

Capítulo 6

Punto flotante

6.1. Representación de punto flotante

6.1.1. Números binarios no enteros

Cuando se discutieron los sistemas numéricos en el primer capítulo, sólo se trataron los valores enteros. Obviamente, debe ser posible representar números no enteros en otras bases como en decimal. En decimal, los dígitos a la derecha del punto decimal tienen asociados potencias de 10 negativas.

$$0,123 = 1 \times 10^{-1} + 2 \times 10^{-2} + 3 \times 10^{-3}$$

No es sorprendente que los números binarios trabajen parecido.

$$0,101_2 = 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 0,625$$

Esta idea se puede combinar con los métodos para enteros del capítulo 1 para convertir un número.

$$110,011_2 = 4 + 2 + 0,25 + 0,125 = 6,375$$

Convertir de decimal a binario no es muy difícil. En general divida el número decimal en dos partes: entero y fracción. Convierta la parte entera usando a binario usando los métodos del capítulo 1. La parte fraccionaria se convierte usando el método descrito abajo:

Considere una fracción binaria con los bits etiquetados como $a, b, c, \dots$ entonces el número se ve como:

$$0,abcdef \dots$$

Multiplique el número por dos. La representación binaria del nuevo número será:

$$a,bcdef \dots$$

$0,5625 \times 2 = 1,125$	first bit = 1
$0,125 \times 2 = 0,25$	second bit = 0
$0,25 \times 2 = 0,5$	third bit = 0
$0,5 \times 2 = 1,0$	fourth bit = 1

Figura 6.1: Convertir 0.5625 a binario

$0,85 \times 2 = 1,7$
$0,7 \times 2 = 1,4$
$0,4 \times 2 = 0,8$
$0,8 \times 2 = 1,6$
$0,6 \times 2 = 1,2$
$0,2 \times 2 = 0,4$
$0,4 \times 2 = 0,8$
$0,8 \times 2 = 1,6$

Figura 6.2: Convertir 0.85 a binario

Observe que el primer bit está ahora en el lugar del uno. Reemplace a con 0 y obtiene:

$$0, bcdef \dots$$

y multiplique por dos otra vez para obtener:

$$b, cdef \dots$$

Ahora el segundo bit (b) está en la posición del uno. Este procedimiento se puede repetir hasta los bits que se deseen encontrar. La Figura 6.1 muestra un ejemplo real que convierte 0.5625 a binario. El método para cuando la parte fraccionaria es cero.

Como otro ejemplo, considere convertir 23.85 a binario. Es fácil convertir la parte entera ($23 = 10111_2$), pero ¿Qué pasa con la parte fraccionaria (0,85)? La Figura 6.2 muestra el comienzo de este cálculo. Si uno mira cuidadosamente los números, se encuentra con un bucle infinito. Esto significa que

0.85 en un binario periódico (opuesto a los decimales periódicos en base 10).

¹ Hay un patrón de los números calculados. Mirando en el patrón, uno puede ver que $0,85 = 0,11\overline{0110}_2$. Así $23,85 = 10111,11\overline{0110}_2$.

Una consecuencia importante del cálculo anterior es que 23.85 no se puede representar *exactamente* en binario usando un número finito de bits (tal como $\frac{1}{3}$ no se puede representar en decimal con un número finito de dígitos). Como muestra este capítulo, las variables `float` y `double` en C son almacenados en binario. Así, valores como 23.85 no se pueden almacenar exactamente en estas variables. Sólo se puede almacenar una aproximación a 23.85

Para simplificar el hardware, los números de puntos flotante se almacenan con un formato consistente. Este formato usa la notación científica (pero en binario, usando potencias de dos, no de diez). Por ejemplo 23.85 o $10111,11011001100110\dots_2$ se almacenará como:

$$1,011111011001100110\dots \times 2^{100}$$

(Donde el exponente (100) está en binario). Un número de punto flotante *normalizado* tiene la forma:

$$1,ssssssssssssssss \times 2^{eeeeeee}$$

Dónde 1,ssssssssssssss es la *mantisa* y eeeeeee es el *exponente*.

6.1.2. Representación IEEE de punto flotante

El IEEE (Institute of Electrical and Electronic Engineer) es una organización internacional que ha diseñado formato binarios específicos para almacenar números de punto flotante. Este formato se usa en la mayoría (pero no todos) los computadores hechos hoy día. A menudo es soportado por el hardware de computador en sí mismo. Por ejemplo el coprocesador numérico (o matemático) de Intel (que está empujado en todas las CPU desde el Pentium) lo usa. El IEEE define dos formatos con precisión diferentes: precisión simple y doble, la precisión simple es usada por las variables `float` en C y la precisión doble es usada por la variable `double`.

El coprocesador matemático de Intel utiliza un tercer nivel de mayor precisión llamado *precisión extendida*. De hecho, todos los datos en el coprocesador en sí mismo están en esta precisión. Cuando se almacenan en memoria desde el coprocesador se convierte a precisión simple o doble automáticamente.² La precisión extendida usa un formato general ligeramente

¹No debería sorprenderse de que un número pudiera repetirse en una base, pero no en otra. Piense en $\frac{1}{3}$, es periódico en decimal, pero en ternario (base3) sería $0,1_34$.

²Algunos compiladores (como Borland) los tipos `long double` usa esta precisión extendida. Sin embargo, otros compiladores usan la precisión doble para `double` y `long double`. (Esto es permitido por ANSI C).

