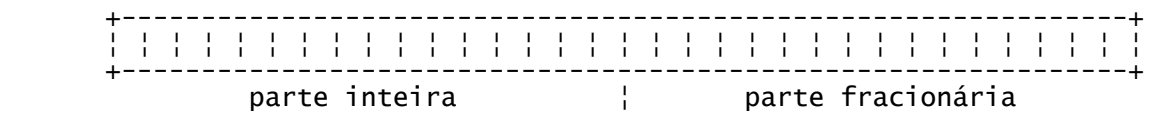
+-------------+
¦ ASSEMBLY XX ¦
+-------------+

Impressionante como as demonstraçöes gráficas (DEMOS) conseguem ser tão rápidas com todas aquelas transformaçöes geométricas (objetos movimentando-se no espaço tridimensional), musicas em background, etc... A complexidade sugere a utilização de rotinas em ponto-flutuante para os calculos "cabeludos"... Opa! Ponto-flutuante?! Mas isso é muito lerdo!!!! Toma muito tempo de CPU... E nem sempre o feliz proprietário de um microcomputador tem um 486DX ou um 386 com co-processador! Como é que esses caras conseguem tanta velocidade?!

A resposta pode estar num método conhecido como "aritimética de ponto-fixo", que é o objetivo deste texto!

Imagine que possamos escrever um número "quebrado" (com casas decimais) da seguinte maneira:



A "casa" mais a esquerda é o bit mais significativo, e a mais a direita o menos significativo. Assim os 16 bits mais significativos (parte inteira) nos diz a "parte inteira" do número (lógico, né?). E os 16 bits menos significativos (parte fracionária) nos diz a parte fracionária do número (outra vez, lógico!). De forma que o bit menos significativo destes 32 bits é equivalente a 2 elevado a potência de -16 (ou seja: 1/65536). Eis um exemplo:

```
 +-----------------------------------------------------------------+
 ¦  0000000000000000.1000000000000000b = 0.5 = 1/2                 ¦
 ¦  0000000000000000.0100000000000000b = 0.25 = 1/4                ¦
 ¦  0000000000000000.0010000000000000b = 0.125 = 1/8               ¦
 ¦  0000000000000000.1110000000000000b = 0.875                     ¦
 ¦  0000000000000001.1000000000000000b = 1.5 = 1 + 1/2             ¦
 ¦  0000000000000011.0010010000111111b =
 (aprox.)                  ¦
 ¦  0000000000000000.1101110110110011b ¸ cos(
/6) ¸ 0.866 (aprox.) ¦
 +-----------------------------------------------------------------+
```

Não sei se deu para entender, mas do bit menos significativo até o mais significativo, o expoente vai aumentando, só que o bit menos significativo tem expoente -16. Assim, o bit 1 tem expoente -15, o seguinte -14, etc... até o último, 15. O ponto entre os dois conjuntos de 16 bits foi adicionado apenas para facilitar a visualização no exemplo acima.

Ok... entäo é possível representar "números quebrados" em dois conjuntos de 16 bits... a pergunta é: Pra que?!

Aritimética com números inteiros sempre é mais rápida do que a aritimética com números em ponto-flutuante. Tendo co-processador ou näo! Mesmo que vc tenha um 486DX4 100MHz, os calculos em ponto-flutuante serão mais lerdamente efetuados do que os mesmos calculos com números inteiros (usando os registradores da CPU!). Neste ponto entra a aritimética de ponto-fixo (note que o "ponto decimal" não muda de posição...). Vejamos o que acontece se somarmos dois números em ponto fixo:

```
+-----------------------------------------------------------------+
|  0.25 + 1.75 = 2.0                                              |
|                                                                 |
|    0000000000000000.0100000000000000b =    0.25                 |
|  + 0000000000000001.1100000000000000b =  + 1.75                 |
|  ----------------------------------   --------                  |
|    0000000000000010.0000000000000000b =    2.00                 |
+-----------------------------------------------------------------+
```

