PROJETO MEMÓRIA CACHE

ERICH BRENDELL ARAUJO MEDEIROS
FERNANDO LUCAS SOUSA SILVA
TEÓPHILO VITOR DE CARVALHO CLEMENTE

Universidade Federal do Rio Grande do Norte, Departamento de Engenharia de Computação e Automação

1. INTRODUÇÃO

Com a progressão da tecnologia durante os anos, o computador se tornou um exemplo de componente que foi sendo modificado em conjunto, tornando-se fundamental no desenvolvimento de várias atividades no dia-a-dia. Assim, diversas pesquisas e estudos foram realizados com o intuito de aprimorar sua arquitetura e performance. Com isso, após esses desenvolvimentos a arquitetura, atualmente, é composta por três elementos principais: o processador, a memória e os módulos de entrada e saída (E/S).

No mais, o computador tem um elemento central para seu funcionamento, que é o processador ou CPU, ele é responsável por receber, trabalhar e processar os dados e instruções gerados pelos softwares instalados.

Se tratando da memória temos RAM, ROM, a CACHE que trataremos nesse relatório e também HD's e SSD's, cada tipo de memória irá guardar dados específicos para o funcionamento da máquina e de modo específico. Nas arquiteturas vemos que a memória tem papel central, portanto sua proximidade ou distância do processador, irá definir se ela é uma memória de rápido ou lento acesso, de grande ou pequeno espaço, de modo que as informações mais usadas e importantes tendem a ficar mais próximas para facilitar o acesso.

2. A MEMÓRIA CACHE

A memória cache é a mais próxima ao processador, sendo assim ela trabalha com as informações mais acessadas durante a execução dos processos, ela fica justamente antes da memória principal, pois o processador primeiro fará a busca nela, caso não ache que irá para as outras hierarquias de memória, desse modo, ela se torna imprescindível para que o sistema tenha melhores tempos de resposta. “O uso da memória cache visa obter velocidade de memória próxima das memórias mais rápidas que existem e, ao mesmo tempo, disponibilizar uma memória de grande capacidade ao preço de memórias semicondutoras mais baratas.”, (STALLINGS, 2010, cap. 4, pág. 95).

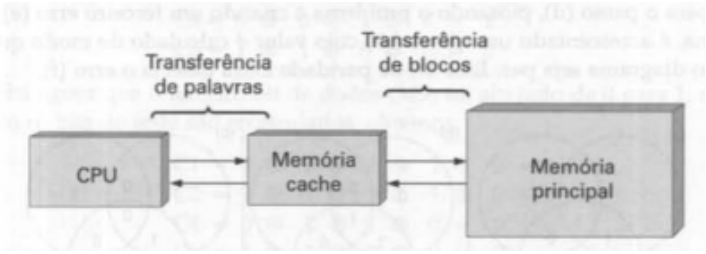

Figura 1 - Relação da memória cache com CPU e memória principal.

2.1 Endereços de cache

Os endereços podem ser armazenados de forma física ou virtual. No caso do virtual, que é a mais comum, utilizamos as chamadas memórias virtuais, elas fazem o endereçamento independente da quantidade física disponível. Para realizar tal processo é utilizada uma unidade de gerenciamento de memória (MMU), ela fará a tradução dos endereços lógicos para físicos na memória principal. Na cache física os dados são armazenados em endereços físicos alocados na memória principal, tendo então que passar pela MMU para realizar a tradução. Já a cache lógica, armazena os seus dados utilizando endereços virtuais, desse modo, o processador consegue acesso direto a cache, sem passar pela MMU, o que aumenta a velocidade do processo.

2.2. Tamanho da memória cache

O tamanho da cache é outro conceito importante para o desenvolvimento da cache, pois o ideal é que se implemente-a de forma a ter um tamanho pequeno o suficiente para o custo por bit ser próximo da memória principal sozinha e possuir o tempo de acesso próximo ao da memória cache sozinha.

