FIXED-POINT DSP PROGRAMMING USING THE ADSP-2100

Somsak Sukittanon
{ssukitta}@ee.washington.edu

Dept of EE, University of Washington
Seattle WA, 98195-2500
FIXED-POINT DSP PROGRAMMING USING THE ADSP-2100

Somsak Sukittanon

April, 2003

Department of Electrical Engineering, University of Washington, Seattle, WA

INTRODUCTION

The DSP microprocessor used in EE443 spring 2003 at the University of Washington is the TMS320C6711 (or C31 in the past). The coding is mainly based on floating point programming. Alternatively, fixed point DSP programming is also commonly used in the industry. This note will review some fixed-point concepts, from number representation to several real-time applications in DSP. Analog Devices’ DSP chip will be used in the discussion and compared to the C67 and C31. Students who read these notes should understand the big picture of fixed-point representation, the architecture, instruction sets, stack, circular buffer, and C-callable of the ADSP-2100. This note will not cover compiler, simulator, and interrupt. For more information on the ADSP manual, visit this Analog Device website:


1. FIXED-POINT REPRESENTATION

First, we need to understand the basic idea of the fixed point system. For both 16 bits and 32 bits, the numbers are represented in integer format and the values are between -1 and 1 for signed mode or 0 and 1 for unsigned mode. To see the dynamic range of an N-bit number, we can use help command in Matlab. For example, >> help uint16 or help int16 for unsigned 16 bits and signed 16 bits information, respectively. Note that the ADSP discussed is the 16 bit fixed point DSP. For a 16 bit system, the range is between \((-2^{16})\) and \((2^{16} - 1)\), or between -32768 and 32767 for signed mode. For unsigned mode, the range of integer numbers is between 0 and \(2^{16} - 1\), or between 0 and 65535.

1.1 16 bit signed mode

For 16 bit signed mode, the number will be between –1 and 1 in decimal format or 0x0000 – 0xFFFF in hex format where 0x0000 – 0x7FFF represent the positive number (0 to 1) and 0x8000 – 0xFFFF represent the negative number (-1 to 0). The binary format is shown in Figure 1. In 2’s complement, if the MSB (Most significant bit, \(2^0\)) is 1 the number will represent the negative (0x8?? – 0xF???) number. Now let say we have 2 numbers, positive and negative, 0.1234 and –0.5678, and we want to represent them in 16 bit signed mode.

\[
\begin{array}{cccccc}
-2^0 & 2^1 & \cdots & 2^{14} & 2^{15} \\
\end{array}
\]

Figure 1: Bit format of 1.15 (16 bits) signed mode

For a positive number, we multiply the number by \(2^{15} – 1\) (32,767), round the scaled number to be an integer, and change it from a decimal number to a hex number. For example, 0.1234 x 32,767 = 4,043.45 → 4,043. In hex mode 4,043 is FCB or 0x0FCB. Using this approach, we can determine that 0x7FFF = 1 and 0x0000 = 0.

For a negative number, first we add 2 to the number then multiply by \(2^{15}\) (32,768) and change it to hex format. For example, –0.5678+2 = 1.4322 → 1.4322*32,768 = 46,930.3 → 46,930. In hex this will be 0xB752. You can see that if the number is negative, the number after adding 2 will be more than 32,767 or 7FFF. That’s why the leftmost number is always more than 8 for negative numbers.
Unlike positive numbers, negative numbers start from 0x8000 – 0xFFFF. Next we want to write Matlab and Microsoft Excel to do this task. Commonly, the author used Microsoft Excel to find the fixed point representation.

Example 1.1

We want to change floating numbers called cos16 and sin16 to fixed point representation. Table 1 shows the final results of example 1.1. Note the fixed point values of -1, 0, and 1. Also note the MSB when the number is negative.

Table 1: The floating point and 16 bit fixed point representation of cos16 and sin16 numbers using Matlab

<table>
<thead>
<tr>
<th></th>
<th>Cos16</th>
<th></th>
<th>Sin16</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>floating</td>
<td>fixed</td>
<td>floating</td>
<td>fixed</td>
<td></td>
</tr>
<tr>
<td>0.9239</td>
<td>0x7641</td>
<td>0.3827</td>
<td>0x30FB</td>
<td></td>
</tr>
<tr>
<td>0.7071</td>
<td>0x5A82</td>
<td>0.7071</td>
<td>0x5A82</td>
<td></td>
</tr>
<tr>
<td>0.3827</td>
<td>0x30FB</td>
<td>0.9239</td>
<td>0x7641</td>
<td></td>
</tr>
<tr>
<td>0.0000</td>
<td>0x0000</td>
<td>1.0000</td>
<td>0x7FFF</td>
<td></td>
</tr>
<tr>
<td>-0.3827</td>
<td>0xCF04</td>
<td>0.9239</td>
<td>0x7641</td>
<td></td>
</tr>
<tr>
<td>-0.7071</td>
<td>0xA57E</td>
<td>0.7071</td>
<td>0x5A82</td>
<td></td>
</tr>
<tr>
<td>-0.9239</td>
<td>0x89BE</td>
<td>0.3827</td>
<td>0x30FB</td>
<td></td>
</tr>
<tr>
<td>-1.0000</td>
<td>0x8000</td>
<td>0.0000</td>
<td>0x0000</td>
<td></td>
</tr>
<tr>
<td>-0.9239</td>
<td>0x89BE</td>
<td>-0.3827</td>
<td>0xCF04</td>
<td></td>
</tr>
<tr>
<td>-0.7071</td>
<td>0xA57E</td>
<td>-0.7071</td>
<td>0xA57E</td>
<td></td>
</tr>
<tr>
<td>-0.3827</td>
<td>0xCF04</td>
<td>-0.9239</td>
<td>0x89BE</td>
<td></td>
</tr>
<tr>
<td>0.0000</td>
<td>0x0000</td>
<td>-1.0000</td>
<td>0x8000</td>
<td></td>
</tr>
<tr>
<td>0.3827</td>
<td>0x30FB</td>
<td>-0.9239</td>
<td>0x89BE</td>
<td></td>
</tr>
<tr>
<td>0.7071</td>
<td>0x5A82</td>
<td>-0.7071</td>
<td>0xA57E</td>
<td></td>
</tr>
<tr>
<td>0.9239</td>
<td>0x7641</td>
<td>-0.3827</td>
<td>0xCF04</td>
<td></td>
</tr>
<tr>
<td>1.0000</td>
<td>0x7FFF</td>
<td>0.0000</td>
<td>0x0000</td>
<td></td>
</tr>
</tbody>
</table>

```matlab
x = exp(-j*2*pi*(15:-1:0)/16);
COS16_DECIMAL = real(x);
SIN16_DECIMAL = imag(x);
for ind = 1:length(COS16_DECIMAL)
    COS16_HEX(ind,:) = getDEC2HEX(COS16_DECIMAL(ind));
    SIN16_HEX(ind,:) = getDEC2HEX(SIN16_DECIMAL(ind));
end
function result = getDEC2HEX(x);
result = '0x0000';
if x > 0
    x = round(x*(2^15-1));
elseif x < 0
    x = round((2+x)*(2^15));
end
```
end
x = dec2hex(x);
if length(x) > 4
    x = x(length(x)-4+1:length(x));
end
result(6-length(x)+1:6) = x;

Figure 2: The Matlab code for changing floating point to 16 bit fixed point numbers

\[
= \text{ROUND}((B1\times2^{15}-1),0.5)
\]
\[
=\text{ROUND}(((B2+2)\times2^{15}),0.5)
\]

B  C  D
1  0.1234  4043  0x0FCB
2  -0.5678  46930  0xB752

Decimal number  scaled decimal  16 bits signed

Figure 3: The formula in Excel spreadsheet for changing floating point to 16 bit fixed point numbers

Table 2: The floating point and 16 bit fixed point representation of cos16 and sin16 numbers using Microsoft Excel

<table>
<thead>
<tr>
<th>Cos16</th>
<th>Sin16</th>
</tr>
</thead>
<tbody>
<tr>
<td>floating</td>
<td>Integer (x2^{15}-1)</td>
</tr>
<tr>
<td>0.9239</td>
<td>30273</td>
</tr>
<tr>
<td>0.7071</td>
<td>23170</td>
</tr>
<tr>
<td>0.3827</td>
<td>12539</td>
</tr>
<tr>
<td>0.0000</td>
<td>0</td>
</tr>
<tr>
<td>-0.3827</td>
<td>-12539</td>
</tr>
<tr>
<td>-0.7071</td>
<td>-23170</td>
</tr>
<tr>
<td>-0.9239</td>
<td>-30273</td>
</tr>
<tr>
<td>-1.0000</td>
<td>-32767</td>
</tr>
<tr>
<td>-0.9239</td>
<td>-30273</td>
</tr>
<tr>
<td>-0.7071</td>
<td>-23170</td>
</tr>
<tr>
<td>-0.3827</td>
<td>-12539</td>
</tr>
<tr>
<td>0.0000</td>
<td>0</td>
</tr>
<tr>
<td>0.3827</td>
<td>12539</td>
</tr>
<tr>
<td>0.7071</td>
<td>23170</td>
</tr>
<tr>
<td>0.9239</td>
<td>30273</td>
</tr>
<tr>
<td>1.0000</td>
<td>32767</td>
</tr>
</tbody>
</table>
In Figure 3, B’s column is the floating point number, C’s column is the scaled integer for 16 bits, and D’s column is the fixed point number. It may be difficult to separate positive and negative numbers, so we can multiply by $2^{15} - 1$ for all numbers. There is a small rounding effect from this method. You can see that -1 becomes 0x8001 instead of 0x8000.

