connectal Documentation

Release 14.12.6

Jamey Hicks, Myron King, John Ankcorn

February 06, 2015

CHAPTER

ONE

CONTENTS

1.1 Introduction

Introduction goes here.

1.2 Connectal Design

1.2.1 Abstract

The cost and complexity of hardware-centric systems can often be reduced by using software to perform tasks which don't appear on the critical path. Alternately, the performance of software can sometimes be improved by using special purpose hardware to implement tasks which *do* appear on the critical path. Whatever the motivation, most modern systems are composed of both hardware and software components.

Given the importance of the connection between hardware and software in these systems, it is surprising how little automated and machine-checkable support there is for co-design space exploration. This paper presents the Connectal framework, which enables the development of hardware accelerators for software applications by generating hardware/software interface implementations from abstract Interface Design Language (IDL) specifications.

Connectal generates stubs to support asynchronous remote method invocation from software to software, hardware to software, software to hardware, and hardware to hardware. For high-bandwidth communication, the Connectal framework provides comprehensive support for shared memory between hardware and software components, removing the repetitive work of processor bus interfacing from project tasks.

This framework is released as open software under an MIT license, making it available for use in any projects.

1.2.2 Introduction

Because they are so small and inexpensive, processors are now included in all but the smallest hardware designs. This grants flexibility to hardware designers because the non-performance-critical components can be implemented in software and the performance-critical components can be implemented in hardware. Using software for parts of the design can decrease the effort required to implement configuration and orchestration logic (for example). It can also offer hardware developers greater adaptability in meeting new project requirements or supporting additional applications.

As a system evolves through design exploration, the boundary between the software and hardware pieces can change substantially. The old paradigm of "separate hardware and software designs before the project starts" is no longer sustainable, and hardware teams are increasingly responsible for delivering significant software components.

Despite this trend, hardware engineers find themselves with surprisingly poor support for the development of the software that is so integral to their project's success. They are often required to manually develop the necessary

software and hardware to connect the two environments. In the software world, this is equivalent to manually re-creating header files from the prose description of an interface implemented by a library. Such ad hoc solutions are tedious, fragile, and difficult to maintain. Without a consistent framework and toolchain for jointly managing the components of the hardware/software boundary, designers are prone to make simple errors which can be expensive to debug.

The goal of our work is to support the flexible and consistent partitioning of designs across hardware and software components. We have identified the following four goals as central to this endeavor:

- Connect software and hardware by compiling interface declarations.
- Enable concurrent access to hardware accelerators from software.
- Enable high-bandwidth sharing of system memory with hardware accelerators.
- Provide portability across platforms (CPU, OS, bus types, FPGAs).

In this paper, we present a software-driven hardware development framework called Connectal. Connectal consists of a fully-scripted tool-chain and a collection of libraries which can be used to develop production quality applications comprised of software components running on CPUs communicating with hardware components implemented in FPGA or ASIC.

When designing Connectal, our primary goal was to create a collection of components which are easy to use for simple implementations and which can be configured or tuned for high performance in more complicated applications. To this end, we adopted a decidedly minimalist approach, attempting to provide the smallest viable programming interface which can guarantee consistent access to shared resources in a wide range of software and hardware execution environments. Because our framework targets the implementation of performance-critical systems rather than their simulation, we have worked hard to remove any performance penalty associated with its use.

We wrote the hardware components of the Connectal libraries in Bluespec System Verilog (BSV) because it enables a higher level of abstraction than the alternatives and supports parameterized types. The software components are implemented in C/C++. We chose Bluespec interfaces as the interface definition language (IDL) for Connectal's interface compiler.

This paper describes the Connectal framework, and how it can be used to flexibly move between a variety of software environments and communication models when mapping applications to platforms with connected FPGAs and CPUs.

