Trabalho 1 – Montador para o computador IAS

MC404 - Organização Básica de Computadores e
Linguagem de Montagem

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 das ferramentas de compilação. 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.

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

[1] http://www.ic.unicamp.br/~edson/disciplinas/mc404/2019-2s/ab/IAS-sim/index.html

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

.word 10
.word 10  # Comentário apos diretiva
# Comentário sozinho

vetor4: ADD 0x0000000100
vetor5: ADD 0x0000000100 # Comentário após instrução

ADD 0x0000000100
```

As seguintes regras gerais devem ser observadas:

- 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.
- A linguagem de montagem 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:

  ```
  ./montador.x arquivoentrada.in
  ```

  que lê um arquivo denominado `arquivoentrada.in` 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. Os tipos de argumentos está descrito na tabela 1.

HEX	Um valor na representação hexadecimal. Estes valores possuem **até** 12 dígitos, sendo os dois primeiros "0" e "x" e os demais dígitos hexadecimais, ou seja, 0-9, A-F, ou 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 saída padrão (`stdout`).
NROT	Nomes de rótulos são compostos por caracteres alfanuméricos e "_" (*underscore*). Não podem começar com número (veja a descrição de rótulos acima) e, neste caso, não devem 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.

Tabela 1: Tipos de argumentos válidos.

Com base nos tipos definidos acima, a Tabela 2 apresenta a sintaxe das diretivas de montagem, incluindo seus nomes e respectivos argumentos.

Diretiva	**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$(-2^{31} : 2^{31} - 1)$ | SYM
.word	HEX | DEC$(-2^{31} : 2^{31} - 1)$ | NROT | SYM	-

Tabela 2: Sintaxe das diretivas de montagem.

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 pode ser encontrada na apostila de programação do computador IAS ().

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 NROT
```

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

```
mnemônico
```

Para as instruções que não requerem o campo endereço, seu montador deve preencher os *bits* referentes ao campo endereço no mapa de memória necessariamente com zeros.

(caso opte pela parte extra): **caso opte por tratar erros, se o programa especificar o campo endereço para estas instruções, seu montador deve emitir uma mensagem de erro e interromper a montagem.**

A Tabela 2.5 apresenta a sintaxe das instruções em linguagem de montagem, incluindo os mnemônicos válidos e as instruções de máquina 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	Caso o parâmetro da instrução JUMP seja um rótulo associado à esquerda de uma palavra de memória com endereço X ou endereços codificados na forma HEX ou DEC, o montador deve emitir a instrução JUMP $M(X, 0 : 19)$. Caso o parâmetro seja um rótulo associado à direita de uma palavra de memória com endereço X, então o montador deve emitir a instrução JUMP $M(X, 20 : 39)$.
JGE	Caso o parâmetro da instrução JGE seja um rótulo associado à esquerda de uma palavra de memória com endereço X ou endereços codificados na forma HEX ou DEC, o montador deve emitir a instrução JUMP+ $M(X, 0 : 19)$. Caso o parâmetro seja um rótulo associado à direita de uma palavra de memória com endereço X, então o montador deve emitir a instrução JUMP+ $M(X, 20 : 39)$.
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
STOREND	Caso o parâmetro da instrução STOREND seja um rótulo associado à esquerda de uma palavra de memória com endereço X ou endereços codificados na forma HEX ou DEC, o montador deve emitir a instrução STOR $M(X, 8 : 19)$. Caso o parâmetro seja um rótulo associado à direita de uma palavra de memória com endereço X, então o montador deve emitir a instrução STOR $M(X, 20 : 39)$.

Tabela 3: Lista de mnemônicos da instruções em linguagem de montagem e mapeamento para instruções do computador IAS.

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*.

Um *token* consiste de uma sequência consecutiva de caracteres finalizada com um separador (espaço, caractere de tabulação, quebra de linha ou fim do arquivo) e a cada *token* é atribuído um significado (o tipo do *token*).

A "análise léxica" é a primeira fase de um processo de tradução (ou compilação) de linguagens de programação e sua função é fazer a leitura do programa fonte, caractere a caractere, agrupar os caracteres em "lexemas" e produzir uma sequência de símbolos léxicos conhecidos como *tokens*.

