Atividade do Capítulo 6

Nome: Mateus Sousa Araújo – **Matrícula:** 374858
Nome: José Wesley Araújo – **Matrícula:** 374855

No capítulo 6 foi pedido para realizar a construção de um compilador do computador hack em uma linguagem de programação aleatória. Fizemos a construção deste compilador em C.

Primeiramente o objetivo desse trabalho é realizar a transformação das instruções do hack em um arquivo txt ou algo semelhante em binário. Para isso foi utilizado as ferramentas disponibilizadas pelo livro como o Assembler para a execução dos binários.

Foi pedido primeiramente para que espaços e comentários fossem retirados do arquivo onde as instruções estão contidas. Fizemos um arquivo chamado compilador.c que faz esse papel.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>

void rm_lineWhite(FILE *p,FILE *write ){
        int i = 0 ;
        int c = 0;
        char ante, atual;
        ante = fgetc(p);
        if (ante == '\n'){
            ante = fgetc(p);
        }
        while(  (atual = fgetc(p) )!= EOF) {
              if ((ante == atual) && (atual == '\n')){
            }else{
                if (((i == 0)) && (ante == '\n')){}
                else fprintf(write,"%c", ante);
            }
            ante = atual;
            i++;
        }
        fprintf(write,"%c",ante);

}
```

O trecho de código acima faz a retirada dos espaços em branco em cada linha vazia. Foi criado variáveis atuais e anteriores para fazer a concatenação adequada do arquivo final que será gerado. A ideia é sempre ir deslocando os espaços em brancos até encontrar um caractere válido. A variável anterior (“ante”) vai recebendo valores onde atual é sempre “\n”. Dessa forma a função acima varre todo o arquivo e atualiza os valores no final escrevendo apenas o que interessa.

```
void rm_ultimo(FILE *p,FILE *write){

        char atual;
        atual = fgetc(p);
        if(atual == '\n') atual = fgetc(p);
        fprintf(write,"%c",atual );
        while(  (atual = fgetc(p) )!= EOF) {
            fprintf(write,"%c", atual);
        }
}
```

O trecho de código acima faz uma verificação e escrita no último caractere referente a quebra de linha. Quando atual é igual a “\n” então atual é escrito no arquivo, deixando apenas as linhas uma a uma.

```
void rm_space(FILE *p,FILE *write){
    char atual;

    while(  (atual = fgetc(p) )!= EOF) {
        if ( (atual != ' ') && ((atual ^ 9) != 0) ){
            atual = toupper(atual);
            fprintf(write,"%c", atual);
        }
    }
}
```

Acima está uma função que faz as instruções ficarem maiúsculas no arquivo final. Se atual assumir linhas válidas então os caracteres serão colocados em maiúsculo pela função “toupper”. No final é escrito no arquivo os caracteres maiúsculos.

```
char rm_comment(FILE *P, FILE *W){
    int i =0;
    char ante;
    int c = 0;
    char  atual;
    ante = fgetc(P);
    while(  (atual = fgetc(P) )!= EOF) {
        if ( (c == 0)){
             i = 0;
        }
        if ( (atual == '/') && (ante == atual) ){  // se encontrar // i = 1 e não printa
            i = 1;
            c = 1;
        }
        else if ( i == 0 ){
                fprintf(W,"%c",ante);
            }
            ante = atual;
            if ( atual == '\n'){
                 c = 0;
            }


    }
}
```

Acima está a parte da retirada dos comentários no arquivo. Fizemos algumas variáveis de controle como “c” e “i”para fazer ou não a escrita de acordo com as condições estabelecidas. A ideia é sempre escrever no arquivo enquanto não encontrar duas barras invertidas. Quando “i” é igual a zero então quer dizer que duas barras foram encontradas e que agora será escrito no arquivo a parte que realmente interessa.

```
int main(void){
    FILE *read;
    FILE *write;
    char nomeArquivo[200];
    char nomeTEMP[]   = "/tmp/arqXXXXXX";
    int fd = mkstemp(nomeTEMP);
    close(fd);
//#########################################################################
    //         PRIMEIRO REMOVEMOS OS COMENTÁRIOS, ESCREVEMOS O RESULTADO TEMPORARIO INSTRUCOES.TXT
//#########################################################################T##########
    printf("Diga o nome do seu arquivo : \n");
    scanf("%[^\n]",nomeArquivo);
    printf("Seu arquivo será salvo em notSpace.txt.\n");
    read = fopen(nomeArquivo,"r");

    write = fopen(nomeTEMP,"w");
    rm_comment(read, write);
    fclose(write);
    fclose(read);


    read = fopen(nomeTEMP,"r");
    write = fopen("notSpace.txt","w");
    rm_space(read,write);
    fclose(write);
    fclose(read);
```

