

Teste 1A de Arquitetura de Computadores

05/05/2017 Duração 2h sem consulta

As perguntas com múltiplas respostas têm de ser respondidas na folha de respostas anexa.
Respostas erradas descontam 1/5 da cotação da pergunta ou alínea.

Número: _____ Nome: _____

1 — (1,5 val) — Considere uma representação de inteiros com sinal (complemento para 2) usando 7 bits.

a) Como é codificado o valor 22 ?

A) 001 0110 B) 001 0010 C) 100 1010 D) 111 0110 E) 001 1010

b) Como é codificado -6 ?

A) 111 1110 B) 100 0110 C) 111 1010 D) 100 0110 E) 100 1010

c) Qual é o maior valor que se pode representar (responda em base 10) ?

A) 64 B) 128 C) 63 D) 127 E) 32

d) Qual é o menor valor que se pode representar (responda em base 10) ?

A) -63 B) -128 C) -32 D) -127 E) -64

2 — (1,5 val) — Considere um CPU que tem uma unidade aritmética e lógica em que as entradas e a saída têm 8 bits. Os inteiros são representados em complemento para 2.

a) Diga qual é o resultado (em base 2) da soma de 01111100_2 com 00010110_2

A) 0110 1010 B) 1001 0010 C) 1111 1110 D) 0110 1110 E) 1011 1010

b) Indique os valores das flags CF, OF, ZF e SF, após a realização da operação (soma) anterior.

A) CF=0; OF=1; ZF=0; SF=1 B) CF=1; OF=1; ZF=0; SF=1 C) CF=0; OF=0; ZF=1; SF=1
D) CF=0; OF=1; ZF=1; SF=0 E) CF=1; OF=0; ZF=1; SF=0

c) Assuma que uma operação aritmética entre números com sinal deu a seguinte configuração de flags: CF=1; OF=1; ZF=0 e SF=1. O que significa esta configuração de flags relativamente à correção do resultado e porquê?

- A) O resultado está correto porque ambas as flags SF e CF estão ativadas.
- B) O resultado está correto porque ambas as flags OF e CF estão ativadas.
- C) O resultado está errado porque a flag CF está ativada.
- D) O resultado está errado porque a flag OF está ativada.
- E) O resultado está errado porque a flag SF está ativada e ZF não.

3 — (1,5 val) — Considere a norma IEEE 754, onde a representação de números reais em precisão simples (32 bits) usa a seguinte interpretação:

- Bit 31: sinal – 0 se maior ou igual a zero, 1 se menor do que zero
- Bits 30 a 23: expoente somado de 127
- Bits 22 a 0: parte fracionária da mantissa (o valor efetivo da mantissa é 1.XXX...XXX)

Apresente a representação (binária, 32 bits) do valor decimal -5.75 nesta representação.

- A) 11000000 10111000 00000000 00000000
- B) 10000010 10111000 00000000 00000000
- C) 01111101 10111000 00000000 00000000
- D) 11111101 10111000 00000000 00000000
- E) 10000010 11100000 00000000 00000000

A) (num >> 1) & 0x1 B) (num >> 16) & 0x1 C) (num & 0x8000) >> 1
D) num % 2 **E) (num >> 15) & 0x1**

```
#include <stdio.h>
int main() {
    float x = 2/10;    printf ("x=%f\n", x);
}
```

6 — (1,0 val) — Considere a função do compilador de uma linguagem (como C) do *assembler* (as) e do *linker* (ld) na construção de um programa executável. Escolha a ordem possível das transformações efectuadas por estas ferramentas e o respectivo código produzido em cada transformação:

7 — (1,5 val) — Estime o tempo necessário para a execução dos dois conjuntos de instruções que somam 20 à var1 deixando o resultado em var2. Qual das afirmações seguintes é verdadeira?

A) O conjunto 1 executa em menos tempo porque tem menos instruções

B) O conjunto 2 executa em menos tempo porque tem menos acessos a memória

C) O conjunto 1 executa em menos tempo porque tem menos instruções e faz as somas sempre sobre a mesma variável, var2

D) O conjunto 2 executa em menos tempo porque só acede uma vez a cada variável e faz as somas sempre usando o mesmo registo, ECX

E) Vão executar no mesmo tempo pois produzem o mesmo efeito no CPU e memória.

