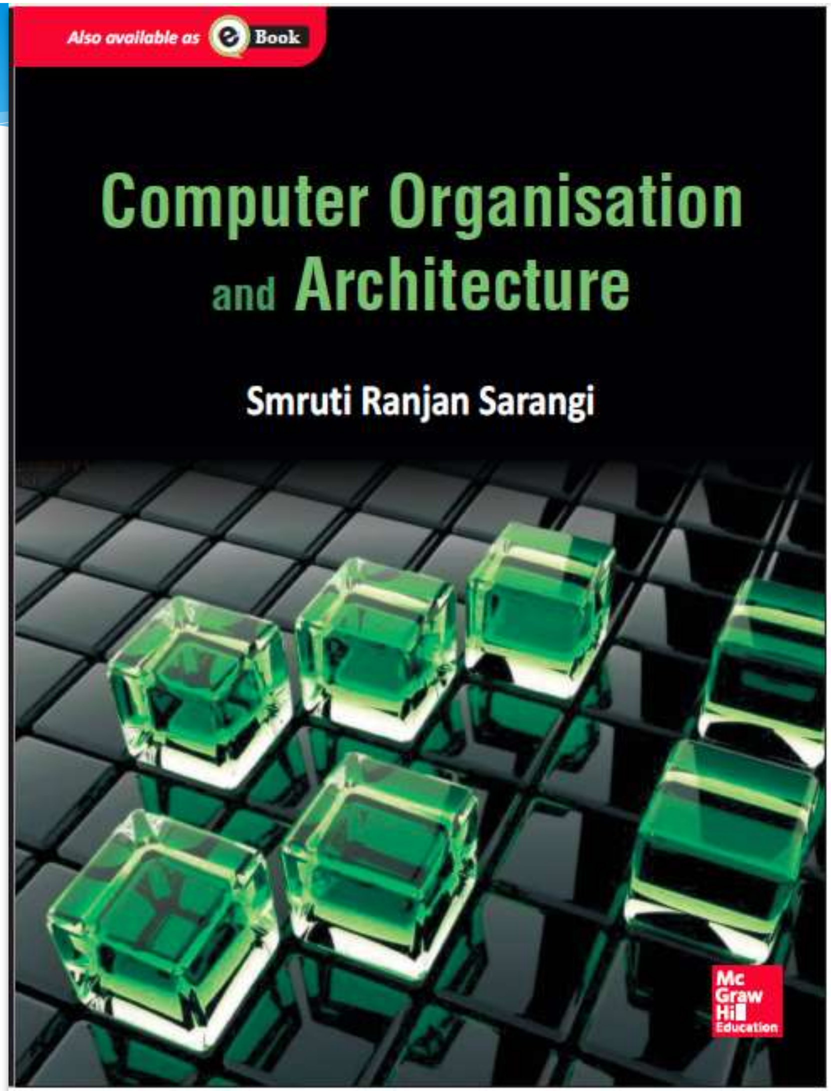
PowerPoint Slides

# Computer Organisation and Architecture


**Smruti Ranjan Sarangi,**
**IIT Delhi**


## Chapter 7 Computer Arithmetic 2




These slides are meant to be used along with the book: Computer Organisation and Architecture, Smruti Ranjan Sarangi, McGrawHill 2015
Visit: http://www.cse.iitd.ernet.in/~srsarangi/archbooksoft.html

# Outline

* Addition
* Multiplication
* Division
* Floating Point Addition
* Floating Point Multiplication
* Floating Point Division

# Integer Division

* Let us only consider positive numbers
  * N = DQ + R
    * N → Dividend
    * D → Divisor
    * Q → Quotient
    * R → Remainder
* Properties
  * [Property 1:] R < D, R >= 0
  * [Property 2:] Q is the largest positive integer satisfying the equation (N = DQ +R) and Property 1

# Reduction of the Divison Problem

$$
\begin{aligned}
N &= DQ + R \\
&= DQ_{1\ \ldots n-1} + DQ_n 2^{n-1} + R
\end{aligned}
$$

$$\underbrace{N - DQ_n 2^{n-1}}_{N\prime} = D\,\underbrace{Q_{1\ldots n-1}}_{Q\prime} + R$$

