

## **Chapter 3**

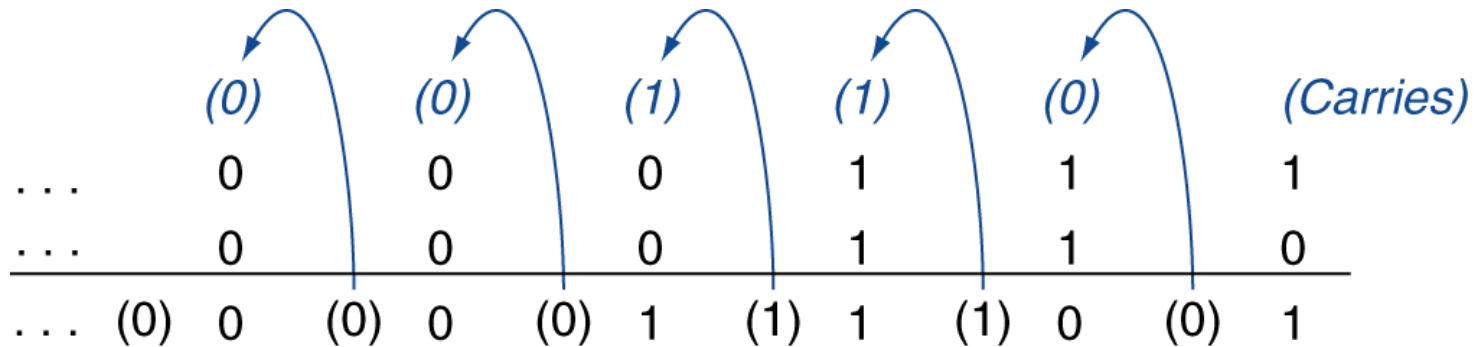
### **Arithmetic for Computers**

# Arithmetic for Computers

- Operations on integers
  - ◆ Addition and subtraction
  - ◆ Multiplication and division
  - ◆ Dealing with overflow
- Floating-point real numbers
  - ◆ Representation and operations

# Integer Addition

## ■ Example: $7 + 6$



## ■ Overflow please write down the

- 8-bit signed integer addition:
- $0100\ 0000_{\text{bin}} + 0100\ 0000_{\text{bin}} = ?$
- $1000\ 0000_{\text{bin}} + 1000\ 0000_{\text{bin}} = ?$
- $0100\ 0000_{\text{bin}} + 1100\ 0000_{\text{bin}} = ?$
- $1100\ 0000_{\text{bin}} + 1100\ 0000_{\text{bin}} = ?$

Please write down the equations in binary and decimal.

# Overflow

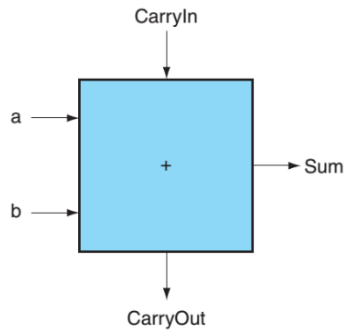
## ■ Examples in last page:

- ◆ 8-bit signed integer range:  $-128 \sim 127$
- ◆  $0100\ 0000_{\text{bin}} + 0100\ 0000_{\text{bin}} = 1000\ 0000_{\text{bin}}$   $64 + 64 = -128$  **Overflow**
- ◆  $1000\ 0000_{\text{bin}} + 1000\ 0000_{\text{bin}} = (1)0000\ 0000_{\text{bin}}$   $-128 + (-128) = 0$  **Overflow**
- ◆  $0100\ 0000_{\text{bin}} + 1100\ 0000_{\text{bin}} = (1)0000\ 0000_{\text{bin}}$   $64 + (-64) = 0$  **No overflow**
- ◆  $1100\ 0000_{\text{bin}} + 1100\ 0000_{\text{bin}} = (1)1000\ 0000_{\text{bin}}$   $-64 + (-64) = -128$  **No overflow**

## ■ Overflow if result out of range

- ◆ no overflow, if adding +ve and -ve operands
- ◆ Overflow, if
  - Adding two +ve operands, get -ve operand
  - Adding two -ve operands, get +ve operand

# 1-bit adder

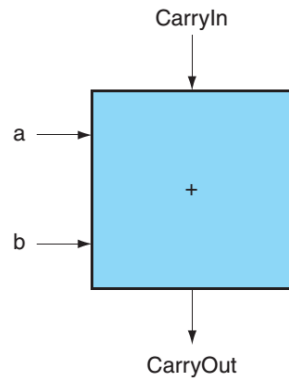


$$\text{Sum} = (a \cdot \overline{b} \cdot \overline{\text{CarryIn}}) + (\overline{a} \cdot b \cdot \overline{\text{CarryIn}}) + (\overline{a} \cdot \overline{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$$

$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b)$$

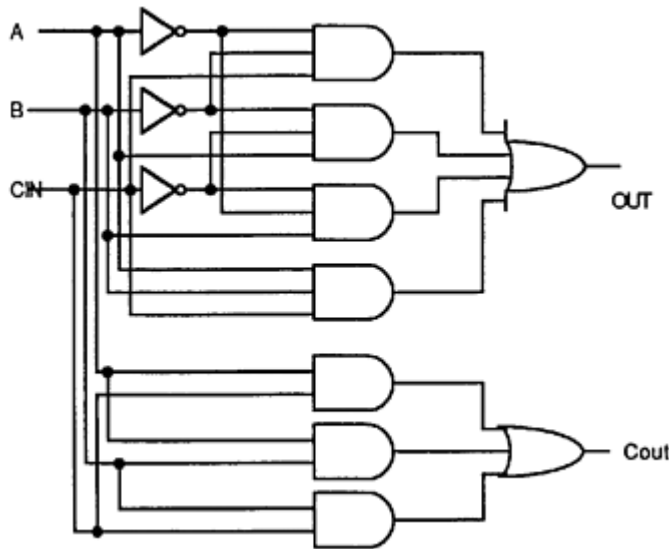
Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00_{\text{two}}$
0	0	1	0	1	$0 + 0 + 1 = 01_{\text{two}}$
0	1	0	0	1	$0 + 1 + 0 = 01_{\text{two}}$
0	1	1	1	0	$0 + 1 + 1 = 10_{\text{two}}$
1	0	0	0	1	$1 + 0 + 0 = 01_{\text{two}}$
1	0	1	1	0	$1 + 0 + 1 = 10_{\text{two}}$
1	1	0	1	0	$1 + 1 + 0 = 10_{\text{two}}$
1	1	1	1	1	$1 + 1 + 1 = 11_{\text{two}}$

# 1-bit Adder

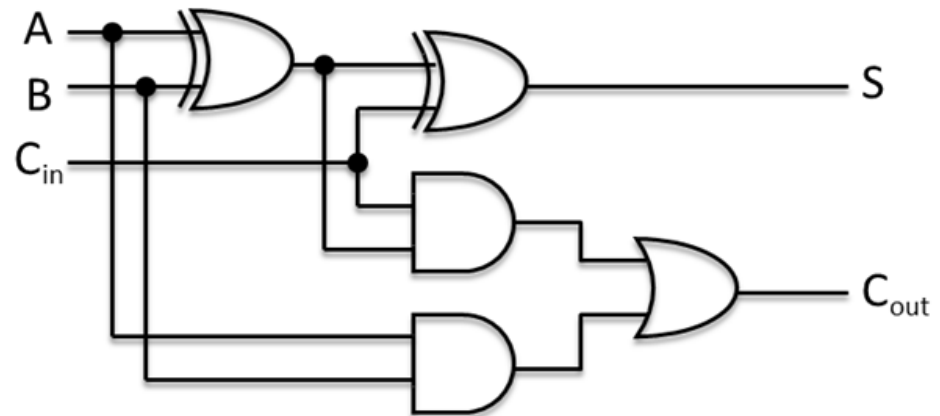


$$\text{Sum} = (a \cdot \bar{b} \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot b \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot \bar{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$$

$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b)$$



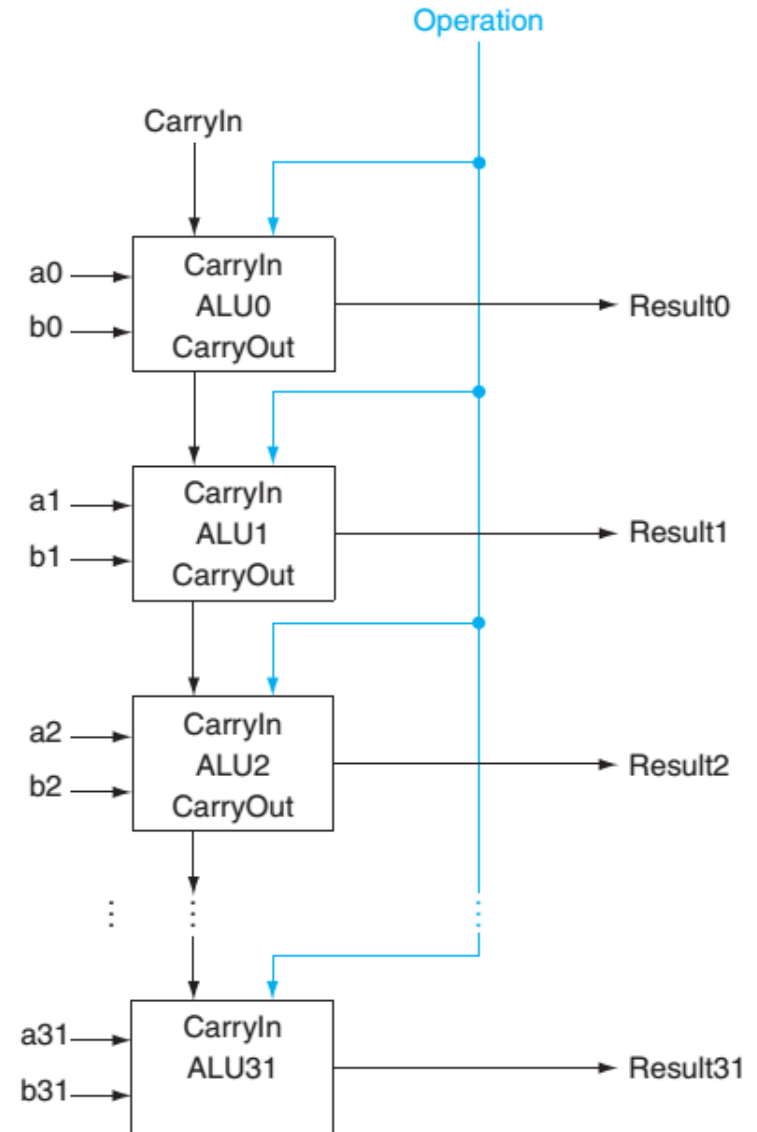
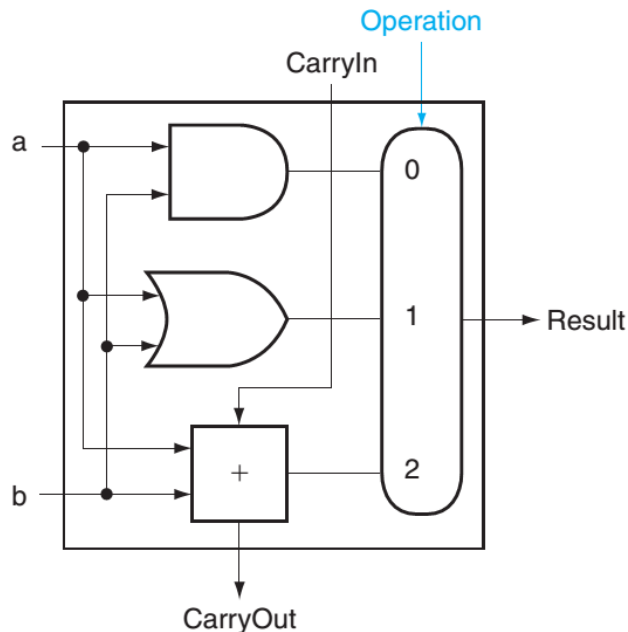
1-bit adder – version 1



