“Brace yourself for what you are about to see!” (Patterson, Henessy; 2014)

Introdução ao x86-64

Paulo Ricardo Lisboa de Almeida

UDESC
UNIVERSIDADE
DO ESTADO DE
SANTA CATARINA

x86-64

- O x86 foi introduzido pela Intel em **1978** em seu processador **8086**
  - Desde então, são mais de 40 anos de remendos e enjambres no conjunto de instruções original!
- Inicialmente lidava com palavras de 8 bits
  - Estendido para 16 bits, e depois para 32
- Em **2003**, estendido para 64 bits pela AMD
  - Conjunto AMD64
    - Também conhecido como x64 ou x86-64
  - Vamos chamar de **x86-64** pela clareza
    - **Extensão do conjunto original, e não um conjunto novo**
- **Amarras de ouro**
  - Toda alteração feita no x86 precisa ser **retrocompatível** com as versões anteriores

Patterson e Hennessy sobre o x86

"…a história ilustra o impacto das 'amarras douradas' da compatibilidade com o x86, pois a base de software existente … era muito importante para ser colocada em risco com mudanças arquitetônicas significativas … Esse ancestral diversificado [x86] levou a uma arquitetura **difícil de explicar e impossível de amar**. Prepare-se [*Brace yourself*] para o que você está prestes a ver …" (Patterson, Hennessy; 2014)

x86-64

- Vamos explorar o básico sobre o assembly do x86-64
  - O suficiente para
    - Depurar programas a nível de linguagem de montagem
    - Entender melhor sobre como nossas máquinas funcionam
- **Não focaremos em desempenho**
  - O x86-64 possui muitos detalhes **complexos** e **contraintuitivos**
  - Não cabe no pouco tempo que temos

Registradores e memória

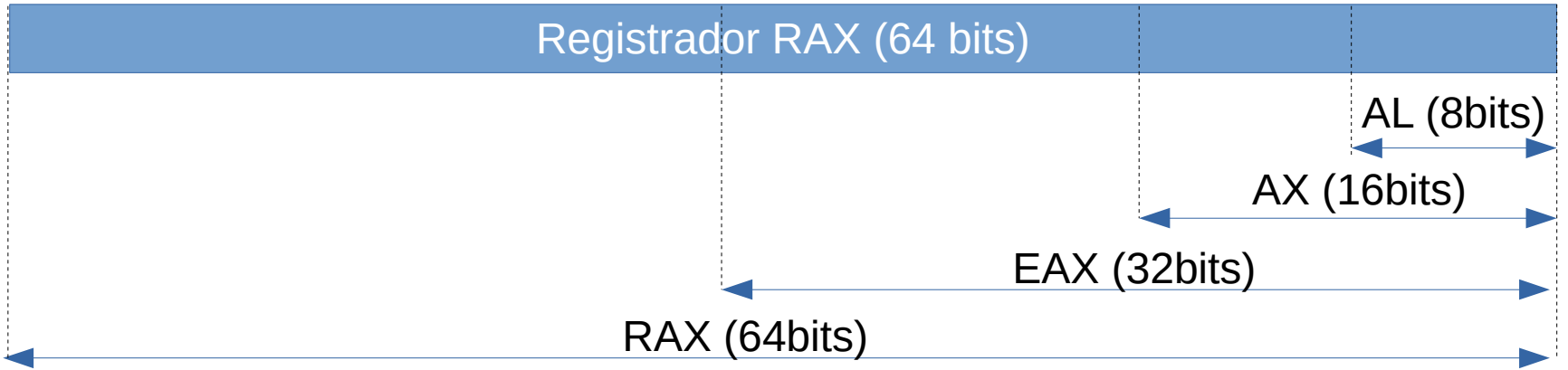
- O x86-64 é **little-endian**
  - Os dados são armazenados em ordem invertida na memória
  - O byte menos significativo está no endereço de memória mais baixo
- A maioria dos registradores comporta 64 bits (no modo 64 bits)

