Arquitectura de Computadores

Licenciatura em Engenharia Informática

1º teste (D) – 2008/04/23 – Duração: 1h30m + 30m

Teste sem consulta.
A interpretação do enunciado faz parte da avaliação. Explicite nas suas respostas todas as hipóteses assumidas.
Por favor, tente focalizar a sua respostas para que estas se enquadrem na zona delimitada.

1 Teórica - Escolha Múltipla

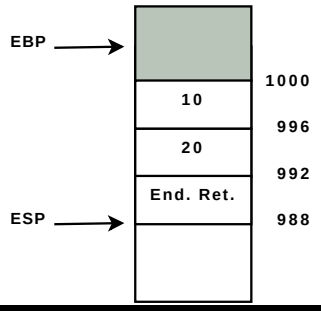
Deve assinalar com um **X** a resposta correcta. Cada resposta errada desconta 25% da cotação da pergunta.
Para todas estas questões considere que está no contexto de um computador Intel com o ISA IA-32 e que o assembly utilizado é o NASM.

Q-1 [1.0 val.] Considere a definição da seguinte variável: `a: resb 32`
Qual das seguintes sequências de instruções coloca o conteúdo do registo `cl` na última posição do array `a`.

1. `mov ebx, a` `mov esi, 31` `mov [ebx+esi*2], cl`
2. `mov ebx, [a]` `mov esi, 31` `mov [ebx+esi*2], cl`
3. `mov ebx, a` `mov esi, 31` `mov [ebx+esi*4], cl`
4. `mov ebx, [a]` `mov esi, 31` `mov [ebx+esi*4], cl`
5. Nenhuma das anteriores

Q-2 [1.0 val.] Considere que está a executar uma subrotina com o ambiente de execução ilustrado na figura à direita. Indique que sequência de instruções é a correcta para terminar a execução da subrotina, de forma a voltar para o ponto de onde foi chamada e retornar o valor 10, de acordo com a convenção adoptada para o retorno de resultados.

1. `mov eax, 10` `pop ebp` `ret`
2. `mov eax, 10` `pop esp` `ret`
3. `pop ebp` `ret 10`
4. `mov eax, 10` `ret`
5. Nenhuma das anteriores



Q-3 [1.0 val.] Qual é o comportamento do seguinte código C?

```
char a[] = ``hello'', *p;
for (p = a; *p != '\0'; p++)
   *p = (*p) + 1;
```

1. O ciclo executa normalmente, colocando em cada posição do array `a` o símbolo que, na tabela ASCII, sucede ao guardado nessa posição, por exemplo se a[i] = 'a' então o novo valor de a[i] será 'b'
2. O ciclo executa normalmente, colocando em cada posição do array `a` o valor do elemento seguinte, por exemplo coloca em a[1] o valor de a[2]
3. O ciclo corre indefinidamente, pois a condição de paragem não está bem definida, devia ser `p != '\0'`
4. A condição inicial (`p = a`) está incorrecta, devia ser `p = &a`
5. Nenhuma das anteriores

Q-4 [1.0 val.] No contexto da unidade de vírgula flutuante

1. Os registos ST0 a ST7 têm 80 bits
2. O topo da pilha de operações é guardado no registo FLAGS
3. Existem instruções da família **`jump`** que testam o conteúdo do registo FSW
4. O estado da última operação efectuada é guardado no registo FLAGS
5. Nenhuma das anteriores

Q-5 [1.0 val.] Considere o seguinte programa C guardado num ficheiro nome `prog.c`.

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

Qual é o resultado da seguinte linha de compilação: **`gcc`** `prog.c` **`-o`** `prog`

1. Um erro de compilação porque o código da função `f` não está disponível
2. Um erro de compilação porque o protótipo da função `f` não está definido
3. Um erro de ligação ("*linkagem*") porque o código da função `f` não está disponível
4. Um erro de compilação porque a utilização da função `f` não está de acordo com o seu protótipo
5. Nenhuma das anteriores

Q-6 [1.0 val.] Dadas as seguintes variaveis (à esquerda) diga quais das instruções (à direita) são correctas quanto ao tipo de dados utilizado:

```
a:      dd 321
b:      db 57
c:      dd 1.0
```

```
a) mov ah, [a]
b) mov dword [a], 0
c) mov ebx, [b]
d) fild dword [a]
e) fstp qword [c]
```

1. b), c) e e)
2. a) e d)
3. c), d) e e)
4. b) e d)
5. Nenhuma das anteriores

Q-7 [1.0 val.] Um endereço de memória:

1. tem sempre 32 bits
2. pode ter 8, 16 ou 32 bits
3. pode ter 16 ou 32 bits
4. tem se ser sempre guardado em registos específicos, como o EIP
5. Nenhuma das anteriores

