

Programando junto com C

Aprendendo a mesclar Assembly e C

Se você leu o conteúdo do livro até aqui já tem uma boa base para entender como o Assembly x86 funciona e como usá-lo. Também já tem uma boa noção do que está fazendo, entende bem o que o assembler faz e o que ele está produzindo como saída, sabe como efetuar cálculos em paralelo usando SSE inclusive com valores de ponto flutuante.

Em outras palavras você já tem a base necessária para realmente entender como as coisas funcionam, não decoramos instruções aqui mas sim entendemos as coisas em seu âmago. Agora está na hora de dar um passo a frente e entender como usar Assembly de uma maneira útil no "mundo real", vamos aprender a usar C e Assembly juntos afim de escrever programas.

Já estamos fazendo isso desde o começo mas não entramos em muitos detalhes pois eu queria que inicialmente o foco fosse em entender como as coisas funcionam, essa é a parte legal 😊.

Ferramentas

Como já mencionado antes vamos usar o GCC para compilar nossos códigos em C. Mas diferente dos capítulos anteriores que usamos o NASM, neste aqui vamos usar o assembler GAS com sintaxe da AT&T **porque** assim aprendemos a ler código nessa sintaxe e a usar o GAS ao mesmo tempo.

Por convenção a gente usa a extensão `.s` (ao invés de `.asm`) para código ASM com sintaxe da AT&T, então é a extensão que irei usar daqui em diante para nomear os arquivos.

Assim como fizemos em [A base](#) aqui está um código de teste para garantir que o seu ambiente está correto:

```
main.c
```

```
#include <stdio.h>

int assembly(void);

int main(void)
{
    printf("Resultado: %d\n", assembly());
    return 0;
}
```

assembly.s

```
.text
.globl assembly
assembly:
    movl $777, %eax
    ret
```

O nome do executável do GAS é **as** e quando você instala o GCC ele vem junto, então você já tem ele instalado aí. Já pode tentar compilar com:

```
$ as assembly.s -o assembly.o
$ gcc main.c -c -o main.o
$ gcc *.o -o test
```

Caso tenha algum problema e precise de ajuda, pode entrar no [fórum do Mente Binária](#) e fazer uma pergunta.

Vendo o código de saída do GCC

Ao usar o GCC é possível passar o parâmetro `-masm=intel` para que o compilador gere código Assembly na sintaxe da Intel, onde por padrão ele gera código na sintaxe da AT&T. Você pode ver o código de saída da seguinte forma:

```
$ gcc main.c -o main.s -S -masm=intel -fno-asynchronous-unwind-tables
```

Onde a *flag* `-S` faz com que o compilador apenas compile o código, sem produzir o arquivo objeto de saída e ao invés disso salvando o código em Assembly. Pode ser útil fazer isso para aprender mais sobre a sintaxe do GAS.

A *flag* `-fno-asynchronous-unwind-tables` serve para desabilitar as [diretivas CFI ↗](#) e melhorar a leitura do código de saída. Essas diretivas servem para gerar informação útil para um depurador mas para fins de leitura do código não precisamos delas.

Você também pode habilitar as otimizações do GCC com a opção `-O2` assim o código de saída será otimizado. Pode ser interessante fazer isso para aprender alguns truques de otimização.

Sintaxe do GAS

Aprendendo a sintaxe AT&T e a usar o GAS

O GNU assembler (GAS) usa por padrão a sintaxe AT&T e neste tópico irei ensiná-la. Mais abaixo irei ensinar a diretiva usada para usar sintaxe Intel meramente como curiosidade e caso prefira usá-la.

Diferenças entre sintaxe Intel e AT&T

A primeira diferença notável é que o operando destino nas instruções de sintaxe Intel é o mais à esquerda, o primeiro operando. Já na sintaxe da AT&T é o inverso, o operando mais à direita é o operando destino. Conforme exemplo:

```
# Isso é um comentário.

# Sintaxe AT&T      # Sintaxe Intel

mov $123, %eax      # mov eax, 123
mov $0x0A, %eax     # mov eax, 0x0A
mov $0b1010, %eax   # mov eax, 0b1010
mov $'A', %eax      # mov eax, 'A'
```

E como já pode ser observado valores literais precisam de um prefixo `$`, enquanto os nomes dos registradores precisam do prefixo `%`.

Tamanho dos operandos

Na sintaxe da Intel o tamanho dos operandos é especificado com base em palavras-chaves que são adicionadas anteriormente ao operando. Na sintaxe AT&T o tamanho do operando é especificado por um sufixo adicionado a instrução, conforme tabela abaixo:

Sufixo	Tamanho	Palavra-chave equivalente no NASM
B	byte (8 bits)	byte

W	word (16 bits)	word
L	long/doubleword (32 bits)	dword
Q	quadword (64 bits)	qword
T	ten word (80 bits)	tword

Exemplos:

# AT&T	# Intel
movl \$5, %eax	# mov dword eax, 5
movb '\$A', (%ebx)	# mov byte [ebx], 'A'

Assim como o NASM consegue identificar o tamanho do operando quando é usado um registrador e a palavra-chave se torna opcional, o mesmo acontece no GAS e o sufixo também é opcional nesses casos.

Far jump e call

Na sintaxe Intel saltos e chamadas distantes são feitas com `jmp far [etc]` e `call far [etc]` respectivamente. Na sintaxe da AT&T se usa o prefixo **L** nessas instruções, ficando: `ljmp` e `lcall`.

Endereçamento

Na sintaxe Intel [o endereçamento](#) é bem intuitivo já que ele é escrito em formato de expressão matemática. Na sintaxe AT&T é um pouco mais confuso e segue o seguinte formato:

```
segment:displacement(base, index, scale).
```

Exemplos com o seu equivalente na sintaxe da Intel:

# AT&T	# Intel
mov my_var, %eax	# mov eax, [my_var]
mov 5(%ebx), %eax	# mov eax, [ebx + 5]
mov 5(%ebx, %esi), %eax	# mov eax, [ebx + esi + 5]
mov 5(%ebx, %esi, 4), %eax	# mov eax, [ebx + esi*4 + 5]
mov (, %esi, 4), %eax	# mov eax, [esi*4]
mov (%ebx), %eax	# mov eax, [ebx]
mov %es:(%ebx), %eax	# mov eax, [es:ebx]
mov %es:5(%ebx), %eax	# mov eax, [es:ebx + 5]
mov my_var(%rip), %rax	# mov rax, [rel my_var]

Como demonstrado no último exemplo o **endereço relativo** na sintaxe do GAS é feito explicitando RIP como base, enquanto na sintaxe do NASM isso é feito usando a palavra-chave `rel`.

Saltos

Na sintaxe da AT&T os saltos para endereços armazenados na memória devem ter um `*` antes do rótulo para indicar que o salto deve ocorrer para o endereço que está armazenado naquele endereço de memória. Sem o `*` o salto ocorre para o rótulo em si. Exemplo:

# AT&T	# Intel
jmp my_code	# jmp my_code
jmp *my_pointer	# jmp [my_pointer]

Saltos que especificam segmento e *offset* separam os dois valores por vírgula. Como em:

# AT&T	# Intel
jmp \$1234, \$5678	# jmp 1234:5678

Aprendendo a usar o GAS

As diretivas do GAS funcionam de maneira semelhante as diretivas do NASM com a diferença que todas elas são prefixadas por um ponto.

Comentários

No GAS comentários de múltiplas linhas podem ser escritos com `/*` e `*/` assim como em C. Comentários de uma única linha podem ser escritos com `#` ou `//`.

Pseudo-instruções de dados

No NASM [existem as pseudo-instruções](#) `db`, `dw`, `dd`, `dq` etc. que servem para despejar bytes no arquivo binário de saída. No GAS isso é feito usando as seguintes pseudo-instruções:

Pseudo-instrução	Tipo do dado (tamanho em bits)	Equivalente no NASM
<code>.byte</code>	byte (8 bits)	<code>db</code>
<code>.short</code> <code>.hword</code> <code>.word</code>	word (16 bits)	<code>dw</code>
<code>.long</code> <code>.int</code>	doubleword (32 bits)	<code>dd</code>
<code>.quad</code>	quadword (64 bits)	<code>dq</code>
<code>.float</code> <code>.single</code>	Single-precision floating-point (32 bits)	<code>dd</code>
<code>.double</code>	Double-precision floating-point (64 bits)	<code>dq</code>
<code>.ascii</code>		

<code>.string</code> <code>.string8</code>	String (8 bits cada caractere)	db
<code>.asciz</code>	Mesmo que <code>.ascii</code> porém com um terminador nulo no final	-
<code>.string16</code>	String (16 bits cada caractere)	-
<code>.string32</code>	String (32 bits cada caractere)	-
<code>.string64</code>	String (64 bits cada caractere)	-

Exemplos:

```
msg1: .ascii "Hello World\0"
msg2: .asciz "Hello World"

value1: .byte 1, 2, 3, 4, 5
value2: .float 3.1415
```

Diretivas de seções e alinhamento

O GAS tem diretivas específicas para declarar algumas seções padrão. Conforme tabela:

GAS	Equivalente no NASM
<code>.data</code>	<code>section .data</code>
<code>.bss</code>	<code>section .bss</code>
<code>.text</code>	<code>section .text</code>

Porém ele também tem a diretiva `.section` que pode ser usada de maneira semelhante a `section` do NASM. Os atributos da seção porém são passados em formato de *flags* em uma string como segundo argumento. As *flags* principais são `w` para dar permissão de escrita e `x` para dar permissão de execução. Exemplos:

# GAS	# NASM
<code>.section .mysection, "wx"</code>	<code># section .mysection write exec</code>
<code>.section .rodata</code>	<code># section .rodata</code>
<code>.text</code>	<code># section .text</code>
<code>.section .text</code>	<code># section .text</code>

A diretiva `.align` pode ser usada para alinhamento dos dados. Você pode usá-la no início da seção para alinhar a mesma, conforme exemplo:

```
.section .rodata
.align 16

# Equivalente no NASM: section .rodata align=16
```

Usando sintaxe Intel

A diretiva `.intel_syntax` pode ser usada para habilitar a sintaxe da Intel para o GAS. Opcionalmente pode-se passar um parâmetro `noprefix` para desabilitar o prefixo `%` dos registradores.

Uma diferença importante da sintaxe Intel do GAS em relação ao NASM é que as palavras-chaves que especificam o tamanho do operando precisam ser seguidas por `ptr`, conforme exemplo abaixo:

```
.intel_syntax noprefix
.text
example:
    mov byte ptr [ebx], 7
    mov word ptr [ebx], 7
    mov dword ptr [ebx], 7
    mov qword ptr [ebx], 7
    ret
```

Exemplo de código na sintaxe AT&T

O exemplo abaixo é o mesmo apresentado no tópico sobre [instruções de movimentação SSE](#) porém reescrito na sintaxe do GAS/AT&T:

main.c

```
#include <stdio.h>

void assembly(float *array);

int main(void)
{
    float array[4];
    assembly(array);

    printf("%f, %f, %f, %f\n", array[0], array[1], array[2], array[3]);
    return 0;
}
```

assembly.s

```
.section .rodata
.align 16
local_array: .float 1.23
              .float 2.45
              .float 3.67
              .float 4.89

.text
.globl assembly
assembly:
    movaps local_array(%rip), %xmm5
    movaps %xmm5, (%rdi)
    ret
```

Convenção de chamada da System V ABI

Aprendendo sobre a convenção de chamada do C usada no Linux.

