Universidade Federal de Ouro Preto - UFOP
Instituto de Ciências Exatas e Biológicas - ICEB
Departamento de Computação - DECOM
Ciência da Computação

Trabalho Prático II

BCC266 - Organização de Computadores

Vitor de Oliveira Silva
Gabriel Carlos Silva

Professor: Mateus Coelho Silva

Ouro Preto
15 de janeiro de 2024

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Códigos Fonte

1 Introdução

Em Organização de Computadores, estuda-se a ISA (Instruction Set Architecture), que são as instruções escritas pelo programador que são lidas e implementadas pelo hardware. Com ela, pode-se fazer operações aritméticas básicas, manipular os valores nos registradores e criar condições e laços de repetição.

Neste trabalho, será emulado um processador baseado na arquitetura de Von Neumann com uma ISA para encontrar todos os números primos entre 1 e 100, além de achar o seno e cosseno de um valor qualquer em radianos.

1.1 Especificações do Projeto

Para a criação desse trabalho, terá algumas regras e obrigações a serem seguidas, sendo essas:

- O programa deverá ter no máximo 16 instruções, sendo elas com no máximo 32 bits.
- O programa poderá ter as operações aritméticas básicas, como soma, subtração, divisão, multiplicação e shifts
- O programa só pode ter no máximo 16 registradores
- O programa deverá ter memória principal (de até 16 MBs) e banco de registradores.
- Programa deve ser carregado em uma unidade de memória especial (memória de programa/-memória de instruções).
- O programa deve ter pelo menos um dispositivo de saída para visualizar o resultado ao final da execução
- Cada unidade ALU, Memória Principal, Memória de Instruções, Banco de Registradores, etc., deve ter uma estrutura de dados dedicada para ela.
- O programa deve ser carregado a partir de um arquivo de texto.

1.2 Ferramentas utilizadas

Algumas ferramentas foram utilizadas para testar a implementação, como:

- Linguagem C
- *Visual Studio Code*

1.3 Especificações da máquina

Para os testes da implementação, serão utilizados três computadores diferentes:

- Computador 1
  - Processador : Intel® Core™ i3-6006U CPU @ 2.00GHz × 4
  - Memória RAM : 8GBs
  - Sistema Operacional: Ubuntu 22.04.3 LTS
- Computador 2
  - Processador AMD Ryzen 5-5500U
  - Memória RAM: 8GBs
  - Sistema Operacional: Windows 10 22H2 (Utilizando WSL)
- Computador 3
  - Processador AMD® Ryzen 3 3200g
  - Memória RAM: 16GBs
  - Sistema Operacional: Ubuntu 22.04.3 LTS

2 Arquitetura de Von Neumann

Este trabalho será montado exclusivamente pensando na Arquitetura de Von Neumann. A Arquitetura de computadores definida por John Von Neumann (1903-1957) é a que é utilizada nos computadores modernos. Em resumo, ela define um computador que seus dados armazenados na memória e os dados de programa sejam uma coisa só, com o mesmo tipo de linguagem e, consequentemente, o mesmo tipo de dado. A arquitetura é dividida em duas partes principais:

- A CPU, que possui internamente a Unidade de Controle e a Unidade de Lógica e Aritmética (ALU)
- A memória, que armazena os dados;

A CPU e a memória conversam entre si, mandando as instruções que o computador irá fazer e as executando, recebendo valores de entrada, normalmente por um teclado, e mandando os valores de saída, normalmente por um monitor.

