XIX. Szybki Umysł : Optymalizacja obliczeniowa

Jaka jest różnica między dobrym programistą a złym? Możliwe, że możesz wymyślić listę 20 rzeczy, które oddzielają dobro od zła. Moim zdaniem jednak wyróżnikiem nie jest wiedza, którą można gromadzić, ale raczej proces myślenia. Nie ma znaczenia, czy istnieją pewne algorytmy, których nie znasz; możesz łatwo pobrać te informacje z sieci. Ważne jest to, aby rozwiązać problem w oryginalny sposób, dostosowany do danego problemu. Podobnie jak większość rzeczy, łatwiej to powiedzieć niż zrobić. Jest to odpowiednik mówienia ludziom, aby nie zawracali sobie głowy zapamiętywaniem algorytmu qsort, i zamiast tego rozumieją proces, który za tym stoi - lub, bardziej odpowiednio, proces myślenia związany z rozwiązaniem problemu. W przypadku optymalizacji niezwykle ważne jest zrozumienie platformy, na której hakujesz. Coś takiego jak podział, który może być rozsądnie szybki na PC, może być makabrycznie wolne na procesorze ARMS, który nie ma instrukcji podziału sprzętu. W tej części omówiono nie tylko komputery PC, ale także urządzenia wbudowane - zazwyczaj są to pozbawione komputera wersje. Aby zrozumieć wady tych urządzeń, zaczniesz od zaglądania w procesor / jednostkę przeliczeniową FPU, aby zobaczyć, jak sprawy są obsługiwane na najniższym poziomie w sensie matematycznym. (Jednostka FPU jest jednostką zmiennoprzecinkową procesora i jest odpowiedzialna za wykonywanie złożonej matematyki za spławikami.) Jest to ważny krok w zrozumieniu, dlaczego jedna funkcja może być wolniejsza od innej. W tej części znajdziesz wiele metod zaprojektowanych w celu zastąpienia funkcji używanych na komputerze, a po drodze kilka użytecznych optymalizacji komputera. Krótko mówiąc, tematy poruszane w tej części są następujące:

■ Format zmiennoprzecinkowy IEEE i arytmetyka

■ Format stałoprzecinkowy i arytmetyka

■ Arytmetyka warunkowa

Punkty stałe

Pracujesz więc nad jednym z tych nowoczesnych urządzeń przenośnych i po prostu odkryłeś, że nie mają absolutnie żadnego wsparcia zmiennoprzecinkowego. Na szczęście twój kompilator obsługuje zmiennoprzecinkowe oprogramowanie, ale naprawdę potrzebujesz całej prędkości, którą możesz wycisnąć z tego urządzenia. W nieśmiertelnych słowach Keanu Reevesa, co robisz? Punkty zmienne są zbyt wolne, a liczby całkowite nie mają dokładności dziesiętnej; czy naprawdę utknąłeś w błocie? Cóż, niezupełnie. Na urządzeniach wbudowanych, takich jak najnowsze telefony i urządzenia PDA, jednostki FPU nie wchodzą w grę. Nawet jeśli nie mają FPU w najnowszych procesorach, działać tak powoli, że nie będzie nawet rozważyć zastosowanie, chyba że potrzebny zakres mógłby dostarczyć lub po prostu nie dbają o prędkości co w grze jest rzadki przypadek . Zamiast tego programiści zaczęli używać stałego punktu. W przeciwieństwie do zmiennoprzecinkowego, który opisuje liczbę, w której przecinek "unosi się" w pobliżu, numery o stałym punkcie mają przecinek ustalony na ustaloną z góry wartość. Co jest jeszcze lepsze? Jeśli pracujesz w przestrzeni osadzonej, istnieje już wiele standardów, które umożliwiają stały punkt jako dane wejściowe. Standardy takie jak OpenGL-ES z łatwością przyjmują ustalone punkty jako dane wejściowe, co sprawia, że zrozumienie ustalonych punktów staje się jeszcze ważniejsze. Ideą stałych punktów jest użycie racjonalnej reprezentacji liczb. Każda wymierna liczba może być zapisana w postaci a-b, gdzie aib to dwie liczby całkowite. Problem z użyciem tej ogólnej definicji jest taki, że podział jest niesamowicie powolny, więc wyraźnie chcesz trzymać się z dala od tego. Ale jeśli na przykład masz naprawić wartość b, a użyjesz wartości, która jest czystą potęgą 2, możesz po prostu przesunąć w prawo, aby uwzględnić podział. Przesunięcie to operacja, która pobiera zmienną całkowitą i przesuwa bity w lewo lub w prawo. Jeśli przekonwertować z dziesiętnych binarny i odwrotnie stosując przesunięcie, można zauważyć, że przesunięcie liczbę binarną naprawdę mnoży lub dzieli liczbę przez dwa, w zależności po której stronie występuje przesunięcie. To dość wydajna

operacja. Podsumowując, dla każdej liczby a-b gdzie b jest rzeczywiście niejawna, a potęga 2, tylko przenosi wartość a dookoła. Ponownie, zakładając, że b jest potęgą 2, jeśli ma się bliżej a, można zauważyć, że bity przyporządkowane do części dziesiętnych (mantysy) są całkowicie różne od bitów przypisanych do liczby całkowitej. To jest piękno tej techniki: przypisuje dolne bity do mantysy i górne bity do części całkowitej. Na przykład, gdybyś wybrał wartość b = 4, każda liczba a byłaby reprezentowana w jawnej formie a / b. Zatem powinieneś być w stanie zobaczyć, że 2 (ponieważ 22 = 4) to liczba bitów przypisanych do mantysy, a pozostałe bity są przypisane do liczby całkowitej. Jeśli weźmiesz liczbę mniejszą niż 4, w ogóle nie otrzymasz liczby całkowitej. Z drugiej strony, dowolna wielokrotność liczby 4 da ci liczbę całkowitą 8/4 = 2, a wartości częściowe dadzą ci wartość z pewnym stopniem dokładności 9/4 = 2,25. Powinno być całkiem jasne, że jest to dość szybkie, ponieważ zmiana jest dość łatwa implementować koncepcyjnie. W rzeczywistości przesunięcie jest zazwyczaj tak szybkie, jak dodanie. Jest to dobry sposób na ustawienie rzeczy, ponieważ jeśli b jest ustalony, nie musisz przenosić żadnej dodatkowej wartości, dlatego każdy bit precyzji jest przypisany do a dla liczby a/b. Istnieje inny sposób na zapisanie ustalonego punktu: Możesz oddzielić mantysę od części całkowitej, mnożąc je przez pomnożenie przez skalar. Tak więc punkt stały a/b reprezentowany jako z niejawnym b może być rzeczywiście postrzegany jako 'liczba całkowita • b + mantysa'. Jeśli odwrócisz proces, dzieląc niejawny stały punkt przez b, zauważysz, że jest to logiczna dedukcja, ponieważ jeśli podzielisz wyrażenie przez b, pozostaniesz tylko z częścią całkowitą (pamiętaj, że jest to dzielenie całkowite) . W rzeczywistości b określa, jak dużą precyzję otrzymasz z ustalonego punktu. Ponieważ masz tylko 32 bity, zwiększenie dokładności dziesiętnej również zmniejsza maksymalną liczbę, którą możesz reprezentować. Jeśli zdecydujesz się umieścić 16 bitów dokładności dla miejsc po przecinku, pozostawi to tylko 16 bitów dla części całkowitej. To jest coś, o czym musisz pomyśleć przy wyborze właściwej liczby całkowitej: stosunek mantysy do twoich potrzeb. Czy potrzebujesz dużego zakresu lub małych i dokładnych wartości? To wszystko jest dobre, ale jak właściwie przekonwertować wartość zmiennoprzecinkową na wartość stałoprzecinkową? To całkiem proste. Jeśli przyjmujesz stałą wartość k, chcesz reprezentować tę wartość w notacji o stałym punkcie a-b bez utraty znacznej precyzji. Wszystko, co musisz zrobić, aby przekonwertować na stały punkt, to pomnożyć zmiennoprzecinkowe przez b, ponieważ podział przez b jest niejawny. Zatem każdy zmiennoprzecinkowy k ma równoważną stałą wartość kb. A co z konwersją z powrotem? To proste: jest to proces odwrotny. Trzeba po prostu podzielić przez b, usuwając w ten sposób niejawny podział. W rzeczywistości zostało to już zrobione w poprzednim przykładzie. Jeśli odwrócisz proces przez pomnożenie przez 4, otrzymasz swój pierwotny stały punkt. Interesujące jest również spojrzenie na punkty stałe pod względem bitowej reprezentacji pamięci, ponieważ istnieje wyraźne rozróżnienie między całkowitą częścią liczby i mantysy. Jeśli spojrzysz na rysunek 19.1, który reprezentuje 32-bitową stałą wartość w pamięci, zauważysz, że całkowita część bitów i mantysa (czyli liczba następująca po punkcie) mają wspólną wartość 32-bitową .

Część Całkowita x bitów	Mantysa (32–x) bitów

W ten sposób możesz łatwo obliczyć wartość całkowitą za pomocą zwykłego przesunięcia (które jest naprawdę równe podziałowi przez b) przez popchnięcie bitów w prawo tak, że mantysa jest zredukowana do 0 bitów, a część całkowita jest reprezentowana przez całe 32 bity . To takie proste. Ponieważ dzielenie przez b jest niejawne, możesz pobrać część całkowitą przez przesunięcie w prawo.

Pozostawione same sobie te liczby są nieco nudne. Aby były przydatne, musisz dać im zestaw operacji. Zobaczmy więc, jak możesz obliczyć różne ważne operacje na tym zestawie.

Dodawanie punktów stałych

W przypadku dodawania punktów stałych otrzymujesz coś podobnego do punktów zmiennoprzecinkowych, używając tylko liczb całkowitych. Aby to było warte twoich kłopotów, musisz jednak upewnić się, że podstawowe operacje, które chcesz zastosować na tych liczbach, są znacznie szybsze niż operacje zmiennoprzecinkowe. Aby to zrobić, musisz rzucić okiem na operacje w punkcie stałym. Poza tym, jeśli planujesz używać stałych punktów na urządzeniach wbudowanych (i powinieneś, jeśli nie masz wydajnego FPU), będziesz potrzebował ich jako czystej separacji części całkowitej i części dziesiętnej. Rozważmy dwie liczby stałe a i c. Aby ułatwić ci pracę, napiszmy je w notatniku sb. W ten sposób masz dwie liczby: a = ib i c = jb. Każda operacja arytmetyczna na ustalonym punkcie powinna logicznie dać inny stały punkt, więc wynik dowolnego stałego punktu powinien również mieć postać sb. Jak się okazuje, jeśli dodasz dwie liczby punktów stałych w ich reprezentacji zmiennoprzecinkowej, otrzymasz:

$$a + c = ib + jb = (i + j)b$$

Ostateczna odpowiedź znajduje się w notacji o stałym punkcie, więc jesteś bezpieczny. To jest najbardziej imponujące ponieważ oznacza to, że dodanie dwóch liczb stałych jest nie mniej skomplikowane niż dodanie dwóch liczb całkowitych. Naprawiono punkty są dość szybko dodawać i odejmować, ponieważ odejmowanie jest po prostu dodaniem liczby ujemnej. Nie możesz zrobić matematyki z notacją a-b, ponieważ musisz pamiętać, że b jest niejawne w wartości.

