Math2Mat: from Octave/Matlab to VHDL

Yann Thoma*, Etienne Messerli†, Michel Starkier*, Daniel Molla*, Sebastien Masle*, Christophe Bianchi*, Oliver Gubler†, Claude Magliocco‡, Philippe Crausaz‡, Samuel Tâche‡, Denis Prêtre§ and Gregory Trolliet¶

†HES-SO // Valais, Sion, Switzerland
‡EIA-FR, Fribourg, Switzerland
§HE-Arc, Neuchâtel, Switzerland
¶hepia, Geneva, Switzerland

*Reconfigurable and Embedded Digital Systems Institute - REDS
HEIG-VD // School of Business and Engineering Vaud; HES-SO // University of Applied Sciences Western Switzerland
CH-1400 Yverdon-les-Bains, Switzerland

Corresponding author: Yann Thoma. E-mail: name.surname@heig-vd.ch. Phone: +41245576273. Web: http://reds.heig-vd.ch

Abstract—Math2Mat aims at automatically generating a VHDL description of a mathematical description written in Octave/Matlab. The generation creates a synthesizable RTL description using floating point operators (32 or 64 bits) combined in a fully pipelined way. Emphasis is put on the throughput attainable by the design, especially in the “for loop” implementation. The software also offers a graphical user interface, letting the developer manage the different parameters before generation. Verification can also be launched from the software, a SystemVerilog testbench being automatically generated.

I. INTRODUCTION

The design of digital systems tends to be more and more under pressure of time constraints for rapid realization of prototypes and final hardware. Developing in a pure VHDL/Verilog description scheme may not necessarily meet these timing requirements. In that context, High-level Synthesis tools (HLS) are gaining interest in the world of embedded systems. Commercial solutions are available, with some C to HDL languages/tools like CoDeveloper [1] of Impulse Accelerated technologies, Handel-C [2] of Celoxica, or Catapult C [3] of Mentor. Other tools offer other types of translation, like SystemC [4] to VHDL (CoCentric of Synopsys) or Matlab/Simulink to DSP (System Generator of Xilinx [5]). Even Altera is now promoting an OpenCL to VHDL project [6]. While these tools are very generic, the project presented in this paper focuses on the generation of mathematical expressions.

At the time the project started, some tools offered automatic synthesis of fixed-point arithmetic hardware, like in the MATCH project [7], [8], but the support for floating point was not really present, except in the tool TANOR [9], which was developed around Xilinx floating point IP cores, and GraphLab HLS [10]. This is mainly due to the fact that a couple of years ago the FPGAs were not really supposed to deal with a considerable number of floating point operators. The amount of hardware resources found on the FPGAs has dramatically increased in the last years, it is now time to go for floating point arithmetic, notably by taking advantage of hardware DSP blocks.

The Math2Mat project aimed at designing a software tool allowing a user to automatically generate a synthesizable VHDL design, starting from a pure Octave/Matlab description. The computation supports 32 or 64-bits floating point operations. A SystemVerilog testbench is also generated, in order to validate the design by exploiting the initial Octave/Matlab description. The Octave syntax was chosen because of its compatibility with Matlab (one of the main mathematical tools used by scientists), and because Octave is a free open source software, as is Math2Mat.

The next section presents the global structure of Math2Mat, by giving an overview of its capabilities. Section III shows the structure of the basic operation blocks, while Section IV explains in detail the structure of the hardware design generated by the tool. The verification of the design is shown in Section V, and its validation on a real hardware in Section VI. Finally the conclusions and future work can be found in Section VII.

II. MATH2MAT PURPOSE

The goal of the Math2Mat tool is to automatically generate a VHDL design corresponding to an Octave/Matlab description. A special emphasis was put on the pipelined structure of the generated hardware, allowing up to one full operation per clock cycle. Currently it supports the following 32 and 64-bits floating point operations:

1) Addition
2) Subtraction
3) Multiplication
4) Division
5) Square root

This set of operations might easily be extended for the need of a particular application, the structure of the software being flexible enough to accept new operators.