Seu código deverá identificar os *tokens* do programa de entrada e produzir uma lista de *tokens* que serão processados pelo *backend* do montador (a segunda parte). Para isso, nesta parte do trabalho, seu programa deve ler o vetor de entrada caractere por caractere e para cada palavra (conjunto consecutivo, sem espaço, de caracteres) encontrada, identificar que tipo de *token* esta palavra representa e registrar o *token*, incluindo a linha do arquivo onde se encontra o *token*, na lista de *tokens*. A estrutura a ser utilizada para representar o *token* é apresentada abaixo:

```
// Instrução: Todos o mnemônicos das 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* devem ser armazenados de forma ordenada em uma lista que deverá ser manipulada pelas 5 funções incluídas no cabeçalho disponibilizado:

```
// Cria um token e o insere na lista de tokens.
// Para criar um token é necessário seu tipo, a palavra que
// representa esse token e a linha em que foi encontrado
// Retorno: 0 se inserido corretamente, -1 caso contrário
unsigned int adicionarToken(TokenType tipo, char* palavra,
                            unsigned int linha);
```

```
// Remove o token no índice pos
// Retorno: Token* se removido corretamente,
//          NULL caso contrário
Token* removerToken(unsigned pos);

// Recupera o token no índice pos
// Retorno: Token* se existente, NULL caso contrário
Token* recuperarToken(unsigned pos);

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

//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. **Não existirá entradas inválidas! Você nâo precisa identificar erros!**

O código base para você começar a implementação está disponível em: codigo_base.tar.gz e, no caso da parte 1, você deve modificar e submeter o arquivo `processarEntrada.c`.

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 rótulos foram declarados em algum lugar do código e se os símbolos foram definidos antes (em uma linha anterior) do uso. Caso não tenham sido, você deverá emitir o seguinte erro:

```
ERRO: Rótulo ou símbolo usado mas não definido: XXX
```

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

Esse é o único erro o qual você deve tratar caso nâo opte pela parte extra do trabalho.

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

5 Requisitos/Informações de Entrega

Os prazos para entrega dos trabalhos são:

- **Parte 1 - Processamento da Entrada**
  - Até 04/11/2020 - 23:59h Fator Multiplicativo = 1,0.
  - Até 11/11/2020 - 23:59h. Fator Multiplicativo = 0,8. Não serão aceitos trabalhos entregues após este prazo.
- **Parte 2 - Emissão de Mapa de Memória**
  - Até 11/11/2020 - 13:59h. Fator Multiplicativo = 1,0.
  - Até 16/11/2020 - 13:59h. Fator Multiplicativo = 0,8. Não serão aceitos trabalhos entregues após este prazo.

6 Extra: tratamento de erros (opcional)

A submissão final do seu trabalho poderá conter um sistema de detecção de entradas inválidas. Ter essa parte implementada, se correto, poderá te adicionar 1.5 no seu trabalho. Ou seja, ao invéz do trabalho ir de 0 a 10, ele poderá ir até 11.5. Mas, note, isso é opcional e sua nota final geral no semestre é de no máximo 10.

Caso prossiga com a parte opcional do seu trabalho, o seu código deverá imprimir um erro, caso a entrada seja inválida, na saída de erro (`stderr`) (agora, differente da parte 1 padrão, será considerado entradas inválidas). Seu código deverá retornar o valor 1 (zero) caso a entrada seja inválida, do contrário, o valor 0 deve ser retornado.

Exemplos de entradas inválidas estão na pasta **extra** na pasta *testes*: codigo_base.tar.gz.

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:

```
MULT 0xx001900000  # 0xx001900000 é um token inválido
.word:             # .word: é um token inválido
0x1400:            # 0x1400: é um token inválido
```

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 uma linha contém uma sequência de *tokens* não esperada. 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:

```
ADDDD 0x0000000000
x: y: MULT 0x0000000010
0x000000001A MULT 0x0000000011
```

Nas expressões acima, a primeira linha não pode começar com um *token* do tipo nome (`ADDDD`[2]). Na segunda linha, *token* de tipo rótulo não pode vir seguido de outro *token* do tipo rótulo. Por fim, na terceira linha, o *token* Hexadecimal (`0x000000001A`) não pode vir antes do *token* tipo instrução. Neste caso, o montador deve interromper a montagem e emitir a seguinte mensagem:

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

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.

6.0.1 Exemplos

A Tabela 4 mostra tipos e exemplos de erros léxicos.

Erro léxico	**Exemplo 1**	**Exemplo 2**
Rótulo inválido	`rotulo:::`	`0rotulo:`
Número Hexadecimal/Decimal inválido	`0x10Y`	`123cactus`
Diretiva inválida/desconhecida	`.`	`.wfillet`
Palavra inválida	`v-ar1`	`v@raiable`

Tabela 4: Tipos e exemplos de erros léxicos.

[2]Note que ADDDD não é do tipo `Instrucao`, pois não é um mnemônico válido.

