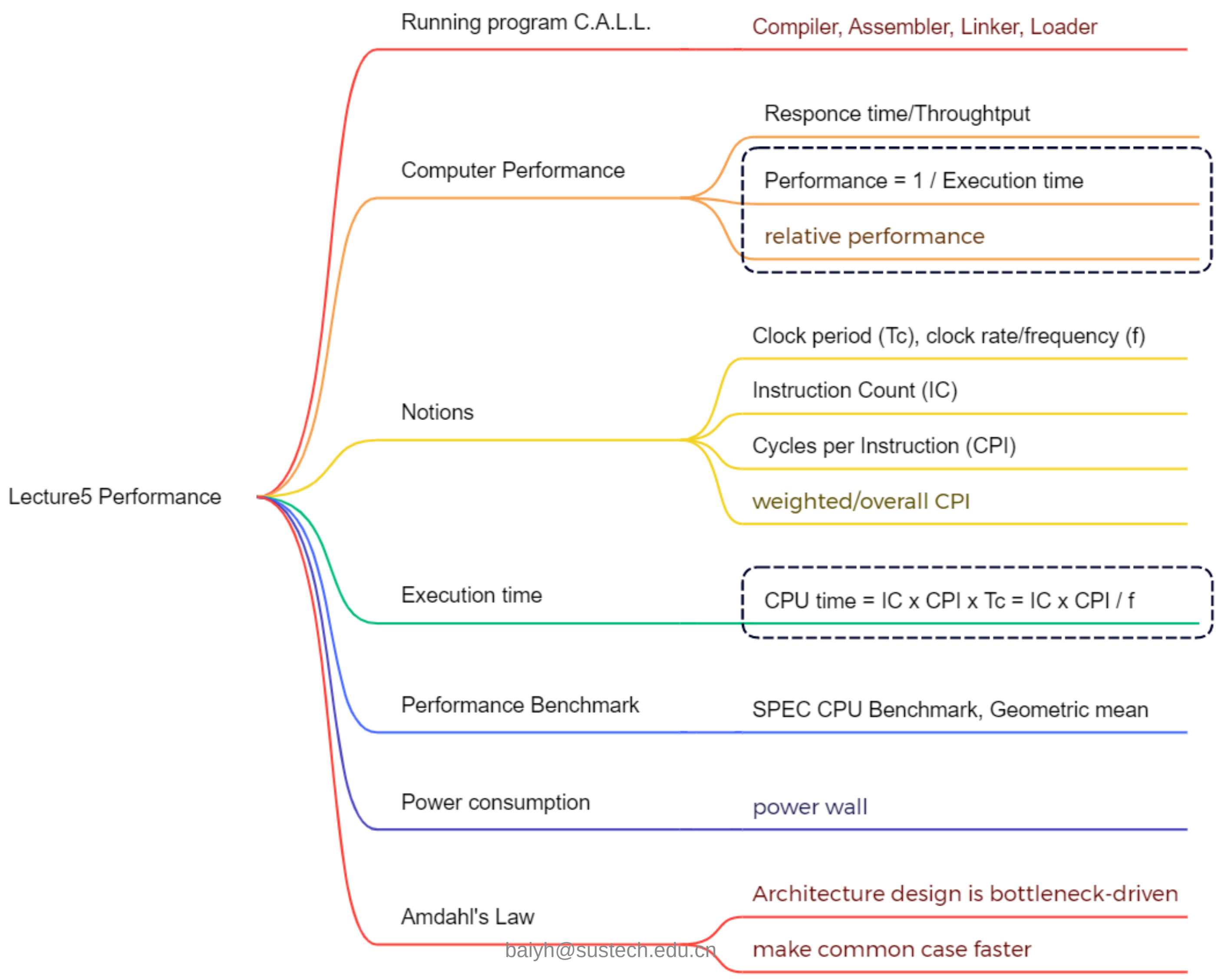
COMPUTER ORGANIZATION

Lecture 6
Arithmetic Operations on Integers

2025 Spring

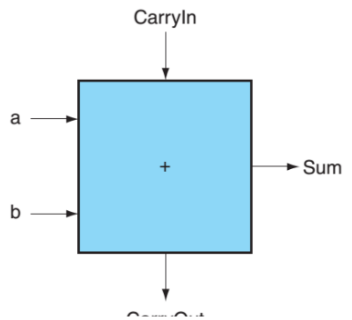
baiyh@sustech.edu.cn

Recap



Lecture5 Performance
Running program C.A.L.L.
Compiler, Assembler, Linker, Loader
Computer Performance
Responce time/Throughtput
Performance = 1 / Execution time
relative performance
Notions
Clock period (Tc), clock rate/frequency (f)
Instruction Count (IC)
Cycles per Instruction (CPI)
weighted/overall CPI
Execution time
CPU time = IC x CPI x Tc = IC x CPI / f
Performance Benchmark
SPEC CPU Benchmark, Geometric mean
Power consumption
power wall
Amdahl's Law
Architecture design is bottleneck-driven
make common case faster

1-bit adder



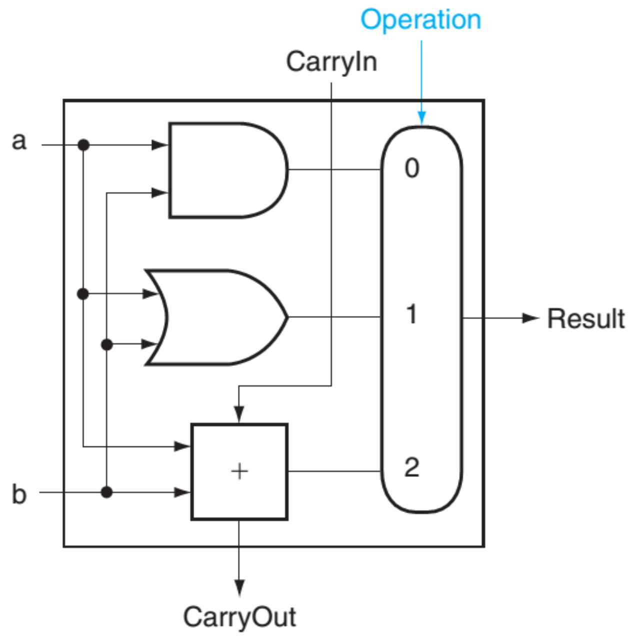
$$\text{Sum} = (a \cdot \overline{b} \cdot \overline{\text{CarryIn}}) + (\overline{a} \cdot b \cdot \overline{\text{CarryIn}}) + (\overline{a} \cdot \overline{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$$

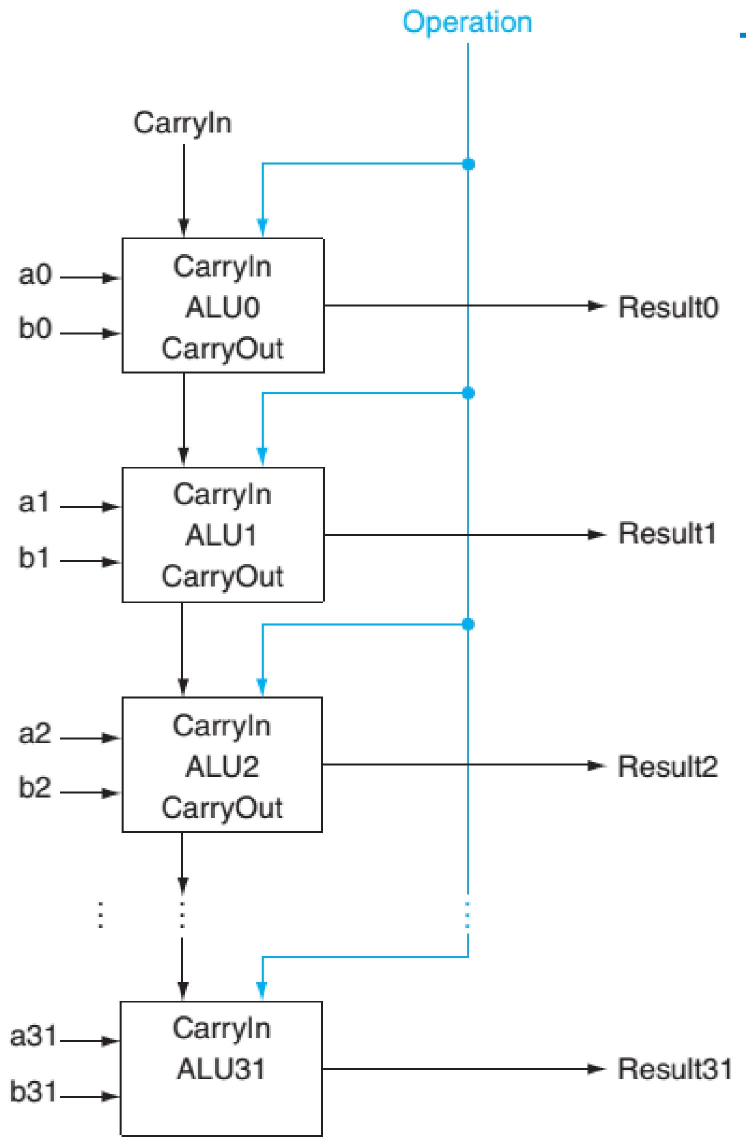
$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b)$$

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 ALU and 32-bit ALU

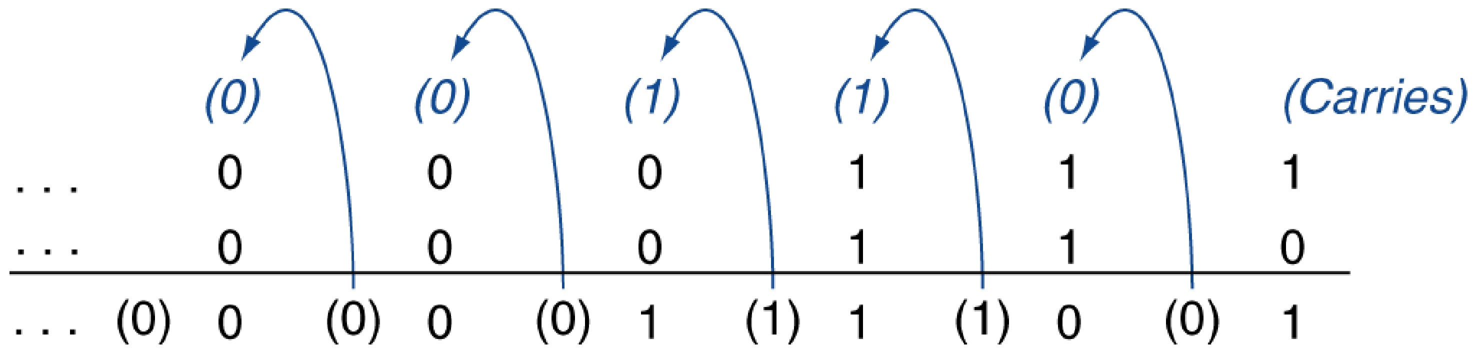
- ALU: arithmetic logical unit
- 1-bit ALU and 32-bit ALU
  - If op = 0, o = a & b (and)
  - If op = 1, o = a | b (or)
  - If op = 2, o = a + b (add)





Integer Addition

- Example: 7 + 6



- Overflow if result out of range
  - For signed integer addition:
  - no overflow, if adding +ve(positive) and –ve(negative) operands
  - Overflow, if:
    - Adding two +ve operands, get –ve operand
    - Adding two -ve operands, get +ve operand

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
  - 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)

Overflow Example

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

- Example, 8-bit signed operation:
  - 12+3=15   120+15=135   12-3=9   -100-50=-150

12-3 = 12+(-3)   -100-50=-100+(-50)

```
   00001100       01111000       00001100       10011100
+  00000011    +  00001111    +  11111101    +  11001110
   --------       --------       --------       --------
   00001111       10000111       00001001       01101010
```

Overflow

Overflow Detection for Signed & Unsigned Addition

- Signed addition

```
add  t0, t1,   t2          # t0 = sum
xor  t3, t1,   t2          # Check if signs differ
slt  t3, t3,   zero        # t3 = 1 if signs differ
bne  t3, zero, No_overflow # t1, t2 signs ≠,no overflow
xor  t3, t0,   t1          # t1, t2 signs =, check sum
slt  t3, t3,   zero        # t3 = 1 if sum sign≠
bne  t3, zero, Overflow    # sum signs ≠ operands; overflow
```