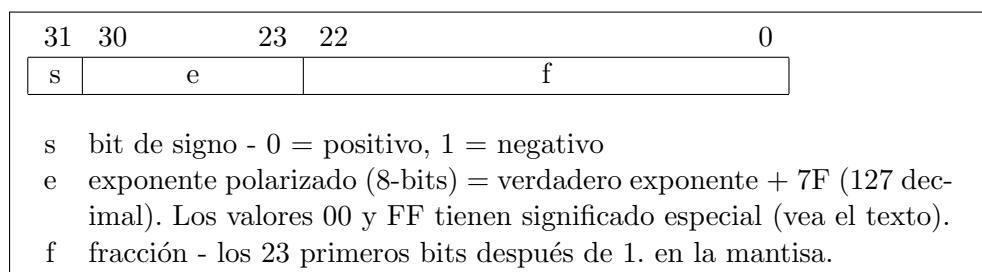


Figura 6.3: Precisión simple de la IEEE

diferente que los formatos float y double de la IEEE y no serán discutidos acá.

Presición simple IEEE

El punto flotante de precisión simple usa 32 bits para codificar el número. Normalmente son exactos los primeros 7 dígitos decimales. Los números de punto flotante son almacenados en una forma mucho más complicada que los enteros. La Figura 6.3 muestra la forma básica del formato de precisión simple del IEEE. Hay varias peculiaridades del formato. Los números de punto flotante no usan la representación en complemento a 2 para los números negativos. Ellos usan la representación de magnitud y signo. El bit 31 determina el signo del número como se muestra.

El exponente binario no se almacena directamente. En su lugar, se almacena la suma del exponente y 7F en los bits 23 al 30. Este *exponente polarizado* siempre es no negativo.

La parte fraccionaria se asume que es normalizada (en la forma 1.*ssssssss*). Ya que el primer bit es siempre uno, éste uno *no se almacena*. Esto permite el almacenamiento de un bit adicional al final y así se incrementa un poco la precisión. Esta idea se conoce como la *representación oculta del uno*.

¿Cómo se podría almacenar 23.85? Primero es positivo, así el bit de signo es 0, ahora el exponente verdadero es 4, así que el exponente es $7F + 4 = 83_{16}$. Finalmente la fracción es (recuerde el uno de adelante está oculto). Colocando todo esto unido (para ayudar a aclarar las diferentes secciones del formato del punto flotante, el bit de signo y la fracción han sido subrayados y los bits se han agrupado en nibles):

$$\underline{0} \ 100 \ 0001 \ 1 \ \underline{011 \ 1110 \ 1100 \ 1100 \ 1100}_2 = 41BECCCC_{16}$$

Esto no es exactamente 23.85 (ya que es un binario periódico). Si uno convierte el número anterior a decimal, uno encuentra que es aproximadamente. Este número es muy cercano a 23.85, pero no es exacto. Actualmente, en C, 23.85 no se puede representar exactamente como arriba. Ya que el bit del

Uno debería tener en cuenta que los bytes 41 BE CC CD se pueden interpretar de diferentes maneras dependiendo qué hace un programa con ellos. Como el número de punto flotante de precisión simple, ellos representan, pero como un entero, ellos representan. La CPU no conoce cuál es la interpretación correcta.

$e = 0$ and $f = 0$	denota el número cero (que no puede ser normalizado) Observe que hay un $+0$ y -0 .
$e = 0$ and $f \neq 0$	denotes un <i>número sin normalizar</i> . Estos se discuten en la próxima sección.
$e = FF$ and $f = 0$	denota infinito (∞). Hay infinitos positivo y negativo.
$e = FF$ and $f \neq 0$	denota un resultado indefinido, Conocido como <i>NaN</i> (Not a Number).

Cuadro 6.1: Valores especiales de f y e

extremo izquierdo más significativo que fue truncado de la representación exacta es 1. El último bit se redondea a 1. Así 23.85 podría ser representado como 41 BE CC CD en hexadecimal usando precisión simple. Convirtiendo esto a decimal el resultado y que es una aproximación ligeramente mejor de 23.85.

¿Cómo se podría representar -23.85? sólo cambie el bit de signo: C1BECCCD
¡no tome el complemento a 2!

Ciertas combinaciones de e y f tienen significado especial para los float IEEE. La Tabla 6.1 describe estos valores especiales. Un infinito se produce por un desborde o por una división por cero. Un resultado indefinido se produce por una operación no válida como tratar de encontrar la raíz cuadrada de un número negativo, sumar dos infinitos, etc.

Los números de precisión simple están en el rango de $1,0 \times 2^{-126}$ ($\approx 1,1755 \times 10^{-35}$) to $1,11111 \dots \times 2^{127}$ ($\approx 3,4028 \times 10^{35}$).

Números sin normalizar

Los números sin normalizar se pueden usar para representar números con magnitudes más pequeñas que los normalizados (menores a $1,0 \times 2^{-126}$). Por ejemplo, considere el número $1,001_2 \times 2^{-129}$ ($\approx 1,6530 \times 10^{-39}$). En la forma normalizada dada, el exponente es muy pequeño. Sin embargo, se puede representar de una forma no normalizada como: $0,01001_2 \times 2^{-127}$. Para almacenar este número, el exponente es fijado a 0 (ver la tabla 6.1) y la función de la mantisa completa del número escrito como un producto con 2^{-127} (todos los bits son almacenados incluyendo el 1 a la izquierda del punto decimal). La representación de $1,001 \times 2^{-129}$ es entonces:

0 000 0000 0 001 0010 0000 0000 0000 0000



Figura 6.4: Precisión doble del IEEE

doble precisión IEEE

La doble precisión IEEE usa 64 bits para representar números y normalmente son exactos los 15 primeros dígitos decimales significativos. Como muestra la Figura 6.4, el formato básico es muy similar a la precisión simple. Se usan más bits para el exponente (11) y la fracción (52) que para la precisión simple.

El gran rango para el exponente tiene dos consecuencias. La primera es que se calcula como la suma del exponente real y 3FF (1023) (no 7F como para la precisión simple). Segundo, se permite un gran rango de exponentes verdaderos (y así usa un gran rango de magnitudes). Las magnitudes de precisión doble van de 10^{-308} hasta 10^{308} aproximadamente.

