开发*Spatial*的*Fringe*支持

简介

本文描述了如何在Intel Arria10开发环境下实现*Fringe*模块。本文描述的实现方式也可以应用在支持其他的*FPGA*平台。

环境参数

本章节定义了在本文中会用到的一些环境参数。

- *Spatial*: *Spatial* 语言。
- *SPATIAL_HOME*: *Spatial* 的安装路径。
- *JAVA_HOME*: *Java* 的安装路径。
- *IP_SRC_HOME*: 放置有*Accel*的IP的路径。
- *FRINGE_MEM_BASEADDR*: *Host*和*Accel*之间通讯DRAM的基底位址。
- *FRINGE_SCALAR_BASEADDR*: *Host*和*Accel*之间寄存器文件(Register File)的基底位址。
- *Accel*: 实现的加速器设计。在本文中，该加速器设计基于Intel Arria10。
- *Host*: CPU端。
- *ArgIn*: *Spatial*的*Host*到*Accel*端通讯用寄存器。
- *ArgOut*: *Spatial*的*Accel*到*Host*端通讯用寄存器。
- *DRAM*: *Spatial*的DRAM。用于*Host*到*Accel*的双向数据交换。

应用样例

本文使用 *Lab1Part1RegExample* 做为例子。该应用详见

```
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 }
```

. 行11～17使用了两个*ArgIn*寄存器。*Host*写入了两个标量。 行19～23运行了加速器并且将结果写入到*ArgOut*里。 行25将结果返回到*Host*端。

*Host*和*Accel*之间的界面

*Accel*和*Host*之间共用了映射到内存的寄存器文件和DRAM。寄存器文件用于储存*ArgIn*, *ArgOut*。*Accel*可以通过Avalon-MM接口访问这个寄存器文件。*Host*可以将这个寄存器文件映射到操作系统的虚拟内存上。DRAM用于在*Host*和*Accel*之间传输高维张量。*Accel*可以通过AXI4-MM访问DRAM。*Host*可以通过将该DRAM映射到操作系统虚拟内存来访问这个DRAM。

用*Spatial*生成*Accel*的RTL设计文件

一个基于*Spatial*的设计文件可以用于生成*Verilog*设计文件。该设计文件最上层使用AXI4 master和Avalon lite slave与系统其他部分进行通讯。以下命令可以生成这个设计文件（假设用户已经有了*Lab1Part1RegExample*这个应用):

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

进入生成*Verilog*文件的路径：

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

*Top.v*文件含有生成的设计。因为SRFF是一个Intel的关键词，我们需要避免命名冲突。执行以下命令：

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

现在*Top.v*含有生成的设计。我们需要把这个设计封装成一个IP。在我们提供的文件夹里，*Top_hw.tcl*文件含有IP的定义。IP的源文件在*ip_src*里。下个章节描述了如何生成这个IP.

生成包含有*Accel*设计的IP

本章节描述了如何将基于*Accel*生成的*Top.v*文件封装入一个可导入Intel Platform Designer的IP。在repo里，我们提供了*Top_hw.tcl*。 该文件内包含了源文件路径和AXI4 Master与Avalon Slave的接口定义。这些接口可以在Platform Designer内连接到系统的数据总线上。

为了生成该IP, 我们需要执行以下命令去更新IP的设计源文件：

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

当把该IP导入Platform Designer时，Platfrom Designer会要求用户提供Avalon Slave和AXI4 Master的基底位址。这些基底位址将被$Host$端映射到虚拟内存上。我们假设Avalon Slave的基底位址位于*FRINGE_SCALAR_BASEADDR*；AXI4 Master的基底位址位于*FRINGE_MEM_BASEADDR*。

在CPU端支持*Spatial Host*

本章节描述了*Host*端的实现方式。我们提供的例子里使用了ARM的g++编译器。如果用户的编译器是非ARM的，可以在以下文件里更改：

```
$SPATIAL_HOME/gen/Lab1Part1RegExample/scripts/arria10.mk.
```

*Host*端API定义

首先，进入*Host*端的源代码文件夹：

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

运行*Host*的逻辑由*Spatial*自动生成。该文件位于*TopHost.cpp*。该文件首先通过*load*函数将*bitstream*加载到加速卡端。然后，*TopHost*用*setArg*函数加载标量到*ArgIn*寄存器里。如果用户的设计里有*DRAM*，*TopHost*也会自动用*setArg*函数设置*DRAM*的基地位置。*TopHost*会使用*memcpy*函数来实现*Host*,*Accel*端的双向张量传递。当这些设置完成后，*TopHost*会用*run*函数来启动*Accel*。当*Accel*运行结束后，*TopHost*会用*getArg*来将结果传递回*Host*。如果设计里有*DRAM*，*TopHost*也会使用*DRAM*将结果传递回*Host*。

