Computer Architecture

Arithmetic for Computers Part 1

Operations on integers

Agenda

- Number systems
  - Representing integers
- Operations on integers
  - Addition and subtraction
  - Dealing with overflow
  - Multiplication and division

Number systems

- Bits have no inherent meaning: their interpretations depends on the instructions applied.
- In digital circuits, numbers are represented by a finite number of bits.
- Precision issues:
    - Overflow
    - Round-off errors in floating point representations

Binary numbers

- A binary string (also called binary number) is a set of columns 0s and 1s.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1

(32 bits wide)

Most significant bit

Least significant bit

- Notations (i.e., maps) between binary strings and integers
  - Unsigned integers
  - 2s-complement notation (signed integers)
  - Excess notation (signed integers, used in representing floating-point numbers)

Unsigned Binary Integers

- Given an n-bit binary number, the corresponding decimal number is

  $$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_1 2^1 + x_0 2^0$$

  Range: 0 to 2^n-1

- For a 32-bit binary number, it can represent any decimal within 0 to +4,294,967,295.

2s-Complement Signed Integers

- The most significant bit is set to be the sign bit:
  - If 0 the number is non-negative (from 0 to 2^{n-1}-1)
  - If 1 the number is negative

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1

(32 bits wide)

Sign bit

- Some specific numbers
  - `0: 0000 0000 …. 0000`
  - `-1: 1111 1111 …. 1111`
  - Most-negative: `1000 0000 …. 0000`
  - Most-positive: `0111 1111 … 1111`

b. Using patterns of length four

Bit pattern	Value represented
0111	7
0110	6
0101	5
0100	4
0011	3
0010	2
0001	1
0000	0
1111	−1
1110	−2
1101	−3
1100	−4
1011	−5
1010	−6
1001	−7
1000	−8

2s-Complement Signed Integers

- Given an n-bit number in two's complement form, we have

  $$x = \boxed{-x_{n-1}2^{n-1}} + x_{n-2}2^{n-2} + \dots + x_1 2^1 + x_0 2^0$$

  Range: -2^{n-1} to $2^{n-1} - 1$ (Note: it is not symmetric.)

- Example: a 32-bit binary
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$

    $= -1\text{x}2^{31} + 1\text{x}2^{30} + \dots + 1\text{x}2^2 + 0\text{x}2^1 + 0\text{x}2^0$

    $= -2{,}147{,}483{,}648 + 2{,}2147{,}483{,}644 = -4_{10}$

Signed Negation

- A quick way to negate a two's complement binary number: invert every 0 to 1 and every 1 to 0, then add one to the result.

$$x + \bar{x} = 1111...111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Example: negate +2

$+2 = 0000\ 0000\ \ldots\ 0010_2$

$-2 = 1111\ 1111\ \ldots\ 1101_2 + 1$

$= 1111\ 1111\ \ldots\ 1110_2$

Sign Extension of 2s-complement numbers

- In digital circuits, we sometime need to represent a number using more bits (but the numeric value should be preserved).
- It can be done by replicating the sign bit to the left
  - E.g., 8-bit to 16-bit
    - +2: 0000 0010 ⇒ 0000 0000 0000 0010
    - -2: 1111 1111 ⇒ 1111 1111 1111 1110
- The following MIPS instructions will extend the numbers when needed
  - `addi`: extend immediate value
  - `lb`, `lh`: extend loaded byte/halfword
  - `beq`, `bne`: extend the displacement

Hexadecimal numbers

In practice, it is often convenient to group 0’s and 1’s into groups of four bits.

Each 4-bit group is called a nibble (a half byte). It can represent a number between 0 and 15, which leads to the hexadecimal notation.

Hexadecimal notation is very useful to denote the memory addresses of the computer.

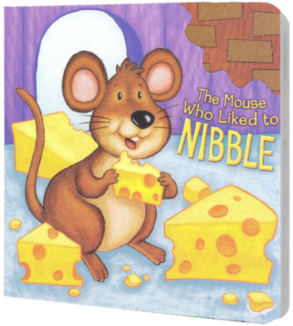
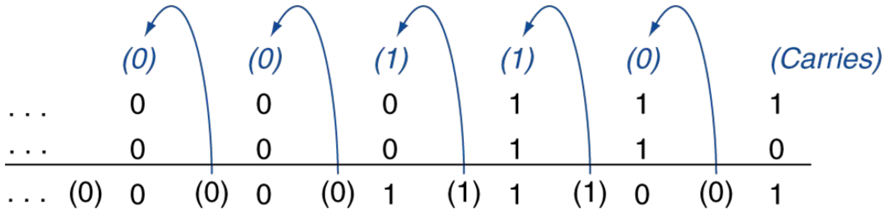


Table 1.2 Hexadecimal number system

Hexadecimal Digit	Decimal Equivalent	Binary Equivalent
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Integer addition

- Digits are added bit by bit from right to left, which carries passed to the next digit to the left. (just as you would do by hand)



- Example: 7+6

$$
\begin{array}{ll}
 & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} = 7_{ten} \\
+ & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{two} = 6_{ten} \\
\hline
= & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_{two} = 13_{ten}
\end{array}
$$

Integer subtraction

- Add negation of the second operand
- Example: 7 - 6 = 7 + (-6)
- In two's complement representation,

$$
\begin{array}{lr}
 & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} = 7_{ten} \\
+ & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1010_{two} = -6_{ten} \\
\hline
= & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}
\end{array}
$$

Subtraction of unsigned numbers

Subtraction of two unsigned integers can also be implemented by addition using two’s complement.

- Example: 7 - 6 = 7 + (-6)
  - Find the two’s complement of 6 (0110): invert the bits in 0110, then add 1⇒1010
  - Add -6 (1010) to 7 (0111) = 0001; the carry out of 4 bits is ignored.
- This approach simplifies the design of arithmetic circuits.
  - Only an adder is required; no need for a separated subtractor.

Overflow

- Overflow occurs when the result from an operation cannot be represented with the available hardware. (in our case, it is 32 bits)
  - If the sum of two numbers requires 33 bits to represent, it is an overflow.
    - e.g., an integer > 2^{n-1}-1 in 2's-complement notation
- Overflow cannot occur:
  - When adding operands with different signs
    - Because the sum is no larger than one of the operands (e.g., -10 + 4 = -6)
  - When subtracting operands with the same signs
    - c - a = c + (-a); because it ends up by adding operands of different signs.

Overflow of unsigned integers

- In some cases, we need to ignore the overflows
  - Unsigned integers are commonly used for memory addresses where overflows are ignored.
- C and Java ignores overflows
- MIPS provides instructions for such cases.
  - To ignore the overflows: using `addu`, `addui`, `subu` instructions

Dealing with Overflow

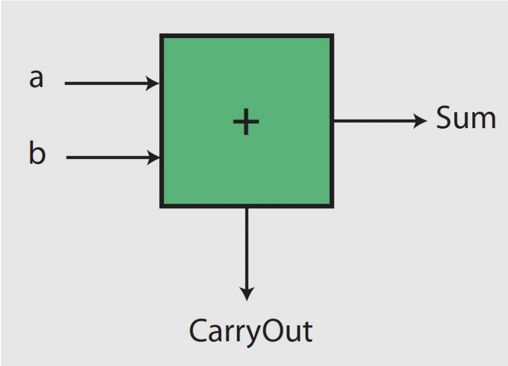
- Some languages (e.g., Ada, Fortran) require raising an exception for overflow.
- Use `add, addi, sub` instructions to raise exceptions
  - On overflow, these instructions invoke exception handler
    - Save PC in exception program counter (EPC) register
    - Jump to predefined handler address

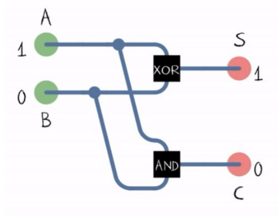
Handling overflow manually

```
# addition with overflow handler

addu $t0, $t1, $t2              #does not interrupt in case of overflow!
xor  $t3, $t1, $t2              #if signs of $t1 and $t2 differ, then, $t3<0
slt  $t3, $t3, $zero            #compare the value of $t3 with zero
bne  $t3, $zero, No_overflow    #no overflow when signs differ

xor  $t3, $t0, $t1              #when signs of $t1 and $t2 match
                                #compare the sign of $t1 with
                                #the sign of the sum $t0
                                #if signs of $t0 and $t1 differ then $t3<0
slt  $t3, $t3, $zero            #compare the value of $t3 with zero
bne  $t3, $zero, Overflow       #jump to Overflow branch when sign is differ
```

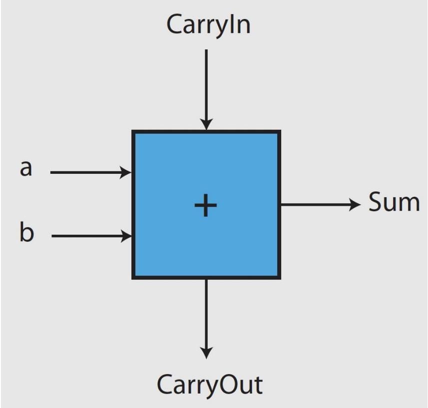
Adder

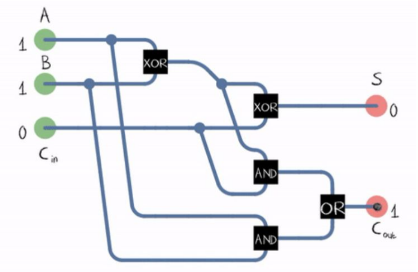
- Half adder: it has two inputs and two outputs: sum and carry out





- Full adder: it has three inputs and two outputs: sum and carry out





Carry propagate adder

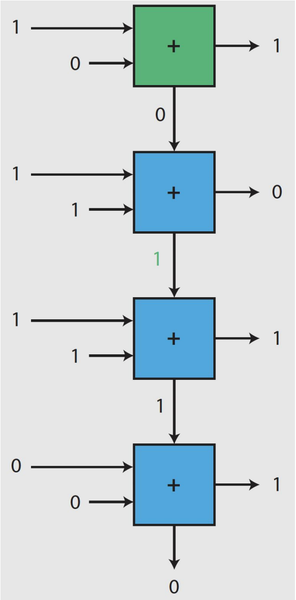
A N-bit adder sums **two** N-bit inputs. It adds the bits in the two inputs with the carries, respectively.

It is called carry propagate adder (CPA) because the carry out of one bit propagates into the next bit.

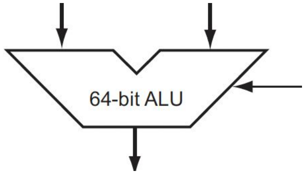
Ripple-Carry Adder: it is built by simply chaining together N full adders. (the first one has a 0 Carry_{in})

Arithmetic Logic Unit: it is a device that performs the arithmetic operations like addition and subtraction or logical operations.

Ripple-carry adder



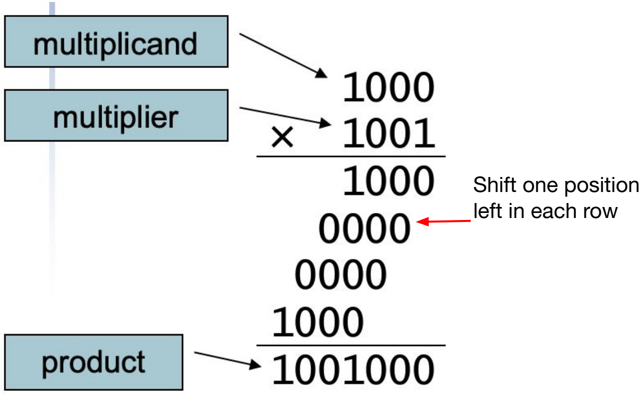
ALU



Multiplication of binary numbers

The long-multiplication approach:

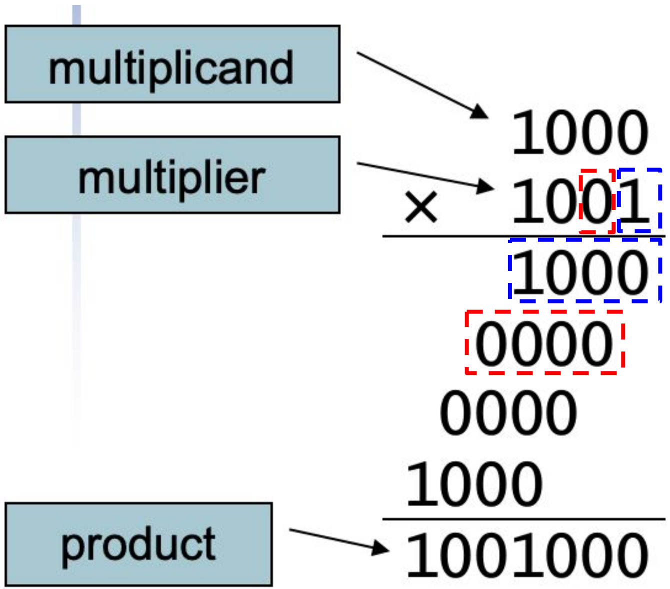
- **Multiply**: for each bit in the multiplier, multiply the entire multiplicand by the bit (0 or 1).
- **Shift**: shift the result left by a number of positions corresponding to the bit's position in the multiplier.
- **Add**: sum the partial products to get the final result.



The (**maximum**) **length** of the product is equal to the sum of operand lengths. In practice, there are leading zeros in the product if its length is smaller than the sum.

Length of product is the sum of operand lengths

Multiplication of binary numbers



In a binary number, each digit can only be 0, or 1. So,

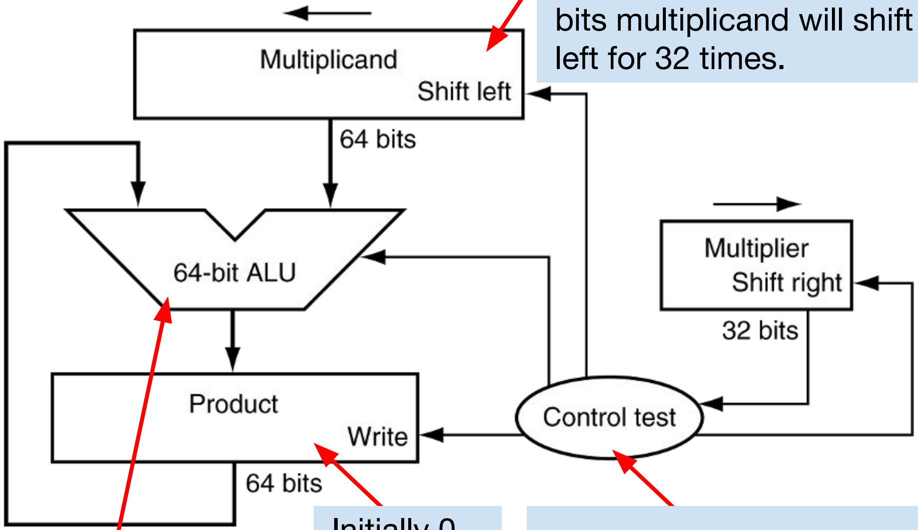
- each row in the column multiplication will be the multiplicand if the digit is 1, or 0 if the digit is 0.

The multiplication can be implemented by the two operations repeatedly:

- add the multiplicand to the product if the corresponding bit in multiplier is 1;
- shift the multiplicand to left for some bits accordingly.

The logic of multiplication

It is 64 bits because a 32 bits multiplicand will shift left for 32 times.

