MaxCompiler
Manager Compiler Tutorial

Version 2014.2

MAXELER
Technologies
MAXIMUM PERFORMANCE COMPUTING

Contents

Preface

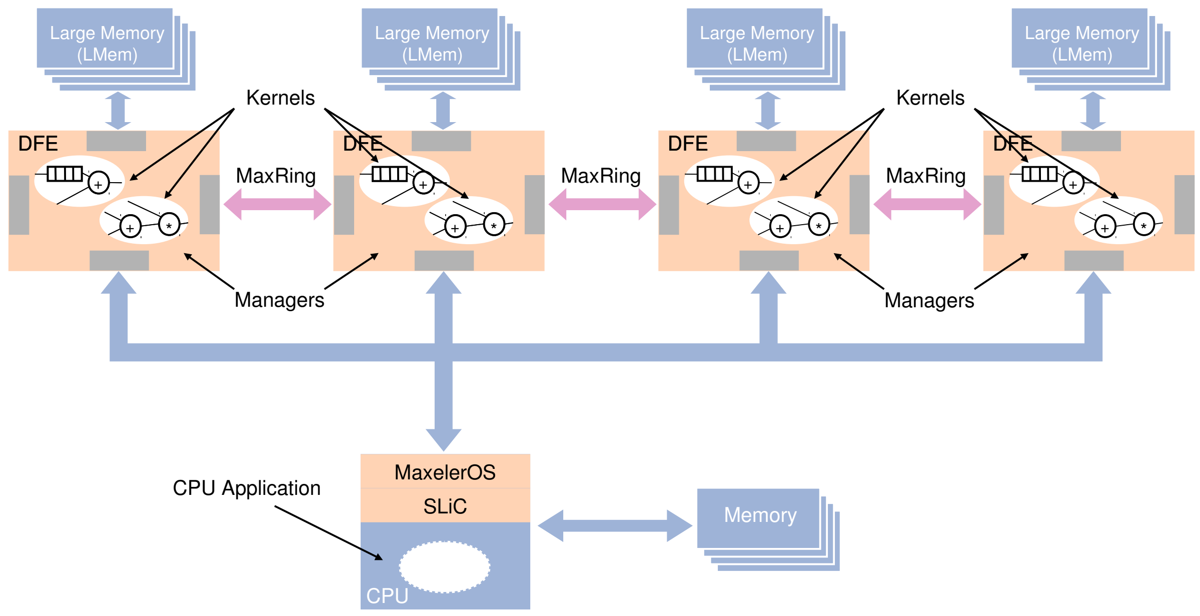


Figure 1: Overview of a Maxeler acceleration system

Figure 1 illustrates the architecture of a Maxeler dataflow supercomputing system which comprises DFEs attached directly to local memories and to a host CPU. In a Maxeler solution, there may be multiple DFEs connected together via high-bandwidth **MaxRing** interconnect. A DFE configuration is made up of one or more **Kernels** and a **Manager**. Within a Kernel, streams provide a predictable environment for you to concentrate on data flow and arithmetic. Managers provide a predictable input and output streams interface to the Kernel.

Managers provide a Java API for:

- configuring connectivity between Kernels and external I/O
- controlling the build process

MaxCompiler provides pre-configured Managers, including the **Standard Manager**, and the **Manager Compiler** for creating complex Managers of your own.

The Standard Manager, which allows a single Kernel in each DFE, is included with MaxCompiler. The Standard Manager is covered at the end of the **Multiscale Dataflow Programming** document.

The Manager Compiler allows you to create complex Manager designs, with multiple Kernels and complex interaction between Kernels and with IO resources.

Purpose of this document

This tutorial focuses on the Maxeler Manager Compiler.

The Maxeler Kernel Compiler is covered in detail in the **Multiscale Dataflow Programming** document, also provided with MaxCompiler.

Each section introduces a new set of features and goes through examples showing their use.

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 displayed in `typewriter` font.

Java methods and classes are shown using the following format:

*DFEVar **mem**.romMapped(**String** name, DFEVar addr, DFEType type, **double**... data)*

C function prototypes are similar:

***void** max_set_mem_uint64t(max_actions_t ∗actions, **const char** ∗block_name, **const char** ∗mem_name, size_t index, uint64_t v);*

Actual Java usage is shown without italics:

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

C usage is similarly without italics:

max_set_mem_uint64t(actions, "hdl_node_mapped_mem_bus", "example", 0, 1);

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

```
package hostloopback;

import com.maxeler.maxcompiler.v2.build.EngineParameters;
import com.maxeler.maxcompiler.v2.managers.custom.CustomManager;
import com.maxeler.maxcompiler.v2.managers.custom.DFELink;
import com.maxeler.maxcompiler.v2.managers.custom.blocks.KernelBlock;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.CPUTypes;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.EngineInterface;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.InterfaceParam;

class HostLoopbackManager extends CustomManager {

    HostLoopbackManager(EngineParameters engineParameters) {

        super(engineParameters);

        KernelBlock k = addKernel(
            new HostLoopbackKernel(makeKernelParameters("HostLoopbackKernel")));


        DFELink x = addStreamFromCPU("x");
```

1 Manager Compiler

The Manager Compiler allows the creation of more complex Managers than the Standard Manager. A Manager created using the Manager Compiler can include multiple Kernels, MaxRing interconnects between devices, multiple memory controller groups and memory command generators, memory control streams generated by Kernels or the CPU and blocks to split and join streams.

1.1 Blocks and Streams

A Manager Compiler design consists of two main types of object: **blocks** and **streams**. A block performs some kind of function or connects to some sort of resource and has a number of input streams and a number of output streams. Blocks include Kernels themselves, blocks for splitting and joining streams and implicit blocks for interfaces to resources external to the Dataflow Engine (DFE).

Streams in the Manager Compiler are unidirectional and asynchronous from each other. The asynchronous nature of streams in the Manager Compiler is an important difference from streams within a Kernel, which are all synchronous. Streams in the Manager Compiler can only be connected to one input stream and the output of a stream can only be connected to one other stream.

1.2 Getting Started

To demonstrate basic use of the Manager Compiler, we walk through a simple example, Example 1, in this section. The example has a single Kernel that takes an input stream, “x”, of 32-bit unsigned numbers, adds one and writes the result to an output stream, “y”. We generate input data on the CPU and stream this to the input of the Kernel. We take the output from the Kernel and stream this back to the CPU.

Listing 1 shows the trivial Kernel for this example. Our Manager design is shown in *Listing 2*.

Listing 1: Trivial example Kernel (HostLoopbackKernel.maxj).

```
/**
 * Document: Manager Compiler Tutorial (maxcompiler-manager-tutorial.pdf)
 * Chapter: 1        Example: 1       Name: Host loopback
 * MaxFile name: HostLoopback
 * Summary:
 *      A simple kernel that increments values in a stream by 1.
 */
package hostloopback;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;

class HostLoopbackKernel extends Kernel {
	HostLoopbackKernel(KernelParameters parameters) {
		super(parameters);

		DFEVar x = io.input("x", dfeUInt(32));
		DFEVar y = x + 1;
		io.output("y", y, dfeUInt(32));
	}
}
```

Listing 2: Manager for our simple example (HostLoopbackManager.maxj).

```
/**
 * Document: Manager Compiler Tutorial (maxcompiler-manager-tutorial.pdf)
 * Chapter: 1      Example: 1      Name: Host loopback
 * MaxFile name: HostLoopback
 * Summary:
 *      Stream data from the host to the kernel and back to the host again.
 */

package hostloopback;

import com.maxeler.maxcompiler.v2.build.EngineParameters;
import com.maxeler.maxcompiler.v2.managers.custom.CustomManager;
import com.maxeler.maxcompiler.v2.managers.custom.DFELink;
import com.maxeler.maxcompiler.v2.managers.custom.blocks.KernelBlock;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.CPUTypes;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.EngineInterface;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.InterfaceParam;

class HostLoopbackManager extends CustomManager {

    HostLoopbackManager(EngineParameters engineParameters) {
        super(engineParameters);

        KernelBlock k = addKernel(
            new HostLoopbackKernel(makeKernelParameters("HostLoopbackKernel")));

        DFELink x = addStreamFromCPU("x");

        k.getInput("x") <== x;

        DFELink y = addStreamToCPU("y");
        y <== k.getOutput("y");
    }

    static EngineInterface interfaceDefault() {
        EngineInterface ei = new EngineInterface();
        InterfaceParam size = ei.addParam("dataSize", CPUTypes.INT32);
        ei.setTicks("HostLoopbackKernel", size);
        ei.setStream("x", CPUTypes.UINT32, size * CPUTypes.UINT32.sizeInBytes());
        ei.setStream("y", CPUTypes.UINT32, size * CPUTypes.UINT32.sizeInBytes());
        return ei;
    }

    public static void main(String[] args) {
        EngineParameters params = new EngineParameters(args);
        HostLoopbackManager manager = new HostLoopbackManager(params);
        manager.createSLiCinterface(interfaceDefault());
        manager.build();
    }
}
```

All Manager Compiler designs inherit from the `CustomManager` class:

```
class HostLoopbackManager extends CustomManager {
```

The Manager design is built when the constructor to this class is called, so we can place our simple Manager design code within the constructor:

```
    HostLoopbackManager(EngineParameters engineParameters) {
```

We call the super-class constructor with the `engineParameters` object, which implicitly specifies things like the DFE model and whether the target is a simulation model or a DFE `.max` file:

```
        super(engineParameters);
```

The first block in our Manager design is our Kernel, which we create using a call to `addKernel` with a new instance of our Kernel:

```
        KernelBlock k = addKernel(
            new HostLoopbackKernel(makeKernelParameters("HostLoopbackKernel")));
```

This method returns a `KernelBlock` object that we can query for its input and output streams. We create a stream from the CPU using the `addStreamFromCPU` method:

```
        DFELink x = addStreamFromCPU("x");
```

Streams are connected together using the connect operator (<==), which connects the input of a stream the output of another stream. In our case, we connect the input “x” of our Kernel to the output of a stream from the CPU that we are also calling “x”:

```
        k.getInput("x") <== x;
```

Note that the string names for streams inside the Kernel do not clash with names in the Manager and are not directly accessible from the CPU, so we can use the same name for streams connected to Kernel inputs and outputs for convenience.

The build process for a Manager is similar to the Standard Manager.
We create an instance of our Manager design parameter object:

```
        EngineParameters params = new EngineParameters(args);
```

We then create an instance of our Manager Design:

```
        HostLoopbackManager manager = new HostLoopbackManager(params);
```

The default engine interface for the SLiC interface is overridden:

```
        manager.createSLiCinterface(interfaceDefault());
```

Finally, we call the `build` method on our Manager, which starts the build process:

```
        manager.build();
```

The build process for a Manager constructs a graph of nodes for the blocks, connected together by edges for the streams. The Manager Compiler produces visual representations of the graph in the same way as the Kernel Compiler.

To generate the graph outputs, we can run `maxRenderGraphs`. This tool takes the directory of the build output and the name of the Manager with `Manager_` prepended as arguments. For example, for our simple example, we can run `maxRenderGraphs` from the build directory:

```
[user@machine HostLoopback]$ maxRenderGraphs ./ Manager_HostLoopback
```

This generates us an output diagram in our `scratch` sub-directory displaying the Manager after compilation. The graph for our example is shown in *Figure 2*.

We can see from the graph that the Manager Compiler has inserted a number of different components into the graph. These include:

- Buffers where necessary, including between blocks that are running at different clock rates.
- Dual-aspect multiplexers, which take wider data streams and multiplex them into narrower data streams (represented by arrows pointing downwards).
- Dual-aspect registers, which take narrower streams and adapt them into wider data streams (represented by arrows pointing upwards).

In our design, we can see that a dual-aspect multiplexer and register have been inserted by the Manager Compiler to adapt our 32-bit Kernel data streams to the 64-bit PCI Express data streams.

This information illustrates the process that the Manager Compiler goes through: the key from a user's point of view is that we can describe our Manager in terms of blocks and streams in our design and the Manager Compiler takes care of the process of implementing a correct circuit that conforms to our specification.

1.2.1 CPU Code

The CPU code for designs using the Manager Compiler uses the same API as for designs using the Standard Manager. The source for our example is shown in *Listing 3*.

The streams in the Manager design are set using the Basic Static SLiC Interface:

```
    HostLoopback(dataSize, dataIn, dataOut);
```

1.3 CPU IO

Multiple independent streams of data can be passed between the DFE and CPU and are automatically multiplexed and demultiplexed by MaxelerOS.

Streams to and from the CPU are created using methods provided by `CustomManager`:

*Stream addStreamToCPU(**String** name)*
*Stream addStreamFromCPU(**String** name)*

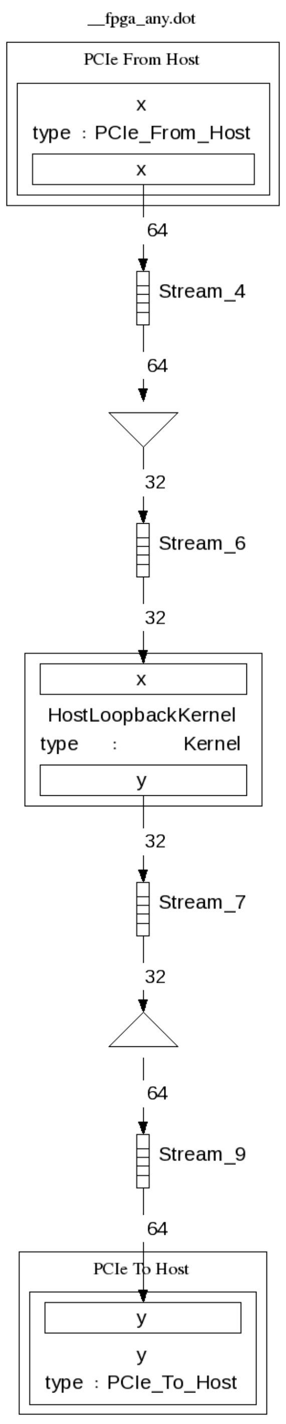


Figure 2: Output graph for the HostLoopback Manager

The total number of streams to and from the CPU can be no more than 8 in each direction on the MAX2, Vectis and Coria DFE models.

Listing 3: CPU code for our simple example (HostLoopbackCpuCode.c).

```
/**
 * Document: Manager Compiler Tutorial (maxcompiler-manager-tutorial.pdf)
 * Chapter: 1        Example: 1       Name: Host loopback
 * MaxFile name: HostLoopback
 * Summary:
 *       Streams integers to the dataflow engine and confirms that in the returned
 *       stream each integer has been incremented.
 */
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <MaxSLiCInterface.h>
#include "Maxfiles.h"

int check(int dataSize, uint32_t *dataOut, uint32_t *expectedOut)
{
     int status = 0;
     for (int i = 0; i < dataSize; i++)
          if (dataOut[i] != expectedOut[i]) {
               fprintf (stderr, "Output data @ %d = %d (expected %d)\n",
                    i, dataOut[i], expectedOut[i]);
               status = 1;
          }
     return status;
}

void HostLoopbackCPU(int size, uint32_t *dataIn, uint32_t *dataOut)
{
     for (int i=0; i<size; i++) {
          dataOut[i] = dataIn[i] + 1;
     }
}

int main()
{
     const int dataSize =1024;
     size_t sizeBytes = dataSize * sizeof(uint32_t);
     uint32_t *dataIn = malloc(sizeBytes);
     uint32_t *dataOut = malloc(sizeBytes);
     uint32_t *expectedOut = malloc(sizeBytes);

     for (int i = 0; i < dataSize; i++) {
          dataIn[i] = i + 1;
          dataOut[i] = 0;
     }

     HostLoopbackCPU(dataSize, dataIn, expectedOut);

     printf ("Running DFE.\n");
     HostLoopback(dataSize, dataIn, dataOut);

     int status = check(dataSize, dataOut, expectedOut);
     if (status)
          printf ("Test failed.\n");
     else
          printf ("Test passed OK!\n");

     return status;

}
```

Listing 4: Kernel for our dual-kernel example (DualKernelKernel.maxj).

```
/**
 * Document: Manager Compiler Tutorial (maxcompiler-manager-tutorial.pdf)
 * Chapter: 1       Example: 2       Name: Dual Kernel
 * MaxFile name: DualKernel
 * Summary:
 *        A kernel that increments a stream when a scalar input is set to 1
 *        and passes data straight back out again otherwise.
 */
package dualkernel;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;

class DualKernelKernel extends Kernel {

    DualKernelKernel(KernelParameters parameters) {
        super(parameters);

        DFEVar x = io.input("x", dfeUInt(32));
        DFEVar doAdd = io.scalarInput("doAdd", dfeBool());

        DFEVar y = doAdd ? x + 1 : x;

        io.output("y", y, dfeUInt(32));
    }
}
```

1.4 Multiple Kernels

In all the examples in the tutorial so far, we have implemented all of our processing in a single Kernel. We can instantiate multiple Kernels or multiple instances of the same Kernel within a Manager Compiler design.

The synchronous nature of streams in a Kernel can make applications where the processing rates of streams are different tricky to implement, requiring complex conditional stream control that renders the Kernel very convoluted. Such processing can be split into two or more Kernels, where the streams within each Kernel are synchronous but the Kernels run asynchronously from each other.

In Example 2, we show a single Kernel being instantiated twice. The source code for the Kernel is shown in *Listing 4*.

The Kernel itself is very simple. We have a scalar input that selects whether or not we pass the input stream "x" directly to the output stream "y" or add 1 before passing to to the output:

```
        DFEVar x = io.input("x", dfeUInt(32));
        DFEVar doAdd = io.scalarInput("doAdd", dfeBool());

        DFEVar y = doAdd ? x + 1 : x;

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

The Manager design is shown in *Listing 5*. We create two instances of our Kernel, giving each one a unique name:

```
        KernelBlock k1 = addKernel(
            new DualKernelKernel(makeKernelParameters("DualKernelKernel1")));
        KernelBlock k2 = addKernel(
            new DualKernelKernel(makeKernelParameters("DualKernelKernel2")));
```

We then connect up the input “x” streams for both Kernels, giving unique names for the streams in the Manager:

```
        DFELink x1 = addStreamFromCPU("x1");
        DFELink x2 = addStreamFromCPU("x2");
        k1.getInput("x") <== x1;
        k2.getInput("x") <== x2;
```

Followed by the “y” output streams:

```
        DFELink y1 = addStreamToCPU("y1");
        DFELink y2 = addStreamToCPU("y2");
        y1 <== k1.getOutput("y");
        y2 <== k2.getOutput("y");
```

In the CPU code, the single Basic Static function in the SLiC interface runs both sets of streams and our two Kernels, and sets all of the scalar inputs:

```
    DualKernel(dataSize, 1, 0, dataIn, dataIn, dataOut1, dataOut2);
```

Listing 5: Manager design for our dual-kernel example (DualKernelManager.maxj).

```
/**
 * Document: Manager Compiler Tutorial (maxcompiler-manager-tutorial.pdf)
 * Chapter: 1      Example: 2      Name: Dual Kernel
 * MaxFile name: DualKernel
 * Summary:
 *      Stream data from the CPU to two kernels and back to the CPU
 *      again from each.
 */
package dualkernel;

import com.maxeler.maxcompiler.v2.build.EngineParameters;
import com.maxeler.maxcompiler.v2.managers.custom.CustomManager;
import com.maxeler.maxcompiler.v2.managers.custom.DFELink;
import com.maxeler.maxcompiler.v2.managers.custom.blocks.KernelBlock;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.CPUTypes;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.EngineInterface;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.InterfaceParam;

class DualKernelManager extends CustomManager {

    DualKernelManager(EngineParameters engineParameters) {
        super(engineParameters);
        KernelBlock k1 = addKernel(
            new DualKernelKernel(makeKernelParameters("DualKernelKernel1")));
        KernelBlock k2 = addKernel(
            new DualKernelKernel(makeKernelParameters("DualKernelKernel2")));

        DFELink x1 = addStreamFromCPU("x1");
        DFELink x2 = addStreamFromCPU("x2");
        k1.getInput("x") <== x1;
        k2.getInput("x") <== x2;

        DFELink y1 = addStreamToCPU("y1");
        DFELink y2 = addStreamToCPU("y2");
        y1 <== k1.getOutput("y");
        y2 <== k2.getOutput("y");
    }

    static EngineInterface interfaceDefault() {
        EngineInterface ei = new EngineInterface();
        InterfaceParam size = ei.addParam("dataSize", CPUTypes.INT32);
        ei.setTicks("DualKernelKernel1", size);
        ei.setTicks("DualKernelKernel2", size);
        ei.setStream("x1", CPUTypes.UINT32, size * CPUTypes.UINT32.sizeInBytes());
        ei.setStream("x2", CPUTypes.UINT32, size * CPUTypes.UINT32.sizeInBytes());
        ei.setStream("y1", CPUTypes.UINT32, size * CPUTypes.UINT32.sizeInBytes());
        ei.setStream("y2", CPUTypes.UINT32, size * CPUTypes.UINT32.sizeInBytes());
        return ei;
    }

    public static void main(String[] args) {
        EngineParameters params = new EngineParameters(args);
        DualKernelManager manager = new DualKernelManager(params);
        manager.createSLiCinterface(interfaceDefault());
        manager.build();
    }
}
```

1.5 Routing Blocks

The Manager Compiler provides 3 different routing blocks for splitting and joining streams:

- Fanout - duplicates a single input data stream to zero or more of multiple output streams, selectable dynamically via a scalar input.
- Demultiplexer - duplicates a single input stream to one of multiple output streams, selectable dynamically via a scalar input.
- Multiplexer - duplicates one of multiple input streams, selectable dynamically via a scalar input, to a single output stream.

Note that multiplexing and demultiplexing streams used within a Kernel is more efficient when done in the Kernel itself rather than in the Manager due to the flow-control overhead in the Manager.

Fanout, multiplexer and demultiplexer input/output selection scalar inputs must be set in the CPU code: if not, the behavior is undefined.

1.5.1 Fanout

A fanout block is instantiated using the `fanout` method, which returns a `Fanout` block:

*Fanout fanout(**String** name)*

We get the single input to a fanout block using the `getInput` method:

Stream Fanout.getInput()

We can add any number of outputs to the fanout block by calling the `addOutput` method, which returns a new `Stream` object for each call:

*Stream Fanout.addOutput(**String** name)*

1.5.2 Demultiplexer

A demultiplexer block is instantiated using the `demux` method, which returns a `Demux` block:

*Demux demux(**String** name)*

We get the single input to a demultiplexer block using the `getInput` method:

Stream Demux.getInput()

We can add any number of outputs to the demultiplexer block by calling the `addOutput` method, which returns a new `Stream` object for each call:

*Stream Demux.addOutput(**String** name)*

1.5.3 Multiplexer

A multiplexer block is instantiated using the `mux` method, which returns a `Mux` block:

*Mux mux(**String** name)*

We get the single output from a multiplexer block using the `getOutput` method:

Stream Mux.getOutput()

We can add any number of inputs to the multiplexer block by calling the `addInput` method, which returns a new `Stream` object for each call:

*Stream Mux.addInput(**String** name)*

1.5.4 Setting routing blocks

Routing blocks can be set either in an engine interface or the CPU code.
In an engine interface, the `route` method is used:

***public void** route(**String** routeString)*

In the CPU code, the same string argument is added to the SLiC Interface, and is also available through the Advanced Dynamic SLiC Interface:

***void** max_route_string(max_actions_t ∗actions, **const char** ∗route_string);*

In all cases, the string argument contains a comma-separated list of `"from -> to"` pairs, where `"from"`is the name of the routing block, and `"to"` is the name of the input or output port. The name of the `"from"` or `"to"` entity may optionally be prepended by the name of the block to which it belongs, separated by a period.
As an example, for a design containing a demultiplexer called `"split"` and a multiplexer called `"join"`, the `routeString` might be `"split -> x1, join.y1 -> join.join"`. Spaces are not significant, so may be used freely to aid readability.
Alternatively, each connecton can be set separately using `max_route`:

***void** max_route(max_actions_t ∗actions, **const char** ∗from_name, **const char** ∗to_name);*

For multiplexers and demultiplexers, any connection supercedes and overwrites any existing connection, since a multiplexer and demultiplexer can have only one connection. For fanouts, any connection will add another output connection to the fanout, since a fanout can have zero or more connections. In order to remove all fanout connections, the symbol `"NULL"` is used, e.g. `"my_fanout -> NULL"`. The NULL form is not valid for multiplexers or demultiplexers.

1.5.5 Demultiplexer/Multiplexer Example

Example 3 demonstrates the use of a demultiplexer and a multiplexer in a simple design. The Kernel is the same as the Kernel for Example 2. The source code for the Manager is shown in *Listing 6*. *Figure 3* shows the initial Manager graph for this Manager design, where we can clearly see the demultiplexer splitting the input stream to the two Kernels and the multiplexer joining the two outputs together.

We have a single input stream, `x`:

```
		DFELink x = addStreamFromCPU("x");
```

We connect `x` to the input of a demultiplexer that we call “split”:

```
		Demux split = demux("split");
		split.getInput() <== x;
```

Listing 6: Manager design demonstrating use of a demultiplexer and a multiplexer (SplitStreamManager.maxj).