### 1.2 32 bit signed mode

For 32 bit signed mode, the steps are very similar to previous discussion. Use `help int32` or `help uint32` in Matlab to see the dynamic range. The range of 32 bit signed mode is between $-\left(2^{32-1}\right)$ and $\left(2^{32-1} - 1\right)$. Everything will be scaled by $2^{31}$ instead of $2^{15}$. Now let’s do the same example of 2 positive and negative numbers, 0.1234 and –0.5678, and we want to represent in 32 bit signed mode.

\[
\begin{align*}
0.1234 \times (2^{31} - 1) &= 264,999,482 \Rightarrow 0x0FCB \text{923A} \\
(2-0.5678) \times (2^{31}) &= 3,075,626,081 \Rightarrow 0xB752 \text{5461}
\end{align*}
\]

We can see that the first 4 numbers are the same as 16 bits but we have 4 more in 32 bits. Let’s do the example 1.1 again using Microsoft Excel. We want to generate 32 bit number of cos16. Figure 4 is the formula to calculate the high (first 4) and the low (last 4) of fixed point 32 bits. Table 3 shows 32 bit fixed point values of cos16. Notice the value of -1, 1, and 0 again. At this point, we should understand how to change the number between -1 and 1 to fixed point, either 16 or 32 bits. Next, we will discuss the fixed point DSP based on ADSP-2100 family. We will use this number representation in the coding.

![Figure 4: The formula in Microsoft Excel for changing floating point to 32 bit fixed point numbers](image-url)
Table 3: The floating point and 32 bit fixed point representation of cos16 numbers using Microsoft Excel

<table>
<thead>
<tr>
<th>Cos16</th>
<th>Floating</th>
<th>Integer ( (x2^{31}-1) )</th>
<th>Fixed (High)</th>
<th>Fixed (Low)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9239</td>
<td>1984016188</td>
<td>0x7641</td>
<td>0xAF3C</td>
<td></td>
</tr>
<tr>
<td>0.7071</td>
<td>1518500249</td>
<td>0x5A82</td>
<td>0x7999</td>
<td></td>
</tr>
<tr>
<td>0.3827</td>
<td>821806413</td>
<td>0x30FB</td>
<td>0xC54D</td>
<td></td>
</tr>
<tr>
<td>0.0000</td>
<td>0</td>
<td>0x0000</td>
<td>0x0000</td>
<td></td>
</tr>
<tr>
<td>-0.3827</td>
<td>-821806413</td>
<td>0xCF04</td>
<td>0x3AB3</td>
<td></td>
</tr>
<tr>
<td>-0.7071</td>
<td>-1518500249</td>
<td>0xA57D</td>
<td>0x8667</td>
<td></td>
</tr>
<tr>
<td>-0.9239</td>
<td>-1984016188</td>
<td>0x89BE</td>
<td>0x50C4</td>
<td></td>
</tr>
<tr>
<td>-1.0000</td>
<td>-2147483647</td>
<td>0x8000</td>
<td>0x0001</td>
<td></td>
</tr>
<tr>
<td>-0.9239</td>
<td>-1984016188</td>
<td>0x89BE</td>
<td>0x50C4</td>
<td></td>
</tr>
<tr>
<td>-0.7071</td>
<td>-1518500249</td>
<td>0xA57D</td>
<td>0x8667</td>
<td></td>
</tr>
<tr>
<td>-0.3827</td>
<td>-821806413</td>
<td>0xCF04</td>
<td>0x3AB3</td>
<td></td>
</tr>
<tr>
<td>0.0000</td>
<td>0</td>
<td>0x0000</td>
<td>0x0000</td>
<td></td>
</tr>
<tr>
<td>0.3827</td>
<td>821806413</td>
<td>0x30FB</td>
<td>0xC54D</td>
<td></td>
</tr>
<tr>
<td>0.7071</td>
<td>1518500249</td>
<td>0x5A82</td>
<td>0x7999</td>
<td></td>
</tr>
<tr>
<td>0.9239</td>
<td>1984016188</td>
<td>0x7641</td>
<td>0xAF3C</td>
<td></td>
</tr>
<tr>
<td>1.0000</td>
<td>2147483647</td>
<td>0x7FFF</td>
<td>0xFFFF</td>
<td></td>
</tr>
</tbody>
</table>

2. ADSP-2100 DSP

In this section, we will briefly review the Texas Instrument DSP chip. Then, we discuss the architecture of the Analog Device DSP chip. The registers, memory addressing, circular buffer, and C-callable for the ADSP-2100 will be compared to the C67 and C31.

2.1 C67 and C31

For TMS320C6711, there are 32 registers, A0-A15 and B0-B15. Besides holding data and address, some registers are assigned to specific tasks such as circular buffering or subroutine calling. There are 4 functional units, .M, .S, .L, and .D, used for multiply, arithmetic, and logic operations. For TMS320C31, it is much simpler. There are 3 sets of 8 registers, AR0-AR7, for holding address, R0-R7 for holding data, and I0-I7 for indexing. There are no separate functional units like for the C67.

2.2 ADSP-2100

The ADSP-2100 is in between the C67 and C31. There are several sets of registers such as the address register, indexing register, and length register for circular buffering. There are 3 functional units called the ALU, MAC, and SHIFTER for arithmetic, multiply, and logic operations, respectively. In each unit, there are specific registers assigned for calculation.

2.3 ALU (Arithmetic Logic Unit)

This unit is responsible for addition, subtraction, AND/OR/XOR, increment, decrement, negate, clear, etc. However, there is no multiplication in this unit. The assigned registers will start with the letter A, for example AX0, AY0 or AR. First, let’s do an easy example of addition, 5+6 = 11. Normally there are 2 inputs and one output. Input registers, AX? and AY? are mainly used for holding input and AR is used to hold output, \((R = \text{result})\). If you read the online manual, the format is shown in this way.

Syntax: \[ AR = \text{Xop + Yop} \]

AF
xop = AX0, AX1, AR etc.
yop = AY0, AY1, AF

Notes:
- The result is always AR or AF.
- The inputs are AX? and AY?.

Example codes of ALU

\[
\begin{align*}
AR &= AR + AF; \\
AF &= AR - AY0 \\
AR &= AX1 - AY1; \\
AR &= AX0 \text{ XOR } AY0 \\
AF &= \text{ABS } AX0 \\
AF &= -AY0 \\
AR &= AX0 + 1;
\end{align*}
\]

2.4 MAC (Multiplier/Accumulator)
This unit is responsible for multiplication or multiplication and addition (used in filtering). The assigned registers will start with the letter M, for example, MX0, MY0, MR0 etc. Let’s see an example of multiplication, 2 x 5 = 10. MX? and MY? are mainly used for holding input data and MR(MR2, MR1, MR0) is used to hold the result. The MR has to be twice as big as the input in order to save the result. If each input is 16 bits the result register should be 32 bits. In the ADSP-21xx, MR is 40 bits. In the manual, the instruction is shown in this way.

Syntax: \( MR = \text{xop} \times \text{yop} \text{(SS)} \)

\[
\begin{align*}
xop &= \text{MX0, MX1, MR0, MR1, MR2 etc.} \\
yop &= \text{MY0, MY1, MF}
\end{align*}
\]

Notes:
- Either MR or MF is used for holding the result. AR is only the ALU register used in MAC.
- SS means signed (for xop) and signed (for yop) multiplication. If RND, the 16 bits MSB will be saved into MR1.

Example codes of MAC

\[
\begin{align*}
MR &= \text{MX0} \times \text{MY0} \text{(SS)}; \\
MR &= \text{MR} - \text{MX1} \times \text{MY1} \text{(SS)}; \\
MR &= \text{MR} \text{(RND)}; \\
MR &= \text{AR} \times \text{MY1} \text{(SS)}; \\
MR &= \text{MR} + \text{MX0} \times \text{MY1} \text{(US)}; \\
MR &= \text{MR} + \text{MX1} \times \text{MY0} \text{(SS)}; \\
MR &= \text{MR} + \text{MX1} \times \text{MY0} \text{(SU)}; \\
MR &= \text{MR} + \text{MX1} \times \text{MY1} \text{(SS)};
\end{align*}
\]
2.5 SHIFTER
This unit is responsible for shifting. For example, in IIR filtering, we may have a pregain value (power of 2). At the end of the process we have to multiply postgain back and check overflow. To do multiplication, we can simply shift left (or right) instead. Shifting left one bit is equal to multiplying by 2 and shifting right one bit is equal to dividing by 2. The assigned register will start with the letter S.

