PSI 2432 - Projeto e Implementação de Filtros Digitais

Programação em Assembly no ADSP-2189M

Vítor H. Nascimento

17 de julho de 2007

1 Introdução

A maneira de programar em Assembly é um pouco diferente da usual em linguagens de alto nível. Um objetivo usual de compiladores para C, C++, Fortran90 ou outras linguagens de alto nível é deixar a tarefa de programação independente do processador que será usado. Assim, para programar em C, por exemplo, não é necessário saber muito sobre a arquitetura do processador em que o programa rodará de fato. A maior parte dos programas não precisa de muitas alterações para rodar em um PC Intel, em uma estação rodando Unix com processador RISC, ou em um Macintosh.

Em Assembly a situação é bem diferente. O objetivo é chegar a um montador simples, e que permita ao máximo que o programador use recursos avançados do processador escolhido, a fim de economizar memória e tempo de execução do programa. Hoje em dia estão aparecendo compiladores otimizados que podem construir programas em linguagem de máquina razoavelmente rápidos e econômicos (em termos de memória). No entanto, em diversas aplicações é necessário que um programador experiente trabalhe diretamente em Assembly, para aproveitar ao máximo os recursos disponíveis no sistema em uso. Em muitos casos um programa bem-feito pode reduzir consideravelmente o custo de um produto. Além disso, em aplicações em processamento de sinais costuma ser necessário ter-se uma boa idéia do tempo que uma rotina leva para ser executada, por exemplo para verificar quantas instruções podem ser executadas entre dois instantes de amostragem.

Esta apostila explica com algum detalhe alguns conceitos úteis para programação em Assembly, a fim de facilitar o trabalho com as placas EZKIT-2189M doadas pela Analog Devices. Na apostila estão descritas a estrutura do processador ADSP-2819M, as principais instruções para programação desse processador, alguns aspectos de aritmética de ponto fixo e informações sobre a interface do processador com o CODEC (conversor A/D e D/A). O objetivo da apostila é fornecer informações suficientes para que o aluno possa implementar os algoritmos mais simples e usuais de processamento digital de sinais, em especial, filtros recursivos e não-recursivos. No caso de projetos mais complexos, serão necessários mais detalhes sobre programação e o *hardware* dos processadores ADSP-218x, que podem ser encontrados nos diversos manuais preparados pela Analog Devices (veja a seção 13).

Vamos começar agora a ver a programação no ADSP-2189M, começando com os formatos numéricos, e em seguida com uma descrição rápida da arquitetura do processador, do ponto

de vista do programador. Depois descreveremos o mapa de memória do kit utilizado, o vetor de interrupções e a inicialização de um programa, e em seguida a estrutura básica de um programa em Assembly, com as instruções de posicionamento de partes do código na memória, e as principais instruções de controle da unidade lógico-aritmética (ULA), do multiplicador-acumulador (MAC) e do controle de programa. Na última parte serão tratados a comunicação do DSP com o codec, e alguns problemas do kit.

2 Formatos numéricos

O ADSP-2189M trabalha diretamente com quatro tipos números, todos de 16 bits:

- Inteiros com sinal, em notação complemento de dois ($-32768 \leq N \leq 32767$);
- Inteiros sem sinal ($0 \leq N \leq 65535$);
- Números fracionários com sinal, em notação complemento de dois ($-1 \leq n \leq 1-2^{-15}$);
- Números fracionários sem sinal ($0 \leq n \leq 2-2^{-15}$).

2.1 A posição da vírgula — Números inteiros ou fracionários

Precisamos pensar um pouco sobre como um número é armazenado na memória de um processador. Como foi dito acima, o ADSP-2189M trabalha com dados de 16 bits. Considere por exemplo que o registrador AX0 do processador esteja carregado com

$$\mathsf{AX0} \leftarrow (0101\ 0000\ 1000\ 0000)_b \quad (= 0\text{x}5080)$$

(vamos usar a notação 0x5080 para referenciar um número em hexadecimal). A qual número decimal corresponde o valor carregado em AX0?

A resposta mais comum a essa pergunta é considerar o número inteiro, ou seja,

$$0\text{x}5080 \leftrightarrow 5 \times 16^3 + 8 \times 16^1 = 2^{14} + 2^{12} + 2^7 = 20.608.$$

No entanto, isso é verdade apenas se convencionarmos de antemão que o número armazenado em AX0 é inteiro, pois a casa decimal poderia ter sido colocada em qualquer lugar. Por exemplo, poderíamos interpretar o valor armazenado em AX0 como 0,101 0000 1000 0000, e nesse caso (cuidado com a multiplicação por dois na transformação direta de hexadecimal para decimal: note que escolhemos 0,101..., não 0,0101...!)

$$0\text{x}5080 \leftrightarrow 2\big(5 \times 16^{-1} + 8 \times 16^{-3}\big) = 2^{-1} + 2^{-3} + 2^{-8} = 0{,}62890625.$$

Note que a posição da vírgula não é armazenada fisicamente de maneira alguma. Se você ler uma posição de memória de um processador, lerá sempre algo como 0101 0000 1000 0000. Você é que interpreta esse código como 20.608 ou como 0,62890625, dependendo de onde você convenciona que está a vírgula.

Para fazer somas e subtrações, o lugar onde está a vírgula é irrelevante:

Considerando números inteiros,

$$\begin{array}{r} 0101\ 0000\ 1000\ 0000 \\ \underline{+0000\ 0000\ 0010\ 0000} \\ 0101\ 0000\ 1010\ 0000 \end{array} \tag{1}$$

ou, em decimal e hexadecimal,

$$20.640 = 20.608 + 32 \leftrightarrow 0\text{x}5080 + 0\text{x}0020 = 0\text{x}50\text{A}0 \leftrightarrow 5 \times 16^3 + 10 \times 16^1,$$

Para números fracionários a operação realizada com os dados na memória é exatamente a mesma (1). Apenas a interpretação muda:

$$0{,}6298828125 = 0{,}62890625 + 2^{-10} \leftrightarrow 0\text{x}5080 + 0\text{x}0020 = 0\text{x}50\text{A}0 \leftrightarrow 2\big(5 \times 16^{-1} + 10 \times 16^{-3}\big).$$

Ou seja, se quisermos fazer apenas operações de somas e subtrações, podemos convencionar de antemão que a vírgula está entre quaisquer dois bits do número armazenado na memória do processador, e o resultado sempre estará correto, satisfazendo à mesma convenção.

Para multiplicações, no entanto, o caso é um pouco mais complicado. Vamos fazer primeiro um exemplo com números decimais para ser mais fácil entender o problema. Suponha que estejamos trabalhando com números de 4 dígitos, inteiros. Assim, por exemplo, para multiplicar 1234 por 0021, obtemos

$$\begin{array}{r} 0234. \\ \underline{\times 0021,} \\ 00000234, \\ 00004680, \\ 00000000, \\ \underline{+00000000,} \\ 0000\boxed{4914,} \end{array}$$

A caixa marca os quatro dígitos permitidos no resultado (no caso, não houve estouro nem foi necessário fazer nenhum arredondamento).

No entanto, se convencionarmos que os números têm duas casas antes e duas depois da vírgula, a conta realizada seria

$$\begin{array}{r} 02{,}34 \\ \underline{\times 00{,}21} \\ 0000{,}0234 \\ \underline{+0000{,}4680} \\ 00\boxed{00{,}49}14 \end{array}$$

Como o resultado deve estar em ponto fixo, *no mesmo formato dos dados iniciais*, depois de fazer as mesmas operações do caso anterior, foi necessário deslocar o resultado duas

casas para a esquerda (os dois últimos dígitos têm de ser desprezados, arredondando-se o resultado). Observe então, que, para números inteiros de quatro algarismos em ponto fixo, os dados armazenados na memória de um computador que trabalhasse diretamente com números em decimal seriam tais que $0021 \times 0234 = 4914$, enquanto que para números decimais com quatro algarismos e duas casas decimais em ponto fixo, os mesmos dados e a mesma operação resultariam $0021 \times 0234 = 0049$.

Esse resultado mostra como a posição da vírgula afeta o resultado de uma multiplicação. Uma operação de multiplicação com duas parcelas de n algarismos (*algarismos* podem ser bits ou dígitos) tem em geral como resultado um número com $2n$ algarismos. Dependendo da posição da vírgula, varia o alinhamento para retirar o resultado *no mesmo formato das parcelas*.

No caso de números binários com 4 bits (para simplificar), se quisermos multiplicar os números 0101 por 0011, teríamos

1. Para números inteiros:

$$\begin{array}{r} 0101 \\ \times 0011 \\ \hline 0000\ 0101 \\ +0000\ 1010 \\ \hline 0000\ \boxed{1111} \end{array}$$

   ou seja, $5 \times 3 = 15 = 2^3 + 2^2 + 2^1 + 2^0$.

2. Para números fracionários, por outro lado,

$$\begin{array}{r} 0{,}0101 \\ 0{,}0011 \\ \hline 0{,}00000101 \\ 0{,}00001010 \\ \hline 0{,}\boxed{0000}1111 \end{array}$$

   ou seja, $\left(5 \times 2^{-4}\right) \cdot \left(3 \times 2^{-4}\right) = 15 \times 2^{-8}$. Arredondando, o resultado final seria 2^{-4}, ou, em binário, 0,0001.

O resultado importante desta seção é que, para se multiplicar dois números em ponto fixo com n bits cada um, precisa-se saber de antemão a posição (convencionada) da vírgula, para que os n bits relevantes do resultado (de $2n$ bits) sejam armazenados corretamente no registrador de resultado. O ADSP-2189M permite que se façam multiplicações com números inteiros ou com números fracionários, nos formatos

1. Números inteiros: formato 16.0 (a vírgula está à direita de todos os 16 bits),
2. Números fracionários: formato 1.15 (a vírgula está logo à direita do bit mais significativo).

Assim, no formato 16.0, o código 0x5080 corresponde a 20.608,0, e no formato 1.15, o mesmo código corresponde ao número 0,62890625, como vimos anteriormente.

