Scamp5d Programming Guide

version 0.81

This document gives an introduction to the programming of the Scamp5d system. It presents the overview of the SCAMP-5 architecture, and various system components. It introduces the basic concepts of the Scamp5d software programming framework, and presents examples of implementing various common functions. The features of the Scamp5d vision system, related system control and I/O operations are also covered. This is intended as an introduction that provides the user with the basic understanding of the system, and enables the user to find appropriate details in the online documentation.

1 Introduction

We will start by recalling that the main processing system that we consider here consist of the SCAMP-5 vision chip, which contains image sensor and a powerful SIMD processor array, and the Arm Cortex M0 processor core, which is a simple 32-bit integer microcontroller. A simplified system diagram is shown in Figure 1.1.

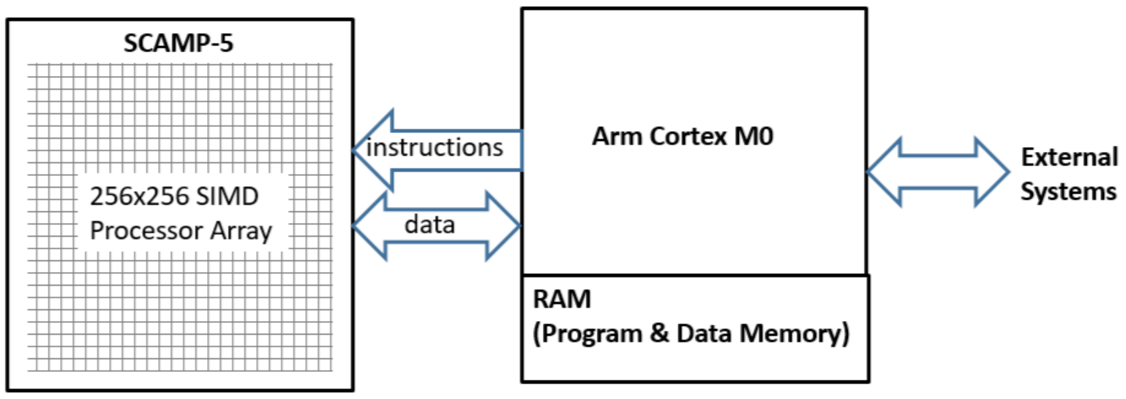

Figure 1.1. Overview of the main processing resources on the Scamp5d vision system. The main program is executed on the M0 core, which instructs the SCAMP-5 vision chip to carry out operations on image arrays using the massively-parallel SIMD processor array.

The vision algorithms are written in C/C++ and compiled onto the M0 core. The M0 core is used to provide interfaces to the external world, and the overall control flow of the vision algorithm. It can also execute sequential parts of the vision algorithm, in particular some higher-level operations on the image-derived data. The majority of vision computations, in particular the operations on images, are executed on the SCAMP-5 chip itself. The program running on the M0 core explicitly instructs the vision chip to carry out these operations, at specific points in the M0 code - this is done via *scamp5 function* calls and definitions of *scamp5 kernel* code.

Like a conventional CMOS image sensor, the SCAMP-5 is capable of acquiring the images (video frames), but in addition to that, it is capable of processing them in-situ, using processing elements adjacent to the image pixels. In general, the large arrays of data used in the image processing algorithms do not leave the SCAMP-5 device. Instead, they are processed concurrently, using a massively-parallel array of processor cores, resulting in high-speed and low-power operation. As the M0 broadcasts the instructions to all the processors in the array, they execute operations on local data. Eventually, the results of these computations are read-out from the SCAMP-5 vision chip.

In this programming guide we will learn how we can use this system to efficiently implement a range of common operations that are used to construct vision algorithms.

2 SCAMP-5 Architecture

To understand the SCAMP-5 programming, it is essential to appreciate some details of its internal architecture. The SCAMP-5 chip contains an array of 256x256 processors, one per image pixel. The architecture of a single processor is shown in Figure 2.1. A single processor contains: six general-purpose analog registers: **A,B,C,D,E,F,** thirteen binary (1-bit) registers, **RF, RP, RN, RE, RW, RS, S0-S6** (some of which have special functions), special-purpose registers used for temporary storage and inter-processor communication (**NEWS**), image sensing (**PIX**), and input (**IN**), an activity-control register (**FLAG**), Arithmetic Logic Unit (ALU) implementing basic arithmetic and logic operations as well as asynchronous accelerations for some spatial functions (Blur: low-pass filter, Prop: flood-fill), and I/O communication circuits, including array selection registers (SLCT and RECT).

