UNIVERSIDADE ESTADUAL DO PARANÁ - *CAMPUS* APUCARANA

JOÃO PEDRO DE SOUZA OLIVO TARDIVO

DOCUMENTAÇÃO SOBRE O SIMULADOR DE CICLOS DE INSTRUÇÕES

APUCARANA – PR
2023

JOÃO PEDRO DE SOUZA OLIVO TARDIVO

DOCUMENTAÇÃO SOBRE O SIMULADOR DE CICLOS DE INSTRUÇÕES

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 *simulador_instrucoes.exe*

Linux

Abra o terminal na pasta que contém o *simulador_instrucoes*

Conceda permissão de execução para o arquivo através do comando

```
chmod +x simulador_instrucoes
```

Execute o programa através do comando

```
./simulador_instrucoes
```

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:**

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.

QEvent: Uma classe do PyQt6 usada para lidar com eventos..

QApplication, **QMainWindow**, **QWidget**, **QLabel**, **QPushButton**, **QVBoxLayout**, **QHBoxLayout**, **QStackedLayout**, **QMessageBox**: Classes PyQt6 para criação de elementos GUI, aplicação gerencia várias coisas mas aqui é utilizada para obter as dimensões da janela, main window para definir a janela principal, widget 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 horizonta e Stacked é o layout em pilha, finalmente MessageBox é uma caixa de pop up utilizada para alertas.

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

ManualInputScreen, **FileInputScreen**: Classes personalizadas de módulos externos usados como parte da aplicação, elas direcionam o usuário para as telas de entrada manual ou por arquivo 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:**

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):**

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 o título e o ícone da janela.

Define o widget central da janela principal.

Recuperamos as dimensões da tela para ajustes de layout.

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

- **Cabeçalho da aplicação:**

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

Definimos o alinhamento do rótulo do título para o centro.

- **Layout em pilha:**

Configuramos um widget de contêiner (stacked_layout_container) e um layout em pilha (stacked_layout) para gerenciar múltiplas visualizações.

Garantimos que o alinhamento do layout empilhado esteja centralizado.

- **Primeira tela empilhada: Boas-vindas:**

Criamos um widget de contêiner (welcome_layout_container) com dimensões mínimas.

Configuramos um layout vertical (welcome_layout) para esta tela e garantimos que seu alinhamento esteja centralizado.

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

Criamos botões para telas de entrada manual e de arquivos e os conecta às suas respectivas funções.

Adicionamos os botões a um layout horizontal (welcome_buttons_layout) dentro de um contêiner (welcome_buttons_container).

Adicionamos o contêiner ao layout de boas-vindas.

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

Adicionamos o contêiner de layout de boas-vindas ao layout em pilha.

- **Finalizando o layout empilhado:**

  Definimos o layout empilhado para o contêiner de layout empilhado.

  Adicionamos o contêiner de layout empilhado ao layout principal.

- **Personalização de layout:**

  Definimos espaçamento e margens para o layout principal.

- **Funções dessa classe principal:**

Definimos uma função changeEvent para lidar com alterações no estado da janela (por exemplo, maximizar ou restaurar a janela).

Definimos uma função center_on_screen para centralizar a janela na tela.

Definimos uma função show_alert_box para exibir uma caixa de mensagem informativa com título e texto que são passados como argumentos, desta forma toda vez que precisarmos de uma mensagem customizada de alerta na aplicação podemos invocar esta função.

Definimos funções para mostrar a tela de boas-vindas e passar para as telas de entrada manual e de arquivos.

Definimos uma função destroy_input_widget para remover e excluir o widget de entrada atual ao alternar entre telas, poupando memória.

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

  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 entrada manual e uma tela de entrada de arquivo. Ele também lida com alterações de estado da janela e fornece funções para exibir mensagens de alerta e centralizar a janela na tela.

manual_input_screen.py

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

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

QTimer: Uma classe do PyQt6 usada para lidar com a contagem de tempo.

QWidget, QPushButton, QVBoxLayout, QLabel, QHBoxLayout, QLineEdit, QScrollArea, QComboBox, QGridLayout: Classes PyQt6 para criação de elementos GUI, similar a explicação de main_window.py, as únicas 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 e o GridLayout, que utiliza um sistema de coordenadas para posicionar elementos.

instruction: Classe personalizada para armazenar as informações de cada instrução de uma forma mais organizada, como se fosse um struct de outras linguagens.

instruction_list_processor, **run_instructions**: Arquivo com a lógica da execução do ciclo de instruções, ele é importado pois também é reutilizado na entrada por arquivo.