Example codes of SHIFTER

```
SE   = MY0;
SR   = ASHIFT MR1 (HI)
```

Note:
- SE will hold the bits that we want to shift.
- SR will hold the result after shifting the value in MR.

2.6 For loop, Jump, If –cause
When we use for-loop in C31, there is a counter register, called RC, for tracking the number of iterations. For the C61, conditional registers, A0-A1 and B0-B2 can be used for tracking the number of iterations.

For the ADSP-2100, CNTR register is used to track the number of iterations. The coding format is similar to C programming. Let’s see the example.

```
...                        /* Call command */
MY1=AR;
CALL cordic;
...

...                        /* Jump command */
AF = PASS MR0;
IF NE JUMP initialized;
...

...                        /* Forloop command */
CNTR=128;
DO convolution UNTIL CE;
    MF=MX0*MY0(SS), MX0=PM(I6,M5);
    convolution: MR=MR+MX0*MF(SS), MX0=DM(I1,M1), MY0=PM(I5,M6);
...
```

Notes:
- Observe the difference of commenting in C67 and ADSP-2100
- The end of instruction is finished with “;” like Matlab.

2.7 Loading data from memory/ storing data to memory
This section describes how we move data between register/memory and register/memory. To load data between registers, ^ is used for pointer and % is used for the length.

```
...                       .VAR/PM  cosnsin[30];
.INIT                     cosnsin[0]: 0x5A82, 0x5A82, 0x7641, 0x30FB, 0x7D8A, 0x18F8, 0x7F62, 0x0C8B,
                           0x7FD8, 0x0647;
.INIT                     cosnsin[10]: 0x7FF6, 0x0324, 0x7FFD, 0x0192, 0x7FFF, 0x00C9, 0x7FFF, 0x0064,
                           0x7FFF, 0x0032;
```
.INIT  
cosnsin[20]: 0x7FFF, 0x0019, 0x7FFF, 0x000C, 0x7FFF, 0x0006, 0x7FFF, 0x0003,
0x7FFF, 0x0001;

<Entry>  
mag2_;  

mag2_:  
MR1=TOPPCSTACK;  
CALL ___lib_save_small_frame;  

I6=^cosnsin;  /* I6 has the start address of cosnsin array */  
L6=%cosnsin;  /* L6 has the length */  
AY0=0x8000;  
AR=MX0;  

Notes:  
• PM stands for program memory.  
• Notice how to initialize the constant.  
• I6 contains the starting address of the cosnsin variable.  
• L6 contains the length of the cosnsin variable, 30 in this case.  
• Notice how to load memory into a register and transfer a value between registers. In this case, AR will have the same value as  
MX0 afterward.  

DAG (data address register) includes DAG1 and DAG2. DAG1 is used only for DM (data memory). DAG2 is used for both DM and PM  
(program memory). In DAG1 or DAG2, there are 12 registers that we can use. We cannot mix the regiser between DAG1 and DAG2.  

Table 4: Data address registers for data memory and program memory

<table>
<thead>
<tr>
<th></th>
<th>DAG1</th>
<th>DAG2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index Registers</td>
<td>I0 – I3</td>
<td>I4-I7</td>
</tr>
<tr>
<td>Modify Registers</td>
<td>M0-M3</td>
<td>M4-M7</td>
</tr>
<tr>
<td>Length Registers</td>
<td>L0-L3</td>
<td>L4-L7</td>
</tr>
</tbody>
</table>

For direct and indirect reading/writing of data and program memory, we can use DAG1 or DAG2 but we cannot mix between DAG1 and  
DAG2. For the indirect mode, the I register will post-modify based on the number in the M register. In C/C++, x = *ptr++, the ptr will be  
increment after the loading.  

For reading:

Syntax:  
reg = DM(<address>);  /* direct */  
dreg = DM (I?,M?);  
dreg = PM (I?,M?);  

For writing, we just switch the left command to the right command.

Syntax:  
DM (<address>)=reg ;  /* direct */  
DM (I?,M?)=dreg;  
PM (I?,M?)=reg;
Let’s see an example for this section.

\[ I_6 = \text{cosnsin}; \]
\[ M_5 = 1; \]
\[ MY_1 = PM(I_6, M_5); \]

\[ I_6 = \text{MR}_1; \] /*Get the oldest delay*/
\[ M_5 = AX_0; \]
\[ \text{modify}(I_6, M_5); \]

\[ MR_0 = DM(I_6, M_6); \]
\[ AF = \text{PASS} MR_0; \]
\[ \text{IF NE JUMP} \text{ initialized}; /*Yes, skip*/ \]
\[ DM(I_6, M_6) = MR_1; /*No, Save ptr of delay line*/ \]

initialized:
\[ M_5 = AX_1; /*select type of filter*/ \]
\[ MR_1 = DM(I_6, M_6); /*Fetch pointer to delay line*/ \]

load:
\[ M_5 = 1; \]
\[ \text{MODIFY}(I_6, M_6); \]
\[ AR = PM(I_6, M_5); \]
\[ PM(I_5, M_5) = AR; \]
\[ AR = PM(I_6, M_5); \]
\[ PM(I_5, M_5) = AR; \]

Notes:
- See how to indirectly load the data in memory into a register.
- See how to indirectly save the data in a register into memory.
- See the difference between loading from data memory (such as temporal results) and program memory (such as constant value).

Like the C67 or C31, the ADSP also has a multifunctional feature. We can load/save data while using the ALU/MAC/Shifter in one clock cycle. This feature is useful when we want to perform multiplying and summation like convolution. Let’s see an example.

\[ MR_0 = 0, MX_0 = DM(I_1, M_1), MY_0 = PM(I_5, M_6); \]
\[ CNTR = AX_0; \]
\[ \text{DO convolution UNTIL} CE; \]
\[ MF = MX_0 \ast MY_0(\text{SS}), MX_0 = PM(I_6, M_5); \]

convolution:
\[ MR = MR + MX_0 \ast MF(\text{SS}), MX_0 = DM(I_1, M_1), MY_0 = PM(I_5, M_6); \]

\[ MR = SR_0 \ast MY_0(\text{UU}), MY_0 = PM(I_5, M_4); \]
\[ MR = MR + MX_0 \ast MY_0(\text{UU}), MY_0 = PM(I_5, M_6); \]
\[ MR = MR + AR \ast MY_0(\text{UU}), MY_0 = PM(I_5, M_4); \]
2.8 C-callable

Recall from the EE443 lecture, assembly coding is quite difficult to debug but its efficiency is high. For C programming, it is easier but less optimized than pure assembly. C-callable is the way that we write a subfunction for calculation in assembly code and write the main function that passes the variables in C code. The main question is how to pass input and output from C-callable.

For the C67, the registers A4, B4, A6 are used for holding input variables. A4 is also used to hold the output value if it is a scalar. However, if the output is a vector, A4 is used to hold the address. For the ADSP-2100, AR and AY1 are used to hold the input and AR is used to hold the output. If the input has more than 2 variables, the 3rd variable and so on will be on the stack. Figure 5 shows the big picture of C-callable for the ADSP. Notice the file extension difference between a main function and a subfunction.

Let's see the partial code of main.c. The system after initialization is in the idle state until the interrupt occurs every sampling period. When the interrupt occurs, the new sample will be retrieved into inLeftHigh, inLeftLow, inRightHigh, and inRightLow variables. The values in outLeftHigh and outLeftLow are sent out to the codec. Compared to the C61, the input is retrieved from the input_sample() function and the output is sent out using the output_sample() function. Between retrieving input and sending output is our DSP code such as filtering or absolute square.

```c
#include "\inc\myproj.h"
#include <def2181.h>
extern int inLeftHigh;
extern int inLeftLow;
extern int outLeftHigh;
extern int outLeftLow;
extern int maglowpass;
void new_sample();
void sync_isr();
int output[2];
int output_ptr;
int main()
{
    SystemInit();
    while(1)
    {
        asm("IDLE; ");
        // assign pointer to output
    }
}
```
After the interrupt, input are in inRightHigh and inRightLow. Assume we process filtering and the outputs are in output [0] and output [1].

```c
realpart = hmfir(inLeftHigh, costaps, delayFIR5_ptr, taps, skip);
imagpart = hmfir(inLeftHigh, sintaps, delayFIR6_ptr, taps, skip);
maglowpass = mag2(realpart, imagpart);
```

```c
outLeftHigh = output[0];
outLeftLow = output[1];
```

Comments on writing main.c:

- Header files contain constant values such as filter coefficients. Header files also contain subfunction definitions. The output of the subfunction can be scalar (int ...) or vector (void ...).
- Can declare global or local variables. For variables will be declared in other files, we use extern with the declaration.
- SystemInit() is used to initialize the interrupt flag, clear the delay taps, and set input and output parameters.
- After initialization, the process starts. The system will be in an idle state until there is an interrupt which occurs every 1/Fs. The system will jump to an ISR to retrieve the new 32 bits of input data from the input port and save into inLeftHigh and inLeftLow (each variable is 16 bits) and send the output into the output port at the same time. Note that the output is delayed by one period. To do filtering, the new data will be saved into the circular buffer.
- Our DSP algorithm will start after we have the new sample. In this example, FIR filtering is performed and followed by the absolute of the complex number.
- After finishing all processes, the output samples have to be in outLeftHigh and outLeftLow. Then the process goes back to an idle state again for the next turn.
```c
int SystemInit()
{
    ISR_Init();
    flag = 0;
    ...
    asm("IFC = 0x3f; "); /* clear any pending Interrupts */
    ...
    asm("IMASK = 0x040; "); /* enable only transmitter */
    ...
    clrdelay(); /* initial FIR and IIR delay lines */
}
```

