UNIVERSIDADE FEDERAL DE MINAS GERAIS

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Software Básico
Trabalho Prático 1 – Mini-Sistema de Programação

Caio Felipe Zanatelli

Professor: Marcos Augusto M. Vieira

Belo Horizonte
23 de setembro de 2017

Sumário

1 Introdução

Este trabalho tem como objetivo a criação de um mini-sistema de programação capaz de executar programas em uma linguagem de máquina previamente definida (M2 – Linguagem do Marcos). Para tanto, este documento aborda todas as etapas individuais que compõem o trabalho como um todo: criação de um montador (*assembler*) para a linguagem; um editor de ligação (*linker*), de forma a permitir a tradução de módulos em separado e a realocação de programas; um carregador e um simulador de M2, de forma a possibilitar a execução dos códigos gerados; e, por fim, a utilização de um processador de macros (*GNU M4*) para a criação e expansão de macros.

2 Solução do Problema

Esta seção tem como objetivo apresentar cada etapa do desenvolvimento, como citado anteriormente. Assim, de forma a proporcionar uma melhor visualização, a Figura 1 ilustra a sequência dos módulos, os quais serão abordados na sequência.

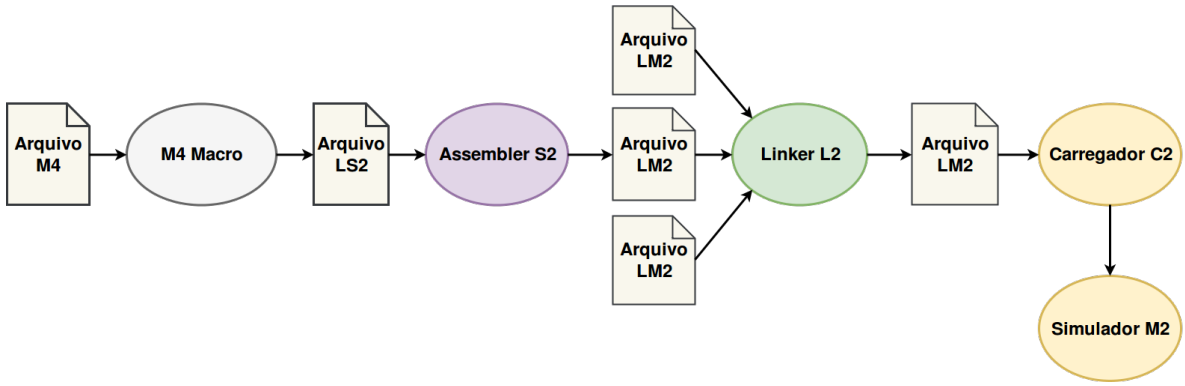

Figura 1 – Diagrama de etapas do mini-sistema

.

2.1 Simulador M2

Para que fosse possível a execução de códigos em linguagem LS2, foi criada uma máquina de simulação M2, isto é, uma máquina virtual capaz de realizar operações bem definidas pela linguagem. Para tanto, inicialmente o programa é carregado na memória de M2, e então a execução é iniciada. A máquina em questão possui as seguintes características:

- **Unidade endereçável**: palavra (correspondente a um inteiro).
- **Dados**: somente inteiro.
- **Registradores**: PC (*Program Counter*) e AC (acumulador).
- **Formato de instrução**: Operação | deslocamento.
- **Endereçamento**: relativo ao contador do programa (PC).

Como a estrutura adotada neste trabalho é do tipo acumulador, o simulador possui apenas dois registradores (AC e PC). Além disso, arquitetura adotada é *Von Neumann*, ou seja, as instruções e os dados são armazenados em uma única memória. Com isso, cabe ao programador fazer o controle de contexto em seus códigos, uma vez que o acumulador pode ser facilmente sobrescrito. A Tabela 1 apresenta o conjunto de instruções suportadas.

