MONTARILO V3.0: MONTADOR PARA 8085

Murilo Marques Marinho
murilomarinho@lara.unb.br

Resumo Este texto descreve a criação de um montador para 8085 usando ANSI C.
Palavras Chaves: 8085, montador, ANSI C, C, ELF
Abstract: This text is the description of the implementation of an 8085 assembler coded in ANSI C.
Keywords: 8085, assembler, ANSI C, C, ELF

1 INTRODUÇÃO

Esse texto trata sobre a implementação de um montador para o 8085 usando ANSI C. Para isso, foram criadas diversas estruturas de dados e métodos que se resumem a fazer essa tarefa. Em uma primeira visão, o trabalho não apresenta nada de especial mas seus métodos de implementação são diferenciados e mostram bons resultados até em fase inicial.

2 METODOLOGIA DA MONTAGEM

A seguir são descritos todos os passos para a montagem de um programa escrito em Assembly 8085 e montado com o Montarilo.

2.1 Preprocessamento

Inicialmente o programa realiza o preprocessamento que consiste em limpar código de todos os dados irrelevantes para a montagem. Isto é, a identação, linhas em branco, espaços a mais, comentários. O arquivo de entrada é selecionado como um dos argumentos da chamada do programa. Assim, com a função ***preProcess()*** o programa:

- Elimina espaços em branco;
- Elimina linhas em branco;
- Elimina comentários;
- Elimina espaços no início e no final de linhas;
- Reduz qualquer espaçamento maior que 1 para 1;
- Elimina combinações: linhas apenas com espaços e comentários.

O programa lê o arquivo de entrada escolhido pelo usuário e imprime uma versão preprocessada em outro arquivo, chamado temp.txt, que por default será eliminado após o uso, mas em outra função. Assim que a função termina ela retorna um ponteiro para esse novo arquivo.

2.2 Análise Léxica

A análise léxica consiste em criar a TokenTable, que seria o equivalente do código escrito transformado em uma lista encadeada de cadeias de Tokens. A estrutura de dados de uma TokenTable é como ilustrada abaixo:

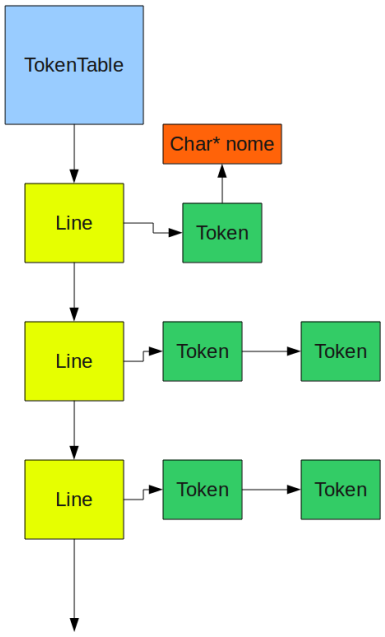

Figure 1 Estrutura de uma TokenTable

TokenTable representa a raiz dessa tabela de Tokens, cuja única finalidade é apontar para a primeira Line. Cada Line aponta para uma Token, e cada Token aponta para a próxima, se houver, e possui uma String que armazena sua representação, seu nome. Para que ela seja criada, usamos a função ***lexicAnalise()*** que tem como entrada um ponteiro para essa TokenTable que será criada, e o arquivo que armazena o código. Lendo linha a linha, essa função inicialmente quebra o código em Strings onde cada uma representa uma linha, as Strings que serão entrada para a função ***tokenMaker()***. Na função ***tokenMaker()*** a String representativa da linha será quebrada em Tokens. A análise de uma Token consiste em percorrer a linha até encontrar um dos caracteres de parada. Caracteres esses que podem ser: dois pontos, ponto e vírgula, espaço, fim de linha. A função então aponta o nome da Token para essa nova String e prossegue na linha até que ela acabe. Chegando no fim da linha ela retorna para a função ***lexicAnalise()*** que seguirá para a próxima linha e esse processo se repete. Quando acabam as linhas do programa, a função retorna e a TokenTable está completa.

