Organizando e emulando um processador através de um projeto de ISA

Gustavo Henrique Correia Ferreira e Marcelo Vieira Magalhães

***Resumo –* O documento conhecido como ISA (Instruction Set Architecture) refere-se a um conjunto de instruções que um processador pode executar, definindo desde as operações que serão suportadas até o número de registradores. Nesse trabalho, será desenvolvida a organização de um processador com sua ISA consolidada, além de um emulador para realizar dois algoritmos propostos.**

# INTRODUÇÃO

O design de computadores modernos é baseado na arquitetura de John Von Neumann, que é caracterizada pela presença de alguns princípios chave como uma unidade central de processamento, um sistema de memória principal e um sistema de entrada/saída. A figura 1 representa o conceito da arquitetura de Von Neumann [1].

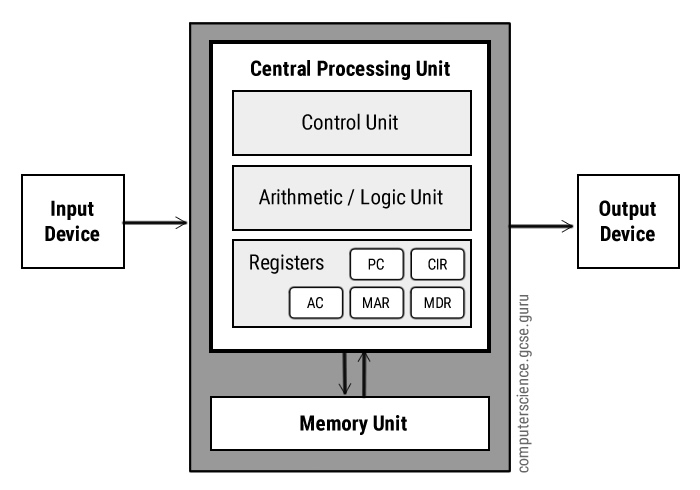


Figura 1. Representação da arquitetura de John Von Neumann

Uma ISA, do inglês Instruction Set Architecture, é um documento que organiza os elementos de um projeto de processador, definindo alguns parâmetros como as operações suportadas, número de registradores, tamanho das instruções e número máximo de instruções.

No presente trabalho, abordaremos a organização de um processador com uma ISA consolidada, através de algumas regras de projeto pré estabelecidas. Um emulador também deve ser construído e ser capaz de realizar dois algoritmos: armazenar todos os números primos entre 1 e 100 em memória e depois imprimir os valores lidos e calcular o seno e o cosseno de um valor em radianos.

A seção II apresenta um breve referencial teórico sobre o projeto e uma contextualização do cenário no qual ele foi desenvolvido. Em seguida, na seção III, será abordada a metodologia adotada no desenvolvimento da ISA e na implementação do emulador, apresentando o passo-a-passo realizado e os métodos utilizados para avaliar os resultados. Na seção IV serão apresentados os resultados obtidos por meio da abordagem adotada. Finalmente, na seção V, serão feitas as conclusões e reflexões sobre o trabalho realizado.

# REFERENCIAL TEÓRICO

A arquitetura de Von Neumann contém a organização básica de um computador e seus princípios, e baseado nesses conceitos é possível projetar uma ISA que possa ser executada eficientemente na arquitetura. O conjunto de instruções deve ser pensado para funcionar de acordo com as características da arquitetura.

Para o desenvolvimento da ISA algumas regras de projeto foram estabelecidas: máximo de 16 instruções; operações aritméticas podem ser soma, subtração, multiplicação, divisão ou shifts; máximo de 16 registradores; instruções de no máximo 32 bits; existência de memória principal e banco de registradores, sem cache; memória principal com até 16MB; 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.

