Computer Architecture and Organization
Computer Arithmetic
(CSE 2001)

Dr. R. Bhargavi &
Dr. A.K. Ilavarasi
SCSE

Data Representation

- Data types found in the registers/Memory of digital computers
  - ***Numbers*** used in arithmetic computations
  - ***Letters*** of the alphabet used in data processing
  - ***Other discrete symbols*** used for specific purpose

Number Systems

- ***Base*** or ***Radix r system*** : uses distinct symbols for ***r digits***
- A number system of radix r uses a string consisting of r distinct symbols to represent a value.
- Most common number system : Decimal, Binary, Octal, Hexadecimal
- In any number base, the value of i^{th} digit d is given by

  $d \times base^i$

Where i starts at 0 and increases from right to left

Number Systems (cont…)

- Decimal System/Base-10 System
  - Composed of 10 symbols or numerals - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- Binary System/Base-2 System
  - Composed of 2 symbols or numerals - 0, 1
- Hexadecimal System/Base-16 System
  - Composed of 16 symbols or numerals - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Integer Conversion - Decimal to Another Base

1. Divide the decimal number by the base (e.g. 2)

2. The *remainder* is the lowest-order digit

3. Repeat the first two steps until no *divisor* remains.

4. For binary the even number has no remainder ‘0’, while the odd has ‘1’

Converting a Fraction *from* Decimal *to* Another Base

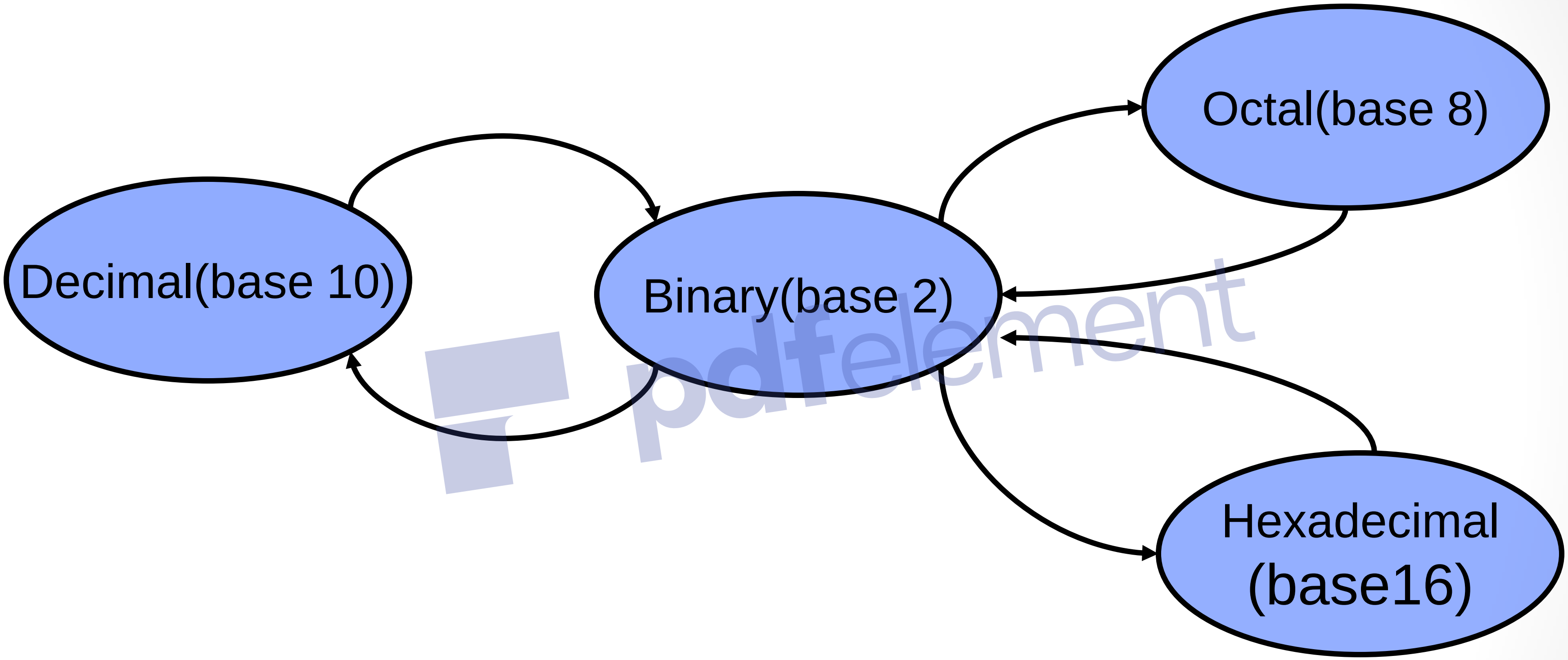
- Multiply decimal number by the base (e.g. 2)
- The *integer* is the highest-order digit
- Repeat the first two steps until fraction becomes zero.

Decimal to Binary Conversion (Integer + Fraction)

- Separate the decimal number into integer and fraction parts.
- Repeatedly divide the integer part by 2 to give a quotient and a remainder and
- Remove the remainder. Arrange the sequence of remainders right to left from the period. (Least significant bit first)
- Repeatedly multiply the fraction part by 2 to give an integer and a fraction part

  and remove the integer. Arrange the sequence of integers left to right from the period. (Most significant fraction bit first)

Conversion Between Number Bases



Number conversions (cont…)

Binary-to-Decimal Conversions

$1011.101_2 = (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3})$

$= 8_{10} + 0 + 2_{10} + 1_{10} + 0.5_{10} + 0 + 0.125_{10}$

$= 11.625_{10}$

Decimal-to-Binary Conversions (Repeated division)

18 / 2 = 9 remainder 0

9 / 2 = 4 remainder 1

4 / 2 = 2 remainder 0

2 / 2 = 1 remainder 0

1 / 2 = 0 remainder 1

Read the result upward to give an answer of

$18_{10} = 10010_2$

Number conversions (cont…)

$0.375 \times 2 = 0.750$ integer 0

$0.750 \times 2 = 1.500$ integer 1

$0.500 \times 2 = 1.000$ integer 1

Read the result downward $.375_{10} = .011_2$

Hex-to-Decimal Conversion

$$2AF_{16} = (2 \times 16^2) + (10 \times 16^1) + (15 \times 16^0)$$
$$= 512_{10} + 160_{10} + 15_{10}$$
$$= 687_{10}$$

Decimal-to-Hex Conversion

$423_{10} / 16 = 26$ remainder 7

$26_{10} / 16 = 1$ remainder 10

$1_{10} / 16 = 0$ remainder 1

Read the result upward to give an answer of $423_{10} = 1A7_{16}$

Number Conversions (cont…)

Hex-to-Binary Conversion

$9F2_{16} = 9 \quad F \quad 2$

$\downarrow \quad \downarrow \quad \downarrow$

$= 1001 \quad 1111 \quad 0010$

$= 100111110010_2$

Binary-to-Hex Conversion

$1\,1\,1\,0\,1\,0\,0\,1\,1\,0_2 = \underbrace{0\,0\,1\,1}_{3}\;\underbrace{1\,0\,1\,0}_{A}\;\underbrace{0\,1\,1\,0}_{6}$

$= 3A6_{16}$

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

ASCII Code

- American Standard Code for Information Interchange
- ASCII is a 7-bit code, frequently used with an 8th bit for error detection (more about that in a bit).

Character	ASCII (bin)	ASCII (hex)	Decimal	Octal
A	1000001	41	65	101
B	1000010	42	66	102
C	1000011	43	67	103
…				
Z				
a				
…				
1				
‘				

Number Representation

- Need to represent both +ve and –ve numbers.
- For computers, it is desirable to represent everything as bits.
- Three types of signed binary number representations:
  - sign-and-magnitude
  - 1's complement
  - 2's complement

Sign-and-Magnitude Representation

- Left most bit is sign bit
- 0 means positive
- 1 means negative
- +5 = 0000 0101
- -5 = 1000 0101
- Problems
    - Need to consider both sign and magnitude in arithmetic
    - Two representations of zero (+0 and -0)

One's Complement Representation

- The one's complement of a binary number involves inverting all bits.
- Negative values are obtained by complementing each bit of the corresponding +ve number.
- Example: +3 - 00000011

  -3 - 11111100

Two's Complement Representation

- The two's complement of a binary number involves inverting all bits and adding 1.
- Negative values are obtained by complementing each bit of the corresponding +ve number and adding 1.
- Example: +3 - 00000011

  -3 - 11111100 +1 = 11111101
- Range of numbers : Using n bits

  $+\ 2^{n-1} - 1$ to $-\ 2^{n-1}$
- ex: 8 bits : 127 to -128

  16 bits: 32767 to -32768

B	Values represented		
$b_3b_2b_1b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+7	+7	+7
0 1 1 0	+6	+6	+6
0 1 0 1	+5	+5	+5
0 1 0 0	+4	+4	+4
0 0 1 1	+3	+3	+3
0 0 1 0	+2	+2	+2
0 0 0 1	+1	+1	+1
0 0 0 0	+0	+0	+0
1 0 0 0	−0	−7	−8
1 0 0 1	−1	−6	−7
1 0 1 0	−2	−5	−6
1 0 1 1	−3	−4	−5
1 1 0 0	−4	−3	−4
1 1 0 1	−5	−2	−3
1 1 1 0	−6	−1	−2
1 1 1 1	−7	−0	−1

Two's Complement Representation

- The two's complement of a binary number involves inverting all bits and adding 1.
- Negative values are obtained by complementing each bit of the corresponding +ve number and adding 1.
- Example: +3 - 00000011

  -3 - 11111100 +1 = 11111101

Benefits:

- One representation of zero
- Arithmetic works easily
- Negating is fairly easy

Binary Addition

Example Add $(11110)_2$ to $(10111)_2$

	1	*1*	*1*	*1*	*1*	*1*		← *carries*
		1	1	1	1	0	1	
+			1	0	1	1	1	
	1	0	1	0	1	0	0	

$(111101)_2 + (10111)_2 = (1010100)_2$ ← *carry*

Binary Subtraction

Example subtract $(0111)_2$ from $(1101)_2$

```
        10 10          <------------------ Borrows
    1   1   0   1
-   0   1   1   1
----------------------
    0   1   1   0
```


$(1101)_2 + (0111)_2 = (0110)_2$

Addition and Subtraction – Signed Numbers

Addition:

1. Add the n-bit representations
2. Ignore the carry out of the MSB position.
3. Result is in 2's complement representation as long as the result is in the range $+ 2^{n-1} -1$ to $- 2^{n-1}$

Subtraction: To perform X-Y,

1. Take 2's complement of Y.
2. Add to X.
3. Result is in 2's complement representation as long as the result is in the range $+ 2^{n-1} -1$ to $- 2^{n-1}$