The accepted control structures are the following:

1) if/then/else
2) for loop

Math2Mat can process nested structures combining both of these control elements. The first version of the tool does not support other control structures, but as with the operators, new controls might be added later.
The code shown in Figure 1 illustrates a possible input Octave code, illustrating the support of multiple inputs/multiple outputs, nested control structures, a for loop, and the reaffectation of a variable.

```
function [s1,s2]=example(a,b)
    s1 = 0;
    for i=1:1:10
        if (b<10)
            s1=s1+a;
        else
            s1=s1-a;
        end
    end
    s2 = s1+b*4;
endfunction
```

Fig. 1. Supported Octave code example

Math2Mat has been written in Java and can be launched from the command line. A standard VHDL design can be directly generated from the Octave/Matlab description, without user intervention. The operators are automatically chosen to be pipelined, and the design deals with 32-bits floating point numbers. However some advanced settings can be exploited and in that case a Math2Mat project file containing a XML description of the useful data is required. In order to easily deal with all the possible settings, a graphical interface (see Figure 2) allows the user to interact with the tool. It is based on Eclipse and provides a more ergonomical way to perform some useful operations than directly writing into a XML file.

Some of the main features of the GUI tool are:

- An Octave/Matlab editor, with syntax highlighting.
- An Octave/Matlab parser that dynamically analyzes the code everytime the user types a letter. A message view allows the visualization of the erroneous lines in order to directly check the validity of the code.
- A dynamic structural view of the future hardware design. This view is automatically updated everytime the Octave/Matlab code changes and is valid. It allows to directly visualize the potential hardware structure.
- A selection of the operator for each operation, letting the user choose one of the supplied implementations.
- A single button that allows to generate the VHDL design, the SystemVerilog testbench, and to run the simulation. The simulation results are displayed in a Log view.

Math2Mat is free, open source, and can be downloaded at the following website: http://www.math2mat.ch.

III. BASIC OPERATORS BLOCKS

Math2Mat exploits basic blocks and combines them in order to generate the proper function. An operation can be implemented by different blocks, letting for instance a multiplication being combinational or fully pipelined. Through the Math2Mat GUI the user is then able to choose, for each operation, which hardware block should be used.

Each block is made accessible to the application by means of a Java class describing its behavior and structure. The most important fields required by the software are:

- The name of the operator or of a function (for instance \( \sin() \) or \( \cos() \)).
- The implementation type (combinational, sequential, or pipelined).
- The number of clock cycles necessary to get a valid output, especially useful to describe the pipeline depth.
- The data input rate, allowing to use a combinational block by forcing a valid calculation only every two clock cycles, the block being placed between registers.
- The name of the VHDL file containing the description of the block.

In order to be as generic as possible, every basic blocks has been built with respect to a standard interface (see figure 3).

The input data is associated with a \( \text{valid}_i \) signal, that has to be asserted in order to indicate the data is valid. The block itself can indicate that it is ready or not to accept data through the \( \text{ready}_o \) signal. For instance, a standard fully pipelined operator would let this signal be always asserted, while an operator that performs the operation in multiple clock cycles without pipelining would act on it according to its sequential process. Finally, a \( \text{stall}_i \) port allows to stall the entire block. This feature is required by the auto-delay mechanism presented in the next section.
During the Math2Mat project, operators have been designed for the following operations: addition, subtraction, multiplication, division, and square root. All of them have been implemented in 32 and 64-bits, according to the IEEE 754 format, with a "round to nearest" rounding mode and exception handling (NaN and ∞). Denormalized numbers are not currently supported by the operators.

Table I shows all the synthesis results achieved with the synthesizer Precision and the FPGA Virtex5 5VLX110FF676 (contains 17280 Slices).

IV. GENERATED STRUCTURE

The blocks presented in the previous section might be directly integrated into a specific design by hand. However this task may lead to some hard work and takes many hours. The main goal of Math2Mat is to take care of this integration by quickly combining these blocks into a very efficient structure. The blocks have been implemented in a combinational and pipelined manner, but the real emphasis of Math2Mat is the potential data rate allowing up to one function computation per clock cycle.

A. Pipeline and auto-delay

To achieve this goal, the generated structure is fully pipelined, and takes advantage of the inherent parallelism of hardware. Each operator (basic block) is encapsulated into a wrapper having an interface as shown in figure 4.

![Fig. 4. Interface of a basic block wrapper](image)

Each input and output data is associated with two control signals: valid and ready. For an input data, the valid signal allows the system to tell the block that the input data is valid, and the ready signal lets the block indicate if it is ready to accept new data. Basically, while ready is down, the system should keep the data unchanged, this data being changed only when ready is asserted (signifying that the data has been processed).

