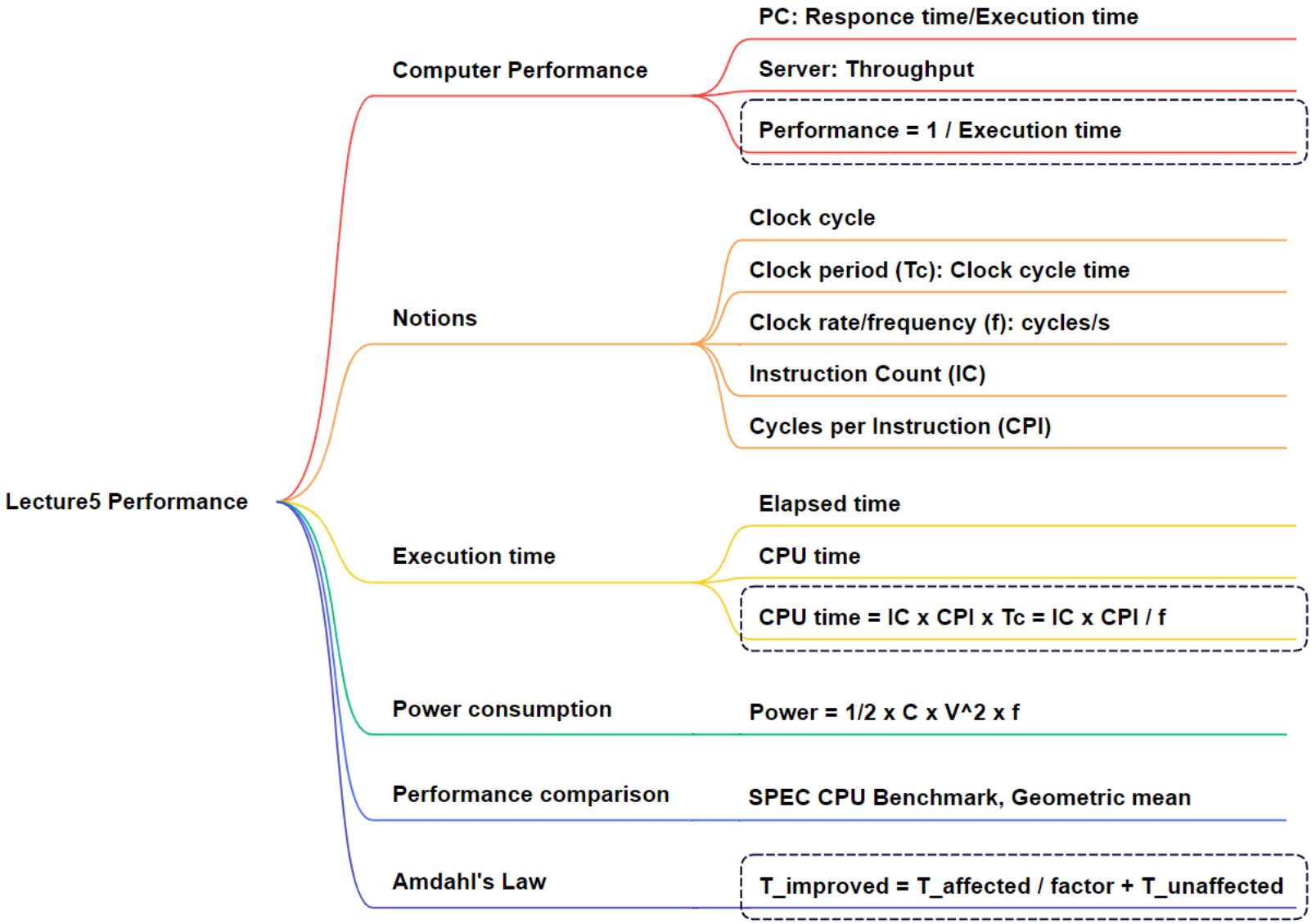
Lecture 6
Arithmetic for Computers (1)

CS202 2023 Spring

Today's Agenda

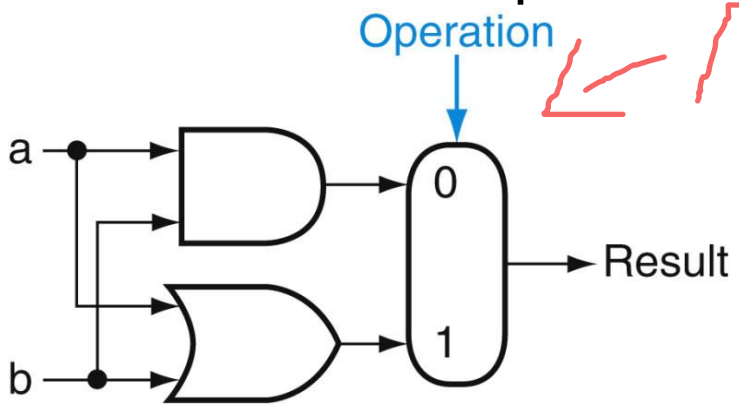
- Recap
- Context
  - Operations on integers
    - Addition and subtraction
    - ALU
    - Dealing with overflow
    - Multiplication and division
- Reading: Textbook 3.1-3.4

Recap

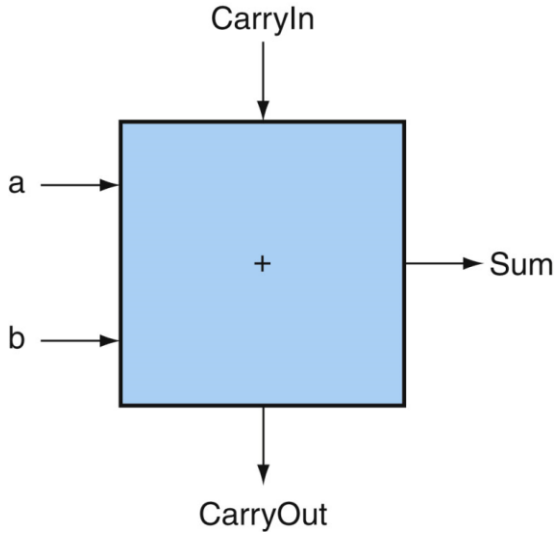

Lecture5 Performance
Computer Performance
PC: Responce time/Execution time
Server: Throughput
Performance = 1 / Execution time
Notions
Clock cycle
Clock period (Tc): Clock cycle time
Clock rate/frequency (f): cycles/s
Instruction Count (IC)
Cycles per Instruction (CPI)
Execution time
Elapsed time
CPU time
CPU time = IC x CPI x Tc = IC x CPI / f
Power consumption
Power = 1/2 x C x V^2 x f
Performance comparison
SPEC CPU Benchmark, Geometric mean
Amdahl's Law
T_improved = T_affected / factor + T_unaffected

Basic Arithmetic Logic Unit

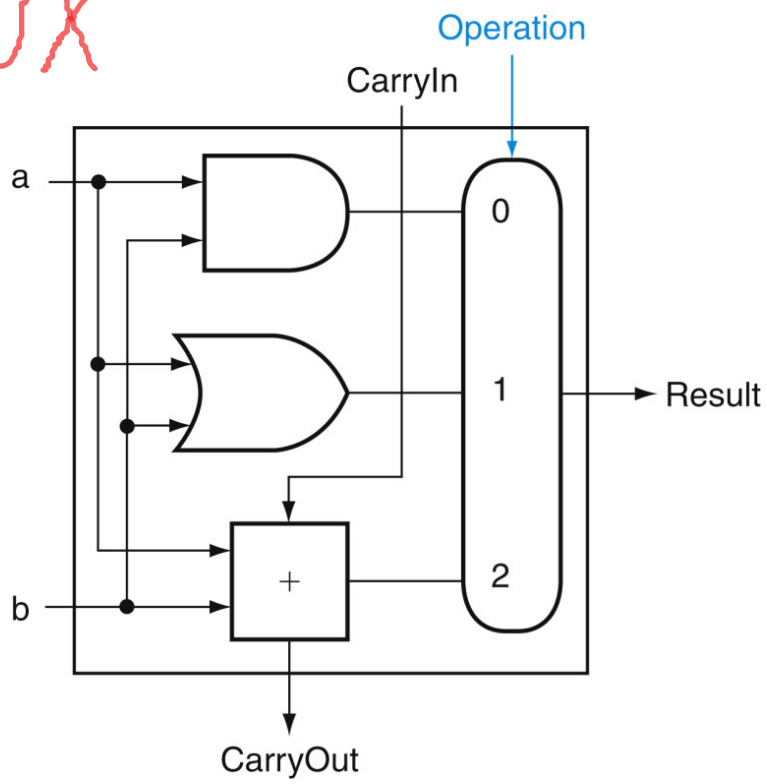
- One-bit ALU that performs AND, OR, and addition



1-bit logical unit for AND and OR



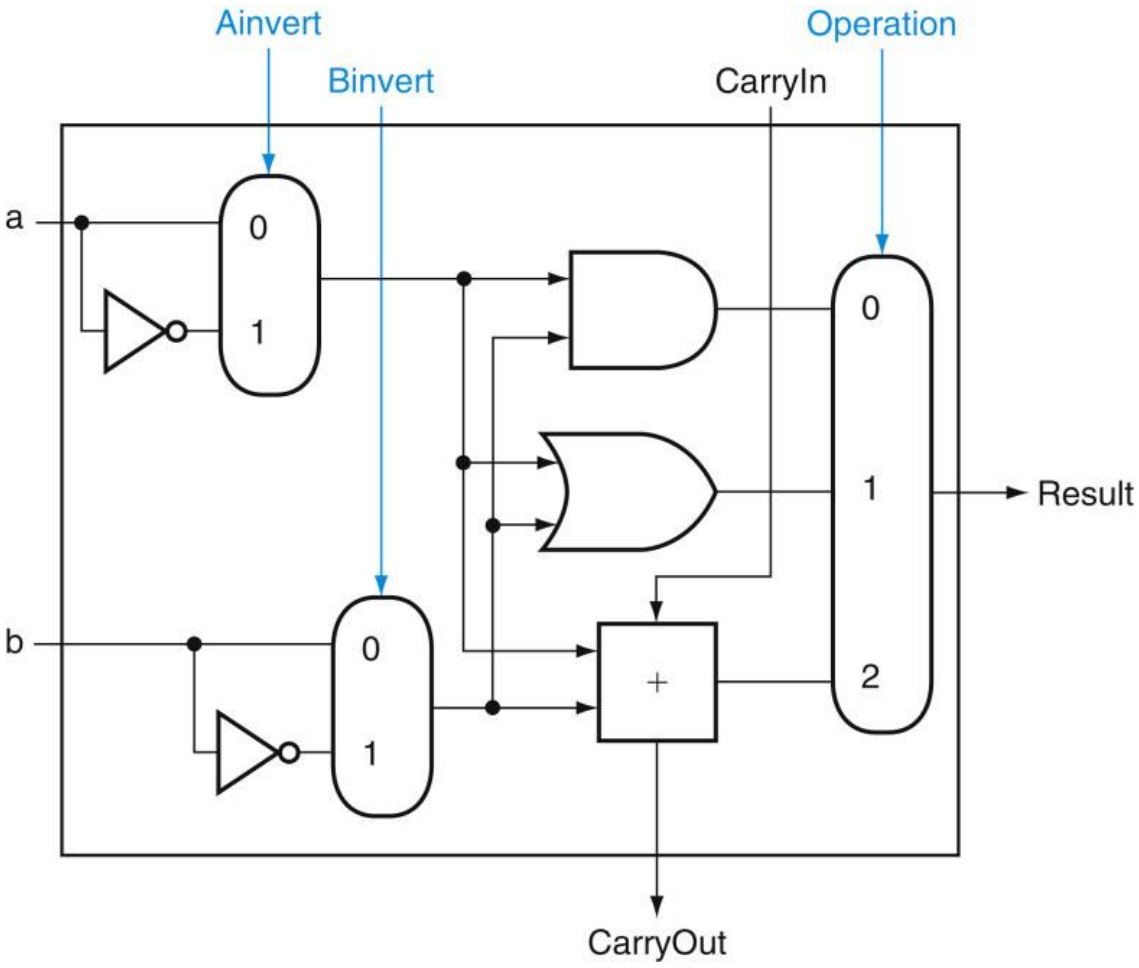
1-bit adder



op = 0, o = a & b (AND)
op = 1, o = a | b (OR)
op = 2, o = a + b (ADD)

Enhanced Arithmetic Logic Unit

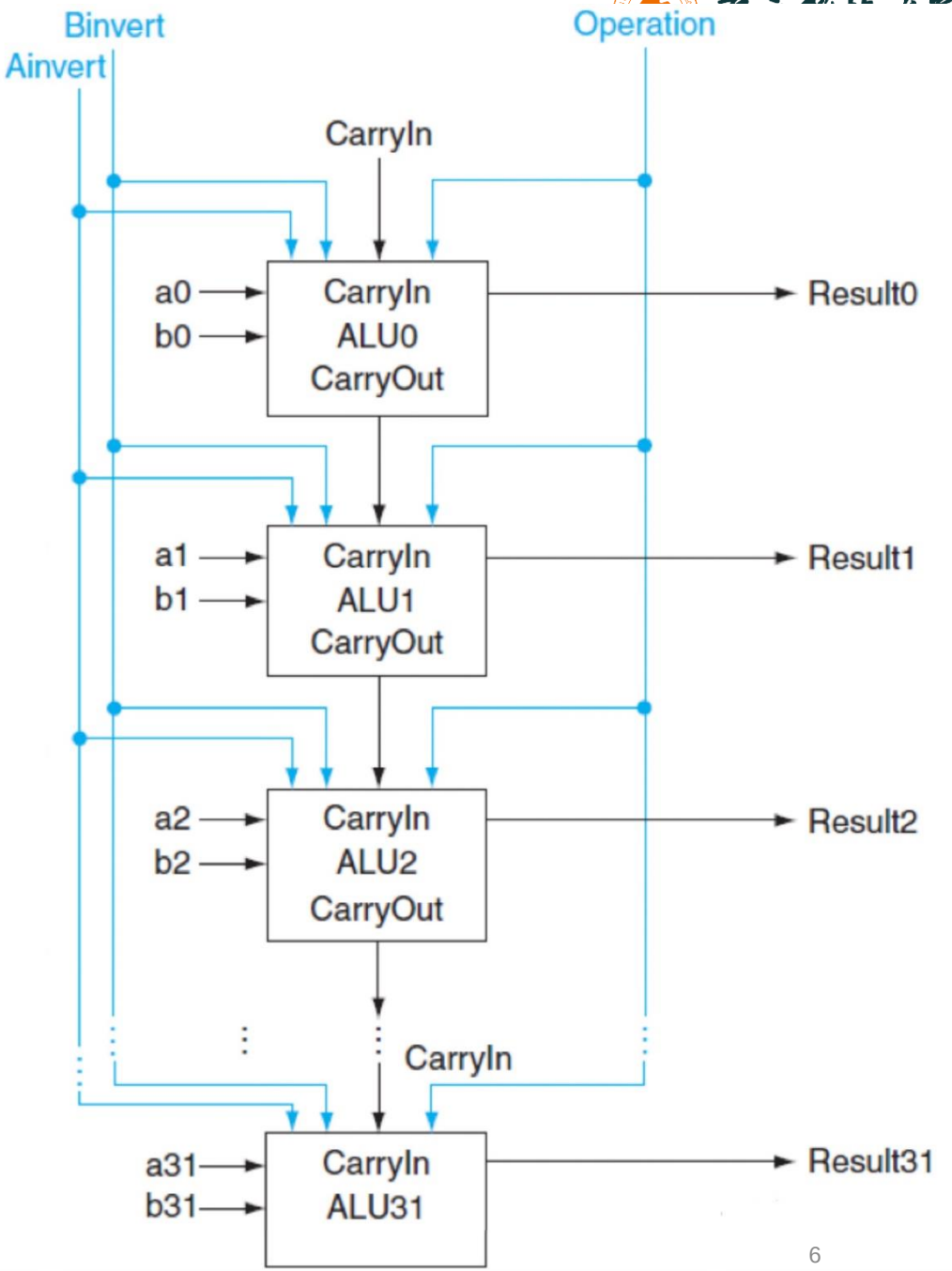
- ALU that have NAND/NOR operation, and subtraction



1-bit ALU that performs AND, OR, and addition on a and b or a' and b'