Realmente simples... é apenas uma soma binária... Suponha que tenhamos um número em ponto fixo no registrador EAX e outro no EDX. O código para somar os dois números ficaria tão simples quanto:

```
+-----------------------------------------------------------------+
|  ADD     EAX,EDX                                                |
+-----------------------------------------------------------------+
```

O mesmo ocorre na subtração... Lógicamente, a subtração é uma adicão com o segundo operando complementado (complemento 2), então não há problemas em fazer:

```
+-----------------------------------------------------------------+
|  SUB     EAX,EDX                                                |
+-----------------------------------------------------------------+
```

A adição ou subtração de dois números em ponto fixo consome de 1 a 2 ciclos de máquina apenas, dependendo do processador... o mesmo não ocorre com aritimética em ponto-flutuante!

A complicação começa a surgir na multiplicação e divisão de dois números em ponto-fixo. Não podemos simplesmente multiplicar ou dividir como fazemos com a soma:

```
+-----------------------------------------------------------------+
|    0000000000000001.0000000000000000                            |
|  * 0000000000000001.0000000000000000                            |
|  ----------------------------------                             |
|    0000000000000000.0000000000000000 + carry                    |
+-----------------------------------------------------------------+
```

Nultiplicando 1 por 1 deveriamos obter 1, e não 0. Vejamos a multiplicação de dois valores menores que 1 e maiores que 0:

```
+-----------------------------------------------------------------+
|    0000000000000000.1000000000000000     0.5                   |
|  * 0000000000000000.1000000000000000   * 0.5                    |
|  ---------------------------------- -------                     |
|    0100000000000000.0000000000000000  16384.0                   |
+-----------------------------------------------------------------+
```

Hummm... o resultado deveria dar 0.25. Se dividirmos o resultado por 65536 (2^16) obteremos o resultado correto:

```
+-----------------------------------------------------------------+
|    0100000000000000.0000000000000000 >> 16 =                    |
|    0000000000000000.0100000000000000       = 0.25               |
+-----------------------------------------------------------------+
```

Ahhh... mas, e como ficam os números maiores ou iguais a 1?! A instrução IMUL dos microprocessadores 386 ou superiores permitem a multiplicação de dois inteiros de 32 bits resultando num inteiro de 64 bits (o resultado ficará em dois registradores de 32 bits separados!). Assim, para multiplicarmos dois números em ponto fixo estabelecemos a seguinte regra:

```
resultado = (n1 * n2) / 65536          ou
resultado = (n1 * n2) >> 16
```

Assim, retornando ao primeiro caso de multiplicação (em notação hexa agora!):

```
0001.0000h * 0001.0000h = 000000010000.0000h

Efetuando o shift de 16 bits para a direita:

00010000.0000h >> 16 = 0001.0000h
```

Em assembly isso seria täo simples como:

```
PROC    FixedMul
ARG     m1:DWORD, m2:DWORD

    mov     eax,m1
    mov     ebx,m2
    imul    ebx
    shrd    eax,edx,16
    ret

ENDP
```

A instrução IMUL, e não MUL, foi usada porque os números de ponto fixo são sinalizados (o bit mais significativo é o sinal!). Vale aqui a mesma regra de sinalização para números inteiros: Se o bit mais significativo estiver setado o número é negativo e seu valor absoluto é obtido através do seu complemento (complemento 2). Quanto a manipulação dos sinais numa multiplicação... deixe isso com o IMUL! :)

A divisão também tem as suas complicações... suponha a seguinte divisão:

```
 0001.0000h
------------ = 0000.0000h (resto = 0001.000h)
 0002.0000h
```

A explicação deste resultado é simples: estamos fazendo divisão de dois números inteiros... Na aritimética inteira a divisão com o dividendo menor que o divisor sempre resulta num quociente zero!

