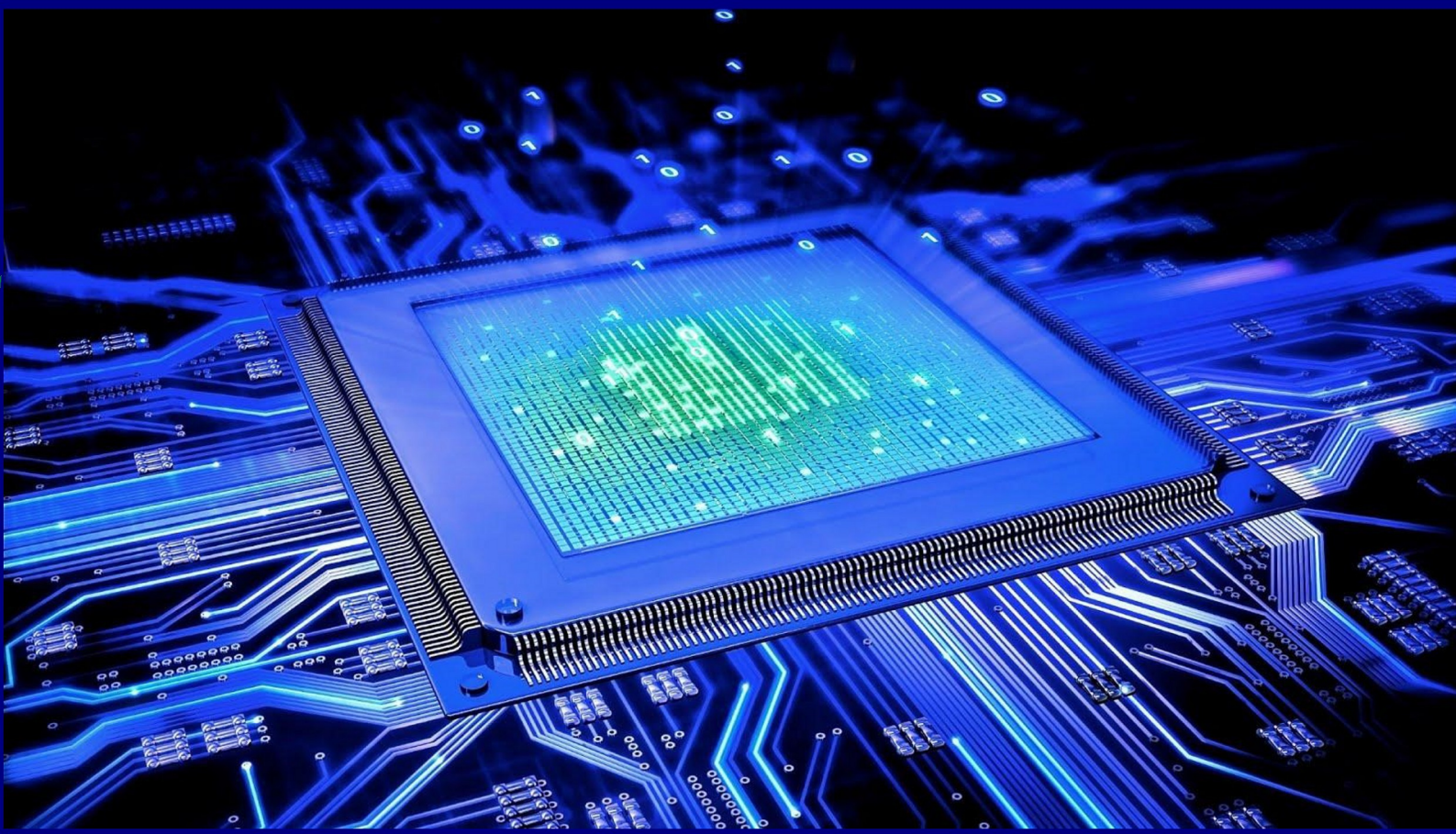
Linguagem Assembly para i386 e x86-64



Frederico Lamberti Pissarra

Linguagem Assembly para i386 e x86-64

Versão 0.8.5

3 de março de 2017

Título: Linguagem Assembly para i386 e x86-64

Autor: Frederico Lamberti Pissarra

Ano de publicação: 2017

Este material é protegido pela licença GFDL 1.3:

É importante, no entanto, advertir ao leitor que este material encontra-se em processo de produção. Trata-se, então, de um rascunho e não reflete o material finalizado.

Índice

Introdução

Em todo material técnico relacionado com a linguagem Assembly costumo escrever a mesma ladainha, massageando o meu ego. O leitor que me perdoe, mas o que vai ai embaixo não será exceção...

Sou o autor do, hoje nem tão famoso assim, “Curso de Assembly da RBT”. Escrevi 26 capítulos sobre o assunto, em 1994, e fiz certo sucesso na grande rede. Infelizmente o curso está meio caduco, obsoleto, e merece uns retoques e adições.

Os capítulos que seguem são a minha tentativa de explicar melhor e mais profundamente o que é assembly, bem como fornecer algum material de referência e um pouco de como funciona aquele chip da placa-mãe que você lambuza de pasta térmica e coloca um grande cooler em cima.

Outro aviso importante é sobre o foco deste texto. O leitor não encontrará informações importantes aqui que permitam o desenvolvimento, por exemplo, de um sistema operacional. Não falarei quase nada sobre registradores de controle, inicialização do processador, multi-threading (e multi-processing), interfaces com a BIOS, etc. O que mostro aqui é a linguagem assembly e seu uso no *userspace*.

Dicas de como ler este livro

É tradicional, em textos desse tipo, que a introdução contenha material essencial para o bom entendimento das partes mais “hardcore”. Coisas como base numéricas e um pouquinho de conceitos de eletrônica digital... Vamos a elas logo no próximo capítulo, mas antes, precisamos ter certeza que falamos a mesma língua e usamos as mesmas ferramentas do ofício...

Tentei seguir alguns padrões para facilitar sua leitura:

- **Negritos** são usados para ressaltar alguma informação que pode passar desapercebida pelo leitor, bem como a introdução de algum termo técnico que, mais tarde, aparecerá numa formatação mais “normal”;
- Aspas também são usadas como recurso de chamada de atenção, ou para tornar algo mais (no caso de uma piada) ou menos (no caso de eu querer dizer exatamente o que a palavra significa) ambíguo. Usar aspas é uma mania minha... assim como usar reticências...;
- Palavras em *itálico* são usadas em termos estrangeiros, na maioria das vezes;
- Códigos fonte, palavras reservadas de uma linguagem e referências a um código fonte no texto explicativo usam fonte `mono espaçada`;
- Às vezes textos aparecerão dentro de uma caixa cinzenta:

> Como este aqui, por exemplo!

Ela é outro desses recursos usados para chamar sua atenção para uma explicação importante.

Existem outras táticas estilísticas para ajudar na leitura: Todos os códigos fonte, nomes de símbolos e referências a códigos fonte são escritas em caracteres **minúsculos**. Faço isso porque tanto a linguagem C quanto o NASM são *case sensitive*. Isso significa que nomes de variáveis como `abc`, `abC`, `aBc` ou `Abc` **não** são a mesma variável. O outro motivo é pessoal. Sou um sujeito sensível e sempre que leio códigos escritos em letras maiúsculas acho que estão gritando comigo e fico muito nervoso (especialmente se vejo uma burrada no código!)...

Nomes de registradores poderão aparecer tanto em letras minúsculas quanto maiúsculas. O último caso serve como destaque estilístico, para não tornar as sentenças confusas ao leitor.

Sobre o tamanho de blocos

Você observará que ao informar tamanhos de blocos usarei a abreviação “KiB” ao invés da mais tradicional “kB”. Existe uma diferença... O Sistema Internacional de unidades, mantido pela ISO, especifica que o prefixo “k” significa multiplicar a unidade por 10^3. Acontece que 1 “quilobyte” não é exatamente igual a 1000 bytes. Existe um prefixo específico para esse tipo de grandeza, cuja base numérica é baseada em 2: O “kibibyte”. O prefixo “Ki” significa multiplicar por 1024 ou 2^{10}. Da mesma forma que o MiB (Mebibyte) significa multiplicar por 2^{20} e o GiB (Gibibyte) por 2^{30}. Note que esse fator multiplicador sempre é expresso em letra maiuscula seguida de um 'i', minúsculo. Isso é intencional: No SI o prefixo 'k' é sempre minúsculo (provavelmente para diferenciar da unidade de temperatura Kelvin).

Da mesma maneira, a unidade abreviada “B” significa “byte” e a letra minúscula “b” significa “bit”.

Isso não é invenção minha. Trata-se das especificações IEEE 1541 de 2002 e IEC 80000-13 de 2008.

Ferramentas do ofício

Para trabalharmos com assembly usaremos, pelo menos, 2 ferramentas: Um compilador C e um *assembler*[1]. O compilador C usado é o GCC (na versão 4.8 ou superior). Este é o meu compilador preferido, por ser *free software* e, ainda por cima, estar disponível para qualquer plataforma. Além disso, ele é um dos melhores compiladores C/C++ disponíveis, não deixando nada a desejar ao *Intel C/C++ Compiler* e é diversas ordens de grandeza superior ao compilador C/C++ da Microsoft, do *Visual Studio*.

Quanto ao assembler, uso o *Netwide Assembler* (NASM, na versão 2.11 ou superior). Ele implementa uma notação de mnemônicos semelhante a da Intel – nada impede que você use outro, como o GAS (GNU Assembler), mas esteja avisado de que a linguagem assembly usada nesse compilador é diferente da que você verá aqui. Falarei sobre os “sabores” de assembly logo. Não falarei coisa alguma sobre o MASM (Microsoft Macro Assembler), TASM (Turbo Assembler, da antiga Borland) ou outros assemblers mais antigos ou exotéricos como o YASM e o FASM.

Dito isto, a sintaxe do GAS é útil quando estamos escrevendo blocos de assembly *inline* no GCC. Não significa que não possamos usar o sabor Intel, mas o sabor AT&T (o do GAS) é tradicionalmente usado nas ferramentas do projeto GNU.

Por que diabos precisamos de um compilador C?

É perfeitamente possível desenvolver aplicações inteiras em assembly, só não é nada prático! O compilador C será usado aqui como base de comparação e como hospedeiro (*host*) para testes de rotinas desenvolvidas em assembly. Com relação à “base de comparação”, o compilador C pode ser usado para obtermos o código que o compilador cria para uma função, via opção de compilação “-S”. De fato, para obter códigos em assembly de suas rotinas, sugiro a seguinte linha de comando:

```
$ gcc -O3 -S -masm=intel -mtune=native -fverbose-asm test.c
```

Isso ai criará um arquivo chamado *test.s* (por causa da opção -S), contendo as funções descritas em *test.c* com uma grande quantidade de comentários no código gerado (opção -fverbose-asm). E o

1 Note que *assembler* é o nome que se dá ao compilador da linguagem *Assembly*.

código em assembly usará o sabor Intel (-masm=intel) otimizado ao máximo (-O3) para o seu processador (-mtune=native)[2].

A linguagem C, diferente do assembly, é portável. Quero dizer, o que você fizer em C pode, em teoria, recompilar em qualquer plataforma e ter mais ou menos a certeza de que vai funcionar. Isso não ocorre com linguagens de baixo nível. E aqui vale uma explicação sobre esse "baixo nível": Não é que isso seja um insulto. O nível de uma linguagem tem haver com as abstrações que ela oferece ao desenvolvedor... Quanto mais abstrata for uma linguagem, maior o seu nível. Conceituo "abstração", aqui, como o quão mais perto podemos chegar de uma linguagem natural[3], coloquial.

Voltando à portabilidade, em muitos casos uma rotina escrita em assembly, para a plataforma i386, não funcionará na plataforma x86-64 e vice versa. É preciso tomar muito cuidado se você quiser garantir a portabilidade ou, em casos extremos, que seu código continue funcionando em processadores similares aos que temos hoje, no futuro[4]. Um exemplo disso é que códigos escritos para a arquitetura 8086, no início dos anos 90, provavelmente não funcionarão bem nas arquiteturas i386 ou x86-64.

É um fato incontestável que somente rotinas cuidadosamente desenvolvidas completamente em assembly **podem** atingir a melhor performance possível. Mas também é verdade que a otimização que você faz hoje, aproveitando recursos de hardware, podem ser completamente inválidas daqui a alguns anos. Apenas o desenvolvedor muito experiente consegue criar código totalmente portável **em qualquer nível** (e, mesmo assim, isso não é garantido!). Criar código totalmente portável em assembly é um desafio e tanto, confie em mim... pelo menos dessa vez!

Sobre o modo preferido de operação do processador

Em todo o texto que você vai ler farei referência aos registradores do processador a partir de seus nomes especificados no modo x86-64. Quebrarei essa regra quanto estiver falando especificamente do modo i386.

Como você verá mais adiante, EAX é um caso particular de RAX, assim como AX é um caso particular de EAX. Me parece mais lógico escrever RAX quando faço referência ao "acumulador" (o "A" de RAX e derivados). Fique ciente que RAX não pode ser usado no modo i386 e, neste caso, você poderá substituir o nome por EAX.

Quanto ao "modo preferido", a Intel nos diz que o modo protegido i386 é o modo "nativo", no entanto, seu processador acorda no chamado modo "real", de 16 bits. De fato, a Intel nos diz que o processador começa sua jornada no modo protegido e é colocado no modo real durante o *reset*. O modo "real" é o mesmo usado pelos processadores 8086, no antigo MS-DOS. Não falarei muito sobre ele aqui. Acredito que já disse o que tinha para falar no meu "Curso de Assembly da RBT", que pode ser baixado no link https://goo.gl/2Na5LX.

2 Importante notar que "-mtune=native" é apenas uma dica para o compilador. Se usássemos "-march=native" o código final será forçado a estar em conformidade com o seu processador, usando recursos disponíveis só nele, por exemplo.

3 Linguagens de programação são ditas "livre de contexto". Isso quer dizer que uma ordem deve ser interpretada tal como está escrita... Uma linguagem "natural" não é assim. Depende do humor dos interlocutores, de *inuendos*, etc... Assim, uma linguagem de programação jamais poderá ser "natural".

4 Não quero incorrer no mesmo erro que cometi no antigo "Curso de assembly da RBT". Aqui eu estou te dizendo que, provavelmente, um grande conjunto de dicas serão obsoletas em alguns anos! Tome cuidado!

Capítulo 1 - Conceitos Básicos

Ao introduzir a linguagem assembly e/ou eletrônica digital, iniciar com conceitos sobre sistemas de numeração é sempre útil para entendermos **como** o processador lida valores numéricos inteiros e, mais tarde, com números fracionários (ponto flutuante). Além de constranger você, leitor, fazendo-o sentir-se como se estivesse de volta ao início do ensino fundamental, quero mostrar algumas curiosidades sobre números que talvez alegrem o seu dia...

O que são números?

Claro que o conceito é bastante básico e não deveria nem estar aqui... Tenha um pouco de paciência e continue lendo...

Um número é uma sequência de símbolos onde a posição onde se encontram têm significados específicos, ou seja, a posição onde um símbolo está, nessa sequência, é importante. Esses símbolos são chamados de "algarismos"[5]. Na era moderna, se não me engano lá pelo século X, houve a padronização, na Europa, pelo uso de 10 algarismos (de 0 até 9). Antes disso, há milênios, povos assírios usavam 60 algarismos diferentes e ainda hoje temos resquícios dessa base em nossa cultura (1 hora tem 60 minutos; Uma circunferência tem 360° – que é um múltiplo de 60), mais tarde, outros povos já experimentaram sistemas numéricos que usavam 12 e isso também ainda ressoa entre nós. O termo "dúzia" vem do uso antigo de 12 algarismos.

O conceito de sistema numérico ou **base numérica** é interessante porque, ao lidarmos com programação, especialmente em assembly, teremos que lidar com outros tipos de números que não são decimais. Usaremos sistemas com dois algarismos (0 e 1), 8 algarismos (raramente!), 10 algarismos (o nosso preferido do dia a dia!) e 16 algarismos... O nome dado para esses "sistemas de numeração" são, respectivamente: binário, octal, decimal e hexadecimal. São referências ao número de algarismos aceitáveis no sistema numérico.

Como números são formados por uma sequência de algarismos, a posição mais à esquerda dessa sequência tem maior significado do que a posição mais à direita. Por exemplo, o número 32 tem significado completamente diferente do número 23, mesmo que ambos usem os mesmos algarismos '2' e '3'. Por causa da importância da posição de cada algarismo na sequência é que dizemos que usamos uma **notação posicional** para escrevermos números.

Escrevendo números de bases diferentes

Para diferenciar os tipos de números que escreveremos, as bases numéricas serão especificadas com prefixos. No caso das bases octal e hexadecimal usarei os prefixos '0' (zero, para octal) e '0*x*' (para hexadecimais), respectivamente. No caso da base binária usarei o prefixo '0*b*'.

Assim, 0*x*A3 (ou 0*x*a3, tanto faz se usarmos maiúsculas ou minúsculas) é um valor em hexadecimal, 031 é um valor octal (note o '0' na frente) e 0*b*0101 é um valor em binário. O mesmo número 31, sem o '0' na frente, está em base decimal.

Explicando melhor a notação posicional

A posição onde o algarismo se encontra no número diz a *grandeza* que esse número representa. Numeramos as posições de cada algarismo a partir da unidade, atribuindo o valor 0 (zero) a essa

5 Como sou chato pra caramba, ai vai a origem da palavra. Trata-se de uma corruptela do nome de um filósofo arabe do século IX: *Al-Khwärizmï*., de onde a palavra **algoritmo** também é derivada.

posição e incrementando esse valor para as demais posições à esquerda. No número 175 o algarismo '5' está na posição 0 (zero), o algarismo '7' na posição 1 e o algarismo '1' na posição 2. O número da posição aumenta a medida que escrevemos mais algarismos à esquerda (e diminui se formos para a direita – você verá isso quando chegar no capítulo sobre “ponto flutuante”).

Por causa dessa notação posicional podemos escrever o valor 175 como:

$$175 = 1 \cdot 10^2 + 7 \cdot 10^1 + 5 \cdot 10^0$$

Fica evidente que a posição onde o algarismo está, no número, determina o expoente da potência de base 10 ao qual é multiplicado. A base, é claro, nos diz a quantidade de algarismos possíveis, em cada posição, que podemos usar no número. É por isso que dizemos que o número 175, neste caso, está expresso na **base decimal**.

O mesmo princípio é usado quando lidamos com quaisquer outras bases numéricas. Na base binária (base 2) só podemos usar 0 ou 1 em cada uma das posições. O mesmo valor 175 pode ser escrito, em binário, como 0*b*10101111. Basta usar o mesmo método, descrito acima, para comprovar isso:

$$\begin{gathered} 0b10101111 = \\ 1 \cdot 2^7 + 1 \cdot 2^5 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = \\ 128 + 32 + 8 + 4 + 2 + 1 = \\ 175 \end{gathered}$$

Essas são as duas bases numéricas (binária e decimal) que de fato nos interessam. A base binária porque é usada pelos circuitos do seu computador e a base decimal porque a usamos todo dia. Então, pra quê diabos servem as bases octal e hexadecimal?

Quanto menor a base, maior é o número

Repare que um valor binário usa uma sequência maior de algarismos do que a base decimal. Enquanto escrevemos 175 com 3 algarismos, temos que fazer o mesmo com 8 bits[6], em binário. Como regra geral, quanto menor a base numérica, mais espaço o número ocupa no papel. Então, se usarmos uma base maior que a decimal escreveremos menos. E esta é a chave para a existência das bases octal e hexadecimal.

A beleza da base hexadecimal é que cada algarismo corresponde exatamente a um conjunto de 4 bits. Com 4 bits podemos escrever até 16 valores diferentes: de 0*b*0000 até 0*b*1111. Ao invés de escrever um byte com 8 bits, agora fica mais fácil escrevê-lo com 2 algarismos hexadecimais! Como não temos mais que 10 símbolos “numéricos”, tomamos emprestado as letras de A até a F para completar o conjunto e fazemos 'A' corresponder ao 10, 'B' ao 11,... até 'F', que será 15. A partir dai usamos a notação posicional para obter o valor decimal equivalente ao número hexadecimal. Por exemplo:

$$0xAF = 10 \cdot 16^1 + 15 \cdot 16^0 = 160 + 15 = 175$$

Já que o valor 0xAF, quando expresso em binário, é exatamente 0*b*10101111 e, como vimos, o valor decimal é 175, repare que se 'A' é 10 (em binário é 0*b*1010) e 'F' é 15 (em binário, é 0*b*1111). Achar o valor decimal, de volta fica mais fácil, basta usar potências de 16 para cada posição.

A base octal funciona do mesmo jeito, só que representa 3 bits por algarismo ao invés de 4 (em binário, de 0b000 até 0b111). O mesmo valor binário 0*b*10101111 pode ser expresso como 0257.

> Note que, diferente das bases hexadecimal e octal, a base 10 **não** possibilita o agrupamento de bits da mesma maneira... Aliás, se você fizer isso (por exemplo, escrever 193 como 0*b*000110010011) obterá uma codificação especial conhecida

6 A palavra **bit** é a contração, em inglês, de *binary digit*. Literalmente “digito binário”.

como **BCD** (*Binary Coded Decimal*) que, normalmente, **não** é reconhecida pelo processador... Ou seja, **não faça isso!**

Um ponto interessante é que existem instruções preparadas para lidar com BCD, como AAA, AAM, DAA... Mas elas **não** estão disponíveis no modo x86-64.

A base octal não é tão usual, hoje em dia, porque a hexadecimal oferece uma "compressão" maior. Ou seja, 32 bits podem ser expressos com 8 algarismos hexadecimais, mas exigem 11 algarismos octais para obtermos o mesmo efeito. Mas, é claro, que se você deseja agrupar um valor de 3 em 3 bits o sistema octal é o ideal para você.

O sistema octal apresenta um outro problema. Notaram que com 3 algarismos na base octal poderíamos escrever 9 bits. Mas, para escrevermos 8 bits, em octal, temos que limitar o algarismo mais significativo entre 0 e 3 (em binário, entre 0*b*00 e *0b*11). Já o sistema hexadecimal se encaixa perteiftamente em tamanhos de 4, 8, 16, 32, 64, 128, 256 e 512 bits (tamanhos usados na plataforma Intel).

Mais sobre os significados dos "lados" de um número...

Acho que já estamos de acordo que, quando você escreve um número com mais de um algarismo, um deles tem mais "importância" que o outro. Outro exemplo clássico é aquele em que você tem R$ 1001,00 no bolso e, se você tiver que perder um dos dois "1" do valor, qual você preferiria perder? O da esquerda ou o da direita?

A não ser que você esteja internado num hospício porque rasga dinheiro, acredito que perder o "1" mais à direita seria a sua escolha óbvia. Saber que está faltando R$ 1,00 no seu bolso é menos desesperador do que dar pela falta de R$ 1000,00. Pode-se dizer, então, que algarismos localizados à direita de um número são **menos significativos** que aqueles localizados à esquerda.

Com valores binários e hexadecimais é comum usarmos as siglas **MSB** (*Most Significative Bit* [ou *byte*]) para falarmos de bits mais à esquerda e **LSB** (*Least Significative Bit* [ou *byte*]) para os mais à direita. Veja o comentário dessa instrução em assembly:

```
and al,0xf0     ; zera os 4 lsb.
```

O significado da instrução, por enquanto, ainda não nos interessa... O interessante aqui é: Quem quer que tenha criado esse código te disse que essa instrução colocará zeros nos 4 bits menos significativos (LSB), ou seja, nos bits 0 até 3.

As siglas MSB e LSB não se aplicam apenas a bits. Podemos estar falando de bytes também. No número 0xAF32 o valor 0x32 localiza-se no LSB e 0xAF no MSB, por exemplo... O uso dessas siglas para referenciar bits ou bytes depende do contexto e elas existem apenas para simplificar o texto. Ao invés de dizermos "os 8 bits menos significativos da variável x", podemos dizer "o LSB de x". Para diferenciar se estamos falando de bits ou bytes é comum escrevermos "o bit LSB" ou "o byte MSB".

Aritmética inteira

Sinto que talvez você já esteja me xingando, achando que eu o esteja tratando como alguém que ainda está no jardim de infância, né? Não se desespere, meu rapaz (ou moça)! Esses conceitos, por mais básicos que pareçam, são importantes! Vou continuar abusando de sua paciência mostrando como fazer adições e subtrações.

Somar e subtrair dois números inteiros, binários (a-há!), é a mesma coisa que somar e subtrair dois números inteiros decimais... bem... somar é igual, subtrair nem tanto.

Numa adição estamos acostumados com o artifício do transporte (*carry*, em inglês). Acontece quando tentamos somar dois algarismos que resultam em um valor que não pode ser expresso numa única posição (ou num único algarismo). Por exemplo, 8 + 3 precisa de dois algarismos como resultado: 11. Então esse "1" excedente é "transportado" para a posição seguinte e somado ao resultado da adição na outra posição... Eis um exemplo de adição parcial:

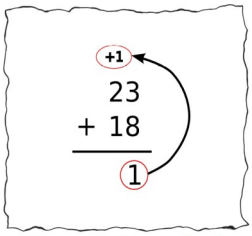


Figura 1: Exemplo de adição parcial

O resultado final, obviamente, é 41, já que na segunda posição temos 2 + 1 adicionados ao valor excedente da adição anterior, transportada (*carry out)*, para a posição atual (*carry in*).

A mesma coisa ocorre com a base binária, com dois detalhes importantes: Apenas os algarismos 0 e 1 são válidos (devido a base numérica) e o processador tem que lidar com quantidade limitada de bits. Ou seja, ao somar dois números de 32 bits você obterá um resultado em 32 bits... Se houver um *carry out* do MSB para o bit 32, então esse bit é colocado em outro lugar, mas não aparecerá no resultado final.

Esse último detalhe é importante... Se o processador pudesse lidar apenas com 2 bits de cada vez e você tentasse fazer a adição de, por exemplo, 0*b*10 + 0*b*10, necessariamente obteria 0*b*00 como resposta e um *carry out*, que não estará presente no resultado.

Isso é estranho! Como assim $2 + 2 = 0$? Este é precisamente o caso porque o valor 4 só pode caber exatamente em 3 bits (um bit a mais do que esse nosso processador hipotético suportaria). Então, na adição acima, com apenas dois bits, o resultado não pode ser representado corretamente... Dizemos que a operação **transbordou** (*overflow*).

Figura 2: Eis um odômetro

O contrário de *overflow* é *underflow*. *Underflows* acontecem na subtração... Se você tentar subtrair 1 de 0, num processador de 8 bits, obterá um valor com todos os bits setados: $0x00 - 0x01 = 0xFF$! Para entender isso uma analogia é necessária: A aritmética inteira pode ser comparada a um odômetro. Se ele está todo zerado e você der uma volta para trás, obterá todos os algarismos em "9". A figura ao lado ilustra o ponto...

É fácil entender o *carry* numa adição, mas ele também acontece numa subtração – neste caso o nome muda de *carry* para *borrow* (empréstimo). Em decimal, ao tentar subtrair 23 de 30 (isto é, $30 - 23$), você percebe que o algarismo menos significativo de 30 é menor que o mesmo algarismo de 23 (0 é menor que 3). Para poder fazer a subtração você "pega emprestado" (borrows!) uma dezena da próxima posição e realiza a subtração ($10 - 3 = 7$), não é isso?

Mas, o que acontece se você tentar subtrair, em binário, os valores 0*b*10 de 0*b*01 (ou seja, 0*b*01 - 0*b*10)? O LSB é simples: 1 menos 0 resulta em 1, mas e o próximo bit (0 - 1)? Da mesma forma que na adição, não dá para subtrair um algarismo maior de um menor, assim o processador tem que "tomar emprestado" o valor 1 de algum lugar e fazer ($0b10 - 1 = 2 - 1 = 1$)... Se isso acontecer no bit MSB, o processador não tem escolha a não ser "fazer de conta" que existe um "1" além deste bit para tomá-lo emprestado... Esse empréstimo será sinalizado no *flag* de *Carry*.

É por isso que 0*b*00000000 menos 0*b*00000001 resulta em 0*b*11111111 (e o flag de *carry* estará setado). E é por isso, também que o flag de *carry* pode ser usado para determinar *overflows* e *underflows* em operações de adição e subtração inteiras **sem o uso de sinal** (apenas considerando valores positivos). Sobre flags, veremos como funcionam mais adiante!

Carry e "borrow" são complementares

Note que, assim como na matemática, subtrações podem ser feitas através de adições, mas isso implica no uso diferente do carry. Por exemplo: Se fizermos "x = x + (-1)" estamos somando um valor com todos os bits setados a *x* e, se *x* for 1, o resultado final será 0x00 e o carry estará setado. No contexto de uma subtração o carry deveria estar zerado, já que $1 - 1 = 0$ e não há empréstimo algum. Ao realizar uma adição com um valor negativo, obteremos o mesmo resultado que obteríamos numa subtração, mas a indicação de overflow será **invertida**.

Mas note que numa instrução de subtração um *flag* de carry representa o borrow, ocorrido na operação. Quero dizer que uma instrução do tipo:

```
sub eax,1
```

Se EAX for zero, por exemplo, resultará no *flag* de carry setado. Da mesma forma que ocorreria se você fizesse uma adição de 1 a um valor 0xFFFFFFFF em EAX. Mas, observe que, se mantivermos EAX com zero e fizermos:

```
add eax,-1
```

O *flag* de carry estará zerado. Afinal, somar 0xFFFFFFFF a 0 continua dando 0xFFFFFFFF e não há nenhum "vai um".

Falo sobre flags mais adiante...

Adição e subtração com sinais

Por incrível que pareça, processadores não sabem o que é um número negativo.

Números inteiros negativos são uma interpretação do seu programa, não do processador. A convenção é a de que, se você quer lidar com tipos inteiros sinalizados (quer dizer: que podem ser interpretados como negativos), então, se o bit MSB estiver setado, indicará que esse valor é negativo. Mas a coisa **não é tão simples** como dizer que, em 8 bits, 0*b*10000001 simboliza -1 (bit 7 setado e o restante do valor binário é 1). Afinal, se você subtrair 1 de 0*b*00000000 obterá 0*b*11111111! Neste exemplo o bit 7 está setado, mas o restante do número não lembra em nada o valor 1, em decimal ou binário, lembra?

O macete é o seguinte: Sempre que você vir um inteiro com o bit MSB setado e quer usá-lo como se ele tivesse sinal, inverta todos os bits e adicione 1. Por exemplo, o valor 0*b*11111111 tem o bit 7 setado, então ele pode ser interpretado como se fosse negativo. Daí, invertendo todos os bits obteremos 0*b*00000000 e depois de adicionarmos 1 e obtemos 0*b*000000001. Isso significa que 0*b*11111111 equivale a -1.

Este é o motivo pelo qual o tipo *char*, em C, acomoda valores entre -128 e 127. O maior valor positivo que podemos obter com 8 bits é 127 (0b01111111) e o menor é -128 (0b10000000).

> Lembre-se disso: Nas operações de adição ou subtração inteiras o processador **não** entende o que é um valor negativo! Se o seu programa está usando essa valor como se fosse negativo, estará usando a técnica acima, chamada de **complemento 2**.

Os modificadores de tipos, *signed* e *unsigned*, existem para que o compilador saiba como usar os flags na hora de comparar valores. Veremos os flags mais adiante... O fato é que as operações aritméticas e lógicas (e a movimentação de dados) não ligam se o tipo integral tem ou não sinal!

Com relação aos números inteiros dou uma pausa para que você pare de me xingar por estar te ensinando a somar e subtrair e mudar um pouco de assunto...

Modos de operação dos processadores 386 ou superiores

Todas as arquiteturas de processadores da família 80x86 modernos são derivados, diretamente, do bom e velho 80386. E no 386 temos, basicamente, 2 modos de operação:

- **Modo Real**: Este é o modo no qual o processador se encontra depois de um *reset* ou *power up*. E é o modo em que a BIOS trabalha. Neste modo podemos acessar, no máximo, 1 MiB de memória; não existe divisão de espaços entre o *kernelspace* e o *userspace*; não existe *memória virtual*; e multitarefa é algo bem limitado (o processador suporta, mas os sistemas operacionais nesse modo não costumam suportar! – é o caso do MS-DOS). Este é o modo de "16 bits".
- **Modo Protegido (i386)**: O termo "protegido" aqui significa que, neste modo, é possível separar regiões da memória acessíveis apenas para o *kernelspace* e *userspace*, separadamente. Ou seja, podemos proteger um do outro; A capacidade total de acesso à memória é de 4 GiB (no modo não paginado) e até 64 GiB (usando paginação); também temos o suporte total do processador para multitarefa. Este é o modo de 32 bits, é o modo que chamo aqui de "i386".

Em processadores mais recentes existe um terceiro modo: **Modo x86-64**: Funciona de forma parecida com o modo i386, mas os endereços agora têm 64 bits de tamanho[7], os registradores têm 64 bits de tamnho, temos mais registradores e outras pequenas diferenças interessantes...

Organização da memória do seu programa

Eis um dos princípios mais básicos de toda a ciência de computação: Todo computador exige que estejam presentes, pelo menos, 4 grandes blocos funcionais[8]:

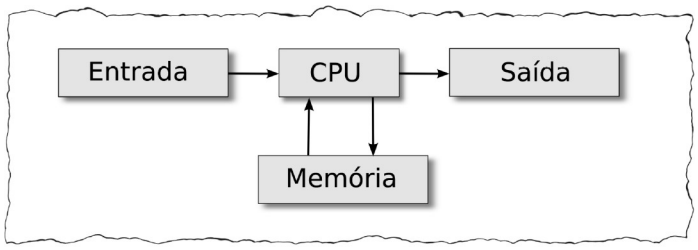


Figura 3: As quatro funções essenciais de um computador

Dentre outras coisas, o diagrama acima mostra que a CPU precisa de memória para funcionar. Essa "memória" é algum dispositivo onde seja possível armazenar "bytes". O bloco chamado de CPU até tem alguma "memória" limitada, como veremos mais adiante, mas ele precisa de mais... Existem memórias temporárias (RAM), permanentes (ROM) e de armazenamento (HD, CD-ROM, Pendrives, etc)... Estamos interessados aqui apenas em memórias temporárias (RAM)...