Addition & Subtraction Examples

Add 4 , -6

```
   0 1 0 0 (4)
 + 1 0 1 0 (-6)
 --------------
   1 1 1 0 (-2)
```

Subtract -5 from -7

```
   1 0 0 1 (-7)
 - 1 0 1 1 (-5)
 --------------
```

```
   1 0 0 1 (-7)
   0 1 0 1  (5)
 --------------
   1 1  1 0 (-2)
```

Subtract 3 from 6

6 - 3

```
   0 1 1 0
   1 1 0 1
 ---------
   0 0 1 1
```

Overflow

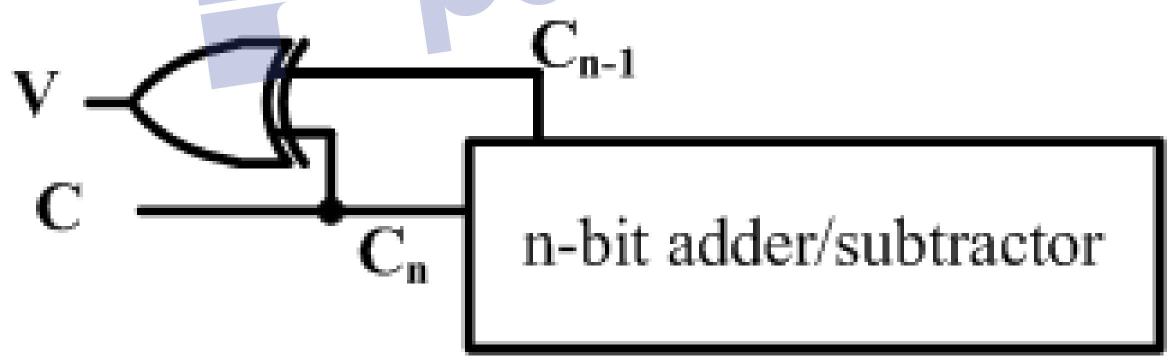
- When adding 2 n-bit numbers it is possible to get a n+1 bit result if there is a carry out.
- Overflow is said to occur when two numbers of n digits each are added and the sum occupies n+1 digits
- n + 1 bit cannot be accommodated in a register with a standard length of n bits(*many computer detect the occurrence of an overflow, and a corresponding F/F is set*)

Overflow indication

- In 8-bit 2's complement notation the range that can be represented is -128 to +127.
- Then the operation to add +70 to +80 is
  - Carries 0 1
  - +70 0 100 0110
  - +80 0 101 0000
  - +150 1 001 0110
- When both the operands have the same sign, an overflow is said to occur if the sign of the resultant is different from that of the operands.

Overflow indication (cont...)

- The rule – if the carry into the msb position differs from the carry out from the msb position then an overflow has occurred.



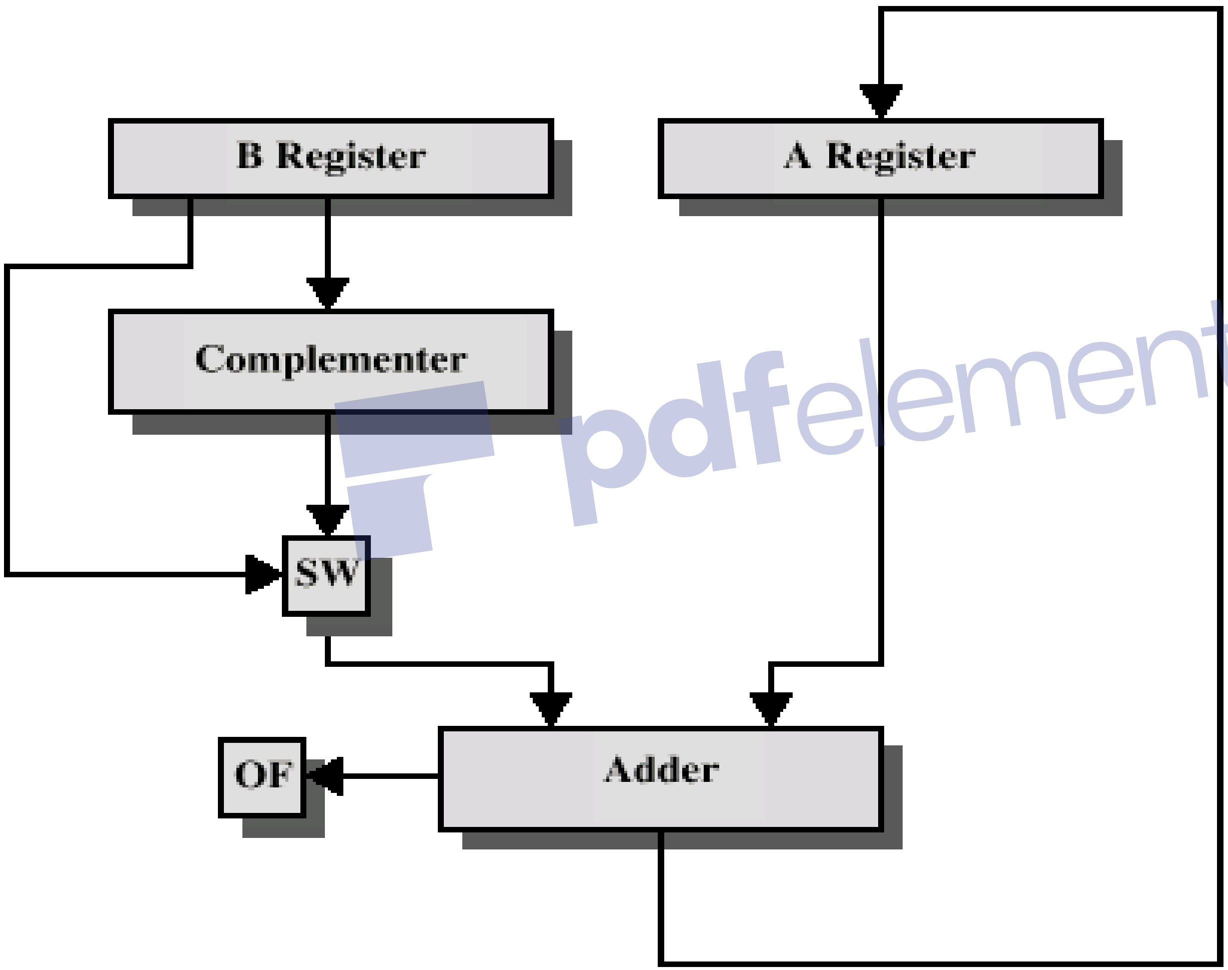
Full- Adder

- Adds 3-bits - Has 3-inputs and 2-outputs
- We will use x, y and z for inputs and s for sum and c for carry are the two outputs.

The truth table

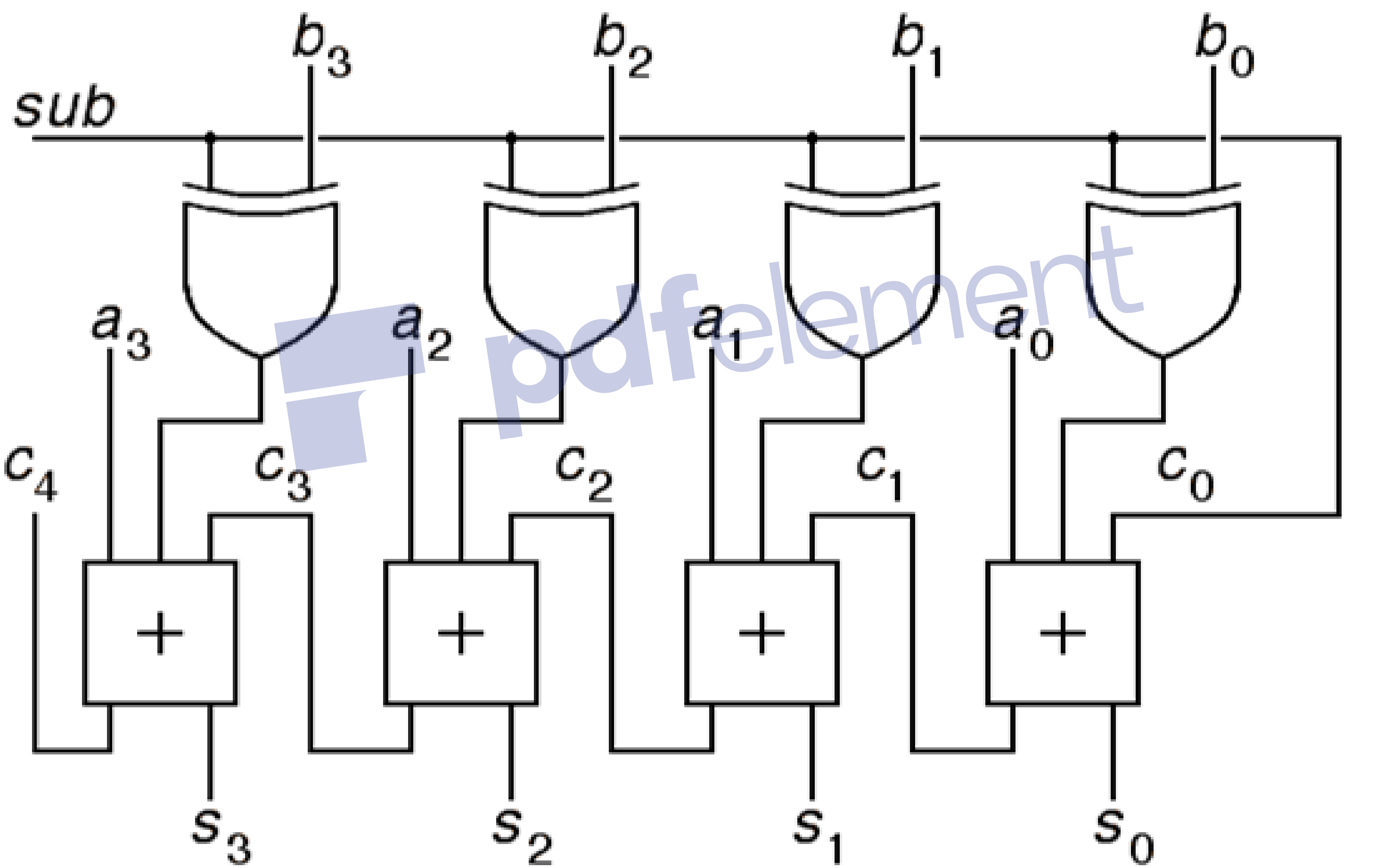
x	y	z	c	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Hardware for Addition and Subtraction