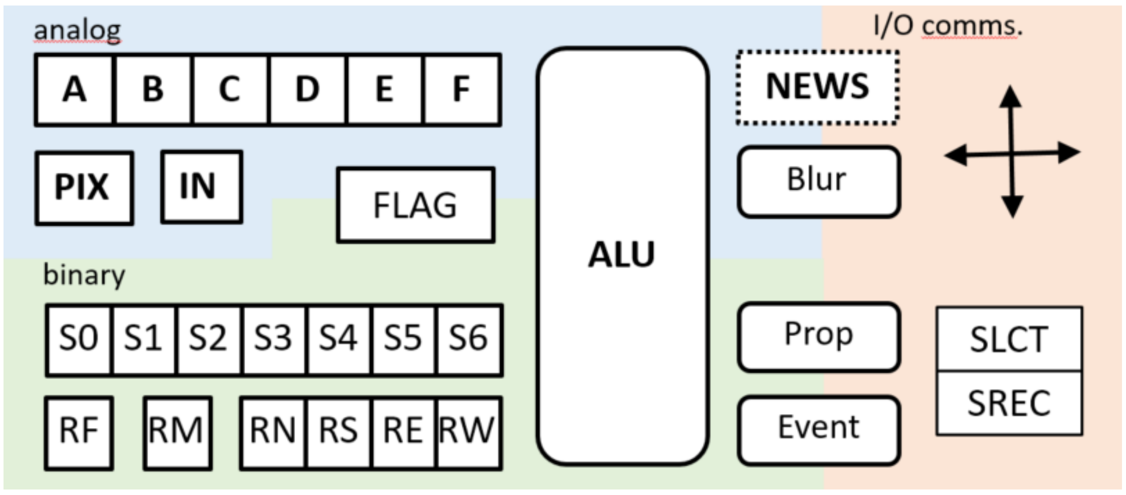

Figure 2.1. SCAMP-5 processing element architecture.

The processors operate with a common controller, forming a pixel-parallel SIMD processor array. SIMD stands for Single Instruction Multiple Data, it means that the same instruction is executed concurrently on all processors in the array, but they operate on their own data, e.g. local pixel values. The processors can also exchange data with their direct neighbours in the array (in the North, East, West and South directions), and can conditionally execute some instructions. SCAMP-5 chip also includes mechanisms that allow selecting individual and groups of processors, perform some global (array-wide) operations, and a variety of input/output schemes. These will be introduced gradually in this document.

2.1 Pixel-Parallel Operations

To understand the programming model, it is most convenient to think about all the processors in the array collectively as an 'array processor'. Individual processors in the array are termed *processing elements* (PE's). The registers (local memories) in each PE form distributed memories that can store array data. For example, registers **A** of all processors in the array form a register array **A**. This array **A** can store a 256x256 pixels gray-level image. This view is illustrated in Figure 2.2. In the following, we will simply refer to the register arrays **A**, **B**, **C**, etc. as *registers*.

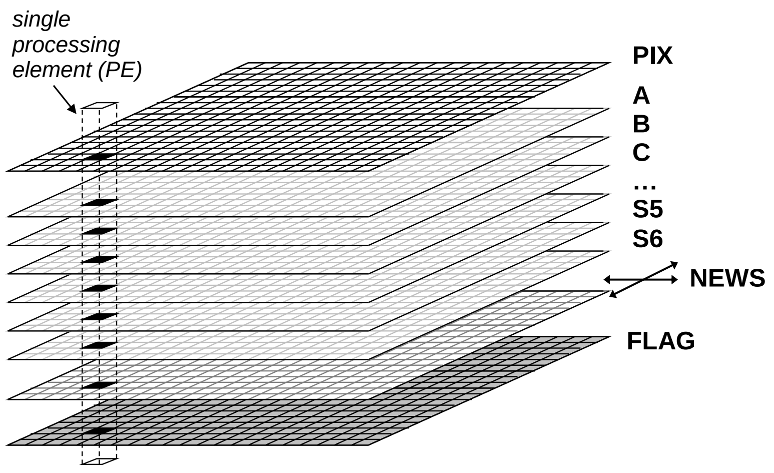

Figure 2.2. SIMD array. Each single processor (processing element, PE) is responsible for one location in the array, and performs operations on data stored in its local memories. As all PE's execute the same instructions in parallel, effectively the processor array executes array-wide operations.