The advantage of this 2-signal approach resides in the fact that the output subsystem (the component that would be connected to the Math2Mat core) can indicate if it is ready or not to accept data, and thus control the flow. Deasserting the ready signal will cause the pipelines to stall by propagating the deassertion through registers, this backpropagation being done at the rate of one clock cycle per operator. Minimal FIFOs are integrated into the block wrappers, in order to keep the data unchanged during this process and in order to deal with data being requested by multiple operators. The backpropagation of the ready signal being done sequentially, registers must thus be present between the operator blocks. This allows to keep the highest operational frequency, every potential combinational path being cut. Therefore, even a huge design won’t suffer from long combinational paths that would lower the operating frequency.

This auto-delay mechanism allows to deal with one function computation per clock cycle if the output is capable of dealing with such data rate, or to go slower if needed.

B. Load balancing

As an example, the expression \( a + b + c + d \) can be interpreted as:

1. \( a + (b + (c + d)) \)
2. \( (a + b) + (c + d) \).

Passing from one representation to the other might be done automatically, by using standard techniques of tree manipulation, well known in the context of hardware design. As Math2Mat allows to directly observe the expected hardware structure while typing the Octave code, it seemed natural to let the user play with his code in order to generate the desired design. Following the example, depending on the code typed, the structure will be well-balanced or not.

Closely related to load balancing, the optimization of the tree might allow to deal with functions like the following:

1. \( ax^2 + bx + c \)
2. \( x(ax + b) + c \)

These representations are arithmetically equivalent, but the hardware resources and datapaths are clearly different. Math2Mat is currently being enhanced in order to automatically optimize the hardware design, an option offering it to the user.

C. Compensation FIFOs

Supporting one function computation per clock cycle seems to be an excellent idea. However some troubles may arise if, for example, the structure looks like the one presented in Figure 5. In the figure, the numbers left to the operators represent the number of clock cycles required by the data to reach this point, with the assumption that the operators show a latency of 10 clock cycles. In that case, the computation would require the values \( c \) and \( d \) to be supplied 10 cycles before \( a \) and \( b \), because of the operator Op3. A FIFO of depth 10 at the input of Op3 can therefore allow to supply \( b \) at the same time as \( c \) and \( d \). A second FIFO is also required at the input of Op4 in order to let \( a \) being read at the same time as the other inputs.

While load balancing is not automated, FIFOs insertion can be automatically managed by the tool, the optimal FIFO size...
TABLE I
SYNTHESIS RESULTS WITH THE SYNTHESIZER PRECISION AND THE FPGA VIRTEX5 5VLX110FF676.

<table>
<thead>
<tr>
<th>TYPE</th>
<th>Single precision</th>
<th>Double precision</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Slices</td>
<td>%</td>
</tr>
<tr>
<td>Add2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combinatorial</td>
<td>106</td>
<td>0.61%</td>
</tr>
<tr>
<td>Pipelined</td>
<td>122</td>
<td>0.71%</td>
</tr>
<tr>
<td>Mult2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wired multiplier</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combinatorial</td>
<td>27</td>
<td>0.16%</td>
</tr>
<tr>
<td>Pipelined</td>
<td>47</td>
<td>0.30%</td>
</tr>
<tr>
<td>Carry save adder</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combinatorial</td>
<td>288</td>
<td>1.66%</td>
</tr>
<tr>
<td>Pipelined</td>
<td>294</td>
<td>1.70%</td>
</tr>
<tr>
<td>Div2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRT4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combinatorial</td>
<td>211</td>
<td>1.22%</td>
</tr>
<tr>
<td>Pipelined</td>
<td>321</td>
<td>1.86%</td>
</tr>
<tr>
<td>Array of subtractors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combinatorial</td>
<td>263</td>
<td>1.52%</td>
</tr>
<tr>
<td>Pipelined</td>
<td>547</td>
<td>3.17%</td>
</tr>
<tr>
<td>Successive approximations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combinatorial</td>
<td>500</td>
<td>2.89%</td>
</tr>
<tr>
<td>Sqrt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRT2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combinatorial</td>
<td>112</td>
<td>0.65%</td>
</tr>
<tr>
<td>Pipelined</td>
<td>147</td>
<td>0.85%</td>
</tr>
<tr>
<td>Non-restoring</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combinatorial</td>
<td>183</td>
<td>1.06%</td>
</tr>
<tr>
<td>Pipelined</td>
<td>261</td>
<td>1.51%</td>
</tr>
<tr>
<td>Compare</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combinatorial</td>
<td>17</td>
<td>0.10%</td>
</tr>
<tr>
<td>Mult2 with a constant value (wired multiplier)</td>
<td>40</td>
<td>0.23%</td>
</tr>
</tbody>
</table>

