EA870 - Laboratório de Computação

Laboratório 3
Montador
Representação em Ponto Flutuante
Padrão IEEE 754

1. Introdução

No laboratório 1 foi exercitada a codificação manual das instruções a partir da representação mnemônica. Esta codificação manual, se utilizada para a inserção de programas, requer intensa concentração, sendo extremamente ineficiente e susceptível a erros. Já no laboratório 2 foram introduzidas as facilidade de codificação e decodificação das instruções oferecidas pelo firmware TUTOR. Através das opção "; DI" dos comandos do TUTOR é possível inserir e inspecionar instruções no formato mnemônico. Comparada com a codificação manual, a opção "; DI" significa um avanço, já que agora não a cansativa tarefa de codificação/decodificação é realizada automaticamente pelo TUTOR. Entretanto, as referências a posições de memória, utilizadas principalmente nas instruções de desvio, exigem ainda a atenção do programador. Outra deficiência da solução "; DI" está associada à modificação de um programa já residente em memória - a inserção de uma nova instrução, por exemplo, exige o cuidadoso deslocamento do código já inserido. Para suprir estas deficiências e oferecer um ambiente mais amigável para o desenvolvimento de programas, é introduzido neste laboratório um programa montador para o MC68000.

A representação e manipulação de números reais é outro tema tratado neste roteiro.

Este roteiro está organizado com se segue.

- Seção 2: Montador – nesta seção discute-se os conceitos associados ao processo de montagem;
- Seção 3: Montador 68000 – nesta seção é apresentado o programa montador que será utilizado no decorrer da disciplina;
- Seção 4: Representação em Ponto Flutuante – nesta seção discute-se os conceitos básicos associados à representação em ponto flutuante;
- Seção 5: Padrão IEEE 754 – nesta seção é apresentado e discutido o padrão IEEE 754 para a representação em ponto flutuante;
- Seção 6: Exercícios

2. Montador (*Assembler*)

Um **Montador** é um programa que converte um programa escrito em linguagem de montador (também conhecida como linguagem de Montagem, ou linguagem Simbólica, ou ainda linguagem *Assembly*) em linguagem de máquina (fig. 3.1). Genericamente utiliza-se as denominações Código Fonte e Código Objeto para designar a entrada e a saída do processo de montagem. Assim, o Código Fonte de um programa Montador é um programa escrito em linguagem de montador; o Código Objeto é o resultado da tradução deste programa fonte para linguagem de máquina. O Código Objeto gerado por um Montador é composto pela seqüência de instruções especificada no código fonte codificadas em formato binário executável. Um programa em linguagem de máquina pode ser carregado na memória de um computador e executado.

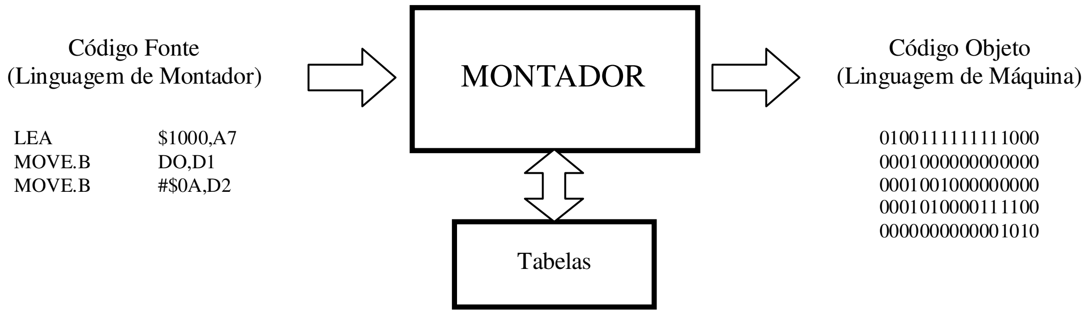

Fig. 3.1: Esquema de funcionamento de um Montador típico

Em geral, um programa em linguagem de montador consiste de uma seqüência de linhas de código. Cada uma destas linhas é, em geral, formada por conjunto de caracteres ASCII, sendo terminada pelos caracteres line feed ($0A) e/ou carriage return ($0D). Tipicamente, as linhas do código fonte de um montador possuem o formato:

```
     RÓTULO: OPERAÇÃO      OPERANDO,OPERANDO;COMENTÁRIO
```

Por exemplo:

```
     UM:     EQU           01               ; define constante
= 1

     LOOP:   MOVEL.B       D0,(A0)+     ; armazena valor em
memória
```

O campo RÓTULO é um identificador simbólico (um nome) utilizado para identificar valores constantes ou endereços de memória.

O campo OPERAÇÃO especifica uma instrução de máquina ou diretiva de montador. As operações que compõem um programa em linguagem de montador podem ser classificadas em dois grupos: 1) instruções de máquina; e 2) pseudo-instruções. As instruções de máquina são instruções, no formato mnemônico, definidas no jogo de instruções do processador. O montador converte cada uma destas instruções no correspondente código executável. Já as pseudo-instruções, ou diretivas do montador, são comando que direcionam a atividade do montador. As pseudo-instruções não são instruções passíveis de serem executadas pelo processador - elas servem apenas para direcionar o processo de montagem. Exemplos típicos de pseudo-instruções são apresentadas na tabela 3.1.

Pseudo-instrução	Exemplo	Comentários
ORG	ORG $1000	(ORiGin) Define a origem. O código (dado ou instrução) que se segue será alocado na posição de memória especificada (no exemplo, posição $1000).
EQU	UM: EQU 1	(EQUate) Define identificador simbólico (rótulo), atribuindo-lhe um valor constante (no exemplo, o símbolo UM foi definido como tendo o valor constante 1).
DC.B, DC.W, DC.L	DC.B 00,01,02	(Define Constant) Define constante. Define e armazena valores constantes em memória (no

		exemplo foram definidos, e armazenados em seqüência, os bytes 00, 01, e 02).
DS.B, DS.W, DS.L	DS.W 04	(Define Storage) Reserva posições de memória (no exemplo foram reservadas 4 palavras consecutivas em memória)
END	END	Sinaliza o final do código fonte. Tudo o que vier após esta pseudo-instrução é ignorado pelo montador.

Tabela 3.1: Exemplos de Pseudo-instruções

Os campos OPERANDO definem os operandos da operação especificada separados por vírgulas.

O campo COMENTÁRIO é composto pelo conteúdo do resto da linha de código após o último operando, sendo utilizado pelo programador para documentar o código fonte. O campo COMENTÁRIO não é interpretado pelo Montador, sendo apenas copiado para o arquivo de listagem (*listing file*), se este for gerado.

2.1 O processo de montagem

Para traduzir o programa fonte em linguagem simbólica para o correspondente programa em linguagem de máquina, o programa montador deve:

- determinar os valores dos identificadores simbólicos (rótulos);
- substituir todas as referências a um dado identificador simbólico pelo valor binário correspondente;
- substituir os mnemônicos da instrução e especificadores de modo de endereçamento dos operando pelo código binário equivalente;

Para determinar os valores dos identificadores simbólicos (rótulos), o Montador processa o Código Fonte sequencialmente, a partir do início, interpretando as pseudo-instruções e avaliando o comprimento das instruções de máquina. Observar que um rótulo pode ser utilizado para representar um endereço de memória ou um valor constante. Para a determinação do valor de um rótulo que representa uma posição de memória, o Montador mantém um registro da ocupação da memória a partir de um endereço inicial. Este registro é, em geral, implementado na forma de uma variável denominada Contador de Posição – CP. A variável CP contém o endereço atual de memória. Assim que uma instrução é processada, o CP é atualizado (o comprimento da instrução é somado ao valor do CP) para apontar para a próxima posição de memória. No instante que um rótulo que representa um endereço é detectado no programa fonte, o Montador atribui a este rótulo o endereço da posição de memória corrente. O valores de um rótulos que representa uma constante é determinado pela interpretação da pseudo-instrução que define o valor constante.

Exemplo de tradução do montador a ser utilizado neste curso.

Endereço	**Código Objeto**	**Código Fonte**		
00000000	-	ZERO:	EQU	00
00000000	-	CONT:	EQU	10
00000000	-	TUTOR:	EQU	228

00001000	-	INICIO:	ORG	$1000
00001000	4FF8 1000		LEA	INICIO,A7
00001004	41F9 0000101E		LEA	AREA,A0
0000100A	103C 0000		MOVE.B	#ZERO,D0
0000100E	323C 0009		MOVE.W	#(CONT-1),D1
00001012	10C0	DENOVO:	MOVE.B	D0,(A0)+
00001014	51C9 FFFC		DBF	D1,DENOVO
00001018	1E38 00E4		MOVE.B	TUTOR,D7
0000101C	4E4E		TRAP#14	
0000101E	-	AREA:	DS.B	CONT
00001028	-		END	

Tabela 3.2: Exemplo de tradução.

2.1.1 Estruturas de dados do processo de montagem

Para realizar o processo de montagem, o montador utiliza duas tabelas estáticas (fixas), que relacionam os mnemônicos utilizados na linguagem de montagem com os códigos binários correspondente. Estas tabelas são:

- Tabela de Operações de Máquina - TOM;
- Tabela de Pseudo-Operações – TPO.

Durante o processo de montagem, o Montador cria e utiliza ainda uma tabela dinâmica que relaciona os identificadores simbólicos (rótulos) com os valores binários correspondentes. Esta tabela é denominada :

- Tabela de Símbolos - TS.

Observar que as tabelas TOM e TPO fazem parte da implementação do programa Montador. São estas tabela que definem o jogo de instruções e pseudo-instruções suportado pelo montador. Já a tabela de símbolos é uma tabela temporária construída durante o processo de montagem, que só faz sentido para o programa fonte que estiver sendo montado em um determinado momento.

2.1.1.1 Tabela de Operações de Máquina - TOM

A tabela TOM associa o mnemônico de uma instrução com a correspondente instrução de máquina codificada no formato binário. A tabela TOM pode possuir vários campos com informações necessárias ao processo de montagem. O número de campos que compõem a tabela TOM depende de características específicas do processador e da sintaxe da linguagem de montador utilizada. Entretanto, dois campos estão sempre presentes na tabela TOM:

- Campo MNEMÔNICO - onde estão os mnemônicos das instruções, definidos pela sintaxe da linguagem simbólica.
- Campo CÓDIGO-DE-OPERAÇÃO - onde estão os códigos de operação das instruções de máquina.

Outros campos podem completar a informação :

- Campo COMPRIMENTO - contém o comprimento da instrução.

- Campo ROTINA-DE-TRATAMENTO-DE-OPERANDOS - onde estão relacionadas as rotinas que o montador deve ativar para calcular o tamanho dos operandos, codificar os modos de endereçamento dos operandos.

Um exemplo de trecho de tabela TOM é apresentado na tabela 3.3.

Mnemônico	**Código de Operação (binário)**	**Comprimento (bytes)**	**Rotina de Tratamento de Operandos**
…	…	…	…
MOVE.W	0011????????????	Variável	RMOVEW
RTS	0100111001110101	2	RRTS
…	…	…	…

Tabela 3.3: Exemplo de tabela TOM.

