Universidade de Brasília - UnB
Faculdade de Tecnologia - FT
Departamento de Engenharia Elétrica - ENE

Aplicações em Comunicações Móveis

Desenvolvimento de um Programa Montador para um Processador RISC 16 bits

Relatório Final – Estágio supervisionado II

Aluno de projeto final de graduação
Ryan Martins Dias Rangel

Orientador
Prof. José Camargo da Costa

Brasília, Setembro de 2002

Índice

Índice de Figuras

Índice de Tabelas

Índice de códigos-fonte apresentados

1. Introdução

A utilização de circuitos integrados para aplicações em telecomunicações tem crescido de forma sensível nas últimas décadas. Com o aumento da densidade de integração e miniaturização, milhares de componentes que antes ocupavam uma placa inteira são colocados, através dos recursos da microeletrônica, em pequenos Chips, com milhões de minúsculos transistores.

Os computadores executam suas funções em linguagem de máquina (binária), o que dificulta em muito a utilização pelo ser humano. Como exemplo, seja o processador RISC de 16 bits de palavra, como é o caso deste projeto, para um programa de 1000 instruções seria necessário escrever 16000 zeros e uns, o que além de ser extremamente enfadonho, implica numa possibilidade enorme de erro. Dentro deste universo, a fim de se utilizar as funcionalidades e potencialidades existentes no processador, faz-se necessária uma interface entre o CI e o usuário, mas não em forma de linguagem de máquina.

Propõe-se, então, neste projeto final de graduação em Engenharia Elétrica, a construção de um software que utilize uma linguagem mais amigável para o usuário, mas que forneça os dados em linguagem de máquina para o processador, ou seja, ele deve ser um tradutor de código.

O ponto de partida para cumprir esta tarefa foi a especificação do hardware e a partir dele, a definição do conjunto de instruções existentes no processador, sendo todos estes tópicos de definição de objetivos a serem alcançados pelo sistema e especificação do projeto apresentados no capítulo 4 deste trabalho, em conjunto com a revisão bibliográfica, onde são apresentados os tópicos e referências para que seja desenvolvido um projeto da natureza deste.

Outro importante fator em empreendimentos que requerem a utilização de tecnologias é o modo como o projeto deve desenvolver-se, suas estratégias para a obtenção de todos os requisitos pré-definidos. Por isso, é de suma importância

ao leitor deste texto observar quais foram as metodologias utilizadas neste projeto, sendo as mesmas apresentadas no capítulo 5.

O capítulo 6 apresenta os resultados obtidos durante o projeto do programa Montador – v.1.1, onde são destacados os algoritmos essenciais ao seu funcionamento e as propriedades e facilidades que o programa apresenta para tornar o trabalho de geração de código mais produtivo.

O documento encerra-se comentando as possibilidades de otimização do programa e o que mais pode ser criado, para que em conjunto com o montador, possam ser desenvolvidas aplicações mais complexas que utilizem o sistema de comunicações sem fio projetado na Universidade de Brasília.

2. Resumo

Com a evolução de tecnologias e a utilização cada vez maior da microeletrônica em sistemas de comunicação, surge a necessidade da utilização de processadores com o máximo de funcionalidades integradas em seu hardware. Contudo, a complexidade no hardware leva a um preço final maior, e o como nos dias atuais existem compiladores de linguagem poderosos, transfere-se as instruções mais complexas para o nível de software. O processador em análise segue esta metodologia, que é parte da definição de um processador RISC e no caso em particular, com palavras de 16 bits. Para a conversão de uma linguagem de alto-nível para binário (linguagem de máquina), dentro da hierarquia de softwares de sistema, insere-se o montador, programa que converte o código fonte, em assembly, para o código objeto (linguagem de máquina), sendo este o alvo do projeto final de graduação em engenharia elétrica.

3. Palavras-chave

Processador, RISC, Montador, Assembly, C++, Software de Sistema, Micro-instruções, Pseudo-instruções.

4. Formulação do problema

O capítulo começa com a descrição dos requisitos teóricos para se implementar o programa através da revisão bibliográfica. Passa-se então para a apresentação em detalhes do sistema de comunicação sem fio no qual o software insere-se. Após isto, são mostrados os requisitos que o software deve obedecer para operar em conjunto com o processador.

4.1. Revisão bibliográfica

Objetivando-se a construção de um programa montador, é indicado ao projetista o conhecimento das seguintes áreas (com referências recomendadas):

- Arquitetura de processadores – referências [1, 2 e 3];
- Softwares de sistema – referência [4];
- Linguagem de programação (recomenda-se a utilização de C++) – referências [5, 6 e 7];
- Descrição de projetos similares – referências [8, 9 e10].

4.2. Descrição do sistema de comunicações sem fio

O projeto sobre comunicações móveis propõe a implementação, em um circuito monolítico, utilizando-se tecnologia CMOS, de um transceptor de RF operando na faixa de freqüências da 902-928 MHz, cujas aplicações, segundo a ANATEL, devem ser de equipamentos de radiação restrita que utilizam técnicas de espalhamento espectral nos serviços fixos e móveis (Resolução n° 409 de

14.01.00); associado a um processador digital com memória residente e interfaces de comunicação. Uma estrutura desta natureza presta-se à aplicações em telemetria, redes locais sem fio, telefonia, sensoreamento entre outras. Estão sendo desenvolvidas células básicas, incluindo dispositivos ativos e passivos (reativos ou não), necessários para a implementação das estruturas de modulação, demodulação, filtragem, equalização, etc. O projeto tem como objetivos a especificação, desenvolvimento, implementação (construção do processador em uma indústria do exterior, com tecnologia capaz de suprir as necessidades do projeto) e teste.

O hardware do sistema pode ser visualizado na Figura 4.2.a. É composto por uma placa de circuito impresso, o qual possui numa face o circuito integrado que realiza a comunicação e os circuitos de apoio, servindo também de substrato para os outros componentes que integram o sistema.

A tecnologia empregada para a fabricação do CI deverá ser a AMS 0.8 μm CMOS, dois níveis de metal e silício poli-cristalino. A faixa de freqüências já foi especificada anteriormente. Com a utilização da modulação MSK esperam-se taxas de transmissão entre 125 e 256 kbps e alcances de até 100 metros confinados numa edificação com emissão de potência de até 100 mW. A unidade digital de processamentos possui um microprocessador de 16 bits operando a 250 MHz, com 8 kB de memória SRAM para instruções, dados e pilha, cujo endereço de inicialização é #0001h; uma interface serial padrão RS – 232 e uma interface sigma-delta. O processador deve realizar a aquisição, tratamento e envio dos dados, além do gerenciamento da transmissão e controle do enlace de comunicação. A estrutura de I/O é apresentada no Anexo, páginas 2 e 3.

Além de conectores e cabos para o funcionamento do sistema, a placa também possui dispositivos para auxiliar o funcionamento do CI de comunicação, abrangendo um regulador de tensão (CI 7803 e dispositivos passivos), um driver para a interface serial (CI MAX 3232 ou MAX 3431), 4 capacitores de 1μF e um transformador de impedância (balun) para a ligação da antena ao chip.

O processador pode ser programado para uma tarefa específica, através de instruções pré-definidas, recebendo os dados, processando e enviando as informações resultantes. O conjunto de instruções é definido a partir da arquitetura escolhida para o processador. O diagrama de blocos do sistema é apresentado na Figura 4.2.b.

O projeto deve consumir baixa potência e possuir área reduzida, através de uma tensão baixa, mas sem comprometer a velocidade de processamento. O tamanho da palavra e o número de registradores deve ser suficiente para que as aplicações sejam desenvolvidas sem o comprometimento da taxa de transferência ou estender os limites da área.

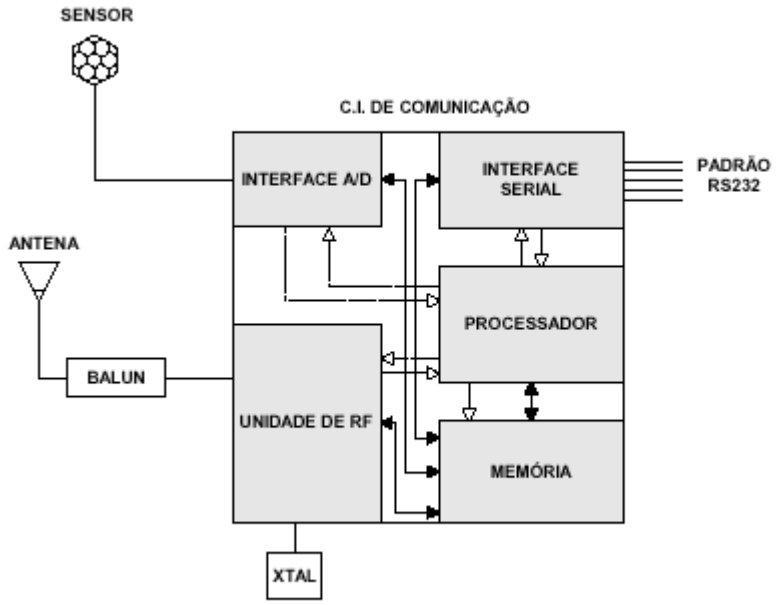
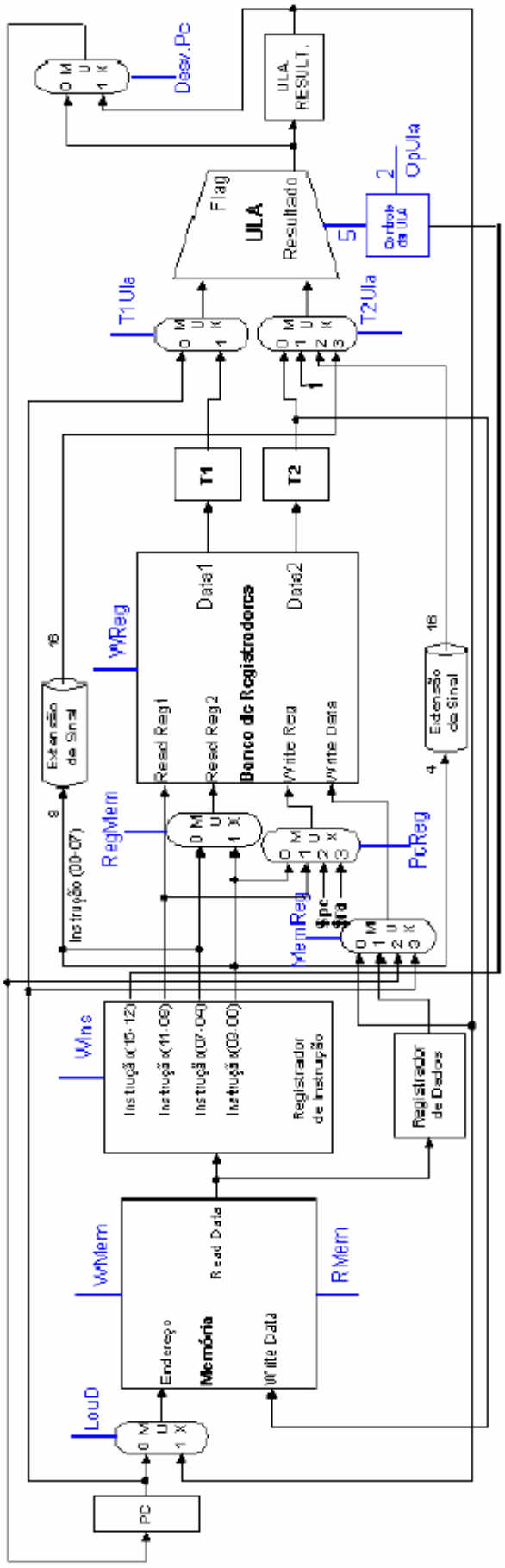


Figura 4.2.a – Hardware do sistema de comunicação sem fio


Desv.Pc
ULA RESULT.
Flag
ULA
Resultado
OpUla
Controle da ULA
T1Ula
T2Ula
T1
T2
Data1
Data2
Banco de Registradores
Read Reg1
Read Reg2
Write Reg
Write Data
WReg
RegMem
PcReg
MemReg
Extensão de Sinal
Instrução (00-07)
Instrução (15-12)
Instrução (11-08)
Instrução (07-04)
Instrução (03-00)
Registrador de Instrução
WIns
Registrador de Dados
WMem
RMem
Endereço
Memória
Read Data
Write Data
LouD
PC

4.3. Definição da arquitetura do processador

A possibilidade para a escolha de arquitetura mais comum e sobre quem nos deteremos é: CISC x RISC.

4.3.1. CISC

O padrão CISC (Computador com Conjunto de Instruções Complexas) foi desenvolvido nas décadas de 60 e 70, onde cria-se que o custo do hardware diminuiria com o tempo devido a aceleração da produção em massa de componentes eletrônicos com relativa confiabilidade e em grande quantidade. Por outro lado, como a construção de softwares dependia de mão-de-obra altamente qualificada, a tendência natural seria a de um aumento nos custos de manutenção de tais profissionais. Esta conjuntura, deslocou a implementação de instruções complexas do software para o hardware, o que implicou na necessidade de chips maiores para abrigarem todas as operações necessárias, pois as instruções eram decodificadas através dos microcódigos implementados em ROM, o que dificultava em muito sua execução em paralelo. Outro problema era que para instruções complexas exigia-se um elaborado e grande sistema de controle, chegando algumas vezes a serem gastos 70% da área do CI para área de controle e 30% para a execução das instruções.

Antes da década de sessenta, os softwares eram raramente programados em linguagem de alto nível, daí advém a necessidade de tão grande especialização citada anteriormente. Para simplificar a construção de programas para as máquinas CISC, pesquisadores sugeriram que a linguagem utilizada fosse a mais próxima possível das linguagem de alto-nível, de onde veio a HLLCA – Arquitetura Computacional de Linguagem de Alto Nível, onde as instruções eram quase que totalmente implementadas em hardware, visando diminuir assim os custos de implementação dos processadores.

Mesmo sem ser o principal objetivo, a arquitetura CISC obteve uma melhora no desempenho da máquina pois possui um conjunto de instruções complexas em hardware, o que levaria mais tempo para executá-las se estivessem em software.

Nesta arquitetura, o modo de endereçamento é memória-memória, o que permite operações diretamente na memória e desta forma reduz o número de transistores necessários para a implementação dos registradores.

4.3.2. RISC

Devido ao desenvolvimento de linguagens de programação mais eficientes e novas técnicas de projeto de compiladores, não era mais necessário um hardware tão complexo como o existente nas máquinas CISC, o que implica numa diminuição direta do tamanho do chip e transferência do suporte à HLLCA para o software.

A filosofia de máquinas voltadas para linguagens de alto nível é então substituída pela RISC - Computador de Conjunto de Instruções Reduzido - reduzido pois foram retiradas do hardware todas as instruções complexas

dispensáveis, sendo as mesmas trabalhadas em nível de software, buscando-se assim aumentar a velocidade do caso comum pois seriam executadas no hardware instruções pequenas e rápidas. Praticamente todos os conjuntos de instruções desenvolvidos em meados da década de 80 seguem esta arquitetura. Ela possui instruções que executam apenas as funções básicas do montador, e quando combinadas geram as instruções complexas do CISC. O acesso a memória é realizado através de load-store, o que implica que todas as instruções são baseadas em registradores, gerando um maior gasto em transistores pela necessidade de um número maior de registradores mas utiliza a técnica de pipeline, que permite a sobreposição temporal das diversas fases da execução de um programa afim de se reduzir o número de ciclos por instrução (no caso, pseudo-instrução).

A arquitetura escolhida para a implementação do processador foi a RISC, principalmente devido a cinco características desta filosofia que são vantagens em relação a CISC:

- Conjunto de instruções simples: o que permite uma configuração mais simples de hardware. As instruções complexas são construída através de software como sendo a combinação das instruções simples;

- Utilização de pipeline: permite o processamento de mais instruções em menos tempo, pois não é necessário terminar uma instrução para se começar outra (no caso de pseudo-instruções);

- Velocidade maior de processamento: como as instruções são simples, com a utilização do pipeline, os processadores RISC conseguem uma performance neste quesito de 2 a 4 vezes maior que os CISC, utilizando-se tecnologias de fabricação de chip similares e mesma freqüência de clock;

- Hardware mais simples: se é mais simples, consequentemente é mais barato, além de diminuir-se a utilização do espaço do chip o que permite a colocação de funções extras no processador, como por exemplo, operações em ponto-flutuante;

- Ciclo de projeto: como a arquitetura é mais simples que a CISC, as melhorias tecnológicas podem ser incorporadas num menor espaço de tempo que a filosofia concorrente, o que implica numa maior velocidade de lançamento de produtos novos e diversificados no mercado.

A partir da escolha da arquitetura a ser utilizada, passa-se então a descrição dos registradores a serem utilizados, das micro-instruções (implementadas em hardware) e das pseudo-instruções (implementadas em software).

4.4. O processador como máquina de 16 bits

Historicamente, tem-se que a utilização de máquinas de 16 bits foi realizada em maior escala que as de 8-bits. Porém, com o aumento dos requisitos de softwares modernos, sobretudo os que envolvem processamento de imagens, surgiu a necessidade de sistemas de 32 e até 64 bits. Contudo, uma arquitetura com mais de 16 bits, apesar de ser muito poderosa, implica num hardware muito complicado, e de acordo com as propostas para o sistema de comunicação sem fio em desenvolvimento de utilização e requisitos de tecnologia, os 16 bits seriam suficientes para implementar as instruções necessárias a uma operação satisfatória do sistema. Além disso, conforme dito anteriormente, uma das características utilizadas para a escolha da arquitetura RISC foi a simplicidade, e este mesmo princípio será utilizado quanto a questão do tamanho de instruções, visto que como a complexidade das instruções foi transferida para o software não há necessidade de se criar um hardware tão grande e complexo. Poder-se-ia colocar 16 bits no hardware e instruções de 32 bits, mas isso influiria na velocidade, pois seriam necessários dois ciclos para se executar uma instrução, o que leva a uma escolha de hardware e instruções de 16 bits.

4.5. Registradores

Registradores são parte do hardware da máquina utilizados para armazenar valores de interesse, sobretudo em operações aritméticas. São formados por unidades de memória, como por exemplo, latches, sendo controlados pelos pulsos do clock do processador. Também podem ser utilizados flip-flops para esta função de registro de palavras.

Os registradores que comporão o chip possuem 16 bits, sendo que o seu número limitado restringe as possibilidades de instruções, pois alguns serão utilizados no próprio código de tratamento das pseudo-instruções.

O projeto possui 16 registradores, sendo para tanto, necessários 4 bits (faixa de 0000 – 1111) para a codificação destes componentes. Desta forma, os campos correspondentes a registradores ocupam 4 bits cada dentro da palavra de instrução (16 bits). Os registradores foram caracterizados e descritos nas referências [8 e 9], sendo exibidos na Tabela 4.5.

Código	Símbolo	Significado	Observações
0000	$zero	Constante zero	Na verdade não é um registrador, mas apenas linhas de dados aterradas para sinalizar zero.
0001	$t0	Temporários	Usados para auxiliar o cálculo.
0010	$t1		
0011	$t2		
0100	$a0	Argumento	Recebe os argumentos para cálculo.
0101	$a1		
0110	$a2		
0111	$s0	Salvos	Guarda os resultados.
1000	$s1		
1001	$s2		
1010	$s3		
1011	$int	Cód. de interrupção	Armazena o código de erro gerado pelo processador.
1100	$gp	Apontador global	Aponta o topo da pilha geral
1101	$sp	Apontador de pilha	Aponta o topo da pilha de dados
1110	$pc	Contador do programa	Contém o número da linha do programa que está sendo executada.
1111	$ra	Endereço de retorno	Contém o endereço de retorno de uma sub-rotina.

Tabela 4.5 – Registradores do processador

4.6. O montador

Conforme mostrado anteriormente, na implementação de arquitetura RISC a complexidade das instruções é trabalhada em nível de software, o que cria a necessidade de um programa que converta de uma linguagem de mais alto nível que a de máquina para a linguagem de máquina verificando quaisquer erros de sintaxe para enviar o código, devidamente montado, para o microprocessador.

Pode-se mostrar a execução de um programa de acordo com a Figura 4.6.

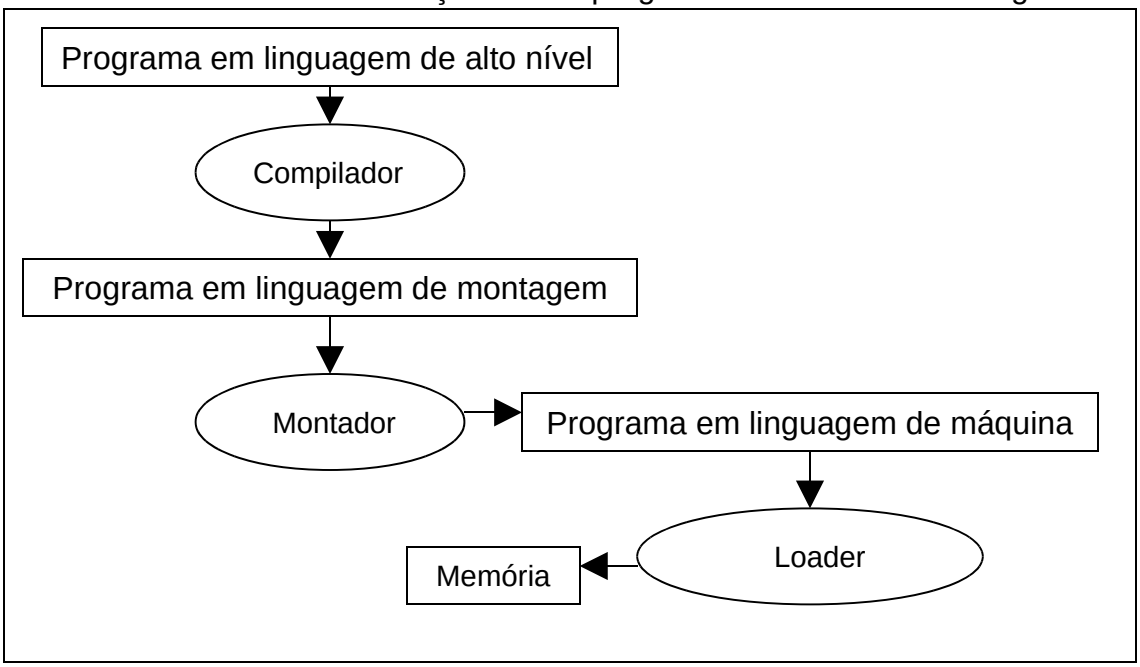


Figura 4.6 – A hierarquia da tradução do programa

O compilador é o programa que traduz a linguagem de alto nível para uma forma simbólica que se aproxima mais da linguagem de máquina. A linguagem de alto nível, normalmente, precisa de menos linhas para expressar uma determinada função, em relação a quantidade exigida em nível do código de montagem, o que a torna uma programação mais eficiente, porém exige o tempo de compilação durante a execução do que foi programado para a tradução desta linguagem para o código de montagem. Porém, as questões de desempenho devem ser estudadas caso a caso. Para este projeto, o compilador é opcional, sendo necessário somente se as especificações da aplicação onde o sistema será utilizado requiser níveis de abstração maiores.

O montador está um nível abaixo do compilador, e utiliza como fonte o código gerado pelo compilador (de montagem) e transforma em linguagem de máquina. Esta é a importância do montador no processo, e por isso ele se torna uma peça relevante para programação de microprocessadores. Neste nível, as pseudo-instruções são transformada nas micro-instruções correspondentes e tanto micro como pseudo-instruções são traduzidas em linguagem de máquina, por exemplo, seja a instrução – Add $t0,$t1,$t2 – a mesma será convertida, de acordo com a tabela guardada como referência de valores no montador, em 0010000100100011, ou em hexadecimal, 2123, linguagem que a máquina

consegue interpretar. O arquivo gerado pela conversão de todas as instruções pode ser ainda codificado pelo montador e enviado para o próximo estágio.

O código devidamente traduzido é então conduzido ao *loader* (carregador) que aloca o espaço necessário para o código e os copia para a memória, fazendo as inicializações necessárias para que o programa possa ser executado pelo processador. Os próximos dois tópicos descrevem as unidades fundamentais utilizadas como fontes para o montador, as micro e pseudo-instruções, sendo apresentado também o conjunto de instruções utilizadas pelo processador.

4.6.1. As micro-instruções

No processador RISC, como o caso em estudo, as micro-instruções serão implementadas no hardware da máquina. De acordo com as especificações as instruções possuem 16 bits, e são divididas em três tipos, de acordo com os campos e funções pertencentes a instrução:

R – com três registradores;
I – com dois registradores e uma constante de 4 bits;
J – com um registrador e uma constante de 8 bits.

4.6.1.a. Instruções do tipo R

Operação	Reg. Origem	Reg. Origem 2	Reg. Destino
4 bits	4 bits	4 bits	4 bits

Figura 4.6.1.a – Instrução tipo R

Conforme mostrado na Figura 4.6.1.a as instruções do tipo R são registrador - registrador, pois possuem dois registradores como operandos (Reg.Origem e Reg.Origem 2) armazenando o resultado também em um registrador. Desta forma, conforme decidido para a codificação dos registradores, tem-se que eles ocupam 12 bits, restando 4 bits para o operando da instrução.

4.6.1.b. Instruções do tipo I

Operação	Reg. Origem	Reg. Origem 2	Deslocamento
4 bits	4 bits	4 bits	4 bits

Figura 4.6.1.b – Instrução tipo I

As instruções do tipo I realizam operações de comparação entre dois registradores desviando para o valor de deslocamento caso a comparação for verdadeira.