- Unsigned addition(unsigned overflow is also called carry)

```
add  t0, t1,   t2          # t0 = sum
xori t3, t1,   -1          # t3 = NOT t1 (i.e. 2^32–1 – t1)
sltu t3, t3,   t2          # (2^32-1 - t1) < t2
bne  t3, zero, Overflow    # if(2^32–1 < t1+t2), overflow
```

Arithmetic for Multimedia – Saturating Operation

- 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
    - c.f. 2s-complement modulo arithmetic
  - E.g., change the volume and brightness in audio or video

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

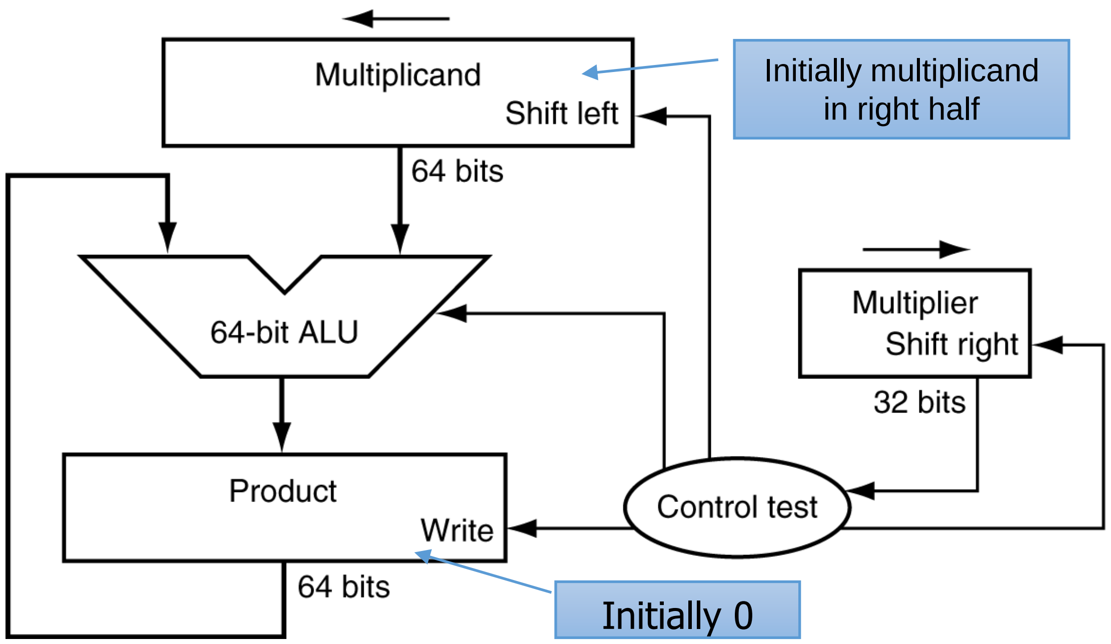
multiplicand

multiplier

product

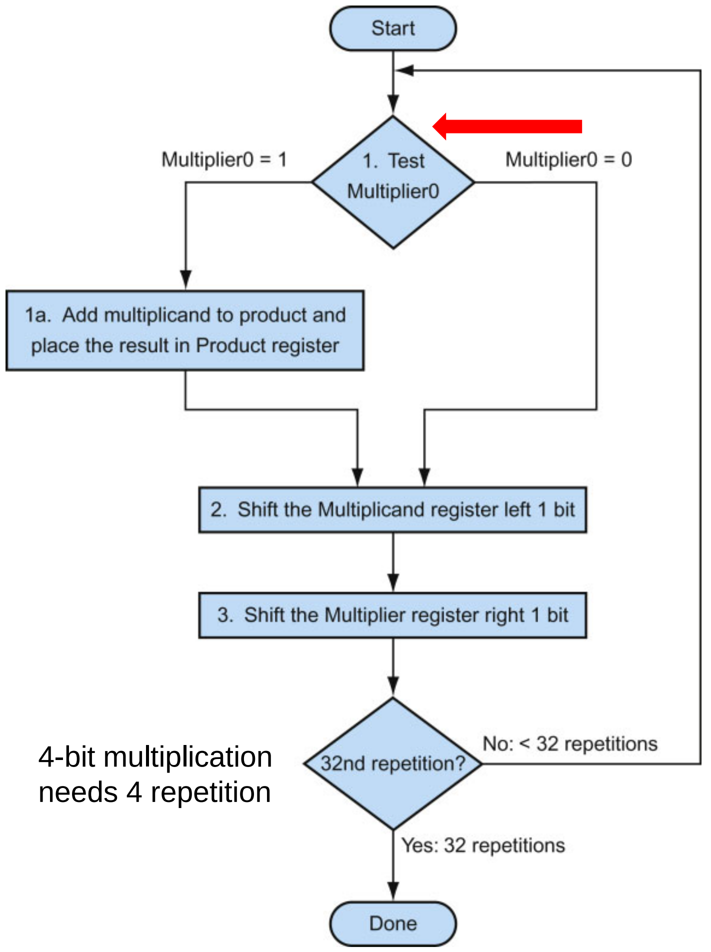
```
        1000
   ×    1001
   ---------
        1000
       0000
      0000
     1000
   ---------
     1001000
```

Length of product is the sum of operand lengths



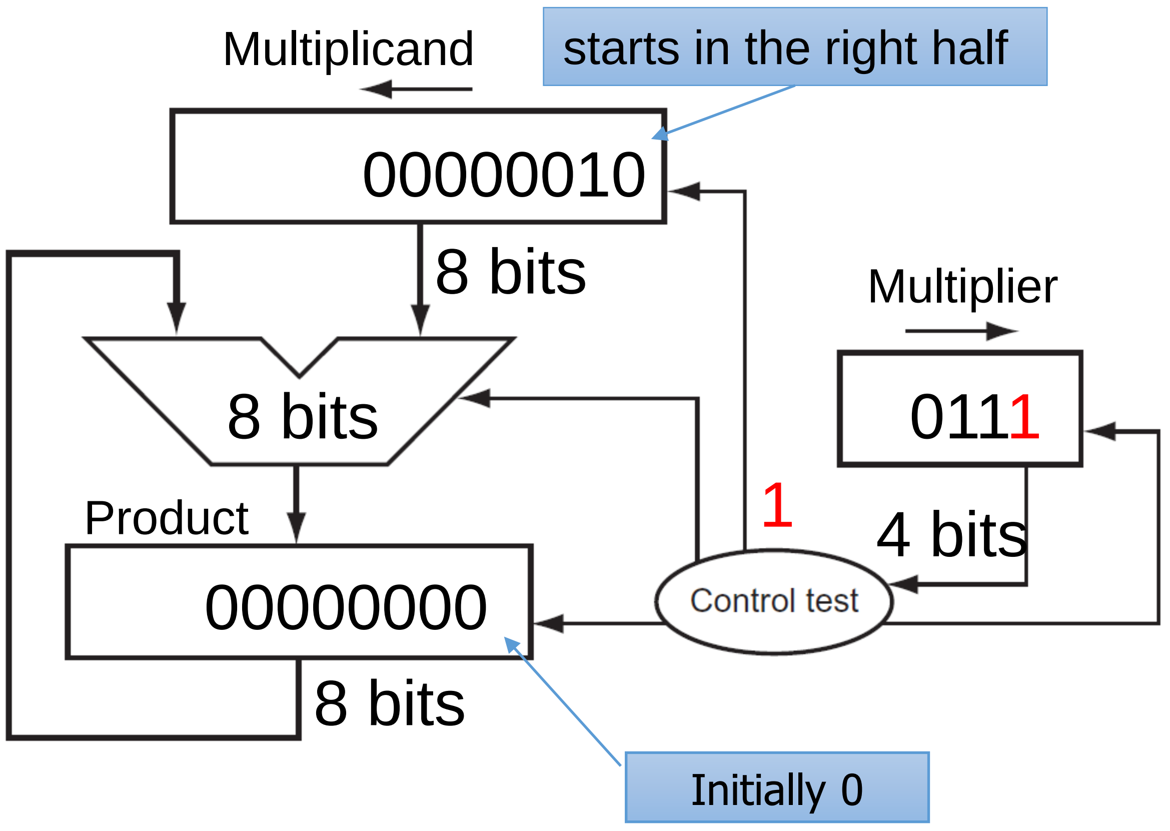
Multiplication Example

- 4-bit version Multiply 2_{ten} (0010_{two}) by 7_{ten} (0111_{two})



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

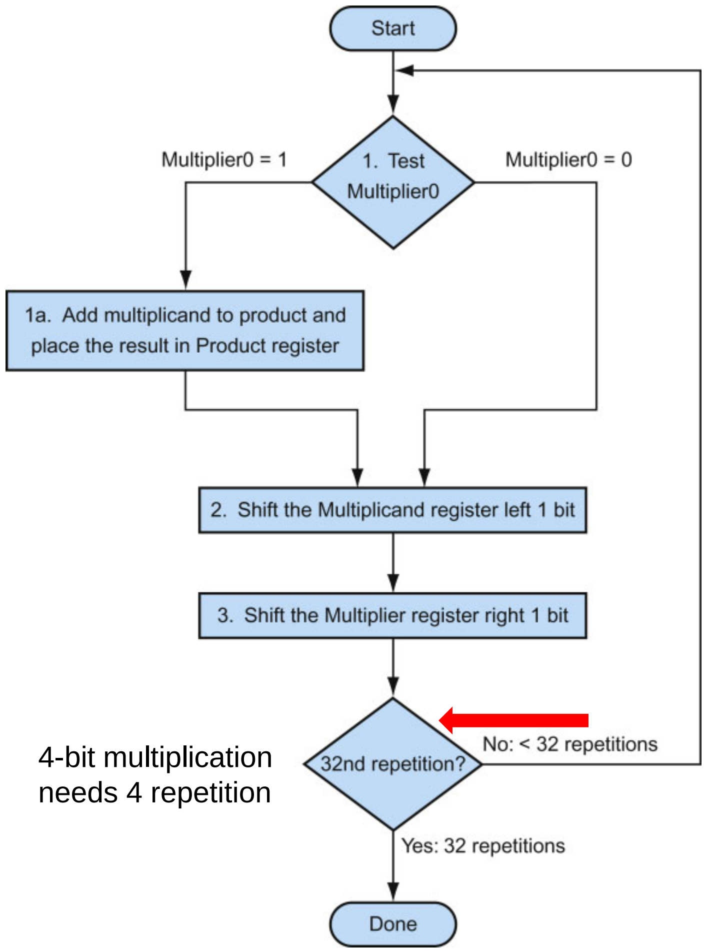
4-bit multiplication needs 4 repetition



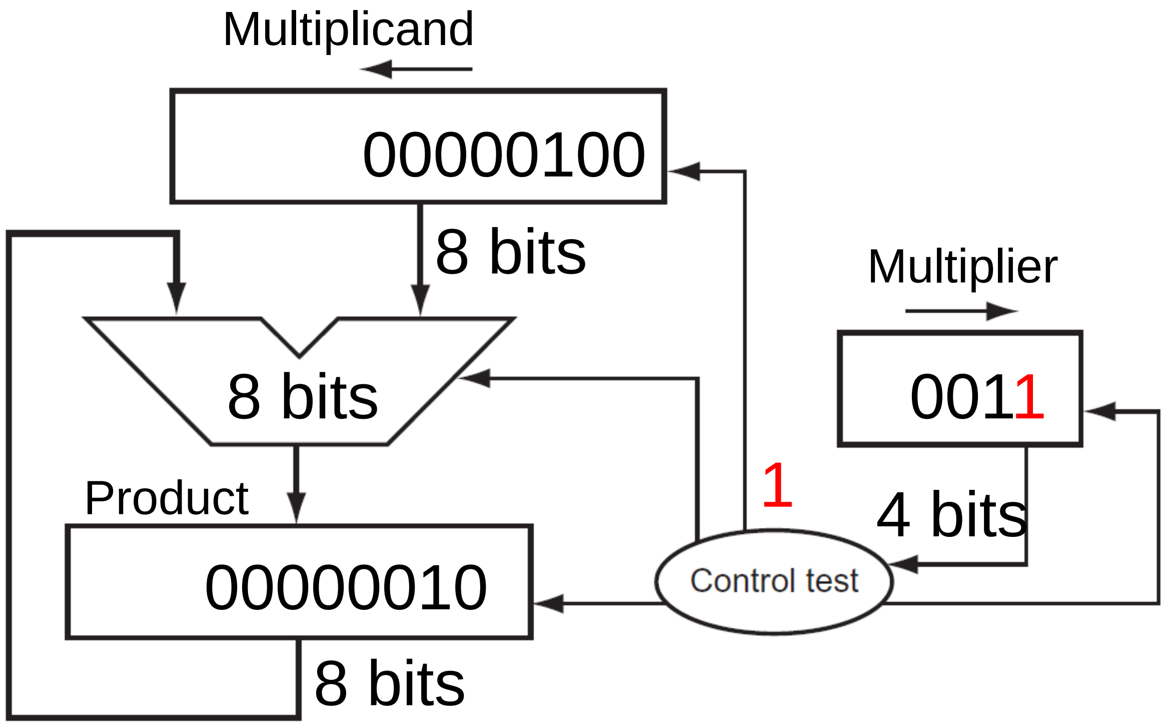
Initial

Multiplication Example

- Multiply 2_{ten} (0010_{two}) by 7_{ten} (0111_{two})