1-bit adder – version 2

# 1-bit ALU

- ALU: arithmetic logical unit
- 1-bit ALU and 32-bit ALU
  - ◆ If  $op = 0$ ,  $o = a \& b$  (and)
  - ◆ If  $op = 1$ ,  $o = a \mid b$  (or)
  - ◆ If  $op = 2$ ,  $o = a + b$  (add)



# Integer Subtraction

- Add negation of second operand
- Example:  $7 - 6 = 7 + (-6)$

+7:	0000 0000 ... 0000 0111
-6:	1111 1111 ... 1111 1010
<hr/>	
+1:	0000 0000 ... 0000 0001

- Overflow if result out of range
  - ◆ No overflow, if subtracting two +ve or two -ve operands
  - ◆ Overflow, if:
    - Subtracting +ve from -ve operand, and the result sign is 0 (+ve)
    - Subtracting -ve from +ve operand, and the result sign is 1 (-ve)

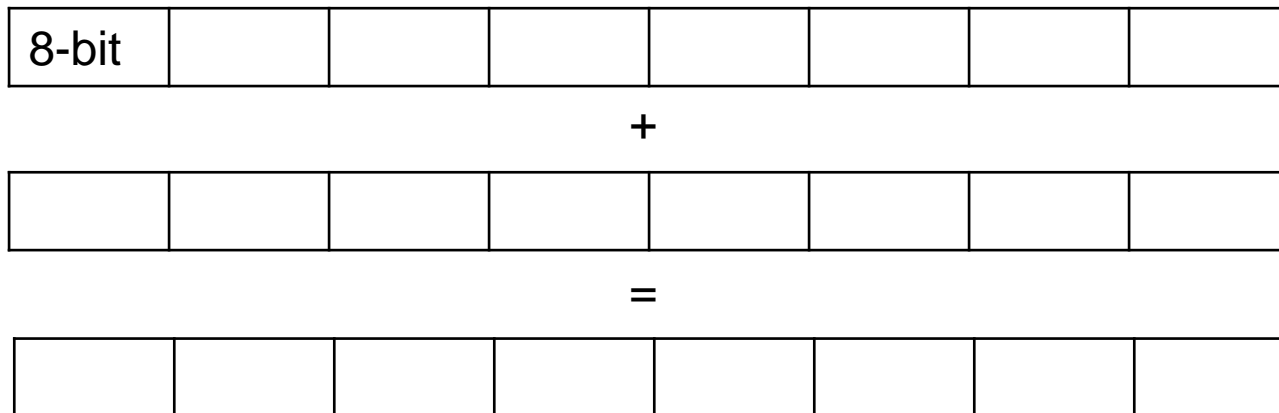


# Dealing with Overflow

- Some languages (e.g., C) ignore overflow
  - ◆ Use MIPS `addu`, `addiu`, `subu` instructions
- Other languages (e.g., Ada, Fortran) require raising an exception
  - ◆ Use MIPS `add`, `addi`, `sub` instructions
  - ◆ On overflow, invoke exception handler
    - Save PC in exception program counter (EPC) register
    - Jump to predefined handler address
    - `mfc0` (move from coprocessor reg) instruction can retrieve EPC value, to return after corrective action
- Note: `addiu`: “u” means it doesn’t generate overflow exception, but the immediate can be a signed number

# Arithmetic for Multimedia-SIMD

- Graphics and media processing operates on vectors of 8-bit and 16-bit data
  - ◆ Use 64-bit adder, with partitioned carry chain
    - Operate on  $8 \times 8$ -bit,  $4 \times 16$ -bit, or  $2 \times 32$ -bit vectors
  - ◆ SIMD (single-instruction, multiple-data)
  - ◆ `addv rd, rs, rt`

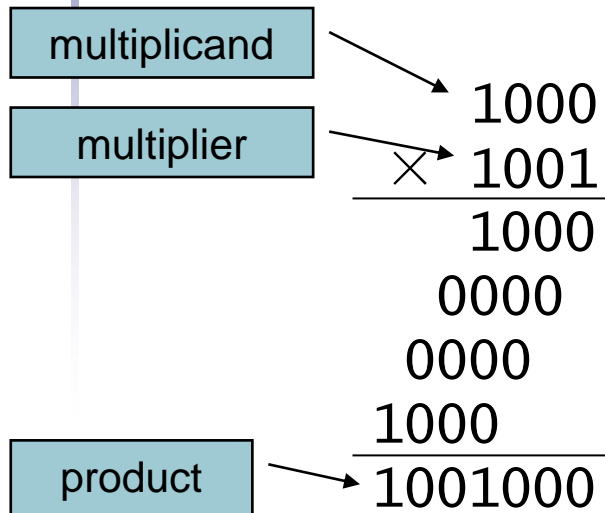


# Arithmetic for Multimedia – Saturating Operation

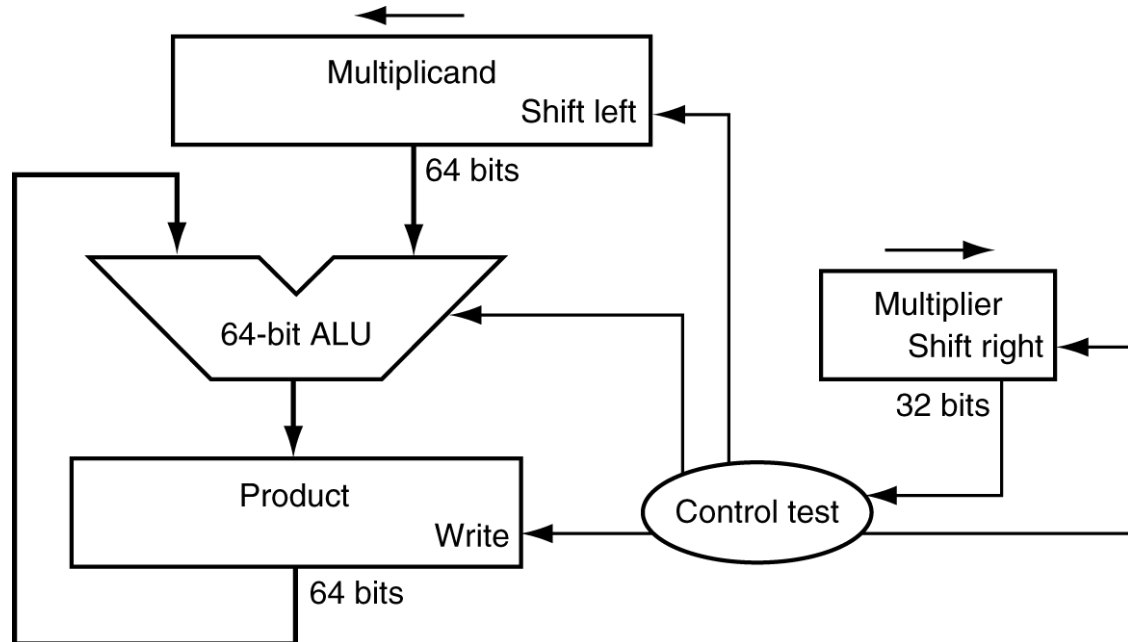
- Pixel representation:
  - ◆ RGB, each using 8 bit to represent, range: 0-255
- Saturating operations
  - ◆ On overflow, result is largest representable value
    - Instead of 2s-complement modulo arithmetic
  - ◆ E.g., change the volume and brightness in audio or video
  - ◆ Original brightness of three pixels: 100, 150, 200, make them brighter by adding 100, the result should be 200, 250, 44? Or 200, 250, 255?

