ETISB
Embedded Signal Processing

Indhold

Introduction, number formats and Blackfin

1.1 Lektion 30-01-2018

1. Course introduction
2. Typical embedded system
3. Number formats (fixed- and floating-point)
4. Conversion between different number formats
5. (Blackfin) DSP Architecture

- ESP Chapter 1.1 + 1.2
- ESP 5.1 + 5.2.1
- ESP Chapter 6.1.1 (only p.217-p.222) and 6.1.3 - 6.1.5

1.1.1 Typical embedded system

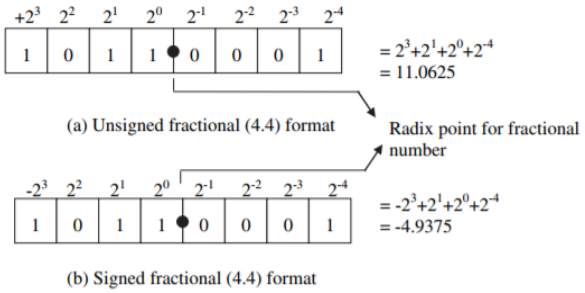
- **Dedicated functions:** Embedded systems usaully execute a specific task repeatedly.
- **Tight constraints:** There are many constraints in designing an embedded system, such as; cost, processing speed, size, power consuption.
- **Reactive and real-time performance:** Many embedded systems must continously react to changes of the system's input.

1.1.2 Number formats (fixed- and floating-point)

- Fixedpoint
- Floatingpoint
- Blockfloatingpoint

Fixed-point

- Binary data format - signed and unsigned
  - The 2's complement format is the most popular signed number in DSP processors.
  - Most DSP processors support both integer and fractional data formats.
    - In an integer format, the radix point is located to the right of the least significant bit (LSB).
    - In a fractional number format, the radix point is located within the binary number.
      - The number to the right of the radix point assumes a fractional binary bit, with a weighting of 2^{-p}.
      - For the number to the left of the radix point, the weighting increases from 2^q. The weighting of the MSB (or sign bit) depends on whether the number is signed or unsigned.
    - (N.M) notation
      - N is the number of bits to the left of the radix point (integer part).
      - M is the number of bits to the right of the radix point (fractional part).
      - The symbol "." represents the radix point.
      - Total number of bits in the data word is B = N + M.



Figur 1.1: Example of 8-bit binary data formats for a fractional number.

- Dynamic Ranges and Precisions
  - The maximum positive number in (1.15) format is 12^{15}(= 0.999969482421875) (0x7FFF).
  - The minimum negative number in (1.15) format is 1 (0x8000).
  - The 1.15 format has a **dynamic range** of [+0.999969482421875 to 1]
    - Numbers exceeding this range cannot be represented in 1.15 format.
  - The smallest increment (or precision) within the (1.15) format is 2^{15}.

Format (*N.M*)	Largest Positive Value (0x7FFF)	Least Negative Value (0x8000)	Precision (0x0001)
(1.15)	0.999969482421875	–1	0.00003051757813
(2.14)	1.99993896484375	–2	0.00006103515625
(3.13)	3.9998779296875	–4	0.00012207031250
(4.12)	7.999755859375	–8	0.00024414062500
(5.11)	15.99951171875	–16	0.00048828125000
(6.10)	31.9990234375	–32	0.00097656250000
(7.9)	63.998046875	–64	0.00195312500000
(8.8)	127.99609375	–128	0.00390625000000
(9.7)	255.9921875	–256	0.00781250000000
(10.6)	511.984375	–512	0.01562500000000
(11.5)	1,023.96875	–1,024	0.03125000000000
(12.4)	2,047.9375	–2,048	0.06250000000000
(13.3)	4,095.875	–4,096	0.12500000000000
(14.2)	8,191.75	–8,192	0.25000000000000
(15.1)	16,383.5	–16,384	0.50000000000000
(16.0)	32,767	–32,768	1.00000000000000

Figur 1.2: Dynamic ranges and precisions of 16-bit numbers using different formats.

- Scaling Factors
  - A number in (N.M) format cannot be represented in the programs because most compilers and assemblers only recognize numbers in integer or (16.0) format.
    * Convert the fractional number in (N.M) format into its integer equivalent.
    * Its radix point must be accounted for by the programmers.
      · To convert a number 0.6 in (1.15) format to its integer representation, multiply it by 2^{15} (or 32 768) and round the product to its nearest integer to become 19 661 (0x4CCD).

In Table 1.2 all 16 possible (N.M) formats for 16-bit numbers. Different formats give different dynamic ranges and precisions. There is a trade-off between the dynamic range and precision.

- As the dynamic range increases, precision becomes coarser.

Format	Scaling Factor (2^M)	Range in Hex (fractional value)
(1.15)	2^{15} = 32,768	0x7FFF (0.99) → 0x8000 (–1)
(2.14)	2^{14} = 16,384	0x7FFF (1.99) → 0x8000 (–2)
(3.13)	2^{13} = 8,192	0x7FFF (3.99) → 0x8000 (–4)
(4.12)	2^{12} = 4,096	0x7FFF (7.99) → 0x8000 (–8)
(5.11)	2^{11} = 2,048	0x7FFF (15.99) → 0x8000 (–16)
(6.10)	2^{10} = 1,024	0x7FFF (31.99) → 0x8000 (–32)
(7.9)	2^9 = 512	0x7FFF (63.99) → 0x8000 (–64)
(8.8)	2^8 = 256	0x7FFF (127.99) → 0x8000 (–128)
(9.7)	2^7 = 128	0x7FFF (511.99) → 0x8000 (–512)
(10.6)	2^6 = 64	0x7FFF (1,023.99) → 0x8000 (–1,024)
(11.5)	2^5 = 32	0x7FFF (2,047.99) → 0x8000 (–2,048)
(12.4)	2^4 = 16	0x7FFF (4,095.99) → 0x8000 (–4,096)
(13.3)	2^3 = 8	0x7FFF (4,095.99) → 0x8000 (–4,096)
(14.2)	2^2 = 4	0x7FFF (8,191.99) → 0x8000 (–8,192)
(15.1)	2^1 = 2	0x7FFF (16,383.99) → 0x8000 (–16,384)
(16.0)	2^0 = 1(integer)	0x7FFF (32,767) → 0x8000h (–32,768)

Figur 1.3: Scaling factors and dynamic ranges for 16-bit numbers using different formats.

1.1.3 Conversion between different number formats

Fixed-Point Data Types

The Blackfin C compiler supports eight scalar data types and two fractional data types.

Type	Number Representation
`char`	8-bit signed integer
`unsigned char`	8-bit unsigned integer
`short`	16-bit signed integer
`unsigned short`	16-bit unsigned integer
`int`	32-bit signed integer
`unsigned int`	32-bit unsigned integer
`long`	32-bit signed integer
`unsigned long`	32-bit unsigned integer
`fract16`	16-bit signed (1.15) fractional number
`fract32`	32-bit signed (1.31) fractional number

Figur 1.4: Fixed-Point Data Types.

1.1.4 (Blackfin) DSP Architecture

- Von Neumann architecture uses a single memory to hold both data and instructions.
- The Harvard architecture uses separate memories for data and instructions, providing higher speed.
- The Super Harvard Architecture improves upon Harvard design by adding an instruction cache and a dedicated I/O controller.

1.1.5 Choice of processor

- DSP
  - Harvard memory architecture (optimized for the operational needs of digital signal processing).
  - Handle real time processing, architecture is specially designed to fetch multiple data at the same time.
  - Can handle floating numbers directly in the data flow path.
  - Calculations are usually carried out by fixed point arithmetic process in order to speed them up.