As únicas instruções em que se deve tomar cuidado para saber com qual formato se está trabalhando são as instruções relacionadas a multiplicações (e a transmissão e recepção de dados dos conversores D/A e A/D, que sempre assumem que está sendo usado o formato 1.15). A escolha entre um modo e outro é feita através do valor de um bit no registrador MSTAT. A instrução[1]

```
DIS M_MODE;
```

faz com que, em todas as operações de multiplicação seguintes, o processador interprete as parcelas como estando no modo 1.15. Usando a instrução

```
ENA M_MODE;
```

o processador interpreta as parcelas de uma multiplicação como números inteiros no modo 16.0. Note que qualquer uma das instruções acima permanece válida para todas as multiplicações posteriores, até que seja explicitamente dada a instrução contrária (ou que o processador seja reinicializado, caso em que o modo 1.15 é escolhido por padrão).

3 Memória

O ADSP-2189M tem dois bancos de memória, que podem ser acessados simultaneamente: a *memória de dados* e a *memória de programa*. A memória de dados tem palavras de 16 bits (o mesmo número que os registradores internos, e que a saída do conversor A/D presente na placa do curso, o AD 73322). A memória de programa, por outro lado, tem palavras de 24 bits (como algumas instruções do processador podem conter constantes — dados — para serem usadas em alguma operação, o tamanho da palavra na memória de programa deve ter mais de 16 bits).

Como veremos mais adiante, é possível ler uma posição da memória de dados ao mesmo tempo que uma posição da memória de programa. Isso é útil ao se implementar uma operação de convolução (para implementar um filtro não-recursivo, por exemplo), em que é necessário somar produtos dos elementos de dois vetores. Um vetor é normalmente fixo (os coeficientes do filtro), e é armazenado na memória de programa. O outro vetor, com a entrada do filtro, vai sendo armazenado na memória de dados. Como é possível ler a memória de dados e a memória de programa ao mesmo tempo, as leituras de um coeficiente do filtro e de um valor da entrada podem ser feitas simultaneamente, fazendo o filtro operar mais rapidamente.

Note, no entanto, que, como os registradores internos para operações aritmétricas têm apenas 16 bits, não adianta tentar armazenar os coeficientes com 24 bits — ao passar o valor do coeficiente da memória de programa para um registrador de trabalho, só os primeiros 16 bits serão armazenados. Portanto, ao se calcular o valor dos coeficientes, deve-se fazer o arredondamento para 16 bits, mesmo que seja usada a memória de programa.

[1] O Assembler dos processadores 218x não diferencia letras maiúsculas ou minúsculas para comandos ou nomes de registradores. Assim, o comando deste exemplo poderia ser escrito também `ena m_mode;`. Repare que *ena* é abreviação de *enable*, e *dis*, de *disable*.

4 Principais instruções de controle do Assembly

O programa *VisualDSP++* é um ambiente para o desenvolvimento de programas para diversos processadores da Analog Devices. As placas em que esses processadores estão montados podem ser os kits de desenvolvimento da própria Analog Devices, ou podem ser placas construídas pelo usuário ou por terceiros. Portanto, para poder escrever um programa que possa rodar corretamente em uma determinada placa, o *VisualDSP++* precisa das seguintes informações:

1. Descrição da memória disponível no processador e na placa (faixas de endereços): arquivo do *linker*, `xxx.ldf`;
2. Descrição do programa propriamente dito: arquivo de programa, `xxx.dsp`;
3. Arquivos extras, como definições de endereços de dispositivos: arquivos de cabeçalho (*header*), `xxx.h`.

4.1 Arquivo de descrição da placa

No caso do kit disponível nesta disciplina, a descrição da memória e outras informações necessárias para a montagem do programa que realmente vai ser carregado no DSP estão armazenadas no arquivo `2189ezkitmod.ldf`.[2] Cada banco de memória tem algumas posições reservadas, marcadas como *segments* no arquivo de descrição. Alguns segmentos têm funções específicas, as principais são descritas a seguir.

1. Memória de programa (24 bits por palavra)
   (a) Segmento `seg_rth` (posições de memória: 0x00000–0x0002F — 48 palavras): contém a tabela de vetores de interrupções do processador (veja a Seção 4.4).
   (b) Segmento `seg_code` (posições de memória: 0x00030–0x01840 — 6160 palavras): região onde deve ser colocado o código do programa propriamente dito.
   (c) Segmento `pm_data` (posições de memória: 0x02000–0x02600 — 1537 palavras): região da memória de programa onde são colocadas constantes, dados e variáveis em geral.
   (d) Além dos segmentos acima, a região 0x01940–0x01FFF é reservada para o programa monitor, que permite a comunicação entre a placa de DSP e o computador.
   (e) A região 0x02000–0x03FFF pode ser usada para acessar até 3 páginas diferentes de memória, usando o registrador PMOVLAY.
2. Memória de dados (16 bits por palavra)
   (a) Segmento `seg_input` (posições de memória: 0x00000–0x01FFF — 8.192 palavras): região onde são colocados os dados de trabalho do programa principal. O programa monitor parece usar esta parte da memória, evite usá-la.

[2] Esse arquivo foi modificado para permitir a utilização de uma parte maior da memória de dados.

(b) Segmento `seg_novl` (posições de memória: 0x02000–0x036AF — 5808 palavras): região que pode ser usada para armazenamento de dados.

(c) Segmento `seg_init` (posições de memória: 0x036B0–0x036BF — 16 palavras): região onde são armazenadas constantes para inicialização do CODEC.

(d) Segmento `seg_page` (posições de memória: 0x036C0–0x036CF — 16 palavras): armazenam informações para trabalhar com *overlays* de memória (permitem o acesso a um banco de memórias maior do que a capacidade de endereçamento direto do processador).

(e) Segmento `seg_data` (posições de memória: 0x36D0–0x03C00 — 1.329 palavras): outra região para armazenamento de dados gerais.

(f) Além dos segmentos acima, a região 0x03C80–0x03FE0 está reservada para dados do programa monitor.

(g) Na região 0x00000–0x01FFFF podem ser mapeadas várias páginas diferentes de memória, usando o registrador DMOVLAY.

O arquivo do *linker* não precisa ser modificado — deve apenas ser incluído em um projeto para que os programas escritos sejam montados corretamente.

4.2 Arquivos de cabeçalho

Nos arquivos de cabeçalho (`xxx.h`) são incluídas definições que podem ser úteis para vários programas diferentes (são semelhantes aos arquivos .h de compiladores C). O arquivo

`constant.h`

contém definições de endereços de dispositivos da placa EZKIT 2189M, como o endereço para acesso às portas seriais e ao temporizador. Também não deve ser alterado, e deve ser incluído em todos os projetos.

4.3 Arquivos do programa principal

Em princípio um programa pode ser separado em diversos arquivos, conforme a conveniência e o tamanho do projeto em desenvolvimento (novamente, como em compiladores C usuais). A extensão padrão para arquivos de programa é `.dsp`.

No arquivo de programa são colocados os componentes usuais: alocação de memória para variáveis, o programa propriamente dito, e diretivas de compilação (ou, no caso, de montagem). As diretivas têm um pouco mais de opções do que o usual em linguagens de alto nível: para programas em linguagem C que devem funcionar em vários tipos de processador diferentes, há diretivas para definição de constantes, para inclusão de arquivos externos, e de compilação condicional. Já no Assembly há também instruções para alocação de memória e escolha de qual banco de memórias e qual segmento devem ser usados para cada variável ou parte do código. Vejamos a seguir as principais instruções:

4.3.1 `#define`

Permite definir uma constante para ser usada ao longo do programa, por exemplo,

```
#define N 10 // número de coeficientes do filtro FIR
```

O símbolo `N` será simplesmente substituído pelo número 10 toda vez que aparecer no arquivo.

4.3.2 `#include`

Permite incluir um outro arquivo na posição do comando,

```
#include "constant.h";
```

4.3.3 `.global`

Informa que um símbolo, definido no mesmo arquivo em que está a declaração `.global`, pode ser usado em outros arquivos.

```
.global codecinit; // Rotina para inicializar o Codec
```

4.3.4 `.extern`

Informa que o símbolo em questão é definido em outro arquivo.

```
.extern codecinit;
```

4.3.5 `.section`

Define o banco e o segmento da memória em que o código ou as variáveis definidas a seguir devem ser colocados. Por exemplo, os comandos a seguir alocam um vetor `x` de comprimento `N` e uma variável `y` na memória de dados, no segmento `seg_input`.

```
.section/data seg_input;
.var x[N];
.var y;
```

Para usar o banco da memória de programa, use

```
.section/pm seg_code;
MR = 0;
```

No exemplo acima a instrução para carregar o registrador MR do MAC com o valor 0 foi colocada na primeira posição disponível do segmento `seg_code` da memória de programa.

4.3.6 `.var`

Aloca memória. Há duas formas principais: a primeira é como no exemplo anterior, em que se definem vetores comuns. A segunda, indicada no exemplo abaixo, define um vetor *circular* de comprimento dado. Vetores circulares (veja a seção 5.1) são especialmente úteis para o cálculo de produtos escalares de dois vetores (e produtos escalares são usados para a implementação de filtros recursivos e não-recursivos). Um vetor circular é definido pela instrução

```
.var/circ z[N];
```

Observe que o nome `x`, `y` ou `z` de uma variável apenas guarda o endereço do primeiro elemento do vetor alocado. Por exemplo, no caso da variável `x`, se o código do exemplo acima for a primeira referência ao segmento `seg_input`, o símbolo `x` vai representar o endereço 0x00000. Se `N`=3, o primeiro comando do exemplo aloca as posições de memória 0x00000–0x00002, e portanto, o símbolo `y` vai representar o endereço 0x00003.

As variáveis podem, se desejado, ser inicializadas diretamente no comando de alocação de memória, ou através de um arquivo de dados:

```
.section/data seg_input;
.var h[3]={ 0x4000,0x2000,0xC000 };
.var g[3]="coeficientes.dat";
.var w[]={ 0.5r,0.25r,-0.5r };
```

Neste exemplo, `h` e `w` têm os mesmos valores: colocando um `r` ao final de um número na faixa $-1 \leq h < 1$, o Assembler traduz o valor para a notação 1.15 (cuidado: o Assembler apenas trunca o valor, não arredonda para o número mais próximo no formato 1.15). Note que no caso de `w`, o Assembler determina automaticamente o comprimento do vetor alocado, pela lista de dados fornecida.