Registradores e memória

- No modo 64 bits, podemos acessar os registradores da seguintes formas
  - Quadword → os 64 bits do registrador
  - Doubleword → os 32 bits mais baixos
  - Word → os 16 bits mais baixos
  - Byte → os 8 bits mais baixos
- Obs.: existe ainda o double quadword (128 bits) utilizado por instruções SSE

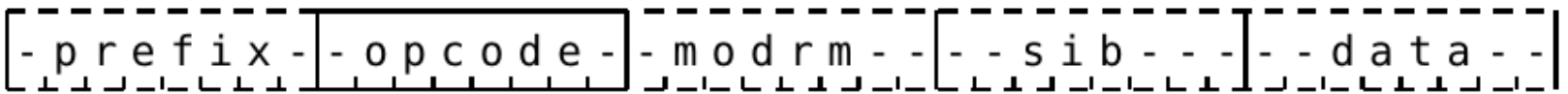
Registradores

- Temos 16 registradores de uso geral
  - RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP e R8-R15.
    - No x86 de 32 bits tínhamos apenas 8 registradores
- Prefixos nos registradores indicam se estamos acessando somente seu Byte, Word, DoubleWord, …
  - Exemplo:



Formato de instruções

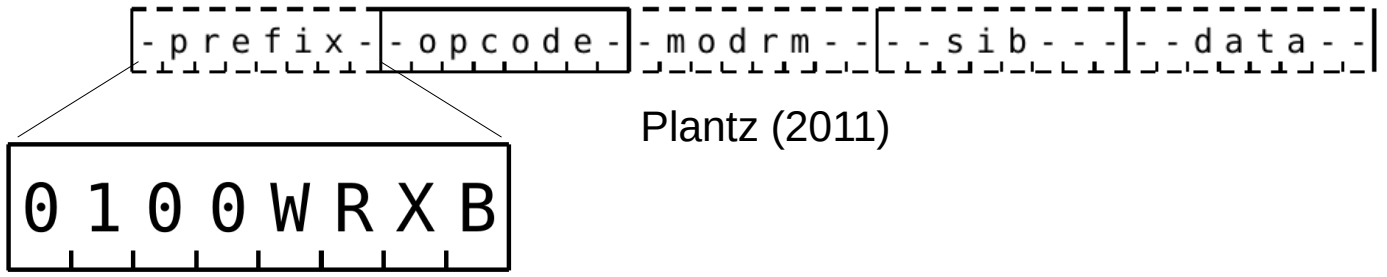
- Instruções x86-64 podem ter **entre 8 e 120 bits** de tamanho
- Formato Geral



Plantz (2011)

- Prefix → Quando inserido, modifica o comportamento da instrução
- Opcode → Código da operação
- ModRM → Especifica o modo de acesso a memória
- SIB → Localização e modo de acesso dos operandos
- Data → Imediatos

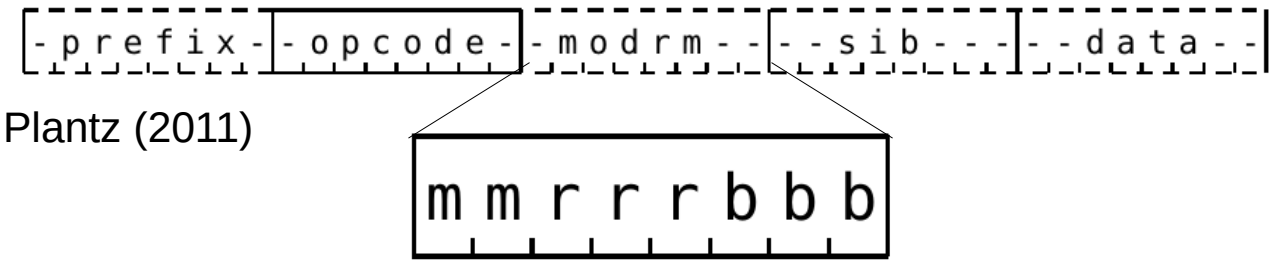
Formato de instruções



Plantz (2011)

