VHDL synthesis: a first step toward a comparison between Vivado HLS and MMAlpha

Adrien Gougeon†
Adrien.Gougeon@ens-rennes.fr

Supervisor: Daniel Massicotte‡
Daniel.Massicotte@uqtr.ca

ENS Rennes†
UQTR‡

Internship at the University of Québec at Trois-Rivières
11/05/2018 to 04/08/2018

Abstract. FPGA (Field Programmable Gate Array) are nowadays largely used to realize various specific tasks. They require to use specific languages such as Verilog or VHDL[1] to design the internal circuit. Writing this kind of language can be very time consuming because they are low level languages and because of the intrinsic complexity of designing digital circuits. High Level Synthesis (HLS) tools emerged to reduce design time often at the cost of performance or by using more ressource of the FPGA. This paper compare MMAlpha to an HLS tool, Vivado HLS (VHLS) to generate FPGA-based simulation for power electronics systems. They seem to have roughly the same performance but VHLS seems to use less resources than MMAlpha.

Keywords: FPGA; VHDL; HLS; MMAlpha; Vivado.

1 Introduction

FPGA are integrated circuits built to be configured and reconfigured at will to execute specific tasks. They are commonly used nowadays in various domains such as video and image processing, security systems, power systems and many others. They can run at high frequency and can achieve a high degree of parallelism natively. This allows them to do more operations per clock cycle and achieve a lower time-step than a CPU, making them much more powerful to realize specific tasks and particularly for real-time processing. However, to do those specific tasks they need to be reprogrammed.

The reprogramming of an FPGA to a specific usage is done by modifying its internal logic. It is realized by using an HDL (Hardware Description Language) to implement the new logic of the digital circuit. There exist many languages

[1] VHSIC (Very High Speed Integrated Circuit) Hardware Description Language

to realize this task; Verilog and VHDL are currently the most used but there is many others. Those HDLs where created to raise the level of abstraction and be independent of the integrated circuit used. Digital circuits relies on data-flow and timing principles. Those principles gives another dimension to the HDLs than classical programming language such as C of Java. Data-flow and timing programming can be error-prone and time-consuming when designing digital circuits, especially for non-expert designers.

The complexity of programming the internal logic of FPGA has motivated the emergence of HLS tools. Those tools aim at automatically produce HDL from a high-level description of the desired algorithm to be implemented. This description is often far from the behavior of the FPGA and does not include the principles of data-flow and timing. The languages used in such tools are close to classical languages which greatly reduce design time. Moreover, it is way easier and faster to apply an optimization on an high-level algorithm and synthesize a new design than directly look deep into the HDL.

However, HLS tools have drawbacks. Due to the lack of details in the high-level algorithm they tend to use more resources on the FPGA than custom designs; this problem can be reduced through the use of optimizations. For complex tasks, i.e., system with high degree of dependence or loop bounds not defined at compilation time for example, the generated design will have lower performances than a custom design [4] [6].

The goal of this paper is to compare VHLS and MMAlpha. The first one is a high end HLS tool commonly used in industry while the second one is a research tool based on the polyhedral model. The comparison will be on the performance achieved and the resources usage. This work is in continuation with [4].

VHLS is a commercial HLS tool produced by Xilinx, vastly used nowadays and available on Windows and Linux. This software is designed to produce HDL from C/C++ language. All the process from the writing of the algorithm to the synthesis of the HDL can be done directly in the software, as it include and IDE (Integrated Development Environment). The tool offer a large panel of optimizations to improve the portability of the algorithm on the FPGA and increase its performance and/or resources usage [1]. As Xilinx is also manufacturer of FPGA the synthesis always aim a device and perform specific optimizations depending on it. The optimization methodology applied in VHDL for the synthesis is depicted in Figure1.

On the other hand, MMAlpha is a research prototype (free) software developed at IRISA and available on MacOS and Linux. However being under free license its utilization is done trough a Mathematica interface, a commercial software. MMAlpha is the environment to manipulate, analyze, and derive Alpha programs into VHDL designs. Alpha is a functional language created to provide a support for expressing algorithms in an extended version of the formalism of recurrence equations proposed by Karp, Miller and Winograd [2]. Alpha has been designed to express regular and systolic array algorithms and transform those algorithm from a mathematical specification into a synchronous parallel

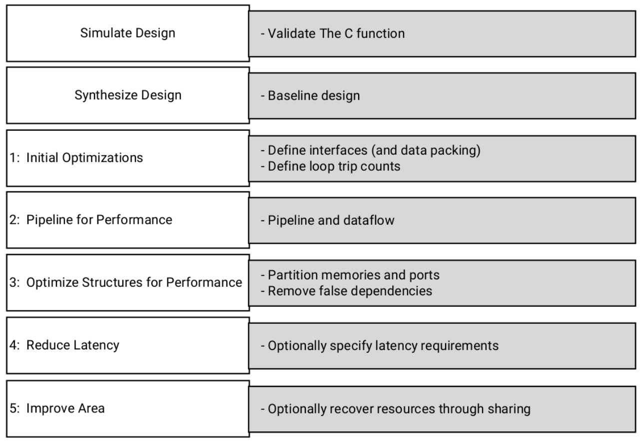

Fig. 1: Optimization methodology applied in Vivado HLS [1].

architecture [3]. Alpha relies on the polyhedral model to analyze and schedule the input program [5].

The comparison in this paper between VHLS and MMAlpha is oriented around power systems in an industrial context. In this direction our approach is centered on matrix vector multiplication. In previous research [4] VHLS was defined as producing efficient HDL in terms of time-step, area efficiency and accuracy, but only for small circuits. For larger circuits it seems that VHLS is not cost-effective anymore. This work is only a first step to define if MMAlpha is capable to outperform VHLS. The research was centered about a better understanding on how VHLS work, the MMAlpha version compared is still basic and require optimizations.

The rest of this paper is organized as follow: Section 2 reviews the related works in this domain. Section 3 presents the comparisons done between MMAlpha and VHLS. Section 4 details the results from the comparisons. Finally, Section 5 concludes.

2 Related work

This work is in the continuation of [4]. The main idea in this work was to compare a HLS tool, Vivado HLS, to custom designs in the domain of hardware-in-the-loop simulation on FPGA. The authors compared them to determine if Vivado HLS could be a viable solution in terms of performance and area used on the FPGA. Their conclusion was that Vivado HLS can be really efficient, close to customs designs, on small designs; but it is inefficient in terms of area used when the design becomes larger. However, due to the practicality of using a higher level tool the authors have no doubt that HLS tools will be more and more used in the future.

On the other hand, [6] compares VHDL synthesized by Vivado HLS to customs designs for different matrix multiplication algorithms. The authors compared a classic algorithm, the Strassen algorithm and the sparse algorithm. Their conclusion is that HLS tools perform good when the algorithm is simple, like the classic algorithm. For complex algorithm, like the sparse algorithm with undetermined loop bounds, HLS tools have a hard time to optimize the synthesis compared to what an experienced VHDL programmer can do. However, as in the other paper, they conclude that considering the time gain from using HLS tools still make them a viable solution.

In this continuation the work here aims to compare the VHDL produced by Vivado HLS to the one produced by MMAlpha to determine if MMAlpha can have the same, or better, performance than Vivado HLS while being more cost-efficient in terms of area used.

3 Matrix vector multiplication

This Section explains and details what are the comparisons done between MMAlpha and Vivado HLS.

The comparison is done on a simple matrix vector multiplication. It was carried out to present the optimizations directives used in Vivado HLS and to look at the performances of Vivado HLS and MMAlpha for a simple algorithm.

We first present how the experiment was done in Vivado HLS considering a multiplication by row of the matrix, then in Vivado HLS considering a multiplication by column of the matrix and finally with MMAlpha considering a multiplication by column.

3.1 Vivado HLS by row

The C code used in this version is presented in Figure 2. This algorithm multiplies a matrix a by a vector b and put the result in a vector c. The matrix is represented as a one dimension array. As we consider here a multiplication by row, the inner loop (loop0) multiplies a row of the matrix by the vector, the outer loop (loop1) process successively the rows of the matrix.

In its first approach Vivado HLS simply compute each step of the loops after another. This is extremely inefficient and take a lot of cycles.

To optimize a design Vivado HLS uses a system of directives to apply optimizations on the C code for the synthesis. Those directives can be either written in a separated directive file or written directly in the C code with *#pragma*, as in our example.

The first idea is to unroll the inner loop to make it parallel. This will require more computing blocks of the FPGA but will greatly increase performance and reduce the number of cycles; this optimization correspond to the directive line 12 of Figure 2: HLS_UNROLL. However, as we can see in line 13 of Figure 2, the inner loop makes access to the arrays containing the matrix and the vector. Thus, the second idea is to split the arrays a and b in order to allow parallel

```
1void prodMatVect(int a[SIZE*SIZE], int b[SIZE], int c[SIZE])
2{
3#pragma HLS ARRAY_PARTITION variable=b complete dim=1
4#pragma HLS ARRAY_PARTITION variable=a cyclic factor=16 dim=1
5        int sum;
6        loop1:for (int i = 0; i < MATSIZE; i++)
7        {
8#pragma HLS PIPELINE
9                sum = 0;
10               loop0:for (int j = 0; j < MATSIZE; j++)
11               {
12#pragma HLS UNROLL
13                       sum += a[i * MATSIZE + j] * b[j];
14               }
15               c[i] = sum;
16       }
17}
```

Fig. 2: Matrix vector multiplication in C. Multiply the matrix a by the vector b and place the result in the vector c. The multiplication is done by row of the matrix. The matrix is represented as a one dimension array. The pragmas are used to optimize the synthesis.

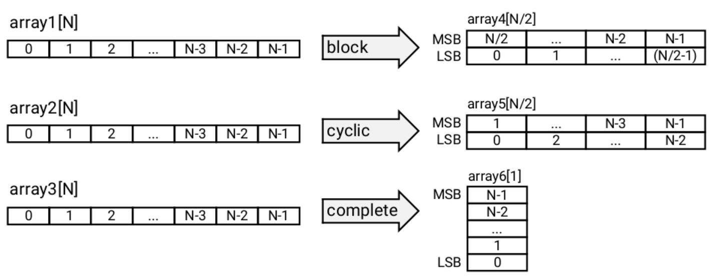

Fig. 3: Partition types available with the optimization directive $HLS\_PARTITION$ [1]. In this example the partition factor is set to 2. The partition splits an array into smaller arrays in order to allow parallel accesses.

accesses of the arrays; this correspond to the directives lines 3 and 4 on the Figure 2: $HLSARRAY\_PARTITION$. The partition requires to chose a type of partition. In one step of the outer loop, and because of the unrolling, we need to access all the values of the b vector, so the partition of it has to be complete. For the matrix, we need to access all values of a row, the solution is to split it in a cyclic way, putting successive values in different arrays; the number of arrays depends on the factor, here it must be the number of rows. The Figure 3 shows the different partition type.