O comprimento variável indica que o tamanho da instrução é dependente de seus operando, isto é, é dependente do modo de endereçamento utilizado para especificar os seus operandos. Para uma dada instruções de tamanho variável, o cálculo de seu comprimento envolve a chamada à rotina de tratamento de operando.

2.1.1.2 Tabela de Pseudo-Operações - TPO

A tabela de Pseudo-operações relaciona cada mnemônico de pseudo-instrução com a rotina que o montador deve ativar para processar a pseudo-instrução. Da mesma forma que a tabela TOM, a tabela TPO pode possuir vários campos com informações relevantes para o processo de montagem. Dois campos, entretanto, estão sempre presentes na tabela TPO :

- MNEMÔNICO - onde estão os mnemônicos das pseudo-instruções, definidos pela sintaxe da linguagem simbólica.

- ROTINA-DE–TRATAMENTO-DE-OPERANDOS - onde estão relacionadas as rotinas que o montador deve ativar para processar a pseudo-instrução.

Outros campos podem completar a informação como por exemplo :

- COMPRIMENTO - contém o número de bytes ou palavras que deve ser reservado para pseudo-instruções como em DC.B (Define Constant Byte) ou DC.W (Define Word).

Um exemplo de trecho de tabela TPO é apresentado na tabela 3.4.

Mnemônico	**Comprimento (bytes)**	**Rotina deTratamento de Operandos**
…	…	…
ORG	-	RORG
EQU	-	REQU
DC.B	Variável	RDCB
…	…	…

Tabela 3.4: Exemplo de tabela TPO.

De maneira similar ao tratamento das instruções de máquina de tamanho variável, para o cálculo do tamanho de uma dada pseudo-instrução de tamanho variável envolve a chamada à sub-rotina de tratamento de operandos.

2.1.1.3 Tabela de Símbolos - TS

A Tabela de Símbolos é construída durante a montagem e atribui a cada símbolo (rótulo) definido no código fonte um valor numérico em representação binária. O valor numérico associado ao símbolo pode representar um endereço de memória ou um valor constante.

A tabela 3.5 apresenta a tabela de símbolos do exemplo apresentado na tabela 3.2

Símbolo	Valor (hexadecimal)
AREA	0000101E
CONT	0000000A
DENOVO	00001012
INICIO	00001000
TUTOR	000000E4
ZERO	00000000

Tabela 3.5: Exemplo de tabela TS.

É interessante observar que a tabela de símbolos deve ser ordenada para agilizar o processo de busca de um símbolo.

2.1.1.4 Dinâmica do processo de montagem

Para realizar a tradução, os montadores geralmente processam o código fonte em 2 passos. No primeiro passo, o montador lê o código fonte e constrói a tabela de símbolos, determinando os valores numéricos (endereços ou valores constantes) dos identificadores simbólicos (rótulos) utilizados. No segundo passo, o código fonte é lido novamente e os mnemônicos das instruções e as referências simbólicas são substituídos pelos respectivos códigos de máquina.

2.1.1.4.1 Passo 1 – Gerando a Tabela de Símbolos

O primeiro passo do processo de montagem está associado à construção da Tabela de Símbolos. Para a geração da Tabela de Símbolos os seguintes processamentos se fazem necessários:

- processar as pseudo-instruções, utilizando a Tabela de Pseudo-Operações - TPO;
- determinar o comprimento das instruções de máquina, utilizando Tabela de Operações de Máquina - TOM;
- atualizar o contador de posição CP, de acordo com o comprimento das instrução e pseudo-instrução;
- armazenar os símbolos (rótulos) e seus respectivos valores na Tabela de Símbolos.

A figura 3.2 apresenta o fluxograma simplificado do Passo 1 de um Montador. Neste fluxograma apenas as pseudo-instruções ORG, DC.B, DS.B, EQU e END são tratadas. Duas variáveis têm papel de destaque neste fluxograma:

- CP (contador de posição) – variável que aponta para a próxima a posição de memória onde a próximo código (instrução ou dado) deve ser inserido;
- T (tamanho) – variável que recebe o tamanho da instrução ou pseudo-instrução que está sendo processada.

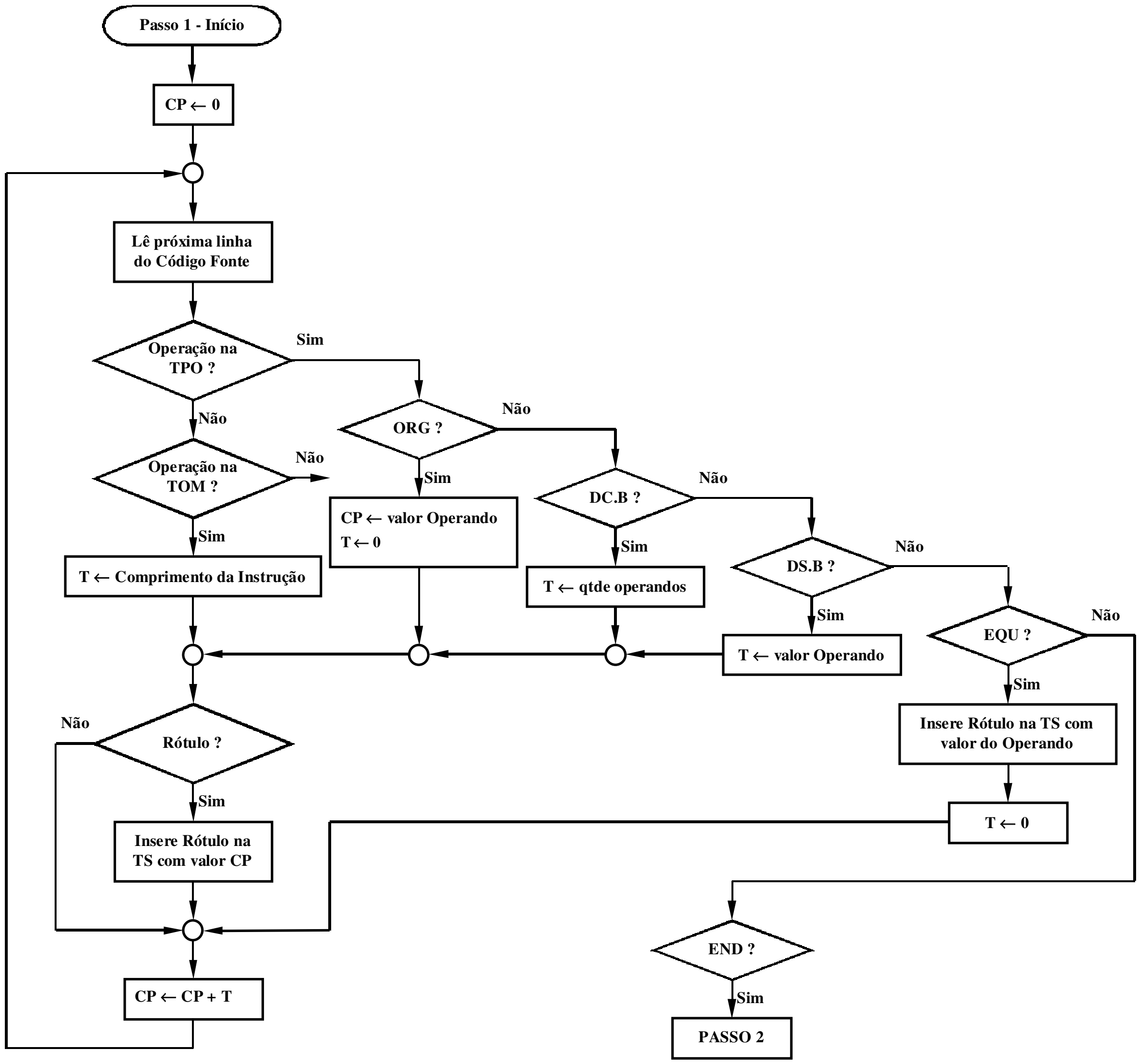

Fig. 3.2: Processo de montagem – Passo 1.

Inicialmente, o contador de posições CP é feito igual a zero e a primeira linha do programa fonte é lida. É feita, então, uma busca na tabela TPO para determinar se o campo OPERAÇÃO é uma pseudo-instrução. Em caso afirmativo, o montador processa a pseudo-instrução; caso contrário, fará uma busca na tabela TOM para verificar se a operação é uma instrução de máquina. Caso a instrução não conste da tabela TOM, é ativado um tratamento de erro que não é apresentado no fluxograma.

Quando a instrução é encontrada na tabela TOM, seu comprimento é obtido e armazenado na variável T. O rótulo, se existir, é armazenado na tabela TS. O contador de posições CP é então atualizado com o comprimento da instrução, passando a apontar para o próximo endereço a ser processado com a leitura da próxima instrução do programa fonte.

No caso da busca na tabela TPO indicar que a instrução é uma pseudo-instrução, o processamento do montador irá depender do tipo da pseudo-instrução.

As pseudo-instruções DC.B e DS.B, que ocupam/reservam posições de memória, são processadas para a determinação do tamanho do espaço de memória alocado, sendo este tamanho armazenado na variável L. O rótulo, se existir, é armazenado na TS. O contador CP é então atualizado, passando a apontar para o próximo endereço a ser processado. A pseudo-instrução EQU não aloca espaço de memória e seu processamento se resume em armazenar o rótulo na tabela TS. Se a instrução for a pseudo-instrução ORG, o CP será carregado com o valor de seu operando, e o programa será montado a partir da posição especificada. Já a pseudo-instrução END, ao ser detectada, faz com que o montador inicie o processamento associado ao Passo 2.

2.1.1.4.2 Passo 2 – Gerando o Código Objeto

O segundo passo do processo de montagem está associado à geração do código objeto. Para a geração do código objeto os seguintes processamentos se fazem necessários:

- substituir os símbolos por seus valores, utilizando a Tabela de Símbolos;
- gerar o código binário associado às instruções de máquina;
- gerar o código binário associado a valores constantes definidos através de pseudo-instruções.

