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

Trabalho Prático I

BCC266 - Organização de Computadores

Vitor Oliveira Diniz
Maria Luiza Aragão
Jessica Machado
Professor: Pedro Silva

Ouro Preto
16 de janeiro de 2023

Sumário

Lista de Códigos Fonte

1 Introdução

A máquina universal, ideia proposta por Alan Turing em 1936, seria uma máquina capaz de computar e executar qualquer máquina computável, vindo a ser tomada como um dos modelos abstratos do computador. Uma máquina abstrata de Turing é constituída de três partes: fita (usada simultaneamente como dispositivo de entrada, saída e memória de armazenamento), unidade de controle (reflete o estado corrente da máquina e possui uma unidade de leitura e gravação) e função de transição (que define o estado da máquina e comanda a leitura, gravação e o sentido do movimento da cabeça da fita).

1.1 Especificações do problema

Para a solução do problema, deveríamos implementar uma máquina em C que fosse capaz de interpretar instruções muito simples, tais como: somar,subtrair, levar dados para a memória,trazer dados da memória e parar (essas já implementadas e apenas necessitando de nossa compreensão). Com esse conhecimento adquirido, deveríamos implementar algumas outras instruções: multiplicação, potenciação, divisão e módulo da divisão,de forma eficiente e sem vazamentos de memória.

1.2 Considerações Iniciais

Algumas ferramentas foram utilizadas durante a criação deste projeto:

- Ambiente de desenvolvimento do código fonte: Visual Studio Code.
- Linguagem utilizada: C.
- Ambiente de desenvolvimento da documentação: Visual Studio Code LaTeXWorkshop.

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: Ryzen 7-5800H.
- Memória RAM: 16 Gb.
- Sistema Operacional: Arch Linux x86_64.

1.5 Instruções de compilação e execução

Para a compilação do projeto, basta digitar:

Compilando o projeto

```
gcc main.c -c -Wall -g
gcc cpu.c -c -Wall -g
gcc generator.c -c -Wall -g
gcc main.o cpu.o generator.o -o exe -g
```

Usou-se para a compilação as seguintes opções:

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

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

- -*c*: Para compilar o arquivo sem linkar os arquivos para obtermos um arquivo do tipo objeto.

- -*o*: Compilar para um arquivo do tipo output (saída).

Para a execução do programa basta digitar um dos exemplos:

```
./exe example [TAMANHO DA RAM]
./exe randomMultiplication [TAMANHO DA RAM]
./exe randomPotentiation [TAMANHO DA RAM]
./exe randomDivision [TAMANHO DA RAM]
./exe randomMod [TAMANHO DA RAM]
```

2 Desenvolvimento

Seguindo as boas práticas de programação, implementamos um tipo abstrato de dados (TAD) para a representação da nossa máquina universal. Iniciando o trabalho a partir código inicial que foi fornecido pelo professor, com o funcionamento básico e algumas operações, iremos realziar a implementação de pelo menos duas instruções baseadas nas quatro básicas e uma baseada nas que criei ou na de multiplicação.

2.1 Operações

A seguir entraremos em detalhe sobre as principais funções utilizadas no programa.

