CS202: COMPUTER ORGANIZATION

Chapter 3

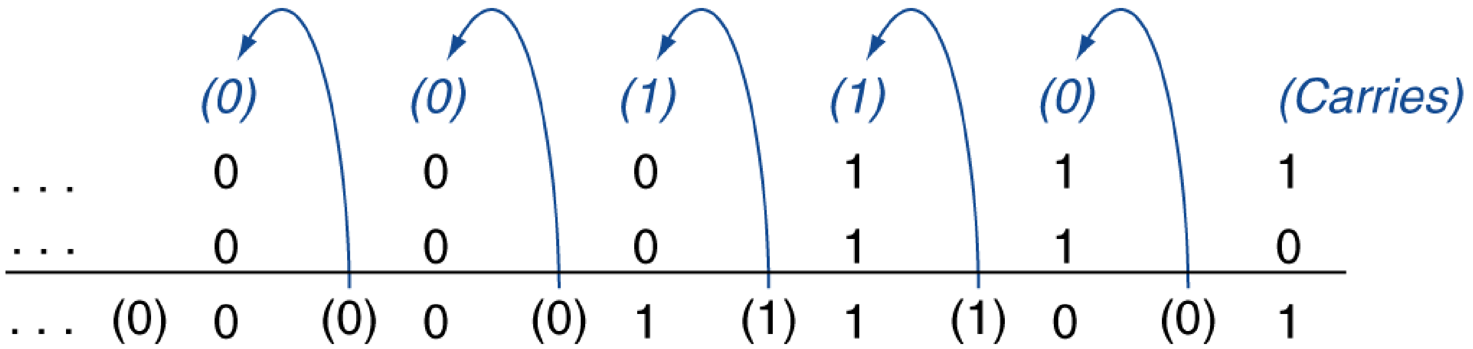
Arithmetic for Computers

Arithmetic for Computers

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

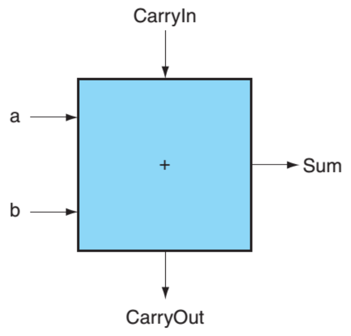
Integer Addition

- Example: 7 + 6



- Overflow 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

1-bit adder

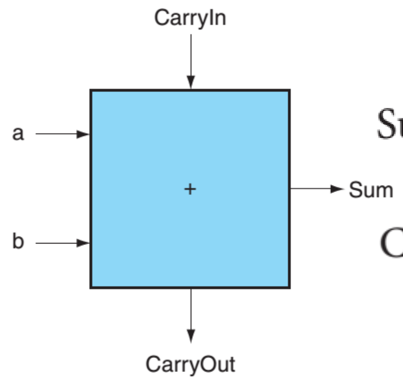


$$\text{Sum} = (a \cdot \overline{b} \cdot \overline{\text{CarryIn}}) + (\overline{a} \cdot b \cdot \overline{\text{CarryIn}}) + (\overline{a} \cdot \overline{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$$

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

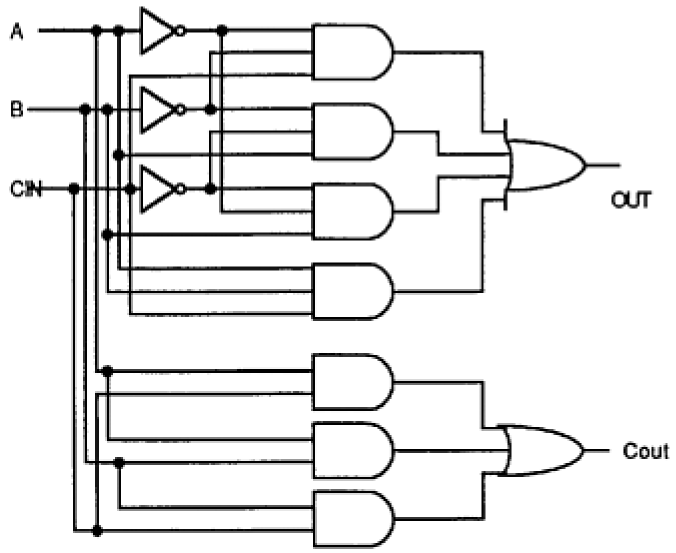
Inputs			Outputs		
a	**b**	**CarryIn**	**CarryOut**	**Sum**	**Comments**
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$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

1-bit Adder

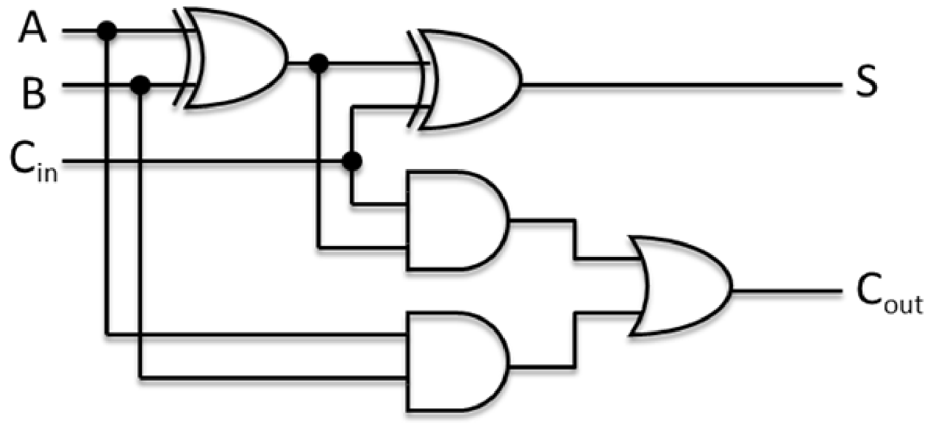


$$\text{Sum} = (a \cdot \overline{b} \cdot \overline{\text{CarryIn}}) + (\overline{a} \cdot b \cdot \overline{\text{CarryIn}}) + (\overline{a} \cdot \overline{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$$

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



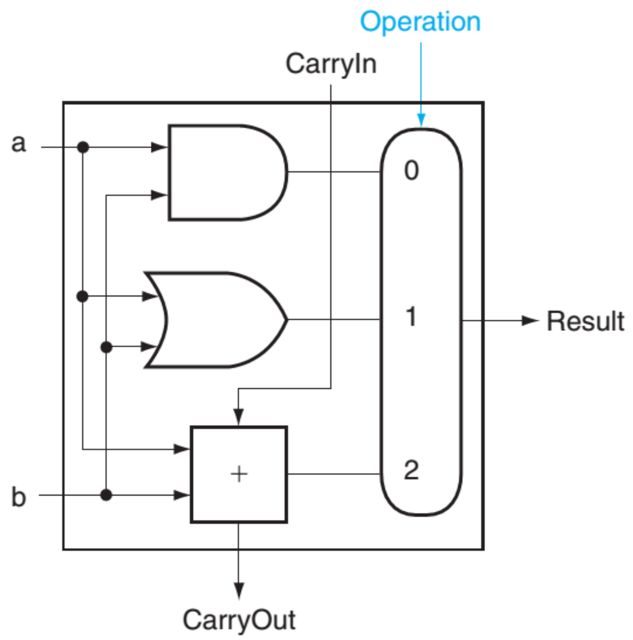
1-bit adder – version 1

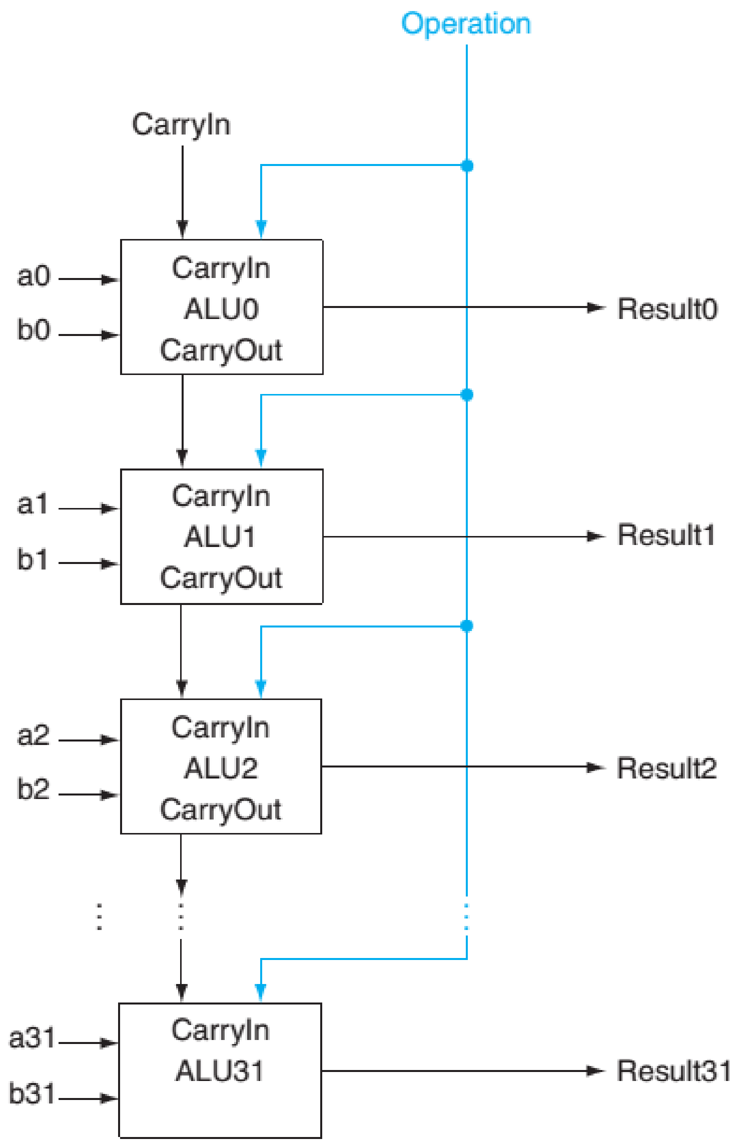


1-bit adder – version 2

1-bit ALU

- ALU: arithmetic logical unit
- 1-bit ALU and 32-bit ALU





Integer 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

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

- Graphics and media processing operates on vectors of 8-bit and 16-bit data
  - Use 64-bit adder, with partitioned carry chain
    - Operate on 8×8-bit, 4×16-bit, or 2×32-bit vectors
  - SIMD (single-instruction, multiple-data)
- 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