- Microcontroller
- Generalpurpose CPU
  - Most general-purpose CPU can also execute DSP algorithms, but dedicated DSPs usually have better power efficiency.
- FPGA
- GPU

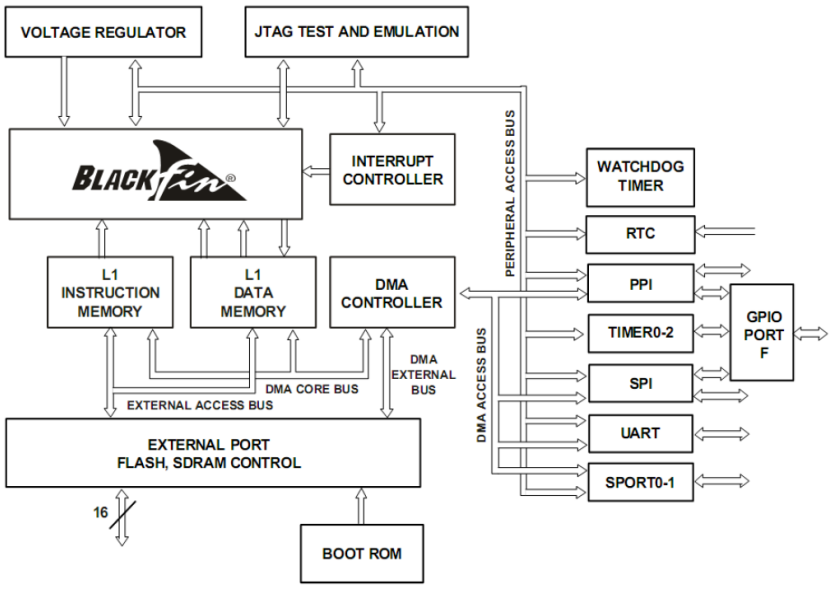
1.1.6 Blackfin

- Blackfin is a *MSA* Processor.
  - MCU and DSP in same processor.
    - Cheap and low power consumption.
- Architecture optimized for multimedia processing (Audio and Video).
- Dataflow oriented tasks (DSP).
- Control flow oriented tasks (MCU).
- Good tools and compiler support.

BF533 Overview

- 16-bit fixed-point processors that are based on the MSA core.
- The processor targets power-sensitive applications (portable audio players, cell phones, digital cameras).
- Direct memory access (DMA) controller transfers data between external devices/memories and the internal memories without processor intervention.
  - L1 cache memory for quick accessing of both data and instructions.
- BF533 system peripherals:
  - Parallel peripheral interface (PPI)
  - Serial peripheral interface (SPI)
  - Serial ports (SPORTs)

- General-purpose timers
- Universal asynchronous receiver transmitter (UART)
- Real-time clock (RTC)
- Watchdog timer
- General-purpose input/output (I/O) ports



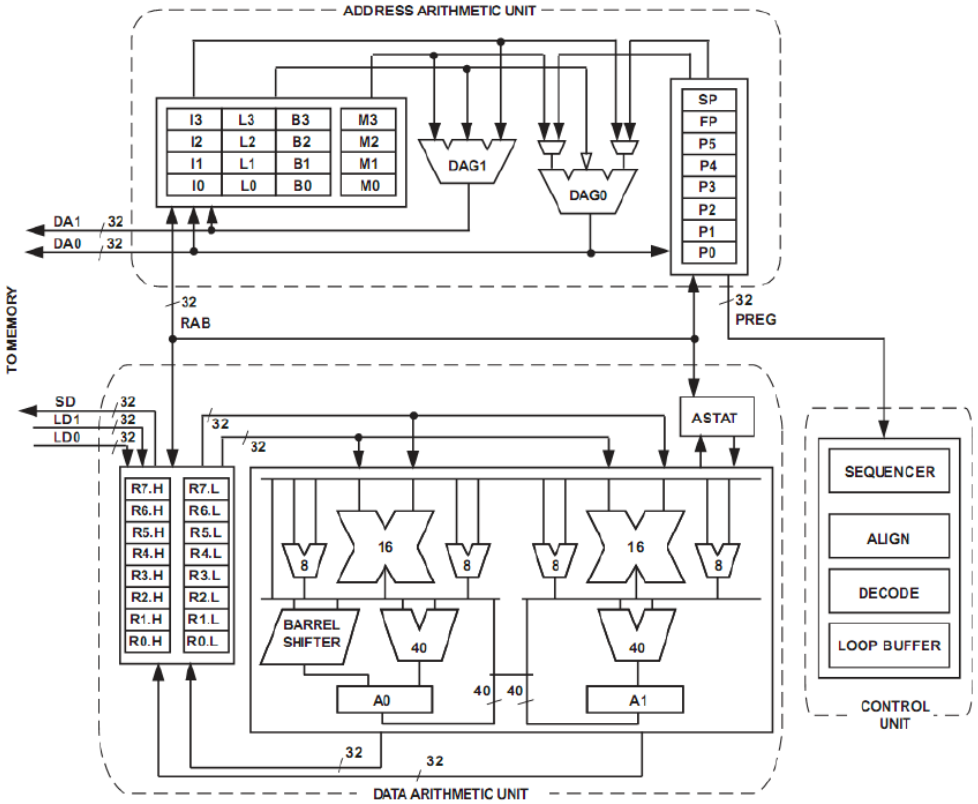
Figur 1.5: BF533 Overview.

Data Arithmetic Unit

Computational units get data from data registers and perform fixed-point operations. The data registers receive data from the data buses and transfer the data to the computational units for processing. Computational results are moved to the data registers before transferring to the memory via data buses.

- Two 16-bit multipliers represented as X_{16} in 1.6.
- Two 40-bit accumulators (ACC0 and ACC1).
  - 16-bit lower-half (A0.L, A1.L)
  - 16-bit upper-half (A0.H, A1.H)
  - 8-bit extension (A0.X, A1.X)
- Two 40-bit arithmetic logic units (ALUs) represented as V_{40} in 1.6.

- Four 8-bit video ALUs represented as V_8 in 1.6.
- A 40-bit barrel shifter.
- Eight 32-bit data registers (R0 to R7) or 16 independent 16-bit registers (R0.L to R7.L and R0.H to R7.H).



Figur 1.6: BF533 Core, Fixed-point processor.

Arithmetic Operations

- Dual and single versions of add and sub
  - Single 16-bit addition using three registers
    - `R3.H = R1.L+R2.H (ns);`
      - Saturation flag (s)
      - No saturation flag (ns)
      - End of instruction *;*
  - Dual 16-bit add/subtract using three registers
    - `R3 = R1+|-R2;`

- Quad 16-bit add/subtract using four registers
  - `R3 = R1+|-R2, R4 = R1-|+R2;`
    - Two operations can be operated on the same pair of 16-bit registers
    - Separate two instructions operated at same cycle *;*
- Single 32-Bit Operations
  - `R3 = R1+R2;`
- Dual 32-Bit Operations
  - `R3 = R1+R2, R4 = R1-R2`

Mode	Option	Example and Explanation
Dual and quad 16-bit operation (opt_mode_0)	S	Saturate the result at 16 bit `R3 = R1+|-R2 (s);`
	CO	Cross option that swaps the order of the results in the destination registers for use in complex math `R3 = R1+|-R2 (co);`
	SCO	Combination of S and CO options
Dual 32-bit and 40-bit operation (opt_mode_1)	S	Saturate result at 32 bit. `R3 = R1+R2, R4 = R1-R2 (s);`
Quad 16-bit operation (opt_mode_2)	ASR	Arithmetic shift right that halves the result before storing to the destination register `R3 = R1+|-R2, R4 = R1-|+R2 (s,asr);` Scaling is performed for the results before saturation.
	ASL	Arithmetic shift left that doubles the result before storing to the destination register

Figur 1.7: Arithmetic Modes and Options.

- Single 16-Bit Multiply Operations
  - `R3 = R1.L*R2.H;`
- Single 16-bit MAC using two registers and an accumulator
  - `A0 += R1.L*R2.L;`
    - Multiplies the contents of R1.L with R2.L, and the result is added to the value in the accumulator A0
- Dual 16-bit multiplications using two registers and two accumulators
  - `A1 = R1.H*R2.H, A0 = R1.L*R2.L;`
    - Two pairs of input are stored in registers R1 and R2

        - Two 32-bit results are stored in the accumulators A0 and A1
    - Dual 16-Bit Multiply Operations Using Two 32-Bit Registers
        * `R0 = R2.H*R3.H, R1 = R2.L*R3.L`
            - Dual 32-bit results are stored in R0 and R1
            - Destination registers must be used in pairs as R0:R1, R2:R3, R4:R5, or R6:R7

Option	Description
Default (no option)	Input data operand is signed fraction.
(FU)	Input data operands are unsigned fraction. No shift correction.
(IS)	Input data operands are signed integer. No shift correction.
(IU)	Input data operands are unsigned integer. No shift correction.
(T)	Input data operands are signed fraction. When copying to the destination half-register, truncates the lower 16 bits of the accumulator contents.
(TFU)	Input data operands are unsigned fraction. When copying to the destination half-register, truncates the lower 16 bits of the accumulator contents.

Figur 1.8: Multiplier Options.

