Chapter 3

Rapid Prototyping of Embedded Video Processing Systems in FPGA Devices

Andrej Trost and Andrej Žemva

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/61136

Abstract

Design of video processing circuits requires a variety of tools and knowledge, and it is difficult to find the right combination of tools for an efficient design process, specifically when considering open tools for evaluation or educational purpose. This chapter presents an overview of video processing requirements, programmable devices used for embedded video processing and the components of a video processing chain. We propose a novel design flow for generating customizable intellectual property (IP) cores used in streaming video processing applications. This design flow is based on domain-specific modules in Python language. Examples of generated cores are presented.

Keywords: Video processing, prototyping, FPGA, Python, IP core

1. Introduction

Embedded sensor data processing is an important concept of future ubiquitous computing technology. Processing of data from video camera requires either a powerful and energy-consuming general purpose processor or an application-specific integrated circuit (ASIC), which is better suited for embedded applications. Field-programmable gate array (FPGA) technology enables rapid development and hardware prototyping of video processing ASICs. The FPGA devices are commonly found in smart camera architectures [1].

The technology is available, but it is not widely adopted due to the complex design process and the cost of specialized tools. A digital system design flow starts with hierarchy of circuit components containing register transfer level (RTL) description of the digital circuit in one of the hardware description languages VHDL or Verilog. The RTL circuit model is used for logic synthesis as well as simulation of the design. The synthesized design runs through imple-

mentation phase (mapping to the FPGA structures, place, and route, optimization), finally producing bitstream that is downloaded to the device. Compiling of the FPGA design and circuit debugging can be very time-consuming processes and large amount of logic cells and flip-flops in modern FPGAs call for an efficient design flow.

There are several solutions to speed up the circuit component development compared to the traditional RTL approach. Domain-specific environments, for example LabVIEW or System Generator for Digital Signal Processing in Matlab/Simulink, can be used to compose signal processing algorithm and produce RTL code. The other approaches are raising the abstraction level by using high-level software languages (C, SystemC, HandelC) and performing high-level circuit synthesis from software description [2]. The high-level tools are either in the research domain or in the commercial tools (for example, Vivado High Level Synthesis from Xilinx) and are targeting a variety of applications. There is still a lot of work for a circuit designer to adapt the synthesized IP components to the specific stream video processing architecture and application [3]. We also consider that open design environment is important for educational purpose and for wide adoption of the programmable technology [4].

In this chapter, we propose to use a high-level programming language Python to produce generators of video processing components. The Python language is efficient for algorithm development and visualization and provides modules for description of concurrent digital systems with automatic conversion to RTL. The Python scripts can be used to produce generic image processing intellectual property (IP) components that are customizable beyond the scope of the RTL languages. We introduce the design methodology and present the preliminary results.

In the next section, we present data processing and storage requirements in embedded video systems and examples of video processing chain. We show that even low end programmable devices from two major FPGA vendors, Xilinx and Altera, have enough resources and speed for real-time implementation of video processing chain.

In further sections, we give an overview of hardware description packages in Python and our IP generator module. The IP generator classes are used to automate several tasks and data transformations in the streaming hardware component development process. We show the usage of IP generator module on design examples.

2. Embedded video processing

Embedded video processing devices are part of smart cameras used in computer vision systems. Table 1 summarizes data processing and storage requirements for various video sources.

The minimal requirements for digital video processing systems are defined by standard definition PAL or NTSC video signals. Digital video stream from standard definition camera in ITU-R BT.656 format [5] is a sequence of 8-bit or 10-bit data words transmitted at 27 MB/s.

http://dx.doi.org/10.5772/61136

The video frame resolution for PAL camera is 720 by 576 pixels and one image frame occupies 405 kB of memory.

Video signal parameter	**PAL TV**	**WVGA**	**HD camera**
resolution [H x V]	720 x 576	752 x 480	2048 x 2048
refresh [fps]	25	63	178
data rate [MB/s]	27	27	760
frame memory [MB]	0.4	0.3	4

Table 1. Parameters of video signals

The next two columns present requirements for two commercial computer vision cameras: high-speed, low–resolution, and high-definition (HD) camera from Optomotive. The first has Wide Video Graphics Array (WVGA) sensor that has equal data rate and similar resolution to PAL camera but substantially higher refresh rate. The digitized PAL video stream contains long synchronization sequences due to the legacy TV transmission signal timing. On the other hand, the computer vision camera consumes most of the 27 MB/s bandwidth for data transmission and thus achieves refresh rates up to 63 frames per second (fps).

Characteristics of the HD camera from Table 1 show substantially higher resolution and roughly 10 times more memory storage per image frame. The sensor peak data rate of 760 MB/s enables image refresh rates of up to 178 fps.

The data rate of a digital video stream and the required amount of operations per image pixel are beyond the capacity of an embedded microprocessor. The video processing operations should be implemented in application-specific hardware. The blocks of the video processing chain operate concurrently in hardware and transformation operations inside each block can be parallelized.

2.1. Video processing chain

Video and image processing algorithms contain a sequence of data transformation operations. When the same sequence is applied to all image data, we can describe the algorithm as video processing chain. The processing chain clearly describes dataflow and can be used as template structure for hardware implementation. Structure of the data and the transformation operations depend on the video processing application [6].

Two examples of video processing chain are presented in Figure 1: (a) edge detection chain and (b) feature extraction chain for human detection. The edge detection transformations produce images that emphasize edges and transitions. The presented processing chain begins with contrast enhancement. The output pixel values are computed from the input pixels using contract stretching equation or lookup table. Next operations are gradient filters that produce

local horizontal (Gx) and vertical (Gy) gradient data. The gradient data stream is aligned, combined in a single stream, and sent to gradient magnitude computation block.

