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].

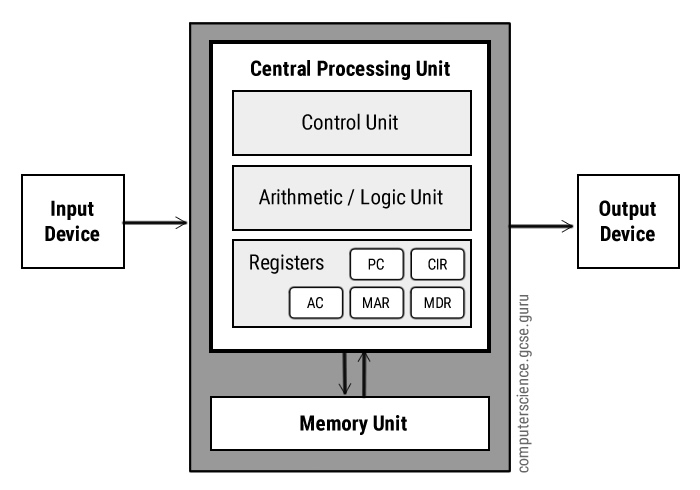


Figura 1. Representação da arquitetura de John Von Neumann

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 onze 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.

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 onze 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 na implementação dos programas foi a conversão da representação binária para caracteres. Para isso, na função principal o usuário digita o texto em binário, que é lido como uma string. Foi utilizada a função *split*, nativa do Python, para transformar cada elemento da string em um elemento de uma lista. Logo após, cada elemento da lista é convertido de binário para inteiro e de inteiro para caractere utilizando as funções *int* e *chr*, respectivamente. Com isso, o primeiro passo é concluído, e ao invés de uma sequência de binários o programa agora deve lidar com uma sequência de caracteres embaralhados. Após esse passo, o texto é passado como parâmetro para uma função denominada *decodifica*, que irá retornar o texto já decodificado.

A partir daqui os caminhos para decodificar a cifra de César e a cifra de substituição se ramificam. Enquanto que na cifra de César serão necessárias apenas 26 iterações para descobrir a frase decodificada, na cifra de substituição foram adotadas 30 mil iterações para alcançar o objetivo.

Começando pela cifra de César, a solução encontrada foi realizar um loop para testar todos os 26 shifts e calcular a probabilidade de cada um ter sido escrito na língua inglesa com o auxílio da biblioteca N-Gram. O shift é feito de acordo com a iteração do loop, por exemplo, para a primeira iteração, o shift será 0, para a segunda o shift será 1 e assim por diante, o alfabeto correspondente ao shift atual do loop consiste em uma lista gerada a partir de outra lista contendo o alfabeto em sua ordenação normal. Com o alfabeto chave em mãos, a cada iteração é chamada a função *atribuiLetras*, que recebe como parâmetro um texto embaralhado e uma lista contendo um alfabeto a ser utilizado como chave. A implementação dessa função pode ser observada na figura 3:

def atribuiLetras(*texto*, *alfabetoAleatorio*):

    textoDecodificado = ''

    for letra in *texto*:

        if letra in alfabeto:

            posicao = *alfabetoAleatorio*.index(letra)

            textoDecodificado += alfabeto[posicao]

        else:

            textoDecodificado += letra

    return textoDecodificado

Figura 3. Implementação da função atribuiLetras

O texto e sua probabilidade gerados por cada shift são armazenados em uma lista, e ao final aquele com a maior probabilidade é retornado pela função *decodifica* e impresso na função principal.

Para o algoritmo que quebra a cifra de substituição é necessário importar a biblioteca Random, que será responsável por gerar as aleatoriedades do programa. A ideia desenvolvida foi iniciar o algoritmo com um alfabeto aleatório gerado através do embaralhamento do alfabeto ordenado. Esse alfabeto aleatório foi chamado de *alfabetoAtual*, e a partir dele, chamando a função apresentada na figura 3, é gerado um texto chamado de *textoAtual*, cuja probabilidade de pertencer à língua inglesa é calculada com o auxílio da biblioteca N-Gram e armazenada em uma variável chamada *scoreAtual*. Também é necessário criar duas variáveis para armazenar o melhor texto e a melhor probabilidade, que serão inicializadas com os dados do *textoAtual* e do *scoreAtual*.