En el campo de la fracción el responsable del incremento en el número de dígitos significativos para los valores dobles.

Como un ejemplo, considere nuevamente 23.85 otra vez. El exponente polarizado será $4 + 3FF = 403$ en hexadecimal. Así la representación doble sería:

0 100 0000 0011 0111 1101 1001 1001 1001 1001 1001 1001 1001 1001 1010

O 40 37 D9 99 99 99 99 9A en hexadecimal. Si uno convierte esto a decimal uno encuentra 23.8500000000000014 (¡hay 12 ceros!) que es una aproximación mucho mejor de 23.85.

La precisión doble tiene los mismos valores especiales que la precisión simple.³ Los números no normalizados son muy similares también. La principal diferencia es que los números dobles sin normalizados usan 2^{-1023} en lugar de 2^{-127} .

6.2. Aritmética de punto flotante

La aritmética de punto flotante en un computador diferente que en la matemática continua. En la matemáticas, todos los números pueden ser considerados exactos. Como se muestra en la sección anterior, en un computador muchos números no se pueden representar exactamente con un número finito de bits. Todos los cálculos se realizan con una precisión limitada. En los

³La única diferencia es que para los valores de infinito e indefinido, el exponente polarizado es 7FF y no FF.

ejemplos de esta sección, se usarán números con una mantisa de 8 bits por simplicidad.

6.2.1. suma

Para sumar dos números de punto flotante, los exponentes deben ser iguales. Si ellos, no son iguales, entonces ellos se deben hacer iguales, desplazando la mantisa del número con el exponente más pequeño. Por ejemplo, considere $10,375 + 6,34375 = 16,71875$ o en binario:

$$\begin{array}{r} 1,0100110 \times 2^3 \\ + 1,1001011 \times 2^2 \\ \hline \end{array}$$

Estos dos números no tienen el mismo exponente así que se desplaza la mantisa para hacer iguales los exponentes y entonces sumar:

$$\begin{array}{r} 1,0100110 \times 2^3 \\ + 0,1100110 \times 2^3 \\ \hline 10,0001100 \times 2^3 \end{array}$$

Observe que el desplazamiento de $1,1001011 \times 2^2$ pierde el uno delantero y luego de redondear el resultado se convierte en $0,1100110 \times 2^3$. El resultado de la suma, $10,0001100 \times 2^3$ (o $1,00001100 \times 2^4$) es igual a $10000,110_2$ o 16.75 . Esto *no* es igual a la respuesta exacta (16.71875) Es sólo una aproximación debido al error del redondeo del proceso de la suma.

Es importante tener en cuenta que la aritmética de punto flotante en un computador (o calculadora) es siempre una aproximación. Las leyes de las matemáticas no siempre funcionan con números de punto flotante en un computador. Las matemáticas asumen una precisión infinita que un computador no puede alcanzar. Por ejemplo, las matemáticas enseñan que $(a+b)-b = a$; sin embargo, esto puede ser exactamente cierto en un computador.

6.2.2. Resta

La resta trabaja muy similar y tiene los mismos problemas que la suma. Como un ejemplo considere $16,75 - 15,9375 = 0,8125$:

Subtraction works very similarly and has the same problems as addition. As an example, consider $16,75 - 15,9375 = 0,8125$:

$$\begin{array}{r} 1,0000110 \times 2^4 \\ - 1,1111111 \times 2^3 \\ \hline \end{array}$$

Desplazando $1,1111111 \times 2^3$ da (redondeando arriba) $1,0000000 \times 2^4$

$$\begin{array}{r} 1,0000110 \times 2^4 \\ - 1,0000000 \times 2^4 \\ \hline 0,0000110 \times 2^4 \end{array}$$

$0,0000110 \times 2^4 = 0,11_2 = 0,75$ que no es exacto.

6.2.3. Multiplicación y división

Para la multiplicación, las mantisas son multiplicadas y los exponentes son sumados. Considere $10,375 \times 2,5 = 25,9375$:

$$\begin{array}{r} 1,0100110 \times 2^3 \\ \times 1,0100000 \times 2^1 \\ \hline 10100110 \\ + 10100110 \\ \hline 1,1001111000000 \times 2^4 \end{array}$$

Claro está, el resultado real podría ser redondeado a 8 bits para dar:

$$1,1010000 \times 2^4 = 11010,000_2 = 26$$

La división es más complicada, pero tiene problemas similares con errores de redondeo.

6.2.4. Ramificaciones para programar

El principal punto de esta sección es que los cálculos de punto flotante no son exactos. El programador necesita tener cuidado con esto. Un error común que el programador hace con números de punto flotante es compararlos asumiendo que el cálculo es exacto. Por ejemplo considere una función llamada $f(x)$ que hace un cálculo complejo y un programa está tratando de encontrar las raíces de la función.⁴ Uno podría intentar usar la siguiente instrucción para mirar si x es una raíz:

```
if ( f(x) == 0.0 )
```

¿Pero si $f(x)$ retorna 1×10^{-30} ? Esto probablemente significa que x es una *muy* buena aproximación a una raíz verdadera; sin embargo, la igualdad será falsa. Puede no haber ningún valor de punto flotante IEEE de x que retorne cero exactamente, debido a los errores de redondeo en $f(x)$.