8 — (1,5 val) — Dado o conteúdo das localizações de memória 0x200 a 0x209 de um computador e sabendo que o processador Intel utiliza uma arquitetura *little-endian*, qual o conteúdo correto dos registos %ebx, %ecx e %edx do processador após a execução das seguintes instruções. Assuma que estes registos estão todos inicializados com o valor zero. O valor '*' num registo quer dizer que, dado que só se conhece os conteúdos da região de memória 0x200 a 0x209, não é possível determinar qual o valor que ficaria nesse registo.

Endereço	Conteúdo	Programa
0x200	2	movl \$0x206, %eax
0x201	0	movl (%eax), %ebx
0x202	3	movb -2(%ebx), %cl
0x203	4	movl -2(%ebx, %ecx, 2), %edx
0x204	0	
0x205	3	
0x206	2	
0x207	2	
0x208	0	
0x209	0	

- A) ebx=0x02020000 ecx=0x00000003 edx=0x00030202
- B) ebx=0x00000202 ecx=0x02000304 edx=*
- C) ebx=0x02020000 ecx=* edx=*
- D) ebx=0x00000206 ecx=0x00000002 edx=*
- E) ebx=0x00000202 ecx=0x00000002 edx=0x02020300**

9 — (1,5 val) — Considere a seguinte sequência de instruções:

```
movl $5, %eax
movl $6, %ebx
push %eax
push %ebx
push %eax
pop %ecx
pop %eax
```

Selecione a opção que apresenta corretamente o conteúdo dos registos após a execução destas instruções. Um '*' no valor do registo significa que com a informação disponibilizada na questão não é possível dizer qual o conteúdo desse registo.

- A) eax=0x00000005 ebx=0x0x00000006 ecx=*
- B) eax=0x00000005 ebx=0x0x00000006 ecx=0x00000005
- C) eax=0x00000005 ebx=0x0x00000006 ecx=0x00000006
- D) eax=0x00000006 ebx=0x0x00000006 ecx=0x00000005**
- E) eax=0x00000005 ebx=0x0x00000005 ecx=0x00000006

10 — (1,5 val) — A comunicação entre o CPU e a memória é assegurada por um conjunto de ligações denominado bus de sistema. Selecione a frase que melhor descreve o que acontece na execução, por um CPU Intel, da instrução máquina correspondente ao seguinte:

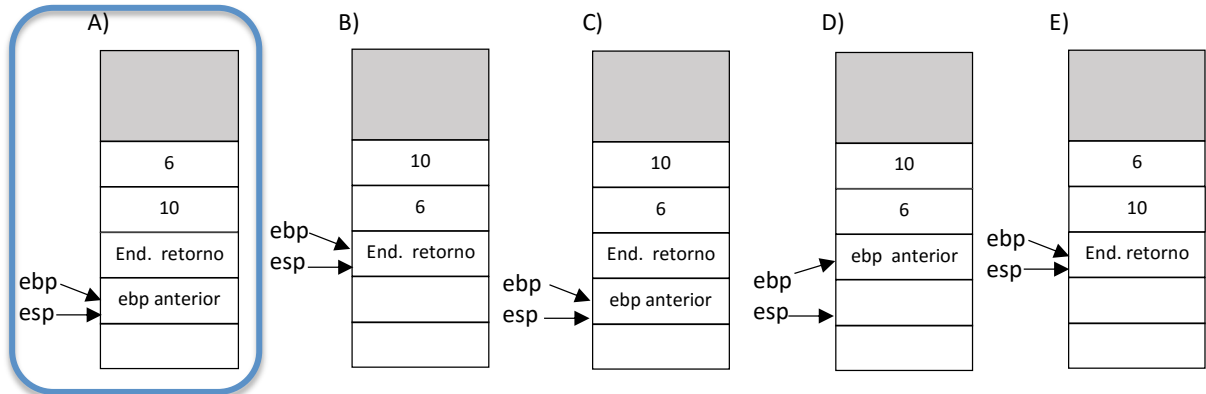
mov %eax, 100(%ebx)