Assim, ainda que possam ser fabricadas com circuitos integrados e utilizadas da mesma forma que as de tamanho pequeno, as caches de tamanhos grandes tendem a ter um desempenho mais lento (STALLINGS, 2010). Portanto, não há como ter tamanhos padrão para as memórias cache, eles serão definidos pela utilidade que se deseja e deve-se realizar o equacionamento para que a memória tenha um desempenho satisfatório para o objetivo do projetista.

2.3. Função de Mapeamento

Essa função é necessária para determinar qual o bloco da memória principal que se encontra na cache, uma vez que essa última possui menos linhas do que os blocos da memória principal. A escolha desse mapeamento determina como a memória cache é organizada. As funções de mapeamento serão melhor abordadas nos próximos tópicos.

2.3.1 Direta

A função de mapeamento direto, funciona mapeando cada bloco da memória principal a apenas uma linha de cache (linhas fixas). Para encontrar a linha da cache a qual o bloco da memória principal será alocado, pega-se o número do bloco e divide pela quantidade de linhas da cache, pegando o resto da divisão para saber onde irá alocar. Dessa forma, quando se percorre os blocos da memória principal e aplica-se a operação descrita, a função checa para qual linha irá alocar aquele determinado bloco. A seguir é possível observar através da figura a esquematização da função de mapeamento direto.

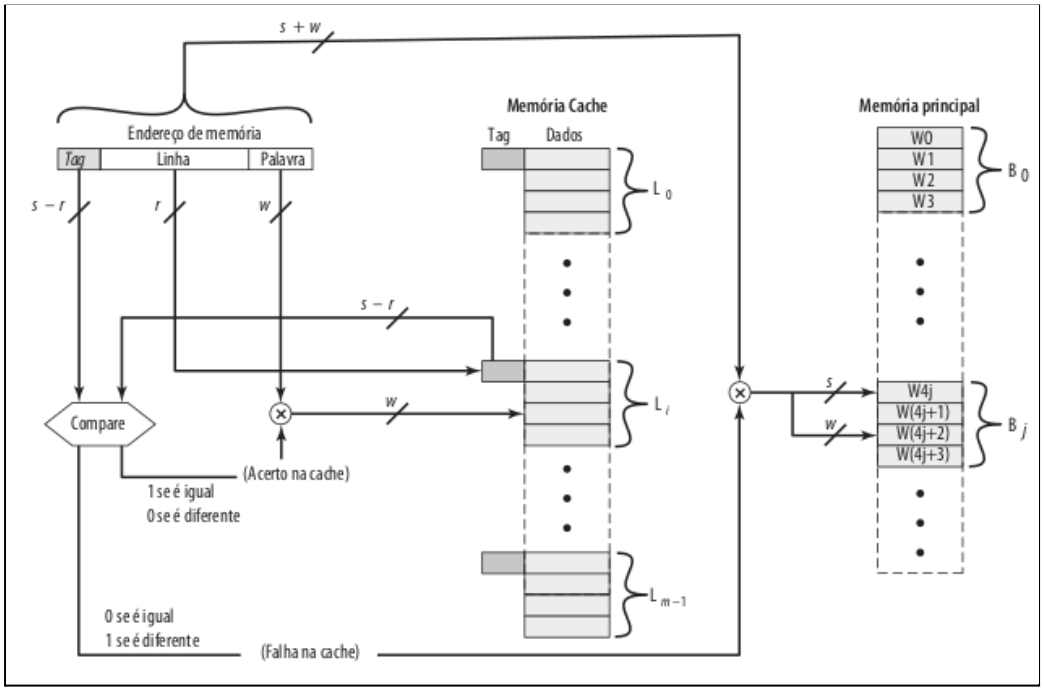

Figura 2 - Mapeamento direto.

O acesso à cache ocorre da seguinte maneira: cada endereço da memória principal é enxergado sendo constituído de três campos, onde os bits menos significativos identificam uma Palavra (word) ou um byte seguidos pelos bits restantes que são interpretados como uma Tag, composta pela parte mais significativa, (digamos **r** bits), e um campo de linha de **r** bits. Há também um segundo campo que é responsável por identificar as linhas da cache.

