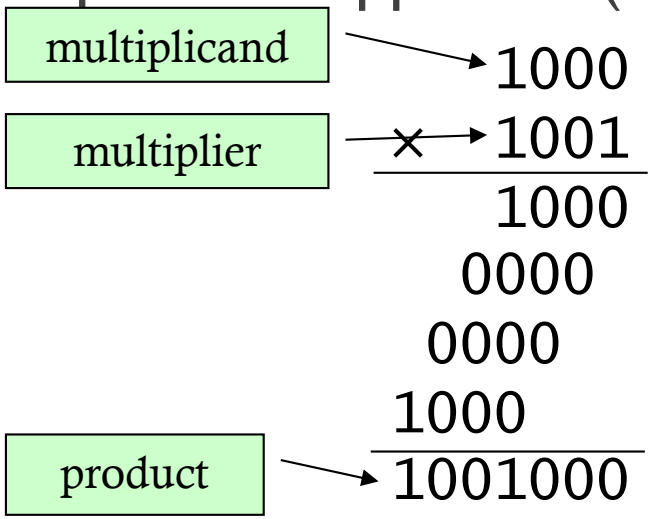
Roadmap

- Operations on integers
  - Addition and subtraction
  - Multiplication
  - Division
- Floating-point numbers
  - Representation
  - Addition and multiplication

Multiplication

- Start with long-multiplication approach (unsigned)

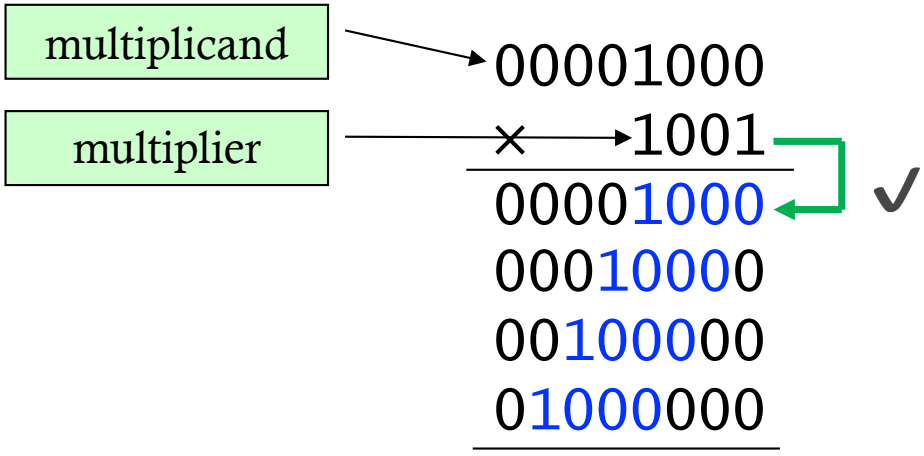


  - Length of product is the sum of operand lengths
    - m bits x n bits = m+n bit product (at most)
- Binary makes it easy:
  - 0 => place 0 (0 x multiplicand)
  - 1 => place a copy (1 x multiplicand)

Hardware Multiplication

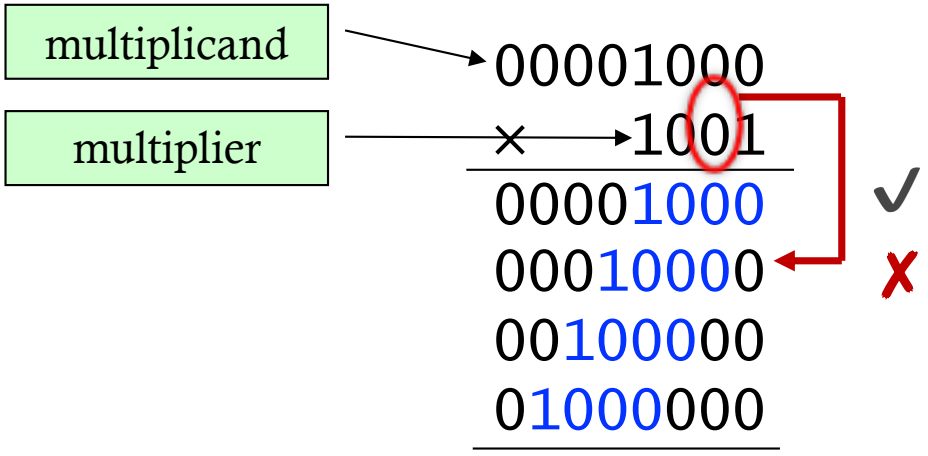
- Basic elements we need
  - Checking whether a digit is 0 or 1
  - Addition
    - We have basic ALU for addition
  - Alignment of results
    - We can shift contents of a register

Long Multiplication Simulated



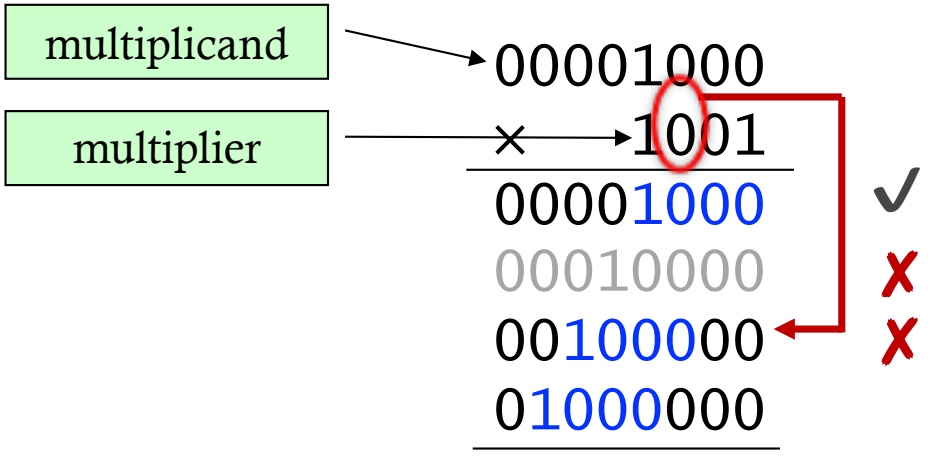
- Left shifting multiplicand
- Check multiplier digit one by one
  - 1→ add
  - 0→ no add

Long Multiplication Simulated



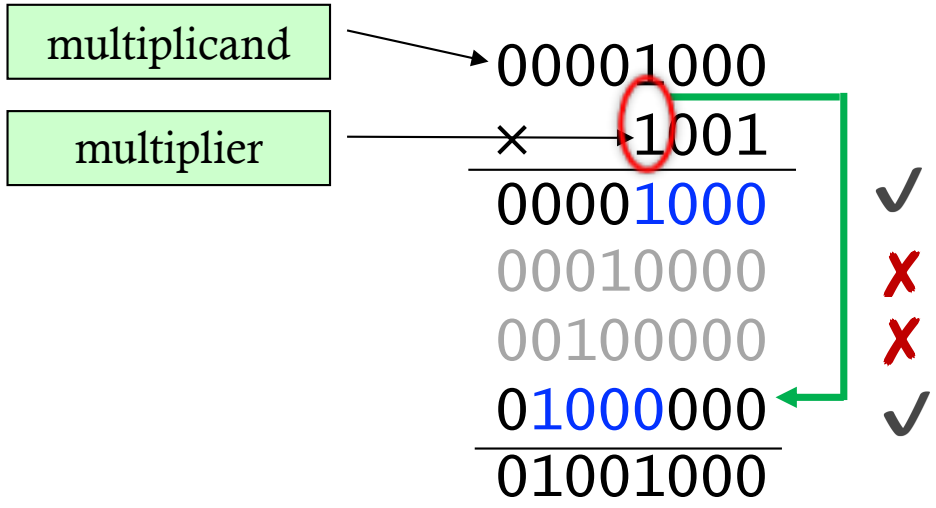
- Left shifting multiplicand
- Check multiplier digit one by one
  - 1→ add
  - 0→ no add

Long Multiplication Simulated



- Left shifting multiplicand
- Check multiplier digit one by one
  - 1→ add
  - 0→ no add