Eis a solução: Se o divisor está deslocado 16 bits para esquerda (20000h é diferente de 2, certo!?), então precisamos deslocar o dividendo 16 bits para esquerda antes de fazermos a divisão! Felizmente os processadores 386 e superiores permitem divisões com dividendos de 64bits e divisores de 32bits. Assim, o deslocamento de 16 bits para esquerda do dividendo não é problemática!

```
0001.0000h << 16 = 00010000.0000h
```

```
00010000.0000h / 0002.0000h = 0000.8000h

    ou seja:

1 / 2 = 0.5
```

Eis a rotina em assembly que demonstra esse algorritmo:

```
PROC    FixedDiv
ARG     d1:DWORD, d2:DWORD

    mov     eax,d1      ; pega dividendo
    mov     ebx,d2      ; pega divisor

    sub     edx,edx

    shld    edx,eax,16
    shl     eax,16

    idiv    ebx
    ret

ENDP
```

Isso tudo é muito interessante, näo?! Hehehe... mas vou deixar vc mais desesperado ainda: A divisäo tem um outro problema! E quanto aos sinais?! O bit mais significativo de um inteiro pode ser usado para sinalizar o número (negativo = 1, positivo = 0), neste caso teremos ainda que complementar o número para sabermos seu valor absoluto. Se simplesmente zeraramos EDX e o bit mais significativo estiver setado estaremos dividindo um número positivo por outro número qualquer (já que o bit mais significativo dos 64bits resultantes será 0!). Vamos complicar mais um pouquinho o código da divisäo para sanar este problema:

```
PROC    FixedDiv
ARG     d1:DWORD, d2:DWORD

    sub     cl,cl       ; CL = flag
                        ; == 0 -> resultado positivo.
                        ; != 0 -> resultado negativo.

    mov     eax,d1      ; pega dividendo

    or      eax,eax     ; é negativo?!
    jns     @@no_chs1   ; näo! entäo näo troca sinal!

    neg     eax         ; é! entäo troca o sinal e...
    inc     cl          ; incrementa flag.
@@no_chs1:

    mov     ebx,d2      ; pega divisor

    or      ebx,ebx     ; é negativo?!
    jns     @@no_chs2   ; näo! entäo näo troca sinal!

    neg     ebx         ; é! entäo troca sinal e...
    dec     cl          ; decrementa flag.
@@no_chs2:
```

```
        sub     edx,edx

        shld    edx,eax,16
        shl     eax,16

        div     ebx             ; divisäo de valores positivos...
                                ; ... não precisamos de idiv!

        or      cl,cl           ; flag == 0?
        jz      @@no_chs3       ; sim! resultado é positivo.

        neg     eax             ; näo! resultado é negativo...
                                ; ... troca de sinal!
    @@no_chs3:
        ret

    ENDP
```

Se ambos os valores são negativos (d1 e d2) então o resultado será positivo. Note que se d1 é negativo CL é incrementado. Logo depois... se d2 também é negativo, CL é decrementado (retornando a 0). A rotina então efetuará divisäo de valores positivos e somente no final é que mudará o sinal do resultado, se for necessário!

Uma consideração a fazer é: Como "transformo" um número em ponto flutuante em ponto-fixo e vice-versa?!

Comecemos pela transformação de números inteiros em ponto-fixo: O nosso ponto-fixo está situado exatamente no meio de uma doubleword (DWORD), o que nos dá 16 bits de parte inteira e 16 de parte fracionária. A transformaçäo de um número inteiro para ponto-fixo é mais que simples:

```
FixP = I * 65536          ou
FixP = I << 16

onde FixP = Fixed Point (Ponto fixo)
     I    = Integer (Inteiro)
```

Desta forma os 16 bits superiores conterão o número inteiro e os 16 bits inferiores estarão zerados (um inteiro não tem parte fracionária, tem?!).