A variável `g` é carregada com os valores armazenados no arquivo externo `coeficientes.dat`, que deve ter a forma (novamente, os números são os mesmos das variáveis `h` e `w`)

```
0x4000
0.25r
b#1100000000000000
```

(números em binário podem ser incluídos diretamente, se for acrescentado o código `b#` na frente do número).

Também é possível realizar operações com constantes pré-definidas para a inicialização de variáveis, por exemplo (note que só é possível fazer operações ou entre inteiros ou entre números fracionários, misturar os dois tipos resulta em erro)

```
#define N 3
#define pi_sobre_10 0.314159r
.section/data seg_init;
.var/circ b[N]="numerador.dat";
.var/circ a[N-1]="denominador.dat";
.var wp=0.5r-pi_sobre_10;
.var wr=0.5r+pi_sobre_10;
```

Para definir variáveis na memória de programa, se as variáveis forem ser usadas para realizar contas diretamente no MAC ou na ULA, terão de ser carregadas nos registradores de trabalho dessas unidades, e só serão usados 16 bits. Para isso, pode-se usar os mesmos comandos que antes:

```
.section/pm pm_data;
.var/circ b[]="0.5r,0x2000,b#0001000000000000";
```

Se, por algum motivo, for necessário usar todos os 24 bits da memória de programa, a inicialização deve ser feita com a diretiva `.var/init24`:

```
.section/pm pm_data;
.var/circ/init24 c1[]="0x123456,0x400000";
.var/init24 dado="0x200000";
```

Note que quando a opção `init24` for usada, devem ser fornecidos realmente 24 bits, ou em hexadecimal, seis algarismos. Definir

```
.var/init24 c2[]="0x4000,0.25r";
```

não resulta no esperado: com o `init24`, o alinhamento dos bits não informados é feito para a direita, ou seja, a operação acima é equivalente a fazer

```
.var/init24 c2[]="0x004000,0x002000";
```

Pequenos problemas para evitar:

1. O valor 0 deve ser escrito `0.0r`, não `0r`;
2. A notação fracionária (como em `0.5r`) só pode ser usada na inicialização da memória (antes de começar o programa propriamente dito).

4.4 A tabela de vetores de interrupção

O segmento `seg_rth` da memória de programa contém a *tabela de vetores de interrupção* do processador: são posições da memória para onde o processador vai caso chegue um comando especial (como uma interrupção ou após uma reinicialização). A tabela de interrupções do ADSP-218x tem 12 linhas com 4 palavras cada uma. Cada linha corresponde a uma situação de exceção diferente, por exemplo, a primeira linha (palavras 0x00000 a 0x00003) são as instruções que devem ser seguidas quando o processador começa a rodar após ser reinicializado. Já a 6ª linha (palavras 0x00014–0x00017) corresponde a uma interrupção da porta serial SPORT0, que, no caso do EZKIT 2189M, indica que uma nova amostra do conversor A/D está pronta para ser lida.

A tabela de interrupções completa, no caso da maioria dos programas que serão feitos para esta disciplina, é programada como abaixo:

```
.section/pm seg_rth;
  JUMP inicio; NOP; NOP; NOP;/* Reinicialização                    0x00 */
  RTI; NOP; NOP; NOP;        /* Não usado na placa      IRQ2       0x04 */
  RTI; NOP; NOP; NOP;        /* (não usar)              IRQ1L      0x08 */
  RTI; NOP; NOP; NOP;        /*                         IRQL0      0x0C */
  JUMP T0_tx; NOP; NOP; NOP;/* Configuração do CODEC    SPORT0_TX  0x10 */
  JUMP T0_rx; NOP; NOP; NOP;/* Recebe e envia amostras  SPORT0_RX  0x14 */
  RTI; NOP; NOP; NOP;        /* Botão "interrupt"       IRQE       0x18 */
  RTI; NOP; NOP; NOP;        /*                         BDMA       0x1C */
  RTI; NOP; NOP; NOP;        /*                         SPORT1 TX  0x20 */
```

```
RTI; NOP; NOP; NOP;       /*                          SPORT1 RX 0x24 */
NOP; NOP; NOP; NOP;       /* Usado pelo monitor      TIMER     0x28 */
/* a primeira interrupção do timer não pode ser um RTI */
RTI; NOP; NOP; NOP;       /*                          POWERDOWN 0x2C */
```

Cada vetor de interrupção da tabela contém instruções para tratar um certo tipo de situação especial. Por exemplo, se for enviada uma interrupção `IRQE` (na placa do EZ-KIT, o botão `INTERRUPT` está ligado à interrupção `IRQE`), o processador pula para a posição 0x0x00018 da memória de programa, ou seja, o endereço reservado na tabela de vetores de interrupção para tratamento da `IRQE`. As quatro instruções da linha correspondente na tabela, ou seja, 0x00018, 0x00019, 0x0001A e 0x0001B, podem ser usadas para tratar o evento relacionado à interrupção `IRQE`. Se não se quiser dar nenhum uso a essa interrupção, o mais seguro é colocar logo no primeiro elemento do vetor de interrupção correspondente o comando `RTI` (retorno de rotina de tratamento de interrupção). Para usar o botão `INTERRUPT`, basta colocar alguma instrução em 0x00018. Por exemplo, se a linha correspondente à interrupção `IRQE` for alterada para

```
toggle FL1; RTI; NOP; NOP;
```

o led marcado *FL1* na placa trocará de estado a cada vez que o botão `INTERRUPT` for pressionado.

Apenas quatro instruções podem não ser suficientes para tratar corretamente o evento que deu origem a uma interrupção. Nesse caso, basta colocar logo na primeira instrução do vetor correspondente uma instrução `JUMP`, para que o processador pule para uma outra posição da memória de programa em que se tratará a interrupção. Na tabela de vetores de interrupção mostrada acima, isso é feito para direcionar o processador para o programa principal após uma reinicialização (`JUMP inicio;` logo na primeira linha), e para tratar as interrupções de comunicação com o CODEC (`JUMP T0_tx;` e `JUMP T0_rx;`). Os nomes `inicio`, `T0_tx` e `T0_rx` são etiquetas de posição (*labels*), e devem ser definidos em alguma posição do arquivo de programa. Veremos mais sobre isso na Seção 7.

As instruções `NOP` (*no operation*) não fazem nada, o processador apenas pula um ciclo de máquina sem fazer nenhuma operação. Uma instrução `NOP` é utilizada ou para esperar algum tempo, por exemplo, para esperar que algum resultado fique pronto antes de ser usado, ou, como aqui, para preencher uma posição que não se precisa usar da memória de programa com uma instrução não perigosa e que não gaste muita energia.

5 Principais registradores de trabalho e instruções para leitura e escrita na memória

O ADSP-2189 tem três unidades de trabalho: a ULA (unidade lógica-aritmética), o MAC (multiplicador-acumulador) e o deslocador. Cada uma dessas unidades é utilizada apenas através de determinados registradores dedicados, como veremos a seguir (veja a figura 1).

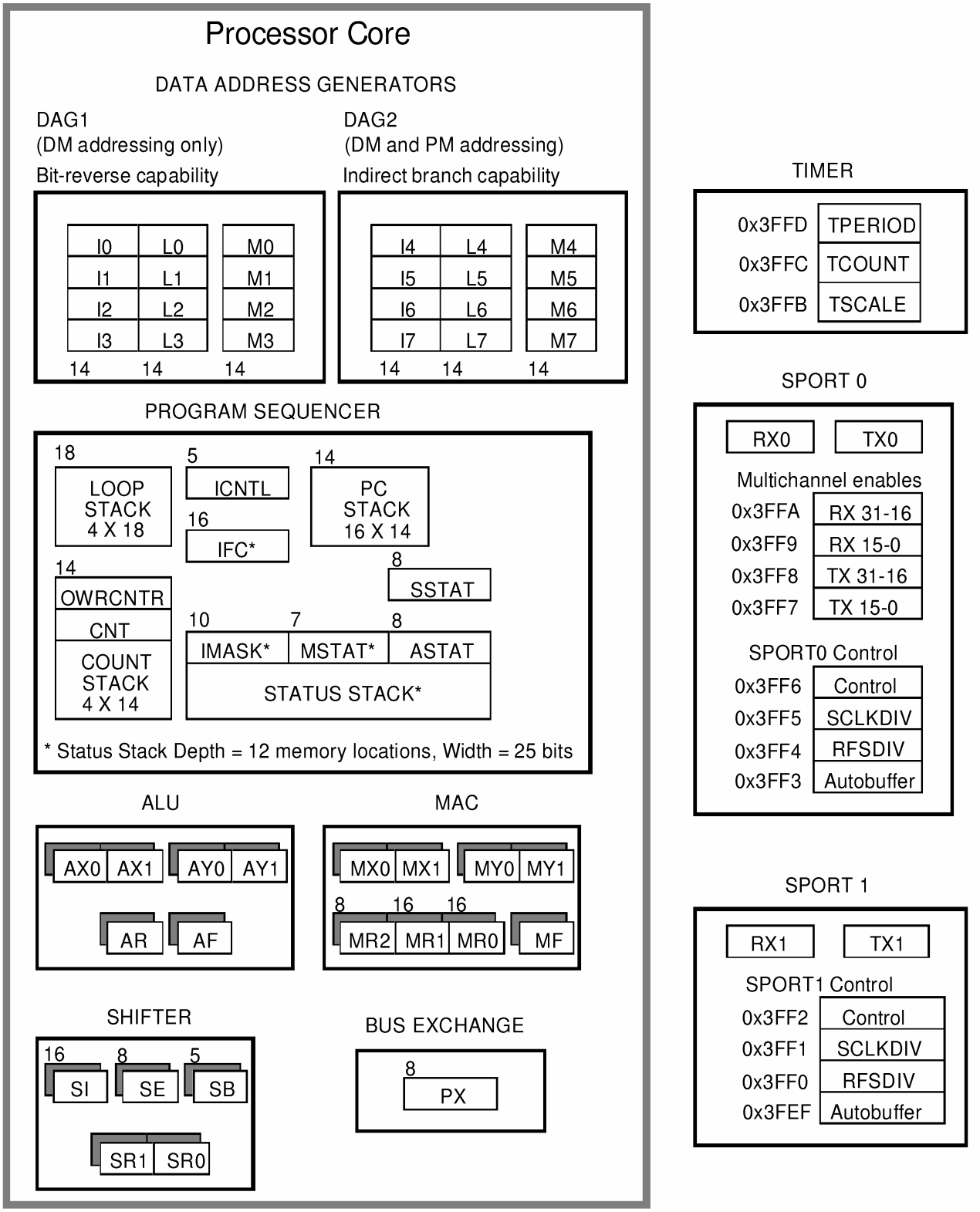

Figura 1: Registradores e unidades lógicas dos processadores da família ADSP-218x (retirado do manual do conjunto de instruções da *Analog Devices*).