Address Arithmetic Unit

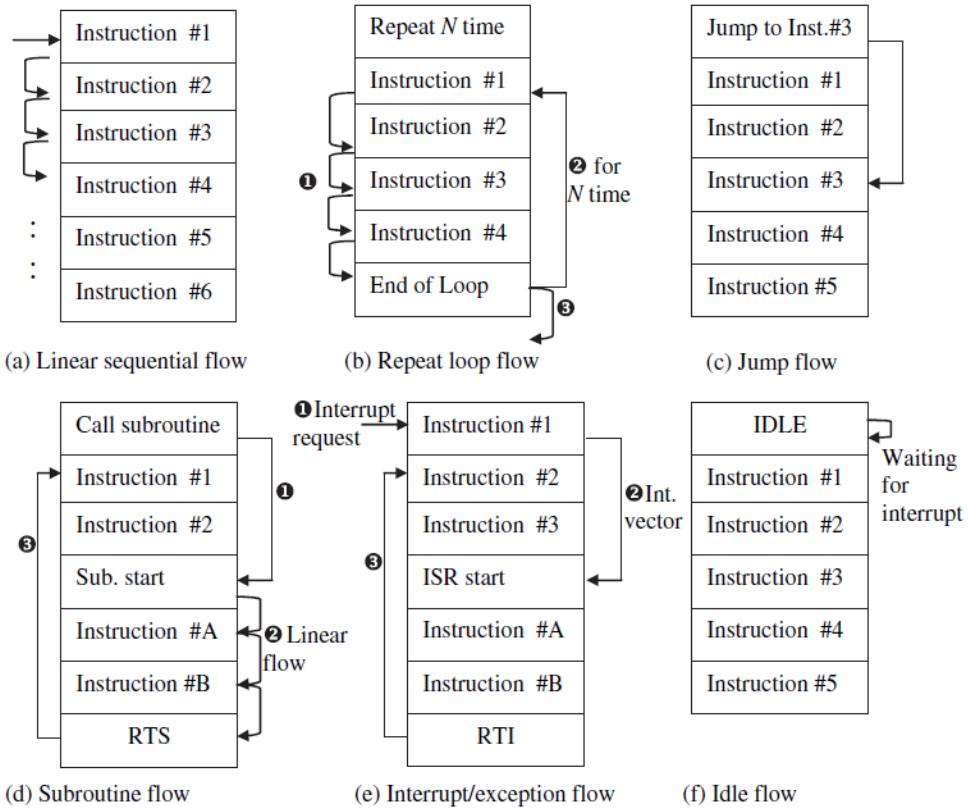
The address arithmetic unit consists of the following hardware units:

- Two data address generators (DAG0 and DAG1):
    - Generate addresses for data moves to and from memory. The advantage of using two data address generators is to allow dual-data fetches in a single instruction.
- Six 32-bit general-purpose address pointer registers (P0 to P5).
- One 32-bit frame pointer (FP) pointing to the current procedure's activation record.
- One 32-bit stack pointer (SP) pointing to the last location on the run time user stack.
- A set of 32-bit data address generator registers:
    - Indexing registers, I0 to I3, contain the effective addresses.

– Modifying registers, M0 to M3, contain offset values for add/subtract with the index registers.
– Base address registers, B0 to B3, contain the starting addresses of the circular buffers.
– Length value registers, L0 to L3, contain the lengths (in byte unit) of the circular buffers.

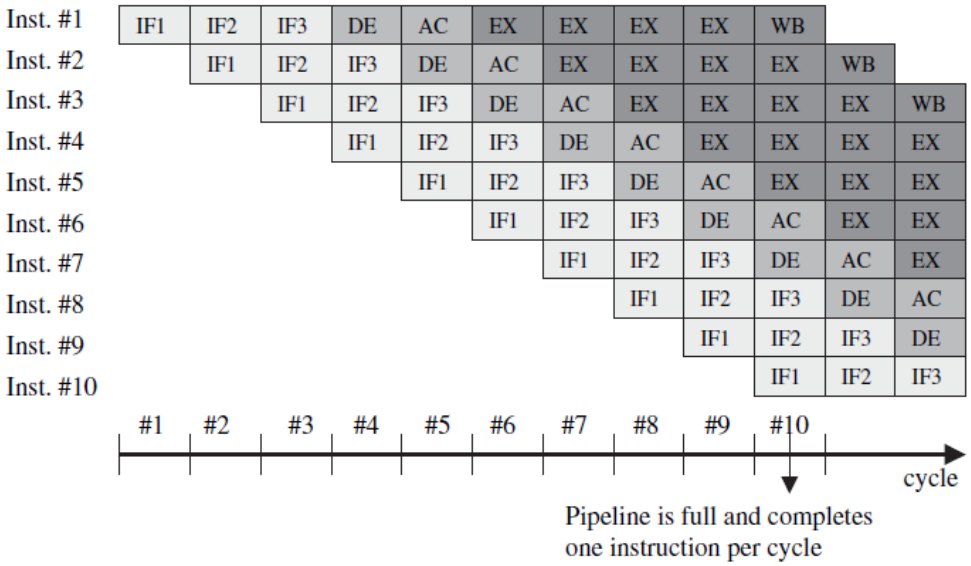
Control Unit

A program sequencer controls the instruction execution flow, which includes instruction alignment and instruction decoding. The address generated by the program sequencer is a 32-bit memory instruction address. A 32-bit program counter (PC) is used to indicate the current instruction being fetched.



Figur 1.9: Different types of program flow.

The pipeline is used to maximize the distribution of workload among the processor's functional units, which results in efficient parallel processing among the processor's hardware. The Blackfin processor has a 10-stage instruction pipeline.



Figur 1.10: Ten pipeline stages of the Blackfin processor.

Each instruction can be executed in a single clock cycle when the pipeline is full, giving a throughput of one instruction per clock cycle. However, any nonsequential program flow (Fig. 1.9) can potentially decrease the processor's throughput.

Pipeline Stage	Description
Instruction fetch 1 (IF1)	Start instruction memory access.
Instruction fetch 2 (IF2)	Intermediate memory pipeline.
Instruction fetch 3 (IF3)	Finish L1 instruction memory access.
Instruction decode (DEC)	Align instruction, start instruction decode, and access pointer register file.
Execute 1 (EX1)	Start access of data memory (program sequencer).
Execute 2 (EX2)	Register file read (data registers).
Execute 3 (EX3)	Finish access of data memory and start execution of dual-cycle instructions (multiplier and video unit).
Execute 4 (EX4)	Execute single-cycle instruction (ALU, shifter, accumulator).
Write back (WB)	Write states to data and pointer register files and process event.

Extracted from Blackfin Processor Hardware Reference [23].

Figur 1.11: Stages of Instruction Pipeline.

Quantization and fixed-point effects

2.1 Lektion 06-02-2018

1. ADC Quantization
2. Coefficient- and Product-Quantization
3. Overflow/Underflow and Coefficient Scaling
4. Notch- and Peak-filters as example

- ESP 6.1.1 (only p. 222 - 229)
- ESP 6.2.2, 6.2.3 (p. 240 - 243)
- ESP 3.4.2 + 3.4,3

2.1.1 ADC Quantization

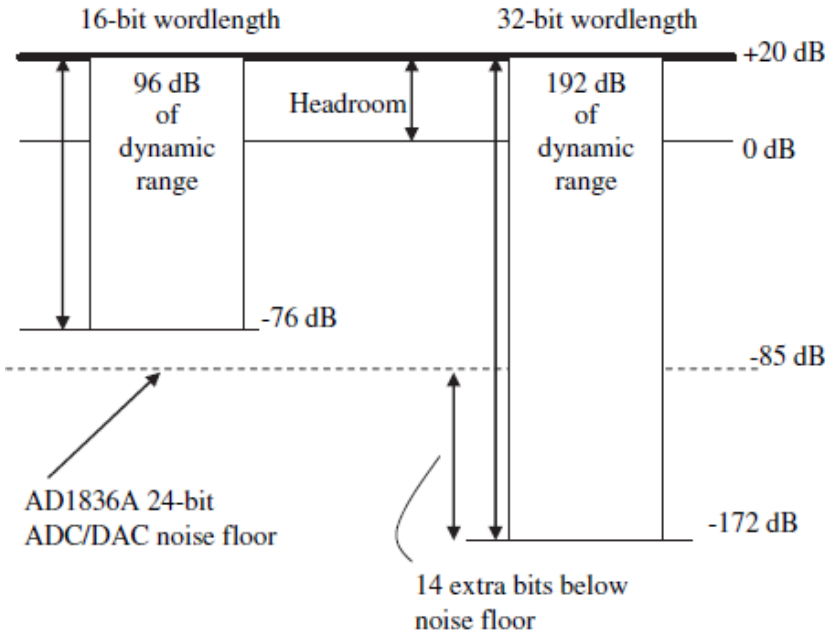
Dynamic Range and Signal-to-Quantization Noise Ratio

The dynamic range of a digital signal is defined as the difference between the largest and smallest signal values. If noise is present in the system, the dynamic range is the difference between the largest signal level and the noise floor. When performing signal processing, a higher precision (>16 bits) is normally preferred to maintain this level of dynamic range.