The SIMD processor array can perform operations in parallel on entire data arrays. For example, operation denoted as **mov (A,B)** corresponds to each processing element in the SIMD array copying the value held in their register **B** to the register **A**. This is done in parallel in all processing elements, therefore the operation moves the content of array **B** into array **A.** Similarly, operation **add (C,A,B)** performs element-wise addition of elements of arrays **A** and **B** and puts the result in array **C** (see Figure 2.3).

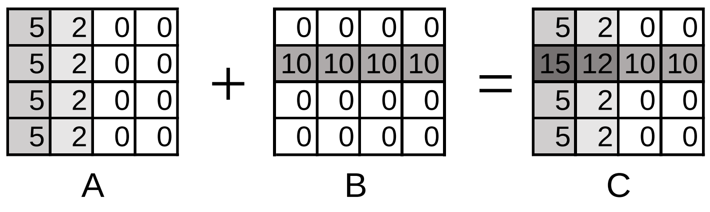

Figure 2.3. Instruction **add (C,A,B)** performs operation **C=A+B** on all array elements in parallel.

This array programming concept works in a way similar to code vectorisation in MATLAB or using numPy arrays in Python, where a single operation on arrays results in element-wise computations on their elements. However, it is worth remembering that in case of the SCAMP system, all array elements are actually operated on concurrently in hardware, giving it high speed – for example two 256x256 arrays of numbers (e.g. two images) are added in one clock cycle.

3 Instruction Set Overview

Here the instructions available on SCAMP-5 are briefly introduced. The details can be found in the **SCAMP-5 Reference Manual** document, and in the online reference documentation.

3.1 Analog Instructions

Registers **A**,**B**,**C**,**D**,**E**,**F** are called "analog" registers, due to the fact that they are implemented using analog (continuous-valued) circuit techniques. The analog registers can store real numbers, but their precision is limited by analog operation error and noise. This will be discussed in more detail in Section

4.1. The numbers stored in analog registers are in the range -128 to 127, and can represent image pixel values, or some other continuous-valued variables.

The ALU supports the following operations on analog registers: transfer: (**mov**), addition (**add**), subtraction (**sub**), sign-inversion (**neg**), absolute value (**abs**), and division-by-two (**div, diva**, **divq**). There is no multiplication operation, except the division-by-two instructions that multiply the analog value by 0.5 (i.e. divide the value by two). The **diva** and **divq** instructions should be normally used, as they provide significantly better accuracy through error-compensation scheme.

Most analog instructions take two clock cycles. Error-compensated divisions take five clock cycles. It is possible to optimise the performance using low-level microinstructions (**bus, icw**), but this is only recommended for advanced users.

Some instructions put constraints on the arguments used, that need to be strictly followed, as certain register combinations are not allowed. For example, **add (A,A,B)** is allowed (A=A+B) but **add (A,B,B)** is not allowed (A=B+B). Details are in the **SCAMP-5 Reference Manual.**

Information for advanced users

1. In contrast to earlier SCAMP implementations, the Scamp5d system instruction set embeds basic error correction strategies ("delta offsets" etc) in kernel macro instructions. Each kernel instruction (such as **mov**, **add**, **div** etc.) can therefore comprise several machine-level microinstructions, or "Instruction Code Words" (ICWs). The analog kernel macro instructions, for example **mov (A,B)**, use internally a temporary register (**NEWS**) to perform sign inversion and error correction, and are recommended to be used in most cases.
2. The instruction set also includes bus transfer microinstructions, such as **bus (A,B,C)**, that are equivalent to individual register-transfer ICWs (**A<-B+C** in this case), and can be used to construct highly optimised code. However, the user needs to manually keep track of the error-correction schemes. Therefore analog bus transfer microinstructions, and similar, (**bus**, **sq**, **blur**, **icw**, etc.) are only recommended for experienced users requiring advanced code optimisations.
3. SCAMP-5 instruction set includes experimental instructions **sq**, **square** and **mult**, providing arithmetic operations of squaring an analog value, and multiplication of two analog values. These instructions are executed with relatively low accuracy and high levels of noise.

