©2016 Society for Imaging Science and Technology
DOI: 10.2352/ISSN.2470-1173.2016.12.IMSE-275

Software Environment for Holistic Vision-System-on-Chip Programming

Peter Reichel, Jens Döge, Nico Peter, Christoph Hoppe, Andreas Reichel and Peter Schneider
Fraunhofer Institute for Integrated Circuits IIS, Germany
E-mail: peter.reichel@eas.iis.fraunhofer.de

Abstract

Image sensors with integrated signal processing – so called "Vision Chips" – allow for execution of computationally intensive processing steps directly after image acquisition. Programmable systems, whose functional units may be utilized in a flexible manner for various image processing tasks, require a flexible, modular toolchain. A programming environment, consisting of an assembler supporting ASIP-based (Application Specific Instruction Set Processor) control units and a Python translator supporting a subset of the Python programming language, will be presented. Library elements are used to further abstract the behavior of the underlying Vision-System-on-Chip (VSoC). For a concrete task, both VSoC-internal and conventional processing steps can be implemented within the same project. When combined with established libraries such as OpenCV, VSoC-internal processing close to the sensor becomes a powerful tool for holistic vision task design.

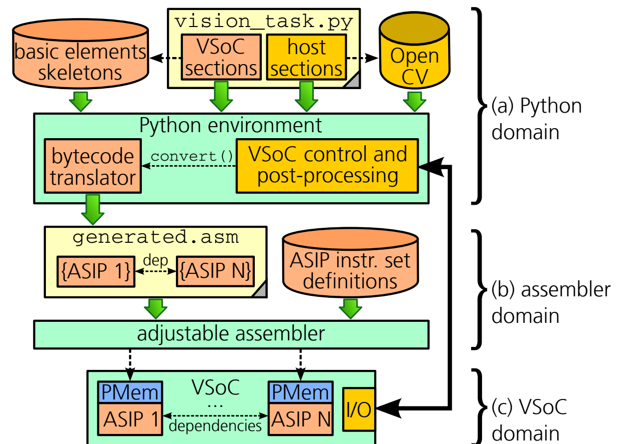

Figure 1. *Coherent depiction of the design flow.*

Introduction

Image sensors with integrated signal processing [28, 10] – so called "Vision Chips" – allow for execution of computationally intensive processing steps directly after image acquisition. Ideally, output data can be reduced to relevant features, thus avoiding the bottleneck of data transfer between image sensor and post-processing systems. This is of increased relevance in cases where conventional systems reach their limits, such as very fast processes requiring minimal latency or high image refresh rates.

Programmable Vision Chips are image processors whose functional units can be flexibly adjusted to different applications. They demand concrete vision tasks to be mapped on and be partitioned between the image processor and additional, possibly conventional, digital image processing. However, most architectures described in the literature do not have an integrated control unit and instead require extensive external control. Thus, both accessing the programmable Vision Chip's functional units as data path elements and the actual control flow is implemented externally.

A novel Vision-System-on-Chip (VSoC) presented by Döge et al. [5] allows for both digital and analog computations directly on the chip. An integrated, programmable control unit makes it possible to directly interact with external sensors and actuators and include them in the image processing algorithm [19]. The partially parallel functional units are controlled by independent ASIPs (Application Specific Instruction-set Processor). Not least because of the platform's heterogeneous nature and the requirement of having to partition concrete tasks into both the analog, digital and post-processing domain, a suitable software component abstraction is of particular importance to the design of image processing algorithms.

Fig. 1 depicts an overview of the design flow incorporating three different domains. In the lowest domain (fig. 1 (c)) there is the VSoC itself. Its control system is made up of multiple ASIPs with possibly interdependent control flows. The second one, a newly developed assembler system (b), is easily adjustable to the specific needs of a control unit based on multiple ASIPs. The individual control flows are described within a single assembler file in order to account for both possible interdependencies and potential future optimizations across several ASIPs. Last but not least, the highest domain (a) contains a programming environment based on the high-level language Python [18, 17]. To allow for VSoC programming using a subset of Python, individual program segments can be assigned to a specific ASIP. The corresponding bytecode is transformed by a special module into the ASIP's respective assembler language. Additionally, the behavior of the underlying VSoC is further abstracted using a set of provided library elements. This encompasses different readout modes which are laid on configurable skeletons as well as basic image processing operations both in the analog and digital domain. Any segment of the input program that has not been assigned to an ASIP is called a host section and, therefore, executed regularly within the Python environment. They can be used to setup the process of bytecode translation, for VSoC control, parametrization and configuration as well as for post-processing the readout data. The latter can be accomplished in combination with arbitrary image processing libraries such as OpenCV. Thus, the proposed programming environment allows for combining VSoC-internal operations featuring image acquisition and image processing with established vi-

©2016 Society for Imaging Science and Technology
DOI: 10.2352/ISSN.2470-1173.2016.12.IMSE-275

sion libraries, making it a powerful tool for holistic vision task design.

The rest of this paper is structured as follows: Firstly, the state of the art of development platforms and programming of various Vision Chips is summarized and after that, a short overview on the VSoC is given. Later on, the adjustable assembler system is introduced, followed by the description of the Python-based programming environment. After addressing the design of vision tasks, the proposed development environment is discussed. Finally, a brief summary is given.

