Introduction to DSP processors

Presented by:

Contents:

- The modern processor’s classification;
- The digital signal processing methods & algorithms;
- The D(i)gital S(ignal) P(rocessing) algorithms implementation
- The SHARC processor - architecture; data types & formats, C & Assembler;
- Getting started.
The modern processor’s classification.

Today, chips are distributed into three groups:

- **ASIC’s (Application Specific Integrated Circuits)**
- **Chips with hardware realization of data processing algorithms (microprocessors & microcontrollers)**
- **FPGA & EPLD**

The modern processor’s classification.

**Microprocessors & microcontrollers**

- **General-purpose microprocessors.**
  This kind of processor is intended for computer systems: PC, workstation & parallel supercomputer.
- **DSP microprocessors.**
  The processors are intended for Real-Time Digital Signal Processing Systems.
- **Microcontrollers.**
  Very special processors are intended for embedded systems and in different household devices.
Review: Processor Classes

- **General Purpose - high performance**
  - Pentiums, Alpha's, SPARC
  - 64-128 bit word size
  - Used for general purpose software
  - Heavy weight OS - UNIX, NT
  - Multiply layers of cache memory
  - Workstations, PC’s

- **DSP processors**
  - SHARC, BlackFin, TMS320C55x, TMS320C67x, TMS320C64x
  - 16-32 bit word size
  - Single program
  - Lightweight, often real-time OS
  - Super Harvard Architecture Support, MAC, Circular buffer, Dual-Port RAM
  - Audio, Image and Video processing, Coding and Decoding, Cellular Base Station, Adaptive Filtering, Real Time operations

- **Embedded processors and processor cores**
  - ARM9, ARC, 486SX, Hitachi SH7000, NEC V800
  - 32 bit word size
  - Single program
  - Lightweight, often real-time OS
  - Code and Data memory cache, DSP support
  - Cellular phones, consumer electronics (e.g. CD players)

- **Microcontrollers**
  - PIC, AVR, HC11, ARM7, 8051,80251
  - Extremely cost sensitive
  - Small word size - 8 bit common
  - Highest volume processors by far
  - Automobiles, toasters, thermostats, ...

---

The digital signal processing methods & algorithms

The analog signal processing example:

\[
\frac{y(t)}{x(t)} = \frac{R_f}{R_i} \frac{1}{1 + jw \frac{R_f}{C_f}} 
\]
The digital signal processing methods & algorithms

The digital signal processing system:

\[ x(t) \xrightarrow{\text{Anti-aliasing filter}} x'(t) \xrightarrow{\text{A/D}} x'(n) \xrightarrow{\text{Digital filter or digital transform}} y'(n) \xrightarrow{\text{D/A}} y'(t) \xrightarrow{\text{Smoothing filter}} y(t) \]

\[ y(n) = \sum_{k=-N}^{N} C_k x(n-k) \]

Aliasing

✓ When sampling at a rate of \( f_s \) samples/s, if \( k \) is any positive or negative integer, it’s impossible to distinguish between the sampled values of a sinewave of \( f_0 \) Hz and a sinewave of \( (f_0 + kf_s) \) Hz.
**ADC and Sampling**

- An ADC performs the following:
  - Sampling
  - Quantization
  - Binary Coding

- Sampling rate must be at least twice as much as the highest frequency component of the analog input signal (Nyquist frequency)

---

**The DSP vs. ASP**

- 5th Order Analog Sallen Key Low-Pass Filter

- 5th Order Digital Filter of Direct Form II
The digital signal processing methods & algorithms

Analog signal processing versus Digital signal processing

Analog signal processing: Digital signal processing:
- Cheaper;
- More compact;
- Power dissipation.
- More accurate;
- More stable for different environments.