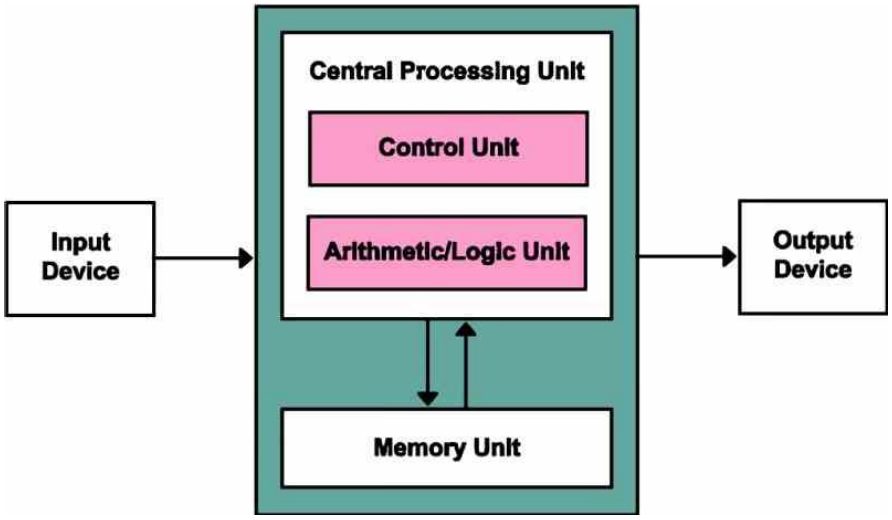

Figura 1: Arquitetura de Von Neumann

A Arquitetura de Von Neumann será mais explorada durante o trabalho, então, nessa parte inicial, será apenas uma breve introdução sobre ela e que ela será a base de todo o trabalho montado a partir deste ponto.

3 Metodologia

O programa terá duas partes: a primeira é montando a ISA, explorando algumas formas de resolver os problemas usando as instruções propostas e a segunda é montando o hardware em si, divindo as partes da CPU e a memória e como elas conversarão entre si.

3.1 Implementação da ISA

Todas as instruções tiveram uma padronização de bits utilizados. A regras das instruções são:

- Como temos no máximo 16 instruções, foram armazenados 4 bits (valores de 0 a 15) para os opcodes
- Como tem-se o máximo de 16 registradores, foram armazenados 4 bits (valores de 0 a 15) para os registradores
- Todo o resto dos bits (depende de cada instrução) são utilizadas para receber valores inteiros ou com ponto flutuante como valores para os registradores ou valor de uma determinada linha do código

Com todas essas informações, podemos montar nossas instruções, descritas na tabela abaixo:

Opcode	Binário	Operandos	Função	Exemplo
SET	0000	Endereço, Valor	Colocar um valor do tipo float em um endereço	SET $6 35.8
HOP	0001	Valor	Alterar o Contador de Programa e pular para uma determinada linha do código	HOP 10
MEMF	0010	Endereço	Mostrar na saída um valor em float	MEMF $6
MEMD	0011	Endereço	Mostrar na saída um valor em inteiro	MEMD $6
SUM	0100	Endereço, Endereço, Endereço	Soma os dois registradores recebidos, armazena o resultado no último registrador.	SUM $5 $1 $12
SUB	0101	Endereço, Endereço, Endereço	Subtrai os dois registradores recebidos, armazena o resultado no último registrador.	SUB $5 $1 $12
MUL	0110	Endereço, Endereço, Endereço	Multiplica os dois registradores recebidos, armazena o resultado no último registrador.	MUL $5 $1 $12
DIV	0111	Endereço, Endereço, Endereço	Divide os dois registradores recebidos, armazena o resultado no último registrador.	DIV $5 $1 $12
MOD	1000	Endereço, Endereço, Endereço	Calcula o resto da divisão dos dois registradores recebidos, armazena o resultado no último registrador.	MOD $5 $1 $12
EXP	1001	Endereço, Endereço, Endereço	Eleva o primeiro registrador ao segundo, armazena o resultado no último registrador.	EXP $5 $1 $12
IET	1010	Endereço, Endereço, Valor	Verifica se os dois registradores possuem o mesmo valor, caso positivo, pula para a linha recebida.	IET $10 $0 12
ILT	1011	Endereço, Endereço, Valor	Verifica se o primeiro registrador é menor que o segundo registrador, caso positivo, pula para a linha recebida.	ILT $10 $0 12
CPY	1100	Endereço, Endereço	Copia o valor do segundo registrador para o primeiro registrador.	CPY $10 $8

