



**UNIVERSIDADE  
ESTADUAL DE MARINGÁ  
DEPARTAMENTO DE INFORMÁTICA**

---

**Ingrid Louise Pereira Lohmann**

**Relatório Trabalho 1  
Calculadora Binária**

**Professor: Felipe Fernandes da Silva**

---

Maringá – PR  
2022

# SUMÁRIO

|  |          |
|--|----------|
| <b>1. Introdução</b>   | <b>4</b> |
| <b>2. Listas de funções, métodos e propriedades padrões da linguagem JavaScript (JS)</b> | <b>5</b> |
| 2.1 String.prototype.substring()   | 5        |
| 2.2 String.prototype.split()   | 5        |
| 2.3 Array.prototype.join()   | 5        |
| 2.4 Array.prototype.reverse()  | 5        |
| 2.5 Array.prototype.reduce()   | 6        |
| 2.6 Math.pow()   | 6        |
| 2.7 String.length  | 6        |
| 2.8 parseInt()   | 6        |
| 2.9 Number.prototype.toString()  | 6        |
| 2.10 String.prototype.slice()  | 7        |
| 2.11 String.prototype.padStart()   | 7        |
| 2.12 String.prototype.lastIndexOf()  | 7        |
| 2.13 Array.prototype.includes()  | 7        |
| 2.14 Array.prototype.indexOf()   | 7        |
| 2.15 Array.prototype.map()   | 8        |
| 2.16 String.prototype.includes()   | 8        |
| 2.17 Array.prototype.some()  | 8        |
| 2.18 Array.prototype.forEach()   | 8        |
| <b>3. Listagem das funções e variáveis globais utilizadas</b>                            | <b>9</b> |
| 3.1 result   | 9        |
| 3.2 operator   | 9        |
| 3.3 splitNumbers   | 9        |
| 3.4 overflow   | 9        |
| 3.5 isTwosComplement   | 9        |
| 3.6 clean()  | 9        |
| 3.7 concatNumber()   | 10       |
| 3.8 backspace()  | 10       |

|  |           |
|--|-----------|
| 3.9 splitInputNumbers()  | 11        |
| 3.10 addZeros()  | 11        |
| 3.11 converteBinaryToDecimal()                                     | 12        |
| 3.12 binarySum()   | 12        |
| 3.13 flipBit()   | 14        |
| 3.14 twosComplement()  | 15        |
| 3.15 isBiggerThan255()   | 15        |
| 3.16 binaryDivision()  | 16        |
| 3.17 binaryMultiplication()  | 16        |
| 3.18 mathOperations()  | 17        |
| 3.19 errorAlert() e infoAlert()                                    | 20        |
| 3.20 operationResult()   | 20        |
| 3.21 handleClick()   | 22        |
| 3.22 keys.forEach()  | 22        |
| <b>4. Dificuldades e facilidades durante a execução do projeto</b> | <b>24</b> |
| 4.1 Facilidades  | 24        |
| 4.2 Dificuldades   | 24        |
| <b>5. Como executar o projeto</b>                                  | <b>25</b> |
| <b>Referências bibliográficas</b>                                  | <b>26</b> |

# 1. Introdução

A calculadora moderna, tal como a conhecemos hoje, surgiu nos anos 70 nos Estados Unidos, porém o conceito de máquina para calcular advém de muito tempo atrás.

Por volta do século XI antes de Cristo, na região da Mesopotâmia, surgiu o avô das calculadoras atuais, o ábaco, podendo ser considerado uma extensão dos cálculos com os dedos, no qual era possível realizar as quatro operações matemáticas básicas, porém com maior ênfase na adição e na subtração.

Em 1642, surgiu a *La pascaline*, a primeira calculadora mecânica que realizava apenas a subtração e adição de até três parcelas de valores até 999.999, criada pelo famoso matemático Blaise Pascal, que na época da invenção tinha entre 19 a 21 anos.

Alguns anos depois, em 1971, Gottfried Leibniz melhorou a máquina de Pascal, fazendo com que fosse possível, além de somar e subtrair, realizar a multiplicação e a divisão.

Desde então outras calculadoras foram surgindo e, à medida que o tempo avançava, mais operações matemáticas foram sendo adicionadas, além da diminuição do tamanho da máquina, chegando ao que temos hoje em dia.

Porém, mesmo com o avanço tecnológico, a lógica por trás dos cálculos realizados por uma calculadora não é simples. Deste modo, este trabalho tem como objetivo, através da aritmética binária, simular a lógica por trás de uma calculadora moderna, seja ela de bolso, a calculadora de um celular ou um software em um computador.

Para o trabalho foram utilizadas a linguagem de programação *JavaScript (JS)* para a lógica das funções, a linguagem de marcação *HTML* e a linguagem de folha de estilo *CSS* (juntamente com a linguagem *Sass*) para construir o front-end da aplicação.