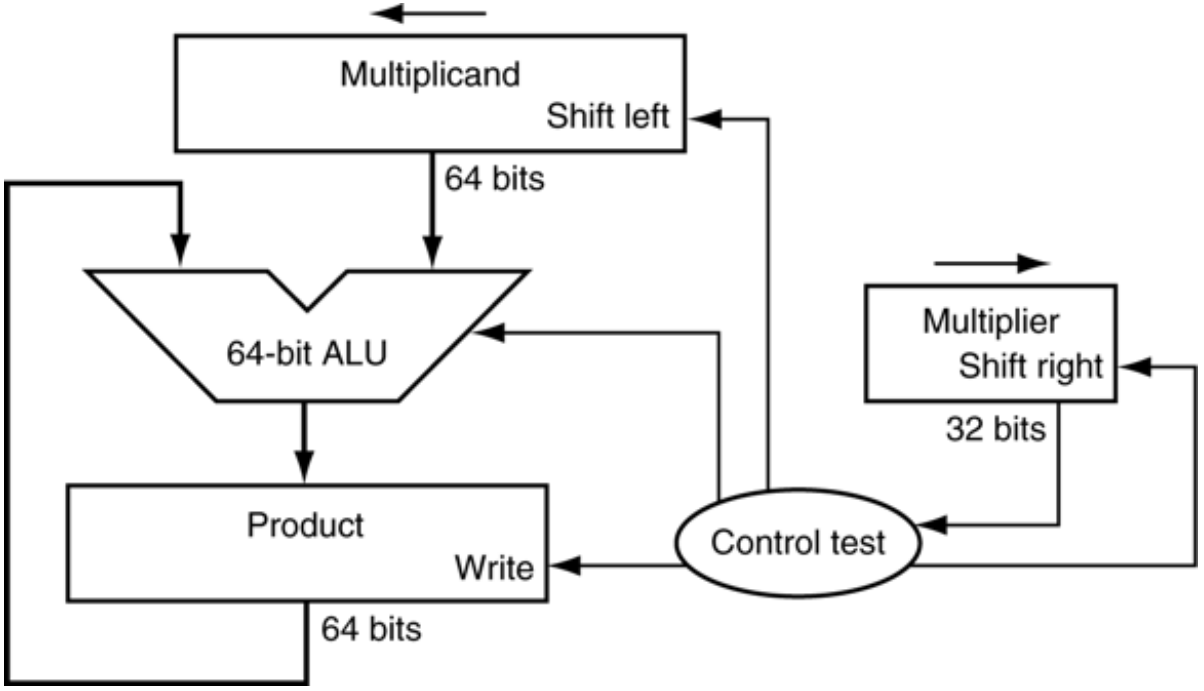
Long Multiplication Simulated



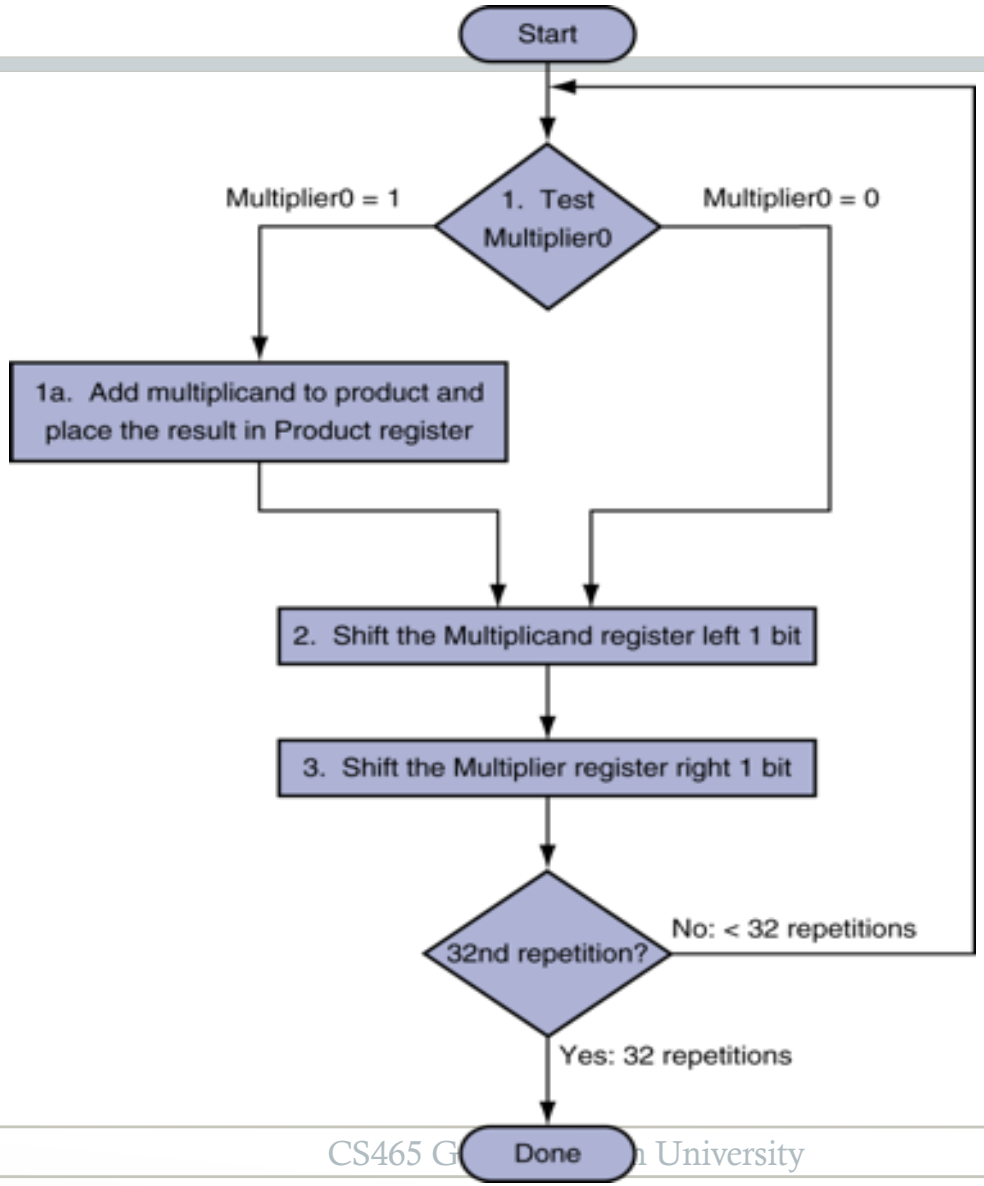
- Left shifting multiplicand
- Check multiplier digit one by one
  - 1→ add
  - 0→ no add

