MANCHESTER 1824
The University of Manchester

Digital Signal Processing (EEEN30029)

Laboratory - Real Time DSP Systems

February 24, 2017

Lecturer: Prof. Gaydecki

Word count: 1499

Students: Rebekah King, James Grierson, Jamol Khushvaktov, Thomas Hollis

Exercise 1 - Gain1

```
/*
Program gain1 is a simple inout program but multiplies the left channel by 3.7
Author: Patrick Gaydecki
Date  : 12.10.2017
*/

.section program;
.align 4;
.global _main;
#include <defBF706.h> //loads all required definitions for the BF706 chip

_main:
call codec_configure; //runs the codec configuration function (further details below)
call sport_configure; //runs the sport configuration function (further details below)

// **** INSERT CODE HERE ****
R1.h=0.925r; // [INSERTED] R1.h stores the multiplier required later to achieve multiplication
of 3.7 (because after two shifts, i.e. x4, if you multiply 4 by 0.925 you get a gain of 3.7)

get_audio: //label for outer loop

wait_left: //label for inner loop (left chan)
// Wait left flag in and read
R0=[REG_SPORT0_CTL_B]; CC=BITTST(R0, 31); if !CC jump wait_left; //will wait until bit goes
high which indicates new data is available before proceeding
R0=[REG_SPORT0_RXPRI_B]; //writes input data to register 0

// **** INSERT CODE HERE ****
R0<<=0x02; // [INSERTED] Data in R0 is shifted by two hence it is effectively multiplied by 4
R0=R0.h*R1.h; // [INSERTED] 4*R0 is 16bit-multiplied by 0.925 which gives an overall 32bit-
destination multiplication of 3.7 for the required gain of 3.7

// Write left out
[REG_SPORT0_TXPRI_A]=R0; //writes register 0 data to output (here after multiplication
operations)

wait_right: //label for inner loop (right chan)

// Wait right flag in and read
R0=[REG_SPORT0_CTL_B]; CC=BITTST(R0, 31); if !CC jump wait_right; //will wait until bit goes
high which indicates new data is available before proceeding
// Write right out
R0=[REG_SPORT0_RXPRI_B]; //writes input data to register 0
[REG_SPORT0_TXPRI_A]=R0; //writes register 0 data to output (here no operations made)

jump get_audio; //return to outer loop indefinitely

rts;
._main.end:

// Function codec_configure initialises the ADAU1761 codec. Refer to the control register
// descriptions, page 51 onwards of the ADAU1761 data sheet.
codec_configure:
[--SP] = RETS; // Push stack (only for nested calls)
R1=0x01(X); R0=0x4000(X); call TWI_write; // Enable master clock, disable PLL
R1=0x7f(X); R0=0x40f9(X); call TWI_write; // Enable all clocks
R1=0x03(X); R0=0x40fa(X); call TWI_write; // Enable all clocks
R1=0x01(X); R0=0x4015(X); call TWI_write; // Set serial port master mode
R1=0x13(X); R0=0x4019(X); call TWI_write; // Set ADC to on, both channels
R1=0x21(X); R0=0x401c(X); call TWI_write; // Enable left channel mixer
R1=0x41(X); R0=0x401e(X); call TWI_write; // Enable right channel mixer
R1=0x03(X); R0=0x4029(X); call TWI_write; // Turn on power, both channels
R1=0x03(X); R0=0x402a(X); call TWI_write; // Set both DACs on
```

```
R1=0x01(X); R0=0x40f2(X); call TWI_write; // DAC gets L, R input from serial port
R1=0x01(X); R0=0x40f3(X); call TWI_write; // ADC sends L, R input to serial port
R1=0x0b(X); R0=0x400a(X); call TWI_write; // Set left line-in gain to 0 dB
R1=0x0b(X); R0=0x400c(X); call TWI_write; // Set right line-in gain to 0 dB
R1=0xe7(X); R0=0x4023(X); call TWI_write; // Set left headphone volume to 0 dB
R1=0xe7(X); R0=0x4024(X); call TWI_write; // Set right headphone volume to 0 dB
R1=0x00(X); R0=0x4017(X); call TWI_write; // Set codec default sample rate, 48 kHz
nop;
RETS = [SP++];                            // Pop stack (only for nested calls)
rts;
codec_configure.end:

// Function sport_configure initialises the SPORT0. Refer to pages 26-59, 26-67,
// 26-75 and 26-76 of the ADSP-BF70x Blackfin+ Processor Hardware Reference manual.
sport_configure:
R0=0x3F0(X); [REG_PORTC_FER]=R0;          // Set up Port C in peripheral mode
R0=0x3F0(X); [REG_PORTC_FER_SET]=R0;      // Set up Port C in peripheral mode
R0=0x2001973; [REG_SPORT0_CTL_A]=R0;      // Set up SPORT0 (A) as TX to codec, 24 bits
R0=0x0400001; [REG_SPORT0_DIV_A]=R0;      // 64 bits per frame, clock divisor of 1
R0=0x1973(X); [REG_SPORT0_CTL_B]=R0;      // Set up SPORT0 (B) as RX frm codec, 24 bits
R0=0x0400001; [REG_SPORT0_DIV_B]=R0;      // 64 bits per frame, clock divisor of 1
rts;
sport_configure.end:

// Function TWI_write is a simple driver for the TWI. Refer to page 24-15 onwards
// of the ADSP-BF70x Blackfin+ Processor Hardware Reference manual.
TWI_write:
R3=R0 <<0x8; R0=R0 >>>0x8; R2=R3|R0;      // Reverse low order and high order bytes
R0=0x3232(X); [REG_TWI0_CLKDIV]=R0;       // Set duty cycle
R0=0x008c(X); [REG_TWI0_CTL]=R0;          // Set pre-scale and enable TWI
R0=0x0038(X); [REG_TWI0_MSTRADDR]=R0;     // Address of codec
[REG_TWI0_TXDATA16]=R2;                   // Address of register to set, LSB then MSB
R0=0x00c1(X); [REG_TWI0_MSTRCTL]=R0;      // Command to send three bytes and enable TX
[--SP] = RETS; call delay; RETS = [SP++]; // Delay
[REG_TWI0_TXDATA8]=R1;                    // Data to write
[--SP] = RETS; call delay; RETS = [SP++]; // Delay
R0=0x050; [REG_TWI0_ISTAT]=R0;            // Clear TXERV interrupt
[--SP] = RETS; call delay; RETS = [SP++]; // Delay
R0=0x010; [REG_TWI0_ISTAT]=R0;            // Clear MCOMP interrupt
rts;
TWI_write.end:

// Function delay introduces a delay to allow TWI communication
delay:
p0=0x8000; //sets delay timing
loop lc0=p0; //inner NOP loop for waiting
nop; nop; nop; //does nothing i.e. waits
loop_end; //return to inner loop
rts;
delay.end:
```

