EEC 417/517
Embedded Systems
Cleveland State University

Lab 10
Fixed-Point Arithmetic

Dan Simon
Rick Rarick
Spring 2018

Fixed-Point Arithmetic

1. **Fixed-Point Representation**
2. Long integer addition
3. Long integer subtraction
4. Multiplication
5. Division
6. Newton’s Method

Fixed-Point Representation

<u>Fundamental Principle</u>:

The **meaning** of an *n*-bit binary symbol depends entirely on its **interpretation**.

Two common interpretations for 4-bit binary symbols: →

Interpretation: Unsigned Integers

Decimal Representation	4-bit Binary Representation
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Interpretation: Signed Integers

Decimal Representation	Two's Complement Representation
7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
0	0000
– 1	1111
– 2	1110
– 3	1101
– 4	1100
– 5	1011
– 6	1010
– 7	1001
– 8	1000

Fixed-Point Representation

1. Calculators and computers display numbers using a fixed-point or floating-point format.

2. In the **fixed-point** format, numbers have a fixed number of digits, and there are a fixed number of digits to the left and right of the decimal point.

Example: 34.277

Integer Part | Fraction Part

10^4	10^3	10^2	10^1	10^0	10^{-1}	10^{-2}	10^{-3}
0	0	0	3	4	2	7	7

Fixed decimal point

Fixed-Point Representation

1. **Floating-point** numbers adapt the concept of scientific notation to computer systems.

2. In scientific notation, the decimal point can "**float**":

$$
\begin{aligned}
135.67 &= 13567.0 \times 10^{-2} \\
&= 1356.7 \times 10^{-1} \\
&= 135.67 \times 10^{0} \\
&= 13.567 \times 10^{1} \\
&= 1.3567 \times 10^{2} \\
&= 0.13567 \times 10^{3} \\
&= 0.013567 \times 10^{4}
\end{aligned}
$$

Fixed-Point Representation

Fixed-point can be implemented in binary systems also.

Example:

$$1101.0110_2 = 1\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0 + 0\times 2^{-1} + 1\times 2^{-2} + 1\times 2^{-3} + 0\times 2^{-4}$$

$$= 8 + 4 + 0 + 1 + 0 + \frac{1}{4} + \frac{1}{8} + 0$$

$$= 13.375_{10}$$

2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
1	1	0	1	0	1	1	0

Fixed binary point

Fixed-Point Representation

1. Fixed-point numbers can always be represented by integers by choosing an appropriate scaling.

   Decimal example:

   $$134.75 = 13475 \times 10^{-2}$$

2. If we assume an implied, fixed decimal point, we can represent 134.75 by 13475.

3. Fixed-point numbers are often assumed to be integers which have an implied decimal (or binary) point after the right-most digit.

   Binary example:

   $$10010011 \;\rightarrow\; 10010011. \;\; (= 147)$$

Fixed-Point Representation

1. Implied binary point:

   In a computer, all numbers are stored in registers, so 0000.0000 → 00000000, that is, the fixed binary point is **implied**, and the numbers must be **interpreted** by the user
   in software.

2. The **natural binary** representation:

   00000000. → 00000000 and all numbers are assumed to be unsigned integers.

Fixed-Point Arithmetic

1. Fixed-Point Representation
2. **Long integer addition**
3. Long integer subtraction
4. Multiplication
5. Division
6. Newton’s Method

Long Integers Addition

1. The PIC is an 8-bit microcontroller
2. Integers are limited to the range [0, 255]
3. How can we represent a wider range of integers?