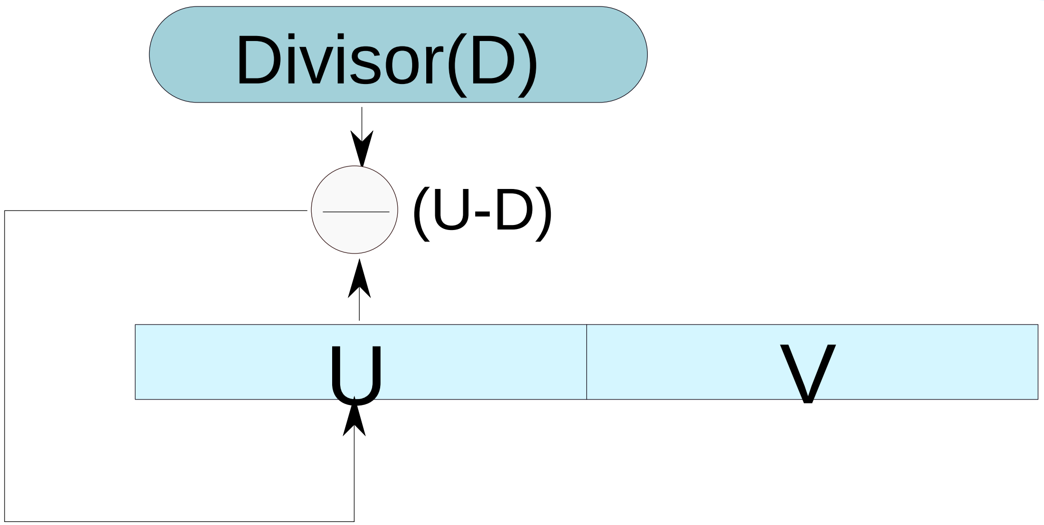
$$N' = DQ' + R$$

We have reduced the original problem to a smaller problem

# How to Reduce the Problem

* We need to find $Q_n$
* We can try both values – 0 and 1
  * First try 1
  * If : $N - D2^{n-1} >= 0$, $Q_n = 1$ (maximize the quotient)
    * Otherwise it is 0
* Once we have reduced the problem
  * We can **proceed recursively**

# Iterative Divider




Initial: V holds the dividend (N), U = 0

# Restoring Division

**Algorithm 3:** Restoring algorithm to divide two 32 bit numbers

**Data**: Divisor in *D*, Dividend in *V*, *U* = 0

**Result**: *U* contains the remainder (lower 32 bits), and *V* contains the quotient

```
i ← 0
for i < 32 do
    i ← i + 1
    /* Left shift UV by 1 position                                  */
    UV ← UV << 1
    U ← U - D
    if U ≥ 0 then
        q ← 1
    end
    else
        U ← U + D
        q ← 0
    end
    /* Set the quotient bit                                         */
    LSB of V ← q
end
```

# Example

| | |
|---|---|
| Dividend (N) | 00111 |
| Divisor (D) | 0011 |

| | | U | V |
|---|---|---|---|
| | beginning: | 00000 | 0111 |
| 1 | after shift: | 00000 | 111X |
| | end of iteration: | 00000 | 1110 |
| 2 | after shift: | 00001 | 110X |
| | end of iteration: | 00001 | 1100 |
| 3 | after shift: | 00011 | 100X |
| | end of iteration: | 00000 | 1001 |
| 4 | after shift: | 00001 | 001X |
| | end of iteration: | 00001 | 0010 |

| | |
|---|---|
| Quotient(Q) | 0010 |
| Remainder(R) | 0001 |

# Restoring Division

* Consider each bit of the dividend
* Try to subtract the divisor from the U register
  * If the subtraction is successful, set the relevant quotient bit to 1
  * Else, set the relevant quotient bit to 0
* Left shift

# Proof

* Let us consider the value stored in UV (ignoring quotient bits)
* After the shift (first iteration)
  * $UV = 2N$
* After line 5, UV contains
  * $UV - 2^n D = 2N - 2^n D = 2 * (N - 2^{n-1} D)$
* If $(U - D) >= 0$
  * $N' = N - 2^{n-1}D.$
  * Thus, UV contains **2N'**

# Proof - II

* If $(U - D) < 0$
  * We know that $(N = N')$
  * Add D to U → Add $2^n D$ to UV
  * partial dividend = $2N = 2N'$
* In both cases
  * The partial dividend = $2N'$
* After 32 iterations
  * V will contain the entire quotient

# Proof - III

* At the end, $UV = 2^{32} * N^{32}$ ($N^{i}$ is the partial dividend after the $i^{th}$ iteration)
* $N^{31} = DQ_1 + R$
  * $N^{31} – DQ_1 = N^{32} = R$
* Thus, U contains the remainder (R)