```
package splitstream;

import com.maxeler.maxcompiler.v2.build.EngineParameters;
import com.maxeler.maxcompiler.v2.managers.custom.CustomManager;
import com.maxeler.maxcompiler.v2.managers.custom.DFELink;
import com.maxeler.maxcompiler.v2.managers.custom.blocks.Demux;
import com.maxeler.maxcompiler.v2.managers.custom.blocks.KernelBlock;
import com.maxeler.maxcompiler.v2.managers.custom.blocks.Mux;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.CPUTypes;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.EngineInterface;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.InterfaceParam;

class SplitStreamManager extends CustomManager {

    SplitStreamManager(EngineParameters engineParameters) {
        super(engineParameters);

        KernelBlock k1 = addKernel(
            new SplitStreamKernel(makeKernelParameters("SplitStreamKernel1")));
        KernelBlock k2 = addKernel(
            new SplitStreamKernel(makeKernelParameters("SplitStreamKernel2")));

        DFELink x = addStreamFromCPU("x");

        Demux split = demux("split");
        split.getInput() <== x;

        DFELink x1 = split.addOutput("x1");
        DFELink x2 = split.addOutput("x2");
        k1.getInput("x") <== x1;
        k2.getInput("x") <== x2;

        Mux join = mux("join");
        join.addInput("y1") <== k1.getOutput("y");
        join.addInput("y2") <== k2.getOutput("y");

        DFELink y = addStreamToCPU("y");
        y <== join.getOutput();
    }

    static EngineInterface interfaceDefault() {
        EngineInterface ei = new EngineInterface();
        InterfaceParam size = ei.addParam("dataSize", CPUTypes.INT32);
        ei.setTicks("SplitStreamKernel1", size);
        ei.setTicks("SplitStreamKernel2", size);
        ei.setStream("x", CPUTypes.UINT32, size * CPUTypes.UINT32.sizeInBytes());
        ei.setStream("y", CPUTypes.UINT32, size * CPUTypes.UINT32.sizeInBytes());
        return ei;
    }

    public static void main(String[] args) {
        EngineParameters engineParamenters = new EngineParameters(args);
        SplitStreamManager m = new SplitStreamManager(engineParamenters);
        m.createSLiCinterface(interfaceDefault());
        m.build();
    }
}
```

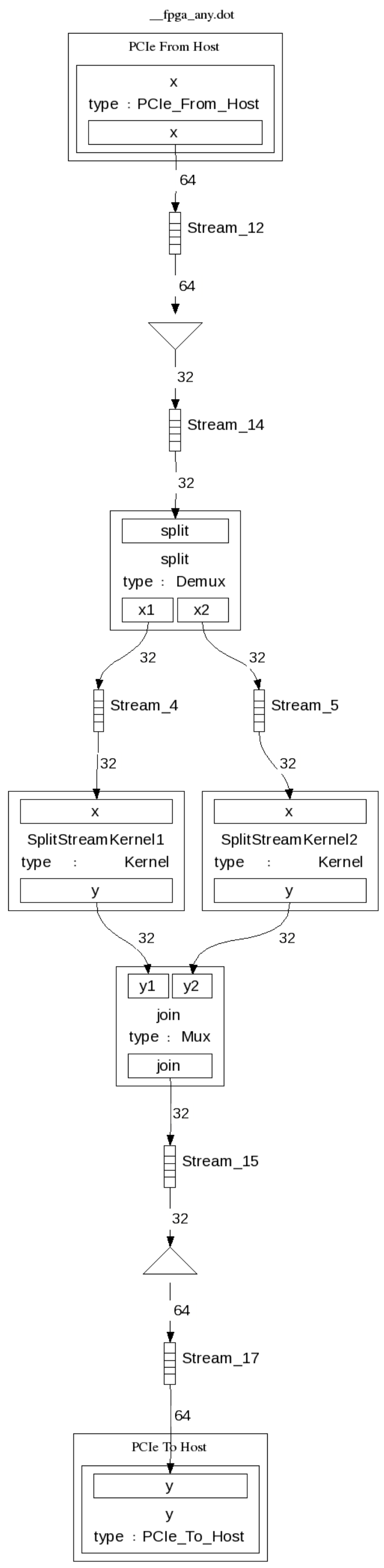


Figure 3: Original graph for the dual-Kernel Manager demonstrating use of a demultiplexer and a multiplexer (*Manager_SplitStream_orig*)

We add two output streams, “x1” and “x2” to our demultiplexer and connect these to the input streams to our two Kernels:

```
DFELink x1 = split.addOutput("x1");
DFELink x2 = split.addOutput("x2");
k1.getInput("x") <== x1;
k2.getInput("x") <== x2;
```

Similarly, we create a multiplexer called “join”, to which we add two inputs connected to the outputs from the two Kernels:

```
Mux join = mux("join");
join.addInput("y1") <== k1.getOutput("y");
join.addInput("y2") <== k2.getOutput("y");
```

Finally, we connect the output from the multiplexer to the output stream “y”:

```
DFELink y = addStreamToCPU("y");
y <== join.getOutput();
```

The main function from the CPU software for our example is shown in *Listing 7*. The scalar inputs, demultiplexer and multiplexer are all set in the Basic Static function call to run the design:

```
SplitStream(dataSize, 1, 0, dataIn, dataOut1, "x1 -> split, y1 -> join");
```

We then call the Basic Static SLiC function again with the demultiplexer and multiplexer set to enable the input and output to the second Kernel:

```
SplitStream(dataSize, 1, 0, dataIn, dataOut2, "x2 -> split, y2 -> join");
```

1.6 Clock Configuration

The `CustomManager` class inherited by all Manager Compiler designs offers further options, accessible via the `config` property. The default clock to use for all blocks in the Manager (unless explicitly set to an alternative clock source) can be set:

void *config.setDefaultStreamClockFrequency(**int** freq_mhz)*

This generates timing constraints and a clock source for this frequency.

Kernels can be connected to different clock sources, which is useful in a multi-Kernel design where some Kernels can achieve higher clock rates than others.

A clock source is generated using the method `generateStreamClock`, which takes a string name for the clock source and a clock rate in MHz:

public *ManagerClock generateStreamClock(**String** name, **int** freq_mhz)*

This generates both the clock source itself in the device and creates timing constraints for blocks connected to this clock. A Kernel block can be connected to the clock source using the `setClock` method on the `KernelBlock` class:

void *KernelBlock.setClock(ManagerClock clock)*

The appropriate clock sources for all other blocks in the Manager design are determined by the Manager Compiler.

Listing 7: CPU code demonstrating use of a demultiplexer and a multiplexer (SplitStreamCpuCode.c).

```
int main()
{
    int const dataSize= 1024;

    size_t sizeBytes = dataSize* sizeof(uint32_t);
    uint32_t *dataIn = malloc(sizeBytes);
    uint32_t *dataOut1 = malloc(sizeBytes);
    uint32_t *dataOut2 = malloc(sizeBytes);
    uint32_t *expected1 = malloc(sizeBytes);
    uint32_t *expected2 = malloc(sizeBytes);

    for (int i = 0; i < dataSize; i++) {
        dataIn[i] = i + 1;
        dataOut1[i] = 0;
        dataOut2[i] = 0;
    }

    SplitStreamCPU(dataSize, 1, dataIn, expected1);
    SplitStreamCPU(dataSize, 0, dataIn, expected2);

    printf("Running DFE.\n");
    SplitStream(dataSize, 1, 0, dataIn, dataOut1, "x1 -> split, y1 -> join");

    SplitStream(dataSize, 1, 0, dataIn, dataOut2, "x2 -> split, y2 -> join");

    int status = check(dataSize, 1, dataOut1, expected1);
    status |= check(dataSize, 2, dataOut2, expected2);
    if (status)
        printf("Test failed.\n");
    else
        printf("Test passed OK!\n");

    return status;
}
```

1.7 Debugging Options

The `CustomManager` class has more debugging options available than the Standard Manager.

A `DebugLevel` object object can be passed to an instance of the `CustomManager` class using the `setDebugLevel` method on its `debug` member, for example:

```
DebugLevel MyDebugLevel = new DebugLevel();
...
debug.setDebugLevel(MyDebugLevel);
```

1.7.1 Stream Status Blocks

Stream status blocks are introduced in Kernel Compiler Tutorial document.

Stream status blocks can be enabled globally in a similar fashion to the Standard Manager via a `DebugLevel` object:

```
void DebugLevel.setHasStreamStatus(true)
```

In addition to the stream status blocks themselves, stream status checksums can also be enabled. These keep a running checksum of the data passing through that stream. This information is displayed in MaxDebug. Stream status checksums can be enabled globally using `setHasStreamStatusChecksums`:

__void__ DebugLevel.setHasStreamStatusChecksums(__boolean__ has_stream_status_checksums)

Enabling stream status checksums globally without enabling stream status blocks has no effect.

Alternatively, stream status blocks and checksums can be enabled on a per-stream basis via the method `streamStatus` on the `debug` object:

Stream debug.streamStatus(Stream to_probe, __boolean__ checksum)

If only one or a small number of streams are of interest, then enabling stream status blocks and checksums on a per-stream basis can save the extra logic used by these blocks when enabled on every stream.

2 LMem

MaxCards have a large external LMem resource available. This allows large amounts of data to be kept local to the board and iterated over by the DFE. The LMem appears as one contiguous piece of memory. Both the Standard Manager and the Manager Compiler allow you to connect multiple streams to the LMem on the MaxCard.

2.1 Memory Controller Architecture

The DFE component of MaxelerOS has a memory controller that provides a command-based interface to the LMem. A stream to or from LMem in a Manager has a **command queue** and a **data buffer**. The memory controller reads the commands and either reads data from the data buffer and writes it to the appropriate location in memory or reads data from the appropriate location in memory and writes it into the data buffer.

Figure 4 shows the architecture of the memory controller as used in Example 1. In this case, we have an example Kernel with two inputs and one output, both connected to the LMem via Manager streams. We also have two memory streams that are connected to the CPU to give it direct access to the LMem over PCI Express. The five streams each have their own memory command generator, command queue and data buffer. The **memory command generator** generates commands instructing the memory controller to read or write a specified amount of data to or from a specified location in the LMem, and places these commands in the command queue for the stream. The **memory controller** reads commands from the queues, and, if the data buffer for the referenced stream is ready (not full for a read stream, not empty for a write stream), executes the memory operations.

✷ Connecting more streams to the LMem creates more logic and can make it harder to meet timing constraints: consider multiplexing streams to and from LMem where streams are not active at the same time.

✷ A maximum of 15 streams can be connected to LMem.

2.2 Memory Command Generators

In the Standard Manager, each stream has its own memory command generator. A Manager Compiler design may have separate memory command generators for each stream or several streams in a memory control group (see *subsection 2.4*) may share a command generator.

There are different access patterns available for the memory:

- `LMEM_LINEAR_1D` addresses the LMem with a simple linear address.
- `LMEM_BLOCKED_3D` addresses the LMem with a 3D address.
- `LMEM_STRIDE_2D` addresses the LMem either linearly or with a “stride”, to address data at a fixed interval through the LMem.

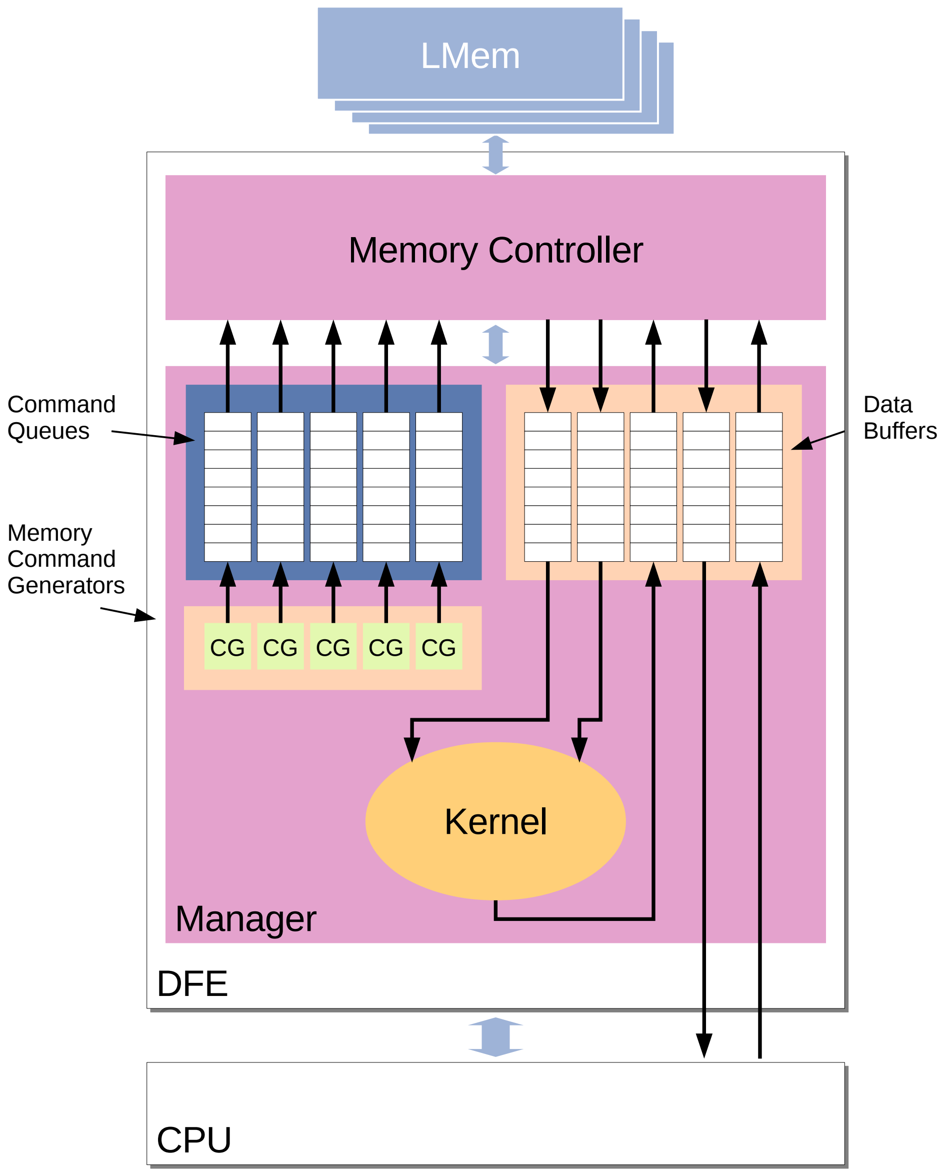


Figure 4: LMem controller architecture

By default, all of the arguments for memory address generators appear in the Basic Static and Advanced Static SLiC Interface. Alternatively, the parameters can be set up either in an engine interface or through the Advanced Dynamic SLiC Interface.

The SLiC Interface automatically enables interrupts for all memory command stream generators, which are raised once all of the data has been read from or written to the LMem. The blocking API calls will return when all the interrupts have been received. When using the non-blocking API calls, `max_wait` will similarly return when all the interrupts have been received.

2.2.1 Burst Size

The memory controller and its memory command generators work in *bursts*. The burst size, in bytes, can be retrieved through the Advanced Dynamic SLiC interface:

```
int max_get_burst_size(max_file_t *maxfile, const char *mem_ctrl_name);
```

The `mem_ctrl_name` argument should always be set to `NULL`: it is there to allow more than one memory controller to be used in future architectures.

The burst size is dependent on the particular DFE being used.

> ✷ All dimensions provided as arguments to SLiC LMem functions are in *bytes* and must be multiples of the *burst size* for the target DFE.

2.2.2 Argument Types

The methods for setting entities in an engine interface often have more than one variation: one method that uses `InterfaceParam` arguments, another that uses `long`, and other combinations of these.

For some LMem methods, the greater number of arguments means that there would be a large number of variations needed to cover all combinations of `InterfaceParam` and `long`. Therefore, the LMem methods have been restricted to provide a version where all the arguments are `InterfaceParam` instances and a version where all the parameters are `long`s. To help using this API with a combination of `long` and `InterfaceParam` arguments, a method is provided to allow an `InterfaceParam` to be created from a `long` or `double` Java variable:

```
InterfaceParam EngineInterface.addConstant(long value)
InterfaceParam EngineInterface.addConstant(double value)
```

2.2.3 Linear Access Pattern

A simple linear access pattern is set up using two integer arguments, `address` and `size`, to address a block of LMem:

```
void max_lmem_linear( max_actions_t *actions, const char *mem_stream_name, int64_t address, int64_t size )
```

And in an engine interface:

```
EngineInterface.setLMemLinear(String streamName, InterfaceParam address, InterfaceParam size);
EngineInterface.setLMemLinear(String streamName, long address, long size);
```

`address` represents the base location in bytes within the block of LMem.

`size` represents the total length, in bytes, of the block of LMem being addressed, and is measured from `address`.

This mode reads `size` bytes from `address`, stopping at the end of the LMem if `address+size > (lmem_size)`.

A more general linear access pattern can be set up using four integer arguments:

```
void max_lmem_linear_advanced( max_actions_t *actions, const char *mem_stream_name, int64_t address, int64_t array_size, int64_t
        rw_size, int64_t offset )
```

And in an engine interface:

```
EngineInterface.setLMemLinearWrapped(String streamName, InterfaceParam address,
                    InterfaceParam arrSize, InterfaceParam rwSize, InterfaceParam offset);
EngineInterface.setLMemLinearWrapped(String streamName, long address,
                    long arrSize,  long rwSize, long offset);
```

`address` again represents the base location in bytes within the block of LMem.

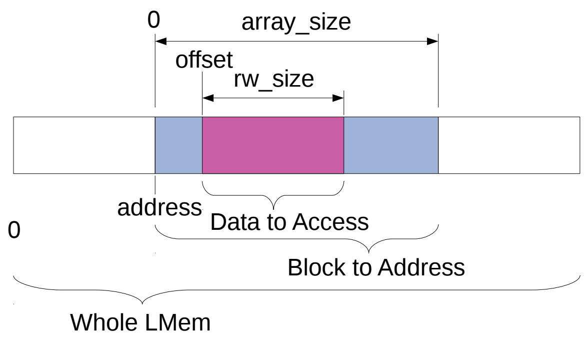
`rw_size`/`rwSize` represents the length, in bytes, of the portion of block of LMem that is to be read or written, and is measured from `offset`.

`array_size`/`arrSize` represents the total length, in bytes, of the block of LMem being addressed, and is measured from `address`.

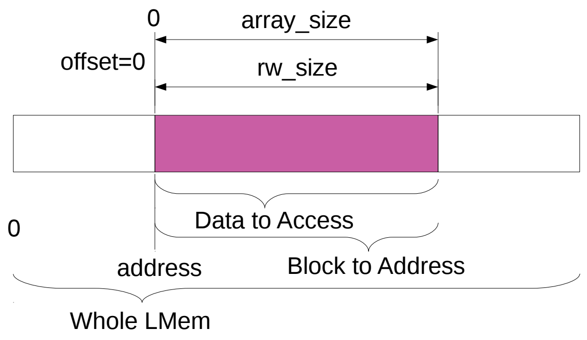
`offset` represents the offset, in bytes, from `address` from which the read or write is to begin.

The simple linear call, `max_lmem_linear`, is a synonym for the advanced call, `max_lmem_linear_advanced`, with `rw_size = array_size` and `offset = 0`; this is illustrated in the second example in *Figure 5b*.

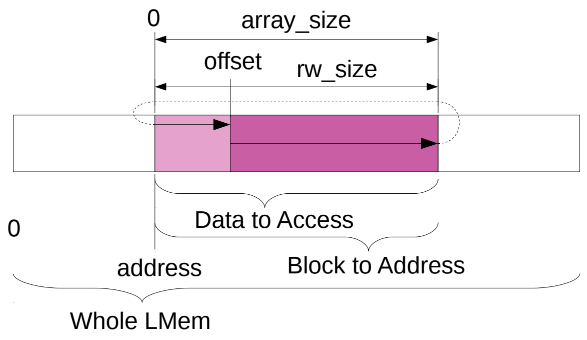
Figure 5 shows various different combinations of `address`, `array_size`, `rw_size` and `offset` used to address a block of LMem.



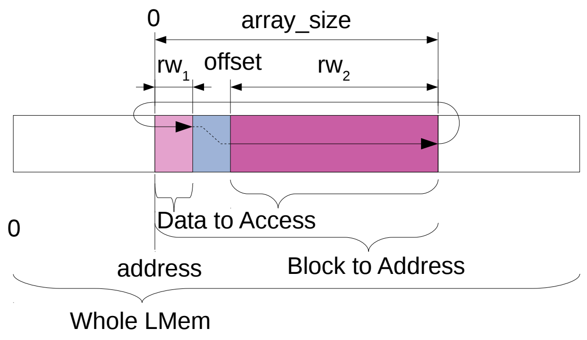
(a) `offset > 0`, `rw_size < array_size`, addressing part of the block of LMem.



(b) `offset = 0`, `rw_size = array_size`, addressing the whole block of LMem.



(c) `offset > 0`, `rw_size = array_size`, addressing the whole block of LMem, starting with an offset of `offset`.



(d) `offset > 0`, `rw_size (rw`$_1$` + rw`$_2$`) < array_size`, addressing part of the block of LMem, starting with an offset of `offset` and skipping the section from `rw_size`$_1$ to `offset`.

Figure 5: Linear memory command generator with different configurations of `address`, `rw_size`, `offset` and `array_size`.

2.2.4 3D Blocking Memory Access Pattern

A 3D Blocking memory command generator operates in a coordinate system where the unit of size in each dimension is in *bytes*, where, as before, the number of byte must be a multiple of the burst size.

A block of size *(rwSizeFast, rwSizeMed, rwSizeSlow)*, with its origin at *(offsetSizeFast, offsetSizeMed, offsetSizeSlow)* is read from a larger block of size *(arraySizeFast, arraySizeMed, arraySizeSlow)* from a start-location at *address*:

```
EngineInterface.setLMemBlocked(String streamName, InterfaceParam address,
                        InterfaceParam arraySizeFast, InterfaceParam arraySizeMed, InterfaceParam arraySizeSlow,
                        InterfaceParam rwSizeFast,    InterfaceParam rwSizeMed,   InterfaceParam rwSizeSlow,
                        InterfaceParam offsetFast,    InterfaceParam offsetMed,   InterfaceParam offsetSlow );
EngineInterface.setLMemBlocked(String streamName, long address,
                        long arraySizeFast, long arraySizeMed, long arraySizeSlow,
                        long rwSizeFast,    long rwSizeMed,   long rwSizeSlow,
                        long offsetFast,    long offsetMed,   long offsetSlow );
```

And via the Advanced Dynamic SLiC Interface:

```
void max_lmem_blocked( max_actions_t *actions, const char *mem_stream_name, int64_t address,
              int64_t   array_size_fast , int  array_size_med, int  array_size_slow,
              int64_t   rw_size_fast ,  int  rw_size_med, int  rw_size_slow,
              int64_t   offset_fast ,  int  offset_med, int  offset_slow );
```

Figure 6 shows the meaning of the arguments in 3D space.

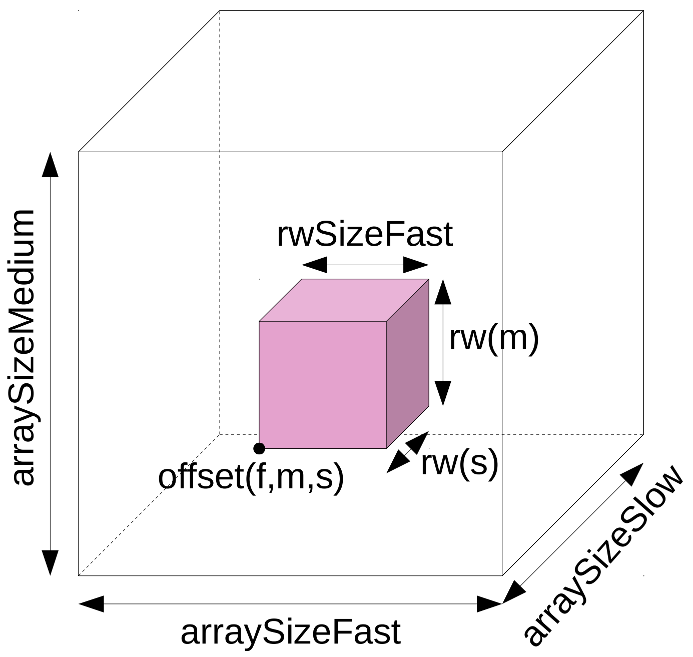


Figure 6: 3D blocking memory command generator

In practice, the 3D block is set into the LMem with a linear mapping, so that the element, of size *B* bytes, and with index *(x, y, z)* will be written to the location, *loc*, in the LMem given by:

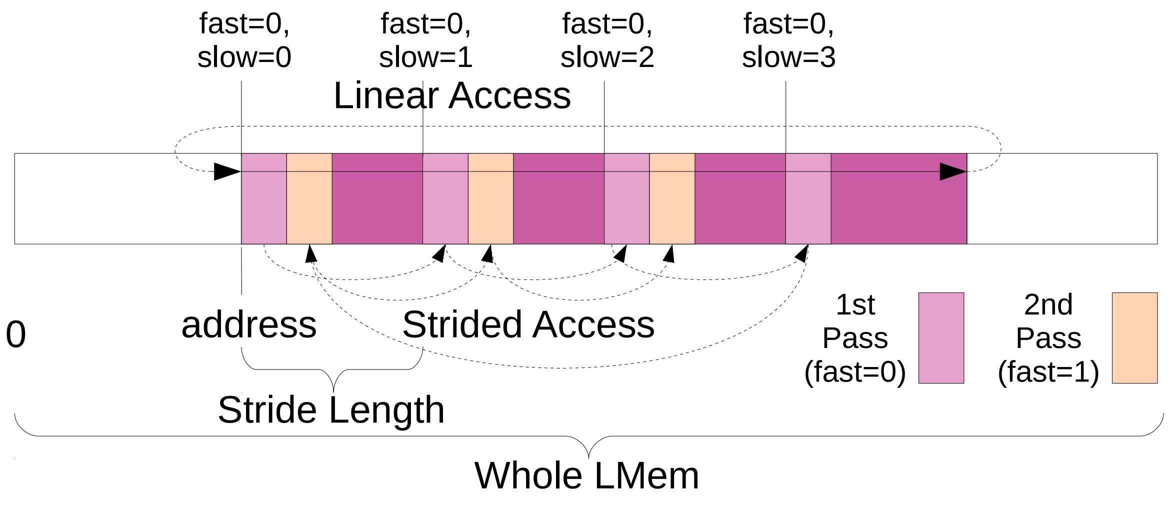
$loc = address + B * (x + arraySizeFast * (y + arraySizeMed * z))$

The dimension arguments should hence be seen as *multiplier*-type arguments; from this it follows that, provided the fast dimension arguments are multiples of the burst size, the medium and slow dimensions must also be, since they are multiplied by the fast dimension.

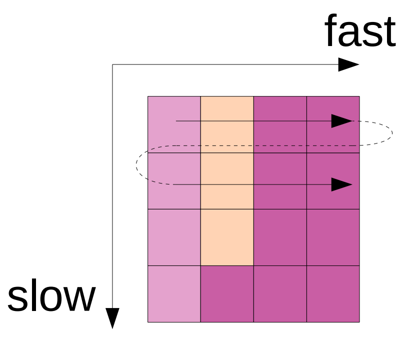
2.2.5 2D Strided Memory Access Pattern

The 2D Strided memory command generator allows both linear access and **strided** access, where the data is accessed efficiently at fixed intervals in the LMem.

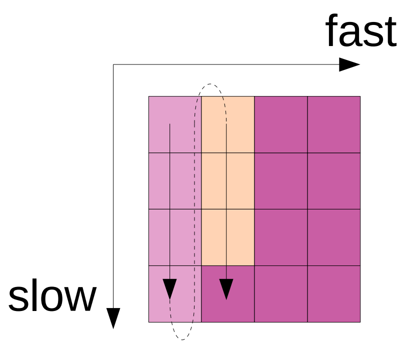
Figure 7 demonstrates the principle of strided access. *Figure 7a* shows a contiguous section of the LMem in which 2D data has been stored being addressed both linearly and with a stride. The linear access reads sequentially in the x dimension, as shown in *Figure 7b*. The strided access reads sequentially in the y dimension, as shown in *Figure 7c*.



(a)



(b) Linear Access

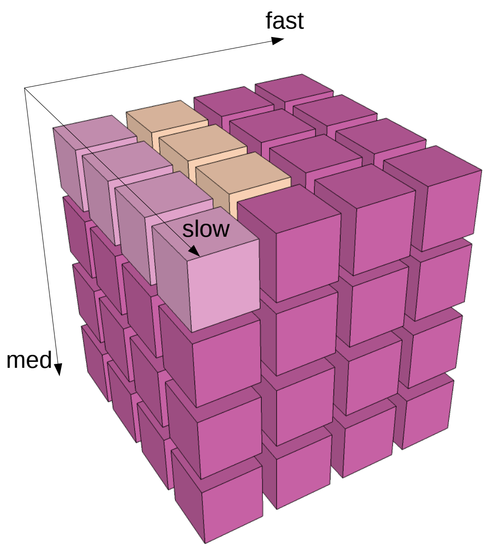


(c) Strided Access

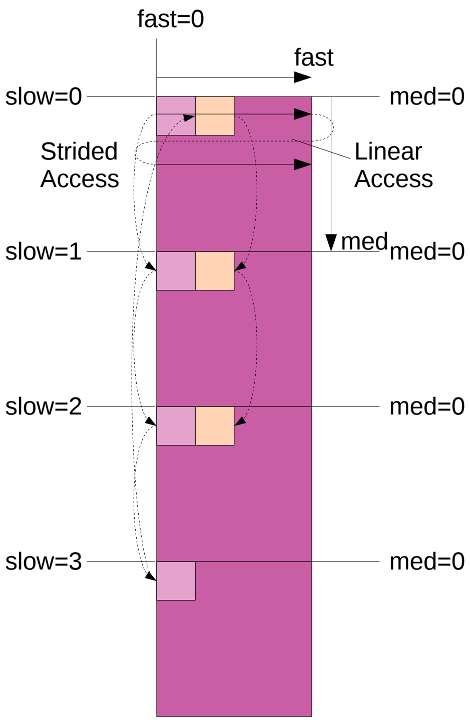
Figure 7: 2D strided memory access

> Although a strided access pattern could be expressed using single bursts in a linear access pattern, using the 2D strided access pattern is much more efficient.

2D Strided access is very useful for addressing 3D data stored in LMem, as shown in *Figure 8*. *Figure 8a* shows a 3D volume of data to be stored in memory. *Figure 8b* shows the data laid out in memory in 2D slices. The linear access pattern address the data in rows in each 2D slice. The strided access pattern addresses the data linearly in the z dimension.



(a) 3D volume



(b) Strided access

Figure 8: 3D volume stored in LMem and accessed in z dimension using 2D strided access pattern

The 2D Strided access pattern is set in the engine interface using two dimensions and a `strideMode`:

```
EngineInterface.setLMemStrided(String streamName, InterfaceParam address,
                    InterfaceParam sizeFast, InterfaceParam sizeSlow, InterfaceParam strideMode);
EngineInterface.setLMemStrided(String streamName, long address,
                    long sizeFast, long sizeSlow, long strideMode);
```

The `strideMode` argument must be set to 0 for *linear* access, or to 1 for *strided* access. `sizeFast` and `sizeSlow` are the dimensions of the 3D data stored in memory.

> The dimensions of all arguments to `setLMemStrided` are in *bytes*, which must be multiples of the burst size.

The Advanced Dynamic SLiC Interface provides simple and advanced functions for using the 2D Strided memory pattern:

```
void max_lmem_strided( max_actions_t *actions, const char *mem_stream_name, int64_t address,
                       int64_t  size_fast ,  int  size_slow,  int  stride_mode)
void max_lmem_strided_advanced( max_actions_t *actions, const char *mem_stream_name, int64_t address,
                       int64_t  size_fast ,  int  size_med, int  size_slow,
                       int64_t  offset ,  int64_t  stride ,  int  stride_mode)
```

The dimensions of all arguments to `max_lmem_strided` and `max_lmem_strided_advanced` are in *bytes*, which must be multiples of the burst size.

`stride` is the size of the stride. `stride` must be greater than $size\_fast * size\_slow$ and less than 65536 (2^{16}). Any value over 127 is rounded to the next highest multiple of 512.

`stride_mode` sets whether to access the memory linearly (`stride_mode = 0`) or strided (`stride_mode = 1`).

`offset` is the offset *in bursts* into the memory block to be addressed by the 2D Strided memory command generator from which the command generator starts. This is not the same `address`, which is the offset into the whole LMem for the memory block to be addressed by the 2D Strided memory command generator. After the first pass through the LMem, the memory command generator will loop back to the start of the block (as specified by `address`). If `offset` is set to `0`, then the address generator will start from the beginning of the block (as specified by `address`).

For optimal performance using 2D Strided, the stride should be set to a multiple of 16384 (2^{14}). If $size\_fast * size\_slow$ is not a multiple of 16384, then the data should be padded to this boundary. In *Figure 9*, the purple area represents the valid data and the gray area represents the padding to the 16384 boundaries. Note that this gives optimal throughput from the RAM but wastes some space in the LMem.

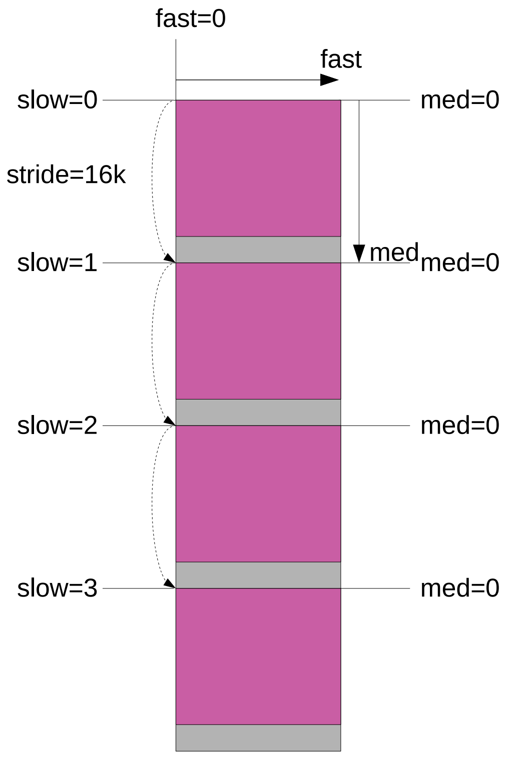


Figure 9: Optimal 2D strided access showing padding to 16K boundaries

Listing 8: Manager demonstrating use of LMem (LMemExampleManager.maxj).

```
class LMemExampleManager extends CustomManager {

    private static final String KERNEL_NAME = "LMemExampleKernel";
    private static final CPUTypes TYPE = CPUTypes.INT32;

    LMemExampleManager(EngineParameters engineParameters) {
        super(engineParameters);
        KernelBlock k = addKernel(new LMemExampleKernel(makeKernelParameters(KERNEL_NAME)));

        DFELink cpu2lmem = addStreamToOnCardMemory("cpu2lmem", MemoryControlGroup.MemoryAccessPattern.LINEAR_1D);
        DFELink lmem2cpu = addStreamFromOnCardMemory("lmem2cpu", MemoryControlGroup.MemoryAccessPattern.LINEAR_1D)
            ;

        DFELink fromcpu = addStreamFromCPU("fromcpu");
        DFELink tocpu = addStreamToCPU("tocpu");

        cpu2lmem <== fromcpu;
        tocpu <== lmem2cpu;

        DFELink inA = addStreamFromOnCardMemory("inA", MemoryControlGroup.MemoryAccessPattern.LINEAR_1D);
        DFELink inB = addStreamFromOnCardMemory("inB", MemoryControlGroup.MemoryAccessPattern.LINEAR_1D);

        k.getInput("inA") <== inA;
        k.getInput("inB") <== inB;

        DFELink oData = addStreamToOnCardMemory("oData",MemoryControlGroup.MemoryAccessPattern.LINEAR_1D);
        oData <== k.getOutput("oData");
    }
```

2.3 Example 1: Simple Loopback

Example 1 creates a pair of arrays and writes each array to a separate location in the LMem on the DFE board. A Kernel running in the DFE reads the arrays from the LMem, computes their element-wise sum, and writes the result to a third location in the LMem. When this operation is complete, the CPU application reads the array of sums from the LMem, and verifies that they have been computed correctly. There are no streams directly connecting the Kernel to the CPU.

2.3.1 Kernel

The Kernel itself is very simple:

```
        DFEVar inA = io.input("inA", dfeUInt(32));
        DFEVar inB = io.input("inB", dfeUInt(32));

        io.output("oData", inA+inB, dfeUInt(32));
```

2.3.2 Manager

The Manager (shown in *Listing 8*) defines much of the flow of data. First the streams for reading and writing to the LMem from the CPU are created with the `LMEM_LINEAR_1D` pattern:

```
        DFELink cpu2lmem = addStreamToOnCardMemory("cpu2lmem", MemoryControlGroup.MemoryAccessPattern.LINEAR_1D);
        DFELink lmem2cpu = addStreamFromOnCardMemory("lmem2cpu", MemoryControlGroup.MemoryAccessPattern.LINEAR_1D)
            ;
```

These are then connected to streams to and from the CPU using the connect (<==) operator:

```
        DFELink fromcpu = addStreamFromCPU("fromcpu");
        DFELink tocpu = addStreamToCPU("tocpu");

        cpu2lmem <== fromcpu;
        tocpu <== lmem2cpu;
```

The streams from the LMem to the Kernel are created similarly:

```
        DFELink inA = addStreamFromOnCardMemory("inA", MemoryControlGroup.MemoryAccessPattern.LINEAR_1D);
        DFELink inB = addStreamFromOnCardMemory("inB", MemoryControlGroup.MemoryAccessPattern.LINEAR_1D);

        k.getInput("inA") <== inA;
        k.getInput("inB") <== inB;
```

Finally, the stream from the Kernel output to the LMem is created:

```
        DFELink oData = addStreamToOnCardMemory("oData",MemoryControlGroup.MemoryAccessPattern.LINEAR_1D);
        oData <== k.getOutput("oData");
```

2.3.3 Engine interfaces

The Manager has 3 engine interfaces, shown in *Listing 8*. There are interfaces for reading and writing directly to the LMem from the CPU and a default interface for running the Kernel.

In the default interface, the 2 streams into the Kernel and the stream out of the Kernel are set to access same size of data in the LMem linearly, but from different addresses:

```
    private static EngineInterface interfaceDefault () {
        EngineInterface ei = new EngineInterface();

        InterfaceParam N    = ei.addParam("N", TYPE);
        ei.setTicks(KERNEL_NAME, N);
        InterfaceParam sizeInBytes = N * TYPE.sizeInBytes();

        InterfaceParam zero = ei.addConstant(0l);
        ei.setLMemLinear("inA", zero, sizeInBytes);
        ei.setLMemLinear("inB", sizeInBytes, sizeInBytes);
        ei.setLMemLinear("oData", 2 * sizeInBytes, sizeInBytes);
        ei.ignoreAll(Direction.IN_OUT);
        return ei;
    }
```

2.3.4 CPU Code

The CPU code first transfers each input array to the LMem:

```
    LMemExample_writeLMem(size, 0, inA);
    LMemExample_writeLMem(size, size, inB);
```

The Kernel is then run using the Basic Static call for the default engine interface:

```
    LMemExample(size);
```

Listing 9: Manager demonstrating use of LMem (LMemExampleManager.maxj).

```
    private static EngineInterface interfaceWrite(String name) {
        EngineInterface ei = new EngineInterface(name);

        InterfaceParam size  = ei.addParam("size", TYPE);
        InterfaceParam start = ei.addParam("start", TYPE);
        InterfaceParam sizeInBytes = size * TYPE.sizeInBytes();

        ei.setStream("fromcpu", TYPE, sizeInBytes );
        ei.setLMemLinear("cpu2lmem", start * TYPE.sizeInBytes(), sizeInBytes);
        ei.ignoreAll(Direction.IN_OUT);
        return ei;
    }

    private static EngineInterface interfaceRead(String name) {
        EngineInterface ei = new EngineInterface(name);

        InterfaceParam size  = ei.addParam("size", TYPE);
        InterfaceParam start = ei.addParam("start", TYPE);
        InterfaceParam sizeInBytes = size * TYPE.sizeInBytes();

        ei.setLMemLinear("lmem2cpu", start * TYPE.sizeInBytes(), sizeInBytes);
        ei.setStream("tocpu", TYPE, sizeInBytes);
        ei.ignoreAll(Direction.IN_OUT);
        return ei;
    }

    private static EngineInterface interfaceDefault() {
        EngineInterface ei = new EngineInterface();

        InterfaceParam N    = ei.addParam("N", TYPE);
        ei.setTicks(KERNEL_NAME, N);
        InterfaceParam sizeInBytes = N * TYPE.sizeInBytes();

        InterfaceParam zero = ei.addConstant(0l);
        ei.setLMemLinear("inA", zero, sizeInBytes);
        ei.setLMemLinear("inB", sizeInBytes, sizeInBytes);
        ei.setLMemLinear("oData", 2 * sizeInBytes, sizeInBytes);
        ei.ignoreAll(Direction.IN_OUT);
        return ei;
    }

    public static void main(String[] args) {
        LMemExampleManager m =
            new LMemExampleManager(new EngineParameters(args));

        m.createSLiCinterface(interfaceWrite("writeLMem"));
        m.createSLiCinterface(interfaceRead("readLMem"));
        m.createSLiCinterface(interfaceDefault());

        m.build();
    }
}
```

Finally, the application retrieves the results from the LMem :

```
 int32_t *outData = malloc(sizeBytes);
LMemExample_readLMem(size, 2 * size, outData);
```

2.4 Memory Control Groups

Memory streams can be grouped into a memory control group. A memory control group can either be associated with a single memory command generator or a memory command stream generated by the user (see *subsection 2.6*).

A memory command generator associated with a memory control group generates commands for each stream in the group that have the same access pattern but different start positions. This mode of operation is termed "pantograph" mode as the identical access pattern, with the same settings, is transcribed to multiple memory streams at different offsets.

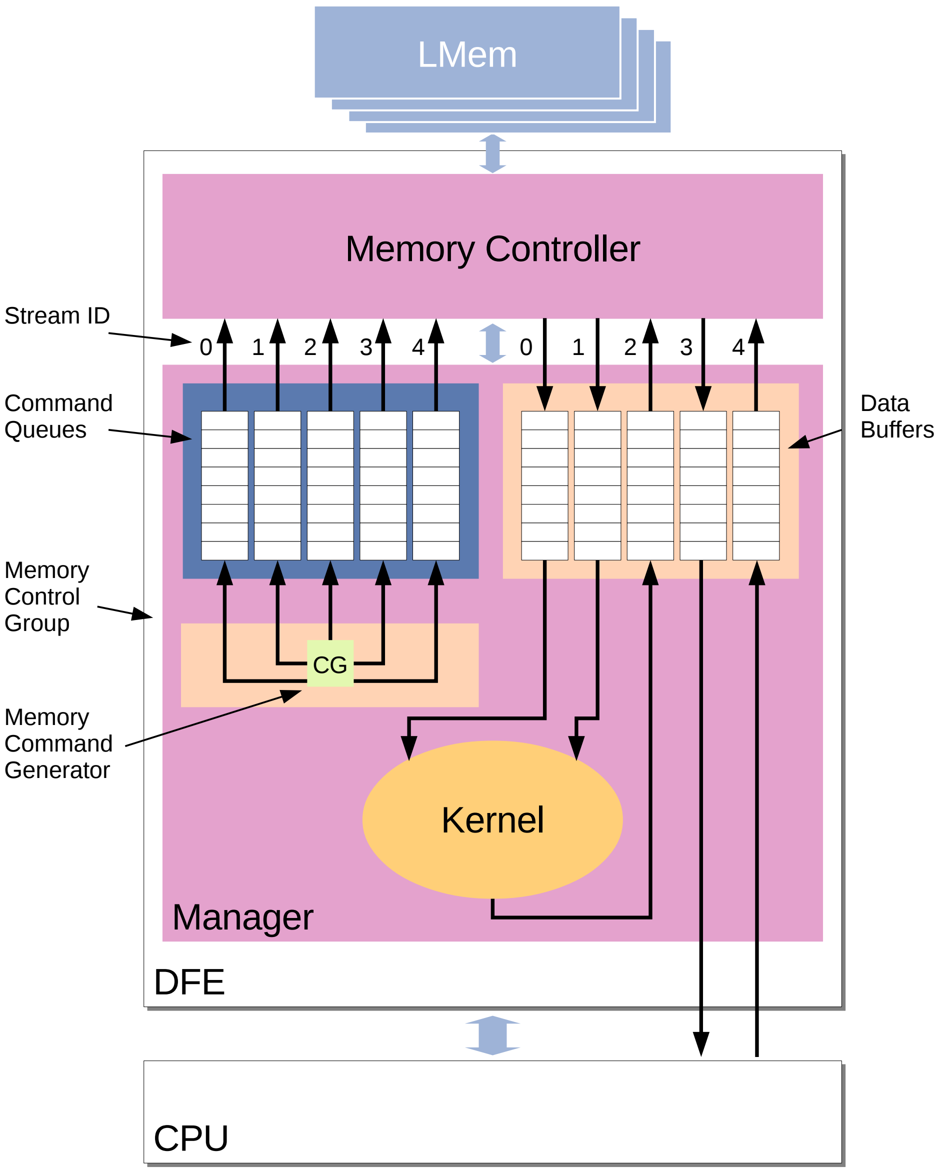


Figure 10: LMem controller architecture with memory control group

Figure 10 shows the architecture for Example 2, where the 5 memory streams have all been placed into a single memory control group.

A memory control group is created by calling the `addMemoryControlGroup` method with the de-

Listing 10: Manager demonstrating use of memory control groups (ControlGroupManager.maxj).

