Introdução à Software Básico: Introdução à Assembly $x64$

Departamento de Ciência da Computação
Instituto de Ciências Exatas
Universidade de Brasília

Sumário

Introdução

- Por anos o Assembly IA-32 (x86) foi utilizado para realizar tarefas críticas. Porém, nos últimos anos ele foi substituído pelo Assembly x64.
- x64 é um nome genérico para a extensão de 64 bits do x86.
- Esta extensão foi introduzida pela AMD com o nome inicial de x86-64 e depois modificado para AMD64
- A arquitetura de 64 bits da INTEL/HP (Itanium) era uma arquitetura nova (não uma extensão da x86)
- Eventualmente a INTEL adotou a mesma arquitetura da AMD, criando a arquitetura INTEL IA-32e, que hoje em dia é conhecida como EMT64.

Modelo Computacional

- De forma similar a IA-32 em x64 uma *word* tem 16 bits, uma *double word* tem 32 bits, uma *quadword* tem 64 bits.
- Um novo conceito de tipo de dados é introduzido sendo a *double quardword* com 128 bits (Já existente na extensão SSE do IA-32).
- O padrão de endereçamento segue sendo "little endian".

Modelo Computacional

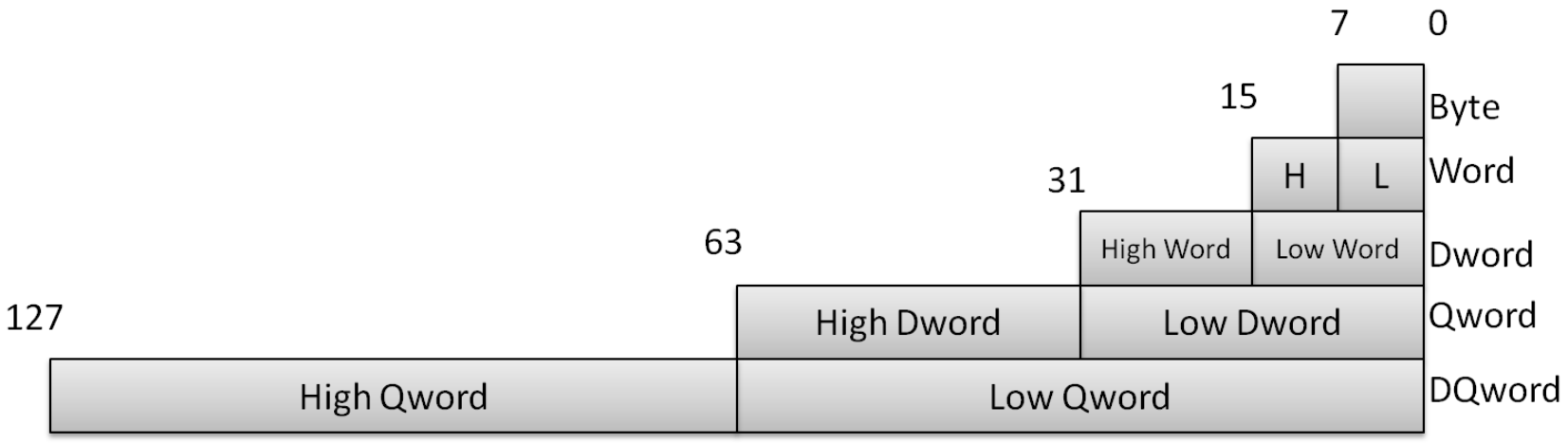


Figura: Tamanho Possíveis de dados em Registradores

Modelo Computacional

Registradores

- A arquitetura IA-32 era uma arquitetura inicialmente desenhada para ser CISC, logo tinha uma pouca quantidade de registradores.
- Com a criação dos circuitos híbridos RISC/CISC, foi mantida a quantidade de registradores nos processadores de 32 bits para manter compatibilidade
- Já na extensão de 64 bits foram introduzidos mais registradores, explorando as vantagens de uma arquitetura RISC.

