

Exame de Arquitetura de Computadores 16/17 (ver. A)

4/7/2017 Duração 3h 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: _____

Q1- 1,5 val Considere a representação de valores inteiros, com ou sem sinal, numa determinada arquitetura.

a) Quantos bits serão necessários para guardar valores inteiros de -1000 a 1000 (escolha o menor possível)?

- A) 8 **B) 12** C) 16 D) 24 E) 32

b) Qual é o maior valor que se pode representar num `int` com 16 bits (em base 10) ?

- A) 8192 B) 16384 **C) 32767** D) 65535 E) 131071

c) Qual é o menor valor que se pode representar num `int` com 10 bits (em complemento para 2) ?

- A) 1111111111 B) -1000000001 **C) 1000000000** D) 1111111110 E) -1111111111

Q2- 1,5 val Considere um CPU que tem uma unidade aritmética e lógica em que as entradas e a saída têm 8 bits.

a) Diga qual é o resultado (em base 2) da soma de $1111\ 1100_2$ com $0001\ 0111_2$

- A) 1001 0011 B) 1111 0011 C) 0010 0111 D) 0110 1110 **E) 0001 0011**

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=1; OF=0; ZF=1; SF=1
D) CF=0; OF=1; ZF=1; SF=0 **E) CF=1; OF=0; ZF=0; SF=0**

c) Assuma que uma operação ADD (soma) entre números inteiros sem sinal deu a seguinte configuração de *flags*: CF=0; 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 OF e SF estão a um.
B) O resultado está correto porque ambas as flags CF e ZF estão a zero.
C) O resultado está correto porque a flag CF está a zero.
D) O resultado está errado porque a flag OF está a um.
E) O resultado está errado porque a flag SF está a um.

Q3- 1 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)

Assumindo que os 32 bits seguintes “01000000 10100000 00000000 00000000” representam um número na normal IEE754, indique qual o valor correspondente (em decimal).

- A) 125,0 B) 10,0 **C) 5,0** D) 0,1 E) 0,5

Q4- 0,5 val Qual das seguintes expressões na linguagem C ou Java, permite “extrair” o bit 3 de um short (16 bits) guardado na variável ‘num’? O resultado da expressão deverá ser 0 ou 1.

- A) (num >> 2) & 0x1 B) (num >> 13) && 0x1 **C) (num & 8) >> 3**
D) num % 2 E) (num >> 13) & 0x1

Q5- 0,5 val Se executarmos o seguinte programa em C

```
#include <stdio.h>
int main() {
    double x = 4/3.0;
    printf ("x=%.6f\n", x);
    return 0;
}
```

qual dos seguintes resultados obtemos e porquê?

- A) x=1 — Porque os operandos na expressão são inteiros e neste caso x fica com o resultado da divisão inteira
B) x=1.000000 — Porque no printf se pede a escrita como um float e x é um double.
C) x=1.333333 — Porque a expressão é do tipo float cujo resultado é guardado no double x.
D) x=1.000000 — Porque os operandos na expressão são inteiros e neste caso x fica com o resultado da divisão inteira
E) x=1.3333333333333325932 — Porque a expressão é do tipo float, a string de formatação especifica um float e o valor em x não é representável de forma exata na norma IEEE754.

Q6- 0,5 val. No processamento de um conjunto de instruções no pipeline de um CPU podem ocorrer problemas (*pipeline hazards*). Qual das situações apresentadas é um exemplo disso?

- A. Uma instrução lê um operando de um registo que foi usado como destino (resultado) noutra instrução que já foi totalmente executada.
B. Uma das instruções é um salto (*jump*) e o CPU dispõe de “*branch prediction*” .
C. Uma instrução lê um operando de um registo que é também lido pela instrução anterior.
D. Uma instrução de salto condicional depende de uma condição que é avaliada na instrução imediatamente antes.
E. O processador inicia a execução de uma subrotina efectuando “push %ebp”.