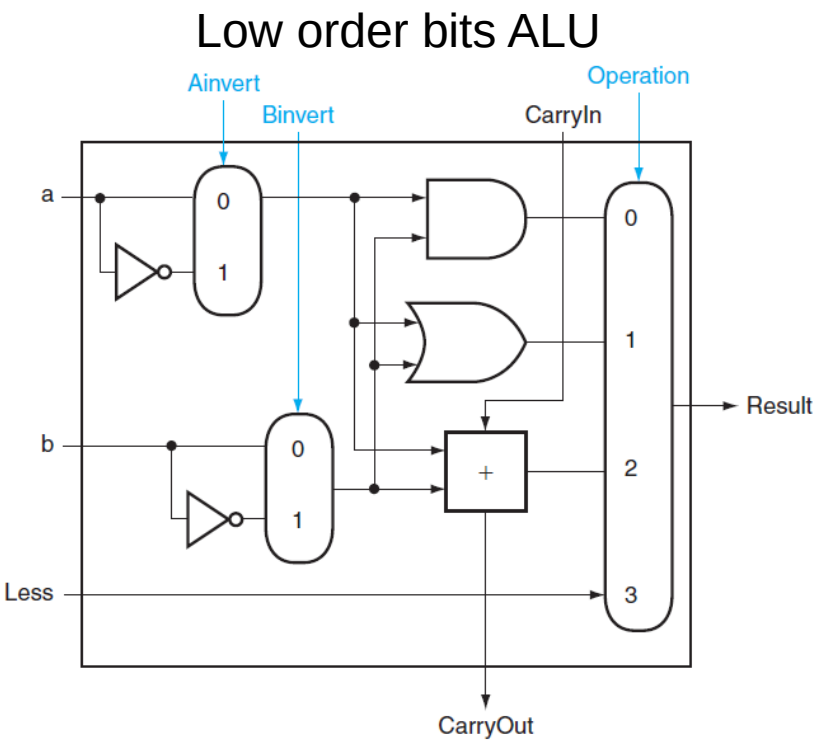
32-bit ALU

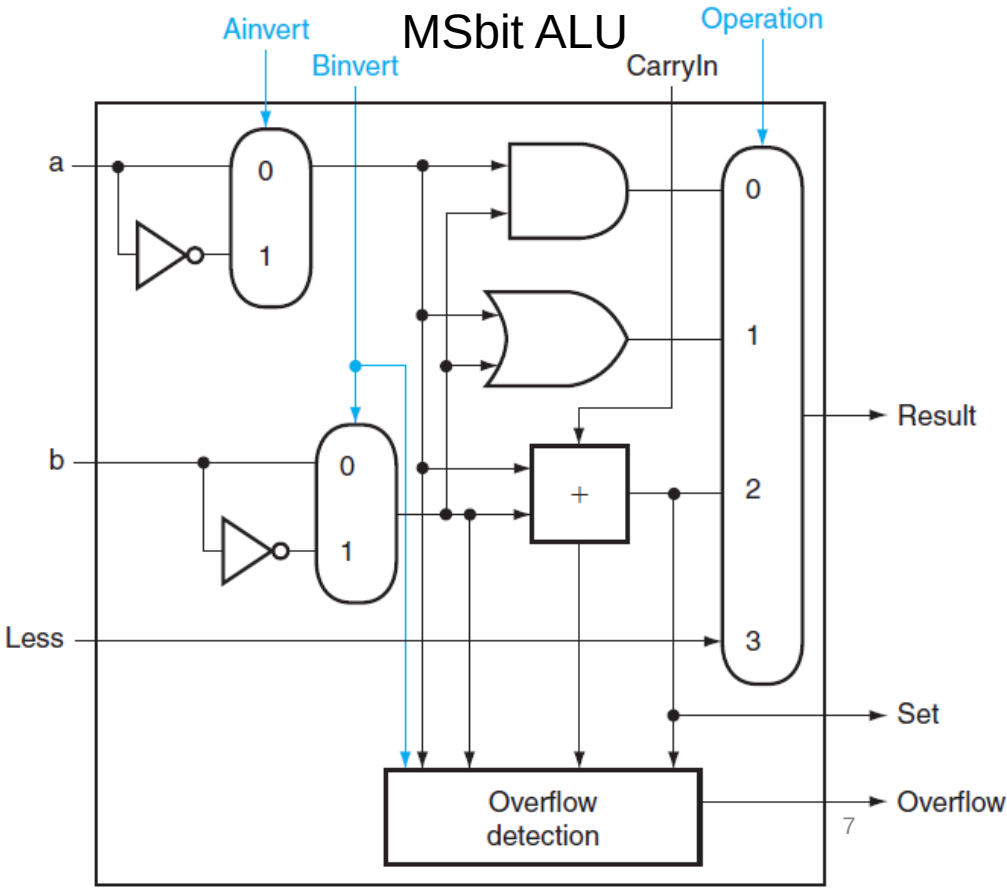
- Ripple-Carry-Adder
- CarryOut of the less significant bit is connected to the CarryIn of the more significant bit.



One-bit ALUs with Set Less Than

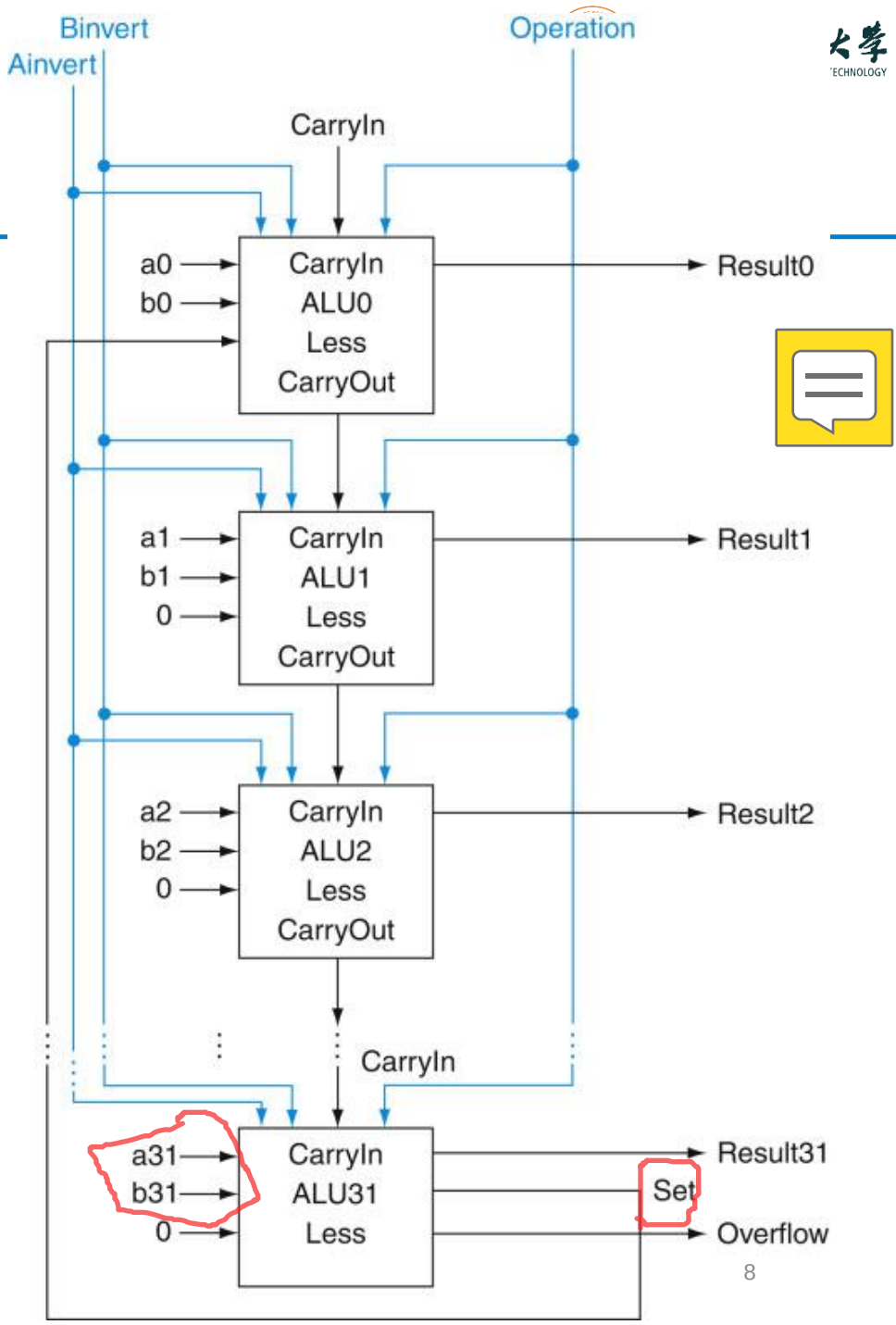
- Slt instruction requires a subtraction and then sets all but the LSbit to 0, with the LSbit set to 1 if a < b
- Less signal line
  - lsb – signed bit
  - 1 if a < b, 0 otherwise





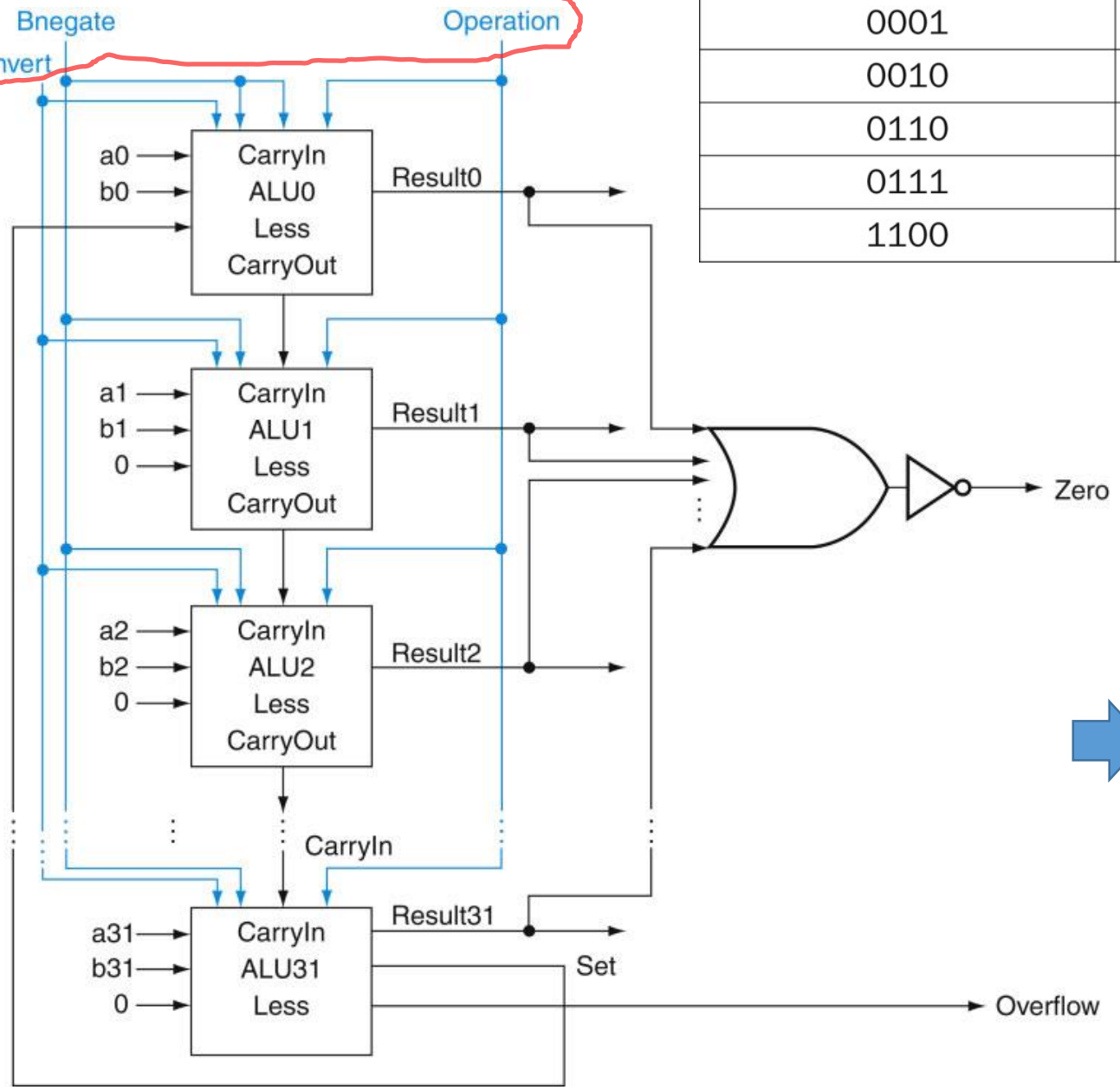
32-bit ALU with Set Less Than

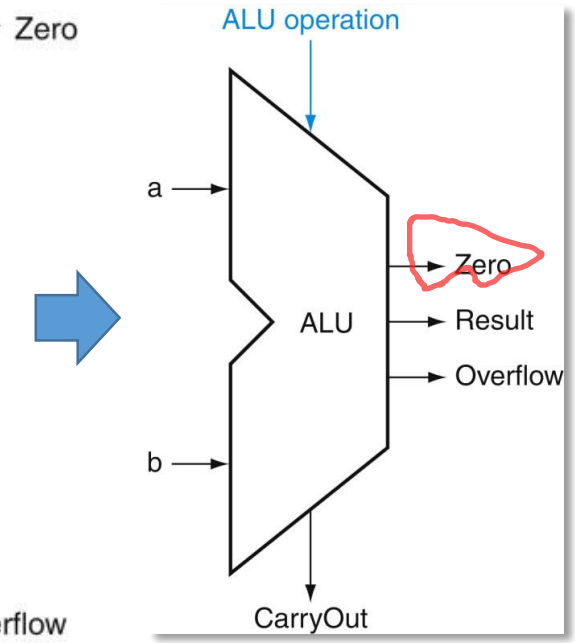
- The Less inputs are connected to 0 except for the least significant bit, which is connected to the Set output of the most significant bit.
- If the ALU performs a - b and we select the input 3 in the multiplexor
  - Result = 0 … 001 if a < b
  - Result = 0 … 000 otherwise



Final 32-bit ALU

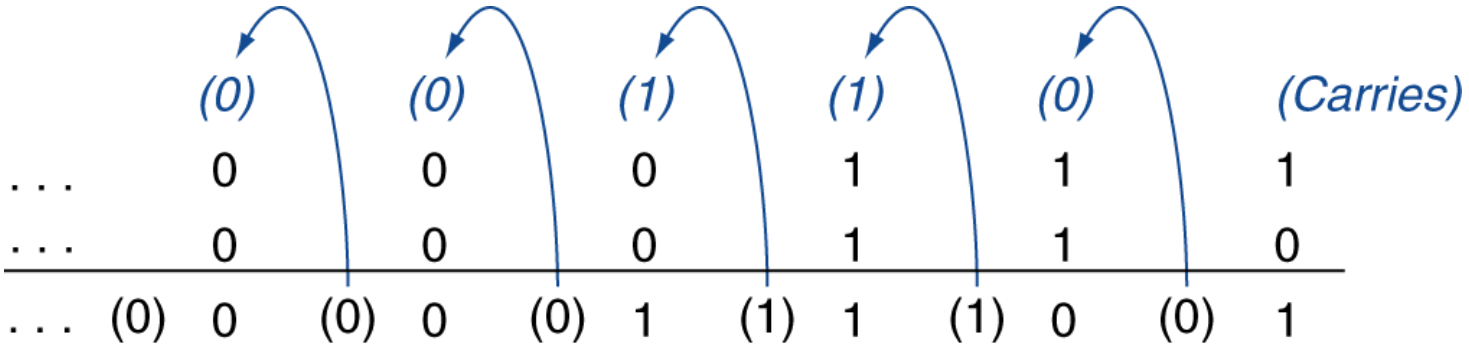
ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR





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

溢出：1.正数加正数是负数，比如111+111 结果是1000
2.负数+负数是正数

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 1
  - Subtracting –ve from +ve operand
    - Overflow if result sign is 0

Integer Addition/Subtraction with Overflow

- Example:
  - 12+3=15 (00001100+00000011)
  - 120+15=135 (01111000+00001111)
  - 12-3=9 (00001100-00000011)
  - -100-50=-150 (10011100-00110010)

