UNIVERSIDADE ESTADUAL DO PARANÁ - *CAMPUS* APUCARANA

JOÃO PEDRO DE SOUZA OLIVO TARDIVO

DOCUMENTAÇÃO SOBRE O SIMULADOR DE MULTIPLICAÇÃO E DIVISÃO BINÁRIA

APUCARANA – PR
2023

JOÃO PEDRO DE SOUZA OLIVO TARDIVO

DOCUMENTAÇÃO SOBRE O SIMULADOR DE MULTIPLICAÇÃO E DIVISÃO BINÁRIA

Trabalho apresentado à disciplina de Arquitetura e Organização de Computadores, do curso de Bacharelado em Ciência da Computação.

Professor: Guilherme Nakahata

APUCARANA – PR
2023

SUMÁRIO

INSTRUÇÕES DE EXECUÇÃO

Windows

Execute *15BitBinaryMultiplierDivider.exe*

Linux

Abra o terminal na pasta que contém o *15BitBinaryMultiplierDivider*

Conceda permissão de execução para o arquivo através do comando:

```
chmod +x 15BitBinaryMultiplierDivider
```

Execute o programa através do comando:

```
./15BitBinaryMultiplierDivider
```

Ou duplo clique.

INSTRUÇÕES DE COMPILAÇÃO

Windows

Instale o Python encontrado em:

https://www.python.org/downloads/

Abra o terminal para instalar as dependências:

```
pip install PyQt6 PyInstaller
```

Abra o terminal na pasta com o código fonte.

Utilize o comando para gerar o executável na pasta dist:

```
pyInstaller main_window.py --onefile --noconsole
--icon=logo.ico --add-data "resources;resources"
```

Após isso siga as instruções de execução.

Linux

Abra o terminal e instale o Python:

Ubuntu/Debian

```
sudo apt install python3
```

Fedora

```
sudo dnf install python3
```

CentOS

```
sudo yum install centos-release-scl
sudo yum install rh-python36
scl enable rh-python36 bash
```

Arch

```
sudo pacman -S python
```

Instale o Package Installer for Python (pip):

Ubuntu/Debian

```
sudo apt install python3-pip
```

Fedora

```
sudo dnf install python3-pip
```

CentOS

```
sudo yum install python3-pip
```

Arch

```
sudo pacman -S python-pip
```

Ou utilizando o próprio Python

```
python3 get-pip.py
```

Instale as dependências:

```
sudo pip3 install pyinstaller pyqt6
```

Abra o terminal na pasta com o código fonte.

Utilize o comando para gerar o arquivo binário na pasta dist:

```
python3 -m PyInstaller main_window.py --onefile --noconsole
--icon=logo.ico --add-data "resources:resources"
```

Após isso siga as instruções de execução.

DOCUMENTAÇÃO

main_window.py

- **Primeiramente os módulos e classes necessários são importados:**

```
import os
import sys
from PyQt6.QtCore import Qt, QTimer, QEvent
from PyQt6.QtWidgets import QApplication, QMainWindow, QWidget, QLabel, QPushButton, QVBoxLayout, QHBoxLayout, QStackedLayout, QSpacerItem, QSizePolicy
from PyQt6.QtGui import QGuiApplication, QIcon
from multiplication_input_view import MultiplicationInputView
from division_input_view import DivisionInputView
```

os e **sys**: Usado para manipulação de caminhos de arquivos e operações relacionadas ao sistema.

PyQt6.QtCore, **PyQt6.QtWidgets**, **PyQt6.QtGui**: Módulos do PyQt6 para criação de GUI.

Qt: é uma enumeração dentro do módulo **PyQt6.QtCore** que contém constantes para vários eventos de teclado, mouse e outros eventos relacionados à entrada.

QTimer: é uma classe que fornece temporizadores repetitivos e de disparo único.

QEvent: é a classe base para todos os objetos de evento em PyQt6.

QApplication, **QMainWindow**, **QWidget**, **QLabel**, **QPushButton**, **QVBoxLayout**, **QHBoxLayout**, **QStackedLayout**, **QSpacerItem, QSizePolicy**: Classes PyQt6 para criação de elementos GUI, Application gerencia a instanciação ou execução da aplicação da além de ser utilizada dentro da própria MainWindow para obter as dimensões da tela do usuário, MainWindow para definir a janela principal, Widgets são estruturas genéricas do Qt estilo divs do HTML, labels são pequenos textos de títulos ou nomes, button é um botão, VBox é o layout vertical, HBox horizontal e Stacked é o layout em pilha, finalmente SpacerItem e SizePolicy controlam espaços em brancos e como um widget pode ser redimensionado.

QGuiApplication, **QIcon**: Classes PyQt6 para aplicações GUI e gerenciamento de ícones.

MultiplicationInputView, **DivisionInputView**: Classes personalizadas de módulos externos usados como parte da aplicação, elas direcionam o usuário para as telas de entrada para realizar as operações de multiplicação ou

divisão binária sem sinal e são dinamicamente adicionadas ou retiradas do layout em pilha caso estão em foco ou não.