Q7- 0,5 val. A programação de entradas/saídas por interrupções:

- A. É menos eficiente do que a espera activa porque usa *buffers*.
B. Permite a troca de dados com os dispositivos periféricos sem usar instruções de I/O.
C. É mais eficiente do que a espera activa porque é executada em modo supervisor.
D. Permite deixar o CPU livre enquanto um ou mais periféricos efetuam as suas operações.
E. Vai levar a que se percam dados de um periférico quando o CPU está ocupado com outro periférico.

Q8- 0,5 val. Assuma que se está a escrever várias sequências de dados contíguos de pequenas dimensões (e.g., 10 bytes de cada vez) num ficheiro num disco SSD – *Solid State Drive*. Pode ser vantajoso ir guardando num *buffer* e adiar esta escrita para que:

- A. Várias dessas sequências sejam escritas numa única operação poupando tempo no acesso ao disco e no desgaste das memórias desse disco.**
B. Garantir que completa um bloco antes do o mandar realmente escrever para reduzir o tempo dos movimentos na cabeça de escrita (*seek time*).
C. Várias dessas sequências sejam escritas numa única operação para evitar ter de apagar o ficheiro várias vezes, com cada escrita.
D. Garantir que completa um bloco antes do o mandar realmente escrever para evitar os erros de escrita e possíveis buracos no ficheiro.
E. Facilitar a execução pelo CPU do código que atende a interrupção de escrita no disco.

Q9- 0,5 val. Para que o Sistema de Operação possa suportar múltiplos processos em execução sem que estes interfiram entre si, a arquitetura *hardware* deve suportar alguns mecanismos. Tal inclui:

- A. Dois modos de execução, para que algumas instruções só possam ser executadas pelo SO, como por exemplo o CALL e o RET.
- B. Dois modos de execução, um para o SO e outro para os processos para os impedir de usar algumas instruções.**
- C. As instruções IN/OUT, para acesso aos periféricos, são privilegiadas e só podem ser executadas em modo utilizador pelos processos autorizados.
- D. Algumas instruções, como por exemplo o CLI e o STI, para que os processos possam aceder aos periféricos de forma segura.
- E. Instruções PUSH e POP que permitem salvar na pilha o endereço da chamada e retornar ao ponto da chamada.

Q10- 0,5 val. Considere uma hierarquia de memórias com uma cache e memória central RAM, em que o tempo acesso à cache é de 2 ns e à memória central é de 12 ns. Indique qual das seguintes expressões é verdadeira:

- A. Com um hit ratio de 90% teremos um tempo médio de acesso à memória de 2,5 ns.
- B. Com um hit ratio de 85% teremos um tempo médio de acesso à memória de 2,5 ns.
- C. Com um hit ratio de 95% teremos um tempo médio de acesso à memória de 2,5 ns.**
- D. Com um hit ratio de 95% teremos um tempo médio de acesso à memória de 3 ns.
- E. Com um hit ratio de 99% teremos um tempo médio de acesso à memória de 2,0 ns.

Q11- 0,5 val. As caches associativas puras:

- A. Podem introduzir muitas falhas (*misses*) por conflito.
- B. Necessitam de muito *hardware* para gestão/pesquisa para cada uma das linhas.**
- C. Um endereço será sempre encontrado na mesma linha, se estiver na cache.
- D. Têm mais falhas (*misses*) que as caches de mapa direto.
- E. De todo o tipo de caches são as que têm tempos de acesso mais lentos.

Q12- 2 val. Uma arquitetura de um computador tem as seguintes características: endereçamento de 32 bits; um nível de cache com organização associativa de 256 grupos(ou conjuntos) com linhas de 32 bytes cada. Nesta cache, os endereços são interpretados do seguinte modo:

- Os 8 bits, do 5 ao 12, referem o número do grupo/conjunto
- Os 19 bits, do 13 ao 31, servem de chave/tag

a) O número de bits necessários para endereçar cada byte numa linha é:

- A. 3
- B. 4
- C. 5**
- D. 6
- E. 7

b) O número de bits necessários para endereçar cada grupo/conjunto é:

- A. 5
- B. 6
- C. 7
- D. 8**
- E. 9

c) Se a cache tiver 128K de capacidade, cada grupo/conjunto tem quantas linhas?