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

Overflow

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

异常处理：先把pc寄存器里面的指令存到EPC里面，跳转到预定义的处理程序地址mfc0（从协处理器reg移动）指令可以检索EPC值，并在纠正操作后返回

Overflow Detection for Signed & Unsigned Addition

- Signed addition

```
addu $t0, $t1,   $t2
xor  $t3, $t1,   $t2
slt  $t3, $t3,   $zero
bne  $t3, $zero, No_overflow
xor  $t3, $t0,   $t1
slt  $t3, $t3,   $zero
bne  $t3, $zero, Overflow
```

```
# $t0 = sum
# Check if signs differ
# $t3 = 1 if signs differ
# $t1, $t2 signs ≠,no overflow
# $t1, $t2 signs =, check sum
# $t3 = 1 if sum sign≠
# All 3 signs ≠; goto overflow
```

- Unsigned addition

```
addu $t0, $t1,   $t2
nor  $t3, $t1,   $zero
sltu $t3, $t3,   $t2
bne  $t3, $zero, Overflow
```

```
# $t0 = sum
# $t3 = NOT $t1
# (2^32 – $t1 – 1) < $t2
# if(2^32–1<$t1+$t2), overflow
```

同号才会可能溢出，先判断是否同号，按位异或，如果符号位不同结果为1，为1的话t3就是负数，去和0比较，如果是负数，说明t1，t2同号，然后t0去和t1比较，如果同号说明没有溢出

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
    - c.f. 2s-complement modulo arithmetic
  - E.g., clipping in audio, saturation in video

Multiplication

- 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

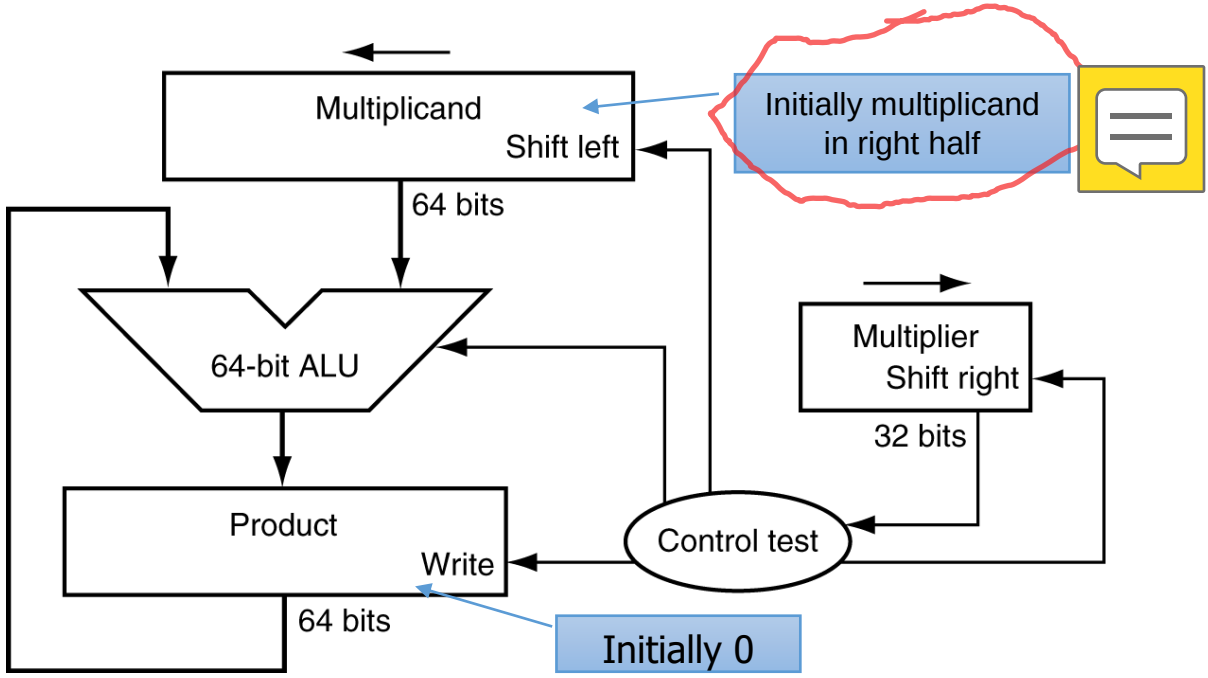
multipicand：被乘数
multiplier：乘数

multiplicand

multiplier

product

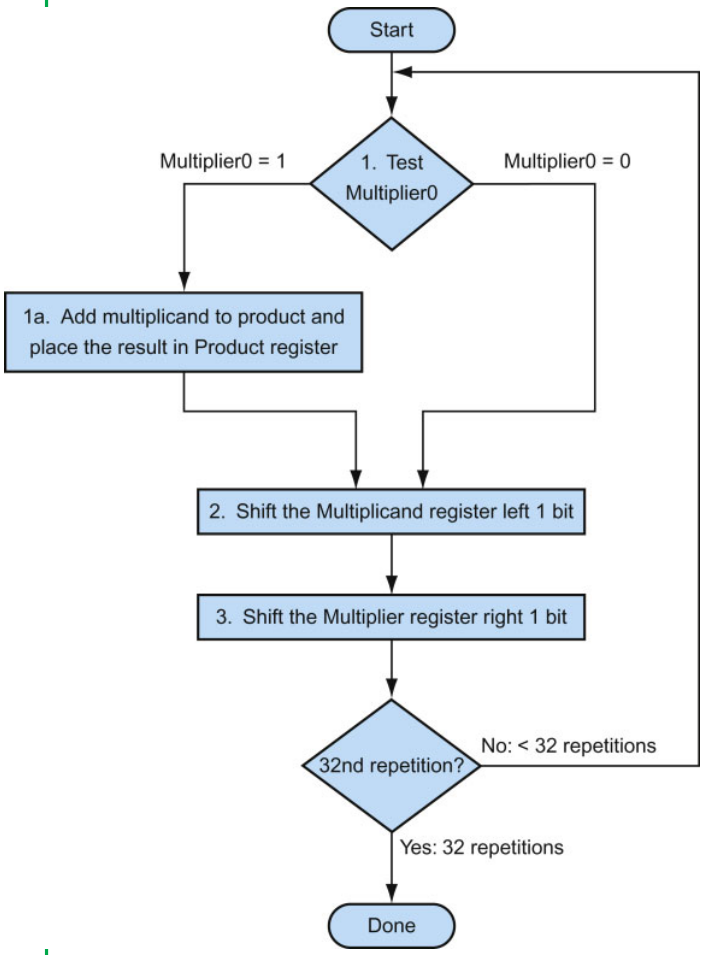
```
  1000
× 1001
------
  1000
 0000
0000
1000
------
1001000
```

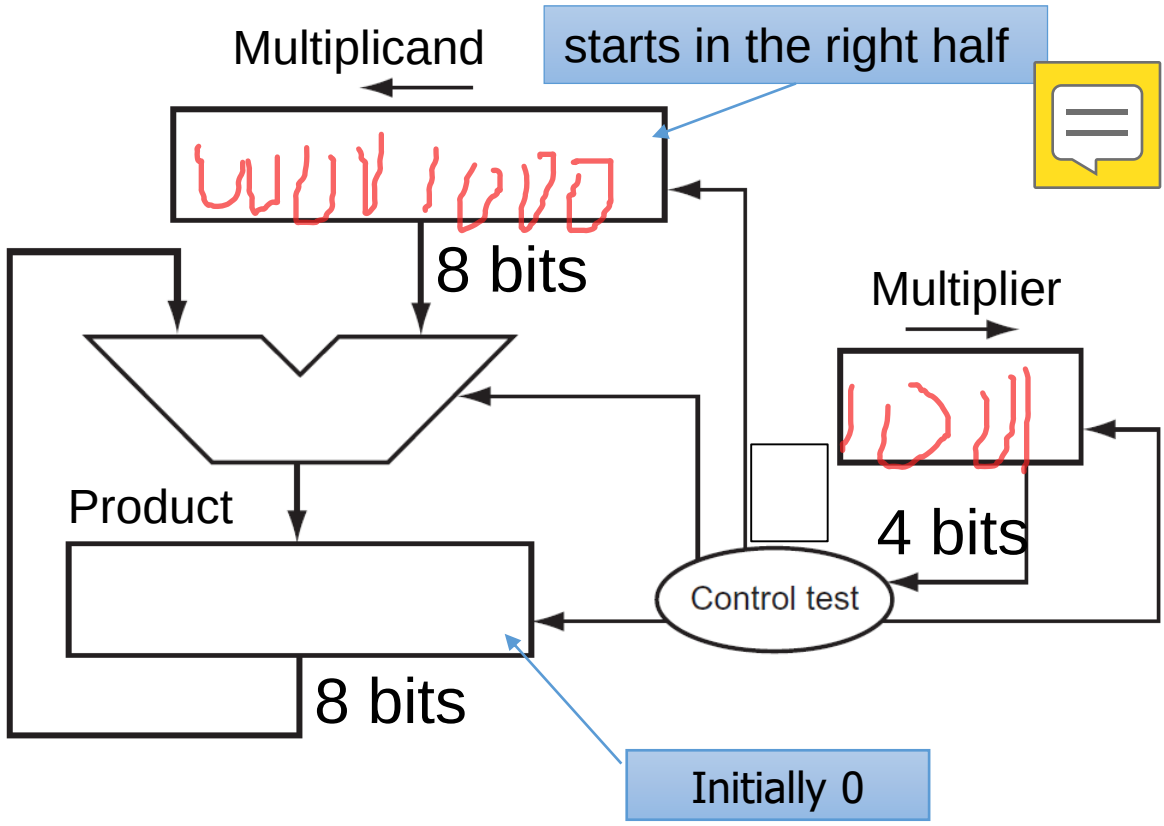


Length of product is the sum of operand lengths

Multiplication Example

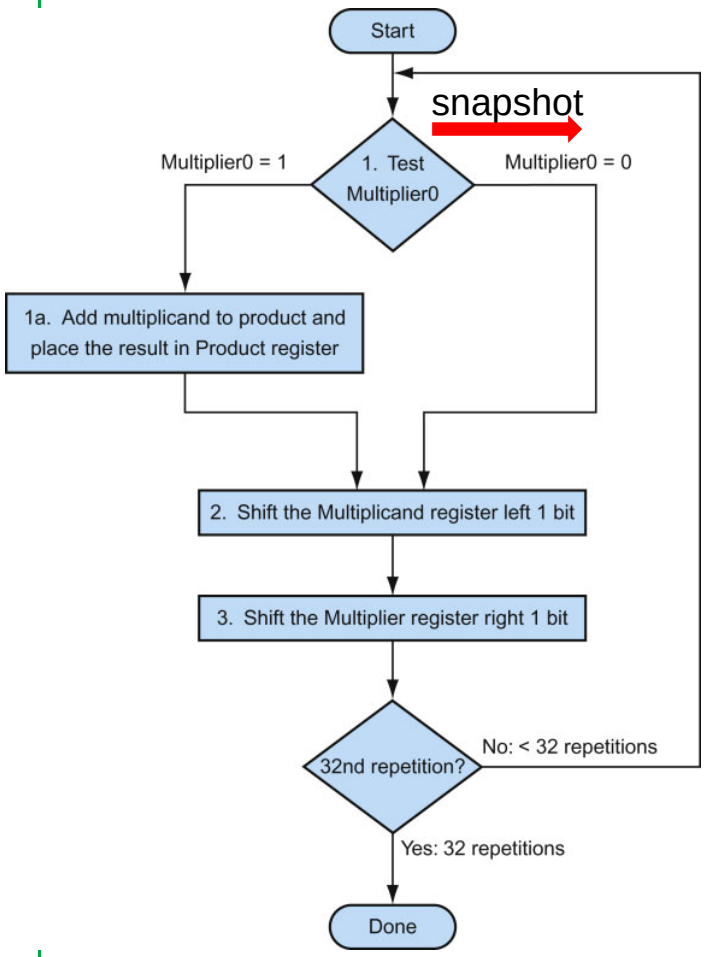
- Multiply 8_{ten} (1000_{two}) by 9_{ten} (1001_{two})
  - How values change in Mcand, Mplier and Product Registers?

