School of Information and Communication Technology

AlgoSil User Manual

Version 1.02

2016/08/03

Table of Contents

1 Introduction

AlgoSil is a system-level compiler that maps a MATLAB code to the DRRA fabric. Currently AlgoSil is in the testing stages, so it may have several limitations and problems.

The rest of this document consists of five sections: execution requirements, creating an AlgoSil-compatible MATLAB code, mapping a MATLAB code to the DRRA fabric, DRRA specification, and AlgoSil limitations.

2 Requirements

Operating System: Microsoft Windows

Required tools: MATLAB, Mentor Graphics ModelSim

Note: It is also possible to use other HDL simulators (e.g. Cadence Incisive or Synosys VCS) but some AlgoSil-created script files for simulation should be modified manually.

3 Preparing an AlgoSil-compatible MATLAB code

This section describes how to prepare a MATLAB code suitable for mapping to the DRRA fabric using the AlgoSil compiler.

3.1 Pragma

Currently, AlgoSil does not have an automatic resource allocation mechanism. So in order to be able to map a MATLAB code to the fabric, the resource allocation task should be done manually, using some pragmas. Pragmas start with **%!** symbol in the code. There are two types of pragmas: line pragmas and block pragmas. Line pragmas are written as a comment in front of the related statement. Block pragmas define a region surrounding a group of statements.

```
[Line Pragma Example]
var1 = zeros(1, 32); %! RFILE<> [row, col]


[Block Pragma Example]
%! RESOURCE_SHARING_BEGIN
. . .
%! RESOURCE_SHARING_END
```

Line pragmas are mainly used for storage and processor allocation while block pragmas are mainly used for defining parallel regions.

The following subsections describe pragmas that should be used for allocating resources to different parts of the input code.

3.1.1 Storage allocation

When defining a variable in MATLAB, a DRRA storage resource should be allocated to it. DRRA storage can be one or more register file or memory. The general format for defining a storage pragma is as follows:

```
var1 = zeros(1, 64); %! RFILE<options> [row_s_index : row_e_index, col_s_index : col_e_index]
var2 = zeros(1, 64); %! MEM<options> [row_s_index : row_e_index, col_s_index : col_e_index]
```

In the above pragma, *RFILE* and *MEM* indicate that if the variable should be allocated to a register file or memory. Indices in the square bracket indicate the location of one or more cells from the processing or memory sections. row_s_index and row_e_index refer to a range in the row direction, and col_s_index and col_e_index refer to a range in the column direction. If there is only one cell in each of the directions, the colon : can be omitted (e.g. RFILE<>[0, 0:1]). The indices can be selected from 0 to (number of rows/columns – 1). The options in the angular bracket can contain many types of customization that will be discussed later in this section. The angular bracket can be omitted if there is no option inside it.

3.1.1.1 Storage initialization

For storage initialization, whether array initialization, or *zeros()* and *ones()* functions can be utilized.

Several examples of array initialization has been shown in the following:

```
var1 = [5:15]; %! RFILE< > [0, 1]
var2 = [4]; %! RFILE< > [0, 1]
var3 = [6 15 10 3 -2 11]; %! RFILE< > [0, 1]
var4 = [0:63] %! RFILE < > [0, 2:3]
```

In the above example, the resource allocation for *var1* means that it has 11 elements with the initial values from 5 to 15 that should be allocated to the register file in row=0 and column=1 of DRRA. In this case, var1(1)=5 is assigned to the first location of RFILE[0,1] and var1(11)=15 is assigned to the 10th location of RFILE[0,1]; the second variable, *var2* has only one element which will be assigned to the 11th location of RFILE[0,1]; the third variable, *var3* elements are assigned to locations 12th to 17th of RFILE[0,1]; and finally *var4* should be assigned to two different register files. The first 32 elements of var4 are assigned to RFILE[0,2] and the second 32 elements are assigned to RFILE[0,3].

Storage initialization can also be performed using zeros() and ones() functions. Since currently AlgoSil does not support multidimensional arrays, so the first argument of zeros and ones functions should always be 1 showing that it is a one dimensional array and the second argument indicates the number of elements that needs to be initialized with 0s or 1s.