4.6.1.c. Instruções do tipo J

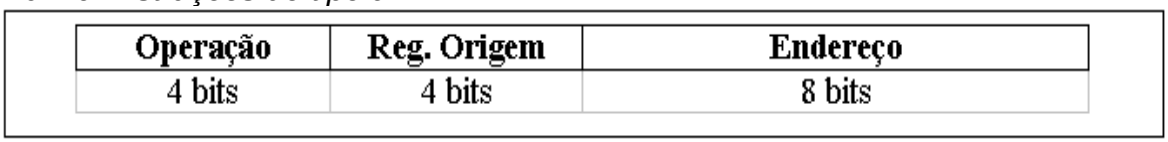


Figura 4.6.1.c – Instrução tipo J

As instruções do tipo J relacionam um registrador com uma constante a fim de se obter desvio para um endereço final, uma operação aritmética com um dos operandos sendo o registrador de origem o outro o campo endereço e o resultado sendo armazenado no próprio registrador ou então uma manipulação de deslocamento dos bits armazenados no registrador. As micro-instruções escolhidas estão na Tabela 4.6.1.

Código	Categ.	Instrução	Exemplo	Significado	F
0010	Aritmética	Add	Add $s0,$t0,$t1	$s0 = $t0 + $t1	R
0011		Sub	Sub $s0,$t0,$t1	$s0 = $t0 - $t1	R
1000		Addi	Addi $t0,45	Soma $t0 com a constante e armazena em $t0	J
1001	Lógica	Shift	Sft $t0,3	Desloca $t0 do valor da constante. Se a constante for positiva, desloca para a direita, se negativa, para a esquerda.	J
0100		And	And $s0,$s1,$s2	AND booleano bit a bit entre $s1 e $s2. Resultado armazenado em $s0	R
0101		Or	Or $s0,$s1,$s2	OR booleano bit a bit entre $s1 e $s2. Resultado armazenado em $s0	R
1010		Not	Not $s0	NOT booleano bit a bit de $s0. Resultado armazenado em $s0	J
0110		Xor	Xor $s0,$s1,$s2	XOR booleano bit a bit entre $s1 e $s2. Resultado armazenado em $s0	R
0000		Lw	Lw $s0,$s1,$s2	Carrega a palavra armazenada no endereço de $1,	R

	Transferência			deslocado de $s2 e guarda em $s0.	
0001		Sw	Sw $s0,$s1,$s2	Carrega a palavra armazenada em $s0 e guarda no endereço $s1 deslocado de $s2.	R
1011		Lui	Lui $s0,45	Carrega a constante nos oito bits mais significativos de $s0.	J
0111	Desvio condicional	Slt	Slt $s0,$s1,$s2	$s0 = 1, se $s1<$s2 senão, $s0 = 0.	R
1100		Beq	Beq $s0,$s1,3	Se $s0 = $s1, desvia para PC constante.	I
1101		Blt	Blt $s0,$s1,2	Se $s0 < $s1, desvia para PC constante.	I
1110	Desvio incondicional	J	J $s0,45	Desvia para o endereço de $s0 deslocado da constante.	J
1111		Jal	Jal $s0,45	Desvia para o endereço de $s0 deslocado da constante salvando origem em $ra.	J

Tabela 4.6.1 – Conjunto de micro-instruções do processador

4.6.2. Pseudo-instruções

Sendo o conjunto de instruções composto pelas micro-instruções (16), seu número é pequeno para aplicações mais complexas, sendo fundamental a criação das pseudo-instruções, as quais são combinações das micro-instruções, implementadas em software e que permitem a expansão do conjunto de instruções do microprocessador. O conjunto completo de pseudo-instruções inclui as micro-instruções, apesar de nesse caso, a pseudo ser a própria micro-instrução.

Como as pseudo-instruções são manipuladas e convertidas em micro-instruções em nível de software, não se está mais preso ao tamanho fixo de 16 bits para a instrução, desde que, ao final da conversão, somente existam micro-instruções de 16 bits. Assim sendo, foram escolhidas as seguintes pseudo-instruções para comporem o conjunto de instruções do chip:

Cat.	Instrução	Significado
Aritméticas	Add $t0,$t1,$t2	Adiciona $t1 a $t2 e armazena em $t0
	Sub $t0,$t1,$t2	Subtrai $t2 de $t1 e armazena em $t0
	Mul $t0,$t1,$t2	Multiplica $t1 e $t2 e armazena em $t0
	Div $t0,$t1,$t2	Divide $t1 por $t2 e armazena em $t0
	Addi $t0,34	Soma a constante a $t0 e armazena em $t0
	Subi $t0,34	Subtrai $t0 da constante e armazena em $t0
	Muli $s0,$s1,34	Multiplica $s1 pela constante e armazena em $s0
	Divi $a0,$a1,12	Divide $a1 pela constante e armazena em $a0
	Rem $t0,$t1	Armazena o resto da divisão de $t0 e $t1 em $t0

Tabela 4.6.2.a – Pseudo-instruções aritméticas

Cat.	Instrução	Significado
Transferência	Lw $s0,(3)$s1	Carrega em $s0 a palavra armazenada no endereço $s1 deslocado da constante
	Sw $s0,(3)$s1	Carrega no endereço $s1 deslocado da constante a palavra armazenada em $s0
	Lb $s0,(3)$s1	Carrega em $s0 o byte menos significativo do conteúdo do endereço ($s1 + constante)
	Sb $s0,(3)$s1	Salva o byte menos significativo de $s0 no endereço ($s1 + constante)
	Ld $s0,$s1,(3)$s2	Carrega em $s0 e $s1 as duas palavras armazenadas em $s2 deslocado da constante e o próximo endereço
	Sd $s0,$s1,(3)$s2	Salva o conteúdo de $s0 e $s1 no endereço $s2 deslocado da constante
	Lui $t0,45	Carrega a constante nos oito bits mais significativos de $t0
	Lwi $t0,345	Carrega a constante de 16 bits no registrador $t0
	Mov $t0,$t1	Move o conteúdo de $t0 para $t1
	Mflo $s1	Move o byte menos significativo de $s1 para o mais significativo
	Mfhi $s1	Move o byte mais significativo de $s1 para o menos significativo
	Move $t0,$t1	Move o conteúdo de $t1 para $t0 preenchendo $t1 com zeros
	Chg $t0,$t1	Troca o conteúdo dos dois registradores

Tabela 4.6.2.b – Pseudo-instruções de transferência

Cat.	Instrução	Significado

Lógica	Sftl $s0,5	Desloca a esquerda $s0, com a constante sendo o número de deslocamentos e armazena em $s0
	Sftr $s0,5	Desloca a direita $s0, com a constante sendo o número de deslocamentos e armazena em $s0
	And $s0,$s1,$s2	AND booleano bit a bit de $s1 com $s2 e armazena em $s0
	Or $s0,$s1,$s2	OR booleano bit a bit de $s1 com $s2 e armazena em $s0
	Nor $s0,$s1,$s2	NOR booleano bit a bit de $s1 com $s2 e armazena em $s0
	Xor $s0,$s1,$s2	XOR booleano bit a bit de $s1 com $s2 e armazena em $s0
	Not $s0	NOT booleano bit a bit de $s0 e armazena em $s0
	Comp $s0	Complemento a dois de $s0 e armazena em $s0
	Andi $t0,34	AND booleano bit a bit de $t0 com a constante e armazena em $t0
	Ori $t0,34	OR booleano bit a bit de $t0 com a constante e armazena em $t0
	Xori $t0,34	XOR booleano bit a bit de $t0 com a constante e armazena em $t0

Tabela 4.6.2.c – Pseudo-instruções lógicas

Cat.	Instrução	Significado
Desvio Condicional	Slt $s0,$s1,$s2	$s0=1 se $s1 < $s2, caso contrário $s0=0
	Sle $s0,$s1,$s2	$s0=1 se $s1 <= $s2, caso contrário $s0=0
	Seq $s0,$s1,$s2	$s0=1 se $s1 = $s2, caso contrário $s0=0
	Sne $s0,$s1,$s2	$s0=1 se $s1 <> $s2, caso contrário $s0=0
	Sgt $s0,$s1,$s2	$s0=1 se $s1 > $s2, caso contrário $s0=0
	Sge $s0,$s1,$s2	$s0=1 se $s1 => $s2, caso contrário $s0=0
	Beq $s0,$s1,5	Se $s0 = $s1, desvia para PC constante
	Bne $s0,$s1,5	Se $s0 <> $s1, desvia para PC constante
	Blt $s0,$s1,5	Se $s0 < $s1, desvia para PC constante
	Bgt $s0,$s1,5	Se $s0 > $s1, desvia para PC constante
	Slti $s0,$s1,5	$s0=1 se $s1 < constante, caso contrário $s0=0
	Seqi $s0,$s1,5	$s0=1 se $s1 = constante, caso contrário $s0=0
	Sgti $s0,$s1,5	$s0=1 se $s1 > constante, caso contrário $s0=0
	Beqi $s0,10,5	Se $s0 = constante1, desvia para PC constante2
	Bnei $s0,10,5	Se $s0 <> constante1, desvia para PC constante2
	Blti $s0,10,5	Se $s0 < constante1, desvia para PC constante2
	Bgti $s0,10,5	Se $s0 > constante1, desvia para PC constante2

Tabela 4.6.2.d – Pseudo-instruções de desvio condicional

Cat.	Instrução	Significado

Desvio Incondicional	J $s1,45	Desvia para endereço $s1 deslocado da constante de 8 bits
	Jr $s1	Desvia para endereço $s1
	Jpc 45	Desvia para endereço $pc deslocado da constante de oito bits
	Jal $s0,45	Desvia para endereço $s0 deslocado da constante de 8 bits salvando origem em $ra
	Jalr $s0,$s1,45	Desvia para endereço $s1 deslocado da constante de 8 bits salvando origem em $s0
	Jalpc $s0,45	Desvia para endereço $pc deslocado da constante de 8 bits salvando origem em $ra
	Jd 45	Desvia para o endereço da constante de 16 bits
	Jald $s0,45	Desvia para endereço $s0 deslocado da constante de 16 bits salvando origem em $ra
	Jalr $s0,$s1,45	Desvia para endereço $s1 deslocado da constante de 16 bits salvando origem em $s0
	Jalpcd 45	Desvia para endereço $pc deslocado da constante de 16 bits salvando origem em $ra

Tabela 4.6.2.e – Pseudo-instruções de desvio incondicional

5. Metodologia

Existem diversos mecanismos para a formulação de algoritmos em programação, mas a forma utilizada está descrita na Figura 5.1, onde é apresentado o fluxograma de desenvolvimento para um software como o Montador.

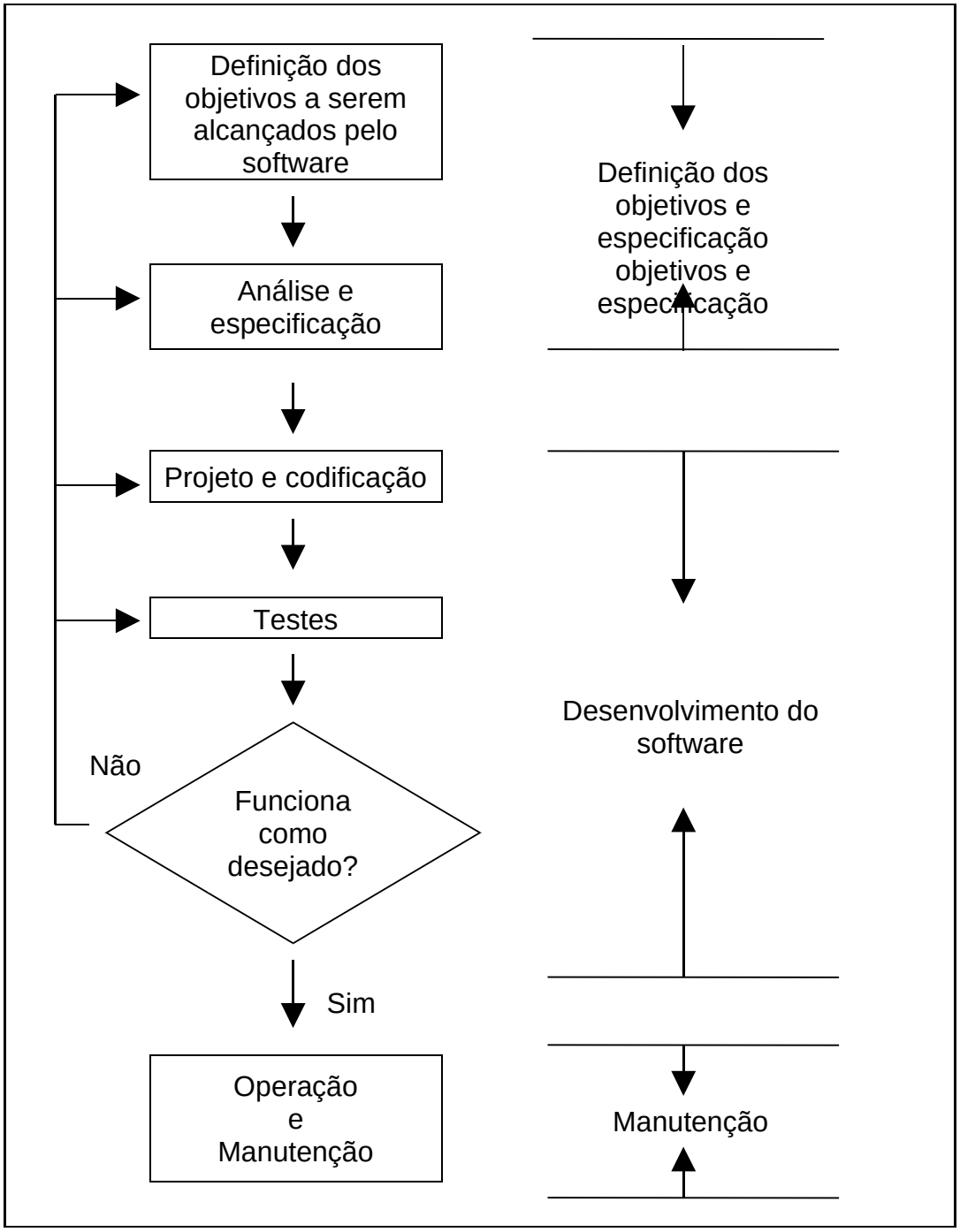


Figura 5.1 – Fluxograma da metodologia de construção de software

O projeto inicia-se com a definição dos objetivos a serem alcançados. É nesta fase que se dá a concepção do que o programa deve realizar e do que não é necessário ao mesmo. Cabe ressaltar que esta fase seria algo importante mas não fechado porque muitas das funcionalidades do programa podem vir de uma necessidade que se tornou relevante em outra etapa do projeto, que se necessário, pode ser repensado para adequar-se ao desejado. Seguindo-se a definição dos objetivos e análise, tem-se a especificação detalhada de todo o projeto, procurando-se clareza nas explicações, pois isso fará com que o trabalho desenvolva-se de maneira mais rápida e eficaz. Deve-se escolher também a linguagem que será utilizada e a metodologia de programação, se estruturada ou orientada a objeto. Outro fator importante é que se deve procurar colocar o maior número possível, se não todas, as funcionalidades que o programa requer nesta etapa, pois para inserções futuras pode haver problemas de compatibilidade de rotinas, principalmente se estiverem sendo utilizadas instruções que acessem muitas vezes a memória, pois basta um pequeno erro de alocação ou carregamento e o programa se perde em sua execução, sendo muitas vezes difícil encontrar-se o erro.

O próximo nível é o de desenvolvimento, onde o código do programa é gerado utilizando-se a linguagem escolhida e são realizados os testes funcionais. Caso o resultado não seja satisfatório, conforme mostrado no fluxograma, pode-se retornar para qualquer um dos níveis anteriores, sendo para isto necessário descobrir onde está o erro ou pode ser feita a melhoria. Algumas vezes é necessário voltar-se ao início para a correção de erros, o que implica que as etapas iniciais devem ser extremamente bem feitas pois assim não haverá a necessidade de se dispor de muito tempo para correções.

Caso o programa esteja funcionando conforme o esperado, passa-se ao nível de manutenção, onde o programa será utilizado pelo usuário, o qual sempre descobre novos erros e espera por funcionalidades que às vezes não estão presentes, o que faz com que sejam geradas novas versões dos programas.

5.1. Especificação do montador

Nos próximos tópicos serão examinadas as especificações referentes à implementação de um software montador, sendo o fluxograma correspondente:

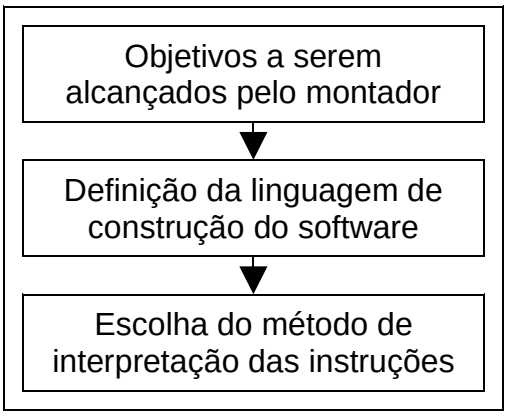


Figura 5.1.a – Especificação do montador

5.1.1. Objetivos a serem alcançados

O montador, conforme dito na seção 4.6 desta monografia, é o software de sistema responsável pela tradução de código mostrada na Figura 5.1.1.

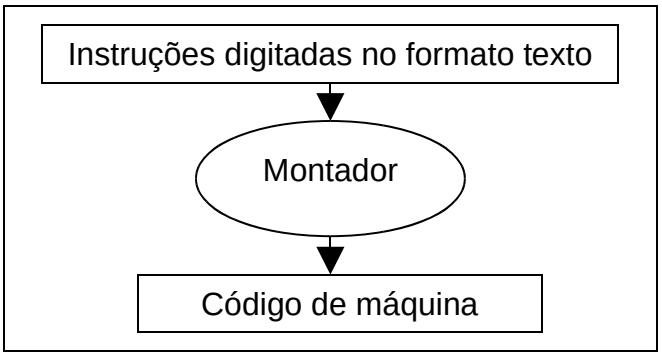


Figura 5.1.1 – Objetivo do montador

Para realizar a conversão do código, o montador deve ser eficiente em velocidade, sem gerar erros de tradução, pois estes poderiam ocasionar sérios danos no sistema onde o processador está inserido. Deve ser um programa capaz de detectar erros passíveis de serem cometidos pelo usuário e indicar a forma correta de utilização da sintaxe das instruções. A linguagem utilizada como fonte para a tradução foi a assembly de um processador RISC padrão, com adaptações para um funcionamento adequado ao hardware que será implementado, conforme mostrado na referência [08].

5.1.2 Definição da linguagem de programação

O programa montador é do tipo “cross-compiler”, pois deverá ser executado num PC e o código objeto gerado por seu algoritmo será carregado no processador RISC de 16 bits, externo ao computador. Assim sendo, seu tempo gasto em rotina de tradução de código deve ser pequeno mas sem implicar numa perda de segurança de que as informações resultantes são as esperadas.

Para a construção do software Montador – v.1.1 a fim de serem satisfeitas as considerações acima e pela vantagem de permitir programação estruturada e orientada a objeto, utilizou-se a linguagem de programação C++ e a plataforma Windows, sendo que o projeto da interface com o usuário procurou adaptá-la para que fosse o mais próximo possível de um editor de texto convencional. Outra linguagem que poderia ser escolhida é a JAVA, mas como possui seu tempo de compilação maior que o do C++, o que dificulta um pouco o desenvolvimento do software, pois a cada alteração perder-se-á um tempo considerável para que o programa seja executado pela primeira vez, e a exclusividade da orientação a objeto, apesar de possuir uma portabilidade maior que o C++ devido à máquina virtual gerada pelo compilador JAVA, não foi selecionada como especificação de linguagem.

5.1.3. Escolha do método de interpretação das instruções digitadas

Mesmo sendo componentes do conjunto de pseudo-instruções, as originariamente micro-instruções serão tratadas de forma distinta às pseudo-instruções pelas facilidades advindas desta escolha.

5.1.3.a. Micro-instruções

Para as micro-instruções, têm-se como opções de métodos de interpretação destas instruções:

- utilização da classificação de acordo com o formato de micro-instrução(R, I ou J);

- comparação entre a micro-instrução digitada e as instruções existentes como referência internamente no programa, uma a uma, até que seja encontrada, ou não em caso de erro, a desejada.

Escolheu-se a classificação de acordo com o tipo. No caso de micro-instruções que não se encaixam especificamente em um dos tipos apresentados na seção 4.6.1. deverá ser utilizada manipulação de tal forma que seja compatível com as especificações de formato de micro-instrução.

5.1.3.b. Pseudo-instruções

Para as pseudo-instruções, têm-se os seguintes métodos de interpretação:

- utilização das divisões de tipo de pseudo-instrução (como por exemplo, tipo aritmética);

- comparação uma a uma da pseudo-instrução digitada com as pseudo-instruções armazenadas como referência.

Escolheu-se a segunda opção, pois o conjunto de pseudo-instruções é extenso, mesmo com a retirada para fins de interpretação das micro-instruções, sendo desta forma mais fácil o isolamento de erros caso existam na compilação da pseudo-instrução. Como desvantagem, tem-se o aumento da extensão do código utilizado pois serão armazenadas as pseudo-instruções individualmente. Vale ressaltar que as pseudo-instruções poderiam ser tratadas como um conjunto de micro-instruções identificados e desta forma, separaram-se as micro-instruções as quais seriam tratadas como um conjunto de micros digitada pelo usuário. Porém, preferiu-se tratar a pseudo-instrução como se fosse uma única instrução, composta por módulos que correspondem às micro-instruções.

5.2. Procedimentos para a implementação de um programa montador

A partir das especificações, seguindo-se os próximos tópicos, torna-se possível a construção de um programa similar ao Montador – v.1.1. O fluxograma correspondente é apresentado na Figura 5.2.

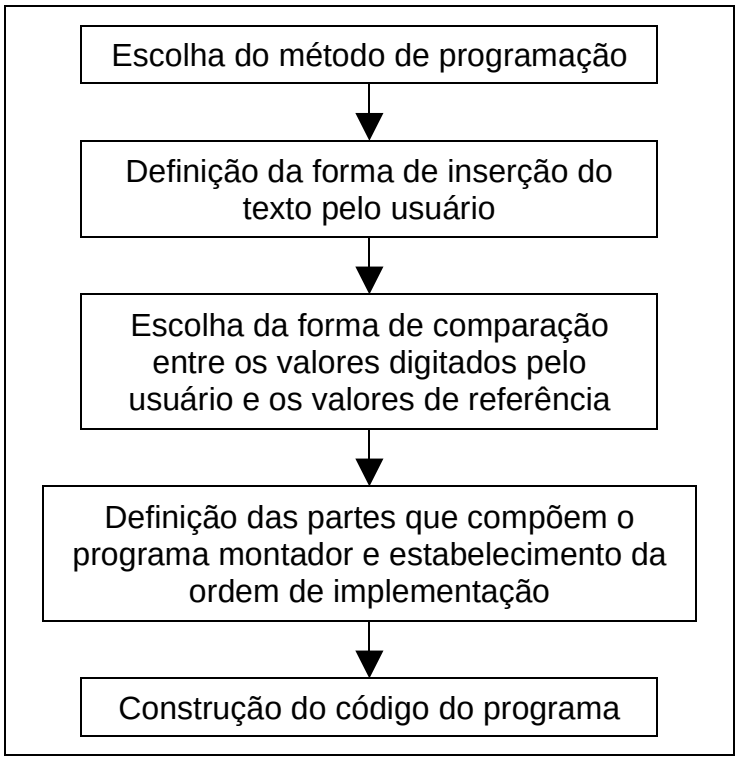


Figura 5.2 – Fluxograma de procedimentos para implementação do montador

5.2.1. Escolha do método de programação

Duas são as possibilidades de escolha de métodos de programação: estruturada e orientada a objeto. Apesar dos comandos estarem encapsulados dentro de funções e a utilização de propriedades de componentes do Borland C++ Builder, a qual dá-se por orientação a objeto, a ênfase na forma estruturada foi escolhida por resultar numa programação mais fácil, em primeira análise, sendo utilizada para a construção do software montador pois atende às especificações desejadas, mas gera um código mais denso e difícil de entender que a programação orientada a objeto.

5.2.2. Definição da forma de inserção de texto pelo usuário

Utilizando-se a ferramenta de desenvolvimento Borland C++ Builder 5.0, para que o processo de construção dos componentes visuais como as janelas fosse acelerado, tem-se como opções para entradas de instruções a serem tratadas as seguintes formas:

Opção para entrada de instruções	**Vantagens**	**Desvantagens**
Caixa de diálogo (“Edit Box”)	Tamanho e facilidade de implementação	O usuário não possui acesso às informações digitadas com facilidade
Editor de texto convencional	Visualização e facilidade de edição do texto digitado	Implementação das rotinas de tratamento mais complicada em C++

Tabela 5.2.2.a – Opções para entrada de instruções

Na primeira opção, o programa funcionaria como uma “caixa registradora”, possuindo um campo onde seria digitada a instrução e botões informando que será inserida uma nova instrução ou de finalizar o programa, porém, as informações digitadas não são facilmente visíveis ao usuário. A segunda opção leva a um editor de texto convencional acrescido das funcionalidades necessárias à utilização do software como um montador.

Analisando-se as opções, preferiu-se a escolha por um editor de texto, pois usuários de computadores têm maior familiaridade com este tipo de interface. Porém, no Borland C++ Builder 5.0 há a também possibilidade de escolha de utilização de componentes para servirem como local de entrada do texto: o **Memo** e o **Rich Edit**, os quais fundamentalmente gozam de praticamente as mesmas propriedades, exceto principalmente que o **Rich Edit** gera automaticamente um documento do tipo *.rtf (*Rich Text Format*) de larga utilização no Microsoft Windows e editores de texto compatíveis, sendo este o fator decisivo da predileção por este componente. Além disto, deve ser construído o maior número possível de funcionalidades associadas a editores de texto no programa montador, pois estas funções facilitam em muito o desenvolvimento de programas para aplicações no processador.

