Lógica, Deslocamento, Rotação e Pilhas

João Marcelo Uchôa de Alencar

4 de maio de 2023

1/42

2/42

Introdução

- Já vimos os operadores lógicos "e" (&&), "ou" (||) e "negação" (!).
- Entretanto, muitas vezes precisamos fazer comparações *bit* a *bit*.
- Quando interagimos com um dispositivo externo, muitas vezes apenas um único *bit* precisa ser verificado ou configurado.
- As linguagens C/C++ já tem funcionalidades para este tipo de manipulação.
- Em MASM, vamos usar instruções de **lógica, deslocamento** e **rotação** para manipular *bits* em registradores e posições de memória.
- De brinde, vamos falar sobre a pilha.

3/42

Instruções Lógicas

Manipulação de *bits*

Ativar (*set*), Limpar (*clear*), Testar (*test*) e Alternar (*toggle*).

Operação	Lógica
Ativar	OR
Limpar	AND
Testar	AND
Alternar	XOR

Operadores em C

& é o AND *bit* a *bit* e | é o OR *bit* a *bit*.

4/42

Instruções Lógicas

- A variável *flag* contém vários *bits* resultantes de um dispositivo de E\S.
- A variável máscara *maskit* tem um *bit* específico em 1, os outros em 0.

```
if (flag & maskit)
        count++;
```

```
  01101110
  00000100
& ————————
  00000100
```

- Um disco pode indicar o final de leitura de um arquivo (EOF) configurando o terceiro *bit* como 1.
- No exemplo, são poucos *bits*, mas na prática é uma palavra na memória ou um registrador.
- Após aplicação do &, qualquer valor não zero indica a operação completada.

5/42

Instruções Lógicas

- Usando diretivas, não é muito diferente em MASM.
- O símbolo & representa uma instrução, logo não pode ter duas referências à memória.

```
mov eax, flag
.if eax & maskit
inc count
.endif
```

- O & na diretiva é equivalente à instrução *and*.
- Podemos então usar *cmp* e as instruções de desvio.

```
if01:    mov eax, flag
         and eax, maskit
         jz endif01
then01:  inc count
endif01: nop
```


6/42

Instruções Lógicas

Instruções AND	Instruções OR	Instruções XOR	Instruções NOT
and reg, reg	or reg, reg	xor reg, reg	not reg
and reg, imm	or reg, imm	xor reg, imm	not mem
and reg, mem	or reg, mem	xor reg, mem	
and mem, reg	or mem, reg	xor mem, reg	
and mem, imm	or mem, imm	xor mem, imm	

- Usamos instruções como *jz* no lugar de instruções como *jne*, *je*, etc.
- No caso, é mais importante averiguar o valor dos *bits* em *eflags* do que o valor numérico.

7/42

Instruções Lógicas

- É possível um meio termo, usar diretivas e instruções lógicas.
- O operador ZERO? permite consultar *eflags*.

```
mov eax, flag
and eax, maskit
.if !ZERO?
inc count
.endif
```

- No lugar de *maskit*, poderíamos usar um valor imediato, constante.
- Fica mais claro se a constante estiver em binário ou hexadecimal.

8/42

Instruções Lógicas

- O número binário deve ter *b* ao final.
- Se for preciso representar todos os 32 *bits*, podemos usar números hexadecimal terminando em *h*.

```
.if flag & 00000010b
inc count
.endif
```

```
and flag, 00000010b
if02: jz endif02
then02: inc count
endif02: nop
```

```
and flag, 00000010b
.if !ZERO?
inc count
.endif
```

- Os *bits* não representados estão em 0.

9/42

Instruções Lógicas

Importante!!!

Ao testar um *bit* individual, os outros são zerados.

- A instrução *and*, por exemplo, não apenas testa como também pode ser usada para *limpar*.
- A instrução *test* em MASM pode ser usada para testar sem limpar, mas não funciona em todos os processadores.