Document Organization

In Section *Accelerating String Search*, we present an example running in a number of different execution environments. In Section *The Connectal Framework*, we give an overview of the Connectal framework and its design goals. In Section *Sec-Impl* we discuss the details of Connectal and how it can be used to implement the example. Section *Sec-ToolChain* describes the implementation of Connectal, supported platforms, and the tool chain used to coordinate the various parts of the framework. The paper concludes with a discussion of performance metrics and related work.

1.2.3 Accelerating String Search

The structure of a hardware/software (HW/SW) system can evolve quite dramatically to reflect changing requirements, or during design exploration. In this section, we consider several different implementations of a simple string search application~cite{mpAlgo}. Each variation represents a step in the iterative refinement process, intended to enhance performance or enable a different software execution environment.

begin{figure}[!h]

centering

includegraphics[width=0.43textwidth]{platform.pdf}

caption{label{Fig:Platform0}Target platform for string search application}

end{figure}

Figure~ref{Fig:Platform0} shows the target platform for our example. The pertinent components of the host system are the multi-core CPU, system memory, and PCI Express (PCIe) bus. The software components of our application will be run on the CPU in a Linux environment. Connected to the host is a PCIe expansion card containing (among other things) a high-performance FPGA chip and a large array of flash memory. The FPGA board was designed as a platform to accelerate ‘‘big data" analytics by moving more processing power closer to the storage device.

Initial Implementation

begin{figure}[!h]

centering

includegraphics[width=0.43textwidth]{data_accel_logical0.pdf}

caption{label{Fig:StringSearch0}Logical components of the string search system}

end{figure}

The design process really begins with a pure software implementation of the algorithm, but the first attempt we consider is the initial inclusion of HW acceleration shown in Figure~ref{Fig:StringSearch0}. The search functionality is executed by software running in user-space which communicates with the hardware accelerator through a device driver running in the Linux kernel. The hardware accelerator, implemented in the FPGA fabric, executes searches over data stored in the flash array as directed by the software.

The FPGA has direct access to the massive flash memory array, so if we implement the search kernel in hardware, we can avoid bringing data into the CPU cache (an important consideration if we intend to run other programs simultaneously). By exploiting the high parallelism of the execution fabric as well as application aware caching of data, an FPGA implementation can outperform the same search executed on the CPU.

Multithreading the Software

The efficient use of flash memory requires a relatively sophisticated management strategy. Our first refinement is based on the observation that there are four distinct tasks which the application software executes (mostly) independently:

- Send search command to the hardware.
- Receive search results from the hardware.
- Send commands to the hardware to manage the flash arrays
- Receive responses from the flash management hardware

To exploit the task-level parallelism in our application, we can assign one thread to each of the four enumerated tasks. To further improve efficiency, the two threads receiving data from the hardware put themselves to sleep by calling textbf{poll} and are woken up only when a message has been received.

begin{figure}[!h]

centering

includegraphics[width=0.43textwidth]{data_accel_logical1.pdf}

caption{label{Fig:StringSearch1}Using a mutex to coordinate user-level access to hardware accelerator}

end{figure}

With the introduction of multithreading, we will need a synchronization mechanism to enforce coherent access to the hardware resources. Because the tasks which need coordinating are all being executed as user-space threads, the

access control must be implemented in software as well. As shown in Figure~ref{Fig:StringSearch1}, a mutex is used to coordinate access to the shared hardware resource between user-level processes.

Refining the Interfaces

begin{figure}[!h] centering includegraphics[width=0.43textwidth]{data_accel_logical2.pdf} caption{label{Fig:StringSearch2}Movement of functionality from

user to kernel space. Software-based coordination between kernel and user processes are prohibitively expensive.}

end{figure}

Figure~ref{Fig:StringSearch2} shows a further refinement to our system in which we have reimplemented the Flash Management functionality as a block-device driver. Instead of directly operating on physical addresses, the string search now takes a file descriptor as input and uses a Linux system-call to retrieve the file block addresses through the file system. This refinement permits other developers to write applications which can take advantage of the accelerator without any knowledge of the internal details of the underlying storage device. It also enables support for different file systems as we now use a POSIX interface to generate physical block lists for the the storage device hardware. The problem with this refinement is that we no longer have an efficient SW mechanism to synchronize the block device driver running in kernel space with the application running in user space.