State of the art

In case of Vision Chips with programmable signal processing, a control system which simplifies algorithm implementation by abstracting the supplied hardware components is of prime importance. *The CNN Chip Prototyping System* (CCPS) [22] has been developed for controlling of various CNN-based (Cellular Neural Network) imagers, encompassing both hardware and software components. While the actual CNN chip interfaces are each adjusted to the system by a special platform module, abstract operations are provided by the special assembler language AMC. Execution of AMC instructions and, as a consequence thereof, control of the CNN chip is performed by an interpreter running on a platform integrated DSP. Operations not supported by the CNN chip may instead be executed by the DSP. The high-level language ALPHA, which is translated into AMC by a separate compiler, abstracts even further by hiding the complexity of the CNNs. ALADDIN [27, 29], the successor to CCPS, is described as a "*High Performance Visual Computer*". It is structured similarly to CCPS, but partially relies on standard components and allows for integration in desktop or industrial PCs. Apart from ALPHA/AMC, programming can also be done in C. A library providing basic image processing functions and hiding the details of CNN-based image processing [24, 29] may be used in both ALPHA and C programs. Carranza et al. [4] proposed a development board for the CNN chip Ace16k [21] which is capable of operating autonomously. Programming of an FPGA-based control unit is done solely using a special assembler language.

The development and simulation environment APRON [1, 2] has been developed specifically for SCAMP Vision Chips [7]. Programming is done in a specially defined assembler language, which is translated into the respective SCAMP control words. An 8 bit micro-controller, which may be emulated on the development board, is used for implementing the control flow. The supplied *Integrated Development Environment* (IDE) incorporates not only the simulator and an editor, but also the test hardware, thereby significantly simplifying algorithm development and debugging. However, APRON neither supports the use of high-level programming languages, nor is it designed for being integrated into post-processing systems.

Vision Chips with bit-serial processor elements (PE) have been developed by Komuro et al. [12]. They have implemented the vision platform VCS-IV [14], which is controlled by its own FPGA-based processor [13] with separate integer and SIMD pipelines. For programming purposes, the compiler VCC [16] has been developed. It extends the C programming language by parallel data types for SIMD processing and mapping onto the Vision Chip and performs the transformation to bit-serial PE processing. However, it does not support to further analyzing extracted

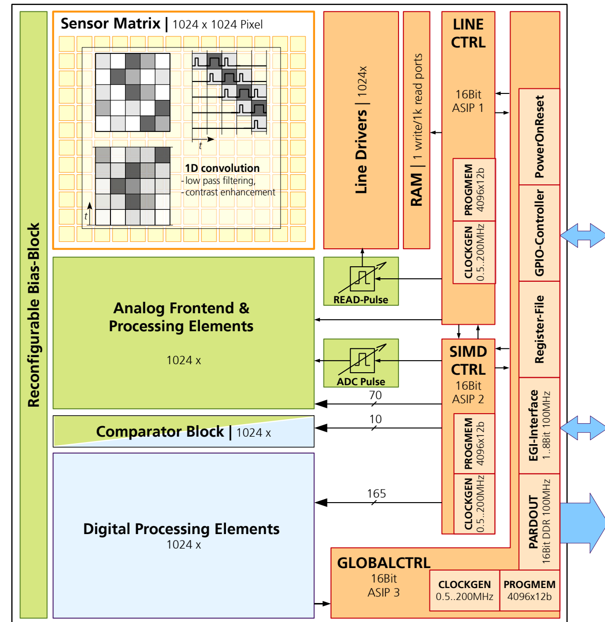

Figure 2. *Structural overview of the image sensor SoC, taken from [5].*

features using a post-processing system. A similar compiler has also been proposed by Zhang et al. [30] for their digital vision system with multiple layers of parallel processors. The Vision Chip PARIS-1 [8] acts as a micro controller's peripheral component and is thus directly incorporated into its address space. Software development is performed in C/C++ with the help of supplied macros for several basic operations.

VSoC

The programmable VSoC presented in more detail in [5] prevents some of the disadvantages [9] of previous Vision Chip concepts, such as low spatial resolution or lacking sensitivity. It is based on 1024×1024 charge-based [6] pixel cells and integrates analog as well as digital, highly parallel signal processing. Three ASIPs serve as basis for the control unit which is directly integrated into the chip [19]. Thus, they enable multiple parallel control flows as well as necessary synchronization and data exchange between them. Each ASIP incorporates a similar stack-based processor with individual extensions for controlling its respective functional units, as well as its own local program and data memory and an adjustable clock generator.