The basis concepts of DSP:

\[ T - \text{sample time}; \]
\[ \tau_a - \text{algorithm time}. \]

\[ \tau_{\text{del}} = T - \tau_a \]
\[ \tau_a = \tau_{\text{op}} \cdot N \]
\[ \tau_{\text{op}} = \frac{1}{f_{\text{CPU CLK}}} \]

than \( \tau_{\text{op}} \) — operation time;
\( f_{\text{CPU CLK}} \) — processor clk frequency;
\( N \) — number of operations in an algorithm.
The D(igital) S(ignal) P(rocessing) algorithms implementation

The major tasks in DSP:

- Filter design (Linear Filtering);
- Speech detection, image recognition (Spectral Analysis);
- Image & Speech compression (Timing-Frequency Analysis);
- Image & signal processing (Adaptation Filtering);
- Coding, median filters (Non-Linear processing);
- Interpolation & decimation (Multi Speed Processing).

The very usable DSP algorithms.

- FIR filter
- IIR filter
- FFT
- STD
- Convolution
- Correlation
The D(igital) S(ignal) P(rocessing) algorithms implementation

FIR – filter
(Finite Impulse Response)

\[ y(n) = \sum_{i=0}^{N-1} b_i x(n-i) \]

IIR – filter
(Infinite Impulse Response)

\[ y(n) = \sum_{i=0}^{N-1} b_i x(n-i) - \sum_{k=1}^{M-1} a_k y(n-k) \]
The **D**igital **S**ignal **P**rocessing algorithms implementation

**Discrete correlation**

\[ y(n) = \frac{1}{N} \sum_{m=0}^{M} f(m)g(n + m) \quad 0 \leq m \leq M \]

**Discrete autocorrelation**

\[ y(n) = \frac{1}{N} \sum_{m=0}^{M} f(m)f(n + m) \quad 0 \leq m \leq M \]

**Discrete convolution**

\[ y(n) = f(n) * g(n) = \sum_{m=-M}^{M} f(m)g(n - m) = \sum_{m=-M}^{M} f(n - m)g(m) \]

**Circular discrete convolution**

\[ y(n) = \sum_{m=0}^{M-1} \left( \sum_{k=-N}^{N} f(m + kM) \right) g(n - m) \]

**STD – standard deviation**

\[ S_N = \sqrt{\frac{1}{N} \sum_{i=0}^{N-1} (x_i - \bar{x})^2} \]
The D(igital) S(ignal) P(rocesing) algorithms implementation

Discrete Fourier Transform

\[ X(k) = \left( \frac{1}{N} \right) \cdot \sum_{n=0}^{N-1} x(n) \cdot e^{-\frac{2\pi i kn}{N}}, \quad k = 0, \ldots, N - 1 \]

\[ W_{N}^{kn} = e^{-\frac{2\pi i kn}{N}} \]

Direct computation of the DFT is basically inefficient because it does not exploit the symmetry and periodicity properties of the phase factor \( W_{N} \). In particular, these two properties are:

- Symmetry property: \( W_{N}^{k+N/2} = -W_{N}^{k} \)
- Periodicity property: \( W_{N}^{k+N} = W_{N}^{k} \)

where \( W_{N}^{kn} = e^{-\frac{2\pi i kn}{N}} \).
The DSP processor’s architecture

Requirement for DSP processors:

1. High speed input data, different interface devices;
2. Input data wide dynamic range;
3. ADD, MULT & SHIFT hardware implementation. Parallel processing;
4. Flexible processing (possibility to “jump” from one process to another);
5. Algorithm’s regularity (Operation “come back”);

DSP processors features:

1. Various interface highspeed ports and timers
2. Parallel access memory architecture;
3. Three mathematical units: ALU, barrel Shifter and Multiplier with fast MAC operation (MBR = MBR + Rx * Ry);
4. Cycles, branches & interrupt fast handling. Addressing special modes;
5. Circular buffer.
DSP programs are different from traditional software tasks in two important respects.

- First, the programs are usually much shorter, say, one-hundred lines versus ten-thousand lines.
- Second, the execution speed is often a critical part of the application.

If assembly is used at all, it is restricted to short subroutines that must run with the utmost speed.

