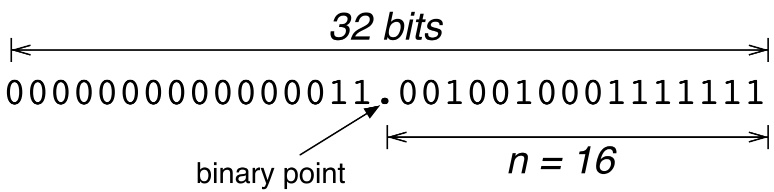
Fixed Point Arithmetic With Scaled Integers

CS 330

August 23, 2010

Fixed Point Numbers as Scaled Integers

- We can a approximate a *real number* x by storing it as a (binary) integer scaled by 2^n where n is the number of bits representing the fractional portion of the number.
- Approximating π by storing $\lfloor \pi \times 2^{16} \rfloor$ as a 32-bit integer. The *binary point* is fixed between bit 15 and 16 (Q16.16):



(review binary and hex number systems; 2's complement)

Motivation

- May not have access to FPU (e.g., embedded DSP).
- Integer operations faster than FPU.
- Consistent results between platforms.
- Unique value of zero (using 2's complement).
- Common in embedded systems

Downside

- Numbers need to be of roughly the same order of magnitude.
- FPU's may provide more operations (e.g., `sin`, `cos`, `exp`, `sqrt`).
- *More tedious:* need to manually manage overflow and rescale numbers after multiplication and division.
- No representation for $\pm\infty$ or `NaN`'s.

Decimal and Binary Fractions

frac	*decimal*	*binary*
1/1	1 or 0.9999 . . .	1 or 0.1111 . . .
1/2	0.5	0.1
1/3	0.3333 . . .	0.010101. . .
1/4	0.25	0.01
1/5	0.2	0.00110011 . . .
1/6	0.1666. . .	0.0010101 . . .
1/7	0.142857142857 . . .	0.001001. . .
1/8	0.25	0.001
1/9	0.111 . . .	0.000111000111 . . .
1/10	0.1	0.000110011...

Note: x/n **terminates in binary iff** $n = 2^i$

1/10 in Fixed Point

$$\begin{aligned}
0.1_{10} &\approx 0.0001100110011001_2 \\
&= 0.F999_{16} \\
&= 0.0999908447_{10} \\
\text{error} &\approx 9.15 \times 10^{-6} \\
0.1 \times 10 &= 0.1 \times (2^3 + 2^1) \\
&\approx 0.1100110011001\underline{000} + \\
&\quad\; 0.001100110011001\underline{0} \\
&= 0.1111111111111010
\end{aligned}$$

Addition and Subtraction

$$
\begin{aligned}
(x \times 2^n) + (y \times 2^n) &= (x + y) \times 2^n \\
(x \times 2^n) - (y \times 2^n) &= (x - y) \times 2^n
\end{aligned}
$$

- We assume binary point at same spot.
- Straight forward use of integer addition and subtraction.

$$\pi + e$$

3.141586 + 2.718277 = 5.859863

$$
\begin{array}{r}
0000000000000011.0010010000111111 \\
+\ 0000000000000010.1011011111100001 \\
\hline
0000000000000101.1101110000100000
\end{array}
$$

$$
\begin{array}{r}
0003.243F_{16} \\
+\,0002.B7E1_{16} \\
\hline
0005.DC20_{16}
\end{array}
$$

Multiplying Two Fixed-Point Numbers

$$(x \times 2^n) \times (y \times 2^n) \;=\; (x \times y) \times 2^{2n}$$

- Note that resulting scaling factor requires twice as many bits for the fraction!
- Will overflow when $|x \cdot y| \geq 1$.
- For our 32-bit integers ($n = 16$) we capture the result into a 64-bit number:

$$\left(x \times 2^{16}\right) \times \left(y \times 2^{16}\right) \;=\; (x \times y) \times 2^{32}$$

  - Upper 32 bits contain *whole portion;*
  - Lower 32 bits contain *fraction*.

$$\pi \times e$$

```
3.141586 * 2.718277 = 8.539701
```

$$\begin{array}{r} \texttt{0003.243F}_{16} \\ *\,\texttt{0002.B7E1}_{16} \\ \hline \underbrace{\texttt{00000008}}_{\substack{\text{upper 32-bits} \\ \text{whole part}}}.\underbrace{\texttt{8A29E45F}}_{\substack{\text{lower 32-bits} \\ \text{fraction}}}{}_{16} \end{array}$$

rescale to 32-bits	$\texttt{0008.8A29}_{16}$
round to nearest	$\texttt{0008.8A2A}_{16}$

Rescale after multiplication

- A 32-bit machine will save the 64-bit result into two 32-bit registers:

```
movl   $0x0003243F, %eax  /* EAX <-- pi */
movl   $0x0002B7E1, %ebx  /* EBX <-- e */
imull  %eax, %ebx         /* EDX:EAX <-- pi * e */
```

- Rescale to 32-bits

```
shll   $16, %edx          /* EDX <<= 16 */
shrl   $16, %eax          /* EAX >>= 16 */
orl    %edx, %eax         /* EAX = 32-bit result */
```

- Not transparent to C. Instead we force the compiler to use 64-bit a multiply and rescale:

```
((int64_t) pi * e) >> 16;
```

Division of two Fixed-Point Numbers

- Naïve division

$$(x \times 2^n) \div (y \times 2^n) \;=\; (x \div y) \times 2^0$$

  We have zero bits of fraction left!