- Dynamic range
  - Typical in relation to "0 dB full scale"

$$dynamicrange = 20\log(\frac{max\ signal\ level}{noise\ floor})[\text{dB}] \tag{2.1}$$

- SQNR [dB] $\approx 6 \cdot n$ dB (n = bits precision)
  - Signal-to-quantization noise ratio = $20\log(2^n)$
  - Different wordlengths of ADC's
    - 16-bit fixed-point 96 dB
    - 24-bit fixed-point 144 dB
    - 32-bit fixed-point 192 dB
- Dynamic [dB] = Peak Level [dB] - Noise Floor [dB]



Figur 2.1: Comparison of different wordlenghts.

Other sources of quantization errors

Besides analog-to-digital and digital-to-analog quantization noise, there are other sources of quantization errors in digital systems.

- Coefficient quantization
- Computational overflow
- Quantization error due to truncation and rounding

2.1.2 Coefficient- and Product-Quantization

Coefficient Quantization

When a digital system is designed for computer simulation, the coefficients are generally represented with floating-point format. However, these parameters are usually represented with a finite number of bits in fixed-point processors with a typical wordlength of 16 bits.

Coefficient quantization of an IIR filter can affect pole/zero locations, thus altering the frequency response and even the stability of the digital filter.

- Only specific values possible for eg. filter coefficients
- Only discrete points in pole-zero plane
- Deterministic effect known in advance

Product Quantization

Truncation or rounding is used after multiplication to store the result back into the memory.

In general, the distribution of errors caused by rounding is uniform, resulting in quantization errors with zero mean and variance of $\Delta 2/12$. Truncation has a bias mean of $\Delta/2$ and a variance of $\Delta 2/12$.

- Happens after multiplication (=product) of both fixed- and floating-point numbers (unless using double number bits)
- Can be considered as an added noise term (stochastic)
- Limit-cycles may occur

2.1.3 Quantization and Noise models

Noise is introduced by multiplications between variables, a noise signal is added e(n).

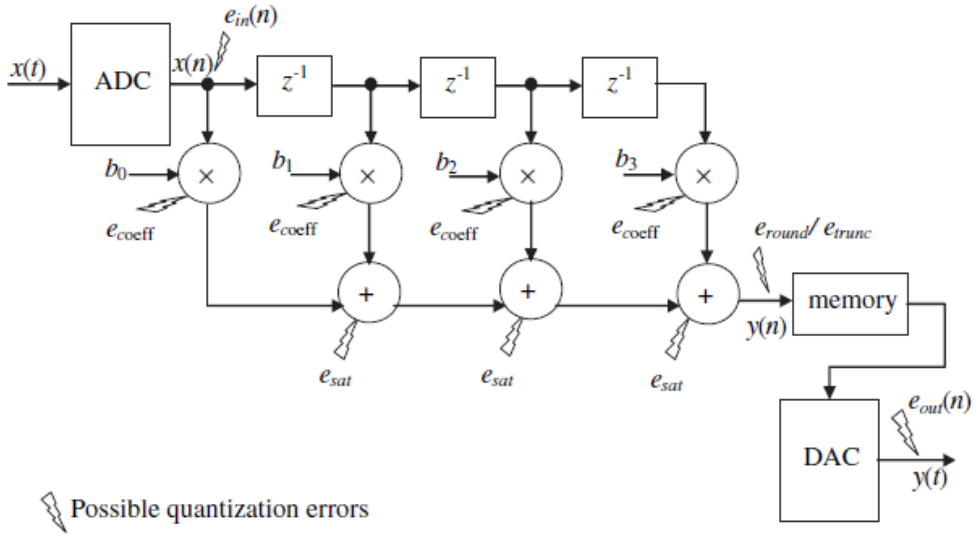
Signal Flow Graphs

- Illustration of implementation
- Useful for illustration of quantization errors

- Different filter-forms available most try to reduce effects of quantization for fixed-point implementations

Assumed that noise e is equal distributed

- Product quantization $0 \leq |e| \leq 2^{-(m+1)}$



Figur 2.2: Quantization error sources in a 4-tap FIR filter.

2.1.4 Overflow/Underflow and Coefficient Scaling

Because of the finite memory/register length, the results of arithmetic operations may have too many bits to be fitted into the wordlength of the memory or register.

- Causing "random"results or no output

A sum of M B-bit numbers requires $B + \log_2 M$ bits to represent the final result without overflow.

- eg. If 256 32-bit numbers are added, a total of $32 + \log_2 256 = 40$ bits are required to save the final result.

Overflow/underflow can be avoided with "clever"scaling or, at least, avoided "with a high probability"

- Block floating point
- Exponent checking
- Saturation
- Coefficient scaling

2.1.5 Notch- and Peak-filters

- Can be designed by pole-zero placement
- Pole radii and angle determine frequency response
- Common buildingblocks for higher-order IIR

Notch-filter

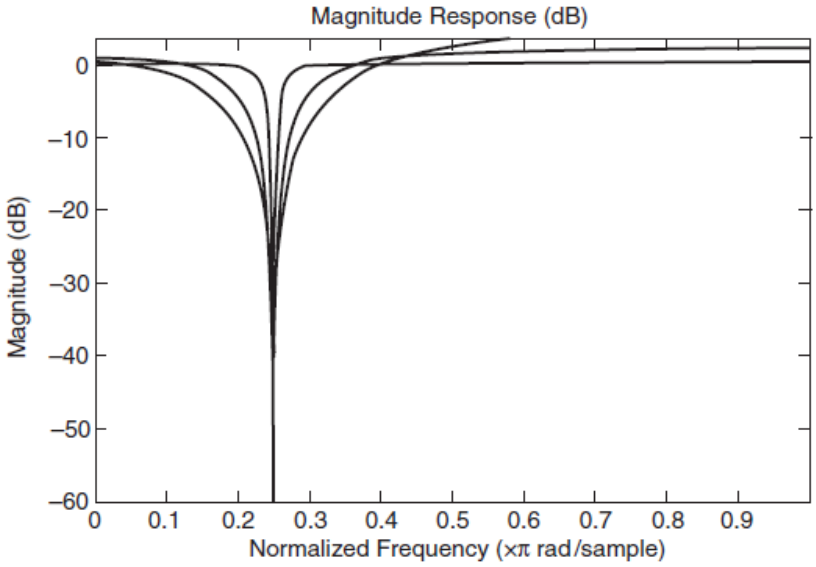
A notch filter contains one or more deep notches in its magnitude response. To create a notch at frequency ω_0, a pair of complex-conjugate zeros on the unit circle at angle ω_0 as $z = e^{\pm j\omega_0}$.

Transfer function of **FIR** notch filter is $H(z) = (1-e^{j\omega_0}z^{-1})(1-e^{-j\omega_0}z^{-1})$. This is a filter of order 2 because there are two zeros in the system. Note that a pair of complex-conjugate zeros guarantees that the filter will have real-valued coefficients.

The magnitude response has a relatively wide bandwidth, which means that other frequency components around the null are severely attenuated. To reduce the bandwidth of the null, we may introduce poles into the system $z_p = re^{\pm j\phi_0}$. r is the radius, ϕ_0 is the angle of poles.

Transfer function of **IIR** notch filter is $H(z) = \frac{(1-e^{j\omega_0}z^{-1})(1-e^{-j\omega_0}z^{-1})}{(1-r\cdot e^{j\phi_0}z^{-1})(1-r\cdot e^{-j\phi_0}z^{-1})}$.

This is a IIR filter of order 2 because there are two poles in the system.



Figur 2.3: Magnitude responses of notch filter for different values of r.