2.3 Montagem: Considerações Iniciais

Para realizar a montagem, o programa faz uso de várias estruturas de dados. Ele usará uma MCTable, uma DirTable, uma SimbolTable, uma UseTable e uma DefTable. Cada uma será

explicada, e como ela é formada, para que o entendimento da montagem seja pleno.

2.3.1 MCTable

A MCTable é o equivalente da região .text da TokenTable escrita em código de máquina. A função da MCTable é guardar toda a tradução feita pelo programa e depois ser impressa em forma de código de máquina. A estrutura de uma MCTable pode ser visualizada no diagrama de blocos abaixo:

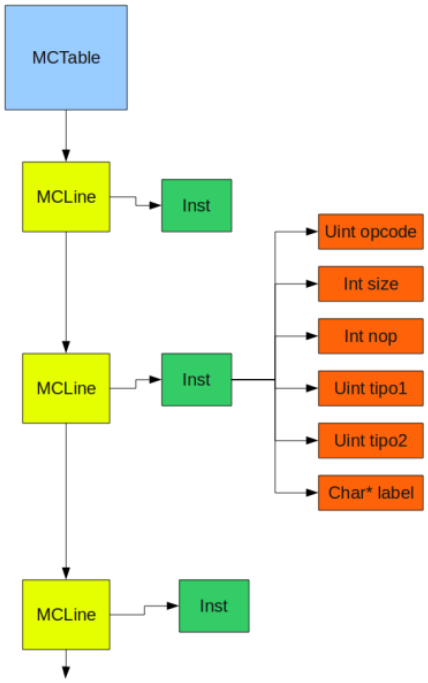

Figure 2- Estrutura de uma MCTable

Ela é constituída da raíz, MCTable, que aponta para a primeira MCLine. Cada MCLine aponta para a próxima MCLine, se houver, e para uma Inst, estrutura de uma instrução. A estrutura de uma instrução possui:

- o int opcode, que guarda o opcode da função;
- int tipo1 que guarda as flags de tipo do primeiro argumento;
- int tipo2 que guarda as flags de tipo do segundo argumento;
- int size que guarda o tamanho da instrução, em bytes;
- int nop que guarda o número de argumentos da função;
- char* label que guarda uma possível label que aponte para essa linha.

A criação das instruções é o processo mais complexo e interessante desse programa, e para entendê-lo é necessário explicar a utilização de cada uma das flags de tipo, que podem ser básicas ou de exceção. As flags de tipo indicam o tipo aceitável de cada argumento. Elas não se sobrepõem e permitem grande flexibilidade para a instruções, por mais que o instruction set do 8085 não mudará. São elas que permitem que a instrução seja criada, e não simplesmente retirada de uma tabela. As flags de tipo básicas são:

- R_FLAG;
- RP_FLAG;
- I_FLAG;
- L_FLAG.

A R_FLAG é a responsável por definir que aquele argumento pode ser do tipo registrador simples, isto é, A, B, C, D, H, L ou M. Uma instrução com essa flag será modificada pela função ***addReg()***. Essa função trata de adicionar a o opcode do registrador na posição correta do byte de opcode, que pode ser em duas posições distintas nesse byte, a saber: nos bits [3,5] ou [6,8], dependendo da função e do número do argumento. Isso permite que instruções como MOV não precisem ter todos os opcodes das suas variações descritas em uma tabela, o programa se encarrega de criar o opcode conforme aparece a necessidade. Podemos descrever, então, a primeira flag de exceção. Algumas instruções, como a DCR e a INR, mudam a regra geral de criação do opcode para instruções que aceitam registrador simples como um, e apenas um, de seus operandos. Ao invés de o registrador ser inserido nos últimos 3 bits, ele é inserido nos bits [3,5]. O programa reconhece isso com a INV_FLAG que trata de inverter a localização do opcode do registrador. Uma outra exceção é a função LDHI que tem como entrada um dos registradores, mas seu opcode não muda. Isso é reconhecido pela flag RNC_FLAG, que permite à função deixar o opcode inalterado.