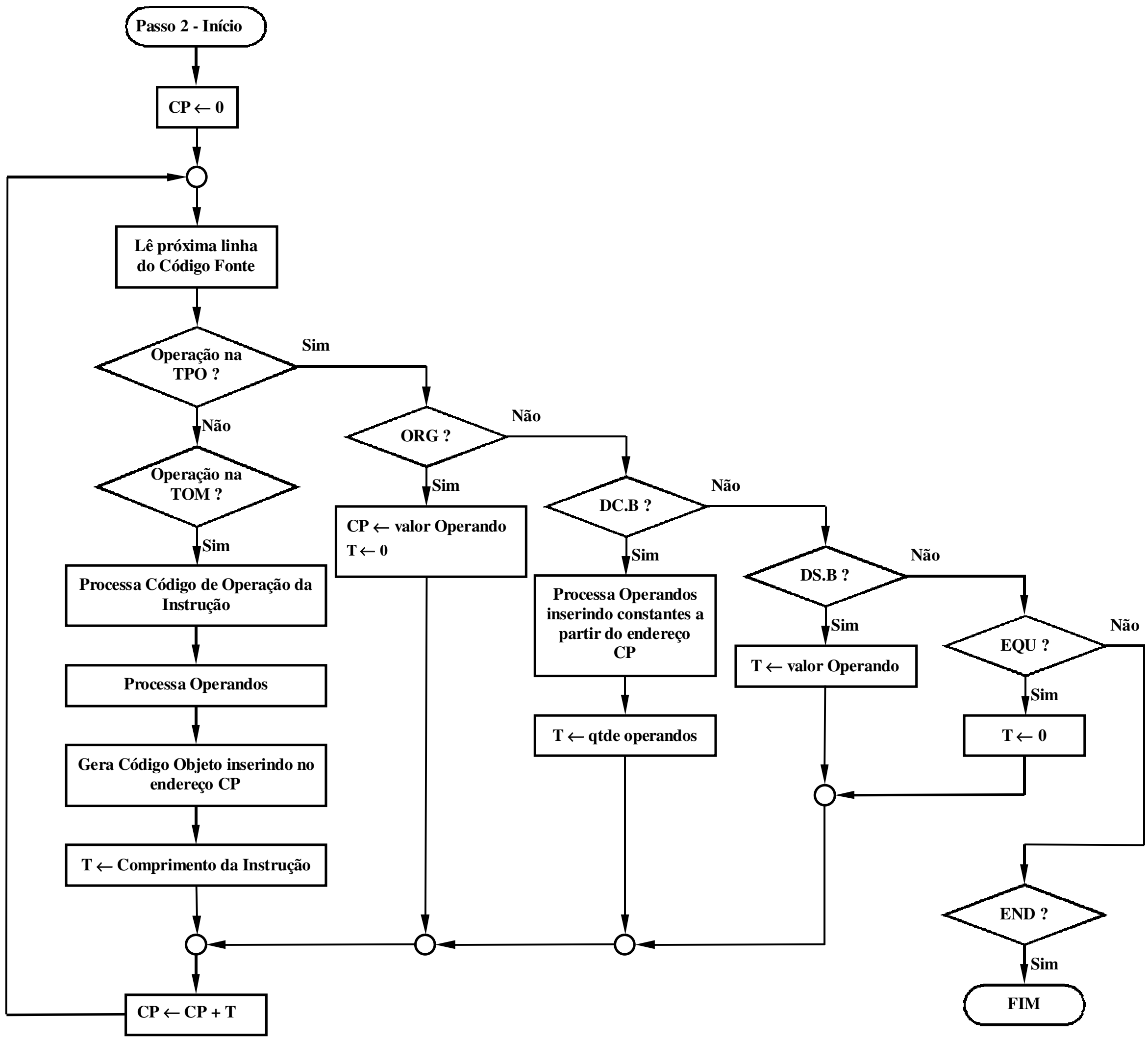

Fig 3.3: Processo de montagem – Passo 2.

A figura 3.3 apresenta um fluxograma simplificado do Passo 2. Assim como no Passo 1 discutido anteriormente, considera-se apenas as pseudos ORG, DC.B, DS.B, EQU e END.

No início do Passo 2 o contador de posições CP é zerado e o programa fonte é lido novamente. A cada linha lida é feita a busca nas tabelas TPO e TOM. No Passo 2 não é necessário prover um tratamento de erro quando a instrução não pertencer às tabelas TOM e TPO; o tratamento de erro é ativado no Passo 1, implicando que o Passo 2 não será executado.

Se a operação é uma instrução de máquina, a operação e seus operandos são traduzidos para linguagem de máquina utilizando as tabelas TOM (código de operação e tratamento de operandos) e TS (para os símbolos). O comprimento da instrução é armazenado no variável T e o contador de posições CP é atualizado para apontar para a próxima posição de memória onde o próximo código de máquina será inserido.

Se a operação lida é uma pseudo-instrução de definição de constante (DC.B), a(s) constante(s) é(são) geradas(s) e inserida(s) no código objeto. O tamanho da(s) constante(s) gerada(s) é(são) utilizado(s) para atualizar o CP. Se a pseudo-instrução for DS.B, o Montador simplesmente atualizará o CP, reservando espaço sem gerar código

O processamento da pseudo-instrução ORG é igual ao do Passo 1, acarretando apenas o carregamento do CP com o endereço especificado na pseudo-instrução. A pseudo-instrução EQU não é processada no Passo 2; e a pseudo-instrução END finaliza o processo de montagem.

3. Programa Montador 68000

A partir deste laboratório utilizaremos o Montador 68000 (*68000 Assembler*) para o desenvolvimento de programas. O Manual do Usuário do Montador 68000 acompanha este roteiro em anexo.

O Montador 68000 encontra-se instalado nos computadores tipo PC. Por ser executado em um tipo de máquina (no caso PC – processador da família Intel) e gerar código para outro tipo de máquina (processador MC68000), este montador recebe a denominação de Montador Cruzado (Cross-Assembler).

Para a geração de um programa executável utilizando o Montador 68000 seguir os seguinte passos:

1. Gerar arquivo no ambiente PC contendo o programa fonte. Para tanto, utilizar o editor de texto ne (Norton Editor) e seguir a sintaxe descrita no manual do montador. Para executar o editor ne, digitar no prompt do DOS

   ne <nome_do_arquivo_fonte>

   Para padronizar e facilitar a identificação dos arquivos, é fortemente recomendado que os arquivo fonte tenham a extensão **.ASM**.

2. Executar o programa Montador, digitando no prompt do DOS a seguinte linha de comando:

   asm –l <nome_do_arquivo_fonte>

Caso o programa fonte contenha algum erro sintático, o montador abortará o processo de montagem, indicando a ocorrência de erro. Em caso de erro, volte ao passo 1 para uma nova edição. Caso o programa fonte não contenha erros, o montador gerará dois novos arquivos, com as seguintes extensões: **.LIS** e **.H68**. O arquivo **.LIS** é um arquivo de listagem que contem informações sobre o código gerado. O arquivo **.H68** é o código objeto no formato S-Record.

3. Executar o programa Download para carregar o código objeto no SBC68K. Para tanto, digite, no prompt do DOS, o seguinte comando:

   download <nome_arquivo_objeto>

4. Representação em Ponto Flutuante

A representação em ponto flutuante permite que números com parte fracionária sejam representados e manipulados por um computador digital. Nesta representação um número é representado na seguinte forma:

$$(-1)^S \times M \times B^E$$

Esta forma implica em uma representação binária composta de 3 campos:

- Campo do Sinal S: (0 ou 1).
- Campo da Mantissa: M
- Campo do Expoente: E

A base B deste sistema de numeração é, em geral, igual a 2.

A figura 3.4 apresenta o formato de um número em ponto flutuante de 32 bits usualmente utilizado. O bit mais significativo (bit 31) armazena o sinal (0 = positivo, 1 = negativo). O expoente de 8 *bits* é armazenado nos bits 30 a 23. Nos 23 bits restantes da representação (bits 22 até 0) é armazenada a mantissa.

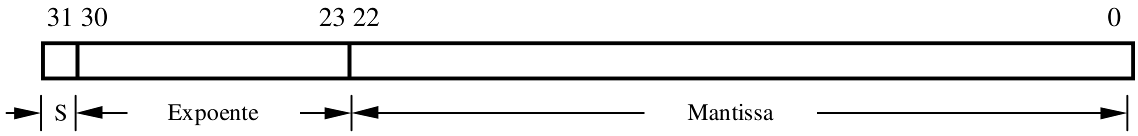

Fig. 3.4:Formato típico para a representação em ponto flutuante com 32 bits.

É importante observar que a representação em ponto flutuante permite que um mesmo valor possa ser representado de várias maneira. Considere, por exemplo, os números (0,1), (0,01 x 10) , e (1,0 x 10^{-1}), observe que todos eles representam o mesmo valor. Para representar um valor de uma única maneira e simplificar as operações envolvendo estes valores, faz-se necessário normalizar a representação em ponto flutuante. Uma forma de se normalizar a representação em ponto flutuante, baseada no mesmo princípio que norteia a representação em notação científica (manter apenas um dígito antes da vírgula), consiste em sempre expressar a mantissa na forma:

$$1,ddddd...$$

Com "d" representando um digito binário (0 ou 1). Nesta representação normalizada, o primeiro digito antes da vírgula é sempre igual a 1. Uma vez que o primeiro bit da mantissa é sempre igual a 1 é possível considerá-lo implícito e omiti-lo do campo da mantissa. A omissão deste bit permite o ganho de mais um bit significativo na mantissa. O bit omitido da representação normalizada é denominado **bit escondido** ou **bit normal**. Assim, considerando o formato de 32 bits apresentado acima, um campo de mantissa igual a 10000000000000000000001 está associado ao seguinte valor:

Binário	**Decimal**
1,10000000000000000000001	$2^0 + 2^{-1} + 2^{-23}$

Para uma melhor compreensão da representação em ponto flutuante normalizada, são apresentados na tabela 3.7 alguns exemplos. Nestes exemplos, observar que:

- O sinal é armazenado no bit mais significativo da palavra de 32 bits;
- O expoente está expresso na representação em excesso de 127 (excess-127). A representação em excesso de n (*excess-n* ou *bias-n*) é um forma de se representar números negativos. Por motivos históricos ela tem sido utilizada para representar o expoente em diversas representações em ponto flutuante. Na representação excesso de n, o inteiro I é representado pelo valor n + I. Desta forma, o valor expresso no campo do expoente deve ser subtraído de n para o cálculo o valor real do expoente. Considerando a representação excesso de 127, os valores expressos no campo do expoente podem assumir os valores 0 até 255 e, conseqüentemente, o expoente real está na faixa –128 até 127 (veja tabela 3.6);
- O primeiro bit após a vírgula é sempre igual a 1 e não é armazenado no campo da mantissa;
- A base da representação é 2.

Ponto Flutuante	**Binário**	**Decimal**
0 10010011 10000010000000000000000	$1{,}1010001\ 10^{(10010011 - 0111111)}$	$(2^{-0} + 2^{-1} + 2^{-7})\ 2^{20}$
1 10010111 10000010000000000000000	$-1{,}1010001\ 10^{(10010011 - 0111111)}$	$-(2^{-0} + 2^{-1} + 2^{-7})\ 2^{20}$
0 01101011 10000010000000000000000	$1{,}1010001\ 10^{(01101011 - 0111111)}$	$(2^{-0} + 2^{-1} + 2^{-7})\ 2^{-20}$
1 01101011 10000010000000000000000	$-1{,}1010001\ 10^{(01101011 - 0111111)}$	$-(2^{-0} + 2^{-1} + 2^{-7})\ 2^{-20}$

Tabela 3.7: Exemplos de números normalizados

Decimal	**Excesso – 127**	**Complemento de 2**
-128	-	10000000
-127	00000000	10000001
-126	00000001	10000010
-125	00000010	10000011
…	…	…
-2	01111101	11111110
-1	01111110	11111111

0	01111111	00000000
…	…	…
125	11111100	01111101
126	11111101	01111110
127	11111110	01111111
128	11111111	-

Tabela 3.6: Representação de números negativos.