Este trabalho estará disponível online, assim como o seu código fonte, através do link <https://didzao.github.io/calculadora-binaria/>.

## 2. Listas de funções, métodos e propriedades padrões da linguagem JavaScript (JS)

Abaixo estão listadas as funções, métodos e propriedades padrões da linguagem JS utilizadas neste trabalho. Os parâmetros passados dentro de colchetes, ou seja, [parâmetro], são considerados opcionais para o uso das funções.

Todas as descrições e sintaxes foram retiradas da documentação oficial da linguagem. Além disso, a ordem listagem está de acordo com a ordem na qual essas funções, métodos ou propriedades foram utilizadas no arquivo *script.js*.

### 2.1 String.prototype.substring()

O método *substring()* retorna a parte da string entre os índices inicial e final, ou até o final da string.

#### Sintaxe

```
str.substring(indexStart[, indexEnd])
```

### 2.2 String.prototype.split()

O método *split()* divide a string em uma lista ordenada de substrings, colocando-as em um array e retornando esse array. A divisão é feita buscando um padrão, que deve ser fornecido como parâmetro.

#### Sintaxe

```
str.split([separator[, limit]])
```

### 2.3 Array.prototype.join()

O método *join()* junta os elementos de um array em uma string e retorna esta string.

#### Sintaxe

```
arr.join([separador = ', '])
```

### 2.4 Array.prototype.reverse()

O método *reverse()* inverte os itens de um array. O primeiro elemento do array se torna o último e o último torna-se o primeiro.

## Sintaxe

```
arr.reverse()
```

## 2.5 Array.prototype.reduce()

O método *reduce()* executa uma função *reducer* (fornecida por você) para cada elemento do array, resultando num único valor de retorno.

## Sintaxe

```
array.reduce(callback( acumulador, valorAtual[, index[, array]] ),  
valorInicial))
```

## 2.6 Math.pow()

A função *Math.pow()* retorna a base elevada ao expoente power, ou seja,  $base^{expoente}$ .

## Sintaxe

```
Math.pow(base, expoente)
```

## 2.7 String.length

A propriedade *length* retorna o comprimento do objeto String.

## Sintaxe

```
str.length
```

## 2.8 parseInt()

A função *parseInt()* analisa o argumento string e retorna um inteiro na base especificada. A base é um inteiro entre 2 e 36.

## Sintaxe

```
parseInt(string, base)
```

## 2.9 Number.prototype.toString()

O método *toString()* retorna uma string representando o objeto Number especificado. O parâmetro radix é um inteiro entre 2 e 36 especificando a base que será utilizada para representar os valores.

## Sintaxe

```
numObj.toString([radix])
```

## 2.10 String.prototype.slice()

O método *slice()* extrai uma parte de uma string e a retorna como uma nova string, sem modificar a string original.

### Sintaxe

```
str.slice([inicio[, fim]])
```

## 2.11 String.prototype.padStart()

O método *padStart()* preenche a string original com um determinado caractere, ou conjunto de caracteres (parâmetro *padString*), até que a string resultante atinja o comprimento fornecido. A string original não é modificada.

### Sintaxe

```
str.padStart(targetLength [, padString])
```

## 2.12 String.prototype.lastIndexOf()

O método *lastIndexOf()* retorna o índice da última ocorrência do valor especificado encontrado na String. Quando *fromIndex* é especificado, a pesquisa é realizada de trás para frente. Retorna -1 se o valor não for encontrado.

### Sintaxe

```
str.lastIndexOf(searchElement[, fromIndex])
```

## 2.13 Array.prototype.includes()

O método *includes()* determina se um array contém um determinado elemento, retornando true ou false apropriadamente.

### Sintaxe

```
array.includes(searchElement[, fromIndex])
```

## 2.14 Array.prototype.indexOf()

O método *indexOf()* retorna o primeiro índice em que o elemento pode ser encontrado no array, retorna -1 caso o mesmo não esteja presente.

### Sintaxe

```
array.indexOf(searchElement, [pontoInicial = 0])
```

## 2.15 Array.prototype.map()

O método *map()* invoca a função *callback* passada por argumento para cada elemento do Array e devolve um novo Array como resultado.

### Sintaxe

```
arr.map(callback[, thisArg])
```

## 2.16 String.prototype.includes()

O método *includes()* determina se um conjunto de caracteres pode ser encontrado dentro de outra string, retornando true ou false.

### Sintaxe

```
str.includes(searchString[, position])
```

## 2.17 Array.prototype.some()

O método *some()* testa se ao menos um dos elementos no array passa no teste implementado pela função atribuída e retorna um valor true ou false.