Através das regras pré-estabelecidas e do estudo do funcionamento geral da arquitetura de John Von Neumann, foram definidas treze instruções que serão suportadas pelo emulador:

* ADD RX RY - O acumulador recebe a soma entre os registradores X e Y;
* SUB RX RY - O acumulador recebe a subtração entre os registradores X e Y;
* MUL RX RY - O acumulador recebe a multiplicação entre os registradores X e Y;
* DIV RX RY - O acumulador recebe a divisão entre os registradores X e Y;
* REM RX RY - O acumulador recebe o resto da divisão entre os registradores X e Y;
* STO RX - O registrador X recebe o conteúdo do acumulador.
* STI RX Y - O registrador X recebe o valor Y.
* BEQ RX RY Z - Caso o conteúdo dos registradores X e Y seja igual, há um salto para a linha Z.
* BLT RX RY Z - Caso o conteúdo do registrador X seja menor que o do registrador Y, há um salto para a linha Z.
* JMP X - Salto para a linha X.
* DSP - Empurra o conteúdo do acumulador para o display.
* WHD RX RY – Escreve o conteúdo de RX na memória principal na posição equivalente ao valor de RY.
* RHD RX RY – Lê na memória principal o float de endereço RY e o armazena no registrador RX.

Também foram definidos tipos abstratos de dados para cada uma das unidades do processador, sendo elas: memória principal, memória de instruções, leitor de programa, banco de registradores e unidade lógica e aritmética.

Utilizando apenas as treze instruções definidas na ISA, foram desenvolvidos dois algoritmos para serem realizados no emulador: primeiro consiste na impressão de todos os números primos entre 1 e 100, enquanto o segundo consiste em calcular o seno e o cosseno de um valor em radianos.

Todas essas definições foram de suma importância para a realização do trabalho, cuja implementação será descrita na seção a seguir.

# METODOLOGIA

O primeiro passo no desenvolvimento do emulador foi a definição e implementação dos tipos abstratos de dados que seriam utilizados.

Como ao final da emulação será necessário armazenar os resultados encontrados, foi criado um TAD para representar a memória principal. Esse TAD irá realizar a manipulação do arquivo de texto que será gerado pelo programa, intitulado “memoria.dat”. O TAD contém 4 funções: duas são responsáveis por inicializar e fechar o arquivo de texto e outras duas por ler e escrever no arquivo. Devido a regra de projeto inicialmente estabelecida que a memória principal deve ter até 16MB, as funções de leitura e escrita foram projetadas para não operar caso o tamanho do arquivo seja maior do que esse valor.

Também se mostrou necessária a criação de um tipo abstrato para representar a unidade lógica e aritmética, a implementação da struct ALU se encontra na figura 2:

typedef struct *alu*

{

    float a, b, acc;

} *ALU*;

Figura 2. Implementação da struct ALU

O TAD da unidade lógica e aritmética possui funções para atribuir um valor ao “a” e ao “b”, além de uma função responsável por retornar o valor contido no acumulador. Outras sete funções são responsáveis pela aritmética em si, realizando as seguintes operações com “a” e “b”: soma, subtração, multiplicação, divisão, resto, verificação de igualdade e verificação de menor elemento.

Para representar a memória de instruções foi criado um TAD de mesmo nome cuja implementação da struct encontra-se na figura 3.

typedef struct *memoria\_instrucoes* {

    char\*\* comando;

    int n;

} *MInst*;

Figura 3. Implementação da struct MInst

A estratégia utilizada foi armazenar linha por linha do arquivo de texto com os comandos em um vetor de strings chamado “comando”, além de realizar uma contagem do número de comandos e armazenar em “n”. O TAD realiza três funções: “guardaPrograma”, responsável por percorrer o arquivo de texto com os comandos e guardar linha por linha no vetor de strings; “contaLinhas”, responsável por contar o número total de comandos (ou linhas) do arquivo de texto; “liberaMemoriaInst”, responsável por liberar qualquer memória que tenha sido alocada dinamicamente pelo próprio TAD.

