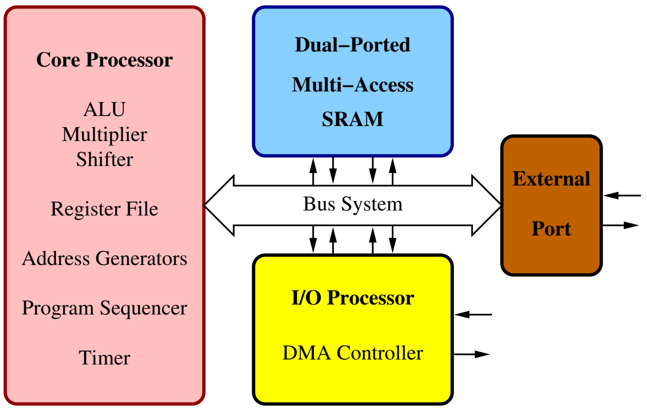
# SIGNAL PROCESSORS

## DSP PROGRAMMING TUTORIAL



Gerhard Doblinger*
Institute of Telecommunications
Vienna University of Technology

August 2011

*phone 043 1 58801 38927, room CG0209 mail: Gerhard.Doblinger@tuwien.ac.at

# Contents

# 1 Introduction

This tutorial describes the exercise part of course 389.112 "Signal Processors". The main topic is centered on programming of basic signal processing problems on a commercially available digital signal processor (DSP). With these exercises, we intent to train practical skills in DSP programming, and to deepen the presentation offered in the lecture part of this course. All steps of the "Top-Down-Design" as discussed in the lecture can be carried out. By writing your own programs, you will get a better understanding of architecture principles of DSPs, and an impression about the complexity of DSP assembly language programming.

There is no need to start writing a DSP program from scratch. You will use prototype programs where only a few lines of code for each problem must be developed. Initialization, DMA, and analog signal input/output handling are managed by the prototype programs. Thus, you can focus on algorithm implementation without the need to fiddle around with tedious bit manipulations.

We use a DSP target system with a SHARC 32 bit floating-point processor from Analog Devices for two reasons: First, using floating-point arithmetic facilitates number processing since nearly no finite word length effects occur. Second, the selected DSP offers an easy to learn assembly language with virtually no pipeline effects. In addition, the SHARC architecture is presented as a representative modern DSP in the lecture part. In this tutorial, we summarize additional material needed to program this DSP in assembly language. A comprehensive description can be found in the processor manuals available as PDFs from `www.analog.com`.

All exercises have been designed to such an extent that you will learn to use more and more features of the DSP when stepping from one problem to the next. In addition, problem difficulty is increased accordingly. It is not required to solve all problems. You may choose where to stop. It is better to leave some problems unsolved and select those ones you are interested in. The main goal should be to solve the selected problems free of errors. Nevertheless, with some motivation and a little endurance you should even be able to solve all problems in the scheduled time (1.5 hours/week course).

In order to successfully solve the problems, you will need a basic knowledge in digital signal processing. Such a knowledge is offered in course 389055 "Signals and Systems 2", and in the appendix of the course book.[1]

You will finish the exercise part of this course by a short report describing your solutions. This report should include theoretical derivations (if present), listings of your code, and results. Plots of the results can simply be made by taking photographs of oscilloscope and spectrum analyzer screens. You can also include MATLAB plots, if you simulate a problem before programming it in assembly language. Your report will be discussed with all group members at the course end.

[1]G. Doblinger, Signalprozessoren, Architekturen - Algorithmen - Anwendungen, 2. Auflage 2004, J. Schlembach Fachverlag, ISBN 3-935340-43-5.

## 2 Development tools and hardware target system

With the development tools used in this course you can perform a complete development process as outlined in the “Top-Down-Design” already mentioned above. DSP programs written in assembly language or in C are tested using a debugger. Alternatively, you can test your programs with a DSP simulator. The hardware target system contains an ADSP-21065L DSP. Analog input/output is done via a 16 bit stereo codec with selectable sampling frequencies between 8 kHz and 48 kHz. In the prototype program, input samples are stored in two data registers during each sampling interval. After processing this samples by your DSP program, these registers are overwritten with processed samples. Thus, the only tasks of your program are initialization of variables and processing samples stored in registers. All other tasks are already coded in the above mentioned prototype programs. Details of these programs can be found in the appendix. In addition, the basic steps of program development and a check list of common errors can also be found in the appendix.

## 3 DSP number representations

The ADSP-21065L operates with 32 bit floating-point and with 32/40 bit fixed-point arithmetic. If you are a DSP novice, it is recommended that you use the floating-point units in order to avoid scaling operations. Nevertheless, it is a good idea to briefly discuss number representations of DSPs. Even with floating-point numbers you may get trapped by certain pitfalls.

### 3.1 Fixed-point numbers

As discussed in the lecture part of this course, fixed-point number representations offer a low hardware complexity, low power consumption, and lower costs as compared to floating-point arithmetic. However, due to the low dynamic range we must take care of underflows and overflows. Additionally, finite word length effects (rounding noise, limit cycles) may affect the performance of the implemented algorithm. These effects are by far less important when using floating-point arithmetic.

Fixed-point number representations in DSPs commonly use the **two’s complement number representation**. This number format offers several advantages like uniqueness of number zero, easy subtraction, and efficient multiplication algorithms. Details can be found in the course book. The 32 bit fixed-point units of the ADSP-21065L include two 80 bit accumulators to calculate a sum of products with extended precision and dynamics. This precision is higher than the 32/40 bit floating-point precision. Therefore, it makes sense to use the processor’s fixed-point arithmetic in certain applications.

Two’s complement numbers of $N$ bits word length use the most significant bit (MSB) as sign bit. Precision is determined by the least significant bit (LSB). The weight of each bit is set by the **binary point (comma)** which can be chosen arbitrarily. Fixed-point numbers may be classified by an $I.Q$ format with integer part I (number of bits left to binary point), and fractional part Q (number of bits right to binary point). In case of a 32 bit word length, a 1.31 format (called fractions), and a 32.0 format (integers) are commonly used. In assembly language, you must specify the format when multiplying fixed-point numbers (see the lecture notes, or the course book). In the sequel, we will use the $I.Q$ format with $I = 1$.

**Problem 3.1:** Repeat the following discussion with the 32.0 format (integers).

The bit weighting of a 32 bit word in 1.31 format is shown below each bit position in Fig. 1.

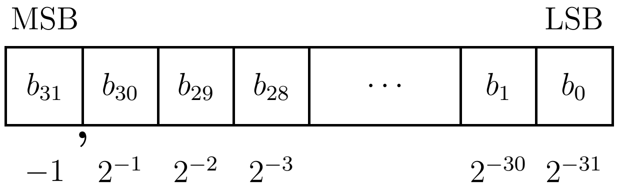


Figure 1: 32 bit two's complement number representation in 1.31 format (binary point (comma) between $b_{31}$ and $b_{30}$).

The decimal value $x$ of an $N$ bit number in two's complement format ($I.Q$ with $I = 1$ and $Q = N - 1$) is determined by

$$x = -b_{N-1} + \sum_{k=1}^{N-1} b_{N-1-k}\, 2^{-k} \tag{1}$$

with $b_{N-1-k} \in \{0, 1\}$, and $-1 \leq x \leq 1 - 2^{-N+1}$. Precision is given by $\Delta = 2^{-N+1}$ (e.g $\Delta = 2^{-31} \approx 5 * 10^{-10}$ for $N = 32$).

Positive numbers (sign bit $b_{31} = 0$) are coded normally like $0.8125 \leftrightarrow 0,1101$. Negative numbers (sign bit $b_{31} = 1$) are obtained by coding $2 - |x|$ normally, e.g. $-0.8125 \rightarrow 2 - 0.8125 \leftrightarrow 1,0011$.

With binary logic, we can easily obtain the two's complement (negation) of a number:

| | | | | | | |
|---|---|---|---|---|---|---|
| binary representation of 0.8125: | 0 | , | 1 | 1 | 0 | 1 |
| invert bits (one's complement): | 1 | , | 0 | 0 | 1 | 0 |
| add LSB: | 0 | , | 0 | 0 | 0 | 1 |
| two's complement result: | 1 | , | 0 | 0 | 1 | 1 |

Thus, subtraction is performed by taking the two's complement of the subtrahend followed by an addition.

The overflow characteristic of two's complement numbers is shown in Fig. 2. Such an overflow results in large amplitude jumps giving rise to strong nonlinear signal distortions. In case of an $I.Q$ fraction format with $I = 1$, the overflow characteristic exhibits a period of 2 (**modulo-2 overflow behavior**). This characteristic is inherently present in a two's complement arithmetic. Although it has the disadvantage of severe signal distortions in

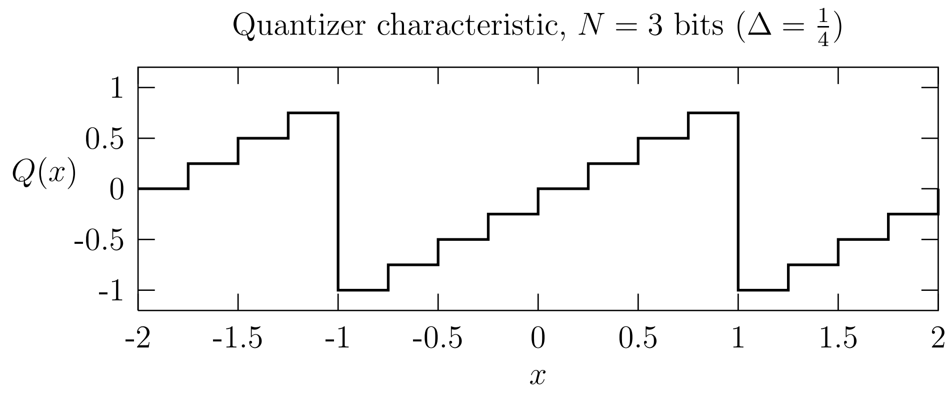


Figure 2: Two's complement quantizer and overflow characteristic (3 bit word length)

case of overflow, there is a benefit in certain applications: When calculating a sum of products, partial overflows have no influence as long as the final result is within $[-1, 1)$. If we add e.g. $\frac{2}{4}$ to $\frac{3}{4}$, we get $-\frac{3}{4}$ according to Fig. 2. This result is numerically wrong. However, if we subtract from this result $\frac{2}{4}$, then we get the correct result $\frac{3}{4}$. This property is unique to the build-in overflow behavior of two's complement numbers.

A modulo-2 overflow must be avoided in feedback loops of recursive systems (systems with feedback) because oscillations (overflow limit cycles) may occur. This instability effect is suppressed (at least in second order systems) by handling overflows by a saturation mode characteristic as shown in Fig. 3.

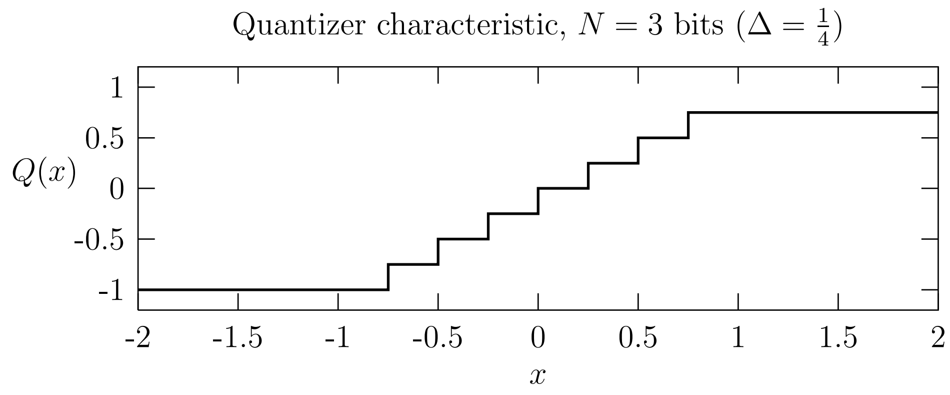


Figure 3: Two's complement quantizer with saturation mode overflow characteristic (3 bit word length)

As outlined in the lecture, modern DSPs use several registers and memories with different word lengths. When transferring two's complement data between storage cells of different word lengths, we must pay attention to the alignment of a lower length word in a higher length cell. Bits are filled with zeros in case of left alignment (shift towards MSB position). Right alignment (shift towards LSB position), however, requires a **sign extension** in order to get numerically correct results. Both alignment schemes are depicted in Fig. 4 with a 4 bit word to 8 bit word transfer. The **ADSP-21065L processor** stores 32 bit fixed-point numbers **left aligned** in 40 bit registers. Note that the magnitude of the number is changed depending on $I$ and $Q$ of an $I.Q$ format.