Long Integers Addition

1. "Double integers" or "long integers"

```
cblock    0x20
          VarH
          VarL
endc
```

2. Now we have a 16-bit integer VarH:VarL with the 8 MSBs in VarH and the 8 LSBs in VarL.

3. Range = $[0, 2^{16}–1] = [0, 65535]$

Long Integer Addition

	C_1		
	Add1H :		Add1L
+	Add2H :		Add2L
C	SumH :		SumL

C_1 is the carry bit from the lower-byte sum

C is the carry bit from the entire 16-bit sum

Long Integer Addition (Example)

```
       1
      FF FF
+     00 01
-----------
C = 0 00 00
```

1. Add each byte separately. The low byte addition causes an overflow (FF + 01 = 00), so C = 1.

2. Add the carry to the top high byte (FF + 01 = 00). Another overflow occurs, so C = 1.

3. Add the two high bytes: 00 + 00 = 00. No overflow occurs, so C = 0.

4. The register bits are correct but **the carry bit is incorrect**.

5. We must handle the propagation of the carry bit from one byte to the next manually in the code.

Long Integer Addition Algorithm

Add1H : Add1L
Add2H : Add2L

SumH : SumL

C_1 = low-byte carry
C = high-byte carry

FF : FF
<u>00 : 01</u>

SumL = FF
SumH = FF

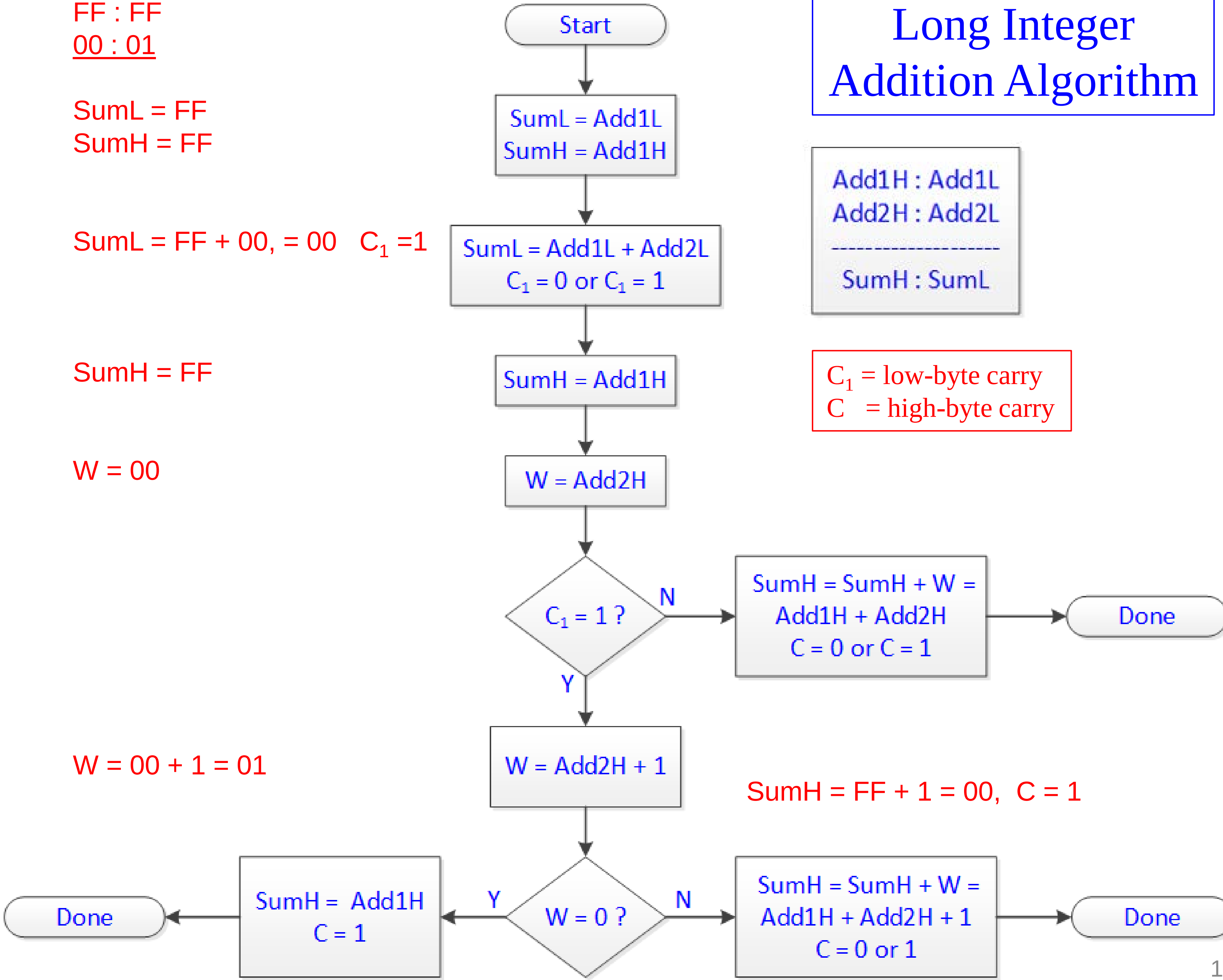
SumL = FF + 00, = 00 C_1 =1

SumH = FF

W = 00

W = 00 + 1 = 01

SumH = FF + 1 = 00, C = 1



Long Integer Addition Code Example

```
; Add1H : Add1L
; Add2H : Add2L
; -------------
;  SumH : SumL

; Initialize variables                 Carry

; 0xFFF8 + 0xF01F = 0x1F017

movlw  0xFF
movwf  Add1H               ; Add1H = 0xFF
movlw  0xF8
movwf  Add1L               ; Add1L = 0xF8

movlw  0xF0
movwf  Add2H               ; Add2H = 0xF0
movlw  0x1F
movwf  Add2L               ; Add2L = 0x1F

call DoubleAdd
```

Long Integer Addition Code

```
DoubleAdd
        ; Add low bytes

        movf    Add1L, W        ; W    = Add1L
        movwf   SumL            ; SumL = Add1L
        movf    Add2L, W        ; W    = Add2L
        addwf   SumL            ; SumL = SumL + W = Add1L + Add2L