Peak-filter

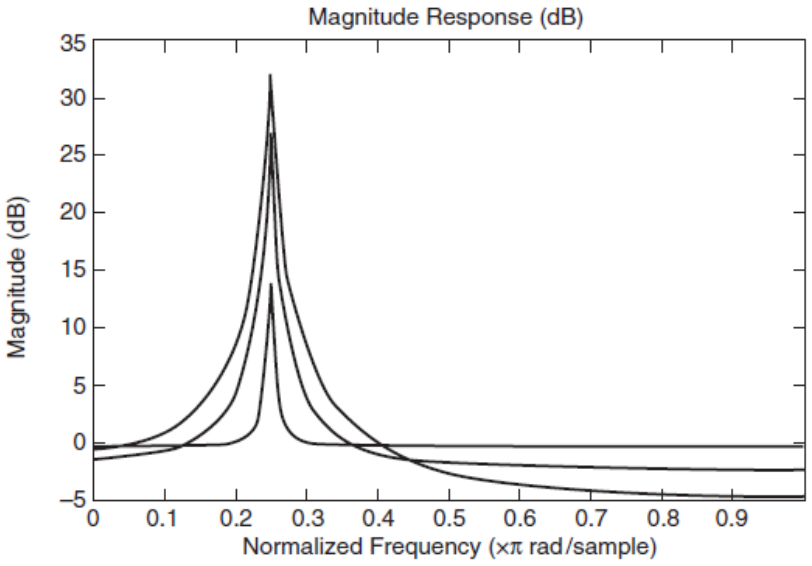
To create a peak (or narrow passband) at frequency ω_0, we may think we can simply follow the example of designing a notch filter by introducing a pair of complex-conjugate poles on the unit circle at angle ω_0. However the resulting second-order IIR filter will be unstable. To solve this problem, we have to move the poles slightly inside the unit circle ($r_p < 1$) as $z_p = r_p e^{\pm j\omega_0}$.

Transfer function of IIR peak filter is $H(z) = \frac{1}{1-r_p e^{j\omega_0} z^{-1})(1-r_p e^{-j\omega_0} z^{-1})}$.

Similar to the second-order IIR notch filter, the magnitude response has a relatively wide bandwidth, which means that other frequency components around the peak will also be amplified. To reduce the bandwidth of the peak, we may introduce zeros into the system.

Transfer function of **IIR** peak filter is $H(z) = \frac{1-r_z e^{j\omega_0} z^{-1})(1-r_z e^{-j\omega_0} z^{-1})}{1-r_p e^{j\omega_0} z^{-1})(1-r_p e^{-j\omega_0} z^{-1})}$.

The poles must be closer to the unit circle as $r_p > r_z$.



Figur 2.4: Magnitude responses of peak filter with poles ($r_p = 0.99$) and different radius of zeros.

The Blackfin Platform EzKit (BF533)

3.1 Lektion 20-02-2018

1. Interfaces to CODEC, SPORT and SPI
2. Sample and Block Processing
3. Memory and DMA

- ESP 5.1.4 (Memory)
- ESP 6.3 (Sample and block processing)
- ESP 7.1 (CODEC and SPORT)
- ESP 7.2 (DMA)

3.1.1 Interfaces to CODEC, SPORT and SPI

Transfer of data between the ADC and memory, within memory spaces, and between memory and peripherals with DMA.

CODEC

- A CODEC consists of both ADC and DAC with associated analog antialiasing and reconstruction low-pass filters.
  - 2 stereo ADC, 3 stereo DAC.
    - Operates in 16-, 18-, 20-, or 24-bit resolution.
  - TDM mode, 48-kHz sampling rate.
  - I2S mode, 96 kHz sampling rate, only primary channels active.

SPI

One SPI provides high-speed serial communication of up to SCLK/4. It interfaces with another processor, data converters, and display.

- CODEC setup using SPI interface.

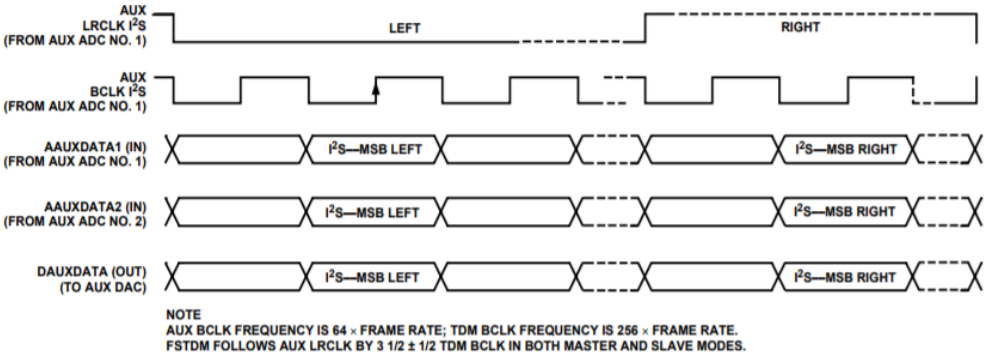
SPORT

Two synchronous serial ports (SPORT0 and SPORT1) provide high-speed serial communication with a maximum speed of SCLK/2. This provides an efficient interface with CODEC.

- Receive clock (Internal or External)
- Transmission synchronized with CODEC
- Buffers to handle Rx and Tx data
- I2S mode is standard for stereo channels (L/R)
- TDM mode (Time Division Multiplex)
- Tansfers 8 channels in every frame
- Configuration: bits, DMA, interrupts, bit order
  - ESP 7.1.2

I2S

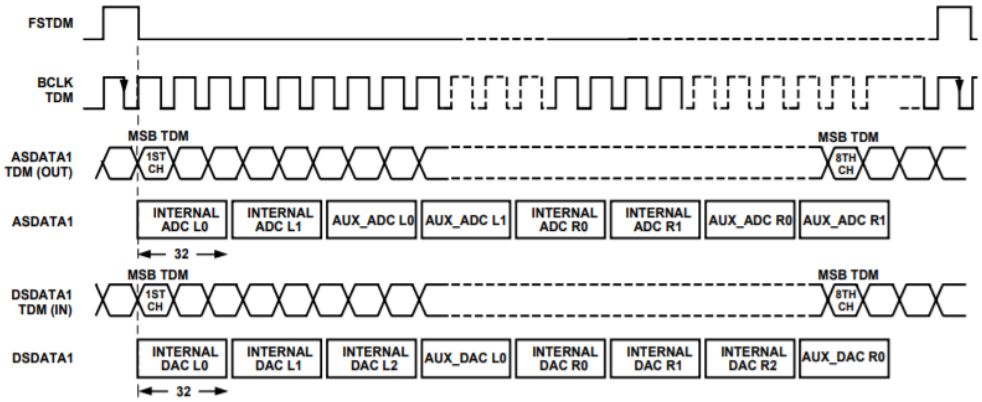
- Three-wire serial bus standard protocol.
- Two time slots for left and right channels.



Figur 3.1: AUX I2S Interface

TDM

- Time-Division Multiplex Mode.
- Two ADC left channels (L0 and L1) and two right channels (R0 and R1) occupy slots #1, #2, #5, and #6 of ASDATA1.
- Six DAC channels occupy the six time slots of DSDATA1.
- Special TDM auxiliary mode allows two external stereo ADCs and one external stereo DAC to be interfaced to form a total of eight input and eight output transfers.

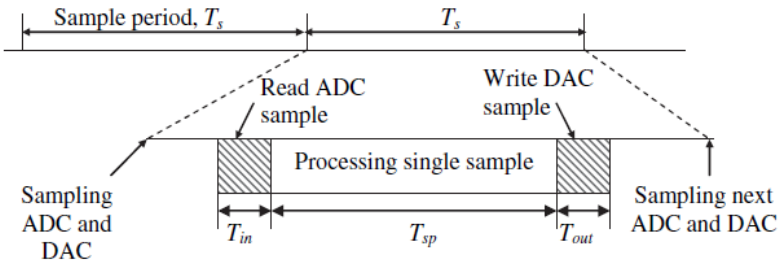


Figur 3.2: TDM Interface.

3.1.2 Sample and Block Processing

- **Sample-by-Sample Processing**
  - Requires that all operations must be completed within the given sampling period.
  - Latency of processing is the total time from the instant the data sample is read to the time the digital output is written to the memory.
    - T_{in} = time needed for the processor to copy current sample from ADC into processor memory. Also includes program access time.
    - T_{sp} = time needed for processing current data samples. Duration depends on complexity of the algorithm and efficiency of the program.
    - T_{out} = time needed to output the processed data to the DAC.