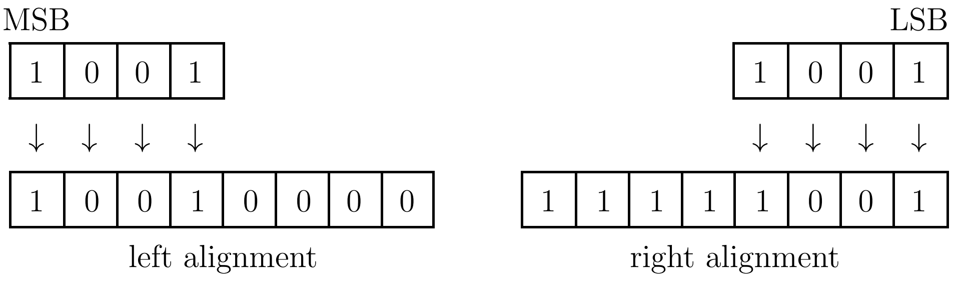


Figure 4: Transfer of a 4 bit two's complement number to an 8 bit register (right alignment with sign extension)

> **Problem 3.2:** A 32 bit two's complement number should be transferred to a 40 bit register. Determine how its value changes at lift/right alignment in case of 1.31 (1.39) format, and in 32.0 (40.0) format, respectively.

Multiplication of two's complement numbers exhibits some subtleties since product word length differs from operands word lengths. If we multiply an $N_1$ bit binary number with an $N_2$ bit number (both in two's complement representation), then the product has $N_1+N_2-1$ binary digits (consider $0,1\times 0,1 = 0,01$ as an example). With 32 bit operands the multiplication results has 63 binary digits. Depending on how (left or right aligned) the product is stored in a 64 bit register, we get an extra LSB or MSB. Therefore, fixed-point multiplications with the ADSP-21065L must be differently specified for integers, and for fractions, respectively. A product of integers is stored right aligned whereas products of fractions are stored left aligned in an 80 bit accumulator register.

> **Problem 3.3:** What values do you get if 0111 is multiplied with 1001? Consider both cases where the numbers are in 1.3 and 4.0 format. What are the results, if the first number is in 1.3 and the second in 2.2 format?

> **Problem 3.4:** Show that the multiplication of an $I_1.Q_1$ number with an $I_2.Q_2$ number yields a product in $(I_1 + I_2 - 1).(Q_1 + Q_2)$ format.

These examples indicate the comma shifting when multiplying numbers in different $I.Q$ formats. The changing comma positions must be taken into account if numbers are added to or subtracted from products.

**Problem 3.5:** Represent decimal number $x = -0.25$ in binary 1.3 format and $y = 1.5$ in 2.2 format. Multiply these numbers and add $z = 0.125$ in 1.7 format to the product. Determine how to shift the binary representation of the product in order to get the correct result of $xy + z$.

## 3.2 Floating-point numbers

Conventional fixed-point DSPs use the two's complement number representation. Although there is no common floating-point format for commercially available DSPs, most of them use the 32 bit IEEE number representation. The bit allocation of this format is illustrated in Fig. 5.

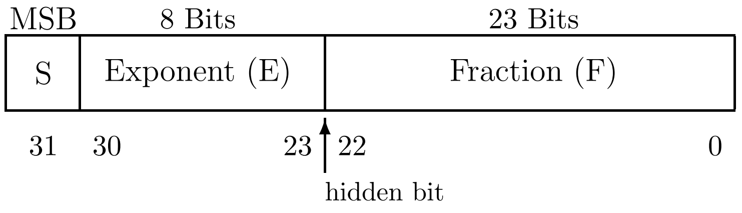


Figure 5: Bit allocation of the 32 bit IEEE floating-point number representation

A floating-point number is decomposed of sign bit $S$, an exponent $E$ which takes care of the dynamic range, and a mantissa $M$ having the maximal possible magnitude. Mantissa $M$ has 24 bit word length including a hidden bit always set to 1. Note that there is no sign bit in the mantissa. Thus, only the fraction $F$ has to be stored. Exponent $E$ uses an offset of 127 to represent positive and negative exponent values. Thus, a 32 bit IEEE format floating-point number $x$ may be represented by

$$x = (-1)^S \; * \; 2^{E-127} \; * \underbrace{1.F}_{M},$$

with $S \in \{0,1\}$, $1 \le E \le 254$, $0 \le F < 1$ $(1 \le M < 2)$. Precision is $\Delta = 2^{-23} \approx 10^{-7}$, and magnitude range is approximately $10^{-38} \ldots 10^{38}$.

The ADSP-21065L DSP supports a 40 bit IEEE format in addition to the 32 bit format. With an extended mantissa of 32 bit the precision is $\Delta = 2^{-31} \approx 5 * 10^{-10}$. The ADSP-21065L floating-point unit operates with 40 bit word length for both operands and result. Rounding occurs if 40 bit results are transferred to 32 bit storage cells.

A precision comparison of fixed-point and IEEE floating-point numbers in Fig. 6 shows the number of relevant binary digits depending on the magnitude $|x|$. With fixed-point numbers the precision increases with $|x|$ since more bits are used for larger values. Floating-point numbers show a precision independent of $x$ because the mantissa always has maximal possible magnitude. Thus, floating-point number representations act like an automatic gain control (AGC) to handle signal amplitudes.

One important and frequently overlooked difference between fixed-point and floating-point numbers is the fact that **not every bit combination yields a valid floating-point number**. Reason is the special bit allocation in Fig. 5. From a programmer's

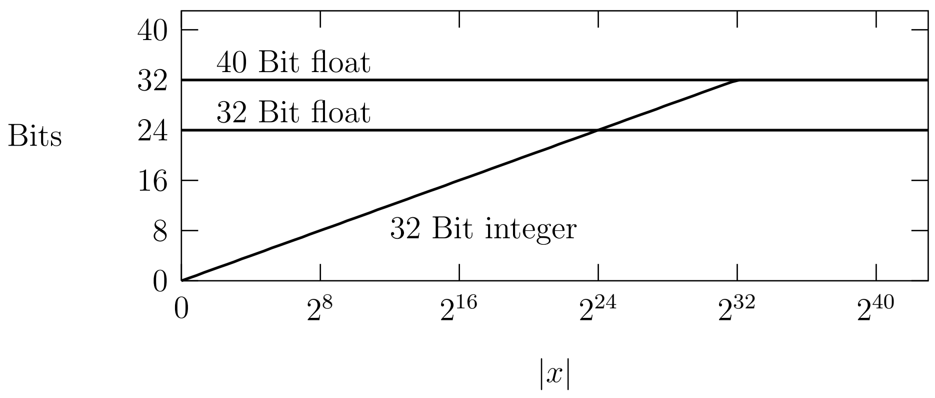


Figure 6: Precision comparison of fixed-point, and IEEE floating-point numbers

point of view it is necessary to **initialize memory locations which store floating-point numbers**. Number crunching with invalid floating-point numbers is a common error of newcomers since in general variables are not initialized by assemblers.

There are the following special 32 bit IEEE floating-point numbers:

| type | exponent (E) | fraction (F) | value |
|---|---|---|---|
| NAN | 255 | $\neq 0$ | not defined |
| $\infty$ | 255 | 0 | $\pm\infty$ |
| 0 | 0 | 0 | $\pm 0$ |

An **invalid result (NAN, not a number)** occurs at undefined expressions like $\frac{0}{0}$, or when using a fixed-point number in a floating-point operation. Note that assemblers do not automatically perform format conversions in arithmetic expression evaluations. Floating point exceptions are handled in various ways. Typically, special bits (flags) are set in an arithmetic unit status register. Alternatively, software interrupts may be triggered. However, the best strategy is to avoid floating-point exceptions by a serious programming style. Furthermore, you should be aware that arithmetic operations can results in $+0$ or $-0$ depending on the rounding operation. Consequently, you may get strange results when checking zero results to perform branches in a program!

**Problem 3.6:** Determine the 32 bit IEEE floating-point format bit allocation (e.g. as hexadecimal numbers) of the decimal numbers

$$2 \quad 1 \quad 0.5 \quad 0.25 \quad -1$$

# 4 ADSP-21065L architecture

In this section, we present a compact description of the DSP architecture used in this course. This architecture has been already discussed in some detail in the lecture part. We will describe only those details which are relevant to program the problems. You can use the SHARC DSP family user manuals as a reference (visit `www.analog.com` to download this documentation).

The ADSP-21065L architecture and instruction set is well matched to the requirements of digital signal processing applications. Nearly all instructions are executed in a single clock cycle with virtually no pipeline effects. Therefore, instruction execution time is 16.7 ns at 60 MHz clock frequency. Although this clock frequency is not impressive from a today's point of view, the DSP is more than fast enough to implement all the problems of this course. There is no need to use a more powerful processor. The reason of a good performance at low clock frequencies is an execution of several tasks in a single clock cycle:

- 3 arithmetic operations,
- 2 data transfers,,
- 2 address register modifications,
- instruction fetch, decode, and execute (3-stage instruction pipeline),
- DMA transfer between peripherals and internal memory.

That kind of parallel processing is enabled by an architecture as depicted in Fig. 7.

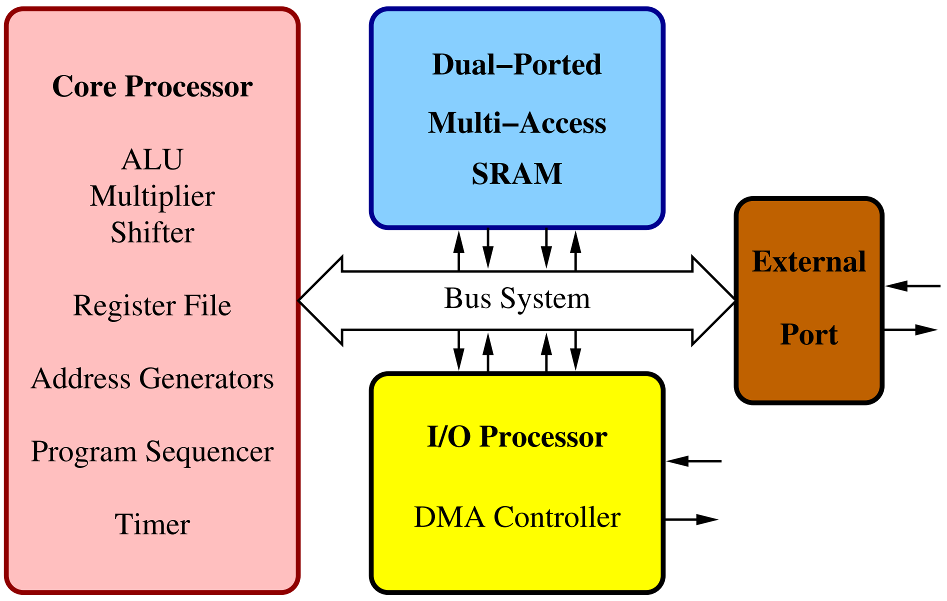


Figure 7: ADSP-21065L architecture overview

Data processing and address generation is done in the **core processor** which obtains data and instructions form a dual-ported multi-access memory. All operations are carried out via registers. The **I/O processor** includes a DMA controller and operates simultaneously to the core processor. Thus, data transfers can take place whilst the DSP core executes instructions. This feature is especially needed in block processing algorithms using swinging buffers (see the lecture notes). An external port allows for connections of additional memory and parallel operation interface devices.