- Dentre outros usos, o Byte de prefixo fornece 3 bits extras (REX.R, REX.X, REX.B) para estender o endereçamento dos registradores
  - Originalmente tínhamos 8 registradores, sendo necessários 3 bits para endereçar cada um
  - No modo 64 bits temos 16 registradores, e precisamos de 4 bits para endereçar
    - Como uma instrução pode precisar de até 3 regs, precisamos de 3 bits extras
- REX.W é 1 quando o operando é de 64 bits, e 0 caso contrário

Formato de instruções



Plantz (2011)

- Byte em MODRM
  - **mm**
    - 00 → Operando na memória. Endereço especificado pelo registrador bbb
    - 01 → Operando na memória. Endereço especificado pelo registrador bbb e offset imediato de 8 bits
    - 10 → Operando na memória. Endereço especificado pelo registrador bbb e offset imediato de 16 bits
    - 11 → Ambos operandos estão nos registradores especificados em rrr e bbb

Formato de instruções

- Esses são apenas alguns detalhes do formato de instruções do x86
  - O montador assembly vai nos poupar de muitas dessas complexidades
  - Se você deseja mais detalhes sobre a codificação das instruções e formatos do x86-64 leia
    - Bob Plantz. Introduction to Computer Organization: A Guide to X86-64 Assembly Language and GNU/Linux. 2011.
      - Seção 9.3
    - Intel® 64 and IA-32 Architectures Software Developer’s Manual. Intel, 2019.

De C para Assembly

- Crie um programa em C chamado doNothing.c que não faz trabalho algum, como a seguir
  - OBS.: Não inclua bibliotecas

```c
int main(){
    return 0;
}
```

- Compile no GCC com o seguinte comando

  gcc -S -O0 doNothing.c -o saida.s
  - -S para gerar o *assembly* como saída
  - -O0 (menos ó maiúsculo zero) para eliminar todas otimizações e tornar o código mais fácil de entender
  - -O (menos ó minúsculo) para especificar que o arquivo de saída é saida.s

De C para Assembly

- Assembly gerado (na minha máquina)
  - O GCC injeta aprimoramentos, alinhamentos e informações de debug
    - Mesmo com o modo debug e com otimizações desabilitadas

```
        .file   "doNothing.c"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl$0, %eax
        popq%rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size main, .-main
        .ident  "GCC: (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0"
        .section    .note.GNU-stack,"",@progbits
```

Programa doNothing

- Versão do programa doNothing.s feita manualmente
  - Somente o necessário para executar
    - Programa original de Plantz (2011)

```
    .text
    .globl  main
    .type  main, @function
main:
    pushq  %rbp         # salvar o frame pointer na pilha
    movq  %rsp, %rbp    # copiar o stack pointer para o frame pointer
    movl  $0, %eax      # o valor retornado deve estar em EAX (return 0)
    movq  %rbp, %rsp    # voltar a pilha para a posição original
    popq  %rbp          # carregar o valor salvo do frame pointer da pilha
    ret                 # retornar ao chamador
```

Antes de iniciar a análise

- Vamos utilizar o GNU Assembler (GAS)
  - Comando *as* no terminal
- Utiliza a sintaxe AT&T para a programação assembly
  - Um pouco diferente da notação Intel (disponível nos manuais Intel)
    - Você pode utilizar o montador *nasm* caso queira uma notação mais similar a utilizada nos manuais Intel
  - Notação Intel tende a ser mais simples, mas gera ambiguidades em alguns cenários, e é menos “didática”
- Uma das principais diferenças é a ordem dos operandos
  - ***instrução fonte, destino***
- Ao especificarmos que desejamos acessar o conteúdo do registrador, utilizamos % na frente do nome do registrador