Modelo Computacional

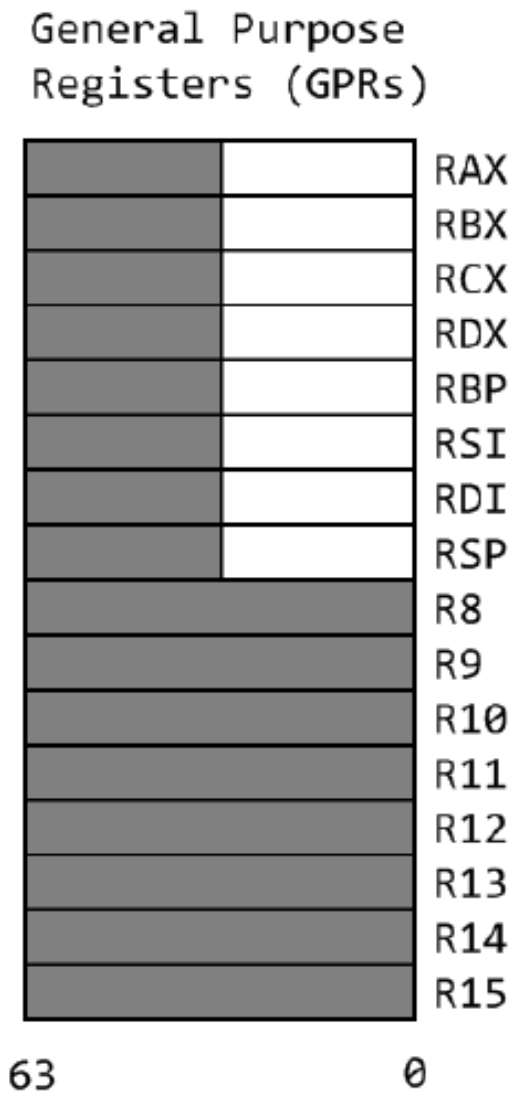


Figura: Registradores de uso geral

Registradores de uso geral

- A arquitetura x64 possui 16 registradores de uso geral. Os primeiros 8 são chamadas para manter compatibilidade de RAX,RBX,RCX,RDX,RBP,RSI,RDI e RSP.
- Substituindo a primeira letra ("R") pela letra "E", é possível acessar os últimos 32 bits do registrador (ex: EAX).
- De maneira similar podemos retirar a "R" para acessar os últimos 16 bits (ex: AX)
- Esses 16 bits podem ser acessados de forma separada como dois bytes substituindo "X" por "H" ou "L" (ex: AH, AL).
- Os novos registradores R8 a R15 podem ser acessados de forma similar da seguinte forma: R8 (qword), R8D (lower dword), R8W (lowest word), R8B (lowest byte - MASM) ou R8L (lowest byte - INTEL).
- Notar que **não** tem como acessar o R8H (em nenhum dos registradores R8 a R15).

Registradores especiais

- O registrador de 64 bits RIP possui o endereço virtual da próxima instrução a ser executada.
- O registrador RSP aponta o topo da pilha, que cresce em direção de endereços menores.
- De forma similar a IA-32, a arquitetura x64 possui registradores para apontar segmentos. Porém, o endereçamento é feito de forma diferente.
- O registrador RFLAGS é uma extensão do antigo registrador EFLAGS. Porém os 32 bits a mais do registrador RFLAGS não são utilizadas até o momento.

