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

Simulando o sistema de mémoria cache

BCC266 - Organização de Computadores

Maria Clara Silva Perpetuo
Leandro Augusto Ferreira Santos

Professor: Pedro Henrique Lopes Silva

Ouro Preto
20 de julho de 2023

Sumário

Lista de Figuras

Lista de Códigos Fonte

1 Introdução

Este trabalho, desenvolvido através da linguagem de programação "C", tem como objetivo simular um computador e seu sistema de memória, particulamente com o sistema cache. Para tanto, foi implementado um novo nível de memória cache. O mapeamento associativo ou/e o mapeamento associativo em conjunto para a troca de linhas entre cache e a memória principal (RAM) é a pauta do projeto.

1.1 Especificações do problema

Neste trabalho prático, foi necessário implementar a simulação de um sistema de memória hiérarquico com memória principal(RAM) e várias camadas de memória cache(L1, L2, L3). A memória principal é dividida em blocos e cada memória cache é dividida em linhas.

Além disso, foi preciso implementar pelo menos duas políticas de substituição de linhas de cache. Nesse trabalho foi utilizado 'LFU'(Least Frequent Used) e 'LRU'(Least Recently Used).

1.2 Considerações iniciais

Algumas ferramentas foram utilizadas durante a criação deste projeto:

- Ambiente de desenvolvimento do código fonte: VSCode; [1]
- Linguagem utilizada: C.
- Ambiente de desenvolvimento da documentação: Overleaf LaTeX. [2]

1.3 Ferramentas utilizadas

Algumas ferramentas foram utilizadas para testar a implementação, como:

- *CLANG*: ferramentas de análise estática do código.
- *Valgrind*: ferramentas de análise dinâmica do código.

1.4 Especificações da máquina

A máquina onde o desenvolvimento e os testes foram realizados possui a seguinte configuração:

- Processador: Intel Core i3 10110u;
- Memória RAM: 8Gb;
- Sistema Operacional: Ubuntu;

1.5 Instruções de compilação e execução

Para a compilação do projeto, basta digitar:

[1] VSCode está disponível em https://code.visualstudio.com
[2] Disponível em https://www.overleaf.com/

Compilando o projeto

```
gcc -c cpu.c -Wall
gcc -c generator.c -Wall
gcc -c instruction.c -Wall
gcc -c instructionGeneratorNotSoRandom.c -Wall
gcc -c memory.c -Wall
gcc -c mmu.c -Wall
gcc -c main.c -Wall
gcc cpu.o generator.o instruction.o instructionGeneratorNotSoRandom.o memory.o mmu.o main.o -o exe
```

Método mais prático

```
make
Se possui makefile instalado
```

Usou-se para compilar o código as seguintes opções:

- -g: para compilar com informação de depuração e ser usado pelo Valgrind.

- -Wall: para mostrar todos os possível warnings do código.

Para executar o programa, basta digitar:

```
./exe Downloads/Source/ random 10 2 4 5
```

Onde "10", "2", "4" e "5" podem ser substituidos por outras quantias, sendo esses, em ordem, o tamanho da RAM, e os valores inseridos nas linhas.

2 Desenvolvimento

2.1 Considerações

A troca dos algorítimos de remoção da cache é dada a partir das funções 'define', contidos nos módulos .h do programa. Seguem, abaixo, as principais adaptações do programa:

```
// in constants.h:

#define LFU

in mmu.c:

#ifdef LFU
    // [LFU]
    int menor = cache->lines[0].count;
    for (int i = 1; i < cache->size; i++)
    {
        // confere count -> o n mero de acessos;
        if (menor > cache->lines[i].count)
        {
            pos = i;
            menor = cache->lines[i].count;
        }
        // retorna a posicao c/ o menor numero
    }
#endif

#ifdef LRU
    // [Least Recently Used]
    for (int i = 0; i < cache->size; i++)
    {
        cache->lines[i].tempo_ac++;
        // confere tempo_ac -> ha qto tempo a linha est  sem ser acessada;
        if (cache->lines[i].tempo_ac > cache->lines[pos].tempo_ac)
        {
            pos = i;
        }
        // retorna o maior tempo em pos;

        if (cache->lines[i].tag == address)
            return i;
    }
    cache->lines[pos].tempo_ac++;

#endif
```

Código 1: Exemplo de código LFU e LRU.

