Compiladores - Turma A - 0225 - Noturno
Prof. Thiago Fernandes Neves

ENTREGA LAB04

Integrantes:

- Isabela Garcia
- Joao Victor Azevedo dos Santos
- Luiz Eduardo Campos Dias
- Yago Peres dos Santos

Link para o Github

1. **Exercícios Bitwise**
   a. Escreva um programa que leia 2 números inteiros a e b, mostrando resultado original e resultado em binário

Código:

```c
#include <stdio.h>

// Função para imprimir um número em formato binário
void imprime_binario(int numero)
{
  // Encontra o bit mais significativo
  int bits_significativos = 32;
  for (int i = 31; i > 0; i--)
  {
    if ((numero >> i) & 1)
    {
      break;
    }
    bits_significativos--;
  }T
  // Garante que pelo menos 4 bits sejam mostrados
  if (bits_significativos < 4)
    bits_significativos = 4;
  // Ajusta para o próximo múltiplo de 4
  else if (bits_significativos % 4 != 0)
    bits_significativos = ((bits_significativos / 4) + 1) * 4;

  // Imprime os bits em grupos de 4
  for (int i = bits_significativos - 1; i >= 0; i--)
  {
    printf("%d", (numero >> i) & 1);
    if (i % 4 == 0 && i != 0)
      printf(" ");
```

```c
    }
    printf("\n");
}
int main()
{
    int a, b;

    // Letra A: Comparação de dois números em binário
    printf("\n=== Letra A: Comparação de dois números em binário ===\n");
    printf("Digite o primeiro número (a): ");
    scanf("%d", &a);
    printf("Digite o segundo número (b): ");
    scanf("%d", &b);

    printf("\nNúmeros em decimal:\n");
    printf("a = %d\n", a);
    printf("b = %d\n", b);

    printf("\nNúmeros em binário:\n");
    printf("a = ");
    imprime_binario(a);
    printf("b = ");
    imprime_binario(b);

    printf("\nOperações bit a bit:\n");

    printf("a & b = %d\n", a & b);
    printf("Binário: ");
    imprime_binario(a & b);

    printf("a | b = %d\n", a | b);
    printf("Binário: ");
    imprime_binario(a | b);

    printf("a ^ b = %d\n", a ^ b);
    printf("Binário: ");
    imprime_binario(a ^ b);

    printf("~a = %d\n", ~a);
    printf("Binário: ");
    imprime_binario(~a);
}
```

Resultado:

```
=== Letra A: Comparação de dois números em binário ===
Digite o primeiro número (a): 5
Digite o segundo número (b): 4

Números em decimal:
a = 5
b = 4

Números em binário:
a = 0101
b = 0100

Operações bit a bit:
a & b = 4
Binário: 0100
a | b = 5
Binário: 0101
a ^ b = 1
Binário: 0001
~a = -6
Binário: 1111 1111 1111 1111 1111 1111 1111 1010
```

b. Leia um número inteiro positivo e um valor n, e exiba:
   i. *O número deslocado para a esquerda (x << n)*
   ii. *O número deslocado para a direita (x >> n)*

Código:

```c
// Letra B: Deslocamentos de bits
printf("\n=== Letra B: Deslocamentos de bits ===\n");
printf("Digite um número inteiro positivo (x): ");
scanf("%d", &x);
printf("Digite o valor do deslocamento (n): ");
scanf("%d", &n);

printf("\nNúmero original:\n");
printf("x = %d (decimal)\n", x);
printf("x = ");
imprime_binario(x);

printf("\nDeslocamento para a esquerda (x << %d):\n", n);
printf("Decimal: %d\n", x << n);
printf("Binário: ");
```

```c
  imprime_binario(x << n);

  printf("\nDeslocamento para a direita (x >> %d):\n", n);
  printf("Decimal: %d\n", x >> n);
  printf("Binário: ");
  imprime_binario(x >> n);
```

