

Operações com Bits

Aritmética com Bits

Como o computador manipula apenas bits, todas as operações matemáticas são realizadas cegamente sobre bits. O computador não sabe o que representam os bits armazenados em uma variável. Felizmente, a aritmética binária é muito semelhante à aritmética no sistema decimal, apenas que estaremos atuando na base 2 e não mais na base 10.

Soma

A adição de dois bits é simples:

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 1 \text{ e "vai 1" para o próximo dígito à esquerda}$$

A soma de dois bits “1” resulta em “10”, equivalente ao valor 2. Isto é uma situação parecida com aquela quando somamos dois dígitos e o resultado excede 10:

$$5 + 5 = 10$$

$$7 + 9 = 16$$

A operação “vai um” funciona da mesma maneira no sistema binário:

$$\begin{array}{r} \begin{array}{cccccc} 1 \uparrow & 1 \uparrow & 1 \uparrow & 1 \uparrow & 1 \uparrow & (vai\ um) \\ & 0 & 1 & 1 & 0 & 1 \\ + & 1 & 0 & 1 & 1 & 1 \\ \hline = & 1 & 0 & 0 & 1 & 0 & 0 \end{array} \end{array}$$

Neste exemplo: os dois números binários somados são 01101_2 (13 em decimal) e 10111_2 (23 em decimal). Começando pela coluna mais à direita, $1 + 1 = 10$. O um “vai para” o próximo dígito e o zero é escrito no resultado. A segunda coluna é somada: $1 + 0 + 1 = 10$. Novamente, “vai um” para o próximo dígito e zero é escrito no resultado. Na terceira coluna a soma é $1 + 1 + 1 = 11$. “Vai um” para o próximo dígito e o um é escrito no resultado. Procedendo desta forma, o resultado final será 100100_2 , que corresponde a 36 na base 10.

Subtração

A subtração funciona da mesma forma:

$$0 - 0 = 0$$

$$0 - 1 = 1 \text{ e empresta 1 do dígito à esquerda}$$

$$1 - 0 = 1$$

$$1 - 1 = 0$$

Para subtrair 10111_2 (23 em decimal) de 1101110_2 (110 em decimal):

converte 10111_2 para negativo, usando o deslocamento negativo: $-23 = -128 + 105$. O número 105 no sistema binário é 1101001_2 .

$$\begin{array}{r}
 \begin{array}{ccccccc}
 1 & 1 & 1 & & & & \\
 \uparrow & \uparrow & \uparrow & & & & \\
 1 & 1 & 1 & 0 & 1 & 1 & 0 \\
 + & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\
 \hline
 = & 1 & 0 & 1 & 0 & 1 & 1 & 1
 \end{array}
 \end{array}
 \quad (vai\ um)$$

Mas espere! Para efetuar subtrações ainda teríamos que converter uma representação de um número positivo na representação do seu negativo, antes de realizar a soma final. Existe um algoritmo simples para fazer isso? Se não existir, todo o esquema de economia eliminando circuitos específicos para subtração ficaria invalidada. Felizmente, a resposta é “sim”, existe um algoritmo bem simples para obter a representação do negativo de um número positivo, a partir da sua representação positiva! Tente achar o algoritmo, antes de avançar na leitura.

O algoritmo é o seguinte:

1. Vamos lendo os bits da representação positiva, da direita para a esquerda.
2. Enquanto encontrarmos bits zero, eles são simplesmente copiados para a saída.
3. O primeiro bit um que encontrarmos também é copiado para a saída.
4. Depois que encontrarmos o primeiro bit um, os demais bits são todos trocados na saída, i.e., se lermos um bit 1, escrevemos 0 na saída, e vice-versa.

Por exemplo, o número +90 é representado por (em 8 bits) 0101 1010. Aplicando o algoritmo sobre essa seqüência de bits obtemos 1010 0110, que é exatamente a representação de -90.

O algoritmo também obtém a representação positiva, dada a representação negativa de um número. No caso anterior, aplicando o algoritmo sobre 1010 0110 recriamos 0101 1010.

E existem circuitos simples que realizam essas operações básicas sobre bits! Assim, mantemos a economia de circuitos no processador.

Examinaremos comandos em C para realizar essas operações logo em seguida.