```
class ControlGroupManager extends CustomManager {
    private static final String KERNEL_NAME = "ControlGroupKernel";
    private static final CPUTypes TYPE = CPUTypes.INT32;

    ControlGroupManager(EngineParameters engineParameters) {
        super(engineParameters);
        KernelBlock k = addKernel(new ControlGroupKernel(makeKernelParameters(KERNEL_NAME)));

        MemoryControlGroup control = addMemoryControlGroup("control", MemoryControlGroup.MemoryAccessPattern.LINEAR_1D)
                ;

        DFELink fromcpu = addStreamFromCPU("fromcpu");
        DFELink tocpu = addStreamToCPU("tocpu");

        DFELink cpu2lmem = addStreamToOnCardMemory("cpu2lmem", control);
        DFELink lmem2cpu = addStreamFromOnCardMemory("lmem2cpu", control);

        cpu2lmem <== fromcpu;
        tocpu <== lmem2cpu;

        DFELink inA = addStreamFromOnCardMemory("inA",control);
        DFELink inB = addStreamFromOnCardMemory("inB",control);

        k.getInput("inA") <== inA;
        k.getInput("inB") <== inB;

        DFELink oData = addStreamToOnCardMemory("oData",control);
        oData <== k.getOutput("oData");
    }
```

sired access pattern:

MemoryControlGroup addMemoryControlGroup(String name, MemoryAccessPattern pattern)

Memory streams are added to the control group when they are created by using versions of the `addStreamFromOnCardMemory` and `addStreamToOnCardMemory` methods that take a memory control group as the second argument, rather than a memory access pattern:

Stream addStreamFromOnCardMemory(String name, MemoryControlGroup control_stream)
Stream addStreamToOnCardMemory(String name, MemoryControlGroup control_stream)

In CPU code or an engine interface, the memory command generator settings are set per stream, with all of the same settings except a different offset into the LMem.

SLiC does not verify that all calls to `max_lmem_*` in CPU code or `setLMem*` in a SLiC engine interface use the same common settings.

2.5 Example 2: Memory Control Group

In Example 2, we take the Manager design from Example 1 and modify it to make use of memory control groups. In fact, we can change this design to use a single control group. The source code for this Manager is shown in *Listing 10*.

We create our memory control group, `control`, with the linear memory command generator that we want all of our streams to use:

```
        MemoryControlGroup control = addMemoryControlGroup("control", MemoryControlGroup.MemoryAccessPattern.LINEAR_1D)
            ;
```

We then add all of our streams to this group:

```
        DFELink cpu2lmem = addStreamToOnCardMemory("cpu2lmem", control);
        DFELink lmem2cpu = addStreamFromOnCardMemory("lmem2cpu", control);
```

```
        DFELink inA = addStreamFromOnCardMemory("inA",control);
        DFELink inB = addStreamFromOnCardMemory("inB",control);
```

```
        DFELink oData = addStreamToOnCardMemory("oData",control);
```

The Manager has engine interfaces for reading and writing the LMem from the CPU (see *Listing 11*) and a default engine interface for running the Kernel:

```
    private static EngineInterface interfaceDefault () {
        EngineInterface ei = new EngineInterface();

        InterfaceParam N    = ei.addParam("N", TYPE);
        ei.setTicks(KERNEL_NAME, N);
        InterfaceParam sizeInBytes = N * TYPE.sizeInBytes();

        InterfaceParam zero = ei.addConstant(0l);
        ei.setLMemLinear("inA", zero, sizeInBytes);
        ei.setLMemLinear("inB", sizeInBytes, sizeInBytes);
        ei.setLMemLinear("oData", 2 * sizeInBytes, sizeInBytes);
        ei.ignoreAll(Direction.IN_OUT);
        return ei;
    }
```

The streams from the LMem to the Kernel are set individually with their start address in the LMem and the *same size*:

```
        InterfaceParam zero = ei.addConstant(0l);
        ei.setLMemLinear("inA", zero, sizeInBytes);
        ei.setLMemLinear("inB", sizeInBytes, sizeInBytes);
        ei.setLMemLinear("oData", 2 * sizeInBytes, sizeInBytes);
```

Any other parameters, including the unused memory streams to and from the CPU, are ignored:

```
        ei.ignoreAll(Direction.IN_OUT);
```

Grouping multiple streams into a Memory Control Group is an optimization: Maxeler recommends verifying correct functionality with separate memory command generators before creating memory control groups.

Listing 11: Engine interfaces for the Manager demonstrating use of memory control groups (ControlGroupManager.maxj).

```
	private static EngineInterface interfaceWrite(String name) {
		EngineInterface ei = new EngineInterface(name);

		InterfaceParam size = ei.addParam("size", TYPE);
		InterfaceParam start = ei.addParam("start", TYPE);
		InterfaceParam sizeInBytes = size * TYPE.sizeInBytes();

		ei.setStream("fromcpu", TYPE, sizeInBytes );
		ei.setLMemLinear("cpu2lmem", start * TYPE.sizeInBytes(), sizeInBytes);
		ei.ignoreAll(Direction.IN_OUT);
		return ei;
	}

	// reads the data back to the CPU from the LMem
	private static EngineInterface interfaceRead(String name) {
		EngineInterface ei = new EngineInterface(name);

		InterfaceParam size = ei.addParam("size", TYPE);
		InterfaceParam start = ei.addParam("start", TYPE);
		InterfaceParam sizeInBytes = size * TYPE.sizeInBytes();

		ei.setLMemLinear("lmem2cpu", start * TYPE.sizeInBytes(), sizeInBytes);
		ei.setStream("tocpu", TYPE, sizeInBytes);
		ei.ignoreAll(Direction.IN_OUT);
		return ei;
	}

	private static EngineInterface interfaceDefault() {
		EngineInterface ei = new EngineInterface();

		InterfaceParam N   = ei.addParam("N", TYPE);
		ei.setTicks(KERNEL_NAME, N);
		InterfaceParam sizeInBytes = N * TYPE.sizeInBytes();

		InterfaceParam zero = ei.addConstant(0l);
		ei.setLMemLinear("inA", zero, sizeInBytes);
		ei.setLMemLinear("inB", sizeInBytes, sizeInBytes);
		ei.setLMemLinear("oData", 2 * sizeInBytes, sizeInBytes);
		ei.ignoreAll(Direction.IN_OUT);
		return ei;
	}

	public static void main(String[] args) {
		ControlGroupManager m =
			new ControlGroupManager(new EngineParameters(args));
		m.createSLiCinterface(interfaceRead("readLMem"));
		m.createSLiCinterface(interfaceWrite("writeLMem"));
		m.createSLiCinterface(interfaceDefault());

		m.build();
	}
```

If a write stream in a memory control group is not available for a long time compared to other read streams in that group, for example if the write stream is scheduled much later in a Kernel or the read and writes streams are at either end of a long pipeline, the command queue for the write stream may become full, causing the memory command generator for the whole group to stop producing commands until that command queue has free space. This can lead to unrecoverable deadlock, where the Kernel cannot continue without more input data, but the memory command generator cannot produce any more commands until the Kernel produces an output.

A 2D strided memory command generator can only have one data stream to or from memory associated with it.

2.6 Custom Memory Command Streams

In addition to the pre-configured memory command generators, a custom memory command stream can be generated. The input stream is a normal stream that can come from any source, for example from a Kernel, the CPU or even the LMem itself (accessed via a different command stream).

For custom memory command streams originating from Kernels, an API (see *subsubsection 2.6.2*) allows memory commands to be generated without explicit reference to their binary format. Its use is strongly recommended for new projects as it abstracts the detail of the command stream implementation, offers better performance and enables MaxCompiler to perform more compile-time analysis. Commands from other sources should conform to the specifications given in *subsubsection 2.8.1*.

There are several rules that apply to commands:

- Commands are not executed until the respective stream data buffer has either enough space for read streams or enough data for write streams.
- Commands are atomic operations: once a command starts, it runs to completion before the next command can be started.
- For a specific memory read or write stream, commands to that stream are executed in the order they enter their respective command queue. Commands that are waiting for space or data block the stream.
- No ordering is guaranteed between different streams either from the same control group or different control groups.

Use custom memory control streams with caution: suboptimal addressing of the LMem can cause a major decrease in the performance of your design.

It is often simpler to generate a command stream from a separate Kernel to a computation Kernel as managing the stream control within the same Kernel can create convoluted Kernel code and has the potential to create a circular dependence between data streams in or out of the Kernel to LMem and the command stream output.

2.6.1 Memory Command Streams and Control Groups

To associate a memory command stream with a single memory stream, there is a variant of the `addStreamToOnCardMemory` method:

*Stream addStreamToOnCardMemory(**String** name, Stream control_stream)*

To create a memory control group associated with a memory command stream, there is a variant of `addMemoryControlGroup`:

*MemoryControlGroup addMemoryControlGroup(**String** name, Stream control_stream)*

Figure 11 shows the LMem controller architecture with the same 5 streams as before in a single memory control group with a memory command stream generated in the CPU and transferred over PCI Express.

A memory control command has a **Stream Select** field because a memory command stream can be associated with either a single memory stream or a memory control group that can contain one or more streams.

For each memory control group, **Stream Select** numbering is independent and in the order: read streams followed by write streams. The numbering of streams is determined when the Manager graph is constructed. For a memory command stream associated with a single memory stream, the stream select number in the command should always be `1`.

The stream select ID for a stream can be determined in the Manager using `getStreamIndexIdWithinGroup` with the stream name:

int *MemoryControlGroup.getStreamIndexIdWithinGroup(**String** name)*

The stream select ID is read by the memory control group to place the command in the command queue for the correct stream. The memory control group does not read any more commands from the incoming command stream if any of the command queues for the group are full.

A single command stream for multiple memory streams should only be used where the data throughput is similar for all memory streams in the group, otherwise there is a risk of the design stalling when a command buffer becomes full.

2.6.2 Kernel API

`LMemCommandStream.makeKernelOutput` can be used within a Kernel to create an output stream that can be used as a custom memory command generator.

There are two versions of this method. The first uses separate streams for each part of the command:

void *LMemCommandStream.makeKernelOutput(**String** output_name, DFEVar **control**, DFEVar address,*
DFEVar size, DFEVar inc, DFEVar stream_num, DFEVar tag)

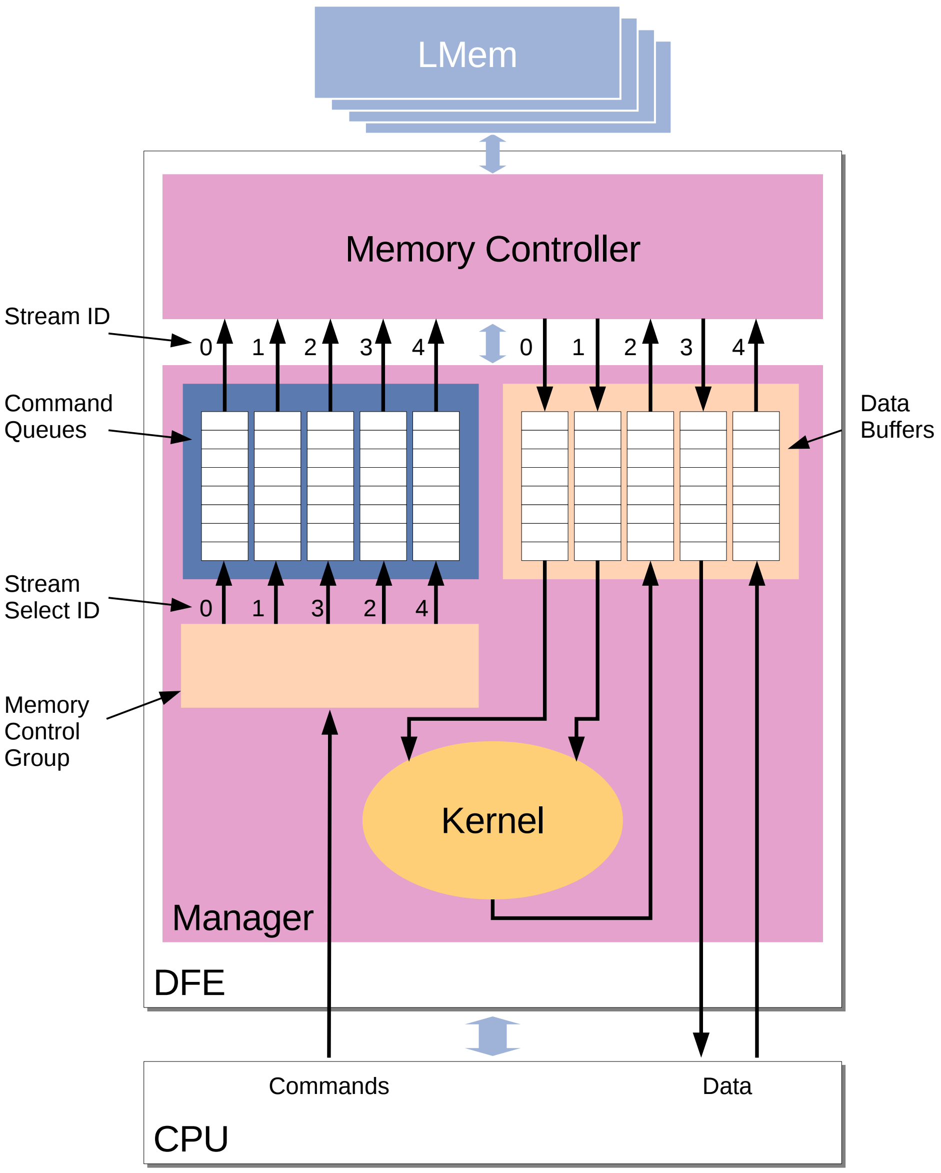


Figure 11: LMem controller architecture with a memory command stream associated with a memory control group.

The second uses a single `DFEStruct` stream containing the complete command command:

__void__ LMemCommandStream.makeKernelOutput(__String__ output_name, DFEVar __control__, DFEStruct command_struct)

The required `DFEStruct` type can be retrieved using:

LMemCommandStream.getLMemCommandDFEStructType()

The names of the arguments in the first version of `LMemCommandStream.makeKernelOutput` match the names of the fields in the `DFEStruct` for the second version. Each argument/field is described in *Table 1*. The field widths in the `DFEStruct` must exactly match those shown in *Table 1*, whereas the previous version automatically adjusts to narrower stream widths by padding as necessary.

A command may specify multiple bursts to read or write, with either contiguous or dispersed locations. A command includes arguments for the **Command Address** (`address`), **Command Size** (`size`) and **Command Increment** (`inc`) of the operation.

The value of `stream_num` should always be 0 if the command is associated with an individual stream or a control group containing only one stream.

Table 1: Memory Command Stream Components

Type	Argument/Field Name	Description	Valid Range
`String`	`output_name`	Name for the stream	n/a
`dfeBool()`	`control`	Control to enable or disable the stream in a cycle	`false`=Disabled `true`=Enabled
`dfeUInt(28)`	`address`	Start address in *bursts* for read or write command	0 to $2^{27}-1$
`dfeUInt(8)`	`size`	Number of *bursts* to read or write	1 to 128 (0 and values greater than 128 are invalid)
`dfeUInt(7)`	`inc`	Number of *bursts* added to address after each burst	1 to 127 (0 is invalid)
`dfeUInt(4)`	`stream_num`	Selects stream to apply command to Command type (read or write) determined by memory stream type	X=0 to 14
`dfeBool()`	`tag`	Tags the command to raise an interrupt	`false`=No Interrupt `true`=Interrupt

The `tag` argument instructs the memory controller to raise an interrupt once the memory command has been executed. This allows the CPU to synchronize with the DFE at the point when all the data for a stream has been written to or read from the LMem.

It is recommended to tag only the final command in the memory command stream for a memory command group as there is no way to differentiate between interrupts raised by different tagged commands.

The CPU code can be set to wait on an interrupt raised by a custom memory command stream generator either in an engine interface or via the SLiC Interface in the CPU code itself:

```
void max_lmem_set_interrupt_on (max_actions_t *actions, const char *mem_stream_name)
```

```
EngineInterface.setLMemInterruptOn(String streamName);
```

2.7 Example 3: Custom Memory Command Stream

Example 3 implements the same functionality as Example 1, except that the streams between the LMem and the Kernel use custom memory command streams.

2.7.1 Kernel

Listing 12 shows the Kernel source code for Kernel that adds two input streams and sends them to an output stream, and also generates the three custom memory command streams needed to control the LMem connected to these streams.

Two scalar inputs, one for the number of words in a burst and one for the total number of bursts, are passed into the Kernel to control a counter chain:

```
        DFEVar totalBursts = io.scalarInput("totalBursts",dfeUInt(32));
        DFEVar wordsPerBurst = io.scalarInput("wordsPerBurst",dfeUInt(32));

        CounterChain chain = control.count.makeCounterChain();
        DFEVar burstCount = chain.addCounter(totalBursts,1);
        DFEVar wordCount = chain.addCounter(wordsPerBurst,1);
```

The input and output streams each transmit a 32-bit word on every cycle, but the command streams issue a new command only at the start of each burst. To synchronize their operation, the counter chain is used to control the output of all the command streams so that they are only enabled at the start of each burst.

The command streams for the inputs differ only in their read address, and simply read one burst at a time, with no increment and no interrupt.:

```
        LMemCommandStream.makeKernelOutput("AcmdStream",
                wordCount === 0,                // control
                burstCount,                     // address
                constant.var(dfeUInt(8), 1),    // size
                constant.var(dfeUInt(1), 0),    // inc
                constant.var(dfeUInt(1), 0),    // stream
                constant.var(false));
        LMemCommandStream.makeKernelOutput("BcmdStream",
                wordCount === 0,                // control
                totalBursts + burstCount,       // address
                constant.var(dfeUInt(8), 1),    // size
                constant.var(dfeUInt(1), 0),    // inc
                constant.var(dfeUInt(1), 0),    // stream
                constant.var(false));
```

The output command stream generates an interrupt when it reaches the last burst, as indicated by

Listing 12: Kernel demonstrating use of custom memory command streams (CmdStreamKernel.maxj)

```
/**
 * Document: Manager Compiler Tutorial (maxcompiler-manager-tutorial.pdf)
 * Chapter: 2      Example: 3      Name: Command Stream
 * MaxFile name: CmdStream
 * Summary:
 *      Computes the sum of the two input streams and sends it to an output
 *      stream. Also generates the three custom memory command streams needed to
 *      control the LMem connected to these streams.
 */
package cmdstream;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.LMemCommandStream;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.CounterChain;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;

class CmdStreamKernel extends Kernel {

    CmdStreamKernel(KernelParameters parameters) {
        super(parameters);
        DFEVar inB = io.input("inB", dfeUInt(32));
        DFEVar inA = io.input("inA", dfeUInt(32));
        DFEVar totalBursts = io.scalarInput("totalBursts",dfeUInt(32));
        DFEVar wordsPerBurst = io.scalarInput("wordsPerBurst",dfeUInt(32));

        CounterChain chain = control.count.makeCounterChain();
        DFEVar burstCount = chain.addCounter(totalBursts,1);
        DFEVar wordCount = chain.addCounter(wordsPerBurst,1);
        LMemCommandStream.makeKernelOutput("AcmdStream",
                wordCount === 0,                // control
                burstCount,                     // address
                constant.var(dfeUInt(8), 1),    // size
                constant.var(dfeUInt(1), 0),    // inc
                constant.var(dfeUInt(1), 0),    // stream
                constant.var(false));
        LMemCommandStream.makeKernelOutput("BcmdStream",
                wordCount === 0,                // control
                totalBursts + burstCount,       // address
                constant.var(dfeUInt(8), 1),    // size
                constant.var(dfeUInt(1), 0),    // inc
                constant.var(dfeUInt(1), 0),    // stream
                constant.var(false));
        LMemCommandStream.makeKernelOutput("OcmdStream",
                wordCount === 0,                  // control
                totalBursts * 2 + burstCount,     // address
                constant.var(dfeUInt(8), 1),      // size
                constant.var(dfeUInt(1), 0),      // inc
                constant.var(dfeUInt(1), 0),      // stream
                burstCount === (totalBursts - 1));

        io.output("oData", inA+inB, dfeUInt(32));
    }
}
```

the condition in its tag argument:

```
44          LMemCommandStream.makeKernelOutput("OcmdStream",
45                  wordCount === 0,                  // control
46                  totalBursts * 2 + burstCount,     // address
47                  constant.var(dfeUInt(8), 1),      // size
48                  constant.var(dfeUInt(1), 0),      // inc
49                  constant.var(dfeUInt(1), 0),      // stream
50                  burstCount === (totalBursts - 1));
```

Listing 13: Manager demonstrating use of custom memory command streams (CmdStreamManager.maxj)

```
class CmdStreamManager extends CustomManager {

    private static final String KERNEL_NAME = "CmdStreamKernel";
    private static final CPUTypes TYPE = CPUTypes.INT32;

    CmdStreamManager(EngineParameters engineParameters) {
        super(engineParameters);
        KernelBlock k = addKernel(new CmdStreamKernel(makeKernelParameters(KERNEL_NAME)));
        debug.setDebugLevel(new DebugLevel(){{setHasStreamStatus(true);}});
        DFELink fromcpu = addStreamFromCPU("fromcpu");
        DFELink tocpu = addStreamToCPU("tocpu");

        DFELink tolmem = addStreamToOnCardMemory("tolmem",MemoryControlGroup.MemoryAccessPattern.LINEAR_1D);
        DFELink fromlmem = addStreamFromOnCardMemory("fromlmem",MemoryControlGroup.MemoryAccessPattern.LINEAR_1D);

        tolmem <== fromcpu;
        tocpu <== fromlmem;

        DFELink inB = addStreamFromOnCardMemory("inB",k.getOutput("BcmdStream"));
        DFELink inA = addStreamFromOnCardMemory("inA",k.getOutput("AcmdStream"));

        k.getInput("inA") <== inA;
        k.getInput("inB") <== inB;

        DFELink oData = addStreamToOnCardMemory("oData",k.getOutput("OcmdStream"));
        oData <== k.getOutput("oData");
    }
```

2.7.2 Manager

The manager in this example differs from that of Examples 1 and 2 only insofar as the `addStream` methods are invoked with the name of a command stream generated by the kernel in some cases rather than a standard access pattern generator, as shown in *Listing 13*.

The memory command streams are associated with streams from the LMem using a variant of the method `addStreamFromOnCardMemory`, then the LMem streams are connected to the inputs of the Kernel:

```
        DFELink inB = addStreamFromOnCardMemory("inB",k.getOutput("BcmdStream"));
        DFELink inA = addStreamFromOnCardMemory("inA",k.getOutput("AcmdStream"));

        k.getInput("inA") <== inA;
        k.getInput("inB") <== inB;
```

Likewise, the memory command stream for the output from the Kernel is associated with a stream to the LMem, which is then connected to the output of the Kernel:

```
        DFELink oData = addStreamToOnCardMemory("oData",k.getOutput("OcmdStream"));
        oData <== k.getOutput("oData");
```

There are engine interfaces for reading accessing the LMem from the CPU (shown in *Listing 14*)

Listing 14: Engine interfaces for Manager demonstrating use of custom memory command streams (CmdStreamManager.maxj)

```
    private static EngineInterface interfaceWrite(String name) {
        EngineInterface ei = new EngineInterface(name);

        InterfaceParam size = ei.addParam("size", TYPE);
        InterfaceParam start = ei.addParam("start", TYPE);
        InterfaceParam sizeInBytes = size * TYPE.sizeInBytes();

        ei.setStream("fromcpu", TYPE, sizeInBytes );
        ei.setLMemLinear("tolmem", start * TYPE.sizeInBytes(), sizeInBytes);
        ei.ignoreAll(Direction.IN_OUT);
        return ei;
    }

    // reads the data back to the CPU from the LMem
    private static EngineInterface interfaceRead(String name) {
        EngineInterface ei = new EngineInterface(name);

        InterfaceParam size = ei.addParam("size", TYPE);
        InterfaceParam start = ei.addParam("start", TYPE);
        InterfaceParam sizeInBytes = size * TYPE.sizeInBytes();

        ei.setLMemLinear("fromlmem", start * TYPE.sizeInBytes(), sizeInBytes);
        ei.setStream("tocpu", TYPE, sizeInBytes);
        ei.ignoreAll(Direction.IN_OUT);
        return ei;
    }

    private static EngineInterface interfaceDefault() {
        EngineInterface ei = new EngineInterface();

        InterfaceParam N    = ei.addParam("N", TYPE);
        InterfaceParam burstSize = ei.addParam("burstSize", TYPE);
        ei.setTicks(KERNEL_NAME, N);

        ei.setScalar(KERNEL_NAME, "totalBursts", (N*(TYPE.sizeInBytes()))/burstSize);
        ei.setScalar(KERNEL_NAME, "wordsPerBurst", burstSize/(TYPE.sizeInBytes()));

        ei.setLMemInterruptOn("oData");
        ei.ignoreAll(Direction.IN_OUT);
        return ei;
    }

    public static void main(String[] args) {
        CmdStreamManager m = new CmdStreamManager(new EngineParameters(args));
        m.createSLiCinterface(interfaceRead("readLMem"));
        m.createSLiCinterface(interfaceWrite("writeLMem"));
        m.createSLiCinterface(interfaceDefault());

        m.build();
    }
```

and a default interface for running the Kernel:

```
        m.createSLiCinterface(interfaceRead("readLMem"));
        m.createSLiCinterface(interfaceWrite("writeLMem"));
        m.createSLiCinterface(interfaceDefault());
```

In the default interface, parameters for the number of words in the stream and the burst size are added, from which the scalar inputs to the Kernel are calculated:

```
private static EngineInterface interfaceDefault () {
    EngineInterface ei = new EngineInterface();

    InterfaceParam N    = ei.addParam("N", TYPE);
    InterfaceParam burstSize = ei.addParam("burstSize", TYPE);
    ei.setTicks(KERNEL_NAME, N);

    ei.setScalar(KERNEL_NAME, "totalBursts", (N*(TYPE.sizeInBytes()))/burstSize);
    ei.setScalar(KERNEL_NAME, "wordsPerBurst", burstSize/(TYPE.sizeInBytes()));

    ei.setLMemInterruptOn("oData");
    ei.ignoreAll(Direction.IN_OUT);
    return ei;
}
```

The LMem interrupt is set to be raised from the output data stream from the Kernel:

```
    ei.setLMemInterruptOn("oData");
```

2.7.3 CPU Application

In the CPU application, the data is loaded and read back from the LMem using the `writeLMem` and `readLMem` engine interfaces.

The burst size in bytes is read from the `.max` file:

```
int burstLengthInBytes = max_get_burst_size(maxfile, "cmd_tolmem");
```

To run the Kernel, the default engine interface is used with the parameters for the size of the stream in words and the burst size in bytes:

```
CmdStream(size, burstLengthInBytes);
```

2.8 Example 4: Custom Command Stream Using Structures

Example 4 is identical to Example 3, except that it uses a `DFEStruct` for specifying the memory commands. The Manager and CPU code are unchanged from Example 3. The Kernel is shown in *Listing 15*.

In this example, the parameter `ACmdStream` is a `DFEStruct`:

```
    DFEStruct AcmdStream = LMemCommandStream.getLMemCommandDFEStructType().newInstance(this);
```

The `DFEStruct AcmdStream` can be initialized using the named fields:

```
    AcmdStream["address"] = burstCount;
    AcmdStream["size"] = constant.var(dfeUInt(8), 1);
    AcmdStream["inc"] = constant.var(dfeUInt(8), 0);
    AcmdStream["stream"] = constant.var(dfeRawBits(15), one);
    AcmdStream["tag"] = constant.var(false);
```

`one` is a bit vector given by:

```
Bits one = new Bits(15);
one.setBit(0, 1);
one.setOthers(0);
```

The output itself is declared with `AcmdStream` as an argument and the control stream as before:

```
LMemCommandStream.makeKernelOutput("AcmdStream", wordCount === 0, AcmdStream);
```

2.8.1 Binary Command Stream Format

Some legacy applications may use a standard 64-bit Kernel output for a memory command stream, where each command is in the format shown in *Table 2*.

Table 2: Memory Command Stream Raw Data Format

Bits	Field Name	Description	Valid Range
27:0	Command Address	Start address in *bursts* for read or write command	0 to $2^{27}-1$
31:28	Not used		
39:32	Command Size	Number of *bursts* to read or write	1 to 128 (0 and values greater than 128 are invalid)
46:40	Command Increment (Mode=0)	Number of *bursts* added to address after each burst	1 to 127 (0 is invalid)
46:40	Command Increment (Mode=1)	Number of *bursts* added to address after each burst (X = 46:40)	$2^9 * X$
47	Mode (see above)		0/1
48+(N-1):48	Stream Select	Selects stream to apply command to (*One-hot* encoding) Command type (read or write) determined by memory stream type	2^X X=0 to N-1 (0 is invalid)
62:48+N	Not used		
63	Tag	Tags the command to raise an interrupt	0=No Interrupt 1=Interrupt

It is not recommended to use a standard Kernel output for a memory command stream, as this is both error-prone and prevents MaxCompiler from performing some analysis on the command stream.

Listing 15: Kernel demonstrating use of custom memory command streams (DFEStructKernel.maxj)

```
/**
 * Document: Manager Compiler Tutorial (maxcompiler-manager-tutorial.pdf)
 * Chapter: 2      Example: 4      Name: DFEstruct
 * MaxFile name: DFEStruct
 * Summary:
 *      Computes the sum of the two input streams and sends it to an output
 *      stream. Also generates the three custom memory command streams needed to
 *      control the LMem connected to these streams using DFEStructs.
 */

package dfestruct;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.LMemCommandStream;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.CounterChain;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.composite.DFEStruct;
import com.maxeler.maxcompiler.v2.utils.Bits;

class DFEStructKernel extends Kernel {

    DFEStructKernel(KernelParameters parameters) {
        super(parameters);

        DFEVar totalBursts = io.scalarInput("totalBursts",dfeUInt(32));
        DFEVar wordsPerBurst = io.scalarInput("wordsPerBurst",dfeUInt(32));
        CounterChain chain = control.count.makeCounterChain();
        DFEVar burstCount = chain.addCounter(totalBursts,1);
        DFEVar wordCount = chain.addCounter(wordsPerBurst,1);
        DFEVar inB = io.input("inB", dfeUInt(32));
        DFEVar inA = io.input("inA", dfeUInt(32));
        DFEStruct BcmdStream = LMemCommandStream.getLMemCommandDFEStructType().newInstance(this);
        DFEStruct OcmdStream = LMemCommandStream.getLMemCommandDFEStructType().newInstance(this);
        DFEStruct AcmdStream = LMemCommandStream.getLMemCommandDFEStructType().newInstance(this);

        Bits one = new Bits(15);
        one.setBit(0, 1);
        one.setOthers(0);

        AcmdStream["address"] = burstCount;
        AcmdStream["size"] = constant.var(dfeUInt(8), 1);
        AcmdStream["inc"] = constant.var(dfeUInt(8), 0);
        AcmdStream["stream"] = constant.var(dfeRawBits(15), one);
        AcmdStream["tag"] = constant.var(false);
        LMemCommandStream.makeKernelOutput("AcmdStream", wordCount === 0, AcmdStream);

        BcmdStream["address"] = totalBursts + burstCount;
        BcmdStream["size"] = constant.var(dfeUInt(8), 1);
        BcmdStream["inc"] = constant.var(dfeUInt(8), 0);
        BcmdStream["stream"] = constant.var(dfeRawBits(15), one);
        BcmdStream["tag"] = constant.var(false);
        LMemCommandStream.makeKernelOutput("BcmdStream", wordCount === 0, BcmdStream);

        OcmdStream["address"] = totalBursts * 2 + burstCount;
        OcmdStream["size"] = constant.var(dfeUInt(8), 1);
        OcmdStream["inc"] = constant.var(dfeUInt(8), 0);
        OcmdStream["stream"] = constant.var(dfeRawBits(15), one);
        OcmdStream["tag"] = burstCount === (totalBursts - 1);
        LMemCommandStream.makeKernelOutput("OcmdStream", wordCount === 0, OcmdStream);

        io.output("oData", inA+inB, dfeUInt(32));
    }
}
```

3 MaxRing

The MaxRing interconnect allows data to be transferred at high speed directly between DFEs connected via a cable between MaxCards in a multi-card Maxeler installation. Each MaxCard in the system has a direct bidirectional connection to up to two other MaxCards, as shown in *Figure 12*.

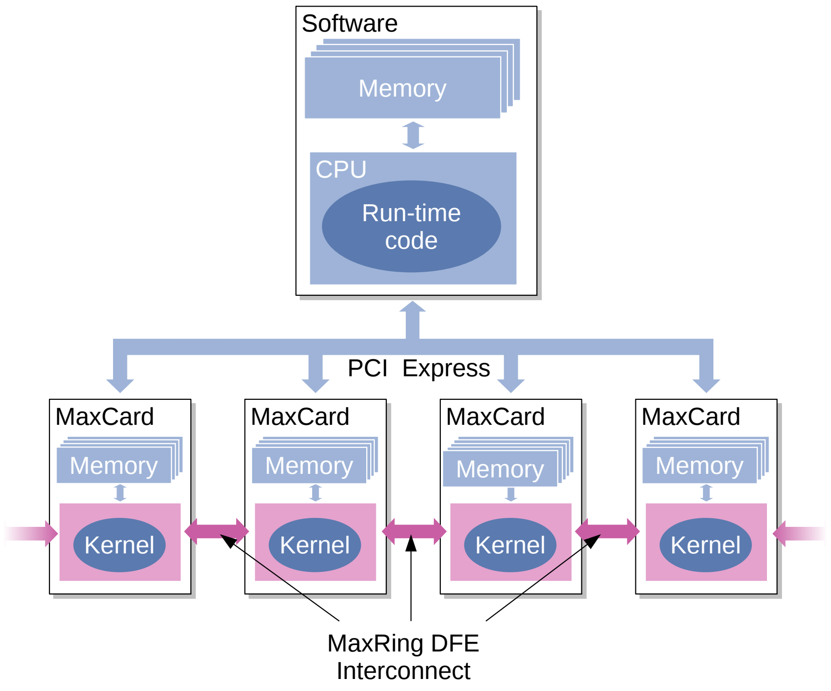


Figure 12: MaxRing connecting multiple MaxCards together directly.

3.1 Adding MaxRing Links to the Manager

The Manager Compiler allows full use of both MaxRing board interconnects on MAX2, Vectis and Coria and the inter-DFE link on the MAX2.

A bi-directional MaxRing link is declared in the Manager using `addMaxRingBidirectionalStream`, which returns a `MaxRingBidirectionalStream` object:

*MaxRingBidirectionalStream addMaxRingBidirectionalStream(**String** name, MaxRingConnection connection)*

For Vectis and Coria, there is one user DFE per card and each card can be connected to up to two other cards via MaxRing. For these DFEs, available connection types are `MAXRING_A` and `MAXRING_B`.

On a MAX2 card, each DFE is directly connected to the other DFE on the card and can be connected to another card via MaxRing. Available connection types on the MAX2 are `DFE_ON_LOCAL_CARD` and `DFE_ON_REMOTE_CARD`.

The returned `MaxRingBidirectionalStream` object has an input stream for the outgoing data and an output stream for the incoming data from the other DFE:

Stream MaxRingBidirectionalStream.getStreamToRemoteDFE()
Stream MaxRingBidirectionalStream.getStreamFromRemoteDFE()

3.2 Multi-DFE Example

Example 1 demonstrates the use of MaxRing between two Vectis DFEs. The same simple Kernel that adds one to an input stream and writes the result to an output stream is loaded into the DFE on each card. The output of the Kernel is connected to the output onto MaxRing link. A multiplexer is used to switch the input to the Kernel between a CPU input and input from the MaxRing link. A demultiplexer is used to switch the input from the MaxRing link between the Kernel input and output to the CPU.

Figure 13 shows the Manager graph for this example duplicated to show the flow of data between the two DFEs. The multiplexer and demultiplexer in DFE 1 are used to bypass the Kernel for the data coming back from the Kernel in DFE 2.

The source code for the Manager is shown in *Listing 16*. The bidirectional stream is defined to connect to MaxRing:

```
		MaxRingBidirectionalStream maxRingStream = addMaxRingBidirectionalStream(
			"maxRingStream", Max3RingConnection.MAXRING_A);
```

The output of the Kernel is connected to the outbound stream of the bidirectional stream:

```
		maxRingStream.getLinkToRemoteDFE() <== k.getOutput("oData");
```

The input from the bidirectional stream is connected to a demultiplexer, which selects whether the data should be passed to the Kernel input or to the CPU output:

```
		Demux bypassSwitch = demux("bypassSwitch");
		tocpu <== bypassSwitch.addOutput("tocpu");
		DFELink toKernel = bypassSwitch.addOutput("toKernel");
		bypassSwitch.getInput() <== maxRingStream.getLinkFromRemoteDFE();
```

A multiplexer selects the input for the Kernel from the CPU input and the input from MaxRing:

```
		Mux inputSwitch = mux("inputSwitch");
		inputSwitch.addInput("fromcpu") <== fromcpu;
		inputSwitch.addInput("fromMaxRing") <== toKernel;
		k.getInput("iData") <== inputSwitch.getOutput();
```

The significant section of the CPU code is shown in *Listing 17*.

An Advanced Static actions array is used to hold the actions for each device

```
	MaxRing_actions_t *actions[2];
	MaxRing_actions_t interface_actions [2];
```

The actions for the Kernel in the first device, designated right, are set up to get input from the CPU and pass the output from the second device back to the CPU:

```
	// Right
	interface_actions [0]. routing_string
		= "fromcpu -> inputSwitch, bypassSwitch -> tocpu";
	interface_actions [0]. instream_fromcpu = iData;
	interface_actions [0]. outstream_tocpu = oData;
	interface_actions [0]. param_dataSize = dataSize;
	interface_actions [0]. ticks_MaxRingKernel = dataSize;
	actions[0] = &interface_actions [0];
```

The actions for the Kernel in the second device, designated left, is set up to receive its input from

MaxRing, and the input and output streams are set to `NULL`:

```
    // Left
    interface_actions [1]. routing_string
        = "fromMaxRing -> inputSwitch, bypassSwitch -> toKernel";
    interface_actions [1]. instream_fromcpu = NULL;
    interface_actions [1]. outstream_tocpu = NULL;
    interface_actions [1]. param_dataSize = 0;
    interface_actions [1]. ticks_MaxRingKernel = dataSize;
    actions[1] = &interface_actions [1];
```

The same maxfile is loaded into two DFEs in a DFE array:

```
    max_file_t *maxfile = MaxRing_init();
    max_engarray_t *array = max_load_array(maxfile, 2, "*");
```

The the array of actions is then run on the DFE array, which loads the settings to the DFEs and streams data to and from the left device:

```
    MaxRing_run_array(array, actions);
```

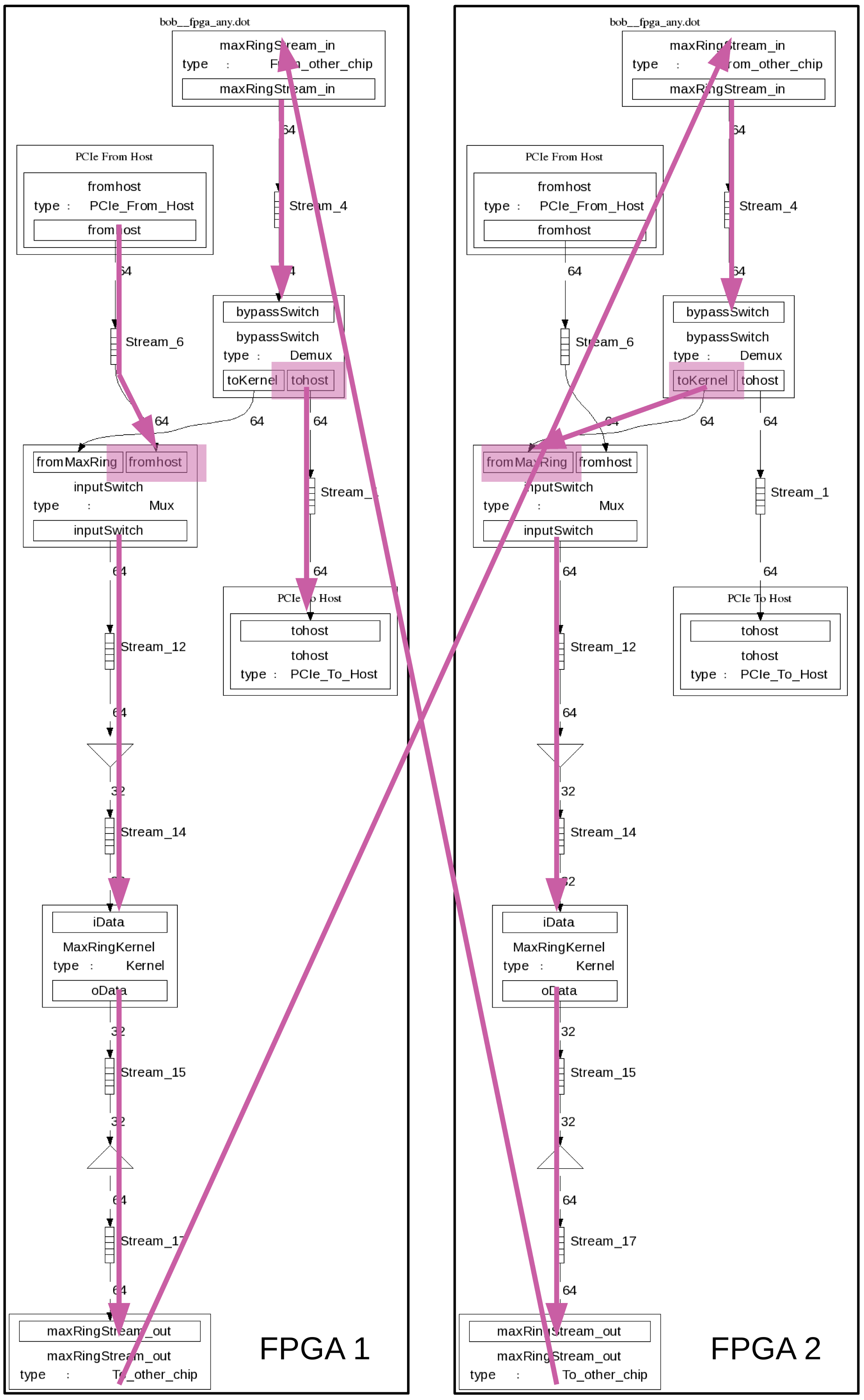


Figure 13: MaxRing example Manager graph showing the flow of data between the two DFEs

Listing 16: Example Manager showing inter-DFE communications (MaxRingManager.maxj).

```
/**
 * Document: Manager Compiler Tutorial (maxcompiler-manager-tutorial.pdf)
 * Chapter: 3        Example: 1       Name: MaxRing
 * MaxFile name: MaxRing
 * Summary:
 *       Create a setup with two maxring-connected dataflow engines each with a mux
 *       with a "fromcpu" and "frommaxring" port and a demux with a "tocpu" and
 *       "tokernel" port.
 */

package maxring;

import com.maxeler.maxcompiler.v2.build.EngineParameters;
import com.maxeler.maxcompiler.v2.managers.custom.CustomManager;
import com.maxeler.maxcompiler.v2.managers.custom.DFELink;
import com.maxeler.maxcompiler.v2.managers.custom.blocks.Demux;
import com.maxeler.maxcompiler.v2.managers.custom.blocks.KernelBlock;
import com.maxeler.maxcompiler.v2.managers.custom.blocks.Mux;
import com.maxeler.maxcompiler.v2.managers.custom.stdlib.Max3RingConnection;
import com.maxeler.maxcompiler.v2.managers.custom.stdlib.MaxRingBidirectionalStream;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.CPUTypes;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.EngineInterface;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.InterfaceParam;

class MaxRingManager extends CustomManager {

    MaxRingManager(EngineParameters engineParameters) {
        super(engineParameters);
        KernelBlock k = addKernel(
            new MaxRingKernel(makeKernelParameters("MaxRingKernel")));

        DFELink fromcpu = addStreamFromCPU("fromcpu");
        DFELink tocpu = addStreamToCPU("tocpu");

        MaxRingBidirectionalStream maxRingStream = addMaxRingBidirectionalStream(
            "maxRingStream", Max3RingConnection.MAXRING_A);

        maxRingStream.getLinkToRemoteDFE() <== k.getOutput("oData");

        Demux bypassSwitch = demux("bypassSwitch");
        tocpu <== bypassSwitch.addOutput("tocpu");
        DFELink toKernel = bypassSwitch.addOutput("toKernel");
        bypassSwitch.getInput() <== maxRingStream.getLinkFromRemoteDFE();

        Mux inputSwitch = mux("inputSwitch");
        inputSwitch.addInput("fromcpu") <== fromcpu;
        inputSwitch.addInput("fromMaxRing") <== toKernel;
        k.getInput("iData") <== inputSwitch.getOutput();
    }

    static EngineInterface interfaceDefault () {
        EngineInterface ei = new EngineInterface();
        InterfaceParam size = ei.addParam("dataSize", CPUTypes.INT32);
        ei.setStream("fromcpu", CPUTypes.UINT32, size * CPUTypes.UINT32.sizeInBytes());
        ei.setStream("tocpu", CPUTypes.UINT32, size * CPUTypes.UINT32.sizeInBytes());
        return ei;
    }

    public static void main(String[] args) {
        EngineParameters engineParameters = new EngineParameters(args);
        MaxRingManager m = new MaxRingManager(engineParameters);
        m.createSLiCinterface(interfaceDefault());
        m.build();
    }
}
```

Listing 17: Snippet of CPU code showing inter-DFE communications (MaxRingCpuCode.c).