Modelo Computacional

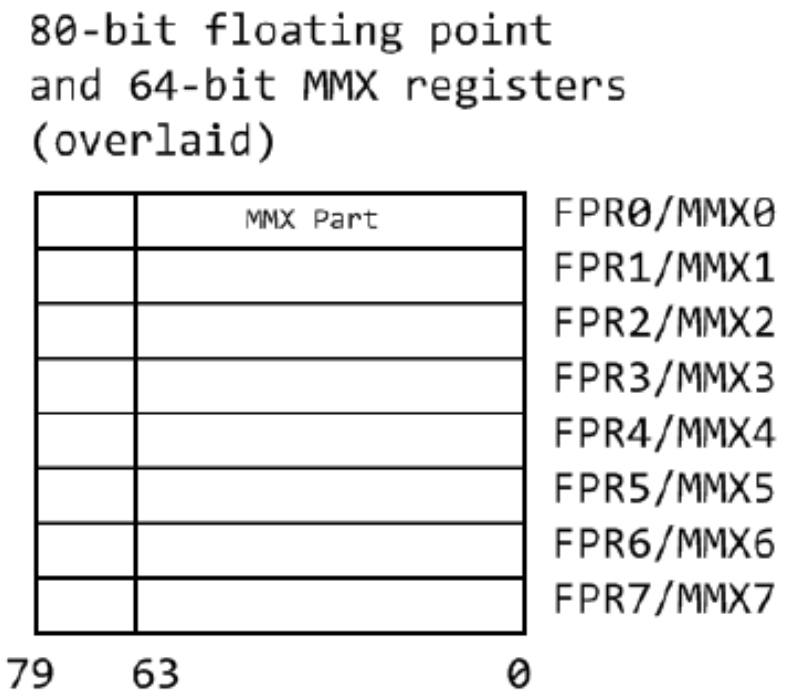


Figura: Registradores de ponto flutuante

Registradores de ponto flutuante

- A FPU (*floating point unit*) possui oito registradores FPR0-FPR7, um registador de status e um de controle.
- A FPU segue o padrão IEEE 754, incluindo precisão estendida.
- Estes registros são sobrepostos com os MMX, logo uma solução que use a FPU e os registradores MMX deve ser feita com muito cuidado.

Data Type	Length	Precision (bits)	Decimal digits Precision	Decimal Range
Single Precision	32	24	7	1.18*10^-38 to 3.40*10^38
Double Precision	64	53	15	2.23 *10^-308 to 1.79*10^308
Extended Precision	80	64	19	3.37*10^-4932 to 1.18*10^4932