2.1.1 generateMultiplicationInstructions

A função de multiplicar vai funcionar basicamente como somas consecutivas. Inicialmente devemos salvar o valor do número que será somado consecutivamente na RAM e o elemento neutro da soma, que é 0. Assim podemos começar a somar n vezes, até que atinjamos o resultado da multiplicação. Para isso teremos 2 op codes fixos no início que levarão um valor para a RAM e um opcode no final que encerrará o programa.

```
1
2       Instruction* generateMultiplicationInstructions(int multiplier, int
            multiplying){
3           Instruction* instructions = (Instruction*) malloc((3 + multiplier) *
                sizeof(Instruction));
4           //Tres instrucoes extras
5               //1 - Salvar o multiplier na memoria
6               //2- Colocando 0 na posicao no resultado na RAM
7           instructions[0].opcode = 0;
8           instructions[0].info1 = multiplying; //Conteudo a ser salvo na RAM
9           instructions[0].info2 = 0; //Posicao da RAM
10
11          instructions[1].opcode = 0;
12          instructions[1].info1 = 0; //Coloca 0 (elemento neutro da soma)
13          instructions[1].info2 = 1; //Posicao da RAM
14
15          for (int i = 0; i < multiplier; i++){
16              instructions[i+2].opcode = 1; //Opcode para soma
17              instructions[i+2].info1 = 0; //Posicao do multiplying
18              instructions[i+2].info2 = 1; //Posicao do resultado da
                    multiplicacao
19              instructions[i+2].info3 = 1; //Posicao do resultado da
                    multiplicacao
20          }
21
22          //Inserindo a ultima instrucao do programa que nao faz nada que presta
23          instructions[multiplier+2].opcode = -1;
24          instructions[multiplier+2].info1 = -1;
25          instructions[multiplier+2].info2 = -1;
26          instructions[multiplier+2].info3 = -1;
27
28          return instructions;
29      }
```

Código 1: Função generateMultiplicationInstructions

2.1.2 generateDivisionInstructions

Para a divisão, devemos fazer sucessivas subtrações, não podendo deixar o valor da subtração ser menor que o divisor e a cada subtração sucessiva somar 1 ao resultado. Assim teremos o valor da divisão, levando em conta que estaremos trabalhando apenas com divisões de inteiros. Assim como na multiplicação, temos 3 op codes fixos, 2 para levar o dividendo e o divisor até a RAM e um para encerrar a lista de instruções.

```
1
2   Instruction* generateDivisionInstructions(int dividend, int divisor){
3
4
5       Instruction* instructions = (Instruction*) malloc((4 + dividend/
            divisor) * sizeof(Instruction));
6
7       instructions[0].opcode = 0;
8       instructions[0].info1 = dividend; //Conteudo a ser salvo na RAM
9       instructions[0].info2 = 0; //Posicao da RAM
10
11      instructions[1].opcode = 0;
12      instructions[1].info1 = divisor; //Conteudo a ser salvo na RAM
13      instructions[1].info2 = 1; //Posicao da RAM
14
15
16      int result = 0; // sempre somando 1 ao resultado, para saber quando
            parar, tambem serve como um indice para manter a contagem
17
18
19      // uma sugestao para o futuro, seria adicionar uma operacao de soma, e
             um valor inicial de 0 em alguma
20      // posicao da RAM para fazer uma contagem mais interessante do
            resultado, e nao depender da variavel
21
22
23      //loop enquanto der para dividir o numero
24      while ((result + 1) * divisor <= dividend){
25
26          instructions[result + 2].opcode = 2; //Opcode para subtracao
27          instructions[result + 2].info1 = 0; //Posicao do dividendo
28          instructions[result + 2].info2 = 1; //Posicao do divisor
29          instructions[result + 2].info3 = 0; //Posicao do resultado da
                divisao
30          result++;
31      }
32
33
34      //salvando o resultado na RAM
35      instructions[result + 2].opcode = 0;
36      instructions[result + 2].info1 = result;    //Contudo a ser salvo na
            RAM
37      instructions[result + 2].info2 = 2;         //Posicao da RAM
38
39
40
41      //inserindo a ultima instrucao do programa que nao faz nada que presta
42
43      instructions[result + 3].opcode = -1;
44      instructions[result + 3].info1 = -1;
45      instructions[result + 3].info2 = -1;
46      instructions[result + 3].info3 = -1;
47
48      return instructions;
```

```
49      }
```

