Arquitectura de Computadores

Licenciatura em Engenharia Informática

Exame de Época Normal (A) – 2007/06/25 – Duração: 2h00m + 15m tolerância

Nome:_____ Número:_____

Total de páginas: 6+___ *páginas* Classificação:_____

Q-1 Considere o ISA IA-32 utilizado nas arquitecturas Intel 80x86.

a) [0.75 val.] Estamos perante uma arquitectura do tipo CISC ou RISC? Justifique.
As arquitecturas Intel são do tipo CISC, oferendo um conjunto complexo de operações que podem operar sobre dados de tamanhos diversos e utilizar diferentes modos de endereçamento, como por exemplo, endereçamento imediato, directo e indirecto por registo numa operação de adição

b) [0.75 val.] Comente a afirmação: *Os modos de endereçamento oferecidos por uma dada arquitectura não contribuem para a complexidade do conjunto de instruções.*
Os modos de endereçamento contribuem para a complexidade do conjunto de instruções. Uma adição que opera sobre dois registos é diferente de uma adição que opera sobre duas posições de memória. Na realidade cada uma constitui uma instrução máquina diferente. Assim sendo, quantos mais modos de endereçamento forem oferecidos pelo ISA mais instruções máquinas serão necessárias e, logo, mais complexo será o conjunto de instruções.

c) [0.25 val.] Coloque a seguinte palavra de 32 bits (0F AE 33 3F) no endereço 0FFF da memória abaixo. Escreva o novo valor na coluna *Conteúdo Actualizado*.

Endereço	Conteúdo	Conteúdo Actualizado
00000FFC:	2F	_____
00000FFD:	10	_____
00000FFE:	04	_____
00000FFF:	FF	3F
00001000:	FC	33
00001001:	00	AE
00001002:	10	0F
00001003:	A3	_____
00001004:	01	_____

d) [0.75 val.] Considerando a mesma zona de memória, diga qual é o resultado das seguintes operações. *Considere que todas as posições de memória não indicadas na tabela acima contêm o valor zero.*

USANDO O CONTEÚDO ORIGINAL.

`mov ax, [1000H]`	ax = 00FC	Lembre-se que o H denota que o número está em hexadecimal.
`mov ax, 1000`	ax = 1000	
`mov ax, bx`	ax = 0FFC	O conteúdo de bx é OFFCH.
`mov ax, [ebx]`	ax = 102F	O conteúdo de ebx é OFFCH.
`mov ax, {1001H}`	ax = 00FC	Suponha que a notação {} denota endereçamento indirecto por memória. Algo que não existe neste assembly.

Q-2 [1.00 val.] Explique a diferença entre o shift lógico e o shift aritmético.
O shift lógico opera sobre números sem sinal e o aritmético sobre números com sinal. Isto resulta que num deslocamento para a esquerda as operações são iguais. Num deslocamento aritmético para a direita, é necessário ter em atenção o sinal. Assim sendo, não se colocam 0s mas sim o valor do bit de sinal. Exemplo:
Shift lógico de 3 posições sobre 10011010 tem como resultado: 00010011e fica 0 (o último a sair) na carry flag.
Shift aritmético de 3 posições sobre 10011010 tem como resultado: 11110011e fica 0 (o último a sair) na carry flag.

Q-3 [1.00 val.] Escreva o código NASM/Intel (IA-32) que efectua a chamada à função `count` ilustrada no seguinte código C: `n = count('a')`. O protótipo da função é: `int count(char)`.

```
push dword 'a'
call count
mov [n], eax
add esp, 4
```