Com o ponto de partida definido, o próximo passo foi gerar alfabetos parecidos com o alfabeto atual, realizando um determinado número de trocas de letras. Para isso, foi criada uma função chamada *criaAlfabetoParecido*, que recebe como parâmetro o alfabeto atual e um número de trocas. A implementação dessa função encontra-se na figura 4:

def criarAlfabetoParecido(*alfabetoAtual*, *trocas*):

    alfabetoParecido = *alfabetoAtual*.copy()

    for i in range(*trocas*):

        letra1 = random.sample(alfabetoParecido, 1)[0]

        indexLetra1 = alfabetoParecido.index(letra1)

        letra2 = random.sample(alfabetoParecido, 1)[0]

        indexLetra2 = alfabetoParecido.index(letra2)

        alfabetoParecido[indexLetra1] = letra2

        alfabetoParecido[indexLetra2] = letra1

    return alfabetoParecido

Figura 4. Implementação da função criarAlfabetoParecido

A ideia consiste em comparar a probabilidade do texto que foi gerado a partir do alfabeto atual com a do texto que foi gerado a partir do alfabeto parecido. Para isso, será chamada a função *atribuiLetras* (apresentada na figura 3) para o alfabeto parecido, e a probabilidade do texto gerado será calculada. Caso o alfabeto parecido seja mais provável, atualizamos o alfabeto atual, o texto atual e a probabilidade atual para receber os dados do alfabeto parecido. Logo após, é necessário comparar a probabilidade atual com a melhor probabilidade encontrada até o momento, e atualizar caso a atual seja mais alta que a melhor. Esse processo deve ser repetido por um número x de iterações, que após vários testes foi definido como 30 mil. O algoritmo possui capacidade para encontrar a frase correta bem antes disso, mas esse número se apresentou como uma margem segura para diferentes entradas.

Inicialmente o número de trocas da função *criarAlfabetoParecido* (apresentada na figura 4) foi determinado como fixo e igual a 3, porém, essa não é uma boa prática, tendo em vista que os testes com o algoritmo não foram satisfatórios. Isso ocorre pois em determinados pontos do algoritmo serão necessários menos que 3 trocas para encontrar um texto com maior probabilidade que o atual. A solução encontrada foi utilizar da aleatoriedade para realizar as trocas: foi definido um salto inicial de 1000 iterações, onde o número de trocas será um número aleatório entre 3 e 5, e após esse salto inicial, o número de trocas será um número aleatório entre 1 e 3. Com essas alterações, resultados melhores foram encontrados.

Após vários testes, foi possível notar que durante várias execuções o algoritmo convergia para uma zona sub-ótima. Essa zona consiste em uma solução que não é a melhor possível e que prende o nosso algoritmo. Para contornar esse problema, foi necessário definir uma reinicialização do alfabeto atual após um número x de iterações sem melhoria na probabilidade. Após vários testes para determinar esse número de iterações, 2000 se mostrou uma boa quantidade. Com isso, o problema da convergência para a zona sub-ótima foi resolvido.

Assim, ao final das 30 mil iterações a função irá retornar o texto decodificado para ser impresso na função principal.

Após finalizada a lógica do programa, foram adicionadas algumas linhas de código a fim de melhorar a experiência do usuário: no início do programa o usuário tem a opção de escolher se a entrada será em string ou em binário; foram adicionadas macros para gerar resultados coloridos no terminal; a biblioteca Os foi importada para limpar o terminal em momentos específicos da execução; foram implementados dois contadores: um para informar ao usuário em qual iteração o programa está (exibe apenas iterações múltiplas de 1000) e outro para informar em qual iteração o melhor texto foi encontrado.