- A. 2
- B. 4
- C. 8
- D. 16**
- E. 32

d) Um determinado endereço está na cache (dá um hit) se (indique a mais correta):

- A. A sua tag existir na cache
- B. A sua tag existir na cache, no grupo respectivo do endereço
- C. A sua tag existir na cache, no grupo respectivo do endereço e na linha respectiva do endereço
- D. A sua tag existir na cache, no grupo respectivo do endereço e com o bit de validade a 1**
- E. A sua tag existir na cache, no grupo respectivo do endereço, na linha respectiva do endereço e com o bit de validade a 1

Q13- 0,5 val No contexto da gestão de memória virtual de programas em execução, o conceito de paginação permite:

- A. Dividir a memória real em blocos mais facilmente manipuláveis pelas caches.
- B. Dividir o espaço de endereçamento de qualquer programa em duas páginas, uma para código e outra para dados.
- C. Evitar a fragmentação interna quando se usam páginas de grandes dimensões.
- D. Ter vários espaços de endereçamento virtual sem fragmentação externa.**
- E. Dividir o espaço de endereçamento dos programa em páginas de tamanho variável.

Q14- 0,5 val. Na gestão da memória virtual baseada em paginação a pedido:

- A. A tabela de páginas de um processo é guardada em memória RAM
- B. A tabela de páginas é guardada na TLB da MMU.
- C. A tabela de páginas de um processo é guardada em disco.
- D. A tabela de páginas é guardada na cache L1.
- E. A tabela de páginas é guardada no controlador de DMA.

Q15- 1 val. Suponha que um dado CPU emite endereços virtuais com 32 bits mas a memória é endereçada por endereços reais de apenas 30 bits. A unidade de transformação de endereços gere a memória física dividindo-a em páginas de 16 KBytes.

- A. O número de bits necessário para identificar o número da página virtual é 18, para o número do frame usa 16 bits e o número de bits necessário para o deslocamento dentro da página é 14.
- B. O número de bits necessário para identificar o número da página virtual é 20, para o número do frame usa 18 bits e o número de bits necessário para o deslocamento dentro da página é 12.
- C. O número de bits necessário para identificar o número da página virtual é 12, para o número do frame usa 16 bits e o número de bits necessário para o deslocamento dentro da página é 24.
- D. O número de bits necessário para identificar o número da página virtual é 18, para o número do frame usa 18 bits e o número de bits necessário para o deslocamento dentro da página é 14.
- E. O número de bits necessário para identificar o número da página virtual é 20, para o número do frame usa 20 bits e o número de bits necessário para o deslocamento dentro da página é 12.

Q16- 2 val. Considere uma arquitetura com endereços virtuais de 28 bits, que suporta páginas com dimensão 4 KBytes (2^{12}) e permite paginação a pedido. Dado o conteúdo do TLB e das primeiras 7 entradas da tabela de páginas indique, para a sequência seguinte de acessos a memória, se ocorre um hit ou miss no TLB, se a tabela é consultada ou não e, se obtiver uma página válida, qual o endereço real obtido. Caso não tenha informação suficiente para preencher alguma entrada, coloque um traço (“—”). Se não existir frame para o endereço indique “***” no endereço real.

Exemplo de preenchimento:

acesso: 0x0006004: miss no TLB; consulta tabela: sim ; endereço real gerado: ***

TLB:

página virtual	página física
0	0x0103
2	0x0233
5	0x0880
—	—

Tabela de Páginas:

página virtual	página física
0	0x0103
1	(inválida)
2	0x0233
3	0x0900
4	0x4087
5	0x0880
6	(inválida)

acesso: 0x0005004 — hit no TLB; consulta tabela: não ; endereço real gerado: 0x0880004

acesso: 0x0001104 — miss no TLB; consulta tabela: sim ; endereço real gerado: ***

acesso: 0x0004004 — miss no TLB; consulta tabela: sim ; endereço real gerado: 0x4087004

acesso: 0x0000103 — hit no TLB; consulta tabela: não ; endereço real gerado: 0x0103103

acesso: 0x0000000 — hit no TLB; consulta tabela: não ; endereço real gerado: 0x0103000

Q17- 2 val Pretende-se que escreva uma função denominada *troca* que implementa a inversão da ordem dos elementos de um vetor de inteiros:

```
void troca( int v[], int len );
```