Exercise 2 - Gain2

```
/*
Program gain2 is a simple inout program but reduces the gain on the left channel by 4 dB, (i.e.
0.631)
using a command to the codec

Author: Patrick Gaydecki
Date  : 15.10.2017
*/

.section program;
.align 4;
.global _main;
#include <defBF706.h>

_main:
call codec_configure;
call sport_configure;
get_audio:
wait_left:
// Wait left flag in and read
R0=[REG_SPORT0_CTL_B]; CC=BITTST(R0, 31); if !CC jump wait_left;
// Write left out
R0=[REG_SPORT0_RXPRI_B]; [REG_SPORT0_TXPRI_A]=R0;
wait_right:
// Wait right flag in and read
R0=[REG_SPORT0_CTL_B]; CC=BITTST(R0, 31); if !CC jump wait_right;
// Write right out
R0=[REG_SPORT0_RXPRI_B]; [REG_SPORT0_TXPRI_A]=R0;
jump get_audio;
._main.end:

// Function codec_configure initialises the ADAU1761 codec. Refer to the control register
// descriptions, page 51 onwards of the ADAU1761 data sheet.
codec_configure:
[--SP] = RETS; // Push stack (only for nested calls)
R1=0x01(X); R0=0x4000(X); call TWI_write; // Enable master clock, disable PLL
R1=0x7f(X); R0=0x40f9(X); call TWI_write; // Enable all clocks
R1=0x03(X); R0=0x40fa(X); call TWI_write; // Enable all clocks
R1=0x01(X); R0=0x4015(X); call TWI_write; // Set serial port master mode
R1=0x13(X); R0=0x4019(X); call TWI_write; // Set ADC to on, both channels
R1=0x21(X); R0=0x401c(X); call TWI_write; // Enable left channel mixer
R1=0x41(X); R0=0x401e(X); call TWI_write; // Enable right channel mixer
R1=0x03(X); R0=0x4029(X); call TWI_write; // Turn on power, both channels
R1=0x03(X); R0=0x402a(X); call TWI_write; // Set both DACs on
R1=0x01(X); R0=0x40f2(X); call TWI_write; // DAC gets L, R input from serial port
R1=0x01(X); R0=0x40f3(X); call TWI_write; // ADC sends L, R input to serial port
R1=0x0b(X); R0=0x400a(X); call TWI_write; // Set left line-in gain to 0 dB
R1=0x0b(X); R0=0x400c(X); call TWI_write; // Set right line-in gain to 0 dB

R1=0xd7(X); R0=0x4023(X); call TWI_write; // [MODIFIED] Set left headphone volume to -4 dB.
Since we need LHPM=1, HPEN=1 and a gain of -4dB thus since LHPVOL ranges from 000000-111111 so
we need 215 -> d7 (in hex)

R1=0xe7(X); R0=0x4024(X); call TWI_write; // Set right headphone volume to 0 dB
R1=0x00(X); R0=0x4017(X); call TWI_write; // Set codec default sample rate, 48 kHz
nop;
RETS = [SP++];                            // Pop stack (only for nested calls)
rts;
codec_configure.end:

// Function sport_configure initialises the SPORT0. Refer to pages 26-59, 26-67,
// 26-75 and 26-76 of the ADSP-BF70x Blackfin+ Processor Hardware Reference manual.
sport_configure:
```

```
R0=0x3F0(X); [REG_PORTC_FER]=R0;          // Set up Port C in peripheral mode
R0=0x3F0(X); [REG_PORTC_FER_SET]=R0;      // Set up Port C in peripheral mode
R0=0x2001973; [REG_SPORT0_CTL_A]=R0;      // Set up SPORT0 (A) as TX to codec, 24 bits
R0=0x0400001; [REG_SPORT0_DIV_A]=R0;      // 64 bits per frame, clock divisor of 1
R0=0x1973(X); [REG_SPORT0_CTL_B]=R0;      // Set up SPORT0 (B) as RX frm codec, 24 bits
R0=0x0400001; [REG_SPORT0_DIV_B]=R0;      // 64 bits per frame, clock divisor of 1
rts;
sport_configure.end:

// Function TWI_write is a simple driver for the TWI. Refer to page 24-15 onwards
// of the ADSP-BF70x Blackfin+ Processor Hardware Reference manual.
TWI_write:
R3=R0 <<0x8; R0=R0 >>>0x8; R2=R3|R0;      // Reverse low order and high order bytes
R0=0x3232(X); [REG_TWI0_CLKDIV]=R0;       // Set duty cycle
R0=0x008c(X); [REG_TWI0_CTL]=R0;          // Set pre-scale and enable TWI
R0=0x0038(X); [REG_TWI0_MSTRADDR]=R0;     // Address of codec
[REG_TWI0_TXDATA16]=R2;                   // Address of register to set, LSB then MSB
R0=0x00c1(X); [REG_TWI0_MSTRCTL]=R0;      // Command to send three bytes and enable TX
[--SP] = RETS; call delay; RETS = [SP++]; // Delay
[REG_TWI0_TXDATA8]=R1;                    // Data to write
[--SP] = RETS; call delay; RETS = [SP++]; // Delay
R0=0x050; [REG_TWI0_ISTAT]=R0;            // Clear TXERV interrupt
[--SP] = RETS; call delay; RETS = [SP++]; // Delay
R0=0x010; [REG_TWI0_ISTAT]=R0;            // Clear MCOMP interrupt
rts;
TWI_write.end:

// Function delay introduces a delay to allow TWI communication
delay:
p0=0x8000;
loop lc0=p0;
nop; nop; nop;
loop_end;
rts;
delay.end:
```