# RESULTADOS

Para testar o algoritmo da cifra de César foi utilizado o código do anexo A, e o algoritmo se mostrou eficiente 100% das vezes. Isso ocorre pois para a cifra de César existem poucas possibilidades de resultado, o que permite que todas sejam testadas e que o algoritmo encontre a melhor alternativa dentro desse pequeno conjunto. A frase encontrada para o texto binário do anexo A foi:

*“it has been said that astronomy is a humbling and character building experience there is perhaps no better demonstration of the folly of human conceits than this distant image of our tiny world to me it underscores our responsibility to deal more kindly with one another and to preserve and cherish the pale blue dot the only home we have ever known”*

Para o algoritmo da cifra de substituição foram utilizados os dois códigos binários que se encontram em anexo, e os resultados se mostraram satisfatórios na grande maioria dos testes. A frase encontrada para o texto binário do anexo B foi:

*“the day we cease the exploration of the cosmos is the day we threaten the continuing of our species in that bleak world arms bearing resource hungry people and nations would be prone to act on their low contracted prejudices and would have seen the last gasp of human enlightenment until the rise of a visionary new culture that once again embraces the cosmic perspective a perspective in which we are one fitting neither above nor below but within”*

Uma referência à música Exist, da banda Avenged Sevenfold.

A fim de realizar mais testes com o algoritmo de substituição foram gerados outros textos codificados em um programa externo. O resultado da decodificação trouxe conclusões interessantes a respeito da maneira como o algoritmo foi feito: foi possível notar uma certa dificuldade do algoritmo em atribuir corretamente as letras X e Z. Um exemplo foi a frase abaixo, encontrada como solução durante várias execuções do programa:

*“brawils rich history unfolds like an epic tale from the indigenous peoples xho first inhabited its lands to the colonial era marked by european conquest the struggle for independence the abolition of slavery and the birth of a diverse nation illustrate its resilience through revolutions and reforms brawils history reflects its vibrant cultural tapestry the countrys journey continues shaped by its people and a vision for a brighter future”*

O motivo disso ocorrer provavelmente se deve ao fato da baixa utilização das letras X e Z na língua inglesa [4], o que faz com que o algoritmo não seja muito preciso para frases onde essas letras apareçam com frequência.

O algoritmo para decodificar a cifra de substituição não é exato, e é fato que quanto mais iterações forem realizadas, mais próximo o melhor texto estará do texto descriptografado.

# CONCLUSÃO

Nas linguagens de programação, é possível converter uma representação binária em um número de base 10, que pode ser convertido para um caractere utilizando a tabela ASCII. Nesse trabalho, foram desenvolvidos algoritmos para descriptografar textos representados na forma binária que foram criptografados utilizando a cifra de César e a cifra de substituição.

Para o desenvolvimento dos algoritmos, foi utilizada a linguagem de programação Python com ajuda da biblioteca N-Gram que é responsável por calcular a probabilidade de determinado texto ter sido escrito na língua inglesa. Em ambos os programas o usuário digita um texto em binário que é convertido para inteiro e depois para caractere, e a string de caracteres é passada para uma função decodificadora. Aqui, para o algoritmo de César, são testadas 26 possibilidades com o auxílio da função *atribuiLetras*, e o resultado é retornado para a função principal. No algoritmo da cifra de substituição, é gerado um alfabeto aleatório e a partir dele é gerado um alfabeto parecido através da função *criarAlfabetoParecido*, os dois são comparados e o melhor é escolhido. Esse processo se repete por 30 mil iterações, e foi desenvolvida uma reinicialização quando o algoritmo atinge 2000 iterações sem melhoria, evitando que ele atinja uma zona sub-ótima.

O algoritmo para a cifra de César apresentou o resultado esperado, sendo eficiente em 100% dos casos, enquanto o algoritmo para a cifra de substituição não alcançou o resultado desejado em algumas execuções.