Multiplication

- Start with long-multiplication approach

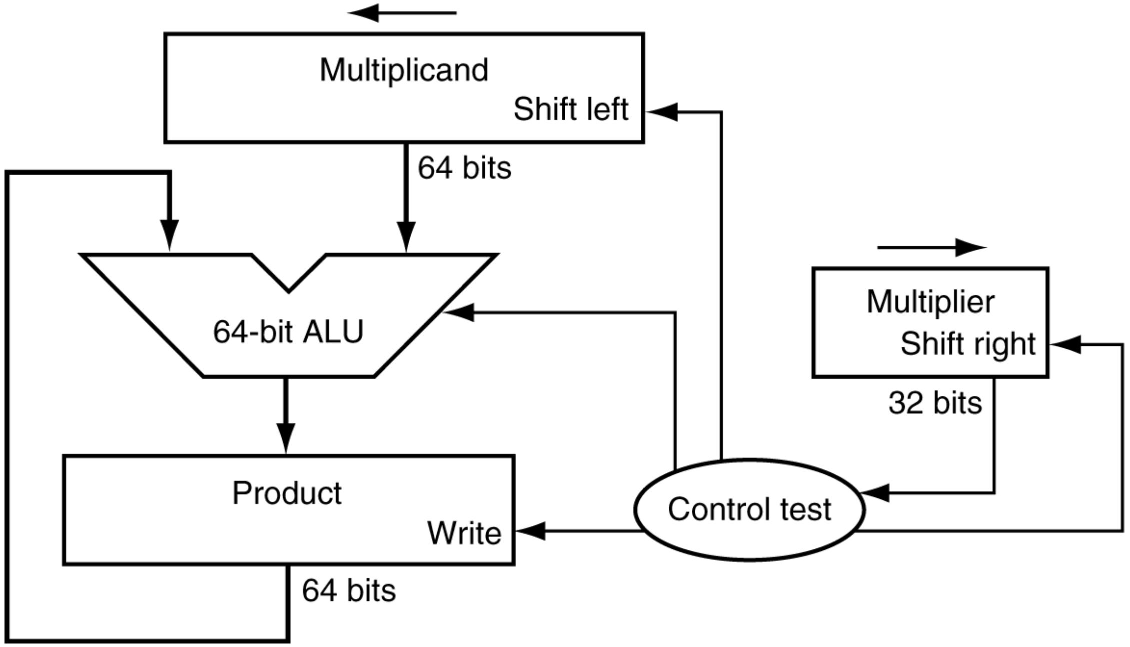
multiplicand

multiplier

product

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

Length of product is the sum of operand lengths

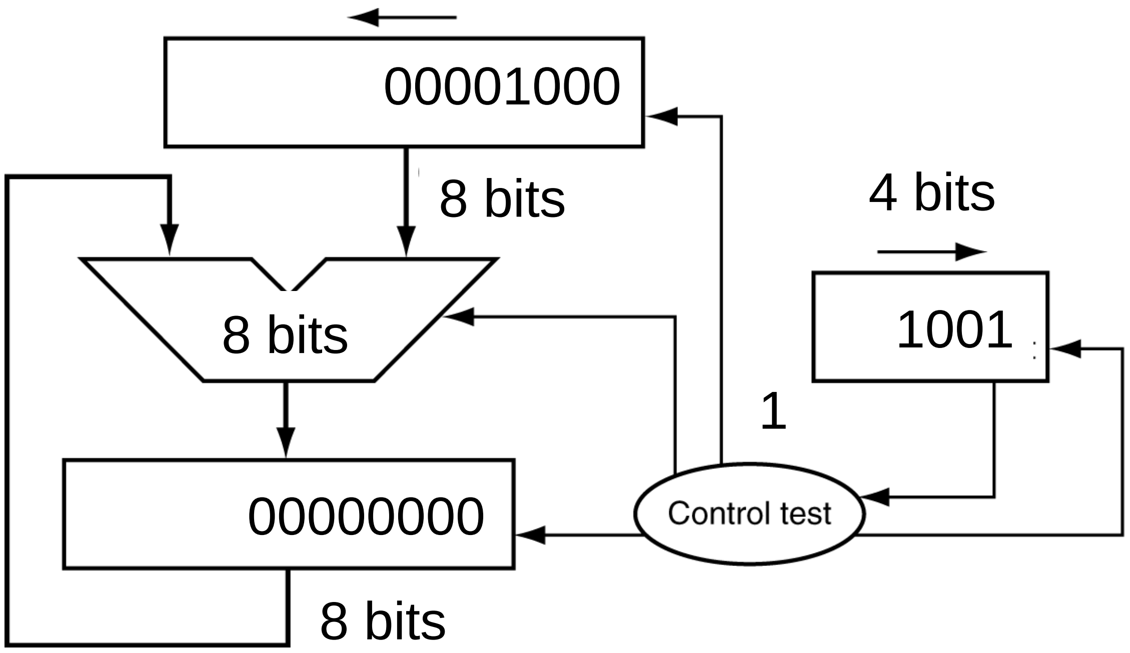


Multiplication

- Start with long-multiplication approach

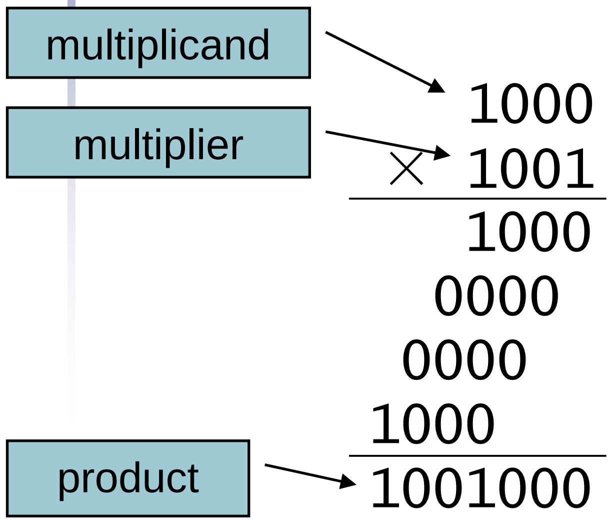
```
multiplicand          1000
multiplier          × 1001
                    ------
                      1000
                     0000
                    0000
                   1000
                  --------
product            1001000
```