Multiplication

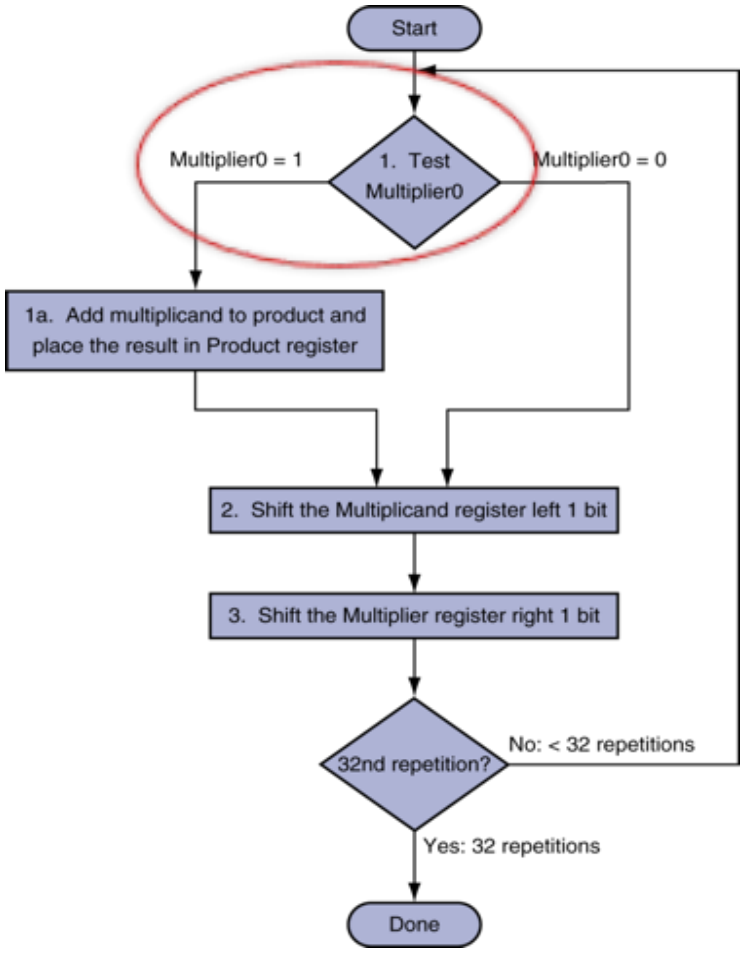
- Hardware



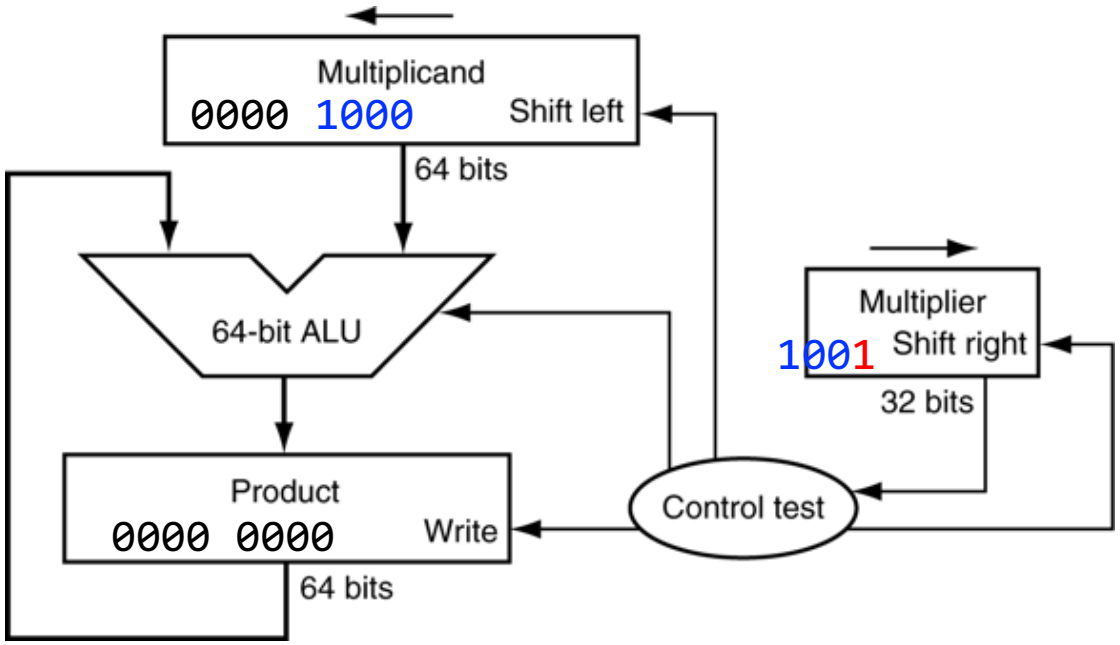
Flow Chart for Multiply



Multiplication Example

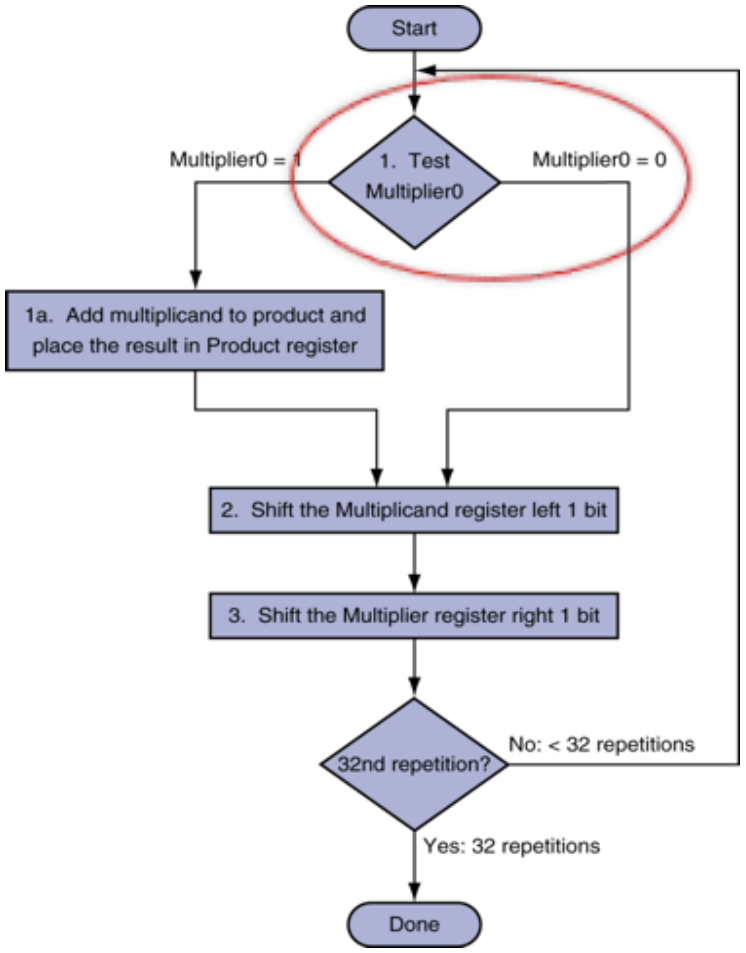


1000 x 1001

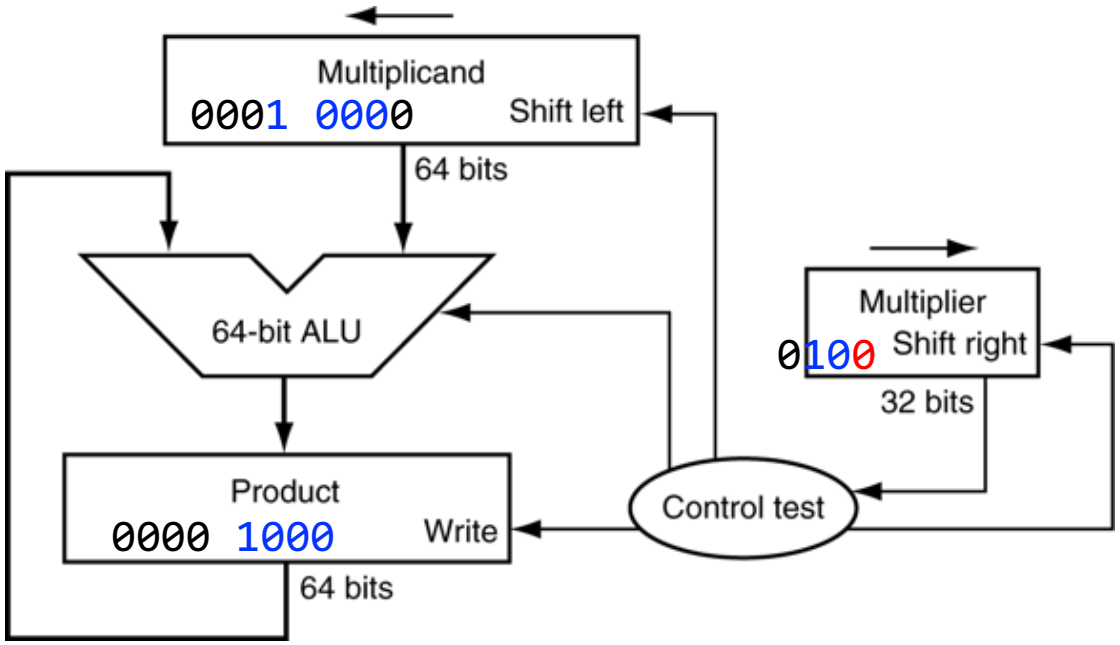


- Multiplicand initially lower half
- Product initially zero

Multiplication Example

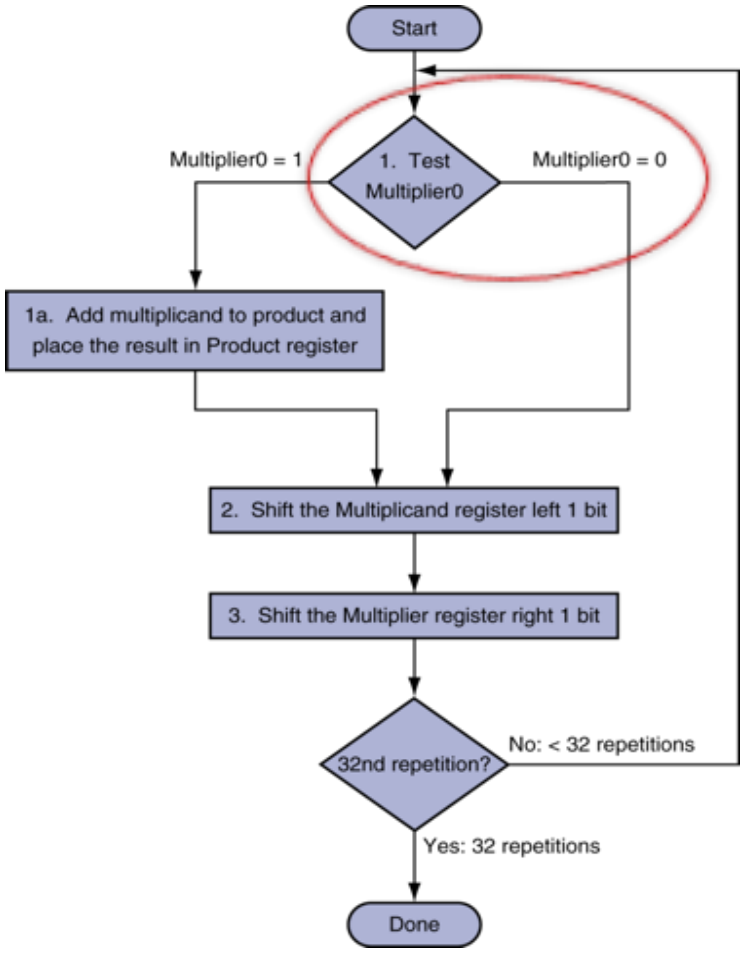


1000 x 1001

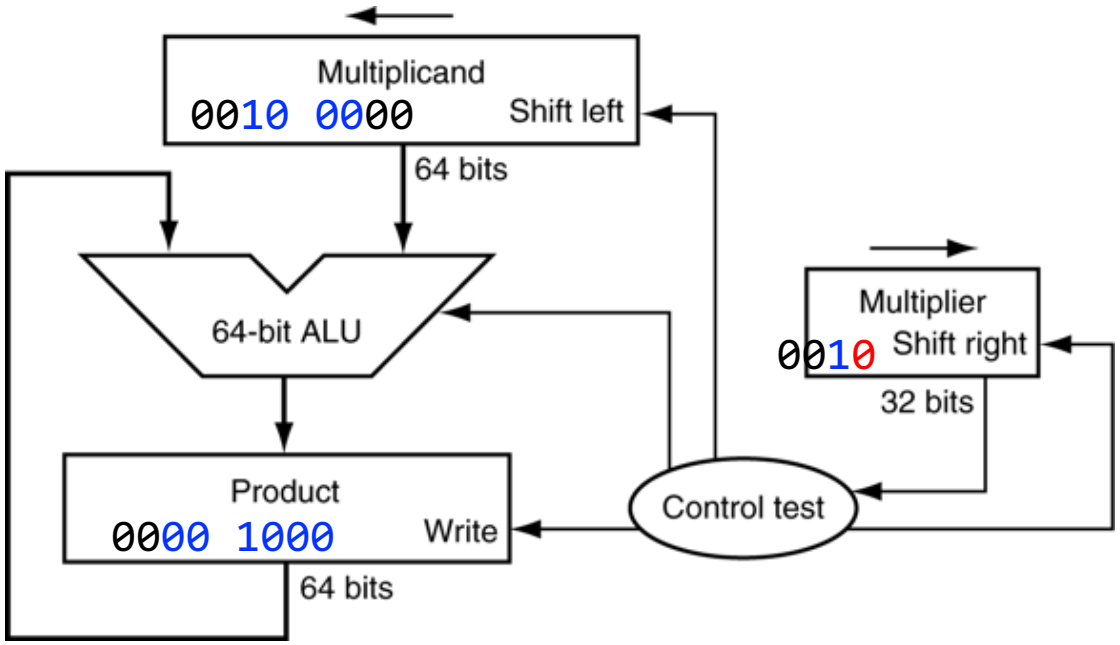


- Product += Multiplicand
- Shift multiplicand and multiplier

Multiplication Example

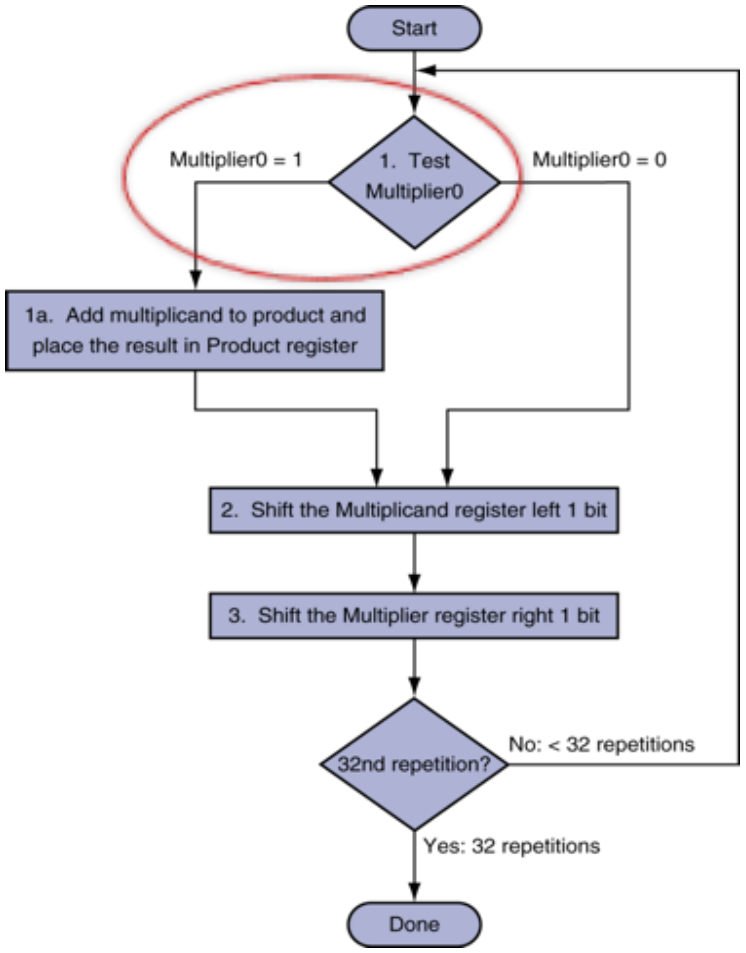


1000 x 1001

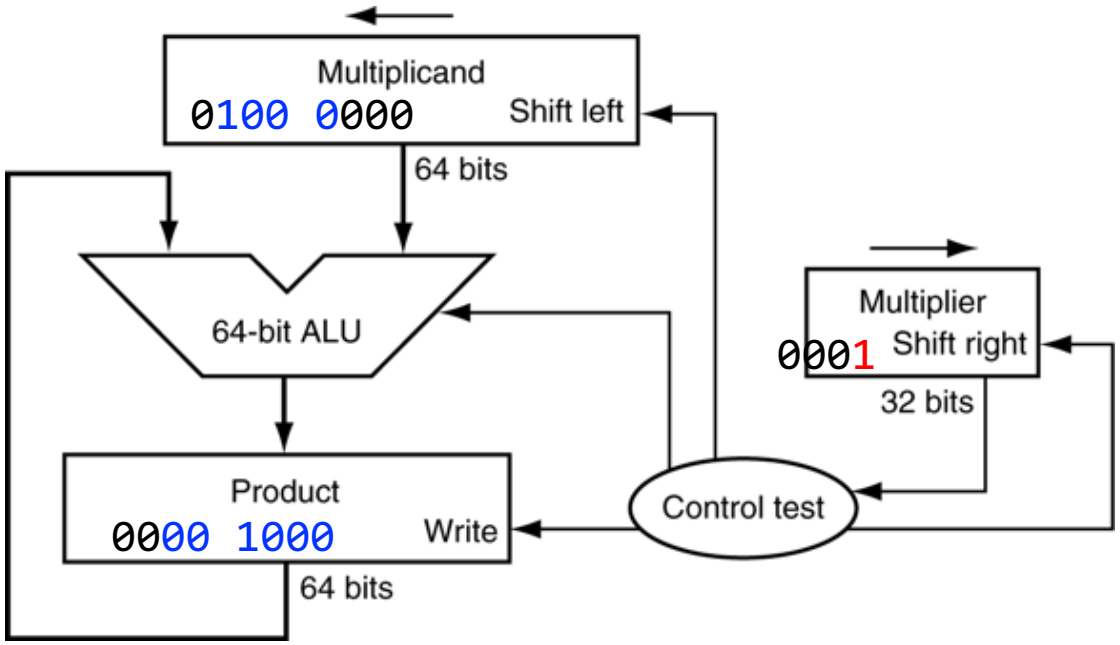


- Product no change
- Shift multiplicand and multiplier

Multiplication Example

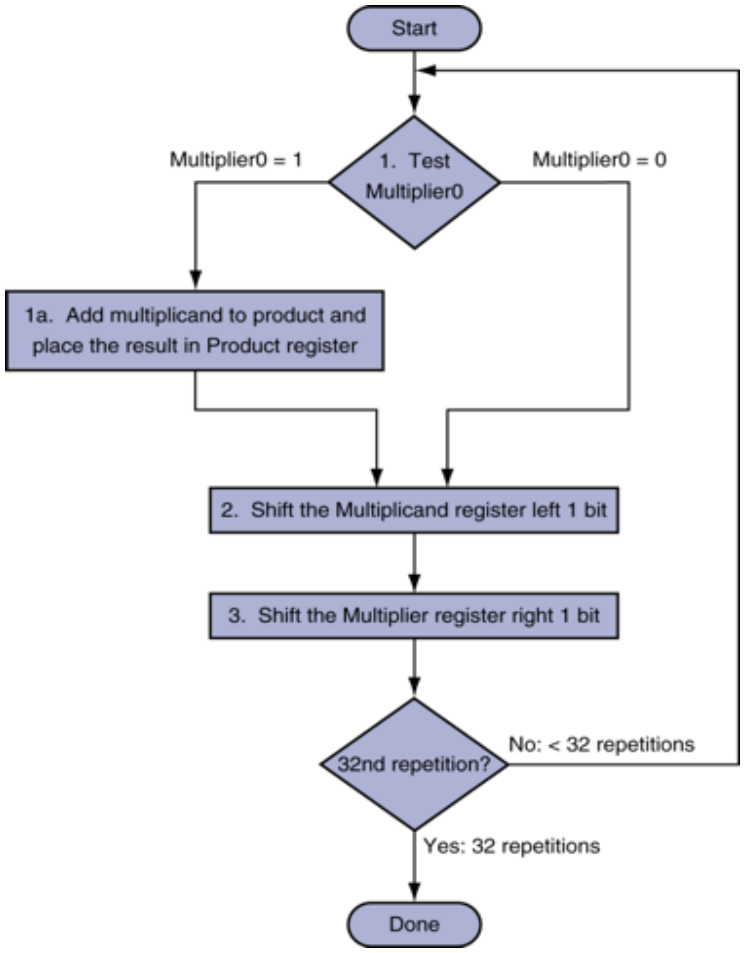


1000 x 1001

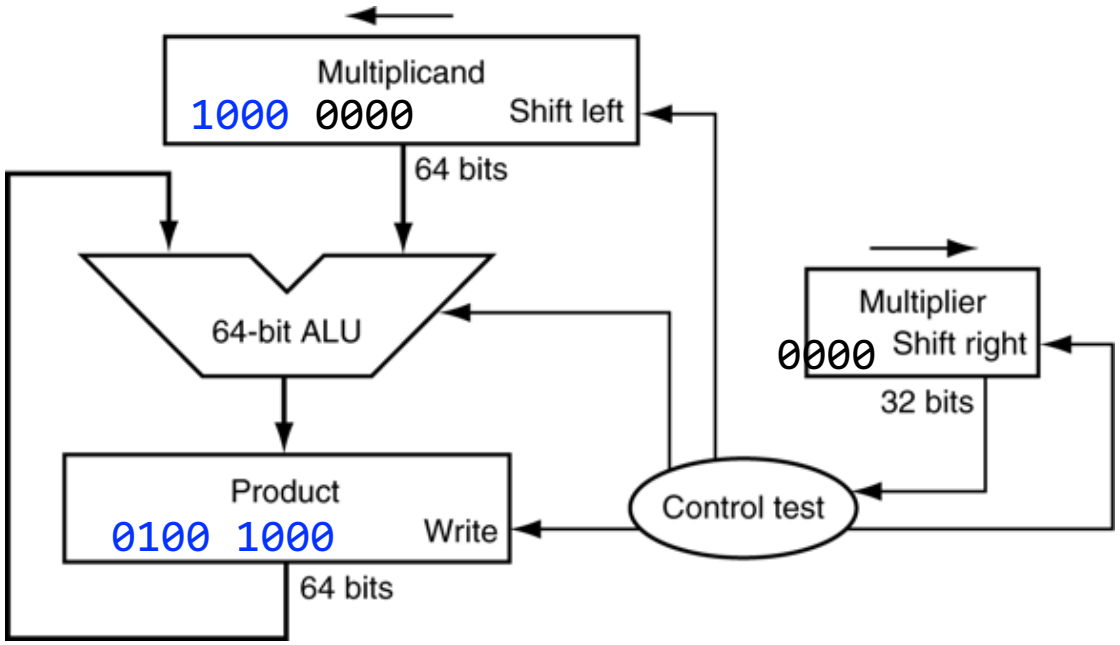


- Product no change
- Shift multiplicand and multiplier

Multiplication Example



1000 x 1001



- Product += Multiplicand
- Done after 4 iterations

Multiplication Example

0010 x 0011

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	001(1)	0000 0010	0000 0000
1	1a: 1 ⟹ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	000(1)	0000 0100	0000 0010
2	1a: 1 ⟹ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	000(0)	0000 1000	0000 0110
3	1: 0 ⟹ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	000(0)	0001 0000	0000 0110
4	1: 0 ⟹ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