Se quisermos obter a componente inteira de um número de ponto fixo basta fazer o shift de 16 bits para direita.

A mesma regra pode ser usada para transformação de ponto-flutuante para ponto-fixo, só que näo usaremos shifting e sim multiplicaremos explicitamente por 65536.0! Suponha que queiramos transforma o número PI em ponto-fixo:

```
FixP = FloatP * 65536.0

FixP = 3.1415... * 65536.0 = 205887.4161
FixP = 205887

FixP = 0003.2439h
```

O que nos dá uma boa aproximação (se transformarmos 32439h em ponto flutuante novamente obteremos 3.14149475...). Apenas a parte inteira do resultado (205887.4161) nos interessa. (205887). Mas apareceu um pequenino problema que talvez vc não tenha notado...

Suponha que o resultado da multiplicação por 65536.0 desse 205887.865 (por exemplo, tá?!). Esse número está mais próximo de 205888 do que de 205887! Se tomarmos apenas a componente inteira do resultado obteremos um erro ainda maior (ponto-fixo não é muito preciso, como vc pode notar pelo exemplo acima!). Como fazer para obter sempre a componente inteira mais aproximada?! A soluçäo é somar 0.5 ao resultado da multiplicação por 65536.0!

Se a componente fracionária for maior ou igual a 0.5 então a soma da componente fracionária com 0.5 dará valor menor que 2.0 e maior ou igual a 1.0 (ou seja, a componente inteira dessa soma será sempre 1.0). Ao contrário, se a componente fracionária do resultado da multiplicação por 65536.0 for menor que 0.5 então a componente inteira da soma dessa componente por 0.5 será sempre 0.0! Então, somando o resultado da multiplicação com 0.5 podemos ou não incrementar a componente inteira de acordo com a proximidade do número real com o inteiro mais próximo!

Se a aproximação não for feita, o erro gira em torno de 15e-6, ou seja: 0.000015 (erro a patir da quinta casa decimal!).

A transformação de um número de ponto-flutuante para ponto-fixo fica então:

```
FixP = (FloatP * 65536.0) + 0.5

FixP = (3.1415... * 65536.0) + 0.5 = 205887.4161 + 0.5
FixP = 205887.9161
FixP = 205887  (ignorando a parte fracionária!)

FixP = 0003.2439h
```

A transformação contrária (de ponto-fixo para ponto-flutuante) é menos traumática, basta dividir o número de ponto fixo por 65536.0. Eis algumas macros, em C, para as transformaçöes:

```
#define INT2FIXED(x)    ((long)(x) << 16)
#define FIXED2INT(x)    ((x) >> 16)
#define DOUBLE2FIXED(x) (long)(((x) * 65536.0) + 0.5)
#define FIXED2DOUBLE(x) ((double)(x) / 65536.0)
```

Aritimética de ponto-fixo é recomendável apenas no caso de requerimento de velocidade e quando não necessitamos de precisäo nos calculos. O menor número que podemos armazenar na configuração atual é ±1.5259e-5 (1/65536) e o maior é ±32767.99998, aproximadamente. Números maiores ou menores que esses não são representáveis. Se o seu programa pode extrapolar esta faixa, não use ponto-fixo, vc obterá muitos erros de precisäo e, ocasionalmente, talvez até um erro de "Division By Zero".

Atençäo... A implementação dos procedimentos (PROC) acima säo um pouquinho diferentes para mixagem de código... Os compiladores C e PASCAL atuais utilizam o par DX:AX para retornar um DWORD, assim, no fim de cada PROC e antes do retorno coloque:

```
  shld    edx,eax,16
  shr     eax,16
```

Ou faça melhor ainda: modifique os códigos!