begin{figure}[htb] centering includegraphics[width=0.43textwidth]{data_accel_logical3.pdf} caption{label{Fig:StringSearch3}Correct interface design

removes the need for coordination between user and kernel threads.}

end{figure}

To solve to this problem (shown in Figure~ref{Fig:StringSearch3}), we can remove the need for explicit SW coordination altogether by giving each thread uncontested access to its own dedicated HW resources mapped into disjoint address regions. (There will of course be implicit synchronization through the file system.)

Shared Access to Host Memory

In the previous implementations, all communication between hardware and software takes place through memory mapped register IO. Suppose that instead of searching for single strings, we want to search for large numbers of (potentially lengthy) strings stored in the flash array. Attempting to transfer these strings to the hardware accelerator using programmed register transfers introduces a performance bottleneck. In our final refinement, the program will allocate memory on the host system, populate it with the search strings, and pass a reference to this memory to the hardware accelerator which can then read the search strings directly from the host memory.

begin{figure}[htb] centering includegraphics[width=0.43textwidth]{data_accel_logical4.pdf} caption{label{Fig:StringSearch4}Connectal support for DMA.}

end{figure}

Efficient high-bandwidth communication in this style requires the ability to share allocated memory regions between hardware and software processes without copying. Normally, a programmer would simply call application space textbf{malloc}, but this does not provide a buffer that can be shared with hardware or other software processes. As shown in Figure~ref{Fig:StringSearch4}, a special-purpose memory allocator has been implemented in Linux, using dmabufcite{dmabuf} to provide reference counted sharing of memory buffers across user processes and hardware.

To conclude, we consider how the HW/SW interface changed to accommodate each step in the refinement process: The hardware interface required by the design in Figure~ref{Fig:StringSearch0} is relatively simple. Command/response queues in the hardware accelerator are exposed using a register interface with accompanying *empty/full* signals. To support the use of *poll* by the refinement in Figure~ref{Fig:StringSearch1}, interrupt signals must be added to the

hardware interface and connected to the Linux kernel. Partitioning the address space as required by the refinement in Figure~ref{Fig:StringSearch3} necessitates a consistent remapping of registers in both hardware and software.

1.2.4 The Connectal Framework

In and of themselves, none of the HW/SW interfaces considered in Section *Accelerating String Search* are particularly complex. On the other hand, implementing the complete set and maintaining correctness as the application evolves is a considerable amount of care, requiring deep understanding of both the application and the platform. The Connectal framework is a collection of tools and library components which was designed to address these challenges with the following features:

- Easy declaration and invocation of remote methods between application components running on the host or in the FPGA.
- Direct user-mode access to hardware accelerators from software.
- High performance read and write bus master access to system memory from the FPGA
- Infrastructure for sharing full speed memory port access between an arbitrary number of clients in the FPGA fabric
- Portability across platforms using different CPUs, buses, operating systems, and FPGAs
- Fully integrated tool-chain support for dependency builds and device configuration.

In this section, we introduce the Connectal framework through a discussion of its prominent features.

Portals

Connectal implements remote method invocation between application components using asynchronous messaging. The message and channel types are application specific, requiring the user to define the HW/SW interface using BSV interfaces as the interface definition language (IDL). These interfaces declare logical groups of unidirectional ‘‘send” methods, each of which is implemented as a FIFO channel by the Connectal interface compiler; all channels corresponding to a single BSV interface are grouped together into a single *portal*.

From the interface specification, the Connectal interface compiler generates code for marshalling the arguments of a method into a message to be sent and unmarshaling values from a received message. It generates a textit{proxy} to be invoked on the sending side and a textit{wrapper} that invokes the appropriate method on the receiving side. Platform specific libraries are used to connect the proxies and wrappers to the communication fabric.

