Tarefas

Aula 8 (EMULADOR)

Aula 9 (Somar 5 + 4)

```
void load_microprog()
{
   //implementar!
   microprog[0] = 0b00000000010000110101000001000010001;
   microprog[1] = 0b00000001000000110101000001000010001;
   microprog[2] = 0b00000001100000010100100000000000010;
   microprog[3] = 0b00000010000000110101000001000010001;
   microprog[4] = 0b00000010100000010100010000000000010;
   microprog[5] = 0b00000000000000011110000000100000010;
}
void load_prog()
{
   memory[0] = 0b00000000;
   memory[1] = 0b00000001;
   memory[2] = 0b00000101;  //5
   memory[3] = 0b00000100;  //4
}
```

Aula 10

A)

```
void load_microprog()
{
   //MAIN
   microprog[0] = 0b00000000010000110101000001000010001; //PC <- PC + 1; fetch;
GOTO MBR;

   //OPC = OPC + memory[end_word];
   microprog[2] = 0b00000001100000110101000001000010001; //PC <- PC + 1; fetch;
   microprog[3] = 0b00000010000000010100000000010100010; //MAR <- MBR; read;
   microprog[4] = 0b00000010100000010100100000000000000; //H <- MDR;
   microprog[5] = 0b00000000000000011110001000000000001000; //OPC <- OPC + H; GOTO
MAIN;

   //memory[end_word] = OPC;
   microprog[6] = 0b00000011100000110101000001000010001; //PC <- PC + 1; fetch;
   microprog[7] = 0b00000100000000010100000000010000010; //MAR <- MBR;
   microprog[8] = 0b00000000000000010100000000101001000; //MDR <- OPC; write;
GOTO MAIN;
```

```
    //goto endereco_comando_programa;
    microprog[9] = 0b00000101000000110101000000001000010001; //PC <- PC + 1; fetch;
    microprog[10] = 0b00000000010000010100000000001000010010; //PC <- MBR; fetch;
GOTO MBR;

    //if OPC = 0 goto endereco_comando_programa else goto proxima_linha;
    microprog[11] = 0b00000110000100010100010000000000001000; //OPC <- OPC; IF ALU =
0 GOTO 268 (100001100)
ELSE GOTO 12 (000001100);
    microprog[12] = 0b00000000000000011010100000010000000001; //PC <- PC + 1; GOTO
MAIN;
    microprog[268] = 0b10000110100000110101000000010000010001; //PC <- PC + 1; fetch;
    microprog[269] = 0b00000000010000010100000000001000010010; //PC <- MBR; fetch;
GOTO MBR;

    //OPC = OPC - memory[end_word];
    microprog[13] = 0b00000111000000110101000000010000010001; //PC <- PC + 1; fetch;
    microprog[14] = 0b00000111100000001010000000000010100010; //MAR <- MBR; read;
    microprog[15] = 0b00001000000000001010010000000000000000; //H <- MDR;
    microprog[16] = 0b00000000000000011111101000000000001000; //OPC <- OPC - H;
GOTO MAIN;
}
```

A e B)

(comentários pertencem a reposta B)

```
void load_prog()
{
    memory[1] = 0b00000010; //fetch do microprog[0]
    memory[2] = 0b00001010; //fetch do microprog[2]
    memory[3] = 0b00000010; //fetch do microprog[0]
    memory[4] = 0b00001011; //fetch do microprog[2]
    memory[5] = 0b00000110; //fetch do microprog[0]
    memory[6] = 0b00001100; //fetch do microprog[6]
    memory[40] = 0b00000101; //read do microprog[3]
    memory[44] = 0b00000011; //read do microprog[3]
}
```

C)

D)

E)

Aula 11

C)

D)

E)

Aula 12

A)
BIPUSH 5
BIPUSH 3
IADD

Aula 13

A)
```
void load_microprog(){
      FILE *microp = fopen("microprog.rom", "rb");
   if (microp != NULL) {
      fread(microprog, sizeof(microcode), 512, microp);
      fclose(microp);
   }
   else {
      std::cout << "Cadê o microprograma? " << std::endl;
      exit(-1);
   }
}
```