Nota-se que devido a estrutura de layouts e widgets do PyQt, muitas vezes temos que criar widgets que estão contidos em layouts e posteriormente criamos widgets contêineres para segurar esses layouts para serem incluídos em outros layouts de hierarquia maior.

Isso inicialmente pode parecer bem confuso, mas é a maneira de conseguir resultados mais previsíveis e definidos para a estruturação e posicionamento de cada elemento de uma aplicação.

Novamente remete-se a analogia com o HTML que também segue uma grande estrutura hierárquica de vários componentes para popular o conteúdo de uma página.

- **Manipulação de caminhos do PyInstaller:**

```
9   ## PyInstaller file path handler
10  if getattr(sys, 'frozen', False):
11      # Running as a PyInstaller executable
12      base_path = sys._MEIPASS
13  else:
14      # Running as a script
15      base_path = os.path.abspath(".")
```

Verifica se o código está sendo executado como um executável PyInstaller ou como um script. Define o base_path de acordo para lidar com caminhos de arquivo, isso previne possíveis erros na execução do arquivo buildado.

- **Criando a janela principal do aplicativo (classe MainWindow):**

```python
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.multiplication_input_view = None
        self.division_input_view = None

        ## User screen's dimenions
        self.screen = QGuiApplication.primaryScreen()
        self.screen_size = self.screen.availableSize()

        self.setWindowTitle("Simulador de Multiplicacao e Divisao Sem Sinal em Binario")
        self.setWindowIcon(QIcon(os.path.join(base_path, 'resources', 'logo-unespar.jpg')))
        self.central_widget = QWidget()
        self.setCentralWidget(self.central_widget)

        layout = QVBoxLayout(self.central_widget)
        layout.setAlignment(Qt.AlignmentFlag.AlignCenter)
```

É uma subclasse do módulo QMainWindow do PyQt6, ou seja, essa será nossa janela “mestre” onde o layout em pilha meramente vai fazer a adição, remoção e troca do elemento ativo que será mostrado ao usuário.

Definimos variáveis auxiliares inicialmente vazias para as outras telas do aplicativo, que serão utilizadas conforme necessário no layout pilha.

Recuperamos as dimensões da tela para ajustes de layout.

Definimos o título e o ícone da janela.

Define o widget central da janela principal.

Criamos o layout principal como um layout vertical (QVBoxLayout) e definimos seu alinhamento.

- **Cabeçalho da aplicação e inicialização do layout em pilha:**

```python
        # Application Header
        title_label = QLabel("<h1>Simulador de Multiplicacao e Divisao Sem Sinal em Binario</h1>")
        title_label.setAlignment(Qt.AlignmentFlag.AlignCenter)
        layout.addWidget(title_label)

        # Stacked layout
        self.stacked_layout = QStackedLayout()
```

Criamos um rótulo de título e o adicionamos ao layout principal, logo estará presente em todas as telas da aplicação.

Definimos o alinhamento do rótulo do título para o centro.

Inicializamos o layout em pilha que será incrementado com conteúdo posteriormente.

- **Primeira tela da pilha: Boas-vindas:**

```
        # First stacked view: Welcome screen
        self.welcome_layout = QVBoxLayout()
        self.welcome_layout.setAlignment(Qt.AlignmentFlag.AlignCenter)

        self.choices_layout_label = QLabel("<h2>Bem vindo, escolha a operacao a ser realizada</h2>")
        self.choices_layout_label.setAlignment(Qt.AlignmentFlag.AlignCenter)
        self.welcome_layout.addWidget(self.choices_layout_label)

        self.buttons_layout = QHBoxLayout()
        self.buttons_layout.setAlignment(Qt.AlignmentFlag.AlignCenter)

        self.button_multiplication_input = (QPushButton("Multiplicacao"))
        self.button_multiplication_input.clicked.connect(self.show_multiplication_input_view)
        self.button_division_input = (QPushButton("Divisao"))
        self.button_division_input.clicked.connect(self.show_division_input_view)

        self.buttons_layout.addWidget(self.button_multiplication_input)
        self.buttons_layout.addWidget(self.button_division_input)

        self.welcome_layout.addLayout(self.buttons_layout)

        ## Finalizing welcome screen layout into a widget
        self.welcome_container = QWidget()
        self.welcome_container.setLayout(self.welcome_layout)
        self.welcome_container.setFixedWidth(int(self.screen_size.width() * 0.80))
```

Criamos um layout vertical (welcome_layout) com alinhamento centralizado.

Criamos um rótulo de boas-vindas e o adicionamos ao layout de boas-vindas.

Criamos botões para telas de multiplicação e de divisão e os conecta às suas respectivas funções.

Adicionamos os botões a um layout horizontal (buttons_layout).

Definimos o layout de boas-vindas para o seu widget contêiner.

Estabelecemos as dimensões do layout para 80% da largura da tela.