```
int main()
{
    const int dataSize = 384;
    size_t sizeBytes = dataSize * sizeof(uint32_t);
    uint32_t *iData = malloc(sizeBytes);
    uint32_t *oData = malloc(sizeBytes);
    uint32_t *expected = malloc(sizeBytes);

    generateInputData(dataSize, iData);

    MaxRingCPU(dataSize, iData, expected);

    MaxRing_actions_t *actions[2];
    MaxRing_actions_t interface_actions [2];

    // Right
    interface_actions [0]. routing_string
        = "fromcpu -> inputSwitch, bypassSwitch -> tocpu";
    interface_actions [0]. instream_fromcpu = iData;
    interface_actions [0]. outstream_tocpu = oData;
    interface_actions [0]. param_dataSize = dataSize;
    interface_actions [0]. ticks_MaxRingKernel = dataSize;
    actions[0] = &interface_actions [0];

    // Left
    interface_actions [1]. routing_string
        = "fromMaxRing -> inputSwitch, bypassSwitch -> toKernel";
    interface_actions [1]. instream_fromcpu = NULL;
    interface_actions [1]. outstream_tocpu = NULL;
    interface_actions [1]. param_dataSize = 0;
    interface_actions [1]. ticks_MaxRingKernel = dataSize;
    actions[1] = &interface_actions [1];

    printf ("Running DFE.\n");
    max_file_t *maxfile = MaxRing_init();
    max_engarray_t *array = max_load_array(maxfile, 2, "*");

    MaxRing_run_array(array, actions);

    max_unload_array(array);
    max_file_free (maxfile);

    int status = check(dataSize, expected, oData);
    if (status)
        printf ("Test failed.\n");
    else
        printf ("Test passed OK!\n");

    return status;
}
```

4 Build Configuration

In addition to connecting together Kernels and IO, Managers control the build process, which includes running third-party tools for the production of a DFE bitstream.

4.1 The Build Process

All Managers, including Custom Managers and the Standard Manager, extend the `maxcompiler.v1.managers.DFEManager` class which provides several methods.

The `build` method launches the build process:

```
abstract void build()
```

`build` can be be called only once.

`logMsg` and `logWarning` allow you to log messages during the build process that are output in the `_build.log` file in the build directory:

```
void logMsg(String msg, Object... args)
void logWarning(String msg, Object... args)
```

Logging build messages in this way is preferable to printing directly to the console as these messages are saved for reference. Messages are formatted using a `printf`-like format.

You can specify named integer constants to be written into the `.max` file and read by the CPU code:

```
void addMaxFileConstant(String name, int value)
```

Note that this does not get built into the DFE in any way: it is purely for sharing information between the DFE and CPU code build environments. The MaxCompilerRT function for accessing these constants is:

```
int max_get_constant_from_maxfile(const max_maxfile_t *maxfile, const char *name);
```

4.2 Build Configuration Objects

`BuildConfig` objects are useful for controlling the configuration of the build process.

```
BuildConfig getBuildConfig()
void setBuildConfig(BuildConfig build_config)
```

4.2.1 Build Effort

Build effort tells the third party tools how much effort to put into trying to find an implementation of the circuit in the DFE to meet the design requirements (clock speed and area):

```
void setBuildEffort(BuildConfig.Effort effort)
```

The options available are `HIGH`, `LOW`, `MEDIUM`, `VERY_HIGH`. Though it may be tempting to always run with high effort levels, builds can take a long time when constraints are tight, so lower effort levels are useful for iterative test and optimization.

4.2.2 Build Level

Build level tells MaxCompiler up to which stage to run the build process:

void *setBuildLevel(BuildConfig.Level level)*

By default, MaxCompiler runs the complete build process and produce a `.max` file. Options available are:

- `FULL_BUILD` (default) runs the complete process
- `COMPILE_ONLY` stops after producing the VHDL output from MaxCompiler
- `SYNTHESIS` compiles the VHDL output
- `MAP` maps the synthesized design to components in the DFE
- `PAR` places and routes the design for the DFE, producing a bit stream

Levels other than `FULL_BUILD` are typically only useful when debugging a build-related problem.

4.2.3 Multi-Pass Place and Route

Multi-pass Place and Route (MPPR) is the term used for automatically running the place and route process on the same input design multiple times with different starting conditions in order to see which run gives the best results. `setMPPRCostTableSearchRange` instructs the Xilinx map and place and route tools to use a range of “cost tables” to initialize a multi-pass run:

void *setMPPRCostTableSearchRange(**int** min, **int** max)*

The number corresponds to a cost table index and results in different place and route strategies.

`setEnableTimingAnalysis` enables the output of a timing report by the place and route tools which can be used to identify timing issues which may help when optimizing the design:

void *setEnableTimingAnalysis(**boolean** v)*

5 Custom HDL Manager Blocks

For some specialized projects, you may need to integrate your own or third-party IP blocks written in another language into a MaxCompiler design. MaxCompiler allows the direct instantiation of VHDL or Verilog source code and compiled netlists as Manager blocks. These blocks can connect to other Manager blocks via streams and make use of the scalar IO and mapped memory APIs from the CPU.

MaxCompiler also provides an interface for running ModelSim to simulate the entire design. This includes the capability to read and write PCI stream data into the DFE, set scalar inputs and write to the mapped memory bus.

If developing a Manager block from scratch, it is recommended to use a Manager State Machine, as described in the State Machine Tutorial, as these are more tightly integrated into MaxCompiler.

If editing any HDL source outside of MaxIDE, you need to refresh or clean the project before rebuilding it to include your changes.

5.1 Overview

Integration of an IP block into a Manager requires three components:

- A custom HDL node to define the interface between the HDL block and MaxCompiler.
- The HDL block with a top-level that defines MaxCompiler inputs and outputs.
- A custom Manager that instantiates the custom HDL node and connects any stream inputs and outputs.

Example 1 is used in this overview to demonstrate the concepts and features being introduced. This example uses stream inputs and outputs in VHDL and integrates these into a MaxCompiler project.

The example has one stream input and one stream output. The output (`count`) is a counter that counts from 0 up to the maximum value specified by the input (`max`), at which point it holds the value for a number of output values before it counts back down to 0 again. It then repeats the cycle. It outputs a new value for the counter every time it receives an input. If the current counter value is larger than the current maximum value from the input stream, then the counter is set to this maximum value for the next output and held.

5.1.1 The Custom HDL Node

A custom HDL node defines the interface between MaxCompiler and an external IP block. The string names of all of the clock, reset, input and output signals in the custom HDL node match the names of the ports declared in the HDL code itself.

The source code for the custom HDL node used in Example 1 is shown in *Listing 18*.

Listing 18: Custom HDL Node wrapper for a VHDL counter (SimpleHDLNode.maxj).

```
/**
 * Document: Manager Compiler Tutorial (maxcompiler-manager-tutorial.pdf)
 * Chapter: 5        Example: 1       Name: Simple HDL
 * MaxFile name: SimpleHDL
 * Summary:
 *      Custom HDL node that defines the interface between MaxCompiler and an
 *      external IP block which implements a counter. The node has one stream
 *      input and one stream output. The output (count) is a counter that counts
 *      from 0 up to the maximum value specified by the input (max), at which point
 *      it holds the value for a number of output values before it counts back down
 *      to 0 again. It then repeats the cycle.
 */
package simplehdl;

import com.maxeler.maxcompiler.v2.managers.custom.CustomManager;
import com.maxeler.maxcompiler.v2.managers.custom.blocks.CustomHDLNode;

/*
 * Custom HDL node which includes some VHDL source.
 */
final class SimpleHDLNode extends CustomHDLNode{
    SimpleHDLNode(CustomManager manager, String instance_name) {
        super(manager, instance_name, ”simple_hdl”);

        CustomNodeClock clock = addClockDomain(”clk”);
        clock.setNeedsReset(”rst”);

        addInputStream(”max”, 32, clock, CustomNodeFlowControl.PULL, 1);
        addOutputStream(”count”, 32, clock, CustomNodeFlowControl.PUSH, 2);

        addVHDLSource(”SimpleHDLSource.vhdl”, false);
    }
}
```

A node inherits from `CustomHDLNode`:

```
final class SimpleHDLNode extends CustomHDLNode{
```

The constructor calls the superclass constructor with the Custom Manager instance, a string name for this instance of the node and a string which is the name of the top-level entity in the VHDL, in this case `"simple_hdl"`:

```
    SimpleHDLNode(CustomManager manager, String instance_name) {
        super(manager, instance_name, ”simple_hdl”);
```

The example has a single clock domain called `clock`, with a clock signal called `"clk"` and a reset signal called `"rst"`:

```
        CustomNodeClock clock = addClockDomain(”clk”);
        clock.setNeedsReset(”rst”);
```

Clocks and resets are described in more detail in *subsection 5.3*.
There are two types of stream input and output interface:

- Push: The source indicates that data is ready and requires that the data be accepted by the sink.
- Pull: The sink requests data from the source.

The signals and protocol for stream inputs and outputs are described in more detail in *subsection 5.4*.

The custom HDL node for this example has a pull input called `"max"`, which is declared to be 32-bits wide, in the clock domain `clock`, a pull interface and to have a latency of 1:

```
        addInputStream("max", 32, clock, CustomNodeFlowControl.PULL, 1);
```

Likewise, the node has a push output, called `"count"`, which is declared to be 32-bits wide, in the clock domain `clock`, a push interface and to have a latency of 2. The stall latency is 2 because the stall signal is registered in the HDL to improve timing:

```
        addOutputStream("count", 32, clock, CustomNodeFlowControl.PUSH, 2);
```

Finally, the VHDL source code is included:

```
        addVHDLSource("SimpleHDLSource.vhdl", false);
```

5.1.2 The VHDL Source

Listing 19 and *Listing 20* show the VHDL source for this example. The names of all of the inputs and outputs of this entity match the names used in the custom HDL node declaration (see *Listing 18*).

The entity for this counter is called `simple_hdl`:

```
entity simple_hdl is
```

Listing 19: Counter implemented in VHDL with a MaxCompiler pull input and push output (SimpleHDL-Source.vhdl).

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

entity simple_hdl is
    port (
        -- external IP block can have any number of clock domains
        -- each streaming interface is synchronous to one of the clock domains
        -- each clock domain has a synchronous reset (controllable by host software)
        clk : in std_logic;
        rst : in std_logic;

        -- pull input streaming interface "max" (synchronous to clock)
        max_empty : in std_logic;
        max_almost_empty : in std_logic;
        max_read : out std_logic;
        max_data : in std_logic_vector (31 downto 0);

        -- push output streaming interface "count" (synchronous to clock)
        count_valid : out std_logic;
        count_stall : in std_logic;
        count_data : out std_logic_vector (31 downto 0)
    );
end entity simple_hdl;

architecture rtl of simple_hdl is
    signal counter : unsigned(31 downto 0) := (others => '0');
    signal hold_counter : unsigned(31 downto 0) := (others => '0');
    constant hold_count : integer := 2;
    signal data_ready : std_logic := '0';
    signal output_data : unsigned(31 downto 0) := (others => '0');
    signal output_valid : std_logic := '0';
    signal stall_reg : std_logic := '1';
    type MODE_TYPE is ( COUNTING_UP, COUNTING_DOWN, HOLD );
    signal mode : MODE_TYPE;
begin
```

Listing 20: Counter implemented in VHDL with a MaxCompiler pull input and push output (SimpleHDL-Source.vhdl).

```

    -- set the read signal if the input is not empty and the output is not stalled
    max_read <= not max_empty and not stall_reg;

    counter_process : process(clk)
    begin
        if rising_edge(clk) then
            if rst = '1' then
                counter <= (others => '0');
                hold_counter <= (others => '0');
                data_ready <= '0';
                output_data <= (others => '0');
                output_valid <= '0';
                stall_reg <= '1';
                mode <= COUNTING_UP;
            else
                -- we will only have data if the input is not empty and the output is not stalled
                data_ready <= not max_empty and not stall_reg;

                stall_reg <= count_stall;
                output_valid <= '0';
                if (data_ready = '1') then
                    counter <= counter;
                    case mode is
                        when COUNTING_UP =>
                            if counter = unsigned(max_data) then
                                mode <= HOLD;
                            else
                                if counter > unsigned(max_data) then
                                    counter <= unsigned(max_data);
                                    mode <= HOLD;
                                else
                                    counter <= counter + 1;
                                end if;
                            end if;
                        when COUNTING_DOWN =>
                            if counter = 0 then
                                counter <= counter + 1;
                                mode <= COUNTING_UP;
                            else
                                counter <= counter - 1;
                            end if;
                        when others =>
                            if hold_counter = hold_count then
                                hold_counter <= (others => '0');
                                mode <= COUNTING_DOWN;
                            else
                                hold_counter <= hold_counter+1;
                            end if;
                    end case;
                    output_valid <= '1';
                    output_data <= counter;
                end if;
            end if;
        end if;
    end process;

    -- pass on the registered values of the outputs
    count_valid <= output_valid;
    count_data <= std_logic_vector(output_data);
end rtl;
```

This example has a single clock domain, with a clock input called `clk` and a reset called `rst`:

```
clk  :  in  std_logic ;
rst  :  in  std_logic ;
```

The signals for the input stream all have the name `max_` prepended to the name of the individual signal:

```
-- pull input streaming interface "max" (synchronous to clock)
max_empty : in std_logic ;
max_almost_empty : in std_logic;
max_read : out std_logic ;
max_data : in  std_logic_vector (31 downto 0);
```

Likewise, the signals for the output stream all have the name `count` prepended to the name of the individual signal:

```
count_valid  :  out std_logic ;
 count_stall  :  in  std_logic ;
count_data :  out std_logic_vector (31 downto 0)
```

The architecture of `simple_hdl` updates the counter whenever there is data at the input and the output is not stalled.

5.1.3 The Manager

The java source for the Custom Manager is shown in *Listing 21*.

The Manager instantiates an instance of the custom HDL node and wraps it in a Manager block:

```
CustomHDLNode hdlNode = new SimpleHDLNode(this, "simpleHDLNode");
CustomHDLBlock hdlBlock = addCustomHDL(hdlNode);
```

IO is then connected to this Manager block in the same way as other Manager blocks:

```
hdlBlock.getInput("max") <== addStreamFromCPU("max");
addStreamToCPU("count") <== hdlBlock.getOutput("count");
```

5.1.4 The Testbench

The Java source for running the simulation, abridged to show the relevant code, is shown in *Listing 22*.

The simulation runner creates an instance of the custom Manager with a target of HDL simulation:

```
EngineParameters engineParameters = new EngineParameters(
     "SimpleHDLSim", BOARDMODEL, Target.HDL_SIM);
SimpleHDLManager m = new SimpleHDLManager(engineParameters);
```

It then adds a build manager for a full build, which includes compiling the VHDL source:

```
m.setBuildConfig(new BuildConfig(Level.FULL_BUILD));
```

A testbench called `sim` is created:

```
HDLTestBench sim = m.getHDLTestBench();
```

The first step of the testbench is to reset the device, which passes the reset signal through to the included custom HDL node:

```
        sim.resetDevice();
```

It then streams data to and from the device via the PCI Express bus:

```
        sim.streamFromCPU("max", inDataPacked);

        sim.streamToCPU("count", DATA_SIZE / 2);

        sim.syncStreamFromCPU("max", 1000);
        sim.syncStreamToCPU("count", 1000);
```

The source for this example has the GUI disabled, so running the test launches ModelSim, runs the simulation and closes ModelSim once the simulation is complete:

```
        // Disable the ModelSim GUI
        sim.setEnableSimGUI(false);
```

The test is run using `runTest()`:

```
        // Run the simulation
        sim.runTest();
```

If the ModelSim GUI is enabled, running the test launches the ModelSim GUI and the simulation can be run manually. *Figure 14* shows a screen-shot of the "sim" window in ModelSim showing the path to the top-level entity instantiated from the custom HDL node.

The Java process completes when ModelSim finishes and the data in the PCIe stream out of the design is read back:

```
        List<Long> outDataPacked = sim.getOutputData("count");
```

5.2 Including IP Blocks

A custom HDL node can include IP blocks in three ways:

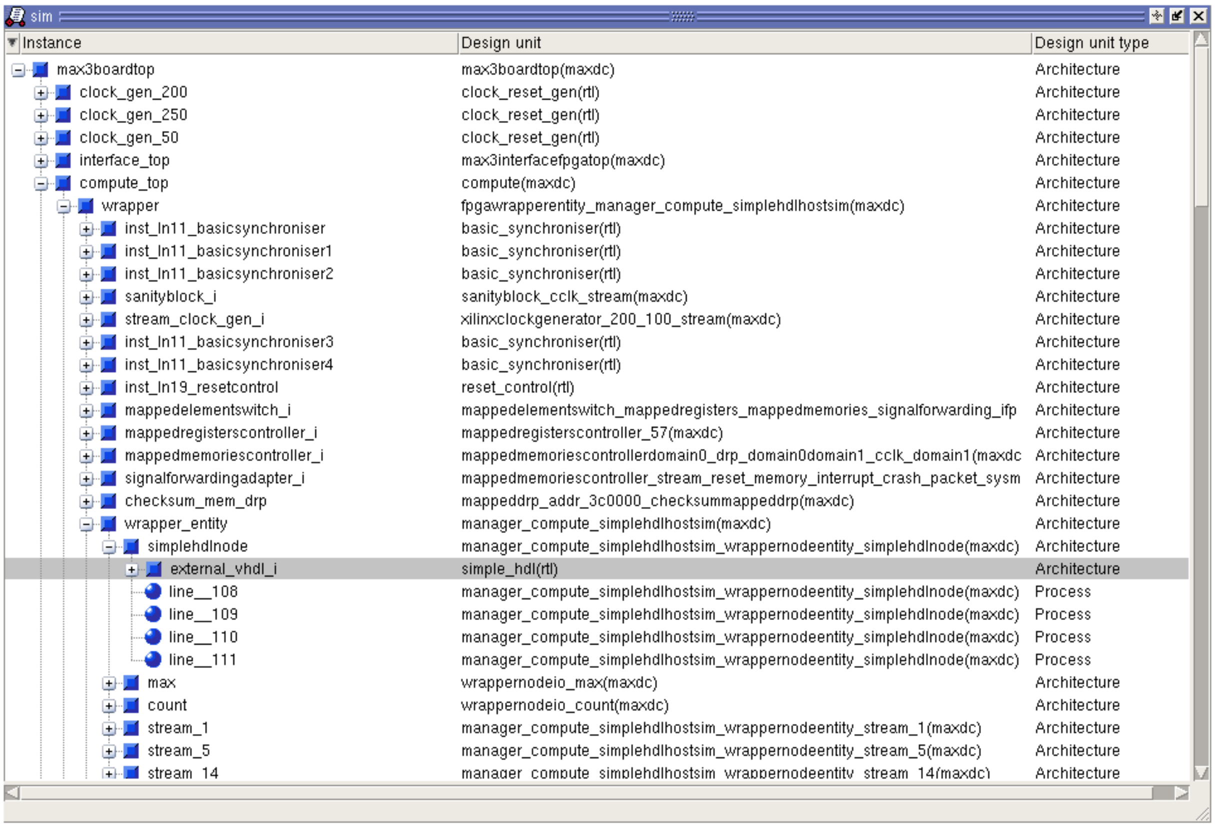
- *addVHDLSource(**String** name, **boolean** is_on_filesystem)*: adds VHDL source code.
- *addVerilogSource(**String** name, **boolean** is_on_filesystem)*: adds Verilog source code.
- *addNetlist(**String** name, **boolean** is_on_filesystem)*: adds an EDIF or Xilinx `.ngc` netlist.

In each case, the `is_on_filesystem` argument specifies whether this is an absolute path to the file on the file system (`is_on_filesystem==true`) or is in the source tree (`is_on_filesystem==false`).

Depending on the simulated DFE model your custom HDL block appears in the design hierarchy in the following location, where `"CustomHDLNodeInstanceName"` is the name of the instance of the custom HDL node declared in the Manager.
Vectis based DFEs:

```
max3boardtop->compute_top->wrapper->wrapper-entity->
    CustomHDLNodeInstanceName->external_vhdl_i
```

Coria based DFEs:



sim

Instance	Design unit	Design unit type
max3boardtop	max3boardtop(maxdc)	Architecture
clock_gen_200	clock_reset_gen(rtl)	Architecture
clock_gen_250	clock_reset_gen(rtl)	Architecture
clock_gen_50	clock_reset_gen(rtl)	Architecture
interface_top	max3interfacefpgatop(maxdc)	Architecture
compute_top	compute(maxdc)	Architecture
wrapper	fpgawrapperentity_manager_compute_simplehdlhostsim(maxdc)	Architecture
inst_ln11_basicsynchroniser	basic_synchroniser(rtl)	Architecture
inst_ln11_basicsynchroniser1	basic_synchroniser(rtl)	Architecture
inst_ln11_basicsynchroniser2	basic_synchroniser(rtl)	Architecture
sanityblock_i	sanityblock_cclk_stream(maxdc)	Architecture
stream_clock_gen_i	xilinxclockgenerator_200_100_stream(maxdc)	Architecture
inst_ln11_basicsynchroniser3	basic_synchroniser(rtl)	Architecture
inst_ln11_basicsynchroniser4	basic_synchroniser(rtl)	Architecture
inst_ln19_resetcontrol	reset_control(rtl)	Architecture
mappedelementswitch_i	mappedelementswitch_mappedregisters_mappedmemories_signalforwarding_ifp	Architecture
mappedregisterscontroller_i	mappedregisterscontroller_57(maxdc)	Architecture
mappedmemoriescontroller_i	mappedmemoriescontrollerdomain0_drp_domain0domain1_cclk_domain1(maxdc	Architecture
signalforwardingadapter_i	mappedmemoriescontroller_stream_reset_memory_interrupt_crash_packet_sysm	Architecture
checksum_mem_drp	mappeddrp_addr_3c0000_checksummappeddrp(maxdc)	Architecture
wrapper_entity	manager_compute_simplehdlhostsim(maxdc)	Architecture
simplehdlnode	manager_compute_simplehdlhostsim_wrappernodeentity_simplehdlnode(maxdc)	Architecture
external_vhdl_i	simple_hdl(rtl)	Architecture
line__108	manager_compute_simplehdlhostsim_wrappernodeentity_simplehdlnode(maxdc)	Process
line__109	manager_compute_simplehdlhostsim_wrappernodeentity_simplehdlnode(maxdc)	Process
line__110	manager_compute_simplehdlhostsim_wrappernodeentity_simplehdlnode(maxdc)	Process
line__111	manager_compute_simplehdlhostsim_wrappernodeentity_simplehdlnode(maxdc)	Process
max	wrappernodeio_max(maxdc)	Architecture
count	wrappernodeio_count(maxdc)	Architecture
stream_1	manager_compute_simplehdlhostsim_wrappernodeentity_stream_1(maxdc)	Architecture
stream_5	manager_compute_simplehdlhostsim_wrappernodeentity_stream_5(maxdc)	Architecture
stream_14	manager_compute_simplehdlhostsim_wrappernodeentity_stream_14(maxdc)	Architecture

Figure 14: Screen-shot of ModelSim showing the design hierarchy.

```
max4boardtop->inst_ln1049_hdltestbench->MAX4FabricTop_i->wrapper->
    wrapper_entity->CustomHDLNodeInstanceName->external_vhdl_i
```

5.3 Clocks and Resets

A custom HDL node has one or more clock domains.