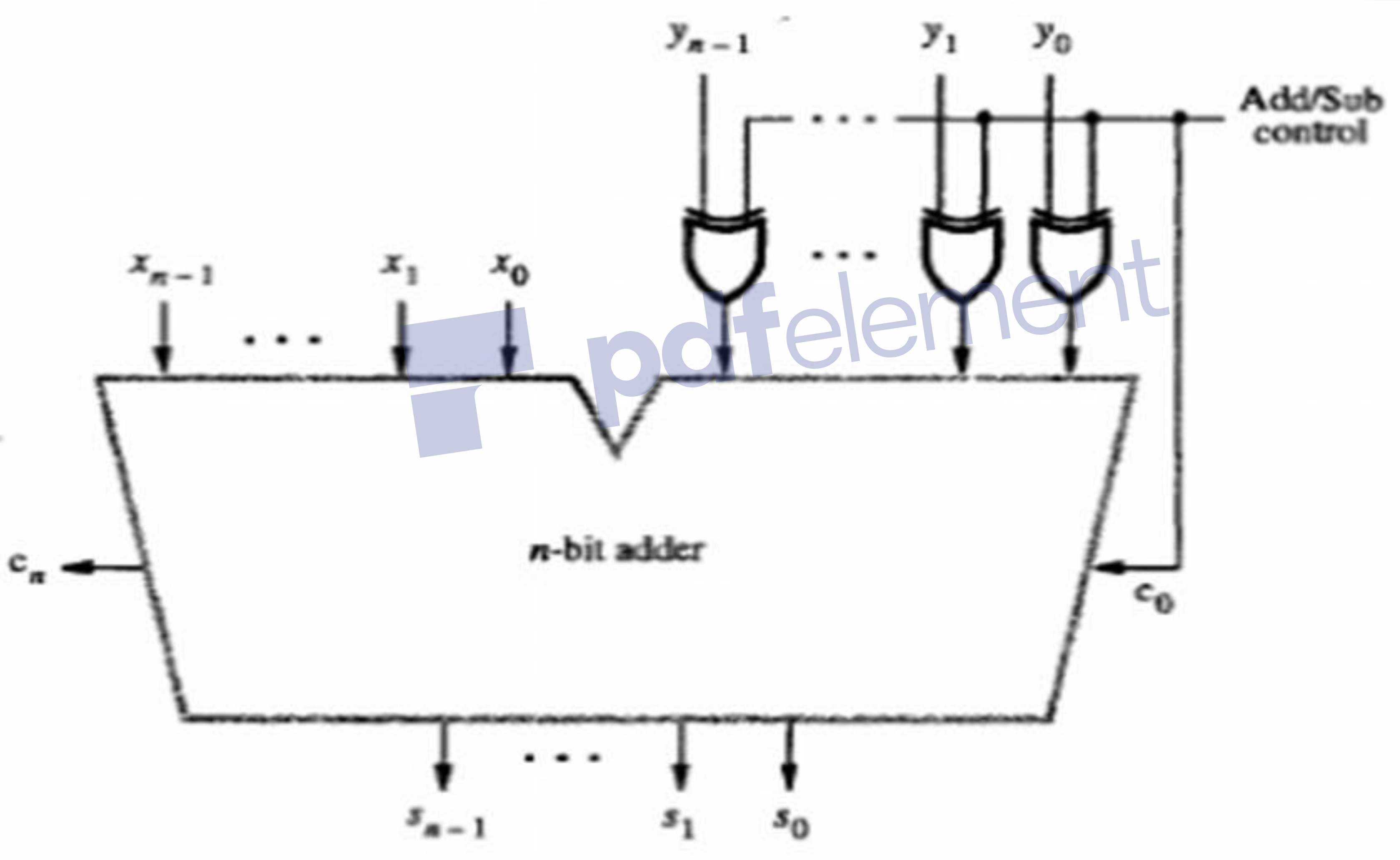


OF = overflow bit
SW = Switch (select addition or subtraction)

Adder Subtractor (4-bit Ripple-carry adder)



Adder Subtractor (cont…)



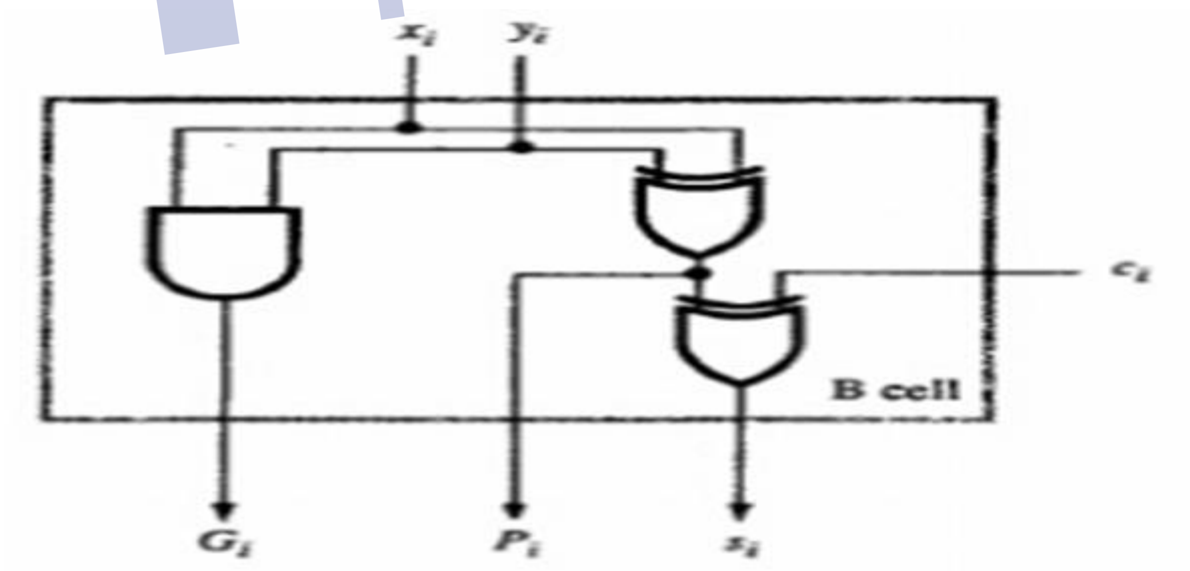
Design of Fast Adders

- Delay - sum of logic-gate delays along the longest path.
- In case of ripple carry adder longest path is from x_0, y_0, c_0 at LSB position to c_n and s_{n-1} at MSB

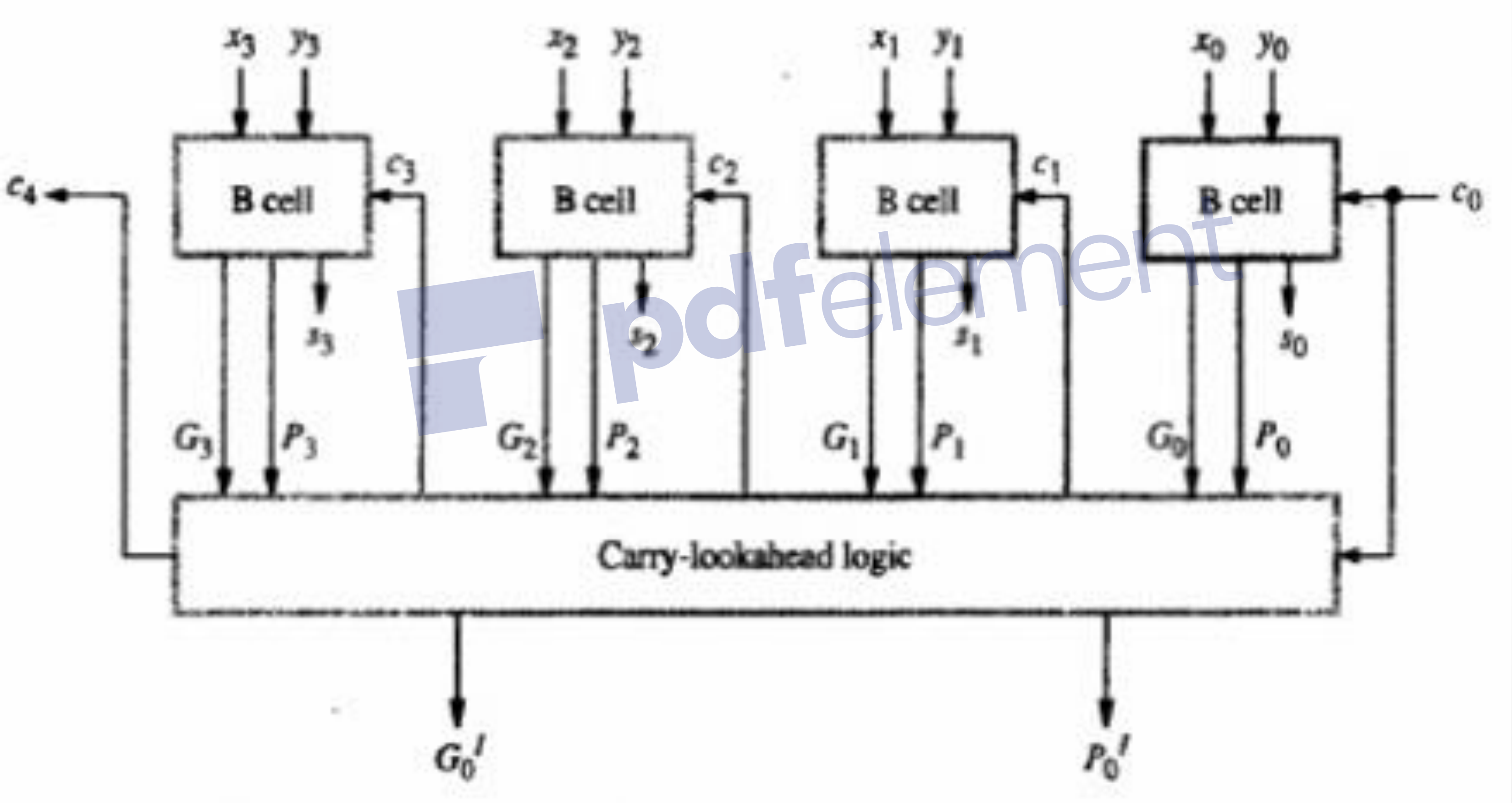
Carry Lookahead Adder

- $S_i = x_i \oplus y_i \oplus c_i$
- $c_{i+1} = x_i\, y_i + x_i\, c_i + y_i\, c_i$
- $c_{i+1} = x_i\, y_i + (x_i + y_i)c_i$

$$c_{i+1} = g_i + c_i p_i$$



Carry Lookahead Adder (cont...)



