Trabalho 1

MC 404 - Organização Básica de Computadores e
Linguagem de Montagem

September 5, 2018

1 Introdução

Um montador (*assembler*) é uma ferramenta que converte um código em linguagem de montagem (*assembly code*) para código em linguagem de máquina, sendo comumente uma parte essencial dos compiladores. Para este trabalho, você irá implementar um montador para a linguagem de montagem do computador IAS, que produza como resultado (saída) um mapa de memória para ser utilizado no simulador do IAS [1].

A entrega deste trabalho será dividida em 2 partes: (1) processamento da entrada, e (2) emissão de mapa de memória.

2 Especificações do Montador

Nesta Seção serão apresentadas as especificações gerais do montador e da linguagem de montagem que deve ser aceita e montada por ele.

2.1 Especificações Gerais

Como entrada ao montador deve ser fornecido um arquivo de texto onde cada linha do mesmo obedece a seguinte expressão:

```
[rotulo:]  [instrução | diretiva]  [# comentário]
```

Para tanto, os colchetes determinam elementos opcionais e a barra vertical separa as opções de elementos válidos. Desta forma, qualquer coisa é opcional, podendo então haver linhas em branco no arquivo de entrada, ou apenas linhas de comentário, ou linhas só com uma instrução ou uma diretiva, etc. É possível que haja múltiplos espaços em branco ou tabulações no início ou fim da linha ou entre os elementos da mesma. Entretanto, quando houver mais de um elemento na mesma linha, estes devem respeitar a ordem definida acima. Por exemplo, caso haja um rótulo e uma instrução na mesma linha, o rótulo deve vir antes da instrução. Assim a sequência é importante.

[1] http://www.ic.unicamp.br/ edson/disciplinas/mc404/2017-2s/abef/IAS-sim/index.html

Como exemplo, as seguintes linhas são válidas num arquivo de entrada para o montador:

```
vetor1:
vetor2:   .word 10
vetor3:   .word 10 # Comentario apos diretiva

.word 10
.word 10  # Comentario apos diretiva
# Comentario sozinho

vetor4: ADD 0x0000000100
vetor5: ADD 0x0000000100 # Comentario apos instrucao

ADD 0x0000000100
```

E as seguintes linhas são **inválidas**:

```
rotulo1: outro_rotulo: mais_um_rotulo:   ##Mais de um rotulo na linha
vetor: .word 10 ADD 0x000000000100       ##Diretiva e instrução na linha
.word 10 .align 5                              ##Diretiva e instrução na linha
vetor: ADD 0x000000000100 .word 10       ##Rotulo com diretiva e instrução na linha
ADD 0x000000000100 ADD 0x000000000100       ##Duas instruções na linha
ADD 0x0000000001 laco:                       ##Instrução depois laço na linha
```

Para linhas inválidas (que não obedecem a expressão geral) o montador deve emitir um código de erro apropriado (veja Seção 3.1). Portanto, algumas regras gerais do montador se seguem:

- É possível que um programa possua palavras de memória com apenas uma instrução (veja a diretiva `.align`). O seu montador deve completar a parte "não preenchida" da palavra com zeros.
- O montador não é sensível à caixa das letras. Sendo assim, o mnemônico LD é válido e ld também é, com o mesmo efeito.
- Seu executável deverá aceitar um argumento, que será o nome do arquivo de entrada, ou seja, o nome do arquivo que contém o programa em linguagem de montagem. O mapa de memória deve ser impresso na saída padrão (`stdout`) em vez de imprimi-lo em um arquivo. Desta forma, um exemplo válido é:

  ```
  ./ias-as arquivoentrada.s
  ```

  que lê um arquivo denominado `arquivoentrada.s` e gera o mapa de memória na saída padrão.

- Os casos de teste não possuirão palavras acentuadas, portanto, não é necessário tratar acentos no montador.

As demais seções descrevem as regras referentes à linguagem de montagem.