Estas peculiaridades tornam as operações aritméticas extremamente simples de serem realizadas no computador. Além disso, encontrou-se formas para utilizar o mesmo circuito para subtração, multiplicação, divisão e resto! A simplicidade do circuito binário e sua capacidade de realizar todas as operações aritméticas foi o motivo para a popularização do sistema binário em praticamente todos os computadores.

Ainda existem calculadoras que utilizam outras bases numéricas, mas seu desenvolvimento é mais caro e trabalhoso, pois não apresentam as mesmas características vantajosas que o sistema binário.

Operações sobre bits

Os operadores aritméticos (soma, subtração, multiplicação, divisão e resto) interpretam uma seqüência de bits como a representação dos dígitos de um número binário. Mas

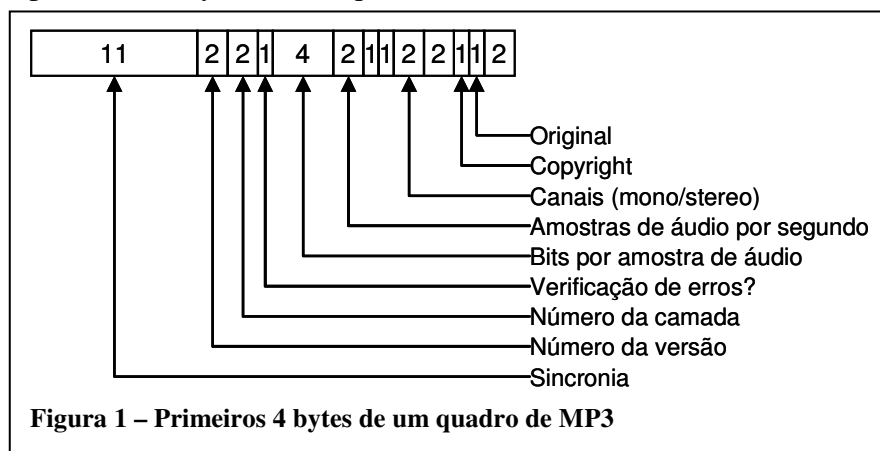
podemos também manipular diretamente os bits da sequência através de certos operadores lógicos.

Seqüências de Bits

Em aplicações de comunicação na internet ou na interação com dispositivos eletrônicos, os dados ocorrem como seqüências de bits que representam números, enumerações, estados ou outras composições específicas. Estas estruturas não necessariamente coincidem com os tipos de dados da linguagem C.

Exemplo

Um arquivo de músicas MP3 é composto por um grande número de seqüências de bits (chamadas de quadros ou *frames*). Cada quadro é decodificado de forma independente e depois transformado em sinal de áudio. Cada quadro inicia com bits que informam alguns parâmetros que controlam o decodificador. A Figura 1 mostra as principais informações contidas nos primeiros 4 bytes de um quadro de MP3.



Apesar desta seqüência de bits compor exatamente 4 bytes, igual ao tipo `int`, não faz sentido trabalhar com a mesma como se fosse um número inteiro. Cada campo precisa ser tratado de forma independente. Por exemplo, o campo *sincronia* é um número inteiro de 11 bits, cujo valor é 11111111111_2 . Sempre que encontrar esta seqüência no arquivo MP3, o decodificar saberá que encontrou um novo quadro.

Mas, como o decodificador poderá reconhecer este número, se não existe nenhum tipo em C que represente um inteiro com exatamente 11 bits? Ou como obter o número de amostras de áudio por segundo, que é um número de 2 bits localizado no meio de outros 30 bits? Como verificar se um quadro de MP3 possui ou não proteção de copyright?

Para lidar cm essas questões, a linguagem C oferece operadores específicos, com os quais é possível verificar determinados bits de uma seqüência (por exemplo, se o copyright está ativo ou não) ou modificar somente alguns bits da seqüência.

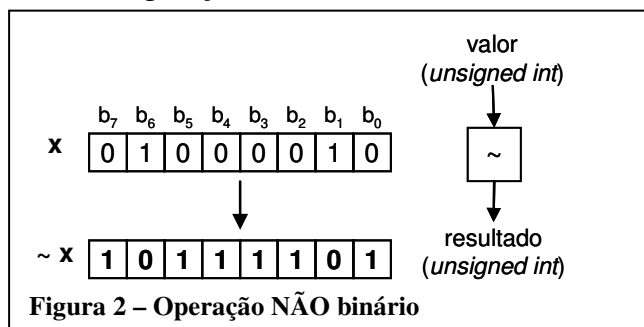
Variáveis com Seqüências de Bits

Em C, seqüências de bits são armazenadas em variáveis declaradas como números inteiros do tipo `unsigned int`. O fato de a variável ser `int` indica somente que desejamos uma variável com dígitos binários. Ela será interpretada como número somente quando realizamos operações aritméticas.