# Time Complexity

- n iterations
  - Each iteration takes log(n) time
  - Total time : n log(n)

# Restoring vs Non-Restoring Division

* We need to restore the value of register U
  * Requires an extra addition or a register move
* Can we do without this ?
  * Non Restoring Division

**Algorithm 4:** Non-restoring algorithm to divide two 32 bit numbers

**Data**: Divisor in $D$, Dividend in $V$, $U = 0$

**Result**: $U$ contains the remainder (lower 32 bits), and $V$ contains the quotient

```
i ← 0
for i < 32 do
     i ← i + 1
     /* Left shift UV by 1 position                                          */
     UV ← UV << 1
     if U ≥ 0 then
          U ← U − D
     end
     else
          U ← U + D
     end
     if U ≥ 0 then
          q ← 1
     end
     else
          q ← 0
     end
     /* Set the quotient bit                                                 */
     lsb of V ← q
end
if U <0 then
     U ← U + D
end
```

McGraw Hill Education

| | |
|---|---|
| Dividend (N) | 00111 |
| Divisor (D) | 0011 |

| | | U | V |
|---|---|---|---|
| | beginning: | 00000 | 0111 |
| 1 | after shift: | 00000 | 111X |
| | end of iteration: | 11101 | 1110 |
| 2 | after shift: | 11011 | 110X |
| | end of iteration: | 11110 | 1100 |
| 3 | after shift: | 11101 | 100X |
| | end of iteration: | 00000 | 1001 |
| 4 | after shift: | 00001 | 001X |
| | end of iteration: | 11110 | 0010 |
| | end (U=U+D): | 0001 | 0010 |

| | |
|---|---|
| Quotient(Q) | 0010 |
| Remainder(R) | 0001 |

# Idea of the Proof

* Start from the beginning : If $(U - D) >= 0$
  * Both the algorithms (restoring and non-restoring) produce the same result, and have the same state
* If $(U - D) < 0$
  * We have a divergence
  * In the restoring algorithm
    * value(UV) = A
  * In the non-restoring algorithm
    * value(UV) = $A - 2^nD$

# Proof - II

* In the next iteration (just after the shift)
  * Restoring : value(UV) = $2A$
  * Non - Restoring : value(UV) = $2A - 2^{n+1}D$
* If the quotient bit is 1 (end of iteration)
  * Restoring :
    * Subtract $2^nD$
    * value(UV) = $2A - 2^nD$
  * Non Restoring :
    * Add $2^nD$
    * value(UV) = $2A - 2^{n+1}D + 2^nD = 2A - 2^nD$

# Proof - III

* If the quotient bit is 0
  * Restoring
    * partial dividend = 2A
  * Non restoring
    * partial dividend = $2A – 2^nD$
  * Next iteration (if quotient bit = 1) (after shift)
    * Restoring : partial dividend : 4A
    * Non restoring : partial dividend : $4A – 2^{n+1}D$
* Keep applying the same logic ….

# Outline

* Addition
* Multiplication
* Division
* Floating Point Addition
* Floating Point Multiplication
* Floating Point Division

# Adding Two Numbers (same sign)

Normalised form of a 32 bit (normal) floating point number.

$$A = (-1)^S \times P \times 2^{E-bias}, \quad (1 \leq P < 2, E \in \mathbf{Z}, 1 \leq E \leq 254) \qquad (7.22)$$

Normalised form of a 32 bit (denormal) floating point number.

$$A = (-1)^S \times P \times 2^{-126}, \quad (0 \leq P < 1) \qquad (7.23)$$

| Symbol | Meaning |
|---|---|
| $S$ | Sign bit (0(+ve), 1(-ve)) |
| $P$ | Significand (form: 1.xxx(normal) or 0.xxx(denormal)) |
| $M$ | Mantissa (fractional part of significand) |
| $E$ | (exponent + 127(bias)) |
| $\mathbf{Z}$ | Set of integers |

- Recap : Floating Point Number System

# Addition

- Add : A + B
  - Unpack the E fields → $E_A$ , $E_B$
  - Let the E field of the result be → $E_C$
- Unpack the significand (P)
  - P contains → 1 bit before the decimal point, 23 mantissa bits (24 bits)
  - Unpack to a 25 bit number (unsigned)
    - W → Add a leading 0 bit, 24 bits of the signficand