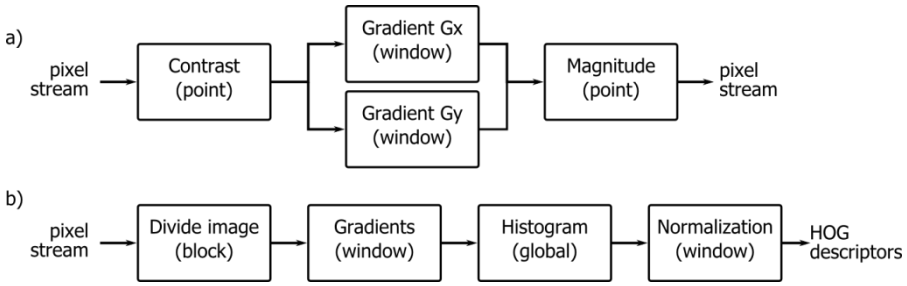


Figure 1. Examples of video processing chains

The feature extraction chain in Figure 1b transforms image pixel data to a set of descriptors used for classification and recognition in computer vision systems. The presented processing chain is based on histogram of oriented gradients (HOG) [7]. The image is first divided into blocks and gradients of the block pixels are computed. Histograms of gradient magnitude for spatial orientations are calculated next and block normalization is performed. The resulting data are image descriptors that can be used in the classification of algorithm to detect human locations.

We divide the video stream processing operations into four categories: point operations, sliding window operations, image block operations, and global operations:

- Point operations take one image pixel at a time and produce output values based on the current input pixel value. Examples of point operations are contract enhancement, image binarization (thresholding), and color conversion. These operations are relatively straightforward for hardware implementation using pipelined arithmetic operations or lookup table approach.
- Sliding window operations or local operations use a local neighborhood of pixels to produce the output. Examples of the sliding window operations are convolution-based operations for image filtering or calculating image gradients. The hardware implementation requires first-in first-out (FIFO) buffers to generate the neighborhood of pixels and pipelined arithmetic operations.
- The image block operations first divide the image frame into smaller blocks and then apply operation to the whole block of pixels. For example, HOG feature extraction divides the image into 64 × 128 pixel blocks. The hardware implementation of block operations on the video stream requires image buffers and blocks memory control logic. The implementation can exploit block-level parallelism in order to achieve the required throughput.
- Global operations require double buffering of the whole frame or block data, since there is no defined locality and any input pixel can be used to calculate the output. Example of global

operations is histogram calculation. These operations are most difficult to implement in the custom hardware at a reasonable cost and are typically implemented as a combination of software control and hardware accelerator.

Considering implementation of the video processing chain in hardware, we begin with initial digital system partitioning according to the dataflow and video transformation operations. The operations present hierarchical circuit blocks that can be reused for different applications.

3. Video processing in FPGA devices

The FPGA technology is used as a programmable alternative to ASICs. The FPGA devices can exploit high degree of data processing parallelism that is necessary for real-time video processing. The programmability of FPGA devices has many benefits in video processing applications due to the constant evolution of new algorithms and standards. This technology is well suited for smart cameras, where the image sampling and application-specific preprocessing are performed before data transmission to the host [8].

A drawback of FPGA devices is relatively high cost compared to massive produced ASICs and relatively complicated design flow compared to microprocessors. The programmable technology grows from simple logic replacement devices introduced in the late 1980s to powerful contemporary generic computation devices with plenty of interfaces, memory, and data processing resources. Even the smallest and low-cost devices today include static memory blocks and hardware support for fast arithmetic operations. In order to leverage usage of FPGA in embedded image processing applications, we consider low end devices and propose a novel design flow.

Table 2 presents characteristics of low end FPGA devices from two major manufacturers: Xilinx and Altera. A range of available resources in terms of logic cells, data flip-flops, embedded memory blocks, and hardware multiplier blocks are presented for FPGA families Cyclone IV and Max 10 from Altera and Spartan-6 and Artix from Xilinx.

FPGA family	**Altera**	**Altera**	**Xilinx**	**Xilinx**
resource	**Cyclone IV**	**Max 10**	**Spartan 6**	**Artix**
logic cells [k]	6 – 149	2 – 50	3.8 – 147	16 – 215
flip-flops [k]	6 – 149	2 – 50	4.8 – 184	20 – 269
memory [kB]	33 – 486	13 – 204	27 – 603	112 – 1642
multipliers	15 – 266	16 – 144	8 – 180	45 – 740

Table 2. Low end FPGA devices from Altera and Xilinx

FPGAs are best suited for signal processing algorithms based on arbitrary precision integer or fixed point operations. The low end families have the smallest amount of data processing

resources operating at a relatively low clock frequency (typically tens or few hundreds of MHz) due to programmability overhead. But even the smallest FPGA devices from the current families have enough resources and speed to implement real-time video image acquisition and some image processing functionality. Devices with more resources benefit from more parallelism that is important for HD or high frame rate video processing.

Emerging type of programmable integrated circuits are systems-on-chip (SoC), a combination of application-grade microprocessor and FPGA fabric. Both Altera and Xilinx provide SoC devices, Cyclone V and Zynq, which include dual core ARM Cortex-A9 processor. The smaller devices compete in terms of cost with the separate FPGA and microprocessor or microcontroller solution and benefit in tight coupling between processor and FPGA. These devices are suitable for embedded image processing applications that are partially implemented in hardware (HW) and in software (SW) and transfer data through microprocessor peripheral interfaces.