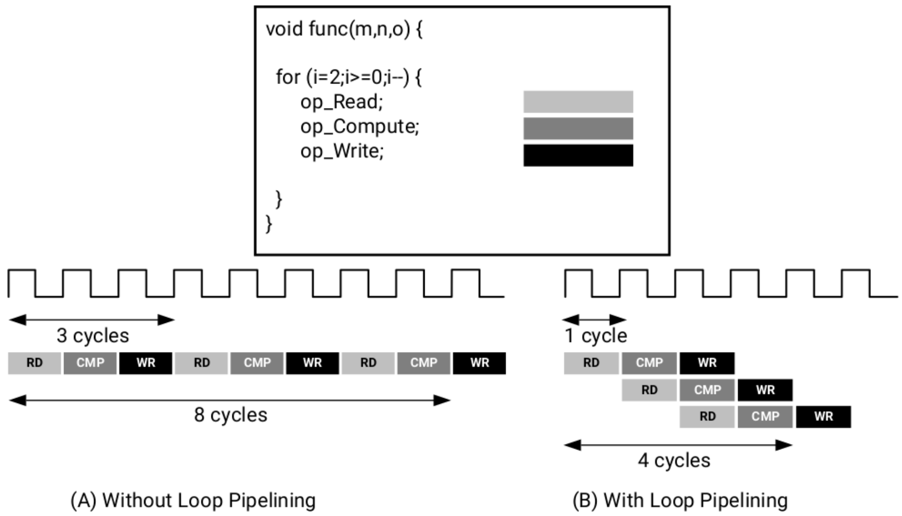

Fig. 4: Presentation of the directive $HLS\_PIPELINE$ [1]. Instead of doing all operations completely one after the other, the pipelining consist in using every stage of a computing unit with different instructions. In this example at the third cycle there is three different instructions using different stages of the computing unit.

Finally, another idea is to pipeline the operations. This in done by applying the pipeline directive to the outer loop[2]; this correspond to the line 8 in the Figure 2: $HLS\_PIPELINE$. Pipelining, as shown in Figure 4, allow to make a continuous use of all the part of the design, reducing again the number of cycles required for the computation

3.2 Vivado HLS by column

The C code used in this version is presented in Figure 5. This algorithm multiplies a matrix a by a vector b and put the result in a vector c. The matrix is represented as a one dimension array. As we consider here a multiplication by column, the inner loop (loop0) multiplies a column of the matrix by a value of the vector. The outer loop (loop1) process successively the columns of the matrix and the values of the vector. The first loop (init_loop) set all values of c at 0. This loop is necessary because otherwise as VHLS does not know the initial values of c it consider it as both an input and an output vector. With this loop is consider it only as an output.

The first partition pragma split the row of the matrix in different arrays, in order to access all values of a column in one cycle. The second partition pragma

[2] This directive placed in a loop automatically unroll the inner loops, making HLS_UNROLL useless, we left it in the example only to show how to use it without the $HLS\_PIPELINE$ directive.

```
void prodMatVect(short a[MATSIZE*MATSIZE], short b[MATSIZE],
    short c[MATSIZE])
{
        #pragma HLS ARRAY_PARTITION variable=a block
            factor=16 dim=1
        #pragma HLS ARRAY_PARTITION variable=c complete dim=1
        init_loop:for (short i = 0; i < MATSIZE; i++)
        {
                #pragma HLS UNROLL
                c[i] = 0;
        }
        loop1:for (short j = 0; j < MATSIZE; j++)
        {
                #pragma HLS PIPELINE
                loop0:for (short i = 0; i < MATSIZE; i++)
                {
                        #pragma HLS UNROLL
                        c[i] += a[i * MATSIZE + j] * b[j];
                }
        }
}
```

Fig. 5: Matrix vector multiplication in C. Multiply the matrix a by the vector b and place the result in the vector c. The multiplication is done by column of the matrix. The matrix is represented as a one dimension array. The pragmas are used for the synthesis optimizations.

split completely the c array, this is necessary because the unrolling of the inner loop require a parallel access to all its values. The first unroll set all values of c to 0 in one cycle. The second unroll pragma makes the multiplication parallel. Finally, the pipeline pragma pipeline the multiplications.

3.3 MMAlpha

The Alpha code used in MMAlpha is presented in Figure 6. Parameter and variables are represented by dimension assigned with a size:

- The first parameter, N, represent the size of the data flow given to the algorithm;
- a is the input matrix. i and j represent the two dimension of the matrix, of size N;
- b is the input vector, i represent its length, of size N;
- c is the output vector, i represent its length, of size N;
- d is an internal variable. Its i dimension is of size N to store the size of the output vector. Its j dimension, of size $N + 1$, is to apply the recurrence principle, explained below.

```
1system prodMatVect: {N | N>1}
2         (a : {i,j|1 <= i,j <= N} of integer;
3         b : {i|1 <= i <= N} of integer)
4returns (c : {i|1 <= i <= N} of integer);
5var
6         d : {i,j|1 <= i <= N; 0<= j <=N} of integer;
7let
8         d[i,j] =
9                 case
10                        {|j=0} : 0[];
11                        {|j>=1} : d[i,j-1] + a[i,j] * b[j];
12                esac;
13        c[i]=d[i,N];
14tel;
```

Fig. 6: Simple matrix vector multiplication in Alpha. Multiply the matrix a by the vector b and place the result in the vector c.

As MMAlpha is based of affine regular expression its syntax rely on recurrence principle. The *let* is where the recurrence takes place. In this part we define how will be calculated d depending on i and j. In the first case, when $j = 0$, we set its initial value to 0. In the second case, when $j >= 1$, we apply the multiplication of a column of the matrix by a value of the vector. This values is stored and accumulated in d. Finally, on line 13, we assign the output vector to the last accumulated values of d.

MMAlpha does partitioning and unrolling by default.

4 Results

This section explains what are the resources surveyed in the comparison and what was the result found.

All experiments were done with Vivado 2018.1 on a Kintex UltraScale+ FPGA, model xcku5p-ffva676-1-i. We used data type of 16 bits integer.

We first present the internal behavior of the different synthesis in the Schematic analysis. After that we discuss the number of cycle needed to realize the task. FPGA designs are synchronous and driven by a clock. The number of cycle represent how many clock cycles are needed from the moment we set the first input data to the moment the last result is available.

Finally, to discuss the main material resources used, we have to introduce what are those resources that we survey on the FPGA:

- Digital Signal Processor (DSP): those units are used to multiply or add two data buses. In our case the model was DSP48E, multiplying or adding 16 bits data buses.

– Flip Flip (FF): those units are registers, used for data storage.
– Look Up Table (LUT): those are basically truth tables.

To give an idea of the utilization of the resources of the FPGA, the model used has:

– 216960 LUT;
– 433920 FF;
– 1824 DSP.

4.1 Schematic analysis:

This part explains the differences between the schematics extracted from the designs, and thus, how they behave.

VHLS by row multiplies at each cycle the data of one row of the matrix with the full vector. As shown in Figures 11 to 16 in Appendix, this design relies on $log(N)$ sum to complete the computation in one cycle. This chain of combinatory block may lead to an excessive WNS. However, VHLS adapt the design to avoid a too long chain by adding a register. This register reduce the WNS but add 1 more cycle to finish the global computation.

VHLS by column and MMAlpha have pretty much the same behaviorr. As shown in Figures 17 and 19 in Appendix, they relies on a multiplier-adder principle. At each cycle, each value of a column of the matrix is multiplied by a single value of the vector. This value is then summed with the last result stored and the new result is stored for the next cycle. This continue until each column has been processed. The length of the data paths does not scale with the size and is fixed.

The difference between VHLS by column and MMAlpha is that MMAlpha has as much input for the vector as the size of the multiplication. In fact, the user of the design has to duplicate each value of the vector depending on the size. On the other hand, VHLS by column has only 1 input for the value, which is pinned to every multiplier-adder.

4.2 Cycles analysis

In the Table 1 are shown the number of cycles needed to do the matrix vector multiplication depending on the size and the synthesis tool used[3].

We observe that VHLS by column and MMAlpha perfectly scale with the size. At each cycle a new column is processed. VHLS by row scale too but with one more cycle starting from size 16. This is explained by the register added to reduce the WNS, as shown in the schematic analysis in Section 4.1.

[3] Note on the results: the design in Vivado HLS by column of size 256 do not work as intended. The C code used to for the VHDL synthesis is the same than for the other size but the VHDL synthesis consider some parts of the output vector as both input and output. We have not discovered why.

Size	VHLS by row	VHDL by column	MMAlpha
8	8	8	8
16	17	16	16
32	33	32	32
64	65	64	64
128	129	128	128
256	257	256	256

Table 1: Number of cycles required to compute the multiplication, depending on the size and the synthesis tool.

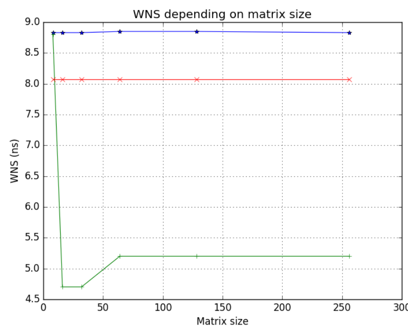

Fig. 7: Worst Negative Slack for the different designs, depending on the size and on the synthesis tool. MMAlpha is shown in blue, VHLS by column in red and VHLS by row in green.

4.3 Slack analysis

In the Figure 7 are shown the WNS found for the different designs, depending on the size and the synthesis tool used.

We observe that the multiplication by column, as used by VHLS by column and MMAlpha, is far more efficient and can run at significantly higher frequency than VHLS by row. As explained in Section 4.1, VHLS by row have quickly a long chain of combinatory block which reduce the slack. All different critical paths are presented in Appendix.