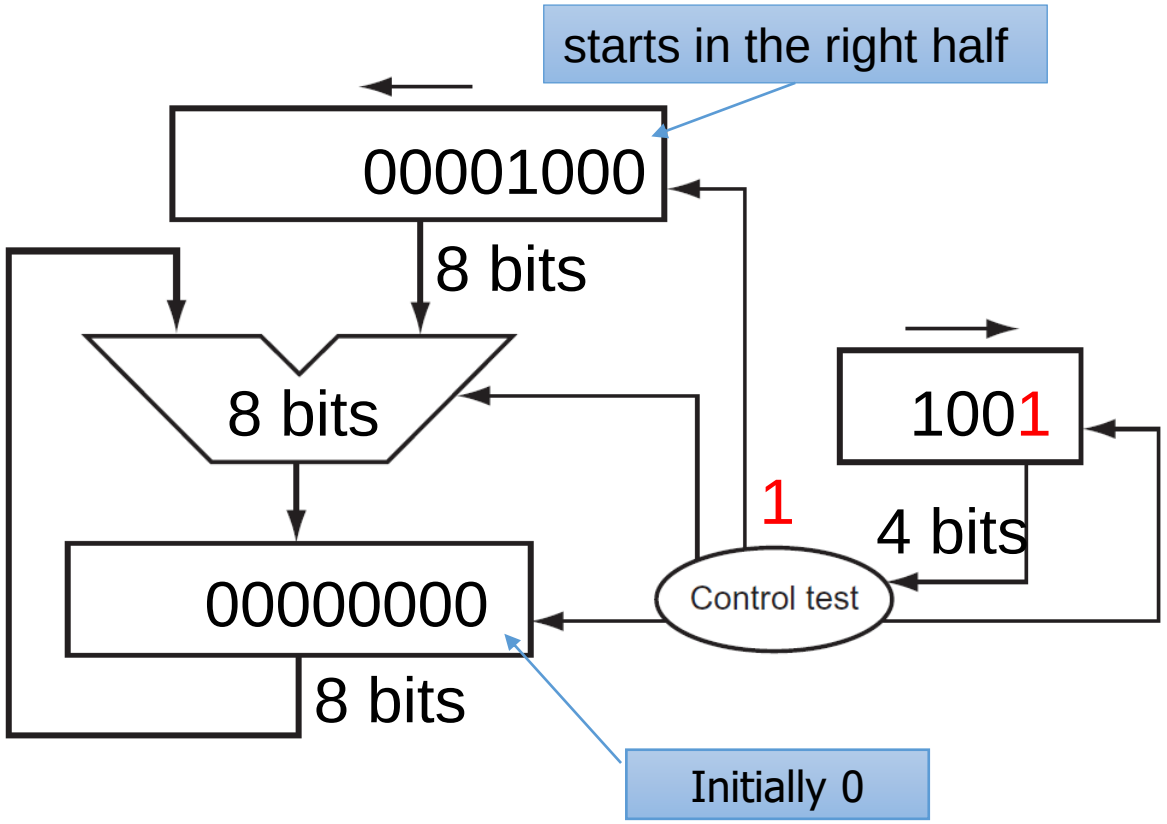




Multiplication Example

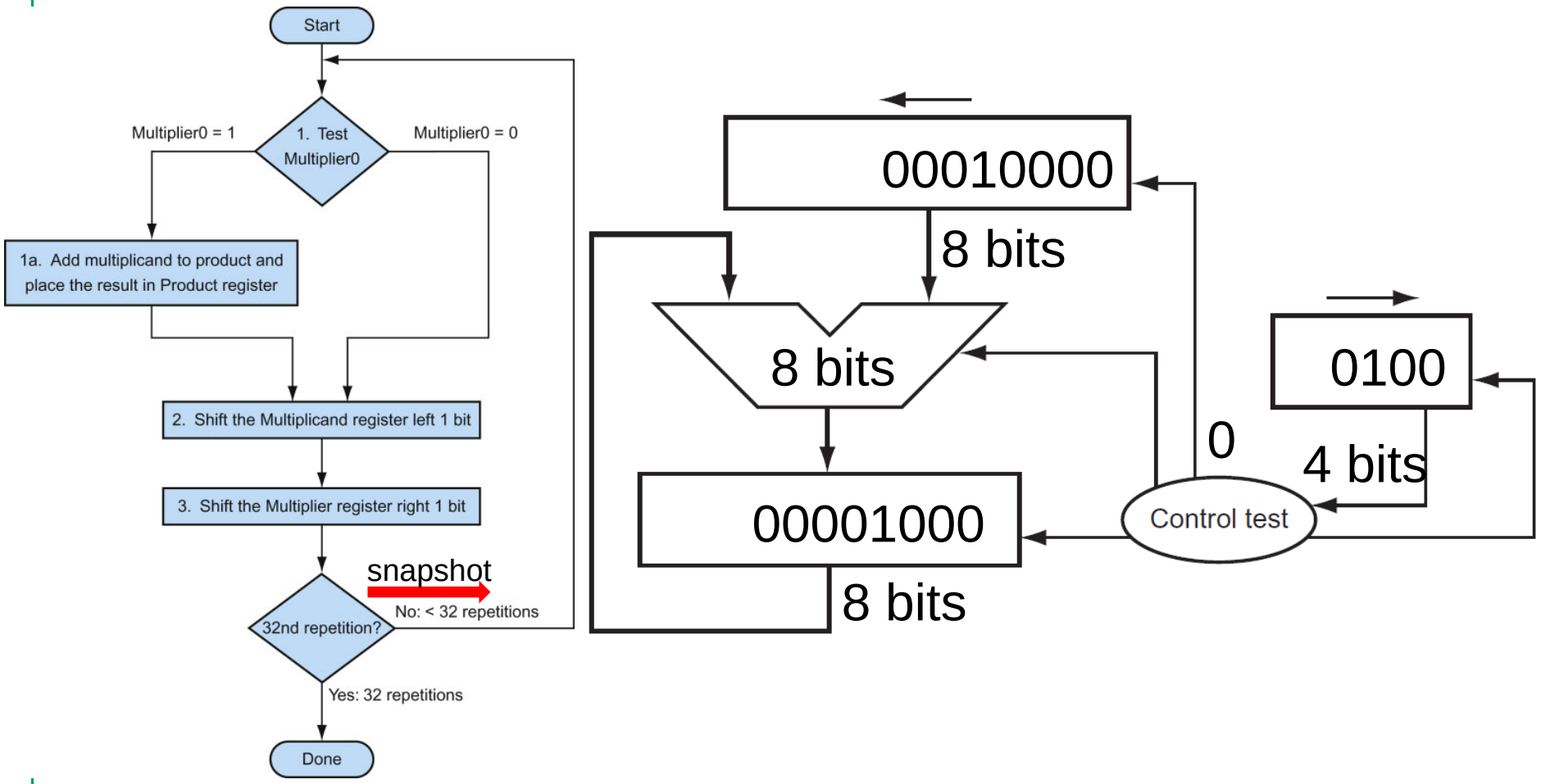
- Multiply 8_{ten} (1000_{two}) by 9_{ten} (1001_{two})





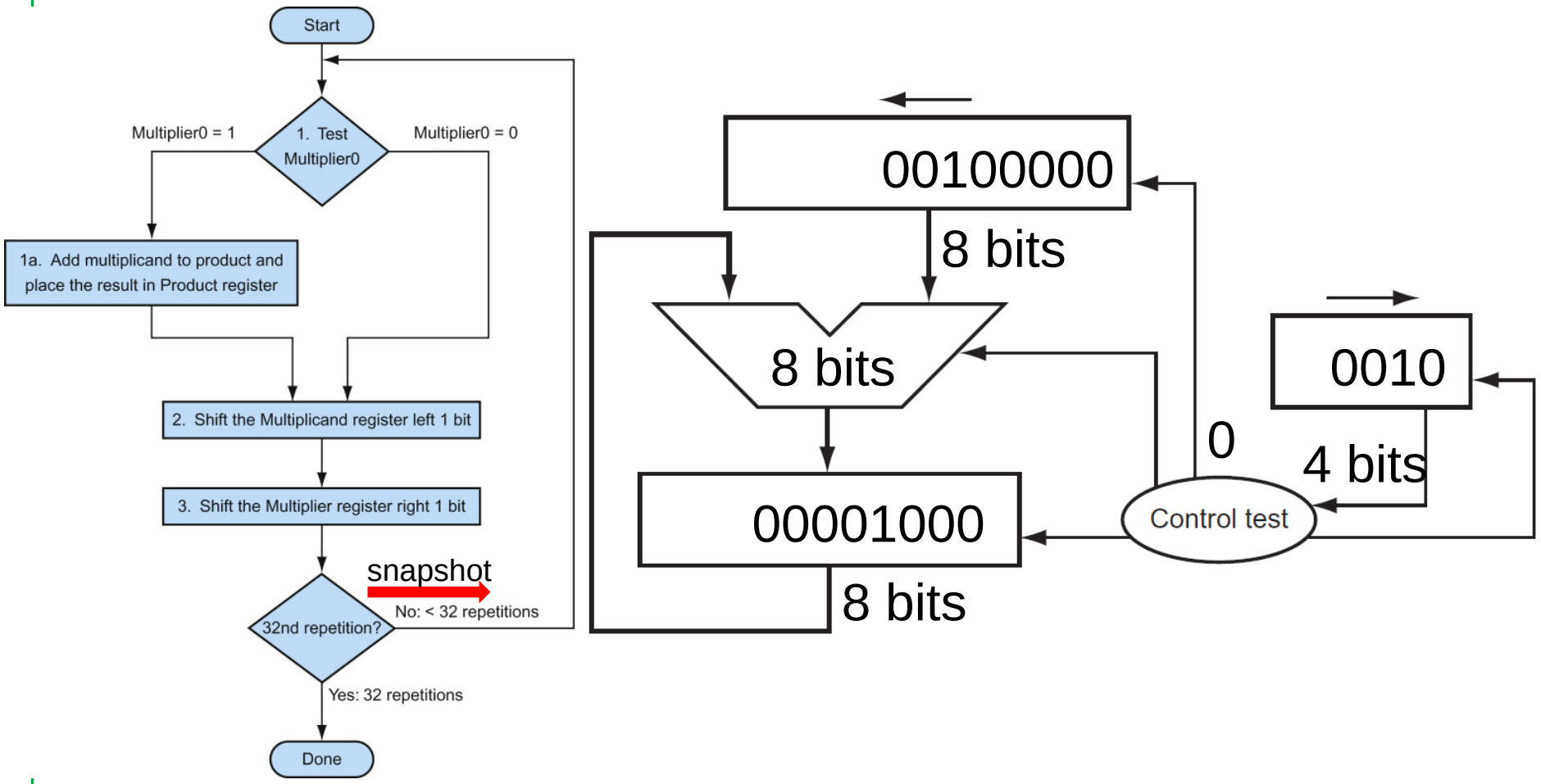
Multiplication Example

- Multiply 8_{ten} (1000_{two}) by 9_{ten} (1001_{two})



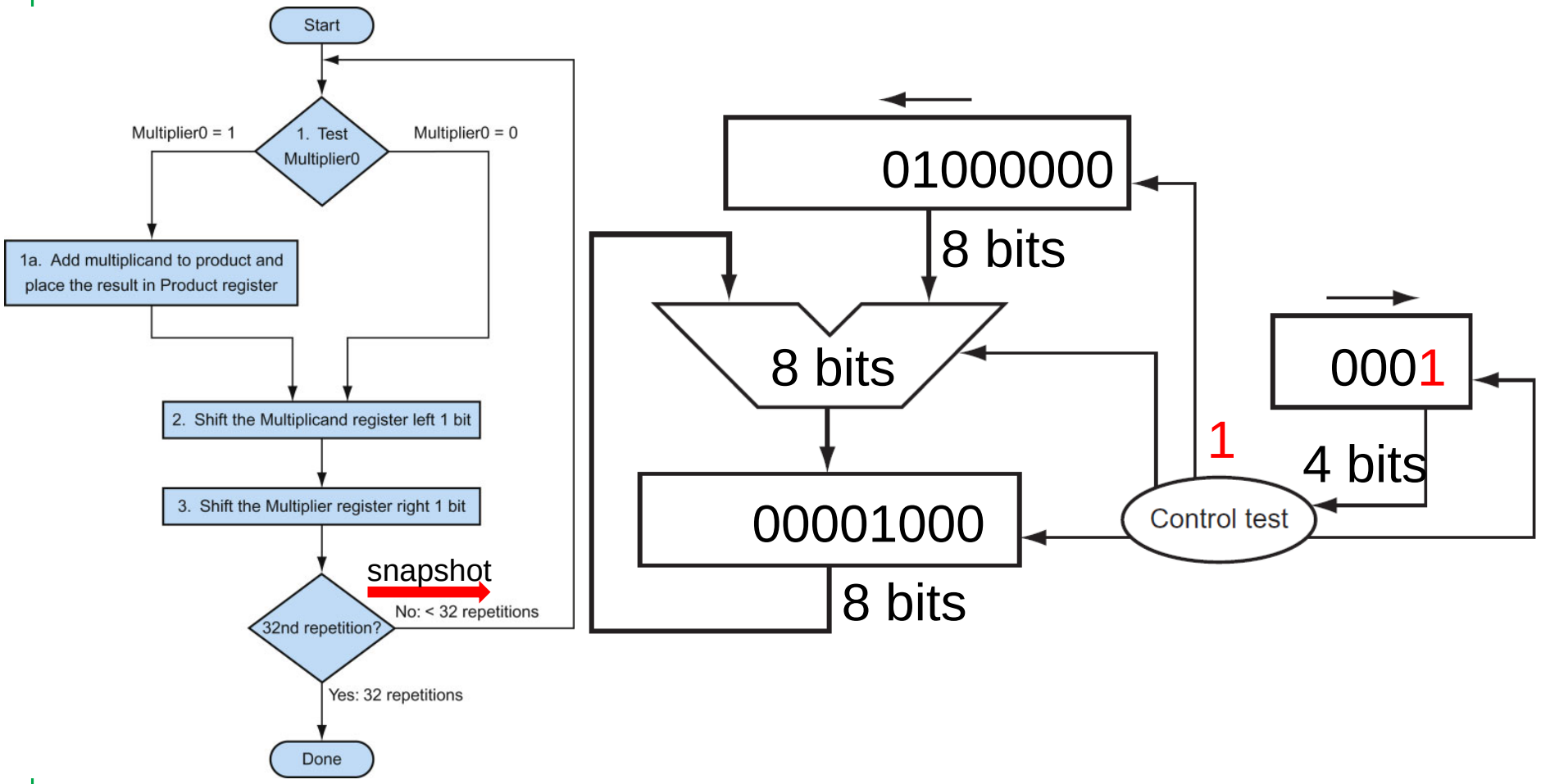
Multiplication Example

- Multiply 8_{ten} (1000_{two}) by 9_{ten} (1001_{two})



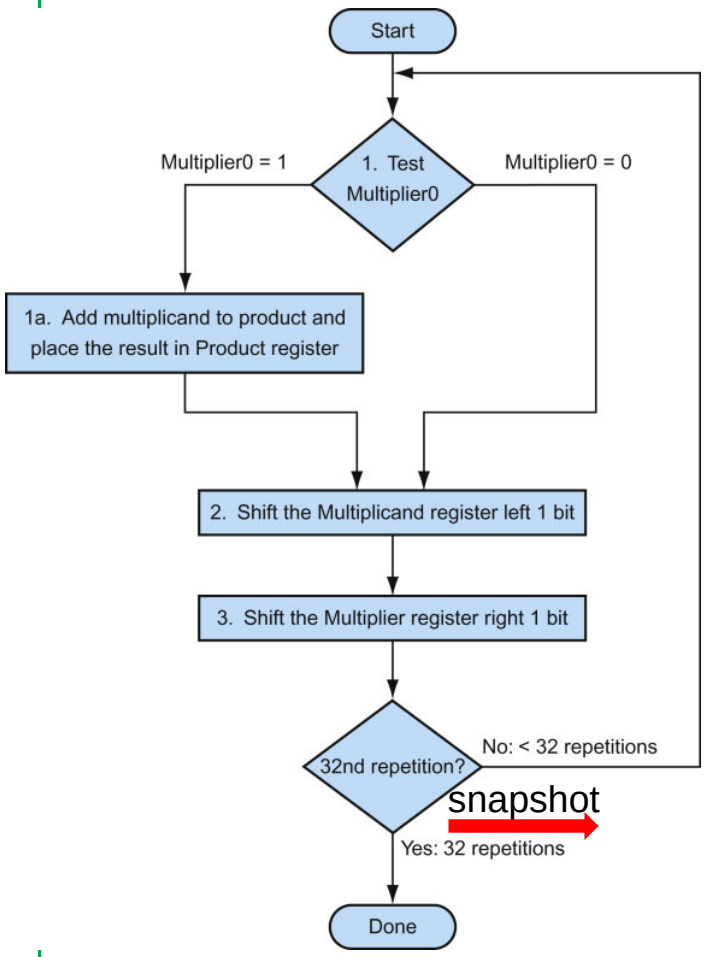
Multiplication Example

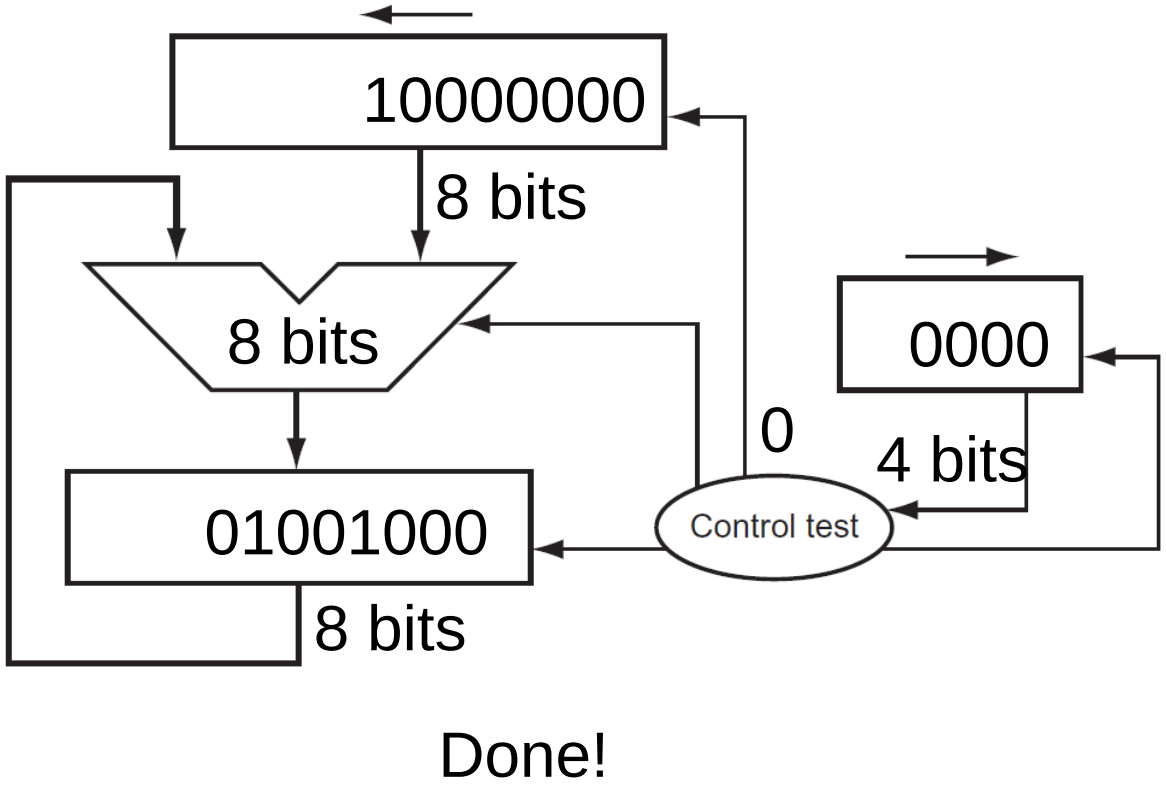
- Multiply 8_{ten} (1000_{two}) by 9_{ten} (1001_{two})



Multiplication Example

- Multiply 8_{ten} (1000_{two}) by 9_{ten} (1001_{two})





