



Estrutura do tema ISA do IA-32

1. Desenvolvimento de programas no IA-32 em Linux
2. Acesso a operandos e operações
3. Suporte a estruturas de controlo
4. Suporte à invocação/regresso de funções
5. **Análise comparativa: IA-32 vs. x86-64 e RISC (MIPS e ARM)**
6. Acesso e manipulação de dados estruturados



Principal diferença na organização interna:

- organização dos registos (na codificação de funções)
 - IA-32: poucos registos genéricos (**só 6**) => variáveis locais em reg e argumentos na *stack*
 - Intel 64: 16 registos genéricos => mais registos, para variáveis locais (**8**) & para passagem e uso de argumentos (**6**)
- impacto na execução de funções:
 - menor utilização da *stack* na arquitetura Intel 64
 - Intel 64 potencialmente mais eficiente

Análise de um exemplo (*swap*) ...

x86-64: 64-bit extension to IA-32

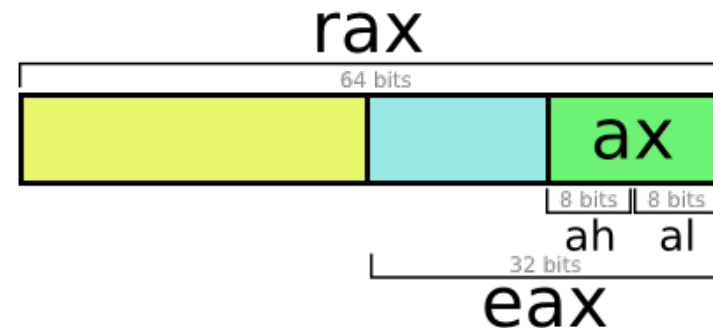
Intel 64: Intel implementation of x86-64



x86-64 Integer Registers

<code>%rax</code> <code>%eax</code>	<code>%r8</code> <code>%r8d</code>
<code>%rbx</code> <code>%ebx</code>	<code>%r9</code> <code>%r9d</code>
<code>%rcx</code> <code>%ecx</code>	<code>%r10</code> <code>%r10d</code>
<code>%rdx</code> <code>%edx</code>	<code>%r11</code> <code>%r11d</code>
<code>%rsi</code> <code>%esi</code>	<code>%r12</code> <code>%r12d</code>
<code>%rdi</code> <code>%edi</code>	<code>%r13</code> <code>%r13d</code>
<code>%rsp</code> <code>%esp</code>	<code>%r14</code> <code>%r14d</code>
<code>%rbp</code> <code>%ebp</code>	<code>%r15</code> <code>%r15d</code>

- Twice the number of registers
- Accessible as 8, 16, 32, 64 bits



x86-64 Integer Registers: Usage Conventions

<code>%rax</code> Return value	<code>%r8</code> Argument #5
<code>%rbx</code> Callee saved	<code>%r9</code> Argument #6
<code>%rcx</code> Argument #4	<code>%r10</code> Caller saved
<code>%rdx</code> Argument #3	<code>%r11</code> Caller Saved
<code>%rsi</code> Argument #2	<code>%r12</code> Callee saved
<code>%rdi</code> Argument #1	<code>%r13</code> Callee saved
<code>%rsp</code> Stack pointer	<code>%r14</code> Callee saved
<code>%rbp</code> Callee saved	<code>%r15</code> Callee saved



Principal diferença na organização interna:

- organização dos registos
 - IA-32: poucos registos genéricos (só 6) => variáveis locais em reg e argumentos na *stack*
 - Intel 64: 16 registos genéricos => mais registos para variáveis locais (8) & para passagem e uso de argumentos (6)
- impacto na execução de funções:
 - menor utilização da *stack* na arquitetura Intel 64
 - Intel 64 potencialmente mais eficiente

Análise de um exemplo (swap) ...