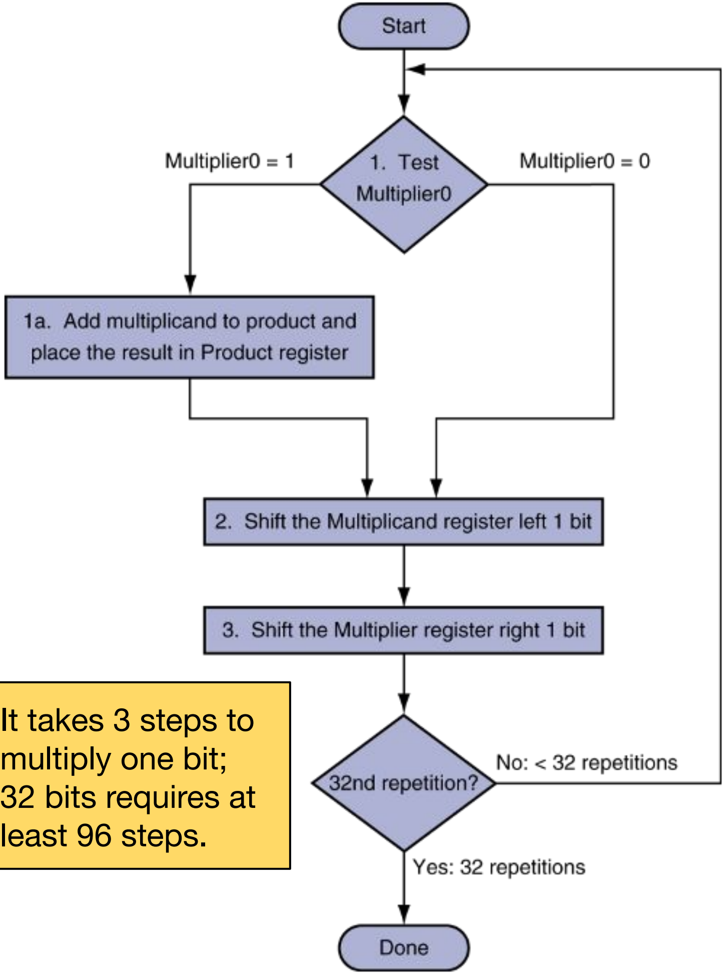


Initially 0

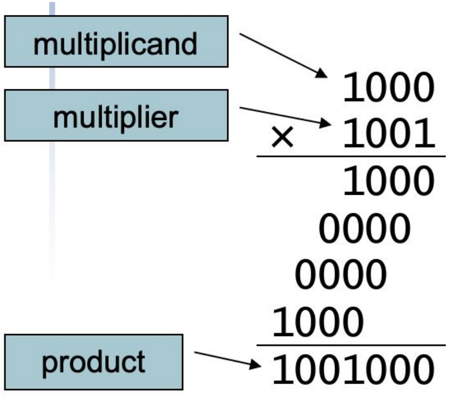
Adding product and the shifted multiplicand

Decide when to shift multiplicand and multiplier registers

It takes 3 steps to multiply one bit; 32 bits requires at least 96 steps.

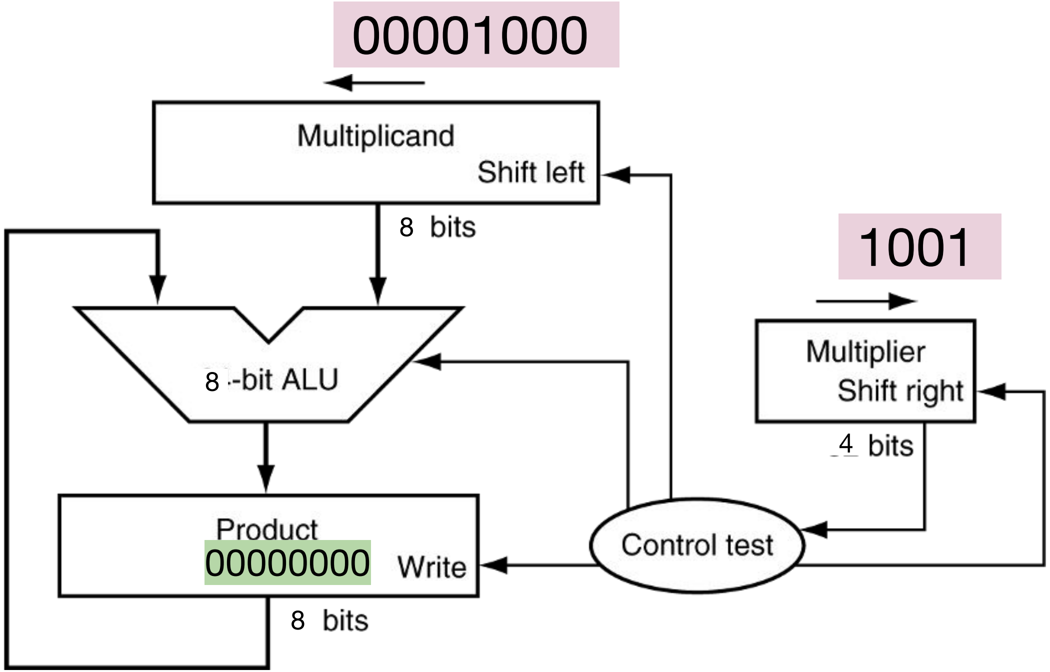


Example

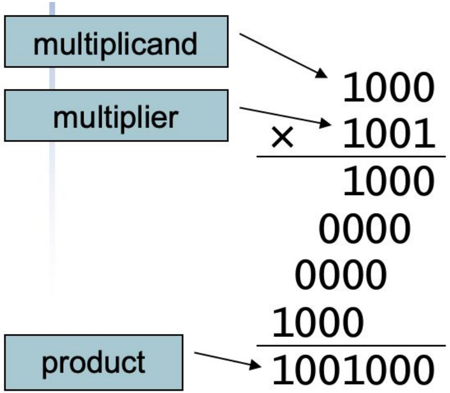


For simplification, we assume the circuits are designed to work on add two 4-bit numbers.

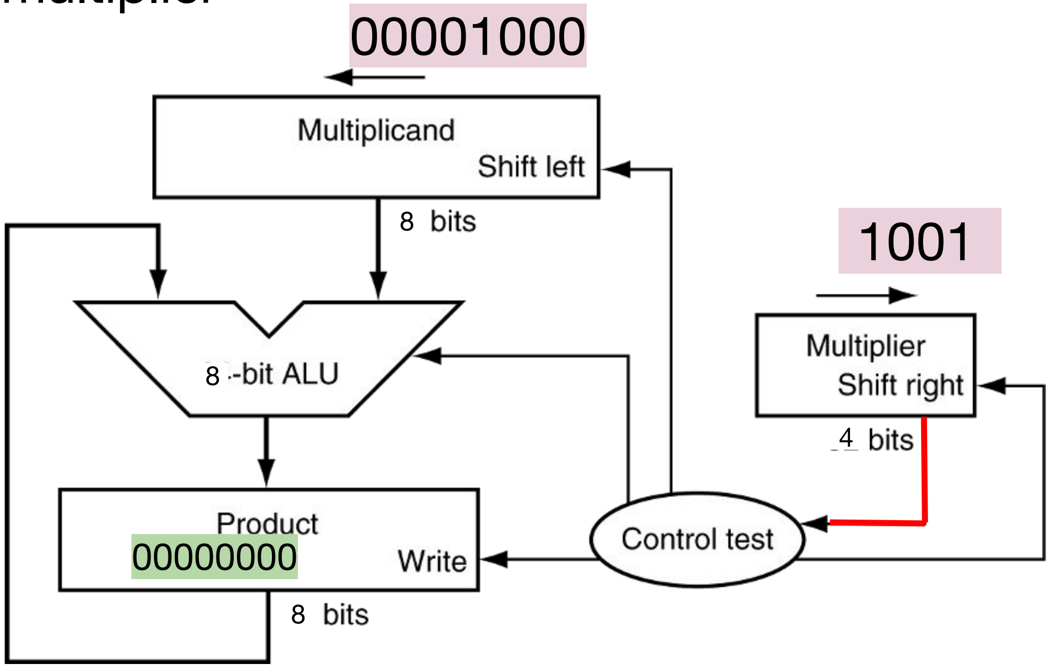
Iteration 0: initializing the registers.



Example



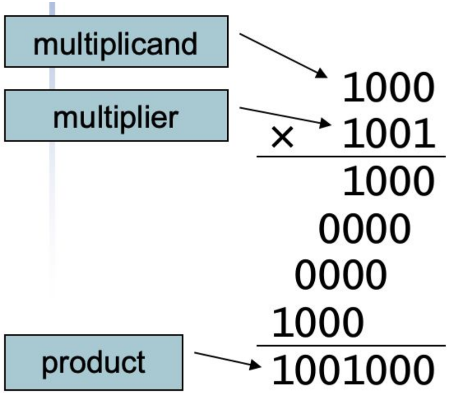
Iteration 1: a. Check the rightmost bit of the multiplier



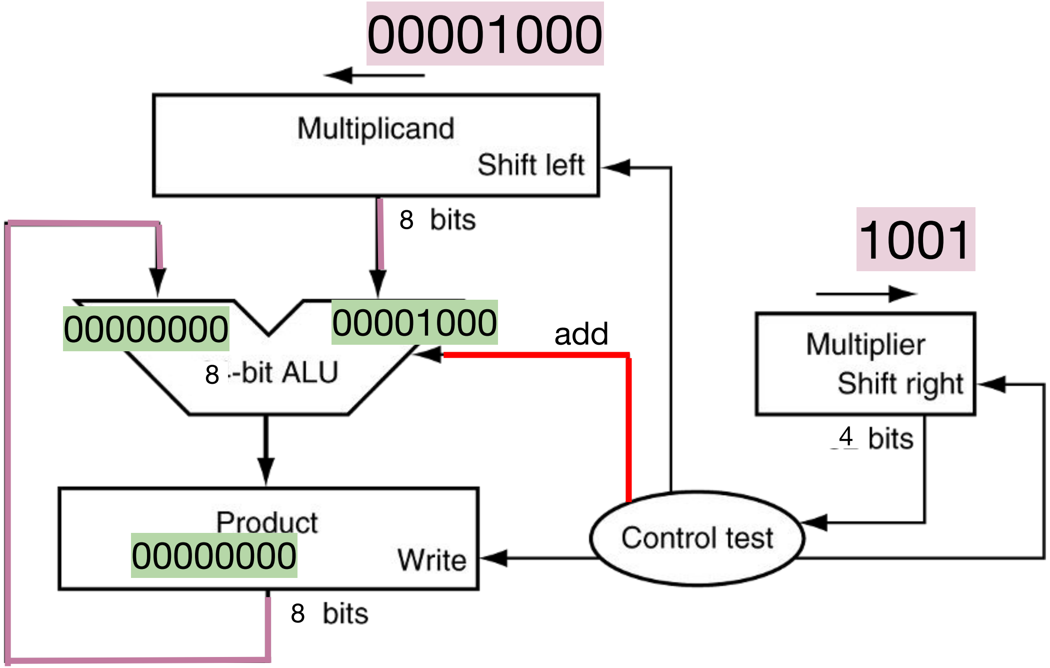
For simplification, we assume the circuits are designed to work on add two 4-bit numbers.

Example

Iteration 1: b. Inform the ALU to add the inputs

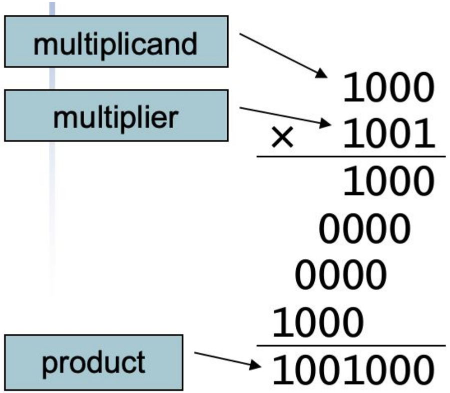


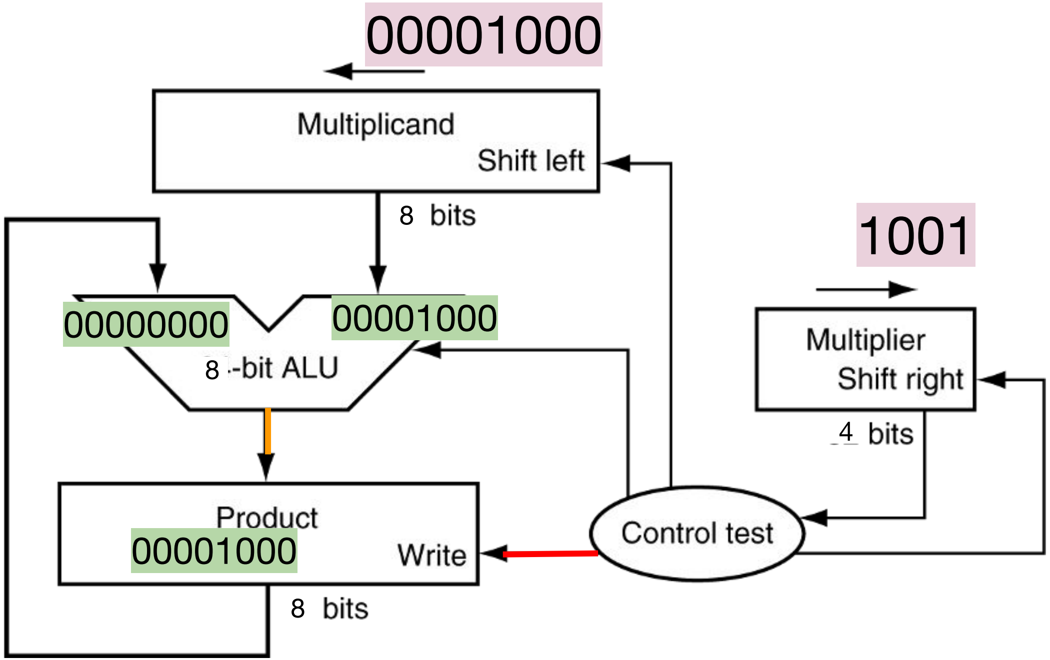
For simplification, we assume the circuits are designed to work on add two 4-bit numbers.



Example

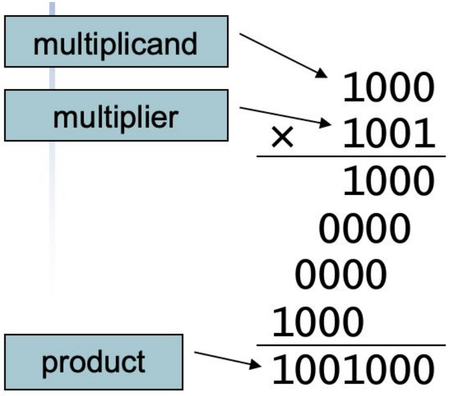
Iteration 1: c. Inform the Product register to write





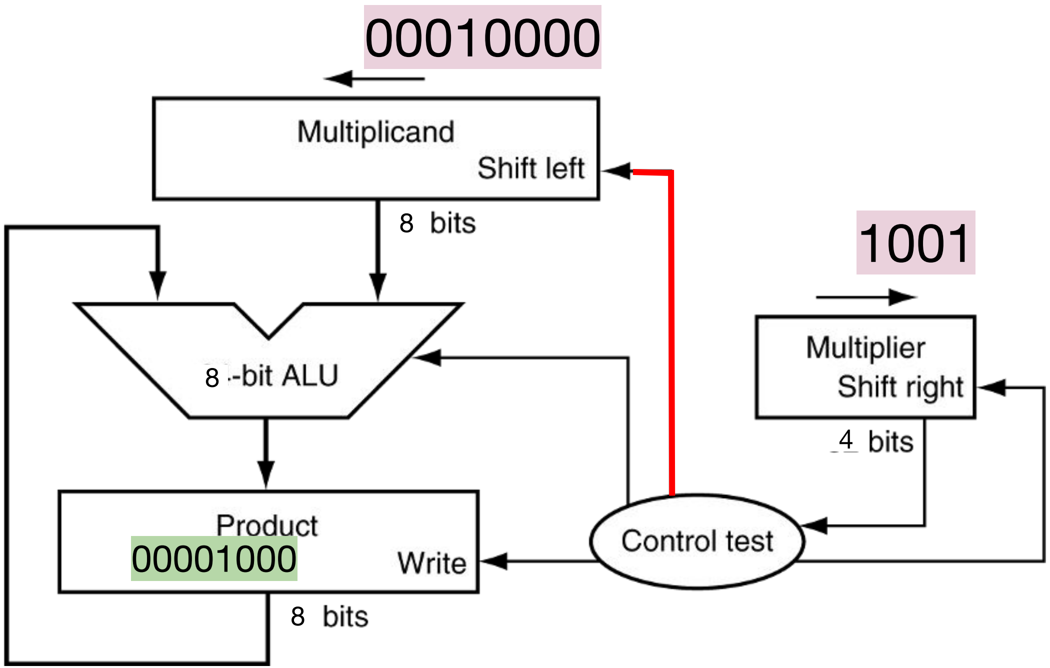
For simplification, we assume the circuits are designed to work on add two 4-bit numbers.