# Addition - II

* With no loss of generality
  * Assume $E_A >= E_B$
* Let significands of A and B be $P_A$ and $P_B$
* Let us initially set $W \leftarrow$ unpack $(P_B)$
* We make their exponents equal and shift W to the right by $(E_A - E_B)$ positions
  *

$$W = W \gg (E_A - E_B)$$
$$W = W + P_A$$

McGraw Hill Education

# Renormalisation

* Let the significand represented by register, W, be $P_W$
  * There is a possibility that $P_W >= 2$
  * In this case, we need to renormalise
    * $W \leftarrow W >> 1$
    * $E_A \leftarrow E_A + 1$
* The final result
  * Sign bit (same as sign of A or B)
  * Significand ($P_W$), exponent field ($E_A$)

# Example

**Example:** Add the numbers: $1.01_2 * 2^3 + 1.11_2 * 2^1$

**Answer:**

The decimal point in *W* is shown for enhancing readability. For simplicity, biased notation not used.

1. $A = 1.01 * 2^3$ and $B = 1.11 * 2^1$
2. $W = 01.11$ (*significand of B*)
3. $E = 3$
4. W = 01.11 >> (3-1) = 00.0111
5. $W + P_A$ = 00.0111 + 01.0100 = 01.1011
6. Result: $C = 1.011 * 2^3$

# Example - II

**Example:** Add : $1.01_2 * 2^3 + 1.11_2 * 2^2$

**Answer:**

The decimal point in *W* is shown for enhancing readability. For simplicity, biased notation not used.

1. $A = 1.01 * 2^3$ and $B = 1.11 * 2^2$
2. $W = 01.11$ (*significand of B*)
3. $E = 3$
4. $W = 01.11 >> (3\text{-}2) = 00.111$
5. $W + P_A = 00.111 + 01.0100 = 10.001$
6. Normalisation: $W = 10.001 >> 1 = 1.0001$, $E = 4$
7. Result: $C = 1.0001 * 2^4$

# Rounding

* Assume that we were allowed only two mantissa bits in the previous example
  * We need to perform rounding
* Terminology :
  * Consider the sum(W) of the significands after we have normalised the result
  * $W \leftarrow (P + R) * 2^{-23} (R < 1)$

# Rounding - II

* P represents the significand of the temporary result
* R (is a residue)
* Aim :
  * Modify P to take into account the value of R
  * Then, discard R
  * Process of rounding : P → P'

# IEEE 754 Rounding Modes

* Truncation
  * $P' = P$
  * Example in decimal : 9.5 → 9, 9.6 → 9
* Round to +∞
  * $P' = \lceil P + R \rceil$
  * Example in decimal : 9.5 → 10, -3.2 → -3

# IEEE 754 Rounding - II

* Round to -∞
  * $P' = \lfloor P+R \rfloor$
  * Example in decimal : 9.5 → 9, -3.2 → -4
* Round to nearest
  * $P' = [P + R]$
  * Example in decimal :
    * 9.4 → 9 , 9.5 → 10 (even)
    * 9.6 → 10 , -2.3 → -2
    * -3.5 → -4 (even)

# Rounding Modes – Summary

| Rounding Mode | Condition for incrementing the significand | |
|---|---|---|
| | Sign of the result (+ve) | Sign of the result (-ve) |
| Truncation | | |
| Round to $+\infty$ | $R > 0$ | |
| Round to $-\infty$ | | $R > 0$ |
| Round to Nearest | $(R > 0.5) \| \| (R = 0.5 \wedge lsb(P) = 1)$ | $(R > 0.5) \| \| (R = 0.5 \wedge lsb(P) = 1)$ |

$\wedge$ (logical AND), $R$ (residue)

# Implementing Rounding

* We need three bits
  * lsb(P)
  * msb of the residue (R) → r (round bit)
  * OR of the rest of the bits of the residue (R) → s (sticky bit)

| Condition on Residue | Implementation |
|---|---|
| $R > 0$ | $r \vee s = 1$ |
| $R = 0.5$ | $r \wedge \overline{s} = 1$ |
| $R > 0.5$ | $r \wedge s = 1$ |
| $r$ (round bit), $s$(sticky bit) | |

# Renormalisation after Rounding

* In rounding : we might increment the significand
  * We might need to renormalise
  * After renormalisation
    * Possible that E becomes equal to 255
    * In this, case declare an overflow