É importante observar que a limitação do número de bits disponíveis para a representação limita os números reais possíveis de serem representados. Considerando a representação de 32 bits discutida acima, as faixas de números reais representáveis são apresentadas na figura 3.5. Da representação e da figura, observar que:

- 1 11111111 11111111111111111111111 = $-(2\text{-}2^{-23})\ 2^{128}$;
- 1 00000000 00000000000000000000000 = -2^{-127};
- 0 00000000 00000000000000000000000 = 2^{-127};
- 0 11111111 11111111111111111111111 = $(2\text{-}2^{-23})\ 2^{128}$;
- Os números negativos representáveis estão na faixa entre $-(2\text{-}2^{-23})\ 2^{128}$ até -2^{-127};
- Os números positivos representáveis estão na faixa entre 2^{-127} até $(2\text{-}2^{-23})\ 2^{128}$;
- Valores negativos menores do que $-(2\text{-}2^{-23})\ 2^{128}$, denominados *overflow* negativo, não são representáveis;
- Valore negativos maiores do que -2^{-127}, denominados *underflow* negativo, não são representáveis;
- Valores positivos menores do que 2^{-127}, denominados *underflow* positivo, não são representáveis;
- Valores positivos maiores do que $(2\text{-}2^{-23})\ 2^{128}$, denominados *overflow* positivo, não são representáveis;
- O valor Zero NÃO é representável.

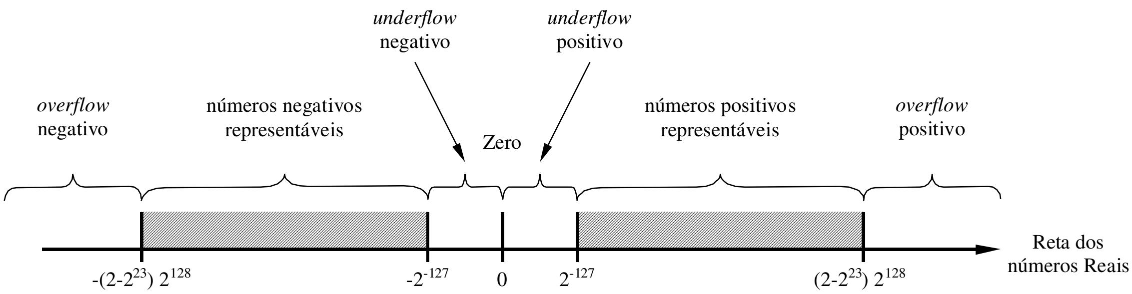

Fig. 3.5: Valores representáveis - formato 32 bits

É importante frisar que a representação em ponto flutuante não representa o contínuo dos números reais. A representação é discreta. Observar que com 32 bits só é possível representar 2^{32} números diferentes. Na representação em ponto flutuante estes valores individuais não estão igualmente espaçados. Os valores representáveis estão mais condensados perto da origem e menos longe desta (figura 3.6). Para um dado valor de expoente E, a diferença entre dois valores consecutivos é dado por 2^{E-23}.

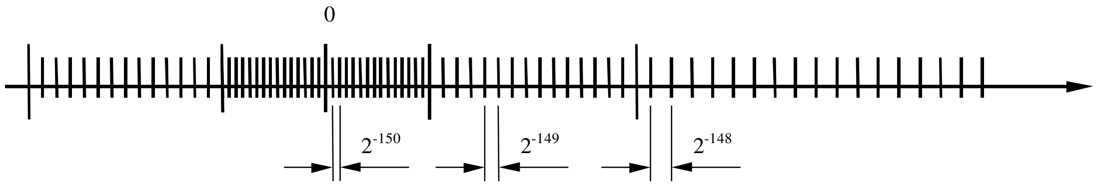

Fig. 3.6: Distribuição dos valores representáveis – formato 32 bits

5. Padrão IEEE 754 para representação em ponto flutuante

Até aproximadamente 1980 vários formatos para a representação em ponto flutuante coexistiam no mercado; um para cada fabricante de computador/processador. Esta profusão de representações impunha um grande entrave para a troca de informação entre máquinas. A insatisfação e os problemas associados a existência de vários formatos levou a um esforço de padronização coordenado pelo IEEE (Institute of Electrical and Electronic Engineers), resultando no padrão IEEE 754. Este padrão, atualmente implementado na maioria das CPU e co-processadores aritmético que processam ponto flutuante (entre eles: Intel, Motorola, SPARC, e MIPS), estabelece formatos e algoritmos para as operações aritméticas envolvendo a esta representação.

O padrãro IEEE 754 estabelece 3 formatos para a representação em ponto flutuante: precisão simples (32 bits), precisão dupla (64 bits), e precisão estendida (80 bits). O formato estendido foi definido para minimizar erros de arredondamento, sendo utilizado apenas internamente às unidades de processamento. Neste laboratório não discutiremos este formato.

A figura 3.7 apresenta os formatos para representações em precisão simples e dupla números do padrão IEEE 754, explicitando o número de bits associados a cada campo.

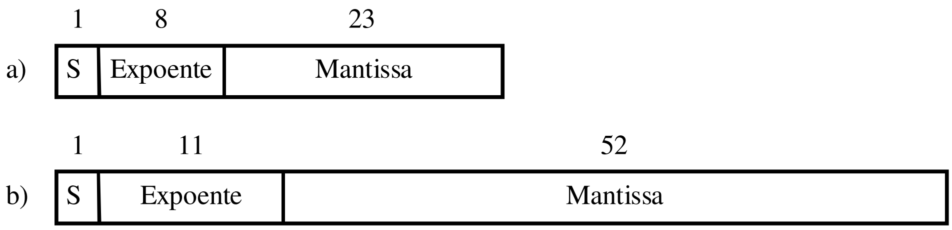

Fig. 3.7: Padrão IEEE 745 – a) precisão simples, b) precisão dupla

Para os dois formatos o bit “S” especifica o sinal (0 → positivo, 1 → negativo). O expoente é especificado em excesso de 127 para os números em precisão simples, e em excesso de 1023 para números em precisão dupla. O campo de expoente com valor igual a 0 é utilizado para representar valores especiais. O campo de expoente com valor máximo também é utilizado para sinalizar situações especiais. O campo da mantissa é representado com 23 e 52 bits nas representações em precisão simples e duplas, respectivamente.

O padrão IEEE estabelece que os números estão representados na seguinte forma normalizada (representação em binário):

$$(-1)^S \times 1{,}dddd\ldots \times 10^E$$

Onde “S” é o bit do sinal; a seqüência “dddd…” representa os bits armazenados no campo da mantissa; e “E” é o expoente. O expoente “E” é igual ao valor armazenado no campo do expoente menos 127 para precisão simples, e menos 1023 para precisão dupla. O bit antes da vírgula é considerado o **bit escondido**, sendo omitindo da mantissa, porém considerado no cálculo do valor representado.

Para representar valores especiais como -∞, +∞, zero, números não normalizados, e números indefinidos (NaN – Not a Number), o padrão IEEE reserva certas configurações de bits. A figura 3.8 apresenta os tipos numéricos suportados pelo padrão IEEE 754.

Número normalizado	±	0 < expoente < Max	qualquer configuração de bits
Número Não Normalizado	±	0000..0	qualquer configuração de bits diferente de 0
Zero	±	0000…0	0000…0
Infinito	±	1111…1	0000…0
NaN	±	1111…1	qualquer configuração de bits diferente de 0

Fig. 3.8: Tipo numéricos do padrão IEEE 754

Uma característica interessante do padrão IEEE 754 é a existência de números não normalizados para representar valores muito pequenos que, de outra forma, estariam na faixa de *underflow* e seriam arredondados para zero. Como apresentado na figura 3.8, os números não normalizados possuem o campo do expoente igual a zero e campo da mantissa diferente de zero, sendo que tal configuração de bits é interpretada da seguinte forma (representação em binário):

$$(-1)^S \times 0{,}dddd... \times 10^E$$

Onde "E" é igual a -126 na representação em precisão simples, e igual a -1022 na precisão dupla.

Considerando que "e" é o valor armazenado no campo do expoente; "m" o valor armazenado no campo da mantissa, e "S" o bit do sinal, os valores representados em precisão simples e dupla são apresentados na tabela 3.8.

Precisão Simples		
Expoente e	**Mantissa m**	**Valor**
255	$\neq 0$	NaN
255	0	$(-1)^S \infty$
0 < e< 255	Qualquer valor	$(-1)^S (1,m)\ 2^{e-127}$
0	$\neq 0$	$(-1)^S (0,m)\ 2^{-126}$
0	0	$(-1)^S\ 0$

Precisão Dupla		
Expoente e	**Mantissa m**	**Valor**
2047	$\neq 0$	NaN
2047	0	$(-1)^S \infty$

0 < e< 2047	Qualquer valor	$(-1)^S\ (1,m)\ 2^{e-1023}$
0	$\neq 0$	$(-1)^S\ (0,m)\ 2^{-1022}$
0	0	$(-1)^S\ 0$

Tabela 3.8: Padrão IEEE 754 – valores dos números em ponto flutuante

Considerando os números <u>normalizados</u> representáveis pelo padrão IEEE é importante observar que:

- A mantissa dos números normalizados representam valores maiores ou iguais a 1 e pouco menores do que 2;
- O menor número positivo representável em precisão simples é 0 00000001 00000000000000000000000 = 2^{-126};
- O maior número positivo em precisão simples é 0 11111110 11111111111111111111111 = $(1 + 2^{-1} + 2^{-2} + 2^{-3} + \ldots + 2^{-23})\ 2^{127} = (2 - 2^{-23})\ 2^{127} =$ $1{,}99999988079071044921875\ 2^{127} \approx 2^{128} \approx 3{,}4\ 10^{38}$.

Considerando os números <u>não normalizados</u> observa-se que:

- Não há bit escondido (o bit escondido só é implícito na representação normalizada). Por exemplo, em precisão simples, o número 0 00000000 10000000000000000000000 representa o valor 2^{-1} x $2^{-126} = 2^{-127}$.
- maior número positivo representável em precisão simples é 0 00000000 11111111111111111111111 = $0{,}99999988079071044921875\ 2^{-126}$ (compare com o menor número positivo normalizado).
- menor número positivo em precisão simples é 0 00000000 00000000000000000000001 = 2^{-23} x $2^{-126} = 2^{-149}$.
- A representação de números não normalizados definida no padrão IEEE permite o tratamento do *underflow* (representação de números pequenos) de forma suave, evitando o arredondamento abrupto para zero.

No padrão IEEE existem dois zero (-0 e +0). Na representação dos zeros também está implícito que não existe bit escondido; ou seja, o bit antes da vírgula é igual a 0.