2.2 Comentários

Comentários são cadeias de caracteres que servem para documentar ou anotar o código. Essas cadeias devem ser desprezadas durante a montagem do programa. Todo texto à direita de um caractere "#" (cerquilha) é considerado comentário.

2.3 Rótulos

Rótulos são compostos por caracteres alfanuméricos e podem conter o caractere "_" (*underscore*). Um rótulo é definido como sendo uma cadeia de caracteres que deve ser necessariamente terminada com o caractere ":" (dois pontos) e **não** pode ser iniciada com um número. Exemplos de rótulos válidos são:

```
varX:
var1:
_varX2:
laco_1:
__DEBUG__:
```

E exemplos de rótulos inválidos são:

```
varx::    ##Dois caracteres ":" ao final do nome do rótulo
:var1     ##Caractere ":" no início da linha
1var:     ##Rótulo começando com caractere numérico
laco      ##Rótulo deve ser terminado com caractere ":"
ro:tulo   ##Caractere ":" no meio do nome do rótulo
```

2.4 Diretivas

Todas as diretivas da linguagem de montagem do IAS começam com o caractere "." (ponto). As diretivas podem ter um ou dois argumentos, tais quais podem ser dos seguintes tipos:

HEX	Um valor na representação hexadecimal. Estes valores possuem exatamente 12 dígitos, sendo os dois primeiros '0' e 'x' e os 10 últimos dígitos hexadecimais, ou seja, 0-9, A-F. Ex: `0x0a0BADbad6`
DEC(min:max)	Valores numéricos na representação decimal. Apenas valores no intervalo (min:max) são válidos e seu montador **deve** realizar esta verificação. Caso o valor não esteja no intervalo (min:max), o montador deve emitir uma mensagem de erro na saída de erro (`stderr`) e interromper o processo de montagem sem produzir o mapa de memória na padrão (`stdout`).
ROT	Caracteres alfanuméricos e "_" (*underscore*). Não pode começar com número (veja a descrição de rótulos acima) e, neste caso, não deve terminar com o caractere ':'. Note que o caractere ':' só deve ser utilizado na **declaração de um rótulo e não em seu uso**.
SYM	Caracteres alfanuméricos e "_" (*underscore*) - não pode começar com número.

Table 1: Tipos de argumentos válidos.

Com base nos tipos definidos acima, a Tabela 2.4 apresenta as regras (sintaxe) dos elementos de diretiva de montagem e seus respectivos argumentos e tipos.

Sintaxe	**Argumento 1**	**Argumento 2**
.set	SYM	HEX | DEC$(0 : 2^{31} - 1)$
.org	HEX | DEC(0:1023)	-
.align	DEC(1:1023)	-
.wfill	DEC(1:1023)	HEX | DEC$(-2^{31} : 2^{31} - 1)$ | ROT | SYM
.word	HEX | DEC$(-2^{31} : 2^{31} - 1)$ | ROT | SYM	-

Por exemplo: a diretiva `.org` pode receber como argumento um valor hexadecimal (HEX) ou decimal no intervalo (0:1023). Dessa forma, as linhas do seguinte programa são válidas:

```
.org 0x0000000000
.org 0x000000000f
.org 100
```

Enquanto que as seguintes linhas são **inválidas**:

```
.org 0x0000000000 | 10
```

```
.org -10
.org 9999999999
org 0x000 20
.org
```

A descrição do comportamento de cada uma das diretivas está na apostila de programação do computador IAS (Programando o IAS). Também podem ser encontradas nos `slides` das aulas.

2.5 Instruções

As instruções que requerem um endereço podem ser descritas utilizando-se qualquer um dos seguintes formatos:

```
mnemônico HEX
mnemônico DEC(0:1023)
mnemônico ROT
```

As instruções que não requerem o campo endereço possuem o seguinte formato (`RSH`, por exemplo):

```
mnemônico
```

Se o programa especificar o campo endereço para estas instruções, seu montador deve emitir uma mensagem de erro e interromper a montagem. Para as instruções que não requerem o campo endereço seu montador deve preencher os bits referentes ao campo endereço no mapa de memória necessáriamente com zeros.

