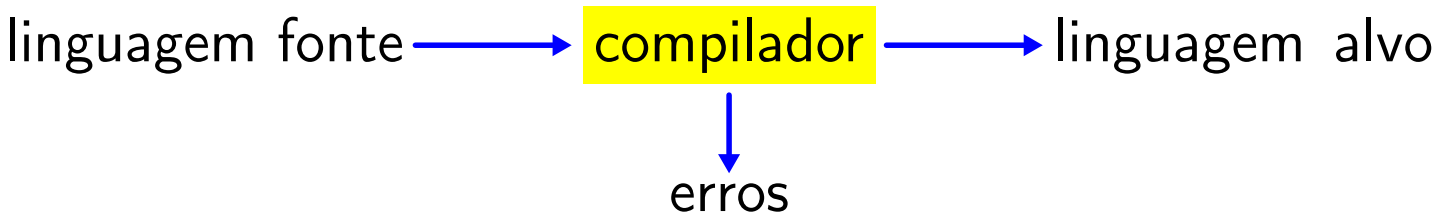
Universidade da Beira Interior

Desenho de Linguagens de Programação e de Compiladores

Simão Melo de Sousa

Aula 1 - Assembly x86-64

Resumidamente, um compilador é um programa que transforma um « programa » de uma linguagem **fonte** para uma linguagem **alvo**, assinalando eventuais erros.



Quando se fala de compilação, pensa-se tipicamente na tradução de uma linguagem de **alto nível**(C, Java, OCaml, ...) para a linguagem **máquina** de um processador (Intel Pentium, PowerPC, ...)

```
% gcc -o sum sum.c
```


```
int main(int argc, char **argv) {
  int i, s = 0;
  for (i = 0; i <= 100; i++) s += i*i;
  printf("0*0+...+100*100 = %d\n", s);
}
```

⟶

```
00100111101111011111111111100000
10101111101111110000000000010100
10101111101001000000000000100000
10101111101001010000000000100100
10101111101000000000000000011000
10101111101000000000000000011100
10001111101011100000000000011100
...
```

nesta aula, vamos de facto interessar-nos à compilação para **assembly**, mas este é só um aspecto da compilação.

Un conjunto relevante de técnicas envolvidas na compilação não estão directamente envolvidas na produção de código **assembly**.

Certas linguagens são, aliás,

- interpretadas (Basic, COBOL, Ruby, Python, etc.)
- compiladas para uma linguagem intermédia que é depois interpretada (Java, OCaml, Scala, etc.)
- compiladas para uma outra linguagem de alto nível
- compilada *on-the-fly*

diferença entre um compilador e um interpretador

um **compilador** traduz um programa P num programa Q tal que para toda a entrada x, a saída $Q(x)$ seja a mesma que a saída de $P(x)$

$$\forall P \; \exists Q \; \forall x ...$$

um **interpretador** é um programa que, dado um programa P e uma entrada x, calcula a saída s de $P(x)$

$$\forall P \; \forall x \; \exists s ...$$

diferença entre um compilador e um interpretador

por outras palavras,

o compilador faz um trabalho complexo **uma só vez**, para produzir um código que funciona para qualquer entrada

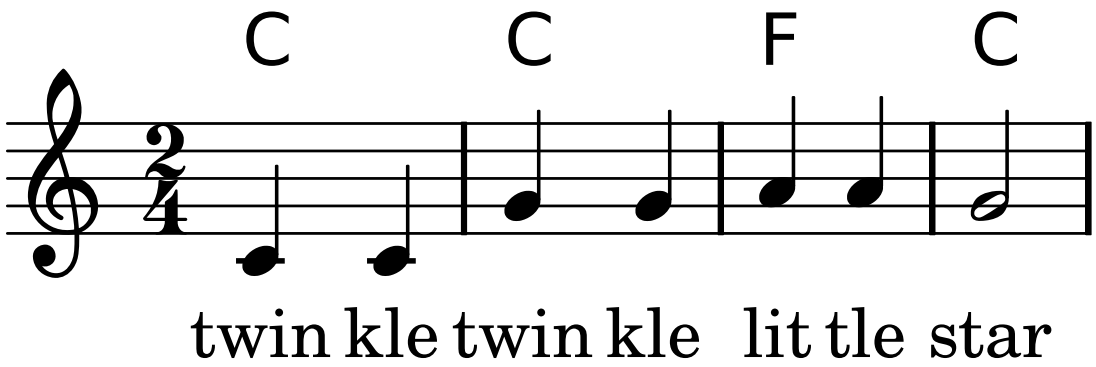
O interpretador realiza uma tarefa mais simples, mas fá-lo novamente sobre qualquer entrada

outra diferença: o código compilado é geralmente bem mais eficiente que o código interpretado.

exemplo de compilação e de interpretação

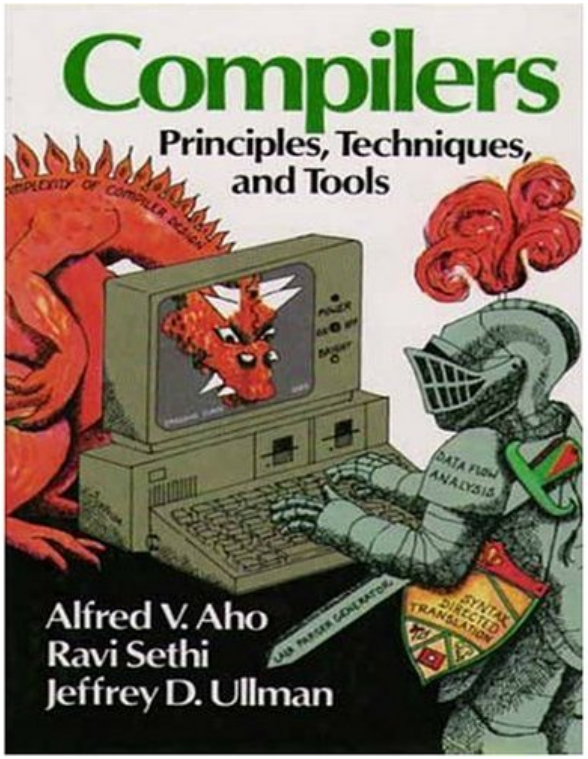
fonte → `lilypond` → ficheiro PostScript → `gs` → imagem

```
«
    \chords { c2 c f2 c }
    \new Staff \relative c' { \time 2/4 c4 c g'4 g a4 a g2 }
    \new Lyrics \lyricmode { twin4 kle twin kle lit tle star2 }
»
```



Quais os critérios para avaliar a qualidade de um compilador ?

- a sua correção
- a eficácia do código que produz
- a sua própria eficiência



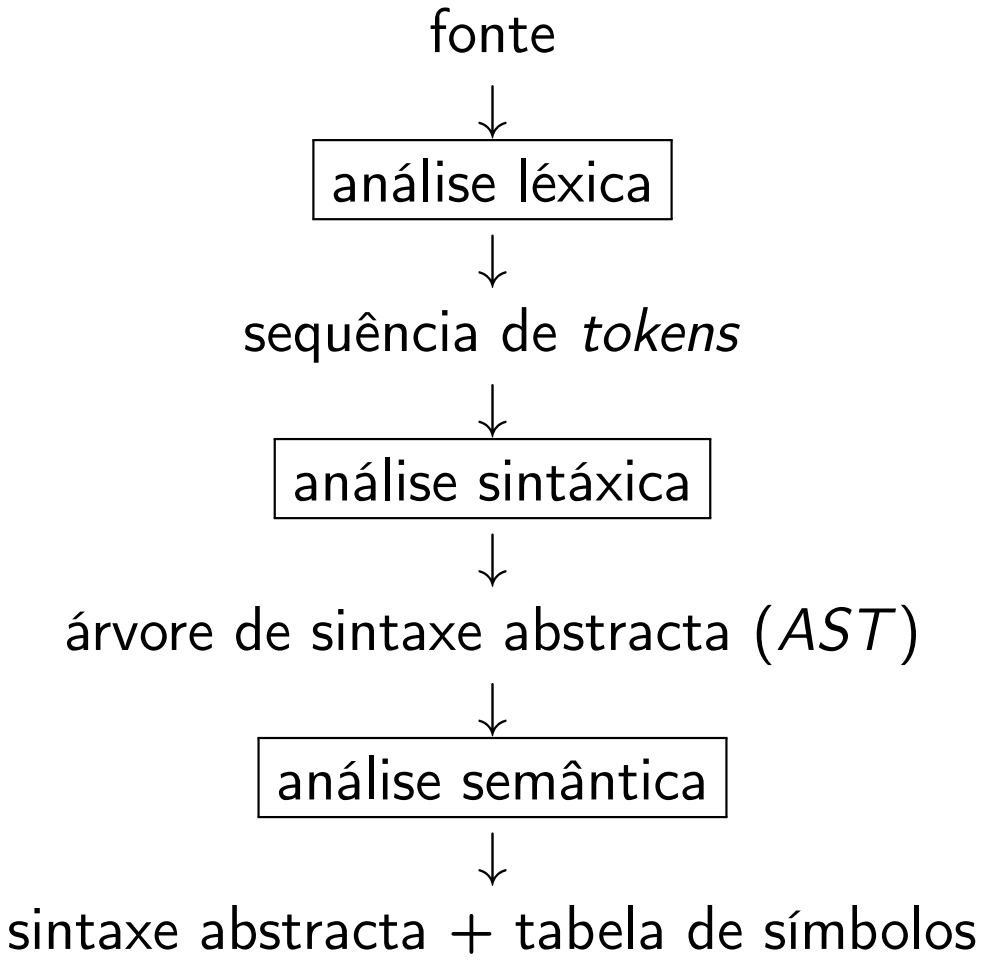
"Optimizing compilers are so difficult to get right that we dare say that no optimizing compiler is completely error-free! Thus, the most important objective in writing a compiler is that it is correct."