Unrolling Carry Recurrence

$$c_i = g_{i-1} + c_{i-1}p_{i-1} =$$

$$= g_{i-1} + (g_{i-2} + c_{i-2}p_{i-2})p_{i-1} = g_{i-1} + g_{i-2}\,p_{i-1} + c_{i-2}p_{i-2}p_{i-1} =$$

$$= g_{i-1} + g_{i-2}\,p_{i-1} + (g_{i-3} + c_{i-3}p_{i-3})p_{i-2}p_{i-1} =$$

$$= g_{i-1} + g_{i-2}\,p_{i-1} + g_{i-3}\,p_{i-2}p_{i-1} + c_{i-3}p_{i-3}p_{i-2}p_{i-1} =$$

$$= \ldots.. =$$

$$= g_{i-1} + g_{i-2}\,p_{i-1} + g_{i-3}\,p_{i-2}p_{i-1} + g_{i-4}p_{i-3}p_{i-2}p_{i-1} + \ldots.. +$$

$$+ g_0p_1p_2\ldots p_{i-2}p_{i-1} + c_0p_0p_1p_2\ldots p_{i-2}p_{i-1} =$$

$$= \boxed{g_{i-1} + \sum_{k=0}^{i-2} g_k \prod_{j=k+1}^{i-1} p_j + c_0 \prod_{j=0}^{i-1} p_j}$$

4-bit Carry-Lookahead Adder

$$c_4 = g_3 + g_2\,p_3 + g_1\,p_2p_3 + g_0p_1p_2p_3 + c_0p_0p_1p_2p_3$$

$$c_3 = g_2 + g_1\,p_2 + g_0\,p_1p_2 + c_0p_0p_1p_2$$

$$c_2 = g_1 + g_0\,p_1 + c_0p_0p_1$$

$$c_1 = g_0 + c_0\,p_0$$

$$s_0 = x_0 \oplus y_0 \oplus c_0 = p_0 \oplus c_0$$

$$s_1 = p_1 \oplus c_1$$

$$s_2 = p_2 \oplus c_2$$

$$s_3 = p_3 \oplus c_3$$

16-bit Carry-Lookahead Adder

$$C_4 = G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3 + c_0 P_0 P_1 P_2 P_3$$

$$C_3 = G_2 + G_1 P_2 + G_0 P_1 P_2 + c_0 P_0 P_1 P_2$$

$$C_2 = G_1 + G_0 P_1 + c_0 P_0 P_1$$

$$C_1 = G_0 + c_0 P_0$$

16-bit Carry-Lookahead Adder (cont…)

$$P_0 = p_3 \cdot p_2 \cdot p_1 \cdot p_0$$

$$P_1 = p_7 \cdot p_6 \cdot p_5 \cdot p_4$$

$$P_2 = p_{11} \cdot p_{10} \cdot p_9 \cdot p_8$$

$$P_3 = p_{15} \cdot p_{14} \cdot p_{13} \cdot p_{12}$$

$$G_0 = g_3 + g_2 p_3 + g_1 p_2 p_3 + g_0 p_1 p_2 p_3$$

$$G_1 = g_7 + g_6 p_7 + g_5 p_6 p_7 + g_4 p_5 p_6 p_7$$

$$G_2 = g_{11} + g_{10} p_{11} + g_9 p_{10} p_{11} + g_8 p_9 p_{10} p_{11}$$

$$G_3 = g_{15} + g_{14} p_{15} + g_{13} p_{14} p_{15} + g_{12} p_{13} p_{14} p_{15}$$

Example

Determine gi, pi, Pi and Gi values of the following 16 bit numbers

a. 0001 1010 0011 0011

b. 1110 0101 1110 1011

Determine what is the final carryout (i.e. C4)

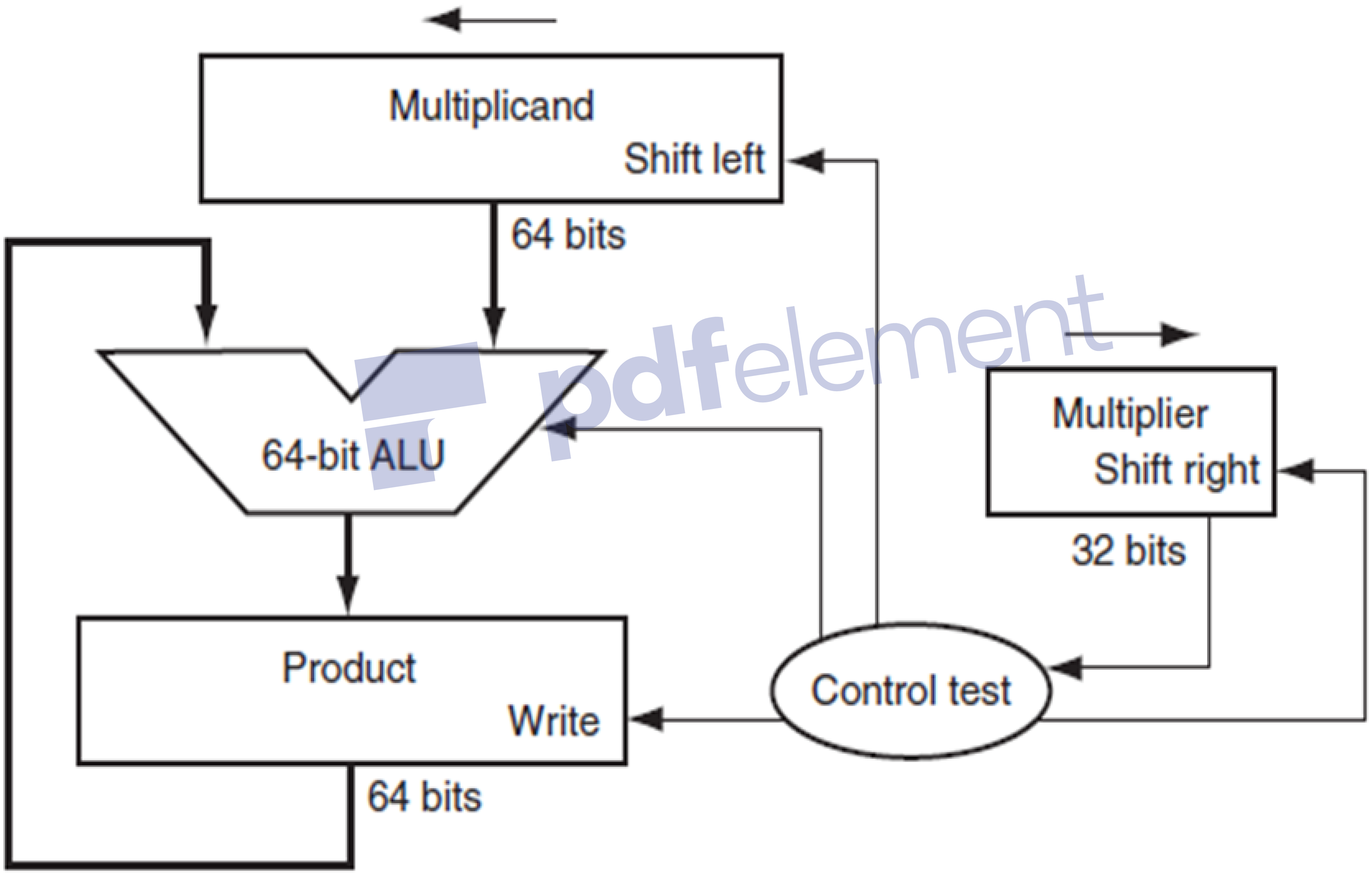
Multiplication – Positive Numbers

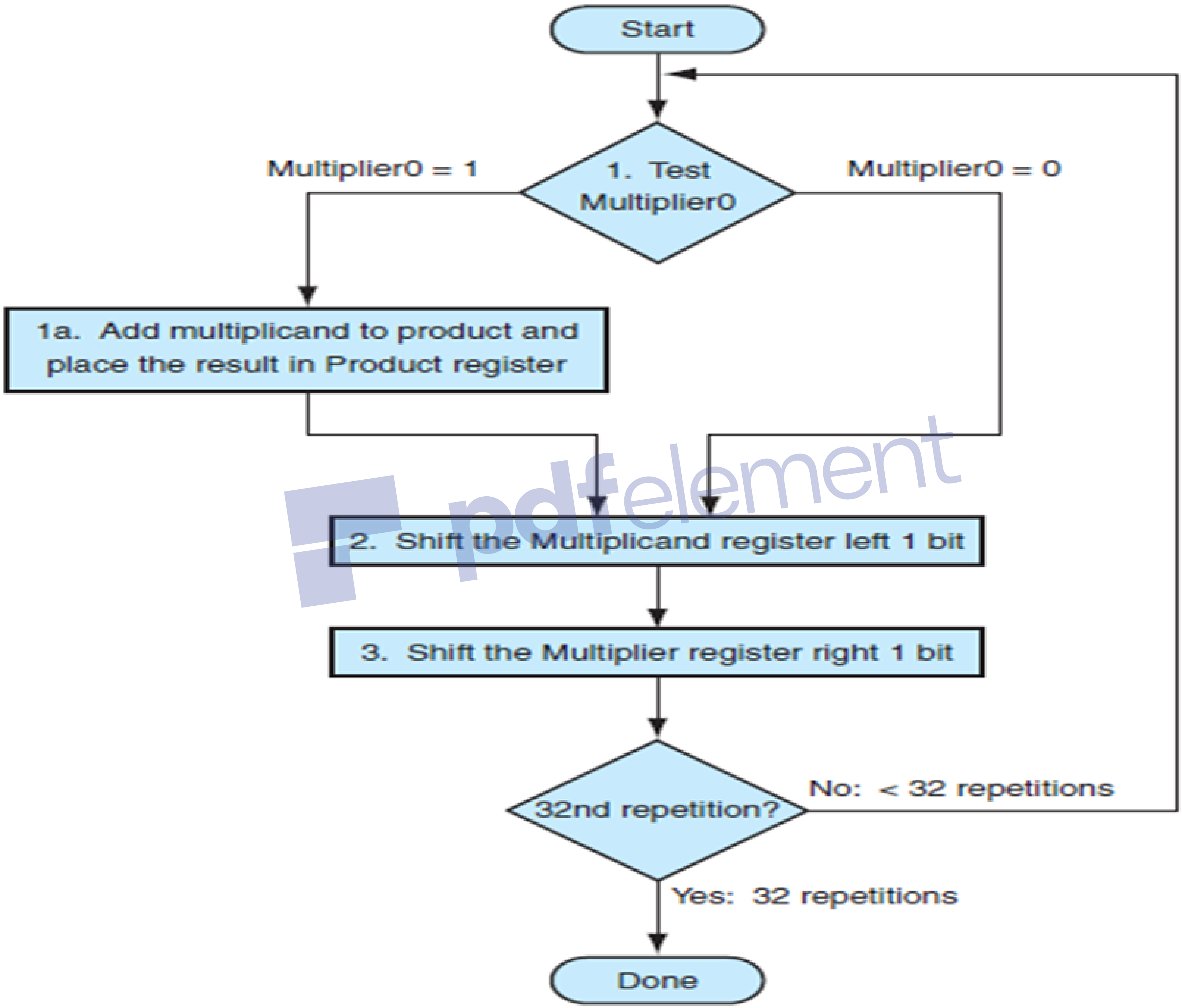
1 0 0 0	Multiplicand
1 0 0 1	Multiplier

```
      1 0 0 0
    0 0 0 0
  0 0 0 0
1 0 0 0
-------------
1 0 0 1 0 0 0
```