Mnożenie stałopunktowe

Dodawanie był powiewny do wykonania, mnożenie jednak da ci trochę więcej żalu. Gdy pomnożymy dwie liczby stałe, ab i cb, wynik, jakiego można się spodziewać, to acb. Niestety, jeśli pomnożysz dwie liczby punktów stałych, otrzymasz dodatkowe mnożenie przez b. To nie koniec świata; możesz po prostu pomnożyć swoje dwie liczby, a następnie przesunąć w prawo, aby podzielić przez b. Jedynym problemem, który możesz napotkać, jest to, że pomnożenie a i b razem może przekroczyć 32-bitowy limit. W tym celu będziesz musiał żonglować trochę więcej. Dosłownie. Na razie jest to proces, który pokazuje, dlaczego musisz dzielić przez b po mnożeniu:

$$(b \cdot a)(b \cdot c) = b^2 a \cdot c$$

Jak radzisz sobie z problemami z przepełnieniem, które możesz napotkać? Możesz to dostroić, aby dopasować swoją własną precyzję bitową, ale pomysł jest taki, że możesz podzielić swoje zmienne a i c na dwie oddzielne części w tym samym rozmiarze. Załóżmy, że a = [wx] i c = [yz] są takie, że dla 32-bitowej wartości każdy podskładnik reprezentuje 16 wysokich / niskich bitów. Niech k reprezentuje wartość, która wynosi połowę mocy przenoszonej przez a i c. Zatem, dla wartości 32-bitowych, k będzie wynosić 65536, co zasadniczo przesuwa wartość w górnych 16 bitach. Wynika następujące rozumowanie:

$$
\begin{aligned}
b^2 a \cdot c &= (k \cdot w + x)(k \cdot y + z) \\
&= k^2 \cdot w \cdot y + k(w \cdot z + x \cdot y) + x \cdot z \\
&= k^2 \cdot w \cdot y + k(w \cdot z + x \cdot y) + x \cdot z + k^2 \left\lfloor \frac{w \cdot z + x \cdot y}{k} \right\rfloor - k^2 \left\lfloor \frac{w \cdot z + x \cdot y}{k} \right\rfloor \\
&= k^2 \left(w \cdot y + \left\lfloor \frac{w \cdot z + x \cdot y}{k} \right\rfloor \right) + \left(\left((w \cdot z + x \cdot y) + -k^2 \left\lfloor \frac{w \cdot z + x \cdot y}{k} \right\rfloor \right) + x \cdot z \right)
\end{aligned}
$$

Brzmi to jak niesamowicie kręty i długi proces, ale pod względem czasu obliczeń jest stosunkowo prosty. Zwróć uwagę, że dwa nawiasy pierwszego poziomu są rozdzielone. W pierwszym przypadku wszystko jest mnożone przez kwadrat k, podczas gdy druga strona nie ma żadnego niejawnego mnożenia. Kwadrat k jest przesunięciem jednego pełnego elementu. Ponownie, jeśli wrócisz do 32-bitowego przykładu, kvv w rzeczywistości przesunie całą wartość o 32 bity. Tak więc wartość zawarta w nawiasach jest faktycznie górnymi 32 bitami mnożenia, podczas gdy pozostałe nawiasy niosą niższe 32 bity. Funkcja posadzki zapewnia, że liczba nie przepełnia się. Jeśli spojrzysz na elementy w kategoriach ile bitów precyzji są przenoszone, możesz zobaczyć, że wszystko pasuje jak pokazano tutaj dla zmiennych n bitów (pamiętaj, że reprezentują one liczbę bitów, a nie rzeczywiste liczby, więc dodanie i multiplikacje są różne pod tym względem):

$$
\begin{aligned}
&= \left\{\frac{n}{2}\right\}^2 \left(\left\{\frac{n}{2}\right\} \cdot \left\{\frac{n}{2}\right\} + \left\lfloor \frac{\left\{\frac{n}{2}\right\} \cdot \left\{\frac{n}{2}\right\} + \left\{\frac{n}{2}\right\} \cdot \left\{\frac{n}{2}\right\}}{\left\{\frac{n}{2}\right\}} \right\rfloor \right) + \left(\left\{\frac{n}{2}\right\} \left(\left\{\frac{n}{2}\right\} \cdot \left\{\frac{n}{2}\right\} + \left\{\frac{n}{2}\right\} \cdot \left\{\frac{n}{2}\right\} \right) - \left\{\frac{n}{2}\right\} \left\lfloor \frac{\left\{\frac{n}{2}\right\} \cdot \left\{\frac{n}{2}\right\} + \left\{\frac{n}{2}\right\} \cdot \left\{\frac{n}{2}\right\}}{\left\{\frac{n}{2}\right\}} \right\rfloor \right) + \left\{\frac{n}{2}\right\} \\
&= \{n\} \left(\{n\} + \left\lfloor \frac{\{n\} + \{n\}}{\left\{\frac{n}{2}\right\}} \right\rfloor \right) + \left(\left\{\frac{n}{2}\right\} (\{n\} + \{n\}) - \left\{\frac{n}{2}\right\} \left\lfloor \frac{\{n\} + \{n\}}{\left\{\frac{n}{2}\right\}} \right\rfloor \right) + \{n\} \\
&= \{n\} \left(\{n\} + \left\{\frac{n}{2}\right\} \right) + \left(\left\{\frac{n}{2}\right\} \left(\{n\} - \left\{\frac{n}{2}\right\} \cdot \left\{\frac{n}{2}\right\} \right) \right) + \{n\} \\
&= \{n\}(\{n\}) + \left(\left\{\frac{n}{2}\right\} (\{n\} - \{n\}) \right) + \{n\} \\
&= \{n\}(\{n\}) + (\{n\} + \{n\}) \\
&= \{n\}(\{n\}) + (\{n\})
\end{aligned}
$$

Oczywiście dodatki mogą powodować przepełnienia, więc w każdym z tych przypadków trzeba nosić od jednego do najbardziej znaczących n bitów, ale nie jest tak źle. Dzieje się tak dlatego, że jeśli weźmiesz pod uwagę fakt, że proces zazwyczaj odcina najważniejsze bity, gdy mnożenie się przepełni, otrzymasz znacznie łatwiejszą do opanowania formułę, w której nawiasy kwadratowe są używane do reprezentacji nasyconego mnożenia:

$$
b^2 a \cdot c = k^2 \left(w \cdot y + \left\lfloor \frac{w \cdot z + x \cdot y}{k} \right\rfloor \right) + \left(\left[k(w \cdot z + x \cdot y) \right] + x \cdot z \right)
$$

W tym momencie możesz podzielić każdą stronę przez b, aby pobrać ostateczne rozwiązanie w ustalonym punkcie. Oczywiście nadal można uzyskać przepełnienie za pomocą tej techniki, ale przepełnienia nie będą wynikiem złych obliczeń. Przeciwnie, wystąpią, ponieważ zmienne mogą

zawierać tylko tyle informacji, podobnie jak to, jak mnożenie dwóch liczb całkowitych może być przepełnione w 32 bitach. Chociaż jest to ogólna metoda obsługi tego typu sytuacji, możesz łatwo dodać kilka skrótów, jeśli zobaczysz, że jedno z obliczeń (powiedzmy, górne 32 bity) wynosi zero. Poniższy kod pokazuje, jak można go wdrożyć:

```
FixedPoint FixMul(FixedPoint A, FixedPoint B)
{
        char           Negative;
        unsigned long AL, AH, BL, BH, RL, RH;
        // A = (AH << 16) + AL, B = (BH << 16) + BL, R = (RH << 16) + RL

        if (!(A | B))
                return 0;

        if (A < 0) {
                A               = -A;
                Negative        = 1;
        } else
                Negative        = 0;

        if (B < 0) {
                B               = -B;
                Negative        ^= 1;
        }

        AL = (unsigned long)(A & 0xFFFF);
        AH = (unsigned long)(A >> 16);
        BL = (unsigned long)(B & 0xFFFF);
        BH = (unsigned long)(B >> 16);
```

```
    // LSW
    RL = AL * BL;         // Calculate the lowest 32 bit value
    RH = AH * BH;         // Calculate the highest 32 bit value
                           // Calculate the middle value
    AL = (AL * BH) + (BL * AH);       // It will never overflow because the
                                        // highs are signed thus 15 bits
    AH = AL << 16;
    AL = RL + AH;         // Add the low 16 bits to the high 16 bits

    If ((AL < AH) || (AL < RL))
         RH++;            // Add the carry bit if it exists

    RH += (AL >> 16);
    RL  = (RL >> PRECISION) + (RH << (32 - PRECISION));

    return (FixedPoint)(Negative ? -((FixedPoint)RL) : RL);
}
```

Wygląda na to, że trzeba bardzo dużo pracy, aby uzyskać mnożenie. Okazuje się jednak, że jest znacznie wydajniejsza niż implementacja zmiennoprzecinkowa. Sekcja o punktach zmiennoprzecinkowych pomoże ci docenić, dlaczego tak się dzieje, ale na razie, wiedz, że na komputerze, sterownik audio Linuksa przetestował pływaki w porównaniu do ustalonych i odkrył, że średnio, operacje o stałym punkcie były dwa do trzy razy szybciej. Pamiętaj, że jest to system, który jest faktycznie wyposażony w jednostkę FPU. Dlaczego więc powinieneś zawracać sobie głowę używaniem pływaków? To proste. Naprawione punkty nie mają bardzo dużego zasięgu, co oznacza, że nie można z nimi reprezentować dużych liczb (lub bardzo małych liczb). Jak zobaczycie później, liczby stałe są bardziej precyzyjne niż pływaki, ale brakuje im zasięgu. Niektóre rzeczy na PC, takie jak czcionki TrueType, są pisane przy użyciu ustalonych punktów, ponieważ oferują większą precyzję i szybkość. Jednak w urządzeniach wbudowanych takich jak Game Boy lub telefon rzeczy działają do sześciu razy wolniej w zależności od urządzenia.