The DSP processor's architecture

"Traditional" von Neumann architecture

Harvard architecture
The DSP processor’s architecture

**Super Harvard** architecture

This is SHARC DSP processor structure

The DSP processor’s architecture

Core Processor

Dual-Ported SRAM

External Port

I/O Processor
The DSP processor’s architecture

ADSP-21160 Features

- 100 MHz - 600 MFLOPS - SIMD Core
- 1024 point, complex FFT benchmark: 90 us
- 4 Mbits on chip SRAM
- 14 zero overhead DMA channels
- Sustained 700 Mbyte/sec over IOP bus
- Two 50 mbit/sec Synchronous Serial Ports
- Six 100 Mbyte/sec link ports
- 64 bit synchronous external port
- Cluster multiprocessing support

The methods for computer performance measurement

- Peak (technical) performance of microprocessor:
  Maximum theoretical microprocessor’s speed in ideal conditions. It’s defined by number of calculating operation which had done in some time.

- Real (sustained) performance of microprocessor:
  Real microprocessor’s speed in real conditions. The real performance is calculated by execution of some popular programs.
  (like FIR, IIR or FFT).
**The DSP processor’s architecture**

**Pipe-Line command execution:**
- Instruction fetching (a);
- Decoding (b);
- Execution (c).

![Pipeline diagram](image)

**SHARC programming model**

- Register files:
  - Ro-R15 (aliased as F0-F15 for floating point)
- Status registers.
- Loop registers.
- Data address generator registers.
**SHARC assembly language**

Algebraic notation terminated by semicolon:

```
R1=DM(M0,I0), R2=PM(M8,I8); // comment
label: R3=R1+R2;
```

data memory access  program memory access

**Simple ALU Instructions**

| Rn = Rx + Ry | Fn = Fx + Fy |
| Rx = Rx - Ry | Fn = Fx - Fy |
| Rn = Rx + Ry + CI (Carry In) | Fn = ABS(Fx + Fy) |
| Rn = Rx - Ry + CI - 1 | Fn = ABS(Fx - Fy) |
| Rn = (Rx + Ry)/2 | Fn = (Fx + Fy)/2 |
| COMP(Rx, Ry) | COMP(Fx, Fy) |
| Rn = Rx + CI - 1 | Fn = -Fx |
| Rn = Rx + 1 | Fn = ABS Fx |
| Rn = Rx - 1 | Fn = PASS Fx |
| Rn = -Rx | Fn = RND Fx |
| Rn = ABS Rx | Fn = SCALB Fx BY Ry |
| Rn = PASS Rx | Rn = MANT Fx |
| Rn = Rx AND Ry | Rn = LOGB Fx |
| Rn = Rx OR Ry | Rn = FIX Fx BY Ry |
| Rn = NOT Rx | Fn = FLOAT Rx BY Ry |
| Rn = MIN(Rx, Ry) | Rn = TRUNC Fx |
| Rn = MAX(Rx, Ry) | Fn = RECIPS Fx |
MAC instructions - mainly INTEGER
Multiply and Accumulate

Rn = Rx * Ry
MRB = Rx * Ry
Rn = MRB + Rx * Ry
MRB = MRB + Rx * Ry
Rn = MRB - Rx * Ry
MRB = MRB - Rx * Ry
Rn = SAT MRB
MRB = SAT MRB
Rn = RND MRB
MRB = RND MRB
Rn = MR
MRF = Rx * Ry
Rn = MRF + Rx * Ry
MRF = MRF + Rx * Ry
Rn = MRF - Rx * Ry
MRF = MRF - Rx * Ry
Rn = SAT MRF
MRF = SAT MRF
Rn = RND MRF
MRF = RND MRF
MR = Rn

Example Multi-Function Instruction

f11=f1*f7, f3=f9+f14, f9=f9-f14, dm(i2,m0)=f13,
f7=pm(i8,m8);

✓ In a SingleCycle the SHARC Performs:
  ✓ 1(2) Multiply
  ✓ 1 (2) Addition
  ✓ 1 (2) Subtraction
  ✓ 1 (2) Memory Read
  ✓ 1 (2) Memory Write
  ✓ 2 Address Pointer Updates

✓ Plus the I/O Processor Performs:
  ✓ Active Serial Port Channels (2 Transmit, 2 Receive)
  ✓ Active Link Ports (6)
  ✓ Memory DMA
  ✓ 2 DMA Pointer Updates
Parallelism