2 Teórica - Desenvolvimento

Q-8 [1.0 val.] Diga qual é a diferença entre as instruções **`fild`** e **`fld`**. Dê um exemplo de uma variável inicializada com um valor a ser carregado pelo **`fild`** e outra pelo **`fld`**, apresentando as instruções correspondentes para carregar esses mesmos valores no registo ST0.
A instrução **`fld`** carrega o conteúdo de uma posição de memória para o topo da pilha de operações da FPU (o registo ST0). Esta instrução deve ser utilizada quando o conteúdo da posição em questão representa um valor em vírgula flutuante na norma IEEE-754.

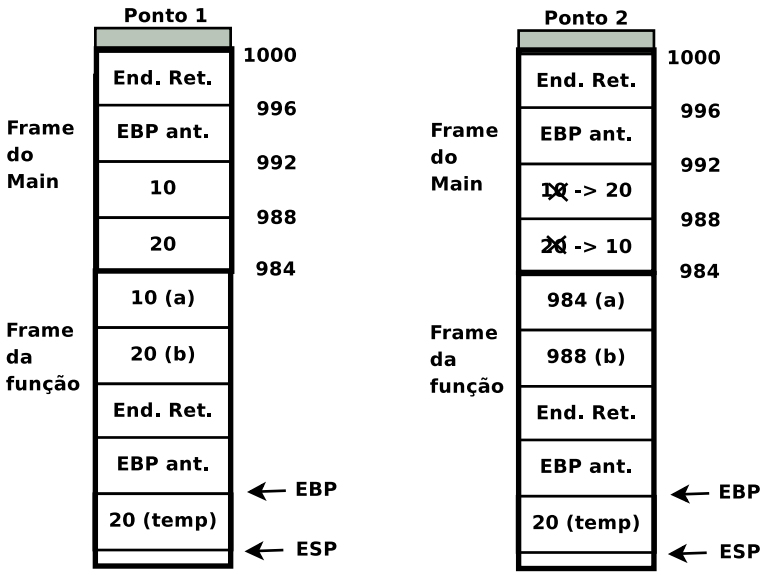
A instrução **`fild`** faz a mesma operação de carregamento. No entanto, assume que o conteúdo da posição a carregar representa um inteiro com sinal. Assim sendo, a operação inclui uma conversão da representação inteira para a norma IEEE-754.
Exemplos:

```
section .data
    ...
    a: dd 1
    b: dd 2.0
    ...
```

```
section .text
    ...
    fild dword [a]
    fld dword [b]
    ...
```

Q-9 [2.0 val.] Considere o seguinte código C e assuma que pára a execução nos pontos 1 e 2 assinalados. Complete as figuras à direita, escrevendo qual é o conteúdo da pilha de execução (*stack*) em cada um dos pontos. Deve também indicar para onde apontam, nessa altura, os registos ESP e EBP.

```
void troca1(int a, int b) {
    int temp = a;
    a = b;
    b = temp; ⟵ 1
}
void troca2(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp; ⟵ 2
}
int main() {
    int x = 10, y = 20;
    troca1(y, x);
    troca2(&y, &x);
    return 0;
}
```



Q-10 [1.00 val.] Explique o porquê de se usar instruçõees diferentes para programar saltos condicionais sobre inteiros com e sem sinal. Por exemplo, **`jg`** e **`ja`**, respectivamente.

Inteiros com e sem sinal são codificados de forma diferente. Os primeiros são usam a norma do complemento para 2, enquanto os segundos são codificados pela simples representação do número em binário. Claro que em ambos os casos temos que a representação tem de ser adequada ao número de bits disponíveis para a representação.

Estas diferentes representações podem dar a um padrão de bits significados diferentes, consoante a sua interpretação. Exemplos numa representação a 8 bits:

- 0000 0001 interpretado como um número com ou sem sinal denota o número 1
- 1111 1111 interpretado como um número sem sinal denota o número 255 e com sinal o número -1

Numa interpretação de números sem sinal 0000 0001 = 1 é menor que 1111 1111 = 255. No entanto, numa interpretação de números com sinal temos o contrário, 0000 0001 = 1 é maior que 1111 1111 = -1. Temos de ter então instruções distintas que interpretem os padrões de forma adequada.