Revisão da codificação de swap e call_swap no IA-32



```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```
void call_swap()
{
    int zip1 = 15213;
    int zip2 = 91125;
    swap(&zip1, &zip2);
}
```

```
_swap:
    pushl    %ebp
    movl    %esp, %ebp
    pushl    %ebx

    movl    12(%ebp), %ecx
    movl    8(%ebp), %edx
    movl    (%ecx), %eax
    movl    (%edx), %ebx
    movl    %eax, (%edx)
    movl    %ebx, (%ecx)

    movl    -4(%ebp), %ebx
    movl    %ebp, %esp
    popl    %ebp
    ret
```

```
_call_swap:
    pushl    %ebp
    movl    %esp, %ebp
    subl    $24, %esp

    movl    $15213, -4(%ebp)
    movl    $91125, -8(%ebp)
    leal    -4(%ebp), %eax
    movl    %eax, (%esp)
    leal    -8(%ebp), %eax
    movl    %eax, 4(%esp)
    call    _swap

    movl    %ebp, %esp
    popl    %ebp
    ret
```

Funções em assembly: IA-32 versus Intel 64

```
IA-32  
_swap:  
    pushl    %ebp  
    movl    %esp, %ebp  
    pushl    %ebx  
  
    movl    8(%ebp), %edx  
    movl    12(%ebp), %ecx  
    movl    (%edx), %ebx  
    movl    (%ecx), %eax  
    movl    %eax, (%edx)  
    movl    %ebx, (%ecx)  
  
    popl    %ebx  
    popl    %ebp  
    ret  
  
_call_swap:  
    pushl    %ebp  
    movl    %esp, %ebp  
    subl    $24, %esp  
  
    movl    $15213, -4(%ebp)  
    movl    $91125, -8(%ebp)  
    leal    -4(%ebp), %eax  
    movl    %eax, (%esp)  
    leal    -8(%ebp), %eax  
    movl    %eax, 4(%esp)  
    call    _swap  
  
    movl    %ebp, %esp  
    popl    %ebp  
    ret  
  
Total:  
63 bytes
```

```
Intel 64  
swap:  
    pushq   %rbp  
    movq   %rsp, %rbp  
  
    movl   (%rdi), %eax  
    movl   (%rsi), %ecx  
    movl   %ecx, (%rdi)  
    movl   %eax, (%rsi)  
  
    popq   %rbp  
    retq  
  
call_swap:  
    pushq   %rbp  
    movq   %rsp, %rbp  
    subq   $16, %rsp  
  
    movl   $15213, -4(%rbp)  
    movl   $91125, -8(%rbp)  
    leaq   -4(%rbp), %rdi  
  
    leaq   -8(%rbp), %rsi  
    callq  _swap  
  
    addq   $16, %rsp  
    popq   %rbp  
    retq  
  
Total:  
54 bytes
```

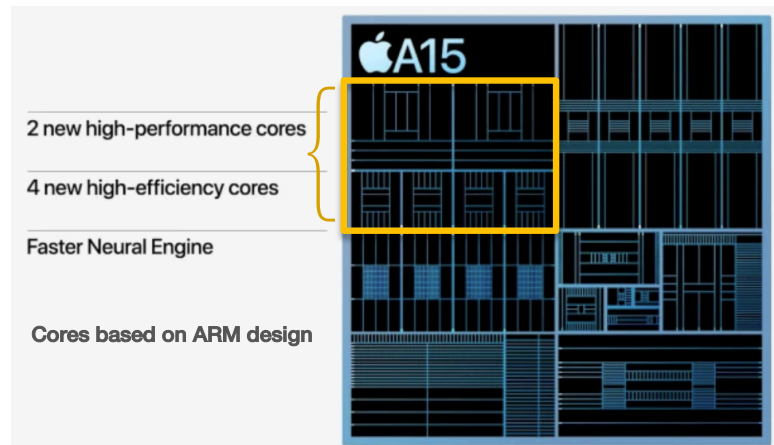
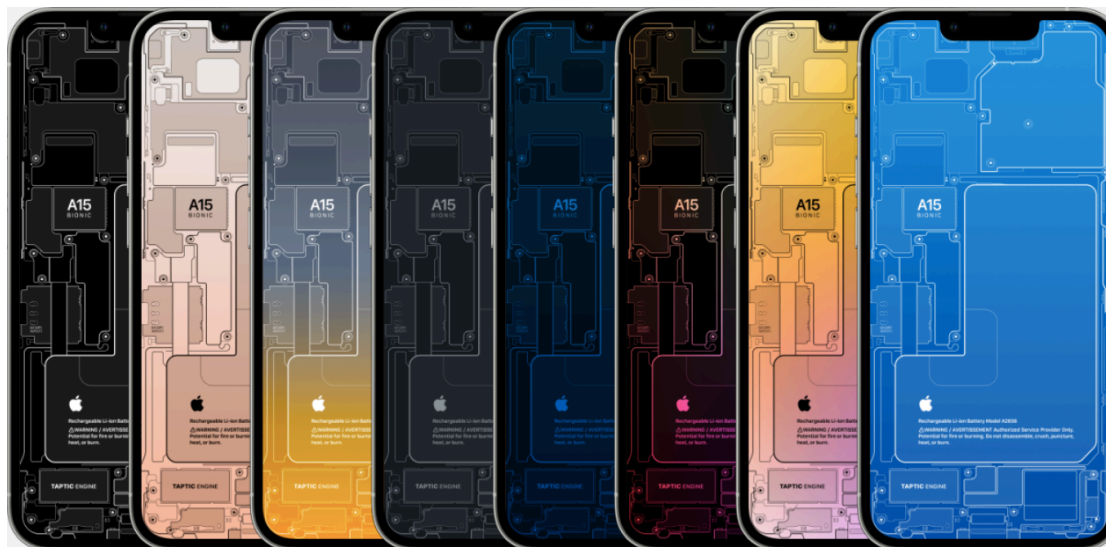
Total de acessos à stack: 15 no IA-32, 9 no Intel 64 !