Dzielenie stałopunktowe

Mnożenie dwóch wartości stałoprzecinkowych jest koncepcyjnie bardzo proste. Co sprawia, że bardziej skomplikowany jest fakt, że musisz konsekwentnie utrzymywać precyzję na wysokim poziomie. Podział, jak można się spodziewać, jest koncepcyjnie prosty, ale nie tak, kiedy przychodzi czas na wdrożenie. Następująca podobna logika występuje dla dwóch wartości stałoprzecinkowych ab i cb:

$$\frac{ab}{cb} = \frac{a}{c}$$

W tym scenariuszu brakuje mnożenia przez b. Dlatego musisz pomnożyć swój finał odpowiedź b, aby uzyskać prawidłowe rozwiązanie. To kolejna konceptualnie łatwa operacja, ale jest dość bolesna do wdrożenia. Aby uzyskać pełną dokładność, nie można dzielić w pierwszej kolejności. Najpierw należy pomnożyć, a następnie podzielić przez c, co po prostu sprawia, że rzeczy są gorsze niż samo rozmnażanie. Jeśli podzielisz a na c, otrzymamy całkowitą część podziału. Tracisz wszystkie szczegóły dotyczące mantysy, co nie jest dobre. Łatwo potem pomnożyć liczbę całkowitą przez b. Można obliczyć pozostałą część podziału, aby otrzymać mantysę, ale nadal trzeba podzielić pozostałą przez c, co nadal

nie rozwiązuje prawdziwego problemu. To, co musisz zaimplementować, to algorytm podziału. Wiesz, że z definicji reszta jest mniejsza niż c. Jeśli podwoisz resztę, a kończy się ona na wartości większej niż c, co ci to mówi? Co ciekawe, informuje, że reszta to co najmniej połowa c. W ten sposób możesz dodać 0,5 do swojej wartości, a mnożenie nowo odkrytej wartości przez c będzie znacznie bliższe rzeczywistej wartości podziału bez większego od niej. Na tym etapie pytanie brzmi, czy dodać 0,5 do swojej wartości. Jeśli, na przykład, bierzesz 63/128, podwojenie 63 pokazuje, że 63 nie jest co najmniej w połowie z 128; w związku z tym nie powinieneś dodawać wartości 0,5 do swojej wartości. Obliczenia zmiennoprzecinkowe pokazują, że liczba ta jest rzeczywiście mniejsza niż 0,5. Jeśli dodałeś 0.5 do wartości, nigdy nie byłbyś w stanie wygenerować mniejszej wartości przez dodanie dowolnej liczby dodatniej. Dokładniej, dodając dowolną inną liczbę dodatnią, nigdy nie byłbyś w stanie wygenerować funkcji, która zbiega się w kierunku prawdziwej answer.Możesz zastosować ten proces rekurencyjnie, aby uzyskać jak najbardziej precyzyjny szacunek swojego numeru. Jeśli dwa razy reszta jest większa niż c, możesz odjąć od niej c; jeśli nie, numer jest już ustawiony, aby sprawdzić 0.25. Poniższy przykład pokazuje, jak to się dzieje:

$$a = 63$$
$$c = 128$$

Math Step	Number (Total of Adder)	Adder Step
$\frac{a}{c} = \frac{63}{128} = 0$	0.0	1
$2\frac{63}{128} = \frac{126}{128} = 0$	0.0	0.5
$2\frac{126}{128} = \frac{252}{128} = 1 + \frac{124}{128}$	0.25	0.25
$2\frac{124}{128} = \frac{248}{128} = 1 + \frac{120}{128}$	0.375	0.125
$2\frac{120}{128} = \frac{240}{128} = 1 + \frac{112}{128}$	0.4375	0.0625
$2\frac{112}{128} = \frac{224}{128} = 1 + \frac{96}{128}$	0.46875	0.03125
$2\frac{96}{128} = \frac{129}{128} = 1 + \frac{64}{128}$	0.484375	0.015625
$2\frac{64}{128} = 1$	0.4921875	0.0078125

W tym przypadku byłeś w stanie powtórzyć proces wystarczająco długo, aby uzyskać dokładne rozwiązanie - choć nie zawsze jest to możliwe. Dzieje się tak, ponieważ ułamki mogą dać nieskończoną serię. Wiesz, że b określa, ile bitów jest dostępnych dla mantysy, a zatem powinieneś powtórzyć ten proces dla jedynej liczby bitów, które b może reprezentować. W kodzie źródłowym można zaimplementować algorytm, jak pokazano poniżej:

```
typedef unsigned long FixedPoint;

FixedPoint FixDiv(FixedPoint A, FixedPoint C)
{
        unsigned long Rem;
        char Negative = 0;

        if (A < 0) {
                A = -A;
                Negative ^= 1;


                }
        if (C < 0) {
                C = -C;
                Negative ^= 1;
        }

        Rem = A % C;
        A   = A / C;

        for (char Bbit = PRECISION ; Bbit-- ;) {
                A   <<= 1;
                Rem <<= 1;
                if (Rem >= (unsigned long)C) {
                        Rem -= (unsigned long)C;
                        A++;
                }

        }

        return (FixedPoint)(Negative ? -((FixedPoint)A) : A);
}
```

Jeśli chodzi o szybkość, czy ta funkcja jest naprawdę lepsza niż ta, którą zapewnia procesor? Jedno jest pewne: nie jest tak ładne jak mnożenie czy dodawanie. Ale jeśli pobierzesz tabelę licznika cykli procesorów, zauważysz, że podział jest zawsze znacznie wolniejszy niż mnożenie. Weźmy na przykład dzisiejsze Pentium 4. Mnożenie odbywa się w siedmiu cyklach procesora, a podział sięga aż 23 razy więcej niż trzykrotność liczby. Czy spodziewałbyś się, że funkcja podziału będzie trzy razy wolniejsza od mnożenia? To zależy od tego, gdzie uruchomisz ten kod. Na komputerze jest mniej więcej równoznaczny z podziałem, ledwo go pokonując. Na urządzeniu osadzonym, gdzie pojęcia takie jak przewidywanie rozgałęzień i wykonywanie poza kolejnością nie są w porządku dziennym, naprawdę zyskujesz tyle, ile robisz z mnożeniem, dwa do trzech razy więcej niż spławikami - znowu dość znaczna ilość.

Pierwiastek Kwadratowy Liczby Całkowitej

Tak łatwo jest poprosić komputer o podanie pierwiastka kwadratowego liczby, ale jak komputer go przetwarza? W urządzeniach wbudowanych, jeśli nie masz jednostki zmiennoprzecinkowej, bardzo możliwe jest, że funkcja pierwiastka kwadratowego jest całkowicie wykluczona. Jeśli tak jest, to musisz sam wdrożyć tę funkcję. Na szczęście istnieje łatwy i skuteczny sposób na jego wdrożenie. Pomysł opiera się na tej używanej do podziału na punkty stałe. Używasz heurystyki, aby określić, czy powinieneś się nieco przełączyć, a ta heurystyka jest przeznaczona tylko do tego obliczyć kwadrat twojego zgadywania. Zaczynasz od sprawdzenia najbardziej znaczącego bitu pierwiastka kwadratowego (czyli 15 bitowej 32-bitowej wartości pierwiastka kwadratowego). Aby to zrobić, przełącz początkowo bit; jeśli kwadrat wartości uzyskanej przez przełączenie bitu jest mniejszy niż kwadrat pierwiastka kwadratowego, którego szukasz, to bit powinien być włączony. Możesz zastosować ten proces na wszystkich dolnych częściach bitów, aby obliczyć pierwiastek kwadratowy. Innymi słowy, po sprawdzeniu, że przełączenie bit 15 daje liczbę, która jest mniejsza niż pierwiastek kwadratowy z liczby, której szukasz, możesz zastosować ten sam test dla bitu 14, zachowując poprzednie bity przełączane zgodnie z testem. Poniższy kod ilustruje ideę:

```
unsigned short sqrt(unsigned long X)
{
        unsigned long Guess = 0;
        unsigned long Bit = 1 << 15;

        do {
                Guess ^= Bit;
                // Verify if the bit should be toggled or not
                if (Guess * Guess > X)
                        Guess ^= Bit;
        } while (Bit >>= 1);

        return Guess;
}
```

To są szkolne rzeczy. Na przykład, weź 49 i oblicz jego pierwiastek kwadratowy. Wiesz, że 15-ty bit nie jest tak małą liczbą, więc możesz go pominąć. Zamiast tego zacznij od 8, z czwartym bitem. Po dotarciu do ostatniego fragmentu, zatrzymujesz się. W ten sposób można stwierdzić, że sqrt (49) = 7. Ten algorytm jest dobry nie tylko dla liczb całkowitych, ale także dla stałych punktów. Jedyną rzeczą, na którą musisz uważać, jest obliczenie trafnego wyniku. (Kod źródłowy takiej wersji zostanie podany na końcu tej sekcji.) Jest to bardzo prosty algorytm, ale jest również bardzo nieefektywny. Namnażanie jest pasożytem, co byłoby miłe zlikwidować tutaj. Możesz to łatwo osiągnąć, jeśli zastosujesz na nim dwukrotne zróżnicowanie forward. To, co otrzymasz, jest następujące: gdzie x to twoje przypuszczenie, a b to wartość dodana przez przełącznik bitowy:

$$\text{Guess}(x) = x^2$$

$$\text{Guess}(x+b) = x^2 + 2xb + b^2$$

$$= \text{Guess}(x) + \left(2xb + b^2\right)$$

and

$$\text{Guess'}(x) = 2xb + b^2$$

$$\text{Guess'}(x+b) = 2(x+b)b + b^2$$

$$= \text{Guess'}(x) + b^2$$

Jeśli śledzisz x^2 i 2x, możesz to osiągnąć. Zabranie kwadratu b jest łatwe, ponieważ możesz podwoić przesunięcie bitu, który chcesz przetestować. A jeśli chodzi o 2xb, możesz go również łatwo obliczyć, ponieważ b jest właściwie tylko przesunięciem 2x. Jeśli śledzisz dwa ostatnie określenia odgadnięcia dwóch ostatnich wyrazów, możesz wyrazić wartości forwardowania jako kombinację jego ostatniej wartości i kwadratu bitów, jak pokazano tutaj:

$$Bit_{n+1} = \frac{Bit_n}{4}$$

$$ForwardGuess_n = Bit_n\left(2Guess + Bit_n\right)$$

$$ForwardGuess_n = \frac{Bit_n}{2}\left(2Guess + \frac{Bit_n}{2}\right)$$

$$= \frac{1}{2}\left(2Guess \cdot Bit_n + \frac{Bit_n^2}{2}\right)$$

$$= \frac{1}{2}\left(2Guess \cdot Bit_n + \frac{Bit_n^2}{2} + \frac{Bit_n^2}{2} - \frac{Bit_n^2}{2}\right)$$

$$= \frac{1}{2}\left(2\left(Guess \cdot Bit_n + Bit_n^2\right) + 2\frac{Bit_n^2}{4}\right)$$

$$= \frac{1}{2}\left(ForwardGuess_n - 2Bit_{n+1}\right)$$

Daje to następujący kod źródłowy:

```
unsigned short sqrt(unsigned long X)
{
        unsigned long Guess, SqrGuess, Bit, TwoSqrBit, ForwardGuess;

        Guess        = 0;
        SqrGuess     = 0;
        Bit          = 1 << 15;
        TwoSqrBit    = 1 << 31;          // 2*Bit^2
        ForwardGuess = 1 << 30;          // 2*Bit*Guess + Bit^2

        do {
                // Verify if the bit should be toggled or not
                if (SquGuess + ForwardGuess <= X) {
                        Guess        |= Bit;
                        SqrGuess     += ForwardGuess;
                        ForwardGuess += TwoSqrBit;
                }

                // Shift everything down to test the next bit
                Bit          >>= 1;
                TwoSqrBit    >>= 2;
                ForwardGuess       = (ForwardGuess - TwoSqrBit) >> 1;
        } while(Bit);

        return Guess;
}
```