- Restrictions on the sources of the operands when operations are combined.
- The operands going to the multiplier must come from \( R_0 \) through \( R_7 \) (or in the case of floating-point operands, \( F_0 \) to \( F_7 \)), with one input coming from \( R_0-R_3/F_0-F_3 \) and the other from \( R_4-R_7/F_0-F_7 \).
- The ALU operands must come from \( R_8-R_{15}/F_8-F_{15} \), with one operand coming from \( R_8-R_{11}/F_8-F_{11} \) and the other from \( R_{12}-R_{15}/F_2-F_{15} \).
- Performs three operations:
  - \( R_6 = R_0 \times R_4 \), \( R_9 = R_8 + R_{12} \), \( R_{10} = R_8 - R_{12} \)

SHARC load/store

- Load/store architecture: no memory-direct operations.
- Two data address generators (DAGs):
  - data memory.
  - program memory.
- Must set up DAG registers to control loads/stores.

Provide indexed, modulo, bit-reverse indexing.

- Bit-reversal addressing can be performed only in \( R_{10} \) and \( R_{18} \), as controlled by the BR0 and BR8 bits in the MODE1 register.
**DAGs registers**

- I0
- I1
- I2
- I3
- I4
- I5
- I6
- I7

- M0
- M1
- M2
- M3
- M4
- M5
- M6
- M7

**SHARC assembly language**

- I register holds start address.
- M register/immediate holds modifier value.
  
  ```
  r0 = DM(I3,M3) // Load
  DM(I2,1) = r1 // Store
  ```

- Circular buffer: I register is buffer start index, B is buffer base address.

- Allows transmission two values of data to/from memory per cycle:
  ```
  f0 = DM(I0,M0), f1 = PM(I9,M8);
  ```

- Compiler allows to programmer to define which memory values are stored in.
DAGs registers

The DSP processor’s architecture

Circular buffer

<table>
<thead>
<tr>
<th>MEMORY ADDRESS</th>
<th>STORED VALUE</th>
<th>MEMORY ADDRESS</th>
<th>STORED VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>0</td>
<td>2000</td>
<td>0</td>
</tr>
<tr>
<td>2001</td>
<td>0.225767</td>
<td>2001</td>
<td>0.225767</td>
</tr>
<tr>
<td>2002</td>
<td>0.269847</td>
<td>2002</td>
<td>0.269847</td>
</tr>
<tr>
<td>2003</td>
<td>0.229918</td>
<td>2003</td>
<td>0.229918</td>
</tr>
<tr>
<td>2004</td>
<td>0.113940</td>
<td>2004</td>
<td>0.113940</td>
</tr>
<tr>
<td>2005</td>
<td>0.048679</td>
<td>2005</td>
<td>0.062222</td>
</tr>
<tr>
<td>2006</td>
<td>0.225767</td>
<td>2006</td>
<td>0.225767</td>
</tr>
<tr>
<td>2007</td>
<td>0.269847</td>
<td>2007</td>
<td>0.269847</td>
</tr>
<tr>
<td>2008</td>
<td>0.229918</td>
<td>2008</td>
<td>0.229918</td>
</tr>
<tr>
<td>2009</td>
<td>0.113940</td>
<td>2009</td>
<td>0.113940</td>
</tr>
<tr>
<td>2010</td>
<td>0.048679</td>
<td>2010</td>
<td>0.062222</td>
</tr>
<tr>
<td>2011</td>
<td>0.225767</td>
<td>2011</td>
<td>0.225767</td>
</tr>
<tr>
<td>2012</td>
<td>0.269847</td>
<td>2012</td>
<td>0.269847</td>
</tr>
<tr>
<td>2013</td>
<td>0.229918</td>
<td>2013</td>
<td>0.229918</td>
</tr>
<tr>
<td>2014</td>
<td>0.113940</td>
<td>2014</td>
<td>0.113940</td>
</tr>
<tr>
<td>2015</td>
<td>0.048679</td>
<td>2015</td>
<td>0.062222</td>
</tr>
</tbody>
</table>

a. Circular buffer at some instant

b. Circular buffer after next sample
Circular buffer

- A circular buffer is an array of n elements; when the n + 1th element is referenced, the reference goes to buffer location 0, wrapping around from the end to the beginning of the buffer.