A RP_FLAG é a responsável por definir que aquele argumento pode ser do tipo par de registradores, isto é, B, D, H ou SP. Uma instrução com essa flag será modificada pela função ***addRegPair()***. Essa função trata de adicionar o opcode, de maneira semelhante ao descrito anteriormente, na posição [3,5]. Existem outras exceções que são tratadas por essa função. As instruções PUSH e POP não aceitam SP como argumento, e em seu lugar é o PSW. Assim, a PSW_FLAG lida em identificar que essas duas funções sofrem essa restrição. No caso das instruções STAX e LDAX, apenas os registradores B e D podem ser usados, o que fica identificado com a BD_FLAG.

A I_FLAG é a responsável por definir que aquele argumento pode ser do tipo imediato. Uma instrução com essa flag será modificada pela função ***addIm()***. Verifica se o imediato está escrito em hexadecimal verificando se o primeiro caractere é um número e se o último é um H. Caso não seja, tenta adicioná-lo como decimal. Essa função armazena também a exceção RST, que aceita números naturais de 1 a 7. Isso é verificado pela flag de exceção RST_FLAG. Caso a função não seja a exceção, seu opcode é deslocado em 1 byte para a esquerda e é adicionado o byte de imediato. Caso seja a exceção, recebe o valor específico do RST conforme a sintaxe da instrução.

A L_FLAG é responsável por definir que aquele argumento pode ser do tipo endereço. Uma instrução com essa flag será modificada pela função ***addLabel()***. Verifica se a Token de argumento inicia com número. Caso inicia, supõe que esse argumento é um endereço em hexadecimal. Caso contrário, considera que esse endereço é um simbolo que depois será buscado na tabela de símbolos.

2.3.2 DirTable

A DirTable é o equivalente da região .data da TokenTable escrita em código de máquina. A função da DirTable é guardar toda a tradução dessa região feita pelo programa e depois ser impressa em forma de código de máquina. A estrutura de uma DirTable pode ser visualizada no diagrama de blocos abaixo:

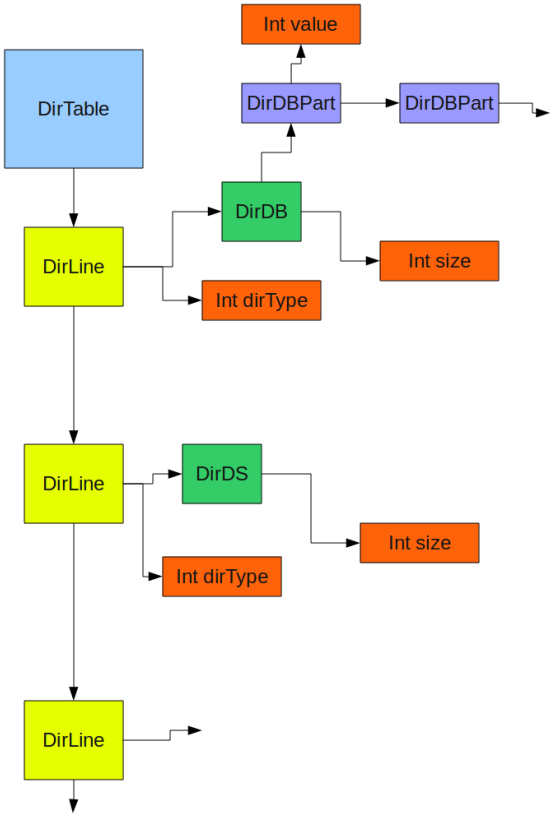

Figure 3 - Estrutura de uma DirTable

Ela é constituída da raíz, DirTable, que aponta para a primeira DirLine. Cada DirLine aponta para a próxima DirLine, se houver, e para uma estrutura de diretiva, que atualmente pode ser DirDS ou DirDB. Qual das duas estruturas está sendo apontada é definido pelo inteiro dirType.

Cada DirDS Possui:

- o int size que armazena a quantidade de espaços que devem ser armazenados pela diretiva DS.
- Cada DirDB Possui:
- o int size que armazena a quantidade de bytes que serão inseridos no codigo de maquina.
- o dirDBPart que armazena o inteiro a ser colocado no espaço. Cada dirDBPart aponta para, se houver, uma próxima dirDBPart.