A clock domain is added to a custom HDL node using the method `addClockDomain`, with a name for the clock signal that matches the name of the signal in the VHDL:

*CustomNodeClock addClockDomain(**String** name)*

This returns a `CustomNodeClock` object that can be passed to other methods on the custom HDL node to associate interfaces with this clock domain.

A reset is added to a clock domain using the method `setNeedsReset` on a `CustomNodeClock` object, with a name for the reset signal that matches the name of the signal in the VHDL:

*setNeedsReset(**String** reset_name)*

Example 1 has a clock domain called `clock`, with a clock signal called `"clk"` and a reset signal called `"rst"`:

```
        CustomNodeClock clock = addClockDomain("clk");
        clock.setNeedsReset("rst");
```

5.3.1 Reset Behavior

The reset signal is generated in a 50 MHz clock domain in the Manager. This signal is then synchronized once for each clock domain in the design.

The reset signal is held high for 32 clock cycles of the 50 MHz domain.

5.4 Stream Interfaces

Stream interfaces are synchronous to a clock specified in the custom HDL node declaration.

There are two types of stream input and output interface:

- Push: The source indicates that data is ready and requires that the data be accepted by the sink.
- Pull: The sink requests data from the source.

5.4.1 Push IO

The control signals for a push input and output are shown in *Figure 15*. The source sets the data and sets the valid signal to `1` on the same cycle and these must be read by the sink in the same cycle.

If the sink does not read the data when the valid signal is set to `1`, then the data is lost.

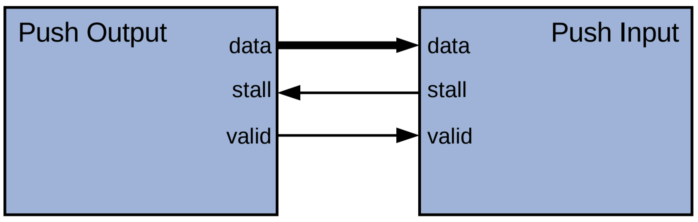


Figure 15: Data and control signals for a push interface.

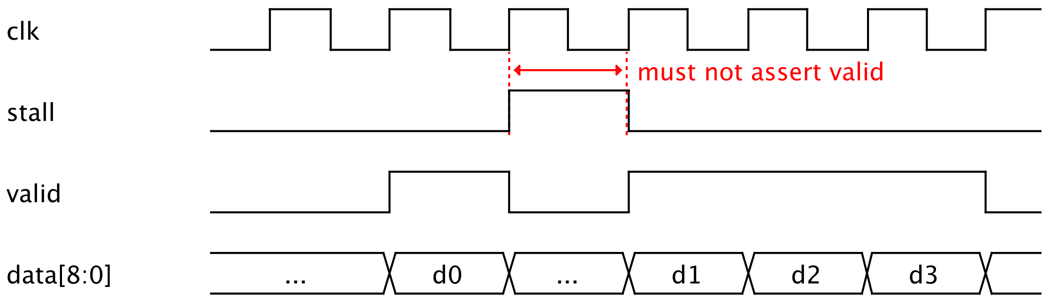


Figure 16: Timing diagram for a push interface with a latency of 0.

5.4.2 Stall Latency

Both push inputs and push outputs specify a stall latency. For an input, this is the number of data inputs that it can receive after the stall signal is set to `1`. For an output, this is the number of data outputs that it produces after receiving the stall signal.

For a stall latency of 0, the valid signal must be set to `0` in the *same* clock cycle as the stall signal is set to `1`. When the stall signal is set to `0`, the source can start producing data in the same cycle.

Figure 16 shows a timing diagram for a push interface with a latency of 0.

For a stall latency of N, the sink must be prepared to receive N more data items when the stall signal is set to `0`.

Figure 17 shows a timing diagram for a push interface with a latency of 1.

If a push input with a stall latency of N data items is connected to a push output specifying a stall latency of M data items and $M > N$, the Manager Compiler inserts buffering before the input to ensure that no data is dropped.

5.4.3 Pull IO

The control signals for a pull input and output are shown in *Figure 18*.

The sink sets the read signal to `1` and the source responds *the following cycle* by setting the data. If the source sets the empty signal to `1`, the sink must not request data (i.e. must set the read signal to `0`) in the *same* cycle.

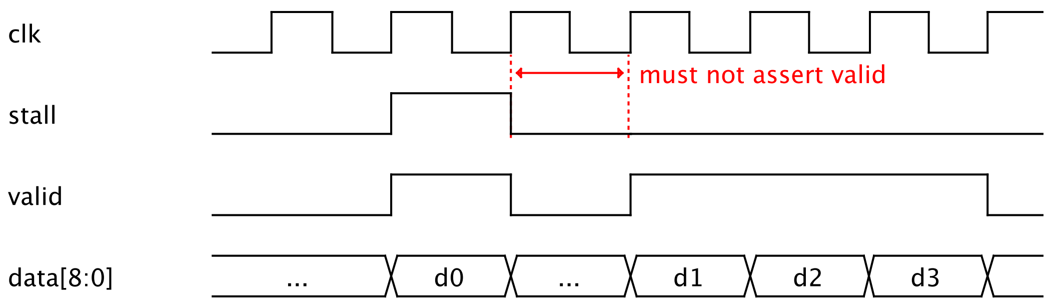


Figure 17: Timing diagram for a push interface with a latency of 1.

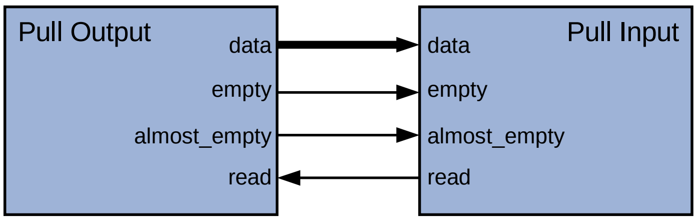


Figure 18: Data and control signals for a pull interface.

5.4.4 Almost Empty Signal

The source can set the almost empty signal to 1 to indicate that it is almost out of data. The **almost empty threshold** on an output defines the number of data items remaining at the source, below which the source sets the almost empty signal.

The almost empty threshold on an input defines the minimum number of remaining data items at the source that the sink requires when the signal is set. This can be used, for example, if the sink reads in bursts of a number of data items and therefore requires that at least the burst-size of data is available when it starts reading.

The default almost empty threshold is 1.

Figure 19 shows a timing diagram for a pull interface.

5.4.5 Pull to Push Example

Example 1 shows the use of a pull input and a push output in VHDL. *Listing 19* and *Listing 20* show the VHDL source for this example.

The entity for this counter is called `simple_hdl`:

```
entity simple_hdl is
```

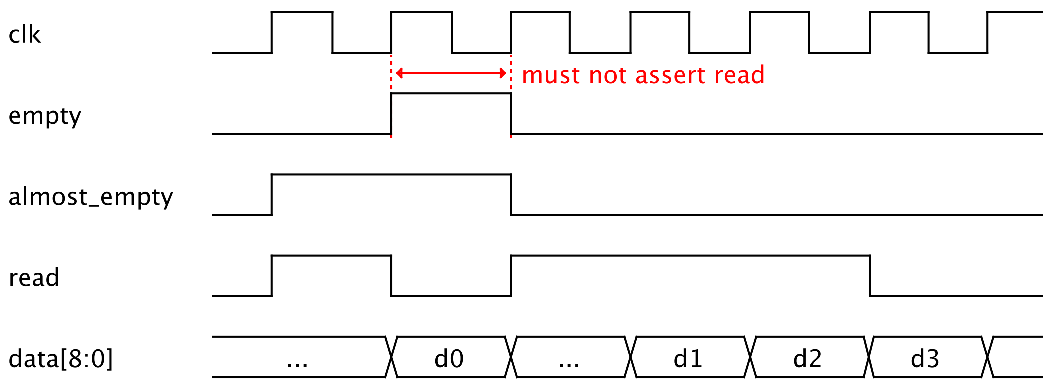


Figure 19: Timing diagram for a pull interface.

This example has a single clock domain, with a clock input called `clk` and a reset called `rst`:

```
        clk  :  in  std_logic ;
        rst  :  in  std_logic ;
```

The ports for the input stream all have the name `max` prepended to the name of the individual signal:

```
        -- pull  input  streaming interface  "max" (synchronous to clock)
        max_empty : in std_logic ;
        max_almost_empty : in std_logic;
        max_read : out std_logic ;
        max_data : in  std_logic_vector (31 downto 0);
```

Likewise the ports for the output stream all have the name `count` prepended to the name of the individual signal:

```
        count_valid  :  out std_logic ;
        count_stall  :  in  std_logic ;
        count_data :  out  std_logic_vector (31  downto 0)
```

The stall signal is delayed for 1 cycle so that the output is not directly connected to the input as this might make it difficult to meet timing:

```
                stall_reg   <= count_stall;
```

A read from the pull input is requested only if the input is not empty and the output is not stalled:

```
    max_read <= not max_empty and not stall_reg;
```

The process `counter_process` implements the counter logic. The status of the read request is registered using a signal:

```
                data_ready <= not max_empty and not stall_reg;
```

The counter is then conditionally updated whenever new data was requested the previous cycle:

```
                    if (data_ready = '1') then
```

5.4.6 Selecting IO Types

You can choose the input and output types that are best suited to the IP block that you are integrating.

If the stall latencies between a push output and a push input don't match, then the Manager Compiler inserts extra buffering to ensure that no data is lost between the output and the input. Likewise, if the almost empty thresholds between a pull output and a pull input do not match, the Manager Compiler inserts extra buffering to ensure that the receiver gets its required almost empty latency. Where a push IO is connected to a pull IO, the Manager Compiler inserts an adapter and some buffering if required.

When choosing which IO type to use, consider the following guidelines:

- Is the IP block passive or active? For example, a FIFO would more naturally have a push input and a pull output. A decoder or encoder that produces a different amount of data than it consumes would more naturally have a pull input and a push output.
- Where are any buffers required in the system? Using a pull input removes the requirement for buffering data at an input in the design of an IP block and, likewise, using a push output removes the requirement for buffering data at an output in the design.

5.5 Simulating a Design

MaxCompiler includes the ability to drive a simulation of a full design using ModelSim.

5.5.1 Requirements

Mentor ModelSim DE 10.0a or greater is required to simulate a full MaxCompiler design in ModelSim.

See the MaxCompiler Install Guide for instructions on setting up ModelSim for use with MaxCompiler.

5.5.2 MaxCompiler Testbenches

The MaxCompiler `HDLTestBench` allows non-interactive driving of the HDL simulation. A simulation runner Java class is written to describe a sequence of interactions with the simulator which is then compiled into a script to drive the simulation.

Calling `runTest()` launches the simulation with this script and waits for the simulation to complete. When the simulation ends, `runTest()` returns and any output from PCI Express streams can be read in the Java.

5.5.3 Resetting the Device

Calling `resetDevice()` sends the reset signal to all the Manager blocks in a custom Manager, in the same way as calling `max_reset_device` in the host code.

5.5.4 Streaming Data in and out of the Simulation

Data is read from and written to PCI Express streams in 64-bit words. Streams that are of a different width must be packed into 64-bit words before being written into the stream and unpacked when read out.

Reading and writing of data to streams is asynchronous: the read or write set up via one method call, then the testbench can wait for the transfer to complete via another.

For data from the CPU to the DFE, the two methods are:

*streamFromCPU(**String** name, List<Long> data)*
*syncStreamFromCPU(**String** name, **int** timeout)*

For data from the DFE to the CPU, the two methods are:

*streamToCPU(**String** name, **int** size_in_quads)*
*syncStreamToCPU(**String** name, **int** timeout)*

Timeouts are specified as a number of clock cycles of the PCI Express clock.

After the simulation has completed, the data from an output stream can be read back, using `getOutputData`:

*List<Long> getOutputData(**String** output_name)*

5.5.5 Delays

There are two different methods of introducing delays in a testbench:

- waitCycles(int num_cycles): Wait for a number of cycles of the PCI Express clock.
- waitNanoSeconds(int time): Wait for a number of nanoseconds.

5.5.6 Simulation of LMem

Simulation of a full design includes simulation of the LMem controller and accurate simulation of the LMem itself. The LMem is only simulated if the MaxCompiler detects that the design makes use of it.

Simulating a design using LMem in ModelSim can take twenty minutes or more on a Vectis before the memory can be accessed, because the memory takes around 75 μs of simulation time to be initialized ($\sim$ 240 μs on a MAX2).

To wait for a memory interrupt in the testbench, you can use `waitForMemoryInterrupt`, with a timeout specified in clock cycles:

*waitForMemoryInterrupt(**int** timeout)*

This does the same as calling `max_memory_stream_interrupt_any` followed by `max_wait_for_interrupt` in the host code.

5.5.7 Running ModelSim

By default, running the test launches ModelSim without the GUI, runs the simulation and closes ModelSim once the simulation is complete.

The ModelSim GUI can also be explicitly enabled and disabled:

*setEnableSimGUI(**boolean** enabled)*

If the ModelSim GUI is enabled, running the test launches the ModelSim GUI and the simulation must be run manually.

An assertion is made when the simulation completes successfully and the simulation ends with the following message:

```
VSIM 3> run -all
# ** Failure: Finished!
```

If any PCIe stream synchronization or wait on a memory interrupt times out, then the simulation ends immediately with the following message:

```
# ** Failure: Failed to get interrupt(s).
```

Exiting ModelSim allows the Java process to complete and any data in PCIe streams out of the design is read back.

5.6 Scalar IO

Scalar input and outputs are used to transfer single values between the host and the HDL block. There is no handshaking of values in either direction and values are not restricted to being changed at any specific time. There is therefore the risk of reading unstable values in each direction.

5.6.1 Scalar Inputs

A scalar input is an asynchronous input that can be set from the host, therefore it should be registered one or more times before the data is used to avoid potentially introducing metastability into the design. This does not, however, eliminate the risk of reading the wrong value during transition, so the application must ensure that scalar input values are not changed while being read. Typically this is achieved by only setting scalar inputs when the Manager block is in a known, stable state, for example if there is no data streaming through the block. A robust method is to set the scalar input and then reset the design: the scalar input itself is not reset and is stable well before the Manager block comes out of reset (see *subsubsection 5.3.1*).

A scalar input is added to a custom HDL node using the method `addScalarInput`, with the name of the input and the width of the data:

*addScalarInput(**String** name, **int** width)*

The name of the input in the Java must match the name of the input in the HDL source.

Example 2 shows the use of a scalar input. The example extends the code from Example 1 by adding a scalar input called `hold_count` that sets the number of outputs that the maximum value of the counter should be held for.

The 32-bit scalar input is added in the custom HDL node:

```
        addScalarInput("hold_count", 32);
```

This matches the input specified in the VHDL entity declaration:

```
        hold_count : in std_logic_vector (31 downto 0);
```

An additional process clocks this input through a register before it is used in the main counter process:

```
    scalar_register_process : process(clk)
    begin
        if rising_edge(clk) then
            hold_count_reg <= unsigned(hold_count);
        end if;
    end process;
```

5.6.2 Scalar Outputs

A scalar output from the HDL block to the host is set synchronously to a clock domain of the HDL block: the Manger implements logic to avoid reading an unstable value. The Manager can, however, read the wrong value during a transition, so the application must ensure that the value is not changed while it is being read. Typically this is achieved by only reading back scalar outputs when the Manager block is in a known, stable state, for example if there is no data streaming through the block.

A scalar output is added to a custom HDL node using the method `addScalarOutput`, with the name of the output and the width of the data:

*addScalarOutput(**String** name, **int** width)*

The name of the output in the Java must match the name of the output in the HDL source.

✷ You cannot read scalar outputs using the Java testbench API.

5.7 Mapped Memory Interface

The mapped memory interface is used for setting and reading back mapped memories in MaxCompiler designs, but the bus can be used for any purpose in a custom HDL component.

5.7.1 Signals and Timing

There is a mapped memory bus per clock domain. Each bus is a master-slave bus, where the Manager itself is the master and multiple Manager blocks can be slaves on the bus. There is a single 16-bit address bus and two data buses, one for read data and one for write data, both of which are 36 bits wide. The same signals are connected to every slave on the bus, with a unique enable signal per slave. The name of the per-slave enable signal has the string `mapped_mem_en_` prepended to a name given to the mapped memory bus in the custom HDL node declaration. *Figure 20* shows the mapped memory bus signals for two Manager blocks in the same clock domain.

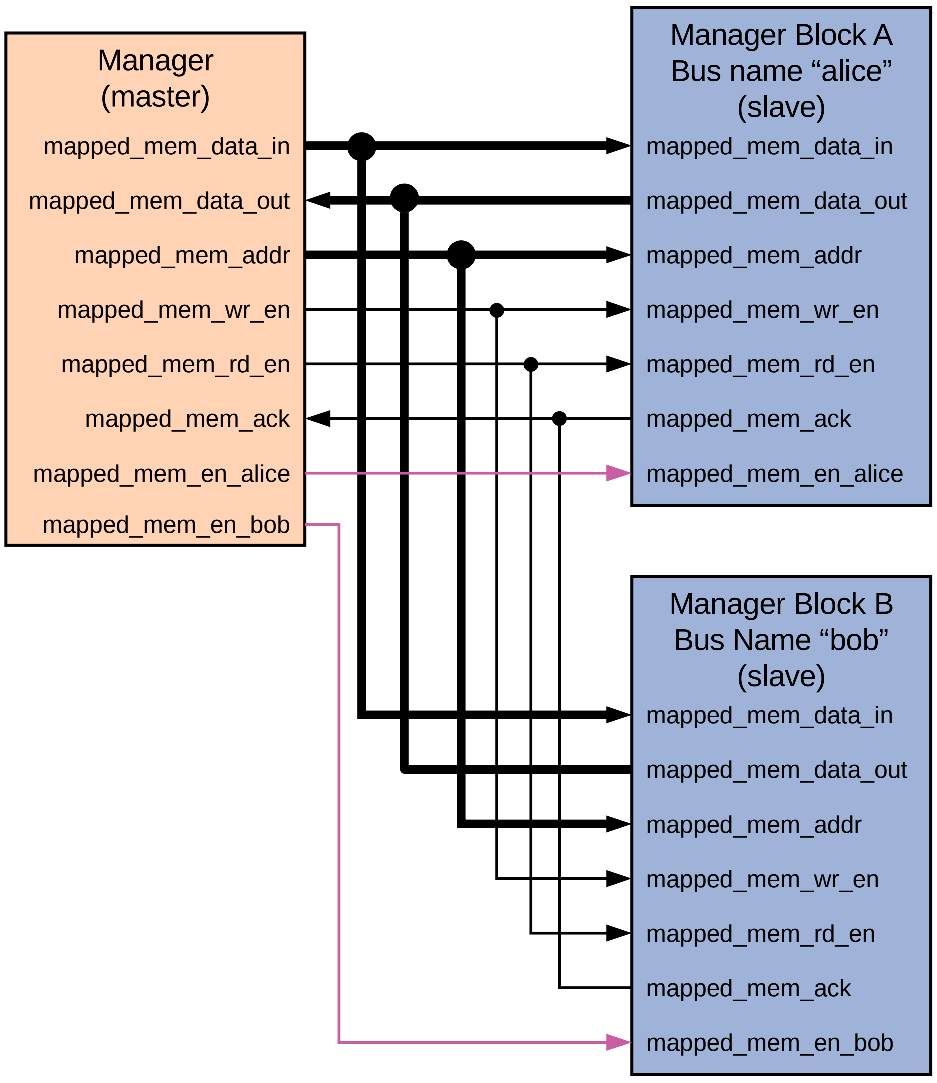


Figure 20: Mapped memory bus for a single clock domain.

The `mapped_mem_data_out` and `mapped_mem_ack` signals from all Manager blocks in the same clock domain are ORed together, so a Manager block must always set these to 0 when not responding to a data request.

To read or write data from a particular Manager block, the Manager asserts the enable signal for that specific Manager block, along with the address, data and read/write enable signals. The Manager block must respond *two* or more cycles later by asserting the `mapped_mem_ack` signal, and the data in the case of a read, for a single cycle.

The `mapped_mem_ack` signal must not be asserted until at least 2 cycles after the read or write request.

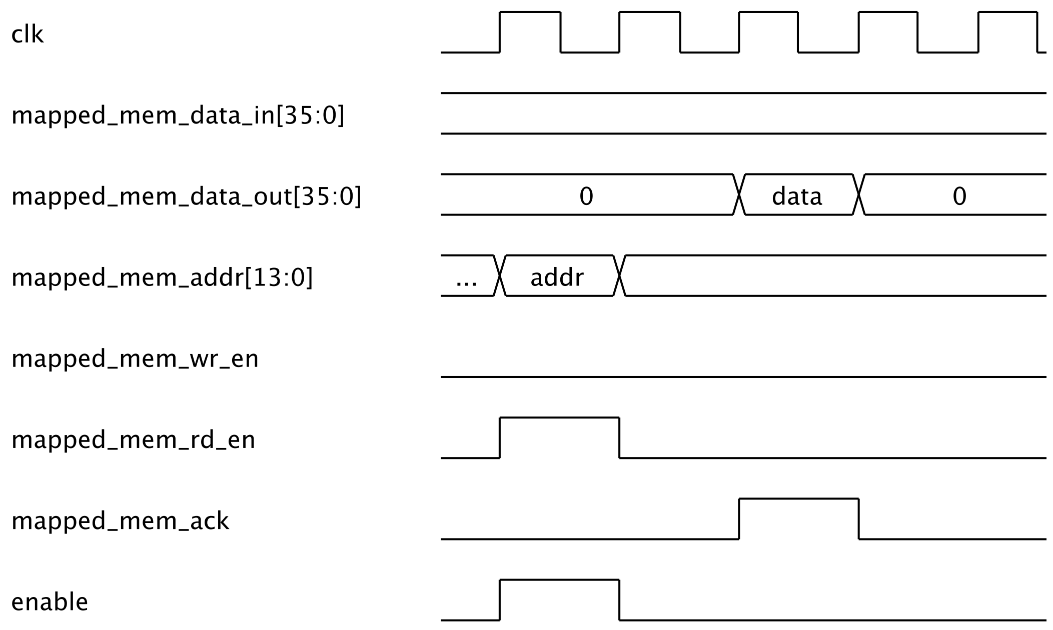


Figure 21: Timing diagram for mapped memory read from the CPU.

The Manager reads or writes one word at a time and waits until it has received acknowledgment for each word before reading or writing another word to any of the Manager blocks connected to the bus.

Figure 21 shows a timing diagram for a read from the mapped memory bus.

Figure 22 shows a timing diagram for a write to the mapped memory bus.

5.7.2 Reading and Writing to the Bus

From the host, mapped memory is read and written using the normal SLiC API methods:

```
void max_set_mem_uint64t(max_actions_t *actions, const char *block_name, const char *mem_name, size_t index, uint64_t value);
void max_set_mem_double (max_actions_t *actions, const char *block_name, const char *mem_name, size_t index, double value);

void max_get_mem_uint64t(max_actions_t *actions, const char *block_name, const char *mem_name, size_t index, uint64_t *value);
void max_get_mem_double (max_actions_t *actions, const char *block_name, const char *mem_name, size_t index, double *value);
```

In simulation, a mapped memory can be set using the method `setMappedMemory` with the name of the mapped memory, the first address to set and a list of one or more values to set the contents:

```
setMappedMemory(String name, long addr, List<Long> data)
```

The 64-bit arguments for address and data are truncated to the appropriate width (16 bits for the address and 36 bits for the data).