The design tools for programmable devices support component-based hierarchical design in order to manage development and verification of complex digital systems. The reusable blocks, called IP components, can be obtained from library of IP cores (Xilinx) or Megafunctions (Altera) or described using hardware description languages. Basic operations and components using specific FPGA structures are available for free, but a lot of IP cores can be obtained only after purchase.

While using IP cores for digital system shortens the design cycle, there is still a huge gap between the algorithm development and the circuit development. In the video processing algorithm development process, we consider new combinations of operations and different data partitioning to get an optimum processing result. New operators are probably not available as IP cores and need to be designed from scratch. The algorithm developer can also consider the cost of the hardware implementation of the operations in the design process. The algorithms are developed with tools and environments that offer strong mathematical and visualization support in order to get quick proof of concept. The tools are either commercial Matlab or LabVIEW or based on computer languages C/C++ (OpenCV) or Python.

3.1. Verification on development boards

For video processing hardware development, we can use either a computer vision smart camera with programmable device or a programmable video development board. In order to introduce design of embedded video processing in university laboratory practice, we developed video interface modules that can be used with low-cost commercial FPGA development boards. This solution is more affordable and the interface modules can be reused when the FPGA vendors offer new development boards based on new families of FPGA devices.

The video input interface module contains video decoder TVP5150A connected to PAL camera module, as presented in Figure 2. A small FPGA is used for the decoder setup, basic data preprocessing, and output video stream configuration. The video decoder TVP5150A converts analog video to the video stream in TU-R BT656 format. The FPGA device Xilinx XC3S50A performs stream decomposition and color space conversion and generates data stream. The

http://dx.doi.org/10.5772/61136

board has a 40-pin parallel video stream connector and a 12-pin serial PMOD for connection to low-cost Xilinx Digilent boards.

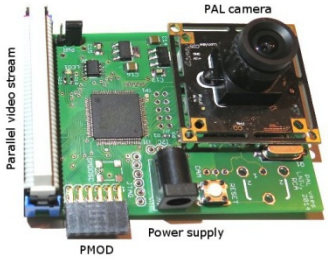


Figure 2. Photo of video interface board

The Digilent FPGA development boards for Xilinx FPGAs contain a computer graphics VGA or HDMI connector that can be used as a visual output for the image processing application. If there is no video output on the board, we can always add a simple extension module with the computer graphics or standard video output.

4. Hardware description in Python

The open-source Python community offers several packages covering digital circuits design. We tested three packages that use Python as a hardware description language [9].

Chips [10] is an HDL Python library that provides a language for designing hardware devices. Chips package introduces a stand-alone synthesizable language built on top of the Python allowing designers to work at a higher level of abstraction. The language provides methods for concurrent elements synchronization and data stream processing. The tool generates synthesizable RTL code using state machines or optimized soft-core processor automatically. The described device can be natively simulated in Python.

Migen [11] is another Python-based tool for building complex digital hardware. The toolbox introduces inside Python a new language FHDL for describing fully synchronous circuits. The language addresses limitations of standard hardware description languages: support for composite types and procedurally generated logic. The FHDL circuit description is on a higher abstraction level compared to the RTL languages and can be automatically converted to synthesizable Verilog. The simulation is supported through conversion and linked to external tools.

MyHDL [12] is an open-source package for using Python as a hardware description and verification language. The code can be converted to VHDL or Verilog automatically. The

introduced language MyHDL does not specifically target synchronous or stream processing circuits and is intended for the description of synchronous or asynchronous logic blocks. The converted code is readable, since it retains all the signal and component names and even block comments. The MyHDL supports native Python simulations and unit tests and can produce outputs in value change dump (VCD) format for graphical inspection. The package MyHDL provides basic RTL modeling concepts:

- Hardware-oriented data types: 1-bit bool and arbitrary length vectors intbv, modbv
- Support for synthesizable subset of arithmetic and logic expressions
- Combinational functions containing concurrent signal assignments
- Synchronous sequential functions with clock edge and optional reset
- Finite state machine abstraction
- Structural modeling.

Table 3 summarizes the basic features of the Python HDL tools. We found the tool MyHDL as the best choice for the development of reusable components due to nice modeling, conversion, simulation, and continuous support by the open-source community.

Package	Circuit model	Verification	Output	Current version
Chips	stream model, custom syntax	Python testbench	VHDL	0.1.2 (2011)
Migen	fragment model, custom syntax	External tools	Verilog	x (2012)
MyHDL	event driven, Python syntax	Python testbench and VCD	VHDL and Verilog	0.9 (2013)

Table 3. Python HDL Tool packages

4.1. MyHDL video graphics example

We will first present an example of a VGA graphics controller designed for ZynqSoC device. The controller is used to produce computer-generated video stream for hardware verification of the video processing chain on the FPGA development board. The controller presented in Figure 3 consists of device library components: Zynq PS, AXI interconnect, BRAM controller, Block memory, and custom components designed in MyHDL. The video stream generator components are VGA synchronization (VGAsync), video direct memory access (DMA), coordinate rotation, character memory, and color transformation.

