Criação de uma ISA (Instruction Set of Architecture)

Andrei Miguel Cristeli
Yann Eduardo de Souza SIlva

Resumo: O programa oferece uma ferramenta básica para a simulação de um emulador, que contém 15 instruções, cada uma com aproximadamente 22 bits. Além disso, o emulador possui uma memória de 16MB. Ele permite a entrada por meio de um arquivo, no qual lê e processa as operações necessárias, exibindo o resultado na instrução DSP.

I. INTRODUÇÃO

A Arquitetura de Conjuntos de Instruções (ISA), também conhecida como Instruction Set Architecture em inglês, constitui a interface visível ao programador em um processador. Essa arquitetura representa a fronteira entre o hardware e o software de baixo nível, sendo crucial para a interação entre ambos. A descrição da ISA de um processador geralmente se baseia em cinco categorias:

- **Armazenamento dos Operandos no CPU:**
  Onde os operandos são armazenados além da memória?
- **Número de Operando Chamados por Instrução:**
  Quantos operando são requeridos em uma instrução típica?
- **Localização dos Operando:**
  Pode algum operando da Unidade Lógica e Aritmética (ALU) estar na memória, ou todos devem residir na memória interna do CPU?
- **Operações:**
  Quais operações são disponibilizadas pela ISA?
- **Tipo e tamanho dos operandos**
  Qual é o tipo e tamanho de cada operando, e como essas características são especificadas?

Dentre esses fatores, a localização do armazenamento dos operandos no CPU é frequentemente a mais crucial. A ISA, no entanto, estabelece padrões de bits para cada instrução, limitando o software a essas instruções. Introduzir inovações torna-se complicado, muitas vezes exigindo alterações no hardware, um processo custoso.

Para evitar custosas alterações, a ISA deve oferecer um conjunto abrangente de instruções que satisfaça diversas necessidades. O programa em questão recebe como entrada um arquivo que contém as operações necessárias para executar determinados cálculos, retornando a saída desejada. Neste trabalho prático, o objetivo era desenvolver um programa que gerasse como saída os números primos, assim como o seno e cosseno de um número em radianos, conforme especificado pelo usuário.

II. REFERENCIAL TEÓRICO

Para criarmos o arquivo, o qual o programa irá interpretar, usamos conjuntos de instruções sendo eles:
• Load-Store: STI STO;
• Operações: ADD SUB MUL DIV POT FAT;
• Comparação: BLT BEQ.

Como referência, adotamos a arquitetura CISC (Complex Instruction Set Computer), a qual é caracterizada por empregar um conjunto de instruções complexas, em quantidade elevada e altamente especializadas. Arquiteturas CISC, como a x86, são projetadas para realizar mais operações com menos linhas de código de baixo nível. Em outras palavras, enquanto uma arquitetura RISC (Reduced Instruction Set Computer) pode demandar várias instruções para executar uma tarefa, uma arquitetura CISC pode realizar o mesmo trabalho com uma única instrução.

Devido à complexidade das instruções em arquiteturas CISC, elas podem executar diversas operações de baixo nível, como acessar a memória, realizar cálculos e processar lógica, em uma única instrução. Entretanto, é importante observar que uma instrução CISC pode demandar vários ciclos de clock para ser concluída.

Em termos gerais, as arquiteturas CISC geralmente resultam em processadores com maior desempenho bruto, porém, isso ocorre à custa de um maior consumo de energia.

III. METODOLOGIA

O código é projetado para simular uma ISA (Instruction Set Architecture). Vamos explicar o funcionamento de cada função do código e o passo a passo do processo. Para isso, vamos dividi-los em três partes: declaração de estruturas (`struct`), as funções e a função `main`.

1. Declaração de estrutura:

Utilizamos duas structs, uma representa a Estrutura de Instruções e a outra representa a Estrutura do Processador. Vamos começar explicando a estrutura de instruções:

a. **Estrutura** `Instrucoes`**:**
   - **Objetivo:** Armazenar informações relacionadas às instruções do programa.
   - **Variáveis:**
     - `char opcode[6];`: Armazena o código de operação da instrução (máximo de 6 caracteres).
     - `double operacao1;`: Armazena o valor associado à operação (número real).