Ativar um *bit*

Enquanto *and* pode zerar os *bits*, *or* pode ativar (colocar em 1) um valor que estava em 0.

```
flag = flag | maskit;
```

```
mov eax, flag
or eax, maskit
mov flag, eax
```


10/42

Instruções Lógicas

- O padrão de *bits* para caracteres ASCII de letras maiúsculas sempre tem o *bit* 5 igual a 0.
- As minúsculas tem o *bit* 5 igual a 1.
- S é igual a 53_{16} ou 01010011_2.
- s é igual a 73_{16} ou 01110011_2.
- Aplicando a máscara 00100000_2 podemos fazer a conversão S para s. Assumindo *letter* como *byte*:

  ```
  or letter, 00100000b
  ```

- A conversão inversa pode ser feita por:

  ```
  and letter, 11011111b
  ```


11/42

Instruções Lógicas

- O ou exclusivo (*xor*) em C é feito pelo ^, mas só funciona *bit* a *bit*:

```
flag = flag ^ maskit;
```

- Em MASM:

```
mov eax, flag
xor eax, maskit
mov flag, eax
```

- A instrução *xor* pode alternar um único *bit* de 0 para 1 e vice-versa.

12/42

Instruções Lógicas

```
xor eax, eax
```

- Na *xor*, 1 *xor* 1 dá 0.
- É uma maneira de limpar os *bits* em 1.
- *xor eax, eax* é mais rápido que *mov eax, 0*.

13/42

Instruções de Deslocamentos Lógicos

- Muitas vezes, desejamos trabalhar com mais de um *bit* de uma vez.
- Instruções de deslocamento ou rotação são mais eficientes nesses casos.
- Em C, temos $<<$ (esquerda) e $>>$ (direita). Portanto, se $num == 2$:

```
num = num << 3;
```

- faz com que $num == 16$.
- 2_{10} é 00000010_2. Deslocar os *bits* em três posições à esquerda daria 00010000_2, ou seja, 16_{10}.
- Usando máscaras, as instruções lógicas e deslocamentos, podemos construir uma palavra com qualquer padrão de *bits* que desejarmos.

14/42

Instruções de Deslocamentos Lógicos

Devemos mover os *bits* na máscara ou no registrador\memória a ser testado? Depende da aplicação.

- Se o registrador ou memória não for ser mais utilizado, podem ser alterado diretamente.
- Caso contrário, é melhor criar uma máscara e preservar os valores originais.
- Uma opção é salvar o conteúdo em uma localização temporária, alterar os dados e depois mover os valores originais de volta.
- Também é possível fazer as alterações na localização temporária.
- Uma última alternativa é desfazer as alterações aplicando instruções inversas.

Veremos que a pilha pode ser usada como localização temporária.

15/42

Instruções de Deslocamentos Lógicos

As instruções de deslocamento não consideram *bit* de sinal.

Instruções Deslocamento à Esquerda	Instruções Deslocamento à Direita
shl reg, cl	shr reg, cl
shl reg, imm	shr reg, imm
shl mem, cl	shr mem, cl
shl mem, imm	shr mem, imm

- Usamos registradores do tipo *l* (1 *byte*) para informar a quantidade deslocamentos pois não faz sentido deslocamentos maiores de 32.
- Processadores antigos só aceitavam constante de valor 1. Os atuais aceitam qualquer número.