Caracterização das arquiteturas RISC

- conjunto reduzido e simples de instruções
- formatos simples de instruções
- uma operação elementar por ciclo máquina
- operandos sempre em registos
- modos simples de endereçamento à memória

Arquiteturas RISC: **em todos os smartphones!**



Análise do nível ISA: o modelo RISC versus IA-32 (1)



RISC versus IA-32 :

- RISC: conjunto reduzido e simples de instruções
 - pouco mais que o *subset* do IA-32 já apresentado...
 - instruções simples, mas muito eficientes em *pipeline*
- operações aritméticas e lógicas:
 - 3-operandos (RISC) versus 2-operandos (IA-32)
 - RISC: operandos sempre em registos,
16/32 registos genéricos visíveis ao programador,
sendo normalmente
 - 1 reg apenas de leitura, com o valor 0 (em 32 registos)
 - 1 reg usado para guardar o endereço de regresso da função
 - 1 reg usado como *stack pointer* (convenção do s/w)





RISC versus IA-32 (cont.):

– RISC: modos simples de endereçamento à memória

- apenas 1 modo de especificar o endereço:

$\text{Mem}[C^{te} + (\text{Reg}_b)]$ ou $\text{Mem}[(\text{Reg}_b) + (\text{Reg}_i)]$

- ou poucos modos de especificar o endereço:

$\text{Mem}[C^{te} + (\text{Reg}_b)]$ e/ou

$\text{Mem}[(\text{Reg}_b) + (\text{Reg}_i)]$ e/ou

$\text{Mem}[C^{te} + (\text{Reg}_b) + (\text{Reg}_i)]$

– RISC: uma operação elementar em cada instrução

- por ex. `push/pop` (IA-32)

substituído pelo par de operações elementares

`sub&store/load&add` (RISC)

– . . .

Análise do nível ISA: o modelo RISC versus IA-32 (3)



RISC versus IA-32 (cont.):

– RISC: formatos simples de instruções

- comprimento fixo e poucas variações
- ex.: MIPS

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

- ex.: ARM

Name	Format	Example								Comments
Field size		4 bits	2 bits	1 bit	4 bits	1 bit	4 bits	4 bits	12 bits	All ARM instructions are 32 bits long
DP format	DP	Cond	F	I	Opcode	S	Rn	Rd	Operand2	Arithmetic instruction format
DT format	DT	Cond	F	Opcode			Rn	Rd	Offset12	Data transfer format
Field size		4 bits	2 bits	2 bits	24 bits					
BR format	BR	Cond	F	Opcode	signed_immed_24					B and BL instructions