A situação de *overflow* (representar números muito grandes - positivos ou negativos) é tratada através da configuração de bits que representa valores infinitos. O padrão define que operações com os valores infinitos segue as regras tradicionais: infinito mais algo é infinito; um número finito dividido por infinito é zero; um número finito divido por zero é infinito.

Finalmente, para situações indefinidas como, por exemplo, a divisão de infinito por infinito, e de zero por zero, o padrão estabeleceu a representação NaN (Not a Number). Operações envolvendo um NaN resulta sempre em um NaN.

5.1 Operações aritméticas em ponto flutuante

Considere os números

$$x = (-1)^{X_S} \times X_M \times B^{X_E} \quad (\ X_S - \text{sinal};\ X_M - \text{mantissa};\ X_E - \text{expoente})$$

$$y = (-1)^{Y_S} \times Y_M \times B^{Y_E} \quad (Y_S - \text{sinal};\ Y_M - \text{mantissa};\ Y_E - \text{expoente})$$

As quatro operações aritméticas básicas são:

$$\left.\begin{array}{l} x + y = \left[\left((-1)^{X_S} \times X_M \times B^{X_E - Y_E} \right) + \left((-1)^{Y_S} \times Y_M \right) \right] \times B^{Y_E} \\ x - y = \left[\left((-1)^{X_S} \times X_M \times B^{X_E - Y_E} \right) - \left((-1)^{Y_S} \times Y_M \right) \right] \times B^{Y_E} \end{array} \right\} \quad X_E \leq Y_E$$

$$x \times y = \left((-1)^{X_S} \times X_M \right) \times \left((-1)^{Y_S} \times Y_M \right) \times B^{X_E + Y_E}$$

$$x \div y = \left((-1)^{X_S} \times X_M \right) \div \left((-1)^{Y_S} Y_M \right) \times B^{X_E - Y_E}$$

5.1.1 Adição e Subtração

A adição e subtração é necessário que os operandos seja ajustados para que ambos tenham o mesmo valor de expoente. Para tanto, o operando com menor valor de expoente deve ter a sua vírgula deslocada para a direita até que o seu expoente se iguale ao do outro operando – esta operação é conhecida como alinhamento das mantissas. O algoritmo de soma e subtração pode ser dividido em quatro fases:

1. Testar a existência de zero, ∞, e NaN
2. Alinhar as mantissas
3. Somar o subtrair as mantissas
4. Normalizar o resultado

Nas figuras 3.9a e 3.9b tem-se o fluxograma de um algoritmo de adição e subtração em ponto flutuante. Para uma melhor compreensão dos detalhes implícitos neste fluxograma será um exemplo em precisão simples. Considere a seguinte adição z = x + y, com x = 0,25 e y = -8 (valores em decimal).

A representação em ponto flutuante do valor 0,25 é 0 01111101 00000000000000000000000.
A representação do valor -8 é 1 10000010 00000000000000000000000.

Antes do início do algoritmo considera-se que os valores do sinal, expoente e mantissa de cada números foram isolados em, por exemplo, posições de distintas de memória denominadas X_S, X_E, X_M, Y_S, Y_E, Y_M. Para o exemplo, ter-se-ia (números iniciados com um $ são números em hexadecimal, seguindo a convenção adotada neste laboratório):

- X_S = \$00; bit do sinal do valor 0,25 representado em um byte;
- X_E = \$7D; expoente do valor 0,25 representado em excesso de 127;
- X_M = \$80000000; mantissa (incluindo o bit escondido) do valor 0,25 representada em 32 bits. A mantissa é estendida para 32 bits para efetuar as operações sem perder precisão. Os 8 bits zeros inseridos na parte menos significativa é denominado de bits de guarda (*guard bits*). Em geral, os processadores suportam operações com operando de comprimento múltiplo de 8, sendo 32 é o múltiplo superior de 8 mais próximo de 24.
- Y_S = \$01; bit do sinal do valor -8 representado em byte;
- Y_E =\$82; expoente do valor –8 representado em excesso de 127;
- Y_M = \$80000000; mantissa do valor –8;

Assim, no fluxograma do figura 3.9a e 3.9b, após os testes da situações especiais (zero, ∞, NaN), tem-se a comparação dos expoentes X_E e Y_E. No exemplo, $X_E < Y_E$, resultando que o expoente X_E será incrementado e a mantissa X_M deslocada para a direita até que X_E fique igual a Y_E. Assim, após a execução do laço de alinhamento, ter-se-á X_E = \$82, e X_M = \$04000000.

Após o alinhamento, tem-se a soma das mantissas considerando-se os sinais. A operação a ser realizada será \$80000000 - \$04000000, resultando em \$7C000000 e sinal \$0001 (negativo). A operação com as mantissa considerando-se os sinais segue a regra: Mantissas com sinais iguais são somadas; mantissas com sinais diferentes são subtraídas (a menor e subtraída da maior e o sinal do resultado é igual ao sinal da maior).

Como não ocorreu *overflow* da mantissa, o próximo passo será normalizar o resultado. Para tanto, a mantissa será deslocada para a esquerda, e o expoente decrementado. Após a normalização tem-se os seguintes valores: mantissa = \$F8000000, expoente = \$81. O resultado final da operação em ponto flutuante será: sinal = 1; expoente = 1000001; mantissa = 1111000000000000000000 (bit escondido). Ou seja:

1 10000001 11110000000000000000000000 = $(-1)^1 (1 + 2^{-1}+ 2^{-2}+ 2^{-3} + 2^{-4}) 2^2 = -7{,}75$

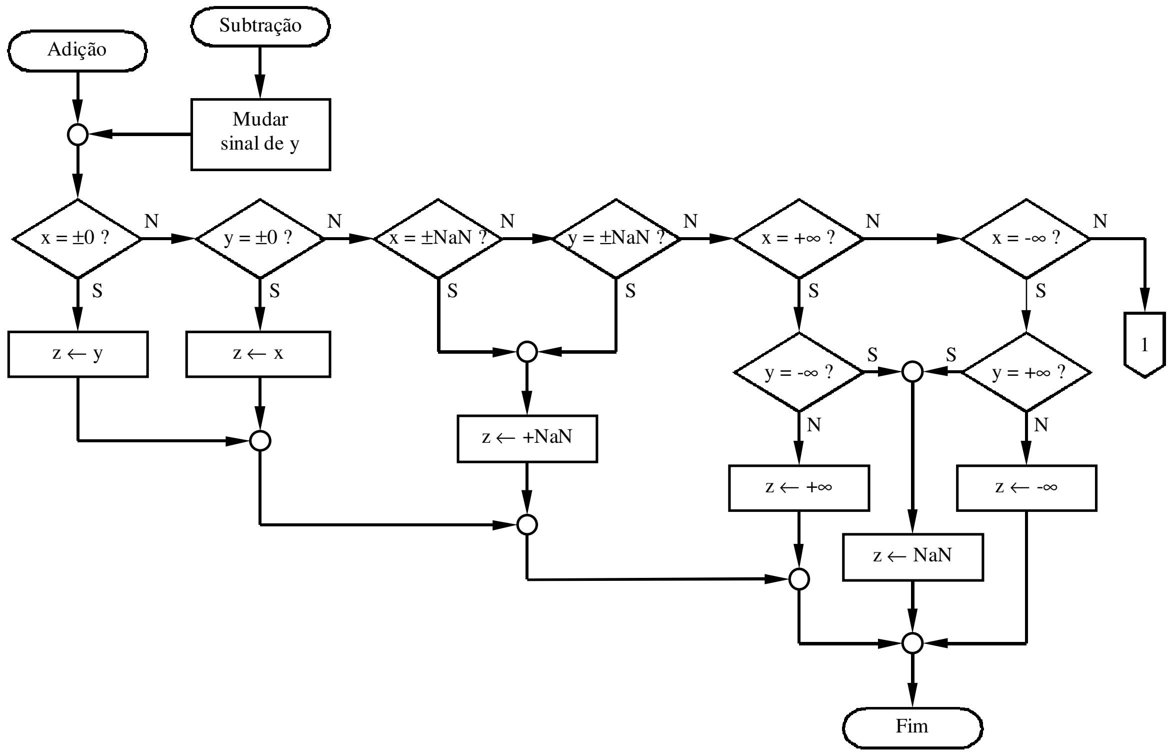

Fig. 3.9a: Fluxograma da adição e subtração em ponto flutuante.

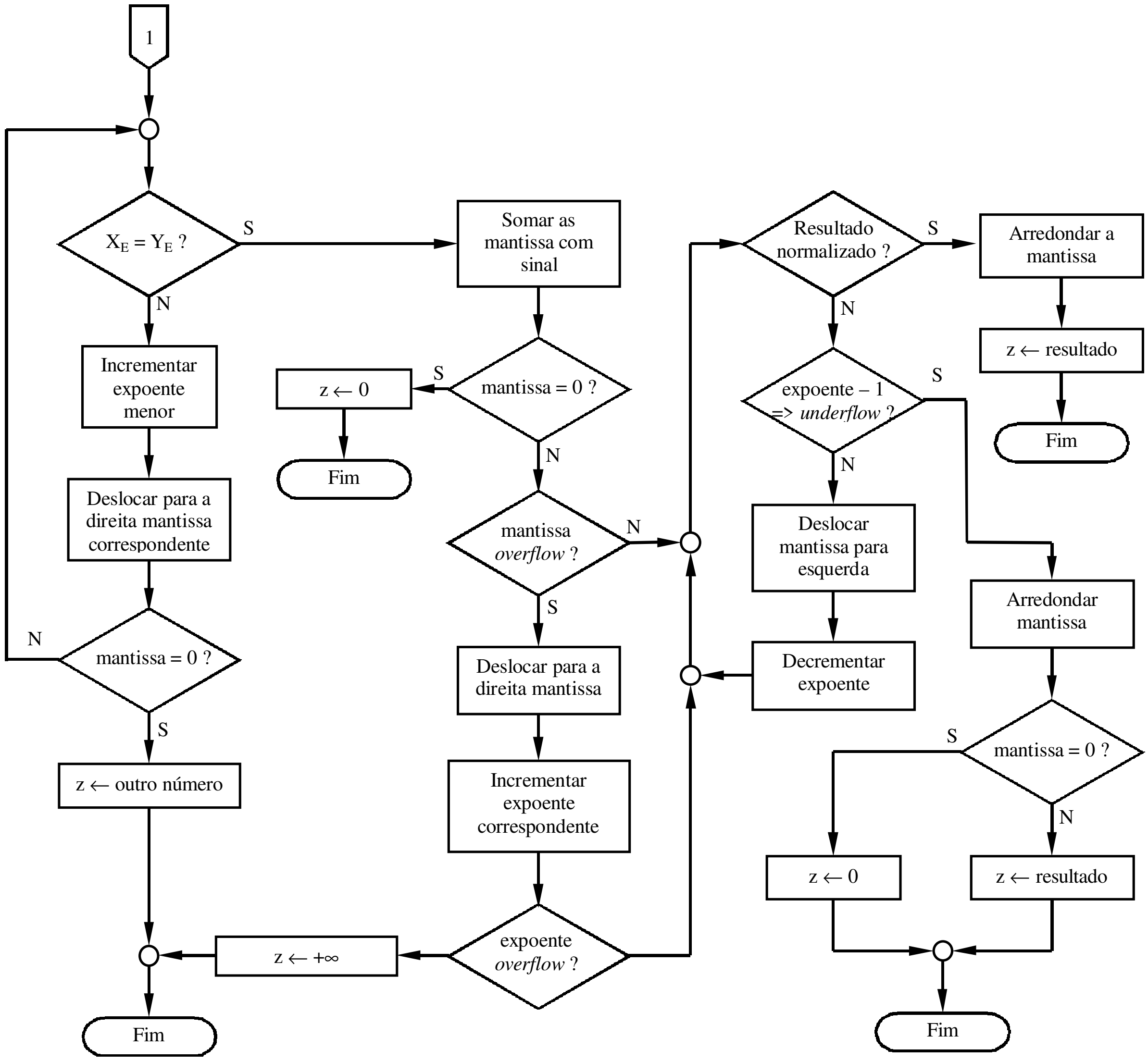

Fig. 3.9b: Fluxograma da adição e subtração em ponto flutuante – continuação.