        ; Add high bytes

        movf    Add1H, W        ; W    = Add1H
        movwf   SumH            ; SumH = Add1H
        movf    Add2H, W        ; W    = Add2H

        btfsc   STATUS, C       ; C = low-carry. If C = 0,
                                ; SumH = SumH + W
        incfsz  Add2H, W        ; If Add2H + 1 = 0, goto return.

        addwf   SumH, F

        return
```

Fixed-Point Arithmetic

1. Fixed-Point Representation
2. Long integer addition
3. Long integer subtraction
4. Multiplication
5. Division
6. Newton’s Method

Integer Subtraction

$\overline{B}$ = no Borrow = C = carry bit

For addition (addlw, addwf), $\overline{B} = C = 1$ means a carry occurred, so the 8-bit result is not valid.

For subtraction (sublw, subwf), $\overline{B} = C = 1$ means no borrow occurred, so 8-bit result is valid.

Subtract a number by adding its two's comp.

Example: 2 – 1 = 2 + (–1) = 2 + (two's comp. of 0000 0001)

$$
\begin{array}{ll}
 & 0000\ 0010 \\
+ & \underline{1111\ 1111} \\
1 & 0000\ 0001
\end{array}
$$

$\overline{B} = 1$, so the 8-bit result (+1) is valid

Integer Subtraction

Example: 1 – 2 = 1 + (two’s comp. of 0000 0010)

$$
\begin{array}{ll}
 & 0000\ 0001 \\
+ & \underline{1111\ 1110} \\
0 & 1111\ 1111
\end{array}
$$

$\bar{B} = 0$, so the 8-bit result (255) is not valid.

Note that the result (1111 1111) is the 8-bit two’s comp. of 1, that is, –1.

So the result is valid if it is interpreted as a two's comp. number.

Integer Subtraction

Example: 1 – 255 = 1 + two’s comp. of 1111 1111

$$
\begin{array}{ll}
 & 0000\ 0001 \\
+ & \underline{0000\ 0001} \\
0 & 0000\ 0010
\end{array}
$$

$\overline{B} = 0$ so the result is not valid.

The result is not the 8-bit two’s comp. of 254 because –254 does not have an 8-bit two’s complement representation. The subtraction is outside the valid range for 8-bit 2's complement numbers (– 128 to +127).

