专题七
通用数字信号处理方法的DSP实现

DSP常见的几种信号处理算法：

- DSP定点基本算术运算
- 定点DSP浮点运算
- 级数展开
- FIR滤波器的实现
- FFT的实现
- 自适应滤波（LMS）的实现

§5.1-1 实现16定点加法

- C54X中提供了多条用于加法的指令，如ADD，ADDC，ADDM和ADDS。其中ADDS用于无符号数的加法运算，ADDC用于带进位的加法运算（如32位扩展精度加法），而ADDM专用于立即数的加法。

```
ld    temp1,a          ；将变量temp1装入寄存器A
add   temp2,a          ；将变量temp2与寄存器A相加，结
                       ；果放入A中
stl   a,add_result     ；将结果（低16位）存入变量
                       ；add_result中。
```

§5.1-2 实现16定点减法

- C54X中提供了多条用于减法的指令，如SUB，SUBB，SUBC和SUBS。其中SUBS用于无符号数的减法运算，SUBB用于带进位的减法运算（如32位扩展精度的减法），而SUBC为移位减，DSP中的除法就是用该指令来实现的。

```
stm  #temp1,ar3     ；将变量temp1的地址装入ar3寄存器
stm  #temp3,ar2     ；将变量temp3的地址装入ar3寄存器
sub  *ar2+, *ar3,b  ；将变量temp3左移16位同时变量
                    ；temp1也左移16位，然后相减，结果
                    ；放入寄存器B（高16位）中，同时ar2加1。
sth  b,sub_result   ；将相减的结果（高16位）存入变量
                    ；sub_result。
```

§5.1-3 实现16定点整数乘法

- 在C54X中提供了大量的乘法运算指令，其结果都是32位，放在A或B寄存器中。
- 乘数在C54X的乘法指令很灵活，可以是T寄存器、立即数、存贮单元和A或B寄存器的高16位。
- 在C54X中，一般对数据的处理都当做有符号数，如果是无符号数乘时，请使用MPYU指令。

16定点整数乘法例子

```
rsbx   FRCT      ；清FRCT标志，准备整数乘
ld     temp1,T   ；将变量temp1装入T寄存器
mpy    temp2,a   ；完成temp2*temp1，结果放
                 ；入A寄存器（32位）
```

- 在C54X中，小数的乘法与整数乘法基本一致，只是由于两个有符号的小数相乘，其结果的小数点的位置在次高的后面，所以必须左移一位，才能得到正确的结果。C54X中提供了一个状态位FRCT，将其设置为1时，系统自动将乘积结果左移移位。

16定点整数乘法例子

```
Ssbx   FRCT        ；FRCT=1，准备小数乘法
ld     temp1,16,a  ；将变量temp1装入寄存器A的高16位
mpya   temp2       ；完成temp2乘寄存器A的高16位，结
                   ；果在B中，同时将temp2装入T寄存器
sth    b,mpy_f     ；将乘积结果的高16位存入变量mpy_f
```

- 0ccdH（十进制的0.1）x 0599aH(十进制的0.7），两数相乘后B寄存器的内容为08f5f0a4H（十进制的0.07000549323857）。
- 可以使用RND或使用MPYR指令对低16位做四舍五入的处理。

§5.1-4 实现16定点整数除法

- 在C54X中没有提供专门的除法指令，一般有两种方法来完成除法。一种是用乘法来代替，除以某个数相当于乘以其倒数，所以先求出其倒数，然后相乘。这种方法对于除以常数特别适用。另一种方法是使用SUBC指令，重复16次减法完成除法运算。
- temp1/temp2为例，说明如何使用SUBC指令实现整数除法:

```
ld    temp1,T    ；将被除数装入T寄存器
mpy   temp2,A    ；除数与被除数相乘，结果放入A寄存器
ld    temp2,B    ；将除数temp2装入B寄存器的低16位
abs   B          ；求绝对值
stl   B,temp2    ；将B寄存器的低16位存回temp2
ld    temp1,B    ；将被除数temp1装入B寄存器的低16位
abs   B          ；求绝对值
rpt   #15        ；重复SUBC指令16次
subc  temp2,b    ；使用SUBC指令完成除法运算
bcd   div_end,agt  ；延时跳转，先执行下面两条指令，
                   ；然后判断A，若A>0，则跳转到标号
                   ；div_end，结束除法运算
stl   B,quot_i   ；将商（B寄存器的低16位）存入变量quot_i
sth   B,remain_i ；将余数（B寄存器的高16位）存入变量remain_i
xor   B          ；若两数相乘的结果为负，则商也应为负。
Sub quot_i,B     ；将商反号
stl   B,quot_i   ；存回变量quot_i中
div_end:
```

§5.1-5 实现16定点小数除法

- 在C54X中实现16位的小数除法与前面的整数除法基本一致，也是使用循环的SUBC指令来完成。但有两点需要注意：
- 第一，小数除法的结果一定是小数（小于1），所以被除数一定小于除数。这与整数除法正好相反。所以在执行SUBC指令前，应将被除数装入A或B寄存器的高16位，而不是低16位。其结果的格式与整数除法一样，A或B寄存器的高16位为余数，低16位为商。
- 第二，与小数乘法一样，应考虑符号位对结果小数点的影响。所以应对商右移一位。

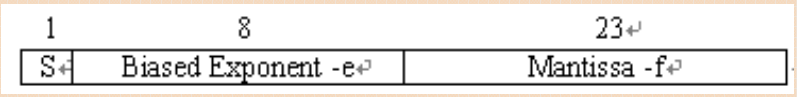
§5-2 C54X的浮点数的算术运算

浮点数的表示方法

- 在定点运算中，小数点是在一个特定的固定位置。在定点运算系统中，虽然在硬件上实现简单，但是表示的操作数的动态范围要受到限制。使用浮点数，可以避免这个困难。
- 一个浮点数由尾数**m**、基数**b**和指数**e**三部分组成。即：

$$m * b^{e}$$

IEEE标准里的浮点数表示方法



- 这个格式用带符号的表示方法来表示尾数，指数含有127的偏移。在一个32-bit表示的浮点数中，第一位是符号位，记为S。接下来的8-bit表示指数，采用127的偏移格式（实际是e-127）。然后的23-bit表示尾数的绝对值，考虑到最高一位是符号位，它也应归于尾数的范围，所以尾数一共有24-bit。

IEEE标准里的浮点数表示方法

当$0 < e < 255$时　　为$(-1)^s * 2^{e-127} * (1.f)$

- 比如说：十进制数-29.625可以用二进制表示为-11101.101B，用科学计数法表示为 $-1.1101101*2^4$，其指数为127+4=131，化为二进制表示为10000011B，故此数的浮点格式表示为11000001111011010000000000000000，转换成16进制表示为0xC1ED0000。