Exercise 3 - Modulate

```
/*
Program modulate takes data from both codec input channels and generates the following ouputs:
Right channel out = left x right (pure modulation)
Left channel out =  right + (left x right) (double sideband amplitude modulation, or DSBAM

Author: Patrick Gaydecki
Date  : 28.09.2017
*/

.section program;
.align 4;
.global _main;
#include <defBF706.h>

_main:
call codec_configure;
call sport_configure;
get_audio: //audio in composed of 300Hz & 15kHz

wait_left:
// Wait left flag in
R0=[REG_SPORT0_CTL_B];
CC=BITTST(R0, 31);
if !CC jump wait_left;
// Read and modulate. Note codec is 24-bit;
R0=[REG_SPORT0_RXPRI_B];

// **** INSERT CODE HERE ****
R3=R0*R1;  // [INSERTED] Multiply right & left channels together
R3<<=0x08; // [INSERTED] Shift result to bring in the codex range
R2=R3+R0; // [INSERTED] Sum the multiplication of both channels with left channel

[REG_SPORT0_TXPRI_A]=R2; //Write left out

wait_right:
// Wait right flag in
R1=[REG_SPORT0_CTL_B];
CC=BITTST(R1, 31);
if !CC jump wait_right;
// Read and modulate. Note codec is 24-bit;
// Also add carrier
R1=[REG_SPORT0_RXPRI_B];

// **** INSERT CODE HERE ****
R3 = R1*R0; // [INSERTED] Multiply right & left channels together
R3<<=0x08; // [INSERTED] Shift result to bring in the codex range

[REG_SPORT0_TXPRI_A]=R3; //Write right out

jump get_audio;
._main.end:

// Function codec_configure initialises the ADAU1761 codec. Refer to the control register
// descriptions, page 51 onwards of the ADAU1761 data sheet.
codec_configure:
[--SP] = RETS; // Push stack (only for nested calls)
R1=0x01(X); R0=0x4000(X); call TWI_write; // Enable master clock, disable PLL
R1=0x7f(X); R0=0x40f9(X); call TWI_write; // Enable all clocks
R1=0x03(X); R0=0x40fa(X); call TWI_write; // Enable all clocks
R1=0x01(X); R0=0x4015(X); call TWI_write; // Set serial port master mode
R1=0x13(X); R0=0x4019(X); call TWI_write; // Set ADC to on, both channels
R1=0x21(X); R0=0x401c(X); call TWI_write; // Enable left channel mixer
R1=0x41(X); R0=0x401e(X); call TWI_write; // Enable right channel mixer
```

```
R1=0x03(X); R0=0x4029(X); call TWI_write; // Turn on power, both channels
R1=0x03(X); R0=0x402a(X); call TWI_write; // Set both DACs on
R1=0x01(X); R0=0x40f2(X); call TWI_write; // DAC gets L, R input from serial port
R1=0x01(X); R0=0x40f3(X); call TWI_write; // ADC sends L, R input to serial port
R1=0x0b(X); R0=0x400a(X); call TWI_write; // Set left line-in gain to 0 dB
R1=0x0b(X); R0=0x400c(X); call TWI_write; // Set right line-in gain to 0 dB
R1=0xe7(X); R0=0x4023(X); call TWI_write; // Set left headphone volume to 0 dB
R1=0xe7(X); R0=0x4024(X); call TWI_write; // Set right headphone volume to 0 dB
R1=0x00(X); R0=0x4017(X); call TWI_write; // Set codec default sample rate, 48 kHz
nop;
RETS = [SP++];                            // Pop stack (only for nested calls)
rts;
codec_configure.end:

// Function sport_configure initialises the SPORT0. Refer to pages 26-59, 26-67,
// 26-75 and 26-76 of the ADSP-BF70x Blackfin+ Processor Hardware Reference manual.
sport_configure:
R0=0x3F0(X); [REG_PORTC_FER]=R0;          // Set up Port C in peripheral mode
R0=0x3F0(X); [REG_PORTC_FER_SET]=R0;      // Set up Port C in peripheral mode
R0=0x2001973; [REG_SPORT0_CTL_A]=R0;      // Set up SPORT0 (A) as TX to codec, 24 bits
R0=0x0400001; [REG_SPORT0_DIV_A]=R0;      // 64 bits per frame, clock divisor of 1
R0=0x1973(X); [REG_SPORT0_CTL_B]=R0;      // Set up SPORT0 (B) as RX frm codec, 24 bits
R0=0x0400001; [REG_SPORT0_DIV_B]=R0;      // 64 bits per frame, clock divisor of 1
rts;
sport_configure.end:

// Function TWI_write is a simple driver for the TWI. Refer to page 24-15 onwards
// of the ADSP-BF70x Blackfin+ Processor Hardware Reference manual.
TWI_write:
R3=R0 <<0x8; R0=R0 >>>0x8; R2=R3|R0;      // Reverse low order and high order bytes
R0=0x3232(X); [REG_TWI0_CLKDIV]=R0;       // Set duty cycle
R0=0x008c(X); [REG_TWI0_CTL]=R0;          // Set pre-scale and enable TWI
R0=0x0038(X); [REG_TWI0_MSTRADDR]=R0;     // Address of codec
[REG_TWI0_TXDATA16]=R2;                   // Address of register to set, LSB then MSB
R0=0x00c1(X); [REG_TWI0_MSTRCTL]=R0;      // Command to send three bytes and enable TX
[--SP] = RETS; call delay; RETS = [SP++]; // Delay
[REG_TWI0_TXDATA8]=R1;                    // Data to write
[--SP] = RETS; call delay; RETS = [SP++]; // Delay
R0=0x050; [REG_TWI0_ISTAT]=R0;            // Clear TXERV interrupt
[--SP] = RETS; call delay; RETS = [SP++]; // Delay
R0=0x010; [REG_TWI0_ISTAT]=R0;            // Clear MCOMP interrupt
rts;
TWI_write.end:

// Function delay introduces a delay to allow TWI communication
delay:
p0=0x8000;
loop lc0=p0;
nop; nop; nop;
loop_end;
rts;
delay.end:
```