Sistemas [UNIX-Like](#) [↗], incluindo o Linux, seguem a padronização da System V ABI (ou SysV ABI). Onde ABI é sigla para ***A**pplication **B**inary **I**nterface* (Interface binária de aplicação) que é basicamente uma padronização que dita como código binário deve ser escrito e executado no sistema operacional. Uma das coisas que a SysV ABI padroniza é a convenção de chamada utilizada em cada arquitetura de processador.

Neste tópico vamos aprender sobre a convenção de chamada da SysV ABI e o tamanho dos tipos de dados usados na linguagem C.

Convenção de chamada em x86-64

Registradores

- Os registradores RBP, RBX, RSP e R12 até R15 são considerados como pertencentes a função chamadora. Isto é, se a função que foi chamada precisar modificar esses registradores ela obrigatoriamente precisa preservar seus valores e antes de retornar restaurá-los para o valor anterior. Todos os outros registradores podem ser modificados livremente pela função chamada. Portanto não espere que esses outros registradores tenham seu valor preservado ao chamar uma função.
- A *Direction Flag* (DF) no [RFLAGS](#) precisa obrigatoriamente estar zerada ao chamar ou retornar de uma função.

Stack frame

Cada função chamada pode (se precisar) reservar um pedaço [da pilha](#) para ser usada como memória local da função e pode, por exemplo, ser usada para alocar variáveis locais. Esse espaço é chamado de *stack frame* e o código que aloca e desaloca o *stack frame* é chamado de prólogo e epílogo respectivamente. Exemplo:

```
assembly.s
```

```
.text
.globl assembly
assembly:
    sub $8, %rsp


    movl $12344, (%rsp)    # var_0
    movl $1, 4(%rsp)       # var_4

    mov (%rsp), %eax
    add 4(%rsp), %eax

    add $8, %rsp
    ret
```

O espaço de 128 bytes antes do endereço apontado por RSP é uma região chamada de **redzone** que por convenção pode ser usada por funções folha (*leaf*), que são funções que não chamam outras funções. Ou então pode ser usada em qualquer função onde o valor não precise ser preservado após chamar outra função.

O endereço entre `-128(%rsp)` e `-1(%rsp)` pode ser usado livremente sem a necessidade de alocar um *stack frame*.

 Vale lembrar que [CALL](#) empilha o endereço de retorno, portanto ao chamar uma função `0(%rsp)` aponta para o endereço de retorno da mesma.

Passagem de parâmetros

Os parâmetros inteiros (e ponteiros) são passados em [registradores de propósito geral](#) na seguinte ordem: RDI, RSI, RDX, RCX, R8 e R9. Parâmetros *float* ou *double* são passados nos registradores XMM0 até XMM7 como [valores escalares](#) (na parte menos significativa do registrador).

Caso a função precise de mais argumentos e os registradores acabem, os demais argumentos serão empilhados na ordem **inversa**. Por exemplo caso uma função precise de 9 argumentos inteiros eles seriam definidos na seguinte ordem pela função chamadora:

```
mov $1, %rdi
mov $2, %rsi
mov $3, %rdx
mov $4, %rcx
mov $5, %r8
mov $6, %r9

push $9
push $8
push $7
call my_function
add $24, %esp
```

Assim que a função fosse chamada `8(%rsp)`, `16(%rsp)` e `24(%rsp)` apontariam para os argumentos 7, 8 e 9 respectivamente.

⚠ A função **chamadora** (*caller*) precisa garantir que o último valor empilhado esteja em um endereço alinhado por 16 bytes.

A função **chamadora** é a responsável por remover os argumentos empilhados da pilha.

Retorno de valores

- No caso do retorno de estruturas (*structs*) a função chamadora precisa alocar o espaço necessário para a *struct* e passar o endereço do espaço no registrador RDI como se fosse o primeiro argumento para a função (os outros argumentos usam RSI em diante). A função então precisa retornar o mesmo endereço passado por RDI em RAX.
- O retorno de valores inteiros e ponteiros é feito no registrador RAX.
- Valores *float* ou *double* são retornados no registrador XMM0 na parte menos significativa.

Convenção de chamada em IA-32

Registadores

- Os registradores EBX, EBP, ESI, EDI e ESP precisam ter seus valores preservados pela função chamada. Os demais registradores de propósito geral podem ser usados livremente.
- A *Direction Flag* (DF) no EFLAGS precisa obrigatoriamente estar zerada ao chamar ou retornar de uma função.

Stack frame

O *stack frame* em IA-32 funciona da mesma maneira que o *stack frame* em x86-64, com a diferença de que não existe *redzone* em IA-32 e toda função que precisar de memória local precisa obrigatoriamente construir um *stack frame*.

Vale lembrar que cada valor inserido na *stack* em IA-32 tem 4 bytes de tamanho, enquanto em x86-64 cada valor tem 8 bytes de tamanho.

Passagem de parâmetros

Os argumentos da função são empilhados na ordem inversa, assim como ocorre em x86-64 quando os registradores acabam. Conforme exemplo:

```
push $4
push $3
push $2
push $1
call my_function
add $16, %esp
```

Assim que a função é chamada `4(%esp)`, `8(%esp)`, `12(%esp)` e `16(%esp)` apontam para os argumentos 1, 2, 3 e 4 respectivamente.

⚠ A função **chamadora** precisa garantir que o último valor empilhado esteja em um endereço alinhado por 16 bytes.

A função **chamadora** é a responsável por remover os argumentos empilhados da pilha.

Retorno de valores

- Retorno de *struct* é feito de maneira semelhante do x86-64. Um ponteiro para a região de memória para gravar os dados da *struct* é passado como primeiro argumento para a função (o último valor a ser empilhado). É obrigação da função chamada fazer o *pop* desse ponteiro e retorná-lo em EAX.
- Valores inteiros e ponteiros são retornados em EAX.
- Valores *float* ou *double* são retornados em ST0 (ver [Usando instruções da FPU](#)).

Prólogo e epílogo

Existe uma convenção de escrita do prólogo e do epílogo da função que se trata de preservar o antigo valor de ESP/RSP no registrador EBP/RBP, e depois subtrair ESP/RSP para alocar o *stack frame*. Conforme exemplo:

```
example:
    push %rbp
    mov %rsp, %rbp
    sub $16, %rsp

    # etc...

    mov %rbp, %rsp
    pop %rbp
    ret
```

Também existe a instrução `leave` que pode ser usada no epílogo. Ela basicamente faz a operação de `mov %rbp, %rsp` e `pop %rbp` em uma única instrução (também pode ser usada em 32 e 16 bits atuando com EBP/ESP e BP/SP respectivamente).

Mas como já foi demonstrado em um exemplo mais acima isso não é obrigatório e podemos apenas incrementar e subtrair ESP/RSP no prólogo e no epílogo. Código otimizado gerado pelo GCC costuma apenas fazer isso, já código com a otimização desligada costuma gerar o prólogo e epílogo "clássico".

Tamanho dos tipos da linguagem C

A tabela abaixo lista os principais tipos da linguagem C e seu tamanho em bytes no IA-32 e x86-64. Como também exibe em qual registrador o tipo deve ser retornado.

Tipo	Tamanho IA-32	Tamanho x86-64	Registrador de retorno IA-32	Registrador de retorno x86-64
_Bool char signed char unsigned char	1	1	AL	AL
short signed short unsigned short	2	2	AX	AX
int signed int unsigned int long signed long unsigned long enum	4	4	EAX	EAX
long long signed long long long unsigned long long	8	8	*EDX:EAX	RAX
Ponteiros	4	8	EAX	RAX
float	4	4	ST0	XMM0
double	8	8	ST0	XMM0

*No registrador EDX é armazenado os 32 bits mais significativos e em EAX os 32 bits menos significativos.

**O tipo `long double` ocupa na memória o espaço de 12 e 16 bytes por motivos de alinhamento, mas na verdade se trata de um *float* de 80 bits (10 bytes).

Convenções de chamada no Windows

Aprendendo sobre as convenções de chamada usadas no Windows (x64, cdecl e stdcall).

O Windows tem suas próprias convenções de chamadas e o objetivo desse tópico é aprender sobre as três principais que dizem respeito à linguagem C.

Convenção de chamada x64

Essa é a convenção de chamada padrão usada em x86-64 e portanto é essencial aprendê-la caso vá programar no Windows diretamente em Assembly.

Registradores

Os registradores RBX, RBP, RDI, RSI, RSP, R12 até R15 e XMM6 até XMM15 devem ser preservados pela função **chamada** (*callee*). Caso a função chamada precise alterar o valor de algum desses registradores ela tem a obrigação de preservar o valor anterior e restaurá-lo antes de retornar.

Os demais registradores são considerados voláteis, isto é, podem ter seu valor alterado quando uma chamada de função é efetuada. A função chamada pode modificar o valor dos registradores voláteis livremente.

Passagem de parâmetros

- Os primeiros quatro argumentos inteiros ou ponteiros são passados nos seguintes registradores e na mesma ordem: RCX, RDX, R8 e R9. Os demais argumentos devem ser empilhados na ordem **inversa**.
- Os primeiros quatro argumentos *float* ou *double* são passados nos registradores XMM0 até XMM3 como [valores escalares](#). Os demais também são empilhados na ordem inversa.
- *Structs* e *unions* de 8, 16, 32 ou 64 bits são passados como se fossem inteiros do respectivo tamanho. Se forem de outro tamanho a função chamadora deve então

passar um ponteiro para a *struct/union* que será armazenada em uma memória alocada pela própria função chamadora. Essa memória **deve** estar em um endereço alinhado por 16 bytes.

A função **chamadora** (*caller*) é responsável por alocar um espaço de 32 bytes na pilha chamado de *shadow space*. Ele é alocado com o intuito de ser usado pela função chamada (*callee*) para armazenar os parâmetros passados em registradores caso seja necessário, por exemplo caso a função chamada precise usar esses registradores com outro intuito. Esse espaço vem antes mesmo do primeiro parâmetro empilhado.

Exemplo de protótipo de função:

```
int sum(int a, int b, int c, int d, int e, int f);
```

Assim que a função fosse chamada ECX, EDX, R8D e R9D armazenariam os parâmetros `a`, `b`, `c` e `d` respectivamente. O parâmetro `f` seria empilhado seguido do parâmetro `e`.

O `0(%rsp)` seria o endereço de retorno. O espaço entre `8(%rsp)` e `40(%rsp)` é o *shadow space*. `40(%rsp)` apontaria para o parâmetro `e`, enquanto `48(%rsp)` apontaria para o parâmetro `f`. Como na demonstração abaixo:

```
mov %ecx, 8(%rsp) # Armazenando o parâmetro A no shadow space
mov %edx, 16(%rsp) # Parâmetro B
mov %r8d, 24(%rsp) # Parâmetro C
mov %r9d, 32(%rsp) # Parâmetro D

# Parâmetro E: 40(%rsp)
# Parâmetro F: 48(%rsp)
```

Retorno de valores

- Valores inteiros e ponteiros são retornados em RAX.
- Valores *float* ou *double* são retornados no registrador XMM0.
- O retorno de *structs* é feito com a função chamadora alocando o espaço de memória necessário para a *struct*, ela então passa o ponteiro para esse espaço como primeiro

argumento para a função em RCX. A função chamada (*callee*) deve retornar o mesmo ponteiro em RAX.

Convenção de chamada cdecl

A convenção de chamada `__cdecl` é a convenção padrão usada em código escrito em C na arquitetura IA-32 (x86).