Long Integer Subtraction

$$
\begin{array}{cll}
 & \overline{B}_1 & \\
 & \text{Sub1H} & \text{Sub1L} \\
- & \text{Sub2H} & \text{Sub2L} \\
\hline
\overline{B} & \text{DiffH} & \text{DiffL}
\end{array}
$$

$\overline{B}_1$ is the borrow bit from the lower-byte subtraction

$\overline{B}$ is the borrow bit from the entire 16-bit subtraction

Long Integer Subtraction

$$\begin{array}{rr} & \overline{B}_1 \\ & 00\ \ 00 \\ - & \underline{FF\ \ 01} \\ \overline{B}=1 & 00\ \ FF \end{array}$$

A borrow occurred on the low-byte subtraction, so $\overline{B}_1 = 0$ (borrow). The high-byte subtraction becomes FF – FF = 00 and so $\overline{B} = 1$ (no borrow).

$\overline{B} = 1$ means no borrow, which is incorrect.

STATUS< C > = 1 is not set correctly because the borrow from the lower-byte subtraction caused the underflow .

(00 – 1 $\rightarrow$ FF).

Long Integer Subtraction

```
; Sub1H : Sub1L
; Sub2H : Sub2L
; --------------
;  DiffH : DiffL

; Initialize variables

; 0xFFF8 - 0xF01F = 0x0FD9

movlw  0xFF          ; Load 0xFFF8
movwf  Sub1H
movlw  0xF8
movwf  Sub1L

movlw  0xF0          ; Load 0xF01F
movwf  Sub2H
movlw  0x1F
movwf  Sub2L

call   DoubleSub
```

Long Integer Subtraction

```
DoubleSub
       ; Subtract low bytes
       movf   Sub1L, W      ; W     = Sub1L
       movwf  DiffL         ; DiffL = Sub1L
       movf   Sub2L, W      ; W     = Sub2L
       subwf  DiffL         ; DiffL = DiffL - W = Sub1L -
                            ; Sub2L

       ; Subtract high bytes
       movf          Sub1H, W
       movwf         DiffH
       movf          Sub2H, W

       btfss         STATUS, C ; Low Borrow check
       incfsz        Sub2H, W  ; If C = 1, no borrow
       subwf         DiffH,F
       return
```

Fixed-Point Arithmetic

1. Fixed-Point Representation
2. Long integer addition
3. Long integer subtraction
4. Multiplication
5. Division
6. Newton's Method

Integer Multiplication

Simple pseudo-code for $C = A \times B$ (integer multiplication)

$C = \underbrace{A + A + \ldots + A}_{B \text{ copies}}$ (repeated addition):

```
         C = 0
Loop:    if B = 0 then return
         C = C + A
         B = B – 1
         go to Loop
```

Problem: Execution time depends on B.

(0×255) takes 255 times as long as (255×0)

Integer Multiplication

(1) Multiplication: $C = A \times B$

$B = b_{n-1}\, b_{n-2} \ldots b_1\, b_0$

$C = A \times (b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \ldots + b_1 2^1 + b_0 2^0)$

$\quad = b_{n-1}\, A2^{n-1} + b_{n-2}\, A2^{n-2} + \ldots + b_1\, A2^1 + b_0\, A2^0$

(2) Pseudocode**:**

$C = 0$

if $b_0 = 1$ then $C = C + A \times 2^0$

if $b_1 = 1$ then $C = C + A \times 2^1$

if $b_2 = 1$ then $C = C + A \times 2^2$

if $b_3 = 1$ then $C = C + A \times 2^3$

. . . .

Integer Multiplication

(3) Equivalent Pseudocode:

C = 0

A_temp = A

if $b_0 = 1$ then C = C + A_temp
A_temp= 2 × A_temp ; A_temp = $A \times 2^1$

if $b_1 = 1$ then C = C + A_temp
A_temp = 2 × A_temp ; A_temp = $A \times 2^2$

if $b_2 = 1$ then C = C + A_temp
A_temp = 2 × A_temp ; A_temp = $A \times 2^3$

if $b_3 = 1$ then C = C + A_temp
. . .