ou

```
push word 'a'
call count
mov [n], eax
add esp, 2
```

Q-4 [1.25 val.] Explique porque é que se usa a pilha para passar os valores para os parâmetros das subrotinas.
A pilha permite, de uma forma simples, que se tenha acesso à frame de activação da subrotina a executar (que contém os valores do parâmetros da subrotina). O uso de registos ou posições pré-definidas não escala de forma satisfatória. Os registos não são muitos e ambas as abordagens inviabilizam o uso de recursão.
Com o uso da pilha, temos à disposição uma zona de memória dedicada que pode comportar muitos dados (claro que não é infinita), onde as frames de activação podem ser empilhadas. Estas frames são independentes entre elas e cada uma tem os dados necessários para a sua execução (os argumentos, o endereço de retorno e as variáveis locais), assim como a localização da base da frame anterior (o valor anterior do EBP). Logo, podemos empilhar várias frames relativas à execução da mesma função e assim implementar a recursividade.
Quando uma rotina termina basta retirar os dados da sua frame que determinam o próximo valor do PC (o endereço de retorno) e o valor anterior do EBP, para que possamos continuar a execução do programa na rotina anterior.

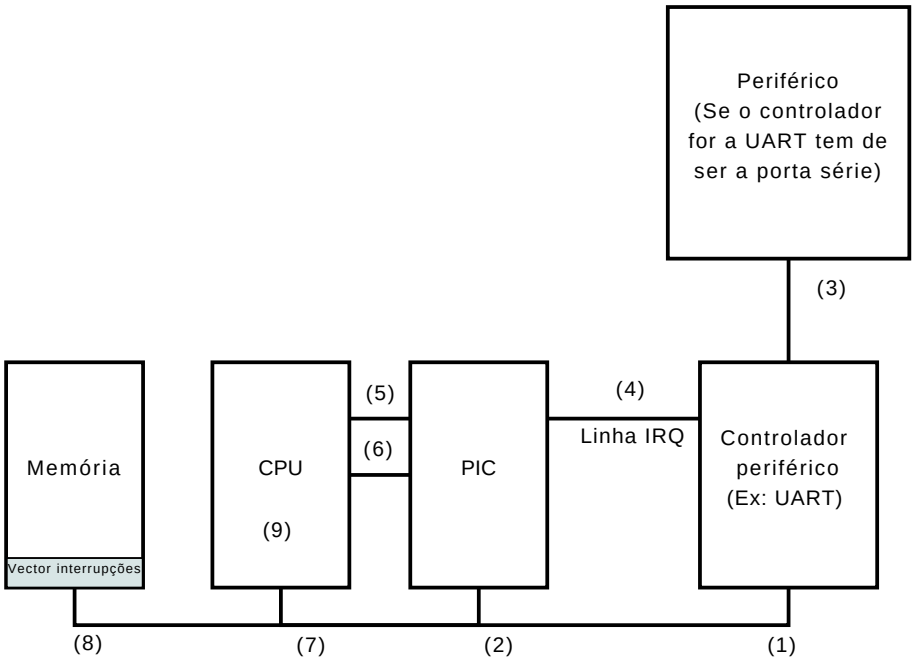
Q-5 [1.00 val.] Construa a *codeword* com o código de Hamming para a seguinte palavra de memória (*word*):

0010 1110 0000 1011

Note que tendo a palavra 16 bits são precisos pelo menos 5 bits de paridade. **Nota:** os bits podem ser numerados da esquerda para a direita (como no livro), ou da direita para a esquerda (como no acetatos). Indique a sua escolha.
VER LIVRO

Q-6 [1.00 val.] Explique o conceito de volume (de discos) RAID e quais as suas vantagens.
VER LIVRO

Q-7 [2.00 val.] Faça um esquema que ilustre o funcionamento de um periférico que usa interrupções - por exemplo, recepção de dados numa UART. Anote o esquema com a enumeração dos passos necessários à recepção dos dados e use essa enumeração para elaborar a legenda do esquema.



Legenda: (1) Configuarar o controlador do periférico para gerar uma interrupção sempre que receba um novo byte.
(2) Configurar o PIC para aceitar as interrupções provenientes da linha de IRQ onde está ligado o controlador do periférico.
(3) O periférico recebe um novo byte que é guardado no buffer do seu controlador.
(4) O controlador envia uma interrupção ao PIC para alertar o CPU da chegada do byte.
(5) O PIC aceita a interrupção (não nos procupemos com o esquema de prioridades para simplificar o processo) e informa o CPU.
(6) O CPU informa o PIC que está disponível para atender a interrupção.
(7) O indíce no vector de interrupções é lido do PIC pelo CPU
(8) O CPU usa esse índice para aceder, na memória, ao vector de interrupções e descobrir a localização do código a executar. (9) O CPU executa esse código (a rotina de atendimento da interrupção).