浮点数运算的基本步骤

- 分离符号、指数、尾数。
- 根据需要对齐指数，按运算法则进行定点数运算。
- 归一化指数
- 按浮点数格式重新组合。

- 下面以浮点数加为例，详细介绍

一. 浮点数加法运算的步骤

- 要在C54X上实现浮点运算，操作数必须变换成定点数，实际上就是一个数据格式的转换问题。
- 执行完下面代码后，操作数1就从原来在op1_hsw和op1_lsw（共32bit）中的浮点数格式转换为三部分：指数和符号位（存储于op1se中）、尾数低位（存储于op1lm中）以及尾数高位（存储于op1hm中），这时所有的数值就都转换为定点数的格式了。

```
dld     op1_hsw,a          ;将OP1装入累加器A中。
sfta    a,8
sfta    a,-8               ;通过先左移后右移使 AG = 0。
bc      op1_zero,AEQ       ;如果OP1是零，转入特殊处理。
sth     a,-7,op1se         ;将符号和指数存储到OP1SE中。
stl     a,op1lm            ;存储尾数的低位。
and     #07Fh,16,a         ;将浮点数格式中的符号和指数去掉
                           ;得到尾数的高位。
add     #080h,16,a         ;给尾数加上小数点前的“1”。
sth     a,op1hm            ;存储尾数的高位。
```

浮点数加法运算

- 由于实行的是带符号位的运算，所以要把尾数变成带符号的形式，即如果是正数就不改变其存储格式，而如果是负数，就要用补码来表示尾数，这部分的代码如下：

```
bitf    op1se,#100h  ；取出op1符号位的值于TC位中
bc      testop2,NTC  ；如果TC = 0则跳转到testop2处
ld      #0,a         ；
dsub    op1hm,a      ；0 – op1的尾数，得到尾数的补码表示
dst     a,op1hm      ；将尾数存入op1hm和op1lm中
testop2:
bitf    op2se,#100h  ；取出op1符号位的值于TC位中
bc      compexp,NTC      ；如果TC = 0则跳转到compexp处
ld      #0,a         ；
dsub    op2hm,a      ；0 – op1的尾数，得到尾数的补码表示
dst     a,op2hm      ；将尾数存入op1hm和op1lm中
```

浮点数加法运算

- 通过上述过程，把浮点数分解为尾数和指数两个部分，对于浮点数的加法而言，计算过程是左移指数较小的操作数的尾数，使得两个操作数的指数一致，相差几位移几位，再把尾数相加，归一化后输出。
- 先比较两个操作数指数的大小，以决定到底是尾数直接相加还是移位以后再相加，该段过程可以用以下的代码来实现：

```
compexp:
    ld    op1se,a        ; 将操作数1的符号和指数位装入acc A中
    and  #00ffh,a        ; 去掉符号位的影响
    ld    op2se,b        ; 将操作数2的符号和指数位装入acc B中
    and  #00ffh,a        ; 去掉符号位的影响
    sub a,b              ; op2的指数 - op1的指数结果赋给acc B
    bc   op1_gt_op2,BLT      ; 跳到进行 op1 > op2的操作处
    bc   op2_gt_op1,BGT      ; 跳到进行 op2 > op1的操作处


a_eq_b:                  ; 执行A = B的操作
    dld     op1hm,a      ; 将操作数1的尾数（32bit）放到acc A中
    dadd  op2hm,a        ; 将操作数2的尾数与操作数1的尾数进行
                         ; 32bit的双精度加法
```

```
op1_gt_op2:
    abs   b              ; 去掉符号位的影响
    sub   #24,b          ; 判断op1_se是否比op2_se大24
    bc    return_op1,BGEQ ; 如果op1_se远远大于op2_se则
                          ; 转入相应的返回op1的操作
    add    #23,b         ; 恢复指数的差值
    stl    b,rltsign     ; 存储指数差值以准备作为RPC使用
    dld    op2hm,a       ; 将OP2的尾数（32bit）装入acc A中
    rpt    rltsign       ; 规范OP2以适应OP1
    sfta   a,-1
    bd     normalize     ; 延迟方式执行跳转到normalize 处
    ld     op1se,b       ; 将指数值装入acc B以准备规一化处理
    dadd op1hm,a         ; 将 OP1和OP2的尾数相加
```

浮点数加法运算

- 完成对操作数之间的计算后就需要对结果数进行归一化操作。所谓归一化操作就是把数据格式转换为第一位为符号位，紧接着的一位是非零的数的格式，即是：**S1xxxxxxxxxxxxxx**的形式。可以用如下的代码来实现：

```
normalize:
    sth   a,rltsign     ；将带符号的尾数高位存入rltsign单元中
    abs   a             ；去掉符号位的影响
    sftl  a,6           ；进行归一化的预处理，将acc中的值
                        ；左移6位
    exp   a             ；对acc A中的值进行规一化操作
    norm  a
    st    t,rltexp      ；存储规一化时左移的数值到rltexp中
    add   #1,b          ；考虑到进位位所以要给指数加上“1”
    sub   rltexp,b      ；完成指数的归一化工作
```

浮点数加法运算

- 在这段代码中acc A中的值是由两个24bit的操作数定点相加而得到的，其MSB可能在bit24-bit0中的任何一位，所以首先左移6位让bit24移至bit30，接下来通过exp和norm指令联合使用之后acc A中的数据变成归一化形式。比如：这两条指令执行之前acc A中的值为0x01800000，执行以后A中的值为0x60000000。

浮点数加法运算

- 操作数左移的位数被存于寄存器T中，如上述操作后寄存器T中的值为0x0006，所以最后的指数的计算还要减去寄存器T中的值才是最后结果的指数值。
- 归一化完后要再把归一化后的定点数转变为浮点数，这是前面的转变浮点数为定点数的逆过程，参照前述步骤可以用如下程序实现：

```
normalized:
    stl   b,rltexp          ；将结果指数存在rltexp中
    bc    underflow,BLEQ    ；如果B〈0 进行下溢处理
    sub #0ffh,b             ；进行上溢处理判断
    bc    overflow,BGEQ     ；如果B〉0 进行上溢处理
    sftl  a,-7              ；将尾数右移7位以方便以后合成为32bit
                            ；的浮点数格式
    stl   a,rltlm           ；存储低位的尾数
    and #07f00h,8,a         ；将最开始中小数点前的“1”省略
    sth   a,rlthm           ；存储高位的尾数
```

```
;----------  Conversion of Floating Point Format to Pack --------
ld     rltsign,9,a      ；将rltsign左移9位后acc A中
and    #100h,16,a       ；取出rltsign中的符号位，使acc A中
                        ；的格式为
                    ；  0000 000S 0000 0000 0000 0000 0000 0000
add    rltexp,16,a      ；将结果的指数存入acc A中满足浮点
                        ；数格式的相应位置。A中格式为：
                    ；  0000 000S EEEE EEEE 0000 0000 0000 0000
sftl   a,7              ；A中格式为：
                    ；  SEEE EEEE E000 0000 0000 0000 0000 0000
dadd   rlthm,a          ；最终在acc A中得到32位的浮点数格
                        ；式如下：
    ；  SEEE EEEE EMMM MMMM MMMM MMMM MMMM MMMM
```

