

Developing Fringe Support for Spatial

Tian Zhao

November 2018

1 Introduction

This document provides a guideline on implementing the Fringe module for Intel Arria10 boards. However, this guideline can be generalized to other FPGA platforms as well.

2 Environment Variables

This section includes specifications for environment variables being used in this documentation.

- *Spatial*: The spatial language.
- *SPATIAL_HOME*: The home directory of your installed Spatial.
- *JAVA_HOME*: The home directory of your installed Java.
- *IP_SRC_HOME*: The directory for storing source files of the *Accel* IP block.
- *FRINGE_MEM_BASEADDR*: The base address for shared DRAM between *Host* and *Accel*.
- *FRINGE_SCALAR_BASEADDR*: The base address for shared registerfile between *Host* and *Accel*.
- *Accel*: The accelerator hardware. In this document, the accelerator is the targeted Intel Arria 10 board.
- *Host*: The host CPU.
- *ArgIn*: The Spatial argument registers. This type of registers transfers scalars from *Host* to *Accel*.
- *ArgOut*: The Spatial argument registers. This type of registers transfers scalars from *Accel* to *Host*.
- *DRAM*: The Spatial DRAMs. DRAMs can transfer data bidirectionally between *Host* and *Accel*.

3 Spatial App Setup

We use *Lab1Part1RegExample* as the driving application in this document. The *Spatial* implementation can be found in Figure 1. The code block between line 11 to 17 creates two *ArgIn* registers and write two scalar values into them. The code block between line 19 to 23 executes *Accel* and writes the result to an *ArgOut*. Line 25 fetches the result at *Host*.

4 Interface between *Host* and *Accel*

Accel and *Host* share a memory-mapped register file and a DRAM. The register file is used for storing *ArgIn* and *ArgOut* registers. *Accel* can access this register file using the Avalon-MM lite interface. *Host* can access this register file by memory-mapping it into *Host*'s Virtual Memory Space. The DRAM is used for transferring data stored in *DRAM* objects. *Accel* can access this DRAM using the AXI4-MM interface. *Host* can access this DRAM by memory-mapping it into *Host*'s Virtual Memory Space.

4.1 Generating *Accel* Code from *Spatial*

A *Spatial* design can be translated into a *Verilog* design. The Top level of this design is exposed in the form of an AXI4 master interface and an Avalon lite slave interface. The following commands will generate the *Verilog* design for *Lab1Part1RegExample*.

```
cd $SPATIAL_HOME
sbt -batch "apps/run-main Lab1Part1RegExample --synth"
cd gen/Lab1Part1RegExample
sbt "runMain top.Instantiator --verilog --testArgs arria10"
```

We can find the generated *Verilog* design by going to:

```
cd $SPATIAL_HOME/gen/Lab1Part1RegExample/verilog-arria10
```

The *Top.v* file includes the generated design. Notice that *SRFF* is a reserved keyword in Intel's toolchain. To avoid naming conflicts, we need to change the name to *SRFF_sp*:

```
sed -i 's/SRFF/SRFF_sp/g' Top.v
```

Now we have the generated design of *Lab1Part1RegExample* in *Top.v*. Next, we need to integrate it into the rest of the system. We wrap the design file, *Top.v*, as an IP in Intel's Platform Designer. The Platform Designer will handle the process of creating interconnects. The next session gives an example on how to create such IP.

```

1  import spatial.dsl._
2  import org.virtualized._
3
4
5  // Register
6  object Lab1Part1RegExample extends SpatialApp {
7      type T = Int
8
9      @virtualize
10     def main() {
11         val N = args(0).to[T]
12         val M = args(1).to[T]
13         val argRegIn0 = ArgIn[T]
14         val argRegIn1 = ArgIn[T]
15         setArg(argRegIn0, N)
16         setArg(argRegIn1, M)
17         val argRegOut = ArgOut[T]
18
19         Accel {
20             val argRegIn0Value = argRegIn0.value
21             val argRegIn1Value = argRegIn1.value
22             argRegOut := argRegIn0Value + argRegIn1Value
23         }
24
25         val argRegOutResult = getArg(argRegOut)
26         println("Result = " + argRegOutResult)
27
28         val gold = M + N
29         println("Gold = " + gold)
30         val cksum = gold == argRegOutResult
31         println("PASS = " + cksum + "(Lab1Part1RegExample)")
32     }
33 }

```

Figure 1: Lab1Part1RegExample. This application performs reading and writing *ArgIn*, *ArgOut* registers

4.2 Creating an IP for *Accel*

In this section, we demonstrate the process of wrapping the generated *Verilog* design in an IP and export it to Intel’s Platform Designer. In the starter project, we provide a folder called *ip_src* and a *tcl* script called *Top_hw.tcl*. *ip_src* contains source files for building this IP. *Top_hw.tcl* specifies the interfaces to communicate with this IP. Specifically, the exposed interfaces are Avalon-MM slave and AXI4-MM master. You can find more information about these interfaces in the *tcl* script.