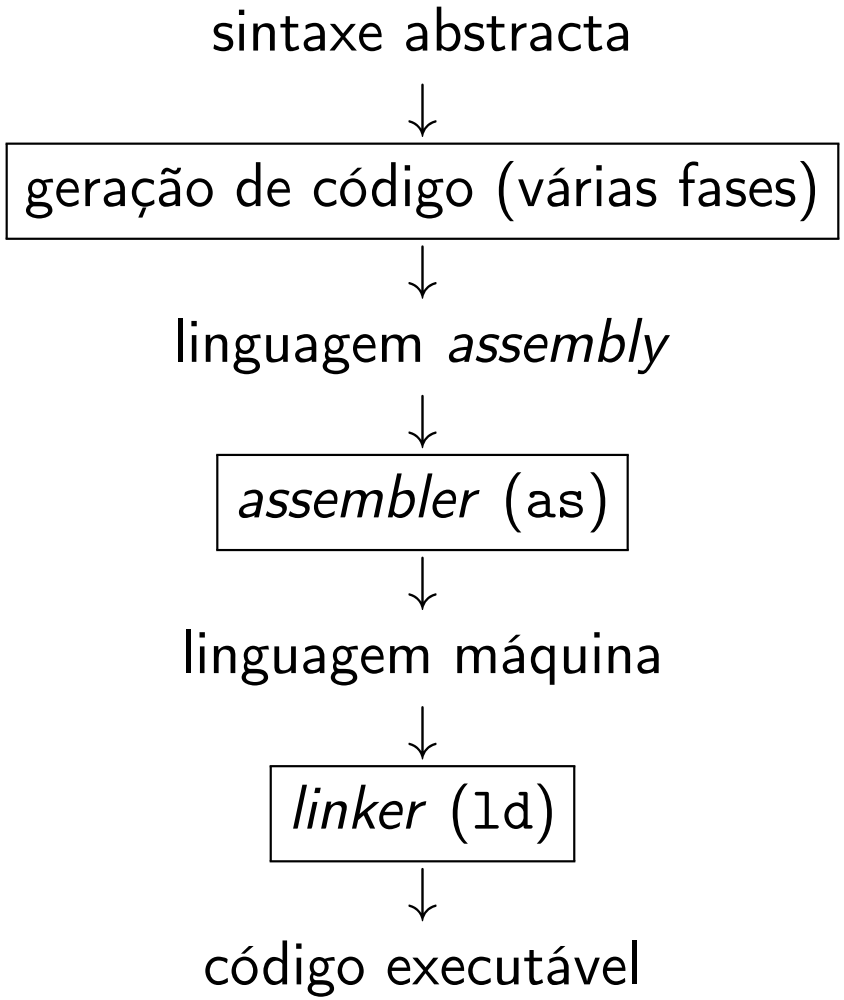
(Dragon Book, 2006)

classicamente, um compilador é composto

- por uma fase de **análise**
  - reconhece um programa por traduzir e o que ele significa
  - assinala os erros e pode então fracassar
    (erros de sintaxe, de porte, de tipagem, etc.)

- seguido de uma fase de **síntese**
  - produção de código na linguagem alvo
  - envolve um conjunto consequente de linguagens intermédias
  - não falha


fonte
análise léxica
sequência de *tokens*
análise sintáxica
árvore de sintaxe abstracta (*AST*)
análise semântica
sintaxe abstracta + tabela de símbolos


sintaxe abstracta
geração de código (várias fases)
linguagem *assembly*
assembler (as)
linguagem máquina
linker (ld)
código executável

hoje:

assembly

um inteiro é representado por n bits,
por convenção, numerados da direita para a esquerda

b_{n-1}	b_{n-2}	...	b_1	b_0

tipicamente, n é 8, 16, 32, ou 64

os bits b_{n-1}, b_{n-2}, etc. são designados de bits de **maior peso**
os bits b_0, b_1, etc. são designados de bits de **menor peso**

$$\begin{aligned} \text{bits} &= b_{n-1} b_{n-2} \ldots b_1 b_0 \\ \text{valor} &= \sum_{i=0}^{n-1} b_i 2^i \end{aligned}$$

bits	valor
$000\ldots000$	0
$000\ldots001$	1
$000\ldots010$	2
$\vdots$	$\vdots$
$111\ldots110$	$2^n - 2$
$111\ldots111$	$2^n - 1$

exemplo : $00101010_2 = 42$

o bit de maior peso b_{n-1} é o **bit do sinal**

$$\begin{aligned} \text{bits} &= b_{n-1}\, b_{n-2} \ldots b_1\, b_0 \\ \text{valor} &= -b_{n-1} 2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i \end{aligned}$$

bits	valor
$100\ldots000$	-2^{n-1}
$100\ldots001$	$-2^{n-1}+1$
$\vdots$	$\vdots$
$111\ldots110$	-2
$111\ldots111$	-1
$000\ldots000$	0
$000\ldots001$	1
$000\ldots010$	2
$\vdots$	$\vdots$
$011\ldots110$	$2^{n-1}-2$
$011\ldots111$	$2^{n-1}-1$

exemplo :

$$\begin{aligned} 11010110_2 &= -128 + 86 \\ &= -42 \end{aligned}$$

consoante o contexto, interpreta-se ou não o bit b_{n-1} como um bit de sinal

exemplo :

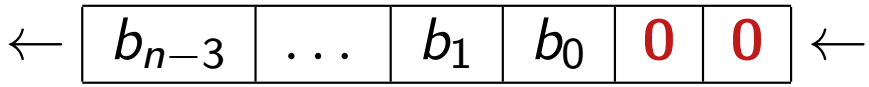
- $11010110_2 = -42$ (8 bits com sinal)
- $11010110_2 = 214$ (8 bits sem sinal)

a máquina fornece operações

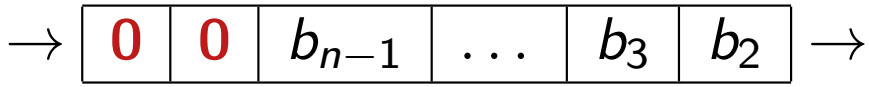
- operações lógicas, ou ainda designadas de *bitwise* (AND, OR, XOR, NOT)
- de *shift*
- aritméticos (adição, subtracção, multiplicação, etc.)

operação		exemplo
negação	x	00101001
	NOT x	11010110
E	x	00101001
	y	01101100
	x AND y	00101000
OU	x	00101001
	y	01101100
	x OR y	01101101
OU exclusivo	x	00101001
	y	01101100
	x XOR y	01000101

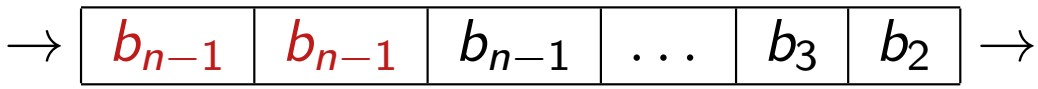
- *shift* lógico para a esquerda (insere zeros nos bits de menor peso)



- *shift* lógico para a direita (insere zeros nos bits de maior peso)



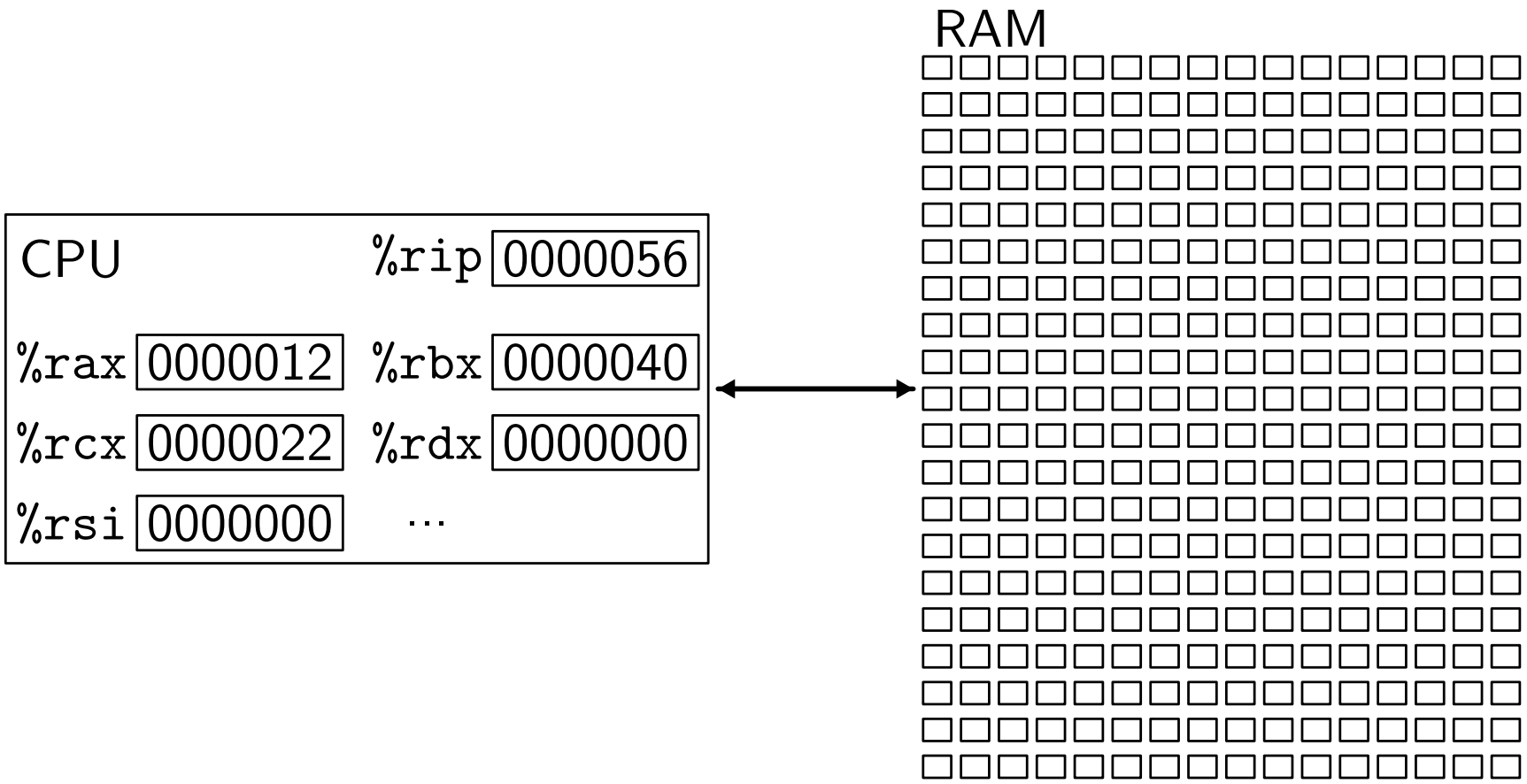
- *shift* aritmético para a direita (duplica o bit do sinal)