Modelo Computacional

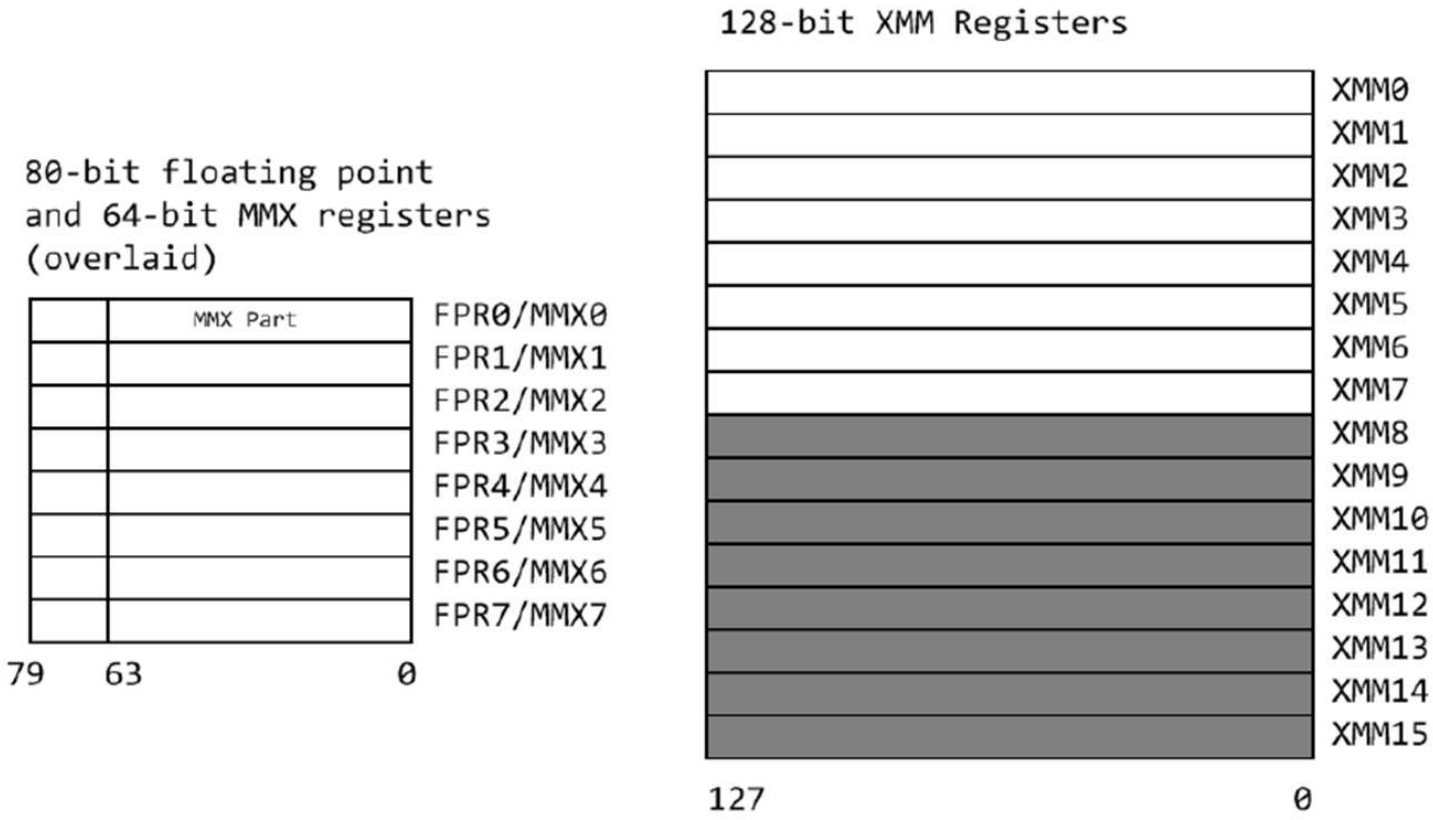


Figura: Registradores SIMD

Registradores SIMD

- A tecnologia SIMD (*single instruction multiple data*) foi introduzida pela INTEL em 1999 para o Pentium III
- Esta tecnologia permite que uma única instrução possa acessar de forma paralela múltiplos dados
- A tecnologia original da INTEL é a SIMD SSE.
- A SSE (*streaming SIMD extention*), utilizava os registradores XMM0 a XMM7. Esses registradores de 128 bits foram expandidos pela AMD incluindo agora XMM0 a XMM15
- Além disso a AMD crio a SIMD MMX, onde outro grupo de instruções podem operar de forma similar utilizando os registradores MMX0 a MMX7 (de 80 bits). Possibilitando dos grupos de instruções paralelas simultâneas.

Registradores SIMD

- Os registradores SIMD permitem operações com dados inteiros e ponto flutuante.

Technology	Register size/type	Item type	Items in Parallel
MMX	64 MMX	Integer	8, 4, 2, 1
SSE	64 MMX	Integer	8,4,2,1
SSE	128 XMM	Float	4
SSE2/SSE3/SSSE3...	64 MMX	Integer	2,1
SSE2/SSE3/SSSE3...	128 XMM	Float	2
SSE2/SSE3/SSSE3...	128 XMM	Integer	16,8,4,2,1

Intel AVX

- Em 2008 a Intel apresentou a proposta para a extensão AVX (*Advanced Vector Extension*) para a arquitetura x64
- Essa extensão foi utilizada pela primeira vez em 2011 pela Intel e AMD.
- As gerações mais novas da INTEL e da AMD possuem a extensão AVX.
- A extensão AVX estende os registradores XMM0 a XMM15 de 128 bits para 256 bits (sendo renomeados para YMM0 a YMM15, os 128 bits menos significativos podem ser acessado mediante o nome antigo de XMM).
- Além disso a quantidade de registradores também é estendido tendo YMM0 a YMM31 registradores.
- A extensão AVX-512 estende os registrados YMM de 256 para 512 bits, mudando o nome para ZMM0 a ZMM32 (Os registradores menores podem ser acessados mediante o nome YMM e XMM).