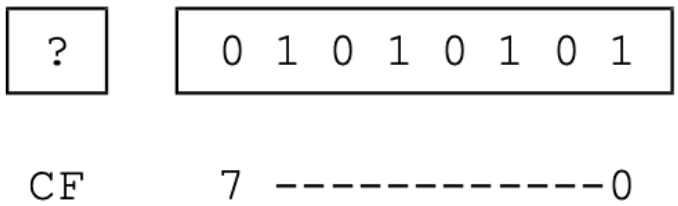
16/42

Instruções de Deslocamentos Lógicos

Considere a instrução:

```
mov al, 85
```

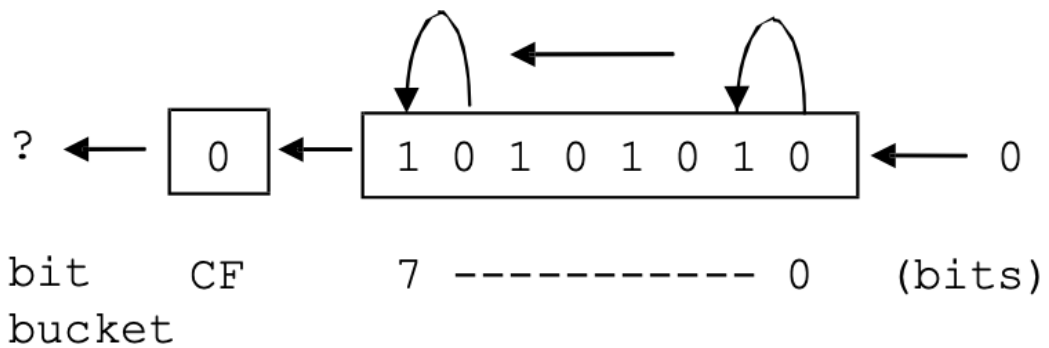
Lembrando que *al* é um registrador de 8 *bits*:



Se executarmos:

```
shl al, 1
```

teríamos:



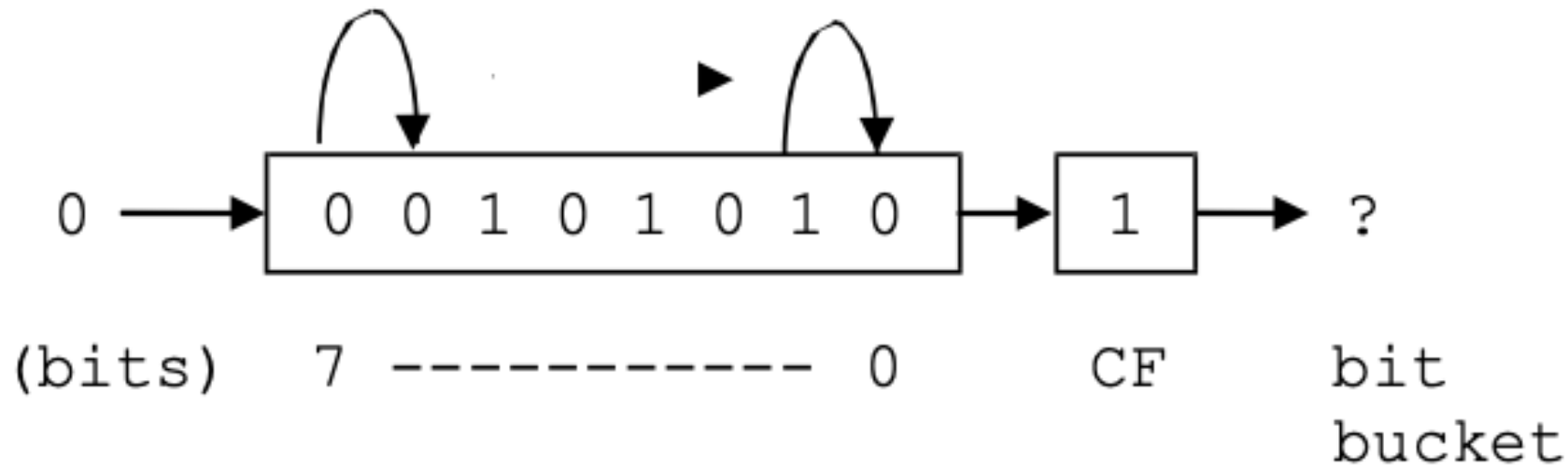
17/42

Instruções de Deslocamentos Lógicos

Se executarmos:

```
shr al, 1
```

teríamos:



18/42

Instruções de Deslocamentos Lógicos

- Considere que cada *bit* no registrador *al* representa um dispositivo.
- Se um *bit* for 1, o dispositivo está ligado. Se for 0, está desligado.
- Assuma que os dados originais devem ser mantidos.
- Como determinar quantos dispositivos estão ligados?