Tabela 1 – Instruções

Código	**Símbolo**	**Descrição**	**Ação**
01	LAD M	Carga AC direto	AC <- mem[M + PC]
02	SAD M	Armazena AC direto	mem[M + PC] <- AC
03	ADD M	Soma	AC <- AC + mem[M +PC]
04	SUB M	Subtrai	AC <- AC - mem[M + PC]
05	INP M	Entrada	mem[M + PC] <- inteiro lido
06	OUT M	Saída	Escreve mem[M + PC]
07	JMP M	Desvio	PC <- M + PC
08	JGZ M	Desvio se AC maior que zero	Se AC >0, então PC <- M + PC
09	JLZ M	Desvio se AC menor que zero	Se AC <0, então PC <- M + PC
10	JZE M	Desvio se AC igual a zero	Se AC == 0, então PC <- M + PC
11	HLT	Para a execução	Termina o programa
12	LXD M	Carrega RX direto	RX <- mem[M + PC]
13	SXD M	Armazena RX direto	mem[M + PC] <- RX
14	LAX	Carrega AC indexado	AC <- mem[RX] RX <- RX + 1
15	SAX	Armazena AC indexado	mem[RX] <- AC RX <- RX + 1
16	LCD M	Carrega RC direto	RC <- mem[M + PC]
17	JCC M	Conta e desvia se RC >0	RC <- RC - 1; Se RC >0, então PC <- M + PC
18	CAL M	Chama subprograma	RX <- PC PC <- M + PC
19	RET	Retorna	PC <- RX
20	LAI	Carrega AC indireto	AC <- mem[mem[RX]] RX <- RX + 1
21	SAI	Armazena AC indireto	mem[mem[RX]] <- AC RX <- RX + 1
22	DOB M	Calcula dobro	mem[M + PC] <- 2 · mem[M + PC] AC <- mem[M + PC]
23	MET M	Calcula metade	mem[M + PC] <- mem[M + PC] / 2 AC <- mem[M + PC]
24	JPA M	Desvio se AC par	Se AC for par, então PC = M + PC

Além das instruções acima, também existem quatro pseudo-instruções. Tais pseudo-instruções e quais são as traduções equivalentes, uma vez que não se trata de instruções, e portanto não são códigos executáveis por M2, são apresentadas pela Tabela 2.

Tabela 2 – Pseudo-instruções

Pseudo-instrução	**Parâmetro**	**Descrição**
DC X (*Define Constant*)	Apenas númerico.	Aloca espaço para um valor X na posição de memória especificada.
DS X (*Define Storage*)	Apenas numérico.	Aloca X espaços da memória.
DA X (*Define Address*)	Apenas simbólico.	Armazena o endereço absoluto da variável X. É utilizado para passagem de parâmetros em subprogramas.
END	Não se aplica.	Indica término das linhas de código.

2.2 Montador S2

Na sequência dos módulos, se encontra o montador (*assembler*), o qual foi intitulado S2. Este módulo é responsável por traduzir códigos LS2 em um código intermediário, dito objeto, o qual possui, além de seu código de máquina gerado, informações que auxiliarão o *linker* a gerar o código executável final, uma vez que podem existir subprogramas em arquivos separados. O processo de tradução é realizado em duas etapas, as quais são descritas em seguida.

Para a tradução de cada código LS2, é necessário criar uma tabela de símbolos. Isso ocorre para que seja possível relacionar uma *label* ao seu endereço de memória, de forma que referências a ela (*jumps*, endereço de dados, chamada de subprogramas, etc) sejam traduzidas em código válido. Assim, a primeira etapa de tradução é responsável por ler todo o arquivo de entrada simulando o PC, e a cada definição de *label* encontrada, seu endereço é armazenado em uma tabela de símbolos.