# Multiplication

- Start with long-multiplication approach

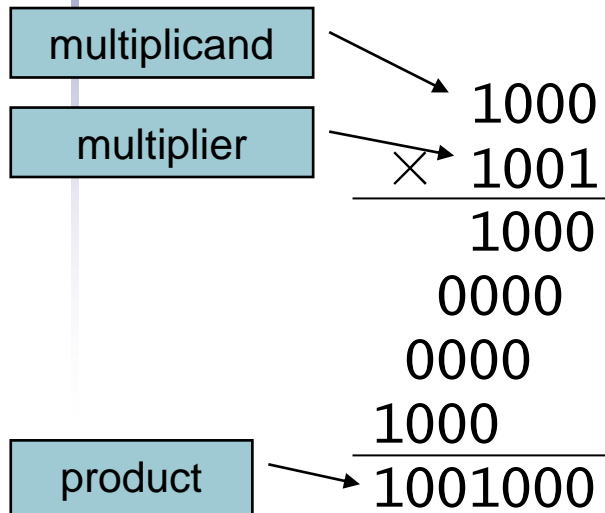


Length of product is  
the sum of operand  
lengths

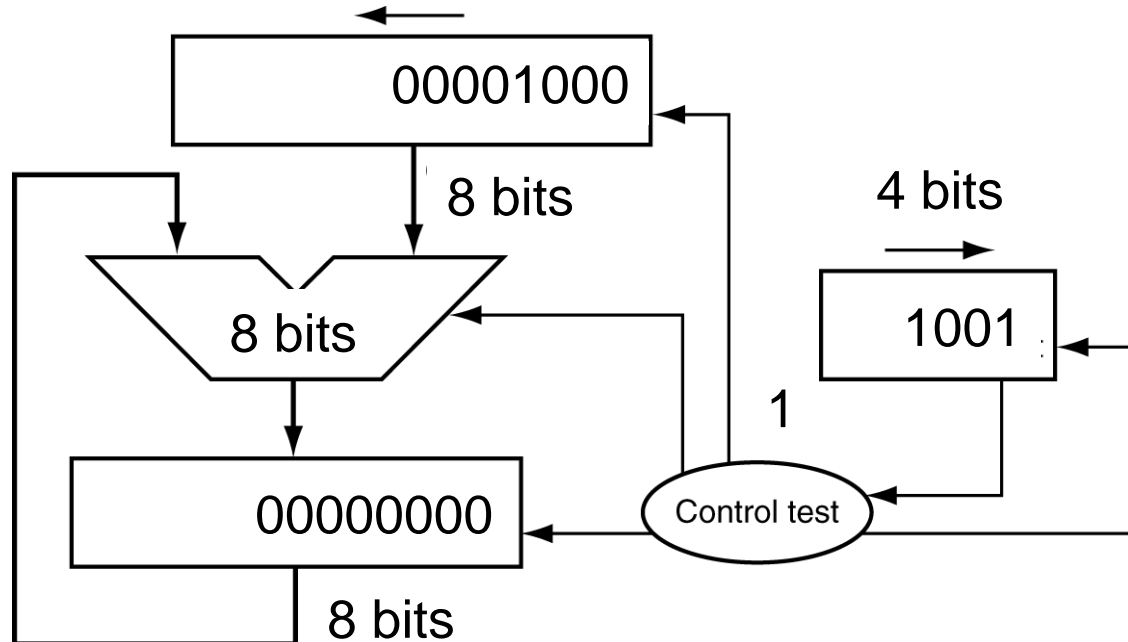


# Multiplication

- Start with long-multiplication approach

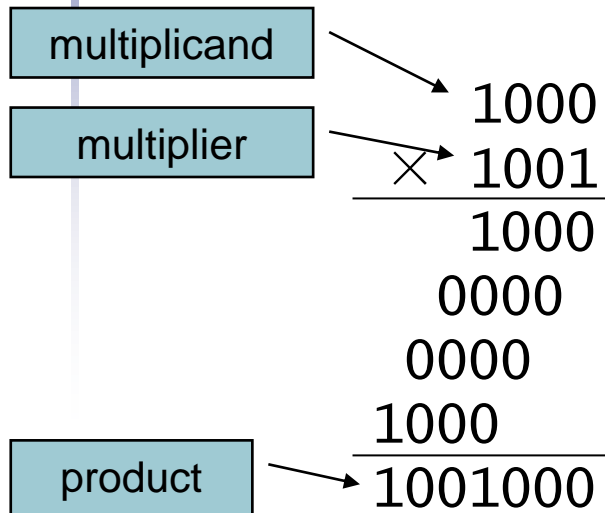


Length of product is the sum of operand lengths

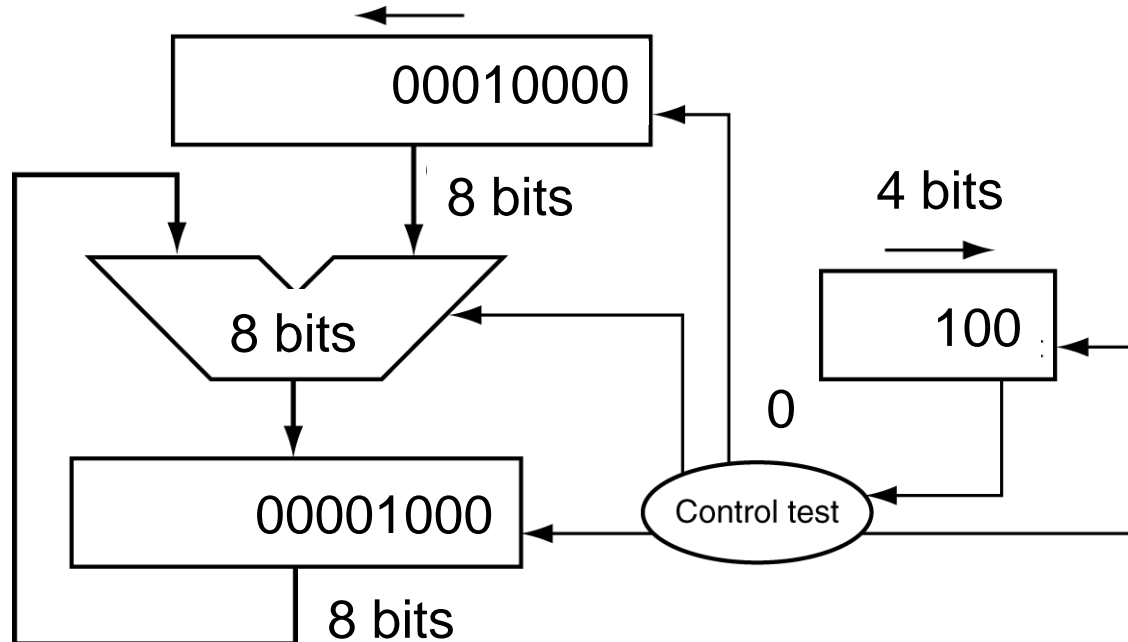


# Multiplication

- Start with long-multiplication approach

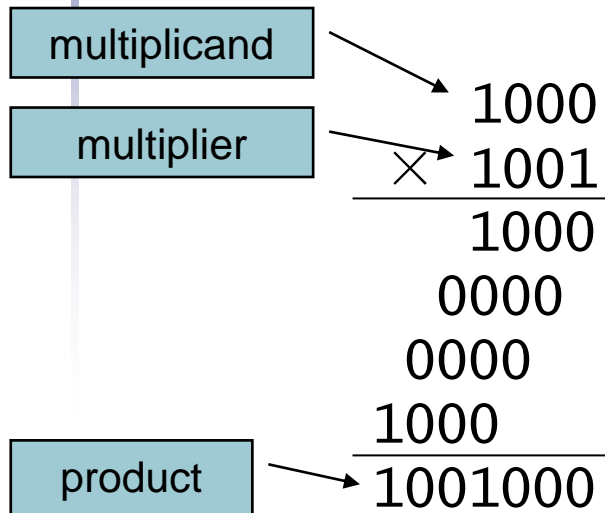


Length of product is the sum of operand lengths

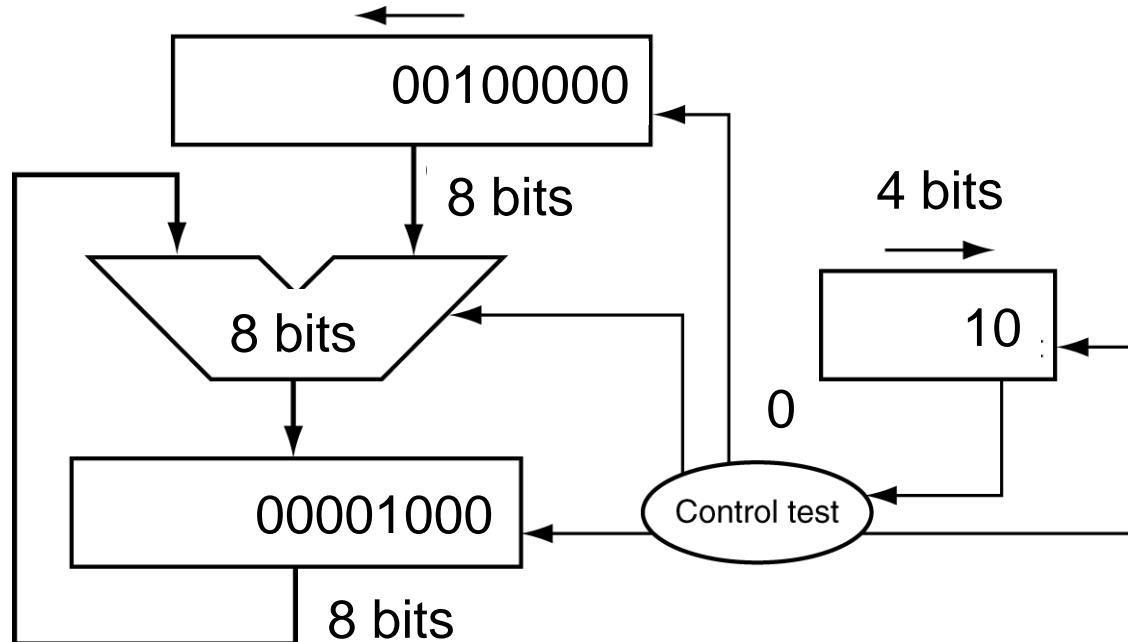


# Multiplication

- Start with long-multiplication approach