Código 2: Função generateDivisionInstructions

2.1.3 generateModInstructions

O módulo, ou resto, da divisão é praticamente igual a divisão, tendo apenas a diferença que não precisamos armazenar o resultado final. Caso não pudermos mais realizar substrações sucessivas de modo que o dividendo seja maior que o divisor, encerramemos a operação e o módulo será o dividendo atual.

```
1       Instruction* generateModInstructions(int dividend, int divisor){
2
3
4           Instruction* instructions = (Instruction*) malloc((4 + dividend/
                divisor) * sizeof(Instruction));
5
6           instructions[0].opcode = 0;
7           instructions[0].info1 = dividend; //Conteudo a ser salvo na RAM
8           instructions[0].info2 = 0; //Posicao da RAM
9
10          instructions[1].opcode = 0;
11          instructions[1].info1 = divisor; //Conteudo a ser salvo na RAM
12          instructions[1].info2 = 1; //Posicao da RAM
13
14
15
16          // sempre somando 1 ao resultado, para saber quando parar, tambem
                serve como um indice para manter a contagem
17          int result = 0;
18
19          //loop enquanto der para dividir o numero
20          while ((result + 1) * divisor <= dividend){
21
22              instructions[result + 2].opcode = 2; //Opcode para subtracao
23              instructions[result + 2].info1 = 0; //Posicao do dividendo
24              instructions[result + 2].info2 = 1; //Posicao do divisor
25              instructions[result + 2].info3 = 0; //Posicao do resultado da
                    divisao
26              result++;
27          }
28
29
30          //inserindo a ultima instrucao do programa que nao faz nada que presta
31
32          instructions[result + 2].opcode = -1;
33          instructions[result + 2].info1 = -1;
34          instructions[result + 2].info2 = -1;
35          instructions[result + 2].info3 = -1;
36
37          return instructions;
38      }
```

Código 3: Funçao generateModInstructions

2.1.4 generatePotentiationInstructions

Para a potenciação, temos uma operação um pouco mais complexa, inicialmente sabemos que a potenciação é a multiplicação do número da base n vezes. Assim podemos alocar uma matriz de instruções em que cada linha corresponderá a uma multiplicacao. Temos algumas nuances no código e tentarei dar uma breve explicação.

Como teremos n-1 multiplicações, a alocação das linhas da nossa matriz terá n-1 linhas, e as colunas serão alocadas pela própria função da multiplicação.

Nossa multiplicação inicial será sempre base x base, por isso a posição 0 da nossa matriz será essa multiplicação. A partir da primeira multiplicação teremos que modificar a memória para modificarmos a multiplicação inicial de 3 para o resultado da multiplicação anterior. Assim utilizaremos a função cópida de memória RAM, e o resultado anteriormente armazenado em uma posição da RAM, agora será a base da nossa multiplicação. Seguiremos essa operação n-1 vezes. Como fazemos essa manipulação da memória RAM, o segundo parâmetro da função de gerar as instruções de multiplicação é responsável apenas pelo valor inicial, então podemos passar qualquer parâmetro que não fará diferença.

A partir daí sempre geraremos o número de somas correspondente a base, e depois disso devemos tanto passar o resultado atual para a posição do número a ser multiplicado, quando zerar o elemento neutro da soma, pois nossa soma se baseia nele.