5.1.2 Multiplicação

As figuras 3.10a e 3.10b apresentam o fluxograma de um algoritmo para a multiplicação de dois número em ponto flutuante. Como para a adição e subtração, este algoritmo será explicado através de um exemplo.

Considere a operação de multiplicação z = x . y, com x = 0,25 e y = –8 (valores em decimal). Como discutido na seção anterior, antes do início do algoritmo tem-se:

- X_S = \$00;
- X_E = \$7D;
- X_M = \$80000000;
- Y_S = \$01;
- Y_E =\$82;
- Y_M = \$80000000;

Como apresentado nas figuras 3.10a, o primeiro passo do algoritmo consiste calcular o sinal do resultado, multiplicando-se os sinais. Esta multiplicação dos sinais é efetuada através do ouexclusivo (*exclusive-or*) bit a bit dos bytes que representam os sinais dos operandos, de tal forma que: positivo x positivo = positivo ($0 \oplus 0 = 0$), negativo x positivo = negativo ($1 \oplus 0 = 1$); positivo x negativo = negativo ($0 \oplus 1 = 1$), e negativo x negativo = positivo ($1 \oplus 1 = 0$). No exemplo o sinal resultante é $01 (negativo).

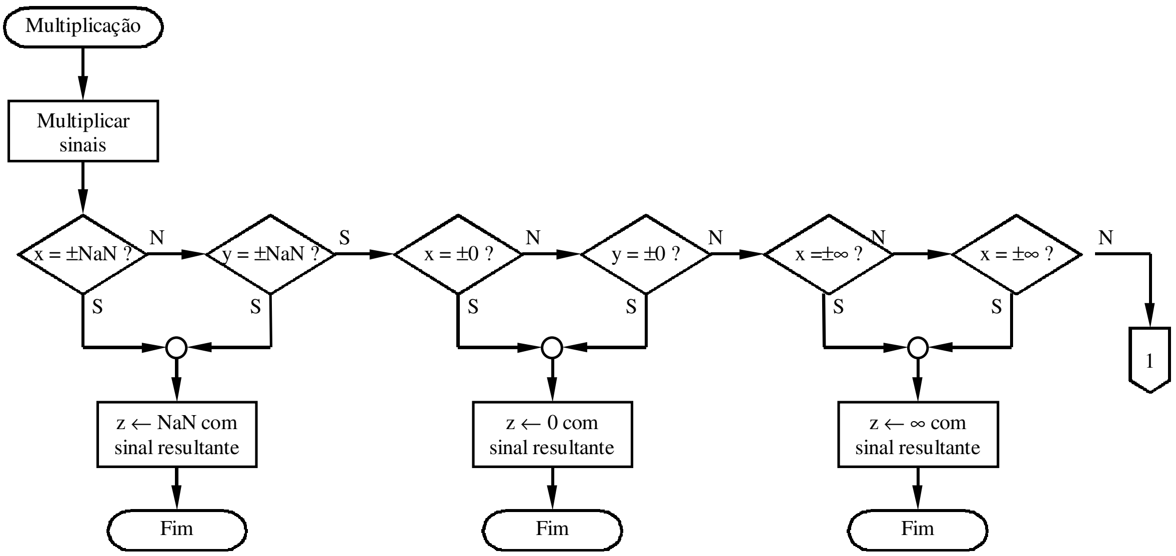

Fig. 3.10a: Fluxograma da multiplicação em ponto flutuante.

Após o cálculo do sinal resultante e os testes das situações especiais (zero, ∞, NaN), tem-se a soma dos expoentes dos operandos $7D + $82 = $FF. Como os expoentes estão representados em excesso de 127, faz-se necessário corrigir o resultado subtraindo-se o valor 127. A razão desta correção fica clara ao se observar que o valor do campo do expoente em excesso de n (n é chamado de *bias* - no exemplo n = 127) possui o valor n + E, onde E é o valor real do expoente. Assim, ao se somar dois expoentes tem-se $n + E_1 + n + E_2 = E_1 + E_2 + 2n$, fazendo-se necessário a subtração do valor n para a correção do resultado para $E_1 + E_2 + n$. No exemplo, a subtração do *bias* igual a 127 resulta $FF - $7F = $80.

Como não ocorreu *overflow*, nem tampouco *underflow* do expoente, o próximo passo do algoritmo é a multiplicação das mantissas dos operandos $80000000 x $80000000 = $4000000000000000. Entretanto, cada mantissa possui 23 bits após a virgula, assim o resultado deve ser deslocado para a direita de 43 bits, resultando em $80000000. Consequentemente, o resultado da multiplicação é:

$$1\ 10000000\ 10000000000000000000000 = 2^1$$

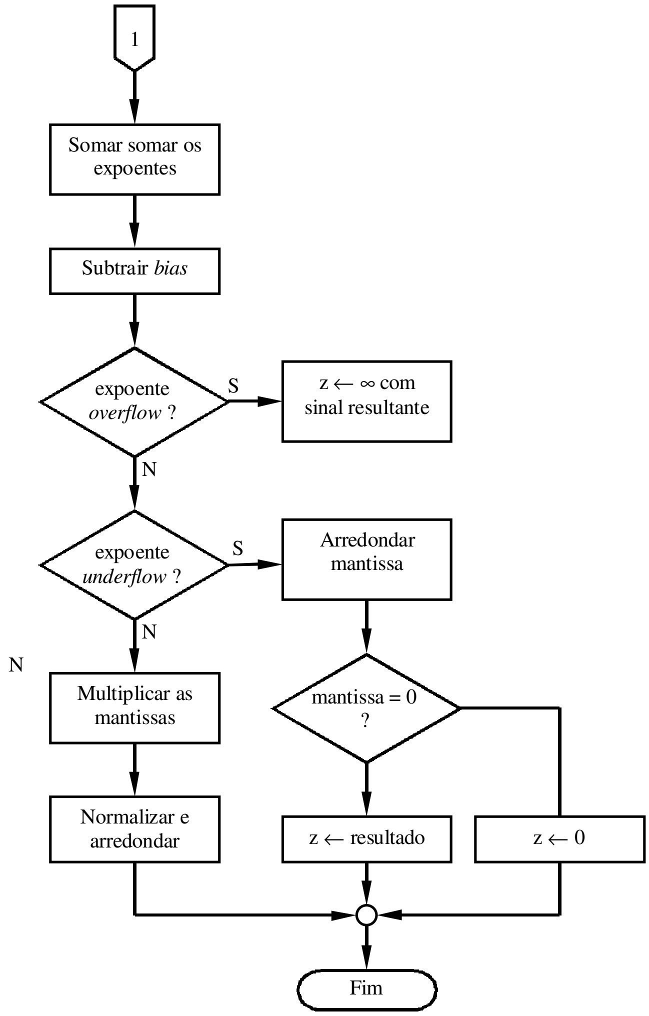

Fig. 3.10b: Fluxograma da multiplicação em ponto flutuante – continuação.

5.1.3 Divisão

A divisão é análoga à multiplicação. As figuras 3.11a e 3.11b apresentam o algoritmo da divisão