2.2 Políticas de mapeamento

Para esse trabalho, foi usado o mapeamento associativo simples, combinado à duas das três políticas de substituição de Cache mais conhecidas: LFU e LRU.

LRU: A ideia principal da política "LRU" consiste em escolher como alvo a linha mais antiga em termos de utilização. Dessa maneira, foi acrescida à struct 'linhas' uma variável auxiliar, nomeada 'tempo-ac'. Essa variável fica responsável por verificar, através de um contador simples, quais são as linhas usadas recentemente, e qual delas é a mais antiga. Uma vez que, dado o acesso à uma linha qualquer, seu contador tempo-ac é zerado, um laço de repetição percorre toda a cache e retorna a linha de maior valor na variável em questão para a função chamadora.

LFU: A política "LFU", por sua vez, escolhe como alvo a linha menos usada em geral, durante a execução do programa. A implementação é dada através de uma outra variável 'int', acrescida à struct 'lines', a qual fica responsável por contar quantas vezes o bloco foi acessado, dentro da mmu.c e outros locais de acesso às linhas. Assim, para implementação da política, um laço de repetição percorre toda a cache e retorna a linha com menor contador.

Ambas as políticas implementadas operam de maneira independente uma da outra. Assim, é definido no módulo "constants.h" qual delas será utilizada.

2.3 Adição de uma terceira cache

Em primeiro momento, foi necessário adicionar uma terceira variável de cache no cpu, mais especificamente na struct que repreenta a máquinha:

```
Instruction *generateMultiInstructions(int multiplicando, int multiplicador)
in cpu.h:

typedef struct
{
    Instruction *instructions;
    RAM ram;
    Cache l1; // cache L1
    Cache l2; // cache L2
    Cache l3; // cache L3
    int missL1, missL2, missL3;
    int hitL1, hitL2, hitL3, hitRAM;
    int totalCost;

} Machine;
```

Código 2: Exemplo de código para a struct máquina.

Após isso, essa terceira cache foi inicializada no start da máquina e finalizada no stop máquina:

```
void start(Machine *machine, Instruction *instructions, int *memoriesSize)
{
        // memoriesSize[0] = ramSize
        startRAM(&machine->ram, memoriesSize[0]);
        startCache(&machine->l1, memoriesSize[1]);
        startCache(&machine->l2, memoriesSize[2]);
        startCache(&machine->l3, memoriesSize[3]);

        machine->instructions = instructions;

        machine->hitL1 = 0;
        machine->hitL2 = 0;
        machine->hitL3 = 0;
        machine->hitRAM = 0;

        machine->missL1 = 0;
        machine->missL2 = 0;
        machine->missL3 = 0;
        machine->totalCost = 0;
}


void stop(Machine *machine)
{
        free(machine->instructions);
        stopRAM(&machine->ram);
        stopCache(&machine->l1);
        stopCache(&machine->l2);
        stopCache(&machine->l3);
}
```

Código 3: Exemplo de código para as funções start e stop.