5.1 Endereçamento de memória

As operações de leitura e escrita nas memórias de dados e de programa são realizadas através de registradores conhecidos como *data address generators* (DAGs), que funcionam de maneira semelhante a ponteiros em C. Há dois conjuntos de DAGs:

1. DAG1 (I0–I3): permite acesso apenas à memória de dados (mas pode acessar a memória de uma maneira especial, conhecida como "bit-reversa", útil para implementação de algoritmos para transformada rápida de Fourier — FFT).

2. DAG2 (I4–I7): permite acesso tanto à memória de dados quanto à memória de programa, mas não tem o modo de acesso bit-reverso.

Os registradores I0 a I7 contêm o endereço da posição de memória que se deseja acessar (por isso a analogia com ponteiros em C). Os comandos para leitura e escrita de memória são apresentados nos exemplos a seguir.

Para acessar uma posição de memória, usa-se o comando `dm(I0,M0)` (para a memória de dados) ou `pm(I4,M4)` (para a memória de programa). Como já vimos, I0 e I4 são apontadores para a posição de memória desejada. Os registradores M0–M7 são registradores de modificação, e indicam de quantas posições o apontador I0 deve ser modificado após uma operação de leitura ou escrita. Por exemplo se I0=0x00000 e M0=1, então a instrução `dm(I0,M0)=0x4000` vai colocar o valor 0x4000 na posição de memória inicialmente apontada por I0, 0x00000, *e avançar* I0 de uma posição. Assim, ao final da operação I0 vai apontar para a posição 0x00001 da memória de dados.

Não é necessário usar apenas M0 com o registrador I0, é permitido usar `dm(I0,M1)`, por exemplo.

Além dos registradores de endereço e de modificação, cada operação de leitura tem um registrador de comprimento associado, L0 a L7. Os registradores de comprimento não aparecem explicitamente nas funções de acesso à memória, pois são ligados aos registradores de endereço de mesmo número: ao contrário do registrador M0, o registrador L0 sempre modifica apenas as operações em que o registrador I0 é usado. O registrador L0 indica se a variável apontada por I0 é um vetor ou um escalar, e, no caso de vetor, o comprimento do vetor (e de maneira similar, o registrador L1 indica o tipo de variável para que o registrador I1 aponta, e assim por diante).

Para entender a utilidade de todos os registradores, vamos escrever agora um programa simples (e não muito elegante) para inicializar o vetor `x`, comparando o que seria feito em Assembly com o equivalente em C. Note que a etiqueta `inicio:` foi definida na posição em que o programa principal deve começar: é para esse ponto que a instrução `JUMP inicio;` da tabela de vetores de interrupção leva o processador após uma reinicialização.

Assembly	C `float *x;`	Valor **final** de I0	Vetor `x`
`.section/data seg_input;`			
`.var/circ x[3];`	`x=malloc(3);`	I0=0x????	0x????, 0x????, 0x????
`.section/pm pm_code;`			
`inicio:`			
`I0 = x;`		I0=0x0000	0x????, 0x????, 0x????
`M0 = 1;`		I0=0x00000	0x????, 0x????, 0x????
`L0 = 3;`		I0=0x00000	0x????, 0x????, 0x????
`DM(I0,M0) = 0x4000;`	`*x=0.5;` `x++;`	I0=0x00001	0x4000, 0x????, 0x????
`DM(I0,M0) = 0x2000;`	`*x=0.25;` `x++;`	I0=0x00002	0x4000, 0x2000, 0x????
`DM(I0,M0) = 0x1000;`	`*x=0.125;` `x = x-2;`	I0=0x00000	0x4000, 0x2000, 0x1000

A variável `x` foi definida como um vetor *circular* de três elementos, e o registrador I0 foi usado para apontar para o primeiro elemento de `x`. Como M0 está carregado com o valor 1, a cada operação de leitura I0 avança uma posição. No entanto, como `x` é um vetor circular, e L0 está carregado com o valor correto do comprimento do vetor, quando I0 chegar ao último elemento de `x`, a operação de incremento faz I0 retornar ao valor inicial[3].

É possível acessar uma posição da memória de programa ao mesmo tempo que uma posição da memória de dados. Assim, se definirmos

```
.section/data seg_input;
.var/circ x[3] = {0.5r, 0.25r, 0.125r};
.var y;

.section/pm pm_data;

.var/circ b[] = {0.5r, 0.5r, 0.5r};

.section/pm pm_code;
inicio:
   I0 = x;
   M0 = 1;
   L0 = 3;
   I4 = b;
   M4 = 1;
   L4 = 3;

   MX0 = dm(I0,M0), MY0 = pm(I4,M4);
```

os registradores do MAC MX0 e MY0 serão carregados com os valores 0x4000 simultaneamente.

[3]Se você inicializar um registrador de comprimento L0–L7 com zero, o registrador de acesso à memória correspondente (I0–I7) não retornará automaticamente ao valor inicial.

Há várias instruções que podem ser realizadas ao mesmo tempo (separadas por vírgulas no código-fonte), mas nem todas as combinações de instruções são possíveis. Para garantir que duas operações sejam realizadas uma após a outra, separe-as com um ponto-e-vírgula. Instruções separadas por vírgula serão, caso permitido, executadas simultaneamente. Em caso de dúvida, consulte os manuais do *hardware* e do conjunto de instruções.

Se você definir uma variável como "não circular", alguns cuidados são importantes.

Considere as definições abaixo.

```
.section/data seg_input;
.var X= 0.5r;
.var Z= 0r;
.var Y[3]= {-0.5r,0r,0.5r};
.section/pm pm_code;
inicio:
I0= X; M0= 0; L0= 0;
I1= Y; M1= 1; L1= 0;
AX0= dm(I0,M1);
dm(I0,M1)= AX0;
```

1. Usar `L0` igual a zero significa que temos um "vetor" de comprimento infinito: a instrução `AX0= dm(I0,M1);` iria fazer `I0` apontar para o início da variável `Y`, e a instrução seguinte iria escrever na primeira posição de `Y` o valor `0.5r`. Isso não aconteceria se usássemos as instruções

   ```
   AX0= dm(I0,M0);
   dm(I0,M0)= AX0;
   ```

   pois nesse caso, como o registrador `M0` está carregado com o valor `0`, o registrador `I0` não se modifica depois de cada instrução.

2. Se usássemos `L1=3;` para marcar o fato de que `Y` deve apenas 3 elementos, o programa

   ```
   AY0= dm(I1,M1);
   AY1= dm(I1,M1);
   ```

   iria na verdade copiar o valor `0.5r` (o valor de `X`) em `AY1`, porque o processador erroneamente iria *decrementar* `I1` na primeira instrução, em vez de incrementar.

Para evitar problemas, quando forem usadas variáveis definidas sem o `/circ`:

1. Carregue o registrador L de comprimento adequado com `0` se você quiser usar um valor diferente de `0` no registrador de modificação M,
2. Ou carregue o registrador de comprimento com um valor qualquer, mas use o valor `0` no registrador de modificação.
3. Variáveis definidas com `.var/circ` podem ser alocadas antes de variáveis definidas com `.var` apenas, não importa a ordem das instruções. No programa acima, se `Y` fosse definida como `.var/circ Y[3];`, as posições de memória alocadas para `Y` viriam *antes* das posições alocadas para `X` e `Z`.

5.2 A unidade lógico-aritmética (ULA)

A ULA permite realizar operações lógicas como `E`, `OU` (bit-a-bit) e operações de soma e subtração, além de poder ser usada para testes de condições, como verificar se o resultado de uma operação é maior ou igual a zero, por exemplo.

Os registradores utilizados pela ULA são AX0, AX1, AY0, AY1, AR e AF. Todos os registradores são de 16 bits. Os quatro primeiros são usados como entradas (operandos), e os dois últimos como saídas (mas AR também pode ser usado como entrada para a ULA). O Assembly da Analog Devices é um pouco mais fácil de ler do que o de outros processadores, porque operações aritméticas podem ser escritas da maneira usual, como mostrado no exemplo a seguir. Para somar dois números podemos usar o programa (suponha que I0, M0, L0 estão como no final do exemplo anterior, ou seja, I0 aponta para a segunda posição do vetor `x`, e I4 aponta para a segunda posição do vetor `b`)

```
I1 = y;
M3 = 0;
L1 = 0;
AX0 = dm(I0,M0), AY0 = pm(I4,M4);
AR = AX0 + AY0;
dm(I1,M3) = AR;
```

Assim, a variável `y`, que está armazenada na posição 0x00003 da memória de dados, será carregada com o valor 0.75r. Algumas operações realizadas com a ULA estão listadas na Tabela 1.

O comando `PASS` simplesmente passa o argumento para a saída, mas de maneira a modificar os indicadores de resultado (*flags*). Assim, se você quiser saber se o valor armazenado no registrador MX0 é negativo, por exemplo, você precisa usar o comando `AR = PASS MX0`, e depois fazer o teste.