5.2.3. Escolha da forma de comparação entre os valores de referência e digitado

Para a comparação entre os valores de referência e o digitado pelo usuário, viu-se como opções as seguintes, apresentadas com suas vantagens e desvantagens:

Método	Vantagem	Desvantagem
Tabela ASCII (Utilização da codificação existente na tabela ASCII)	Possui a codificação dos caracteres pronta em hexadecimal	• É necessária a codificação numérica da soma, por exemplo, dos valores ASCII dos caracteres • Programação enfadonha pois todos os valores ASCII devem ser conhecidos; • Problemas com a detecção de erros de sintaxe.
Comparação de strings de caracteres	Comparação simples por funções presentes no C++ e mais fácil detecção de erros de sintaxe.	• Utilização de ponteiros – maior probabilidade de erro na programação

Tabela 5.2.3 – Formas de comparação entre os valores de referência e digitado

Analisando-se as duas opções, preferiu-se a utilização de comparação de strings de caracteres, apesar dos problemas normalmente encontrados quando da utilização de ponteiros. Esta forma de comparação utiliza vetores para armazenar as strings de caracteres da linha do texto digitada pelo usuário. O mecanismo básico de comparação checa posição por posição a igualdade dos vetores, sem haver a necessidade para a determinação da instrução a checagem de todo o vetor, mas somente dos primeiros caracteres. Os outros caracteres serão comparados com os valores armazenados de registradores e métodos de conversão de valores numéricos para constantes, o que será conseqüência da caracterização da instrução em processo de análise. O método opera da seguinte forma:

Seja o vetor formado a partir do texto ***cp***

A	D	D		$	...	\0
0	1	2	3	4		12

String 1

e os valores de referência para as instruções....

A	D	D	
0	1	2	3

Micro-instrução ADD

A	D	D	I
0	1	2	3

Micro-instrução ADDI

Comparam-se os valores de ***cp***, posição por posição, com os valores de referência. No caso, deve-se comparar até a posição 3 pois ela define a diferença entre as instruções. Portanto, o número de posições a serem comparadas depende das possibilidades existentes entre instruções que possuem caracteres

em comum. Por simplicidade, os valores de referência são apenas caracteres e não vetores, pois desta forma, basta comparar a posição com o caracter existente no mesmo lugar para a instrução. Eventuais problemas com tabulação devem ser tratados pois neste caso há uma alteração das posições de caracteres dentro do vetor.

Após a identificação da instrução (esta forma de comparação foi utilizada para micros e pseudo-instruções) passa-se ao tratamento dos outros campos da instrução, também por comparação entre posição de vetor e caracter, sendo necessária a implementação de rotinas de identificação e tratamento de erros de sintaxe. Os caracteres são varridos um a um até que se finalize a instrução, o que no vetor é indicado pelo valor nulo, **\0**, na sintaxe do C++. Utilizam-se como caracteres de referência na determinação de campos de instrução o “espaço”, vírgulas, parênteses e o caracter ‘$’.

5.2.4. Definição das partes que compõem o programa montador e estabelecimento da ordem de implementação

O programa montador pode ser visto como composto pelas seguintes partes:

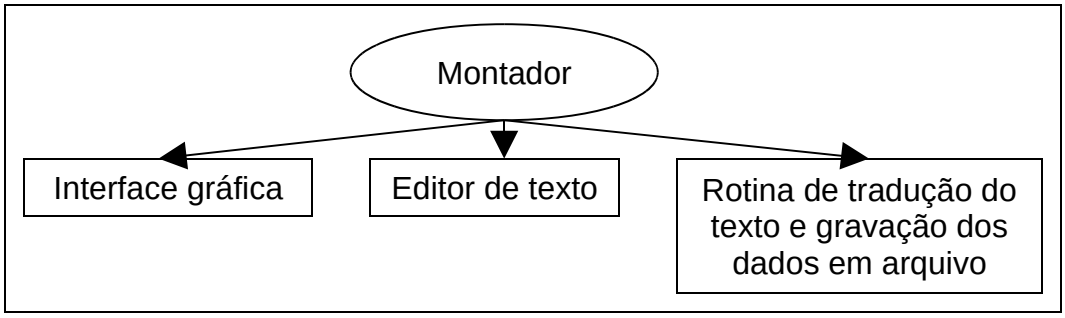


Figura 5.2.4.a – Partes que compõem o montador especificado

Como são partes, de certo modo, independentes pode-se escolher qualquer ordem para a construção do programa montador. Contudo, primando-se pela visualização dos resultados, a seqüência escolhida e recomendada para a construção é a seguinte:

- Implementação da interface gráfica;
- Construção das rotinas pertencentes ao editor de texto;
- Geração das rotinas de tradução do texto digitado para código de máquina e gravação dos dados resultantes em arquivo.

Esta é uma visão geral do problema, sendo melhor tratado quando for mostrada a implementação do programa, no item 6 do presente texto. Em todas estas etapas deve-se tomar o cuidado de tratar os possíveis erros a serem cometidos pelo usuário e não se dever perder do foco a especificação do montador, evitando-se colocar “acessórios” desnecessários ao funcionamento do programa. O passo seguinte é a construção do código do programa, sendo esta

parte analisada nas próximas seções da monografia. Em resumo, o montador será um software que realiza fundamentalmente a seguinte tarefa:

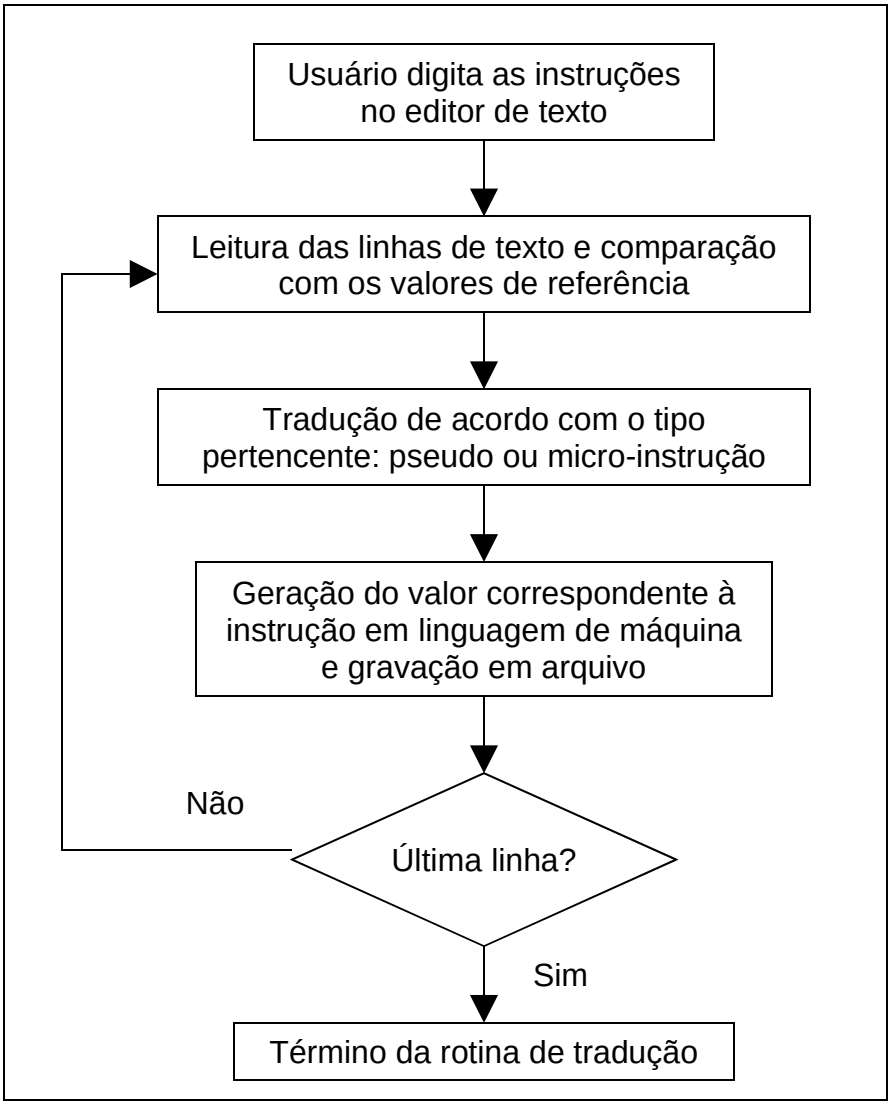


Figura 5.2.4.b – Fluxograma resumo da idéia presente no programa montador

Uma grande dificuldade obtida durante a implementação foi a falta de bibliografia sobre o assunto, sendo por isso utilizada largamente a referência [07], onde a partir do estudo das funções da linguagem C++ e suas propriedades, foi construído o “corpo” do programa.

6. O programa Montador v.1.1

O Montador-v.1.1 possui o seguinte fluxograma de operação:

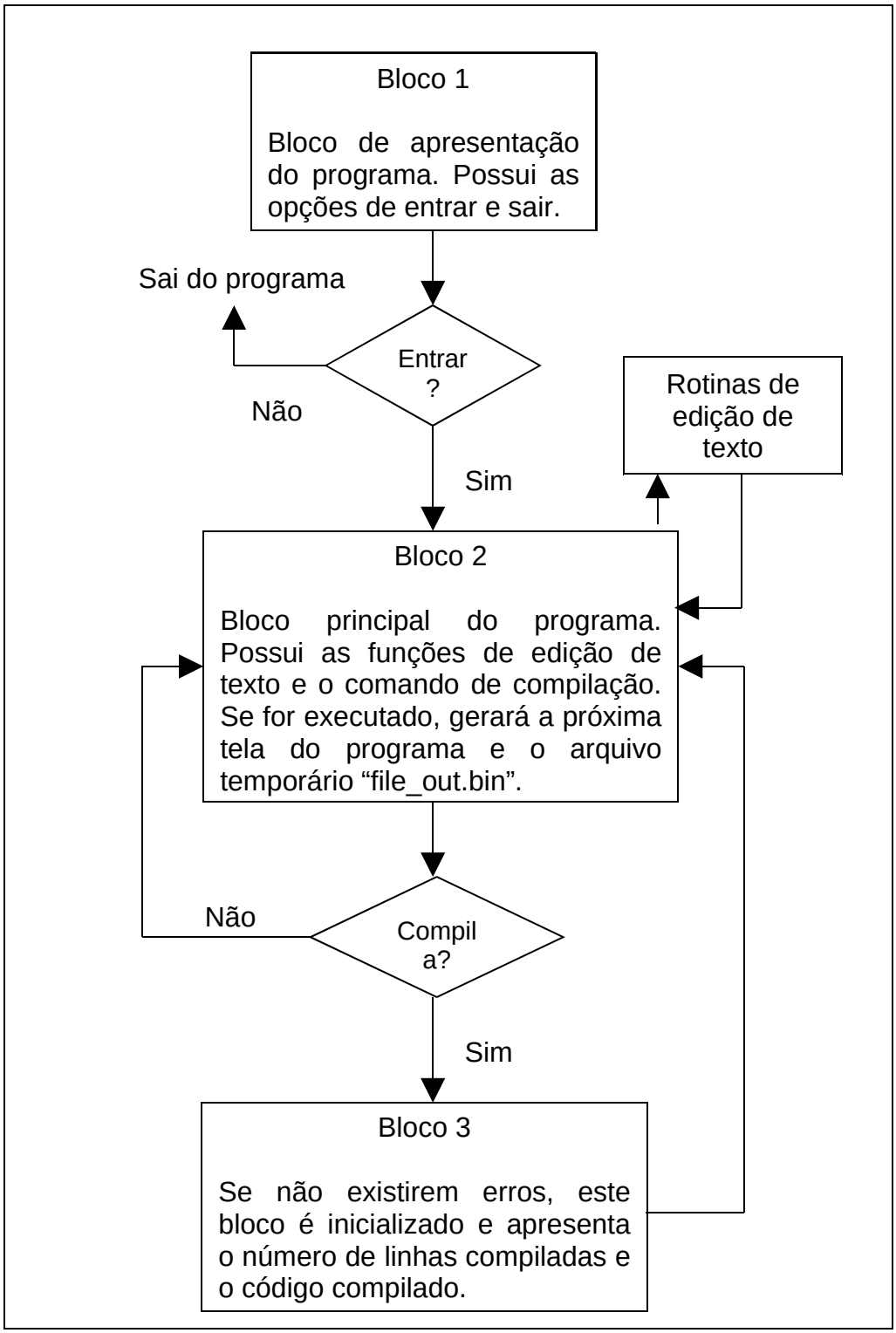


Figura 6.1 – Fluxograma do Montador – v1.1

O Montador - v.1.1 foi desenvolvido no software Borland C++ Builder 5.0. A fim de se seguir a ordem colocada na metodologia, a qual foi a mesma observada na elaboração e construção do programa, foi escolhida a numeração dos tópicos seguintes deste capítulo, compreendendo:

- Interface gráfica do programa;
- O editor de texto;
- Rotinas de “tradução” do texto;
- Ajuda e informações ao usuário.

Para uma diminuição das linhas de código, as funções recorrentes foram declaradas como públicas no arquivo fonte (Unit2.h), fazendo com que possam ser utilizadas por quaisquer rotinas programadas nos arquivos Unit1.cpp e Unit2.cpp, e encapsuladas neste último arquivo (o que remete à programação orientada a objeto). O mesmo procedimento foi adotado para variáveis de interesse utilizadas em várias partes do programa.

6.1. Interface gráfica do programa

O programa foi construído a partir das seguintes unidades básicas fornecidas pelo Borland C++ Builder 5.0:

- Form: correspondem às janelas utilizadas;
- BitBtn: botões com símbolos identificadores das funções Entrar e Sair exibidos na primeira janela do programa;
- SpeedButton: botões utilizados na barra de tarefas com as funções do Montador – v1.1;
- Panel: utilizado como agregador dos botões da barra de tarefas;
- Menu: possuindo também sub-menus com todas as funções do montador;
- Caixas de diálogo: para as operações de salvar, abrir e imprimir arquivos, formatar fonte, configurar impressão e localizar palavras no texto;
- ImageList: colocação da figura na janela inicial do programa;
- About Box: caixa de mensagens para informações sobre o programa;
- Bevel: como moldura de figuras;
- Label: para a inserção de textos na primeira janela (título do programa) e no About Box;
- RichEdit: componente onde o texto digitado pelo usuário é escrito.

Tem-se as seguintes janelas no programa:

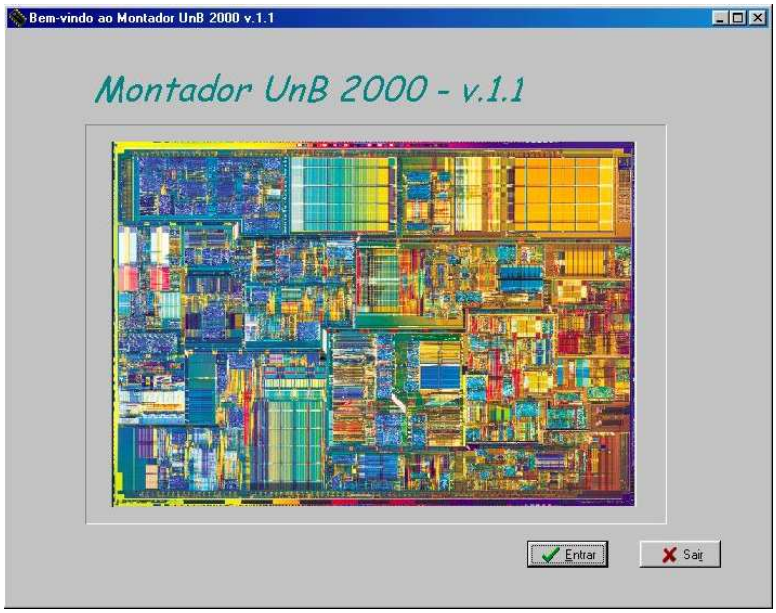


Figura 6.1.1.a – Janela inicial do programa

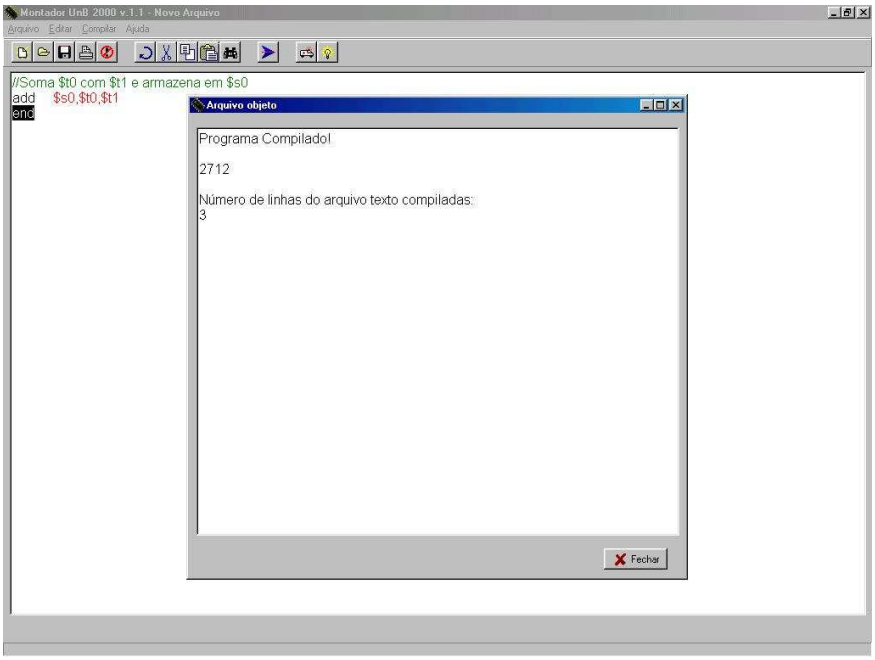


Figura 6.1.1.b – Janelas do editor de texto e código convertido

A janela 2, no fundo da Figura 6.1.1.b é o local onde o texto é digitado pelo usuário, onde estão disponíveis todas as funções de edição, sendo os botões dispostos para corresponderem à ordem dos menus. Já na janela 3 é apresentado

o código do programa convertido, apenas com fins de visualização, sendo vedada ao usuário a possibilidade de alteração do código a partir desta tela.

6.2. O editor de texto

As rotinas básicas do editor de texto foram selecionadas a fim de se ter uma maior comodidade do usuário quando da utilização do software, são elas:

- Criar novo arquivo;
- Salvar e salvar como;
- Abrir arquivo;
- Sair do programa;
- Imprimir;
- Configurar impressão;
- Copiar;
- Recortar;
- Colar;
- Selecionar todo o texto;
- Desfazer;
- Selecionar fonte.

As funções serão apresentadas aqui de acordo com o ícone correspondente, tecla de atalho e caixa de diálogo, quando existirem, bem como o código fonte correspondente a cada funcionalidade.

6.2.1. Funções de arquivo e programa

6.2.1.1. Novo

Figura 6.2.1.1 – Botão Novo Arquivo

Sua tecla de atalho é Ctrl+N. Está presente também no menu “Arquivo”. É utilizado para a criação de um novo arquivo texto, mas caso já se tenha digitado algo na tela, aparecerá ao usuário um caixa de mensagem perguntado-lhe se o mesmo deseja salvar o que já foi digitado, sendo esta parte criada neste projeto. Seu código fonte, extraído da referência [05], é o seguinte:

```
// Função que cria um novo documento
bool  TForm2::NewFile()
{
  if(RichEdit1->Modified)
  {
    switch (Application->MessageBox("Quer gravar as modificações?",
```

```
            "Montador - v.1.1", MB_YESNOCANCEL))
```

```
    {
      case IDYES:
        if (SaveFile())
          return true;
          break;
      case IDCANCEL:
        return true;
    }
  }

  //Limpando a janela de edição para o update dos nomes dos arquivos
    RichEdit1->Text = "";
    RichEdit1->Modified = false;
    Filename = "";
    TForm2::Caption = ("Montador - v.1.1 - Novo Arquivo");
    return false;
  }
}
```

C.F.6.2.1.1 – Código fonte da função que cria um Novo Arquivo

A função, criada como do tipo ***bool*** indica que haverá um valor de retorno do tipo verdadeiro ou falso após o término da operação. Nesta função, o valor de retorno não foi utilizado para outros fins, mas somente para finalizar a função. Ainda na primeira linha tem-se ***TForm2::NewFile()***, o que indica que esta função pertence ao objeto ***TForm2*** e não possui argumentos de entrada (os parênteses estão vazios).

O primeiro laço de teste ***if*** verifica se houve modificação na área de texto destinada ao usuário. Caso tenha existido, sugere-se a gravação daquilo que foi digitado antes de se gerar um novo arquivo. Para tanto, gera-se uma caixa de mensagem baseada num ***switch – case*** no qual são selecionadas opções de tratamento de acordo com o botão pressionado pelo usuário respondendo o questionamento: ***"Quer gravar as modificações?"*** . Se o botão selecionado foi o sim (***case IDYES***) o programa executa a função ***SaveFile()***, retorna o valor ***true*** (verdadeiro) e finaliza o ***switch – case***. Caso seja selecionada a opção não (***case IDNO***), apenas finaliza-se o ***switch – case***. O próximo passo é limpar a tela através dos comandos:

```
RichEdit1->Text = “”;              //Apaga todo o texto digitado na tela
RichEdit1->Modified = false;       //Indica que não houve digitação
```

e executar as atualizações do nome do arquivo e texto aparecendo na moldura da janela:

```
Filename = "";          //Apaga o nome do arquivo
TForm2::Caption = ("Montador - v.1.1 - Novo Arquivo");
```

6.2.1.2. Abrir arquivo

Figura 6.2.1.2.a – Botão Abrir Arquivo

Sua tecla de atalho é Ctrl+A. Está presente também no menu “Arquivo”. É utilizado para a abertura de um arquivo texto preexistente nos formatos *., mas caso já se tenha digitado algo na tela, aparecerá ao usuário um caixa de mensagem perguntado-lhe se o mesmo deseja salvar o que já foi digitado, da mesma forma que o anterior, esta parte foi criada neste projeto. Possui uma caixa de diálogo que opera em conjunto com o algoritmo facilitando a implementação da funcionalidade. Seu código fonte, extraído da referência [05], é o seguinte:

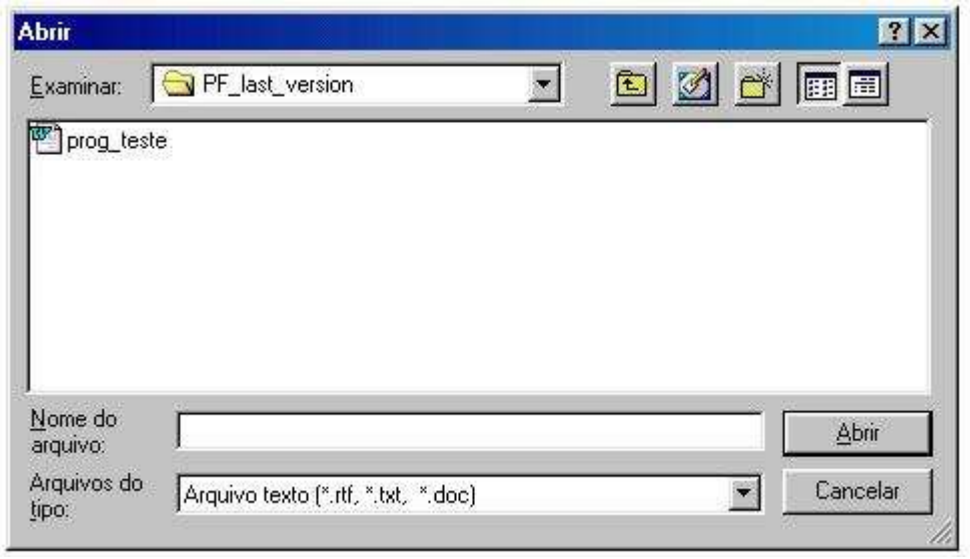


Figura 6.2.1.2.b – Caixa de diálogo OpenDialog

```
bool  TForm2::Openf()
{
  if(RichEdit1->Modified)
  {
            switch (Application->MessageBox("Quer gravar as modificações?",
"Montador - v.1.1", MB_YESNOCANCEL))
    {
      case IDYES:
        if (SaveFile())
          return true;
        break;
      case IDCANCEL:
        return true;
```

```
    }
  }
```

```
   if(OpenDialog1->Execute())
     {
     TStringList *TempList = new TStringList;
     TempList->LoadFromFile(OpenDialog1->FileName);
     RichEdit1->Lines->LoadFromFile(OpenDialog1->FileName);
     Filename=OpenDialog1->FileName;
     TForm2::Caption = ("Montador - v.1.1 - " + Filename);
  }
  return   false;
}
```

C.F.6.2.1.2 – Código fonte da função de Abrir Arquivo

Este algoritmo utiliza o componente ***OpenDialog*** do C++ Builder, com filtros configurados para permitir a abertura de arquivos do tipo texto (*.doc, *.txt e *.rtf), bem como para a abertura de qualquer arquivo. Da mesmo modo que foi realizado para a função Novo Arquivo, o código fonte da função abrir pergunta ao usuário se ele deseja salvar o arquivo atual caso tenha sido digitado algo. Após isso, executa a componente ***OpenDialog*** criando uma nova instância ***TstringList*** chamada ***TempList***, onde será colocado o texto carregado do arquivo ***FileName***, selecionado na caixa de diálogo ***OpenDialog1***. Após isto, as linhas do componente ***RichEdit1*** são preenchidas pelo conteúdo do arquivo através do comando:

RichEdit1->Lines->LoadFromFile(OpenDialog1->FileName);

Finalizando a operação, o nome apresentado na moldura da janela é alterado para incluir o nome do arquivo que foi aberto. Nesta versão do programa Montador - v1.1 não foi permitido ao usuário carregar um arquivo contendo o código objeto (código obtido após a conversão do texto).

