

Maxeler DFE Debugging and Optimization Tutorial

Preview Version 45f1310



Maxeler DFE Debugging and Optimization Tutorial. Copyright © Maxeler Technologies.

Preview Version 45f1310

November 16, 2015

Contact Information

Sales/general information: info@maxeler.com

US Office

California Office

Maxeler Technologies Inc
1928 Old Middlefield Way
Mountain View
California, 94043, USA
Tel. +1 (650) 938 8818

Illinois Office

Maxeler Technologies Inc
215 W Washington, apt #2705
Chicago, IL 60606, USA.
Tel: +1 (312) 620 9711

UK Office

Maxeler Technologies Ltd
3-4 Albion Place
London W6 0QT, UK.
Tel: +44 (0) 203 714 0480

All rights reserved. The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this document may be reproduced or transmitted in any form by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the written permission of the copyright holder.

Maxeler Technologies and the Maxeler Technologies logo are registered trademarks of Maxeler Technologies, Inc. Other product or brand names may be trademarks or registered trademarks of their respective holders.

1	Introduction	1
2	Step by Step to Maxeler Applications	3
3	Simulation Debugging	5
3.1	Simulation watches	5
3.2	Simulation printf	9
3.2.1	Troubleshooting simPrintf does not produce any output	12
4	Hardware Debugging	15
4.1	DFE printf	15
5	Advanced Debugging	17
5.1	Introduction	17
5.2	Launching the MaxIDE's debugger	18
5.3	Kernel halted on input	19
5.4	Kernel halted on output	21
5.5	Stream status blocks	21
5.6	Deadlock	22
5.6.1	A Kernel has scheduled an input before an output	23
5.6.2	A FIFO ends up full	24
5.7	Debugging with MaxDebug	24
5.8	Debugging with MaxDump	34
5.9	Debug directories	35
6	Common Errors	37
6.1	Simulation Compilation Errors	37
6.1.1	Failed to create topology pool in daemon: topology request has timed out	37
6.2	Hardware Compilation Errors	38
6.2.1	Cost table failed	38
6.2.2	Using more resources then available	39
6.3	Runtime Errors	39
6.3.1	Stream alignment issues	39
7	Best DFE Optimization Practices	41
7.1	FramedKernel with regular or non-blocking inputs and outputs	41
7.2	Uninitialized data	41
7.3	Enable/Disable latency annotation	42
8	Selected DFE Design Optimizations	43
8.1	Kernel optimizations	43
8.1.1	Holistic optimization	43
8.1.2	Push-pop optimizations	44
8.1.3	Placement constraints	45
8.1.4	Input registering	45
8.1.5	Per-stream optimizations	46
8.2	Manager optimizations	46
8.2.1	LMem clock frequency	46
8.2.2	Stream clock frequency	48

8.3	Build configuration	48
8.3.1	setBuildEffort(Effort effort)	48
8.3.2	setEnablePhysicalSynthesisRegDuplication(Switch switch)	49
8.3.3	setMPPRCostTableSearchRange(int min, int max)	49
8.3.4	setMPPRParallelism(int numberOfThreads)	50
8.3.5	setMPPRRetryNearMissesThreshold(int retryTreshold)	51
8.3.6	setOptimizationGoal(OptimizationTechnique technique))	51
9	Global DFE Optimization Practices:	
	Getting Maximum Performance	53
9.1	Introduction	54
9.2	Dataflow computing strategy	54
9.3	Fitting procedure	54
9.3.1	Techniques to fit designs onto DFE	55
9.4	How to make it fit?	56
9.4.1	Stage 1: DFE logic utilization >100%: macro optimization	57
9.4.2	Stage 2: DFE logic utilization >80% and <100%: micro optimizations	58
9.4.3	Stage 3: DFE logic utilization <80%, frequency optimization	59
9.5	Optimizing memory bound applications - data size	60
9.5.1	Differences in memory controllers of different cards	60
9.5.2	Data specific compression	60
9.5.3	Data encoding	60
9.5.4	Re-organisation of the order of computations	60
9.6	Optimizing compute performance - clock frequency	60
10	Appendices	65
10.1	Appendix A: Using WebIDE	65
10.1.1	Simulation watches	65
10.1.2	Simulation printf	65
10.1.3	Viewing the execution graph	65
10.2	Appendix B: Datasheet - how much space does it take	66
10.3	Appendix C: Double or Nothing	67
10.3.1	Stratix V D8 (Maia) FMem	69
10.3.2	Stratix V D8 (Maia) DSP blocks	69
10.3.3	Stratix V ALM block	70
10.3.4	Operations	70
10.3.5	LMem bandwidth	71
10.3.6	Space, time, and value story	71
10.3.7	Floating-point in DFE design	72
	SLiC API Index	75
	MaxJ API Index	76
	Index	77

Welcome

Welcome to the Maxeler DFE Debugging and Optimization Tutorial. To achieve Maximum Performance Computing we strive to combine optimizations at all levels, starting from the algorithm level all the way down to the bit level. Different tools and methods for successful debug and optimization of a DFE design are presented in this tutorial.

The source code for the examples, exercise stubs, and solutions in this tutorial are included in the MaxCompiler distribution.

Document conventions

When important concepts are introduced for the first time, they appear in **bold**.

Italics are used for emphasis.

Directories and commands are displayed in `typewriter` font.

Variable and function names are also displayed in `typewriter` font.

Java methods and classes are shown using the following format:

```
DFEVar io.input(String name, DFEVar addr, DFEType type)
```

C function prototypes are similar:

```
max_engine_t* max_load(max_file_t* maxfile, const char* engine_id_pattern);
```

Actual Java usage is shown without italics:

```
io.output("output", myRom, dfeUInt(32));
```

C usage is similarly without italics:

```
PassThrough(DATA_SIZE, dataIn, dataOut);
```

Sections of code taken from the source of the examples appear with a border and line numbers:

```
1 package chap01_gettingstarted.ex1_passthrough;
2 import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
3 import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
4 import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
5
6 public class PassThroughKernel extends Kernel {
7     PassThroughKernel(KernelParameters parameters) {
8         super(parameters);
9     }
10
11     // Input
12     DFEVar x = io.input("x", dfeUInt(32));
13     // Output
14     io.output("y", x, dfeUInt(32));
15 }
```

1

Introduction

Open literature reveals that experienced developers of applications for the Maxeler dataflow platforms keep creating respectable performance gains over the more traditional control flow platforms. However, those who are enthusiastically entering this new computing arena may not be able to match immediately the results obtained by more experienced developers. Consequently, an effort is needed to help speed up the learning curve of new dataflow programmers. This tutorial represents one such effort.

This tutorial introduces the reader to specific knowledge necessary for successful debugging of Maxeler-based implementations, both during the simulation process and during the hardware testing. Also, Maxeler DFE Debugging and Optimization Tutorial covers typical errors that users may encounter during the compilation process. Explanations and solutions are provided. At the end, we introduce techniques to maximize the utilization of hardware and to improve the performance of applications.

After the introduction ([Section 1](#)), it is underlined that MaxCompiler offers a number of methods for debugging a DFE design ([Section 2](#)). Kernels could be debugged using **watches** ([Section 3.1](#)), **simulation printf** ([Section 3.2](#)), and **DFE printf** ([Section 4](#)). Watches tell us the value of any specified DFEVar for every tick of the Kernel run. Simulation printf and DFE printf allow us to print and format the print values explicitly from the streams within the Kernel, on every tick, optionally enabled using a Boolean stream. Watches and simPrintf are available at simulation time only and are ignored at DFE hardware runs, whereas dfePrintf is available for both simulation and hardware runs. More

advanced methods for debugging a DFE while it is running, or after its running is finished, are covered in [Section 5](#). Compilation errors are treated in [Section 6.1](#) and [Section 6.2](#). Runtime errors are treated in [Section 6.3](#). Ways to optimize a design and to utilize the hardware more effectively are given in [Section 7](#), [Section 8](#), and [Section 9](#). Appendices ([Section 10](#)) focus on how to use Maxeler's Web-based Integrated Development Environment (WebIDE) and how to analyse the execution graph. Also, debate on topic of fixed-point and floating-point precision is given in the same section.

Overall, sections 3 to 7 help to create a design that compiles without errors. [Section 6.3](#) helps create a design that runs without errors. Sections 9 to 11 help with the optimization process (less sophisticated optimizations first, in [Section 7](#) and [Section 8](#), followed by more sophisticated optimizations, in [Section 9](#)). Finally, the appendix introduces WebIDE as a tool, and sheds light on the ways to analyse the execution graph as the final output of the application development process.

2

Step by Step to Maxeler Applications

Over the years, Maxeler has developed a method on how to proceed from an algorithm to its fully accelerated implementation, utilizing the Maxeler's static dataflow approach. This section explains the method step by step. Steps from 1 to 5, given below, are rarely done only in single one pass. Typically, at each step, if a problem occurs, the developer has to go back to a previous step, to fix the problem and to continue. The sequence of such iterations could be repeated as many times as necessary. The five steps of the Maxeler method are:

1 Create a Baseline *C* application

The first step is to write a *C* implementation, the so called *C* model. This model could be used as a reference point against which the accelerated version could be compared. Debugging tools available in this process are widely spread and commonly known, so we will not discuss this matter here anymore. The result of this step (1) is a Baseline *C* application.

2 Determine code segments with a potential for acceleration

The next step is to locate the segments of the code that could be accelerated with DFEs. By the nature of dataflow, loops are the code structures targeted first. After we locate the segments of code with a potential for acceleration, we have to transform these segments of code from *C* to *MaxJ*. Other Maxeler tutorials may also be helpful regarding this matter ("Multiscale Dataflow Programming" and "Dataflow Programming for Networking"). Here we will assume that the pro-

programmer has created a *MaxJ* design. The result of this step (2) is a DFE design with a potential for acceleration, which yet has to be proved.

3 Debugging of designs at simulation time

Generating a working hardware implementation is time consuming, so initial functional testing should be done using simulation. Tools that are at programmer's disposal at this point are described in [Section 3](#). Some advanced debugging methods and problems that are not easy to detect (with appropriate solutions) are given in [Section 5](#). Some of the error messages possible during this process are explained in [Section 6.1](#). There are cases when the design passes the compilation process, but fails at runtime; for this matter see [Section 6.3](#). Over the years, Maxeler gathered many best practices, which are described in [Section 7](#). The guidelines related to local optimization are described in [Section 8](#). Global optimizations are also described in [Section 8](#) and are instrumental for maximizing the performance. The result of this step (3) is a design working at simulation time.

4 Debugging designs in hardware

After the programmer is satisfied with the simulation results, the next step is to proceed to hardware build. Maxeler offers tools to debug a design once it gets ported to hardware. Tools that are at programmer's disposal at this point are described in [Section 4](#). Getting an error message you can not cope with could be very frustrating; because of that, some of the error messages are explained in [Section 6.2](#). Programmers may also refer to other sections, as indicated in the previous step. The result of this step (4) is a design working in hardware.

5 Optimizations

Having a working design is never the final goal. It is only a *condition sine qua non* for what the final goal should be: Maximal acceleration. Issues related to speed maximization are in synergy with issues in better utilization of hardware resources. Useful information can be found in [Section 8](#) and in [Section 9](#). The result of this step (5) is a fully optimized DFE design.

3

Simulation Debugging

Simulation is the first step on the way to a fully working design. After the programmer has developed a DFE solution, it should be validated. Validating a design at the hardware level could be very time consuming, and quite often impossible, because of the prohibitively long hardware build times. To address this issue, MaxIDE offers programmers a simulator to recreate behavior of the design without building any hardware at all, which makes the development process much faster. MaxIDE offers two different ways for programmers to follow execution and to debug designs: Simulation watches (`simWatch`, [Section 3.1](#)) and simulation `printf` (`simPrintf`, [Section 3.2](#)).

3.1 Simulation watches

A watch could be added to a stream in simulation, to record the values of the stream elements on every Kernel tick. To add a watch, we use a method of the class `DFEVar` called `simWatch`, which takes a string argument that is used to label the watch. When a Kernel is run in the simulation mode, it generates a report in the comma-separated values (CSV) format, containing data for variables that have been watched.

[Listing 3.1](#) presents the moving average example with watches added to several of the streams. For

3.1 Simulation watches

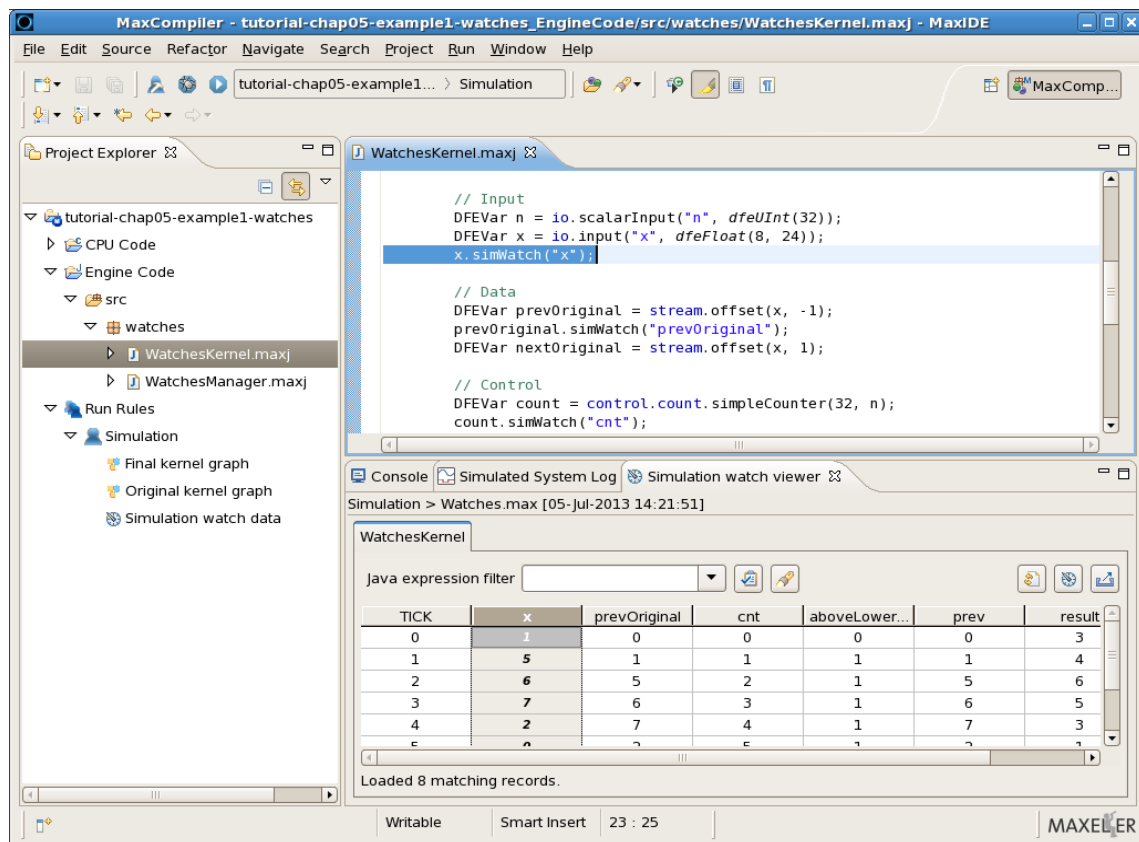


Figure 3.1: MaxIDE displaying simulation watch data

example, we can watch the input to the Kernel, described in [Listing 3.1](#):

```
22 DFEVar x = io.input("x", dfeFloat(8, 24));
23 x.simWatch("x");
```

We can also watch any intermediate values in the middle of the computation:

```
38 DFEVar prev = aboveLowerBound ? prevOriginal : 0;
39 prev.simWatch("prev");
```

Upon completion of the run, the CSV report could be accessed from within MaxIDE, by clicking on the “Simulation watch data” item that appears under the associated RunRule in the Project Explorer, as depicted in [Figure 3.1](#).

The Simulation watch viewer can display large datasets and makes it possible to filter, search through, and export the presented data:

- the data may be filtered using a “Java expression filter” text field: boolean expressions written in the Java language may be used to display only those records for which the expression is true. The string names used in the `simWatch` call may be used as variables in these expressions;
- the records displayed may be searched with the same expressions as in the Java expression filter;
- a column selection tool may be used to display selected columns only;

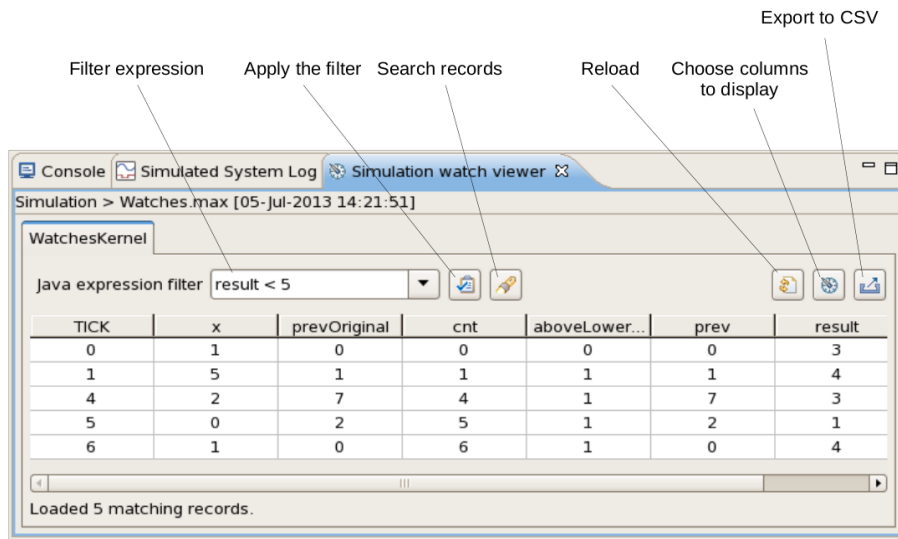


Figure 3.2: The MaxIDE simulation watch viewer utilities

- finally, the filtered data displayed may be exported in the CSV format, for use with external applications.

The tools and widgets used in the above operations are shown in [Figure 3.2](#).

When an application is run from within MaxIDE, the file containing the original data presented in the Simulation watch viewer appears under the debug directory of the RunRule folder of the project. When the application is run from the command line, this file appears under a debug directory in the working directory. This directory may be named debug or may have a name of the form debug_N, where N is an index used to make this directory name unique. The naming of the debug directory is further covered in [Section 5.9](#).

The output file name is formed by concatenating watch_ with the name of MaxFile and the name of your Kernel (e.g., watch_Watches.WatchesKernel.csv).

By default, debug output is produced for all ticks while the kernel is running. The function `max_watch_range` is used to limit debug output, only for ticks in a given range:

```
void max_watch_range(
    max_actions.t *actions,
    const char *kernel_name,
    int start_tick,
    int num_ticks);
```

Watches are also available in WebIDE; see: [Section 10.1.1](#).