A segunda etapa do processo é a tradução propriamente dita. Primeiramente, cada instrução lida é convertida em seu código de operação (*opcode*). Caso a instrução possua parâmetro (em ADD A, por exemplo, a instrução é ADD e a *label* – parâmetro – utilizada é A), é realizada uma busca na tabela de símbolos montada anteriormente para obter seu endereço. Tal instrução traduzida, a qual é da forma <*opcode*> <*endereço*>, é então escrita no arquivo de saída. Entretanto, caso a instrução seja CAL S e a *label* S não existir no arquivo, a tradução realizada é *! S*, onde o símbolo ! indica que se trata de uma referência a um subprograma externo, cuja *label* é S (isso é necessário para que o *linker* faça a ligação dos códigos posteriormente). Vale ressaltar, ainda, que, se a instrução não possuir parâmetro (HLT, LAX, SAX, RET, LAI e SAI), o campo de endereço é zerado, de forma a manter o padrão de dois inteiros por instrução. Por fim, caso seja encontrada uma pseudo-instrução, o tratamento é o seguinte:

- **DC X**: é escrito no arquivo de saída a constante X. O montador indica ao *linker* que se trata da pseudo-instrução DC através do símbolo &, para que a impressão do *linker* contenha a mesma formatação de dois inteiros por linha apenas para instruções. O conteúdo final é da seguinte forma: *& X*.
- **DS X**: são alocados X espaços de memória, sendo escrito no arquivo de saída uma sequência de X zeros.
- **DA X**: é escrito no arquivo de saída o endereço de X, o qual é obtido através de uma consulta à tabela de símbolos. Entretanto, é necessário indicar que se trata de um endereço absoluto, pois o *linker* precisará de tal informação para fazer a ligação corretamente. Isso indicação é feita utilizando um asterísco, sendo o conteúdo final da seguinte forma: ** X*.
- **END**: nada é escrito no arquivo de saída, pois essa pseudo-instrução apenas indica o fim do código.

Por fim, é adicionado um cabeçalho contendo a tabela de símbolos construída na primeira etapa. Isso é feito para auxiliar o *linker* a montar sua tabela de símbolos global, isto é, unificada de todos os arquivos de código. Além disso, a primeira linha do cabeçalho possui um único inteiro, indicando o *offset* do arquivo, o qual representa o deslocamento que o *linker* precisará fazer para obter os endereços corretos da tabela de símbolos, uma vez que a tabela agora deverá ser baseada em todos os arquivos e não apenas em um, como ocorre no montador. O término do cabeçalho é indicado com uma única linha contendo o símbolo #. Para melhor visualização de todo o processo, a Figura 2 ilustra a montagem de um código LS2 que calcula o dobro de um número através do subprograma S, o qual é definido em um arquivo separado.

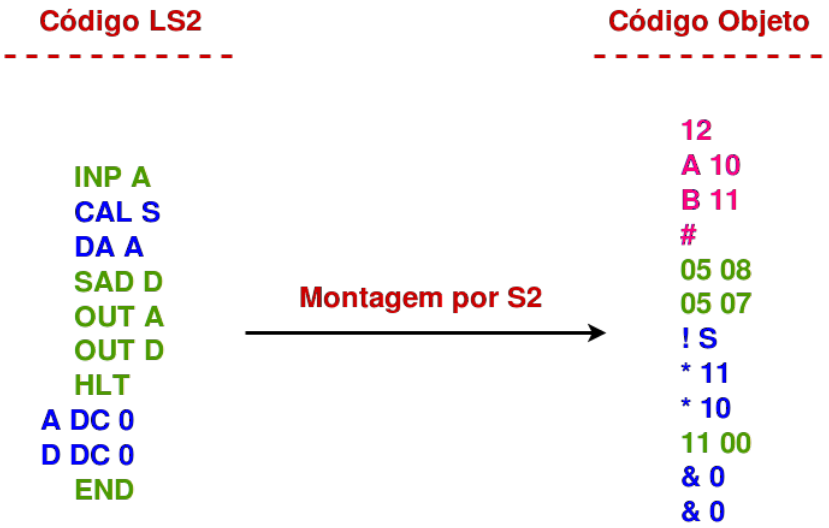

Figura 2 – Exemplo de montagem de um código LS2 com referência externa

.