Exercise 4 - Filter1

```
/*
Program filter1 is a single MAC FIR filtering program. It reads in a file of coefficients
created by Signal Wizard.
Also included is a commented-out .BYTE buffer, which shows how coefficients may be hard-coded
into the program.

Author: Patrick Gaydecki
Date  : 29.09.2017
*/

.section L1_data_b;  // Linker places data starting at 0x11900000

// **** INSERT CODE HERE ****
.BYTE2 filter[]= "2kBandPass.txt"; // [INSERTED] Imports the (16-bit) 2k BandPass filter
coeficient file

.section program;
.global _main;
.align 4;
# include <defBF706.h>

_main:
call codec_configure;
call sport_configure;
P0=length(filter)*2;

// Set up circular buffers

// **** INSERT CODE HERE ****
I0=0x11800000;B0=I0;L0=P0; // [INSERTED] Circular buffer initialised for input data (Base
register, Index register, Length register & Pointer register)
I1=0x11900000;B1=I1;L1=P0; // [INSERTED] Circular buffer initialised for filter coefficients
(Base register, Index register, Length register & Pointer register)
P0=length(filter)-1; // [INSERTED] Pointer register P0 is now intialised to hold the number of
filter coeficients as defined in line 12 (i.e. k-1 iterations)

get_audio:
wait_left:
// Wait for left data then read
R0=[REG_SPORT0_CTL_B];
CC=BITTST(R0, 31);
if !CC jump wait_left;
R0=[REG_SPORT0_RXPRI_B];

// Convolution kernel

// **** INSERT CODE HERE ****
A0 = 0 || W[I0++]= R0.H || R1.H = W[I1++]; // [INSERTED] Acculumator register A0 initialised to
0 (cleared), Input audio data in R0.H fed into circular buffer I0 initialised before (16-bit),
Filter coeficients in index register I0 are fed into R1.H (16-bit) then post-increments both
for the next instruction. (exploits parallelism)
LOOP LC0 = P0; // [INSERTED] Start of convolution loop containing loop counter register LC0,
controlled by condition in pointer register P0 (i.e. filter length-1 iterations)
A0 += R0.H * R1.H || R0.H = W[I0++] || R1.H = W[I1++]; // [INSERTED] Filter left by multiplying
current input audio data in R0.H by current filter coeficients in R1.H via a 16-bit multiply
into a 32-bit accumlator register A0. Then increments both I0 and I1 for the next
multiplication. This shifted multiply accumulate (MAC) is the heart of convolution. (exploits
parallelism)
LOOP_END; // [INSERTED] End of convolution loop
A0 += R0.H * R1.H || I0-=2; // [INSERTED] 16-bit multiplies the final values of input audio
data in R0.H and filter coeficients in R1.H into 32-bit accumulator register A0. In addition,
post-decrements the I0 register by two bites (i.e. 16-bit word) in order to ensure that the
```

```
next value from the codec overwrites the oldest value, not the previously most recently
acquired. (exploits parallelism)

//Write left out
R0=A0; [REG_SPORT0_TXPRI_A]=R0;
wait_right:

// Wait for right data then read
R0=[REG_SPORT0_CTL_B];
CC=BITTST(R0, 31);
if !CC jump wait_right;
R0=[REG_SPORT0_RXPRI_B];
//Write right out
[REG_SPORT0_TXPRI_A]=R0;
jump get_audio;
rts;
._main.end:

// Function codec_configure initialises the ADAU1761 codec. Refer to the control register
// descriptions, page 51 onwards of the ADAU1761 data sheet.
codec_configure:
[--SP] = RETS;                              // Push stack (only for nested calls)
R1=0x01(X); R0=0x4000(X); call TWI_write; // Enable master clock, disable PLL
R1=0x7f(X); R0=0x40f9(X); call TWI_write; // Enable all clocks
R1=0x03(X); R0=0x40fa(X); call TWI_write; // Enable all clocks
R1=0x01(X); R0=0x4015(X); call TWI_write; // Set serial port master mode
R1=0x13(X); R0=0x4019(X); call TWI_write; // Set ADC to on, both channels
R1=0x21(X); R0=0x401c(X); call TWI_write; // Enable left channel mixer
R1=0x41(X); R0=0x401e(X); call TWI_write; // Enable right channel mixer
R1=0x03(X); R0=0x4029(X); call TWI_write; // Turn on power, both channels
R1=0x03(X); R0=0x402a(X); call TWI_write; // Set both DACs on
R1=0x01(X); R0=0x40f2(X); call TWI_write; // DAC gets L, R input from serial port
R1=0x01(X); R0=0x40f3(X); call TWI_write; // ADC sends L, R input to serial port
R1=0x0b(X); R0=0x400a(X); call TWI_write; // Set left line-in gain to 0 dB
R1=0x0b(X); R0=0x400c(X); call TWI_write; // Set right line-in gain to 0 dB
R1=0xe7(X); R0=0x4023(X); call TWI_write; // Set left headphone volume to 0 dB
R1=0xe7(X); R0=0x4024(X); call TWI_write; // Set right headphone volume to 0 dB
R1=0x00(X); R0=0x4017(X); call TWI_write; // Set codec default sample rate, 48 kHz
NOP;
RETS = [SP++];                              // Pop stack (only for nested calls)
RTS;
codec_configure.end:

// Function sport_configure initialises the SPORT0. Refer to pages 26-59, 26-67,
// 26-75 and 26-76 of the ADSP-BF70x Blackfin+ Processor Hardware Reference manual.
sport_configure:
R0=0x3F0(X); [REG_PORTC_FER]=R0;          // Set up Port C in peripheral mode
R0=0x3F0(X); [REG_PORTC_FER_SET]=R0;      // Set up Port C in peripheral mode
R0=0x2001973; [REG_SPORT0_CTL_A]=R0;      // Set up SPORT0 (A) as TX to codec, 24 bits
R0=0x0400001; [REG_SPORT0_DIV_A]=R0;      // 64 bits per frame, clock divisor of 1
R0=0x1973(X); [REG_SPORT0_CTL_B]=R0;      // Set up SPORT0 (B) as RX from codec, 24 bits
R0=0x0400001; [REG_SPORT0_DIV_B]=R0;      // 64 bits per frame, clock divisor of 1
RTS;
sport_configure.end:

// Function TWI_write is a simple driver for the TWI. Refer to page 24-15 onwards
// of the ADSP-BF70x Blackfin+ Processor Hardware Reference manual.
TWI_write:
R3=R0 <<0x8; R0=R0 >>>0x8; R2=R3|R0;      // Reverse low order and high order bytes
R0=0x3232(X); [REG_TWI0_CLKDIV]=R0;       // Set duty cycle
R0=0x008c(X); [REG_TWI0_CTL]=R0;          // Set prescale and enable TWI
R0=0x0038(X); [REG_TWI0_MSTRADDR]=R0;     // Address of codec
[REG_TWI0_TXDATA16]=R2;                   // Address of register to set, LSB then MSB
R0=0x00c1(X); [REG_TWI0_MSTRCTL]=R0;      // Command to send three bytes and enable tx
[--SP] = RETS; call delay; RETS = [SP++]; // Delay
```

```
[REG_TWI0_TXDATA8]=R1;                    // Data to write
[--SP] = RETS; call delay; RETS = [SP++]; // Delay
R0=0x050; [REG_TWI0_ISTAT]=R0;            // Clear TXERV interrupt
[--SP] = RETS; call delay; RETS = [SP++]; // Delay
R0=0x010; [REG_TWI0_ISTAT]=R0;            // Clear MCOMP interrupt
rts;
TWI_write.end:

// Function delay introduces a delay to allow TWI communication
delay:
P0=0x8000;
loop LC0=P0;
NOP; NOP; NOP;
loop_end;
RTS;
delay.end:
```