Também foi necessário adicionar a terceira cache na função que imprime toda a memória, ainda na função cpu.c:

```

void printMemories(Machine *machine)
{
        printf("\x1b[0;30;47m      ");
```

```
        printc("RAM", WORDS_SIZE * 8 + 3);
        printc("Cache L3", WORDS_SIZE * 8 + 10);
        printc("Cache L2", WORDS_SIZE * 8 + 10);
        printc("Cache L1", WORDS_SIZE * 8 + 10);
        printf("\x1b[0m\n");
        for (int i = 0; i < machine->ram.size; i++)
        {
                printf("\x1b[0;30;47m%6d|\x1b[0m", i);
                for (int j = 0; j < WORDS_SIZE; j++)
                        printf(" %6d |", machine->ram.blocks[i].words[j]);
                if (i < machine->l3.size)
                {
                        printf("|");
                        printcolored(machine->l3.lines[i].tag, machine->l3.
                            lines[i].updated);
                        for (int j = 0; j < WORDS_SIZE; j++)
                                printf(" %6d |", machine->l3.lines[i].block.
                                    words[j]);

                        if (i < machine->l2.size)
                        {
                                printf("|");
                                printcolored(machine->l2.lines[i].tag, machine
                                    ->l2.lines[i].updated);
                                for (int j = 0; j < WORDS_SIZE; j++)
                                        printf(" %6d |", machine->l2.lines[i].
                                            block.words[j]);
                                if (i < machine->l1.size)
                                {
                                        printf("|");
                                        printcolored(machine->l1.lines[i].tag,
                                             machine->l1.lines[i].updated);
                                        for (int j = 0; j < WORDS_SIZE; j++)
                                                printf(" %6d |", machine->l1.
                                                    lines[i].block.words[j]);
                                }
                        }
                }
                printf("\n");
        }
}
```

Código 4: Exemplo do código printMemories.