Eis a minha implementaçäo para as rotinas FixedMul e FixedDiv para mixagem de código com C ou TURBO PASCAL:

```
  /*
  ** Arquivo de cabeçalho FIXED.H
  */
  #if !defined(__FIXED_H__)
  #define __FIXED_T__

  /* Tipagem */
  typedef long    fixed_t;

  /* Macros de conversäo */
  #define INT2FIXED(x)    ((fixed_t)(x) << 16)
  #define FIXED2INT(x)    ((int)((x) >> 16))
  #define DOUBLE2FIXED(x) ((fixed_t)(((x) * 65536.0) + 0.5))
  #define FIXED2DOUBLE(x) ((double)(x) / 65536.0)

  /* Declaraçäo das funçöes */
  fixed_t pascal FixedMul(fixed_t, fixed_t);
  fixed_t pascal FixedDiv(fixed_t, fixed_t);

  #endif
```

```
  {*** Unit FixedPt para TURBO PASCAL ***}
  UNIT FIXEDPT;

  {} INTERFACE {}

  {*** Tipagem ***}
  TYPE
      TFixed  = LongInt;

  {*** Declaraçäo das funçöes ***}
  FUNCTION FixedMul(M1, M2 : TFixed) : TFixed;
  FUNCTION FixedDiv(D1, D2 : TFixed) : TFixed;

  {} IMPLEMENTATION {}

  {*** Inclui o arquivo .OBJ compilado do código abaixo ***}
  {$L FIXED.OBJ}

  {*** Declara funçöes como externas ***}
  FUNCTION FixedMul(M1, M2 : TFixed) : TFixed; EXTERN;
  FUNCTION FixedDiv(D1, D2 : TFixed) : TFixed; EXTERN;

  {*** Fim da Unit... sem inicializaçöes! ***}
  END.
```

```
  ; FIXED.ASM
  ; Módulo ASM das rotinas de multiplicação e divisäo em
  ; ponto fixo.

  ; Modelamento de memória e modo do compilador.
```

```
IDEAL
MODEL LARGE,PASCAL
LOCALS
JUMPS
P386        ; Habilita instruçöes do 386

; Declara os procedimentos como públicos
GLOBAL FixedMul : PROC
GLOBAL FixedDiv : PROC

; Inicio do segmento de código.
CODESEG

PROC    FixedMul
ARG     m1:DWORD, m2:DWORD

    mov     eax,[m1]
    mov     ebx,[m2]
    imul    ebx
    shr     eax,16  ; Coloca parte fracionária em AX.
                    ; DX já contém parte inteira!
    ret

ENDP

; Divisão em ponto fixo.
; d1 = Dividendo, d2 = Divisor
PROC    FixedDiv
ARG     d1:DWORD, d2:DWORD

    sub     cl,cl       ; CL = flag
                        ; == 0 -> resultado positivo.
                        ; != 0 -> resultado negativo.

    mov     eax,[d1]    ; pega dividendo

    or      eax,eax     ; é negativo?!
    jns     @@no_chs1   ; näo! então näo troca sinal!

    neg     eax         ; é! então troca o sinal e...
    inc     cl          ; incrementa flag.
@@no_chs1:

    mov     ebx,[d2]    ; pega divisor

    or      ebx,ebx     ; é negativo?!
    jns     @@no_chs2   ; näo! então näo troca sinal!

    neg     ebx         ; é! então troca sinal e...
    dec     cl          ; decrementa flag.
@@no_chs2:

    sub     edx,edx     ; Prepara para divisão.
    shld    edx,eax,16
    shl     eax,16

    div     ebx         ; divisão de valores positivos...
                        ; ... näo precisamos de idiv!

    or      cl,cl       ; flag == 0?
    jz      @@no_chs3   ; sim! resultado é positivo.

    neg     eax         ; näo! resultado é negativo...
                        ; ... troca de sinal!
```

```
  @@no_chs3:

      ;
      ; Apenas adequa para o compilador
      ;
      shld    edx,eax,16  ; DX:AX contém o DWORD
      shr     eax,16

      ret

  ENDP

  END
```