### Sintaxe

```
arr.some(callback[, thisArg])
```

## 2.18 Array.prototype.forEach()

O método *forEach()* executa uma dada função em cada elemento de um array.

### Sintaxe

```
arr.forEach(callback(currentValue [, index [, array]]),[, thisArg]);
```



### 3. Listagem das funções e variáveis globais utilizadas

Abaixo estão listadas e explicadas as funções e variáveis globais que foram implementadas neste trabalho. Vale ressaltar que as funções serão listadas na ordem na qual foram implementadas no arquivo *script.js* e, que as funções que têm a finalidade implementar valores no arquivo *index.html* serão apenas citadas, não serão explicadas, uma vez que este não é o foco do trabalho.

#### 3.1 result

Variável que armazena o resultado da operação efetuada.

#### 3.2 operator

Variável que armazena em um array os caracteres das operações (“+”, “-”, “\*” e “/”) selecionados pelo usuário.

#### 3.3 splitNumbers

Variável que armazena em um array os números (antes e depois do sinal da operação) selecionados pelo usuário.

#### 3.4 overflow

Variável booleana que controla se houve ou não *overflow*.

#### 3.5 isTwosComplement

Variável booleana que controla se resultado é ou não complemento de dois.

```
let result;  
let operator;  
let splitNumbers;  
let overflow;  
let isTwosComplement;
```

#### 3.6 clean()

A função *clean()* tem como objetivo limpar os valores que são mostrados ao usuário.

##### **Sintaxe**

*clean()*

## Parâmetros

Não se aplica.

```
const clean = () => {  
  resetInputLabel();  
  overflow = false;  
  result = undefined;  
  inputNumber.innerHTML = "";  
  displayBinary.innerHTML = "";  
}
```

### 3.7 concatNumber()

A função `concatNumber()` é utilizada para concatenar os valores inseridos pelo usuário através do teclado virtual.

#### Sintaxe

`concatNumber(value)`

#### Parâmetros

*value*: valor retornado pela tecla selecionada pelo usuário.

```
const concatNumber = (value) => {  
  overflow = false;  
  resetInputLabel();  
  if (inputNumber.textContent == "Erro!") {  
    clean();  
    inputNumber.innerHTML += value;  
  } else if (result != undefined) {  
    clean();  
    inputNumber.innerHTML += value;  
  } else {  
    inputNumber.innerHTML += value;  
  }  
}
```

### 3.8 backspace()

A função *backspace()* apaga um por um (do sentido da direita para a esquerda) os valores inseridos pelo usuário. Caso o valor mostrado na tela seja igual a **Erro!** a função *clean()* é chamada.

## Sintaxe

*backspace()*

## Parâmetros

Não se aplica.

```
const backspace = () => {
  if (inputNumber.textContent) {
    if (inputNumber.textContent == "Erro!") {
      clean();
    } else {
      let displayed =
document.getElementById("displayNumber").innerHTML;
      inputNumber.innerHTML = displayed.substring(0,
displayed.length - 1);
    }
  }
}
```

### 3.9 splitInputNumbers()

A função *splitInputNumbers()* tem como objetivo separar os valores numéricos inseridos pelo usuário do sinal da operação selecionada. Para isso são usadas expressões regulares (*Regex*) para atribuir os valores dos sinais encontrados e dos números inseridos nas variáveis *operator* e *splitNumbers*, respectivamente.

## Sintaxe

*splitInputNumbers(displayNumber)*

## Parâmetros

*displayNumber*: string inserida pelo usuário através do teclado virtual.

```
const splitInputNumbers = (displayNumber) => {
  operator =
displayNumber.innerHTML.split(/[0-9]/).join("").split("");

  splitNumbers = displayNumber.innerHTML.split(/[+\-\\*\]/);
}
```

### 3.10 addZeros()

A função *addZeros()* adiciona os zeros necessários para que o número inserido pelo usuário (após ser convertido para binário) possua um tamanho total de 8.

## Sintaxe

*addZeros(binaryStr)*

## Parâmetros

*binaryStr*: string do número convertido em binário.

```
const addZeros = (binaryStr) => {  
  return "00000000".substring(binaryStr.length) + binaryStr;  
}
```

### 3.11 *convertBinaryToDecimal()*

A função *convertBinaryToDecimal()*, como o nome sugere, converte o valor binário em decimal. Para isso é utilizado o método *split()*, para transformar o binário em um array. Em seguida é usado o método *reverse()*, para inverter a ordem desse array gerado, então utilizado o método *reduce*, passando como parâmetro uma função passando que por sua vez possui os parâmetros, *x*, *y* e *i*, sendo *x* o acumulador, *y* o valor atual (neste caso o valor que vem do array gerado acima) e *i* o index. Quando o valor de *y* for 1 o valor de *x* será somado com o resultado de  $2^i$ , caso contrário o próprio *x* será retornado e, a cada execução dessa função o valor *x* é acumulado, resultando no valor final que será retornado na função *convertBinaryToDecimal()*.