Example



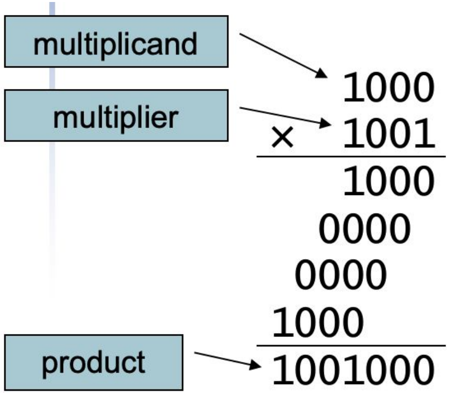
For simplification, we assume the circuits are designed to work on add two 4-bit numbers.

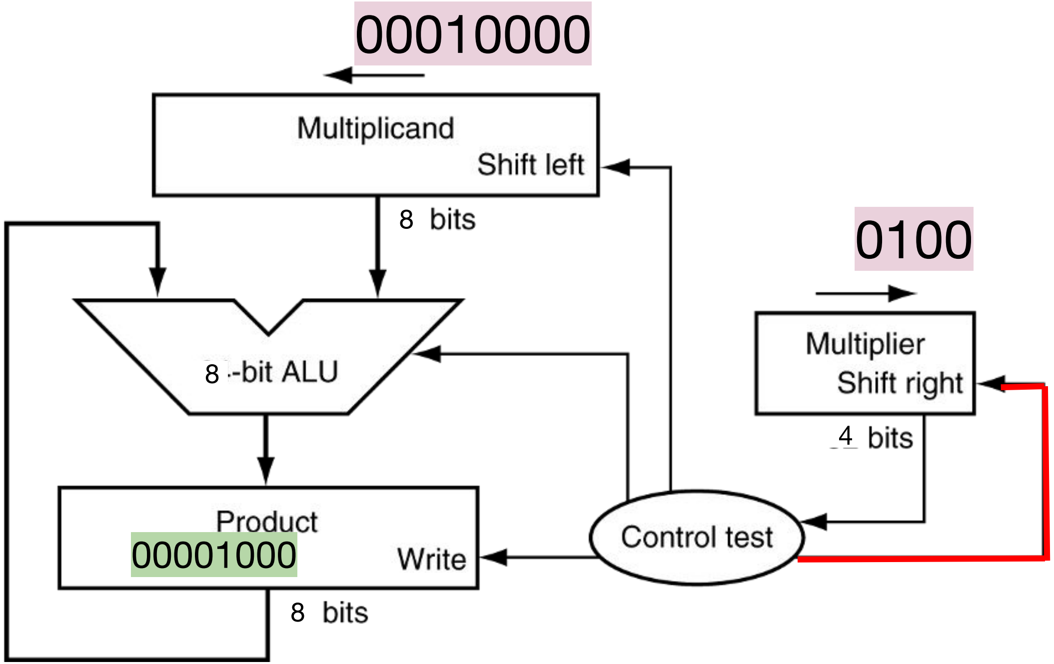
Iteration 1: d. shift the multiplicand left by 1 bit



Example

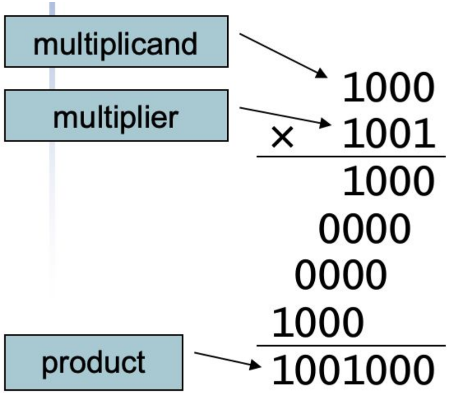
Iteration 1: e. shift the multiplier right by 1 bit



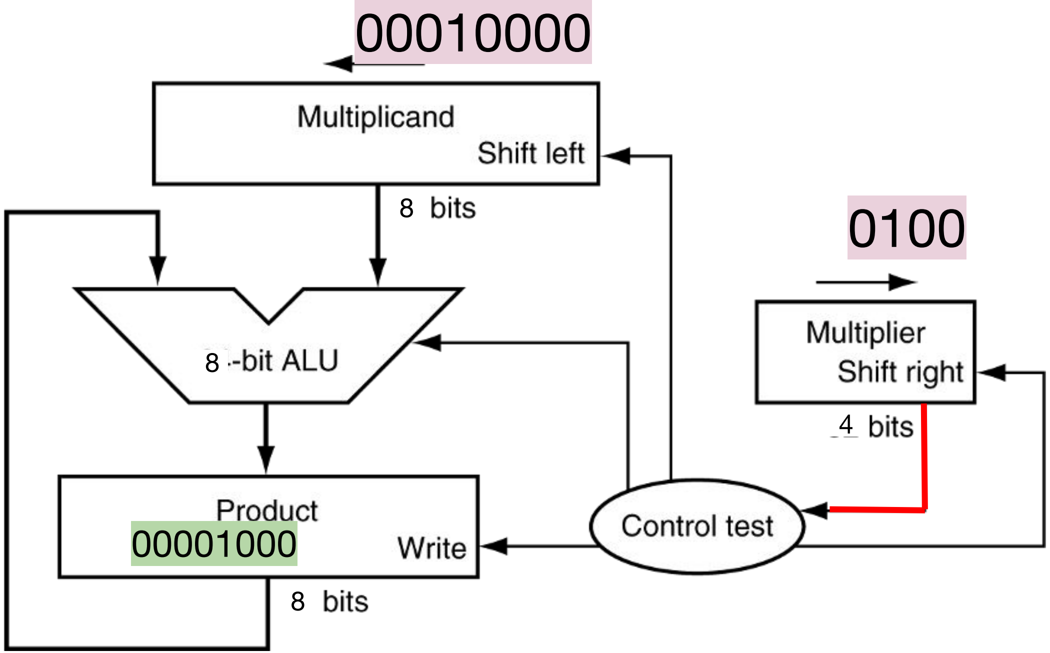


For simplification, we assume the circuits are designed to work on add two 4-bit numbers.

Example

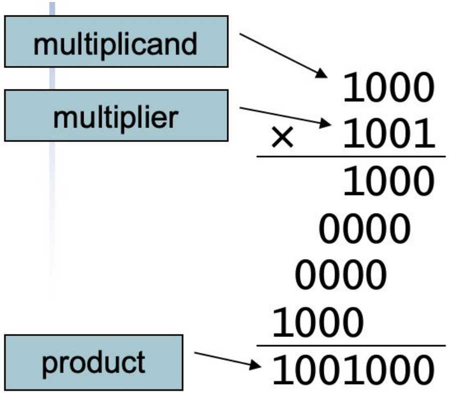


Iteration 2: a. Check the rightmost bit of the multiplier (it is 0 this time)



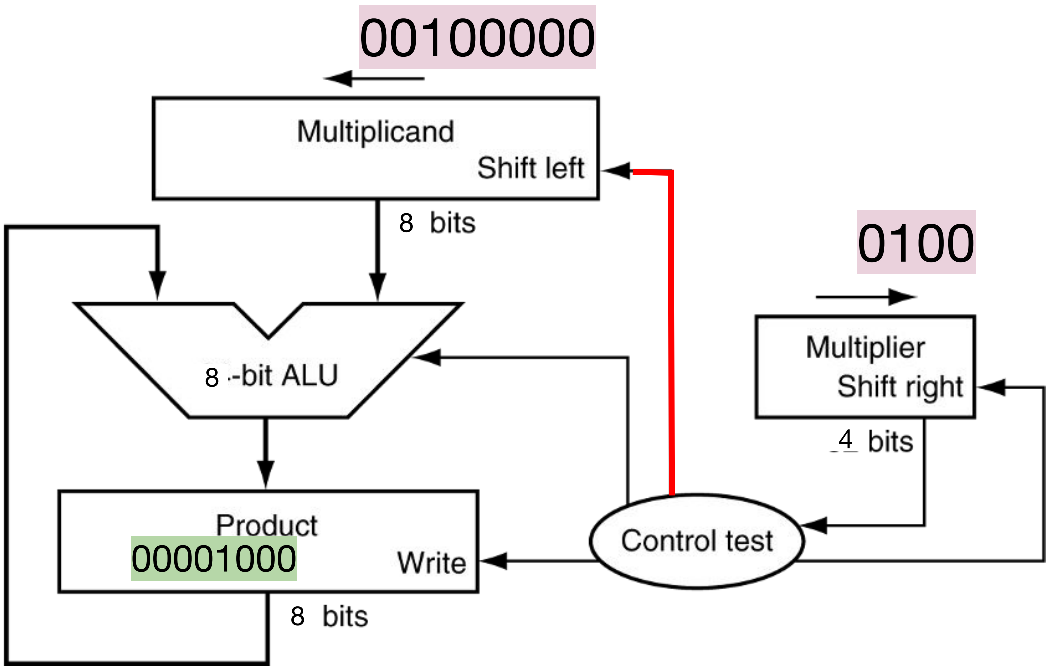
For simplification, we assume the circuits are designed to work on add two 4-bit numbers.

Example

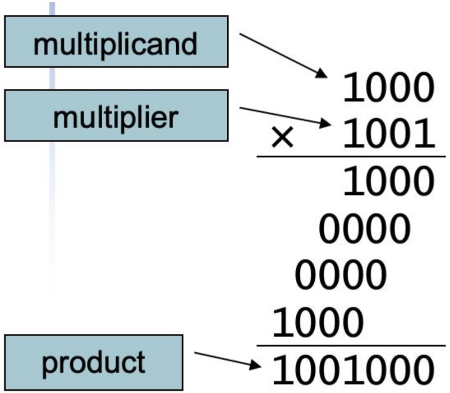


For simplification, we assume the circuits are designed to work on add two 4-bit numbers.

Iteration 2: b. shift the multiplicand left by 1 bit

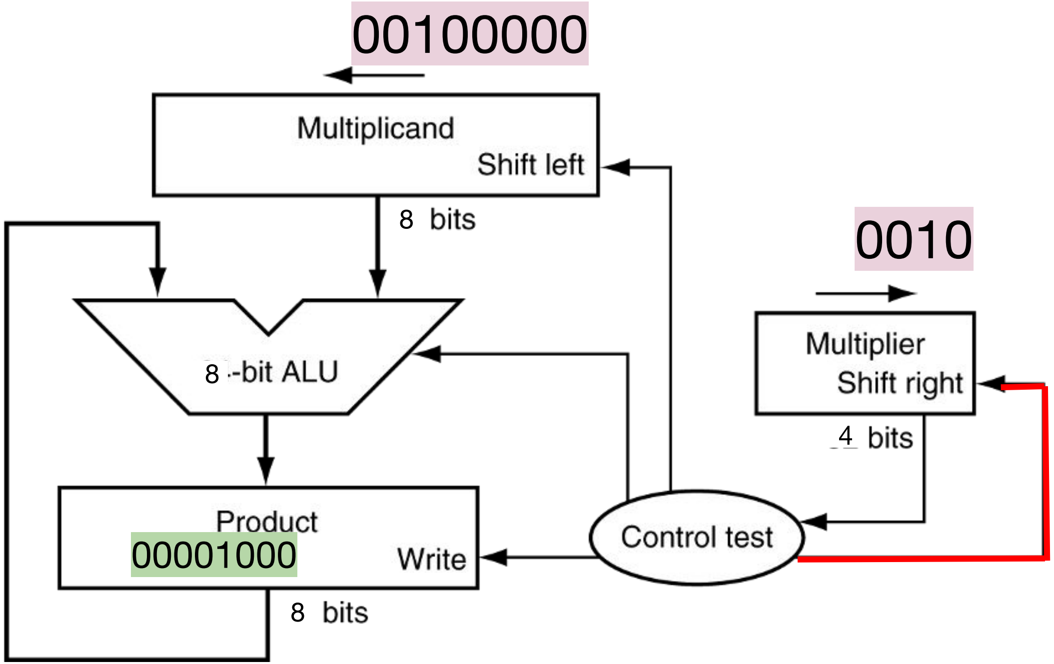


Example



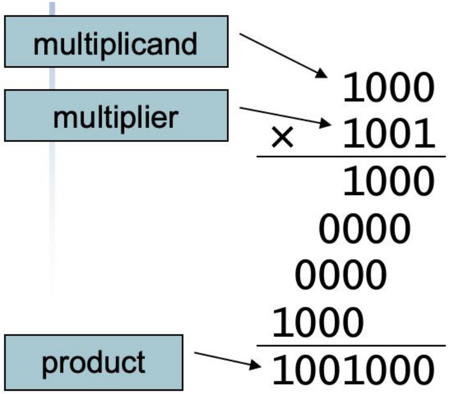
For simplification, we assume the circuits are designed to work on add two 4-bit numbers.

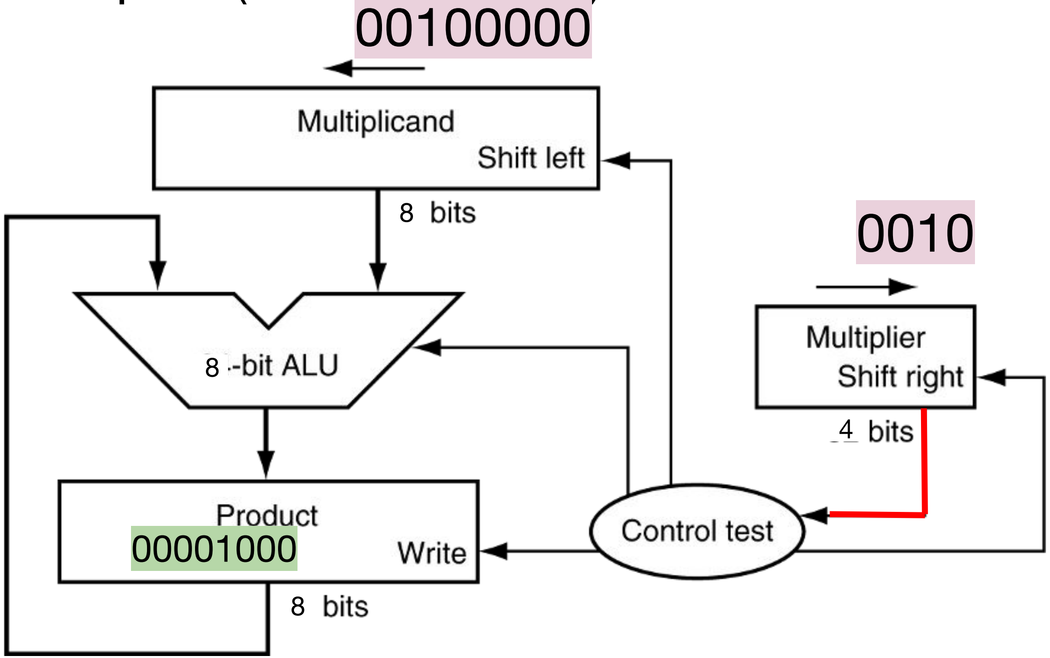
Iteration 2: c. shift the multiplier right by 1 bit



Example

Iteration 3: a. Check the rightmost bit of the multiplier (it is 0 this time)





For simplification, we assume the circuits are designed to work on add two 4-bit numbers.