Możesz zoptymalizować ten kod źródłowy jeszcze bardziej, jeśli ocenisz ForwardGuess, śledząc aktualny numer bitu, który oceniasz. Jeśli naprawdę jesteś szalony szybkością, możesz łatwo rozwinąć tę pętlę, aby uzyskać jeszcze lepsze wyniki. Poniższy kod źródłowy pokazuje, w jaki sposób należy zaimplementować niepodpisaną wersję:

```
unsigned short sqrt(unsigned long X)
{
        unsigned long Guess = 0, Bit = 0x8000, BitShift = 15;

        do {
                // 2*Bit*Guess + Bit^2
                unsigned long ForwardGuess = (Guess + Guess + Bit) << BitShift;
```

```
            // Verify if the bit should be toggled or not
            if (X  >= ForwardGuess) {
                   Guess        |= Bit;
                   X            -= ForwardGuess;
            }

            BitShift--;
      } while (Bit >>= 1);

      return Guess;
}
```

Ale co ze stałymi punktami? Jeśli chcesz uzyskać stałe punkty, będziesz musiał zrobić dokładnie to samo, ale bądź ostrożniejszy, jeśli chodzi o wartość tego odgadnięcia. Poniższy kod pokazuje taką implementację:

```
#define sqrt_Def_1(Bit)                                        \
{                                                              \
   if (Sqr + (Val << Bit) <= (MMUInt32)a) {                    \
       a    -= (Val << Bit) + Sqr;                             \
       Val |= 1 << (Bit - 1);                                  \
   }                                                           \
                                                               \
   Sqr >>= 2;                                                  \
}

#define sqrt_Def_2(Bit)                                        \
{                                                              \
  if (Sqr + (Val >> (8 - Bit)) <= (MMUInt32)a) {               \
       a    -= (Val >> (8 - Bit)) + Sqr;                       \
       Val |= 1 << (Bit - 1);                                  \
   }                                                           \
                                                               \
   Sqr >>= 2;                                                  \
}
FixedPoint fixSqrt(FixedPoint a)
{
   unsigned long Sqr = 0x40000000;
   unsigned long Val = 0;
   MMD_RUN(long Olda = a);

   // Verify that the result is a real number, not a complex
   MMD_ASSERT1(Olda >= 0, Olda, "I don't want to compute a negative square root!");
```

```
    // Figure out the integer portion of our number
    sqrt_Def_1(16);
    sqrt_Def_1(15);
    sqrt_Def_1(14);
    sqrt_Def_1(13);
    sqrt_Def_1(12);
    sqrt_Def_1(11);
    sqrt_Def_1(10);
    sqrt_Def_1(9);
    // Figure out the first byte of the decimal portion
    sqrt_Def_1(8);
    sqrt_Def_1(7);
    sqrt_Def_1(6);
    sqrt_Def_1(5);
    sqrt_Def_1(4);
    sqrt_Def_1(3);
    sqrt_Def_1(2);
    sqrt_Def_1(1);
    // Discover the last byte of decimal precision
    // 'Bit' starts at 8, thus move everything by 8
    a   <<= 8;
    Val <<= 8;
    Sqr <<= 8;
    sqrt_Def_2(8);
    sqrt_Def_2(7);
    sqrt_Def_2(6);
    sqrt_Def_2(5);
    sqrt_Def_2(4);
    sqrt_Def_2(3);
    sqrt_Def_2(2);
    sqrt_Def_2(1);

    return Val;
}
```

Liczby zmiennoprzecinkowe IEEE

Wielu programistów uważa, że zmienne punkty mogą osiągnąć prawie wszystko. Jeśli wcześniej zaimplementowałeś metodę znajdowania kolizji między elementami gry (np. Rakietą lub karabinem uderzającym w trójkąt), wiesz, że jest to całkowicie nieprawdziwe. Aby naprawdę zrozumieć, dlaczego tak się dzieje i jak zły jest ten efekt, należy zagłębić się w procesor i obserwować, jak się sprawy mają. Prawdopodobnie nie chcesz zaimplementować swojej jednostki zmiennoprzecinkowej, ale zrozumienie matematyki za nią pomoże ci zrozumieć, gdzie są wolne funkcje, a także pomoże ci nauczyć się kilku sztuczek. Szokujące jest to, że nie są liczbami rzeczywistymi. Rzeczywiste liczby mają charakter potencjalnie trzymający nieskończony wzór cyfr dziesiętnych. Pierwiastek kwadratowy z dwóch jest takim przykładem. Jeśli rozwiniesz wartość, nigdy nie powinieneś znaleźć powtarzającego się wzoru. Jednak w świecie gry zajmujesz się skończoną przestrzenią; w związku z tym nie można przedstawiać liczb rzeczywistych bezpośrednio, chyba że zacznie się numery w ich pierwotnej postaci (na przykład pierwiastek kwadratowy z 2). Pływaki nie mogą dokładnie reprezentować π, Ale jak często (w programowaniu gry) potrzebujesz 200-cyfrowej dokładnej wartość π. Punkty zmiennoprzecinkowe są w rzeczywistości liczbami wymiernymi, czyli liczbami, które można zapisać jako ułamek a/b. Może to przynieść nieskończenie dokładne wartości, takie jak 1/3, ale ze względu na skończony aspekt

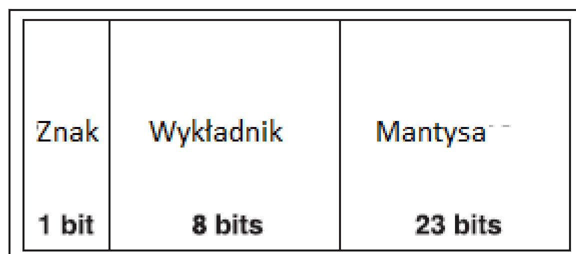
reprezentacji, zawsze będzie skończony wzór w cyfrach dziesiętnych. Punkty zmienne IEEE mogą być dowolnymi z 32, 64 lub 80 bitów na komputerze. Ciekawostką jest to, że pływaki mają całkiem niezły zasięg. Tak więc, koniecznie, niektóre bity w liczbie będą używane do reprezentowania samej liczby, a inne bity będą używane do reprezentowania mocy, przez którą jest pomnożona. Oznacza to, że stały punkt, na przykład, jest faktycznie bardziej precyzyjny niż spławik, jeśli chcesz, aby był. Innym sposobem na stwierdzenie, że możesz reprezentować więcej liczb w zakresie [0, 2^{32}] z long (czyli , stały punkt) niż z single float, który ma tylko 19 bitów dla rzeczywistej liczby. Wielu nie pomyślałoby, że longs są "bardziej precyzyjne" niż single float, ale tak naprawdę są. To wszystko jest interesujące, ale tylko odsuwa prawdziwe pytanie, czyli w jaki sposób pojedyncze pływaki są reprezentowane zgodnie ze standardem IEEE? Okazuje się, że wszystkie formaty zmiennoprzecinkowe można wyrazić za pomocą następującej formuły, w której mantysa może być zapisana jako liczba wymierna od 0 w górę, a wykładnik jest liczbą całkowitą od ujemnej do dodatniej:

$$\pm\left(1+2^{-mantissa}\right)2^{exponent}$$

W szczególności dla 32 -bitowe pływaki, formuła jest następująca, gdzie mantysa waha się od [0, 223-1] i gdzie wykładnik ma zakres [0, 255]:

$$\pm\left(1+\frac{mantissa}{2^{23}}\right)2^{127-exponent}$$

Jeśli zastąpisz ekstrema w równaniu, otrzymasz, że pojedynczy float może wytworzyć liczby o zakresie [-3,4028 • 10^{38}, 3,4028 • 10^{38}] z najmniejsza liczba dodatnia to 1,1754 • 10^{-38}. Jeśli spojrzysz na wszystko poprawnie, zauważysz, że 23 bity są przypisane do mantysy, osiem bitów do wykładnika i jeden bit używany jako znak. Reprezentację pamięci pojedynczego float pokazano na rysunku 19.2.