After generating *Top.v*, you can copy it into *ip_src*. Now you can import *Accel* as an IP in your *Quartus* project.

```
cp Top.v $IP_SRC_HOME/ip_src/
```

A little more information is needed for the *Host* software. When instantiating the *Accel* IP in your design, the Platform Designer will ask for the base addresses of the Avalon-MM and the AXI4-MM interfaces. These base addresses will be used by the *Host* code for reading and writing the register files and the memory-mapped DRAM. For convenience, we assume that the base address of Avalon-MM is at *FRINGE_SCALAR_BASEADDR*, and the base address of AXI4-MM is at *FRINGE_MEM_BASEADDR*.

5 Supporting *Host* Code of *Spatial*

In this tutorial, we assume that the *Host* uses *ARM* compilers. If you are using a different compiler for *c++*, you can change the compiler options in *\$SPATIAL_HOME/gen/Lab1Part1RegExample/scripts/arria10.mk*.

5.1 *Host* API Specifications

First, move into the *Host* code directory by running:

```
cd $SPATIAL_HOME/gen/Lab1Part1RegExample/cpp/
```

The logic for running *Accel* is autogenerated by *Spatial* and is stored in *TopHost.cpp*. *TopHost.cpp* first loads the bitstream to *Accel* using *load* function. Then, it sets the *ArgIn*, *ArgOut* registers using *setArg* functions. If your design contains *DRAM*, *TopHost* will also set addresses for the *DRAM* using *setArg* functions. To move data from *Host* to *Accel*, *TopHost* will call *memcpy*. After the setup is done, *TopHost* calls *run* to start *Accel*. Once *Accel* finishes, *TopHost* will call *getArg* to collect the final results. If your design contains *DRAM*, *TopHost* will call *memcpy* to copy data from *Accel* to *Host*. *setArg* and *getArg* functions are implemented using *readReg* and *writeReg* functions. *run* function is implemented using *setArg*, *getArg*.

We can see that *load*, *readReg*, *writeReg*, *memcpy* are the key functions for *TopHost* to control *Accel*. To support *Spatial*’s *Host* code on your platform, you will need to implement these functions as shown in Table 1. These *APIs* are defined in as shown in *fringeArria10/FringeContextBase.h*. You will also need

<code>void load()</code>	Loads a bitstream to <i>Accel</i> .
<code>uint64_t malloc(size_t bytes)</code>	Allocates <i>bytes</i> memory.
<code>void free(uint64_t buf)</code>	Free the memory pointer <i>buf</i> .
<code>void memcpy(uint64_t devmem, void hostmem, size_t size)</code>	<i>Accel</i> side memcpy. Memcpy data of <i>size</i> from <i>devmem</i> to a virtual address at <i>host</i> .
<code>void memcpy(void hostmem, uint64_t devmem, size_t size)</code>	<i>Host</i> side memcpy. Memcpy data of <i>size</i> from <i>hostmem</i> to a physical address at <i>devmem</i> .
<code>void writeReg(uint32_t reg, uint64_t data)</code>	Write <i>data</i> to the register located at <i>reg</i> starting from the scalar offset.
<code>uint64_t readReg(uint32_t reg)</code>	Read data stored at the register located at <i>reg</i> starting from the scalar offset.

Table 1: Specifications for *Host* Functions

to change *FRINGE_SCALAR_BASEADDR*, *FRINGE_MEM_BASEADDR* in *fringeArria10/Arria10AddressMap.h*.

Table 1 provides a specifications on what each function does. For a detailed example, check *fringeArria10/FringeContextArria10.h*. Figure 2 shows the complete API.

After implementing these functions, you can generate an executable for the *Host* code and try to use it to control the execution of *Accel*. If the flow works, you will need to update the templates for *Host* and *Accel* generation. The *Host* templates are stored at:

```
$SPATIAL_HOME/spatial/core/resources/cppgen/fringeArria10/
```

The *Accel* templates are stored at:

```
$SPATIAL_HOME/spatial/core/resources/chiselgen/ \
  template-level/fringeArria10/build/
```

Then, you will need to update the project resources by running:

```
cd $SPATIAL_HOME
bash bin/update_resources.sh
```

You will also need to update the *Makefile* with your build flow. The *Makefile* is at

```
$SPATIAL_HOME/spatial/core/resources/chiselgen/app-level/Makefile
```

6 OPAE Implementation

6.1 *Host*

The OPAE flow on the *Host* side can be decoupled into the following stage. Each stage can be implemented in a *FringeContext* function as specified in Table 1.