The VSoC's architecture is depicted in fig. 2. Each pixel contains both a photo-FET as its light-sensitive element [15, 6] and a current memory (SI) cell in order to save gray values. Apart from pulse-based output of the residual current of photo-FET and SI-cell onto the column wire's capacity, ASIP 1 is capable of controlling additional operations such as RESET, EXPOSURE or MEM line by line. By simultaneously or consecutively reading out multiple cells with varying pulse widths, one-dimensional convolution of digital coefficients with analog image contents is achieved. Mixed-signal processor elements (PE), which are thought of as SIMD processors controlled by ASIP 2, process the corresponding charge column-parallel. Analog-digital conversion is being done by compensational charges using current

©2016 Society for Imaging Science and Technology
DOI: 10.2352/ISSN.2470-1173.2016.12.IMSE-275

pulses with opposite sign. These charges are added to the column capacities and the resulting voltage is compared to a reference value. The ADC algorithm is thereby described in program form and executed by ASIP 2. Each PE features its own, dedicated 8-bit ALU (ADD, SUB, logic, shift) as well as 24 8-bit registers. In order to ensure local autonomy, individual PEs may be turned off depending on their internal state. The calculation results of each PE may be accessed via a common bus, thereby reaching ASIP 3. Aside from a 16 bit wide general-purpose IO port (GPIO) and a configurable, SPI-based interface (EGI), it also features a parallel, 16 bit wide double-data-rate (DDR) data bus (PARDOUT) for the purpose of communicating with the outside world. The EGI-interface is used to access the various global control registers as well as each individual ASIP's program memory from the outside. For software and application development, a simulation framework [20] based on VSoC's architecture has been designed.

Assembler

Overview

The assembler system was developed specifically with the VSoC's ASIP-based control unit in mind. The most important feature is its ability to concurrently support multiple control flows of a heterogeneous multi-ASIP architecture with differing instruction sets. Thereby, the individual processors' instruction set architecture is not fixed – instead, it can easily be modified to meet specific requirements. Furthermore, it allows for embedding Python code into the assembler files for creating macros as well as for assembler adjustment.

The assembler's architecture is depicted in figure 3. First, the parser fragments input data into individual blocks and extracts structured sequences of single operations. Those operations are mapped onto registered functions, causing the input program to be interpreted as a sequence of processor-specific function calls. Each processor of the target platform is assigned to its own namespace containing all its respective symbols.

At first, types encapsulating integer or string constants are registered at the T-Pool (type-pool). The F-Pool (function-pool) on the other hand contains callable functions with parameters corresponding to types registered at the T-Pool. Furthermore, each function possesses a short identifier, making it accessible as an operation in assembler code further down the line. Operations are assigned the respective functions from the F-Pool by matching the given parameters' types. Code segments which have already been processed are internally stored in the form of control flow graphs (CFG) [25] within the C-Pool (CFG-pool). Both source files and CFGs, as well as functions, types and constants may be loaded from external libraries and can thus be included in the current environment. Finally, apart from generating machine code for the individual processors in different formats (bin, hex, ...), the output generator also allows for directly outputting a graphical representation of the CFGs.

Parser and general language definition

Programs are structured by blocks and sub blocks of arbitrary depth, with curved brackets being used for marking purposes similar to the C programming language. Highest-level blocks, which structure a program into sections, are attached to a namespace and therefore to a processor. In addition, for every such block, a CFG is created in the C-Pool and assigned the block's

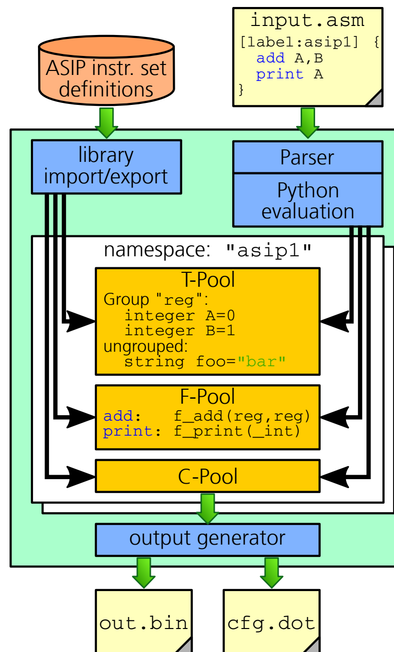

Figure 3. Depiction of the assembler system's architecture.

operations. In order to being able to label and reference arbitrary blocks, labels in the format `[label:namespace]` are used. The namespace is omitted in case of sub block labels, because they are only visible in the current program section.

An example of a single assembler file containing code segments for two ASIPs is depicted in 4. In general, statements take the form "`func par1, par2, ...`". They are treated as independent blocks, thus allowing for label assignment. The parser resolves function names by querying the F-Pool for functions with the respective identifier "`func`". Then, the T-Pool is used to match the parameters to the types given in the function declaration. For example, the statement "`add A, B`" is interpreted as a call to the function "`add(reg, reg)`". Thus, multiple overloaded functions with the same name, but differing by their parameters' types, may exist. The found function is then finally called with its given parameters. Possible parameters can also be labels as well as Python expressions, where the latter can e.g. be used for declaring tables of values. The expression is thereby evaluated even before function and parameter lookup.

Assembler adjustment

In order to adapt the assembler to a given processor architecture, a corresponding namespace has to be created and types and functions in the T- and F-Pools have to be declared. This is achieved by directly including Python code segments within the input file (see fig. 5). This approach allows for using one and the same input language for utilization as well as adaptation of the assembler.