Un método mucho mejor sería usar:

```
if ( fabs(f(x)) < EPS )
```

Dónde EPS es un macro definido a ser un valor positivo muy pequeño (como 1×10^{-10}). Esto es cierto si $f(x)$ está muy cercano a cero. En general, comparar un valor de punto flotante (digamos x) con otro (y) use:

```
if ( fabs(x - y)/fabs(y) < EPS )
```

⁴Una raíz de una función es un valor x tal que $f(x) = 0$

6.3. El coprocesador numérico

6.3.1. Hardware

Los primeros procesadores Intel no tenían soporte de hardware para las operaciones de punto flotante. Esto no significa que ellos no podían efectuar operaciones de punto flotante. Esto sólo significa que ellas se realizaban por procedimientos compuestos de muchas instrucciones que no son de punto flotante. Para estos primeros sistemas, Intel suministraba un circuito integrado adicional llamado *coprocesador matemático*. Un coprocesador matemático tiene instrucciones de máquina que realizan instrucciones de punto flotante mucho más rápido que usando procedimientos de software (en los primeros procesadores se realizaban al menos 10 veces más rápido). El coprocesador para el 8086/8088 fue llamado 8087. Para el 80286 era el 80287 y para el 80386 un 80387. El procesador 80486DX integró el coprocesador matemático en el 80486 en sí mismo⁵. Desde el Pentium, todas las generaciones del 80X86 tienen un coprocesador matemático empujado; sin embargo él todavía se programa como si fuera una unidad separada. Aún los primeros sistemas sin un coprocesador pueden instalar un software que emula el coprocesador matemático. Estos emuladores se activan automáticamente cuando un programa ejecuta una instrucción del coprocesador y corre un procedimiento que produce los mismos resultados que el coprocesador (mucho más lento claro está).

El coprocesador numérico tiene ocho registros de punto flotante. Cada registro almacena 80 bits. Los números de punto flotante se almacenan en estos registros *siempre* como números de 80 bits de precisión extendida. Los registros se llaman ST0, ST1, ... ST7. Los registros de punto flotante se usan diferentes que los registros enteros en la CPU. Los registros de punto flotante están organizados como una pila. Recuerde que una *pila* es una lista LIFO (*Last In First Out*). ST0 siempre se refiere al valor en el tope de la pila. Todos los números nuevos se añaden al tope de la pila. Los números existentes se empujan en la pila para dejarle espacio al nuevo número.

Hay también un registro de estado en el coprocesador numérico. Tiene varias banderas. Sólo las 4 banderas usadas en comparaciones serán estudiadas. C₀, C₁, C₂ and C₃. El uso de ellas se discutirá luego.

6.3.2. Instrucciones

Para distinguir fácilmente las instrucciones normales de la CPU de las del coprocesador, todos los nemónicos del coprocesador comienzan por F.

⁵Sin embargo el 80486SX no tiene el un coprocesador integrado

Carga y almacenamiento

Hay varias instrucciones que cargan datos en el tope de la pila de registro del coprocesador:

FLD <i>source</i>	Carga un número de punto flotante de la memoria en el tope de la pila. La <i>source</i> puede ser un número de precisión simple doble o extendida o un registro del coprocesador
FILD <i>source</i>	Lee un <i>entero</i> de memoria, lo convierte a punto flotante y almacena el resultado en el tope de la pila. <i>source</i> puede ser una palabra, palabra doble o una palabra cuádruple.
FLD1	almacena un uno en el tope de la pila.
FLDZ	almacena un cero en el tope de la pila.

Hay también varias instrucciones que almacenan datos de la pila en memoria. Algunas de estas instrucciones también *sacan* el número de la pila.

FST <i>dest</i>	Almacena el tope de la pila (ST0) en memoria. El <i>destino</i> puede ser un número de precisión simple o doble o un registro de coprocesador.
FSTP <i>dest</i>	Almacena el tope de la pila en la memoria tal como FST; sin embargo luego de que se almacena el número, su valor se saca de la pila. El <i>destino</i> puede ser un número de precisión simple o doble o un registro del coprocesador.
FIST <i>dest</i>	Almacena el valor del tope de la pila convertido a entero en memoria. El <i>destino</i> puede ser una palabra o palabra doble. La pila no sufre ningún cambio. Cómo se convierte el número de punto flotante a entero depende de algunos bits de la <i>palabra de control</i> del coprocesador. Este es un registro especial (no de punto flotante) que controla cómo trabaja el coprocesador. Por defecto, la palabra de control se inicia tal que redondea al entero más cercano cuando se convierte a entero. Sin embargo las instrucciones FSTCW (Store Control Word) y FDLCW (load control word) se pueden usar para cambiar este comportamiento.
FISTP <i>dest</i>	Lo mismo que FIST, excepto por dos cosas. El tope de la pila se saca y el <i>destino</i> también puede ser una palabra cuádruple.

Hay otras dos instrucciones que pueden mover o quitar datos de la pila en sí misma.

FXCH ST <i>n</i>	intercambia los valores en ST0 y ST <i>n</i> en la pila (donde <i>n</i> es el número del registro de 1 a 7).
FFREE ST <i>n</i>	libera un registro en la pila, marcando el registro como no usado o vacío.

```

1 segment .bss
2 array      resq SIZE
3 sum        resq 1
4
5 segment .text
6     mov     ecx, SIZE
7     mov     esi, array
8     fldz                    ; ST0 = 0
9 lp:
10    fadd     qword [esi]     ; ST0 += *(esi)
11    add      esi, 8          ; se mueve al próximo double
12    loop     lp
13    fstp     qword sum       ; almacena el resultado en sum

```

Figura 6.5: Ejemplo sumar arreglo

Suma y resta

Cada una de las instrucciones de suma calcula la suma de ST0 y otro operando, el resultado siempre se almacena en un registro del coprocesador.

FADD *src* **ST0 += *src*.** *src* puede ser cualquier registro del coprocesador o un número de precisión simple o doble en memoria.

FADD *dest*, ST0 ***dest* += ST0.** El *destino* puede ser cualquier registro del coprocesador

FADDP *dest* o ***dest* += ST0** entonces se saca de la pila. El *destino* puede ser cualquier registro del coprocesador.

FIADD *src* **ST0 += (float) *src*.** suma un entero con ST0. *src* debe ser una palabra o una palabra doble en memoria.