- Discover / Search AFU. This stage should be implemented in the constructor of *FringeContextArria10* class.

```

1  #ifndef __FRINGE_CONTEXT_BASE_H__
2  #define __FRINGE_CONTEXT_BASE_H__
3
4  template <class T>
5  class FringeContextBase {
6  public:
7      T *dut = NULL;
8      std::string path = "";
9
10     FringeContextBase(std::string p) {
11         path = p;
12     }
13     virtual void load() = 0;
14     virtual uint64_t malloc(size_t bytes) = 0;
15     virtual void free(uint64_t buf) = 0;
16     virtual void memcpy(uint64_t devmem, void* hostmem, size_t size) = 0;
17     virtual void memcpy(void* hostmem, uint64_t devmem, size_t size) = 0;
18     virtual void run() = 0;
19     virtual void writeReg(uint32_t reg, uint64_t data) = 0;
20     virtual uint64_t readReg(uint32_t reg) = 0;
21     virtual uint64_t getArg(uint32_t arg, bool isIO) = 0;
22     virtual void setArg(uint32_t reg, uint64_t data, bool isIO) = 0;
23     virtual void setNumArgIns(uint32_t number) = 0;
24     virtual void setNumArgIOs(uint32_t number) = 0;
25     virtual void setNumArgOutInstrs(uint32_t number) = 0;
26     virtual void setNumArgOuts(uint32_t number) = 0;
27     virtual void flushCache(uint32_t kb) = 0;
28
29     ~FringeContextBase() {
30         // delete dut;
31     }
32 };
33
34 // Fringe error codes
35
36
37 // Fringe APIs - implemented only for simulation
38 void fringeInit(int argc, char **argv);
39
40 #endif

```

Figure 2: FringeContextBase.h

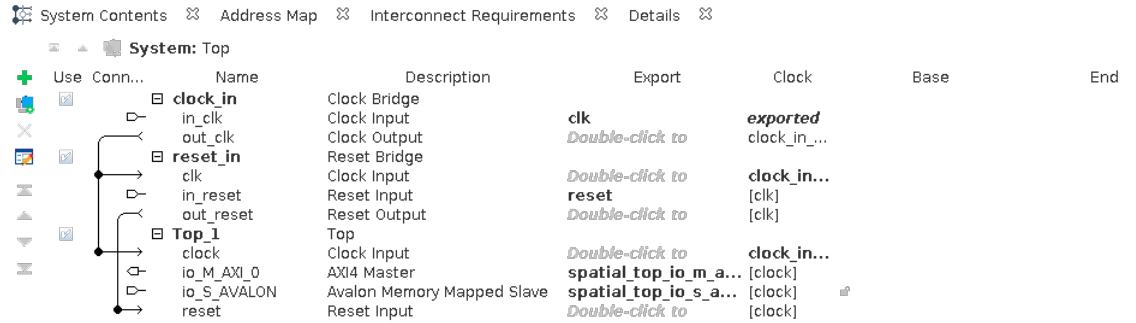


Figure 3: Example System Diagram in Platform Designer

- Acquire ownership of an AFU. This stage should be implemented in the constructor of *FringeContextArria10* class.
- Map AFU registers to user space. This stage should be implemented in the constructor of *FringeContextArria10* class.
- Allocate / Define shared memory Space. This stage should be implemented in the constructor of *FringeContextArria10* class.
- Start / Stop computation on AFU and wait for results. Implement *writeReg*, *readReg* and *memcpy* functions in *FringeContextArria10*. Spatial Host will automatically start and stop the communication on AFU. More details can be found in the *run* function defined in *FringeContextArria10* class.
- Deallocated shared memory. This stage should be implemented in the *free* function.
- Relinquish ownership of an AFU. This stage should be implemented in the destructor of *FringeContextArria10*.

6.2 Accel

The OPAE AFU side can communicate with the *Host* through Avalon-MM (for scalar communication) and AXI4-MM (for DRAM communication). The system diagram is shown in Figure 3. The *Accel* IP is instantiated as *Top_1*. You can connect *io_M_AXI_0* to *EMIF* via Avalon-MM and *io_S_AVALON_0* to the CCI-P bus. The Platform Designer will translate the protocol automatically once you make the connection.