We present the MyHDL code and hardware modeling constructs on a simplified example of the VGA synchronization generator:

http://dx.doi.org/10.5772/61136

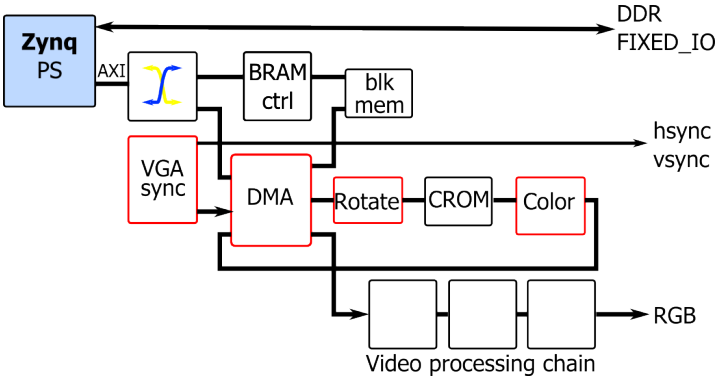


Figure 3. Video processing rapid prototyping components in ZynqSoC

```
from myhdl import *            # libraries and initialization code
HP=1040; VP=666
def VGAsync(clk, hsync, vsync):         # top level function and ports
        h = Signal(intbv(0)[11:])        # internal signals
        v = Signal(intbv(0)[10:])
        @always(clk.posedge)             # sequential function
def timing():
if h < HP-1:
                    h.next = h + 1
else:
                    h.next = 0
if v < VP-1:
                    v.next = v + 1
else:
                    v.next = 0
@always_comb                        # combinational function
def synchro():
hsync.next = 1 if h>=856 and h<976 else 0
vsync.next = 1 if v>=637 and v<643 else 0
return timing, synchro
clk = Signal(bool(0))             # port signal declaration
hsync = Signal(bool(0))
vsync = Signal(bool(0))
if __name__ == '__main__':     # conversion to VHDL
         toVHDL(VGAsync, clk, hsync, vsync)
```

The MyHDL hardware model contains declaration of signal objects (Signal) and functions describing combinational or synchronous sequential logic using Python decorators (@always). Integer values can be used for single bit signals (bool) as well as for bit vectors (intbv) which greatly simplifies the code compared to strict VHDL typing rules. The MyHDL hardware description requires less code compared to automatically generated or even handwritten VHDL or Verilog description. We can write a test bench using MyHDL objects and verify the operation of the design in Python. The verification is performed by printing the consecutive

signal values or dumping all the data to a timing waveform file. Python scripts can be written for automatic verification and unit testing.

The RTL circuit description requires careful design of the circuit parts with specified clock cycle behavior, such as interfaces. Figure 4 presents simplified algorithmic state diagram of the region movement DMA component. The state machine leaves initial IDLE state when a pixel read request (rdreq) or write request (wrreq) is received. The read request is asserted for each image line, and during video blanking period the controller returns to IDLE state. When the controller receives write request and is not reading the memory, it starts data movement by first getting the data from one bus (state GET) and then writing the data to frame memory (state PUT). The internal counter dxy is used to repeat these cycles and move 64 data pixels. The data movement cycle can be interrupted by read request and continued when the rdreq is de-asserted.

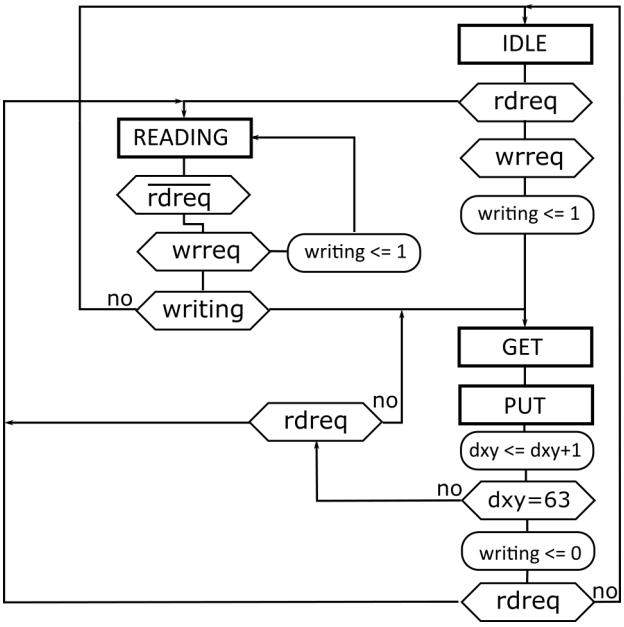


Figure 4. Video Direct Memory Access (DMA) controller state diagram

The RTL implementation of the DMA requires the designer to specify all the control and status signals of the data busses involved in the communication. For example, the MyHDL description of IDLE and GET is stated as follows:

```
if st==tst.IDLE:                # bus1 write request
if stb_i&we_i:
        writing.next=1
        stb2_o.next=1; we2_o.next=0;     # start reading cycle on bus2
```

```
        st.next=tst.GET
...
elif st==tst.GET:
        stb2_o.next=1; we2_o.next=0      # continue reading on bus2
if ack2_i:                          # if read acknowledge
        stb2_o.next=0                    # stop reading bus2 and
        stb3_o.next=1; we3_o.next=1      # start writing cycle on bus3
        dat3_o.next=data;
        adr3_o.next = concat(adrH, adrL)
        st.next=tst.PUT
```

The presented circuit description is time consuming and error-prone process and can be avoided by using higher level of abstraction for the specific domain.

5. IP component generator in Python

An IP component generator written in Python language can be used to raise the MyHDL hardware description level of abstraction and target video stream processing IP components. The proposed design flow is presented in Figure 5. We created a collection of Python classes in IPgen module used for object-oriented hardware description. The IP designer creates the IP generator script file composed of IPgen objects and methods. When the script is executed, transformations generate IP description file with RTL circuit description in MyHDL. This Python file can be used for functional IP verification with test bench and automatic conversion to HDL code.