**Figure 8: SystemInit function in Init.c**

**Comments on writing subfunction in C-callable:**

- Start the function with .MODULE and end with .ENDMOD
- Start the code with .ENTRY
- Set stack pointer (MR1=TOPPCSTACK) and start to retrieve the input parameters. The first input will always be in AR and the second will be in AY1. If there are more than two parameters, we can retrieve the third and so on from stack. MR1, AX0, AX1 in the code example contain the third to fifth input parameters.
- The output will be in AR. If it is a scalar, AR contains a value. If it is a vector, AR contains the address of the first output in program memory (PM).
- After we retrieve the input parameters, we are almost finished. The rest will be our DSP algorithms.
- Observe how to call another function in assembly code.
- The following code is the example of the absolute function using CORDIC
Figure 9: C-callable function, mag2.dsp
3. DSP ALGORITHMS

Understanding DSP algorithms is critical in implementing real-time DSP. For example, if we want to do spectrum analysis, FFT is commonly used for this task. But if we care about only a few frequencies such as 1 KHz or 2 KHz, we should use another implementation to save cycles. Another example is an implementation of a transform such as DCT or MLT. With FFT toolbox, we can easily relate these transforms to the FFT. This section will review the DSP concepts and implementation of many algorithms.

3.1 CORDIC

When we deal with a complex number such as the result from the FFT or Hilbert transform, and we would like to find the absolute of the complex number and the phase using only multiply and summation. CORDIC is the option used for this task. CORDIC is done by rotating the vector to the x-axis so the real number will be the absolute and the accumulated angle will be the phase of the vector.

Original vector:

\[ X_r + jX_i = |X| e^{j\theta}, \]

Clockwise rotation by \( \theta_{\text{shift}} \):

\[
X'_r + jX'_i = |X| e^{j\theta_{\text{shift}}} e^{-j\theta_{\text{approx}}}; \text{ clockwise (sign(accumulated angle) = +)}
\]

\[
= (X_r + jX_i)(\cos \theta_{\text{shift}} - j \sin \theta_{\text{shift}}) \\
= (X_r \cos \theta_{\text{shift}} + X_i \sin \theta_{\text{shift}}) + j(X_i \cos \theta_{\text{shift}} - X_r \sin \theta_{\text{shift}})
\]

Counter clockwise rotation by \( \theta_{\text{shift}} \):

\[
X'_r + jX'_i = |X| e^{j\theta_{\text{shift}}} e^{j\theta_{\text{approx}}}; \text{ counterclockwise (sign(accumulated angle) = -)}
\]

\[
= (X_r + jX_i)(\cos \theta_{\text{shift}} + j \sin \theta_{\text{shift}}) \\
= (X_r \cos \theta_{\text{shift}} - X_i \sin \theta_{\text{shift}}) + j(X_i \cos \theta_{\text{shift}} + X_r \sin \theta_{\text{shift}})
\]
Final answer after N iteration

\[ |X| e^{j\theta_x} e^{-j\theta_{app}} \approx |X| \]
\[ \approx \text{Re}(|X| e^{j\theta_x} e^{-j\theta_{app}}) \]
\[ \theta_\alpha \approx \theta_{\text{shift}} \]
\[ \approx \sum_{i=1}^{\text{Iterations}} \theta_i \]

Implementation

- Determine the number of iterations such as 8 or 16.
- Compute the constant values of cosine and sine with angle starting from 45 degree and decreasing by half every step.
- Start the iteration by rotating the vector either clockwise or counter clockwise by checking if it is in the first quadrant.
- Keep track the sign of the accumulated angle of every step.
- Finally, the real part is the absolute while the accumulated angle (with sign) is the phase of the vector.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{flowchart.png}
\caption{The flowchart of CORDIC}
\end{figure}

Example 3.1: CORDIC

- \( N=16 \)
- Compute cosine and sine of 16 angles, \( \frac{45}{2^0}, \frac{45}{2^1}, \frac{45}{2^2}, \frac{45}{2^3}, \ldots, \frac{45}{2^{15}} \)
- Reset the accumulated angle and move the vector to the first quadrant.
- Start iteration.
- Let’s do an example. \( x = 3.2e^{\frac{75\pi}{180}} \)
Table 5: The absolute and phase result of a complex number using CORDIC

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Complex number</th>
<th>Imag sign</th>
<th>Quadrant</th>
<th>Rotation</th>
<th>Angle(degree)</th>
<th>Multiplied result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.8282+3.0909j</td>
<td>+</td>
<td>1</td>
<td>clockwise</td>
<td>45</td>
<td>2.7712+1.600j</td>
</tr>
<tr>
<td>2</td>
<td>2.7712+1.600j</td>
<td>+</td>
<td>1</td>
<td>clockwise</td>
<td>22.5</td>
<td>3.1726+0.4716j</td>
</tr>
<tr>
<td>3</td>
<td>3.1726+0.4176j</td>
<td>+</td>
<td>1</td>
<td>clockwise</td>
<td>11.25</td>
<td>3.1931-0.2092j</td>
</tr>
<tr>
<td>4</td>
<td>3.1931-0.2092j</td>
<td>-</td>
<td>4</td>
<td>counter clockwise</td>
<td>-5.625</td>
<td>3.1982+0.1047j</td>
</tr>
<tr>
<td>5</td>
<td>3.1982+0.1047j</td>
<td>+</td>
<td>1</td>
<td>clockwise</td>
<td>2.8125</td>
<td>3.1995-0.0523j</td>
</tr>
<tr>
<td>6</td>
<td>3.1995-0.0523j</td>
<td>-</td>
<td>4</td>
<td>counter clockwise</td>
<td>-1.40625</td>
<td>3.1998+0.0261j</td>
</tr>
<tr>
<td>7</td>
<td>3.1998+0.0261j</td>
<td>+</td>
<td>1</td>
<td>clockwise</td>
<td>0.703125</td>
<td>3.1999-0.0130j</td>
</tr>
<tr>
<td>8</td>
<td>3.1999-0.0130j</td>
<td>-</td>
<td>4</td>
<td>counter clockwise</td>
<td>-0.35156</td>
<td>3.1999+0.0065j</td>
</tr>
<tr>
<td>9</td>
<td>3.1999+0.0065j</td>
<td>+</td>
<td>1</td>
<td>clockwise</td>
<td>0.17578</td>
<td>3.1999-0.0032j</td>
</tr>
<tr>
<td>10</td>
<td>3.1999-0.0032j</td>
<td>-</td>
<td>4</td>
<td>counter clockwise</td>
<td>-0.08789</td>
<td>3.1999+0.0016j</td>
</tr>
<tr>
<td>11</td>
<td>3.1999+0.0016j</td>
<td>+</td>
<td>1</td>
<td>clockwise</td>
<td>0.04394</td>
<td>3.1999-0.0008j</td>
</tr>
<tr>
<td>12</td>
<td>3.1999-0.0008j</td>
<td>-</td>
<td>4</td>
<td>counter clockwise</td>
<td>-0.02197</td>
<td>3.1999+0.0004j</td>
</tr>
<tr>
<td>13</td>
<td>3.1999+0.0004j</td>
<td>+</td>
<td>1</td>
<td>clockwise</td>
<td>0.01098</td>
<td>3.1999-0.0002j</td>
</tr>
<tr>
<td>14</td>
<td>3.1999-0.0002j</td>
<td>-</td>
<td>4</td>
<td>counter clockwise</td>
<td>-0.00549</td>
<td>3.1999+0.0001j</td>
</tr>
<tr>
<td>15</td>
<td>3.1999+0.0001j</td>
<td>+</td>
<td>1</td>
<td>clockwise</td>
<td>0.00274</td>
<td>3.1999-0.0005j</td>
</tr>
<tr>
<td>16</td>
<td>3.1999-0.0005j</td>
<td>-</td>
<td>4</td>
<td>counter clockwise</td>
<td>-0.00137</td>
<td>3.1999+0.0000j</td>
</tr>
</tbody>
</table>

| absolute  | 3.1999         |
| phase     | 74.9995        |

\[
\text{Smag} = 3.2;
\text{Sphs} = 75; \quad \text{%% degree}
\text{s} = \text{Smag}*(\exp(j*\text{Sphs}*\text{pi}/180)); \quad \text{%% change to radian}
\text{Nitersions} = 16;
\text{Preangle} = 45*\text{ones}(1,\text{Nitersions})./(2.^(0:\text{Nitersions}-1));
\text{Preangle} = \text{Preangle}\cdot\text{pi}/180;
\text{Presin} = \sin(\text{Preangle});
\text{Precos} = \cos(\text{Preangle});
\text{SignAcc} = \{\};
\text{x} = \text{real}(\text{s});
\text{y} = \text{imag}(\text{s});

\text{for iter} = 1:\text{Nitersions}
  \text{if sign(y)} == 1
    \text{temp_real} = x\cdot\text{Precos(iter)}+y\cdot\text{Presin(iter)};
    \text{temp_imag} = -x\cdot\text{Presin(iter)}+y\cdot\text{Precos(iter)};
  \text{elseif sign(y)} == -1
    \text{temp_real} = x\cdot\text{Precos(iter)}-y\cdot\text{Presin(iter)};
    \text{temp_imag} = x\cdot\text{Presin(iter)}+y\cdot\text{Precos(iter)};
  \text{end
SignAcc(iter) = sign(y);
x = temp_real;
y = temp_imag;
end

magnitude = x
phase = sum(Preangle.*SignAcc)*180/pi

Figure 12: The Matlab code of CORDIC algorithm

3.2 FIR, IIR, Fourier Transform

The Fourier transform is used to map the time domain into the frequency domain. Given the signal of length $N$, the forward and inverse transform is given by the following equations.