A função acima é responsável pela impressão das memórias na tela. Foi acrescentada a representação da memória cache L3 ao código original, disponibilizado no .zip.
Em mmu.c tambem foram feitas adaptações para a adição da terceira cache, no caso, isso foi feito de forma bem padronizada, basicamente repetindo o que ja tinha sido feito tanto com a primeira cache quanto com a segunda cache:

```

Line *MMUSearchOnMemorys(Address add, Machine *machine, WhereWasHit *
    whereWasHit)
{
    int l1pos = memoryCacheMapping(add.block, &machine->l1);
    int l2pos = memoryCacheMapping(add.block, &machine->l2);
    int l3pos = memoryCacheMapping(add.block, &machine->l3);

    Line *cache1 = machine->l1.lines;
    Line *cache2 = machine->l2.lines;
    Line *cache3 = machine->l3.lines;

    MemoryBlock *RAM = machine->ram.blocks;
```

```
13      int cost = 0;
14
15      // [BLOCO NA L1]
16      if (cache1[l1pos].tag == add.block)
17      {
18          /* Block is in memory cache L1 */
19          cost = COST_ACCESS_L1;
20          *whereWasHit = L1Hit;
21          cache1[l1pos].tempo_ac = 0;
22      }
23
24      // [BLOCO NA L2]
25      else if (cache2[l2pos].tag == add.block)
26      {
27          /* Block is in memory cache L2 */
28          cache2[l2pos].tag = add.block;
29          cost = COST_ACCESS_L1 + COST_ACCESS_L2;
30          *whereWasHit = L2Hit;
31          {
32              // tinha um tmp aqui, mas n o serviu p/ nada..
33
34              if (!canOnlyReplaceBlock(cache1[l1pos]))
35              {
36                  if (!canOnlyReplaceBlock(cache2[l2pos]))
37                  {
38                      cache3[l3pos] = cache2[l2pos];
39                  }
40                  else
41                  {
42                      cache2[l2pos] = cache1[l1pos];
43                  }
44                  cache2[l2pos].tempo_ac = 0;
45                  cache2[l2pos] = cache1[l1pos];
46              }
47          }
48          cache2[l2pos].tempo_ac = 0;
49      }
50
51      // [BLOCO NA L3]
52      else if (cache3[l3pos].tag == add.block)
53      {
54
55          cache3[l3pos].tag = add.block;
56
57          cost = COST_ACCESS_L1 + COST_ACCESS_L2 + COST_ACCESS_L3;
58          *whereWasHit = L3Hit;
59
60          // [SE   O DE BACKUP]
61          if (!canOnlyReplaceBlock(cache1[l1pos]))
62          { // se nao puder substituir -> backup
63              if (!canOnlyReplaceBlock(cache2[l2pos]))
64              {
65                  if (!canOnlyReplaceBlock(cache3[l3pos]))
66                  {                                                   // se nao
                        puder colocar em l3, joga p/ ram;
67                      RAM[cache3[l3pos].tag] = cache3[l3pos].block; // bloco
                            diferente de linha, substituo so o bloco aq entao;
68                  }
69                  else
70                  { // se puder colocar em l3, coloca;
71                      cache3[l3pos] = cache2[l2pos];
72                  }
```

```
73                  cache3[l3pos].tempo_ac = 0;    // acessou a linha l3pos p/
                        conferir se pode substituir;
74                  cache3[l3pos] = cache2[l2pos]; // backup de l2 em l3;
75              }
76              else
77              { // se puder colocar em l2, coloca;
78                  cache2[l2pos] = cache1[l1pos];
79              }
80              cache2[l2pos].tempo_ac = 0;    // acessou a linha l2pos p/
                    conferir se pode substituir;
81              cache2[l2pos] = cache1[l1pos]; // backup de l2 em l1;
82          }
83          cache1[l1pos].tempo_ac = 0; // acessou a linha l1pos p/ conferir se
                pode substituir, logo time = 0;
84
85          updateMachineInfos(machine, whereWasHit, cost);
86          return &(cache3[l3pos]); // achou entao ja retorna q ta em l3;
87      }
88
89      // [BLOCO NA RAM]
90      else
91      {
92          /* Block only in memory RAM, need to bring it to cache and manipulate
                the blocks */
93          int l2pos = lineWhichWillLeave(cache1[l1pos].tag, &machine->l2); /*
                Need to check the position of the block that will leave the L1 */
94          int l3pos = lineWhichWillLeave(cache1[l1pos].tag, &machine->l3);
95
96          if (!canOnlyReplaceBlock(cache1[l1pos]))
97          {
98              /* The block on cache L1 cannot only be replaced, the memories
                    must be updated */
99              if (!canOnlyReplaceBlock(cache2[l2pos]))
100             {
101                 /* The block on cache L2 cannot only be replaced, the memories
                        must be updated */
102                 if (!canOnlyReplaceBlock(cache3[l3pos]))
103                 {
104                     // The block on cache L3 cannot only be replaced, the
                            memories must be updated //
105                     RAM[cache3[l3pos].tag] = cache3[l3pos].block;
106                 }
107                 else
108                 { // posso substituir l3
109                     cache3[l3pos] = cache2[l2pos];
110                 }
111                 cache3[l3pos].tempo_ac = 0;
112                 cache3[l3pos] = cache2[l2pos];
113             }
114             else
115             {
116                 cache2[l2pos] = cache1[l1pos];
117             }
118             cache2[l2pos] = cache1[l1pos]; // se entrar no if, tem q passar
                    por esse estagio aqui ;
119             cache2[l2pos].tempo_ac = 0;
120         }
121         cache1[l1pos].tempo_ac = 0;
122
123         cache1[l1pos].block = RAM[add.block];
124         cache1[l1pos].tag = add.block;
125         cache1[l1pos].updated = false;
```

```
126             cost = COST_ACCESS_L1 + COST_ACCESS_L2 + COST_ACCESS_RAM;
127             *whereWasHit = RAMHit;
128             updateMachineInfos(machine, whereWasHit, cost);
129             return &(cache1[l1pos]);
130         }
131 
132         return &(cache1[l1pos]);
133     }
```

Código 5: Exemplo da função MMUSearchOnMemorys.

