CHAPTER 3-1

Arithmetic for Computers

By Pattama Longani
Collage of arts, media and Technology

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

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

		(0)		*(0)*		*(1)*		*(1)*		*(0)*		*(Carries)*
. . .		0		0		0		1		1		1
. . .		0		0		0		1		1		0
. . .	(0)	0	(0)	0	(0)	1	(1)	1	(1)	0	(0)	1

INTEGER SUBTRACTION

EXAMPLE

7 - 6

Subtracting 6_{ten} from 7_{ten} can be done directly:

$$
\begin{array}{llcl}
 & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} & = & 7_{ten} \\
- & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{two} & = & 6_{ten} \\
\hline
= & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} & = & 1_{ten}
\end{array}
$$

or via addition using the two's complement representation of –6:

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

OVERFLOW

if the result is out of range

Adding +v and –v	operands no overflow
Adding two +v operands	Overflow if the sign bit of its result is equal to 1
Adding two -v operands	Overflow if the sign bit of its result is equal to 0

ADDITION-SUBTRACTION

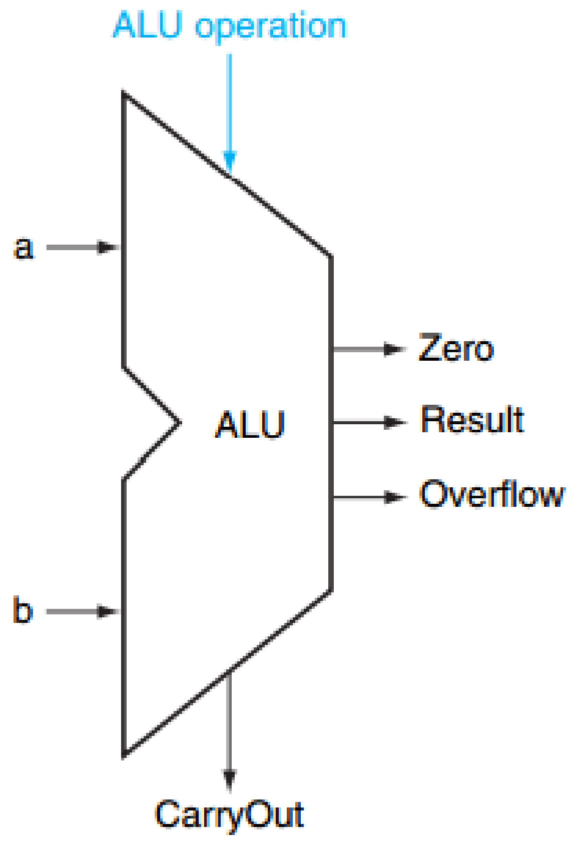
Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