Simple comparison entre ARM e MIPS

- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

	ARM	MIPS
Date announced	1985	1985
Instruction size	32 bits	32 bits
Address space	32-bit flat	32-bit flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Registers	15 × 32-bit	31 × 32-bit
Input/output	Memory mapped	Memory mapped



Principal diferença na organização interna:

- organização dos registos (na codificação de funções)
 - IA-32: poucos registos genéricos => variáveis e argumentos normalmente na *stack*
 - RISC: 16/32 registos genéricos => mais registos, para variáveis locais & registos para passagem de argumentos & registo para endereço de regresso
- impacto na execução de funções:
 - menor utilização da *stack* nas arquiteturas RISC
 - RISC potencialmente mais eficiente

Análise de um exemplo (*swap*) ...



Principal diferença na organização interna:

- organização dos registos
 - IA-32: poucos registos genéricos => variáveis e argumentos normalmente na *stack*
 - RISC: 16/32 registos genéricos => mais registos, para variáveis locais & registos para passagem de argumentos & registo para endereço de regresso
- impacto na execução de funções:
 - menor utilização da *stack* nas arquiteturas RISC
 - RISC potencialmente mais eficiente

Análise de um exemplo (*swap*) ...

Revisão da codificação de swap e call_swap no IA-32



```
void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```
_swap:
    pushl   %ebp
    movl   %esp, %ebp
    pushl   %ebx

    movl   12(%ebp), %ecx
    movl   8(%ebp), %edx
    movl   (%ecx), %eax
    movl   (%edx), %ebx
    movl   %eax, (%edx)
    movl   %ebx, (%ecx)

    movl   -4(%ebp), %ebx
    movl   %ebp, %esp
    popl   %ebp
    ret
```

```
void call_swap()
{
    int zip1 = 15213;
    int zip2 = 91125;
    swap(&zip1, &zip2);
}
```

```
_call_swap:
    pushl   %ebp
    movl   %esp, %ebp
    subl   $24, %esp

    movl   $15213, -4(%ebp)
    movl   $91125, -8(%ebp)
    leal   -4(%ebp), %eax
    movl   %eax, (%esp)
    leal   -8(%ebp), %eax
    movl   %eax, 4(%esp)
    call   _swap

    movl   %ebp, %esp
    popl   %ebp
    ret
```

Convenção na utilização dos registos MIPS



MIPS

Name	Register	Usage
\$zero	\$0	Always 0 (forced by hardware)
\$at	\$1	Reserved for assembler use
\$v0 – \$v1	\$2 – \$3	Result values of a function
\$a0 – \$a3	\$4 – \$7	Arguments of a function
\$t0 – \$t7	\$8 – \$15	Temporary Values
\$s0 – \$s7	\$16 – \$23	Saved registers (preserved across call)
\$t8 – \$t9	\$24 – \$25	More temporaries
\$k0 – \$k1	\$26 – \$27	Reserved for OS kernel
\$gp	\$28	Global pointer (points to global data)
\$sp	\$29	Stack pointer (points to top of stack)
\$fp	\$30	Frame pointer (points to stack frame)
\$ra	\$31	Return address (used by jal for function call)

Funções em assembly: IA-32 versus MIPS (RISC) (1)



		IA-32
_swap:		
pushl	%ebp	
movl	%esp, %ebp	
pushl	%ebx	
movl	8(%ebp), %edx	
movl	12(%ebp), %ecx	
movl	(%edx), %ebx	
movl	(%ecx), %eax	
movl	%eax, (%edx)	
movl	%ebx, (%ecx)	
popl	%ebx	
popl	%ebp	
ret		
_call_swap:		
pushl	%ebp	
movl	%esp, %ebp	
subl	\$24, %esp	
movl	\$15213, -4(%ebp)	
movl	\$91125, -8(%ebp)	
leal	-4(%ebp), %eax	
movl	%eax, (%esp)	
leal	-8(%ebp), %eax	
movl	%eax, 4(%esp)	
call	_swap	
movl	%ebp, %esp	
popl	%ebp	
ret		
Total:		63 bytes