A primeira linha do trecho de código a seguir contém um erro léxico:

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

Já os erros gramaticais podem se encaixar em quatro categorias, indicadas na Tabela 5.

Erro gramatical	**Exemplo 1**	**Exemplo 2**
Número de parâmetros diferentes do esperado	`.word 0x30 0x40 100`	`.set`
Paramêtros inválidos	`.wfill 15 .align`	`.word .word`
Valores fora do intervalo	`.wfill 1000000 0x10`	`.align 0xFFFFFFFF`
Formato de linha inválido	`rot1:  rot2:`	`ADD x rot1:`

Tabela 5: Tipos e exemplos de erros gramaticais.

A primeira linha do seguinte trecho de código contém um erro gramatical:

```
ADD VETOR: # Depois de uma instrução ADD era esperado um número
           # ou um nome de rótulo (NROT), e não a definição
           # de um rótulo.
```

Não é necessário informar qual categoria um erro léxico ou gramatical se encaixa. Entretanto a mensagem padrão para cada erro deve ser informada.

7 Observações Importantes

- A linguagem de programação a ser utilizada para desenvolver o montador deve ser **obrigatoriamente a linguagem C**. Não serão aceitos trabalhos que façam uso de alguma biblioteca não-padrão, ou seja, apenas podem ser utilizadas as rotinas da biblioteca padrão do compilador (Não modifique o Makefile para incluir novas bibliotecas!).
- O trabalho é individual e em nenhuma hipótese você pode compartilhar seu código ou fazer uso de código de outros. No caso de fraude, como plágio, todos os envolvidos serão desonrados com atribuição de nota 0 na média da disciplina.
- Utilize a plataforma Moodle para entrega.
- Deverão ser entregues os dois arquivos: `processarEntrada.c` (para parte 1) e `emitirMapaDeMemoria.c` (para parte 2). Você também pode submeter mais quatro arquivos adicionais, que auxiliem estes dois. O tamanho total não deve ser maior que 1MB. É importante que sua implementação não dependa de modificações em outros arquivos e você não deve entregar os arquivos `main.c`, `token.c` e `token.h`. A correção será realizada levando em consideração os arquivos `main.c` e `token.c/token.h` originais (sem alterações).

- Os arquivos de teste (em linguagem de montagem) terão no máximo 4096 caracteres.
- Caso não seja especificada por uma diretiva `.org`, a posição inicial de montagem é no endereço 0 à esquerda.
- Serão 12 testes ao todo: 9 testes abertos e 3 testes fechados.
- O código base (`codigo_base.tar.xz`) está disponível em: https://ic.unicamp.br/~edson/disciplinas/mc404/2020-2s/ab/anexos/codigo_base.tar.xz

8 Dicas

- Consulte a apostila de programação do IAS para informações sobre a semântica das diretivas e a codificação das instruções da linguagem de montagem.
- O código-fonte do montador, em C, deve ser bem comentado e organizado. Recomendamos o uso de variáveis com nomes amigáveis, tabulações que facilitem a leitura, etc.
- Você pode utilizar estruturas auxiliares para armazenar informações temporárias.
- O arquivo `LEIAME_FAQ.txt` disponibilizado em codigo_base.tar.gz contém informações sobre a infraestrutura de compilação, execução e teste.

9 FAQ

- Como não é possível diferenciar na primeira parte se uma palavra é um nome de rótulo ou símbolo, unimos os dois em um único tipo: nome.
- É preciso checar se um rótulo ou símbolo já foi declarado na primeira parte? Não, esse erro deve ser tratado na segunda parte.
- Qual é o tamanho máximo dos rótulos? 64 caracteres, incluindo o ":".
- Qual é o tamanho máximo dos símbolos na diretiva `.set`? 64 caracteres.
- Haverá testes onde um símbolo será declarado múltiplas vezes? Não.
- Quando o parâmetro das instruções JUMP, JGE ou STOREND 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 deve ser tratado como espaço? Sim.

- A diretiva `.set` pode ser utilizada com uma palavra reservada, como um mnemônico? Não, deve ser utilizado apenas para definir constantes numéricas.
- O que fazer caso o programa produza conteúdo de memória além das 1024 palavras endereçáveis? Este tipo de teste não será realizado.
- Valores decimais fora do intervalo definido devem ser tratados como erros gramaticais? Sim.
- É permitido o uso das funções da biblioteca padrão da linguagem C? Sim. Veja as funções em https://en.cppreference.com/w/c/header.
- (parte extra) Os casos de testes possuirão mais de um erro por teste? Não.