Length of product is the sum of operand lengths

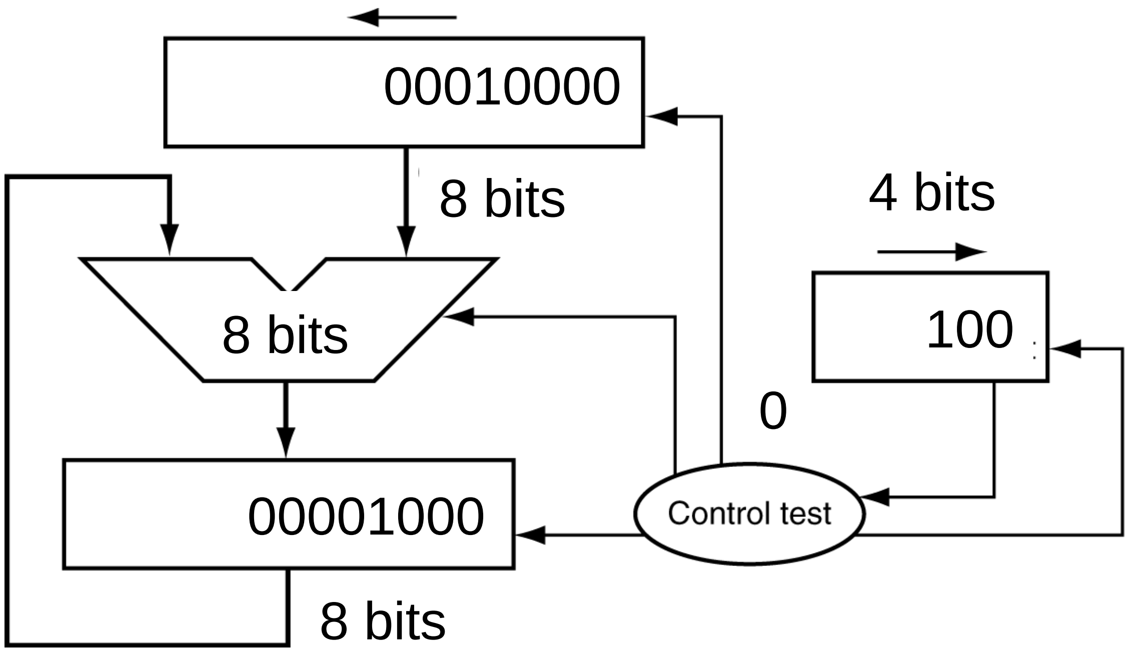


Multiplication

- Start with long-multiplication approach

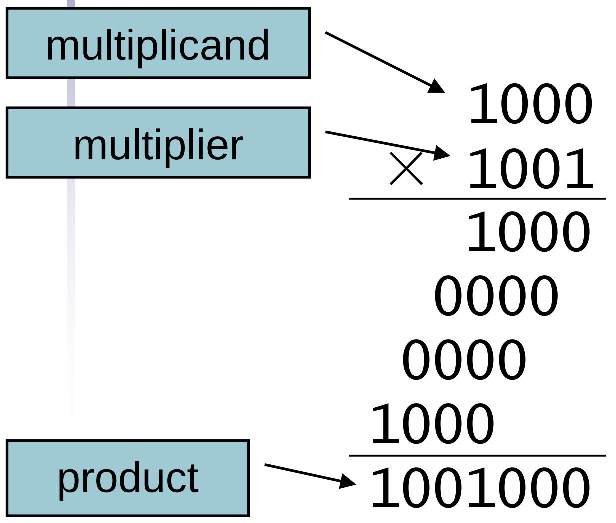


Length of product is the sum of operand lengths

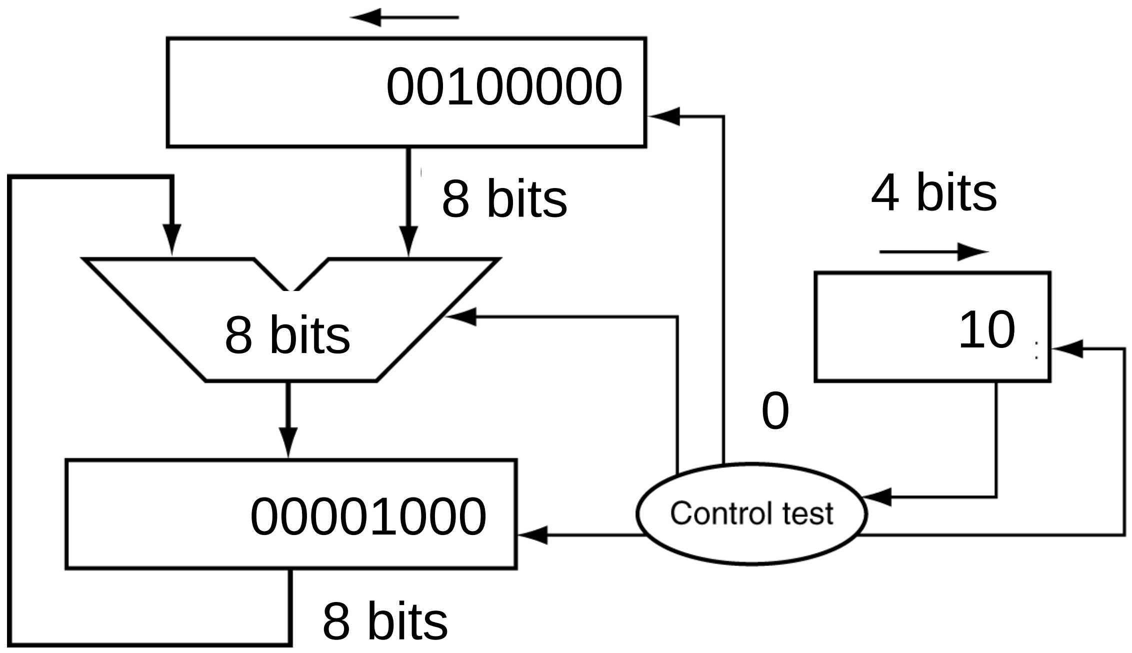


Multiplication

- Start with long-multiplication approach

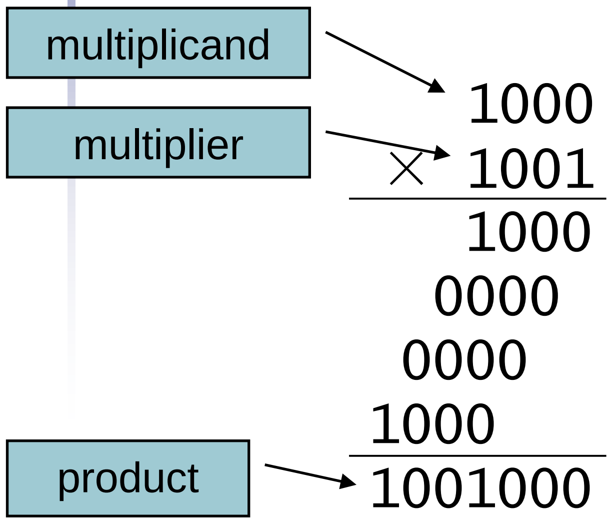


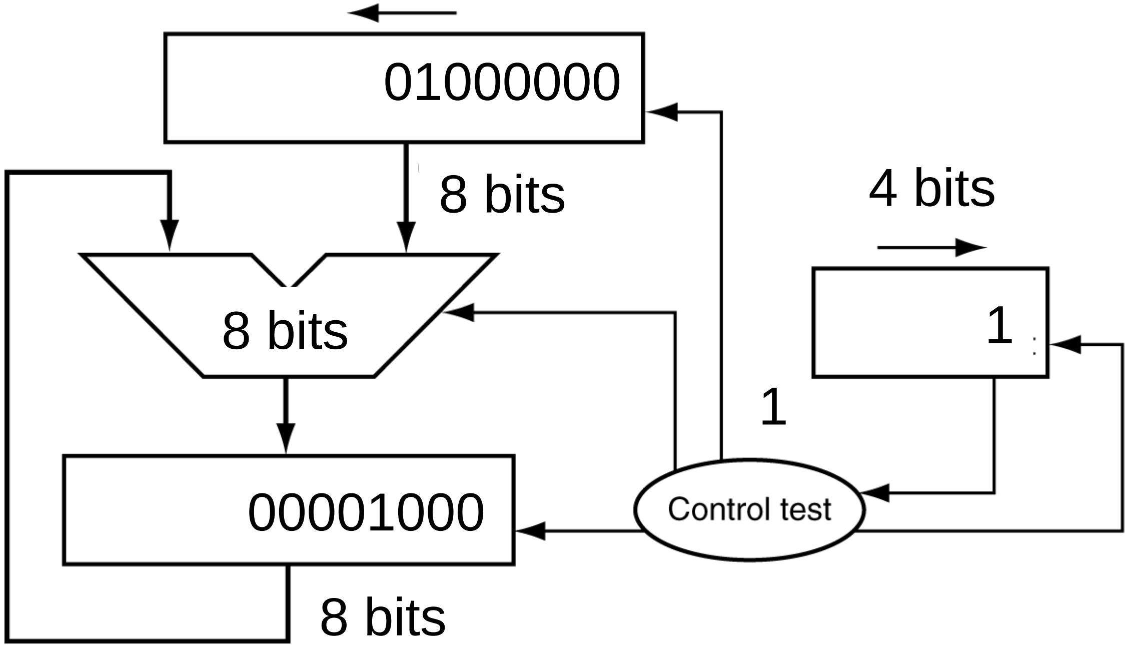
Length of product is the sum of operand lengths