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j\frac{2\pi kn}{N}} ; k = 0, 1, \ldots N-1$$

$$= DFT(x, N)$$

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] e^{j\frac{2\pi kn}{N}} ; n = 0, 1, \ldots N-1$$

$$= IDFT(X, N)$$

Now, we want to relate the DFT to other DSP algorithm such as IDFT, FIR, and IIR.

**IDFT**

We want to implement IDFT using DFT function.

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] e^{j\frac{2\pi kn}{N}}$$

$$= \frac{1}{N} \left( \sum_{k=0}^{N-1} X[k] e^{-j\frac{2\pi k}{N}} \right)^*$$

$$= \frac{1}{N} \left( DFT(X^*, N) \right)^*$$

Figure 13: A block diagram of IDFT using DFT
N = 8;
x = [1, 2, 3, 4, 5, 6, 7, 8];
Xf = fft(x, N); % forward
xx = ifft(Xf, N); % reverse

%% Experiment ifft by using fft
Yf = conj(Xf);
yy = fft(Yf, N);
yy = conj(yy)/N;
error1 = max(abs(xx-yy))
%% error1 = 0

Figure 14: The Matlab code of Inverse DFT

FIR
In some applications, we want to find only a few frequency contents such as 1 KHz and/or 2 KHz. Using the DFT may not be efficient since it provides several frequencies which we don’t care about. Since the Fourier transform can be viewed as a filterbank or match filter, we can implement it with FIR filtering. The kernel of the Fourier transform is a complex exponential which can be decomposed into cosine and sine.

Convolution
\[
\sum_{n=0}^{N-1} x[n] \cos\left(\frac{2\pi nk}{N}\right) = \left( x[n] \otimes \cos\left(\frac{2\pi (N - n - 1)k}{N}\right) \right)_{n=N-1}
\]

DFT
\[
X[k] = \sum_{n=0}^{N-1} x[n] e^{-j\frac{2\pi nk}{N}}
= \sum_{n=0}^{N-1} x[n] \cos\left(\frac{2\pi nk}{N}\right) - j \sum_{n=0}^{N-1} x[n] \sin\left(\frac{2\pi nk}{N}\right)
= \left( \text{filter}(\cos\left(\frac{2\pi (N - n - 1)k}{N}\right), 1, x[n]) \right)_{n=N-1} - j \left( \text{filter}(\sin\left(\frac{2\pi (N - n - 1)k}{N}\right), 1, x[n]) \right)_{n=N-1}
\]

Implementation
- Choose the frequency \( k_1 \)
- Choose the size of the buffer \( N \)
- Generate the cosine and sine filter coefficients and reverse the order
- Perform FIR filtering and the outputs from cosine and sine filters are the real and imaginary parts, respectively.
Figure 15: A block diagram of DFT using FIR

%% \( N = 1024 \)
%% \( \text{frequency} = 125 \) Hz
%% Bin number \( k = 16 \)
n = 0:1023;
Fs = 8000;
K = 16;
x = sin(2*pi*n*125/Fs);  \( \% \) 125 Hz
X = exp(-2*j*pi*n'*n/length(n))*x';
X_FFT = X(K+1);

%%%% FFT using FIR %%%%%
N = length(x);

\[
\cosFIR = \text{real}(\exp(2*j*pi*K*(N-n-1)/N));
\]
\[
\sinFIR = \text{imag}(\exp(2*j*pi*K*(N-n-1)/N));
\]

\[
X_{\text{FIR\_real}} = \text{filter}(\cosFIR,1,x);
\]
\[
X_{\text{FIR\_imag}} = \text{filter}(\sinFIR,1,x);
\]
\[
X_{\text{FIR}} = X_{\text{FIR\_real}}(\text{end})-j*X_{\text{FIR\_imag}}(\text{end});
\]
\[
\text{error2} = \text{max}(|X_{\text{FFT}}-X_{\text{FIR}}|);
\]


Alternatively, we can perform the Fourier transform using IIR filtering.

\[
X[k] = \sum_{m=0}^{N-1} x[m] e^{-j\frac{2\pi mk}{N}}
\]
\[
= \sum_{m=0}^{N-1} x[m] e^{-j\frac{2\pi mk}{N}} e^{-j2\pi Nk/N}
\]
\[
= \sum_{m=0}^{N-1} x[m] W_N^m W_N^{-Nk}
\]
\[
= \sum_{m=0}^{N-1} x[m] W_N^{-(N-m)k}
\]
\[
= \left( x[m] \otimes h[m] \right)_{m=N}
\]

where
Finally we can write

\[
X[k_1] = \left( \text{filter}(1, \{1, -W_N^{-1}\}, x[n]) \right)_{n=N}
\]

\[
= \left( \text{filter}(1, \{1, -2\cos \frac{2\pi k_1}{N}\}, 1, x[n]) \right)_{n=N}
\]

\[
= \left( \text{filter}(1, \{-\cos \frac{2\pi k_1}{N}\}, 1, -2\cos \frac{2\pi k_1}{N}, x[n]) \right) + j \left( \text{filter}(0, \sin \frac{2\pi k_1}{N}, 1, -2\cos \frac{2\pi k_1}{N}, x[n]) \right)
\]

%%% N = 1024
%%% frequency = 125 Hz
%%% Bin number k = 16
%%% Fs = 8000;
K = 16;
x = sin(2*pi*n*125/Fs);      %% 125 Hz
X = exp(-2*j*pi*n'*n/length(n))*x';
X_FFT = X(K+1);
%%% FFT using IIR %%%
y_real = filter([1 -cos(2*pi*K/N)], [1 -2*cos(2*pi*K/N) 1], sin(2*pi*[0:1024]*125/Fs));
y_imag = filter([0 sin(2*pi*K/N)], [1 -2*cos(2*pi*K/N) 1], sin(2*pi*[0:1024]*125/Fs));
y = y_real+j*y_imag;
X_IIR = y(end);
error3 = max(abs(X_FFT-X_IIR));
%%% error3 = 1.3655e-010;
3.3 DFT with half length

If the data is length N, we usually do the FFT with N points input to get N points output. There is a special case, when the input is real, we can perform the FFT with only half length of the data, N/2.

Review some backgrounds

\[ x[n] \leftrightarrow \hat{X}[k] \]
\[ x'[n] \leftrightarrow \hat{X}'[-k] \mod N \]

Look at the difference between \( \hat{X}[k] \) and \( \hat{X}[-k] \) in Figure 17. Just leave \( \hat{X}[0] \) and reverse the order from \( \hat{X}[1] \) to \( \hat{X}[N-1] \) and call \( \hat{X}[-k] \)

![Diagram showing \( \hat{X}[k] \) and \( \hat{X}[-k] \)]

\[ \hat{X}[k] \]
\[ \hat{X}[-k] \]

Figure 17: The illustration of \( \hat{X}[k] \) and \( \hat{X}[-k] \)

If \( x[n] \) is a complex number

\[
\begin{align*}
\text{Re}\{x[n]\} & = \frac{1}{2}\{x[n] + x'[n]\} \\
\text{Im}\{x[n]\} & = \frac{1}{2j}\{x[n] - x'[n]\}
\end{align*}
\]

Therefore after we take DFT

\[
\begin{align*}
\text{DFT} (\text{Re}\{x[n]\}) & = \frac{1}{2}\{\hat{X}[k] + \hat{X}'[-k]\} \\
\text{DFT} (\text{Im}\{x[n]\}) & = \frac{1}{2j}\{\hat{X}[k] - \hat{X}'[-k]\}.
\end{align*}
\]

How to do DFT for only half length?