New assembler operations are added by first declaring a new function expecting the corresponding amount of arguments. Then, when registering the function as an operation, both its name in assembler code and the parameters' expected types have to be indicated. In general, any assembler file can contain code segments which can be used to define new operations as well as to

©2016 Society for Imaging Science and Technology
DOI: 10.2352/ISSN.2470-1173.2016.12.IMSE-275

```
1   .import "asip1"                  } library imports
2   .import "asip2"
3
4   [main1:asip1] {                  } code segment
5     ...                              for asip1
6     add A,B
7     print A
8     sync_notify "asip2"
9     ...
10  }
11
12  [main2:asip2] {                  } code segment
13    ...                              for asip2
14    sync_wait "asip1"
15    ...
16  }
17
18  [tab1:asip1]                     } table for
19  table <? math.sqrt(25) ?>          asip1
```

Figure 4. *Example of an assembler file containing code for two ASIPs.*

```
1   #import library into namespace "asip1"
2   [:asip1] .import "asip_base"
3
4   [cmd_def:asip1] {
5     <?python
6       def f_add(r1, r2):
7         n=cdfg.insert_node("add")
8         n.sem.opc("1001 10%d%d" % (r1, r2))
9         ...
10
11      #register new function with 2 parameters
12      regmn("add", "reg, reg", f_add)
13    ?>
14  }
```

Figure 5. *Example of adjusting the assembler by registering a new function for namespace "`asip1`".*

create macros.

Due to the library concept, the actual architecture description does not have to be processed during each assembler run. Besides a significant speed-up, which is especially important for systems with limited computing power such as micro-controllers, data is also being compressed by conversion into intermediate code. Furthermore, it provides a simple protection of intellectual property. Objects deposited within the T-, F- and C-Pools are exported to database files along with dependencies to other libraries and can thus be imported as libraries within other programs. Thereby, importing a library into a specific namespace is exceptional in that all its symbols are integrated into the current namespace, no matter what namespace they actually reside in. This allows for defining a common instruction set which serves as the basis for the VSoC's individual ASIPs.

CFGs

When executing the function belonging to a specific operation, instead of directly generating machine code, the current CFG is expanded with new nodes and interconnections. Using CFGs as an intermediate format allows for special operations, such as repeating blocks, embedding other CFGs or annotating sub-graphs with attributes, e.g. labeling loops. The attribution of CFGs is the basis for future optimizations, particularly between control flows. Additionally, unreachable code segments can be easily identified.

While for most operations only a single node, directly connected to its predecessor, is attached to the current CFG, a link to the destination is added in case of jump operations. Because branch destination resolution depends on the order in which the CFGs were created, they are only resolved once all the input data have been processed. Referring to nodes of the same CFG is just as possible as doing so to nodes within other graphs. Thus, because the assembler description contains code for multiple processors, a block's address is available within the program memory of another processor. This may be used for purposes of inter-processor-communication, e.g. by instructing another ASIP to call a specific sub program.

Python integration

Overview

Using a high-level programming language for control flow implementation marks a significant simplification for the user. The Python programming language [18, 17] was chosen due to platform independence and the availability of a wide range of libraries. The resulting software environment's most important feature is the usage of Python for programming heterogeneous multi-ASIP architectures. For this purpose, program sections in the form of individual functions are each assigned to an ASIP of the VSoC control unit. While [11] makes use of a lightweight VM in order to run Python on a micro-controller, the target architecture is supposed to refrain from doing so. Both the ASIPs' common processor core [19] and the virtual machine (VM) of the Python reference implementation CPython [18] are stack-based. Hence, it is easily possible to transform a subset of the generated Python bytecode into assembler code as described above. This encompasses function calls and loops (`while`) as well as conditions (`if/elif/else`). All calculations and comparisons as well as parameters and function return types are restricted to pure integer processing.

Figure 6 depicts an architectural overview of the Python translation. Similarly to the concept used for the assembler system, each processor is first assigned its own namespace containing lists of all the functions and macros within. Functions hold transformable and thus VSoC-executable code, made up of only a subset of the Python programming language. On the other hand, macros are not bound by any such restrictions and may rely on arbitrary Python libraries. They can be called from functions as well as from within other macros and are used for code generation during translation, incorporating in-line assembler sections. A global convertor translates each registered function separately, whereby a processor-specific convertor, which is attached to the respective namespace, is used for the actual translation of individual bytecodes into the target machine's instructions. The result is a common assembler description holding for each translated function a block associated with the respective namespace.