FIGURE 3.6 Multiply example using algorithm in Figure 3.4. The bit examined to determine the next step is circled in color.

Multiplication Optimizations

- Signed multiplication
  - Unsigned multiplication on absolute values, then set the sign
- Optimized multiplier
  - Keep multiplicand fixed and shift product
  - Combine product and multiplier
  - Register space saved
- Parallelize multiple steps
  - Requires more hardware resources
  - Reduced time

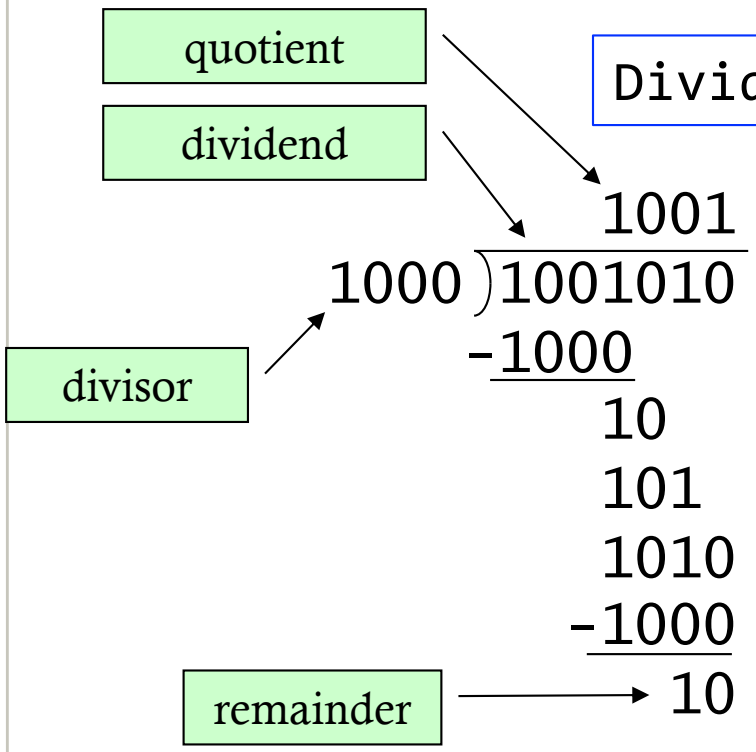
MIPS Multiplication

- Two 32-bit registers for product
  - HI: most-significant 32 bits
  - LO: least-significant 32-bits
- Instructions
  - `mult rs, rt / multu rs, rt`
    - 64-bit product in HI/LO
  - `mfhi rd / mflo rd`
    - Move from HI/LO to rd
    - Can test HI value to see if product overflows 32 bits
  - `mul rd, rs, rt`
    - Least-significant 32 bits of product –> rd
    - Ignore overflow

Roadmap

- Operations on integers
  - Addition and subtraction
  - Multiplication
  - Division
- Floating-point numbers
  - Representation
  - Addition and multiplication

Division

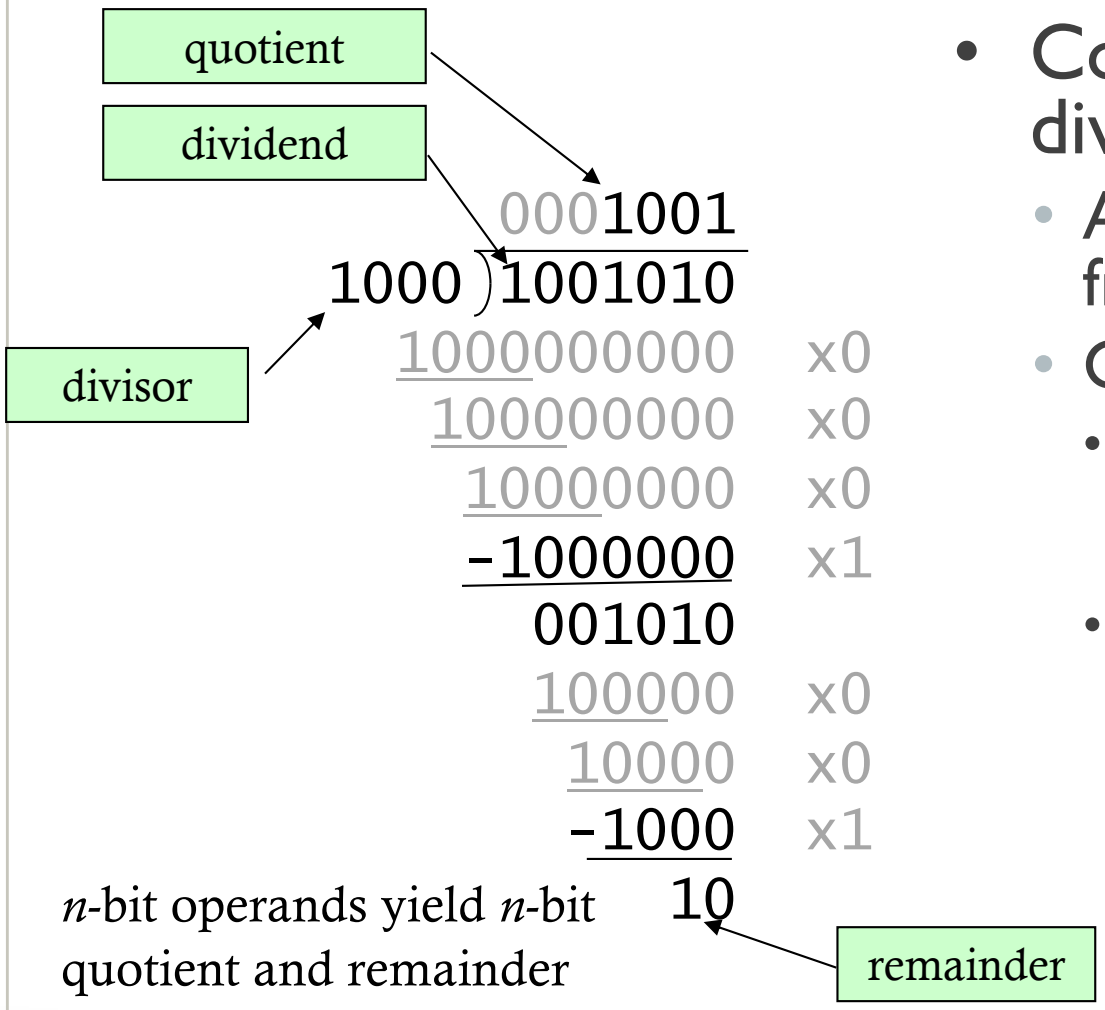


`Dividend = quotient * divisor + remainder`

- Check for 0 divisor
- Long division
  - If divisor ≤ dividend bits
    - Set 1 bit in quotient, subtract
  - Otherwise
    - Put 0 bit in quotient, bring down next dividend bit
- Computer hardware
  - More trials and fails