3.2 Neighbour communication

The processors in the array can communicate with their nearest neighbours, so that data transfers and arithmetic operations can be carried out between the adjacent processors in the array. This corresponds to the operations on the arrays shifted by one-pixel in one of the four directions (denoted as ***east***, ***west***, ***north***, ***south***). For instance, (see Figure 3.1) transfer instruction **movx (A, B, south)** loads the register **A** of one processor with the value held in a register **B** of its neighbour below (i.e. in the South direction). If array **B** stores and image, then this will result in the array **A** storing an image shifted by one pixel in the North direction (i.e. up). Boundary pixels are loaded with zero.

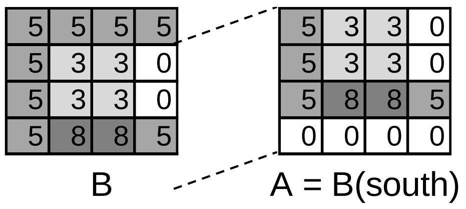

Figure 3.1: Instruction **movx (A,B,*south*)** loads the array from B to A, such that elements of A at location (x,y) are assigned the corresponding South-neighbour values from B, i.e. A(x,y)=B(x,y-1). Note that this results in image A shifted by one-pixel up (i.e. the North direction) as compared with image in B.

Neighbour communications can be incorporated into arithmetic operations, for example instruction **addx (A,B,C,*south*)** performs the pixel-wise addition of the image **B** and image **C** shifted by one pixel up, and the result is stored in image **A** (see Figure 3.2).

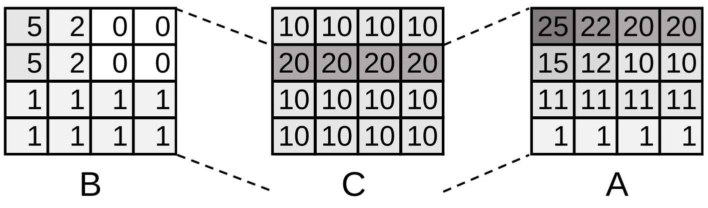

Figure 3.2: Instruction **addx (A,B,C,south)** performs operation A(x,y)=B(x,y)+C(x,y+1) on all array elements in parallel.

Speed-optimised operations **mox2x**, **add2x**, **sub2x** are provided to perform transfers and arithmetics using second-neighbours (in a 4-connected neighbourhood). This allows diagonal data transfers (e.g. ***north*** + ***west***) or 2 pixels away (e.g. ***east*** + ***east***).

3.3 Conditional execution (FLAG)

While all processors in the SIMD array receive the same instruction stream, it is often required to provide some degree of local autonomy, so that different operations can be performed on different elements of the array, in data-dependent fashion. For instance, to carry out a rectification operation (such as used, for instance in a ReLU layer of a neural network) those elements of the array that store negative values should become zero, while positive values remain unchanged.

Conditional execution is implemented using a local activity **FLAG** register. This register controls the execution of analogue instructions in a processors. Those processors in the array that have **FLAG**=1 execute analog operations, those where **FLAG**=0 do not execute. In this way, while a single instruction stream is delivered to all processors, only a subset of the array actually performs the operations, achieving data-dependent program execution.

Execution of all instructions operating on analog registers is conditional upon the **FLAG** value. However, it should be noted that conditional execution of the second-neighbour instructions (**mov2x**, **add2x**, **sub2x**) is not supported and will lead to erroneous results. Also, the **FLAG** register status does not affect the operations on binary registers.

Conditional instruction **where (A)** sets the local activity flag register **FLAG** based on analog register value, pixels where **A**>0 become enabled (**FLAG**=1), others become disabled (**FLAG**=0).

Instruction **all()** sets **FLAG**=1 in all pixels.

Conditional instructions are also available to allow conditions based on the sum of analog registers, e.g. **where(A,B)** sets FLAG only where **A+B>0**, as well as binary registers, e.g. **where (S1)** sets the **FLAG** equal to the binary register **S1**.

Example 3.1: Rectification

Consider the following scamp kernel:

Listing 3.1. Rectification

```
neg (A, B)
where (A)
  res (B)
all()
```

The result of executing this sequence of instructions is shown in Figure 3.3. The first instruction **neg (A,B)** is negating, i.e. inverting the sign of the value held in register **B**, and storing it into register **A.** The second instruction sets **FLAG** to 1 only where **A** is greater than zero (which are here corresponding to the locations where **B** was negative), other locations have **FLAG**=0 Only these "flagged" pixels will be affected by the next instruction, **res (B)** which sets register **B** values to zero. It can be seen that, as a result, all pixels that had negative values in **B** get set to zero. Finally, instruction **all()** enables all processors in the array (**FLAG**=1 everywhere), so that following operations can continue normally.