O mesmo programa escrito por um programador

- Note que a sintaxe é similar a que aprendemos no MIPS
  - O rótulo main é visível globalmente
  - Além disso, é necessário informar ao montador que main é uma função
    - Podemos ter funções e dados

```
        .text
        .globl  main
        .type   main, @function
main:
        pushq   %rbp        # salvar o frame pointer na pilha
        movq    %rsp, %rbp# copiar o stack pointer para o frame pointer
        movl    $0, %eax    # o valor retornado deve estar em EAX (return 0)
        movq    %rbp, %rsp # voltar a pilha para a posição original
        popq    %rbp        # carregar o valor salvo do frame pointer da pilha
        ret                 # retornar ao chamador
```

O mesmo programa escrito por um programador

- No x86-64 temos
  - Registrador **rbp** é o **frame pointer** (Registrador $fp do MIPS)
  - Registrador **rsp** é o **stack pointer** (Registrador $sp do MIPS)
  - O R em **r**sp e **r**bp indica que estamos utilizando todos 64 bits do registrador

```
        .text
        .globl  main
        .type  main, @function
main:
        pushq  %rbp        # salvar o frame pointer na pilha
        movq  %rsp, %rbp# copiar o stack pointer para o frame pointer
        movl  $0, %eax     # o valor retornado deve estar em EAX (return 0)
        movq  %rbp, %rsp # voltar a pilha para a posição original
        popq  %rbp         # carregar o valor salvo do frame pointer da pilha
        ret                # retornar ao chamador
```

O mesmo programa escrito por um programador

- A operação **push**
  - Decrementa o valor ponteiro de pilha (rsp) pelo valor necessário
    - Lembre-se que a pilha é “ao contrário” e um decremento aloca espaço na pilha
  - **Depois** armazena o conteúdo na pilha

Aloca 8 bytes na pilha e salva os 8 bytes do rbp (stack frame) na pilha

```
        .text
        .globl  main
        .type   main, @function
main:
        pushq   %rbp        # salvar o frame pointer na pilha
        movq    %rsp, %rbp  # copiar o stack pointer para o frame pointer
        movl    $0, %eax    # o valor retornado deve estar em EAX (return 0)
        movq    %rbp, %rsp  # voltar a pilha para a posição original
        popq    %rbp        # carregar o valor salvo do frame pointer da pilha
        ret                 # retornar ao chamador
```

O mesmo programa escrito por um programador

- Noque o ***q*** no final do push
  - Muitas instruções seguem o seguinte formato instrução***S*** *fonte, destino*
  - Onde **S** é substituído por
    - b (byte) para operandos de 8 bits
    - w (word) para operandos de 16 bits
    - l (long) para operandos de 32 bits
    - q (quadword) para operandos de 64 bits
- Como o push está empilhando rbp (64 bits), realizamos um push**q**

```
        .text
        .globl  main
        .type  main, @function
main:
        pushq  %rbp
        movq  %rsp, %rbp
        movl  $0, %eax
        movq  %rbp, %rsp
        popq  %rbp
        ret
```

O mesmo programa escrito por um programador

- A instrução mov copia o conteúdo da fonte para o destino
  mov fonte, destino
  - Tanto fonte quanto destino **podem** ser um registrador, ou um endereço de memória
    - O fonte pode ainda ser um imediato
    - Fonte e destino **não podem ser ambos endereços de memória**

```
        .text
        .globl  main
        .type  main, @function
main:
        pushq  %rbp         # salvar o frame pointer na pilha
        movq  %rsp, %rbp# copiar o stack pointer para o frame pointer
        movl  $0, %eax     # o valor retornado deve estar em EAX (return 0)
        movq  %rbp, %rsp # voltar a pilha para a posição original
        popq  %rbp          # carregar o valor salvo do frame pointer da pilha
        ret                 # retornar ao chamador
```

O mesmo programa escrito por um programador

- O Linux espera que o valor retornado pelo programa ao S.O. seja devolvido em **eax**
  - **32 bits mais baixos de rax**
  - Movemos o imediato 0 para dentro de eax antes de retornar
  - Note que
    - Utilizamos mov**l**
    - O **$** indica que o valor é um **imediato**

```
        .text
        .globl  main
        .type  main, @function
main:
        pushq  %rbp        # salvar o frame pointer na pilha
        movq  %rsp, %rbp# copiar o stack pointer para o frame pointer
        movl  $0, %eax     # o valor retornado deve estar em EAX (return 0)
        movq  %rbp, %rsp # voltar a pilha para a posição original
        popq  %rbp         # carregar o valor salvo do frame pointer da pilha
        ret                # retornar ao chamador
```

