 0101 1
4	1d:11⇒no operation	0010	1111 0101 1
	2: Shift right Product	0010	1111 1010 1

Division

- Somewhat more difficult than multiplication
- Critical point: Divide by 0
- Example

Dividend ÷ Divisor = Quotient

$1001010_{10} \div 1000_{10} = 1001_{10}$

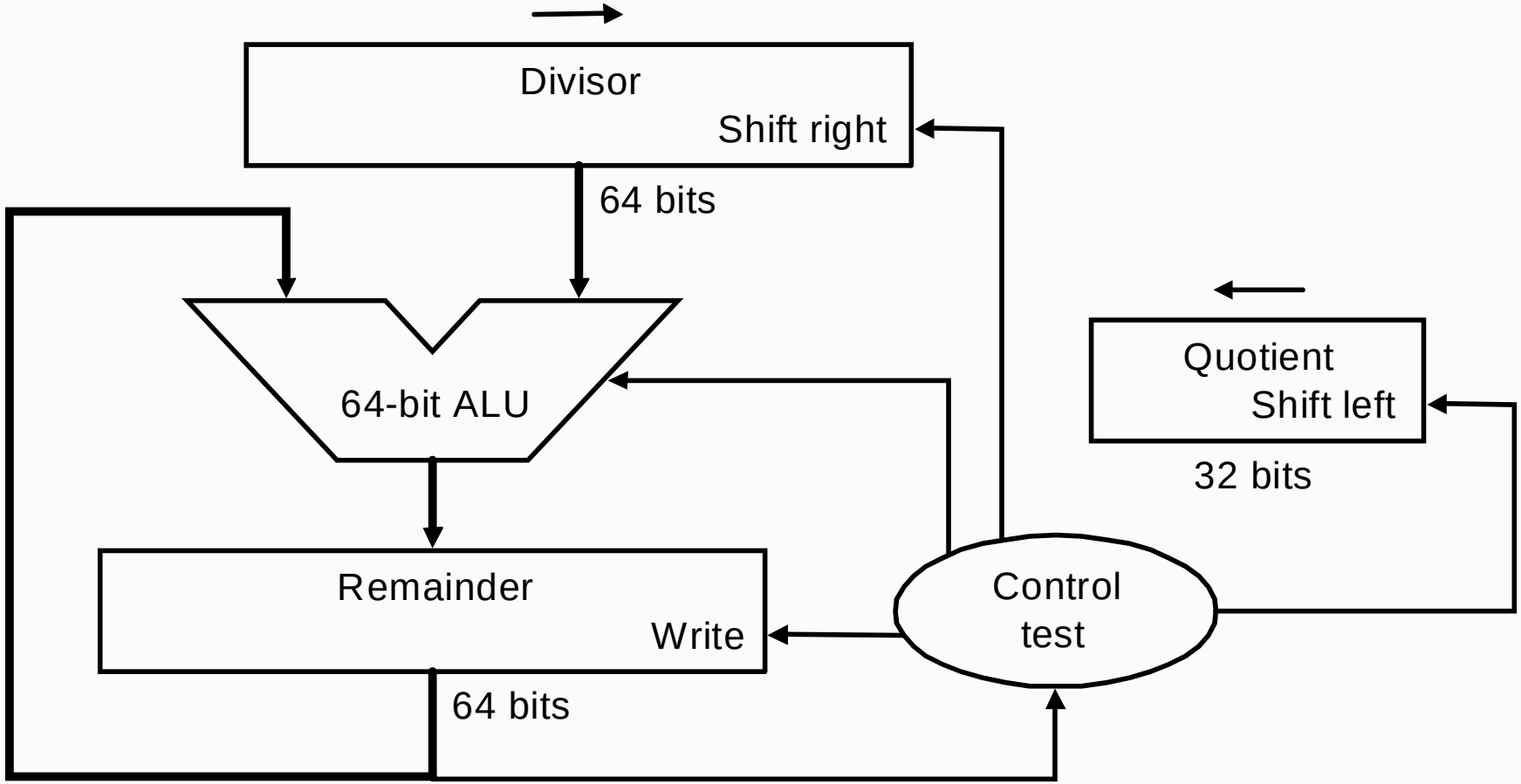
```
      1001
1000|1001010
     1000
      0010
      0000
       0101
       0000
        1010
        1000
          10 Remainder
```

Division

- Dividend = quotient * divisor + remainder
- Remainder < divisor
- Basic hardware for division
- Iterative subtraction
- Result:
  - Greater 0: then we get a 1
  - Smaller 0: then we get a 0

