

Programação de Baixo Nível em C

Prof. Vandermi Silva

Baseado nas notas de aula do
Prof. Raimundo Barreto



Introdução



- Linguagem C é de alto nível, contudo
... também possui certos recursos de baixo nível (*Assembly*
ou linguagem de máquina)
- Por exemplo:
 - Armazenar valores de variáveis em registradores
 - Manipulação de bits individuais de uma palavra (*word*)
 - Rolar (*shift*) bits para a direita ou esquerda Bits podem ser invertidos
 - Mascarados (extração seletiva de valores)

Operações Bitwise



- Algumas aplicações precisam **manipular bits individuais** de uma palavra de memória (**word**)
- A linguagem **assembly** é normalmente necessária nesses tipos de operação
- A **linguagem C** contém vários operadores especiais para que essas operações sejam executadas **fácil** e eficientemente
- São divididos em **três** categorias: operador complementar, operadores *bitwise* lógicos e operadores de deslocamento (*shift*)

Operador Complementar



- O operador complementar (~) é um operador unário que inverte os bits de seu operando, ou seja, 1 se torna 0 e 0 se torna 1.
- Exemplo (considerando palavra de 16 bits)
 - Qual será o complemento de 0x7FF?
 - Representação em bits é: 0000 0111 1111 1111
 - Complemento é: 1111 1000 0000 0000
 - Que corresponde à: 0xF800
 - Ou seja: $\sim 0x7FF = 0xF800$

Operador Complementar (32 bits)



- Outros exemplos
 - $\sim 0xC5 = 0xFFFFFFFF3A$
 - $\sim 0x1111 = 0xFFFFEEEE$
 - $\sim 0xFFFF = 0xFFFF0000$
 - $\sim 0x5B3C = 0xFFFFA4C3$

```
0000 0000 0000 0000 0000 0000 1100 0101 = 0x000000C5
1111 1111 1111 1111 1111 1111 0011 1010 = 0xFFFFFFFF3A
```

Operadores Bitwise Lógicos



- Existem 3 operadores bitwise lógicos
 - Bitwise “e” (&)
 - Bitwise “ou exclusivo” (^)
 - Bitwise “ou” (|)
- Cada um desses operadores requer dois operandos inteiros
- As operações são executadas em cada par de bits, a partir dos bits menos significativos

Operadores Bitwise Lógicos



b1	b2	b1 & b2	b1 ^ b2	b1 b2
1	1	1	0	1
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

Operadores Bitwise Lógicos



- Suponha que a e b sejam variáveis inteiras sem sinal de valores $0x6DB7$ e $0xA726$

- Qual o valor de $\sim a$?

– $a = 0110\ 1101\ 1011\ 0111 = 0x6DB7$

– $\sim a = 1001\ 0010\ 0100\ 1000 = 0x9248$

Operadores Bitwise Lógicos



- Suponha que a e b sejam variáveis inteiras sem sinal de valores $0x6DB7$ e $0xA726$
- Qual o valor de $\sim b$?

$b = 1010 \ 0111 \ 0010 \ 0110 = 0xA726$

$\sim b = 0101 \ 1000 \ 1101 \ 1001 = 0x58D9$

Operadores Bitwise Lógicos



- Suponha que a e b sejam variáveis inteiras sem sinal de valores $0x6DB7$ e $0xA726$

- Qual o valor de $a \& b$?

$a = 0110\ 1101\ 1011\ 0111 = 0x6DB7$

$b = 1010\ 0111\ 0010\ 0110 = 0xA726$

$a \& b = 0010\ 0101\ 0010\ 0110 = 0x2526$

Operadores Bitwise Lógicos



- Suponha que a e b sejam variáveis inteiras sem sinal de valores $0x6DB7$ e $0xA726$

- Qual o valor de $a \wedge b$?