Hay el doble de instrucciones de resta que de suma porque el orden de los operandos es importante. ($a + b = b + a$, pero $a - b \neq b - a$!). Por cada instrucción, hay una alterna que resta en el orden inverso. Esas instrucciones al revés todas culminan con **R** o **RP**. La Figura muestra un pequeño código que suma los elementos de un arreglo de doubles. En las líneas 10 y 13, uno debe especificar el tamaño del operando de memoria. De otra forma el ensamblador no conocería si el operando de memoria era un float (dword) o double (qword).

FSUB <i>src</i>	ST0 -= <i>src</i> . <i>src</i> puede ser cualquier registro del coprocesador o un número de precisión doble o simple en memoria.
FSUBR <i>src</i>	ST0 = <i>src</i> - ST0. <i>src</i> puede ser cualquier registro del coprocesador o un número de precisión doble o simple en memoria.
FSUB <i>dest</i> , ST0	<i>dest</i> -= ST0. <i>dest</i> puede ser cualquier registro del coprocesador.
FSUBR <i>dest</i> , ST0	<i>dest</i> = ST0 - <i>dest</i> . <i>dest</i> puede ser cualquier registro del coprocesador.
FSUBP <i>dest</i> o	<i>dest</i> -= ST0 entonces sale de la pila. <i>dest</i> puede ser cualquier registro del coprocesador.
FSUBP <i>dest</i> , ST0	
FSUBRP <i>dest</i> o	<i>dest</i> = ST0 - <i>dest</i> entonces sale de la pila. <i>dest</i> puede ser cualquier registro del coprocesador.
FSUBRP <i>dest</i> , ST0	
FISUB <i>src</i>	ST0 -= (float) <i>src</i> . Resta un entero de ST0. <i>src</i> debe ser una palabra o palabra doble en memoria.
FISUBR <i>src</i>	ST0 = (float) <i>src</i> - ST0. Resta ST0 de un entero. <i>src</i> debe ser una palabra o palabra doble en memoria.

Multiplicación y división

La instrucción de multiplicación son totalmente análogas a las instrucciones de suma.

FMUL <i>src</i>	ST0 *= <i>src</i> . <i>src</i> puede ser cualquier registro del coprocesador o un número de precisión simple o doble en memoria.
FMUL <i>dest</i> , ST0	<i>dest</i> *= ST0. <i>dest</i> puede ser cualquier registro del coprocesador.
FMULP <i>dest</i> o	<i>dest</i> *= ST0 entonces sale de la pila. <i>dest</i> puede ser cualquier registro del coprocesador.
FMULP <i>dest</i> , ST0	
FIMUL <i>src</i>	ST0 *= (float) <i>src</i> . Multiplica un entero con ST0. <i>src</i> debe ser una palabra o palabra doble en memoria.

No es sorprendente que las instrucciones de división son análogas a las instrucciones de resta. La división por cero de un infinito.

FDIV <i>src</i>	ST0 /= <i>src</i> . <i>src</i> puede ser cualquier registro del coprocesador o un número de precisión simple o doble en memoria.
FDIVR <i>src</i>	ST0 = <i>src</i> / ST0. <i>src</i> puede ser cualquier registro del coprocesador o una número de precisión simple o doble en memoria.
FDIV <i>dest</i> , ST0	<i>dest</i> /= ST0. <i>dest</i> puede ser cualquier registro del coprocesador.
FDIVR <i>dest</i> , ST0	<i>dest</i> = ST0 / <i>dest</i> . <i>dest</i> puede ser cualquier registro del coprocesador.
FDIVP <i>dest</i> o FDIVP <i>dest</i> , ST0	<i>dest</i> /= ST0 entonces sale de la pila. <i>dest</i> puede ser cualquier registro del coprocesador.
FDIVRP <i>dest</i> o FDIVRP <i>dest</i> , ST0	<i>dest</i> = ST0 / <i>dest</i> entonces sale de la pila. <i>dest</i> puede ser cualquier registro del coprocesador.
FIDIV <i>src</i>	ST0 /= (float) <i>src</i> . Divide ST0 por un entero. <i>src</i> debe ser una palabra o palabra doble en memoria.
FIDIVR <i>src</i>	ST0 = (float) <i>src</i> / ST0. Divide un entero por ST0. <i>src</i> debe ser una palabra o palabra doble en memoria.

Comparaciones

El coprocesador también realiza comparaciones de números de punto flotante. La fórmula de instrucciones FCOM hacen esta operación.

FCOM <i>src</i>	compara ST0 y <i>src</i> . <i>src</i> puede ser un registro del coprocesador o un float o double en memoria.
puede ser un FCOMP <i>src</i>	compara ST0 y <i>src</i> , luego sale de la pila. <i>src</i> puede ser un registro del coprocesador o un float o double en memoria.
FCOMPP	compara ST0 y ST1, entonces saca de la pila dos veces.
FICOM <i>src</i>	compara ST0 y (float) <i>src</i> . <i>src</i> puede ser una palabra o palabra dobel en memoria.
FICOMP <i>src</i>	compara ST0 y (float) <i>src</i> , entonces saca de la pila. <i>src</i> puede ser una palabra o palabra doble entera en memoria.
FTST	compara ST0 and 0.

Estas instrucciones cambian los bits C₀, C₁, C₂ y C₃ del registro de estado del coprocesador. Desafortunadamente no es posible que la CPU acceda a estos bits directamente. Las instrucciones de salto condicional usan el registro FLAGS, no el registro de estado del coprocesador. Sin embargo es relativamente fácil transferir los bits de la palabra de estado a los bits correspondientes del registro FLGS usando algunas instrucciones nuevas.

```

1  ;    if ( x > y )
2  ;
3      fld    qword [x]          ; ST0 = x
4      fcomp  qword [y]          ; compara ST0 and y
5      fstsw  ax                  ; mueve los bits C a FLAGS
6      sahf
7      jna    else_part          ; si x no es mayor que y,
8                                  ; vaya a else_part
9  then_part:
10     ; código para parte de entonces
11     jmp     end_if
12  else_part:
13     ; código para parte si no
14  end_if:

```

Figura 6.6: Ejemplo de comparación

FSTSW *dest* Almacena la palabra de estado del coprocesador in memoria o en el registro AX.