7 Na verdade, até 52... falo disso depois...

8 Dedicarei aqui minha atenção a arquitetura *Von Newmann*, deixando a *Harvard* de lado...

Para ler ou gravar um bloco de dados (bytes) na memória o processador precisa de uma maneira de localizá-lo. A maneira mais simples de compreender como isso é feito é imaginando a memória como se fosse um grande *array* de bytes, onde cada byte é encontrado através de um índice. Assim, temos um byte na posição 0, outro na posição 1 e assim por diante.

Cada posição desse array é conhecido como *endereço*. Em termos de circuitos, o processador possui um barramento de endereços e um barramento de dados. O barramento de endereços fornece o endereço do dado que o processador quer acessar (ler ou gravar) no circuito contido no seu pente de memória RAM. O pente decodificará esse endereço e selecionará o conjunto de bits que são colocados (quando lidos pela CPU) ou obtidos (quando gravados pela CPU) do barramento de dados.

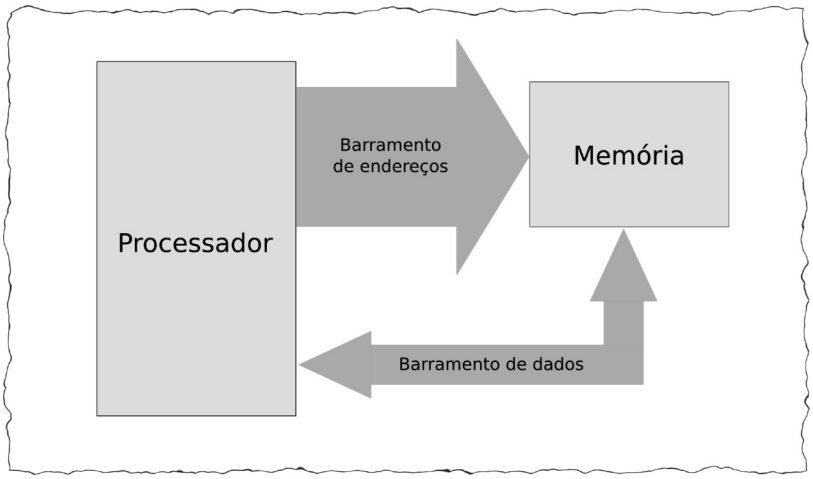


Figura 4: Barramentos do processador

O que a figura acima, simplificada, tenta te dizer é exatamente o que falei antes... O processador coloca um valor binário no **barramento de endereços** e o circuito das memórias usa esses bits para selecionar o dado que vai ser lido ou gravado através do **barramento de dados**[9].

Todo esse papo é para te deixar ciente de um simples fato: O processador só consegue armazenar algo em um circuito externo chamado memória e o faz através de endereçamento, como se fosse um grande *array* de bytes, que começa no endereço *zero* e, no caso da arquitetura x86-64, termina, potencialmente, num endereço 0*x*000FFFFFFFFFFFF (do MSB para o LSB: 12 zeros e 52 1's)[10]. É claro que, quando eu digo "potencialmente" estou me referindo à *memória física*, que corresponde a toda a memória endereçável (RAM, nos pentes DIMM, e ROM)... Os 52 bits do barramento de endereços te darão a capacidade teórica de endereçar até 4 PiB (ou seja, mais ou menos 4 seguido de 15 zeros) de RAM. É claro que isso é economicamente impraticável! Hoje, temos máquinas potentes com alguns *giga bytes* de memória RAM (8 ou 16 pareciam estar se tornando o padrão para desktops caseiros quando escrevi esse livro).

Além dessa organização genérica da RAM, seu programa só enxerga uma fatia da memória, reservada pelo sistema operacional para o uso do processo corrente. No modo i386 é comum que o sistema operacional só deixe você acessar até 2 GiB de RAM. No modo x86-64 esse tamanho máximo pode ser bem maior... Como regra geral, os Oss costumam disponibilizar um espaço virtual de metade da memória física endereçável. No modo i386 um endereço físico tem 32 bits de tamanho, assim, 31 bits são endereçaveis pelo seu processo (em teoria). No modo x86-64, com 52 bit possíveis no endereço, até 2 PiB (2048 TiB) são endereçáveis... Na prática você tem acesso a

9 Ok, além de ser super simplificada existem alguns exagegeros tradicionais aqui. Em processadores mais modenos o barramento de endereços é, de fato, **menor** que o de dados. Em processadores da arquitetura *Sandy Bridge* ou *Nehalem* (ambos da Intel) o barramento de endereços tem 52 bits de tamanho, enquando que o de dados tem 64 ou até 128 bits... Sim! 128 bits! Para acelerar recursos como SSE, por exemplo.

10 Estou deixando de lado, por enquanto, coisas como *memory mapped I/O* e o *Virtual Address Space*.

bem menos que isso, já que o limite físico mais o espaço reservado para *swap* limitam a faixa.

O endereço inicial do bloco de memória acessível por seu programa depende do sistema operacional. No Windows e no Linux, no modo x86-64, costuma ser 0x400000, mas não confie muito neste valor. Existem métodos de tornar esse endereço lógico aleatório.

Organização da memória do seu PC

Para quem for desenvolver o próprio kernel é interessante conhecer como o PC organiza a memória. Tanto no que se refere à memória RAM, quanto a ROM e memória mapeada para I/O. Diferente do que eu disse lá em cima, a memória do seu sistema **não** é linear. O PC "quebra" os espaços usados pela RAM, ROM e I/O em blocos não contínuos. Os primeiros 640 KiB, por exemplo, são dedicados a RAM chamada de "memória baixa". Segue um bloco de uns 128 KiB dedicados à memória de vídeo (VRAM de "*Video RAM*") seguido ainda de espaço de ROM que pode ou não estar presente. Outros 128 KiB são dedicados à ROM-BIOS. Na verdade esses 128 kiB finais da memória baixa podem ser a própria ROM-BIOS ou uma "sombra" dela, um pedaço da RAM marcada como *read-only*, via algum artifício de hardware ou software.

Pode parecer que os primeiros 640 KiB são completamente usáveis pelo seu processo, mas isso não é verdade. Primeiro, o espaço de 1 KiB inicial é usado para conter uma tabela de endereços no estilo *segmento:offset* chamada de IVT (*Interrupt Vectors Table*). Essa tabela tem 256 entradas, correspondentes a cada uma das 256 rotinas de tratamento de interrupção (ISRs ou *Interrupt Service* Routines) possíveis[11]. Em segundo lugar, existe um espaço de 512 bytes usado para conter variáveis usadas pela BIOS, chamada de BDA (*Bios Data Area*), do endereço físico 0x400 até 0x5ff. Embora essa área seja parte da RAM, o uso é dirigido e não deve ser modificado levianamente.

Na parte superior da memória baixa, antes da memória de vídeo, temos uma região de 64 KiB que **pode** ser usada para I/O mapeada em memória, do endereço físico 0*x*90000 até 0*x*9ffff. Como vimos o espaço entre 0*x*A0000 até 0*x*Bffff é dedicada à VRAM. A partir de 0*x*C0000 até 0*x*Dffffff (128 KiB) podemos ter ROMs contendo código e dados de dispositivos que estejam espetados no seu computador e, à partir de 0*x*E0000 até 0*x*fffff temos a ROM-BIOS.

Isso nos deixa com a região entre 0*x*600 até 0*x*8ffff disponível para RAM, ou pouco mais que 574 KiB.

Essa "memória baixa" é endereçável no primeiro 1 MiB, onde o endereço pode ter 20 bits de tamanho ($2^{20} = 1048576$ bytes) e, portando, acessível ao esquema *segmento:offset*, usado no modo real. Por outro lado, a "memória alta" não é acessível nesse modo. Qualquer coisa acima do primeiro 1 MiB só pode ser acessada nos modos de operação mais avançados: No modo protegido.

Geralmente, quando um sistema operacional é inicializado em modo protegido ele "se copia" para o início da região da memória alta e continua sua carga de lá. O motivo é simples: Temos **muita** memória RAM disponível nessa região... Num sistema com 8 GiB de RAM, 99,98% de toda a memória RAM está localizada na memória alta... Existem outros motivos: O sistema precisará usar a memória baixa para lidar com DMA, por exemplo...

Mas, da memória alta, existem regiões reservadas: Dispositivos PCIe, USB, Controladora de disco, Chipset de áudio etc, usam uma fatia da memória alta para si... Essa fatia não é RAM, é mapeamento de I/O em endereços de memória. Para maiores detalhes, consulte a documentação dos PCH (Peripheral Hub Controller) PIIX3 (82371SB chip), PIIX4 (82371AB chip), ICH7 e ICH9, onde PIIX é a sigla de *PCI*, *IDE*, *ISA Xcelerator* e ICH é a de *I/O Controller Hub*. Ambas as especificações da Intel. Ambas usadas em muitos PCs.

11 Falarei sobre *interrupções* mais tarde!

Por exemplo, certos PCs reservam os primeiros 15 MiB de memória alta para si (COMPAQ) e não colocam nada no lugar. Este é o primeiro motivo pelo qual o Linux, por exemplo, faz a cópia do kernel, em tempo de boot, para o endereço físico 0x100000, ou seja, a partir dos 16 MiB. Da mesma forma, alguns dispositivos gráficos podem reservar um espaço de 1 MiB entre 15 e 16 MiB de RAM (placas AGP, por exemplo)...

Quanto aos endereços reservados para os hubs (PCH, ICH, PIIX), geralmente começam no endereço físico 0*x*e0000000 a vão até 0*x*ffffffff, no finalizinho do espaço endereçável em 32 bits. A figura à seguir tenta mostrar essas regiões reservadas sem muitos detalhes (podemos ter blocos intermediários reservados também!):

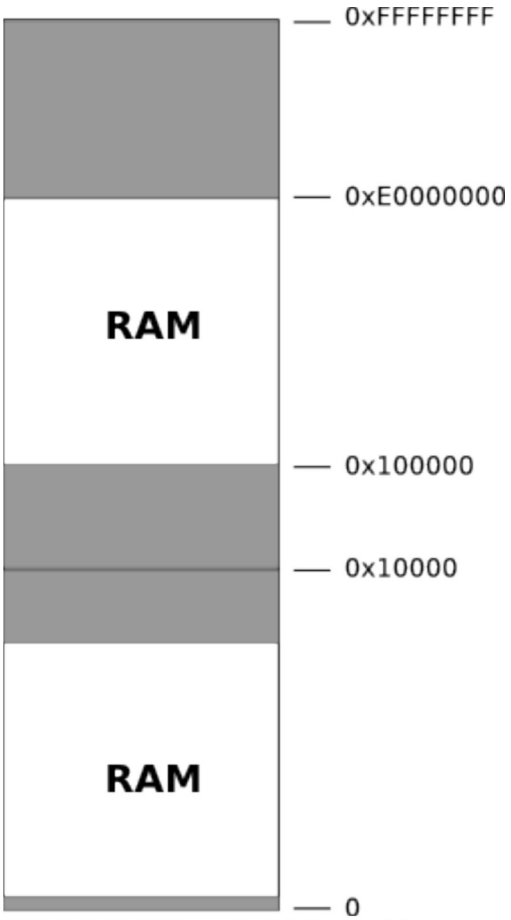


Figura 5: Regiões reservadas da memória do PC

Registradores

Embora o processador precise de memória para armazenamento, tanto do programa a ser executado quanto dos dados que serão lidos e gravados, ele também possui alguma memória interna. Não estou falando sobre *caches*, ainda, e explorarei isso mais tarde. Aqui eu me refiro a algumas "variáveis" internas que um programa escrito em assembly usará. Na literatura técnica essas "variáveis" internas são chamadas de "registradores".

O modo i386 define 10 registradores de uso geral ou GPRs (*General Purpose Registers*). São eles: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP e EFLAGS. Todos possuem 32 bits de tamanho. Os registradores EIP e EFLAGS são especiais e, normalmente, não são diretamente acessíveis.

Os nomes desses registradores têm fundamento:

Registrador	Descrição
EAX	“a” significa acumulador.
EBX	“b” significa “base”.
ECX	“c” significa “contador”.
EDX	“d” significa “dados”.
ESI	“si” significa *source index* (índice fonte).
EDI	“di” significa *destination index* (índice destino).
EBP	“bp” vem de *base pointer* (ponteiro base – da pilha).
ESP	“sp” vem de *stack pointer* (ponteiro da pilha – do “topo” da pilha).
EIP	“ip” é o *instruction pointer* (ponteiro de instrução).
EFLAGS	Este é o registrador que contém os *flags* de *status* do processador.

Tabela 1: Significado dos nomes dos registradores de uso geral

Essa nomenclatura existe porque, em determinadas circunstâncias, alguns registradores têm que ser usados de acordo com o nome que têm. Às vezes devemos usar ECX como contador... outras vezes podemos usar EBP como um ponteiro para a base da pilha. Os registradores ESI e EDI, com certas instruções, são usados como índices (ou, mais precisamente, ponteiros) para manipulação de blocos.

No entanto, é interessante notar que apenas ESP, EIP e EFLAGS têm destino certo e não podem ser usados para outra coisa (ou, pelo menos, é imprudente fazê-lo). Os demais podem ser usados de maneira geral daí o nome GPR...

Mas, se temos apenas registradores de 32 bits, como fazer para manipular tamanhos menores de dados contidos nesses registradores? Quero dizer, como lidar com 8 ou 16 bits (bytes e words)? Bem... existem apelidos para pedaços desses registradores. O registrador EAX, por exemplo, pode ser acessado também pelos pseudo-registradores AX, AH ou AL, onde AX corresponde aos 16 bits LSB de EAX. Os pseudo-registradores AH e AL, por sua vez, correspondem aos byte MSB e LSB de AX, respectivamente (veja a figura no próximo tópico).

Não dá para fazer isso com todos eles. ESI, EDI, EBP e ESP, por exemplo, só possuem o equivalente de 16 bits (SI, DI, BP e SP) no modo i386. No modo x86-64 a coisa muda de figura, à seguir...

Além dos GPRs temos também registradores *seletores de segmentos* e de controle. Existem ainda outros registradores “mapeados” em memória e um monte de outros *dependentes* de arquitetura (chamados MSRs – *Machine Specific Registers*). Não falarei quase nada sobre esses últimos. Apenas os GPRs, seletores e talvez alguns registradores de controle são necessários (mas apenas os GPRs são “essenciais”!) para o entendimento de uma rotina escrita em assembly.

Registradores nas arquiteturas i386 e x86-64

Tanto no modo i386 quanto no modo x86-64 o tamanho *default* de registradores e dados é de 32 bits. O modo x86-64 nos dá a vantagem de podermos usar versões **estendidas**, de 64 bits, dos GPRs. Ainda mais: Existem mais GPRs disponíveis.

No modo x86-64 os GPRs são nomeados com um “R” na frente, ao invés de um “E”. Temos RAX, RBX, RCX, RDX, RSI, RDI, RBP e RSP, bem como RFLAGS e RIP. Temos também mais 8 GPRs nomeados de R8 até R15.

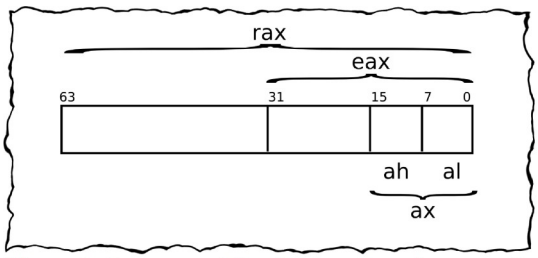


Figura 6: Estrtutura típica de um registrador no modo x86-64.

Do mesmo modo que ocorre com os GPRs de 32 bits, os registradores estendidos podem ser referenciados através de pseudo-registradores. RAX, por exemplo, possui os apelidos EAX, AX, AH e AL, de acordo com a figura acima. Essa facilidade é estendida para RDI, RSI, RBP e RSP. Como exemplo, o registrador RDI tem apelidos como EDI, DI e DIL (mas não tem DIH!). O mesmo acontece os registradores adicionais: R11, por exemplo, pode também ser referenciado via seus apelidos R11D, R11W e R11L, com tamanhos de 32 (doubleword), 16 (word) e 8 bits (byte), respectivamente.

Essa nova maneira de usar os pseudo-registradores tem o seu preço: **Não existe** uma maneira de acessarmos os 8 bits superiores de DI através de um "DIH". Isso vale para todos os GPRs, exceto RAX, RBX, RCX e RDX, que possuem seus respectivos AH, BH, CH e DH por motivos de compatibilidade.

Existe um macete interessante no modo x86-64 para inicializar registradores de uso geral. Se você atribuir valores a EAX, por exemplo, a porção superior do registrador RAX é automaticamente zerada. Por exemplo:

```
mov eax,0x7fffffff
```

Essa instrução inicializará EAX com 0*x*7fffffff, zerando a porção superior de RAX, inicializando-o com 0*x*000000007fffffff. Isso só funciona com as versões 32 bits dos GPRs. Se inicializarmos AX, por exemplo, os bits superiores de EAX e RAX **não** serão afetados.

Isso têm consequências interessantes. Se queremos zerar um registrador podemos fazê-lo via instrução MOV ou via uma instrução lógica XOR. Eis as μ_{op}s equivalentes de 4 métodos usando essas duas instruções:

```
b8 00 00 00 00                  mov eax,0
48 b8 00 00 00 00 00 00 00 00   mov rax,0
31 c0                           xor eax,eax
48 31 c0                        xor rax,rax
```

Do lado esquerdo temos os bytes correspondentes à instrução, em linguagem de máquina. Repare que "XOR EAX,EAX" é a menor instrução das quatro. Repare também que "MOV EAX,0" é menor que "MOV RAX,0".

Segmentos de memória

Na época em que processadores de 8 bits eram comuns na arquitetura de microcomputadores (Z80 e 6502, por exemplo), a memória era um array monolítico, com no máximo 64 KiB de tamanho. O processador conseguia acessar do endereço 0*x*0000 até 0*x*FFFF, sem nenhum problema. Uma instrução só tinha que informar o endereço e *voilà*! Eis uma instrução do Z80 como exemplo:

```
ld  a,(0x4002)       ; Carrega A com o byte localizado no endereço 0x4002.
```

Felizmente os processadores modernos, além de mais rápidos, são capazes de acessar bem mais que 64 KiB de memória. Os processadores *Intel* que suportam apenas o modo i386 podem, em teoria, acessar toda a memória do sistema do mesmo jeito que o Z80 fazia. Esses processadores têm barramento de endereços de, pelo menos, 32 bits de tamanho. Mas eles não acessam memória desse jeito... Por quê? Por causa da compatibilidade com o antigo 8086.

Na época em que a IBM lançou o PC ela escolheu a família de processadores 8088 da Intel[12]. A ideia era criar um microcomputador para uso caseiro (PC é a sigla de *Personal Computer*), mas com arquitetura de 16 bits. No final dos anos 70, e início dos 80, processadores de 8 bits eram mais comuns e todos os circuitos de acesso a memória usavam barramentos de 16 bits. Com 16 bits o máximo de memória endereçável era 64 KiB... O 8088 não era exceção. O barramento de endereços tinha 16 bits, mas usando um artifício o processador podia endereçar até 1 MiB! Como isso era possível?

Um endereço, quando colocado no barramento, era fornecido em duas fases: Primeiro os 4 bits superiores eram fornecidos e armazenados num *latch* (uma pequena memória) e, logo depois, os 16 bits inferiores, montando os 20 bits necessários para acessar um byte específico na memória. Resta saber agora como a Intel conseguiu especificar um endereço de 20 bits no software se os registradores do processador só tinham 16... Ela resolveu *segmentar* a memória em blocos de 64 KiB.

Um endereço era calculado através do valor contido num registrador que seleciona um *segmento*, deslocado 4 bits para a esquerda, e depois somando a um deslocamento de 16 bits:

```
Endereço = (segmento shl 4) + offset
```

Assim, um endereço é sempre especificado através de um par de valores especificados assim: *segmento:offset*. Os processadores modernos, da família 80x86, continuam usando essa técnica com algumas modificações...

Problemas com ponteiros "longínquos" no modo real

Um ponteiro é apenas uma variável que possui, em seu interior, um endereço de memória. Como vimos, no modo "real" um endereço é especificado por um par de valores: *segmento:offset*. Acontece que esse esquema, criado para possibilitar o uso de endereços de 20 bits em uma arquitetura de 16, permite a especificação de um mesmo endereço "físico" de várias formas... Suponha que queiramos especificar o endereço físico 0*x*0437. Podemos usar todas essas notações: 0*x*0000:0*x*0437, 0*x*0001:0*x*0427, 0*x*0002:0x0417, … 0*x*0043:0*x*0007. Ou seja, existem 67 maneiras diferentes, neste caso, para especificar o mesmo endereço "físico"!

Compiladores de alto nível, como C, preferiam usar dois critérios de especificação de segmentos: Ou usar o menor valor possível (e o maior offset possível) ou o maior... O segundo caso era o preferido, já que o offset poderia começar com valores inferiores a 16.

A notação *segmento:offset* é chamada de ponteiro "far" (o "longínquo" no nome do tópico) justamente porque podemos especificar qualquer lugar da memória. Ponteiros "near" (próximos) são aqueles que contém apenas a porção "offset" do ponteiro... O "segmento" é assumido como sendo o segmento default, de acordo com a operação (veja mais adiante).

Registradores que selecionam "segmentos" da memória

Temos 6 registradores de 16 bits usados para selecionar segmentos: CS, DS, ES, FS, GS e SS. E, já

12 Diz a lenda que por recomentação da Microsoft, que foi contratada para desenvolver o PC-DOS.

que eles “selecionam” segmentos, nada mais justo que sejam conhecidos como **seletores de segmentos**. Para escrever menos, daqui para frente, chamarei esses registradores pelo apelido carinhoso de “seletores”...

No modo “real” (16 bits) os seletores têm que ser carregados com o endereço do início de um segmento de 64 KiB de tamanho, deslocado em 4 bits para a direita e outro registrador ou valor constante é usado como deslocamento (offset) dentro deste segmento... O endereço 0*x*07C0:0*x*003F, por exemplo, faz referência à memória física localizada em 0*x*7C3F, já que a base do segmento é 0*x*7C00 (0*x*07C0 shl 4) e o deslocamento é 0*x*003F.

Outro detalhe importante do modo “real”, nos processadores anteriores ao 386, é que os seletores FS e GS não existiam. Eles podem ser usados se o processador for 386 ou superior, mas não nos mais antigos...

No modo i386 a coisa complica: Não é possível colocar um endereço base de um segmento num seletor, já que um endereço físico passou a ter 32 bits de tamanho e um seletor tem 16. Além do mais, temos todos aqueles recursos de proteção e tamanhos de segmentos maiores que 64 KiB! A solução é usarmos uma tabela contendo a descrição de um segmento e fazer com que um seletor aponte para um item dessa tabela... Surge o conceito de **descritores de segmentos**...

Nessa tabela, localizada na memória, o desenvolvedor é obrigado a criar entradas de 8 bytes que descrevem um segmento. A estrutura de cada entrada é essa:

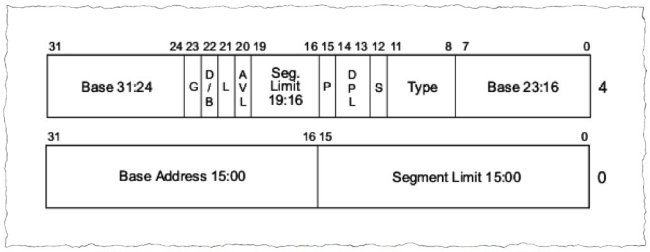


Figura 7: Descritor de segmento no modo i386

Onde: *Segment Limit* é o tamanho do segmento. Note que esse campo tem 20 bits de tamanho (nos bits 0~15 no offset 0 e nos bits 16~19 a partir do offset 4). Isso condiz com o que acontece no modo real, mas existem um bit nos **atributos** do descritor que aumentam essa capacidade (o bit G, de *Granulidade*, multiplica o limite por 4096).

Junto com o limite temos o *Base Address*, que é o endereço linear (por enquanto, podemos encarar isso como sendo o endereço “físico”) do início do segmento, de 32 bits de tamanho. Quanto aos atributos, vimos o bit G e temos os bits D/B, que informa se o segmento é de 32 bits ou 16 (serve para compatibilizar o i386 com o antigo 286); o bit L é usado apenas no modo x86-64; o bit AVL está “disponível” (AVaiLable) para o programador (não tem significado para o processador); o bit P indica se o segmento está presente na memória ou não; e o bit S, em conjunto com os bits do campo *Type*, indica o tipo de segmento (código, dados, ou algum segmento especial).

O campo DPL ou *Descriptor Privilege Level* informa o nível de privilégio necessário para acessar esse segmento. Privilégio 0 é usado no *kernelspace* e 3 no *userspace*.

A tabela de descritores contém entradas com essa estrutura. Cada entrada corresponde a um índice que será colocado nos seletores, que agora têm 3 informações: O índice para a tabela de descritores, o “tipo” de tabela usada (existem duas possíveis) e o nível de privilégio *requisitado* pelo seletor (RPL ou *Requestor Privilege Level*):

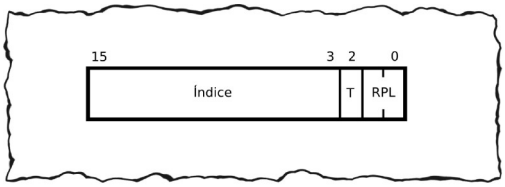


Figura 8: Estrutura de um seletor

Note que o índice tem apenas 13 bits de tamanho, assim, podemos ter “apenas” 8192 (2^{13}) entradas na tabela de descritores.

A não ser que você vá desenvolver um sistema operacional ou um *device driver*, raramente terá que se preocupar com os seletores. Escreva algumas rotinas em C e obtenha a listagem em assembly e verá que em **nenhuma vez** o compilador criará código que use, explicitamente, esses registradores ou sequer os modifique!

Embora a configuração necessária para usarmos seletores seja complicada, você pode estar curioso em como o processador os usa... A ideia é que cada um deles existe para isolar segmentos acessíveis pelo processador. O seletor CS especifica um segmento que conterá apenas instruções. O seletor SS especifica um segmento usado para “pilha”. Os seletores DS, ES, FS e GS especificam segmentos de dados:

Seletor	**Significado**
CS	Segmento de código (Code Segment)
SS	Segmento de pilha (Stack Segment)
DS	Segmento de dados (Data Segment)
ES	Segmento de dados extra (Extra data Segment)
FS	Outro segmento de dados extra (FS é a sequência a partir de ES. 'f' vem depois de 'e').
GS	Ainda outro segmento de dados extra (GS é a sequência a partir de FS. 'g' vem depois de 'f').

Tabela 2: Registradores seletores de segmento

Assim, um seletor seleciona (óbvio!) um segmento da memória. Ele contém um índice para uma tabela que contém o endereço base e o tamanho desse segmento. De posse desse endereço base, o valor contido em um ponteiro, que agora é apenas um offset dentro de um segmento, é adicionado e obteremos um *endereço efetivo*.

Seletores “default”

Dependendo do tipo de acesso à memória o processador usa, automaticamente, um ou outro seletor. O registrador RIP, por exemplo, usa o **sempre** seletor CS para montar o endereço efetivo da próxima instrução que será executada.

Da mesma maneira, se um ponteiro é formado a partir dos registradores RBP ou RSP, então o seletor SS (da pilha) é automaticamente usado. Qualquer outro tipo de acesso à memória é feito via seletor DS.

Existem instruções que usam o seletor ES e as veremos mais tarde. Mas, os seletores FS e GS geralmente estão disponíveis apenas como uma extensão e são usados apenas pelo sistema operacional.

Nossos programas, no *userspace*, tiram vantagem do comportamento default do processador e, por

isso, não precisamos especificar um seletor quando fazemos referência à memória.

Instruções

Todo programa é codificado numa sequência de bytes, chamados micro-códigos ou micro-operações (μ_{op}) que são interpretadas pelo processador como uma *instrução*. Instruções são ordens para o processador e costumam ser bem simples: “mova um valor para EAX”, “salte para o endereço 0x063045” ou “compare o valor no endereço contido no registrador RBX com o registrador CL”.

Os processadores da família 80x86 têm um conjunto bem grande de instruções. A maioria delas é tão simples como mostrei acima, outras são bastante especializadas. Uma instrução pode ser tão simples como “retorne!” ou tão complexas que precisem de três parâmetros adicionais. Por exemplo:

```
  mov eax,-1    ; EAX = -1.
  cmp edi,esi   ; Compara EDI com ESI.
  jz  .L1       ; Se a comparação deu “igual”, salta para “.L1”.
  inc eax       ; Incrementa EAX.
.L1:
  ret           ; Retorna!
```

Acima, temos uma pequena rotina escrita em **linguagem assembly**. Aqui temos instruções com nenhum, um ou dois **operandos**. Essa sequência de instruções é traduzida por um compilador para:

```
 b8 ff ff ff ff            mov    eax,0xffffffff
 39 f7                     cmp    edi,esi
 74 01                     je     +1
 40                        inc    eax
 c3                        ret
```

Onde, na coluna da esquerda, temos a sequência de bytes correspondentes as instruções escrita de maneira mnemônica do lado direito. A essa sequência de bytes dá-se o nome de **linguagem de máquina**.

Existem instruções que exigem 3 operandos, como é o caso de:

```
shufps xmm0,xmm1,0xb1
vfmadd132pd xmm0,xmm1,xmm2
```

E até algumas com 4 operandos, como:

```
vperm2f128 ymm0, ymm1, ymm2, 0xe4
```

Essas são intruções “avançadas” e você terá a chance de ver algumas delas em outro capítulo...

“Tipos” em assembly

Nem o processador nem a linguagem assembly entende tipos como *char*, *int*, *long*... Tudo o que o processador entende são *bytes*, words, *quadwords*... Os “tipos”, neste contexto, têm haver apenas com o tamanho do dado que queremos manipular:

“Tamanho” em assembly	Tipo em C	Tamanho (em bytes)
byte	char	1
word	short	2
dword	int	4
qword	long long	8
tbyte	long double	10

Tabela 3: Equivalência de tipos entre C e assembly

Esses são “tipos” que podem ser manipulados com o auxilio de um registrador, mas existem instruções que não usam registradores:

```
cmp [myvar],0    ; Compara o conteúdo da memória apelidada por 'myvar' com zero.
                 ; ISSO VAI DAR ERRO DE COMPILAÇÃO!
```

Ao tentar compilar uma instrução dessas o “compilador” assembly vai reclamar porque, embora ele saiba em qual lugar da memória estará alocado espaço para o símbolo *myvar*, ele não sabe o tamanho desse símbolo... Será que “myvar” deve ser tratado como tendo tamanho de 1 byte? Ou será de um dword? Quem sabe não é um qword?

Para esse tipo de construção ambígua temos que dar uma dica ao compilador, dizendo que *myvar* tem um tamanho conhecido:

```
cmp dword [myvar],0   ; Compara o conteúdo da memória apelidada por 'myvar' com zero.
                      ; Esse símbolo referencia uma região da memória cujo tamanho
                      ; deve ser tratado como dword (4 bytes).
```

Só desse jeito a instrução poderá ser codificada para lidar com um valor imediato (o zero) de 32 bits.

Essas dicas de tamanho são desnecessárias se estivermos usando um registrador. Como eles têm tamanho fixo, previamente conhecidos, o compilador já saberá, de antemão, o tamanho do dado a ser comparado:

```
cmp [myvar],eax       ; Compara o conteúdo apontado por 'myvar' com o registrador eax.
                      ; O compilador não reclama porque sabe que EAX tem 32 bits de tamanho.
```

Quanto ao tipo “tbyte”, ele é reservado para ponteiros de valores em ponto flutuante de precisão estendida, *long double*s. Falarei mais sobre o tipo *tbyte* no capítulo sobre ponto flutuante.

A pilha, funções e os registradores RSP *e* RBP

O processador precisa de uma região na memória que ele possa usar para armazenamento temporário chamada de “pilha”. Não é aquele tipo de pilha usada no controle remoto da TV... A analogia está mais para uma pilha de pratos. O último prato que você coloca em cima da pilha de pratos é o primeiro que você vai pegar quando tentar esvaziá-la. Na pilha do processador os dados são “empilhados” e “desempilhados” do mesmo jeito: O último dado empilhado será o primeiro desempilhado.

O registrador RSP contém o endereço, isto é “aponta”, do topo da pilha. E o “topo” da pilha é definido como a posição onde o último dado foi empilhado. É o lugar do último prato.

O detalhe é que essa pilha cresce para baixo. Quando “empurramos” (*push*) um valor para a pilha o registrador RSP é decrementado e depois o dado empurrado é gravado no endereço dado pelo RSP. É como se a gravidade funcionasse de maneira inversa. O primeiro prato fica grudado no teto e a

pilha cresce para baixo.