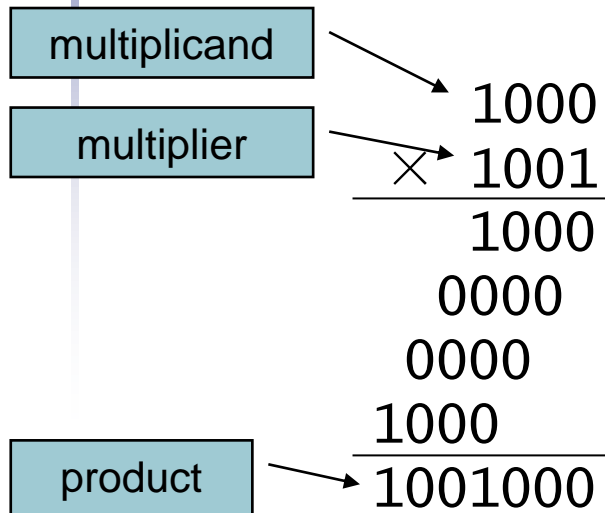


Length of product is the sum of operand lengths

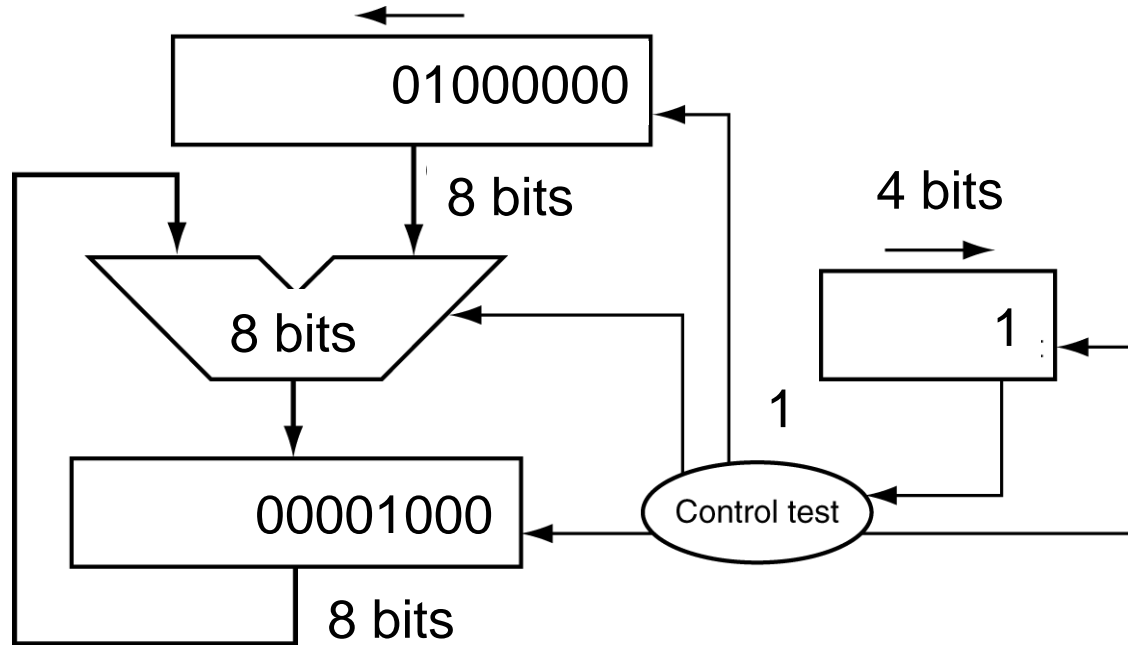


# Multiplication

- Start with long-multiplication approach



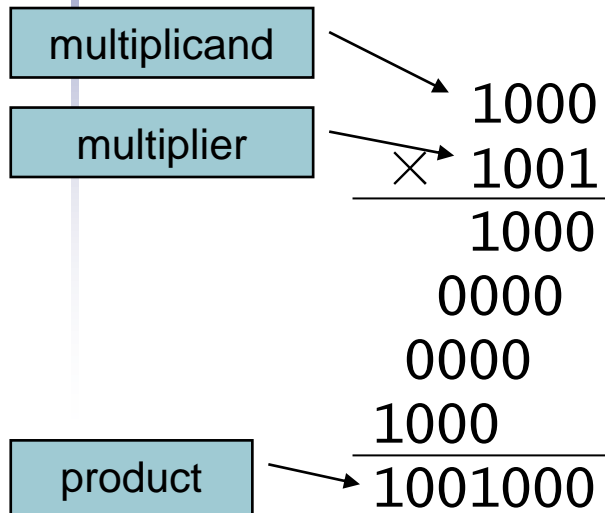
Length of product is the sum of operand lengths



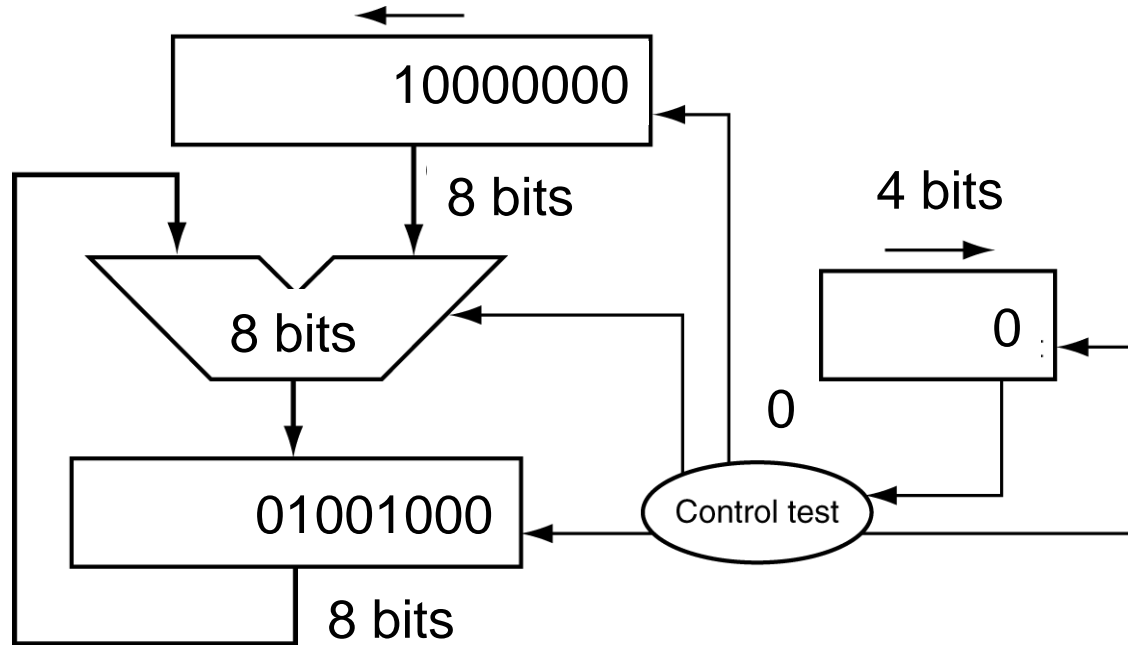


# Multiplication

- Start with long-multiplication approach

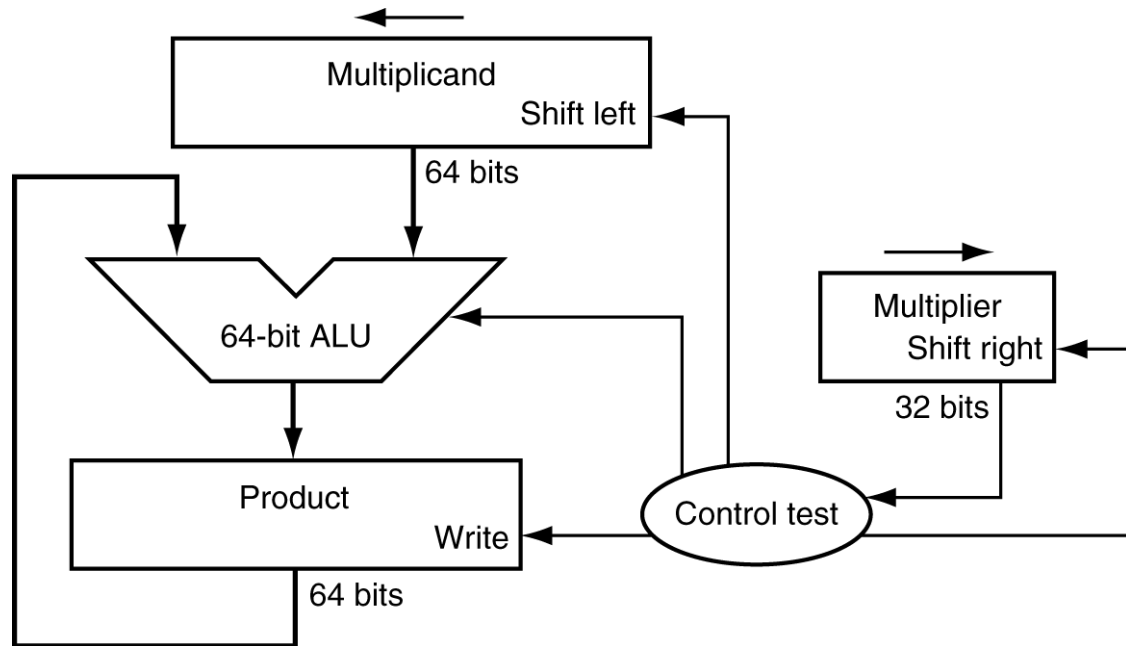
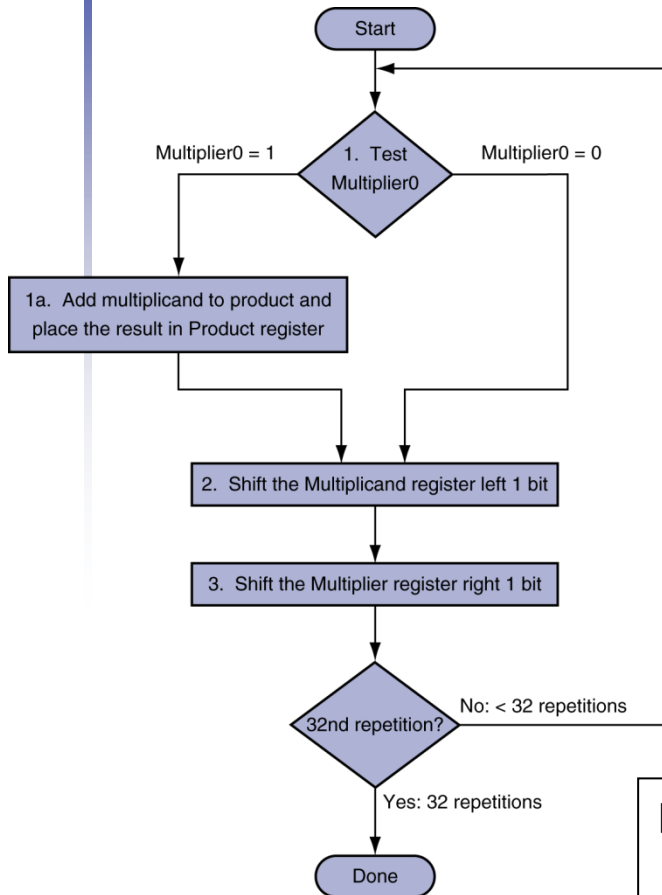


Length of product is  
the sum of operand  
lengths



**Finish!**

# Multiplication Hardware

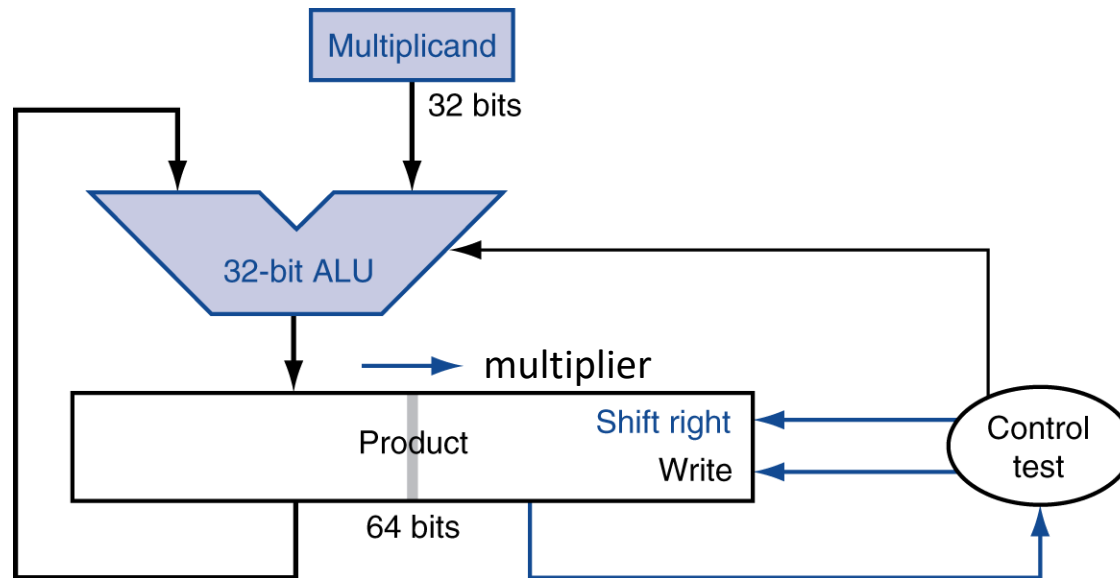


In every step:

- ✓ multiplicand is shifted
- ✓ next bit of multiplier is examined (also a shifting step)
- ✓ if this bit is 1, shifted multiplicand is added to the product

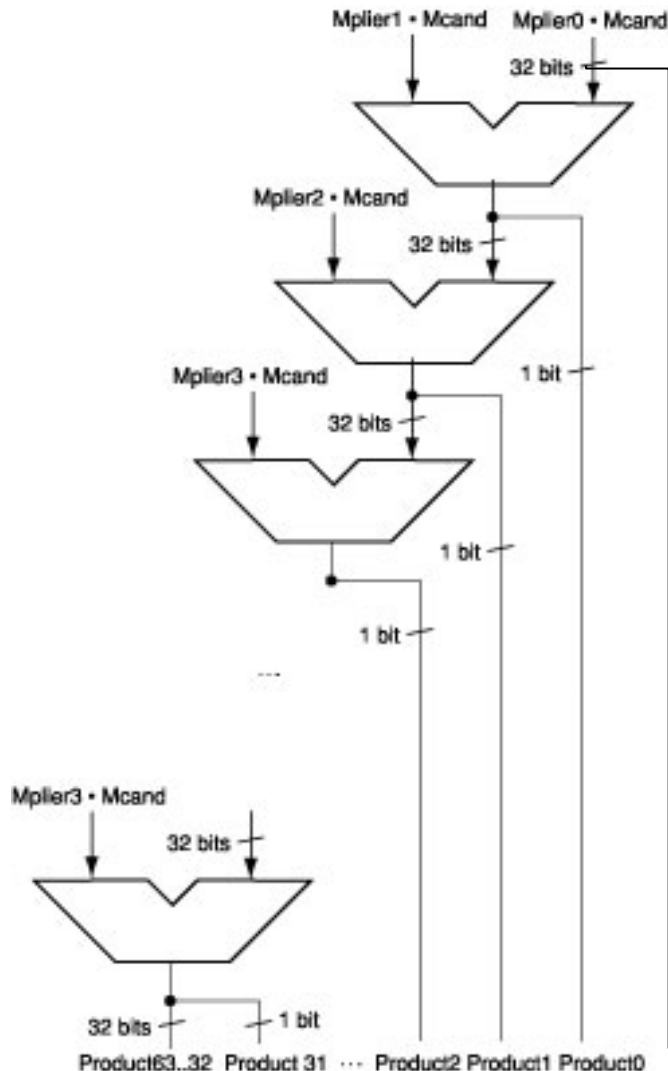
# Optimized Multiplier

- Perform steps in parallel: add/shift



- ✓ The multiplier is initially stored in the **right half** of product register
- ✓ check the 0<sup>th</sup> bit in Product register, if 1, add left half of product with multiplicand
- ✓ the sum keeps shifting right
- ✓ at every step, number of bits in product + multiplier = 64, hence, they share a single 64-bit register
- ✓ for signed multiplication, it also works

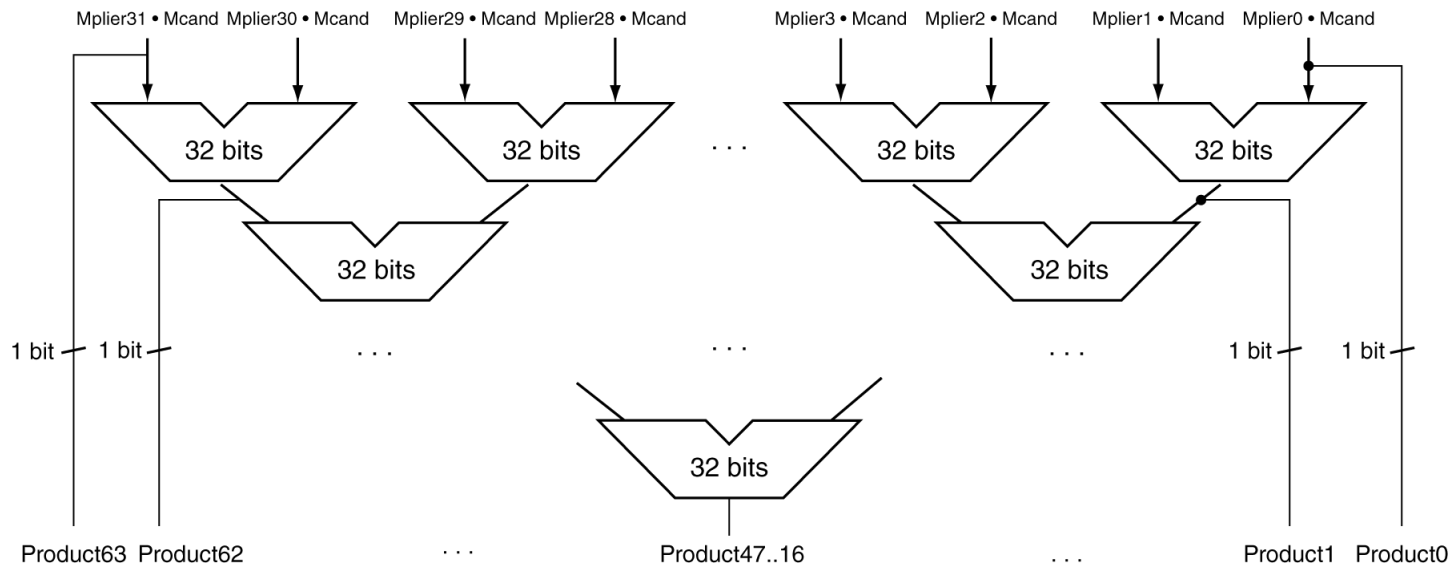
# Faster Multiplier



- The previous algorithm requires a clock to ensure that the earlier addition has completed before shifting
- This algorithm can quickly set up most inputs – it then has to wait for the result of each add to propagate down – faster because no clock is involved
- high transistor cost