Todas as instruções acima podem ser executadas condicionalmente, colocando-se um `IF` no início da linha. Por exemplo,

```
AX0 = 0.2r;
AY0 = 0.4r;
AR = AY0 - AX0;
IF NEG AR = -AR;
```

Operação	Comando
Adição	`AR = AX0 + AY0;`
Subtração	`AR = AX0 - AY0;`
Subtração	`AR = AY0 - AX0;`
Negação (compl. de dois)	`AR = -AX0;`
Adição com vai-um	`AR = AR + CI;`
Subtração com empréstimo	`AR = AY0 - AX0 + C - 1;`
Valor absoluto	`AR = ABS AX0;`
Zerar saída	`AR = 0;`
Passar um valor pela ULA (para depois realizar testes)	`AR = PASS AX0;`
E lógico	`AR = AX0 AND AY0;`
OU lógico	`AR = AX0 OR AY0;`
OU exclusivo	`AR = AX0 XOR AY0;`
Negação lógica	`AR = NOT AX0;`
Colocar bit em 1	`AR = SETBIT 3 OF AX0;`
Zerar bit	`AR = CLRBIT 3 OF AX0;`
Alterar valor de bit	`AR = TGLBIT 3 OF AX0;`
Testar valor de bit	`AR = TSTBIT 3 OF AX0;`

Tabela 1: Principais operações realizadas pela ULA

é uma forma longa de calcular o módulo de AY0-AX0.

Nem todos os registradores podem ser usados em qualquer posição na ULA. A Tabela 2 mostra quais registradores podem ser usados como primeiro operando (X), como segundo operando (Y), ou como saída.

Entrada X	Entrada Y	Saída
AX0, AX1	AY0, AY1	AR
AR	AF	AF
MR0, MR1, MR2		
SR0, SR1		

Tabela 2: Registradores usados pela ULA.

Uma operação da ULA também pode não ter saída nenhuma, servindo apenas para modificar os indicadores (*flags*) de resultado. Novamente para testar se MX0 é negativo, se você não quiser modificar o registrador AR, pode simplesmente escrever `PASS MX0` e fazer o teste em seguida.

Uma característica muito útil da ULA dos processadores ADSP-218x é que ela pode verificar se o resultado de uma operação excedeu sua capacidade de representação (foi maior ou igual a 1 ou menor que -1), e saturar o resultado. Isso é importante, pois uma soma (em notação 1.15) `0.5r + 0.5r = 0x4000 + 0x4000 = 0x8000=-1.0r`. Em notação decimal, isso significa que $0{,}5 + 0{,}5 = -1$! (uma *flag* de estouro seria ativada). Com a opção de saturação habilitada, a operação `0.5r + 0.5r` resultaria em `0x7FFF`, que é o maior número

positivo na notação 1.15 (a *flag* de estouro ainda seria ativada).

Para usar a opção de saturação, um bit do registrador MSTAT deve ser colocado em 1, o que pode ser feito com a instrução `ENA AR_SAT`. Para desabilitar essa função, use o comando `DIS AR_SAT`. Note que a saturação só funciona se o registrador de saída for o AR, e não o AF.

6 O multiplicador-acumulador

O componente que define um DSP, ou seja, um processador especial para processamento digital de sinais, é o multiplicador-acumulador (MAC). A operação mais comum em algoritmos para processamento digital de sinais é o produto escalar de dois vetores (o exemplo mais comum é a implementação de um filtro não-recursivo, ou de partes de um filtro recursivo). Por exemplo, para implementar um filtro FIR com função de rede $H(z) = -0{,}5 + 0z^{-1} + 0{,}5z^{-2}$ com entrada $x(n)$ e saída $y(n)$, deve-se fazer a convolução de $x(n)$ com a resposta impulsiva do filtro, $h(n)$, ou seja,

$$\begin{aligned} &\text{Instante } n = 0: && y(0) = -0{,}5x(0) + 0x(-1) + 0{,}5x(-2), \\ &\text{Instante } n = 1: && y(1) = -0{,}5x(1) + 0x(0) + 0{,}5x(-1), \\ &\qquad\vdots \end{aligned}$$

Em notação vetorial, as operações realizadas são

$$y(n) = \begin{bmatrix} -0{,}5 & 0 & 0{,}5 \end{bmatrix} \begin{bmatrix} x(n) \\ x(n-1) \\ x(n-2) \end{bmatrix},$$

o que tem a forma de um produto escalar entre vetores. O MAC é projetado especialmente para realizar esse tipo de operação eficientemente, em poucos ciclos de máquina e com uma precisão elevada.

O MAC do ADSP-2189M implementa o produto escalar com precisão extendida usando seus quatro registradores principais de entrada, MX0, MX1, MY0 e MY1, cada um com 16 bits, e seus três registradores de saída: MR0 e MR1, com 16 bits cada, e MR2, com 8 bits. Em uma operação no MAC, os três registradores de saída são concatenados em um registrador único, MR, formado como MR2 MR1 MR0 (ou seja, MR2 contém os 8 bits mais significativos de MR, MR1 contém os bits intermediários, e MR0 contém os 16 bits menos significativos). O registrador MF também pode ser usado como saída, mas tem apenas 16 bits, e portanto não permite o uso de precisão extendida.

Vamos ver como usar os recursos do MAC para calcular um produto escalar com precisão extendida (Note que na tabela 3, comandos separados por vírgulas são executados no mesmo ciclo de máquina no DSP — estão separados em linhas diferentes apenas por problemas de diagramação).

Note que, como os dois vetores foram definidos como circulares, os registradores I0 e I4 retornaram aos seus valores iniciais após o cálculo do produto escalar. O registrador I1, por outro lado, não é circular, e terminou com o valor 0x0004. Note também que começamos o

Comando	I0	I4	I1	MR2	MR1	MR0	MR (em decimal)
`.section/pm pm_code;`							
`inicio:`							
`DIS M_MODE;`	????	????	????	????	????	????	????
`I0 = X;`	0x00000	????	????	????	????	????	????
`M0 = 1;`	0x00000	????	????	????	????	????	????
`L0 = 3;`	0x00000	????	????	????	????	????	????
`I1 = Z;`	0x00000	????	0x0003	????	????	????	????
`L1 = 0;`	0x00000	????	0x0003	????	????	????	????
`I4 = Y;`	0x00000	0x20000	0x0003	????	????	????	????
`M4 = 1;`	0x00000	0x20000	0x0003	????	????	????	????
`L4 = 3;`	0x00000	0x20000	0x0003	????	????	????	????
`MR=0, MX0=dm(I0,M0), MY0=pm(I4,M4);`	0x00001	0x20001	0x0003	0x00	0x0000	0x0000	0,0
`MR=MR+MX0*MY0(ss), MX0=dm(I0,M0), MY0=pm(I4,M4);`	0x00002	0x20002	0x0003	0x00	0x2000	0x0000	0,25
`MR=MR+MX0*MY0(ss), MX0=dm(I0,M0), MY0=pm(I4,M4);`	0x00000	0x20000	0x0003	0x00	0xA000	0x0000	1,25
`MR=MR+MX0*MY0(rnd);`	0x00000	0x20000	0x0003	0x00	0x6000	0x0000	0,75
`IF MV SAT MR;`	0x00000	0x20000	0x0003	0x00	0x6000	0x0000	0,75
`dm(I1,M0)=MR1;`	0x00000	0x20000	0x0004	0x00	0x6000	0x0000	0,75

Tabela 3: Exemplo de uso do MAC com precisão extendida.

programa com a instrução `DIS M_MODE;`, que configura o MAC para fazer produtos assumindo que as entradas estão no formato 1.15.

Os comandos do MAC têm a forma `MR = MR + MX0 * MY0(ss)`. A indicação entre parênteses no final do comando — `(ss)` — tem a função de dizer se:

1. O resultado deve ser arredondado ou não.

2. As parcelas do produto são números com ou sem sinal.

Para arredondar o resultado, deve-se terminar o comando com (`rnd`). Neste caso, o resultado final (arredondado) da operação terá apenas 16 bits, e estará armazenado no registrador MR1. Em todas as outras opções, o resultado não será arredondado, e ficará disponível em precisão extendida, precisando de todos os registradores que formam MR.

O conjunto completo de opções é:

1. As duas parcelas são números com sinal

   (a) Sem arredondamento (precisão extendida): `(ss)`;

(b) Com arredondamento (resultado com 16 bits no registrador MR1): `(rnd);`

2. A parcela `X` é sem sinal, mas `Y` é com sinal `(us)`;

3. `X` com sinal e `Y` sem sinal `(su)`;

4. Ambas as parcelas sem sinal `(uu)`.

Dessa forma, para se calcular um produto escalar com o MAC, todas as multiplicações e somas deverão ser realizadas com precisão extendida, e só a última operação deverá usar a opção de arredondamento, como feito no exemplo. Como o resultado final pode ter ultrapassado a faixa de valores permitida ($-1 \leq x < 1$), há uma instrução para saturar o resultado final caso necessário (`IF MV SAT MR;`)[4]. Repare que é necessário um comando explícito para que o resultado de uma operação no MAC seja saturado[5], enquanto que na ULA todas as operações realizadas depois de dado o comando `ENA AR_SAT;` terão o resultado saturado se houver estouro, até que seja dado um comando `DIS AR_SAT;`.

Números sem sinal são úteis, pois filtros recursivos de segunda ordem têm coeficientes no denominador na faixa $-2 < a_i < 2$, ou seja, às vezes é necessário um coeficiente com valor maior do que um, mas menor do que dois, exatamente a faixa conseguida com um número sem sinal no formato 1.15, como vimos na Seção 2.1.

As operações realizadas no MAC estão listadas na Tabela 4, e os registradores que podem ser usados como entrada `X` ou `Y`, ou como saída, estão listados na Tabela 5.