muito resumidamente, um computador é composto por

- uma unidade de calculo (CPU), que contém
  - um numero reduzido de registos inteiros ou flutuantes
  - capacidade de cálculo

- uma memória (RAM)
  - composta de um número alargado de bytes (8 bits)
    por exemplo, 1 Gb = 2^{30} bytes = 2^{33} bits, ou seja $2^{2^{33}}$ estados possíveis
  - contém dados e instruções



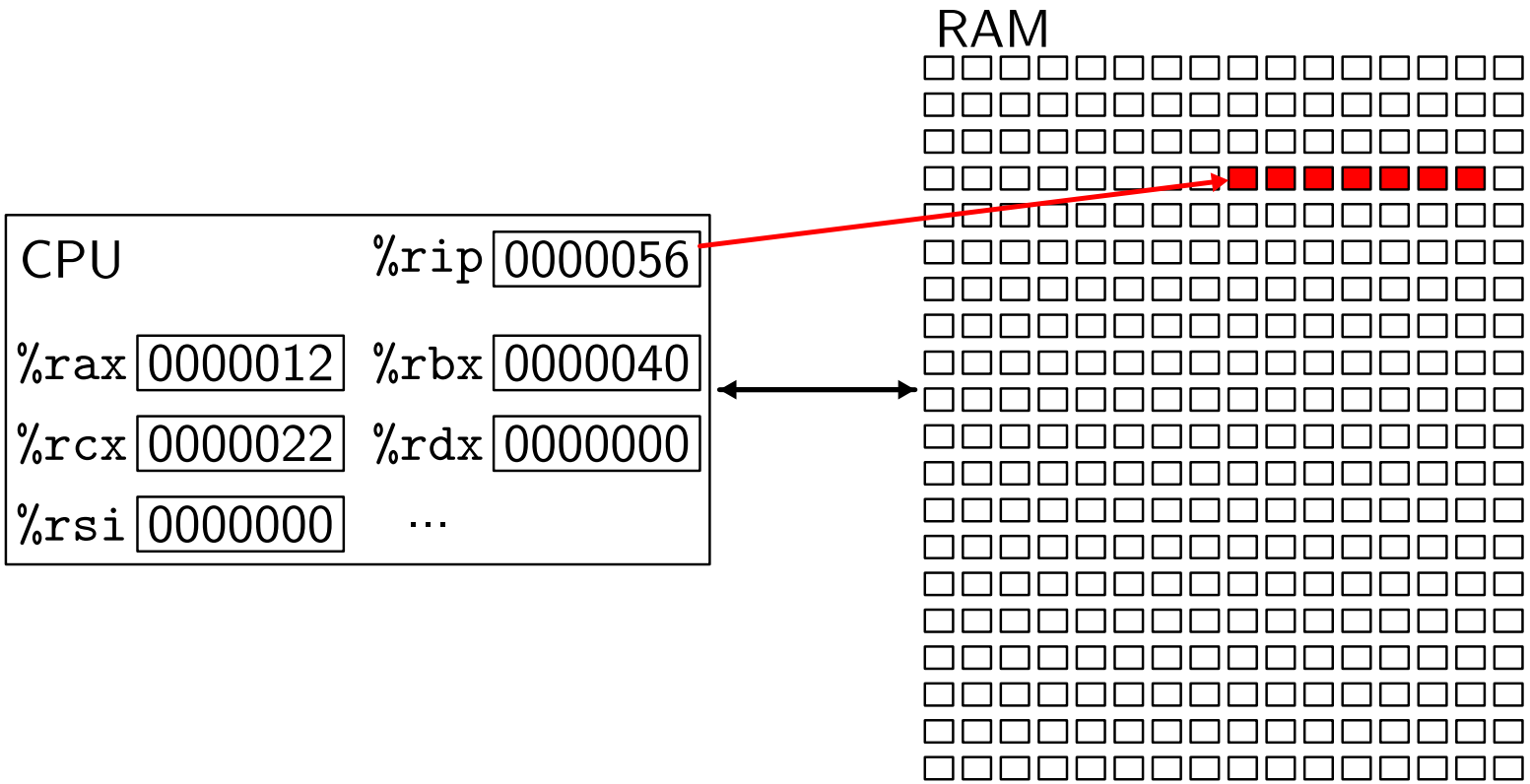
o acesso memória é custoso (com um débito de mil milhões de instruções por segundo, a **luz** só percorre 30 cms entre a execução de 2 instruções !)

a realidade é bem mais complexa

- vários (co)processadores, alguns deles dedicados aos cálculos de flutuantes
- uma ou várias memórias cache
- uma virtualização de memória (MMU)
- etc.

de forma esquemática, a execução de um programa se desenvolve da seguinte forma

- um registo (`%rip`) contém o endereço da instrução por executar
- lê-se um ou mais bytes neste endereço (*fetch*)
- interpreta-se estes bits como uma instrução (*decode*)
- executa-se a instrução (*execute*)
- modifica-se o registo `%rip` para passar à instrução seguinte (tipicamente a instrução que se segue, excepto no caso de um salto)



instrução : 48 c7 c0 2a 00 00 00

descodificação : movq %rax 42

i.e. colocar 42 no registo %rax

a realidade é na verdade bem mais complexa

- pipelines
  - várias instruções executadas em paralelo
- *branch prediction*
  - para optimizar o pipeline, tenta-se prever os saltos condicionais

que arquitectura escolhemos nesta aula ?

duas grandes famílias de microprocessadores

- CISC (*Complex Instruction Set*)
  - muitas instruções
  - muitos modos de endereçamento
  - muitas instruções de leitura e escrita em memória
  - poucos registos
  - exemplos : VAX, PDP-11, Motorola 68xxx, Intel x86
- RISC (*Reduced Instruction Set*)
  - menos instruções, regulares
  - número pequeno de instruções de leitura e escrita em memória
  - muitos registos, uniformes
  - exemplos : Alpha, Sparc, MIPS, ARM

escolhemos o **X86-64** nesta UC (nas aulas práticas e no trabalho)

a arquitectura X86-64

x86 uma família de arquitecturas compatíveis

- 1974 Intel 8080 (8 bits)
- 1978 Intel 8086 (16 bits)
- 1985 Intel 80386 (32 bits)

x86-64 uma extensão 64-bits

- 2000 introduzida pela AMD
- 2004 adotada pela Intel

- 64 bits
  - operações aritméticas, lógicas e de transferência sobre 64 bits
- 16 registos
  - `%rax`, `%rbx`, `%rcx`, `%rdx`, `%rbp`, `%rsp`, `%rsi`, `%rdi`, `%r8`, `%r9`, `%r10`, `%r11`, `%r12`, `%r13`, `%r14`, `%r15`
- endereçamento da memória sobre pelo menos 48 bits ($\geq$ 256 Tb)
- numeroso modos de endereçamento

Não programamos em linguagem máquina, mas sim em *assembly*

o *assembly* fornece um determinado conjunto de facilidades :