Q-8 Admita uma arquitectura de um computador com as seguintes características: endereçamento de 24 bits, cache com mapeamento directo de 2 MBytes, onde cada linha tem 128 Bytes.

a) [0.50 val.] Qual é o tamanho máximo para o espaço de endereçamento de um processo?
2^{24}

b) [0.50 val.] Indique para o endereço seguinte qual é a chave do bloco de memória e qual o byte referenciado dentro desse bloco:

0110 1111 0000 0001 0000 1011

chave: 011 linha: 0 1111 0000 0001 0 byte dentro do bloco: 000 1011

c) [0.75 val.] Suponha o seguinte cenário:

- a cache está vazia;
- a cache usa uma política de *write-through*;
- o acesso que se está a efectuar é de escrita.

Explique que passos são efectuados até que os bytes sejam escritos nos endereços pretendidos.
A cache está vazia logo temos um miss. Passamos recursivamente ao nível seguinte até encontramos o bloco pretendido. Uma vez encontrado este é trazido para o nível desejado e actualizado. Esta actualização é propagada para o nível seguinte e para os outros, consoante a sua política de escrita.

d) [0.50 val.] Considere que a cache tem um tempo de acesso de 5 ns, que memória central tem um tempo de acesso de 30 ns e que a taxa de sucesso (*hit ratio*) no acesso à cache é de 80%. Indique o tempo médio de acesso à memória.
$TA = 0.8 * 5 + -0.2 * 30 = 4 + 6 = 10ns$

e) [0.50 val.] Considere ainda que em média 20% do processamento é passado em acessos a memória. Calcule qual é o impacto da cache da alínea anterior na performance global do sistema.
$S = \frac{1}{(1-p)+(p/s)}$ VER Lei de Amdahl.
s = 30/10 tempo de acesso à memória antes e depois da introdução da cache. Temos portanto um speedup de 3.
Aplicado a lei de Amdahl temos que $S = \frac{1}{(1-0.2)+(0.2/3)} = \frac{1}{0.8+0.07} = 1.15$

Q-9 [1.00 val.] Diga o que entende por, e em que cenários se recorre à técnica de *branch prediction*.
VER LIVRO

Q-10 Para a função que se segue, em assembly, poderá fazer uso das instruções que desejar; contudo, fica aqui uma sugestão: as instruções `shl` ou `shr` (shift left ou right), deslocam o registo alvo para a esquerda ou direita, colocando o bit que "salta fora"no carry bit.

a) [de 1.00 a 2.00 val.] Implemente em assembly uma função, que vamos designar por `func`, que recebe como parâmetro um byte e devolve como resultado (em EAX) um inteiro que indica o número de bits a 1 encontrado no byte (Exemplo: se o byte passado como argumento for 01010111, devolve 5).

- **[1.00 val.]** O argumento é passado para a função num registo à sua escolha.
- **[1.50 val.]** O argumento é passado para a função na pilha, e acedido na função via ESP.
- **X [2.00 val.]** O argumento é passado para a função na pilha, e acedido na função via EBP.

```
func:
    push ebp
    mov ebp, esp
    mov ebx, [ebp+8]
    mov ecx, 8
ciclo:
    shr ebx, 1
    jnc salto
    inc eax
salto:
    loop ciclo
    pop ebp
    ret
```

b) [2.00 val.] Implemente em assembly uma função, de nome `setParity`, que tem como parâmetros um byte e um inteiro. Para simplificar a leitura do enunciado denotemos esses parâmetros por `byte` e `parmode`.

`setParity` deve recorrer à função `func` da alínea anterior para determinar o número de bits a 1 de `byte` e deve usar esse valor, mais o de `parmode`, para definir o valor do bit mais significativo de `byte` (a paridade).

