Trabalho 1: Montador para a arquitetura do computador IAS

Avisos

Nesta seção serão apresentados os avisos importantes referentes ao trabalho 1.

- 13/09/2017: Saída do teste 12 corrigida - Teste corrigido disponível em testes-abertos_v2.zip (./testes-abertos_v2.zip)
- 11/09/2017: Casos de testes abertos divulgados em testes-abertos_v1.zip (NOTA: este arquivo foi substituído pelo arquivo testes-abertos_v2.zip).
- 11/09/2017: Grupo de discussões da disciplina criado. Alunos da disciplina devem se registrar em https://groups.google.com/forum/#!forum/unicamp-mc404-2s2017 (https://groups.google.com/forum/#!forum/unicamp-mc404-2s2017) e as dúvidas sobre o trabalho devem ser enviadas para a lista.
- 11/09/2017: Adicionamos a resposta de algumas dúvidas no texto, unimos os tipos de tokens "Rotulo" e "Simbolo" e definimos um novo tipo de erro.
- 06/09/2017: Enunciado do trabalho 1 divulgado!

Introdução

Um montador (*assembler*) é uma ferramenta que converte código em linguagem de montagem (*assembly code*) para código em linguagem de máquina. Neste trabalho, você irá implementar um montador para a linguagem de montagem do computador IAS, que produza como resultado um mapa de memória para ser utilizado no simulador do IAS, que pode ser encontrado em: http://www.ic.unicamp.br/~edson/disciplinas/mc404/2017-2s/abef/IAS-sim/index.html (http://www.ic.unicamp.br/%7Eedson/disciplinas/mc404/2017-2s/abef/IAS-sim/index.html).

A entrega do trabalho será dividida em 2 partes: (1) processamento da entrada e (2) emissão de mapa de memória. Os prazos de entrega estão descritos na seção Requisitos/informações da entrega.

Especificações do montador

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

Especificações gerais

O arquivo de entrada do montador deve ser um arquivo de texto tal que cada linha deve ser como uma das seguintes:

```
[rotulo:] [instrucao] [# comentario]
```

ou

```
[rotulo:] [diretiva] [# comentario]
```

Nas linhas acima, os colchetes determinam elementos opcionais - ou seja, 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, etc. É possível que haja múltiplos espaços em branco no início ou fim da linha ou entre os elementos. 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.

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:

vetor: .word 10 ADD "0x000000000100"

.word 10 .align 5

vetor: ADD "0x000000000100" .word 10

ADD "0x000000000100" ADD "0x000000000100"

ADD "0x0000000001" laco:
```

Algumas regras gerais do montador:

- 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 deve ser sensível à caixa das letras (o mnemônico LOAD é válido enquanto que o mnemônico load não é). O montador deve interromper a montagem e produzir uma mensagem de erro (ver abaixo) quando esta regra for violada.
- Os casos de teste não possuirão palavras acentuadas, portanto, não é necessário tratar acentos no montador.
- O executável do montador deve aceitar um argumento.
  - o argumento deverá ser o nome do arquivo de entrada, ou seja, o nome do arquivo que contém o programa em linguagem de montagem.
  - você deve imprimir o mapa de memória na saída padrão (`stdout`), em vez de imprimi-lo em arquivo.
  - Um exemplo válido é:

    ```
    ./ias-as entrada.s
    ```

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

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

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.

Rótulos

Rótulos são compostos por caracteres alfanuméricos e podem conter o caractere "_" (*underscore*). Um rótulo é definido como uma cadeia de caracteres que deve ser 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__:
```

Exemplos de rótulos **inválidos** são:

```
varx::
:var1
1var:
laco
ro:tulo
```

Diretivas

Todas as diretivas da linguagem de montagem do IAS começam com o caractere "." (ponto). As diretivas podem ter um ou dois argumentos - tais argumentos 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 ou A-F. Ex: `0x0a0Ef217D0`
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`).
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.

A Tabela 1, abaixo, apresenta a sintaxe das diretivas de montagem e os tipos de seus argumentos.

Sintaxe	**Argumento 1**	**Argumento 2**
.set	SYM	HEX | DEC(-2^{31}: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^{32}-1) | ROT | SYM	

Tabela 1: Sintaxe das diretivas de montagem e os tipos dos argumentos.

O caractere "|" separa as opções. 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 0x000 20
.org
```

A descrição do comportamento de cada uma das diretivas está na apostila de programação do computador IAS (programando_o_IAS.pdf (http://ic.unicamp.br/%7Eedson/disciplinas/mc404/2017-2s/abef/anexos/programando_o_IAS.pdf)). Também podem ser encontradas nos *slides* das aulas.

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 nao requerem o campo endereço possuem o seguinte formato:

- `mnemônico`

Note que no segundo campo é possível usar um endereço em hexadecimal (HEX) ou decimal (DEC), ou o identificador de um rótulo (sem o caractere ':'). O endereço, ou o rótulo, deve ser descrito entre aspas duplas. Exemplos válidos são:

```
ADD "laco"
SUB "0x00000000F4"
```

Algumas instruções não requerem o campo endereço (RSH, por exemplo). 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 com zeros.

A Tabela 2, abaixo, apresenta os mnemônicos válidos e as instruções que devem ser emitidas para cada um dos casos.

Mnemônico	**Instrução a ser emitida**
LOAD	LOAD M(X)
LOAD-	LOAD -M(X)
LOAD|	LOAD |M(X)|
LOADmq	LOAD MQ
LOADmq_mx	LOAD MQ,M(X)
STOR	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.
JMP+	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)
ADD|	ADD |M(X)|
SUB	SUB M(X)
SUB|	SUB |M(X)|
MUL	MUL M(X)
DIV	DIV M(X)

LSH	LSH
RSH	RSH
STORA	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.

Tabela 2: Mnemônicos válidos e as instruções que devem ser emitidas para cada um dos casos.

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*. 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, ou *token*, e guardar em qual linha do arquivo de entrada essa palavra está. Essas informações vão compor o que chamaremos de *token* e será representado pela seguinte estrutura:

```
enum TipoDoToken {Instrucao, Diretiva, DefRotulo, Hexadecimal, Decimal, Nome};
// 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 struct Token {
  TipoDoToken tipo;
  char* palavra;
  unsigned linha;
} Token;
```

Esses *tokens* serão armazenados de forma ordenada em uma lista que deverá ser manipulada pelas 3 funções incluídas no header disponibilizado:

```
  // Retorna a posição do token
  unsigned adicionarToken(Token novoToken);

  void removerToken(unsigned pos);

  Token recuperaToken(unsigned pos);
```

Você também pode utilizar as seguintes funções para inspecionar o tamanho da lista ou imprimir a lista:

```
  unsigned numberOfTokens();

  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 (http://ic.unicamp.br/%7Eedson/disciplinas/mc404/2017-2s/abef/labs/trab01/codigo_montador_v2.zip) e, no caso da parte 1, você deve modificar e submeter apenas o arquivo

`processarEntrada.c`.

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*. Já um erro gramatical ocorre quando um *token* é seguido de outro *token* não esperado. Por exemplo, "vos1e" não é uma palavra válida na norma padrão do português, portanto um erro léxico. Já na frase: "você, come.", apesar dos *tokens* "você", ",", "come" e "." serem válidos, entre um sujeito e um verbo não é esperado uma vírgula. 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.
```

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.

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!
```

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 analizar 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.

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 (http://ic.unicamp.br/%7Eedson/disciplinas/mc404/2017-2s/abef/labs/trab01/codigo_montador_v2.zip). No caso da parte 2, você deve modificar e submeter apenas o arquivo `emitirMapaDeMemoria.c`.

Requisitos/informações da entrega

Os prazos para a entrega do trabalho são:

- **parte 1: processamento de entrada.**
  - **Até 19/09 - 23:59h. Fator multiplicativo = 1.0**
  - **Até 20/09 - 11:59h. Fator multiplicativo = 0.8**
  - **Até 09/10 - 11:59h. Fator multiplicativo = 0.6**
- **parte 2: emissão do mapa de memória.**
  - **Até 05/10 - 23:59h. Fator multiplicativo = 1.0**
  - **Até 07/10 - 23:59h. Fator multiplicativo = 0.8**
  - **Até 09/10 - 11:59h. Fator multiplicativo = 0.6**
- **OBS: trabalhos entregues após 09/10 - 11:59h não serão aceitos.**

Outras 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.
- O trabalho é individual e em nenhuma hipótese você pode compartilhar seu código ou fazer uso de código de outros. Caso haja qualquer tentativa de fraude, como plágio, todos os envolvidos serão desonrados com atribuição de **nota 0** na média da disciplina.
- Utilize a plataforma SuSy para entrega. A primeira parte deve ser entregue em: https://susy.ic.unicamp.br:9999/mc404abef/T1p1 (https://susy.ic.unicamp.br:9999/mc404abef/T1p1) e a segunda parte em: https://susy.ic.unicamp.br:9999/mc404abef/T1p2 (https://susy.ic.unicamp.br:9999/mc404abef/T1p2).
- Deverão ser entregues apenas dois arquivos: `processarEntrada.c` e `emitirMapaDeMemoria.c`. O primeiro referente à parte 1 e o segundo referente à parte 2. É importante que sua implementação não dependa de modificações em outros arquivos.
- 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.

Dicas

- Consulte a apostila de programação do IAS (programando_o_IAS.pdf (http://ic.unicamp.br/%7Eedson/disciplinas/mc404/2017-2s/abef/anexos/programando_o_IAS.pdf)) 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.

Perguntas Frequentes e Outras Dúvidas

- **O campo palavra dos tokens Hexadecimal, Decimal e Nome não devem conter as aspas.**
- **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`, `JMP+` ou `STORA` 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 (\t) 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.
- É necessário que a diretiva .set trate outros valores além de números, como mnemônicos ou rótulos em geral? Não, os casos de teste apenas vão contemplar `.set SYM HEX | DEC(-`2^{31}`:`2^{31}`-1)`, por exemplo, `.set TEMP 23`.