Exercise 5 - Filter2

```
/*
Program filter2 is a single MAC FIR filtering program similar to filter1. However, this is a
32-bit version, requiring
a combined A1:A0 72-bit accumulator register. It can handle up to 8050 coefficients, although
the available memory
limits this to 7680 (unless using an L2 to L1 transfer).

Author: Patrick Gaydecki
Date  : 29.09.2017
*/

.section L1_data_b;  // Linker places data starting at 0x11900000

// **** INSERT CODE HERE ****
.BYTE4/r32 filter[]= "Wav4Filter.txt"; // [INSERTED] Imports the (32-bit) stop band filter
coeficient file

.section program;
.global _main;
.align 4;
# include <defBF706.h>

_main:
call codec_configure;
call sport_configure;

// Set up circular buffers

// **** INSERT CODE HERE ****
P0=length(filter)*4; // [INSERTED] Pointer register P0 holds the length of the filter in bytes
I0=0x11800000;B0=I0;L0=P0; // [INSERTED] Circular buffer initialised for input data (Base
register, Index register, Length register & Pointer register)
I1=0x11900000;B1=I1;L1=P0; // [INSERTED] Circular buffer initialised for filter coefficients
(Base register, Index register, Length register & Pointer register)
P0=length(filter)-1; // [INSERTED] Pointer register P0 is now intialised to hold the number of
filter coeficients as defined in line 12 (i.e. k-1 iterations)

get_audio:
wait_left:
// Wait for left data then read
R0=[REG_SPORT0_CTL_B];
CC=BITTST(R0, 31);
if !CC jump wait_left;
R0=[REG_SPORT0_RXPRI_B];

// Convolution kernel


// **** INSERT CODE HERE ****
A0 = 0; // [INSERTED] First part of 72-bit accumulator register (A0) initialised to 0 (cleared)
A1 = 0 || [I0++]= R0|| R1 = [I1++]; // [INSERTED] Rest of 72-bit acculumator register (A1)
initialised to 0 (cleared), Input audio data in R0 fed into circular buffer I0 initialised
before (32-bit), Filter coeficients in index register I0 are fed into R1 (32-bit) then post-
increments both for the next instruction. (exploits parallelism)
LOOP LC0=P0; // [INSERTED] Start of convolution loop containing loop counter register LC0,
controlled by condition in pointer register P0 (i.e. filter length-1 iterations)
A1:0 += R0 * R1|| R0 = [I0++] || R1 = [I1++]; // [INSERTED] Filter left by multiplying current
input audio data in R0 by current filter coeficients in R1 via a 32-bit multiply into the 72-
bit accumlator register A1:0. Then increments both I0 and I1 for the next multiplication. This
shifted multiply accumulate (MAC) is the heart of convolution. (exploits parallelism)
LOOP_END; // [INSERTED] End of convolution loop
A1:0 += R0 * R1||I0-=4; // [INSERTED] 32-bit multiplies the final values of input audio data in
R0 and filter coeficients in R1 into 72-bit accumulator register A1:0. In addition, post-
```

```
decrements the I0 register by two bites (i.e. 32-bit word) in order to ensure that the next
value from the codec overwrites the oldest value, not the previously most recently acquired.
(exploits parallelism)

//Write left out
R1=A1:0; [REG_SPORT0_TXPRI_A]=R1;
wait_right:
// Wait for left data then dummy read
R0=[REG_SPORT0_CTL_B];
CC=BITTST(R0, 31);
if !CC jump wait_right;
R0=[REG_SPORT0_RXPRI_B];
// Copy from left to write
R0=A1:0;
// Write right out
[REG_SPORT0_TXPRI_A]=R0;
jump get_audio;
rts;
._main.end:

// Function codec_configure initialises the ADAU1761 codec. Refer to the control register
// descriptions, page 51 onwards of the ADAU1761 data sheet.
codec_configure:
[--SP] = RETS;                              // Push stack (only for nested calls)
R1=0x01(X); R0=0x4000(X); call TWI_write; // Enable master clock, disable PLL
R1=0x7f(X); R0=0x40f9(X); call TWI_write; // Enable all clocks
R1=0x03(X); R0=0x40fa(X); call TWI_write; // Enable all clocks
R1=0x01(X); R0=0x4015(X); call TWI_write; // Set serial port master mode
R1=0x13(X); R0=0x4019(X); call TWI_write; // Set ADC to on, both channels
R1=0x21(X); R0=0x401c(X); call TWI_write; // Enable left channel mixer
R1=0x41(X); R0=0x401e(X); call TWI_write; // Enable right channel mixer
R1=0x03(X); R0=0x4029(X); call TWI_write; // Turn on power, both channels
R1=0x03(X); R0=0x402a(X); call TWI_write; // Set both DACs on
R1=0x01(X); R0=0x40f2(X); call TWI_write; // DAC gets L, R input from serial port
R1=0x01(X); R0=0x40f3(X); call TWI_write; // ADC sends L, R input to serial port
R1=0x0b(X); R0=0x400a(X); call TWI_write; // Set left line-in gain to 0 dB
R1=0x0b(X); R0=0x400c(X); call TWI_write; // Set right line-in gain to 0 dB
R1=0xe7(X); R0=0x4023(X); call TWI_write; // Set left headphone volume to 0 dB
R1=0xe7(X); R0=0x4024(X); call TWI_write; // Set right headphone volume to 0 dB
R1=0x00(X); R0=0x4017(X); call TWI_write; // Set codec default sample rate, 48 kHz
NOP;
RETS = [SP++];                              // Pop stack (only for nested calls)
RTS;
codec_configure.end:

// Function sport_configure initialises the SPORT0. Refer to pages 26-59, 26-67,
// 26-75 and 26-76 of the ADSP-BF70x Blackfin+ Processor Hardware Reference manual.
sport_configure:
R0=0x3F0(X); [REG_PORTC_FER]=R0;          // Set up Port C in peripheral mode
R0=0x3F0(X); [REG_PORTC_FER_SET]=R0;      // Set up Port C in peripheral mode
R0=0x2001973; [REG_SPORT0_CTL_A]=R0;      // Set up SPORT0 (A) as TX to codec, 24 bits
R0=0x0400001; [REG_SPORT0_DIV_A]=R0;      // 64 bits per frame, clock divisor of 1
R0=0x1973(X); [REG_SPORT0_CTL_B]=R0;      // Set up SPORT0 (B) as RX from codec, 24 bits
R0=0x0400001; [REG_SPORT0_DIV_B]=R0;      // 64 bits per frame, clock divisor of 1
RTS;
sport_configure.end:

// Function TWI_write is a simple driver for the TWI. Refer to page 24-15 onwards
// of the ADSP-BF70x Blackfin+ Processor Hardware Reference manual.
TWI_write:
R3=R0 <<0x8; R0=R0 >>>0x8; R2=R3|R0;      // Reverse low order and high order bytes
R0=0x3232(X); [REG_TWI0_CLKDIV]=R0;       // Set duty cycle
R0=0x008c(X); [REG_TWI0_CTL]=R0;          // Set prescale and enable TWI
R0=0x0038(X); [REG_TWI0_MSTRADDR]=R0;     // Address of codec
[REG_TWI0_TXDATA16]=R2;                   // Address of register to set, LSB then MSB
```

```
R0=0x00c1(X); [REG_TWI0_MSTRCTL]=R0;      // Command to send three bytes and enable tx
[--SP] = RETS; call delay; RETS = [SP++]; // Delay
[REG_TWI0_TXDATA8]=R1;                    // Data to write
[--SP] = RETS; call delay; RETS = [SP++]; // Delay
R0=0x050; [REG_TWI0_ISTAT]=R0;            // Clear TXERV interrupt
[--SP] = RETS; call delay; RETS = [SP++]; // Delay
R0=0x010; [REG_TWI0_ISTAT]=R0;            // Clear MCOMP interrupt
rts;
TWI_write.end:

// Function delay introduces a delay to allow TWI communication
delay:
P0=0x8000;
loop LC0=P0;
NOP; NOP; NOP;
loop_end;
RTS;
delay.end:
```