A fim de cumprir o requisito estabelecido inicialmente, um TAD para representar o banco de registradores também foi criado, e sua implementação encontra-se na figura 4.

typedef struct *banco\_registradores* {

    float registrador[16];

} *BancoReg*;

Figura 4. Implementação do TAD banco de registradores

O TAD tem como objetivo armazenar todos os registradores, com um número máximo de 16 conforme definido nas regras de projeto.

Por fim, fez-se necessário a criação de um TAD responsável por ler e executar o programa em si, a esse TAD foi dado o nome de “leitor de programa”. O leitor de programa recebe as informações de todas as outras TAD: ALU, banco de registradores, memória principal e memória secundária. É através do leitor que o programa será lido e executado linha a linha, utilizando a função “executaPrograma”. A estratégia da função consiste em percorrer todas as linhas do programa através do vetor de strings localizado no TAD da memória de instruções, identificar pelos três primeiros caracteres da string qual deverá ser a operação executada e os seus argumentos.

Para simplificar o código e evitar grandes quantidades de texto no leitor, foi criado um arquivo chamado “opcodes.c”, contendo a implementação das funções de comando utilizadas no leitor. Assim, é feita uma conexão entre o leitor e a ALU, que é acionada toda vez que uma operação aritmética é realizada, além da conexão entre o leitor e a memória principal toda vez que os comandos WHD e RHD são executados. O leitor também é capaz de identificar comentários no código, que se iniciam com “//”, e de identificar comandos que não são suportados pelo emulador, imprimindo uma mensagem que informa ao usuário que o comando utilizado é inválido.

Para finalizar a implementação do emulador, a função “main” localizada no arquivo “main.c” dá início ao programa, permitindo que o usuário escolha qual algoritmo será executado e realizando a chamada das funções pertencentes aos tipos abstratos de dados implementados.

# RESULTADOS

Para testar o emulador implementado, foram desenvolvidos algoritmos escritos nos moldes da ISA, utilizando apenas os treze comandos suportados. As instruções de compilação se encontram no anexo A.

O primeiro dos algoritmos realiza o cálculo de todos os números primos entre 1 e 100, armazena esses números na memória principal e os imprime na tela, o código completo do algoritmo se encontra no anexo B. A estratégia utilizada consiste em imprimir os quatro primeiros números primos: 2, 3, 5 e 7, e depois verificar o resto da divisão de todos os números de 8 a 100 pelos 4 primos iniciais. Caso o resto de alguma das divisões seja igual a 0, o número atual não é impresso e sofre um incremento de 1, caso o resto de nenhuma das divisões seja igual a 0 o número é impresso e o loop se repete até que o número atual seja igual a 100, alcançando a condição de parada do algoritmo.

O emulador apresentou eficiência na resolução do problema, imprimindo os resultados corretos na tela e salvando-os no arquivo de texto “memoria.dat”, que representa a memória principal.

Para o problema de cálculo de seno e cosseno foram desenvolvidos os algoritmos que se encontram nos anexos C e D, é necessário inserir o número para o cálculo no próprio arquivo txt. O emulador executa os algoritmos com eficiência, sendo capaz de realizar o passo a passo corretamente, porém, devido a algum erro na forma como o algoritmo foi estruturado, o cálculo do seno e do cosseno não apresenta os resultados esperados.

# CONCLUSÃO

O design de computadores modernos é baseado na arquitetura de John Von Neumann, caracterizada pela presença de alguns princípios chave como uma unidade central de processamento, um sistema de memória principal e um sistema de entrada/saída. Uma ISA, do inglês Instruction Set Architecture, é um documento que organiza os elementos de um projeto de processador, definindo alguns parâmetros como as operações suportadas, número de registradores, tamanho das instruções e número máximo de instruções. No presente trabalho, foi desenvolvido um emulador de processador com uma ISA consolidada capaz de realizar dois algoritmos implementados.

Foram definidos tipos abstratos de dados para representar as unidades do processador, dentre elas: memória principal, memória de instruções, banco de registradores, unidade lógica e aritmética e leitor de programa. Cada tipo de dados possui estratégias para conseguir se relacionar com as outras unidades.