Associating functions and macros

Associating functions and macros to a concrete namespace is done by assigning a decorator [23] using Python's commonly used @-notation (see fig. 6). If the surrounding Python module is executed, the associated decorator is called after creating the function object with a reference to the newly created object. The decorator `@assign()` creates a corresponding `FunctionEntry` within the respective namespace containing, apart from the name and number of parameters, the defining module as well as the function's bytecode. Accordingly, for macros, `@macro()` defines a `MacroEntry`. Including the bytecode, however, is not necessary

©2016 Society for Imaging Science and Technology
DOI: 10.2352/ISSN.2470-1173.2016.12.IMSE-275

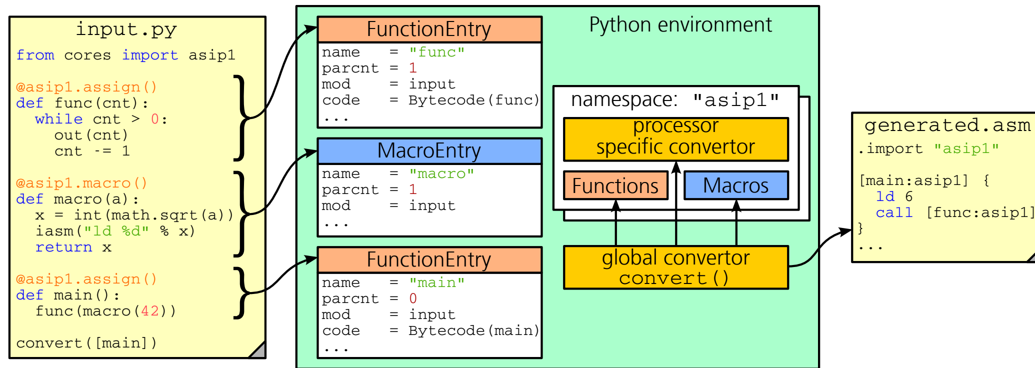

Figure 6. *Depiction of the Python translator's architecture.*

in this case. Methods, i.e. functions that are bound to a Python class object, can be associated as functions or macros to the individual ASIPs, as well. Naturally, they can only be registered after an object of the respective class has been created. Functions and macros within modules and/or classes can be assigned to any processor. Program code that has neither been assigned by `@assign` nor `@macro`, on the other hand, defines so called host sections and is executed regularly. Host sections are necessary to control the translation process, to parametrize and configure the VSoC as well as for further processing data received from the VSoC.

Translation process

Calling the function `convert(entry_list)` within regular program code initiates the actual translation process by sequentially converting all existing namespaces. The functions registered in each namespace are thereby considered individually. For each processor, one function used as entry point for program execution is passed to `convert()`.

A function's stack layout, i.e. the position of arguments and variables, can be assumed as being invariant before and after execution of arbitrary Python statements. During statement execution, variables, constants and/or symbols are being pushed onto the stack and may be arbitrarily modified. Afterwards, the original stack layout is inherently restored. However, a special macro `iasm()` allows for inserting in-line assembler expressions at arbitrary locations inside a function. Within such an in-line assembler section, elements can be pushed onto the stack. They can remain there even beyond the end of the statement for use in a subsequent assembler section. However, because of its possibly non-linear nature due to branches and loops, an analysis of all possible paths within the function's control flow is mandatory. Therefore, a function's given bytecode is structured into linear segments, with each segment being closed by a control statement such as a (conditional) jump operation or a return statement. Additionally, two representations of the current function's operand stack are held in order to perform the transformation: on the one hand, the representation on the target architecture `tstack`, on the other hand, the internal representation `istack`. Both are initialized with the parameters passed to the function and the utilized variables before the actual conversion can take place. Stack representations are updated both for transformed bytecodes and operations within in-line assembler sections. Once conversion reaches the end of the current linear segment, snapshots of the current `tstack` and `istack` are taken and attached to the addresses of all following linear segments. The translation process follows all possible control flow paths and, upon entering a linear segment, restores the already known snapshot of both stacks. If a follow-up linear segment is reached via multiple paths, the stack snapshots have to match wrt. their depth and deposited types. While this assertion is always met in case of regular Python expressions, it may be violated by stack modifications within an in-line assembler section.

Individual bytecodes are translated using the processor-specific convertor associated with the current namespace. Integer constants and variables are always loaded and processed, e.g. with calculations, on both stack representations in parallel. Because concrete variable values and intermediate results are only known at run-time, however, the stacks only hold the respective type. Based on that, in order to load and save local variables, or to delete the current function's stack-frame on RETURN, it is necessary to calculate the current offset to the concrete position relative to the stack pointer. Loaded symbols, i.e. references to modules, objects, functions or macros, are stored solely on the `istack` and can be resolved using the module or object stored within the `FunctionEntry` or `MacroEntry`. In case of a `CALL_FUNCTION` Python bytecode, the type stored on the `istack` is used to distinguish macro from actual function calls. In case of a function, the code necessary to pass the parameters and to execute the function, is generated. Macros on the other hand are thereby directly called by the convertor during translation.

Vision task design

Library concept

VSoC programming is abstracted by a library supplying predefined basic elements for essential tasks in the form of functions or macros. This involves e.g. algorithms for controlling analog-digital-conversion (ADC, e.g. *single-slope* or *successive approximation* with parametrizable precision), various pixel matrix readout modes (e.g. global shutter or global reset), selected digital post-processing within the SIMD array (e.g. finding interest points) or varying output data formats. The individual routines are generally implemented directly in assembler and optimized for maximal throughput and minimal latency respectively.