Registradores

Apenas os registradores EAX, ECX e EDX são considerados voláteis, ou seja, registradores que podem ser modificados livremente pela função chamada. Todos os demais registradores precisam ser preservados e restaurados antes do retorno da função.

Passagem de parâmetros

Todos os parâmetros são passados na pilha e devem ser empilhados na ordem **inversa**. A função **chamadora** (*caller*) é a responsável por remover os argumentos da pilha após a função retornar.

Exemplo:

```
push $3
push $2
push $1
call my_function
add $12, %esp

# 12 é o tamanho em bytes dos três valores empilhados
```

Retorno de valores

- Valores inteiros ou ponteiros são retornados em EAX.

- Valores *float* ou *double* são retornados em ST0.
- O retorno de *structs* ocorre da mesma maneira que na convenção de chamada x64. Com a diferença que o primeiro argumento é, obviamente, passado na pilha.

Convenção de chamada stdcall

A convenção de chamada `__stdcall` é a utilizada para chamar funções da [WinAPI ↗](#).

Registradores

Assim como na `__cdecl` os registradores EAX, ECX e EDX são voláteis e os demais devem ser preservados pela função chamada.

Passagem de parâmetros

Todos os argumentos são passados na pilha na ordem inversa. A função **chamada** (*callee*) é a responsável por remover os argumentos da pilha. Exemplo:

```
push $3
push $2
push $1
call my_function
```


Retorno de valores

O retorno de valores funciona da mesma maneira que o retorno de valores da `__cdecl`.

Variáveis em C

Entendendo como variáveis em C são representadas em Assembly.

Como já vimos no capítulo [A base](#), variáveis nada mais são do que um espaço de memória que pode ser manipulado pelo programa. Em C existem diversas nuances em como variáveis são alocadas e mantidas pelo compilador e aqui vamos entender essas diferenças.

 Na linguagem C existem palavra-chaves que são chamadas de *storage-class specifiers*, onde elas determinam o *storage-class* de uma variável. Na prática isso determina como a variável deve ser armazenada no programa. No C11 existem os seguintes *storage-class specifiers*:

- `extern`
- `static`
- `_Thread_local`
- `auto` (esse é o padrão)
- `register`

Variáveis globais

As variáveis globais em C são alocadas na seção `.data` ou `.bss`, dependendo se ela foi inicializada ou não. Como no exemplo:

```
int data_var = 1;
int bss_var;
```

Se compilamos com `gcc main.c -S -o main.s -fno-asynchronous-unwind-tables` obtemos a seguinte saída:

```
main.s
```

```
.globl data_var
.data
.align 4
.type data_var, @object
.size data_var, 4
data_var:
.long 1
.comm bss_var,4,4
```

A variável `data_var` foi alocada na seção `.data` e teve seu símbolo exportado com a diretiva `.globl data_var`, que é equivalente a diretiva `global` do NASM.

Já a variável `bss_var` foi declarada com a diretiva `.comm symbol, size, align` que serve para declarar *common symbols* (símbolos comuns). Onde ela recebe como argumento o nome do símbolo seguido de seu tamanho (em bytes) e opcionalmente um valor de alinhamento. Em arquivos objetos ELF o argumento de alinhamento é um alinhamento em bytes, nesse exemplo a variável será alocada em um endereço alinhado por 4 bytes.

Já em arquivos objetos PE (do Windows) o alinhamento é um valor em potência de dois, logo para alinhar em 4 bytes deveríamos passar 2 como argumento ($2^2 = 4$). Se a gente passar 4 como argumento então seria um alinhamento de 2^4 que daria um alinhamento de 16 bytes.

Os símbolos declarados com a diretiva `.comm` que não foram inicializados em qualquer arquivo objeto são alocados na seção `.bss`. Logo nesse caso o uso da diretiva seria equivalente ao uso de `res*` do NASM, com a diferença que no NASM precisamos usar explicitamente na seção onde o espaço será alocado.

Variável static global

As variáveis globais com *storage-class* `static` funcionam da mesma maneira que as variáveis globais comum, com a diferença que seu símbolo não é exportado para que possa ser acessado em outro arquivo objeto. Como no exemplo:

```
static int data_var = 1;
static int bss_var;
```

Onde obtemos a saída:

```
.data
.align 4
.type    data_var, @object
.size    data_var, 4
data_var:
.long    1
.local   bss_var
.comm    bss_var,4,4
```

Repare que dessa vez o símbolo `data_var` não foi exportado com a diretiva `.globl`, enquanto o `bss_var` foi explicitamente declarado como local com a diretiva `.local` (já que a diretiva `.comm` exporta como global por padrão).

Variável extern

Variáveis `extern` em C são basicamente variáveis que são definidas em outro módulo. O GAS tem uma diretiva `.extern` que é equivalente a diretiva `extern` do NASM, isto é, indica que o símbolo será definido em outro arquivo objeto. Porém qualquer símbolo não declarado já é considerado externo por padrão pelo GAS. Experimente ver o código de saída do exemplo abaixo:

```
extern int extern_var;

int main(void)
{
    int x = extern_var;
    return 0;
}
```

Você vai reparar que na função `main` o símbolo `extern_var` foi lido porém ele não foi declarado.

Variáveis locais

Variáveis locais em C são comumente alocadas no *stack frame* da função, porém em alguns casos o compilador também pode reservar um registrador para armazenar o valor da variável.

Em C existe o *storage-class* `register` que serve como um "pedido" para o compilador alocar aquela variável de forma que o acesso a mesma seja o mais rápido possível, que geralmente é em um registrador (daí o nome da palavra-chave). Mas isso não garante que a variável será realmente alocada em um registrador. Na prática o único efeito colateral garantido é que você não poderá obter o endereço na memória daquela variável com o operador de endereço (`&`), e muitas vezes o compilador já vai alocar a variável em um registrador mesmo sem o uso da palavra-chave.

Variável static local

Variáveis `static` local são armazenadas da mesma maneira que as variáveis `static` global, a única coisa que muda é no ponto de vista do código-fonte em C onde a visibilidade da variável é limitada para o escopo onde ela foi declarada. Isso faz com o que o compilador gere um símbolo de nome único para a variável, como no exemplo abaixo:

test.c

```
int test(void)
{
    static int data_var = 5;
    static int bss_var;

    return data_var + bss_var;
}
```

test.s

```
.data
.align 4
.type    data_var.1913, @object
.size    data_var.1913, 4
data_var.1913:
.long    5
.local   bss_var.1914
.comm    bss_var.1914,4,4
```

Repare como `data_var.1913` não teve seu símbolo exportado e `bss_var.1914` foi declarado como local.

Variáveis `_Thread_local`

O *storage-class* `_Thread_local` foi adicionado no C11. Assim como o nome sugere ele serve para alocar variáveis em uma região de memória que é local para cada [thread](#) do processo. Vamos analisar o exemplo:

test.c

```
_Thread_local int global_thread_data = 5;
_Thread_local int global_thread_bss;

int test(void)
{
    _Thread_local static int local_thread_data = 5;
    _Thread_local static int local_thread_bss;

    return global_thread_data
        + global_thread_bss
        + local_thread_data
        + local_thread_bss;
}
```

test.s

```

.text
.globl global_thread_data
.section .tdata,"awT",@progbits
.align 4
.type global_thread_data, @object
.size global_thread_data, 4
global_thread_data:
.long 5
.globl global_thread_bss
.section .tbss,"awT",@nobits
.align 4
.type global_thread_bss, @object
.size global_thread_bss, 4
global_thread_bss:
.zero 4
.section .tdata
.align 4
.type local_thread_data.1915, @object
.size local_thread_data.1915, 4
local_thread_data.1915:
.long 5
.section .tbss
.align 4
.type local_thread_bss.1916, @object
.size local_thread_bss.1916, 4
local_thread_bss.1916:
.zero 4
.text
.globl test
.type test, @function
test:
endbr64
pushq %rbp
movq %rsp, %rbp
movl %fs:global_thread_data@tpoff, %edx
movl %fs:global_thread_bss@tpoff, %eax
addl %eax, %edx
movl %fs:local_thread_data.1915@tpoff, %eax
addl %eax, %edx
movl %fs:local_thread_bss.1916@tpoff, %eax
addl %edx, %eax
popq %rbp
ret

```

No Linux, em x86-64, a região de memória local para cada *thread* (*thread-local storage* - TLS) fica no segmento apontado pelo [registrador de segmento](#) FS, por isso os valores das variáveis estão sendo lidos desse segmento.

Repare que as seções são diferentes, `.tdata` (equivalente a `.data` só que *thread-local*) e `.tbss` (equivalente a `.bss`) são utilizadas para armazenar as variáveis.

O sufixo `@tpoff` (*thread pointer offset*) usado nos símbolos indica que o *offset* do símbolo deve ser calculado levando em consideração a TLS como endereço de origem. Por padrão o *offset* é calculado com o segmento de dados "normal" como origem.

Lidando com os tipos da linguagem C

Agora que já entendemos onde e como as variáveis são alocadas em C, só falta entender "o que" está sendo armazenado.

Arrays e strings

O tipo *array* em C é meramente uma sequência de variáveis do mesmo tipo na memória. Por exemplo podemos inicializar um `int arr[4]` na sintaxe do GAS da seguinte forma:

```
arr:
    .long 1, 2, 3, 4
```

Onde os valores `1`, `2`, `3` e `4` são despejados em sequência.

Em C não existe um tipo *string* porém por convenção as *strings* são uma *array* de `char`, onde o último `char` contém o valor zero (chamado de terminador nulo). Esse último caractere `'\0'` é usado para denotar o final da *string* e funções da libc que lidam com *strings* esperam por isso. Exemplos:

```
string1:
    .ascii "Hello World", 0
string2:
    .ascii "Hello World\0"
string3:
    .asciz "Hello World"
```

As três *strings* acima são equivalentes na sintaxe do GAS.

Sobre a passagem de *arrays* (incluindo obviamente *strings*) como argumento para uma função, isso é feito passando um ponteiro para o primeiro elemento da *array*.

Ponteiros

Ponteiros em C, na arquitetura x86/x86-64, são traduzidos meramente como o *offset* do objeto na memória. O segmento não é especificado como parte do valor do ponteiro.

Experimente ler o código de saída do seguinte programa:

```
#include <stdio.h>

_Thread_local int my_var = 111;

int main(void)
{
    int *test = &my_var;
    *test = 777;

    printf("%d, %d\n", my_var, *test);
}
```

A leitura do endereço de `my_var` vai ser compilada para algo como:

```
movq    %fs:0, %rax
addq    $my_var@tpoff, %rax
movq    %rax, -8(%rbp)

# Com otimização ligada o GCC usa LEA:

movq    %fs:0, %rax
leaq    my_var@tpoff(%rax), %rdi
```

Onde primeiro é obtido o endereço do início do segmento FS que depois é somado ao *offset* de `my_var`. Assim obtendo o endereço efetivo da variável na memória.

Estruturas

As estruturas em C são compiladas de forma que os valores dos campos da estrutura são dispostos em sequência na memória, seguindo a mesma ordem que foram declarados na estrutura. Existe a possibilidade do GCC adicionar alguns bytes extras no final da estrutura afim de manter o alinhamento dos dados, esses bytes extras são chamados de *padding*. Exemplo:

```
#include <stdio.h>

typedef struct
{
    int x;
    char y;
} my_test_t;

my_test_t test = {
    .x = 5,
    .y = 'A',
};

int main(void)
{
    printf("%d, %c | sizeof: %zu\n", test.x, test.y, sizeof test);
}
```

Isso produziria o seguinte código para a inicialização da variável `test`:

```
.globl test
.data
.align 8
.type test, @object
.size test, 8
test:
    .long 5
    .byte 65
    .zero 3
```

Repare a diretiva `.zero 3` que foi usada para despejar 3 bytes zero no final da estrutura, afim de alinhar a mesma em 4 bytes. No total a estrutura acaba tendo 8 bytes de tamanho: 4 bytes do `int`, 1 byte do `char` e 3 bytes de *padding*.