Podczas projektowania reprezentacji zmiennoprzecinkowej grupa IEEE mogła również zastosować odchylenie dla znaku wartości zmiennoprzecinkowej, aby objąć jeszcze bardziej możliwe wartości, ale tego nie zrobiły. To była dobra decyzja, ponieważ wartości zmiennoprzecinkowe również się zajmują wiele innych rzeczy, z którymi liczba całkowita ma problemy. Ponieważ wiele zestawów bitów jest zbędnych, grupa IEEE zdefiniowała także zbiór wartości, które mogą przenosić wartości zmiennoprzecinkowe. Podczas projektowania reprezentacji zmiennoprzecinkowej grupa IEEE mogła również zastosować odchylenie dla znaku wartości zmiennoprzecinkowej, aby objąć jeszcze bardziej możliwe wartości, ale tego nie zrobiły. To była dobra decyzja, a mianowicie dlatego, że wartości zmiennoprzecinkowe również zajmują się wieloma innymi rzeczami, z którymi mają problemy z liczbami całkowitymi. Ponieważ wiele zestawów bitów jest zbędnych, grupa IEEE zdefiniowała również zbiór wartości, które wartości zmiennoprzecinkowe mogą przenosić. Zero i nieskończoność są dość oczywistymi wartościami. Jak tylko przepełnisz zmiennoprzecinkowy zasięg, dostajesz nieskończoność. W przypadku niedoboru otrzymasz 0 (pozytywny lub negatywny, w zależności od tego skąd pochodzi). Bardziej interesujące są NaN. NaN oznacza Not a Number (Nie liczba). Można go łatwo uzyskać, jeśli na

przykład obliczono pierwiastek kwadratowy z liczby ujemnej. Q poprzedzające NaN oznacza Quiet. Dlatego podczas wykonywania operacji nie jest sygnalizowany żaden wyjątek. Z drugiej strony S oznacza sygnał, który uruchamia wyjątek. Semantycznie, QNaN oznacza nieokreśloną operację, podczas gdy SNaN oznacza niepoprawną operację. Jest jedna ostatnia nieprzyjemna koncepcja, o której musisz wiedzieć: wartości normalne w stosunku do wartości denormalnych. To, co widzisz do tej pory, pasuje do kategorii normalnych wartości. Wykładnik równy 0 o niezerowej maniaczy identyfikuje wartość denormalną. Denormalne wartości są trochę inne od swoich kuzynów, próbując jeszcze bardziej rozszerzyć nieskończenie małą granicę. równanie wartości denormalnej jest następujące:

$$\pm\left(\frac{mantissa}{2^{23}}\right)2^{-126}$$

Krótko mówiąc, jeśli możesz usunąć dominującą wartość normalną, możesz łatwo uzyskać wartość denormalną. Wszystko, co możesz kontrolować w takiej wartości, to mantysa. Zatem wartości, które to równanie może generować, mają zakres [$1.1754 \bullet 10^{-38}$, $1.4012 \bullet 10^{-45}$]. Jeśli myślisz o implementacji własnej wersji zmiennoprzecinkowej w oprogramowaniu po prostu dlatego, że potrzebujesz dużych zakresów i twoje urządzenie nie obsługuje go w oprogramowaniu lub sprzęcie, możesz łatwo zapomnieć o denormalizowanych wartościach, jeśli nie są one dla ciebie ważne.

Dodawanie Floats

To, co jest proste na papierze, nie zawsze jest takie proste na komputerze. Dodawanie wartości zmiennoprzecinkowych wymaga nieco myślenia. Prawdopodobnie nie jest to coś, co chciałbyś zaimplementować samodzielnie, chyba że Twój kompilator zostałby pozbawiony takiej rzeczy (co zazwyczaj nie jest prawdą), ale zawsze dobrze jest zrozumieć, co dzieje się pod maską. To pomoże ci zrozumieć, dlaczego pływaki są tak powolne. W tej sekcji jedynym problemem będą znormalizowane wartości. Możesz wyodrębnić drobne szczegóły implementacji, patrząc na nie pod względem czynników. Wszelkie pływaki można przedstawić jako a • 2b. Załóżmy na przykład, że a jest współczynnikiem mantysy o zakresie [1, 2], a b jest wykładnikiem czynnik. Tak więc, weź dwie liczby a2b i c2d. Bez utraty ogólności, załóżmy, że b jest większe niż d. Można wywnioskować następującą logikę:

$$\begin{aligned} a\cdot 2^b + c\cdot 2^d &= a\cdot 2^b + c\cdot 2^d\cdot(1) \\ &= a\cdot 2^b + c\cdot 2^d\cdot\left(2^b\cdot 2^{-b}\right) \\ &= a\cdot 2^b + \left(c\cdot 2^d\cdot 2^b\right)\cdot 2^{-b} \\ &= \left(a + c\cdot 2^{d-b}\right)2^b \end{aligned}$$

Idea równań jest raczej prosta. Powinieneś zawsze odrzucić najmniej znaczące bity. Tak więc, bierzesz największą liczbę pomiędzy tymi dwoma, i próbujesz zmienić drugą liczbę tak, że jej czynnik wykładniczy może być obliczony z innym numerem. Po ich obliczeniu równanie staje się łatwe do obliczenia. Jest to prosta kwestia przesunięcia wartości c na prawo, tak aby moc d odpowiadała wartości b. To właśnie tam tracisz precyzję, ponieważ zabijasz tutaj wszystkie najmniej znaczące bity. Po zakończeniu dodawania wystarczy normalizować wartości, aby część mantysy nie przekroczyła wartości 2. Jeśli tak się stanie, bit przenoszenia należy dodać do b automatycznie. To jest piękno tej reprezentacji. Zobaczmy, jak to by działało na przykładzie:

$$
\begin{aligned}
a &= 1.75 \\
b &= 10 \\
c &= 1.5 \\
d &= 5 \\
a2^b + c2^d &= 1.75 \cdot 2^{10} + 1.5 \cdot 2^5 \\
&= 1.75 \cdot 2^{10} + 2^{10}\left(1.5 \cdot 2^{-5}\right) \\
&= 2^{10}\left(1.75 + 0.046875\right) \\
&= 1.796875 \cdot 2^{10}
\end{aligned}
$$

Ten sam proces można zastosować również do liczb ujemnych (lub odejmowania, jeśli wolisz). W takim przypadku musisz po prostu sprawdzić, czy pożyczają bity. Jeśli współczynnik mantysy wynosi mniej niż 1, musisz zmniejszyć moc. To jest to samo, co robiłeś w szkole podstawowej, ale z nieco większym zaangażowaniem. Proces ten jest nieco chaotyczny, ponieważ musisz zająć się tymi wszystkimi przypadkami, co jest również powodem, dla którego zmienne punkty nie były zbyt popularne, gdy nie istniało coś takiego jak FPU.

Mnożenie floats

Jeśli zrozumiałeś proces dodawania, nie będziesz miał problemu ze zrozumieniem procesu mnożenia, ponieważ obowiązuje ta sama podstawowa idea. Jedyna różnica polega na tym, że musisz uciekać się do prostej, wykładniczej tożsamości. Poniżej przedstawiono, w jaki sposób można to osiągnąć:

$$
a \cdot 2^b \cdot b \cdot 2^d = \left(a \cdot b\right) \cdot 2^{b+d}
$$

W rzeczywistości jest to proste. Jedyne, na co musisz uważać, to mnożenie a i b. Z powodów, które zobaczyłeś w sekcji ustalonego punktu, będziesz musiał przesunąć w prawo wynik b • o liczbę bitów, które przenosi mantysa (która wynosi 23 dla pojedynczych pływaków). Na początku może wydawać się to bardzo dziwne, ale jest to właściwy sposób robienia rzeczy. Ponownie, musisz pamiętać, aby uwzględnić przelew, który możesz uzyskać z tego mnożenia. Ale po raz kolejny, jeśli wystąpi jakakolwiek przepełnienie, bity zostaną przesunięte w bity wykłanika czyli dokładnie tam, gdzie powinny iść bity przenoszenia. Poniższy przykład ilustruje sposób, w jaki pomnożysz dwa elementy float:

$$
\begin{aligned}
a &= 1.75 \\
b &= 10 \\
c &= 1.5 \\
d &= 5 \\
a2^b \cdot c2^d &= 1.75 \cdot 2^{10} \cdot 1.5 \cdot 2^5 \\
&= 1.75 \cdot 1.5 \cdot 2^{15} \\
&= 2.625 \cdot 2^{15} \\
&= 1.3125 \cdot 2^{16}
\end{aligned}
$$

Double float

Double float są bardzo podobne do pojedynczych pływaków, ale działają na 64 bitach zamiast na 32 bitach. Zasięg jest już całkiem dobry dla pojedynczych pływaków, dlatego organizacja IEEE zdecydowała się zwiększyć dokładność bardziej niż zasięg. Podwójny pływak ma proporcje 1:11:52. To trochę dla znaku, 11 dla wykładnika i 52 dla mantysy, jak pokazano na rysunku 19.3.

Znak	Wykładnik	Mantysa
1 bit	11 bits	52 bits

Z powodu tego współczynnika odchylenie wynosi połowę wartości maksymalnej. Tak więc, ponieważ 11 bitów jest zarezerwowanych dla wykładnika, odchylenie dla 64-bitowego zmiennopozycyjnego wynosi 1023. Równanie, które może reprezentować dowolną podwójną zmienną, jest podane przez:

$$\pm\left(1+\frac{mantissa}{2^{52}}\right)2^{1023-exponent}$$

Biorąc pod uwagę te szczegóły, można łatwo sprawdzić, czy liczby mają zakres [-1.1897 • 10^{4932}, 1.1897 • 10^{4932}], przy czym najmniejsza liczba to 3,3621 • 10^{-4932} i 3,6451 • 10^{-4951} dla wartości denormalizowanych. Denormalizowane wartości są zgodne z dokładnie tymi samymi regułami, które wymieniono wcześniej, z zaznaczonym wyjątkiem dodanych bitów dokładności. Zatem wszystkie denormalizowane wartości mają wykładnik równy 0, a ich równania można zapisać w następujący sposób:

$$\pm\left(\frac{mantissa}{2^{52}}\right)2^{-1022}$$

Warunek arytmetyczny

Odgałęzienia i warunki to dwa bardzo odrębne podmioty. Oddziały modyfikują położenie wskaźnika instrukcji, a warunki są funkcjami zwracającymi wartość. Czy wkładasz dużo wysiłku w to, w jaki sposób prezentujesz swoje warunki kompilatorowi? Czy wolałbyś powiedzieć, że szklanka jest w połowie pełna lub w połowie pusta? Zaufaj mi, proste pytania, takie jak te, mogą faktycznie wpłynąć na twój kod. Ważne jest, aby uważnie przyjrzeć się wyrażeniom warunkowym, ponieważ pierwsze wyrażenie, jakie przychodzi na myśl, nie zawsze jest najlepszym wyrażeniem dla maszyny, aby zrozumieć, co chcesz osiągnąć. Zanim zaczniesz analizować metody optymalizacji warunków, najpierw przyjrzyjmy się, co generuje kompilator po podaniu warunku. Prosty warunek jest prosty. Zamiast tego, przyjmijmy wyrażenie, które jest nieco bardziej skomplikowane - powiedzmy (A && B), gdzie A i B są predykatami. Wierzcie lub nie, jest to jednak przekształcone w dwie gałęzie, a nie jedną. Kompilator zwykle konwertuje górne wyrażenie na:

if (A)

    jeśliif (B)

DoIt ()

Tak właśnie powinno się to odbywać zgodnie ze standardem C. Ponieważ B jest funkcją, która zwraca wartość, nie powinno się jej wywoływać, chyba że A jest prawdziwe. Jest to oczywiście problematyczne, ponieważ wyrażenia mogą być bardzo rozbudowane. Ta książka nie jest książką optymalizacyjną, więc nie będę się zastanawiać, dlaczego tak się dzieje. Jednak chcesz pozbyć się gałęzi na komputerze najlepiej, jak to możliwe, ponieważ mogą ponosić kary do 20% z powodu algorytmu przewidywania rozgałęzień. Ponadto, jeśli pracujesz z wyrażeniem zawierającym wiele warunków, a następnie na podstawie poprzedniego kodu, najlepiej jest najpierw umieścić najbardziej prawdopodobny przypadek. Czasami jednak przypadki są równomiernie rozłożone i nie można powiedzieć, czy jeden warunek atomowy jest bardziej prawdopodobny niż reszta. W dalszej części tej sekcji przyjmij, że warunki są mniej lub bardziej prawdopodobne. Z tych powodów wyrażenia w tej sekcji zostaną zoptymalizowane w dwóch etapach. Pierwsza polega na usunięciu wielu gałęzi z wyrażenia, a druga na optymalizacji samego wyrażenia w celu zminimalizowania liczby kontroli, które należy zastosować.