The below example shows storage initialization using zeros/ones functions.

```
var5 = zeros(1, 512); %! MEM< > [0, 0]
var6 = ones(1, 32); %! RFILE< > [0, 0]
```

As can be seen *var5* has 512 elements that should be mapped to the SRAM memory in row=0 and column=0. Note that each SRAM word can contain 16 DRRA words; so in this case, *var5* will occupy 32 words of MEM[0,0] all initialized with 0s. The second variable, *var6* has 32 elements that are mapped to 32 elements of RFILE[0,0] with 1 as the initial value.

Note

SRAM Memory words are 256 bits that can contains 16 words of DRRA. So while mapping a vector variable to memory, the size of the vector should be a multiple of 16.

When assigning a variable or several variables to a storage unit, it is important to be careful about the capacity of that storage unit. Also it should be considered that a register file can only provide a maximum of 2 read and 2 write accesses in each clock cycle.

3.1.1.2 Options

Storage allocation pragma can have different options that are placed within the angular bracket. Currently there are three options for storage allocations which are distribution type, variable type, and indirect addressing mode.

Distribution Type:

Distribution type option is used when a variable is going to be allocated to more than one storage cell and in that case distribution type determines if the elements of variable should be distributed evenly between several cells or not. Below example shows different cases of storage distribution.

```
var1 = zeros(1, 36); %! RFILE<even_dist> [0, 0:1]
var2 = zeros(1, 36); %! RFILE<full_dist> [1, 0:1]
var3 = zeros(1, 36); %! RFILE< > [0, 2:3]
```

In the above example, *var1* elements are evenly distributed to two register file. So var1(1:18) is mapped to RFILE[0,0] and var1(19:36) is mapped to RFILE[0,1]. For *var2* the distribution is *full_dist*, which means that before filling a register file, the previous register file should be filled completely to its maximum capacity. So in this case, var2(1:32) is mapped to RFILE[1,0] and var2(33:36) is mapped to RFILE[1,1]. By default, when not specifying any distribution type, as the case of *var3*, the variable elements are distributed evenly between the storage units.

Variable type:

Variable type option determines if a variable is an input, an output, or an intermediate variable. This option is used for guiding Sylva system-level synthesis tool to determine input and output buffers and their active time duration. By default, when omitting this option, a variable is considered as an intermediate variable.

```
var1 = zeros(1, 10); %! RFILE<input> [0, 0]
var2 = zeros(1, 15); %! RFILE<output> [1, 0]
var3 = zeros(1, 36); %! RFILE<full_dist, input> [0, 1:2]
```

Indirect addressing mode:

Indirect addressing mode is used to determine if a variable is going to be used for indirect addressing.

```
addr = [0]; %! RFILE<address> [0, 0]
```

3.1.2 Processor allocation

All arithmetic statements in a MATLAB code that need DPU operations should be instrumented with processor allocation pragmas in order to specify which part of the DRRA fabric should perform that arithmetic computation. The general format for defining a processor pragma is as follows:

```
res = op1 + op2; %! DPU<options> [row_s_index : row_e_index, col_s_index : col_e_index]
```

In the above pragma, indices in the square bracket indicate the location of one or more DRRA cells. row_s_index and row_e_index refer to a range in the row direction, and col_s_index and col_e_index refer to a range in the column direction. If there is only one cell in each of the directions, the colon : can be omitted (e.g. DPU<>[0, 0:1]). The indices can be selected from 0 to (number of rows/columns – 1). The options in the angular bracket can contain many types of DPU customization that will be discussed later in this section. The angular bracket can be omitted if there is no option inside it.

Assigning more than one DPU to an arithmetic statement means that it the operands of that arithmetic statement has been mapped to more than one cell and they are going to be computed in parallel.

```
op1 = [1:64];         %! RFILE [0, 0:1]
op2 = [64:1];         %! RFILE [1, 0:1]
res  = zeros(1, 64); %! RFILE[0:1, 2]
res = op1 + op2;     %! DPU [0, 0:1]
```

In the above example, each of variables *op1*, *op2*, and *res* are assigned to two register files. So when assigning two DPUs to the arithmetic statement, the first DPU will work on the first set of register files and the second DPU will work on the second set of register files. Thus, DPU[0,0] is connected to RFILE[0,0], RFILE[1,0], RFILE[0,2] and DPU[0,1] is connected to RFILE[0,1], RFILE[1,1], RFILE[1,2].