Operação	Comando
Multiplicação	`MR = MX0*MY0(ss);`
Multiplicar e acumular	`MR = MR + MX0*MY0(ss);`
Multiplicar e subtrair	`MR = MR - MX0*MY0(ss);`
Elevar ao quadrado	`MR = MX0*MX0(ss);`
Elevar ao quadrado e acumular	`MR = MR + MX0*MX0(ss);`
Elevar ao quadrado e subtrair	`MR = MR - MX0*MX0(ss);`
Zerar o registrador	`MR = 0;`
Copiar o registrador	`MR = MR;`
Cópia com arredondamento	`MR = MR(rnd);`
Saturação condicional	`IF MV SAT MR;`

Tabela 4: Principais operações realizadas pelo MAC (nos exemplos foi sempre usado o caso de dois operandos com sinal, todas as outras opções podem ser usadas). A instrução de cópia pode ter como saída também o registrador MF. No caso da saída ser o próprio MR, a utilidade é atualizar o valor do bit de saturação (`MV`). Todas as instruções acima podem ser condicionais, como visto no caso da ULA (exceto a instrução para saturação, que já é condicional).

[4] MV é um bit do registrador MSTAT.

[5] Lembre-se de que saturação significa verificar se o resultado ultrapassou os limites permitidos para o formato 1.15, e caso tenha ultrapassado, colocar os valores mínimo (-1) ou máximo ($1 - 2^{-15}$), conforme o caso.

Entrada X	Entrada Y	Saída
MX0, MX1	MY0, MY1	MR (MR2, MR1, MR0)
MR	MF	MF
MR0, MR1, MR2		
AR		
SR0, SR1		

Tabela 5: Registradores usados pelo MAC.

Há dois conjuntos completos de registradores para a ULA e para o MAC, o primário e o secundário. Troca-se entre os dois conjuntos com as instruções `ENA SEC_REG;` e `DIS SEC_REG;` — as instruções executadas após `ENA SEC_REG` usarão os registradores do conjunto secundário até que seja dada uma instrução `DIS SEC_REG`. A razão de haver dois conjuntos de registradores é poder armazenar o estado da ULA ou do MAC antes de se chamar uma sub-rotina. Assim, a sub-rotina pode realizar as operações que precisar, e no retorno podemos recuperar os valores originais dos registradores de trabalho com apenas uma instrução.

7 Desvios e laços

Já vimos um exemplo de desvio incondicional: a instrução `JUMP` usada na tabela de vetores de interrupção. As instruções para desvio precisam de uma etiqueta indicando para onde o programa deve ir. A etiqueta é uma palavra terminada com dois pontos (`:`). Assim, o programa

```
   AX0 = 0x4000;
   JUMP instrucao;
   AX0 = 0x0000;

      ...

instrucao:
   AY0 = 0x2000;

     ...
```

terminará com o registrador AX0 com o valor 0x4000.

Também é possível pular para o endereço contido em um dos registradores DAG2, ou seja, I4, I5, I6 ou I7:

```
JUMP (I4);
```

A instrução para chamar uma subrotina funciona de maneira semelhante ao `JUMP`, com a diferença que o endereço atual é armazenado na pilha, para permitir o retorno após o término da subrotina. Para chamar uma subrotina, a instrução utilizada é `CALL etiqueta;`, e para retornar para o programa principal após completar a subrotina, usa-se `RTS;`.

Para implementar laços, usa-se a instrução `DO ...UNTIL`, e o registrador especial CNTR (de 14 bits). No exemplo a seguir, inicializamos dois vetores de `N` elementos.

```
#define N 10

.section/data seg_input;
.var/circ x[N], y[N];

.section/pm pm_code;
   I0 = x;
   M0 = 1;
   L0 = length(x);
   I1 = y;
   L1 = length(y);

   CNTR = N;
   DO aqui UNTIL CE;
      dm(I0,M0) = 0x4000;
aqui: dm(I1,M0) = 0x0000;

   AX0 = 0;
       .
       .
       .
```

A condição para término `CE` significa "Counter Expired", ou seja, o laço é repetido até que o contador se anule — no caso do exemplo, 10 vezes.

> O contador CNTR é decrementado no início do laço, mas o teste é feito só na última instrução. Assim, se o contador for inicializado com zero (`CNTR = 0;`), o laço será executado 2^{14} vezes!

Há várias opções para usar a instrução `DO ...UNTIL`, mas o exemplo acima é suficiente para os trabalhos solicitados para esta disciplina. Dois cuidados especiais:

1. Pode-se usar no máximo 4 níveis de laços `DO ...UNTIL` encaixados um dentro do outro;
2. Dois laços encaixados não podem terminar na mesma instrução, ou seja, o programa a seguir é inválido:

```
CNTR = 5;
DO ali UNTIL CE;
       ...
   CNTR = 4;
   DO ali UNTIL CE;
       ...
```

```
ali:     AR = 0;
aqui: NOP;
```

Trocando o `ali` do laço externo para `aqui`, o programa fica correto.

Também é possível realizar uma instrução apenas se uma condição for verdadeira (normalmente, a condição depende de resultados da ULA). As instruções condicionais possíveis são

```
IF <COND> JUMP para_la;

IF <COND> RTS;

IF <COND> RTI;

IF <COND> AR = AX0 + AY0;
```

Algumas das condições possíveis estão listadas na Tabela 6.

Código	Condição
`EQ`	último resultado da ULA é zero
`NE`	ULA diferente de zero
`LT`	ULA menor que zero
`GE`	ULA maior ou igual a zero
`LE`	ULA menor ou igual a zero
`GT`	ULA maior que zero
`AC`	ULA vai-um
`NOT AC`	ULA vai-um zerado
`AV`	ULA estouro
`NOT AV`	ULA não estouro
`MV`	MAC estouro
`NOT MV`	MAC não estouro
`NEG X`	ULA entrada `X` negativa
`POS`	ULA entrada `X` positiva
`CE`	contador expirou
`NOT CE`	contador não expirou

Tabela 6: Alguns códigos para instruções condicionais.

Tecnicamente, há mais uma instrução de "desvio" condicional: a instrução `IDLE`. Esta instrução coloca o processador em um estado de baixa energia, apenas esperando que chegue uma interrupção. Essa opção é útil para que o processador economize energia enquanto espera que chegue uma nova amostra do conversor A/D, por exemplo.

8 Divisão

Operações de divisão são mais difíceis de implementar do que somas ou multiplicações, é necessário usar um algoritmo iterativo. O ADSP 2189M tem duas instruções especiais para tratar de divisão: `DIVS` e `DIVQ`, que funcionam da seguinte maneira para dividir números com sinal (para números sem sinal as operações são um pouco diferentes). O numerador deve ter 32 bits e ser armazenado nos registradores AY1 (mais significativo) e AY0 ou AF (mais significativo) e AY0. O denominador tem 16 bits e pode ser armazenado em qualquer dos registradores indicados na tabela 7. A divisão é calculada com uma operação `DIVS` seguida de 15 operações `DIVQ`, como indicado abaixo. Ao final, o resultado estará no registrador usado como denominador.

```
AY1=0.25r;
AY0=0r;  % o número completo é 0,25
AX0=-0.5r;
DIVS AY1, AX0;
DIVQ AX0; DIVQ AX0; DIVQ AX0;
DIVQ AX0; DIVQ AX0; DIVQ AX0;
DIVQ AX0; DIVQ AX0; DIVQ AX0;
DIVQ AX0; DIVQ AX0; DIVQ AX0;
DIVQ AX0; DIVQ AX0; DIVQ AX0; % o resultado está em AY0
```

Denominador	Numerador (+sig)	Numerador (-sig)
AX1, AX0, AR, MR2, MR1, MR1 SR1, SR0	AY1, AF	AY0

Tabela 7: Registradores usados pelos comandos de divisão.

Caso o denominador seja negativo (e diferente de -1), o resultado da divisão vai estar errado por um bit. Os manuais da AD fornecem rotinas para consertar esse problema (caso um erro adicional de um bit realmente vá causar problemas), e também para verificar que o resultado vai estar dentro da faixa válida. Em geral, deve-se tomar cuidado para que não haja estouro em uma operação de divisão.

9 Pegadinhas

A placa da Analog Devices foi feita para permitir testar e corrigir (*debugar*) programas. Por isso, a placa sempre carrega na memória de programa o *monitor*, que permite a comunicação entre o DSP e o computador em que o usuário está trabalhando. Como o programa monitor está (quase) sempre rodando, há algumas limitações (que não estão bem documentadas) que precisam ser atendidas durante o uso da placa:

1. O monitor se perde se a ULA for passada para o modo de saturação (ou seja, se for usado o comando `ena ar_sat;`). Para ser possível usar esse modo, deve-se desabilitar

o monitor (por exemplo, desabilitando o Timer, pondo o bit menos significativo de IMASK em 0). A comunicação entre a placa e o computador é perdida, mas o programa funciona corretamente.

2. O monitor também parece ter problemas se for usado o conjunto de registradores secundários. Assim, não use o comando `ena sec_reg;` (se você usar esse comando em algum programa, a placa continuará funcionando de maneira estranha até que você dê um comando `dis sec_reg;`).

3. Se for colocado um *breakpoint* nas últimas três instruções de um laço `do...until`, o monitor se perde.

4. O processador pode ter até quatro laços `do...until` encaixados (um dentro do outro); mas o monitor só permite que seja usado um laço de cada vez.

5. A região de memória de dados 0x00000–0x01FFF está marcada no manual como de livre uso, mas aparentemente o programa monitor altera alguns dados nesta região; é melhor evitar usá-la.

10 Comunicação com o CODEC

Para completar este resumo sobre programação na placa EZKIT 2189M, falta falar sobre a comunicação entre o processador e o CODEC AD 73322, que realiza as operações de conversão A/D e D/A (e também implementa os filtros anti-rebatimento e de reconstrução). Essa comunicação é a parte mais problemática do EZKIT, provavelmente por causa de interferências do programa monitor, que fica rodando para comunicação com o ambiente de programação.

Para reduzir os problemas, use sempre a taxa de amostragem de 8kHz, e a segunda taxa mais baixa de comunicação entre a SPORT e o DSP, como ajustado nos programas-exemplo nos computadores da sala C1-10. Note que os problemas só ocorrem se for utilizada a comunicação com o CODEC, assim os programas podem ser testados enviando dados para a memória através do `VisualDSP++`, e lendo os resultados da memória.