being recursively calculated based on the chain of operators. Actually, the internal FIFOs of the operators would allow a correct calculation of the function, even without the compensation FIFOs, but at the cost of a reduced throughput, because some inputs would have to wait for the others. Therefore there is a tradeoff between the number of resources required for the FIFOs and the achievable throughput, and so the user can thus choose whether he wants FIFOs insertion or not.

D. Conditional statements

Math2Mat supports if/then/else constructs. These constructs can be nested, and are implemented using multiplexers in order to select the value that has to be stored into a variable. The main issue resides in the synchronization between the values and the control (derived from the condition). Compensation FIFOs are inserted at the right place in order to allow the maximal throughput. In general the values require more cycles than the condition in order to be ready, and so the FIFO is applied to the output of the condition computation. The advantage of this is clearly the fact that the output of the condition calculation is stored in one bit, as it is a boolean, instead of a 32-bit floating point variable.

E. For loop

The pipeline structure of the design should allow up to one function computation per clock cycle. However, a for loop can dramatically compromise this ideal case. For instance, the internal body of the loop presented in Figure 6 can be pipelined. However, the fact of requiring to go through this pipeline 10 times does not allow other data to come into the loop.

In Math2Mat the loop body is expected to be implemented in a pipelined manner. If that pipeline is composed of $n$ stages, up to $n$ computations might be performed in parallel. In that case, the first data is inserted at the first clock cycle, the second at the second, and so on, until the pipeline is full. At this time the first data will go back to the beginning of the pipeline and starts its second operation.

1This example does not show interesting computation, but is used for the sake of clarity.
If the *for loop* is composed of an iterator having the form `for i in 1:1:10`, i.e. fixed values for the iterator, then the data will leave the loop in the same order as their entrance sequence. This case can be quite easily implemented. However, for a code like the one presented on Figure 7, where the iterator depends on the value of the function inputs, the successive function computations will lead to different numbers of loop iterations. In that case, data will leave the loop in another order as their entrance, as the second computation can exit the loop before the first one, leading to a wrong appearance of the data at the output of the Math2Mat core.

This mechanism allows to know the number associated to any pipeline output computation, and can then be used to fill the reorder buffer.

Figure 9 presents the general structure of a *for loop*, that consists of three distinct parts:

- On the left, the first part allows the initialization and management of the loop inputs. There are two types of loop inputs: the inputs used in a read-only mode (in) and those used in a read-write mode (inout or out). In the listing of figure 1, `a` and `b` are pure inputs while `s1` is an inout variable of the loop.

  – The pure *in* variables are associated with FIFOs because they need to be accessed at every iteration of the loop.
  – The *inout* variables are used within the loop body and then reinjected into the loop body, up to the last iteration. At this point their value is stored into the reorder buffer.
  – The pure *out* variables are considered like *inout* variables, as their only difference concerns their
Fig. 9. General structure of a for loop

initial value being a constant instead of any kind of value.

- The second part represents the content of the loop (its body). It also encapsulates the control structure required to calculate its end condition.
- The third and final part, namely the reorder buffer, manages the results of the loop.

The different parts are closely linked. The remainder of this section presents their content and their interactions. The loop body is simply a pipeline responsible for calculation, but the blocks Init (at the left of Figure 9) and reorder buffer (at the right of Figure 9) are more specific.

The block Init includes two different types of blocks:

- The Init_Input block allows to select the value that has to be sent to the loop body. When starting a computation it corresponds to the input of the for loop block, while a new iteration should get the value of the previous iteration.
- The Counter_Wr block corresponds to the counter that allows association of a number to each data in the loop. This number is then forwarded to the reorder buffer in order to know the writing index of output data.

The block Reorder buffer includes three different types of blocks:

- The Memory blocks store the output data of the loop. Given that the data does not emerge automatically in the same order in which they came, this memory allows a reordering of the data.
- The Read_Result block manages the loop communication with the output signals. It allows reading data in the memory in the correct sequence number. It is also responsible for keeping this data while the next block is not ready to accept a new data.
- The Counter_Rd block provides the index of the next data to be read according to the size of the memory.