- **Finalizando o layout em pilha:**

```
        ## Adding all views to stacked layout
        self.stacked_layout.addWidget(self.welcome_container)
        self.stacked_layout.setAlignment(Qt.AlignmentFlag.AlignCenter)

        ## Finalizing stacked layout into a widget
        self.stacked_layout_widget = QWidget()
        self.stacked_layout_widget.setLayout(self.stacked_layout)

        ## Finalizing widgets into main application layout
        self.main_application_layout = QHBoxLayout()

        self.left_spacer = QSpacerItem(0, 0, QSizePolicy.Policy.Expanding, QSizePolicy.Policy.Minimum)
        self.main_application_layout.addItem(self.left_spacer)

        self.main_application_layout.addWidget(self.stacked_layout_widget)

        self.right_spacer = QSpacerItem(0, 0, QSizePolicy.Policy.Expanding, QSizePolicy.Policy.Minimum)
        self.main_application_layout.addItem(self.right_spacer)
        self.main_application_layout.setAlignment(Qt.AlignmentFlag.AlignCenter)

        layout.addLayout(self.main_application_layout)

        self.showMaximized()
```

Adicionamos o widget de boas vindas ao layout empilhado.

Definimos o widget de layout empilhado e colocamos na layout principal da aplicação.

Estabelecemos alguns itens de espaçamento para centralizar o layout empilhado.

Inicializamos a aplicação maximizada.

- **Funções dessa classe principal:**

```
    def changeEvent(self, event):
        if event.type() == QEvent.Type.WindowStateChange:
            if self.windowState() & Qt.WindowState.WindowMaximized:
                self.isMaximized = True
            else:
                if self.isMaximized:
                    self.center_on_screen()
                self.isMaximized = False
```

Definimos uma função changeEvent para lidar com a alteração no estado da janela de maximizado para janela.

```
 99      def center_on_screen(self):
100          screen_geometry = QApplication.primaryScreen().availableGeometry()
101
102          center_x = int((screen_geometry.width() - self.width()) / 2)
103          center_y = int((screen_geometry.height() - self.height()) / 2.5)
104
105          self.move(center_x, center_y)
```

Definimos uma função center_on_screen para centralizar a janela na tela calculando a diferença entre o espaço total da tela do usuário e o ocupado pela janela da aplicação.

```
112      def show_main_menu(self):
113          self.stacked_layout.setCurrentWidget(self.welcome_container)
114
115          self.destroy_multiplication_input_view()
116          self.destroy_division_input_view()
117
118      def show_multiplication_input_view(self):
119          self.multiplication_input_view = MultiplicationInputView(self.show_main_menu)
120          self.multiplication_input_view.setFixedWidth(int(self.screen_size.width() * 0.80))
121
122          self.stacked_layout.addWidget(self.multiplication_input_view)
123          self.stacked_layout.setCurrentWidget(self.multiplication_input_view)
124
125          if not self.isMaximized:
126              QTimer.singleShot(0, self.center_on_screen)
127
128      def show_division_input_view(self):
129          self.division_input_view = DivisionInputView(self.show_main_menu)
130          self.division_input_view.setFixedWidth(int(self.screen_size.width() * 0.80))
131
132          self.stacked_layout.addWidget(self.division_input_view)
133          self.stacked_layout.setCurrentWidget(self.division_input_view)
134
135          if not self.isMaximized:
136              QTimer.singleShot(0, self.center_on_screen)
```

Definimos funções para mostrar a tela de boas-vindas e passar para as telas de multiplicação e divisão.

Essencialmente criamos uma nova instância do layout desejado, e colocamos ele no topo da pilha.

```
138     ## Clean up functions
139     def destroy_multiplication_input_view(self):
140         if self.multiplication_input_view:
141             self.multiplication_input_view.deleteLater()
142             self.stacked_layout.removeWidget(self.multiplication_input_view)
143             self.multiplication_input_view.deleteLater()
144             self.multiplication_input_view = None
145 
146     def destroy_division_input_view(self):
147         if self.division_input_view:
148             self.division_input_view.deleteLater()
149             self.stacked_layout.removeWidget(self.division_input_view)
150             self.division_input_view.deleteLater()
151             self.division_input_view = None
```

Definimos funções auxiliares destroy_multiplication_input_widget e destroy_division_input_widget para remover e excluir os widgets atuais ao alternar entre telas, poupando memória.

Essencialmente tiramos o layout da pilha e o deletamos quando o botão de voltar nas outras telas é clicado.

- **Finalmente, o bloco if __name__ == "__main__":**

```
139 if __name__ == "__main__":
140     app = QApplication(sys.argv)
141     window = MainWindow()
142     sys.exit(app.exec())
```

Inicializa a aplicação PyQt6 (app).

Cria uma instância da classe MainWindow (janela).

Inicia o loop de eventos da aplicação com app.exec().