## Sintaxe

*convertBinaryToDecimal(binary)*

## Parâmetros

*binary*: string do binário que se deseja converter para decimal.

```
const convertBinaryToDecimal = (binary) => {  
  return binary.split('').reverse().reduce((x, y, i) => {  
    return (y === '1') ? x + Math.pow(2, i) : x;  
  }, 0);  
}
```

### 3.12 *binarySum()*

A função *binarySum()* é a mais importante deste trabalho, pois é através dela que as outras operações serão realizadas.

Essa função recebe dois parâmetros, os quais são transformados em binário (através do método *toString()*) e são completados com os zeros (através da função *addZeros()*). Após esse processo as variáveis serão chamadas de *firstBinary* e *secondBinary*, respectivamente.

Duas variáveis são inicializadas no começo dessa função, sendo elas a *sumResult* e a *carry*, uma para armazenar o valor do resultado final da operação e a outra que faz o trabalho do “vai um” durante a soma, respectivamente.

Então um laço *while* é chamado e será executado sempre quando um dos valores *firstBinary*, *secondBinary* ou *carry* for diferente de vazio.

Dentro desse laço de repetição a variável *sum* é criada, recebendo os último valor das string *firstBinary* e *secondBinary* (isso é possível pois é passado um valor negativo para o método *slice()*) e mais o valor da variável *carry*.

*Obs:* Vale ressaltar que o sinal de + logo a frente das variáveis *firstBinary* e *secondBinary*, nada mais é do que um recurso para transformar esses valores em um tipo Number.

Ainda dentro do *while* é verificado se o valor da variável *sum* é maior que 1, caso isso seja verdade a variável *sumResult* recebe o valor do resto da divisão da variável *sum* por 2 e soma com si própria e a variável *carry* recebe o valor 1. Caso contrário o *sumResult* receberá o valor da soma entre ela e a variável *sum* e o *carry* receberá o valor 0.

Após essa verificação as variáveis *firstBinary* e *secondBinary* receberão o último valor delas mesmas, respectivamente.

Em caso do valor da variável *SumResult* for maior ou igual a 256, a variável *overflow* irá receber o valor de true.

Por fim, quando o laço de repetição for finalizado a função irá retornar o valor da variável *SumResult* subtraindo o caractere mais à esquerda.

## Sintaxe

*binarySum(firstValue, secondValue)*

## Parâmetros

*firstValue*: primeiro valor (numérico) encontrado no array *splitNumbers*.

*secondValue*: segundo valor (numérico) encontrado no array *splitNumbers*.

```
const binarySum = (firstValue, secondValue) => {
  let sumResult = "";
  let carry = 0;

  let firstBinary = firstValue;
  let secondBinary = secondValue;

  if (typeof firstValue === "number") {
    firstBinary = addZeros(parseInt(firstValue).toString(2));
  }
}
```

```

    if (typeof secondValue === "number") {
        secondBinary = addZeros(parseInt(secondValue).toString(2));
    }

    while (firstBinary || secondBinary || carry) {

        let sum = +firstBinary.slice(-1) + +secondBinary.slice(-1) +
carry;

        if (sum > 1) {
            sumResult = sum % 2 + sumResult;
            carry = 1;
        }
        else {
            sumResult = sum + sumResult;
            carry = 0;
        }

        firstBinary = firstBinary.slice(0, -1);
        secondBinary = secondBinary.slice(0, -1);
    }

    if (parseInt(sumResult, 2) >= 256) {
        overflow = true;
    }

    return sumResult.substring(sumResult.length - 8);
}

```

### 3.13 flipBit()

A função *flipBit()* inverte os valores de 1 para 0 e de 0 para 1.

#### Sintaxe

*flipBit(bit)*

#### Parâmetros

*bit*: string com o valor do “bit” em questão.

```

const flipBit = (bit) => bit === '1' ? '0' : '1';

```

### 3.14 twosComplement()

A função *twosComplements()*, realiza a conversão do binário em seu complemento de dois. Para tal é usado a técnica de encontrar o primeiro número 1 (da direita para a esquerda) e inverter (utilizando a função *flipBit()*) os números a partir desse ponto.

#### Sintaxe

*twosComplements(value)*

#### Parâmetros

*value*: string do binário que se quer obter o complemento de dois.

```
const twosComplement = (value) => {

  let twosComplementValue = '';
  let binaryValue = addZeros(value.toString(2));
  const lastOne = binaryValue.lastIndexOf('1');

  if (lastOne === -1) {
    return '1' + binaryValue;
  } else {
    for (let i = 0; i < lastOne; i++) {
      twosComplementValue += flipBit(binaryValue[i]);
    }
  }
  twosComplementValue += binaryValue.substring(lastOne);
  return twosComplementValue;
}
```

### 3.15 isBiggerThan255()

A função *isBiggerThan255()*, realiza a verificação se algum dos números inseridos pelo usuário ou o resultado da operação é maior que 255.