Example

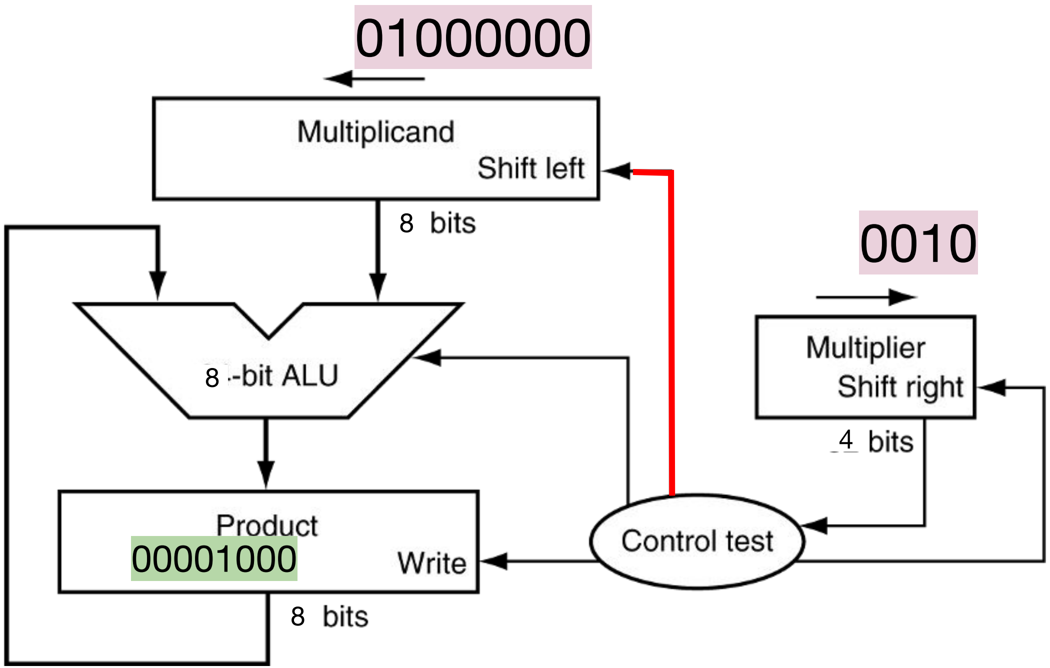
Iteration 3: b. shift the multiplicand left by 1 bit

multiplicand → 1000

multiplier → × 1001

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

product → 1001000



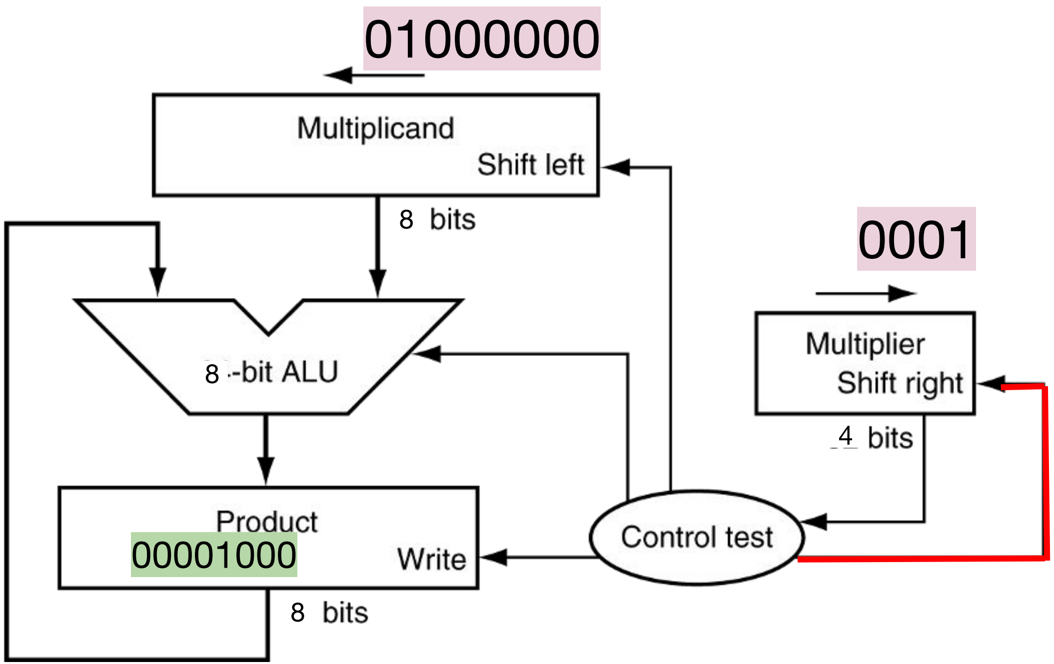
For simplification, we assume the circuits are designed to work on add two 4-bit numbers.

Example

Iteration 3: c. shift the multiplier right by 1 bit

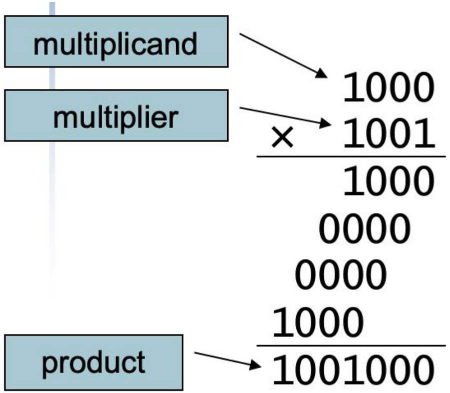
multiplicand
multiplier
product

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



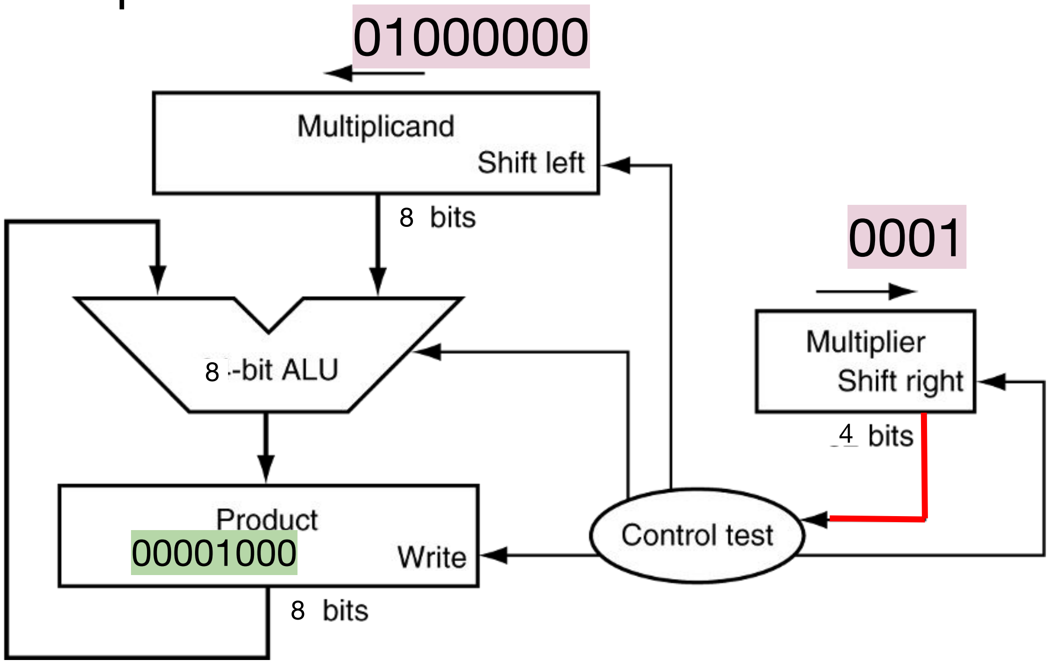
For simplification, we assume the circuits are designed to work on add two 4-bit numbers.

Example

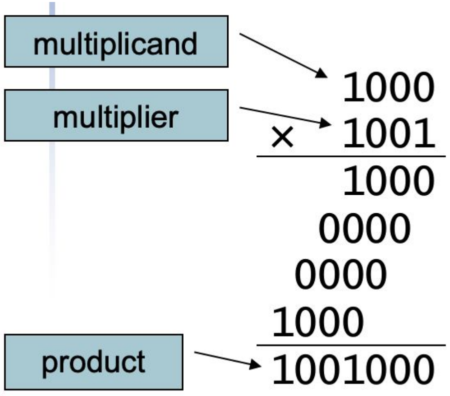


For simplification, we assume the circuits are designed to work on add two 4-bit numbers.

Iteration 4: a. Check the rightmost bit of the multiplier

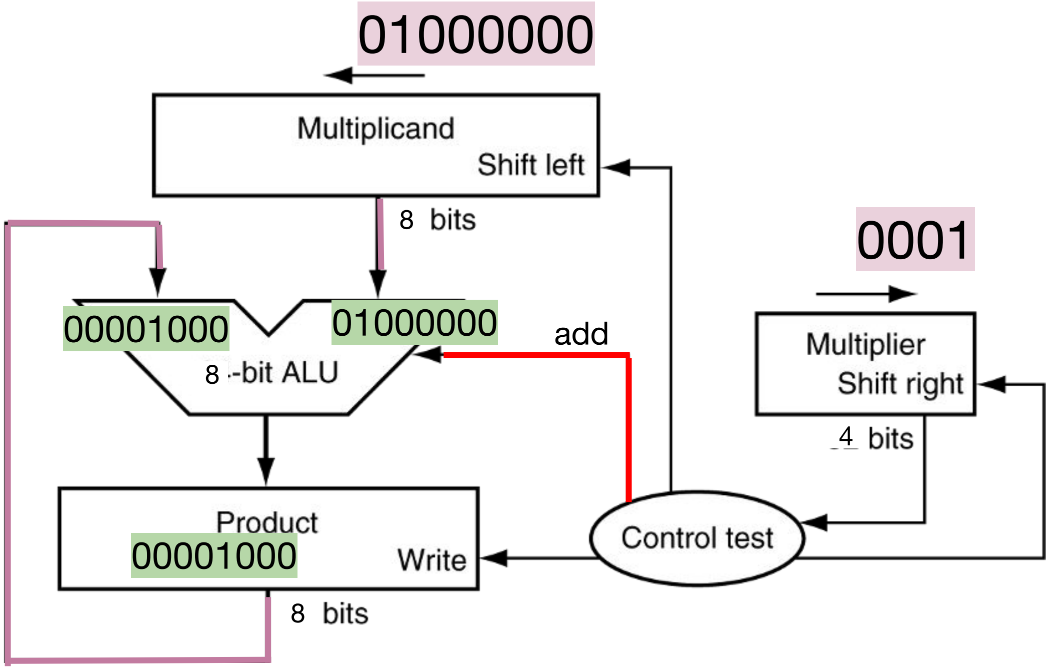


Example



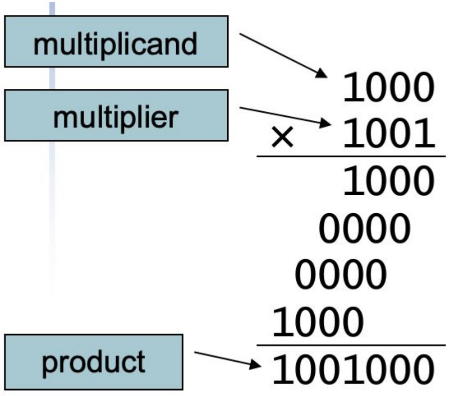
For simplification, we assume the circuits are designed to work on add two 4-bit numbers.

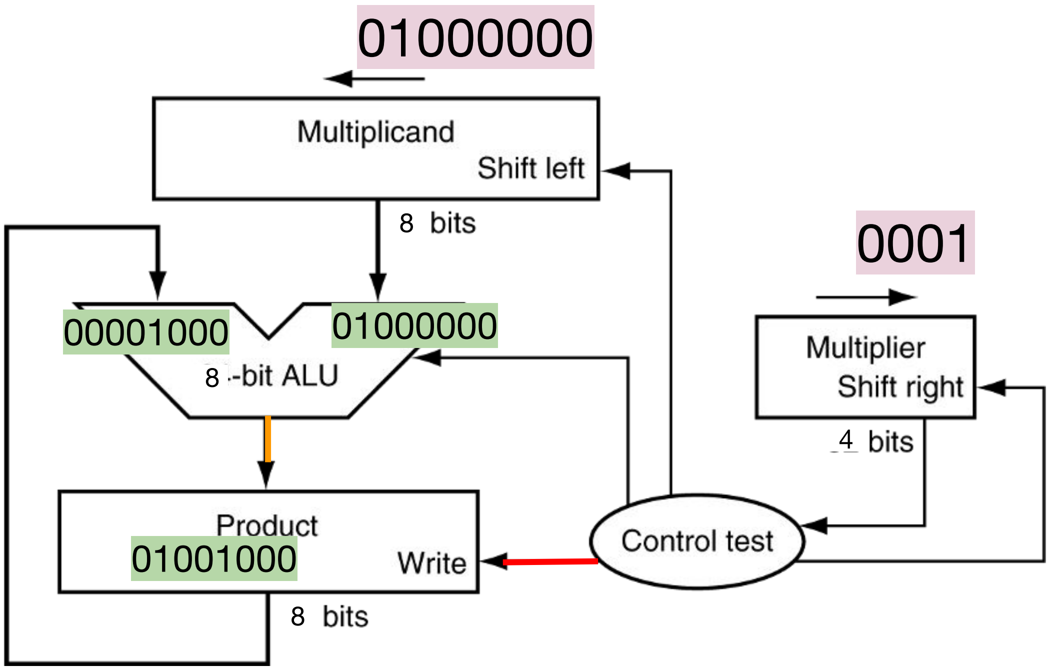
Iteration 4: b. Inform the ALU to add the inputs



Example

Iteration 4: c. Inform the Product register to write

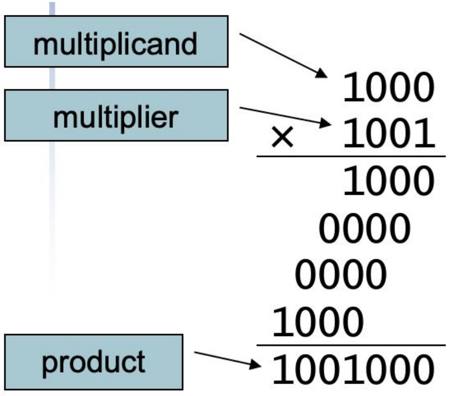


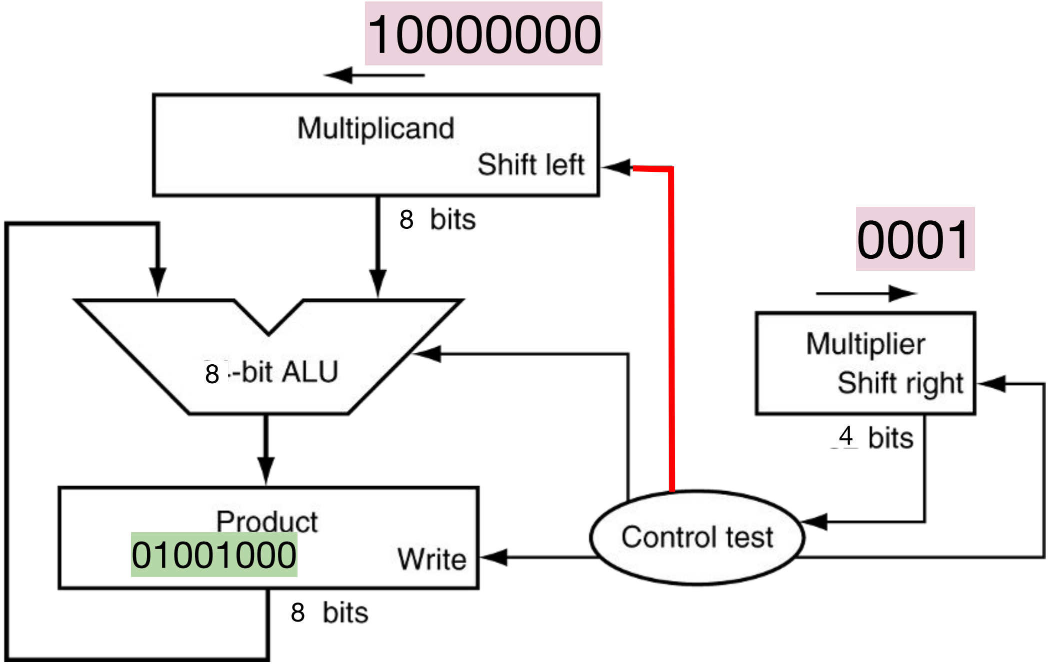


For simplification, we assume the circuits are designed to work on add two 4-bit numbers.