		MIPS
swap:		
lw	\$v1, 0(\$a0)	
lw	\$v0, 0(\$a1)	
sw	\$v0, 0(\$a0)	
sw	\$v1, 0(\$a1)	
j	\$ra	
call_swap:		
subu	\$sp, \$sp, 32	
sw	\$ra, 24(\$sp)	
li	\$v0, 15213	
sw	\$v0, 16(\$sp)	
li	\$v0, 0x10000	
ori	\$v0, \$v0, 0x63f5	
sw	\$v0, 20(\$sp)	
addu	\$a0, \$sp, 16	# &zip1= sp+16
addu	\$a1, \$sp, 20	# &zip2= sp+20
jal	swap	
lw	\$ra, 24(\$sp)	
addu	\$sp, \$sp, 32	
j	\$ra	
Total:		72 bytes

Funções em assembly: IA-32 versus MIPS (RISC) (2)



call_swap

1. Invocar swap

- salvaguardar registos
- passagem de argumentos
- chamar rotina e guardar endereço de regresso

```
leal    -4(%ebp), %eax
pushl   %eax
leal    -8(%ebp), %eax
pushl   %eax
call    swap
```

Não há reg para salvag.

Calcula &zip2

Push &zip2

Calcula &zip1

Push &zip1

Invoca swap

IA-32

Acessos
à stack

MIPS

```
sw      $ra, 24($sp) Salvag. reg c/ endereço regresso
addu   $a0, $sp, 16 Calcula & coloca &zip1 no reg arg 0
addu   $a1, $sp, 20 Calcula & coloca &zip2 no reg arg 1
jal    swap Invoca swap
```

Funções em assembly: IA-32 versus MIPS (RISC) (3)



swap

1. Inicializar swap

- atualizar *frame pointer*
- salvar registos
- reservar espaço p/ locais

swap:

`pushl %ebp`

`movl %esp, %ebp`

`pushl %ebx`

Salvag. antigo %ebp

%ebp novo frame pointer

Salvag. %ebx

Não é preciso espaço p/ locais

IA-32

Acessos
à stack

MIPS

Frame pointer p/ atualizar: NÃO

Registos p/ salvar: NÃO

Espaço p/ locais: NÃO

Funções em assembly: IA-32 versus MIPS (RISC) (4)



swap

2. Corpo de swap ...

<code>movl 12(%ebp), %ecx</code>	<i>Coloca y em reg</i>
<code>movl 8(%ebp), %edx</code>	<i>Coloca x em reg</i>
<code>movl (%ecx), %eax</code>	<i>Coloca y em reg</i>
<code>movl (%edx), %ebx</code>	<i>Coloca x em reg</i>
<code>movl %eax, (%edx)</code>	<i>Armazena y em *xp</i>
<code>movl %ebx, (%ecx)</code>	<i>Armazena x em *yp</i>

IA-32

Acessos
à memória
(todas...)

MIPS

<code>lw \$v1, 0(\$a0)</code>	<i>Coloca x em reg</i>
<code>lw \$v0, 0(\$a1)</code>	<i>Coloca y em reg</i>
<code>sw \$v0, 0(\$a0)</code>	<i>Armazena y em *xp</i>
<code>sw \$v1, 0(\$a1)</code>	<i>Armazena x em *yp</i>

Funções em assembly: IA-32 versus MIPS (RISC) (5)



swap

3. Término de swap ...

- libertar espaço de var locais
- recuperar registos
- recuperar antigo *frame pointer*
- regressar a `call_swap`

```
popl %ebx  
movl %ebp, %esp  
popl %ebp  
ret
```

Não há espaço a libertar

Recupera %ebx

Recupera %esp

Recupera %ebp

Regressa à função chamadora

IA-32

Acessos
à stack

MIPS

```
j $ra
```

Espaço a libertar de var locais: NÃO

Recuperação de registos: NÃO

Recuperação do *frame ptr*: NÃO

Regressa à função chamadora

Funções em assembly: IA-32 versus MIPS (RISC) (6)



`call_swap`

2. Terminar invocação de swap...

- libertar espaço de argumentos na *stack*...
- recuperar registos

```
addl $8, (%esp)
```

Atualiza stack pointer
Não há reg's a recuperar

IA-32

Acessos
à stack

MIPS

```
lw $ra, 24($sp)
```

Espaço a libertar na stack: NÃO
Recupera reg c/ ender regresso

Total de acessos à memória/stack (incl. inicialização de var's em mem):

14(+2) no IA-32, 6(+2) no MIPS !