6.2.1.3. Gravar arquivo

Figura 6.2.1.3 – Botão Salvar

Sua tecla de atalho pode ser tanto Ctrl+S quanto Ctrl+D, sendo que a última abre a caixa de diálogo Salvar Como. Também pertence ao menu “Arquivo”. Os códigos fontes para as funções ***SaveFile()*** e ***SaveFileAs()*** foram extraídas da referência [05]. Já a função responsável pela gravação do arquivo objeto foi uma alteração feita neste projeto do algoritmo de salvar arquivo, tanto na função ***SaveFile()*** como na função ***SaveFile_obj()***. Todas as funções deste item são do tipo bool, ou seja, retornam um valor verdadeiro ou falso, dependendo das condições respondidas durante a rotina das funções.

As funções de gravação de arquivo utilizam o componente do C++ Builder ***SaveDialog***. Para as funções relacionadas a arquivo texto, utilizou-se o componente ***SaveDialog1***. Para gravar o arquivo objeto, ***SaveDialog2***.

Os parâmetros de filtro utilizados são os seguintes:

- Arquivo texto: *.rtf, *.txt, *.doc;
- Arquivo em código de máquina: *.bin.

As funções de gravação possuem uma caixa de diálogo idêntica a da função Abrir Arquivo. Os códigos fontes utilizados para as funções ***SaveFile()*** e ***SaveFileAs()*** estão apresentados no item Anexo do relatório, capítulo 5. Para a função que salva o arquivo objeto, tem-se o seguinte código:

```
// Função para salvar o arquivo objeto.
bool  TForm2::SaveFile_obj()
{
     FILE *in;
     if ((in = fopen("file_out.bin", "r"))
     == NULL)
     {
        MessageBox(Handle, "Não foi possível abrir o arquivo temporário",
        "Mensagem de erro #01", MB_OK);
        return false;
     }
     fclose(in);

     SaveDialog2->Execute();
     if (FileExists(SaveDialog2->FileName))
     {
        DeleteFile(SaveDialog2->FileName);
     }
     if (SaveDialog2->FileName == "file_out")
     {
        return true;
     }
     RenameFile("file_out.bin", SaveDialog2->FileName);
     return false;
}
```

C.F.6.2.1.3.c – Código fonte da função Salvar Arquivo Objeto

O código inicia-se com o conceito de stream (fluxo de dados de entrada e/ou saída), onde é criado o stream ***FILE *in***. Testa-se, então, se "***in***" pode receber o arquivo gerado pela conversão do texto, contendo o código objeto "***file_out.bin***", configurando o recebimento como somente para leitura pela inclusão de "***r***" no código da instrução. O teste será verdadeiro caso não se

consiga carregar o arquivo (teste==***NULL***), sendo exibida uma mensagem de erro na tela. Caso consiga-se transferir o arquivo, é então selecionado um nome através da caixa de diálogo SaveDialog2. O nome do arquivo "***file_out.bin***" é então substituído pelo nome criado pelo usuário, finalizando-se o código com o comando ***return false***.

6.2.1.4. Imprimir

Figura 6.2.1.4.a – Botão Imprimir

Sua tecla de atalho é Ctrl+P. Para configuração da impressão, a tecla de atalho é Ctrl+W. As configurações e possibilidades existentes são as fornecidas pelo MS Windows. Ambas funções estão presentes no menu arquivo. Os códigos fontes foram extraídos da referência [05]. Associado ao botão estão a seguinte caixa de diálogo e o código fonte:

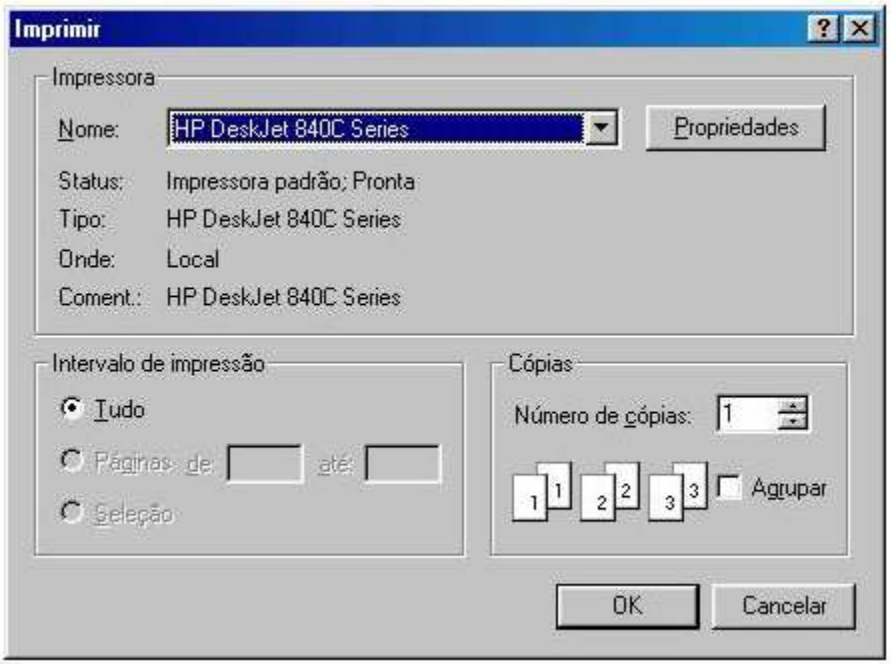


Figura 6.2.1.4.b – Caixa de diálogo PrintDialog

```
void __fastcall Tform2::SpeedButton4Click(Tobject *Sender)
{
   if (PrintDialog1->Execute())
      RichEdit1->Print ("");
}
```

C.F.6.2.1.4.a. – Código fonte do botão Imprimir

O código fonte do botão apenas executa a caixa de diálogo e imprime o texto contido dentro do componente ***RichEdit1***.

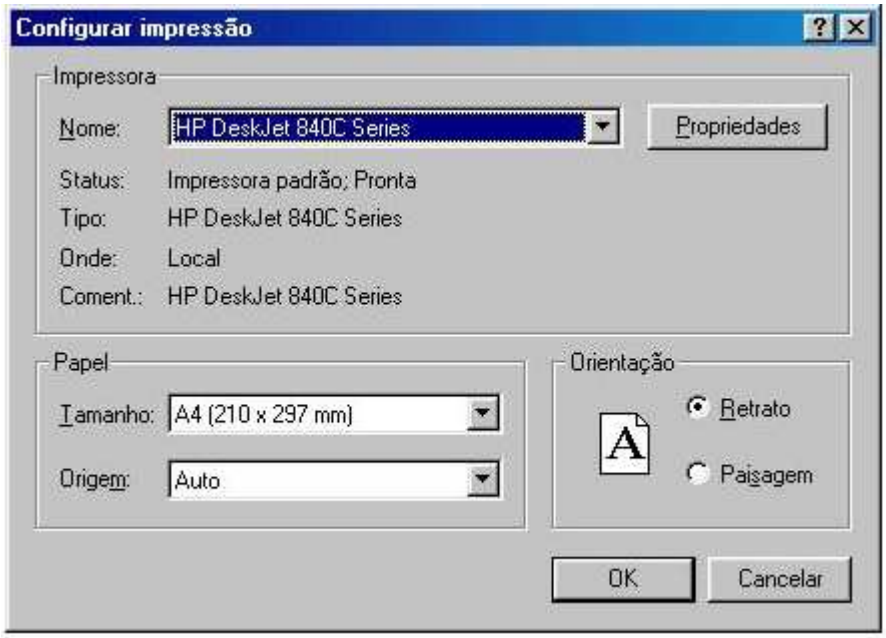


Figura 6.2.1.4.c – Caixa de diálogo PrintSetupDialog

```
void __fastcall TForm2::ConfigurarImpresso1Click(TObject *Sender)
{
  PrinterSetupDialog1->Execute();
}
```

C.F.6.2.1.4.b. – Código fonte do item de menu configurar Impressão

```
void __fastcall TForm2::Imprimir1Click(TObject *Sender)
{
  int cy;
  if (!PrintDialog1->Execute())
    return;
  Printer()->BeginDoc();
  Printer()->Title = Caption;

  Printer()->Canvas->Font = RichEdit1->Font;

  cy = Printer()->Canvas->TextHeight("X");

  for(int Linha =0; Linha < RichEdit1->Lines->Count; Linha++)
          Printer()->Canvas->TextOutA(10, cy* Linha + 5, RichEdit1->Lines-
>Strings[Linha].c_str());
  Printer()->EndDoc();
}
```

C.F.6.2.1.4.c. – Código fonte do item de menu Imprimir

O código fonte da função de configurar impressão apenas executa a respectiva caixa de diálogo. Por outro lado, na função ***Imprimir1Click*** é criada a rotina de impressão do texto, utilizando-se também a caixa de diálogo ***PrintDialog1***, imprimindo-se do início ao fim do documento através do laço ***for***, onde o parâmetro utilizado para finalizar a rotina é o número de linhas do texto.

6.2.1.5. Sair

Figura 6.2.1.5 – Botão Sair

As funções de sair do programa podem ser acessadas na tela de apresentação e no editor de texto, através do botão sair, do item “Sair” do menu “Arquivo” ou tecla de atalho Ctrl+Q (permite cancelar o pedido de sair), pela tecla de atalho Atl+F4 (sai do programa sem permitir cancelamento) ou pelo canto superior direito da janela. Foram criadas neste projeto a partir da função ***exit*** da linguagem C++. Os código fonte utilizados estão em Anexo ao relatório, no capítulo 5, item 5.2.

6.2.2. Funções de edição

O menu de edição foi adaptado para permitir algumas operações somente quando há texto digitado pelo usuário, a fim de se proteger o programa de interferências de programas externos. Para isso, utilizou-se o seguinte algoritmo, extraído da referência [05]:

```
void __fastcall TForm2::Edit1Click(TObject *Sender)
{
  //Somente habilita as funções se houve digitação de texto
  Voltar1->Enabled = RichEdit1->Modified;
  Recortar1->Enabled = RichEdit1->SelLength>0;
  Copiar1->Enabled = Recortar1->Enabled;
  Colar1->Enabled = Clipboard()->HasFormat(CF_TEXT);

  if (SendMessage (RichEdit1->Handle, EM_CANUNDO, 0, 0))
    Voltar1->Enabled = true;
  else
    Voltar1->Enabled = false;
}
```

C.F.6.2.2 – Código fonte de habilitação do menu “Edit”

Os itens de menu somente são passíveis de serem utilizados caso haja modificação no texto inserido no componente ***RichEdit1***. Esse controle é feito através da propriedade ***Enabled*** dos itens de menu criados. O mesmo

procedimento foi realizado para os botões, sendo apresentado no Anexo do relatório, capítulo 5, item 5.2.

6.2.2.1. Voltar

Figura 6.2.2.1 – Botão Voltar

A função voltar desfaz a digitação que ficou armazenada no algoritmo deste comando. É utilizada para correções rápidas a medida que o texto é digitado, contudo, sua má utilização por parte do usuário pode ser desastrosa. Possui como tecla de atalho Ctrl+Z e foi construída a partir da referência [05], e seu algoritmo é apresentado no Anexo, capítulo 5, item 5.2.

6.2.2.2. Recortar

Figura 6.2.2.2 – Botão Recortar

Assim como nos editores de texto convencionais, este comando simplesmente move o conteúdo selecionado no texto para a área de transferência. Sua tecla de atalho é Ctrl+X e foi construída de acordo com a referência [06].

```
void __fastcall TForm2::Recortar2Click(TObject *Sender)
{
  RichEdit1->CutToClipboard();
}
```

C.F.6.2.2.2 – Código fonte da função Recortar

6.2.2.3. Copiar

Figura 6.2.2.3 – Botão Copiar

Assim como o comando recortar, esta é uma funcionalidade comum nos editores de texto convencionais. Copia o texto selecionado para a área de transferência, podendo o mesmo ser utilizado posteriormente até que o usuário coloque outros dados na área de transferência ou finalize o aplicativo. Sua tecla de atalho é Ctrl+C, sendo que o algoritmo foi extraído da referência [06].

```
void __fastcall TForm2::Copiar1Click(TObject *Sender)
{
  RichEdit1->CopyToClipboard();
}
```

C.F.6.2.2.3 – Código fonte da função Copiar

6.2.2.4. Colar

Figura 6.2.2.4 – Botão Colar

Presente em todos os editores de texto, este comando insere no texto o conteúdo da área de transferência sem esvaziá-la. Possui como tecla de atalho Ctrl+V. O algoritmo foi obtido da referência [06].

```
void __fastcall TForm2::Colar1Click(TObject *Sender)
{
   RichEdit1->PasteFromClipboard();
}
```

C.F.6.2.2.4 – Código fonte da função Colar

6.2.2.5. Selecionar Tudo

Este comando não está presente na barra de tarefas do processador, contudo é de grande utilidade quando associado ao comando colar, caso queira-se inserir o conteúdo de determinado arquivo em outro. Sua tecla de atalho é Ctrl+T. O código fonte foi retirado da referência [06].

```
void __fastcall TForm2::SelecionarTudo1Click(TObject *Sender)
{
  RichEdit1->SelStart = 0 ;
  RichEdit1->SelLength = -1 ;
  return ;
}
```

C.F.6.2.2.5 – Código fonte da função Selecionar Tudo

Pelo primeiro comando, a seleção inicia-se na posição ***0*** através da propriedade ***SelStart***, sendo o tamanho arbitrado para ser o tamanho de todo o texto através da utilização da constante **–1** na propriedade ***SelLength*** do componente ***RichEdit1***. Esta função não retorna vazio após ser executada, pois é do tipo ***void***, e por este mesmo motivo é finalizada apenas com o comando ***return***.

6.2.2.6. Localizar

Figura 6.2.2.6.a – Botão Localizar

Sua tecla de atalho é Ctrl+L. Esta é mais uma função característica dos editores de texto, mas que á utilizada em inúmeros softwares. Permite ao usuário

encontrar rapidamente uma palavra em toda a extensão do texto, podendo a busca ser repetida a fim de encontrar-se outras ocorrências. Assim como as funções Salvar e Abrir, utiliza uma caixa de diálogo, chamada ***FindDialog***, mostrada na Figura 6.2.2.6.b. A chamada à função e o código utilizado para efetuar a busca foram extraídos da referência [05].

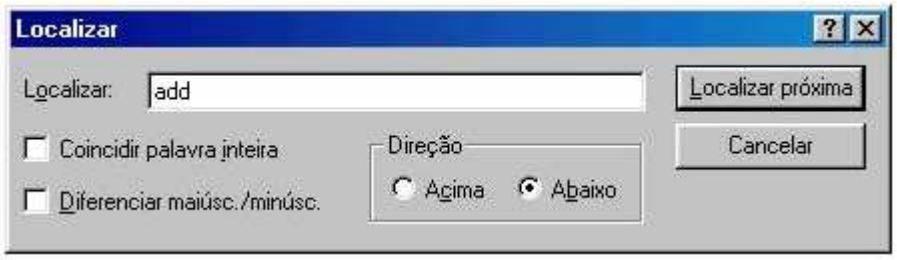


Figura 6.2.2.b – Caixa de diálogo FindDialog

```
void __fastcall TForm2::Localizar1Click(TObject *Sender)
{
  FindDialog1->Execute();
}
```

C.F.6.2.2.a – Código fonte da chamada à função Localizar

```
void __fastcall TForm2::FindDialog1Find(TObject *Sender)
{
  TFindDialog *dialog = (TFindDialog *)(Sender);
  int position; //posição onde a string foi encontrada

  // Copia os parâmetros de procura do dialog box
  TSearchTypes options;
  if (dialog->Options.Contains (frMatchCase))
    options << stMatchCase;
  if (dialog->Options.Contains (frWholeWord))
    options << stWholeWord;

  // Vendo se estamos procurando para cima ou para baixo
  if (dialog->Options.Contains (frDown))
  {
    int start = RichEdit1->SelStart;
        if (RichEdit1->SelLength !=0) //Isto captura a posição do cursor a ser
comparada
      ++ start;
          position = RichEdit1->FindText (dialog->FindText, start, RichEdit1-
>Text.Length()-RichEdit1->SelStart-1,options);
  }
```

```
    else if (RichEdit1->SelStart >0)
    {
     // Para procurar para baixo ficaremos em loop até encontrarmos a palavra anterior
    int search = -1;
    do
    {
      position= search;
         search = RichEdit1->FindText (dialog->FindText, search +1, RichEdit1->SelStart - search -1, options);
    }while (search>=0);
  }
  else
  {
    //Estamos procurando acima do início do buffer
    position = -1;
  }
  // Se encontrado, então selecione o texto
  if (position>=0)
  {
    RichEdit1->SelStart = position;
    RichEdit1->SelLength = dialog->FindText.Length();
  }
  else
  {
       MessageBox (Handle, "Palavra não localizada." , "Montador - v.1.1", MB_ICONINFORMATION);
    return;
  }
}
```

C.F.6.2.2.6.b – Código fonte para busca da palavra

A rotina para localizar texto inicia-se copiando os parâmetros obtidos a partir da caixa de diálogo ***FindDialog1***, verificando se a palavra deve coincidir em maiúsculas e minúsculas *(**frMatchCase**)* ou se a palavra deve ser toda escrita igual *(**frWholeWord**).* Depois, checa-se se a procura deve ser do ponto onde o cursor de inserção de texto está, para cima, ou para baixo. Mais uma vez são utilizados como parâmetros as propriedades ***SelStart*** e ***SelLength*** do componente ***RichEdit1***, indicando o início da procura e o comprimento da mesma. A procura é realizada em toda as linhas contidas no intervalo entre ***SelStart*** e ***SelLength***. A variável inteira ***position*** armazena o local onde a palavra procurada foi encontrada. Caso não seja encontrada nenhuma ocorrência do verbete

pesquisado, a função retorna com uma caixa de mensagem informando o fato e finalizando a operação.

6.2.2.7. Fonte

Este comando está presente no menu Editar. Sua tecla de atalho é Ctrl+F. Possui associado a ele uma caixa de diálogo, a ***FontDialog***, dando ao usuário liberdade para escolher a fonte e características da mesma como: cor, tamanho, negrito, sublinhado, etc. Para facilitar a visualização do que é digitado pelo usuário, foram definidas algumas cores padrão, tomando como referência as teclas “espaço”, “ENTER”, “backspace”, “tab” e as indicativas de um texto comentário (* e //), quando são pressionadas pelo usuário. A rotina de execução do FontDialog foi retirada da referência [06], mas a rotina sensível às teclas pressionadas pelo usuário foi desenvolvida neste projeto.

Figura 6.2.2.7 – Caixa de diálogo FontDialog

```
void __fastcall TForm2::Opes1Click(TObject *Sender)
{
  //Utiliza a fonte escolhida pelo usuário no texto a partir deste ponto.
  if (FontDialog1->Execute())
     {
     RichEdit1->SelAttributes->Assign(FontDialog1->Font);
     }
}
```

C.F.6.2.2.7.a – Código fonte da função Fonte

Esta rotina abre a caixa de diálogo ***FindDialog1***, recebendo os valores advindos dela e mudando os parâmetros do texto localmente através da propriedade ***SelAttributes***.

```
void __fastcall TForm2::RichEdit1KeyPress(TObject *Sender, char &Key)
{
  /*
      A rotina utiliza as codificações das teclas:
        Enter = \r
        Backsapce = \b
        Espaço = ´´
      além de variáveis auxiliares que funcionam como ponteiros para indicar
      o número de caracteres digitados, além da variável Key que armazena o
      último caracter digitado.

      As cores da fonte são mudadas pelo comando:

      RichEdit1->SelAttributes->Color=clBlack;

      onde estão descritos o RichEdit utilizado, que se deseja mudar atributos
      e o tipo de cor que será utilizada.
  */
  if(contachar==0)
  {
    RichEdit1->SelAttributes->Color=clBlack;
  }
  if(Key == '\r')
   {
    cont='0';
    contachar=-1;
    contacoment=0;
    return;
   }
  if(Key == '(')
  {
    RichEdit1->SelAttributes->Color=clBlue;
    cont='2';
  }
  if((cont=='0')&(contacoment == 0))
  {
    contacoment=6;
  }
  if((cont=='4')&(Key == '\b'))
  {
    RichEdit1->SelAttributes->Color=clBlue;
    cont='5';
```

```
  }
  if(cont=='4')
    RichEdit1->SelAttributes->Color=clRed;
```

```
  if(Key == ')')
  {
    RichEdit1->SelAttributes->Color=clBlue;
    cont='4';
  }

  if(Key == ' '|Key =='\t')
    switch (cont)
    {
      case '0':
        RichEdit1->SelAttributes->Color=clRed;
        contacoment++;
        break;
      case '1':
        RichEdit1->SelAttributes->Color=clGreen;
        contacoment++;
        break;
      case '2':
        RichEdit1->SelAttributes->Color=clBlue;
        break;
    }

  if((Key != ' ')&(Key !='\t')&(Key !='\b')&(cont=='1'))
    contacoment++;
  if((Key == '\/')|(Key == '*'))
  {
    if(cont!='1')
    {
      contb=cont;
      contacoment = 11;
    }
    RichEdit1->SelAttributes->Color=clGreen;
    cont='1';
  }
  if((Key=='\b')&(cont=='1'))
    contacoment--;
  if((Key=='\b')&(cont=='0'))
    contacoment--;
  if(contacoment==5)
  {
    RichEdit1->SelAttributes->Color=clBlack;
    cont='0';
```

```
    contacoment=0;
  }
```

```
  if(contacoment==10)
  {
    cont=contb;
    switch (cont)
    {
      case '0':
        RichEdit1->SelAttributes->Color=clRed;
        contacoment++;
        break;
      case '1':
        RichEdit1->SelAttributes->Color=clGreen;
        contacoment++;
        break;
      case '2':
        RichEdit1->SelAttributes->Color=clBlue;
        break;
    }
  }
  if((contachar>0)&(Key == '\b'))
    contachar=contachar-2;
  contachar++;
}
```

C.F.6.2.2.7.b – Código fonte da atribuição de cores a segmentos do texto

Para a utilização de cores de fonte diferentes para cada campo digitado pelo usuário utilizou-se como parâmetro as teclas de tabulação e o <ENTER>, tomando como base a codificação existente no C++, ou seja, as teclas foram identificadas da seguinte forma:

- <ENTER> = ‘ \r ’
- Espaço = ‘ ‘
- Tab = ‘ \t ’
- BackSpace = ‘\b’

Utilizou-se ainda as variáveis auxiliares ***contachar*** e ***contacoment***, que recebem o número de caracteres imprimíveis digitados no texto e os caracteres pertencentes a comentário, respectivamente. A identificação da digitação de comentários é realizada quando o usuário digita os caracteres **** ou *****, sendo que estes símbolos devem estar iniciando cada linha de comentário. Há ainda as variáveis que armazenam caracteres, ***Key***, ***cont*** e ***contb***. A primeira é originária do evento ***OnKeyPress*** do componente ***RichEdit1***, recebendo o caracter digitado pelo usuário a cada nova digitação. A segunda, guarda o valor correspondente à

cor que será utilizada. A variável ***contb*** armazena o número identificador de cor utilizado antes da última mudança de cor. Os valores iniciais das variáveis utilizadas na contagem de caracteres foram escolhidos para serem referência para as mudanças de cores quando são utilizados caracteres de tabulação, em especial a tecla backspace. A cada caracter digitado, os contadores são incrementados, exceto quando é pressionada a tecla backspace. Neste caso, quando a variável ***cont == '4'***, muda-se para um valor sem utilidade para se manter a cor em vigor. Para ***cont == '1'*** é decrementada de uma unidade a variável ***contacoment*** e para todas as possibilidades de valores de ***cont***, se há caracter digitado na linha, ***contachar*** é então decrementada de uma unidade.

Caso seja digitada a tecla <ENTER>, as variáveis de contagem são reiniciadas e a cor é trocada para a preta.