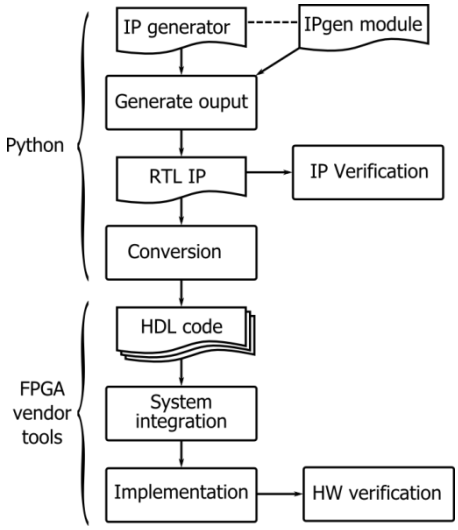


Figure 5. Python based IP generator design flow

The resulting HDL code presents RTL description of one IP in the video processing chain. The FPGA vendor tools are then used to package IP, integrate IP in the video processing hardware system, and perform circuit implementation. The implemented circuit is downloaded to FPGA for hardware verification on video development board.

5.1. IP generator module

First, we present semantics of object-oriented hardware description in our IP generator module. We define the IP component as a tuple where *IPname* stores top level function name, *IPinit* defines initialization code, and *IPinterface* defines interface type. The IP component contains a set of functions and a set of signals.

Statements are a collection of MyHDL statement code and identifiers. The identifiers are used in code transformation process. The set of signals is defined with name, type, and scope for each signal. The signal type is one of the MyHDL provided types. The scope defines the position of the signal declaration:

- External signals are part of initialization code.
- Port signals are on port description of the top level function.
- Internal signals are declared inside top level function.

A function transformation converts function to a set of generated functions and signals:

$$generate(\mathcal{F}) \rightarrow \left\{\left(name_0, \mathcal{F}_{g0}, \mathcal{S}_{g0}\right), \ldots, \left(name_k, \mathcal{F}_{gk}, \mathcal{S}_{gk}\right)\right\}$$

The generated functions are objects of the basic combinational or sequential MyHDL functions and can be converted to MyHDL code using *getcode()* method. The generated signal objects have *declare()* method that outputs appropriate signal declarations. The algorithm for producing MyHDL IP description is presented in pseudocode:

```
init_code = IPinit                    # get initialization code
(iname, if, sig) = IPif.generate()    # generate interface logic
F.append(if)                           # and append to functions list

for fun in F:                         # for each function generate
     (fname, ff, fsig) = fun.generate()   # list of function names
     lname += fname;                      # list of function names
     sig += fsig;                     # list of signals
     code += ff.getcode()             # and function code

for sig in S:                         # for each signal generate
if sig.stype==port:
     port_code += sig.name;               # list of port names and
     psig_decl += sig.declare()           # port declarations
else:
     sig_decl += sig.declare()            # internal signal declarations
```

http://dx.doi.org/10.5772/61136

```
print init_code                    # print out initialization code
print "def" + name + port_code      # top level function and ports
print sig_decl                     # signal declarations
print code                         # function code
print "return" + lname              # top level return
print psig_decl                    # port signal declarations
print conversion function          # and conversion to VHDL/Verilog
```

The structure of our stream video processing domain module is presented in UML diagram in Figure 6. Python classes are provided for description of the IP component that is an object of type IPgen. The object IPgen is a collection of signal and function objects defining the IP hardware structure. A name property is used for MyHDL top level function as well as for output file name for MyHDL circuit description. The interface property provides various automatically generated IP component interfaces for streaming video components.

The IPgen signals are external or internal signals described in integer (si) or binary (sb) signal objects. The classes provide declaration method that is used to generate MyHDL signal declarations. The IPgen class *function()* is basic class for description of the IP behavior. The *code()* method is used to add MyHDL statements into statement code list (C[]). The function code is translated into a set of MyHDL combinational and sequential functions using *getcode()* method. All classes derived from the *function()* should provide the *generate()* method that returns a triple containing list of MyHDL function names, list of basic functions objects, and list of signals.

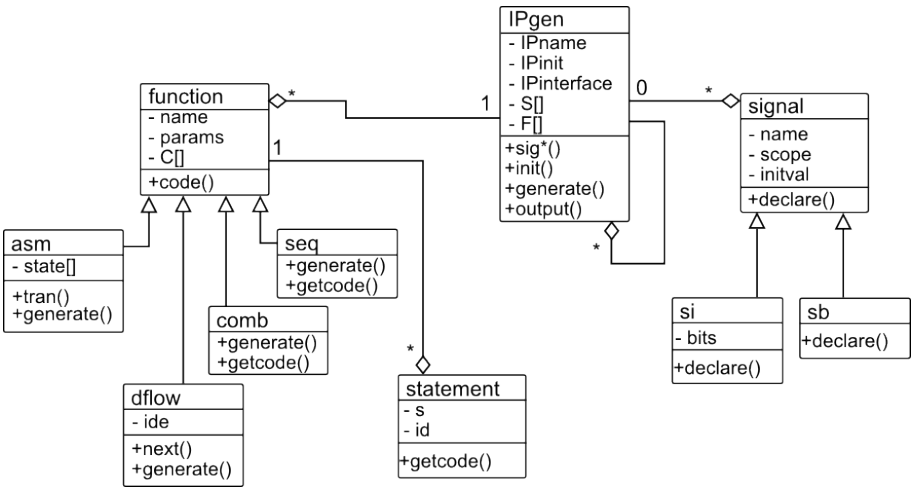


Figure 6. UML diagram of Python IP generator module

The presented generic MyHDL IP generator module can be extended by domain-specific classes describing interface and functions objects. We developed extensions for streaming video processing applications that are presented in the design examples.