- Overall overhead time for sample-by-sample processing is denoted as T_{os}.
  - Includes T_{in} and T_{out} and response time to interrupt.
- Terminated processing before next sample/block.
  - $T_{sp} \leq T_s - T_{os}$
- Advantages?
  - Predictable processing time (T_s).
  - Delay between input and output within one sample.
  - Low memory usage for storage will though increase for multichannel applications.
  - Results are kept current within the sampling period.
- Disadvantages?
  - Overhead of setup, access, interrupt for every sample.
  - DSP must be fast enough to complete all task before the arrival of next input sample.
  - Not suitable for block based algorithms like Fourier transformation.

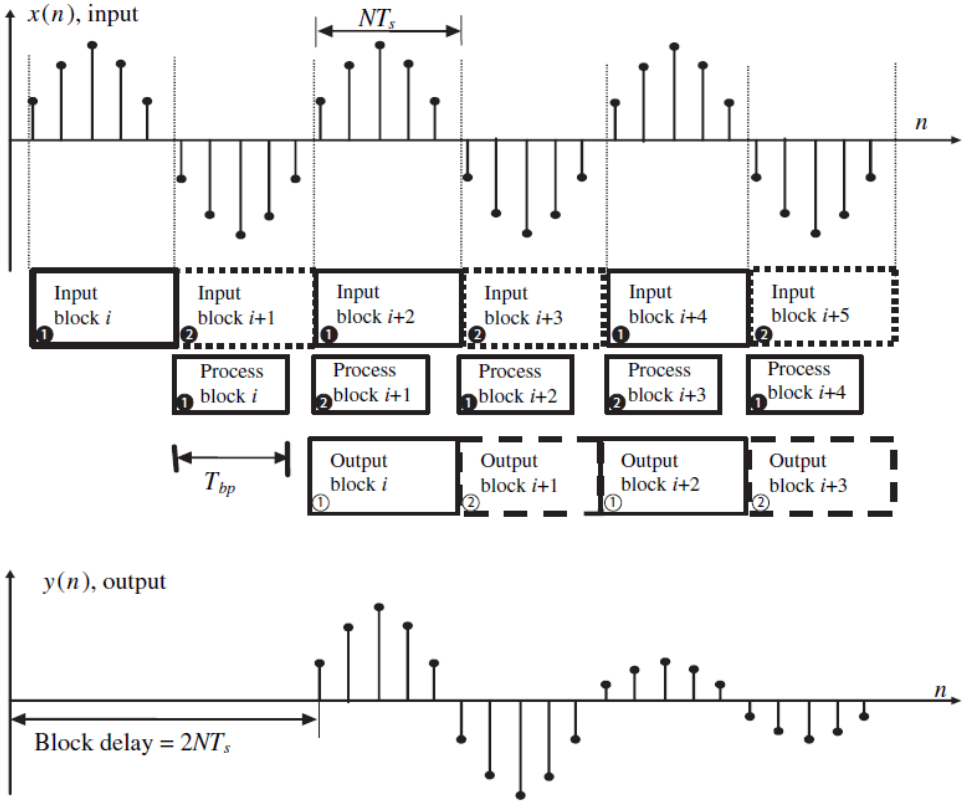


Figur 3.3: Timing details for sample-by-sample processing mode.

- **Block Processing**
  - Samples are gathered in input buffer for N samples received
  - Interrupt for every sample block of size N
  - Block delay = $2 \cdot N \cdot T_s$
  - Doublet buffering for input and output samples
  - DMA used to move data between SPORT and buffers (Sample buffer)

- Advantages?
  - Instruction cycle to compute a block of samples is shorter compared to sample-by-sample processing.
- Disadvantages?
  - Memory of 4 buffers for holding input and output data samples.
  - Block delay $2 \cdot N \cdot T_s$.
  - Complexity in programming switching between buffers.

The block processing system starts by sampling the first five input samples from the ADC to form block *i*. The system continues to sample another five data samples to form block *i + 1*. At the same time, the processor operates on data samples in block *i* and sends the five previously processed samples to the DAC. During the next block period, *i + 2*, another five newer samples are acquired. The processor operates on the data samples in block *i + 1* and outputs the processed data samples in block *i*.



Figur 3.4: Block processing mode (N = 5).

Example of Sample-by-Sample Processing

```
EX_INTERRUPT_HANDLER(Sport0_RX_ISR)
{
// confirm interrupt handling
*pDMA1_IRQ_STATUS = 0x0001;
// Move data from receive buffer to local buffer
// Retrieve all the samples from receive buffer to process buffer
sCh0LeftIn = sDataBufferRX[INTERNAL_ADC_L0];
sCh0RightIn = sDataBufferRX[INTERNAL_ADC_R0];
sCh1LeftIn = sDataBufferRX[INTERNAL_ADC_L1];
sCh1RightIn = sDataBufferRX[INTERNAL_ADC_R1];

Process_Data();

sDataBufferTX[INTERNAL_DAC_L0] = sCh0LeftOut;
sDataBufferTX[INTERNAL_DAC_R0] = sCh0RightOut;
sDataBufferTX[INTERNAL_DAC_L1] = sCh1LeftOut;
sDataBufferTX[INTERNAL_DAC_R1] = sCh1RightOut;
}
```

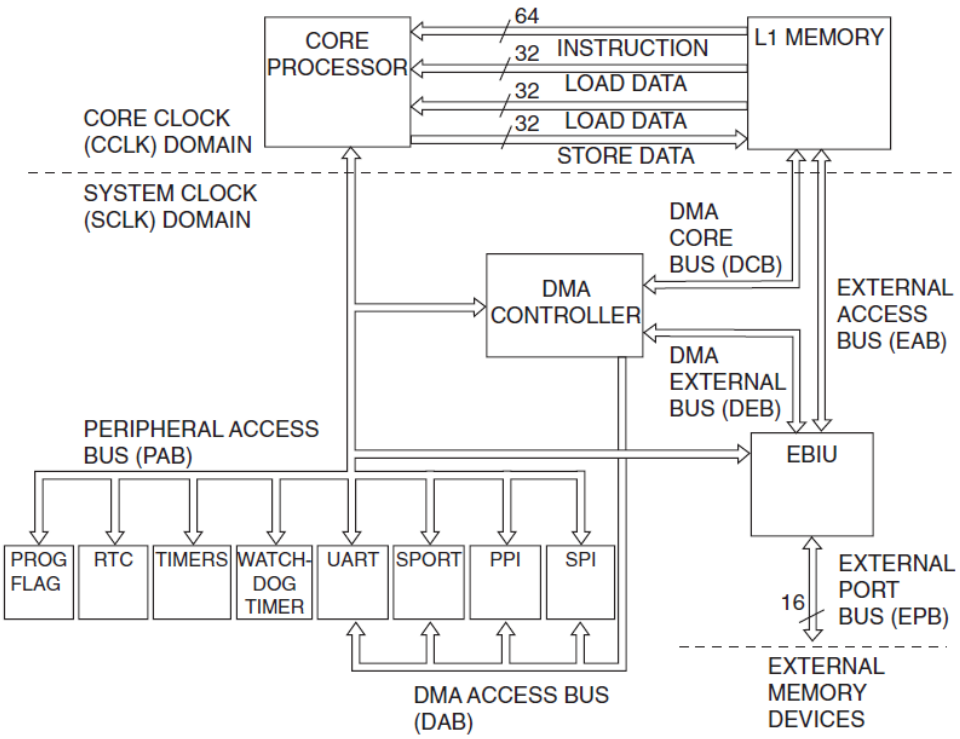
Example of Block Processing

```
EX_INTERRUPT_HANDLER(Sport0_RX_ISR)
{
int i;
static short j=0;
// confirm interrupt handling
*pDMA1_IRQ_STATUS = 0x0001;
// Move data from receive buffer to local buffer
for (i = 0; i < INPUT_SIZE; i++)
{
        // Retrieve all the samples from receive buffer to process buffer
        sCh0LeftIn[i] = sDataBufferRX[INTERNAL_ADC_L0+j];
        sCh0RightIn[i] = sDataBufferRX[INTERNAL_ADC_R0+j];
        sCh1LeftIn[i] = sDataBufferRX[INTERNAL_ADC_L1+j];
        sCh1RightIn[i] = sDataBufferRX[INTERNAL_ADC_R1+j];
        // use the builtin circular buffer to update the index
        j = circindex(j, 4, 4*INPUT_SIZE*TOTAL_FRAME);
}
Process_Data();
...
```

3.1.3 Memory and DMA

Memory Types

- L1 Instruction and Data memory
  - Blackfin processor supports a hierarchical memory model.
  - Single-cycle execute and access.
  - 10's of kBytes - speed $> 600\,\text{MHz}$ (CCLK).
  - Transfer data from memory to registers is arranged in a hierarchy from the slowest (L3 memory) to the fastest (L1 memory).
- L3 Extern memory
  - External off-chip memory like SDRAM.
  - Used to hold large program and data.
  - 100's of Mbytes speed $< 133\,\text{MHz}$ (SCLK).