SHARC assembly language

M6 = 1;
R0 = dm(I4, M6); // post-modify
// means: R0 = dm(I4), and then I4 = I4 + M6
// However:
R0 = dm(M6, I4); // offset index only
// means: R0 = dm(M6 +I4), and still keeps I4 = I4
### SHARC assembly language

#### Post-incrementing and Offset

B4 = 4000;
I4 = 0; // set to 0
I4 = 4002;
M6 = 1;

R0 = dm(M6, I4); // offset index only
R1 = dm(M6, I4); // offset index only
// means R0 = dm(4002 + 1) and R1 = dm(4002 + 1)
// with I4 = 4002 still unchanged at the end of the code

R0 = dm(I4, M6); // post-modify
R1 = dm(I4, M6); // post-modify
// means R0 = dm(4002) and R1 = dm(4003)
// with I4 = 4004 at the end of the code

#### Circular buffer implementation

B4 = 4000;
I4 = 3;
I4 = 4002;
M6 = 1;

R0 = dm(M6, I4); // offset index only
R1 = dm(M6, I4); // offset index only
// means R0 = dm(4002 + 1) and R1 = dm(4002 + 1)
// with I4 = 4002 still

R0 = dm(I4, M6); // post-increment
R1 = dm(I4, M6); // post-increment
// means R0 = dm(4002) with I4 = 4003,
// however R1 = dm(4000) {4003 - 3} with I4 = 4001
Example: C assignments

C:
\[ x = (a + b) - c; \]

Assembler:
\[
\begin{align*}
    r0 &= DM(_a) \quad // Load a \\
    r1 &= DM(_b); \quad // Load b \\
    r3 &= r0+r1; \\
    r2 &= DM(_c); \quad // Load c \\
    r3 &= r3-r2; \\
    DM(_x) &= r3; \quad // Store result in x
\end{align*}
\]

Example: C assignments

C:
\[ y = a \times (b+c); \]

Assembler:
\[
\begin{align*}
    r1 &= DM(_b); \quad // Load b \\
    r2 &= DM(_c); \quad // Load c \\
    r2 &= r1 + r2; \\
    r0 &= DM(_a); \quad // Load a \\
    r2 &= r2 \times r0; \\
    DM(_y) &= r2; \quad // Store result in y
\end{align*}
\]
Example: C assignments

✔ Shorter version using pointers:

```c
// Load b, c
r2 = DM(I1,M5), r1 = PM(I8,M13);
// load a in parallel with multiplication
r0 = r2+r1, r12 = DM(I0,M5);
r8 = r12*r0;
DM(I0,M5)= r8; // Store in y
```

Example: C assignments

✔ C:

```
z = (a << 2) | (b & 15);
```

✔ Assembler:

```
r0 = DM(_a); // Load a
r0 = LSHIFT r0 by 2; // Left shift
r1 = DM(_b), r3 = 15;// Load immediate
r1 = r1 AND r3;
r0 = r1 OR r0;
DM(_z) = r0;
```
Unconditional flow of control change:

JUMP label;

Three addressing modes:
- Direct (specifies a 24-bit address in immediate);
- Indirect (supply by DAG2 data address generator);
- PC-relative (specifies an immediate value that is added to the current PC).

All Instructions may be executed conditionally
- if EQ r1=pm(i15,0x11);
- if LE r0 = LSHIFT r0 by 2;

Conditions come from:
- arithmetic status (ASTAT)
- mode control 1 (MODE1)
- loop register

Example: C if statement

C:
    if (a > b)
        y = c + d;
    else y = c - d;

Assembler:

// if condition
r0 = DM(_a);
r1 = DM(_b);
COMP(r0,r1); // Compare
IF GT JUMP label;
// False block
r0 = DM(_c);
r1 = DM(_d);
r1 = r0 - r1;
DM(_y) = r1;
JUMP other; // Skip false block
// True block
label: r0 = DM(_c);
r1 = DM(_d);
r1 = r0 + r1;
DM(_y) = r1;
other: // Code after if
The best if implementation