Start
Multiplier0 = 1
1. Test Multiplier0
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

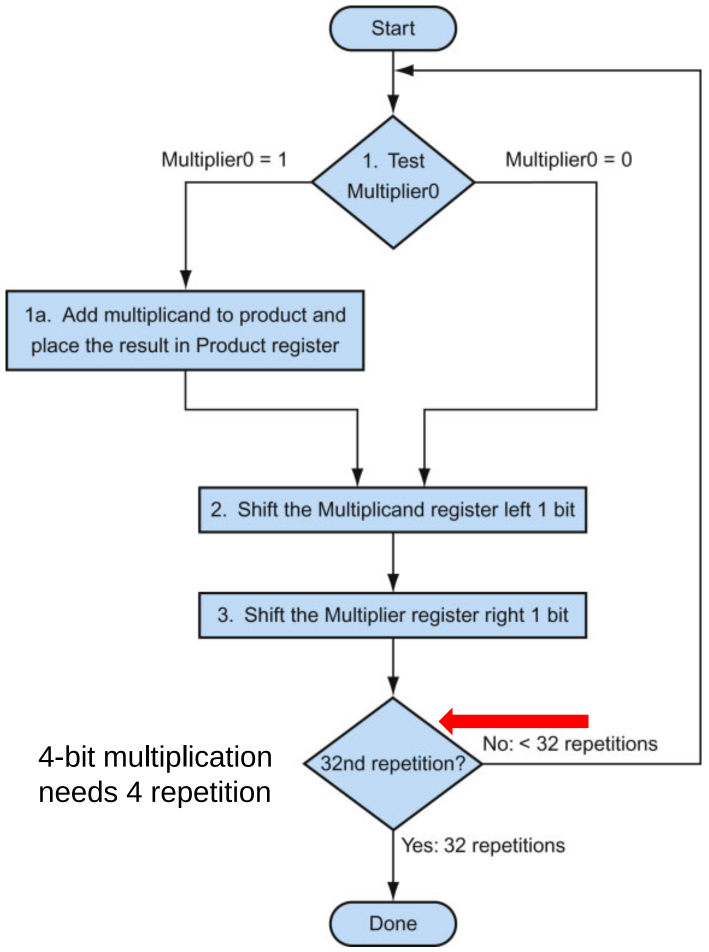
4-bit multiplication needs 4 repetition

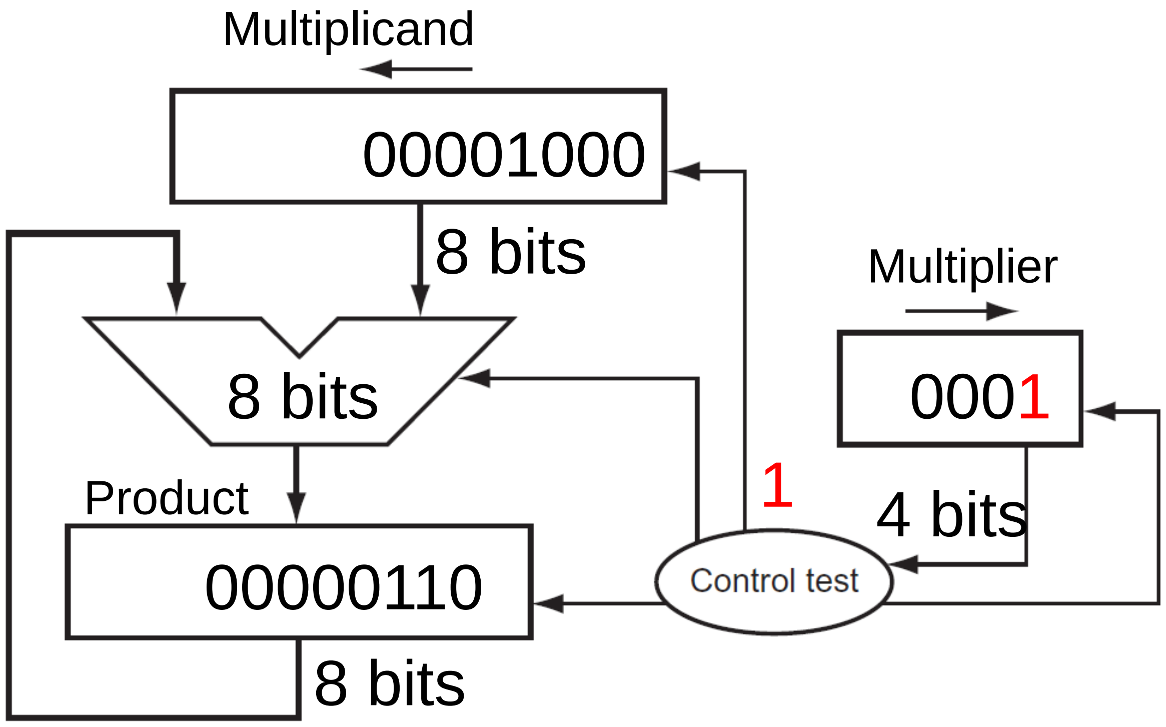


end of iteration 1

Multiplication Example

- Multiply 2_{ten} (0010_{two}) by 7_{ten} (0111_{two})

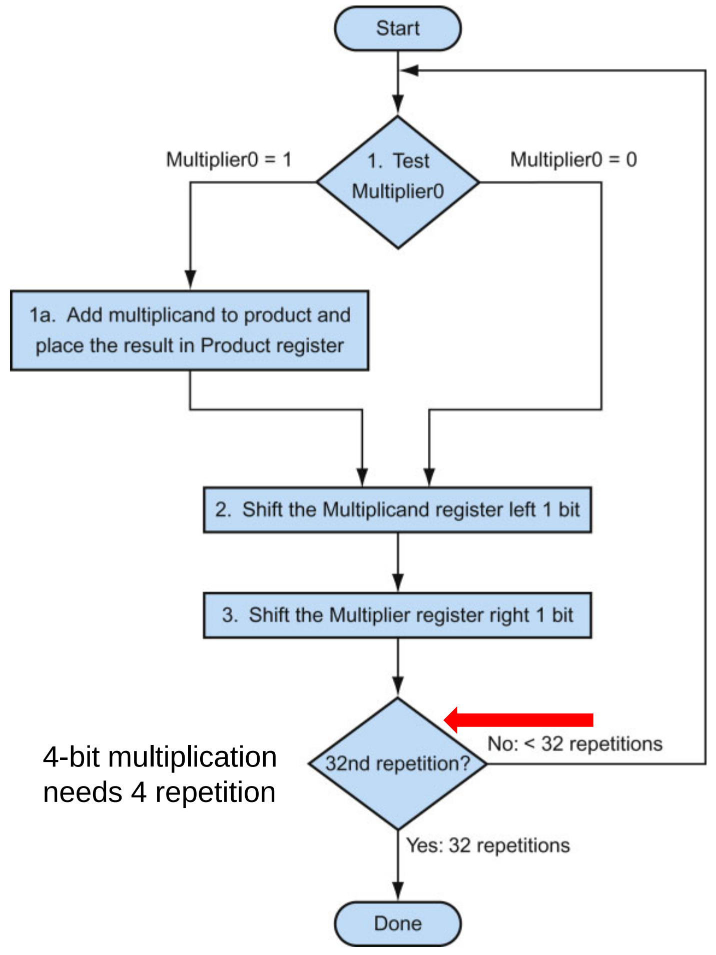

Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done
4-bit multiplication needs 4 repetition



end of iteration 2

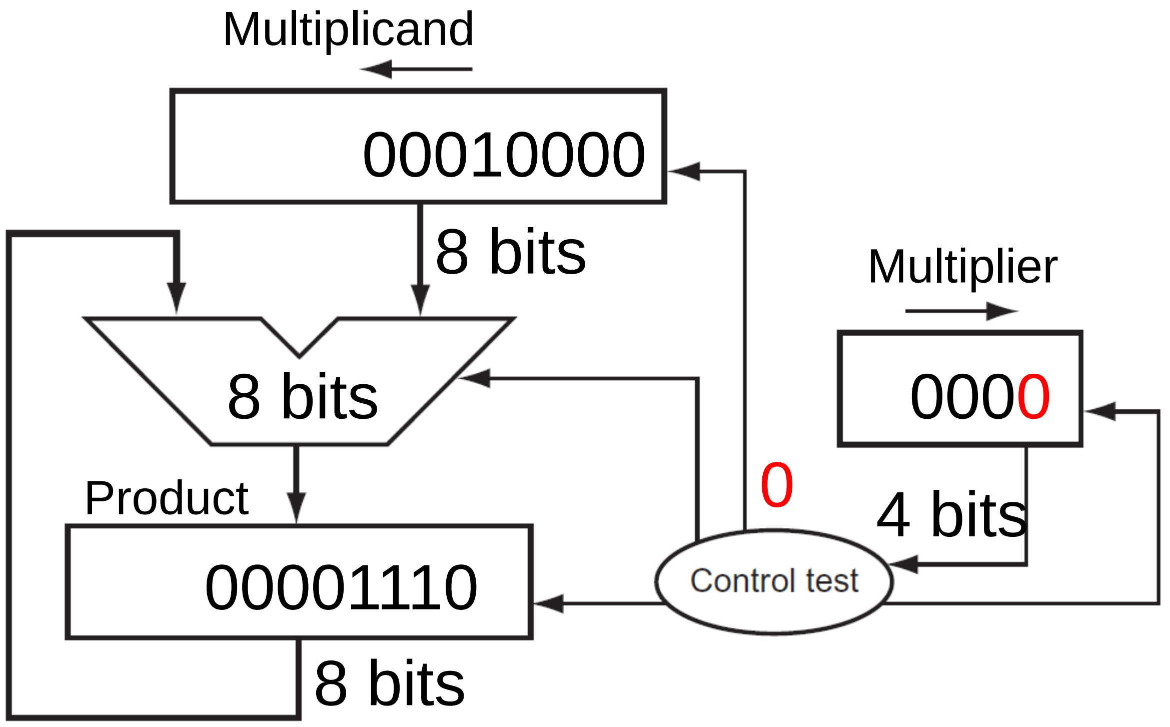
Multiplication Example

- Multiply 2_{ten} (0010_{two}) by 7_{ten} (0111_{two})



Start
Multiplier0 = 1
1. Test Multiplier0
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