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

Division

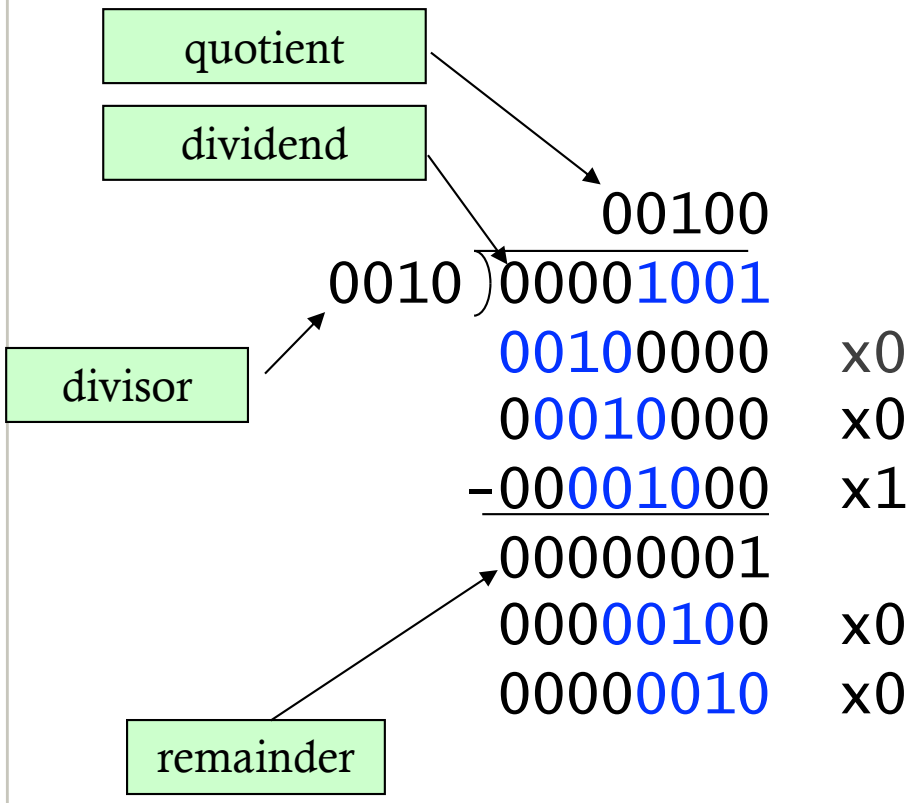


- Computer "restore division"
  - Always subtract divisor bits from dividend
  - Check subtraction result
    - if remainder goes < 0, add divisor back to restore dividend (set 0 in quotient)
    - Otherwise set 1 in quotient and move on

Hardware Division

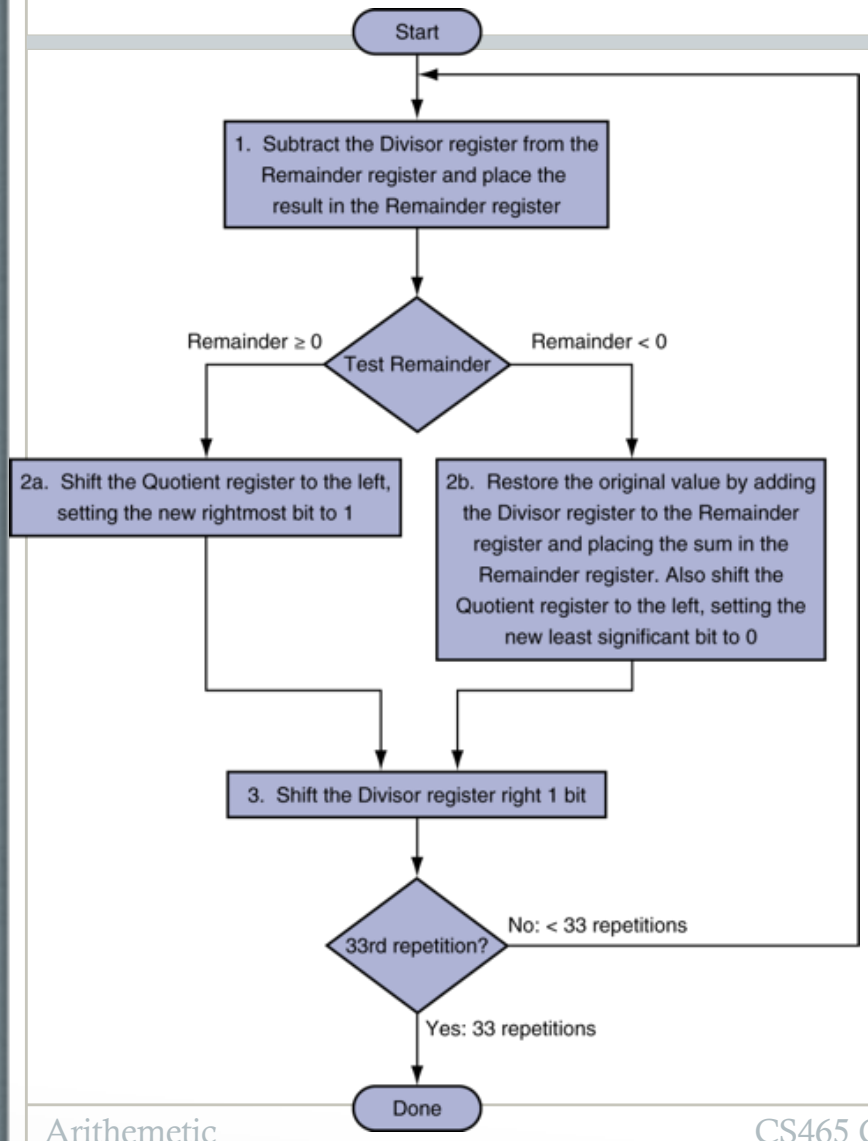
- Basic elements we need
  - Align divisor with the different dividend bits for subtraction
    - Again, we can shift contents of a register
  - Subtraction / Addition
    - We have basic ALU that can do both
  - Check whether a value is negative or not