- **C:**
  ```c
  if (a > b)
      y = c + d;
  else y = c - d;
  ```

- **Assembler:**
  ```assembly
  // Load values
  r1 = DM(_a), r2 = PM(_b);
  r3 = DM(_c), r4 = PM(_d);
  // Compute both sum and difference
  r12 = r3 + r4, r0 = r3 - r4;
  // Choose which one to save
  comp(r2,r1);
  if GT r0 = r12;
  dm(_y) = r0 // Write to y
  ```

DO UNTIL loops

The SHARC processor allows up to six nested loops

DO UNTIL instruction provides efficient looping:

- LCNTR = 30, DO label UNTIL LCE;
- r0 = DM(I0,M0), r2 = PM(I8,M8);
- r1 = r0 - r15;
- label: r4 = f2 + f3;

**Loop length (16 bit)**
- **Last instruction in loop**
- **Termination condition**
  - LCE: Loop counter expired
  - NOT LCE: Loop counter not expired

Another version of loop:

- DO label UNTIL EQ;
- R0 = R0-1;
- label: comp(R0,R1);
Example: FIR filter

C:
```c
for (n=0; n<N-M; n++) {
    acc = 0;
    for (i=0; i<M; i++)
        acc += a[i]*x[n+i];
    y[n] = acc;
}
```

FIR filter assembler

```c
// setup
I0 = _a; I8 = _x; // a[0] (DAG0), x[0] (DAG1)
r12 = 0; // f = 0;
M0 = 1; M8 = 1; // Set up increments
// Loop body
LCNTR = N, DO loopend UNTIL LCE;
// Use post-increment mode
r1 = DM(I0,M0), r2 = PM(I8,M8);
r8 = r1 * r2 (uui);
loopend: r12 = r12 + r8;
```
Example: C main + ASM function

C:
```c
int dm c[4] = {1,2,3,4};
int pm x[7] = {1,2,3,4,5,6,7};

int dm y;
extern int fir(int dm *,int pm *);

//main
void main()
{
  y = fir(c,x);
}
```

Assembler:
```asm
#include <asm_sprt.h>
.SEGMENT/PM seg_pmco;
.global _fir;
.extern _c, _x, _y;
_fir: entry;
  // setup
  I0=_c; I8=_x; // c[0](DAG0),x[0](DAG1)
  // or I0 = r4, I8 = r8
  r12 = 0; // i = 0;
  M0=1; M8=1; // Set up increments
  // Loop body
  LCNTR = 4, DO loopend UNTIL LCE;
  r1 = DM(I0,M0), r2 = FM(I8,M8);
  r3 = r1 * r2 (ssi);
  loopend: r12 = r12 + r3;
  r0 = r12; // or dm(_:y)=r12;
  exit;
_fir.end: .ends;
```
Example: Using MAC operation

Assembler:

```c
#include <asm_sprt.h>
.SEGMENT/PM seg_pmco;
global _fir;
.extern _c, _x, _y;
_fir: entry;
//setup
    IO =_c; I8 = _x; // c[0](DAG0),x[0](DAG1)
//or IO = r4, I8 = r8
    r12 = 0;       // f = 0;
    M0=1; M8=1;    // Set up increments
//Loop body
    LCNTR = 4, DO loopend UNTIL LCE;
    r1 = DM(IO,M0), r2 = PM(I8,M8);
    loopend: MRF = MRF + r1 * r2 (ssi);
    r0 = MR0F;
    exit;
_fir.end:
.endseg;
```

Example: C main + ASM function (work with STACK)

```c
int a,b,c,d,e,f;

extern int asm_proc( int a, int b, int c,
                     int d, int e );

void main()
{
    a = 0xAAAAAA;
    b = 0xBBBBBB;
    c = 0xCCCCCC;
    d = 0xDDDDDD;
    e = 0xEEEEEE;
    f = asm_proc(a,b,c,d,e);
}
```
Example: C main + ASM function (work with STACK)

```c
#include "asm_sprt.h"

.SEGMENT/PM     seg_pmco;
.GLOBAL _asm_proc;
_asm_proc:
start:
// m7 = -1 (compiler definition)
// m6 = 1 (compiler definition)
  r15 = i6;
// i6 - save C sp (stack pointer)
// i7 - asm sp (stack pointer)
  i2 = r15;
    modify(i2,m6);
r0 = r4;
r1 = r8;
r2 = r12;
r3 = dm(i2,m6);
  // C sp + 2 (fourth argument place)
  r4 = dm(i2,m6);
  // C sp + 3 (fifth argument place)
  r5 =0x555555;
  r0 = r0 + r5;
  // r0 = return()
_asm_proc.end:
.endseg;
exit;
```