5.2. IP generator design examples

We present IP description of square root operation on integer data that is commonly found in image processing transformations. The code is based on modified Dijkstra's square root algorithm [13]. The square root in plain Python code is calculated in a loop:

```
def sqr(number, bits):
     mask = 1 << (bits-2)
     r = 0
     I = 1
     rem = number
while (mask>0):

if ((r+mask)<=rem):
           rem -= r+mask
           r += 2*mask
       r >>= 1
       mask>>= 2

return r
```

The loop has a constant number of iterations that can be calculated from the input number bit size, for example, 8-bit numbers require 4 loop iterations. The square root of an 8-bit input data can be calculated as a sequence of operations presented in Figure 7a. The combinational functions of the corresponding MyHDL circuit description are given in Figure 7b.

The circuit description can be generated with a script using IP generator classes. We first define IPgen object and declare arrays of vector signals in dataflow diagram:

```
bits = 8
w = (bits-2)/2 + 1
ip = IPgen("sqr", IFpoint) #set the IP name and interface type
ip.siga('r', w, bits)
ip.siga('m', w, bits)
ip.siga('rem', w, bits)
```

A member function siga() is used for declaration of multiple signal instances; for example, a function call siga('r', 4, 8) produces the following MyHDL declaration:

```
r0, r1, r2, r3 = (Signal(intbv(0)[8:]) for i in range(0,4))
```

The initialization logic is described as:

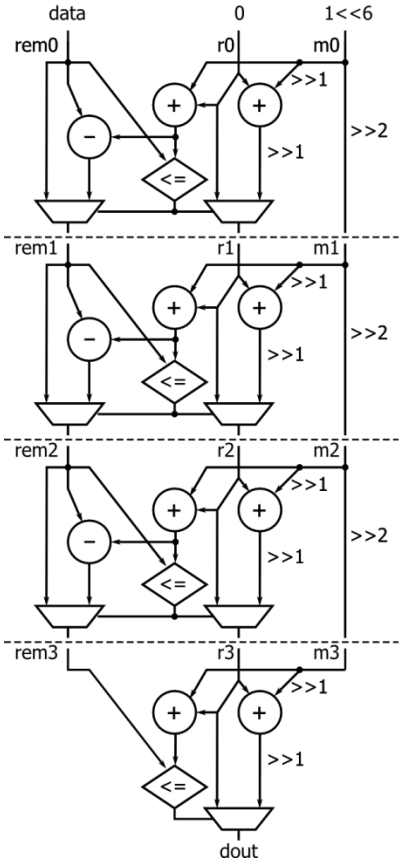
```
log = comb("init_logic")
log.code("rem0.next = data")
log.code("r0.next = 0")
log.code("m0.next = 1 << {0}".format(bits-2))
```

The code for dataflow object is generated in a for-loop:

```
p = dflow("p",DFcomb)
for i inrange (0, w):
```

http://dx.doi.org/10.5772/61136

a)



b)

```
@always_comb
def init_logic():
    rem0.next = data
    r0.next = 0
    m0.next = 1 << 6

@always_comb
def p_logic():
    if (r0+m0) <= rem0:
        rem1.next = rem0-(r0+m0)
        r1.next = (r0 + 2*m0 ) >> 1
    else:
        rem1.next = rem0
        r1.next = r0 >> 1
    m1.next = m0 >> 2
    if (r1+m1) <= rem1:
        rem2.next = rem1-(r1+m1)
        r2.next = (r1 + 2*m1 ) >> 1
    else:
        rem2.next = rem1
        r2.next = r1 >> 1
    m2.next = m1 >> 2
    if (r2+m2) <= rem2:
        rem3.next = rem2-(r2+m2)
        r3.next = (r2 + 2*m2 ) >> 1
    else:
        rem3.next = rem2
        r3.next = r2 >> 1
    m3.next = m2 >> 2
    if (r3+m3) <= rem3:
        dout.next = (r3 + 2*m3 ) >> 1
    else:
        dout.next = r3 >> 1
```

Figure 7. a) 8-bit integer square root dataflow, b) MyHDL combinational square root description

```
        p.code("if (r{0}+m{0}) <= rem{0}:".format(i))
    if (i == w-1):
            p.code(" dout.next = (r{0} + 2*m{0}) >> 1".format(i))
    else:
            p.code(" rem{1}.next = rem{0}-(r{0}+m{0})".format(i, i+1))
            p.code(" r{1}.next = (r{0} + 2*m{0}) >> 1".format(i, i+1))
            p.code("else:")
    if (i == w-1):
            p.code(" dout.next = r{0} >> 1".format(i))
    else:
            p.code(" rem{1}.next = rem{0}".format(i, i+1))
            p.code(" r{1}.next = r{0} >> 1".format(i, i+1))
            p.code("m{1}.next = m{0} >> 2".format(i, i+1))
            p.next()
```

In the dataflow loop description, we call a method next() to mark computation stages denoted with a dashed line in Figure 7a. The stages are used in the automatic generator of RTL combinational or sequential dataflow descriptions. The dataflow object accepts parameter that defines the dataflow synthesis mode with the following values:

- DFcomb for combinational logic implementation (for example in Figure 7b)
- DFreg for combinational description with a register in the last stage
- DFpipe for sequential description with registers in all stages.