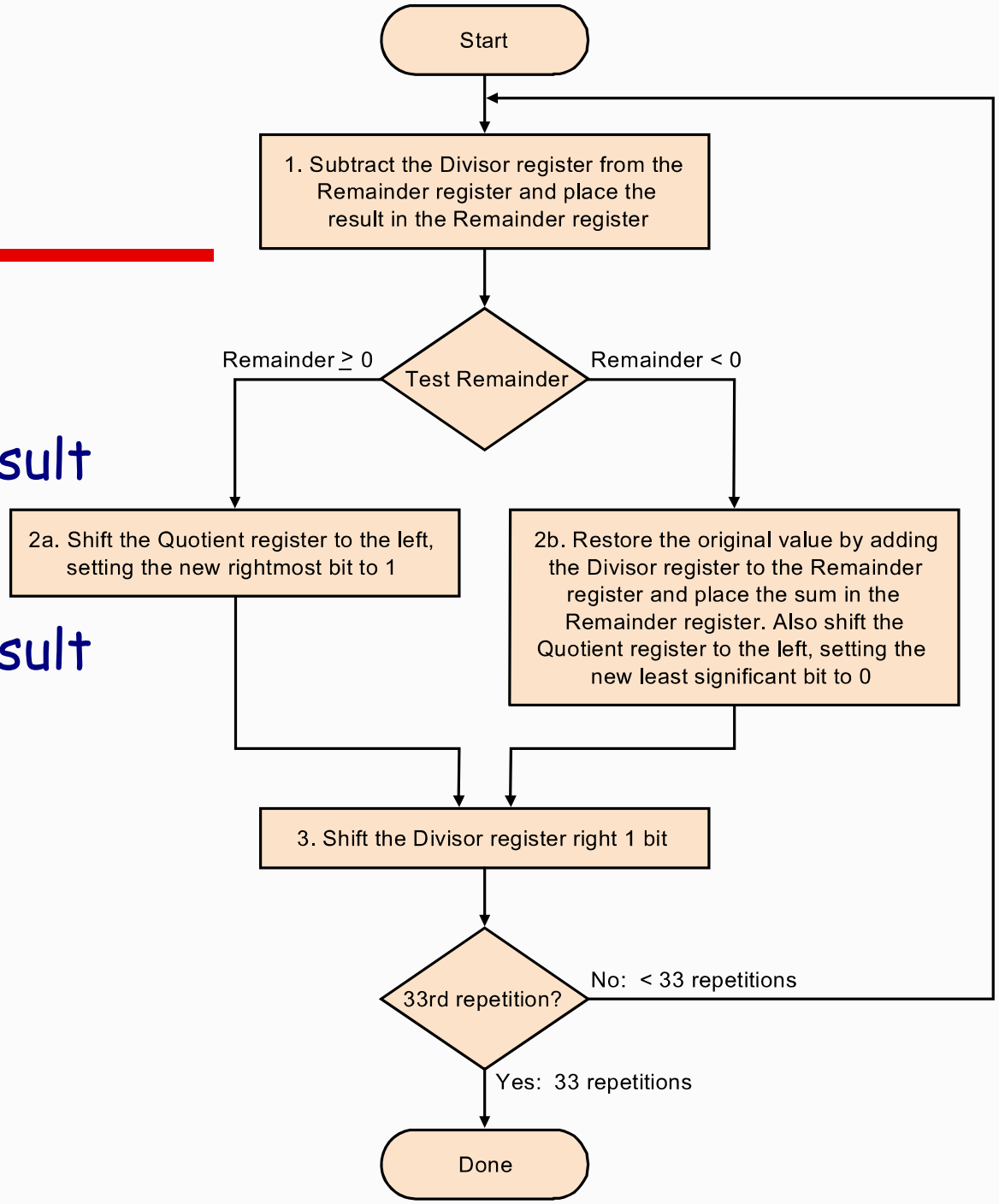
Division V1

- Divisor in left half
- Shift right each step



Algorithm V1

- Each step:
  - Subtract divisor
  - Depending on Result
    - Leave or
    - Restore
  - Depending on Result
    - Write '1' or
    - Write '0'
- N-bit Divisor
  - N+1 steps