Caso o processador necessite consultar ciclicamente informações que são alocadas na mesma linha da cache e por vezes elas estejam em blocos diferentes da memória principal, ou seja, que tem o mesmo resultado da operação (número do bloco módulo quantidade de linhas), vai ser gerado um *thrashing*. Onde esse problema irá acarretar num baixo desempenho computacional.

2.3.2 Associativa

O mapeamento associativo permite que os blocos da memória principal sejam alocados em qualquer linha da cache. Dessa forma elimina-se o problema de *thrashing* que é gerado pelo forma direta. Para saber onde o bloco vai ser carregado, a cache interpreta um endereço da memória como sendo apenas composto pela campos Tag e Palavra, cada qual com suas respectivas interpretações. A Tag identifica o bloco da memória principal e comparando simultaneamente a tag de cada linha, é possível determinar se aquele determinado já foi alocado na cache.

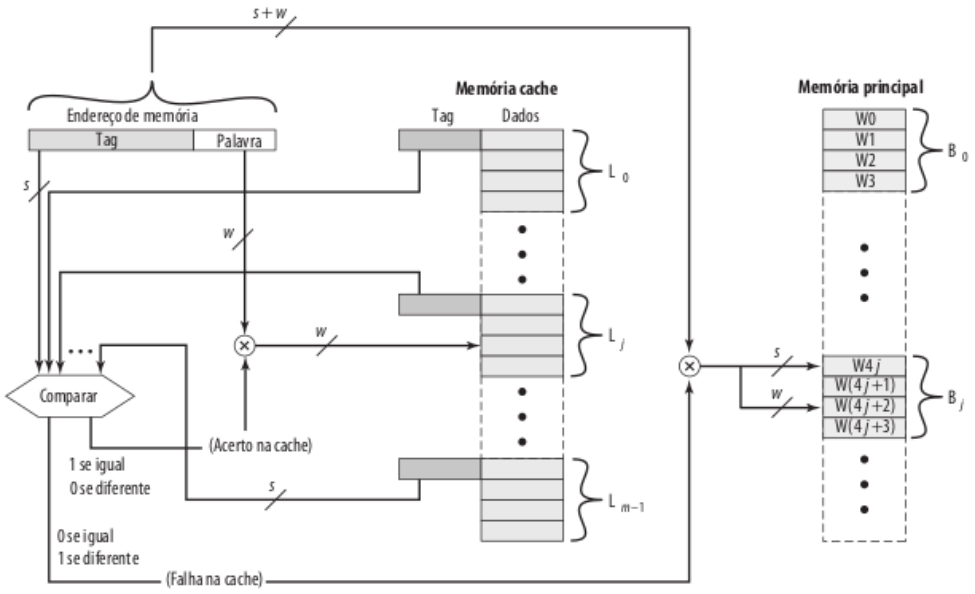

Figura 3 - Mapeamento associativo.

Com todas as linhas ocupadas, é necessário utilizar um algoritmo de substituição de linha de cache que será descrito na seção seguinte.

2.3.3 Associativa por conjuntos

Esta forma de mapeamento faz uma mescla entre as vantagens do direto e o totalmente em conjunto, tentando reduzir as suas desvantagens. Através de uma função

que executa o seguinte algoritmo: o número do bloco módulo número de conjuntos da memória cache. Onde cada bloco que contém o endereço a ser consultado, é alocado na primeira linha livre do conjunto. Caso todas as linhas do conjunto estejam ocupadas e uma informação necessite ser alocada, é utilizado um algoritmo de substituição. A figura seguinte ilustra como funciona a função de mapeamento associativo por conjunto.