For example, to set a single value of `0xdeadbeefL` at address 3:

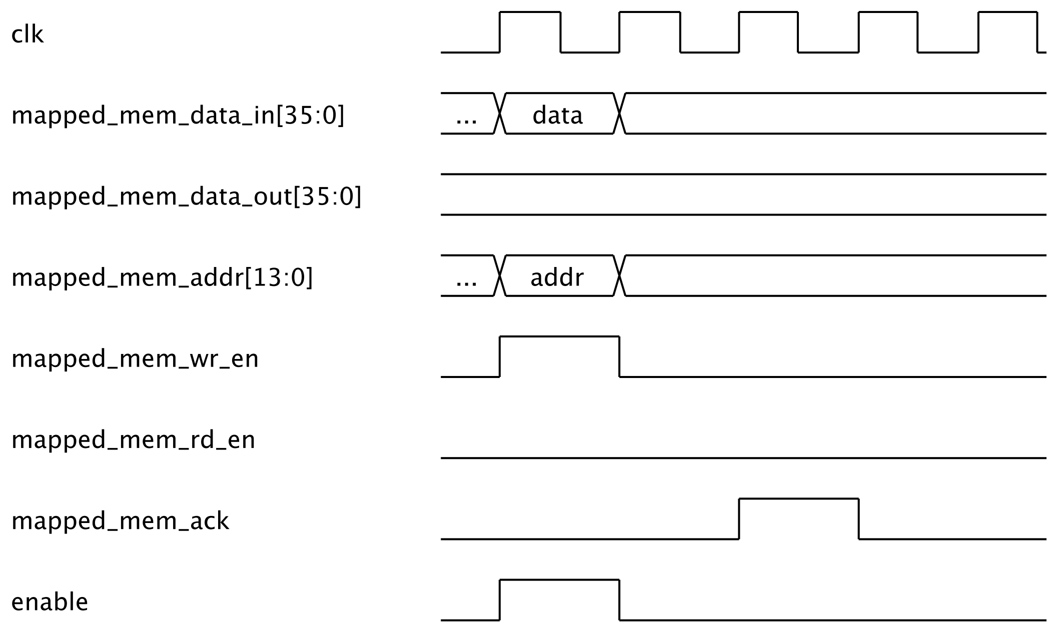


Figure 22: Timing diagram for mapped memory write from the CPU.

```
sim.setMappedMemory("myManagerBlock.busName", 3, Arrays.asList(0xdeadbeefL));
```

Alternatively, to set a range of addresses with some data, starting at address 3:

```
List<Long> data = new ArrayList<Long>();
for (long i=0; i < 8; i++)
    data.add(i);

sim.setMappedMemory("myManagerBlock.busName", 3, data);
```

This sets the value at address 3 to 0, address 4 to 1, address 5 to 2, etc., up to address 11.

✷ You cannot read mapped memories using the Java testbench API.

5.7.3 Mapped Memory Bus Example

Example 3 demonstrates the use of the mapped memory bus. This example uses writes to the mapped memory bus to stop and start an output stream and to set the output value of the stream. Mapped memory bus reads simply return back the read address as the output data.

Mapping the Ports The custom HDL node is shown in *Listing 23*. The VHDL is split between *Listing 24* and *Listing 25*.

The custom HDL node declares a clock called `clk` with a reset signal called `rst`:

```
        CustomNodeClock m_clk;

        m_clk = addClockDomain(”clk”);
        m_clk.setNeedsReset(”rst”);
```

These match the clock and reset ports in the VHDL source:

```
        clk  :  in  std_logic ;
        rst  :  in  std_logic ;
```

The mapped memory bus is set to use this clock and given the name for the Manager block on the bus of `example`:

```
        setMappedMemoryClock(m_clk);
        addMappedMemoryInterface(”example”);
```

The mapped memory bus is made up of 7 ports in the VHDL entity, the first 6 of which have the same name for all Manager blocks:

```
        mapped_mem_data_in : in std_logic_vector(35 downto 0);
        mapped_mem_addr : in   std_logic_vector (15 downto 0);
        mapped_mem_wr_en : in  std_logic ;
        mapped_mem_rd_en : in  std_logic ;
        mapped_mem_ack  :  out std_logic ;
        mapped_mem_data_out: out std_logic_vector(35 downto 0);
```

The last mapped memory port has a unique name for this custom HDL node, `mapped_mem_en_example` which is derived from the name of the mapped memory bus defined in the custom HDL node:

```
        mapped_mem_en_example : in std_logic
```

The Read and Write Processes The example uses a set of registers to delay the responses to the read and write requests:

```
    signal mapped_mem_wrack_reg : std_logic := '0';
    signal mapped_mem_rdack_reg : std_logic := '0';
    signal mapped_mem_ack_reg : std_logic := '0';

    signal mapped_mem_data_out_reg : std_logic_vector(35 downto 0);
    signal mapped_mem_data_out_reg_reg : std_logic_vector(35 downto 0);
```

There are three processes for this example:

- A process for handling the reads from the bus.
- A process for handling the writes to the bus.
- A process for delaying the acknowledgment of a read or a write and the setting of the data until the appropriate clock cycle.

The read process simply returns the address of the read request as the data whenever the read enable signal and Manager block enable signal are high:

```
process(clk)
begin
    if (rising_edge(clk)) then
        mapped_mem_rdack_reg <= '0';
        mapped_mem_data_out_reg <= (others => '0');

        if (rst = '1') then
        -- If this Manager block is enabled and a read is requested
        elsif (mapped_mem_en_example = '1' and mapped_mem_rd_en = '1') then
            mapped_mem_data_out_reg <= "0000000000000000000" & mapped_mem_addr;
            mapped_mem_rdack_reg <= '1';
        end if;
    end if;
end process;
```

When the write enable signal and Manager block enable signal are high, the write process either sets the output stream data or enables the stream, depending on the address:

```
process(clk)
begin
    if (rising_edge(clk)) then
        mapped_mem_wrack_reg <= '0';

        if (rst = '1') then
            output_enable <= '0';
            output_value  <= (others => '0');
        -- If this Manager block is enabled and a write is requested
        elsif (mapped_mem_en_example = '1' and mapped_mem_wr_en = '1') then
            case mapped_mem_addr is
            when "0000000000000000" =>
                -- Enable the output
                output_enable <= mapped_mem_data_in(0);
            when "0000000000000001" =>
                -- Set the output to the data from the mapped memory bus
                output_value <= mapped_mem_data_in(31 downto 0);
            when others =>
                null;
            end case;

            mapped_mem_wrack_reg <= '1';
        end if;
    end if;
end process;
```

The Simulation Testbench The testbench source is shown in *Listing 26*.

The test first resets the device:

```
        // Reset the device
        sim.resetDevice();
```

The mapped memory bus is first used to set the value for the output stream to `0xdeadbeefL` by sending that value to address `1`:

```
        // Set the output value
        sim.setMappedMemory("hdl_node_mapped_mem_bus.example", 1, Arrays
            .asList(testConstant));
```

Then the output stream is enabled by writing `1L` to address `0`:

```
        // Start the output stream
        sim.setMappedMemory("hdl_node_mapped_mem_bus.example", 0, Arrays
            .asList(1L));
```

Running on a DFE From the VHDL listing, we see that when a reset is issued, the output-enable is turned off.

```
    process(clk)
    begin
        if (rising_edge(clk)) then
            mapped_mem_wrack_reg <= '0';

            if (rst = '1') then
                output_enable <= '0';
                output_value  <= (others => '0');
            -- If this Manager block is enabled and a write is requested
            elsif (mapped_mem_en_example = '1' and mapped_mem_wr_en = '1') then
                case mapped_mem_addr is
                when "0000000000000000" =>
                    -- Enable the output
                    output_enable <= mapped_mem_data_in(0);
                when "0000000000000001" =>
                    -- Set the output to the data from the mapped memory bus
                    output_value <= mapped_mem_data_in(31 downto 0);
                when others =>
                    null;
                end case;

                mapped_mem_wrack_reg <= '1';
            end if;
        end if;
    end process;
```

The default SLiC actions implicitly include a reset which we need to suppress for this example. Since that is not possible within the SLiC Basic Interface, the Dynamic Interface is used in the CPU code, and there is no SLiC engine interface built from the Manager code, `MappedMemBusHDLManager.maxj`.

The full CPU code listing is shown in *Listing 27*.

The test first allocates memory for the data arrays and actions, initializes the `maxfile`, and allocates and loads the `maxfile` onto a DFE `engine`.

```
    uint32_t *outputData   = malloc(size * sizeof(uint32_t));
    uint32_t *expectedData = malloc(size * sizeof(uint32_t));
    uint64_t *mem = malloc(numMemRead * sizeof(uint64_t));

    max_file_t     *maxfile = MappedMemBusHDL_init();
    max_engine_t *engine  = max_load(maxfile, "*");
    max_actions_t *actions;
```

Next the read-back operation of the VHDL is tested: the actions are initialized, and `NUM_MEM_READ` mapped-memory read commands are added to the actions. Note that a reset command is always implicitly added to the actions. The actions are then submitted to the DFE to be run, after which they are freed.

```
actions = max_actions_init(maxfile, NULL);
for (int i = 0; i < numMemRead; i++) {
    max_get_mem_uint64t(actions, "hdl_node_mapped_mem_bus", "example", i, &mem[i]);
}
max_run(engine, actions);
max_actions_free(actions);
```

The call used in this example to run the actions

max_run(max_engine_t ∗engine, max_actions_t ∗actions);

is a blocking call, and so when it returns, the mapped-memory values read from VHDL are available to be used. They are checked against their expected value; we recall that the VHDL read process simply returns the address of the read request as the data whenever the read enable signal and Manager block enable signal are high:

```
int status = 0;
for (int i = 0; i < numMemRead; i++) {
    if (mem[i] != (uint64_t) i) {
        fprintf (stderr, "error: expected %d.\n", i);
        status = 1;
    }
}
```

The VHDL write process is tested next. From the VHDL we see that the order in which the mapped-memories are written is critical: address `1` must be written first to set the output stream data, and then address `0` is written to enable the stream. The `actions` are initialized as before, but this time the function

max_disable_reset(max_actions_t ∗actions);

is called, to prevent the reset command from being issued. The mapped-memories are set in the correct order, and an output stream is queued to receive the data from the VHDL. As before, the actions are submitted to be run on the DFE, and then freed.

```
actions = max_actions_init(maxfile, NULL);
max_disable_reset(actions);
max_set_mem_uint64t(actions, "hdl_node_mapped_mem_bus", "example", 1, testValue);
max_set_mem_uint64t(actions, "hdl_node_mapped_mem_bus", "example", 0, 1);
max_queue_output(actions, "output", outputData, size * sizeof(uint32_t));
max_run(engine, actions);
max_actions_free(actions);
```

Finally, the data streamed into the `outputData` array from the VHDL is checked to ensure it matches the expected data.

```
status = status | check(outputData, expectedData, size);
```

Listing 21: Custom Manager that includes a custom HDL node (SimpleHDLManager.maxj).

```
/**
 * Document: Manager Compiler Tutorial (maxcompiler-manager-tutorial.pdf)
 * Chapter: 5        Example: 1       Name: Simple HDL
 * MaxFile name: SimpleHDL
 * Summary:
 *       Manager design that creates an instance of the custom HDL node and
 *       wraps it in a Manager block. All IO is between the CPU and the DFE.
 */
package simplehdl;

import com.maxeler.maxcompiler.v2.build.EngineParameters;
import com.maxeler.maxcompiler.v2.managers.custom.CustomManager;
import com.maxeler.maxcompiler.v2.managers.custom.blocks.CustomHDLBlock;
import com.maxeler.maxcompiler.v2.managers.custom.blocks.CustomHDLNode;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.CPUTypes;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.EngineInterface;
import com.maxeler.maxcompiler.v2.managers.engine_interfaces.InterfaceParam;

class SimpleHDLManager extends CustomManager {

    SimpleHDLManager(EngineParameters engineParameters) {
        super(engineParameters);

        CustomHDLNode hdlNode = new SimpleHDLNode(this, "simpleHDLNode");
        CustomHDLBlock hdlBlock = addCustomHDL(hdlNode);

        hdlBlock.getInput("max") <== addStreamFromCPU("max");
        addStreamToCPU("count") <== hdlBlock.getOutput("count");
    }

    private static EngineInterface interfaceDefault () {
        EngineInterface ei = new EngineInterface();

        InterfaceParam n = ei.addParam("N", CPUTypes.INT);

        ei.setStream("max", CPUTypes.UINT32, n * CPUTypes.UINT32.sizeInBytes());
        ei.setStream("count", CPUTypes.UINT32, n * CPUTypes.UINT32.sizeInBytes());

        return ei;
    }

    public static void main(String[] args) {
        SimpleHDLManager m = new SimpleHDLManager(new EngineParameters(args));
        m.createSLiCinterface(interfaceDefault());
        m.build();
    }


}
```

Listing 22: Simulation runner for the ModelSim simulation of the complete design using a custom HDL node (SimpleHDLSimRunner.maxj).

```
/*
 * Simulation runner for ModelSim simulation of a complete device.
 */
public class SimpleHDLSimRunner {

    private static final int DATA_SIZE = 256;
    private static final int holdCount = 2;

    static int [] inData = new int[DATA_SIZE];
    static ArrayList<Long> inDataPacked = new ArrayList<Long>();

    private static final DFEModel BOARDMODEL = DFEModel.VECTIS;

    public static void main(String args[]) {
        EngineParameters engineParameters = new EngineParameters(
            "SimpleHDLSim", BOARDMODEL, Target.HDL_SIM);
        SimpleHDLManager m = new SimpleHDLManager(engineParameters);
        m.setBuildConfig(new BuildConfig(Level.FULL_BUILD));

        GenerateData();

        HDLTestBench sim = m.getHDLTestBench();

        sim.resetDevice();

        sim.streamFromCPU("max", inDataPacked);

        sim.streamToCPU("count", DATA_SIZE / 2);

        sim.syncStreamFromCPU("max", 1000);
        sim.syncStreamToCPU("count", 1000);

        // Disable the ModelSim GUI
        sim.setEnableSimGUI(false);

        // Run the simulation
        sim.runTest();

        List<Long> outDataPacked = sim.getOutputData("count");

        int [] outData = new int[DATA_SIZE];

        for (int i = 0; i < DATA_SIZE; i += 2) {
            outData[i] = (int) (outDataPacked[i / 2] & 0 xffffffffl );
            outData[i + 1] = (int) (outDataPacked[i / 2] >>> 32);
        }

        int status = test(inData, outData, holdCount);
        if (status == 0)
            m.logMsg("Test passed OK!");
        else {
            m.logMsg("Test failed!");
            System.exit(status);
        }
    }
```

Listing 23: Custom HDL Node demonstrating the use of the mapped memory bus (MappedMemBusHDLNode.maxj).

```
/**
 * Document: Manager Compiler Tutorial (maxcompiler-manager-tutorial.pdf)
 * Chapter: 5
 * Example: 3
 * Summary:
 *        Custom HDL Node demonstrating the use of the mapped memory bus.
 */
package mappedmembushdl;

import com.maxeler.maxcompiler.v2.managers.custom.CustomManager;
import com.maxeler.maxcompiler.v2.managers.custom.blocks.CustomHDLNode;

class MappedMemBusHDLNode extends CustomHDLNode {

    MappedMemBusHDLNode(CustomManager manager, String instance_name) {
        super(manager, instance_name, "mapped_mem_hdl");

        CustomNodeClock m_clk;

        m_clk = addClockDomain("clk");
        m_clk.setNeedsReset("rst");

        addOutputStream("output", 32, m_clk, CustomNodeFlowControl.PUSH, 1);

        setMappedMemoryClock(m_clk);
        addMappedMemoryInterface("example");

        addVHDLSource("MappedMemBusHDLSource.vhdl", false);
    }
}
```

Listing 24: VHDL showing tyhe use of the mapped memory bus (MappedMemBusHDLSource.vhdl).

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

entity mapped_mem_hdl is
    port (
        clk : in std_logic;
        rst : in std_logic;

        output_valid : out std_logic;
        output_stall : in std_logic;
        output_data : out std_logic_vector (31 downto 0);

        mapped_mem_data_in : in std_logic_vector(35 downto 0);
        mapped_mem_addr : in std_logic_vector (15 downto 0);
        mapped_mem_wr_en : in std_logic;
        mapped_mem_rd_en : in std_logic;
        mapped_mem_ack : out std_logic;
        mapped_mem_data_out: out std_logic_vector(35 downto 0);
        mapped_mem_en_example : in std_logic
    );
end mapped_mem_hdl;

architecture arch of mapped_mem_hdl is
    signal output_enable : std_logic := '0';
    signal output_value : std_logic_vector (31 downto 0) := (others => '0');

    signal mapped_mem_wrack_reg : std_logic := '0';
    signal mapped_mem_rdack_reg : std_logic := '0';
    signal mapped_mem_ack_reg : std_logic := '0';

    signal mapped_mem_data_out_reg : std_logic_vector(35 downto 0);
    signal mapped_mem_data_out_reg_reg : std_logic_vector(35 downto 0);
begin

    -- Only the the output as valid when the output is not stalled
    output_valid <= output_enable and not output_stall;
    output_data <= output_value;

    -- Assign the delayed output signals to the output ports
    mapped_mem_ack <= mapped_mem_ack_reg;
    mapped_mem_data_out <= mapped_mem_data_out_reg_reg;
```

Listing 25: VHDL showing the use of the mapped memory bus (MappedMemBusHDLSource.vhdl).

```
    -- Delay the output signals so that the response is written a cycle later

    -- Process managing reads from the mapped memory bus
    process(clk)
    begin
        if (rising_edge(clk)) then
            mapped_mem_rdack_reg <= '0';
            mapped_mem_data_out_reg <= (others => '0');

            if (rst = '1') then
            -- If this Manager block is enabled and a read is requested
            elsif (mapped_mem_en_example = '1' and mapped_mem_rd_en = '1') then
                mapped_mem_data_out_reg <= "0000000000000000000" & mapped_mem_addr;
                mapped_mem_rdack_reg <= '1';
            end if;
        end if;
    end process;

    -- Process managing writes to the mapped memory bus
    process(clk)
    begin
        if (rising_edge(clk)) then
            mapped_mem_wrack_reg <= '0';

            if (rst = '1') then
                output_enable <= '0';
                output_value  <= (others => '0');
            -- If this Manager block is enabled and a write is requested
            elsif (mapped_mem_en_example = '1' and mapped_mem_wr_en = '1') then
                case mapped_mem_addr is
                when "0000000000000000" =>
                    -- Enable the output
                    output_enable <= mapped_mem_data_in(0);
                when "0000000000000001" =>
                    -- Set the output to the data from the mapped memory bus
                    output_value <= mapped_mem_data_in(31 downto 0);
                when others =>
                    null;
                end case;

                mapped_mem_wrack_reg <= '1';
            end if;
        end if;
    end process;

    process(clk)
    begin
        if (rising_edge(clk)) then
            if (rst = '1') then
                mapped_mem_ack_reg <= '0';
            else
                -- Or the read and write ack signals together
                mapped_mem_ack_reg <= mapped_mem_wrack_reg or mapped_mem_rdack_reg;
                mapped_mem_data_out_reg_reg <= mapped_mem_data_out_reg;
            end if;
        end if;
    end process;
end arch;
```

Listing 26: Simulation test bench for mapped memory example (MappedMemBusHDLSimRunner.maxj).

```
/*
 * Simulation runner for ModelSim simulation of a complete device.
 */
public class MappedMemBusHDLSimRunner {

    final static long testConstant = 0xdeadbeefL;

    private static final DFEModel BOARDMODEL = DFEModel.VECTIS;

    public static void main(String argv[]) {
        EngineParameters engineParameters = new EngineParameters(
            "MappedMemBusHDLExample3Sim", BOARDMODEL, Target.HDL_SIM);
        MappedMemBusHDLManager m = new MappedMemBusHDLManager(engineParameters);
        m.setBuildConfig(new BuildConfig(Level.FULL_BUILD));

        HDLTestBench sim = m.getHDLTestBench();

        // Reset the device
        sim.resetDevice();

        sim.streamToCPU("output", 10);

        // Set the output value
        sim.setMappedMemory("hdl_node_mapped_mem_bus.example", 1, Arrays
            .asList(testConstant));
        // Start the output stream
        sim.setMappedMemory("hdl_node_mapped_mem_bus.example", 0, Arrays
            .asList(1L));

        sim.syncStreamToCPU("output", 1000);

        // Disable the ModelSim GUI
        sim.setEnableSimGUI(false);

        // Run the simulation
        sim.runTest();

        List<Long> out_data = sim.getOutputData("output");

        for (int i = 0; i < out_data.size(); i++) {
            System.out.printf(
                "%08x %08x\n",
                (out_data[i] >>> 32),
                (out_data[i] & 0 xffffffffl ));
        }

        int status = testData(out_data, testConstant);

        if (status == 0)
            m.logMsg("Test passed OK!");
        else {
            m.logMsg("Test failed!");
            System.exit(status);
        }
    }
```

Listing 27: CPU code for running mapped memory example on DFE (MappedMemBusHDLCpuCode.c).

```

void MappedMemBusHDLCpu(int size, int testValue, uint32_t *dataOut)
{
    for (int i = 0; i < size; i++) {
        dataOut[i] = testValue;
    }
}

int check(uint32_t *dataOut, uint32_t *expected, int size)
{
    int status = 0;
        for(int i = 1; i < (size-1); i++) {
            if (dataOut[i] != expected[i]) {
                fprintf (stderr, "Output data @ %d = %d (expected %d)\n",
                    i, dataOut[i], expected[i]);
                status = 1;
            }
        }

    return status;
}

int main()
{
    const int size = 384;
    const int testValue = 0xdeadbeef;
    const int numMemRead = 8;
    uint32_t *outputData  = malloc(size * sizeof(uint32_t));
    uint32_t *expectedData = malloc(size * sizeof(uint32_t));
    uint64_t *mem = malloc(numMemRead * sizeof(uint64_t));

    max_file_t     *maxfile = MappedMemBusHDL_init();
    max_engine_t *engine = max_load(maxfile, "*");
    max_actions_t *actions;

    actions = max_actions_init(maxfile, NULL);
    for (int i = 0; i < numMemRead; i++) {
        max_get_mem_uint64t(actions, "hdl_node_mapped_mem_bus", "example", i, &mem[i]);
    }
    max_run(engine, actions);
    max_actions_free(actions);

    int status = 0;
    for (int i = 0; i < numMemRead; i++) {
        if (mem[i] != (uint64_t) i) {
            fprintf (stderr, "error:  expected %d.\n", i);
            status = 1;
        }
    }

    MappedMemBusHDLCpu(size, testValue, expectedData);

    actions = max_actions_init(maxfile, NULL);
    max_disable_reset(actions);
    max_set_mem_uint64t(actions, "hdl_node_mapped_mem_bus", "example", 1, testValue);
    max_set_mem_uint64t(actions, "hdl_node_mapped_mem_bus", "example", 0, 1);
    max_queue_output(actions, "output", outputData, size * sizeof(uint32_t));
    max_run(engine, actions);
    max_actions_free(actions);

    status = status | check(outputData, expectedData, size);
```