二. 减法的运算

- 减法运算的实现与加法运算几乎完全一样，只是在具体进行运算时使用的是减法指令而不是加法指令。

三. 浮点乘法的运算步骤

- 当两个数相乘时，同样要经过分离指数和尾数，进行算术运算，以及最后的归一化存储这几个过程
- 运算时要遵循乘法运算的规则把指数相加，尾数相乘。
- 需要注意的是，由于C54X的乘法器在一个指令周期内只能完成17bit*17bit的二进制补码运算，故相乘运算需要分步实现。

乘法运算

乘法运算

- **进行的这种32bit*32bit的运算是有精度损失的。如上图所示，32位数0QRS乘以32位数0XYZ所得的结果应该是一个64bit的值。由于0Q*0X的高16bit始终为0，所以结果可以用一个48位的值准确表示。但是用于存储结果的存储单元长度只有32bit，于是只有将最低的16bit省略掉，即把RS*YZ的低16bit略去，因此最后得到的是一个有精度损失的32bit结果。**

乘法运算

- 由于进行的是带符号的运算，所以在进行相乘运算时首先要确定最后结果的符号位。根据“同号为正，异号为负”的原则，可以写出以下代码：

```
Ld      op5se,a         ；装入op5的指数和符号位
xor     op6se,a         ；与op6的指数和符号位相异或
and     #00100h,a       ；屏蔽指数得到符号位
stl     a,rltsign       ；得到结果的符号位
```

```
;------ exponent summation ----------
        ld      op5se,a
        and     #00FFh,a        ；去掉符号位的影响
        ld      op6se,b
        and     #00FFh,b        ；去掉符号位的影响
        sub     #07fh,b         ；op6的指数减去127（避免结果加
                                ；上两个127偏移)
        add     b,a             ；加上op5的指数
        stl     a,rltexp        ；将指数结果存于rltexp中
        bc      underflow,ALT           ；如果exp < 0则跳转到下溢处
                                        ；理underflow处
        sub     #0FFh,a                 ；测试是否产生上溢
        bc      overflow,AGT            ；如果exp > 255则跳转到下溢
                                        ；处理overflow处
```

尾数运用上面的公式进行运算:

```
ld  op5lm,t        ; 将OP5的低位尾数装入T寄存器
mpyu  op6lm,a      ; RS*YZ
mpyu  op6hm,b      ; RS*0X
add   a,-16,b      ; B=(RS*YZ)+(RS*0X)
ld  op5hm,t        ; 将OP5的高位尾数装入T寄存器
mpyu  op6lm,a      ; A=0Q*YZ
add   b,a          ; A=(RS*YZ)+(RS*0X)+(0Q*YZ)
mpyu  op6hm,b      ; B=0Q*0X
stl  b,rlthm       ; 得到0Q*0X的低16bit
add   rlthm,16,a   ; A=最后的结果
```

§5-3 级数求和的实现

PLOY指令的应用

利用POLY指令计算泰勒级数

- 三角函数、对数、指数等超越函数都可以用级数展开，如常用的泰勒级数。
- 不同的级数展开有不同的收敛速度。
- 以指数 e^x 的展开为例：

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!}$$

（实际一般取9项）

利用POLY指令计算泰勒级数

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!}$$

$$= 1 + x(1 + \frac{x^1}{2!} + \frac{x^2}{3!} + \ldots + \frac{x^{n-1}}{n!})$$

$$= 1 + x(1 + \frac{1}{2!} + x(\frac{1}{3!} + x(\frac{1}{4!} + \ldots x(\frac{1}{7!} + \frac{x}{8!}))))$$

利用POLY指令计算泰勒级数

- POLY指令的含义:　poly Smem

Round(A(32-16)) x T + B -> A
Smem < 16 -> B

x -> T

Smem -> 1/n!

- 指数展开

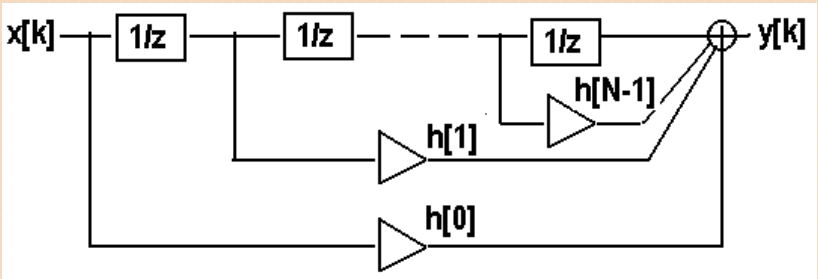
$$=1+x(1+\frac{1}{2!}+x(\frac{1}{3!}+x(\frac{1}{4!}+...x(\frac{1}{7!}+\frac{x}{8!}))))$$

利用POLY指令计算泰勒级数

```
a9 .int 1, 7, 46, 273, 1365, 5461, 16384, 32767, 0, 0

taylor:
     STM a9, AR3      ;AR3 points to coefficient in
                      ;Taylor's equ.
     LD X, T          ;set up running environment  using
     LD *AR3+, 16, A ;powerful poly instruction on 54x DSP
     LD *AR3+, 16, B

     RPT #7           ;loop 8 times enough for audio app.
     POLY *AR3+       ;AH=fractional part in Q15
                      ;format
```

§5-4 FIR滤波器

FIR滤波器的结构

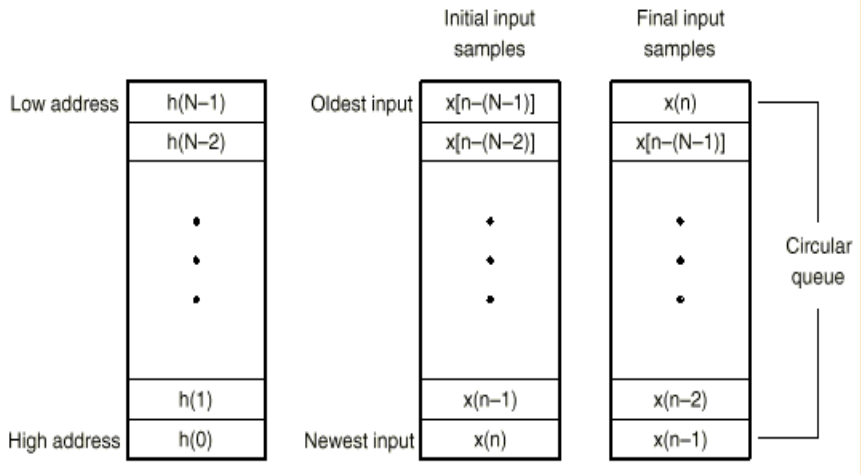


$$y(n) = h(0)x(n) + h(1)x(n-1) + \ldots + h(N-1)x[n-(N-1)]$$

DSP实现FIR滤波器

- 可以使用数字滤波器辅助设计软件包或自行计算FIR滤波器的系数。
- 本实验例子中使用的是一个34阶的对称结构的FIR低通滤波器，其采样频率Fs为25KHZ，通带截止频率 1.5KHZ，阻带截止频率为3KHZ，阻带衰减为-40dB。（系数使用DFDP4自动生成）

FIR滤波器的数据存储方式



FIR滤波器的DSP实现

- 为了能正确使用循环寻址，数据缓冲区和系数的开始地址必须正确指定。
- 滤波系数指针初始化时指向h(N-1)，经过一次FIR滤波计算后，在循环寻址的作用下，仍然指向h(N-1)。
- 而数据缓冲区指针指向的是需要更新的数据，如x(n)。在写入新数据并完成FIR运算后，该指针指向x(n-(N-1))。
- 使用循环寻址可以方便地完成滤波窗口数据的自动更新.