Multiplication

- Start with long-multiplication approach

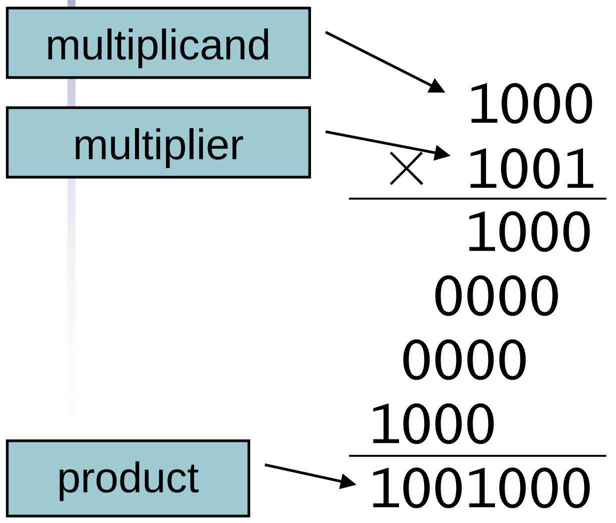




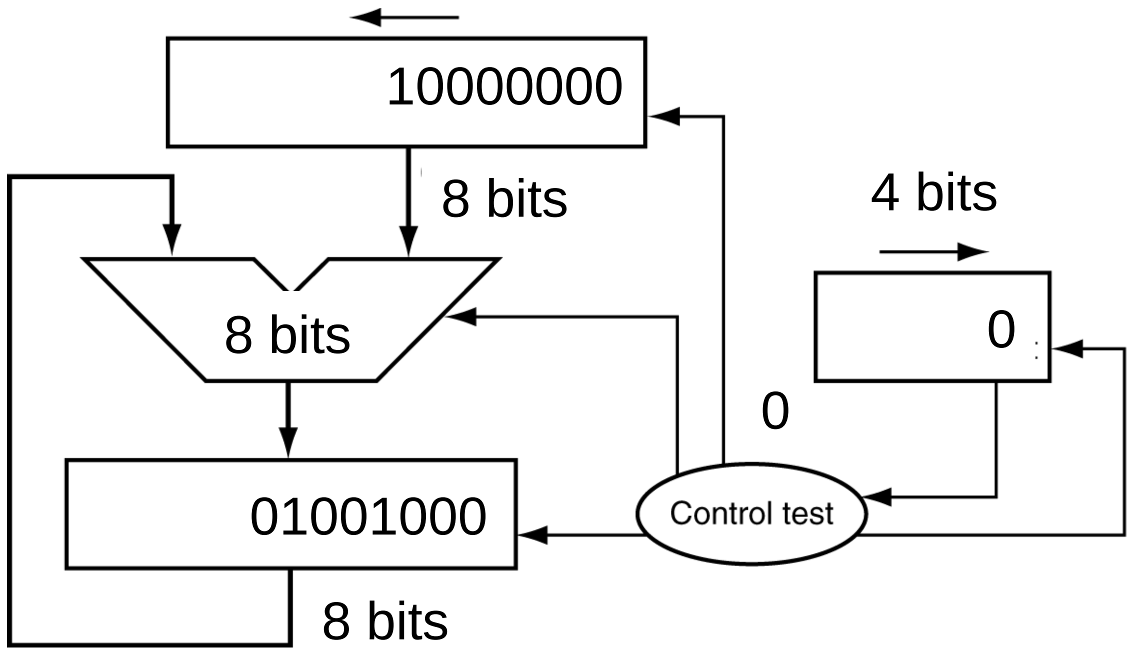
Length of product is the sum of operand lengths

Multiplication

- Start with long-multiplication approach

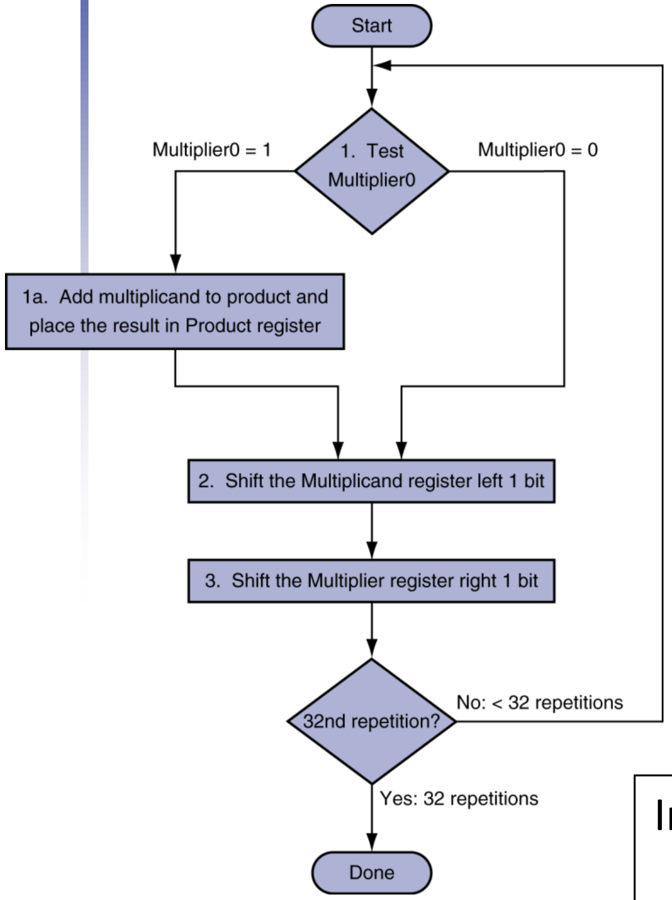


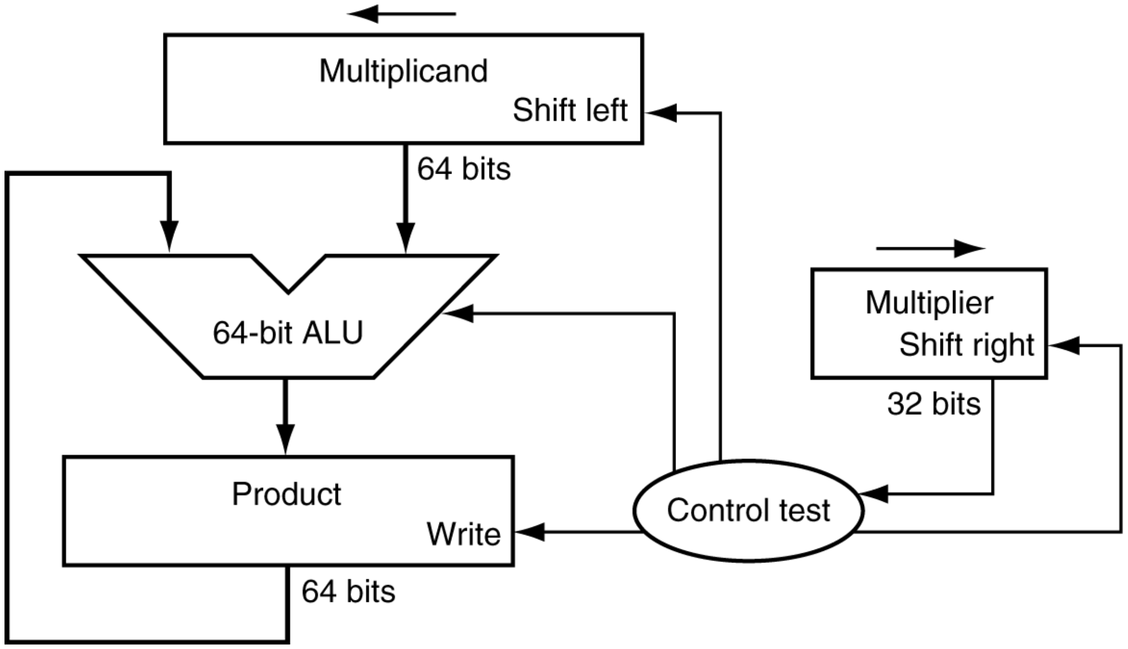
Length of product is the sum of operand lengths



Finish!

Multiplication Hardware


Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

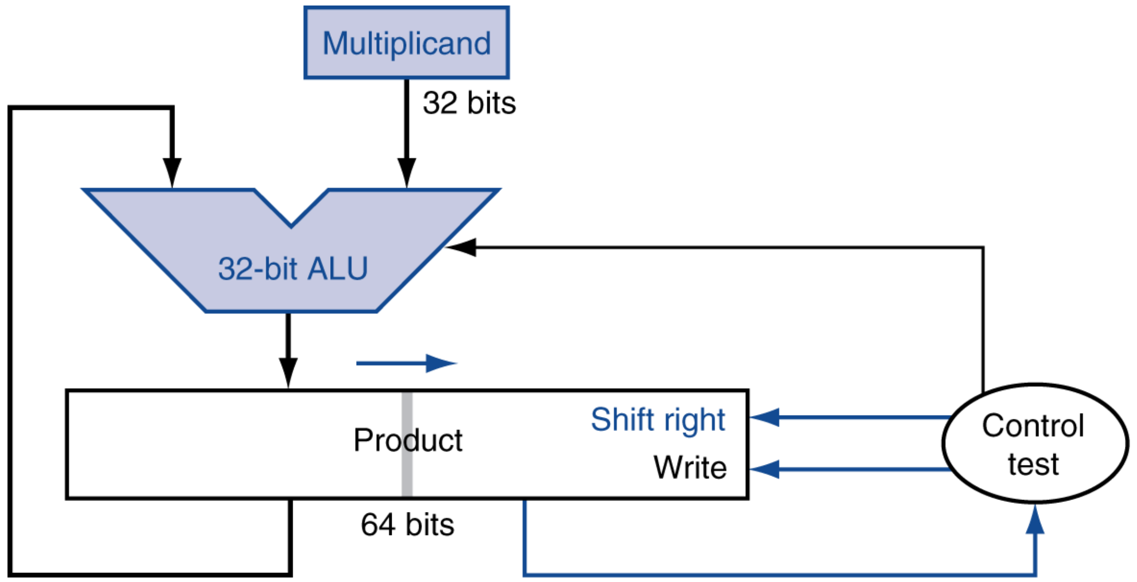


In every step:

- ✓ 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

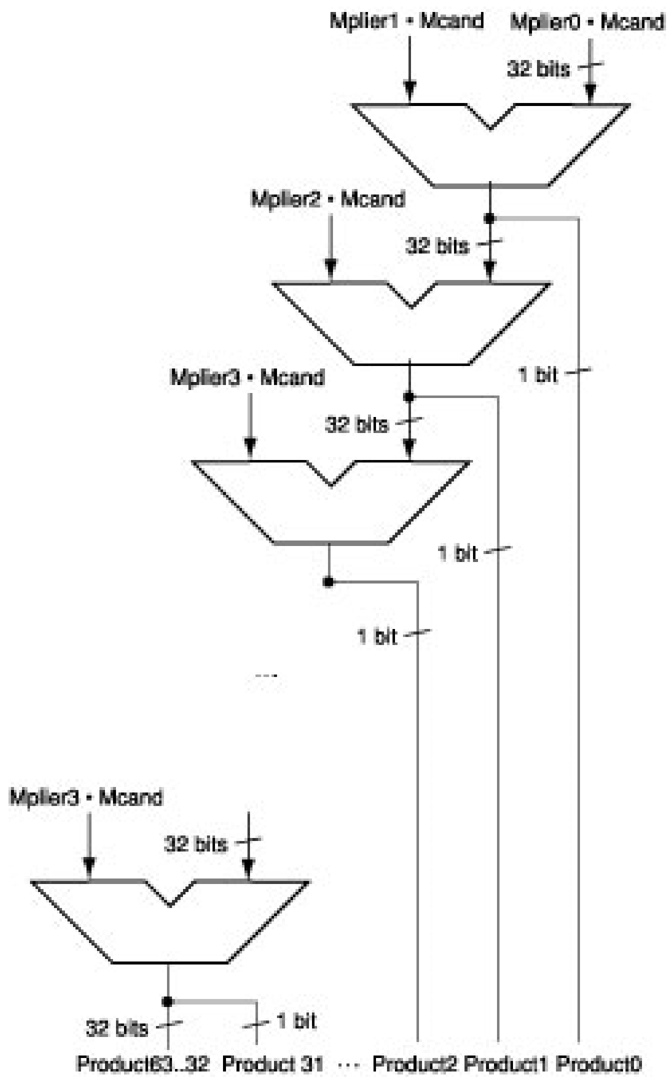
Optimized Multiplier

- Perform steps in parallel: add/shift



- ✓ 64-bit “Product” stores product and multiplier
- ✓ the sum keeps shifting right
- ✓ at every step, number of bits in product + multiplier = 64, hence, they share a single 64-bit register
- ✓ 32-bit ALU and multiplicand are used instead of 64-bit ones.