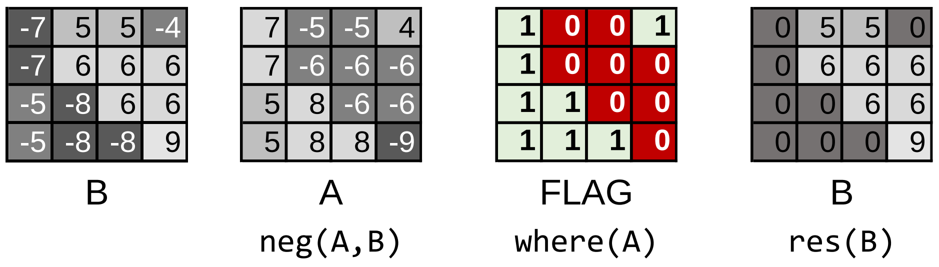

Figure 3.3: Example program calculating rectified value. Initial value held in register **B** is shown on the left. Instruction **neg (A,B)** results in **A = - B**. After **where (A)** instruction, the **FLAG** register is set to 1, or reset to 0, depending on the sign of the value stored in **A**. The following instruction **res (B)** is executed only at locations where **FLAG**=1.

Example 3.2: Absolute value

Consider the following scamp kernel:

Listing 3.2. Absolute value

```
neg (B, C)
where (C)
  mov (B, C)
all()
```

The result is shown in Figure 3.4. The first instruction **neg (B,C)** is putting the inverted values of **C** into **B.** The second instruction enables only the processors where **C** was greater than zero. In the enabled locations, the next instruction, **mov (B,C)** will copy values from **C** to **B**. The overall effect is that **B** will contain only positive values, equal to the absolute value of pixels in **C**. All processors are enabled at the end using **all()**.

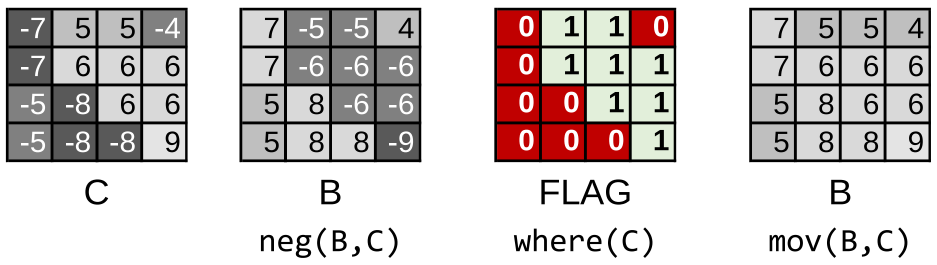

Figure 3.4: Example program calculating absolute value. Original value held in register **C** is shown on the left. Instruction **neg (B,C)** results in **B = - C**. After **where (C)** instruction, the **FLAG** register is set to 1, or reset to 0, depending on the sign of the value stored in **C**. The following instruction **mov (B,C)** is executed only at locations where **FLAG**=1.

Note, that absolute value calculation can be achieved executing an instruction **abs (C,B)**, which carries out an optimised computation of **C = |B|** in fewer clock cycles than the above program.

3.4 Binary Instructions.

General-purpose digital registers (**S0**-**S6**) and special-purpose digital registers (**RN,RE,RS,RW,RP,RF**) can store binary data (one bit per array element). The instruction set supports transfers and logic operations on these binary arrays. Binary instructions mnemonics are written in uppercase, for example instruction **MOV (S0, RE)** copies the content of register **RE** into register **S0**.

Instructions **SET** and **CLR** can be used to set one or several binary register arrays to 1 or 0. Basic logic operations of **NOT**, **OR**, **AND**, **NOR**, **NAND** and **XOR** are carried out bit-wise on the binary registers (for example, see Figure 3.5).