2.3 Editor de Ligação L2

O editor de ligação (*linker*) tem como função ligar os códigos gerados pelo montador, uma vez que um programa pode ser escrito utilizando subprogramas definidos em arquivos separados. Assim, seu trabalho é *(i)*: relocação, ou seja, conserto de endereços absolutos, uma vez que esses não serão preservados quando rearranjados em um único arquivo; e *(ii)* resolução de referências, isto é, interpretar referências a subprogramas externos e escrevê-las corretamente no arquivo de saída, permitindo assim a chamada a subprogramas em módulos distintos. Vale ressaltar, neste ponto, que a sequência de ligação seguida pelo *linker* é com base na ordem dos parâmetros passados na execução, ou seja, os arquivos são ligados da esquerda para a direita.

Para que as tarefas *(i)* e *(ii)* sejam efetuadas, o *linker* utiliza o cabeçalho definido pelo montador, como apresentado anteriormente. Assim, a primeira etapa do *linker* é ler o cabeçalho de todos os arquivos e montar a tabela de símbolos geral. Além disso, para cada código objeto, é associado seu *offset*, o qual é obtido no cabeçalho, e um código identificador (*ID*) do arquivo. Isso será necessário para a obtenção correta do *offset* durante a ligação.

Após isso, o código (posterior ao cabeçalho) é lido. Se a linha lida não possuir referências a procedimentos e referências de memória, ela é simplesmente copiada para o arquivo final, pois se trata de endereçamento relativo ao PC, o qual já estará correto quando o programa for carregado

em memória. Entretanto, para resolver o item (i), isto é, a relocação, o *linker* identifica que se trata de um endereço absoluto através do símbolo *, o qual foi previamente colocado pelo montador, e então o endereço escrito no arquivo final será $address = address + load\_address + link\_offset$, onde *address* é o endereço obtido pela tabela de símbolos, *load_ address* é o endereço de carga recebido pelo *linker* e *link_ offset* é o *offset* cumulativo do primeiro arquivo até aquele que contém a definição do endereço absoluto em questão. Por fim, para resolver o item *(ii)*, que trata a questão de referências externas, é utilizado o sinalizador cujo símbolo é !, o qual também foi previamente colocado no código na etapa de montagem. Assim, o endereço *address* de tal procedimento é consultado na tabela de símbolos e é ajustado através da seguinte expressão: $address = address + load\_address + link\_offset - PC$. De forma a proporcionar uma melhor visualização do processo, a Figura 3 ilustra a ligação do código objeto apresentado pela Figura 2 com o código objeto do subprograma da Figura 3.