Important programming reminders

- Registers for C function parameters transfer: r4, r8, r12
- C return with r0;
- Interrupt does not occur until 2 instructions after delayed branch (needs 2 NOPs);
- Some DAG register transfers are disallowed in assembler routine;
- Compiler definition m7 = -1, m6 = 1, m5 = 0
- C stack pointer - i6, asm stack pointer - i7
SOS filters creation

- Coefficients calculation for Bandpass filter

```matlab
% The cut off frequency normalization with fs/2 format long
Wn = [200 1000] / (fs/2);
N = 5
[b, a] = butter(N, Wn);
```

- Conversion zero-pole-gain filter parameters to second-order sections form

```matlab
[z, p, k] = butter(N, Wn);
[sos, g] = zp2sos(z, p, k);
```

Data preparation for using in VisualDSP environment

- Inversion of each array of coefficients and negate the a-coefficients

```matlab
>> b = flipud(b');
>> a = -flipud(a');
```

- Creation of two text files which will be coefficient and input data files.

- .dat file should look like this

```plaintext
-0.80080264666571,
-2.20207663611883,
-3.29772682581611,
2.46161336595189
```

05/19/2016
Implementing the IIR filter in C

- It's need to do is include filters.h. At the top of C program, insert the following:

```
#include <filters.h>
```

- Define the preprocessor variable TAPS:

```
#define TAPS 4
```

- Declare filter variables and initialize them. Make sure that 2 data files are in the same directory as .C file.

```
float dm states[TAPS+1];
float pm ACoeffs[TAPS] =
    { #include "acoeffs.dat" }; 
float pm BCoeffs[TAPS+1] = 
    { #include "bcoeffs.dat" }; 
```

Fixed-Point Design

- Digital signal processing algorithms
  - Often developed in floating point
  - Later mapped into fixed point for digital hardware realization

- Fixed-point digital hardware
  - Lower area
  - Lower power
  - Lower per unit production cost
**Fixed-Point Binary Representation**

- Representation of a number with integer and fractional part: \((d_{n-1}d_{n-2}...d_0.d_{-1}d_{-2}...d_{-m})_2\)

\[
N = \sum_{i=-m}^{n-1} d_i \cdot r^i
\]

- This is denoted as \(Qnm\) representation
- The binary point is implied
- It will affect the accuracy (dynamic range and precision) of the number
- Purely a programmer's convention and has no relationship with the hardware

---

**Fixed-Point Binary Representation**

\[x = 0100\ 1000\ 0001\ 1000b\]

- \(Q0.15\) => \(x = 2^{\times(-1)} + 2^{\times(-4)} + 2^{\times(-11)}+2^{\times(-12)}\)
- \(Q1.14\) => \(x = 2^{\times0} + 2^{\times(-3)} + 2^{\times(-10)} + 2^{\times(-11)}\)
- \(Q2.13\) => \(x = 2^{\times1} + 2^{\times(-2)} + 2^{\times(-9)} + 2^{\times(-10)}\)
- \(Q7.8\) => \(x = ?\)
- \(Q12.3\) => \(x = ?\)
**Coefficient Quantization**

Coefficient quantization effect on pole locations:

1. tightly spaced poles (e.g. for narrow band filters) imply high sensitivity of pole locations to coefficient quantization.
2. hence preference for low-order systems (parallel/cascade).

Example: implementation of a band-pass IIR 12-order filter.

Overflow error

- Signals and coefficients normalized in the range of $-1$ to $1$ for fixed-point arithmetic, the sum of two B-bit numbers may fall outside the range of $-1$ to $1$.

- Severely distorts the signal.