O mesmo programa escrito por um programador

- Antes de acabar, devemos voltar a pilha ao estado original
  - A base da pilha original coincide com o frame pointer depois de removido o valor salvo do frame pointer da pilha
  - Primeiro Copiamos rbp (frame pointer) para rsp (stack pointer)
  - Restauramos o valor salvo do frame para rbp
    - E automaticamente o valor do stack pointer é incrementado, retornando assim ao seu valor original

```
        .text
        .globl  main
        .type  main, @function
main:
        pushq  %rbp        # salvar o frame pointer na pilha
        movq  %rsp, %rbp# copiar o stack pointer para o frame pointer
        movl  $0, %eax     # o valor retornado deve estar em EAX (return 0)
        movq  %rbp, %rsp # voltar a pilha para a posição original
        popq  %rbp         # carregar o valor salvo do frame pointer da pilha
        ret                # retornar ao chamador
```

O mesmo programa escrito por um programador

- Instrução ret retorna para quem chamou a função atual
  - O endereço de retorno deve estar no topo da pilha
  - Geralmente invocamos funções via *call*, que automaticamente salva o retorno na pilha
    - Como eram feitas as chamadas de função no MIPS? Onde era salvo o endereço de retorno?

```
        .text
        .globl  main
        .type   main, @function
main:
        pushq   %rbp        # salvar o frame pointer na pilha
        movq    %rsp, %rbp  # copiar o stack pointer para o frame pointer
        movl    $0, %eax    # o valor retornado deve estar em EAX (return 0)
        movq    %rbp, %rsp  # voltar a pilha para a posição original
        popq    %rbp        # carregar o valor salvo do frame pointer da pilha
        ret                 # retornar ao chamador
```

O mesmo programa escrito por um programador

- Instrução ret retorna para quem chamou a função atual
  - O endereço de retorno deve estar no topo da pilha
  - Geralmente invocamos funções via *call*, que automaticamente salva o retorno na pilha
    - No MIPS, fazemos um jal e endereço de retorno fica no registrador $ra

```
        .text
        .globl  main
        .type   main, @function
main:
        pushq   %rbp        # salvar o frame pointer na pilha
        movq    %rsp, %rbp# copiar o stack pointer para o frame pointer
        movl    $0, %eax    # o valor retornado deve estar em EAX (return 0)
        movq    %rbp, %rsp # voltar a pilha para a posição original
        popq    %rbp        # carregar o valor salvo do frame pointer da pilha
        ret                 # retornar ao chamador
```

O mesmo programa escrito por um programador

- Note que nesse programa, as duas únicas instruções “úteis” são
  - movl
  - ret
- As demais instruções estão ajustando o frame e a pilha
- Mas como não estamos utilizando a pilha em nosso programa diretamente, elas se tornam desnecessárias
- **Acostume-se com essas instruções**
  - Elas fazem parte do prólogo e do epílogo do programa (veremos adiante)
  - É convenção ajustar a pilha no início da chamada de toda função

Curiosidades

- Existem diversas instruções no x86-64 que tentam fazer o máximo sozinhas, para simplificar a vida do programador e economizar memória
  - O x86-64 vem de uma era onde criar programas em Assembly era comum
- leave
  - A instrução leave, por exemplo, volta a pilha para a posição original (que foi salva em rbp) e carrega o valor salvo de rbp da pilha, substituindo os comandos

    ```
    movq  %rbp, %rsp
    popq  %rbp
    ```
  - Alguns compiladores podem utilizar o leave
  - **Não vamos utilizar**
    - Nosso objetivo é entender o que está acontecendo

Curiosidades

- enter TAMANHO_FRAME, NEST_LEVEL
  - De forma similar, a instrução enter
    - Salva o rbp na pilha
    - Estabelece um novo frame pointer
    - Aloca o espaço na pilha (não fizemos isso no exemplo anterior)
  - Equivalente a
    - pushq %rbp
    - movq %rsp, %rbp
    - sub espaço_na_pilha, %rbp
  - Tudo em uma única instrução!