It is important to notice that the bottleneck for VHLS by column is the data path, but it is the controller path for MMAlpha.

It is noticeable that MMAlpha should be as stable as VHLS by column is, but it is slightly higher for the size 64 and 128. The reason is that the design synthesized for those size use LUT4 instead of LUT6, which have a lower processing time, as shown in the critical paths in Appendix.

4.4 Material resources

Resource usage for the different designs are presented in Figures 8, 9 and 10.

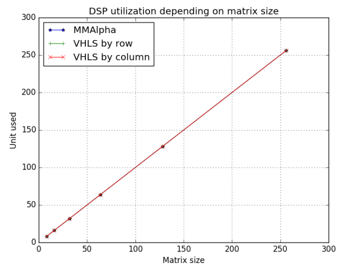

Fig. 8: Number of DSP used for the design, depending on the matrix size and the synthesis tool.

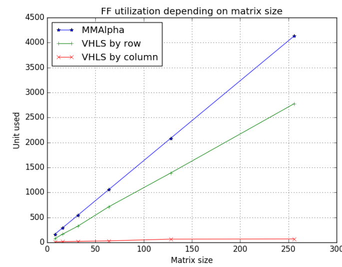

Fig. 9: Number of FF used for the design, depending on the matrix size and the synthesis tool.

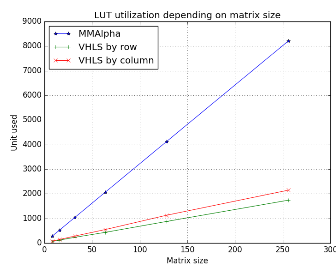

Fig. 10: Number of LUT used for the design, depending on the matrix size and the synthesis tool.

We observe that all versions use the same amount of DSP. MMAlpha use more FF and LUT than both VHLS versions. Table 2 show that the MMAlpha design use more than 3.5 times more LUT than the VHLS by column version, while the VHLS by row version use around 25% less. Table 3 show that the FF utilization is far worse: MMAlpha use 16 to 59 times more FF than the VHLS by column version.

5 Conclusion

This paper aimed toward a comparison of Vivado HLS and MMAlpha. HLS tools are more and more used nowadays due notably to the practicality of an higher level environment. As seen in previous works [4,6] they provide an overall good performance but tend to use more area than customs designs. This comparison aimed to determine if MMAlpha makes a better use of the material available on the FPGA while keeping the same performances as Vivado HLS. The comparison was on a simple algorithm, the matrix vector multiplication for various size: 8, 16, 32, 64, 128 and 256.

Size	VHDL by column	VHLS by row	MMAlpha
8	1x	0.79x	3.58x
16	1x	0.82x	3.66x
32	1x	0.82x	3.66x
64	1x	0.79x	3.77x
128	1x	0.78x	3.65x
256	1x	0.81x	3.82x

Table 2: Comparison of the LUT usage. VHLS by column is considered as the reference.

Size	VHDL by column	VHLS by row	MMAlpha
8	1x	7.7x	16.4x
16	1x	10.31x	18.25x
32	1x	14.91x	24.91x
64	1x	23.06x	34.19x
128	1x	21.38x	32.06x
256	1x	40.25x	59.88x

Table 3: Comparison of the FF usage. VHLS by column is considered as the reference.

The cycle analysis has shown MMAlpha and VHLS by column to have the same performance: they both perfectly scale with the size and compute a column of the matrix at each cycle. VHLS by row, due to its increasing chain of combinatory block has to add a register to reduce its WNS. This has for consequence an additional cycle to perform the calculation starting from size 16.

The WNS analysis has shown MMAlpha to have a slightly better performance than VHLS by column. VHLS by row is completely out of the comparison due to its chain of combinatory blocks reducing greatly its WNS.

The material analysis has shown MMAlpha and VHLS to use the same amount of DSP. However, MMAlpha use far more LUT than both version of VHLS, and incredibly more FF the VHLS by column version(up to 59 times more).

In this particular example Vivado HLS is simple to use in the sense that the matrix vector multiplication algorithm in C in both version, by row or by column, is straightforward. The optimizations are easy because the reflexion is on the split needed on the data arrays, which is simple in this case. On the other hand, the algorithm is noticeably more difficult to write in the Alpha language. However, Alpha automatically apply unrolling and partitioning.

This work is an introduction to the problem. We treated here only a simple algorithm and the basic version of MMAlpha, without optimizations. The result shows that MMAlpha, even without optimizations, to have slightly better performances considering the WNS. However, its resources utilization is catastrophic in this case.

Future work should be around a better understanding on how Vivado HLS and MMAlpha differ in their dependence analysis for more complex design. MMAlpha is supposed to perform a better analysis and scheduling due to the polyhedral libraries it relies on. To verify this hypothesis research should be on how to optimize it properly. Considering its resources consumption, it is necessary to optimize it too. This work is expected to be pursued by using MMAlpha to reproduce the experiment done in [4].

References

1. Vivado hls optimization methodology guide. UG1270 (v2017.4), December 2017.
2. Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, July 1967.
3. Hervé Le Verge, Christophe Mauras, and Patrice Quinton. The alpha language and its use for the design of systolic arrays. *Journal of VLSI signal processing systems for signal, image and video technology*, 3(3):173–182, Sep 1991.
4. Federico Montano, Tarek Ould Bachir, and Jean-Pierre David. An evaluation of a high-level synthesis approach to the fpga-based submicrosecond real-time simulation of power converters. PP:1–1, 06 2017.
5. P. Quinton, A. M. Chana, and S. Derrien. Efficient hardware implementation of data-flow parallel embedded systems. In *2012 International Conference on Embedded Computer Systems (SAMOS)*, pages 364–371, July 2012.
6. S. Skalicky, C. Wood, M. Łukowiak, and M. Ryan. High level synthesis: Where are we? a case study on matrix multiplication. In *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 1–7, Dec 2013.

A Data Paths

A.1 VHLS by row data paths

A.2 VHLS by column data path

A.3 MMAlpha data path

B Critical paths

B.1 VHLS by row critical paths

B.2 VHLS by column critical paths

B.3 MMAlpha critical paths

C VHDL

C.1 Vivado HLS VHDL

C.2 MMAlpha VHDL

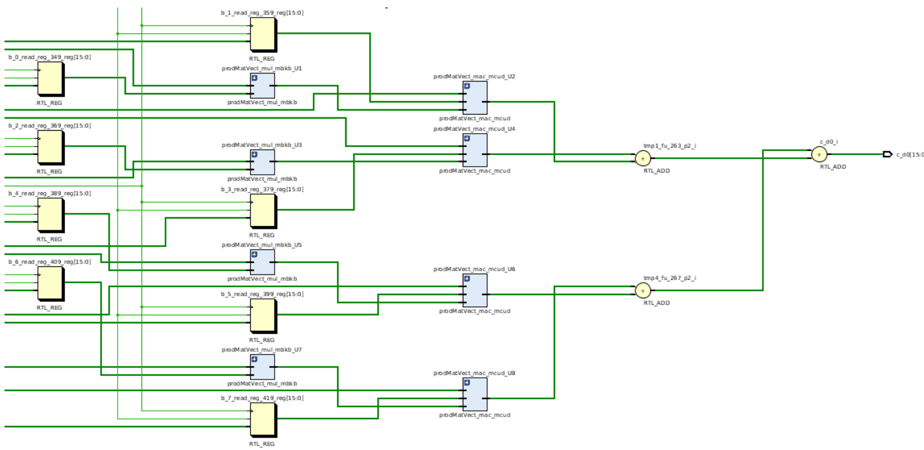

Fig. 11: Data paths in the design from Vivado HLS for a matrix vector multiplication by row of size 8.

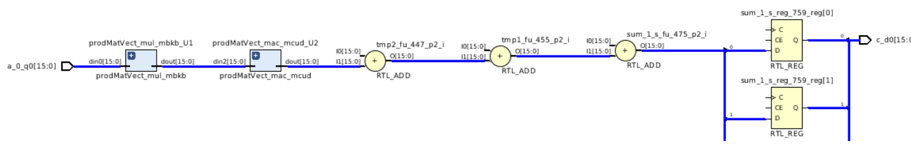

Fig. 12: Data path in the design from Vivado HLS for a matrix vector multiplication by row of size 16. Zoomed on 1 path.

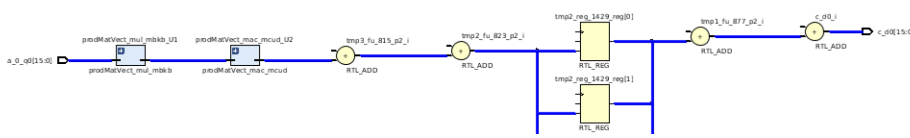

Fig. 13: Data path in the design from Vivado HLS for a matrix vector multiplication by row of size 32. Zoomed on 1 path.

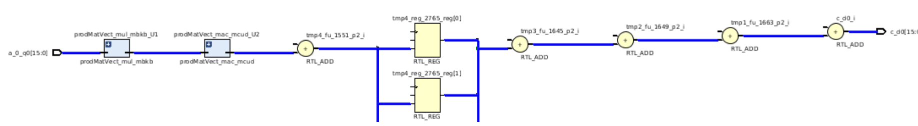

Fig. 14: Data path in the design from Vivado HLS for a matrix vector multiplication by row of size 64. Zoomed on 1 path.

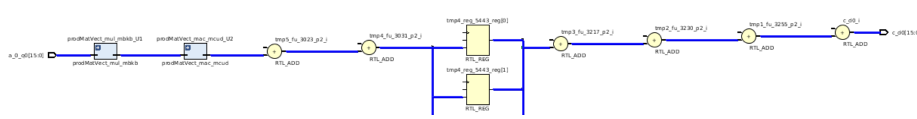

Fig. 15: Data path in the design from Vivado HLS for a matrix vector multiplication by row of size 128. Zoomed on 1 path.

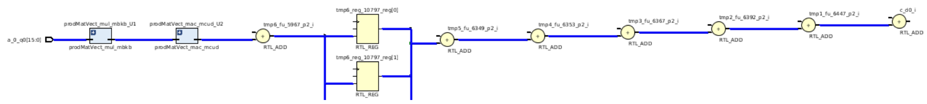

Fig. 16: Data path in the design from Vivado HLS for a matrix vector multiplication by row of size 256. Zoomed on 1 path.