FIR滤波器的DSP实现

- 使用带MAC指令的循环寻址模式实现FIR滤波器，程序片段如下：（输入数据在AL中, 滤波结果在AH中)

```
STM   #1,AR0        ; AR0=1
STM   #N,BK         ; BK=N,循环寻址BUFFER大小为N
STL   A,*FIR_DATA_P+%    ; 更新滤波窗口中的采样数据
RPTZ  A,#(N-1)      ;  重复MAC指令N次, 先将A清零
MAC  *FIR_DATA_P+0%,*FIR_COFF_P+0%,A
                    ; 完成滤波计算。注意FIR滤波系数存
                    ;  放在数据存储区
```

```
    .title  " example for FIR filter !"
    .mmregs
    .global mainstart

OFF_INTR_3    .set     0Ch
k_win_size    .set     34              ; 34 taps FIR LP Filter
win_data      .set     2000h           ; FIR data window

      .data
filter_coff   .word    8ch     ;low_pass,1.5kHz(3kHz),34  h(N-1)
              .word    59h     ;    fs=25 kHz, fc= 1.5kHz
              .word    0FF78h
                  ......
              .word    0Fe56h
              .word    0FF78h
              .word    59h
end_coff      .word    8ch     ; h(0)
```

```
    .text
_c_int00:
    ssbx intm                   ; disable all interrupt !
    stm  #VECTOR,ar0            ; vector'start address -> ar0
    st   #INSTR_B,*ar0(#OFF_INTR_3)
    st   #fir,*ar0(#OFF_INTR_3+1)      ; init A/D int vector !

    ld   #temp,dp               ;--- the following codes for MAC
    st   #win_data,t_ar3        ; ar3 -> 2000h data windows
    st   #filter_coff,t_ar2     ; ar2 -> filter coff

    xor  a,a                    ; clear a
    xor  b,b                    ; clear b
    rsbx intm                   ; enable all int !

g:  idle 1
    b    g
```

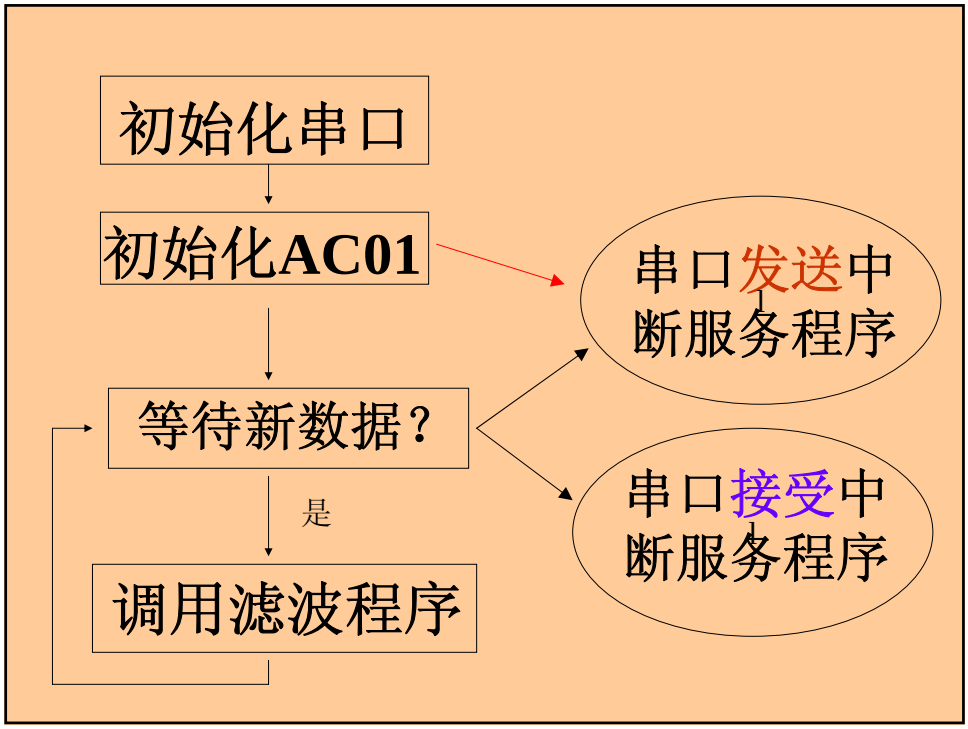
```
;-------------------------------------------------------------------------
;    These codes may be called  by serial rev int ! all registers don't
;    change !
;    When enter this subroutine, the AD data has been put in A !
;-------------------------------------------------------------------------
fir:
    ld   #temp,dp
    stl  a,temp            ; a -> AD data !
    call low_pass_mac
    ld   a,-16,b

    ld   #0,dp
    stl  b,2,TDXR          ; sent result to DA !

    rete
```

```
;*********************************************************
;   LOWPASS FILTER (use MAC)
;   Input data -> A, output data -> A
;   used  AR2 -> coff,AR3 -> data_buffer !
;*********************************************************
low_pass_mac:
    pshm st1
    pshm st0
    pshm ar0
    pshm bk

    mvdm #t_ar2,ar2        ; restore ar2
    mvdm #t_ar3,ar3        ; restore ar3
```

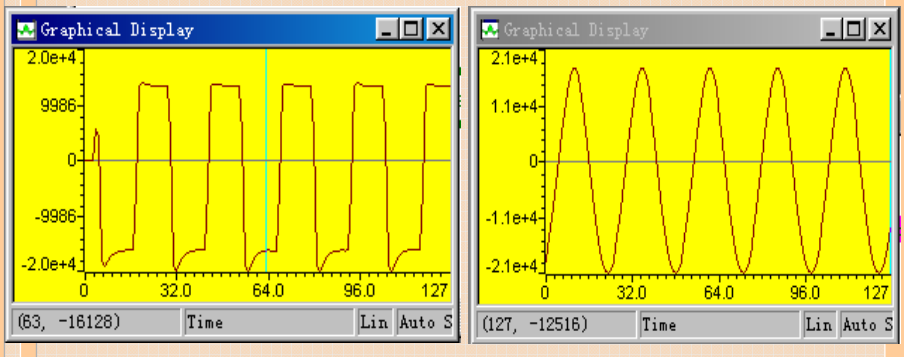
```
stm  #1,ar0
stm  #N,bk                    ; set circular addressing size
stl  a,*ar3+%
rptz a,#(N-1)                 ; 0 -> a, then repeat 34 times
mac  *ar2+0%,*ar3+0%,a        ; done FIR filter, result in a

mvmd ar3,#t_ar3               ; save ar3
mvmd ar2,#t_ar2               ; save ar2
popm bk
popm ar0
popm st0
popm st1

ret
```