Outro problema é que os filtros analógicos anti-rebatimento e de reconstrução da placa começam a cortar bem cedo: para uma freqüência igual a um quarto da taxa de amostragem, a atenuação conjunta dos filtros anti-rebatimento e de reconstrução já é de aproximadamente 6dB (veja mais comentários na Seção 11).

Vamos analisar a seguir um programa que utiliza o CODEC, com explicações sobre os pontos principais. O programa mostrado abaixo implementa um filtro não-recursivo (FIR) em um dos canais de entrada[6], e no outro canal, coloca-se na saída o dobro do sinal de entrada.

No programa abaixo, temos seis partes principais:

1. Alocação de memória para as variáveis utilizadas no filtro propriamente dito;

[6] O CODEC tem dois conversores A/D e dois D/A independentes.

2. Alocação de memória para variáveis usadas na configuração do CODEC (como há dois canais, as variáveis de configuração são definidas aos pares — se você tentar mudar alguma das variáveis de configuração, lembre-se de mudar da mesma maneira as duas variáveis de cada par);

3. Tabela de vetores de interrupção;

4. Três sub-rotinas para inicialização do CODEC: `CodecStop`, `CodecInit` e `SPORT0_tx_int_handler`;

5. Programa "principal", com inicializações de variáveis para o filtro (`inicio:`);

6. Tratamento de dados: o filtro propriamente dito (`SPORT0_rx_int_handler:`).

Há duas maneiras para comunicação com o CODEC, uma em que recebe-se uma vez um dado do canal direito (uma interrupção), e depois um dado do canal esquerdo (nova interrupção), ou seja, uma interrupção por canal. Na outra forma define-se um vetor de entrada e um de saída (no exemplo com comprimento 2 cada um). O CODEC preenche o vetor de entrada com uma amostra de cada canal, e só então manda uma interrupção: neste caso, quando a interrupção chega já há uma nova amostra disponível para os dois canais de entrada. Esta segunda opção é a que está exemplificada no programa abaixo.

Os vetores para comunicação com o CODEC são `rx_buf` (que contém os valores *recebidos* do CODEC, do conversor A/D), e `tx_buf` (que contém as amostras que serão *transmitidas* para o CODEC, para o conversor D/A). Os valores de taxa de amostragem, e diversos ganhos programáveis, estão definidos no vetor `InitCommands`, de 16 posições. O vetor já está ajustado para valores convenientes para as atividades desta disciplina. Para alterar valores de taxa de amostragem, basta ler os comentários do programa (e seria conveniente também consultar o manual do AD 73322).

Duas observações adicionais: antes de inicializar as variáveis, no programa principal, usa-se a instrução `IMASK` para desabilitar interrupções (assim garantindo que a inicialização será completada antes que o processador passe a tratar dados do CODEC). A instrução `IMASK` é repetida depois de chamar as rotinas de inicialização do CODEC, pois estas rotinas usam interrupções. Depois de terminadas todas as inicializações, o programa entra em um laço infinito, com apenas uma instrução `IDLE`. O objetivo do laço é ficar esperando, no estado de baixo consumo, que chegue um novo par de amostras do CODEC para ser tratado. A rotina que realmente implementa o filtro é a marcada com a etiqueta `SPORT0_rx_int_handler`.

```
#include  "constant.h";

#define comprimento 6 /* Comprimento do filtro */
#define comprimento_menos_um 5

.global     CodecStop;          /* pára o CODEC stop: irq e auto-buffering */
.global     CodecInit;          /* inicialização do CODEC  */
.global     SPORT0_TX_INT;      /* comandos para inicialização do CODEC */

.section/data seg_input;
```

```
.VAR/circ entrada[comprimento];            /* vetor de entrada circular */

.VAR/circ rx_buf[2]; /* recebimento de dados */
.VAR/circ tx_buf[2]; /* transmissão de dados */

.section/pm pm_data;
.VAR/circ b[comprimento]="coefV.dat";         /* coeficientes do filtro */

.section/data seg_init;
.VAR InitCommands[16] = b#1000100100000100,   /* CRB1 */
   b#1000000100000100,   /* CRB0 */
   /*        |---  --
             |  |-- |
             |  | | +---Sample rate 00=DMCLK/2048, 01=DMCLK/1024,
             |  | |             10=DMCLK/512, 11=DMCLK/256
             |  | +-----Serial Clock Divider 00=SCLK/8, 01=SCLK/4,
             |  |               10=SCLK/2, 11=SCLK
             |  +-------Master Clock divider 000=MCLK, 001=MCLK/2,
             |                  010=MCLK/3, 011=MCLK/4, 101=MCLK/5
            +----------Control Echo Cancel */

   b#1000101001111001,   /* CRC1 */
   b#1000001001111001,   /* CRC0 */
  /*        ||||||||
             |||||||+---Power-Up device  0=powerdown, 1=powerup
             ||||||+----Analog gain tap power (0=powerdown 1=powerup)
             |||||+-----Input amplifier power (0=powerdonw 1=powerup)
             ||||+------ADC power (0=powerdown, 1=powerup)
             |||+-------DAC power (0=powerdown, 1=powerup)
             ||+--------REF power (0=powerdown, 1=powerup)
             |+---------REFOUT use (0=disapbe refout, 1=enable refout)
            +----------1=Enable 5V 0=Disable 5V */

   b#1000101100000001,   /* CRD1 */
   b#1000001100000001,   /* CRD0 */
   /*        |---|---
             |  ||  |
             |  ||  +----Input gain select 111 +38db
             |  |+-------Reset ADC Modulator (0=off 1=reset enabled)
             |  +--------Output gain select 000 +6db
            +-----------Output mute (0=mute off, 1=mute enabled) */

   b#1000110000000000,   /* CRE1 */
   b#1000010000000000,   /* CRE0 */
   /*        |||-----
```

```
            |||    |
            |||    +----DAC Advance Settings 0-4
            ||+---------IBYP Interpolator bypass (0=disabled 1=enabled)
            |+----------Digitaly gain tap enable 0=disabled, 1=enabled
            +-----------RESERVED (0) */

  b#1000110100000000,   /* CRF1 */
  b#1000010100000000,   /* CRF0 */
/*          ||||||||
            |||-----
            |||  |
            |||  +-----Analog Gain Tap Coefficient 0-4
            ||+----------Single-ended 0=disabled, 1=enabled
            |+-----------Input Invert (0=disabled, 1=enabled)
            +------------Analog loopback 1=enable, 0=disable */

  b#1000111011111111,   /* CRG1 */
  b#1000011011111111,   /* CRG0 */
  /*        --------
            +-----Digital Gain Tap 0-7 */

  b#1000111111111111,   /* CRH1 */
  b#1000011111111111,   /* CRH1 */
  /*        --------
            +-----Digital Gain Tap 8-15 */

  b#1000100000010001,   /* CRA1 */
  b#1000000000010001;   /* CRA0 */
  /*        |---||||
            |  ||||+-----Data/Program operating mode (0=program, 1=data)
            |  |||+------Mixed Mode (0=off, 1=on)
            |  ||+-------Digital loopback 1=enable, 0=disable
            |  |+--------SPORT loopback mode 0=off, 1=on
            |  +---------Device count
            +------------Software Reset (0=off, 1=initiate) */


.section/data seg_data;
.VAR TX_InitEnable = 0;




.section/pm seg_rth;
             JUMP inicio; NOP; NOP; NOP;      /* reset */
```

```
              RTI; NOP; NOP; NOP;              /* IRQ2 */
              RTI; NOP; NOP; NOP;              /* IRQ1L */
              RTI; NOP; NOP; NOP;             /* IRQL0 */
SPORT0_TX_INT:   JUMP SPORT0_tx_int_handler; NOP; NOP; NOP; /*SPORT0_TX*/
SPORT0_RX_INT:   JUMP SPORT0_rx_int_handler; NOP; NOP; NOP; /*SPORT0_RX*/
            RTI; NOP; NOP; NOP;             /*IRQE*/
              RTI; NOP; NOP; NOP;             /* BDMA */
              RTI; NOP; NOP; NOP;             /* SPORT1 TX */
              RTI; NOP; NOP; NOP;             /* SPORT1 RX */
              NOP; NOP; NOP; NOP;             /* TIMER */
              RTI; NOP; NOP; NOP;             /* POWERDOWN */

/* Início do programa principal */
.section/pm seg_code;
inicio:
   imask = 0x0001; /* Desabilita interrupções, exceto o timer */


   I0 = entrada;
   M0 = 1;
   L0 = comprimento;

   CNTR = L0;
   do zero until CE;
zero: DM(I0,M0) = 0;

   call CodecStop;            /* pára o codec */
   call CodecInit;            /* inicializa o codec */

IMASK = 0x0001;               /* Desabilita interrupções, exceto o timer */

   I0 = entrada;  /* apontador para vetor de entrada */
   L0 = comprimento;
   M0 = 1;
   I4 = b; /* apontador para vetor de coeficientes */
   L4 = comprimento;
   M4 = 1;
   I1 = rx_buf; /* vetor de recebimento de dados */
   L1 = length(rx_buf);
   I2 = tx_buf; /* vetor para transmissão de dados */
   L2 = length(tx_buf);
   M1 = 1;
   dm(I2,M1) = 0;
   dm(I2,M1) = 0;
   DIS M_MODE;  /* Configura o DSP para operações com números fracionarios */
```

```
   ENA AR_SAT;  /* Habilita saturação nas operações da ULA */
IMASK = 0x0071; /* habilita interrupções */


ESPERA:
   nop;
   IDLE;
   nop;
JUMP ESPERA;
/* final do programa */


/* Rotina de parada do Codec - vamos pular */
/************************************************************************
 *
 *  CODEC stops IRQ and auto-buffering; codecInit require afterward.
 *
 *  REGISTER USAGE SUMMARY:
 *
 *  input  : ar, ax0, ay0
 *  update : imask
 *  output : none
 *  destroy: none
 *  keep   : none
 *  memory : none
 *  calls  : none
 *
 ************************************************************************/
.section/pm seg_code;
CodecStop:
   imask = 0x0001;

/*    ...  */

/* Rotina de inicializacao do Codec - tambem vamos pular */
/************************************************************************
 *
 *  CODEC initialization routine.
 *
 *  REGISTER USAGE SUMMARY:
 *
 *  input  : none
 *  update : none
 *  output : none
 *  destroy: ar, af, ax0, ay0, i5, m5, l5, i6, l6, i7, l7,
```

```
 *  keep   : none
 *  memory :
 *      variables:
 *           dm(stat_flag)
 *           dm(tx_buf)
 *           dm(rx_buf)
 *           dm(init_cmds)
 *
 *      memory mapped control registers:
 *           dm (SPORT0_Autobuf)
 *           dm (SPORT0_RFSDIV)
 *           dm (SPORT0_SCLKDIV)
 *           dm (SPORT0_Control_Reg)
 *           dm (SPORT0_TX_Channels0)
 *           dm (SPORT0_TX_Channels1)
 *           dm (SPORT0_RX_Channels0)
 *           dm (SPORT0_RX_Channels1)
 *           dm (System_Control_Reg)
 *  calls  : none
 *
 ************************************************************************/
.section/pm seg_code;
CodecInit:
        /*   shut down sport 0 during re-initialization */
        call CodecStop;
        /* ... */

/* Rotina de inicializacao do Codec (envio de dados)  - vamos pular */
/************************************************************************
*
*
*
*
************************************************************************/

/*-----------------------------------------------------------------------
 -
 -  transmit interrupt used for Codec initialization
 -
 -  REGISTER USAGE SUMMARY:
 -
 -  input  : m5 = 1
 -  update : i7
 -  output : tx0
 -  destroy: none (second register bank)
```

```
 -  keep   : none
 -  memory : dm (tx_buf), dm (stat_flg), init_cmds
 -  calls  : none
 -
 ------------------------------------------------------------------------------*/
.section/pm seg_code;
SPORT0_tx_int_handler:
   ena sec_reg;

/* ... */

/*********************************/
/* Rotina de tratamento de dados */
/*********************************/
.section/pm seg_code;
SPORT0_rx_int_handler:
   MY0 = dm(I1,M1); /* Recebe amostra do Codec */
   DM(I0,M0) = MY0; /* e guarda na linha de atrasos */

    MR = 0, MX0 = DM(I0,M0); MY0 = PM(I4,M4);

    CNTR = comprimento-1;

    do convl until ce;
       MR = MR + MX0 * MY0(ss), MX0 = DM(I0,M0); MY0 = PM(I4,M4);
convl:
    MR = MR + MX0 * MY0(rnd);
    if MV sat MR;

   dm(I2,M1) = MR1; /* põe saida do filtro no buffer de transmissão */

segundo_canal:
   AY0 = dm(I1,M1);
   AX0 = AY0;
   AR = AX0 + AY0;
   dm(I2,M1)=AR;
   rti;

/* ////////////////////////// Final ////////////////////////// */
```