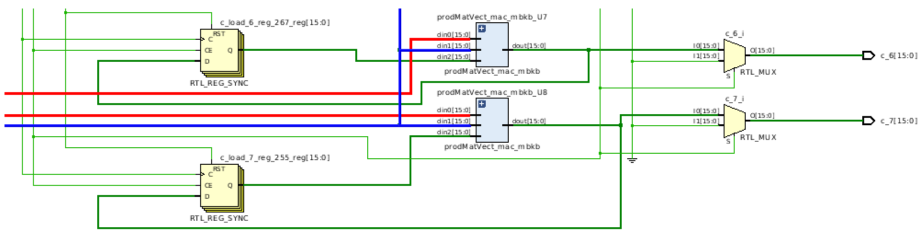

Fig. 17: Data paths in the design from Vivado HLS for a matrix vector multiplication by column of size 8. Zoomed on 2 paths. Data from the matrix are shown in red and data from the vector are shown in blue. The block following (blue square) is detailed in the next figure.

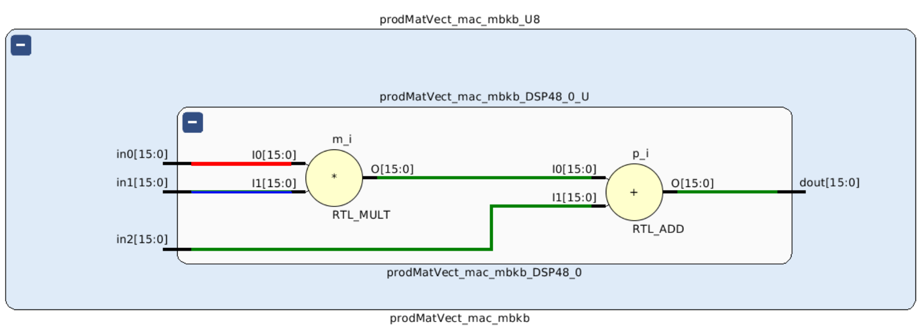

Fig. 18: Internal view of a multiplier-adder from the design by Vivado HLS for a matrix vector multiplication by column of size 8.

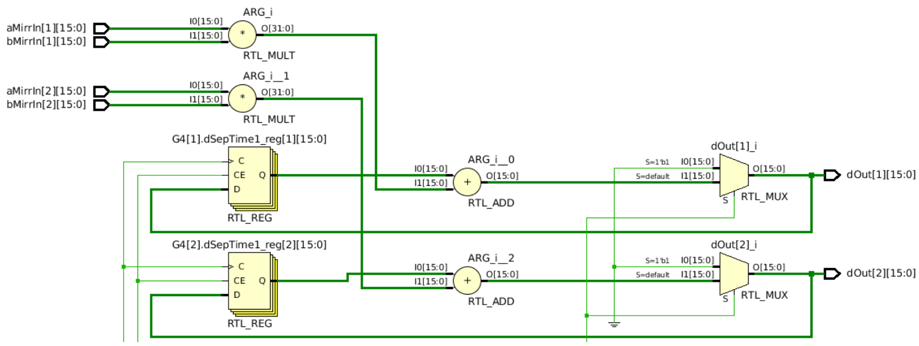

Fig. 19: Data paths in the design from MMAlpha for the size 8. Zoomed on 2 paths.

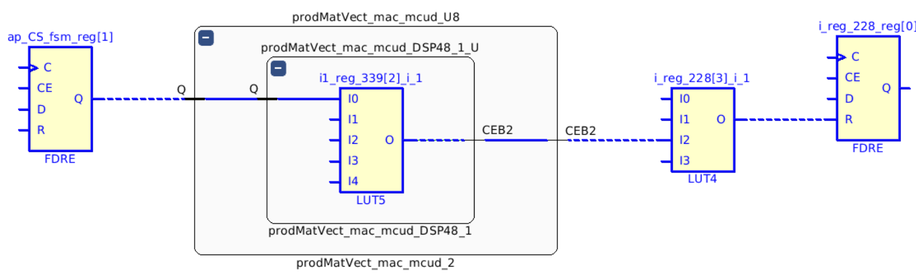

Fig. 20: Critical path in the design from Vivado HLS for a matrix vector multiplication by row of size 8.

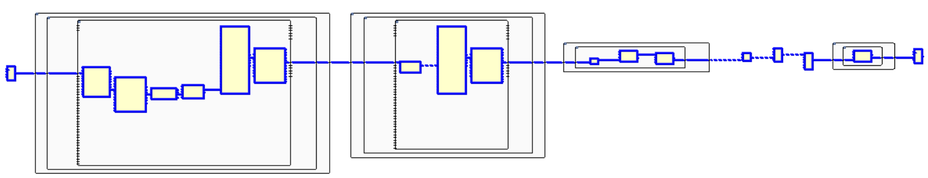

Fig. 21: Critical path in the design from Vivado HLS for a matrix vector multiplication by row of size 16.

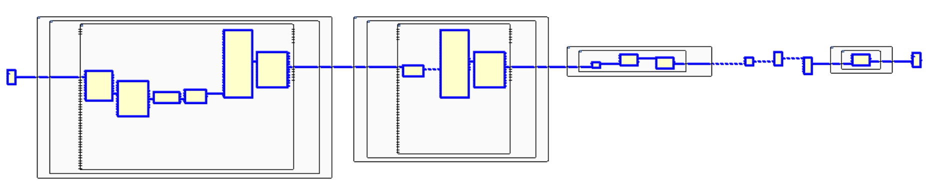

Fig. 22: Critical path in the design from Vivado HLS for a matrix vector multiplication by row of size 32.

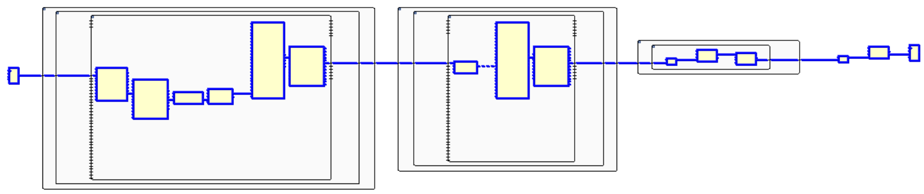

Fig. 23: Critical path in the design from Vivado HLS for a matrix vector multiplication by row of size 64.

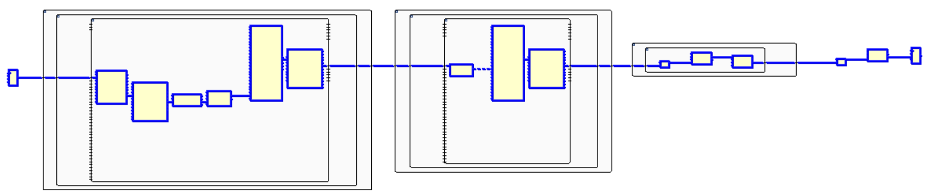

Fig. 24: Critical path in the design from Vivado HLS for a matrix vector multiplication by row of size 128.

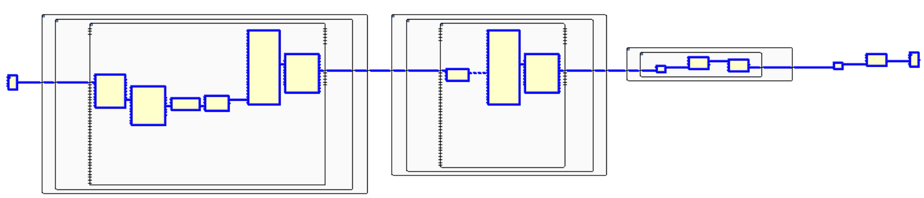

Fig. 25: Critical path in the design from Vivado HLS for a matrix vector multiplication by row of size 256.

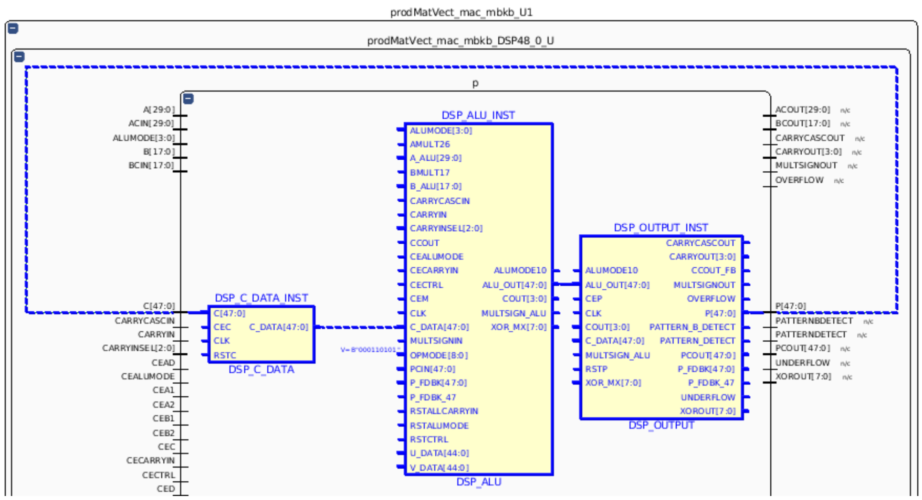

Fig. 26: Critical path in the design from Vivado HLS for a matrix vector multiplication by column of size 8.

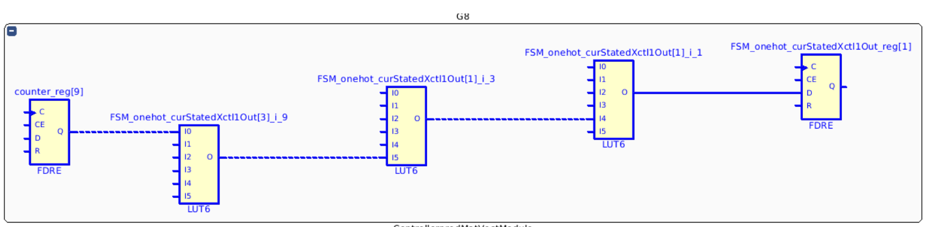

Fig. 27: Critical path in the design from MMAlpha for a matrix vector multiplication of size 8.