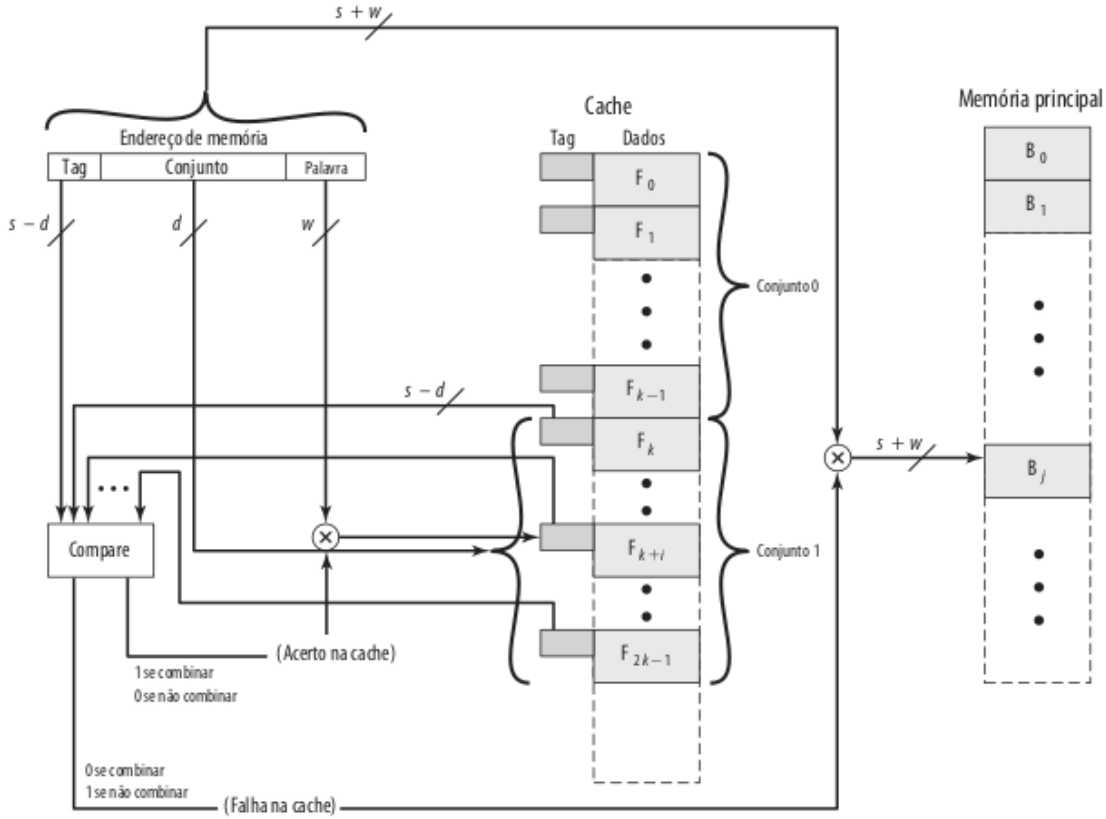

Figura 4 - Mapeamento associativo conjunto.

Como é possível observar há três campos para a lógica da cache interpretar, que são eles: Tag, Conjunto e Palavra cujo funcionamento é semelhante aos anteriores, ou seja, é uma mescla entre eles.

2.4. Algoritmo de Substituição

Esse algoritmo serve para realizar a substituição de um bloco da cache quando há a necessidade, devido à adição de um novo bloco cache e a cache já estiver cheia. Com isso, faz-se necessário a implementação de um algoritmo para os métodos: associativo e associativo por conjuntos, no caso do mapeamento direto não é necessário pois só há uma linha possível para o bloco.

Um dos algoritmos mais eficientes é o último recentemente usado (LRU, em inglês), o qual faz a substituição do bloco que está a mais tempo sem ser referenciado. Isso é feito utilizando um bit da linha chamado de USE, o qual assume estado 1 quando a linha é referenciada e 0 caso não seja. (STALLINGS, 2010).

2.5. Política de escrita

No que diz respeito à escrita em memórias podemos enfrentar alguns problemas, nos quais escritas podem não corresponder, como por exemplo: múltiplos processadores são conectados ao mesmo barramento e cada processador tem sua própria cache local, então, se uma palavra for alterada em uma cache, ela possivelmente poderia invalidar uma palavra em outras cache (STALLINGS, 2010). Para resolver esse problema utilizamos as políticas de escrita, são técnicas para manter a coerência entre o que está armazenado na cache e na memória principal.