19/42

Instruções de Deslocamentos Lógicos

```
mov count, 0 ; initialize count to zero
mov ecx, 8 ; initialize loop counter to zero
mov temp, al ; save al in temp
.repeat
mov ah, al ; mov data in al to ah for testing
and ah, 00000001b ; test bit position zero
.if !ZERO? ; is the bit set?
inc count ; yes, count it
.endif
shr al, 1 ; shift al right one bit position
.untilcxz
mov al, temp ; restore al from temp
```


20/42

Instruções de Deslocamentos Lógicos

A instrução *test* realiza o *and* e atualiza *eflags*, mas mantém o registrador intacto.

```
mov count, 0 ; initialize count to zero
mov ecx, 8 ; initialize loop counter to zero
mov temp, al ; save al in temp
.repeat
test al, 00000001b ; test bit position zero
.if !ZERO? ; is the bit set?
inc count ; yes, count it
.endif
shr al, 1 ; shift al right one bit position
.untilcxz
mov al, temp ; restore al from temp
```


21/42

Instruções de Deslocamento Aritméticas

- Instruções que levam em consideração o *bit* de sinal.

Instruções de Deslocamento Aritméticas à Esquerda	Instruções de Deslocamento Aritméticas à Direita
sal reg, cl	sar reg, cl
sal reg, imm	sar reg, imm
sal mem, cl	sar mem, cl
sal mem, imm	sar mem, imm

- A instrução *sal* funciona da mesma forma que a instrução *shl*, ela existe por uniformidade.
- A instrução *sar*, no lugar de colocar 0 no *bit* mais à esquerda, copia o *bit* que estava lá antes.
- Números negativos em complemento de 2 continuarão negativos.

22/42

Instruções de Deslocamento Aritméticas

Por simplicidade, assuma palavras de 8 *bits*:

- Considere que a variável *number* contém 5_{10} como 00000101_2.
- Deslocamento em um *bit* à esquerda, o resultado seria 00001010_2, que é o número 10_{10}, que é o mesmo que $5x2$.
- Mais um deslocamento em um *bit* à esquerda, o resultado seria 00010100_2, que é o número 20_{10}, que é o mesmo que $5x2^2$.
- Em outras palavras, deslocamentos à esquerda equivalem a multiplicar por uma potência de 2, deslocamentos à direita equivalem a dividir por uma potência de 2.

23/42

Instruções de Deslocamento Aritméticas

Agora considere números negativos:

- -4_{10} é 11111100_2, deslocado 1 *bit* à direita com *shr*, daria 0111110_2, que é 125_{10}.
- Usando a instrução *sar*, teríamos 11111110_2, pois no lugar de 0, copiaríamos o 1 que estava na posição antes do deslocamento.
- 11111110_2 é -2_{10}, que é correto.
- Na multiplicação, tanto *sal* quanto *shl* produzem o mesmo resultado, mas é bom usar *sal* quando o objetivo for mesmo multiplicar por potência de 2.

24/42

Instruções de Deslocamento Aritméticas

Como implementaríamos usando instruções de deslocamento?

```
product = num1 * 8;
```

Assumindo palavras de 32 *bits* e a instrução *sal*

```
mov eax, num1 ; load eax with num1
sal eax, 3 ; multiply by 8
mov product, eax ; store eax in product
```

- Veja que *num1* não deve ter seu valor alterado.
- Também usamos 3 como operando em *sal* porque $2^3 = 8$.

25/42

Instruções de Deslocamento Aritméticas

Como implementaríamos usando instruções de deslocamento?

```
answer = amount / 4
```

- Se *amount* for positivo, podemos usar tanto *slr* quanto *sar*.
- Mas como não podemos afirmar com certeza, o correto é usar *sar*.

```
mov eax, amount
sar eax, 2 ; divide by 4
mov answer, eax
```

- Sempre que o objetivo for multiplicar ou dividir, use *sal* ou *sar*.
- Por que usar *sal* e *sar* no lugar de *imul* e *idiv*? Porque são mais **rápidas**.

26/42

Instruções de Rotação

- Nas instruções de deslocamento, os *bits* que "caem"em um dos extremos são perdidos.
- Nas instruções de rotação, os *bits* de um extremo são carregados no outro lado.