1 0 0 1 0 0 0	Product

Simple Multiplication



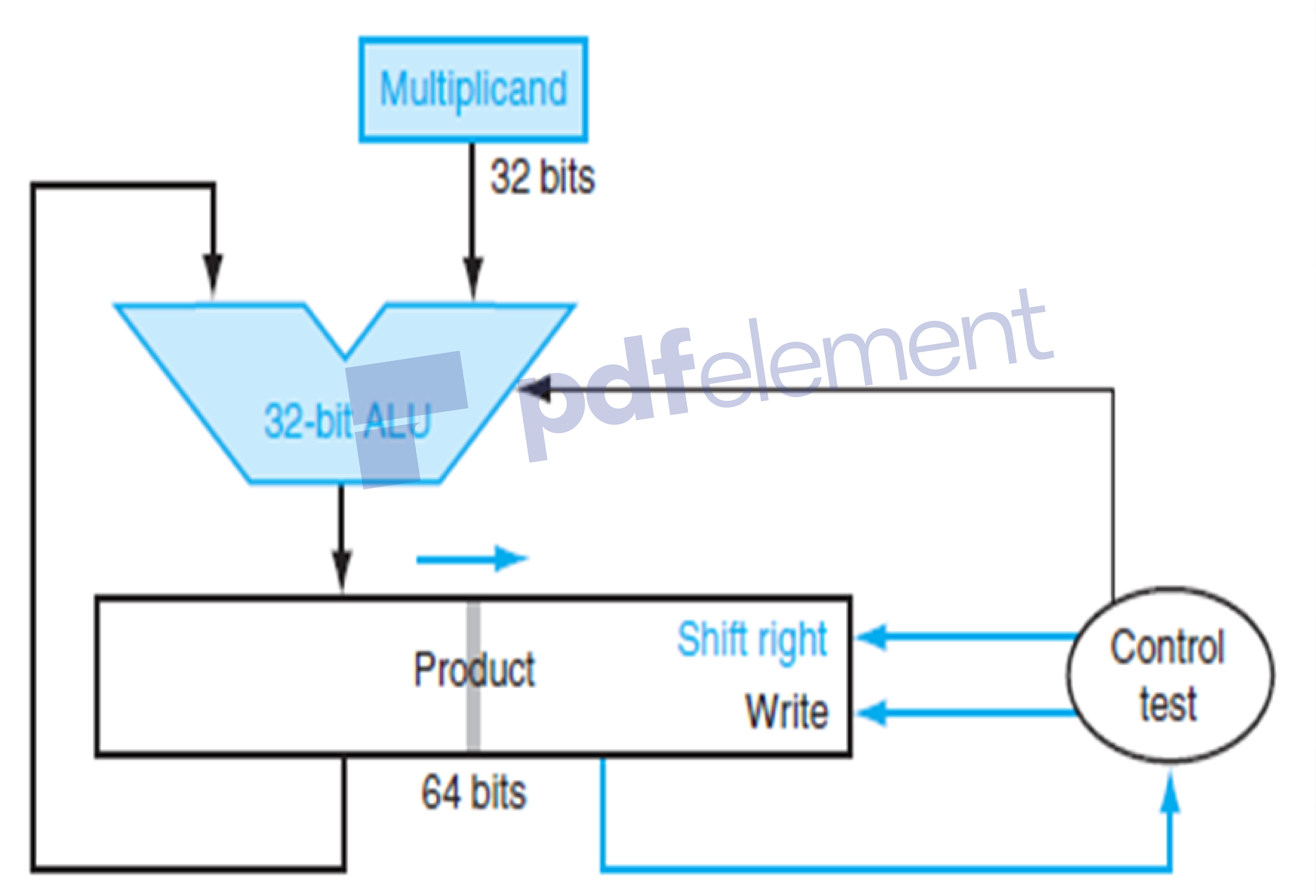


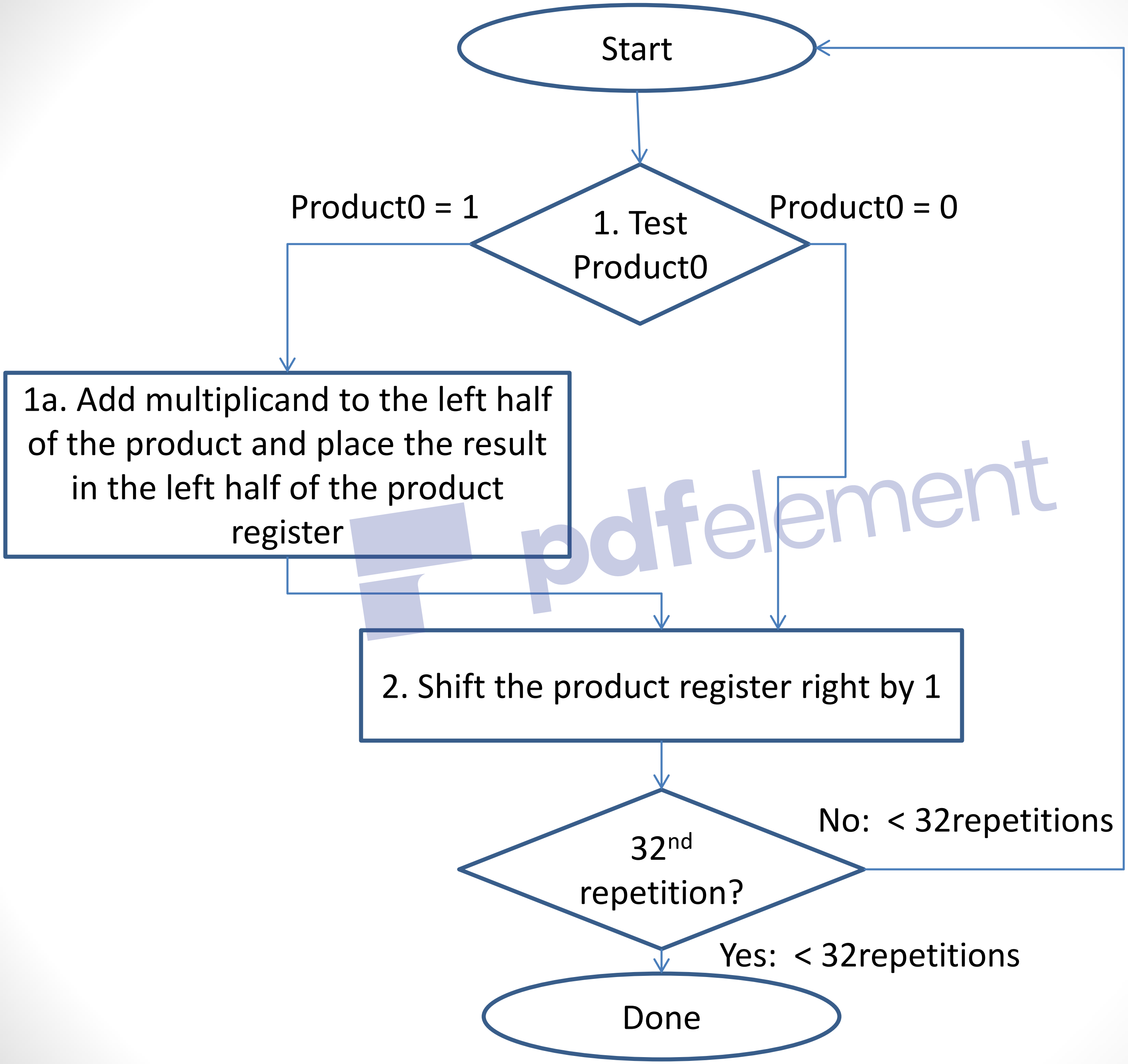
Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

Multiplication Example

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial Value	0011	0000 0010	0000 0000
1	1a. Prod + Mcand	0011	0000 0010	0000 0010
	2. Slt Mcand	0011	0000 0100	0000 0010
	3. Srght Multiplier	0001	0000 0100	0000 0010
2	1a. Prod + Mcand	0001	0000 0100	0000 0110
	2. Slt Mcand	0001	0000 1000	0000 0110
	3. Srght Multiplier	0000	0000 1000	0000 0110
3	1. No operation	0000	0000 1000	0000 0110
	2. Slt Mcand	0000	0001 0000	0000 0110
	3. Srght Multiplier	0000	0001 0000	0000 0110
4	1. No operation	0000	0001 0000	0000 0110
	2. Slt Mcand	0000	0010 0000	0000 0110
	3. Srght Multiplier	0000	0010 0000	0000 0110

Multiplication – Version II




Start
Product0 = 1
1. Test Product0
Product0 = 0
1a. Add multiplicand to the left half of the product and place the result in the left half of the product register
2. Shift the product register right by 1
32nd repetition?
No: < 32repetitions
Yes: < 32repetitions
Done

Multiplication Example

Iteration	Step	Multiplicand	Product
0	Initial Value	0010	0000 0011
1	1a. Prod = Prod + Mcand	0010	0010 0011
	2. Sh.rght prod	0010	0001 0001
2	1a. Prod = Prod + Mcand	0010	0011 0001
	2. Sh.rght prod	0010	0001 1000
3	1. No operation	0010	0001 1000
	2. Sh.rght prod	0010	0000 1100
4	1. No operation	0010	0000 1100
	2. Sh.rght prod	0010	0000 0110

Booth's Algorithm – Signed Numbers

- Initialize product register with multiplier (Right half)
- Test the two bits (LSB, Previous LSB) and do one of the following steps
  - 00: No arithmetic operation.
  - 01: Add multiplicand to the left half of the product.
  - 10: Subtract multiplicand from left half of the product
  - 11: No arithmetic operation
- Shift the product register right by 1
- While doing the shift operation, sign of the result (product) must be preserved.