In the hardware, each portal is assigned a disjoint address range. On the host, Connectal assigns each portal a unique Linux device (/dev/portal*n*$) which is accessed by the application software using the generated wrappers and proxies. An application can partition methods across several portals, to control access to the interfaces by specific hardware or software modules. To support bi-directional communication, at least two portals are required: one which allows software to ‘‘invoke” hardware, and another for hardware to ‘‘invoke” software. Each portal may be accessed by different threads, processes, or directly from the kernel.

Direct user-mode access to hardware

We designed Connectal to provide direct access to accelerators from user-mode programs in order to eliminate the need for device-drivers specific to each accelerator. We have implemented a kernel module for both X86 and ARM architectures with a minimal set of functionality: the driver implements textbf{mmap} to map hardware registers into user space and textbf{poll} to enable applications to suspend a thread waiting for interrupts originating from the hardware accelerators. These two pieces of functionality have been defined to be completely generic; no modification is required to kernel drivers as the HW/SW interface evolves. All knowledge of the interface register semantics (and

corresponding changes) is encoded by the interface compiler in the generated proxies and wrappers which are compiled as part of the application and executed in user-mode.

This approach is known as user-space device drivers~cite{Khalidi:1995:EZI:974947,UIO:Howto} and has a number of distinct advantages over traditional kernel modules. To begin with, it reduces the number of components that need to be modified if the HW/SW interface changes, and eliminates the need for device-driver development expertise in many cases. Secondly, after the hardware registers have been mapped into user address space, the need for software to switch between user and kernel mode is all but eliminated since all ‘‘driver” functionality is being executed in user-space.

Shared Access to Host Memory

Connectal generates a hardware FIFO corresponding to each method in the portal interface, and the software reads and writes these FIFOs under certain conditions. To improve throughput, Connectal libraries also support credit-based flow-control. Though credit-based flow-control with interrupts is more efficient than polling status registers from software, there is often the need for much higher bandwidth communication between the hardware and software.

Hardware accelerators often communicate with the application through direct access to shared memory. An important feature of Connectal is a flexible, high performance API for allocating and sharing such memory, and support for reading and writing this memory from hardware and software. The Connectal framework implements this through the combination of a Linux kernel driver, C++ libraries, and BSV modules for the FPGA. We implemented a custom kernel memory allocator for Connectal, textbf{portalmem}, using the kernel dmabuf support. Any solution which allocates and shares memory between hardware and software must meet two high-level requirements:

- **Allocated buffers must have reference counts to prevent memory** leaks.
- **Efficient mechanisms must be provided to share the location of** allocated regions.

Using the portalmem driver, programs can allocate regions of system memory (DRAM) and map it into their own virtual address space. Reference-counted access to shared memory regions allocated using portalmem can be granted to other SW processes by transmitting the file descriptor for the allocated region. Reference counting has been implemented in the driver so that once an allocated memory region has been dereferenced by all SW and HW processes, it will be deallocated and returned to the kernel free memory pool.

Simple hardware accelerators often require contiguous physical addresses. Unfortunately, when allocating memory from a shared pool in a running system, obtaining large areas of contiguous memory is often problematic, limiting the size of the region that can be allocated. To support indexed access to non-contiguous memory aggregates, Connectal provides address translation support to hardware accelerators in the FPGA, similar to the MMU functionality on the CPU side.

Distributed Access to Memory Ports

When building accelerators for an algorithm, multiple parameters are often accessed directly from system memory using DMA. As the hardware implementation is parallelized, multiple accesses to each parameter may be required. In these cases, the number of memory clients in the application hardware usually exceeds the number of host memory ports. Sharing these ports requires substantial effort, and scaling up a memory interconnect while maximizing throughput and clock speed is extremely challenging.