DEALING WITH OVERFLOW

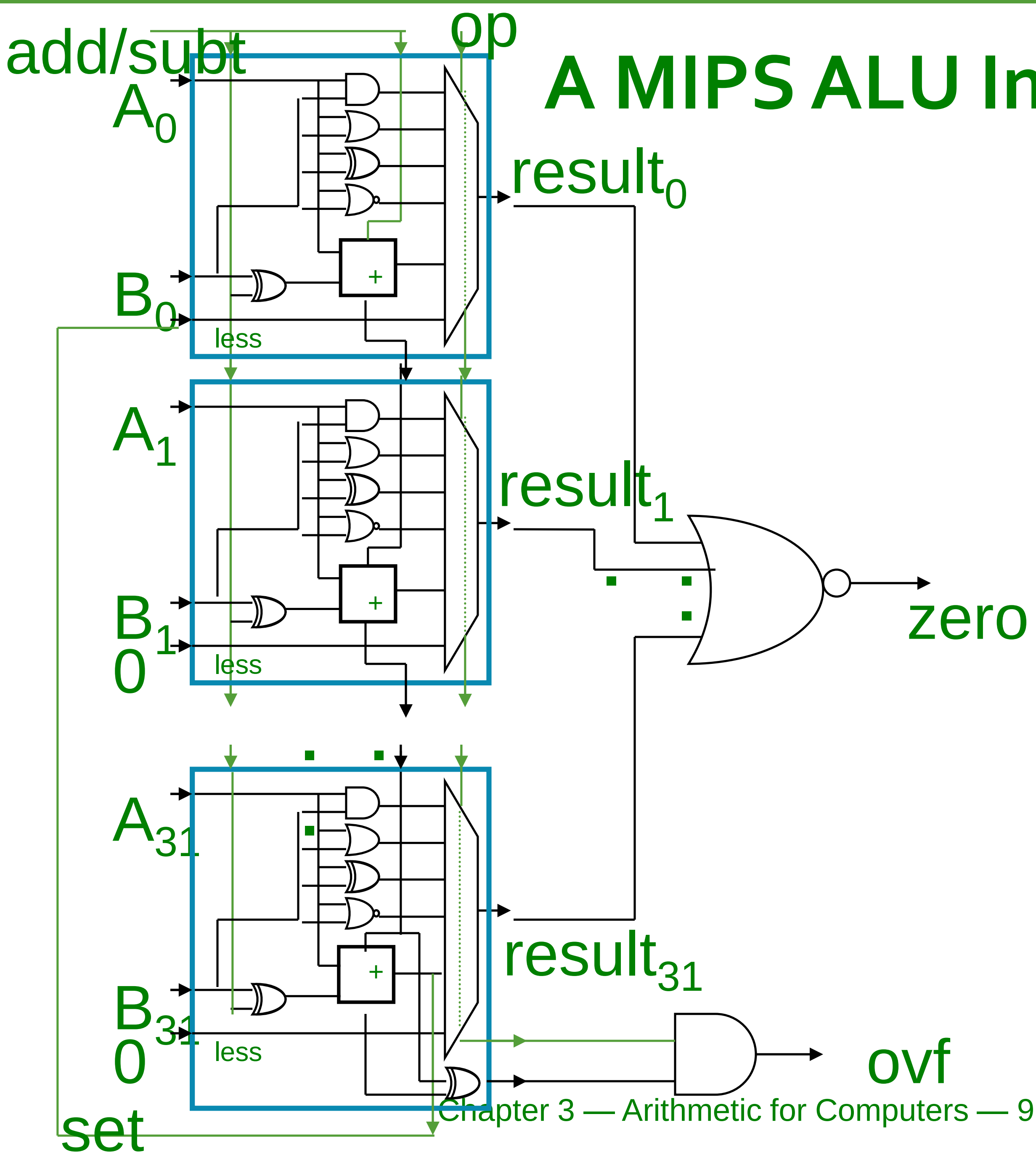
- Some languages (e.g., C) **ignore overflow**
- Other languages (e.g., Ada, Fortran) **require raising an exception**

ARITHMETIC LOGIC UNIT (ALU)

- The device that performs
  - **arithmetic operations**
    - addition
    - subtraction
  - **logical operation**
    - AND
    - OR