- A) O CPU coloca no bus de endereços o valor 100 e depois o valor em EBX, em resposta, a memória coloca no bus de dados o conteúdo dos 4 bytes a partir desse endereço
- B) O CPU coloca no bus de endereços o valor 100, no bus de dados o conteúdo do registo EAX e no bus de controlo o conteúdo do registo EBX
- C) O CPU coloca no bus de endereços o valor resultante da soma de 100 com EBX e, em resposta, a memória coloca no bus de dados o conteúdo dos 4 bytes a partir desse endereço
- D) O CPU coloca no bus de endereços o valor resultante da soma de 100 com EBX, no bus de dados o conteúdo do registo EAX e no bus de controlo o comando para escrever na memória**
- E) O CPU coloca no bus de endereços o valor 100 e depois o valor em EBX, no bus de dados o conteúdo do registo EAX e no bus de controlo o comando para escrever na memória

11 — (2,0 val) — Considere a seguinte função recursiva implementada na linguagem C:

```
unsigned int faz_mult(unsigned int x, unsigned int y) {
    if (y == 0)
        return 0;
    return x + faz_mult(x, y-1);
}
```

- a) Indique, na folha de respostas, qual das seguintes configurações da pilha (stack) está correta para a chamada `faz_mult(10, 6)`, se pararmos a execução do programa antes da execução da instrução apontada pela seta.



- b) Implemente a função `faz_mult` em assembly IA-32, seguindo as convenções de chamadas de funções da linguagem C:

`faz_mult:`

```
push %ebp
mov %esp, %ebp

mov 12(%ebp), %eax    # y
cmp 0, %eax
je fim

dec %eax              # y-1
push %eax
mov 8(%ebp), %eax     # x
push %eax
call faz_mult
add $8, %esp

add 8(%ebp), %eax     # x + faz_mult(x,y-1)

fim: pop %ebp
ret
```

12 — (2,0 val) — Pretende-se que complete um programa na linguagem C que, dado um vetor de cadeias de caracteres (strings) contendo a representação de números inteiros, converte cada uma dessas cadeias para o inteiro correspondente, criando um vetor de inteiros. O programa principal recebe na linha de comando a sequência de cadeias de caracteres a converter e invoca a função denominada “*convInt*” que implementa essa conversão. Esta função tem a assinatura seguinte:

```
int *convInt(char *cad[], int len);
```

- Recebe como parâmetros de entrada:
 - o endereço inicial de um vetor *cad*; cada posição deste vetor tem assim o endereço de uma cadeia a converter para inteiro;
 - o número de inteiros que estão contidos no vetor de cadeias de caracteres e que é necessário converter
- Retorna o vetor de inteiros criado.

Assumindo que o programa se chama “converter”, o exemplo seguinte ilustra uma execução deste programa:

```
$ ./converter 7 -15 4 7 -2 0 257 1000
v[0]=7 v[1]=-15 v[2]=4 v[3]=7 v[4]=-2 v[5]=0 v[6]=257 v[7]=1000
```

Implemente em C a função “*convInt*” descrita acima e complete o código da função *main* nos locais indicados. Lembre-se que existem as seguintes funções:

```
int atoi( char *str );
void * malloc( int size );
```

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

int *convInt(char *cad[], int len);

extern void printInt(int *v, int len); // assuma que esta função já está implementada

int main(int argc, char *argv[])
{
    int numInt, *vecPtr ;

    if(argc == 1) {
        printf("Utilização: %s <num1> <num2> <num3> ...\\n", argv[0]);
        return 1;
    }
    numInt = argc-1;    // número de elementos a converter
    vecPtr = convInt( &argv[1], numInt);  // &argv[1] é o endereço do primeiro elemento
    if(vecPtr != NULL) {
        printInt(vecPtr, numInt);
        free(vecPtr);    // liberta o espaço reservado dinamicamente
    }
    return 0;
}

int *convInt(char *cad[], int len)  // função a implementar
{

    int * intvec = (int*)malloc( len*sizeof(int) );
    int i;

    if ( intvec != NULL )
        for ( i=0; i<len; i++ )
            intvec[i] = atoi( cad[i] );

    return intvec;