Example

Iteration 4: d. shift the multiplicand left by 1 bit





For simplification, we assume the circuits are designed to work on add two 4-bit numbers.

Example

multiplicand

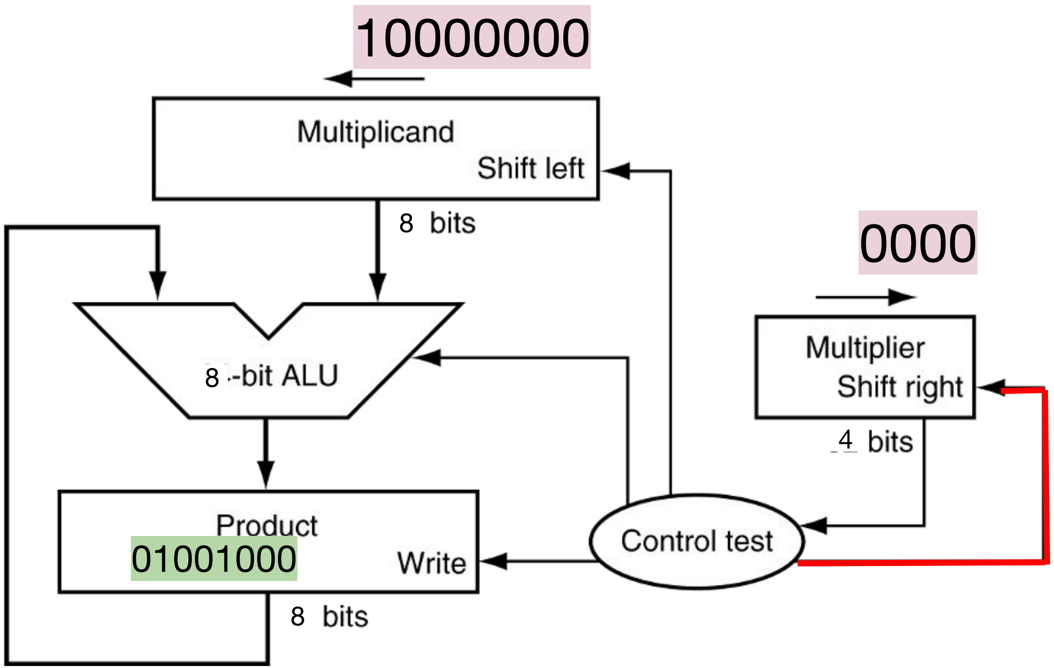
multiplier

product

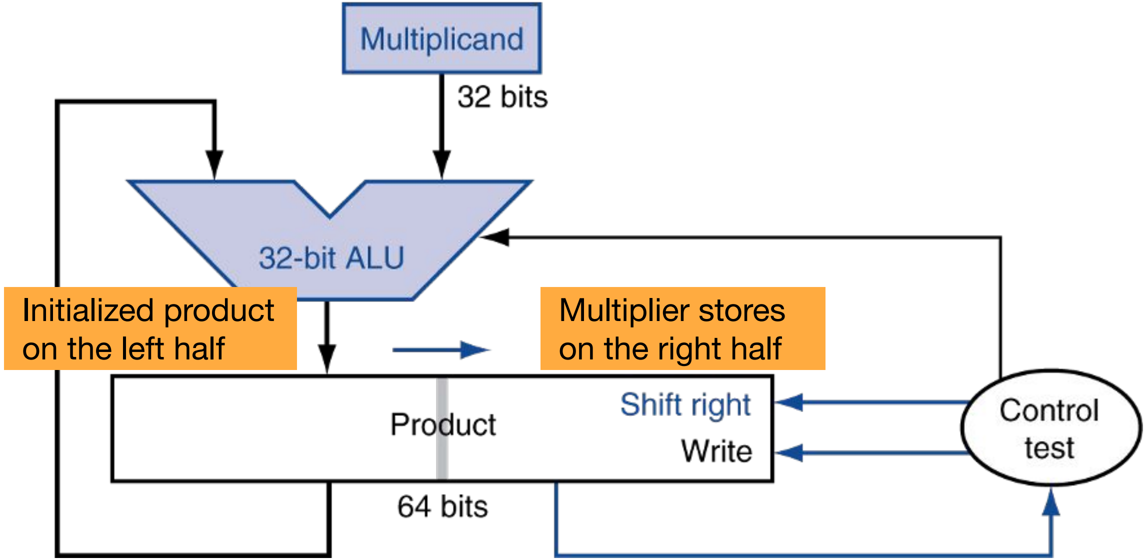
```
       1000
  ×    1001
  ---------
       1000
      0000
     0000
    1000
  ---------
    1001000
```

For simplification, we assume the circuits are designed to work on add two 4-bit numbers.

Iteration 4: e. shift the multiplier right by 1 bit



Optimized Multiplier



- Perform add/shift in parallel
- Only the product is 64 bits.
- The multiplier is placed at the right 32 bits in Product.
- When the Multiplicand is add to the Product, it will shift right.

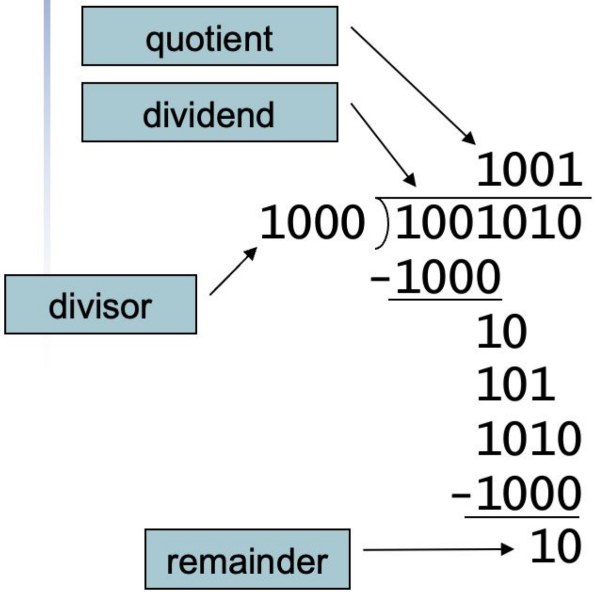
Signed Multiplication

- Keep the sign bit; leave it out of the calculation.
- Multiply the numbers in the same way as the multiplication of positive numbers.

MIPS Multiplication

- Two special 32-bit registers to handle cases of 64-bit 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`
    - It is a pseudoinstruction
    - It only retrieve the least-significant 32 bits (from the LO) of product to rd

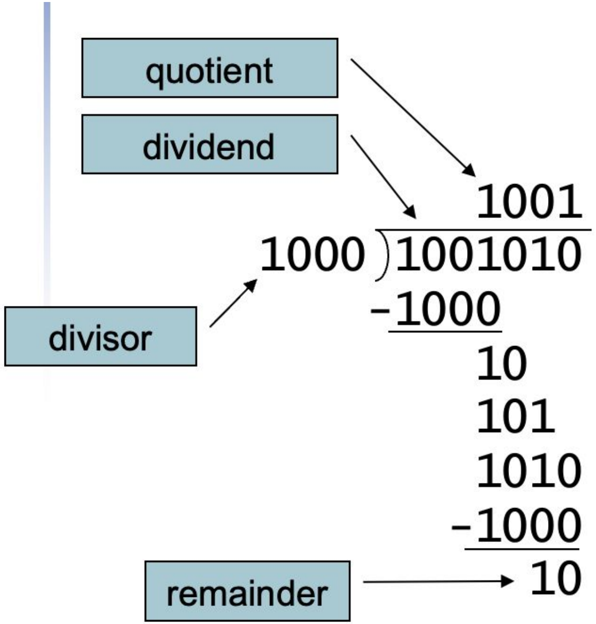
Division



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

- Firstly, check for 0 divisor
- Then, the long division approach:
  - If divisor <= dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Signed division
  - Divide using absolute values
  - Adjust sign of quotient and remainder as required

Implement the long-division approach



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

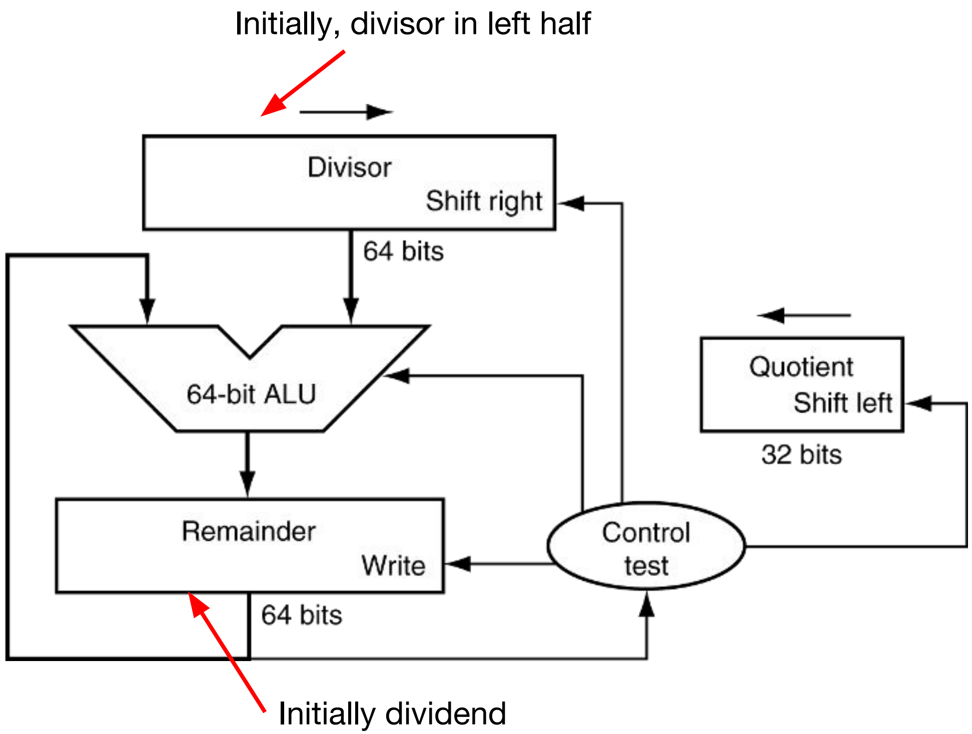
Comparison in computer is essentially done by subtraction. So,

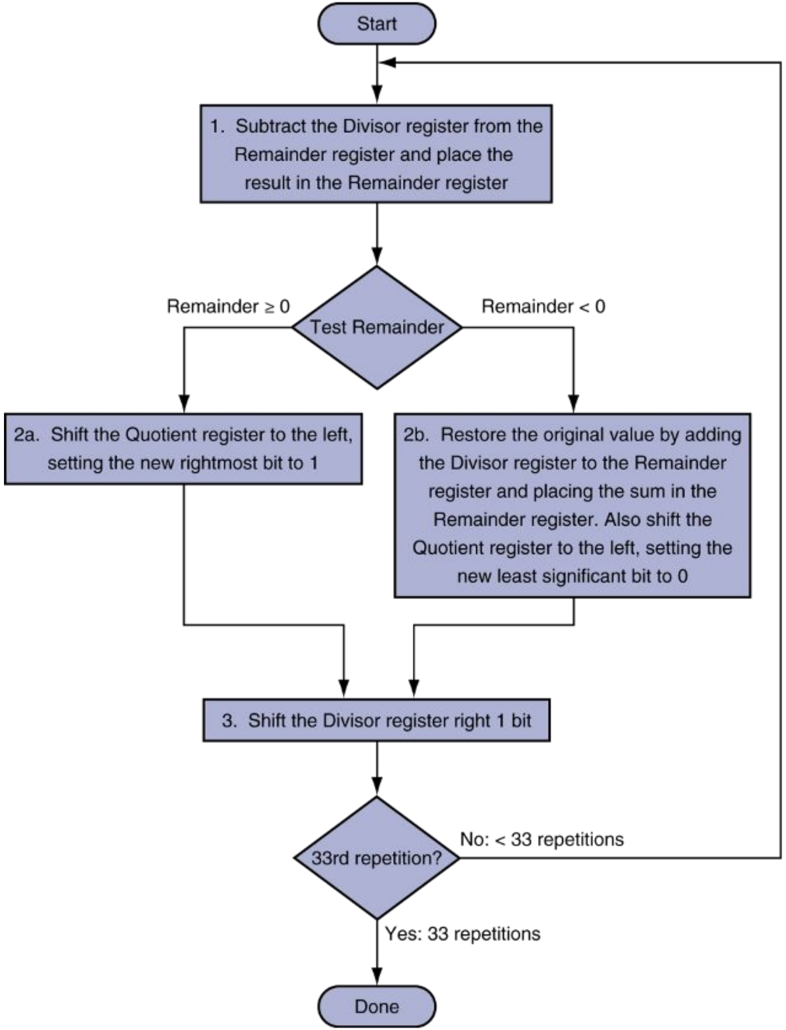
Step 1: “divisor <= dividend bits” ⇒ subtract the divisor from the dividend bits.

- If the remainder >=0, generate a 1 in quotient;
- If the remainder < 0, restore the dividen bit by adding the divisor back to the remainder and generate a 0 in the quotient.

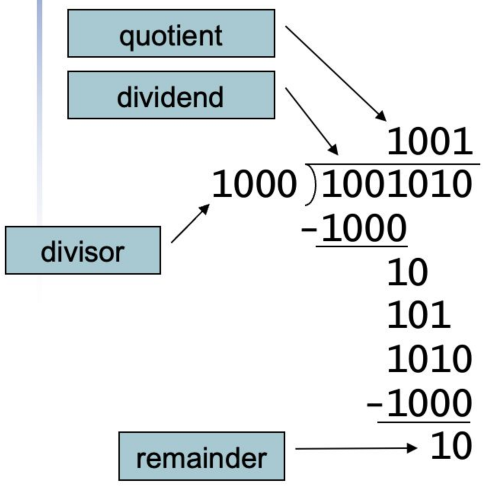
Step 2: shift the divisor right, then go back to **Step 1.**

Logic of division



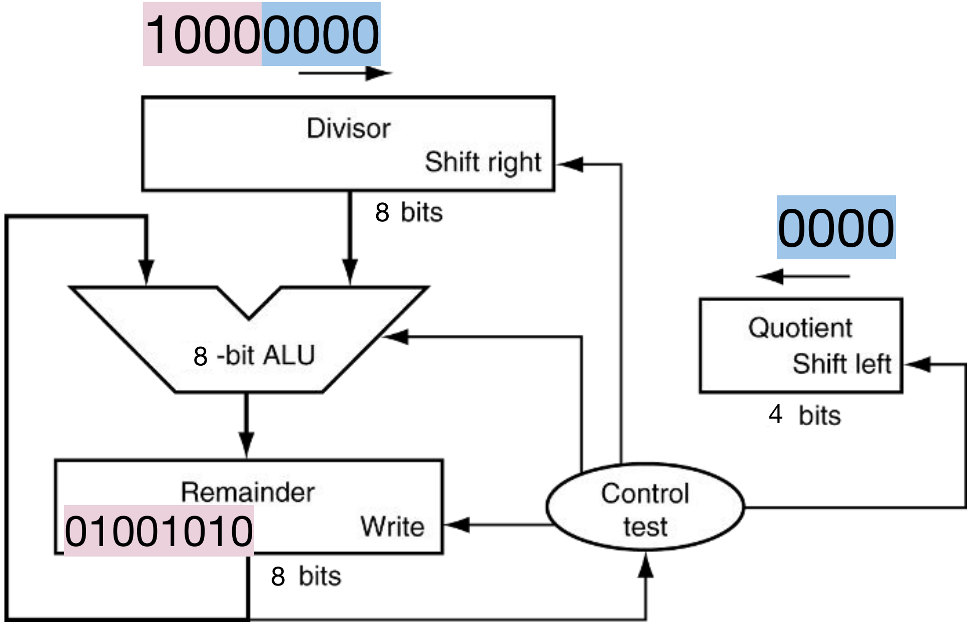


Example

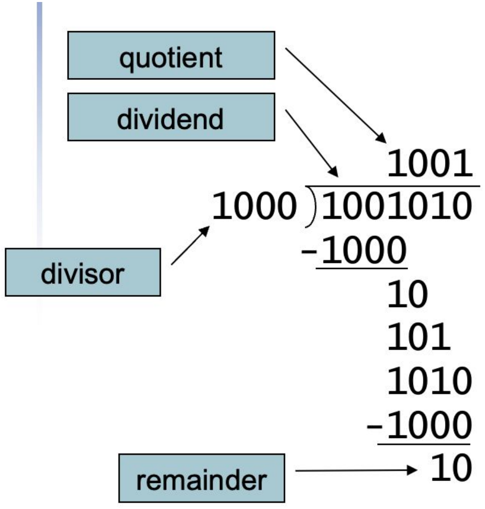


For simplification, we assume the circuits are designed to work on add two 4-bit numbers.