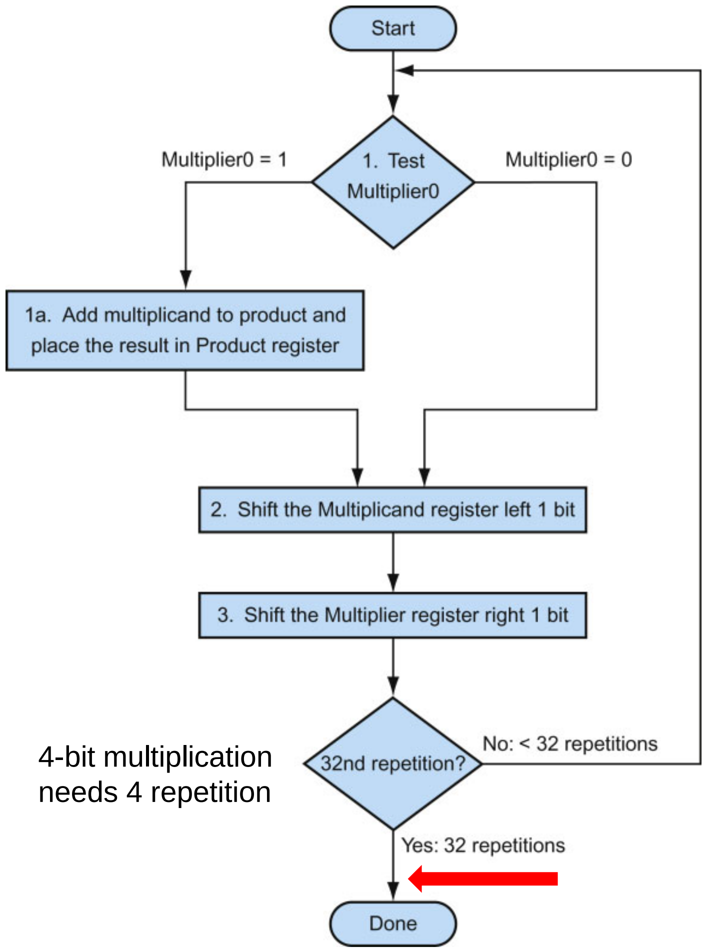
4-bit multiplication needs 4 repetition



end of iteration 3

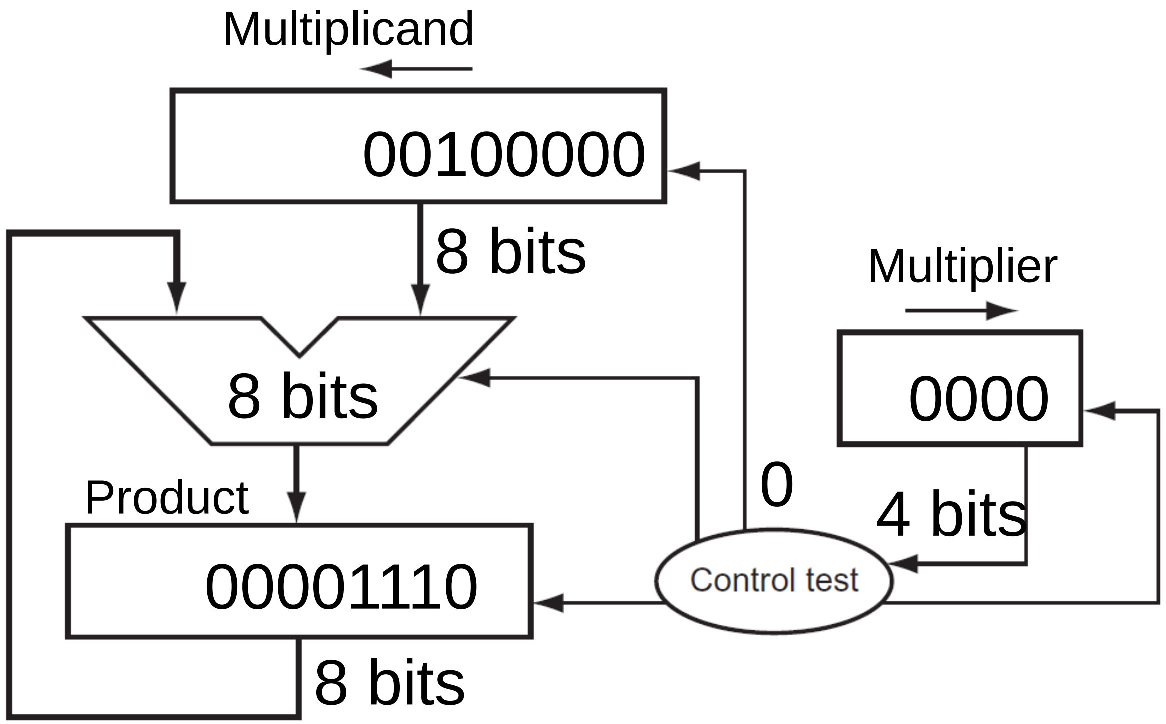
Multiplication Example

- Multiply 2_{ten} (0010_{two}) by 7_{ten} (0111_{two})



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

4-bit multiplication needs 4 repetition



Done!

end of iteration 4

Multiplication Example

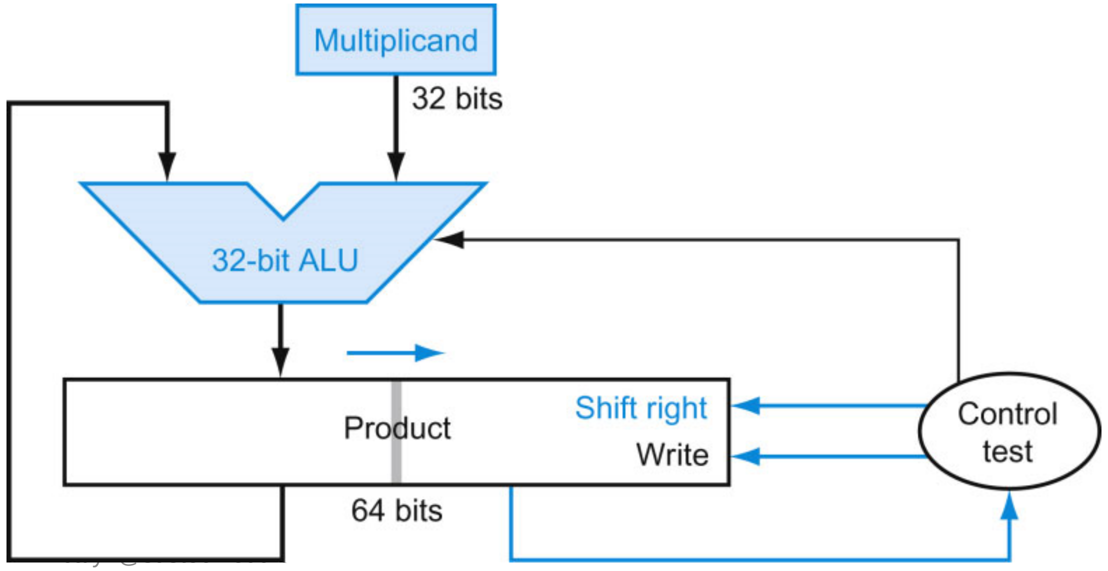
- Multiply 2_{ten} (0010_{two}) by 7_{ten} (0111_{two})
  - How values change in Mcand, Mplier and Product Registers?

Iter	Step	Multiplier	Multiplicand	Product
0	Initial values	0111	0000 0010	0000 0000
1	1 ⇒ Prod = Prod + Mcand Shift left Multiplicand Shift right Multiplier	0111 0111 0011	0000 0010 0000 0100 0000 0100	0000 0010 0000 0010 0000 0010
2	Same steps as 1	0011 0011 0001	0000 0100 0000 1000 0000 1000	0000 0110 0000 0110 0000 0110
3	Same steps as 1	0001 0001 0000	0000 1000 0001 0000 0001 0000	0000 1110 0000 1110 0000 1110
4	0 ⇒ No operation Shift left Multiplicand Shift right Multiplier	0000 0000 0000	0001 0000 0010 0000 0010 0000	0000 1110 0000 1110 0000 1110

Slow Multiplier → Optimized version

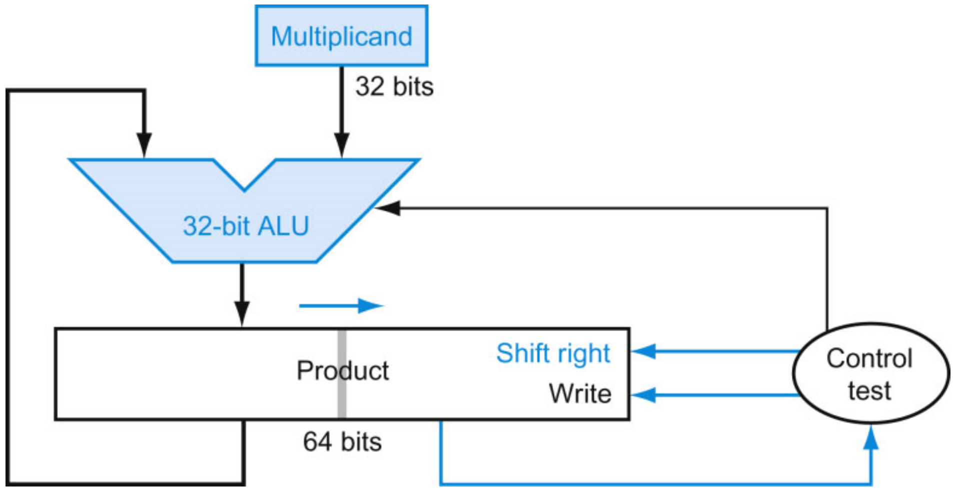
- Oberservation
  - Half of the bits in multiplicand always 0
    - 64-bit adder is wasted
  - 0’s inserted in right of multiplicand as shifted
    - least significant bits of product never changed once formed
- Instead of shifting multiplicand to left, shift product to right?
- Product register wastes space => combine Multiplier and Product register

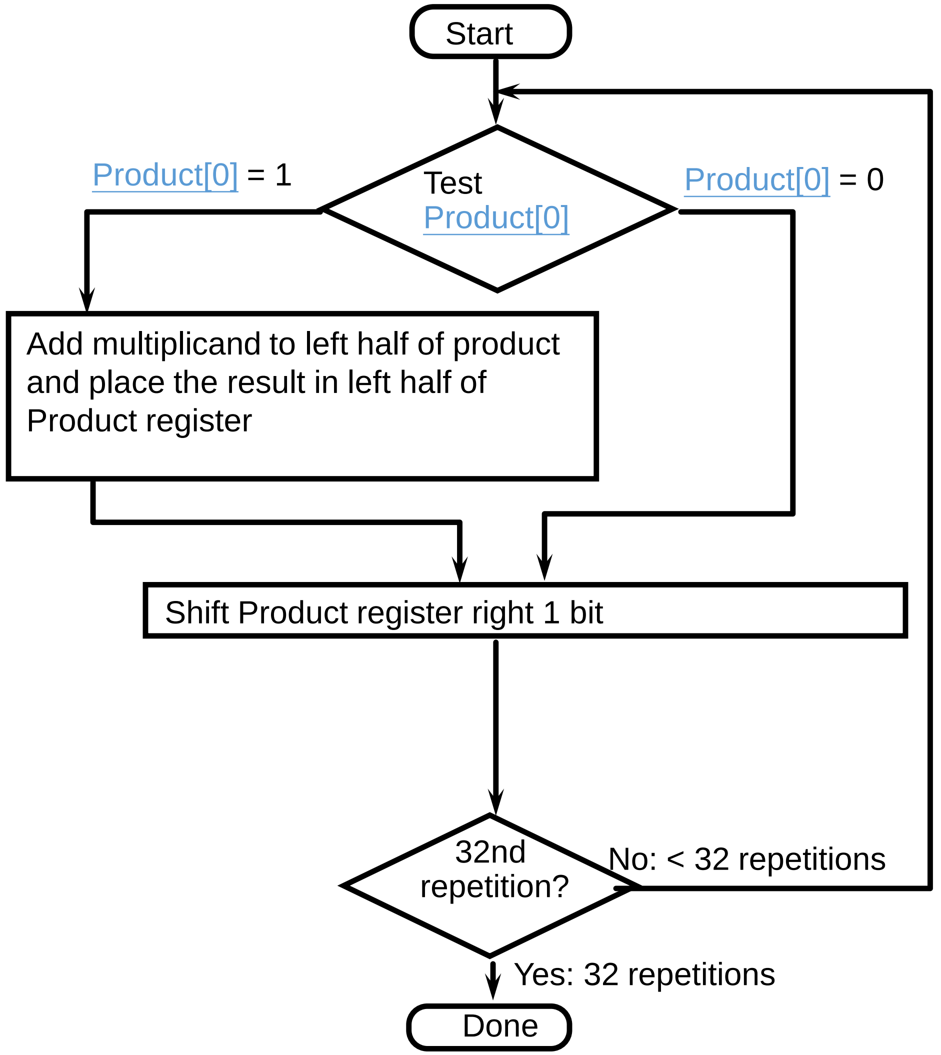
```
        1000
  ×     1001
  ----------
        1000
       00000
      000000
     1000000
  ----------
     1001000
```



Optimized Multiplier

- Multiplier initially in right half of product register, 32-bit ALU and multiplicand is untouched
- Check the 0th 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