The RTL IP component description in MyHDL is generated by

```
ip.output(log, p)
```

By changing the parameter in the dataflow object, the user can quickly produce different versions of the circuit and choose the version that satisfies timing and area constraints. The output description includes generated interface control logic for the video stream IP.

The IP core interfaces in digital systems are either generic function-specific interfaces or standard bus interfaces. The standard bus interfaces are based on proprietary processor bus architectures (Avalon, AXI, PLB) or open architectures for digital systems (Wishbone). We consider using a simplified version of WISHBONE [14] interface specifically tailored for video data stream transmission. The video stream interface should provide variable size pixel data bus, clock, and data strobe signal used to mark active data transmission cycle. The proposed video bus timing waveform for the interface type IFpoint is presented in Figure 8.

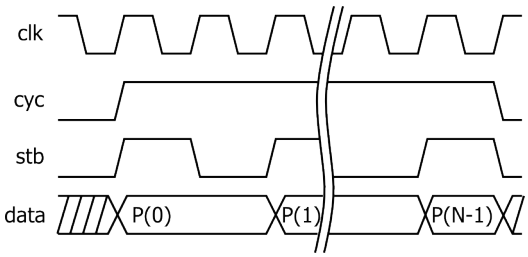


Figure 8. Video data stream bus timing waveform

The interface signals and control logic are automatically generated. The control logic is used to count pipeline cycles, produce output strobe for valid output data, and provide additional cycles to finish the pipeline computation of the data from the input burst. The generated interface code for four-stage pipeline of 8-bit square root circuit is

```
        # Interface internal signal declarations
        pipe = Signal(bool(0))         # enable additional cycles
        cycEnd = Signal(bool(0))        # end output cycle
        cycnt = Signal(intbv(0)[8:])    # cycle counter
...
```

http://dx.doi.org/10.5772/61136

```
          # Interface control logic
          stbo.next = 0; pipe.next = 0; cycEnd.next = 0
if stb and cyc:                  # count pipeline cycles
if cycnt == 3:
               stbo.next = 1
               cyco.next = 1
else:
               cycnt.next = cycnt + 1
elif (not stb) and (not cyc): # additional pipe cycles
if cycnt> 0:
               cycnt.next = cycnt - 1
               pipe.next = 1
if pipe:
          stbo.next = 1
if cycnt == 0:
               cycEnd.next = 1
if cycEnd == 1:
          cyco.next = 0
```

The initial Python description of the square root calculation is 35 lines of code and is automatically transformed to 72 lines of MyHDL and 126 lines of VHDL or Verilog code.

The next example is a sliding window operation for image filtering. The filtering is performed by a discrete convolution of the input data with filter coefficients C_0, C_1,...,C_n. Horizontal convolution equation in Z domain:

$$P_{out} = C_0 P + C_1 P z^{-1} + C_2 P z^{-2} + \cdots + C_n P z^{-n}$$

The convolution can be decomposed into multiply–add operators and delay elements (z^{-1}). The convolution core is generated with one combinational and one sequential function:

```
n = 3
c = [1, 2, 1]
log = comb("filt_log")
reg = seq("filt_reg")
reg.code("if en:");
for i inrange (0, n):
     log.code("r{0}.next = s{0} + {2}*data".format(i,i+1,c[i]))
if (i < n-1):
          reg.code(" s{1}.next = r{0}".format(i,i+1))
```

For implementation of the filter in stream processing IP component, we have to add interface logic and consider boundary conditions in computation of the output values. Pixels outside image boundary should have zero value in convolution computation. To insert this data into convolution circuit core, we add an algorithmic state machine (ASM) driving data multiplexer and input FIFO buffer. The dataflow structure of the stream processing filter is presented in Figure 9.

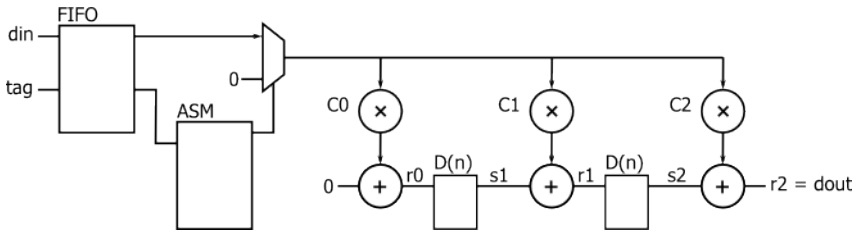
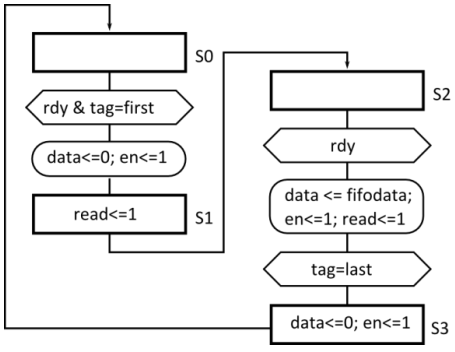


Figure 9. Dataflow diagram for convolution filter with 3 coefficients

The image boundary pixels are marked with additional tag signal on the input. When the first line pixel is received, the ASM switches input multiplexer to zero value and starts computation cycles. The FIFO is used to hold incoming pixels before they are used in convolution computation. Similar sequence is performed for the last line pixel. State diagram and the corresponding IP generator code of the ASM is presented in Figure 10.




```
a=asm("st","S0")
a.default("en.next=0; read.next=0")

a.tran("S0", "", "rdy and tag==FIRST",
 "data.next=0; en.next=1", "S1")
a.tran("S1", "read.next=1", "", "", "S2")
a.tran("S2", "", "rdy", "data.next=fifodata;
 en.next=1; read.next=1", "")
a.tran("S2", "", "rdy and tag==LAST","","S3")
a.tran("S3", "data.next=0; en.next=1", "",
 "", "S0")
```