According to the **Harvard architecture** concepts as outlined in the lecture, internal memory is subdivided into two independent blocks to simultaneously read an instruction and a data word. An additional data word can be read when the internal cache memory is utilized. The 544 kbit storage area can be configured per software to allow for various combinations of **program memory (PM)** and **data memory (DM)**. Table 1 lists the available memory including segment names as used by the DSP assembler (see the prototype program in appendix A). Program memory can store 48 bit instructions and 48/32 bit data. Data memory contains 32 bit data words only.

| memory | count | word length | type | segment name |
|---|---:|---|---|---|
| PM | 3840 | 48 bit | instruction word | `pm_code` |
| PM | 1024 | 32 bit | data word in PM | `pm_data` |
| DM | 7168 | 32 bit | data word in DM | `dm_data` |
| DM | 1048320 | 32 bit | data word in ext. SDRAM | `sdram_data` |

Table 1: ADSP-21065L target system memory map I for assembly programs

| memory | count | word length | type | segment name |
|---|---:|---|---|---|
| PM | 2664 | 48 bit | instruction word | `seg_pmco` |
| PM | 1024 | 48 bit | data word in PM | `seg_pmda` |
| DM | 4096 | 32 bit | data word in DM | `seg_dmda` |
| DM | 1280 | 32 bit | heap data word | `seg_heap` |
| DM | 2816 | 32 bit | stack data word | `seg_stak` |

Table 2: ADSP-21065L target system memory map II for C programs

There are two memory maps used in C programming. Only internal memory is used in Table 2 giving rise to a fast simultaneous access of PM and DM. The memory map in Table 3 offers a larger address space due to the external SDRAM. However, a simultaneous access to PM and DM requires an extra clock cycle at least. During SDRAM auto-refresh even 13 clock cycles are needed!

In this course, you will use the functional units of the core processor only. Therefore, we focus in our architecture description on the core processor units shown in Fig. 8. All

| memory | count | word length | type | segment name |
|---|---|---|---|---|
| PM | 3688 | 48 bit | instruction word | `seg_pmco` |
| PM | 2048 | 32 bit | data word in PM | `seg_pmda` |
| DM | 6144 | 32 bit | data word in DM | `seg_dmda` |
| DM | 4096 | 32 bit | heap data word (ext. SDRAM) | `seg_heap` |
| DM | 4096 | 32 bit | stack data word (ext. SDRAM) | `seg_stak` |

Table 3: ADSP-21065L target system memory map III for C programs

operations of the 3 functional units are performed via a set of 16 universal **40 bit data registers**. Universal registers hold operands and results of all operations. Registers are **dual-ported** which implies that the same register can store an operand and a result in a single clock cycle. This feature simplifies assembly programming since we do not need to take care on proper register selection and a possibly copying of register contents between operations. There is a second register set (shadow register set) to be used e.g. during interrupts to avoid push/pop of registers to save/restore their contents.

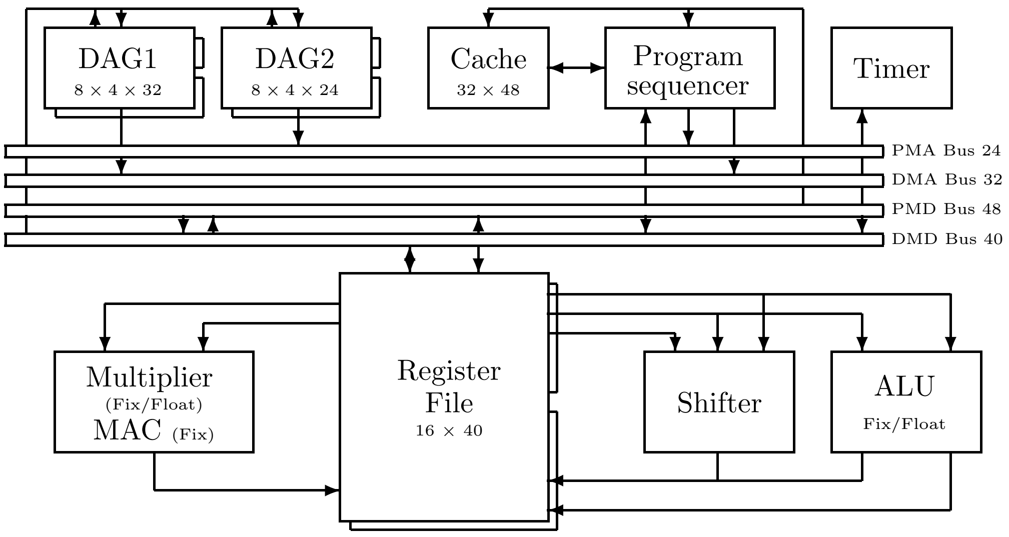


Figure 8: ADSP-21065L core architecture

A register word length of 40 bit is used with floating-point operations. Fixed-point data have 32 bits word length. They are stored **left aligned** in the register file.

Address computations are carried out in **data address generators DAG1, DAG2** to access data memory (DM, addressed by DAG1) and program memory (PM, addressed by DAG2). Both memories are not shown in Fig. 8. All storage operations run via registers since direct memory to memory transfers are not supported. Address computations are executed in parallel to other functional units allowing for up to 2 data transfers per clock cycle in addition to computations. Figure 9 gives an idea about the functioning of DAG1. There are 8 address registers I0 to I7 containing the data addresses. A **base register B** and a **length register L** are associated with each I register. When cyclic buffer addressing is used, then B holds the buffer base address and L the buffer length. Cyclic buffer addresses are updated by a modulo-L arithmetic. Modifications of I registers use offset values in a **modify register M**. Alternatively, an I register can also be modified by a value contained in the instruction word (direct addressing mode).

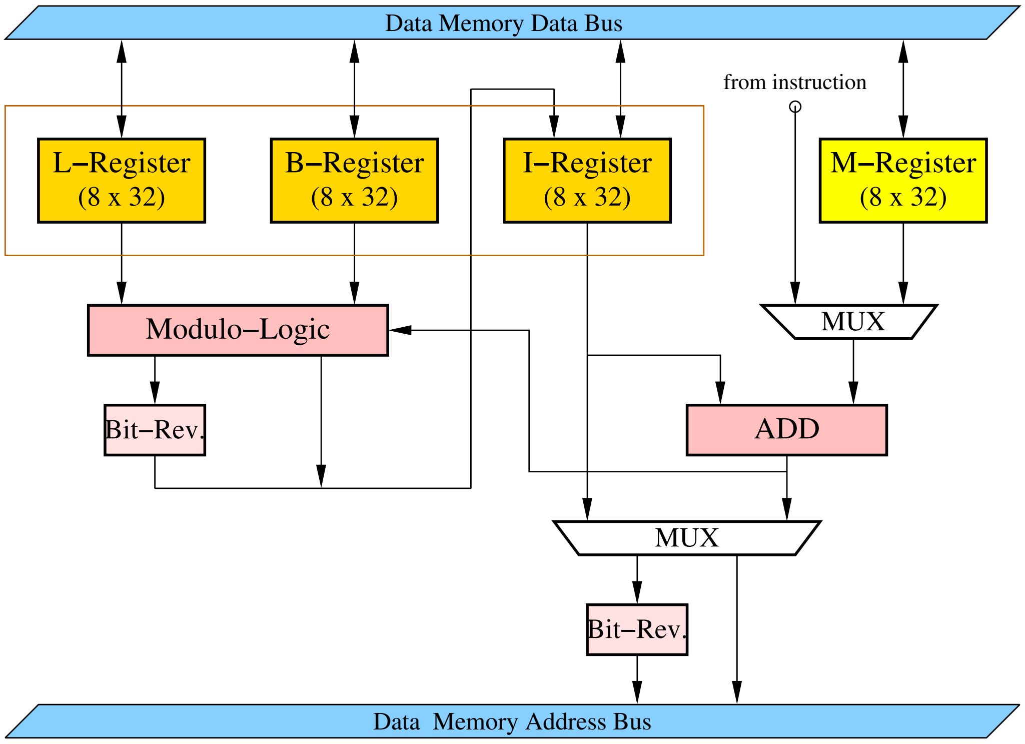


Figure 9: Block diagram of data address generator DAG1

The following two automatic I register modifications are supported:

- pre-modify without update: address to bus = $I + M$, contents of $I$ of address register I is not modified,
- post-modify: address to bus = $I$, $I + M \rightarrow I$ after address has been sent to bus.

Note that according to Fig. 9 only the post-modify mode uses a modulo-operation in case of cyclic buffers! Furthermore, it is always a good programming practice to **initialize the length registers L** with the desired buffer length, or by setting L to zero if a linear addressing mode is needed. You will use cyclic buffers to implement delay lines in the problems. An example of cyclic buffer addressing is illustrated in Fig. 10. Numbers outside the storage boxes numerate memory addresses whereas numbers within the boxes are used to index the sequence of computed addresses. A modulo operation wrap around is indicated by arrows.

A great advantage of cyclic buffers is the fact that address pointers can be changed within the buffer address range only. There are no unwanted pointer moves in memory

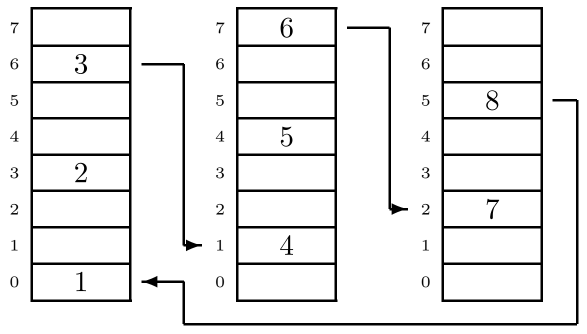


Figure 10: Modulo addressing of cyclic buffers with $B = 0$, $L = 8$, $M = 3$

areas outside the buffer range due to programming errors. In addition, address register I is automatically resetted if a length $L$ buffer is incremented/decremented $L$ times.

For completeness, we mention the **bit-reversed** addressing mode used in fast Fourier transform (FFT) algorithms.

Address generator DAG2 exhibits the same structure as DAG1 with the exception that address registers have a 24 bit word length and addresses are sent to PM program memory bus. In addition, no bit-reversed addressing mode is implemented. In assembler, the address registers are denoted by I8 – I15, B8 – B15 and L8 – L15 (see Fig. 11).

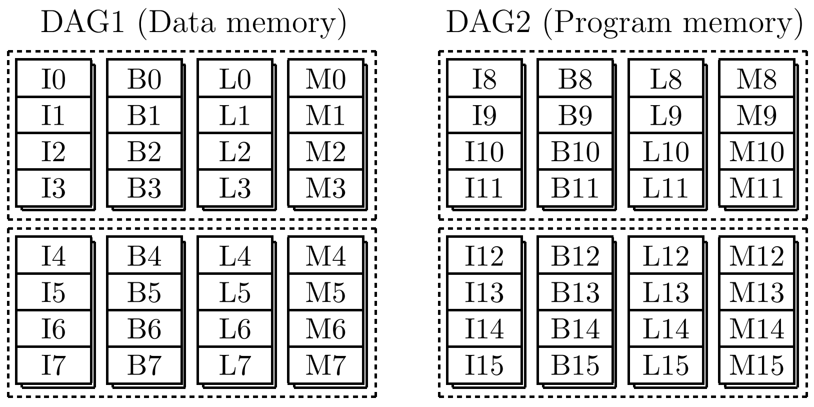


Figure 11: DAG1,2 register notations

Registers B, I, L form a unit and must not be mixed (e.g. B1, I0, L2 is not allowed). M registers can be freely used within a data address generator. Each of the 4 register sets in a dotted box in Fig. 11 has its own shadow register set allowing to switch registers in a single clock cycle (e.g. during interrupt service routines).