A Tabela 2.5, apresenta os mnemônicos válidos e as instruções que devem ser emitidas para cada um dos mesmos.

Mnemônico	Instrução a ser emitida
LD	LOAD M(X)
LDINV	LOAD -M(X)
LDABS	LOAD —M(X)—
LDMQ	LOAD MQ
LDMQMX	LOAD MQ,M(X)
STORE	STOR M(X)
JUMP	JUMP M(X,0:19) se o alvo for uma instrução à esquerda da palavra de memória (endereços do tipo HEX ou DEC sempre indicam endereços à esquerda enquanto que rótulos podem indicar endereços à esquerda ou direita). JUMP M(X,20:39) se o alvo for uma instrução à direita de uma palavra de memória.
JUMPP	JUMP+M(X,0:19) se o alvo for uma instrução à esquerda da palavra de memória (endereços do tipo HEX ou DEC sempre indicam endereços à esquerda enquanto que rótulos podem indicar endereços à esquerda ou direita). JUMP+M(X,20:39) se o alvo for uma instrução à direita de uma palavra de memória.
ADD	ADD M(X)
ADDABS	ADD —M(X)—
SUB	SUB M(X)
SUBABS	SUB —M(X)—
MULT	MUL M(X)
DIV	DIV M(X)
LSH	LSH
RSH	RSH
STOREADR	STOR M(X,8:19) se X for o endereço de uma instrução à esquerda de uma palavra. STOR M(X,28:39) se X for o endereço de uma instrução à direita de uma palavra.

3 Parte 1: Processamento da Entrada

Para separar a lógica de leitura da emissão final, compiladores e montadores normalmente são divididos em duas partes, separados por uma estrutura intermediária que permite que cada parte seja independente. Dessa forma, em um futuro, se por decisão de projeto fosse decidido trocar os mnemônicos, nenhuma alteração precisaria ser feita na parte de emissão. Nessa primeira parte, o arquivo de entrada é lido em um vetor de caracteres e você deve implementar um código que processa este vetor e gera uma lista de *tokens*.

De maneira geral, um *token* consiste de uma sequencia consecutiva de caracteres finalizado com um espaço e a este atribuído um significado (tipo do *token*). Este par (nome do *token*, tipo *token*) é comumente utilizado na construção de

linguagens de programação. De tal forma, uma sequência de *tokens* define uma regra e um conjunto de regras definem uma gramática léxica. Similarmente ao montador, temos um conjunto de regras que definem a ordem onde cada *token* é esperado a partit do arquivo de entrada para que a expressão (neste caso, uma linha) seja aceita (faça parte da gramática). Este processo de separação de palavras e sua análise é chamado de análise léxica.

Assim, você pode ler o vetor de entrada caractere por caractere e para cada palavra (conjunto consecutivo, sem espaço, de caracteres) lida, decidir qual o tipo dessa palavra e guardar em qual linha do arquivo de entrada essa palavra está. A estrutura utilizada para representar o *token* é apresentada abaixo:

```
// Instrucao: Todas as possíveis instruções
// Diretiva: Diretivas como ".org"
// DefRotulo: Tokens de definiçao de rótulos, ex.: "label:"
// Hexadecimal, Decimal: São as mesmas definições da seção de Diretivas
// Nome: São as palavras dos símbolos e rótulos.
typedef enum TipoDoToken
    {Instrucao=1000, Diretiva, DefRotulo, Hexadecimal, Decimal, Nome} TokenType;

typedef struct Token {
TokenType tipo;
char* palavra;
unsigned int linha;
} Token;
```