Figure 10. Algorithmic state machine for the convolution controller

The ASM states and state transitions are described with tran() method using five parameters: current state, state logic, transition condition, transition logic, and next state. For example, in state S0 the transition condition is FIFO ready signal (rdy) and tag that marks the first pixel in a line. If the condition is satisfied, the data signal for the convolution core is set to 0 and one convolution cycle is enabled (en). The default value of control signals en and FIFO read is 0 and the signals are activated only in specified state or transition logic. The ASM description is automatically converted to MyHDL sequential function and state machine signal declarations.

The FIFO buffer description and interface logic is generated by setting sliding window interface in IPgen object. The generated MyHDL code for convolution filter with three coefficients contains 115 lines and is transformed to 160 lines of VHDL or Verilog. Vertical convolution is computed by replacing the delay registers D(n) in the convolution core with

circular buffers for one image line. The ASM logic should be changed to detect first and last image line and replace the boundary lines with zero values.

6. Conclusion

We presented unified embedded video processing circuit design flow in the Python language that speeds up the design cycle and automates error-prone domain conversion process. Python scripting language is a good choice for testing domain-specific algorithm and data transformations using open-source packages. We selected the package MyHDL for RTL hardware description and automatic conversion to readable VHDL or Verilog code. The advantage of MyHDL in comparison to traditional hardware description languages is the ability to use all the scripting power of the language for unit testing or simulation verification and to raise the level description abstraction in the same environment. The drawbacks of MyHDL code are limited support of some advanced HDL features from standard language packages and complex error reporting. We overcome this limitation by using MyHDL as intermediate language and use the proposed IPgen module for the streaming IP component hardware description.

The IPgen module has generic object structures used for generating MyHDL components with combinational and sequential functions and domain-specific streaming functions. Code transformations in the output MyHDL generation process are used to determine the produced circuit timing behavior and add video processing interfaces, components, and signals. We presented examples of square root calculation and convolution filter designs. For the future work, we will extend the IPgen classes to include more interface options and provide resource mapping transformations.

Author details

Andrej Trost* and Andrej Žemva

*Address all correspondence to: andrej.trost@fe.uni-lj.si

Faculty of Electrical Engineering, University of Ljubljana, Ljubljana, Slovenia

References

[1] A.W. Malik, B. Thoörnberg, and P. Kumar. Comparison of Three Smart Camera Architectures for Real-Time Machine Vision System. *Int J Adv Robot Syst*. 2013;10(402): 1–12. DOI: 10.5772/57135

[2] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 2011;30(4):473–491.

[3] J. Serot, F. Berry, and S. Ahmed. Implementing Stream-Processing Applications on FPGAs : A DSL-Based Approach. In: *Field Programmable Logic and Applications, FPL*; 2011; Chania, Crete, Greece. pp. 130–137.

[4] A. Trost and A. Žemva. Teaching Design of Video Processing Circuits. *International Journal of Electrical Engineering Education*. 2012;49(2):170–178. DOI: 10.7227/IJEEE.49.2.7

[5] International Telecommunication Union. Recommendation ITU-R BT.656-5 [Internet]. 2007. Available from: http://www.itu.int/ [Accessed: February 2015].

[6] D.G. Bailey. *Design for Embedded Image Processing on FPGAs*. John Wiley & Sons (Asia) Pte Ltd, Singapore; 2011. 416 p

[7] N. Dalal and B. Triggs. Histograms of Oriented Gradients for Human Detection. In: *Histograms of Oriented Gradients for Human Detection*; 2005; IEEE Computer Society Washington. pp. 886–893. DOI: 10.1109/CVPR.2005.177

[8] J. Sérot, F. Berry, and C. Bourrasset. High-Level Dataflow Programming for Real-Time Image Processing on Smart Cameras. *Journal of Real-Time Image Processing*. 2014;:1–13. DOI: 10.1007/s11554-014-0462-6

[9] A. Trost. Design of a Graphical Controller with Reusable Components. In: B. Zajc and A. Trost, editors. *Proceedings of the 22nd international Electrotechnical and Computer Science Conference* ERK 2013; September 2013; Portorož, Slovenia. p. 31–34.

[10] J. Dawson. Chips—Hardware Design in Python [Internet]. 2011. Available from: http://dawsonjon.github.io/chips/ [Accessed: February 2015].

[11] S. Bourdeauducq. Migen Manual [Internet]. March 2014. Available from: http://m-labs.hk/migen.pdf [Accessed: February 2015].

[12] J.I. Villar,J. Juan, M.J. Bellido, J. Viejo, D. Guerrero, and J. Decaluwe. Python as a Hardware Description Language: A Case Study. In: *VII Southern Conference on Programmable Logic (SPL)*; 2011; Cordoba. pp. 117–122.

[13] M.T. Tommiska. Area-Efficient Implementation of a Fast Square Root Algorithm. In: *Proceedings of Third IEEE International Caracas Conference on Devices, Circuits and Systems*; 2000; Caracas. pp. S18/1–S18/4.

[14] OpenCores. WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores [Internet]. 2002. Available from: http://cdn.opencores.org/downloads/wbspec_b3.pdf [Accessed: February 2015].