Assume \( y[n] \) is real for \( n = 0...2N-1 \).
$$DFT\left(y[n]\right) = \sum_{n=0}^{2N-1} y[n]W_{2N}^{nk}$$

$$= \sum_{n=even} y[n]W_{2N}^{nk} + \sum_{n=odd} y[n]W_{2N}^{nk}$$

$$= \sum_{m=0}^{N-1} y[2m]W_{2N}^{2mk} + \sum_{m=0}^{N-1} y[2m+1]W_{2N}^{(2m+1)k}$$

Let

$$y[2m] = y\ even \Rightarrow y_{e}[m]$$

$$y[2m+1] = y\ odd \Rightarrow y_{o}[m]$$

So, we can write

$$DFT\left(y[n]\right) = \sum_{m=0}^{N-1} y[2m]W_{2N}^{2mk} + \sum_{m=0}^{N-1} y[2m+1]W_{2N}^{(2m+1)k}$$

$$= \sum_{m=0}^{N-1} y_{e}[m]W_{N}^{mk} + \sum_{m=0}^{N-1} y_{o}[m]W_{2N}^{(2m+1)k}$$

$$= DFT\left(y_{e}[m], N\right) + W_{2N}^{k}DFT\left(y_{o}[m], N\right)$$

If \(x[n] = y_{e}[n] + jy_{o}[n]\) when \(n = 0\ldots N - 1\), then

$$\text{DFT}\left(y_{e}[n]\right) = \frac{1}{2}\{\tilde{X}[k] + \tilde{X}^*[-k]\}$$

$$\text{DFT}\left(y_{o}[n]\right) = \frac{1}{2j}\{\tilde{X}[k] - \tilde{X}^*[-k]\}.$$

**Implementation**

1. Goal is to find DFT of \(y[n]\) with length 2N, \(n = 0\ldots 2N - 1\).
2. We form \(x[n]\), complex numbers length N, by \(x[n] = y_{e}[n] + jy_{o}[n]\) such as
   
   \[\begin{array}{c}
   x[0] = y[0] + jy[1] \\
   \vdots \\
   \end{array}\]

3. Find the DFT of \(x[n]\) with length N to get \(\tilde{X}[k]\)
4. Find \(\tilde{X}^*[-k]\) from \(\tilde{X}[k]\).
5. Form

   $$\text{DFT}\left(y_{e}[n]\right) = \frac{1}{2}\{\tilde{X}[k] + \tilde{X}^*[-k]\}$$

   $$\text{DFT}\left(y_{o}[n]\right) = \frac{1}{2j}\{\tilde{X}[k] - \tilde{X}^*[-k]\}$$

6. Form DFT of \(y[n]\) length N by
\[ DFT(y[n]) = DFT(y_s[m], N) + W_{2N}^k DFT(y_o[m], N) \]

And we just need to form the second half using complex conjugate properties.

7. Finally, \( \hat{Y}[N] = \hat{Y}_s[0] - \hat{Y}_o[0] \). See matlab code.

```matlab
y = randperm(16); % use 16 points
Y_FFT = fft(y,16);

%% use 8 points
x = y(1:2:16)+j*(y(2:2:16));
X = fft(x,8);
Xconjrev = conj([X(1) fliplr(X(2:length(X)))]);

k = 0:(length(y)/2)-1;
W = exp(-j*2*pi*k/length(y));
Ye = (1/2)*(X+Xconjrev);
Yodd = (1/(2*j)) * (X-Xconjrev) .* W;
Y_halfFFT = Yeven+Yodd;
Y_halfFFT = [Y_halfFFT, Yeven(1)-Yodd(1), conj(fliplr(Y_halfFFT(2:length(Y_halfFFT))));

E = Y_FFT-Y_halfFFT;
%% max(abs(E)) = 5.4026e-015
```

**Figure 18: The matlab code for half length DFT**

### 3.4 Autocorrelation

Autocorrelation is used to calculate the correlation of the signal with itself. It is useful to detect the periodicity of the random signal.

\[ R_x[\tau] = \sum_{n=0}^{N-1} x[n]x[n+\tau] \quad ; 0 \leq |\tau| \leq N-1 \]

\[ = x[n] \otimes x[-n] \]

**Notes:**
- The autocorrelation function is symmetric function, \( R_x[\tau] = R_x[-\tau] \), therefore we only compute the positive time lag, \( \tau \).
- The maximum value occurs at \( \tau = 0 \) which represents the energy of the signal, \( \sum_n x^2[n] \).
- The Fourier transform of autocorrelation is the magnitude square of Fourier transform of the signal.

\[ \Im \{ R_x[\tau] \} = \Im \{ X[n] \} \times \Im \{ x[-n] \} \]

\[ = X(e^{j\omega})X^*(e^{j\omega}) \]

\[ = |X(e^{j\omega})|^2 \]

- That’s why if the signal has a dominant frequency (a peak in power spectrum), we can see the periodicity in its autocorrelation.
%% Simulate input signal
%% period sample = 2*64 = 128
n = 0:2047;
x = sin(n*pi/64);
x = (x/(max(x)-min(x)))+0.2*randn(size(x));
%% Demean input to make bipolar value
%% Normalize Lag zero equal one
%% The second peak location is period
maxlag = 512;
[Rx,Lag] = xcorr(x,maxlag);
Lagzero = find(Lag==0);
figure(1);
subplot(211);plot(n,x);axis([0 2047 -1.1 1.1]);
title('input');ylabel('Mag');xlabel('index');
subplot(212);plot(Lag(Lagzero:length(Lag)),Rx(Lagzero:length(Lag))/Rx(Lagzero));
axis([0 512 -1 1]);title('Autocorrelation');ylabel('Mag');xlabel('Lag');

Figure 19: Autocorrelation of the random signal

Figure 20: The Matlab code for autocorrelation

Implementation
- If we care only a few specific frequencies, or lag, the fast implementation can be useful.
- Instead of computing the multiply/summation of a whole buffer, we can add and subtract the new samples and old samples respectively. See the following figures for lag 0 and lag 2.
3.5 Discrete Cosine Transform (DCT)

In this section, we will discuss other frequency transforms such as Discrete Cosine Transform and Modulated Lapped Transform. Since the kernel in the DFT is a complex exponential function, the output is a complex number. In some application such as image processing, a real number is preferred. To avoid complex numbers, the DCT is used instead of the DFT. Let’s see some facts of the Fourier transform.

The simple idea of the DCT is duplicating of the original signal and then taking the DFT. Let’s see the math.

![Figure 22: The extension of a time signal for the DCT](http://www.delphion.com/cgi-bin/viewpat.cmd/US05619004)
\[ X[k] = \sum_{n=0}^{2N-1} x[n]W_{2N}^{-nk} \]

\[ = \sum_{n=0}^{N-1} x[n]W_{2N}^{-nk} + \sum_{n=N}^{2N-1} x[n]W_{2N}^{-nk} \quad ; n' = 2N - 1 - n \]

\[ = \sum_{n=0}^{N-1} x[n]W_{2N}^{-nk} + \sum_{n=0}^{N-1} x[n]2N - 1 - nW_{2N}^{-2(N-1-n)k} \]

\[ = \sum_{n=0}^{N-1} x[n]W_{2N}^{-nk} + \sum_{n=0}^{N-1} x[n]W_{2N}^{-(1-n)k} \quad ; W_{2N}^{-2Nk} = 1 \]

\[ = W_{2N}^{\frac{k}{2}} \left[ \sum_{n=0}^{N-1} x[n] \left( 2W_{2N}^{-nk} + \frac{W_{2N}^{nk}}{2} \right) \right] \]

\[ = W_{2N}^{\frac{k}{2}} \left[ 2 \sum_{n=0}^{N-1} x[n] \cos \left( \frac{2\pi(n + \frac{1}{2})k}{2N} \right) \right] \]

**DCT II**

\[ X[m] = \sqrt{\frac{2}{N}} K_m \sum_{n=0}^{N-1} x[n] \cos \left( \frac{m\pi(2n + 1)}{2N} \right) \quad ; m = 0, 1, \ldots, N - 1 \]

\[ K_m = \begin{cases} 1 & ; m = 0 \\ \frac{1}{\sqrt{2}} & ; m \neq 0 \end{cases} \]

How can we construct the DCT from the DFT? Look at the summation term and relate it to the DFT.

\[ \cos \left( \frac{m\pi(2n + 1)}{2N} \right) = \Re \left\{ W_{2N}^{\frac{mk}{2}} \right\} \]

\[ \sum_{n=0}^{N-1} x[n] \cos \left( \frac{m\pi(2n + 1)}{2N} \right) = \Re \left\{ \sum_{n=0}^{N-1} x[n]W_{2N}^{\frac{mk}{2}} \right\} \]

\[ = \Re \left\{ W_{\frac{2N}{2}}^{k} \text{DFT}(x[n], 2N) \right\} \]

**Figure 23: The block diagram of DCT using DFT**
Comments on DCT:

- DCT coefficients are purely real but the DFT is complex.
- The energy compaction of the DCT is equivalent to the optimal transform. Since the energy compaction of the DCT is better than the DFT, it is used in coding applications.
- The magnitude of the DCT is shift variant so it is difficult to interpret the same signal with a different starting time.
- The DCT is based on the DFT of 2N length of the signal, and can be looked at as N frequency sampled points from 0 to $\pi$ whereas the DFT can be looked at as N frequency sampled points from 0 to $2\pi$.
- For long data, we need to process in several blocks and concatenate the results together. However, the discontinuity between each block can cause artifacts resulting in a clicking sound. The overlap-add method is commonly used to smooth out this effect. However, overlap-add of the DFT and DCT can cause oversampling (more samples than original time signal samples).
- To avoid oversampling and block artifacts, the Modulated Lapped Transform (MLT) can be used.
Figure 25: The frequency resolution of DCT and DFT

\[ N = 64; \]
\[ K = N; \]
\[ k = \{0:K-1\}'; \]
\[ n = \{0:N-1\}; \]
\[ x = \{0:N-1\}'; \]