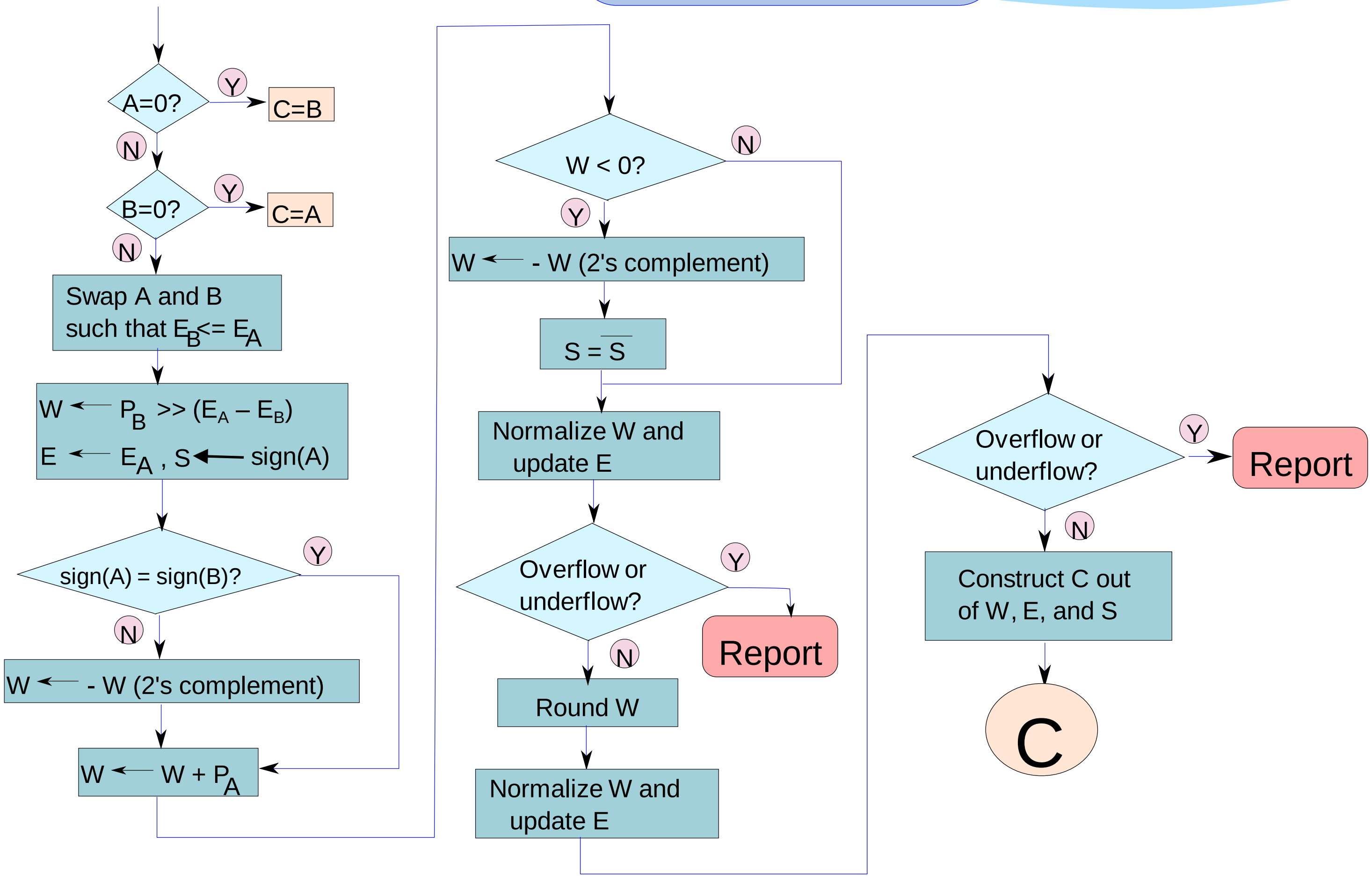
# Addition of Numbers (Opposite Signs)

* $C = A + B$
  * Same assumption $E_A >= E_B$
* Steps
  * Load W with the significand of B ($P_B$)
  * Take the 2's complement of W (W = -B)
  * $W \leftarrow W >> (E_A - E_B)$
  * $W \leftarrow W + P_A$
  * If (W < 0) replace it with its 2's complement. Flip the sign of the result.