Tabela 1: ISA montada para o projeto

3.1.1 Montando o código dos números primos

Um número primo é um número em que só é divisível por 1 e ele mesmo. Um número só é divísel por outro caso o resto de divisão entre eles for igual a zero. Pensando nisso, conclui-se duas coisas:

- Se contarmos a quantidade de divisores que tem entre 1 e o número em questão, conseguimos definir se ele é primo ou não;
- Melhor ainda, para facilitar na conta, deve-se olhar apenas para os números entre 2 e o número em questão - 1. Caso a quantidade de divisores for igual a zero, pode-se afirmar que o número é primo.

Com isso, tem-se o seguinte código:

primo.txt

```
SET $3 2
MEMD $3
SET $2 3
SET $5 100
MOD $2 $3 $4
IET $4 $0 10
SUM $3 $1 $3
ILT $3 $2 5
MEMD $2
SUM $2 $1 $2
SET $3 2
ILT $2 $5 5
```

Nos próximos tópicos, será executado esse código com os outros próximos. Por enquanto, só será necessário saber que esse programa foi nomeado de "primo.txt", que será jogado no terminal para ser executado.

3.1.2 Montando o código do Seno e Cosseno

Para encontrar os valores de seno e cosseno, será utilizado as Séries de Maclaurin. As Séries de Maclaurin são uma forma de representar uma função de classe C^∞, ou seja, que possuem derivadas de todas as ordens e que todas elas são contínuas, centradas em 0. A fórmula da série de MacLaurin é descrita como:

$$\sum_{n=0}^{\infty} \frac{f^{(n)}(0)}{x!} x^n$$

Seja f(x) = sen(x). Observe o comportamento das derivadas da função em f(0):

$$f(0) = \sin(0) = 0$$

$$f'(0) = \cos(0) = 1$$

$$f''(0) = -\sin(0) = 0$$

$$f'''(0) = -\cos(0) = -1$$

$$f^{iv}(0) = \sin(0) = 0$$

...

e assim por diante. A partir disso, o ciclo se repete, tendo, dessa maneira, a seguinte soma:

$$\sum_{n=0}^{\infty} \frac{sin^{(n)}(0)}{x!} x^n = \frac{0 * x^0}{0!} + \frac{1 * x^1}{1!} + \frac{0 * x^2}{2!} + \frac{-1 * x^3}{3!} + ...$$

Perceba que quando n é par, o valor é nulo. Além disso, quando o valor é ímpar, ele alterna entre positivo e negativo. Com isso, podemos definir que apenas os valores elevados a 2n+1 serão utilizados, sendo que, quando n for par, o número da soma será positivo e, quando for ímpar, será negativo. Com isso, temos a representação do seno em uma série de maclaurin:

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!} = \frac{x^1}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ...$$

Com essa soma, pode-se montar o seguinte código utilizando a ISA proposta:

sen.txt

```
SET $15 3.14
SET $3 40
CPY $2 $15
CPY $4 $15
SET $5 3
SET $6 1
SET $7 -1
MUL $6 $7 $6
EXP $15 $5 $4
MUL $4 $6 $4
CPY $8 $4
CPY $9 $5
CPY $10 $9
SUB $10 $1 $10
IET $10 $0 18
MUL $10 $9 $9
HOP 14
DIV $8 $9 $11
SUM $11 $2 $2
SUM $5 $1 $5
SUM $5 $1 $5
ILT $5 $3 8
MEMF $2
```

Esse código foi nomeado como "sen.txt". O primeiro valor colocado no registrador 15 é o valor em radianos recebido. Um detalhe importante desse programa é que, para facilitar na velocidade mas tendo uma precisão numérica decente, foram feitas somas até n=40, sendo representado com o seguinte somatório:

$$\sin(x) = \sum_{n=0}^{40} \frac{(-1)^n x^{2n+1}}{(2n+1)!}$$

Para o Cosseno, o procedimento de encontrar as derivadas e montar o somatório baseado no padrão deles é exatamente o mesmo. Seguindo os mesmos passos, obtem-se a série que define o cosseno:

$$cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!} = \frac{x^0}{0!} - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^8}{8!} + ...$$