Resultado:

```
=== Letra B: Deslocamentos de bits ===
Digite um número inteiro positivo (x): 5
Digite o valor do deslocamento (n): 1

Número original:
x = 5 (decimal)
x = 0101

Deslocamento para a esquerda (x << 1):
Decimal: 10
Binário: 1010

Deslocamento para a direita (x >> 1):
Decimal: 2
Binário: 0010
```

c. Implemente três funções (Ativa, Desativa, Alterna), depois leia um número e uma posição de bit e teste as três funções

Código:

```c
// Funções de manipulação de bits (Letra C)
int ativa_bit(int numero, int posicao)
{
  return numero | (1 << posicao);
}

int desativa_bit(int numero, int posicao)
{
  return numero & ~(1 << posicao);
}

int alterna_bit(int numero, int posicao)
{
  return numero ^ (1 << posicao);
}
```

```c
  // Letra C: Manipulação de bits
  printf("\n=== Letra C: Manipulação de bits ===\n");
  printf("Digite um número inteiro: ");
  scanf("%d", &x);
  printf("Digite a posição do bit (0-31): ");
  scanf("%d", &n);

  printf("\nNúmero original: ");
  imprime_binario(x);

  int ativado = ativa_bit(x, n);
  printf("Após ativar bit %d: ", n);
  imprime_binario(ativado);

  int desativado = desativa_bit(x, n);
  printf("Após desativar bit %d: ", n);
  imprime_binario(desativado);

  int alternado = alterna_bit(x, n);
  printf("Após alternar bit %d: ", n);
  imprime_binario(alternado);
```

Resultado:

```
=== Letra C: Manipulação de bits ===
Digite um número inteiro: 5
Digite a posição do bit (0-31): 2

Número original: 0101
Após ativar bit 2: 0101
Após desativar bit 2: 0001
Após alternar bit 2: 0001
```

d. Escreva uma função que conte quantos bits 1 existem em um número inteiro positivo

Código:

```c
// Função para contar o número de bits 1 em um inteiro (Letra D)
int conta_bits_1(int numero)
{
  int contador = 0;
  for (int i = 0; i < 32; i++)
  {
    if (numero & (1 << i))
```

```
    {
      contador++;
    }
  }
  return contador;
}

  // Letra D: Contagem de bits 1
  printf("\n=== Letra D: Contagem de bits 1 ===\n");
  printf("Digite um número inteiro positivo: ");
  scanf("%d", &x);
  printf("Número em binário: ");
  imprime_binario(x);
  printf("Quantidade de bits 1: %d\n", conta_bits_1(x));
```

Resultado:

```
=== Letra D: Contagem de bits 1 ===
Digite um número inteiro positivo: 7
Número em binário: 0111
Quantidade de bits 1: 3
```

e. Crie uma função que armazena dois números de 4 bits (0 a 15) em um único unsigned char, depois recupere os dois números originais.

Código:

```
// Funções Letra E: Compactação e descompactação de números

// Função para compactar dois números de 4 bits em um único byte
unsigned char compactar(unsigned char a, unsigned char b)
{
  if (a > 15 || b > 15)
    return 0; // Verifica se os números são válidos (máximo 4 bits)
  return (a << 4) | b;
}

// Função para descompactar o byte em dois números de 4 bits
void descompactar(unsigned char compactado, unsigned char *a, unsigned char
*b)
{
  *a = (compactado >> 4) & 0x0F; // Recupera os 4 bits mais significativos
  *b = compactado & 0x0F;        // Recupera os 4 bits menos significativos
}
```

```c
  // Letra E: Compactação de números
  printf("\n=== Letra E: Compactação de números de 4 bits ===\n");
  unsigned char num1, num2, compactado;

  printf("Digite o primeiro número (0-15): ");
  scanf("%hhu", &num1);
  printf("Digite o segundo número (0-15): ");
  scanf("%hhu", &num2);

  compactado = compactar(num1, num2);
  printf("\nNúmeros compactados em binário: ");
  imprime_binario(compactado);

  unsigned char valor1, valor2;
  descompactar(compactado, &valor1, &valor2);
  printf("Números descompactados: %d e %d\n", valor1, valor2);
```