%% Implement using the matrix
\[
DCT_{\text{MATRIX}} = \cos(k*(2*n+1)*\pi/(2*N));
\]
\[
X_{\text{DCT}} = DCT_{\text{MATRIX}}*x;
\]
\[
Km = \sqrt{2/N}*\text{ones}(\text{size}(X_{\text{DCT}}));
\]
\[
Km(1) = \sqrt{1/N};
\]
\[
X_{\text{DCT}} = X_{\text{DCT}}.*Km;
\]

%% Implement using DFT
\[
x_{\text{zeropad}} = [x;\text{zeros(\text{size}(x))}];
\]
\[
X_{\text{temp}} = \text{fft}(x_{\text{zeropad}},2*N);
\]
\[
X_{\text{temp}} = X_{\text{temp}}(k+1);
\]
\[
X_{\text{DCTFFT}} = \text{real}(X_{\text{temp}}.*\exp(-j*k*\pi/(2*N))).*Km;
\]

\[
\text{Error} = \max(\text{abs}(X_{\text{DCTFFT}}-X_{\text{DCT}}));
\]
\[
\% \text{Error} = 4.3586\times10^{-13}
\]

%% Reconstruction back, same as forward
%% Implement using the matrix
\[
IDCT_{\text{MATRIX}} = DCT_{\text{MATRIX}}';
\]
\[
X_{\text{DCTscale}} = X_{\text{DCT}}.*Km;
\]
\[
xr_{\text{MATRIX}} = IDCT_{\text{MATRIX}}*X_{\text{DCTscale}};
\]

%% Implement using DFT
\[
X_{\text{DCTFFTscale}} = X_{\text{DCTFFT}}.*Km.*\exp(-j*\pi*k/(2*N));
\]
\[ X_{\text{zeropad}} = [X_{\text{DCTFFTscale}}; \text{zeros(size}(X_{\text{DCTFFTscale}}))]; \]
\[ X_{\text{temp}} = \text{fft}(X_{\text{zeropad}}, 2*N); \]
\[ X_{\text{temp}} = X_{\text{temp}}(k+1); \]
\[ x_{r_{\text{DCTFFT}}} = \text{real}(X_{\text{temp}}); \]
\[ \text{Error2} = \max(\text{abs}(x_{r_{\text{MATRIX}}}-x_{r_{\text{DCTFFT}}})); \]
\[ \%\text{Error2} = 4.5830e-013 \]

Figure 26: Matlab code for computing DCT from DFT

3.6 Modulated Lapped Transform (MLT)

The ideals of the MLT are the following:
- Each block has \( N \) data samples. Consecutive blocks are overlapped by 50% or \( N/2 \) samples.
- For analysis, the MLT is mapped from \( N \) samples to \( N/2 \) samples.
- For synthesis, the MLT is mapped from \( N/2 \) coefficients to \( N \) samples and added with the consecutive blocks.

**Analysis**

\[
X[k] = \sqrt{\frac{2}{N}} \sum_{n=0}^{N-1} x[n] h_a[n] \cos \left[ \frac{2\pi}{N} \left( n + \frac{N}{4} + \frac{1}{2} \right) \left( k + \frac{1}{2} \right) \right] : 0 \leq k \leq \frac{N}{2} - 1
\]

where

\( h_a[n] = \text{analysis windows length } N \)

**Synthesis**

\[
x[n] = \sqrt{\frac{2}{N}} \sum_{k=0}^{N-1} X[k] h_s[k] \cos \left[ \frac{2\pi}{N} \left( n + \frac{N}{4} + \frac{1}{2} \right) \left( k + \frac{1}{2} \right) \right] : 0 \leq n \leq N - 1
\]

where

\( h_s[n] = \text{synthesis windows length } N \)

Commonly sine window is used for analysis and synthesis window,

\[
h_s[n] = h_a[n] = \sin \left( \frac{\pi (n + \frac{1}{2})}{N} \right)
\]
We can implement the MLT using the DFT as following.

**Analysis**

\[
X[k] = \sqrt{\frac{2}{N}} \sum_{n=0}^{N-1} x[n] h_a[n] \cos \left( \frac{2\pi}{N} \left( n + \frac{N}{4} + \frac{1}{2} \right) \left( k + \frac{1}{2} \right) \right) \quad ; k = 0, 1, \ldots \frac{N}{2} - 1
\]

\[
= \sqrt{\frac{2}{N}} \text{Re} \left\{ \sum_{n=0}^{N-1} x[n] h_a[n] e^{\frac{j2\pi}{N} \left( n + \frac{N}{4} \right) \left( k + \frac{1}{2} \right)} \right\} \quad ; N' = \frac{N}{4} + \frac{1}{2}
\]

\[
= \sqrt{\frac{2}{N}} \text{Re} \left\{ \sum_{n=0}^{N-1} x[n] h_a[n] e^{\frac{j2\pi}{N} \left( nk + \frac{N}{2} + \frac{N'}{2} \right)} \right\}
\]

\[
= \sqrt{2 \frac{N}{N}} \text{Re} \left\{ e^{-\frac{j2\pi}{N} \left( N' \frac{N}{2} \right)} \sum_{n=0}^{N-1} x[n] \left\{ \overline{h_a[n]} e^{\frac{j\pi}{N} \left( n + \frac{N}{2} \right) k} \right\} e^{\frac{j2\pi nk}{N}} \right\}
\]

\[
= \sqrt{2 \frac{N}{N}} \text{Re} \left\{ e^{\frac{j2\pi}{N} \left( N' \frac{N}{2} \right)} DFT \left( x[n] h_a[n] e^{-\frac{j\pi n}{N}}, N \right) \right\} \quad \text{take only } k = 0, 1, \ldots \frac{N}{2} - 1
\]
Synthesis

\[ x[n] = \sqrt{\frac{2}{N}} \sum_{k=0}^{N-1} X[k] h[n] \cos \left[ \frac{2\pi}{N} \left( n + \frac{N}{4} + \frac{1}{2} \right) \left( k + \frac{1}{2} \right) \right] \]

\[ = \sqrt{\frac{2}{N}} \text{Re} \left\{ \sum_{k=0}^{N-1} \tilde{X}[k] h[n] e^{-\frac{j2\pi}{N} \left( n + \frac{N}{4} + \frac{1}{2} \right) \left( k + \frac{1}{2} \right)} \right\} : \tilde{X} = [X, 0, 0, \ldots] \]

\[ = \sqrt{\frac{2}{N}} \text{Re} \left\{ e^{-\frac{j2\pi}{N} \left( \frac{n}{2} - \frac{1}{2} \right)} h[n] \sum_{k=0}^{N-1} X[k] e^{-\frac{j2\pi N k}{N}} e^{-\frac{j2\pi n k}{N}} \right\} \]

\[ = \sqrt{\frac{2}{N}} \text{Re} \left\{ e^{-\frac{j2\pi}{N} \left( \frac{n}{2} - \frac{1}{2} \right)} h[n] \text{DFT} (X[k] e^{-\frac{j2\pi N k}{N}}, N) \right\} \quad \text{take } n = 0, 1, \ldots N - 1 \]

Analysis

\[ x[n] \]
\[ n = 0, 1, \ldots N - 1 \]

\[ h[n] e^{\frac{j\pi n}{N}} \]

\[ \tilde{X}[k] \quad e^{-\frac{j2\pi N k}{N}} \]

\[ X[k] \quad k = 0, 1, \ldots \frac{N}{2} - 1 \]

Synthesis

\[ X[k] \]
\[ k = 0, 1, \ldots \frac{N}{2} - 1 \]

\[ h[n] \quad e^{\frac{j\pi n}{N}} \]

\[ \text{DFT} \]

\[ x[n] \quad n = 0, 1, \ldots N - 1 \]

\[ e^{-\frac{j2\pi N k}{N}} \]

\[ Z_n \]

\[ \text{Real and only } N/2 \text{ points} \]

\[ \text{Zero padding} \]

\[ \text{DFT} \]

\[ \text{Real and } n=0,1,\ldots N-1 \]

\[ \text{DFT} \]

\[ \text{Real and } n=0,1,\ldots N-1 \]

\[ e^{-\frac{j2\pi N k}{N}} \]

\[ h[n] \quad e^{\frac{j\pi n}{N}} \]

Figure 28: The block diagrams of MLT implementation using DFT
\( N = 64; \)
\( K = N/2; \)
\( k = [0:(N/2)-1]'; \)
\( n = [0:N-1]; \)
\( x = \sin(2\pi*50*(0:N-1)'/1000); \)