Exercise 6 - Sinegen

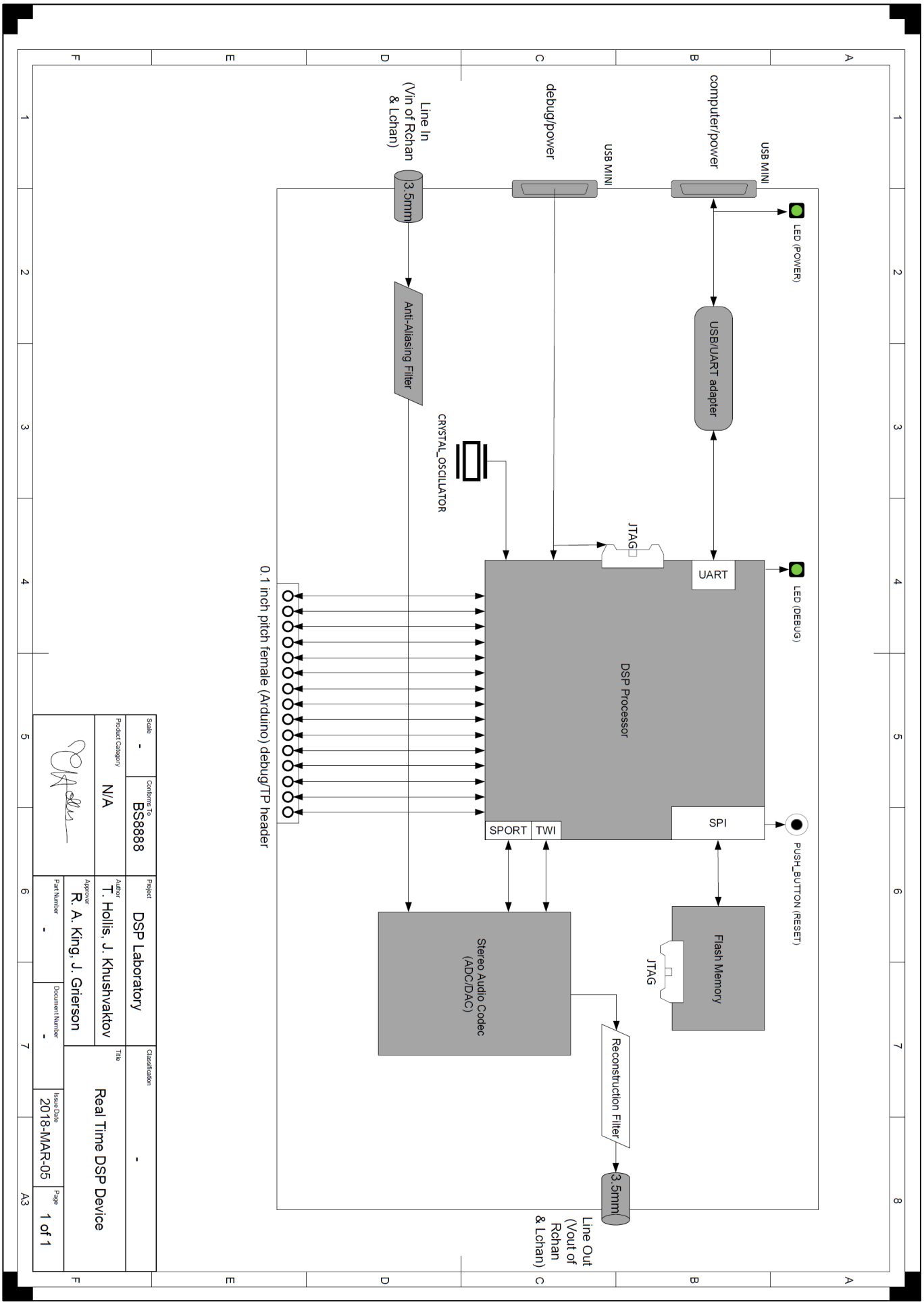
```
/*
Program sinegen is a simple dual channel sine wave generator, which uses a look-up table for
the coefficients.
In the example below, it produces frequencies of 6 kHz and 12 kHz.

Author: Patrick Gaydecki
Date  : 02.10.2017
*/

.section L1_data_a;  // Linker places 6 kHz LUT starting at 0x11800000

// **** INSERT CODE HERE ****
.BYTE4/r32 sin6[]=0.0r,0.7071068r,1.0r,0.7071068r,0.0r,-0.7071068r,-1.0r,-0.7071068r; //
[INSERTED] Sets the 8 values required (since sampling frequency is 48kHz, 48/8=6kHz) for a 6kHz
sine wave (used Signal Wizard). Uses interpolation.

.section L1_data_b;  // Linker places 12 kHz LUT starting at 0x11900000

// **** INSERT CODE HERE ****
.BYTE4/r32 sin12[]=0.0r,1.0r,0.0r,-1.0r,0.0r,1.0r,0.0r,-1.0r; // [INSERTED] Sets the 4 values
required (since sampling frequency is 48kHz, 48/4=12kHz) for a 12kHz sine wave (used Signal
Wizard). Uses interpolation. Repeated to two periods in order to have the same length as 6kHz
sine for later sections.

.section program;
.global _main;
.align 4;
# include <defBF706.h>

_main:
call codec_configure;
call sport_configure;

// Set modulo addressing for each channel

// **** INSERT CODE HERE ****
P0=length(sin6)*4; // [INSERTED] Pointer register P0 holds the length of the 6kHz sine data
(line 12) in bytes which is the same as the extended 12kHz sine data (line 17).
I0 = 0x11800000; B0 = I0; L0 = P0; // [INSERTED] Circular buffer initialised for 6kHz sine data
(Base register, Index register, Length register & Pointer register)
I1 = 0x11900000; B1 = I1; L1 = P0; // [INSERTED] Circular buffer initialised for 12kHz sine
data (Base register, Index register, Length register & Pointer register)

get_audio:
wait_left:
// Wait for left data then dummy read
R0=[REG_SPORT0_CTL_B];
CC=BITTST(R0, 31);
if !CC jump wait_left;
R0=[REG_SPORT0_RXPRI_B];
// Load 6 kHz data from LUT and write to codec
// Note 8-bit shift right since codec is 24-bit

// **** INSERT CODE HERE ****
R0 = [I0++]; // [INSERTED] Writes the 6kHz sine data for the left channel to the R0 register
and post-increments I0
R0 >>= 0x08; // [INSERTED] Brings the data (too big) into the codec's range


[REG_SPORT0_TXPRI_A]=R0;
wait_right:
// Wait for right data then dummy read
R0=[REG_SPORT0_CTL_B];
```

```
CC=BITTST(R0, 31);
if !CC jump wait_right;
R0=[REG_SPORT0_RXPRI_B];
// Load 12 kHz data from LUT and write to codec
// Note 8-bit shift right since codec is 24-bit

// **** INSERT CODE HERE ****
R1 = [I1++]; // [INSERTED] Writes the 12kHz sine data for the right channel to the R0 register
and post-increments I1
R1 >>= 0x08; // [INSERTED] Brings the data (too big) into the codec's range


[REG_SPORT0_TXPRI_A]=R1;
jump get_audio;
._main.end:

// Function codec_configure initialises the ADAU1761 codec. Refer to the control register
// descriptions, page 51 onwards of the ADAU1761 data sheet.
codec_configure:
[--SP] = RETS;                          // Push stack (only for nested calls)
R1=0x01(X); R0=0x4000(X); call TWI_write; // Enable master clock, disable PLL
R1=0x7f(X); R0=0x40f9(X); call TWI_write; // Enable all clocks
R1=0x03(X); R0=0x40fa(X); call TWI_write; // Enable all clocks
R1=0x01(X); R0=0x4015(X); call TWI_write; // Set serial port master mode
R1=0x13(X); R0=0x4019(X); call TWI_write; // Set ADC to on, both channels
R1=0x21(X); R0=0x401c(X); call TWI_write; // Enable left channel mixer
R1=0x41(X); R0=0x401e(X); call TWI_write; // Enable right channel mixer
R1=0x03(X); R0=0x4029(X); call TWI_write; // Turn on power, both channels
R1=0x03(X); R0=0x402a(X); call TWI_write; // Set both DACs on
R1=0x01(X); R0=0x40f2(X); call TWI_write; // DAC gets L, R input from serial port
R1=0x01(X); R0=0x40f3(X); call TWI_write; // ADC sends L, R input to serial port
R1=0x0b(X); R0=0x400a(X); call TWI_write; // Set left line-in gain to 0 dB
R1=0x0b(X); R0=0x400c(X); call TWI_write; // Set right line-in gain to 0 dB
R1=0xe7(X); R0=0x4023(X); call TWI_write; // Set left headphone volume to 0 dB
R1=0xe7(X); R0=0x4024(X); call TWI_write; // Set right headphone volume to 0 dB
R1=0x00(X); R0=0x4017(X); call TWI_write; // Set codec default sample rate, 48 kHz
NOP;
RETS = [SP++];                          // Pop stack (only for nested calls)
RTS;
codec_configure.end:

// Function sport_configure initialises the SPORT0. Refer to pages 26-59, 26-67,
// 26-75 and 26-76 of the ADSP-BF70x Blackfin+ Processor Hardware Reference manual.
sport_configure:
R0=0x3F0(X); [REG_PORTC_FER]=R0;        // Set up Port C in peripheral mode
R0=0x3F0(X); [REG_PORTC_FER_SET]=R0;    // Set up Port C in peripheral mode
R0=0x2001973; [REG_SPORT0_CTL_A]=R0;    // Set up SPORT0 (A) as TX to codec, 24 bits
R0=0x0400001; [REG_SPORT0_DIV_A]=R0;    // 64 bits per frame, clock divisor of 1
R0=0x1973(X); [REG_SPORT0_CTL_B]=R0;    // Set up SPORT0 (B) as RX from codec, 24 bits
R0=0x0400001; [REG_SPORT0_DIV_B]=R0;    // 64 bits per frame, clock divisor of 1
RTS;
sport_configure.end:

// Function TWI_write is a simple driver for the TWI. Refer to page 24-15 onwards
// of the ADSP-BF70x Blackfin+ Processor Hardware Reference manual.
TWI_write:
R3=R0 <<0x8; R0=R0 >>>0x8; R2=R3|R0;    // Reverse low order and high order bytes
R0=0x3232(X); [REG_TWI0_CLKDIV]=R0;     // Set duty cycle
R0=0x008c(X); [REG_TWI0_CTL]=R0;        // Set prescale and enable TWI
R0=0x0038(X); [REG_TWI0_MSTRADDR]=R0;   // Address of codec
[REG_TWI0_TXDATA16]=R2;                 // Address of register to set, LSB then MSB
R0=0x00c1(X); [REG_TWI0_MSTRCTL]=R0;    // Command to send three bytes and enable tx
[--SP] = RETS; call delay; RETS = [SP++]; // Delay
[REG_TWI0_TXDATA8]=R1;                  // Data to write
[--SP] = RETS; call delay; RETS = [SP++]; // Delay
```

```
R0=0x050; [REG_TWI0_ISTAT]=R0;            // Clear TXERV interrupt
[--SP] = RETS; call delay; RETS = [SP++]; // Delay
R0=0x010; [REG_TWI0_ISTAT]=R0;            // Clear MCOMP interrupt
rts;
TWI_write.end:

// Function delay introduces a delay to allow TWI communication
delay:
P0=0x8000;
loop LC0=P0;
NOP; NOP; NOP;
loop_end;
RTS;
delay.end:
```