# Addition of Numbers (Opposite Signs)-II

* Normalise the result
  * Possible that $W < 1$
  * In this case, keep shifting W to the left till it is in normal form. (simultaneously decrement $E_A$)
* Round and Renormalise

# C = A + B



A=0?
Y
C=B
N
B=0?
Y
C=A
N
Swap A and B such that $E_B$<= $E_A$
W ← $P_B$ >> ($E_A$ – $E_B$)
E ← $E_A$ , S ← sign(A)
sign(A) = sign(B)?
Y
N
W ← - W (2's complement)
W ← W + $P_A$
W < 0?
N
Y
W ← - W (2's complement)
S = $\overline{S}$
Normalize W and update E
Overflow or underflow?
Y
Report
N
Round W
Normalize W and update E
Overflow or underflow?
Y
Report
N
Construct C out of W, E, and S
C

# Outline

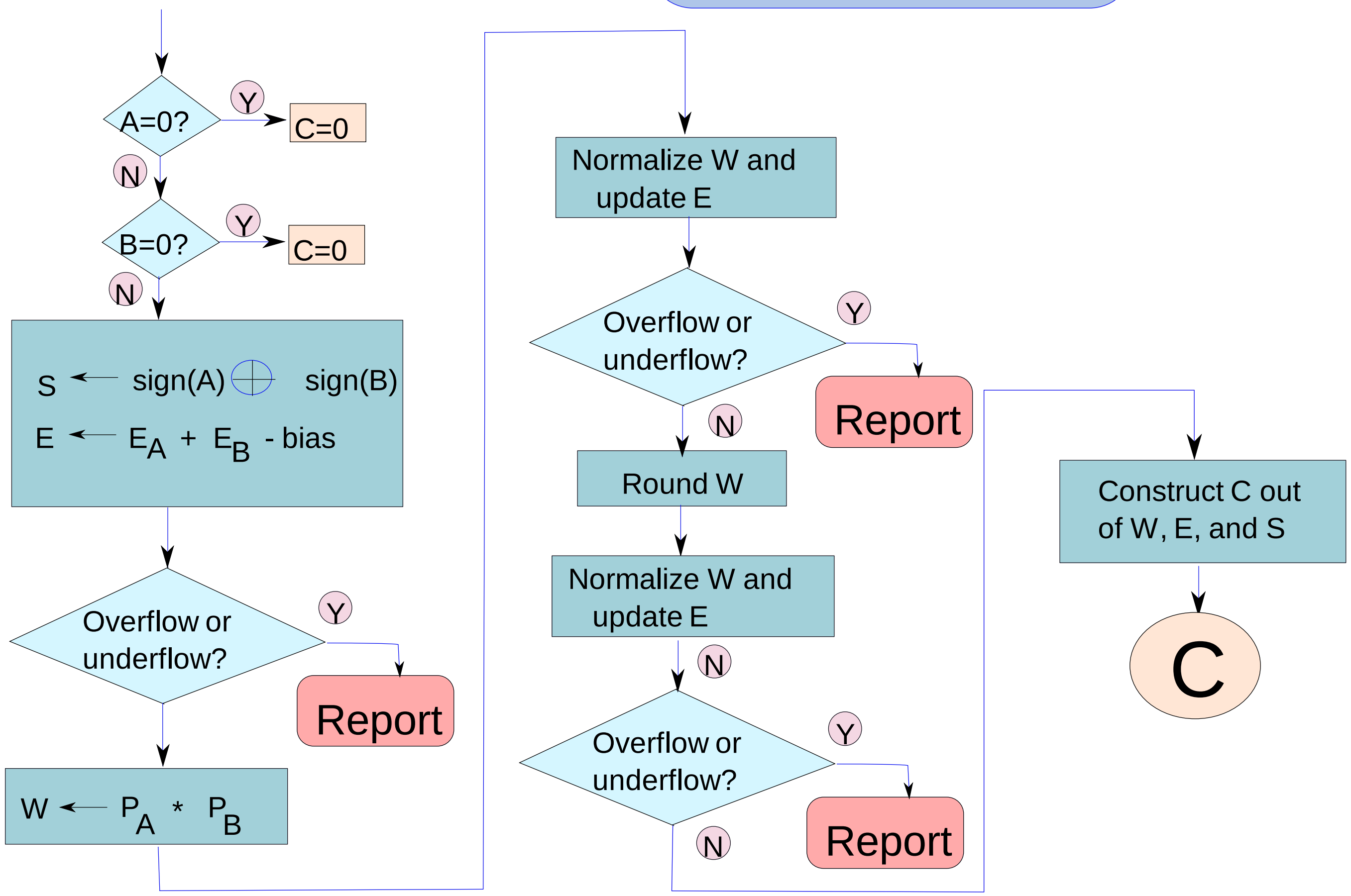
* Addition
* Multiplication
* Division
* Floating Point Addition
* Floating Point Multiplication
* Floating Point Division