Por exemplo, no Visual Studio, o tipo `unsigned int` tem 32 bits, ou seja, a sequência poderá ter até 32 dígitos binários.

Operador NÃO binário

Inverte o valor de todos os dígitos binários. Os dígitos 0 tornam-se 1 e os valores 0 passam para 1. A Figura 2 ilustra esta operação.



Trata-se de um operador unário, ou seja, ele recebe somente um operando, que é o valor cujos dígitos binários terão seu valor invertido. O resultado é do tipo `unsigned int`.

O operador NÃO binário é representado pelo símbolo `~`.

Observação: Não confunda este operador com o operador `!` (não lógico) utilizado nas estruturas condicionais. O operador `~` (não binário) inverte o valor de todos os dígitos binários, enquanto que o operador `!` troca o valor *falso* (zero) para *verdadeiro* (não zero) e vice versa.

Exemplo:

Infelizmente, `scanf` e `printf` não possuem modificadores para ler ou imprimir um número usando a representação binária. Por este motivo, vamos assumir que existe o comando `escreveBinario`, que imprime o valor de uma variável em representação binária. De forma semelhante ao `scanf`, vamos assumir que existe um comando `leBinario` (que salva um valor binário digitado pelo usuário em uma variável). Agora, considere o código:

```
unsigned int v, not_binario_v, not_logico_v;

printf("Digite um numero binario: ");
leBinario(&v);

not_binario_v = ~ v;
not_logico_v = ! v;

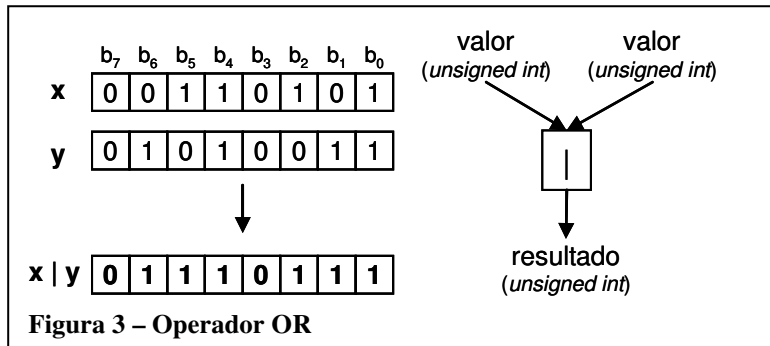
printf("Resultados:");
printf("\n    v = ");
escreveBinario(v);
printf("\n  ~ v = ");
escreveBinario(not_binario_v);
printf("\n  ! v = ");
escreveBinario(not_logico_v);
```

Consulte: RepresentacaoBinaria\Not01.vcproj

Vamos supor que o usuário tenha digitado o valor 0001 0111. A saída seria:

1	1	1
---	---	---

A Figura 3 ilustra uma operação OR.



Observação: Não confunda este operador com o operador `||` (OU lógico) utilizado nas estruturas condicionais.

Exemplo:

Suponha que existam os comandos `escreveBinario` e `leBinario`, descritos no exemplo anterior, e veja o código:

```
unsigned int u, v;
unsigned int or_binario, or_logico;

printf("Digite dois numeros binarios: ");
leBinario(&u);
leBinario(&v);

or_binario = u | v;
or_logico = u || v;

printf("Resultados:");
printf("\n      u = ");
escreveBinario(u);
printf("\n      v = ");
escreveBinario(v);
printf("\n u  | v = ");
escreveBinario(or_binario);
printf("\n u || v = ");
escreveBinario(or_logico);
```

Consulte: RepresentacaoBinaria\Or01.vcproj

Este programa lê duas variáveis, `u` e `v`, declaradas como `unsigned int`. O usuário deve escrever os valores em binário. Em seguida, o programa aplica o operador OU binário (`|`) e o operador OU lógico (`||`, utilizado em estruturas condicionais). Por fim, imprimem-se os resultados. A impressão ocorre com 32 dígitos, pois este é o tamanho de uma variável de tipo `unsigned int`. Uma saída seria:

Digite dois numeros: 0001100 0001010

,Consulte: RepresentacaoBinaria\And01.vcproj

Este programa lê duas variáveis `u` e `v` declaradas como `unsigned int`. O usuário deve escrever os valores em binário. Em seguida, o programa aplica o operador E binário (`&`) e o operador E lógico (`&&`, utilizado em estruturas condicionais). No fim, imprimem-se os resultados. A impressão ocorre com 32 dígitos, pois este é o tamanho da variável de tipo `unsigned int`. A saída de uma possível execução seria:

[illegible]

Note como o resultado `u & v` tem bits 1 apenas nas posições onde `u` e `v` têm bits 1. O operador `&&` tem um comportamento diferente. Ele interpreta os valores de `u` e `v` como verdadeiro (pois são não nulos) e, portanto resulta em *verdadeiro* (não nulo).

Operador OU EXCLUSIVO

Esse operador utiliza dois operandos, que são duas seqüências de bits de mesmo comprimento. Ele produz uma nova seqüência de mesmo comprimento aplicando a operação OU EXCLUSIVO sobre cada bit da primeira seqüência com o bit correspondente da outra seqüência. Se esses dois bits forem diferentes, então o bit da seqüência resultante será 1, caso contrário, será 0.

O operador OU EXCLUSIVO é representado pelo símbolo \wedge . A tabela a seguir resume sua operação:

Bit da	Bit da	Bit na seqüência
--------	--------	------------------


```

escreveBinario(v);
printf("\n v << d = ");
escreveBinario(v_esquerda);
printf("\n v >> d = ");
escreveBinario(v_direita);

```

Consulte: RepresentacaoBinaria\Deslocamento01.vcproj

Este programa lê uma variável `v` declarada como `unsigned int` e duas quantidades para o deslocamento. O usuário deve escrever o valor de `v` em binário. Em seguida, o programa aplica os dois operadores de deslocamento (`<<` e `>>`). Uma possível saída seria:

```

Digite um numero binario: 100100101
Digite um deslocamento para esquerda: 2
Digite um deslocamento para direita: 3
Resultados:
v = 0000000000000000000000000100100101
v << d = 0000000000000000000000000100100100
v >> d = 00000000000000000000000000000100100

```

Operações sobre bits

Vamos retornar ao exemplo dos bits codificados no cabeçalho de um quadro de áudio MP3 (consulte a Figura 1). Suponha que a sequência de bits que representa este cabeçalho seja armazenada em uma variável de tipo `unsigned int`.

Máscaras de bits

Uma constante, denominada **máscara**, indica a posição do(s) dígito(s) binário(s) que desejamos consultar ou alterar. Na representação binária, esta constante possui 1s nas posições dos dígitos desejados, e 0s nas demais posições.

Em C, as máscaras são sempre números inteiros de tipo `unsigned`. Elas devem ser escritas como números nas bases decimal, octal ou hexadecimal. Por motivos didáticos, escreveremos também as máscaras como números binários, com grupos de 8 dígitos separados por espaço, para facilitar a leitura. Você pode determinar o número decimal correspondente à máscara com o algoritmo de multiplicações sucessivas. Para encontrar o número hexadecimal correspondente é muito mais fácil, basta utilizar a tabela para converter quatro bits da máscara em um dígito hexadecimal. Por este motivo, os programadores C preferem escrever as máscaras como números em hexadecimal, pelo simples motivo de ser mais fácil fazer a conversão para essa base do que para a base decimal.