The **program sequencer** of the ADSP-21065L core architecture as shown in some detail in Fig. 12 operates in parallel to other functional units in Fig. 8. The 3-stage instruction pipeline uses a fetch address register, a decode address register, and an execute address register (denoted program counter PC).

In case of a linear program flow, the program sequencer calculates the next program memory address simply by incrementing the program counter. However, the ADSP-

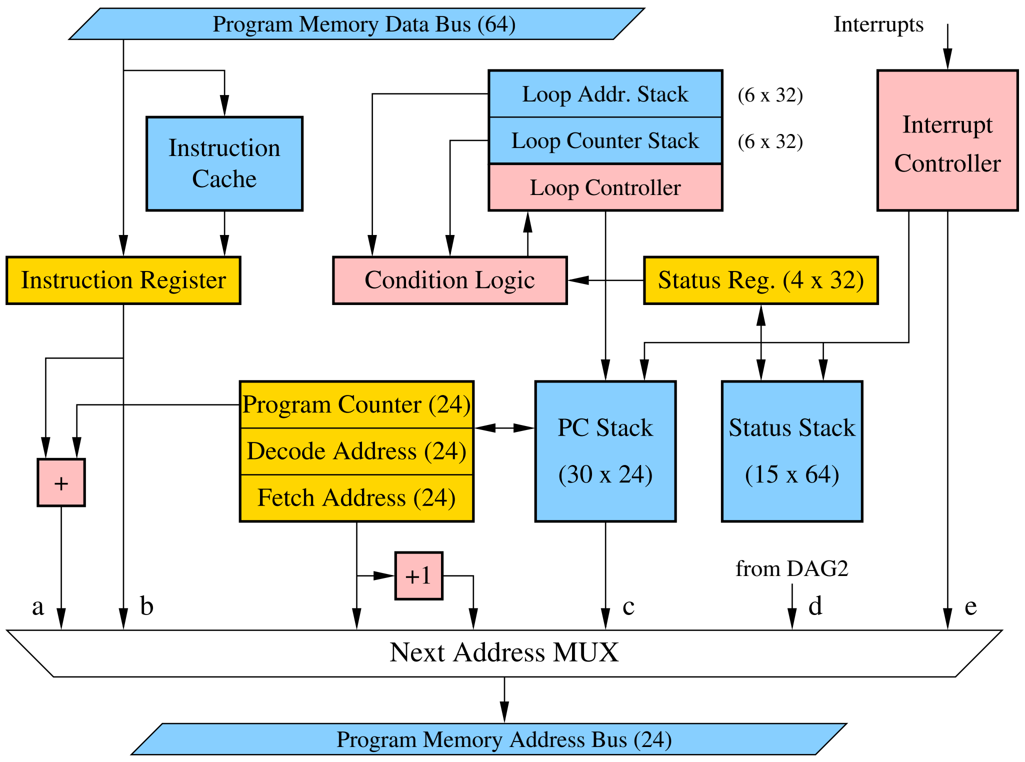


Figure 12: ADSP-21065L program sequencer
(a ... PC-relative address, b ... direct (unconditioned) jump, c ... return address, or first address of a loop, d ... register-indirect jump, e ... interrupt vector
DAG2 ... data address generator 2)

21065L program sequencer can perform more advanced tasks as well:

- automatic handling of subroutine calls and returns,
- interrupt handling (masking, nesting),
- automatic loop management (nested loops, loop counters, termination, stack),
- save and restore of ALU status at interrupts,
- management of conditioned jumps (checking of conditions),
- cache memory operation,
- hold, trap, and idle operations to reduce power consumption (e.g. during wait for interrupt loops).

Caused by the 3-stage instruction pipeline, branches and calls are executed with a delay of 2 clock cycles (delayed branches). However, a non-delayed branch is provided which executes the next 2 instructions following the branch instruction. The difference between these two branch types is illustrated in Table 4, and in Table 5, respectively.

At a **non-delayed branch**, 2 NOP instructions are executed instead of the instructions already in fetch and decode stage. The instruction at jump address $J$ is executed afterwards resulting in a 2 cycle delay. A call instruction at address $N$ initiates a push of instruction at $N+1$ to the PC stack to continue program execution at $N+1$ after return from subroutine.

| cycle | fetch | decode | execute | remark |
|---|---|---|---|---|
| 1 | $N+2$ | $N+1 \rightarrow$ `nop` | $N$ | instruction $N+1$ is suppressed |
| 2 | $J$ | $N+2 \rightarrow$ `nop` | `nop` | instruction $N+2$ is suppressed, $N+1$ pushed on PC stack (for call) |
| 3 | $J+1$ | $J$ | `nop` | |
| 4 | $J+2$ | $J+1$ | $J$ | instruction at jump address $J$ is executed |

Table 4: Instruction pipeline of a **non-delayed branch** ($N$ = branch address, $J$ = jump address

With a **delayed branch**, NOPs are avoided since instructions at $N+1$ and $N+2$ are executed before the instruction at address $J$. Typically, the 2 instructions at $N+1$ and $N+2$ may set register values needed by the subroutine (call at $N$). Returns from subroutines and from interrupts may also be delayed.

| cycle | fetch | decode | execute | remark |
|---|---|---|---|---|
| 1 | $N+2$ | $N+1$ | $N$ | |
| 2 | $J$ | $N+2$ | $N+1$ | instruction $N+3$ pushed on PC stack (for call) |
| 3 | $J+1$ | $J$ | $N+2$ | |
| 4 | $J+2$ | $J+1$ | $J$ | instruction at jump address $J$ is executed |

Table 5: Instruction pipeline of a **delayed branch** ($N$ = branch address, $J$ = jump address)

The program sequencer instruction register stores the address of the next instruction in case of **direct jumps**. It is loaded directly via program memory bus, or from cache memory if both data and instructions must be loaded from program memory. Cache memory only holds those instructions which create a conflict (2 simultaneous accesses of program memory).

There are two further types of jumps: A **PC relative jump** using an offset to the PC as jump address, and an **indirect jump** where the jump address is supplied by an I register of data address generator DAG2.

The remaining program sequencer functional units perform loop processing and interrupt handling. **Loops** are executed automatically using a set of loop counters, a loop stack, a loop address stack, and a loop controller. The loop controller compares the actual address with the loop end address and checks the loop termination condition if the loop end is reached. Loops are terminated if a loop counter value reaches zero, or if a certain ALU status occurs.

The ADSP-21065L **interrupt controller** can handle 32 types of interrupts including hardware interrupts (e.g. serial port interrupt) and software interrupts (e.g. floating-point exception). When an interrupt is granted, the PC is pushed on the PC stack and the processor status is pushed on the status stack (at external interrupts only). Afterwards, the interrupt controller supplies the interrupt vector address to the program memory bus. Execution of an interrupt service routine starts after a pipeline delay of 2 cycles.

Finally, we briefly discuss **cache memory operation** transparent to the programmer. Cache memory can store 32 instructions and automatically collects instructions which create conflicts in program memory accesses. As an example, if digital filter coefficients are stored in program memory (to simultaneously transfer coefficients plus filter variables to registers used in arithmetic operations), then program memory has to be accessed twice per clock cycle: One access to get the coefficient, and a second to read the next instruction. During program execution, cache memory looks for conflicting instructions. If such an instruction is already in cache, the instruction register is loaded from cache freeing program memory from instruction access. Otherwise, the instruction is stored in cache. Therefore, loops with no more than 32 conflicting instructions are processed using cache memory after the first loop execution.

## 5 ADSP-21065L instruction set

The ADSP-21065L instruction set uses an algebraic notation for arithmetic and transfer operations resulting in highly readable assembly programs. In addition, instruction set training is rather simple. You will need only a small subset of the many instructions available. There are the following instruction categories classified as

- computational,
- data move,
- program flow control,
- miscellaneous,
- multi-function instructions.

**Computational instructions** including ALU, multiplier, and shifter operations can be executed conditionally. A conditional instruction checks status flags and executes the instruction if the tested bits are set. A NOP is performed otherwise. Conditional instructions are useful to implement short program jumps without leaving the instruction pipeline. Remember that every jump causes two additional clock cycles due to pipeline processing.

**Data move instructions** are register move, register immediate load, DM and PM read/write operations. Address registers can be accessed and modified in a single clock cycle. Nearly all data transfers may occur simultaneously to compute instructions.

Branch instructions like jumps, subroutine call/return, interrupt return, and loops are summarized as **program flow control instructions**. Some important **miscellaneous instructions** are used to modify processor status, and to carry out no operation (NOP).

Finally, powerful **multi-function instructions** combine arithmetic operations with up to two data transfer. As an example, we can execute an add, multiply, and two data transfers with register post-modify in a single instruction cycle. Sequential program code can be made more compact to reduce code size, and to speed-up program execution. For beginners, however, it is recommended to start with a purely sequential code. There is no need to implement the basic problems of this course with a highly compacted program code.

## 5.1 ALU instructions

The two arithmetic units carry out fixed-point and floating-point instructions via a register file. In assembler, fixed-point and floating-point operations are distinguished by naming registers as R0 (r0) to R15 (r15) in case of fixed-point operations. Floating-point arithmetic requires F0 (f0) to F15 (f15) as register names. Note that R0 and F0 denote the same register hardware! The different names are needed only to inform the assembler about the operand data format.

Table 6 gives a summary of important **fixed-point ALU instructions**. Rn, Rm, Rx, Ry stand for registers R0 ... R15 of the data register file. Only the high 32 bits of a 40 bit registers are used.

| | |
|---|---|
| Rn = Rx + Ry | Rn = Rx − Ry |
| Rn = Rx + 1 | Rn = Rx − 1 |
| Rn = (Rx + Ry)/2 | Rn = −Rx |
| Rn = `ABS` Rx | Rn = `PASS` Rx |
| Rn = `MIN`(Rx,Ry) | Rn = `MAX`(Rx,Ry) |
| Rn = Rx `AND` Ry | Rn = Rx `OR` Ry |
| Rn = Rx `XOR` Ry | Rn = `NOT` Rx |
| Rn = Rx + Ry, Rm = Rx − Ry | |
| Rn = `CLIP` Rx `BY` Ry | |
| `COMP`(Rx,Ry) | |

Table 6: Important fixed-point ALU instructions

Some examples of fixed-point ALU instructions in assembly language:[2]

```
r0 = r0 + 1;      /* increment register r0 */
r2 = abs r2;      /* take absolute value */
r15 = pass r13;   /* r15 = r13, changes ALU status flag accordingly */
r0 = r0 - r0;     /* r0 = 0, for all initial values of r0 */
```

Only a few instructions of Table 6 need some comments. CLIP limits the number range similar to a saturation mode overflow characteristic (see Fig. 3 on page 8). COMP checks Rx $\leq$ Ry and sets the respective status flag. Multi-instruction Rn = Rx + Ry, Rm = Rx − Ry executes an add/subtract in a single clock cycle using the same operands (e.g. `r1 = r2 + r3, r4 = r2 - r3;`).

ALU operations can be executed conditionally, e.g. `if LT r0 = pass r1;`. Condition LT (last ALU result negative) is set by an instruction before the if-statement. If condition LT is true, then the operation in the if-statement is executed. A NOP is performed otherwise.