Foram realizados teste condicionais para algumas possibilidades de digitação pelo usuário. A cada vez que as teclas TAB e espaço são pressionadas existe a mudança da cor de preto para o vermelho, a não ser que se esteja digitando dentro de um comentário, onde a cor é colocada como verde. Caso utilize-se instruções do montador que possuem os parênteses em sua sintaxe, eles serão identificados e a cor será mudada para azul. As cores são mudadas através do comando:

RichEdit1->SelAttributes->Color=clBlue;

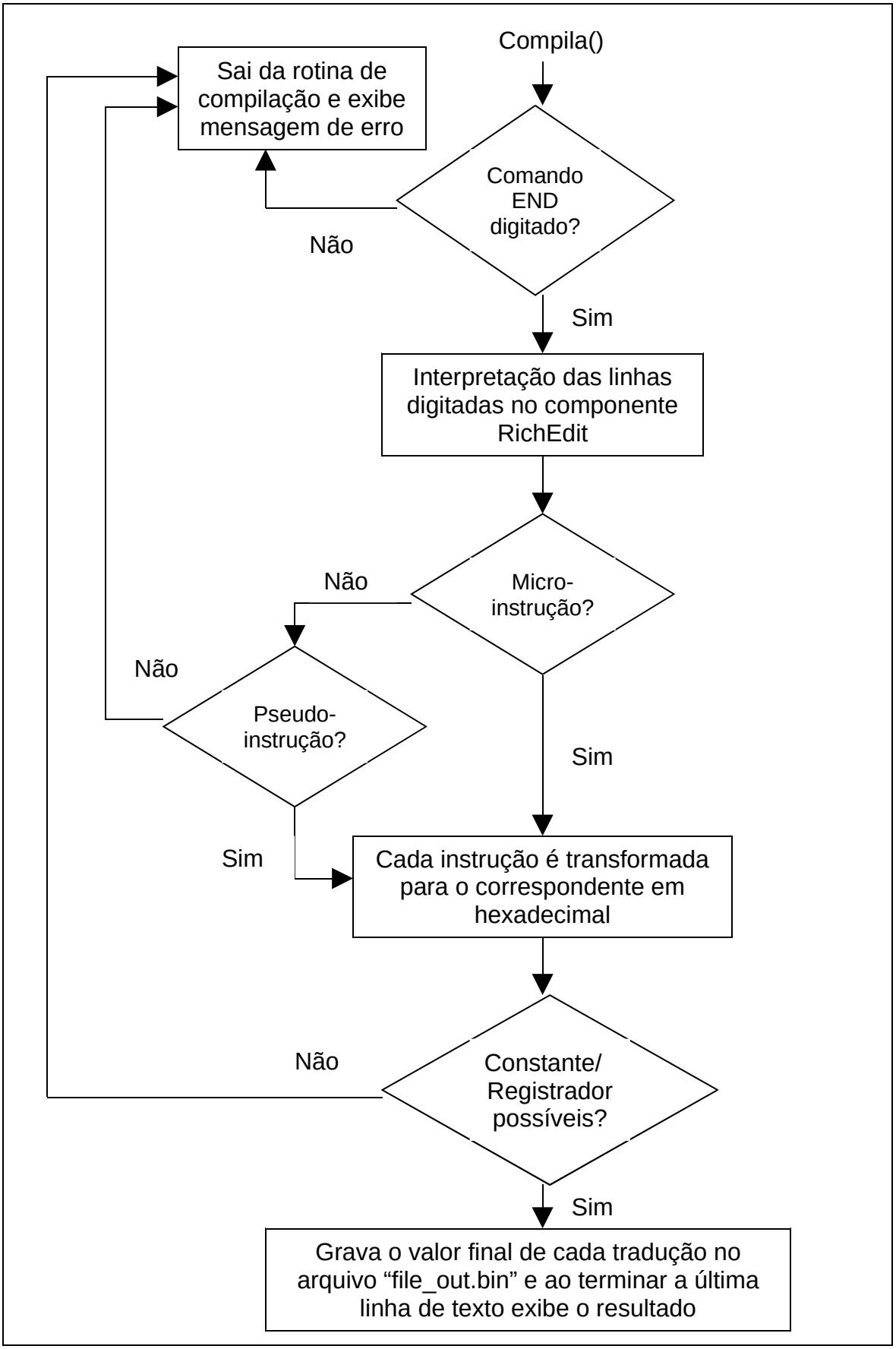
Onde mais uma vez utiliza-se a propriedade ***SelAttributes*** do componente ***RichEdit1***, mas agora altera-se a característica de cor.

6.3. Rotinas de conversão do texto

A rotina de conversão é a principal parte do programa Montador - v1.1, pois nela estão os algoritmos responsáveis pela codificação do texto apresentado pelo usuário para a linguagem de máquina. As rotinas criadas para esta função, podem ser separadas em quatro grandes grupos:

- Interpretação das linhas do texto;
- Comparação com valores armazenados e passagem para hexadecimal;
- Geração e gravação do arquivo contendo o código traduzido;
- Apresentação dos resultados.

Estes serão explorados com mais detalhes nas próximas subseções. Todas as linhas de programa desta função, Compila, foram desenvolvidas neste projeto final com o auxílio do Borland C++ Builder Help. O funcionamento da rotina de conversão pode ser melhor visualizado através da Figura 6.3.a.


Compila()
Sai da rotina de compilação e exibe mensagem de erro
Comando END digitado?
Não
Sim
Interpretação das linhas digitadas no componente RichEdit
Micro-instrução?
Não
Não
Pseudo-instrução?
Sim
Sim
Cada instrução é transformada para o correspondente em hexadecimal
Constante/ Registrador possíveis?
Não
Sim
Grava o valor final de cada tradução no arquivo “file_out.bin” e ao terminar a última linha de texto exibe o resultado

6.3.1. Interpretação das linhas de texto

O algoritmo de interpretação deste módulo baseia-se na utilização de variáveis como índices e de strings (conjunto de caracteres) de texto. Seu funcionamento pode ser melhor visualizado na Figura 6.3.1.a.

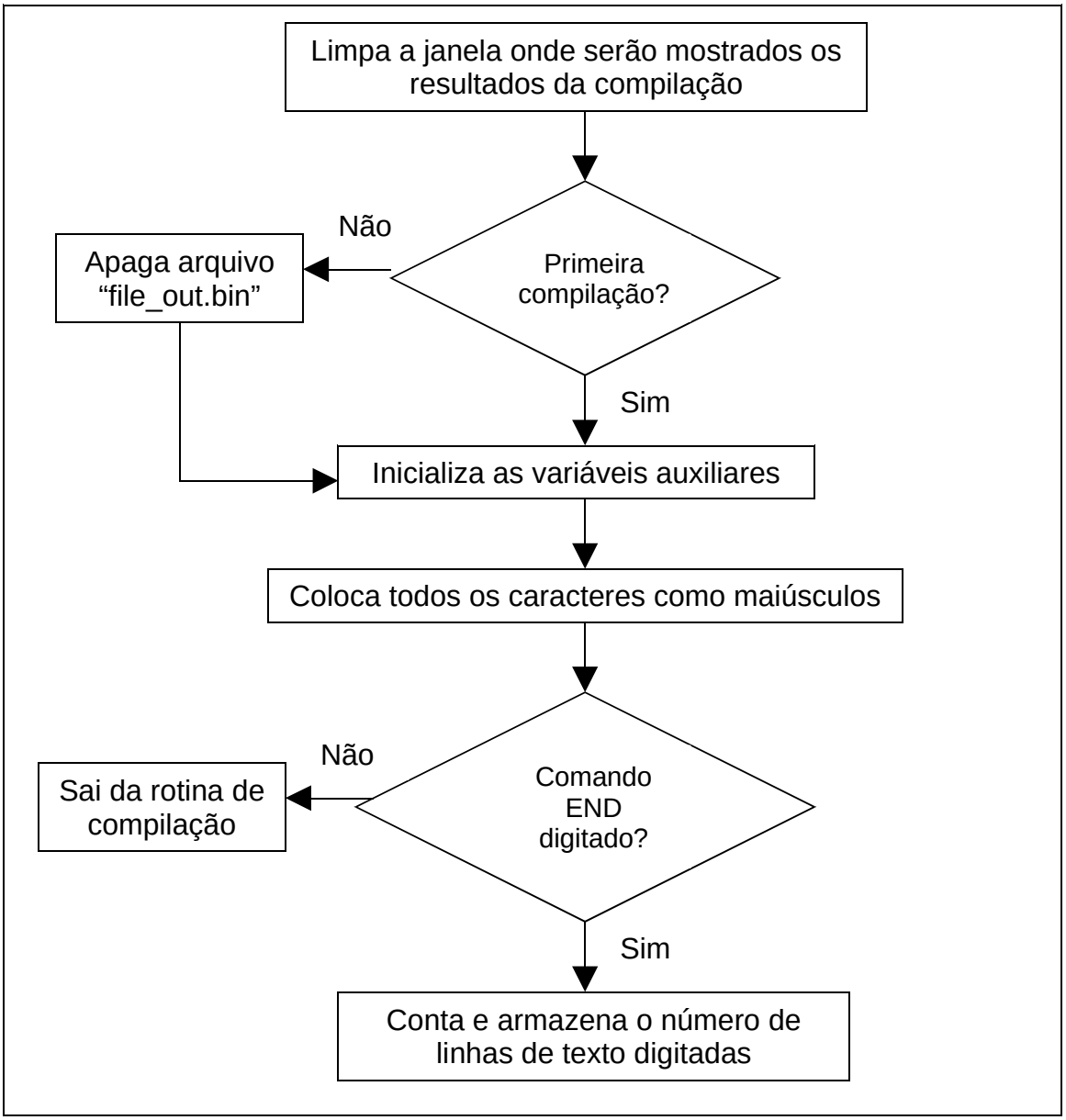


Figura 6.3.1.a – Interpretação das linhas de texto

Primeiramente o programa prepara a janela onde o resultado da conversão será apresentado e apaga o arquivo que contém o código traduzido caso esta não seja a primeira tradução do texto digitado. Para melhor explicação, o código fonte da função Compila() foi dividido em blocos mas a função é a soma de todos eles.

```
bool  TForm2::Compila()
{
  /*
  Função utilizada para transformar o texto digitado em código fonte para o
  processador. Esta função é o núcleo do Montador - v.1.1. Possui o
  algoritmo de leitura das linhas de texto e interpretação dos caracteres
  digitados, gravação do resultado no arquivo de saída "file_out.bin" e
  apresentação na tela dos valores obtidos para apenas visualização por par-
  te do usuário, não sendo possível alterar estes valores.
  */

  if (comp==1)
  {
    //Se já houve conversão anterior, apaga o arquivo do código traduzido
    DeleteFile("file_out.bin");
  }

  if (RichEdit1->Modified)
  {
    Form3->RichEdit1->Lines->Clear(); //Janela com os resultados é limpa.
    //Coloca as próximas strings na tela de programa convertido
    Form3->RichEdit1->Lines->Add("Programa Compilado!");
    Form3->RichEdit1->Lines->Add(" ");

    i=0;
    int j=0;          //Variável utilizada como índice
    int n=0;          //Variável que armazena o número de linhas do texto
    short volta=0;    //Variável que armazena o valor final em hexa da linha..
                      //..do texto
```

C.F.6.3.1.a – Código fonte do início da função Compila

Nesta primeira parte do código, verifica-se a existência de tradução anterior a esta através do teste condicional ***if(comp==1)***, onde a variável ***comp*** é a variável de controle. A cada conversão, esta variável é colocada igual a 1, mas na inicialização do programa, antes da primeira conversão, seu valor é igual a zero. Se houve conversão, o arquivo temporário que armazena o código objeto é apagado.

Caso exista texto digitado pelo usuário, ***if(RichEdit1->Modified)***, a janela de apresentação dos resultados da conversão, ***Form3***, é limpa (***Form3->RichEdit1->Clean()***), recebe a frase ***"Programa Compilado"*** e possui uma linha saltada (***Form3->RichEdit1->Lines->Add(" ")***) para posterior inserção do código objeto.

Para que haja uma correta tradução, as próximas linhas de código fonte verificam se o usuário digitou a palavra ***END*** para que seja contabilizado o número de linhas do texto. Caso o programa não tenha sido finalizado, é apresentado o MessageBox da Figura 6.3.1.b e termina-se a conversão, colocando a variável

comp = 1 para que na próxima tradução o arquivo gerado seja apagado e criado um novo arquivo “file_out.bin” e a linha contendo o erro é destacada.

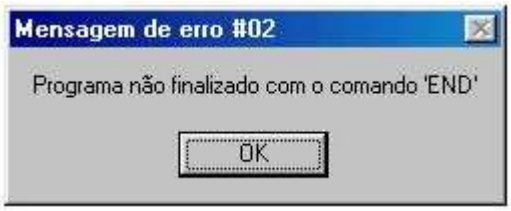


Figura 6.3.1.b – MessageBox informando erro

Na segunda parte do algoritmo de interpretação das linhas de texto, mostrado na próxima caixa contendo o código fonte, tem-se a transformação de todos os caracteres digitados pelo usuário para maiúsculos através do comando ***RichEdit1->Lines->Text.UpperCase()***, a variável inteira ***pos2*** recebe o comprimento total do texto e a variável inteira **pos** recebe a localização do comando ***END***, a qual é obtida através do procedimento **FindTextA** (mesmo procedimento utilizado internamente nos procedimentos da caixa de diálogo ***FindDialog***, exibida anteriormente, mas sem a presença da caixa de diálogo). Caso o comando não tenha sido digitado, ***pos<0*** e é exibida a mensagem da Figura 6.3.1.b, sendo finalizada a conversão.

Contudo, com a presença do comando ***END*** no código, passa-se a testar linha a linha através dos seguintes comandos:

```
RichEdit1->SelStart=RichEdit1->Perform(EM_LINEINDEX, i, 0);
RichEdit1->SelLength = RichEdit1->Lines->Strings [i].Length ();
```

Utilizando-se os mesmos procedimentos citados anteriormente para as propriedades ***SelStart*** e ***SelLength*** do componente ***RichEdit1***, com a variável ***i*** contendo o número da linha, até que se encontre o comando ***END*** para que se tenha o intervalo de linhas a serem lidas e comparadas com as instruções armazenadas no programa.

A variável ***teste***, do tipo ***AnsiString*** – uma forma de armazenamento de conjunto de caracteres – recebe a cada iteração o conteúdo da linha do texto, com caracteres maiúsculos e com os caracteres não imprimíveis (espaços e tabulações) retirados tanto da direita como da esquerda do texto, através dos comandos:

```
AnsiString teste = RichEdit1->SelText.UpperCase();
teste = teste.Trim();
```

Os dois testes finais: ***if(teste==”END”)*** e ***if(i==100)*** servem para identificar o número de linhas existentes e evitar um loop infinito de procura da linha contendo o comando ***END*** respectivamente. No primeiro, a variável ***n*** receberá o número de linhas e a variável ***j*** receberá o valor 1 para finalizar o laço formado pelo comando ***while***. O segundo, informa ao usuário a possibilidade de erro e faz

com que a rotina continue ou seja interrompida de acordo com o que for selecionado.

```
//Rotina de checagem do número de linhas do programa:
     RichEdit1->Lines->Text.UpperCase(); //Coloca as letras em maiúsculas
     int pos2=RichEdit1->Text.Length();  //Armazena o comprimento do texto
     //Checagem da existência da palavra "end" no texto
                  int   pos=RichEdit1->FindTextA("end",i,   pos2,   TSearchTypes()<<
stWholeWord);
     if (pos<0)
     {
       MessageBox(Handle, "Programa não finalizado com o comando 'END'",
       "Mensagem de erro #02", MB_OK);
       comp=1;
       return false;
     }

     while(j!=1)
     {
       //Seleciona a linha de índice "i"
       RichEdit1->SelStart=RichEdit1->Perform(EM_LINEINDEX, i, 0);
       RichEdit1->SelLength = RichEdit1->Lines->Strings [i].Length ();
       //var. teste recebe a linha do texto com todos os caracteres maiúsculos
       AnsiString teste = RichEdit1->SelText.UpperCase();
       // Retira os caracteres não imprimíveis a direita e a esquerda
       teste = teste.Trim();
       if(teste == "END")
       {
         j=1;
         n=i;
         i=0;
       }
       if (i==100) //Teste construído para evitar loop eterno
       {
         switch (Application->MessageBox(
            "Você está certo de ter digitado o comando 'END' corretamente?",
            "Montador - v.1.1", MB_YESNO))
         {
           case IDYES:
             break;
           case IDNO:
             return false;
         }
       }
       i++;
     }
```

C.F.6.3.1.b – Verificação do comando END e do número de linhas do texto

6.3.2. Comparação com os valores armazenados e passagem para hexadecimal

Esta parte do algoritmo da função Compila() baseia-se na utilização de vetores dinâmicos e ponteiros, bem como manipulação de strings a fim de que o texto digitado pelo usuário possa ser comparado com os valores padrões armazenados, devidamente traduzidos para hexadecimal e preparados para gravação a posteriori. Seu fluxograma básico pode ser visto na Figura 6.3.2.

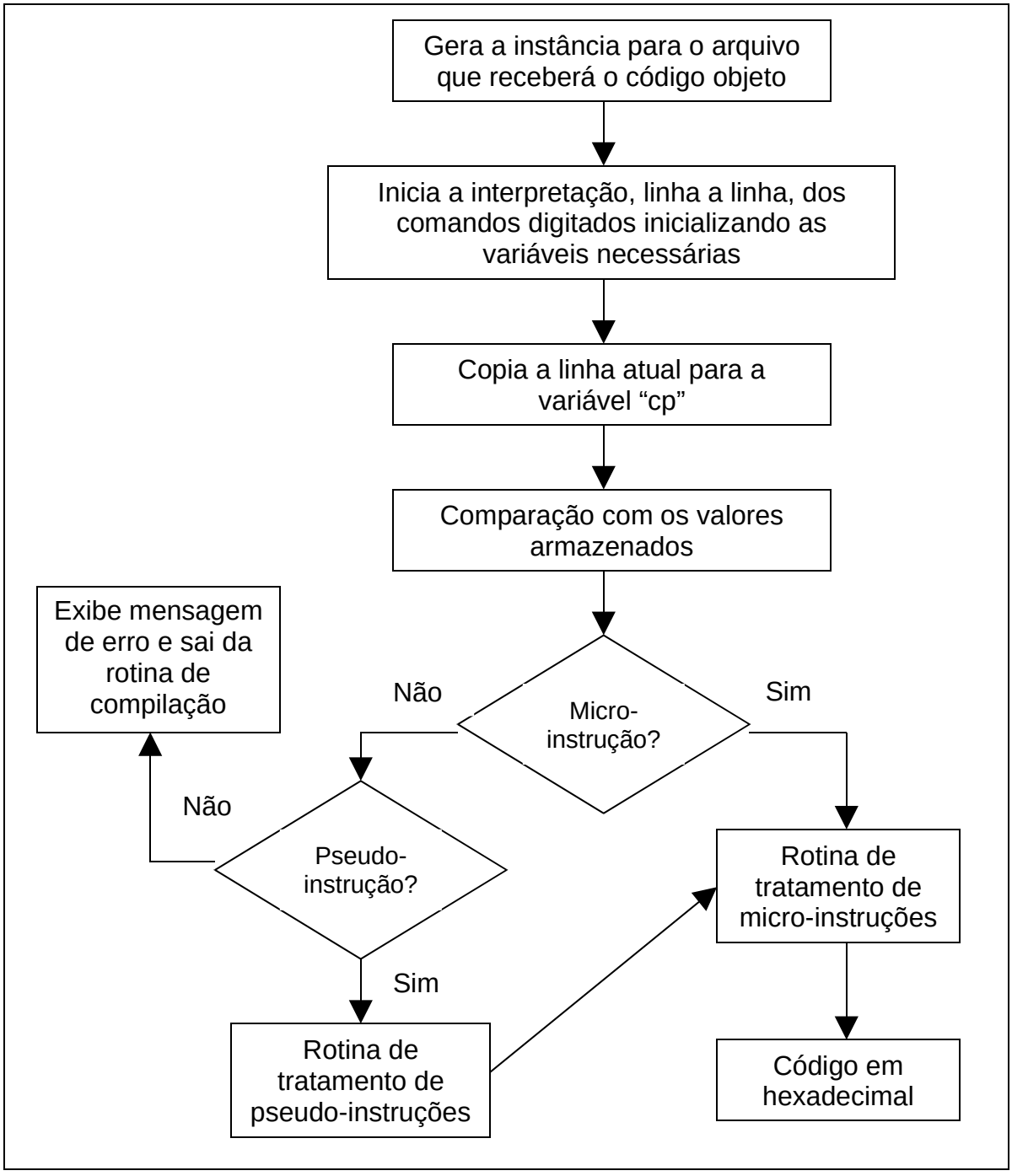


Figura 6.3.2 – Fluxograma da tradução de texto para hexadecimal

O código fonte desta parte da função compila foi dividido em módulos, pois dentro deste estão também as rotinas de tratamento de micro e pseudo-instruções, as quais serão analisadas a parte, bem como a rotina de gravação do número obtido em hexadecimal no arquivo de saída.

```
/*Criação do stream de saída out*/
   FILE *out;
   i=0;
   //Rotina de interpretação dos caracteres
   while (i!=n)
     {
        AnsiString texto = RichEdit1->Lines->Strings[i].Trim();
        texto = texto.UpperCase();
        char* cp = new char[texto.Length() + 1 ];
        strcpy( cp, texto.c_str() );
        int temp = texto.Length();

        int e=0;                  //Caracteriza a inexistência da instrução se e==0
        pseu=0;       //Se pseu==1, tratamento de pseudo-instrução. Se ==0, micro.
        shf=0;          //shf == 1, indica instrução com o formato da instr. shift
        off_shf=0;      //off_shf == 1, limite de deslocamento == + - 16
        pseu_ld=0;    //indica tratamento da pseudo LD se pseu_ld ==1
        .
        .
        .
        Rotinas de tratamento de micro-instruções (vide item 6.3.2.1)
        .
        .
        .
        Rotinas de tratamento de pseudo-instruções (vide item 6.3.2.2)
        .
        .
        .
         // Teste de instrução válida

        if (e==0)
        {
          MessageBox(Handle, "Instrução inexistente no set do processador",
                              "Mensagem de erro #06", MB_OK);
          RichEdit1->SelStart = RichEdit1->Perform (EM_LINEINDEX , i, 0 );
          RichEdit1->SelLength = RichEdit1->Lines->Strings [i] . Length ();
          comp=1;
          return false;
        }
        i++;
```

```
    }
```

C.F.6.3.2.a – Início da interpretação das linhas e teste de instrução existente

O arquivo de saída começa a ser gerado quando a atribuição de ponteiro, ***FILE *out*** é feita. Este ponteiro indica que será gerado um arquivo de saída cuja posição inicial é apontada pelo ponteiro ***out***.

A variável ***texto***, do tipo ***AnsiString***, recebe a linha de índice ***i*** do componente ***RichEdit1*** (linha atual), e possui todos os seus caracteres colocados como maiúsculos (***UpperCase***). Cria-se então o vetor dinâmico ***cp***, do tipo **char** (caracteres), cujo comprimento é igual ao número de caracteres da variável ***texto*** mais um, pois será colocado um caracter "***nulo***" (\0) na última posição para sinalizar o final da string. Como o vetor ***cp*** é dinâmico, dentro das rotinas de micro e pseudo-instruções, dependendo de qual seja utilizada ou no final do processo de conversão, ele será destruído para evitar o estouro de pilha, pois esta posição fica reservada, e como há utilização de outros ponteiros, o indicador de pilha perder-se-ia durante a execução do programa. Copia-se então o conteúdo de ***texto*** para ***cp*** utilizando-se o comando ***StrCopy*** em conjunto com a função ***.c_str()*** a qual fará a conversão de ***AnsiString*** para um vetor de caracteres.

A variável ***temp*** recebe o comprimento de ***text***, ou seja, o número de caracteres válidos, e então todas as variáveis auxiliares são inicializadas. Após o tratamento da instrução, como micro ou pseudo e com o auxílio das funções que tratam o tipo de instrução (R,I e J), encapsuladas externamente em relação à função estudada, testa-se a existência da instrução através da verificação da variável ***e***, que sendo nula, gera uma mensagem de erro e finaliza a rotina de conversão. Caso a instrução exista, incrementa-se a linha (***i++***) e a rotina de tratamento das linhas recomeça.

Mais uma vez ressaltando, o tratamento de micro e pseudo-instruções foi separado, mesmo sabendo-se que as micro-instruções fazem parte do conjunto de pseudo-instruções, mas conforme dito anteriormente, pela facilidade de separação em tipos das originalmente micro-instruções optou-se por seguir-se esta metodologia

6.3.2.1. Tratamento das micro-instruções

Esta rotina, em linhas gerais, é descrita pelo fluxograma da Figura 6.3.2.1. Trata as micro-instruções de acordo com o tipo de distribuição de campos nos 16 bits pertencentes a cada instrução, havendo a possibilidade de manipulação da formação da instrução para que ela se adeqüe a um dos três tipos (R, I e J) caso não esteja inclusa neste conjunto (micro-instrução ***NOT***, por exemplo). A fim de se exemplificar, será mostrado um dos conjuntos de micro-instruções e um exemplar de rotinas de tratamento de tipos de micro-instrução.