- **Em resumo, main_window.py:**

Cria um aplicativo GUI com um layout empilhado que alterna entre uma tela de boas-vindas, uma tela de multiplicação e uma tela de divisão. Ele também lida com alterações de estado da janela e fornece funções para centralizar a janela na tela.

multiplication_input_view.py

- **Primeiramente os módulos e classes necessários são importados:**

```
1  from PyQt6.QtCore import Qt
2  from PyQt6.QtWidgets import QWidget, QLabel, QPushButton, QVBoxLayout, QHBoxLayout, QLineEdit, QMessageBox, QScrollArea
3  from logic import binaryMultiply, convertDecimalToBinary, convertBinaryArrayToDecimal, generateArr
```

PyQt6.QtCore, PyQt6.QtWidgets: Módulos do PyQt6 para criação de GUI.

Qt: é uma enumeração dentro do módulo **PyQt6.QtCore** que contém constantes para vários eventos de teclado, mouse e outros eventos relacionados à entrada.

QWidget, QPushButton, QVBoxLayout, QLabel, QHBoxLayout, QLineEdit, QScrollArea, QMessageBox: Classes PyQt6 para criação de elementos GUI, similar a explicação de main_window.py, as novidades aqui são a ScrollArea que como o nome indica cria uma área de espaço definido que é expandida através do uso de barras de rolagem, LineEdit que possibilita o input do usuário através de uma linha como o nome implica, e finalmente MessageBox que é uma caixa de pop up utilizada para alertas ou mensagens de erro.

binaryMultiply, convertDecimalToBinary, convertBinaryArrayToDecimal, generateArr: Funções auxiliares que contém a lógica de execução da multiplicação, bem como tratamento de entrada, convertendo um número decimal em uma array de binários, e este para um número decimal novamente para mostrar ao usuário.

- **Criando o widget da tela de multiplicação (classe MultiplicationInputView):**

```python
class MultiplicationInputView(QWidget):
    def __init__(self, show_main_menu_callback):
        super().__init__()

        self.show_main_menu_callback = show_main_menu_callback

        self.manual_input_layout = QVBoxLayout()
        self.manual_input_layout.setAlignment(Qt.AlignmentFlag.AlignCenter)

        self.manual_input_layout_label = QLabel("<h2>Multiplicacao Sem Sinal</h2>")
        self.manual_input_layout_label.setAlignment(Qt.AlignmentFlag.AlignCenter)
        self.manual_input_layout.addWidget(self.manual_input_layout_label)
```

É uma subclasse do módulo widget do PyQt, como se fosse uma div do HTML, extremamente customizável.

Iniciamos a classe passando uma função da classe principal como argumento na forma de “callback” para que a mesma função seja invocada. Essa função serve para voltar a tela de boas vindas.

Criamos um rótulo de título e o adicionamos ao layout principal.

Definimos o alinhamento do rótulo do título para o centro.

- **Cabeçalho com botões de multiplicar, resetar e voltar:**

```python
        ## Header buttons
        self.continue_button_layout = QHBoxLayout()
        self.continue_button_layout.setAlignment(Qt.AlignmentFlag.AlignCenter)
        self.button_script_validate_input = (QPushButton("Multiplicar"))
        self.button_script_validate_input.clicked.connect(self.validate_and_multiply)
        self.continue_button_layout.addWidget(self.button_script_validate_input)
        self.manual_input_layout.addLayout(self.continue_button_layout)

        self.choices_bottom = QHBoxLayout()
        self.choices_bottom.setAlignment(Qt.AlignmentFlag.AlignCenter)
        self.button_script_show_options_menu = (QPushButton("Resetar"))
        self.button_script_show_options_menu.clicked.connect(self.reset)
        self.choices_bottom.addWidget(self.button_script_show_options_menu)
        self.manual_input_layout.addLayout(self.choices_bottom)
```

```
        self.back_button_layout = QHBoxLayout()
        self.back_button_layout.setAlignment(Qt.AlignmentFlag.AlignCenter)
        self.manual_input_layout_back_button = (QPushButton("Voltar"))
        self.manual_input_layout_back_button.clicked.connect(self.show_main_menu_callback)
        self.back_button_layout.addWidget(self.manual_input_layout_back_button)
        self.manual_input_layout.addLayout(self.back_button_layout)
```

Criamos alguns botões para fornecer diversas funcionalidades ao usuário.

Estão localizados no topo da tela para serem mais acessíveis aos usuários em diversos estados da aplicação.

Multiplicar verifica as entradas e realiza a lógica com os dados inseridos caso validados.

Resetar volta a tela ao estado padrão.

Voltar retorna a tela de boas vindas.

Conectamos os botões às suas respectivas funções.