- `int registradorDestino;`: Armazena o número do registrador de destino.
- `int registradorDestino2;`: Armazena o número do segundo registrador de destino (quando aplicável).

b. **Estrutura** `Processador`:
   - **Objetivo:** Representar o estado do processador durante a execução do programa.
   - **Variáveis:**
     - `double registradores[MAX_REGISTRADORES];`: Array para armazenar os valores nos registradores (com um limite de `MAX_REGISTRADORES` registradores).
     - `double memoria[MAX_MEMORY_SIZE];`: Array para armazenar valores na memória (com um limite de 16MB).
     - `int cont;`: Contador para rastrear o número de elementos na memória.

c. **Explicação:**
   - As estruturas `Instrucoes` e `Processador` são utilizadas para organizar e manter informações relacionadas a instruções e ao estado do processador, respectivamente.
   - `MAX_REGISTRADORES` ,`MAX_INTR_SIZE` e `MAX_MEMORY_SIZE` são macros definidas para limitar o número máximo de registradores , o tamanho máximo das instruções e o tamanho da memória, fornecendo assim uma base para a alocação de arrays.
   - As estruturas proporcionam uma forma organizada de agrupar dados relacionados, facilitando a manipulação e passagem desses dados entre as funções do programa.
   - O uso de arrays nas estruturas permite armazenar múltiplos valores, como registradores e memória, proporcionando flexibilidade na implementação das operações do processador.
   - As variáveis adicionais, como contadores e acumulador, são utilizadas para rastrear o progresso do programa durante a execução.

2. Funções:

a. alu(): Função responsável por realizar operações aritméticas e lógicas com base na instrução fornecida. Recebe como argumentos uma linha de instrução, uma estrutura de instrução (`Instrucoes`), um ponteiro para o processador (`Processador`), e um ponteiro para um acumulador (`double`).

b. sti(): Função que implementa a instrução `STI` (Store Immediate). Recebe uma linha de instrução, uma estrutura de instrução (`Instrucoes`), e um ponteiro para o processador.

c. sto(): Função que implementa a instrução STO (Store). Recebe uma linha de instrução, uma estrutura de instrução (`Instrucoes`), um ponteiro para o processador, e um ponteiro para um acumulador (`double`).
d. blt(): Função que implementa a instrução `BLT` (Branch if Less Than). Recebe uma linha de instrução, uma estrutura de instrução (`Instrucoes`), um ponteiro para o processador, e um ponteiro para um inteiro (`int`) que armazenará o destino do salto.
e. beq(): Função que implementa a instrução `BEQ` (Branch if Equal). Recebe uma linha de instrução, uma estrutura de instrução (`Instrucoes`), um ponteiro para o processador, e um ponteiro para um inteiro (`int`) que armazenará o destino do salto.
f. jump(): Função que realiza um salto para uma linha específica. Recebe um ponteiro para o contador de linhas (`contlinha`) e o número da linha de destino.

3. **Função Principal (main()):**
   a. **Declaração de Variáveis:**
      - **`Instrucoes inst;`:** Declaração de uma variável do tipo `Instrucoes` para armazenar a instrução atual.
      - **`Processador reg = {0};`:** Declaração e inicialização de uma variável do tipo `Processador` chamada `reg` com todos os registradores zerados.
      - **`int contlinha = 0, contlinha2 = 0;`:** Declaração de variáveis de controle para as linhas do arquivo.
      - **`int linhadestino, result;`:** Declaração de variáveis para armazenar o destino do salto e o resultado de comparações.
      - **`char linha[MAX_INTR_SIZE];`:** Declaração de um array de caracteres para armazenar cada linha do arquivo.
      - **`double acc;`:** Declaração de uma variável para armazenar um acumulador.
   b. **Leitura e Alocação de Memória:** O programa começa alocando memória dinamicamente para armazenar as instruções lidas de um arquivo. O usuário é solicitado a inserir o nome do arquivo a ser lido.
   c. **Leitura do Arquivo e Impressão das Instruções:** O código lê o arquivo, armazena as instruções no array `vetDeString`, e imprime as instruções lidas.
   d. **Execução das Instruções:** Utilizando um loop while, o programa interpreta e executa as instruções presentes no array `vetDeString`. Dependendo do código da operação, a execução é redirecionada para as funções correspondentes.
   e. **Saída e Liberação de Memória:** Ao final da execução, o programa imprime a quantidade de bits de memória consumidos e os números armazenados na memória, além de liberar a memória alocada dinamicamente.