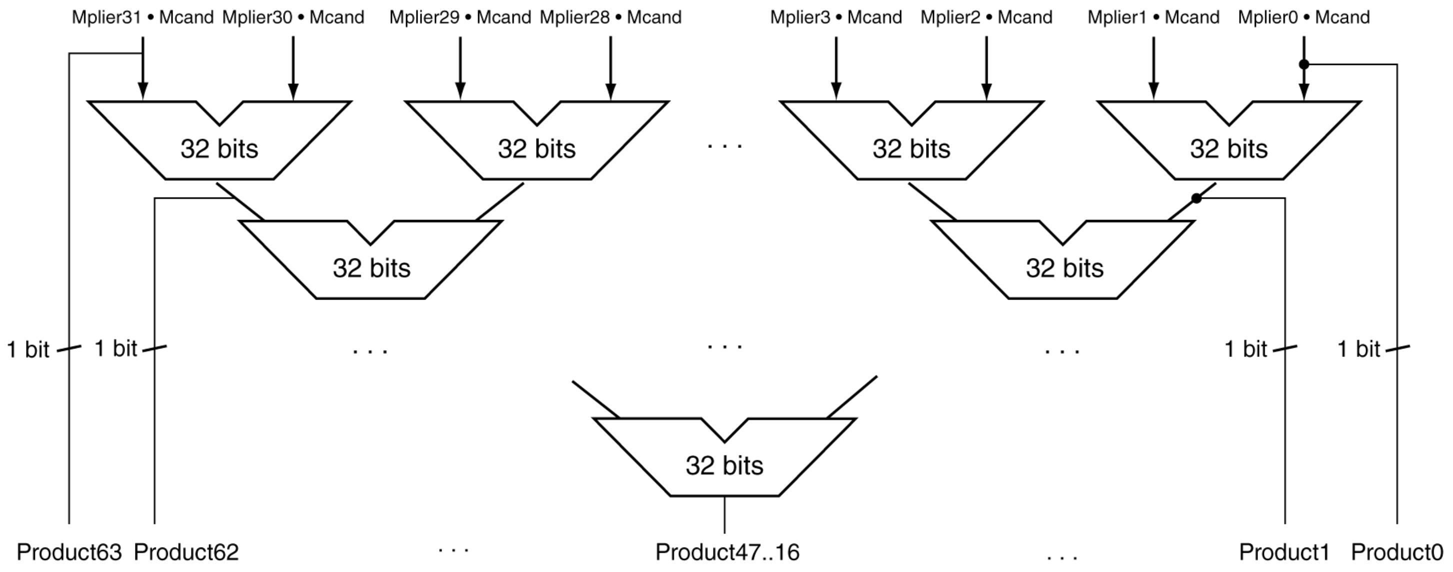
Faster Multiplier



- 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

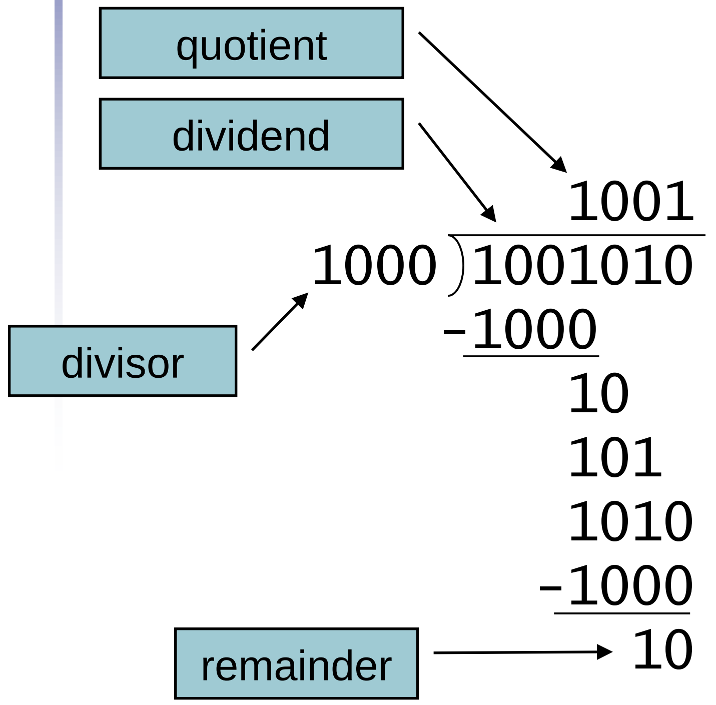


- Can be pipelined
  - Several multiplication performed in parallel

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

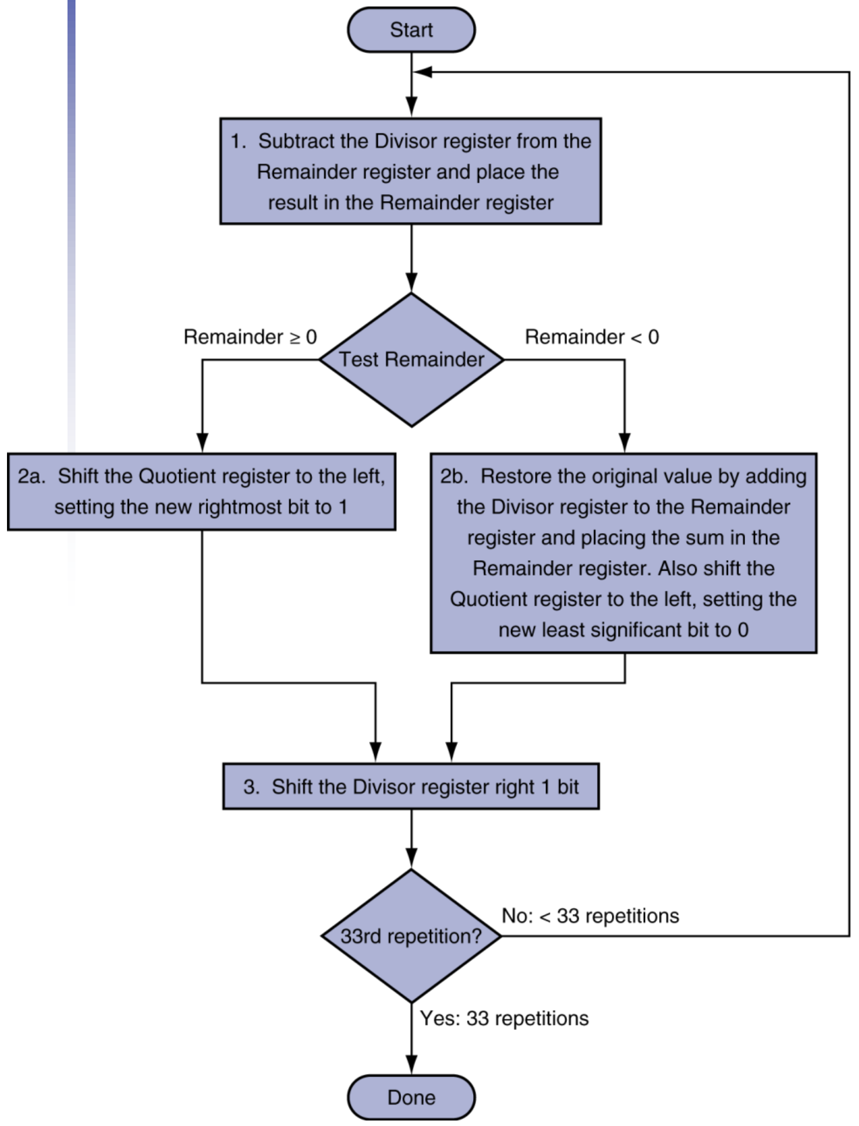
- Check for 0 divisor
- Long division approach
  - If divisor ≤ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division
  - Do the subtract, and if remainder goes < 0, add divisor back
- Signed division
  - Divide using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