3.1 Simulation watches

Listing 3.1: Program for the moving average Kernel with watches (WatchesKernel.maxj).

```
1  /**
2   * Document: MaxCompiler Tutorial (maxcompiler-tutorial.pdf)
3   * Chapter: 5      Example: 1      Name: Watches
4   * MaxFile name: Watches
5   * Summary:
6   *      Kernel that computes a three point moving average with boundaries,
7   *      while printing the watch information.
8   */
9  package watches;
10
11  import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
12  import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
13  import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
14
15  class WatchesKernel extends Kernel {
16
17      WatchesKernel(KernelParameters parameters) {
18          super(parameters);
19
20          // Input
21          DFEVar n = io.scalarInput("n", dfeUInt(32));
22          DFEVar x = io.input("x", dfeFloat(8, 24));
23          x.simWatch("x");
24
25          // Data
26          DFEVar prevOriginal = stream.offset(x, -1);
27          prevOriginal.simWatch("prevOriginal");
28          DFEVar nextOriginal = stream.offset(x, 1);
29
30          // Control
31          DFEVar count = control.count.simpleCounter(32, n);
32          count.simWatch("cnt");
33          DFEVar aboveLowerBound = count > 0;
34          DFEVar belowUpperBound = count < n - 1;
35          DFEVar withinBounds = aboveLowerBound & belowUpperBound;
36          aboveLowerBound.simWatch("aboveLowerBound");
37
38          DFEVar prev = aboveLowerBound ? prevOriginal : 0;
39          prev.simWatch("prev");
40          DFEVar next = belowUpperBound ? nextOriginal : 0;
41
42          DFEVar divisor = withinBounds ? constant.var(dfeFloat(8, 24), 3) : 2;
43
44          DFEVar result = (prev + x + next) / divisor;
45          result.simWatch("result");
46
47          // Output
48          io.output("y", result, dfeFloat(8, 24));
49      }
50  }
51 }
```

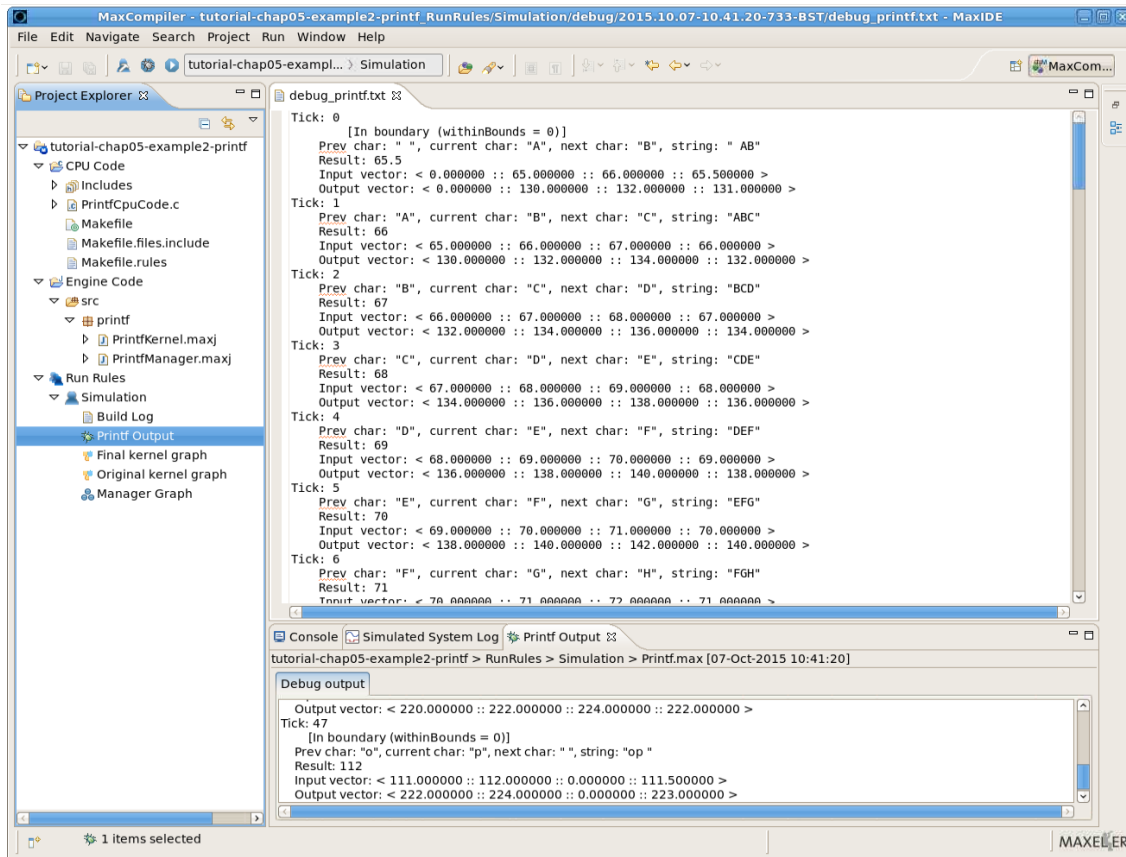


Figure 3.3: Simulation printf output in MaxIDE

3.2 Simulation printf

`simPrintf` behaves similarly to `printf` in C.

The basic form of `simPrintf` and `dfePrintf` prints the formatted message to a standard output of the application with the current value of the supplied list of objects for every tick:

```
void debug.simPrintf(String message, Object... args)
```

To print the message only when a certain condition is met, a Boolean `DfEVar` stream could be supplied as a condition:

```
void debug.simPrintf(DfEVar condition, String message, Object... args)
```

The message is only printed when `condition` is true, i.e., when `condition` evaluates to 1.

The argument(s) `args` may contain Java variables (`Byte`, `Short`, `Integer`, `Long`, `Float`, or `Double`) or `DfEVar` streams. The argument message should contain corresponding format specifiers. All familiar format specifiers from C `printf` are supported, with the only difference that `%o` prints in *binary*, rather than *octal*.

The outputs of `simPrintf` appear on the standard output of the application and are also saved to files upon completion of a run:

- the outputs of `simPrintf` are collated into a single `debug_printf.txt` file in the debug directory

3.2 Simulation printf

of the RunRule;

- `simPrintf` accepts an extra string argument `NAME` for outputs that should be written to a separate file:

```
void debug.simPrintf(String NAME, String message, Object... args)
void debug.simPrintf(String NAME, DFEVar condition, String message, Object... args)
```

The output will then appear in a file named `debug_printf_NAME.txt` in the debug directory.

[Listing 3.2](#) presents an example with several `simPrintfs`.

The first `simPrintf` shows the current value output by the counter, which gives us the current tick count on every Kernel tick:

```
34 debug.simPrintf("Tick: %d\n", count);
```

The second `simPrintf` conditionally prints a message when the current position in the stream is in a boundary:

```
38 debug.simPrintf("withinBounds, "[In boundary (withinBounds = %d)]\n", withinBounds);
```

The third `simPrintf` prints previous, current, and next values of the input `x` (as characters) and also prints these three characters concatenated as a string. If it gets some non-printable value, it prints spaces:

```
43 debug.simPrintf("Prev char: \"%c\", current char: \"%c\", next char: \"%c\", string: \"%s\"\n", prev, x, next, string);
```

The fourth `simPrintf` prints the result produced in the current Kernel tick:

```
48 debug.simPrintf("Result: %.3g\n", result);
```

The final `simPrintf` shows how we can print arguments of composite type by using custom specifier `%KObj%`. In this example, we have printed two `DFEVectors`:

```
57 debug.simPrintf("Input vector: %KObj%\nOutput vector: %KObj%\n", inVector, outVector);
```

The output from the application is presented in [Listing 3.3](#).

When the application is run from within MaxIDE, this output is also visible in files that appear under the associated RunRule in the Project Explorer, upon completion of the run, as seen in [Figure 3.3](#). The on-disk file containing the `simPrintf` outputs, similar to watch data, is available in the debug directory of the associated RunRule folder.

When the application is run from the command line, the output files are available in the debug sub-directory of the current directory, or in a directory of the form `debug_N`, where `N` is chosen to ensure that this directory name is unique. The naming of the debug directories and the configuration variables that affect the debug directories is described in [Section 5.9](#).

Simulation `printf` is also available in WebIDE; see: [Section 10.1.2](#).

Listing 3.2: Program for the moving average Kernel with printf (PrintfKernel.maxj).

```

1  /**
2   * Document: MaxCompiler Tutorial (maxcompiler-tutorial.pdf)
3   * Chapter: 5      Example: 2      Name: Printf
4   * MaxFile name: Printf
5   * Summary:
6   *      Kernel that prints debug information.
7   */
8  package printf;
9
10 import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
11 import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
12 import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
13 import com.maxeler.maxcompiler.v2.kernelcompiler.types.composite.DFEVector;
14 import com.maxeler.maxcompiler.v2.kernelcompiler.types.composite.DFEVectorType;
15
16 class PrintfKernel extends Kernel {
17
18     PrintfKernel (KernelParameters parameters) {
19         super(parameters);
20
21         DFEVectorType<DFEVar> vectorType =
22             new DFEVectorType<DFEVar>(dfeFloat(8, 24), 4);
23
24         // Input
25         DFEVar n = io.scalarInput("n", dfeUInt(8));
26         DFEVar x = io.input("x", dfeUInt(8));
27
28         // Data
29         DFEVar prevOriginal = stream.offset(x, -1);
30         DFEVar nextOriginal = stream.offset(x, 1);
31
32         // Control
33         DFEVar count = control.count.simpleCounter(8, n);
34         debug.simPrintf("Tick: %d\n", count);
35         DFEVar aboveLowerBound = count > 0;
36         DFEVar belowUpperBound = count < n-1;
37         DFEVar withinBounds = aboveLowerBound & belowUpperBound;
38         debug.simPrintf("withinBounds, "[In boundary (withinBounds = %d)]\n", withinBounds);
39
40         DFEVar prev = aboveLowerBound ? prevOriginal : 0;
41         DFEVar next = belowUpperBound ? nextOriginal : 0;
42         DFEVar string = prev.cat(x).cat(next);
43         debug.simPrintf("Prev char: \"%c\", current char: \"%c\", next char: \"%c\", string: \"%s\"\n", prev, x, next, string);
44
45         DFEVar divisor = withinBounds ? constant.var(dfeFloat(8, 24), 3) : 2;
46
47         DFEVar result = (prev.cast(dfeFloat(8, 24)) + x.cast(dfeFloat(8, 24)) + next.cast(dfeFloat(8, 24))) / divisor;
48         debug.simPrintf("Result: %.3g\n", result);
49
50         DFEVector<DFEVar> inVector = vectorType.newInstance(this);
51         inVector[0] <== prev.cast(dfeFloat(8, 24));
52         inVector[1] <== x.cast(dfeFloat(8, 24));
53         inVector[2] <== next.cast(dfeFloat(8, 24));
54         inVector[3] <== result;
55         // Double vector by multiplying all elements by a single value
56         DFEVector<DFEVar> outVector = inVector * 2;
57         debug.simPrintf("Input vector: %KObj%\nOutput vector: %KObj%\n", inVector, outVector);
58
59         // Output
60         io.output("y", result, dfeFloat(8, 24));
61     }
62 }

```

3.2 Simulation printf

3.2.1 Troubleshooting `simPrintf` does not produce any output

In some versions of MaxCompiler, some `simPrintf` commands may not produce any output. In order to solve this issue, try to run an empty SLIC (Simple Live CPU Interface; for more details please refer to "Multiscale Dataflow Programming") action after calling `max_load`:

```
max_actions_t *actions = max_actions_init(maxfile, NULL);  
max_run(engine, actions);  
max_actions_free(actions);
```

Listing 3.3: Example printf output

```

Tick: 0
    [In boundary (withinBounds = 0)]
    Prev char: " ", current char: "A", next char: "B", string: " AB"
    Result: 65.5
    Input vector: < 0.000000 :: 65.000000 :: 66.000000 :: 65.500000 >
    Output vector: < 0.000000 :: 130.000000 :: 132.000000 :: 131.000000 >
Tick: 1
    Prev char: "A", current char: "B", next char: "C", string: "ABC"
    Result: 66
    Input vector: < 65.000000 :: 66.000000 :: 67.000000 :: 66.000000 >
    Output vector: < 130.000000 :: 132.000000 :: 134.000000 :: 132.000000 >
Tick: 2
    Prev char: "B", current char: "C", next char: "D", string: "BCD"
    Result: 67
    Input vector: < 66.000000 :: 67.000000 :: 68.000000 :: 67.000000 >
    Output vector: < 132.000000 :: 134.000000 :: 136.000000 :: 134.000000 >
.
.
.
Tick: 46
    Prev char: "n", current char: "o", next char: "p", string: "nop"
    Result: 111
    Input vector: < 110.000000 :: 111.000000 :: 112.000000 :: 111.000000 >
    Output vector: < 220.000000 :: 222.000000 :: 224.000000 :: 222.000000 >
Tick: 47
    [In boundary (withinBounds = 0)]
    Prev char: "o", current char: "p", next char: " ", string: "op "
    Result: 112
    Input vector: < 111.000000 :: 112.000000 :: 0.000000 :: 111.500000 >
    Output vector: < 222.000000 :: 224.000000 :: 0.000000 :: 223.000000 >

```

4

Hardware Debugging

Once a design is verified by simulation, it should be transformed into hardware. Although simulation gives means for easier and faster testing and debugging, programmer may find it convenient to have messages getting back to him, while the application is running in hardware. For that purpose `dfePrintf` could be useful.

4.1 DFE printf

The `dfePrintf` calls behave similarly to `printf` in C.

The basic form of a `dfePrintf` prints a formatted message to the standard output of the application, with the current value of the supplied list of objects for every tick:

```
void debug.dfePrintf(String message, Object... args)
```

To print the message only when a certain condition is met, a Boolean `DFEVar` stream could be supplied as a condition:

```
void debug.dfePrintf(DFEVar condition, String message, Object... args)
```

The message is only printed when `condition` is true, e.g., when `condition` evaluates to 1.

The argument(s) `args` can contain Java variables (`Byte`, `Short`, `Integer`, `Long`, `Float`, or `Double`)

4.1 DFE printf

or DFEVar streams. The argument `message` should contain corresponding format specifiers. All familiar format specifiers from C `printf` are provided, with the only difference that `%o` prints in *binary*, rather than *octal*.

The outputs of `dfePrintf` appear on the standard output of the application and are also saved to files upon completion of a run:

- `dfePrintf` produces an output file per allocated engine. These files, available in the debug directory, are named using the scheme `debug_dfeprintf.TAG.txt`, where `TAG` is chosen to ensure that file names are unique. The outputs of `dfePrintf` are flushed upon deallocation of the engine used or, if the basic static interface is used, upon deallocation of the maxfile using the `MAXFILE_free()` function, where `MAXFILE` is the name of the maxfile;

When the application is run from within MaxIDE, this output is also visible in files that appear under the associated RunRule in the Project Explorer, upon completion of the run, in the same way as described in [Section 3.2](#) and depicted in [Figure 3.3](#). The on-disk file containing the `dfePrintf` outputs, similar to watch data, is available in the debug directory of the associated RunRule folder.

When the application is run from the command line, the output files are available in the debug subdirectory of the current directory, or in a directory of the form `debug_N`, where `N` is chosen to ensure that the selected directory name is unique. The naming of the debug directories and the configuration variables that affect the debug directories are described in [Section 5.9](#).

5

Advanced Debugging

5.1 Introduction

Debugging with **watches**, **simulation** `printf`, and **DFE** `printf` makes it possible to solve a range of problems that may appear during a design process; these methods are primarily useful for inspecting the state of Kernels.

Since Manager is a parallel and asynchronous system with many processing units, coding errors that lead to deadlocks are possible, such as insufficient data being either produced or consumed by Kernels. This section presents three further methods that help identify such problems with Manager and with the control software on the CPU:

- the graphical debugger in MaxIDE makes it possible to checkpoint the CPU code of the application and to inspect the state of the DFE interactively;
- the MaxDebug tool is a command line utility that obtains similar information on systems where MaxIDE may not be present, and allows inspection of debug snapshots that are optionally produced by SLiC applications;
- the MaxDump tool is a command line utility, similar to maxdebug in many ways. MaxDump outputs a *json* file which can be attached to a *.max* file in the IDE.

5.2 Launching the MaxIDE's debugger

Examples in this section are based on a bitstream that performs a computation of the type $s = x + y$, where x and y are input streams and s is an output stream, as seen in the following code snippet:

Listing 5.1: Simple sum Kernel (HaltedOnInputKernel.maxj).

```
15 DFEVar x = io.input("x", type);  
16 DFEVar y = io.input("y", type);  
17 DFEVar sum = x + y;  
18 io.output("s", sum, type);
```

This Kernel expects to read one word of data per tick from each input stream, and to write one word of data per tick to the output stream. The design malfunctions if the amount of data supplied or consumed by the CPU does not match the number of ticks of the Kernel. Examples below demonstrate how to diagnose such a problem in live DFEs.

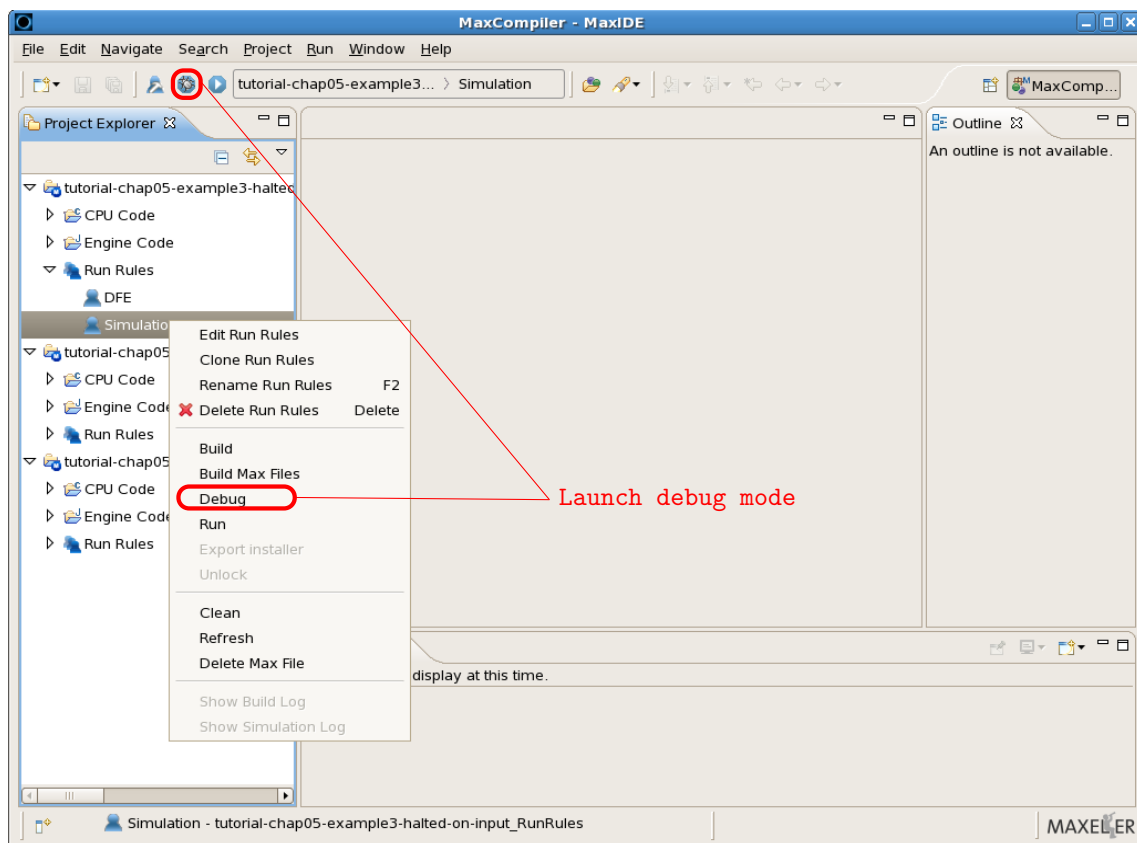


Figure 5.1: Launching a MaxCompiler project in debug mode.

5.2 Launching the MaxIDE's debugger

A MaxCompiler project could be launched in the the debug mode from within MaxIDE, by pushing the debug button in the toolbar, or by selecting the “Debug” item in the contextual menu of RunRule. This is illustrated in [Figure 5.1](#).

After the project is built, MaxIDE switches to a debugging perspective and presents various views of the program being run, as displayed in [Figure 5.2](#). This mode makes it possible to execute the program one instruction at a time, enabling us, in particular, to find locations where the program might stall. At any time after a run was launched, it is possible to obtain a snapshot of the state of the DFE, by pushing the “Get snapshot” button in the debugging toolbar; this is the button with the bug icon towards the left-hand of the toolbar. MaxIDE then shows a graphical view of the Manager graph.

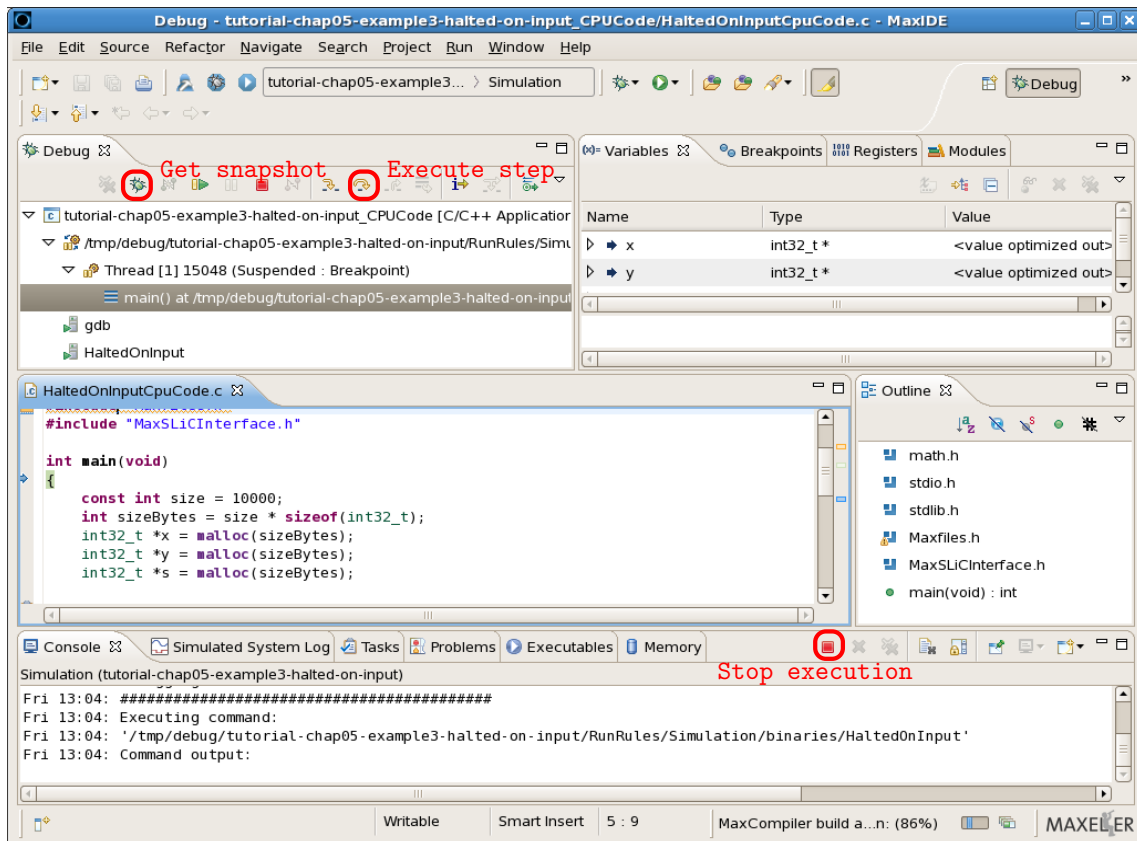


Figure 5.2: MaxIDE entering the debug mode.