Então, tem-se o seguinte código:

cos.txt

```
SET $15 3.14
SET $2 1
CPY $3 40
CPY $4 $15
SET $5 2
SET $6 1
SET $7 -1
MUL $6 $7 $6
EXP $15 $5 $4
MUL $4 $6 $4
CPY $8 $4
CPY $9 $5
CPY $10 $9
SUB $10 $1 $10
IET $10 $0 18
MUL $10 $9 $9
HOP 14
DIV $8 $9 $11
SUM $11 $2 $2
SUM $5 $1 $5
SUM $5 $1 $5
ILT $5 $3 8
MEMF $2
```

Esse código foi nomeado de "cos.txt". Seguindo a mesma lógica do código do seno, o primeiro valor representa o valor em radiano abritrariamente escolhido. Além disso, o somatório também possui valor de n=40, ou, escrevendo na notação de série:

$$cos(x) = \sum_{n=0}^{40} \frac{(-1)^n x^{2n}}{(2n)!}$$

Com isso, obtem-se dois somatórios infinitos que definem o seno e o cosseno.

3.2 Implementação do Processador e da Memória Principal

O Processador e a Memória Principal montado tem 5 partes principais:

- **Memória Principal**: Dividida em Memória RAM e memória de instrução, ela lê o arquivo do programa e, baseado no Contador de Programa(PC), envia a instrução de uma determinada linha para a memória de instrução e manda para a Unidade de Controle;
- **Unidade de Controle**: Interpreta as instruções enviadas da Memória Principal (mais especificamente a memória de instrução);
- **Unidade de Lógica e Aritmética (ALU)**: Realiza as operações aritméticas, condicionais e saltos das instruções;
- **Banco de Registradores**: Onde ficam os registradores que irão guardar e manipular os valores baseado nas instruções enviadas;
- **CPU**: Possui internamente a **Unidade de Controle**, a **ALU** e **o Banco de Registradores**, é ela que controla o funcionamento de tudo, recebendo informações da Memória Principal, mandando as informações para as suas partes etc;

Cada uma dessas partes foram divididas e, para algumas, foram criadas alguma Estruturas Heterogêneas de dados para facilitar na criação do programa.

Um detalhe importante a comentar é sobre a representação dos registradores no programa. Como temos o limite de 16 registradores a serem utilizados, foi criado uma estrutura heterogênea para ele que é um vetor de 16 posições no qual cada posição representa um registrador, facilitando a ilustração dos registradores além de ser mais simples a manipulação de seus valores.

```
typedef struct{
    double info;
}regs;

regs R[16]; //representacao da criacao do vetor de registradores
```

Código 1: Estrutura dos **Registradores**

3.2.1 Analisando a Memória Principal

Como dito anteriormente, a memória principal irá ler o arquivo, além de mandar para a CPU qual a instrução que ele solicitou. Para isso, temos duas funções diferentes:

- **GastoMemoria()**: recebe o nome do arquivo e calcula o custo de armazenamento desse arquivo na memória. Ele irá contar quantos bits cada instrução tem e, caso ela o total de 32 bits ou a Memória RAM fique cheia (ultrapasse os 16MBs), o programa é abortado por sobrecarregamento.
- **EnviaInstrucao()**: recebe da CPU o Contador de Programa e o nome do arquivo, definindo qual instrução a memória deve acessar e retorna a instrução;

Para o **GastoMemoria()**, a memória de instrução lê o nome do opcode para saber qual o tamanho da instrução que ela está lidando. Com isso, ela manda o valor em bits para a RAM e vai executando até o fim do arquivo.

```
    for(int i=0; i<max_linhas; i++){

        fgets(linha,100, entry);
        mem.Mem_Instrucao = analise_opcode(linha);
        if(mem.Mem_Instrucao<=MAX_INSTRU){
            mem.RAM=mem.RAM+mem.Mem_Instrucao;
        }
        if(mem.RAM>MAX_RAM || mem.Mem_Instrucao>MAX_INSTRU){
            printf("Memoria Principal Sobrecarregada. Abortar Execucao!\n");
            return 0;
        }
    }
```