Aula 14

A)

```
    memory[1] = 0x73; //init (bytes 2 e 3 são descartados por conveniência de implementação)

   memory[4] = 0x00000006; //(CPP inicia com o valor 0x0006 guardado na palavra 1 – bytes 4 a 7.)

   word tmp = 0x00001001; //LV

   memcpy(&(memory[8]), &tmp, 4); //(LV inicia com o valor de tmp guardado na palavra 2 – bytes 8 a 11)

   tmp = 0x00000400; //PC

   memcpy(&(memory[12]), &tmp, 4); //(PC inicia com o valor de tmp guardado na palavra 3 – bytes 12 a 15)

   tmp = 0x00001003; //SP
//SP (Stack Pointer) é o ponteiro para o topo da pilha.
//A base da pilha é LV e ela já começa com algumas variáveis empilhadas (dependendo do programa).
//Cada variável gasta uma palavra de memória. Por isso a soma de LV com num_of_vars.
```

```
    memcpy(&(memory[16]), &tmp, 4); //(SP inicia com o valor de tmp guardado na palavra 4 – bytes 16 a 19)

    memory[1025] = 0x19;
    memory[1026] = 0x15;
    memory[1027] = 0x22;
    memory[1029] = 0x19;
    memory[1030] = 0x0C;
    memory[1031] = 0x19;
    memory[1032] = 0x03;
    memory[1033] = 0x02;
    memory[1035] = 0x01;
    memory[1036] = 0x1C;
    memory[1037] = 0x01;
    memory[1038] = 0x1C;
    memory[1039] = 0x00;
    memory[1040] = 0x4B;
    memory[1042] = 0x08;
    memory[1043] = 0x1C;
    memory[1044] = 0x01;
    memory[1045] = 0x3C;
    memory[1046] = 0xFF;
    memory[1047] = 0xF2;
    memory[1048] = 0x01;
```

B)

```
bipush 7
istore a
bipush 0
istore b
soma iload b
bipush 9
iadd
istore b
iload a
bipush 1
isub
istore a
iload a
bipush 0
if_icmpeq fim
goto soma
fim nop
```

Aula 15 (MONTADOR)

Aula 16

A)

```
//MAIN
[  0] PC <- PC + 1; fetch; GOTO MBR;

//OPC = OPC + memory[end_word];
[  2] PC <- PC + 1; fetch;
[  3] MAR <- MBR; read;
[  4] OPC <- OPC + MDR; GOTO MAIN;

//memory[end_word] = OPC;
[  6] PC <- PC + 1; fetch;
[  7] MAR <- MBR;
[  8] MDR <- OPC; write; GOTO MAIN;

//goto endereco_comando_programa;
[  9] PC <- PC + 1; fetch;
[ 10] PC <- MBR; fetch; GOTO MBR;

//if OPC = 0 goto endereco_comando_programa else goto proxima_linha;
[ 11] OPC <- OPC; IF ALU = 0 GOTO 268 (100001100)
                          ELSE GOTO 12 (000001100);
[ 12] PC <- PC + 1; GOTO MAIN;
[268] PC <- PC + 1; fetch;
[269] PC <- MBR; fetch; GOTO MBR;

//OPC = OPC - memory[end_word];
[ 13] PC <- PC + 1; fetch;
[ 14] MAR <- MBR; read;
[ 15] OPC <- OPC - MDR; GOTO MAIN;
```

Consegui diminuir das instruções de soma e sub 1 ciclo.

Aula 17

A)

B) Com a Ryzen, a AMD está lançando seu novo modelo de hardware Neural Net Prediction junto com o Smart Pre-Fetch. A AMD está anunciando isso como uma "verdadeira rede artificial dentro de cada processador Zen que cria um modelo de decisões com base na execução do software". Isso pode significar uma das várias coisas, desde a modelagem física real do fluxo de trabalho das instruções até a identificação de caminhos críticos a serem acelerados (improvável) ou a análise estatística do que está passando pelo mecanismo e a tentativa de trabalhar durante o tempo de inatividade que pode acelerar instruções futuras (como inserir uma instrução para decodificar em um decodificador inativo em preparação para quando ele realmente ocorrer, portanto, acaba usando o cache