}
```

13 — (1,5 val) — Pretende-se que escreva, em *assembly* Intel 32 bits, usando as mnemónicas e convenções do *as* (assembler), o código de uma subrotina denominada *somasParciais* que implementa as somas parciais de um vetor de inteiros e que tem a assinatura seguinte:

```
void somasParciais( int v[], int resV[], int len );
```

- Recebe como parâmetros de entrada:
 - *v*: o endereço inicial de um vetor *v* com inteiros
 - *resV*: o endereço inicial de um vetor que conterá as somas parciais
 - *len*: o número de inteiros no vetor *v* e no vetor *resV* (sempre maior que zero)
- Os valores de *resV* são preenchidos do seguinte modo:
 - *resV*[0] = *v*[0]
 - *resV*[*i*] = *resV*[*i*-1] + *v*[*i*], para todos os valores de *i* = 1..(*len*-1)

Assuma que esta subrotina é chamada pelo código C seguinte:

```
#define MAX 6

int v[MAX] = {2, 1, 3, 1, 7, 1};
int resV[MAX];

somasParciais(v, resV, MAX);

printf("Vetor original:\n");
printf("%d %d %d %d %d %d\n", v[0], v[1], v[2], v[3], v[4], v[5]);
printf("Somas parciais:\n");
printf("%d %d %d %d %d %d\n", resV[0], resV[1], resV[2], resV[3], resV[4], resV[5]);
```

A execução deste programa deve produzir o seguinte output:

```
Vetor original:
2 1 3 1 7 1
Somas parciais:
2 3 6 7 14 15
```

Implemente em *assembly* a função *somasParciais*.

```
.text
# código da sua solução:

# uma das muitas solucoes possiveis:
.global somasParciais

somasParciais:
    push %ebp
    mov %esp, %ebp
    push %ecx
    push %ebx

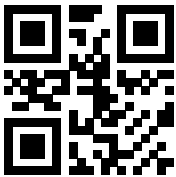
    cmp $0, 16(%ebp)
    je fim

    mov 12(%ebp), %ebx # resV
    mov 8(%ebp), %eax # v
    mov (%eax), %ecx
    mov %ecx, (%ebx) # resV[0] = v[0]
ciclo:
    dec 16(%ebp) # Len = Len-1
    jz fim