- *labels* simbólicos
- alocação de dados globais

a linguagem *assembly* é transformada em linguagem máquina por um programa designado de **assembler** (é um compilador)

utilizaremos aqui Linux (e afins) e as ferramentas da GNU

em particular o *assembly* GNU com a sintaxe **sintaxe AT&T**

noutros sistemas operativos, as ferramentas podem apresentar diferença

em particular, o *assembly* pode utilizar a **sintaxe Intel**, diferente.

```
        .text                       # instruções seguem
        .globl  main                # torna main visível para ld
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movq    $message, %rdi      # argumentos de puts
        call    puts
        movq    $0, %rax            # código de retorno 0
        popq    %rbp
        ret

        .data                       # dados seguem
message:
        .string "Hello, world!"     # terminados por 0
```

montagem

```
> as hello.s -o hello.o
```

linkagem (gcc invoca ld)

```
> gcc hello.o -o hello
```

execução

```
> ./hello
Hello, world!
```

podemos desmontar o *assembly* (**disassembler**) com a ferramenta `objdump`

```
> objdump -d hello.o
0000000000000000 <main>:
   0: 55                          push   %rbp
   1: 48 89 e5                    mov    %rsp,%rbp
   4: 48 c7 c7 00 00 00 00        mov    $0x0,%rdi
   b: e8 00 00 00 00              call   10 <main+0x10>
  10: 48 c7 c0 00 00 00 00        mov    $0x0,%rax
  17: 5d                          pop    %rbp
  18: c3                          ret
```

notemos

- que os endereços da string e de `puts` ainda não são conhecidos
- que o programa começa no endereço 0

podemos desmontar o executável

```
> objdump -d hello
0000000000401126 <main>:
  401126: 55                          push   %rbp
  401127: 48 89 e5                    mov    %rsp,%rbp
  40112a: 48 c7 c7 30 40 40 00        mov    $0x404030,%rdi
  401131: e8 fa fe ff ff              call   401030 <puts@plt>
  401136: 48 c7 c0 00 00 00 00        mov    $0x0,%rax
  40113d: 5d                          pop    %rbp
  40113e: c3                          ret
```

observamos agora

- um endereço efectivo para a string ($0x404030)
- um endereço efectivo para a função `puts` ($0x401030)
- que o programa começa no endereço $0x401126

observa-se que os bytes do inteiro `0x601020` estão arquivados em memória na ordem 20, 10, 60, 00

diz-se da máquina considerada que ela é **little-endian**

outras arquitecturas optam pelo contrário e são assim designadas de **big-endian** ou ainda **bi-endian**

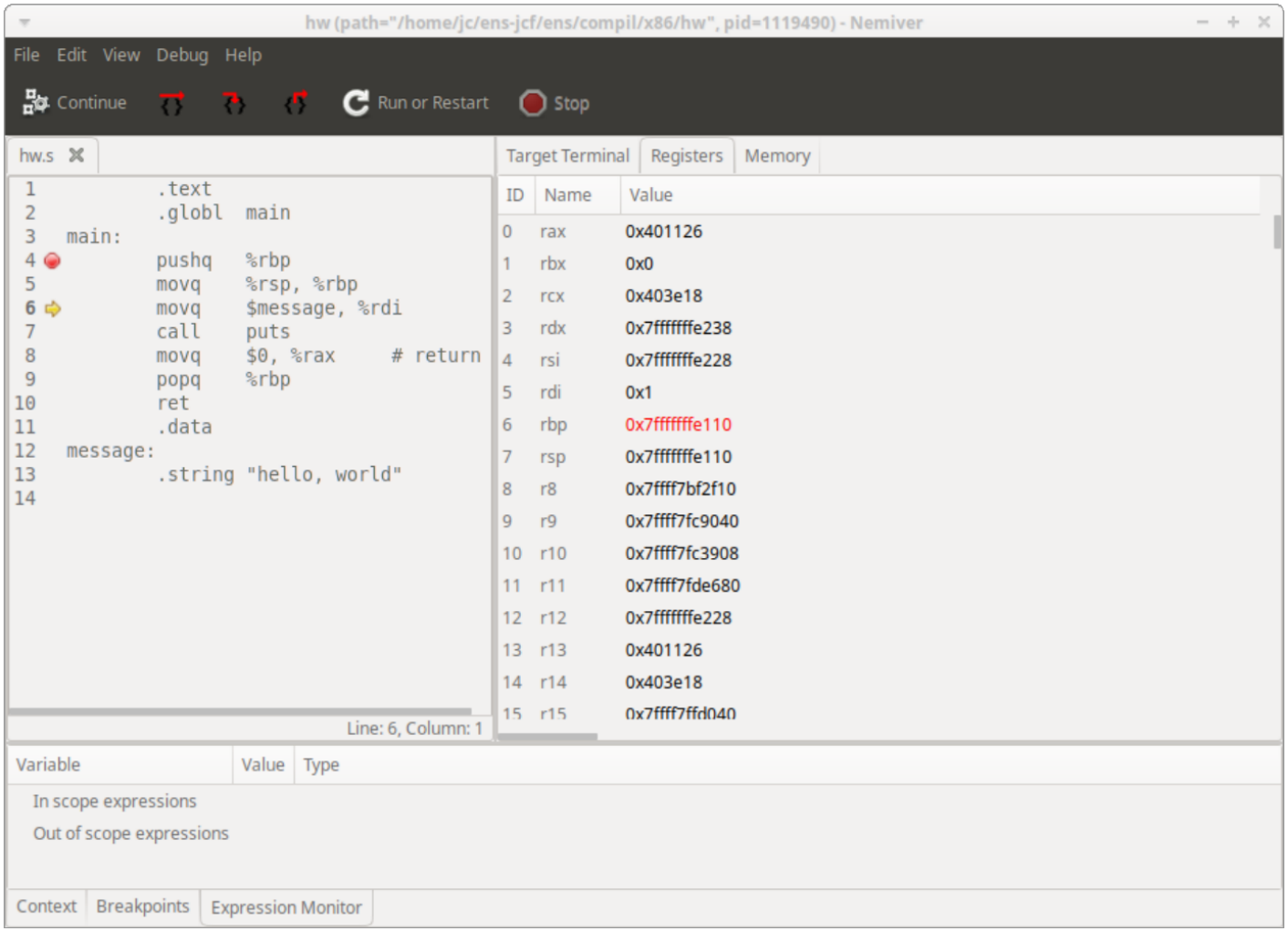
(referência a : *As viagens de Gulliver* de Jonathan Swift)

uma execução passo a passo é possível com gdb (*the GNU debugger*)

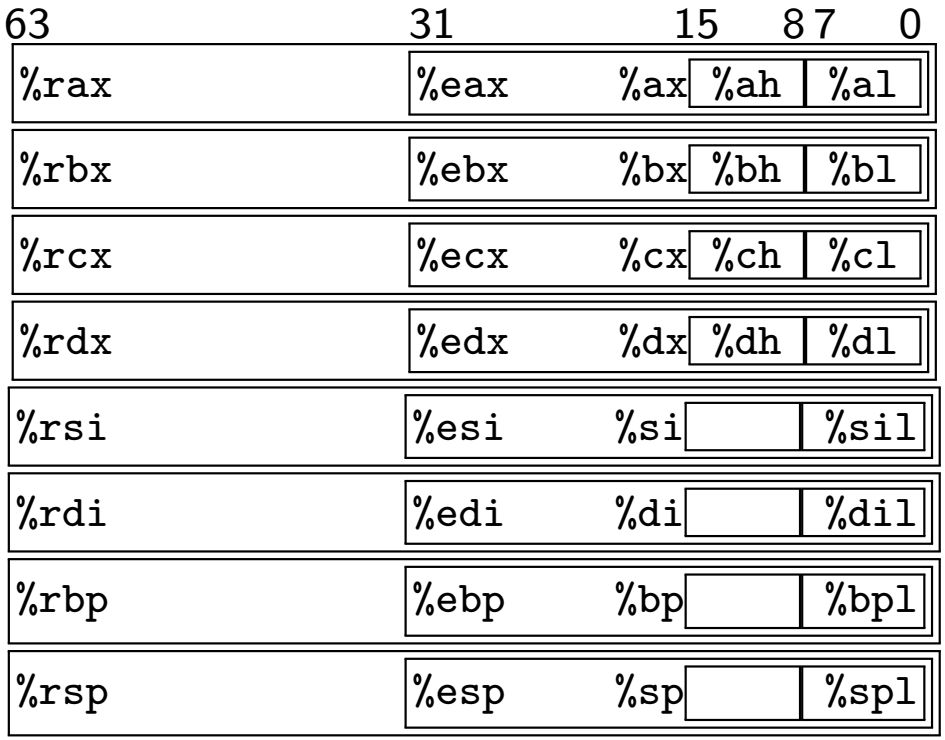
```
> gcc -g -no-pie hello.s -o hello
> gdb hello
GNU gdb (GDB) 7.1-ubuntu
...
(gdb) break main
Breakpoint 1 at 0x401126: file hello.s, line 4.
(gdb) run
Starting program: .../hello

Breakpoint 1, main () at hello.s:4
4               pushq   %rbp
(gdb) step
5               movq    %rsp, %rbp
(gdb) info registers
...
```

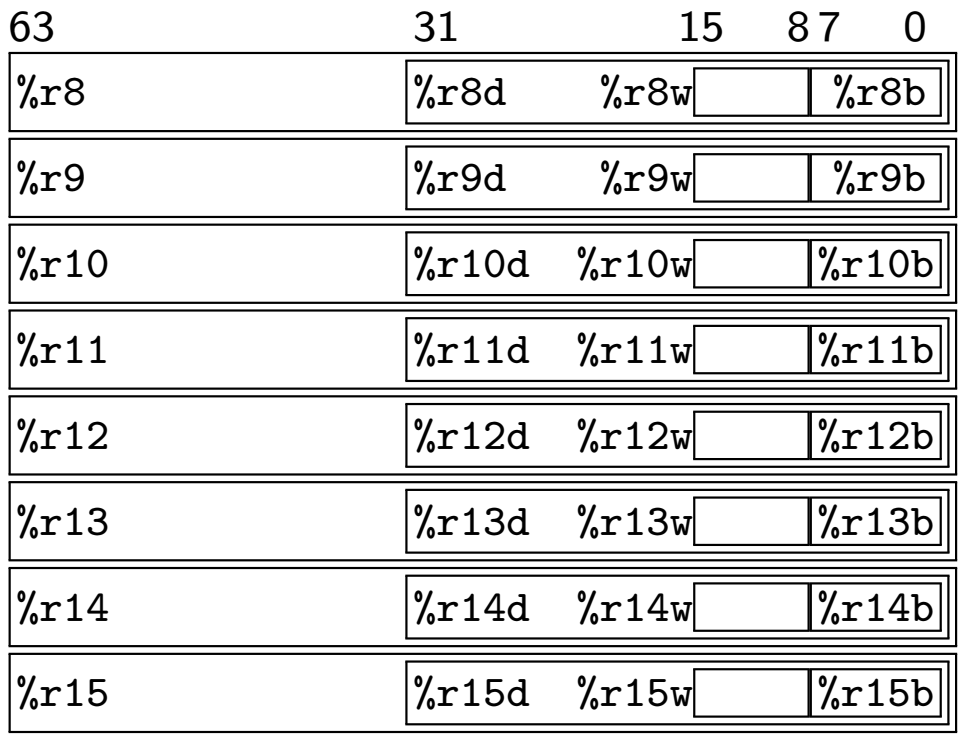
podemos igualmente utilizar o *Nemiver* (por instalar na sua máquina....)

```
> nemiver hello
```