micro-op e tornando-o mais rápido). Os processadores modernos já executam trabalhos decentes quando o trabalho repetitivo está sendo usado, como identificar quando cada quarto elemento de uma matriz de memória está sendo acessado e podem extrair esses dados mais cedo para ficarem prontos caso sejam usados. O perigo de preditores inteligentes, no entanto, é ser excessivamente agressivo - extrair muitos dados para que os dados antigos possam ser descartados porque nunca são usados (sob predição), extrair muitos dados para que eles já sejam despejados quando os dados forem necessários (agressivo previsão), ou simplesmente desperdiçando excesso de poder com previsões ruins (previsão estúpida ...). A AMD está declarando que o Zen implementa modelos de aprendizado de algoritmo para predição de instruções e pré-busca, o que sem dúvida será interessante ver se eles encontraram o equilíbrio certo entre agressão de pré-busca e trabalho extra na previsão. Vale a pena notar aqui que a AMD provavelmente utilizará a largura de banda L3 aumentada no novo núcleo como um elemento-chave para auxiliar na pré-busca, especialmente porque o cache L3 compartilhado é um cache vítima e projetado para conter dados já usados / despejados para uso novamente em uma data posterior.

Aula 18

A)Xbox: RISC, Xbox 360: RISC, Xbox One:CISC, PS2: RISC, PS3: RISC, PS4: CISC

B)

Aula 19

A)A linguagem assembly x86 tem dois principais sintaxe ramos: Intel sintaxe, originalmente usado para documentação da plataforma x86 , e AT & T sintaxe . Sintaxe Intel é dominante no MS-DOS e do Windows mundo, e AT & T sintaxe é dominante no Unix mundo, desde Unix foi criado na AT & T Bell Labs. As principais diferenças entre a sintaxe Intel e sintaxe AT & T:

-Solicitação de parâmetro

AT & T
Fonte antes do destino.
mov $5, %eax
Intel
Destino antes de fonte.
mov eax, 5

-Tamanho do parâmetro

AT & T
Mnemônicas são sufixo com uma letra que indica o tamanho dos operandos: q para QWORD, l longo (DWORD), w para a palavra, e b a byte.
addl $4, %esp

Intel
Derivado do nome do registo que é usado (por exemplo rax, eax, machado, al implica q, l, w, b , respectivamente).
add esp, 4

-Sigilos

AT & T
Valores imediatos prefixados com um "$", registros prefixados com um "%".
Intel
A montadora detecta automaticamente o tipo de símbolos; ou seja, se são registros, constantes ou outra coisa.

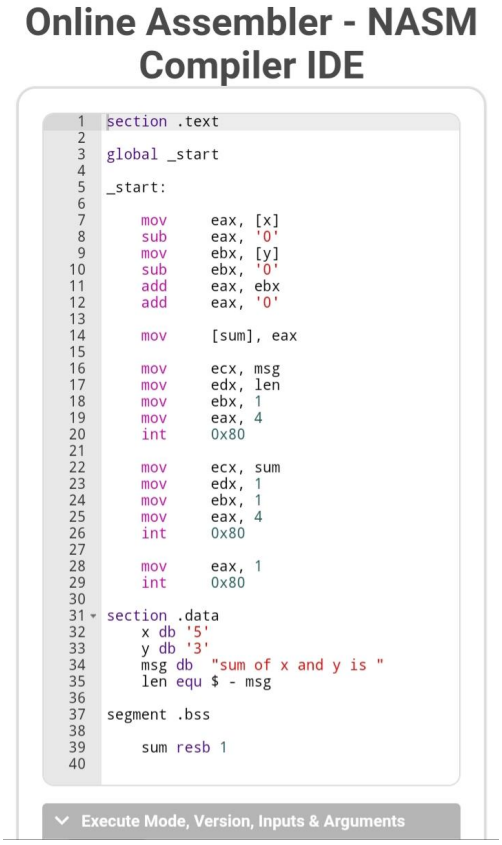
-Eficazes endereços

AT & T
Sintaxe geral de DISP (BASE, INDEX, ESCALA) . Exemplo:
movl mem_location(%ebx,%ecx,4), %eax
Intel
Expressões aritméticas em colchetes; Além disso, palavras-chave de tamanho como byte , palavra , ou dword tem que ser usado se o tamanho não pode ser determinado a partir dos operandos. Exemplo:
mov eax, [ebx + ecx*4 + mem_location].