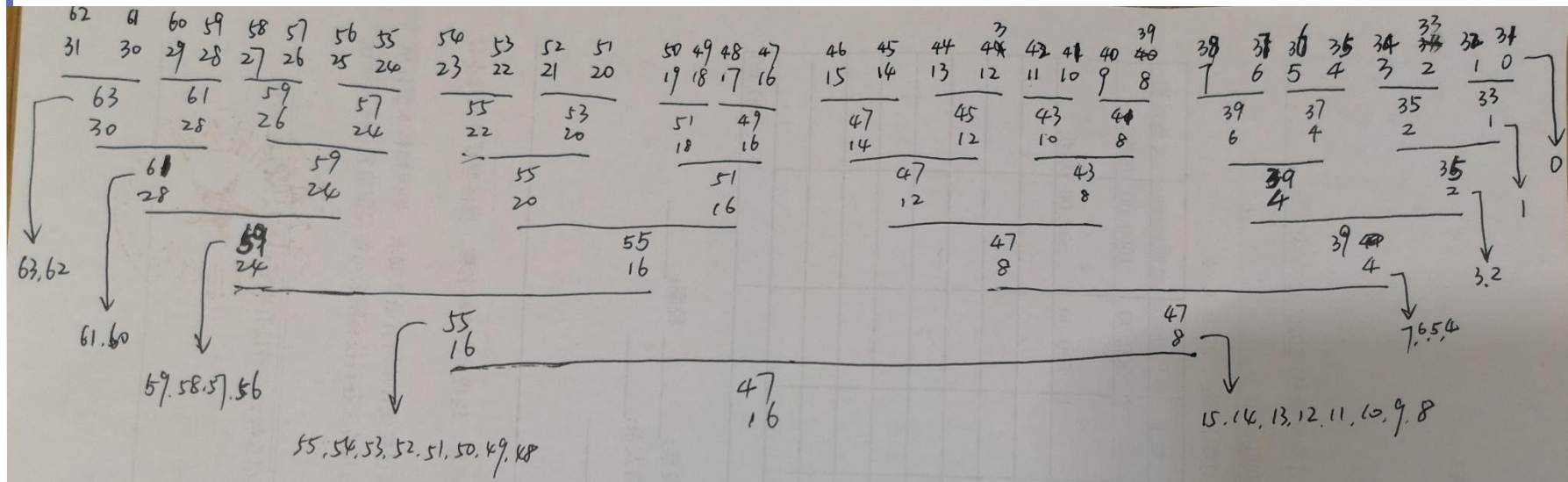
# Faster Multiplier

- Uses multiple pipelined adders
  - ◆ Cost/performance tradeoff



- Can be pipelined
  - ◆ Several multiplication performed in parallel

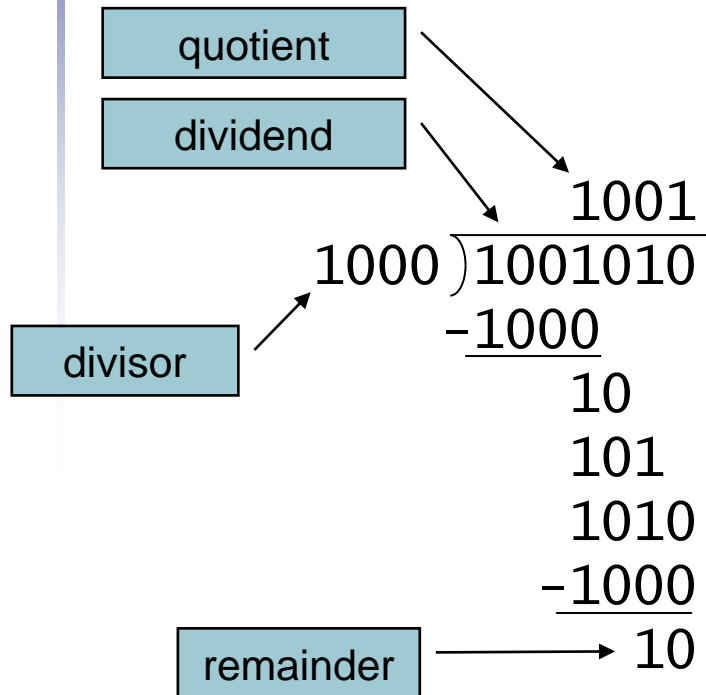
# 32-bit Faster Multiplier



# 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

# 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

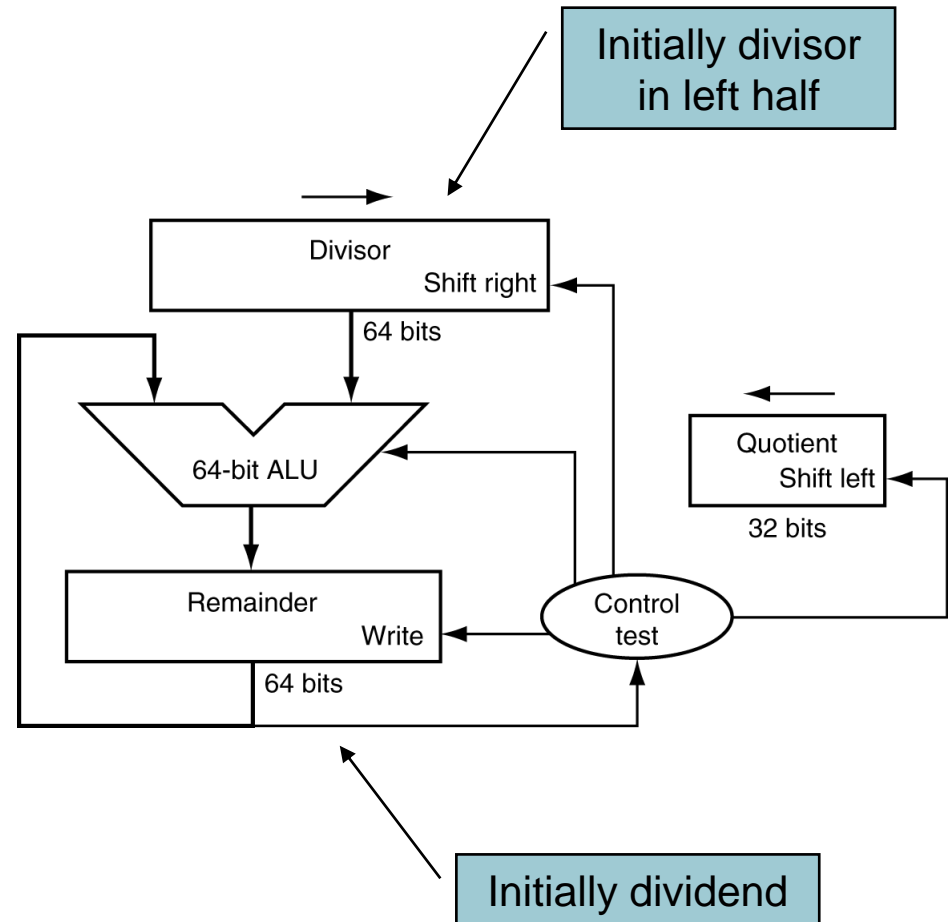
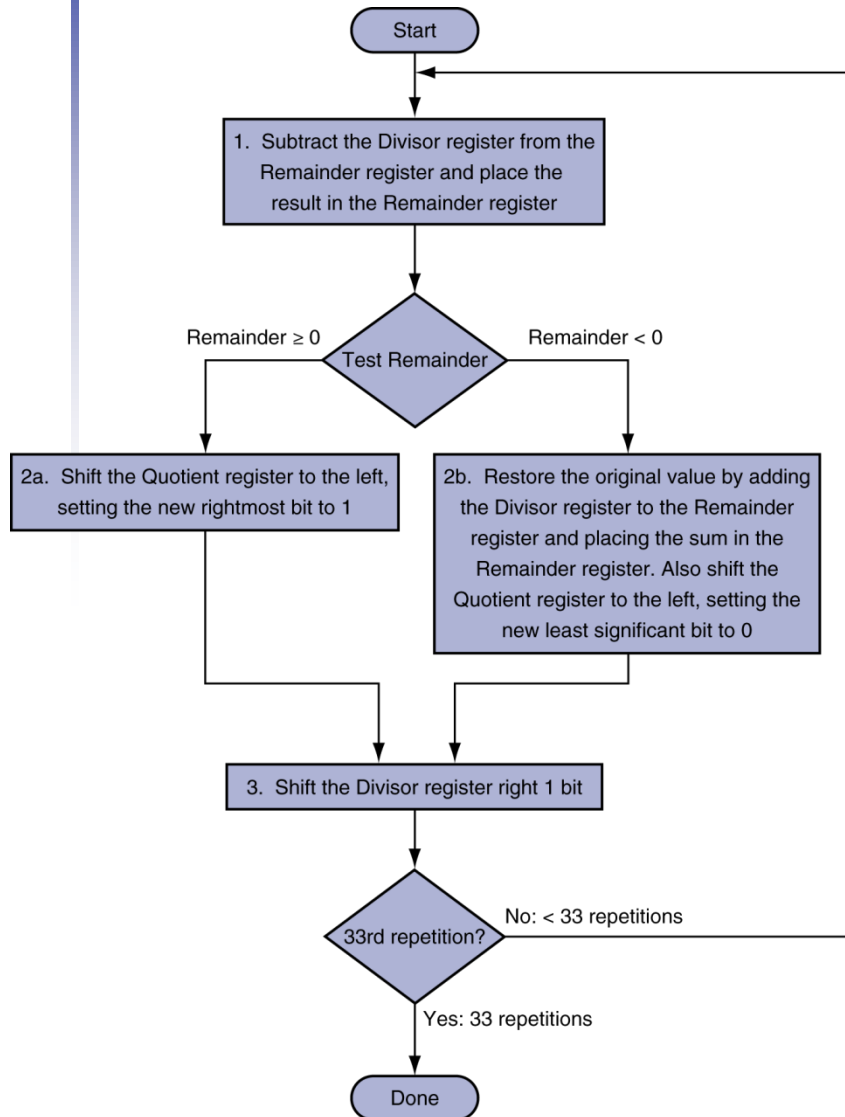


# Divide Example

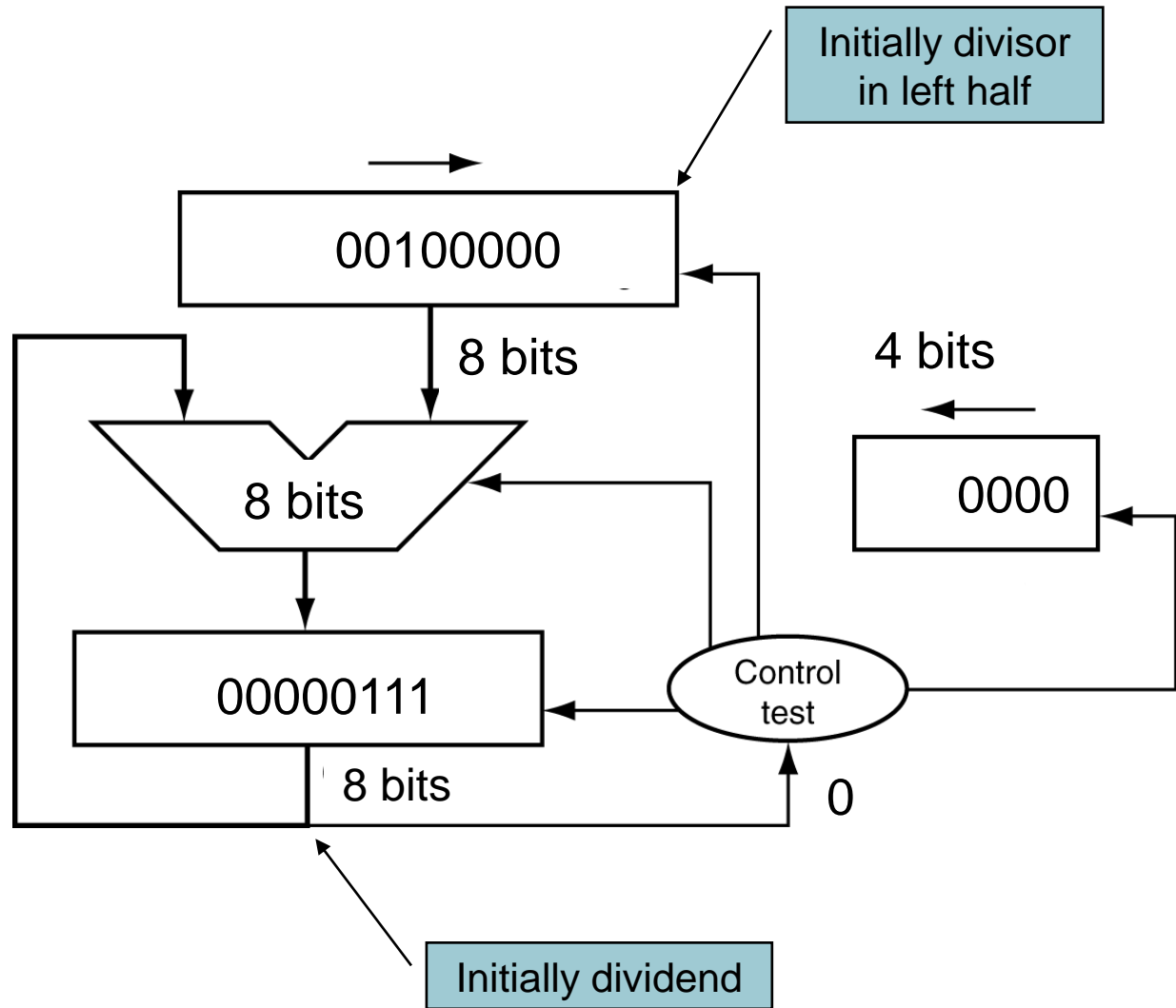
- Divide  $7_{\text{dec}}$  ( $0000\ 0111_{\text{bin}}$ ) by  $2_{\text{dec}}$  ( $0010_{\text{bin}}$ )

Iter	Step	Quot	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	Rem = Rem – Div	0000	0010 0000	1110 0111
	Rem < 0 → +Div, shift 0 into Q	0000	0010 0000	0000 0111
	Shift Div right	0000	0001 0000	0000 0111
2	Same steps as 1	0000	0001 0000	1111 0111
		0000	0001 0000	0000 0111
		0000	0000 1000	0000 0111
3	Same steps as 1	0000	0000 0100	0000 0111
4	Rem = Rem – Div	0000	0000 0100	0000 0011
	Rem >= 0 → shift 1 into Q	0001	0000 0100	0000 0011
	Shift Div right	0001	0000 0010	0000 0011
5	Same steps as 4	0011	0000 0001	0000 0001