Unions

As *unions* são bem simples, são alocadas com o tamanho do maior tipo declarado para a *union*. Por exemplo:

```
typedef union
{
    int x;
    char y;
} my_test_t;
```

Essa *union* é alocada na memória da mesma forma que um `int`, que tem 4 bytes de tamanho.

Funções em C

Entendendo as funções em C do ponto de vista do Assembly.

A linguagem C tem algumas variações à respeito de funções e o objetivo deste tópico é explicar, do ponto de vista do baixo-nível, como elas funcionam.

Entendendo os protótipos

As funções na linguagem C têm protótipos que servem como uma "assinatura" indicando quais parâmetros a função recebe e qual tipo de valor ela retorna. Um exemplo:

```
int add(int x, int y);
```

Esse protótipo já nos dá todas as informações necessárias que saibamos como fazer a chamada da função e como obter seu valor de retorno, desde que nós conheçamos a [convenção de chamada](#) utilizada. Os parâmetros são considerados da esquerda para a direita, logo o parâmetro `x` é o primeiro e o parâmetro `y` é o segundo. Na convenção de chamada da SysV ABI esses argumentos estariam em EDI e ESI, respectivamente. E o retorno seria feito em EAX.

Existem alguns protótipos um pouco diferentes que vale explicar aqui para deixar claro seu entendimento. Como este:

```
void do_something(int a);
```

De acordo com a especificação do C11 uma expressão do tipo `void` é um tipo cujo o valor não existe e **deve** ser ignorado. Funções assim são compiladas retornando sem se preocupar em modificar o valor de RAX (ou qualquer outro registrador que poderia ser usado para retornar um valor) e portanto não se deve esperar que o valor nesse registrador tenha alguma informação útil.

```
int do_something(void);
```


Quando `void` é usado no lugar da lista de parâmetros ele tem o significado especial de indicar que aquela função não recebe parâmetro algum ao ser chamada.

```
int do_something();
```

Embora possa ser facilmente confundido com o caso acima, onde se usa `void` na lista de parâmetros, na verdade esse protótipo de função não diz que a função não recebe parâmetros. Na verdade esse é um protótipo que não especifica quais tipos ou quantos parâmetros a função recebe, logo o compilador aceita que a função seja chamada passando qualquer tipo e qualquer quantidade de parâmetros, inclusive sem parâmetro algum também. Veja o exemplo:

main.c

```
#include <stdio.h>

int do_something();

int main(void)
{
    printf("Resultado: %d\n", do_something(1, 2, 3, 4.5f, "teste"));
}
```

main.s

```
movq    .LC1(%rip), %rax
leaq    .LC0(%rip), %rcx
movq    %rax, %xmm0
movl    $3, %edx
movl    $2, %esi
movl    $1, %edi
movl    $1, %eax
call    do_something@PLT
```

Na convenção de chamada da SysV ABI os argumentos para esse tipo de função são passados da mesma maneira que uma chamada com o protótipo "normal". A única diferença é que a função recebe um argumento extra no registrador AL indicando quantos registradores de vetor foram utilizados para passar argumentos de ponto-

flutuante. Nesse exemplo apenas um argumento era um *float* e por isso há a instrução `movl $1, %eax` indicando esse número. Experimente usar mais argumentos *float* ou não passar nenhum para ver se o número passado em AL como argumento irá mudar de acordo.

```
int do_something(int x, ...);
```

Funções com [argumentos variáveis](#) também seguem a mesma regra de chamada do que foi mencionado acima.

Funções static

Funções *static* são visíveis apenas no mesmo módulo em que elas foram declaradas, ou seja, seu símbolo não é exportado. Exemplo:

```
static int add(int a, int b)
{
    return a + b;
}
```

Function specifiers

Existem dois especificadores de função no C11, onde eles são:

inline

O especificador `inline` é uma sugestão para que a chamada para a função seja a mais rápida possível. Isso tem o efeito colateral no GCC de inibir a geração de código para a função em Assembly. Ao invés disso as instruções da função são geradas no local onde ela foi chamada, e portanto o símbolo da função nunca é de fato declarado.

⚠ O GCC, mesmo para uma função *inline*, ainda vai gerar o código para a chamada da função caso as otimizações estejam desligadas e isso vai acabar produzindo um erro de referência

por parte do *linker*. Lembre-se de sempre ligar as otimizações de código quando estiver usando funções *inline*.

_Noreturn

Funções com o especificador `_Noreturn` nunca devem retornar para a função chamadora. Quando esse especificador é utilizado o compilador irá gerar código assumindo que a função nunca retorna. Como podemos ver no exemplo abaixo compilado com `-O2`:

main.c

```
#include <stdio.h>
#include <stdlib.h>

_Noreturn void goodbye(const char *msg)
{
    puts(msg);
    exit(EXIT_SUCCESS);
}

int main(void)
{
    goodbye("Sayonara onii-chan! ^-^");
}
```

main.s

```
.text
.p2align 4
.globl  goodbye
.type   goodbye, @function
goodbye:
    endbr64
    pushq  %rax
    popq   %rax
    subq   $8, %rsp
    call   puts@PLT
    xorl   %edi, %edi
    call   exit@PLT
    .size  goodbye, .-goodbye
    .section .rodata.str1.1,"aMS",@progbits,1
.LC0:
    .string "Sayonara onii-chan! ^-^"
    .section .text.startup,"ax",@progbits
    .p2align 4
    .globl  main
    .type   main, @function
main:
    endbr64
    pushq  %rax
    popq   %rax
    leaq   .LC0(%rip), %rdi
    subq   $8, %rsp
    call   goodbye
```

Funções aninhadas

Nested functions é uma extensão do GCC que permite declarar funções aninhadas. O símbolo de uma função aninhada é gerado de maneira semelhante ao símbolo de uma variável local com *storage-class* `static`. Exemplo:

```
int calc(int a, int b)
{
    int add(int a, int b)
    {
        return a + b;
    }

    return add(a, b) + add(3, 4);
}
```

Atributos de função

Os atributos de função é uma extensão do GCC que permite modificar algumas propriedades relacionadas à uma função. Se define atributos para uma função usando a palavra-chave `__attribute__` e entre dois parênteses uma lista de atributos separado por vírgula. Exemplo:

```
__attribute__((cdecl))
int add(int a, int b)
{
    return a + b;
}
```


Alguns atributos recebem parâmetros onde estes devem ser adicionados dentro de mais um par de parênteses, se assemelhando a sintaxe de uma chamada de função. Exemplo:

```
__attribute__((section (".another"), cdecl))
```

Abaixo alguns atributos que podem ser usados na arquitetura x86 e acho interessante citar aqui:

ms_abi, sysv_abi, cdecl, stdcall, fastcall, thiscall

Esses atributos fazem com que o compilador gere o código da função usando a convenção de chamada [ms_abi](#), [sysv_abi](#), [cdecl](#), [stdcall](#), [fastcall](#) ou [thiscall](#) respectivamente. Também é útil usá-los em protótipos de funções onde a função utiliza uma convenção de chamada diferente da padrão.

 Os atributos `cdecl`, `stdcall`, `fastcall` e `thiscall` são ignorados em 64-bit.

section ("name")

Por padrão o GCC irá adicionar o código das funções na seção `.text`, porém é possível usar o atributo `section` para que o compilador adicione o código da função em outra seção. Como no exemplo abaixo:

main.c

```
__attribute__((section(".another")))  
int add(int a, int b)  
{  
    return a + b;  
}
```

main.s

```
.section    .another,"ax",@progbits  
.p2align 4  
.globl    add  
.type     add, @function  
add:  
    endbr64  
    leal    (%rdi,%rsi), %eax  
    ret
```

naked

O atributo `naked` é usado para desativar a geração do [prólogo e epílogo](#) para a função. Isso é útil para se escrever funções usando [inline Assembly](#) dentro das mesmas.

target ("option1", "option2", ...)

Esse atributo serve para personalizar a geração de código do compilador para uma função específica, permitindo selecionar quais instruções serão utilizadas ao gerar o código. Também é possível adicionar o prefixo `no-` para desabilitar alguma tecnologia e impedir que o compilador gere código para ela. Por exemplo `__attribute__((target ("no-sse")))` desativaria o uso de instruções ou registradores [SSE](#) na função.

Alguns dos possíveis alvos para arquitetura x86 são:

Ativar as instruções	Desativar as instruções
3dnow	no-3dnow
3dnowa	no-3dnowa
abm	no-abm
adx	no-adx
aes	no-aes
avx	no-avx
avx2	no-avx2
avx512fmaps	no-avx512fmaps
avx512vnniw	no-avx512vnniw
avx512bitalg	no-avx512bitalg
avx512bw	no-avx512bw
avx512cd	no-avx512cd
avx512dq	no-avx512dq
avx512er	no-avx512er
avx512f	no-avx512f
avx512ifma	no-avx512ifma
avx512pf	no-avx512pf
avx512vbmi	no-avx512vbmi
avx512vbmi2	no-avx512vbmi2

avx512vl	no-avx512vl
avx512vnni	no-avx512vnni
avx512vpopcntdq	no-avx512vpopcntdq
mmx	no-mmx
sse	no-sse
sse2	no-sse2
sse3	no-sse3
sse4	no-sse4
sse4.1	no-sse4.1
sse4.2	no-sse4.2
sse4a	no-sse4a
ssse3	no-ssse3

PLT e GOT

Já vimos alguns exemplos de código chamando funções da libc, essas funções porém estão em uma biblioteca dinâmica e não dentro do executável. A resolução do endereço (*symbol binding*) das funções na biblioteca é feito em tempo de execução onde os endereços são salvos na seção GOT (*Global Offset Table*).

A seção PLT (*Procedure Linkage Table*) simplesmente armazena saltos para os endereços armazenados na GOT. Por isso o GCC gera chamadas para funções da libc assim:

```
call    puts@PLT
```

O sufixo `@PLT` indica que o endereço do símbolo está na seção PLT. Onde nessa seção há uma instrução `jmp` para o endereço que será resolvido em tempo de execução na GOT. Algo parecido com a ilustração abaixo:


```
# Esse código não funciona, é apenas uma ilustração.

.section .got
real_puts_address.got:
    .quad 0
real_printf_address.got:
    .quad 0

.section .plt
puts.plt:
    jmp *real_puts_address.got
printf.plt:
    jmp *real_printf_address.got

.data
message:
    .asciz "Hello World!"

.text
my_func:
    lea message(%rip), %rdi
    call puts.plt

ret
```

Na sintaxe do NASM o equivalente ao uso do sufixo com `@` do GAS é a palavra-chave `wrt` (*With Reference To*), conforme exemplo:

```
extern puts

section .data
    message: db "Hello World!", 0

section .text
global assembly
assembly:
    lea rdi, [rel message]
    call puts wrt ..plt

ret
```

Ambiente hosted

Entendendo a execução de código em C no ambiente hosted.

Na especificação da linguagem C é descrito dois ambientes de execução de código: Os ambientes *hosted* e *freestanding*. Neste tópico vamos entender alguns pontos em relação a como funciona a estrutura e a execução de um programa em C no ambiente *hosted*.