Depois disso liberaremos a matriz de instruções que não é utilizada mais e retornaremos o vetor final de instruções que foi formado a partir da concatenação dos vetores das linhas da matriz.

```
Instruction* generatePotentiationInstructions(int base, int exponent){


    //matriz de instrucoes, vulgo vetor de vetor, porque a potenciacao sao varias multiplicacoes.
    //cada linha da matriz representa uma instrucao de uma multiplicacao
    Instruction **instructionMatrix;
    Instruction *instructions;


    //alocacao da matriz de instrucoes, que em a^b sempre teremos b - 1 linhas
    instructionMatrix = (Instruction**) malloc( (exponent - 1) * sizeof(Instruction));


    instructionMatrix[0] = generateMultiplicationInstructions(base, base);

    // loop de cada linha de instrucao para gerar elas
    for ( int i = 1; i < exponent - 1; i++){

        // nas proximas iteracoes, o valor inicial da multiplicacao ja esta na memoria RAM, isso desde que
        //nao esquecamos de substituir os comandos de salvar valor na memoria ( opcode = 0), por isso nao fara diferenca
        // o segundo parametro da funcao.
        instructionMatrix[i] = generateMultiplicationInstructions(base , 0);
    }


    // alocaco das instrucoes finais que serao retornadas no fim da funcao.
    // a alocacao se da por
    //4 + base * (exponent - 1) + 2 * (exponent - 2))  * sizeof(Instruction))
    // em que temos 3 operacoes que sempre vao aparecer, o 0 0 nas duas primeiras instrucoes
    // e -1 para indicar o fim das instrucoes
    // base * (exponent -1) e a quantidade de somas ( opcode 1) que podemos encontrar no meio da matriz.
    // essas serao as quantidades de somas necessarias
    // 2 * ( exponent - 2 ) serao o numeros de operacoes auxiliares ( opcode 3 e 0) para a manipulacao necessaria da Ram
```

```
34
35
36      instructions = (Instruction*) malloc((3 + base * (exponent - 1) + 2 * (
            exponent - 2))  * sizeof(Instruction));
37
38      instructions[0].opcode = 0;
39      instructions[0].info1 = base; //Conteudo a ser salvo na RAM
40      instructions[0].info2 = 0; //Posicao da RAM
41
42      instructions[1].opcode = 0;
43      instructions[1].info1 = 0; //Coloca 0 (elemento neutro da soma)
44      instructions[1].info2 = 1; //Posicao da RAM
45
46
47      int index_counter = 2; // INDICE DO VETOR DE INSTRUCOES FINAL
48
49
50      //loop para percorrer a matriz de isntrucoes e copiar o valor das
            instrucoes para nosso vetor final
51      for (int i = 0; i < exponent -1;i++){
52          for ( int j = 2; j < 3 + base - 1; j++){
53
54              instructions[index_counter].opcode = instructionMatrix[i][j].
                    opcode;
55              instructions[index_counter].info1 = instructionMatrix[i][j].info1;
56              instructions[index_counter].info2 = instructionMatrix[i][j].info2;
57              instructions[index_counter].info3 = instructionMatrix[i][j].info3;
58              index_counter++;
59
60          }
61          // como no final de cada linha devemos ter as operacoes auxiliares
                para modificar
62          // valores na RAM, precisamos inserilas depois de cada sequencia de
                soma
63          //exceto se for a ultima linha da matriz, assim
64          // nao devemos inserir nem o 3 nem o 0
65          if (i >= exponent - 2){
66              {};
67          } else {
68
69              instructions[index_counter].opcode = 3;
70              instructions[index_counter].info1 = 1;
71              instructions[index_counter].info2 = 0;
72              index_counter++;
73
74              instructions[index_counter].opcode = 0;
75              instructions[index_counter].info1 = 0;
76              instructions[index_counter].info2 = 1;
77              index_counter++;
78          }
79
80
81      }
82
83
84      instructions[index_counter].opcode = -1;
85      instructions[index_counter].info1 = -1;
86      instructions[index_counter].info2 = -1;
87      instructions[index_counter].info3 = -1;
88
89
90      for (int i = 0; i < exponent - 1; i++){
```

```
91          free(instructionMatrix[i]);
92      }
93      free(instructionMatrix);
94
95
96      return instructions;
97  }
```