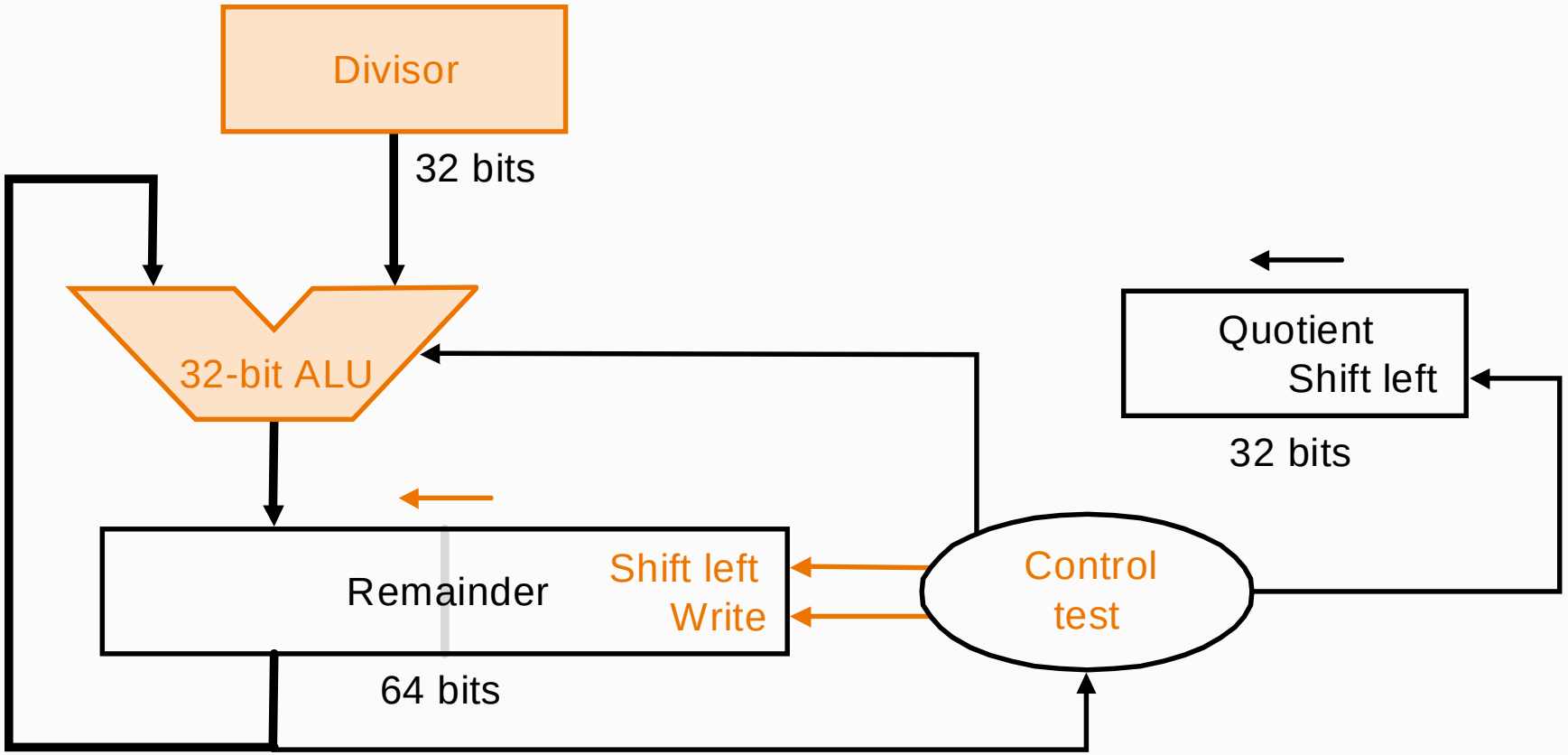
Example

- 7 : 2 or 0000 0111 : 0010

Iteration	Step	Quotient	Divisor	Remainder
0	Initial Values	0000	0010 0000	0000 0111
1	1: Rem = Rem - Div	0000	0010 0000	1110 0111
	2b: Rem<0 → +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: shift Right Div	0000	0001 0000	0000 0111
2	1: Rem = Rem - Div	0000	0001 0000	1111 0111
	2b: Rem<0 → +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: shift Right Div	0000	0000 1000	0000 0111
3	1: Rem = Rem - Div	0000	0000 1000	1111 1111
	2b: Rem<0 → +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: shift Right Div	0000	0000 0100	0000 0111
4	1: Rem = Rem - Div	0000	0000 0100	0000 0011
	2a: Rem>0 → sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: shift Right Div	0001	0000 0010	0000 0011
5	1: Rem = Rem - Div	0001	0000 0010	0000 0001
	2a: Rem>0 → sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: shift Right Div	0011	0000 0001	0000 0001