O ambiente *hosted* essencialmente é o ambiente de execução de um código em C que executa sobre um sistema operacional. Nesse ambiente é esperado que haja suporte para múltiplas *threads* e todos os recursos descritos na especificação da biblioteca padrão (libc). A inicialização do programa ocorre quando a função **main** é chamada e antes de inicializar o programa é esperado que todos os objetos com [storage-class](#) `static` estejam inicializados.

A função main

A função **main** pode ser escrita com um dos dois protótipos abaixo:

```
int main(void)
{
    // ...
}
```

```
int main(int argc, char *argv[])
{
    // ...
}
```


Ou qualquer outro protótipo que seja equivalente a um desses. Como por exemplo `char **argv` também seria válido por ter equivalência a `char *argv[]`. Também pode-se usar qualquer nome de parâmetro, `argc` e `argv` são apenas sugestões.

O primeiro parâmetro passado para a função **main** indica o número de argumentos e o segundo é uma *array* de ponteiros para `char` onde cada índice na *array* é um argumento

e `argv[argc]` é um ponteiro NULL.

Se o tipo de retorno da função **main** for `int` (ou equivalente), o valor de retorno da primeira chamada para **main** é equivalente a chamar a função **exit** passando esse valor como argumento.

C startup code

 Os detalhes de implementação descritos aqui são baseados no código-fonte da glibc e podem ser diferentes em outras implementações da libc. Consulte [as referências](#) para ver a lista de completa de arquivos fonte consultados.

O código na glibc responsável pela inicialização do programa é chamado de *C startup* (CSU). Ele se encarrega de obter os argumentos de linha de comando, inicializar o TLS, executar o código na seção `.init` dentre outras tarefas de inicialização do programa.

O arquivo `start.S` é o que declara o símbolo `_start`, ou seja, a função de *entry point* do programa. A última chamada nessa função é para outra função chamada `__libc_start_main` que recebe o endereço da função **main** como primeiro argumento. Depois de algumas inicializações essa função chama a **main**, obtém o valor retornado em EAX e passa como argumento para a função responsável por finalizar o programa no sistema operacional (`exit_group` no Linux e `ExitProcess` no Windows).

Todos esses códigos estão em arquivos objetos pré-compilados no seu sistema operacional. Eles são *linkados* por padrão quando você invoca o GCC mas não são *linkados* por padrão se você chamar o *linker* (`ld`) diretamente.

No meu Linux o arquivo objeto `Scrt1.o` ("crt" é sigla para "*C runtime*") é o que contém o *entry point* (código do `start.S`). Os arquivos `crti.o` e `crtb.o` contém o prólogo e o epílogo, respectivamente, para as seções `.init` e `.fini`.

No meu Linux esses arquivos estão na pasta `/usr/lib/x86_64-linux-gnu/` e sugiro que consulte o conteúdo dos mesmos com a ferramenta **objdump**, como por exemplo:

```
$ objdump -d /usr/lib/x86_64-linux-gnu/Scrt1.o
```

Fazendo seu próprio startup code

- ⓘ Apenas para fins de curiosidade e dar uma noção mais "palpável" de como isso ocorre, irei ensinar aqui como você pode desabilitar a *linkedição* do CSU e programar uma versão personalizada do mesmo no Linux. Não recomendo que isso seja feito em um programa de verdade tendo em vista que você perderá diversos recursos que o *C runtime* padrão da glibc provém.

Use o seguinte código de teste:

start.s

```
STDOUT_FILENO = 1
SYS_WRITE = 1
SYS_EXIT_GROUP = 231

.section .rodata
init_msg:
.string "* Initializing...\n"
MSG_LENGTH = . - init_msg

.text
.globl _start
_start:
    mov $STDOUT_FILENO, %rdi
    lea init_msg(%rip), %rsi
    mov $MSG_LENGTH, %rdx
    mov $SYS_WRITE, %rax
    syscall          # write(STDOUT_FILENO, init_msg, MSG_LENGTH)

    pop %rdi         # argc: RDI
    mov %rsp, %rsi   # argv: RSI
    call main

    mov %rax, %rdi
    mov $SYS_EXIT_GROUP, %rax
    syscall          # exit_group( main(argc, argv) )
```

main.c

```
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("argc = %d\n", argc);

    for (int i = 0; i < argc; i++)
    {
        printf("argv[%d] = '%s'\n", i, argv[i]);
    }

    // Esperamos que argv[argc] seja um ponteiro NULL
    printf("argv[argc] = %s\n", argv[argc]);
    return 0;
}
```

Compile com:

```
$ as start.s -o crt1.o
$ gcc main.c -o main.o -c
$ gcc *.o -o test -nostartfiles
```

A opção `-nostartfiles` desabilita a *linkedição* dos arquivos objeto de inicialização.

O que o nosso `start.s` está fazendo é simplesmente chamar a syscall `write` para escrever uma mensagem na tela, chama a função **main** passando `argc` e `argv` como argumentos e depois chama a syscall `exit_group` passando como argumento o retorno da função **main**.

i No Linux, logo quando o programa é iniciado no *entry point*, o valor contendo o número de argumentos de linha de comando (`argc`) está em `(%rsp)`. E logo em seguida (`RSP+8`) está o início da *array* de ponteiros para os argumentos de linha de comando, terminando com um ponteiro NULL.

Experimente rodar `objdump -d test` nesse executável "customizado" e depois compare compilando com o CSU comum. Verá que o programa comum contém diversas funções que foram *linkadas* nele.

Seções `.init` e `.fini`

As seções `.init` e `.fini` contém funções construída nos arquivos `crti.o` e `crtn.o`.

O propósito da função em `.init` é chamar todas as funções na *array* de ponteiros localizada em outra seção chamada `.init_array`. Essas funções são invocadas antes da chamada para a função **main**.

Já a função em `.fini` invoca as funções da *array* na seção `.fini_array` na finalização do programa (após **main** retornar ou na chamada de `exit()`).

No GCC você pode adicionar funções para serem invocadas na inicialização do programa com o atributo `constructor`, e para a finalização do programa com o atributo `destructor`. Experimente ver o código Assembly do exemplo abaixo:

```
#include <stdio.h>

__attribute__((constructor))
void constructor1(void)
{
    puts("* constructor 1");
}

__attribute__((constructor))
void constructor2(void)
{
    puts("* constructor 2");
}

__attribute__((destructor))
void destructor1(void)
{
    puts("* destructor 1");
}

__attribute__((destructor))
void destructor2(void)
{
    puts("* destructor 2");
}

int main(void)
{
    puts("* main");
}
```

Ao [ver o Assembly gerado](#) do programa acima irá notar que os endereços das funções são despejados nas seções `.init_array` e `.fini_array`, como em:

```
.section    .init_array,"aw"
.align 8
.quad     constructor1
```

Funções de saída

exit

Quando a função `exit()` é invocada (ou **main** retorna), funções registradas pela função `atexit()` são executadas. Onde as funções registradas devem seguir o protótipo:

```
void funcname(void);
```

As funções registradas por `atexit()` são invocadas na ordem inversa a que foram registradas.

quick_exit

Quando a função `quick_exit()` é invocada o programa é finalizado sem invocar as funções registradas por `atexit()` e sem executar quaisquer *handlers* de sinal.

As funções registradas por `at_quick_exit` são invocadas na ordem inversa em que foram registradas.

Exemplo:

```
#include <stdio.h>
#include <stdlib.h>


void func_atexit(void)
{
    puts("* exiting...");
}

void func_at_quick_exit(void)
{
    puts("* Quick exiting...");
}

int main(void)
{
    atexit(func_atexit);
    at_quick_exit(func_at_quick_exit);

    puts("* main");
    // quick_exit(EXIT_SUCCESS);
    return 0;
}
```


Experimente executar o programa acima e depois recompilar com a chamada para **quick_exit** na linha 20.

 A quantidade máxima de funções que podem ser registradas com **atexit** ou **at_quick_exit** depende da implementação. Mas a especificação do C11 garante que no mínimo 32 funções podem ser registradas por cada uma destas funções.

_Exit

A função `_Exit()` finaliza a execução do programa sem executar quaisquer funções registradas por **atexit** ou **at_quick_exit**. Também não executa nenhum *handler* de sinal.

Ambiente freestanding

Entendendo a execução de código em C no ambiente freestanding.

O ambiente de execução *freestanding* é normalmente usado quando o código C é compilado para executar fora de um sistema operacional. Nesse ambiente nenhum dos recursos providos do ambiente *hosted* são garantidos e sua existência ou não depende da implementação.

Os únicos recursos que são oferecidos pela libc são os declarados nos seguintes *header files*:

<float.h>, **<iso646.h>**, **<limits.h>**, **<stdalign.h>**, **<stdarg.h>**, **<stdbool.h>**, **<stddef.h>**, **<stdint.h>** e **<stdnoreturn.h>**.

Quaisquer outros recursos são dependentes de implementação.


No GCC para compilar um código visando o ambiente *freestanding* é possível usar a opção `-ffreestanding`. Também se pode usar a opção `-fhosted` para compilar para ambiente *hosted* mas esse já é o padrão.

Já a opção `-nostdlib` desabilita a *linkedição* da libc.

Inline Assembly no GCC

Aprendendo a usar o inline Assembly do compilador GCC.

Inline Assembly é uma extensão do compilador que permite inserir código Assembly diretamente no código de saída do compilador. Dessa forma é possível misturar C e Assembly sem a necessidade de usar um módulo separado só para o código em Assembly, além de permitir alguns recursos interessantes que não são possíveis sem o *inline Assembly*.

 O compilador Clang contém uma sintaxe de *inline Assembly* compatível com a do GCC, logo o conteúdo ensinado aqui também é válido para o Clang.

Inline Assembly básico

A sintaxe do uso básico é: `asm [qualificadores] (instruções-asm)`.

Onde qualificadores é uma (ou mais) das seguintes palavra-chaves:

- **volatile**: Isso desabilita as otimizações de código no *inline Assembly*, mas esse já é o padrão quando se usa o *inline ASM* básico.
- **inline**: Isso é uma "dica" para o compilador considerar que o tamanho do código Assembly é o menor possível. Serve meramente para o compilador decidir se vai ou não expandir uma [função como inline](#), e usando esse qualificador você sugere que o código é pequeno o suficiente para isso.

As instruções Assembly ficam dentro dos parênteses como uma *string* literal e são despejadas no código de saída sem qualquer alteração por parte do compilador.

Geralmente se usa `\n\t` para separar cada instrução pois isso vai ser refletido literalmente na saída de código. O `\n` é para iniciar uma nova linha e o `\t` (TAB) é para manter a indentação do código de maneira idêntica ao código gerado pelo compilador.

Exemplo:

```
#include <stdio.h>

int main(void)
{
    asm(
        "mov $5, %eax\n\t"
        "add $3, %eax\n\t"
    );

    return 0;
}
```

Isso produz a seguinte saída ao [visualizar o código de saída](#):

```
main:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    #APP
    # 5 "main.c" 1
        mov $5, %eax
        add $3, %eax

    # 0 "" 2
    #NO_APP
        movl   $0, %eax
        popq   %rbp
        ret
```

Entre as diretivas `#APP` e `#NO_APP` fica o código despejado do *inline* Assembly. A diretiva `# 5 "main.c" 1` é apenas um atalho para a diretiva `#line` onde ela serve para avisar para o assembler de qual linha (5) e arquivo ("main.c") veio aquele código. Assim se ocorrer algum erro, na mensagem de erro do assembler será exibido essas informações.