5.3 Kernel halted on input

Running the DFE or Simulation RunRules for the example project `tutorial-chap5-example3`, results in applications that hang: the applications either fail with a timeout or need to be terminated manually, using the red *Terminate* button in the console window.

★ When running on a DFE, SLiC functions return an error when streaming operations last more than the prescribed time; simulation does not support such time-outs, and simulated applications with stalling designs should be terminated manually.

Launching the application in the debug mode and tracing its execution step by step allows us to see that the function `Ha1ted0nInput()` in the CPU code was called but did not return.

5.3 Kernel halted on input

At this stage, the state of Manager could be retrieved by pushing the snapshot button, producing the view shown in [Figure 5.3](#). Kernel is represented by the yellow rectangle in the middle of the graph, surrounded by nodes that connect its inputs and outputs to the CPU.

Selecting the Kernel, by clicking with the left mouse button, indicates that its status is “Halted on input”. In general, the yellow coloring of this state helps to locate problems in the data flow. The Kernel would be colored green if it had run for the required number of ticks.

Examining the CPU code, we could see that the number of ticks to run the Kernel has a value that is too large (exceeds by 5000): Kernel was running for too long and its input could not read data after size ticks.

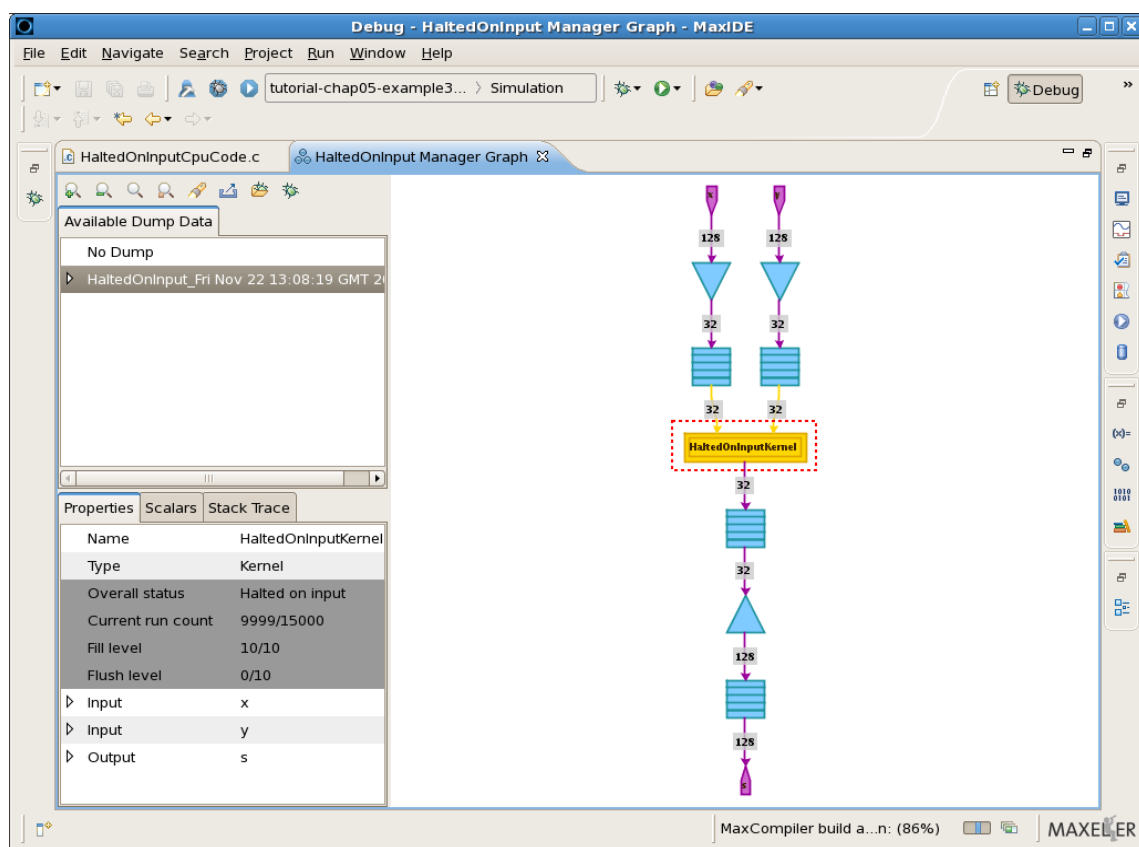


Figure 5.3: State of Manager while the function `HaltedOnInput()` is hanging: Kernel requires more data than is available.

23 `HaltedOnInput(size + 5000, x, sizeBytes, y, sizeBytes, s, sizeBytes);`



Note that the current tick count in the MaxDebug output for a Kernel may not be exactly the number expected, given the number of inputs consumed or the number of ticks that it is set to run for, as MaxCompiler may schedule extra ticks into the Kernel.

5.4 Kernel halted on output

The project `tutorial-chap5-example4` displays a problem different from the previous one: In the code listing below, the size of the output stream `s` is now too small to accommodate the data produced by Kernel, leading to a stall.

```
22   HaltedOnOutput(size, x, sizeBytes, y, sizeBytes, s, sizeBytes - 5000*sizeof(int32_t));
```

The result of this is shown graphically in [Figure 5.4](#): Kernel is marked as “Halted on output” and the output stream `s` is marked as “stalled”.

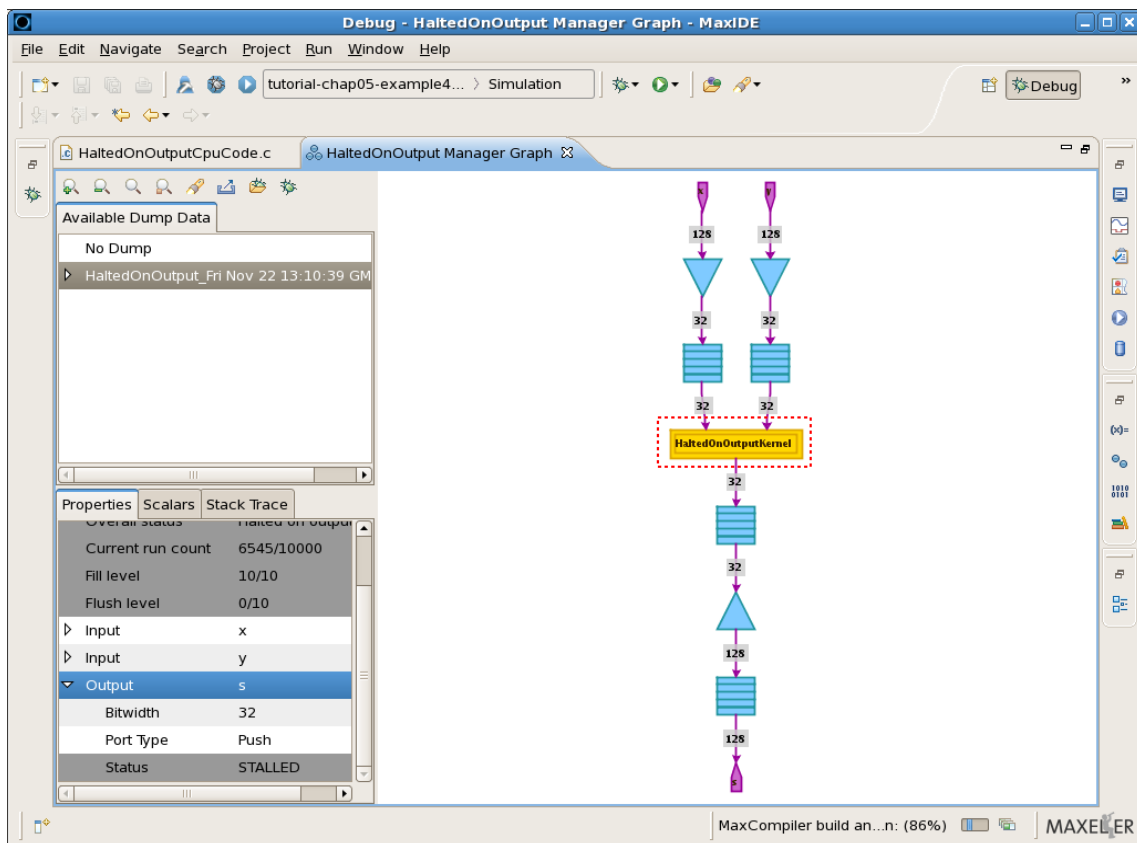


Figure 5.4: State of Manager while the function `HaltedOnOutput()` is hanging: Kernel produces more data than is consumed by the output stream `s`.

When the extra subtraction is removed, the application functions correctly; if the application is run in the debugger with a breakpoint added after the call to `HaltedOnOutput()`, then the ensuing snapshot shows a graph wherein each status is colored green.

5.5 Stream status blocks

Managers have an option to globally enable **stream status blocks**, which are additional blocks that collect information on the streams in and out of a Kernel. The status of these blocks could be read by the debugging tools, to get additional information from the DFE at run-time.

5.6 Deadlock

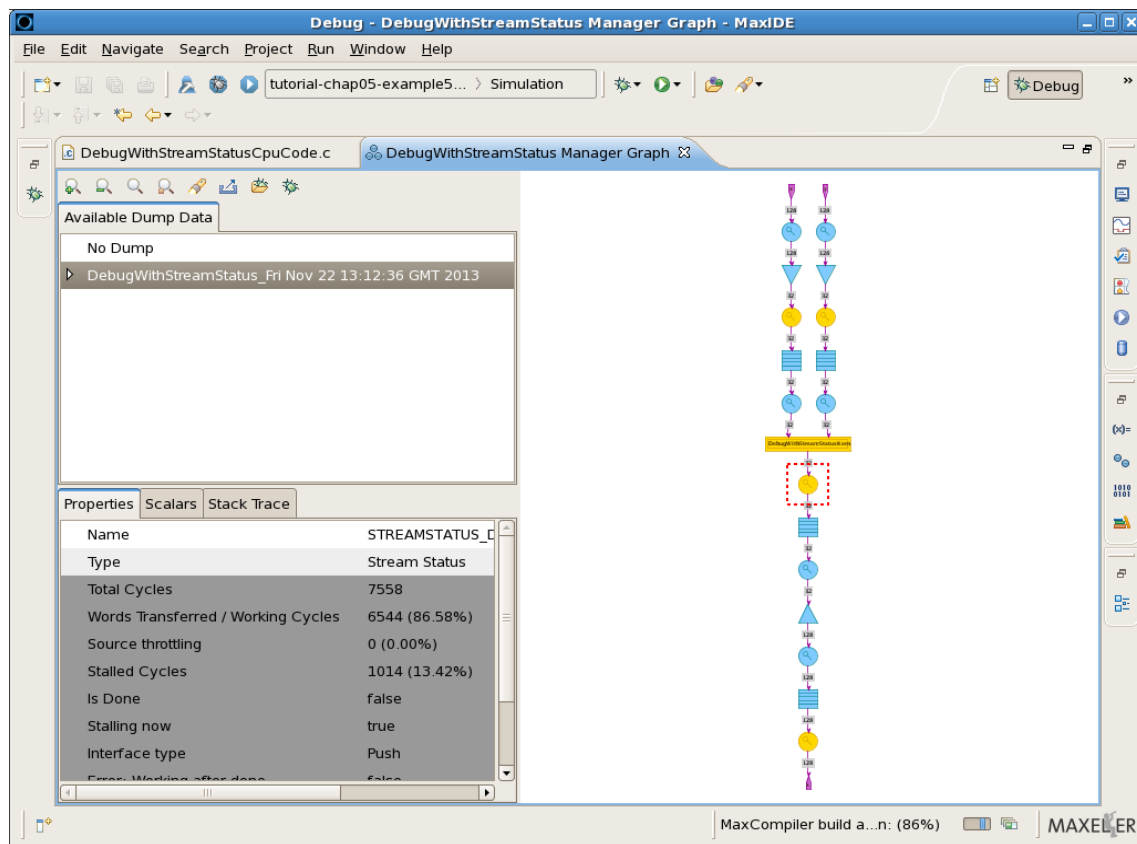


Figure 5.5: State of Manager after the function `DebugWithStatusBlocks()` has completed: **stream status blocks** indicate the streams' data throughput and the stream's overall status.

★ Manager requires rebuilding when stream status blocks are enabled, as they must be built into the `.max` file.

We could take the previous example and enable the stream status blocks using the `setEnabledStreamStatusBlocks()` method on the Standard Manager:

```
30 manager.setEnabledStreamStatusBlocks(true);
```

For a Custom Manager, stream status blocks are enabled in the `debug` property of Manager (see the "MaxCompiler Manager Compiler Tutorial" document).

The project `tutorial-chap5-example5` contains a Manager with stream status blocks enabled. Launching the application in the debug mode with a breakpoint after the call to `DebugWithStatusBlocks()`, and displaying the state of the DFE, yields the graph in Figure 5.5. Selecting a stream status block by clicking in the graph, displays a number of properties of the stream, including the amount of data transported and its overall status, along with various performance-related indicators.

5.6 Deadlock

A deadlock generally happens in two situations, as described below:

5.6.1 A Kernel has scheduled an input before an output

A Kernel has scheduled an input before an output, but something else (another Kernel, the memory controller, another DFE, etc...) requires the output before it can produce the input. The most common case is when a Kernel generates memory address commands to get data from LMem, but attempts to read from LMem before the command is sent out.

Kernel in [Listing 5.2](#) is designed to deadlock. Since the counter is used in the sum, the counter will be scheduled to output a number just after the input *x* has arrived. However, since the counter is also used in the memory command, the memory command output will be scheduled after the counter, so also after the input *x*.

Since the memory commands must leave the Kernel before the memory controller will produce any output, but the Kernel will not be producing any output due to not having any input data available, the whole system will deadlock.

To fix this bug we can force the command's output to be scheduled before the input *x* (`io.forceExactIOSeparation("x", "memCmds", -500)`). If the separation number is too small, then performance will suffer, as the memory controller takes time to produce output after receiving a command, causing the Kernel to stall. If the number is too large then the system will break due to the Manager FIFOs filling up with data and causing the design to deadlock. Choosing a high number will also consume more BRAM for buffering the counter inside the Kernel (although in this simple case you could also just replicate the counter).

Since the minimum depth of a BRAM block is 512, a separation of -500 is used. This is more than large enough for full performance, without being large enough to require extra resources.



Deadlock caused by scheduling an input before an output is generally solved with:
`io.forceExactIOSeparation("x", "memCmds", -500)`

Kernel in [Listing 5.4](#) is designed to deadlock. Further more, lets assume that there are two of these Kernels connected in the way as depicted at [Listing 5.5](#). Since *x* and *y* (*swapIn*) are added together, they will be scheduled to arrive on the same cycle. Since *swapOut* sends *x* out, then it must be scheduled after *x*, and thus after *swapIn*. With both copies of the Kernel behave in the same manner, the design will deadlock, as neither of these two Kernels will produce an output until it receives the input from the other.

To fix this bug, we can force *swapOut* to be scheduled before *swapIn*, by forcing IO separation `io.forceExactIOSeparation("swapOut", "swapIn", -50)`. If the separation number is too small, then performance will suffer, since the FIFOs between Kernels will be almost empty all of the time. Kernels will trickle waiting for more data to fill the FIFO. On the other hand, if the number is too large, then the FIFO between Kernels will not be large enough to hold all the data that is sent out before it will be read in, so the design will deadlock.

Since the depth of the FIFO is 512, a separation of -50 is used. This is large enough for full performance (any number above 10 should be fine), without being large enough to cause any problems (higher than 512 would cost more resources and would risk a deadlock).

In general, if two Kernels are running completely synchronised, then they should be merged into a single Kernel. The most common case where you really need to perform a swap is when the Kernels are running on different DFEs and the swap happens over MaxRing. In this case the separation should be larger to compensate for the latency of running over MaxRing. Making the separation equal to -500 should work at full speed without requiring extra resources.

5.6.2 A FIFO ends up full

A FIFO may end up full, causing a Kernel to stall, which starves another FIFO, causing the Kernel, that is supposed to be consuming the data from the full FIFO, to stall. A simple example is when two Kernels are connected together with two streams, each one outputting something on every cycle, where the inputs are scheduled together and the outputs are scheduled more than 512 cycles apart. Now one FIFO will be full before the other receives any data. This could be solved by using an exact IO separation, or by increasing the size of the FIFO in Manager.

Kernels in [Listing 5.6](#) and [Listing 5.7](#) are designed to deadlock. Since `prevS` is offset from `s` by 1000 cycles, the output will be scheduled 1000 cycles earlier. By default, all FIFOs in the Manager are 512 deep, so the `prevS` FIFO will be full before any output comes out of `s`. Since these are scheduled together in the subsequent Kernel, the second Kernel will be stalled waiting for `s`, while this Kernel is stalled due to not being able to output `prevS`.

There are two ways to fix this bug. The first is to force the output to come out on the same cycle using `io.forceOutputsTogether`. The second is to increase the size of the FIFO in the Manager.

In this simple case, the best option is to force the outputs together in the Kernel, as that will use no more buffering than required, and does not require any manual calculations for how much that is. In more complex examples, the outputs may be going via different Kernels, one of which may have a large latency. In this case, you will need to increase the size of the FIFOs in the Manager.

In general, it is the best to avoid these complications wherever possible. In this case, there is clearly no need for the computation to be spread across two separate Kernels. Multiple Kernels should only be used when they need to run asynchronously.

5.7 Debugging with MaxDebug

MaxDebug is a command line utility that makes it possible to inspect the state of a dataflow engine in a non-interactive fashion, while it is running or after the running is finished. MaxDebug's capabilities are as follows:

- it can obtain the status of a local hardware dataflow engine, while it is running or after the running is finished;
- it can obtain the status of a simulated dataflow engine, while it is running or after the running is finished, as long as the simulated system is running;
- it can be used to display debug snapshots optionally generated by SLiC applications at the completion of each action, both for local engines and engines provided by an MPC-X appliance.

To enable the tool to have this flexibility, some of its features are controlled by environment variables, and these must be modified according to the nature of the dataflow engine to be interrogated:

- for hardware dataflow engines, the environment variable MAXELEROSDIR must point to the MaxelerOS installation, and the library \$MAXELEROSDIR/lib/libmaxeleros.so must be preloaded by MaxDebug:

```
export LD_PRELOAD=$MAXELEROSDIR/lib/libmaxeleros.so:$LD_PRELOAD
```

- for simulation, the environment variable MAXCOMPILERDIR must point to the simulation libraries in the MaxCompiler installation, and the environment must be set as:

```
export MAXELEROSDIR=$MAXCOMPILERDIR/lib/maxeleros-sim
export LD_PRELOAD=$MAXELEROSDIR/lib/libmaxeleros.so:$LD_PRELOAD
```

Further, in the case of simulation, the name of the simulated engine must be specified to MaxDebug by way of the `-d` option. This name is based on the socket name of the simulated system, which is available in the Simulator tab of the Run Rule editor of the corresponding project.

Listing 5.8: The MaxDebug command-line options

```

1 $ maxdebug
2 MaxDebug version 45f1310
3
4 Usage:
5     maxdebug [-v] [-g <prefix>] [-s <prefix>] [-n] [-r]
6             [-k <kernel>] [-L|-a <name>|-i <id>] [-R <mpcx>]
7             [-d <device>] [<maxfile>]
8 where:
9     -r          dump scalar inputs / mapped register values
10    -m          dump mapped memories
11    -g          draw a manager graph annotated with runtime information
12                as a .png file for every device
13    -k <kernel> limit output to the given kernel
14    -x          print all numbers in hexadecimal format
15    -v          more verbose output
16    -s          draw the static manager graph from the .max file
17    -d <device> use <device> when debugging the design (e.g sim0:sim,
18                /dev/maxeler0, or index if used with -a, -i or -f)
19    -f <file>   file containing the maxdebug snapshot to use
20    -a <action> name of the debug snapshot to retrieve from an MPC-X
21                appliance
22    -i <id>     id of the debug snapshot to retrieve from an MPC-X
23                appliance
24    -L          retrieve a list of maxdebug snapshots held by an MPC-X
25                appliance
26    -R <mpcx>   use <mpcx> as the IP address or hostname of an MPC-X
27                appliance
28    <maxfile>   bitstream .max file (mandatory, unless -L is present)

```

Kernel halted on input on a simulated dataflow engine Running the Simulation Run Rules in the example project `tutorial-chap5-example3`, as above, produces applications that hang. To investigate this problem using MaxDebug, we open a terminal, change directory to the root of the MaxCompiler project, and set the environment for simulation:

```
$ export MAXELEROSDIR=$MAXCOMPILERDIR/lib/maxeleros-sim/  
$ export LD_PRELOAD=$MAXELEROSDIR/lib/libmaxeleros.so:$LD_PRELOAD
```

We can then launch MaxDebug:

```
$ maxdebug -d jHaltedOnInput0:jHaltedOnInput -g graph RunRules/Simulation/  
maxfiles/HaltedOnInput.max
```

Here, the arguments supplied to MaxDebug are:

- `-d jHaltedOnInput0:jHaltedOnInput` specifies the name of the engine, which is constructed as `<engine-name>:<socket-name>`, based on the index of the engine in the simulated system and the socket name of the simulated system.
In this instance, the index of the engine is “0” (there being only one engine), and the socket name of the simulated system is “jHaltedOnInput”.
For a hardware dataflow engine, this argument is not required.
- `-g graph` instructs MaxDebug to output an image of the state of the design. The generated file will have “graph” as a prefix;
- `RunRules/Simulation/maxfiles/HaltedOnInput.max` is the path to the `.max` file.