%% sine window
\( w = \sin(\pi*(n'+0.5)/N); \)
% \( w_{\text{reverse}} = \text{fliplr}(w); \)
% \( w_{\text{halfshift}} = \text{fftshift}(w); \)
% see \( w = w_{\text{reverse}} \)
% see \( (w_{\text{halfshift}}.^2+w_{\text{reverse}}.^2) = 1; \)

%% Implement using the matrix
\[
\text{MLT\_MATRIX} = \cos(2\pi*(k+0.5)*(n+(N/4)+0.5)/N); \\
\text{MLT\_IMATRIX} = \sqrt{2/N} \cos(2\pi*(n+(N/4)+0.5)'*(k+0.5)'/N); \\
\text{X\_MLT} = \sqrt{2/N} \text{MLT\_MATRIX}*(x.*w);
\]

%% Implement using FFT
\( N_{\text{prime}} = (N/4)+(1/2); \)
\( \text{temp1} = x.*w.*(\exp(-j*pi*n'/N)); \)
\( \text{temp2} = \text{fft(temp1,N)}; \)
\( \text{temp2} = \text{temp2}(k+1); \)
\( \text{temp3} = \text{temp2}.*\exp(-j*2*pi*Nprime*k/N); \)
\( \text{X\_MLTFFT} = \sqrt{2/N} \text{real}(\exp(-j*pi*Nprime/N)*\text{temp3}); \)

\( \text{Error} = \text{max}(\text{abs(X\_MLT-X\_MLTFFT)}); \)
\( \text{\% Error} = 6.1489e-015 \)

Figure 29: The Matlab code of MLT using DFT

3.7 Hilbert Transform

For a certain signal such as an amplitude modulated (AM) signal, the low frequency signal (information) is modulated by the high frequency signal (the carrier). At the decoder, we can retrieve the low frequency signal by modulating the same high frequency carrier followed by a low pass filter. However, for real time application, we want to decode sample by sample. The Hilbert transform is used to solve this problem. The Hilbert transform is a type III FIR filter (odd and anti-symmetric), and the result after the Hilbert transform is a complex number who’s magnitude is the low frequency signal and the phase derivative is the carrier frequency. The frequency characteristic of the Hilbert filter is

\[
H_{\text{hil}}(e^{j\omega}) = \begin{cases} 
-j e^{-j\omega} & ; 0 < \omega \leq \pi \\
-j e^{j\omega} & ; -\pi < \omega < 0 
\end{cases}
\]

Now, we want to find the impulse response of the Hilbert transform.
\[
\begin{align*}
  h_{hil}[n] &= \frac{1}{2\pi} \int_{-\pi}^{\pi} H_{hil}(e^{j\omega}) e^{j\omega n} d\omega \\
  &= \frac{1}{2\pi} \int_{-\pi}^{0} e^{-jn\omega} e^{j\omega n} d\omega - \frac{1}{2\pi} \int_{0}^{\pi} e^{-jn\omega} e^{j\omega n} d\omega \\
  &= \frac{1}{2\pi} \int_{-\pi}^{0} e^{j\omega(n-\alpha)} d\omega - \frac{1}{2\pi} \int_{0}^{\pi} e^{j\omega(n-\alpha)} d\omega \\
  &= \frac{1}{2\pi} \left( \frac{1-e^{-j\pi(n-\alpha)}}{j(n-\alpha)} \right) - \frac{1}{2\pi} \left( \frac{e^{j\pi(n-\alpha)} - 1}{j(n-\alpha)} \right) \\
  &= \frac{2 - 2\cos \pi(n-\alpha)}{2\pi(n-\alpha)} \quad ; n \neq \alpha \\
  &= \begin{cases} 
  \frac{2\sin^2 \frac{\pi(n-\alpha)}{2}}{\pi(n-\alpha)} & ; n \neq \alpha \\
  0 & ; n = \alpha 
  \end{cases}
\end{align*}
\]

**Implementation**

- Choose the size of buffer, N. Trade off between the delay and the mainlobe width.
- Compute the Hilbert filter from the above equation.
- Perform FIR filtering and combine with delayed original samples.
- Find the absolute and phase of the complex number
- It is really important to have the appropriate delay due to FIR filtering.
- Other tapping windows such as hanning or hamming windows can be used to reduce the sidelobe with mainlobe trade off.

---

```
%% Generate signal
n = 0:2047;
Fs = 8000;
Fm = 50;
Fc = 1000;
x = sin(2*pi*n*Fc/Fs).*(1+sin(2*pi*n*Fm/Fs));
x = x-mean(x);x = x/(max(x)-min(x));

B = fir1(64,0.1);
Abs_x = abs(x).^2;
Abs_x = filter(B,1,Abs_x);
```

---

**Figure 30:** The block diagram of Hilbert filter
%% Using hilbert transform
xh = hilbert(x);
Abs_xh = abs(xh);

figure(1);
subplot(221);plot(x);axis([0 2048 -0.5 0.5]);
subplot(222);plot(Abs_x);axis([0 2048 0 0.15]);
subplot(223);plot(Abs_xh);axis([0 2048 0 0.5]);

%% Design hilbert
n = 0:128;
a = 128/2;
h_hilbert = (2/pi)*(sin(pi*(n-a)/2).^2)./(n-a);
h_hilbert(a+1) = 0;
h_hilbert = h_hilbert.*hanning(length(h_hilbert));
x_imag = filter(h_hilbert,1,x);
x_real = [zeros(1,a),x];
x_real = x_real(1:length(x));
xh2 = x_real+j*x_imag;

subplot(224);plot(abs(xh2));axis([0 2048 0 0.5]);
xh_phase = unwrap(angle(xh2));

freq_est = diff(xh_phase);
smooth_filter = hanning(32);
smooth_filter = smooth_filter/sum(smooth_filter);
freq_smooth = filter(smooth_filter,1,freq_est);

figure(2);
subplot(211);plot(x);title('FSK signal');
subplot(212);plot(freq_smooth);

%% FSK signal
x_one = sin(2*pi*1000*(0:47)/8000);
x_zero = sin(2*pi*800*(0:39)/8000);
x = [x_one x_zero];
x = [x x x x x x x x x x];
x_imag = filter(h_hilbert,1,x);
x_real = [zeros(1,a),x];
x_real = x_real(1:length(x));
xh3 = x_real+j*x_imag;
xh_phase = unwrap(angle(xh3));

freq_est = diff(xh_phase);
smooth_filter = hanning(32);
smooth_filter = smooth_filter/sum(smooth_filter);
freq_smooth = filter(smooth_filter,1,freq_est);
3.8 32 bit multiplication using 16 bit DSP

Although the ADSP-2100 is a 16 bit DSP, we still can perform 32 bit filtering by saving the data into 2 memory buffers. The algorithm of 32 bit multiplication is quite complicated. The author has been using an algorithm written by his boss, Mr. Stephen G. Dame, during his internship. The basic idea is the following.

Implementation

- First, multiply X_low and Y_low and then shift the result by 16 bits corresponding to divide by $2^{16}$.
- Second, multiply X_high and Y_low and save into temp1, multiply X_low and Y_high and save into temp2.
- Sum temp1, temp2, and the shifted result from step1. The final result is again shifted by 16 bits corresponding to divide by $2^{16}$.
- Third, multiply X_high and Y_high and then sum with the shifted result from step 2.
- The final result is the first 32 bits.
**Example 3.2**

Assume a 4 bit DSP, we want to multiply 1101 1100 with 1001 1110

Exact result = 1000 0111 1100 1000

**Step1:**  \(X_{\text{low}} \text{ and } Y_{\text{low}} = 1100 \times 1110 = 1010 1000 \) after shift 0000 10 10

**Step2:**  \(X_{\text{high}} \text{ and } Y_{\text{low}} = 1101 \times 1110 = 1011 0110\)

\(X_{\text{low}} \text{ and } Y_{\text{high}} = 1100 \times 1001 = 0110 1100\)

After sum 3 values the result is  = 0001 0010 1100 after shift 0001 0010

**Step3:**  \(X_{\text{high}} \text{ and } Y_{\text{high}} = 1101 \times 1001 = 0111 0101\)

Finally is 0111 0101 + 0001 0010  = 1000 0111

---

**SUMMARY**

These notes discussed the fixed point DSP. The number representation of 16 bits and 32 bits was discussed with examples of Matlab and Excel code. The architecture of the ADSP-2100 was discussed and compared to the C67 and C31. The C-callable of the ADSP-2100 was introduced with the sample code of the CORDIC algorithm. Finally several DSP algorithms were provided. These algorithms are useful for real-time DSP implementation, both floating point and fixed point.

**Acknowledgement**

These notes were modified from several class notes of the author when he was a teaching assistant. The author would like to thank Stephen G. Dame for supporting software and discussion of the ADSP. Some contents of these notes are what the author had learned during a summer internship in 1999. Although there are a lot of things missing, there is still one thing the author has still remembered from him. *For a circular buffer, before doing convolution the pointer will point to the location of the upcoming sample (or the oldest delayed sample), not the current sample.* The author also would like to thank Chad R. Heinemann for improving the manuscript.