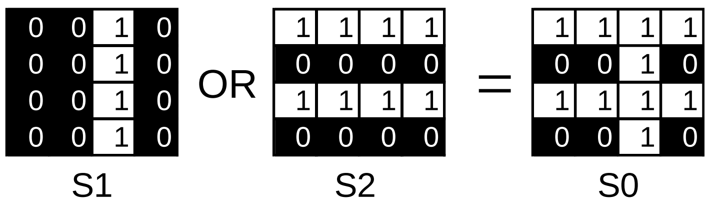

Figure 3.5. Binary instructions, such as **OR (S0,S1,S2)** operate element-wise on all array elements. In this example, **S0 = S1 OR S2**.

In addition to basic Boolean functions, optimised instructions are provided to implement several other logic functions, such as implication (**IMP**) and its inverse (**NIMP**), a “clear if” function (**CLR_IF**) and a multiplexor/selector function (**MUX**). These are often useful when performing binary manipulations, their truthtables are shown in Figure 3.6.

Figure 3.6. Truth tables for logic instructions. (a) results of executing AND (Rc,Rb,Ra)

Overall, logic instructions are straightforward to use, but it has to be remembered that some of them modify registers **RF** and **RM.** For instance AND (S0, S1, S2) instruction implements S0 = S1 AND S2, but also sets RM = NOT (S1), and RF = NOT (S2)

ANDX and NANDX perform logic AND and NAND operations, but they also modify one of the arguments. These instructions are also 25% faster than AND and NAND instructions.

3.5 Conditional execution of binary operations (RM, RF)

Binary registers can be used as arguments in a conditional instruction, e.g. **where(S1)**, to set/reset the **FLAG**, however, the binary operations are <u>not</u> executed conditionally inside the **where()** statements.

Binary instructions execute always, on all elements of the binary arrays, irrespective of the **FLAG** status.

Register **RM** is a special register. Writing to this register is gated (i.e. flagged by) register **RF**. The elements of the **RM** array are updated after any of the following binary operations (**SET**, **CLR**, **MOV**, **NOT**, **NOR**, **OR**) only in the locations where **RF**=1. Elements of **RM** where **RF**=0 remain unaffected. This is illustrated in Figure 3.7.

Figure 3.7. Illustration of **RM** operation "flagged" by **RF**. Instruction **OR (RF,S0)** implements logic operation **RF = RF OR R0** in pixels where **RF**=1. **RM** does not change in pixels where **RF**=0.

Note that registers **RM** and **RF** cannot be used as arguments to all binary instructions, and they are modified as a by-product of executing many binary instructions, such as **AND**, **NAND**, **XOR**, **IMP**, **MUX** etc. as detailed in the **Scamp5 Reference Manual**. Therefore the user must be careful when using these registers for general storage or computation.

The "conditional" operations can be also achieved employing correct Boolean logic operations. For example, if register **S1** contains a mask that determines which elements of the register **S0** should be set to zero, keeping other elements of register **S0** unchanged, this can be achieved using the following Boolean function: **S0 = S0 AND (NOT(S1))** as shown in Figure 3.6. This useful logic function is implemented by the **CLR_IF** instruction. Similar functionality can be obtained using **AND**, **OR**, **IMP** and **NIMP** functions, and this should be generally used if possible, instead of using instructions based on **RM**, **RF** conditional write.

Figure 3.8. Illustration of executing **CLR_IF (S0, S1).** This implements a logic operation **S0 = S0 AND (NOT (S1))** that can be interpreted as "conditional clearing" of **S0** in locations where **S1**=1.

3.6 Acceleration.

To be written…

3.7 Sensor and I/O

To be written…

4 Important notes

4.1 Numerical Precision of Analog Computing

By convention, the numerical values stored in the analog registers are represented by real numbers in the range -128 to 127. However, the SCAMP-5 system actually operates with an **analog data-path**, i.e. the local memories and arithmetic circuits are implemented using analog circuitry. This has some implications regarding the numerical precision and the nature of operations. Each transfer and arithmetic operation will be carried out with a small error value added to the true result of the operation. Some of this error is seen as systematic inaccuracy (the same for all pixels), some as fixed-pattern noise (specific to locations in the array), and some as entirely random, non-deterministic noise (i.e. the error will be slightly different each time). The user should never expect absolute numerical precision, or deterministic outcome, of any operation on the analog registers.