$a = 0110\ 1101\ 1011\ 0111 = 0x6DB7$

$b = 1010\ 0111\ 0010\ 0110 = 0xA726$

$a \wedge b = 1100\ 1010\ 1001\ 0001 = 0xCA91$

Operadores Bitwise Lógicos



- Suponha que a e b sejam variáveis inteiras sem sinal de valores $0x6DB7$ e $0xA726$
- Qual o valor de $a | b = 0xEFB7$

$a = 0110\ 1101\ 1011\ 0111 = 0x6DB7$

$b = 1010\ 0111\ 0010\ 0110 = 0xA726$

$a | b = 1110\ 1111\ 1011\ 0111 = 0xEFB7$

Máscara

- Suponha que $a = 0x6DB7$ e que queiramos **extrair os 6 bits mais a direita** de a e assinalar à variável b . Como fazer isso?
- Observe que, a partir de uma representação binária original, queremos transformá-la em outra representação binária
- Para tanto, vamos utilizar o conceito de **máscara** e de uma operação bitwise lógica
 - O primeiro operando é a representação original
 - O segundo operando é a máscara



Máscara



- Assim, parte da representação original será “mascarada” no resultado final
- Existem diferentes tipos de operações com máscaras.
- Nesse exemplo, uma parte da representação binária é copiada para o resultado, enquanto que o restante dos bits é preenchido com **0**
- O operador **&** será usado nesse tipo de operação

Máscara



- A operação deve ser: $b = a \ \& \ 0x3F$
- Assim, $b=0x37$
- A validade do resultado pode ser vista examinando as representações binárias
 - $a = 0110 \ 1101 \ 1011 \ 0111 = 0x6DB7$
 - $M = 0000 \ 0000 \ 0011 \ 1111 = 0x3F$
 - $b = 0000 \ 0000 \ 0011 \ 0111 = 0x37$
- Observe que a máscara evita que os 10 bits mais à esquerda sejam copiados de a para b .

Máscara

- Suponha, novamente, que a seja uma variável inteira sem sinal de valor $0x6DB7$. Agora, **extraia os 6 bits mais à esquerda** desse valor e assinale à variável inteira sem sinal b . Assinale 0s aos 10 bits mais à *direita* de b .
- Para executar essa operação podemos escrever a expressão bitwise $b = a \ \& \ 0xFC00$
- Assim, a constante $0xFC00$ servirá como máscara
- O valor de b será $0x6C00$.



Máscara



- $b = a \ \& \ 0xFC00$
- A validade desse resultado pode ser verificada examinando-se as representações binárias correspondentes
 - $a = 0110 \ 1101 \ 1011 \ 0111 = 0x6DB7$
 - $M = 1111 \ 1100 \ 0000 \ 0000 = 0xFC00$
 - $b = 0110 \ 1100 \ 0000 \ 0000 = 0x6C00$
- A máscara agora bloqueia os 10 bits mais à direita de a .

Máscara

- Um outro tipo de operação de máscara permite que **uma parte** de uma dada representação binária seja **copiada** em uma nova palavra enquanto o restante da nova palavra é preenchido com **1s**.
- O operador bitwise ou (|) é usado para isso
- Note a diferença entre essa operação e a anterior



Máscara

- Suponha que a variável a seja uma inteira sem sinal de valor $0 \times 6DB7$, como antes. Transforme a sua correspondente representação binária em uma outra representação binária na qual os 8 bits mais à direita são todos 1s e os 8 bits mais à esquerda permanecem com seus valores originais. Assinale essa representação binária à variável inteira sem sinal b .



Máscara



- A operação será:

$$-b = a \mid 0xFF$$

- Vamos examinar o resultado:

$$-a = 0110 \ 1101 \ 1011 \ 0111 = 0x\mathbf{6D}B7$$

$$-M = 0000 \ 0000 \ 1111 \ 1111 = 0xFF$$

$$-b = 0110 \ 1101 \ 1111 \ 1111 = 0x\mathbf{6D}FF$$



Máscara

- Suponha, agora, que desejamos transformar a representação binária de a em outra representação binária, na qual os 8 bits mais à esquerda são todos 1s e os 8 bits mais à direita permanecem com os seus valores originais.