- **Criando o widget da tela de entrada manual(classe ManualInputScreen):**

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

Iniciamos a classe passando algumas funções da classe principal como argumento na forma de “callbacks” para que a mesma função seja invocada. Essas funções são para voltar a tela de boas vindas, criar uma mensagem de alerta e centralizar a aplicação na tela.

Definimos uma lista vazia de instruções.

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

- **Tela de visualização, adição e remoção de instruções:**

Criamos um layout como contêiner desta tela, inicialmente invisível.

Criamos o cabeçalho desta dela com um QLabel e adicionando espaçamento.

Criamos uma ScrollArea para mostrar a lista de instruções, isso depende muito da resolução da tela do usuário mas em algum momento se existem muitas instruções será criada uma barra de rolamento.

Criamos um layout Grid inicialmente vazio.

Criamos um layout horizontal abaixo da Grid e ScrollArea para manipular a entrada do usuário composto de uma Combo Box com todos os códigos de instrução, dois QLineEdits para receber os valores dos operandos e um botão de confirmação.

Criamos um botão de voltar abaixo de todos esses elementos.

Definimos a hierarquia entre todos esses layouts com o contêiner no topo.

- **Tela inicial de entrada manual:**

Criamos um layout e contêiner para a tela inicial da entrada manual.

Criamos o cabeçalho e os 3 botões com as opções do usuário de ver a tela de visualização, adição e remoção definida anteriormente, a tela de resultado da execução das instruções, e voltar para a tela de boas vindas.

Definimos a hierarquia entre todos esses layouts com o contêiner no topo.

- **Tela inicial de entrada manual:**

Criamos um layout e contêiner para a tela de resultado da execução das instruções, inicialmente invisível.

Criamos uma ScrollArea para caso o conteúdo seja muito extenso uma barra de rolagem seja criada.

Criamos os 3 headers para as seções de log, valor do MBR, instruções realizadas e a fita dos endereços de memória e seus valores.

Criamos um botão de voltar abaixo.

Definimos a hierarquia entre todos esses layouts com o contêiner no topo.

- **Bloco if not self.isMaximized:**

Centralizamos a aplicação caso esteja minimizada para melhor visualização.

- **Funções dessa classe de entrada manual:**

Definimos uma função show_menu_view para voltar à tela inicial desta classe de entrada manual manipulando a visibilidade dos grandes contêineres da classe.

Definimos uma função show_add_instructions_view para mostrar a tela de visualização, adição e remoção de instruções. Esta função realiza a verificação da sequência atual de instruções e sempre inicializa a Combo Box na primeira instrução.

Definimos uma função auxiliar update_add_instructions_view que utiliza o sistema de Grid Layout para popular a Grid vazia criada quando a classe foi inicializada. O Grid é útil aqui para estruturar as instruções em uma forma de tabela, com os headers indicando a linha da instrução, seu código, operandos se existirem, além de um botão que possibilita a remoção desta instrução específica.

Definimos uma função on_main_combobox_changed conectada a Combo Box que permite o usuário selecionar uma instrução para adicionar à sequência. Esta função altera dinamicamente quais caixas de entrada ficam ativas ou não baseadas nos operandos de cada instrução.

Definimos uma função insert_new_instruction que cuida da adição de uma instrução para a sequência, verificando se a entrada foi um valor válido ou não. Apenas valores inteiros são operandos válidos, com a exceção do código “000010” que permite float em seu segundo operando.

Definimos uma função remove_instruction que utiliza o índice armazenado no botão de remoção para deletar o índice certo da instrução clicada pelo usuário.

Definimos uma função create_delete_button_callback que dinamicamente cria funções únicas de remoção para cada botão de remoção, desta forma o seu índice sempre corresponderá ao item correto.

Finalmente, definimos uma função run_instructions_function que utiliza a array de instruções armazenada na classe para executar a sequência de instruções atual. Esta função está ligada a tela de resultado, recebe os logs e os mostra para o usuário saber o que aconteceu. Esta função utiliza a lógica importada da função run_instructions do arquivo instruction_list_processor.py.