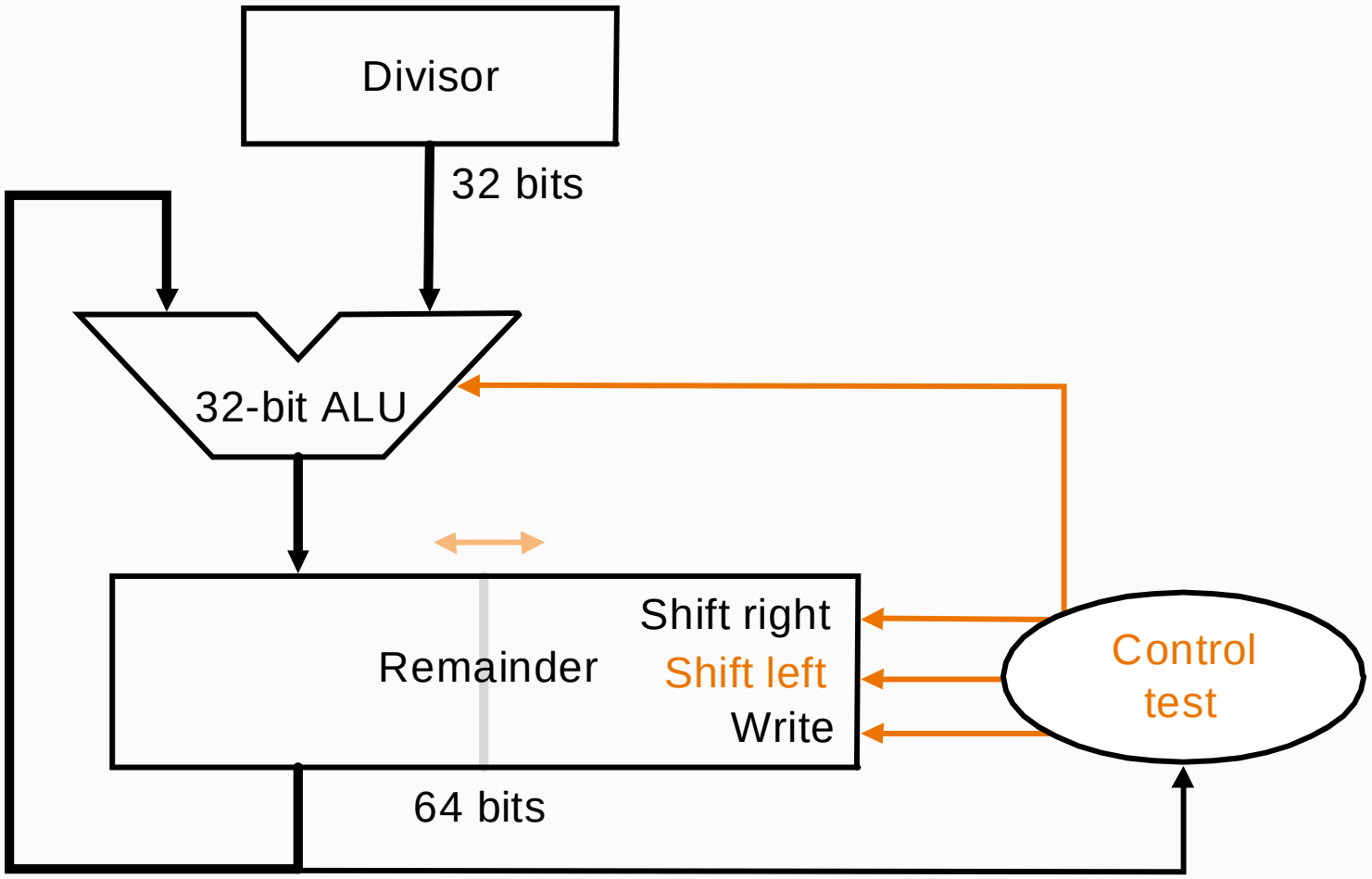
Division V2

- Reduction of Divisor and ALU width by half
- Shifting of the remainder
- Saving 1 iteration