SAHF Almacena el registro AH en el registro FLAGS.

LAHF Carga el registro AH con los bits del registro FLAGS.

La Figura 6.6 muestra un ejemplo cortico. Las líneas 5 y 6 transfieren los bits C_0 , C_1 , C_2 y C_3 de la palabra de estado del coprocesador al registro FLAGS. Los bits son transferidos tal que ellos son análogos al resultado de una comparación de dos enteros *sin signo*. Este es el porque la línea 7 usa la instrucción JNA.

El Pentium Pro (y procesadores posteriores II y III) soportan 2 nuevos operadores de comparación que directamente modifican el registro FLAGS de la CPU.

FCOMI *src* compara ST0 y *src*. *src* debe ser un registro del coprocesador.

FCOMIP *src* compara ST0 y *src*, entonces se saca de la pila. *src* debse ser un registro del coprocesador.

La Figura 6.7 muestra una subrutina de ejemplo que encuentran el máximo de dos números tipo double usando la instrucción FCOMIP. No confundan esta instrucción con la función de comparación de enteros (FICOMP y FICOM).

Instrucciones miscelaneas

Esta sección cubre otras instrucciones que suministra el procesador:

FCHS $ST0 = -ST0$ cambia el bit de signo de $ST0$
 FABS $ST0 = |ST0|$ Toma el valor absoluto de $ST0$
 FSQRT $ST0 = \sqrt{ST0}$ Toma la raíz cuadrada de $ST0$
 FSCALE $ST0 = ST0 \times 2^{[ST1]}$ multiplica $ST0$ por una potencia de dos rápidamente. $ST1$ no se quita de la pila del coprocesador. La Figure 6.8 muestra un ejemplo de cómo usar esta instrucción.

6.3.3. Ejemplos

6.3.4. Fórmula cuadrática

El primer ejemplo muestra cómo se puede programar la fórmula cuadrática en ensamblador. Recuerde que la fórmula cuadrática calcula la solución de una ecuación cuadrática:

$$ax^2 + bx + c = 0$$

La fórmula en sí misma da dos soluciones para x : x_1 y x_2 .

$$x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

La expresión dentro de la raíz cuadrada ($b^2 - 4ac$) es llamada *discriminante*, su valor es útil para determinar las 3 posibilidades para la solución:

1. Hay solo un real en la solución degenerad. $b^2 - 4ac = 0$
2. Hay dos soluciones reale. $b^2 - 4ac > 0$
3. Hay dos soluciones complejas. $b^2 - 4ac < 0$

Este es un pequeño programa en C que usa la subrutina en ensamblador.

```

                                quadt.c


---



#include <stdio.h>

int quadratic( double, double, double, double *, double *);

int main()
{
    double a,b,c, root1, root2;

    printf("Enter a, b, c: ");
    scanf("%lf%lf%lf", &a, &b, &c);
    if (quadratic( a, b, c, &root1, &root2) )

```

```

    printf (" roots: %.10g %.10g\n", root1, root2);
else
    printf (" No real roots\n");
return 0;
}

```

quadt.c

A continuación está la rutina en ensamblador.

```

1  ; función quadratic
2  ; Halla la solución de la ecuación cuadrática:
3  ;      a*x^2 + b*x + c = 0
4  ; Prototipo de C:
5  ;   int quadratic( double a, double b, double c,
6  ;                  double * root1, double *root2 )
7  ; Parámetros:
8  ;   a, b, c - Coeficientes de la ecuación cuadrática (ver arriba)
9  ;   root1   - Apuntador al double que almacena la primera raíz
10 ;   root2   - Apuntador al double que almacena la segunda raíz
11 ; Valor de retorno:
12 ;   devuelve 1 si las raíces son reales si no 0
13
14 %define a          qword [ebp+8]
15 %define b          qword [ebp+16]
16 %define c          qword [ebp+24]
17 %define root1      dword [ebp+32]
18 %define root2      dword [ebp+36]
19 %define disc       qword [ebp-8]
20 %define one_over_2a qword [ebp-16]
21
22 segment .data
23 MinusFour          dw          -4
24
25 segment .text
26     global  _quadratic
27 _quadratic:
28     push    ebp
29     mov     ebp, esp
30     sub     esp, 16          ; asigna 2 doubles (disc & one_over_2a)
31     push    ebx              ; debe guardar el valor original de ebx
32

```



```

33      fild    word [MinusFour]; pila -4
34      fld     a                ; pila: a, -4
35      fld     c                ; pila: c, a, -4
36      fmulp   st1              ; pila: a*c, -4
37      fmulp   st1              ; pila: -4*a*c
38      fld     b
39      fld     b                ; pila: b, b, -4*a*c
40      fmulp   st1              ; pila: b*b, -4*a*c
41      faddp   st1              ; pila: b*b - 4*a*c
42      ftst
43      fstsw   ax                ; prueba con 0
44      sahf
45      jb      no_real_solutions ; if disc < 0, no hay soluciones reales
46      fsqrt
47      fstp    disc              ; pila: sqrt(b*b - 4*a*c)
48      fstp    disc              ; almacena y saca de la pila
49      fld1
50      fld     a                ; pila: 1.0
51      fscale
52      fdivp   st1              ; pila: a, 1.0
53      fst     one_over_2a       ; pila: a * 2^(1.0) = 2*a, 1
54      fld     b                ; pila: 1/(2*a)
55      fld     disc              ; pila: b, 1/(2*a)
56      fsubrp  st1              ; pila: disc, b, 1/(2*a)
57      fmulp   st1              ; pila: disc - b, 1/(2*a)
58      mov     ebx, root1        ; pila: (-b + disc)/(2*a)
59      fstp    qword [ebx]       ; almacena en *root1
60      fld     b                ; pila: b
61      fld     disc              ; pila: disc, b
62      fchs
63      fsubrp  st1              ; pila: -disc, b
64      fmul    one_over_2a       ; pila: -disc - b
65      mov     ebx, root2        ; pila: (-b - disc)/(2*a)
66      fstp    qword [ebx]       ; almacena en *root2
67      mov     eax, 1            ; valor de retorno es 1
68      jmp     short quit
69
70 no_real_solutions:
71     mov     eax, 0              ; valor de retorno es 0
72
73 quit:
74     pop     ebx
75     mov     esp, ebp