IV. RESULTADO

O resultado dos números primos foi o esperado, porém sempre será o mesmo, logo, não foi possível simular algumas entradas diferentes, algo que foi possível no seno e cosseno. A memória consumida ao rodar o código para poder armazenar os números primos foram de 100 bits, uma vez que, os números de 1 a 100 totalizam em 25 números que são primos, e como eles são inteiros, 25 * 4 = 100 bits. Ao fazermos testes com várias entradas (Números em Radianos), vimos que era sempre bem aproximado do real valor, então fomos atrás do motivo de porque isso acontecia e vimos que era devido ao número de interações que o programa executava a fórmula de Taylor. Trocamos o número de interações e vimos que quanto mais se aproximava do infinito, melhor era a precisão e aproximação do real resultado, logo abaixo está a diferença com cada número de interações:

Seno de 1.57 (90º)

```
Digite o número que deseja saber o sen:
seno: 0.9250178333
```

(Print do resultado do código com 1 interação)

```
Digite o número que deseja saber o sen:
seno: 1.0045086605
```

(Print do resultado do código com 2 interações)

```
Digite o número que deseja saber o sen:
seno: 0.9998434952
```

(Print do resultado do código com 3 interações)

```
Digite o número que deseja saber o sen:
seno: 1.0000032059
```

(Print do resultado do código com 4 interações)

```
Digite o número que deseja saber o sen:
seno: 0.9999996270
```

(Print do resultado do código com 5 interações)

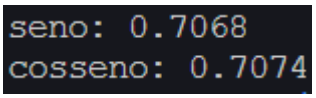
E cada vez mais se aproxima de 1, resultado esperado. Ocorre o mesmo com o cosseno, cada interação a mais é uma aproximação melhor. Algumas saídas com o seno e cosseno de outros ângulos:

Seno e Cosseno de 0.523 (30°)

```
seno: 0.4995
cosseno: 0.8663
```

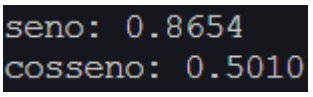
(Print do resultado do código com 5 interações)

Seno e Cosseno de 0.785 (45°)



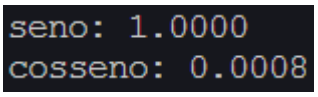
(Print do resultado do código com 5 interações)

Seno e Cosseno de 1.046 (60°)



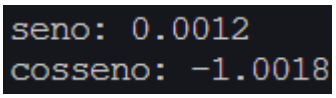
(Print do resultado do código com 5 interações)

Seno e Cosseno de 1.57 (90°)



(Print do resultado do código com 5 interações)

Seno e Cosseno de 3.14 (180°)



(Print do resultado do código com 5 interações)

V. CONCLUSÕES

Chegamos no resultado esperado e conseguimos concluir o trabalho com a ajuda de alguns artigos sobre ISA, o que nos ajudou a planejar melhor e conseguir executar esse trabalho pratico conseguindo alcançar uma boa aproximação do seno e cosseno usando as séries de Taylor e MacLaurin. No final do trabalho consolidamos melhor a teoria e a prática do que é uma ISA e como funciona.

Fontes usadas:

https://tecnoblog.net/responde/qual-e-a-diferenca-entre-arquitetura-risc-e-cisc-processador/#:~:text=CISC%20(Complex%20Instruction%20Set%20Computer)%20é%20um%20tipo%20de%20arquitetura,de%20código%20de%20baixo%20nível.