- Se `parmode` = 0 e `func(byte)` é par então o bit mais significativo de `byte` será 0.
- Se `parmode` = 0 e `func(byte)` é ímpar então o bit mais significativo de `byte` será 1.
- Se `parmode` = 1 então o bit mais significativo de `byte` será 0.

- Se `parmode` = 2 então o bit mais significativo de `byte` será 1.

Sugestão: Considere a divisão com sinal e use a instrução IDIV. Esta instrução recebe apenas um operando src, realizando a seguinte operação $EAX \leftarrow EDX : EAX/src$, sendo que em EDX fica o resto da divisão.

Nota: Para simplificar a resolução usou-se sempre valores de 32 bits. Tal não era necessário, pois a pilha também aceita valores em 16 bits, que eram suficientes para conter byte e parmode. Nesse caso, usaria-seos registos bx e cx, e o código seria ligeiramente diferente.

```
setParity:
    push ebp
    mov ebp, esp
    mov ebx, [ebp+12]    ;  byte
    mov ecx, [ebp+8]     ;  parmode
    mov edx, 0
    cmp ecx, 0
    jne parmode_eq_one
    push ebx
    call func
    add esp, 4
    push dword 2        ; usar a pilha para guardar o valor local 2
    idiv dword [ebp-4]
    add esp, 4
    mov ebx, [ebp+12]   ;  repor byte em ebx, pois este ultimo foi alterado em func
    cmp edx, 0
    je set_zero
    jmp set_one
parmode_eq_one:
    cmp ecx, 1
    je set_zero
    cmp ecx, 2
    jne end
set_one:
    or ebx, 10000000b
    jmp end
set_zero:
    and ebx, 01111111b
end:
    mov eax, ebx        ;  resultado em eax
    pop ebp
    ret
```

Q-11 Relembre a recepção de dados da porta série por via de interrupções. Considere que tem à sua disposição as funções:

- `ligaPIC`/`desligaPIC` que configuram o PIC para aceitar/ignorar interrupções da UART;
- `ligaUART`/`desligaUART` que ligam/desligam as interrupções na UART;
- `confVI/repoeVI` que configuram/repõem o vector de interrupções para associar a função trataByte (a implementar na alínea b)) às interrupções provenientes da UART;
- `bufGet`/`bufPut`/`bufFull`/`bufEmpty` que permitem operar sobre um buffer circular.

a) [1.00 val.] Complemente a implementação de um programa principal que: configura o sistema para a recepção de bytes da UART via interrupções; para cada byte recebido invoca func e usa o resultado para acumular em count o número total de bits recebidos a 1; repõe a configuração inicial do sistema (desliga as interrupções) e, por fim, imprime o número de bits contabilizados.

```
#define EOT    0x04
#define EOI    0x20
#define PICCMD 0x20
#define RBR    0x3F8

int main() {
    unsigned char c;
    int count;

    confVI();
    ligaUART();
    ligaPIC();
    while (c != EOT) {
        c = bufGet();
        count += func(c);
    }
    desligaPIC();
    desligaUART();
    repoeVI();
    printf("Numero de bits a 1: %d", count);
    return 0;
}
```

b) [1.50 val.] Implemente a função que trata a recepção do byte da porta série.

```
void trataByte() {
    unsigned char c;

    c = inByte(RBR);
    if (bufFull())
       ; // Nao se faz nada, o byte perde-se
    else
       bufPut(c);
    outByte(PICCMD, EOI);
}
```

c) [1.00 val.] Indique quais são os passos necessários para obter o executável a partir das linguagens fontes assembly e C, admitindo que o programa se chama *prog.c* e a função *func.asm*.

nasm -f elf func.asm -> gera o ficheiro func.o
gcc -o omeuprog prog.c func.o -> gera o ficheiro prog.o e linka-o com o func.o para obter o executável omeuprog

ou
nasm -f elf func.asm -> gera o ficheiro func.o
gcc -c prog.c -> gera o ficheiro prog.o
ld -o omeuprog prog.o func.o -> linka os ficheiros prog.o e func.o para obter o executável omeuprog