```

```

75         pop     ebp
76         ret

```

quad.asm

6.3.5. Leer arreglos de archivos

Este ejemplo, es una rutina en ensamblador que lee doubles de un archivo. Sigue un pequeño programa de prueba en C.

readt.c

```

/*
 * Este programa prueba el procedimiento en ensamblador bit read_doubles().
 * Lee doubles de stdin. (use la ó redirección para leer desde un archivo.)
 */
#include <stdio.h>
extern int read_doubles( FILE *, double *, int );
#define MAX 100

int main()
{
    int i,n;
    double a[MAX];

    n = read_doubles(stdin, a, MAX);

    for( i=0; i < n; i++ )
        printf( " %3d %g\n", i, a[i] );
    return 0;
}

```

readt.c

A continuación la rutina en ensamblador

read.asm

```

1 segment .data
2 format db      "%lf", 0          ; format for fscanf()
3
4 segment .text
5     global _read_doubles
6     extern _fscanf
7
8 %define SIZEOF_DOUBLE 8

```

```

9  %define FP                dword [ebp + 8]
10 %define ARRAYP            dword [ebp + 12]
11 %define ARRAY_SIZE        dword [ebp + 16]
12 %define TEMP_DOUBLE        [ebp - 8]
13
14 ;
15 ; función _read_doubles
16 ; prototipo de C:
17 ; int read_doubles( FILE * fp, double * arrayp, int array_size );
18 ; Esta función lee dobles de un archivo de texto hasta EOF o hasta
19 ; que se llene el arreglo
20 ; Parámetros:
21 ; fp          - apuntador FILE pointer a leer desde (se debe abrir para entrada)
22 ; arrayp      - apuntador a un arreglo de doubles para leer en él
23 ; array_size - número de elementos en el arreglo
24 ; Valor de retorno:
25 ; número de doubles almacenado en el arreglo (en EAX)
26
27 _read_doubles:
28     push    ebp
29     mov     ebp, esp
30     sub     esp, SIZEOF_DOUBLE    ; define un double en la pila
31
32     push    esi                  ; guarda esi
33     mov     esi, ARRAYP          ; esi = ARRAYP
34     xor     edx, edx              ; edx = índice del arreglo (inicialmente en 0)
35
36 while_loop:
37     cmp     edx, ARRAY_SIZE      ; si edx < ARRAY_SIZE?
38     jnl     short quit           ; si no, salir del bucle
39 ;
40 ; llama a fscanf() para leer un double en TEMP_DOUBLE
41 ; fscanf() podría cambiar edx y así guardarlo
42 ;
43     push    edx                  ; guarda edx
44     lea     eax, TEMP_DOUBLE
45     push    eax                  ; push &TEMP_DOUBLE
46     push    dword format         ; push &format
47     push    FP                   ; push el apuntador al archivo
48     call    _fscanf
49     add     esp, 12
50     pop     edx                  ; restaura edx

```

```

51      cmp     eax, 1                ; fscanf retornó 1?
52      jne     short quit           ; si no, salir del bucle
53
54      ;
55      ; copia TEMP_DOUBLE en ARRAYP[edx]
56      ; (Los ocho bytes del double son copiados por las dos copias de 4 bytes)
57      ;
58      mov     eax, [ebp - 8]
59      mov     [esi + 8*edx], eax    ; primero copia los 4 bytes inferiores
60      mov     eax, [ebp - 4]
61      mov     [esi + 8*edx + 4], eax ; ahora copia los 4 bytes superiores
62
63      inc     edx
64      jmp     while_loop
65
66 quit:
67      pop     esi                  ; restaura esi
68
69      mov     eax, edx             ; almacena el valor de retorno en eax
70
71      mov     esp, ebp
72      pop     ebp
73      ret

```

read.asm

6.3.6. Hallar primos

Este último ejemplo halla números primos una vez más. Esta implementación es más eficiente que la anterior. Almacena los primos que ha encontrado en un arreglo y solo divide por los primos que ha encontrado antes, en lugar de cada número impar, para encontrar nuevos primos.

Otra diferencia es que calcula la raíz cuadrada del número para buscar el próximo primo a determinar en qué punto parar la búsqueda de factores. Altera la palabra de control de coprocesador tal que cuando almacena la raíz cuadrada como un entero, y la trunca en lugar de redondearla. Esto es controlado por los bits 10 y 11 de la palabra de control. Estos bits se llaman los bits RC (Rounding Control). Si ellos son ambos 0 (el defecto) el coprocesador redondea cuando convierte a entero. Si ellos son ambos 1, el coprocesador trunca las conversiones enteras. Observe que la rutina se cuida de guardar la palabra de control original y restaurarla antes de retornar.