- **Entrada do multiplicando e multiplicador:**

```
        ## Input fields
        self.variables_layout = QHBoxLayout()
        self.variables_layout.setAlignment(Qt.AlignmentFlag.AlignCenter)

        self.variables_layout_multiplicand_label = QLabel("<h3>Multiplicando</h3>")
        self.variables_layout_multiplicand_label.setAlignment(Qt.AlignmentFlag.AlignCenter)
        self.variables_layout.addWidget(self.variables_layout_multiplicand_label)

        self.variables_layout_multiplicand_input = QLineEdit()
        self.variables_layout_multiplicand_input.setAlignment(Qt.AlignmentFlag.AlignCenter)
        self.variables_layout_multiplicand_input.setPlaceholderText("Insira o multiplicando")
        self.variables_layout.addWidget(self.variables_layout_multiplicand_input)

        self.variables_layout_multiplier_label = QLabel("<h3>Multiplicador</h3>")
        self.variables_layout_multiplier_label.setAlignment(Qt.AlignmentFlag.AlignCenter)
        self.variables_layout.addWidget(self.variables_layout_multiplier_label)

        self.variables_layout_multiplier_input = QLineEdit()
        self.variables_layout_multiplier_input.setAlignment(Qt.AlignmentFlag.AlignCenter)
        self.variables_layout_multiplier_input.setPlaceholderText("Insira o multiplicador")
        self.variables_layout.addWidget(self.variables_layout_multiplier_input)

        self.manual_input_layout.addLayout(self.variables_layout)
```

Criamos um layout como contêiner deste item.

Criamos QLabels e QLineEdits para o multiplicando e o multiplicador para que o usuário preencha com os valores que desejar em um layout horizontal.

Também utilizamos texto placeholder para auxiliar o usuário.

- **Display do resultado com os valores inseridos:**

```
67          self.output_layout = QVBoxLayout()
68          self.output_layout.setAlignment(Qt.AlignmentFlag.AlignCenter)
69
70          self.output_layout_label = QLabel("<h2>Passos realizados na multiplicacao binaria sem sinal</h2>")
71          self.output_layout_label.setAlignment(Qt.AlignmentFlag.AlignCenter)
72          self.output_layout.addWidget(self.output_layout_label)
73
74          self.output_contents_layout = QHBoxLayout()
75          self.output_contents_layout.setAlignment(Qt.AlignmentFlag.AlignCenter)
76
77          self.output_contents_layout_label = QLabel("")
78          self.output_contents_layout_label.setAlignment(Qt.AlignmentFlag.AlignCenter)
79          self.output_contents_layout.addWidget(self.output_contents_layout_label)
80
81          self.output_layout.addLayout(self.output_contents_layout)
82
83
84          self.manual_input_layout.addLayout(self.output_layout)
85
86          # output_layout items start invisible
87          for i in range(self.output_layout.count()):
88              item = self.output_layout.itemAt(i)
89
90              if isinstance(item.widget(), QWidget):
91                  item.widget().setVisible(False)
```

Criamos um layout como contêiner deste item, inicialmente é invisível com o loop for abaixo.

Criamos apenas um QLabel para este item em um layout vertical.

Quando a aplicação realiza a multiplicação binária, o conteúdo do QLabel é alterado para mostrar cada passo do fluxograma com as devidas somas e bit shifts realizados ao usuário.

- **ScrollArea e finalização do layout:**

```
        self.scroll_container_layout = QVBoxLayout()

        scroll_area = QScrollArea()
        scroll_area.setWidgetResizable(True)

        scroll_content = QWidget()
        scroll_content.setLayout(self.manual_input_layout)

        scroll_area.setWidget(scroll_content)
```

```

        self.scroll_container_layout.addWidget(scroll_area)


        self.setLayout(self.scroll_container_layout)
```

Finalmente, encapsulamos o layout em uma ScrollArea para possibilitar visibilidade mesmo com resoluções pequenas ou outputs grandes.

- **Funções dessa classe de multiplicação:**

```
    def validate_and_multiply(self):
        multiplicand = self.variables_layout_multiplicand_input.text()
        multiplier = self.variables_layout_multiplier_input.text()

        if not self.validate_value(multiplicand, "multiplicando"):
            return
        else:
            multiplicand = int(self.variables_layout_multiplicand_input.text())

        if not self.validate_value(multiplier, "multiplicador"):
            return
        else:
            multiplier = int(self.variables_layout_multiplier_input.text())

        binary_multiplicand = convertDecimalToBinary(multiplicand)
        binary_multiplier = convertDecimalToBinary(multiplier)
        multiplicand_binary_array = generateArr(binary_multiplicand)
        multiplier_binary_array = generateArr(binary_multiplier)
```

```
        result, log = binaryMultiply(multiplicand_binary_array, multiplier_binary_array)
        decimalResult = convertBinaryArrayToDecimal(result)

        log.insert(0, f"{multiplicand} X {multiplier} = {decimalResult}")
        log.insert(0, "@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@")

        self.output_contents_layout_label.setText('\n'.join(log))
```

Definimos uma função validate_and_multiply para verificar se os dados inseridos pelo usuário são válidos e caso positivo executar a lógica da multiplicação binária.

Utilizamos funções auxiliares para realizar a verificação, se o input de fato contém alguma coisa, se esta coisa é um número, se é positivo, inteiro e menor que 32767 (limite representado por 15 bits). Caso contrário as funções auxiliares retornam False, e a função validate_and_multiply acaba neste ponto com um retorno precoce. Esta verificação é realizada para os dois valores inseridos pelo usuário.