```
    read  = fopen("notSpace.txt","r");
    write = fopen(nomeTEMP,"w");
    rm_lineWhite(read,write);
    fclose(write);
    fclose(read);
    read  = fopen(nomeTEMP,"r");
    write = fopen("notSpace.txt","w");
    rm_ultimo(read,write);
    fclose(write);
    fclose(read);
    unlink(nomeTEMP);

    return 0;
}
```

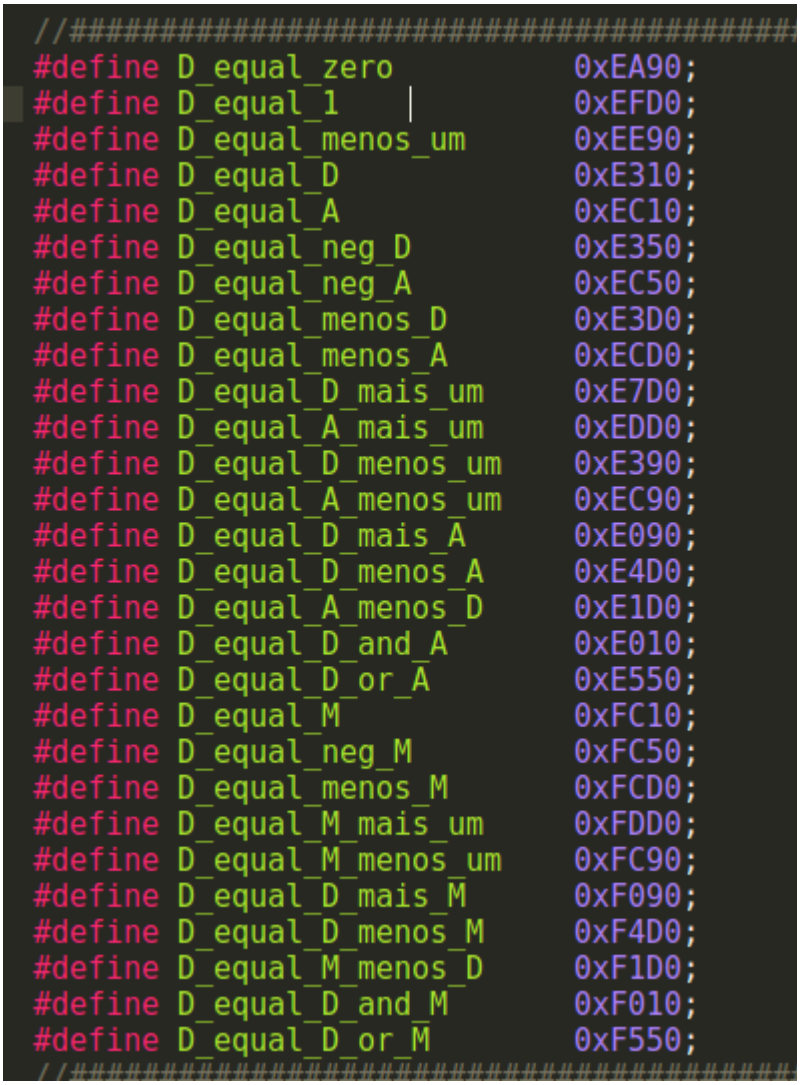
Os códigos acima se encontram no mesmo arquivo e faz a chamada e leitura de arquivos pré-temporários que utilizamos para a realização de testes durante a construção do programa. A ideia foi chamar as funções acimas descritas aos poucos para não executar tudo de uma vez.

A seguir, no arquivo gera_obj.c fizemos o mapeamento das instruções do tipo A, M e D. Fizemos todos os casos possíveis. Deu ao todos 84 instruções. Fizemos uma macro para a definição de cada instrução em hexa. Depois fomos fazendo as chamadas dessas macros de acordo com cada instrução que ia sendo trabalhada.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define A_equal_zero          0xEAA0;
#define A_equal_1             0xEFE0;
#define A_equal_menos_um      0xEEA0;
#define A_equal_D             0xE320;
#define A_equal_A             0xEC20;
#define A_equal_neg_D         0xE360;
#define A_equal_neg_A         0xEC60;
#define A_equal_menos_D       0xE3E0;
#define A_equal_menos_A       0xECE0;
#define A_equal_D_mais_um     0xE7E0;
#define A_equal_A_mais_um     0xEDE0;
#define A_equal_D_menos_um    0xE3A0;
#define A_equal_A_menos_um    0xECA0;
#define A_equal_D_mais_A      0xE0A0;
#define A_equal_D_menos_A     0xE4E0;
#define A_equal_A_menos_D     0xE1E0;
#define A_equal_D_and_A       0xE020;
#define A_equal_D_or_A        0xE560;
#define A_equal_M             0xFC20;
#define A_equal_neg_M         0xFC60;
#define A_equal_menos_M       0xFCE0;
#define A_equal_M_mais_um     0xFDE0;
#define A_equal_M_menos_um    0xFCA0;
#define A_equal_D_mais_M      0xF0A0;
#define A_equal_D_menos_M     0xF4E0;
#define A_equal_M_menos_D     0xF1E0;
#define A_equal_D_and_M       0xF020;
#define A_equal_D_or_M        0xF560;
```