The console output of this command is shown in [Listing 5.9](#) and the `.png` file depicted in [Figure 5.6](#) is produced.

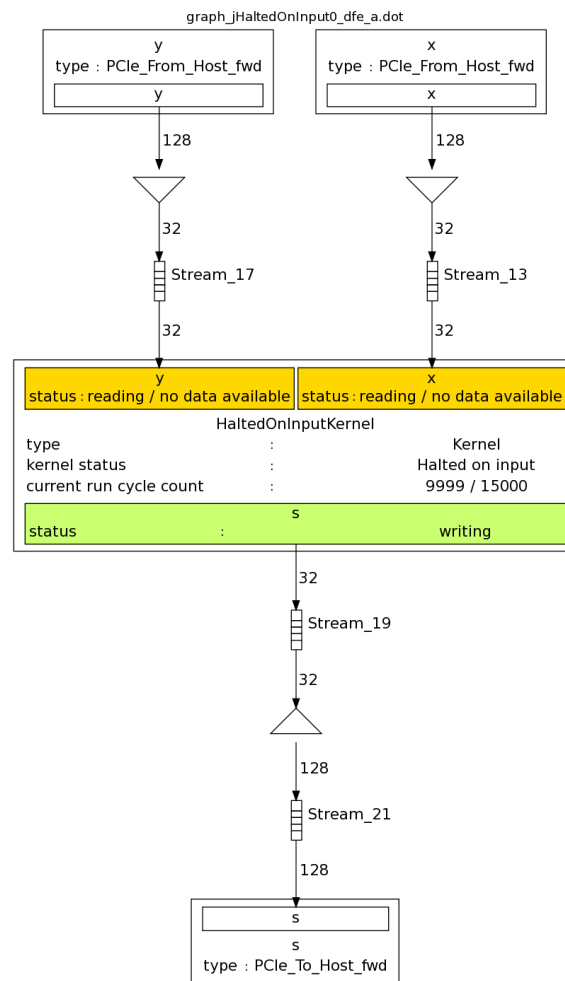


Figure 5.6: State of Manager while the function `HaltedOnInput()` is hanging, as represented by MaxDebug.

Listing 5.2: Deadlock Kernel - Missing data from memory (DeadlockKernel.maxj).

```

1  /**
2   * Document: MaxDebug Tutorial (maxdebug-tutorial.pdf)
3   * Chapter: 5      Example: 1      Name: Deadlock
4   * MaxFile name: Deadlock
5   * Summary:
6   *      The following kernel is designed to deadlock. Since the counter is used in the sum, the counter
7   *      will be scheduled to output a number just after the input x has arrived. However, since
8   *      the counter is also used in the memory command, the memory command output will be scheduled
9   *      after the counter, so also after the x input.
10  */
11  package deadlock;
12
13  import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
14  import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
15  import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.LMemCommandStream;
16  import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.CounterChain;
17  import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEType;
18  import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
19  import com.maxeler.maxcompiler.v2.kernelcompiler.types.composite.DFEStruct;
20
21  class DeadlockKernel extends Kernel {
22
23      private static final DFEType type = dfeInt(32);
24
25      protected DeadlockKernel(KernelParameters parameters) {
26          super(parameters);
27          //to fix kernel, uncomment the next line
28          // io.forceExactIOSeparation("x", "memCmds", -500);
29
30          CounterChain chain = control.count.makeCounterChain();
31          DFEVar burstNumber = chain.addCounter(1 < 27, 1);
32          DFEVar positionInB = chain.addCounter(3072 / type.getTotalBits(), 1);
33
34          DFEStruct cmd = LMemCommandStream.getLMemCommandDFEStructType().newInstance(this);
35          cmd["address"] = burstNumber.cast((DFEType)cmd["address"].getType());
36          cmd["size"] = constant.var((DFEType)cmd["size"].getType(), 128);
37          cmd["inc"] = constant.var((DFEType)cmd["inc"].getType(), 1);
38          cmd["stream"] = constant.var((DFEType)cmd["stream"].getType(), 1);
39          cmd["tag"] = constant.var((DFEType)cmd["tag"].getType(), 0); //not generating interrupt, as we don't need to check for
40                          it (reading, not writing).
41          LMemCommandStream.makeKernelOutput("memCmds", positionInB == 0, cmd);
42
43          DFEVar x = io.input("x", type);
44          DFEVar y = io.input("y", type);
45          DFEVar a = io.scalarInput("a", type);
46
47          DFEVar sum = x + y + a + burstNumber.cast(type);
48
49          io.output("s", sum, type);
50      }
51  }

```

Listing 5.3: Deadlock Manager - Missing data from memory (DeadlockManager.maxj).

```

1 package deadlock;
2
3 import com.maxeler.maxcompiler.v2.managers.BuildConfig;
4 import com.maxeler.maxcompiler.v2.managers.custom.CustomManager;
5 import com.maxeler.maxcompiler.v2.managers.custom.DFELink;
6 import com.maxeler.maxcompiler.v2.managers.custom.blocks.KernelBlock;
7 import com.maxeler.maxcompiler.v2.managers.custom.stdlib.DebugLevel;
8 import com.maxeler.maxcompiler.v2.managers.custom.stdlib.MemoryControlGroup.MemoryAccessPattern;
9 import com.maxeler.maxcompiler.v2.managers.engine_interfaces.CPUTypes;
10 import com.maxeler.maxcompiler.v2.managers.engine_interfaces.EngineInterface;
11 import com.maxeler.maxcompiler.v2.managers.engine_interfaces.EngineInterface.Direction;
12 import com.maxeler.maxcompiler.v2.managers.engine_interfaces.InterfaceParam;
13
14 public class DeadlockManager extends CustomManager {
15
16     private static final String s_kernelName = "DeadlockKernel";
17
18     public DeadlockManager(DeadlockEngineParameters engineParameters) {
19         super(engineParameters);
20
21         KernelBlock block = addKernel(new DeadlockKernel(makeKernelParameters(s_kernelName)));
22
23         DFELink cpu2lmem = addStreamToOnCardMemory("cpu2lmem", MemoryAccessPattern.LINEAR_1D);
24         DFELink lmem2cpu = addStreamFromOnCardMemory("lmem2cpu", MemoryAccessPattern.LINEAR_1D);
25
26         cpu2lmem <== addStreamFromCPU("cpu_to_lmem");
27         addStreamToCPU("tocpu") <== lmem2cpu;
28
29         block.getInput("x") <== addStreamFromOnCardMemory("x", block.getOutput("memCmds"));
30         block.getInput("y") <== addStreamFromCPU("y");
31         addStreamToCPU("s") <== block.getOutput("s");
32
33         createSLICInterface(modeDefault());
34         createSLICInterface(modeWrite("writeLMem"));
35
36         debug.setDebugLevel(new DebugLevel() { @Override public boolean hasStreamStatus() { return true; } });
37
38         configBuild(engineParameters);
39     }

```

Listing 5.4: Deadlock Kernel - Mutually connected Kernels (Deadlock2Kernel.maxj).

```

1  /**
2   * Document: MaxDebug Tutorial (maxdebug-tutorial.pdf)
3   * Chapter: 5      Example: 2      Name: Deadlock
4   * MaxFile name: Deadlock
5   * Summary:
6   *   This kernel is designed to deadlock. Since x and y (swapIn) are added together, they will be
7   *   scheduled to arrive on the same cycle. Since swapOut sends x out, then it must be scheduled
8   *   after x, and thus after swapIn. With both copies of the kernel doing the same thing, the
9   *   design will deadlock as neither will output until it receives the others output.
10  */
11  package deadlock2;
12
13  import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
14  import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
15  import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEType;
16  import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
17
18  class Deadlock2Kernel extends Kernel {
19
20      private static final DFEType type = dfeInt(32);
21
22      protected Deadlock2Kernel(KernelParameters parameters) {
23          super(parameters);
24          // to fix kernel, uncomment the next line
25          // io.forceExactIOSeparation("swapIn", "swapOut", -50);
26
27          DFEVar x = io.input("x", type);
28          io.output("swapOut", x, type);
29          DFEVar y = io.input("swapIn", type);
30          DFEVar a = io.scalarInput("a", type);
31
32          DFEVar sum = x + y + a;
33
34          io.output("s", sum, type);
35      }
36  }
37

```

Listing 5.5: Deadlock Manager - Mutually connected Kernels (Deadlock2Manager.maxj).

```

1 package deadlock2;
2
3 import com.maxeler.maxcompiler.v2.managers.BuildConfig;
4 import com.maxeler.maxcompiler.v2.managers.custom.CustomManager;
5 import com.maxeler.maxcompiler.v2.managers.custom.blocks.KernelBlock;
6 import com.maxeler.maxcompiler.v2.managers.engine_interfaces.CPUTypes;
7 import com.maxeler.maxcompiler.v2.managers.engine_interfaces.EngineInterface;
8 import com.maxeler.maxcompiler.v2.managers.engine_interfaces.InterfaceParam;
9
10 public class Deadlock2Manager extends CustomManager {
11
12     private static final String s_kernelName = "Deadlock2Kernel";
13
14     public Deadlock2Manager(Deadlock2EngineParameters engineParameters) {
15         super(engineParameters);
16
17         KernelBlock block1 = addKernel(new Deadlock2Kernel(makeKernelParameters(s_kernelName+1)));
18         KernelBlock block2 = addKernel(new Deadlock2Kernel(makeKernelParameters(s_kernelName+2)));
19
20         block1.getInput("x") <== addStreamFromCPU("x");
21         block2.getInput("x") <== addStreamFromCPU("y");
22
23         block1.getInput("swapIn") <== block2.getOutput("swapOut");
24         block2.getInput("swapIn") <== block1.getOutput("swapOut");
25
26         addStreamToCPU("s1") <== block1.getOutput("s");
27         addStreamToCPU("s2") <== block2.getOutput("s");
28
29         createSLCInterface(modeDefault());
30
31         configBuild(engineParameters);
32     }
33
34     private static EngineInterface modeDefault() {
35         EngineInterface engine_interface = new EngineInterface();
36         CPUTypes type = CPUTypes.INT32;
37         int size = type.sizeInBytes();
38
39         InterfaceParam a = engine_interface.addParam("A", CPUTypes.INT);
40         InterfaceParam N = engine_interface.addParam("N", CPUTypes.INT);
41
42         engine_interface.setScalar(s_kernelName+1, "a", a);
43         engine_interface.setScalar(s_kernelName+2, "a", a);
44
45         engine_interface.setTicks(s_kernelName+1, N);
46         engine_interface.setTicks(s_kernelName+2, N);
47         engine_interface.setStream("x", type, N * size);
48         engine_interface.setStream("y", type, N * size);
49         engine_interface.setStream("s1", type, N * size);
50         engine_interface.setStream("s2", type, N * size);
51         return engine_interface;
52     }
53
54     private void configBuild(Deadlock2EngineParameters params) {
55         BuildConfig buildConfig = getBuildConfig();
56         buildConfig.setMPPRCostTableSearchRange(params.getMPPRStartCT(), params.getMPPREndCT());
57         buildConfig.setMPPRParallelism(params.getMPPRThreads());
58         buildConfig.setMPPRRetryNearMissesThreshold(params.getMPPRRetryThreshold());
59     }
60
61     public static void main(String[] args) {
62         Deadlock2Manager manager = new Deadlock2Manager(new Deadlock2EngineParameters(args));
63         manager.build();
64     }
65 }

```

Listing 5.6: Deadlock Kernel - A FIFO ends up full (Deadlock3Kernel1.maxj).

```

1  /**
2   * Document: MaxDebug Tutorial (maxdebug-tutorial.pdf)
3   * Chapter: 5      Example: 6      Name: Deadlock
4   * MaxFile name: Deadlock
5   * Summary:
6   *   This kernel is designed to deadlock. Since prevS is offset from s by 1000 cycles, the output
7   *   will be scheduled ~1000 cycles earlier. By default, all FIFOs in the manager are 512 deep, so
8   *   the prevS FIFO will be full before any output comes out of s. Since these are scheduled
9   *   together in the subsequent kernel, the second kernel will be stalled waiting for s, while
10  *   this kernel is stalled due to not being able to output prevS.
11  */
12  package deadlock3;
13
14  import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
15  import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
16  import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEType;
17  import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
18
19  class Deadlock3Kernel1 extends Kernel {
20
21      private static final DFEType type = dfeInt(32);
22
23      protected Deadlock3Kernel1(KernelParameters parameters) {
24          super(parameters);
25
26          DFEVar x = io.input("x", type);
27          DFEVar y = io.input("y", type);
28          DFEVar a = io.scalarInput("a", type);
29
30          DFEVar sum = x + y + a;
31
32          io.output("s", sum, type);
33
34          DFEVar cycle = control.count.simpleCounter(48);
35          DFEVar prevSum = cycle < 1000 ? 0 : stream.offset(sum, -1000);
36          io.output("prevS", prevSum, prevSum.getType());
37
38          // To fix this design, either uncomment the following line, or a line in the manager
39          // io.forceOutputsTogether("s", "prevS");
40      }
41  }

```

Listing 5.7: Deadlock Kernel - A FIFO ends up full (Deadlock3Kernel1.maxj).

```

1 package deadlock3;
2
3 import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
4 import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
5 import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEType;
6 import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
7
8 class Deadlock3Kernel2 extends Kernel {
9
10     private static final DFEType type = dfelnt(32);
11
12     protected Deadlock3Kernel2(KernelParameters parameters) {
13         super(parameters);
14
15         DFEVar x = io.input("x", type);
16         DFEVar y = io.input("y", type);
17
18         DFEVar sum = x + y;
19
20         io.output("s", sum, type);
21     }
22 }
23

```

Listing 5.9: MaxDebug console output

```

1 MaxDebug version 45f1310
2
3 =====
4 Kernel : HaltedOnInputKernel
5 =====
6
7 Kernel summary
8 -----
9 Name           : HaltedOnInputKernel
10 Fill level      : 10 / 10
11 Flush level     : 0 / 10
12 Flushing        : False
13 Ran for         : 9999 / 15000 cycles
14 Derived status  : Halted on input
15
16
17 Stream summary
18 -----
19 Name  Id    Type      #Outstanding Reads    Derived Status
20 ----  --    ----      -
21 x     0     input      1                      reading / no data available
22 y     1     input      1                      reading / no data available
23 s     0     output      0                      writing

```

Saving MaxDebug snapshots for later use In the sections above, the information provided by MaxDebug was taken directly from an engine at runtime. In certain cases, we may want to save a snapshot of the content of the DFE for later use. To do this, we instruct SLiC to save a snapshot automatically on completion of an action, by setting `default_maxdebug_mode` in the `SLIC_CONF` environment variable:

```
$ env SLIC_CONF="default_maxdebug_mode=MAX_DEBUG_ALWAYS" ./HaltedOnInput
```

In the SLiC's dynamic interface, the same result could be achieved on a per-action basis:

```
void max_set_debug(
    max_actions_t *actions,
    const char *name,
    max_debug_mode_t debug_mode);
```

where the enum `debug_mode` must be one of:

- `MAX_DEBUG_NEVER`: where no debug snapshot should be saved for the action;
- `MAX_DEBUG_ON_ERROR`: where a debug snapshot should only be saved in the case of stall;
- `MAX_DEBUG_ALWAYS`: where SLiC should always produce a debug snapshot;

and where `name` is used to identify the snapshot.

In SLiC's basic and advanced static interfaces, the `maxfile` stem is used as the name of the snapshot.

On completion of the actions, the debug snapshots are saved in the default debug directory with file names of the form "`maxdebug_NAME.TIMETAG`", where `NAME` is the parameter specified in the call to `max_set_debug` or the `MaxFile` stem name, and where `TIMETAG` is a timestamp used to make the file unique. These snapshots may be examined by the `maxdebug` command, using an additional `-f <file>` flag to specify the snapshot file to use, for example:

```
$ maxdebug -r -f <snapshot> <maxfile>
```

Where arrays of engines are used, the content of a specific engine may be obtained by using the `-d <index>` option, where the index ranges between 0 and $N - 1$ for an array of N devices.

5.8 Debugging with MaxDump

MaxDump is a command line utility that makes it possible to inspect the state of a dataflow engine in a non-interactive fashion, while it is running or after the running is finished. It outputs a `json` file that could be attached to a `.max` file in IDE. MaxDump's capabilities are as follows:

- it can obtain the status of a local hardware dataflow engine, while it is running or after the running is finished;
- it can obtain the status of a simulated dataflow engine, while it is running or after the running is finished, as long as the simulated system is running;
- it can be used to display debug snapshots optionally generated by SLiC applications at the completion of each action, both for local engines and engines provided by an MPC-X appliance.

To enable the tool to have this flexibility, some of its features are controlled by environment variables, and these must be modified according to the nature of the dataflow engine to be interrogated:

- for hardware dataflow engines, the environment variable `MAXELEROSDIR` must point to the MaxelerOS installation, and the library `$MAXELEROSDIR/lib/libmaxeleros.so` must be preloaded by Max-Dump:

```
export LD_PRELOAD=$MAXELEROSDIR/lib/libmaxeleros.so:$LD_PRELOAD
```

- for simulation, the environment variable `MAXCOMPILERDIR` must point to the simulation libraries in the MaxCompiler installation, and the environment must be set as:

```
export MAXELEROSDIR=$MAXCOMPILERDIR/lib/maxeleros-sim
export LD_PRELOAD=$MAXELEROSDIR/lib/libmaxeleros.so:$LD_PRELOAD
```

Further, in the case of simulation, the name of the simulated engine must be specified to Max-Dump using the `-d` option. This name is based on the socket name of the simulated system, which is available in the `Simulator` tab of the Run Rule editor of the corresponding project.

Listing 5.10: The MaxDump command-line options

```

1 $ maxdump -h
2 maxdump 2015.1
3
4 Utility to dump state of a DFE.
5
6 Usage: maxdump [OPTIONS]...
7
8 -h, --help          Print help and exit
9 -V, --version       Print version and exit
10 -m, --maxfile=STRING Maxfile to dump data for.
11 -d, --daemon=STRING Daemon to connect to (simulation only).
12 -r, --remote=STRING IP address of MPC-X node
13 -a, --memall        Include all mapped memories. (default=off)
14 -e, --memexc=STRING Include all mapped memories except this
15                    comma-separated list.
16 -i, --meminc=STRING Dump only this comma-separated list of
17                    mapped memories.
18 -o, --outfile=STRING Output file. Use '-' for stdout.
19                    (default='maxdump.json')
20 -n, --device_num=INT Device number to dump data for. Default all devices
21                    dumped.
```

5.9 Debug directories

A number of debug output files could be produced, but two cases are most relevant:

5.9 Debug directories

Option string	Type	Default	Description
<code>default_pcie_timeout</code>	integer	30	Default timeout for PCIe stream transfer (seconds)
<code>default_wfi_timeout</code>	integer	30	Default timeout for interrupt wait (seconds)
<code>default_topology_timeout</code>	integer	-1 (infinite)	Default timeout for topology allocation (seconds)
<code>default_maxdebug_mode</code>	enum	never	Default debug mode
<code>verbose</code>	Boolean	false	Enable full debug output
<code>eventlog_ignore_errors</code>	Boolean	false	Ignore errors in the event logging module
<code>eventlog_enable</code>	Boolean	false	Enable event monitoring, even if it is not enabled in the CPU code
<code>dfeprintf_enable</code>	Boolean	true	Enable dfePrintf output
<code>find_next_debug_dir</code>	Boolean	true	Change the debug directory name if the current one already exists
<code>printf_to_stdout</code>	Boolean	true	Stream Debug.printf to standard output
<code>default_engine_resource</code>	string	NULL	Default location of the engines
<code>use_simulation</code>	string	NULL	Simulation server socket
<code>default_eventlog_server</code>	string	NULL	Name of the event logging server
<code>default_eventlog_process_name</code>	string	NULL	Event logging process name
<code>debug_dir</code>	string	debug	Directory where debug output is written

Table 5.1: The SLiC configuration options

- when the application is run from within MaxIDE, these output files are generated in a time-stamped directory present under debug in the corresponding RunRule folder, e.g., `RunRules/Simulation/debug/2013.07.03-14.16.30-380-BST`;
- when the application is run from the command line, these outputs are generated in a directory below the current directory. By default, the name of this debug directory is debug if no preexisting directory with that name is present; otherwise, a suffix is added to the debug keyword, yielding debug_1, debug_2, etc. This behavior could be modified using the `debug_dir` and `find_next_debug_dir` configuration keywords presented in [Table 5.1](#).

6

Common Errors

The following section introduces the reader to common errors that may occur in Compile time, both for simulation and hardware, and in Runtime.

6.1 Simulation Compilation Errors

In this section we look at some common problems and solutions when compiling designs for simulation.