Conjunto de instruções


63
31
15
8 7
0
%rax
%eax
%ax
%ah
%al
%rbx
%ebx
%bx
%bh
%bl
%rcx
%ecx
%cx
%ch
%cl
%rdx
%edx
%dx
%dh
%dl
%rsi
%esi
%si
%sil
%rdi
%edi
%di
%dil
%rbp
%ebp
%bp
%bpl
%rsp
%esp
%sp
%spl


63
31
15
8 7
0
%r8
%r8d
%r8w
%r8b
%r9
%r9d
%r9w
%r9b
%r10
%r10d
%r10w
%r10b
%r11
%r11d
%r11w
%r11b
%r12
%r12d
%r12w
%r12b
%r13
%r13d
%r13w
%r13b
%r14
%r14d
%r14w
%r14b
%r15
%r15d
%r15w
%r15b

- carregamento de uma constante num registo

```
        movq    $0x2a, %rax    # rax <- 42
        movq    $-12,  %rdi
```

- carregamento de um endereço de um rótulo (*label*) num registo

```
        movq    $label, %rdi
```

- cópia de um registo noutro

```
        movq    %rax, %rbx    # rbx <- rax
```

- soma de dois registos

```
        addq    %rax, %rbx          # rbx <- rbx + rax
```

(de forma semelhante, `subq`, `imulq`)

- soma de um registo e de uma constante

```
        addq    $2, %rcx            # rcx <- rcx + 2
```

- caso particular

```
        incq    %rbx                # rbx <- rbx+1
```

(de forma semelhante, `decq`)

- negação

```
        negq    %rbx                # rbx <- -rbx
```

- negação lógica

```
        notq    %rax                # rax <- not(rax)
```

- e, ou, ou exclusivo

```
        orq     %rbx, %rcx          # rcx <- or(rcx, rbx)
        andq    $0xff, %rcx         # apaga os bits >= 8
        xorq    %rax, %rax          # reset (para zero)
```

- shift para a esquerda (inserção de zeros)

```
        salq    $3, %rax    # 3 vezes
        salq    %cl, %rbx   # cl vezes
```

- shift aritmético para a direita (cópia do bit de sinal)

```
        sarq    $2, %rcx
```

- shift lógico para a direita (inserção de zeros)

```
        shrq    $4, %rdx
```

- rotação

```
        rolq    $2, %rdi
        rorq    $3, %rsi
```

o sufixo `q` nas instruções prévias
significa uma operação envolvendo 64 bits (*quad words*)

outros sufixos são aceites

sufixo	#bytes	
`b`	1	(*byte*)
`w`	2	(*word*)
`l`	4	(*long*)
`q`	8	(*quad*)

```
        movb    $42, %ah
```

quando os tamanhos dos dois operandos diferem,
pode ser necessário detalhar o modo de **extensão**

```
        movzbq  %al, %rdi       # com extensão de zeros
        movswl  %al, %rdi       # com extensão de sinal
```

um operando colocado entre parêntesis designa um **endereçamento indireto**
i.e. um local memória neste endereço

```
        movq    $42, (%rax)       # mem[rax] <- 42
        incq    (%rbx)            # mem[rbx] <- mem[rbx] + 1
```

nota : o endereço pode ser um rótulo

```
        movq    %rbx, (x)
```

a maior parte das operações não aceitam mais do que um endereçamento indireto

```
addq    (%rax), (%rbx)
```

```
Error: too many memory references for ‘add’
```

deve-se então passar por registos

```
movq    (%rax), %rcx
addq    %rcx, (%rbx)
```

de forma mais geral, um operando

$$A(B, I, S)$$

designa o endereço $A + B + I \times S$ onde

- A é uma constante sobre 32 bits com sinal
- I vale 0 se for omitido
- $S \in \{1, 2, 4, 8\}$ (vale 1 se omitido)

```
movq    -8(%rax,%rdi,4), %rbx  # rbx <- mem[-8+rax+4*rdi]
```

a operação `lea` calcula o endereço efectivo que corresponde ao operando

$$A(B, I, S)$$

```
leaq    -8(%rax,%rdi,4), %rbx    # rbx <- -8+rax+4*rdi
```

nota : podemos utilizar este recurso até para fazer aritmética

```
leaq    (%rax,%rax,2), %rbx    # rbx <- 3*rax
```

a maior parte das operações manipulam bandeiras (**flag**) do processador conforme o resultado que calculam

flag	significado
ZF	o resultado é 0
CF	um *resto* para além do bit de peso maior
SF	o resultado é negativo
OF	*overflow*

(exceção notável: **lea**)

três instruções permitam testar as *flags*

- salto condicional (`j`*cc*)

```
    jne     label
```

- posiciona-se para 1 (verdade) ou 0 (falso) (`set`*cc*)

```
    setge   %bl
```

- `mov` condicional (`cmov`*cc*)

```
    cmovl   %rax, %rbx
```

sufixo		significado
`e`	`z`	$= 0$
`ne`	`nz`	$\neq 0$
`s`		< 0
`ns`		≥ 0
`g`		$>$ com sinal
`ge`		$\geq$ com sinal
`l`		$<$ com sinal
`le`		$\leq$ com sinal
`a`		$>$ sem sinal
`ae`		$\geq$ sem sinal
`b`		$<$ sem sinal
`be`		$\leq$ sem sinal

podemos posicionar as *flags* sem guardar o resultado,
no caso da substração e de o E lógico

```
        cmpq    %rbx, %rax      # flag para rax - rbx
```

(cuidado com o sentido !)

```
        testq   %rbx, %rax      # flag para and(rax, rbx)
```

- para um *label*

```
        jmp     label
```

- para um endereço calculado

```
        jmp     *%rax
```

muito, muito mais instruções disponíveis
ver - *Enumerating x86-64 — It's Not as Easy as Counting*

por exemplo as instruções de tipo **SIMD** (Single Instruction, Multiple Data) ou ainda **SSE** (streaming SIMD extensions) que permitam uma gestão dos registos **vetoriais** que podem conter inteiros ou flutuantes

é o de traduzir um programa de alto nível para este conjunto de instruções

em particular, deve-se

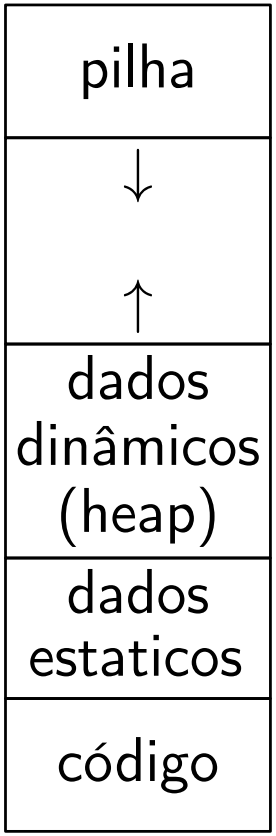
- traduzir as estruturas de controlo (testes, ciclos, excepções, etc.)
- traduzir as chamadas de funções
- traduzir as estruturas de dados complexas (*arrays*, *records*, objectos, fechos, etc.)
- alocar memória dinâmica

um facto por considerar : as chamadas a funções podem estar arbitrariamente aninhadas

$\Rightarrow$ os registos disponíveis podem não serem suficientes para todas as variáveis

$\Rightarrow$ é preciso alocar memória para tal

as funções operam no modo *last-in first-out*, isto é na forma de uma **pilha**



a **pilha** está arquivada na parte de cima e cresce no sentido dos endereços decrescentes; `%rsp` aponta para o topo da pilha

os dados dinâmicos (que sobrevivem às chamadas de funções) são alocadas na **heap** (eventualmente por um *GC*), na parte de baixo da zona de dados, imediatamente acima da zona dos dados estáticos

desta forma, está tudo organizado e ninguém pisa ninguém

nota : cada programa tem a ilusão de ter toda a memória para ele só; é o SO que cria e gere esta ilusão