Resultado:

```
=== Letra E: Compactação de números de 4 bits ===
Digite o primeiro número (0-15): 7
Digite o segundo número (0-15): 3

Números compactados em binário: 0111 0011
Números descompactados: 7 e 3
```

2. **Atividade 1 - Compilar com sucesso o Makefile e teste com entrada 1234 + 6789**

Código (utilizado código do LAB03 disponibilizado, feitas alterações solicitadas):

```
Makefile:
OBJECTS=  ./build/compiler.o \
      ./build/cprocess.o \
      ./build/lexer.o \
      ./build/token.o \
      ./build/lex_process.o \
      ./build/parser.o \
      ./build/node.o \
      ./build/helpers/buffer.o \
      ./build/helpers/vector.o
INCLUDES= -I./

all: ${OBJECTS}
  gcc main.c ${INCLUDES} ${OBJECTS} -g -o ./main
```

```
./build/compiler.o: ./compiler.c
  gcc ./compiler.c ${INCLUDES} -o ./build/compiler.o -g -c

./build/cprocess.o: ./cprocess.c
  gcc ./cprocess.c ${INCLUDES} -o ./build/cprocess.o -g -c

./build/lexer.o: ./lexer.c
  gcc ./lexer.c ${INCLUDES} -o ./build/lexer.o -g -c

./build/token.o: ./token.c
  gcc ./token.c ${INCLUDES} -o ./build/token.o -g -c

./build/lex_process.o: ./lex_process.c
  gcc ./lex_process.c ${INCLUDES} -o ./build/lex_process.o -g -c

./build/parser.o: ./parser.c
  gcc ./parser.c ${INCLUDES} -o ./build/parser.o -g -c

./build/node.o: ./node.c
  gcc ./node.c ${INCLUDES} -o ./build/node.o -g -c

./build/helpers/buffer.o: ./helpers/buffer.c
  gcc ./helpers/buffer.c ${INCLUDES} -o ./build/helpers/buffer.o -g -c

./build/helpers/vector.o: ./helpers/vector.c
  gcc ./helpers/vector.c ${INCLUDES} -o ./build/helpers/vector.o -g -c

clean:
  rm ./main
  rm -rf ${OBJECTS}
```

Resultado:

```
❯ ./main test.c output.c
Compiladores - TURMA A/B - GRUPO 7

#Input file: test.c
#Output file: output.c

TOKEN   NU: 1234         PARENTESES: (null)
TOKEN   OP: +
TOKEN   NU: 6789         PARENTESES: (null)
Todos os arquivos foram compilados com sucesso!
```

3. **Atividade 2 - Elaborar 3 testes para verificar a precedência de operadores**

Código (feitas alterações solicitadas):

```
Novo Makefile:
OBJECTS=  ./build/compiler.o \
      ./build/cprocess.o \
      ./build/lexer.o \
      ./build/token.o \
      ./build/lex_process.o \
      ./build/parser.o \
      ./build/node.o \
      ./build/helpers/buffer.o \
      ./build/helpers/vector.o \
      ./build/expressionable.o

INCLUDES= -I./

all: ${OBJECTS}
  gcc main.c ${INCLUDES} ${OBJECTS} -g -o ./main

./build/compiler.o: ./compiler.c
  gcc ./compiler.c ${INCLUDES} -o ./build/compiler.o -g -c

./build/cprocess.o: ./cprocess.c
  gcc ./cprocess.c ${INCLUDES} -o ./build/cprocess.o -g -c

./build/lexer.o: ./lexer.c
  gcc ./lexer.c ${INCLUDES} -o ./build/lexer.o -g -c

./build/token.o: ./token.c
  gcc ./token.c ${INCLUDES} -o ./build/token.o -g -c

./build/lex_process.o: ./lex_process.c
  gcc ./lex_process.c ${INCLUDES} -o ./build/lex_process.o -g -c

./build/parser.o: ./parser.c
  gcc ./parser.c ${INCLUDES} -o ./build/parser.o -g -c

./build/node.o: ./node.c
  gcc ./node.c ${INCLUDES} -o ./build/node.o -g -c

./build/helpers/buffer.o: ./helpers/buffer.c
  gcc ./helpers/buffer.c ${INCLUDES} -o ./build/helpers/buffer.o -g -c
```

```
./build/helpers/vector.o: ./helpers/vector.c
  gcc ./helpers/vector.c ${INCLUDES} -o ./build/helpers/vector.o -g -c

./build/expressionable.o: ./expressionable.c
  gcc ./expressionable.c ${INCLUDES} -o ./build/expressionable.o -g -c

clean:
  rm ./main
  rm -rf ${OBJECTS}
```