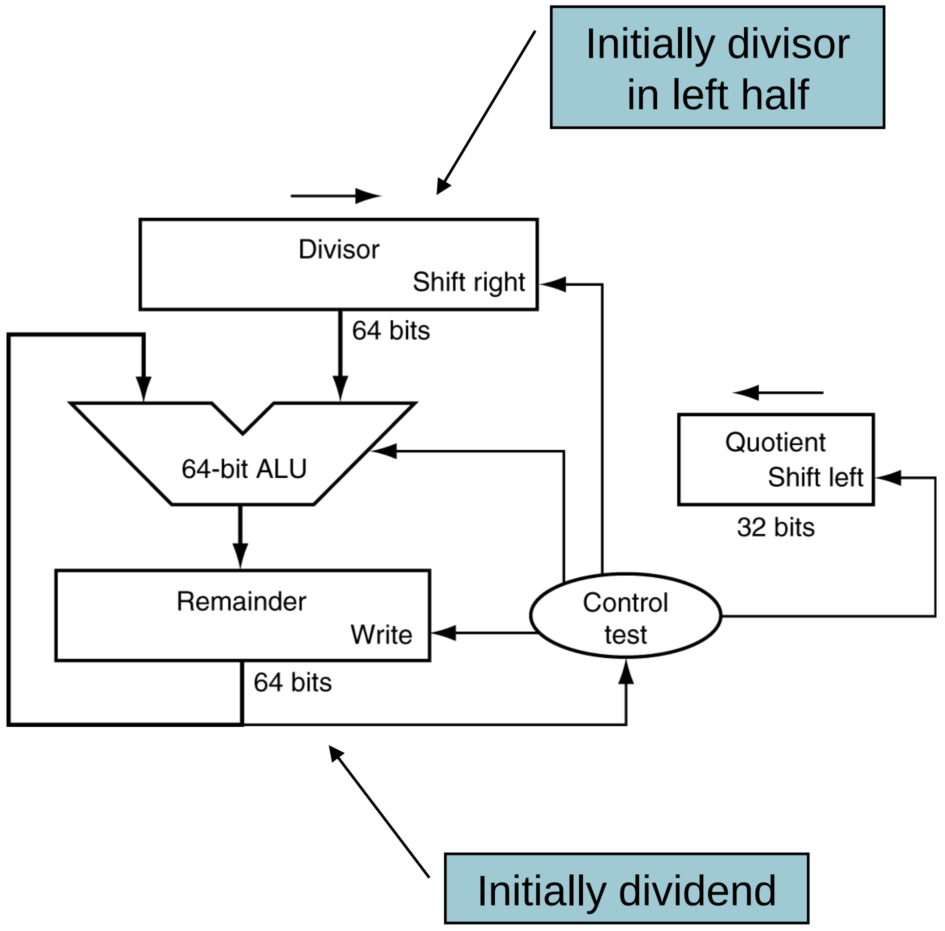
Divide Example

- Divide 7ten (0000 0111two) by 2ten (0010two)

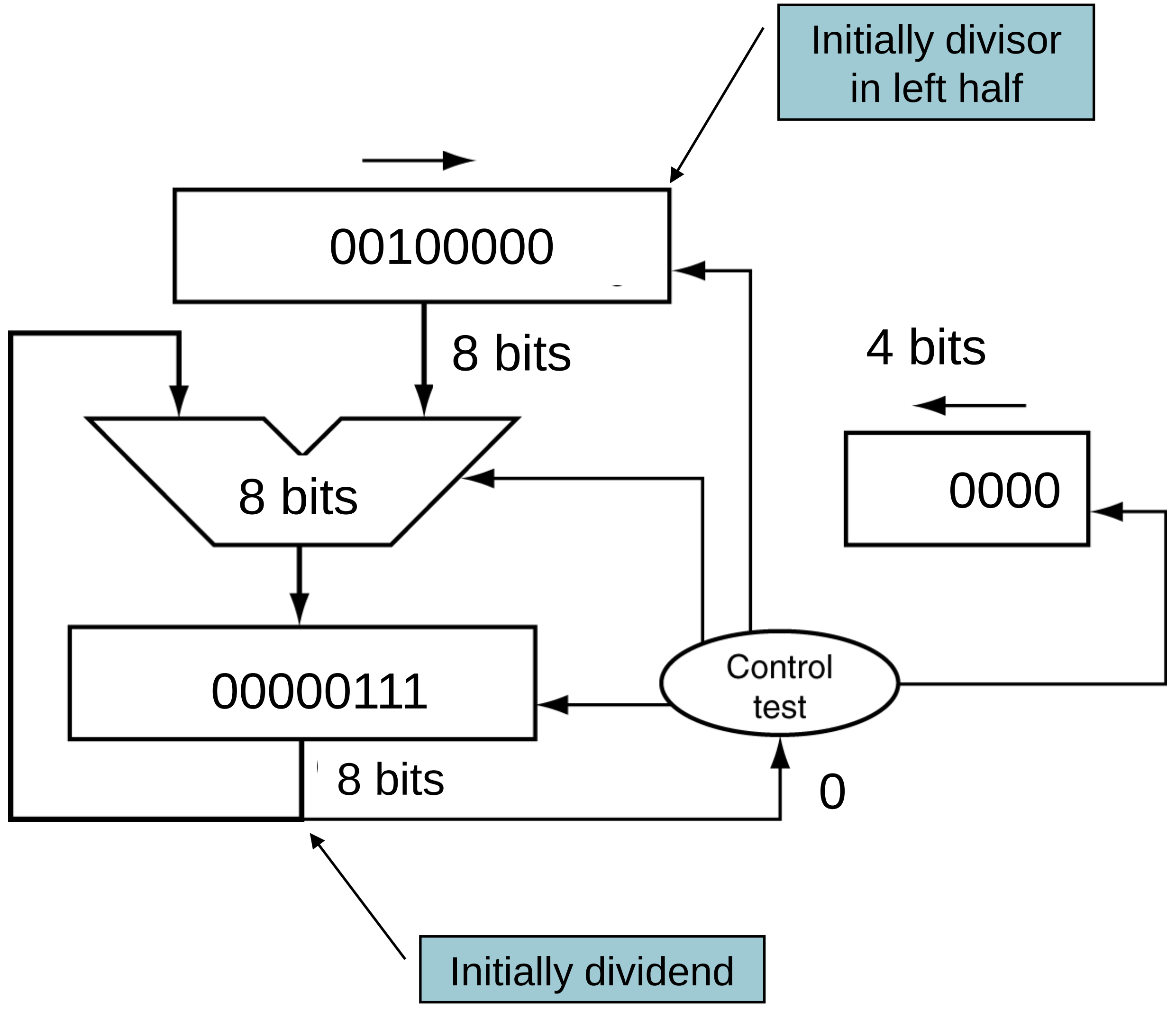
Iter	Step	Quot	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	Rem = Rem – Div	0000	0010 0000	1110 0111
	Rem < 0 → +Div, shift 0 into Q	0000	0010 0000	0000 0111
	Shift Div right	0000	0001 0000	0000 0111
2	Same steps as 1	0000	0001 0000	1111 0111
		0000	0001 0000	0000 0111
		0000	0000 1000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

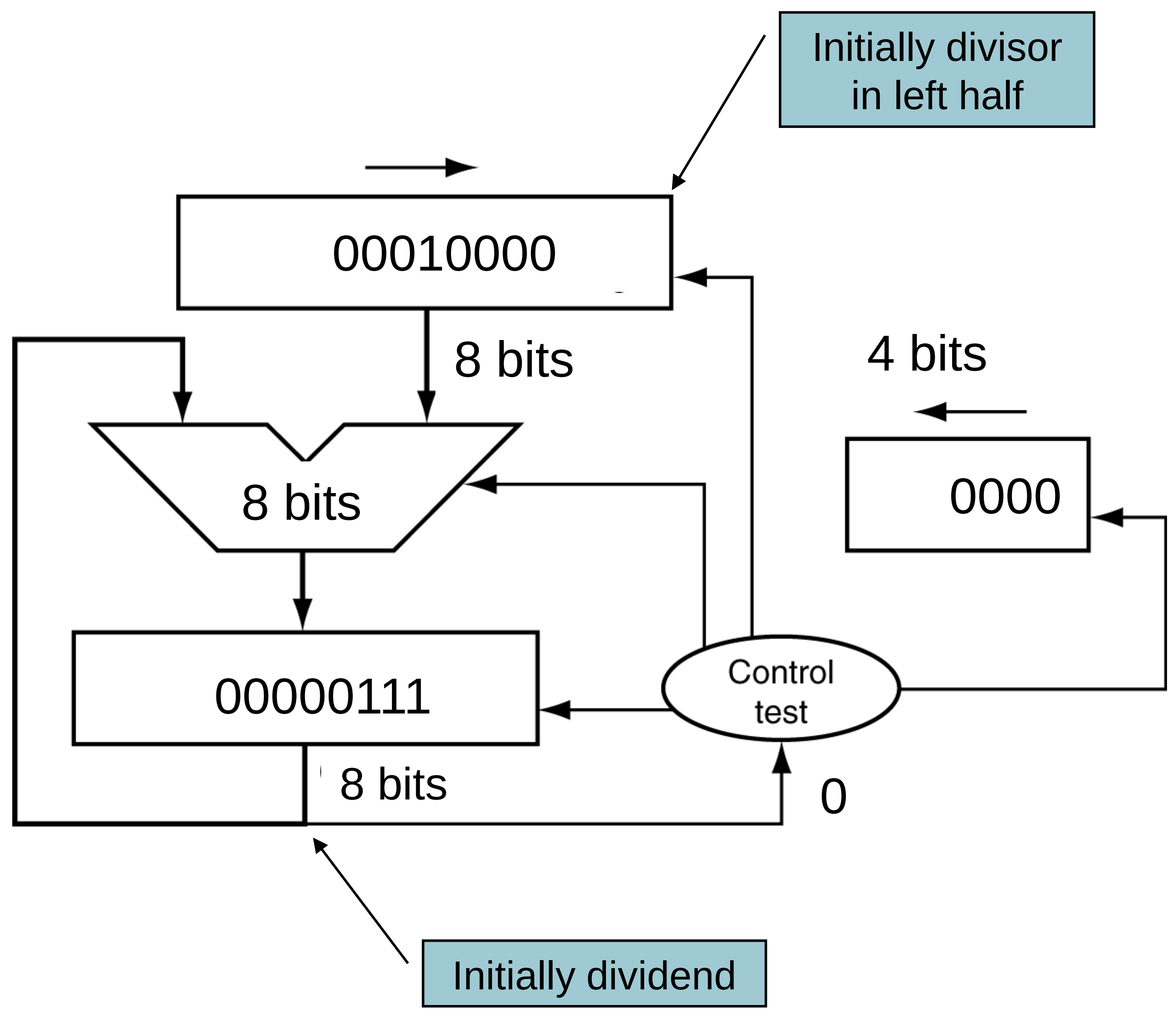

Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
33rd repetition?
No: < 33 repetitions
Yes: 33 repetitions
Done