Curiosidades

- **Nem seu compilador, nem você devem usar enter**
  - Mais lento do que realizar as operações uma a uma (apesar de economizar memória)
  - 10 ciclos de clock para o enter, versus 6 se executarmos uma a uma (Stallings; 2012)
- Esses detalhes complicam o desenvolvimento para x86-64
- Processadores atuais mantém a instrução *enter* **devido a retrocompatibilidade**

Montando e rodando o programa

- Rode o seguinte comando

  as doNothing.s --gstabs -o doNothing.o
  - as é o GNU Assembler (GAS)
  - --gstabs inclui informações de debug que precisaremos posteriormente
    - Remova esse comando para gerar a versão “para produção” dos seus programas
  - -o para especificar o arquivo de saída
    - doNothing.o é o arquivo objeto gerado
      - Não está pronto para executar, precisa ainda de uma etapa de **linkedição**

Linkedição

- O programa para realizar a linkedição é o GNU Linker
  - Comando ld
  - Problema
    - O ld vai precisar que você especifique o caminho para todas as bibliotecas para montar o binário final, e como elas serão ligadas no programa
- Para contornar essa dificuldade, vamos utilizar o GCC
  - O GCC **não vai compilar nosso código**
  - Vai apenas **tratar automaticamente da linkedição**
    gcc doNothing.o -o doNothing

Exercício

1. Faça o programa doNothing na sua máquina. Monte o programa, realize a linkedição e execute.

2. No Linux, a variável ? armazena o retorno do último programa que foi executado.
   - Veja com o comando *echo $?*
   - Modifique o valor retornado por doNothing e veja os valores no S.O.

Utilizando o GDB

- Vamos depurar o programa com o GDB
  - GNU Debugger
- Digite o comando
  - gdb doNothing
  - Seu programa vai ser aberto para debug no GDB
- Digite li para exibir as 10 primeiras linhas do seu programa
  - li NUMERO_LINHA exibe 10 linhas, centrado em NUMERO_LINHA
- Insira um *breakpoint* na instrução que move 0 para eax (movl $0, eax)
  - br NUMERO_LINHA
  - OBS.: talvez não seja possível parar nas duas primeiras instruções

Executando

- Para executar, digite run
  - O programa vai parar no breakpoint
- Para inspecionar os registradores, digite
  - i r eax rsp rbp
    - Onde *i r* é o atalho para o comando *info register*
- Para executar a próxima instrução, utilize o comando
  - si (Single instruction)
- Para continuar até o fim ou até o próximo breakpoint
  - c (Continue)
- Para matar o processo sendo executado
  - kill
- Para limpar os breakpoints
  - clear
- Para sair do gdb
  - q

Exercício

3. Execute o programa no GDB passo a passo
   - Entenda as alterações sendo realizadas nos registradores eax, rbp e rsp

4. Modifique o programa doNothing para operar com 32 bits ao invés de 64.
   - Atenção: você não vai conseguir montar esse programa no GAS
     - O GAS assume que o programa é 64 bits, e não deixa você realizar operações de 32 bits na pilha.
     - Não se preocupe com isso (mande o programa assim mesmo)

Referências

- Bob Plantz. **Introduction to Computer Organization: A Guide to X86-64 Assembly Language and GNU/Linux.** 2011.
- **Intel® 64 and IA-32 Architectures Software Developer's Manual.** Intel, 2019.
- D. Patterson; J. Henessy. **Organização e Projeto de Computadores**: **a Interface Hardware/Software.** 5a Edição. Elsevier Brasil, 2017.
- STALLINGS, W. **Arquitetura e Organização de Computadores.** 10 ed. Prentice Hall. São Paulo, 2018.