- empilha-se com `pushq`

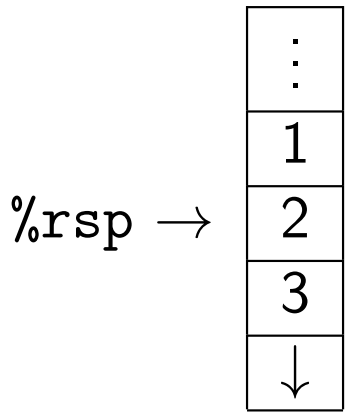
```
pushq   $42
pushq   %rax
```

- desempilha-se com `popq`

```
popq    %rdi
popq    (%rbx)
```

exemplo :

```
pushq   $1
pushq   $2
pushq   $3
popq    %rax
```



quando uma função f (**caller**)
pretende chamar uma função g (**callee**),
não podemos somente fazer

```
    jmp    g
```

porque será necessário voltar para o código de f quando g terá terminado

a situação piora quando consideramos chamadas aninhadas

a solução consiste em servir-se da pilha

duas instruções para isso

a instrução

```
        call    g
```

1. empilha o endereço da instrução situada logo a seguir à chamada (após o `call`)
2. transfere o controlo para o endereço de g

e a instrução

```
        ret
```

1. desempilha um endereço
2. e transfere o controlo para lá

problema : qualquer registo utilizado por g ficará perdido para f

existe várias formas de ultrapassar esta situação,
mas em geral acorda-se e sgue-se uma **convenção de chamada**

- até 6 argumentos são passados para os registos `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`
- os outros são colocados na pilha, se necessário
- o valor de retorno é colocado no registo `%rax`

- os registos `%rbx`, `%rbp`, `%r12`, `%r13`, `%14` et `%r15` ficam **callee-saved** i.e. o *callee* deve salvaguardá-los ; coloca-se ali valores cuja duração de vida é considerada longa - em particular porque precisam de sobreviver ao retorno da função
- os outros registos são **caller-saved** i.e. o *caller* deve salvaguardá-los caso necessário ; coloca-se ali dados que tipicamente não precisam de sobreviver às chamadas

- `%rsp` é o apontador de pilha, `%rbp` é o apontador de *frame* (optional)

na entrada de função, $\%rsp + 8$ deve ser um múltiplo de 16

em particular, funções de bibliotecas (como `puts`, `printf`, `scanf`...) podem causar o fim abrutpo da execução (... `segmentation fault`) caso esta condição não se verifica

no exemplo do `hello world` esta condiçao não foi verificada, tivemos sorte!

o alinhamento da pilha pode ser feito explicitamente

```
f: subq $8, %rsp # alinhamento da pilha
   ...
   ...           # porque fazemos aqui
   ...           # chamadas a funções externas
   ...
   addq $8, %rsp
   ret
```

ou então ser feito "gratuitamente"

```
f: pushq %rbx # guardar %rbx
   ...
   ...        # porque o utilizamos aqui
   ...
   popq %rbx  # restaurar o valor inicial aqui
   ret
```

... são precisamente isso, convenções

em particular, podemos desrespeitá-las por conveniência desde que o código em causa tenha uma execução totalmente controlada localmente

desde que o código em questão tenha de recorrer a código externo, ou que esse possa ser usado por outros
assume-se o respeito das convenções

há quatro tempos na chamada de uma função

1. para o caller, mesmo antes da chamada
2. para o callee, no início da chamada
3. para o callee, no fim da chamada
4. para o caller, logo a seguir ao retorno da chamada

organizam-se a volta de um segmento no topo da pilha, chamado de **tabela de activação** (em inglês **stack frame** ou ocasionalmente **activation record**) localizado entre `%rsp` e `%rbp`

1. passa os argumentos para `%rdi`,...,`%r9`, os restantes são passados para a pilha, se estes foram mais do que 6
2. salvaguarda os registos *caller-saved* que entende utilizar após a chamada (na sua própria tabela de activação)
3. executa

```
call  callee
```

callee, no início da chamada

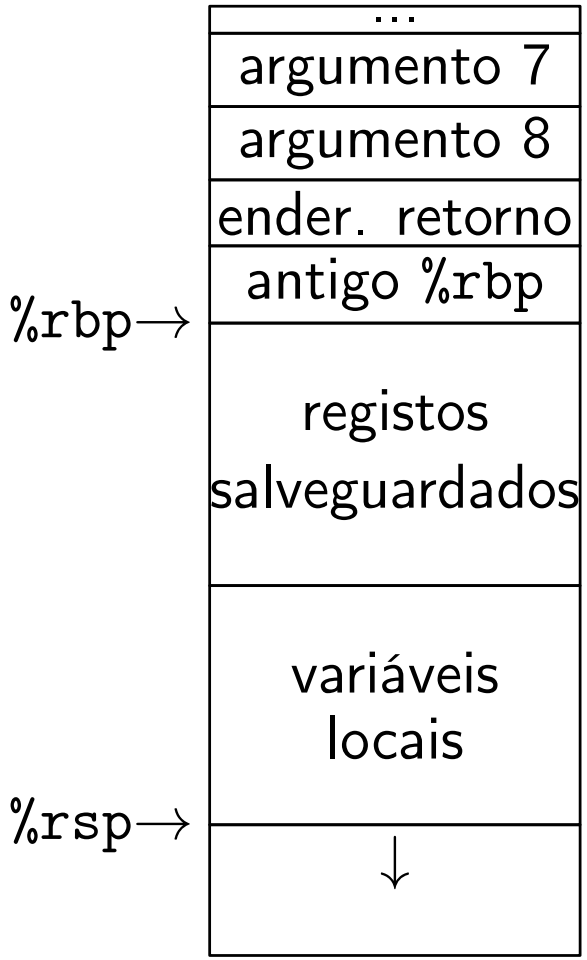
1. salvaguarda %rbp e actualiza a sua posição, por exemplo

```
pushq   %rbp
movq    %rsp, %rbp
```

2. aloca a sua tabela de activação, por exemplo

```
subq    $48, %rsp
```

3. salvaguarda os registos *callee-saved* de que precisa



%rbp permite atingir facilmente os argumentos e as variáveis locais, com um deslocamento fixo qualquer que seja o estado da pilha

callee, no fim da chamada

1. coloca o resultado em `%rax`
2. restabelece os registos salvaguardados
3. desempilha a sua tabela de activação e restabelece `%rbp` com

```
        leave
```

   o que equivale a

```
        movq    %rbp, %rsp
        popq    %rbp
```

4. executa

```
        ret
```

caller, logo a seguir ao fim da chamada

1. desempilha os eventuais argumentos 7, 8, ...
2. restabelece os registos *caller-saved*, caso necessário

exercício : implemente a função seguinte

```
isqrt(n) ≡
    c ← 0
    s ← 1
    while s ≤ n
        c ← c + 1
        s ← s + 2c + 1
    return c
```

que valor devolve isqrt(17)?

exercício : implemente a função factorial

- como uma função iterativa (com base num ciclo)
- como uma função recursiva

- uma máquina fornece
  - um conjunto limitado de instruções, de baixo nível e muito primitivas
  - registos eficazes, um acesso custoso à memória
- a memória é organizada em
  - código / dados estáticos / *heap* (dados dinâmicos) / pilha
- as chamadas às funções organizam-se recorrendo
  - a noção das tabelas de activação
  - de convenções de chamadas

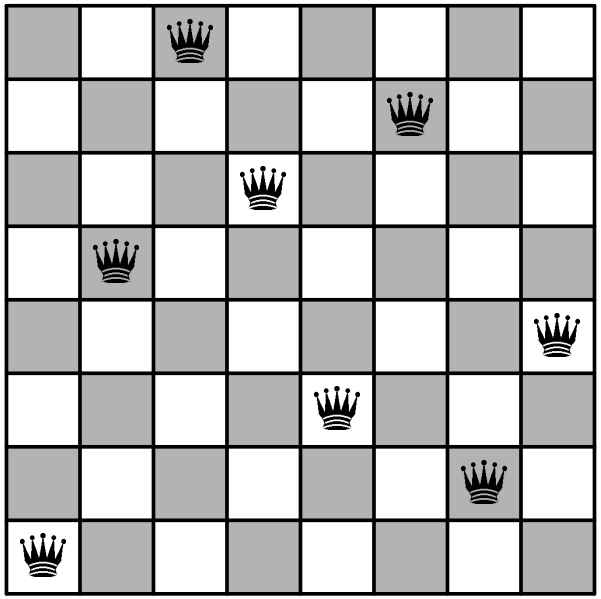
```
t(a,b,c){int d=0,e=a&~b&~c,f=1;if(a)
for(f=0;d=(e-=d)&-e;f+=t(a-d,(b+d)*2,
(c+d)/2));return f;}main(q){scanf("%d",
&q);printf("%d\n",t(~(~0<<q),0,0));}
```

```
int t(int a, int b, int c) {
  int d=0, e=a&~b&~c, f=1;
  if (a)
    for (f=0; d=(e-=d)&-e; f+=t(a-d, (b+d)*2, (c+d)/2));
  return f;
}

int main() {
  int q;
  scanf("%d", &q);
  printf("%d\n", t(~(~0<<q), 0, 0));
}
```