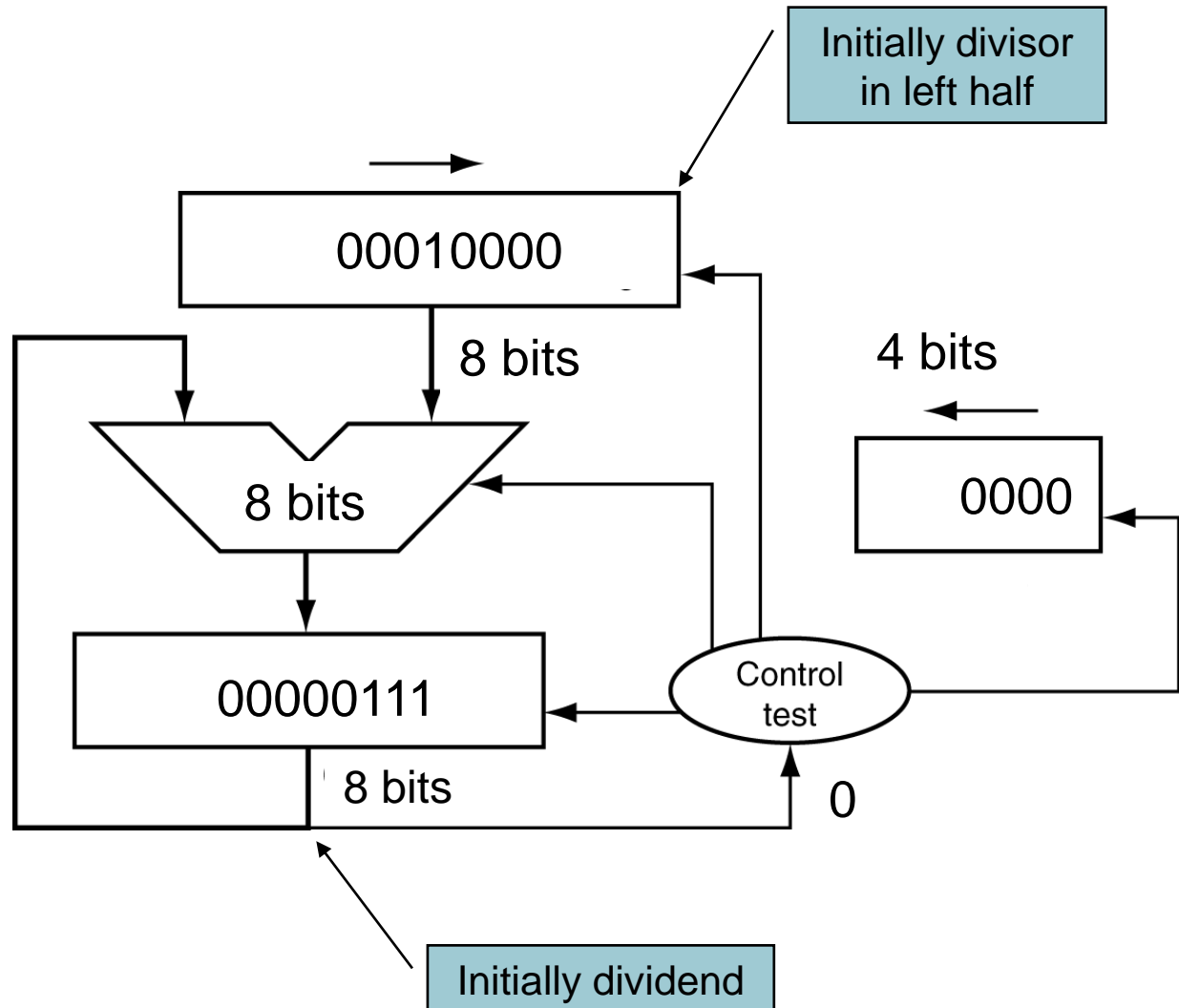
# Division Hardware



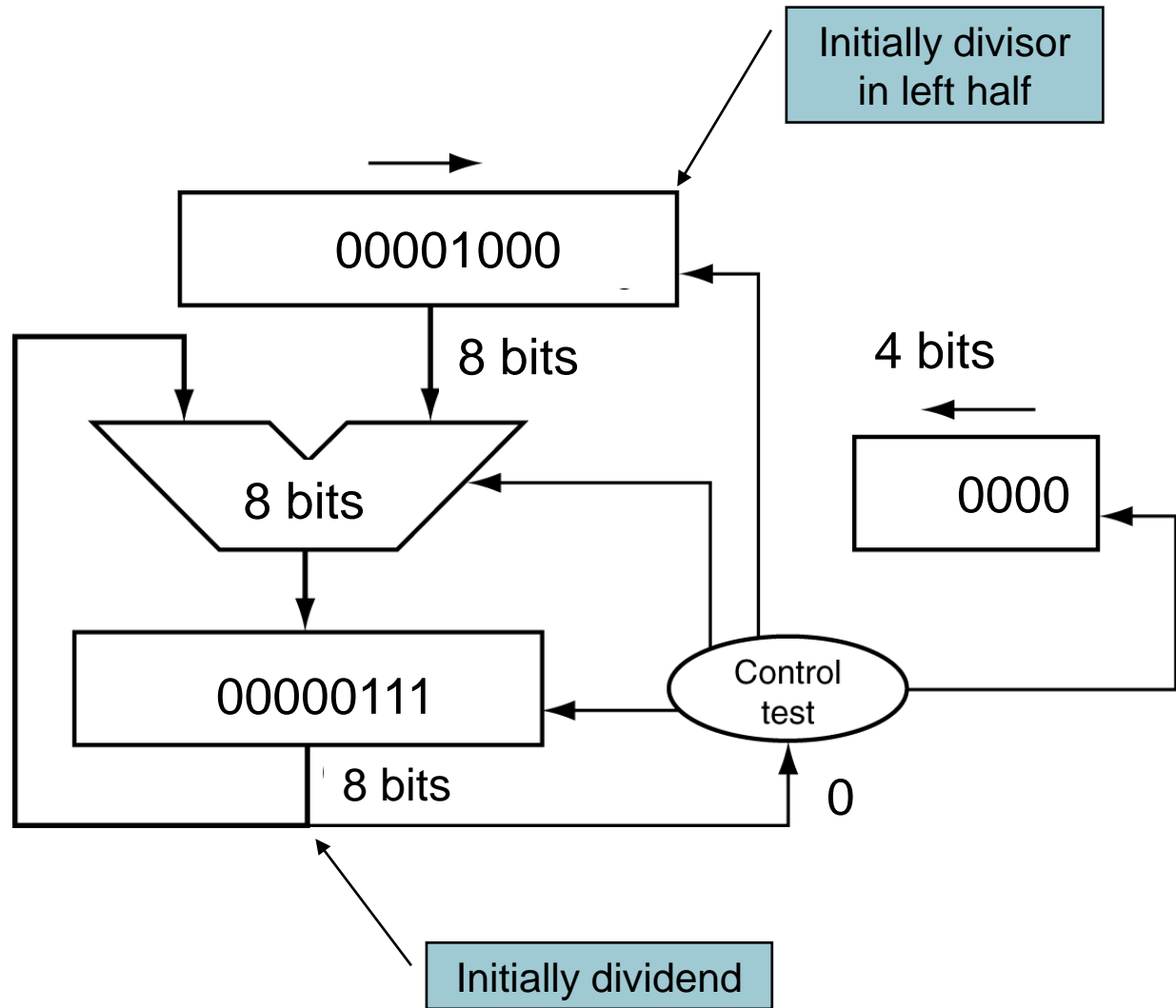
# Division Hardware



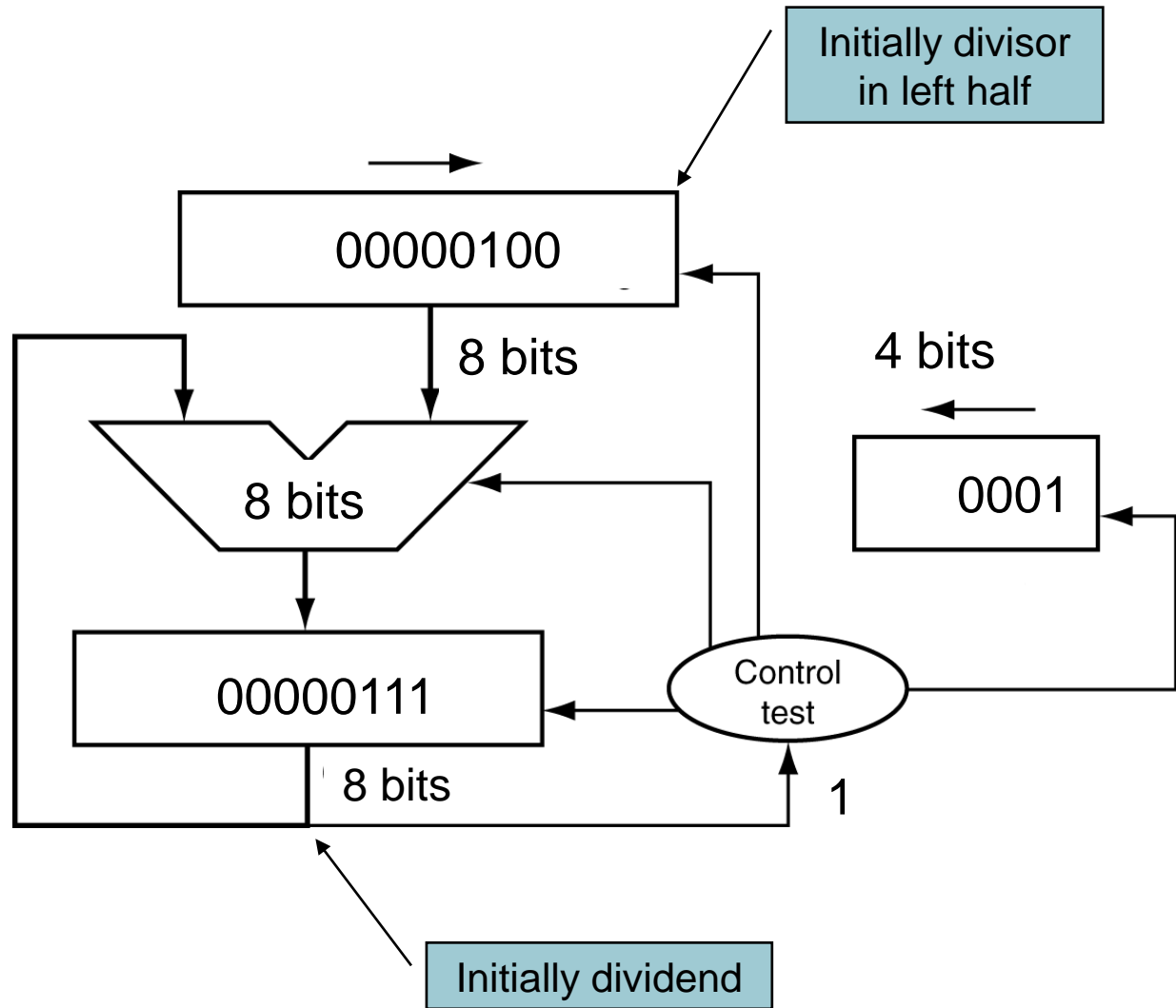
# Division Hardware



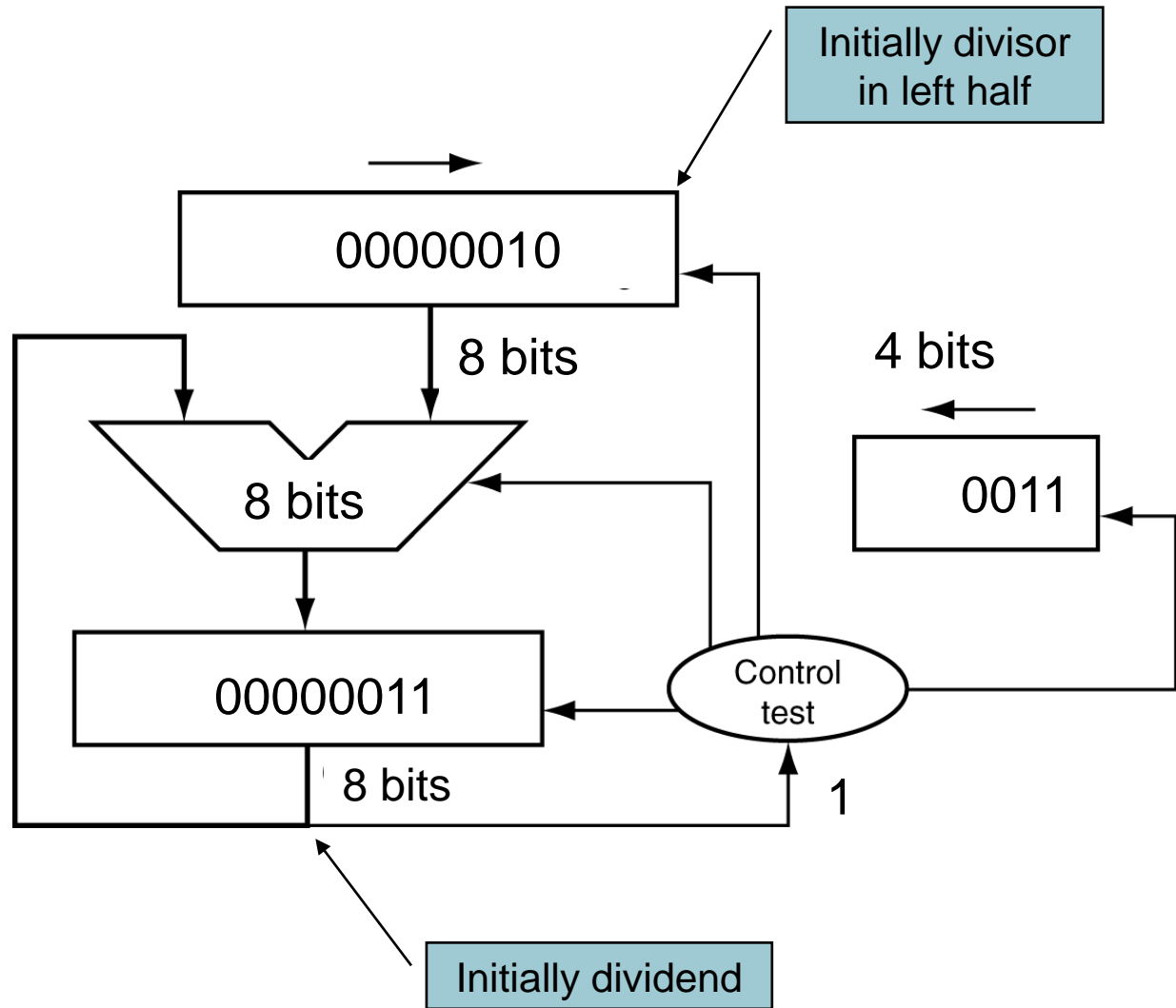
# Division Hardware



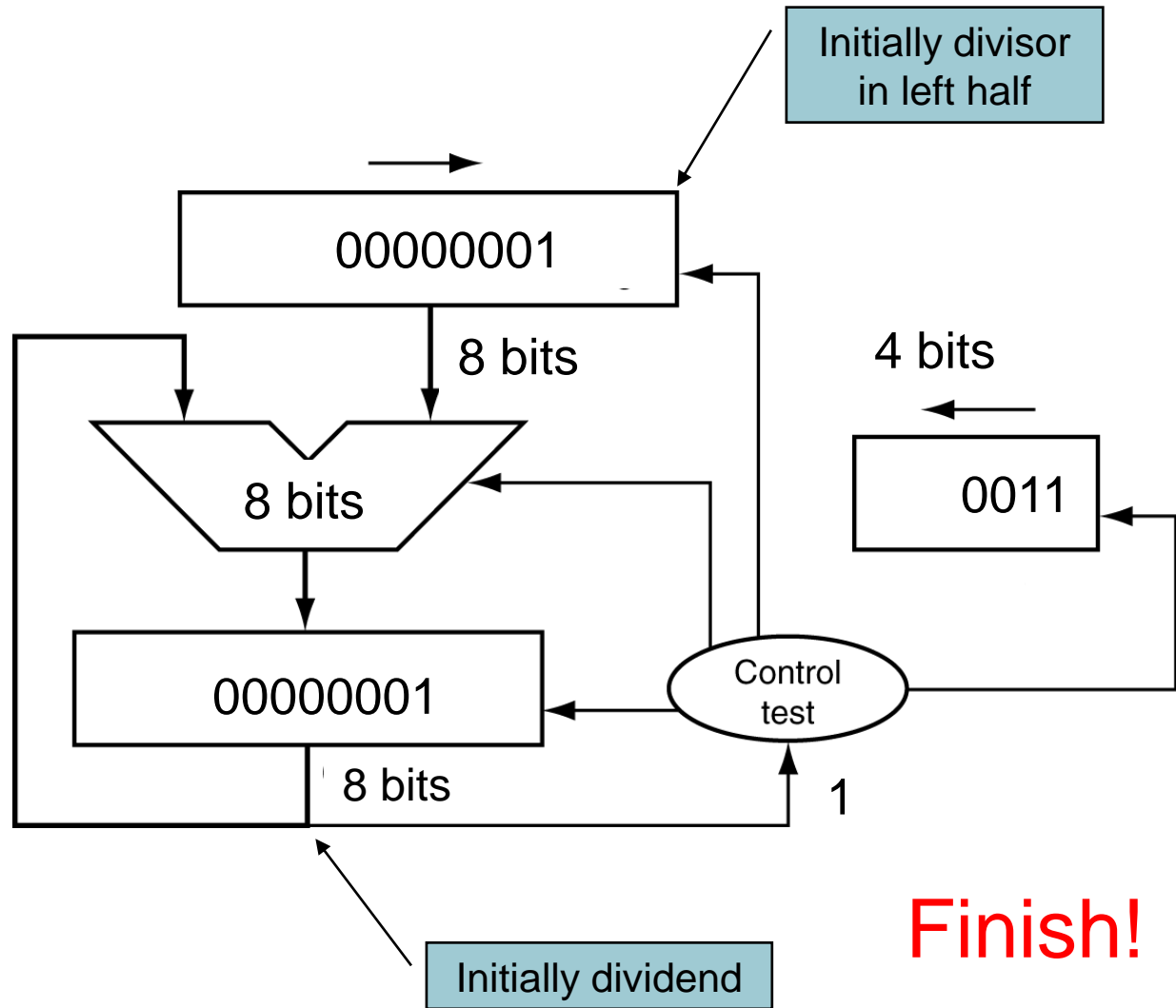
# Division Hardware



# Division Hardware

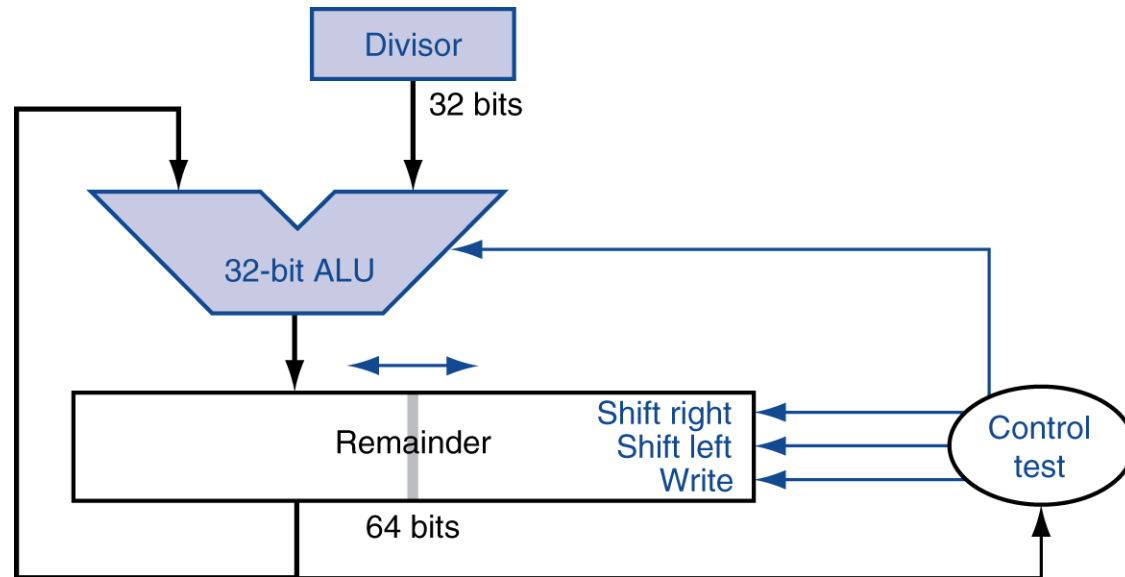


# Division Hardware





# Optimized Divider



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

# Signed Division

- $(+7) \div (-2) = (-3) \cdots (+1)$
- $(-7) \div (-2) = (+3) \cdots (-1)$
- The quotient is +, if the signs of divisor and dividend agrees, otherwise, quotient is –
- The sign of the remainder matches that of the 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

# 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 checks if required
  - ◆ Use `mfhi`, `mflo` to access result