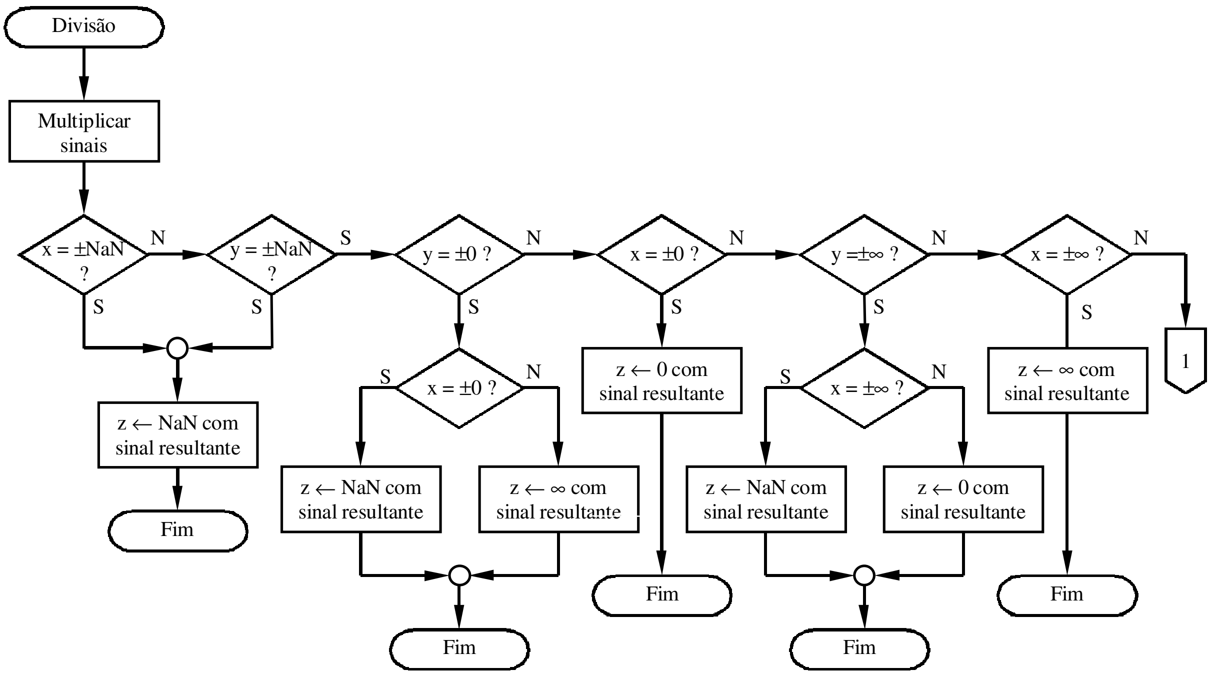

Fig. 3.11a – Fluxograma da divisão em ponto flutuante

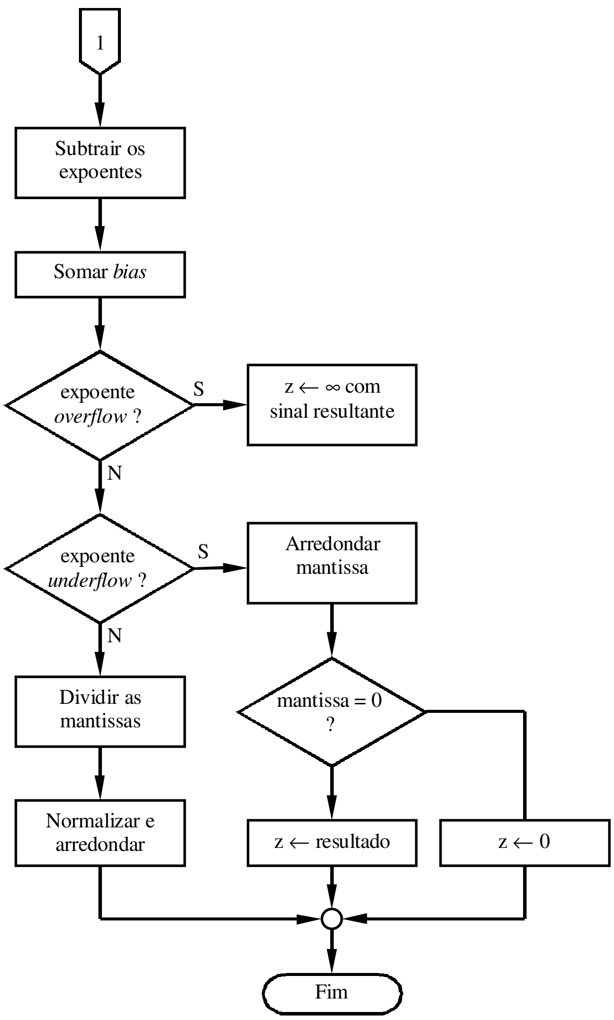

Fig. 3.11b – Fluxograma da divisão em ponto flutuante - continuação

6. Exercícios

1. (2,0 pts) Fazer sub-rotina que converta números decimais com parte fracionária em representação em ponto flutuante – IEEE 754 precisão simples. Esta sub-rotina tem como entrada um *buffer* de caracteres ASCII, com A5 apontando para o seu início. Considerar que o *buffer* de entrada é formado por 11 caracteres no seguinte formato: ±NN,NNNE±NN. Onde "±" pode ser o caracter ´+´ ($2B) ou o ´– ($2D); "N" é um caracter numérico de 0 a 9; "," é o caracter ´,´ ($2C); "E" é o caracter ´E´ ($45). Assim, por exemplo, o número +12,345 10^{-67} é armazenado no *buffer* como: $2B $31 $32 $2C $33 $34 $35 $45 $2D $36 $37, que equivale à seguinte seqüência de caracteres *(string)* ´+12,345E-67´.

2. (2,0 pts)Fazer sub-rotina que mostre na tela do terminal o valor decimal de um número em ponto flutuante IEEE 754 – precisão simples armazenado em D0.

3. (2,0 pts) Fazer sub-rotina para a adição e sub-rotina para a subtração de dois números em ponto flutuante conforme o padrão IEEE 754. Considera apenas precisão simples. Estas sub-rotinas têm como parâmetros de entrada D0 e D1 e retornam o resultado em D0. O resultado da sub-rotina de subtração deve ser D0 – D1.

4. (2,0) Fazer sub-rotina para a multiplicação de dois números em ponto flutuante conforme o padrão IEEE 754. Considera apenas precisão simples. Esta sub-rotina tem como parâmetros de entrada D0 e D1 e retorna o resultado em D0.

5. (2,0) Fazer sub-rotina para a divisão de dois números em ponto flutuante conforme o padrão IEEE 754. Considerar apenas precisão simples. Esta sub-rotina tem como parâmetros de entrada D0 e D1 e retorna o resultado em D0 = D0 / D1.

7. Referências

[1] Arnewsh Inc.: *SBC68K User's Manual Revison 1.1*, 2nd Edition, 1990.

[2] Andrew S. Tanenaum: Strutured Computer Organization; 3rd Edition; Prentice Hall International Editions; 1990.

[3] William Stallings: Computer Organization and Architecture – Principles of Structure and Function; 3rd Edition; Prentice Hall International Editions; 1993.

[4] Robert Baron, Lee Higbie: Computer Architecture; Addison-Wesley Publishing Company; 1994.

[5] B. Daltrini, M. Jino, L.P. Magalhães: Introdução a Sistemas de Computação Digital; Apostila do curso EA869.

ANEXO A

68000 ASSEMBLER

USER'S MANUAL

Paul McKee

1. Introduction

The program described here, 68000 Assembler, is a basic two-pass assembler for the 68000 and 68010 microprocessors. It supports the complete instruction set of both processors as well as a modest but capable set of assembler directives. The program produces formatted listing files as well as object code files in S-record format.

The program was written in VAX-11 C by Paul McKee during the fall semester, 1986. The program should be portable (with reasonable changes) to any C language implementation that supports 32-bit integers.

2. Source Code Format

2.1 Source Line Format

The input to the assembler is a file containing instructions, assembler directives, and comments. Each line of the file may be up to 256 characters long. It is recommended, however, that the source lines be no longer that 80 characters, as this will guarantee that the lines of the listing file do not exceed 132 characters in length. The assembler treats uppercase and lowercase identically.

Each line of the source code consists of the following fields:

```
LABEL   OPERATION    OPERAND,OPERAND,...   COMMENT
```

For example,

```
LOOP        MOVE.L   (A0)+,(A1)+   Sample source line
```

The fields may be separated by any combination of spaces and tabs. Except for the comment field and quoted strings, there must be no spaces or tabs within a field.

2.2 Label Field

Legal labels follow the rules for forming symbol names described in section 2.2. Labels may be distinguished in one of two ways: (1) They may begin in column 1, or (2) they may end in a colon, which does not become part of the label but simply serves to mark its end. A line may consist of a label alone. When a label is encountered in the source code, it is defined to have a value equal to the current location counter. This symbol may be used elsewhere is the program to refer to that location.

2.3 Operation Field

The operation field specifies the instruction that is to be assembled or the assembler directive that is to be performed. A size code (.B, .W, .L, or .S) may be appended to the operation code if allowed, to specify Byte, Word, Long, or Short operations, respectively. The operation field must not begin in the column 1, because the operation would be confused with a label.

2.4 Operand Field

The operand field may or may not be required, depending on the instruction or directive being used. If present, the field consists of one or more comma-separated items with no intervening spaces or tabs. (There may be spaces or tabs within an item, but only within quoted strings.)

2.5 Comment Field

The comment field usually consists of everything on a source line after the operand field. No special character is needed to introduce the comment, and it may contain any characters desired.

A comment may also be inserted in the source file in another way: An asterisk ("*") at the beginning of the line or after the label field will cause the rest of the line to be ignored, i.e., treated as a comment.

3. Symbols

Symbols appear in the source code as labels, constants, and operands. The first character of a symbol must be either a letter (A-Z) or a period ("."). The remaining characters may be letters, dollar signs ("$"), periods ("."), or underscores("_"). A symbol may be of any length, but only the first 8 characters are significant. Remember that capitalization is ignored, so symbols which are capitalized differently are really the same.

4. Expressions

An expression may be used in the source program anywhere a number is called for. An expression consists of one or more operands (numbers or symbols), combined with unary or binary operators. These components are described below. The value of the expression and intermediate values are always computed to 32 bits, with no account being made of any overflow that may occur. (Division by zero, however, will cause an error.)

4.1 Operands in Expressions

An operand in an expression is either a symbol or one of the following sorts of constants.

4.1.1 Decimal Numbers

A decimal number consists of a sequence of decimal digits (0-9) of any length. A warning will be generated if the value of the number cannot be represented in 32 bits.

4.1.2 Hexadecimal Numbers

A hexadecimal number consists of a dollar sign ("$") fol lowed by a sequence of hexadecimal digits (0-9 and A-F) of any length. A warning will be generated if the value of the number cannot be represented in 32 bits.

4.1.3 Binary Numbers

A binary number consists of a percent sign ("%") followed by a sequence of binary digits (0 and 1) of any length. A warning will be generated if the number consists of more that 32 digits.

4.1.4 Octal Numbers

An octal number consists of a commercial at sign ("@") followed by a sequence of octal digits (0-7) of any length. A warning will be generated if the value of the number cannot be represented in 32 bits.

4.1.5 ASCII Constants

An ASCII constant consists of one to four ASCII characters enclosed in single quote marks. If it is desired to put a single quote mark inside an ASCII constant, then two consecutive single quotes may be used to represent one such character.

If the ASCII constant consists of one character, then it will be placed in the bottom byte of the 32 bit value; two characters will be placed in the bottom word, with the first character in the higher-order position. If four characters are used, then all four bytes will contain characters, with the first in the highest-order location. However, if three characters are used, then they will be placed in the three highest-order bytes of the 32-bit value, with 0 in the low byte (this is to accommodate the high-byte-first addressing used on the 68000).

Note that ASCII constants in expressions are different from strings in DC directives, as the latter may be of any length.

4.2 Operators in Expressions

The operators allowed in expressions are shown in the following table, in order of decreasing precedence. Within each group, the operators are evaluated in left-to-right order (except for group 2, which is evaluated right-to-left).

4.2.1.1.1.1 Operators in Expressions

1.	()	Parenthesized subexpressions
2.	-	Unary minus (two´s complement)
	~	Bitwise not (one´s complement)
3.	<<	Shift left (x<<y produces x shifted left by y bits and zero filled)
	>>	Shift right
4.	&	Bitwise And
	!	Bitwise Or
5.	*	Multiplication
	/	Integer division
	\	Modulus (x\y produces the remainder when x is divided by y)
6.	+	Addition
	-	Subtration

5. Addressing Mode Specifications

The 68000 and 68010 provide 14 general addressing modes. The formats used to specify these modes in assembly language programs are listed in the table below. The following symbols are used to describe the operand formats:

Dn = Data Register

An	=	Address Register (SP may used instead of A7)
Xn	=	Data or Address register
.s	=	Index register size code (either .W or .L, .W will be assumed if omitted)
<ex8>	=	Expression that evaluates to an 8-bit value (may be empty, in which case 0 will be used)
<ex16>	=	Expression that evaluates to a 16-bit value (may be empty, in which case 0 will be used)
<ex>	=	Any expression
PC	=	Program Counter

Addressing Mode Specifications

Mode	Assembler Format
Data Register Direct	Dn
Address Register Direct	An
Address Register Indirect	(An)
Address Register Indirect with Predecrement	-(An)
Address Register Indirect with Postincrement	(An)+
Address Register Indirect with Displacement	<ex16>(An)
Address Register Indirect with Index	<ex8>(An,Xn.s)
Absolute Short or Long (chosen by assembler)	<ex>
Program Counter with Displacement	<ex16>(PC)
Program Counter with Index	<ex8>(PC,Xn.s)
Immediate	#<ex>

In addition to the general addressing modes, the following register names may be used as operands in certain instructions (e.g., MOVEC or EORI to CCR):

SR	=	Status Register
CCR	=	Condition Code Register
USP	=	User Stack Pointer
VBR	=	Vector Base Register (68010)
SFC	=	Source Function Code Register (68010)
DFC	=	Destination Function Code Register (68010)

6. Assembly Details

6.1 Branch Instructions

The branch instructions (Bcc, BRA, and BSR) are unique in that they can take a ".S" size code. This suffix directs the assembler to assemble these as short branch instructions, i.e., one-word instructions with a range to -128 to +127 bytes. If the ".S" size code is used, and the destination is actually outside this range, then the assembler will print an error message. If the ".L" size code is used, the assembler will use a long branch, which is a two-word instruction with a range of -32768 to +32767 bytes. If neither size code is specified, then the assembler will use a short branch if possible (the branch destination must be known on the first pass to be within the short branch range); otherwise it will use long branch.