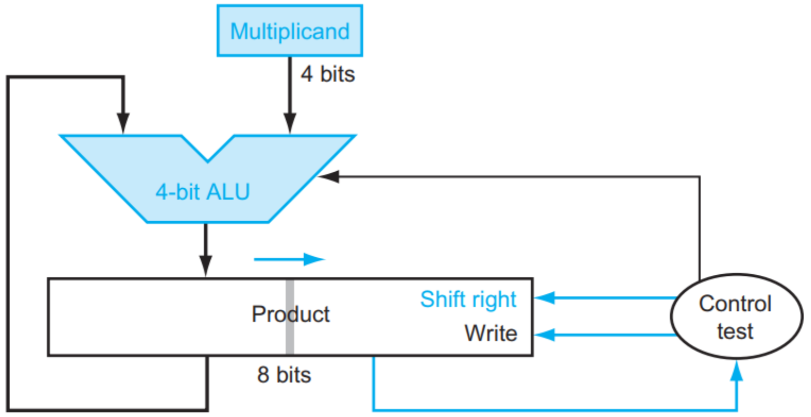



Start
Test Product[0]
Product[0] = 1
Product[0] = 0
Add multiplicand to left half of product and place the result in left half of Product register
Shift Product register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

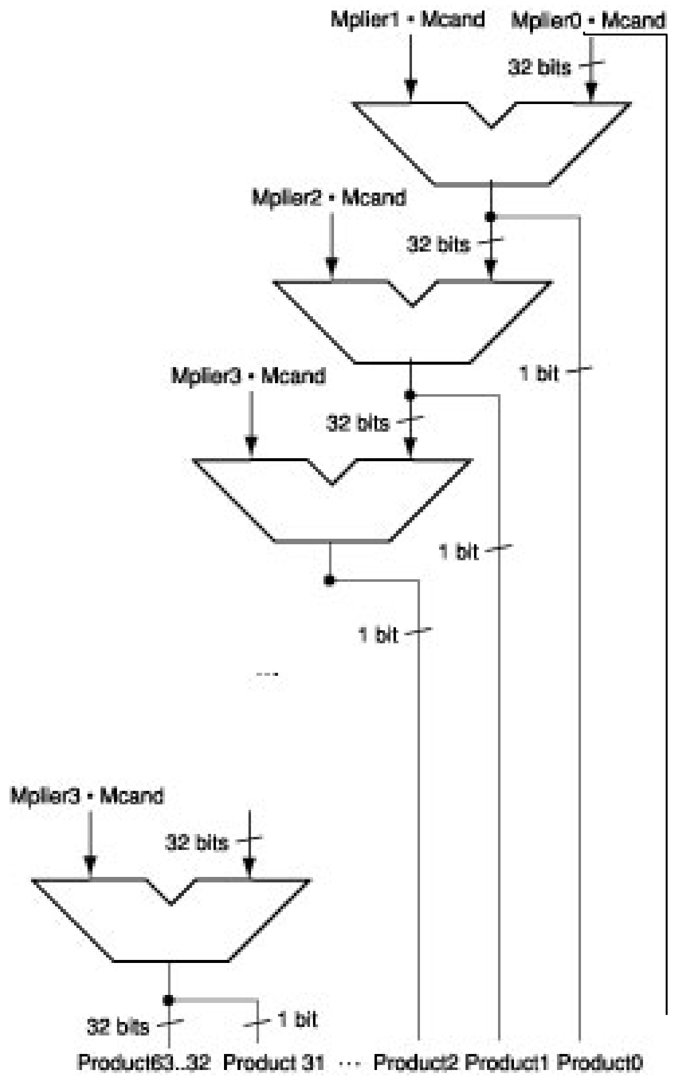
Optimized Multiplier Example

- Example:
  - Multiply 2_{ten} (0010_{two}) by 7_{ten} (0111_{two})
  - result = 00001110_{two} (14_{ten})

iter	Multiplicand	Product	Operation
0	0010	0000 0111	
1	0010	0010 0111	**1**: Prod left half accumulate
	0010	0001 0011	Shift right Prod
2	0010	0011 0011	**1**: Prod left half accumulate
	0010	0001 1001	Shift right Prod
3	0010	0011 1001	**1**: Prod left half accumulate
	0010	0001 1100	Shift right Prod
4	0010	0000 1110	**0**: Shift right Prod
		res=00001110	done



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

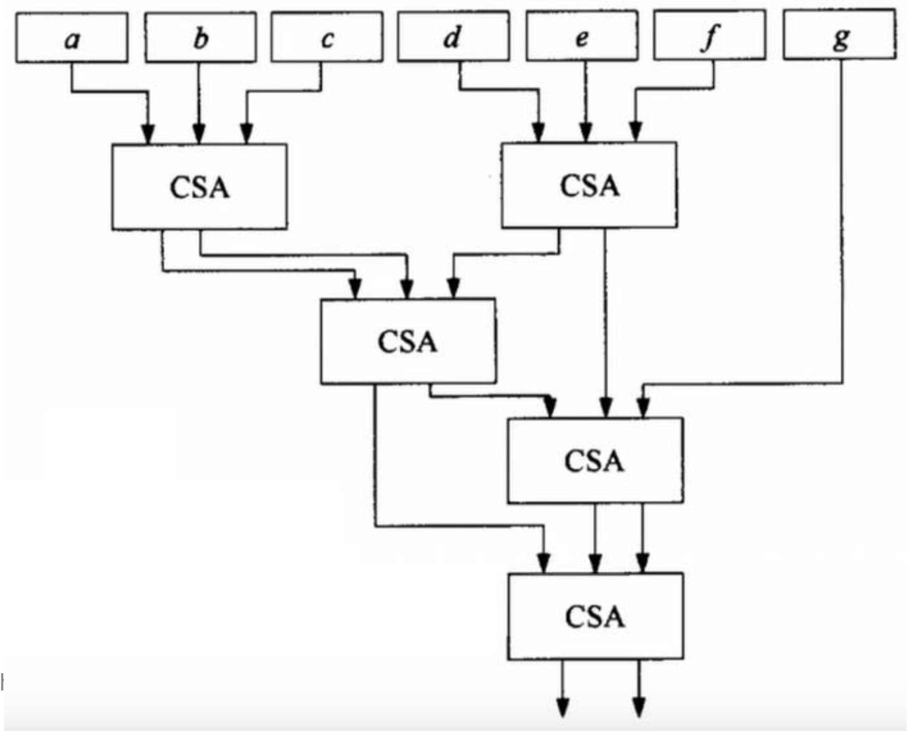
Faster Multiplier

- Wallace Tree (Carry Save)
  - Use Carry-Save Adder for partial products addition
  - Carry-save adder passes (saves) the carries to the output, rather than propagating them.
  - With this technique, we can avoid carry propagation until final addition
  - Carry-save is fast (no carry propagation) and inexpensive (full adders)

```
        0   1   0   1
        0   1   1   0
    +   1   0   0   1
  -------------------
    0   1   0   1   0   carry
+   0   1   0   1   0   sum
  -------------------
    1   0   1   0   0
```

```
a: 0101
b: 0110
c: 1001
carry save: a+b+c = c+s
```

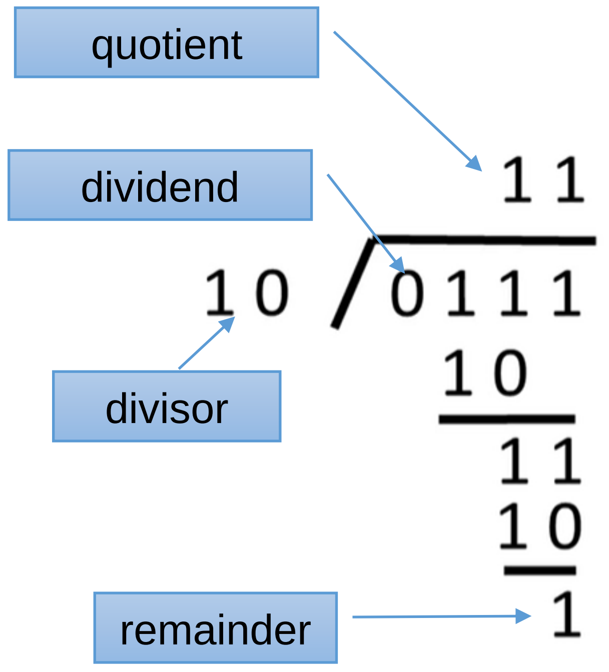
a~g : partial products



RISC-V Multiplication

- Four multiply instructions:
  - mul: multiply
    - Gives the lower 32 bits of the product
  - mulh: multiply high
    - Gives the upper 32 bits of the product, assuming the operands are signed
  - mulhu: multiply high unsigned
    - Gives the upper 32 bits of the product, assuming the operands are unsigned
  - mulhsu: multiply high signed/unsigned
    - Gives the upper 32 bits of the product, assuming one operand is signed and the other unsigned

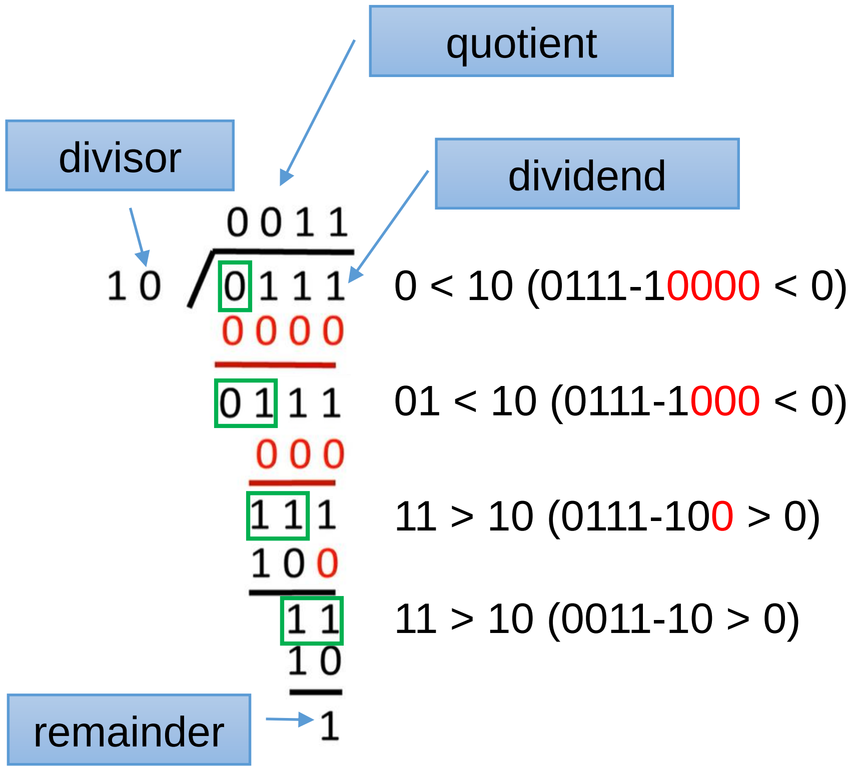
Division



n-bit operands yield n-bitquotient 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

Division Hardware

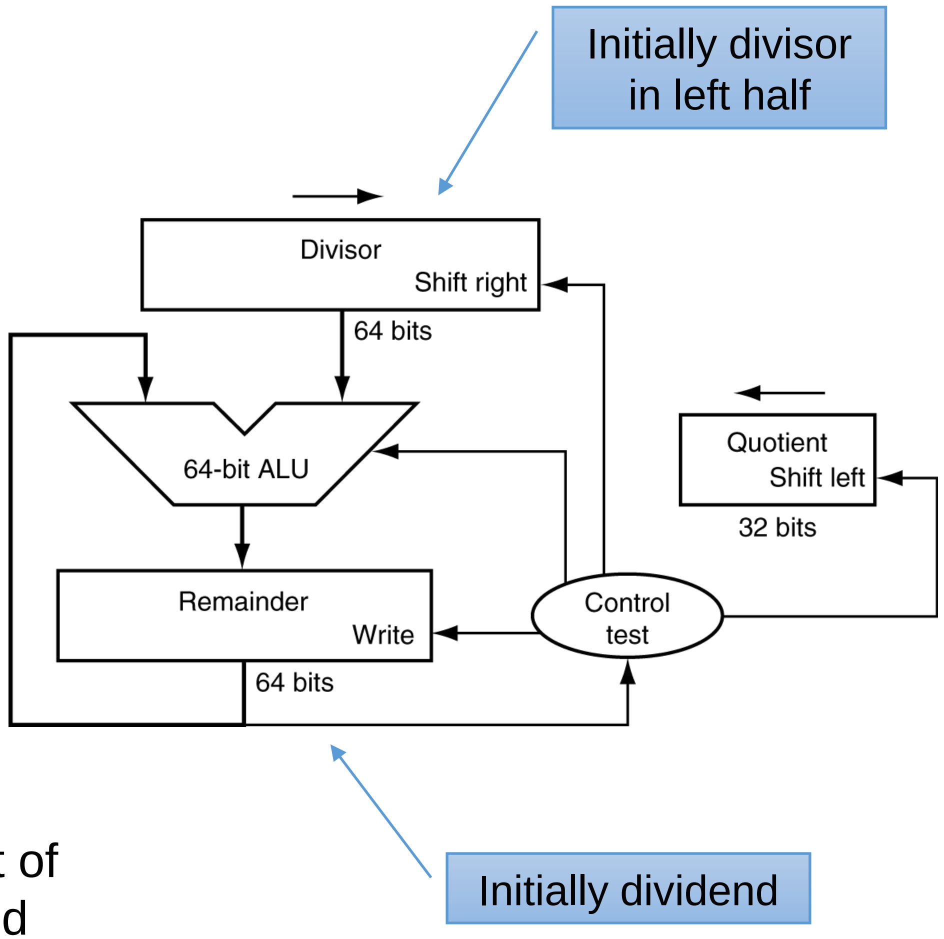


0 < 10 (0111-10000 < 0)