A função acima recebe como parâmetro um endereço específico, gerado de maneira aleatória pela aba de instruções. O endereço em questão será operado por uma das operações da máquina, e a função 'mmuSearchOnMemories' fica responsável por sua busca. Assim, as quatro memórias são vasculhadas e, quando o endereço alvo é encontrado, é realizado o cálculo do custo do vasculho, a verificação se o bloco alvo pode ser substituído ou não e a atualização das novas informações na máquina.
Outros ajustes adicionais também são feitos para manter o funcionamento correto da máquina.

2.4 LRU

O código abaixo mostra o algorítimo LRU (Last Recently Used):

```
in constants.h:

#define LFU

in mmu.c:

#ifdef LRU
    // [Least Recently Used]
    for (int i = 0; i < cache->size; i++)
    {
        cache->lines[i].tempo_ac++;
        // confere tempo_ac -> ha qto tempo a linha est  sem ser acessada;
        if (cache->lines[i].tempo_ac > cache->lines[pos].tempo_ac)
        {
            pos = i;
        }
        // retorna o maior tempo em pos;

        if (cache->lines[i].tag == address)
            return i;
    }
    cache->lines[pos].tempo_ac++;

#endif

    return pos;
}
```

Código 6: Exemplo de código do algorítimo LRU.

1. O código começa com um 'ifdef LRU' indicando que esse código só será compilado se a macro 'LRU' estiver definida na constants.h.
2. O loop 'for' passa por todas as linhas de cache, de 0 até cache-¿size - 1, onde cache-¿size é o total de linhas na cache.
3. Para cada linha de cache, o 'tempoac'é imcrementado. Este representa o tempo passado desde que a linha foi acessada.
4. No loop, o código mantém a linha cache com o maior valor de 'tempoac' comparando o valor de tempo da linha atual com a linha na posição 'pos', posterior. Se o tempo da linha atual for maior que da linha na posição 'pos', o 'pos' é atualizado para a linha atual.
5. O loop continua até que todas as linhas da cache sejam examinadas.
6. Depois de encontrar a linha com maior 'tempoac', o código checa se a 'tag' da linha de cache combina com o endereço dado. Se uma linha cache que combine for achada, isso significa que uma informação correspondente ao endereço ja está presente na cache e retorna 'i'.
7. Se o loop não achar uma tag que combine, o código retorna 'pos'.

O propósito é substituir a cache menos usada recentemente(LRU) quando necessário.

2.5 LFU

O código abaixo mostra o algorítimo LFU (Least Frequently Used)::

```
in constants.h:

#define LRU

in mmu.c

#ifdef LFU
    // [LFU]
    int menor = cache->lines[0].count;
    for (int i = 1; i < cache->size; i++)
    {
        // confere count -> o n mero de acessos;
        if (menor > cache->lines[i].count)
        {
            pos = i;
            menor = cache->lines[i].count;
        }
        // retorna a posicao c/ o menor numero
    }
#endif
```

Código 7: Trecho de código referente a LFU

1. O código começa com um 'ifdef LFU' indicando que esse código só será compilado se a macro 'LFU' estiver definida na constants.h.
2. O algoritimo usa a variável 'menor' para definir o menor valor de 'count' em todas as linhas de cache na cache.
3. O loor 'for' passa por todas as linhas da cache.
4. Para cada linha cache, algoritimo compara o 'cont' com o atual valor de 'menor'. Contador representa o número de vezes que a linha cache foi acessada.
5. Se o contador da atual linha cache analizada for menor que 'menor', isso significa que atual foi menos acessada qur a 'menor'. Nesse caso, a 'pos' é atualizada e o valor de menor é atualizado para o atual.
6. O loop continua por todas as linhas da cache.
7. Depois que o loop for completo, a variável 'pos' vai armazenar o index da linha cache com o menor valor de contador.
8. O algorítimo vai retornar 'pos', valor pra trocar a linha cache com a usada menos frequentemente.