11 Resposta dos filtros analógicos do EZKIT 2189M

Como todo sistema amostrado, o EZKIT 2189M tem um filtro anti-rebatimento e um filtro de reconstrução analógicos. Os filtros feitos para o EZKIT têm freqüência de corte variável conforme a taxa de amostragem utilizada (o que é uma vantagem), mas têm uma queda apreciável dentro da faixa $0 \leq \Omega \leq f_a/2$, em que f_a é a freqüência de amostragem. A Figura 2 apresenta o gráfico da resposta em freqüência (módulo) da placa quando o sistema digital apenas coloca na saída a metade do sinal recebido na entrada. Pode-se notar que os ganhos dos filtros analógicos devem ser levados em conta ao se calcular o ganho total do sistema (os ganhos dos filtros analógicos são programáveis por *software*, e através de um *jumper* na placa do EZKIT 2189M). Um problema para usar a placa em projetos próprios é que os filtros analógicos impõem uma freqüência de corte baixa, em torno de 1,5kHz para freqüência de amostragem $f_a = 8$kHz.

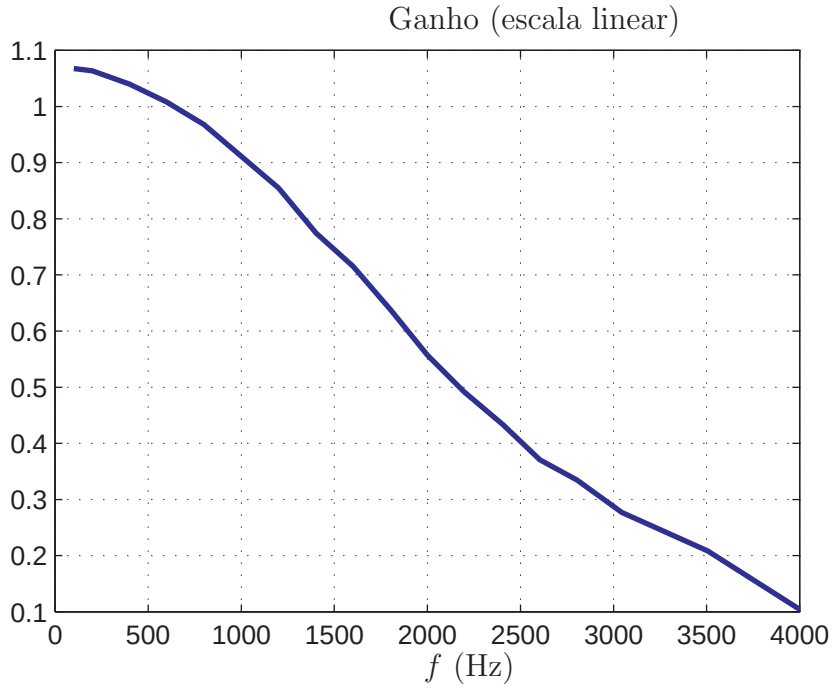

Figura 2: Resposta em freqüência medida com o EZKIT 2189. A freqüência de amostragem utilizada é de 8kHz, e a parte digital apenas coloca um ganho de 0,5 no sinal de entrada, independente da freqüência. A resposta em freqüência observada é o produto dos ganhos dos filtros analógicos e dos ganhos internos do CODEC, pela resposta (plana) do filtro digital.

12 Observações finais

Esta apostila é apenas um resumo das principais características da linguagem de programação do ADSP-2189M. Muitas instruções não foram tratadas, pois a ênfase é mostrar o material mais importante para os trabalhos pedidos no curso. Alguns detalhes não foram tratados,

como quais instruções podem ser executadas simultaneamente. Sempre é bom ter em mente que, se erros de montagem ocorrerem, pode ser conveniente uma consulta ao manual do conjunto de instruções.

Também foi tratada a resposta em freqüência do filtro anti-rebatimento para o conversor A/D, e do filtro de reconstrução, para o conversor D/A. A resposta combinada desses filtros já provoca uma atenuação bastante considerável em um quarto da freqüência de amostragem: um projeto realmente bem feito levaria em conta esses filtros para escolher a resposta do filtro digital.

Sugestões sobre a apostila, e correções de erros encontrados no texto, são bem-vindas.

13 Manuais da Analog Devices

Os seguintes manuais da Analog Devices estão disponíveis no subdiretório

`c:\Arquivos de Programas\Analog Devices\VisualDSP\Docs`

nos computadores da sala C1-10:

1. Manual do Assembler (`218x_asm`) — Esse manual descreve o Assembler e o Pré-Processador para os DSPs da família 218x. São descritas instruções para
   (a) Definição e inicialização de "variáveis" (como veremos a seguir, uma variável em Assembly é pouco mais do que um nome para uma posição de memória, algo semelhante a um ponteiro em C),
   (b) Descrição do sistema de *hardware* utilizado (tamanho e faixas de endereços dos bancos de memória internos do processador — de programa e de dados—, e dos bancos de memória externos),
   (c) Definição de qual seção de memória (banco de memória e posição) será usada para cada bloco do programa e para cada variável,
   (d) Definição de constantes que podem ser usadas no programa, e inclusão de código armazenado em arquivos separados.
2. Manual do *Hardware* (`218x_hwr`) — Neste manual é descrita a arquitetura dos processadores 218x:
   (a) Unidades computacionais (MAC, ULA, deslocador), registradores de trabalho,
   (b) Seqüenciador de programa (estruturas para geração de laços, desvios condicionais, chamada de subrotinas),
   (c) Interrupções,
   (d) Registradores de status,
   (e) Registradores para geração de endereços,
   (f) Interface com portas seriais,
   (g) Temporizador,

(h) Interface com dispositivos externos (memórias e DMA, informações para projeto de placas usando o processador).

3. Referência do conjunto de instruções (`218x_is`) — este manual descreve todas as instruções disponíveis para programação nos ADSP-218x.

4. Manual do Kit ADSP-2189M EZ-KIT (`ADSP-2189M EZ-KIT`) — Descreve sucintamente a placa para testes do ADSP-2189M.

5. Manual do Visual DSP++ (`21xx_usm`) — Aqui é descrito o ambiente integrado de programação para os processadores da família ADSP-21xx. Muitas das suas funções são bem fáceis de usar, bastando seguir as instruções nos menus.

6. Manual do Codec (`AD73322_b.pdf`) — Descreve o codificador-decodificador, ou seja, o CI que realiza as operações de conversão analógico/digital (A/D) e digital/analógico (D/A) disponível na placa do EZ-KIT. Neste manual descreve-se como configurar o AD 73322 para ajuste de taxa de amostragem, ganho de entrada, ganho de saída, entre outros.

   Outros manuais menos úteis para o trabalho nesta disciplina são:

7. Manual do Linker (`21xx_lkm`) — Neste manual é descrito o funcionamento do *linker*, que junta as várias partes de um programa em um código executável, e o *loader*, que carrega um arquivo com programa e dados nas posições corretas de memória do processador (ou, mais exatamente, na placa em que o processador está montado).

8. Manual do Compilador C (`218x_ccm`) — Este manual descreve o compilador C para os DSPs da família 218x, principalmente comandos especiais para posicionamento dos vários blocos de programa, de dados e de variáveis na memória. Infelizmente o compilador C não fez parte do pacote doado pela Analog Devices, e não está disponível para este curso.