Exercise 7

1. Block Diagram & Logical connections



2. Components

There are 8 main components in our custom designed Real Time DSP device.

- **DSP Processor**

  An important criterion is that this processor is low power. This is because the lower the processor power, the lower the power of peripherals thus the lower the overall power levels on the tracks and through the components in the board. Hence, low power in the board and components means you can have a smaller separation distance in semiconductors (i.e. tracks in the PCB closer together) and a much smaller overall device. If this DSP device is to be mounted, on an instrument for example, it should be as non-invasive as possible. In addition, if ran off a battery it should be made to last as long as possible. Finally, low power means lower interference noise in circuits.

  Another important criterion is word length. The DSP processor used for this design is based on a 16/32-bit fixed point architecture hence should support the processing of 8, 16 or 32-bit words. This is in order to maintain a reasonable level of quantisation error (since the lower the word length, the worse the quantisation error).

  In terms of memory organisation, a Harvard or super Harvard architecture is preferred as it allows far more pipelining optimisations. This is because data and code are stored in different sections so can be access simultaneously. In addition, the DSP processor should have an ultrafast L1 cache with a storage capacity around 136K Bytes. This offers a very high speed but at a compromised storage capacity. Hence a slightly slower but larger L2 cache should also be used with a capacity of around 1M Byte (with a 512K Byte ROM, and 512K Byte SRAM). This memory should be error-correcting code (ECC) memory since cosmic rays (amongst others) have been found to occasionally flip bits in memory which could cause significant upset, especially for real time systems like this. The boot ROM is also required to store the boot steam (user software or OS) produced by the boot loader. In addition, 4 kB of one-time-programmable (OTP) memory would be useful to store a unique identification code for secure operation. CRC-Protection can also be implemented on all these memories to allow for CRC checksum comparisons to be calculated on the fly during memory transfers. A CRC engine would also protect data being loaded during the boot process.

  The peripheral support needed is SPI (for flash memory), UART (for UART to USB power/computer), JTAG (for debugging), SPORT and I²C a.k.a TWI (for ADC/DAC audio codec). Details of why these peripherals have been chosen are explained in each of the respective devices' section bellow.

  A useful bonus feature would be for the processor to support encryption and decryption (secure boot) to protect any intellectual property (IP) loaded to it. This would allow the original owners not to get market share taken as a consequence of IP theft and resale by cheaper manufacturers.

  Overall this processor should be expected to perform at around 800 MMACS.

  From all the above requirements, an excellent device to use would be the ADSP-BF706 Blackfin Processor.