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

Cria um widget que pode ser instanciado e adicionado para o layout em pilha da classe principal. Este widget contém uma lógica similar ao layout em pilha com a visibilidade de sub layouts que direcionam o usuário para as funções de inserção ou execução de uma sequência de instruções. Esta classe tem o objetivo de proporcionar uma boa experiência de entrada manual de dados ao usuário.

file_input_screen.py

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

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

QTimer: Uma classe do PyQt6 usada para lidar com a contagem de tempo.

QWidget, QPushButton, QVBoxLayout, QLabel, QTextEdit, QScrollArea, QFileDialog: Classes PyQt6 para criação de elementos GUI,

similar a explicação de main_window.py e manual_input_screen.py, as únicas novidades aqui são o QTextEdit que é basicamente um QLineEdit só que é uma caixa de texto ao invés de apenas uma linha de entrada e o QFileDialog que permite o usuário explorar a estrutura de pastas de seu computador para abrir um arquivo de texto com instruções pré-definidas.

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

instruction: Classe personalizada para armazenar as informações de cada instrução de uma forma mais organizada, como se fosse um struct de outras linguagens.

instruction_list_processor, **run_instructions**: Arquivo com a lógica da execução do ciclo de instruções, ele é importado pois também é reutilizado na entrada manual.

- **Criando o widget da tela de entrada por arquivo(classe FileInputScreen):**

Estrutura extremamente similar a tela de resultado da classe de entrada manual, porém com a caixa de texto no começo do layout vertical para que o usuário insira várias linhas de texto rapidamente, agilizando a velocidade de teste de instruções diferentes.

Além disso, temos o botão de navegação da estrutura de pastas e arquivos do computador do usuário.

- **Funções dessa classe de entrada por arquivo:**

Definimos uma função open_file_button que utiliza um FileDialog para mostrar a tela de explorar os arquivos do computador do usuário com o filtro para arquivos do formato .txt. O conteúdo deste arquivo será escrito na caixa TextEdit da interface e pode ser editada pelo usuário a qualquer momento.

Definimos uma função get_data que recebo o texto da QTextEdit para variáveis mais “amigáveis” ao processamento de dados, dividindo o conteúdo em uma array de strings, onde cada índice corresponde a uma linha de texto.

Definimos uma função parse_instructions para verificar se as instruções foram inseridas com a formatação correta, "[CODE],[VALUE_A],[VALUE_B]" em uma linha separada, dividindo cada valor com uma vírgula, com a mesma

verificação de número, inteiro ou float da entrada manual. Caso nenhum problema seja encontrado, as instruções são executadas com a lógica importada da função run_instructions do arquivo instruction_list_processor.py e o log resultante é mostrado ao usuário.

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

Cria um widget que pode ser instanciado e adicionado para o layout em pilha da classe principal. Este widget é bem similar ao manual_input_screen.py porém com algumas nuances para lidar com a entrada de um grande string de texto ao invés de uma entrada mais controlada como a manual. Nota-se que ambas as classes utilizam a mesma lógica importada de outro arquivo.

instruction.py

- **Criando a classe Instruction**:

Esta classe é extremamente simples, quase como um struct de outras linguagens, definindo alguns valores a serem armazenados para uma conveniência um pouco maior do desenvolvimento da aplicação.

Em síntese, o código da instrução é sempre armazenado e os valores a e b são opcionais, tendo em vista que dependem do próprio código de instrução. Desta forma a lógica para a adição de uma instrução cuida desta verificação e inserção correta na array de sequência de instruções.

instruction_list_processor.py

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

math com a função sqrt: Realiza a operação de raiz quadrada da instrução “001010”.

- **Criando a função run_instructions**:

Esta função é a espinha dorsal de toda a aplicação e contém a lógica que executa a sequência de instruções.

De início, ela verifica qual é a maior posição de memória para retornar uma fita de endereços de memória mais “verídica”.

Todas as variáveis auxiliares são inicializadas em zero, vazias ou falsas.

Um loop while é utilizado para executar toda a sequência de instruções devido a possibilidade de existir JUMPs que modificam a ordem de execução.

Cada iteração cai em um switch case, ou match case em Python que verifica o código da instrução e executa as operações apropriadas, bem como armazena o que aconteceu no log.

Como uma estrutura de array é utilizada, existe a possibilidade de um JUMP ser executado para um índice fora de seu escopo, desta forma a booleana out_of_bounds_error serve como um gatilho para interromper a execução e informar o usuário caso isso aconteça.

Por fim, o MBR, a fita e o log são retornados pela função, dados que são utilizados nas classes ManualInputScreen e FileInputScreen para serem mostrados na tela da aplicação.