Esses tokens serão armazenados de forma ordenada em uma lista que deverá ser manipulada pelas 5 funções incluídas no cabeçalho disponibilizado:

```
//Insere um token na lista de tokens
//Retorno: 0 se inserido corretamente, -1 caso contrário
unsigned int pushToken(Token novoToken);

//Remove o token no indice pos
//Token se removido correto, NULL caso contrário
Token popToken(unsigned pos);

//Recupera o token no índice pos
//Token se existente, NULL caso contrário
Token getToken(unsigned pos);

//Recupera o tamanho da lista de tokens. Retorna: Tamanho da lista
unsigned int numberOfTokens();

//Imprime a lista com os tokens adicionados.
void imprimeListaTokens();
```

Você deve implementar a função `processarEntrada`, que recebe como parâmetro uma *string* (contendo todo o conteúdo do arquivo). No final da execução desta

função, a lista de *tokens* deverá estar corretamente preenchida ou ter impresso um erro na saída de erro (*stderr*). Caso a lista tenha sido preenchida corretamente, a função deve retornar o valor 0 (zero), do contrário, o valor 1 deve ser retornado. O código base para você começar a implementação está disponível em: codigo_montador_v2.zip e, no caso da parte 1, você deve modificar e submeter apenas o arquivo `processarEntrada.c`.

3.1 Tratamento de Erros

Durante o processamento e criação dos *tokens* você deverá checar dois tipos de erros: léxicos e gramaticais. Um erro léxico ocorre quando uma palavra da entrada não se encaixa em nenhum tipo de *token*. Por exemplo, "vos1e" não é uma palavra válida na norma padrão do português, portanto um erro léxico. Por exemplo, as seguintes expressões caracterizam erros léxicos:

```
ADDDD 0x0000000000
MULT 0xx001900000
.word:
0x1400:
```

Neste caso, o montador deve interromper a montagem e emitir uma mensagem de erro no seguinte padrão:

```
ERRO LEXICO: palavra inválida na linha XX!
```

onde XX é o número da linha do arquivo de entrada que causou o erro.

Já um erro gramatical ocorre quando um *token* é seguido de outro *token* não esperado. Já na frase: "você, come.", apesar dos *tokens* "você", ",", "come" e "." serem válidos, entre um sujeito e um verbo não é esperado uma vírgula. Por exemplo, as seguintes expressões caracterizam erros gramaticais:

```
x: y: MULT 0x0000000010
0x000000001A MULT 0x0000000011
```

Nas expressões acima, o *token* tipo rotulo, não pode vir seguido de outro token do tipo rótulo. O *token* Hexadecimal não pode vir antes da instrução. Da mesma forma, enquanto você processa a entrada, você deverá identificar esses dois tipos de erro e, caso encontre, parar o montador e imprimir o erro na tela.

A seguinte linha contém um erro léxico:

```
0x1000: #Se é um número, não deveria terminar com ':' e,
        #Se é uma label, não poderia começar com 0x.
```

A seguinte linha contém um erro gramatical:

```
ADD VETOR: # Depois de uma instrução ADD era esperado um número e não um label.
```

Neste caso, o montador deve interromper a montagem e emitir a seguinte mensagem:

```
ERRO GRAMATICAL: palavra na linha XX!
```

4 Parte 2: Emissão do mapa de memória

Uma vez que o arquivo de entrada foi lido e analisado e a lista de *tokens* preenchida, o montador deve emitir o mapa de memória. Para isso, você deve implementar a função `emitirMapaDeMemoria`, que deve analisar a lista de *tokens* por meio das funções de manipulação da lista apresentadas anteriormente e produzir o mapa de memória. Caso o mapa de memória tenha sido produzido corretamente, a função deve retornar o valor 0 (zero), do contrário, o valor 1 deve ser retornado.