Usunięcie przez Obliczenia

Większość programistów pisze kod tak, jak o nim myślą. Nie zawsze jest to jednak najlepszy sposób robienia rzeczy, ponieważ ludzki sposób myślenia o rozwiązaniu niekoniecznie jest najlepszym sposobem postępowania dla maszyny. Ta sekcja zawiera wiele interesujących formuł, ale jeśli pamiętasz tylko jedną rzecz po jej przeczytaniu, pomyśl o problemie z punktu widzenia maszyny. Pozwól mi zilustrować za pomocą przykładów. Bardzo popularnym prymitywem jest obliczenie minimum i maksimum dwóch wartości (np. Liczb całkowitych). Co jest pierwszą rzeczą, która przychodzi ci na myśl przez min? Cokolwiek to jest, jest to prawdopodobnie jedno z tych dwóch rozwiązań:

```
if  (A <B)

X = A;

else

X = B; // Rozwiązanie 1

X = A <B ? O: B; // Rozwiązanie 2
```

Czy się myliłem? Obie te funkcje są w rzeczywistości takie same i kończą na jednym odgałęzieniu warunkowym. To jest typowy wynik kompilatora, jeśli uruchomisz funkcję abs z biblioteki matematycznej. Jeśli myślisz o funkcjach min SIMD, możesz dać sobie jeden punkt za zabicie gałęzi, ale ta trasa nie jest zbyt dobra jeśli chodzi o prędkość, jeśli nie planujesz równoległych obliczeń. Więc jeśli nie jest to jedno z powyższych rozwiązań, co to może być? Co powiesz na następujące:

$$\min\{a,b\} = \frac{a+b-|a-b|}{2}$$

To oczywiście nie ma żadnych oddziałów, matematycznie rzecz biorąc, i jeśli o tym pomyśleć, to działa. Jeśli tego nie widzisz, być może ponowne przetworzenie równania jako takiego będzie miało więcej sensu:

$$\min\{a,b\} = \frac{b+\left(a-|a-b|\right)}{2}$$

Bez utraty ogólności przypuśćmy, że a jest największą liczbą. Grupując równanie tak, jak zrobiono to wcześniej, a ponieważ jest to największa liczba, otrzymujesz następujący wynik:

$$\begin{aligned}\min\{a,b\} &= \frac{b+\left(a-(a-b)\right)}{2}\\ &= \frac{b+b}{2}\\ &= b\end{aligned}$$

Jeśli, z drugiej strony, b jest największą liczbą, po prostu zmień nawiasy i kolejność zmiennych w wartości bezwzględnej, a wynikiem będzie a. Możesz także zmienić znak przed wartością bezwzględną, aby uzyskać maksimum zamiast minimum. Interesujący sposób na to spojrzeć, prawda? Jeśli cię złapię niespodziewanie, nie martw się; Mam dla ciebie jeszcze jeden. Jak obliczyć wartość bezwzględną liczby całkowitej? Ten wymaga nieco głębszej znajomości PC. Spróbuj wykonać następujące czynności:

$$x = \left((x \gg 31) \wedge x\right) - (x \gg 31)$$

Trudno uwierzyć, że ta funkcja jest szybsza niż warunek, ale wierz mi, to wymaga około 70 procent tak długo, jak inne funkcje podczas testowania. Kluczem do zrozumienia tego jest fakt, że prawo do zmiany liczby całkowitych ze znakiem zastępuje najwyższe bity wartością 1. Zatem, jeśli liczba jest ujemna, otrzymujesz:

$$\begin{aligned}x &= \left(\text{0xFFFFFFFF} \wedge x\right) - \text{0xFFFFFFFF}\\ &= (-x) - 1\end{aligned}$$

Wręcz przeciwnie, jeśli x jest dodatnie, otrzymuje się następujące, które absolutnie nic nie robi:

$$\begin{aligned}x &= (0 \wedge x) - 0\\ &= x\end{aligned}$$

Połącz minimalną funkcję z tym jednak, a otrzymasz w przybliżeniu 25-procentowy wzrost prędkości w stosunku do oddziału dla zbioru liczb losowych. Nieźle. Nie dlatego, że znalazłeś równanie dla ogólnego przypadku, w którym nie musisz myśleć o konkretnym przypadku. Wiele razy możesz zoptymalizować swój kod przez ponowne przemyślenie problemu z konkretnymi rzeczami, które znasz o ogólnym rozwiązaniu. Oto kolejne wyzwanie dla ciebie: Oblicz minimum dwie liczby całkowite bez znaku.

(Podpowiedź: będziesz potrzebował instrukcji montażu, aby to osiągnąć.) Oczywiście, możesz użyć powyższego kodu, aby to osiągnąć, ale spróbujmy innego podejścia. Załóżmy, że eax i ebx zawierają dwie liczby i chcesz znaleźć minimum, które jest ostatecznie przechowywane w eax. Kod jest następujący:

sub ebx, eax

sbb ecx, ecx

i ecx, ebx

add eax, ecx

Uwaga

Fakt, że jest to napisane w kodzie asemblera, nie powinien mieć znaczenia. Kompilator wykonuje doskonałą pracę optymalizując kod C w tak małych porcjach; z tego powodu kod asemblera jest ogólnie porównywalny z kodem C dla bardzo małych funkcji (chyba że można użyć niektórych specjalnych funkcji, takich jak w tym przypadku). Niektóre testy porównawcze z poprzedniego kodu pokazują, że jest on w rzeczywistości około 30 procent szybszy niż pierwsza wersja. Jedną rzeczą, na którą trzeba uważać, jest to, że usunięcie gałęzi nie oznacza usunięcia warunku. Załóżmy na przykład, że masz funkcję, która zwraca zdarzenie - powiedzmy, zdarzenie, które mówi, że kliknięto przycisk myszy. Dalej załóżmy, że musisz oznaczyć zmienną w swoim kodzie, aby powiadomić główną pętlę o tym fakcie. Kiedy główna pętla zauważy, że flaga została ustawiona, może użyć tych informacji do wykonania określonej akcji. Po wydaniu ta sama funkcja jest wywoływana, ale z innym zdarzeniem (na przykład nie kliknięto myszką). Tradycyjnym sposobem radzenia sobie z tym jest napisanie czegoś w tym stylu:

If (Event == MOUSE_CLICKED)

Flaga = 1;

else

Flaga = 0;

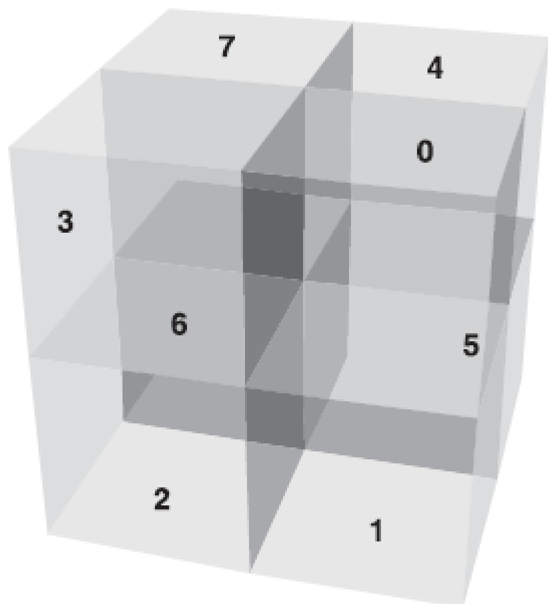
Zaawansowaną metodą uzyskania tego samego jest napisanie:

Flag = (Event == MOUSE_CLICKED);

To nie przekształca się w gałąź, ponieważ kompilator używa do tego wewnętrznych znaczników warunków. W takim przypadku zapisz oddział. Inną podobną sztuczką jest użycie funkcji, która ustawia daną zmienną na 1, jeśli liczba jest dodatnia lub? 1, jeśli jest ujemna. Rozwiązanie? Bardzo podobne do tego, co widzieliście do tej pory:

Znak = ((x> 0) << 1) - 1

W tym przypadku należy pamiętać, że wyrażenie warunkowe zwraca 0 lub 1, w zależności od tego, czy jest oznaczone jako prawdziwe czy fałszywe. Możesz użyć tego faktu na swoją korzyść, aby obliczyć wynik zmiennej na podstawie wyrażenia. Możesz także użyć tabel wyszukiwania lub, co ciekawsze, wartości odnośników, aby uzyskać ostateczną wartość, której szukasz. Jednym z takich przykładów jest funkcja określania oktanta (w ośmiu), w którym znajduje się dany wierzchołek (x, y, z). Załóżmy, że z jakiegoś praktycznego powodu jesteś zmuszony opisywać oktanty zgodnie z ruchem wskazówek zegara, zaczynając od oktanta 0 będącego prawym górnym oktanem. Ponadto przypuśćmy, że przednia połowa zaczyna się od 0, podczas gdy najdalsza połowa używa tej samej reguły, ale zaczyna się od 4, jak pokazano na rysunku 19.4.