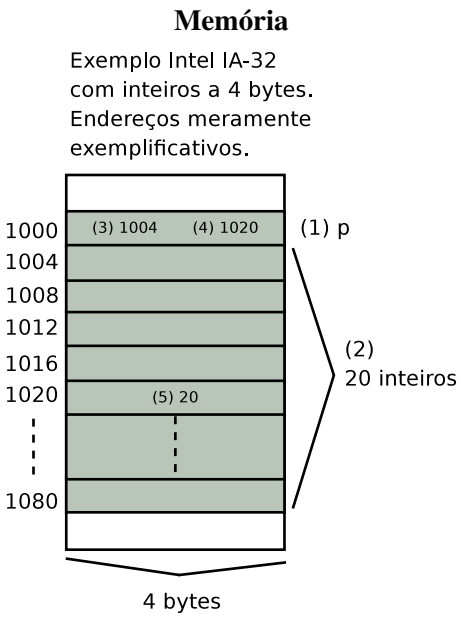
Nos processadores Intel esta interpretação consiste em analisar flags diferentes no registo EFLAGS que guarda o estado da última operação efectuada pela ALU, ou seja, da última operação lógica ou aritmética. As instruções da família dos saltos condicionais consultam este registo para avaliarem se o salto programado deve ou não ser efectuado. A flag, ou conjunto de flags a analisar, depende da instrução. Por exemplo, a instrução **`jz`** analisa o valor da flag ZF e salta para a etiqueta destino caso o seu valor seja 1.

Os saltos disponíveis para comparação de inteiros sem sinal consultam as flags CF e ZF (**`ja`**, **`jb`**, **`je`** e variantes). Os saltos disponíveis para comparação de inteiros com sinal consultam as flags OF, SF e ZF (**`jg`**, **`jl`**, **`je`** e variantes). A único salto que é comum é o **`je`** que verifica se ZF=1, ou seja, se o resultado da última

operação foi 0. Uma vez que a instrução **cmp** subtrai os dois operandos (sem guardar o resultado), caso estes sejam iguais o resultado será 0. Este facto é válido para qualquer representação.

Q-11 [2.0 val.] Considere a seguinte sequência de acções em linguagem C e as sua numeração. Faça um esquema que ilustre cada acção numerada e use essa mesma numeração como legenda para a explicação.

```
(1)  int *p;
     p = (int *) malloc (sizeof(int) * 20));
      (3)                (2)
(4)  p = p + 4;
(5)  *p = 20;
```



(1) Declaração de uma variável p que é um apontador para inteiro, ou seja, o seu conteúdo é interpretado como sendo o endereço de uma zona de memória onde está guardado um valor inteiro.
(2) Reserva (alocação) de uma zona de memória contígua com o tamanho de 20 inteiros. No gcc onde um inteiro são 4 bytes, seriam reservados 80 bytes. No TurboC onde um inteiro são 2 bytes seriam apenas reservados 40.
(3) Atribuir a p o endereço do início da zona de memória reservada (1004 no exemplo).
(4) Incrementar p em 4 vezes o tamanho do seu tipo, neste caso 4 inteiros. Portanto o valor no gcc o valor de p seria incrementado em 16. No exemplo p fica com o valor 1020.
(5) Alterar o conteúdo da posição de memória apontada por p. Neste caso o novo valor será 20.

3 Prática - Assembly

Q-12 Esta pergunta **tem vários níveis de dificuldade**:
Para ter a cotação máxima (4 valores) deve usar as convenções de passagem de parametros e retorno de valores dada nas teóricas. Se usar qualquer outro método (à sua escolha) para passar parametros/retornar resultados, a pergunta será cotada para 80% (3.2 valores). Se não quiser usar funções pode responder tudo na alínea b), com um só programa equivalente ao que é pedido. Nesse caso terá no máximo a cotação da alínea b) (2.5 valores).
a) [1.5 val.] Programe em NASM IA/32, uma função `min`, que dados dois valores, **inteiros de 32 bits com sinal**, devolva o menor dos dois.

Solucao 1 - 100%

```
min:
    push ebp
    mov ebp, esp
    mov eax, [ebp+8]    ; [ebp+8] - primeiro parametro
    cmp eax, [ebp+12]   ; [ebp+12] - segundo parametro
    jl fim_min          ; jl -> numeros com sinal
    mov eax, [ebp+12]
fim_min:
    pop ebp
    ret 8               ; libertar 2 parametros 32 bits -> 8 bytes
```

---

solucao 2 - 80%

```
min:
    cmp eax, ebx        ; eax - 1o parametro; ebx 2o parametro
    jl fim_min
    mov eax, ebx
fim_min:
    ret                 ; resultado em eax
```

b) [2.5 val.] Dado o seguinte programa (em pseudo código)

```
int v[21];
int m;
m = v[0];
for( int i = 0; i < 21; i++ )
 m = min(v[i],m);
```

apresente o pedaço de programa em NASM IA/32 correspondente a este código, incluindo a declaração das variaveis. Deve usar a função `min` definida na alinea anterior.