FIR滤波器的工程实现

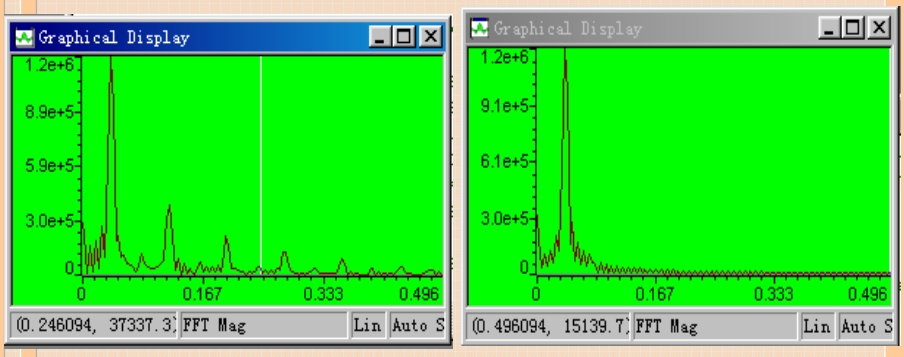
- 程序、数据的存储器安排
- 程序功能框图
- 相关外设的准备：DSP，AD/DA，……
- 相关外设的软件设置：McBSP串口初始化、AC01的初始化、……
- 硬件电路设计、调试
- 软件设计、调试



CCS图形工具显示FIR滤波效果



图形显示FIR输入/输出频谱

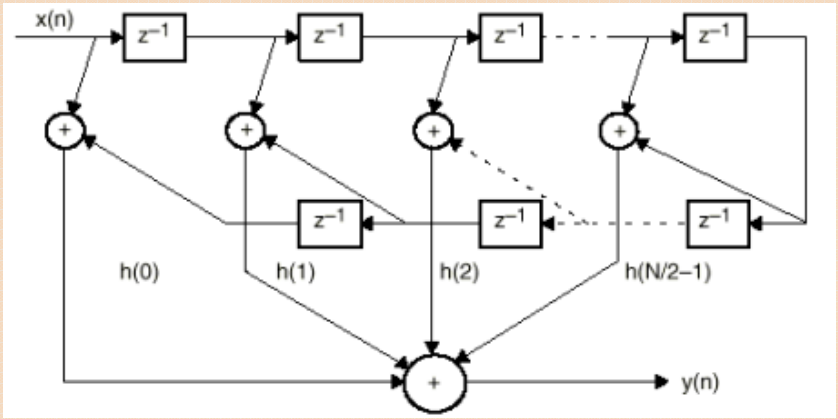


FIRS指令来实现FIR滤波器

- 一种有限单位冲激响应呈现对中心点对称的**FIR**滤波器。长度为**N** 的线性相位**FIR**滤波器的输出表达式为：

$$y(n)=\sum_{k=0}^{N/2-1}h(k)[x(n-k)+x(n-(N-1-k))]$$

FIRS指令来实现FIR滤波器



FIRS指令来实现FIR滤波器

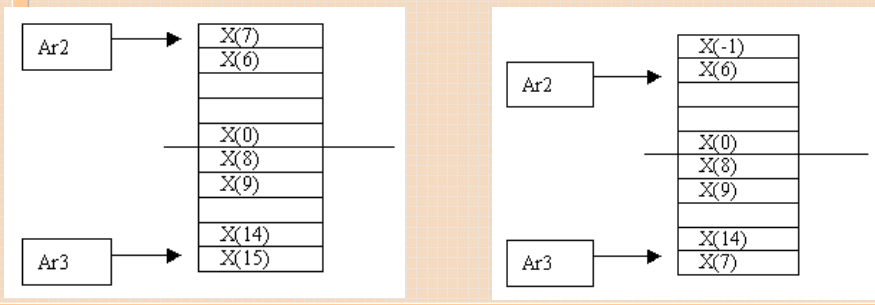
- FIRS Xmem,Ymem,pmad含义：

B +（A(32-16)) x Pmad -> B
(Xmem + Ymem) << 16 -> A
PAR++

- Pmad寻址FIR滤波器系数，Xmem和Ymem分别指向窗口数据的上下两部分。

16点FIRS滤波数据存放

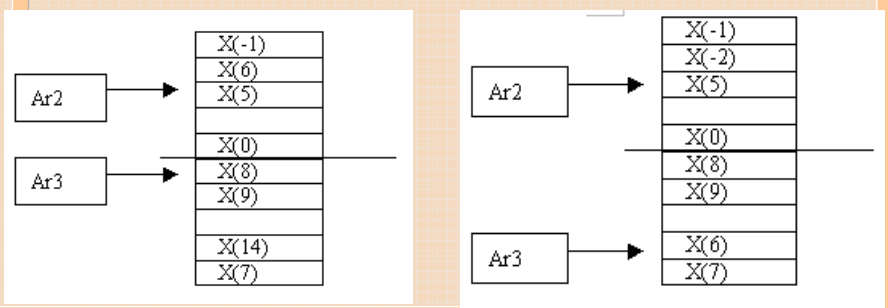
*ar2 → *ar3
new → *ar2++



16点FIRS滤波数据存放

FIR *AR2+0%,*AR3+0%#FIR_COF

AR2--, AR3-=2
*ar2 → *ar3
new → *ar2++



使用FIRS指令完成滤波

- 利用FIRS指令，需要将输入数据缓冲分成两个，大小为N/2。
- 初始状态将AR2指到缓冲区1的顶部，AR3指到缓冲区2的底部。每次滤波之前，应先将缓冲区1顶部的数据移到缓冲区2的底部，新来的一个样本存储到缓冲区1中时，并对缓冲区1指针AR2加1（使用循环寻址）。

使用FIRS指令完成滤波

- 处理器然后使用FIRS指令进行乘加运算。当然，在使用FIRS指令前，需要预先计算一次求和，以初始化A 。
- 在RPTZ重复指令和循环寻址的配合下，完成FIR滤波。
- 滤波完成后，需要对两个数据缓冲的指针进行修正，以便对下一个点进行处理。将Buffer1的指针减1和Buffer2的指针减2，使他们指向各自缓冲的数据队列的最后。