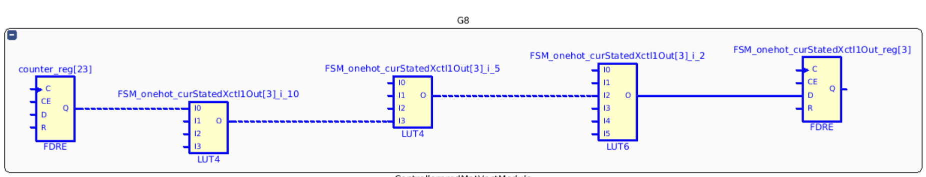

Fig. 28: Critical path in the design from MMAlpha for a matrix vector multiplication of size 64.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity prodMatVect_mul_mbkb_DSP48_0 is
port (
a: in std_logic_vector(16 - 1 downto 0);
b: in std_logic_vector(16 - 1 downto 0);
p: out std_logic_vector(16 - 1 downto 0));

end entity;

architecture behav of prodMatVect_mul_mbkb_DSP48_0 is
signal a_cvt: signed(16 - 1 downto 0);
signal b_cvt: signed(16 - 1 downto 0);
signal p_cvt: signed(16 - 1 downto 0);

attribute keep : string;
attribute keep of a_cvt : signal is "true";
attribute keep of b_cvt : signal is "true";
attribute keep of p_cvt : signal is "true";

begin

a_cvt <= signed(a);
b_cvt <= signed(b);
p_cvt <= signed (resize(unsigned (signed (a_cvt) *
    signed (b_cvt)), 16));
p <= std_logic_vector(p_cvt);

end architecture;

Library IEEE;
use IEEE.std_logic_1164.all;

entity prodMatVect_mul_mbkb is
generic (
ID : INTEGER;
NUM_STAGE : INTEGER;
din0_WIDTH : INTEGER;
din1_WIDTH : INTEGER;
dout_WIDTH : INTEGER);
port (
din0 : IN STD_LOGIC_VECTOR(din0_WIDTH - 1 DOWNTO 0);
din1 : IN STD_LOGIC_VECTOR(din1_WIDTH - 1 DOWNTO 0);
dout : OUT STD_LOGIC_VECTOR(dout_WIDTH - 1 DOWNTO
    0));
end entity;
```

Fig. 29: VHDL code from prodMatVect_mul_mbkb.vhd, generated by Vivado HLS, part one.

```
architecture arch of prodMatVect_mul_mbkb is
component prodMatVect_mul_mbkb_DSP48_0 is
port (
a : IN STD_LOGIC_VECTOR;
b : IN STD_LOGIC_VECTOR;
p : OUT STD_LOGIC_VECTOR);
end component;



begin
prodMatVect_mul_mbkb_DSP48_0_U :  component
    prodMatVect_mul_mbkb_DSP48_0
port map (
a => din0,
b => din1,
p => dout);

end architecture;
```

Fig. 30: VHDL code from prodMatVect_mul_mbkb.vhd, generated by Vivado HLS, part two.

```
-- ==============================================================
-- File generated by Vivado(TM) HLS - High-Level
    Synthesis from C, C++ and SystemC
-- Version: 2018.1
-- Copyright (C) 1986-2018 Xilinx, Inc. All Rights
    Reserved.
--
-- ==============================================================

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity prodMatVect_mac_mcud_DSP48_1 is
port (
in0:  in  std_logic_vector(16 - 1 downto 0);
in1:  in  std_logic_vector(16 - 1 downto 0);
in2:  in  std_logic_vector(16 - 1 downto 0);
dout: out std_logic_vector(16 - 1 downto 0));

end entity;

architecture behav of prodMatVect_mac_mcud_DSP48_1 is
signal a       : signed(27-1 downto 0);
signal b       : signed(18-1 downto 0);
signal c       : signed(48-1 downto 0);
signal m       : signed(45-1 downto 0);
signal p       : signed(48-1 downto 0);
begin
a  <= signed(resize(signed(in0), 27));
b  <= signed(resize(signed(in1), 18));
c  <= signed(resize(signed(in2), 48));

m  <= a * b;
p  <= m + c;

dout <= std_logic_vector(resize(unsigned(p), 16));

end architecture;

Library IEEE;
use IEEE.std_logic_1164.all;
```

Fig. 31: VHDL code from prodMatVect_mac_mcud.vhd, generated by Vivado HLS, part one.

```
entity prodMatVect_mac_mcud is
generic (
ID : INTEGER;
NUM_STAGE : INTEGER;
din0_WIDTH : INTEGER;
din1_WIDTH : INTEGER;
din2_WIDTH : INTEGER;
dout_WIDTH : INTEGER);
port (
din0 : IN STD_LOGIC_VECTOR(din0_WIDTH - 1 DOWNTO 0);
din1 : IN STD_LOGIC_VECTOR(din1_WIDTH - 1 DOWNTO 0);
din2 : IN STD_LOGIC_VECTOR(din2_WIDTH - 1 DOWNTO 0);
dout : OUT STD_LOGIC_VECTOR(dout_WIDTH - 1 DOWNTO
    0));
end entity;

architecture arch of prodMatVect_mac_mcud is
component prodMatVect_mac_mcud_DSP48_1 is
port (
in0 : IN STD_LOGIC_VECTOR;
in1 : IN STD_LOGIC_VECTOR;
in2 : IN STD_LOGIC_VECTOR;
dout : OUT STD_LOGIC_VECTOR);
end component;



begin
prodMatVect_mac_mcud_DSP48_1_U :  component
    prodMatVect_mac_mcud_DSP48_1
port map (
in0 => din0,
in1 => din1,
in2 => din2,
dout => dout);

end architecture;
```

Fig. 32: VHDL code from prodMatVect_mac_mcud.vhd, generated by Vivado HLS, part two.

```
-- ==============================================
-- RTL generated by Vivado(TM) HLS - High-Level
    Synthesis from C, C++ and SystemC
-- Version: 2018.1
-- Copyright (C) 1986-2018 Xilinx, Inc. All Rights
    Reserved.
--
-- ==============================================

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity prodMatVect is
port (
ap_clk : IN STD_LOGIC;
ap_rst : IN STD_LOGIC;
ap_start : IN STD_LOGIC;
ap_done : OUT STD_LOGIC;
ap_idle : OUT STD_LOGIC;
ap_ready : OUT STD_LOGIC;
a_0_address0 : OUT STD_LOGIC_VECTOR (1 downto 0);
a_0_ce0 : OUT STD_LOGIC;
a_0_q0 : IN STD_LOGIC_VECTOR (15 downto 0);
a_1_address0 : OUT STD_LOGIC_VECTOR (1 downto 0);
a_1_ce0 : OUT STD_LOGIC;
a_1_q0 : IN STD_LOGIC_VECTOR (15 downto 0);
a_2_address0 : OUT STD_LOGIC_VECTOR (1 downto 0);
a_2_ce0 : OUT STD_LOGIC;
a_2_q0 : IN STD_LOGIC_VECTOR (15 downto 0);
a_3_address0 : OUT STD_LOGIC_VECTOR (1 downto 0);
a_3_ce0 : OUT STD_LOGIC;
a_3_q0 : IN STD_LOGIC_VECTOR (15 downto 0);
b_0 : IN STD_LOGIC_VECTOR (15 downto 0);
b_1 : IN STD_LOGIC_VECTOR (15 downto 0);
b_2 : IN STD_LOGIC_VECTOR (15 downto 0);
b_3 : IN STD_LOGIC_VECTOR (15 downto 0);
c_address0 : OUT STD_LOGIC_VECTOR (1 downto 0);
c_ce0 : OUT STD_LOGIC;
c_we0 : OUT STD_LOGIC;
c_d0 : OUT STD_LOGIC_VECTOR (15 downto 0) );
end;
```

Fig. 33: VHDL code from prodMatVect.vhd, generated by Vivado HLS, part one.

```
architecture behav of prodMatVect is
attribute CORE_GENERATION_INFO : STRING;
attribute CORE_GENERATION_INFO of behav :
    architecture is
"prodMatVect,hls_ip_2018_1,{HLS_INPUT_TYPE=cxx,␣
    HLS_INPUT_FLOAT=0,HLS_INPUT_FIXED=0,␣
    HLS_INPUT_PART=xcku5p-ffva676-1-i,␣
    HLS_INPUT_CLOCK=10.000000,␣
    HLS_INPUT_ARCH=others,HLS_SYN_CLOCK=7.650000,␣
    HLS_SYN_LAT=6,HLS_SYN_TPT=none,HLS_SYN_MEM=0,␣
    HLS_SYN_DSP=4,HLS_SYN_FF=76,HLS_SYN_LUT=92}";
constant ap_const_logic_1 : STD_LOGIC := '1';
constant ap_const_logic_0 : STD_LOGIC := '0';
constant ap_ST_fsm_state1 : STD_LOGIC_VECTOR (2
    downto 0) := "001";
constant ap_ST_fsm_pp0_stage0 : STD_LOGIC_VECTOR (2
    downto 0) := "010";
constant ap_ST_fsm_state4 : STD_LOGIC_VECTOR (2
    downto 0) := "100";
constant ap_const_lv32_0 : STD_LOGIC_VECTOR (31
    downto 0) := "00000000000000000000000000000000";
constant ap_const_boolean_1 : BOOLEAN := true;
constant ap_const_lv32_1 : STD_LOGIC_VECTOR (31
    downto 0) := "00000000000000000000000000000001";
constant ap_const_boolean_0 : BOOLEAN := false;
constant ap_const_lv1_0 : STD_LOGIC_VECTOR (0 downto
    0) := "0";
constant ap_const_lv1_1 : STD_LOGIC_VECTOR (0 downto
    0) := "1";
constant ap_const_lv3_0 : STD_LOGIC_VECTOR (2 downto
    0) := "000";
constant ap_const_lv3_4 : STD_LOGIC_VECTOR (2 downto
    0) := "100";
constant ap_const_lv3_1 : STD_LOGIC_VECTOR (2 downto
    0) := "001";
constant ap_const_lv32_2 : STD_LOGIC_VECTOR (31
    downto 0) := "00000000000000000000000000000010";

signal ap_CS_fsm : STD_LOGIC_VECTOR (2 downto 0) :=
    "001";
attribute fsm_encoding : string;
attribute fsm_encoding of ap_CS_fsm : signal is
    "none";
signal ap_CS_fsm_state1 : STD_LOGIC;
attribute fsm_encoding of ap_CS_fsm_state1 : signal
    is "none";
signal i_reg_140 : STD_LOGIC_VECTOR (2 downto 0);
signal exitcond1_fu_151_p2 : STD_LOGIC_VECTOR (0
    downto 0);
signal exitcond1_reg_202 : STD_LOGIC_VECTOR (0
    downto 0);
signal ap_CS_fsm_pp0_stage0 : STD_LOGIC;
attribute fsm_encoding of ap_CS_fsm_pp0_stage0 :
    signal is "none";
```

Fig. 34: VHDL code from prodMatVect.vhd, generated by Vivado HLS, part two.