- Overflow handling
  - Saturation arithmetic
    - “Clips” the signal, although better than overflow
    - Should only be used to guarantee no overflow, but should not be the only solution
  - Scaling of signals and coefficients
Coefficient representation

- Fractional 2’s complement (Q) representation is used
- To avoid overflow, often scaling down by a power of two factor (S) (right shift) is used.
- The scaling factor is given by the equation: 
  \[ S = \text{Imax}(|h(0)| + |h(1)| + |h(2)| + \ldots) \]
- Furthermore, filter coefficient larger than 1, cause overflow and are scaled down further by a factor B, in order to be less than 1

Example

- Given the FIR filter
  - \[ y(n) = 0.1x(n) + 0.25x(n-1) + 0.2x(n-2) \]
  - Assuming the input range occupies \( \frac{1}{4} \) of the full range
  - Develop the DSP implementation equations in Q0.15 format. What is the coefficient quantization error?

- Solution:
  - \[ S = \frac{1}{4 \cdot (|h(0)| + |h(1)| + |h(2)|)} = \frac{1}{4 \cdot (0.1 + 0.25 + 0.2)} = \frac{1}{4} \times 0.575 = 0.14375 \]
  - Overflow cannot occur, no input (S) scaling required
  - No coefficients > 1, no coefficient (B) scaling required
Example 2

Given the FIR filter
- \( y(n) = 0.8x(n) + 3x(n-1) + 0.6x(n-2) \)
- Assuming the input range occupies \( \frac{1}{4} \) of the full range
- Develop the DSP implementation equations in Q0.15 format. What is the coefficient quantization error?

Solution:
- \( S = \frac{1}{4}(|h(0)| + |h(1)| + |h(2)|) = \frac{1}{4}(0.8 + 3 + 0.6) = 4.4/4 = 1.1 \)
- Therefore: \( S = 2 \)
- Largest coefficient: \( h(1) = 3 \), therefore \( B = 4 \)
- \( y_s(n) = 0.2x_s(n) + 0.75x_s(n-1) + 0.15x_s(n-2) \)

FIR Software Implementation

```c
int yn=0; //filter output initialization
short xdly[N+1]; //input delay samples array

interrupt void c_int11() //ISR
{
    short i;
    yn=0;
    short h[N] = {}; //coefficients
    xdly[0]=input_sample();
    for (i=0; i<N; i++)
        yn += (h[i]*xdly[i]);
    for (i=N-1; i>0; i--)
        xdly[i] = xdly[i-1];
    output_sample(yn >> 15); //filter output
    return; //return from ISR
}
```
The DSP processor ‘s architecture

DSP processors with fixed and floating point.

**Floating point advantages:**
- Increases accuracy;
- Wide dynamic range;
- Doesn’t have problem with data overflow;
- Friendly for C compiler.

**Fixed point advantages:**
- Cheaper;
- Compact.

**Fixed versus Floating:**
- Fixed point arithmetic operations are more simple for hardware realization;
- Floating point DSP processor has more data types and commands;

Fixed point data preparation

- To normalize a and b coefficients according to example
- To scale input to 1/4 of dynamic range
- To round a, b and input
- Save a, b and input to files as described in ee-83
- Define buffers as integers in C code.
- Prepare Fixed point simulation and compare that with flouting point
- Implement that in VisualDSP++
Getting started
```c
int main()
{
    printf("Hello SHARC!");
    return 0;
}
```
48 DSP lab
Paths to examples

C:\Program Files\Analog Devices\VisualDSP\211xx\examples
C:\Program Files\Analog Devices\VisualDSP\21k\examples
or
D:\Program Files\Analog Devices\VisualDSP\211xx\examples
D:\Program Files\Analog Devices\VisualDSP\21k\examples

Filter Implementation

- Finite word-length effects (fixed point implementation)
  - Coefficient quantization
  - Overflow & quantization in arithmetic operations
    - scaling to prevent overflow
    - quantization noise statistical modeling
    - limit cycle oscillations
Coefficient Quantization

Coefficient quantization effect on pole locations:
1. tightly spaced poles (e.g. for narrow band filters) imply high sensitivity of pole locations to coefficient quantization
2. hence preference for low-order systems (parallel/cascade)

Example: Implementation of a band-pass IIR 12-order filter

Cascade structure with 16-bit coeff. Direct form with 16-bit coeff.

Introduction to DSP processors

The END