使用带FIRS指令的循环寻址模式实现FIR滤波器，程序片段如下：(输入数据在AL中, 滤波结果在B中)

```
STM   #1,AR0          ;  AR0=1
STM   #（N/2）,BK  ;  BK=N/2,循环寻址BUFFER大小为N
MVDD  *ar2, *ar3      ;  更新Buffer2
STL   A, *ar2+%       ;  更新滤波窗口中的采样数据
ADD   *ar2+0% ,  *ar3+0%      ;  初始化A
RPTZ  B, #(N/2-1)     ;  重复FIRS指令N/2次, 先将B清零
FIRS  *ar2+0%, *ar3+0%,filter_coff+N/2
                  ; 完成滤波计算。注意FIR滤波系数存放
                  ; 在程序存贮区，filter_coff为系数起始地址
MAR   *ar2-%      ;  修改Buffer1指针
MAR   *+ar3(-2)%  ;  修改Buffer2指针
```

§ 5-5 FFT实现

FFT是数字信号处理中重要的工具

- **FFT是一种高效实现离散付氏变换的算法。(P.400 §7.5 && P425 Goertzel算法）**
- **离散付氏变换的目的是把信号由时域变换到频域，从而可以在频域分析处理信息，得到的结果再由付氏逆变换到时域。**
- **DFT的定义为：**

$$X(k)=\sum_{n=0}^{N-1}x[n]e^{-j(\frac{2\pi}{N})nk} \qquad k=0,1,...,N-1$$

DFT的定义

- **可以方便的把它改写为如下形式：**

$$X(k)=\sum_{n=0}^{N-1}x[n]W_N^{nk} \qquad k=0,1,...,N-1$$

- **W_N（旋转因子）的周期性是DFT的关键性质之一。为了强调起见，常用表达式W_N取代W以便明确其周期是N。**

$$W_N^{nk} = e^{-j(\frac{2\pi}{N})nk} = \cos(\frac{2\pi}{N}nk)-j\sin(\frac{2\pi}{N}nk)$$

FFT是DFT的快速算法

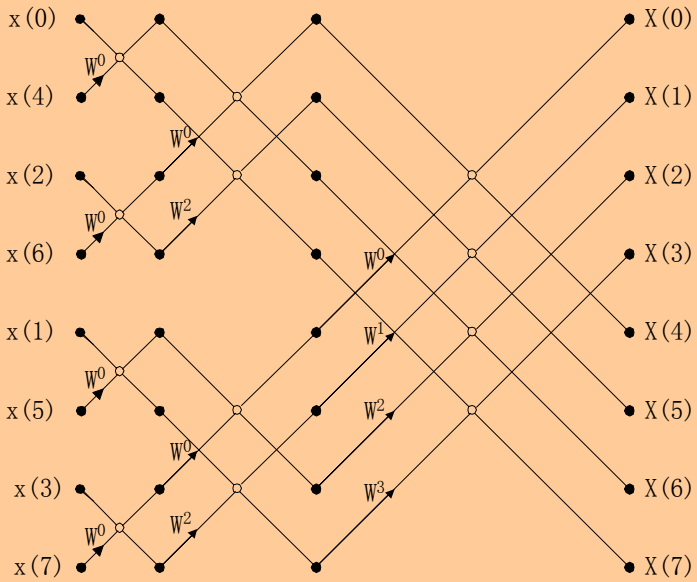
- 由DFT的定义可以看出，在x[n]为复数序列的情况下，完全直接运算N点DFT需要（N-1）^{2}次复数乘法和N（N-1）次加法。
- FFT的基本思想在于，将原有的N点序列序列分成两个较短的序列，这些序列的DFT可以很简单的组合起来得到原序列的DFT。

FFT是DFT的快速算法

- 例如，若N为偶数，将原有的N点序列分成两个（N/2）点序列，那么计算N点DFT将只需要约$[(N/2)^2 \cdot 2]=N^2/2$次复数乘法。即比直接计算少作一半乘法。
- 该处理方法可以反复使用，即（N/2）点的DFT计算也可以化成两个（N/4）点的DFT（假定N/2为偶数），从而又少作一半的乘法。这样一级一级的划分下去一直到最后就划分成两点的FFT运算的情况。

FFT是DFT的快速算法

- 比如，一个N = 8点的FFT运算按照这种方法来计算FFT可以用下面的流程图来表示：

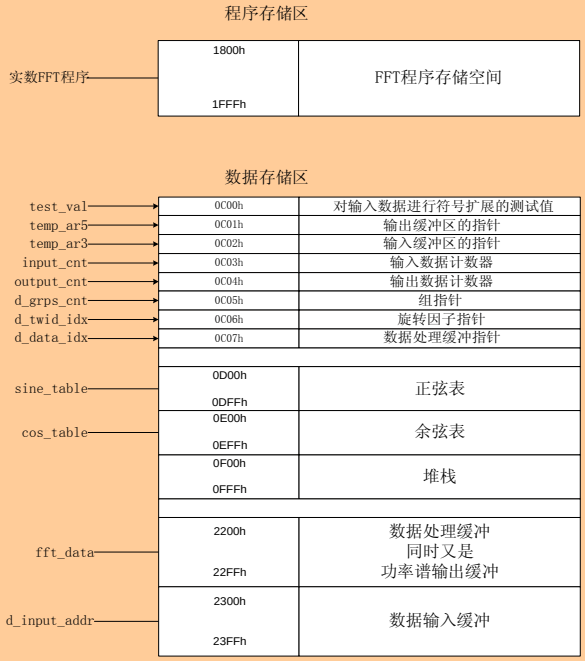


实数FFT运算

- 对于离散傅立叶变换（DFT）的数字计算，FFT是一种有效的方法。一般假定输入序列是复数。当实际输入是实数时，利用对称性质可以使计算DFT非常有效。
- 一个优化的实数FFT算法是一个组合以后的算法。原始的2N个点的实输入序列组合成一个N点的复序列，之后对复序列进行N点的FFT运算，最后再由N点的复数输出拆散成2N点的复数序列。

实数FFT运算

- 使用这种方法，在组合输入和拆散输出的操作中，FFT运算量减半。
- 这样利用实数FFT算法来计算实输入序列的DFT的速度几乎是一般复FFT算法的两倍。
- 本实验就用这种方法实现了一个256点实数FFT（2N = 256）运算。



实数FFT运算序列的存储分配

基二实数FFT运算的算法

- **第一步，输入数据的组合和位倒序：（1）把输入序列作位倒序，是为了在整个运算最后的输出中得到的序列是自然顺序。（2）原始的输入的2N = 256个点的实数序列复制放到标记有“d_input_addr”的相邻单元，当成N = 128点的复数序列d[n]。其中，奇数地址是d[n]的实部，偶数地址是d[n]的虚部。复数序列经过位倒序，存储在数据处理缓冲器中 “fft-data”。**