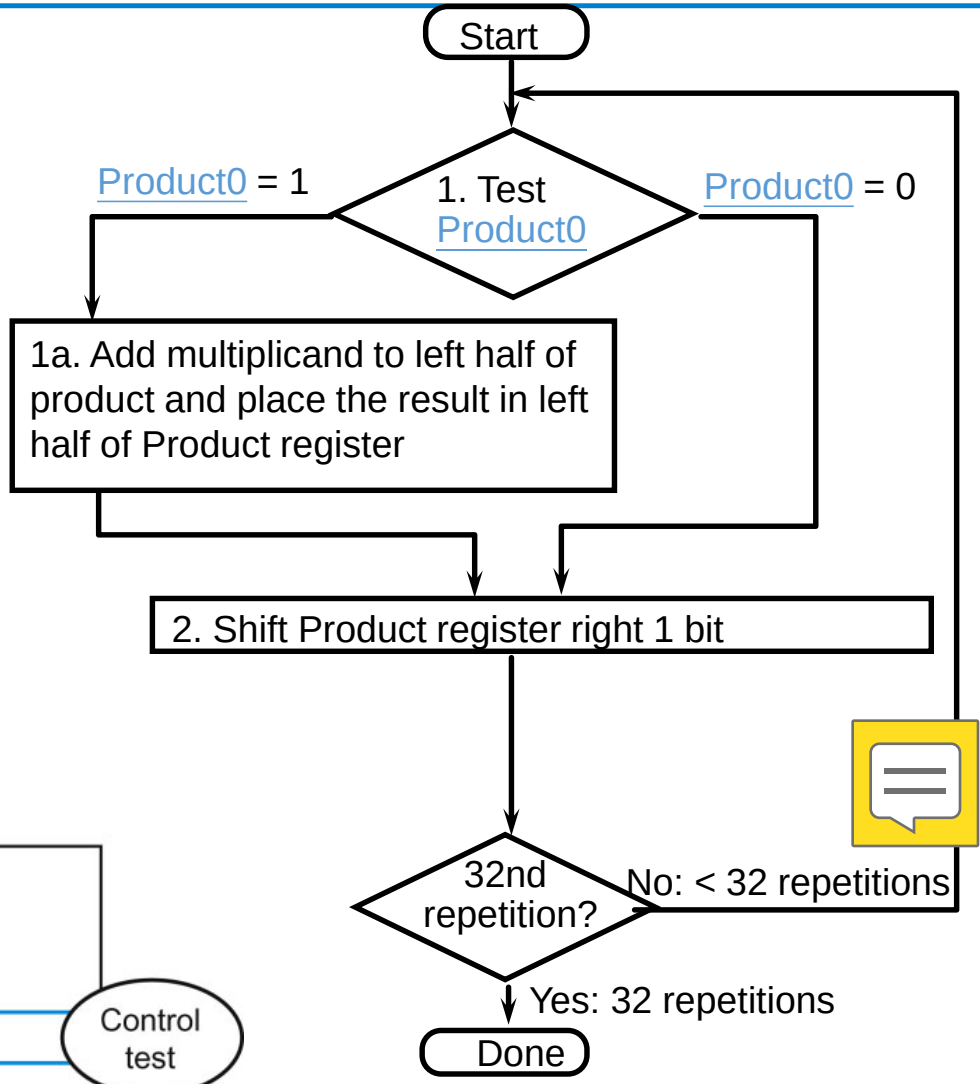
Multiplication Example

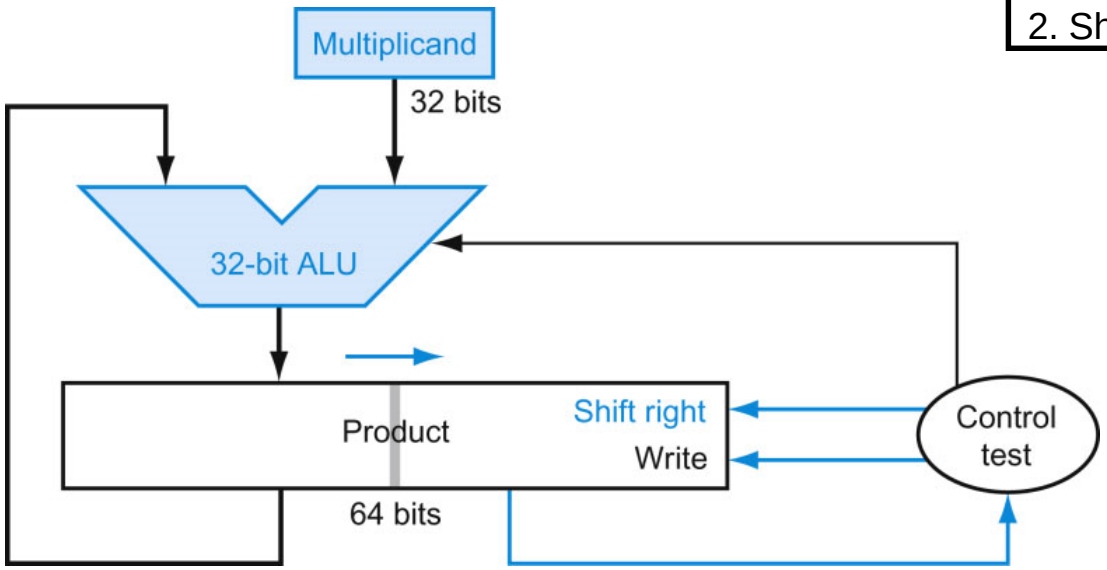
- Multiply 8_{ten} (1000_{two}) by 9_{ten} (1001_{two})

Iter	Step	Multiplier	Multiplicand	Product
0	Initial values	1001	0000 1000	0000 0000
1	1 ⇒ Prod = Prod + Mcand	1001	0000 1000	0000 1000
	Shift left Multiplicand	1001	0001 0000	0000 1000
	Shift right Multiplier	0100	0001 0000	0000 1000
2	0 ⇒ No operation	0100	0001 0000	0000 1000
	Shift left Multiplicand	0100	0010 0000	0000 1000
	Shift right Multiplier	0010	0010 0000	0000 1000
3	Same steps as 2	0010	0010 0000	0000 1000
		0010	0100 0000	0000 1000
		0001	0100 0000	0000 1000
4	Same steps as 1	0001	0100 0000	0100 1000
		0001	1000 0000	0100 1000
		0000	1000 0000	0100 1000

Optimized Multiplier Hardware

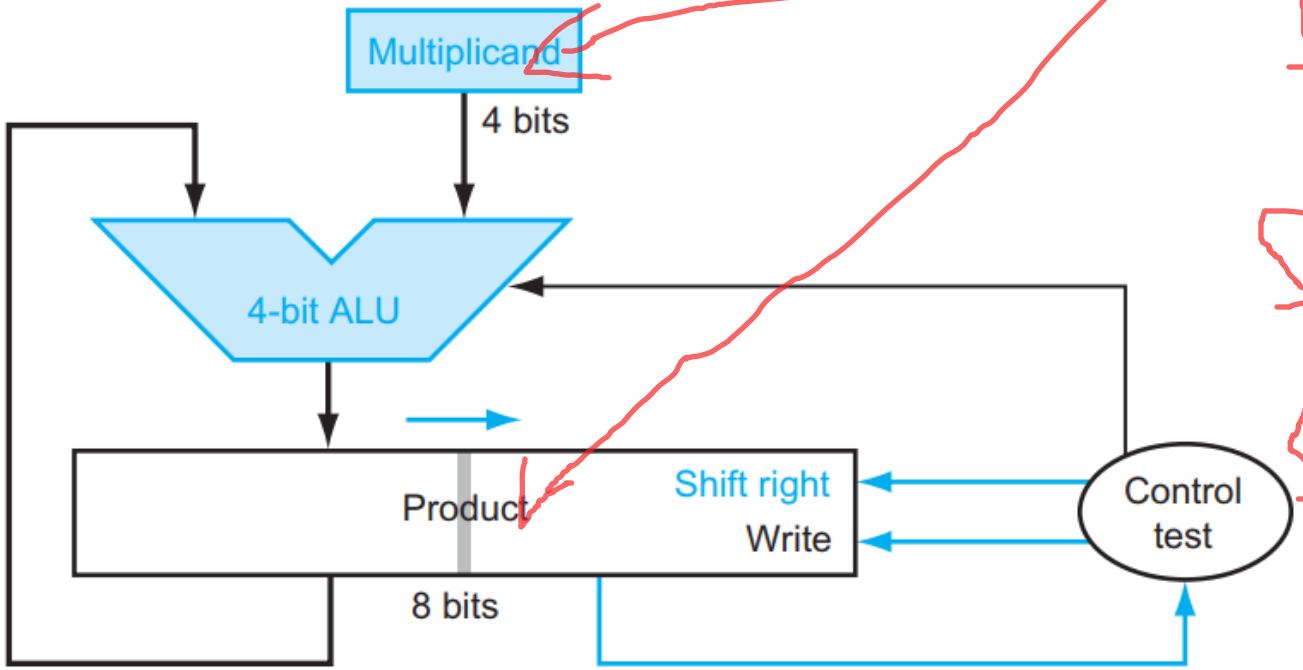
- Multiplier initially in right half of product register, 32-bit ALU and multiplicand is untouched
- Check the 0th bit in Product register, if 1, add left half of product with multiplicand
- The sum keeps shifting right, at every step, number of bits in product + multiplier = 64





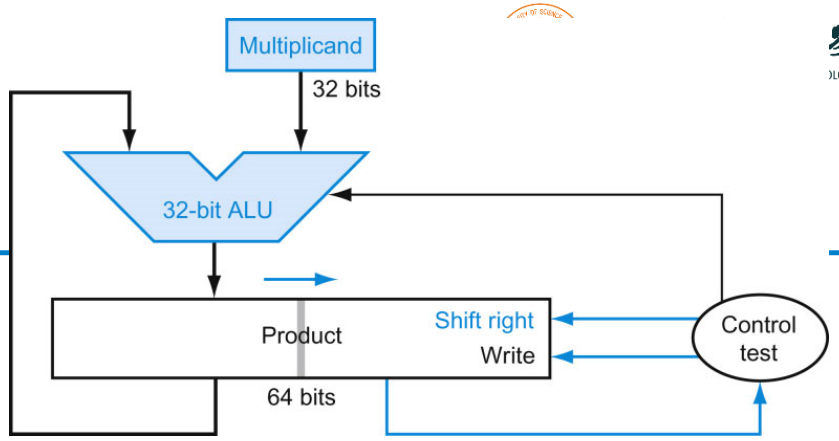
Optimized Multiplier Example

- Multiply 8_{ten} (1000_{two}) by 11_{ten} (1011_{two}) :
- Values in Product register?
- How about Multiplicand register?



Upper half	Lower half	
0000	**1011**	
1000		Add
1000	**1011**	Shift
0100	0**101**	
1000		Add
1100	0**101**	Shift
0110	00**10**	
0110	00**10**	Shift
0011	000**1**	
1000		Add
1011	000**1**	Shift
0101	1000	Done

Optimized Multiplier Example



- Multiply 8_{ten} by 9_{ten} :
- Value in Product Register?
- How about Multiplicand?

0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1001

1 0000 0000 0000 0000 0000 0000 0000 0100 0000 0000 0000 0000 0000 0000 0000 0100

2 0000 0000 0000 0000 0000 0000 0000 0010 0000 0000 0000 0000 0000 0000 0000 0010

3 0000 0000 0000 0000 0000 0000 0000 0001 0000 0000 0000 0000 0000 0000 0000 0001

4 0000 0000 0000 0000 0000 0000 0000 0100 1000 0000 0000 0000 0000 0000 0000 0000

5 0000 0000 0000 0000 0000 0000 0000 0010 0100 0000 0000 0000 0000 0000 0000 0000

6 0000 0000 0000 0000 0000 0000 0000 0001 0010 0000 0000 0000 0000 0000 0000 0000

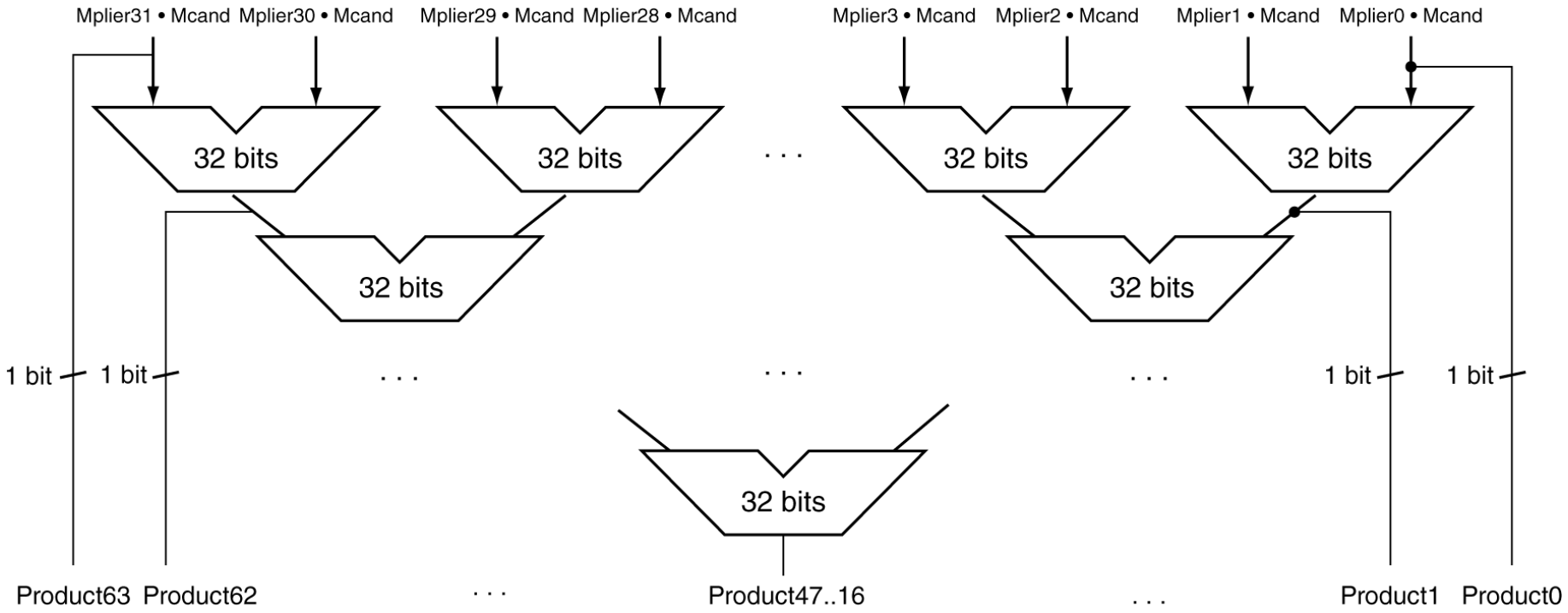
...