A técnica mais comum é a write through, ela implica que toda operação seja feita na memória principal e na memória cache, de modo que impede que endereços divergentes sejam trabalhados, porém tem a desvantagem de gerar muito tráfego. Nesse caso, também temos a técnica write back, ela realiza a atualização apenas na cache e utiliza um bit de modificação para informar o que precisa ser atualizado na memória principal no ciclo seguinte, porém ele também tem a desvantagem de que pode deixar os blocos da memória principal inválidos e assim alguns acessos só serão permitidos através da cache.

2.6. Tamanho da linha

O tamanho da linha pode ser mudado quando se adiciona novas informações a um bloco tornando-o maior que anteriormente, isso irá influenciar na taxa de acertos e falhas da cache em um bloco, uma vez que a medida que o tamanho dos blocos aumentam, inicialmente, a taxa de acertos irá aumentar por causa do princípio da localidade. Esse princípio afirma que os dados em torno de uma palavra referenciada poderão ser referenciados em um futuro próximo.

Contudo, quando se aumenta ainda mais, ocorre a diminuição dos acertos, pois as chances de se utilizar uma informação recente torna-se menor do que reutilizar um dado que precisa ser substituído. Como cita Stallings (2010, cap. 4, pág. 111): "O relacionamento entre o tamanho do bloco e a razão de acerto é complexo, dependendo das características de localidade de um programa em particular, e nenhum valor ideal definitivo foi encontrado."

2.7. Números de memórias cache

Esse aspecto tem haver com o multinível das caches, uma vez que, quando começaram a ser implementadas, a memória cache possuía apenas um nível, isto é, era apenas um componente. Atualmente, a técnica de multinível permitiu uma melhoria de desempenho e redução da utilização do sistema de barramento para a transmissão de dados de uma memória cache L2 (segunda camada) para o processador, por exemplo. Em geral, as camadas possuem a letra L e um número. A camada L1 é a mais próxima do processador, com isso, quanto maior o número, mais longe do processador essa camada vai estar.

Com isso é possível descrever que a L1 (on chip) por estar mais próxima do processador, reduz a atividade que o mesmo necessita fazer caso fosse consultar uma L2 por exemplo (que geralmente se encontra em um barramento externo). Dessa forma aumentasse o desempenho geral do sistema.

Há ainda duas ressalvas a serem citadas a respeito das caches que se encontram fora do chip do processador, que seriam as L2.A primeira é que para reduzir a carga sobre o

barramento do sistema usa-se um caminho de dados separado e a segunda seria que atualmente com o avanço da tecnologia e o encolhimento contínuo dos componentes, alguns processadores incorporam também outros níveis de cache além da primeira.

3. PROJETANDO MEMÓRIA CACHE

Para o projeto da memória cache vamos fazer seu mapeamento, através da técnica direta e associativa, no caso o mapeamento direto é o mais simples, de modo que, cada bloco da memória principal é associado apenas a uma linha de cache, tendo três campos na memória principal para representar um endereço, o índice, a tag e o endereço da palavra.

O processo do mapeamento direto começa com a busca da linha desejada, após é realizada a comparação dos respectivos valores. Se os valores não forem iguais, a linha desejada não está na cache e então deve ser procurada na memória principal e substituída na linha referida da cache. Caso sejam iguais, a palavra é transferida para a CPU.

Já o mapeamento associativo faz com que um bloco da memória principal seja transportado por qualquer linha da cache. Desse modo, para que ela funcione é preciso que ocorra uma comparação da TAG de cada linha com a TAG do endereço para determinar se o bloco está na cache. Se existir a correspondência, os bytes em questão serão transferidos para a CPU, caso contrário, o endereço será procurado na memória principal e guardado na cache.

3.1. Implementação em Python

Utilizando de conhecimentos teóricos e práticos adquiridos acerca da memória cache que foram apresentados anteriormente, além de embasamento anterior com a linguagem de programação Python, foi possível realizar a implementação para os métodos de mapeamento direto e associativo visando realizar simulações para determinados valores de endereços e observar como ambos os métodos se diferenciam.