Modos de Endereçamento

- Imediato: add EAX, 14
- Registrador: ADD R8L, AL;
- Indireto: Possibilita um offset de 8,16 ou 32 bits. Utilizando um registrador (ou número imediato) como base, outro como index e outro como escala. Exemplos:
  - MOV R8W, 1234[8*RAX+RCX]; move a palavra de 8 bits do endereço 8*RAX+RCX+1234 para o registrador R8W
  - Existem também as seguintes formas equivalentes:
    - MOV ECX, dword table[RBX][RDI]
    - MOV ECX, dword table[RDI][RBX]
    - MOV ECX, dword table[RBX+RDI]
    - MOV ECX, dword [table+RBC+RDI]

Modos de Endereçamento

- Existe o novo mode de endereçamento RIP-relativo. Onde é possivel acessar dados apontados pelo RIP. Exemplos:
  - MOV DWORD [RIP+10], 1 ;coloca 1 10 bytes depois do final da instrução
  - MOV DWORD [symb+RIP], 1 ;some o endereço de symb ao endereço do final da instrução e coloca 1 nessa posição

Endereçamento de Memória

- O arquitetura x64 a principio permite endereçamento de 2^{64} bytes de dados, ou seja 16 exabytes (18 446 744 073 709 551 616 bytes).
- Porém, AMD decidiu utilizar somente 48 bits para endereçar e não 64. Os bits 48 a 63 devem ser uma copia do bit 47 se não gera uma exceção de endereçamento.
- Com 48 bits a arquitetura x64 consegue endereçar 256 TB. Suficiente para qualquer PC.
- Porém, os OS limitam isso:
  - Windows 8: 128 GB
  - Windows 8 Professional/Enterpise 512 GB
  - Red Hat v.5: 1 TB
  - Red Hat v.6: 64 TB

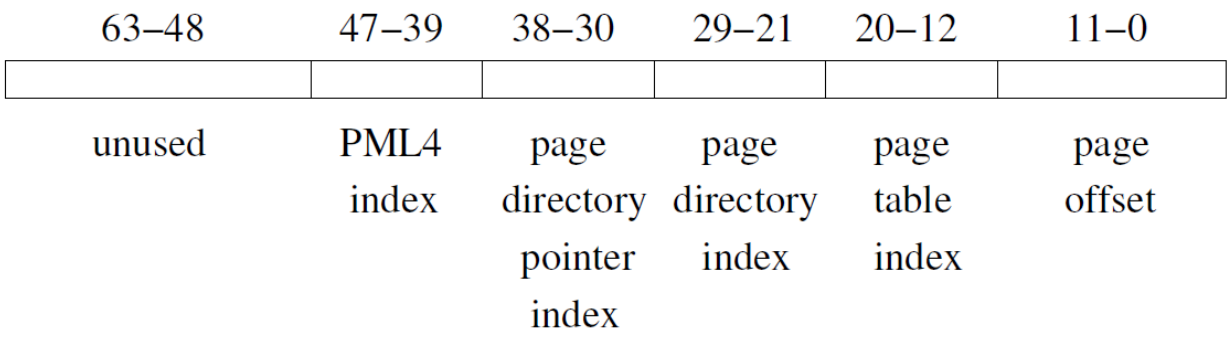
Mapeamento de Memória

- Da mesma forma que na arquitetura IA-32 o endereçamento é realizado mediante memória virtual. Possibilitando a criação de páginas.
- O endereço físico do nível mais acima de mapeamento é indicado pelo registrador CR3 (*control register 3*).
- Existem 4 níves de mapeamento
- O registrador CR3 aponta a posição em memória onde esta a tabela PML4 (Page Map Level 4)

Mapeamento de Memória

- Cada página possui 2^{12} (4KB). Porém isso pode ser modificado.
- Cada endereço possui 8 bytes
- Logo numa página é possível armazenar 512 (2^9) endereços de memória
- Assim, o endereço é formado pelo offset de 12 bits para indicar a posição na página e 4 mapeamentos de 9 bits criando 2^{36} páginas possiveis.