Aula 20

A)BIPUSH: Endereçamento imediato
ILOAD: Endereçamento direto
GOTO: Endereçamento de registrador

Aula 21
Como estou sem computador eu busquei por simuladores.



Aula 22

A)

Começando do básico, os endereços de IRQ são interrupções de hardware, canais que os dispositivos podem utilizar para chamar a atenção do processador. Apesar de podermos rodar muitos programas ao mesmo tempo, os processadores são capazes de fazer apenas uma coisa de cada vez. A multitarefa surge de um chaveamento muito rápido entre os aplicativos abertos, dando a impressão de que todos realmente estão sendo executados ao mesmo tempo. Mas, o que fazer quando o processador está ocupado, processando qualquer coisa e você digita um caracter do teclado, o modem precisa transmitir dados para o processador, ou qualquer coisa do gênero? É neste ponto que entram os endereços de IRQ. Ao ser avisado através de qualquer um destes canais, o processador imediatamente para qualquer coisa que esteja fazendo e dá atenção ao dispositivo, voltando ao trabalho logo depois. Um mesmo IRQ não pode ser compartilhado entre dois dispositivos e existem apenas 16 endereços disponíveis, que não podem ser expandidos, ou seja, temos que nos virar com o que temos. O número do endereço de IRQ indica também a sua prioridade, começando do 0 que é o que tem a prioridade mais alta. Não é à toa que o IRQ 0 é ocupado pelo sinal de clock da placa mãe, pois é ele quem sincroniza o trabalho de todos os componentes, inclusive do processador. Logo depois vem o teclado, que ocupa o IRQ 1. Veja que o teclado é o dispositivo com um nível de prioridade mais alto, para evitar que as teclas digitadas se percam.Em seguida vêm os demais periféricos, como as portas seriais e o drive de disquetes. A IRQ2 ficava livre para a instalação de um periférico qualquer. Na verdade, na maioria das vezes o 5 também ficava livre.Apartir do 286, houve uma evolução

neste esquema, pois finalmente os PCs passaram a ter 16 endereços de IRQ, numerados de 0 a 15, como nos dias de hoje. Como quase todas as evoluções na família PC, foi preciso manter compatibilidade com o padrão anterior, para que as placas para XT pudessem funcionar nos PCs 286 em diante.Assim, resolveram manter o controlador de IRQs original para que tudo continuasse funcionando da mesma maneira que antes e simplesmente adicionar um segundo controlador para obter os 8 novos endereços. Este segundo controlador passou a ser ligado no IRQ 2, que costumava ficar livre. Todos os pedidos de interrupção dos periféricos ligados aos endereços entre 8 e 15, controlados pelo segundo controlador, passam primeiro pelo IRQ 2, para só depois chegar ao processador. Isto é chamado de cascateamento de IRQs.Em primeiro lugar que o IRQ 2 não pode mais ser utilizado por nenhum periférico. Caso você jumpeie um modem para usar o IRQ 2, ele será remapeado para o IRQ 9. Ou seja, na prática, não temos 16 endereços de IRQ, mas apenas 15. Em segundo lugar, como o segundo controlador está ligado ao IRQ 2, todas as placas que utilizarem os endereços de 8 a 15, terão prioridade sobre as que usarem os IRQs de 3 a 7, pois, do ponto de vista do processador, estão ligadas ao IRQ 2, que é por onde todos os pedidos chegam a ele.Num PC atual, os endereços de IRQ, esta é a configuração de endereços mais comum:

IRQ 0 – Sinal de clock da placa mãe (fixo)
IRQ 1 – Teclado (fixo)
IRQ 2 – Cascateador de IRQs (fixo)
IRQ 3 – Porta serial 2
IRQ 4 – Porta serial 1
IRQ 5 – Livre
IRQ 6 – Drive de disquetes
IRQ 7 – Porta paralela (impressora)
IRQ 8 – Relógio do CMOS (fixo)
IRQ 9 – Placa de vídeo
IRQ 10 – Livre
IRQ 11 – Controlador USB
IRQ 12 – Porta PS/2
IRQ 13 – Coprocessador aritmético
IRQ 14 – IDE Primária
IRQ 15 – IDE Secundária