Repare que o *inline* Assembly apenas despeja literalmente o conteúdo da *string* literal. Logo você pode adicionar o que quiser aí incluindo diretivas, comentários ou até mesmo instruções inválidas que o compilador não irá reclamar.

Também é possível usar *inline* Assembly básico fora de uma função, como em:

```
#include <stdio.h>

int add(int, int);

int main(void)
{
    printf("%d\n", add(2, 3));
    return 0;
}

asm (
    "add:\n\t"
    "    lea (%edi, %esi), %eax\n\t"
    "    ret"
);
```

Porém não é possível fazer o mesmo com *inline* Assembly estendido.

Inline Assembly estendido

A versão estendida do *inline* Assembly funciona de maneira semelhante ao *inline* Assembly básico, porém com a diferença de que é possível acessar variáveis em C e fazer saltos para rótulos no código fonte em C.

A sintaxe da versão estendida segue o seguinte formato:

```
asm [qualificadores] (
    "instruções-asm"
    : operandos-de-saída
    : operandos-de-entrada
    : clobbers
    : rótulos-goto
)
```

Os qualificadores são os mesmos da versão básica porém com mais um chamado **goto**. O qualificador **goto** indica que o código Assembly pode efetuar um salto para um dos rótulos listados no último operando. Esse qualificador é necessário para se usar os rótulos no código ASM. Enquanto o qualificador **volatile** desabilita a otimização de código, que é habilitada por padrão no *inline* Assembly estendido.

Dentre esses operandos somente os de saída são "obrigatórios", os demais podem ser omitidos. E todos eles podem conter uma lista vazia exceto o de rótulos.

Existe um limite **máximo de 30 operandos** com a soma dos operandos de saída, entrada e rótulos.

operandos-de-saída

Cada operando de saída é separado por vírgula e contém a seguinte sintaxe:

```
[nome] "restrições" (variável)
```

Onde `nome` é um símbolo opcional que você pode criar para se referir ao operando no código Assembly. Também é possível se referir ao operando usando `%n`, onde `n` seria o índice do operando (contando a partir de zero). E usar `%[nome]` caso defina um nome.

Como o `%` é usado para se referir à operandos, no *inline* Assembly estendido se usa dois `%` para se referir à um registrador. Já que `%%` é um escape para escrever o próprio `%` na saída, da mesma forma que se faz na função `printf`.

[As restrições](#) é uma *string* literal contendo letras e símbolos indicando como esse operando deve ser armazenado (`r` para registrador e `m` para memória, por exemplo). No caso dos operandos de saída o primeiro caractere na *string* deve ser um `=` ou `+`. Onde o `=` indica que a variável terá seu valor modificado, enquanto `+` indica que terá seu valor modificado e lido.

Operandos de saída com `+` são contabilizados como dois, tendo em vista que o `+` é basicamente um atalho para repetir o mesmo operando também como uma entrada.

Essas informações são necessárias para que o compilador consiga otimizar o código corretamente. Por exemplo caso você indique que a variável será somente escrita com `=` mas leia o valor da variável no Assembly, o compilador pode assumir que o valor da variável nunca foi lido e portanto descartar a inicialização dela durante a otimização de código. Isso criaria um comportamento estranho no *inline* Assembly onde se obteria lixo como valor da variável.

Um exemplo deste **erro**:

```
#include <stdio.h>

int main(void)
{
    int x = 5;

    asm("addl $3, %0"
        : "=rm"(x));

    printf("%d\n", x);
    return 0;
}
```

A otimização de código pode remover a inicialização `x = 5` já que não informamos que o valor dessa variável é lido dentro no *inline* Assembly. O correto seria usar `+` nesse caso.


Um exemplo (dessa vez correto) usando um nome definido para o operando:

```
#include <stdio.h>

int main(void)
{
    int x;

    asm("movl $5, %[myvar]"
        : [myvar] "=rm"(x));

    printf("%d\n", x);
    return 0;
}
```

 Caso utilize um operando que você não tem **certeza** que será armazenado em um registrador, lembre-se de usar [o sufixo na instrução](#) para especificar o tamanho do operando. Para evitar erros é ideal que sempre use os sufixos.

operandos-de-entrada

Os operandos de entrada seguem a mesma sintaxe dos operandos de saída porém sem o `=` ou `+` nas restrições. Não se deve tentar modificar operandos de entrada (embora tecnicamente seja possível) para evitar erros, lembre-se que o compilador irá otimizar o código assumindo que aquele operando não será modificado.

Também é possível passar expressões literais como operando de entrada ao invés de somente nomes de variáveis. A expressão será avaliada e seu valor passado como operando sendo armazenado de acordo com as restrições.

clobbers

Clobbers (que eu não sei como traduzir) é basicamente uma lista, separada por vírgula, de efeitos colaterais do código Assembly. Nele você **deve** listar o que o seu código ASM modifica além dos operandos de saída. Cada valor de *clobber* é uma *string* literal contendo o nome de um registrador que é modificado pelo seu código. Também há dois nomes especiais de *clobbers*:

Clobber	Descrição
cc	Indica que o código ASM modifica as <i>flags</i> do processador (registrador EFLAGS).
memory	<p>Indica que o código ASM faz leitura ou escrita da/na memória em outro lugar que não seja um dos operandos de entrada ou saída. Por exemplo em uma memória apontada por um ponteiro de um operando.</p> <p>Esse <i>clobber</i> evita que o compilador assuma que os valores das variáveis na memória permanecem os mesmos após a execução do código ASM. E também garante que o compilador escreva o valor de todas as variáveis na memória antes de executar o <i>inline</i> ASM.</p>
rax	Indica que o registrador RAX será modificado.
rbx	Indica que o registrador RBX será modificado.
etc.	...

Qualquer nome de registrador é válido para ser usado como *clobber* exceto o *Stack Pointer* (RSP). É esperado que no final da execução do *inline* ASM o valor de RSP seja o mesmo de antes da execução do código. Se não for o código muito provavelmente irá ter problemas no restante da execução.

Quando você adiciona um registrador a lista de *clobbers* ele não será utilizado para armazenar operandos de entrada ou saída, assim garantindo que o registrador pode ser utilizado livremente no *inline* ASM sem causar qualquer erro. Isso também garante que o compilador não irá assumir que o valor do registrador permanece o mesmo após a execução do *inline* ASM.

Exemplo:

```
int add(int a, int b)
{
    int result;

    asm("movl %[a], %%eax\n\t"
        "addl %[b], %%eax\n\t"
        "movl %%eax, %[result]"
        : [result] "=rm"(result)
        : [a] "r"(a),
          [b] "r"(b)
        : "cc",
          "eax");

    return result;
}
```

rótulos-goto

Ao usar `asm goto` pode-se referir à um rótulo usando o prefixo `%l` seguido do índice do operando de rótulo. Onde a contagem inicia em zero e é contabilizado também os operandos de entrada e saída.

Exemplo:

```
#include <stdio.h>
#include <stdbool.h>

int do_anything(bool value)
{
    int result = 3;

    asm goto("test %[value], %[value]\n\t"
            "jz %l1"
            :
            : [value] "r"(value)
            : "cc"
            : my_label);

    result += 2;

my_label:
    return result;
}

int main(void)
{
    printf("%d, %d\n", do_anything(true), do_anything(false));
    return 0;
}
```

Mas felizmente também é possível usar o nome do rótulo no *inline* Assembly, bastando usar a notação `%l[nome]`. O exemplo acima poderia ter a instrução de salto reescrita para `jz %l[my_label]`.

Restrições

As restrições (*constraints*) são uma lista de caracteres que determinam onde um operando deve ser armazenado. É possível indicar múltiplas alternativas para o compilador simplesmente adicionando mais de uma letra indicando tipos de armazenamento diferentes.

Abaixo a lista de algumas restrições disponíveis no GCC.

Restrições comuns

Restrição	Descrição
<code>m</code>	Operando na memória.
<code>r</code>	Operando em um registrador de propósito geral .
<code>i</code>	Um valor inteiro imediato.
<code>F</code>	Um valor <i>floating-point</i> imediato.
<code>g</code>	Um operando na memória, registrador de propósito geral ou inteiro imediato. Mesmo efeito que usar <code>"r1m"</code> como restrição.
<code>p</code>	Um operando que é um endereço de memória válido.
<code>X</code>	Qualquer operando é permitido. Basicamente deixa a decisão nas mãos do compilador.

Restrições para família x86

Restrição	Descrição
<code>R</code>	Registradores legado. Qualquer um dos oito registradores de propósito geral disponíveis em IA-32.
<code>q</code>	Qualquer registrador que seja possível ler o byte menos significativo. Como RAX (AL) ou R8 (R8B) por exemplo.
<code>Q</code>	Qualquer registrador que seja possível ler o segundo byte menos significativo, como RAX (AH) por exemplo.
<code>a</code>	O registrador "A" (RAX, EAX, AX ou AL).
<code>b</code>	O registrador "B" (RBX, EBX, BX ou BL).
<code>c</code>	O registrador "C" (RCX, ECX, CX ou CL).
<code>d</code>	O registrador "D" (RDX, EDX, DX ou DL).

S	RSI, ESI, SI ou SIL.
D	RDI, EDI, DI ou DIL.
A	O conjunto AX:DX.
f	Qualquer registrador do x87 .
t	ST0
u	ST1
y	Qualquer registrador MMX.
x	Qualquer registrador SSE .
Yz	XMM0
I	Um inteiro constante entre 0 e 31, usado para <i>shift</i> com valores de 32-bit.
J	Um inteiro constante entre 0 e 63, usado para <i>shift</i> com valores de 64-bit.
K	Inteiro sinalizado de 8-bit.

Dicas

Rótulos locais no inline Assembly

Se você simplesmente declarar rótulos dentro do *inline* Assembly pode acabar se deparando com uma redeclaração de símbolo por não ter uma garantia de que ele seja único. Mas uma dica é usar o escape especial `%=` que expande para um número único para cada uso de `asm`, assim sendo possível dar um nome único para os rótulos.

Exemplo:

```
#include <stdio.h>
#include <stdbool.h>

int do_anything(bool value)
{
    int result = 3;

    asm("test %[value], %[value]\n\t"
        "jz .my_label%=\n\t"
        "addl $2, %[result]\n\t"
        ".my_label%=:"
        : [result] "+a"(result)
        : [value] "r"(value)
        : "cc");

    return result;
}

int main(void)
{
    printf("%d, %d\n", do_anything(true), do_anything(false));
    return 0;
}
```

Usando sintaxe Intel

Caso prefira usar sintaxe Intel é possível fazer isso meramente compilando o código com `-masm=intel`. Isso porque o *inline* Assembly simplesmente despeja as instruções no arquivo de saída, portanto o código irá usar a sintaxe que o *assembler* utilizar.

Outra dica é usar a diretiva `.intel_syntax noprefix` no início, e depois `.att_syntax` no final para religar a sintaxe AT&T para o restante do código. Exemplo:

```
int add(int a, int b)
{
    int result;

    asm(".intel_syntax noprefix\n\t"
        "lea %[result], [ %[a] + %[b] ]\n\t"
        ".att_syntax"
        : [result] "=a"(result)
        : [a] "r"(a),
          [b] "r"(b));

    return result;
}
```

Escolhendo o registrador/símbolo para uma variável

Ao usar o *storage-class* `register` é possível escolher em qual registrador a variável será armazenada usando a seguinte sintaxe:

```
register int x asm("r12") = 5;
```

Nesse exemplo a variável `x` **obrigatoriamente** seria alocada no registrador R12.