Integer Multiplication

(4) Equivalent Pseudocode:

; A, B are 8-bit integers

; C, A_temp are 16-bit integers

C = 0

A_temp = A

for i = 0 to 7

    if b_i = 1 then C = C + A_temp

    A_temp = 2 × A_temp

next i

Integer Multiplication

Possible Enhancements:

- Overflow bit
- Multiplication of negative numbers
  (two's complement multiplication)

Fixed-Point Arithmetic

1. Fixed-Point Representation
2. Long integer addition
3. Long integer subtraction
4. Multiplication
5. Division
6. Newton's Method

Integer Division

Example: 111001 / 110:   57 / 6 = 9, Remainder 3

```
         001001   remainder 11
110     |111001
         110
           1001
            110
             11
```

Integer Division

$$\begin{array}{r|l} & C_3C_2C_1C_0 \\ \hline B & A_5A_4A_3A_2A_1A_0 \end{array}$$

$$\begin{array}{r|l} & \;\;\;1001 \text{ rem } 11 \\ \hline 110 & 111001 \\ & \underline{110} \\ & \;\;\;1001 \\ & \;\;\;\;\underline{110} \\ & \;\;\;\;\;\;11 \end{array}$$

1. Put MSBs of A into Temp, one at a time, until Temp $\geq$ B
   Temp = 111
   Temp $\geq$ B (111 $\geq$ 110), so $C_3 = 1$
   Temp = Temp – B = 111 – 110 = 1

2. Include next MSB of A (A_2) in Temp
   Temp = 10
   Temp < B (10 < 110) so $C_2 = 0$

3. Include next MSB of A (A_1) in Temp
   Temp = 100
   Temp < B (100 < 110) so $C_1 = 0$

4. Include next MSB of A (A_0) in Temp
   Temp = 1001
   Temp $\geq$ B (1001 $\geq$ 110) so $C_0 = 1$

5. Remainder = Temp – B = 11

Integer Division

Pseudocode from previous page:

1. Put MSBs of A into Temp, one at a time, until Temp $\geq$ B

2. Set $C_j = 1$
   Temp = Temp – B

3. For all remaining bits i from (j – 1) to 0
   Include next A_i in Temp

   if Temp < B then
   $C_i = 0$
   else
   $C_i = 1$
   Temp = Temp – B
   end if

   Next i

C = A / B algorithm (8-bit):

```
Temp = C = 0
for i = 7 to 0
    Temp = 2 × Temp + A_i

    if (Temp ≥ B) then
        C_i = 1
        Temp = Temp – B
    end if

next i
```

Note that after the algorithm is done, Temp contains the remainder

Integer Division

<u>$C = A / B$ algorithm:</u>

N = # of bits in A and B
$Temp = C = 0$

for $i = (N - 1)$ to 0
    $Temp = 2 \times Temp + A_i$

    if ($Temp \geq B$) then
        $C_i = 1$
        $Temp = Temp - B$
    end if

next i

<u>Possible Enhancements:</u>

- Rounding the remainder
- Division by zero
- Division of negative numbers (2's comp. division)

Microchip’s Math Routines

- Microchip has an application note (AN617) for fixed-point arithmetic routines for the PIC16F877.
- There is also an application note (AN544) for fixed-point, BCD conversion, and random number generators for the PIC17C42. These can be adapted for the PIC16F877 if required.
- These application notes and asm files can be found in the Resources page on the course website.

Fixed-Point Arithmetic

1. Fixed-Point Representation
2. Long integer addition
3. Long integer subtraction
4. Multiplication
5. Division
6. Newton's Method

Newton's Method

- Problem: Find x such that $f(x) = a$, where a is a known constant.