A MIPS ALU Implementation



MULTIPLICATION

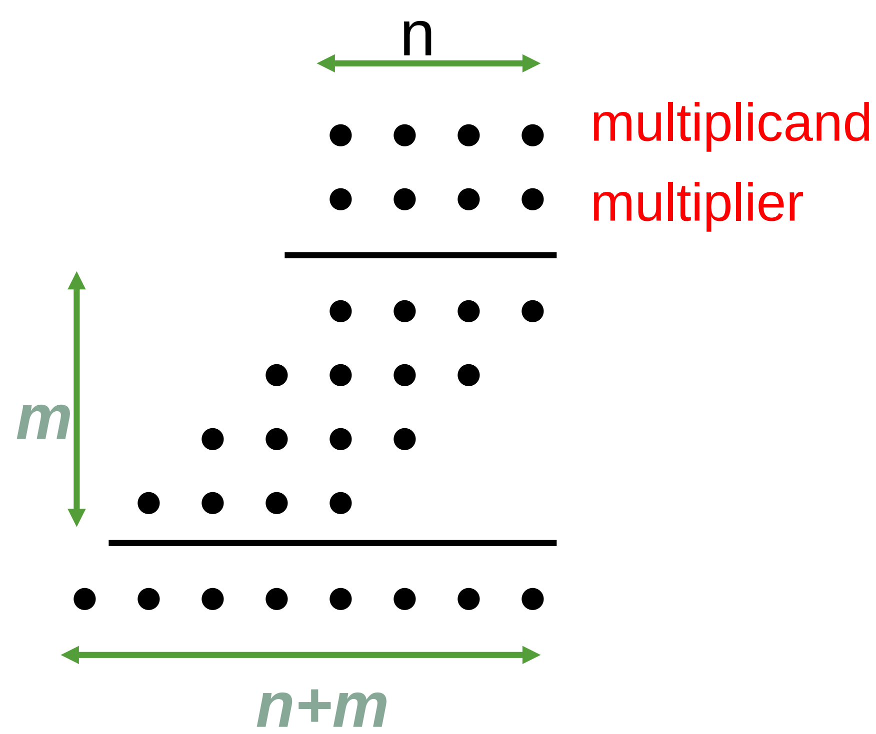
multiplicand

multiplier

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

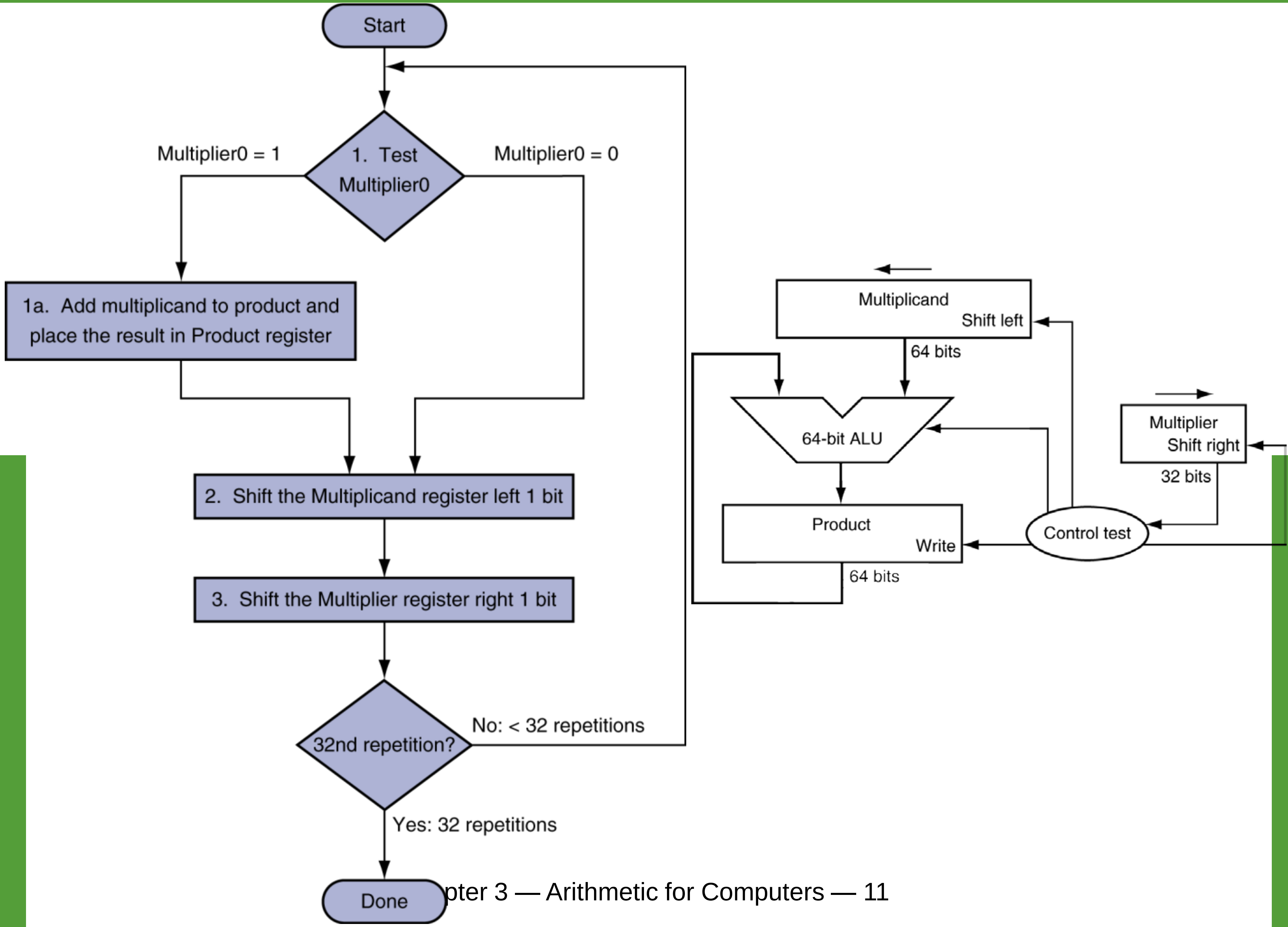
product

Length of product is the sum of operand lengths



- ***First observation:***

  length of the multiplication = *n-bit multiplicand* + *m-bit multiplier*
- SO, Multiplication of 2 numbers of 32-bit MIPS results as 64-bit products


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
Multiplicand
Shift left
64 bits
64-bit ALU
Product
Write
64 bits
Multiplier
Shift right
32 bits
Control test

MULTIPLY $2_{ten} \times 3_{ten}$ ($0010_{two} \times 0011_{two}$)

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 => Prod = Prod + Mcand			0000 0010
	2: Shift left Mcand		0000 0100	
	3: Shift right Multiplier	0001		
2	1a: 1 => Prod = Prod + Mcand			0000 0110
	2: Shift left Mcand		0000 1000	
	3: Shift right Multiplier	0000		
3	1: 0 => No add operation			
	2: Shift left Mcand		0001 0000	
	3: Shift right Multiplier	0000		
4	1: 0 => No add operation			
	2: Shift left Mcand		0001 0000	
	3: Shift right Multiplier	0000		
	Total step: 12		**Final Product:**	0000 0110