2200h	
2201h	
2202h	
2203h	
2204h	
2205h	
2206h	
2207h	
2208h	
2209h	
220Ah	
220Bh	
22FFh	
2300h	a[0]
2301h	a[1]
2302h	a[2]
2303h	a[3]
2304h	a[4]
2305h	a[5]
2306h	a[6]
2307h	a[7]
2308h	a[8]
2309h	a[9]
230Ah	a[10]
230Bh	a[11]
23FFh	a[255]

2200h	r[0]=a[0]
2201h	i[0]=a[1]
2202h	r[64]=a[128]
2203h	i[64]=a[129]
2204h	r[32]=a[64]
2205h	i[32]=a[65]
2206h	r[96]=a[192]
2207h	i[96]=a[193]
2208h	r[16]=a[32]
2209h	i[16]=a[33]
220Ah	r[80]=a[160]
220Bh	i[80]=a[161]
22FEh	r[127]=a[254]
22FFh	i[127]=a[255]
2300h	a[0]
2301h	a[1]
2302h	a[2]
2303h	a[3]
2304h	a[4]
2305h	a[5]
2306h	a[6]
2307h	a[7]
2308h	a[8]
2309h	a[9]
230Ah	a[10]
230Bh	a[11]
23FEh	a[254]
23FFh	a[255]

```
bit_rev:
    STM #d_input_addr,ar3 ；在AR3中放入输入地址
    STM #fft_data,ar7     ；在AR7中放入处理后输出的地址
    MVMM DATA_PROC_BUF,ar2 ；AR2中装入第一个位倒序
                           ；数据指针
    STM #K_FFT_SIZE-1,BRC
    STM #K_FFT_SIZE,ar0 ；AR0=输入数据数目的一半(128)
    RPTB bit_rev_end
    MVDD *ar3+,*ar7+    ；将原始输入缓冲中的数据放入到
                        ；位倒序缓冲中去之后输入缓冲
                        ；(AR3)指针加1, 位倒序缓冲(AR2)指
                        ；针也加1
    MVDD *ar3-,*ar7+    ；将原始输入缓冲中的数据放入到
                        ；位倒序缓冲中去之后输入缓冲(AR3)
                        ；指针减1以保证位倒序寻址正确
    MAR *ar3+0B         ；按位倒序寻址方式修改AR3
bit_rev_end:
```

基二实数FFT运算的算法

- 第二步，N点复数FFT：（1）在数据处理缓冲器里进行N点复数FFT运算。由于在FFT运算中要用到旋转因子W_N，它是一个复数。我们把它分为正弦和余弦部分，用Q15格式将它们存储在两个分离的表中。每个表中有128项，对应从0度到180度。因为采用循环寻址来对表寻址，$128 = 2^7 < 2^8$，因此每张表排队的开始地址就必须是8个LSB位为0的地址。

基二实数FFT运算的算法

- 我们把128点的复数FFT分为七级来算，第一级是计算两点的FFT蝶形结，第二级是计算四点的FFT蝶形结，然后是八点、十六点、三十二点六十四点、一百二十八点的蝶形结计算。最后所得的结果表示为：

$$D[k] = F\{d[n]\} = R[k] + j\, I[k]$$

- 其中，R[k]、I[k]分别是D[k]的实部和虚部。

这一步中，实现FFT计算的具体程序如下：

```
;---------stage1 ：计算FFT的第一步，两点的FFT
    .asg AR1,GROUP_COUNTER   ；定义FFT计算的组指针
    .asg AR2,PX      ；定义AR2为指向参加蝶形运算第一个数据的指针
    .asg AR3,QX      ；定义AR3为指向参加蝶形运算第二个数据的指针
    .asg AR4,WR      ；定义AR4为指向余弦表的指针
    .asg AR5,WI      ；定义AR5为指向正弦表的指针
    .asg AR6,BUTTERFLY_COUNTER
                     ；定义AR6为指向蝶形结的指针
    .asg AR7,DATA_PROC_BUF
                     ；定义在第一步中的数据处理缓冲指针
    .asg AR7,STAGE_COUNTER
                     ；定义剩下几步中的数据处理缓冲指针
    pshm st0
    pshm ar0
    pshm bk          ；保存环境变量
```

```
   SSBX SXM          ；开启符号扩展模式
   STM #K_ZERO_BK,BK   ；让BK=0 使 *ARn+0% = *ARn+0
   LD #-1,ASM        ；为避免溢出在每一步输出时右移一位
   MVMM DATA_PROC_BUF,PX ；PX指向参加蝶形结运算的
                         ；第一个数的实部（PR）
   LD *PX,16,A       ；AH := PR
   STM #fft_data+K_DATA_IDX_1,QX   ；QX指向参加蝶形
                     ；结运算的第二个数的实部（QR）
   STM #K_FFT_SIZE/2-1,BRC   ；设置块循环计数器
   RPTBD stage1end-1         ；语句重复执行的范围到地址
                             ；stage1end-1处
   STM #K_DATA_IDX_1+1,AR0   ；延迟执行的两字节的指令
                             ；（该指令不重复执行）
   SUB *QX,16,A,B            ；BH := PR-QR
   ADD *QX,16,A              ；AH := PR+QR
   STH A,ASM,*PX+            ；PR':= (PR+QR)/2
```

```
ST B,*QX+                ;  QR':= (PR-QR)/2
   ||LD *PX,A            ;  AH := PI
SUB *QX,16,A,B           ;  BH := PI-QI
ADD *QX,16,A             ;  AH := PI+QI
STH A,ASM,*PX+0%         ;  PI':= (PI+QI)/2
ST B,*QX+0%              ;  QI':= (PI-QI)/2
   ||LD *PX,A            ;  AH := next PR

stage1end:
```

$$(R[0] + jI[0]) \pm (R[1]+jI[1])*W^0$$

```
; -------Stage 2 ：计算FFT的第二步，四点的FFT
    MVMM DATA_PROC_BUF,PX        ；PX 指向参加蝶形结
                                 ;运算第一个数据的实部（PR）
    STM #fft_data+K_DATA_IDX_2,QX ；QX 指向参加蝶形结
                                 ;运算第二个数据的实部（QR）
    STM #K_FFT_SIZE/4-1,BRC  ；设置块循环计数器
    LD *PX,16,A              ；AH := PR
    RPTBD stage2end-1        ；语句重复执行的范围到地址
                             ；stage1end-1处
    STM #K_DATA_IDX_2+1,AR0；初始化AR0以被循环寻址
;以下是第二步运算的第一个蝶形结运算过程
    SUB *QX,16,A,B           ；BH := PR-QR
    ADD *QX,16,A             ；AH := PR+QR
    STH A,ASM,*PX+           ；PR':= (PR+QR)/2
    ST B,*QX+                ；QR':= (PR-QR)/2
    ||LD *PX,A               ；AH := PI
```

```
    SUB *QX,16,A,B                ;  BH := PI-QI
    ADD *QX,16,A                  ;  AH := PI+QI
     STH A,ASM,*PX+               ;  PI':= (PI+QI)/2
     STH B,ASM,*QX+               ;  QI':= (PI-QI)/2
; 以下是第二步运算的第二个蝶形结运算过程
      MAR *QX+                    ;  QX中的地址加一
      ADD *PX,*QX,A               ;  AH := PR+QI
      SUB *PX,*QX-,B              ;  BH := PR-QI
      STH A,ASM,*PX+              ;  PR':= (PR+QI)/2
      SUB *PX,*QX,A               ;  AH := PI-QR
      ST B,*QX                    ;  QR':= (PR-QI)/2
      ||LD *QX+,B                 ;  BH := QR
      ST A, *PX                   ;  PI':= (PI-QR)/2
      ||ADD *PX+0%,A              ;  AH := PI+QR
      ST A,*QX+0%                 ;  QI':= (PI+QR)/2
      ||LD *PX,A                  ;  AH := PR
stage2end:
```

```
; Stage 3 thru Stage logN-1：从第三步到第六步的过程如下
(略）
```