01 < 10 (0111-1000 < 0)

11 > 10 (0111-100 > 0)

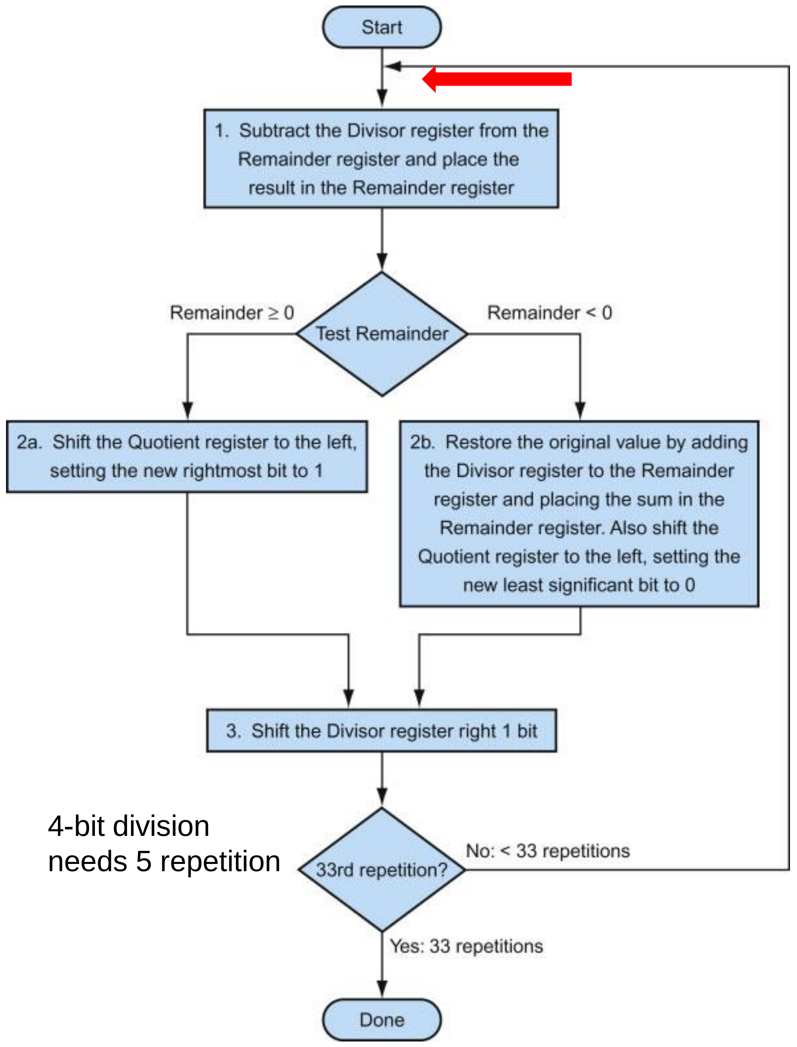
11 > 10 (0011-10 > 0)



Remainder: the secondary result of a division, is initialized to dividend

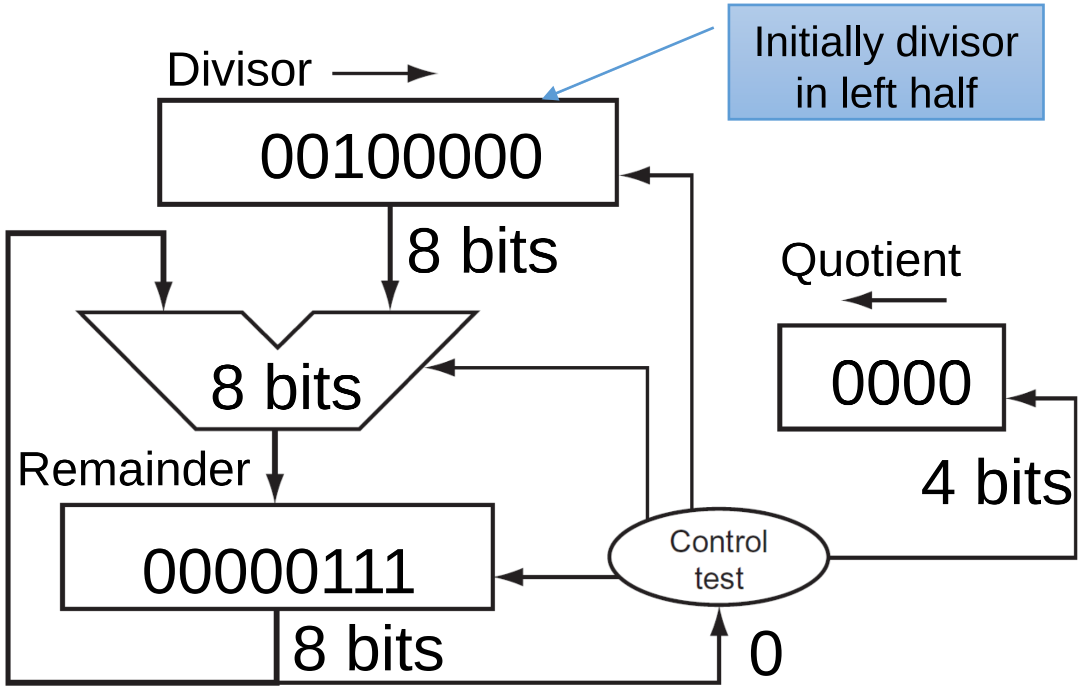
Division Example

- 4-bit version Divide 7_{ten} (0111_{two}) by 2_{ten} (0010_{two})



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

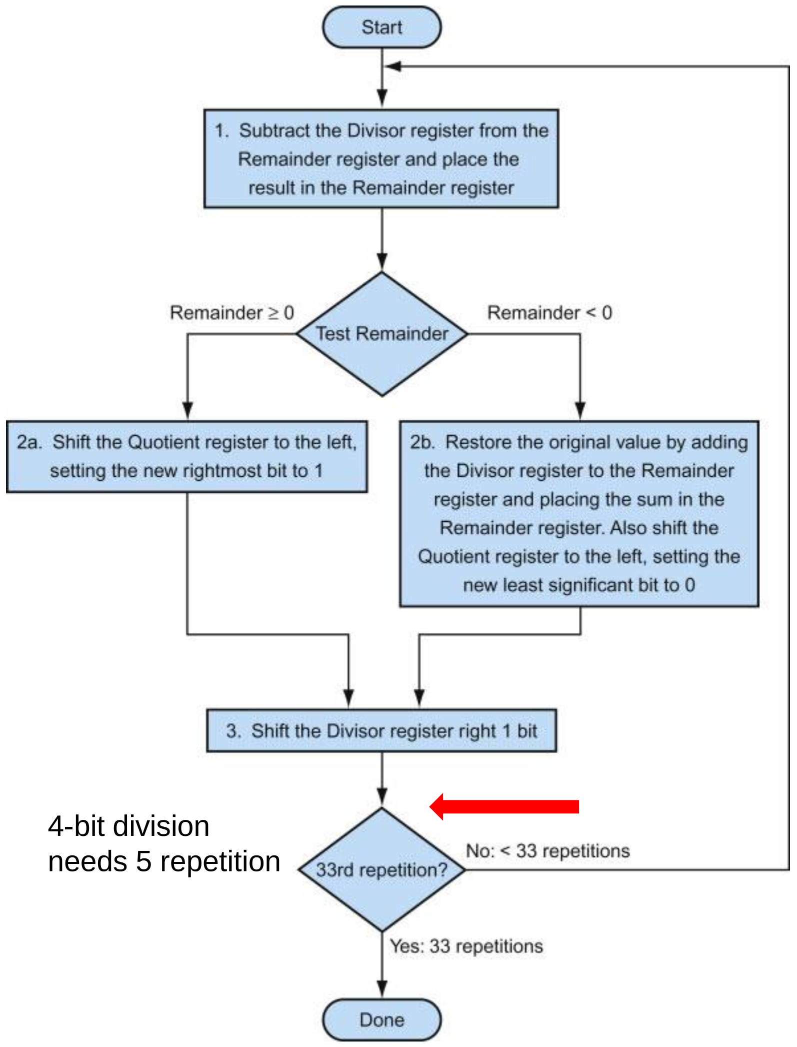
4-bit division needs 5 repetition



initial

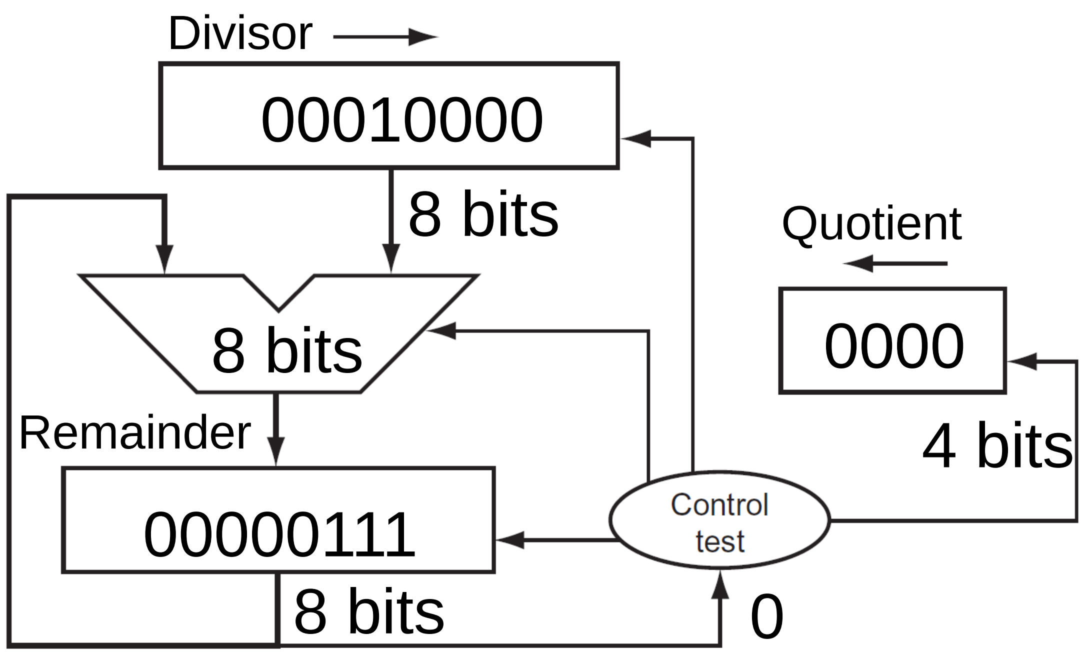
Division Example

- Divide 7_{ten} (0111_{two}) by 2_{ten} (0010_{two})