6.2 MOVEM Instruction

The MOVEM instruction, which is used for saving and restoring sets of registers, has one the following two forms:

```
MOVEM   <register_list>,<effective_address>

MOVEM   <effective_address>,<register_list>
```

The register list may be an explicit register list of the form described in Section 4.2.3. On the other hand, if a particular set of registers is to be saved and restored repeatedly, the REG directive (Section 4.2.3) can be used to define a register list symbol that specifies the registers. For example, if the register list symbol WORKSET is defined as follows:

```
WORKSET REG A0-A4/D1/D2
```

then the following instructions will perform the same function:

```
MOVEM.L WORKSET,-(SP)

MOVEM.L A0-A4/D1/D2,-(SP)
```

If a register list symbol is used, it must be defined before it appears in any MOVEM instructions.

6.3 Quick Instructions (MOVEQ, ADDQ, SUBQ)

The MOVE, ADD, and SUB instructions have one-word "quick" variations which can be used certain addressing modes and operand values. The assembler will use these faster variations automatically when possible, or they may be specified explicitly by writing the mnemonic as MOVEQ, ADDQ, or SUBQ.

The MOVEQ instruction may be used for moving an immediate value in the range -128 to +127 into a data register. The assembler will assemble a MOVE.L #<value>,Dn as a MOVEQ if the value is known on the first pass.

The ADDQ (SUBQ) instruction adds (subtracts) an immediate value from 1 to 8 to (from) any alterable destination. The assembler will use the quick form if the value is known on the first pass to be in the range 1 to 8.

7. Assembler Directives

7.1 ORG - Set Origin

The assembler maintains a 32-bit location counter, whose value is initially zero and which is incremented by some amount whenever an instruction is assembled or a data storage directive is carried out. The value of this location counter may be set with the ORG directive. This is typically done at the start of a program and at appropriate places within it. The format of the ORG directive is

```
<label> ORG <expression>
```

where <expression> is an expression containing no forward references, i.e., its value must be known on the first pass at the point where the ORG directive appears. An error will result if an attempt is made to set the location counter to an odd value; in this case the location counter will be set to the specified value plus one. The <label> is optional and, if present, the specified symbol will be set to the new value of the location counter.

8. Symbol Definition Directives

8.1 EQU - Equate Symbol

The equate directive is used to define symbols whose value will not change within the program. The format of this directive is

```
<label> EQU <expression>
```

where <expression> is an expression containing no forward references, i.e., its value must be known on the first pass at the point where the EQU directive appears. The <label> must be specified, since it tells what symbol is being defined. If <label> is omitted, an error will result. If an attempt is made to redefine a symbol that was defined with EQU, either as a label or using any symbol definition directive, an error message will be printed.

8.2 SET - Set Symbol

The SET directive is similar in function and format to the equate directive, with one important difference: symbols defined using SET may be redefined later using another SET directive (but not using a EQU or REG directive). The format of this directive is

```
<label> SET <expression>
```

8.3 REG - Register Range

Register ranges consist of lists of registers separated by slashes ("/"). Each register range may be either a single register ("An" or "Dn") or a range of registers ("An-Am" or "Dn-Dm"), which denotes all the registers between the two registers listed (they may be given in either order). For example, the following register list specifies that D0, D1, D2, D3, D7, A1, A2, and A3 are to be saved (or restored):

```
D3-D0/D7/A1-A3
```

The registers and ranges may be specified in any order. The same format for register lists may be used with the MOVEM instruction directly. In order to avoid confusion, it is best to avoid specifying a range that includes both an address register and a data register, although the assembler will not treat this as an error.

8.4 Data Storage Directives

8.4.1 DC - Define Constant

The define constant directive is used to store strings and lists of constants in memory. The format of a DC directive is

```
<label> DC.<size>    <item>,<item>,...
```

The label will be defined to equal the address of the start of the list of data. The size code specifies that a list of bytes (.B), words (.W), or longwords (.L) is being defined; if omitted, word size is used.

A list of items follows the directive; each item may be an expression or a string. If an item is an expression, the expression is evaluated and stored as the size indicated, i.e., a byte, a word, or a longword. An error is generated if the value will not fit as either a signed or unsigned value in the specified size. If an item is a string, delimited by single quotes, then the string will be stored in successive entities of the size specified; if words or longwords are being generated, and the string does not fit into an whole number of words or longwords, then the string will be padded with zeros at the end to make a whole number of words or longwords. Strings and expressions may intermixed in a single DC directive.

If words (DC.W) or longwords (DC.L) are being generated, then the start of the list of constants will be aligned on a word boundary by increasing the location counter by one, if necessary. This is not performed for DC.B directives, so that strings of bytes may be contiguous in memory. If an instruction follows a DC.B directive, the assembler will automatically adjust the location counter (if necessary) to place the instruction on a word boundary.

An example of a DC directive that defines a null-terminated string:

TEXT DC.B 'DC Example',$0D,$0A,0

This directive results in the following data at location TEXT:

44 43 20 45 78 61 6D 70 6C 65 0D 0A 00 (hexadecimal)

8.4.2 DCB - Define Constant Block

The define constant block directive generates a block of bytes, words, or longwords that are all initialized to the same value by the assembler. The format of the directive is

```
<label> DCB.<size>   <length>,<value>
```

The label will be defined to equal the address of the start of the block. The size code specifies that a block of bytes (.B), words (.W), or longwords (.L) is being set up; if omitted, word size is used.

The length argument is an expression that tells the number of bytes, words, or longwords that are to be in the block. This value must be known on the first pass at the point where the DCB directive appears, and it must be non-negative. The value argu ment is an expression whose value is to be placed in each data item in the block; it needn't be known on the first pass. An warning message will be printed if the value will not fit (as a signed or unsigned number) in the data size selected.

If word or longword size is selected, then the start of the block will be placed on a word boundary by increasing the location counter by one, if necessary. If an instruction follows a

DCB.B directive, the assembler will automatically adjust the location counter (if necessary) to place the instruction on a word boundary.

8.4.3 DS - Define Storage

The define storage directive generates an uninitialized block of bytes, words, or longwords. The format of the directive is

```
<label> DS.<size>    <length>
```

The label will be defined to equal the address of the start of the block. The size code specifies that a block of bytes (.B), words (.W), or longwords (.L) is being set up; if omitted, word size is used.

The length argument is an expression that tells the number of bytes, words, or longwords that are to be in the block. This value must be known on the first pass at the point where the DCB directive appears, and it must be non-negative. The effect of the DS directive is basically to increase the value of the location counter by <length> times one (if DS.B is used), two (if DS.W is used), or four (if DS.L is used)

If word or longword size is selected, then the start of the block will be placed on a word boundary by increasing the location counter by one, if necessary. Thus, DS.W 0 can be used to force the location counter to be aligned on a word boundary without allocating any space. However, if an instruction follows a DS.B directive, the assembler will automatically adjust the location counter (if necessary) to align the instruction on a word boundary.

8.5 END - End of Source File

The end directive is used to mark the end of the source file. It is purely optional. The format is simply

```
END
```

The assembler will ignore anything in the source file after the END directive.

9. Usage

9.1 Command Line

The 68000 Assembler is run by typing a command line of the following form:

```
ASM <options> <filename>
```

The options are a string of letters, preceded by a dash, which alter the behavior of the assembler. The following option letters are allowed:

C = Show all the Constants produced by DC directives(see Section 5.2)

L = Produce a Listing file

N = Produce No object code file

If these options are not specified, the defaults are to show only one line of data from a DC directive, to produce no listing, and to produce an object file.

The filename is the name, including directory specifica tions, of the file to be assembled. No default file extension is applied. The names of the listing and object code files, if generated, are constructed by using the source file name with an extension of ".LIS" (for the listing) or ".H68" (for the object file); the output files are always placed in the user's default directory.

The program will print "68000 Assembler by PGM" when it begins work. If any errors occur, the program will print "ERROR in line X" or "WARNING in line X" (this information is also placed in the listing file). Upon conclusion, it will print the number of errors encountered and warnings generated during the assembly.

If there is an error in the command line, e.g., if no file name is specified, then the assembler will print a brief usage summary and abort.

10. Listing File Format

The assembler produces a listing file which shows the source code alongside the the object code produced by the assembler. A typical listing file line might appear as follows (not to scale):

```
0000102E  22D8   200  LOOP  MOVE.L (A0)+,(A1)+    Sample
```

The eight digit hexadecimal number at the left is the assembler's location counter; the generated instruction word, $22D8, is placed at that address. The next number is the source file line number, and the remainder of the line simply repeats the source line. Remember that if the source lines are no longer than 80 columns, then the listing file lines will not exceed 132 columns.

If an error is encountered or a warning is generated for a given source line, then in the listing that line will be followed by a line that describes the error. At the end of the listing, the program prints the total number of errors and warnings.

There is only limited space to list the object code in this format. There is sufficient space for the longest possible instruction, but the DC directive poses a problem, since it may generate, e.g., dozens of longwords from a single source line. The assembler's -C command line option controls the assembler's actions when the object code exceeds the space available on one line. If -C is not specified, then the assembler will print only one listing line with an ellipsis ("...") at the end of the object code field, indicating that some of the data produced by the directive was omitted from the listing. If -C is included on the command line, then the assembler will use as many source lines as are needed to print all the data produced by the directive. Each line after the first will contain only the location counter and the object code field (the source line is not repeated).

11. Object Code File Format

The 68000 Assembler produces an object code output file in S-record format. The object file name is the source file name, with the extension changed to ".H68". The object file and the listing file are always placed in the user's default directory.

The S-record format is designed to allow files containing any data to be interchanged in text file format with checksum error detection. The format of these files will not be described here, but the following technical information will be provided: The first line of the object file is an S0 (header) record and the last line is an S9 (termination) record. The lines in be tween are S1, S2, or S3 records, depending on the whether the address of the first byte in a record requires 2, 3, or 4 bytes to be represented. No record is longer than 80 characters.