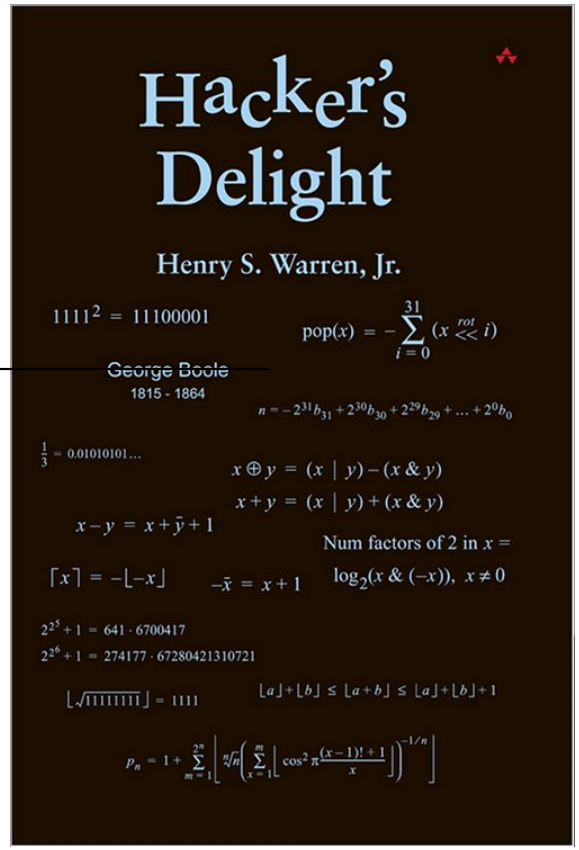
clarificação (continuação)

```
int t(int a, int b, int c) {
  int f=1;
  if (a) {
    int d, e=a&~b&~c;
    f = 0;
    while (d=e&-e) {
      f += t(a-d, (b+d)*2, (c+d)/2);
      e -= d;
    }
  }
  return f;
}
int main() {
  int q;
  scanf("%d", &q);
  printf("%d\n", t(~(~0<<q), 0, 0));
}
```

este programa calcula o número de soluções do problema dito das *n* rainhas



- procura por força bruta (*backtracking*)
- inteiros utilizados como conjuntos :
  por ex. $13 = 0 \cdots 01101_2 = \{0, 2, 3\}$



inteiros	conjuntos
0	$\emptyset$
a&b	$a \cap b$
a+b	$a \cup b$, quando $a \cap b = \emptyset$
a-b	$a \setminus b$, quando $b \subseteq a$
~a	$\complement a$
a&-a	$\{min(a)\}$, quando $a \neq \emptyset$
~(~0«n)	$\{0, 1, \ldots, n-1\}$
a*2	$\{i + 1 \mid i \in a\}$, designado de $S(a)$
a/2	$\{i - 1 \mid i \in a \wedge i \neq 0\}$, designado de $P(a)$

no complemento a dois : `-a` = `~a+1`

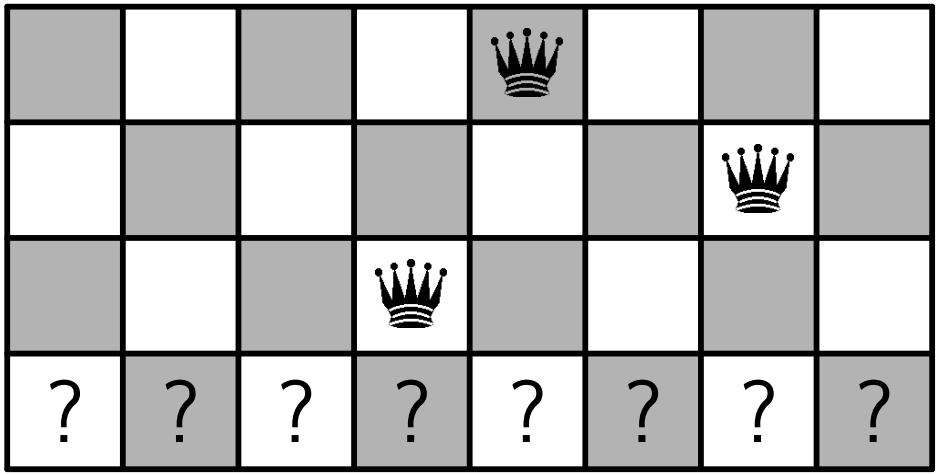
$$
\begin{aligned}
\texttt{a} &= b_{n-1}b_{n-2}\ldots b_k 10\ldots 0 \\
\texttt{\~{}a} &= \overline{b_{n-1}b_{n-2}}\ldots \overline{b_k}01\ldots 1 \\
\texttt{-a} &= \overline{b_{n-1}b_{n-2}}\ldots \overline{b_k}10\ldots 0 \\
\texttt{a\&-a} &= \quad 0 \quad 0\ldots \quad 010\ldots 0
\end{aligned}
$$

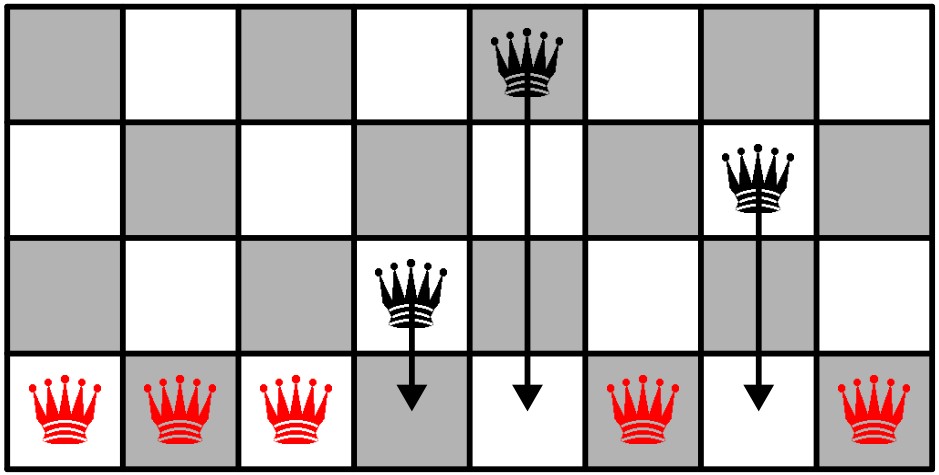
exemplo :

$$
\begin{aligned}
\texttt{a} &= 00001100 &&= 12 \\
\texttt{-a} &= 11110100 &&= -128 + 116 \\
\texttt{a\&-a} &= 00000100
\end{aligned}
$$

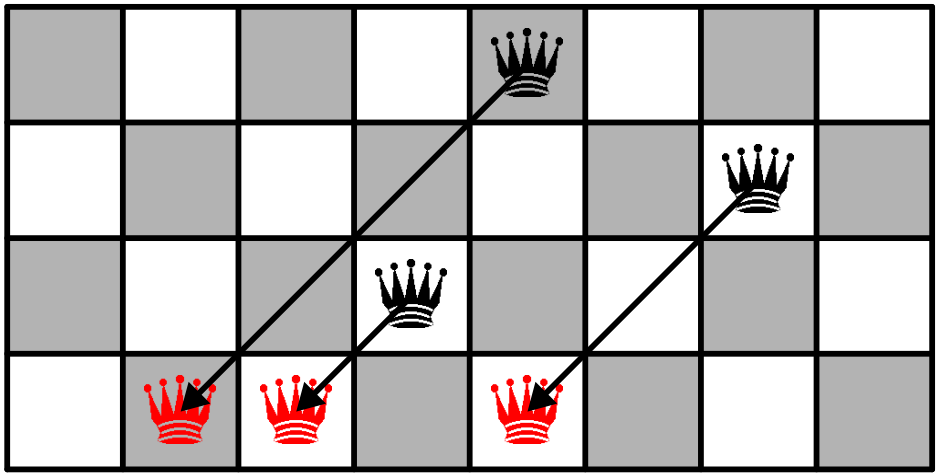
```
int t(a, b, c)
    f ← 1
    if a ≠ ∅
        e ← (a\b)\c
        f ← 0
        while e ≠ ∅
            d ← min(e)
            f ← f + t(a\{d}, S(b ∪ {d}), P(c ∪ {d}))
            e ← e\{d}
    return f

int queens(n)
    return t({0, 1, ..., n − 1}, ∅, ∅)
```

significado de a, b e c

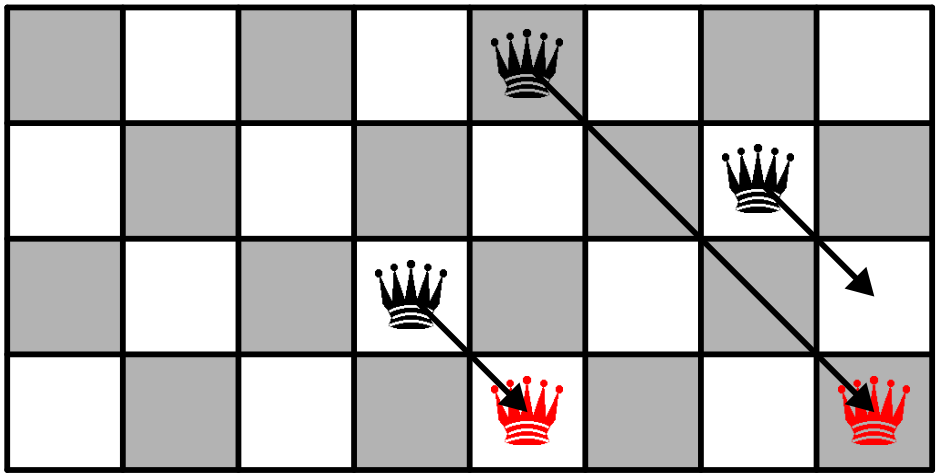




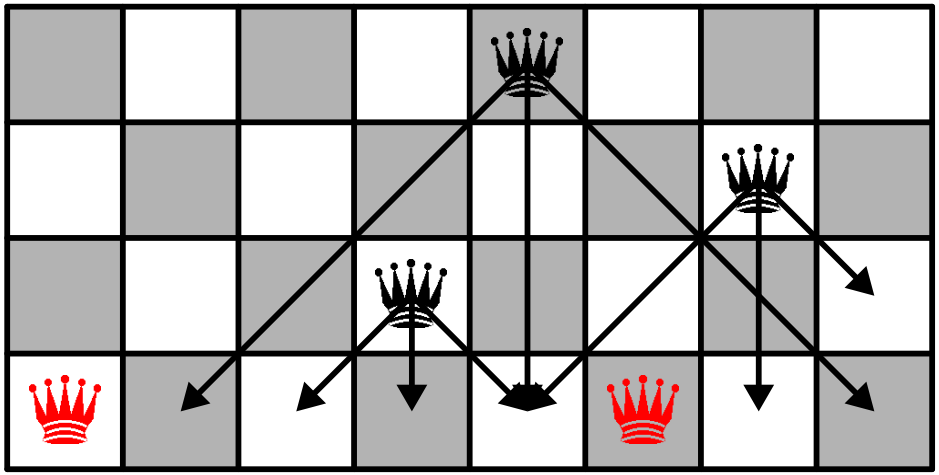
a = colunas por preencher = 11100101_2



b = posições proibidas por causa das diagonais para a esquerda = 01101000_2



c = posições proibidas por causa das diagonais para a direita = 00001001_2



a&~b&~c = posições por tentar = 10000100_2

interesse deste programa para a compilação

```
int t(int a, int b, int c) {
  int f=1;
  if (a) {
    int d, e=a&~b&~c;
    f = 0;
    while (d=e&-e) {
      f += t(a-d,(b+d)*2,(c+d)/2);
      e -= d;
    }
  }
  return f;
}
int main() {
  int q;
  scanf("%d", &q);
  printf("%d\n", t(~(~0<<q), 0, 0));
}
```