A listing of some important fixed-point ALU conditions is shown in Table 7. AN, AV, and AZ denote ALU status flags. Using these flags needs some attention. As an example, AN = 1 and AV = 1 does not set the LT condition in case of a two's complement overflow mode since 1 XOR 1 = 0.

| syntax | condition | ALU status flags |
|---|---|---|
| EQ | $= 0$ | AZ = 1 |
| NE | $\neq 0$ | AZ = 0 |
| LT | $< 0$ | AN xor AV = 1 |
| LE | $\leq 0$ | (AN xor AV) or AZ = 1 |
| GT | $> 0$ | (AN xor AV) or AZ = 0 |
| GE | $\geq 0$ | AN xor AV = 0 |
| AV | ALU overflow | AV = 1 |
| NOT AV | no ALU overflow | AV = 0 |

Table 7: ALU conditions and ALU status flags ALU Zero (AZ), ALU Overflow (AV), and ALU negative (AN)

Table 8 is a list of essential floating-point instructions where Fn, Fm, Fx, Fy represent data registers storing 40 bit floating-point operands. Some floating-point instructions in assembly language:

```
f0 = (f1 + f2)/2; /* compute arithmetic mean */
f0 = float r2;    /* convert integer value in r2 to float */
r15 = fix f10;    /* convert float value in f10 to integer */
f0 = f0 - f0;     /* f0 = 0, not valid, if f0 is a NAN! */
                  /* therefore, always use r0 = r0 - r0 to clear r0 */
```

[2]Do not use special characters like ä,Ä,ß to comment your program!

| | |
|---|---|
| Fn = Fx + Fy | Fn = Fx − Fy |
| Fn = (Fx + Fy)/2 | Fn = −Fx |
| Fn = `ABS`(Fx + Fy) | Fn = `ABS`(Fx − Fy) |
| Fn = `ABS` Fx | Fn = `PASS` Fx |
| Fn = `MIN`(Fx,Fy) | Fn = `MAX`(Fx,Fy) |
| Rn = `FIX` Fx | Fn = `FLOAT` Rx |
| Rn = `MANT` Fx | Rn = `LOGB` Fx |
| Fn = Fx + Fy, Fm = Fx − Fy | |
| Fn = `CLIP` Fx `BY` Fy | Fn = `RND` Fx |
| `COMP`(Fx,Fy) | Fn = Fx `COPYSIGN` Fy |

Table 8: Important floating-point ALU instructions

Instruction Rn = `MANT` Fx extracts the mantissa, and Rn = `LOGB` Fx delivers the exponent of a floating-point number in Fx. Fn = Fx `COPYSIGN` Fy replaces the sign bit in Fx by the sign bit in Fy. This instruction can neatly be used to implement a zero-crossing detector, e.g. to convert a triangular signal to a rectangular signal (see problem 6.3). Floating-point instructions can be executed conditionally with similar conditions as in Table 7 but different overflow behavior.

## 5.2 Multiplier instructions

A 40 bit floating-point multiplication is accomplished by

$$\text{Fn} = \text{Fx} * \text{Fy}$$

resulting in a 40 bit floating-point result. As already discussed in section 3, fixed-point multiplications require more attention due to word length change, and due to the difference between fraction and integer multiplication. It is recommended that you use floating-point multiplications to implement the problems in assembly language. It is a very good exercise, however, to solve some of the problems using fixed-point arithmetic.

As opposed to fixed-point arithmetic, the ADSP-21065L offers no instruction to directly program a sum of products $z \leftarrow z \pm xy$. As shown in section 5.6, accumulations can easily be performed with **multi-function instructions**.

## 5.3 Shifter instructions

The ADSP-21065L shifter unit operates on **32 bit fixed-point numbers** and is primarily used for scaling (multiplication with powers of 2 or 1/2), shifting, rotation, and for bit manipulations as shown in Table 9.

Shift operations can be **arithmetic shifts** (sign extension at shifts to the right) or **logical shifts** (with zero filling and no sign extension). The number of shift positions is

| | |
|---|---|
| Rn = `LSHIFT` Rx `BY` Ry | Rn = Rn `OR LSHIFT` Rx `BY` Ry |
| Rn = `ASHIFT` Rx `BY` Ry | Rn = Rn `OR ASHIFT` Rx `BY` Ry |
| Rn = `LSHIFT` Rx `BY dw8` | Rn = Rn `OR LSHIFT` Rx `BY dw8` |
| Rn = `ASHIFT` Rx `BY dw8` | Rn = Rn `OR ASHIFT` Rx `BY dw8` |
| Rn = `ROT` Rx `BY` Ry | Rn = `ROT` Rx `BY dw8` |
| Rn = `BCLR` Rx `BY` Ry | Rn = `BCLR` Rx `BY dw8` |
| Rn = `BSET` Rx `BY` Ry | Rn = `BSET` Rx `BY dw8` |
| Rn = `BTGL` Rx `BY` Ry | Rn = `BTGL` Rx `BY dw8` |
| `BTST` Rx `BY` Ry | `BTST` Rx `BY dw8` |

Table 9: Important shifter instructions (ASHIFT means arithmetic shift, LSHIFT logical shift, `dw8` $= -128 \ldots 127$ for shift immediate, `dw8` $= 0 \ldots 31$ for bit manipulations)

specified in operand register Ry, or by an immediate value `dw8`. In addition, a logical OR with Rn can be applied before saving the result in Rn. Arithmetic shifts correspond to multiplications (shift left), or divisions (shift right) by powers of 2. **Bit manipulations** (Bit clear, set, toggle) change a bit in the 32 bit register part with bit position given in Ry or by `dw8`. **Bit test operation** BTST sets the shifter status flag SZ if the tested bit is 1.
Examples of shifter instructions in assembly language:

```
r0 = ashift r1 by -1;   /* r0 = r1*1/2 */
r0 = ashift r1 by 2;    /* r0 = 4*r1, if no overflow occurs */
btst r0 by 0;           /* LSB of r0 set? */
if SZ r0 = r0 - r0;     /* if set, then clear r0 */
```

## 5.4 Data transfer instructions

Data transfers between registers and memories are listed in Table 10. Only indirect addressing needs some explanations: The order of I and M register in the statements determines whether pre- or post-modify is used to update address register I. If an M registers comes first as in `r0 = dm(m1,i0)`, then a **pre-modify without update** is applied to the address in I (but I is not changed!). Otherwise, a **post-modify with update** is done resulting in a modification of I after its content is used for addressing (e.g. `r0 = dm(i0,m1)`).

**Cyclic buffer addressing** needs buffer base address in base register B and buffer length in register L. This initialization must be done before first buffer usage. In case of **linear buffer addressing**, register L must be initialized with zero to turn off the DAG modulo-addressing unit.

As an example, initialization and usage of a length 3 cyclic buffer looks like the following:[3]

[3]Exercise: Fill in the register contents in the comment statements.

```
b0 = b_start;      /* b0 = buffer start address, also sets i0 = b0
                      Note: b_start is a symbolic address resolved by
                      the linker, do not use any physical addresses like
                      0x2000, etc. */
l0 = 3;            /* set length register to buffer length */

                   /* as an exercise fill in the register values */
dm(i0,1) = 1;      /* i0 =                */

dm(i0,1) = 2;      /* i0 =                */

dm(i0,1) = 3;      /* i0 =                */

m3 = 2;
r0 = dm(m3,i0);    /* i0 =           , r0 =             */

r1 = dm(i0,m3);    /* i0 =           , r1 =             */
```

Data transfers except register and memory load with constants can be combined with compute instructions. A conditional data transfer is not supported with exception of the `pass` instruction. This instruction passes data through the ALU and sets status flags according to the operand value.
Example: `if EQ r0 = pass r1;` but `if EQ r0 = r1;` is not allowed!

## 5.5 Program flow control

You will need this group of instructions to implement branches (jumps), subroutine calls, and program loops. All of these instructions can be executed conditionally. There are direct jumps and calls as well as indirect ones. Jumps and calls can be delayed or non-delayed branches as discussed on page 18.

The syntactical statement of a **direct jump** is `jump` *label*; where *label* denotes the jump address (in symbolic form, typically). With a conditional jump, an *if-else-endif* code segment can be programmed as follows:

```
   ...
   comp(f0,f1);
   if LT jump neg;
   ...             /* section for case f0 >= f1 */
   jump done;
neg:
   ...             /* section for case f0 < f1 */
done:
   ...
```

**Subroutine calls** are initiated by `call` *subroutine*; with subroutine name *subroutine*. Note that there are no parameters passed in this call statement. All variables needed

| description | syntax | examples |
|---|---|---|
| register load with constant | REG $= const$ | `r10 = 0;` |
| | *const* is a decimal number, | `f1 = 3.14;` |
| | hexadecimal number, | `r0 = 0x7fffffff;` |
| | or symbol | `m0 = OFFSET;` |
| register to register | REG $=$ REG | `r1 = r2;` |
| transfer | | `i2 = r0;` |
| direct addressing | REG $=$ DM($const$) | `r0 = dm(ADC);` |
| (DM and PM) | DM($const$) $=$ REG | `dm(DAC) = r2;` |
| | | `r1 = pm(FILT_COEFF);` |
| indirect addressing | REG $=$ DM($I_d, M_d$) | `r0 = dm(i0,m1);` |
| | REG $=$ DM($I_d, const6$) | `r0 = dm(i0,-1);` |
| | DM($I_d, M_d$) $=$ REG | `dm(i6,m4) = i0;` |
| | DM($I_d, const6$) $=$ REG | `dm(i6,0) = r0;` |
| | DM($I_d, M_d$) $= const$ | `dm(i0,m0) = 0x8000;` |
| | REG $=$ PM($I_p, M_p$) | `r11 = pm(i9,m9);` |
| | PM($I_p, M_p$) $=$ REG | `pm(i10,m8) = f9;` |
| | REG $=$ PM($I_p, const6$) | `r11 = pm(i14,2);` |
| | PM($I_p, const6$) $=$ REG | `pm(i8,-2) = f9;` |
| | REG $=$ DM($M_d, I_d$) | `r0 = dm(m1,i0);` |
| | REG $=$ DM($const6, I_d$) | `r0 = dm(-1,i0);` |
| | DM($M_d, I_d$) $=$ REG | `dm(m4,i6) = i0;` |
| | DM($const6, I_d$) $=$ REG | `dm(0,i6) = r0;` |
| | DM($M_d, I_d$) $= const$ | `dm(m0,i0) = 0x8000;` |
| | REG $=$ PM($M_p, I_p$) | `r11 = pm(m9,i9);` |
| | PM($M_p, I_p$) $=$ REG | `pm(m8,i10) = f9;` |
| | REG $=$ PM($const6, I_p$) | `r11 = pm(2,i14);` |
| | PM($const6, I_p$) $=$ REG | `pm(-2,i8) = f9;` |

Table 10: Data transfers (REG = all registers, DM = data memory, PM = program memory, *const* at direct addressing can be a symbolic address (variable name) too, $const6 = -64 \ldots 63$).
$I_d \in \{$`i0, i1, i2, i3, i4, i5, i6, i7`$\}$
$M_d \in \{$`m0, m1, m2, m3, m4, m5, m6, m7`$\}$
$I_p \in \{$`i8, i9, i10, i11, i12, i13, i14, i15`$\}$
$M_p \in \{$`m8, m9, m10, m11, m12, m13, m14, m15`$\}$

by a subroutine must be set before the call statement. This is different to programming in C where parameters may be supplied as function arguments. Subroutine returns must be explicitly stated by the `rts` instruction.