    add $4, %eax
    mov (%eax), %ecx
    add (%ebx), %ecx
    add $4, %ebx
    mov %ecx, (%ebx)
    jmp ciclo
fim:
    pop %ebx
    pop %ecx
    pop %ebp
    ret
```

First Name
Last Name
Course Name AC 2016-17: Teste 1
Date
Version

Student Number

[illegible]

Marking Instructions

Completely fill in the appropriate bubble.

Figure 1 illustrates the experimental design, showing two panels (a) and (b) representing different conditions. Each panel displays a sequence of trials, numbered 1-a) to 11-a) and 1-b) to 11-b). The trials are organized into two columns, (a) and (b), with each trial consisting of a stimulus (A, B, C, D, E) and a response option (A, B, C, D, E). The response options are presented in a grid format, with the correct response highlighted in a darker shade. The trials are arranged in two columns, (a) and (b), with each trial consisting of a stimulus (A, B, C, D, E) and a response option (A, B, C, D, E). The response options are presented in a grid format, with the correct response highlighted in a darker shade.

Intel x86 (IA32) Assembly Language Cheat Sheet

Suffixes: b=byte (8 bits); w=word (16 bits); l=long (32 bits). Optional if instruction is unambiguous.

Operands: immediate/constant (not as *dest*): \$10, \$0xff ou \$0b01101 (decimal, hex or bin)

32-bit registers: %eax, %ebx, %ecx, %edx, %esi, %edi, %esp, %ebp

16-bit registers: %ax, %bx, %cx, %dx, %si, %di, %sp, %bp

8-bit registers: %al, %ah, %bl, %bh, %cl, %ch, %dl, %dh

direct addr: (2000) or (0x1000+53)

indirect addr: (%eax) or 16(%esp) or 200(%edx, %ecx, 4)

Note that it is not possible for **both** *src* and *dest* to be memory addresses.

Instruction	Effect	Examples
Copying Data		
mov <i>src,dest</i>	Copy <i>src</i> to <i>dest</i>	mov \$10,%eax movw %ax,(2000)
Arithmetic		
add <i>src,dest</i>	dest = dest + <i>src</i>	add \$10, %esi
sub <i>src,dest</i>	dest = dest - <i>src</i>	sub %eax,%ebx
cmp <i>src,dest</i>	Compare using sub (<i>dest</i> is not changed)	cmp \$0,%eax
inc <i>dest</i>	Increment destination	inc %eax
dec <i>dest</i>	Decrement destination	decl (0x1000)
Bitwise and Logic Operations		
and <i>src,dest</i>	dest = <i>src</i> & <i>dest</i>	and %ebx, %eax
test <i>src,dest</i>	Test bits using and (<i>dest</i> is not changed)	test \$0xffff,%eax
or <i>src,dest</i>	dest = <i>src</i> <i>dest</i>	or (0x2000),%eax
xor <i>src,dest</i>	dest = <i>src</i> ^ <i>dest</i>	xor \$0xffffffff,%ebx
shl <i>count,dest</i>	dest = dest << <i>count</i>	shl \$2,%eax
shr <i>count,dest</i>	dest = dest >> <i>count</i>	shr \$4,(%eax)
sar <i>count,dest</i>	dest = dest >> <i>count</i> (preserving signal)	sar \$4,(%eax)
Jumps		
je/jz <i>label</i>	Jump to <i>label</i> if <i>dest</i> == <i>src</i> /result is zero	je endloop
jne/jnz <i>label</i>	Jump to <i>label</i> if <i>dest</i> != <i>src</i> /result not zero	jne loopstart
jg <i>label</i>	Jump to <i>label</i> if <i>dest</i> > <i>src</i>	jg exit
jge <i>label</i>	Jump to <i>label</i> if <i>dest</i> >= <i>src</i>	jge format_disk
jl <i>label</i>	Jump to <i>label</i> if <i>dest</i> < <i>src</i>	jl error
jle <i>label</i>	Jump to <i>label</i> if <i>dest</i> <= <i>src</i>	jle finish
ja <i>label</i>	Jump to <i>label</i> if <i>dest</i> > <i>src</i> (unsigned)	ja exit
jae <i>label</i>	Jump to <i>label</i> if <i>dest</i> >= <i>src</i> (unsigned)	jae format_disk
jb <i>label</i>	Jump to <i>label</i> if <i>dest</i> < <i>src</i> (unsigned)	jb error
jbe <i>label</i>	Jump to <i>label</i> if <i>dest</i> <= <i>src</i> (unsigned)	jbe finish
jz/je <i>label</i>	Jump to <i>label</i> if all bits zero	jz looparound
jnz/jne <i>label</i>	Jump to <i>label</i> if result not zero	jnz error
jmp <i>label</i>	Unconditional jump	jmp exit
Function Calls / Stack		
call <i>label</i>	Call (Push eip and Jump)	call format_disk
ret	Return to caller (Pop eip and Jump)	ret
push <i>src</i>	Push item to stack	pushl \$32
pop <i>dest</i>	Pop item from stack	pop %eax

Directives (examples):

.data – data section (global variables)

.text – text section (code)

.int – 32bits space(s) for integer value(s)

.comm *label, length* – length bytes space

.ascii – char sequence

.global *label* -- export *label* symbol/address

Functions Linux/32bits:

caller:

- push args (right to left)
- call function
- free stack space used with args

C types:

char 1 byte

short 2 bytes

int, float, long and *pointer* 4 bytes

double 8 bytes

callee (function):

- initialise: push %ebp
mov %esp, %ebp
sub \$4, %esp #space for local var.
- use ebp based address, e.g.: movl 8(%ebp), %eax
- result at %eax
- finalise: mov %ebp, %esp #free local var.
pop %ebp
ret

Folha para rascunho.
Pode destacar depois do início do teste.