Sztuką w tym momencie jest nadanie unikalnego indeksu każdemu oktantowi. Możesz łatwo określić różnicę między z <0 i z> 0 przez obliczenie (z <0), które daje 1 lub 0. Możesz łączyć tę samą logikę dla x i y, pod warunkiem, że nie tracą informacji zebranych na z . Jeśli przydzielisz im oddzielne bity, powinieneś być dobry. Załóżmy, że pierwszy bit jest przypisany do x, drugi do y, a trzeci do z. Możesz przesunąć wynik warunku w lewo, aby oddzielić bity. Z trzema bitami wygenerujesz osiem możliwości, a tym samym jednoznacznie zidentyfikujesz osiem oktantów. To, co obliczałeś do tej pory, jest jednak tylko wskaźnikiem. Indeks nie jest zgodny z zamówieniem, które otrzymałeś od swoich oktantów. W rezultacie musisz odwzorować każdą z nich na inną wartość pomiędzy 0 a 7. Taką wartość można przedstawić na trzech bitach; tak się składa, że w 32-bitowym numerze można zapisać do 10 wartości 3-bitowych lub do ośmiu 4-bitowych wartości. Odpowiedź powinna być dość oczywista w tym momencie: Możesz zbudować 32-bitową wartość podzieloną na osiem elementów z czterech bitów, dla których indeks zrównoważy elementy, i dla których przesunięcie zostanie odwzorowane na odpowiadający oktant. Ponieważ każdy element jest traktowany jako element 4-bitowy, przesunięcie będzie musiało zostać przesunięte w lewo o 4, aby poprawnie zaadresować bity. Pozostaje tylko zamaskować wszystko tak, aby uzyskać tylko trzy ostatnie bity. Poniższy kod ilustruje to:

```
unsigned long Index((X < 0) << 2);
Index |= (Y < 0) << 3;
Index |= (Z < 0) << 4;
Octant = (0x65742130 >> Index) & 7;
```

Jak to działa? Cóż, wartość indeksu zasadniczo izoluje nibble, który jest wymagany. Możliwe wartości indeksu to {0, 4, 8, 12, 16, 20, 24, 28}. Tak więc "wartość magiczną" przesuwa się o jedną z tych wartości. Sztuką w tym momencie jest po prostu dopasowanie sprawy do indeksu i dopasowanie tych dwóch do wartości ósemkowej. Każda wartość jest reprezentowana na 4 bitach, dlatego indeks przeskakuje z przyrostem 4 bitów. Na przykład, jeśli miałeś przypadek <-2, 2, 2>, przeszedłby tylko pierwszy test. W rezultacie otrzymasz indeks 4, druga wartość w zestawie możliwych wartości. To mapuje na oktant 3, jeśli obliczysz go za pomocą poprzedniego równania.

A potem nie było żadnych

Tytuł tej sekcji wynika ze słynnej książki autorstwa Agathy Christie - interesującej, intrygującej, nie do końca realistycznej, ale wciąż zabawnej. Przedstawia historię gości, którzy zostali zaproszeni do domu,

aby zostać zabitym, jeden po drugim. W całej historii powtarza się refren "A potem było x"; zgodnie z oczekiwaniami, książka kończy się na: "A potem ich nie było". Podobnie, celem tej sekcji jest zabicie problematycznych operatorów jeden po drugim (być może nie tak radykalnie, jak to zrobiono w książce Agaty, ale jestem pewien, że twoja wyobraźnia może przynieść wiele scenariuszy).

Logiczny operator AND

Zacznijmy od logicznego operatora AND, oznaczonego &&. Logiczny operator AND pobiera dwie zmienne. Załóżmy na przykład, że masz dwa predykaty, A i B. Twój stan (A && B) można również przekonwertować na jego odpowiednik bitowy (A i B). Oznacza to, że oba predykaty zostaną obliczone, ale bitowe I zagwarantuje usunięcie jednej dodatkowej gałęzi. Zamiast konwersji do przykładowego kodu, który widziałeś wcześniej, po prostu otrzymujesz jeden dłuższy warunek, po którym następuje akcja. Bit-mądry AND działa na zasadzie bit-na-bit. Aby to zadziałało, musisz upewnić się, że przecięcie (AND) wszystkich możliwych wartości niezerowych będzie niezerowe, lub nie ma dopasowania jeden do jednego. Jeśli zawsze używasz operatorów, którzy konwertują twoje warunki na 0 lub 1, to nie jest to problem. Ale jeśli jesteś podobny do mnie i wolisz używać krótszego formularz do przetestowania elementów niezerowych, a następnie bitowy AND i nie będzie działał poprawnie, chyba że jest tu podane:

if (a && b) DoIt ();

Weźmy na przykład a = 2 i b = 1. I je razem, a otrzymasz 0, nawet jeśli oba są niezerowe. Zamiast tego, należy zmienić instrukcję na coś podobnego do następującego, które konwertuje wartości na 0 lub 1:

if ((a! = 0) & (b! = 0))

Operator OR

Teraz zwróć uwagę na operację pierwotną OR. Kompilator zwykle konwertuje OR stan podobny do tego, w jaki sposób przekształcił warunek ORAZ. W istocie, dla dwóch predykatów, A i B, wyrażenie (A || B) daje:

Jeśli)

goto Zrobione;

Jeśli (! B)

goto Zrobione;

Zrób to();

Gotowe:

Ponownie przedstawiono dwie struktury rozgałęzień i, podobnie jak w przypadku AND, można wyrazić (A || B) do równoważnej wersji bitowej (A | B). Uważaj na jedną pułapkę, która może się wynurzyć, kiedy to robisz. Operator bitowy AND ma większy priorytet przed operatorem XOR (^). Dlatego upewnij się, że twoje nawiasy są poprawnie skonfigurowane, jeśli używasz XOR w swoim kodzie. Poza tym małym niedociągnięciem operator OR nie przedstawia tego samego problemu, co operator AND, ponieważ bity są addytywne.

Wyraźnie się wyrażaj

Po przekonwertowaniu wyrażenia na wyrażenie w jednym oddziale można je przesłać do następnego kroku, aby zmniejszyć liczbę weryfikacji. Można to osiągnąć na dwa sposoby:

■ Pierwszą metodą jest stworzenie tabeli prawdy i znalezienie minimalnego wyrażenia reprezentuje bity, które chcesz aktywować w wyrażeniu.

■ Druga metoda wykorzystuje pojęcia algebraiczne poprzez manipulowanie wyrażeniem z zestawem tożsamości, dopóki wyrażenie nie może być już zmniejszone.

Tabele prawdy na ratunek

Pomysł polega na zadaniu sobie pytania, jaki jest wynik wszystkich możliwych danych wejściowych. Aby to zrobić, musisz wyraźnie wyodrębnić swoje parametry, wyliczyć wszystkie możliwe wartości tych parametrów w kategoriach bitów i umieścić je w tabeli. Załóżmy na przykład, że masz trzy predykaty wejściowe A..C, dla których wykonywane są akcje D..F zgodnie z opisem w tabeli 19.5. Wartość 1 oznacza, że wynik jest brany, a wartość 0 oznacza, że wyniku nie należy podejmować.

A	B	C	D	E	F
0	0	0	-	0	1
0	0	1	0	0	0
0	1	0	-	0	1
0	1	1	0	0	0
1	0	0	-	1	0
1	0	1	0	0	0
1	1	0	-	1	1
1	1	1	0	0	1

Dla niektórych może się to wydawać nudne, ale inni, którzy są bardzo wizualni, wolą mieć takie ustawienia. W tej tabeli znajdziesz wszystkie możliwe predykaty i wszystkie możliwe wyniki tych predykatów. (Może być tylko jeden, jeśli robisz ten proces tylko na jednej gałęzi.) Dla predykatów umieszczasz 0, gdy twój warunek się nie powiedzie, a 1 oznacza, że warunek musi przejść, a myślnik nie reprezentuje żadnych preferencji. Teraz twoim zadaniem jest znaleźć wzór w tej tabeli, który minimalnie opisuje wynik. Zacznijmy od wyniku D. Dla każdej parzystej wartości D nie ma preferencji; dla każdej wartości nieparzystej jest ustawiona na 0. Odtąd D zawsze musi wynosić 0. Oznacza to, że D nie jest gałęzią. W rzeczywistości ten kod powinien zostać natychmiast zabity, ponieważ nigdy nie musi osiągnąć tego rezultatu. W następnej kolumnie, E, sprawy są trochę trudniejsze. Możesz zauważyć, że A musi być prawdą, aby rozważyć choć odrobinę. Ponadto wydaje się, że C musi być fałszywe. Stąd:

$$E = (A \& !C)$$

Dla ostatniej kolumny, F, w najgorszym przypadku, będziesz miał cztery porównania, z których każdy ma trzy kontrole na każdym predykacie dla łącznej liczby 12 oddziałów. Starajmy się zrobić lepiej. Zauważ, że kiedy A = C, pokrywasz 75 procent przypadków i otrzymujesz jedną błędną odpowiedź. Brakujący przypadek, którego nie uwzględniłeś, zadziałałby, gdybyś mógł odwrócić B i C. To nie tylko zajmowałoby się złą wartością, ale zawierałoby również brakującą wartość. Obserwujesz, że dzieje się tak tylko wtedy, gdy ustawiono A. Rozumowanie jest następujące: Jeśli A jest ustawione, zamień B i C. Ostatnim testem jest to, że jeśli A == C, wynik jest prawdziwy. Ostatnim pytaniem jest: jak uzyskać inwersję? Sztuką jest ponowne rozważenie tego samego rodzaju pytań, które poprzednio widzieliście w tym rozdziale. Spróbuj trochę to przemyśleć. Poddać się? Równanie, które poprawnie rozwiązuje wynik dla F, jest następujące:

$$F = \left(A == \left(\left(\left(C + B + B\right) \gg A\right) | 1\right)\right)$$

Ten przykład to pięć operatorów zamiast 16. Jest to rozwiązanie bardziej tajemnicze, ale o wiele szczuplejsze niż pełne wyrażenie, które byłoby następujące:

$$F = (!A \& !B \& !C) | (!A \& B \& !C) | (A \& B \& !C) | (A \& B \& C)$$

W najgorszym przypadku zawsze powinieneś mieć mniej niż połowę wszystkich możliwości. Jeśli podejmiesz decyzję, że można uzyskać więcej niż połowę, weź uzupełnienie (odwrotnie) tego, a pozostanie mniej niż połowa przypadków. Ta metoda jest niezwykle przydatna, abyś mógł myśleć nieszablonowo. Nie wiąże cię żadnymi ścisłymi zasadami algebraicznymi. Ogólnie rzecz biorąc, ta metoda może przynieść ulepszenia o wiele większe niż prosta algebraiczna manipulacja przy użyciu zestawu reguł. Kluczem do pokonania następnej metody jest myślenie. Pomyśl o związku każdego pojedynczego operatora, zastanów się, jak zmieniają bity w zmiennej i pomyśl o operatorze logicznym (>, <, ==, ...), ale także operatory bitowe (<<, >>, ^, |, &&,. ...). W rzeczywistości wyrażenia algebraiczne mogą pomóc w zrozumieniu zachowania między tymi funkcjami, więc przejdźmy do tego tematu.