A figura 5, apresentada abaixo, mostra a primeira parte do código, importamos uma biblioteca do python que se chama numpy, ela é responsável por realizar a conversão dos

números decimais para números binários presentes em nosso arquivo de endereços, com 16 bits. Além disso, há a declaração das variáveis necessárias para realizar a simulação da memória cache: erros e acertos como inteiros, inicialmente com o valor 0, já que serão incrementados durante a simulação; "enderecos" com a quantidade de endereços do arquivo "enderecos.dat", e memoria, uma lista (um array) responsável por armazenar, em cada endereço, o bit de validade e a TAG, estes últimos serão extremamente importantes para definição de certo e errado.

```
import numpy as np
erros = 0
acertos = 0
enderecos = 11699
memoria = []
```

Figura 5 - Declaração das variáveis e importação da biblioteca numpy.

No primeiro for é feito o loop por todos os endereços, ele fará com que a memória seja inicializada para todos os endereços, já que o bit está setado como False e a TAG estará vazia (none), o alocamento na memória é feito pela função append(). Com isso, a variável memória funciona como uma matriz, em que a coluna 0 apresentará os bits de validade, sendo true ou false, já a coluna 1 apresenta o valor da TAG armazenado no respectivo endereço. Após é feita a leitura do arquivo de endereços, nele em cada linha serão feitas diversas operações.

```
for i in range(enderecos):
  lista = [False, None]
  memoria.append(lista)
```

Figura 6 - Primeiro laço de repetição contada.

Além disso, de acordo com a codificação apresentada na figura 7, há a leitura dos endereços contidos no arquivo "enderecos.dat" disponibilizado. Assim, para cada linha do arquivo através do comando: (for line in arq), serão realizadas operações, dentre elas: converter o valor da linha de decimal para binário de 16 bits com o comando: np.binary_repr().

Após isso, se tornou necessário conhecer cada parte desse binário através dos bits de cada divisão (TAG, Endereço e Palavras), com os dados fornecidos previamente de: 4 palavras (words) por bloco e a cache com 256 linhas, foi possível calcular a quantidade de bits para cada divisão, sendo: 8 bits para as linhas já que 2^8 = 256; 2 bits para as 4 palavras (words) sabendo que 2^2 = 4; a TAG é definida pelos 16 bits do número binário subtraído dos bits destinados às outras divisões, sendo assim, 16 bits (tamanho do endereço) - 8 bits (destinados a linha) - 2 bits (destinados a word) = 6 bits para referenciar a TAG.

Assim, utilizamos da variável "num" responsável por armazenar o número em binário e dividiu em quantidades de bits calculadas anteriormente resultando em: os primeiros 6 bits foram destinados à TAG (bit 0 a 5, visto no comando num[0:6]), os 8 bits do meio foram destinados à linha (bit 6 a 14, visto no comando num[6:14]) e os 3 últimos bits foram destinados à palavra (bit 14 a 16, visto no comando num[14:16]). Após realizar as divisões de acordo com os bits, foram criadas duas variáveis para armazenar esses valores na base decimal novamente, são elas: "endereco" e "tag".

```
for line in arq:
  num = np.binary_repr(int(line), width=16)
  tag_bin = num[0:6]
  endereco_bin = num[6:14]
  word_bin = num[14:16]
  endereco = int(endereco_bin, 2)
  tag = int(tag_bin, 2)
```

Figura 7 - Leitura do arquivo e definição das partes binárias.

3.1.1 Situações de mapeamento

Para fazer o mapeamento direto propriamente dito, temos três possíveis situações que resultaram no código apresentado na figura 8, a seguir:

```
if memoria[endereco][0] == False:
  memoria[endereco][0] = True
  memoria[endereco][1] = tag
  erros += 1
elif memoria[endereco][1] != tag:
  memoria[endereco][1] = tag
  erros += 1
else:
  acertos += 1
```

Figura 8 - Implementação das situações de erros e acertos.

3.1.1.1 Situação 1

Na primeira situação (no if) o bit de validade está com o valor falso, nesse caso ele está fazendo uma conferência na memória para aquele endereço. Caso realmente seja falso, mudamos para true o bit de validade e a TAG irá armazenar o valor da variável tag codificada do endereço lido. Como a situação contabiliza um erro, temos a respectiva variável incrementada.