Mesmo com a imperfeição do algoritmo de substituição, é evidente o poder de ambos os algoritmos e suas inúmeras aplicações práticas. O comportamento geral apresentado pelos programas foi conforme o esperado, e trabalhos subsequentes podem explorar a imprecisão do algoritmo de substituição quanto às letras X e Z.

# 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. *Representação binária de texto 1*

1011000 1001001 100000 1010111 1010000 1001000 100000 1010001 1010100 1010100 1000011 100000 1001000 1010000 1011000 1010011 100000 1001001 1010111 1010000 1001001 100000 1010000 1001000 1001001 1000111 1000100 1000011 1000100 1000010 1001110 100000 1011000 1001000 100000 1010000 100000 1010111 1001010 1000010 1010001 1000001 1011000 1000011 1010110 100000 1010000 1000011 1010011 100000 1010010 1010111 1010000 1000111 1010000 1010010 1001001 1010100 1000111 100000 1010001 1001010 1011000 1000001 1010011 1011000 1000011 1010110 100000 1010100 1001101 1000101 1010100 1000111 1011000 1010100 1000011 1010010 1010100 100000 1001001 1010111 1010100 1000111 1010100 100000 1011000 1001000 100000 1000101 1010100 1000111 1010111 1010000 1000101 1001000 100000 1000011 1000100 100000 1010001 1010100 1001001 1001001 1010100 1000111 100000 1010011 1010100 1000010 1000100 1000011 1001000 1001001 1000111 1010000 1001001 1011000 1000100 1000011 100000 1000100 1010101 100000 1001001 1010111 1010100 100000 1010101 1000100 1000001 1000001 1001110 100000 1000100 1010101 100000 1010111 1001010 1000010 1010000 1000011 100000 1010010 1000100 1000011 1010010 1010100 1011000 1001001 1001000 100000 1001001 1010111 1010000 1000011 100000 1001001 1010111 1011000 1001000 100000 1010011 1011000 1001000 1001001 1010000 1000011 1001001 100000 1011000 1000010 1010000 1010110 1010100 100000 1000100 1010101 100000 1000100 1001010 1000111 100000 1001001 1011000 1000011 1001110 100000 1001100 1000100 1000111 1000001 1010011 100000 1001001 1000100 100000 1000010 1010100 100000 1011000 1001001 100000 1001010 1000011 1010011 1010100 1000111 1001000 1010010 1000100 1000111 1010100 1001000 100000 1000100 1001010 1000111 100000 1000111 1010100 1001000 1000101 1000100 1000011 1001000 1011000 1010001 1011000 1000001 1011000 1001001 1001110 100000 1001001 1000100 100000 1010011 1010100 1010000 1000001 100000 1000010 1000100 1000111 1010100 100000 1011010 1011000 1000011 1010011 1000001 1001110 100000 1001100 1011000 1001001 1010111 100000 1000100 1000011 1010100 100000 1010000 1000011 1000100 1001001 1010111 1010100 1000111 100000 1010000 1000011 1010011 100000 1001001 1000100 100000 1000101 1000111 1010100 1001000 1010100 1000111 1001011 1010100 100000 1010000 1000011 1010011 100000 1010010 1010111 1010100 1000111 1011000 1001000 1010111 100000 1001001 1010111 1010100 100000 1000101 1010000 1000001 1010100 100000 1010001 1000001 1001010 1010100 100000 1010011 1000100 1001001 100000 1001001 1010111 1010100 100000 1000100 1000011 1000001 1001110 100000 1010111 1000100 1000010 1010100 100000 1001100 1010100 100000 1010111 1010000 1001011 1010100 100000 1010100 1001011 1010100 1000111 100000 1011010 1000011 1000100 1001100 1000011