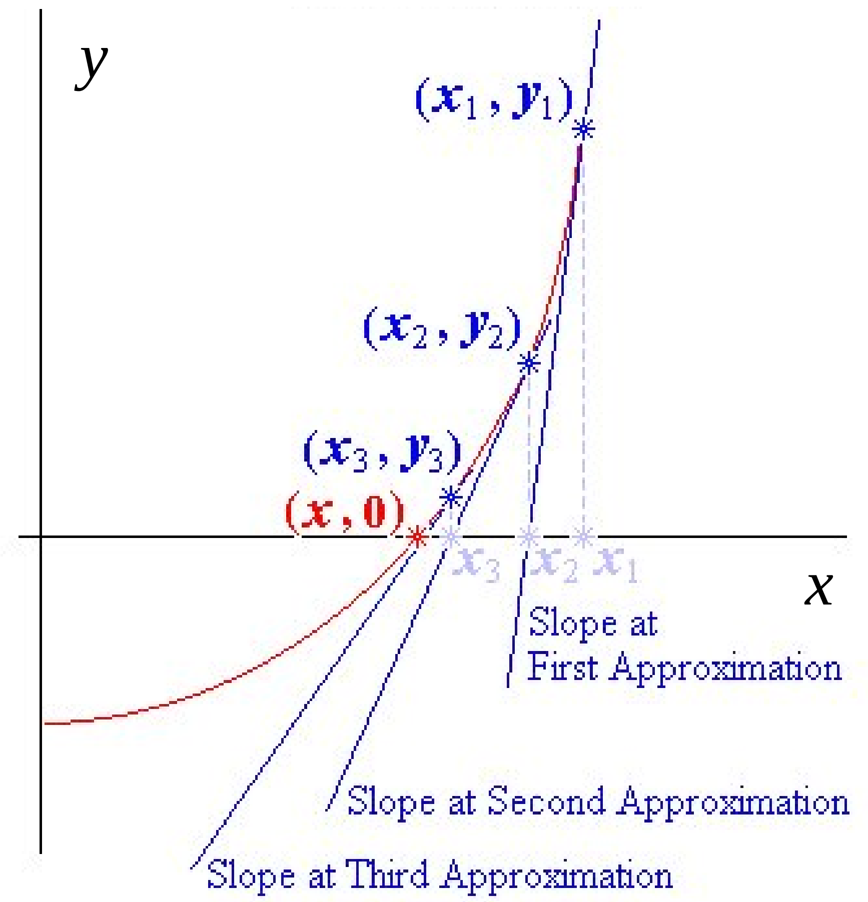
- For example, a square root algorithm can be implemented by finding x such that $x^2 = a$

- Define $y(x) = f(x) - a$

  The above problem can be written as:

  Find x such that $y(x) = 0$

Newton's Method



Guess a solution to $y(x) = 0$
First guess = x_1

$$y'(x_1) = y(x_1) / (x_1 - x_2)$$

So, $x_2 = x_1 - y(x_1) / y'(x_1)$

$$y'(x_2) = y(x_2) / (x_2 - x_3)$$

So, $x_3 = x_2 - y(x_2) / y'(x_2)$

…

$$x_{n+1} = x_n - y(x_n) / y'(x_n)$$

When x_n "stops changing," or $y(x_n) \approx 0$, we are done.

Note: You must have a division routine to use Newton's Method.

Newton's Method

- Example: Find the square root of a

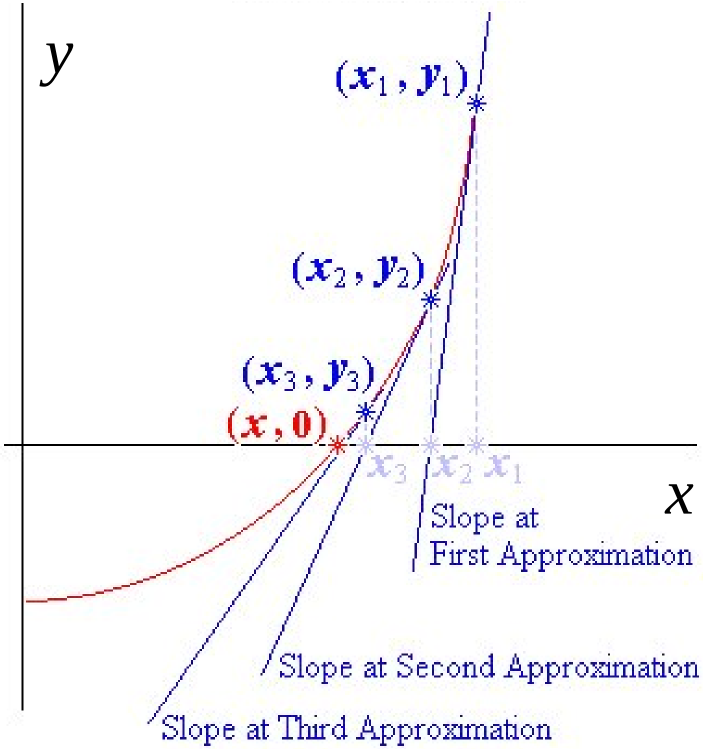
  Find x such that $x^2 = a$
  Find x such that $y(x) = x^2 - a = 0$