Código 2: função **GastoMemoria()**

Na função **EnviaInstrucao()**, para facilitar na leitura da instrução pela Unidade de Controle, a instrução será armazenada em uma string, depois será criada um novo arquivo chamado **instrucao.txt** e, em seguida, armazenará a string e retornando 1 para mandar para a CPU que a função funcionou.

```
    for(int i=0; i<PC; i++)
        fgets(mem.instrucao, 100, entry);

    FILE* instrucao = fopen("instrucao.txt", "w");
    fprintf(instrucao, "%s", mem.instrucao);
    fclose(entry);
    fclose(instrucao);
    return 1;
```

Código 3: função **EnviaInstrucao()**

3.2.2 Analisando a Unidade de Controle

Após a Memória Principal enviar o instrucao.txt para a CPU, a Unidade de Controle irá acessar esse arquivo e processar a instrução. A instrução poderia ter 4 tipo de dados:

- Um opcode de 3 a 4 letras (obrigatória em todas as instruções, sempre no começo da linha da instrução);
- Um valor de endereço, representado por "$"seguido do número que define a posição no vetor de registradores;
- Um valor do tipo float para armazenar nos registradores;
- Um valor do tipo inteiro para representar uma determinada linha que vai ser mandada para o Contador de Programa;

Inicialmente, a Unidade de Controle abre o arquivo instrucao.txt e recebe a primeira string, que é o opcode da instrução. Então, ele lê a variável e analisa qual opcode se trata, dependendo do opcode, ele irá pegar o resto dos valores que estão na instrução. Veja o exemplo da instrução SET, que recebe o valor em float e o coloca em um registrador:

```
int escolha(regs* R, int PC, FILE* exit){
    char command[4], regc[4];
    limpeza(command);
    limpeza(regc);
    FILE* entry = fopen("instrucao.txt", "r");
    fscanf(entry, "%s", command);
    if(!strcmp(command, "SET")){
        int reg;
        float info;
        fscanf(entry, "%s", regc);
        reg=EnderecoToInt(regc);
        fscanf(entry, "%f", &info);
        fclose(entry);
        SET(R, reg, info);
}
    //...
```

Código 4: funcionamento da interpretação dos opcodes na Unidade de Controle

Um detalhe importante é que, como os registradores são representados com um "$"do lado, o programa lê o registrador como uma string e o converte para inteiro, removendo o cifão do lado do número e mandando o número para uma variável. Esse processo é feito na função **EnderecoToInt()**, que recebe a string e retorna um número:

```
int EnderecoToInt(char endereco[4]){

    int num,aux=2;
    char enderecoC[3];
    if(endereco[3]=='\0')
        aux=3;
    for(int i=0; i<aux; i++)
        enderecoC[i]=endereco[i+1];
    num=atoi(enderecoC);
    return num;
}
```

Código 5: função **EnderecoToInt()**

A função **Escolha()** sempre retorna uma variável inteira chamada PC, que representa o Contador de Programa, isso porque, após processar as aritméticas da ALU, o programa pode alterar o Contador e mudar o fluxo do programa. Caso isso não seja feito, o programa retorna o mesmo valor do PC que recebeu. Essa mudança pode ser vista no opcode HOP, ILT e IET, que alteram o valor do PC:

```
    if(!strcmp(command, "IET")){

        int reg[2], line;
        for(int i=0; i<2; i++){
            fscanf(entry, "%s", regc);
            reg[i]= EnderecoToInt(regc);
        }
        fscanf(entry, "%d", &line);
        fclose(entry);
        return IET(R, reg[0], reg[1], line,PC);
    }
```

Código 6: Exemplo do funcionamento da função **IET**, que altera o fluxo do programa e, por consequência, é utilizada como retorno da função **Escolha()**