Máscara



- A operação será:

$$- b = a \mid 0xFF00$$

- Vamos examinar o resultado:

$$- a = 0110 \ 1101 \ 1011 \ 0111 = 0x6D\mathbf{B7}$$

$$- M = 1111 \ 1111 \ 0000 \ 0000 = 0xFF00$$

$$- b = 1111 \ 1111 \ 1011 \ 0111 = 0xFF\mathbf{B7}$$

Máscara

- Suponha que a seja uma variável inteira sem sinal cujo valor é $0x6DB7$, como nos exemplos anteriores. Vamos agora **inverter** os 8 bits mais à *direita* e **preservar** os 8 bits mais à *esquerda*. Essa nova representação binária será assinalada à variável inteira sem sinal b .
- Para isso vamos utilizar a expressão:
 - $b = a \wedge 0xFF$, portanto:
 - $b = 0x6D48$

Máscara



- Vamos verificar a validade:

– a = **0110 1101 1011 0111** = 0x6DB7

– M = 0000 0000 1111 1111 = 0xFF

– b = **0110 1101 0100 1000** = 0x6D48

Máscara

Se quisermos inverter os 8 bits mais à esquerda de a e manter os 8 bits mais à direita originais:

$$b = a \wedge 0xFF00$$

$$a = \text{0110 1101 1011 0111} = 0x6DB7$$

$$M = 1111 1111 0000 0000 = 0xFF00$$

$$b = \text{1001 0010 1011 0111} = 0x92B7$$

Máscara

- Suponha que a seja uma variável inteira sem sinal cujo valor é $0x6DB7$, como nos exemplos anteriores.
- Qual o resultado da expressão $a \wedge 0x4$?
 - inverterá o valor do bit número 2 (o terceiro bit a partir da direita) de a .
- Se essa operação for executada repetidamente, o valor de a será alternado entre $0x6DB7$ e $0x6DB3$
- Nesse caso, o terceiro bit será “ligado” e “desligado” alternadamente.



Máscara



- Vamos verificar a validade:

$$- a = 0110 \ 1101 \ 1011 \ 0\textcolor{red}{1}11 = 0x6DB7$$

$$- M = 0000 \ 0000 \ 0000 \ 0100 = 0x4$$

$$- b = 0110 \ 1101 \ 1011 \ 0\textcolor{red}{0}11 = 0x6D48$$

$$- M = 0000 \ 0000 \ 0000 \ 0100 = 0x4$$

$$- c = 0110 \ 1101 \ 1011 \ 0\textcolor{red}{1}11 = 0x6D48$$

Máscara

- E se eu quiser alterar alguns bits para um valor pré-estabelecido?
- Por exemplo, eu quero trocar os dois bits menos significativos para o valor “10”.
 - $B = ((B \& M1) | M2)$
 - Onde $M1=11111100$ e $M2=00000010$



Máscara



0xAF

1	0	1	0	1	1	1	1
---	---	---	---	---	---	---	---

B

0xFC

1	1	1	1	1	1	0	0
---	---	---	---	---	---	---	---

M₁

0xAC

1	0	1	0	1	1	0	0
---	---	---	---	---	---	---	---

B & M₁

0x02

0	0	0	0	0	0	1	0
---	---	---	---	---	---	---	---

M₂

0xAE

1	0	1	0	1	1	1	0
---	---	---	---	---	---	---	---

((B & M₁) | M₂)

Máscara

- Outro exemplo, eu quero trocar os bits 5,4 e 3 para o valor “101” e mantendo os outros bits intactos.
 - $B = ((B \& M1) | M2)$
 - Onde $M1=11100011$ e $M2=00010100$