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

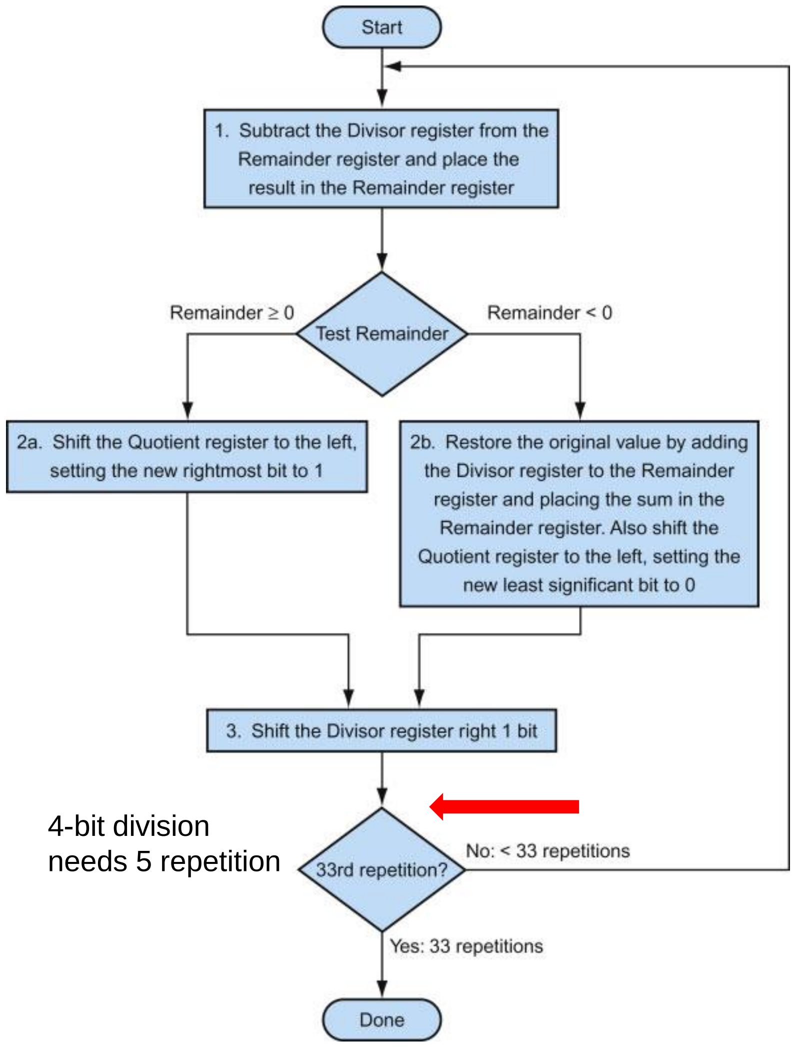
4-bit division
needs 5 repetition



end of iteration 1

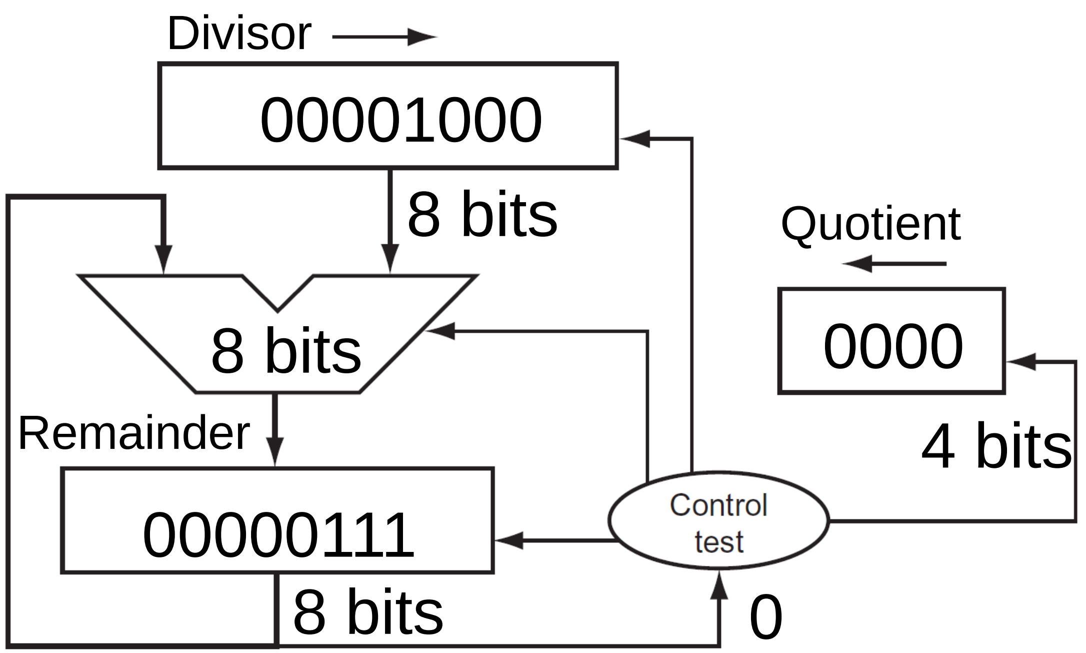
Division Example

- Divide 7_{ten} (0111_{two}) by 2_{ten} (0010_{two})



Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Remainder ≥ 0
Test Remainder
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

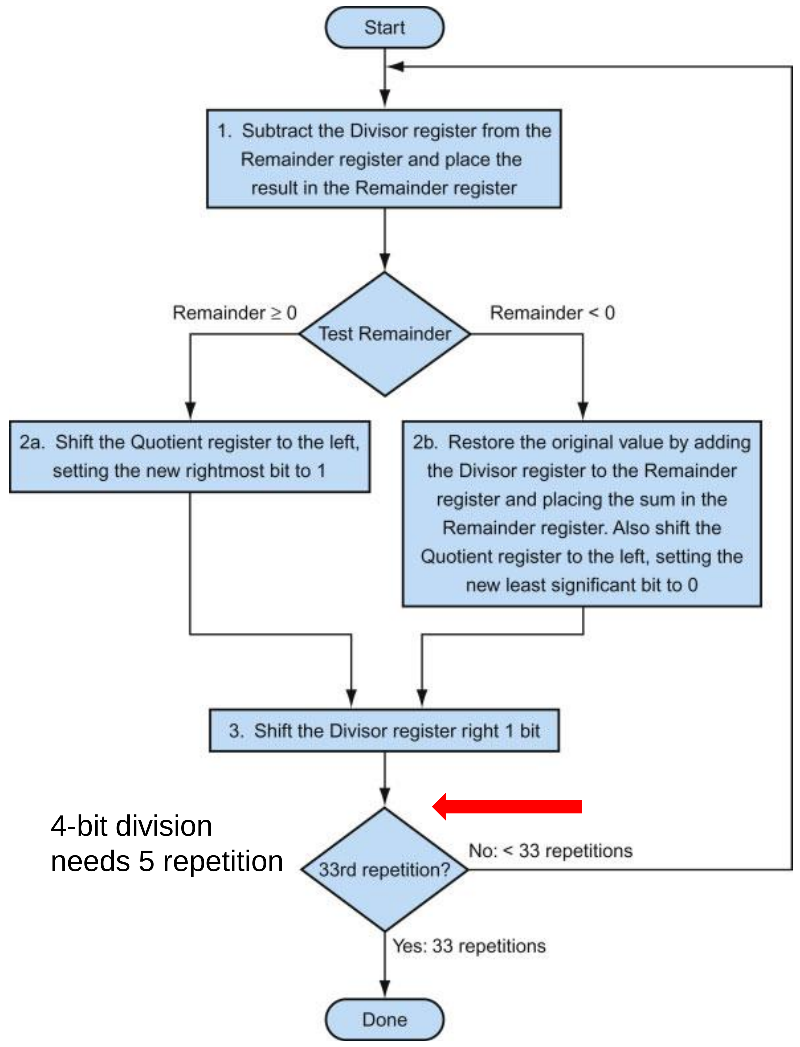
4-bit division needs 5 repetition



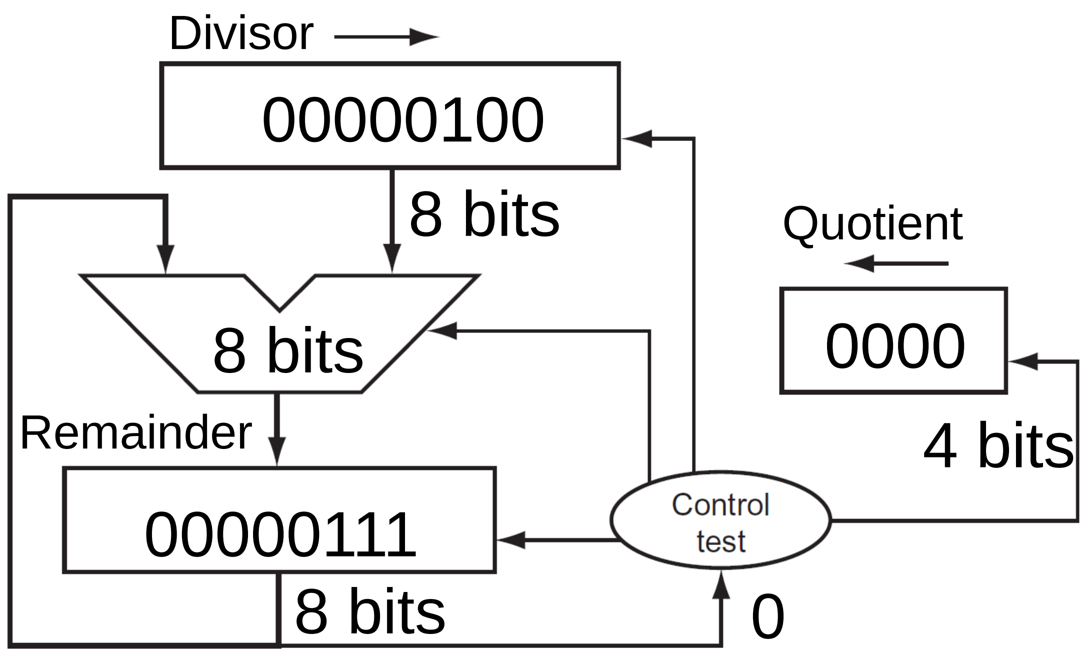
end of iteration 2

Division Example

- Divide 7_{ten} (0111_{two}) by 2_{ten} (0010_{two})