3.2.3 Analisando a ALU

A ALU é um conjunto de funções que cada uma delas faz uma operação aritmética diferente baseado no nome da sua instrução e seus "Parâmetros"recebidos da Unidade de Controle. Um exemplo disso é na função **SUM()**, que soma os valores de dois registradores e armazena no último. Ele possui como parâmetro apenas os valores da posição do vetor dos registradores e os utiliza para fazer as operações.

```
    void SUM(regs* R, int r1, int r2, int r3){
        R[r3].info = R[r1].info + R[r2].info;
    }
```

Código 7: função **SUM()**

Como comentado no tópico "Analisando a Unidade de Controle", existem funções da ALU que alteram o fluxo do programa, mais especificamente a variável PC, que representa o Contador de Programa. Todas essas funções são variações da função **HOP()** que, dado uma linha, ele retorna essa linha:

```
    int HOP(int line){
        if(line < 0)
            return 0;
        return line;
    }
```

Código 8: função **HOP()**

As funções **IET()** e **ILT()** possuem a função **HOP()** na sua lógica, com a diferença de possuirem uma condição e, caso a condição seja atendida, eles retornam a chamada da função de salto:

```
//  Verifica se r1 eh menor que r2, caso positivo, pule para linha recebida (
    r1 < r2 ? line)
int ILT(regs* R, int r1, int r2, int line, int PC){
    if(R[r1].info < R[r2].info){
        return HOP(line)-1;
    }
    else{
        return PC;
    }
}
```

Código 9: função **ILT()**, (Is Less Then)

3.2.4 Analisando a CPU

Com todas as estruturas montadas, foi feito a CPU em si, que iria trabalhar com a informação recebida das outras estruturas. Primeiramente, a CPU cria o seu tipo e define os primeiros dois valores dos registradores para 0 e 1, respectivamente.

```c
int main() {
    CPU cpu;
    cpu.R[0].info = 0.0;
    cpu.R[1].info = 1.0;
    //...
```

Código 10: iniciação dos registradores na função **main()**

A CPU recebe o nome do Arquivo que vai ser executado e, em seguida, manda para a memória armazená-lo. Caso tenha espaço, ele inicia o Contador de Programa, que o manda como parâmetro novamente para a memória e retorna um valor de saída. Caso o valor de saída seja 0, significa que o contador já alcançou a última linha do arquivo, finalizando a execução do programa. Caso seja 1, ele executa a função da Unidade de Controle e retorna o valor do PC, que pode ter sido alterado ou não.

Não só isso, mas, para que a memória sempre continue sendo consumida durante o programa, mostrando seu gasto durante a execução, foi criado um loop que mantém o código aceitando mais arquivos para a execução, sendo que ele só termina a execução quando a palavra "sair"é escrita no terminal.

```c
do {
        saida = 1;
        printf("Nome do arquivo de entrada: ");
        scanf("%s", Nome_arquivo);

        if(!strcmp(Nome_arquivo, SAIR_STRING)){
            printf("Finalizando...\n");
            return 0;
        }
        //...
        cpu.Contador_Programa = 1;
        while (saida == 1) {
            saida = EnviaInstrucao(cpu.Contador_Programa, Nome_arquivo);
            if (saida){
                cpu.Contador_Programa = escolha(cpu.R, cpu.Contador_Programa,
                    result);
                cpu.Contador_Programa++;
            }
        }
        //...
        printf("Programa Finalizado! \nRAM utilazada: %d bits\n\n", GastoM);

    } while (strcmp(Nome_arquivo, SAIR_STRING));
```

Código 11: função **main()**

Como é possível ver, ao final de cada execução do programa, a mensagem falando que o programa foi finalizado é mostrada junto com o valor de bits consumidos pela RAM durante o decorrer do programa.