As diretivas são criadas de duas maneiras, mas inicialmente são enviadas para a mesma função. Na função ***addDiretive()*** são enviadas o ponteiro para a nova diretiva e a token que pode representar essa diretiva. Ela é comparada com ds ou db, caso seja igual a uma das duas, será trabalhada por sua função específica.

A ds é trabalhada pela função ***newDirDS()*** que tenta ler a próxima Token e, se for um inteiro, adiciona como sendo seu size.

A db é trabalhada pela função ***addDirDB()*** que tenta ler a próxima Token e, se for um inteiro, adiciona como sendo sua primeira dirDBPart. Tenta, então, ler a próxima Token. Se for uma vírgula, reinicia esse ciclo até não encontrar mais Tokens na linha. Seu método de alocação permite que seja apenas utilizada a memória necessária e, ignorando outros gargalos, que não haja limite para a quantidade de bytes nessa diretiva, dando flexibilidade.

2.3.3 SimbolTable

Ela é usada para armazenar os símbolos e seus respectivos endereços. Sua estrutura é conforme ilustrado abaixo:

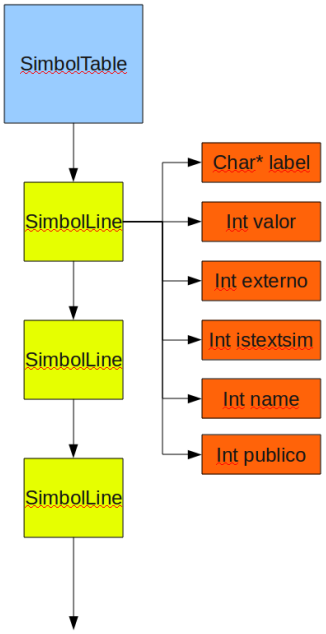

Figure 4 Estrutura de uma SimbolTable

Ela é constituída da raíz, SimbolTable, que aponta para a primeira SimbolLine. Cada SimbolLine aponta para a próxima SimbolLine, se houver, e possui:

- o char* label: a string que representa essa label;
- o int valor que armazena o endereço dessa String, assim que for decodificada;
- o int externo que indica se o simbolo é externo ou não.
- o int istextsim: 1 significa que é um símbolo relativo à seção .text, outro significa simbolo relativo à seção .data.
- o int name que armazena o índice, na .strtab, da string que representa esse simbolo.
- o int publico, que indica se o símbolo é público ou não.

Vê-se que essa estrura foi alterada desde a última versão, isso para adicionar na estrutura outros requisitos do formato ELF.

Seu método de criação é bem simples. Na região .text ou .data, quando uma label é encontrada, ela passa por uma função de análise: ***isLabel()*** que verifica se o último caractere da label é um dois pontos. Depois, passa pela função ***labelClear()*** que retira esse dois pontos e a adiciona na SimbolTable, com o endereço atual do contador de endereços. Dependendo em qual das duas seções esse simbolo foi encontrado, istextsim é modificado.

A cada adição de símbolos, a estrutura que armazena a .strtab é atualizada com a nova String de cada símbolo, e seu campo name é atualizado devidamente.

Simbolos externos são procurados em outra região (entre .begin e .text ou .data). Quando uma diretiva EXTERN é encontrada, o simbolo é adicionado sem passar por ***isLabel()*** ou ***labelClear()***. É adicionado com externo sendo igual a um.

Perto do momento da impressão em ELF, com a função, ***updateStWithDt()***, o programa decodifica os simbolos que são publicos e ajusta seu valor.

2.3.4 UseTable

Ela é utilizada pelo ligador para preencher as posições de memória nas quais as labels EXTERN foram utilizadas. Sua estrutura é conforme ilustrado abaixo:

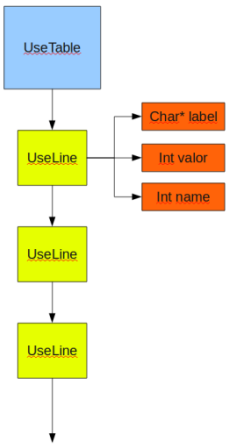

Figure 5 Estrutura de uma UseTable

Ela é constituída da raíz, UseTable, que aponta para a primeira UseLine. Cada UseLine aponta para a próxima UseLine, se houver, e possui:

- o char* label: a string que representa a label usada;
- int valor que armazena o valor da posição onde essa label externa foi chamada.
- o int name que armazena o índice, na .strtab, da String que representa esse símbolo.