A saída do mapa de memória deverá ser feita na saída padrão. O mapa de memória deve ser formado por linhas no seguinte formato:

```
AAA DD DDD DD DDD
```

Na linha acima, AAA é uma sequência de 3 dígitos hexadecimais que representa o endereço de memória, totalizando 12 bits. Já DD DDD DD DDD é uma sequência de 10 dígitos hexadecimais, que totaliza 40 bits e representa um dado ou duas instruções do IAS, conforme já visto em aula. Note que existem caracteres de espaço na linha, num total de exatos 4 espaços - é importante que seu montador produza um mapa de memória EXATAMENTE nesse formato para permitir a execução dos casos de teste. Não deve haver caracteres extras ou linhas em branco, apenas linhas no formato acima.

Nessa parte do trabalho você deverá checar se um rótulo ou símbolo foi declarado em algum lugar do código. Caso não tenha sido, você deverá emitir o seguinte erro:

```
USADO MAS NÃO DEFINIDO: XXXX!
```

Onde o XXXX deverá ser o nome do símbolo ou rótulo não definido.

Além disso, você deverá emitir um aviso na saída de erro (`stderr`), caso o nome do símbolo ou rótulo foi definido mas nunca utilizado, uma mensagem igualmente a abaixo deve ser emitida.

```
AVISO: XXX NÃO UTILIZADO
```

Onde o XXXX deverá ser o nome do símbolo ou rótulo que não foi utilizado. Nota-se que esse não é um erro, mas apenas um aviso que deve ser impresso na saída de erro e portanto o montador não deve ser interrompido e o mapa de memória deve ser gerado.

Por fim, qualquer outro erro como palavra desalinhada ou sobreescrição de código, deverá resultar na parada da montagem com a seguinte mensagem de erro:

```
Impossível montar o código!
```

O código base para você começar a implementação da parte 2 também está disponível em: codigo_montador_v2.zip. No caso da parte 2, você deve modificar e submeter apenas o arquivo `emitirMapaDeMemoria.c`.

5 Dicas

- Consulte a apostila de programação do IAS para informações sobre a semântica das diretivas e a codificação das instruções da linguagem de montagem.
- Casos de teste propositalmente errados serão usados. O montador não deve gerar um mapa de memória nesses casos, nem mesmo parcial. Ele deve apontar o primeiro erro encontrado (de cima para baixo, da esquerda para a direita) e parar a montagem (veja a seção de tratamento de erros).
- O código-fonte do montador, em C, deve ser bem comentado e organizado. Variáveis com nomes amigáveis, tabulações que facilitem a leitura, etc serão levadas em conta na correção e podem aumentar ou diminuir a nota.

6 FAQ

- Não é preciso fazer a checagem do tamanho dos números. Isso não será testado.
- Não há um defSimbolo! Uma declaração de símbolo, como ".set algumNome 10", teria a seguinte lista de tokens: Diretiva Nome Dec.
- Como não é possível diferenciar na primeira parte se uma palavra é um rótulo ou símbolo, unimos os dois em um único tipo: nome.
- Na primeira parte não é preciso checar se um rótulo ou símbolo já foi declarado. Esse erro deve ser tratado na segunda parte.
- Tamanho máximo dos rótulos: 64 caracteres, incluindo o ":".
- Tamanho máximo dos símbolos na diretiva .set: 64 caracteres.
- Tratamento de diretivas .word ou .wfill quando a posição de montagem estiver apontando para o lado direito de uma palavra de memória: interromper o processo de montagem com mensagem de erro.
- Quando o parâmetro das instruções JUMP, JUMPP ou STORADR for um número (HEX ou DEC), em vez de rótulo, a instrução gerada deve considerar que o endereço é relativo à parte esquerda da palavra de memória.
- O caractere TAB ($\widehat{)}$ deve ser tratado como espaço? Sim.
- Caso seja encontrada uma palavra reservada (por exemplo um mnemônico) como argumento da diretiva .set o que deve ser feito? Você pode ignorar este caso pois não serão realizados testes em que o argumento da diretiva set seja uma palavra reservada.
- A diretiva .set pode ser utilizada com uma palavra reservada, como um mnemônico? Não.

- O que fazer caso o programa produza conteúdo de memória além das 1024 palavras endereçáveis? Este tipo de teste não será realizado.