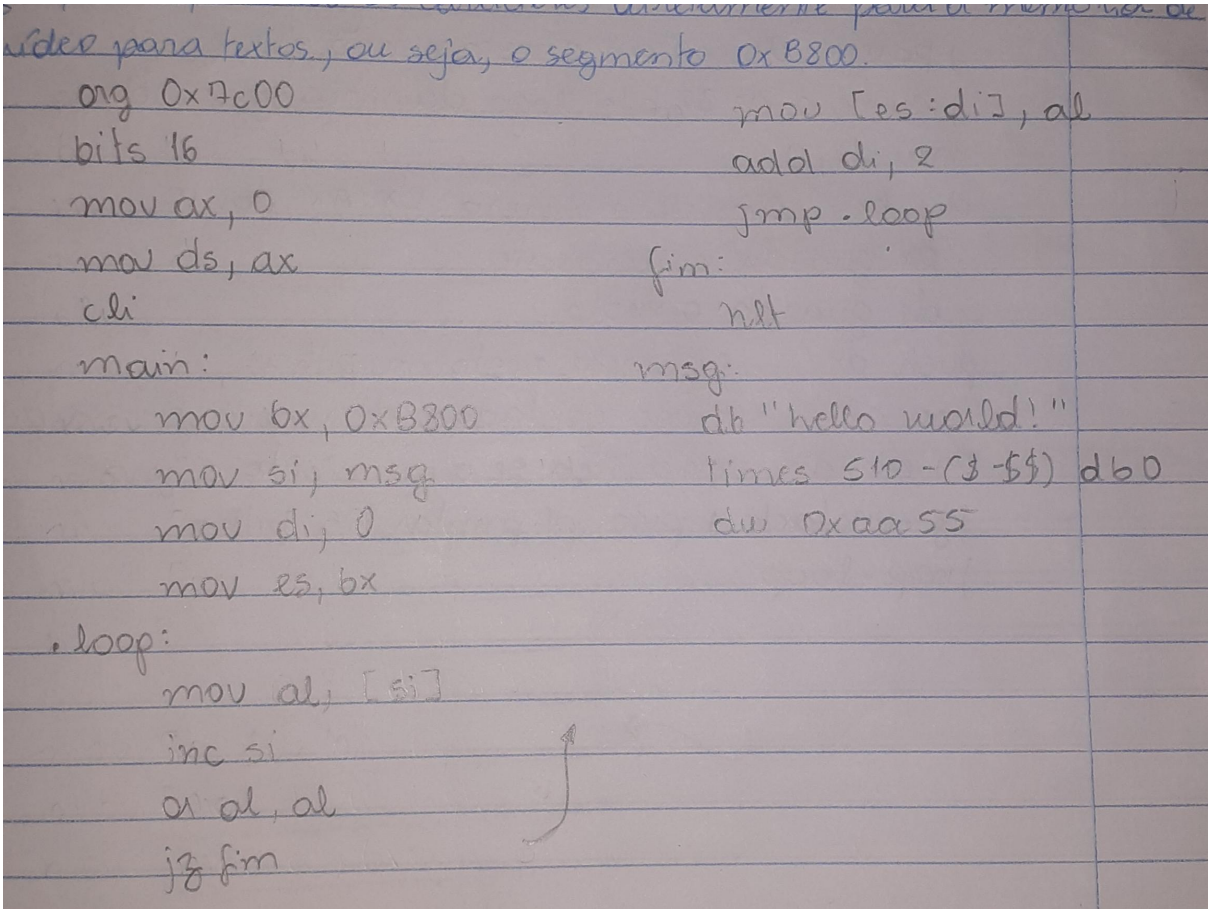
Quando desenvolveram o barramento PCI, incluindo o recurso de PCI Steering, que permite que dois, ou mais periféricos PCI compartilhem o mesmo endereço de IRQ. Neste caso, o controlador PCI passa a atuar como uma ponte entre os periféricos e o processador. Ele recebe todos os pedidos de interrupção, os encaminha para o processador e, ao receber as respostas, novamente os encaminha para os dispositivos corretos. Como o controlador é o único diretamente conectado ao processador é possível ocupar apenas um endereço de IRQ.Além do barramento PCI, outros barramentos usados atualmente permitem compartilhar um único IRQ entre vários periféricos. O USB é um bom exemplo, o controlador ocupa um único IRQ, que é compartilhado entre todas as portas USB e todos os dispositivos conectados a elas. Mesmo que a sua placa mãe tenha 6 portas USB e você utilize todas, terá ocupado apenas um endereço.Caso você utilizasse apenas periféricos