- Newton's method:

  $y'(x) = 2x$

  $x_{n+1} = x_n - y(x_n) / y'(x_n)$

  $= x_n - (x_n^2 - a) / (2x_n)$



- Names:
  - Recursive method
  - Iterative method
  - Numerical method

What if $x_n = 0$?
What if $y'(x_n) = 0$?

Newton's Method

Example: Find the square root of 15

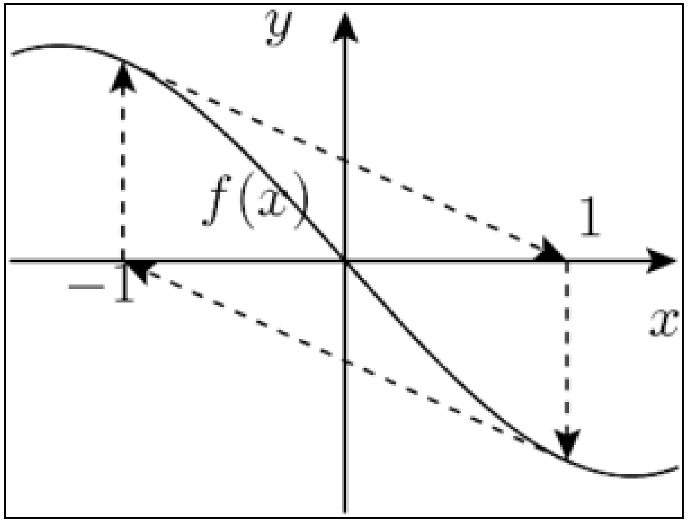
- For square root, $x_{n+1} = (x_n + 15 / x_n) / 2$
- Use integer arithmetic
- Assume no rounding in division routine

1. $x_1 = 3$ (Guess)
2. $x_2 = 4$
3. $x_3 = 3$
4. . . .

Rounding will take care of the problem *for this example*

Newton's Method

May fail to converge
on some functions.



Newton's Method

- Advantages
  - Simple and straightforward
  - Can be used with many different functions
- Disadvantages
  - May not converge
  - Number of iterations is not predictable
- Alternatives to Newton's method
  - Table lookup
  - Closed-form algorithms

Closed-Form Binary Square Root

<u>Root = square root of A:</u> N = number of bits in A (assume N is even)

Temp = A
Rem = 2 MSBs of Temp
if Rem = 0 then Dig = 0 else Dig = 1
Rem = Rem – Dig × Dig
Root = 0

for i = 1 to N/2
    Root = 2 × Root + Dig
    Temp = 4 × Temp
    Rem = 4 × Rem + (2 MSBs of Temp)
    Divisor = 4 × Root

    if (Divisor + 1) ≤ Rem then Dig = 1 else Dig = 0

    Rem = Rem – Dig × (Divisor + Dig)

next i

> *Math Toolkit for Real-Time Programming*, by Jack Crenshaw
>
> "Integer Square Roots," by Jack Crenshaw, www.embedded.com

lab10.asm

1. No hardware required.

2. Run in simulation mode. Select Debugger/ Select Tool / MPLAB SIM

3. Use a breakpoint at various locations to verify operations

4. Use a breakpoint to demonstrate to Instructor/TA .

End of Lab 10