Instruções do tipo A



```
//#########################################
#define D_equal_zero          0xEA90;
#define D_equal_1             0xEFD0;
#define D_equal_menos_um      0xEE90;
#define D_equal_D             0xE310;
#define D_equal_A             0xEC10;
#define D_equal_neg_D         0xE350;
#define D_equal_neg_A         0xEC50;
#define D_equal_menos_D       0xE3D0;
#define D_equal_menos_A       0xECD0;
#define D_equal_D_mais_um     0xE7D0;
#define D_equal_A_mais_um     0xEDD0;
#define D_equal_D_menos_um    0xE390;
#define D_equal_A_menos_um    0xEC90;
#define D_equal_D_mais_A      0xE090;
#define D_equal_D_menos_A     0xE4D0;
#define D_equal_A_menos_D     0xE1D0;
#define D_equal_D_and_A       0xE010;
#define D_equal_D_or_A        0xE550;
#define D_equal_M             0xFC10;
#define D_equal_neg_M         0xFC50;
#define D_equal_menos_M       0xFCD0;
#define D_equal_M_mais_um     0xFDD0;
#define D_equal_M_menos_um    0xFC90;
#define D_equal_D_mais_M      0xF090;
#define D_equal_D_menos_M     0xF4D0;
#define D_equal_M_menos_D     0xF1D0;
#define D_equal_D_and_M       0xF010;
#define D_equal_D_or_M        0xF550;
//#########################################
```

Instruções do tipo D

```
#define M_equal_zero        0xEA88;
#define M_equal_1           0xEFC8;
#define M_equal_menos_um    0xEE88;
#define M_equal_D           0xE308;
#define M_equal_A           0xEC08;
#define M_equal_neg_D       0xE348;
#define M_equal_neg_A       0xEC48;
#define M_equal_menos_D     0xE3C8;
#define M_equal_menos_A     0xECC8;
#define M_equal_D_mais_um   0xE7C8;
#define M_equal_A_mais_um   0xEDC8;
#define M_equal_D_menos_um    0xE388;
#define M_equal_A_menos_um    0xEC88;
#define M_equal_D_mais_A      0xE088;
#define M_equal_D_menos_A     0xE4C8;
#define M_equal_A_menos_D     0xE1C8;
#define M_equal_D_and_A       0xE008;
#define M_equal_D_or_A        0xE548;
#define M_equal_M             0xFC08;
#define M_equal_neg_M         0xFC48;
#define M_equal_menos_M       0xFCC8;
#define M_equal_M_mais_um     0xFDC8;
#define M_equal_M_menos_um    0xFC88;
#define M_equal_D_mais_M      0xF088;
#define M_equal_D_menos_M     0xF4C8;
#define M_equal_M_menos_D     0xF1C8;
#define M_equal_D_and_M       0xF008;
#define M_equal_D_or_M        0xF548;
```

Instruções do tipo M

```
void to_binario(char binario[],int num){
    int mask = 0x8000;
    if(mask &  num){
        strcpy(binario,"1");
        }
        else{
         strcpy(binario,"0");
        }
        mask = mask >> 1;
    while(mask !=0 ){
        if(mask &  num){
        strcat(binario,"1");
        }
        else{
         strcat(binario,"0");
        }
        mask = mask >> 1;
    }
}
```

Acima foi criada uma função para a conversão de hexa em binário, resultado esse que será posteriormente escrito no arquivo final em binário.

```
void cmp_variable(FILE *r, FILE *e,FILE *w, int i, char atual, int *valor_linha){
        char binario
         [16];
        int pos = 0;
        char igual = '=';
        char variavel [150];
        variavel[0]=atual;
        while(atual != '\n'){
            pos++;
            atual = fgetc(r);
            variavel[pos]=atual;
        }

             fprintf(e, "%s", variavel);
            // fprintf(e, "%d",*valor_linha );
             to_binario(binario,*valor_linha);
             fprintf(w, "%s\n", binario);
              *valor_linha  += 1;
        return;
}
//#############################################################################
```