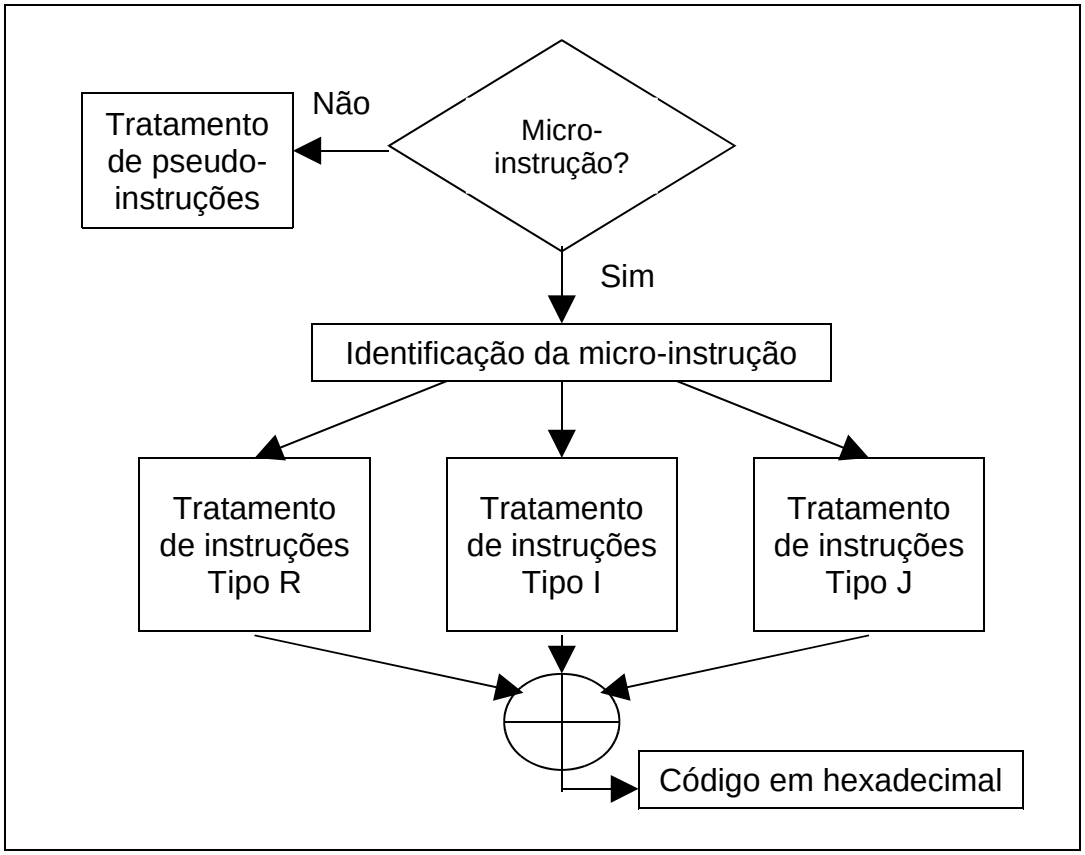


Figura 6.3.2.1 – Tratamento das micro-instruções

```
  .
 .
 if ((cp[0]=='J' & cp[1]==' ')|
    (cp[0]=='J' & cp[1]=='\t')|
    (cp[0]=='J' & cp[1]=='R')|
    (cp[0]=='J' & cp[1]=='A' & cp[2]=='L'& cp[3]!='P'&cp[3]!='R'
     &cp[3]!='D'&cp[4]!='D'&cp[4]!='C')|
    (cp[0]=='S' & cp[1]=='F' & cp[2]=='T' & cp[3]!='L' & cp[3]!='R')|
    (cp[0]=='L' & cp[1]=='U' & cp[2]=='I')|
    (cp[0]=='A' & cp[1]=='D' & cp[2]=='D' & cp[3]=='I')|
    (cp[0]=='N' & cp[1]=='O' & cp[2]=='T' & cp[3]!='I'))
{
  .
  .
```

C.F.6.3.2.1.a – Identificação da micro-instrução

A identificação da micro-instrução é realizada comparando-se as primeiras posições do vetor ***cp*** com valores padrão. Teve-se o cuidado de restringir as operações quanto ao uso de tabulação e de similares pseudo-instruções, pois se isto não fosse feito, a pseudo-instrução ***JALPC*** seria tratada como ***JAL***, pois o

tratamento da micro-instrução vem primeiro e ambas possuem os três primeiros caracteres idênticos.

```
        .
        .
         If (temp==1&cp[0]==’J’&cp[1]!=’A’)
         {
MessageBox(Handle, “Instrução JUMP sem offset”,
                          “Mensagem de erro #03”, MB_OK);
           comp=1;
           RichEdit1->SelStart = RichEdit1->Perform (EM_LINEINDEX , i, 0 );
           RichEdit1->SelLength = RichEdit1->Lines->Strings [i] . Length ();
           return false;
         }

         int l=0, t=0;
         while (cp[l]==’ ‘ |cp[l]==’J’|cp[l]==’A’|cp[l]==’L’|
               cp[l]==’\t’|cp[l]==’S’|cp[l]==’F’|cp[l]==’T’|
               cp[l]==’N’ |cp[l]==’O’|cp[l]==’U’|cp[l]==’I’|
               cp[l]==’T’|cp[l]==’D’)
          l++;
          cp2 = new char[50];
          while(l!=temp+1)
          {
              cp2[t] = cp[l];
              t++;
              l++;
           }
         .
         .
```

C.F.6.3.2.1.b – Transferência do vetor cp para cp2 e teste de offset

Após a restrição quanto a qual conjunto de instruções está sendo analisado, caso a instrução seja ***J*** (jump), testa-se se a mesma possui offset. Caso não possua, o programa exibe uma mensagem de erro e interrompe a tradução. O algoritmo prossegue contando o número de caracteres que compõem a instrução antes dos caracteres complementares, ou seja, para a instrução ***Not $s0*** ele contará quantos caracteres estão antes da declaração ***$s0***, que será utilizada como fonte para o tratamento dos tipos de instrução sendo copiada para ***cp2***. Portanto, neste exemplo, ***cp2=$s0***, lembrando-se do espaço destinado ao caracter sinalizador de final de vetor de caracteres: **\0**. Os tamanhos dos vetores utilizados foi escolhido de forma a garantir que toda a extensão da instrução seja analisada.

```
if (cp[0]==’N’)
{
   t--;
   cp2[t]=’,’;
   t++;
   cp2[t]=’0’;
   t++;
   cp2[t]=’\0’;
}

if ((cp[0]=='J')&(cp[1]=='R'))
{
   t--;
   cp2[t]=',';
   t++;
   cp2[t]='0';
   t++;
   cp2[t]='\0';
}

int testa_shift = 0;
if (cp[0]=='S')
{
   shf=1;
   off_shf=1;
   testa_shift=1;
}

if (cp[0]=='L')
{
   shf=1;
}

if (cp[0]=='A')
   shf=1;

volta = op_deslocamento();

if(testa_shift==1)
   if(volta==-140)
      return false;
else
   if(volta==-1)
      return false;
```

C.F.6.3.2.1.c – Tratamento das características de cada micro-instrução

As micro-instruções são tratadas individualmente. Na parte do código mostrada, a fim de se atender as especificações de campos da instrução de deslocamento (tipo J), nas instruções ***NOT*** e ***JR*** foi incluído via tratamento de posições num vetor, um offset nulo. Para isso, voltou-se em uma posição o ponteiro indicador de posição na matriz para que o espaço pertencente ao caracter **\0** fosse utilizado. Após a colocação de todos os valores, inseriu-se o sinalizador de fim de vetor, **\0**, em ***cp2***. Para as instruções ***Shift***, ***Lui*** e ***Addi*** foram configuradas as variáveis ***shf*** e ***off_shf*** para que o valor numérico apresentado em suas expressões fosse corretamente analizado, principalmente na instrução ***Shift***, para que o valor da constante ficasse compreendido no intervalo entre –16 e + 16.

Chama-se então a função do programa Montador – v1.1. responsável pelo tratamento de micro-instruções do tipo J, a função ***op_deslocamento()***, cujo código fonte será mostrado em seguida.

A variável de retorno ,***volta***, foi declarada como ***short***, pois este tipo de variável compreende números inteiros de ***16 bits***, conforme desejado pelo tamanho das micro-instruções, e recebe o valor advindo da função de tratamento do tipo de instrução, no caso, ***op_deslocamento()***. Se o valor for igual ao arbitrado como indicativo de erro uma mensagem é mostrada ao usuário e a conversão é finalizada.

```
short TForm2::op_deslocamento()
{
  /*
  Tipo J
  Operação Reg. Origem    Endereço
     4 bits         4bits             8 bits
  */

  ret=0;
  int exist_reg=0;
  char* cp4= new char[50];
  if (pseu==0)
    StrCopy(cp4,cp2);
  else
    StrCopy(cp4,rpseudo);
  short tipo_s=0;
  short tipo_t=0;
  short tipo_a=0;
  int l=1;
  char k='0';
```

C.F.6.3.2.1.d – Primeira parte da função op_deslocamento()

No início das funções de tratamento do tipo de micro-instrução, as variáveis de retorno (**ret**) e teste de existência de registrador (**exist_reg**, no caso), são

inicializadas, e como estas funções também são utilizadas por pseudo-instruções, testa-se a variável **pseu**, que no caso é igual a zero, o que faz com que o vetor **cp4** receba o conteúdo de **cp2**. As variáveis de tipo de registrador, para registradores que possuem mais de uma unidade em seu conjunto (como em $t que pode ser $t0, $t1 ou $t2), são inicializadas, bem como a variável de indicação do campo a ser analisado(**k**), a qual pode assumir os valores 0 e 2 nesta função, pois o primeiro campo é determinado pelo valor correspondente ao operando e será acrescentado no final da rotina de tratamento das instruções, fora da função **op_deslocamento()**, conforme será mostrado mais adiante.

```
while(cp4[l]!='\0')
{
  if(cp4[l]=='Z'&cp4[l+1]=='E'&cp4[l+2]=='R'&cp4[l+3]=='O')
  {
    ret=0;
    l=l+4;
    k='2';
    exist_reg=1;
  }

   if(cp4[l]=='T')
    {
      l++;
      tipo_t=256;
      if(cp4[l]=='0')
        tipo_t=tipo_t;   //a soma depende da posição na estrutura
      if(cp4[l]=='1')
        tipo_t=tipo_t+256;
      if(cp4[l]=='2')
        tipo_t=tipo_t+512;
      if(cp4[l]!='0'&cp4[l]!='1'&cp4[l]!='2')
      {
        MessageBox(Handle,"Registrador inexistente no set do processador",
                                  "Mensagem de erro #06", MB_OK);
        RichEdit1->SelStart = RichEdit1->Perform (EM_LINEINDEX , i, 0 );
        RichEdit1->SelLength = RichEdit1->Lines->Strings [i] . Length ();
        comp=1;
        return (-1);
      }
      ret = tipo_t;
      exist_reg=1;
      k='2';
      l++;
   }
   .
   .
```

C.F.6.3.2.1.e – Segunda parte da função op_deslocamento()

Na segunda parte do algoritmo, faz-se a codificação do registrador presente na instrução digitada pelo usuário, lembrando que esta instrução possui como campos: operando, registrador base e constante de deslocamento (4, 4 e 8 bits respectivamente). Caso o usuário tenha digitado de forma incorreta ou o registrador não seja compatível com os valores armazenados pelo programa, será apresentada uma mensagem de erro ao usuário e a conversão termina. Para cada registrador há um valor específico, de acordo com a posição por ele ocupada dentro dos campos da instrução. Foram mostrados apenas a codificação de dois registradores no código fonte apresentado na página anterior (C.F.6.3.2.1.e), pois todos os outros registradores seguem um destes dois modelos. O próximo passo é a codificação da constante.

```
      .
      .
      .
      if(k=='2')
      {
        if(cp4[l]==',')
          l++;
        if(cp4[l]=='$')
        {
          MessageBox(Handle,
          "Valor de offset utilizado inválido. O mesmo deve ser numérico",
          "Mensagem de erro #14", MB_OK);
          RichEdit1->SelStart = RichEdit1->Perform (EM_LINEINDEX , i, 0 );
          RichEdit1->SelLength = RichEdit1->Lines->Strings [i] . Length ();
          ret = (-1);
          return ret;
        }
        if(cp4[l]==' '|cp4[l]=='\0')
        {
          MessageBox(Handle,
            "Valor de offset utilizado inválido ou com espaço entre a vírgula e o
número",
                              "Mensagem de erro #07", MB_OK);
          RichEdit1->SelStart = RichEdit1->Perform (EM_LINEINDEX , i, 0 );
          RichEdit1->SelLength = RichEdit1->Lines->Strings [i] . Length ();
          ret = (-1);
          return ret;
        }

        char *ctemp=new char[6];
        int f=0;
```

```
for(f=0;f<7;f++)
        {
          ctemp[f]=cp4[l];
          l++;
        }

if(ctemp[0]!='0'&ctemp[0]!='1'&ctemp[0]!='2'&ctemp[0]!='3'&
            ctemp[0]!='4'&ctemp[0]!='5'&ctemp[0]!='6'&ctemp[0]!='7'&
            ctemp[0]!='8'&ctemp[0]!='9'&ctemp[0]!='-')
        {
          MessageBox(Handle,
 "Valor de offset utilizado inválido ou espaçamento entre a vírgula e o número",
                    "Mensagem de erro #07", MB_OK);
          RichEdit1->SelStart = RichEdit1->Perform (EM_LINEINDEX , i, 0 );
          RichEdit1->SelLength = RichEdit1->Lines->Strings [i] . Length ();
          ret = (-1);
          return ret;
        }
        if(ctemp[1]!='0'&ctemp[1]!='1'&ctemp[1]!='2'&ctemp[1]!='3'&
            ctemp[1]!='4'&ctemp[1]!='5'&ctemp[1]!='6'&ctemp[1]!='7'&
            ctemp[1]!='8'&ctemp[1]!='9'&ctemp[1]!='\0'&ctemp[1]!='\r')
        {
          MessageBox(Handle,
 "Valor de offset utilizado inválido ou espaçamento entre a vírgula e o número",
                    "Mensagem de erro #07", MB_OK);
          RichEdit1->SelStart = RichEdit1->Perform (EM_LINEINDEX , i, 0 );
          RichEdit1->SelLength = RichEdit1->Lines->Strings [i] . Length ();
          ret = (-1);
          return ret;
        }
        if(ctemp[2]!='0'&ctemp[2]!='1'&ctemp[2]!='2'&ctemp[2]!='3'&
            ctemp[2]!='4'&ctemp[2]!='5'&ctemp[2]!='6'&ctemp[2]!='7'&
            ctemp[2]!='8'&ctemp[2]!='9'&ctemp[2]!='\0'&ctemp[2]!='\r'&
            ctemp[1]!='\0'&ctemp[1]!='\r')
        {
          MessageBox(Handle,
 "Valor de offset utilizado inválido ou espaçamento entre a vírgula e o número",
                    "Mensagem de erro #07", MB_OK);
          RichEdit1->SelStart = RichEdit1->Perform (EM_LINEINDEX , i, 0 );
          RichEdit1->SelLength = RichEdit1->Lines->Strings [i] . Length ();
          ret = (-1);
          return ret;
        }
        l=l-f;
```

```
short ret2=0;
```

```
 if(shf==1)
{
   ret2=atoi(ctemp);
   shf=0;
   if((off_shf==1)&(ret2>16))
   {
      MessageBox(Handle,"Valor de deslocamento utilizado maior que 16"
                 ,"Mensagem de erro #08", MB_OK);
      RichEdit1->SelStart = RichEdit1->Perform (EM_LINEINDEX , i, 0 );
      RichEdit1->SelLength = RichEdit1->Lines->Strings [i] . Length ();
       off_shf=0;
      ret = (-140);
      return ret;
   }
    if((off_shf==1)&(ret2<-16))
   {
      MessageBox(Handle,"Valor de deslocamento utilizado menor que -16"
                 ,"Mensagem de erro #08", MB_OK);
      RichEdit1->SelStart = RichEdit1->Perform (EM_LINEINDEX , i, 0 );
      RichEdit1->SelLength = RichEdit1->Lines->Strings [i] . Length ();
      off_shf=0;
      ret = (-140);
      return ret;
   }
   if((off_shf==0)&(ret2>127)|(off_shf==0)&(ret2<-127))
   {
  MessageBox(Handle,"Valor de offset utilizado maior que 255 ou menor que
-128","Mensagem de erro #09", MB_OK);
      RichEdit1->SelStart = RichEdit1->Perform (EM_LINEINDEX , i, 0 );
      RichEdit1->SelLength = RichEdit1->Lines->Strings [i] . Length ();
      ret = (-1);
      return ret;
   }
   if(ret2<0)
   {
      unsigned short ret3=ret2;
      ret3=ret3<<8;
      ret3=ret3>>8;
      ret2=ret3;
    }
}
else
{
```

```
        ret2= atoi(ctemp);

        if(ret2>127|ret2<-128)
        {
          MessageBox(Handle,
               "Valor de offset utilizado maior que 127 ou menor que -128",
                  "Mensagem de erro #10", MB_OK);
          RichEdit1->SelStart = RichEdit1->Perform (EM_LINEINDEX , i, 0 );
          RichEdit1->SelLength = RichEdit1->Lines->Strings [i] . Length ();
          ret = (-1);
          return ret;
        }
        if(ret2<0)
        {
           unsigned short ret3=ret2;
           ret3=ret3<<8;
           ret3=ret3>>8;
           ret2=ret3;
        }
     }
    ret = ret | ret2;
    cp4[l+1]='\0';
    delete ctemp;
    }
```

C.F.6.3.2.1.f – Terceira parte da função op_deslocamento

Nesta parte da função, será comparado o número digitado pelo usuário com os intervalos possíveis de valores, de acordo com cada tipo de instrução.

O tratamento da constante inicia-se testando se o valor existente na posição ***cp[l]*** do vetor é numérico. Caso seja um **'$'**, indicando que o usuário digitou um registrador, é mostrada uma mensagem de erro e a tradução termina. Testa-se em seguida se há espaço entre a vírgula e a constante ou se não foi digitado um valor de offset, e da mesma forma é apresentada a mensagem de erro correspondente ao problema e a função ***Compila()*** é finalizada.

Cria-se então o vetor **ctemp**, que será utilizado para a conversão da constante de caracter para valor numérico. Assim como foi para as transferências entre vetores anteriormente, utiliza-se a indexação para separar a parte de interesse no texto da instrução: a constante. Para uma conversão correta, são criadas e inicializadas as variáveis **minus** e **ret2**, utilizadas para indicar se a constante é negativa e armazenar o resultado temporário, respectivamente.

Caso ***shf == 1***, a constante é convertida e se ***off_shf ==1***, tem-se que o limite de valores está entre –16 e +16, pois só há sentido em existir deslocamento de no máximo o mesmo número de bits da palavra. Em cada teste de intervalo para as possibilidades de ***off_shf***, caso seja encontrado algum valor discrepante, é apresentada uma mensagem de erro e o programa interrompe a tradução. Para a conversão de um vetor de caracteres para um valor numérico foi utilizada a

função ***atoi*** da linguagem C++. Se ***ret2<0***, é necessária a troca das partes mais e menos significativas, pois a indicação de números negativos começa com 1, o que trará problemas no final das funções de tipo de instrução (no caso, ***op_deslocamento()***), pois será realizada uma operação ***OU*** lógica ao final do algoritmo a fim de se obter o resultado da codificação, armazenado na variável ***ret***. Para o término do laço ***while(cp4[l]!='\0')***, coloca-se como o próximo caracter após a constante o ***nulo***, pois quando o índice ***l*** for incrementado, apontará para este caracter. A cada volta no laço, o vetor ***ctemp*** é criado e destruído, para que não interfira em outros espaços de memória, conforme justificado anteriormente.

```
        if (cp4[l]=='.')
        {
          MessageBox(Handle,"Ponto utilizado no lugar de vírgula na instrução",
          "Mensagem de erro #13", MB_OK);
          RichEdit1->SelStart = RichEdit1->Perform (EM_LINEINDEX , i, 0 );
          RichEdit1->SelLength = RichEdit1->Lines->Strings [i] . Length ();
          comp=1;
          return (-1);
        }
        if ((cp4[l]!=','&cp4[l]!='$'&cp4[l]!='\0'&cp4[l]!='\t'&cp4[l]!='0'
           &cp4[l]!='1'&cp4[l]!='2'&cp4[l]!='3'&cp4[l]!='4'&cp4[l]!='5'
           &cp4[l]!='6'&cp4[l]!='7'&cp4[l]!='8'&cp4[l]!='9'&cp4[l]!='-') &(exist_reg==0))
        {
          MessageBox(Handle,"Registrador inexistente no set do processador",
                                  "Mensagem de erro #11", MB_OK);
          RichEdit1->SelStart = RichEdit1->Perform (EM_LINEINDEX , i, 0 );
          RichEdit1->SelLength = RichEdit1->Lines->Strings [i] . Length ();
          comp=1;
          return (-1);
        }
      l++;
      exist_reg=0;
    }
    delete cp4;
    return ret;
}
```

C.F.6.3.2.1.g – Última parte da função op_deslocamento

A função de tratamento dos tipos de campo é finalizada com os testes para verificar se o usuário digitou “.” ao invés de “,” durante a instrução e se algum caracter presente na linha é inválido. A variável que indica a existência do registrador é colocada para seu valor original para um novo laço, o contador de caracter é incrementado e volta-se ao início da rotina. Caso os caracteres da linha tenham terminado, o programa sai do laço de repetição, o vetor **cp4** é destruído e é retornado o valor contido em ***ret***. Como as instruções são de 16 bits, a função é

declarada como ***short op_deslocamento()*** pois o valor de retorno deve ser também de 16 bits. As outras funções (***op_tipo_r()*** e ***op_imeddiate()***) foram criadas da mesma forma que ***op_deslocamento()*** e seus códigos estão mostrados no item Anexo, capítulo 5, item 5.2.

```
if(cp[0]=='N')
{
  short not = 40960;
  volta = volta|not;
}
if(cp[0]=='L')
{
  short lui = 45056;
  volta = volta|lui;
}
if(cp[0]=='J'&cp[1]!='A')  //instrução J
{
  unsigned short jump = 57344;
  volta = volta | jump;
}
if(cp[0]=='J'&cp[1]=='A')
{
  unsigned short jal = 61440;
  volta = volta | jal;
}
if(cp[0]=='A')
{
  short addi = 32768;
  volta = volta | addi;
}
if(cp[0]=='S')
{
  unsigned short shift = 36864;
  volta = volta | shift;
}
```

C.F.6.3.2.1.h – Geração do valor final da micro-instrução

Para a obtenção do valor final da instrução em hexadecimal, é acrescentado o valor do operando na variável ***volta*** (contém o valor ***ret***, originário da função de tratamento de tipos de campo), através das funções lógicas OU e E, dependendo do valor do operando. A rotina de tratamento da instrução é finalizada com a gravação do valor obtido no arquivo temporário de saída.

6.3.2.2 Tratamento das pseudo-instruções

Conforme dito anteriormente no capítulo 4, seção sobre as pseudo-instruções, elas são combinações de micro-instruções, e para o conjunto de pseudo-instruções especificado tem-se os arranjos de micro-instruções descritos para cada pseudo nas próximas tabelas. O algoritmo de tratamento de pseudo-instruções está descrito na Figura 6.3.2.2.

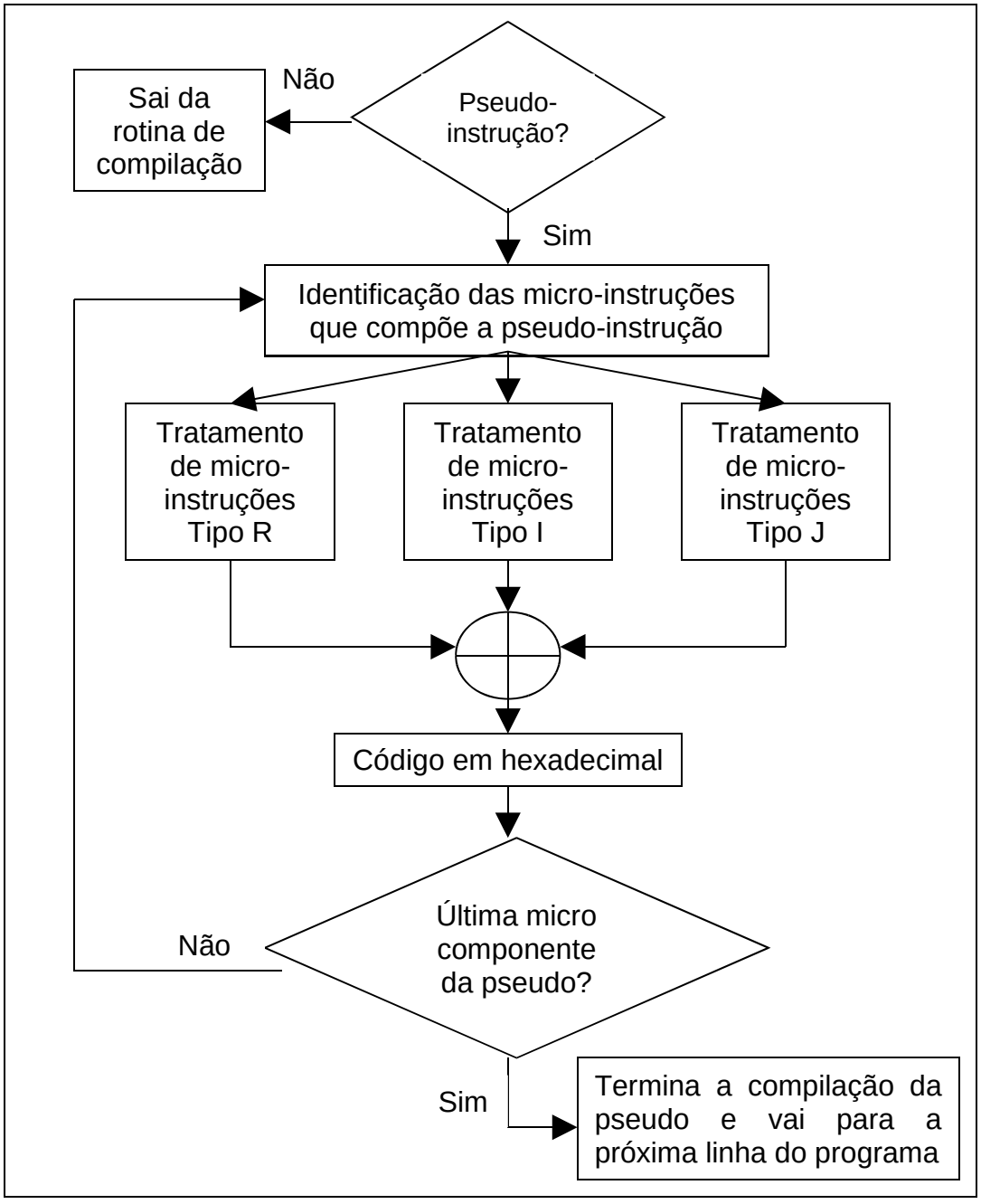


Figura 6.3.2.2 – Fluxograma de tratamento de pseudo-instruções

Add $s3, $s1 , $s2	Soma o valor de $s1 com $s2 e armazena em $s3
Add $s3, $s1 , $s2	Operação direta da micro-instrução

Tabela 6.3.2.2.1 – Pseudo-instrução Add

Sub $s3, $s1 , $s2	Subtrai o valor de $s2 de $s1 e armazena em $s3
Sub $s3, $s1 , $s2	Operação direta da micro-instrução