Para visualizar a estrutura de uma pilha usarei um diagrama como mostrado à esquerda, na figura abaixo (o exemplo usa ESP ao invés de RSP, no modo i386):

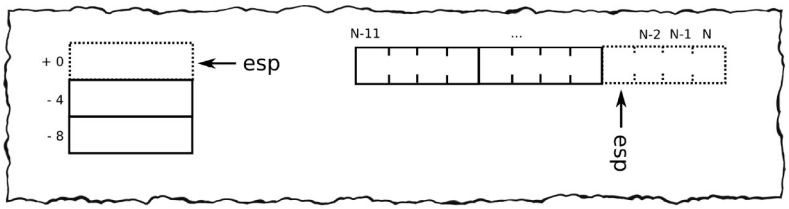


Figura 9: Ilustração do diagrama de uma pilha (i386)

A figura à direita mostra como a pilha realmente é na memória, onde N é um endereço de memória qualquer (o valor inicial de ESP). No modo i386, ESP sempre aponta para um bloco de 4 bytes (32 bits) de tamanho... Se estivermos lidando com o modo x86-64, então RSP sempre apontará para um bloco de 8 bytes (64 bits) de tamanho. Assim, o decremento de ESP ou RSP é feito de 4 em 4 ou de 8 em 8, dependendo se estamos no modo i386 ou x86-64, respectivamente.

No exemplo de diagrama acima, do lado esquerdo, eu indico o deslocamento que deve ser somado a ESP para obter o valor contido na posição correspondente da pilha. No desenho, ESP está apontando para o topo e, portanto, para obter o dado contido no topo basta usar ESP como ponteiro... Se quiséssemos obter o dado empilhado **antes** do topo teríamos que **somar** 4 (para pegar a posição “acima” do topo – essas posições eu não desenhei, acima).

A descrição das instruções, abaixo, considera o modo x86-64...

Para empilhar valores usa-se a instrução PUSH (de *empurrar*, em inglês). Ela decrementa RSP e depois grava o valor:

```
push eax       ; empilha o conteúdo de EAX (32 bits superiores serão zerados).
push 10        ; empilha o qword imediato 10.
```

É sempre bom lembrar: Primeiro RSP é decrementado e depois o valor é colocado na posição apontada por RSP!

A operação inversa, o desempilhamento, é feita pela instrução POP (sigla para *pull operation*. “Pull” é *puxar*). Essa instrução lerá o conteúdo apontado por RSP e depois incrementará RSP:

```
pop ecx        ; RCX = [RSP]; RSP = RSP + 8.
```

> Neste ponto você poderia perguntar se é possível empilhar valores com tamanhos diferentes. A resposta é **não**! PUSH e POP sempre manterão um alinhamento por *dword* (no caso do modo i386) ou *qword* (no modo x86-64).
>
> O alinhamento da pilha é importante por questões de performance. Embora o desenvolvedor possa lidar com o registrador ESP diretamente e modificá-lo de tal forma que possamos decrementá-lo e incrementá-lo um byte de cada vez, isso não é recomendável.
>
> No exemplo acima usei POP ECX de propósito para mostrar que RCX será atualizado depois da operação de puxar o valor da pilha, mesmo que especifiquemos ECX como destino (os bits superiores serão zerados!).

A pilha não serve apenas como um local para armazenamento temporário. Ela também é usada pelo processador para manter o registro das chamadas de funções. A instrução CALL salta para um

endereço definido em seu operando, mas também empilha o endereço da próxima instrução (depois de CALL) que seria executada. A rotina que foi "chamada" pode retornar à chamadora via instrução RET, que desempilhará o endereço que CALL colocou na pilha e saltará para ele:

```
    call myfunc
    cmp  eax,3       ; O endereço dessa instrução será
                     ; colocado na pilha pela instrução CALL.
    …

  myfunc:
    add  eax,eax
    ret              ; "ret" desempilha o endereço de retorno e salta para ele.
```

Como uma pilha é uma estrutura do tipo LIFO (*Last In First Out* – ou "o último que entra será o primeiro que sai"), múltiplas chamadas de função empilharão os correspondentes endereços de retorno em sequência. Sem esse recurso ficaria um pouco complicado escrever "funções" ou "sub-rotinas"[13]. Voltarei a falar sobre funções e o relacionamento delas com a pilha mais adiante, quando apresentarei o *prólogo* e *epílogo*, pedaços de código que o compilador cria para manter o registro das variáveis locais e parâmetros da função...

Registrador RFLAGS

O registrador RFLAGS é um registrador especial conhecido como *processor status register*. O nome diz tudo: Ele mantém bits que informam o "estado" do processador em qualquer momento.

> No modo x86-64 o nome do registrador é RFLAGS, mas apenas a porção dos 32 bits inferiores (EFLAGS) tem algum significado. Os demais bits são reservados e não devem ser modificados.

Um processador executa instruções em sequência, uma depois da outra. Cada instrução executada tem a possibilidade de mudar certos estados, certas características do processador. O registrador RFLAGS permite que seu programa tome conhecimento desses "estados".

Por exemplo: Depois de uma operação aritmética, podemos obter um estado de *overflow* (quando o valor resultante não cabe num registrador). Existe um bit em RFLAGS que nos informa isso... Da mesma forma, uma operação de subtração pode resultar num valor zero e, da mesma forma, existe um bit em RFLAGS que indicará isso... Normalmente estamos preocupados apenas com 4 bits contidos em RFLAGS: ZF (Zero Flag), CF (Carry Flag), SF (Signal Flag) e OF (Overflow Flag).

Sempre que uma operação aritmética (soma, subtração, multiplicação e divisão) ou lógica (AND, OR, XOR, NEG e NOT) resultar em zero, o bit ZF do registrador EFLAGS será setado. Da mesma forma, sempre que o bit MSB do resultado de uma operação aritmética aparecer setado, o bit SF será setado (indicando que o valor pode ser interpretado como negativo). O bit CF você já viu como funciona quando expliquei sobre *carry*, em adições e subtrações. Mas e o bit OF?

Lá atrás eu disse que o carry é uma espécie de *overflow*. E é, mas somente para valores **não sinalizados**. A diferença entre OF e CF é que o *overflow* aqui considera uma operação aritmética (e esse tipo de overflow ocorre **somente** com operações aritméticas!) **com sinal**.

O bit OF será setado se o resultado da operação não couber na representação **com sinal**... Por exemplo, para simplificar a explicação, façamos de conta que temos registradores de apenas 4 bits: Se fizermos uma adição do tipo "6 + 7", em binário, teremos 0*b*0110 + 0*b*0111 = 0*b*1101. Neste caso, depois da adição, teremos ZF=0 (o resultado **não** é zero), CF=0 (**não** houve "vai um" para

13 Mas não é impossível! Processadores como ARM, por exemplo, tendem a não usar a pilha para realizar chamadas de funções, mas explicar isso está fora do escopo deste livro!

fora dos 4 bits), OF=1 (o resultado correto, +13, **não** pode ser expresso em 4 bits considerando sinais) e SF=1 (o MSB do resultado está setado, dizendo que o valor pode ser interpretado como negativo). Num processador real esses flags funcionam exatamente assim, mas o tamanho dos operandos da operação devem ser considerados.

Esses 4 flags são muito importantes, como veremos no próximo capítulo, porque é através deles que poderemos tomar decisões... Em assembly não existem estruturas de controle como “if...then...else”. Precisaremos usar os flags para isso.

Outro bit de RFLAGS que pode ser de nosso interesse é o PF (*Parity Flag*). Esse flag nos indica se a quantidade de bits setados, no resultado da operação aritmética ou lógica, é par ou impar (1, para “par” e 0, para “impar”). Isso pode parecer inútil, mas o flag PF também é usado para indicar alguns tipos de erros de operações aritméticas usando ponto flutuante.

Mais um flag, e nesse caso é um dos que controla um comportamento do processador, é o flag de direção (DF, Direction Flag). Se estiver zerado, instruções de manipulação de blocos incrementarão ponteiros automaticamente, senão, decrementarão. Veremos esse flag mais tarde.

RFLAGS possui alguns bits extras, normalmente interessantes apenas para desenvolvedores de sistemas operacionais, como IF (Interrupt Flag), TF (Trap Flag), NT (Nested Task Flag), RF (Resume Flag), VM (Virtual 8086 Mode Flag), AC (Alignment Check Flag), IOPL (I/O Privilege Level), VIF e VIP (Virtual Interrupt e Virtual Interrupt Pending flags).

Retorno de valores booleanos em C e Assembly

Já reparou que em C não existem, de fato, valores booleanos? A convenção de *falso*, em C é um valor igual a zero, já *verdadeiro* é qualquer valor diferente de zero[14]. Quando escrevemos algo assim:

```
int check_if_equal(int a, int b) { return (a == b); }
```

O compilador, necessariamente, fará algo assim[15]:

```
; EDI = a, ESI = b
check_if_equal:
  xor eax,eax
  cmp edi,esi  ; Faz EDI - ESI
  jz  .L1      ; Se ZF == 1, salta para .L1...
  inc eax      ; ...senão, EAX = EAX + 1.
.L1:
  ; neste ponto EAX=0 se FALSO ou EAX=1 se VERDADEIRO.
  ret
```

A rotina que chamar essa função deverá verificar se EAX é 0 ou não para determinar a falsidade ou veracidade, respectivamente. Ou seja, o uso dos flags é restrito às instruções contidas na função e jamais são “exportados” ou, pelo menos, testados diretamente pela função chamadora.

Isso é diferente do que acontece com algumas funções do MS-DOS[16] ou da BIOS, por exemplo. Considere a função do MS-DOS que muda o diretório corrente:

```
pathname:  db "z:\\",0

…
  mov  ah,0x3b     ; Serviço 0x3B da INT 0x21 (chdir).
  mov  dx,pathname ; ds:dx aponta para pathname.
```

14 Isso acontece nas comparações... Na avaliação de uma expressão booleana o compilador retorna 0 e 1 para falso e verdadeiro, respectivaemnte!

15 Ok. Existe outra compilação possível, mostrarei mais tarde!

16 As funções da BIOS e do MS-DOS não são objeto de estudo desse livro, já que elas não são acessíveis no modo protegido – usado na maioria dos sistemas operacionais modernos.

```
  int  0x21        ; Executa o serviço.
  jc   .error      ; em caso de erro, CF=1, e pula para ".error", senão continua.
…
.error:
…
```

Aqui o flag CF é usado como um retorno booleano para determinar o sucesso ou falha da função do MS-DOS. Em outros serviços o flag ZF pode ser usado. Geralmente um desses dois flags é usado como retorno booleano de funções do MS-DOS ou BIOS, mas **não são usados** dessa forma em sistemas operacionais modernos ou em nossas rotinas escritas em C.

Considere a mesma função, presente tanto em compiladores para MS-DOS quanto para sistemas operacionais mais modernos:

```
int chdir(const char *);
```

No caso de C a função retorna 0 em caso de sucesso ou -1 em caso de falha...

Um pouco mais sobre endereços (usando ponteiros)

Já vimos o que é um endereço e, *en passant*, como usá-los em assembly. Se quisermos usar o conteúdo de um registrador como se fosse um ponteiro ou especificar um endereço numérico (ou via um símbolo), usamos a notação entre colchetes ('[' e ']'). Este é o operador de derreferência ou indireção do assembly, mas existem regras rígidas do que pode ir entre os colchetes: É necessário escrever um ponteiro que siga a regra "base + índice*escala + deslocamento".

Essa maneira de escrever um endereço tem 3 pedaços **opcionais** (onde um deles têm que estar presente!). A "base" é um registrador qualquer, usado como "endereço base". Pense nisso como o endereço de um array, em C: O nome da variável é o endereço da posição 0 do array.

O "índice" é um registrador que é usado como deslocamento a partir da base. Esse índice pode ser multiplicado por uma "escala" para que o endereço aponte para tipos como *byte*, *word*, *dword* ou *qword*. As escalas possíveis são 1, 2, 4 ou 8, correspondendo ao tamanho desses "tipos" (se for 1, não é necessário escrever a escala, por motivos óbvios).

O "deslocamento" é um valor numérico (nunca um registrador) e **com sinal**. É importante guardar essa informação: Esse "deslocamento", embora possa ser usado como "endereço" é, de fato, um deslocamento dentro de um segmento. Assim, todas as construções abaixo são válidas:

```
mov  eax,[0x0647340]   ; Endereço numérico (apenas o deslocamento).
mov  eax,[rbx]         ; rbx é usado como endereço base.
Mov  eax,[rbx-2]       ; rbx é o endereço-base e -2 é o deslocamento.
mov  eax,[rbx+rsi]     ; rbx é o endereço-base e rsi o índice.
mov  eax,[rbx+rdx+2]   ; rbx é o endereço-base, rdx o índice e 2 o deslocamento
mov  eax,[rcx+2*rbx+4] ; rcx é o endereço-base, rbx o índice, escalonado para word (2) e
                       ; 4 o deslocamento.
Mov  eax,[4*rdx]       ; Usa apenas o índice, escalonado por 4.
```

Em todos esses exemplos um dword é lido da memória e colocado em EAX, já que o registrador destino tem tamanho de um doubleword.

Em assembly raramente lidamos co endereços numéricos diretamente (a primeira instrução, no exemplo acima). Assim, se um deslocamento numérico precisa ser usado isoladamente, geralmente é especificado como o nome de um símbolo. Fique atento porque as instruções abaixo são completamente diferentes se você usa o NASM:

```
a:   dd   10

mov rdx,a      ; Coloca o endereço do símbolo a em edx.
mov rdx,[a]    ; Coloca o conteúdo apontado pelo símbolo a (10) em rdx.
```

E só para deixar fresquinho na sua mente: Essas referências à memória, através dos colchetes, sempre estão relacionadas ao segmento selecionado por DS. Se quiséssemos obter um dado contido num endereço de um outro segmento poderíamos indicar outro seletor assim:

```
mov  [cs:rdx],al    ; rdx aqui contém um endereço relativo a cs.
                    ; A instrução tenta gravar o conteúdo do registrador AL
                    ; no endereço dado por cs:edx.
```

Mas não espere que isso funcione muito bem. Por motivos de *proteção*, oferecida pelo seu sistema operacional e pelo processador, provavelmente você obterá um *segmentation fault* ou um *access violation* (no caso do Windows) na cara. Isso acontece porque os descritores de segmentos de código geralmente são marcados como "apenas para leitura" (*read only*).

Um pouco mais sobre seletores default e a derreferenciação de ponteiros

Vimos que, por *default*, o seletor DS é usado em referências à memória, a não ser que uma instrução de manipulação blocos seja executada e, neste caso, o seletor ES também é usado.

Se os registradores RSP ou RBP forem usados como **endereço base** num ponteiro, então o seletor SS é usado automaticamente. Repare que falei "endereço base". Se usarmos RBP ou RSP como índice, então o seletor DS continuará sendo o default:

```
mov  eax,[rbp+rsi]       ; Usa o seletor SS. RBP é o endereço base.
mov  eax,[rsi+rbp]       ; Usa o seletor DS. RSI é o endereço base.
```

O NASM, e suspeito que a maioria dos assemblers que usam o sabor Intel, **sempre** considerará o primeiro registrador como sendo a "base" numa notação de ponteiro. Em teoria, ambas as instruções fazem a mesma coisa, já que o ponteiro é a adição de RBP com RSI, mas elas acessam segmentos diferentes.

A área "doce" da pilha

Ainda sobre os modos de endereçamento e a derreferência de ponteiros, se usarmos deslocamentos maiores que 127 e menores que -128, a instrução compilada será maior que o necessário. Isso acontece porque existem μ_{op}s especiais para deslocamentos do tamanho de um byte. Dê uma olhada:

```
 0: 8B 46 01            mov eax,[rsi+1]
 3: 8B 46 FF            mov eax,[rsi-1]
 6: 8B 86 80 00 00 00   mov eax,[rsi+128]
12: 8B 86 7F FF FF FF   mov eax,[rsi-129]
```

Isso significa que as instruções que usam deslocamento maior que um byte (sinalizado), além de consumirem o dobro do espaço no cache L1i, poderão gastar 1 ciclo adicional de clock para serem executadas.

Isso é uma *feature* dos processadores Intel: Quanto mais perto do endereço base estiver o dado, menor ficará a instrução.

Diferenças entre endereços de 32 bits e 64 bits

No modo i386 um endereço tem sempre 32 bits de tamanho e é um valor não sinalizado. No modo x86-64 a coisa é um pouco mais complicada. Nesse modo um endereço tem 64 bits de tamanho, mas nem todos os bits são usados.

Alguns processadores da família Intel usam apenas 48 dos 64 bits e os demais bits (do bit 48 até o MSB) serão uma cópia do bit 47. Outros, mais modernos, usam 52 dos 64 bits e copiam o bit 51 para os bits superiores... Esse modo de especificar um endereço é chamado de **endereçamento**

canônico e deve ser sempre observado, caso contrário o processador gerará um erro conhecido como *General Protection Fault*, ou seja, o famoso *segmentation fault*.

Na prática, você não tem que se preocupar muito com isso, já que o compilador e o sistema operacional é quem vão se preocupar com a atribuição de endereços.

Capítulo 2 - A linguagem Assembly

O que é uma "linguagem" de programação? Trata-se, como em qualquer outra linguagem, de uma série de regras gramaticais que evoluíram de forma que nós, humanos, possamos trocar informações. Ou seja, para nos fazermos compreender usamos um monte de grunhidos regidos por um conjunto de regras que aceitamos, de comum acordo. Este livro, por exemplo, é todo escrito em português e tenta seguir um conjunto de regras para que você entenda o que eu, o autor, estou tentando comunicar.

Seu cérebro não trabalha com "português". As ideias que entram na sua cabeça são transformadas em algum esquema, não muito bem compreendido, na forma de conceitos. A linguagem cerebral é totalmente diferente da linguagem falada ou escrita. E, por analogia, podemos aplicar o mesmo princípio para **linguagem de máquina** (que só o processador entende) e outras linguagens como assembly e C.

Muitas **linguagens de programação** oferecem facilidades intrínsecas como operações para lidar com strings, funções para imprimi-las e assim por diante. Linguagens como assembly não tem muitas dessas facilidades porque não é uma **linguagem estruturada**. O termo "estruturada" aqui significa que o foco não está em estruturas de dados, mas sim em instruções simples e de execução extremamente rápidas.

Já linguagens como C são semiestruturadas. "C" permite a declaração de estruturas de dados mais complexas, mas do mesmo jeito que assembly, não existem facilidades que permitam a linguagem lidar com strings ou arrays associativos, por exemplo. Tudo o que você faz para acessar periféricos, por exemplo, tem que ser feito por rotinas especializadas, disponíveis em bibliotecas padronizadas (a *libc*, por exemplo).

No fim das contas, a diferença entre C e Assembly está apenas no fato de que a última está mais próxima da linguagem de máquina do que a primeira. Abaixo temos um exemplo do clássico código "hello, world" em ambas as linguagens:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  puts("Hello, world!");

  return 0;
}
-----%<----- corte aqui -----%<-----
  ; A mesma rotina, em Assembly!
  bits 64

  section .data

; Sequência de bytes colocados no segmento de dados.
str:  db "Hello, world!", 0

  section .text
  ; Declara o símbolo 'puts' como externo.
  ; (i.e: "está em outro lugar, não aqui!")
  extern puts

  ; Exporta o símbolo 'main' para o linker.
  global main
```

```
; Ponto de entrada da função main().
main:
  mov rdi,str
  call puts
  xor eax,eax
  ret
```

Na listagem em assembly, as linhas depois de “main:” são **mnemônicos** de instruções que depois de compiladas o processador entenderá.

> Do dicionário:
>
> **mnemônico** (adj.)
> 1. Relativo à memória;
> 2. Que facilmente se retém na memória.

Na definição dos sinônimos, no quadro acima, “memória” quer dizer aquilo que existe na pasta cinzenta que localiza-se entre suas orelhas (e não estou falando daquela que você tira com cotonete!). Trata-se de um recurso de memorização, de facilitar o entendimento. Veja de novo as linhas contendo as instruções de “main”, da última listagem: A palavra “mov” significa “move” (mover, em inglês, o que mais poderia ser?). “call” e “ret” são os mnemônicos para “chamar” e “retornar”.

Assembly é simples assim! As outras linhas são diretivas, dicas, para o compilador...

Informando o modo de operação para o compilador

A diretiva “bits 64” diz ao compilador que o que segue é destinado ao modo x86-64. Se quiséssemos trabalhar com o modo i386 basta usar 32. Esses valores, é claro, correspondem ao número de bits suportados nos registradores de cada modo.

O modo 8086 também está disponível, bastando usar “bits 16”.

Especificando os “segmentos” da memória usados pelo programa

A diretiva “section” diz ao compilador com qual segmento de memória estamos lidando. Existe um padrão de nomeação para essas “sessões” (que é só um outro nome para “segmentos”):

Tipo de segmento	**Nome da *section***
Código	.text
Dados não inicializados	.bss
Dados inicializados	.data
Dados constantes (inalteráveis)	.rodata

Tabela 1: Nomes de segmentos

Assim, quando dizemos “section .text”, na listagem em assembly, o que vem embaixo trata-se de informações que serão colocadas no segmento de código. No exemplo anterior, os dados que compõem a string “Hello, world” são colocados no segmento de dados (.data).

Segmentos de dados diferentes têm usos diferentes. O segmento “.data” é destinado às variáveis pré-inicializadas. O segmento “.bss” é destinado para variáveis cujo espaço é reservado, mas não são inicializadas. Já o segmento “.rodata” é a mesma coisa que o “.data”, mas é apenas para leitura (os dados contidos ali não podem ser modificados pelo seu programa).

No caso da nossa “string”, usei a diretiva DB (Define Byte) que é usada para declarar bytes individuais. Como uma string é definida, em C, como uma sequência de bytes terminada por '\0', a diretiva DB poderia também ser escrita assim:

```
str:  db  0x48, 0x65, 0x6c, 0x6c, 0x6f, 0x2c, 0x20, 0x77
      db  0x6f, 0x72, 0x6c, 0x64, 0x00
```

Mas, assim como em C, a linguagem assembly permite usarmos o atalho de declararmos uma sequência de bytes entre aspas... Diferente de C, assembly não coloca, automaticamente, um '\0' no final dessa sequência e também não diferencia entre aspas duplas e aspas simples.

Declarando símbolos

Lembra dos endereços de memória? Seria meio chato se tivéssemos que usar um número de 52 bits (no caso do modo x86-64) toda vez que quiséssemos chamar uma função ou acessar uma variável, não? Para evitar isso linguagens de programação nos permitem criar “apelidos” para endereços. Esses apelidos são chamados de “símbolos” ou *labels* (rótulos).

No exemplo de código em assembly anterior, temos a declaração “global main” que diz ao compilador para exportar o símbolo “main”. O próprio símbolo “main” só é declarado depois (mas, poderia ser antes, tanto faz)... É aquela linha “main:” no código...

Note que temos também um símbolo “str” que, neste caso, não é exportado e, portanto, é visível apenas pelo módulo onde foi declarado.

Assembly **não é** linguagem de máquina e portanto não precisamos saber precisamente qual será o endereço atribuído aos símbolos. Aliás, nem o compilador sabe. O *linker* é que tratará disso e, mesmo assim, ele atribuirá um endereço “relativo”... Somente quando a imagem binária contida num arquivo executável for carregada na memória, pelo sistema operacional, é que um endereço será atribuído a cada um desses símbolos[17].

Diferentes sabores de assembly

Existem diferentes “linguagens assembly”. Cada fabricante de processador tem a sua e cada compilador também tem lá suas variações. Até mesmo na mesmíssima plataforma podemos ter sabores diferentes... No caso dos modos i386 e x86-64 temos, pelo menos, dois sabores: Intel e AT&T.

Adianto que eu prefiro o sabor Intel e o uso em todo o livro. Acho mais simples e mais fácil de encontrar referências de consulta para as instruções, mas ambientes como Linux e o GCC usam o sabor AT&T por *default*. A história do padrão AT&T do assembly confunde-se com a usada nos microprocessadores da família Motorola MC68000 (e, suspeito, com um “processador” ainda mais antigo, o do PDP-11). Se você observar a sintaxe do sabor AT&T verá muitas semelhanças com o MC68000.

Como exemplo desses dois sabores, eis duas instruções idênticas:

```
mov  dword [rsi+rdx*2+4],0    ; sabor Intel.
movl $0,4(%rsi,%rdx,2)        ; sabor AT&T
```

Considero mais fácil entender que o ponteiro é calculado como “RSI+RDX*2+4”, na sintaxe da Intel, do que na sintaxe da AT&T. Na sintaxe intel temos que usar o modificador de tamanho *dword* por causa do valor imediato zero, mas no sabor AT&T o próprio mnemônico precisa sofrer uma

17 Como isso realmente acontece não vem ao caso. Mas, para entender o processo é interessante que você procure pelas especificações do seu sistema operacional. Procure por “Portable Executable”, no caso do Windows, ou por “Executable and Linkable Format”, no caso dos sistemas POSIX.

pequena alteração (MOV vira MOVL, onde a letra “L” depois de MOV significa “long”). Para mover valores de outros tamanhos temos que modificar o mnemônico usando os sufixos B, W ou Q para *byte*, *word* ou *qword*, respectivamente.

Existem outras diferenças: Registradores normalmente são precedidos de um “%” no sabor AT&T. Isso nos dá a vantagem de permitir a criação de símbolos que tenham o mesmo nome de registradores (o que não é possível no sabor Intel). Ainda, lembra-se do caso onde usamos um registrador que não seja RSP ou RBP como base e o seletor default é DS? No caso da notação AT&T isso aparece sem ambiguidades. O endereçamento é sempre feito como *offset(base,indice,multiplicador)*, onde “base” é obrigatório e todo o resto é opcional, mas a ordem é garantida. No modelo Intel [base+indice] ou [indice+base] dão o mesmo resultado e nada dizem, além da posição relativa, quem é a base e quem é o índice...

Sabores de compiladores

Além desses dois, cada compilador implementa suas próprias maneiras de lidar com mnemônicos, labels e diretivas. Veja dois exemplos: Um escrito no MASM (*Microsoft Macro Assembler*), e outro escrito para o NASM (*Netwide Assembler*) – ambos para 32 bits.

```
; Código de f() para o Microsoft Macro Assembler (MASM):
_TEXT segment para public use32

public f
f proc near
  mov eax,[esp+4]
  add eax,eax
  ret
f endp

_TEXT ends

end
----- corte aqui -----
  ; O mesmo código de f() para o Netwide Assembler (NASM):
  bits 32
  section .text

  global f
f:
  mov eax,[esp+4]
  add eax,eax
  ret
```

Todos os exemplos citados neste livro referem-se ao compilador NASM (*Netwide Assembler*). Se você quiser usar outros compiladores (MASM, YASM, FASM, etc) terá que obter os manuais do fabricante e se acostumar com a sintaxe deles. Não vou discuti-los aqui.

A linguagem assembly

Um programa escrito em assembly é composto de instruções simples, arranjadas linha a linha (uma linha tem uma única instrução). Uma instrução em assembly pode ser escrita com a seguinte estrutura num código-fonte:

```
label:  instrução  op_dest,op_fonte,...   ; comentário
```

Já vimos o que é um *label* ou símbolo. É um apelido para um endereço...

A instrução, como vimos antes, é um *mnemônico*, uma abreviação que indica o que ela faz. Essa instrução pode ser seguida argumentos ou operandos. A maioria das instruções têm 2 operandos. O primeiro é o “operando destino”, onde o resultado da instrução será colocado e os demais são operando “fonte”, de onde argumentos adicionais são obtidos. Eis um exemplo:

```
f:    mov eax,ebx     ; eax ← ebx
```

Algumas instruções tomam apenas um único operando, outras tomam mais que dois deles e, ainda, outras não os usam. Pior: Algumas instruções podem ter 1, 2 ou 3 argumentos, dependendo do uso (é o caso das instruções de multiplicação):

```
imul ebx        ; edx:eax ← eax * ebx
imul ecx,ebx    ; ecx ← ecx * ebx
imul edx,eax,3  ; edx ← eax * 3
```

A notação EDX:EAX significa somente que esse par de registradores será usado como destino de 64 bits (os 32 bits LSB em EAX e os MSB em EDX).

Nas operações lógicas e aritméticas o operando destino é também usando como fonte. Ao realizar uma adição, por exemplo:

```
add eax,ecx     ; eax = eax + ecx
```

Note a semelhança dessa técnica com operações como “+=”, em C.

A pergunta agora deveria ser: “Mas, como vou saber quais argumentos posso usar?”. Bem... consultando os manuais da Intel ou da AMD que listam as instruções! Para facilitar a vida da pessoa que vai consultá-los, esses manuais adotam um esquema que permite saber quais são os “tipos” de operandos válidos para uma instrução. Eles usam uma nomenclatura composta de uma sigla e de um valor:

Sigla	**Significado**
imm	Valor imediato.
r	Registrador
m	Memória
r/m	Registrador ou memória
xmm	Registrador SSE
ymm	Registrador AVX
zmm	Registrador AVX-512

Tabela 2: Siglas usadas pela Intel

Uma sigla tipo “imm8” significa “valor imediato de 8 bits”. “Imediato”, aqui significa que esse operando não é um registrador ou um ponteiro, ele é apenas um número.

A sigla “r/m32” significa “registrador ou memória (ponteiro) de 32 bits”. A instrução MOV, por exemplo, aceita um bocado de combinações entre esses argumentos. Eis algumas delas:

```
mov r/m8,imm8
mov r/m16,imm16
mov r/m32,imm32
mov r/m64,imm64
mov r8,r/m8
mov r16,r/m16
…
mov r/m8,r8
mov r/m16,r16
…
```

A lista não é tão grande assim, eu só não quero encher esse livro de listagens desnecessárias. Os possíveis argumentos para MOV podem ser vistos nos manuais de *instruction sets* que citei... Essas listas também vão te mostrar outra coisa: Não é possível construir instruções onde ambos os

operandos sejam referências à memória! Isso aqui **não** existe:

```
mov byte [eax], byte [ebx]       ; Isso não pode!
```

Tipos de instruções

Existem, essencialmente, 3 tipos:

1. Instruções de movimentação;
2. Instruções aritméticas e lógicas;
3. Instruções de controle de fluxo do programa.
4. Outras, especializadas.

A instrução MOV é um bom exemplo do primeiro tipo (o nome entrega, né?) e existem outras mais especializadas. Já o segundo tipo são todas as instruções que lidam com as operações aritméticas (as elementares e as mais especializadas, no caso de ponto flutuante), bem como operações lógicas (OR, AND, XOR, NOT, NEG). E essas instruções são aquelas que afetam os flags ZF, CF, SF, OF e PF.

O terceiro tipo é aquele que permite que você construa um programa! O processador executa instruções em sequência, uma depois da outra. Só que precisamos criar loops, saltos “condicionais” (if...then...else) e saltos “incondicionais” (goto). Para isso temos instruções como JMP (“jump”, dã!), CALL, RET e algumas outras que levam os flags em consideração.

Existem outras instruções destinam-se ao *setup* do processador ou recursos especiais. Essas instruções especiais normalmente não estão disponíveis para seus programas no *userspace* (muitas delas executam apenas no nível do kernel). Existem, também, instruções e registradores especiais que não são acessíveis fora do kernel.

Usando RFLAGS para tomar decisões

No capítulo anterior vimos que os flags podem ser usados para tomada de decisões, já que construções como *if..then..else*, por exemplo, não existem em assembly. Eis como isso é feito...

Vamos começar com uma construção bem simples, em C, e veremos como o compilador traduz isso:

```
…
if (x == 2)
  x = 0;
x++;
…
```

O código em assembly desse fragmento, considerando que 'x' seja o registrador EDX, ficaria mais ou menos assim:

```
      …
      cmp  edx,2            ; Compara EDX com 2 (Faz EDX – 2).
      jne  .L1              ; Se não forem iguais, ZF == 0, salta para .L1 ...
      xor  edx,edx          ; … senão, faz EDX = 0;
.L1:  inc  edx              ; EDX = EDX + 1;
      …
```

A instrução CMP compara o operador do lado esquerdo com o do lado direito. Se você pensar bem, uma comparação é uma subtração disfarçada. Sempre podemos substituir a pergunta “x é igual a 2?” por “x menos 2 é igual a zero?” e é exatamente isso que o processador faz. Essa instrução faz exatamente a mesma coisa que uma instrução SUB faria (subtrair um operando do outro), mas não

coloca o resultado da operação em lugar algum a não ser nos flags.

A instrução seguinte é um salto “condicional”. JNZ é o mnemônico para “*Jump if Not Zero*”. O operando dessa instrução é o endereço para onde o salto será feito se a condição for satisfeita.

> Um salto condicional sempre é feito em relação a próxima instrução. O “endereço” que segue o mnemônico é traduzido para um deslocamento em relação ao endereço da próxima instrução. No exemplo acima, a instrução “MOV EDX,0” será codificada em 6 bytes. Assim, a instrução JNZ será codificada como sendo “JNZ +6”, ou seja, “pule os próximo 6 bytes se ZF=0”.
>
> Diferente dos saltos condicionais, instruções como JMP e CALL são saltos **incondicionais**, onde o salto sempre é feito, isto é, independem de testes contra os flags.
>
> Assim como JNZ, JMP e CALL podem usar endereçamento relativo e o compilador assembly tende a usar essa técnica, mas essas instruções também permitem o uso de endereços absolutos. Endereços absolutos, no i386, terão 32 bits de tamanho e, no x86-64, 64 bits.

Repare que o compilador C resolveu usar uma lógica contrária, em assembly, do que estava na listagem original. Em C comparamos a igualdade de 'x' com 2, mas em assembly o compilador resolveu testar a desigualdade (NOT Zero!) para fazer um *bypass* na próxima instrução, pulando para o label “.L1” se ZF == 0.

Comparando desigualdades

Se queremos comparar a igualdade só precisamos lidar com o flag ZF, que indica se o resultado de uma operação aritmética ou lógica resultou em zero... Mas e quanto as outras comparações?

Ao subtrair um número **sem sinal** maior de um menor (a – b, onde a < b) teremos o flag CF setado, indicando um “empréstimo” ou *borrow*. Com isso, se substituirmos a comparação para “(x < 2)”, no exemplo acima, só precisaremos substituir o salto condicional de JNZ para JNC (*Jump if Not Carry*).

Se estivermos trabalhando com inteiros **com sinal** (*int*), teremos que usar os flags OF e SF... Ao invés de usarmos JNC poderíamos usar JNO (*Jump if Not Overflow*) ou JNS (*Jump if Not Signaled*)... Eu disse “poderíamos” porque há outra complicação envolvendo sinais: Para comparar valores sinalizados **temos que usar ambos os flags**:

Op_A	Op_B	Op_A - Op_B	Flags	
			SF	OF
1	2	-1	1	0
-1	2	-3	1	0
-2	-1	-1	1	0

Tabela 3: Comparação dos operandos A e B para os casos “comuns”.

A tabela acima mostra o que acontece numa instrução genérica do tipo “CMP A,B”. As subtrações acima não causam *overflow* (o resultado, sinalizado, cabe num registrador!). Então, de acordo com a tabela acima você poderia inferir que basta usar o flag de sinal (SF) para determinar se A é menor que B, mas no caso de acontecer um *overflow* o bit de sinal será invertido (faça as contas!). Ou seja,

se quisermos comparar se “A < B”, então temos que testar se OF ≠ SF.

A coisa complica um pouco mais se quisermos comparar se “A > B”... Claro que a lógica inverte e deveremos comparar se OF=SF para comparações do tipo “maior ou igual”. Mas, para comparações do tipo “maior que” temos que levar em consideração se o flag ZF também está zerado.

Felizmente, para comparações de inteiros sinalizados temos “atalhos”. Instruções de salto condicional especializadas: JL (*Jump if Less than*), JG (*Jump if Greater than*), JLE (*Jump if Less than or Equal*) e JGE (*Jump if Greater than or Equal*). E para comparar os valores **não** sinalizados podemos usar JB, JA, JBE e JAE, onde 'B' significa *Below* (abaixo) e 'A', *Above* (acima).

Um lembrete para o uso dessas instruções é que *less* é “menor” (considera sinal) e *below* é “abaixo” (não considera sinal). A instrução JL é feita se OF≠SF, enquanto JG é feita se OF=SF **e** ZF=0. E, assim como instruções JZ tem seu equivalente em lógica contrária, JNZ, todas as demais também têm... A instrução JL, por exemplo, é a mesma coisa que JNGE, assim como JNL é a mesma coisa que JGE. E JNG é a mesma coisa que JLE.

Para manter a coerência na nomenclatura dos mnemônicos a Intel resolveu nomear as instruções JZ e JNZ, também, como JE e JNE. Ou seja, os quatro mnemônicos existem, onde JZ e JE são a mesma instrução (assim como JNZ e JNE).

> Todo “if” é feito sempre em relação aos flags. É preciso alterar o status do processador para depois usar um salto “condicional”.

O uso dos flags não está restrito apenas aos saltos condicionais. Existem duas outras instruções condicionais que nada têm haver com saltos. Movimentações condicionais (*Conditional Moves* ou CMOVs) e Ajustes condicionais (*Set On Condition*):

```
cmovne eax,ebx    ; Move ebx para eax se ZF=0.
setge  al         ; Faz AL=1 se SF=OF, caso contrário AL=0.
```

Ambas as instruções são bastante úteis quando falarmos de “previsões de saltos”...

Loops

Qualquer estudante de programação sabe que existem duas estruturas básicas de loops: *while* e *do..while*. A diferença está na posição onde o critério do controle do loop é colocado. No caso do loop do tipo *while* o teste é feito logo no início:

```
while (critério)
{
  // faz algo aqui.
}
```

No caso do tipo *do..while*, é feito no final:

```
do {
  // faz algo aqui.
} while (critério);
```

Numa listagem assembly gerada por um compilador como o GCC, geralmente ambos os estilos de loop têm seus testes realizados **no final** da rotina. Considere o seguinte fragmento de código:

```
x = 0;
while (x < 10)
  x++;
…
```

A provável listagem equivalente, em assembly, considerando que 'x' seja o registrador EDX, é essa:

```
    xor edx,edx     ; edx ← 0
    jmp .L1
.L2:
    inc edx         ; x++;
.L1:
    cmp edx,10      ; if (x < 10)
    jl  .L2         ;   goto 2
    …
```

Pode parecer uma perda de tempo e espaço realizar esse JMP logo depois de zerarmos EDX, mas existem três motivos pelo qual essa rotina é melhor que a mais óbvia mostrada à seguir:

```
    xor edx,edx
.L1:
    cmp edx,10
    jge .L2
    inc  edx
    jmp .L1
.L2:
    …
```

- Nessa segunda rotina a instrução JGE não realizará o salto em 9 iterações, mas o realiza na 10ª. Nas 9 iterações que ela "não salta" tempo é gasto à toa;
- Um salto incondicional é executado 9 vezes.

E o terceiro motivo é mais complicado de explicar. Uma instrução de salto condicional pode gastar 1 ou 2 ciclos de máquina para ser executado. Ele gastará 1 ciclo se a condição **não** for satisfeita (por exemplo, se ZF == 0 e a instrução for JZ), mas gastará 2 ciclos se realizar o salto e o processador estiver esperando que o salto não seja feito.

Estranho esse negócio de "esperar que não seja feito o salto", não é? Existe um troço chamado *branch prediction*[18] (ou "previsão de salto"), onde o processador mantém uma contagem da direção em que saltos condicionais já foram feitos... Se tivermos mais saltos para trás então, se a condição for satisfeita, provavelmente a instrução não gastará esse ciclo adicional se ela saltar também para trás... Da mesma forma, se um salto em uma certa direção não foi feito, o processador passará a esperar que os demais saltos na mesma direção não sejam feitos!

Compare a rotina gerada pelo GCC com a última: Apenas um salto incondicional e 9 saltos condicionais que serão feitos, onde apenas o último não o é... Se o compilador C segue a regra de coloca a comparação no final do loop isso significa que a maioria dos saltos condicionais serão feitos "para trás" **na maioria das vezes e em todo o seu código**, acelerando os saltos em 1 ciclo. Assim, a última rotina é claramente mais rápida que a "mais intuitiva".

A outra vantagem de colocar a comparação no final do loop é que, no caso de *do..while*, basta retirar o salto incondicional. Eis como fica:

```
/* Fragmento em C */
x = 0;
do { x++; } while (x < 10);
…
----- corte aqui -----
; Fragmento em assembly
    xor edx,edx
    ; cadê o jmp que tava aqui?
.L1:
    inc edx
    cmp edx,10
    jl  .L1
    …
```

18 Mais sobre *branch prediction* no capítulo sobre "performance".

Capítulo 3 – Assembly e outras linguagens

Desenvolver aplicações inteiras em assembly é perfeitamente viável, mas vá por mim... É uma daquelas tarefas que você não vai querer fazer! Dá um trabalho dos diabos! E não há grandes vantagens... Compiladores C e até mesmo de linguagens como PASCAL, fazem um bom trabalho ao traduzir um código-fonte de uma linguagem mais abstrata para a linguagem de máquina.

Acontece que seu código pode ter pedaços que precisem mesmo ser feitos em assembly. Pedaços críticos que precisam sofrer um ajuste fino que seria muito difícil de obter através das regras de otimização de um compilador de nível mais alto. Para esses casos existem métodos documentados de interfacear suas rotinas em assembly para serem usadas junto com códigos escritos em C ou PASCAL (ou com algum outro compilador de linguagens decentes!). É sobre isso que falarei neste capítulo.

Portabilidade é um problema!

Existem pelo menos dois níveis de portabilidade. O primeiro, e mais evidente, é relacionado ao sistema operacional... Sistemas operacionais diferentes têm métodos diferentes de lidar com os registradores, flags e pilha. Tanto em suas rotinas internas quanto através das interfaces que disponibilizam para o *userspace*. Sem contar que eles disponibilizam rotinas básicas diferentes, acessíveis por meios diferentes.

A maioria das pessoas que conheceram meu antigo curso de assembly viram o jeito MS-DOS de usar rotinas do sistema operacional. No Windows (em modo protegido e preemptivo) a coisa é bem diferente. Ainda, se você lida com Linux, essas rotinas básicas são chamadas de outra maneira (que diferem, inclusive, de acordo com a arquitetura usada pelo sistema... i386 faz de um jeito e x86-64, de outro).

Além de métodos diferentes de acesso às rotinas básicas (que chamarei, daqui para frente, de *syscalls* – abreviação de *System Calls*), cada compilador usa um meio diferente de "passar parâmetros" e retornar os resultados de funções. Um compilador C faz de um jeito, um compilador PASCAL faz de outro totalmente diferente...

Então, quando te dizem que assembly não é portável, essas pessoas têm toda a razão. Mas, eis um detalhe: A mesma coisa pode ser dita de um programa escrito em C, C++, PASCAL e até mesmo JAVA! Sim, JAVA! Em teoria o Java foi criado para ser multiplataforma, bastando usar uma camada de abstração, uma "virtual machine" que compatibilizaria o ambiente operacional onde seu programa roda. Na prática isso **não** funciona tão bem assim. Já vi sistemas desenvolvidos em JAVA, para PCs, rodando Windows, que não rodavam de jeito nenhum numa máquina da Sun Microsystems, rodando Solaris...

A manutenção de problemas de portabilidade deve ser de responsabilidade exclusiva do desenvolvedor, não de uma "camada de abstração" ou de uma linguagem... Dito isso, vamos ao tópico do capítulo...

Convenções de chamada

Em outro livro expliquei o que vem a ser uma "convenção de chamada". Aqui, vou fazer o mesmo,

mas com um pouco mais de detalhes, já que estamos lidando com duas arquiteturas: i386 e x86-64.

Em linguagens como C, por exemplo, seu código é escrito em termos de funções. Cada função é um bloco de afirmações (statements) que executam instruções a partir de argumentos (ou parâmetros) e pode haver o retorno de algum valor. De outra forma, uma função tem uma lista de variáveis onde você coloca valores que serão passados para ela e um valor é retornado, como se fosse uma função matemática. Outras linguagens, como PASCAL, operam da mesma maneira.

Para que isso funcione, a maneira de informar esses parâmetros para uma função e obter o valor de retorno têm que ser **padronizada**. Temos que seguir uma **convenção** "universal"... Dai o nome "convenção de chamada [de uma função]".

Vou mostrar à seguir as convenções de chamada para i386 e x86-64 usadas nos ambientes POSIX (Linux, FreeBSD, OS/X, …). Essas convenções não são atreladas a uma linguagem específica, embora algum compilador possa usar sua própria convenção, o padrão POSIX determina um padrão universal *de facto* para ambientes UNIX.

No ambiente Windows isso pode variar. Embora a Win32 e Win64 API usem uma convenção de chamada específica (*stdcall*), existem muitos compiladores que usam convenções diferentes. O Delphi, por exemplo, usa convenções diferentes das usadas pelo Visual Studio. Para ver as diferenças do padrão usado no Win64 para o usado no POSIX, para x86-64, consulte meu outro livro[19].

A convenção "cdecl" e o modo i386

A convenção de chamada *default*, usada num compilador C, é a *cdecl*. É uma abreviação óbvia de "*C declaration*". Ela usa a pilha para passar os argumentos da função. Todos os parâmetros têm que ser **empilhados de trás para frente** (do último parâmetro para o primeiro) antes que a função seja chamada.

Outro detalhe é que o desempilhamento desses parâmetros tem que ser feito pela função que as empilhou, não pela função que os usa!

Quando a função retorna, podemos obter o resultado através do registrador EAX, ou um subconjunto dele, ou ainda, pelo par EDX:EAX (no caso do tipo de retorno seja um *long long*).

Como os valores dos parâmetros são empilhados, para obtermos os valores passados para a função teremos que usar o registrador ESP para localizá-los. Eis um exemplo de função simples e a listagem assembly equivalente:

```
/* Função em C */
int mult2(int a)
{
  return a + a;
}
----- corte aqui -----
; Função em asm
bits 32
section .text

global mult2
mult2:
  mov eax,[esp+4]     ; Pega 'a' da pilha.
  add eax,eax
  ret
```

Por que esse "[esp+4]"? Considere o uso da função *mult2*. Quando ela é chamada o compilador fará algo assim:

19 C & Assembly para arquittura x86-64.

```
; O fragmento em C equivalente seria:
;
;    y = mult2(10);
;
push dword 10
call mult2
add  esp,4     ; “Limpa” a pilha.
mov  [y],eax
…
```

A instrução PUSH “empurra” o *dword* 10 para a pilha e a instrução CALL “empurra” o endereço de retorno (a próxima instrução depois de CALL)... Com isso, temos 2 valores empilhados, onde ESP aponta para o topo da pilha que contém o endereço de retorno.

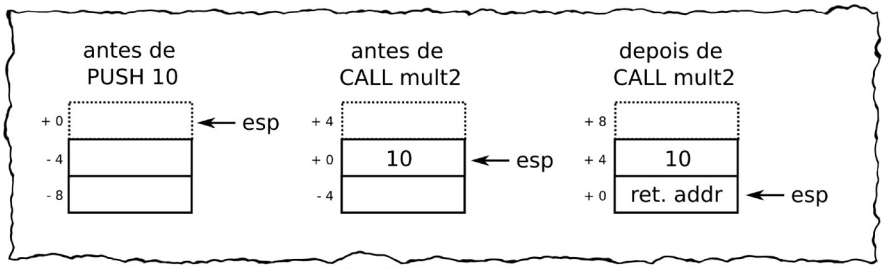


Figura 1: Estado da pilha antes e depois de CALL mult2

Ao entrar na rotina *mult2* o registrador ESP estará apontando para o endereço de retorno, então, precisamos adicionar 4 para obter o endereço da pilha onde o parâmetro *a* está! Assim, a primeira instrução da rotina *mult2*, em assembly, obtém da pilha o valor do parâmetro e o coloca em EAX. Daí é só manipulá-lo e retornar para o chamador...

Ao retornar para a função que chamou *mult2* o compilador limpará a pilha livrando-se do valor previamente empilhado. Uma maneira de fazer isso é executando instruções POP, outra é incrementando ESP.

Registradores que devem ser preservados entre chamadas de funções

Na convenção *cdecl* os registradores EBX e EBP devem **sempre** ser preservados entre chamadas. Ambos são usados pela *libc* e pelos códigos de inicialização e manutenção do programa que o GCC insere.

Você pode usá-los nas suas funções, mas o valor que eles tinham antes de sua função ser executada têm que ser os mesmos que terão quando sua função retornar. Todos os outros GPRs podem ser alterados à vontade.

Armazenamento temporário “local” na pilha

Se fornecermos um deslocamento positivo, usando ESP como endereço base, podemos obter os valores dos parâmetros passados para uma função. O empilhamento dos argumentos antes de uma chamada é coerente com a semântica de chamadas de funções, em C, já que as variáveis dos parâmetros são cópias dos argumentos originais.

Mas também podemos fornecer deslocamentos negativos e usarmos o espaço onde “não tem” valores empilhados para armazenamento temporário. Esse armazenamento temporário podem ser variáveis temporárias usadas internamente pelo compilador, para a função, ou “variáveis locais”, se o compilador escolher não mantê-las em registradores. Manter variáveis temporárias, no domínio da função, em registradores é parte da prioridade do compilador: Quanto menos acesso à memória, mais rápida será a função...

Nesse esquema de usar espaço “além do topo” da pilha como espaço temporário é comum vermos,

no código em assembly de uma função em C, dois fragmentos chamados de *prólogo* e *epílogo*. O prólogo é um conjunto de 3 instruções que “reserva” esse espaço temporário na pilha e reajusta ESP . O epílogo são duas instruções que desfazem essa “reserva”.

Eis um exemplo de código em C e seu equivalente em assembly, usando prólogo e epílogo:

```
extern int c;

int mult2addc(int x)
{
  int y;

  y = x + x;
  return y + c;
}
----- corte aqui -----
mult2addc:
  ; Prólogo
  push ebp              ; EBP é um daqueles registradores que não devem ser
                        ; alterados entre chamadas!
  mov  ebp,esp          ; Guarda, em EBP o “topo” real da pilha.
  sub  esp,4            ; Reserva espaço para 'y'.
                        ; ESP aponta para o novo “topo”, além do atual.

  ; Nossa rotina
  mov  eax,[ebp+8]      ; Pega o parâmetro 'x'.
  add  eax,eax
  mov  [ebp-4],eax      ; 'y' é a variável temporária, alocada na pilha!


  mov  eax,[c]
  add  eax,[ebp-4]      ; EAX = c + y;

  ; Epílogo
  mov  esp,ebp          ; Coloca ESP no topo real, de novo.
  pop  ebp              ; ... e recupera EBP antes de retornar.
  ret
```

O prólogo guarda na pilha o conteúdo do registrador EBP e muda o topo da pilha (ESP) para além do espaço reservado para armazenamento local. Isso tem que ser feito porque nossa função pode chamar outras e ESP precisa apontar para o último item “empilhado”, sempre!

Durante a execução do prólogo temos:

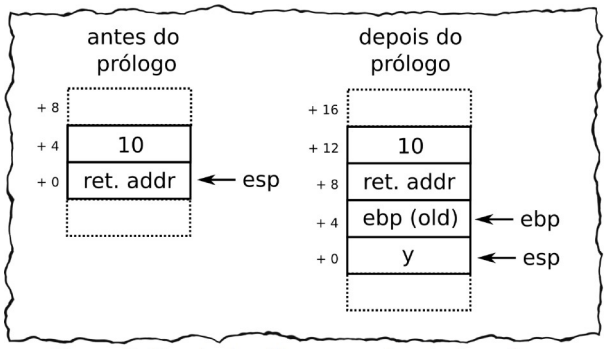


Figura 2: Estado da pilha antes e depois do prólogo

Depois disso o compilador tende a usar EBP como topo da pilha, dentro da função... No final da rotina o epílogo só precisa retornar ESP para o topo original e recuperando o valor de EBP.

Instruções especiais para lidar com prólogos e epílogos

Desde o 80386 a Intel inseriu duas novas instruções para lidar com prólogos e epílogos. São elas: ENTER e LEAVE... LEAVE é simples: ela faz a mesma coisa que o par de instruções “mov esp,ebp/pop ebp”.

A instrução ENTER é **lenta** e **complicada**. Por isso você não vai encontrá-la sendo usada pelo GCC ou pela maioria dos compiladores que conheço. Ela possui dois operandos imediatos (constantes): O primeiro é o tamanho da região temporária que deve ser reservada na pilha. O segundo parâmetro é um "nível de aninhamento" onde, geralmente, é usado o valor 0 para que o comportamento seja o mesmo do prólogo mostrado anteriormente. Valores diferentes de zero causam uma complicação desnecessária, cuja explicação pode ser encontrada nos manuais de desenvolvimento da Intel. Não vou explicá-lo aqui.

Se ENTER é ignorada pelo compilador, LEAVE é mais comum. Além do estilo de prólogo/epílogo mostrado anteriormente, você pode topar com este:

```
; Prólogo
push  ebp
mov   ebp,esp
add   esp,4     ; reserva 4 bytes (1 dword).
…
; Epílogo
leave

ret
```

Parâmetros em ponto flutuante

No modo i386 eles são **sempre** passados na pilha da CPU, mas o valor de retorno, se for um *float*, *double* ou *long double* é sempre retornado no topo da pilha **do coprocessador matemático** (*x87*). Mostro como lidar com ponto flutuante num capítulo específico sobre o assunto...

No GCC, se a opção de compilação *-fpmath=sse* for usada, em conjunto com a opção *-msse*, então o registrador XMM0 será usado como retorno. Isso deve ser evitado, já que outras funções podem esperar o retorno na pilha do *x87*. Se for usar SSE no modo i386, em C, o ideal é usar os tipos intrínsecos destinados para isso, como *__m128*.

A convenção de chamada para o modo x86-64

Para o modo de 64 bits a convenção de chamada padrão foi otimizada... Ao invés de passar parâmetros pela pilha, ela usa os registradores RDI, RSI, RDX, RCX, R8 e R9, nessa ordem, para os 6 primeiros parâmetros e só então usa a pilha, no caso dos parâmetros além do sexto. Isso diminui um bocado a quantidade de acessos à memória e, no geral, causa um aumento de performance sensível. Funções com mais de 6 parâmetros é algo bem raro de encontrarmos.

Diferente do modo i386, os cálculos em ponto flutuante são feitos através de instruções SSE por *default*. O x87 só é usado em caso de uso do tipo *long double* ou se o tipo não existir na versão escolhida para o SSE (O SSE original não suporta *double*, por exemplo). Dessa forma, os registradores de XMM0 até XMM7 são usados para passagem de parâmetros e XMM0 é sempre usado para retornar um ponto flutuante.

Isso aumenta um bocado a quantidade de registradores disponíveis para passarmos parâmetros para funções. Podemos ter funções com 14 parâmetros (6 inteiros e 8 em ponto flutuante) sem usarmos a pilha! O compilador só usará prólogos e epílogos (se usar) caso esses limites sejam extrapolados.

Registradores que devem ser preservados entre chamadas (x86-64)

Assim como no modo i386, alguns registradores não devem ser modificados entre as chamadas. São eles:

- RBX, RBP;

- Os registradores de R12 até R15;
- Os registradores XMM6 até XMM15 precisam ser preservados apenas no Windows. Essa restrição não existe no POSIX.

E quanto a convenção PASCAL?

Essa convenção é, essencialmente, idêntica à convenção *cdecl* com apenas duas diferenças:

1. Os parâmetros são empilhados na ordem do primeiro para o último. Na ordem contrária do que é feito na convenção *cdecl*;
2. A própria função é responsável por "limpar" a pilha.

Quanto a limpeza da pilha, existe uma variação da instrução RET que aceita um operando imediato que é adicionado a RSP. Esse valor imediato é indicado em bytes, portanto, se uma função toma 2 parâmetros do tipo *int*, a instrução RET na nossa rotina terá que ser (na convenção PASCAL apenas!):

```
ret 8
```

No modo x86-64, que exige alinhamento de RSP por quadword, esse valor deverá ser 16, já que a função chamadora vai empilhar dois *qwords*, mesmo que o tipo do parâmetro seja *int* (32 bits).

Quanto a ordem de empilhamento, a convenção PASCAL não permite que funções possam ter número variável de argumentos. A função tem que saber exatamente quantos parâmetros existem, já que o primeiro parâmetro estará empilhado na "parte de baixo" da pilha (porque é empilhado primeiro).

Em minha opinião essa convenção é útil em apenas dois casos: O primeiro e óbvio é se você estiver usando um compilador PASCAL como o Delphi, por exemplo. O segundo caso é para aqueles que desenvolvem aplicações para Windows. A Win32 API usa uma convenção bem parecida com PASCAL chamada *stdcall*.

Um código fonte escrito em C, para Windows, esconde isso de você através de alguns *typedefs*. A função *WinMain*, por exemplo, tem o protótipo:

```
int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int);
```

Esse modificador *WINAPI* ai é, na verdade, o modificador de convenção chamado *stdcall*. A convenção STDCALL foi criada pela Microsoft, na época do Windows NT, para permitir a passagem de número variável de parâmetros (do mesmo jeito que *cdecl*, empilhando do último para o primeiro), mas mantém a regra de que a função chamadora é que é a responsável por limpar a pilha.

O assembly inline

Alguns compiladores, de algumas linguagens de alto nível, oferecem ao programador a possibilidade de codificarem rotinas em assembly nas próprias linhas (inline) do programa. É o caso da maioria dos compiladores C e alguns compiladores PASCAL, como o Delphi e o antigo Turbo Pascal. Abaixo, mostro como codificar essas rotinas no GCC.

Para não esculhambar com o trabalho do compilador, o assembly inline do GCC funciona como se fosse uma função separada, onde o programador pode ter que tomar alguns cuidados... Nessa função podemos ter informações de entrada, saída e de preservação de registradores. A sintaxe é esta:

```
__asm__ __volatile__ (
  "código em asm"
  : descritores de saída
  : descritores de entrada
  : registradores modificados
);
```

Esses descritores são strings descrevendo o tipo esperado e um dado (ou variável) entre parêntesis. Por exemplo:

```
/* Faz "y = x + 2" */
__asm__ __volatile__ (
  "movl %%ecx,%%eax;"
  "addl $2,%%eax"
  : "=a" (y)
  : "c" (x)
);
```

O descritor de entrada "c" diz ao compilador que a variável *x* será colocada em ECX (dependendo do tamanho de *x*) **antes** de "chamar" a rotina descrita no bloco de código. O descritor de saída "=a" indica que o valor contido em EAX será colocado na variável *y* **depois** que a rotina for executada. Se ambas as variáveis forem globais, por exemplo, o fragmento acima criará algo assim:

```
mov  ecx,[x]     ; de acordo com o descritor de entrada.
mov  eax,ecx
add  eax,2
mov  [y],eax     ; de acordo com o descritor de saída.
```

Note que essa nossa rotina não precisa preservar quaisquer registradores porque apenas usamos EAX e ECX, que já constam das listas de entrada e saída. Se usássemos o registrador EBX, por exemplo:

```
/* Faz "y = x + 2", mas preserva EDX */
__asm__ __volatile__ (
  "movl %%ecx,%%eax;"
  "xorl %%ebx,%%ebx;"
  "addl $2,%%eax"
  "adcl %%ebx,%%eax"
  : "=a" (y)
  : "c" (x)
  : "%ebx"
);
```

Neste caso é prudente colocar EBX na lista de preservação, já que ele não consta das outras duas listas (que usam "a" e "c") e é alterado dentro do bloco assembly. O compilador pode escolher salvar EBX em algum lugar e recuperá-lo logo no final da rotina:

```
push  ebx        ; de acordo com os descritores de preservação.

mov   ecx,[x]    ; de acordo com o descritor de entrada.

mov   eax,ecx
xor   ebx,ebx
add   eax,2
adc   eax,ebx

mov   [y],eax    ; de acordo com o descritor de saída.

pop   ebx        ; de acordo com os descritores de preservação.
```

A maneira como o compilador fará a preservação depende exclusivamente dele. Inclusive o método de preservação... No modo x86-64 o compilador poderia escolher manter uma cópia de EBX em R10D, por exemplo!

Existem dois descritores adicionais, além dos registradores, para a lista de preservação. "memory" e "cc". O descritor "cc" nada mais é do que o registrador EFLAGS. Já o descritor "memory" é uma

dica ao compilador, caso sua rotina modifique memória, como acontece com funções como STOS e MOVS.

Descritores, no assembly inline

A lista de descritores, tanto de saída quanto de entrada, é numerada a partir de 0 (zero). O primeiro descritor é referenciado no código em assembly como %0, o segundo como %1 e assim por diante. Para ilustrar, poderíamos ter escrito a função anterior como:

```
/* Faz “y = x + 2” */
__asm__ __volatile__ (
  "movl %1,%0;"
  "addl $2,%0"
  : "=a" (y) : "c" (x)
);
```

Para os modos i386 e x86-64 os seguintes descritores são mais comuns:

Descritor	**Significado**
a, b, c, d, S, D	Registradores RAX, RBX, RCX, RDX, RSI e RDI, respectivamente, dependendo do tamanho do parâmetro.
A	O par de registradores EDX:EAX. (RDX:RAX **não** pode ser usado com isso!).
f	“Registrador” do x87
g	O compilador escolhe colocar o parâmetro num registrador de uso geral ou usá-lo através de um ponteiro.
i, n	Um valor imediato.
m	Referência à memória (ponteiro).
r	Registrador de uso geral.
t	“Registrador” ST(0) do x87.
u	“Registrador” ST(1) do x87.
x	Um registrador SSE.
Yz	O registrador XMM0

Tabela 1: Descritores comuns para assembly inline.

Repare que um descritor pode ser modificado: No caso do descritor de saída é comum que o modificador “=” seja colocado antes para dizer ao compilador que este valor será sobrescrito por alguma instrução no código, mas não será lido. Se quisermos dizer que alguma instrução vai ler e escrever neste parâmetro podemos usar o modificador “+”, como no exemplo:

```
/* Faz “y = x + 2”, mas preserva EDX */
__asm__ __volatile__ (
  "xchgl %%ecx,%%eax”
  : "+a" (y) : "0" (y), "c" (x)
);
```

Aqui, os descritores de entrada nos dizem que *y* será colocado no mesmo lugar do descritor de saída (o de índice 0) e *x* será colocado em ECX. Já o descritor de saída diz ao compilador que usará o registrador EAX que poderá ser lido e escrito. Como a instrução XCHG lê e escreve em ambos os registradores que usa, é conveniente que isso seja informado ao compilador, para dar-lhe a chance de realizar alguma otimização. Existem outras modificações que podem ser feitas a um descritor, consulte o manual do GCC para maiores detalhes.

Ainda quanto aos descritores, alguns direcionam os dados para registradores específicos, como os descritores “a”, “b”, “c”, “d”, “S” e “D”, por exemplo. Mas também temos os genéricos, que deixam o compilador escolher por nós. É o caso de “r”, “m” e “g”.

Poderíamos reescrever nosso pequeno código assim:

```
/* Faz “y = x + 2” */
__asm__ __volatile__ (
  "movl %1,%%eax;"
  "addl $2,%%eax"
  : "=a" (y) : "r" (x)
);
```

Ao usar um descritor de entrada “r” deixamos o compilador escolher qual registrador de uso geral ele usará. Se usássemos um descritor “m” o compilador substituirá o operando %1 por uma referência à memória, ou seja, um ponteiro para onde o parâmetro de entrada está.

Ambos “r” e “m”, mesmo sendo genéricos, podem causar uma pequena alteração no código final. Suponha que a variável *x* esteja no registrador EDX antes que o bloco seja executado, se usássemos o descritor “m”, o compilador seria obrigado a armazenar o conteúdo de EDX em algum endereço de memória (provavelmente local, na pilha) para obedecer o descritor. O código acima, depois de compilado poderia ficar assim:

```
mov  [esp-4],edx     ; Adicionado para obedecer o descritor “m”.
mov  eax,[esp-4]
add  eax,2
```

A mesma coisa acontece com “r”, mas no sentido contrário. Se a variável *x* for global, por exemplo, o compilador não teria outra alternativa a não ser carregar um registrador qualquer com o dado contido no ponteiro que referencia a variável:

```
mov  edi,[x]         ; Adicionado para obedecer o descritor “r”.
mov  eax,edi
add  eax,2
```

Para evitar essas adições inconvenientes podemos usar o descritor “g”, que diz ao compilador que qualquer coisa (registrador ou memória) pode ser usada no operando.

Combinando descritores

Existem descritores adicionais que podem ser combinados aos tradicionais, listados anteriormente. O descritor “N”, por exemplo, especifica um valor de 8 bits. Assim, um descritor de entrada “dN” especifica o registrador DL (ou DH, o compilador decide!). O descritor “I” especifica um valor entre 0 e 15... Ao especificar um descritor de entrada “cI” o compilador pode adicionar a instrução abaixo para garantir que apenas os 4 bits inferiores sejam usados:

```
and rcx,0x0f      ; ou “and cl,0x0f”, dependendo do caso.
```

Dê uma olhada no manual do gcc para maiores detalhes.

Usando o “sabor” Intel na linguagem assembly inline

GCC usa o sabor AT&T por default, para usar o sabor Intel, basta usar a diretiva “.intel_syntax” no seu código, sem esquecer de retornar ao “sabor” original, usando “.att_syntax”:

```
/* Faz “y = x + 2” */
__asm__ __volatile__ (
  ".intel_syntax;"
  "mov  eax,%1;"
  "add  eax,2;"
```

```
  ".att_syntax;"
  : "=a" (y) : "g" (x)
);
```

Na sintaxe Intel os registradores não precisam ser precedidos de “%%”, assim como constantes numéricas não precisam ser precedidas de “$”. Além disso, os registradores “salvos” (clobbered), se forem especificados, não podem ter um prefixo “%”.

A única limitação significativa do sabor Intel é que variáveis locais nomeadas da mesma forma que registradores não podem ser usadas (ao tentar usar uma variável chamada EAX, o compilador reclama!). Mas, é claro, se estamos lidando com essa variáveis de forma indireta, via descritores, então isso não é um problema.

E os registradores estendidos, de R8 até R15?

Reparou que não existem descritores para esses registradores estendidos? Isso não significa que você não possa usá-los no bloco assembly ou usá-los para passar parâmetros! Felizmente, no GCC, podemos forçar a barra e dizer para o compilador que uma variável **local** deverá ser alocada num registrador específico, declarando-a desse jeito:

```
register int x asm("r8");
```

Essa é uma extensão do GCC... Assim, quando passarmos a variável **local** *x* para um bloco de assembly inline, através de um descritor de entrada “r”, o compilador irá usar R8 nos operandos associados a este descritor.

Reitero o aviso: Esse macete de associar uma variável local com um registrador só é válido para **variáveis locais** e, mesmo assim, devemos respeitar a convenção de chamada. Evitar usar registradores que devem ser preservados é sempre uma boa ideia.

Labels “relativos” dentro do código assembly

No assembly inline é imprudente criarmos labels com nomes absolutos. Eles podem ser símbolos usados no programa em C e isso deixará o compilador confuso.

Saltos condicionais, por exemplo, podem tirar vantagem de uma notação “relativa” para nomeação de labels. Basta numerar os labels e usarmos um sufixo “f” ou “b” para “forward” (para frente) e “backwards” (para trás):