3.1.1.2 Situação 2

Nessa o bit de validade se apresenta com valor true, entretanto, o valor na TAG da cache para o endereço é diferente da TAG vinda do endereço lido do arquivo. No elif que vemos no código é feita a conferência para o endereço em questão, verificando se a TAG realmente apresenta um valor diferente. Caso realmente seja true, atualizamos o valor da TAG do respectivo endereço de memória lido e também irá contabilizar uma falha, incrementando em 1 a variável falhas.

3.1.1.3 Situação 3

Para a última situação temos que o bit de validade será verdadeiro e a TAG retirada do endereço lido é igual a TAG armazenada no respectivo endereço da memória cache. Tal situação é vista no else, já passadas as outras

duas verificações, com isso contabilizamos um acerto e incrementamos a variável de acertos.

3.1.2 Resultado de mapeamento

Concluída a leitura de todas as linhas do arquivo e realizada a checagem de falhas e acertos para cada respectivo endereço do arquivo, o loop feito pelo for é finalizado, tendo contabilizado erros e acertos, após isso vamos realizar o cálculo da taxa de falha e de acerto que em seguida serão apresentados como o resultado do código. O cálculo da taxa de acerto é a quantidade de acertos dividida pela quantidade total de endereços, já a taxa de falha é a quantidade de falhas dividida pela quantidade total de endereços.

```
taxa_erros = erros/enderecos
taxa_acertos = acertos/enderecos
print(f'Taxa de erros: {taxa_erros} ou {taxa_erros*100}%')
print(f'Taxa de acertos: {taxa_acertos} ou {taxa_acertos*100}%')
```

Figura 9 - cálculo das taxas de erros e acertos.

O resultado após a compilação está apresentado na figura 10 e o código completo apresentado após a figura:

```
Taxa de erros:  14.368749465766307%
Taxa de acertos: 85.63979827335669%
```

Figura 10 - Resultado final.

```
import numpy as np

erros = 0

acertos = 0

enderecos = 11699

memoria = []

for i in range(enderecos):
```

```
    lista = [False, None]
    memoria.append(lista)
arq = open('./enderecos.dat', 'r')
for line in arq:
    num = np.binary_repr(int(line), width=16)
    tag_bin = num[0:6]
    endereco_bin = num[6:14]
    word_bin = num[14:16]
    endereco = int(endereco_bin, 2)
    tag = int(tag_bin, 2)
    if memoria[endereco][0] == False:
        memoria[endereco][0] = True
        memoria[endereco][1] = tag
        erros += 1
    elif memoria[endereco][1] != tag:
        memoria[endereco][1] = tag
        erros += 1
    else:
        acertos += 1
taxa_erros = erros/enderecos
taxa_acertos = acertos/enderecos
print(f'Taxa de erros:  {taxa_erros*100}%')
print(f'Taxa de acertos: {taxa_acertos*100}%')
```

4. CONCLUSÕES

A partir das análises vistas em sala e pela prática do relatório vemos que a memória cache é fundamental para que o computador apresente um bom desempenho, visto que, a sua inserção permite aumentar a velocidade de resposta dos processos, uma vez que agiliza a entrega de informações importantes ao processador. Além disso, foi possível aprender sobre as características que devem ser levadas em consideração na hora de se projetar uma memória cache.

Assim a simulação requisitada na atividade, foi possível perceber o funcionamento de dois dos tipos de mapeamento, o mais simples, chamado de mapeamento direto e o associativo que é um pouco mais complexo. Utilizando da linguagem de programação Python foi possível realizar a simulação, possibilitando assim o cálculo da taxa de acertos e de erros obtida a partir da busca e checagem da presença de um endereço em determinada linha.

5. REFERÊNCIAS

[1] STALLINGS, Willian. **Arquitetura e Organização de Computadores**. 8. ed. São Paulo: Pearson Education do Brasil, 2010.

[2] TANENBAUM, Andrew S.; AUSTIN, Todd. **Organização estruturada de computadores**. 6. ed. São Paulo: Pearson Education do Brasil, 2013.