Instruções de Rotação à Esquerda	Instruções de Rotação à Direita
rol reg, cl	ror reg, cl
rol reg, imm	ror reg, imm
rol mem, cl	ror mem, cl
rol mem, imm	ror mem, imm

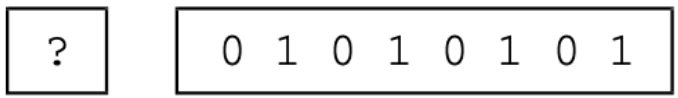
27/42

Instruções de Rotação

Considere a instrução:

```
mov al, 85
```

Lembrando que *al* é um registrador de 8 *bits*:

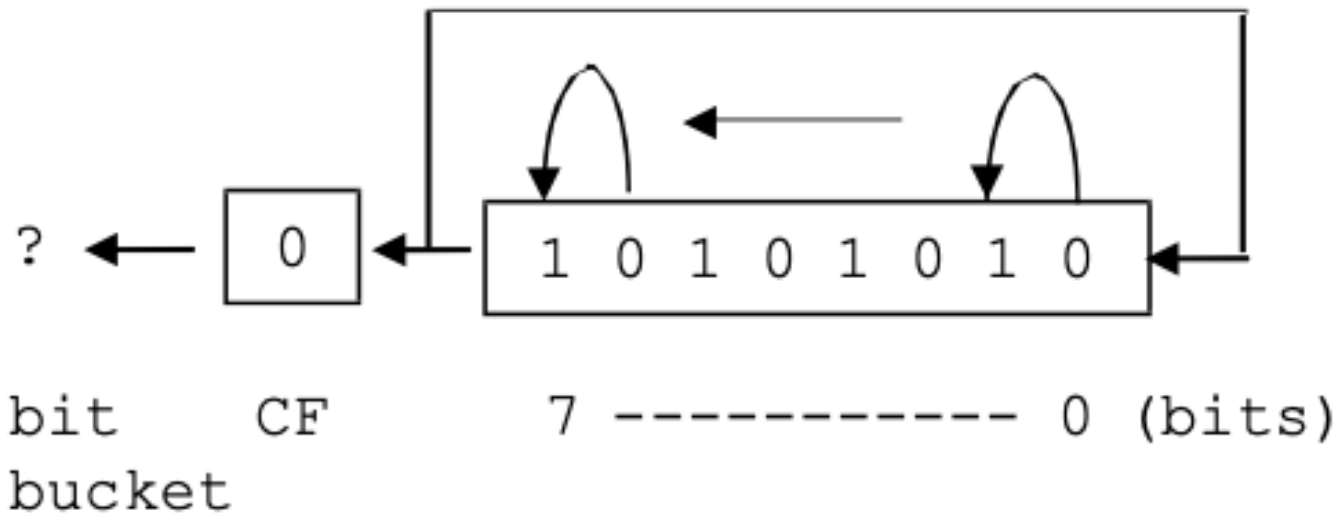



```
CF      7 ------------0
```

Se executarmos:

```
rol al, 1
```

teríamos:



28/42

Instruções de Rotação

- Se o número de rotações for igual ao número de *bits* do registrador ou memória, o estado original é restaurado.
- Você não precisa salvar o valor original se for inspecionar cada *bit*.

```
mov count, 0 ; initialize count to zero
mov ecx, 8 ; initialize loop counter to zero
.repeat
test al, 00000001b ; test bit position zero
.if !ZERO? ; is the bit set?
inc count          ; yes, count it
.endif
rol al, 1 ; shift al left one bit position
.untilcxz
```


29/42

Operações de Pilha

- Estrutura de dados **pilha** e métodos *push* e *pop*.
- LIFO (*last in first out*).
- Por ser um conceito muito importante em computação, a maioria dos processadores incluem instruções para manipulação de pilha.
- *.stack 100h* reserva 256 *bytes* para a pilha.
- Registradores de 16, 32 ou 64 *bits* funcionam com instruções *push* e *pop*.
- Se o registrador *al* precisa ser colocado na pilha, os registradores *ax*, *eax* ou *rax* precisariam ser utilizados.

30/42

Operações de Pilha

Instruções de *Push*	Instruções de *Pop*
push reg	pop reg
push mem	pop mem
push imm	

- Alguns processadores otimizam a pilha, mesmo sendo na memória, seu acesso é mais rápido do que outros endereços da memória.
- A principal vantagem é conveniência, você não precisa declarar valores temporários.
- A pilha é muito útil para guardar o conteúdo original de um registrador ou memória antes da manipulação dos *bits*.