```
__asm__ __volatile__ (
  ".intel_syntax;"
  "   xor eax,eax;"
  "   cmp %1,2;"
  "   jl  1f;"
  “   inc eax;"
  "1: ;"
  ".att_syntax"
  : "=a" (y) : "g" (x)
);
```

O label “1f”, usado em conjunto com a instrução JL, diz ao compilador para saltar para o label “1” à frente. Se o label “1” estivesse colocado **antes** da instrução JL teríamos que usar a notação “1b”.

Capítulo 4 – Instruções

Embora você não tenha sido apresentado formalmente aos mnemônicos das instruções que o processador entende, já vimos algumas delas. A instrução MOV, por exemplo, cria uma cópia de um dado de um lugar para outro, seja esse "lugar" um registrador ou memória. Uma instrução CALL empilha o endereço da próxima instrução e salta para o endereço especificado na instrução. Um RET desempilha um endereço e salta para ele... Vimos também algumas instruções aritméticas como ADD, SUB, INC e DEC.

Vou te apresentar outras aqui:

As diferentes instruções de multiplicação inteiras

Se você der uma olhada, verá que existem duas instruções de multiplicação e duas de divisão inteiras: MUL e IMUL, DIV e IDIV. As instruções MUL e DIV lidam com multiplicação e divisão **desconsiderando** o sinal dos operandos, já IMUL e IDIV levam em consideração o sinal.

A instrução IMUL aceita diferentes número de operandos. Historicamente, essas instruções usam, implicitamente, o registrador RAX como um dos fatores e um operando como o outro. O resultado é colocado no mesmo par de registradores RDX:RAX, onde RAX recebe o quociente e RDX o resto:

```
imul rbx        ; rax:rdx = rax * rbx
imul rcx,rax    ; rcx = rcx * rax
imul rdx,rax,3 ; rdx = rax * 3
```

As duas outras variações de IMUL, mostradas acima, podem causar problemas: Se o resultado da multiplicação não couber no operando destino, haverá overflow, afetando os flags OF e/ou CF.

O mesmo vale para multiplicações de "tipos" menores. Se formos realizar multiplicações de 32 bits, EAX é usado implicitamente e o resultado é colocado em EDX:EAX. Se a multiplicação for em 16 bits, AX é usado implicitamente e resulta em DX:AX. A mesma coisa acontece com 8 bits, mas AL é o registrador implícito e resultado é colocado em AX.

Resta-nos entender como os flags CF e OF são afetados. Obviamente o flag OF é afetado de acordo com o sinal dos operandos e do resultado. Ou seja, se o resultado, sinalizado, não couber no resultado, OF será setado. Já o flag CF desconsidera o sinal, mas a lógica é a mesma.

E quanto às divisões?

As instruções DIV e IDIV funcionam de forma parecida. Só que o par de registradores RDX:RAX será usado como dividendo, implicitamente (RDX:RAX para 64 bits, EDX:EAX para 32, DX:AX para 16 e AX para 8). O resultado dessas instruções é colocado, de volta, no par RDX:RAX, onde RAX conterá o quociente e RDX o resto.

Isso significa que, ao fazer a divisão de dois *int*s, por exemplo, o conteúdo de EDX deverá ser ser ajustado **antes** da divisão:

```
; Assumindo que EAX contém o dividendo e ECX o divisor...
cdq           ; Estende o sinal de EAX para EDX.
idiv ecx      ; Divide EDX:EAX por ECX.
```

As instruções CWD (*Convert Word to Doubleword*), CDQ (*Convert Doubleword to Quadword*) e CQO (Convert *Quadword to Octaword*) simplesmente estenderão o sinal (bit de mais alta ordem) de AX, EAX e RAX, respectivamente, para DX, EDX e RDX, respectivamente. Existe também a instrução CBW (*Convert Byte to Word*), que estende o sinal de AL para AX.

Tome cuidado que também existem duas instruções estendidas, CWDE e CDQE, que estendem o sinal de AX para EAX e EAX para RAX, respectivamente. Mas elas **não** afetam EDX ou RDX.

NEG e NOT

Essas instruções podem parecer equivalentes, mas a primeira é uma instrução aritmética e a segunda, uma instrução *lógica*. A instrução NOT simplesmente inverte todos os bits do operando. Já a instrução NEG realiza o *complemento 2* do operando, isto é, além de invertê-lo, adiciona 1. Dessa forma, NEG é a instrução que você quer quando precisa inverter o sinal de um valor inteiro.

INC e DEC não são tão boas assim

As instruções de incremento e decremento, INC e DEC, têm um problema: Elas são instruções aritméticas mas não afetam todos os flags, especialmente o CF (*Carry Flag*)! Isso significa que o processador terá que ler EFLAGS, modificá-lo e reescrevê-lo, quando executa um INC ou um DEC. Algo mais ou menos assim:

```
tmp_EFLAGS = EFLAGS & 0x01; /* Isola CF */
dest++;
if (tmp_EFLAGS != (EFLAGS & 0x01))
{
  EFLAGS &= ~0x01
  EFLAGS |= tmp_EFLAGS;
}
```

Ou seja, usar INC ou DEC cria código mais lento do que usar ADD ou SUB.

É preferível usar:

```
add eax,1
```

Do que usar um incremento do tipo “INC EAX”. Só tem um probleminha: A instrução acima ocupa 5 bytes em seu μ_{op}. A inscrição INC ocupa apenas um. Vale a pena avaliar se a perda de performance é significativa ao ponto de justificar esse tipo de pressão adicional ao cache L1i.

Realizando cálculos simples com uma única instrução

E, por “calculo”, quero dizer aritmética elementar.

Lá atrás eu disse que ponteiros podem ser formados por estruturas do tipo “[base + n*escalar + deslocamento]”, onde *n* pode ser 1, 2, 4 ou 8, “base” e “escalar” são registradores e “deslocamento” é uma constante.

Existe uma instrução que, com base nessa especificação, obtém o calculo do endereço e o coloca num registrador. A instrução LEA (*Load Effective Address*) faz isso. Por exemplo: Se quisermos multiplicar EAX por 5, poderíamos fazer de duas maneiras[20]:

```
mov ebx,eax             ; Guarda o valor original de eax em ebx.
add eax,eax             ; Multiplica eax por 2.
add eax,eax             ; Multiplica eax por 2, de novo.
add eax,ebx             ; Adiciona o valor original de eax.
----- corte aqui -----
lea eax,[eax + 4*eax]   ; faz a mesma coisa.
```

LEA calculará o que está dentro dos colchetes e colocará esse valor em EAX. No exemplo acima, multiplicar EAX por 5 usando as adições gastará cerca de 3 ciclos de máquina. No caso do uso da instrução LEA gasta-se 1 ciclo a menos (ou seja, 2 ciclos).

20 Claro que também poderíamos fazer “imul eax,eax,5”, mas esse é o método **lerdo** de fazer isso!

Repare que, usando o método do LEA, podemos:

```
lea eax,[eax*2]        ; multiplicar por 2.
lea eax,[eax+eax*2]    ; multiplicar por 3.
lea eax,[eax*4]        ; multiplicar por 4.
lea eax,[eax+eax*4]    ; multiplicar por 5.
lea eax,[eax*8]        ; multiplicar por 8.
lea eax,[eax+eax*8]    ; multiplicar por 9.
```

As multiplicações por 6, 7 e valores maiores que 9 não são possíveis assim... Mas, nada nos impede de usar o macete anterior:

```
; Multiplica eax por 7.
lea ebx,[eax+eax*4]
add eax,ebx
```

E, se combinarmos vários LEAs, podemos fazer multiplicações até maiores[21]!

Existem duas desvantagens: LEA só lida com valores sem sinal e a instrução **não** afeta dos flags.

A limitação de MOV e a cópia de strings

A instrução MOV *move* dados, mas tem um porém: Não existe uma instrução MOV que mova dados na memória para outro endereço na memória... Bem... Na verdade existe...

Lembra-se que falei que alguns registradores de uso geral, às vezes, têm significados específicos? O registrador RCX (ou ECX) é também chamado de “contador” porque ele é usado, em certas instruções, com esse propósito. Da mesma forma que os registradores RSI e RDI são conhecidos, respectivamente, como *índices fonte* e *destino*.

Existe uma instrução MOVS que deve ter um sufixo B, W, D ou Q, dependendo do tamanho do dado que estamos manipulando (1 byte, word, doubleword ou quardword, respectivamente) que usa o par de registradores DS:RSI como ponteiro para o dado que será movido e ES:RDI como ponteiro para onde esse dado será movido. E essa instrução, quando usada com o prefixo REP, será repetida RCX vezes, incrementando ou decrementando RSI e RDI (depende do valor do flag de direção DF) em cada iteração e decrementando RCX também.

O mnemônico da instrução deriva do inglês *MOVe String [of Bytes, Words, Dwords or Quadwords]*. Então, se quiséssemos mover 30 bytes do array *a* para um array *b*, poderíamos fazer assim:

```
mov rsi,a
mov rdi,b
mov ecx,30
rep movsb
; Ao terminar rep movsb, o registrador ecx estará zerado e
; os registradores esi e edi estarão incrementado exatamente em 30.
```

Não confundir *string*, neste caso, como sequência de caracteres terminada em '\0'. O termo, em assembly é somente um sinônimo para “sequência”.

Outra coisa importante a se lembrar é que o seletor de segmento *fonte* é diferente do seletor de segmento *destino*. Mas, na maioria dos sistemas operacionais modernos os seletores *ds* e *es* têm o mesmo valor e, portanto, selecionam o mesmo segmento de dados.

Existem duas instruções MOVSD?!

Se você consultar a lista de mnemônicos oficiais dos processadores Intel e AMD verá que existem **duas** instruções MOVSD diferentes. A primeira lida com movimentação de blocos e a outra é usada

21 Não é prudente usarmos LEA para realizarmos multiplicações por múltiplos de 2^n. Para tanto a instrução SHL é, além de mais simples, mais rápida.

para carregar um registrador XMM com um escalar *double*.

A diferença entre elas está nos operandos. No primeiro caso não há operandos. A instrução moverá um *dword* de *ds:rsi* para *es:rdi*. No segundo caso, um dos operandos tem que ser um registrador XMM:

```
movsd           ; move ds:[rsi] para es:[rdi]
movsd xmm0,[x]  ; move um double de [x] para xmm0 (sse).
```

Preenchendo um array

Da mesma forma que existe um MOVS, existe um STOS. O mnemônico vem de *STOre into String* e, da mesma forma que MOVS deve ser sucedido de uma letra indicando o tamanho de cada item do array, neste caso, apontado pelo par *es:edi*.

Também, da mesma forma que MOVS, STOS pode ser prefixado com REP e, neste caso, *rcx* indica quantos itens serão afetados.

STOS usa o acumulador para obter o dado que será armazenado. Se usarmos STOSB, o registrador *al* será usado. STOSW usa *ax*, STOSD usa *eax* e STOSQ (no modo x86-64) usará *rax*. Assim, para preecher os 30 bytes de um array *a* com zeros, poderíamos fazer assim:

```
mov rdi,a
xor al,al
mov ecx,30
rep stosb
```

Movimentações e preenchimentos de strings nas arquiteturas modernas

Essas instruções de manipulação de strings prefixadas com REP foram introduzidas na família 80x86 desde os primeiros processadores 8088. Mas, desde os 386 elas se tornaram lentas, em comparação com rotinas mais discretas como:

```
.L1:
  movsb
  dec ecx
  jnz .L1
```

Mas, recentemente, as instruções REP MOVS e REP STOS, especialmente as REP MOVSB e REP STOSB voltaram a ficar bem rápidas... O processador tenta mover os maiores blocos de dados possíveis, de uma só vez, quando pode. Isso torna REP MOVSB e REP STOSB mais rápidos do que usar, por exemplo, SSE.

Comparações e buscas de strings

Além de mover e armazenar valores em strings podemos também compará-los. Isso é o que a instrução CMPS faz. Essa instrução funciona da mesma forma que a instrução CMP, só que usa DS:RSI e ES:RDI como ponteiros para os dois dados.

Da mesma forma que MOVS e STOS, precisamos saber o tamanho dos dados e colocar um sufixo na instrução. E também, se usarmos o prefixo REP, obteremos repetição com o tamanho da “string” limitado pelo registrador RCX.... Mas, diferente de MOVS e STOS, a instrução REP agora depende da avaliação do flag ZF.

O objetivo de CMPS é comparar duas strings até que uma diferença ou igualdade seja encontrada... Assim, se usarmos o prefixo REPZ ou REPE estaremos dizendo para a instrução CMPS que deve continuar as comparações **enquanto Z=1**. Ou seja, enquanto as strings são iguais. Se invertemos o

sentido de REP para REPNZ ou REPNE a contagem continuará **enquanto Z=0**.

> É importante ressaltar isso: REPZ ou REPNZ não é uma referência ao conteúdo de RCX. Quem altera o conteúdo desse registrador é a instrução que sucede o prefixo REP(N)Z. Quando a instrução encontra um RCX=0 ela abandona o loop... O que REP(N)Z testa é o flag ZF **antes** da próxima iteração na instrução.

Diferente da comparação de duas strings a instrução SCAS (de *SCAn String*) usará o acumulador como fonte, bem como DS:ESI. A comparação é feita entre esses dois **enquanto** o ZF for avaliado, no caso no uso de REPNZ ou REPZ, exatamente como descrito acima.

Eis uma rotina interessante que poderia substituir a rotina *strlen()* da biblioteca padrão de C, no modo x86-64:

```
; Protótipo da rotina:
;   size_t _mystrlen(char *);
_mystrlen:
  mov   rsi,rdi     ; rdi é o ponteiro da string passado para a rotina.
  mov   rcx,-1      ; Inicializa rcx com 0xffffffffffffffff
  xor   eax,eax     ; Vamos procurar por '\0'.
  repne scasb       ; Enquando [rsi] != 0 e rcx != 0, continua buscando.
  not   rcx         ; Obtém a quantidade de bytes “bypasados” invertendo rcx.
  dec   rcx         ; Desconsidera o '\0'.
  mov   rax,rcx     ; Coloca o valor em rax para o retorno.
  ret
```

Aqui, se o valor '\0' estiver na 5ª posição do array, o valor de RCX depois de REPNE SCASB será 0xfffffffffffffffa, que ao ser invertido pela instrução NOT nos dá 5. Precisamos desconsiderar o '\0' encontrado e por isso decrementa-se RCX.

Hummmm... porque não inicializei RCX com zero e evitei esse decremento final? Acontece que a instrução SCAS testa, primeiro, o conteúdo do contador. Se ele for zero, nada é feito. Por isso inicializei RCX com o valor -1 (todos os bits setados), que é o valor “positivo” máximo para um registrador de 64 bits.

Compare *_mystrlen* com a rotina original:

```
/* strlen.c */
size_t strlen(char *p)
{
  size_t sz = 0;
  while (*p++) sz++;
  return sz;
}
----- corte aqui -----
; código equivalente em asm:
strlen:
  xor eax,eax
  cmp byte [rdi],0
  je  .L2
.L1:
  inc rax
  cmp byte [rdi+rax],0
  jne .L1
  ; Neste ponto eax terá o valor do tamanho da string.
.L2:
  ret
```

Infelizmente, para mim, a rotina da glibc é bem mais rápida que a minha[22].

22 REPNZ SCASB acaba sendo mais lenta que o trio INC/CMP/JNE, acima... E também existe um problema insignificante em minha rotina: No modo x86-64, se a string tiver mais que 2^{52}-1 bytes teremos um overflow no endereço canônico e obteremos uma exceção geral de proteção (GPL). No modo i386 isso não aconteceria, já que o máximo que temos por seletor são 4 GiB.

O prefixo REP pode ser usado para alinhamento de código

O GCC, às vezes, adiciona um REP na frente de um retorno de função (RET). Esse REP é inócuo, ele estará lá apenas para garantir que a próxima instrução esteja alinhada em *dword* ou *qword*, de acordo com o modo de operação do sistema operacional e do próprio GCC.

Outras instruções de movimentação interessantes

Além das instruções de manipulação de blocos, existem outras que nos ajudam um bocado, de tempos em temos. É o caso de MOVSX e MOVZX. O "S" ai não é de *String*, mas de **sinal** (ou *signal*). A instrução MOVSX "eXtends" (estende) o bit de sinal do operando fonte para os bits restantes do operando destino, enquanto MOVZX "zera" os bits adicionais.

O detalhe é que tanto MOVSX quanto MOVZX são instruções que movem um valor de tamanho menor para um de tamanho maior (um byte ou word para um dword, por exemplo). No exemplo abaixo, movemos um tipo *char* para um tipo *int*:

```
mov   al,-1         ; AL = 0xff
movsx ebx,al        ; EBX será também -1!
```

Para estender o sinal de AL, o bit 7 será replicado nos bits 7 e superiores de EBX.

A instrução MOVZX, por outro lado sempre preencherá os bits superiores do *alvo* com zeros, copiando *ispsis literis* o conteúdo do operando *fonte*. As instruções:

```
mov   al,-1         ; AL = 0xff
movxz ebx,al        ; EBX será 0x000000ff
```

Uma outra instrução de movimentação interessante, introduzida no modo x86-64, é MOVBE, onde "BE" vem de *Big Endian*. O que essa instrução faz é inverter a ordem dos bytes do operando fonte e copiar para o operando destino. Ela não é tão flexível quanto MOV, MOVSX e MOVZX, já que um dos operandos obrigatoriamente tem que ser um registrador e o outro tem que ser uma referência à memória:

```
movbe eax,[rbx]      ; ok
movbe [rdi],edx      ; ok
movbe eax,ebx        ; isso não existe!
```

MOVBE pode não estar disponível para o seu processador, por isso os compiladores C tendem a não usá-la.

Porque a instrução LOOP nunca é usada pelo GCC?

Sim, existe uma instrução *LOOP* que funciona exatamente como na sequência abaixo:

```
.L1:
  …
  loop .L1       ; Decrementa RCX. Salta para L1 se RCX != 0.
                 ; Ela substitui perfeitamente as instruções "dec rcx/jnz .L1".
```

A instrução LOOP é até menor que as duas instruções acima, teoricamente colocando menos pressão no cache L1i. Acontece que LOOP é um pouco mais **lenta** que essas instruções e, por isso, você provavelmente não vai ver seu compilador C favorito usando-a.

Existem também as variações LOOPZ e LOOPNZ, com a mesma semântica do REP(N)Z: Primeiro a instrução testa o flag ZF, depois decrementa RCX e realiza o salto se ele não for zero.

De qualquer maneira, não é uma instrução que valha a pena ser usada...

Disse "insignificante" porque é extremamente improvável que venhamos a lidar com strings de 4 GiB ou mais.

XLAT é outra instrução "inútil"

Imagine que você tenha um array de 256 bytes e queira obter o 20º item. Chamarei o endereço inicial desse array de "a", então você poderá obter esse item assim:

```
mov al,[a+20]
```

Nos primórdios dos processadores da família 80x86 foi criada uma instrução especializada para fazer justamente isso. A instrução XLAT. Nela, o endereço inicial do array tem que ser colocado em EBX e o índice do array de bytes em AL:

```
mov ebx,a
xlat         ; faz al = [ebx+al]
```

Por que essa instrução é "inútil"? Primeiro porque podemos usar um simples *MOV* para fazer a mesma coisa. Depois, porque XLAT é **lenta**. E, finalmente, arrays de bytes estão limitados ao tamanho de 256 bytes (AL só tem 8 bits!). Você jamais verá seu compilador gerar código que use essa instrução... E você também não deveria usá-la em seus códigos em assembly.

Trocando valores em uma tacada só...

Eis uma instrução útil que tem que ser usada com alguma cautela. XCHG.

Imagine que você queira trocar os valores contidos em RAX e RBX. Uma das maneiras de fazer isso é usando um terceiro registrador:

```
; faz rax ↔ rbx, usando rdx como intermediário.
mov rdx,rax
mov rax,rbx
mov rbx,rdx
```

Felizmente a instrução XCHG não precisa desse registrador intermediário:

```
xchg rax,rbx
```

Simples assim... Acontece que existe uma "pegadinha": Se um dos operandos for uma referência à memória, como o dado contido nesse ponteiro precisa ser lido e depois gravado, o processador alterará o estado do barramento para *locking* (é um sinal elétrico colocado num dos pinos do processador) que informará que, num sistema multiprocessado, ninguém mais pode gravar nesse endereço.

Assim, todas as threads que tentarem ler/gravar esse endereço ficarão num estado de espera até que a "trava" seja liberada (até que XCHG termine de executar). A instrução XCHG pode piorar a performance de rotinas que executem em threads diferentes e acessem a mesma variável... De fato, a instrução XCHG é usada num esquema de sincronização de thread chamado *spin lock*, mas isso fica à cargo do sistema operacional.

> É preferível usar o método que usa um terceiro registrador ou uma variável em memória, usando 3 MOVs se a instrução XCHG equivalente usar uma referência à memória.

Tome sempre cuidado ao usar XCHG... Se ambos os operandos forem registradores, então não há com que se preocupar...

Outras instruções que "travam" o barramento

Toda instrução que efetua o ciclo leitura-modificação-escrita sinaliza para os demais processadores

que o barramento não pode ser usado até que a instrução seja completada. Mas, de novo, isso só acontece se o operando destino for uma referência à memória.

Graças a esse fato, é prudente evitar instruções como:

```
add dword [rbp-4],2     ; Lê o dword, soma 2 e escreve de volta (lock!).
```

Em alguns casos essa sequência pode ser substituída por:

```
mov eax,[rbp-4]
add eax,2
mov [rbp-4],eax
```

Embora tenhamos 3 instruções aqui, a reordenação feita internamente pelo processador e a ausência de *locking* faz essa sequência ser mais rápida que a instrução ADD anteriormente mostrada.

Lembre-se **toda** instrução que lê e depois escreve no mesmo operando realiza *locking*. Incluindo todas as instruções aritméticas elementares (ADD, SUB, INC, DEC, NEG) e lógicas (AND, OR, XOR, TEST, NOT). Isso inclui instruções mais especializadas também, como as do SSE, AVX, etc... Se performance é algo muito importante no seu código o uso do operando destino referenciando memória deve ser evitado ao extremo.

"Comparação" lógica

Vimos, mais cedo, que a instrução CMP realiza uma subtração e não coloca o resultado em lugar algum senão nos flags. Essa comparação é uma comparação aritmética (porque uma subtração é feita!). Existe também uma comparação lógica, que existe para verificar o estado de um conjunto de bits.

A ideia é a seguinte: Numa operação lógica AND, segundo as regras de identidade da álgebra de Boole, temos que:

$$1 \wedge x = x$$

$$0 \wedge x = 0$$

Onde "∧" é o operador lógico AND, em notação matemática. A variável *x*, aqui, tem apenas 1 bit de tamanho... Isso quer dizer que se fizermos um AND com 0 e *x*, o resultado **sempre** será zero, mas se a operação for feita entre 1 e x, então o resultado depende inteiramente do valor de *x*.

Isso nos permite "isolar" bits criando uma "máscara". Uma máscara "esconde" partes do rosto de quem a usa, então é comum usarmos o termo *bitmask* para dizermos que queremos ZERAR alguns bits de um valor. Por exemplo... Se queremos "mascarar" apenas os 4 bits superiores do registrador AL, poderíamos fazer:

```
and al,0x0f ; todos os bits acima do bit 3 de AL serão zerados!
```

A operação AND, por ser uma operação lógica, afetará o flag ZF[23]. Se o resultado da operação for zero então ZF=1. Isso nos permite testar um bit isolado. Suponha que queremos saber se o bit 0 de ECX está setado ou não. Basta fazer:

```
and ecx,1
jnz .L1      ; se está setado, salta para .L1.
…
```

Só tem um problema ai: O conteúdo do registrador ECX será destruído no processo. Para isso existe a instrução de "comparação" (ou teste) lógico TEST. Ela faz a mesma coisa que um AND, mas não coloca o resultado em lugar algum, só afetando o flag ZF.

23 E, por não ser uma operação aritmética, **não** afetará CF, SF ou OF.

Além da instrução TEST, podemos também usa uma das quatro instruções para teste individual de bits: BT, BTR, BTS ou BTC. A instrução BT copia o conteúdo de um bit específico do operando destino para o flag CF. Por exemplo, se quisermos testar o bit 3 de EAX poderíamos fazer:

```
  bt eax,3
  jc esta_setado
  …
esta_setado:
  …
```

A instrução aceita um registrador de igual tamanho do operando destino em seu segundo operando.

As outras instruções (BTR, BTS e BTC) também colocarão zero (reset), um (set) ou complementarão o bit correspondente no operando destino, respectivamente. Substitua BT por BTR no exemplo acima e o bit 3 de EAX será zerado (e o CF conterá o bit original).

Invertendo bits individuais e zerando registradores

Além das instruções AND e BT, podemos usar a operação lógica XOR ("ou exclusivo") para manipular bits. As regras de identidade de XOR são:

$$0 \oplus x = x$$

$$1 \oplus x = \neg x$$

Onde a operação $\oplus$ é o XOR e $\neg$ é o NOT.

Isso significa que ao usar um dos operandos como *bitmask* numa operação XOR, os bits setados **invertem** aqueles que estiverem na mesma posição. Com XOR podemos fazer um NOT seletivo!

Um exemplo simples é a rotina que determina se um valor inteiro é par ou não... Um compilador esperto pode detectar que estamos querendo fazer e usar as instruções AND e XOR para tal, ao invés de realizar uma divisão por 2:

```
/* Testa se x é par. */
int isEven(int x) { return (x % 2) == 0; }
----- corte aqui -----
isEven:
  mov eax,edi
  and eax,1      ; Isola apenas o bit 0 de EAX, zerando os demais.
  xor eax,1      ; Inverte o bit 0 de eax.
  ret
```

Ué? Cadê a divisão que estava aqui?!

Deslocando e rodando

Além das operações lógicas AND, OR, XOR e NOT, temos também como fazer deslocamentos binários para a esquerda e para a direita. Existem 3 tipos de instruções de deslocamento: Deslocamento simples, rotações simples e rotações através do flag de carry.

Para diferenciar esses tipos de deslocamentos temos *shifts* e *rotations*. *Shifts* delocam o conteúdo de um registrador (ou memória) de um bit para a esquerda ou direita, jogando no CF o bit que sair do registrador.

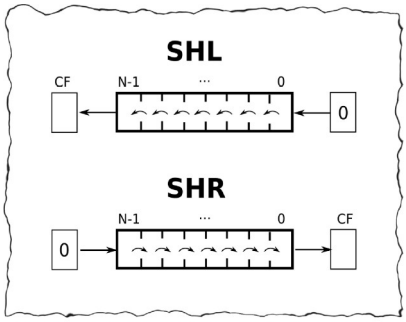


Figura 1: Como SHR e SHL funcionam

Nessa classe de deslocamentos existem os *shifts lógicos* e os *shifts aritméticos*. As instruções são SHL e SHR para os shifts lógicos para a esquerda e direita, respectivamente e, para shifts aritméticos, SAL e SAR. As instruções SAL e SHL são, de fato, a mesma instrução (elas fazem o mesmo trabalho). O motivo da diferença entre SAR e SHR é o bit de mais alta ordem que corresponde ao sinal. SAR repetirá esse bit depois do deslocamento para direita, mas SHR sempre o zerará. Eis dois exemplos usando deslocamento lógico para a esquerda e deslocamento aritmético para a direita.

```
shl rax,2           ; desloca rax 2 bits para a esquerda,
                    ; colocando zeros nos 2 bits inferiores.
sar dword [ebx],3   ; desloca o dado em [ebx] em 3 bits para a direita,
                    ; colocando nos 2 bits superiores o bit 31 original.
```

> De fato, seu compilador C vai usar SAR e SHR de acordo com o "tipo" do operando. No entanto, vale notar que a especificação da linguagem diz que shifts para a direita de valores integrais sinalizados (*char*, *int*, *long*, *long long*) são ambíguos e o compilador pode escolher muito bem inserir zeros nos bits superiores, ignorando o sinal. Portanto, se estiver programando em C, tome cuidado com isso!

Dos shifts lógicos, além de SHL e SHR, existem também as versões com dois registradores – SHLD e SHRD. Elas funcionam do mesmo modo que SHL e SHR, mas não preenchem o destino com zeros. Elas copiam os bits do segundo operando, de acordo com o deslocamento feito:

```
shld r8,rax,4       ; desloca r8 4 bits para a esquerda,
                    ; copiando os 4 bits MSB de rax nos LSBs de r8.
```

Já as rotações simples, além de colocarem o último bit que saiu do registrador no CF, fazem que que os bits deslocados sejam colocados de volta no registrador, mas do outro lado, fazendo uma rotação. Assim, temos ROL, ROR.

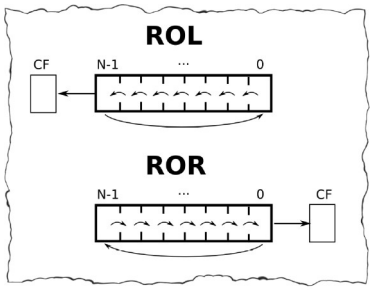


Figura 2: Como funcionam ROR e ROL

Se você der uma olhada no código gerado pelo GCC para a função *htons* ou *ntohs*, verá que a instrução ROR (ou ROL) é usada com um registrador de 16 bits[24]:

24 No caso de *ntohl* e *htonl*, o GCC usa a instrução **bswap**.

```
ror  ax,8        ; troca ah por al e vice-versa.
```

É claro que uma instrução “XCHG AH,AL” poderia ser usada nesse caso, mas a rotação funciona muito bem também.

Existem também rotações **através** de CF. Elas fazem com que o conteúdo do CF seja colocado no bit “vago” e o bit “que sair” seja colocado no CF. As instruções são RCL e RCR.

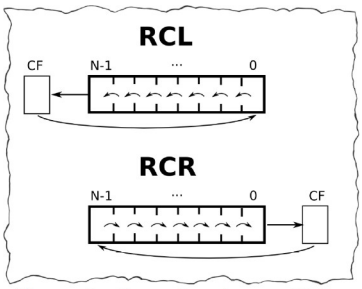


Figura 3: Como functionam RCL e RCR

O problema com RCL e RCR é que elas são cerca de 6 vezes mais lentas que os outros deslocamentos...

Contando bits zerados e setados de um lado ou de outro

As instruções BSF (*Bit Scan Forward*) e BSR (*Bit Scan Reverse*) retornam a posição do primeiro bit **setado** que encontrarem no operando fonte e colocam no operando destino. BSF procura pelo bit 1 a partir do LSB (do lado direito do operando fonte), enquanto BSR o faz a partir do lado esquerdo ou MSB.

Essas instruções tem a desvantagem de colocarem valores indefinidos (pode ser qualquer coisa) se o operando fonte for zero, mas elas também retornam ZF == 1, neste caso:

```
mov eax,0b00110000
bsf ecx,eax        ; ECX será 4 (bit 4 setado a partir do LSB).
Bsr edx,eax        ; EDX será 5 (bit 5 setado a partir do MSB).
```

Note que essas instruções não “contam” os bits, elas devolvem a posição do primeiro 1 que encontrarem. BSF e BSR existem desde o processador 386, mas existem instruções mais especializadas, porém não estão presentes em todos os processadores.

TZCNT (*Trailing Zeroes Count*) e LZCNT (*Leading Zeroes Count*) contam a quantidade de bits zerados mais ou menos da mesma forma que BSF e BSR fazem, mas retornam a quantidade encontrada, não a posição do bit setado. No frigir dos ovos isso não parece muito diferente do que BSF e BSR fazem, mas se o operando fonte estiver zerado, as duas novas instruções não retornarão um valor indefinido... A maneira de usá-las é semelhante a de BSF e BSR.

Outra instrução de contagem de bits é POPCNT. Ela retorna a quantidade de bits 1 que existem no operando fonte. Diferente de TZCNT e LZCNT, não importa onde os bits estejam no operando. Se fizermos algo assim:

```
mov eax,0b011010110
popcnt ecx,eax
```

O registrador ECX será ajustado para o valor 5, porque temos 5 bits setados em EAX.

Além dessas instruções, existe um outro conjunto chamado de BMI (*Bit Manipulation Instructions*), que também não estão presentes em todos os processadores das famílias Intel ou AMD.

Capítulo 5 – Ponto flutuante

Até o momento lidamos apenas com valores inteiros, manipulados através dos registradores de uso geral do processador e, é claro, existem problemas que exigem um pouco mais de "precisão" do que a aritmética inteira. Felizmente, para nós, os processadores modernos da arquitetura i386 e x86-64 nos fornecem recursos para lidarmos com *ponto flutuante* em hardware...

Este capítulo só arranha a superfície sobre o assunto. Para uma discussão mais aprofundada recomendo meu outro livro: "C & Assembly para arquitetura x86-64".

Números, em ponto flutuante

Assim como na aritmética inteira os números em ponto flutuante obedecem a notação posicional, só que as posições "depois da vírgula" (à direita) têm índice negativo:

$$1,523 = 1 \cdot 10^0 + 5 \cdot 10^{-1} + 2 \cdot 10^{-2} + 3 \cdot 10^{-3}$$

Existe uma diferença entre valores em "ponto flutuante" e "fracionários". Um valor fracionário pode ter qualquer valor inteiro que você quiser: 102,32 é um valor fracionário. Já os valores em ponto flutuante seguem uma regra chamada **notação científica**.

Todo número em ponto flutuante que segue o padrão IEEE 754 tem um padrão: A parte inteira é sempre $1.0_{(2)}$ seguida da parte fracionária. Note que escrevi o valor em binário. E todo número binário fracionário é multiplicado por 2^n. Assim, um valor fracionário é sempre descrito como:

$$valor = m \cdot 2^e$$

Onde *m* é chamada de *mantissa* e contém o valor binário entre 0.0 e 1.0. E o expoente *e*, com base 2, desloca o ponto para um dos lados. Ou seja, faz o ponto "flutuar" (daí o nome). Por exemplo: O valor 0,5 pode ser escrito como $1 \cdot 2^{-1}$. Note que o expoente diz para deslocarmos (flutuarmos?) o ponto em uma casa para a esquerda, obtendo $0.1_{(2)}$.

E já que estamos falando de "notação científica binária", esse valor inteiro '1' é **implícito** em toda codificação de um número em ponto flutuante. A mantissa *m* contém apenas a parte fracionária do número. Assim, a equação acima fica assim:

$$valor = (1 + m) \cdot 2^e$$

Onde *m* é um valor no intervalo entre 0 e 1, excluindo o valor 1.

Um *float* ou um *double* é codificado assim:

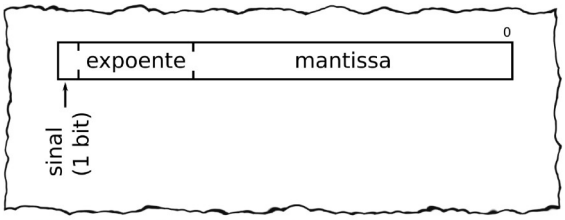


Figura 1: Formato de um valor em ponto flutuante

A diferença entre os tipos *float* e *double* é a quantidade de bits da mantissa e do expoente:

Tipo	Bits do expoente	Bits da mantissa
float	8	23
double	11	52

Table 1: Diferença entre float e double

O expoente também é um valor binário (óbviamente), mas ele precisa poder conter um valor negativo. Infelizmente esse valor não usa a técnica do **complemento 2**. No caso de um *float* o valor de *e* é calculado assim:

$$e = E - 127$$

Onde *E* é o valor do expoente contido num *float*. Com valores menores que 127 o valor de *e* torna-se negativo. No caso de um *double* o expoente tem 11 bits de tamanho e o valor constante é 1023.

Vimos, acima que 0.5 pode ser escrito, em "notação científica binária" como $1.0_{(2)} \cdot 2^{-1}$. Assim, na codificação dos 32 bits de um *float* teríamos: 0 01111110 $00000000000000000000000_{(2)}$ ou o valor 0x3f000000 (s=0, E=126 e m=0). Isso pode ser fácilmente comprovado com o fragmento de programa abaixo, em C:

```
float f = 0.5f;
unsigned int *p = (unsigned int *)&f;

printf("%#08X\n", *p);
```

Maneiras de lidar com ponto flutuante

Essencialmente existem duas maneiras de lidar com ponto flutuante: Usando os recursos do que era conhecido como *co-processador matemático*, que ainda existe nos processadores modernos, ou usando um recurso conhecido como SIMD.

O primeiro método é mais lento que o segundo mas tem uma vantagem: Esse método existe desde a época dos antidos processadores 8086 e não mudou quase nada de lá para cá. Usar esse método permite criar rotinas mais portáveis...

O segundo método é o preferido da arquitetura x86-64 e ele permite fazermos mais de uma operação ao mesmo tempo (SIMD significa *Single Instruction Multiple* Data, ou "uma instrução, múltiplos dados").

Usando o co-processador matemático

Neste método, chamado de *x87*, temos uma pilha de 8 níveis que conterá valores de 80 bits. Nesses 80 bits estará codificado um valor em ponto flutuante onde temos uma parte fracionária, binária, de 63 bits, um único bit "inteiro", um expoente de 15 bits e 1 bit de sinal.

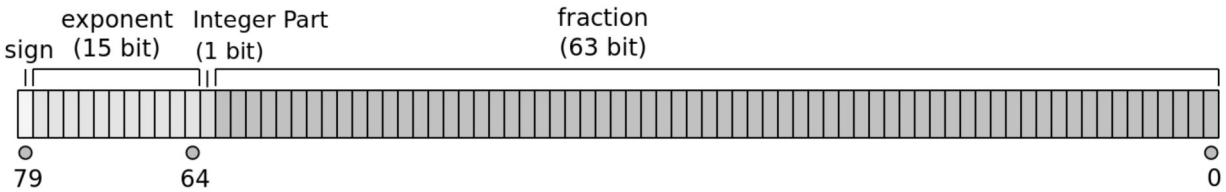


Figura 2: Formato estendido de 80 bits do x87

Todas as operações em ponto flutuante são feitas com essa precisão "estranha" de 64 bits da mantissa e, é claro, um valor em ponto flutuante deste tamanho não caberá num simples registrador de uso geral, mesmo no modo x86-64. É claro que podemos trabalhar com os tipos *float* e *double*,

como definidos no padrão IEEE. Internamente o “co-processador” trabalhará com 80 bits.

Como funciona esse “co-processador”? Da mesma forma que uma calculadora HP ou qualquer uma que use a notação polonesa reversa: Primeiro você empilha os valores e depois realiza as operações. Assim, para realizar uma operação tão simples como “2*5+3” você terá que fazer “3 5 2 * +”, ou seja, “empilhar” os 3 valores e, em cada operação elementar dois desses valores são desempilhados e o resultado empilhado de volta. Assim, a multiplicação desempilhará os valores 2 e 5 e colocará 10 de volta na pilha, que ficará com “3 10” e a operação de adição desempilhará esses dois valores e colocará 13 na pilha.

Não é necessário que coloquemos todos os valores na pilha para depois realizarmos as operações. Poderíamos ter feito “2 5 * 3 +”, que dá na mesma e poupa um nível da pilha (usa apenas 2 níveis ao invés de 3). É importante lembrar que operações como divisão e subtração são sensíveis à ordem com que os argumentos são empilhados, ou seja, $2 - 3$ não é a mesma coisa que $3 - 2$, assim como $\frac{1}{2}$ não é a mesma coisa que $\frac{2}{1}$.

A pilha do x87

A pilha do co-processador matemático é circular e pode conter até 8 valores em ponto flutuante. Por “pilha circular” quero dizer que, ao empilhar o 9º item, o primeiro será perdido.

O topo da pilha é sempre conhecido como ST(0) e os valores empilhados anteriormente como ST(1), ST(2), …, até ST(7). Por exemplo, considerando uma pilha vazia, se tivermos a sequência de instruções abaixo, ST(0) e ST(1) conterão os valores armazenados nas variáveis *x* e *y*, respectivamente (*ld* é abreviação de *load*, ou “carga”):

```
fld dword [y]   ; Empilha “y”. ST(0) = y
fld dword [x]   ; Empilha “x”. ST(1) = ST(0) e ST(0) = x.
```

Uma vez empilhados os valores podemos usar operações aritméticas como *fadd*. Ela realizará a adição de ST(0) com ST(1) e colocará o valor sobre o ST(0):

```
fadd            ; ST(0) = ST(0) + ST(1)
```

Não é necessário empilharmos todo os operandos, apenas o primeiro. Se tivéssemos empilhado o valor de *x* e usado a instrução *fadd* deste jeito:

```
fld  dword [x]  ; ST(0) = x
fadd dword [y]  ; ST(0) = ST(0) + y
```

Funcionaria do mesmo jeito. A diferença é que, no primeiro caso onde usamos *fadd* sem operandos, teríamos uma pilha com 2 níveis usados. Seria necessário realizar uma operação de POP que se livrasse de ST(1) – o que não é possível, já que a operação POP livra-se do conteúdo do topo da pilha, ST(0), **sempre**! A alternativa seria fazer algo assim:

```
fld dword [y]
fld dword [x]
fadd           ; ST(0) = ST(0) + ST(1).
fxch           ; Troca ST(0) com ST(1).
fincstp        ; Incrementa o topo da pilha, fazendo ST(1) ser o novo ST(0).
```

Para evitar esse tipo de complicação, existem variações das instruções aritméticas que fazem o POP de ST(0) automaticamente, logo depois da operação. É o caso de *faddp*[25]:

```
fld dword [y]
fld dword [x]
faddp st1,st0  ; Faz: ST(1) = ST(1) + ST(0) e desempilha ST(0),
               ; fazendo do ST(1) o novo ST(0)...
```

25 Esse “p” no fim do mnemônico significa exatamente “pop” ou “pull operation”.

Note que a ordem em que os operandos são usados é invertida. Isso porque, ao final, *faddp* vai fazer o POP e livrar-se de ST(0).

> Do mesmo jeito que acontece com *fadd*, a instrução *faddp* pode ser usada sem parâmetros, mas a ordem dos operandos é invertida por causa do POP.
>
> O fato de podermos especificar os pseudo-registradores nessas instruções nos permite acessar qualquer item da pilha. Se tivéssemos 3 itens na pilha, poderíamos fazer algo assim:
>
> ```
> fadd st0,st2   ; ST(0) = ST(0) + ST(2)
> ```
>
> A maioria das instruções aritméticas tem como operando destino obrigatório o ST(0), exceto aquelas em que POP é conduzido depois da operação. Neste caso, o operando fonte é que é ST(0) obrigatóriamente.
>
> Repare também que, no NASM, o nome dos pseudo-registradores que usamos para lidar com a pilha do co-processador não é ST(n), mas STn. Essa é uma exigência do NASM.

Exemplo de calculos com o x87

Eis um caso real, obtido através da compilação de uma pequena função em linguagem C:

```
; Rotina para i386 equivalente de:
;   float f(float a, float b, float c)
;   { return a * (1.0f + b/c); }
f:
  fld   dword [c]        ; st(0) = c
  fdivr dword [b]        ; st(0) = b / st(0)
  fld1                   ; st(1) = st(0), st(0) = 1.0
  faddp st1,st0          ; st(1) = st(1) + st(0) e pop.
  fmul  dword [a]        ; st(0) = st(0) * a
  ret
```

Graças à característica circular da pilha do co-processador não precisamos limpá-la, mas pode ser interessante não deixá-la poluida (por isso o uso do *faddp* acima). Se eu tivesse usado *fadd* a rotina funcionaria perfeitaemnte, mas teríamos o resultado em st(0) e um resultado parcial em st(1) – que é ignorado pelo compilador C...

A necessidade de manter apenas um item na pilha tem haver com algumas otimizações inter-chamadas que o compilador pode fazer. Em teoria, tudo o que a convenção de chamada exige, no modo i386, é que o valor retornado esteja em st(0).

O x87 tem algumas constantes "built-in"

Reparou na instrução *fld1*? Algumas constantes muito usadas estão incorporadas no co-processador matemático e são empilhadas via instruções de carga especiais.

Os compiladores C tendem a **não** usar essas instruções. Nem mesmo para valores evidentes como 0.0 e 1.0.

Eis as instruções:

Instrução	Constante
fldz	0.0
fld1	1.0
fldl2t	$\log_2 10$ = 3.321928095...
fldl2e	$\log_2 e$ = 1.442695041...
fldpi	π = 3.141592654...
fldlg2	$\log_{10} 2$ = 0.301029996...
fldln2	$\log_e 2$ = 0.693147181...

Tabela 1: Constantes "built-in" no x87

A instrução *fld1* na listagem anterior tende a ser substituída, pelo GCC, pela carga explícita de uma constante mantida na memória. A função do tópico anterior ficaria mais ou menos assim:

```
section .rodata
const1:   dd 1.0

section .text

f:
  fld   dword [c]         ; st(0) = c
  fdivr dword [b]         ; st(0) = b / st(0)
  fld   dword [const1]    ; st(1) = st(0), st(0) = 1.0f
  faddp                   ; st(1) = st(1) + st(0), pop stack.
  fmul  dword [a]         ; st(0) = st(0) * a
  ret
```

Uma ventagem de usar essas constantes *buiit-in*, no caso do *x87* é que a máxima precisão (*long double*) será usada na carga. Mas, nem sempre essa "máxima precisão" é o ideal. No GCC, por exemplo, a constante M_PI, definida em *math.h* tem o valor de 3.14159265358979323846, que tem precisão um pouco maior que a "máxima" (21 algarismos, contra 19 do *long double*). O compilador arredondará isso, é claro, mas o ponto é que o x87 pode ter algum bug ou inicializar uma posição de sua pilha com um valor de π com menos precisão, dependendo de como o coprocessador esteja configurado.

Equações "built in"

O *x87* também possui algumas instruções para calculos simples como *seno*, *cosseno*, *tangente*, *arco-tangente*, *raiz quadrada*, *obter valor absoluto*, *arredondamento*[26]. E todas elas são usadas pelo compilador C. Mas, existem algumas mais especializadas que **não** são:

Instrução	Equação
f2xm1	st(0) = 2 · st(0) - 1
fyl2x	st(1) = st(1) · $\log_2$ st(0). E faz "pop" da pilha.
fyl2xp1	st(1) = st(1) · $\log_2$(st(0) + 1). E faz "pop" da pilha.

Table 2: Equações não usadas pelo GCC

Desconfio que ele não as use por serem lentas, embora pareçam bem úteis no caso desses tipos de equações serem necessárias...

26 A diferenciação entre as funções *ceil* e *floor* é feita através da reprogramação do arredondamento do *fp87*.

Fazendo comparações no modo x87

O que acontece quando comparamos duas variáveis do tipo *float*?

```
if (x < y) …
```

Lembra-se que na aritmética inteira, dependendo se os valores *x* e *y* têm ou não sinal, usamos os flags OF e SF ou CF, respectivamente? Ocorre que os valores em ponto flutuante não usam o modelo de **complemento 2** para codificar os valores armazenados nessa estrutura. A mantissa é **sempre** positiva. O sinal do valor é dado pelo bit de mais alta ordem da estrutura usada para armazenamento do tipo. Esse sinal é usado internamente, é claro, mas isso significa que os flags OF e SF (*overflow* e sinal) **não** são usados em comparações com ponto flutuante. De fato, durante uma comparação, o processador irá zerá-los.

A instrução *fcomi* faz exatamente o que uma instrução CMP faz, só que para valores em ponto flutuante. A tabela abaixo nos mostra como CF e ZF são afetados nas comparações possíveis:

Comparação	**ZF**	**CF**	**PF**
st(0) < valor2	0	1	0
st(0) == valor2	1	0	0
st(0) > valor2	0	0	0
inválida	1	1	1

Tabela 2: Como ZF e CF são afetados por fcomi

Uma vez que *fcomi* afeta os flags, esses podem ser usados junto com as instruções de salto condicional para a tomada de decisão do que fazer em seguida.

Mas, há um problema... A estrutura de um ponto flutuante podem conter valores que são *não-números*, como NaN e infinitos e realizar comparações com esses valores gera um resultado inválido. Para resolver esse problema temos duas opções: Uma delas é usar a instrução *fucomi* que gerará uma exceção no caso de comparações inválidas. Neste caso é necessário capturar a exceção (um *signal*, no caso do POSIX, talvez?!). A segunda opção é verificar se ambos os flags ZF e CF estão setados (veja a opção “inválida” na tabela acima). Essa é uma situação teóricamente impossível, já que uma subtração resultar num valor zerado que gere um *carry* nunca ocorreria com valores válidos. Felizmente *fcomi* também afeta o flag PF de forma que, quando a comparação for inválida, PF seja setado.

Assim, depois de comparar valores em ponto flutuante, deve-se usar apenas saltos condicionais do tipo JZ, JC, JA, JAE, JB ou JBE e seus inversos (JNZ, JNC, …) e tomar cuidado com as comparações inválidas, usando JP ou JNP para determinar se houve um erro.

Saltos como JNL e JGE, depois de uma comparação desse tipo, passam a ser saltos *incondicionais* com um grave problema de performance (o salto **sempre** será feito, já que SF == OF). Instruções como JL e JNGE **nunca** farão o salto já que SF ≠ OF.

Usando SIMD no modo escalar

Ao invés de usar uma pilha para armazenamento de valores em ponto flutuante, as arquiteturas mais modernas usam registradores especializados. Eles são nomeados de XMM e têm 128 bits de tamanho. Cada um desses registradors suporta o armazenamento de até 4 valores do tipo *float* em seu interior (ou 2 valores do tipo *double*).

No modo i386 temos 8 registradores XMM, nomeados de XMM0 até XMM7. No modo x86-64 existem 16 registradores deste tipo: de XMM0 até XMM15.

O processador usa um conjunto de instruções estendidas para lidar com esses registradores, chamadas de SSE (de *Streamming SIMD Extension*). Com essas extensões é possível realizar operações com vários valores simultâneamente (dependendo do tamanho desses valores). Por exemplo: Podemos subtrair 4 *floats* de outros 4 *floats* de uma tacada só. Daí o nome SIMD (*Single Instruction, Multiple Data*).

> Nos primórdios do SIMD, na plataforma Intel, tinhamos a especificação MMX (de *MultiMedia eXtension*). Além de ser um nome derivado puramente de *marketing*[27], este padrão era limitado a 64 bits e usava a pilha do co-processador matemático como se ela fosse um conjunto de 8 registradores (`mm0` até `mm7`).
>
> Não foi lá uma ideia muito boa... Primeiro porque esses registradores não permitiam o uso de mais que um valor em ponto flutuante em seu interior e, em segundo lugar, as instruções MMX usavam um conjunto de regras diferentes do que as funções do x87. Ou seja, não era possível usar a pilha do x87 e as instruções MMX ao mesmo tempo! Ou, pelo menos, não de forma fácil! Com SSE (e sua versão mais poderosa, o AVX) o co-processador é um recurso à parte e pode ser usado sem medo de interferências.
>
> MMX ainda existe no interior dos processadores modernos, mas é recomendável **não** usá-lo! Por isso nem vou falar dele aqui.

SSE tem dois modelos: Escalar e vetorizado. O modelo “escalar”, SSE só lida com um único valor em ponto flutuante de cada vez em seus registradores. O modo “vetorizado” será discutido no próximo capítulo.

Todas as instruções que lidam com escalares *floats* são “scalar single”, sufixadas com “ss” e as que lidam com escalares *double*, com “sd”. Assim, uma instrução que carrega um registrador XMM com um *float* a partir de alguma variável contida na memória será algo assim:

```
movss xmm0,[var]
```

Operações como *add*, *sub*, *mul* e *div* seguem exatamente a mesma regra... Coloca-se “ss” ou “sd” de acordo com a precisão desejada e os operandos podem ser registradores XMM e/ou ponteiros, seguindo as mesmas regras gerais das instruções comuns. Mas, saiba que existem muitas outras instruções interessantes: raiz quadrada, seno, cosseno, recíproco ($\frac{1}{x}$) e, no caso de usarmos vetores (tipos não escalares) temos até mesmo produto escalar (“dot product”)... Existem até mesmo instruções que não estão presentes na modalidade *x87* como o recíproco da raiz quadrada de um número[28]:

$$rsqrt(x) = \frac{1}{\sqrt{x}}$$

Pode parecer que não há muita diferença entre SSE e x87 no que se refere à manipulação de valores escalares, mas a vantagem é óbvia se você perceber que, agora, não precisamos nos preocupar com

27 A sigla SSE **também é** uma jogada de marketing... A sigla significa *Streamming SIMD Extension* e o “streamming” aqui insinua que esse modo serve para lidar com vídeo, áudio ou fluxo de dados para rede. Na realidade essas extensões estão mais ligadas ao conceito matemático de *vetores* e, por isso, quando usamos o poder total do SIMD dizemos que temos um código *vetorizado*.

28 O motivo para a existência dessa instrução no SSE é que extrair a raiz quadrada e depois calcular a reciprocidade, na modalidade fp87, implica em grande perda de precisão quando *x* é pequeno.

a ordem em que tínhamos que empilhar os valores. Além disso, se precisarmos usar um determinado registrador que já esteja em uso, podemos muito bem gravar seu conteúdo em algum lugar da memória e recuperá-lo mais tarde. Isso pode ser problemático quando usamos pilhas... Na arquitetura x86-64 os valores em ponto flutuante são mapeados diretamente em registradores XMM.

Comparação de valores escalares usando SSE

Da mesma forma que no *x87*, usamos uma instrução de comparação de escalares que alterará os flags ZF e CF e PF apenas. A instrução *ucomss* compara os escalares no índice 0 de dois registradores XMM e seta esses flags de acordo. Novamente os flags OF e SF serão zerados.

Existe uma instrução *comss*, mas ao invés de apenas setar os flags, ela gera uma exceção em casos de comparações inválidas.

Convertendo valores

Os registradores XMM suportam alguns formatos diferentes. Já vimos que podemos lidar com eles, no modelo vetorizado, como se contivessem 4 *floats*, mas podemos usá-los para conter, também, 4 *doublewords*, 2 *quadwords*, 2 *doubles* ou outros valores (8 words, 16 bytes,...). Existem instruções para converter valores nos diversos formatos.

Por exemplo: Se tivermos 4 *quadwords* e quisermos converter esses valores inteiros para 4 *floats*, podemos usar a instrução CVTDQ2PS:

```
; Neste ponto xmm1 tem 4 dwords
cvtdq2ps xmm0,xmm1
; Neste ponto xmm0 terá 4 floats.
```

Que mnemônico estranho, não? A estranheza existe porque a instrução CVTPI2PS (*Convert Packed Integers to Packed Singles*) não faz o que você espera... Ela converte apenas os dois primeiros *doublewords* do operando fonte para 2 *floats* do operando destino.

A maioria desses mnemônicos de conversão é bem fácil de entender. É o exemplo de CVTSI2SS (*Convert Scalar Integer to Scalar Single*):

```
; Converte o valor em EAX para float em xmm0
cvtsi2ss xmm0,eax
```

No contexto do uso de SSE para manipular valores escalares temos apenas duas instruções que lidam com *floats*: CVTSS2SI e CVTSI2SS. Para lidarmos com *doubles* basta substituir o “SS” por “SD”.

O tipo “long double”

A única vantagem de usar o *x87* nos dias de hoje está relacionada ao tipo *long double*. O método que usa SIMD pode ser bastante poderoso, como continuaremos vendo no próximo capítulo, mas é incapaz de lidar com o tipo estendido de 80 bits... Somente o co-processador matemático pode fazê-lo.

Eis um exemplo de uma função usando o tipo *long double* no modo x86-64:

```
; Rotina equivalente a:
;   long double divide(long double a, long double b) { return a/b; }
divide:
  fld  tbyte [a]
  fdiv tbyte [b]
  ret
```

Nada diferente do modo i386, não é?

A dica aqui é: Evite usar o tipo *long double* no modo x86-64. Isso vai gerar código bem ineficiente em relação aos outros tipos de ponto flutuante.

Capítulo 6 – SIMD Vetorizado

Desde o lançamento do Pentium III a Intel decidiu incorporar em seus processadores um conjunto de instruções e registradores que ela chamou de *Streamming SIMD Extension*. Como o nome diz é uma extensão aos recursos do processadores. Mais uma vez, assim como fez com o nome MMX, o termo *Streamming* é uma "jogada de marketing". Essa extensão era, segundo a Intel, dedicada a oferecer suporte, em hardware, para aplicações gráficas e de multimedia. Não é nada disso, é claro...

SIMD significa apenas uma coisa: Podemos lidar com vários dados em uma única instrução.

Realizando 4 operações com floats em uma tacada só

A ideia por trás do SIMD (e do SSE) é que 4 valores codificados como *floats* (com 32 bits de tamanho) podem ser "enpacotados" (packed) num mesmo registrador. O termo "packed" é importante aqui, já que SSE também pode ser usado para realizar operações com valores simples (single), como mostrei no capítulo anterior.

Os registradores XMM são particionados em quatro pedaços de 32 bits e, portanto, são registradores de 128 bits de tamanho. É conveniente tratar esses registradores como se fossem um array de 4 *floats*, onde os 32 bits inferiores são alocados para o primeiro *float* desse array (tendo o índice 0), seguido dos *floats* de mais alta ordem (índices 1, 2 e 3).

Em C temos extensões do compilador que lidam com esses tipos de 128 bits, separados em floats, nomeados de *__m128*. Inicializá-los é muito simples:

```
__m128 x = { 1.0f, 2.0f, 3.0f, 4.0f };
```

Assim como no modo escalar, podemos usar os registradores XMM para conter *doubles*, mas neste caso apenas dois *doubles* cabem nele. Do ponto de vista da linguagem C o tipo que pode ser usado para representar esse empacotamento é o *__m128d* e o inicializamos da mesma forma:

```
__m128d x = { 1.0, 2.0 };
```

No texto deste livro estou interessado em mostrar apenas o empacotamento com o tipo *float* e, por isso, todos os exemplos usam *__m128*. Mas saiba que também existe o tipo *__m128i*, usado para empacotar diversos tipos de inteiros.

A inicialização inicial, do ponto de vista do assembly, pode muito bem ser codificado como:

```
section .data

global x
align 16
x:  dd  1.0
    dd  2.0
    dd  3.0
    dd  4.0
```

O valor 1.0 está localizado no primeiro item desse "array", seguido de 2.0, 3.0 e 4.0, nesta ordem...

Repare que o tipo *__m128* é **sempre** alinhado por "paragrafo", ou seja, na fronteira de 16 bytes. Isso significa que o endereço inicial de um tipo *__m128* tem, necessariamente, os 4 bits inferiores zerados. Esse é um requerimento essencial na arquitetura x86 para que a performance das operações com SSE não gastem ciclos extras, ou seja, para garantir a máxima performance. Portanto, na conversão para *assembly* a diretiva "align 16" torna-se necessária para garantir esse alinhamento.

Isso não significa que esses quatro *floats* não possam fugir desse alinhamento. Eles podem! Mas,

com isso, uma operação de carga de registrador, como:

```
movups xmm0,[x]
```

Gastaria um ciclo adicional por causa do desalinhamento.

Notou o “ps” depois do “mov”? O “p” vem de *packed*, ou “empacotado”. Essa instrução, diferente de *movss* (que carrega um simples float), carregará 4 floats do endereço de *x* para dentro do registrador `xmm0`. A letra “u” significa “unaligned”, então essa função tentará carregar os 4 floats a partir de um endereço que pode estar desalinhado.

Um aviso sobre a carga de registradores XMM via MOVAPS

Além da instrução MOVUPS, SSE disponibiliza a instrução MOVAPS que tenta carregar um *packed single* **alinhado**. Essa última instrução causará uma exceção do tipo *segmentation fault* se, num de seus operandos, topar com um endereço fora de alinhamento dos 16 bytes (se um ponteiro não tiver os 4 bits inferiores zerados).

MOVUPS é mais lenta que MOVAPS quando lidamos com carga a partir de ou para a memória.

Embaralhando um registrador

Além das operações aritméticas que podem ser feitas com esses 4 *floats* contidos num registrador XMM, podemos manipular esse array segundo alguns conceitos básicos. Um deles é o “embaralhamento”.

Imagine que tenhamos os valores {x,y,z,w} dentro de um registrador XMM e queremos trocar os itens *x* e *y* de lugar (*x* irá para a posição de *y* e este último para a posição do primeiro). Segundo nossa regra de particionamento de um registrador XMM o valor *x* está na posição 0 do array e o valor *y* na posição 1. Como temos apenas 4 *floats* num desses registradores, os índices só podem variar entre 0 e 3 e sabemos que esses valores podem ser expressos em apenas 2 bits...

A instrução SHUFPS (*Shuffle Packed Singles*) toma três parâmetros: O operadando destino, o fonte e um valor imediato de 8 bits. Cada dois bits desse valor expressa como *shufps* copiará os valores contidos nos dois operandos para o destino...

Por exemplo, para trocar *x* por *y*, contidos no registrador XMM0, podemos usar a instrução assim:

```
shufps xmm0,xmm0,0xe1
```

Na constante 0xE1 cada dois bits desse valor correspondem ao índice de cada um dos 4 *floats*. Em binário o valor é expresso como $1110.0001_{(2)}$. Note que separei os 4 bits superiores do valor com um ponto. Fiz isso porque os 4 bits superiores especificam os índices do operando *fonte* (o segundo operando) e os 4 bits inferiores especificam os índices do operando destino (o primeiro operando). Ou seja, a ordem dos bits correspondem a ordem com que os operandos aparecem na instrução. A posição dos grupos de 2 bits, neste valor, corresponde exatamente à posição do array em que o resultado da instrução ocorrerá (no operando destino).

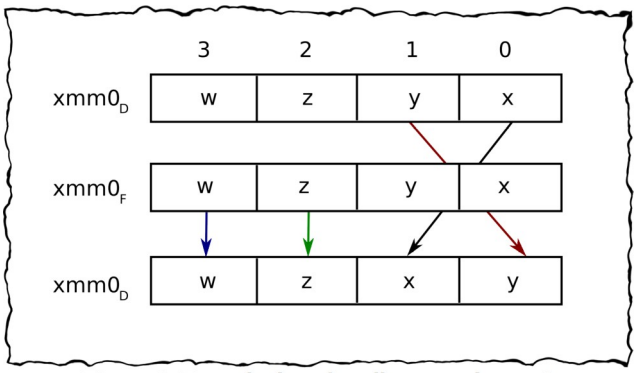


Figura 1: Exemplo de embaralhamento de xmm0

Do valor imediato usado em SHUFPS, expresso em binário, os bits em preto correspondem ao índice 0 do array e, como temos o valor $01_{(2)}$, estamos dizendo à instrução de embaralhamento: "Peque o conteúdo do índice 1 (valor $01_{(2)}$) do array de *floats* do operando fonte e coloque na posição do índice 0 (os dois bits estão no grupo 0, mais à direita, de dois bits do valor) do operando destino". Da mesma forma, os bits em azul dizem para copiar o *float* nº 0, do operando fonte, para o float nº 1. O grupo verde copiará o 2º float do operando destino para o 2º float do operando destino e o terceiro fará o mesmo...

A figura acima ilustra exatamente o que esse valor 0xE1 significa. $XMM0_D$ destino e $XMM0_F$ é o fonte. Usei o mesmo código de cores do valor binário e os índices numéricos, acima dos blocos, correspondem aos índices dos *floats* no interior dos registradores.

Essa instrução de embaralhamento pode parecer muito exotérica porque os operandos fonte e destinos podem ser destintos, permitindo um "embaralhamento" até mais elaborado.

Exemplo de uso de SSE: O produto vetorial

Eis um exemplo de cálculo do *produto vetorial* entre dois vetores tridimensionais:

```
void cross(float *out, const float *v1, const float *v2)
{
  out[0] = (v1[1] * v2[2]) – (v1[2] * v2[1]);
  out[1] = (v1[2] * v2[0]) - (v1[0] * v2[2]);
  out[2] = (v1[0] * v2[1]) - (v1[1] * v2[0]);
}
```

Essa função, em assembly para x86-64, **sem** o uso de vetorização, pode ser expressa assim:

```
<cross>:
   0:   f3 0f 10 46 04        movss  xmm0,DWORD PTR [rsi+4]
   5:   f3 0f 10 4e 08        movss  xmm1,DWORD PTR [rsi+8]
   a:   f3 0f 59 42 08        mulss  xmm0,DWORD PTR [rdx+8]
   f:   f3 0f 59 4a 04        mulss  xmm1,DWORD PTR [rdx+4]
  14:   f3 0f 5c c1           subss  xmm0,xmm1
  18:   f3 0f 11 07           movss  DWORD PTR [rdi],xmm0
  1c:   f3 0f 10 46 08        movss  xmm0,DWORD PTR [rsi+8]
  21:   f3 0f 10 0e           movss  xmm1,DWORD PTR [rsi]
  25:   f3 0f 59 02           mulss  xmm0,DWORD PTR [rdx]
  29:   f3 0f 59 4a 08        mulss  xmm1,DWORD PTR [rdx+8]
  2e:   f3 0f 5c c1           subss  xmm0,xmm1
  32:   f3 0f 11 47 04        movss  DWORD PTR [rdi+0],xmm0
  37:   f3 0f 10 06           movss  xmm0,DWORD PTR [rsi]
  3b:   f3 0f 10 4e 04        movss  xmm1,DWORD PTR [rsi+4]
  40:   f3 0f 59 42 04        mulss  xmm0,DWORD PTR [rdx+4]
  45:   f3 0f 59 0a           mulss  xmm1,DWORD PTR [rdx]
  49:   f3 0f 5c c1           subss  xmm0,xmm1
  4d:   f3 0f 11 47 08        movss  DWORD PTR [rdi+8],xmm0
  52:   c3                    ret
```

UAU! Um monte de instruções e a rotina tem o tamanho de 83 bytes!! Agora, compare com uma

versão mais “complicada”, usando vetorização:

```
void cross_ps(float *out, const float *v1, const float *v2)
{
  __m128 t1, t2, a1, a2;

  t1 = _mm_load_ps(v1);
  t2 = _mm_load_ps(v2);

  a1 = _mm_mul_ps(
         _mm_shuffle_ps(t1, t1, _MM_SHUFFLE(3,0,2,1)),
         _mm_shuffle_ps(t2, t2, _MM_SHUFFLE(3,1,0,2))
       );

  a2 = _mm_mul_ps(
         _mm_shuffle_ps(t1, t1, _MM_SHUFFLE(3,1,0,2)),
         _mm_shuffle_ps(t2, t2, _MM_SHUFFLE(3,0,2,1))
       );

  _mm_store_ps(out, _mm_sub_ps(a1, a2));
}
```

O código, em assembly, ficará assim:

```
  0:   0f 28 0e              movaps xmm1,XMMWORD PTR [rsi]
  3:   0f 28 02              movaps xmm0,XMMWORD PTR [rdx]
  6:   0f 28 d9              movaps xmm3,xmm1
  9:   0f 28 d0              movaps xmm2,xmm0
  c:   0f c6 d9 c9           shufps xmm3,xmm1,0xc9
  10:  0f c6 d0 d2           shufps xmm2,xmm0,0xd2
  14:  0f c6 c9 d2           shufps xmm1,xmm1,0xd2
  18:  0f c6 c0 c9           shufps xmm0,xmm0,0xc9
  1c:  0f 59 d3              mulps  xmm2,xmm3
  1f:  0f 59 c1              mulps  xmm0,xmm1
  22:  0f 5c d0              subps  xmm2,xmm0
  25:  0f 29 17              movaps XMMWORD PTR [rdi],xmm2
  28:  c3                    ret
```

A tabela abaixo lista a quantidade de operações realizadas por ambos os códigos:

	Operações	
	Rotina escalar	**Rotina vetorizada**
Carga de registradores XMM via ponteiros *v1* e *v2*	6	2
Cópia de registradores	0	2
Embaralhamentos	0	4
Multiplicações*	6	2
Subtrações	3	1
Armazenamento do resultado via ponteiro *out*	3	1
Total de operações	**18**	**12**

Tabela 1: Comparação de operações das rotinas do produto vetorial

Olhando para o número de operações, você poderia supor que a rotina vetorizada é só 50% mais rápida que a escalar. Acontece que as multiplicações gastam mais tempo que qualquer outra instrução usada (por isso coloquei um asterisco nessa contagem). Como temos 3 vezes menos multiplicações na rotina vetorizada você também poderia supor que essa rotina é 3 vezes mais rápida que a rotina escalar, mas temos que considerar que ela tem 4 embaralhamentos e duas cópias de registradores (que não existem na outra rotina)... Assim, é razoavel supor que a nova função terá performance cerca de 100% maior, para nivelar por baixo... É claro que só saberemos se isso é verdadeiro se **medirmos** o tempo gasto de ambas as rotinas. Mostro como podemos fazer isso no meu outro livro “C & Assembly para arquitetura x86-64”.

De qualquer maneira, temos um ganho potencial de performance... E também de tamanho do código! A nova rotina, embora pareça maior que a original, em C, tem apenas 41 bytes de tamanho! Isso coloca menos “pressão” no cache e tem o potencial de não “desacelerar” a executção deste código com *cache misses*.

Desempacotamento, outro tipo de embaralhamento

A instrução SHUFPS é bem útil, mas existem outras que fazem um tipo especial de embaralhamento chamado de desempacotamento (unpacking). A ideia é pegar pedaços de dois operandos e colocá-los numa ordem específica no operando destino. Existem dois desempacotamentos: O que desempacota os itens inferiores (unpack low) e o que desempacota os itens superiores (unpack high). A figura que segue mostra como funciona as instruções *unpcklps* (*unpack low packed single*) e *unpckhps* (*unpack high packed single*):

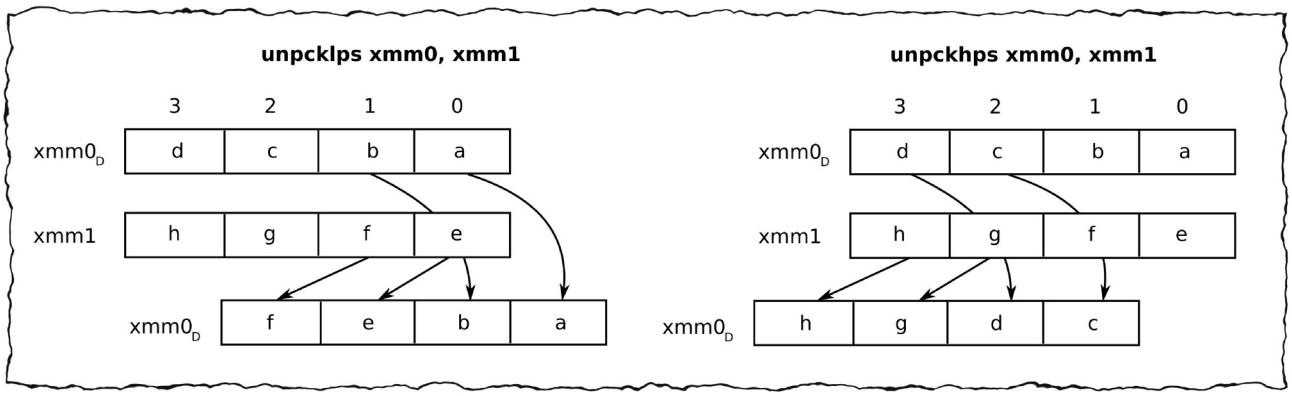


Figura 2: Instruções unpckhps e unpcklps

A vantagem é que, por essas instruções serem especializações de *shufps*, elas são um pouquinho menores (a codificação, em número de bytes, do μ_{op} é menor).

SSE 2, 3 e 4

Durante a escrita deste livro a última versão do SSE era a 4.2. SSE surgiu como uma maneira de lidar com *floats*, via SIMD. As novas versões aumentaram tanto a quantidade de instruções quanto o formato dos registradores XMM. A partir da versão 2 tornou-se possível lidar com 2 *doubles* dentro de um registrador XMM. Na versão 3 algumas instruções interessantes foram adicionadas: por exemplo, aquelas que permitem a adição e subtração horizontal dos itens de um registrador. Na versão 4 algumas instruções realmente especializadas surgiram, como o produto escalar vetorial e instruções vetoriais que só existiam no modo escalar.

A versão 4.2 adicionou também a instrução para cálculo de CRC32[29] e, em alguns processadores, a instrução para lidar com o padrão de criptografia AES (*Advances Encryption System*).

Eis um exemplo do uso da instrução de cálculo de produto escalar vetorial usando SSE (função *dot_sse*) e SSE 4.1 (função *dot_sse4*):

```
bits 64
section .text

; Protótipos em C:
;   float dot_sse(const float *a, const float *b);
;   float dot_sse4(const float *a, const float *b);
global dot_sse
global dot_sse4
```

29 Embora CRC32, do SSE 4.2, só seja realmente útil em aplicações de rede. O polinômio base da instrução não é o mesmo usado na maioria das aplicações que usam esse recurso!