#### Sintaxe

*isBiggerThan255(value)*

#### Parâmetros

*value*: valor a ser verificado.

```
const isBiggerThan255 = (value) => value > 255;
```

### 3.16 binaryDivision()

A função *binaryDivision()* é responsável pela divisão binária de dois números. Partindo do pressuposto que a divisão nada mais é do que uma sucessão de subtrações até que o numerador seja maior ou igual ao denominador (em casos de divisões exatas), tem-se que esse raciocínio foi aplicado nessa função.

Para tal é usado uma verificação, se os dois parâmetros fornecidos são iguais, em caso verdadeiro, o valor retornado será retornado o valor igual a 1, em caso contrário será chamado um laço de repetição *while* que sempre será executado quando o primeiro valor for maior que zero e quando o primeiro valor for maior ou igual ao segundo valor. Dentro desse laço o parâmetro *firstValue* recebe o resultado obtido com a função *binarySum*, na qual é passado o segundo parâmetro como complemento de dois, para que a subtração possa ser realizada. Além disso, cada vez que o laço for executado a variável *i* é incrementada e no final, quando a condição de execução do *while* for falsa, o valor *i* recebe um tratamento para ser convertido para binário e por fim o valor é retornado.

#### Sintaxe

*binaryDivision(firstValue, secondValue)*

#### Parâmetros

*firstValue*: primeiro valor (numérico) encontrado no array *splitNumbers*.

*secondValue*: segundo valor (numérico) encontrado no array *splitNumbers*.

```
const binaryDivision = (firstValue, secondValue) => {
  let i = 0;
  if (firstValue === secondValue) {
    i = 1;
  }
  while (firstValue >= secondValue) {
    firstValue = convertBinaryToDecimal(binarySum(firstValue,
twosComplement(secondValue)));
    i++;
  }
  const quotient = i.toString(2)
  return addZeros(quotient);
}
```

### 3.17 binaryMultiplication()

A função *binaryMultiplication()* é responsável pela multiplicação binária de dois números. Seguindo o mesmo raciocínio apresentado na explicação da função *binaryDivision()* tem-se a multiplicação nada mais é do que uma sucessão de soma até que determinada condição seja atingida.



Deste modo, primeiramente é feita uma verificação se algum dos parâmetros recebidos é igual a zero, caso isso seja verdade o valor retornado será zero, em caso negativo um laço de repetição *while* cuja condição de execução se baseia na verificação de que o valor da variável *i* tem que ser menor que o valor do parâmetro *secondValue*. No laço de repetição, toda vez que for executado a variável *multiplicationResult* receberá o valor retornado por *binarySum* e a variável *i* será incrementada.

Por fim a variável *multiplicationResult* é retornada.

## Sintaxe

*binaryMultiplication(firstValue, secondValue)*

## Parâmetros

*firstValue*: primeiro valor (numérico) encontrado no array *splitNumbers*.

*secondValue*: segundo valor (numérico) encontrado no array *splitNumbers*.

```
const binaryMultiplication = (firstValue, secondValue) => {
  let i = 0;
  let multiplicationResult = 0;

  if (firstValue === 0 || secondValue == 0) {
    multiplicationResult = addZeros("0");
  } else {
    while (i < secondValue) {
      multiplicationResult = binarySum(multiplicationResult,
firstValue);
      i++;
    }
  }

  return multiplicationResult;
}
```

## 3.18 mathOperations()

A função *mathOperations()* é responsável pela verificação de qual operação matemática será realizada. Para tal foram criadas quatro grandes condicionais que abrangem as quatro operações matemáticas e dentro de cada uma delas são verificados outras condições para checar se a operação será realizada com ambos os números sendo positivos, ambos negativos ou se apenas um deles será negativo. Em caso de resultados que são negativos, a constante *isTwosComplement* é alterada para *true*.

Por fim a variável *result* é retornada.