Máscara



0xAF

1	0	1	0	1	1	1	1
---	---	---	---	---	---	---	---

B

0xE3

1	1	1	0	0	0	1	1
---	---	---	---	---	---	---	---

M₁

0xA3

1	0	1	0	0	0	1	1
---	---	---	---	---	---	---	---

B & M₁

0x14

0	0	0	1	0	1	0	0
---	---	---	---	---	---	---	---

M₂

0xB7

1	0	1	1	0	1	1	1
---	---	---	---	---	---	---	---

((B & M₁) | M₂)

Operadores de Deslocamento



- Os dois operandos bitwise shift são à esquerda (\ll) e à direita (\gg).
- Cada operador exige 2 operandos.
- O primeiro é um operando inteiro sem sinal que mostra a representação binária a ser “rolada”.
- O segundo é um inteiro sem sinal que indica o número de deslocamentos.
- Esse valor **não pode exceder** o número de bits associado ao tamanho da palavra.

Operadores de Deslocamento



Deslocando bits à esquerda

```
x = 1; // 0000 0001
```

```
x0 = (x << 0); // 0000 0001 Não deslocado
```

```
x1 = (x << 1); // 0000 0010
```

```
x2 = (x << 2); // 0000 0100
```

```
x3 = (x << 3); // 0000 1000
```

```
x4 = (x << 4); // 0001 0000
```

```
x5 = (x << 5); // 0010 0000
```

```
x6 = (x << 6); // 0100 0000
```

```
x7 = (x << 7); // 1000 0000
```

Operadores de Deslocamento



Deslocando bits à direita

```
x = 128; // 1000 0000
```

```
x0 = (x >> 0); // 1000 0000 Não deslocado
```

```
x1 = (x >> 1); // 0100 0000
```

```
x2 = (x >> 2); // 0010 0000
```

```
x3 = (x >> 3); // 0001 0000
```

```
x4 = (x >> 4); // 0000 1000
```

```
x5 = (x >> 5); // 0000 0100
```

```
x6 = (x >> 6); // 0000 0010
```

```
x7 = (x >> 7); // 0000 0001
```

Operadores de Deslocamento



- O operador mais à esquerda deslocará todos os bits do primeiro operando para a esquerda o número de posições indicado pelo segundo operando.
- Os bits mais à esquerda da representação original serão perdidos
- As posições mais à direita que ficarem vazias serão preenchidas com 0s



Operadores de Deslocamento

- Assumindo que a e b são **short int**:

$$b = a \ll 6$$

- Deslocará todos os bits de a **seis** posições à **esquerda** e o resultado será atribuído à variável b .

$$b = 0x6DC0$$

Serão
perdidos

bits para ver o resultado final

0	1	1	0	1	1	0	1	1	0	1	1	0	1	1	1
0	1	1	0	1	1	0	1	1	1	0	0	0	0	0	0

Operadores de Deslocamento



- O operador mais à *direita* desloca todos os bits do primeiro operando para a direita o número de posições indicado pelo segundo operando.
- Os bits mais à direita da representação original serão perdidos.
- As posições mais à esquerda que ficarem vazias serão preenchidas com 0s



Operadores de Deslocamento

- Exemplo:

- $b = a \gg 6$

- Deslocará todos os bits de a **seis** posições à **direita** e o resultado será atribuído à variável b .

- $b = 0x1B6$

- Vamos ver o resultado final

0	1	1	0	1	1	0	1	1	0	1	1	0	1	1	1
0	0	0	0	0	0	0	1	1	0	1	1	0	1	1	0

Serão
perdidos

Operadores Bitwise de Atribuição



Expressão	Equivalente	Valor Final
<code>a &= 0x7F</code>	<code>a = a & 0x7F</code>	0x37
<code>a ^= 0x7F</code>	<code>a = a ^ 0x7F</code>	0x6DC8
<code>a = 0x7F</code>	<code>a = a 0x7F</code>	0x6DFF
<code>a <<= 5</code>	<code>a = a << 5</code>	0xB6E0
<code>a >>= 5</code>	<code>a = a >> 5</code>	0x36D