Finally the loop body executes the function computation. Depending on the in/out direction of data, different paths are used:

- The first one is used by read-only data. Since these data are not changed by the body of the loop, it is necessary to keep them unchanged before the body. These data should also be delayed with FIFOs to compensate the latency of the body.
- The second one is used by read/write data. Since these data are modified by the loop body, it is necessary to feed them back after the body. Provided that all the outputs of the loop body have the same latency, it is not necessary to insert FIFOs. If their latencies are different they have to be delayed into the body by internal FIFOs.

This for loop implementation, while being quite complex, allows to keep the highest throughput possible, by trying to keep all the basic operators actively computing.

V. Verification

One of the problems a user of a tool such as Math2Mat can face is to trust the generated design. Therefore, one of the aspects of Math2Mat consisted in the generation of automated verification methods allowing to fully verify the generated design.

SystemVerilog has been chosen as the verification language, and a complete testbench is automatically generated for any Math2Mat design. The testbench structure shown in Figure 10 exploits Transaction-level modelling, a transaction corresponding to a simple function computation.

As the input is an Octave code, the verification suite exploits Octave to calculate the expected results. These results are then loaded by the SystemVerilog testbench in order to validate the
observed results. The use of Octave has been chosen in order to guarantee that the results produced by the hardware design will truly correspond to the software computation.

Actually, the testbench offers more than just a black box verification in which the expected result is compared to the result of the Design Under Test (DUT). The user can, through the dynamic view available in the GUI, choose variables (signals) to monitor. In Figure 2, these points correspond to the small squares present on the schematics. By selecting some data to be monitored, the user will force the verification tool to check the value of these data. This is also done by using Octave for generating the expected results of these internal variables. The original Octave code is actually regenerated into a new Octave code that executes the same functionality in terms of calculation, but that also generates the expected internal values. On the testbench side, SystemVerilog allows to access any signal in the hierarchy of the DUT, and some scopes are added in order to get the internal values that are sampled every time they are valid. This white box internal monitoring adds power in terms of verification capabilities, letting the user find the exact point of failure in the design, if any.

The verification capability of Math2Mat is definitely useful for many purposes:

- Validation of new basic blocks
  - Allowing a developer to add new block functions, such as \( \sin() \), \( \exp() \), ...
- Validation of new potential control structures

- Could allow to extend control to \textit{switch} and \textit{while} loops

- Validation of the tool itself
- Validation of the generated structure

A useful action provided to the user consists in the recursive validation of a directory: Math2Mat recursively scans for all Math2Mat projects, generates their corresponding VHDL and testbench, and validates them. This feature allows to quickly and automatically validate any change in the tool or in an operator, by using a set of benchmarking functions.

VI. HARDWARE VALIDATION

The previous section presented the testbench that is exploited in order to verify the design by comparing its output with the output generated by Octave. In order to fully validate Math2Mat, experiments on a real FPGA have also been conducted, using two different frameworks: RECOMS and Ubidules. After the presentation of these two experiments, we show placement and routing results for two functions.

A. RECOMS

The RECOMS framework [11] is composed of a high-end platform coupled with a software environment based on Matlab/Simulink. The hardware platform mainly consists in an ARM9 processor and a Xilinx Virtex5 FPGA. A Simulink wrapper for the Math2Mat components has been designed, allowing a Math2Mat subsystem to be embedded in a RECOMS system.

This framework allowed us to create some sample designs in order to validate the "1 operation per clock cycle" feature of Math2Mat.

B. Ubidule

The aforementioned hardware validation allowed to validate the generated design on a FPGA, but required the intervention of a developer in order to modify the RECOMS design, depending on the number of input/output. The second hardware framework allowed to eliminate the human intervention, letting the software taking care of the entire process.

A Ubidule [12] consists in a board embedding an ARM PXA270 and a SpartanXC3S5000 Xilinx FPGA. The ARM runs embedded Linux, and can directly interact with the FPGA by means of a parallel 32-bits port. An ethernet connection allows to remotely access the processor, a useful feature in the context of a fully automated flow. The Ubidule is a low-end hardware platform compared to the one of RECOMS. It can not really compete with a pure software approach, because of the data latency observed on the communication between the ARM and the FPGA. However, its structure allowed to easily automate the validation process.