Foram desenvolvidos algoritmos de acordo com os comandos suportados pelo emulador, a fim de testar a integração entre as unidades do emulador e a eficiência de um código escrito nos padrões da ISA definida.

Para o algoritmo que calcula os números primos de 1 a 100 tanto o código quanto o emulador se mostraram eficientes na resolução do problema, imprimindo na tela os números corretos e salvando em um arquivo de texto responsável por representar a memória principal.

Para o algoritmo que calcula o seno e o cosseno de um número em radianos o emulador se mostrou eficiente, realizando o passo a passo corretamente, porém, o código desenvolvido apresentou erros. O algoritmo desenvolvido em opcode não realiza o cálculo do seno e cosseno corretamente, e o motivo desse erro não foi encontrado.

A dificuldade na escrita de código em opcode reflete a importância do desenvolvimento das linguagens de alto nível em programação, que facilitam o entendimento do código criando níveis de abstração. Trabalhos subsequentes podem explorar a resolução dos erros encontrados no algoritmo para cálculo de seno e cosseno.

# REFERÊNCIAS

[1] ComputerScience.GSCE.GURU. Von Neymann Architecture. Disponível em: https://www.computerscience.gcse.guru/theory/von-neumann-architecture. Acesso em: [13 de janeiro de 2024].

# ANEXOS

1. *Instruções de compilação*

gcc \*.c -o exe

1. *Opcode números primos 0 a 100*

STI 0 0

STI 1 1

STI 2 7

STI 3 2

STI 4 3

STI 5 5

STI 6 4

STI 12 7

STI 7 100

ADD 3 0

DSP

WHD 3 0

ADD 4 0

DSP

WHD 4 1

ADD 5 0

DSP

WHD 5 3

ADD 2 0

DSP

WHD 2 4

ADD 2 1

STO 2

BEQ 7 2 43

REM 2 3

STO 8

REM 2 4

STO 9

REM 2 5

STO 10

REM 2 12

STO 11

BEQ 8 0 22

BEQ 9 0 22

BEQ 10 0 22

BEQ 11 0 22

ADD 2 0

DSP

WHD 2 6

ADD 6 1

STO 6

JMP 21

STI 0 0

1. *Opcode cálculo de seno*

STI 0 0

STI 1 1

STI 2 2

STI 6 0

SUB 6 1

// 6 armazena -1

STO 6

// 3 armazena n

STI 3 0

// 4 armazena x

STI 4 30

// 5 armazena aux1

STI 5 0

// 7 armazena maxIteracoes

STI 7 10

// 8 armazena aux2

STI 8 0

// 9 armazena saida

STI 9 0

// 10 armzn contadorGenerico

STI 10 0

// 11 armazena aux3

STI 11 0

// começo do loop

BEQ 3 7 82

STI 10 0

STI 5 1

// loop -1^n ini

BEQ 3 10 32

MUL 5 6

STO 5

JMP 27

// loop -1^n fim

STI 10 0

MUL 3 2

STO 8

ADD 8 1

STO 8

STO 11

// loop fat 2n+1 ini

BEQ 8 10 49

MUL 11 10

STO 11

ADD 10 1

STO 10

JMP 41

// loop fat 2n+1 fim

// trata 0!

BEQ 8 0 51

JMP 53

STI 8 1

// divisao dos termos

DIV 5 8

STO 5

// 2n+1

MUL 2 3

STO 11

ADD 11 1

STO 11

STI 10 0

STO 8 1

// loop x^2n+1 ini

BEQ 11 10 71

MUL 8 4

STO 8

ADD 10 1

STO 10

JMP 63

// loop x^2n+1 fim

// multiplica os termos

MUL 5 8

STO 5

// soma na saida

ADD 5 9

STO 9

// n++

ADD 3 1

STO 3

// volta pro começo

JMP 25

// fim do loop

DSP 9

// fim do programa

1. *Opcode cálculo de cosseno*