Suas linhas são construídas durante a região .data ou .text do programa. Quando uma label é encontrada, o programa verifica se ela é externa com a função ***isExtern()***. Essa percorre toda a SimbolTable e verifica igualdade, e se ela é externa. Caso seja encontrada e seja externa, é adicionada na tabela de uso com seus respectivos campos.

Para encontrar seu name, o programa usa a função ***addStrToStrTab()***, de ***ce.c***, que tenta adicionar essa label na .strtab. Antes, contudo, essa função faz uma busca ***findStrInStrTab()*** e verifica se esse símbolo já existe. Caso exista, ela retorna seu índice, sem adicionar Strings repetidas na .strtab.

2.3.5 DefTable

Ela é utilizada pelo ligador para que ele saiba quais simbolos são PUBLIC dentro desse módulo. Sua estrutura é conforma ilustrado abaixo:

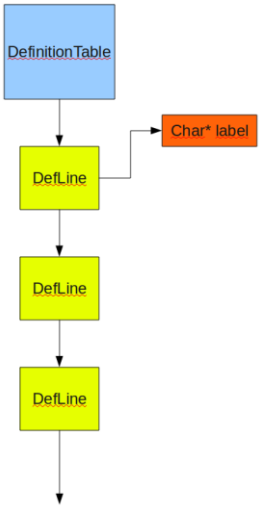

Figure 6 Estrutura de uma DefinitionTable

Ela é constituída pela raíz, DefTable, que aponta para a primeira DefLine. Cada DefLine aponta para a próxima DefLine, se houver, e também possúi:

- o char* label: a string que representa o símbolo público;

Suas linhas são criadas na mesma região das da UseTable, .begin. Quando uma diretiva PUBLIC é encontrada, é criada uma DefLine com a label especificada. Seu valor é definido depois da primeira passagem do montador, quando a SimbolTable está completa, comparando labels até que cada uma da DefLine seja decodificada.

Como já mencionado anteriormente, a decodificação de uma DefLine resulta na modificação da variável publico, na SimbolLine correspondente.

2.4 Montagem: Detalhamento

A montagem do código fonte é realizado pela função ***monta()*** que possui as duas passagens do montador. A primeira passagem do montador é realizado no FOR relativo à TokenTable. Essa parte do código fica extremamente pesada pelas inúmeras verificações de erro. A montagem pode ser definida em algumas regiões:

1. Inicialização: a primeira Token do programa DEVE ser .begin seguida de um identificador. Falhas em seguir essa sintaxe causam um erro fatal no programa, que avisa devidamente essa incorreção. Isso é verificado antes do for que percorre a TokenTable.
2. For que percorre toda a TokenTable. Acaba com a TokenTable ou se achar um .end.
   2.1. Região .begin (Verificada pelo EXTERN/PUBLIC IF): entre .being e .data ou .text ou .end. Aqui são declarados os símbolos com as diretivas EXTERN e PUBLIC. Falha em seguir essa sintaxe causa um erro fatal no programa, que avisará devidamente essa incorreção.
   2.2. Região .text (Verificada pelo FOR .TEXT): entre .text e .data ou .end. Aqui são decladas todas as instruções e alguns símbolos. O programa recebe a Token de cada Line, verifica se é uma label válida. Se for, ou se não

for, verifica a próxima Token. Se for uma instrução, verifica seu número de operandos. Se possuir um ou mais, tenta adicioná-los com as funções descritas na sessão 2.3.1 MCTable, uma a uma. Se todas as funções de adição retornarem FALSE (zero) o programa avisa que o tipo de operando está incorreto. Ele verifica se o número de Tokens na linha não ultrapassa o necessário.