Resultado:

Teste 1:

```
2 * 3 + 2 / 2 - 4
```

output:

```
❯ ./main test_precedence1.c output.c
Compiladores - TURMA A/B - GRUPO 7

#Input file: test_precedence1.c
#Output file: output.c

TOKEN   NU: 2    PARENTESES: (null)
TOKEN   OP: *
TOKEN   NU: 3    PARENTESES: (null)
TOKEN   OP: +
TOKEN   NU: 2    PARENTESES: (null)
TOKEN   OP: /
TOKEN   NU: 2    PARENTESES: (null)
TOKEN   OP: -
TOKEN   NU: 4    PARENTESES: (null)


Arvore de nodes:
└── EXPRESSION (op: +)
|   ├── EXPRESSION (op: *)
|   |   ├── NUMBER (2)
|   |   └── NUMBER (3)
|   └── EXPRESSION (op: -)
|   |   ├── EXPRESSION (op: /)
|   |   |   ├── NUMBER (2)
|   |   |   └── NUMBER (2)
```

```
│   │   └── NUMBER (4)


Todos os arquivos foram compilados com sucesso!
```

Teste 2:

```
1 + 2 * 3 - 4 / 2
```

output:

```
❯ ./main test_precedence2.c output.c
Compiladores - TURMA A/B - GRUPO 7

#Input file: test_precedence2.c
#Output file: output.c

TOKEN   NU: 1    PARENTESES: (null)
TOKEN   OP: +
TOKEN   NU: 2    PARENTESES: (null)
TOKEN   OP: *
TOKEN   NU: 3    PARENTESES: (null)
TOKEN   OP: -
TOKEN   NU: 4    PARENTESES: (null)
TOKEN   OP: /
TOKEN   NU: 2    PARENTESES: (null)


Arvore de nodes:
└── EXPRESSION (op: +)
│   ├── NUMBER (1)
│   └── EXPRESSION (op: -)
│   │   ├── EXPRESSION (op: *)
│   │   │   ├── NUMBER (2)
│   │   │   └── NUMBER (3)
│   │   └── EXPRESSION (op: /)
│   │   │   ├── NUMBER (4)
│   │   │   └── NUMBER (2)


Todos os arquivos foram compilados com sucesso!
```

Teste 3:

```
0 + 2 / 2 * 5 - 4
```

output:

```
❯ ./main test_precedence3.c output.c
Compiladores - TURMA A/B - GRUPO 7

#Input file: test_precedence3.c
#Output file: output.c

TOKEN   NU: 0    PARENTESES: (null)
TOKEN   OP: +
TOKEN   NU: 2    PARENTESES: (null)
TOKEN   OP: /
TOKEN   NU: 2    PARENTESES: (null)
TOKEN   OP: *
TOKEN   NU: 5    PARENTESES: (null)
TOKEN   OP: -
TOKEN   NU: 4    PARENTESES: (null)


Arvore de nodes:
└── EXPRESSION (op: +)
|   ├── NUMBER (0)
|   └── EXPRESSION (op: -)
|   |   ├── EXPRESSION (op: /)
|   |   |   ├── NUMBER (2)
|   |   |   └── EXPRESSION (op: *)
|   |   |   |   ├── NUMBER (2)
|   |   |   |   └── NUMBER (5)
|   |   └── NUMBER (4)


Todos os arquivos foram compilados com sucesso!
```