Booth's Multiplication Example

Iteration	Step	Multiplicand	Product
0	Initial Value	0010	0000 0110 0
1	1a. No operation	0010	0000 0110 0
	2. Sh.rght prod	0010	0000 0011 0
2	1c. Prod = Prod - Mcand	0010	1110 0011 0
	2. Sh.rght prod	0010	1111 0001 1
3	1d. No operation	0010	1111 0001 1
	2. Sh.rght prod	0010	1111 1000 1
4	1b. Prod = Prod + Mcand	0010	0001 1000 1
	2. Sh.rght prod	0010	0000 1100 0

Division

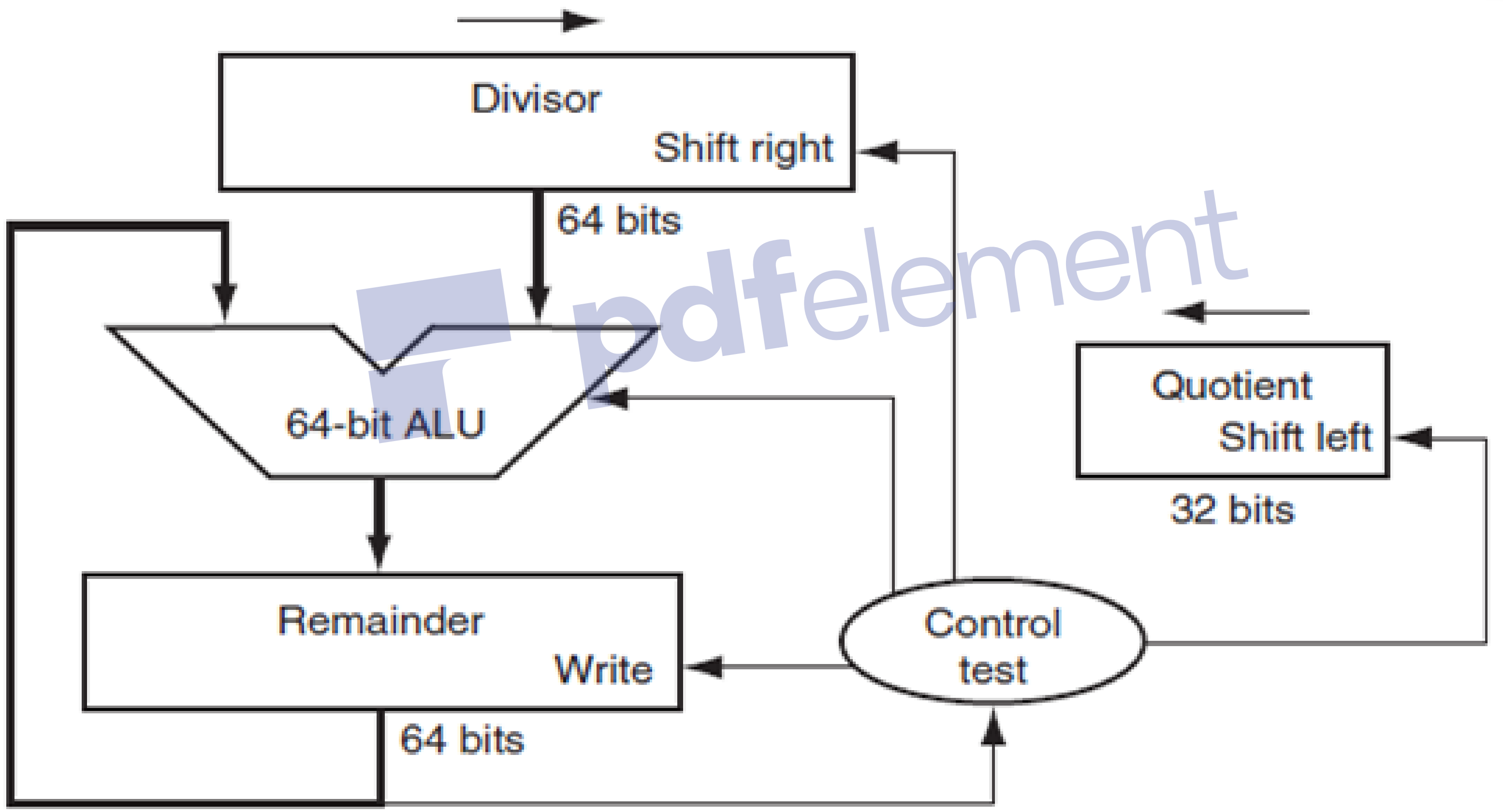
Dividend A number being divided.

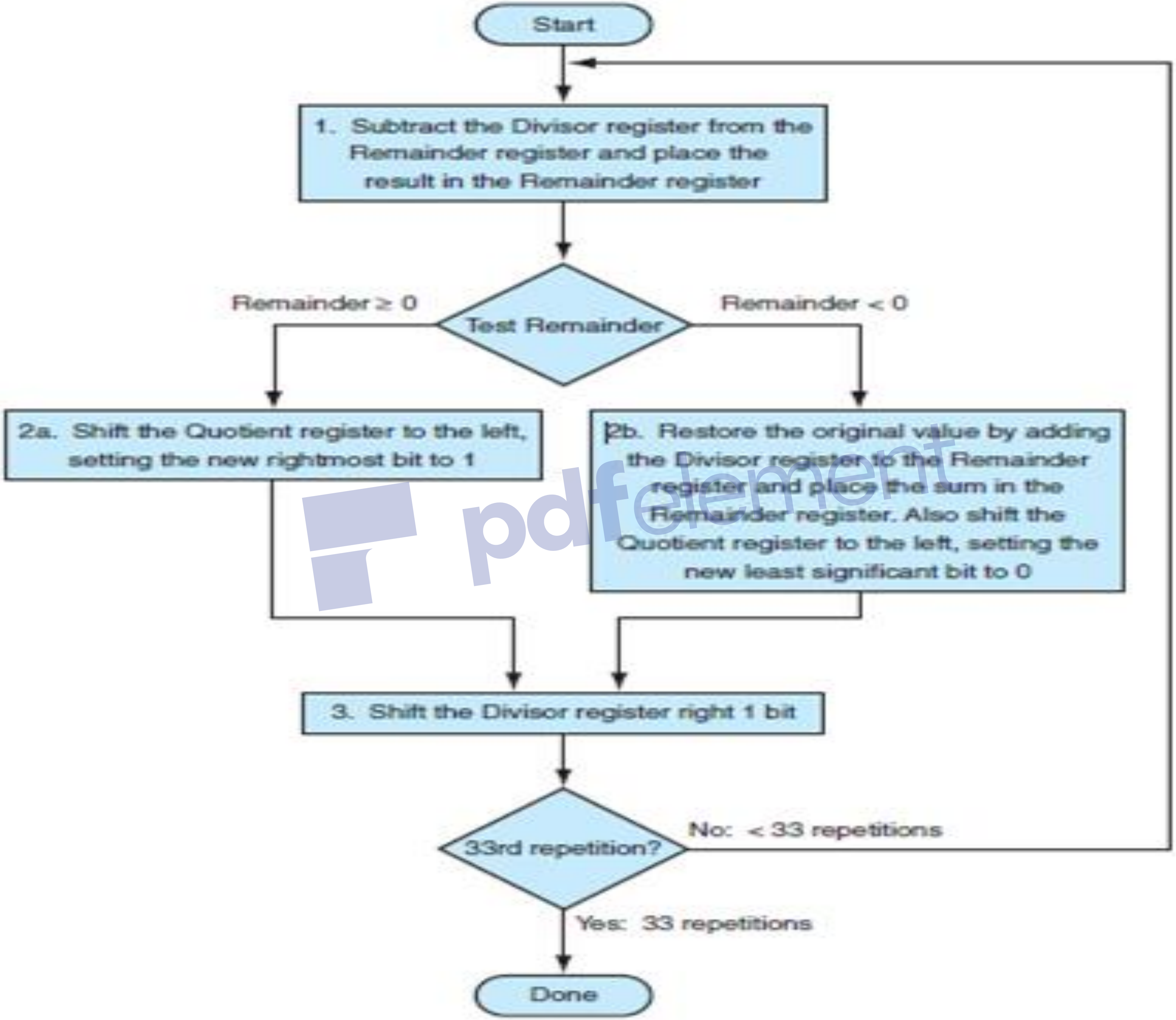
Divisor A number that the dividend is divided by.

Quotient, Remainder The primary and secondary results of a division

Dividend = Divisor X Quotient + Remainder

Simple Division Algorithm




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 place 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

Simple Division Algorithm (cont...)