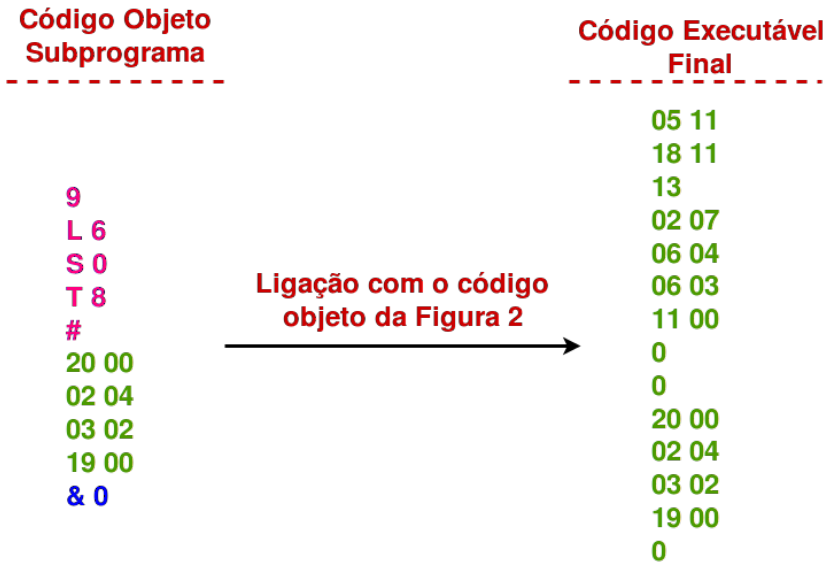

Figura 3 – Exemplo de montagem de um código LS2 com referência externa

.

2.4 Processador de Macros M4

A última etapa do trabalho consiste em criar um conjunto de macros de forma a proporcionar a escrita de códigos LS2 em estilo C. Para tanto, foi utilizado o processador de macros *GNU M4*. A Tabela 3 apresenta os comandos da pseudo-linguagem definida.

Grande parte das macros foram criadas utilizando a diretiva *define*, a qual faz uma expansão direta de uma *string* para outra. Entretanto, para tratar as macros *se* e *para*, é necessária a utilização de mais *labels* distintas para que seja possível fazer *jumps*, de forma a garantir que tais macros possam ser utilizadas mais de uma vez sem comprometer o código com ambiguidade de *labels*. Isso é feito através da utilização de *pushdef* e *popdef* dentro da expansão da macro, permitindo assim empilhar e desempilhar uma outra definição. Para tanto, foi definido que as *labels* de *a-z* serão exclusivas do processador de macros, isto é, o usuário, ao utilizar a pseudo-linguagem criada, só poderá utilizar *labels* compreendidas entre *A-Z*. Isso é necessário devido ao fato de algumas expansões de macros precisarem utilizar *labels* como forma de escape, como é o caso das macros *begin*, *se* e *para*.

Tabela 3 – Comandos da pseudo-linguagem

Rótulo	**Descrição**
programa	Indica o início do código.
end_programa	Indica o fim do código.
begin	Indica o fim das declarações de variáveis e o começo das instruções.
int(A)	Declara um inteiro com valor inicial 0.
ler(A)	Lê um valor inteiro e armazena em A (*stdin*).
escrever(A)	Imprime o valor de A (*stdout*).
soma(C, A, B)	C = A + B.
sub(C, A, B)	C = A - B.
inc(A)	Incrementa a variável A.
dec(A)	decrementa a variável A.
se(A, *condição*, B)	Aplica um operador condicional sobre as variáveis A e B. A condição pode ser *menor, menor_igual, maior, maior_igual* ou *igual*.
end_se	Indica o término da subrotina condicional criada pela macro *se*.
para(A, *Início*, *Fim*)	Cria um laço de repetição variando A do valor inicial ao final.
end_para	Indica o término da subrotina do laço de repetição criado pela macro *para*.

3 Conclusão

Este trabalho apresentou o desenvolvimento de um mini-sistema de programação, o qual é composto pelas etpas de montagem, ligação, simulação e a definição de uma pseudo-linguagem estilo C. Com isso, ficou clara a importância de cada etapa do processo de compilação de códigos e todo o funcionamento do proceso de cada módulo em individual. Além disso, a utilização do processador de macros *GNU M4* foi proveitosa no contexto de aprendizado de uma nova ferramenta, além de proporcionar uma melhor compreensão acerca da expansão de macros.

4 Referências Bibliográficas

Tanenbaum, Andrew S.; Austin, Todd. Structured Computer Organization, 6th ed. Prentice Hall, 2012.

GNU M4 Manual. GNU M4 1.4.18 macro processor. Acesso em: 22 set 2017. Disponível em: <https://www.gnu.org/software/m4/manual/m4.html>.

ZIVIANI, Nivio. Projeto de Algoritmos com implementações em PASCAL e C, 3 ed. Cengage Learning, 2011.