```
    signal ap_block_state2_pp0_stage0_iter0 : BOOLEAN;
    signal ap_block_state3_pp0_stage0_iter1 : BOOLEAN;
    signal ap_block_pp0_stage0_11001 : BOOLEAN;
    signal i_1_fu_157_p2 : STD_LOGIC_VECTOR (2 downto 0);
    signal ap_enable_reg_pp0_iter0 : STD_LOGIC := '0';
    signal i1_fu_163_p1 : STD_LOGIC_VECTOR (63 downto 0);
    signal i1_reg_211 : STD_LOGIC_VECTOR (63 downto 0);
    signal b_0_read_reg_221 : STD_LOGIC_VECTOR (15 downto 0);
    signal b_1_read_reg_231 : STD_LOGIC_VECTOR (15 downto 0);
    signal b_2_read_reg_241 : STD_LOGIC_VECTOR (15 downto 0);
    signal b_3_read_reg_251 : STD_LOGIC_VECTOR (15 downto 0);
    signal ap_block_pp0_stage0_subdone : BOOLEAN;
    signal ap_condition_pp0_exit_iter0_state2 : STD_LOGIC;
    signal ap_enable_reg_pp0_iter1 : STD_LOGIC := '0';
    signal ap_block_pp0_stage0 : BOOLEAN;
    signal grp_fu_194_p3 : STD_LOGIC_VECTOR (15 downto 0);
    signal grp_fu_181_p3 : STD_LOGIC_VECTOR (15 downto 0);
    signal tmp_8_fu_176_p2 : STD_LOGIC_VECTOR (15 downto 0);
    signal tmp_8_2_fu_189_p2 : STD_LOGIC_VECTOR (15 downto 0);
    signal ap_CS_fsm_state4 : STD_LOGIC;
    attribute fsm_encoding of ap_CS_fsm_state4 : signal is "none";
    signal ap_NS_fsm : STD_LOGIC_VECTOR (2 downto 0);
    signal ap_idle_pp0 : STD_LOGIC;
    signal ap_enable_pp0 : STD_LOGIC;

    component prodMatVect_mul_mbkb IS
    generic (
    ID : INTEGER;
    NUM_STAGE : INTEGER;
    din0_WIDTH : INTEGER;
    din1_WIDTH : INTEGER;
    dout_WIDTH : INTEGER );
    port (
    din0 : IN STD_LOGIC_VECTOR (15 downto 0);
    din1 : IN STD_LOGIC_VECTOR (15 downto 0);
    dout : OUT STD_LOGIC_VECTOR (15 downto 0) );
    end component;
```

Fig. 35: VHDL code from prodMatVect.vhd, generated by Vivado HLS, part three.

```
        component prodMatVect_mac_mcud IS
        generic (
        ID : INTEGER;
        NUM_STAGE : INTEGER;
        din0_WIDTH : INTEGER;
        din1_WIDTH : INTEGER;
        din2_WIDTH : INTEGER;
        dout_WIDTH : INTEGER );
        port (
        din0 : IN STD_LOGIC_VECTOR (15 downto 0);
        din1 : IN STD_LOGIC_VECTOR (15 downto 0);
        din2 : IN STD_LOGIC_VECTOR (15 downto 0);
        dout : OUT STD_LOGIC_VECTOR (15 downto 0) );
        end component;



        begin
        prodMatVect_mul_mbkb_U1 : component
            prodMatVect_mul_mbkb
        generic map (
        ID => 1,
        NUM_STAGE => 1,
        din0_WIDTH => 16,
        din1_WIDTH => 16,
        dout_WIDTH => 16)
        port map (
        din0 => a_0_q0,
        din1 => b_0_read_reg_221,
        dout => tmp_8_fu_176_p2);

        prodMatVect_mac_mcud_U2 : component
            prodMatVect_mac_mcud
        generic map (
        ID => 1,
        NUM_STAGE => 1,
        din0_WIDTH => 16,
        din1_WIDTH => 16,
        din2_WIDTH => 16,
        dout_WIDTH => 16)
        port map (
        din0 => a_1_q0,
        din1 => b_1_read_reg_231,
        din2 => tmp_8_fu_176_p2,
        dout => grp_fu_181_p3);
```

Fig. 36: VHDL code from prodMatVect.vhd, generated by Vivado HLS, part four.

```
        prodMatVect_mul_mbkb_U3 : component prodMatVect_mul_mbkb
        generic map (
        ID => 1,
        NUM_STAGE => 1,
        din0_WIDTH => 16,
        din1_WIDTH => 16,
        dout_WIDTH => 16)
        port map (
        din0 => a_2_q0,
        din1 => b_2_read_reg_241,
        dout => tmp_8_2_fu_189_p2);

        prodMatVect_mac_mcud_U4 : component prodMatVect_mac_mcud
        generic map (
        ID => 1,
        NUM_STAGE => 1,
        din0_WIDTH => 16,
        din1_WIDTH => 16,
        din2_WIDTH => 16,
        dout_WIDTH => 16)
        port map (
        din0 => a_3_q0,
        din1 => b_3_read_reg_251,
        din2 => tmp_8_2_fu_189_p2,
        dout => grp_fu_194_p3);





        ap_CS_fsm_assign_proc : process(ap_clk)
        begin
        if (ap_clk'event␣and␣ap_clk␣=␣␣'1')␣then
␣␣␣␣␣␣␣␣␣if␣(ap_rst␣=␣'1')␣then
␣␣␣␣␣␣␣␣␣ap_CS_fsm␣<=␣ap_ST_fsm_state1;
␣␣␣␣␣␣␣␣␣else
␣␣␣␣␣␣␣␣␣ap_CS_fsm␣<=␣ap_NS_fsm;
␣␣␣␣␣␣␣␣␣end␣if;
␣␣␣␣␣␣␣␣␣end␣if;
␣␣␣␣␣␣␣␣␣end␣process;
␣␣␣␣␣␣␣␣␣
```

Fig. 37: VHDL code from prodMatVect.vhd, generated by Vivado HLS, part five.

```
        ap_enable_reg_pp0_iter0_assign_proc : process(ap_clk)
        begin
        if (ap_clk'event␣and␣ap_clk␣=␣␣'1')␣then
␣␣␣␣␣␣␣␣␣if␣(ap_rst␣=␣'1')␣then
␣␣␣␣␣␣␣␣␣ap_enable_reg_pp0_iter0␣<=␣ap_const_logic_0;
␣␣␣␣␣␣␣␣␣else
␣␣␣␣␣␣␣␣␣if␣(((ap_const_logic_1␣=␣ap_CS_fsm_pp0_stage0)␣and␣
    (ap_const_logic_1␣=␣ap_condition_pp0_exit_iter0_state2)␣
    and␣(ap_const_boolean_0␣=␣ap_block_pp0_stage0_subdone)))␣
    then
␣␣␣␣␣␣␣␣␣ap_enable_reg_pp0_iter0␣<=␣ap_const_logic_0;
␣␣␣␣␣␣␣␣␣elsif␣(((ap_start␣=␣ap_const_logic_1)␣and␣
    (ap_const_logic_1␣=␣ap_CS_fsm_state1)))␣then
␣␣␣␣␣␣␣␣␣ap_enable_reg_pp0_iter0␣<=␣ap_const_logic_1;
␣␣␣␣␣␣␣␣␣end␣if;
␣␣␣␣␣␣␣␣␣end␣if;
␣␣␣␣␣␣␣␣␣end␣if;
␣␣␣␣␣␣␣␣␣end␣process;


␣␣␣␣␣␣␣␣␣ap_enable_reg_pp0_iter1_assign_proc␣:␣process(ap_clk)
␣␣␣␣␣␣␣␣␣begin
␣␣␣␣␣␣␣␣␣if␣(ap_clk'event and ap_clk =  '1') then
        if (ap_rst = '1') then
        ap_enable_reg_pp0_iter1 <= ap_const_logic_0;
        else
        if (((ap_const_logic_1 =
            ap_condition_pp0_exit_iter0_state2) and
            (ap_const_boolean_0 =
            ap_block_pp0_stage0_subdone))) then
        ap_enable_reg_pp0_iter1 <= (ap_const_logic_1 xor
            ap_condition_pp0_exit_iter0_state2);
        elsif ((ap_const_boolean_0 =
            ap_block_pp0_stage0_subdone)) then
        ap_enable_reg_pp0_iter1 <= ap_enable_reg_pp0_iter0;
        elsif (((ap_start = ap_const_logic_1) and
            (ap_const_logic_1 = ap_CS_fsm_state1))) then
        ap_enable_reg_pp0_iter1 <= ap_const_logic_0;
        end if;
        end if;
        end if;
        end process;
```

Fig. 38: VHDL code from prodMatVect.vhd, generated by Vivado HLS, part six.