The actual error/noise levels can be found through experimentation, and are to some extend modelled in the simulator. Details can be found in the document 'Understanding Noise and Error of Analogue Computations on SCAMP-5'**.** As a rough approximation, it can be thought that the operations on the analog values are carried out with an accuracy corresponding to about 8-bits digital operations (0.4% precision), but unlike digital limited precision computations, analog computations exhibit noise, and error accumulation.

Transfers, additions, and subtraction operations are less noisy than a division-by-two operation. Additionally, several variants of a division operation are provided, (**div**, **diva**, **divq**) that can be selected depending on the required speed, accuracy, and the type of arguments.

Input operations, and output (readout) operations can be more noisy than arithmetic operations. It should not be assumed that instruction such as **scamp_in (A, 45)** will actually load a precise value of 45 to the analog register **A**. The value will be near 45. It should also not be assumed that the pixel value read-out off-chip as 45 (as seen, for example, in the image displayed in the Scamp Host GUI) corresponds to 45 being stored in the output register. Multiple sources of input and read-out noise exist in the system, and the values processed internally are typically more accurate than what might be seen in the input/output values seen from the outside.

4.2 Memory Volatility

Analog memory is volatile. The analog register values degrade, at a rate of several % per second. This is because analog numbers are stored as charge on capacitors, and that charge slowly "leaks" away. These leaks cannot be well controlled, so the level of leak varies from register to register and from pixel to pixel. Usually this is not a problem for fast frame-rate operation, but it has to be remembered that long-term storage in analog registers in not possible.

Binary registers are also volatile, but they can be reliably refreshed. This is achieved by periodically reading them, and writing back the read value. This restores the charge on the capacitor that is interpreted as the binary value '0' or '1'.

4.3 Scamp Simulator Limitations

Scamp Simulator is provided to aid the code development and debugging. It allows to examine the contents of all registers on the SCAMP chip, stepping through the code, etc. When used in this way, it is a very helpful tool. However, it is not an exact emulator, it is only an approximation.

1. When using the Simulation configuration, the "M0 code" is compiled for (and runs natively on) the host processor (e.g. Intel x64). A regular and not-too-fancy C code should work just fine, but the user needs to be aware of the M0 limitations in terms of available RAM (program and data memory), lack of the floating-point unit, etc. None of this is simulated.

2. The processors on the SCAMP chip have very unusual design, they employ analog memories and analog computing circuits in the datapath. These circuits are efficient, but they are also of **limited accuracy** and **noisy**. The Scamp Simulator includes models of noise and analogue processing errors in the SCAMP chip, however, these only provide an **approximation** of what is actually happening in hardware. There is no guarantee that the code that executes well on the simulator will work on the actual hardware. In particular, any 'fine tuning' of the code to compensate for the analog processing errors, by making minute adjustments to the numerical values, etc, is a futile exercise. Instead, the code has to be written in such a way, that it is robust against the analog processing errors. That is a skill that comes with some practice, but there are some simple rules that definitely need to be followed:

    a. Do not store analog register values for a long time (definitely not more than a few hundred ms). The stored values will 'leak'. Temporal effects are not simulated precisely.
    b. Do not think that if the simulator shows the error of the addition operation is 0.034 it is actually that. It is only an approximation. The actual values are unknown, and change in time.

c. Do not "chain" too many analog operations. A few tens of shifts are fine. A couple of "compensated" divisions are fine. Beyond that, the analog operation errors will accumulate. If you discover some amazing error compensation strategy, it is likely that it is an artefact of the simulation, not the real thing. Verify in hardware.

To help the user to appreciate these issues, the Scamp Simulator contains several levels of error modelling. From an error-free operation (unrealistic, but sometimes useful to get the basic algorithm to work), to a pessimistic (exaggerated error and noise) operation, designed to help the user to spot potential issues. Please consult the 'Scamp Simulator User Manual' for a full description of these levels.

5 Basic SCAMP-5 Program

Let's have a look at a structure of a simple SCAMP-5 program, containing some commonly used components. The details will be discussed later, however, at this stage it is important to understand the basic program format, so that we can explore and test the example code.

It is assumed that the reader is familiar with the contents of the 'Getting Started with Scamp5d' document, and has successfully installed SCAMP software programming framework. The features of the **Scamp Simulator** and **Scamp Host GUI** are described in separate documentation.

The code of a basic program is shown in **Listing 1.** Recall, that this program will be compiled and executed on the M0 core (the controller). The M0 core will then issue instructions to the SCAMP-5 chip, when a SCAMP-5 function or instruction is encountered.