Apart from basic elements, parametrizable *skeletons* are provided as foundation for concrete image acquisition and processing operations. A skeleton thereby determines the global communication and synchronization scheme of the VSoC's individual ASIPs,

©2016 Society for Imaging Science and Technology
DOI: 10.2352/ISSN.2470-1173.2016.12.IMSE-275

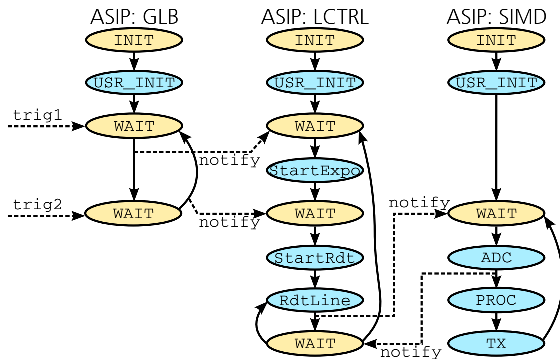

Figure 7. *Depiction of a skeleton's synchronization for image acquisition.*

```
 1 #instance of the vsoc connection                      } (a)
 2 vsoc = VSoC(...)
 3
 4 #own processing function                              } (b)
 5 @simd.assign()
 6 def my_proc():
 7   ...
 8
 9 #skeleton instance                                    } (c)
10 skel = skeletons.SimpleImgRead(vsoc)
11 skel.reg_slot("ADC", adc.single_slope)
12 skel.reg_slot("PROC", my_proc)
13 ...
14
15 #conversion and programing                            } (d)
16 vsoc.program(convert_and_asm(...))
17
18 #set region-of-interest                               } (e)
19 skel.set("roi", (0,0),(1024,1024))
20
21 #image acquisition and processing                     } (f)
22 import cv2 as cv
23 import numpy as np
24 krnl = np.ones((5,5),np.uint8)
25
26 while True:
27   img=skel.acquire_img()
28   erosion = cv.erode(img,krnl)
29   cv.imshow("Image", erosion)
30   ...
```

Figure 8. *Example of a vision task with OpenCV-based post-processing.*

both among each other and with their environment, thus acting as a macro-based code generator. Each skeleton is represented by a sub class of `BaseSkeleton` and allows for the registration of functions to be executed at certain points of the synchronization scheme. In the following, these points are called *slots*. Besides the methods for slot registration, the skeleton's interface to the host sections of the Python program may also provide methods for parametrization wrt. code generation or for communication with the VSoC at run-time. These methods may then set options defined by the skeleton (e.g. filter coefficients, region of interest) as well as readout and further process image and/or feature data.

In fig. 7 the simplified sequence of image acquisition using global reset readout is depicted. For each ASIP, `BaseSkeleton` defines a dedicated `main` function as a method executing a code generation macro based on the skeleton object's state and the functions registered at the slots – which are drawn in light blue in fig. 7. Currently, besides different readout modes such as global or rolling shutter, some application-specific skeletons are defined. In order to use a skeleton, the respective object is created and the functions are registered at the individual slots. While registering a function is mandatory for some slots (e.g. "ADC" in fig. 7), others do not require that (e.g. "PROC" (SIMD-based digital post-processing)). Apart from the basic elements provided by the library itself, own functions can be defined and hence integrated into the task. Substituting individual processing steps is therefore possible at any time.

Holistic design

The proposed environment enables programming a concrete vision task consisting of VSoC-based image acquisition and preprocessing, possibly initial feature extraction as well as corresponding post-processing, all within a common program. In fig. 8, such an example program is given. Communication with the VSoC is realized using the respective library (fig. 8 (a)). At first, a suitable skeleton is chosen (c) and basic elements supplied by the library or own functions (b) are registered at the slots. This is followed by converting, assembling as well as programming of the VSoC (d). Using the methods defined by the skeleton object, it is possible to communicate with the VSoC e.g. by setting options (e) or reading out data. For further processing (f) of the readout data, any desired library, such as OpenCV [3], can be used.

Discussion of development status

Programming the VSoC in a high-level language significantly alleviates the burden of vision task implementation, especially when compared to APRON [1, 2] for the Vision Chip SCAMP-3 [7], which only allows for pure assembler descriptions. In contrast to developing a separate one as in [22], consistent use of Python – an established and widespread high-level programming language – makes the proposed system easily extensible. At first, a skeleton outlining the process of image acquisition and processing, is chosen. At specific points, both basic elements extracted from the library and separate functions may be integrated into the process. The use of established libraries such as OpenCV for post-processing readout data allows for complex tasks to be holistically viewed as well as both quickly and easily implemented. This concept has been successfully applied to a texture-based 2d presence detection system for LED light control [26].

However, the individual Python bytecodes are translated one-to-one with only a few optimizations regarding its length or achievable performance. At the moment, the only ones performed during transformation are *Constant-Propagation*, *Constant-Folding* and, if possible, *Variable-Elimination*. Nevertheless, important and especially time-critical functions – just like the basic elements of the described library – can be implemented in assembler.

The actual translation into assembler is still restricted to fundamental constructs (functions, loops and conditions as well as calculations with integer variables). Special ASIP instructions can only be executed using in-line assembler or the respective macros. Similar to the compilers described in [16] and [30], automatically creating code for the VSoC's SIMD array is a future possibility. This could be achieved using a special data type which is treated separately by the convertor if used within expressions.