Note that not all arithmetic operations need a processor allocation pragma. Only operations that are physically mapped to the DPU need a processor pragma. For example, if there are some address calculation statements, there is no need to write a pragma for them.

```
for i = 2 : 2 : 6
  s = i + 3;  % no pragma needed
  res(i) = op1(s+2) * op2(s+i); %! DPU[1,1]
end
```

Note

During storage and processor allocation, it is important to take care of the DRRA sliding window. Each cell can be directly connected to others cells in a distance of at most 2 cells to the right and 2 cells to the left.

3.1.2.1 Options

Processor allocation pragma can have different options that are placed within the angular bracket. Currently there are three options for storage allocations which are DPU mode, output port, and saturation mode.

DPU mode:

Most of the arithmetic operations are detected by the compiler itself; so there is no need to specify a DPU mode in the options. But there are some cases that finding the DPU mode is not easily possible (e.g. FFT mode). In those cases DPU mode should be explicitly specified.

```
Temp1 = Tr1(k) + Di(kh) .* Wi(twi); %! DPU<mod=5, out=0> [r+1, c]
Temp2 = Tr0(k) - Di(kh) .* Wi(twi);  %! DPU<mod=5, out=1> [r+1, c]
```

Output port:

As DPUs in DRRA have two outputs, in some DPU mode, there is a need to specify that which port should be utilized. In those cases, *out* keyword can be added to the options to indicate a specific output port (see the above example).

Saturation mode:

In fixed point arithmetic, in order to indicate if the DPU should saturate the results or not, *sat* keyword can be used in the pragma options.

```
imageAvg = imageA - imageB; %! DPU<sat> [0, 0]
```

3.1.3 Resource sharing region

Sometimes there is a need that several arithmetic statements share some resources. For example DRRA has a *multicast* mode where a register file can send data to more than one DPU. So in such cases, the compiler should be guided to avoid creating unnecessary instructions. For example, in the *multicast* mode, only one register file instruction (REFI) should be created for multiple DPU accesses. In order to create such regions a resource sharing block pragma is used as follows:

```
%! RESOURCE_SHARING_BEGIN
convolution1(k) = sum(imageA(  1:16) .* kernel); %! DPU<sat> [0, 2]
convolution2(k) = sum(imageA(17:32) .* kernel); %! DPU<sat> [1, 2]
%! RESOURCE_SHARING_END
```

In the above example, since the arithmetic statements are defined within a resource sharing region, so for variables (*kernel*) pointing to the same location of register files a *multicast* instruction is created. Other variables (*convolution1, convolution2 imageA(1:16), imageA(17:32)*) will have their normal instructions.

3.2 Utilizing MATLAB constructs

This section discusses utilizing some of MATLAB constructs. For a complete overview of the whole supporting constructs, refer to the examples in the *testcases* directory.

3.2.1 Constant variables

Constant variables can be used anywhere in the MATLAB code and they are replaced by their value after compilation.

```
row = 0;
col = 1;
iter_no = 10;
var_size = 32;
var1 = zeros(1, var_size); %! RFILE[row, col]
for i = 1 : iter_no
  . . .
end
```

3.2.2 Constant operations

AlgoSil and DRRA has limited support for constants in arithmetic operations. Currently, DPU has two modes for constant operations which are load-with-constant and add-with-constant. The following example shows the corresponding MATLAB codes.

```
vec1 = zeros(1, 16); %! RFILE[0,0]
vec2 = zeros(1, 16); %! RFILE[0,0]

vec1 = 5;           %! DPU [0,0]
vec2 = vec1 + 3; %! DPU[0,0]
```

If a constant appears in an arithmetic operation, it is passed as an operand in the DPU instruction. As currently in the DPU instruction, the number of available bits for an operand is limited to 8 bits, so the constant value should be in the range of [-128:+127].