```
        i_reg_140_assign_proc : process (ap_clk)
        begin
        if (ap_clk'event␣and␣ap_clk␣=␣'1')␣then
␣␣␣␣␣␣␣␣␣if␣(((ap_const_boolean_0␣=␣
    ap_block_pp0_stage0_11001)␣and␣(exitcond1_fu_151_p2␣=␣
    ap_const_lv1_0)␣and␣(ap_enable_reg_pp0_iter0␣=␣
    ap_const_logic_1)␣and␣(ap_const_logic_1␣=␣
    ap_CS_fsm_pp0_stage0)))␣then
␣␣␣␣␣␣␣␣␣i_reg_140␣<=␣i_1_fu_157_p2;
␣␣␣␣␣␣␣␣␣elsif␣(((ap_start␣=␣ap_const_logic_1)␣and␣
    (ap_const_logic_1␣=␣ap_CS_fsm_state1)))␣then
␣␣␣␣␣␣␣␣␣i_reg_140␣<=␣ap_const_lv3_0;
␣␣␣␣␣␣␣␣␣end␣if;
␣␣␣␣␣␣␣␣␣end␣if;
␣␣␣␣␣␣␣␣␣end␣process;
␣␣␣␣␣␣␣␣␣process␣(ap_clk)
␣␣␣␣␣␣␣␣␣begin
␣␣␣␣␣␣␣␣␣if␣(ap_clk'event and ap_clk = '1') then
        if (((ap_const_boolean_0 =
            ap_block_pp0_stage0_11001) and
            (exitcond1_fu_151_p2 = ap_const_lv1_0) and
            (ap_const_logic_1 = ap_CS_fsm_pp0_stage0))) then
        b_0_read_reg_221 <= b_0;
        b_1_read_reg_231 <= b_1;
        b_2_read_reg_241 <= b_2;
        b_3_read_reg_251 <= b_3;
        i1_reg_211(2 downto 0) <= i1_fu_163_p1(2 downto 0);
        end if;
        end if;
        end process;
        process (ap_clk)
        begin
        if (ap_clk'event␣and␣ap_clk␣=␣'1')␣then
␣␣␣␣␣␣␣␣␣if␣(((ap_const_boolean_0␣=␣
    ap_block_pp0_stage0_11001)␣and␣(ap_const_logic_1␣=␣
    ap_CS_fsm_pp0_stage0)))␣then
␣␣␣␣␣␣␣␣␣exitcond1_reg_202␣<=␣exitcond1_fu_151_p2;
␣␣␣␣␣␣␣␣␣end␣if;
␣␣␣␣␣␣␣␣␣end␣if;
␣␣␣␣␣␣␣␣␣end␣process;
␣␣␣␣␣␣␣␣␣i1_reg_211(63␣downto␣3)␣<=␣
    "000000000000000000000000000000000000000000000000␣
    0000000000000";
␣␣␣␣␣␣␣␣␣
```

Fig. 39: VHDL code from prodMatVect.vhd, generated by Vivado HLS, part seven.

```
ap_NS_fsm_assign_proc : process (ap_start,
    ap_CS_fsm, ap_CS_fsm_state1,
    exitcond1_fu_151_p2, ap_enable_reg_pp0_iter0,
    ap_block_pp0_stage0_subdone)
begin
case ap_CS_fsm is
when ap_ST_fsm_state1 =>
if (((ap_start = ap_const_logic_1) and
    (ap_const_logic_1 = ap_CS_fsm_state1))) then
ap_NS_fsm <= ap_ST_fsm_pp0_stage0;
else
ap_NS_fsm <= ap_ST_fsm_state1;
end if;
when ap_ST_fsm_pp0_stage0 =>
if (not(((exitcond1_fu_151_p2 = ap_const_lv1_1) and
    (ap_enable_reg_pp0_iter0 = ap_const_logic_1) and
    (ap_const_boolean_0 =
    ap_block_pp0_stage0_subdone)))) then
ap_NS_fsm <= ap_ST_fsm_pp0_stage0;
elsif (((exitcond1_fu_151_p2 = ap_const_lv1_1) and
    (ap_enable_reg_pp0_iter0 = ap_const_logic_1) and
    (ap_const_boolean_0 =
    ap_block_pp0_stage0_subdone))) then
ap_NS_fsm <= ap_ST_fsm_state4;
else
ap_NS_fsm <= ap_ST_fsm_pp0_stage0;
end if;
when ap_ST_fsm_state4 =>
ap_NS_fsm <= ap_ST_fsm_state1;
when others =>
ap_NS_fsm <= "XXX";
end case;
end process;
a_0_address0 <= i1_fu_163_p1(2 - 1 downto 0);
```

Fig. 40: VHDL code from prodMatVect.vhd, generated by Vivado HLS, part eight.

```
a_0_ce0_assign_proc : process(ap_CS_fsm_pp0_stage0,
    ap_block_pp0_stage0_11001,
    ap_enable_reg_pp0_iter0)
begin
if (((ap_const_boolean_0 =
    ap_block_pp0_stage0_11001) and
    (ap_enable_reg_pp0_iter0 = ap_const_logic_1) and
    (ap_const_logic_1 = ap_CS_fsm_pp0_stage0))) then
a_0_ce0 <= ap_const_logic_1;
else
a_0_ce0 <= ap_const_logic_0;
end if;
end process;

a_1_address0 <= i1_fu_163_p1(2 - 1 downto 0);

a_1_ce0_assign_proc : process(ap_CS_fsm_pp0_stage0,
    ap_block_pp0_stage0_11001,
    ap_enable_reg_pp0_iter0)
begin
if (((ap_const_boolean_0 =
    ap_block_pp0_stage0_11001) and
    (ap_enable_reg_pp0_iter0 = ap_const_logic_1) and
    (ap_const_logic_1 = ap_CS_fsm_pp0_stage0))) then
a_1_ce0 <= ap_const_logic_1;
else
a_1_ce0 <= ap_const_logic_0;
end if;
end process;

a_2_address0 <= i1_fu_163_p1(2 - 1 downto 0);

a_2_ce0_assign_proc : process(ap_CS_fsm_pp0_stage0,
    ap_block_pp0_stage0_11001,
    ap_enable_reg_pp0_iter0)
begin
if (((ap_const_boolean_0 =
    ap_block_pp0_stage0_11001) and
    (ap_enable_reg_pp0_iter0 = ap_const_logic_1) and
    (ap_const_logic_1 = ap_CS_fsm_pp0_stage0))) then
a_2_ce0 <= ap_const_logic_1;
else
a_2_ce0 <= ap_const_logic_0;
end if;
end process;

a_3_address0 <= i1_fu_163_p1(2 - 1 downto 0);
```

Fig. 41: VHDL code from prodMatVect.vhd, generated by Vivado HLS, part nine.

```
a_3_ce0_assign_proc : process(ap_CS_fsm_pp0_stage0,
    ap_block_pp0_stage0_11001,
    ap_enable_reg_pp0_iter0)
begin
if (((ap_const_boolean_0 =
    ap_block_pp0_stage0_11001) and
    (ap_enable_reg_pp0_iter0 = ap_const_logic_1) and
    (ap_const_logic_1 = ap_CS_fsm_pp0_stage0))) then
a_3_ce0 <= ap_const_logic_1;
else
a_3_ce0 <= ap_const_logic_0;
end if;
end process;

ap_CS_fsm_pp0_stage0 <= ap_CS_fsm(1);
ap_CS_fsm_state1 <= ap_CS_fsm(0);
ap_CS_fsm_state4 <= ap_CS_fsm(2);
ap_block_pp0_stage0 <= not((ap_const_boolean_1 =
    ap_const_boolean_1));
ap_block_pp0_stage0_11001 <= not((ap_const_boolean_1
    = ap_const_boolean_1));
ap_block_pp0_stage0_subdone <=
    not((ap_const_boolean_1 = ap_const_boolean_1));
ap_block_state2_pp0_stage0_iter0 <=
    not((ap_const_boolean_1 = ap_const_boolean_1));
ap_block_state3_pp0_stage0_iter1 <=
    not((ap_const_boolean_1 = ap_const_boolean_1));

ap_condition_pp0_exit_iter0_state2_assign_proc :
    process(exitcond1_fu_151_p2)
begin
if ((exitcond1_fu_151_p2 = ap_const_lv1_1)) then
ap_condition_pp0_exit_iter0_state2 <=
    ap_const_logic_1;
else
ap_condition_pp0_exit_iter0_state2 <=
    ap_const_logic_0;
end if;
end process;
```

Fig. 42: VHDL code from prodMatVect.vhd, generated by Vivado HLS, part ten.

```
        ap_done_assign_proc : process(ap_CS_fsm_state4)
        begin
        if ((ap_const_logic_1 = ap_CS_fsm_state4)) then
        ap_done <= ap_const_logic_1;
        else
        ap_done <= ap_const_logic_0;
        end if;
        end process;

        ap_enable_pp0 <= (ap_idle_pp0 xor ap_const_logic_1);

        ap_idle_assign_proc : process(ap_start,
            ap_CS_fsm_state1)
        begin
        if (((ap_start = ap_const_logic_0) and
            (ap_const_logic_1 = ap_CS_fsm_state1))) then
        ap_idle <= ap_const_logic_1;
        else
        ap_idle <= ap_const_logic_0;
        end if;
        end process;


        ap_idle_pp0_assign_proc :
            process(ap_enable_reg_pp0_iter0,
            ap_enable_reg_pp0_iter1)
        begin
        if (((ap_enable_reg_pp0_iter0 = ap_const_logic_0)
            and (ap_enable_reg_pp0_iter1 =
            ap_const_logic_0))) then
        ap_idle_pp0 <= ap_const_logic_1;
        else
        ap_idle_pp0 <= ap_const_logic_0;
        end if;
        end process;


        ap_ready_assign_proc : process(ap_CS_fsm_state4)
        begin
        if ((ap_const_logic_1 = ap_CS_fsm_state4)) then
        ap_ready <= ap_const_logic_1;
        else
        ap_ready <= ap_const_logic_0;
        end if;
        end process;

        c_address0 <= i1_reg_211(2 - 1 downto 0);
```

Fig. 43: VHDL code from prodMatVect.vhd, generated by Vivado HLS, part eleven.

```
        c_ce0_assign_proc : process(ap_CS_fsm_pp0_stage0,
            ap_block_pp0_stage0_11001,
            ap_enable_reg_pp0_iter1)
        begin
        if (((ap_const_boolean_0 =
            ap_block_pp0_stage0_11001) and (ap_const_logic_1
            = ap_CS_fsm_pp0_stage0) and
            (ap_enable_reg_pp0_iter1 = ap_const_logic_1)))
            then
        c_ce0 <= ap_const_logic_1;
        else
        c_ce0 <= ap_const_logic_0;
        end if;
        end process;

        c_d0 <= std_logic_vector(signed(grp_fu_194_p3) +
            signed(grp_fu_181_p3));

        c_we0_assign_proc : process(exitcond1_reg_202,
            ap_CS_fsm_pp0_stage0, ap_block_pp0_stage0_11001,
            ap_enable_reg_pp0_iter1)
        begin
        if (((ap_const_boolean_0 =
            ap_block_pp0_stage0_11001) and
            (exitcond1_reg_202 = ap_const_lv1_0) and
            (ap_const_logic_1 = ap_CS_fsm_pp0_stage0) and
            (ap_enable_reg_pp0_iter1 = ap_const_logic_1)))
            then
        c_we0 <= ap_const_logic_1;
        else
        c_we0 <= ap_const_logic_0;
        end if;
        end process;

        exitcond1_fu_151_p2 <= "1" when (i_reg_140 =
            ap_const_lv3_4) else "0";
        i1_fu_163_p1 <=
            std_logic_vector(IEEE.numeric_std.resize(
            unsigned(i_reg_140),64));
        i_1_fu_157_p2 <=
            std_logic_vector(unsigned(i_reg_140) +
            unsigned(ap_const_lv3_1));
        end behav;
```

Fig. 44: VHDL code from prodMatVect.vhd, generated by Vivado HLS, part twelve.