Iteration 0: initializing the registers.

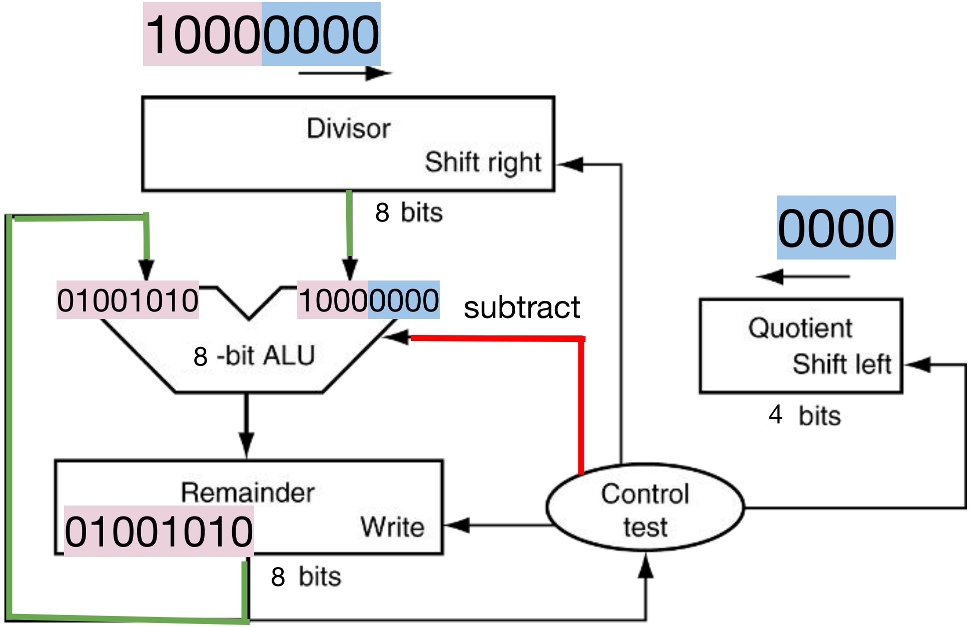


Example

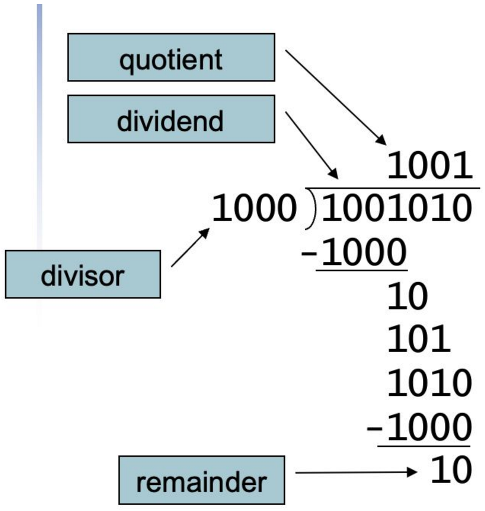


For simplification, we assume the circuits are designed to work on add two 4-bit numbers.

Iteration 1: a. Remainder - Divisor

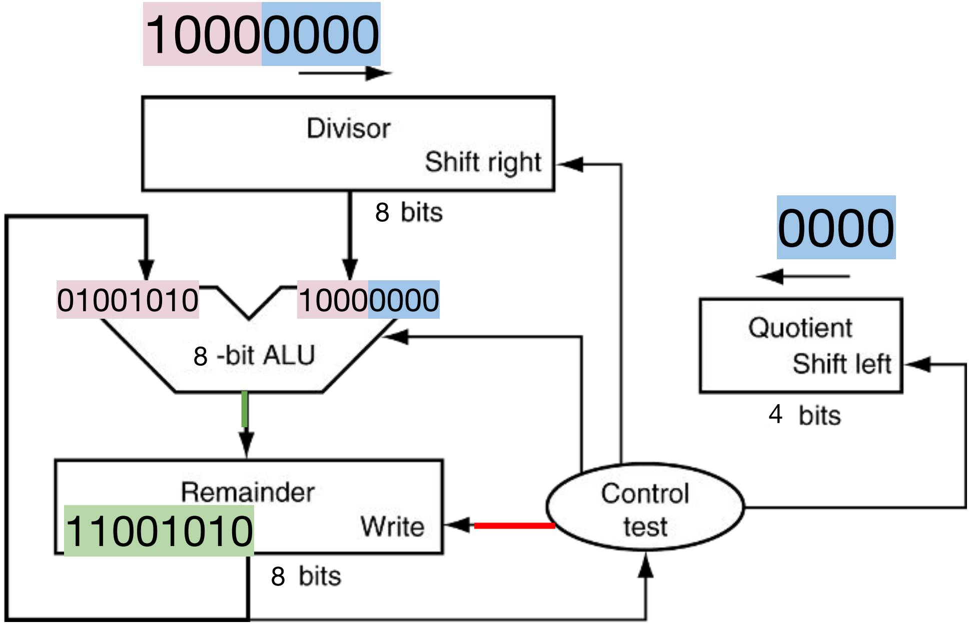


Example

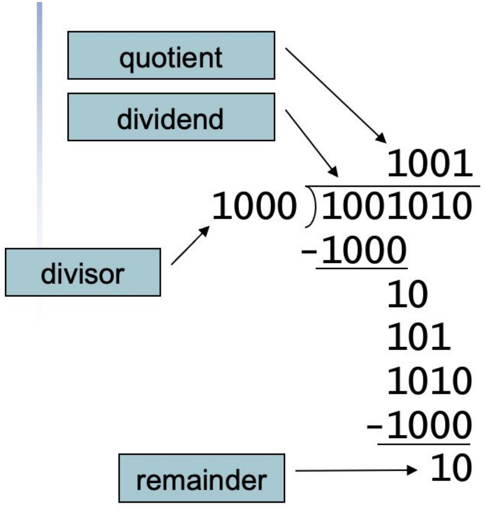


For simplification, we assume the circuits are designed to work on add two 4-bit numbers.

Iteration 1: b. Update the remainder

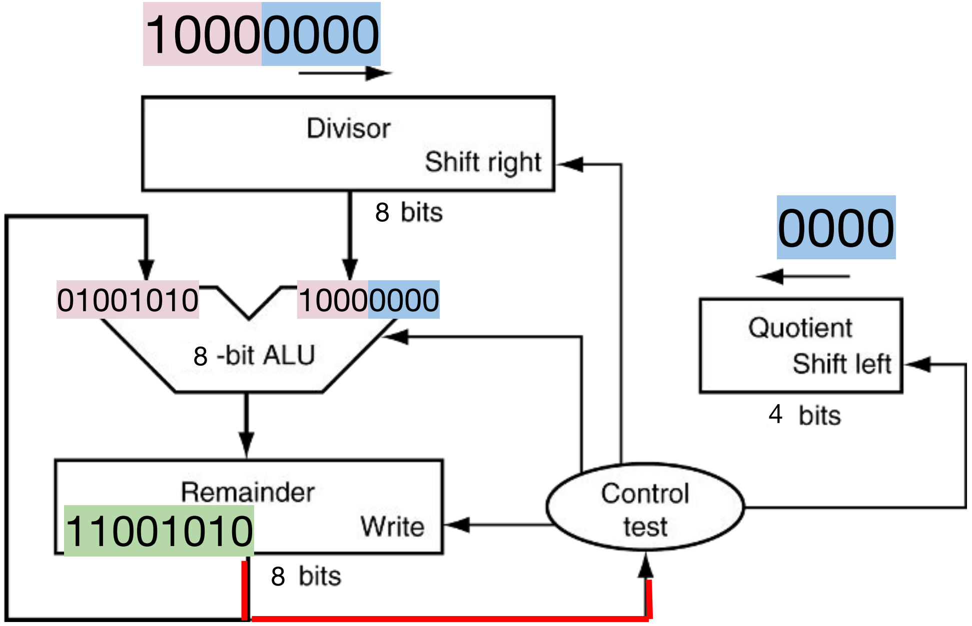


Example

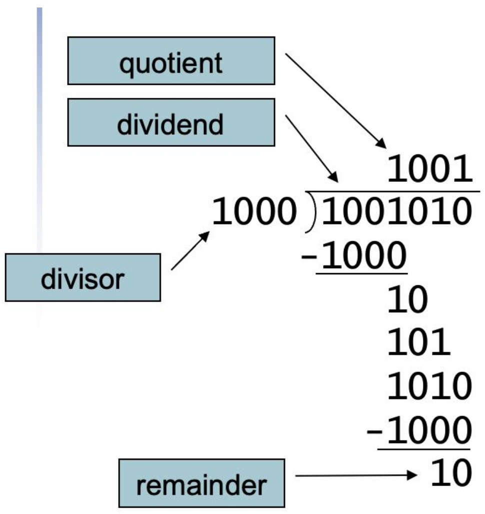


For simplification, we assume the circuits are designed to work on add two 4-bit numbers.

Iteration 1: c. Remainder < 0

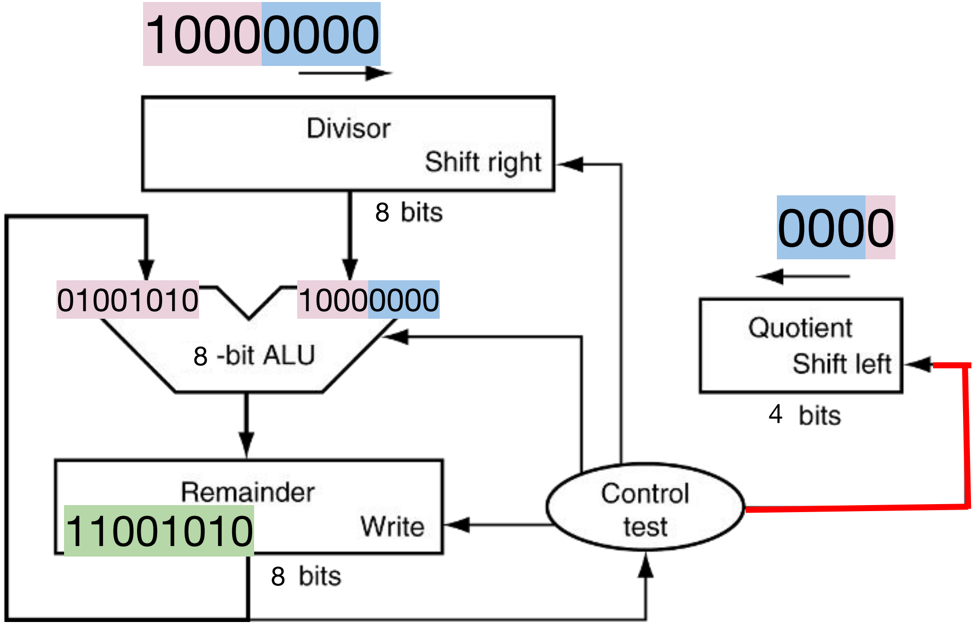


Example



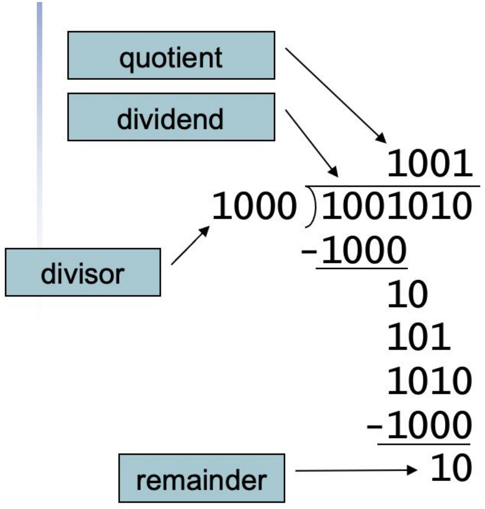
For simplification, we assume the circuits are designed to work on add two 4-bit numbers.

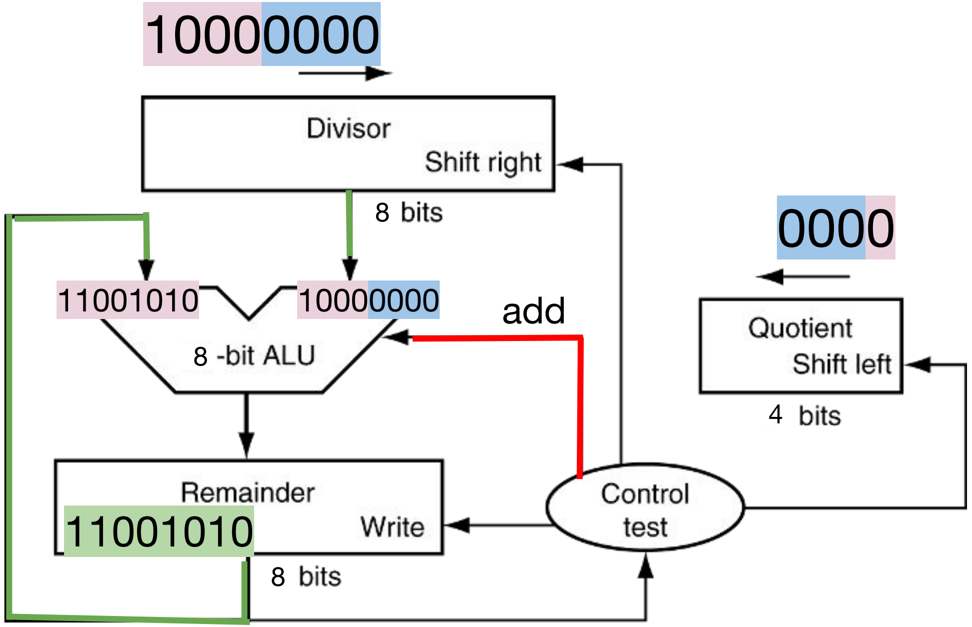
Iteration 1: d. add 0 to Quotient



Example

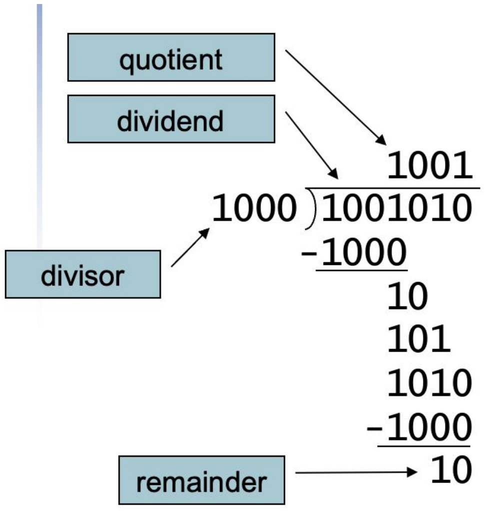
Iteration 1: e. Add divisor back to remainder





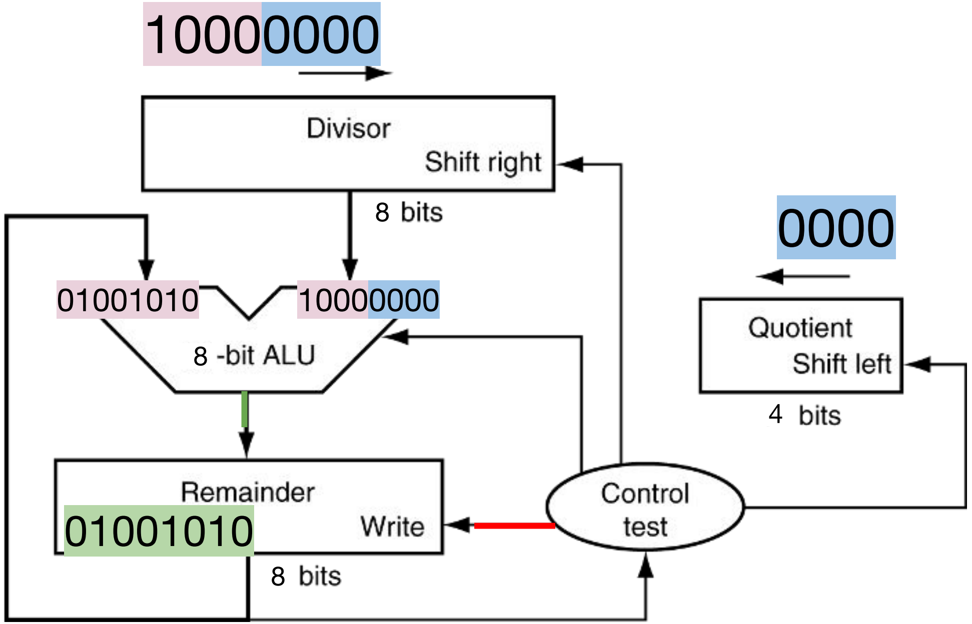
For simplification, we assume the circuits are designed to work on add two 4-bit numbers.