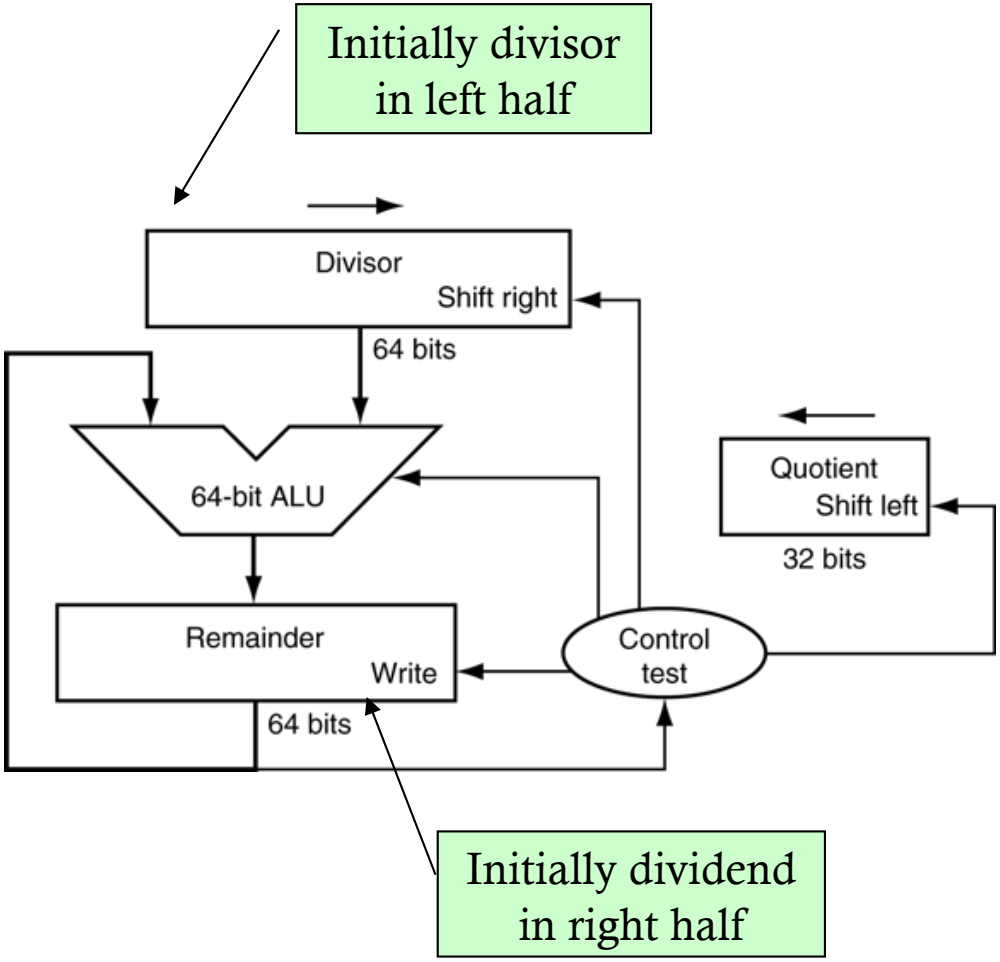
Division



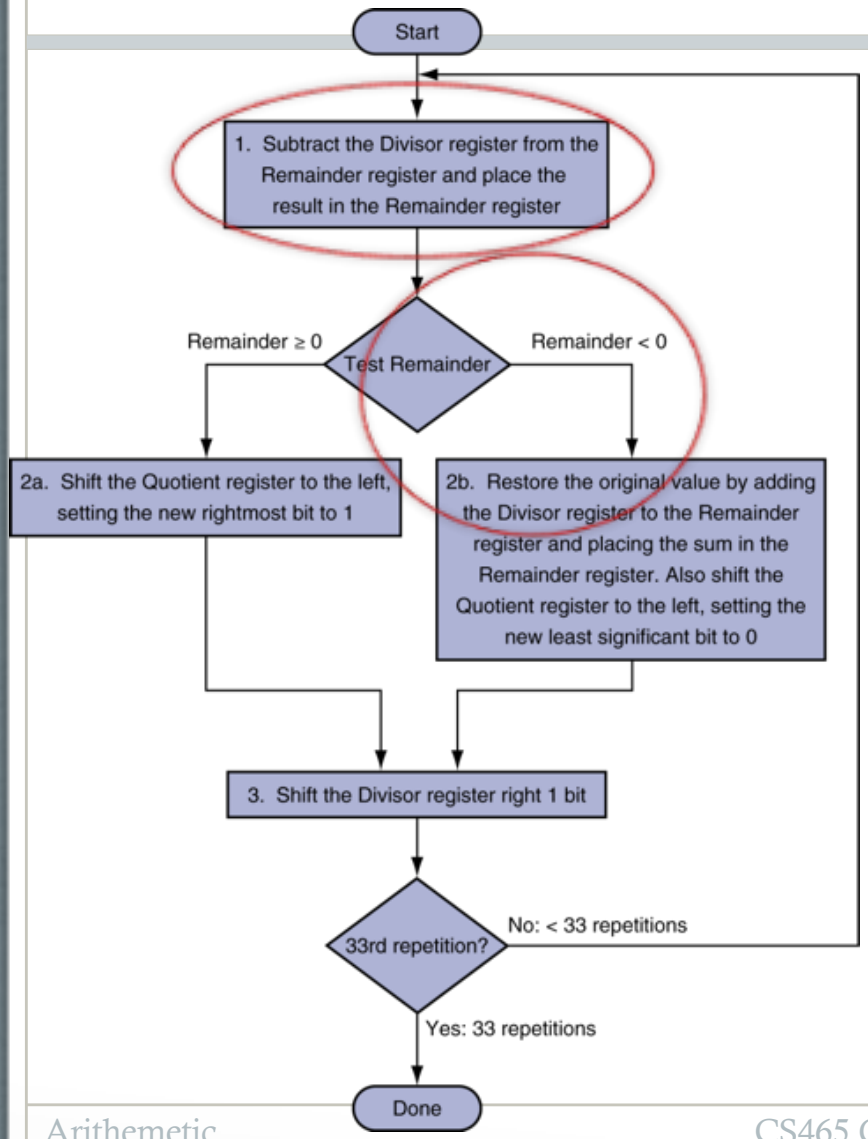
- Division of two 4-bit unsigned integers
  - One register for dividend
  - One register for divisor
- 1001/0010