**Program loops** are important code elements. They are processed automatically by the ADSP-21065L processor when using the DO UNTIL statement. It is of course possible to implement loops with jumps and ALU instructions. But this costs extra clock cycles and is tedious in case of nested loops. Normally, a DO UNTIL loop is initialized by setting a loop counter value followed by the statement `do` *endlabel* `until` *condition*`;`. The last address in the loop code segment is specified by *endlabel*. Loop termination is determined by *condition* which typically is LCE (loop counter expired). Because termination conditions are checked at the loop end, DO UNTIL loops are processed at least once. The following example illustrates a typical counter-based loop:

```
   lcntr = 10, do DSP until LCE;   /* loop setup */
      f0  = f0 + ...;              /* first instruction of loop */
      f1 = ...;
      ...
DSP:  dm(i0,m0) = ...;             /* last instruction of loop */
```

It is possible to use arithmetic conditions to terminate DO UNTIL loops by checking ALU status flags. It is a good programming practice, however, to do the best to avoid endless loops in such cases. There are some restrictions in using loops due to pipeline processing. As an example, no program branches must be present in the last 3 instructions of a loop. If you are unsure, then include 3 NOPs at the loop end (and don't forget to remove them when not needed).

## 5.6 Multi-function instructions

Multi-function instructions enable a reasonable increase in data throughput despite the rather low clock frequency of the ADSP-21065L. The simplest form of this parallel processing feature is the combination of compute instructions with data transfer instructions. Instructions of ALU or multiplier or shifter can be combined with transfer instructions listed in Table 10. Exceptions are data moves with constants. Additionally, there are the following two multi-function instructions showing a maximum of parallelism:

$$\textit{compute},\ \mathrm{DREG1}\ =\ \mathrm{DM}(I_d, M_d),\ \mathrm{DREG2}\ =\ \mathrm{PM}(I_p, M_p);$$

$$\textit{compute},\ \mathrm{DM}(I_d, M_d) =\ \mathrm{DREG1},\ \mathrm{PM}(I_p, M_p) =\ \mathrm{DREG2};$$

where *compute* is any compute statement, and DREG1,2 are any registers R0...R15.

A further increase in efficiency is obtained by parallel operations in compute instructions. Up to 3 arithmetic operations can be executed in a single instruction cycle. In this course, you will need parallel floating-point instructions as listed in Table 11.

Unfortunately, there are some restrictions in using registers in multi-function instructions. Only the 4 registers specified in Table 11 are allowed at the respective operand positions. As an example, F0, F1, F2, F3 can be used as left multiplication operand.

| | |
|---|---|
| Fm = F0–3 ∗ F4–7, | Fa = F8–11 + F12–15 |
| Fm = F0–3 ∗ F4–7, | Fa = F8–11 − F12–15 |
| Fm = F0–3 ∗ F4–7, | Fa = (F8–11 + F12-15)/2 |
| Fm = F0–3 ∗ F4–7, | Fa = ABS F8–11 |
| Fm = F0–3 ∗ F4–7, | Fa = MAX(F8–11,F12–15) |
| Fm = F0–3 ∗ F4–7, | Fa = MIN(F8–11,F12–15) |
| Fm = F0–3 ∗ F4–7, | Fa = FIX F8–11 BY R12–15 |
| Fm = F0–3 ∗ F4–7, | Fa = FLOAT R8–11 BY R12–15 |
| Fm = F0–3 ∗ F4–7, | Fa = F8–11 + F12–15, |
| | Fs = F8–11 − F12–15 |

Table 11: Multi-function instructions of the ADSP-21065L floating-point unit

Consequently, multi-function instructions require an anticipatory program development in regard to register allocation. Therefore, it is best to start with a purely sequential version of the program segment followed by some trials to successively apply multi-function instructions.

We will conclude with some examples of multi-function instructions in assembly language:

```
f0 = f1 * f1, i0 = i1;
f0 = f2 + f9, dm(i5,0) = f2;
f0 = f1 * f4, f2 = f9 + f13, f3 = f5;
f0 = f1 * f4, f2 = f9 + f13, f3 = dm(i0,-1);
f0 = f1 * f4, f2 = f9 + f13, f3 = dm(i0,m1), f5 = pm(i8,m9);
f1 = f0 * f7, f8 = abs f9, dm(i2,m2) = f2;
f0 = f1 * f4, f2 = f8+f12, f3 = f8-f12, f9 = dm(i0,m1), f5 = pm(i8,m9);
if EQ r0 = ashift r1 by -2, f3 = dm(i0,m0);
if LT f0 = -f0, f1 = f0;
```

## 5.7 Assembler instructions

Programming a DSP in assembly language requires special statements to control the compilation process, and to initialize variables. You will need only a few of these assembler instructions.

- A file can be imported with `#include` by adding its content to the source code during compilation:

  ```
  #include "labor.h"
  ```

- **Definition of constants and macros:**

  The `#define` statement can be used to create constants and text macros:

```
#define N      100
#define M      (N) + 4
#define PI     3.14159265358979323846
#define EPS    1e-10

#define ADD(z,x,y)   z = x + y

#define COPY(src,dest) \
r0 = dm(src); \
dm(dest) = r0
```

A text macro as defined above is called e.g. by `ADD(f0,f1,f2);`.

- **Definition and initialization of variables:**

  The `.var` statement creates variables in memory segments including an optional initialization. Memory segments are defined in the target system memory map (see Table 1 on page 13). Segments start with `.segment/dm  dm_data;` (segments in DM), or with `.segment/pm  pm_data;` (data in PM). Segments are terminated by `.endseg;` as shown in the following examples.

```
.segment/dm   dm_data;
.var x;
.var y = 0.5;
.var z_vec[3] = 1.0, 2.0, 3.0;
.var coeffs[N] = "fir_coeffs.dat";
.endseg;

.segment/pm   pm_data;
.var a;
.var angles[3] = 0.0, PI/2, PI;
.endseg;
```

- **Initialization of buffer pointers (address registers):**

  Indirect addressing of buffers by data address generator registers requires initialization of buffer base register <u>and</u> buffer length register:

```
b0 = z_vec;      /* set base register b0 and pointer
                    i0 to begin of buffer z_vec (in DM) */
l0 = @z_vec;     /* set length register l0 to buffer length
                    (for cyclic buffers) */
b8 = coeffs;     /* b8 = i8 -> coeffs (buffer in PM) */
l8 = 0;          /* use zero length for linear buffers */
```

# 6 Problems

With the following problems, we offer a representative set of basic DSP algorithms suitable for programming with an ADSP-21065L processor. As already mentioned in the introduction, the difficulty increases from problem to problem. Thus, we recommend that you select the problems in their given order. Furthermore, it may be convenient to use a MATLAB simulation of a problem before starting DSP assembler programming. This may serve as a reference and could ease debugging of your code. Of course, the final solution should be a functioning assembly program.

## 6.1 Sampling and interpolation of discrete-time signals

One method to sample discrete-time signals is to multiply signal $x[n]$ with unit impulses ($\delta$–pulse $p[n]$) with period $N$:

$$y[n] = x[n]p[n] = x[n] \sum_{k=-\infty}^{\infty} \delta[n - kN]. \tag{2}$$

This multiplication (modulation) results in zeroing every sample except those at $n = kN$. Therefore, such an operation may be called sampling similar to the process of sampling an analog (time-continuous) signal. However, there are no zeros inserted at analog signal sampling.

The spectrum of such a sampled signal is obtained with

$$p[n] \Leftrightarrow P(e^{j\theta}) = \frac{2\pi}{N} \sum_{k=-\infty}^{\infty} \delta(e^{j(\theta - \frac{2\pi}{N}k)}) \tag{3}$$

in combination with the Fourier transform applied to Eq. 2:

$$Y(e^{j\theta}) = \frac{1}{N} \sum_{k=-\infty}^{\infty} X(e^{j(\theta - \frac{2\pi}{N}k)}). \tag{4}$$

An alternative to zeroing of $N-1$ of $N$ samples is to hold samples $x[n]$ at $n = kN$ in consecutive intervals of length $N$. This operation results in a staircase-like signal similar to the staircase analog output signal of a conventional digital-to-analog converter.

**Problem 6.1:** Modify program `muster.asm` to implement discrete-time sampling/interpolation by inserting zeros, and by applying a hold operation. Try to invent different methods of programming the modulo-N operation (e.g. by usage of a data address generator). Observe and explain aliasing, and the influence of the hold operation using the spectrum analyzer.

## 6.2 Quantizer characteristic

In this problem you will study quantization effects based on quantization characteristics of two's complement numbers as shown in Fig. 2, and in Fig. 3, respectively.

**Problem 6.2:** Write an assembly program to quantize 16 bit ADC samples to a word length $0 < N < 16$. Watch (and measure) the level of quantization noise as a function of $N$ with the spectrum analyzer. In a second experiment include overflow handling in your program and test your solution with a sinusoidal input signal. Limit the quantizer input range (e.g. to $[-0.5, 0.5)$) to avoid an ADC overload when testing the overflow characteristic.

## 6.3 Digital function generator

Analog function generators create sinusoidal signals by transforming a triangular signal by a nonlinear characteristic. This principle can be used with discrete-time signals too. The required nonlinear transform $y = \sin x$ is approximated by the Taylor series expansion

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \ldots$$

where $x$ represents the triangular input signal.

**Problem 6.3:** First, develop an assembly program to create a triangular signal by incrementing/decrementing a register variable. Select the step size to obtain a low-frequency triangular signal with most of its harmonics in the frequency range up to half the sampling frequency ($f_s$ = 24kHz, typically). Note that the triangular signal must be precisely symmetric. In addition, the signal amplitude must be set accurately to insure that the sine-characteristic is driven to its full range ($\pm\pi/2$). A correct solution does not show any harmonics of the resulting sinusoidal signal when observed with the spectrum analyzer.

Second, extend your program to create a rectangular signal from the triangular input signal. You may want to use the COPYSIGN instruction for that purpose.

If you are very eager, then a nice programming task would be the generation of a sweeping sinusoidal signal.

## 6.4 Non-recursive digital filters

The previous problems mainly use ALU, multiplier, and shifter instructions of the DSP. Digital filters, however, additionally need signal delays. These delays can be efficiently implemented with cyclic buffers. Therefore, in case of a delay line we do not need to move signal samples in memory. We simply change pointers (address registers) instead.

The standard implementation of an FIR filter uses a transversal structure (see page 191 of the course book). We will start with the most basic form of a transversal filter where all coefficients are zero except the first and the last one (see Fig. 13). Such a filter is known as a comb filter due to its special transfer function with equidistant notches.

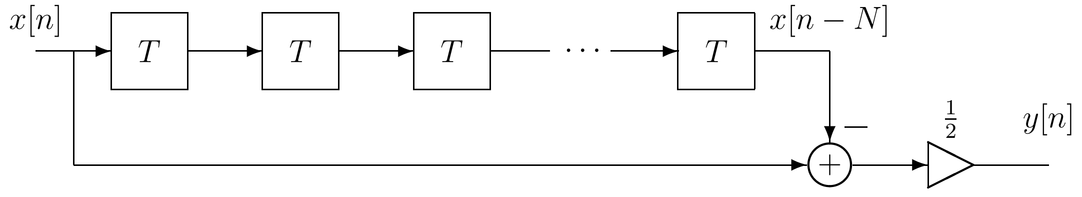


Figure 13: Digital comb filter using a delay line of $N$ delay elements

The comb filter impulse response is immediately obtained from Fig. 13:

$$h[n] = \frac{1}{2}(\delta[n] - \delta[n-N]). \tag{5}$$

From Eq. 5 we get the comb filter frequency response

$$H(e^{j\theta}) = e^{-j(\frac{N}{2}\theta - \frac{\pi}{2})} \sin\left(\frac{N}{2}\theta\right). \tag{6}$$

According to Eq. 6, the comb filter is a linear phase filter (apart from the phase jumps at multiples of $\frac{2\pi}{N}$, i.e. at the notch frequencies).

**Problem 6.4:** Write a DSP program of the comb filter using a cyclic delay line buffer. Use the data address generator (DAG) registers for modulo-($N$+1) addressing. As mentioned above, there is no need to shift samples in memory. Only an index register (DAG I register) must be modified. Check the notch frequencies of your solution with sinusoidal input signals. Alternatively, you can use the spectrum analyzer to plot the transfer function. Do you observe the correct number of notch frequencies for a given $N$?

