

Relatório Técnico: Emulador SAP-1 com Interface Gráfica Moderna

Grupo:

- Rafael Rehfeld Martins de Oliveira
- Filipe Nery Rabelo
- João Vitor de Alvarenga Alvares
- Henrique Gonçalves Pimenta Velloso
- Enzo Moraes Martini

Disciplina: Arquitetura de Computadores I

Professor: Claudio

1. Introdução

Este relatório apresenta uma versão aprimorada do emulador SAP-1, desenvolvido como ferramenta didática para o estudo da arquitetura básica de computadores. A nova versão mantém os princípios fundamentais do projeto original descrito no relatório anterior, porém com significativas melhorias na interface gráfica, usabilidade e organização do código. O emulador continua baseado na arquitetura SAP-1 (Simple-As-Possible 1) conforme descrito no livro "Digital Computer Electronics" de Albert Paul Malvino (Capítulo 10), mas agora apresenta uma implementação mais modular e um design visual modernizado.

2. Arquitetura do Sistema

2.1. Componentes Principais

O sistema foi redesenhado em duas classes principais que separam claramente a lógica de emulação da interface gráfica:

1. NucleoSAP1: Classe que encapsula toda a lógica de emulação da CPU SAP-1
2. AplicativoSimulador: Classe responsável pela interface gráfica e interação com o usuário

2.2. Melhorias na Implementação

As principais melhorias em relação à versão anterior incluem:

- Separação mais clara entre lógica de negócios e interface gráfica
- Design visual modernizado com esquema de cores mais atraente
- Sistema de animações refinado para melhor visualização do fluxo de dados
- Aprimoramentos na entrada de expressões matemáticas
- Código mais organizado e documentado

3. Implementação do Software

3.1. Módulo da CPU (Classe NucleoSAP1)

A classe NucleoSAP1 implementa fielmente a arquitetura SAP-1 com os seguintes componentes:

```
class NucleoSAP1:  
    def __init__(self):  
        self.registros = {  
            "ContadorPrograma": 0,      # PC - 4 bits  
            "Acumulador": 0,          # ACC - 8 bits  
            "RegistradorEndereco": 0, # REM/MAR - 4 bits  
            "RegistradorInstrucao": 0, # RI/IR - 8 bits  
            "RegistradorB": 0,        # Reg B - 8 bits  
            "RegistradorSaida": 0,    # Output - 8 bits  
            "Flags": {"Zero": 0, "Carry": 0}  
        }  
        self.memoria_principal = [0] * CAPACIDADE_MEMORIA # RAM 16x8
```

O conjunto de instruções foi implementado como um dicionário que mapeia opcodes para suas funções de execução:

```
self.conjunto_instrucoes = {  
    0b0000: ("CAR", self._executar_car), # Load Accumulator  
    0b0001: ("SOM", self._executar_som), # Add  
    0b0010: ("SUB", self._executar_sub), # Subtract  
    0b1110: ("SAI", self._executar_sai), # Output  
    0b1111: ("PAR", self._executar_par) # Halt  
}
```

3.2. Interface Gráfica (Classe AplicativoSimulador)

A interface gráfica foi completamente redesenhada com:

- Tema moderno usando o estilo 'clam' do Tkinter
- Esquema de cores roxo e preto para melhor contraste visual
- Sombreado nos componentes para efeito de profundidade
- Organização espacial mais lógica dos componentes da CPU

3.2.1. Componentes da Interface

```
def _construir_interface(self):  
    # Painel esquerdo com controles  
    painel_esquerdo = ttk.Frame(container_principal, width=350)  
  
    # Seção de entrada de expressão  
    secao_expressao = ttk.LabelFrame(painel_esquerdo, text="Entrada de Expressão")
```

```

# Editor de código Assembly
secao_editor = ttk.LabelFrame(painel_esquerdo, text="Editor de Código")

# Painel de visualização da CPU
painel_cpu = ttk.LabelFrame(container_principal, text="Visualização da CPU")

```

3.2.2. Representação Visual da CPU

O método `_desenhar_cpu` cria uma representação gráfica fiel da arquitetura SAP-1:

```

def _desenhar_cpu(self):
    # Barramento principal
    self.canvas_cpu.create_line(50, 400, 800, 400, width=3, fill="#8e44ad",
                                tags="barramento")

    # Componentes (PC, REM, RI, ACC, Reg B, ULA, RAM, Saída)
    criar_componente(100, 50, 100, 70, "CP", "bloco_cp", "0x0")
    criar_componente(250, 50, 100, 70, "REM", "bloco_rem", "0x0")
    criar_componente(450, 50, 100, 70, "RI", "bloco_ri", "0x0")
    # ... outros componentes ...

    # Memória RAM (16 bytes)
    for i in range(CAPACIDADE_MEMORIA):
        self.canvas_cpu.create_rectangle(...) # Célula de memória
        self.canvas_cpu.create_text(...)      # Valor da memória
        self.canvas_cpu.create_text(...)      # Endereço

    # LEDs de saída
    for i in range(8):
        self.canvas_cpu.create_oval(...)     # LED
        self.canvas_cpu.create_text(...)    # Rótulo do bit

```

3.3. Sistema de Animação

O emulador possui três tipos principais de animações:

1. Transferência via barramento: Mostra o fluxo de dados entre componentes
2. Transferência direta: Para operações que não usam o barramento principal
3. Destaque de componentes: Indica qual componente está ativo

```

def _animar_transferencia(self, origem, destino, duracao=0.3):
    # 1. Destacar componente de origem e conexão
    # 2. Destacar barramento
    # 3. Destacar componente de destino e conexão
    # 4. Retornar ao estado normal

```

3.4. Montador de Código Assembly

O método `_montar_codigo` implementa um montador completo que suporta:

- Instruções básicas (CAR, SOM, SUB, SAI, PAR)

- Diretivas ORG e DB
- Comentários (linhas iniciando com ;)
- Verificação de erros sintáticos e semânticos

```
def _montar_codigo(self):
    # Processa cada linha do código Assembly
    for num_linha, linha in enumerate(linhas, 1):
        # Remove comentários
        linha_processada = linha.split(';')[0].strip()

        # Processa diretivas ORG/DB
        if mnemonic == "ORG":
            # Define posição atual na memória
        elif mnemonic == "DB":
            # Armazena dados na memória

        # Processa instruções
        opcode = {
            "CAR": 0b0000,
            "SOM": 0b0001,
            # ... outras instruções
        }.get(mnemonic)

        # Combina opcode e operando
        memoria[ponteiro_instrucao] = (opcode << 4) | operando
```

3.5. Ciclo Fetch-Execute

O método `_executar_passo` implementa o ciclo de busca-execução em 6 estados (T1-T6):

```
def _executar_passo(self):
    # 1. CICLO DE BUSCA
    # T1: PC -> MAR
    self._animar_transferencia("bloco_cp", "bloco_rem")

    # T2: Incrementar PC
    self._destacar_componente("bloco_cp")

    # T3: Mem[MAR] -> RI
    self._animar_transferencia("ram", "bloco_ri")

    # 2. CICLO DE EXECUÇÃO
    opcode = self.nucleo.registradores['RegistradorInstrucao'] >> 4
    operando = self.nucleo.registradores['RegistradorInstrucao'] & 0x0F

    # Executa a instrução apropriada
    if opcode in self.nucleo.conjunto_instrucoes:
        nome_instrucao, funcao = self.nucleo.conjunto_instrucoes[opcode]
```

```
parar = funcao(operando)
```

4. Novos Recursos e Melhorias

4.1. Entrada de Expressões Matemáticas

O sistema agora inclui um teclado virtual para entrada de expressões simples

```
def _processar_expressao(self):
    expressao = self.valor_campo_expressao.get()
    componentes = re.findall(r'(\d+)|([+-])', expressao)

    # Gera código Assembly para a expressão
    codigo = "; Código gerado para: " + expressao + "\n"
    codigo += f"CAR {inicio_dados}:01X} ; Carrega o primeiro número\n"

    for i, op in enumerate(operadores):
        if op == "+":
            codigo += f"SOM {inicio_dados+i+1}:01X} ; Soma\n"
        elif op == "-":
            codigo += f"SUB {inicio_dados+i+1}:01X} ; Subtrai\n"

    # ... código completo ...
```

4.2. Melhorias Visuais

- Tema moderno: Uso do tema 'clam' do Tkinter com cores personalizadas
- Sombreamento: Efeito 3D nos componentes da CPU
- LEDs de saída: Representação visual dos bits no registrador de saída
- Barramento destacado: Linha grossa que conecta todos os componentes

4.3. Aprimoramentos no Código

- Separação de responsabilidades: Lógica da CPU separada da interface
- Métodos mais curtos e especializados: Melhor legibilidade e manutenção
- Tratamento de erros aprimorado: Mensagens mais claras para o usuário
- Documentação interna: Comentários explicativos em todos os métodos principais

5. Como Utilizar o Emulador

1. Pré-requisitos:
 - Python 3.x instalado
 - Bibliotecas: tkinter, re, time
2. Execução:
colocar no terminal do VS code: python Trabalho_sap1.py
3. Funcionalidades:
 - Digitar expressões matemáticas ou código Assembly diretamente
 - Montar o programa (compilar para linguagem de máquina)

- Executar o programa (contínuo ou passo a passo)
 - Ajustar velocidade de execução
 - Reiniciar o simulador
4. Controles:
- Montar Programa: Converte Assembly para código de máquina
 - Executar: Roda o programa continuamente
 - Passo a Passo: Executa uma instrução por vez
 - Reiniciar: Reseta todos os registradores e memória

6. Conclusão

Esta nova versão do emulador SAP-1 representa um avanço significativo em termos de organização de código, qualidade da interface gráfica e experiência do usuário. A separação clara entre a lógica de emulação e a interface gráfica torna o código mais sustentável e facilita futuras expansões.

As melhorias visuais e o sistema de animações tornam o processo de aprendizado mais intuitivo e engajador, permitindo que estudantes visualizem claramente o fluxo de dados entre os componentes da CPU durante a execução de cada instrução.

O emulador cumpre com excelência seu papel como ferramenta didática para o ensino de arquitetura de computadores, proporcionando uma ponte clara entre os conceitos teóricos e sua implementação prática.