Tabela 6.3.2.2.2 – Pseudo-instrução Sub

Mul $s3, $s1 , $s2	Multiplica o valor de $s1 por $s2 e armazena em $s3 versão 1
Add $t2, $Zero,$Zero	Armazena zero em $t2
Addi $t3, $Zero,16	Armazena 16 em $t3
Add $t1, $s1, $Zero	Armazena $s1 em $t1
Shift $t1, 15	Desloca $t1 15 para a direita
Shift $t1, -15	Desloca $t1 15 para a esquerda
Beq $t1 , $Zero,3	Se $t1 for zero salta três linhas
Add $s3, $s3, $s2	Soma $s3 com $s2 três armazena em $s3
Shift $s2, -1	Desloca $s2 um bit a esquerda
J $pc, 4	Salta quatro linhas
Beq $s2, $zero, 4	Se $s2 for zero salta quatro linhas
Shift $s1, 1	Desloca $s1 um bit a direita
Addi $t2, $t2 , 1	Soma um a $t2
Blt $t2, $t3, -10	Se $t2 for menor que $t3 volta dez linhas

Tabela 6.3.3.2.3 – Pseudo-instrução Mul Versão 1

Mul $s3, $s1 , $s2	Multiplica o valor de $s1 por $s2 e armazena em $s3 versão 2

Add $t2, $Zero,$Zero	Armazena zero em $t2
Addi $t3, $Zero,16	Armazena 16 em $t3
Add $t1, $s1, $Zero	Armazena $s1 em $t1
Shift $t1, 15	Desloca $t1 15 para a direita
Shift $t1, -15	Desloca $t1 15 para a esquerda
Beq $t1 , $Zero,3	Se $t1 for zero salta três linhas
Add $s3, $s3, $s2	Soma $s3 com $s2 três armazena em $s3
Shift $s2, -1	Desloca $s2 um bit a esquerda
J $pc, 5	Salta cinco linhas
Beq $s2, $zero, 4	Se $s2 for zero salta quatro linhas
Shift $s1, 1	Desloca $s1 um bit a direita
Beq $s1, $zero, 2	Se $s1 for zero salta duas linhas
Addi $t2, $t2 , 1	Soma um a $t2
Blt $t2, $t3, -11	Se $t2 for menor que $t3 volta onze linhas

Tabela 6.3.2.2.4 – Pseudo-instrução Mul Versão 2

Mul $s3, $s1 , $s2	Multiplica o valor de $s1 por $s2 e armazena em $s3 versão 3
Add $s3, $zero, $zero	Armazena zero em $s3
Addi $t2, $zero, 1	Armazena 1 em $t2
And $t1, $s1, $t2	Calcula e booleano de $t2 e $s1 e armazena em $t1
Beq $t1 , $Zero,1	Se $t1 for zero salta uma linha
Add $s3, $s3, $s2	Soma $s3 com $s2 três armazena em $s3
Shift $s1, 1	Desloca $s1 um bit à direita
Shift $s2, -1	Desloca $s2 um bit à esquerda
Beq $s1, $zero, 1	Se $s1 for zero salta uma linha
J -6	Volta seis linhas

Tabela 6.3.2.2.5 – Pseudo-instrução Mul Versão 3

Div $s3, $s1 , $s2	Divide o valor de $s2 por $s1 e armazena em $s3
Add $s3, $zero, $zero	Armazena zero em $s3
Slt $t1 , $s1, $s2	Seta $t1 se $s1 for menor que $s2
Shift $s3, -1	Desloca $s3 um bit à esquerda
Beq $t1, $zero, 1	Se $t1 for zero salta uma linha
Add $s3, $s3 , $t1	Soma $s3 com $t1 três armazena em $s3
Shift $s2 , 1	Desloca $s2 um bit à direita
Beq $s2, $zero, 1	Se $s2 for zero salta uma linha
J -6	Volta seis linhas

Tabela 6.3.2.2.6 – Pseudo-instrução Div

Addi $s1 , 100	Soma o valor de $s1 com uma constante e armazena em $s1
Addi $s1 , 100	Operação direta da micro-instrução

Tabela 6.3.2.2.7 – Pseudo-instrução Addi

Subi $s1 , 100	Subtrai o valor de uma constante de $s1 e armazena em $s1
Addi $t1, $Zero, 100	Armazena a constante em $t1
Sub $s1, $s1, $t1	Efetua a subtração

Tabela 6.3.2.2.8 – Pseudo-instrução Subi

Muli $s3, $s2 , 100	Multiplica o valor de $s2 por uma constante e armazena em $s3
Addi $s1, $s1, 100	Armazena a constante em $s1
Add $s3, $zero, $zero	Armazena zero em $s3
Addi $t2, $zero, 1	Armazena 1 em $t2
And $t1, $s1, $t2	Calcula e booleano de $t2 e $s1 e armazena em $t1
Beq $t1 , $Zero, 1	Se $t1 for zero salta uma linha
Add $s3, $s3, $s2	Soma $s3 com $s2 três armazena em $s3

Shift $s1, 1	Desloca $s1 um bit à direita
Shift $s2, -1	Desloca $s2 um bit à esquerda
Beq $s1, $zero, 1	Se $s1 for zero salta uma linha
J -6	Volta seis linhas

Tabela 6.3.2.2.9 – Pseudo-instrução Muli

Divi $s3, $s2 , 100	Divide o valor de $s2 por uma constante e armazena em $s3
Addi $s1, $s1, 100	Armazena a constante em $s1
Add $s3, $zero, $zero	Armazena zero em $s3
Slt $t1 , $s1, $s2	Seta $t1 se $s1 for menor que $s2
Shift $s3, -1	Desloca $s3 um bit à esquerda
Beq $t1, $zero, 1	Se $t1 for zero salta uma linha
Add $s3, $s3 , $t1	Soma $s3 com $t1 três armazena em $s3
Shift $s2 , 1	Desloca $s2 um bit à direita
Beq $s2, $zero, 1	Se $s2 for zero salta uma linha
J -6	Volta seis linhas

Tabela 6.3.2.2.10 – Pseudo-instrução Divi

Rem $s1, $s2	Armazena o resto da divisão de $s1 por $s2 em $s1
Add $s3, $zero, $zero	Armazena zero em $s3
Slt $t1 , $s1, $s2	Seta $t1 se $s1 for menor que $s2
Shift $s3, -1	Desloca $s3 um bit à esquerda
Beq $t1, $zero, 1	Se $t1 for zero salta uma linha
Add $s3, $s3 , $t1	Soma $s3 com $t1 três armazena em $s3
Shift $s2 , 1	Desloca $s2 um bit à direita
Beq $s2, $zero, 1	Se $s2 for zero salta uma linha
J -6	Volta seis linhas

Tabela 6.3.2.2.11 – Pseudo-instrução Rem

Sftl $s1, $s2, 100	Desloca $s1 à esquerda do valor da

	soma de $s2 com a constante de 7 bits e armazena em $s1
Shift $s1 , $s2, -100	Operação direta da micro-instrução atribuindo um valor negativo à constante de 7 bits

Tabela 6.3.2.2.12 – Pseudo-instrução Sftl

Sftr $s1 , $s2, 100	Desloca $s1 à direita do valor da soma de $s2 com a constante de 7 bits e armazena em $s1
Shift $s1 , $s2, 100	Operação direta da micro-instrução atribuindo um valor positivo à constante de 7 bits

Tabela 6.3.2.2.13 – Pseudo-instrução Sftr

And $s1 , $s2, $s3	AND booleano bit a bit entre $s2 e $s3 e armazena em $s1
And $s1 , $s2, $s3	Operação direta da micro-instrução

Tabela 6.3.2.2.14 – Pseudo-instrução And

Or $s1 , $s2, $s3	OR booleano bit a bit entre $s2 e $s3 e armazena em $s1
Or $s1 , $s2, $s3	Operação direta da micro-instrução

Tabela 6.3.2.2.15 – Pseudo-instrução Or

Nor $s1 , $s2, $s3	NOR booleano bit a bit entre $s2 e $s3 e armazena em $s1
Or $s1 , $s2, $s3	Operação direta da micro-instrução
Not $s1	Operação direta da micro-instrução

Tabela 6.3.2.2.16 – Pseudo-instrução Nor

Xor $s1 , $s2, $s3	XOR booleano bit a bit entre $s2 e $s3 e armazena em $s1
Xor $s1 , $s2, $s3	Operação direta da micro-instrução

Tabela 6.3.2.2.17 – Pseudo-instrução Xor

Not $s1	NOT booleano bit a bit de $s1 e armazena em $s1
Not $s1	Operação direta da micro-instrução

Tabela 6.3.2.2.18 – Pseudo-instrução Not

Comp $s1	Complemento a dois de $s1 e armazena em $s1
Sub $s1, $ Zero, $s1	Subtrai $s1 de $Zero resultando no complemento a dois de $s1 e armazena em $s1

Tabela 6.3.2.2.19 – Pseudo-instrução Comp

Andi $s1 , 100	AND booleano bit a bit entre $s1 e uma constante e armazena em $s1
Addi $t1, $Zero,100	Armazena a constante em $t1
And $s1 $s1, $t1	Operação direta da micro-instrução

Tabela 6.3.2.2.20 – Pseudo-instrução Andi

Ori $s1 , 100	OR booleano bit a bit entre $s1 e uma constante e armazena em $s1
Addi $t1, $Zero,100	Armazena a constante em $t1
Or $s1 $s1, $t1	Operação direta da micro-instrução

Tabela 6.3.2.2.21 – Pseudo-instrução Ori

Xori $s1 , 100	XOR booleano bit a bit entre $s1 e uma constante e armazena em $s1
Addi $t1, $Zero,100	Armazena a constante em $t1
Xor $s1 $s1, $t1	Operação direta da micro-instrução

Tabela 6.3.2.2.22 – Pseudo-instrução Xori

Lw $s1, $s2, $s3	Carrega em $s1 a palavra armazenada no endereço $s2 deslocado do valor armazenado em $s3
Lw $s1, $s2, $s3	Operação direta da micro-instrução

Tabela 6.2.3.3.23 – Pseudo-instrução Lw

Sw $s1, $s2, $s3	Salva a palavra armazenada em $s1 no endereço $s2 deslocado do valor armazenado em $s3
Sw $s1, $s2, $s3	Operação direta da micro-instrução

Tabela 6.3.2.2.24 – Pseudo-instrução Sw

Lb $s1, $s2, $s3	Carrega em $s1 o byte menos significativo armazenado no endereço $s2 deslocado do valor armazenado em $s3
Lw $t1, $s2, $s3	Carrega palavra armazenada no endereço $s2 deslocado do valor armazenado em $s3e salva em $s1
Shift $t1, $Zero,-8	Desloca o registrador oito vezes para a esquerda
Shift $t1, $Zero,+8	Desloca o registrador oito vezes para a direita
Shift $s1, $Zero,+8	Desloca o registrador oito vezes para a direita
Shift $s1, $Zero,-8	Desloca o registrador oito vezes para a esquerda

Add $s1, $s1, $t1	Concatena os oito bits mais significativos com os oito bits menos significativos da palavra armazenada no endereço $s2 e guarda em $s1

Tabela 6.3.2.2.25 – Pseudo-instrução Lb

Sb $s1, $s2, $s3	Salva o byte menos significativo de $s1 no endereço $s2 deslocado da constante do valor armazenado em $s3
Add $t1, $Zero,$s1	Carrega o valor armazenado em $s1 em $t1
Shift $t1, $Zero,-8	Desloca o $t1 oito vezes para a esquerda
Shift $t1, $Zero,+8	Desloca o $t1 oito vezes para a direita
Lw $t2, $s2, $s3	Carrega palavra armazenada no endereço $s2 deslocado do valor armazenado em $s3 e salva em $t2
Shift $t2, $Zero,+8	Desloca o $t2 oito vezes para a direita
Shift $t2, $Zero,-8	Desloca o $t2 oito vezes para a esquerda
Add $s1, $t1, $t2	Concatena os oito bits mais significativos de $s1 com os oito bits menos significativos da palavra armazenada no endereço $s2 e guarda em $s1
Sw $s1, $s2, $s3	Salva a palavra armazenada em $s1 no endereço $s2 deslocado do valor

	armazenado em $s3

Tabela 6.3.2.2.26 – Pseudo-instrução Sb

Ld $s1, $s2, (100)$s3	Carrega em $s1 e $s2 duas palavras armazenadas no endereço $s3 deslocado da constante e no próximo
Addi $t1, $Zero, 100	Armazena o valor da constante em $t1
Lw $s1, $t1, $s3	Carrega palavra armazenada no endereço $t1 deslocado do valor armazenado em $s3 e salva em $s1
Addi $s3, $s3, 1	Soma 1 ao valor de $s3
Lw $s2, $t1, $s3	Carrega palavra armazenada no endereço $t1 deslocado do valor armazenado em $s3 e salva em $s2

Tabela 6.3.2.2.27 – Pseudo-instrução Ld

Sd $s1, $s2, (100)$s3	Salva o conteúdo de $s1 e $s2 no endereço $s3 deslocado da constante
Addi $t1, $Zero, 100	Armazena o valor da constante em $t1
Sw $s1, $t1, $s3	Salva a palavra armazenada em $s1 no endereço $t1 deslocado do valor armazenado em $s3
Addi $s3, $s3, 1	Soma 1 ao valor de $s3
Sw $s2, $t1, $s3	Salva a palavra armazenada em $s2 no endereço $t1 deslocado do valor armazenado em $s3

Tabela 6.3.2.2.28 – Pseudo-instrução Sd

Lui $s1, 100	Carrega a constante nos oito bits mais significativos de $s1

Lui $s1, 100	Operação direta da micro-instrução

Tabela 6.3.2.2.29 – Pseudo-instrução Lui

Lwi $s1, 100	Carrega uma constante de 16 bits num registrador
Add $s1, $Zero,$Zero	Armazena o valor 0 em $s1
Lui $s1, mais significativa	Carrega a parte mais significativa da constante em $s1
Addi $s1, $s1, menos significativa	Soma $s1 com a parte menos significativa da constante e armazena em $s1

Tabela 6.3.2.2.30 – Pseudo-instrução Lwi

Mov $s1, $s2	Move o conteúdo de $s1 para $s2
Add $t1, $s1, $Zero	Armazena o valor de $s1 em $t1
Add $s1, $s2, $Zero	Armazena o valor de $s2 em $s1
Add $s2, $t1, $Zero	Armazena o valor de $t1 em $s2

Tabela 6.3.2.2.31 – Pseudo-instrução Mov

Mflo $s1	Move o byte menos significativo de $s1 para o mais significativo
Shift $s1, $Zero,-8	Desloca o $s1 oito vezes para a esquerda

Tabela 6.3.2.2.32 – Pseudo-instrução Mflo

Mfhi $s1	Move o byte mais significativo de $s1 para o menos significativo
Shift $s1, $Zero,+8	Desloca o $s1 oito vezes para a direita

Tabela 6.3.2.2.33 – Pseudo-instrução Mfhi

Move $s1, $s2	Move o conteúdo de $s2 para $s1 guardando 0 em $s2
Add $s1, $s2, $Zero	Copia o conteúdo de $s2 para $s1
Add $s1, $Zero,$Zero	Armazena 0 em $s2

Tabela 6.3.2.2.34 – Pseudo-instrução Move

Chg $s1, $s2	Move o conteúdo de $s2 para $s1 e o conteúdo de $s1 para $s2
Add $t1, $s1, $Zero	Copia o conteúdo de $s1 para $t1
Add $s1, $s2, $Zero	Copia o conteúdo de $s2 para $s1
Add $s2, $t1, $Zero	Copia o conteúdo de $t1 para $s2

Tabela 6.3.2.2.35 – Pseudo-instrução Chg

Slt $s1, $s2, $s3	Torna $s1=1 se $s2 < $s3 senão $s1=0
Slt $s1, $s2, $s3	Operação direta da micro-instrução

Tabela 6.3.2.2.36 – Pseudo-instrução Slt

Sle $s1, $s2, $s3	Torna $s1=1 se $s2 < ou = $s3 senão $s1=0
Slt $s1, $s3, $s2	Torna $s1=1 se $s3 < $s2 senão $s1=0
Not $s1	Inverte o conteúdo de $s1

Tabela 6.3.2.2.37 – Pseudo-instrução Sle

Seq $s1, $s2, $s3	Torna $s1=1 se $s2 = $s3 senão $s1=0
Beq $s1, $s2, 3	Pula as próximas duas linhas se $s1 for igual a $s2
Addi $s1, $Zero,$Zero	Atribui o valor “zero” a $s1
J $pc, 2	Vai para o fim
Addi $s1, $Zero,1	Atribui o valor “um” a $s1

Tabela 6.3.2.2.38 – Pseudo-instrução Seq

Sne $s1, $s2, $s3	Torna $s1=1 se $s2 < > $s3 senão $s1=0
Beq $s1, $s2, 3	Pula as próximas duas linhas se $s1 for

	igual a $s2
Addi $s1, $Zero,1	Atribui o valor “1” a $s1
J $pc, 2	Vai para o fim
Addi $s1, $Zero,$Zero	Atribui o valor “0” a $s1

Tabela 6.3.2.2.39 – Pseudo-instrução Sne

Sgt $s1, $s2, $s3	Torna $s1=1 se $s2 > $s3 senão $s1=0
Slt $s1, $s3, $s2	Operação direta da micro-instrução

Tabela 6.3.2.2.40 – Pseudo-instrução Sgt

Sge $s1, $s2, $s3	Torna $s1=1 se $s2 > ou = $s3 senão $s1=0
Slt $s1, $s2, $s3	Operação direta da micro-instrução
Not $s1	Inverte o conteúdo de $s1

Tabela 6.3.2.2.41 – Pseudo-instrução Sge

Beq $s1, $s2, 5	Se $s1 = $s2 desvia para PC mais a constante
Beq $s1, $s2, 5	Operação direta da micro-instrução

Tabela 6.3.2.2.42 – Pseudo-instrução Beq

Bne $s1, $s2, 5	Se $s1 <> $s2 desvia para PC mais a constante
Beq $s1, $s2, 2	Se $s1 = $s2 pula a próxima linha
J $pc, 5	Desvia para Pc mais a constante

Tabela 6.3.2.2.43 – Pseudo-instrução Bne

Blt $s1, $s2, 5	Se $s1 < $s2 desvia para PC mais a constante
Blt $s1, $s2, 5	Operação direta da micro-instrução

Tabela 6.3.2.2.44 – Pseudo-instrução Blt

Bgt $s1, $s2, 5	Se $s1 > $s2 desvia para PC mais a constante
Blt $s2, $s1, 5	Operação direta da micro-instrução

Tabela 6.3.2.2.45 – Pseudo-instrução Bgt

Slti $s1, $s2, 100	Torna $s1=1 se $s2 < constante senão $s1=0
Addi $s3, $Zero, 100	Armazena o valor da constante em $s3
Slt $s1, $s2, $s3	Operação direta da micro-instrução

Tabela 6.3.2.2.46 – Pseudo-instrução Slti

Seqi $s1, $s2, 100	Torna $s1=1 se $s2 = constante senão $s1=0
Addi $s3, $Zero, 100	Armazena o valor da constante em $s3
Beq $s2, $s3, 2	Pula as próximas duas linhas se $s3 for igual a $s2
J $pc, 2	Vai para o fim
Addi $s1, $Zero, 1	Atribui o valor “um” a $s1

Tabela 6.3.2.2.47 – Pseudo-instrução Seqi

Sgti $s1, $s2, 100	Torna $s1=1 se $s2 > constante senão $s1=0
Addi $s3, $Zero, 100	Armazena o valor da constante em $s3
Slt $s2, $s1, $s3	Operação direta da micro-instrução

Tabela 6.3.2.2.48 – Pseudo-instrução Sgti

Beqi $s1, 100, 5	Se $s1 = constante 1 desvia para PC mais a constante 2
Addi $s2, $Zero, 100	Armazena o valor da constante em $s2
Beq $s1, $s2, 5	Vai para $pc mais a constante se $s2 for igual a $s1

Tabela 6.3.2.2.49 – Pseudo-instrução Beqi

Bnei $s1, 100, 5	Se $s1 <> constante 1 desvia para PC mais a constante 2
Addi $s2, $Zero, 100	Armazena o valor da constante em $s2
Beq $s1, $s2, 2	Pula a próxima linha se $s2 for igual a $s1

J $pc, 5	Desvia para Pc mais a constante

Tabela 6.3.2.2.50 – Pseudo-instrução Bnei

Blti $s1, 100, 5	Se $s1 < constante 1 desvia para PC mais a constante 2
Addi $s2, $Zero, 100	Armazena o valor da constante em $s2
Blt $s2, $s1, 5	Operação direta da micro-instrução

Tabela 6.3.2.2.51 – Pseudo-instrução Blti

Bgti $s1, 100, 5	Se $s1 > constante 1 desvia para PC mais a constante 2
Addi $s2, $Zero, 100	Armazena o valor da constante em $s2
Blt $s1, $s2, 5	Operação direta da micro-instrução

Tabela 6.3.2.2.52 – Pseudo-instrução Bgti

J $s1, 100	Desvia para o endereço $s1 deslocado da constante de oito bits
J $s1, 100	Operação direta da micro-instrução

Tabela 6.3.2.2.53 – Pseudo-instrução J

Jr $s1	Desvia para o endereço $s1
J $s1, 0	Operação direta da micro-instrução

Tabela 6.3.2.2.54 – Pseudo-instrução Jr

Jpc 100	Desvia para o endereço $pc deslocado da constante de oito bits
J $pc, 100	Operação direta da micro-instrução

Tabela 6.3.2.2.55 – Pseudo-instrução Jpc

Jal $s1, 100	Desvia para o endereço $s1 deslocado da constante de oito bits salvando origem em $ra
Jal $s1, 1000	Operação direta da micro-instrução

Tabela 6.3.2.2.56 – Pseudo-instrução Jal

Jalr $s1, $s2, 100	Desvia para o endereço $s2 deslocado da constante de oito bits salvando origem em $s1
Jal $s2, 100	Operação direta da micro-instrução
Add $s1, $ra, $Zero	Copia o valor de $ra para $s1

Tabela 6.3.2.2.57 – Pseudo-instrução Jalr

Jalpc 100	Desvia para o endereço $pc deslocado da constante de oito bits salvando origem em $ra
Jal $pc, 100	Operação direta da micro-instrução

Tabela 6.3.2.2.58 – Pseudo-instrução Jalpc

Jd 100	Desvia para o endereço da constante de 16 bits
Add $s1, $Zero,$Zero	Armazena o valor 0 em $s1
Lui $s1, mais significativa	Carrega $t1 com a parte mais significativa da constante
J $s1, menos significativa	Operação direta da micro-instrução

Tabela 6.3.2.2.59 – Pseudo-instrução Jd

Jald $s1, 100	Desvia para o endereço $s1 deslocado da constante de 16 bits salvando origem em $ra
Add $t1, $Zero,$Zero	Armazena o valor 0 em $t1
Lui $t1, mais significativa	Carrega $t1 com a parte mais significativa da constante
Add $s1, $s1, $t1	Soma o endereço base com $t1
Jal $s1, menos significativa	Operação direta da micro-instrução

Tabela 6.3.2.2.60 – Pseudo-instrução Jald

Jalrd $s1, $s2, 100	Desvia para o endereço $s2 deslocado da constante de 16 bits salvando origem

	em $s1
Add $t1, $Zero,$Zero	Armazena o valor 0 em $t1
Lui $t1, mais significativa	Carrega $t1 com a parte mais significativa da constante
Add $s2, $s2, $t1	Soma o endereço base com $t1
Jal $s2, menos significativa	Operação direta da micro-instrução
Add $s1, $ra, $Zero	Copia o valor de $ra para $s1

Tabela 6.3.2.2.61 – Pseudo-instrução Jalrd

Jalpcd $s1, 100	Desvia para o endereço $pc deslocado da constante de 16 bits salvando origem em $s1
Add $t1, $Zero,$Zero	Armazena o valor 0 em $t1
Lui $t1, mais significativa	Carrega $t1 com a parte mais significativa da constante
Add $t1, $pc, $t1	Soma $pc com $t1
Jal $t1, menos significativa	Operação direta da micro-instrução
Add $s1, $ra, $Zero	Copia o valor de $ra para $s1

Tabela 6.3.2.2.62 – Pseudo-instrução Jalpcd

O tratamento das pseudo-instruções é idêntico ao utilizado nas micro-instruções após a transformação da mesma em um conjunto de micro-instruções. Existem porém, alguns passos que devem ser seguidos para a utilização dos dados digitados pelo usuário, e para isto foram criadas as funções ***op_pseudo_r(), op_pseudo_i()*** e ***op_pseudo_j(),*** ou seja, para cada tipo de pseudo-instrução, lembrando que o tamanho dos campos, em bits, não corresponde ao mesmo utilizado nas micro-instruções, sendo esta apenas uma representação dos tipos de campo presentes na pseudo-instruções. Assim como foi realizado nas micro-instruções, o código fonte será quebrado em módulos para uma melhor aprendizagem, bem como será escolhido um exemplo para demonstrar a conversão de uma pseudo-instrução. Seja como exemplo a pseudo-instrução Subi.

```
//SUBI
/*
Subi   $reg1 ,100
{
    Addi   $t2,100
    Sub   $reg1,$reg1,$t2
}
*/

if (cp[0]=='S'&cp[1]=='U'&cp[2]=='B'&cp[3]=='I')
{
  int l=4, t=0;
```

```
        while (cp[l]==' '|cp[l]=='\t')
          l++;
        cp2 = new char[50];


        while(l!=temp+1)
          {
            cp2[t] = cp[l];
            t++;
            l++;
          }

        pseu=1;
        reg1 = new char [10];
        offset = new char [10];

         op_pseudo_j();
```

C.F.6.3.2.2.a – Identificação e início do tratamento da pseudo-instrução

O primeiro passo para o tratamento da pseudo-instrução é sua identificação, e como para a micro, os caracteres iniciais do vetor ***cp*** (vetor que recebe os caracteres que foram digitados pelo usuário) são utilizados para esta caracterização. Transfere-se então parte da instrução para o vetor ***cp2***. Por exemplo, seja a instrução ***Subi $t0,45*** no vetor ***cp2*** aparecerá **$t0,45**. A variável ***pseu*** recebe o valor **1**, indicando que trata-se de uma pseudo-instrução. Criam-se então dois vetores de caracteres, ***reg1*** e ***offset***, os quais serão utilizados dentro da função ***op_pseudo_j()*** e pelo resto do programa. Será feita uma pausa na análise do código da pseudo-instrução para o estudo do código da função ***op_pseudo_j().*** Vale ressaltar que os nomes escolhidos para as funções que extraem os parâmetros contidos na pseudo-instrução são apenas aproximações do formato das mesmas com o formato das micro-instruções, pois não existem pseudo-instruções do tipo R, I ou J, mas sim um agrupamento de acordo com o tipo de funções, como aritmética, deslocamento, desvio condicional, etc.

```
void TForm2::op_pseudo_j()
{
  char *cp5 = new char[50];
  int l=0;
  while(cp2[l]!='\0')
  {
    cp5[l]=cp2[l];
    l++;
  }
  cp5[l]='\0';
  l=0;
```

```
  while(cp5[l]!=',')
  {
    reg1[l]=cp5[l];
    l++;
  }
  reg1[l]='\0';
  l++;
  int cont=0;
  while(cp5[l]!='\0')
  {
    offset[cont]=cp5[l];
    cont++;
    l++;
  }
  offset[cont]='\0';
  delete cp5;
  return;
}
```

C.F.6.3.2.2.b – Código fonte da função op_pseudo_j()

A função inicia-se copiando o vetor ***cp2*** para ***cp5***, sendo colocado após o término da cópia o caracter indicador de final de vetor de caracteres, **'\0'**. Utilizando-se como parâmetro as vírgulas, faz-se a separação dos dados presentes no vetor ***cp5***, ou seja, com ***cp5 = "$t0,45"*** tem-se que ***reg1*** receberá ***$t0*** e ***offset*** receberá ***45***, valores estes que serão utilizados pelas micro-instruções que seguem-se no algoritmo da pseudo-instrução, para isto, estes vetores (***reg1*** e ***offset***) foram declarados como públicos (podem ser utilizados em qualquer parte do programa, desde que inicializados) no arquivo fonte ***Unit2.h***. A função ***op_pseudo_j()*** termina destruindo o vetor ***cp5*** e retornando para a rotina de tratamento da pseudo-instrução.