Figur 3.5: BF533 processor memory architecture.

DMA Modes

- Stop Mode
  - Stopped after transfer is completed
- Auto Buffer Mode
  - Automatic restart when data transfer is completed
- Descriptor Mode
  - Descriptor list in memory that specifies the next data transfer (source and destination address).
  - Descriptor is automatic read by the DMA controller.

DMA Initialization

- Start address
- Peripheral device map
  - `DMA1_PERIPHERAL_MAP = 0x1000; // channel 1 for SPORT0 RX`
  - `DMA2_PERIPHERAL_MAP = 0x2000; // channel 2 for SPORT0 TX`
- Configuration
  - Set DMA channel 1 for autobuffer mode, enable data interrupt, retain DMA buffer, 1D DMA using 16-bit transfers, DMA is a memory write, and DMA channel 1 is not enabled.
    - `DMA1_CONFIG = 0001 0000 1000 0110b`
  - Set DMA channel 2 for autobuffer mode, disable data interrupt, retain DMA buffer, 1D DMA using 16-bit transfers, DMA is a memory read, and DMA channel 2 is not enabled.
    - `DMA2_CONFIG = 0001 0000 0000 0100b`
- Count (X,Y)
- Modify (X,Y)

Implementation and Optimization

4.1 Lektion 27-02-2018

1. Method steps for DSP projects
2. Implementation
3. Optimization
4. Built-in and Native fractional

- ESP 6.2.3.2 (Overflow and scaling)
- ESP 8.1 - 8.2 except 8.2.5, 8.2.6, 8.2.7 (Code Optimization)
- CrossCore Embedded Studio 1.0 C/C++ Compiler and Library Manual
  - 1-115 - 1-118 (Fixed-Point Types, Intrinsic and Native)
  - 1-417 - 1-421 (Fixed-Point Data Representation)
  - 2-57 - 2-60 (Fixed-Point Examples)

4.1.1 Method steps for DSP projects

- Specification
  - What is the problem?
- Algorithm development
  - What is the solution?
- Algorithm implementation and optimization
  - How good is the solution?

4.1.2 Implementation

- Algorithm decomposition and estimation
  - Requirements to drive decisions
  - Decomposition of algorithm into blocks
  - Estimation of needed precision
    - Fixed point 16, 32bit?
    - Floating point 32, 64bit?
    - Signal to noise ratio
    - Dynamic range
  - Estimate needed computation performance
    - Based on required sample rate and DSP choice (MIPS)
    - Cycle estimation for each block of algorithm
- Porting algorithm to target
  - Choice of DSP processor
    - Number of multiplications per. second (MIPS)
    - Data communication rate (fs)
    - Memory requirements (program and data)
    - Use estimated cycle usages based on similar algorithm implementations
    - Sum of algorithm cycles compared to CPU speed and sampling rate
  - Signal to noise ratio (SNR)
    - Noisee(n) is introduced by multiplications between variables
    - Equal distributed noise with m bits $0 \leq |e| \leq 2^{-(m+1)}$
  - Dynamic range
  - Fixed-point vs. floating-point?
    - Quantization
    - Saturation
  - Verification and optimization
    - Compare performance with high level simulation (Matlab)

4.1.3 Optimization

- Algorithm on DSP
  - Start programming solution in 'C'
  - Separate optimization of modules with compiler options when debugging/testing
  - Use intrinsic built-in or assembler to optimize final design
  - Move solution to 'C++' framework
- Optimization on target
  - Compiler optimization (size vs. speed) in *Compiler Options*
  - Cycle measuring of blocks uses *CYCLES Register*
  - Profiling of code
    - Percentage of time spent executing function
  - Effective C code implementation using the manual
    - Optimizing structures
    - Constants statically
    - Function inline (inline keywords)
    - Inline assembler statements
    - Improving conditional code
    - Fractional recommendations
    - Function #pragmas
    - Intrinsic built-in functions
    - Native fixed-point
  - Looping guidelines
    - Keep loops short and simple
    - Don't use global variables in loops
    - Use circular buffers
    - Use const qualifiers to function calls

4.1.4 Built-in and Native fractional

- Fixed-point and fractional data
  - Integer Arithmatic
    - **BAD** Uses shifts to implement fractional multiplication

- Built-in Functions
  - **GOOD** Uses built-ins to implement fractional multiplication (less intuitive)
  - `#include <fract.h>`
  - `fract16`
  - `fract32`
- Native fixed-point types
  - **GOOD** Uses built-ins to implement fractional multiplication
  - Standard C operators +, , *, and /
  - Are welldefined and clear to the compiler
  - `#include <stdfix.h>`
  - `fract`
  - `accum`

Object Oriented Design

5.1 Lektion 06-03-2018

1. Quality Software - Object Oriented Design and DSP (C++)
2. Design Patterns
3. 3-Layered Architecture
4. DSP Design Framework

5.1.1 Quality Software

Characterization

- Easy to maintain and expand because of its clear architecture and flexibility.
- Have a high degree of reuse - **low coupling** and h**igh cohesion**, modularization.
- Easy to understand both high and low level of abstraction **simplicity**.
- Have easily recognizable **design patterns** - provide understanding.
- Other programmers will not be afraid working on it - **understandable**.

5.1.2 Object Oriented Design C++

- Classes - Abstract Data typing - Incapsulation
- Abstract base classes - Interfaces
- Virtual member functions - Polymorphism
- Operator overloading
- Inheritance
- Templates

Hard and soft real-time systems

A real-time system must react to stimuli from the operator or other devices within time intervals dictated by its environments. A deadline is the instant at which a result must be performed. A system with at least one hard deadline is called a hard real-time system else a soft real-time system.

- **Soft deadline**
  - System can miss some deadlines, but eventually performance will degrade if too many are missed. The result is accepted to be ready after the deadline.
- **Hard deadline**
  - System must absolutely hit every deadline. Missing a deadline constitutes a system failure.

C++ features

- Many C++ features have no performance cost.
  - Namespaces, overloaded functions, inheritance.
- Avoid dynamic allocation using new and delete use static allocations if **hard deadline**.
- Same goes for minimizing use of virtual functions and keeping basic data types in structures.

5.1.3 Design Patterns

- Describes a problem that occurs again and again.
- Provides a general solution to the problem.
- The basis for the implementation of an object-oriented language.
- Improves reuse of design.
- Improve understanding and communication of a design.
- A design pattern consists of 4 elements:
  - Name of the design pattern.
  - Problem description.
  - Solution and its objects (relationships, responsibility and cooperation).
  - Impact of solution (trade-offs and cost/benefits).

Types

- Creational Patterns
  - Responsible for creating objects
- Structural Patterns
  - Describes the composition of classes and objects
- Behavioral Patterns
  - Describes the interaction between classes and objects and allocation of responsibilities

Composite combines items for tree structures to represent part-whole hierarchies. It lets clients treat primitive and composite objects in the same way.

Use the composite pattern when representating a "part-whole"structure. When no distinction should be drawn (from client side) between the individual and composite objects and all objects in the structure (primitive and composite) must be handled in a consistent manner.

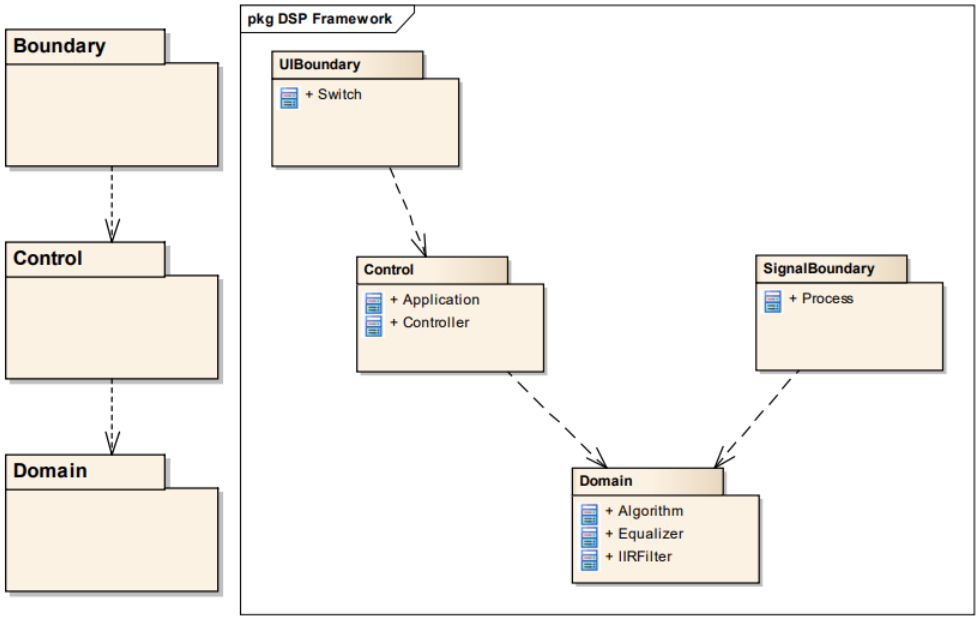
- **Pros**
  - Primitive objects can be combined into complex objects.
  - Makes the client simple (avoid case statements).
  - Easy to add new types of objects; no affect on client's code
- **Cons**
  - Can make the code too general.
  - `Add()` and `Remove()` in primitive objects does not make sense.