31/42

Operações de Pilha

```
push eax
mov count, 0 ; initialize count to zero
mov ecx, 8 ; initialize loop counter to zero
.repeat
test al, 000000001b ; test bit position zero
.if !ZERO? ; is the bit set?
inc count ; yes, count it
.endif
shr al, 1 ; shift al right one bit position
.untilcxz
pop eax
```


32/42

Operações de Pilha

- Se mais de um registrador ou memória precisar ser salvo, a ordem das instruções é importante.

```
w = x  / y  -  z;
```

- Lembre que x, y e z não devem ter seus valores alterados, e que os valores anteriores dos registradores devem ser preservados.

```
push eax
push edx
mov eax, x
cdq
idiv y
sub eax, z
mov w, eax
pop edx
pop eax
```


33/42

Operações de Pilha

- Tanto *eax* quando *edx* foram colocados na pilha, pois *cdq* e *idiv* também alteram *edx*.
- *edx* foi colocado por último, então é o primeiro a ser retirado.
- Usar muito a pilha pode deixar o código confuso:
  - Evite deixar os *pops* e os *pushs* correspondentes muito distantes entre si.
  - Use rótulos ou comentários para identificar pares de *push*/*pop*.

34/42

Trocando usando Registradores, a Pilha e a Instrução *xchg*

```
// Em C
temp = num1;
num1 = num2;
num2 = temp;
```

```
; Em MASM
mov eax, num1
mov temp, eax
mov eax, num2
mov num1, eax
mov eax, temp
mov num2, eax
```

```
; Usando edx
; como intermediário
mov edx, num1
mov eax, num2
mov num1, eax
mov num2, edx
```

```
; Reajustando
; para legibilidade
mov eax, num1
mov edx, num2
mov num1, edx
mov num2, eax
```


35/42

Trocando usando Registradores, a Pilha e a Instrução *xchg*

- Podemos fazer as trocas usando a pilha.
- Não precisamos de registradores extra.

```
push num1
push num2
pop num1
pop num2
```

- Uma outra maneira é usar a instrução *xchg*.

36/42

Trocando usando Registradores, a Pilha e a Instrução *xchg*

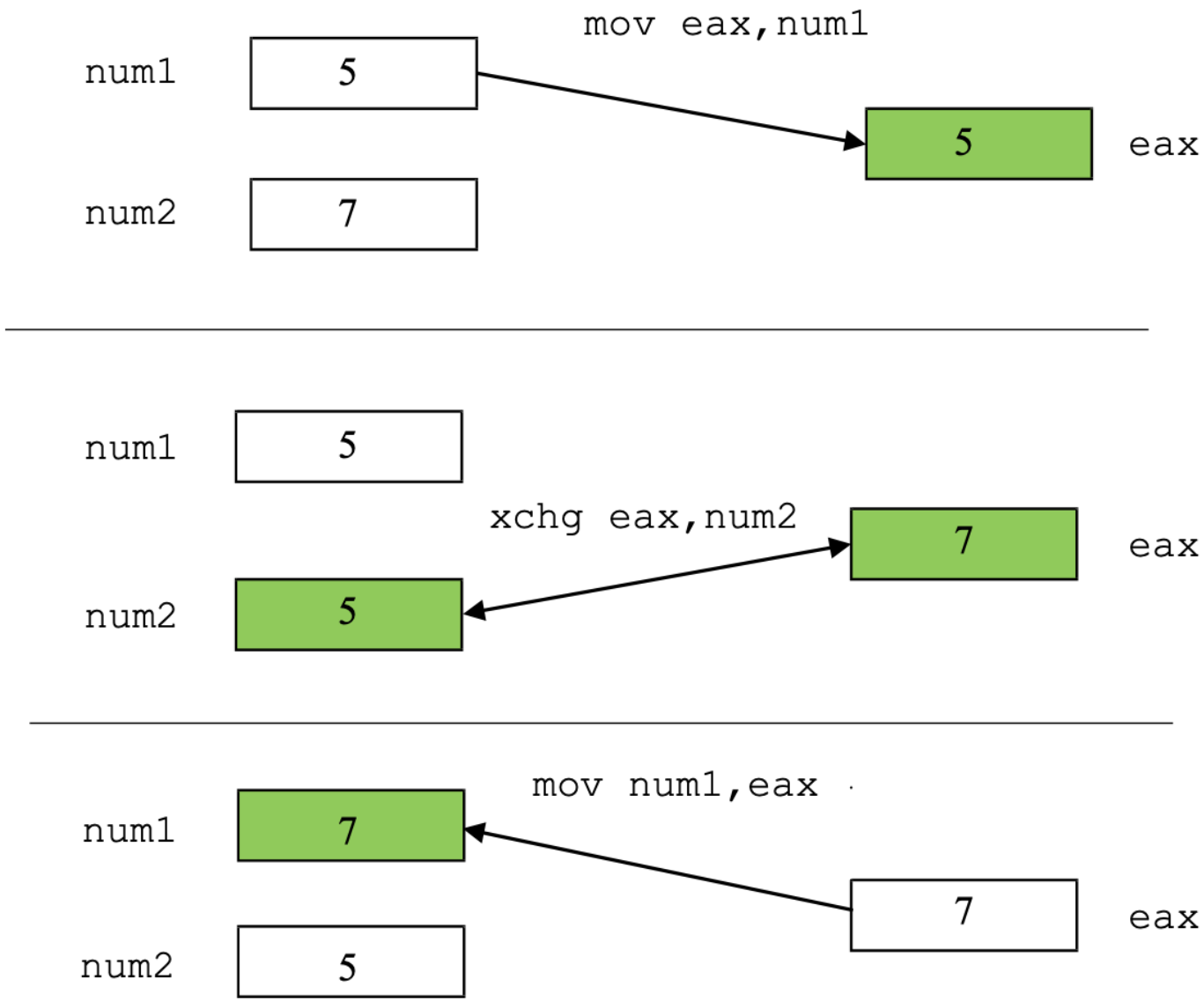
- Novamente, não temos uma instrução para trocar duas regiões de memória.
- Mas a troca agora só precisa de 3 instruções.