USB, mouse, impressora, scanner, etc. poderia desabilitar todas as portas de legado da sua placa mãe: as duas seriais, a paralela e a PS/2. Seriam 4 endereços de IRQ livre.

Aula 23

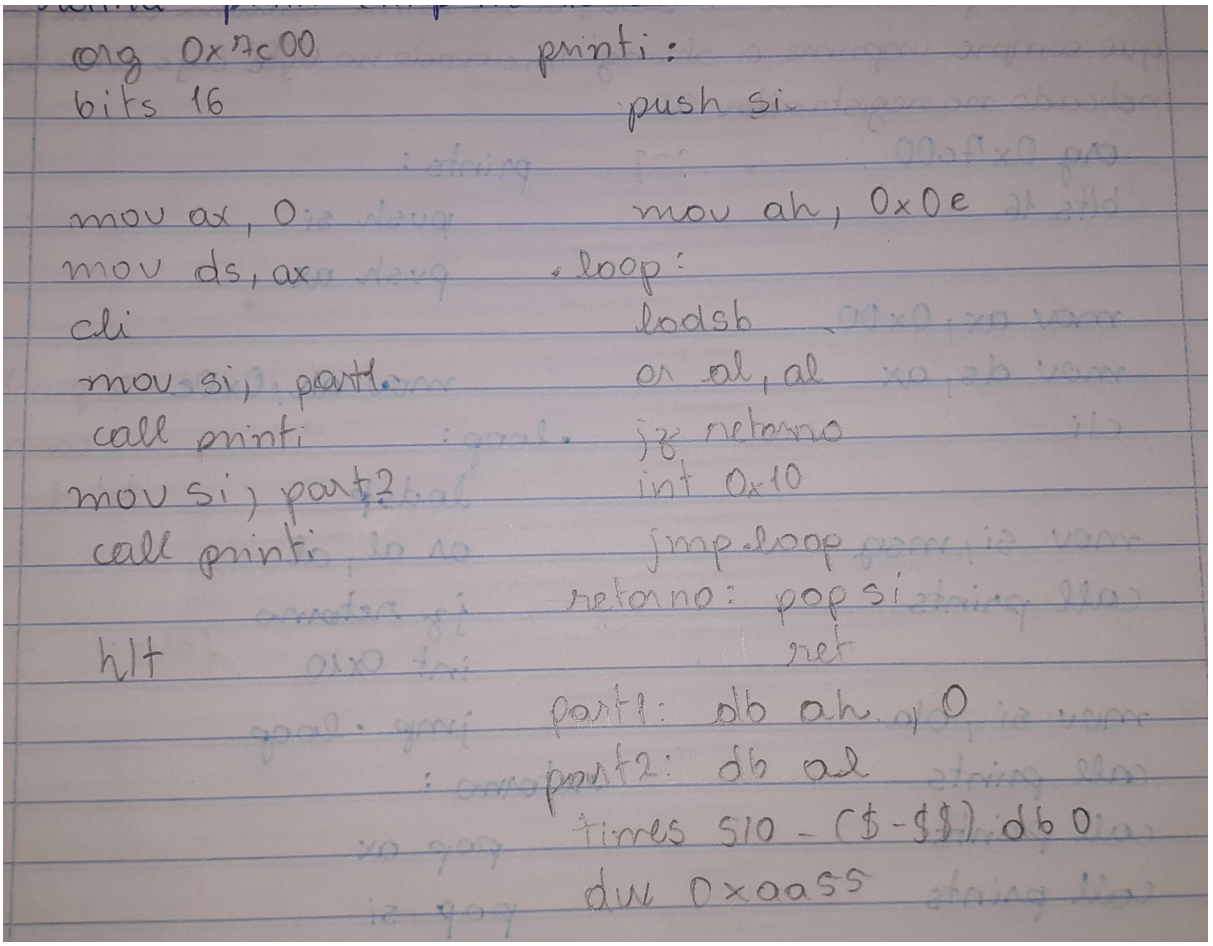
A)db 0x55
db 0xaa
B)
C) cmp al,al
D) db 'o','l',225,' m','u','n','d','o',33
E) db 'h','e','l','l','o',13,10,'w','o','r','l','d'