A second very simple FIR filter is a linear averager (course book on pages 172–173). It is a length $N$ FIR filter with all coefficients set to $1/N$. This signal processing device computes the arithmetic average of an advancing length $N$ signal block. The input/output relation is thus given by

$$y[n] = \frac{1}{N}\sum_{k=0}^{N-1} x[n-k] \tag{7}$$

which leads to the following impulse response, and frequency response, respectively:

$$h[n] = \begin{cases} \frac{1}{N} & 0 \leq n \leq N-1 \\ 0 & n < 0 \text{ and } n \geq N \end{cases} \tag{8}$$

$$H(e^{j\theta}) = \frac{1}{N} e^{-j\frac{N-1}{2}\theta} \frac{\sin\frac{N}{2}\theta}{\sin\frac{\theta}{2}}. \tag{9}$$

> **Problem 6.5:** Extend your comb filter program to implement the linear averager. Check the transfer function with the spectrum analyzer using different values of $N$. Is the number of zeros correct?

Finally, you can apply the following code segment which uses an ADSP-21065L multi-function instruction to efficiently program Eq. 7. We assume that $\frac{1}{N}$ is stored in data register `f4`, and DAG index register `i1` points to $x[n-N+1]$ (end of delay line). The delay line is stored as a length $N$ cyclic buffer in DM.

```
    r12 = r12 - r12, f2 = dm(i1,1);
    f10 = f2 * f4, f2 = dm(i1,1);
    LCNTR = N-2, DO fir_sum UNTIL LCE;
fir_sum: f10 = f2 * f4, f12 = f10 + f12, f2 = dm(i1,1);
    f10 = f2 * f4, f12 = f10 + f12;
    f12 = f10 + f12;
```

The multi-function instruction in the loop is executed in a single clock cycle. The instructions before and after the loop are needed to implement the sum-of-products operation without a multiply-accumulate instruction. After processing this code segment, $y[n]$ is stored in data register `f12`, and index register `i1` again points to $x[n-N+1]$ (due to modulo-N addressing of the cyclic buffer).

> **Problem 6.6:** If you are familiar with MATLAB and a little bit enthusiastic about digital filters, then you should design an FIR filter with MATLAB (using functions `fir1.m` or `firpm.m`). Store filter coefficients in PM and extend the code segment given above to include indirect addressing of these coefficients. As an example, append `f4 = pm(i8,m8)` to the instructions containing multiplications.

## 6.5 Recursive digital filters

With the previous problems you are prepared the implement recursive filters as an alternative to FIR filters. We will start with a recursive version of the linear averager:

$$y[n] = \frac{1}{N}\sum_{k=0}^{N-1} x[n-k] = y[n-1] + \frac{1}{N}(x[n] - x[n-N]). \tag{10}$$

According to Eq. 10, the recursive averager is composed of a comb filter (see Fig. 13) in cascade with an accumulator. Therefore, there is only a slight modification of your comb filter program to get a recursive averager program. Note that the recursive averager has a finite length impulse response. Normally, recursive filters have impulse responses with infinite lengths (IIR filters).

**Problem 6.7:** Modify your comb filter program to implement the recursive averager Eq. 10. Compare code size (and execution time) with the non-recursive averager.

In order to avoid stability problems due to finite word length effects (see chapter A.7 on pages 192–200 in the course book), higher order recursive filters are split into a cascade, or in a parallel connection of first and second order filter blocks. An example of a second order filter block is shown in Fig. 14. The difference equation according to

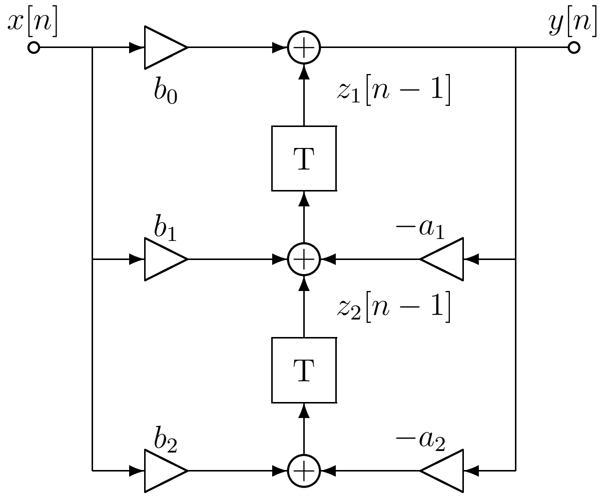


Figure 14: Second order recursive digital filter block

Fig. 14 is given by

$$y[n] = -a_1 y[n-1] - a_2 y[n-2] + b_0 x[n] + b_1 x[n-1] + b_2 x[n-2]. \tag{11}$$

The digital filter frequency response $H(e^{j\theta})$ and transfer function $H(z)$ are immediately obtained from Eq. 11:

$$H(e^{j\theta}) = \frac{b_0 + b_1 e^{-j\theta} + b_2 e^{-j2\theta}}{1 + a_1 e^{-j\theta} + a_2 e^{-j2\theta}}\,, \tag{12}$$

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}} = \frac{b_0 z^2 + b_1 z + b_2}{z^2 + a_1 z + a_2}\,. \tag{13}$$

Since $H(z)$ is a rational function in $z$, stability of a causal system is guaranteed if all poles (zeros of denominator polynomial) are lying within the unit circle in the complex z-plane. Thus, coefficients $a_1$, $a_2$ are restricted to the triangular area in Fig. 15. You may prove this by evaluating the quadratic expression in the denominator of Eq. 13 with the constraint $|z| \le 1$.

If coefficient pairs in Fig. 15 are lying within the hatched area, then poles of $H(z)$ are conjugate complex. Coefficients lying on the parabola yield twofold real poles. The remaining coefficient locations result in two different real poles.

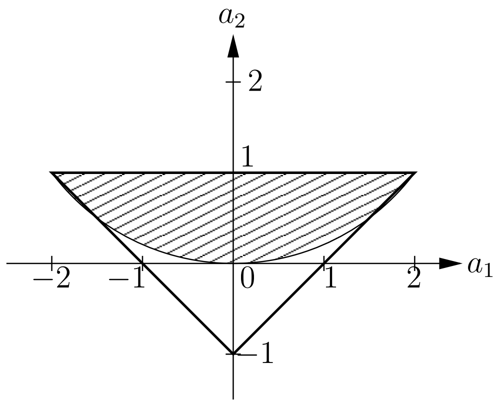


Figure 15: Value range of $a_1$, $a_2$ of a stable and causal second order digital filter

> **Problem 6.8:** Develop an ADSP-21065L program of the digital filter shown in Fig. 14. Use different sets of filter coefficients and observe poles and zeros of $H(z)$ using the spectrum analyzer. Build an oscillator by exact placement of poles on the unit circle in the z-plane. Set the input signal to zero in that case and start the filter by setting initial conditions in the two delay elements. These delay elements should be memory locations in DM.
>
> You can extend this problem by designing a higher order filter with MATLAB and implement this filter in cascade form.

An alternative second order filter is shown in Fig. 16. There are some advantages as compared to the filter block in Fig. 14. First, all arithmetic operations can be carried out as a sum-of-products (MAC) operation. This is especially useful if a fixed-point DSP is employed. Second, delay elements can be organized as cyclic buffers. However, 4 delay elements have to be used. But this is no disadvantage in case of higher order filters in cascade form!

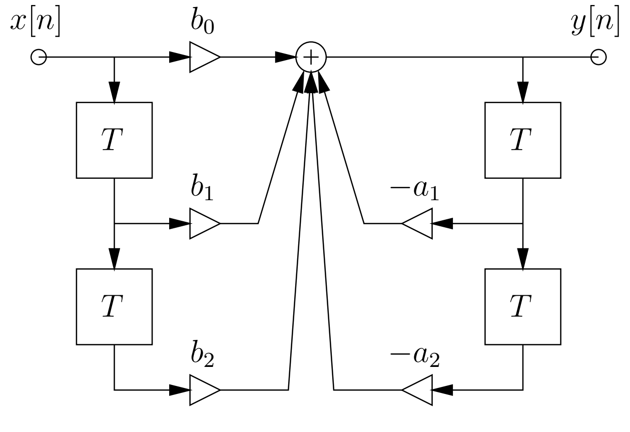


Figure 16: Alternative second order recursive digital filter block

**Problem 6.9:** Implement the second order filter of Fig. 16 with cyclic delay line buffers. If you are very motivated in DSP programming, then use multi-instructions to obtain an optimized code segment of the filter block. Compare your program with that of the preceding problem.

# A Assembly language prototype program

All problems of this course can be solved with program `muster.asm` as a starting point. It is designed to process stereo or monaural scalar input data with basic signal processing algorithms (no multirate processing, or block processing). The source code is as follows:

```
/* prototyping program MUSTER.ASM for ADSP-21065L EZ-KIT
G. Doblinger, 8-2000
*/
#include "labor.h"
/* define variables in internal data memory space */
.segment /dm   dm_data;
.var save_astat;       /* storage place to save ASTAT during interrupts */
.endseg;
/* define variables in internal program memory space */
.segment /pm   pm_data;
.endseg;
/* define variables in external SDRAM */
.segment /dm   sdram_data;
.endseg;
.segment /pm pm_code;
/* ------------------------------------------------------------------ */
/* init. all variables used (called before wait-for-interrupt loop) */
Init_variables:
   bit set mode1 ALUSAT;           /* enable ALU saturation (for FIX) */
   rts;
Init_variables.end:
/* ------------------------------------------------------------------ */
/* subroutine for stereo codec interrupt service routine
   (called every sampling interval)
   sampling rate may be changed in file define_F_sample.h
   Note: codec signal samples are 2-complement data left aligned
         in 32 bit word (1.31 fractions)
*/
DSP_program:
   dm(save_astat) = astat;         /* save ASTAT, not saved automatically
                                      during interrupts */
   r0 = dm(Left_Channel_In);       /* left channel input sample */
   r1 = dm(Right_Channel_In);      /* right channel input sample */
   /* convert samples to floats in [-1.0 1.0) */
   r2 = -31;
   f0 = float r0 by r2;
   f1 = float r1 by r2;
   /* processing of samples using floating-point arithmetic */
   call Process_samples;
   /* convert floats back to 1.31 fractions */
   r2 = 31;
   r0 = fix f0 by r2;
   r1 = fix f1 by r2;
   dm(Left_Channel_Out) = r0;      /* send result to left output channel */
   dm(Right_Channel_Out) = r1;     /* send result to right output channel */

   rts (db);
      astat = dm(save_astat);      /* restore ASTAT register */
           nop;
DSP_program.end:

/* ------------------------------------------------------------------- */
/* user DSP routine called every sampling period
   NOTE: on input f0, f1 contains left and right sample in FLOAT format
         on output f0, f1 must contain processed samples in FLOAT format */
Process_samples:
   f0 = f0 + f1;
   f1 = f0;
   rts;
Process_samples.end:

/* ------------------------------------------------------------------- */
/* flag3 button may be used to change some programm parameters */
Wait_flag3:
   /* wait for flag 3 button press and release */
   if flag3_in jump Wait_flag3;    /* wait for button press   */
release:
   if not flag3_in jump release;   /* wait for button release */
   /* program code for flag 3 handling */
          rts;
Wait_flag3.end:
/* ------------------------------------------------------------------- */
/* IRQ1 interrupt service routine
   (IRQ1 button may be used to change same program parameters) */
IRQ1_isr:
   bit set mode1 SRRFH;            /* enable background register file */
   nop;
   bit clr ustat1 0x3D;            /* turn off Flag5 LED */
   bit set ustat1 0x02;
   bit set ustat1 0x3E;            /* turn on Flag4 LED */
   bit clr ustat1 0x01;
   dm(IOSTAT)=ustat1;
   /* program code for IRQ1 handling */
          rti(db);
      bit clr mode1 SRRFH;         /* switch back to primary register set */
      nop;
IRQ1_isr.end:
/* ------------------------------------------------------------------- */
/* IRQ2 interrupt service routine
   (IRQ1 button may be used to change same program parameters) */
IRQ2_isr:
   bit set mode1 SRRFH;            /* enable background register file */
   nop;
   bit clr ustat1 0x3E;            /* turn off Flag4 LED */
   bit set ustat1 0x01;
   bit set ustat1 0x3D;            /* turn on Flag5 LED */
   bit clr ustat1 0x02;
   dm(IOSTAT)=ustat1;
   /* program code for IRQ2 handling */
          rti(db);
```

```
    bit clr mode1 SRRFH;         /* switch back to primary register set */
    nop;
IRQ2_isr.end:
.endseg;
```

There are several sections marked by `/* ---- */` where you may insert your code segments. Declaration of variables must be done first, like

```
.var x;
.var x = 0.12345;
.var fir_delay[100];
.var i_vec[3] = 1, 2, 3;
.var f_vec[3] = 1.0, 2.0, 3.0;
.var y[N] = "y.dat";           /* initialize y with file data */
```

Variables can be placed in data memory (segment `dm_data`), and in program memory (segment `pm_data`). In addition, there is a larger SDRAM memory area (segment `sdram_data`) which, however, is not needed for solving the problems of this course. A variable name is a symbolic address. It can be referenced e.g. by `dm(x)`, or indirectly with DAG index registers, like `b0 = f_vec`. Remember that setting of `b0` sets the respective index register (`i0`) too.