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

Figur 5.1: Gof Design Patterns, Visual Paradigm Community Circle

5.1.4 3-Layered Architecture

- Change in presentation layer (boundary) makes no changes in the other layers. (perhaps in control)
- Change in control layer (control) may lead to changes in the boundary layer.
- Changes in the domain layer (domain) can cause changes in all layers.



Figur 5.2: DSP Layered Design.

5.1.5 DSP Design Framework

- The framework provides a **clear architecture with flexibility** being able to change UI or adding new algorithms.
- The framework has a degree of reuse with **low coupling** and **high cohesion**, realized with a layered architecture and classes with well defined responsibility.
- Efficient design - use of **static object instantiation**.
- The framework uses recognizable **design patterns** build on the 3 layer model, composite and pipes and filters patterns
- Hopefully **understandable** for other programmers!

Runtime Library

6.1 Lektion 20-03-2018

1. DSP Run-Time Library
2. Cycles Measurement
3. Complex Numbers
4. Filter IIR, FIR- FFT, iFFT
5. Convolution
6. Statistical (Auto+Cross-correlation)

6.1.1 DSP Run-Time Library

- Integer Arithmetic to Encode Fractional Semantic
  - Not supported by Run-Time Lib
- Built-In Functions (compiler intrinsics)
  - `fract16, fract32`
  - Fully supported by Run-Time Lib
- Native Fixed-Point Types (fract and accum)
  - `fract, long fract, accum`
  - Partly supported by Run-Time Lib

Conversions

- The `stdflx.h` provides functions to conversions
- Useful for converting between native types (`fract`, `long fract`) and integer typedefs (`fract16`, `fract32`).
- Convert a bit pattern to a fixed-point type `bitsfx`
- Convert a fixed-point type to a bit pattern `fxbits`
- `[hr, r, lr, uhr, ur, or ulr]`
- Conversion float to fract/fract16
  - `fract16 af16 = float_to_fr16(a);`
  - `a = fr16_to_float(af16);`

Fixed-Point Type Constant Suffixes and Macros

Type	Suffix	Example	Minimum Value	Maximum Value
`short fract`	hr	0.5hr	SFRACT_MIN	SFRACT_MAX
`fract`	r	0.5r	FRACT_MIN	FRACT_MAX
`long fract`	lr	0.5lr	LFRACT_MIN	LFRACT_MAX
`unsigned short fract`	uhr	0.5uhr	0.0uhr	USFRACT_MAX
`unsigned fract`	ur	0.5ur	0.0ur	UFRACT_MAX
`unsigned long fract`	ulr	0.5ulr	0.0ulr	ULFRACT_MAX
`short accum`	hk	12.4hk	SACCUM_MIN	SACCUM_MAX
`accum`	k	12.4k	ACCUM_MIN	ACCUM_MAX
`long accum`	lk	12.4lk	LACCUM_MIN	LACCUM_MAX
`unsigned short accum`	uhk	12.4uhk	0.0uhk	USACCUM_MAX
`unsigned accum`	uk	12.4uk	0.0uk	UACCUM_MAX
`unsigned long accum`	ulk	12.4ulk	0.0ulk	ULACCUM_MAX

Figur 6.1: Suffixes and Macros, CrossCore Embedded Studio 2.2.0

Header Files

DSP header files contain prototypes for the DSP library functions

- `complex.h`
  - Basic complex arithmatic functions
- `cycle_count.h`
  - Basic cycle counting
- `cycles.h`
  - Cycle counting with statistics
- `filter.h`
  - Filters and transformations
- `math.h`
  - Math functions
- `matrix.h`
  - Matrix functions
- `stats.h`
  - Statistical functions
- `vector.h`
  - Vector functions
- `window.h`
  - Window generators

6.1.2 Cycles Measurement

- Build-in macros to measure cycles
- Accumulates statistics
- Enabled by `-d DO_CYCLE_COUNTS`

EXAMPLE cycle_count.h

```
#include <cycle_count.h>
#include <stdio.h>

extern int
main(void)
{
cycle_t start_count;
cycle_t final_count;

START_CYCLE_COUNT(start_count);
Some_Function_Or_Code_To_Measure();
STOP_CYCLE_COUNT(final_count,start_count);

PRINT_CYCLES("Number of cycles: ",final_count);
}
```

EXAMPLE cycles.h

```
#include <cycles.h>
#include <stdio.h>

extern void foo(void);
extern void bar(void);

extern int
main(void)
{
cycle_stats_t stats;
int i;

CYCLES_INIT(stats);

for (i = 0; i < LIMIT; i++) {
        CYCLES_START(stats);
        foo();
        CYCLES_STOP(stats);
}
printf("Cycles used by foo\n");
CYCLES_PRINT(stats);
CYCLES_RESET(stats);
```

```
for (i = 0; i < LIMIT; i++) {
        CYCLES_START(stats);
        bar();
        CYCLES_STOP(stats);
}
printf("Cycles used by bar\n");
CYCLES_PRINT(stats);
}
```

OUTPUT

```
Cycles used by foo
AVG   : 25454
MIN   : 23003
MAX   : 26295
CALLS : 16

Cycles used by bar
AVG   : 8727
MIN   : 7653
MAX   : 8912
CALLS : 16
```

6.1.3 Complex Numbers

Datatypes

- `complex_float`
- `complex_double`
- `complex_fract16`
- `complex_fract32`

Operations (intrinsics only)

- Absolute Value
- Addition
- Subtraction
- Multiply
- Division

- Conjugate
- Exponential
- Normalization

6.1.4 Filter IIR, FIR- FFT, iFFT

FIR

- `fract16, fract32, fract, long fract`

```
#include <filter.h>

fir_init(state, coeffs, delay, NUM_COEFFS, 0);
fir_fr32(input, output, NUM_SAMPLES, &state);
```

IIR

- `fract16, fract32, fract, long fract`

```
#include <filter.h>

iir_init (filter_state,coeffs,delay,NUM_STAGES);
iir_fr16 (signal,output,NUM_SAMPLES,&filter_state);
```

FFT

- Discrete Fourier Transformation
  - Transform from time to frequency domain
    - Sum of products with time-domain sequence of sine and a cosine wave
  - Digital signal $x(n)$, a finite-duration sequence of length N

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j\frac{2\pi}{N}kn} \tag{6.1}$$

- Defining a Twiddle factor W_N

$$W_N = e^{-j\frac{2\pi kn}{N}} = \cos\left(\frac{2\pi kn}{N}\right) - j\sin\left(\frac{2\pi kn}{N}\right) \tag{6.2}$$

- DFT defined by a Twiddle factor

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn} \ , \ k = 0, 1, ..., N-1 \tag{6.3}$$

- k is the frequency index
- Frequency spacing is $\frac{f_s}{N}$
- $X(k)$ are complex values
  - Magnitude and phase components

$$X(k) = Re[X(k)] + jIm[X(k)] = |X(k)|e^{j\phi(k)} \tag{6.4}$$

- FFT is a fast computational version of DFT
  - N-point DFT radix-2, here N must be a power of 2

$$\frac{FFT}{DFT} = \frac{\log_2 N}{2N} \tag{6.5}$$

- Twiddle table

```
twidfftrad2_fr16(m_twiddle_table,N_FFT);
```

- FFT and magnitude

```
rfft_fr16(input,m_fft_output,m_twiddle_table,1,N_FFT,&block_exponent,2);

fft_magnitude_fr16(m_fft_output,m_real_magnitude,N_FFT,block_exponent,1);
```