Código 4: Função generatePotentiationInstructions

2.1.5 OpCode 3

Para o correto funcionamento da nossa função de potenciação precisávamos copiar o valor de uma posição da RAM para outro, assim decidimos implementar o opcode da cópia de valores.

```
1
2       case 3: //Copiando informacao da RAM
3
4       address1 = instruction.info1;       //origem
5       address2 = instruction.info2;       //destino
6
7       RAMContent1 = machine->RAM.items[address1];
8       RAMContent2 = machine->RAM.items[address2];
9
10      machine->RAM.items[address2] = machine->RAM.items[address1];
11
12      printf("  > Copiando RAM[%d] (%f) para RAM[%d] (%f).\n",
13       address1, RAMContent1, address2, RAMContent2);
14
15      break;
```

Código 5: OpCode 3

2.2 Função Main

Na função main invocamos as funções necessárias para a realização dos procedimentos, sendo eles: a leitura dos dados da matriz, a sua alocação, seu tratamento, sua impressão e por último, desalocação.

Como um destaque especial, temos uma leve comparação de quais argumentos foram passados através do terminal, e assim invocar a operação pedida, além do tamanho total da memória. Temos simples funções que são possíveis de chamar para testar as operações implementados. As possíveis funções estão declarados em instruções de compilação e execução. As operações são todas realizadas com números aleatórios.

3 Impressões Gerais

Achamos esse trabalho prático bem difícil, boa parte porque não entendemos bem as instruções passadas no .pdf que o professor disponibilizou. Infelizmente o calendário acadêmico da UFOP não nos permitiu um bom aprendizado da matéria. Além disso, achamos que o .pdf deixou a desejar no quesito de instruções e orientações, o jeito que foi apresentado nos deixou com muita dificuldade de entender a atividade proposta. Algo a ser levado em conta foi o planejamento inicial, que definia que iríamos desenvolver aos poucos o trabalho durante a aula, mas infelizmente a ementa da matéria não entrou em detalhe de possíveis implementações, e sim apenas de conceitos básicos de organização de computadores, o que aumentou ainda mais a dificuldade na matéria. Dito isso, tivemos que correr atrás de bastantes conceitos e como implementar no período de recesso da UFOP, afinal o trabalho era bem complexo e complicado e o tempo dado quase não foi suficiente para terminar.

Nosso grupo se reuniu e pensou em quais instruções adicionais poderíamos incluir neste trabalho, bem como formas de implementá-las e a maneira mais simples de fazer. Ao final, passamos por muitas dificuldades e uma série de erros antes de conseguir finalizar esse trabalho, pois algumas vezes começamos tendo o raciocínio certo mas tivemos dificuldades em implementar da maneira que pensamos.

Para a documentação fizemos o uso do Latex, o que não foi tão difícil já que possuímos uma base por conta da documentação que fizemos para a matéria de ED1 e que nos deu uma visão introdutória do programa e seu funcionamento.

4 Análise

Primeiramente analisamos a fundo as instruções passadas no .pdf referente ao trabalho, o que nos demandou uma quantidade considerável de tempo, já que não estávamos conseguindo entender de uma forma clara alguns conceitos da atividade. Após isso, passamos a decidir quais operações iríamos aplicar a nossa máquina, decidindo pela multiplicação, divisão, potenciação e módulo. Sendo a potenciação a que nos demandou mais tempo e raciocínio devido a sua complexidade.

Após isso, verificamos se haviam memory leaks devido a manipulação de memória utilizando o Valgrind (um software livre que auxilia o trabalho de depuração de programas), confirmando assim que a memória havia sido corretamente liberada.

5 Conclusão

Com este trabalho pudemos compreender o conceito de uma máquina universal e suas instruções, assim como o uso da RAM e a utilização de sua memória. Simulamos assim, o funcionamento de um processador e sua forma de receber instruções. Também aprendemos o significado de alguns conceitos anteriormente passados em aula, tais como: o pipeline (uma fila de instruções que será passada para o processador) e o overflow (que ocorre quando uma operação aritmética tenta criar um valor numérico que está fora do intervalo que pode ser representado com um determinado número de dígitos).

Entendemos também como são implementados e sua importância dentro de um programa.