Caso a verificação for bem sucedida, as funções auxiliares importadas são utilizadas para converter os números decimais inseridos para uma array de binários e chamar a lógica principal para realizar a multiplicação que retorna o valor resultante e uma array de log descrevendo cada passo da operação.

Por fim, o log e o resultado são mostrados para o usuário através da edição do elemento “output_contents_layout_label”.

```
    def validate_value(self, value, what_is_it):
        if not self.is_integer(value):
            QMessageBox.warning(self, "Valor invalido!", f"O valor inserido para o {what_is_it} e inval
            return False
        elif not self.is_integer_float(value):
            QMessageBox.warning(self, "Valor invalido!", f"O valor inserido para o {what_is_it} e inval
            return False
        elif (int(value) > 32767):
            QMessageBox.warning(self, "Valor invalido!", f"O valor inserido para o {what_is_it} e inval
            return False
        elif (int(value) < 0):
            QMessageBox.warning(self, "Valor invalido!", f"O valor inserido para o {what_is_it} e inval
            return False
        else:
            return True
```

```
152     def is_integer(self, value):
153         try:
154             int(value)
155             return True
156         except ValueError:
157             return False
158 
159     def is_integer_float(self, value):
160         try:
161             float_value = float(value)
162             int_value = int(float_value)
163             return float_value == int_value
164         except (ValueError, TypeError):
165             return False
```

Funções auxiliares de verificação.

```
167     def reset(self):
168         self.variables_layout_multiplicand_input.setText("")
169         self.variables_layout_multiplier_input.setText("")
170         self.output_contents_layout_label.setText("")
171 
172         for i in range(self.output_layout.count()):
173             item = self.output_layout.itemAt(i)
174 
175             if isinstance(item.widget(), QWidget):
176                 item.widget().setVisible(False)
```

Função reset retorna a tela ao estado inicial. Consiste em “esvaziar” ou “zerar” variáveis, loops for deixando elementos invisíveis novamente.

- **Em resumo, multiplication_input_view.py:**

Cria um widget que pode ser instanciado e adicionado para o layout em pilha da classe principal. Este widget recebe os valores de entrada do usuário e após sua verificação executam a lógica de multiplicação binária, mostrando o passo a passo na tela.

division_input_screen.py

Extremamente similar a tela de multiplicação porém com nomes dos valores trocados e com chamadas de outra lógica para realizar a operação de divisão. Além disso, a verificação contém uma cláusula adicional no divisor para evitar a divisão por 0.

logic.py

- **Criando a funções auxiliares de conversão de decimal para binário e de binário para decimal:**

```python
import math

def convertDecimalToBinary(integer):
    binaryValue = ""
    while (integer > 0):
        binaryValue = str(integer % 2) + binaryValue
        integer = integer // 2
    return binaryValue

def convertBinaryArrayToDecimal(binaryArray):
    decimalValue = 0

    for i in range(len(binaryArray)):
        if(binaryArray[i] == 1):
            decimalValue += math.pow(2, len(binaryArray)-1-i)

    return int(decimalValue)
```

A biblioteca math do Python é importada para utilizar a potenciação.

Para a conversão de decimal para binário a lógica do resto da divisão é utilizada. Nota-se que os novos números são concatenados antes dos atuais pela lógica de “ler os resultados ao contrário” da conversão binária para decimal. Além disso, utiliza-se a operador % mod e // divisão inteira para separar o quociente do resto até que o valor se torne 0.

Para a conversão de binário para decimal, percorremos a array e realizamos a exponenciação adequada ao número 2. Desta forma, um valor “5” por exemplo = “0101” terá o 1 à esquerda equivalente a 2 elevado ao quadrado e o 1 à direita será equivalente a 2 elevado a 0, resultando em 4 + 1 = 5.

- **Criando a função auxiliar generateArr**:

```python
def generateArr(binaryString):
    binaryArray = [0] * 15
    zeroes = 15 - len(binaryString)

    for i in range(zeroes):
        binaryArray[i] = 0

    for i in range(len(binaryString)):
        binaryArray[zeroes+i] = int(binaryString[i])

    return binaryArray
```

Esta função transforma uma string binária em uma array de inteiros, o que torna a manipulação da soma mais simples nos próximos passos. Como o requisito deste trabalho é 15 bit, este é o tamanho designado para a array de binários.

Em suma, verifica-se o tamanho da string, que sempre terá o bit mais significativo como 1, a não ser que o número seja 0. Desta forma o exemplo do 5 anteriormente terá a string “101” de length “3”. Assim, é possível calcular quantos 0 serão adicionados para preencher os 15 bits da array, bem como onde posicionar os bits que representam o número.

Neste exemplo zeroes será 15 - 3 = 12, ou seja o loop for preencherá os 12 primeiros índices da array com zeros. Após isso os demais índices serão preenchidos com os valores dos bits que representam o número.