Aula 24
A)



Aula 25

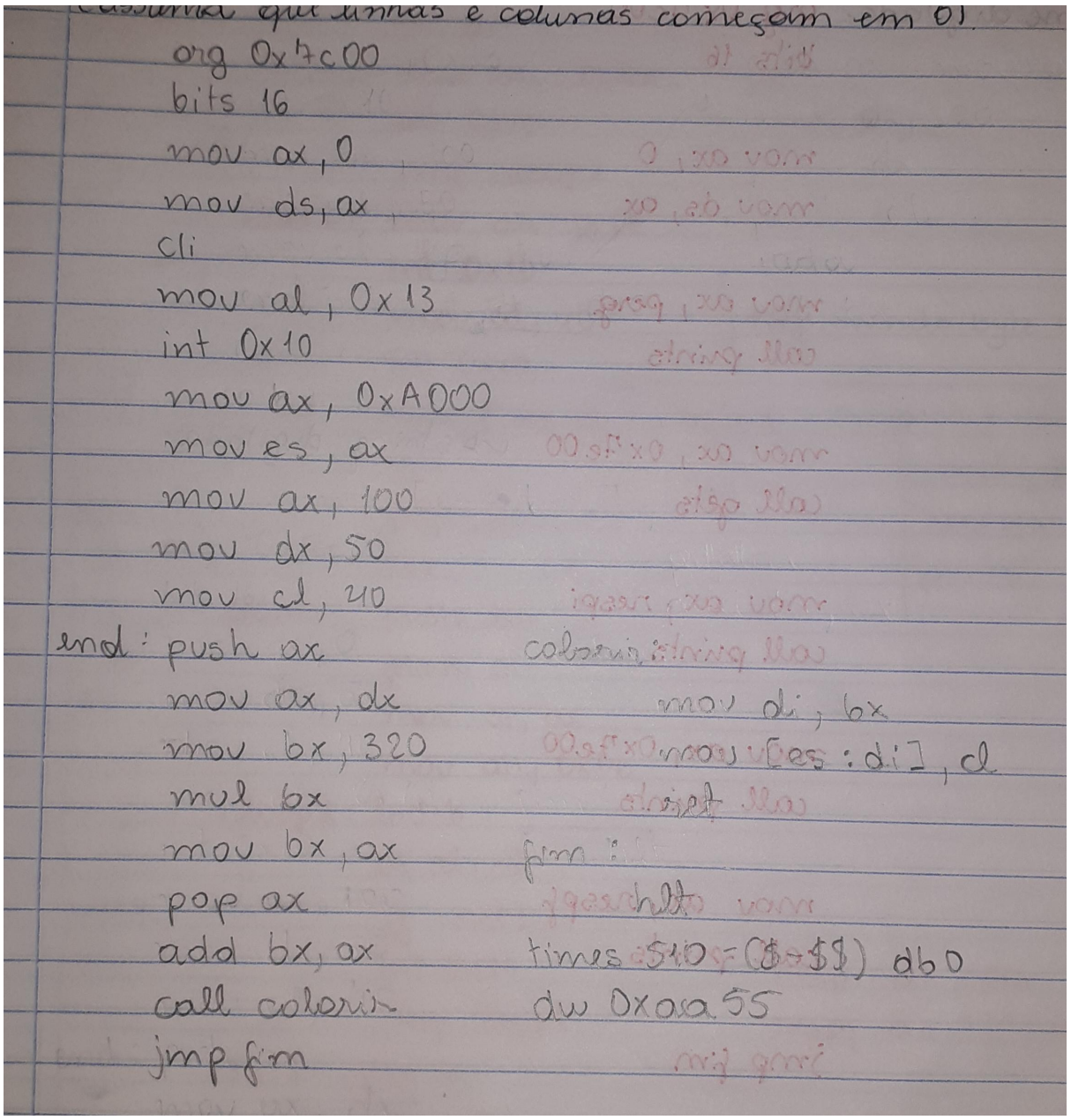
A)



```
org 0x7c00
bits 16

mov ax, 0
mov ds, ax
cli
mov si, part1
call printi
mov si, part2
call printi

hlt

printi:
    push si
    mov ah, 0x0e
.loop:
    lodsb
    or al, al
    jz retorno
    int 0x10
    jmp .loop
retorno: pop si
    ret
part1: db ah, 0
part2: db al
times 510 - ($-$$) db 0
dw 0xaa55
```