6.1.1 Failed to create topology pool in daemon: topology request has timed out

Listing 6.1: Failed to create topology pool in daemon

```
1 Wed 16:40: SLiC Error #620 @ topology_internal.c:1834 -  
2           Failed to create topology pool in daemon:  
3           the topology request has timed out.  
4 Wed 16:40: (Requested topology = D:0::24GB,5SGXMABN2F45C2)  
5 Wed 16:40: SLiC Error #506 @ singlet_internal.c:154 -  
6           Unable to initialise topology pool  
7 Wed 16:40: SLiC Error #526 @ singlet_internal.c:266 -
```

6.2 Hardware Compilation Errors

```
8         Unable to allocate engine
9 Wed 16:40: SLiC Error #518 @ engine.c:26 -
10         Error reported from function "max_load".
11 Wed 16:40: Process terminated with exit code -1.
12 Wed 16:40: Aborted
```

There are many reasons for this error to occur. Be sure to first check the simulation daemon log for any clues.

Common mistakes:

- you are either directly or indirectly calling `max_load` many times, without first unloading, which will cause this error, as there is only one DFE available in simulation;
- you are using basic static; it is important to keep in mind that each generated interface function will call `max_load`. If you wish to run multiple actions on an already loaded DFE, then have a look at the advanced SLiC interfaces.

6.2 Hardware Compilation Errors

The following chapters are about compilation errors, helping how to deal with them.

6.2.1 Cost table failed

The process of fitting a design into hardware can be a challenging task. Design complexity, hardware resource, and desired frequency are just a few of the variables that effect the Place and Route process. Having all these variable factors along with other constraints results in a non deterministic approach to synthesizing a design and due to this multiple heuristics must be employed. Each of these heuristics used in the Place and Route phase of the build process are configured with starting parameters defined in a **cost table**. It is a good idea, especially when building complicated designs, to build with multiple cost tables as no cost table is guaranteed to build a design which meets timing and different cost tables will often result in different timing scores. The programmer may specify how many cost tables to build with in the DFE RunRule definition within MaxIDE - as shown at [Figure 6.1](#).

Note: At the end, the cost table might not be able to give a solution, [Listing 8.1.1](#).

Listing 6.2: Cost tables failed to meet timing

```
1 Wed 14:41: (7/16) - Place and Route DFE (AlteraMPPR)
2 Wed 14:41: Executing MPPR with 4 cost tables and 1 thread.
3 Wed 14:41: MPPR: Starting 4 cost tables
4 Wed 14:54: MPPR: Cost table 1 failed timing with score 4361
5         (best score 4361)
6 Wed 15:07: MPPR: Cost table 2 failed timing with score 13021
7         (best score 4361)
8 Wed 15:20: MPPR: Cost table 3 failed timing with score 274 (best score 274)
9 Wed 15:35: MPPR: Cost table 4 failed timing with score 3421 (best score 274)
```

Two possible solutions for this problem:

1. Change cost tables that are going to be used in the build process; see [Figure 6.1](#).

2. Change clock frequency:

```
1 manager.setClockFrequency(desiredFrequency);
```

6.2.2 Using more resources than available

With large and complex designs it is possible to exhaust all hardware resources on a DFE. You may try some of the techniques demonstrated in [Section 9](#), in order to fit your entire design onto a single DFE, or you may consider splitting your design over multiple DFEs.

6.3 Runtime Errors

Some errors may appear after the build process is successfully completed - in runtime. Those errors and methods of dealing with them are presented in this chapter.

6.3.1 Stream alignment issues

This error occurs on the host when data is transferred to or from the DFE. Since stream data is transferred in 16 byte chunks the length of a stream must be a multiple of 16 bytes. The simplest solution is to provide a properly sized amount of data. More complex solutions such as padding the stream and removing the extra data in the design are also possible.

Listing 6.3: SLiC Error caused by stream alignment

```
1 Wed 13:27: SLiC Error #280 @ pcie_streams_internal.c:67 -  
2     Stream alignment error for stream "x": the length (28 bytes)  
3     of the stream must be a multiple of 16 bytes.  
4 Wed 13:27: SLiC Error #518 @ stream.c:21 - Error reported from function  
5     "max_queue_input".  
6 Wed 13:27: Process terminated with exit code -1.  
7 Wed 13:27: Aborted
```

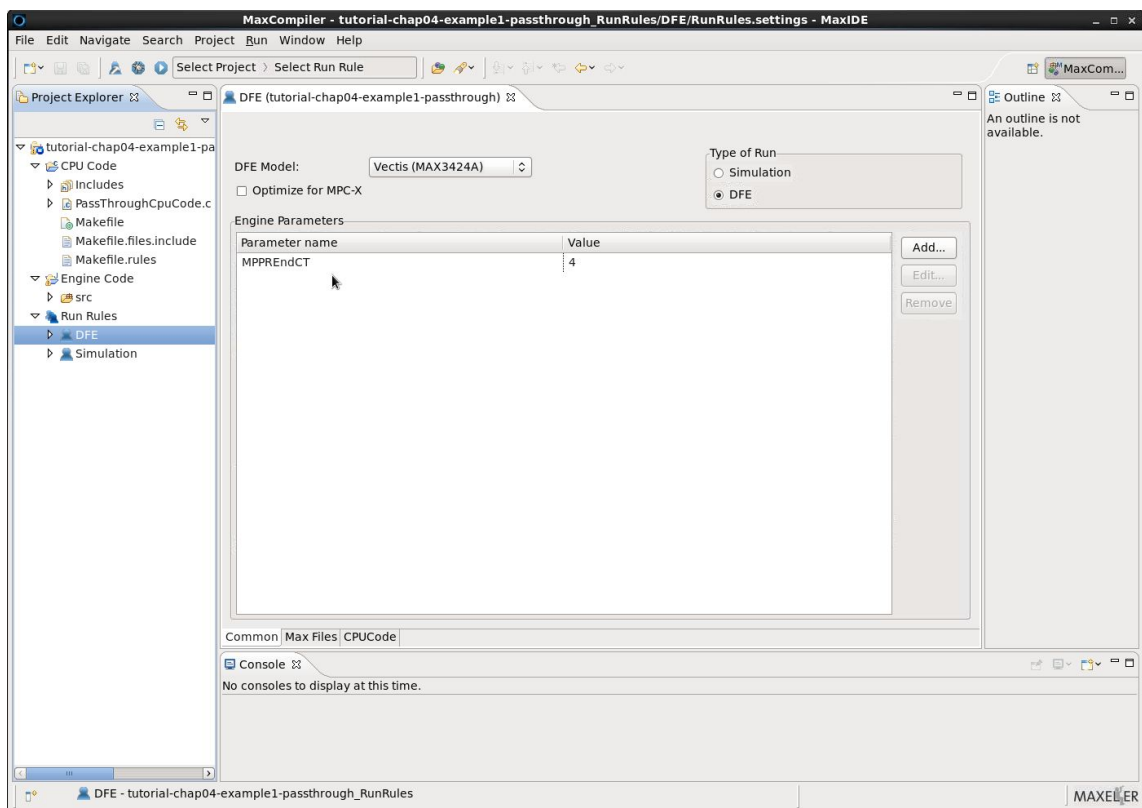


Figure 6.1: MaxIDE: Set number of the last cost table to be used

7

Best DFE Optimization Practices

This chapter introduces a number of best practices useful to know. Some common problems and appropriate solutions are presented.

7.1 FramedKernel with regular or non-blocking inputs and outputs

The `FramedKernel` does a lot behind the scene, and in general it is not a good idea to use non-frame inputs and outputs. One possible problem with simple swapping of a frame input stream with an input stream is data loss. As an alternative, it is recommended to construct each frame manually and to use non-blocking inputs.

7.2 Uninitialized data

Many variables are initialized to zero in simulation, although it is known that their values may not be zero in DFE hardware. For example, the values returned by negative stream offsets and the contents of FMem (BRAM) and LMem (DDR).

You can always load the idle bitstream by running `maxforceidle` on the terminal. This will ensure that the negative stream offsets and FMem are cleared to zero. Note that this does not affect the contents of LMem. If running `maxforceidle` immediately before running your application fixes the output, it is likely that your design is using uninitialized values.

7.3 Enable/Disable latency annotation

A stream offset of $-n$ is undefined for the first n cycles. You can avoid using the undefined values as follows:

```
DFEVar offsetVal = (cycleCount < n) ? initialValue : stream.offset(val, -n);
```

If it is possible that you are using a value read from either FMem or LMem before it was written to, these values also must be initialized. The contents of an FMem could be initialized by mapping it to CPU and initializing it via SLiC. The contents of LMem could also be initialized using SLiC.

7.3 Enable/Disable latency annotation

MaxCompiler offers latency annotated code as a way of helping programmers to better understand latency of specific data paths in his design. An example of latency annotated code is given in [Figure 7.1](#)

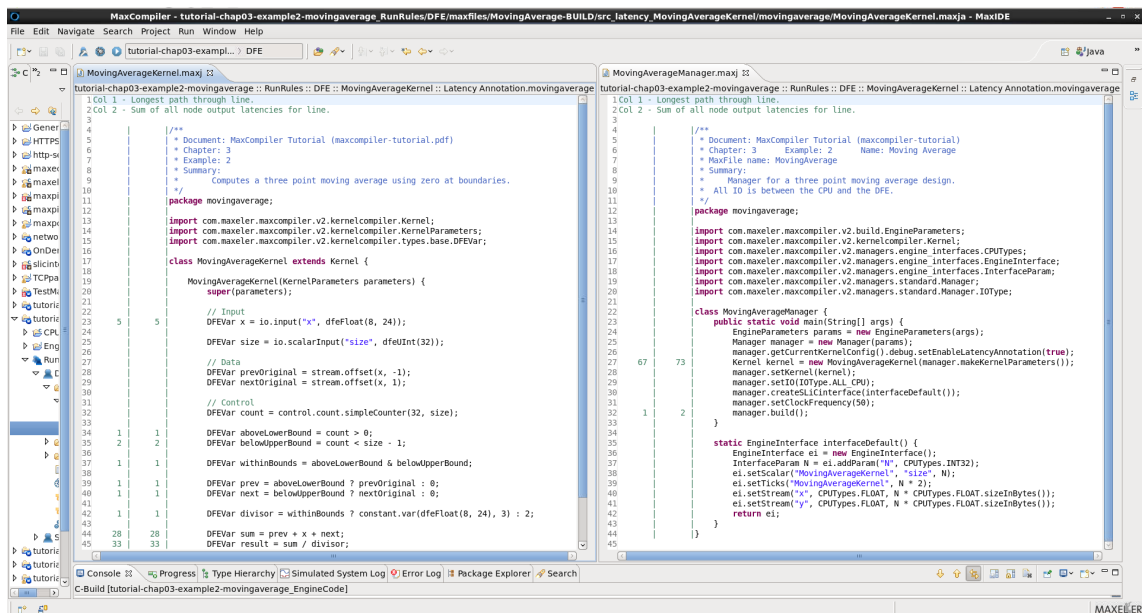


Figure 7.1: Latency annotated code for Kernel and Manager

By default, this option is disabled. To enable it:

```
Manager manager = ...  
manager.getCurrentKernelConfig().debug.setEnableLatencyAnnotation(true);  
manager.setKernel(...);
```

8

Selected DFE Design Optimizations

Optimizing a DFE design can be done in both time and space. Often times there are trade offs between the two. In this section techniques to increase the rate of data processing, and to reduce a design's on chip resources will be discussed.

8.1 Kernel optimizations

The Kernel offers various forms of optimizations:

8.1.1 Holistic optimization

`KernelConfiguration.optimization:`

```
1 Manager manager = new Manager(params);  
2 KernelConfiguration kernelConfiguration = m.getCurrentKernelConfig();
```

`setCEPipelining(int pipelining):` Sets the number of pipelining stages to add to the clock enable signal. This allows the enable signal to travel a shorter distance during each cycle. When you have a timing problems with the clock enable signal, increase the number of pipelining stages. Default value is 2. For pipelining choose a value greater of equal to 2.

8.1 Kernel optimizations

```
1 kernelConfiguration.optimization.setCEPipelining(2);
2 Kernel kernel = new MyKernel( manager.makeKernelParameters() );
```

`setCEReplicationNumPartitions(int log2NumPartitions)`: Sets the number of partitions for the clock enable signal. Where and when appropriate, increase the number of partitions to reduce the fan-out from the clock enable signal. Default value is 0 (i.e., $2^0 = 1$ partition).

```
1 kernelConfiguration.optimization.setCEReplicationNumPartitions(0);
2 Kernel kernel = new MyKernel( manager.makeKernelParameters() );
```

`setUseGlobalClockBuffer(boolean enabled)`: Sets a dedicated, global clock buffer for a Kernel. This can improve the timing for Kernels, but there is a limited number of global clock buffers available. A good practice is to use clock buffering for larger Kernels in a multi-Kernel design. The default value is disabled.

```
1 kernelConfiguration.optimization.setUseGlobalClockBuffer(true);
2 Kernel kernel = new MyKernel( manager.makeKernelParameters() );
```

8.1.2 Push-pop optimizations

`optimization.push<OpName>`:

`pushPipeliningFactor(double pipelining)`: Controls the pipeline depth of all operations in a code segment. Increasing the depth of an operation allows the data and control signals to travel a shorter distance per cycle but spreads the operation out over more cycles. Parameter range is between 0.0 and 1.0, where 0.0 creates unpipelined operations and 1.0 creates fully pipelined operations. Default value is 1.0. Increase the pipelining factor to help meet timing imposed by stream frequencies. Decrease this factor to help reduce latency, though this is usually only a concern when loops are present, and to reduce the number of flip-flops.

```
1 optimization.pushPipeliningFactor(0.5);
2 optimization.popPipeliningFactor();
```

`pushDSPFactor(double dsp_factor)`

`pushDSPFactor(double dsp_factor, MathOps operations)`: Increases or decreases the number of DSPs used for all operations or a specific type of operation. Parameter `dsp_factor` should have a value greater or equal to 0.0 and less or equal to 1.0. Default value is 0.5. If the second variant is used, then operations parameter ought to be specified. Available values for operations are: ADD, SUB, MUL, DIV, NEG, ALL, ADD_SUB, and MUL_DIV.

```
1 optimization.pushDSPFactor(0.5);
2 optimization.pushDSPFactor(0.5, MathOps.ADD);
```

DSP Factor Mapping to Xilinx CoreGen Options

DSP Factor mapping for **floating-point multiplication**:

DSP Factor (d)	Type	DSP Usage
$0.0 \leq d < 0.25$	8,24 or 11,53	No_Usage
$0.25 \leq d < 0.5$	8,24 or 11,53	Medium_Usage
$0.5 \leq d < 1.0$	8,24 or 11,53	Full_Usage
1.0	8,24 or 11,53	Max_Usage
$0.0 \leq d < 0.25$	custom	No_Usage
$0.5 \leq d < 1.0$	custom	Full_Usage
1.0	custom	Max_Usage

DSP Factor mapping for **floating-point addition**:

DSP Factor (d)	Type	DSP Usage
$0.0 \leq d < 0.75$	8,24 or 11,53	No_Usage
$0.75 \leq d < 1.0$	8,24 or 11,53	Full_Usage
$0.0 \leq d \leq 1.0$	custom	No_Usage

DSP Factor mapping for **fixed-point multiplication**:

DSP Factor (d)	Multiplier Construction	Opt Goal
$0.0 \leq d < 0.166$	Use_LUTs	n/a
$0.166 \leq d < 0.333$	Use_Mults	Area
$0.333 \leq d \leq 1.0$	Use_Mults	Speed

8.1.3 Placement constraints

Placement is the process of placing (mapping) a DFE design into hardware. The programmer may effect this process by specifying appropriate constraints for build tools (provided by the FPGA vendor company). These constraints are accessible via `optimization.setGroup<Prim>AreaConstraint`.

It is advised to constrain the placement of the specified primitives to the area defined by: (x0, y0) - (x1, y1). This improves timing in the cases when build tools perform sub-optimal placement. The default value is that no area groups are specified.

```

1 pushGroup("name of area");
2 // <Prim> is one of: "DSP", "RAMB18", "RAMB36" or "Slice"
3 optimization.setGroup<Prim>AreaConstraint(x0, y0, x1, y1);
4 // code that optimization applies to
5 popGroup();

```

8.1.4 Input registering

For every input, there is a register that stores its value. This behaviour could be changed using `io.pushInputRegistering`. This disables input registering, reduces latency, and reduces the flip-flop usage. By default, this optimization is enabled.



Warning: Reading when input is not enabled returns the previously stored value. If input registering is disabled, the behavior of input changes: reading when input is not enabled, reads undefined data.

```
1 io .pushInputRegistering(false);
2 // code that setting applies to
3 io .popInputRegistering();
```

8.1.5 Per-stream optimizations

This feature is accessed via `optimization.<OptName>(T streamName)`, and could be enhanced with appropriate parameters.

`pipeline(T stream)`: Adds a layer of registers to the stream. Enable this optimization if you have a timing failure due to fan-out from a stream.

```
1 DFEVar result = x * x + x;
2 result = optimization.pipeline(result);
```

`limitFanout(T stream, int max_fanout)`: Adds a layer of registers to the stream and tries to duplicate any of these registers, when the fan-out is more than `max_fanout`. Enable this if you have a timing failure due to fan-out from a stream.

```
1 DFEVar result = x * x + x;
2 result = optimization.limitFanout(result);
```

8.2 Manager optimizations

Within Manager, different types of optimization are available. More precisely, these are used as described in the following sections:

8.2.1 LMem clock frequency

This feature is accessed via `config.setOnCardMemoryFrequency(LMemFrequency clock)`, and is enhanced with appropriate arguments.

It sets the base clock frequency for LMem. Decreasing this value helps when there are infrastructure timing errors reported against LMem. Increasing this value gains additional bandwidth out of LMem.

```
1 config.setOnCardMemoryFrequency({MAX4_400});
```

LMem clock frequencies with appropriate name constants for different cards, are given in [Table 8.1](#).

Card	Label	Clock Frequency (MHz)
MAX2C	MAX2_200	200
	MAX2_250	250
	MAX2_300	300
VECTIS	MAX3_300	300
	MAX3_333	333.333
	MAX3_350	350
CORIA	MAX3_400	400
	MAX4_333	333.333
	MAX4_400	400
MAIA	MAX4_533	533.33
	MAX4MAIA_400	400
	MAX4MAIA_533	533.333
GALAVA	MAX4MAIA_666	666.666
	MAX4MAIA_733	733.333
	MAX4MAIA_800	800
ISCA	MAX4GALAVA_400	400
	MAX4GALAVA_666	666.666
	MAX4ISCA_400	400
	MAX4ISCA_533	533.333
	MAX4ISCA_666	666.666
	MAX4ISCA_733	733.333

Table 8.1: Available clock frequencies per specific cards

8.2.2 Stream clock frequency

Default Stream Clock Frequency: It sets the default clock frequency for Manager blocks. Increasing this value typically improves performance. Decreasing it often helps meet the timing. Default value is 100 (MHz).

Typically, the maximum frequency a Kernel is able to achieve is around 250MHz on Coria, Maia, and Isca. It is not recommended to try to go much above this value (250MHz).

Using a Manager class:

```
Manager manager = new Manager(params);  
manager.setClockFrequency(200);
```

Using a CustomManager:

```
CustomManager manager = new CustomManager(params);  
manager.config.setDefaultStreamClockFrequency(200);
```

Configurable Stream Clock Frequency: It sets the default clock frequency for Manager blocks. Increasing this value improves performance. Decreasing meets timing.

```
1 ManagerClock myClk = generateStreamClock(myClk , 120);  
2 myKernel.setClock(myClock);
```

8.3 Build configuration

MaxCompiler offers some embedded build configuration optimization methods to help programmers in optimizing their designs. Let us assume there is a class that extends `EngineParameters` and has `CustomManager` instance passed in as `manager`; then:

```
1 BuildConfig buildConfig = manager.getBuildConfig();
```

will give us the `BuildConfig` instance that we can use to set all desired optimizations.

8.3.1 setBuildEffort(Effort effort)

Cost table represents the process of mapping a design into hardware. Default behaviour of the cost table is to balance between resource usage and processing time; the cost table will try to map a design until a predefined point of resource/ time consumption.

Nevertheless, programmer may effect point till which cost table will do the actual mapping. This changes are available with using `setBuildEffort(Effort effort)`.

Available options:

- `Effort.LOW`
- `Effort.MEDIUM` (This is the default option)
- `Effort.HIGH`

- `Effort.VERY_HIGH`

Choosing a higher build effort will increase resource usage and build time, but a higher build level will also increase chances to meet timing on a single cost table.

On the other hand, increasing the effort may save compilation time due to satisfying an earlier cost table. But, if DFE design is relatively simple and build is going to meet timing anyway, then it will take longer with a higher effort level. For a relatively full chip, the general advice is first to use `Effort.HIGH`, and only to move to `Effort.VERY_HIGH` only if the first did not work.

To set `setBuildEffort(Effort effort)` option use:

```
1 buildConfig.setBuildEffort(Effort.HIGH);
```

8.3.2 `setEnablePhysicalSynthesisRegDuplication(Switch switch)`

This option adds registers when there is a high fan-out, in order to reduce the length of some paths.

Switching this option on will make your builds take a lot longer (around 2-3x per cost table, so 8 hours could easily become 16-24 hours). Using `setEnablePhysicalSynthesisRegDuplication(Switch switch)` will also increase register usage.