3 Experimentos

Os testes foram realizados em computadores com os hardwares presentes na tabela abaixo, a medição de tempo de execução é apresentada a seguir:

Modelo	*RAM*	*Sistema Operacional*	*Processador*
MSI Modern 14	8GB	Ubuntu	Intel Core i3-10110u
HP EliteBook 745 G5	16GB	Ubuntu	AMD Ryzen 7 Pro 2700u

Figura 1: Especificações dos hardwares de teste

3.1 Considerações

Por fim, os resultados obtidos são razoáveis.

Para todos os casos, nos reunimos para analisar o código base e, desse modo, entender a implementação proposta pelo enunciado, tais como operam as empresas para com seus funcionários. A solução das operações não foram as únicas implementadas, graças á dificuldade na implementação e interpretação do código.

Adicionar uma terceira cache e fazé-la funcionar como as outras, foi um verdadeiro desafio. Usamos o código base como exemplo de implementação, mesmo assim, de ínicio, os resultados não foram bons como esperado.

Além disso, adaptar as contas implementadas no primeiro trabalho prático não foi fácil. Foi necessário corrigir alguns erros dos mesmos para que funcionassem no trabalho prático atual.

3.2 Tempo de Execução

Para medir o tempo de execução em cada hardware, foi usada a função "clock()", da biblioteca "time.h". O seguinte trecho de código foi adaptado à função main e, dessa maneira, o cálculo preciso do tempo de execução do código foi retornado.

```
int main() {
    clock_t inicio, fim;
    double tempo_decorrido;

    // [CAPTURA DO TEMPO ATUAL NO IN CIO DO PROGRAMA]
    inicio = clock();

    // [CAPTURA DO TEMPO ATUAL AO FIM DAS EXECU  ES DAS FUN  ES]
    fim = clock();
    tempo_decorrido = ((double)(fim - inicio)) / CLOCKS_PER_SEC;

    printf("Tempo de execucao: %fs\n", tempo_decorrido);

    return 0;
}
```

Código 8: Exemplo de código para a operação de divisão.

As variáveis 'inicio' e 'fim' são do tipo clock-t. Servem para capturar o tempo de execução do programa com precisão de milissegundos no momento em que são chamadas. Dessa forma, a variável 'inicio' é chamada logo após a declaração das variáveis, e a variável 'fim' antes do retorno final do método principal. Assim, o tempo médio de execução para cada hardware é retornado.

A seguir, os gráficos com o tempo estimado para ambos os hardwares:

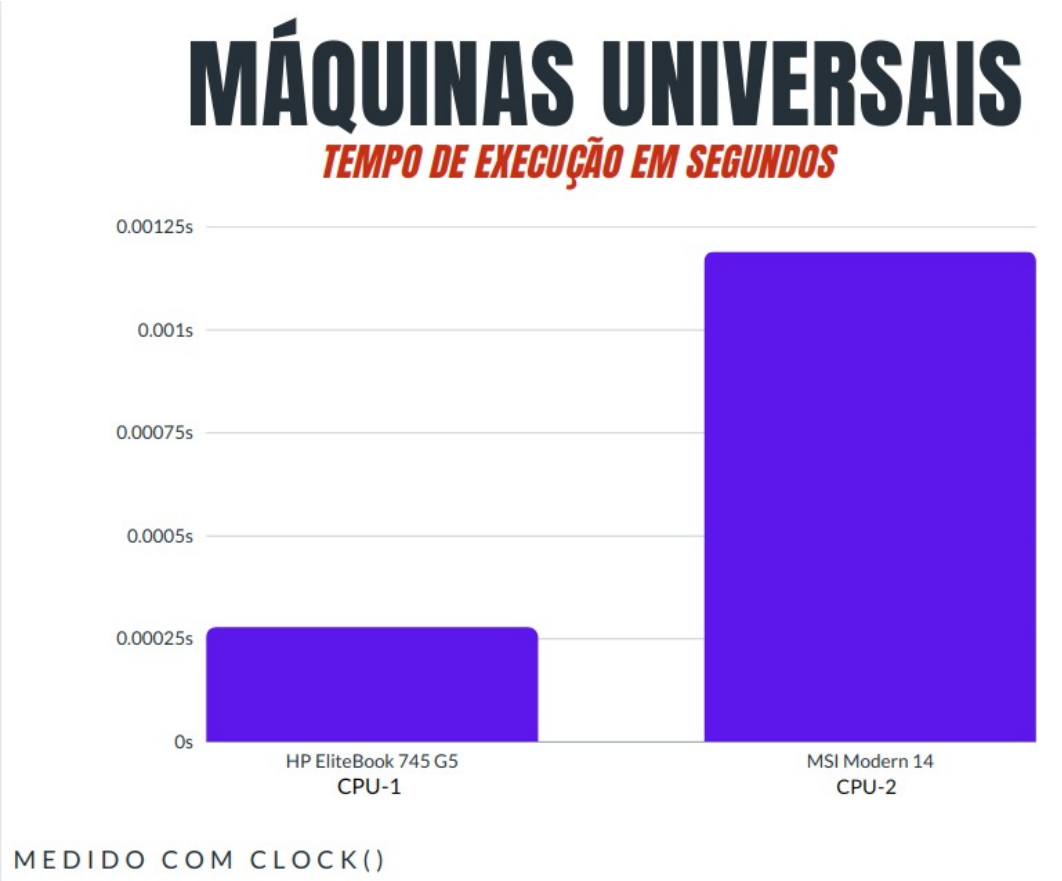

Figura 2: Tempo estimado médio de execução nas CPU's 1 e 2.

4 Calculo de complexidade

É importante ressaltar que a análise de complexidade nesse código depende de qual caso foi usado e o número de instruçoe executados durante a simulação. Para cada instrução excecutada, vai haver chacagem da cache, acesso de memória, e possível mudança de cache, que variam baseados no tamanho da hierarquia de memória e nos padrões de acesso específicos.

No geral, podemos analisar a complexidade das funções da main:

1. 'executeInstruction': Depende do número de instruções do programa. No pior caso, a complexidade seria O(n), onde n é o número de instruções.

2. 'run': Essa função executa as instruções em um loop usando a 'executeInstruction'. Como mencionado, a complexidade é O(n) baseado no número de instruções.

3. 'printMemories': Essa função imprime a hierarquia de memória e seu conteúdo. A complexidade é O(n) onde n é o número de blocos na RAM.

4. 'generateRandomInstructions': A complexidade depende do número de instruções a serem gerados, que é proporcional ao imput ramSize. A complexidade é O(ramSize).

5. 'readInstructions': Essa função lê instruções de um arquivo e retorna elas. A complexidade depende do número de instruções no arquivo, que é proporcional ao imput 'n'. A complexidade é O(n).

5 Resultados

Por fim, o resultado obtido por meio do projeto é razoável. Encontramos muitos problemas durante a implementação do trabalho, alguns com bons resultados no final e outros nem tanto.

6 Considerações Finais

Ao fim do trabalho, adquirimos real conhecimento sobre o funcionamento de um computador, suas funcionalidades e limitações.

A coordenação da simulação de uma máquina em C, onde quem programa "não atua" no papel manual é uma experiência desafiadora, que foge dos padrões aos quais somos habituados. Faz-nos perceber a complexidade de programar sem o auxílio de ferramentas, tais como os operadores lógicos da linguagem e afins.

Adicionar uma nova cache ao programa, demonstrou-se de difícil implementaçã. Embora o conceito seja relativamente simples e, de certa forma padronizado, o resultado se tornou razoavelmente satisfatório após várias tentativas.

A paciência para estudar o código, discutir ideias, e dessa forma entender o proposto também foi essencial para o desenvolvimento do projeto.
O trabalho em questão agrega de diversas formas, pessoais e/ou técnicas.

Referências