Divide Hardware

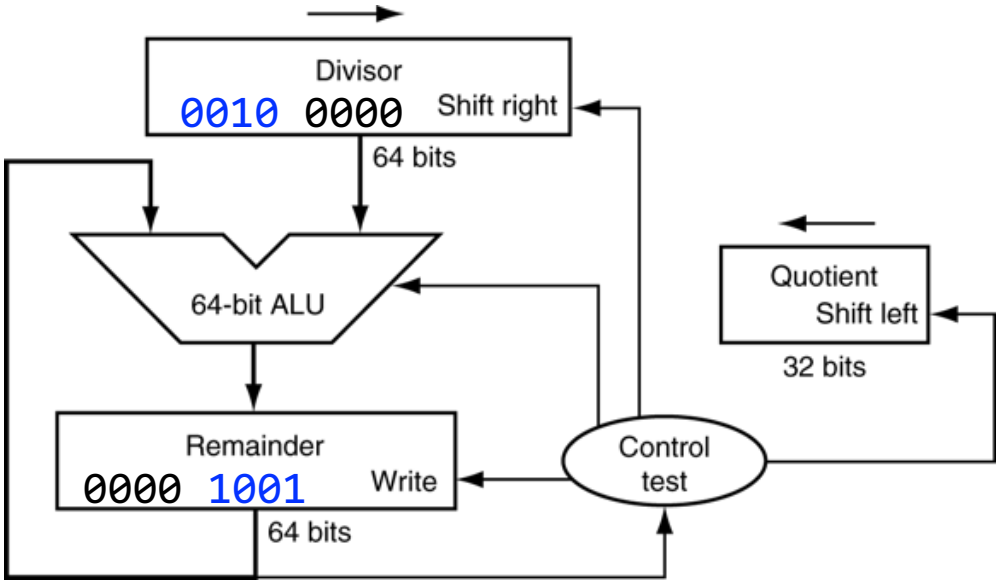




Divide Example

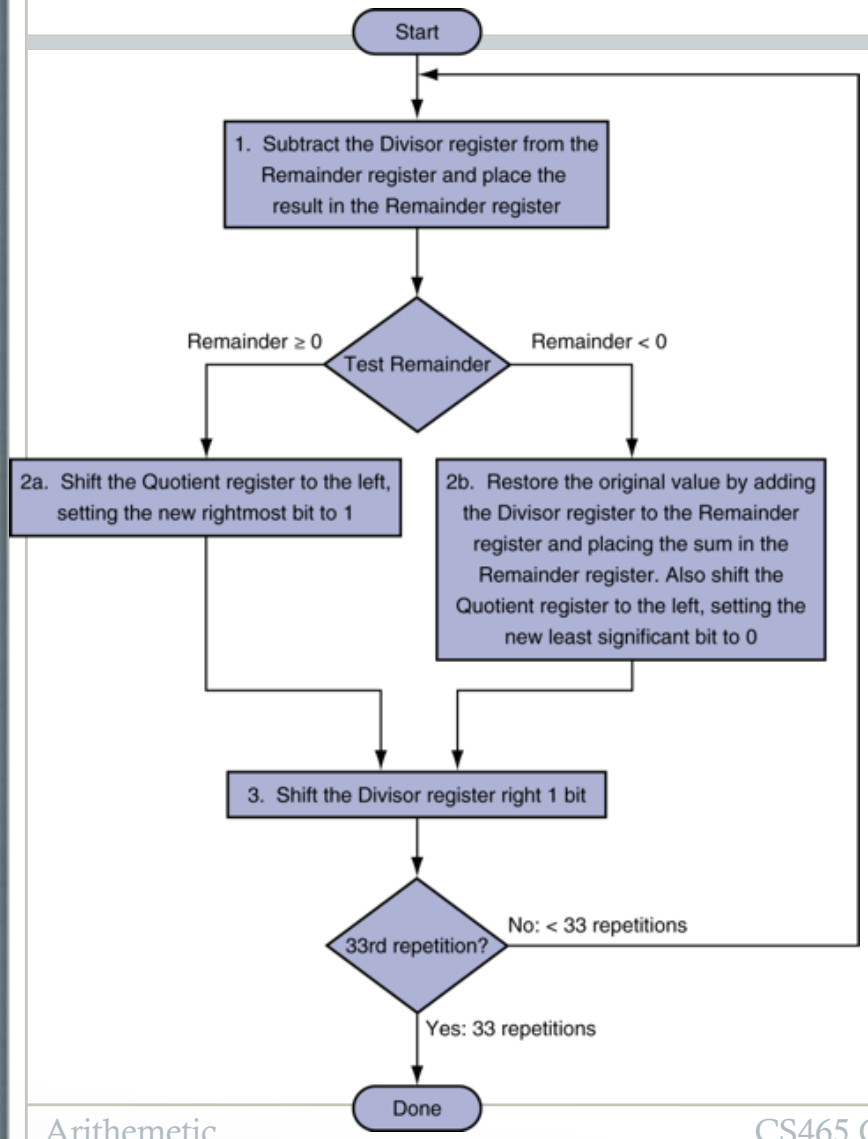


`1001 / 0010`

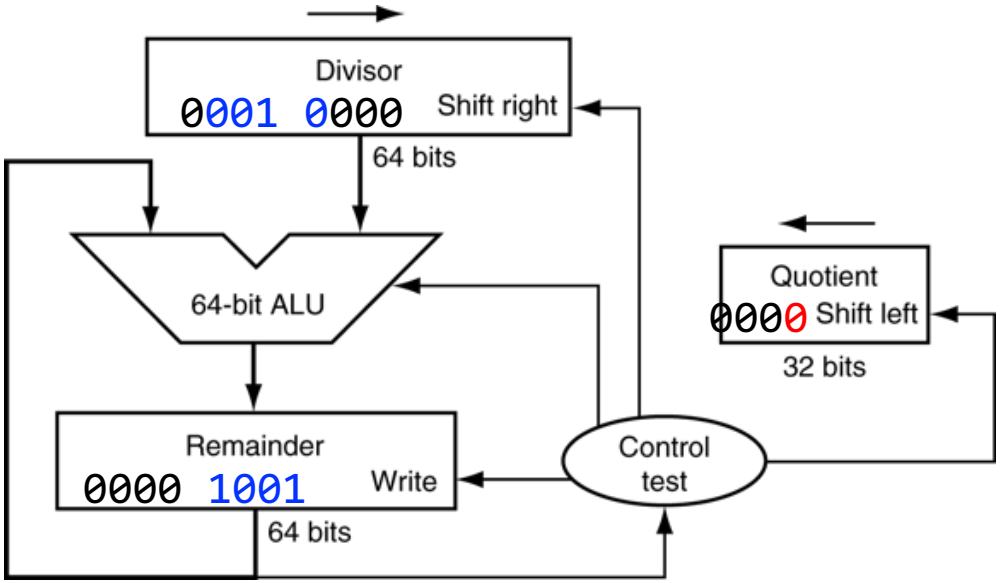


- Divisor initially upper half
- Remainder initially dividend

Divide Example

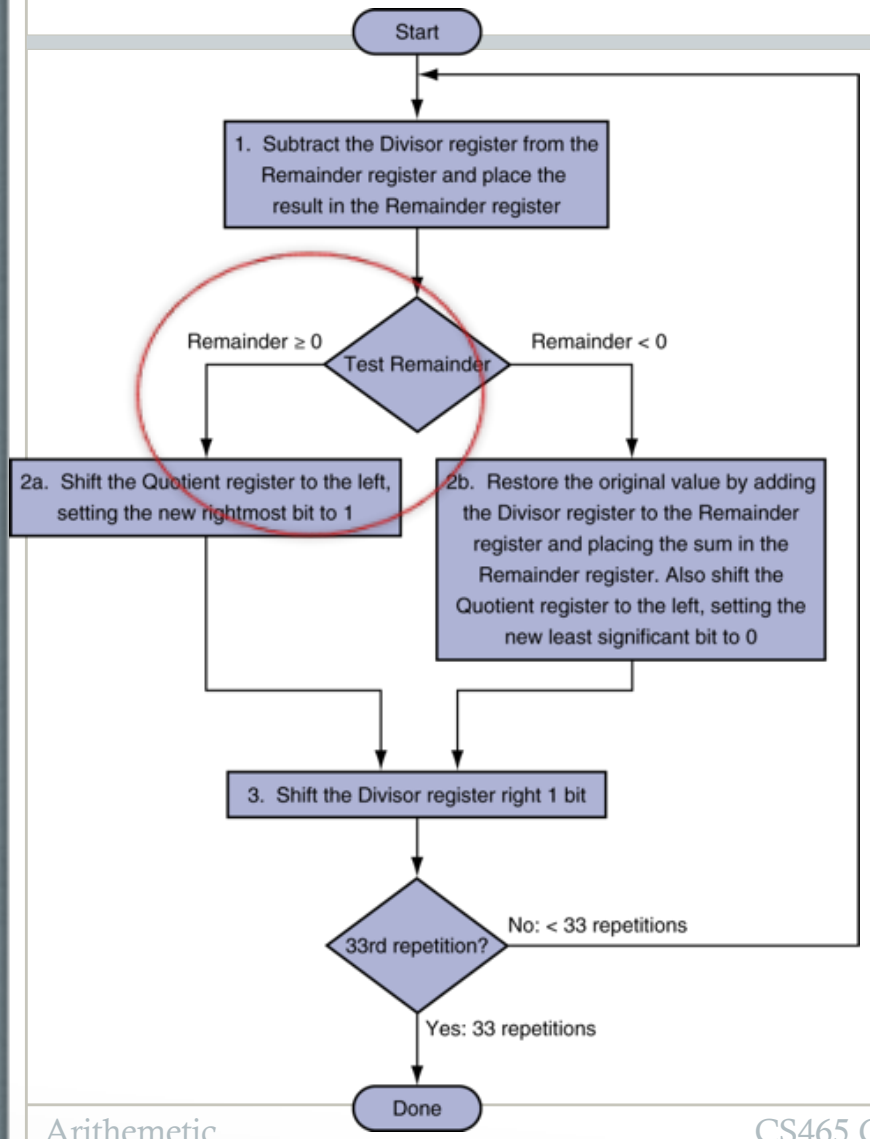



```
1001 / 0010
```

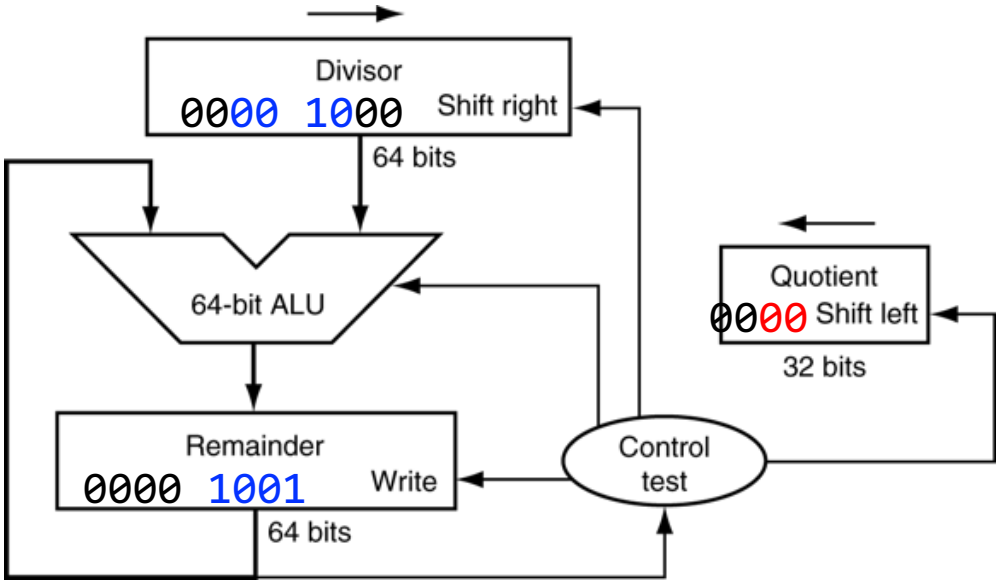


- Set 0 for lsb of quotient; recover remainder
- Shift divisor right
- Repeat …

Divide Example

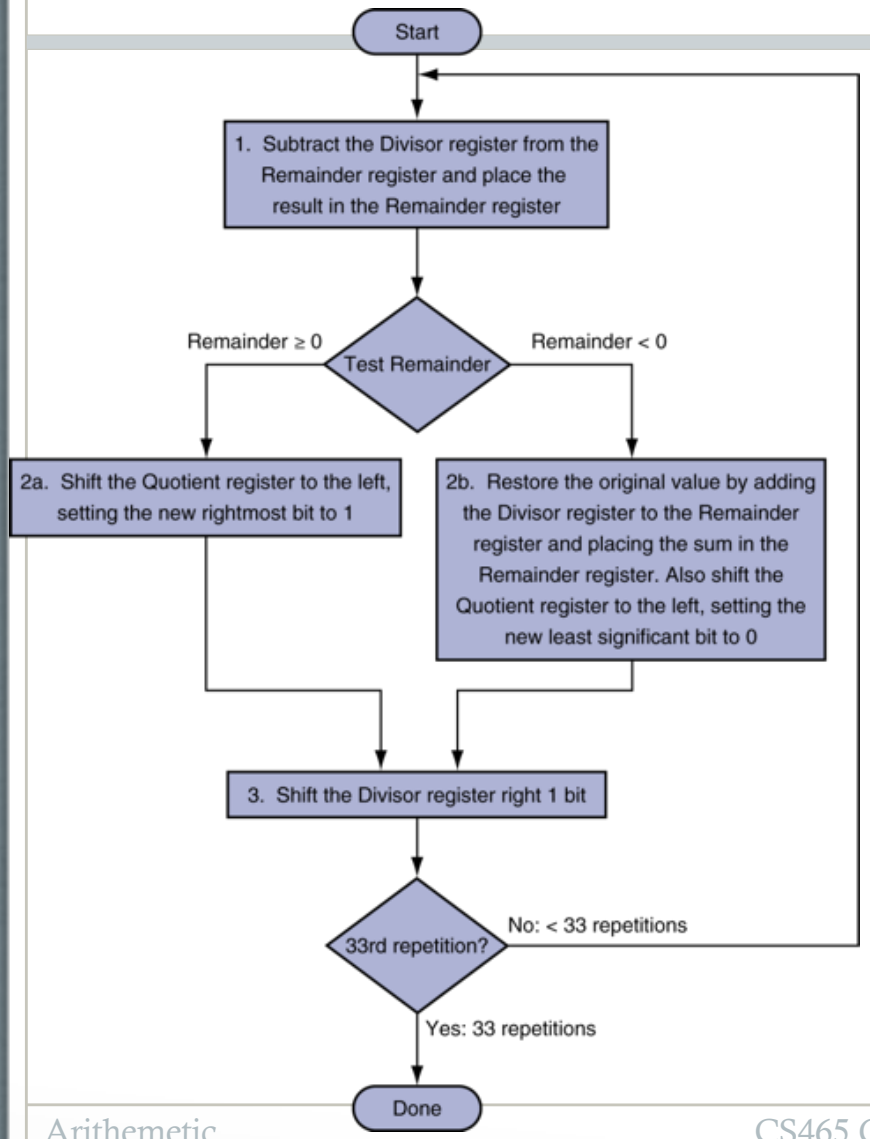


`1001 / 0010`

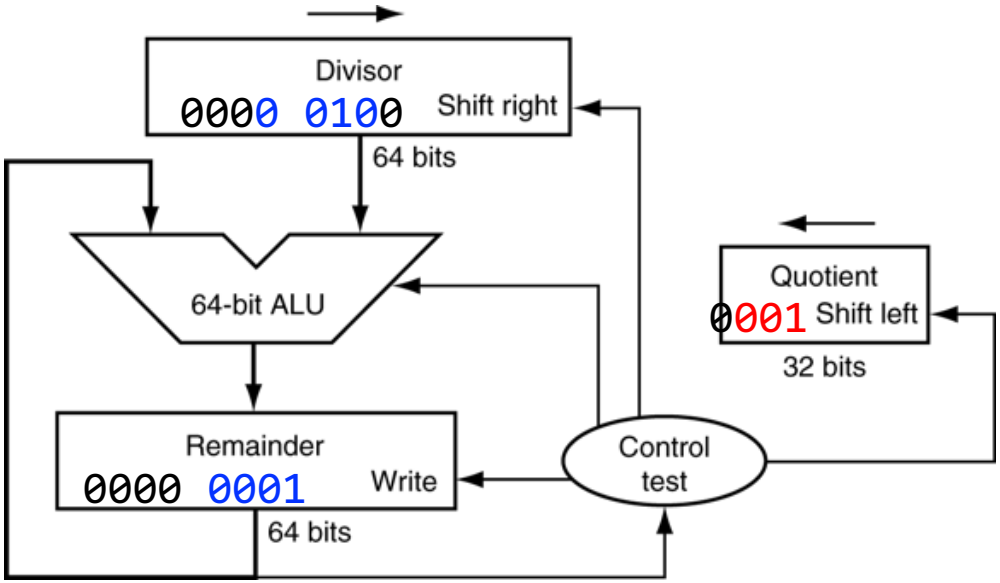


- Set 0 for lsb of quotient; recover remainder
- Shift divisor right
- Repeat …

Divide Example

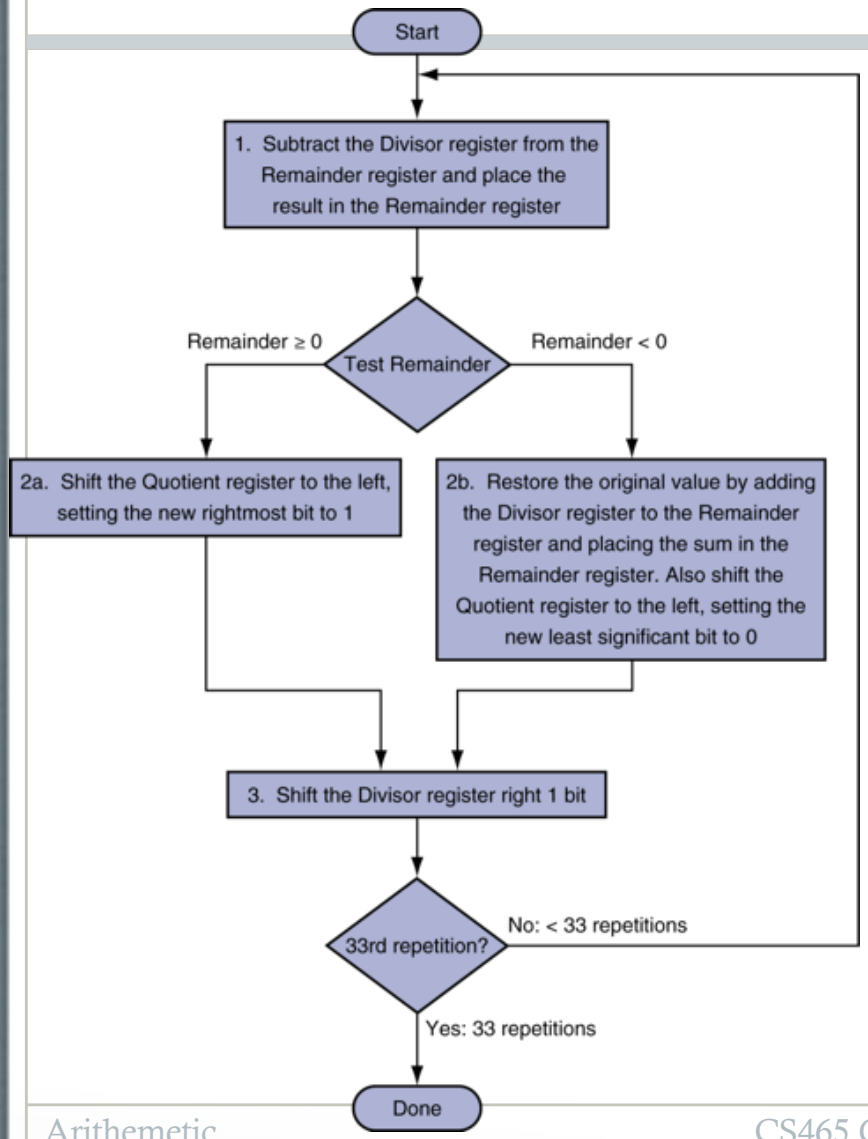


`1001 / 0010`



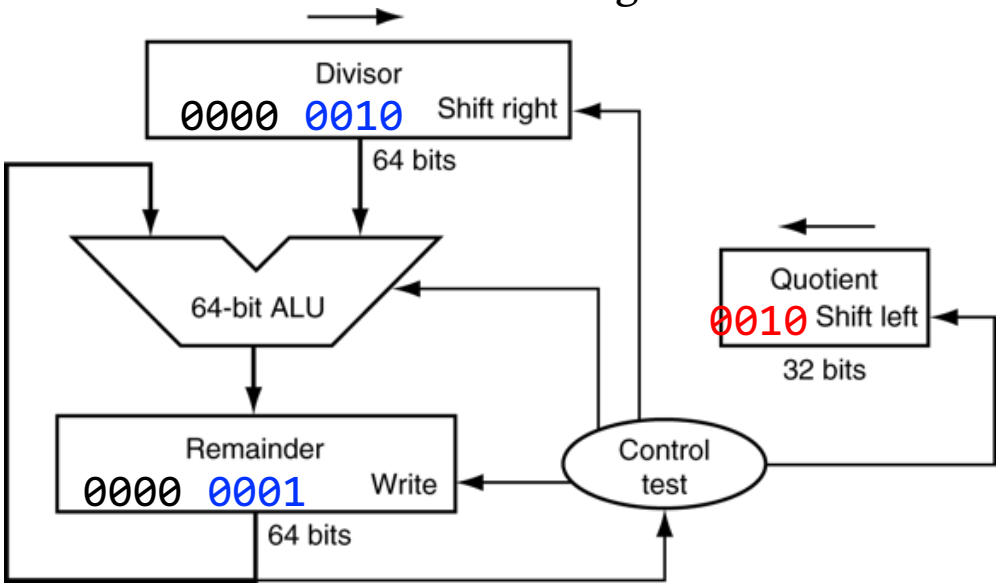
- Set 1 for lsb of quotient; remainder updated
- Shift divisor right
- Repeat…

Divide Example



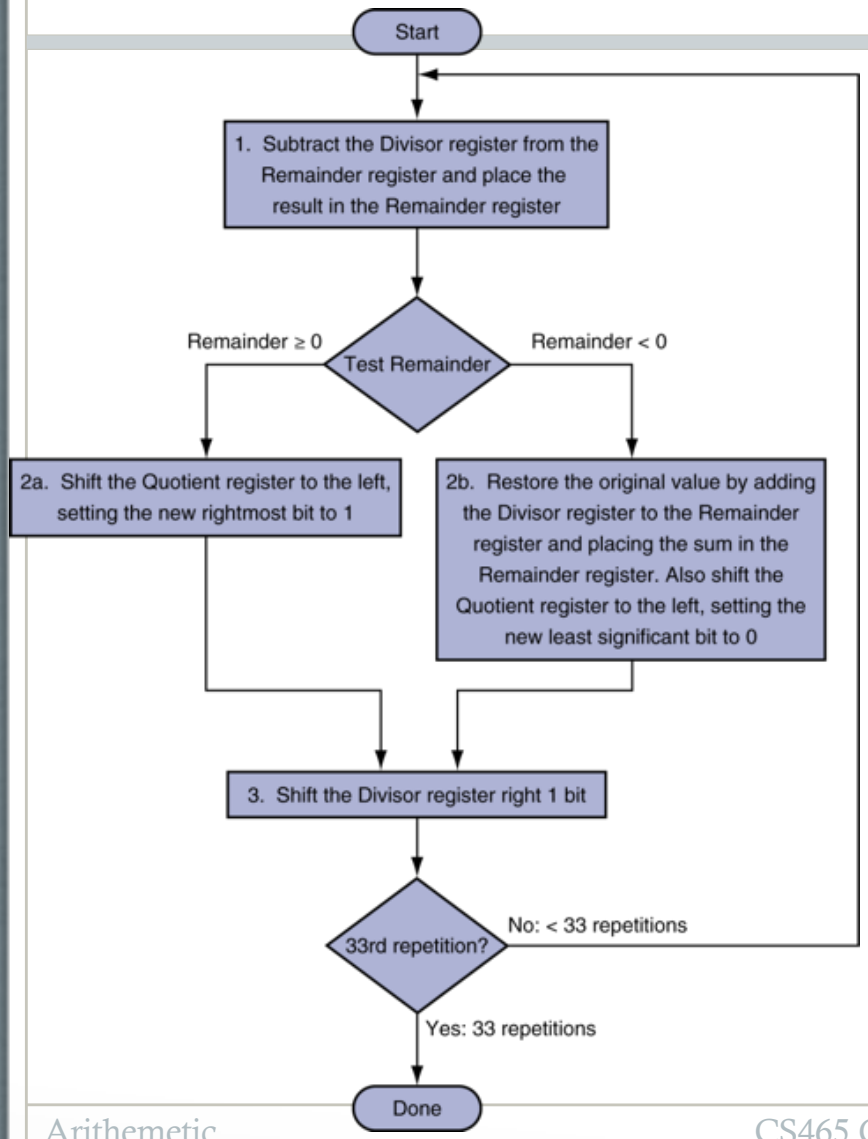
`1001 / 0010`

- After 4 repetitions, divisor is aligned with dividend
- Need another round of checking

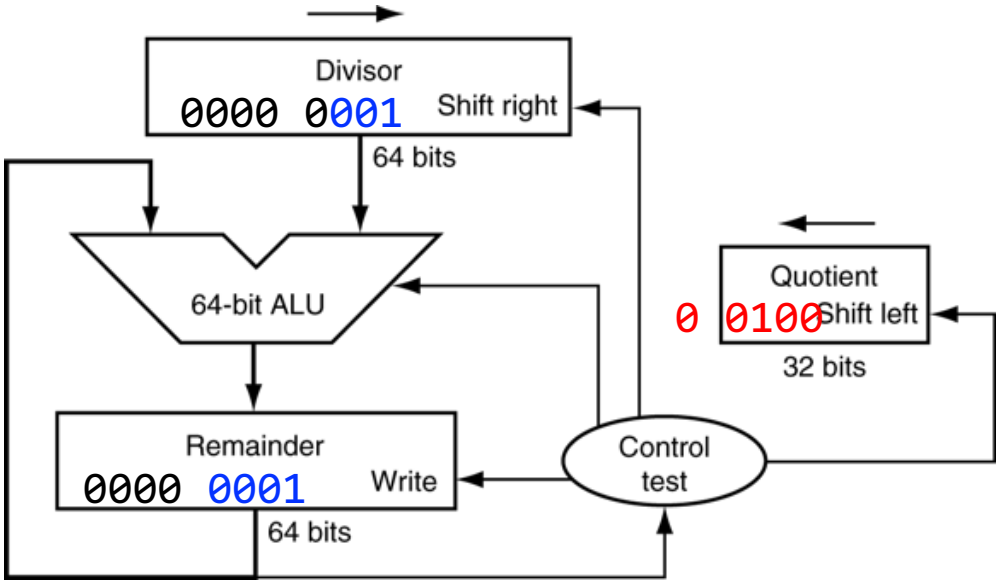


- Set 0 for lsb of quotient; recover remainder
- Shift divisor right
- Repeat?

Divide Example




```
1001 / 0010
```



- Final result after 5 rounds of repetition
  - Quotient
  - Remainder

Division Example