1. *Representação binária de texto 2*

1010110 1000011 1001111 100000 1000001 1010100 1000100 100000 1001000 1001111 100000 1010011 1001111 1010100 1001101 1001111 100000 1010110 1000011 1001111 100000 1001111 1010000 1001010 1001011 1010111 1001001 1010100 1010110 1001110 1010111 1011010 100000 1010111 1000111 100000 1010110 1000011 1001111 100000 1010011 1010111 1001101 1010001 1010111 1001101 100000 1001110 1001101 100000 1010110 1000011 1001111 100000 1000001 1010100 1000100 100000 1001000 1001111 100000 1010110 1000011 1001001 1001111 1010100 1010110 1001111 1011010 100000 1010110 1000011 1001111 100000 1010011 1010111 1011010 1010110 1001110 1011010 1010101 1001110 1011010 1010010 100000 1010111 1000111 100000 1010111 1010101 1001001 100000 1001101 1001010 1001111 1010011 1001110 1001111 1001101 100000 1001110 1011010 100000 1010110 1000011 1010100 1010110 100000 1011000 1001011 1001111 1010100 1011001 100000 1001000 1010111 1001001 1001011 1000001 100000 1010100 1001001 1010001 1001101 100000 1011000 1001111 1010100 1001001 1001110 1011010 1010010 100000 1001001 1001111 1001101 1010111 1010101 1001001 1010011 1001111 100000 1000011 1010101 1011010 1010010 1001001 1000100 100000 1001010 1001111 1010111 1001010 1001011 1001111 100000 1010100 1011010 1000001 100000 1011010 1010100 1010110 1001110 1010111 1011010 1001101 100000 1001000 1010111 1010101 1001011 1000001 100000 1011000 1001111 100000 1001010 1001001 1010111 1011010 1001111 100000 1010110 1010111 100000 1010100 1010011 1010110 100000 1010111 1011010 100000 1010110 1000011 1001111 1001110 1001001 100000 1001011 1010111 1001000 100000 1010011 1010111 1011010 1010110 1001001 1010100 1010011 1010110 1001111 1000001 100000 1001010 1001001 1001111 1001100 1010101 1000001 1001110 1010011 1001111 1001101 100000 1010100 1011010 1000001 100000 1001000 1010111 1010101 1001011 1000001 100000 1000011 1010100 1000010 1001111 100000 1001101 1001111 1001111 1011010 100000 1010110 1000011 1001111 100000 1001011 1010100 1001101 1010110 100000 1010010 1010100 1001101 1001010 100000 1010111 1000111 100000 1000011 1010101 1010001 1010100 1011010 100000 1001111 1011010 1001011 1001110 1010010 1000011 1010110 1001111 1011010 1010001 1001111 1011010 1010110 100000 1010101 1011010 1010110 1001110 1001011 100000 1010110 1000011 1001111 100000 1001001 1001110 1001101 1001111 100000 1010111 1000111 100000 1010100 100000 1000010 1001110 1001101 1001110 1010111 1011010 1010100 1001001 1000100 100000 1011010 1001111 1001000 100000 1010011 1010101 1001011 1010110 1010101 1001001 1001111 100000 1010110 1000011 1010100 1010110 100000 1010111 1011010 1010011 1001111 100000 1010100 1010010 1010100 1001110 1011010 100000 1001111 1010001 1011000 1001001 1010100 1010011 1001111 1001101 100000 1010110 1000011 1001111 100000 1010011 1010111 1001101 1010001 1001110 1010011 100000 1001010 1001111 1001001 1001101 1001010 1001111 1010011 1010110 1001110 1000010 1001111 100000 1010100 100000 1001010 1001111 1001001 1001101 1001010 1001111 1010011 1010110 1001110 1000010 1001111 100000 1001110 1011010 100000 1001000 1000011 1001110 1010011 1000011 100000 1001000 1001111 100000 1010100 1001001 1001111 100000 1010111 1011010 1001111 100000 1000111 1001110 1010110 1010110 1001110 1011010 1010010 100000 1011010 1001111 1001110 1010110 1000011 1001111 1001001 100000 1010100 1011000 1010111 1000010 1001111 100000 1011010 1010111 1001001 100000 1011000 1001111 1001011 1010111 1001000 100000 1011000 1010101 1010110 100000 1001000 1001110 1010110 1000011 1001110 1011010