In section with label `Init_variables`, all declared (and not yet initialized) variables must be initialized in order to start your program from a well defined state. Don't forget to set the length register of any buffer you may want to use! As an example, we initialize a cyclic buffer named `fir_delay` by the following code segment:

```
   b1 = fir_delay;            /* set pointer to buffer begin address */
   l1 = @fir_delay;           /* set length register to buffer length */
   m1 = 1;                    /* set modify register value */
   LCNTR = @fir_delay, DO zero_delay UNTIL LCE;
zero_delay: dm(i1,m1) = 0;
```

The remaining sections are provided to insert your application specific statements. Your DSP program is called once every sampling interval by an interrupt service routine (not visible in `muster.asm`). In section with label `DSP_program`, stereo ADC samples are converted from two's complement numbers to floating-point numbers in range $[-1, 1)$. After calling subroutine `Process_samples` the samples processed by your DSP program are converted back to two's complement numbers to be sent to the stereo DAC.

You must fill in your DSP program in subroutine `Process_samples`. Input signal samples are stored as floating-point data in data registers `f0`, `f1`. After processing these samples you must put the results (output signal) as floating-point values in the same registers. In order to avoid ambiguities with input and output cabling, we add up the input samples and put the output sample in both output registers. The prototype program acts as a feed through system if you do not insert any code in subroutine `Process_samples`.

The clock cycle time of the ADSP-21065L is 16.7 ns. Therefore, a maximum of 2495 instructions (including interrupted handling) per sampling interval is available at 24000 Hz sampling frequency. This is more than sufficient to solve all problems of this course.

However, due to programming errors like endless loops you can easily get timing errors. Input and output samples will be lost in such a case.

The sections after subroutine `Process_samples` are provided to activate switches, and to control LEDs on the DSP board. There is no need to use them in the course problems.

There are a lot of accompanying files in the directory where program `muster.asm` is stored. Do not change any of these files! The only modification you may want to make concerns the sampling frequency. It can be changed in header file `define_fsample.h` by selecting one of the following values: 8000, 9600, 16000, 24000, 32000, or 48000 Hz. Default value is 24000 Hz.

## B C programming of the ADSP-21065L

All problems of this course must be programmed in assembly language since you should get some insight into the troubles of real DSP programming. However, if you would like to compare assembler and C, then some people of your group may solve problems using C. For that reason, we supply the C prototype program `main.c`. Any variable initialization must be included before the `while(1)` loop. This loop is interrupted every sampling interval to process the ADC input samples. Your DSP program must be inserted in the interrupt service routine `Sport1_tx_isr_function()`. The source code of `main.c` looks like this:

```
/*  Main C Program Shell for the 21065L EZ-KITL
G. Doblinger, 8-2002
*/
#include "main.h"
static float xl, xr, yl, yr;  /* just to see values in debugger */

void main()
{
   /* initial DSP and stereo codec */
   call_init_functions();
   /* wait-for-interrupt loop */
   while (1)
      asm("idle;");
}
/* ------------------------------------------------------------------ */
void Sport1_tx_isr_function()
/* this function is called once every sampling interval */
{
   get_audio_data();      /* get new samples from ADC, stored as ints */
   xl = ((float) Left_Channel_In) * ADC_SCALE;
   xr = ((float) Right_Channel_In) * ADC_SCALE;
   yl = xl + xr;          /* simple data processing */
   yr = yl;
   Left_Channel_Out = (int) (yl * DAC_SCALE);
   Right_Channel_Out = (int) (yr * DAC_SCALE);
   put_audio_data();      /* send processed samples to DAC */
}
```

Note that the ADC samples are stored as 32 bit floating-point numbers (within $[-1, 1)$) in variables `xl`, `xr`. As opposed to standard C, all floating-point variables use a 32 bit word length. 64 bit variables are supported too but processing is terribly slow.

There are several other example programs (FIR filters, IIR filters, and FFT applications) in the course directory which may serve as a reference. In order to speed up C program execution, **the optimizer switch of the C compiler must be selected**.

# C Software development tools

The integrated software development tool VDSP++ includes an editor, an assembler, a C compiler, a simulator, and a debugger. The simulator can be used to emulate the DSP and does not need a DSP board. It is a good tool to evaluate instructions and code segments. Input/output may be carried out via buffers. The debugger, however, communicates with the DSP board via a serial interface.

In the sequel, we will present a short description of the main steps of the design phase. You will learn the design procedure very fast since you will often repeat the individual steps during debugging of your code. In addition, each group will get an individual briefing at the first course.

Before beginning a programming session, you should copy the example directory "Labor" to another directory, e.g. "Problem-1". Files in "Labor" should not be modified since they serve as a backup copy if something goes wrong. Create a directory for every problem you may want to solve and leave it unchanged after your solution works. Only experienced programmers may use a single file for all problems and insert `#if` - `#endif` blocks to select certain code segments.

When you start VDSP++ (always use version 4.5 with the ADSP-21065L board), a session from the menu bar must be selected first. Choose EZKIT serial debugger from the list. Press the reset button on the board when asked and wait until communication between PC and DSP board is established. Afterwards, you can open a project from the menu bar by selecting "Project - Open". Select a directory and click to file `labor.dpj` (or `labor1.dpj`). The project will appear in the left window of VDSP++. If you double-click on `muster.asm`, then the source code is shown in the editor window.

Compile your project by pressing function key F7. The executable program is loaded automatically to the DSP memory if no compilation errors occur. Otherwise you must correct syntax errors and try again. The program is loaded correctly if the code line with label `main:` is high-lighted in the editor window. Press F5 to run your program and SHIFT+F5 to halt execution. No debugging is possible during program run.

If you want to inspect registers and memory, then halt the DSP and select the respective windows from the menu bar. Select "Memory - Two Column" for data memory DM, and "Memory - Three Column" to display program memory PM. The display format of storage locations can be changed by clicking the right mouse button within the respective window and left click on "Select Format".

You can run your code to a breakpoint by selecting the source code line where to stop program execution. Press F9 to activate the breakpoint, press F9 again to clear a breakpoint. Start your program with F5. In case of a runtime error (endless loop, erroneous pointer moves etc.), you can select “Debug - Reset” from the menu bar and wait patiently for the timeout. In certain cases like overwriting the communication interface code by a programming error, you must close VDPS++, make a hard reset of the DSP board (by shortly removing the power supply plug), and start again.

Nearly all problems are simple and can be programmed by a short number of code lines. Therefore, a simple debugging is performed by commenting out suspicious lines of code to encircle an error. Additionally, the following list of common errors may help:

- DSP board is not powered on,
- no program loaded, or not compiled after source code changed (press F7),
- program not started (press F5),
- wrong input and/or output channel connected,
- no input signal,
- oscilloscope input grounded, or wrong channel selected,
- F0, F1 registers not used correctly,
- number format error, or operation with NANs,
- wrong usage of constants or expressions like 1/10 instead of 1./10 (1 is an integer, and 1. is a floating-point number),
- cyclic buffer registers not initialized,
- buffer length register not set to zero in case of linear addressing,
- undesired endless loop, or loop counter LCNTR initialized with a negative value,
- timing error (samples are omitted if program execution is to long),
- overflow, and aliasing may produce strange signal wave forms.

# D Using the HP FFT spectrum analyzer

There are two powerful FFT spectrum analyzers available in the lab to display signal spectra and digital filter transfer functions. You will need only a small set of the analyzer’s rich features. All buttons on the front panel are organized in functional groups. Most function parameters can be selected with SOFTKEYS on the right hand side of the screen. To restore a predefined setup, press RECALL in function block INSTR STATE and select STATE 1 with the respective SOFTKEY.

### Changing the frequency axis

The frequency span can be changed with button FREQ in function block MEASUREMENT. The maximum range is 0 Hz up to 100 kHz with a frequency resolution of 250

Hz (400 FFT frequency points). You can zoom into the frequency axis by selecting a start/stop frequency. The respective frequency resolution is changed accordingly since the FFT length is fixed to 400. Select the start frequency with SOFTKEY labeled DEFINE START and the frequency span with DEFINE SPAN. Enter the values with the numeric key pad or with UPARROW and DOWNARROW. Alternatively, you can define a center frequency with DEFINE CENTER and a frequency span symmetrical to the center frequency.

### Vertical axis scaling

The vertical axis of the display can be modified by pressing VERT SCALE in function block DISPLAY (located below the screen). Use the SOFTKEY labeled with LINEAR LOG to toggle between a linear and a logarithmic (dB) axis. Set the maximum axis value with DEFINE FULL SCL.

### Internal signal sources

Internal signal sources (supplied via BNC connector SOURCE on the back panel) can be selected with function block MEASUREMENT and key SOURCE. You can choose between impulse, pulse, and two types of noise-like signals. Use DEFINE ATTEN to change the signal amplitude.

Filter transfer functions should be measured with PERIODIC NOISE which is a pseudo-random noise signal with a period equal to the FFT length. Thus, no windowing effects are present, and no averaging is needed as in the case of a truly random signal. Since you get a smooth transfer function, any overload of your digital filter is easily detected by a noisy transfer function. In contrast to a delta-impulse to test your system, a pseudo-random noise has its signal energy distributed over the period length and not in a very short interval. Therefore, overloading the tested system by sharp impulses is avoided.

### Time window functions

An FFT signal analysis requires finite-length signals. Consequently, time windows must be used to limit the signal length. Window functions are selected in function block MEASUREMENT with button WINDOW. Select a rectangular (UNIFORM) window when measuring filter transfer functions with impulses, or with periodic noise. Otherwise, the transfer function would be corrupted by the FFT of the window function. Line spectra of periodic signals, however, should be displayed with a HANNING, or with a FLAT TOP window to get sharp spectral lines.

### Marker

Function block MARKER supports the measurement of frequencies and amplitudes. Position the vertical marker with LEFTARROW/RIGHTARROW keys. Pressing these keys simultaneously with the FAST key speeds up marker movement.

## E Style of your report

You will finish this course by writing a short report describing your solutions of the problems. This report should contain a short statement of the problem followed by theoretical investigations (if any) and results (plots). An easy way tho create plots is to take a screen shot with a digital camera. You should also report on your failures (if any) since it will give us a feedback to improve the course material. Source code of your solution must be included in the report. Do not print the entire `muster.asm` of each problem solution but only the code lines you inserted in the prototype program.