- Recebe como parâmetros de entrada:

- *v*: o endereço inicial de um vetor com inteiros
- *len*: o número de inteiros no vetor *v* (**len>0**)

Como exemplo, assuma que esta subrotina é chamada pelo código C seguinte:

```
int v[6] = {2, 1, 3, 1, 7, 1};
```

```
troca(v, 6);
```

A execução deste troço de programa deve deixar o vetor **v** com o seguinte conteúdo:

v: 1, 7, 1, 3, 1, 2

a) Implemente em C a função *troca*.

```
void troca( int v[], int len )
{
    int i;
    for ( i=0; i<len/2; i++ ) {
        int x = v[i];
        v[i] = v[len-i-1];
        v[len-i-1] = x;
    }
}
```

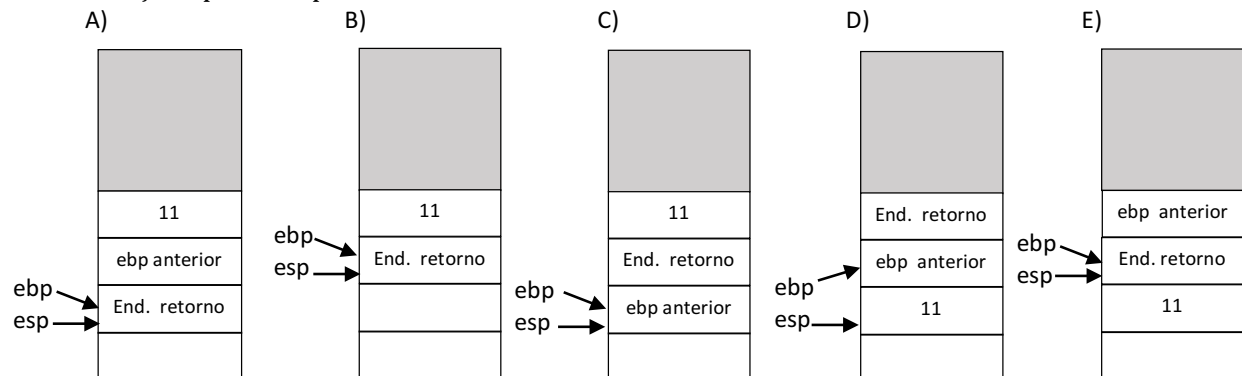
b) Implemente agora em *assembly* a função *troca* usando as convenções das aulas.

.text	
# código da sua solução:	
<pre>.global troca # uma solucao possivel... troca: push %ebp mov %esp, %ebp push %ebx push %ecx push %edx # salvaguarda registos mov 8(%ebp), %ebx # endereco de v mov %ebx, %ecx add 12(%ebp), %ecx sub \$4, %ecx # end. de v[len-1] ciclo: cmp %ebx, %ecx jbe fim mov (%ebx), %eax mov (%ecx), %edx mov %edx, (%ebx) mov %eax, (%ecx) add \$4, %ebx sub \$4, %ecx jmp ciclo fim: pop %edx pop %ecx pop %ebx pop %ebp ret</pre>	<pre>.global troca # outra solucao possivel... troca: push %ebp mov %esp, %ebp push %ebx push %ecx push %edx # salvaguarda registos mov 8(%ebp), %ebx # %ebx = endereco de v mov \$0, %ecx # %ecx = 0 mov 12(%ebp), %edx dec %edx # %edx = len - 1 ciclo: cmp %ecx, %edx jbe fim # termina quando %edx <= %ecx mov (%ebx, %ecx, 4), %eax push %eax # guarda v[%ecx] na pilha mov (%ebx, %edx, 4), %eax mov %eax, (%ebx, %ecx, 4) # v[%ecx] = v[%edx] pop %eax mov %eax, (%ebx, %edx, 4) # v[%edx] = v[%ecx] inc %ecx dec %edx jmp ciclo fim: pop %edx pop %ecx pop %ebx pop %ebp ret</pre>

Q18- 1,5 val Considere a seguinte função recursiva implementada na linguagem C:

```
unsigned int impar(unsigned int x) {
    if (x == 0)
        return 0;
    else if (x == 1)
        return 1;
    else return impar(x-2);
}
```

- a) Indique, na folha de respostas, qual das seguintes configurações da pilha (stack) está correta para a chamada `impar(11)`, se pararmos a execução do programa antes da execução da instrução apontada pela seta.