- **Criando a função auxiliar binarySum**:

```python
def binarySum(binaryArrayA, binaryArrayB):
    binaryArraySum = []
    carryOut = 0

    for i in reversed(range(len(binaryArrayA))):
        total = binaryArrayA[i] + binaryArrayB[i] + carryOut
        binaryArraySum.insert(0, total % 2)
        carryOut = total // 2

    return binaryArraySum, carryOut
```

Esta função recebe duas arrays binárias como argumentos e retorna uma tupla que contém uma array binária de soma e o carry out caso tenha ocorrido um overflow de bits para seguir a lógica da multiplicação binária.

A soma é realizada de uma forma simples, inicializamos uma variável local de carry out, inicialmente 0, e realizamos um loop invertido das arrays de entrada da função seguindo a mesma forma que são representadas no fluxograma, ou seja, quanto maior o índice menos significativo é o bit e vice e versa.

Um total é criado para cada iteração, somando os dois valores do mesmo índice das duas arrays mais o carry out. Utilizamos o operador mod % e a soma inteira // para criar uma lógica similar a uma tabela verdade de valores:

Valor Decimal	Soma % 2	Carry Out // 2
0	0	0
1	1	0
2	0	1
3	1	1

Nota-se que os valores sempre seguem a representação binária, 0 ou 1, os valores inseridos seguem a lógica de 0 + 1 = 1, 1 + 1 = 0 com o carry out = 1 e finalmente 1 + 1 + 1 = 1 com o carry out permanecendo 1. Assim é possível realizar a soma binária de forma extremamente simples.

Utilizamos o insert sempre no índice 0 para seguir a mesma lógica mencionada anteriormente de quanto maior o índice menos significativo é o bit e vice e versa.

- **Criando a função auxiliar twoComplement**:

```python
def twoComplement(binaryArray):
    complement = [0] * len(binaryArray)
    oneArray = [0] * len(binaryArray)
    oneArray[len(binaryArray) - 1] = 1

    for i in range(len(binaryArray)):
        if (binaryArray[i] == 0):
            complement[i] = 1
        else:
            complement[i] = 0

    complement, carryOut = binarySum(complement, oneArray)

    return complement
```

Função simples para realizar o complemento de 2, a array de entrada é percorrida com um loop e os valores 0 são transformados em 1 e similarmente os valores 1 são transformados em 0.

Ademais, a função cria uma array auxiliar com o valor binário 1, para realizar o último passo do complemento de 2. A função auxiliar binarySum é utilizada para este passo.

Nota-se a variável inutilizada carryOut na chamada da função, ela ainda é relevante pois a função binarySum retorna uma tupla com a array e o carry out, desta forma, duas variáveis são necessárias para receber o valor de cada índice da tupla, caso contrário “complement” receberia a tupla inteira o que não é um comportamento desejável.

- **Criando as funções auxiliares shiftRight e shiftLeft**:

```
57  def shiftRight(carryOut, binaryArrayA, binaryArrayB):
58      transferredValue = 0
59
60      for i in reversed(range(len(binaryArrayA))):
61          if(i == len(binaryArrayA) - 1):
62              transferredValue = binaryArrayA[i]
63
64          if(i > 0):
65              binaryArrayA[i] = binaryArrayA[i-1]
66          else:
67              binaryArrayA[i] = carryOut
68
69      for i in reversed(range(len(binaryArrayB))):
70          if(i > 0):
71              binaryArrayB[i] = binaryArrayB[i-1]
72          else:
73              binaryArrayB[i] = transferredValue
74
75      return binaryArrayA, binaryArrayB
```

Função fundamental para o passo de mover todos os bits para a direita no fluxograma da multiplicação, recebe o carry out atual e duas arrays binárias como argumento. Esta função utiliza uma lógica simples de realizar um loop inverso, trocando o valor do índice atual para o próximo, o que efetivamente causa o efeito de mudar a posição de cada valor para a direita.

Uma variável auxiliar e o valor do carry out são utilizados para armazenar e atribuir os valores que “pulam” de uma array para outra corretamente.

```
77  def shiftLeft(binaryArrayA, binaryArrayB):
78      transferredValue = 0
79
80      for i in range(len(binaryArrayB)):
81          if(i == 0):
82              transferredValue = binaryArrayB[i]
83
84          if(i < len(binaryArrayB) - 1):
85              binaryArrayB[i] = binaryArrayB[i+1]
86          else:
87              binaryArrayB[i] = 0
88
89      for i in range(len(binaryArrayA)):
90          if(i < len(binaryArrayA) - 1):
91              binaryArrayA[i] = binaryArrayA[i+1]
92          else:
93              binaryArrayA[i] = transferredValue
94
95      return binaryArrayA, binaryArrayB
```

shiftLeft utiliza o mesmo conceito, porém ao contrário para realizar o deslocamento dos bits para a esquerda.