The reason for high time costs is that `setEnablePhysicalSynthesisRegDuplication(Switch switch)` effectively does Place and Route without the registers, to get an idea of where it should put them, and then runs again with the registers at selected positions (double Place and Route).

Available options:

- `Switch.AUTO` (This is the default option)
- `Switch.ON`
- `Switch.OFF`

You generally only want to use this option when you have exhausted other options and you are trying to get the frequency higher.



This is only an option on Altera (MAX4), not Xilinx (MAX2 and MAX3).

To set this option use:

```
1 buildConfig.setEnablePhysicalSynthesisRegDuplication(Switch.ON);
```

8.3.3 `setMPPRCostTableSearchRange(int min, int max)`

Place and route is performed using heuristic algorithms, such as simulated annealing, which is in essence a random process. Each cost table is equivalent to running a random process with a different seed - the same cost table will give the same result, but different cost tables may meet timing when other cost tables do not. Since the cost tables are run sequentially, if you meet timing on the first one,

8.3 Build configuration

then MaxCompiler will not bother with the rest, so the time for the build does not change whether you are using this option or not. If your design fails to meet timing on all cost tables, then it will take longer time. As a general rule of thumb, if running only one or two cost tables has not worked, then just run more (this also helps identify patterns in the timing report). On the other hand, if you have already run 8 cost tables that have not met timing, then your chances of meeting timing are not high, and you should probably consider other options.

Available range: $1 \leq \text{min} \leq \text{max} \leq 100$

To set `setMPPRCostTableSearchRange(int min, int max)` you can directly put `min` and `max` values:

```
1 buildConfig.setMPPRCostTableSearchRange(1, 4);
```

However, it is preferable to define `min` and `max` values as `EngineParameter`. To do so, double click on appropriate `RunRule` and use Engine Parameters, section as depicted at [Figure 8.1](#). Then access defined values with getters:

```
1 buildConfig.setMPPRCostTableSearchRange(getMPPRStartCT(), getMPPREndCT());
```

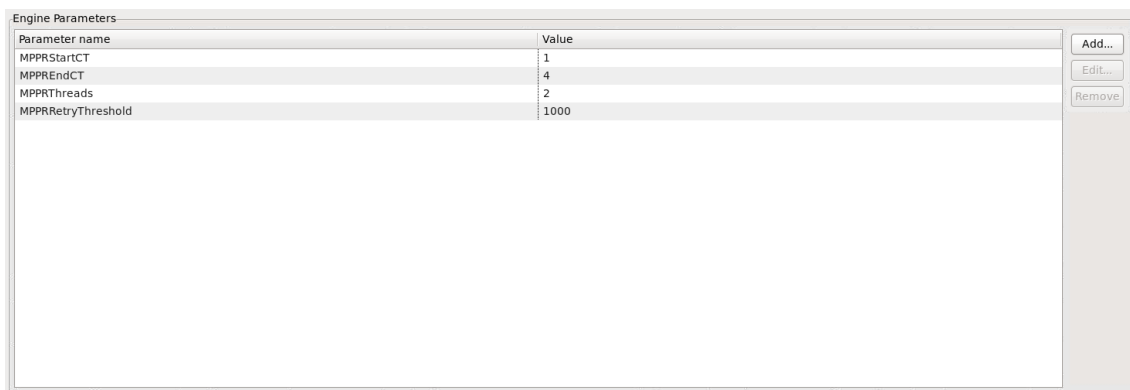


Figure 8.1: Set Engine Parameters that are going to be accessible through appropriate getters

8.3.4 setMPPRParallelism(int numberOfThreads)

It sets the number of different cost tables to be run in parallel. Higher values will use a lot more of processor and memory resources, but will decrease the time to build (assuming it takes several cost tables to meet timing).

To set this option you can directly put the `numberOfThreads` value:

```
1 buildConfig.setMPPRParallelism(4);
```

It is preferable to define the `numberOfThreads` value as `EngineParameter`. To do so, double click on appropriate `RunRule` and use Engine Parameters, section as depicted at [Figure 8.1](#). Then access defined values with getters:

```
1 buildConfig.setMPPRParallelism(getMPPRThreads());
```

8.3.5 `setMPPRRetryNearMissesThreshold(int retryTreshold)`

If you are running a build with the effort set to HIGH, and you set `retryTreshold`, for example, to 1000, then for any cost table that scores less than 1000, it will run the same cost table again with the VERY_HIGH effort.

To set `setMPPRRetryNearMissesThreshold(int retryTreshold)` you can directly put the `retryTreshold` value:

```
1 buildConfig.setMPPRRetryNearMissesThreshold(1000);
```

However, it is preferable to define the `retryTreshold` value as `EngineParameter`. To do so, double click on appropriate `RunRule` and use `Engine Parameters` section, as depicted at [Figure 8.1](#). Then access defined values with getters:

```
1 buildConfig.setMPPRRetryNearMissesThreshold(getMPPRRetryThreshold());
```

8.3.6 `setOptimizationGoal(OptimizationTechnique technique)`

It sets the primary optimization target.

- `OptimizationTechnique.SPEED`
- `OptimizationTechnique.AREA`
- `OptimizationTechnique.BALANCED` (This is the default option)

In general, you should use `SPEED` if you have a small design that you are trying to run at a high frequency (250MHz+), and use `OptimizationTechnique.AREA` when you are building something that is going to occupy a significant proportion of the reconfigurable area. Although the default is `BALANCED`, once you are using more than 70% of the logic resources, `AREA` tends to produce much better results.

To set this option:

```
1 buildConfig.setOptimizationGoal(OptimizationTechnique.AREA);
```

9

Global DFE Optimization Practices: Getting Maximum Performance

Programming in Space requires a revision of the programming strategy, as well as a revision of the debugging and optimization processes. Programming Maxeler Dataflow Engines (DFEs), in particular, is a lower level activity than writing conventional software. This is because DFEs are static compute engines and their memory system is purely physical. The programmer's task is to make design fit onto limited hardware resources available in DFEs.

On the macro level, we start by deciding which part of the data should be mapped to the memories on the DFE. Once data location is set, the corresponding pieces of code get transferred over to the DFE side, to minimize data movement. In essence, computation moves close to where the data is. Now, all we have to do is to maximize utilization of the CPU, the DFE resources, the memories, and all interconnect channels. If all units are doing useful work all of the time, the implementation is successful. Of course, at that point, the computation required on the DFE might not fit (compile).

Consequently, Appendix B ([Section 10.2](#)) has the datasheets with area requirements for various DFE units, operators and components.

9.1 Introduction

On the micro level, the underlying chip technology allows for different clock frequencies, as well as for adjustment of pipelining. This is a quite unusual optimization opportunity for software engineers, and a very familiar one to circuit designers.

Essentially, the structure of the dataflow graph needs to be mapped onto a flexible underlying substrate, and DFE abstractions provide a programming interface to low level optimizations.



Beware of toy problems: DFEs have up to 96GB of LMem and 5MB of FMem with over 10TB/s access into the FMem. Toy problems are easy to parallelise on ManyCore processors and do not offer any opportunity for DFEs to shine. Getting maximum performance is as much about the implementation effort as it is about picking sufficient scope, problem size, and algorithmic complexity.

A useful addition to this document is also AppGallery ("<http://appgallery.maxeler.com>") with various examples, and the Maxeler standard library providing highly optimised components that could be re-used to optimise relatively complex common cases, such as multi-input accumulators, flexible FMem-LMem buffers, etc. There are also continuously improved lecture slides (e.g., at "<http://cc.doc.ic.ac.uk/openspl14>").

9.2 Dataflow computing strategy

When programming MultiCore processors (Computing in Time), the main task is to identify the pieces of code that need to be moved around the Multicore machine. In contrast, when programming Dataflow processors (Computing in Space), the first step is to identify the data that should move to the DFE side. This goes hand-in-hand with looking into the loopflow graph to extract the dataflow: Flow of data is essentially a loop, where the loop index is used to index into an array.

Given the data movement requirements between loops, and the time spent at each loop nest, we decide on the location of data (CPU main memory, FMem, or LMem). Once the location of data is set, the corresponding pieces of code are moved to the DFE side, to minimize data movements. Computation goes close to the data locations. Efficiency depends of overlapping computation on the CPU and the DFE with data transfers between CPU and DFE, as well as LMem streaming. On the CPU side, we have the automatically generated SLiC interfaces where the advanced static and the dynamic interfaces allow for non-blocking execution of calls to the DFE. Key to maximum performance is maximum utilization: Keeping every resource busy with useful work at all times.

What if it does not work? Debugging is a key component in getting maximal performance. There are several features for debugging described in this tutorial. Process-wise, the key bit is to stick to a conservative process, changing only one thing at a time, in going from original software to a complete DFE implementation: see Lectures 15/16 at "<http://cc.doc.ic.ac.uk/openspl14>".

9.3 Fitting procedure

Computing in Time requires optimization of the time taken by individual functions. Computing in Space (see OpenSPL, "www.openspl.org"; see especially the video under "What is OpenSPL") requires optimization of space, to map a problem onto one of the potentially multiple DFEs. Rather than

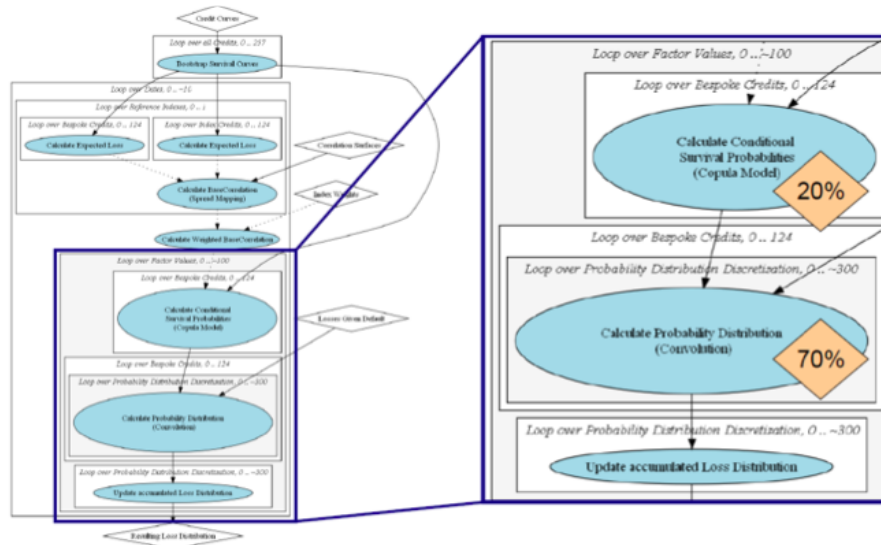


Figure 9.1: Loopflow Graph. The box represents a loop and ellipse represents a computational block without side effects, i.e., without well-defined inputs and outputs. Arrows show data movement. Percentages denote how much time is spend in the loop, out of the total execution time.

starting with a fixed space (such as with a microprocessor), we start with assuming that there is infinite space and we implement the solution assuming it can have any amount of space needed. Then, as the second step, we fold the computation onto the space that is actually available on the targeted substrate. This strategy was proposed in Alan Huang's MS Thesis "Computational Origami":

["http://www.opticsinfobase.org/ao/abstract.cfm?uri=ao-31-26-5419"](http://www.opticsinfobase.org/ao/abstract.cfm?uri=ao-31-26-5419).

The following image shows the process: We start by assuming infinite space and infinite bandwidth, then we consider the solution with infinite space and finite bandwidth, next with finite space and infinite bandwidth, and finally we arrive at the best solution for the real case, with finite space and finite bandwidth.

Figure 9.2 refers to the highest level strategy. More specifically, estimating performance for a dataflow implementation could be done easily since Kernels are static compile-time objects. Consequently, execution time is simply the product of bandwidth and data. Bandwidth is either set by pin limitations or by the loopflow graph (i.e., the amount of data flowing into a loop) and the speed (clock frequency) at which it can be processed. For Maximum Performance Computing, we assume we can process one wave of inputs and produce one wave of outputs, in steady state, given that there is a lot more data than is the length of any pipeline.

9.3.1 Techniques to fit designs onto DFE

With Maxeler DFEs, we are not physically folding circuits, but rather partitioning and molding the substrate. The overall process could be roughly split into:

1. Identify what data should be on the DFE at which point in time;
2. Identify what functionality should be on the DFE because its data is there;

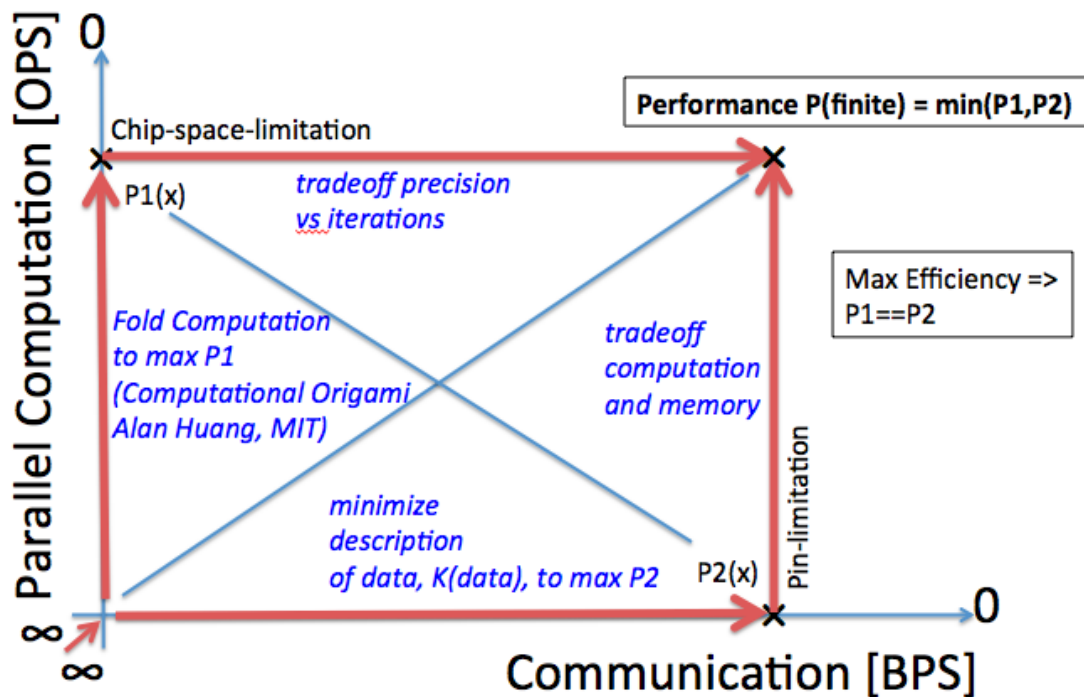


Figure 9.2: Design space exploration, starting with infinite resources, by traversing the rectangle, we arrive at the really constrained resources (Area and Bandwidth) case.

3. Identify how many DFEs will be needed, and if these DFE configurations should operate in parallel on multiple DFEs, or in sequence on a single DFE;
4. Make each DFE configuration fit;
5. Optimize performance for each one of the DFE configurations.

How do we make it fit? We have to vary the number of bits per variable, the amount of operations per DFE configuration, the level of **multi-piping**, and the amount of explicit and implicit FMem used (StreamOffset). In order to effectively optimize the space, it is important to consider how much space each one of the lines of MaxJ code (in your DFE program) consumes (see: [Figure 9.3](#)).

The resources are: (1) logic, lookup tables (LUTs), (2) registers (FFs), holding a single bit of data, (3) FMem (BRAM), memory blocks inside the chip, and (4) Fast Multipliers (DSPs).

Typically, the fitting exercise consists of replacing the expensive operations with equivalent cheaper ones (e.g., the divide operation is best done via multiplication by $1/x$), selectively moving certain computations back to the CPU, and trading off computation versus memory space (i.e., storing the result versus recomputing it).

9.4 How to make it fit?

There are three main stages of DFE programming, each one of them requiring a slightly different approach. In **Stage 1**, the DFE substrate is more than 100% full. In that case, we need to reconsider

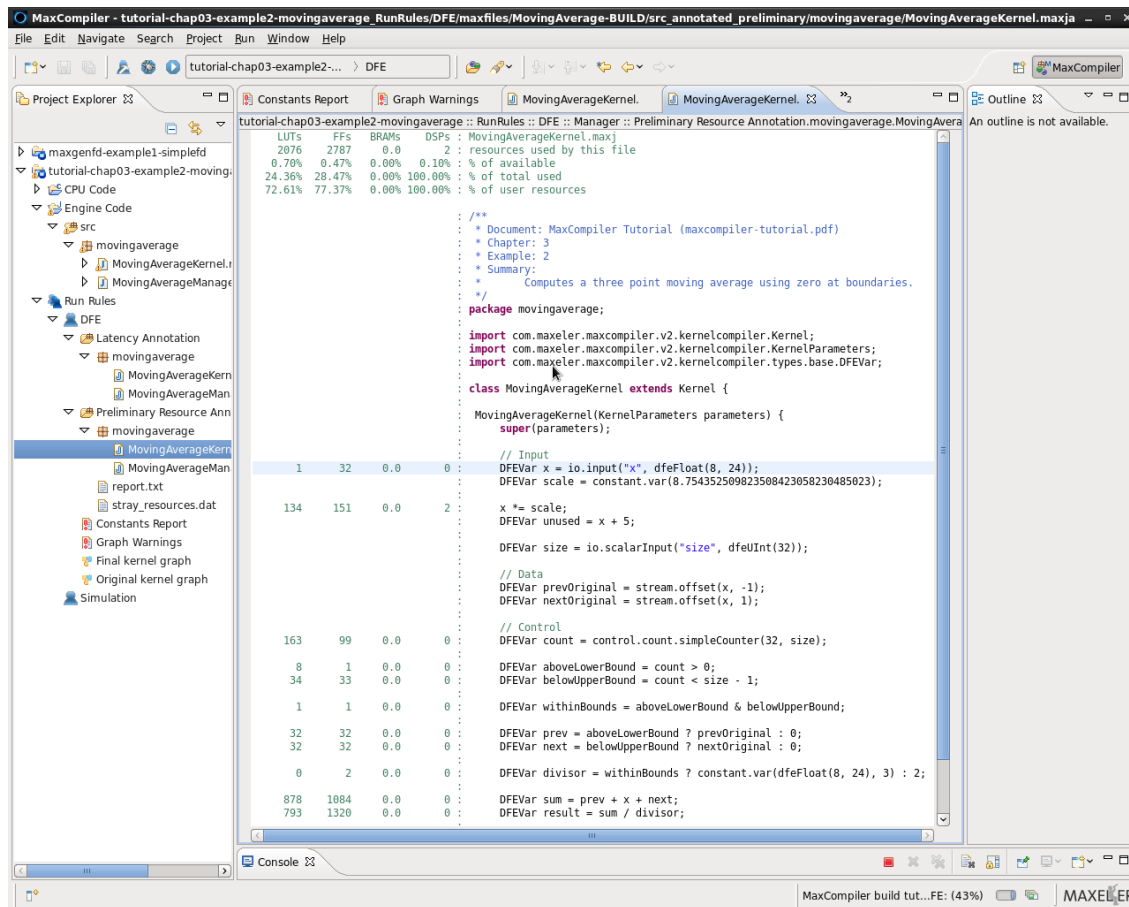


Figure 9.3: Each line in the DFE program has four values, showing how many resources of type 1, 2, 3, and 4 are used.

how much computation and data has been moved to the dataflow side, and we have to employ marco optimizations. In **Stage 2**, the DFE substrate is filled up close to 100% and we need to employ micro optimizations, to make the design fit. Finally, in **Stage 3**, once the DFE substrate is less than about 80% full, the final fine tuning of the clock frequency has to be done.

And now in more details:

9.4.1 Stage 1: DFE logic utilization > 100%: macro optimization

The following macro optimizations should lead your implementation to Stage 2.

1 Partition onto multiple DFEs (parallel or sequential on the same DFE)

- same .max file vs different .max file, with and without MaxRing

Using multiple DFEs requires careful consideration where the data remains and how it moves between DFEs, in order to minimize bottlenecks and maximize utilization. By splitting a computation onto two simultaneous DFEs, we now have to send data between the two DFEs and might be limited by the bandwidth between DFEs (PCI Express, Infiniband, MaxRing, etc.), which is many orders of magnitude slower than communication inside a single DFE.

2 Fold the design on the same DFE (create shared sets of reused arithmetic units)

The DFE programming lays out computation in space. As such, every operator in a DFE program creates a physical arithmetic unit in the DFE.

3 Trading Off Computation and Memory: recalculate versus store

Sometimes it pays off to recompute certain results, rather than to store and retrieve them. On the arithmetic level, it is well known that one can store the results of a function in a table or use a polynomial to recompute it each time. Similarly, for large scale simulations, we can store all timesteps, or we can store only every X timesteps and recompute the timesteps in the middle when needed.

4 Adjust the CPU - DFE balance

While optimising your design, keep watching out for opportunities to move computations from the DFE back to the CPU, and vice versa. The goal is to have both CPU and DFE busy and productive at all times.

5 Reduce floating-point precision and consider using fixed-point for some parts

Floating-point units take a huge amount of space compared to fixed-point. However, fixed-point computations could be scary for floating-point programmers. As a compromise, the solution is to create a dynamic floating-point design using dynamic scaling during the computation and managing the exponent separately from the fixed-point mantissa.