Solucao 1-100%

```
...
section .bss
m:    resd 1            ; variáveis de 32 bits não inicializadas
v:    resd 21
; i não está declarada porque é uma variável local interna ao ciclo
; usamos o registo esi para guardar i

section .text
_start:
    mov eax, [v]        ; são precisas 2 instrucões; não há mov
    mov [m], eax        ; de memória para memória
                        ; [v] = [v+0] - primeiro elemento do vector v
    mov esi, 0          ; inicializar i
ciclo:
    cmp esi, 21
    jae fim_ciclo       ; ou jnb ou jnc que são equivalentes
                        ; estamos a considerar que a variável i
                        ; corresponde a um número sem sinal
    push dword [m]        ; passar parametros à função pela ordem inversa
    push dword [v+esi*4]  ; usar acesso indexado escalar ao vector
    call min
    mov [m],eax         ; buscar resultado e guardar em m
    inc esi
    jmp ciclo
fim_ciclo:
    mov eax, SYS_EXIT
    mov ebx, 0
    int LINUX_SYSCALL
```

---

solucao 2-80%

```
...
section .bss
m:    resd 1
v:    resd 21

section .text
_start:
    mov eax, [v]
    mov [m], eax
    mov esi, 0
ciclo:
    cmp esi, 21
    jae fim_ciclo
    mov eax, [v+esi*4]  ; passar parametros à função
```

```
    mov ebx, [m]
    call min
    mov [m],eax         ; buscar resultado e guardar em m
    inc esi
    jmp ciclo
fim_ciclo:
    mov eax, SYS_EXIT
    mov ebx, 0
    int LINUX_SYSCALL
```

Solucao 3 - 2.5 val

```
...
section .bss
m:    resd 1
v:    resd 21

section .text
_start:
    mov eax, [v]
    mov [m], eax
    mov esi, 0
ciclo:
    cmp esi, 21
    jae fim_ciclo
    mov eax, [m]
    cmp eax, [v+esi*4]
    jl menor
    mov eax, [v+esi*4]
    mov [m],eax
menor:
    inc esi
    jmp ciclo
fim_ciclo:
    ; nota - podiamos usar o eax como m durante o programa
    ; evitando os acessos à variável m e poupando 3 linhas de código.
    ; nesse caso, era preciso fazer, neste ponto:
    ; mov [m], eax
    mov eax, SYS_EXIT
    mov ebx, 0
    int LINUX_SYSCALL
```

4 Prática - Linguagem C

Q-13 Os espaços existentes no programa principal em C abaixo indicado deverão ser preenchidos de forma a definir uma matriz quadrada, de 8x8 inteiros que, depois de executada a função `preenche`, fique com os seguintes valores (onde `v` é um valor a "ler do teclado"):

$$\begin{vmatrix} v & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ v & v & 0 & 0 & 0 & 0 & 0 & 0 \\ v & v & v & 0 & 0 & 0 & 0 & 0 \\ v & v & v & v & 0 & 0 & 0 & 0 \\ v & v & v & v & v & 0 & 0 & 0 \\ v & v & v & v & v & v & 0 & 0 \\ v & v & v & v & v & v & v & 0 \\ v & v & v & v & v & v & v & v \end{vmatrix}$$

a) [1.0 val.] O programa abaixo, "lê v do teclado", chama a função `preenche` e depois, imprime o somatório dos elementos da matriz. Preencha as caixas de forma a criar um programa correcto, obedecendo às especificações dadas.

```
void preenche(int matriz[8][8], int dim, int v);
int somatorio(int matriz[8][8], int dim);

int main() {
  int matriz[8][8]; // complete a definição da variável matriz
  int i, j, v; // e outra(s) que considere necessária(s)

  scanf("%d", &v); // coloque aqui a instrução para "ler v do teclado"
  preenche(matriz, 8, v); // coloque aqui os argumentos da função
  printf("Somatório: %d \n", somatorio(matriz, 8)); // coloque os itens em falta
  return 0;
}
```

Nota: Na função `preenche` o parâmetro `dim` que indica a dimensão da matriz, e `v` o valor a passar para o seu preenchimento.

b) [2.0 val.] Agora, implemente a função `preenche` para que esta inicialize a matriz da forma indicada. **Nota: Não tem de usar apontadores.**

```
void preenche(int x[8][8], int dim, int v)
  int i, j;

  for (i=0; i < dim; i++)
    for (j=0; j < dim; j++)
      x[i][j] = ( i <= j ? v : 0);
    /* ou, um if que é equivalente:
      if ( i <= j )
         x[i][j] = v;
      else
         x[i][j] = 0;
    */
```