- We first scale the dividend by 2^n

$$\left(x \times 2^{2n}\right) \div (y \times 2^n) \;=\; (x \div y) \times 2^n$$

- For our 32-bit fixed-point numbers we scale the dividend by 2^{16} and store it as a 64-bit number.

$$\left(x \times 2^{32}\right) \div \left(y \times 2^{16}\right) \;=\; (x \div y) \times 2^{16}$$

$$\pi \div e$$

$$3.141586 \div 2.718277 = 1.155727$$

$$\begin{array}{r}
\overbrace{\texttt{00000003}}^{\text{upper 32-bits whole part}}\texttt{.}\overbrace{\texttt{243F}\underbrace{\texttt{0000}}_{\text{zero fill}}}^{\text{lower 32-bits fraction}}{}_{16} \\
\div\,\texttt{0002.B7E1}_{16} \\
\hline
\texttt{0001.27DD}_{16}
\end{array}$$

Pre-scale dividend before division

- For a machine with 32-bit registers we store the 64-bit dividend in two registers for a 32-bit divide operation:

```
movl   $0x0003243F, %eax  /* EAX <-- pi */
movl   $0x0002B7E1, %ebx  /* EBX <-- e */
movl   %eax, %edx
sarl   $16, %edx     /* EDX <-- whole part of pi */
sall   $16, %eax     /* EAX <-- fraction of pi   */
idivl  %ebx          /* EAX <-- quotient pi / e  */
```

- Not transparent to C. Instead we scale the dividend by 2^{16} and perform a 64-bit divide:

```
((int64_t) pi << 16) / e
```

- Digits of precision for 16-bit fraction:

$$\begin{aligned} 2^{16} &= 10^d \\ \log 2^{16} &= \log 10^d \\ 16 \log 2 &= b \log 10 \\ d &= 16 \frac{\log 2}{\log 10} \approx 4.816 \text{ decimal digits} \end{aligned}$$

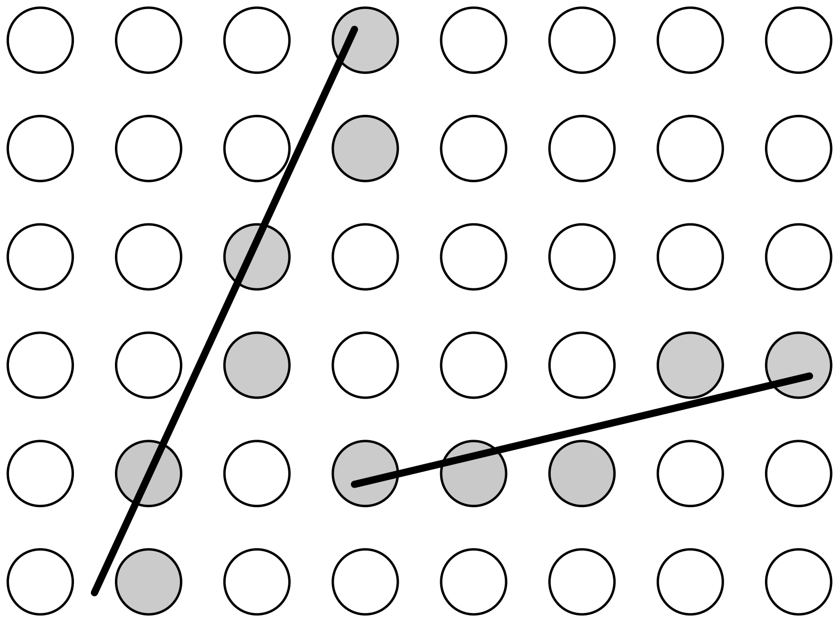
- $\epsilon =$ smallest positive integer we can store

$$\epsilon = 2^{-16} \approx 0.00001526 = 0.1526 \times 10^{-4}$$

- Minimum and Maximum

$$\begin{aligned} 8000.0000_{16} &= -2^{15} = -32,768_{10} \\ \text{7FFF.FFFF}_{16} &= 2^{15} - \epsilon \approx 32,767.9999847_{10} \end{aligned}$$

Fixed Point Line Rasterization



drawLine(x_0, y_0, x_1, y_1)

$\Delta y = y_1 - y_0$
$\Delta x = x_1 - x_0$
if $|\Delta y| \geq |\Delta x|$ *(vertical)*
$\quad m = \Delta x / \Delta y$
$\quad y_{\text{start}} = \text{round}(y_0)$
$\quad y_{\text{end}} = \text{round}(y_1)$
$\quad x = x_0$
$\quad$ **for** $y = y_{\text{start}} \cdots y_{\text{end}}$
$\qquad \text{setpixel}(\text{round}(x), y)$
$\qquad x = x + m$
else . . . *(horizontal)*

drawLine(int x0, int y0, int x1, int y1)
(fixed-point coordinates, $n = 16$)

```
int dy = y1 - y0;
int dx = x1 - x0;
if (abs(dy) > abs(dx)) {
    int m = ((int64_t) dx << 16) / dy;
    int y = (y0 + 0x08000) >> 16;
    int yend = (y1 + 0x08000) >> 16;
    int yinc = (dy < 0) ? -1 : +1;
    int x = x0 + 0x08000;
    for (; y != yend; y += yinc) {
        setpixel(x >> 16, y);
        x += m;
    }
} else ...
```