6 Multi-piping

Multi-piping helps in increasing parallelism, by replicating a certain set of computations (pipes), while at the same time allows sharing of (arithmetic) resources across these pipes. Of course, when creating multiple pipes, it is necessary to make sure there is enough data available for all the pipes to be fully utilised.

9.4.2 Stage 2: DFE logic utilization >80% and <100%: micro optimizations

1 Resource annotation and types of resources

Optimization requires understanding of resources. MaxCompiler's way to assist a programmer in this activity is by generating the resource annotated code. Resource annotated code gives programmers a knowledge of resource usage for each statement. This may help programmers to better understand hardware utilization and could allow him to, optionally, consider transforming the code, to change the usage of different resource types. Files that contain annotated code use the extension `.maxja` (see: [Figure 9.3](#)) and they can be found in the building directory on path:

`src_annotated/<package_name>/<Kernel.name>.maxja`

(in our example: `src_annotated/movingaverage/MovingAverageKernel.maxja`).

Alternatively, the annotated code can be access from the appropriate RunRule directory, e.g.,

`/maxideworkspace/tutorial-chap03-example2-movingaverage/RunRules/DFE/maxfiles/`

`MovingAverage-BUILD/src_annotated_preliminary/movingaverage/MovingAverageKernel.maxja`.

2 Adjust precision, remove division

Class `KernelLib` with its methods (`dfeRawBits(int width)`, `dfeInt(int width)`, `dfeFloat(int mantissa, int exponent)`, etc.) gives a programmer the ability to specify variables down to the bit level precision. This highly increases the efficiency and utilization. Parameter `width` defines the number of bits to be used for defined DFE variables (`DFEVar`). Parameter `mantissa`, specifies the number of bits for mantissa, and `exponent` specifies the number of bits for exponent.

Instead of using division, use multiplication with $1/x$, where x represents the divisor.

3 Move control flow within the graph

In order to share an arithmetic unit, it is necessary to share the operator in the program. For branches of an IF statement, for example:

$$a = (b > d) ? (b * c) : (d * c); \Rightarrow t = (b > d) ? b : d; a = t * c;$$

This operation is similar to pushing the mux ($?$ operator) up in the graph. Of course, this can also be done on a larger granularity, sharing bigger blocks of computation.

4 Share large units (e.g., function evaluation)

This is the same as before, but instead of sharing a multiplier, we are sharing the whole function call.

- adjust pipelining for different parts of the graphs;
- adjust partitioning into Kernels;
- adjust/merge/interleave inputs to Kernels;
- reduce `stream.offsets`:
 - consider blocking data and processing in multiple passes (a la MaxGenFD) for different parts of the data set;
 - Reorder data + computation, so smaller dimensions are innermost and need to be offset. E.g., offsets are often $n_{fast} * n_{med}$ in the total problem space of $n_{fast} * n_{med} * n_{slow}$. In this case, make sure n_{slow} is the largest dimension, since it is “free”: offsets are only $O(n_{fast} * n_{med})$;
 - Consider that an explicit memory use (`mem.alloc`) may be more efficient;
 - Consider compressing data before offsetting it, e.g., reduce the bit width.
- optimize the Multiplier/DSP, LMem, and FMem usage.

9.4.3 Stage 3: DFE logic utilization <80%, frequency optimization

- start with low **frequency** and push up slowly;
use many **cost tables**, and adjust/optimize pipelining factor;
- view chip tools **timing score** (reports), and find lines of code that limit timing;
- utilize `optimization.xxx`: e.g., `x=optimization.PIPELINE(x)`;
- control the pipelining: adjust pipelining in the control path.

9.5 Optimizing memory bound applications - data size

Memory operations could easily be a bottleneck for application performances. The bottleneck is typically the memory capacity or more likely the memory BUS width and/or the traffic intensity on BUS.

9.5.1 Differences in memory controllers of different cards

Memory controllers differ from card to card. The BUS width and the LMem clock frequency are two most important differences. This could lead to different behavior of same applications for different cards.

The LMem clock frequencies, with appropriate name constants for different cards, are given in [Table 8.1](#).

9.5.2 Data specific compression

Knowledge and understanding of data plays a crucial role in data specific compression. Having in mind data characteristics, the programmer may reduce memory usage and BUS load, by choosing different data presentations.

As an example: If programmer's design needs to operate on large matrices with a small number of non-zero elements, it would be much more efficient to store triplets (x coordinate, y coordinate, value) for each non-zero element, instead of having the whole matrix in memory.

9.5.3 Data encoding

Another way to reduce memory usage and BUS load is to encode data. Using available data encoding algorithms or a custom algorithm, the programmer could achieve a significant reduction of data size.

9.5.4 Re-organisation of the order of computations

Kernels or code segments that handle memory operations may be forced to wait longer than expected, due to busy BUS. This is especially the case if multiple Kernels or code segments try to access memory simultaneously. On the other hand, some other Kernels or code segments that are not bind to memory will be blocked, as a result of stalling parts of design before them.

To resolve this kind of problem, the programmer should consider re-organisation of the design - the order of computations. The first step is to pay attention to the moment when data will be requested: Sending the request right when data is needed may reduce traffic on BUS (instead of sending requests for all of the necessary data at once). The next step is rescheduling the non-memory related parts of design, in order to avoid unnecessary stalling, and to increase parallelism.

9.6 Optimizing compute performance - clock frequency

The compute performance of a Kernel is, broadly, set by its number of pipes (parallelism) multiplied by the clock frequency. Designs should normally be written to be flexible in the number of pipes, and then the number of pipes could be selected to fit on the targeted chip (see: [Section 9.4](#)). Once a number of pipes is selected, the maximum frequency for the design could be explored.



Remember that performance is $\text{pipes} \times \text{frequency}$, so maximum performance may be achieved with a slightly smaller than the maximum number of pipes, if it allows the whole design to run at a considerably higher frequency. E.g., a 10-pipe design at 250MHz will provide better performance than a 12-pipe design at 200MHz (2500 vs 2400 Mpoints computed per second).

Frequency optimization is an iterative process that consists of: test builds, evaluate results, apply optimizations, repeat:

STAGE 1: Explore frequency range: Run a series of maxfile builds at differing frequencies, each one with multiple cost tables.

- Use a small number of cost tables, but more than one. Individual cost tables are different random starting points, so you want to avoid looking at a single result because this may just fail randomly. A good number is 4 cost tables for an initial evaluation.
- Choose several frequencies around your target frequency. Depending on the Place and Route compute resources available to you, you may do this in parallel or sequentially. Typical frequency ranges for large designs are 150-200MHz on MAX3, and 200-250MHz on MAX4. Small designs may run as fast as 350MHz (but, if employing insufficient parallelism, may still deliver lower performance than larger designs running more slowly). A good practice is to use 10MHz intervals.
- Find the frequency at which the design just does not meet timing. If it meets timing at 180MHz, and fails at 190MHz, then take the 190MHz as your optimization starting point.

STAGE 2: Narrow frequency range

- Each cost table has a timing score that indicates how close the design was to being able to run at the target frequency. If the timing score is equal to zero, design has met the timing; however, if the timing score is larger than zero, design will fail to meet timing.
- If your minimum timing score was in the range 1-50000, this indicates that the design stands a good chance of meeting timing, if you run more cost tables.
 - * Try this by re-running the design with a larger number of cost tables specified, again depending on the level of compute resources available to you. A good practical number is 16. In theory, you can run up to 100 cost tables, but running >16 is usually only needed for the very last step of optimization of the most complex designs;
 - * If you have not done so, also increase the build effort to maximum (see: [Section 8.3.1](#));
 - * If the design now meets timing, proceed to the next highest frequency, and repeat the previous steps;
 - * If the design still does not meet timing, proceed to stage 3.

STAGE 3: Analyze failing cost tables

- To make a design meet timing at a higher frequency, we should analyze why it fails at present. To filter out the influence of randomness, this should be done across multiple cost tables and ideally across several frequency points. E.g., if you look mostly at a 190MHz build, you may want to cross-check your conclusions against a 200MHz build, as well.

- Open the MaxCompiler timing report for the maxfile. You can do this either using MaxIDE or by entering the path into your web browser (usually: `BUILDDIR/timingreport/index.html`).
- The top of the timing report has a summary of all cost tables for that build, while the detailed body identifies paths that can not function at the required frequency.
- Generally, you will find that there is a small number of paths that show up repeatedly as problematic.

STAGE 4: Optimize failing paths

- Broadly, there are three possible locations for failing paths:
 - * (A) MaxelerOS infrastructure (DRAM, PCIe, MaxRing, etc.) or the Manager;
 - * (B) Your code in a Kernel;
 - * (C) Kernel control infrastructure.
- For any of the above, if you have multiple Kernels in your DFE configuration, confirm that they all need to run on the same high speed clock.
 - * You may have one Kernel that can run slower than others, or one that needs to run the fastest;
 - * If so, modify your Manager code to generate multiple stream clocks (see the Manager Tutorial) and connect each Kernel to the appropriate clock;
 - * This allows the Place and Route process to only optimize each Kernel as much as necessary, rather than pushing all Kernels equally.
- In the case of (A), try one or more of the following options:
 - * If the failing paths are in the memory controller, try reducing the memory frequency (see: [8.2.2](#));
 - * If they are in the MaxRing controllers, try reducing the MaxRing speed or the number of lanes;
 - * Optimize your Kernels to reduce the amount of chip area they require (see: [Section 9.4](#)); This will free resources and will allow the MaxelerOS infrastructure to compile more easily;
 - * Check your Manager graph and consider the width of streams linking Kernels, memory, etc. in the Manager. Minimize stream widths to free resources, if practically possible. E.g., if your Kernel has a 128-bit input that reads data once every 8 cycles, instead consider generating a 16-bit input that reads data every cycle and uses `stream.offset` inside the Kernel to construct the appropriate 128-bit word every 8 cycles, then `stream.hold` to preserve that value as needed. Alternatively, consider increasing the width of some streams, if it means they will match the other streams they are connected to, in width, allowing aspect changing logic to be removed by the MaxCompiler when compiling Manager code.
- In the case of (B), try one or more of the following:
 - * Identify the responsible line of code using the timing report. Check that the operations match your expectations from the report, so you definitely have the right line (e.g., if the report says `Add->Mult`, then there should be code like $(a + b) * c$);
 - * If you have reduced the pipelining factor for that line of code, consider increasing it (`optimization.pushPipeliningFactor`);

- * Consider using `x = optimization.pipeline(x)` to add an additional register into the middle of an expression;
- * Look for fan-out issues. One major challenge for Place and Route is routing the same signal to many places; this can occur in your design easily, if you have a structure like:

```
DFEVar x =      ;
for (int i = 0 ; i < 200; i++) {
    output[i] = f(x, input[i]);
}
```

In this case, `x` must be connected to all 200 instances of the function `f`. You can optimize fan-outs by:

```
DFEVar x_orig =      ;
for (int i = 0 ; i < 200; i++) {
    if (i % 50 == 0) {
        x = optimization.pipeline(x_orig);
        output[i] = f(x, input[i]);
    }
}
```

This will generate a new `x` at `i=0`, `i=50`, `i=100`, `i=150`, so each `x` is only used 50 times. Obviously, the number 50 could be tuned, depending on the degree of fan-out minimization desired. You can also try `optimization.limitFanout()` (see: 8.1.5), which asks the synthesis tool to do the same function, but this is not as reliable.

- * Replace complex expressions with simpler ones that do the same job, or remove duplicate expressions (MaxCompiler will not perform common sub-expression removal automatically, because the effect on the graph schedule is unpredictable);
- * Consider increasing the DSP usage, to reduce the amount of logic that must be routed (see: 8.1.2), or transform some expressions into multiplies, so they could be put inside DSPs. Most obviously $x/y \rightarrow x * (1/y)$ if $(1/y)$ can be precomputed, but also consider combining multiple operations.

- In the case of (C), try one or more of the following:

- Increase the amount of CE (*Clock Enable*) pipelining. The CE signal is distributed automatically to all the registers in your Kernel, to allow the Kernel to stop and start as necessary; it fans out via a tree, so increasing the depth of this tree should reduce the fan-out on this signal. You can use `Kernelconf.optimization.setCEPipelining(N)` (see: 8.1.1) for this (note this is in the Kernel configuration, created in the Manager before the Kernel was instantiated, not in the Kernel itself).
- Use a global clock buffer to distribute the clock and CE for a Kernel. Do as above, but `setUseGlobalClockBuffer` (see: 8.1.1). This is mostly relevant for MAX3, particularly if you have a single large Kernel. There is a limited number of global clocks available, so you can not use this approach for 100 Kernels on the same chip.
- Reduce the number of registers in your Kernel, e.g., by reducing the pipelining factor (`optimization.pushP` (see: Section 8.1.2), or by removing the unnecessary `optimization.pipeline` calls.

STAGE 5: Rebuild and compare results

- Test individual changes or small sets of changes independently;

- Not all changes that should help will actually do so. Some may make things worse, so you need be aware of this, and you have to reconsider using them in your case;
- Make sure you clearly name your builds such that you can tell later what optimizations were being tested, and keep records in a spreadsheet (build name, optimization set, results).

Once you have succeeded in meeting a certain frequency, you can stop, or you can move onto the next higher clock frequency step. However, expect that, as you progress, you will eliminate the “low hanging fruit” quickly, so eventually you will be left with large amounts of work required for modest gains.

10

Appendices

10.1 Appendix A: Using WebIDE

WebIDE offers a subset of possibilities offered by MaxIDE. Some of possibilities are presented in following paragraphs:

10.1.1 Simulation watches

Upon completion of the run, the CSV report could be accessed from within WebIDE by opening “Debug output files”, as it could be seen in [Figure 10.1](#).

10.1.2 Simulation `printf`

The basic form of `simPrintf` prints the formatted message to the standard output of the application with the current value of the supplied list of objects for every tick. The WebIDE version of the same is given in [Figure 10.2](#).

10.1.3 Viewing the execution graph

The generated graphs, both original and final, are also available for viewing within WebIDE, by clicking to: *View Graphs* (see: [Figure 10.3](#)).

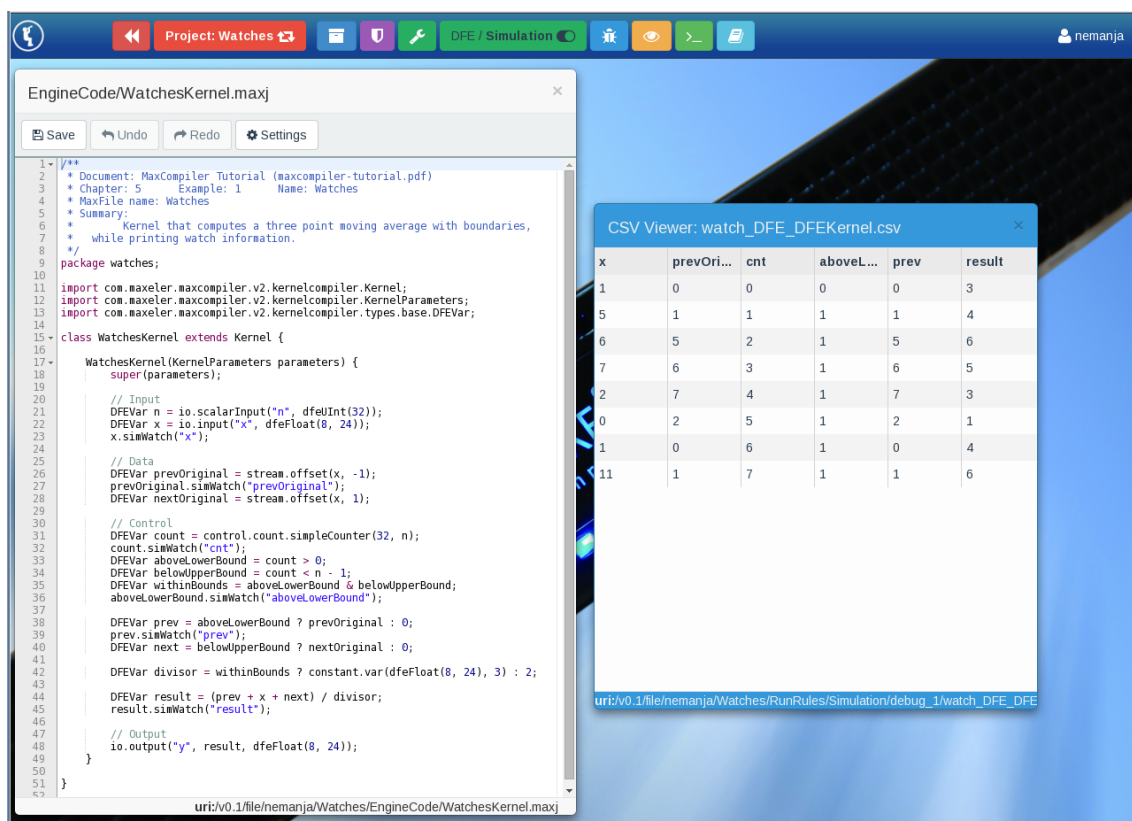


Figure 10.1: WebIDE displaying simulation watch data

10.2 Appendix B: Datasheet - how much space does it take

The following section depends on the underlying substrate (Maia, Isca, Galava, etc). It is always possible to find out all these details yourself, by looking at resource annotations, and by trying various ways of writing things.

Generally speaking, you need to have a reasonable estimate (just the order of magnitude) for how big the bitstreams are going to be. This requires building a budget for the 4 major resources on the FPGA.

Block memory could be reasonably estimated by taking the total number of bits, and by dividing it with the size of the memory; so, 36864bits on Vectis, or 20480bits in Maia/Isca/Galava.

The LUT and Register estimates are best done by building a small representative design using the estimated precisions and operators. You can use the detailed resource annotation to examine how many logic resources each operator takes, and you can use this as a basis for spreadsheet modelling. Multiplier blocks (DSPs) are “tiled” into as rectangles produced by the multiply, so for a 18x24 bit multiplier, you have $\text{ceil}(54/18) * \text{ceil}(54/24) = 9$ DSPs, for the IEEE double-precision floating-point multiply (53bit mantissa + 1 sign bit).

The table for DSP usage on Vectis is shown in [Figure 10.4](#).

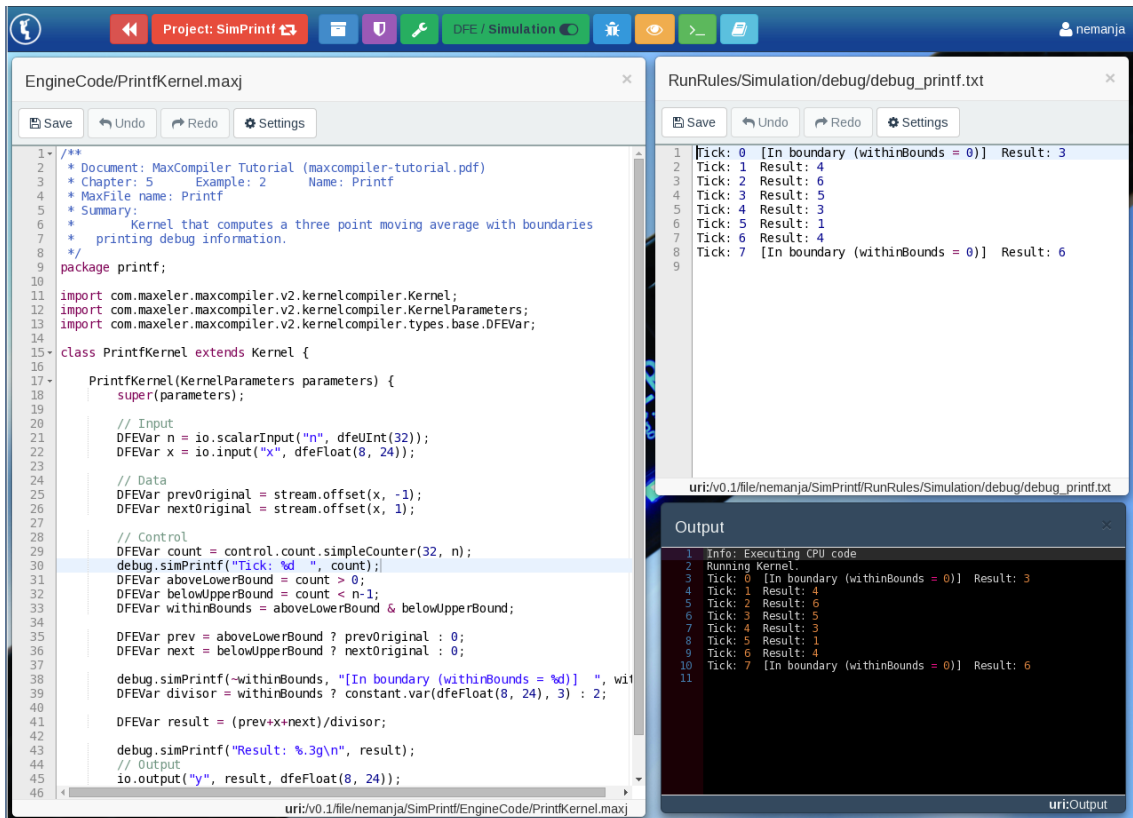


Figure 10.2: Simulation printf output in WebIDE

Scheduling resources are hard to estimate (unless large buffers are needed, i.e., stencils), and small scheduling buffers of 1-2 cycles could add up dramatically; careful examination of the `KernelName_photon_stats.csv`, along with the kernel graph, could show where the scheduling is coming from, and potentially where the re-ordering operations may remove scheduling FIFOs.