curto, mas contém

- um teste (`if`)
- um ciclo (`while`)
- uma função recursiva
- alguns cálculos

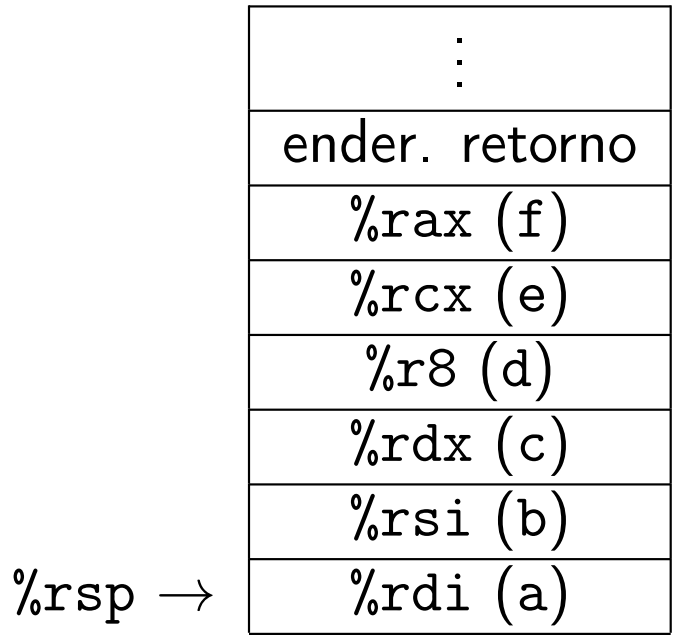
é também uma solução **fantástica** ao problema das n rainhas

comecemos pela função recursiva `t` ; deve-se

- alocar os registos
- compilar
  - o teste
  - o ciclo
  - a chamada recursiva
  - os diferentes cálculos

- `a`, `b` e `c` são passados para `%rdi`, `%rsi` e `%rdx`
- o resultado é devolvido via `%rax`
- as variáveis locais `d`, `e` e `f` são armazenadas em `%r8`, `%rcx` e `%rax`

- no caso da chamada recursiva, `a`, `b`, `c`, `d`, `e` e `f` precisarão de salvaguarda, porque são todos utilizados após a chamada $\Rightarrow$ salvaguardados na pilha



criação/destruição da tabela de activação

```
t:
        subq    $48, %rsp
        ...
        addq    $48, %rsp
        ret
```

```
int t(int a, int b, int c) {
  int f=1;
  if (a) {
    ...
  }
  return f;
}
```

```
        movq    $1, %rax
        testq   %rdi, %rdi
        jz      t_return
        ...
t_return:
        addq    $48, %rsp
        ret
```

caso geral (a $\neq$ 0)

```
if (a) {
  int d, e=a&~b&~c;
  f = 0;
  while ...
}
```

```
xorq  %rax, %rax  # f <- 0
movq  %rdi, %rcx  # e <- a & ~b & ~c
movq  %rsi, %r9
notq  %r9
andq  %r9, %rcx
movq  %rdx, %r9
notq  %r9
andq  %r9, %rcx
```

realça-se a utilização do registo temporário %r9 (não salvaguardado)

```
while (expr) {
  body
}
```

```
        ...
L1:     ...
           cálculo de expr para %rcx
        ...
        testq   %rcx, %rcx
        jz      L2
        ...
           body
        ...
        jmp     L1
L2:     ...
```

existe no entanto uma melhor solução

```
while (expr) {
  body
}
```

```
        ...
        jmp     L2
L1:     ...
          body
        ...
L2:     ...
          expr
        ...
        testq   %rcx, %rcx
        jnz     %rcx, L1
```

assim, faz-se um só salto por exploração do corpo do ciclo
(com a exepção da primeira vez)

```
while (d=e&-e) {
   ...
}
```

```
        jmp     loop_test
loop_body:
        ...
loop_test:
        movq    %rcx, %r8
        movq    %rcx, %r9
        negq    %r9
        andq    %r9, %r8
        testq   %r8, %r8  # inútil
        jnz     loop_body
t_return:
        ...
```

compilação do ciclo (cont.)

```
while (...)  {
  f += t(a-d,
         (b+d)*2,
         (c+d)/2);
  e -= d;
}
```

```
loop_body:
        movq    %rdi, 0(%rsp)   # a
        movq    %rsi, 8(%rsp)   # b
        movq    %rdx, 16(%rsp)  # c
        movq    %r8, 24(%rsp)   # d
        movq    %rcx, 32(%rsp)  # e
        movq    %rax, 40(%rsp)  # f
        subq    %r8, %rdi
        addq    %r8, %rsi
        salq    $1, %rsi
        addq    %r8, %rdx
        shrq    $1, %rdx
        call    t
        addq    40(%rsp),%rax   # f
        movq    32(%rsp), %rcx  # e
        subq    24(%rsp), %rcx  # -= d
        movq    16(%rsp), %rdx  # c
        movq    8(%rsp), %rsi   # b
        movq    0(%rsp), %rdi   # a
```

```c
int main() {
  int q;
  scanf("%d", &q);
  ...
}
```

```
main:
        movq    $input, %rdi
        movq    $q, %rsi
        xorq    %rax, %rax
        call    scanf
        movq    (q), %rcx
        ...

        .data
input:
        .string "%d"
q:
        .quad   0
```

programa principal (cont.)

```c
int main() {
  ...
  printf("%d\n",
         t(~(~0«q), 0, 0));
}
```

```
main:
        ...
        xorq    %rdi, %rdi
        notq    %rdi
        salq    %cl, %rdi
        notq    %rdi
        xorq    %rsi, %rsi
        xorq    %rdx, %rdx
        call    t
        movq    $msg, %rdi
        movq    %rax, %rsi
        xorq    %rax, %rax
        call    printf
        xorq    %rax, %rax
        ret
```

este código não é optimo

(por exemplo, cria-se desnecessariamente uma tabela de activação quando a = 0, poder-se-ia salvaguardar s5 registo somente)

mas é tão eficiente quanto à versão produzida pelo `gcc -O2`

uma diferença fundamental, no entanto : escrevemos o código *assembly* **específico** para este programa à mão, e não via um compilador !

- produzir código *assembly* não é tarefa fácil
  (basta observar o código produzido com `gcc -S -fverbose-asm` ou ainda `ocamlopt -S`)

- agora é necessário automatizar todo este processo

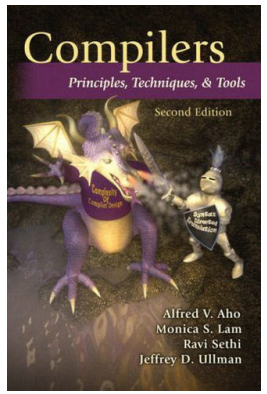
ler

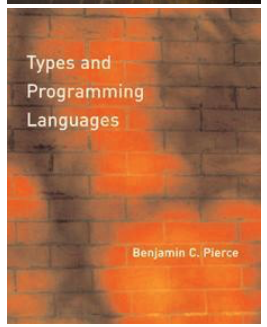
- *Computer Systems: A Programmer's Perspective*
  (R. E. Bryant, D. R. O'Hallaron)
- o seu complemento PDF *x86-64 Machine-Level Programming*

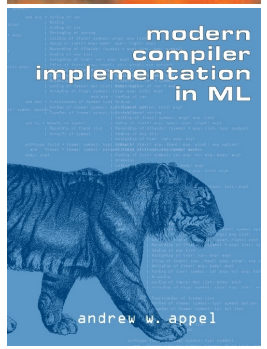
- Prática desta semana
  - pequenos exercícios sobre *assembly*
  - geração de código para uma linguagem básica de expressões aritméticas
- a próxima semana
  - sintaxe abstrata
  - semântica
  - interpretador

leituras de referência

estes acetatos resultam essencialmente de uma adaptação do material pedagógico gentilmente cedido pelo Jean-Christophe Filliâtre (link1, link2)







adicionalmente poderá consultar as obras seguintes

- **Modern Compilers: Principles, Techniques, and Tools**, Alfred V. Aho, Monica S. Lam, Ravi Sethi et Jeffrey D. Ullman
- **Types and Programming Languages**, Benjamin C. Pierce
- **Modern Compiler Implementation**, Andrew W. Appel (3 versões: ML, C, Java)