```
; SSE tradicional
dot_sse:
  movaps   xmm0,[rdi]
  mulps    xmm0,[rsi]
  movaps   xmm1,xmm0
  unpckhps xmm1,xmm0
  addps    xmm0,xmm1
  movaps   xmm1,xmm0
  shufps   xmm1,xmm0,0xE5
  addps    xmm0,xmm1
  ret

; SSE 4.1
dot_sse4:
  movaps  xmm0,[rdi]
  dotps   xmm0,[rsi],0x71
  ret
```

Ou seja, uma única instrução em SSE 4.1 substitui 7...

SSE não é a única extensão SIMD

Também enquanto escrevo este livro, existem pelos menos duas outras extensões que valem nota: AVX e AVX2 (ou AVX-512). A sigla AVX significa *Advanced Vector eXtension* que, essencialemnte, é a mesma coisa que SSE, só que permite a manipulação de registradores ainda maiores.

O AVX lida com registradores de 256 bits nomeados de YMM0 até YMM15. Nesses registradores podemos colocar o dobro de valores escalares (8 *floats* ou 4 *doubles*). E o AVX-512, como o nome sugere, lida com registradores de 512 bits (nomeados de ZMM0 até ZMM31). Em ambos AVX os registradores são extensões dos XMM e, como você pode notar, no modo AVX-512 o número foi dobrado de 16 para 32 registradores.

Não vou falar dessas extensões aqui por dois motivos: Primeiro, ainda não tive acesso a um processador que implemente AVX-512 e, o outro motivo: O funcionamento da extensão AVX é muito similar à SSE, exceto que o uso deles não é parte da convenção de chamada usada pela arquitetura x86-64.

Capítulo 7 – Performance

Acho que ficou claro, até agora, que um compilador como o GCC faz um excelente trabalho do ponto de vista da performance das rotinas. A maioria dos fragmentos de códigos em assembly que vimos até agora foram obtidos através de rotinas similares escritas em C... Acontece que se você quiser espremer até o último ciclo de máquina de uma rotina terá que “fazer justiça com as próprias mãos” e aproveitar recursos do processador que o compilador C abstrai, para manter coerência entre a chamada de funções...

Contando ciclos

O que diabos é um ciclo de máquina? Ciclos de clock é fácil: O “clock” é uma onda quadrada que é usada como base de temporização para o processador. Ou seja, em cada período ou “pulso” dessa onda o processador faz “algo”. O diagrama abaixo ilustra esses ciclos e o comportamento de alguns sinais nos pinos do processdor *Pentium*. O significado disso não nos interessa muito:

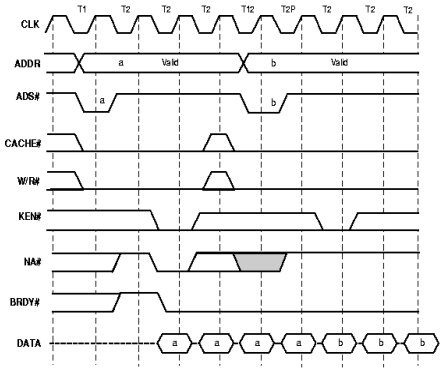


Figura 1: Exemplo de diagrama de temporização do processador

O que nos interessa é observar o primeiro sinal, CLK. Ele não parece uma onda “quadrada”, certo? O aspecto trapezoidal é apenas para lembrar ao engenheiro que a variação do sinal, dada a sua velocidade, pode ser significativa. Mas, podemos encarar essas variações como “verticais”, para facilitar o entendimento.

Um ciclo de máquina, por outro lado, é um conjunto de ciclos de clock. Nos processadores antigos um “ciclo de máquina” equivalia a quatro ciclos de clock. Isso foi simplificado nos processadores mais modernos através da multiplicação da frequência do clock, internamente. Assim, um ciclo de máquina passou a ser sinônimo de “ciclo de clock”.

Latência e throughput

Existem duas medidas de “velocidade de execução” de uma instrução em linguagem de máquina. A primeira é a *latência* e tem haver com o tempo gasto para reintroduzir a mesma introdução novamente no *stream* de execução de instruções. A outra é o *throughput* e nos diz o tempo real de execução da instrução isolada.

Esses dois tempos são medidos em ciclos de clock. Mas não a previsão do gasto de clocks de ums instrução não é feita meramente olhando para esses valores numa tabela. Existem várias circunstâncias que alteram esses tempos. Uma delas é a reordenação das instruções feitas dentro do próprio processador.

A reordenação de instruções ocorre para manter as “portas” da unidade de execução ocupadas por mais tempo possível. Quer dizer, para que as “portas” não fiquem ociosas com tanta frequência. Cada porta é responsável pela execução de uma única instrução.

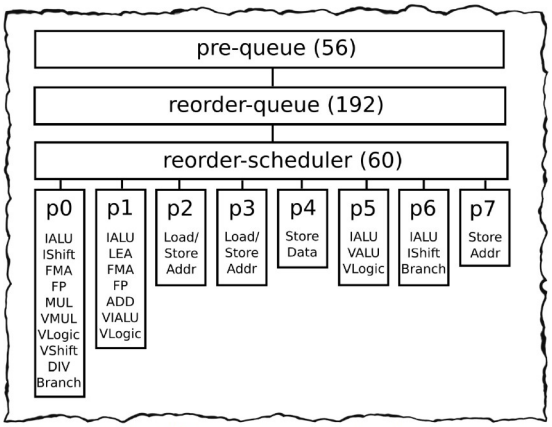


Figura 2: Simplificação das unidades de execução da arquitetura Haswell

Na arquitetura *Haswell*, as portas de 0,1,5 e 6 podem executar instruções aritméticas e lógicas inteiras, por exemplo. As portas 2 e 3 podem fazer cálculos de endereços efetivos e tranferências de e para a memória (caches). As portas 0 e 6 podem realizar saltos e assim por diante. É tarefa do *reorder scheduler* reorganizar grupos de até 60 instruções na fila do *scheduler* de reordem, que as obtém de outra fila, que pode conter até 192, de maneira a manter todas essas portas funcionando...

Normalmente, quando falamos de tempo de execução lidamos com o termo *throughput*. Ele geralmente é bem menor que o tempo de latência e é o tempo que uma das portas leva para executar a instrução. Para a nossa sorte, a maioria das instruções na família Intel, especialmente desde o processador Pentium, têm *thoughput* de apenas 1 ciclo de clock... Algumas instruções gastam apenas ½ ciclo (como MOV entre registradores) e até ¼ de ciclo (como a maioria das instruções aritméticas e lógicas que não envolvam acesso à memória). Existem circunstâncias até onde uma instrução não gasta ciclo de clock algum!

Instruções mais complicadas podem gastar bem mais tempo. É o caso de DIV e IDIV, que gastam de 23 a 30 ciclos (thoughput).

Outro detalhe está relacionado aos operadores das instruções... Sempre que um deles faz referência à memória (uso de ponteiros), um ou mais ciclos têm que ser adicionado ao *throughput*. Se for um operador destino que também seja usado como fonte, mais outro ciclo é adicionado. Como no exemplo abaixo:

```
add [edi],eax      ; Gasta 2.5 ciclos, pelo menos.
```

A instrução ADD gasta ½ ciclo, mas o operador destino, além de ser um ponteiro, também é usado como fonte e destino, então 2 ciclos são adicionados: Um para a leitura e outro para a escrita do dado apontado... Mas, espere um pouco! E quanto às portas que lidam com carga e armazenamento (load and store)? Mais adiante veremos que esses dois ciclos adicionais **podem** não acontecer por causa dessas portas!

A correlação entre especialização e lentidão é quase sempre verdadeira, exceto para as instruções MUL e IMUL, que gastam de 1 a 5 ciclos. Um IDIV, por exemplo, gasta entre 23 a 30 ciclos, dependendo dos valores contidos nos registradores RDX:RAX e no operando da instrução. O guia de otimização da Intel nos diz que se RDX for diferente de zero e a divisão for feita com um divisor de 64 bits, essas instruções não gastarão menos que 30 ciclos.

Outras instruções, como RDRAND e CPUID podem gastar 100 ciclos ou mais (CPUID com

EAX≠0 gasta mais que 200 ciclos, segundo a Intel).

Outras penalidades podem ser aplicadas ao *throughput* de uma instrução. O cálculo de endereço efetivo raramente é um deles já que existe um recurso nos processadores modernos chamado de *micro-fusão*. No entanto, o acesso ao dado apontado sobre a influência do estado atual dos caches. Um cache "esformeado" e, pior, uma falta de página (*page fault*), adiciona dezenas, senão centenas ou milhares de ciclos na execução de uma instrução.

A execução paralela de instruções nas "portas" da unidade de execução tem, também um efeito interessante: Mesmo que até 8 instruções possam ser executadas simultaneamente, o tempo gasto por todas as oito é sempre medido com base no *thoughput* da instrução **mais lenta**. Por exemplo, suponha que as 4 instruções abaixo sejam executadas ao mesmo tempo:

```
mov eax,[var1]     ; gasta 1 ciclo.
xor edx,edx        ; gasta ¼ de ciclo.
div ecx            ; gasta 20 ciclos (no mínimo!).
mov [ebp-4],eax    ; gasta 1 ciclo.
```

Como elas estão sendo executadas simultaneamente, gastam juntas 20 ciclos de máquina! Neste caso...

Esse comportamento é algo que você não conseguirá facilmente contornar... Lembre-se que o processador reordena as instruções que estão na fila de reordenação **automaticamente**. Ele tenta manter as portas ocupadas, mas nada pode fazer para isolar ou misturar instruções com grande *throughput* da forma como você, programador, escolher...

Esses são alguns motivos pelos quais "contar ciclos" não é tão simples assim... Você pode evitar usar instruções "gulosas" ao máximo, mas terá que **medir** a performance de suas rotinas de qualquer jeito...

Para obter a tabela com a latência e *throughput* de cada uma das instruções dos processadores, recomendo o download do manual "*Intel® 64 and IA-32 Architectures Optimization Reference Manual*", no site da Intel. Para uma melhor discussão sobre o funcionamento dos caches e do sistema de paginação da arquitetura x86-64, consulte meu outro livro.

Micro-fusão e macro-fusão

Graças ao *scheduler* que citei no tópico anterior os processadores mais modernos permitem a fusão de suas ou mais instruções em uma só. A "micro-fusão" acontece, essencialmente, quando temos instruções que lidam com endereçamento. Instruções como:

```
add eax,[rsi+3]
```

Na realidade são três instruções em uma só: O cálculo do endereço efetivo (RSI+3), a leitura do *dword* da memória e a adição desse valor lido ao registrador EAX. O processador "funde" essas três operações em uma só. Mas Isso não nos ajuda muito... As instruções cujo operando destino seja um endereço, como em:

```
add [rsi+3],eax
```

Têm um ganho de performance considerável com a micro-fusão, já que existem unidades especializadas em "armazenamento" e "carga" nos processadores modernos. Mas perde alguns ciclos porque o operando é fonte e destino, ao mesmo tempo.

O que nos interessa mesmo são as "macro-fusões". Neste caso, duas instruções diferentes podem ser misturadas em uma só pelo processador. É o caso de algumas operações aritméticas e lógicas seguidas de saltos condicionais.

No caso da instrução CMP, se um salto condicional **sem sinal** for tomado, as instruções CMP e os saltos usando ZF ou CF são mesclados numa única instrução. É o caso de JZ, JNZ, JC, JNZ, JA, JAE, JB e JBE.

No caso de TEST, isso inclui os flags de sinal e overflow. Assim, TEST pode ser macro-fundida com os saltos listados acima e também JL, JLE, JG e JGE. Já as instruções ADD, SUB, INC e DEC também podem ser fundidas com os mesmos saltos, exceto INC e DEC, que só se fundem com JZ, JNZ, JL, JLE, JG e JGE (note que, se o flag CF também for testado não há fusão!).

Assim, em C, é mais performático termos um loop usando uma variável de controle **não** sinalizada, como em:

```
unsigned int i;
for (i = 0; i < N; i++) { … faz algo aqui … }
```

Porque *i* será comparado com *N* via instrução CMP em algum ponto e um salto condicional será feito. Como não temos sinal, o compilador criará código que aproveita a macro-fusão!

Outro detalhe importante com as otimizações do GCC é que testes contra o valor zero geralmente são feitas usando a instrução TEST. Essa instrução é macro-fundida com quaisquer outros saltos condicionais sempre. Abaixo, o compilador preferirá a primeira hipótese do que a segunda:

```
test eax,eax
jz   .L1
-----%< corte aqui %<-----
cmp  eax,0
jz   .L1
```

E o motivo dele preferir a primeira hipótese de código não tem haver somente com a macro-fusão. Acontece que TEST, usado dessa maneira, é uma instrução de apenas dois bytes. Já CMP, nessa configuração, usa 6.

Dividir ou multiplicar?

É muito comum que o desenvolvedor escreva suas rotinas, baseadas em equações, de forma literal. Um exemplo disso é o cálculo de um vetor unitáio $\hat{u}$, calculado a partir de um vetor $\vec{v}$ qualquer de 3 dimensões:

$$\hat{u} = \frac{\vec{v}}{\sqrt{x^2+y^2+z^2}} = \left(\frac{v_x}{\sqrt{x^2+y^2+z^2}}, \frac{v_y}{\sqrt{x^2+y^2+z^2}}, \frac{v_z}{\sqrt{x^2+y^2+z^2}} \right)$$

Como cada um dos componentes de $\vec{v}$ têm que ser dividido pela magnitude do próprio vetor, é comum escrever a função dessa forma:

```
void normalize(float *vout, const float *vin)
{
  float length;

  length = sqrtf(vin[0] * vin[0] +
                 vin[1] * vin[1] +
                 vin[2] * vin[2]);

  /* Divisões são lentas! Executar 3, em sequência,
     causa problemas de performance! */
  vout[0] = vin[0] / length;
  vout[1] = vin[1] / length;
  vout[2] = vin[2] / length;
}
```

Deixando de lado a possibilidade do vetor $\vec{v_{in}}$ ter tamanho zero, note que realizamos 3 divisões. Acontece que a função acima pode também ser escrita assim:

```
void normalize(float *vout, const float *vin)
{
  float length;

  /* Uma divisão... */
  length = 1.0f / sqrtf(vin[0] * vin[0] +
                        vin[1] * vin[1] +
                        vin[2] * vin[2]);

  /* ... e três multiplicações! */
  vout[0] = vin[0] * length;
  vout[1] = vin[1] * length;
  vout[2] = vin[2] * length;
}
```

Qual é a diferença? Bem... Uma única divisão escalar em ponto flutuante, usando SSE, costuma consumir cerca de 6 ciclos, enquanto uma multiplicação consome apenas $\frac{1}{2}$. Ou seja, uma multiplicação é 12 vezes mais rápida do que uma divisão. Se a primeira rotina faz 3 divisões e a segunda as substitui por uma divisão e três multiplicações, é claro que a última tende a ser mais rápida que a anterior.

A segunda rotina pode ficar um pouco mais rápida se o compilador usar a instrução RSQRTSS (que calcula o recíproco de uma raíz quadrada). Ela gasta apenas 1 ciclo, enquanto SQRTSS gasta 7. Podemos forçar a barra para que o GCC use instruções recíprocas, quando lida com SSE, informando a opção *-ffast-math:*

Primeira rotina

```
normalize:
  movss xmm3,[rsi]
  movss xmm2,[rsi+4]
  movaps xmm0,xmm3
  movss xmm1,[rsi+8]
  mulss xmm0,xmm3
  mulss xmm2,xmm2
  mulss xmm1,xmm1
  addss xmm0,xmm2
  addss xmm0,xmm1
  sqrtss xmm1,xmm0      ; 7 ciclos

  divss xmm3,xmm1       ; 6 ciclos
  movss [rdi],xmm3
  movss xmm0,[rsi+4]
  divss xmm0,xmm1       ; 6 ciclos
  movss [rdi+4],xmm0
  movss xmm0,[rsi+8]
  divss xmm0,xmm1       ; 6 ciclos
  movss [rdi+8],xmm0

  ret
```

Segunda rotina (com *-ffast-math*)

```
normalize:
  movss xmm3,[rsi]
  movss xmm2,[rsi+4]
  movaps xmm0,xmm3
  movss xmm1,[rsi+8]
  mulss xmm0,xmm3
  mulss xmm2,xmm2
  mulss xmm1,xmm1
  addss xmm0,xmm2
  addss xmm0,xmm1
  rsqrtss xmm1,xmm0      ; 1 ciclos

  mulss xmm3,xmm1        ; ½  ciclo
  movss [rdi],xmm3
  movss xmm0,[rsi+4]
  mulss xmm0,xmm1        ; ½  ciclo
  movss [rdi+4],xmm0
  movss xmm0,[rsi+8]
  mulss xmm0,xmm1        ; ½ ciclo
  movss [rdi+8],xmm0

  ret
```

Coloquei-as lado a lado para assinalar, em vermelho, as diferenças. Somente substituindo a raiz quadrada pela instrução RSQRTSS e as divisões por multiplicações temos um ganho de velocidade num fator de quase 10 vezes nessas instruções!

Observe que as rotinas acima lidam com escalares. Podemos melhorar um bocado a performance usando *vetorização* e, se usarmos SSE 4.1, podemos usar a instrução DPPS para calcular o produto escalar antes de extrairmos a raiz quadrada... Deixo a rotina vetorizada, usando DPPS, aqui para sua avaliação:

```
normalize:
  movaps xmm0,[rsi]        ; 1 ciclo
  movaps xmm1,xmm0         ; 1 ciclo
  dpps   xmm2,xmm1,0x77    ; 2.5 ciclos
  rsqrt  xmm1,xmm2         ; 1 ciclo
  mulps  xmm0,xmm1         ; ½ ciclo
  movaps [rdi],xmm0        ; 1 ciclo
  ret
```

Essa rotina gasta apenas 7 ciclos, descontando o par de instruções CALL/RET. A rotinas anterior, otimizada com *-ffast-math*, gasta uns 20 ciclos, também sem o CALL/RET.

Se você não especificar a otimização *-ffast*-math o compilador continuará usando a instrução SQRTSS e fará uma divisão adicional para obter o recíproco. Mesmo assim, a segunda rotina em C ainda é mais rápida que a primeira.

Evite multiplicações e divisões inteiras!

Ao olhar as tabelas de *thoughput* do processador fica evidente que instruções como MUL e DIV são mais lentas que simples instruções aritméticas. Especialmente as rotinas de divisão. Usar o recurso de micro-fusão torna multiplicações simples bem rápidas. Por exemplo, suponha que tenhamos um valor em EAX e queremos multiplicá-lo por valores entre 2 até 10:

```
add eax,eax         ; EAX = EAX * 2

lea eax,[eax+2*eax] ; EAX = EAX * 3

lea eax,[4*eax]     ; EAX = EAX * 4

lea eax,[eax+4*eax] ; EAX = EAX * 5

lea eax,[eax+2*eax] ; EAX = EAX * 6
add eax,eax         ;

lea edx,[8*eax]     ; EAX = EAX * 7
sub eax,edx

lea eax,[8*eax]     ; EAX = EAX * 8

lea eax,[eax+8*eax] ; EAX = EAX * 9

lea eax,[eax+4*eax] ; EAX = EAX * 10
add eax,eax
```

Estes são apenas alguns exemplos e podem até ser feitos de forma diferente. Mas, note que o uso de LEA e a micro-fusão no cálculo do endereço efetivo e o uso adicional de alguns instrução aritmética, provavelmente serão executados em apenas 1 ciclo de clock. Uma multiplicação com MUL pode gastar até 3.

É claro que poderíamos efetuar multiplicações por fatores 2^n podem ser feitas com *shifts* para esquerda. As divisão com esses mesmos fatores é só questão de deslocamentos para o outro lado.

Ainda, muitas vezes queremos realizar divisões para obtermos o resto. E, algumas vezes, queremos o resto de uma divisão com divisores com valores 2^n. A maneira mais simples e super-rápida de fazer isso é através de uma simples instrução AND contra o valor $2^n - 1$:

```
and eax,0x7f    ; EAX = EAX % 128.
```

Note, no entanto, que esses macetes **não são a mesma coisa** que usar as instruções de multiplicação e divisão. Os flags afetados, se o são, têm semântica diferente. A instrução LEA, por exemplo, **não** afeta quaisquer flags e, no caso do cálculo do resto de divisões por 2^n, a instrução AND sempre

zerará o CF, por exemplo.

Outro detalhe é que as instruções de divisão podem causar exceções por divisão de um grande valor por um muito pequeno. É que DIV e IDIV dividem o par de registradores RDX:RAX (ou EDX:EAX, ou ainda DX:AX) pelo operando informado. O quociente é colocado em RAX e o resto em RDX. Se o quociente não couber em RAX, a instrução causará uma exceção... Isso não acontece com o AND.

Evite usar instruções para o coprocessador x87!

Essas instruções têm *thoughput* medonhos! A instrução FDIV gasta 30 ciclos quando usada com tipos *float*, compare com DIVSS, que gasta apenas 6. A raiz quadrada, via FSQRT gasta de 30 a 58 ciclos, já SQRTSS gasta apenas 7. A coisa fica ainda pior quando lidamos com instruções transcedentais: FSIN e FCOS gastam uns 120 ciclos cada.

Não é à toa que a convenção de chamada para x86-64 prefere o uso de SSE ao invés do x87. A performance do último é horrível!

Para comparar, eis a implementação da mesma rotina de normalização, otimizada, de um vetor tridimensional, mas para o x87, usando o tipo *float*:

```
.section .rodata

const_1  dd 1.0

.section .text

global normalize
normalize:
  fld DWORD PTR [rsi]
  fld DWORD PTR [rsi+4]
  fld DWORD PTR [rsi+8]

  fxch  st(1)
  fmul  st, st(0)
  fld st(2)
  fmul  st, st(3)
  faddp st(1), st
  fxch  st(1)
  fmul  st, st(0)
  faddp st(1), st
  fsqrt                    ; 30 ciclos, no mínimo

  fdivr DWORD PTR const_1  ; 30 ciclos, no mínimo.
  fmul  st(1), st
  fxch  st(1)
  fstp  DWORD PTR [rdi]

  fld DWORD PTR [rsi+4]
  fmul  st, st(1)
  fstp  DWORD PTR [rdi+4]

  fmul  DWORD PTR [rsi+8]
  fstp  DWORD PTR [rdi+8]
  ret
```

Só com as instruções FSQRT e FDIV já gastamos 60 ciclos, no mínimo. Lembre-se que pode chegar a 200! Usando SSE a rotina original gasta 20 e a otimizada com vetorização só 8!

Não use o coprocessador matemático! Fuja dele!

Nem sempre o compilador faz o que você quer

Quando estava depurando o exemplo do *produto escalar vetorial*, lá no capítulo sobre SSE, escrevi o seguinte programinha em C:

```
#include <x86intrin.h>

union xmm_e {
  __m128 x;
  float f[4];
};

float dot(const float *a, const float *b)
{
  union xmm_e r;

  r.x = _mm_dp_ps(_mm_load_ps(a), _mm_load_ps(b), 0x71);
  return r.f[0];
}
```

Minha esperança era que a última linha da função fosse ignorada na listagem assembly gerada pelo compilador, já que o registrador `xmm0` seria usado para conter o retorno e a função. Se só estamos interessados no valor escalar, pra que o compilador se preocuparia com os demais valores no registrador, certo?

Bem... eu tinha que ter certeza disso e então compilei o bicho e verifiquei o código em assembly:

```
; Fragmento de código, compilado com:
;   gcc -O3 -msse4.2 -S -masm=intel sse_test.c
dot:
  ; O compilador deveria ter usado xmm0 ao invés de xmm1!!!
  movaps xmm1,[rdi]
  dpps xmm1,[rsi],0x71   ; Essa rotina deveria terminar aqui!!!

  movaps [rsp-24],xmm1   ; Hã?! Mas que merda é essa?
  mov rax,[rsp-24]       ;
  mov [rsp-24],eax       ;
  movss xmm0,[rsp-24]    ;

  ret
```

Essas quatro linhas finais, antes do RET, além de serem inúteis, poderiam ser simplificadas pelo compilador por:

```
movaps [rsp-24],xmm1
movss  xmm0,[rsp-24]
```

Essa instrução MOVSS tomará conta de zerar os 3 *floats* superiores do registrador da mesma maneira que as quatro instruções na rotina original faria, efetuando dois acessos à memória a menos! Você poderia supor que essas mesmas instruções poderiam ser substituídas por:

```
movss xmm0,xmm1
```

Acontece que essa variação de MOVSS manterá os três floats superiores inalterados.

Bem que o compilador poderia ser esperto o suficiente para entender que se meu código só vai interpretar o primeiro *dword* de um registrador XMM como *float*, ele não precisa se preocupar com os demais! Felizmente, no caso do SSE e das funções intrínsecas, isso é fácil de resolver. A rotina abaixo faz exatamente o que eu queria:

```
float dot(const float *a, const float *b)
{
  return _mm_cvtss_f32( _mm_dp_ps(_mm_load_ps(a), _mm_load_ps(b), 0x71) );
}
----- corte aqui -----
; Rotina equivalente, gerada pelo GCC:
dot:
  movaps xmm0,[rdi]
  dpps   xmm0,[rsi],0x71
  ret
```

A dica aqui é **sempre** fique de olho no código que seu compilador gera...

Os efeitos do "branch prediction"

Antes, mostrei que compiladores de alto nível como o GCC tendem a colocar as comparações que são condições de permanência ou saída de loops no final. O motivo que dei era que dessa forma a condição era satisfeita a maior quantidade de vezes possível... E, por causa, disso e também do recurso implementado em processadores modernos chamado de "branch prediction" poupava 1 ciclo de máquina na execução do salto condicional.

O que é o "branch prediction"? Bem... Graças a natureza superescalar dos processadores modernos, as instruções lidas da memória são enfileiradas e decodificadas **antes** que sejam executadas. Para poder reorganizar as instruções o processador precisa ter como "prever" quando um salto vai ser executado **antes** que ele seja, de fato, executado.

Assim, o processador mantém um contador interno de quandos saltos **condicionais** foram executados e em que direção. Por que somente saltos condicionais? Porque os saltos incondicionais são facílimos de prever. Eles sempre são feitos!

A coisa funciona mais ou menos assim: Quando um salto condicional é encontrado o processador pergunta a si mesmo "quantos saltos deste tipo, e nessa direção (para frente ou para trás) eu já fiz ultimamente?". Se a resposta for "mais que um", então ele assume que o salto será realizado no futuro próximo. Com isso ele pode manter a maior quantidade possível de instruções na fila de pré-busca...

Isso funciona muito bem, se seu código tiver um comportamento previsível. Mas se saltos condicionais forem executados de forma aleatória (ou se mudarem de direção o tempo todo), então bagunçamos o algoritmo de previsão de saltos e, quanto um salto não feito quando deveria (ou vice-versa) o processador gastará 1 ciclo extra limpando a fila de pré-busca e preenchendo com um novo *stream* de instruções..

Para evitar esses gastos extras não só os loops devem ser criados de forma controlada, mas os *statements* do tipo *if...then...else* também! Uma sequência *if...then* nada mais é do que uma comparação seguida de um salto condicional onde a condição **contrária** salta por cima do código ligado ao *then*:

```
if (x == 3)
  x++;
----- corte aqui -----
; Código equivalente:
  cmp dword [x],3
  jne .L1           ; Salta se (x != 3), "bypassando" o incremento!
  inc dword [x]
.L1:
```

Não há muita coisa que possamos fazer a respeito desse tipo de código a não ser tentar garantir que *x* seja igual a 3 na maioria das vezes que ele for executado (o que nem sempre é possível). O real problema encontra-se em construções do tipo *if...then...else*. Neste caso, pode aparecer um salto incondicional após o bloco *then*:

```
if (x == 3)
  x++;
else
  x--;
----- corte aqui -----
; Possível código equivalente
  cmp dword [x],3
  jne .L1
  inc dword [x]
  jmp .L2
.L1:
  dec dword [x]
.L2:
```

O código, em C, acima, pode ser escrito de duas formas totalmente equivalentes:

```
/* Forma #1 */
…
if (x == 3)
  x++;
else
  x--;
…
```

```
/* Forma #2 */
…
if (x != 3)
  x--;
eles
  x++;
…
```

Ambas as formas dão o mesmo resultado prático, mas também podem ter performances diferentes, dependendo do algorítmo... Se *x* for diferente de 3 na maioria das vezes, a primeira forma tem o potencial de acrescentar 1 ciclo de máquina quando este *if* estiver dentro de um loop (porque o salto condicional "para frente" será tomado) e, neste caso, a segunta forma tem o potencial de ser mais eficiente. Mas, se *x* for igual a 3 na maioria das vezes a situação inverte...

> A dica é: Se você tiver um bloco *if..then..else*, escolha sua condição de forma que ela seja **verdadeira** na maioria das vezes. Assim, o bloco *else* será executado com menos frequência, fazendo com que menos saltos condicionais "para frente" sejam tomados, aumentando a performance!
>
> **OBS**: Isso vale para **qualquer** linguagem que compile o código original, mesmo Java ou C#, que geram código intermediário... Mas, pode não ser válido para *script languages* como PHP, Perl, Python, Javascript etc.

Dando dicas ao processador sobre saltos condicionais

É possível dizer ao compilador se um salto condicional vai ser executado mais vezes ou não. Existem dois prefixos, que valem apenas para saltos condicionais *Jcc*. Infelizmente, não há mnemônicos para esses prefixos[30]. Se quisermos usá-los, devemos colocá-los no cogido explicitamente.

Os prefixos são 0x2E (salto não será feito) e 0x3E (salto será feito). Esses prefixos são **dicas** para o processador, para que o algorítmo interno de *branch prediction* sofra um "tunning"... Eis um exemplo:

```
cmp eax,[rbp-8]
db  0x2E          ; Dica: O salto condicional à seguir não será feito, na maioria das
                  ;       vezes que passar aqui!
jz  .L1
…
```

Por que, então, não usamos essas dicas o tempo todo? Porque se dissermos ao processador que um salto provavelmente não será feito e ele o for, haverá uma penalidade. É preferível deixar o processador decidir isso e construirmos nossas rotinas de forma mais cuidadosa.

O GCC implementa algo semelhante em uma de suas funções *built in.* A função *__builtin_expect*, no entanto, **não** coloca um dos prefixos citados acima numa comparação. O que ela faz é rearranjar o salto condicional para que satisfaça sua dica... Num loop *while*, por exemplo, a condição normalmente é testada no final e, se satisfeita, o código "pula" para o corpo do loop. Neste caso o compilador assume que a confição vai ser satisfeita mais vezes do que rejeitada. Ao colocar um *__builtin_expect(cond, 0)* no *while*, você diz ao compilador que a confição será **falsa** mais vezes do que verdadeira e ele rearranjará o teste da comparação de forma tal que a rotina continue tão rápida

30 Se você for curioso o bastante notará que os prefixos para os seletores CS e DS são os mesmos... 0x2E para CS e 0x3E para DS. Quando você usa uma referência do tipo [ds:rip], numa instrução, ela é prefixada com 0x3E. Mas prefixar um label com CS ou DS, em assembly, não é possível com instruções de salto condicional!

quanto possível. Outro exemplo é o uso de *if..then..else*:

```
if (cond)
  do_something();
eles
  do_something_else();
```

O compilador provavelmente fará algo assim:

```
  cmp dword [cond],0
  jne  .L1
  call do_something
  jmp .L2
.L1:
  call do_something_else
.L2:
```

Se o compilador tenta sempre fazer com que saltos condicionais “para frente” não sejam tomados, isso significa que o teste de *cond* é assumido como verdadeiro na maioria das vezes (e JNE não consumirá um ciclo extra por causa do *branch prediction*). Lembre-se que JMP não sofre as penalidades das previsões de salto... Acontece que podemos querer que o compilador assuma que a condição é **falsa** na maioria das vezes. Poderíamos fazer isso de duas formas:

1. Mudando a ordem do bloco *then* e *eles* e o teste de *cond* para *!cond*;
2. Usando *__builtin_expect(cond,0)* na condição do *if*.

Quantos mais dados você manipular, pior a performance

Essa dica parece óbvia. Afinal, se você percorrer um array com 10 itens vai gastar tempo num fator 10 vezes maior que se você lidasse com apenas uma variável... Se percorrer 100 mil itens, o tempo gasto será 1000 vezes maior do que se lidasse com apenas 10 e assim por diante.

Mas, não é sobre isso que estou falando... O problema aqui é que esses arrays precisam estar armazenados na memória. Eles não podem ser contidos num registrador, por exemplo, e acesso à memória é lento!

Os processadores modernos tentam mitigar esse *delay* de acesso à memória mantendo parte dela numa memória mais rápida no interior do processador, chamada de *cache*. Só que os caches (existem mais que um!) tem capacidade limitada e a mecânica de acesso a esses dados internos não é somente através de um endereço. Isto é, a forma com que os caches são organizados é mais complicada.

Como regra geral, estamos mais interessados no cache de primeiro nível (o cache L1), que está atrelado mais intimamente com as unidades de execução do processador. Mas, o que se aplica ao cache L1, aplica-se ao cache L2 e L3 também, mas de uma maneira um pouco diferente...

O cache L1 geralmente é organizado em blocos de 64 bytes. Não é possível lidar com blocos menores do que esse tamnho, no que concerne o cache... Isso significa que sempre que uma porção da memória RAM é transferida para o cache, essa porção terá **sempre** o tamanho de 64 bytes. Outro fato digno de nota é que o cache L1 é composto de, no máximo, 256 linhas. Isso nos dá, no máximo 16 KiB de RAM que podem estar copiadas no cache.

Sabendo disso, não é de surpreender que um código como mostrado à seguir tenha baixa performance:

```
int a[8192], i, sum=0;

for (i = 0; i < 8192; i++)
  sum += a[i];
```

Suponha que o cache L1 não contenha nada do array 'a'. Na primeira tentativa de ler o array 64 bytes dele será copiado para o cache L1... A partir do item 16, o segundo bloco será copiado para o cache (porque um *int* tem 4 bytes de tamanho) e assim por diante.

A cada vez que esses 64 bytes tenham que ser copiados para o cache, o processador fará uma parada rápida (8 ciclos de máquina) para ler a linha. Na rotina acima o array contém 512 blocos de 64 bytes. Isso quer dizer que 4096 ciclos de máquina (512 blocos vezes 8 ciclos) serão gastos apenas lendo linhas para o cache!

> A dica é clara: Mantenha seus arrays o mais curtos possível, de preferência dentro da fronteira dos 64 bytes!

Códigos grandes também geram problemas com o cache!

A regra dos 64 bytes acima também vale para o código em assembly e é especialmente problemático se as μ_{op}s cruzarem a fronteira de uma linha do cache.

A boa notícia é que você não tem que se preocupar com a competição entre seu código e os dados que ele manipula, se for avaliar a pressão no cache L1. Existe um cache L1 para dados e outro cache L1 para código (são chamados cache L1d e L1i, de dados e instruções, respectivamente).

Então, para zerar o conteúdo de um registrador, é sempre uma boa idéia **não** usar um MOV, mas usar um XOR:

```
0:     b8 00 00 00 00      mov eax,0
5:     31 c0               xor eax,eax
```

A instrução *xor* ocupa apenas 2 bytes no cache L1i, enquanto o *mov* ocupará 5 bytes. Em certos casos, a coisa pode piorar se usarmos o modo x86-64. Lembre-se que, ao lidar com registradores como *rax* estaremos lidando com valores que ocupam 8 bytes (64 bits)[31].

O uso da instrução XOR para esse fim nem sempre é possível, já que ela afeta o flag ZF, quanto *mov* não afeta flag algum... Mas esse é um "macete" que vale a pena lembrar, para não gastar ciclos extras lendo linhas para o cache L1i.

Felizmente, compiladores como o GCC tomam conta disso para nós... Mas, nem sempre é bom confiar no compilador, certo?

Se puder, evite usar os registradores estendidos, no modo x86-64

Por "registradores estendidos" quero dizer aqueles que começam com 'r', como *rax*, *r8* etc. O problema é justamente o tamanho das instruções. Quando usamos um desses registradores um prefixo é adicionado à instrução (prefixo "REX", código 0x48, por exemplo), indicando que trata-se de uma instrução que lida com 64 bits. Não somente o prefixo é adicionado, mas outros campos podem ser aumentados também. Na prática, uma instrução pode ter até 15 bytes de tamanho. Só que se você ficar usando instruções grandes, uma linha do cache L1i vai ser preenchida com menos instruções e o atraso pela carga/descarga/decodificação do cache vai influenciar muito a performance do seu código.

Para minimizar isso, ao inicializar um registrador como EAX, por exemplo, a porção superior de RAX é **automaticamente** zerada.

---

31 Um outro lembrete: Ambas as instruções, do jeito que estão, também colocarão 0 (zero) em RAX, se estivermos lidanco om o modo x86-64. Ao mexermos com EAX, estamos **automaticamente** zerando os 32 bits superiores de *rax*.

Além disso, a não ser por questões de endereçamento, se você estiver lidando com tipos como *int*, é sempre bom usar registradores como *eax* ao invés de seu irmão maior. As instruções ficarão menores.

Os acessos à memória também podem ser reordenados

Além da reordenação automática das instruções o processador tenta fazer o mesmo com as operações de leitura e escrita na memória. Ele faz isso para tentar ganhar tempo no cálculo do endereço efetivo e para tentar manter os dados o menor tempo possível (ou maior, depende como você encara) nos caches... Isso pode gerar alguns problemas de performance, uma vez que o cache L1d tem tamanho limitado.

Às vezes precisamos dar uma forcinha ao processador, dizendo que ele deve "esperar" que todos os dados tenham sido lidos ou gravados na memória (ou no cache L1d). Para tanto, existem as instruções LFENCE, SFENCE e MFENCE. A primeira, LFENCE, espera que todas as leituras (Loads) pendentes sejam feitas. Por analogia, SFENCE espera por todas as escritas (Stores) pendentes. MFENCE espera por ambas leituras e escritas.

Outra forcinha que podemos dar ao processador é relativo ao cache L1d... Podemos querer que o processador tenha certeza que uma região da memória esteja presente no cache para não perdermos tempo esperando que essa região seja copiada. Este é o caso do uso das instruções PREFETCHT0, PREFETCHT1, PREFETCHT2, PREFETCHNTA, PREFETCHW e PREFETCHWT1. O termo "prefetch" significa pré-busca ou pré-carga. T0, T1, T2 e NTA são dicas sobre a "temporalidade" do dado. "Temporalidade" aqui só quer dizer "por quanto tempo" o dado vai ser usado pelo processador... Se a ideia é usar o dado apenas para leitura sem considerar o tempo em que ele vai ser usado, usar PREFETCHNTA pode ser uma boa pedida (NTA, de *Non Temporal All caches*). As dicas de T0 até T2 referem-se ao nível de cache (T0 sendo L1 e T2 sendo L3).

Já PREFETCHW é dedicado a carga, no cache L1, para escrita. Sua variante, PREFETCHWT1 faz o mesmo, mas prefere a carga no cache L2.

Essas instruções devem ser usadas apenas em casos muito especiais, para tentar controlar o comportamento das leituras e escritas de dados nos caches. MFENCE tem *throughput* de 35 ciclos e emparelhá-la com instruções que acessam memória só atrapalha. Ela tende a ser executada sozinha... LFENCE e SFENCE são mais rápidas (5 e 1 ciclos, respectivamente), mas sofrem do mesmo mal. As instruções PREFETCH são mais rápidas, geralmente gastam de 1 a 3 ciclos, dependendo se a linha já esteja nos caches ou não...

Na maioria das vezes você vai ter que confiar no processador para não causar uma queda considerável de performance... Lidar com *fences* faz com que o processador **serialize** a execução de instruções e o acesso à memória, jogando no lixo a melhoria de performance causada pelas reordenações.

Medindo a performance

Com todas as regras de execução de instruções (e apenas mostrei algumas!) fica difícil contar ciclos "manualmente". O melhor que podemos fazer é medir quantos ciclos nossas rotinas gastam. Para tanto, precisamos saber quantos ciclos de clock se passaram a partir do momento do início da medição até o final.

Felizmente, desde o *Pentium*, a intel mantém um contador de ciclos interno ao processador, acessível via uma instrução que pode ser usada no *userspace*: RDTSC. Essa instrução lê (ReaD) um registrador chamado TSC (*Time-Stamp Counter*), que mantém a contagem de clocks desde o momento que o processador foi resetado pela última vez. Este é um valor de 64 bits de tamanho,

assim, mesmo que seu processador funcione numa frequência de 3.4 GHz, levaria cerca de 172 anos (sim! Mais que um século e meio!) para que o contador sofresse um overflow.

A instrução RDTSC ajusta o par de registradores *edx:eax*, mesmo na arquitetura x86-64, para conter o valor de 64 bits correspondente ao número de clocks. E essa é uma daquelas instruções que quase não tem latência, mas tem um "tempo de recuperação" alto, que é desconsiderável pelo uso que faremos dela. No entanto, essa instrução pode devolver valores de clock não condizentes com a execução que queremos graças ao recurso de reordenação de instruções que o processador faz. Daí, é aconselhável esperar que as instruções sejam todas executadas **antes** de chamar *RDTSC*.

Para conseguir esse efeito, alguns processadores possuem a instrução RDTSCP. Essa instrução lê, também, um outro registrador interno chamado IA32_TSC_AUX, que tem o efeito de "esperar" que a fila de execução seja esvaziada. Infelizmente nem todos os processadores têm essa instrução disponível, então temos que ficar RDTSC mesmo e usar outra maneira de esperar pela limpeza da fila de execução. Isso é feito com a instrução CPUID.

As rotinas para leitura do clock têm que ser divididas em duas. Uma para iniciar a medida e outra para finalizá-la. Chamarei ambas aqui de *begin_tsc()* e *end_tsc()*, para facilitar:

```
static unsigned long long _tsc;

inline void begin_tsc(void)
{
  unsigned int a, d;

  asm volatile ("xorl %%eax,%%eax\n"
                "cpuid\n"
                "rdtsc"
                : "=a" (a), "=d" (d)
                : : "%ebx" );
  _tsc = ((unsigned long long)d << 32) | a;
}

inline unsigned long long end_tsc(void)
{
  unsigned int a, d;

  asm volatile ("rdtscp"
                : "=a" (a), "=d" (d));
  return (((unsigned long long)d << 32) | a) - _tsc;
}
```

Em teoria essas funções são capazes de medir até mesmo a quantidade de ciclos gastos por uma única instrução em assembly. Na prática, no entanto, isso não é possível... Ao medir os ciclos gastos no código condigo entras as chamadas das funções, muita coisa pode acontecer. Todos os fatores, desde *branch prediction*, *page faults*, interrupções, *cache misses* etc, influem no valor lido. Tudo o que podemos conseguir é uma medida mais ou menos precisa, mas não **exatamente** precisa.

Como exemplo de medição, suponha que queiramos saber quantos ciclos gasta uma função *dosomething()*, em C:

```
…
unsigned long long cycles;

begin_tsc();
dosomething();
cycles = end_tsc();

printf("Ciclos gastos por dosomething(): %llu", cycles);
```

Nada pode ser mais simples que isso, pode? No entanto, sempre existem problemas.

Medições são, por natureza, imprecisas. Pense no caso da medição de uma distância usando uma trena ou uma régua: Provavelmente sua precisão máxima será na ordem de milímetros, deixando

qualquer intervalo intermadiário fora da sua faixa de precisão... E, mesmo assim, réguas diferentes podem apresentar precisões diferentes. A espessura da tinta usada para fazer as marquinhas de milímetros pode variar, por exemplo.

A mesma coisa acontece com qualquer tipo de medição. Sempre existe uma margem de erro aceitável... No caso da medição de ciclos, você não obterá sempre o mesmo valor quando medir a velocidade da mesma rotina várias vezes. Como já falei antes, existem fatores "aleatórios" que devem ser levados em consideração (cache, *page faults* etc). O que as duas rotinas acima te darão é uma aproximação da quantidade de ciclos gastos pela rotina medida.

Uma maneira de diminuir esse erro é fazendo diversas medidas e calculando uma média aritmética simples. Os erros acumulados serão diluídos nas **n** medidas feitas e você conseguirá um valor aproximado mais condizente com a realidade. Mas, mesmo nesse caso é preciso tomar alguns cuidados. Se você fizer algo assim:

```
#define NUM_SAMPLES 100

begin_tsc();
for (i = 0; i < NUM_SAMPLES; i++)
  dosomething();
cycles = end_tsc() / NUM_SAMPLES;
```

Você terá que levar em conta como o compilador tratará o loop. É possível que o próprio loop tenha uma grande influência na medição:

```
  push rbx
  xor  ebx,ebx

  ; begin_tsc();
  xor eax,eax
  cpuid
  rdtsc
  mov eax,eax
  shl rdx,32
  or  rax,rdx
  mov [_tsc],rax

  ; Todo esse código será medido, não apenas as chamdas!
  jmp  .L1
.L2:
  inc  ebx
  call dosomething
.L1:
  cmp  ebx,NUM_SAMPLES
  jb   .L2

  ; end_tsc()
  rdtscp
  mov  eax,eax
  shl  rdx,32
  or   rax,rdx
  sub  rax,[_tsc]
  xor  edx,edx
  div  rbx
  pop  rbx
  ; rax conterá a média aqui.
```

Repare que não estamos medindo somente a chamada a *dosomething*, mas também a estrutura de controle do loop! O jeito correto de fazer essa medição através da média seria algo assim:

```
begin_tsc();
dosomething();
dosomething();
… chamar dosomething() mais 96 vezes …
dosomething();
dosomething();
cycles = end_tsc() / 100;
```

Calculando o "ganho" ou "perda" de performance

Sempre que você quer calcular o quanto ganhou ou perdeu tem que fazê-lo em relação a um valor original. Aqui estamos falando de ganho ou perda de ciclos. A simples divisão de um pelo outro:

$$r = \frac{ciclos_{nova}}{ciclos_{velha}}$$

Te dá uma noção da relação entre ciclos gastos pela rotina nova ($ciclos_{nova}$) sobre os ciclos gastos pela rotina velha ($ciclos_{velha}$). Se o valor de *r* for menor que 1, tanto melhor (ganho de "velocidade" no processamento)...

Embora essa relação seja clara como cristal, o valor que ela expressa não é tão intuitivo assim... Considere uma rotina original que gaste 120 ciclos e uma nova rotina que construímos em assembly que gaste 30 ciclos. A relação *r* será 0,25. Isso significa que a rotina nova gasta 25% dos ciclos gastos pela rotina velha, mas não expressa o quanto ganhamos de performance.

O "ganho", é claro, pode ser expresso como o recíproco de *r* (ou seja, $g = \frac{1}{r}$). No caso acima, temos um ganho de 4 vezes (a rotina nova é "4 vezes" mais rápida que a rotina velha). Da mesma forma, se a rotina nova for mais lenta que a velha (suponha que $ciclos_{nova}$=120 e $ciclos_{velha}$=30), o "ganho" será inferior a 1, indicando uma perda (se você tem 0,25 do que tinha antes então você perdeu um bocado!). Mas, note que neste caso a relação acima te dará justamente o valor 4.

Ambas as interpretações são válidas... Eu prefiro medir a performance em termos de ganhos e perdas.

O ganho ou perda de performance pode ser calculado em relação ao que se quer medir:

$$G[\%] = \frac{ciclos_{velha} - ciclos_{nova}}{ciclos_{nova}} \cdot 100$$

Estamos medindo o ganho da nova rotina em relação a diferença entre a velha e a nova...

Nos exemplos anteriores, quando $ciclos_{velha}$=120 e $ciclos_{nova}$=30, teremos um ganho de performance de 300%. Se fizermos o contrário, fizermos $ciclos_{velha}$ ser 120 e $ciclos_{nova,}$ 30, o valor de *G* será de -75%. A nova rotina é 75% mais lenta que a original.

Apêndice A – Se você é uma Maria "Gasolina"...

Para quem prefere o GNU Assembler (GAS), o sabor da linguagem assembly é diferente do da Intel, bem como as instruções e diretivas usadas pelo compilador. É claro que, no fim das contas, acabaremos com o mesmo código em **linguagem de máquina**, mas o assembly não é o mesmo.

O GAS faz as coisas de um modo ligeiramente diferente dos outros assemblers. A única grande vantagem, em minha opinião, é que ele é multiplataforma. Você pode escrever código para i386, x86-64, ARM, MIPS, MC68000, Intel 80960, SPARC, Z8000, VAX, dentre outros processadores. Isso não é, normalemnte, possível quando usamos NASM, MASM, FASM, YASM e um monte de outros assemblers disponíveis por ai.

Os tópicos abaixo são restritos aos modos i386 e x86-64... E tenho uma outra boa notícia: Todos os recuros mostrados abaixo, incluindo a declaração de macros, estão disponíveis no *assembly inline* do GCC!

Usando o GAS

Compilar seus códigos em assembly com o GAS é simples:

```
$ as hello.S -o hello.o
```

Históricamente, os arquivos com seus códigos em assembly são nomeados com a extensão ".S" (maiúsculo) para diferenciar dos arquivos gerados pelo GCC, que têm extensão ".s" (minúsculo). Como qualquer compilador, o GAS gera um arquivo objeto no formato ELF, de acordo com a plataforma em que você compila. Não há necessidade, como no NASM, de especificar ELF32 ou ELF64. O formato correto para a sua arquitetura vai ser gerado.

É possível usar opções de *tunning* como **-march** ou **-mtune**, mas não existe uma opção "native", como no GCC.

Se quiser gerar uma listagem com os μ_{op}s a partir do código em assembly, pode usar a opção **-ahlns**, opcionalmente especificando o nome do arquivo adicionando um "=filename.lst" como em:

```
$ as hello.S -ahlns=hello.lst-o hello.o
```

Comentários são iniciados com #

Ao contrário de outros assemblers, o caracter ";" não é usado para marcar o início de um comentário. Usamos o caracter "#" para isso. O ponto-e-virgula marca o final de uma linha e é opcional.

Instruções contém o tamanho dos operandos

Ao invés de usar uma simples instrução MOV, no sabor AT&T temos que atrelar o tamanho dos operandos à instrução, escrevendo MOVB, MOVW, MOVL ou MOVQ, para os tipos *byte*, *word*, *dword* (long) ou *qword*, respectivamente. Isso é válido para a maioria das instruções, exceto aquelas onde a especificação do tipo é irrelevante como MOVSS ou STOSB.

Algumas instruções possuem mnemônicos estranhos, como MOVSLQ que é a mesma coisa que equivalente a MOVSX, na notação Intel. O "L" e o "Q" informam ao compilador que um operando

long será movido para um *quad*... Assim, temos MOVSBW, MOVSWL e MOVSLQ, todas essas são a MOVSX.

Prefixos REP, REPZ e REPNZ são "instruções" separadas

Se precisar usar o prefixo REP, REPZ ou REPNZ, é necessário usá-los separadamente:

```
repz
stosb
```

Ou, usando o ";" poderíamos escrever "repz; stosb".

A ordem dos operandos é contrária a usada no sabor Intel

Quando escrevemos:

```
movl $0,%eax
```

Estamos movendo o valor imediato, decimal, 0 (zero) para dentro de EAX. Isso pode confundir um pouco quando se usa instruções com dois operandos, onde ambos são registradores:

```
shll %cl,%edx       # EDX = EDX << CL.
outb %al,%dx        # Escreve AL na porta DX.
addw %rdx,%rax      # RAX = RAX + RDX.
```

Pode ser ainda mais estranho com instruções com 3 ou mais operandos:

```
shufps $0x1b,%xmm1,%xmm0    ; Intel: SHUFPS XMM0,XMM1,0x1b
```

Uma vantagem do sabor AT&T

É necessário especificar os nomes de registradores precedidos do caracter "%". Isso é vantajoso, já que poderemos criar símbolos que tenham o mesmo nome de registradores e, ao usá-los sem o "%", não haverá ambiguidades. O sabor Intel não permite isso.

Valores imediatos

Todo valor imediato (numérico) deve ser precedido de um "$", como em:

```
movw $10,%ax      # AX = 10.
```

Isso vale também para obtenção do endereço de um símbolo:

```
movq $var1,%rsi     # RSI recebe o endereço de "var1"
movq var1,%rsi      # Mesma coisa que: movq var1(,,1),%rsi
```

O uso obrigatório de $ só não é válido quando usamos um modo de endereçamento explícito (com parentesis):

```
movq $var1(%edi),%eax     # Isso causa erro de compilação!
movq var1(%edi),%eax      # Este é o certo!
```

Obtendo o endereço atual

Em outros compiladores o símbolo $ é usado para obtermos o endereço atual. Podíamos fazer, no NASM, algo assim:

```
msg     db   "hello!\n"
msg_len equ $ - msg
```

A expressão “$ - msg” significa “o endereço atual menos o endereço do símbolo msg”. No GAS o $ é usado para indicar se estamos lidando com um valor imediato ou com um símbolo (ou seu conteudo). Assim, o caracter usado para especificar o “endereço atual” é um ponto (“.”).

A mesma sequência pode ser escrita, no GAS, como:

```
msg: .ascii  "hello!\n"
     .equ msg_len, . - msg
```

Os modos de endereçamento são especificados de forma diferente

Ao invés de usarmos colchetes como operador de indireção, usa-se parentesis, onde o offset fica de for a e podemos ter 3 termos:

```
movb v(%rsi,%rbx,2),%al   ; Intel: mov al,[rsi + 2*rbx + v]
movw %cx, 4(%rax,%rbx)    ; Intel: mov [rax+rbx+4],cx
```

A notação de parentesis segue o padrão “deslocamento(base, índice, escala)”. Onde “deslocamento” é sempre um valor inteiro (ou uma referência ao símbolo).

Podemos omitir quaisquer dos 4 valores (mas não todos!) nesse modo de endereçamento. Por exemplo, “4(,%eax,2)” é equivalente a “[eax*2+4]”, na notação da Intel.

Se precisarmos prefixar o modo de endereçamento com um seletor, tomemos FS como exemplo, basta prefixar o modo de endereçamento com “*%fs:*”. O endereço será obtido da expressão que segue o prefixo, mas a instrução usará o seletor indicado. É claro, o seletor *default* é usado de acordo com a base no modo de endereçamento. Se usarmos as variações de *%rsp* ou *%rbp* como base, o seletor SS é automaticamente usado, caso contrário, o DS... Essa é uma regra do processador, não do GAS.

Saltos e chamadas de funções

Assim como no NASM, a maioria das chamadas e saltos são codificadas por endereços relativos ao registrador RIP. Ao fazer:

```
  jmp  .L1
…
.L1:
  ret
```

O código gerado para a instrução JMP usará um endereçamento relativo. No caso do NASM, para informar ao compilador para usar um endereço absoluto deve-se adicionar o modificador ABS:

```
jmp  abs .L1
```

No caso do GAS o endereçamento **absoluto**, é forçado com um atalho: basta usar um asterisco antes do endereço:

```
  jmp *1f       ; usa endereçamento absoluto
  jmp 1f        ; usa endereçamento relativo ao RIP.
1:
  ret
-----%<----- corte aqui -----$<-----
0: FF 24 25 00 00 00 00  jmp *1f
7: FB 00                 jmp 1f
9: C3                    ret
```

No primeiro JMP o linker substituirá os quatro bytes (em vermelho) com o endereço absoluto do

salto. A mesma coisa funciona com CALL.

Além do salto “próximo”, o sabor AT&T fornece também um salto “para longe”, onde o seletor do segmento deve ser fornecido. As instruções LJMP e LCALL fazem isso.

Resta-me só explicar essa notação estranha do uso do label “1”. No NASM podemos criar labels “locais” de funções usando um “.” na frente do nome. Assim, se tivermos:

```
global f:function
f:
  jmp .L1
.L1:
  ret
```

O label “.L1” pertence à função *f*.

O GAS permite o uso de um tipo diferente de label “local”: Se numerarmos as posições (1, 2, 3, …), podemos referenciá-las incluindo a informação dizendo se o label está atrás ou à frente da posição atual, usando “f” para *forward* e “b” para *backward*.

Isso permite termos diversos labels com nomes iguais que não são conflitantes. Por exemplo:

```
.globl f
.type f,@function
f:
1:
  jmp 1f    ; salta para o 1 da linha abaixo.--
1:          ;        <-------------------------
  nop
2:          ;        <------------------------
  jmp 2b    ; salta para o 2 da linha acima.--
2:
```

Algumas diretivas do compilador

Da mesma forma que o NASM, o GAS tem algumas diretivas facilitadoras. Podemos especificar as *sections* default *.text* e *.data* de forma resumida assim:

```
# Código equivalente de:
#
# long f(int i) { return x[i]; }
#
.text         # segmento de código.

.globl f
.type f,@function
f:
  movl x(%edi),%eax
  ret

.data         # segmento de dados.

.type x,@object
.size x,12
x:
  .long 1, 2, 3
```

Para os outros segmentos podemos usar a diretiva *.section*, que pode possuir atributos:

```
.section .rodata    # usa a section .rodata definida pelo linker.
…
.section .myconsts,"rd"    # usa uma section chamada “.myconsts” que é
                           # read-only (r) e só possui dados (d).
```

É claro que nomes de *sections* precisam estar definidos no script do linker para podermos usar “.myconsts”...

Códigos em 16, 32 e 64 bits

Existem duas diretivas que diferenciam códigos escritos para 16 e 32 bits: *.code16* e *.code32*, no entanto, elas tendem a não serem usadas. Isso existe porque, em 16 bits, se usarmos um registrador de 32 bits, a instrução tem que ser prefixada com 0x66.

Já entre 32 e 64 bits não existem diretivas que os diferencie. O motivo está no uso do prefixo REX, que é colocado antes de uma instrução se ela usar um registrador de 64 bits, caso contrário, a variação de 32 bits continua valento.

No modo de 16 bits, nos códigos gerados pelo GCC 4.9, a diretiva *.code16gcc é* incorporada na listagem, indicando que o código será escrito para 16 bits.

Labels têm "tipos" e "tamanhos"

Existem, basicamente, dois tipos: *@function* e *@object*. O tipo @object pode também ser especificado como *@data*. Da mesma forma que no NASM, isso pode ser ignorado, mas é útil.

Já a diretiva *.size* especifica o tamanho do "objeto", seja ele uma variável ou uma função. Repare que atribui o valor 12 ao tamanho do símbolo *x*, no código de exemplo acima. Isso é opcional, mas pode ajudar ao linker no alinhamento de dados. A diretiva *.size* tem um significado similar paara funções, só que devemos especificar o início e o fim, assim:

```
f:
  …
  ret
.size f, .-f
```

Também é uma diretiva opcional, mas útil.

Definindo dados

Ao invés de usar diretivas como DB, DW, DD e DQ, usamos as diretivas *.byte*, *.word*, *.long* e *.quad*, respectivamente, seguidas dos dados.

Existe um atalho para declaração de strings através das diretivas *.ascii* e *.asciz*. Não é possível escrever strings com a diretiva *.byte*, por exemplo... A diferença entre *.ascii* e *.asciz* é que a última coloca um '\0' no final, assim como é feito na linguagem C.

Ainda no tópico de definição de dados, a diretiva EQU, correspondente à diretiva #define em C, também é usada como *.equ*, seguida do símbolo e da sua definição:

```
.equ FILENO_STDIO,1   # FILENO_STDIO=1
```

Alinhamento de dados e código

Funciona quase da mesma maneira que nos outros compiladores. As diretivas *.align* e *.palign* fazem a mesma coisa. Essas diretivas aceitam 3 parâmetros. O primeiro é o número de bits inferiores **zerados** do endereço final. Se usamos ".align 3", isso significa um alinhamento por *quadword* (os 3 bits inferiores do endereço tem que ser zerados).

Alinhamento significa que o GAS ira incrementar o contador de endereços até um que satisfaça a regra imposta na diretiva... O segundo parâmetro nos dá o byte que será usado para preenchimento (pode ser vazio, deixnado o compilador decidir) e o terceiro parâmetro é a quantidade máxima de deslocamentos do contador (que também pode ser vazio).

Assim, "*.align 3,,5*" significa que o GAS ira preencher a posição atual com alguma coisa até chegar

num endereço com os 3 bits inferiores zerados. Mas o alinhamento só será feito se a quantidade de preenchimentos não exceder 5.

Exportando e importando símbolos

A diretiva *.globl* exporta um símbolo para o linker e a diretiva *.extern* informa que o símbolo é externo ao módulo compilado. Mas, *.extern* só existe por questão de compatibilidade. **Todo** símbolo não definido no módulo é encarado como externo e cabe ao linker resolvê-lo.

Algumas diretivas especiais

É comum, nas listagens obtidas pelo GCC, observarmos diretivas com prefixo "*.cfi_*". Essas diretivas são extensões do GAS chamadas *Call Frame Information* e está relacionada a como o compilador lidará com a pilha e outras características do código fonte.

As diretivas *.cfi_startproc* e *.cfi_endproc* podem ser vistas, nas listagens geradas pelo GCC, delineando uma funçao. Em alguns casos essas diretivas adicionarão código na sua rotina, por exemplo, estabelecendo o *stack frame.* A rotina abaixo:

```
.globl f
.type f,@function
f:
  .cfi_startproc
  leal (%edi,2),%eax
  ret
  .cfi_endproc
```

Pode (embora provavelmente não vá fazê-lo!) gerar algo assim, no modo i386:

```
; Listagem no sabor Intel.
push ebp
mov  ebp,esp
lea  eax,[edi*2]
leave
ret
```

Essas diretivas **não** são necessárias para que seu código compile com sucesso, embora algumas possam ajudar a mudar alguns comportamentos do GAS.

Macros

O GAS também possui o recurso de macros. Funcionam um pouquinho diferente do NASM. Aqui, todos os parâmetros tem que ser especificados na diretiva e referenciados, de alguma forma, no corpo do macro:

```
.macro MYMACRO name, x, y, z
\name:
  .long \x, \y, \z
.endm
```

Usar o macro fica muito fácil:

```
MYMACRO myvar,1,2,3
```

Há diretivas condicionais (.if, .ifdef, .else, .endif), que funcionam da mesma forma que #if, #ifdef, #else e #endif, em C. Existe uma diretitva de atribuitção de valores para símbolos (.set) e também uma diretiva de saída de macros (.exitm).

Podemos fazer algo assim, por exemplo:

```
.macro MYMACRO name, x, y=-1
\name:
  .long \x
  .if (\y == -1)
    .exitm
  .endif
  .long \y
.endm

.data

MYMACRO x, -1, 2
MYMACRO y, 3
```

Isso ai vai criar os labels *x* e *y* definidos como:

```
x:  .long -1
    .long 2
y:  .long 3
```