31 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1001 0000

32 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0100 1000

Faster Multiplier

- Uses multiple adders
  - Cost/performance tradeoff
  - Can be pipelined
  - Several multiplication performed in parallel



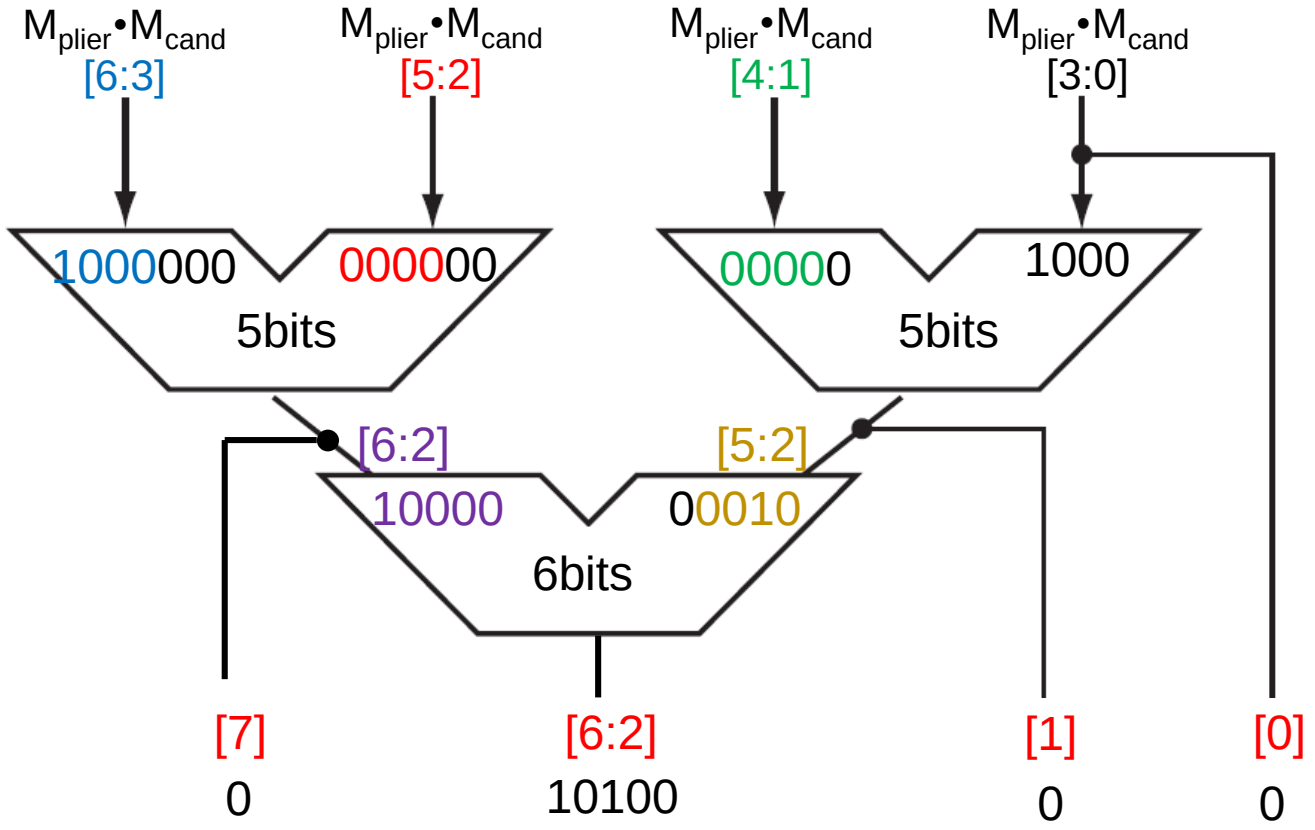
Faster Multiplier

- 4-bits example

bit order 76543210

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

bit order 76543210



final result: 01001000

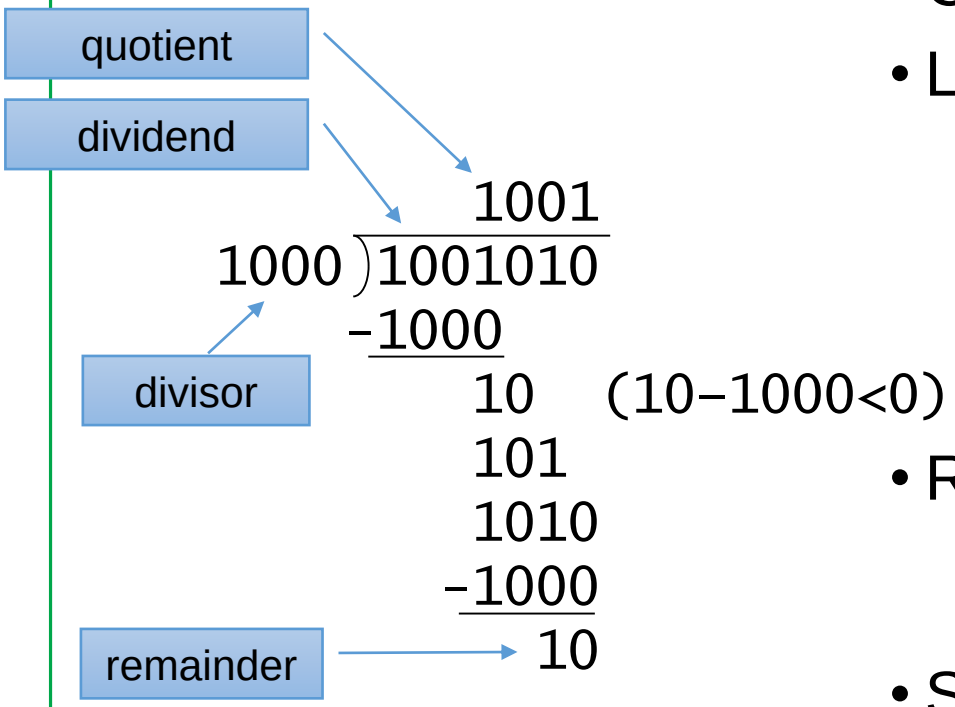
Notes

- The previous algorithm also works for signed numbers (negative numbers in 2's complement form)
- We can also convert negative numbers to positive, multiply the magnitudes, and convert to negative if signs disagree
- The product of two 32-bit numbers can be a 64-bit number
  - hence, in MIPS, the product is saved in two 32-bit registers

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



- 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

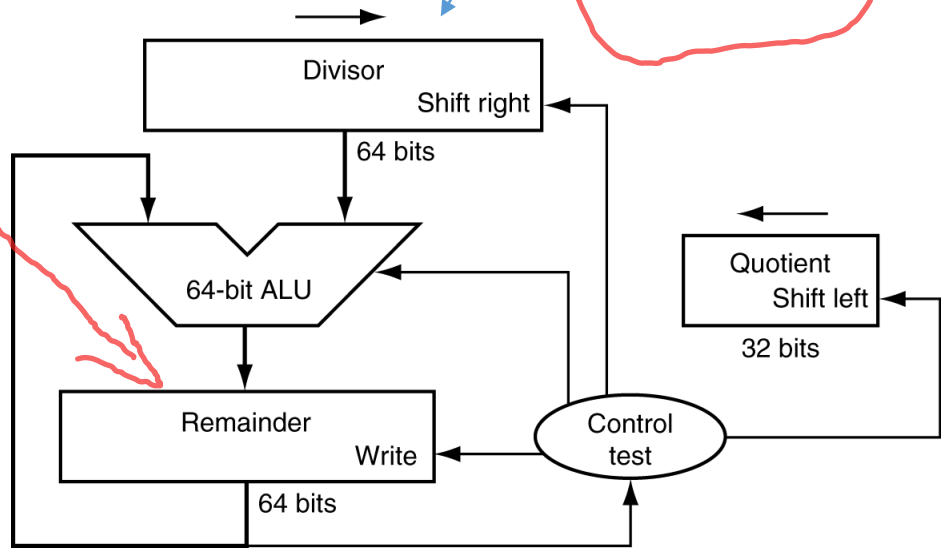
quotient

dividend

divisor

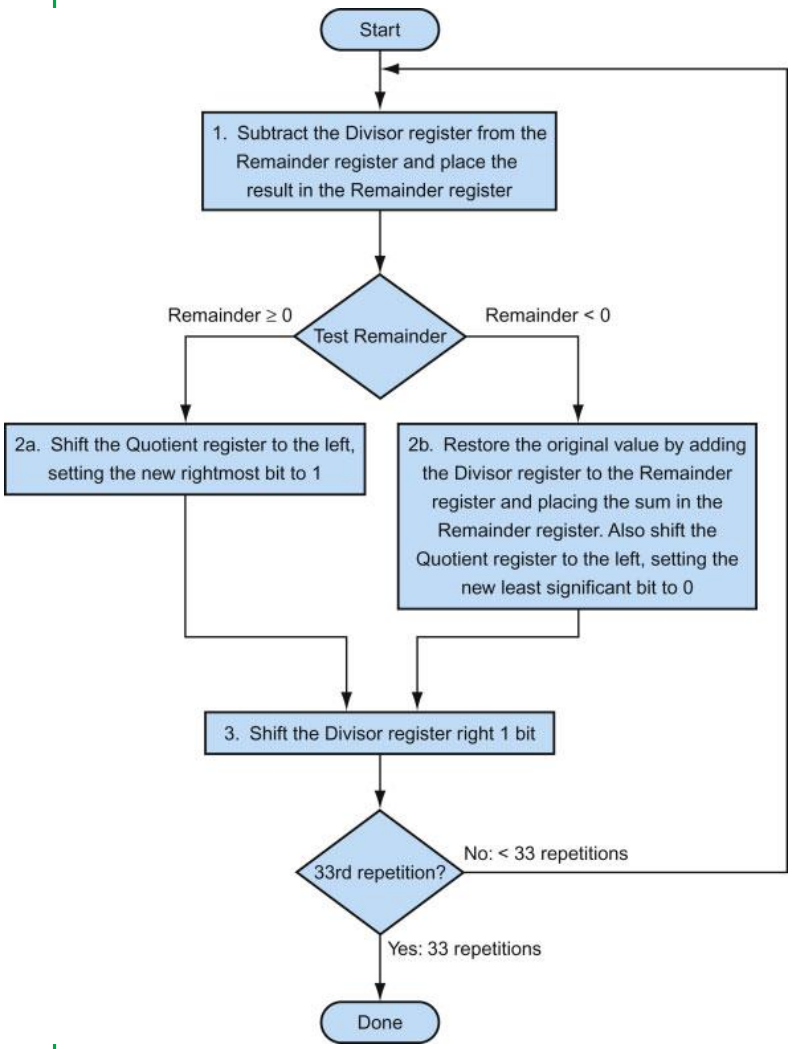
remainder

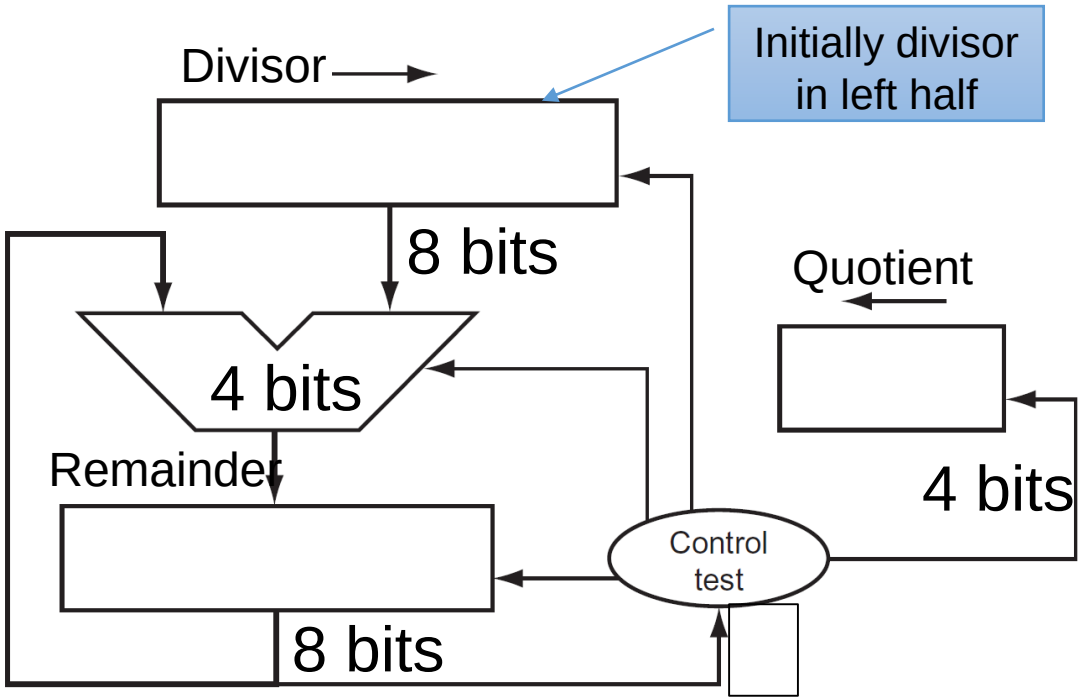
```
          1001
1000 ) 1001010
       -1000
          10
          101
          1010
        -1000
          10
```



Division Example

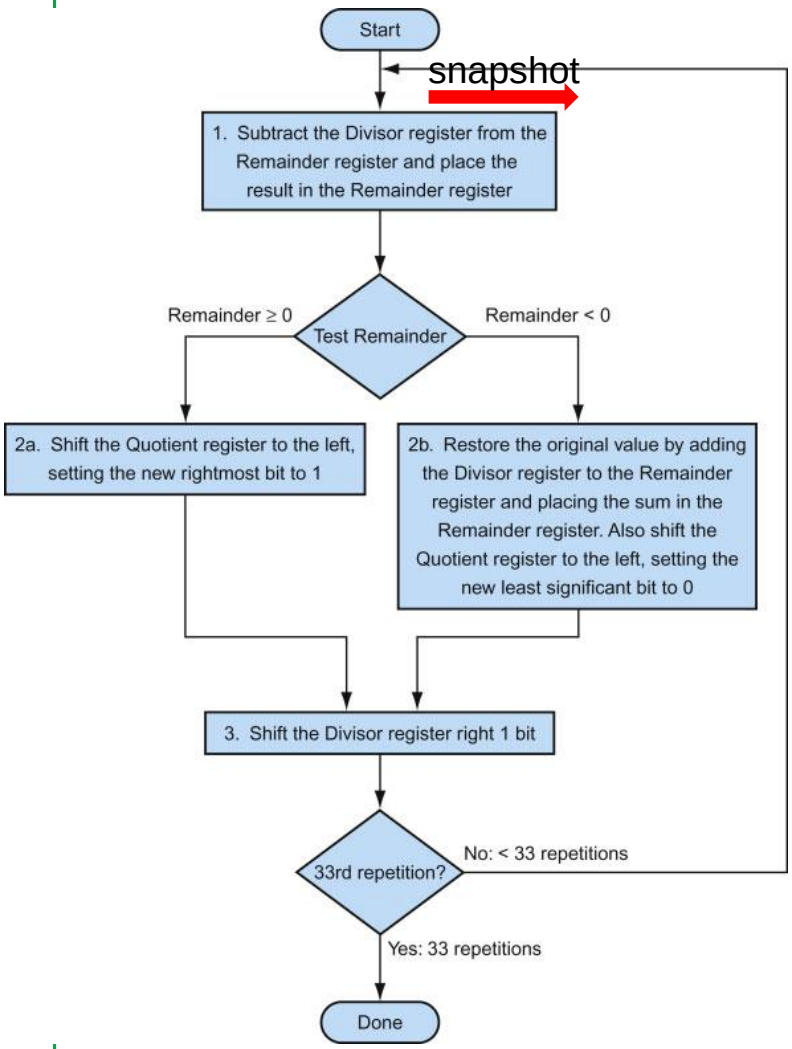
- Divide 7_{ten} ($0000\ 0111_{two}$) by 2_{ten} (0010_{two})