4 Resultados

Com tudo montado, os testes foram realizados. As métricas de compilação foram colocadas em um MakeFile, que possui as seguintes compilações:

Métricas de Compilação

```
gcc CPU.c -c -Wall
gcc memoria.c -c -Wall
gcc ALU.c -c -Wall
gcc Unit_Control.c -c -Wall
gcc Unit_Control.o CPU.o ALU.o memoria.o -o exe -lm
```

Então, para executar, é dado o comando:

Métricas de Compilação

```
make
./exe
```

Com isso, o programa é executado. Será observado, em partes, o funcionamento de cada código.

4.1 Resultados do código dos números primos

Quando executado o código.Vemos a seguinte mensagem no terminal:

```
Nome do arquivo de entrada: primo.txt
Programa Finalizado!
RAM utilazada: 320 bits


Nome do arquivo de entrada:
```

Analisando o arquivo de saída do programa "saida.txt", temos os seguintes valores:

saida.txt

```
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97
```

Assim, percebe-se que o programa funciona exatamente como esperado, tendo os valores formatados em inteiros.

4.2 Resultados do código do Seno

Executando agora o "sen.txt"no terminal, vê-se, novamente, uma mensagem de confirmação:

```
Nome do arquivo de entrada: sen.txt
Programa Finalizado!
RAM utilazada: 796 bits


Nome do arquivo de entrada:
```

Analisando novamente o arquivo "saida.txt", temos o seguinte valor:

saida.txt

```
0.001593
```

O valor em radiano colocado no arquivo "sen.txt", assim como no arquivo "cos.txt", é 3.14, uma aproximação da constante π.

Olhando uma calculadora qualquer, vemos que:

$$sen(3.14) = 0.0015926529164868$$

Assim, nota-se que o valor obtido é bem próximo do valor real.

4.3 Resultados do código do Cosseno

Por fim, executando o arquivo "cos.txt"no terminal, vê-se, novamente, uma mensagem de confirmação:

```
Nome do arquivo de entrada: cos.txt
Programa Finalizado!
RAM utilazada: 1292 bits


Nome do arquivo de entrada:
```

Analisando novamente o arquivo "saida.txt", temos o seguinte valor:

saida.txt

```
-0.999999
```

Novamente, analisando uma calculadora de cosseno, utilizando o mesmo valor de pi, tem-se:

$$cos(3.14) = -0.99999873172754$$

Assim como o seno, os valores são muito próximos do real, mostrando, dessa forma, que todos os resultados de seno e cosseno funcionam como esperado.

Então, com todos os testes terminados, basta colocar o comando "sair"para fechar o programa.

```
Nome do arquivo de entrada: sair
Finalizando...
```

5 Conclusão

O objetivo do trabalho era consolidar as nossas próprias instruções em um programa feito em C emulando um hardware baseado na arquitetura de Von Neumann. Essa emulação deveria ser capaz de realizar duas funções: calcular os 100 primeiros números primos e calcular o seno e cosseno de um valor em radianos.

Para isso, foram feitos duas coisas: a ISA, definindo as instruções a serem utilizadas junto com os códigos dos primos, seno e cosseno, e a segunda parte focada na montagem da emulação do hardware, definindo funções específicas para cada um e montando a lógica de como cada componente conversa entre si, sendo todos controlados pela CPU. Para a primeira parte, foram usados os conceitos que definem um primo para montar seu programa e, para o seno e cosseno, foi utilizado a expansão de série de MacLaurin para definir a função de seno e cosseno em uma soma infinita de termos. Como não é possível fazer uma soma infinita em C, foi escolhido um valor abirtrário de somas de 40 termos.

Os resultados do código saíram como esperados, mostrando os valores dos primos e os valores de seno e cosseno. Na saída, também mostra o consumo de RAM baseado no tamanho das instruções, sendo que o código continua sendo executado pedindo novos arquivos até que a palavra "sair"seja escrita no terminal.

Com isso, concluímos que, tanto a CPU quanto a memória montada funcionam como esperado afim de resolver os problemas propostos. Não só isso, mas coloca em prova o funcionamento da Arquitetura de Von Neumann, mostrando que o modelo é válido até mesmo na emulação de hardware.