McGraw Hill Education

# Multiplication of FP Numbers

* Steps
  * $E \leftarrow E_A + E_B$ - bias
  * $W \leftarrow P_A * P_B$
  * Normalise (shift left or shift right)
  * Round
  * Renormalise

# C = A * B



A=0?
Y
C=0
N
B=0?
Y
C=0
N
S ← sign(A) ⊕ sign(B)
E ← E_A + E_B - bias
Overflow or underflow?
Y
Report
W ← P_A * P_B
Normalize W and update E
Overflow or underflow?
Y
Report
N
Round W
Normalize W and update E
N
Overflow or underflow?
Y
Report
N
Construct C out of W, E, and S
C

# Outline

* Addition
* Multiplication
* Division
* Floating Point Addition
* Floating Point Multiplication
* Floating Point Division

McGraw Hill Education

# Simple Division Algorithm

- Divide A/B to produce C
  - There is no notion of a remainder in FP division
- Algorithm
  - $E \leftarrow E_A - E_B + \text{bias}$
  - $W \leftarrow P_A / P_B$
  - normalise, round, renormalise
- Complexity : O(n log(n))

# Goldschmidt Division

* Let us compute the reciprocal of B (1/B)
  * Then, we can use the standard floating point multiplication algorithm
* Ignoring the exponent
  * Let us compute $(1/P_B)$
* If B is a normal floating point number
  * $1 <= P_B < 2$
  * $P_B = 1 + X \ (X < 1)$

# Goldschmidt Division - II

$$\frac{1}{P_B} = \frac{1}{1+X} \; (P_B = 1 + X, 0 < X < 1)$$

$$= \frac{1}{1 + 1 \; - X'} \; (X' = 1 \; - X, X' < 1)$$

$$= \frac{1}{2 - X'}$$

$$= \frac{1}{2} * \frac{1}{1 \; - \frac{X'}{2}}$$

$$= \frac{1}{2} * \frac{1}{1 \; - Y} \; (Y = \frac{X'}{2}, Y < \frac{1}{2})$$

McGraw Hill Education

$$
\begin{aligned}
\frac{1}{1-Y} &= \frac{1+Y}{1-Y^2} \\
&= \frac{(1+Y)(1+Y^2)}{1-Y^4} \\
&= \ldots \\
&= \frac{(1+Y)(1+Y^2)\ \ldots\ (1+Y^{16})}{1-Y^{32}} \\
&\approx (1+Y)(1+Y^2)\ \ldots\ (1+Y^{16})
\end{aligned}
$$

- No point considering $Y^{32}$
- Cannot be represented in our format

McGraw Hill Education

# Generating the 1/(1-Y)

$$(1+Y)(1+Y^2)\ \ldots\ (1+Y^{16})$$

* We can compute $Y^2$ using a FP multiplier.
  * Again square it to obtain $Y^4$, $Y^8$, and $Y^{16}$
  * Takes 4 multiplications, and 5 additions, to generate all the terms
  * Need 4 more multiplications to generate the final result (1/1-Y)
* Compute $1/P_B$ by a single right shift

# GoldSchmidt Division Summary

* Time complexity of finding the reciprocal
  * $(\log(n))^2$
* Time required for all the multiplications and additions
  * $(\log(n))^2$
* Total Time : $(\log(n))^2$

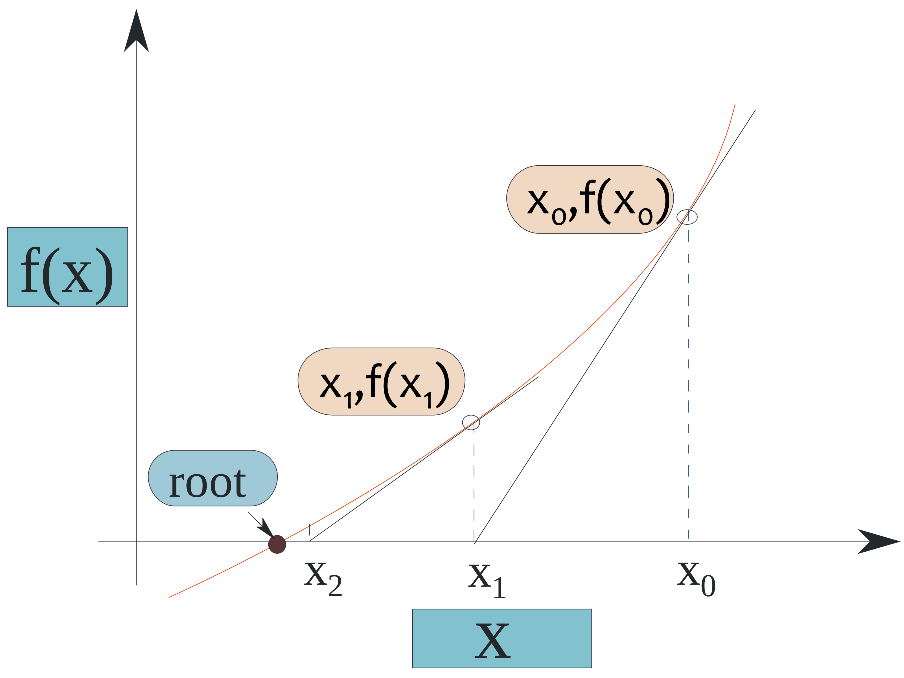
# Division using the Newton Raphson Method