A função acima faz uma comparação de variável para passar para binário somente aquilo que interessa depois do sinal de igual.

```
int lerNumero(FILE *r,FILE *w ,int i,char num){
        char binario[16];
        char atual;
        int potencia = 0;
        int instrucao = 0;
        int controle = 0;
        int numero;
        int res = 1; // res recebe resultado caso der erro em @

        atual = num;
        while((atual != '\n') && (controle == 0)){
                if     ( atual == '1') {instrucao = instrucao*10 + 1;}
                else if( atual == '2') {instrucao = instrucao*10 + 2;}
                else if( atual == '3') {instrucao = instrucao*10 + 3;}
                else if( atual == '4') {instrucao = instrucao*10 + 4;}
                else if( atual == '5') {instrucao = instrucao*10 + 5;}
                else if( atual == '6') {instrucao = instrucao*10 + 6;}
                else if( atual == '7') {instrucao = instrucao*10 + 7;}
                else if( atual == '8') {instrucao = instrucao*10 + 8;}
                else if( atual == '9') {instrucao = instrucao*0  + 9;}
                else if( atual == '0') {instrucao = instrucao*10 + 0;
                }else {
                    printf("erro na linha : %d \n", i);
                    return 0;
                }
                atual = fgetc(r);
                if(atual == '\n') {
                    controle = 1;
                }
                potencia++;
        }
        if(instrucao > 0x7fff ){
            printf("Dumped value max line %d\n",i);
            exit(0);
        }
        to_binario(binario,instrucao);
```

A função acima faz a conversão do caractere depois do @ para binário. Também foi feito o controle caso o número seguido de @ ultrapasse o limite permitido que é “0xFFFF”. É dado uma mensagem para o usuário caso isso venha a acontecer.

```
void erro(int i){
 printf("Erro na linha %d\n",i);
 exit(0);
}
```

Considerado como uma das funções mais importantes do compilador, o código acima faz o print de erro em linha x ao decorrer do programa.

```
int main(){

    FILE *e = fopen("so_varia.txt", "w");
    int valor_linha = 16;
    char binario[16];   // armazena string binaria no arquivo
    int i = 0 ;  // diz   a linha de execucao
    char atual;  // ultimo caractere lido do arquivo
    int instrucao = 0; // instrucao binaria
    int res = 1; // res recebe 0 caso der erro no @
    FILE *r,*w;
    r = fopen("instrucoes.txt","r");
    w = fopen("instrucoes.hack", "w");
    int t = 0;
```

Acima temos o início do compilador. É pego o arquivo de interesse que foi removido todos os espaços e comentários e faz a sua leitura. Em diante fizemos tudo na mão com diversos casos possíveis para os 3 tipos de instrução. A ideia foi acompanhar o esquemático do livro.

comp (when a=0)	c1	c2	c3	c4	c5	c6	*comp* (when a=1)
0	1	0	1	0	1	0	
1	1	1	1	1	1	1	
-1	1	1	1	0	1	0	
D	0	0	1	1	0	0	
A	1	1	0	0	0	0	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0	0	1	0	D+M
D-A	0	1	0	0	1	1	D-M
A-D	0	0	0	1	1	1	M-D
D&A	0	0	0	0	0	0	D&M
D|A	0	1	0	1	0	1	D|M

A tabela acima nos guia em todos os possíveis casos para cada instrução. Foi bem repetitivo o que fizemos depois. Analisamos todas as possibilidades paras os 3 tipos de instruções e fomos comparando com os caracteres do arquivo de entrada. Fizemos a instrução A, B e D para +, -, &, |, !. Fizemos todos os 84 casos possíveis para a tabela acima. O decorrer do código visa a verificação de todas as possibilidades para o A, M e D. O código será posto em anexo para verificação.

```
while(((atual = fgetc(r)) != EOF) && (t == 0)){
    i++;
    switch(atual){
        case  'A':
            if(((atual = fgetc(r)) == EOF) || atual == '\n'){
                erro(i);
            }else if(atual == '='){
                if(((atual = fgetc(r)) == EOF)|| atual == '\n'){
                    erro(i);
                    t =1;
                }else if (atual == '0'){
                    atual = fgetc(r);
                    if(atual == '\n'){
                        instrucao = A_equal_zero;
                        to_binario(binario,instrucao);
                        fprintf(w, "%s",binario);
                        fprintf(w, "%c",atual);
                    }else{
                        erro(i);
                    }
                }else if (atual == '1'){
                    atual = fgetc(r);
                    if(atual == '\n'){
                        instrucao = A_equal_1;
                        to_binario(binario,instrucao);
                        fprintf(w, "%s",binario);
                        fprintf(w, "%c",atual);
                    }else{
                        erro(i);
                    }
```

Pegamos um exemplo do nosso código acima para explicar. Se o atual do arquivo receber uma instrução do tipo A, então devemos tratar todos os casos possíveis. Podemos ter A = a muitos tipos de possibilidades. Fomos listando isso analisando caractere por caractere. Se o A for seguido de igual, então fazemos a análise. Se A = 0, por exemplo, então chamamos a macro de define com a respectiva instrução em hexa. Depois chamamos a função para conversão em binário e depois printamos aquela instrução respectiva em um arquivo final.