Por exemplo, para indicar somente o bit 0, utiliza-se a máscara 1 (0x00000001):

```
00000000 00000000 00000000 00000001
```

Para indicar somente os bits 1, 4 e 6, utiliza-se a máscara 82 (0x00000052):

```
00000000 00000000 00000000 01010010
```

Para indicar somente os bits de 4 até 15, utiliza-se a máscara 65520 (0x0000FFF0):

```
00000000 00000000 11111111 11110000
```

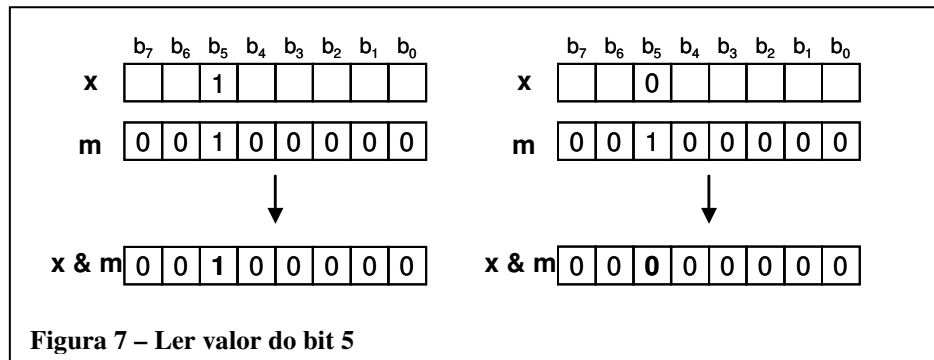
Para indicar todo os bits, exceto o bit 6, utiliza-se a máscara 4294967231 (0x FFFFFFFB):

11111111 11111111 11111111 10111111

Ler o valor de um determinado dígito binário

Isole o bit desejado utilizando uma máscara que indica este bit e aplique a operação E binário tendo como operandos o valor dado e a máscara. Se o resultado for não nulo, então este bit possui valor 1, caso contrário, ele possui valor 0.

Com a operação E binário, o resultado será 0 nas posições dos dígitos que na máscara tem valor 0. E o resultado será igual ao do valor dado nas posições que na máscara têm bits 1. Desta forma transformamos o problema de testar um bit em teste de uma condição zero/não zero.



A Figura mostra como determinar se o bit 5 é 0 ou 1. Por uma questão de simplicidade, ilustramos somente os primeiros 8 bits de um inteiro.

Código Fonte:

```
unsigned int v, resultado, mascara;

printf("Digite um numero binario: ");
leBinario(&v);

// Verificar se o bit 5 é 0 ou 1
// ...111 00100000 em binário
mascara = 0x00000020;

resultado = v & mascara;

printf("\n    v   ");
escreveBinario(v);
printf("\n  & m   ");
escreveBinario(mascara);
printf("\n    =   ");
escreveBinario(resultado);

if (resultado != 0) {
    printf("\nBit b5 é 1");
} else {
    printf("\nBit b5 é 0");
}
```

Consulte: RepresentacaoBinaria\LerBit01.vcproj

Executando o programa, obtemos, por exemplo:

Digite um numero binario: **00111000**
v 00000000000000000000000000111000
& m 00000000000000000000000000100000
= 00000000000000000000000000100000
Bit b5 é 1

ou, em outra execução:

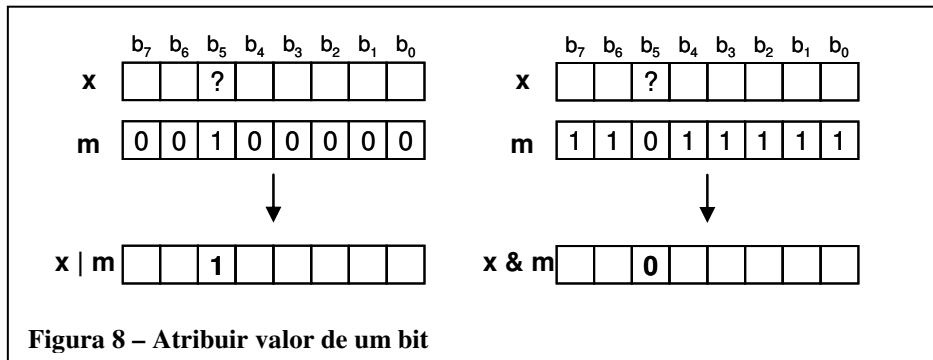
```
Digite um numero binario: 00000111
      v 0000000000000000000000000000000111
    & m 0000000000000000000000000000100000
    =   0000000000000000000000000000000000
Bit b5 é 0
```

Atribuir o valor para determinado dígito binário

Para atribuir o valor 1 a uma certa posição, indique a posição desejada na máscara como 1 e as demais posições na máscara como zero. Aplique a operação OU binário tendo como operandos a máscara e o valor dado. Com a operação OU binário, o resultado será igual ao original dado nas posições onde os dígitos na máscara são 0. E o resultado será 1 nas posições onde os dígitos na máscara são 1.