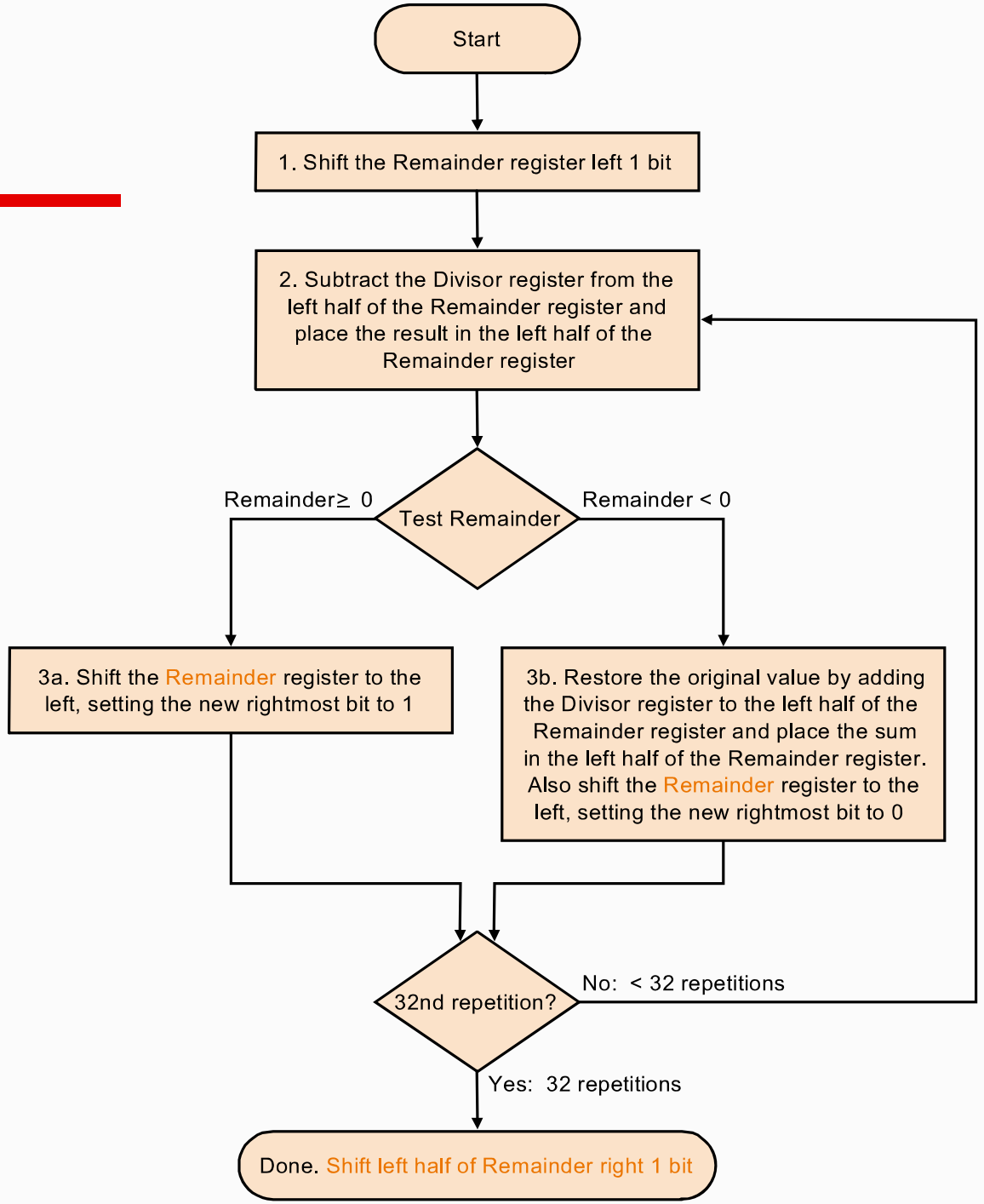
Division V3

- Remainder register keeps quotient
- No quotient register required



Algorithm V3

- Much the same than the last one
- Except change of register usage



Example

- Well known numbers: 0000 0111 : 0010

Iteration	Step	Divisor	Remainder
0	Initial Values	0010	0000 0111
	Shift left Rem 1	0010	0000 1110
1	2: Rem = Rem - Div	0010	1110 0111
	3b: Rem<0 → +Div, sll R, R0 = 0	0010	0001 1100
2	2: Rem = Rem - Div	0010	1111 0111
	3b: Rem<0 → +Div, sll R, R0 = 0	0010	0011 1000
3	2: Rem = Rem - Div	0010	0001 1000
	3a: Rem>0 → sll R, R0 = 1	0010	0011 0001
4	2: Rem = Rem - Div	0010	0001 0001
	3a: Rem>0 → sll R, R0 = 1	0010	0010 0011
	Shift left half of Remainder right 1 bit	0010	0001 0011

Signed division

- Keep the signs in mind for
- Dividend and Remainder
  - + 7 : + 2 = + 3 Remainder = +1
    7 = 3 x 2 + (+1) = 6 + 1
  - - 7 : + 2 = - 3 Remainder = -1
    -7 = -3 x 2 + (-1) = - 6 - 1
  - + 7 / - 2 = - 3 Remainder = +1
  - - 7 / - 2 = + 3 Remainder = -1
- One 64 bit register : Hi & Lo
  - Hi: Remainder, Lo: Quotient
- Instructions: div, divu
- Divide by 0 / overflow : Check by software