- **Audio Codec (ADC/DAC)**

  Here a low power device is chosen for the same reasons as presented above. A 24-bit stereo codec is used due to its suitability for audio data. The communication protocol used to interface with the DSP Processor chip is through TWI and SPORT interfaces. This is because TWI allows the efficient transmission of control information in a simple way and SPORT allows a performant transmission of audio data. SPORT is particularly suitable for audio data as it allows memory to be transferred through dedicated DMA channels with each serial port working in conjunction to provide TDM support. More information regarding this interface is shown in the timing diagram in section 3 (Timing Diagrams). These requirements are best met by the low power ADAU1761 SigmaDSP device.

- **Flash Memory**

  A 4 MB external flash memory should be more than enough capacity for this audio application. In addition, quad SPI is a reasonable interface to the DSP processor for easy and simple communications protocol at sufficiently high

speed. This can be justified since DSP devices have rather restricted address spaces since the programs operate in real-time so are by definition quite small in size.

- **UART USB Adapter**

  In order to allow the programming of the DSP device, a connection to a computer is paramount. Because of the universality of the universal serial bus (USB) protocol, a USB to UART adapter must be used to connect the DSP's input UART to the output USB of the computer. UART is chosen as this allows full-duplex, DMA-based, asynchronous transfers of serial data. This is henceforth used to send the code written in software by the user to the device. Further details regarding this interface is shown in section 4 (Firmware Based OS).

- **Crystal Oscillator Clock Module**

  A crystal oscillator clock module was chosen to externally clock the DSP processor. This is because this external clock is a TTL compatible signal which cannot be halted, changed, or operated below the specified frequency during normal operation. This allows for optimal reliability during operation. The external crystal oscillator will have to be relatively low frequency to prevent RFI and to avoid overly complicated PCB designs. Hence, a base clock of around 50 MHz would be ideal. This would allow an 8x clock multiplier to therefore be used to derive all the required clock signals from the base crystal oscillator. Hence the maximum clock speed would be 400 MHz and should be used by all core ALU operations but not necessarily by all other systems.

- **Reconstruction Filter**

  A 2nd order anti-aliasing output filter is commonplace in codec signal sanitisation. This is done to prevent imaging, which is the reverse process of aliasing where in-band frequencies are mirrored out of the band. This is essentially a trade off between bandwidth and aliasing and should approach but not exceed the Nyquist limit.

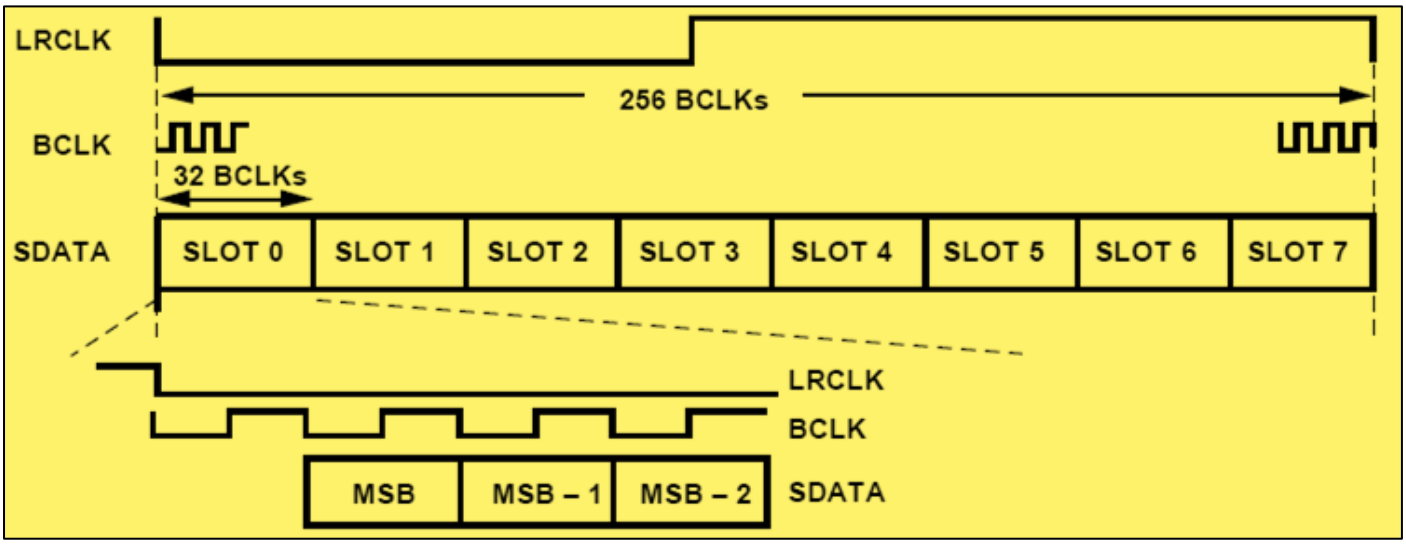
- **Anti-Aliasing Filter**

  A 1st order anti-aliasing input filter is commonplace in codec signal sanitisation. This is to restrict the bandwidth to satisfy the sampling theorem over the relevant frequency band.

- **Extra: status LEDs (on), Push Button (reset & interrupt request switches), external connectors, JTAG**

  Here, we can have ultra-low power LEDs to indicate visually power and debug status without requiring large high-power PCB tracks. In addition, a few push-button reset and interrupt request switches are always useful. Some external connectors were also added to allow connection to an Arduino or any device that supports 0.1-inch pitch header connections (extremely common). JTAG connections were also placed anywhere that could be of use at the manual test and hardware debugging stage.

3. Timing diagram



Here the timing diagram shows the two interfaces that are in use to control the codec and send/receive audio data. TWI communicates control information (sample rate, volume, mode…) while the SPORT transmits the digitized serial audio data. This is all synchronised by the Bit Clock (BCLK) and the Left Right Clock (LRCLK) a.k.a frame clock. When LRCLK transitions from high to low, the DSP device is told that this is the start of new data (as illustrated above). The bit clock (BCLK) is then used to store all the required data into slots via the SDATA signal.

4. General information of firmware-based OS

The OS of the device is to be programmed by the user using a computer. A common Eclipse based IDE such as CrossCore could be used as this is platform independent (macOS, Windows, Linux & Solaris) and universal. The assembly language used to program the device should try to follow the style of high-level languages to ease the programming effort required by end-users. The processor instruction set should be optimised for a 16-bit structure to facilitate efficient and compact code. Complex instructions can be encoded as 32-bit opcodes which will incur a processing overhead but will allow for an algebraic-like syntax.

Once the code has been written, it should be capable of live debugging through the USB/UART connection, while the device is still directly connected to the computer. This can be implemented by having a system watchpoint unit (SWU) which is a single module which connects to a single system bus and allows transaction monitoring. One SWU should be attached for each slave to prove ports for all system bus address channel signals. This would allow an independent operation with common interrupt, trigger and other event outputs. In addition, having a debug access power (DAP) allows the support of JTAG debugging which is extremely useful to diagnose hardware issues.

Finally, the firmware-based OS should be stored in non-volatile memory to allow the device to run in standalone mode without being connected to a computer.