6.3.2.2.1 Controle de registradores utilizado pelo algoritmo da pseudo-instrução

Em algumas pseudo-instruções, o algoritmo utiliza registradores para suporte a operações com micro-instruções. Para uma correta execução, os registradores utilizados não podem ser acionados pelo usuário no comando da instrução. Criou-se então uma rotina que checa os registradores digitados pelo usuário na instrução, também o auxílio de vetores e o teste condicional **if**. Continuando o exemplo, seja o código fonte:

```
        int r1 = StrComp (reg1,"$T2");
        //Testa se os registradores não estão sendo usados na pseudo
        if(r1==0)
        {
          MessageBox(Handle,
      "Registrador '$T2' sendo utilizado pelo montador na pseudo-instrução",
```

```
                "Mensagem de erro #04", MB_OK);
   RichEdit1->SelStart = RichEdit1->Perform (EM_LINEINDEX , i, 0 );
   RichEdit1->SelLength = RichEdit1->Lines->Strings [i] . Length ();
   comp=1;
   return (false);
}

l=0;
```

C.F.6.3.2.2.1 – Teste de registrador utilizado na pseudo-instrução

A variável ***r1*** fará a comparação entre o conteúdo do vetor ***reg1*** (registrador digitado pelo usuário) com o registrador ***$t2***, pois o último é utilizado dentro do algoritmo da pseudo-instrução. Se forem iguais, ***r1*** será igual a zero, uma mensagem de erro é mostrada ao usuário e a linha de texto onde encontra-se o erro é destacada. Se forem diferentes, a rotina continua e serão tratadas as micro-instruções que compõem a pseudo-instrução.

6.3.2.2.2 Tratamento da micro-instrução componente de pseudo-instrução

Conforme dito anteriormente, a pseudo-instrução pode ser considerada como uma combinação de micro-instruções, as quais serão tratadas individualmente. A rotina que se segue é utilizada para todas as micro-instruções pertencentes ao bloco da pseudo-instrução em análise.

Seja a continuação do código fonte:

```
//Addi  $t2,offset

rpseudo = new char [30];
l=0;
t=0;
rpseudo[l]='$';
l++;
rpseudo[l]='T';
l++;
rpseudo[l]= '2';
l++;
rpseudo[l]=',';
l++;
t=0;
while(offset[t]!='\0')
{
   rpseudo[l]=offset[t];
   l++;
   t++;
}
```

```
        rpseudo[l]='\0';

        shf=1;
        volta = op_deslocamento();

        if(volta==-1)
          return false;

        short addi = 32768;
        volta=volta|addi;
```

C.F.6.3.2.2.d – Tratamento da micro-instrução interna à pseudo-instrução

Para que a micro-instrução seja construída, utilizou-se o vetor ***rpseudo***, sendo o mesmo formado a partir dos dados obtidos da pseudo-instrução digitada pelo usuário (no caso, utilizou-se o vetor ***offset***), manipulando os caracteres a fim de ser formada a frase que compõem a micro. Após isto, utilizam-se as mesmas funções designadas para tratamento de micro-instruções (no caso, ***op_deslocamento***), sendo após isto acrescentado o campo operando na palavra de 16 bits. As rotinas de gravação em arquivo e exibição do código traduzido serão mostradas nas duas próximas seções.

6.3.3. Geração e gravação no arquivo de saída

Estas próximas seções são idênticas para a micro e pseudo-instruções. Seu algoritmo pode ser resumido no fluxograma apresentado na Figura 6.3.3. Cria-se o arquivo temporário de saída “file_out.bin” e armazena-se a palavra em hexadecimal contendo a codificação da instrução no arquivo de forma seqüencial, ou seja, instrução por instrução. Seu algoritmo está apresentado em C.F.6.3.3.a. O exemplo de aplicação continua sendo a pseudo-instrução Subi e sua micro-instrução Addi.

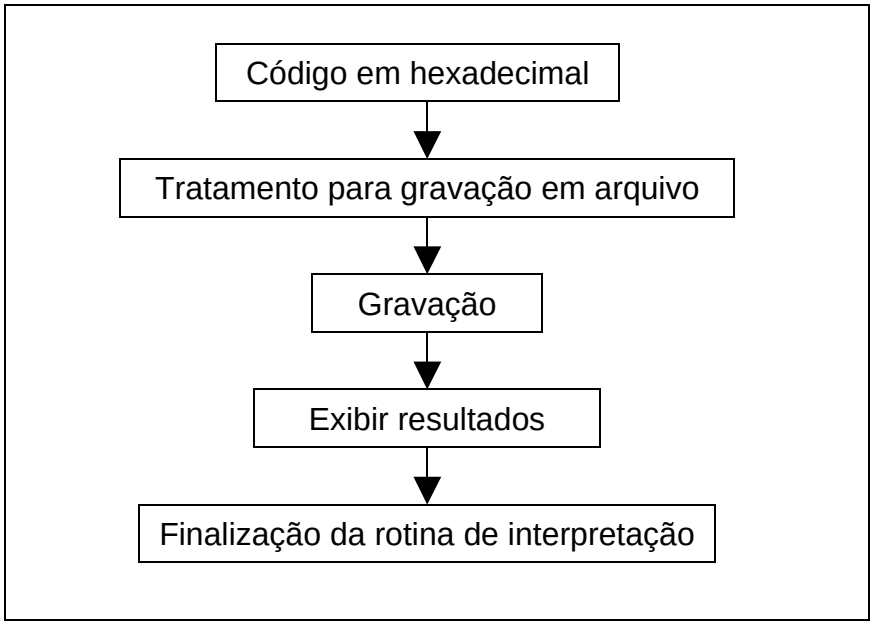


Figura 6.3.3 – Fluxograma de gravação e apresentação de dados

```
volta = op_deslocamento();

if(volta==-1)
   return false;

short addi = 32768;
volta=volta|addi;
unsigned short volta2=volta;
unsigned short volta3=volta;
volta2=volta2<<8;
volta3=volta3>>8;
volta=volta2|volta3;
short data[1],a;
a=0;
```

```
if ((out = fopen("file_out.bin", "ab+"))
 == NULL)
 {
  MessageBox(Handle,"Não foi possível abrir o arquivo para gravação",
  "Mensagem de erro #01", MB_OK);
   return false;
 }

 data[a]=volta;
 fwrite(&data, sizeof(data), 1, out);
 delete rpseudo;
```

C.F.6.3.3.a – Algoritmo de gravação da instrução no arquivo de saída

Colocou-se no código acima apresentado o final da interpretação da micro-instrução Addi, a fim de situar-se o local onde é apresentada a rotina de geração e gravação em arquivo.

A variável volta contém a palavra completa da interpretação do texto referente à micro-instrução. Porém, o compilador C++, presente no software Borland C++ Builder, armazena valores numéricos no formato *big endian*, ou seja, com o byte menos significativo na frente do mais significativo. Só que o processador RISC em estudo utiliza o formato *little endian* onde as posições dos byte são primeiro o mais e depois o menos significativo. Para corrigir este

problema, utilizaram-se as variáveis auxiliares do tipo ***unsigned short*** (inteiros de 16 bits sem sinal, para evitar problemas com os números negativos na operação lógica que será apresentada mais adiante) volta2 e volta3, as quais são deslocadas de 8 bits para a esquerda e oito bits para a direita, representando os bytes menos e mais significativos, respectivamente. Os bytes são unidos através da operação lógica OU e armazenados novamente em volta. Como o montador gerará os números da mesma forma que antes, com a inversão via o algoritmo apresentado, quando os números forem armazenados no arquivo estarão da forma correta a interpretação pelo processador RISC.

Para a gravação em arquivo, criaram-se ainda as variáveis tipo ***short a*** e ***data***, sendo esta última um vetor de uma posição, para que fosse obedecido o procedimento necessário à utilização da função ***fwrite***, conforme indicado pela referência [07].

No início do algoritmo de tradução, em C.F.6.3.2.a, criou-se o stream de saída do tipo ***FILE***, com nome ***out***, indicando ser um arquivo de saída. Este mesmo arquivo é agora utilizado para a criação do arquivo de saída "***file_out.bin***" através do teste condicional utilizando-se o comando ***fopen*** (abrir arquivo) do C++ e com o parâmetro ***ab+*** cujo ***a+*** indica que o arquivo será aberto (ou criado caso não exista) para inserção de dados, e o ***b*** indica que o formato do arquivo é binário.

O vetor ***data*** recebe na posição ***a*** o valor obtido pela compilação, armazenado em volta, e através do comando ***fwrite*** que possui a seguinte forma:

fwrite (endereço do vetor onde está a informação, tamanho do vetor, número de elementos, FILE *stream);

o valor é gravado no arquivo.

```
//finalização da pseudo:SUBI
fclose(out);
delete rpseudo;
delete reg1;
delete offset;
delete cp2;
delete cp;
e=1;
comp=1;
pseu=0;
```

C.F.6.3.3.b – Finalização da pseudo-instrução Subi

Caso este seja a última instrução a ser gravada na pseudo-instrução ou seja o final de uma micro-instrução, após a passagem dos dados também para tela de apresentação de resultados (tela 3), é necessária a utilização do comando ***fclose(out)*** para fechar o arquivo "***file_out.bin***" e finalizar-se o processo de gravação sem que haja erros de alocação de memória e para que os dados

armazenados não sejam perdidos. Destroem-se, então, todos os vetores dinâmicos criados, e as variáveis de controle são configuradas para permitir a tradução de uma nova linha de texto ou a saída do laço de conversão de linhas e finalização da rotina de tradução.

6.3.4. Apresentação dos resultados ao usuário

Antes da finalização da rotina de interpretação, os resultados obtidos, armazenados na variável volta precisam ser rearranjados para que possam ser apresentados. Para isto, utilizou-se o algoritmo descrito em C.F.6.3.4.

```
//Codificação para apresentação na tela...
volta2=volta2>>8;
volta3=volta3<<8;
volta=volta2|volta3;
AnsiString Palavra = IntToHex(volta,2);
Palavra_temp = new char [10];
Palavra_final = new char [10];
StrCopy(Palavra_temp, Palavra.c_str());
int num=4;
t=0;
while (Palavra_temp[num]!='\0')
{
  Palavra_final[t]=Palavra_temp[num];
  num++;
  t++;
}
Palavra_final[t]='\0';
Form3->RichEdit1->Lines->Add(" ");
Form3->RichEdit1->Lines->Add("Pseudo-Instrução: Subi");
Form3->RichEdit1->Lines->Add("{");
Form3->RichEdit1->Lines->Add(Palavra_final);
delete Palavra_temp;
delete Palavra_final;
```

C.F.6.3.4 – Codificação para apresentação dos resultados

O código apresentado está no final da micro-instrução ***Addi***, dentro da pseudo ***Subi***, sendo esta micro a primeira a ser tratada pela rotina de tradução. Caso fosse a última micro pertencente a uma pseudo-instrução, as declarações de tipo de variáveis (***int, AnsiString***) seriam omitidas, pois sua presença causaria erro de compilação no C++ Builder. Para as micro-instruções do processador, não foram colocados os caracteres identificadores de instrução na tela, como por exemplo o texto “Pseudo-Instrução: Subi”e as chaves, mas somente o valor em hexadecimal.

Para uma correta apresentação no componente RichEdit1 contido no Form3 deve-se, pelo motivo inverso apresentado quando da gravação de dados em arquivo, utilizando-se variáveis auxiliares, inverter-se as posições dos bytes dentro da palavra contida em volta. Para a conversão de número para texto, utilizou-se a função IntToHex, sendo que o texto possuirá o valor numérico expresso em hexadecimal. O resultado é armazenado na variável Palavra, do tipo AnsiString, a qual é copiada para os vetores de caracter Palavra_temp e desta última para Palavra_final a fim de que sejam retirados os bytes que não são necessários na apresentação, pois a função de conversão número -> texto utiliza como entrada variável do tipo inteiro, 32 bits, sobrando portanto os 2 bytes mais significativos que não devem ser apresentados pois as instruções do processador possuem apenas 16 bits. Toda a manipulação realizada para apresentação do resultado na tela não influencia o valor gravado em arquivo. Como exemplo, sejam digitados pelo usuário as seguintes instruções:

```
Subi   $t0,45
Beq    $s0,$s1,3
```

O resultado após a tradução é apresentado de acordo com a Figura 6.3.4. Vale ressaltar a presença de cabeçalho e chaves na pseudo-instrução e a apresentação somente do código para a micro-instrução, conforme foi dito anteriormente.

Os caracteres adicionados não são gravados no arquivo que contém o código objeto, mas somente existem para facilitar o entendimento do código convertido por parte do usuário.

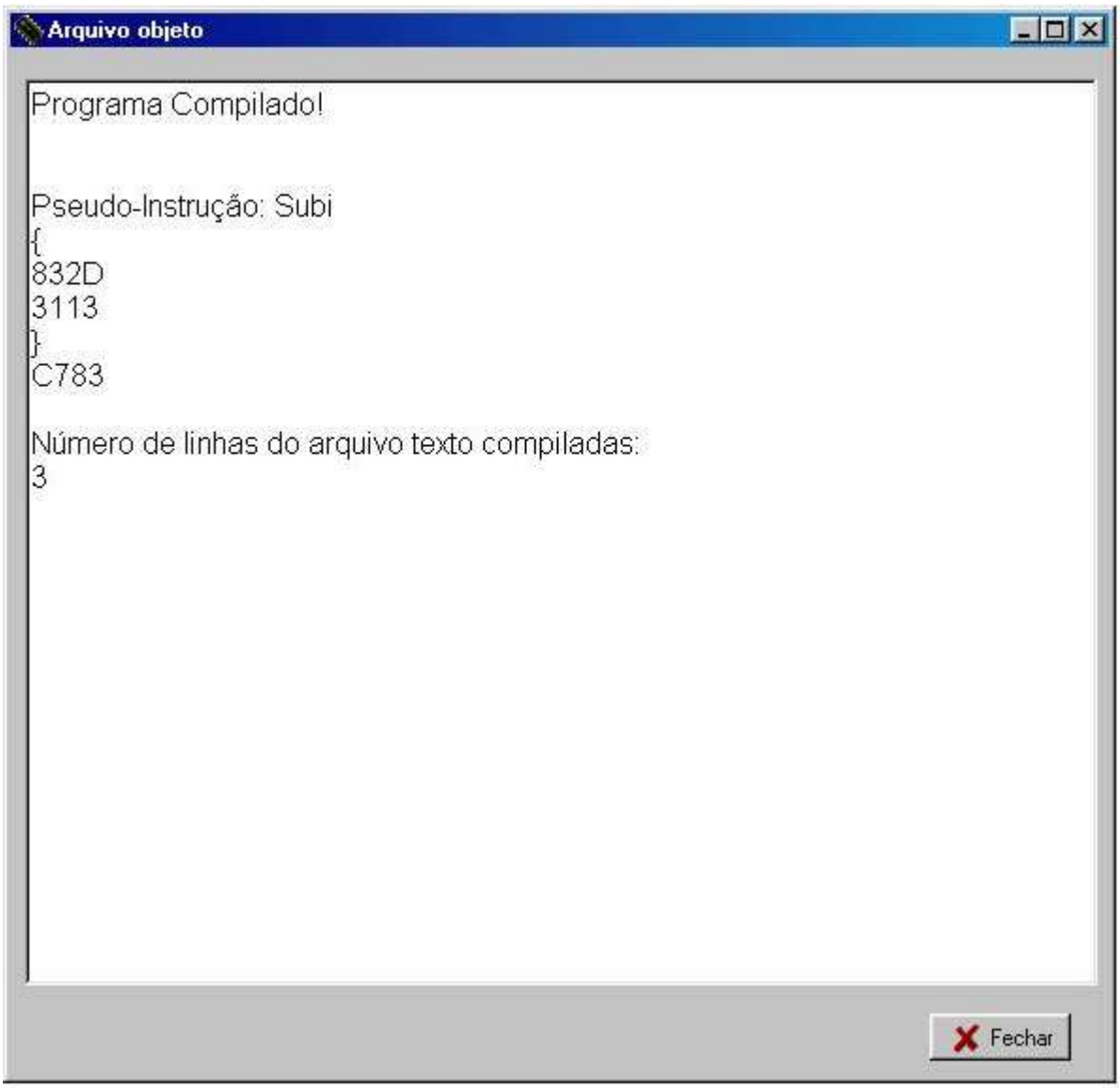


Figura 6.3.4 – Apresentação dos resultados da rotina de conversão

Figura 6.3.b – Botão Converter

A figura 6.3.b. apresenta o botão correspondente à função Compila() no programa Montador – v.1.1, possui ainda a tecla de atalho como sendo F9 e está presente como iten de menu.

6.4. Ajuda e informações ao usuário

Esta parte do programa fornece auxílio ao usuário para solução de problemas e informações sobre o programa.

6.4.1. Arquivo de ajuda ao usuário

Figura 6.4.1 – Botão Ajuda

Possuirá como tecla de atalho Ctrl+F1. O arquivo de ajuda será projetado e construído para a próxima versão do “software” Montador.

6.4.2. Sobre o programa

Figura 6.4.2.a – Botão About

Sua tecla de atalho é Ctrl+J Para a construção desta propriedade do programa, utilizou-se o modelo de aplicativo “About” do software Borland C++ Builder 5.0, apresentado após a conversão do código na Figura 6.4.2.b.

Figura 6.4.2.b – Janela About

Esta tela possui o seguinte algoritmo para sua chamada:

```
void __fastcall TForm2::SpeedButton14Click(TObject *Sender)
{
  AboutBox->ShowModal();
}
```

C.F.6.4.2 – Código para a chamada da janela About

A forma de apresentação “ShowModal” implica que o usuário só pode voltar a janela principal após fechar a janela em destaque.

6.5. Rotinas de teste do software Montador – v.1.1

As rotinas de teste foram desenvolvidas para testar os algoritmos que compõem o software. O conjunto de testes foi desenvolvido pela aluna Juliana Zago Diniz na referência [10], Um comentário a respeito destes testes está no capítulo 3 do Anexo. Contudo, para a implementação do software, foram desenvolvidos testes para verificar o funcionamento das instruções, neste trabalho, os quais estão apresentados no Anexo, páginas 4 a 9. Estes últimos são testes simples, não conclusivos, mas somente para testar o funcionamento da codificação do montador para micro e pseudo-instruções. As mensagens de erro existentes no programa montador estão mostradas no Anexo, páginas 11 e 12.

Alguns erros foram encontrados durante a rotina de teste. Os mais significativos foram a restrtição à entrada de dados das micro-instruções tipo I e tipo J, pois apresentam entrada de registrador e constante e caso fosse digitado um resgistrador no campo destinado a constante, a tradução prosseguia retornando um resultado falso. Este problema foi devidamente corrigido com o aumento das propriedades de filtragem de tipos de entrada dentro das funções utilizadas para o tratamento destes tipos de instrução.

Outro problema relevante encontrado dentro da rotina de conversão, quando muitas pseudo-instruções são inseridas pelo usuário, foi a tentativa de apagar-se o lugar da memória utilizado por um vetor dinâmico que já havia sido apagado pelo programa, o que não é possível pois o vetor que apontava para esta posição de memória não mais existe. O algoritmo foi examinado durante a execução do programa e o problema foi devidamente corrigido

6.6. Configurações mínimas recomendadas

Para uma correta operação e visualização do programa Montador – v1.1 sugere-se a utilização da seguinte configuração mínima para um bom desempenho:

- Processador Intel PENTIUM® II 266Mhz ou superior;
- 64 MB de memória RAM;
- Monitor de vídeo em cores com resolução de 1024x768;
- Sistema operacional MS Windows 98® ou superior;
- Teclado e mouse.

7. Conclusão

As metas estabelecidas para este projeto final de graduação em engenharia elétrica foram plenamente alcançadas. Desenvolveu-se um software montador que possui um relativamente alto grau de segurança quanto a erros de sintaxe da

linguagem, com todas as facilidades dos editores de texto condicionais. Justifica-se, portanto, a construção deste software de sistema pois considerando-se a divisão de níveis de linguagem, pode-se verificar a necessidade da utilização de uma linguagem de mais alto nível, no caso o assembly, para a programação de um processador (linguagem de máquina), a fim de obter-se uma maior confiabilidade no código a ser colocado dentro da memória do processador.

As funções criadas para que o programa cumprisse a especificação proposta podem ser melhoradas em seus algoritmos de construção, a fim de que o tempo de tradução do texto digitado pelo usuário diminua.

Para versões futuras do programa, muito do código pode ser encapsulado em funções, o que geraria uma diminuição do número de linhas programadas. A desvantagem deste encapsulamento, é que caso se deseje fazer alguma alteração em uma instrução o efeito poderá ser sentido em todas as instruções que utilizam este código, já com a forma que o programa atualmente é composto, as alterações são locais, o que facilita o teste e busca de erros. Poderá ser acrescentada uma rotina que busque dentro do código digitado a ocorrência de registradores que são internos ao algoritmo de alguma pseudo-instrução, informando o fato ao usuário pois dados contidos nestes registradores serão sobre-escritos quando a pseudo-instrução for executada, permitindo o cancelamento da conversão ou prosseguimento da mesma.

Pode-se também expandir o montador para o modelo de dois passos, a fim de que o mesmo reconheça *labels* e comandos de atribuição como *equ* o qual pode ser utilizado para atribuições de constantes.

Já está sendo testado um simulador VHDL para o processador, o qual poderá indicar com sua utilização que serão necessárias alterações no programa montador ou até mesmo na especificação de micro e pseudo-instruções. Pode-se também buscar a criação de um compilador, o qual talvez possua um simulador onde o programa será carregado e executado, a fim de se obter a resposta que deveria ser conseguida com a utilização do processador RISC.

Houve neste trabalho um bom desenvolvimento do conhecimento da linguagem de programação C++, bem como do entendimento do funcionamento dos processadores e softwares de sistema.

8. Bibliografia

[01] Patterson, David A.; Henessy, John L.. *Organização e projeto de computadores – A interface hardware/software.* 2° Edição. LTC Editora. Brasil, Rio de Janeiro, RJ, 2000.

[02] Taub, Herbert. *Circuitos digitais e microprocessadores.* McGraw-Hill Editora. Brasil – São Paulo, SP, 1984.

[03] Zelenovsky, Ricardo; Mendonça, Alexandre*. PC: Um guia prático de hardware e interfaceamento.* 3° Edição. MZ Editora LTDA. Brasil – Rio de Janeir o, 2002.

[04] Beck, Leland L.. *System Software: An Introduction to Systems Programming.* 3° Edição. Addison-Wesley Ed. , USA – San Diego.

[05] John Miano, Thomas Cabanski, Harold Howe . *C++ Builder How-To.* Waite Group Press. USA – Corte Madera, CA, 1997.

[06] Spanghero, A.. *Aprendendo C++ Builder 3 – Guia Prático.* Makron Books do Brasil Editora Ltda, Brasil – São Paulo, SP, 1999.

[07] *C++ Builder Help Files*

[08] Linder, R. R.. *Linguagem de Máquina para Processador num Sistema em CHIP (SOC).* Dissertação de Mestrado, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF.

[09] Benício, G. M.. *Projeto de Microprocessador RISC 16-Bit para Sistema de Comunicação sem Fio em Chip.* Dissertação de Mestrado, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF.

[10] Diniz, Juliana Z.. *Desenvolvimento de Aplicação Básica para Sistema de Comunicação em Chip (SOC) sem fio*. Projeto final de graduação em Engenharia Elétrica, Departamento de Engenharia Elétrica, Universidade de Brasília,DF.