3.2.3 Fixed-point arithmetic

In MATLAB, fixed-point variables are defined by fi() function. During compilation, any variable that is initialized with fi() function is considered as a fixed-point operand and all arithmetic statements involving a fixed-point variable will be considered as fixed-point operations. The compiler automatically configures the corresponding DPU in the fixed-point calculation mode.

DRRA fixed-point arithmetic calculation is in Q15 format and by default DPU truncates the extra bits of the result. So in order for the MATLAB results to be comparable to the fabric results, fi() function should be configured with proper parameters as shown in the following example.

```
imageA  = fi(zeros(1,   RFILE_SIZE), 1, 16, 15, 'RoundingMethod', 'Floor'); %! RFILE [0,0]
kernel   = fi(zeros(1, BLOCK_SIZE), 1, 16, 15, 'RoundingMethod', 'Floor'); %! RFILE [0,1]
conv      = fi(zeros(1, BLOCK_SIZE), 1, 16, 15, 'RoundingMethod', 'Floor'); %! RFILE [0,2]

conv(k) = sum(imageA(1:16) .* kernel); %! DPU<sat> [0,2]
```

3.2.4 Calculation with accumulation

AlgoSil and DRRA support different kinds of calculations with accumulation. The supported operations are Mulitply-Accumulation (MAC), Symmetric MAC, Preserving MAC, and Absolute Subtract-Accumulation.

The following example shows the corresponding MATLAB code for each of these operations.

```
res1 = sum(vec1 .* vec2);                   % MAC
res2 = sum(vec1 .* (vec2 + vec3));          % Symmetric MAC
res3 = res3 + sum(vec1 .* vec2);            % Preserving MAC
res4 = sum(abs(vec1 – vec2));               % Absolute Subtract-Accumulation
```

3.2.5 min/max functions

AlgoSil supports min/max calculation by using MATLAB min() and max() functions. The input should be a vector of integers and the result is the minimum or maximum of that vector.

```
vec = [2 : 30]; %! RFILE[0,0]
res = [0 0] %! RFILE[0,0]

res(1) = min(vec); %! DPU[0,0]
res(2) = max(vec); %! DPU[0,0]
```

3.2.6 Conditional statements

Currently AlgoSil supports if statement and if-then-else statement outside loops and without nested conditional statements. In order to compute the condition of if statement, it should be mapped to a DRRA cell. This mapping can be done manually or automatically. If user does not mention any processor allocation for the if condition, it will always be automatically mapped to DPU of the cell in row=0 and column=0.

Some examples of conditional statements have been shown in the following:

```
% The condition will be mapped to DPU[0,0]
if A(2) == B(1)
   B(1) = B(1) + A(2); %! DPU [0,0]
end
```

```
if A(2) == B(1) %! DPU[0,0]
   B(1) = B(1) + A(2); %! DPU [1,0]
end
```

```
if A(2) == B(1) %! DPU[0,0]
   B(1) = B(1) + A(2); %! DPU [0,0]
else
   B(2) = B(2) + A(3); %! DPU [0,0]
end
```

Constant values are supported in the condition and body of if statement. But note that as there are limited space for placing constant values in the instruction (currently 8-bits), the value should be within the valid range.

```
if A(2) == 1
   B(1) = A(2) + 1; %! DPU [0,0]
else
   B(2) = A(2) + 2; %! DPU [0,0]
end
```

For usage limitations of conditional statements see section 6.2.

4 Mapping a MATLAB code to the DRRA fabric

In this part, the structure of the workspace is presented in section 4.1. Section 4.2 describes how to run a MATLAB testcases and map it to the fabric and section 4.3 describes the files that are automatically generated by AlgoSil in more detail.