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

```
impar:
.global impar
    push    %ebp
    mov     %esp, %ebp

    mov     8(%ebp), %eax    # x

    cmp     $0, %eax
    je      fim

    cmp     $1, %eax
    je      fim

    sub     $2, %eax
    push    %eax
    call    impar
    add     $4, %esp

fim:
    pop     %ebp
    ret
```

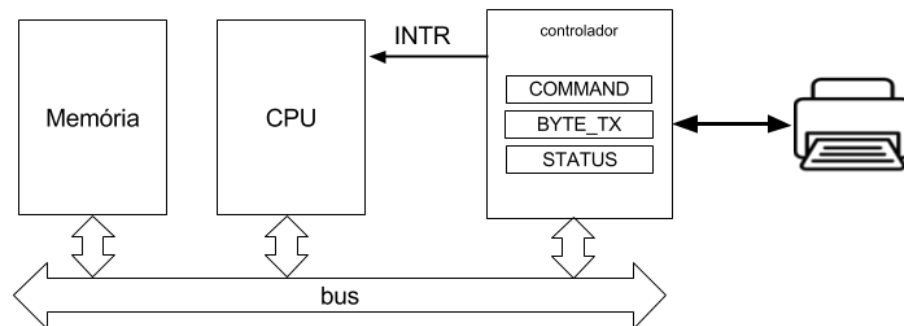
Q19- 2,5 val. As várias alíneas desta pergunta supõem um ambiente, do ponto de vista do *software*, semelhante ao usado nas aulas práticas e no mini-projecto, como as seguintes funções:

```
unsigned char inportb(unsigned int portid); // IN de um byte
void outportb(unsigned int portid, unsigned char value); // OUT de um byte
void enable(void); // STI - ligar flag de interrupções
void disable(void); // CLI - desligar a flag de interrupções
setvect(int intrnum, void interrupt(*isr)()); //afetar o vetor de interrupções
FILE *fopen(char *fname, char *mode); // abre ficheiro no modo indicado
int fgetc( FILE*f ); // lê próximo char ou EOF
int fclose( FILE*f ); // fecha ficheiro
```

Pode usar também as operações sobre um buffer circular como as usadas nas aulas práticas:

```
bufPut( unsigned char byte ), unsigned char bufGet() ,
int bufFull() , int bufEmpty() .
```

Do ponto de vista do *hardware*, considere um controlador de uma interface que permite imprimir texto numa impressora. A impressão é efetuada enviando a sequência de bytes que representa o texto a imprimir, sendo enviado um de cada vez. Assuma que impressora é **muito mais lenta** que o nosso computador, como é habitual. Nesta questão, assuma também que este é também o único controlador ligado à linha de interrupção do CPU. Considere um mecanismo de interrupções vectorizadas onde este periférico usa a entrada número 3 para identificar a interrupção.



Todos os registos do controlador têm 8 bits. Os registos relevantes são os seguintes, com a dimensão e os endereços de IO (ports) indicados:

- o **COMMAND (8bits - 0x10)** — só pode ser escrito. O valor escrito descreve qual o modo de funcionamento deste controlador, onde cada bit tem o seguinte significado:
 - o bit 0 - colocando o valor 1 significa que o controlador deve enviar interrupções. Quando o bit é colocado a 1 e o controlador está desocupado *é logo gerada uma interrupção*. Colocar este bit a zero significa que o controlador nunca deve enviar interrupções.
 - o bit 1 - colocado a 1 indica que o bit 1 do registo de STATUS deve reportar se ocorrem erros na impressora. Zero significa que o bit 1 de STATUS nunca será usado.
- o **BYTE_TX (8bits - 0x11)** — só pode ser escrito e identifica o byte a ser enviado para a impressora. De cada vez que este registo é escrito, o controlador começa imediatamente a enviar esse byte para a impressora de acordo com o modo antes indicado em COMMAND. Se o controlador já estava a enviar algo para a impressora, o byte anterior ou o corrente podem-se perder ou serem corrompidos.
- o **STATUS (8bits - 0x12)** - só pode ser lido;
 - o bit 0 - está a um se o controlador está ocupado com uma transferência; a zero se está livre
 - o bit 1 - fica a 1 se houve um erro durante a impressão do último byte e 0 se tudo correu bem, desde que programado em COMMAND para tal. Note que o controlador pode indicar vários tipos de erros (impressora desligada, avaria da impressora, etc.), mas tal está fora do âmbito desta questão.