2.3. Região .data (Verificada pelo FOR .DATA): entre .data e .text ou .end. Aqui são declaradas todas as diretivas e alguns símbolos. O programa segue o algoritmo descrito acima, com alteração para diretivas.

3. Impressões: as estruturas de dados são devidamente impressas no arquivo de saída, os símbolos não decodificados são decodificados nessa parte. Os opcodes da MCTable são ajustadas para little endian, depois de terem sido decodificados nessa parte. O DefTable é atualizada com seus valores de endereçamento também nessa região.

4. Limpeza: as estruturas de dados são liberadas da memória.

3 METODOLOGIA DO ARQUIVO DE SAIDA

A saída do programa foi adaptada conforme os padrões ELF 32 encontrado em [1]. A parte de debug foi adapatada conforme o padrão STAB, encontrado em [2]. O arquivo ***ce.h*** armazena todas as definições, estruturas e macros necessárias para criação nesse formato.
A saída do programa é um arquivo ELF 32 Relocável dividido, sempre, em 6 seções:

1- .shstrtab → Nomes das seções

2- .data (Obs:) Optou-se por não usar a seção .bss. Isso permite a flexibilidade ao programador de intercalar várias seções .bss e .data, com diferentes endereços de início e a simplicidade de só ter que trabalhar com uma seção .data na decodificação e codificação do programa.

3- .text

4- .rela.text → Dados de relocação da região .text

5- .strtab → Nomes dos Simbolos

6- .symtab → Tabela de Simbolos, Tabela de Definições e Tabela de Uso.

7- .stab →Tabela de Simbolos e nome do arquivo, para DEBUG.

Todas elas com funções de criação existentes em ***ce.c***. A função ***monta()*** recebeu adição de algumas linhas de código para conseguir a impressão da estrutura elf. Inicialmente foram adicionadas várias variáveis que computam os tamanhos de cada uma dessas seções. Durante a primeira passagem da função ***monta()***, todo o cabeçalho ELF já pode ser criado pela função ***ce()***, que usa essas variáveis em ***monta()*** e as definições em ***ce.h*** para criar todo o cabeçalho e imprimi-lo.

A criação da ***printelf.c*** substituiu totalmente as outras funções de impressão, e cada uma é responsável por uma ou mais seções do arquivo elf.

1- ***elfPrintData()***: Imprime a parte .data do programa, que armazena dados das diretivas ds e db, sem fazer distinção.

2- ***elfPrintText()***: Imprime a parte .text e .rela.text do programa.

3- ***elfPrintStrTab()***: Imprime a parte .strtab

4- ***elfPrintSimbolTable()***: Imprime a parte .symtab, depois de atualizada pela função ***updateStWithDt()***;

5- ***elfPrintStab()***: Imprime a parte .stab, depois de atualizada dentro da função ***elfPrintSimbolTable()***.

4 RESULTADOS

O programa montado usa algoritmos e estruturas de dados criados especialmente para esse projeto. Sua funcionalidade compreende, salvo erros ainda não encontrados, todas as instruções do 8085, as diretivas DS, DB, EXTERN e PUBLIC.

O uso diferenciado da maneira de decodificar as instruções reduziu o tamanho do armazenamento de instruções. A quantidade total de instruções do 8085 é de 255, enquanto o programa armazena 94. Isso implica que a tabela total tem menos de 37% da tabela completa. Isso, sem nenhum algoritmo de organização, já reduz em muito a etapa de busca de instruções e eles serão ainda mais eficientes nessa tabela reduzida.

A saída do programa segue o padrão ELF32 e pode ser devidamente testado com programas de leitura de arquivos ELF, e também pode trabalhar em conjunto com outros softwares básicos que sigam o padrão ELF32.

Nessa versão houve a restruturação da função de erros, com códigos de erro mais precisos e relevantes, e adição dos simbolos de DEBUG automaticamente no arquivo de saída.

5 BIBLIOGRAFIA

[1] < http://www.skyfree.org/linux/references/ELF_Format.pdf>

[2] <http://www.informatik.uni-frankfurt.de/doc/texi/stabs_1.html#SEC1>