4.1 Workspace organization

Both the DRRA fabric and the AlgoSil compiler are located in the *workspace* directory. The *workspace* directory is organized as follows:

```
- workspace
  | - algosil
      | - bin
      |   | + required dlls
      | + debug
      | + doc
      | + make
      | - test
          | + test_util
          | + testbenches
          | + testcases
  | - fabric
      | + dimarch
      | + mtrf
```

A brief description of the subdirectories has been shown in the following table.

workspace	The main directory
algosil	The AlgoSil compiler directory
bin	Location of the release libraries and executable files
required dlls	required visual studio DLLs
debug	Location of the debug libraries and executable files
doc	Corresponding documentations
make	Make files for automatically running testcases
test	The main directory for testing
test_util	Location of some utility functions for testing
testbenches	Location of the automatically created test folders after running testcases
testcases	Location of the MATLAB source codes for mapping to the fabric
fabric	The DRRA fabric directory
dimarch	Memory part of the fabric
mtrf	DRRA cells

4.2 Running testcases

The simplest way to map a MATLAB code to the DRRA fabric is by using make files. Below shows a sample make file that is located in the *make* directory.

```
:: ------------ Setting Variables -------------
set file_name=test01 test02
set files_path=..\test\testcases
set test_path=..\test\testbenches

:: ----------- AlgoSil compilation -----------
cd ..\bin
for %%a in (%file_name%) do (
        AlgoSil.exe %files_path%\%%a.m
)
:: --------------- Simulation ---------------
cd code
for %%a in (%file_name%) do (
        call make_firstrun_%%a.bat
)
:: --------------- Comparison ---------------
cd ..
for %%a in (%file_name%) do (
        CompareResult.exe %test_path%\%%a
)

pause
```

As can be seen in the make file, it has 4 main sections for setting variables, AlgoSil compilation, HDL simulation, and result comparison. In the first section, the name of the MATLAB file(s) should be given to the *file_name* variable. If there are more than one file, they can be added to the same line with space delimiter. Usually MATLAB files are located in the *testcases* directory. For reading from other locations, the *files_path* variable should be modified.

In the AlgoSil compilation section, the compiler is called for each of the testcase files. By default, AlgoSil uses a dynamic mechanism (RACCU) for nested loops of more than 2 levels. For static loop unrolling, there is an optional argument '-unroll' that can be added to the AlgoSil command in order to use loop unrolling instead of dynamic loop management.

Note

If AlgoSil fails because of a visual studio DLL missing, the needed DLLs should be copied from the *"required dlls"* directory according to the OS type (x86/x64). Alternatively, the Microsoft Visual C++ redistributable package for Visual Studio 2012 can be installed from the following address: http://www.microsoft.com/en-us/download/details.aspx?id=30679.

In the simulation section, an AlgoSil-created batch file is called for each of the testcases. The main tasks of this batch file are creating a working folder with the same name of a testcase in the *testbenches* directory, copying several generated files to that folder, and then simulating the testbench with MATLAB and ModelSim simulators.

And finally in the comparison section, the results from MATLAB and HDL simulation are compared and it will give a PASS or FAIL notification at the end of its execution.

There are already several testcases in the *testcase* directory that can be run using several make files in the *regression* folder of the *make* directory.

4.3 Generated files in more detail

There are several files that are created by AlgoSil during a testcase compilation. These files are mainly used for automating the simulation process. The below table shows a list of the generated files and a brief description about them.

const_package_[name].vhd	It contains some constant values corresponding to the testcase. These constants are used by *profiler.sv* and *testbench.vhd*.
instrumented_[name].m	This file adds several printing functions to the original MATLAB code. It is used for generating the golden values from the original code.
instructions_*[name]*.txt	This file contains the created instructions in all DRRA cell sequencers in a readable manner. It is mostly used for debugging purposes and it will be moved to the *report* folder in the testcase working directory after running the *make_firstrun* script.
make_firstrun_[name].bat	A script file used for creating a working folder for the testcase in the *testbenches* directory, moving all required files to that folder, and then performing an automatic MATLAB/ModelSim simulation.
make_rerun_[name].bat	A script file used to automate the MATLAB/ModelSim simulation of the testcase for several times.
manas_[name].c	It is the DRRA machine code that is not used during simulation.
profiler_[name].sv	This file collects data from the DRRA fabric during its execution. It generates some files for comparing the actual results with the original MATLAB results.
run_cmd_[name].do	A script file for running the example in the ModelSim command-line mode. It is used by *make_firstrun* and *make_rerun* script files.
run_gui_[name].do	A script file for running the testcase in the ModelSim graphical interface mode.
testbench_[name].vhd	This file is the generated VHDL testbench file for mapping the application to the DRRA fabric. It generates the initial values for all register files and memories and also loads the proper instructions to the DRRA fabric sequencers.