## Sintaxe

*mathOperations(arrayOfNumbers)*

## Parâmetros

*arrayOfNumbers*: array gerado na função *splitInputNumbers* e convertido para números.

```
const mathOperations = (arrayOfNumbers) => {

  if (
    operator.includes("+")
    && operator.indexOf("-") === -1
    && operator.indexOf("*") === -1
    && operator.indexOf("/") === -1
  ) {
    result = binarySum(arrayOfNumbers[0], arrayOfNumbers[1]);
  } else if (
    operator.includes("-")
    && operator.indexOf("*") === -1
    && operator.indexOf("/") === -1
  ) {
    if (operator.length === 1 && operator.indexOf("-") === 0) {
      if (arrayOfNumbers[0] < arrayOfNumbers[1]) {
        isTwosComplement = true;
      }
      result = binarySum(arrayOfNumbers[0],
twosComplement(arrayOfNumbers[1]));
    } else if (operator.length >= 2 && operator.indexOf("-") ===
0 && operator.indexOf("-", 1) === 1) {
      isTwosComplement = true;
      result = binarySum(twosComplement(arrayOfNumbers[1]),
twosComplement(arrayOfNumbers[2]))
    } else if (operator.length >= 2 && operator.indexOf("-") ===
0) {
      if (arrayOfNumbers[1] > arrayOfNumbers[2]) {
        isTwosComplement = true;
      }
      result = binarySum(twosComplement(arrayOfNumbers[1]),
arrayOfNumbers[2])
    }
  } else if (operator.includes("*")) {
    if (operator.includes("-")) {
      if (operator.indexOf("-") === 1) {
```

```

        isTwosComplement = true;
        result =
twosComplement(binaryMultiplication(arrayOfNumbers[0],
arrayOfNumbers[2]));
    } else if (operator.length === 2 && operator.indexOf("-")
=== 0) {
        isTwosComplement = true;
        result =
twosComplement(binaryMultiplication(arrayOfNumbers[1],
arrayOfNumbers[2]));
    } else if (operator.length === 3 && operator.indexOf("*")
=== 1) {
        result = binaryMultiplication(arrayOfNumbers[1],
arrayOfNumbers[3]);
    }
    } else {
        result = binaryMultiplication(arrayOfNumbers[0],
arrayOfNumbers[1]);
    }
    } else if (operator.includes("/")) {
        if (operator.includes("-")) {
            if (operator.indexOf("-") === 1) {
                isTwosComplement = true;
                result =
twosComplement(binaryDivision(arrayOfNumbers[0], arrayOfNumbers[2]));
            } else if (operator.length === 2 && operator.indexOf("-")
=== 0) {
                isTwosComplement = true;
                result =
twosComplement(binaryDivision(arrayOfNumbers[1], arrayOfNumbers[2]));
            } else if (operator.length === 3 && operator.indexOf("/")
=== 1) {
                result = binaryDivision(arrayOfNumbers[1],
arrayOfNumbers[3]);
            }
        } else {
            result = binaryDivision(arrayOfNumbers[0],
arrayOfNumbers[1]);
        }
    }
    }
    return result;

```

}

### 3.19 `errorAlert()` e `infoAlert()`

As funções `errorAlert()` e `infoAlert()` servem apenas para, respectivamente, retornar a mensagem de erro para o usuário e apresentar informações sobre o projeto.

#### Sintaxe

`errorAlert(message)`

`infoAlert()`

#### Parâmetros

Não se aplica.

### 3.20 `operationResult()`

A função `operationResult()` é responsável por tratar os possíveis erros cometidos pelo usuário e principalmente por mostrar os resultados (binário e decimal) na tela. No caso da conversão de binário para decimal a variável `isTwosComplement` é avaliada, para que a conversão de números negativos seja feita de forma correta.

#### Sintaxe

`operationResult(arrayOfNumbers)`

#### Parâmetros

Não se aplica.

```
const operationResult = () => {
  let decimalResult;

  splitInputNumbers(inputNumber);

  const arrayOfNumbers = splitNumbers.map(Number);

  if (
    inputNumber.textContent.includes("/0")
    || inputNumber.textContent.includes("/-0")
    || inputNumber.textContent.includes("/+0")
    || inputNumber.textContent.includes("/*0")
  ) {
    decimalResult = result = "Erro!";
    errorAlert("Parece que você está tentando cometer o pecado de
```

```

dividir por 0!");
    } else if (inputNumber.textContent === "") {
        errorAlert("Parece que você não inseriu nenhum valor!");
    } else if (
        inputNumber.textContent.match(/[\+\-\*\\/]/) &&
        !inputNumber.textContent.match(/[0-9]/)
    ) {
        errorAlert("Parece que você apenas inseriu os operadores e esqueceu de selecionar os valores!");
    } else if (arrayOfNumbers.some(isBiggerThan255)) {
        decimalResult = result = "Erro!";
        errorAlert("Parece que você inseriu um valor maior do 255!");
    } else {
        mathOperations(arrayOfNumbers);
    }

    if (overflow) {
        result = "Erro!";
        decimalResult = result;
        errorAlert("Pelo visto tivemos um overflow! Tente novamente!")
    }

    if (result !== "Erro!" && !overflow) {
        if (isTwosComplement) {
            decimalResult =
convertBinaryToDecimal(twosComplement(result)) * (-1)
        } else {
            decimalResult = convertBinaryToDecimal(result);
        }
        isTwosComplement = false;
    }

    inputLabel.innerText = "Resultado decimal";
    document.getElementById("displayBinary").innerHTML = result;
    document.getElementById("displayNumber").innerHTML =
decimalResult;
}

```

### 3.21 handleClick()

A função *handleClick()* retorna qual tecla do teclado virtual foi selecionada pelo usuário.