* Let us focus on just finding the reciprocal of a number
* Let us designate $P_B$ as b $(1 <= b < 2)$
  * Aim is to compute 1/b
* Let us create a function $f(x) = 1/x – b$
  * $f(x) = 0$, when $x = 1/b$
* Problem of computing the reciprocal
  * same as computing the root of f(x)

# Idea of the Method

* Start with an arbitrary value of x → $x_0$
  * Locate $x_0$ on the graph of f(x)
  * Draw a tangent to f(x) at ($x_0$ , f($x_0$))
  * Let the tangent intersect the x axis at $x_1$
  * Draw another tangent at ($x_2$, f($x_2$))
* Keep repeating
  * Ultimately, we will converge to the root

# Newton Raphson Method



# Analysis

- $f(x) = 1/x - b$
  - $f'(x) = d\,f(x) / d(x) = -1 / x^2$
  - $f'(x_0) = -1/x_0^2$
- Equation of the tangent : $y = mx + c$
  - $m = -1/x_0^2$
  - $y = -x/x_0^2 + c$
- At $x_0$, $y = 1/x_0 - b$

# Algebra

$$\frac{1}{x_0} - b = -\frac{x_0}{x_0^2} + c$$
$$\Rightarrow \frac{1}{x_0} - b = -\frac{1}{x_0} + c$$
$$\Rightarrow c = \frac{2}{x_0} - b$$

- The equation of the tangent is :
  - $y = -x/x_0^2 + 2/x_0 - b$
  - Let this intersect the x axis at $x_1$

# Intersection with the x-axis

$$-\frac{x_1}{x_0^2} + \frac{2}{x_0} - b = 0$$
$$\Longrightarrow \ x_1 = 2x_0 - bx_0^2$$

- Let us define : E(x) = bx – 1
  - E(x) = 0, when x = 1/b

McGraw Hill Education

# Evolution of the Error

$$\varepsilon\,(x_0) = bx_0\ - 1$$

$$\begin{aligned}
\varepsilon\,(x_1) &= bx_1\ - 1 \\
&= b(2x_0\ - bx_0^2)\ - 1 \\
&= 2bx_0\ -\ b^2x_0^2\ - 1 \\
&= -\,(bx_0\ - 1)^2 \\
&= -\,\varepsilon\,(x_0)^2
\end{aligned}$$

$$|\varepsilon\,(x_1)| =\ |\varepsilon\,(x_0)|^2$$

# Bounding the Error

- $1 <= b < 2$ (significand of a normal floating point number)
  - Let $x_0 = ½$
  - The range of $(bx_0 – 1)$ is [-1/2, 0]
  - Hence, $|E(x_0)| <= ½$
  - The error thus reduces by a power of 2 every iteration

# Evolution of the Error - II

| Iteration | $\max(\varepsilon(x))$ |
|---|---|
| 0 | $\frac{1}{2}$ |
| 1 | $\frac{1}{2^2}$ |
| 2 | $\frac{1}{2^4}$ |
| 3 | $\frac{1}{2^8}$ |
| 4 | $\frac{1}{2^{16}}$ |
| 5 | $\frac{1}{2^{32}}$ |

* E(x) = bx – 1 = b (x – 1/b)
  * x – 1/b is the difference between the ideal value and the actual estimate (x). This is near $2^{-32}$, which is too small to be considered.
* No point considering beyond 5 iterations
* Since, we are limited to 23 bit mantissas

# Time Complexity

* In every step, the operation that we need to perform is :
  * $x_n = 2x_{n-1} - bx_{n-1}^2$
  * Requires a shift, multiply, and subtract operation
  * O(log(n)) time
* Number of steps: O(log(n))
* Total time : $O(\log(n)^2)$

# THE END