Divide 7 by 2 using four bit division

Dividend: 0111

Divisor: 0010

Initialization

Dividend: 0000 0111

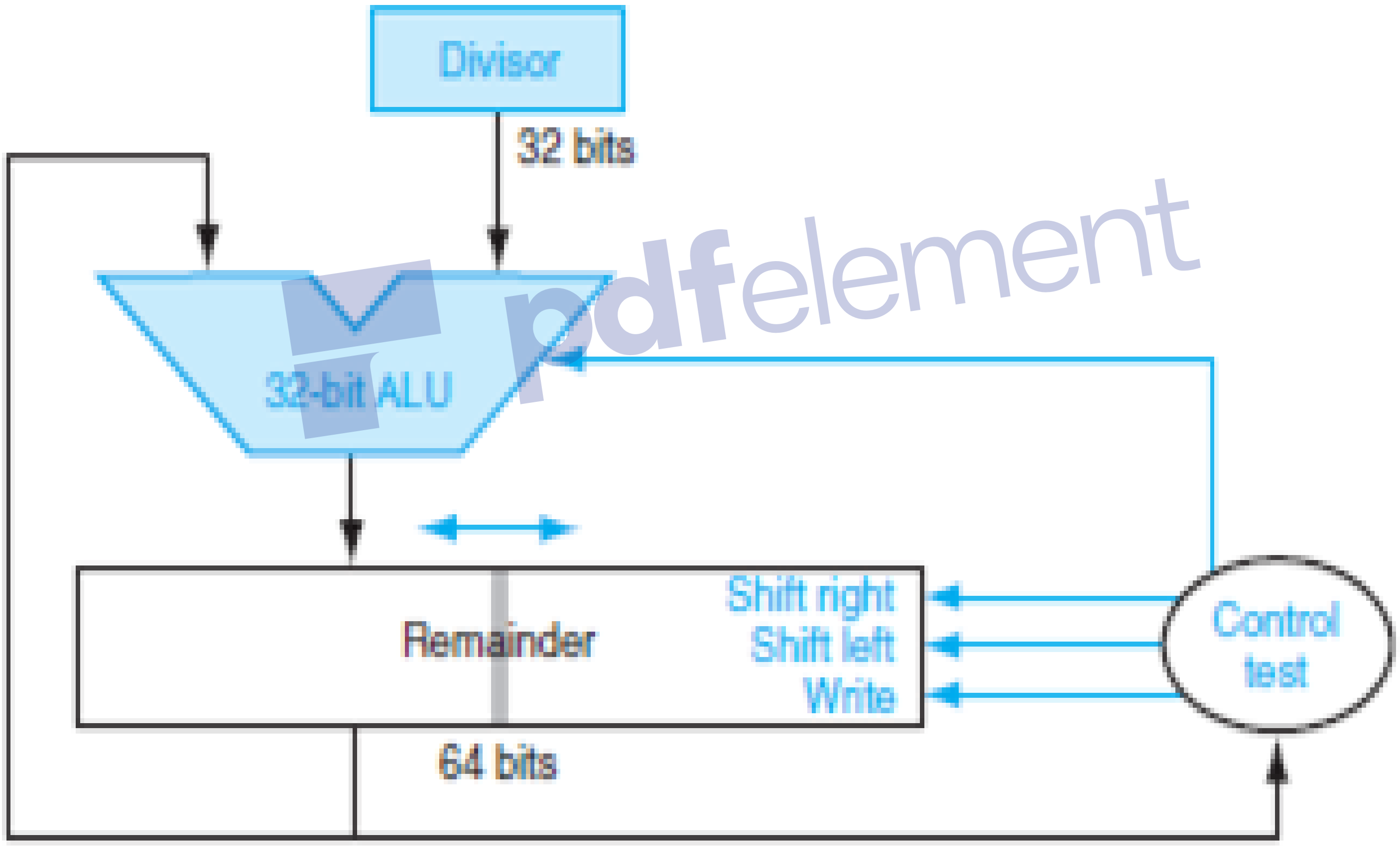
Divisor: 0010 0000

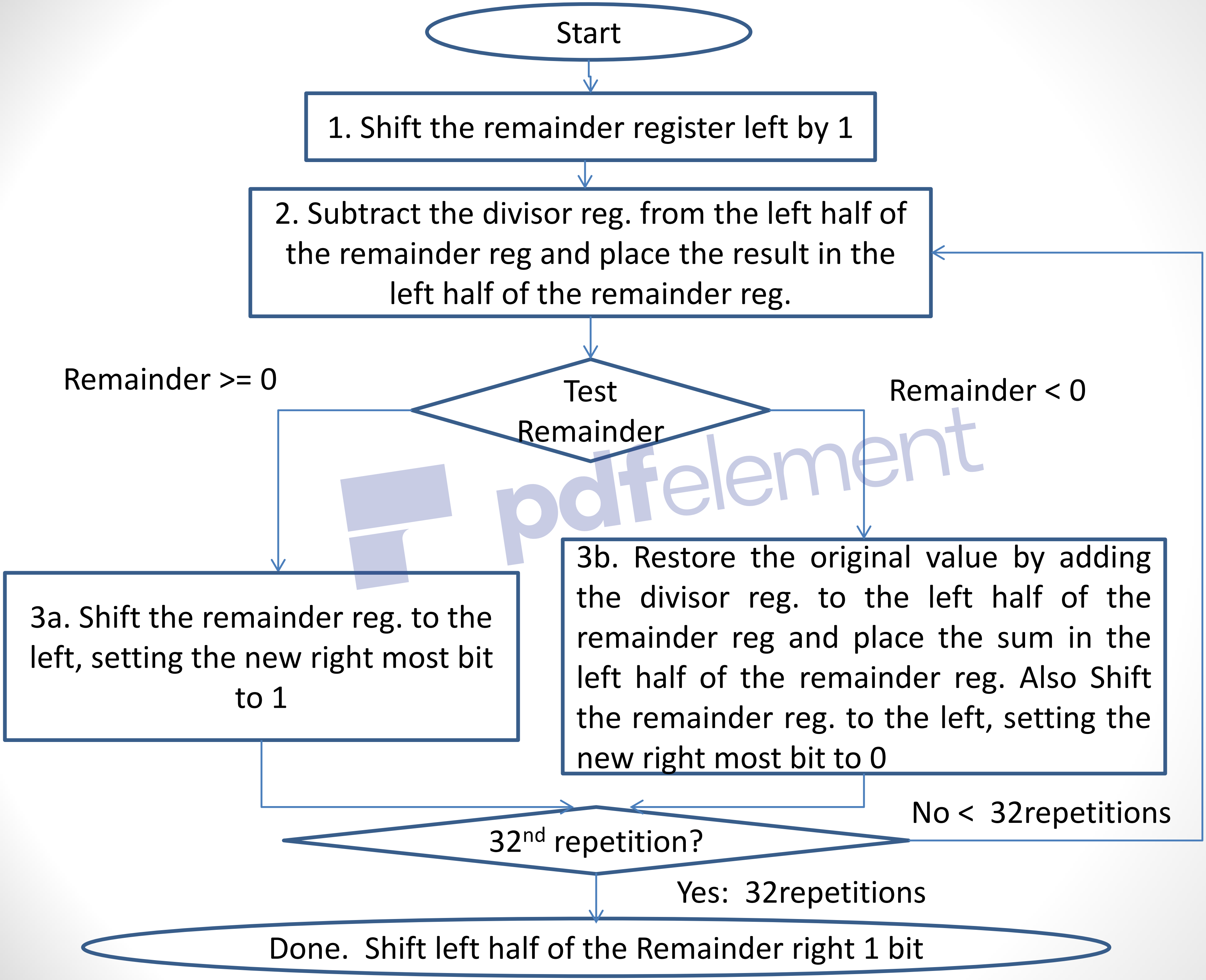
Quotient : 0000

Remainder: 0000 0111

Iteration	Step	Quotient	Divisor	Remainder
1	1: Rem = Rem – Div	0000	0010 0000	1110 0111
	2b: Rem < 0 → +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	1111 0111
	2b: Rem < 0 → +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	1111 1111
	2b: Rem < 0 +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	0000 0011
	2a: Rem >= 0 sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	0000 0001
	2a: Rem >= 0 sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

Division Algorithm –version II




Start
1. Shift the remainder register left by 1
2. Subtract the divisor reg. from the left half of the remainder reg and place the result in the left half of the remainder reg.
Test Remainder
Remainder >= 0
Remainder < 0
3a. Shift the remainder reg. to the left, setting the new right most bit to 1
3b. Restore the original value by adding the divisor reg. to the left half of the remainder reg and place the sum in the left half of the remainder reg. Also Shift the remainder reg. to the left, setting the new right most bit to 0
32nd repetition?
No < 32repetitions
Yes: 32repetitions
Done. Shift left half of the Remainder right 1 bit

Iteration	Step	Divisor	Remainder
0	Initialization	0010	0000 0111
	Shift Rem. Left 1	0010	0000 1110
1	2: Rem = Rem – Div	0010	1110 1110
	3b: Rem < 0 +Div, sll R, R0 = 0	0010	0001 1100
2	2: Rem = Rem – Div	0010	1111 1100
	3b: Rem < 0 +Div, sll R, R0 = 0	0010	0011 1000
3	2: Rem = Rem – Div	0010	0001 1000
	3a: Rem > =0 , sll R, R0 = 1	0010	0011 0001
4	2: Rem = Rem – Div	0010	0001 0001
	3a: Rem > =0 , sll R, R0 = 1	0010	0010 0011
	Shift left half of Rem. right	0010	0001 0011

Signed division

- Perform the division with positive operands.
- Negate the quotient if the signs of the operands are opposite
- Make the sign of the nonzero remainder match the dividend.

Nonrestoring division

Floating Point Representation

- 0.000000001_{ten} or $1.0_{ten} \times 10^{-9}$
- $3{,}155{,}760{,}000_{ten}$ or $3.15576_{ten} \times 10^{9}$
- **Scientific notation** : A notation that renders numbers with a single digit to the left of the decimal point.
- **Normalized form:** A number in scientific notation that has no leading 0s.

For example 0.1×10^{-9} , 10.0×10^{9} are not normalized.

Normalized binary floating point in scientific notation $1.xxxxx_{two} \times 2^{yyyy}$

Floating Point Representation (cont…)

single precision A floating point value represented in a single 32-bit word.

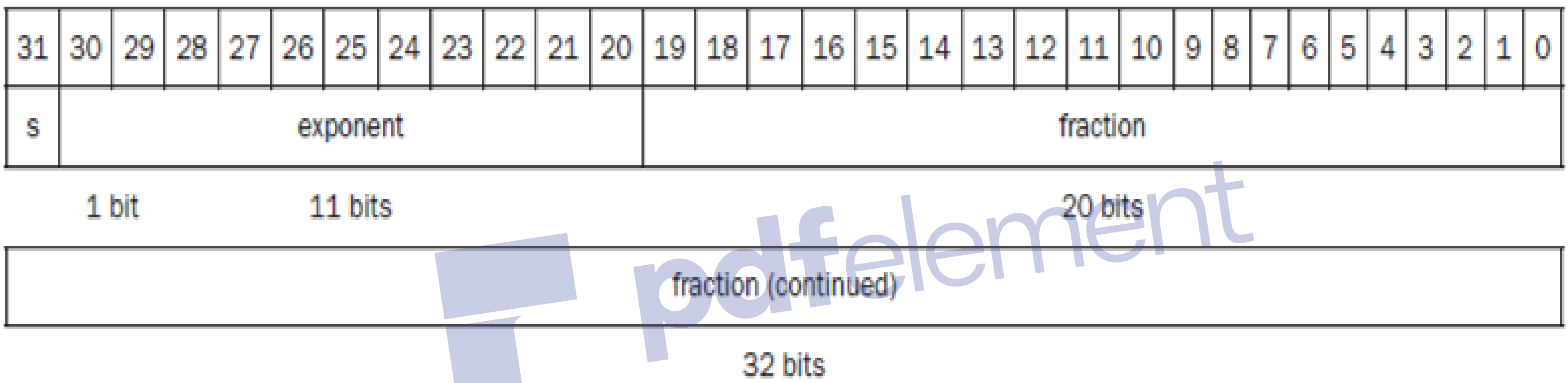
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s	exponent								fraction																						
1 bit	8 bits								23 bits																						

In general floating point numbers are of the form

$$(-1)^S \times F \times 2^E$$

Floating Point Representation (cont…)

Double precision A floating point value represented in two 32-bit words.



overflow (floating-point) : When positive exponent becomes too large to fit in the exponent field.

Underflow: When a negative exponent becomes too large to fit in the exponent field.

IEEE 754 Format

- Standard for floating point storage
- 32 and 64 bit standards
- 8 and 11 bit exponent respectively.
- To pack even more bits into the significand, IEEE 754 makes the leading 1 bit of normalized binary numbers implicit.

$$(-1)^S \times (1 + \text{Fraction}) \times 2^E$$

where the bits of the fraction represent a number between 0 and 1 and E specifies the value in the exponent field

IEEE 754 Format (cont…)

If we number the bits of the fraction from *left to right* s1, s2, s3, . . . , then the value is

$(-1)^S \times (1 + (s1 \times 2^{-1}) + (s2 \times 2^{-2}) + (s3 \times 2^{-3}) + (s4 \times 2^{-4}) + \ldots) \times 2^E$

1.0×2^{-1}

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	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

1.0×2^{1}

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	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

IEEE 754 Format (cont…)

Biased notation

- Most negative represented as $00 \ldots 00_{two}$ and the most positive as $11 \ldots 11_{two}$
- **Bias**: number subtracted from the normal, unsigned representation to determine the real value
- Single precision : 127
- Double precision : 1023
- Biased exponent means that the value represented by a floating-point number is really

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

Exercise

Show the IEEE 754 binary representation of the number – 0.75_{ten} in single and double precision.