Assuma que não existe mais nenhum *software* a manipular este controlador.

a) Escreva, usando C, a função `imprimeByte` que envia para a impressora um byte sem usar interrupções, mas garantindo que avalia o resultado da operação devolvendo 1 se o byte foi impresso sem erros e 0 em caso contrário.

```
int imprimeByte( unsigned char byte )
{
    while ( ((s=inportb( 0x12 )) & 1) == 1 )
        ;

    outportb( 0x10, 2 );
    outportb( 0x11, byte );           // manda escrever

    while ( ((s=inportb( 0x12 )) & 1) == 1 )
        ;

    if ( (s & 2) == 0 ) return 1;      // houve erro ou nao?
    else return 0;
    // alternativa ao if-else: return !(s & 2);

}
```

b) Escreva, usando C, a função `imprimeFich` que envia para a impressora todo o conteúdo de um ficheiro sem usar interrupções, mas garantindo que a impressão decorre corretamente. Se algum erro for detectado deve terminar a impressão e devolver logo zero. Devolve 1 se nenhum erro for detectado.

```
int imprimeFich( char *filename )
{
    int s = 1;
    int c;
    FILE *f=fopen(filename, "r");

    while ( (c=fgetc( f ))!=EOF && s )
        s = imprimeByte(c);

    fclose( f );
    return s;

}
```

c) Escreva, em C (tipo turbo C), as duas funções seguintes que realizam a escrita de um ficheiro **usando interrupções e assumindo que não ocorrem erros**:

```
void initImpInt( )
```

Esta função deve programar o controlador e o mecanismo de interrupções para que seja possível imprimir de acordo com a descrição seguinte.

```
void imprimeFichInt( char *filename )
```

Esta função deve iniciar a impressão do conteúdo do ficheiro indicado, usando a função anterior e garantindo que a rotina de serviço (meuISR) vai imprimir os vários bytes do ficheiro via um buffer circular onde esta função vai colocar os vários bytes do ficheiro. A leitura do ficheiro deve poder ocorrer em simultâneo com a impressão de bytes já lidos. Esta função só termina quando confirma que todos os bytes do ficheiro foram enviados para a impressora.

```
void interrupt meuISR( )
```

Esta função realizará o atendimento de interrupções deste controlador e deve pedir a escrita do próximo byte; ou não fazer nada, caso não haja mais dados para imprimir.

```
/* Variáveis globais partilhadas por todas as funções do programa: */  
/* buffer circular acessível por bufPut, bufGet, bufFull e bufEmpty */
```

```
void initImpInt() {
```

```
    setvect( 3, meuISR );  
    outportb(0x10, 1 );
```

```
}
```

```
. . . . .
```

```
void imprimeFichInt( char *filename )  
{
```

```
    FILE *f = fopen( filename, "r");  
    int c=fgetc(f);  
    if ( c!=EOF ) {  
        bufPut(c);  
        initImpIntr();    // so ligar intr. quando  
                           // temos um byte no buffer
```

```
    }  
    while ( (c=fgetc(f))!= EOF ) {  
        while ( bufFull() )  
            ;  
        bufPut(c);  
    }
```

```
    fclose( f );  
    while ( !bufEmpty() )  
        ;  
    // outportb(0x10, 0); // nada foi dito sobre como terminar
```

```
}
```

```
. . . . .
```

```
void interrupt meuISR( )  
{
```

```
    enable();
```

```
    if ( ! bufEmpty() )  
        outportb( 0x11, bufGet() ); // manda escrever proximo byte
```

```
}
```

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