Para atribuir o valor 0, use uma máscara indicando todas as posições desejadas colocando zero nessas posições na máscara, e colocando 1 nas demais posições da máscara. Aplique a operação E binário entre a máscara e o valor dado. Com a operação E binário, o resultado será igual ao original nas posições que na máscara têm dígitos 1, e será 0 nas posições que na máscara têm dígitos 0.

Desta forma transformamos o problema de atribuir um bit em uma operação de OU binário (para atribuir 1) ou de E binário (para atribuir zero).



A Figura 8 mostra como atribuir os valores 1 e 0 no bit 5.

Código Fonte:

```
unsigned int v, resultado, mascara;

printf("Digite um numero binario: ");
leBinario(&v);
```

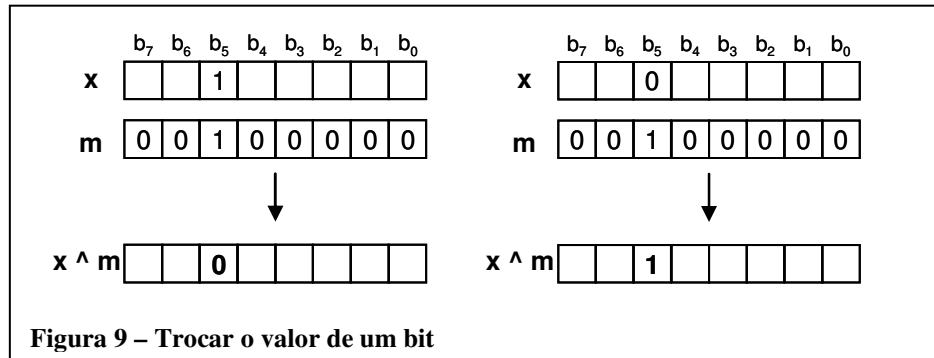
Consulte: RepresentacaoBinaria\AtribuirBit01.vcproj

= 0000000000000000000000000000000010001111

Trocar o valor de determinado dígito binário

Para trocar o valor de um bit, indique-o na máscara. Aplique a operação OU EXCLUSIVO. Com a operação OU EXCLUSIVO, o resultado será igual ao original nos dígitos com 0 na máscara. O resultado será o valor inverso nos dígitos com 1 na máscara.

Desta forma transformamos o problema de trocar um bit em uma operação de OU EXCLUSIVO.



A Figura 9 mostra como trocar o valor do bit 5.

Código Fonte:

```
unsigned int v, resultado, mascara;

printf("Digite um numero binario: ");
leBinario(&v);

// Trocar valor do bit 5
// Mascara que indica bit 5
// ...000 00100000 em binário
mascara = 0x00000020;
resultado = v ^ mascara;

printf("\n      v      ");
escreveBinario(v);
printf("\n      ^ m      ");
escreveBinario(mascara);
printf("\n      =      ");
escreveBinario(resultado);
```

Consulte: RepresentacaoBinaria\TrocarBit01.vcproj

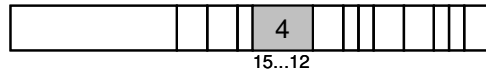
Executando o programa teríamos, por exemplo:

Digite um numero binario: **00011100**

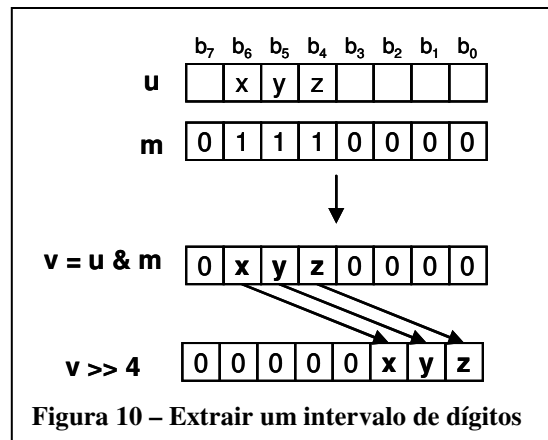
$$\begin{array}{lcl} v & \text{0000000000000000000000000011100} \\ \wedge m & \text{0000000000000000000000000100000} \\ = & \text{0000000000000000000000000111100} \end{array}$$

Ler um número oriundo de um intervalo de bits

No exemplo do cabeçalho de um quadro de áudio MP3, o número de bits por amostra de áudio está armazenado nos dígitos 12 a 15. A questão é como extrair apenas estes quatro dígitos e interpretá-los como um número.