Lines 1-2 declare the **scamp5** library and namespace. The M0 programs will use the **scamp5** library functions to communicate with the scamp chip, and other system components. The **scamp5** library defines three main types of functions:

- **Vision system functions (vs_)** – these are system level functions, that provide overall system setup, synchronisation and control, including interactions with a host GUI.
- **Scamp functions (scamp5_)** – these are higher-level functions that execute on the SCAMP-5 chip. Many of these functions involve i/o transfers between the chip and the M0 core or the external interface, or provide some other often used and/or relatively complex functionality
- **Scamp kernels** – these are short "programs", written in the machine-level language of the SCAMP-5 processor array. The M0 issues these instructions to the SCAMP-5 chip, which executes the operations. The hardware acceleration is primarily achieved through these kernels.

Line 5 defines a variable that will be used as a parameter to the SCAMP program, and set via a 'slider' control in Scamp Host GUI

Lines 10 and 13 are *vs functions* needed for system and GUI initialisation. The default initialisations enable callback procedures to control things such as GUI frame rate control sliders, etc.

Lines 16-19 are *vs functions* that add two data display windows, and one slider control, to the Scamp Host GUI. This is implemented by the Scamp5d system sending appropriate messages to the Scamp Host via the USB interface.

Lines 22-44 contain the main frame loop

Line 25 contains a *vs function* call typically required at a start of each frame loop iteration. It provides frame synchronisation, and other housekeeping tasks (handling GUI events, such as update of slider variables, frame counting, etc.)

Line 28 is a *scamp5 function* that loads the image acquired by the sensor into register **C**. This function applies a gain to the image acquisition, which in this case is from default variable FRAME_GAIN, which corresponds to a frame gain slider in the host GUI.

Line 31 is a *scamp5 function* that loads a value into register **E** of the processor array. In this case, this value is a variable corresponding to the slider.

Lines 34-38 contain a *SCAMP kernel*. The kernel is an inline assembler code, in the machine language of the SCAMP chip. These instructions will be executed on the SCAMP device. In this example, a thresholding operation is carried out on the acquired image **C**, at a level determined by **E** (which now contains slider_val), with the result of the operation stored into binary register array **S1**.

Line 42-43 are *scamp5 function*s. They read the corresponding analog and binary images from the SCAMP chip, and send them over the default interface (in this case USB, to be displayed on the corresponding Scamp Host windows that were created in Lines 16-17).

Listing 1. Example program

```
#include <scamp5.hpp>
using namespace SCAMP5_PE;

//variable to hold some program parameter
int slider_val;

int main(){

        // initialise the vision system
        vs_init();

        // initialise standard GUI elements (frame_rate, sensor_gain)
        vs_gui_init ();

        // add GUI elements: output displays, input slider)
        auto display_1 = vs_gui_add_display("hello",0,0);
        auto display_2 = vs_gui_add_display("world",0,0);

        //add slider

        // main frame loop
        while(1){

                // frame sync and housekeeping
                vs_frame_loop_control ();

                // load image from sensor to register C
                scamp5_getimage (C, FRAME_GAIN);

                // load value from GUI slider to register E
                scamp5_in (E, slider_val)

                // this kernel performs threshold
                scamp5_kernel_begin();
                  add (A, C, E);
                  where (A);
                  mov (S1, FLAG);
                  all();
                scamp5_kernel_end();

                //output results to GUI
                scamp5_output_image(C,display_1);
                scamp5_output_image(R5,display_2);

     }
     return 0;
}
```

Contents (to be written…)

10. Displays & Palettes

11. Scopes, histograms

12. Inputs, “plotting into array”, load image

13. External Memory

14. GPIO, LEDs

15. Timers stc

16. Simulator

    a. Simulating pixel acquisition
    b. Load and save files
    c. Simulator limitations

17. Analog computing

    - Microarchitecture - ICWs
    - UNDER THE HOOD – NEWS, TMP….

    Code optimisations using bus()
    - Scale, dynamic range – detail
    - Leakage
    - Also digital – refresh
    - Other gotcha’s

18. Algorithms
    a. ADC & DAC
    b. HDR
    c. FAST
    d. Optic Flow
    e. etc…

Version History

Version number	Date	Author	Comments
v 0.81	03/04/2019	P.Dudek	work in progress
…			