SIGNED MULTIPLICATION

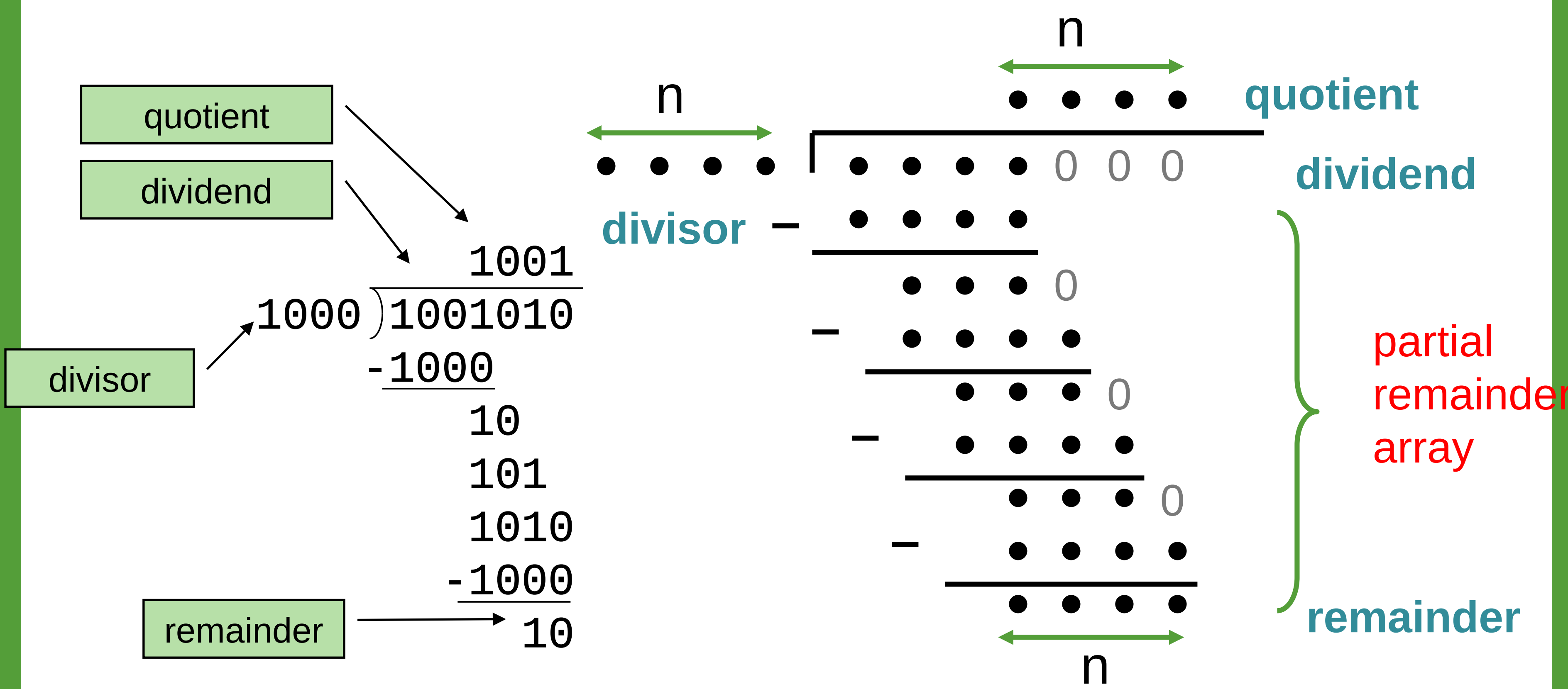
- First:
  - Convert the multiplier and multiplicand to positive numbers
  - **<u>Save the original signs (will be checked later)</u>**
- Then:
  - Calculate positive numbers
  - leaving the signs out of the calculation
- Finally:
  - if the original signs are disagree,
  - Convert result product to negative number

Multiply $5_{ten} \times -4_{ten}$

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0100	0000 0101	0000 0000
1	1: 0 : no operation			0000 0000
	2: Shift left Mcand		0000 1010	
	3: Shift right Multiplier	0010		
2	1: 0 : no operation			0000 0000
	2: Shift left Mcand		0001 0100	
	3: Shift right Multiplier	0001		
3	1: 1: Mcand + Product			0001 0100
	2: Shift left Mcand		0010 1000	
	3: Shift right Multiplier	0000		
4	1:			0001 0100
	2: Shift left Mcand		0101 0000	
	3: Shift right Multiplier	0000		
	Sign: negate => convert			1110 1100
	Total step:		**Final Product:**	1110 1100