在这个流程里，*load, readReg, writeReg, memcpy* 函数被用于构建其他函数。用户需要实现这四个函数。全部的API定义可以在以下

```
#ifndef __FRINGE_CONTEXT_BASE_H__
#define __FRINGE_CONTEXT_BASE_H__

template <class T>
class FringeContextBase {
public:
  T *dut = NULL;
  std::string path = "";

  FringeContextBase(std::string p) {
    path = p;
  }
  virtual void load() = 0;
  virtual uint64_t malloc(size_t bytes) = 0;
  virtual void free(uint64_t buf) = 0;
  virtual void memcpy(uint64_t devmem, void* hostmem, size_t size) = 0;
  virtual void memcpy(void* hostmem, uint64_t devmem, size_t size) = 0;
  virtual void run() = 0;
  virtual void writeReg(uint32_t reg, uint64_t data) = 0;
  virtual uint64_t readReg(uint32_t reg) = 0;
  virtual uint64_t getArg(uint32_t arg, bool isIO) = 0;
  virtual void setArg(uint32_t reg, uint64_t data, bool isIO) = 0;
  virtual void setNumArgIns(uint32_t number) = 0;
  virtual void setNumArgIOs(uint32_t number) = 0;
  virtual void setNumArgOutInstrs(uint32_t number) = 0;
  virtual void setNumArgOuts(uint32_t number) = 0;
  virtual void flushCache(uint32_t kb) = 0;

  ~FringeContextBase() {
//    delete dut;
  }
};

// Fringe error codes


// Fringe APIs - implemented only for simulation
void fringeInit(int argc, char **argv);

#endif
```

里找到：

```
$SPATIAL_HOME/gen/Lab1Part1RegExample/cpp/fringeArrai10/FringeContextBase.h
```

用户同时需要更新*FRINGE_SCALAR_BASEADDR*和*FRINGE_MEM_BASEADDR*常数。这些常数定义在*Arria10AddressMap.h*里。 以下表格描述了用户需要实现的每个函数的功能：

函数	描述
void load()	加载bitstream到加速器端
uint64_t malloc(*size_t* bytes)	在内存中要求*bytes*大小的记忆块
void free(*uint64_t* buf)	放弃*bytes*大小的记忆块
void memcpy(*uint64_t* devmem, *void*$ hostmem, *size_t* size)	将*Host*端*size*大小的数据拷贝到*Accel*端
void memcpy(*void* hostmem, *uint64_t* devmem, *size_t* size)	将*Accel*端*size*大小的数据拷贝到*Host*端
void writeReg(*uint32_t* reg, *uint64_t* data)	往位于*reg*的寄存器写入*data*
uint64_t readReg(*uint32_t* reg)	从位于*reg*的寄存器读出一个*uint64_t*类型的数据

在完成这些函数后，用户可以尝试生成一个可执行文件以执行*Accel*。如果这个流程可以工作，用户需要将这些模版在*Spatial*源代码里更新。*Host*端模版位于：

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

*Accel*端模版位于：

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

在更新完模版后，用户需要重新更新*Spatial*的文件依赖。执行以下命令：

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

如果想自动化以上的任务流程，用户可以将以上流程写入*Makefile*。*Makefile*位于

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

OPAE实现

本章节讨论可能的OPAE实现方式。

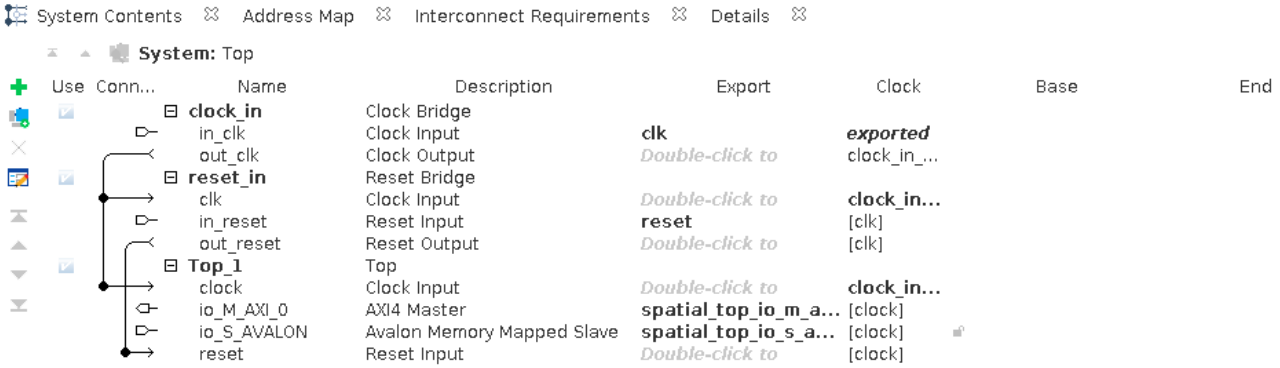
Host

OPAE在*Host*端的工作流程可以被分割为以下几个阶段。*TopHost*已经可以实现这些执行流程。用户需要实现*load, free, memcpy, writeReg, readReg*。以下几个阶段在OPAE文档内可以找到。

- 寻找AFU. 可以在 *FringeContextArria10*构造函数里实现。
- 获得AFU使用全选。可以在 *FringeContextArria10*构造函数里实现。
- 将AFU 寄存器地址映射到用户端记忆区内。可以在*FringeContextArria10*构造函数里实现。
- 定义*Host*和*Accel*共享的记忆去区地址空间。可以在*FringeContextArria10*构造函数里实现。
- 开始/结束 AFU;等待执行结果。*TopHost*会自动执行这一流程。
- 释放共享的记忆区。用户需要实现*free*函数。
- 释放AFU权限。可以在 *FringeContextArria10*的析构函数里实现。

Accel

用户可以在Platform Designer内导入*Accel*的IP。见



样例。*Accel*被导入后重命名为*Top_1*。用户可以将*io_M_AXI_0*链接到EMIF的Avalon-MM接口上。Platform Designer会自动完成接口协议转换。用户需要将*io_S_AVALON_0*链接到CCI-P数据总线上。Platform Designer也会自动实现接口协议转换。