Convenção na utilização dos registos ARM

ARM

Name	Register number	Usage	Preserved on call?
a1 - a2	0-1	Argument / return result / scratch register	no
a3 - a4	2-3	Argument / scratch register	no
v1 - v8	4-11	Variables for local routine	yes
ip	12	Intra-procedure-call scratch register	no
sp	13	Stack pointer	yes
lr	14	Link Register (Return address)	yes
pc	15	Program Counter	n.a.

Funções em assembly: IA-32 versus ARM (RISC)



```

_swap:
    pushl    %ebp
    movl    %esp, %ebp
    pushl    %ebx

    movl    8(%ebp), %edx
    movl    12(%ebp), %ecx
    movl    (%edx), %ebx
    movl    (%ecx), %eax
    movl    %eax, (%edx)
    movl    %ebx, (%ecx)

    popl    %ebx
    popl    %ebp
    ret

_call_swap:
    pushl    %ebp
    movl    %esp, %ebp
    subl    $24, %esp

    movl    $15213, -4(%ebp)
    movl    $91125, -8(%ebp)
    leal    -4(%ebp), %eax
    movl    %eax, (%esp)
    leal    -8(%ebp), %eax
    movl    %eax, 4(%esp)
    call    _swap

    movl    %ebp, %esp
    popl    %ebp
    ret
    
```

**Total:
63 bytes**

IA-32

```

_swap:
    str     fp, [sp, #-4]!
    add     fp, sp, #0           ; IA-32: mov sp, fp

    ldr     r3, [r0, #0]        ; IA-32: mov 0(r0), r3
    ldr     r2, [r1, #0]
    str     r2, [r0, #0]        ; IA-32: mov r2, 0(r0)
    str     r3, [r1, #0]

    add     sp, fp, #0
    pop     {fp}                ; pop é pseudo-instr
    bl     lr                    ; branch & link

_call_swap:
    push   {fp, lr}            ; push é pseudo-instr
    add     fp, sp, #4
    sub     sp, sp, #8

    ldr     r3, .L3
    str     r3, [fp, #-12]
    ldr     r3, .L3+4
    str     r3, [fp, #-8]
    sub     r0, fp, #12         ; &zip1= fp+12
    sub     r1, fp, #8          ; &zip2= fp+8
    bl     _swap

    sub     sp, fp, #4
    pop     {fp, pc}           ; pop {pc} = ret

.L3:
    .word 15213
    .word 91125
    
```

ARM

GCC (GNU Compiler Collection): linguagens de programação suportadas e ...



Linguagens de programação que GCC suporta

- C com dialetos:
 - ANSI C original (X3.159-1989, ISO standard ISO/IEC 9899:1990) aka **C89** ou **C90**; usar com `'-ansi'`, `'-std=c90'`
 - **C94** ou **C95**, usar com `'-std=iso9899:199409'`
 - **C99**, usar com `'-std=c99'` or `'-std=iso9899:1999'`
 - **C11**, usar com `'-std=c11'`
 - **C17**, usar com `'-std=c17'`
 - por omissão GCC usa **C11** com extensões `'-std=gnu11'`
- C++ (usado com comando `'g++'`) com dialetos:
 - **C++98**, **C++03**, **C++11**, **C++14**, **C++17**, ...
- Fortran (usado com comando `'gfortran'`) com dialetos:
 - **Fortran 95**, **Fortran 2003**, **Fortran 2008**, ...

GCC (GNU Compiler Collection): ... e processadores suportados



Processadores que GCC suporta

- especificado sempre com opção `'-m'`
- opção x86, usar com `'-march=cpu-type'`; algumas escolhas:
 - `'native'` (o sistema local)
 - `'i386'... 'x86-64'... 'kn1'... 'icelake'... 'znver3'`
- opção MIPS, usar com `'-march=arch'`
 - é possível ainda selecionar um dado processador MIPS
- opção ARM, usar com `'-march=name'`
 - é possível ainda selecionar um dado processador ARM; mais comuns:
 - os da última geração de 32 bits, `'armv7'`
 - da 1ª geração de 64 bits, `'armv8-a'`