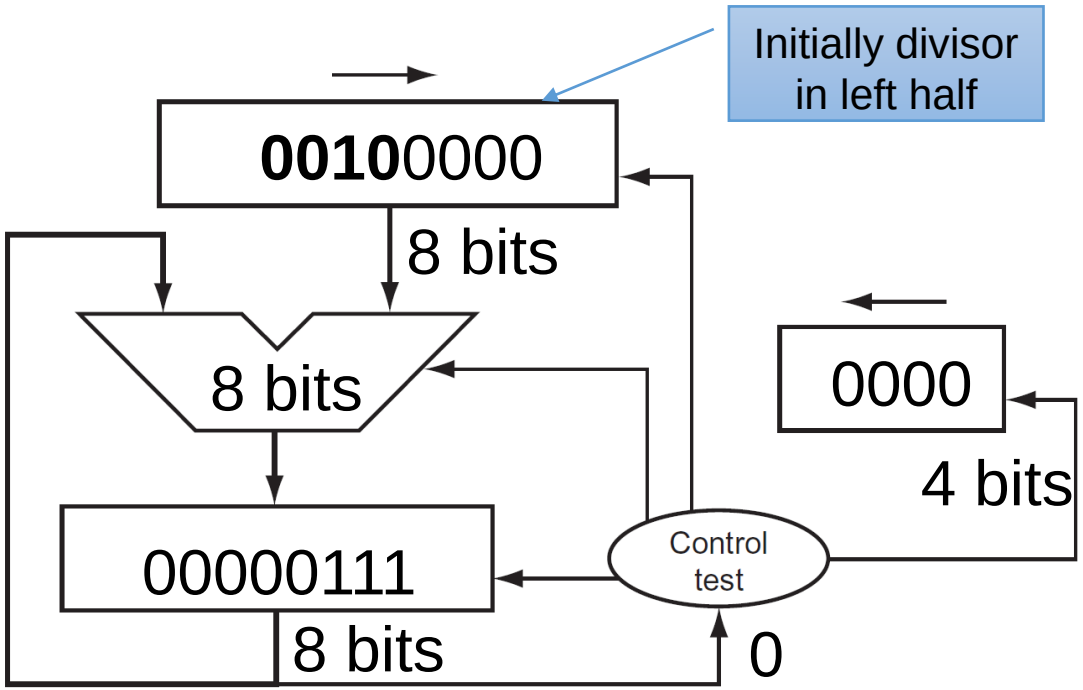




Division Example

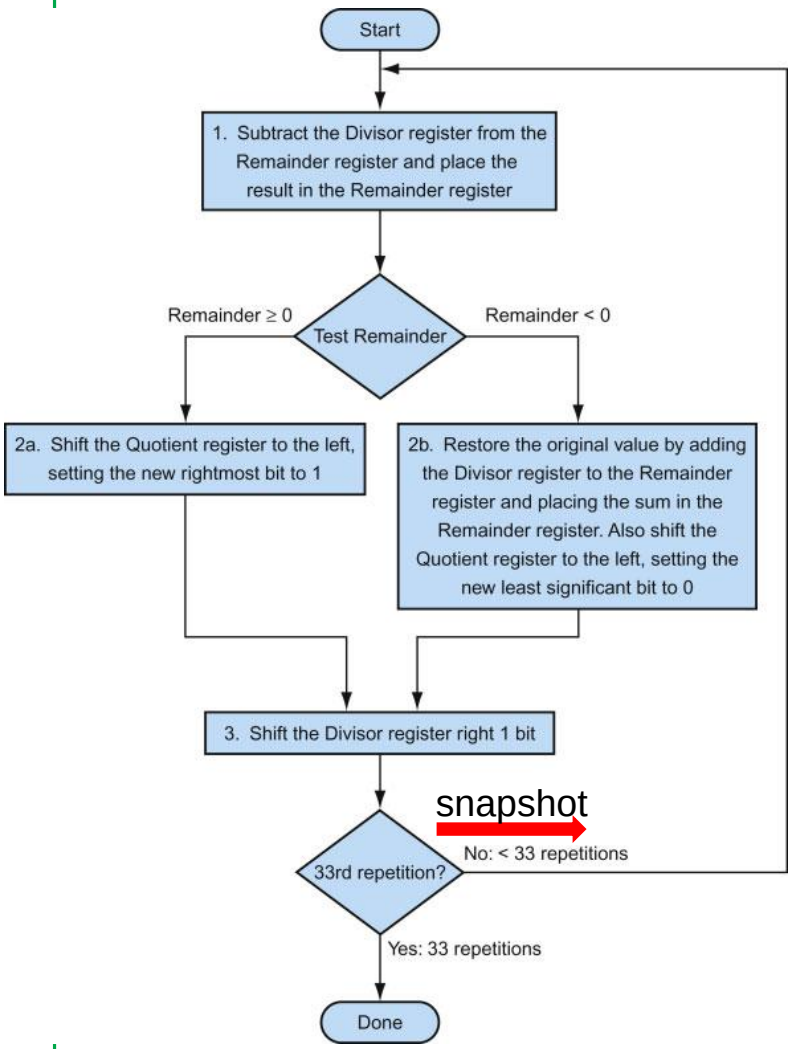
- Divide 7_{ten} ($0000\ 0111_{two}$) by 2_{ten} (0010_{two})

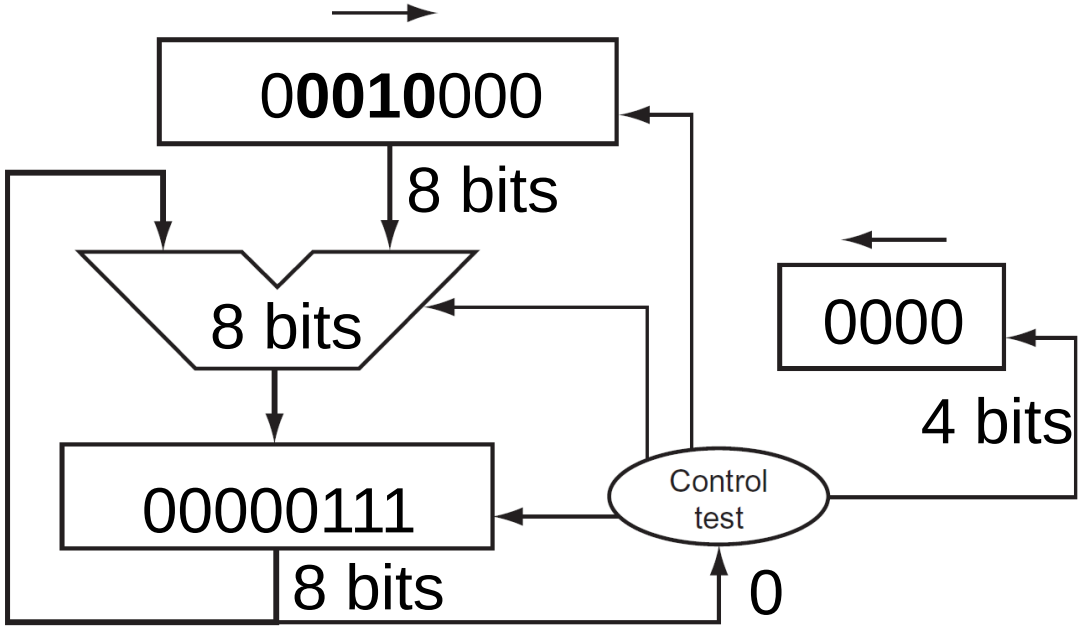




Division Example

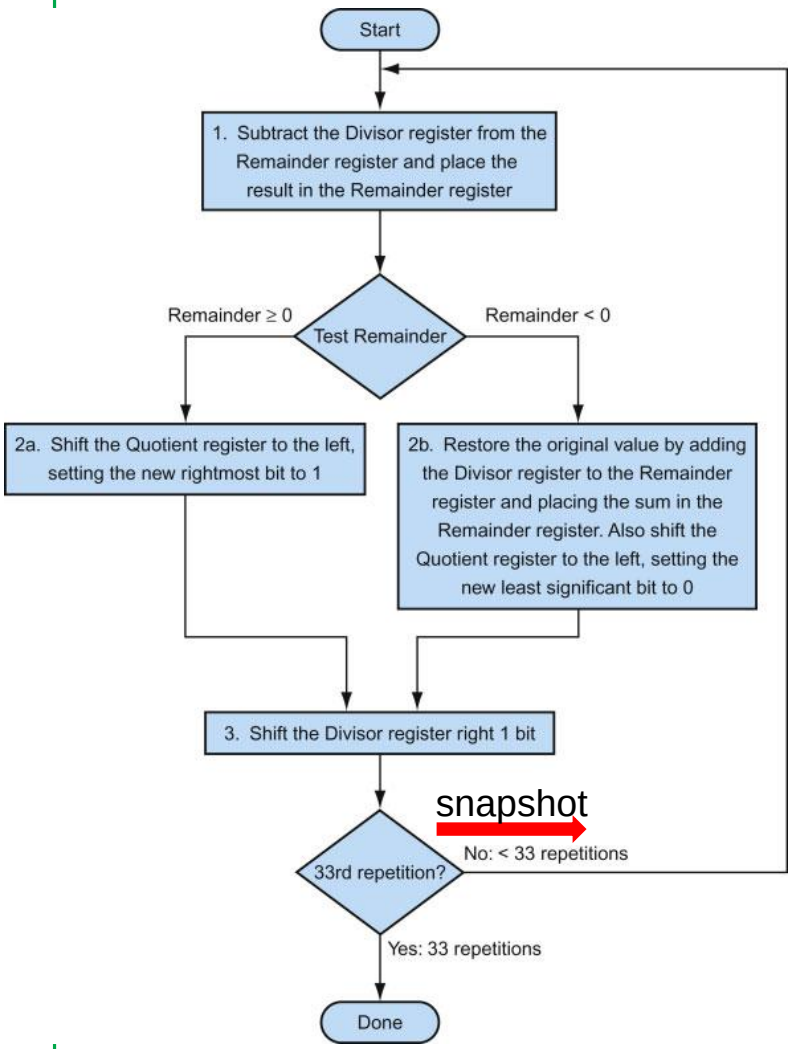
- Divide 7_{ten} ($0000\ 0111_{two}$) by 2_{ten} (0010_{two})

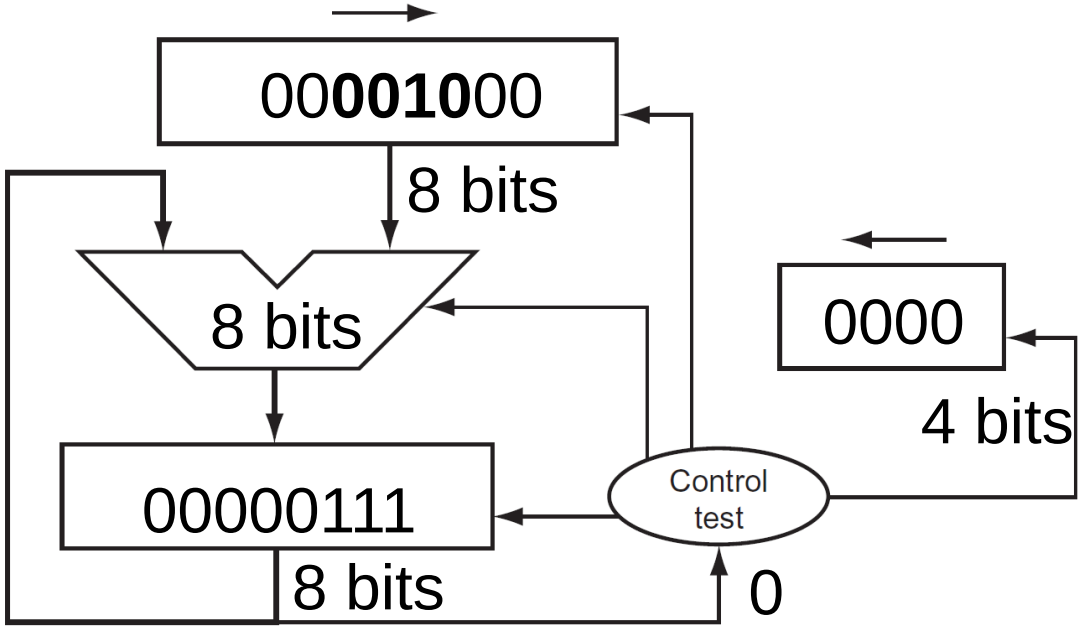




Division Example

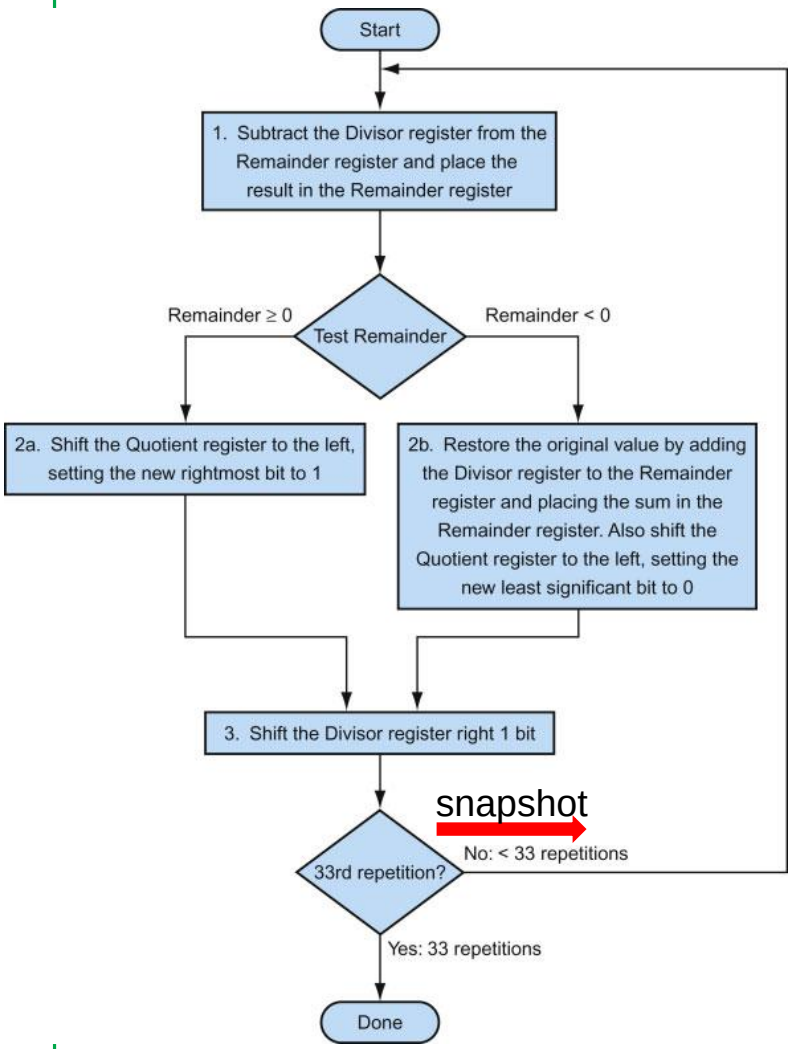
- Divide 7_{ten} ($0000\ 0111_{two}$) by 2_{ten} (0010_{two})

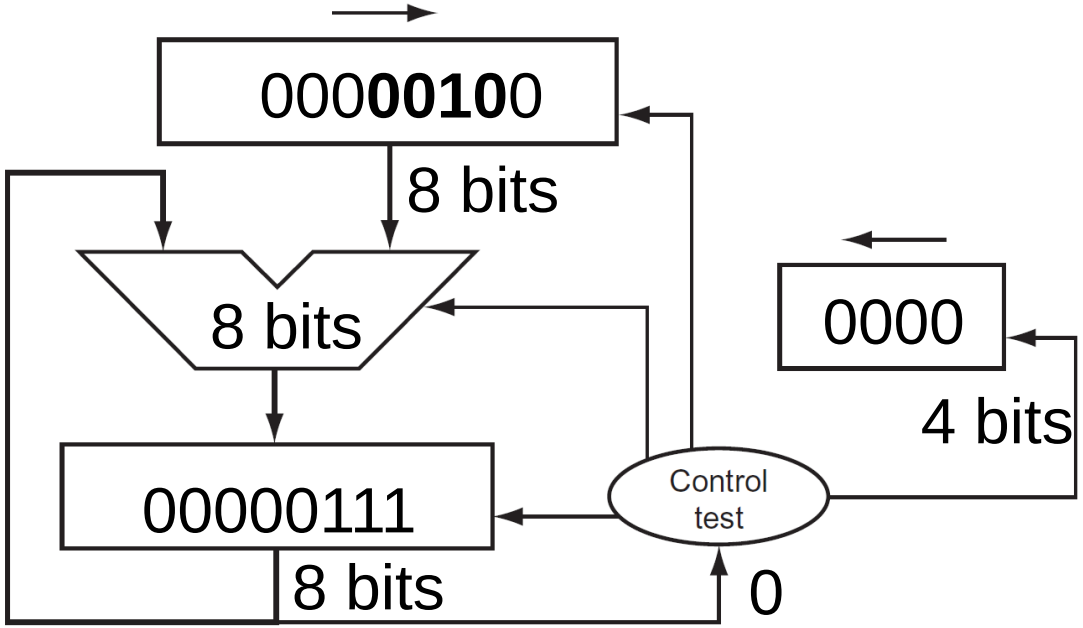




Division Example

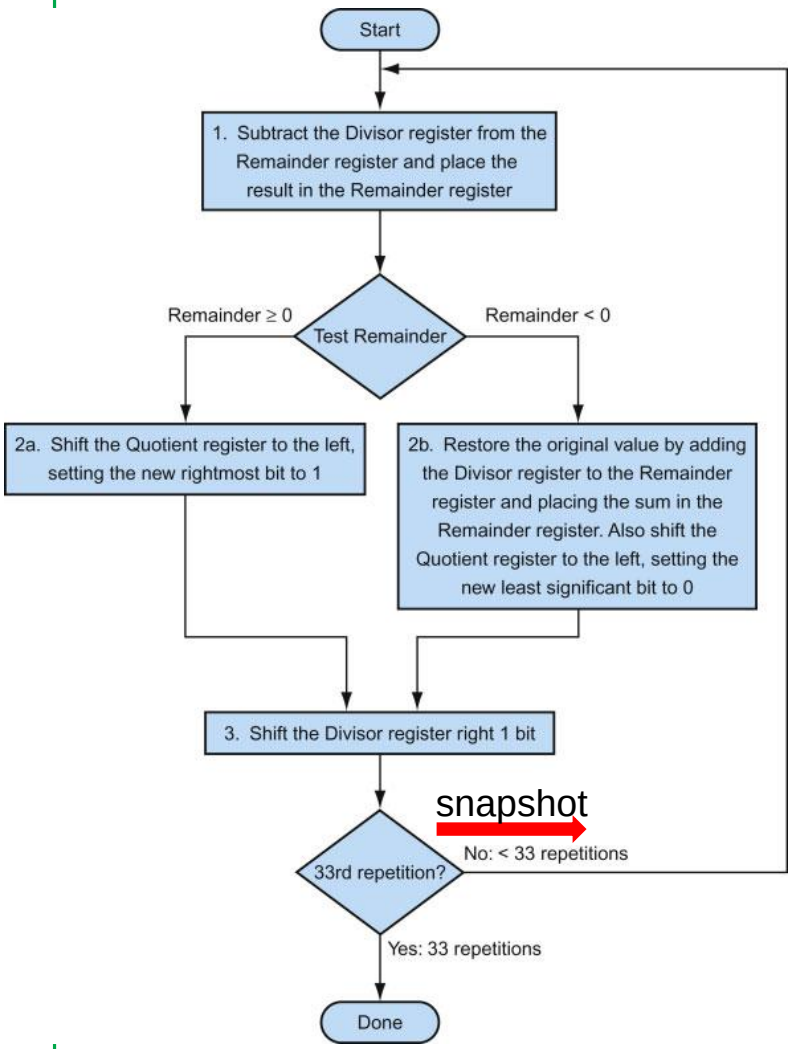
- Divide 7_{ten} ($0000\ 0111_{two}$) by 2_{ten} (0010_{two})