timing_*[name]*.txt	This file contains the timing model of the given testcase which is used by SYLVA system-level synthesis tool. It is moved to the *report* folder in the testcase working directory after running the *make_firstrun* script.
wave_[name].do	A script file to facilitate viewing results in the ModelSim Waveform Viewer.

After simulation, several files are created in the *result* folder of the testcase working directory. These files are briefly described in the below table.

mt_results_[name].txt	This file contains all value changes of variables, extracted from the MATLAB code. It is used as the golden model by the *ResultCompare* tool for verifying the actual results.
mt_results1_exc_[name].txt	This file captures all events of execution in the MATLAB code.
mt_results2_reg_[name].txt	This file contains the initial and final values of MATLAB variables.
sv_results_[name].txt	This file contains all value changes of register files, extracted from the HDL simulation. It is used as the hardware model by the *ResultCompare* tool for verifying the actual results.
sv_results1_exc_[name].txt	This file captures all events of execution in the HDL simulation.
sv_results2_reg_[name].txt	This file contains register file values during the HDL simulation.
sv_results3_seq_[name].txt	This file contains sequencer instruction changes during the HDL simulation.

Even though, the *ResultCompare* tool automatically compares the results from MATLAB and HDL simulation, but for a deeper debugging in case of failures, the aforementioned files can be taken into consideration.

5 DRRA specification

During MATLAB code preparation by adding pragmas, DRRA specification should be taken into consideration.

Current DRRA specification:

- DRRA words are 16 bits.
- It has 2 rows and 5 columns of DRRA cell.
- It has 1 row and 5 columns of DiMArch SRAM.
- Sliding window: direct access to adjacent cells is within 2 columns (left or right) and 1 row (top or bottom).
- The register file size is 32 and its width is 16 bits.
- The register file has two read and two write ports.
- SRAM size is 128 and its width is 256 bits. So each SRAM word fits 16 DRRA words.
- RACCU supports 4 levels of loop.
- RACCU register file depth is 8 (supporting 8 variables).

- Sequencer instruction register size is 64.
- DPU fixed-point operation is in Q15 format.
- The number of available bits for a constant value within a constant instruction is limited to 8 bits, so the constant value should be in the range of [-128:+127].

6 Supporting MATLAB subset and limitations

Only a subset of MATLAB is supported as the input code. There are many examples in the testcase directory that can be investigated to find the supporting subset.

6.1 General limitations

Some of the important limitations are the following:

- Limited support for MATLAB functions
  - Supported functions are: zeros, ones, min, max, sum, fi, abs.
- No support for multidimensional arrays
- Supports only one DPU operation in each assignment statement
  - The corresponding statement for a DPU operation can contain several MATLAB operations, For example, *result = sum(vector1 .* vector2)* implements a MAC operation.
- Limited support for arithmetic operations to be mapped to DPU
  - Supported operations: addition, subtraction, multiplication, MAC, Symmetric MAC, Preserving MAC, Comparison, subtract-accumulation, absolute subtract-accumulation.
- Address functions should be affined.
- Nested loops of maximum 6 levels are supported.

6.2 Conditional statement limitations

- Limited supports for conditional statements inside loop
- No nested conditional statements
- The supported expression in the if-condition is a binary operation with the following operators: **<, <=, >, >=, ==, ~=**
- As currently a DRRA cell is not capable of changing the program counter of another DRRA cell, so the following limitations should be considered when allocating DPUs:
  - For **<, <=, >, >=, ==** operators**,** the utilized DPU for the if condition should be the same with the DPU of the first statement in the else-part or the first statement after the if block statement (when there is no else-part).

- For **~=** operator, as it is converted to operator = by swapping the then-part and else-part, so the utilized DPU for the if condition should be the same with the DPU of the then-part.