Once again the entire flow is automatically launched using a single button. Bitstream generation is performed first by generating a specific wrapper for the Ubidule board, second by generating the TCL scripts for the Xilinx tools, and third by launching these tools (ISE + Impact). This bitstream is then sent to a server running on the PXA270 of the Ubidule.

Fig. 10. SystemVerilog testbench structure
server reconfigures the FPGA, and then waits for the data. The software sends the input data, and waits for the output to be sent back. After receiving the output data it compares it with the Octave results in order to fully validate the design.

C. Synthesis results

Many Matlab code samples where used to validate the tool. We give here the results of a synthesis/placement/routing process for two of these samples. The first one is a for loop embedding a couple of operations, as illustrated in figure 11. The second one is a Lattice-Boltzmann computation, directly taken from [13]. Table II shows the implementation results in terms of resources and clock frequency for a Xilinx Virtex6 xc6vlx550t-2ff1760 device.

![Fig. 11. For loop example](image)

**TABLE II**

<table>
<thead>
<tr>
<th>Feature</th>
<th>For loop</th>
<th>Lattice-Boltzmann</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Frequency</td>
<td>245 MHz</td>
<td>194 MHz</td>
</tr>
<tr>
<td>Slices</td>
<td>6/627 (7%)</td>
<td>16/764 (19%)</td>
</tr>
<tr>
<td>DSP blocks</td>
<td>0 (0%)</td>
<td>50 (5%)</td>
</tr>
<tr>
<td>RAM blocks</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Math2mat blocks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addition</td>
<td>12</td>
<td>15</td>
</tr>
<tr>
<td>Subtraction</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>Multiplication</td>
<td>3</td>
<td>22</td>
</tr>
<tr>
<td>Division</td>
<td>0</td>
<td>13</td>
</tr>
</tbody>
</table>

server reconfigures the FPGA, and then waits for the data. The software sends the input data, and waits for the output to be sent back. After receiving the output data it compares it with the Octave results in order to fully validate the design.

C. Synthesis results

Many Matlab code samples where used to validate the tool. We give here the results of a synthesis/placement/routing process for two of these samples. The first one is a for loop embedding a couple of operations, as illustrated in figure 11. The second one is a Lattice-Boltzmann computation, directly taken from [13]. Table II shows the implementation results in terms of resources and clock frequency for a Xilinx Virtex6 xc6vlx550t-2ff1760 device.

**TABLE II**

<table>
<thead>
<tr>
<th>Feature</th>
<th>For loop</th>
<th>Lattice-Boltzmann</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Frequency</td>
<td>245 MHz</td>
<td>194 MHz</td>
</tr>
<tr>
<td>Slices</td>
<td>6/627 (7%)</td>
<td>16/764 (19%)</td>
</tr>
<tr>
<td>DSP blocks</td>
<td>0 (0%)</td>
<td>50 (5%)</td>
</tr>
<tr>
<td>RAM blocks</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Math2mat blocks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addition</td>
<td>12</td>
<td>15</td>
</tr>
<tr>
<td>Subtraction</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>Multiplication</td>
<td>3</td>
<td>22</td>
</tr>
<tr>
<td>Division</td>
<td>0</td>
<td>13</td>
</tr>
</tbody>
</table>

VI. CONCLUSION AND FUTURE WORK

Math2Mat is currently accessible to anyone through the Math2Mat website. The tool is free to use and allows to rapidly generate a VHDL description of a standard mathematical expression in a fully pipelined way. This first version, while being already usable, could be enhanced in many ways, illustrated by the following topics that will be addressed in the future:

- Math2Mat has been designed to exploit floating point arithmetic. However, the presence of floating point is only visible in the basic operator blocks. The generated structure is independent from the data representation, and so a new set of operators could allow new ways of calculation. In that way Math2Mat might be extended for use with standard integers, or more interestingly with decimal floating point operators.
- The control operations being if/then/else and for might be extended to switch and while loops. switch might be easily implemented using the same structure as the if/then/else. while would require more work, as it is a little more complex than a simple for, because of the condition evaluation.
- Finally, new operators might easily be added to the present library. For instance: exp, log, and trigonometric functions.

The authors warmly encourage interested people to participate to the development of this free tool.

ACKNOWLEDGMENT

The authors would like to thank the HES-SO for the funding support of this project, approved by the ISYS\(^4\) network of competences.

REFERENCES


\(^2\)The Matlab source can be found in the reference.

\(^4\)http://isys.hes-so.ch