Aula 26

A)

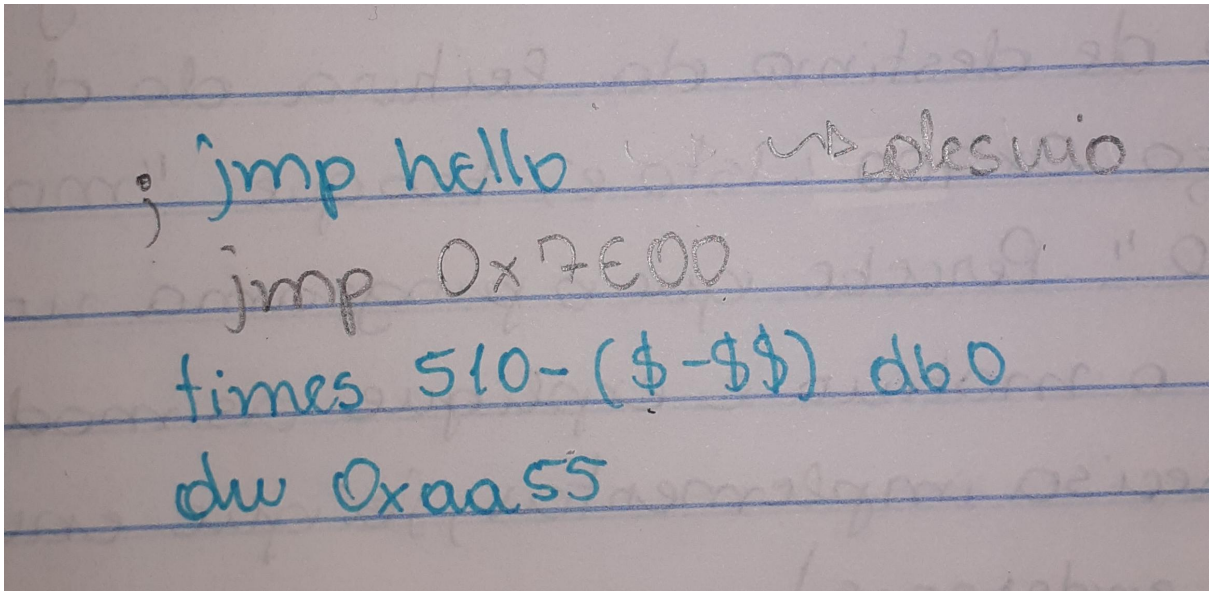


```
; assumindo que linhas e colunas começam em 0!
org 0x7c00
bits 16
mov ax, 0
mov ds, ax
cli
mov al, 0x13
int 0x10
mov ax, 0xA000
mov es, ax
mov ax, 100
mov dx, 50
mov cl, 40
end: push ax
mov ax, dx
mov bx, 320
mul bx
mov bx, ax
pop ax
add bx, ax
call colorir
jmp fim
```

```
colorir:
mov di, bx
mov [es:di], cl
fim:
jmp $
times 510-($-$$) db 0
dw 0xaa55
```

Aula 27
A)

Aula 28
A)



Aula 29

Aula 30