ASIP interdependencies can be accounted for due to a sin-

©2016 Society for Imaging Science and Technology
DOI: 10.2352/ISSN.2470-1173.2016.12.IMSE-275

gle input file holding the complete description. CFGs, which are thereby created by the assembler as an intermediate format, are right now only visualized and subsequently transferred into their respective machine code representation. Because of the clock ratios of the individual ASIPs, the time it takes to execute specific code segments as well as the position of synchronization points are known, generated code may be relocated by taking data flow into account, thereby optimizing execution time.

Conclusion

A programming environment allowing for the design of vision tasks using VSoC-based image acquisition and preprocessing combined with established libraries has been proposed. Describing the program flow within the VSoC, parametrizing and configuring it as well as post-processing readout data is being done using the Python programming language. Individual program segments are assigned to the control unit's ASIPs and transformed into the respective assembler language. A library provides basic functionalities such as analog-digital-conversion. A parametrizable skeleton which is extensible by library elements or separate processing functions at defined locations, outlines the process of image acquisition and actual processing. A flexible assembler system allows for considering multiple, interdependent ASIP control flows simultaneously.

Due to a library concept as well as embedding complex code segments in the Python programming language, the proposed system can be easily adjusted for use with specific processors.

References

[1] David RW Barr and Piotr Dudek. A cellular processor array simulation and hardware prototyping tool. In *Cellular Neural Networks and Their Applications, 2008. CNNA 2008. 11th International Workshop on*, pages 213–218. IEEE, 2008.

[2] David RW Barr and Piotr Dudek. APRON: A cellular processor array simulation and hardware design tool. *EURASIP Journal on Advances in Signal Processing*, 2009.

[3] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. " O'Reilly Media, Inc.", 2008.

[4] Luis Carranza, Francisco Jimenez-Garrido, Gustavo Linan-Cembrano, Elisenda Roca, Servando Espejo Meana, and Angel Rodríguez-Vázquez. Ace16k based stand-alone system for real-time pre-processing tasks. In *Microtechnologies for the New Millennium 2005*, pages 872–879. International Society for Optics and Photonics, 2005.

[5] J. Döge, C. Hoppe, P. Reichel, and N. Peter. A 1 megapixel HDR image sensor SoC with highly parallel mixed-signal processing. In *International Image Sensor Workshop (IISW)*. IEEE, June 2015.

[6] Jens Döge. *Ladungsbasierte analog-digitale Signalverarbeitung für schnelle CMOS-Bildsensoren*. TUDpress, Dresden, 2008.

[7] Piotr Dudek. SCAMP-3: A vision chip with SIMD current-mode analogue processor array. In *Focal-plane sensor-processor chips*, pages 17–43. Springer, 2011.

[8] Antoine Dupret, Jacques-Olivier Klein, and Abdallah Nshare. A dsp-like analogue processing unit for smart image sensors. *International Journal of Circuit Theory and Applications*, 30(6):595–609, 2002.

[9] Antoine Dupret, Michael Tchagaspanian, Arnaud Verdant, Laurent Alacoque, and Arnaud Peizerat. Smart imagers of the future. In *DATE*, pages 437–442, 2011.

[10] Daniel Durini. *High Performance Silicon Imaging: Fundamentals and Applications of CMOS and CCD sensors*. Elsevier, 2014.

[11] Damien George. MicroPython python for microcontrollers, aug 2015.

[12] M. Ishikawa and T. Komuro. Digital vision chips and high-speed vision systems. In *VLSI Circuits, 2001. Digest of Technical Papers. 2001 Symposium on*, pages 1–4, June 2001.

[13] S. Kagami, T. Komuro, I. Ishii, and M. Ishikawa. A real-time visual processing system using a general-purpose vision chip. In *Robotics and Automation, 2002. Proceedings. ICRA '02. IEEE International Conference on*, volume 2, pages 1229–1234, 2002.

[14] S. Kagami, T. Komuro, and M. Ishikawa. A high-speed vision system with in-pixel programmable ADCs and PEs for real-time visual sensing. In *Advanced Motion Control, 2004. AMC '04. The 8th IEEE International Workshop on*, pages 439–443, March 2004.

[15] SD Kirkish, JC Daly, L Jou, and Shing-Fong Su. Optical characteristics of CMOS-fabricated MOSFET's. *IEEE Journal of Solid-State Circuits*, 22:299–301, 1987.

[16] T. Komuro, S. Kagami, M. Ishikawa, and Y. Katayama. Development of a bit-level compiler for massively parallel vision chips. In *Computer Architecture for Machine Perception, 2005. CAMP 2005. Proceedings. Seventh International Workshop on*, pages 204–209, July 2005.

[17] Mark Pilgrim and Simon Willison. *Dive Into Python 3*. Springer, 2009.

[18] Python Software Foundation. Python website, aug 2015.

[19] Peter Reichel, Jens Döge, Nico Peter, and Christoph Hoppe. An ASIP-based control system for vision chips with highly parallel signal processing. In *The 24th IEEE International Symposium on Industrial Electronics (ISIE)*. IEEE, June 2015.