#### Sintaxe

*handleClick(key)*

#### Parâmetros

*key*: id da tecla selecionada.

```
const handleClick = (key) => {  
  switch (key) {  
    case "C":  
      clean();  
      break;  
    case "«":  
      backspace();  
      break;  
    case "=":  
      operationResult();  
      break;  
    default:  
      concatNumber(key);  
      break;  
  }  
}
```

### 3.22 keys.forEach()

Utiliza o array *keys* para gerar dinamicamente o teclado virtual na tela.

#### Sintaxe

Não se aplica.

#### Parâmetros

Não se aplica.

```
keys.forEach(key => {  
  const keyboardButton = document.createElement("button");  
  keyboardButton.textContent = key;  
  keyboardButton.setAttribute("key", key);  
  keyboardButton.addEventListener("click", () => handleClick(key));  
  keyboard.append(keyboardButton);  
});
```

```
});
```

## 4. Dificuldades e facilidades durante a execução do projeto

### 4.1 Facilidades

Devido ao fato de conhecimento prévio das linguagens *JS*, *HTML* e *CSS*, o projeto mostrou certa facilidade do ponto de vista geral. Não houve dificuldades na integração da aplicação *JS* com o front-end desenvolvido.

### 4.2 Dificuldades

A principal dificuldade encontrada nesse projeto foi simular a lógica de soma binária, pois uma vez que o JavaScript nos force métodos e funções tais como `eval()` e `toString()` (o que de certa forma diminuiria o trabalho em diversas linhas), não utilizá-los se mostrou penoso em alguns momentos. A lógica de conversão de número binário para decimal quando se era usado o complemento de dois para a conversão do resultado final, foi outra dificuldade encontrada nesse projeto.

Além disso, toda a lógica de tratamento de erros e de condicionais para identificar se o usuário está realizando a operação com nenhum, um ou ambos os números negativos também se mostrou bem desafiadora.



## 5. Como executar o projeto

É sugerido que a aplicação seja executada pelo programa VSCode, juntamente com as extensões Live Server (imagem 1) e Live Sass Compiler (imagem 2).

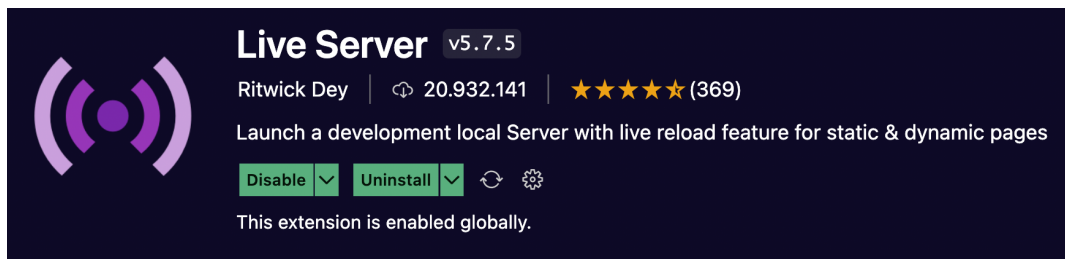


Imagem 1

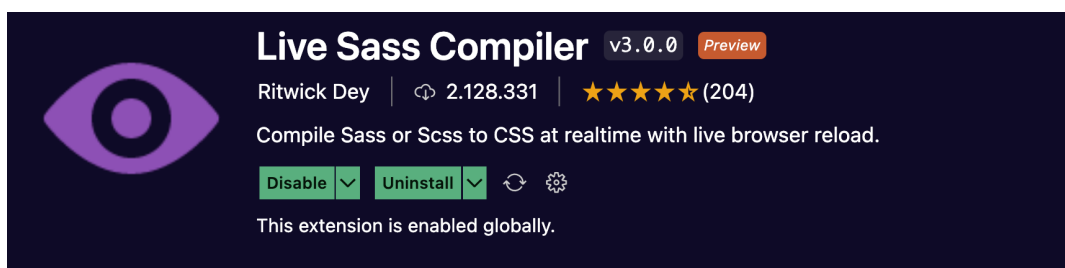


Imagem 2

Uma vez que o projeto esteja na máquina (seja através da clonagem do repositório ou os arquivos que foram disponibilizados), basta abrir a pasta principal do projeto no VSCode, clicar em Go Live no canto inferior direito e em Watch Sass (imagem 3).