Instruções de *xchg*
xchg reg, reg
xchg reg, mem
xchg mem, reg

```
mov eax, num1
xchg eax, num2
mov num1, eax
```


37/42

Trocando usando Registradores, a Pilha e a Instrução *xchg*



38/42

Programa Completo

- Simular o funcionamento de um *scanner* OCR.
- Vários erros podem ocorrer durante o processamento de um lote de documentos alimentados ao *scanner*.

Bit	Mensagem	Significado
0	Documento Curto	Documento menor do que o antecipado
1	Documento Longo	Documento maior do que o antecipado
2	Alimentação Próxima	Documentos muito próximos
3	Alimentação Múltipla	Documentos em colisão
4	Desvio Excessivo	Documento mal ajustado
5	Documento Ausente	Documento não está no *scanner*
6	Documento Obstruído	Documento preso dentro do *scanner*
7	Erro Desconhecido	Não se sabe a razão do erro

39/42

Programa Completo

- O programa tem um *byte* que indica a presença de um ou mais erros.
- *Document Status Byte* (DSB).
- O usuário irá fornecer um *byte* em hexadecimal ao programa.
- O programa irá informar quais erros aconteceram.
- Poderíamos usar *ifs* aninhados ou simular *switch*/*case*, o uso de diretivas *ifs* não aninhadas é mais simples.

40/42

Resumo

- O **ou inclusivo** inclui o caso em que ambos operandos são verdadeiros e o resultado é verdade, enquanto o **ou exclusivo** exclui esse caso e o resultado é falso quando ambos operandos são verdadeiros.
- Para configurar, limpar, testar ou alternar *bits* use as instruções *or*, *and* ou *xor*, respectivamente.
- Se os dados forem ser reutilizados, certifique-se de salvá-los ao usar as instruções *shl* e *shr*.
- Como efeito colateral de outras operações, um registrador ou localização de memória pode ser limo para zero usando as instruções *shl* ou *shr*.

41/42

Resumo

- Para multiplicar ou dividir por potências de 2, use *sal* e *sar*.
- Se um padrão de *bits* sofrer rotação no mesmo sentido na mesma quantidade do tamanho do registrador ou memória, não precisa ser restaurado.
- Lembre-se que a pilha é LIFO.
- Use rótulos, comentários para parear *pushs* e *pops*.
- Para trocar dados, a melhor opção é a instrução *xchg*.

42/42