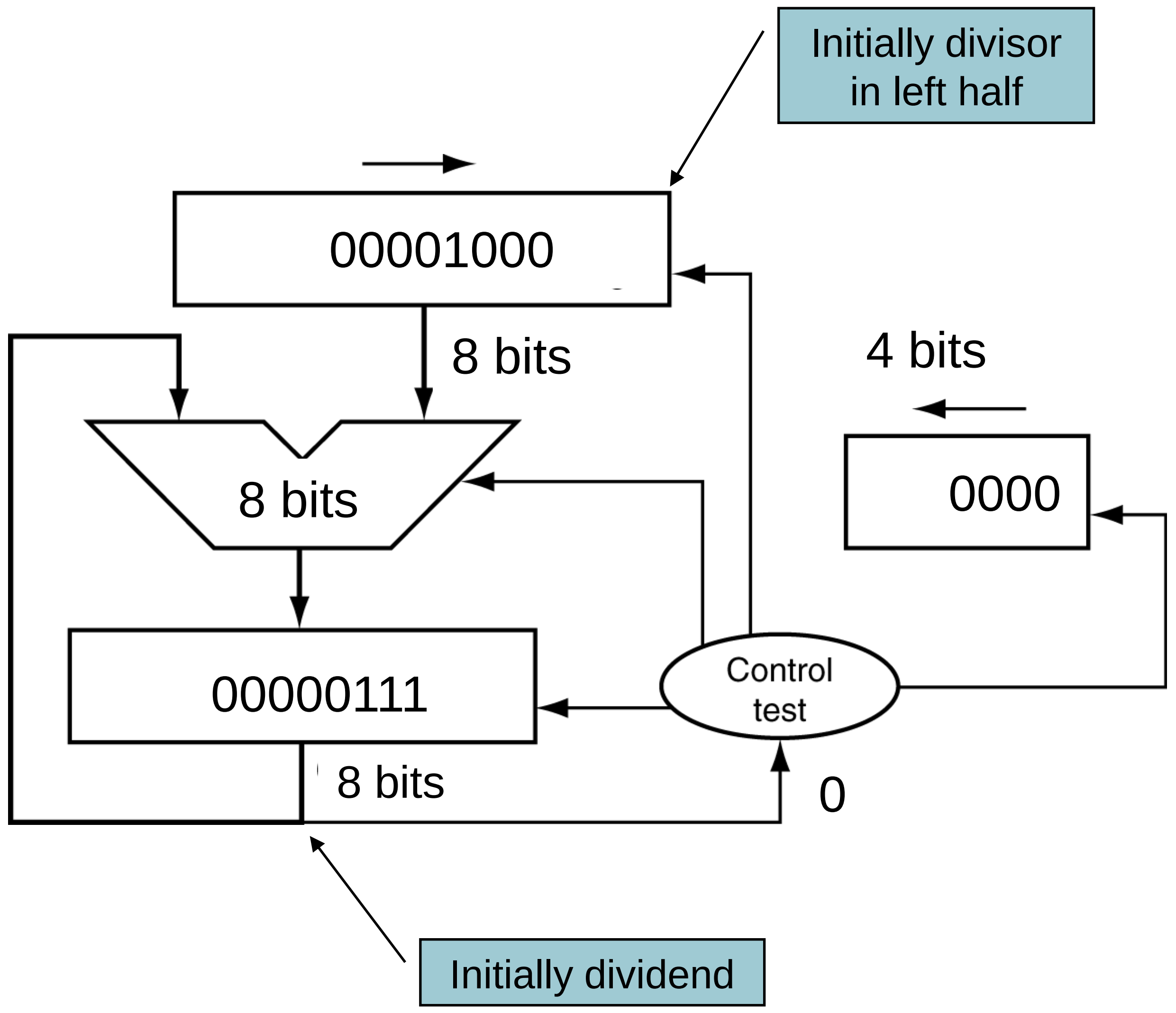
Division Hardware



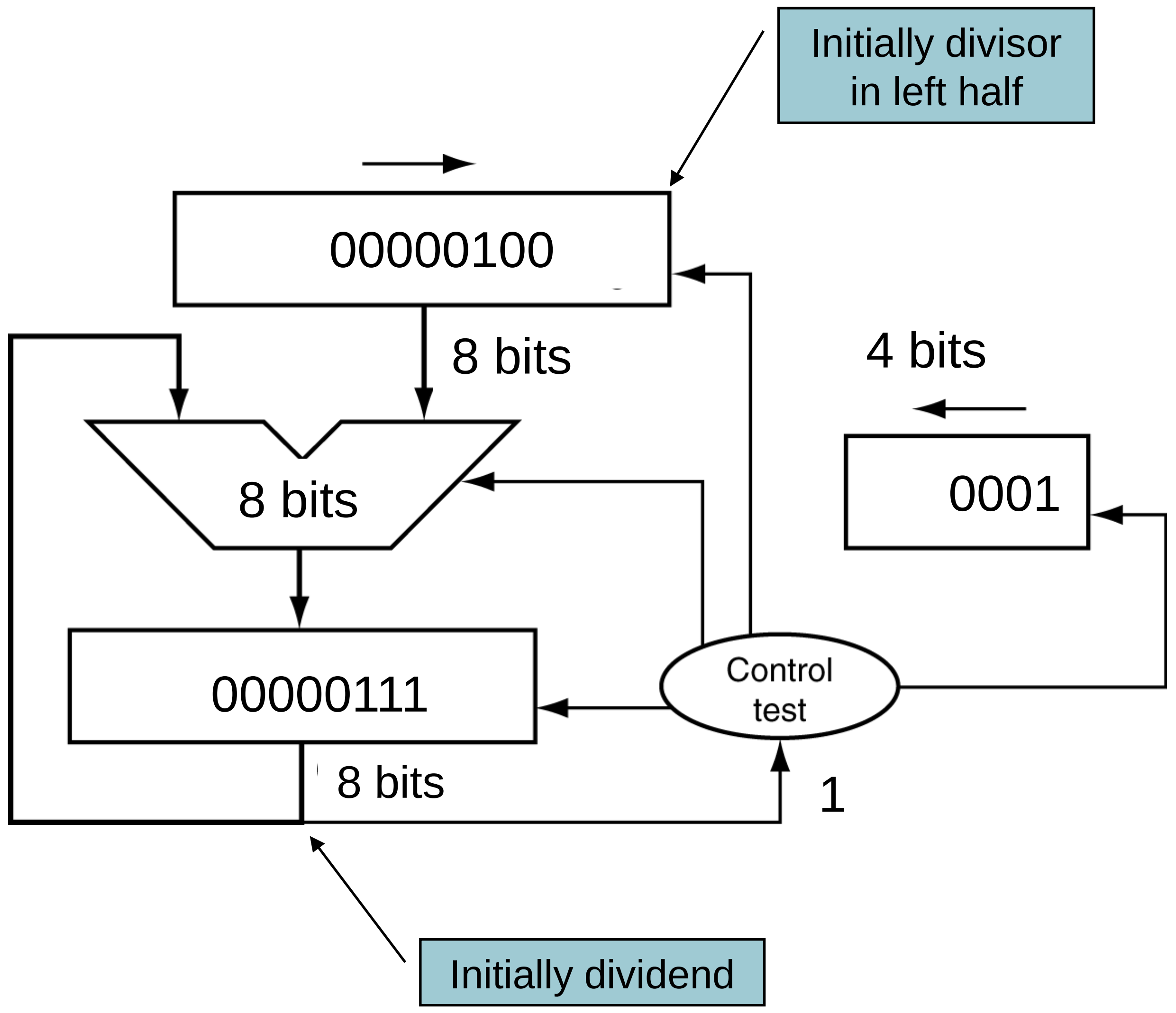
Division Hardware



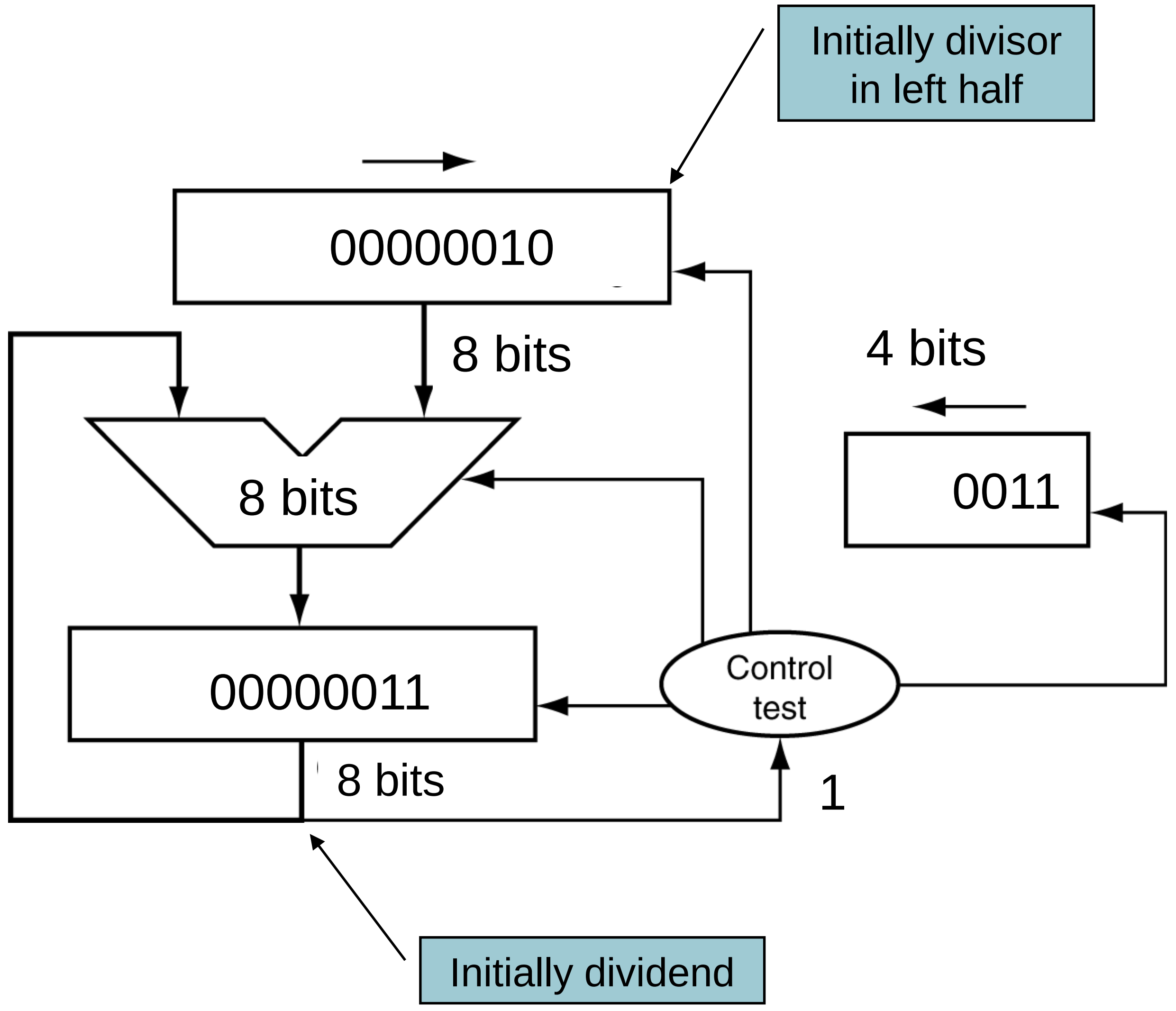
Division Hardware



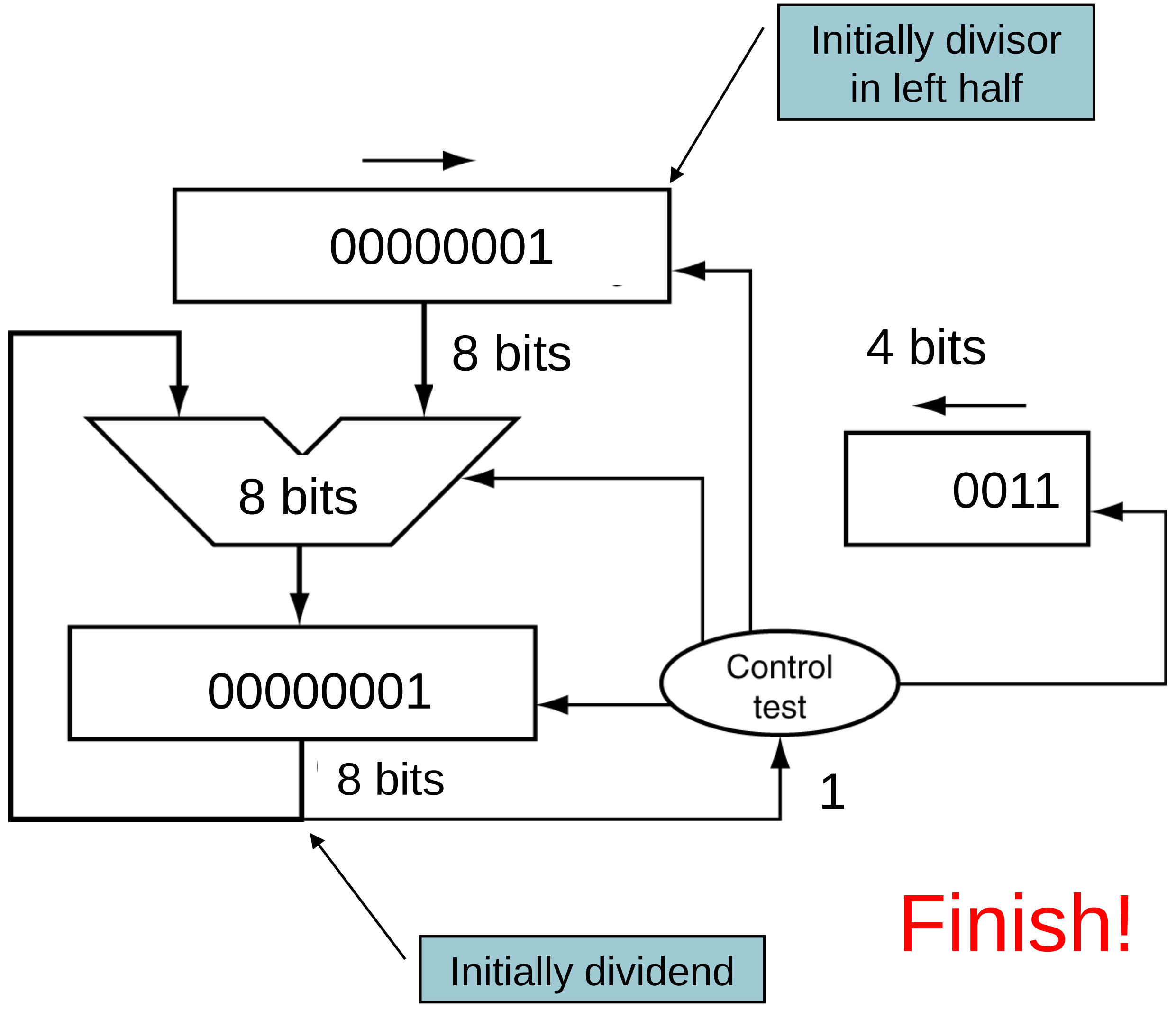
Division Hardware