```
-- VHDL Model Created for "system ControllerprodMatVectModule"
-- 4/7/2018 9:49:46.597984
-- Alpha2Vhdl Version 0.9

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_signed.all;
USE IEEE.numeric_std.all;
ENTITY ControllerprodMatVectModule IS

GENERIC(
counterDelay: INTEGER := 0
);
PORT(
clk: IN STD_LOGIC;
CE : IN STD_LOGIC;
Rst : IN STD_LOGIC;
counter: OUT INTEGER;
dXctl1Out : OUT STD_LOGIC
);
END ControllerprodMatVectModule;

ARCHITECTURE BEHAVIOURAL OF ControllerprodMatVectModule IS

-- Declaration of the states
TYPE state_type IS (initState, trueState, falseState, finalState);
ATTRIBUTE ENUM_ENCODING: STRING;
ATTRIBUTE ENUM_ENCODING OF state_type : TYPE IS "00 01 10 11";

SIGNAL curStatedXctl1Out, nextStatedXctl1Out : STATE_TYPE;

BEGIN
```

Fig. 45: VHDL code from ControllerprodMatVectModule.vhd, generated by MMAlpha, part one.

```
        -- Synchronous reset process
        PROCESS (clk, rst)
        BEGIN
        IF clk = '1' AND clk'event␣THEN
␣␣␣␣␣␣␣␣␣IF␣CE␣=␣'1'␣THEN
␣␣␣␣␣␣␣␣␣IF␣rst␣=␣'0'␣THEN
␣␣␣␣␣␣␣␣␣counter␣<=␣0;
␣␣␣␣␣␣␣␣␣curStatedXctl1Out␣<=␣initState;
␣␣␣␣␣␣␣␣␣ELSE
␣␣␣␣␣␣␣␣␣counter␣<=␣counter␣+␣1;
␣␣␣␣␣␣␣␣␣curStatedXctl1Out␣<=␣nextStatedXctl1Out;
␣␣␣␣␣␣␣␣␣END␣IF;
␣␣␣␣␣␣␣␣␣END␣IF;
␣␣␣␣␣␣␣␣␣END␣IF;
␣␣␣␣␣␣␣␣␣END␣PROCESS;

␣␣␣␣␣␣␣␣␣--␣Controller␣for␣signal␣dXctl1Out
␣␣␣␣␣␣␣␣␣PROCESS(counter,␣curStatedXctl1Out)
␣␣␣␣␣␣␣␣␣BEGIN
␣␣␣␣␣␣␣␣␣CASE␣curStatedXctl1Out␣IS
␣␣␣␣␣␣␣␣␣WHEN␣initState␣=>␣IF␣counter␣=␣1␣THEN
␣␣␣␣␣␣␣␣␣nextStatedXctl1Out␣<=␣trueState;
␣␣␣␣␣␣␣␣␣ELSE␣nextStatedXctl1Out␣<=␣initState;
␣␣␣␣␣␣␣␣␣END␣IF;
␣␣␣␣␣␣␣␣␣WHEN␣trueState␣=>␣IF␣counter␣=␣2␣THEN
␣␣␣␣␣␣␣␣␣nextStatedXctl1Out␣<=␣falseState;
␣␣␣␣␣␣␣␣␣ELSE␣nextStatedXctl1Out␣<=␣trueState;
␣␣␣␣␣␣␣␣␣END␣IF;
␣␣␣␣␣␣␣␣␣WHEN␣falseState␣=>␣IF␣counter␣=␣9␣THEN
␣␣␣␣␣␣␣␣␣nextStatedXctl1Out␣<=␣finalState;
␣␣␣␣␣␣␣␣␣ELSE␣nextStatedXctl1Out␣<=␣falseState;
␣␣␣␣␣␣␣␣␣END␣IF;
␣␣␣␣␣␣␣␣␣WHEN␣OTHERS␣=>␣nextStatedXctl1Out␣<=␣finalState;
␣␣␣␣␣␣␣␣␣END␣CASE;
␣␣␣␣␣␣␣␣␣END␣PROCESS;
␣␣␣␣␣␣␣␣␣
```

Fig. 46: VHDL code from ControllerprodMatVectModule.vhd, generated by MMAlpha, part two.

```
-- Output function for signal dXctl1Out
PROCESS(curStatedXctl1Out)
BEGIN
CASE curStatedXctl1Out is
WHEN initState => dXctl1Out <= '0';
WHEN falseState => dXctl1Out <= '0';
WHEN trueState => dXctl1Out <= '1';
WHEN finalState => dXctl1Out <= '0';
WHEN others => dXctl1Out <= '0';
END CASE;
END PROCESS;
END BEHAVIOURAL;
```

Fig. 47: VHDL code from ControllerprodMatVectModule.vhd, generated by MMAlpha, part three.

```
-- VHDL Model Created for "system␣prodMatVectModule"
-- 4/7/2018 9:49:46.768462
-- Alpha2Vhdl Version 0.9
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_signed.all;
USE IEEE.numeric_std.all;
PACKAGE TYPESprodMatVectModule IS
TYPE Array1To4OfBoolean IS  ARRAY (1 TO 4) OF
    STD_LOGIC;
TYPE Array1To4OfInteger IS  ARRAY (1 TO 4) OF
    SIGNED (15 DOWNTO 0);
END TYPESprodMatVectModule;
```

Fig. 48: VHDL code from prodMatVectModule.vhd, generated by MMAlpha.

```

-- VHDL Model Created for "system␣prodMatVectModule"
-- 4/7/2018 9:49:46.768462
-- Alpha2Vhdl Version 0.9

LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE IEEE.std_logic_signed.all;
USE IEEE.numeric_std.all;USE
    work.TYPESprodMatVectModule.all;


ENTITY prodMatVectModule IS
GENERIC(
counterDelay: INTEGER := 0
);
PORT(
clk: IN STD_LOGIC;
CE : IN STD_LOGIC;
Rst : IN STD_LOGIC;
aMirrIn : IN Array1To4OfInteger;
bMirrIn : IN Array1To4OfInteger;
dOut : OUT Array1To4OfInteger
);
END prodMatVectModule;

ARCHITECTURE behavioural OF prodMatVectModule IS

SIGNAL counter : INTEGER; -- Counter

SIGNAL dXctl1XInOut : Array1To4OfBoolean;
SIGNAL dXctl1 :  STD_LOGIC;
SIGNAL a01 : Array1To4OfInteger;
SIGNAL b01 : Array1To4OfInteger;
SIGNAL TSep2Out : Array1To4OfInteger;
SIGNAL dSepTime1 : Array1To4OfInteger;
SIGNAL TSep1 : Array1To4OfInteger;


-- Insert missing components here!
```

Fig. 49: VHDL code from prodMatVectModule.vhd, generated by MMAlpha.

```
        -- Components for calls to external functions

        -- Component for ControllerprodMatVectModule
        COMPONENT ControllerprodMatVectModule
        GENERIC(
        counterDelay: INTEGER := 0
        );
        PORT(
        clk: IN STD_LOGIC;
        CE : IN STD_LOGIC;
        Rst : IN STD_LOGIC;
        counter: OUT INTEGER;
        dXctl1Out : OUT STD_LOGIC
        );
        END COMPONENT;

        -- $MissingComponents$

        BEGIN

        -- Translation of the definition of a01
        G1 : FOR p IN 1 TO 4 GENERATE
        a01(p) <= aMirrIn(p);
        END GENERATE;

        -- Translation of the definition of b01
        G2 : FOR p IN 1 TO 4 GENERATE
        b01(p) <= bMirrIn(p);
        END GENERATE;

        -- Translation of the definition of dOut
        G3 : FOR p IN 1 TO 4 GENERATE
        dOut(p) <= resize("0000000000000000", 16) WHEN
            dXctl1XInOut(p) = '1' ELSE TSep2Out(p);
        END GENERATE;

        -- Translation of the definition of dSepTime1
        G4 : FOR p IN 1 TO 4 GENERATE
        PROCESS(clk) BEGIN IF (clk = '1' AND clk'EVENT)␣THEN
␣␣␣␣␣␣␣␣␣IF␣CE='1'␣THEN␣dSepTime1(p)␣<=␣dOut(p);␣END␣IF;
␣␣␣␣␣␣␣␣␣END␣IF;
␣␣␣␣␣␣␣␣␣END␣PROCESS;
␣␣␣␣␣␣␣␣␣END␣GENERATE;
␣␣␣␣␣␣␣␣␣
```

Fig. 50: VHDL code from packageprodMatVectModule.vhd, generated by MMAlpha.

```
        -- Translation of the definition of dXctl1XInOut
        G5 : FOR p IN 1 TO 4 GENERATE
        dXctl1XInOut(p) <= dXctl1;
        END GENERATE;

        -- Translation of the definition of TSep1
        G6 : FOR p IN 1 TO 4 GENERATE
        TSep1(p) <= resize((a01(p) * b01(p)), 16);
        END GENERATE;

        -- Translation of the definition of TSep2Out
        G7 : FOR p IN 1 TO 4 GENERATE
        TSep2Out(p) <= resize((dSepTime1(p) + TSep1(p)), 16);
        END GENERATE;

        G8 : ControllerprodMatVectModule
        GENERIC MAP (counterDelay => counterDelay)
        PORT MAP (clk, CE, rst, counter, dXctl1);

        END BEHAVIOURAL;
```

Fig. 51: VHDL code from prodMatVectModule.vhd, generated by MMAlpha.