Para extrair um número codificado em um intervalo de dígitos binários, é necessário primeiro criar uma máscara para identificar as posições desse intervalo. Em seguida, aplica-se um E binário entre a máscara e o valor dado para isolar estes dígitos. Então, desloca-se estes dígitos totalmente para a direita. O resultado será justamente a representação utilizada pelo computador para números inteiros sem sinal.



A Figura 10 mostra como extrair o intervalo de dígitos de 4 a 6 e obter um número inteiro.

Código Fonte:

```
unsigned int v, resultado, intervalo, mascara;

printf("Digite um numero binario: ");
leBinario(&v);
// Extrair a faixa de bit 4 a 6
// ...000 001110000 em binário
mascara = 0x00000070;

intervalo = v & mascara;
resultado = intervalo >> 4;

printf("\n      v      ");
escreveBinario(v);
printf("\n      m      ");
escreveBinario(mascara);
printf("\n v & m = ");
escreveBinario(intervalo);
printf("\n >> 4 = ");
escreveBinario(resultado);
```

Consulte: RepresentacaoBinaria\LerFaixaBits01.vcproj

Digite um numero binario: *01101001*

```
v      00000000000000000000000000001101001
m      00000000000000000000000000001110000
v & m = 00000000000000000000000000001100000
>> 4 = 00000000000000000000000000000000110
```

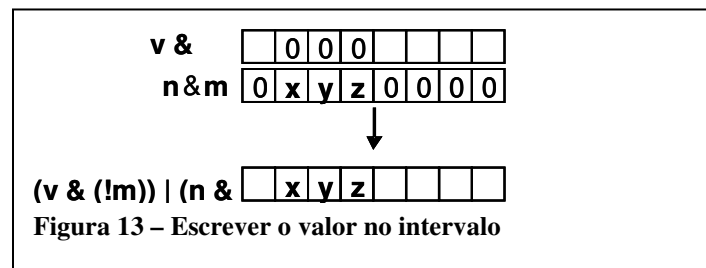
Esta é a operação inversa da leitura do valor compreendido em um intervalo de posições. Criamos uma máscara identificando as posições do intervalo desejado. Nos exemplos, desejamos sobrescrever os bits no intervalo das posições 4 a 6 com um novo valor de 3 dígitos.

Primeiro, zera-se os valores já existentes no intervalo com uma operação E binário e o complemento da máscara (obtido aplicando-se o operador NÃO binário sobre a máscara original). A Figura 11 exemplifica esta operação.



Figura 12 – Mover o número para o intervalo desejado

Finalmente, a operação OU binário com o número e a variável destino escreve os valores nas posições desejadas (Figura 13).



Código Fonte:

```
unsigned int v, v_limpo, n, n_deslocado, mascara, resultado;

printf("Digite a sequencia binaria antiga: ");
leBinario(&v);
printf("Digite o numero novo: ");
leBinario(&n);

// ...000 001110000 em binário
mascara = 0x00000070;

// Apagar valores já existentes no intervalo
v_limpo = v & (~mascara);
printf("\n v = ");
escreveBinario(v);
printf("\n v_limpo = ");
escreveBinario(v_limpo);
printf("\n");

// Mover os dígitos do número para a esquerda até o intervalo
n_deslocado = (n << 4) & mascara;
printf("\n n = ");
escreveBinario(n);
printf("\n n_deslocado = ");
escreveBinario(n_deslocado);
printf("\n");

// Escrever os valores
resultado = v_limpo | n_deslocado;
printf("\n resultado = ");
escreveBinario(resultado);
```

Consulte: RepresentacaoBinaria\AtribuirBits01.vcproj

Executando o programa teríamos, por exemplo:

Digite a sequencia binaria antiga: 1010101010
Digite o numero novo: 110

```

v          = 0000000000000000000000001010101010
v_limpo    = 0000000000000000000000001010001010

n          = 00000000000000000000000000000000110
n_deslocado = 000000000000000000000000000000001100000

resultado  = 0000000000000000000000001011101010

```