Example

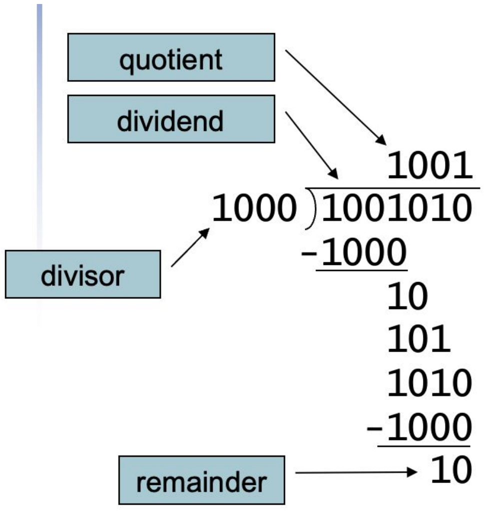


For simplification, we assume the circuits are designed to work on add two 4-bit numbers.

Iteration 1: f. Update the remainder

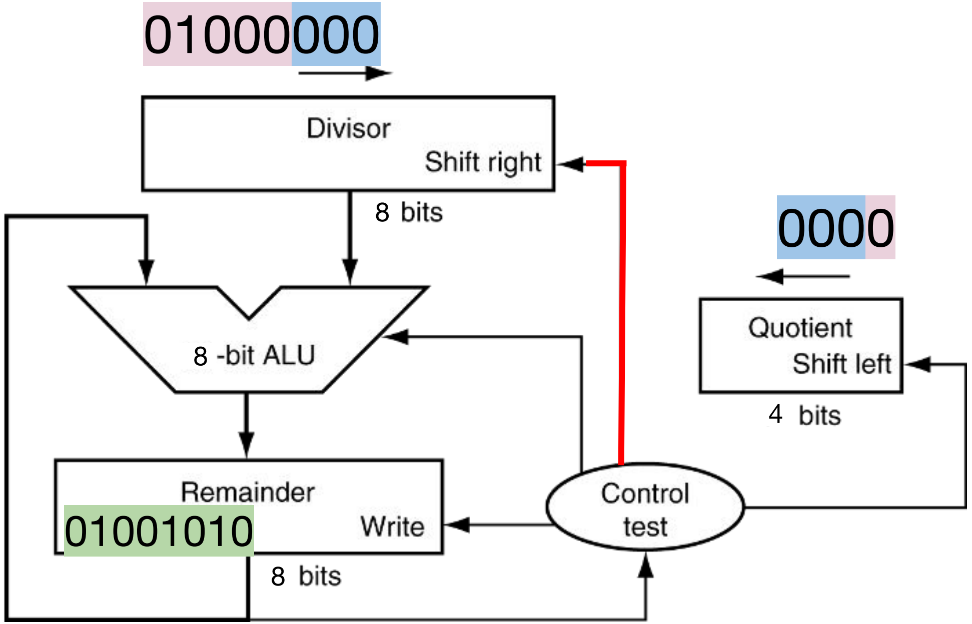


Example

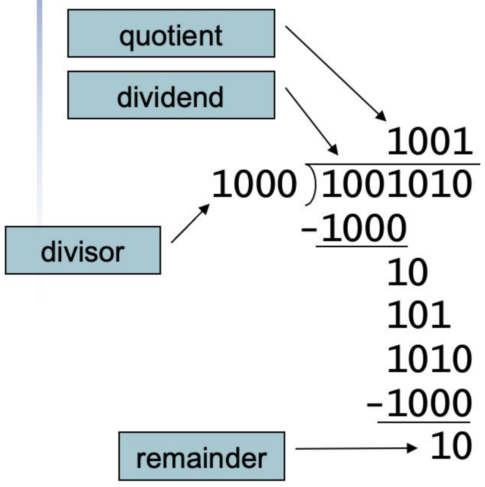


For simplification, we assume the circuits are designed to work on add two 4-bit numbers.

Iteration 1: g. Shift divisor right for a bit

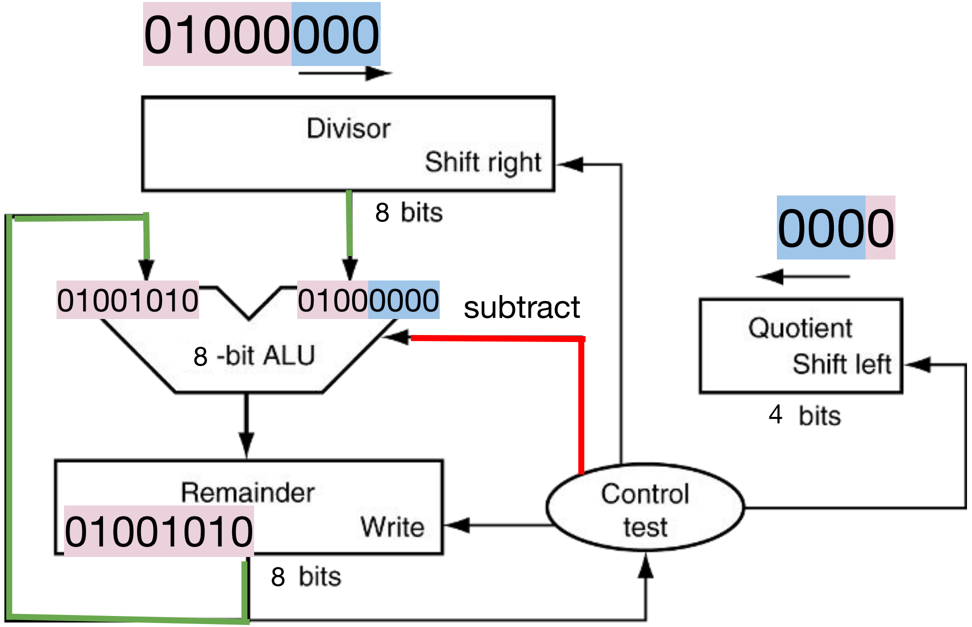


Example

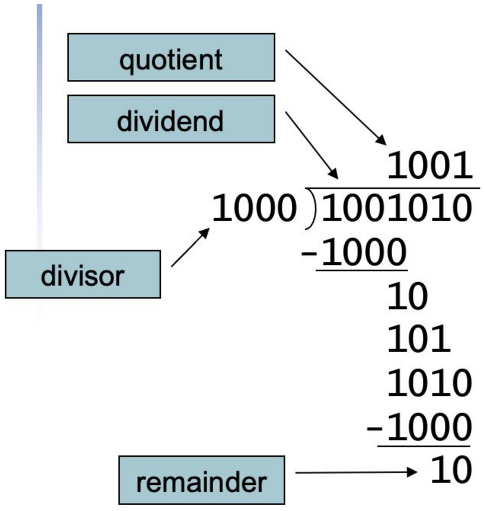


For simplification, we assume the circuits are designed to work on add two 4-bit numbers.

Iteration 2: a. Remainder - Divisor

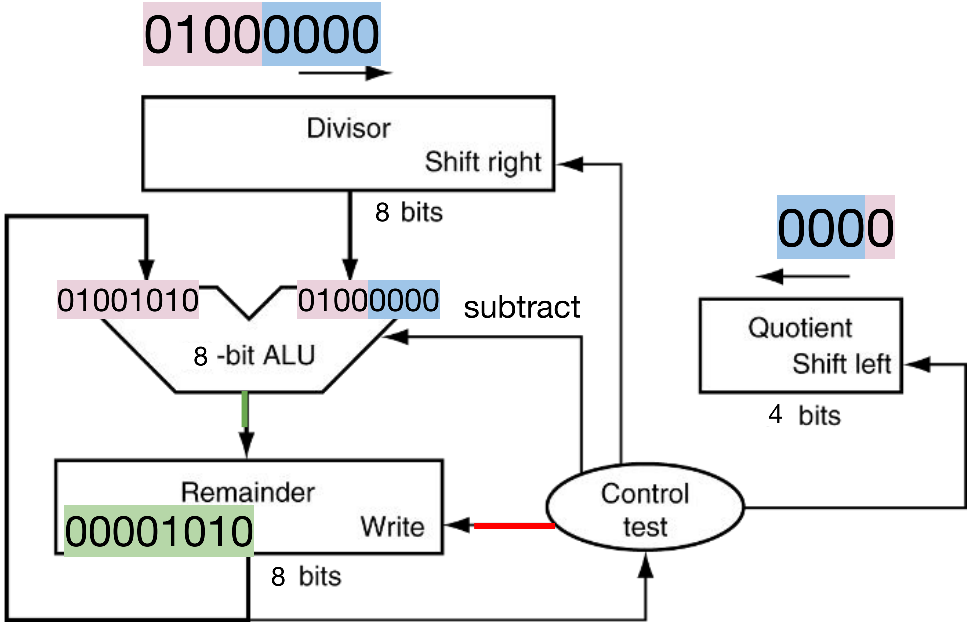


Example

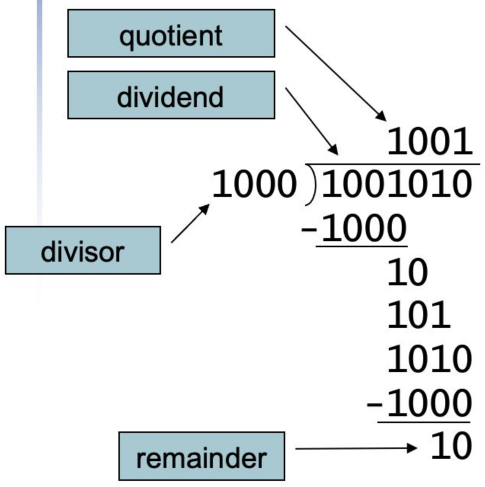


For simplification, we assume the circuits are designed to work on add two 4-bit numbers.

Iteration 2: b. Update the remainder

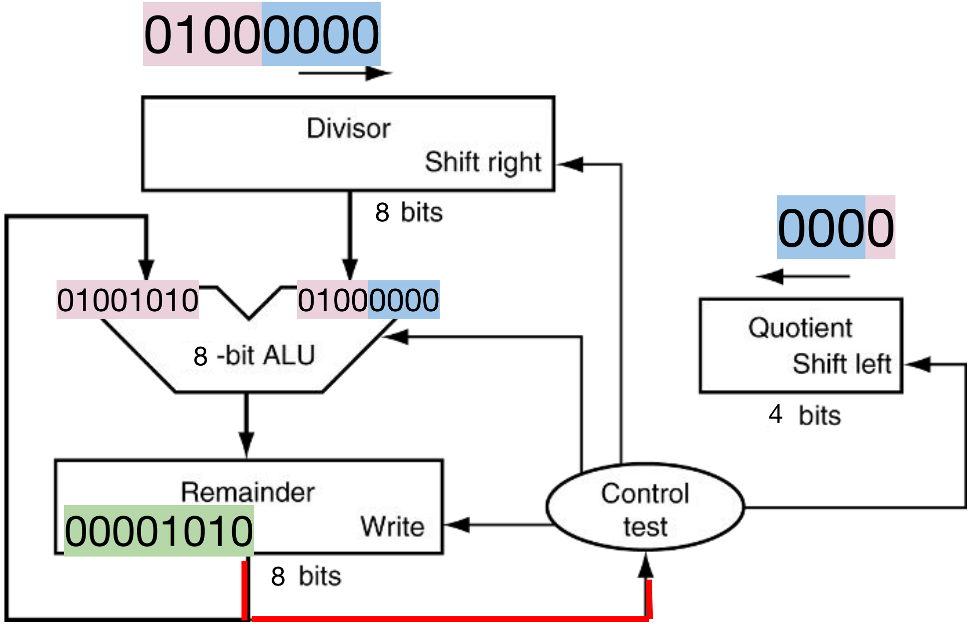


Example

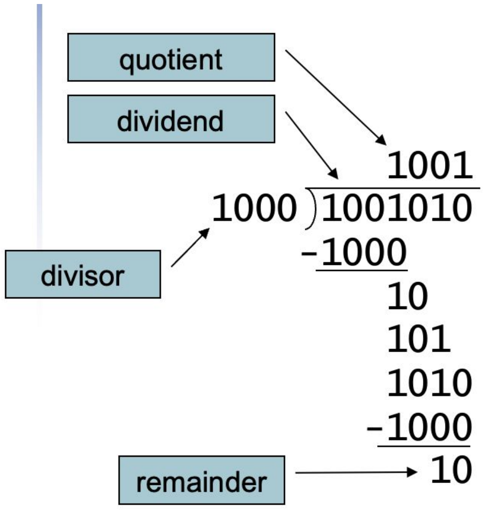


For simplification, we assume the circuits are designed to work on add two 4-bit numbers.