[20] Peter Reichel, Christoph Hoppe, Jens Döge, and Nico Peter. Simulation environment for a vision-system-on-chip with integrated processing. In *Proceedings of the International Conference on Distributed Smart Cameras (ICDSC)*, 2015.

[21] Angel Rodríguez-Vázquez, Gustavo Liñán-Cembrano, L Carranza, Elisenda Roca-Moreno, Ricardo Carmona-Galán, Francisco Jiménez-Garrido, Rafael Domínguez-Castro, and S Espejo Meana. Ace16k: The third generation of mixed-signal SIMD-CNN ACE chips toward VSoCs. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 51(5):851–863, 2004.

[22] Tamas Roska, Akos Zarandy, Sandor Zold, Peter Foldesy, and Peter Szolgay. The computational infrastructure of analogic CNN computing part I: The CNN-UM chip prototyping system. *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on*, 46(2):261–268, 1999.

[23] Kevin D. Smith. PEP 0318 – decorators for functions and methods, aug 2015.

[24] ISTVÁN Szatmari, PÉTER Földesy, CSABA Rekeczky, and ÁKOS Zarándy. Image processing library for the aladdin visual computer. *Proceedings of the CNNA-2002 Frankfurt, Germany*, 2002.

[25] Jürgen Teich. *Digitale Hardware/Software-Systeme: Synthese und Optimierung*. Springer-Verlag, 2013.

[26] Werner Weber, Lex James, Arjan van Velzen, Bruno Vulcano, Micha Stalpers, Arend van de Stadt, Jens Döge, and Wilfried Rabaud. Peripheral devices for the right light. *LED professional Review*, 48:92–97, 2015.

[27] Ákos Zarándy. ACE box: High-performance visual computer based

©2016 Society for Imaging Science and Technology
DOI: 10.2352/ISSN.2470-1173.2016.12.IMSE-275

on the ACE4k analogic array processor chip. In *Proc. of ECCTD*, volume 1, 2001.

[28] Ákos Zarándy. *Focal-plane sensor-processor chips*. Springer, 2011.

[29] Ákos Zarándy, Csaba Rekeczky, Péter Földesy, and István Szatmári. The new framework of applications: The ALADDIN system. *Journal of Circuits, Systems, and Computers*, 12(06):769–781, 2003.

[30] Wancheng Zhang, Qiuyu Fu, and Nan-Jian Wu. A programmable vision chip based on multiple levels of parallel processors. *Solid-State Circuits, IEEE Journal of*, 46(9):2132–2147, 2011.

Author Biography

Peter Reichel *studied Information Systems Engineering at the Technische Universität Dresden where he received the Dipl.-Ing. degree in 2008. After working as development engineer for Digital Imaging, he joined the "Optical Sensors" group of the Design Automation Division (EAS) at Fraunhofer Institute for Integrated Circuits (IIS) and has been working on the design of image sensors with integrated signal processing and image processing systems based on them. Currently, he is working on his PhD.*

Jens Döge *received the Dipl.-Ing. and Dr.-Ing. degrees in electrical engineering from the Technische Universität Dresden in 1999 and 2008, respectively. After working as research assistant at the Institute for the fundamentals of Electrical Engineering until 2005, he joined the Design Automation Division (EAS) at Fraunhofer Institute for Integrated Circuits (IIS), founding the "Optical Sensors" group. His research interest is in the field of analog and mixed-signal integrated circuits and modeling, design, and characterization of image sensors and optical measuring systems.*

Nico Peter *received the Dipl.-Ing. degree in electrical engineering with specialization in microelectronics from the Technische Universität Dresden in 2013. In the same year he joined the Design Automation Division (EAS) of Fraunhofer Institute for Integrated Circuits (IIS), where he is working in the "Optical Sensors" group. His main research interest is in the field of mixed-signal integrated circuits as well as design of embedded optical measuring systems.*

Christoph Hoppe *received the Dipl.-Ing. degree in electrical engineering from the Technische Universität Dresden in 2012. In 2010, he joined the Design Automation Division (EAS) of Fraunhofer Institute for Integrated Circuits (IIS), continuing his work in the group "Optical Sensors" from 2012 onwards. His main research interest is in the field of compact and low-power analog and mixed-signal integrated circuits as well as modeling, design, and characterization of image sensors and optical measuring systems.*

Andreas Reichel *is a student of computer science at the Technische Universität Dresden. He joined the Design Automation Division (EAS) of Fraunhofer Institute for Integrated Circuits (IIS) in 2011 as a student assistant. His main interest is in photogrammetry and digital image processing. Currently, he is working on the Dipl.-Inf. degree.*

Peter Schneider *received his Dipl.-Ing. and PhD from Technische Universität Dresden in 1993 and 2010, respectively. In 1993, he joined the Design Automation Division (EAS) at Fraunhofer Institute for Integrated Circuits (IIS) as a research assistant. He became group manager in 2000, head of the department of heterogeneous systems in 2006 and EAS' director in 2011. His research interests are modeling methodology for micro-systems and mechatronics, simulation of electromagnetic systems, 3-D system integration, development of adaptive algorithms for control and signal processing.*