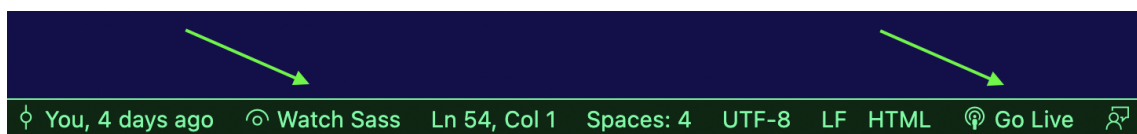


Imagem 3

Caso não seja do desejo do usuário executar o programa assim, basta acessar o link <https://didzao.github.io/calculadora-binaria/> para conferir a aplicação.

## Referências bibliográficas

1. Neldson, M. (n.d.). *Máquina de calcular*. Revista Pesquisa Fapesp. Retrieved April 22, 2022, from <https://revistapesquisa.fapesp.br/maquina-de-calcular/>
2. *Array.prototype.forEach() - JavaScript | MDN*. (2021, July 16). MDN Web Docs. Retrieved April 21, 2022, from [https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global\\_Objects/Array/forEach](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach)
3. *Array.prototype.includes() - JavaScript | MDN*. (2021, February 11). MDN Web Docs. Retrieved April 21, 2022, from [https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global\\_Objects/Array/includes](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array/includes)
4. *Array.prototype.indexOf() - JavaScript | MDN*. (2021, June 4). MDN Web Docs. Retrieved April 21, 2022, from [https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global\\_Objects/Array/indexOf](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array/indexOf)
5. *Array.prototype.join() - JavaScript | MDN*. (2021, June 4). MDN Web Docs. Retrieved April 21, 2022, from [https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global\\_Objects/Array/join](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array/join)
6. *Array.prototype.map() - JavaScript | MDN*. (2020, December 8). MDN Web Docs. Retrieved April 21, 2022, from

[https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global\\_Objects/Array/map](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array/map)

7. *Array.prototype.reduce()* - JavaScript | MDN. (2022, April 8). MDN Web Docs.

Retrieved April 24, 2022, from

[https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global\\_Objects/Array/Reduce](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce)

8. *Array.prototype.reverse()* - JavaScript | MDN. (2020, December 8). MDN Web Docs.

Retrieved April 24, 2022, from

[https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global\\_Objects/Array/reverse](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array/reverse)

9. *Array.prototype.some()* - JavaScript | MDN. (2021, July 16). MDN Web Docs. Retrieved

April 21, 2022, from

[https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global\\_Objects/Array/some](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Array/some)

10. *Math.pow()* - JavaScript | MDN. (2021, June 4). MDN Web Docs. Retrieved April 24,

2022, from

[https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global\\_Objects/Math/pow](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Math/pow)

11. *Number.prototype.toString()* - JavaScript | MDN. (2021, July 16). MDN Web Docs.

Retrieved April 21, 2022, from

[https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global\\_Objects/Number/toString](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Number/toString)

12. *parseInt()* - JavaScript | MDN. (2021, July 16). MDN Web Docs. Retrieved April 21,

2022, from

[https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global\\_Objects/pars  
eInt](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/pars<br/>eInt)

13. *String.length* - *JavaScript* | *MDN*. (2020, December 8). MDN Web Docs. Retrieved April 21, 2022, from

[https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global\\_Objects/Strin  
g/length](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Strin<br/>g/length)

14. *String.prototype.includes()* - *JavaScript* | *MDN*. (2020, December 8). MDN Web Docs. Retrieved April 21, 2022, from

[https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global\\_Objects/Strin  
g/includes](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Strin<br/>g/includes)

15. *String.prototype.lastIndexOf()* - *JavaScript* | *MDN*. (2020, December 8). MDN Web Docs. Retrieved April 21, 2022, from

[https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global\\_Objects/Strin  
g/lastIndexOf](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Strin<br/>g/lastIndexOf)

16. *String.prototype.padStart()* - *JavaScript* | *MDN*. (2021, June 4). MDN Web Docs. Retrieved April 21, 2022, from

[https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global\\_Objects/Strin  
g/padStart](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Strin<br/>g/padStart)

17. *String.prototype.slice()* - *JavaScript* | *MDN*. (2021, July 16). MDN Web Docs. Retrieved April 21, 2022, from

[https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global\\_Objects/Strin  
g/slice](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Strin<br/>g/slice)

18. *String.prototype.split()* - *JavaScript* | *MDN*. (2021, August 2). MDN Web Docs. Retrieved April 21, 2022, from

[https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global\\_Objects/String/split](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/String/split)

19. *String.prototype.substring()* - *JavaScript* | *MDN*. (2020, December 8). MDN Web Docs.

Retrieved April 21, 2022, from

[https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global\\_Objects/String/substring](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/String/substring)