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 Div right	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 Div right	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 Div right	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 Div right	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 Div right	0011	0000 0001	0000 0001

FIGURE 3.10 Division example using the algorithm in Figure 3.9. The bit examined to determine the next step is circled in color.

Signed Division

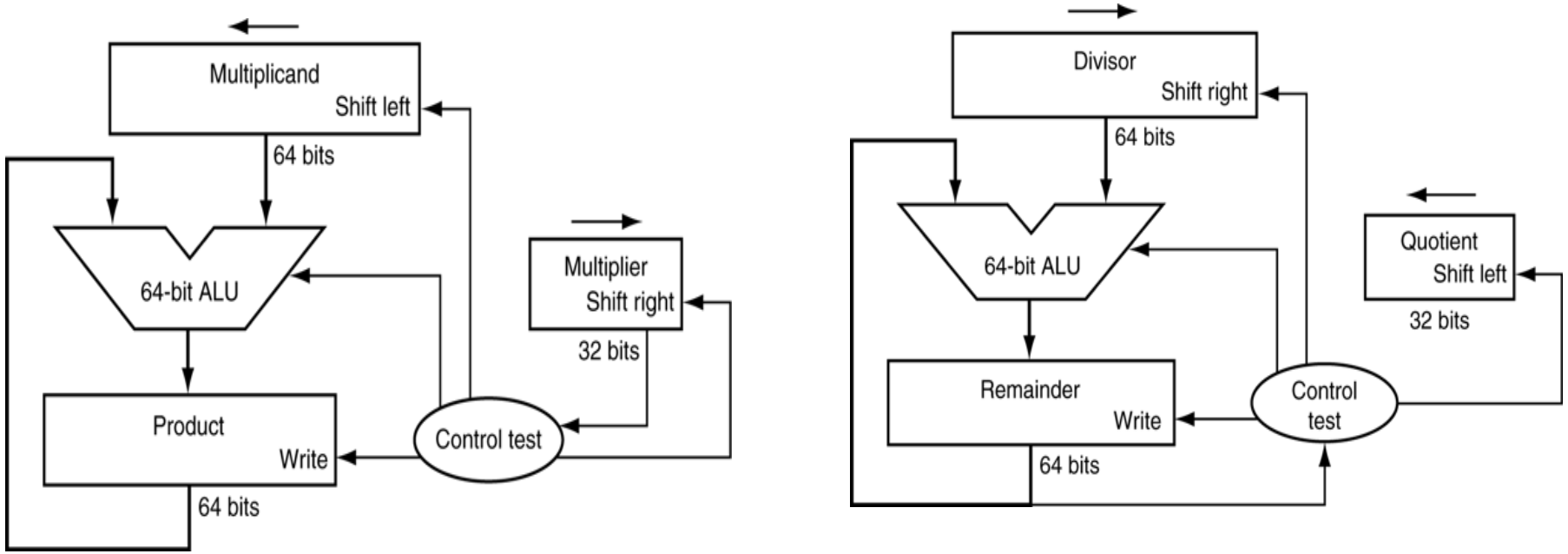
- Simplest approach: remember signs, make non-negative, and complement quotient and remainder if necessary
  - Satisfy Dividend =Quotient x Divisor + Remainder
  - Note: Dividend and Remainder must have same sign
  - Note: Quotient negated if Divisor sign & Dividend sign disagree
  - E.g., –7 ÷ 2 = –3, remainder = –1
    - Quotient = -4 and remainder =1 : incorrect

*This is the common approach implemented by hardware. Each language may have its own definition.

MIPS Division

- Use HI/LO registers for result
  - HI: 32-bit remainder
  - LO: 32-bit quotient
- Instructions
  - `div rs, rt  /  divu rs, rt`
  - No overflow or divide-by-0 checking
    - Software must perform checking if required
  - Use `mfhi`, `mflo` to access result

Multiply/Divide Hardware



Roadmap

- Operations on integers
  - Addition and subtraction
  - Multiplication
  - Division
- Floating-point numbers
  - Representation (check the review material)
  - Addition and multiplication