Assembly x64



Mapeamento de Memória

- Bits 47-39 índice da tabela PLM4
- Bits 38-30 ponteiro para a tabela de diretório de páginas
- Bits 29-31 ponteiro para o diretório de páginas
- Bits 20-12 ponteiro para a página
- Bits 11-0 offset dentro da página

Exemplo de Mapeamento de Memória

- Vamos assumir que CR3 aponta a posição de memória 0x4ffff000

	PML4 at 0x4ffff000
0	0x3466000
1	0x3467000
2	0x3468000
	...
511	unused

- Vamos tentar traduzir um endereço virtual
- Bits 47-39 do nosso endereço virtual são 0x001
- Logo, acessamos a segunda entrada (index 1) tendo o endereço 0x3467000
- Notar que nosso OS deixou alguns endereços não utilizados.

Exemplo de Mapeamento de Memória

Page Directory Pointer Table at `0x3467000`

0	`0x3587000`
1	unused
2	`0x3588000`
	...
511	unused

- Bits 38-30 do nosso endereço virtual são 0x002
- Logo, logo temos que acessar a terceira tabela de diretórios no endereço 0x3588000

Exemplo de Mapeamento de Memória

Page Directory Table at **0x3588000**

0	**0x3678000**
1	**0x3579000**
2	unused
	...
511	unused

- Bits 29-21 do nosso endereço virtual são 0x000
- Logo, logo temos que acessar o primeiro diretório no endereço 0x3678000

Exemplo de Mapeamento de Memória

Page Table
at `0x3678000`

0	`0x5788000`
1	`0x5789000`
2	`0x578a000`
	...
511	`0x5799000`

- Bits 20-12 do nosso endereço virtual são 0x1ff
- Logo, logo temos que acessar a página no endereço 0x5799000
- Temos que somar o offset, assumindo que os bits 11-0 do nosso endereço virtual eram 0xfa8
- Então nosso endereço físico é 0x5799fa8.

System Calls em x64

- Colocar o número da chamada ao sistema em RAX
- Os argumentos, em ordem, em RDI, RSI, RDX, R10, R8 e R9.
- Fazer a chamada "syscall"
- A chamada modifica os valores de RCX e R11, os outros registradores mantém o seu valor antes da chamada

Exemplo de Mapeamento de Memória

```
        .global _start

        .text
_start:
        # write(1, message, 13)
        mov     rax, 1                  # system call 1 is write
        mov     rdi , 1                 # file handle 1 is stdout
        mov     rsi , message           # address of string to output
        mov     rdx, 13                 # number of bytes
        syscall                         # invoke operating system to do the write

        # exit(0)
        mov     rax, 60                 # system call 60 is exit
        xor     rdi, rdi                # we want return code 0
        syscall                         # invoke operating system to exit
message:
        .ascii  "Hello, world\n"
```

Estrutura de dados

- O Assembly x64 permite a criação de registros (*structs*)
- No compilador yasm deve–se iniciar o registro com “struc” e terminar com “endstruc”
- É possivel fazer array de strucs.

Registros em x64

```
        segment .data
name    db      "Calvin", 0
address db      "12 Mockingbird Lane",0
balance dd      12500
        struc   Customer
c_id      resd    1
c_name    resb    64
c_address resb    64
c_balance resd    1
        endstruc
c       dq      0
```

- Neste casso é possível acessar os campos como “Customer.id”
- Também é automaticamente definido “Customer_size” como o tamanho do registro, podendo faze alocação dinâmica de memória.

Registros em x64

```
        segment .text
        global  main
        extern  malloc, strcpy
main:   push    rbp
        mov     rbp, rsp
        sub     rsp, 32
        mov     rdi, Customer_size
        call    malloc
        mov     [c], rax    ; save the pointer
        mov     [rax+c_id], dword 7
        lea     rdi, [rax+c_name]
        lea     rsi, [name]
        call    strcpy
        mov     rax, [c]    ; restore the pointer
        lea     rdi, [rax+c_address]
        lea     rsi, [address]
        call    strcpy
        mov     rax, [c]    ; restore the pointer
        mov     edx, [balance]
        mov     [rax+c_balance], edx
        xor     eax, eax
        leave
        ret
```

Próxima Aula

Revisão para a prova