Solution: $-0.75_{10} = -.11_{(2)} = -.11_{(2)} \times 2^0$

Normalized scientific notation $= -1.1 \times 2^{-1}$

The general representation for a single precision number is $(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 000_2) \times 2^{(126-127)}$

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
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit 8 bits 23 bits

Floating-Point Addition

Consider a 4-digit decimal example

$9.999 \times 10^{1} + 1.610 \times 10^{-1}$

1. Align decimal points

   Shift number with smaller exponent

   $9.999 \times 10^{1} + 0.016 \times 10^{1}$

2. Add significands

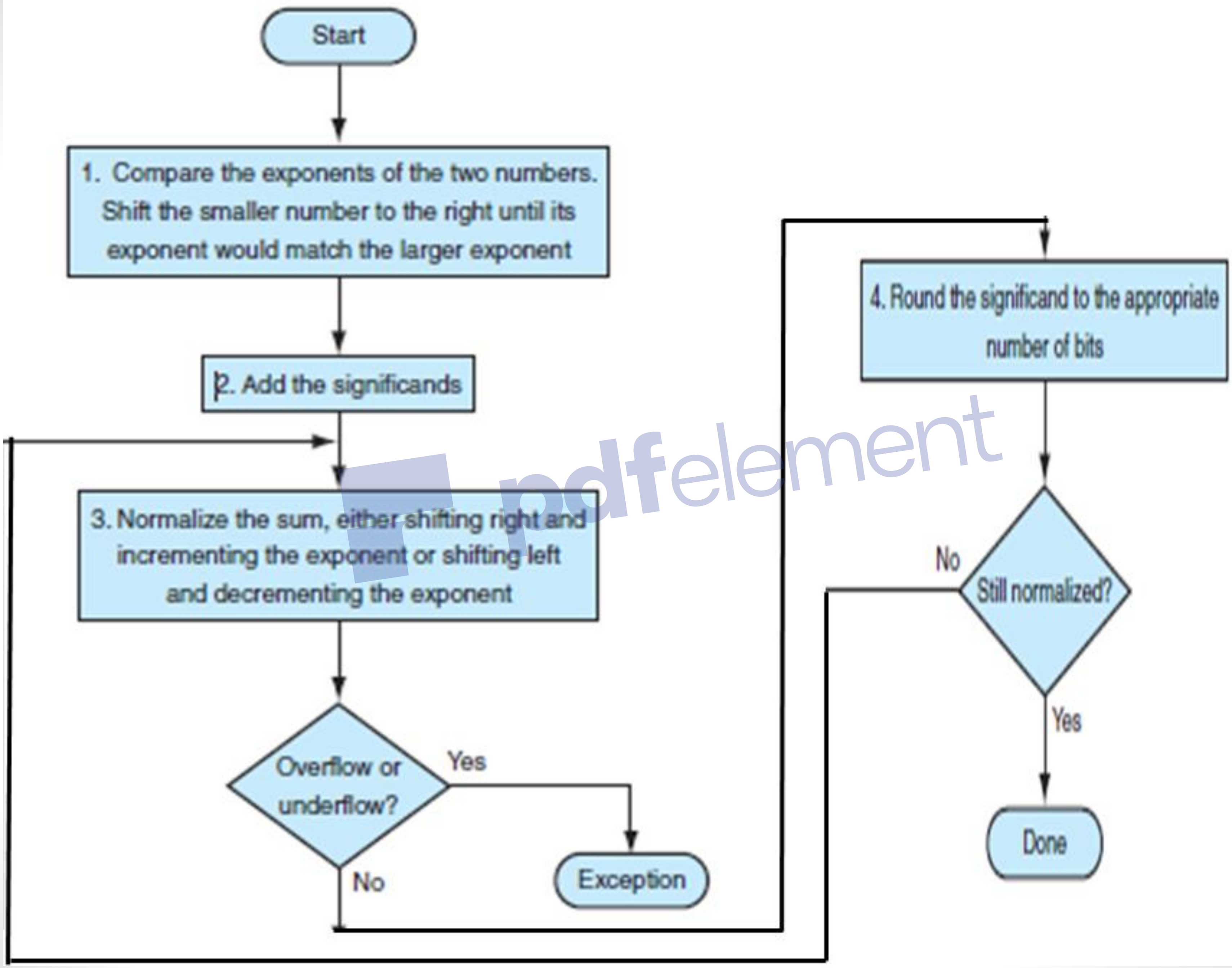
   $9.999 \times 10^{1} + 0.016 \times 10^{1} = 10.015 \times 10^{1}$

3. Normalize result & check for over/underflow

   1.0015×10^{2}

4. Round and renormalize if necessary

   1.002×10^{2}


Start
1. Compare the exponents of the two numbers. Shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
No
Exception
4. Round the significand to the appropriate number of bits
Still normalized?
No
Yes
Done

Floating-Point Addition (cont…)

Now consider a 4-digit binary example

Add 0.5 + –0.4375

$1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$

1. Align binary points

Shift number with smaller exponent

$1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$

2. Add significands

$1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$

3. Normalize result & check for over/underflow

$1.000_2 \times 2^{-4}$, with no over/underflow

4. Round and renormalize if necessary

$1.000_2 \times 2^{-4}$ (no change) = 0.0625

Floating-Point Multiplication

Consider a 4-digit decimal example

$1.110 \times 10^{10} \times 9.200 \times 10^{-5}$

1. Add exponents

   For biased exponents, subtract bias from sum

   New exponent = 10 + –5 = 5

2. Multiply significands

   $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^5$

3. Normalize result & check for over/underflow

   1.0212×10^6

4. Round and renormalize if necessary

   1.021×10^6

5. Determine sign of result from signs of operands

   $+1.021 \times 10^6$

Floating-Point Multiplication

Now consider a 4-digit binary example (0.5×-0.4375)

$1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$

1. Add exponents

   Unbiased: $-1 + -2 = -3$

   Biased: $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$

2. Multiply significands

   $1.000_2 \times 1.110_2 = 1.110_2 \Rightarrow 1.110_2 \times 2^{-3}$

3. Normalize result & check for over/underflow

   $1.110_2 \times 2^{-3}$ (no change) with no over/underflow

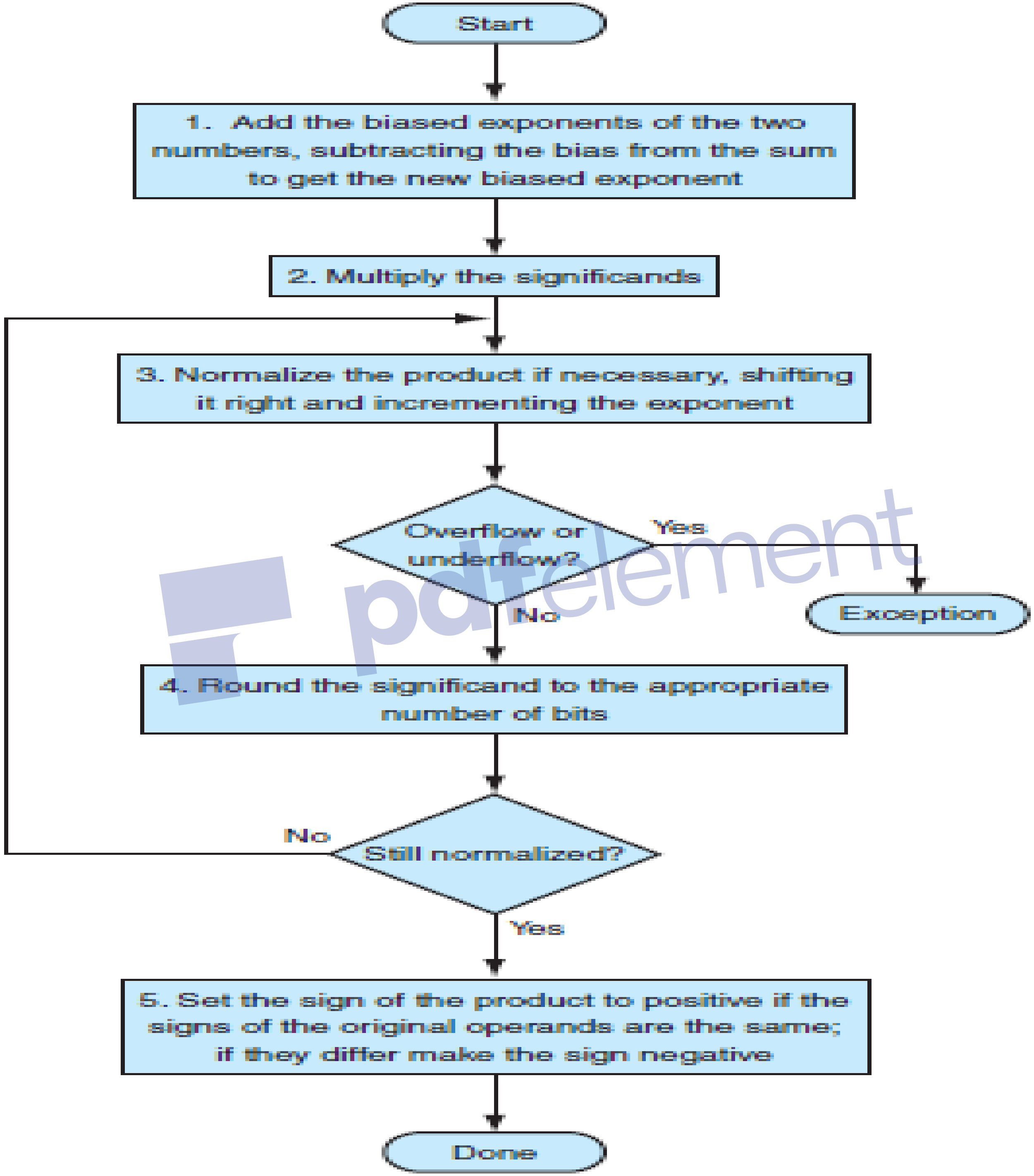
4. Round and renormalize if necessary

   $1.110_2 \times 2^{-3}$ (no change)

5. Determine sign: +ve $\times$ –ve $\Rightarrow$ –ve

   $-1.110_2 \times 2^{-3} = -0.21875$

- 0011 1111 0000 0000 0000 0000 0000 0000
- 1011 1110 1110 0000 0000 0000 0000 0000



Start
1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent
2. Multiply the significands
3. Normalize the product if necessary, shifting it right and incrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
5. Set the sign of the product to positive if the signs of the original operands are the same; if they differ make the sign negative
Done

Problems

1. Add $2.85_{ten} \times 10^3$ to $9.84_{ten} \times 10^4$, assuming that you have only three significant digits.
2. For the same number perform the multiplication also