Também é possível escolher o nome do símbolo para variáveis locais com *storage-class* `static` ou para variáveis globais. Como em:

```
static int x asm("my_var") = 5;
```

A variável no código fonte é referida como `x` mas o símbolo gerado para a variável seria definido como `my_var`.


Instruções intrínsecas

Aprendendo sobre as instruções intrínsecas na arquitetura x86-64

As instruções intrínsecas é um recurso originalmente fornecido pelo compilador Intel C/C++ mas que também é implementado pelo GCC. Se tratam basicamente de tipos especiais e funções que são expandidas *inline* para alguma instrução do processador, ou seja, é basicamente uma alternativa mais prática e legível do que usar *inline* Assembly para tudo.

Usando instruções intrínsecas é possível obter o mesmo resultado de usar *inline* Assembly com a diferença de ter a sintaxe amigável de uma chamada de função.

Para usar instruções intrínsecas é necessário incluir o *header* `<immintrin.h>` onde ele declara as funções e os tipos.

 Para entender apropriadamente as operações e tipos indicados aqui, sugiro que já tenha lido o tópico sobre [SSE](#).

Tipos de dados

Os tipos de dados na tabela abaixo servem para indicar como os valores usados na instrução intrínseca serão armazenados.

Tipo	Descrição
<code>__m64</code>	Tipo usado para representar o conteúdo de um registrador MMX. Pode armazenar 8 valores de 8-bit, 4 valores de 16-bit, 2 valores de 32-bit ou 1 valor de 64-bit.
<code>__m128</code>	Representa o conteúdo de um registrador SSE . Pode armazenar 4 valores <i>floating-point</i> de 32-bit.
<code>__m128d</code>	Também um registrador SSE porém armazenando 2 <i>floating-point</i> de 64-bit.
<code>__m128i</code>	Registrador SSE que pode armazenar 16 valores inteiros de 8-bit, 8 valores inteiros de 16-bit, 4 valores inteiros de 32-bit ou 2 valores inteiros de 64-bit.

<code>__m256</code>	Representa o conteúdo de um registrador YMM usado pela tecnologia AVX. armazenar 8 valores <i>floating-point</i> de 32-bit.
<code>__m256d</code>	Registrador YMM que pode armazenar 4 <i>floating-point</i> de 64-bit.
<code>__m256i</code>	Registrador YMM que pode armazenar 32 valores inteiros de 8-bit, 16 valores inteiros de 16-bit, 8 valores inteiros de 32-bit ou 4 valores inteiros de 64-bit.
<code>__m512</code>	Representa o conteúdo de um registrador ZMM usado pela tecnologia AVX-512. armazenar 16 valores <i>floating-point</i> de 32-bit.
<code>__m512d</code>	Registrador ZMM que pode armazenar 8 valores <i>floating-point</i> de 64-bit.
	Registrador ZMM que pode armazenar 64 valores inteiros de 8-bit, 32 inteiros de 16-bit, 16 inteiros de 32-bit ou 8 inteiros de 64-bit.

Nomenclatura

A maioria das instruções intrínsecas (SIMD) seguem a seguinte convenção de notação:

```
_mm_<operação>_<sufixo>
```

Onde **<operação>** é a operação que será executada com os dados. O **<sufixo>** indica o tipo de dado na operação. A primeira ou as duas primeiras letras do sufixo indicam se o dado é *single-precision* (**s**), *double-precision* (**d**) ou *integer* (**i** ou **u**). Os demais caracteres do sufixo indicam o tipo de dado, como mostra a tabela abaixo:

Sufixo	Tipo
<code>s</code>	<i>single-precision floating-point</i> (float de 32-bit)
<code>d</code>	<i>double-precision floating-point</i> (double de 64-bit)
<code>i128</code>	Inteiro sinalizado de 128-bit.
<code>i64</code>	Inteiro sinalizado de 64-bit.
<code>u64</code>	Inteiro não-sinalizado de 64-bit.
<code>i32</code>	Inteiro sinalizado de 32-bit.
<code>u32</code>	Inteiro não-sinalizado de 32-bit.

i16	Inteiro sinalizado de 16-bit.
u16	Inteiro não-sinalizado de 16-bit.
i8	Inteiro sinalizado de 8-bit.
—	


Exemplo:

```
double array[] = { 1.0, 2.0 };
__m128d data = _mm_load_pd(array);
```

Instruções

Abaixo irei listar apenas algumas instruções intrínsecas, em sua maioria relacionadas à tecnologia SSE. Para ver a lista completa sugiro que consulte a referência oficial da Intel no link abaixo:

- <https://software.intel.com/sites/landingpage/IntrinsicsGuide/> ↗

 Algumas instruções intrínsecas não são compiladas para uma só instrução mas sim uma **sequência** de várias delas.

Operações load, store e extract

Tecnologia	Protótipo	Instrução
SSE2	<code>__m128d _mm_load_pd (double const* mem_addr)</code>	<code>movapd xmm, m128</code>
SSE	<code>__m128 _mm_load_ps (float const* mem_addr)</code>	<code>movaps xmm, m128</code>
SSE2	<code>__m128d _mm_load_sd (double const* mem_addr)</code>	<code>movsd xmm, m64</code>

SSE2	<code>__m128i _mm_load_si128 (__m128i const* mem_addr)</code>	<code>movdqa xmm, m128</code>
SSE	<code>__m128 _mm_load_ss (float const* mem_addr)</code>	<code>movss xmm, m32</code>
SSE	<code>__m128i _mm_loadu_si16 (void const* mem_addr)</code>	Sequência
SSE2	<code>__m128i _mm_loadu_si32 (void const* mem_addr)</code>	<code>movd xmm, m32</code>
SSE	<code>__m128i _mm_loadu_si64 (void const* mem_addr)</code>	<code>movq xmm, m64</code>
SSE2	<code>__m128i _mm_loadu_si128 (__m128i const* mem_addr)</code>	<code>movdqu xmm, m128</code>
SSE2	<code>void _mm_store_pd (double* mem_addr, __m128d a)</code>	<code>movapd m128, xmm</code>
SSE	<code>void _mm_store_ps (float* mem_addr, __m128 a)</code>	<code>movaps m128, xmm</code>
SSE2	<code>void _mm_store_sd (double* mem_addr, __m128d a)</code>	<code>movsd m64, xmm</code>
SSE2	<code>void _mm_store_si128 (__m128i* mem_addr, __m128i a)</code>	<code>movdqa m128, xmm</code>
SSE	<code>void _mm_store_ss (float* mem_addr, __m128 a)</code>	<code>movss m32, xmm</code>
SSE2	<code>int _mm_extract_epi16 (__m128i a, int imm8)</code>	<code>pextrw r32, xmm, imm8</code>
SSE4.1	<code>int _mm_extract_epi32 (__m128i a, const int imm8)</code>	<code>pextrd r32, xmm, imm8</code>
SSE4.1	<code>long long int _mm_extract_epi64 (__m128i a, const int imm8)</code>	<code>pextrq r64, xmm, imm8</code>
SSE4.1	<code>int _mm_extract_epi8 (__m128i a, const int imm8)</code>	<code>pextrb r32, xmm, imm8</code>
SSE	<code>int _mm_extract_pi16 (__m64 a, int imm8)</code>	<code>pextrw r32, mm, imm8</code>

SSSE1 1

int _mm_extract_ps (__m128 a, const

extractps r32 xmm im

As operações **load** carregam um valor da memória para um registrador, o conteúdo que deve estar na memória apontada pelo argumento tem que estar de acordo com o tipo da instrução identificado pelo sufixo.

Operações **store** leem um ou mais dados do registrador e escrevem os mesmos no endereço passado como primeiro argumento.

Já a operação **extract** obtém um valor de uma parte do registrador identificado pelo valor imediato passado como segundo argumento. Esse valor é o índice do campo do registrador contando da direita para a esquerda começando em zero.

Exemplos:

load_store.c

```
#include <stdio.h>
#include <immintrin.h>

int main(void)
{
    float number;
    float array[] = {1.0f, 2.0f, 3.0f, 4.0f};
    __m128 data = _mm_load_ps(array);

    _mm_store_ss(&number, data);
    printf("%f\n", number);
    return 0;
}
```

load_extract.c

```
#include <stdio.h>
#include <immintrin.h>

int main(void)
{
    int array[] = {111, 222, 333, 444};
    __m128i data = _mm_load_si128((__m128i *)array);

    int number = _mm_extract_epi32(data, 2);
    printf("%d\n", number);
    return 0;
}
```

Operações set

As instruções intrínsecas de **set** definem o valor de todos os campos do registrador ao mesmo tempo sem a necessidade de usar uma *array* para isso. Elas não são traduzidas para uma mas sim várias instruções em sequência, portanto pode haver uma penalidade de desempenho.

Tecnologia	Protótipo
SSE2	<code>__m128i _mm_set_epi16 (short e7, short e6, short e5, short e4, short e3, short e2, short e1, short e0)</code>
SSE2	<code>__m128i _mm_set_epi32 (int e3, int e2, int e1, int e0)</code>
SSE2	<code>__m128i _mm_set_epi64 (__m64 e1, __m64 e0)</code>
SSE2	<code>__m128i _mm_set_epi64x (long long int e1, long long int e0)</code>
SSE2	<code>__m128i _mm_set_epi8 (char e15, char e14, char e13, char e12, char e11, char e10, char e9, char e8, char e7, char e6, char e5, char e4, char e3, char e2, char e1, char e0)</code>
SSE2	<code>__m128d _mm_set_pd (double e1, double e0)</code>
SSE	<code>__m128 _mm_set_ps (float e3, float e2, float e1, float e0)</code>
SSE2	<code>__m128d _mm_set_sd (double a)</code>
SSE	<code>__m128 _mm_set_ss (float a)</code>

As operações **set** escalares (`_mm_set_sd` e `_mm_set_ss`) definem o valor da parte menos significativa do registrador e zeram os demais valores.