A continuación el programa en C:

fprime.c

```

#include <stdio.h>
#include <stdlib.h>
/*
 * ófuncin find_primes
 * Halla los únmeros primos indicados
 * áParmetros:
 * a – arreglo que almacena los primos
 * n – áculos primos encontrar
 */
extern void find_primes ( int * a, unsigned n );

int main()
{
    int status;
    unsigned i;
    unsigned max;
    int * a;

    printf ("¿áCuntos primos desea encontrar Ud.? ");
    scanf(" %u", &max);

    a = calloc ( sizeof(int), max);

    if ( a ) {

        find_primes (a,max);

        /* imprime losú 20 ltimos primos encontrados */
        for(i= ( max > 20 ) ? max - 20 : 0; i < max; i++ )
            printf (" %3d %d\n", i+1, a[i]);

        free(a);
        status = 0;
    }
    else {
        fprintf ( stderr , "No se puede crera el arreglo de %u ints\n", max);
        status = 1;
    }

    return status;
}

```

fprime.c

A continuación la rutina en ensamblador:

```

1  segment .text
2      global _find_primes
3  ;
4  ; function find_primes
5  ; Encuentra el número indicado de primos
6  ; Parámetros:
7  ;   array - arreglo que almacena los primos
8  ;   n_find - cuántos primos encontrar
9  ; Prototipo de C
10 ;extern void find_primes( int * array, unsigned n_find )
11 ;
12 %define array      ebp + 8
13 %define n_find     ebp + 12
14 %define n          ebp - 4          ; número de primos a encontrar
15 %define isqrt      ebp - 8          ; el piso de la raíz cuadrada de guess
16 %define orig_cntl_wd  ebp - 10      ; palabra original de control
17 %define new_cntl_wd  ebp - 12      ; nueva palabra de control
18
19 _find_primes:
20     enter    12,0                    ; Hace espacio para las variables locales
21
22     push    ebx                      ; guarda las posibles variables registro
23     push    esi
24
25     fstcw   word [orig_cntl_wd]      ; toma la palabra de control actual
26     mov     ax, [orig_cntl_wd]
27     or      ax, 0C00h                ; fija el redondeo de los bits a 11 (truncar)
28     mov     [new_cntl_wd], ax
29     fldcw   word [new_cntl_wd]
30
31     mov     esi, [array]              ; esi apunta a array
32     mov     dword [esi], 2            ; array[0] = 2
33     mov     dword [esi + 4], 3        ; array[1] = 3
34     mov     ebx, 5                    ; ebx = guess = 5
35     mov     dword [n], 2              ; n = 2
36 ;
37 ; Este bucle externo encuentra un nuevo primo en cada iteración,
38 ; que se añade al final del arreglo. A diferencia del primer programa de

```

```

39 ; primos, esta función no determina los primos dividiendo por todos
40 ; los números impares. Sólo divide por los números primos que ya se
41 ; han encontrado. (Esta es la razón por la cual ellos se han almacenado
42 ; en el arreglo)
43
44 while_limit:
45     mov     eax, [n]
46     cmp     eax, [n_find]           ; while ( n < n_find )
47     jnb     short quit_limit
48
49     mov     ecx, 1                  ; ecx se usa como índice del arreglo
50     push    ebx                     ; almacena guess en la pila
51     fild    dword [esp]             ; carga guess en la pila del coprocesador
52     pop     ebx                     ; saca guess de la pila
53     fsqrt                    ; halla sqrt(guess)
54     fistp   dword [isqrt]           ; isqrt = floor(sqrt(guess))
55
56 ; Este blucle interno divide guess por los números primos ya encontrados
57 ; hasta que encuentre un factor primo de guess (que significa que guess
58 ; no es primo) o hasta que número primo a dividir sea más grande que
59 ; floor(sqrt(guess))
60 ;
61 while_factor:
62     mov     eax, dword [esi + 4*ecx] ; eax = array[ecx]
63     cmp     eax, [isqrt]             ; while ( isqrt < array[ecx]
64     jnbe    short quit_factor_prime
65     mov     eax, ebx
66     xor     edx, edx
67     div     dword [esi + 4*ecx]
68     or      edx, edx                 ; && guess % array[ecx] != 0 )
69     jz      short quit_factor_not_prime
70     inc     ecx                     ; intenta el próximo primo
71     jmp     short while_factor
72
73 ;
74 ; found a new prime !
75 ;
76 quit_factor_prime:
77     mov     eax, [n]
78     mov     dword [esi + 4*eax], ebx ; suma al final de arreglo
79     inc     eax
80     mov     [n], eax                ; inc n

```

```
81
82 quit_factor_not_prime:
83     add     ebx, 2                ; intenta con el impar siguiente
84     jmp     short while_limit
85
86 quit_limit:
87
88     fldcw   word [orig_cntl_wd]   ; restaura la palabra de control
89     pop     esi                   ; restaura las variables de registro
90     pop     ebx
91
92     leave
93     ret
```

prime2.asm


```
1 global _dmax
2
3 segment .text
4 ; function _dmax
5 ; retorna el mayor de dos argumentos double
6 ; Prototipo de C
7 ; double dmax( double d1, double d2 )
8 ; Parámetros:
9 ;   d1 - primer double
10 ;   d2 - segundo double
11 ; Valor de retorno:
12 ;   El mayor de d1 y d2 (en ST0)
13 %define d1    ebp+8
14 %define d2    ebp+16
15 _dmax:
16     enter    0, 0
17
18     fld     qword [d2]
19     fld     qword [d1]          ; ST0 = d1, ST1 = d2
20     fcomip  st1                ; ST0 = d2
21     jna     short d2_bigger
22     fcomp   st0                ; pop d2 de la pila
23     fld     qword [d1]          ; ST0 = d1
24     jmp     short exit
25 d2_bigger:                      ; si d2 is max, no hace nada
26 exit:
27     leave
28     ret
```

Figura 6.7: Ejemplo de FCOMIP

```
1 segment .data
2 x          dq  2.75          ; convertido a formato double
3 five       dw  5
4
5 segment .text
6     fild    dword [five]      ; ST0 = 5
7     fld     qword [x]         ; ST0 = 2.75, ST1 = 5
8     fscale                          ; ST0 = 2.75 * 32, ST1 = 5
```

Figura 6.8: Ejemplo de FSCALE