10.3 Appendix C: Double or Nothing

Understanding of hardware resource consumption is of vital importance for Maxeler Dataflow programmer. Before any further discussion related to debugging and optimization of DFE design, this section will picturesquely describe exploitation of hardware resources. Choosing the right precision to represent data is crucial. Maxeler Dataflow technology enables defining the precision all way down to bit level. Having that in mind, the main topic of this section is comparing differences on hardware level, when using fixed-point, single-precision floating-point, and double-precision floating-point precision.

First of all, let us define Total Cost of Ownership (TCO) as overall expense for implementing DFE application using desired precision.

Fixed-point DFE designs result in two times TCO advantage over competition.

Single-precision floating-point is ten times worse than fixed-point.

Double-precision floating-point is four times worse than single-precision floating-point.

Place and Route process for floating-point is ten times more expensive than for fixed-point.

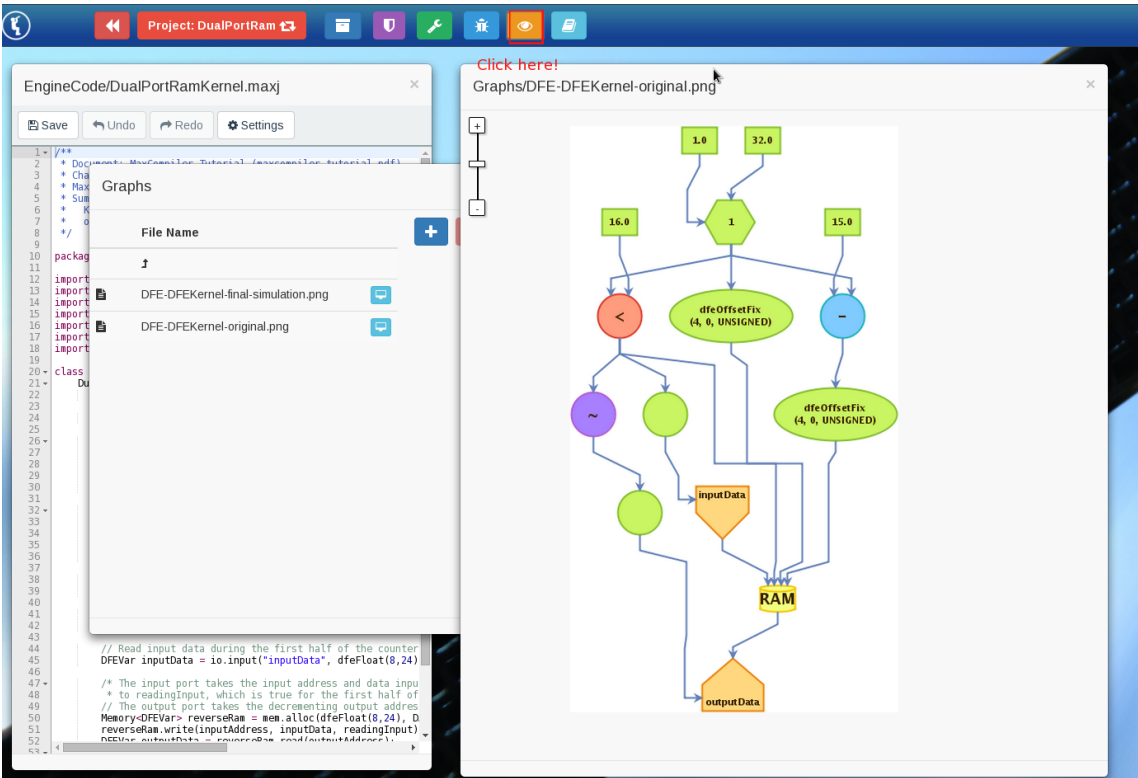


Figure 10.3: Kernel graph in WebIDE

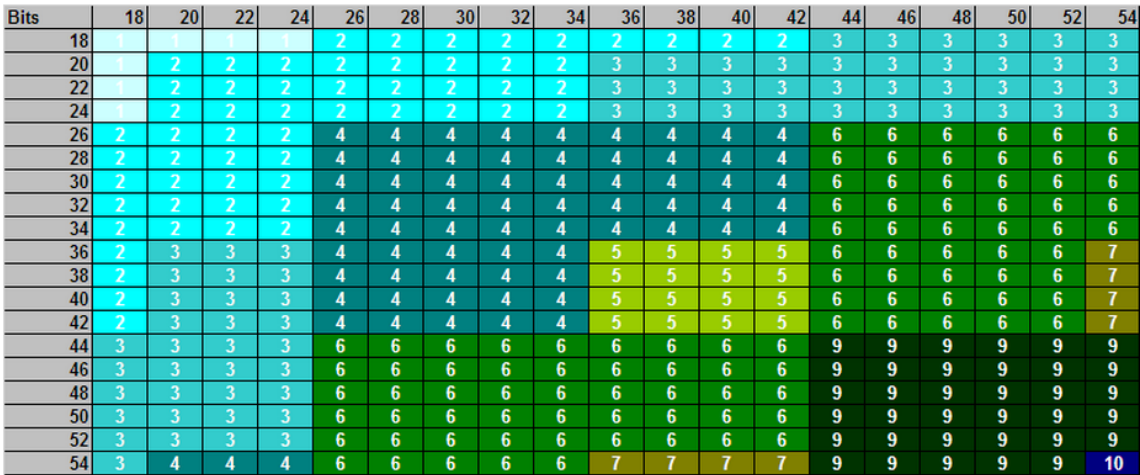


Figure 10.4: DSP usage on Vectis

Altera Stratix-V chips are used for the current generation MAX4 cards. Further discussions are related to Altera's chip and Max4 generation of Maxeler cards.

https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ug/ug_alftp_mfug.pdf

10.3.1 Stratix V D8 (Maia) FMem

Member Code	M20K		MLAB		Total RAM Bit (Kb)
	Block	RAM Bit (Kb)	Block	RAM Bit (Kb)	
D8	2,567	51,340	13,120	8,200	59,540

Figure 10.5: Stratix V D8, M20K, MLAB overview.

The Stratix V devices contain two types of memory blocks:

- 20Kb M20K blocks - blocks of dedicated memory resources. The M20K blocks are ideal for larger memory arrays while still providing a large number of independent ports.
- 640 bit memory logic array blocks (MLABs) - enhanced memory blocks that are configured from dual-purpose logic array blocks (LABs). The MLABs are ideal for wide and shallow memory arrays. The MLABs are optimized for implementation of shift registers for digital signal processing (DSP) applications, wide shallow FIFO buffers, and filter delay lines. Each MLAB is made up of ten adaptive logic modules (ALMs). In the Stratix V devices, you can configure these ALMs as ten 32 x 2 blocks, giving you one 32 x 20 simple dual-port SRAM block per MLAB. You can also configure these ALMs as ten 64 x 1 blocks, giving you 64 x 10 simple dual-port SRAM block per MLAB.

MaxCompiler does not use MLABs. This still leaves 6MB of M20Ks. Bandwidth in/out of FMem is more than 10TB/s!

10.3.2 Stratix V D8 (Maia) DSP blocks

Variant	Member Code	Variable - precision DSP Block	Independent Input and Output Multiplications Operator					18 x 18 Multiplier Adder Mode	18 x 18 Multiplier Summed with 36-bit Input
			9 x 9 Multiplier	16 x 16 Multiplier	18 x 18 Multiplier with 32-bit Resolution	27 x 27 Multiplier	36 x 18 Multiplier		
	D8	1,963	5,889	3,926	3,926	1,963	1,963	3,926	1,963

Figure 10.6: Stratix V D8, DSP overview.

The Stratix V devices variable precision DSP blocks offer the following features:

- high-performance, power-optimized, and fully registered multiplication operations;
- 9-bit, 18-bit, 27-bit, and 36-bit word lengths;
- 18 x 25 complex multiplications for FFTs;
- floating-point arithmetic formats;

- built-in addition, subtraction, and 64-bit accumulation unit to combine multiplication results;
- cascading 18-bit and 27-bit input bus to form the tap-delay line for filtering applications;
- cascading 64-bit output bus to propagate output results from one block to the next block without external logic support;
- hard pre-adder supported in 18-bit and 27-bit mode for symmetric filters;
- supports 18-bit and 27-bit with internal coefficient register bank for filter implementation;
- 18-bit and 27-bit systolic finite impulse response (FIR) filters with distributed output adder.

10.3.3 Stratix V ALM block

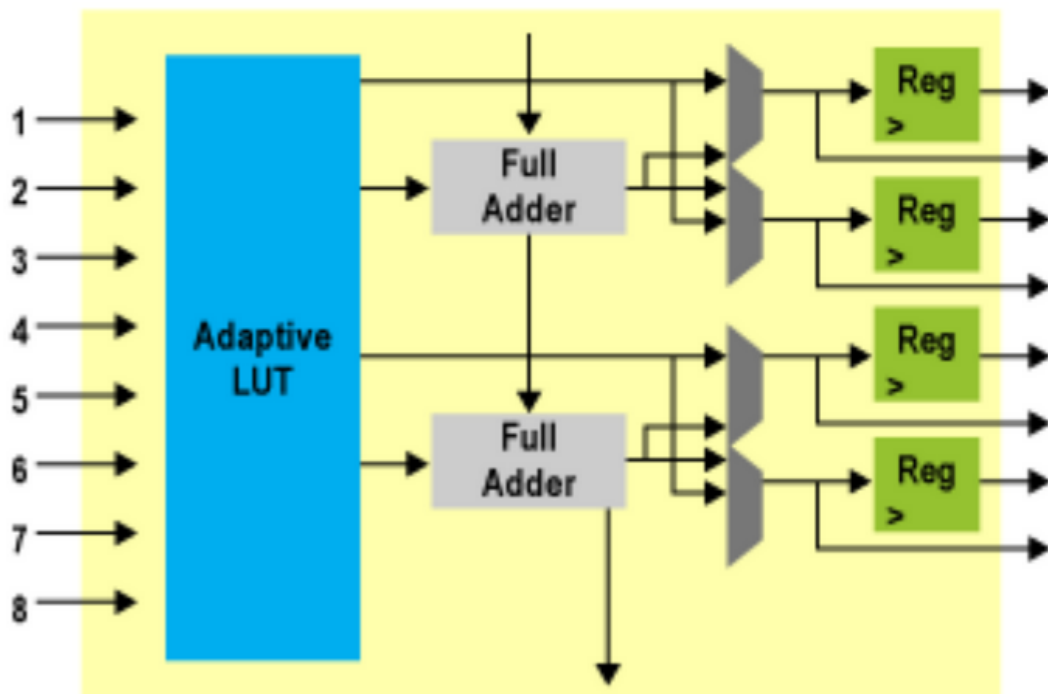


Figure 10.7: Stratix V ALM block.

One ALM can add two 2-bit numbers. One Stratix V D8 has 262 400 ALMs; when using fixed-point 32-bit precision, 16 400 adders are at your disposal.

10.3.4 Operations

We will compare hardware resource consumption for fixed-point and floating-point when performing addition and multiplication on Stratix V.

Addition As depicted in [Figure 10.11](#) and [Figure 10.12](#), 64-bit fixed-point addition is ten times smaller, using less hardware resources, than 32-bit single-precision floating-point on Stratix V. It is one pipeline stage for fixed-point addition versus five pipeline stages for floating-point, exponentially reducing the need for FMem buffers globally. When it comes to Place and Route, ten times smaller means up to ten times faster.

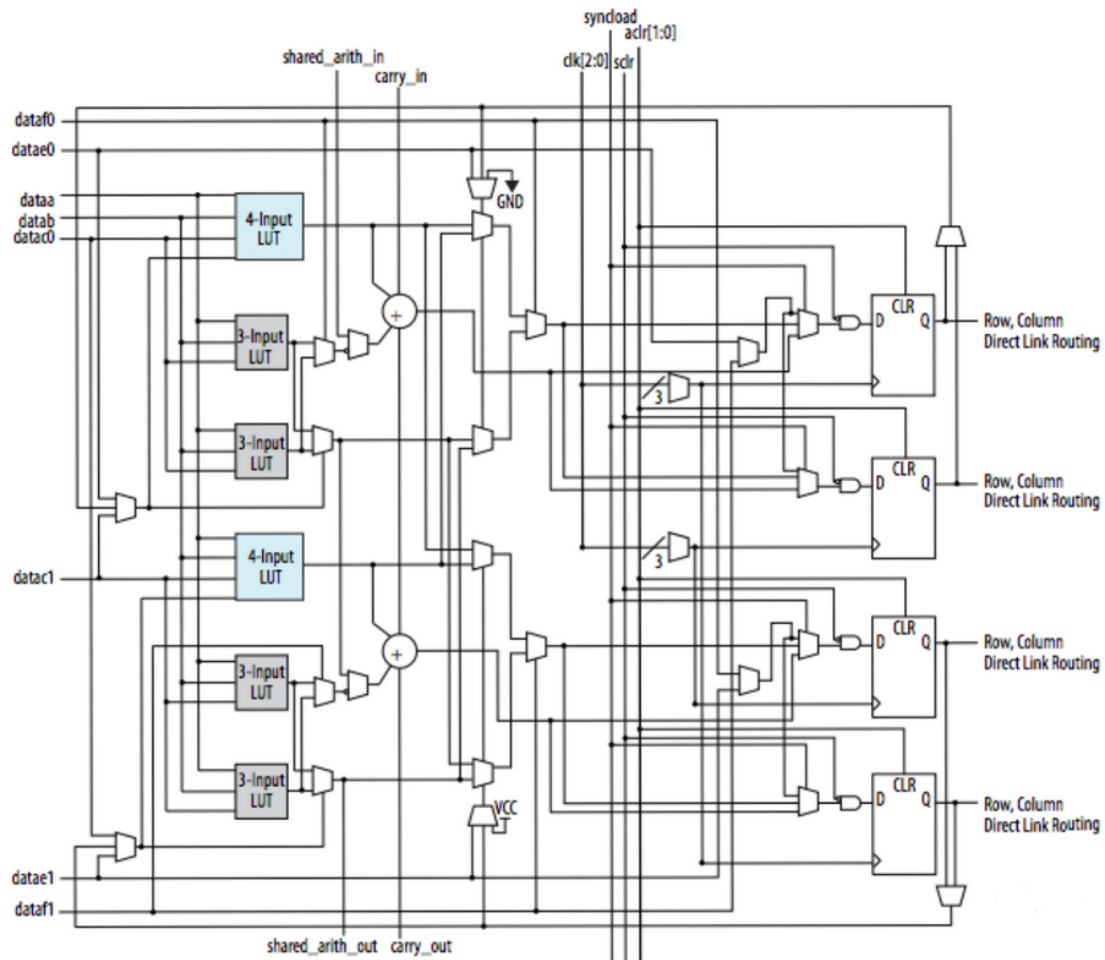


Figure 10.8: Logic implementation of ALM block.

Multiplication Conveniently, DSPs have “magic” bit numbers: 27, 54, ... and there are 2,000 DSPs in Stratix V D8 (see: [Figure 10.13](#) and [Figure 10.14](#)). It is one pipeline stage for fixed-point addition versus three to five pipeline stages for floating-point, exponentially reducing the need for FMem buffers globally. Fixed-point brings huge improvement in FMem usage. 27-bit fixed-point is optimal. 136 ALMs for single-precision floating-point means up ten times slower to Place and Route.

10.3.5 LMem bandwidth

When comes to LMem bandwidth, Max4 Maia has peak 77GB/s. If LMem bandwidth is an issue, programmer need to employ data compression and data decompression in and out of LMem. Have in mind that non-integer codes may result in lossy compression. More about compression and memory bound applications in general you can find in [Section 9.5](#).

10.3.6 Space, time, and value story

Preparing data in proper manner may actually prove to be most beneficial to computing. Process of data preparation should be directed towards discretization of space, time, and value.

Of course, some problems are easier to convert to fixed-point than others.

Function	Precision	Latency	f_{MAX}	ALMs	M10K	M20K	DSP Blocks	Logic Registers	
								Primary	Secondary
Add	Single	5	364.83	366	--	0	0	299	19
	Double	7	329.49	834	--	0	0	801	53
AddSubtract	Single	5	354.74	489	--	0	0	411	29
	Double	7	338.41	1,106	--	0	0	1,039	134
Cube Root	Single	8	420.17	114	--	5	2	124	11
	Double	20	277.7	520	--	11	10	997	17
Divide	Single	13	363.5	377	--	3	4	591	71
	Double	23	270.86	1,091	--	20	15	2,274	120
Multiply	Single	3	399.52	136	--	0	1	72	1
	Double	4	250.75	312	--	0	4	237	5
Power	Single	31	261.23	1,171	--	8	12	1,492	83
	Double	60	267.81	3,555	--	13	37	5,347	244
Square Root	Single	6	393.7	112	--	3	2	129	7
	Double	17	274.12	458	--	8	9	1,019	41
Subtract	Single	5	320.41	360	--	0	0	299	14
	Double	7	338.52	835	--	0	0	801	51

Figure 10.9: Stratix V Single-Double factor.

10.3.7 Floating-point in DFE design

Transferring segments of code from software implementation to DFE, should be done in several iterations. First of all, software implementation should be done using floating-point. However, the first DFE implementation has to be in fixed-point. Later, possibly and carefully introduce floating-point where needed.

Function	Precision	Latency	f _{MAX}	ALMs	M10K	M20K	DSP Blocks	Logic Registers	
								Primary	Secondary
Add	Single	4	296.4	49	--	0	1	0	0
	Double	7	296.3	840	--	0	0	779	67
AddSubtract	Single	5	319.39	483	--	0	0	408	37
	Double	7	289.77	1,106	--	0	0	1,006	156
Cube Root	Single	10	432.9	126	--	5	2	121	0
	Double	24	282.09	594	--	11	10	1,155	29
Divide	Single	16	347.34	394	--	3	4	561	66
	Double	30	258.26	1,208	--	20	15	2,175	136
Multiply	Single	3	288.4	49	--	0	1	0	0
	Double	5	288.35	312	--	0	4	236	26
Power	Single	40	262.12	1,335	--	8	14	1,523	127
	Double	73	237.7	3,957	--	13	37	5,362	305
Square Root	Single	8	432.9	124	--	3	2	118	8
	Double	22	249.25	539	--	8	9	1,000	34
Subtract	Single	4	296.9	49	--	0	1	0	0
	Double	7	296.82	842	--	0	0	783	76

Figure 10.10: Stratix 10 Single-Double factor.

	Function	Precision	Latency	f _{MAX}	ALMs	M10K	M20K	DSP Blocks	Logic Registers	
									Primary	Secondary
Stratix V	Add	Single	5	364.83	366	--	0	0	299	19
		Double	7	329.49	834	--	0	0	801	53
Stratix 10	Add	Single	4	296.4	49	--	0	1	0	0
		Double	7	296.3	840	--	0	0	779	67

Figure 10.11: Comparing StratixV and Stratix10 for Add operation.

FIXED POINT	4 bits	8 bits	16 bits	32 bits	64 bits	128 bits
	2 ALM	4 ALMs	8 ALMs	16 ALMs	32 ALMs	64 ALMs

Figure 10.12: ALM consumption for Add operation using fixed-point.

	Function	Precision	Latency	f_{MAX}	ALMs	M10K	M20K	DSP Blocks	Logic Registers	
									Primary	Secondary
Stratix V	Multiply	Single	3	399.52	136	--	0	1	72	1
		Double	4	250.75	312	--	0	4	237	5
Stratix 10	Multiply	Single	3	288.4	49	--	0	1	0	0
		Double	5	288.35	312	--	0	4	236	26

Figure 10.13: Comparing StratixV and Stratix10 for Mul operation.

FIXED POINT	4 bits	8 bits	16 bits	27 bits	54 bits	108 bits
with ALMs only	16 ALMs	64 ALMs	256 ALMs	378 ALMs	1,512 ALMs	5,832 ALMs
OR						
with DSP	1 DSP	1 DSP	1 DSP	1 DSP	4 DSPs+ x ALMs	16 DSPs+ x ALMs

Figure 10.14: ALM/DSP consumption for Mul operation using fixed-point.

SLiC API Index

`max_watch_range`, [7](#)

MaxJ API Index

`dfePrintf`, [9](#), [15](#)

`DFEVar`, [1](#), [5](#)

`simPrintf`, [5](#), [9](#), [10](#), [65](#)

`simWatch`, [5](#)

Index

debugging
 MaxDebug, *see* MaxDebug
 printf, [9](#), [15](#), *see* printf
 watches, *see* watches

directories
 debug output, [35](#), [36](#)

MaxDebug, [24](#)
 stream status blocks, [21](#)

maxdebug, [17](#)

maxdump, [17](#)

maximumperformance, [53](#)

printf, [9](#), [15](#), [35](#)
 output directory, [36](#)
 standard output, [36](#)

stream status blocks, [21](#)

timeouts, [36](#)

watches, [5](#)
 output directory, [35](#), [36](#)
 range, limiting, [7](#)