- **Criando a função binaryMultiply**:

```
97  def binaryMultiply(binaryArrayA, binaryArrayB):
98      log = []
99      carryOut = 0
100     varA = [0] * len(binaryArrayA)
101     varM = binaryArrayA
102     varQ = binaryArrayB
103     result = [0] * (len(binaryArrayA) + len(binaryArrayB))
104
105     for i in range(len(binaryArrayA)):
106         log.append("=============================================")
107         log.append("Inicio da iteracao")
108         log.append("Contador: "+str(15-i))
109         log.append("Q0: "+str(varQ[len(varQ)-1]))
110         log.append("C: "+str(carryOut))
111         log.append("A: "+str(varA))
112         log.append("Q: "+str(varQ))
113
114         if(varQ[len(varQ)-1] == 1):
115             varA, carryOut = binarySum(varA, varM)
116             log.append("Apos soma")
117             log.append("C: "+str(carryOut))
118             log.append("A: "+str(varA))
119
120         varA, varQ = shiftRight(carryOut, varA, varQ)
121         carryOut = 0
122
123         log.append("Apos deslocamento")
124         log.append("C: "+str(carryOut))
125         log.append("A: "+str(varA))
126         log.append("Q: "+str(varQ))
```

```
128     for i in range(len(varA)):
129         result[i] = varA[i]
130
131     for i in range(len(varQ)):
132         result[i+len(varA)] = varQ[i]
133
134     log.append("=============================================")
135     log.append("Final da operacao")
136     log.append("Q0: "+str(varQ[len(varQ)-1]))
137     log.append("C: "+str(carryOut))
138     log.append("A: "+str(varA))
139     log.append("Q: "+str(varQ))
140     log.append("M: "+str(varQ))
141     log.append("Resultado em binario: "+str(result))
142     log.append("Resultado em decimal: "+str(convertBinaryArrayToDecimal(result)))
143     log.append("=============================================")
144
145     return result, log
```

Esta função realiza a lógica principal do fluxograma da multiplicação binária. Várias variáveis locais auxiliares são instanciadas, representando o C,

A, Q, M do fluxograma. Além disso uma variável vazia com o tamanho de A + Q é criada para receber o resultado e uma array log é criada para salvar cada passo realizado do fluxograma para ser mostrado ao usuário.

Loop for que depende do tamanho da array binária, ou seja, a quantidade de bits. Se o primeiro valor de Q é igual a 1, uma operação de soma com A e M é realizada, o resultado sobrescreve o valor atual de A e também o valor atual do carry out. Por fim os bits são deslocados para a direita e o ciclo se repete N vezes onde N é o número de bits dos números binários inseridos, neste caso 15 conforme os requisitos do trabalho.

Por fim, os bits de A e Q são preenchidos na array de resultado. A função retorna uma tupla com a array binária do resultado e a array de log.

- **Criando a função binaryDivide**:

```
147  def binaryDivide(binaryArrayA, binaryArrayB):
148      log = []
149      varA = [0] * len(binaryArrayA)
150      varM = binaryArrayA
151      varMTwoComplement = twoComplement(varM)
152      varQ = binaryArrayB
153
154      for i in range(len(binaryArrayA)):
155          log.append("================================================")
156          log.append("Inicio da iteracao")
157          log.append("Contador: "+str(15-i))
158          log.append("Q0: "+str(varQ[len(varQ)-1]))
159          log.append("A: "+str(varA))
160          log.append("Q: "+str(varQ))
161
162          varA, varQ = shiftLeft(varA, varQ)
163          log.append("Apos deslocamento")
164          log.append("Q0: "+str(varQ[len(varQ)-1]))
165          log.append("A: "+str(varA))
166          log.append("Q: "+str(varQ))
167
168          varA, carryOut = binarySum(varA, varMTwoComplement)
169          log.append("Apos subtracao (soma com complemento de 2)")
170          log.append("A: "+str(varA))
```

```
172         if(varA[0] == 1):
173             varQ[len(varQ)-1] = 0
174             varA, carryOut = binarySum(varA, varM)
175             log.append("A < 0")
176             log.append("Apos soma")
177             log.append("Q0: "+str(varQ[len(varQ)-1]))
178             log.append("A: "+str(varA))
179             log.append("Q: "+str(varQ))
180         else:
181             varQ[len(varQ)-1] = 1
182             log.append("A > 0")
183             log.append("Q0: "+str(varQ[len(varQ)-1]))
184             log.append("Q: "+str(varQ))
185 
186     log.append("================================================")
187     log.append("Final da operacao")
188     log.append("Q0: "+str(varQ[len(varQ)-1]))
189     log.append("M: "+str(varQ))
190     log.append("Quociente em binario: "+str(varQ))
191     log.append("Quociente em decimal: "+str(convertBinaryArrayToDecimal(varQ)))
192     log.append("Resto em binario: "+str(varA))
193     log.append("Resto em decimal: "+str(convertBinaryArrayToDecimal(varA)))
194     log.append("================================================")
195 
196     return varQ, varA, log
```

Função similar a multiplicação, porém obviamente seguindo o fluxograma da divisão. Algumas diferenças chave são: uma variável separada para armazenar o complemento de 2 de M utilizando a função auxiliar descrita anteriormente e a verificação do bit mais significativo de A para verificar se o número binário sem sinal é positivo ou não.

A função retorna uma tupla com a array binária do quociente, resto e a array de log.