Division Hardware

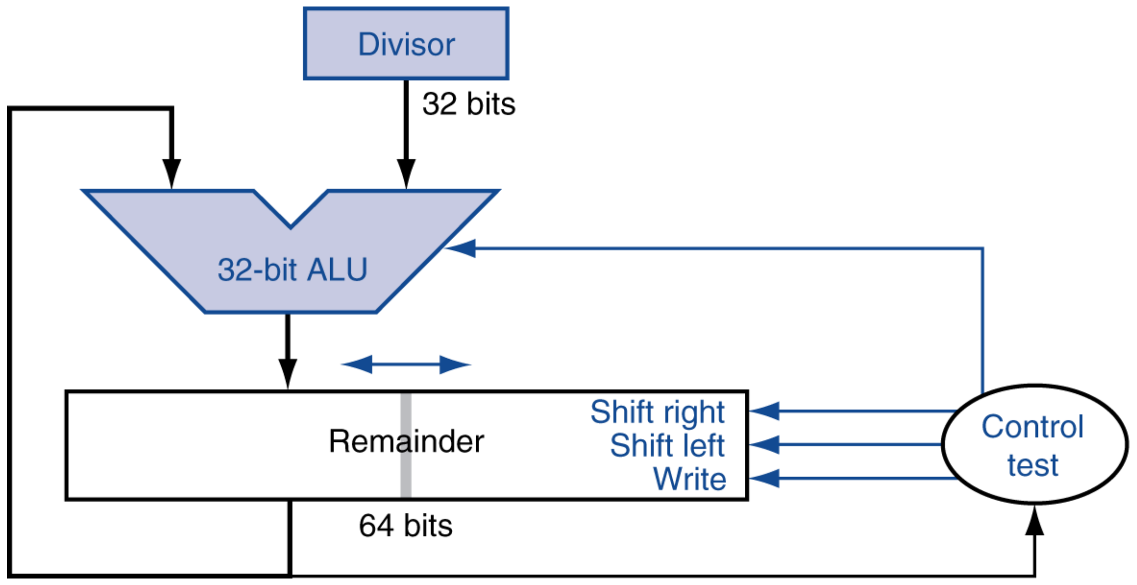


Division Hardware



Finish!

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 (e.g. SRT devision) 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

Homework

- Chapter3: 3.9 3.10 3.12 3.13