Iteration 2: c. Remainder > 0

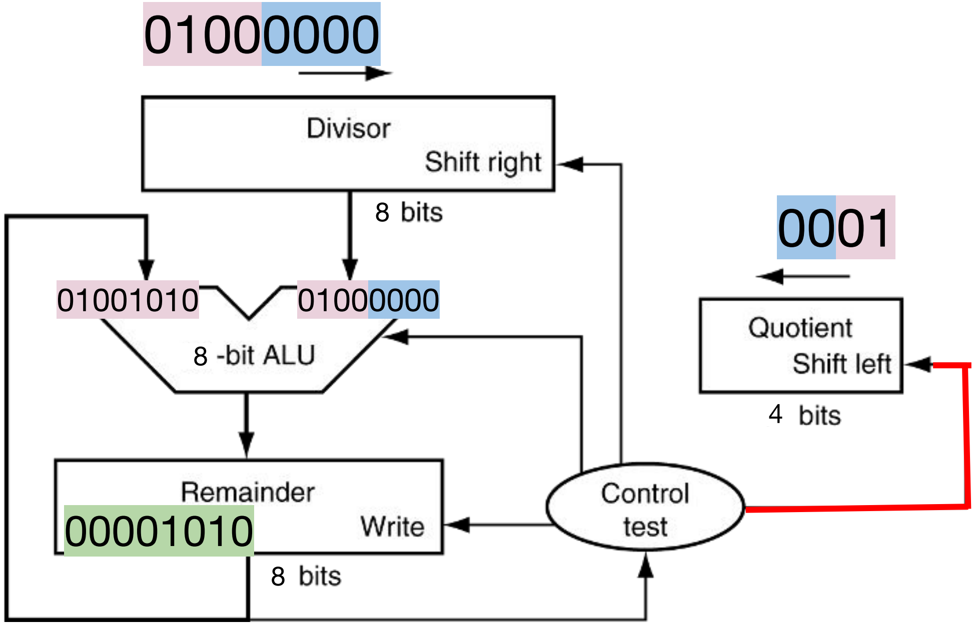


Example

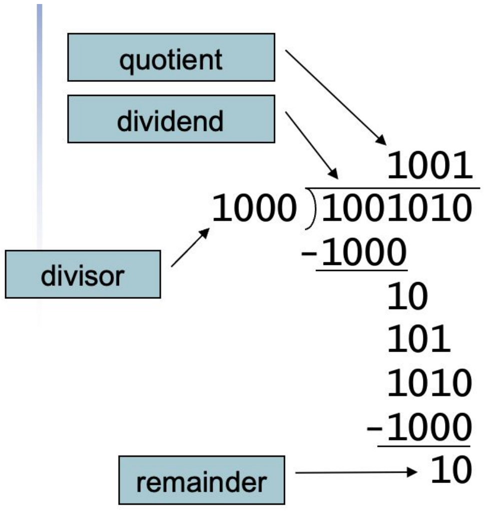


For simplification, we assume the circuits are designed to work on add two 4-bit numbers.

Iteration 2: d. add 1 to Quotient

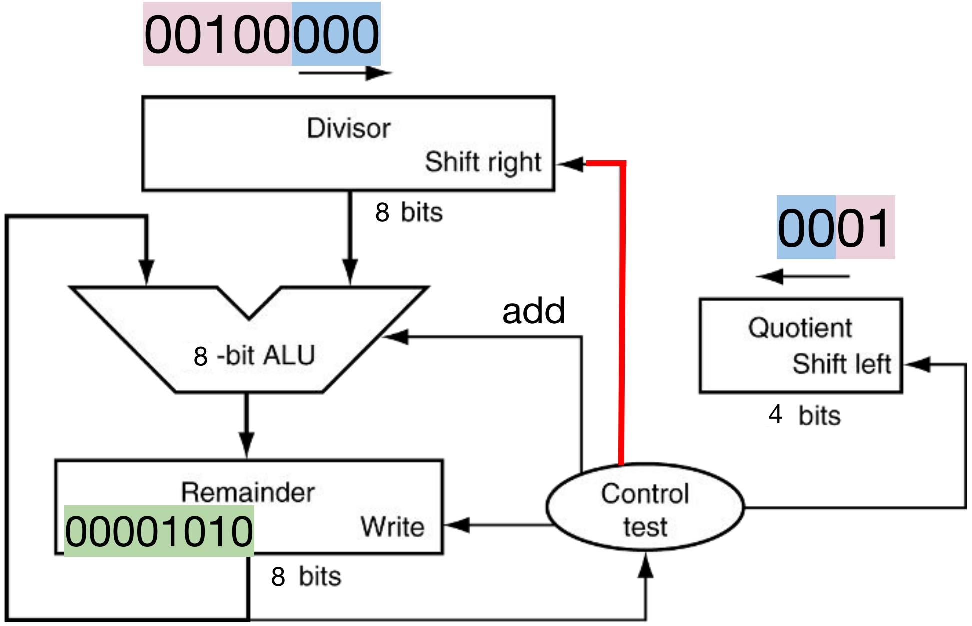


Example

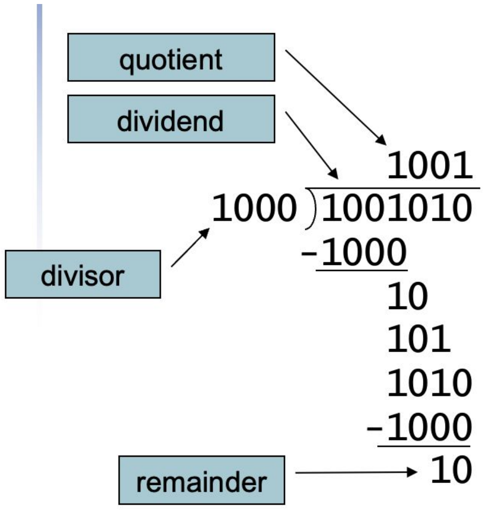


For simplification, we assume the circuits are designed to work on add two 4-bit numbers.

Iteration 2: e. Shift divisor right for a bit

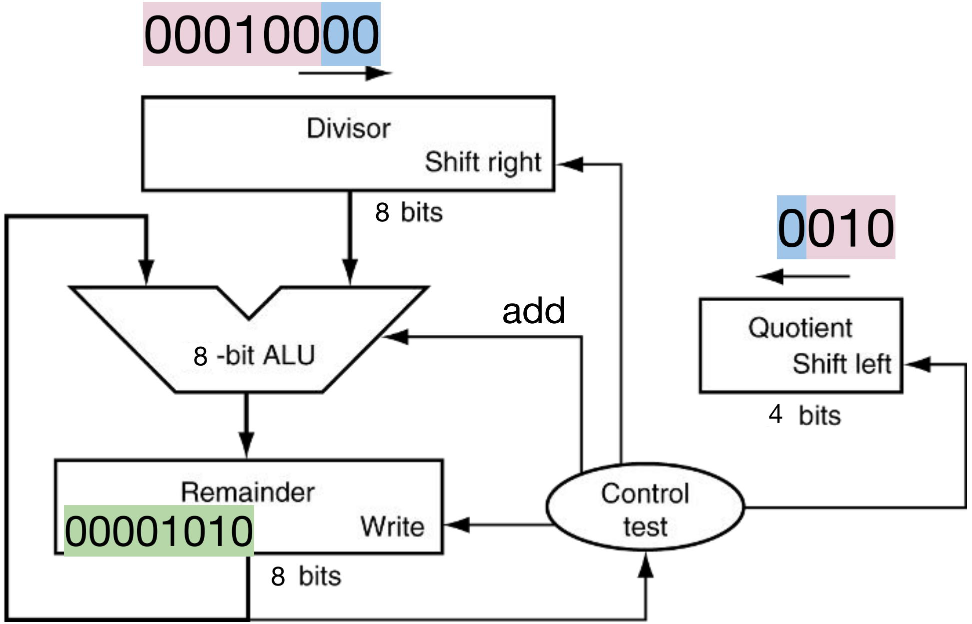


Example

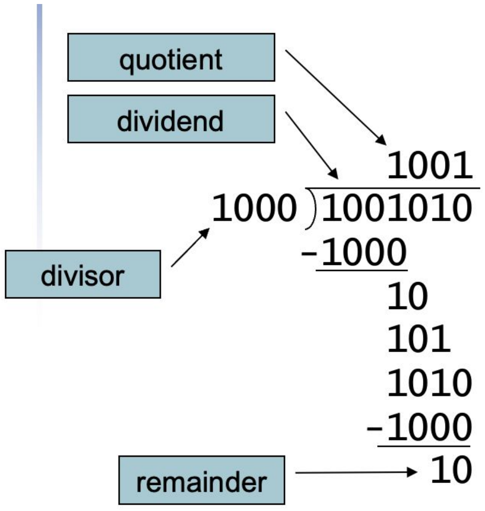


For simplification, we assume the circuits are designed to work on add two 4-bit numbers.

Iteration 3: similar to Iteration 1

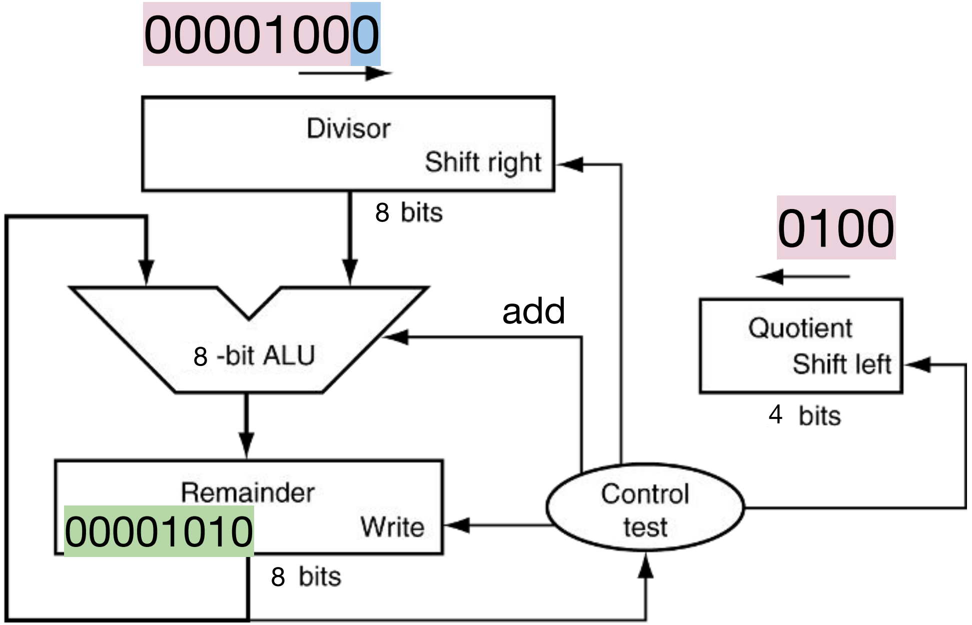


Example

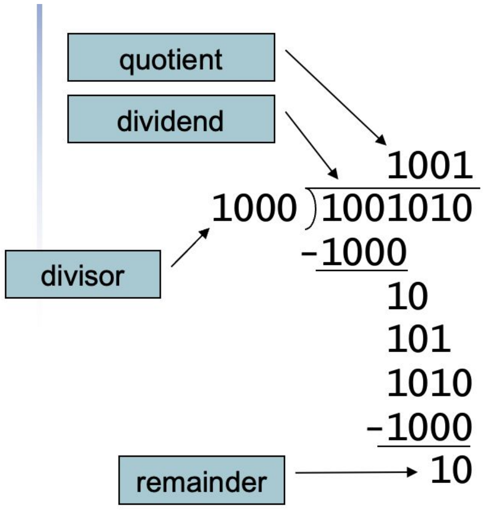


For simplification, we assume the circuits are designed to work on add two 4-bit numbers.

Iteration 4: similar to Iteration 1

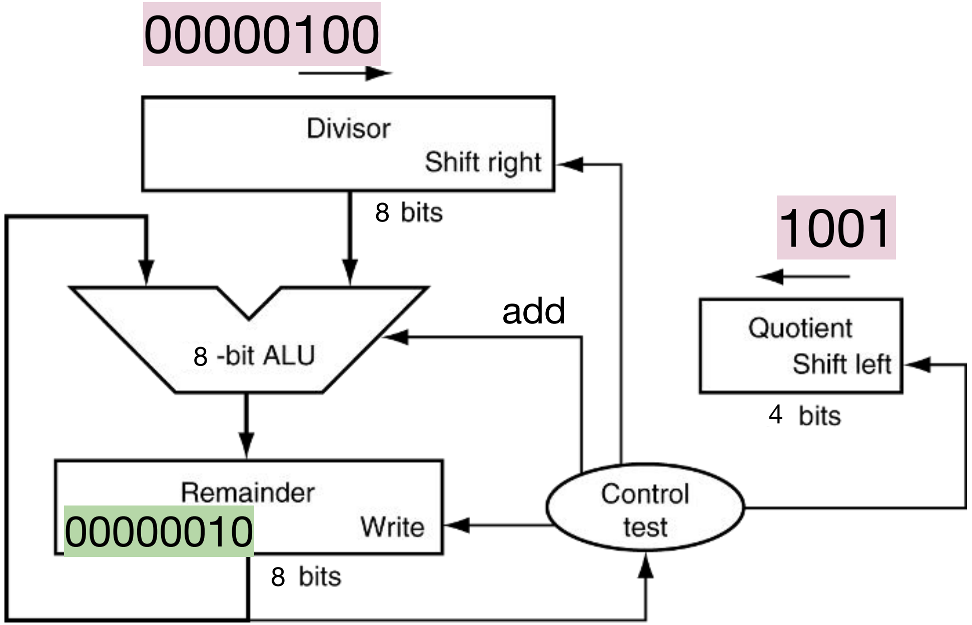


Example

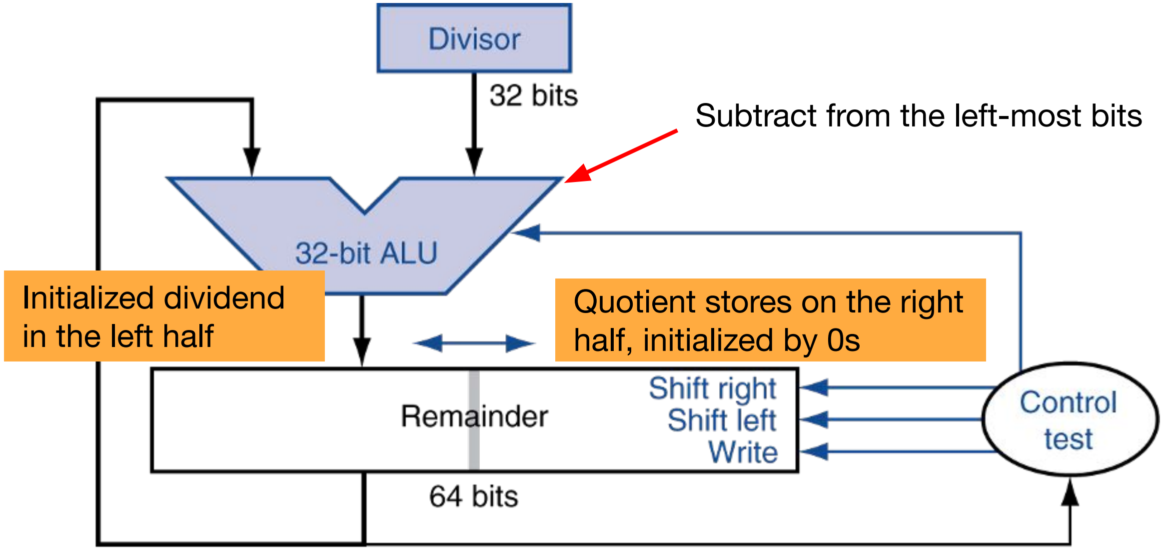


For simplification, we assume the circuits are designed to work on add two 4-bit numbers.

Iteration 5: similar to Iteration 2



Optimized Divider



- The left 32 bits in the remainder is initialized as the dividend, the right 32 bits is initialized by 0.
- When reminder subtract divisor >0, the remainder shift left one bit and the new shift bit is 1. Otherwise, add the divisor back to the remainder, and set the quotation bit to 0.

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

Signed Division

- We can remember the sign of the divisor and dividend then negate the quotient if the sign disagree.

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

Summary

- MIPS instructions for arithmetic operations

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow detected
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow detected
	add immediate	addi $s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow detected
	add unsigned	addu $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow undetected
	subtract unsigned	subu $s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow undetected
	add immediate unsigned	addiu $s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow undetected
	move from coprocessor register	mfc0 $s1,$epc	$s1 = $epc	Copy Exception PC + special regs
	multiply	mult $s2,$s3	Hi, Lo = $s2 × $s3	64-bit signed product in Hi, Lo
	multiply unsigned	multu $s2,$s3	Hi, Lo = $s2 × $s3	64-bit unsigned product in Hi, Lo
	divide	div $s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Lo = quotient, Hi = remainder
	divide unsigned	divu $s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Unsigned quotient and remainder
	move from Hi	mfhi $s1	$s1 = Hi	Used to get copy of Hi
	move from Lo	mflo $s1	$s1 = Lo	Used to get copy of Lo