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

4 — (1,5 val) — Sabendo que na representação de números com sinal o bit mais significativo representa o sinal (1=negativo; 0=positivo), qual das seguintes expressões na linguagem C ou Java, permite "extrair" o bit de sinal de um inteiro de 2 bytes (16 bits) guardado na variável 'num'? O resultado da expressão deverá ser "0" se o inteiro for positivo e "1" se for negativo.

A) (num >> 1) & 0x1 B) (num >> 16) & 0x1 C) (num & 0x8000) >> 1

D) num % 2 E) (num >> 15) & 0x1

5 — (1,5 val) — Se executarmos o seguinte programa em C

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

qual dos seguintes resultados obtemos e porquê?

A) `x=0` — Porque os valores usados na expressão (2 e 10) são inteiros.

B) `x=0.2` — Porque a variável x é do tipo float.

C) `x=0.200000` — Porque a variável x é do tipo float, e a string de formatação especifica um float (%f).

D) `x=0.000000` — Porque os valores usados na expressão (2 e 10) são inteiros e a string de formatação especifica um float (%f).

E) `x= 0.20000000000000001110` — Porque a variável x é do tipo float, a string de formatação especifica um float (%f) e o valor é apenas aproximado pois não é representável de forma exata na norma IEEE754.

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:

A) Código C > compilador > código máquina > assembler > código assembly > linker > executável

B) Código C > compilador > código assembly > assembler > código máquina > linker > executável

C) Código C > compilador > código assembly > linker > código assembly > assembler > executável

D) Código assembly > assembler > código C > compilador > código máquina > linker > executável

E) Código assembly > compilador > código C > assembler > código máquina > linker > executável

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?

```
conjunto1: mov (var1), %ebx
           mov $0, (var2)
           add %ebx, (var2)
           add $20, (var2)
```

```
conjunto2: mov (var1), %ebx
           mov $0, %ecx
           add %ebx %ecx
           add $20, %ecx
           mov %ecx, (var2)
```

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***
0x200	2
0x201	0
0x202	3
0x203	4
0x204	0
0x205	3
0x206	2
0x207	2
0x208	0
0x209	0

Programa

```
movl $0x206, %eax
movl (%eax), %ebx
movb -2(%ebx), %cl
movl -2(%ebx, %ecx, 2), %edx
```

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=0x00000006 ecx=*`

B) `eax=0x00000005 ebx=0x00000006 ecx=0x00000005`

C) `eax=0x00000005 ebx=0x00000006 ecx=0x00000006`

D) `eax=0x00000006 ebx=0x00000006 ecx=0x00000005`

E) `eax=0x00000005 ebx=0x00000005 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

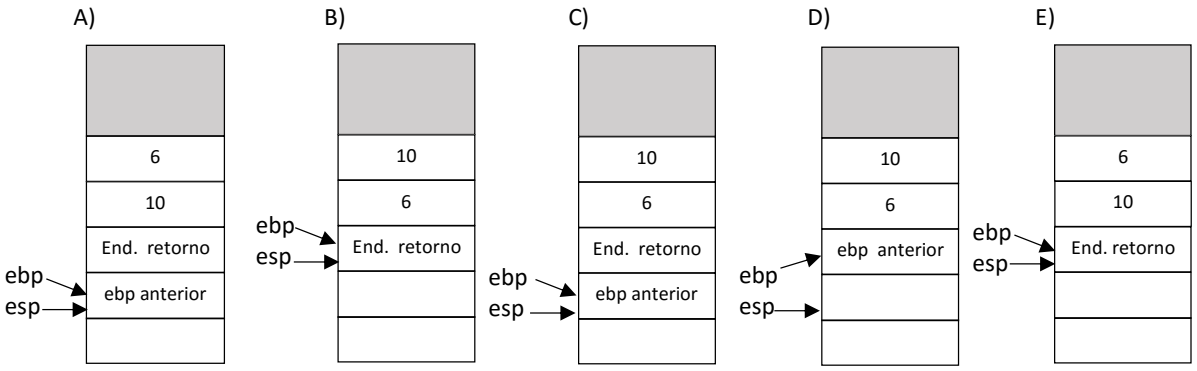
D) O CPU coloca no bus de endereços o valor resultante da soma de 100 com EBX, no bus de dados o conteúdo do registo EAX e no bus de controlo o comando para escrever na memória

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:
```

12 — (2,0 val) — Pretende-se que complete um programa na linguagem C que, dado um vetor de cadeias de carateres (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
{









}
```

13 — (1,5 val) — Pretende-se que escreva, em *assembly* Intel 32 bits, usando as mnemónicas e convenções do *as* (**as**sembler), 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:
```

Teste 1 - AC 2016-17

AKINDI

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

Student Number

0.	0	0	0	0	0	0	0	0	0
1.	1	1	1	1	1	1	1	1	1
2.	2	2	2	2	2	2	2	2	2
3.	3	3	3	3	3	3	3	3	3
4.	4	4	4	4	4	4	4	4	4
5.	5	5	5	5	5	5	5	5	5
6.	6	6	6	6	6	6	6	6	6
7.	7	7	7	7	7	7	7	7	7
8.	8	8	8	8	8	8	8	8	8
9.	9	9	9	9	9	9	9	9	9

Marking Instructions
Completely fill in the appropriate bubble.

Correct	Incorrect
●	⊗ ✓ ◔ ◉

	A	B	C	D	E	A	B	C	D	E
1-a)										
1-b)										
1-c)										
1-d)										
2-a)										
2-b)										
2-c)										
3.										
4.										
5.										
6.										
7.										
8.										
9.										
10.										
11–a).										

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

Directives (examples):

.data – data section (global variables)

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

.ascii – char sequence

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

.text – text section (code)

.comm *label*, *length* – length bytes space

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.