As duas operações abaixo definem todos os campos do registrador para o mesmo valor passado como argumento:

Tecnologia	Protótipo
SSE2	<code>__m128d _mm_set_pd1 (double a)</code>
SSE	<code>__m128 _mm_set_ps1 (float a)</code>

Exemplo:

```
#include <stdio.h>
#include <immintrin.h>

int main(void)
{
    __m128i data = _mm_set_epi32(444, 333, 222, 111);

    int number = _mm_extract_epi32(data, 2);
    printf("%d\n", number);
    return 0;
}
```

Operações matemáticas

Tecnologia	Protótipo	Instrução
SSE2	<code>__m128i _mm_add_epi16 (__m128i a, __m128i b)</code>	<code>paddw xmm, xmm</code>
SSE2	<code>__m128i _mm_add_epi32 (__m128i a, __m128i b)</code>	<code>paddq xmm, xmm</code>
SSE2	<code>__m128i _mm_add_epi64 (__m128i a, __m128i b)</code>	<code>paddq xmm, xmm</code>
SSE2	<code>__m128i _mm_add_epi8 (__m128i a, __m128i b)</code>	<code>paddb xmm, xmm</code>

SSE2	<code>__m128d _mm_add_pd (__m128d a, __m128d b)</code>	<code>addpd xmm, xmm</code>
SSE	<code>__m128 _mm_add_ps (__m128 a, __m128 b)</code>	<code>addps xmm, xmm</code>
SSE2	<code>__m128d _mm_add_sd (__m128d a, __m128d b)</code>	<code>addsd xmm, xmm</code>
SSE2	<code>__m64 _mm_add_si64 (__m64 a, __m64 b)</code>	<code>paddq mm, mm</code>
SSE	<code>__m128 _mm_add_ss (__m128 a, __m128 b)</code>	<code>addss xmm, xmm</code>
SSE2	<code>__m128d _mm_div_pd (__m128d a, __m128d b)</code>	<code>divpd xmm, xmm</code>
SSE	<code>__m128 _mm_div_ps (__m128 a, __m128 b)</code>	<code>divps xmm, xmm</code>
SSE2	<code>__m128d _mm_div_sd (__m128d a, __m128d b)</code>	<code>divsd xmm, xmm</code>
SSE	<code>__m128 _mm_div_ss (__m128 a, __m128 b)</code>	<code>divss xmm, xmm</code>
SSE2	<code>__m128d _mm_mul_pd (__m128d a, __m128d b)</code>	<code>mulpd xmm, xmm</code>
SSE	<code>__m128 _mm_mul_ps (__m128 a, __m128 b)</code>	<code>mulps xmm, xmm</code>
SSE2	<code>__m128d _mm_mul_sd (__m128d a, __m128d b)</code>	<code>mulsd xmm, xmm</code>
SSE	<code>__m128 _mm_mul_ss (__m128 a, __m128 b)</code>	<code>mulss xmm, xmm</code>
SSE2	<code>__m64 _mm_mul_su32 (__m64 a, __m64 b)</code>	<code>pmuludq mm, mm</code>
SSE2	<code>__m128d _mm_sqrt_pd (__m128d a)</code>	<code>sqrtpd xmm, xmm</code>
SSE	<code>__m128 _mm_sqrt_ps (__m128 a)</code>	<code>sqrtps xmm, xmm</code>
SSE2	<code>__m128d _mm_sqrt_sd (__m128d a, __m128d b)</code>	<code>sqrtsd xmm, xmm</code>
SSE	<code>__m128 _mm_sqrt_ss (__m128 a)</code>	<code>sqrtss xmm, xmm</code>

SSE2	<code>__m128i _mm_sub_epi16 (__m128i a, __m128i b)</code>	<code>psubw xmm, xmm</code>
SSE2	<code>__m128i _mm_sub_epi32 (__m128i a, __m128i b)</code>	<code>psubd xmm, xmm</code>
SSE2	<code>__m128i _mm_sub_epi64 (__m128i a, __m128i b)</code>	<code>psubq xmm, xmm</code>
SSE2	<code>__m128i _mm_sub_epi8 (__m128i a, __m128i b)</code>	<code>psubb xmm, xmm</code>
SSE2	<code>__m128d _mm_sub_pd (__m128d a, __m128d b)</code>	<code>subpd xmm, xmm</code>
SSE	<code>__m128 _mm_sub_ps (__m128 a, __m128 b)</code>	<code>subps xmm, xmm</code>
SSE2	<code>__m128d _mm_sub_sd (__m128d a, __m128d b)</code>	<code>subsd xmm, xmm</code>
SSE2	<code>__m64 _mm_sub_si64 (__m64 a, __m64 b)</code>	<code>psubq mm, mm</code>
SSE	<code>__m128 _mm_sub_ss (__m128 a, __m128 b)</code>	<code>subss xmm, xmm</code>
SSSE3	<code>__m128i _mm_abs_epi16 (__m128i a)</code>	<code>pabsw xmm, xmm</code>
SSSE3	<code>__m128i _mm_abs_epi32 (__m128i a)</code>	<code>pabsd xmm, xmm</code>
SSSE3	<code>__m128i _mm_abs_epi8 (__m128i a)</code>	<code>pabsb xmm, xmm</code>
SSSE3	<code>__m64 _mm_abs_pi16 (__m64 a)</code>	<code>pabsw mm, mm</code>
SSSE3	<code>__m64 _mm_abs_pi32 (__m64 a)</code>	<code>pabsd mm, mm</code>
SSSE3	<code>__m64 _mm_abs_pi8 (__m64 a)</code>	<code>pabsb mm, mm</code>
SSE4.1	<code>__m128d _mm_ceil_pd (__m128d a)</code>	<code>roundpd xmm, xmm, imm8</code>
SSE4.1	<code>__m128 _mm_ceil_ps (__m128 a)</code>	<code>roundps xmm, xmm, imm8</code>
SSE4.1	<code>__m128d _mm_ceil_sd (__m128d a, __m128d b)</code>	<code>roundsd xmm, xmm, imm8</code>

SSE4.1	<code>__m128 _mm_ceil_ss (__m128 a, __m128 b)</code>	<code>roundss xmm, xmm, imm8</code>
SSE4.1	<code>__m128d _mm_floor_pd (__m128d a)</code>	<code>roundpd xmm, xmm, imm8</code>
SSE4.1	<code>__m128 _mm_floor_ps (__m128 a)</code>	<code>roundps xmm, xmm, imm8</code>
SSE4.1	<code>__m128d _mm_floor_sd (__m128d a, __m128d b)</code>	<code>roundsd xmm, xmm, imm8</code>
SSE4.1	<code>__m128 _mm_floor_ss (__m128 a, __m128 b)</code>	<code>roundss xmm, xmm, imm8</code>
SSE2	<code>__m128i _mm_max_epi16 (__m128i a, __m128i b)</code>	<code>pmaxsw xmm, xmm</code>
SSE4.1	<code>__m128i _mm_max_epi32 (__m128i a, __m128i b)</code>	<code>pmaxsd xmm, xmm</code>
SSE4.1	<code>__m128i _mm_max_epi8 (__m128i a, __m128i b)</code>	<code>pmaxsb xmm, xmm</code>
SSE4.1	<code>__m128i _mm_max_epu16 (__m128i a, __m128i b)</code>	<code>pmaxuw xmm, xmm</code>
SSE4.1	<code>__m128i _mm_max_epu32 (__m128i a, __m128i b)</code>	<code>pmaxud xmm, xmm</code>
SSE2	<code>__m128i _mm_max_epu8 (__m128i a, __m128i b)</code>	<code>pmaxub xmm, xmm</code>
SSE2	<code>__m128d _mm_max_pd (__m128d a, __m128d b)</code>	<code>maxpd xmm, xmm</code>
SSE	<code>__m64 _mm_max_pi16 (__m64 a, __m64 b)</code>	<code>pmaxsw mm, mm</code>
SSE	<code>__m128 _mm_max_ps (__m128 a, __m128 b)</code>	<code>maxps xmm, xmm</code>
SSE	<code>__m64 _mm_max_pu8 (__m64 a, __m64 b)</code>	<code>pmaxub mm, mm</code>
SSE2	<code>__m128d _mm_max_sd (__m128d a, __m128d b)</code>	<code>maxsd xmm, xmm</code>
SSE	<code>__m128 _mm_max_ss (__m128 a, __m128 b)</code>	<code>maxss xmm, xmm</code>

SSE2	<code>__m128i _mm_min_epi16 (__m128i a, __m128i b)</code>	<code>pminsw xmm, xmm</code>
SSE4.1	<code>__m128i _mm_min_epi32 (__m128i a, __m128i b)</code>	<code>pminsd xmm, xmm</code>
SSE4.1	<code>__m128i _mm_min_epi8 (__m128i a, __m128i b)</code>	<code>pminsb xmm, xmm</code>
SSE4.1	<code>__m128i _mm_min_epu16 (__m128i a, __m128i b)</code>	<code>pminuw xmm, xmm</code>
SSE4.1	<code>__m128i _mm_min_epu32 (__m128i a, __m128i b)</code>	<code>pminud xmm, xmm</code>
SSE2	<code>__m128i _mm_min_epu8 (__m128i a, __m128i b)</code>	<code>pminub xmm, xmm</code>
SSE2	<code>__m128d _mm_min_pd (__m128d a, __m128d b)</code>	<code>minpd xmm, xmm</code>
SSE	<code>__m64 _mm_min_pi16 (__m64 a, __m64 b)</code>	<code>pminsw mm, mm</code>
SSE	<code>__m128 _mm_min_ps (__m128 a, __m128 b)</code>	<code>minps xmm, xmm</code>
SSE	<code>__m64 _mm_min_pu8 (__m64 a, __m64 b)</code>	<code>pminub mm, mm</code>
SSE2	<code>__m128d _mm_min_sd (__m128d a, __m128d b)</code>	<code>minsd xmm, xmm</code>
SSE	<code>__m128 _mm_min_ss (__m128 a, __m128 b)</code>	<code>minss xmm, xmm</code>
SSE2	<code>__m128i _mm_avg_epu16 (__m128i a, __m128i b)</code>	<code>pavgw xmm, xmm</code>
SSE2	<code>__m128i _mm_avg_epu8 (__m128i a, __m128i b)</code>	<code>pavgb xmm, xmm</code>

Exemplos:

add_int.c

```
#include <stdio.h>
#include <immintrin.h>

int main(void)
{
    int array[4];
    __m128i data1 = _mm_set_epi32(444, 333, 222, 111);
    __m128i data2 = _mm_set_epi32(111, 222, 333, 444);

    __m128i result = _mm_add_epi32(data1, data2);
    _mm_store_si128((__m128i *)array, result);
    printf("%d, %d, %d, %d\n", array[0], array[1], array[2], array[3]);
    return 0;
}
```

sqrt_double.c

```
#include <stdio.h>
#include <immintrin.h>

int main(void)
{
    double array[2];
    __m128d data = _mm_set_pd(81.0, 625.0);

    __m128d result = _mm_sqrt_pd(data);
    _mm_store_pd(array, result);
    printf("%f, %f\n", array[0], array[1]);
    return 0;
}
```

Operações de randomização

As instruções intrínsecas abaixo leem um valor aleatório gerado por hardware:

Tecnologia	Protótipo	Instrução
RDRAND	<code>int _rdrand16_step (unsigned short* val)</code>	<code>rdrand r16</code>

RDRAND	<code>int _rdrand32_step (unsigned int* val)</code>	<code>rdrand r32</code>
RDRAND	<code>int _rdrand64_step (unsigned __int64*</code> <code>val)</code>	<code>rdrand r64</code>

A instrução `rdrand` escreve o valor aleatório obtido no ponteiro passado como argumento. Ela deve ser usada em um *loop* pois não há garantia de que ela irá obter de fato um valor. Se obter o valor a função retorna `1`, caso contrário retorna `0`.

Exemplo:

```
#include <stdio.h>
#include <immintrin.h>

int main(void)
{
    int value;

    while (!_rdrand32_step(&value))
        ;

    printf("Valor: %d\n", value);
    return 0;
}
```

Você **deve** compilar passando a *flag* `-mrdrnd` para o GCC para indicar que o processador suporta a tecnologia. Caso contrário você obterá um erro como este:

error: inlining failed in call to always_inline ‘_rdrand32_step’: **target specific option mismatch**

As instruções intrínsecas abaixo são utilizadas da mesma maneira que **rdrand** porém o valor aleatório não é gerado por hardware.

Tecnologia	Protótipo	Instrução
RDSEED	<code>int _rdseed16_step (unsigned short * val)</code>	<code>rdseed</code> <code>r16</code>
RDSEED	<code>int _rdseed32_step (unsigned int * val)</code>	<code>rdseed</code> <code>r32</code>

RDSEED	<code>int _rdseed64_step (unsigned __int64 * val)</code>	<code>rdseed</code> <code>r64</code>
--------	--	---

É necessário compilar com a *flag* `-mrndseed` para poder usar essas instruções intrínsecas.