Tożsamości algebraiczne

Dlaczego tak wielu uczniów szkół średnich twierdzi, że algebra jest ich najgorszym koszmarem? Moim zdaniem jest tak dlatego, że nigdy nie brali udziału w zajęciach z analizy, ponieważ byłby to ich koszmar. Na szczęście ta sekcja trzyma się analizy algebry i spódnic. Algebra składa się z definicji i aksjomatów. Ta ostatnia jest formą twierdzenia, której nie można udowodnić. Wspólnie do budowania twierdzeń stosowane są definicje i aksjomaty. Ta sekcja ma dostarczyć zestaw twierdzeń, a dokładniej zestaw równości, które można wykorzystać do uproszczenia nierówności warunkowych. Niestety, lista nie jest wyczerpująca, a niektóre mogą wydawać się bardzo proste, ale są tu napisane na wypadek, gdybyś zapomniał. Wielkie litery są używane do reprezentowania predykatów (warunek atomowy), podczas gdy pozostałe litery tworzą elementy atomowe predykatu. Używam notacji C / C ++ dla AND, OR, NOT i tak dalej. Oznaczam dwa typy równości matematycznych:

- Implikacja, która działa tylko po jednej stronie. Na przykład załóżmy, że A ➔ B, A oznacza "jest kompilatorem", a B oznacza "generuje nieoptymalny kod." Mógłbyś powiedzmy, że jeśli A jest kompilatorem, generuje nieoptymalny kod. Nie można jednak wywnioskować, że przy nieoptymalnym kodzie generator był kompilatorem. Ludzie też piszą niezoptymalny kod.

- Operator • jest równością prezentowaną przez dwustronną implikację.

Możesz również dowolnie zastąpić dowolny predykat A znakiem! A, aby uzyskać nowe tożsamości. Pod warunkiem, że robisz to wszędzie w wyrażeniu, relacja będzie nadal ważna. W rzeczywistości jest to konkretny wynik bardziej ogólnej reguły: dowolny predykat A może również zostać zastąpiony bardziej złożonym wyrażeniem. Na przykład można zamienić predykat A na (B | (C & D)). Inna ważna uwaga jest taka, że czasami istnieją specjalne relacje między predykatami. Na przykład, jeśli predykaty A i B są odpowiednio (a> 3) i (a> 6), wówczas można zauważyć, że B ➔ A, ponieważ (a> 6) ➔ (a> 3). Odwrotność jednak nie jest prawdą. Innymi słowy, pierwszy predykat dostarcza informacji o drugim. Co więcej, możesz zoptymalizować wyrażenia jeszcze bardziej, jeśli weźmiesz pod uwagę ten fakt. Tabela 19.7 pokazuje kilka tożsamości w tym kontekście.

```
A → B <–> !A | B
(A → B) & (A → C) <–> A → (B & C)
(A → C) & (B → C) <–> (A | B) → C
(A → B) | (A → C) <–> A → (B | C)
((A → C) | (B → C)) → ((A & B) → C)
```

Lista ta nie jest wyczerpująca, ale dobrą rzeczą jest to, że jeśli wierzysz, że znalazłeś związek pomiędzy zestawem predykatów, możesz użyć tabeli prawdy, aby zweryfikować swoje roszczenie. Niefortunną rzeczą jest to, że ta metoda ogranicza się do myślenia czysto w kategoriach algebraicznych, a nie na zewnątrz (pole to zasady algebraiczne). Jeśli użyjesz kombinacji obu technik przedstawionych do tej pory w połączeniu z odrobiną myślenia, z pewnością znajdziesz bardzo dobre rozwiązanie swojego problemu.

Porównania float

Podczas pisania kodu przy użyciu języka wysokiego poziomu większość programistów nie wkłada wiele zastanowili się nad implikacjami kodu, który piszą. Jak pokazała poprzednia sekcja, niektóre pozornie nieszkodliwe operacje mogą w trakcie montażu przekształcić się w coś znacznie bardziej zaangażowanego. Dobry programista asemblera może zazwyczaj generować lepszy kod języka wyższego poziomu - nie dlatego, że może on konwertować wolne fragmenty kodu na zespół, ale ponieważ ogólnie mają lepszą znajomość maszyny i większy szacunek i uznanie dla szybkości. Jak się okazuje, porównanie liczb całkowitych jest szybsze niż porównanie zmiennoprzecinkowe - nie tylko pod względem czystej liczby cykli, ale także pod względem procesu. Oświadczenie takie jak if (f> 1.0f) zazwyczaj zostanie przekształcone w coś równoważnego następującym:

```
float Memory(1.0f);
Compare(f, Memory);
EAX = _controlfp(0, 0)
If (EAX & 0x41)
...
```

Po pierwsze nie można porównać wartości zmiennoprzecinkowej z bezpośrednią wartością. Po drugie, architektura wymaga pobrania słowa kontrolnego instrukcji zmiennoprzecinkowej, zapisania tego w podstawowych rejestrach (wewnętrzne zmienne PC), a następnie wykonania drugiego testu na słowie kontrolnym w celu ustalenia wyniku zmiennoprzecinkowego porównanie. To wiele operacji do zrobienia. Jeśli jednak traktujesz wartość zmiennoprzecinkową jako surowy adres w pamięci, możesz zoptymalizować porównania, używając po prostu porównań całkowitych. Pierwszym i najbardziej oczywistym testem, który możesz przeprowadzić, jest czysty test równości. Sprawdzenie, czy zmiennoprzecinkowy jest równy innemu zmiennoprzecinkowemu, jest tym samym pytaniem, czy bity jednego zmiennoprzecinkowego są dokładnie równe bitom drugiego zmiennoprzecinkowego. Jako taki, aby przetestować ścisłą równość, możesz po prostu typować na długo i wykonać porównanie bezpośrednio na tym poziomie, jak ilustruje poniższy kod:

```
if (* (unsigned long *) & f == 0x3F800000) // 1.0f =

0x3F800000
```

Dotyczy to również ścisłej nierówności. Możesz pomyśleć, że możesz osiągnąć coś podobnego z większymi / mniejszymi operacjami, a prawda jest taka, że jest to możliwe. Weźmy na przykład 0, co jest szczególnym przypadkiem, ponieważ jest punktem zwrotnym znaku. Dzięki temu możesz sprawdzić, czy wartość zmiennoprzecinkowa jest większa, równa lub mniejsza niż patrząc na znak. Ale

co dzieje się z przypadkami, które są znacznie większe i mniejsze lub równe? Cóż, wiesz, że 0 w float wynosi 0 w int i że każda ujemna liczba będzie miała najbardziej znaczący bit włączony. W związku z tym, jeśli dokonasz porównania z wartością 0, osiągniesz to samo, co porównanie zmiennoprzecinkowe z 0.

Poniższy kod pokazuje to:

if (* (unsigned long *) & f & 0x80000000) // Testy dla f <0.0f

if (! (* (unsigned long *) & f & 0x80000000)) // Testy dla f> = 0.0f

if (* (long *) & f> 0) // Testy dla f> 0.0f

if (* (long *) & f <= 0) // Testy dla f <= 0.0f

Co się stanie, jeśli stała, którą chcesz porównać, jest niezerowa? Spójrzmy najpierw na dodatnią stałą wartość zmiennoprzecinkową i bez utraty ogólności wybierzmy 1,5f. Wiesz już, że 1.5f oznacza 0x3FC00000. Wiesz z wcześniejszej sekcji, że każda dodatnia wartość zmiennoprzecinkowa jest reprezentowana jako (1 + 2-mantysa) * 2exponent-127. Najpierw sprawdź, co się dzieje, gdy mantysa jest inna. Mantysa tej liczby w bitowej reprezentacji to 0x400000. Można zauważyć, że mniejsza lub większa mantysa daje mniejszą lub większą liczbę całkowitą, więc dla mantysy porównanie dla pamięci surowej jest dokładnie takie samo dla liczb całkowitych, jak dla pływających. Z drugiej strony, jeśli wykładnik jest inny, można również sprawdzić, czy dokładnie taka sama logika występuje. Wykładnik jest opisany jako wykładnik? 127, a mniejszy lub większy wykładnik podaje mniejszą lub większą wartość całkowitą. Jeśli połączysz oba, to nadal działa poprawnie, ponieważ wykładnik leży przed mantysa, co daje mu przewagę w porównaniu. Jest to dość interesujące, ponieważ jeśli rzucisz blok pamięci typu float na liczbę całkowitą ze znakiem, możesz wtedy uzyskać porównania z wartościami zmiennoprzecinkowymi, po prostu przeliczając stałą na jej odpowiednik całkowity i wykonując porównania w czystych liczbach całkowitych. Poniższy kod demonstruje to:

if (* (long *) & f <0x3fC00000) // Testy dla f <1.5f

if (* (long *) & f> = 0x3fC00000) // Testy dla f> = 1.5f

if (* (long *) & f> 0x3fC00000) // Testy dla f> 1.5f

if (* (long *) & f <= 0x3fC00000) // Testy dla f <= 1.5f

A co z ujemnymi stałymi? Twoim głównym problemem z ujemnymi stałymi jest znak. Jeśli wybierzesz liczbę ujemną, masz numer w postaci 0x8. Traktujesz mantysę i wykładnik jako jedność, ponieważ już widziałeś, że są one ustawione poprawnie dla twoich celów i zaczną od zbadania ujemnych wartości. Obie wartości mają postać 0x8. Co się dzieje, jeśli połączona mantysa i wykładnik są większe? Jeśli dokonasz konwersji z pamięci na zmienną, zauważysz, że oznacza to, że wartości zmiennoprzecinkowe są mniejsze. Podobnie, dla porównania niższego, masz większą liczbę. Jeśli twoja zmienna jest dodatnia, możesz również sprawdzić, czy ta obserwacja się utrzymuje. Dlatego w przypadku stałych o wartościach ujemnych wystarczy odwrócić operand, który chcesz przetestować, a wartości równe będą wartościami całkowitymi. Pamiętaj, że odlewanie do podpisu jest tutaj całkowicie bezużyteczne. W rzeczywistości jest to całkowicie fałszywe po przekonwertowaniu na liczbę całkowitą ze znakiem. Poniższy kod pokazuje, jak porównania działają dla -1,5f:

if (* (unsigned long *) & f> 0xBfC00000) // Testy dla f <1.5f

if (* (unsigned long *) & f <= 0xBfC00000) // Testy dla f> = 1.5f

```
if (* (unsigned long *) & f <0xBfC00000) // Testy dla f> 1.5f

if (* (unsigned long *) & f> = 0xBfC00000) // Testy dla f <= 1.5f
```