基二实数FFT运算的算法

- 第三步，分离复数FFT的输出为奇部分和偶部分： 分离FFT输出为相关的四个序列：RP、RM、IP和IM，即偶实数，奇实数、偶虚数和奇虚数四部分，以便第四步形成最终结果。

利用信号分析的理论我们把D[k]通过下面的公式分为偶实数RP[k]、奇实数RM[k]、偶虚数IP[k]和奇虚数IM[k]：

RP[k] = RP[N-k] = 0.5 * (R[k] + R[N-k])
RM[k] = -RM[N-k] = 0.5 * (R[k] - R[N-k])
IP[k] = IP[N-k] = 0.5 * (I[k] + I[N-k])
IM[k] = -IM[N-k] = 0.5 * (I[k] - I[N-k])
RP[0] = R[0]
IP[0] = I[0]
RM[0] = IM[0] = RM[N/2] = IM[N/2] = 0
RP[N/2] = R[N/2]
IP[N/2] = I[N/2]

这一过程的程序代码(略）

基二实数FFT运算的算法

- 第四步，产生最后的N = 256点的复数FFT结果:产生2N = 256个点的复数输出，它与原始的256个点的实输入序列的DFT一致。输出驻留在数据缓冲器中。

通过下面的公式由RP[k]、RM[n]、IP[n]和IM[n]四个序列可以计算出a[n]的DFT：

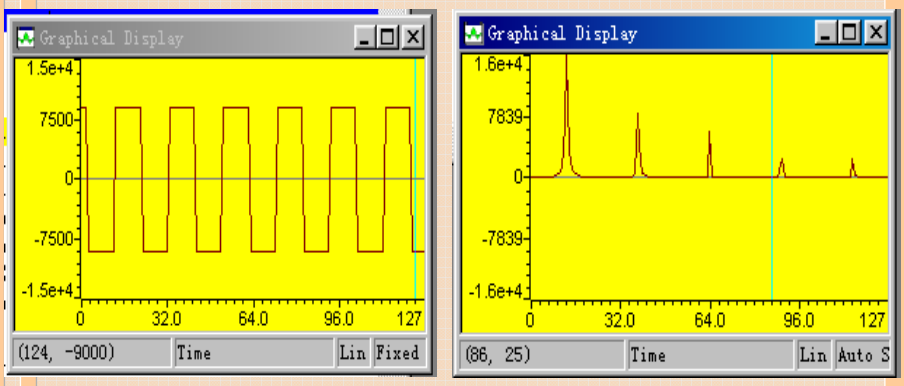
AR[k] = AR[2N-k] =
RP[k] + cos(k π /N) * IP[k] - sin(k π /N) * RM[k]
AI[k] = -AI[2N-k] =
IM[k] - cos(k π /N) * RM[k] - sin(k π /N) * IP[k]
AR[0] = RP[0] + IP[0]
AI[0] = IM[0] - RM[0]
AR[N] = R[0] - I[0]
AI[N] = 0
其中：
A[k] = A[2N-k] = AR[k] + j AI[k] = F{a(n)}

这一过程的程序代码(略）

基二实数FFT运算的算法

- 计算所求信号的功率：由于最后所得的FFT数据是一个复数，为了能够方便的在虚拟频谱仪上观察该信号的特征，我们通常对所得的FFT数据进行处理——取其实部和虚部的平方和，即求得该信号的功率。

CCS中的图形工具显示FFT结果



§5-6 LMS实现

最小均方运算LMS

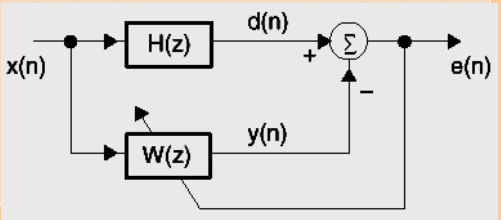
在进行自适应滤波等操作中经常会使用LMS算法，C54x提供的LMS指令方便了编程。如下图所示的自适应滤波器设计中，滤波器系数修正公式为：

$$w_k(i+1) = w_k(i) + 2\beta e(i)x(i)$$

其中，e(i) = d(i) - y(i)。

滤波器输出：

$$y(i) = \sum_{k=0}^{N-1} w_k x(i-k)$$



LMS指令执行过程

- **LMS Xmem, Ymem**

  (A)+(Xmem) <<16+2^{15} ---> A

  (B)+(Xmem) x (Ymem) ---> B

  (*xmem->系数 *ymem -> 输入数据)

- **LMS模块使用LMS、ST||MPY和RPTBD指令计算滤波器输出和更新滤波器的每个系数。**

LMS算法的运算步骤如下

- **初始化设置W（i）（i=0,1,2…,N-1）为任意值，例如均为0。然后对k=1，2，…的每次采样，作以下各步的循环运算**
- **计算滤波器输出：**

$$y_k = \sum_{i=0}^{N-1} w_k(k) x_{k-i}$$

LMS算法的运算步骤如下

- 计算误差估计:

$$e_k = d_k - y_k$$

- 更新N个滤波器权重系数，（**i = 0, 1, …, N -1**）

$$w_{j+1}(i) = w_j(i) + 2\,\mu\, e_j\, x_{j-i}$$

- 重复上述过程

基于LMS的自适应滤波编程

```
   STM     #N-2, BRC      ;initialize the Block
                          ; repeat counter with Store MMR
   LD      ERROR,T        ;initialize a variable
                          ; called ERROR to 0
   RPTBD   END_LOOP-1     ;establish last line of
                          ; loop
   MPY     *AR4,A         ;delay slot instruction
                          ; that initializes accumulator A
   LMS     *AR3,*AR4+0%   ;2nd delay slot instruction
                          ;loop starts here
   ST      A,*AR3+        ;save filter coefficient
|| MPY     *AR4,A         ;new term calculated
   LMS     *AR3, *AR4+0%  ;LMS instruction for
                          ; adaptive filter
                          ;A = A + *AR3<<16 + 2^15
                          ;B = B + *AR3 x *AR4
END_LOOP:
```