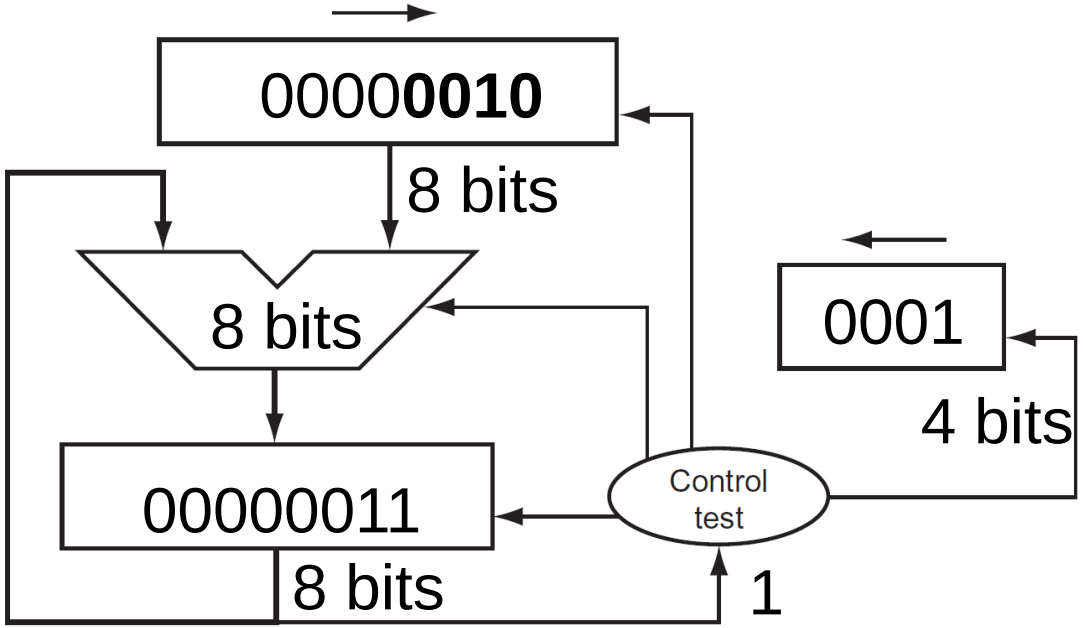




Division Example

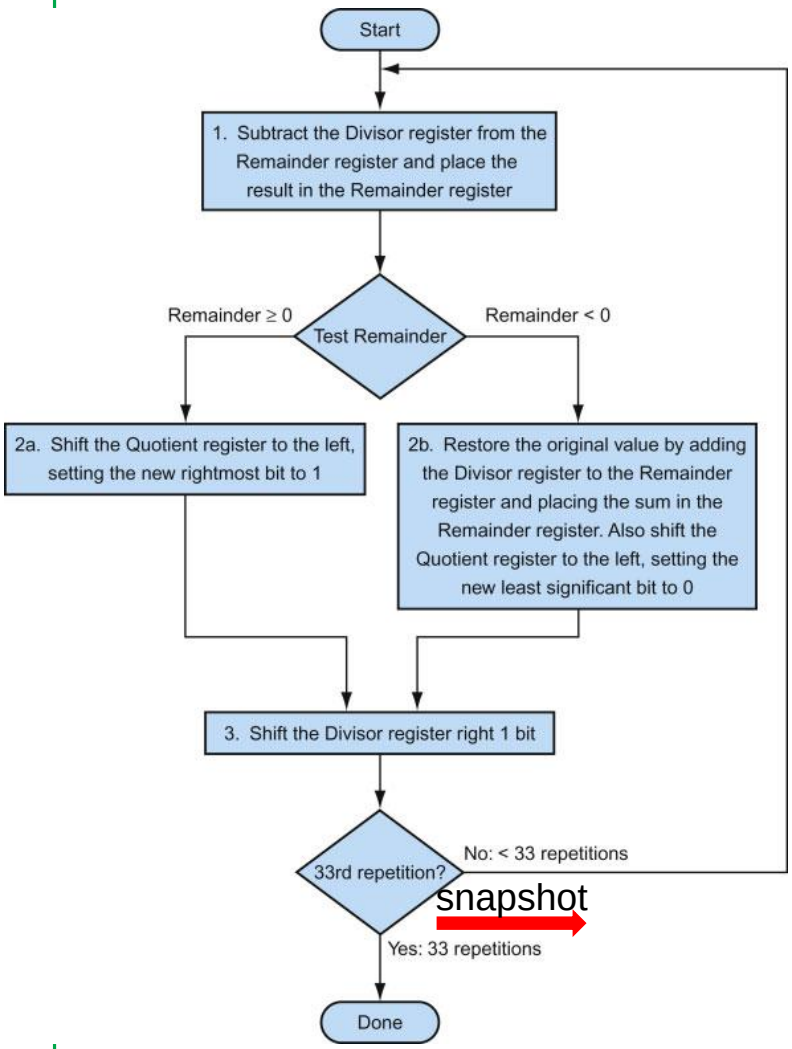
- Divide 7_{ten} ($0000\ 0111_{two}$) by 2_{ten} (0010_{two})

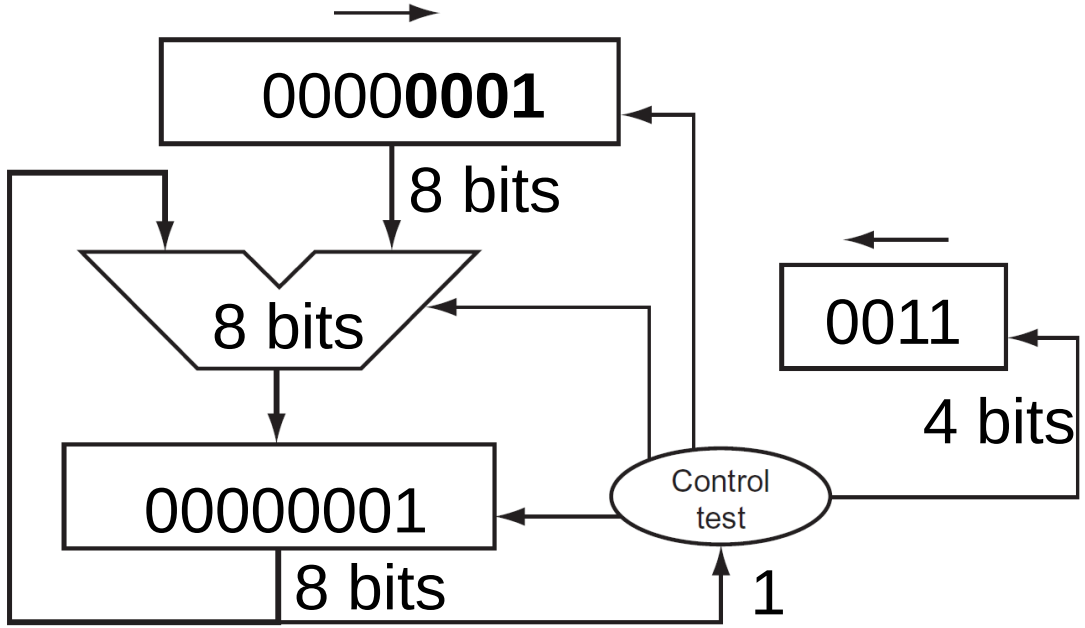




Division Example

- Divide 7_{ten} ($0000\ 0111_{two}$) by 2_{ten} (0010_{two})





Done!

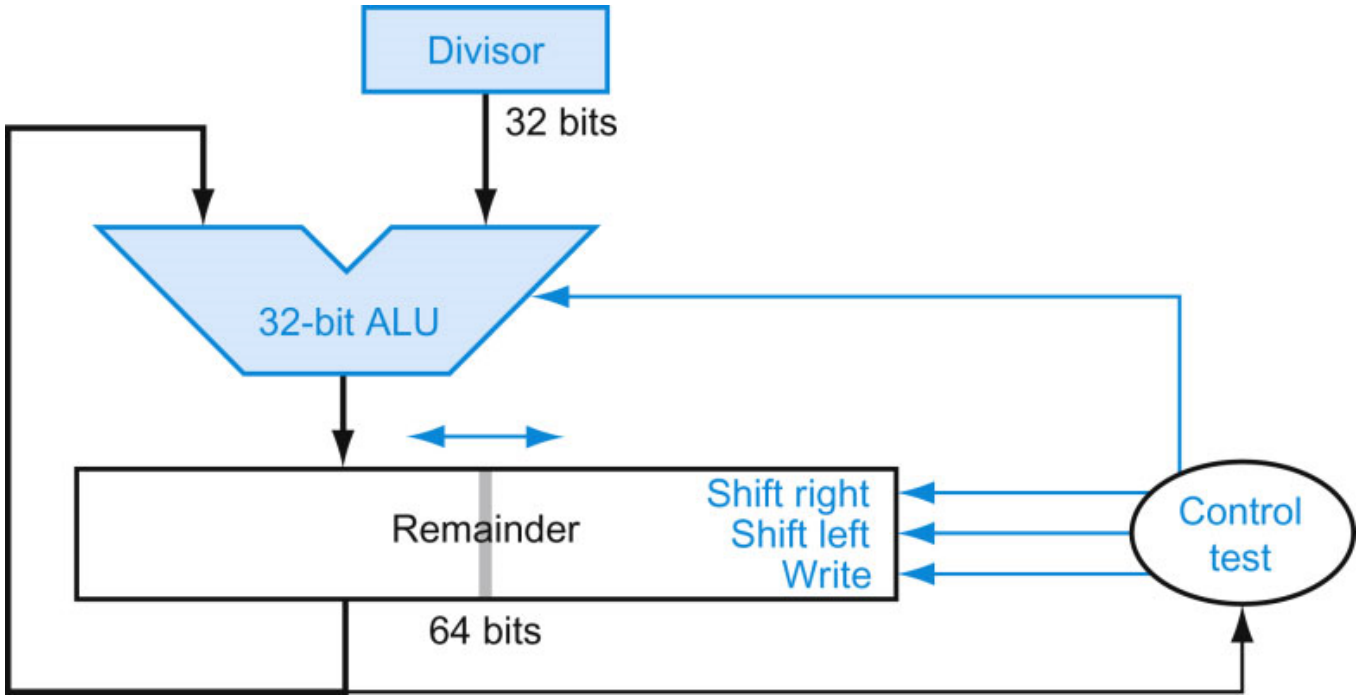
Division Example

Divide 7_{ten} ($0000\ 0111_{two}$) 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 加上他的补码	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

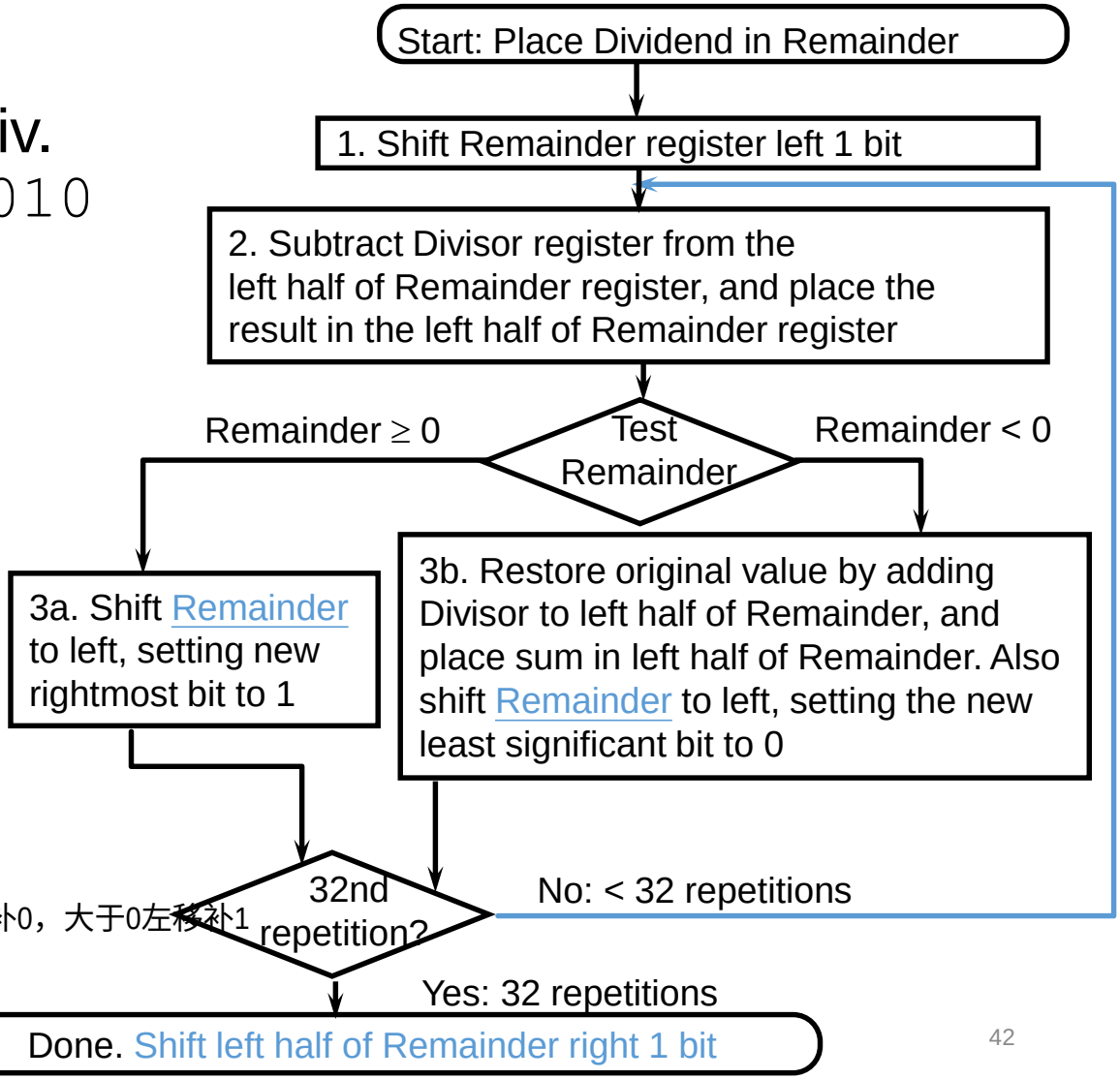
Optimized Divider

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



Optimized Divider

Step	Remainder	Div.
0	0000 0111	0010
1.1	0000 1110	
1.2	1110 1110	
1.3b	$\overline{0}$001 1100	
2.2	1111 1100	
2.3b	$\overline{0}$011 1000	
3.2	0001 1000	
3.3a	$\overline{0}$011 0001	
4.2	0001 0001	
4.3a	$\overline{0}$010 0011	
	0001 0011	



开始先左移一位，然后做减法，小于0左移补0，大于0左移补1

Signed Division

- Convert to positive and adjust sign later
- Note that multiple solutions exist for the equation:
  Dividend = Quotient x Divisor + 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!

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

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

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