DIVISION

dividend = quotient x divisor + remainder

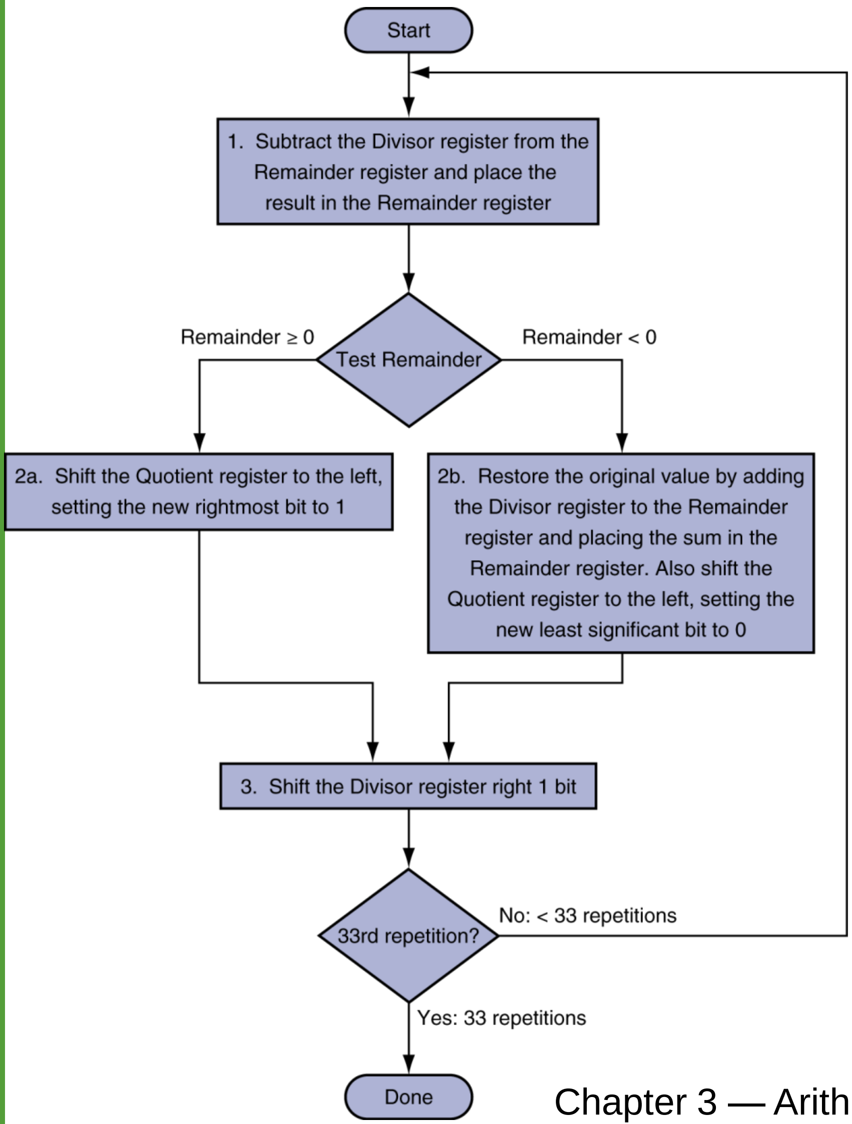


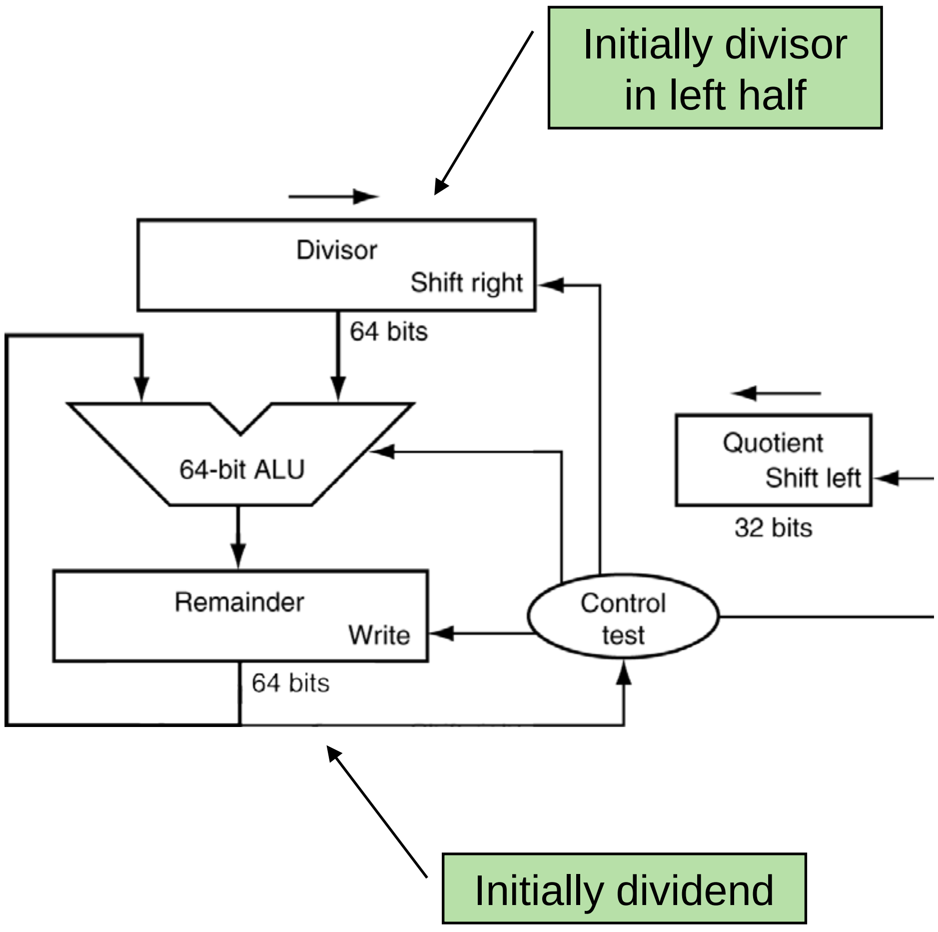
DIVISION

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

Division Hardware





Divide of 74=01001010 and 4=0100

Iteration	Step	Divisor	Remainder	Quotient
0	Initial values	0100 0000	0100 1010	00000
1	1: Sub: Remainder - Divisor		0000 1010	
	2a: Remainder >=0 : Shift left			00001
	3: Shift right Divisor	0010 0000		
2	1: Sub: Remainder - Divisor		1110 1010	
	2b: Remainder < 0 : Restore		0000 1010	00010
	3: Shift right Divisor	0001 0000		
3	1: Sub: Remainder - Divisor		1111 0000	
	2b: Remainder < 0 : Restore		0000 1010	00100
	3: Shift right Divisor	0000 1000		
4	1: Sub: Remainder - Divisor		0000 0010	
	2a: Remainder >=0 : Shift left			01001
	3: Shift right Divisor	0000 0100		
5	1: Sub: Remainder - Divisor		1111 1110	
	2b: Remainder < 0 : Restore		0000 0010	10010
	3: Shift right Divisor	0000 0010		
	Total step: 15	**Final Quotient: 10010 Remainder: 0010**		

EXAMPLE: divide of 6 and 2

Iteration	Step	Divisor	Remainder	Quotient
0	Initial values	0010 0000	0000 0110	0000
1	1: Sub: Remainder - Divisor		1110 0110	
	2: Remainder < 0		0000 0110	0000
	3: Shift right Divisor	0001 0000		
2	1: Sub: Remainder - Divisor		1111 0110	
	2: Remainder < 0		0000 0110	0000
	3: Shift right Divisor	0000 1000		
3	1: Sub: Remainder - Divisor		1111 1110	
	2: Remainder < 0		0000 0110	0000
	3: Shift right Divisor	0000 0100		
4	1: Sub: Remainder - Divisor		0000 0010	
	2: Remainder >= 0			0001
	3: Shift right Divisor	0000 0010		
5	1: Sub: Remainder - Divisor		0000 0000	
	2: Remainder >= 0			0011
	3: Shift right Divisor	0000 0001		
	Total step: 15	**Final Quotient:0011 Remainder:0000**		