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

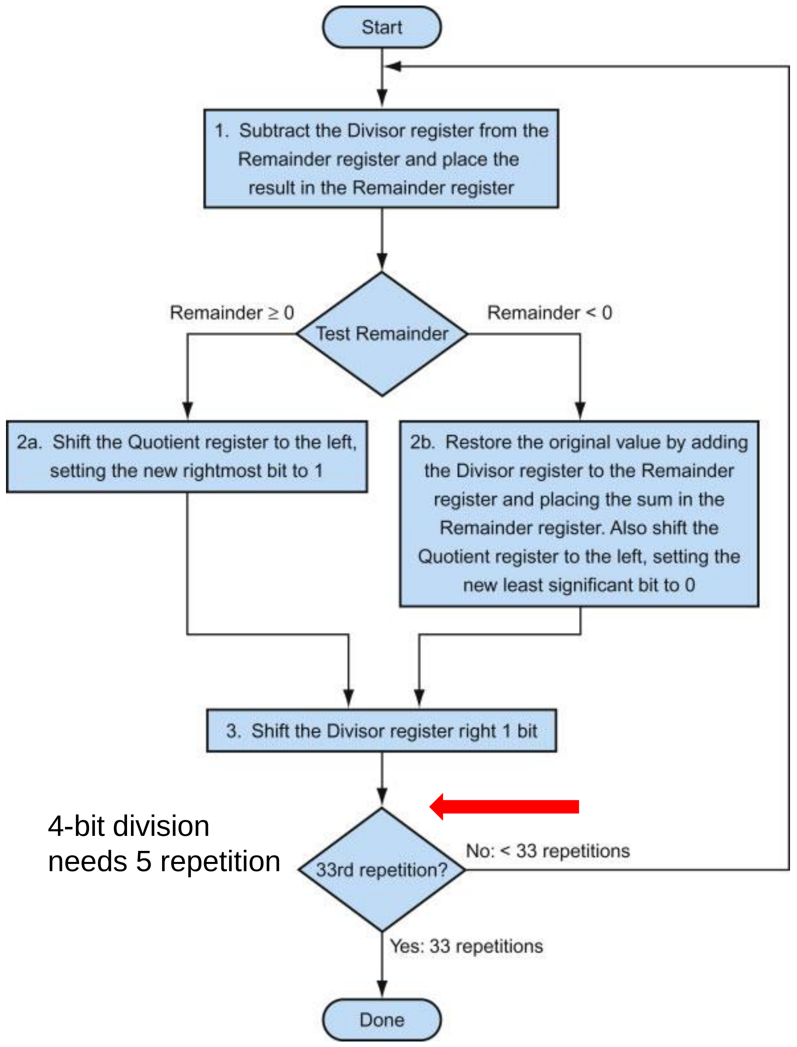
4-bit division
needs 5 repetition



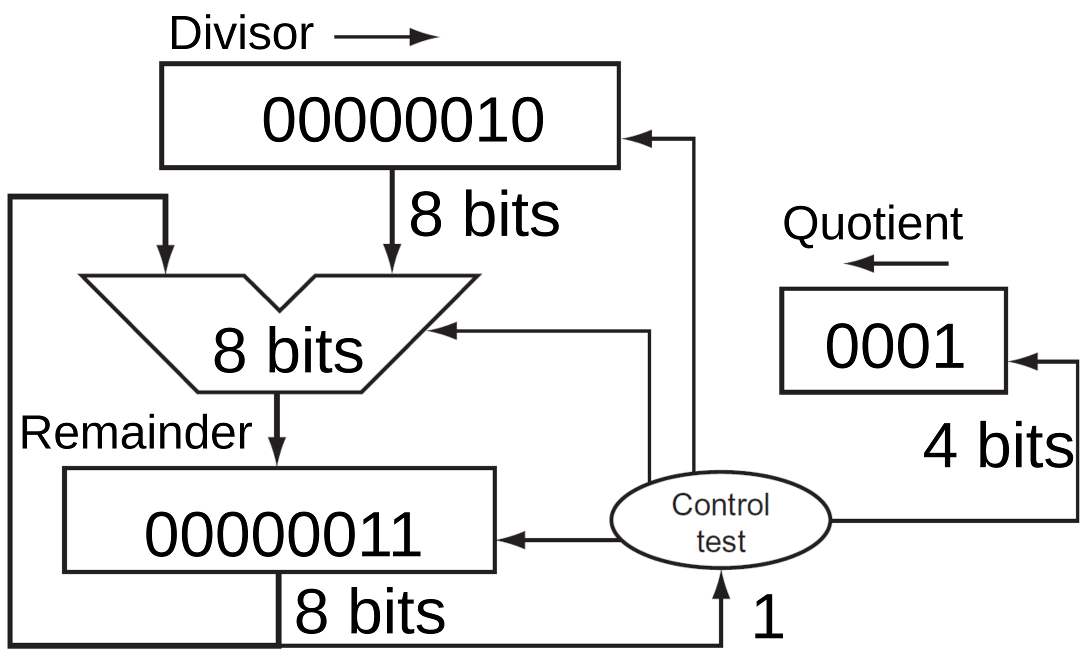
end of iteration 3

Division Example

- Divide 7_{ten} (0111_{two}) by 2_{ten} (0010_{two})


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

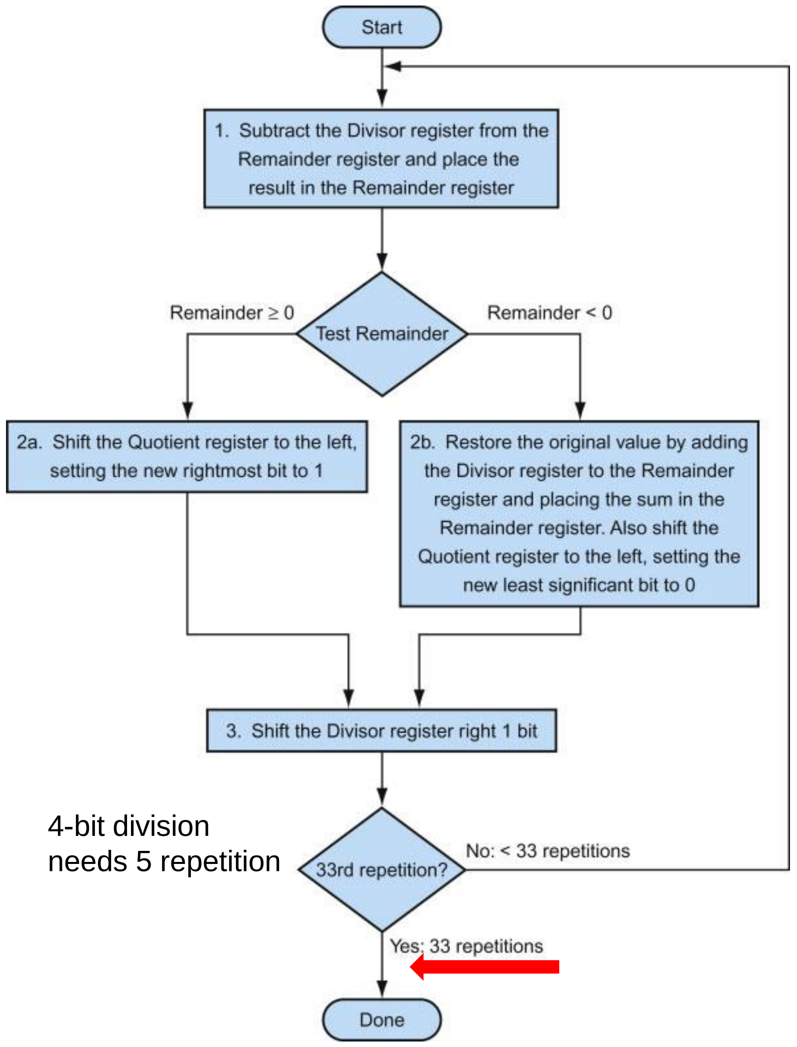
4-bit division
needs 5 repetition



end of iteration 4

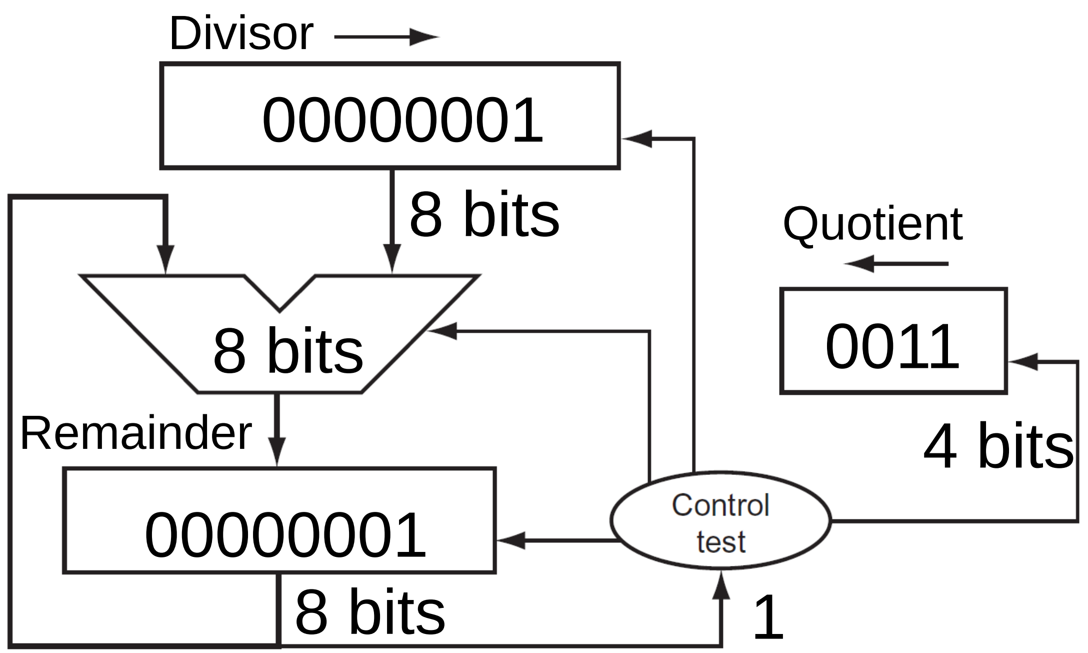
Division Example

- Divide 7_{ten} (0111_{two}) by 2_{ten} (0010_{two})



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

4-bit division needs 5 repetition



end of iteration 5

Done!

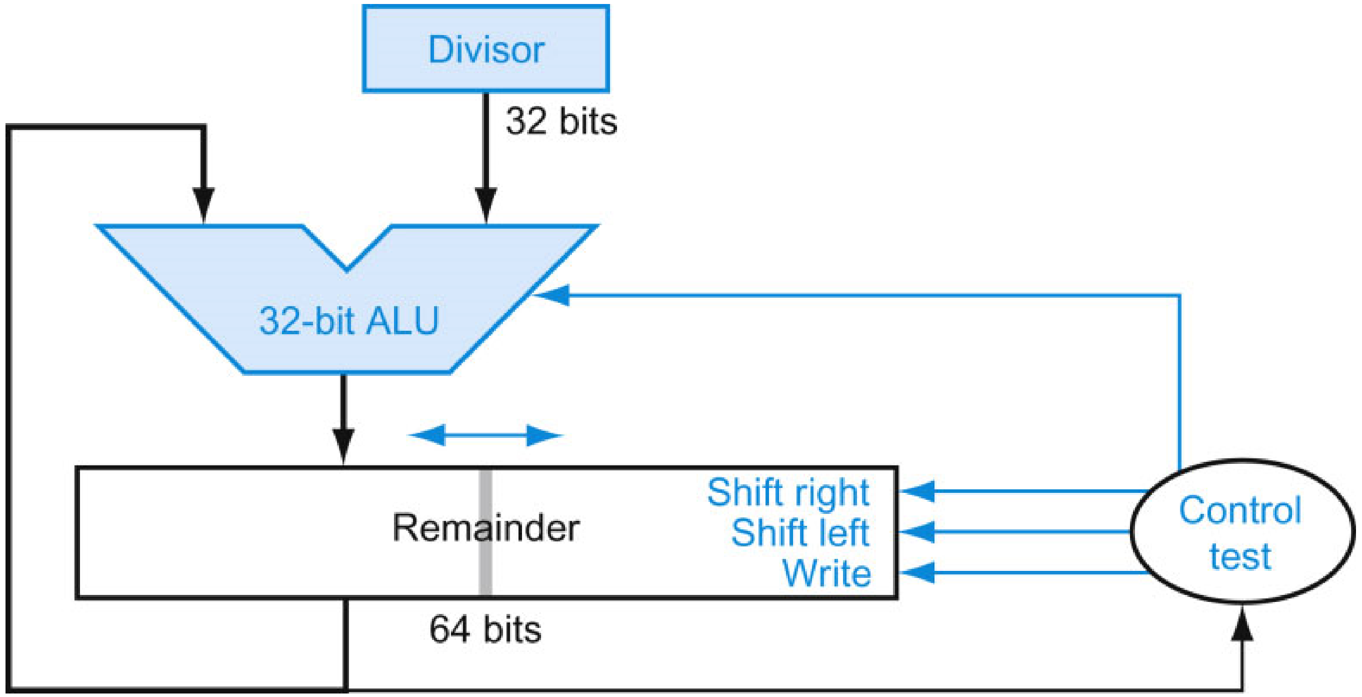
Division Example

- Divide 7_{ten} (0111_{two}) by 2_{ten} (0010_{two})

Iter	Step	Quot	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	Rem = Rem – Div Rem < 0 → +Div, shift 0 into Q Shift Div right	0000 0000 0000	0010 0000 0010 0000 0001 0000	1110 0111 0000 0111 0000 0111
2	Same steps as 1	0000 0000 0000	0001 0000 0001 0000 0000 1000	1111 0111 0000 0111 0000 0111
3	Same steps as 1	0000 0000 0000	0000 1000 0000 1000 0000 0100	1111 1111 0000 0111 00000111
4	Rem = Rem – Div Rem >= 0 → shift 1 into Q Shift Div right	0000 0001 0001	0000 0100 0000 0100 0000 0010	0000 0011 0000 0011 0000 0011
5	Same steps as 4	0001 0011 0011	0000 0010 0000 0010 0000 0001	0000 0001 0000 0001 0000 0001

Optimized Divider

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



Signed Division

- Convert to positive and adjust sign later
- Note that multiple solutions exist for the equation:

$$\boxed{\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}}$$

```
+7  div  +2         Quo = +3         Rem = +1
-7  div  +2         Quo = -3         Rem = -1
```

  - Why not -7 div +2 Quo = -4 Rem = +1?
  - If so, -(x div y) != (-x) div y => programming challenge!

- Convention:
  - Dividend and remainder have the same sign
  - Quotient is negative if signs disagree
  - These rules fulfill the equation above
  - Example:

```
+7  div  -2       Quo = -3       Rem = +1
-7  div  -2       Quo = +3       Rem = -1
```

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

RISC-V Division

- Four instructions:
  - div, rem: signed divide, remainder
  - divu, remu: unsigned divide, remainder

- Overflow and division-by-zero don’t produce errors
  - Just return defined results
  - Faster for the common case of no error