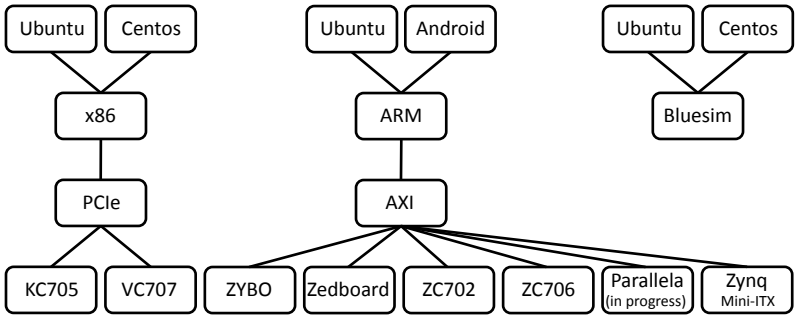
To support this common design pattern, the Connectal framework provides provides a portable, scalable, high performance library that applications can use to to facilitate the efficient sharing of host memory ports. This library is implemented as parameterized Bluespec modules which allow the user to easily configure high-performance memory access trees, supporting both reading and writing.

Platform Portability

We structured Connectal to improve the portability of applications across CPU types, operating systems, FPGAs, and how the CPU and FPGA are connected. The software and hardware libraries are largely platform independent. As a result, applications implemented in the framework can be compiled to run on the range of different platforms.

Supported platforms are shown in Figure *Fig-platforms*. Application software can be executed on x86 and ARM CPUs running either Ubuntu or Android operating systems. A range of different Xilinx FPGAs can be connected to the CPU and system memory via PCI Express or AXI. The BSV simulator (Bluesim) can be used in place of actual FPGA hardware for debugging purposes.

When the target application needs to interact with other Linux kernel resources (for example, a block device or a network interface), the application may run in kernel mode with the logic run either in an FPGA or in Bluesim.



1.2.5 Implementing String Search

Having covered the features of the Connectal at a high level, we now explain more specifically how the framework can be applied to implement the refinements outlined in Section *Accelerating String Search*.

Initial Implementation

The FPGA is connected to the host system with a PCIe bus, and to the memory array with wires. In addition to implementing a search kernel, the hardware accelerator must communicate with the software components and with the flash chips. Communication with the software takes place through portals, whose interface declaration is given below:

```
interface StrstrRequest;
  method Action setupNeedle(Bit#(8) needleChars);
  method Action search(Bit#(32) haystackPtr,
                       Bit#(32) haystackLen);
endinterface
interface StrstrIndication;
  method Action searchResult(Int#(32) v);
  method Action setupComplete();
endinterface
```

The hardware implements the StrstrRequest interface, which the software invokes (remotely) to specify the search string and the location in flash memory to search. The software implements the StrstrIndication interface, which the hardware invokes (remotely) to notify the software of configuration completion or search results. The interface compiler generates a separate portal for each of these interfaces. Within each portal, a dedicated unidirectional FIFO is assigned to each logical interface method.

In our initial implementation the accelerator does not access system memory directly, so the search string is transmitted to the accelerator one character at a time via the {tt setupNeedle} method. We will see in Section :ref:Sec-StringSearchSystemMemory‘ how to use a pointer to system memory instead.

Invoking Hardware from Software

Because the StrStrRequest functionality is implemented in hardware, the Connectal interface compiler generates a C++ textbf{proxy} with the following interface to be invoked by the application software:

```
class StrStrRequestProxy : public Portal {
public:
  void setupNeedle(uint32_t needleChars);
  void search(uint32_t haystackPtr,
              uint32_t haystackLen);
};
```

The implementation of StrStrRequestProxy marshals the arguments of each method and en-queues them directly into their dedicated hardware FIFOs. To execute searches in the FPGA fabric over data stored in flash memory, the software developer simply instantiates *StrStrRequestProxy* and invokes its methods:

```
StrStrRequestProxy *proxy =
            new StrStrRequestProxy(...);
proxy->search(haystackPtr, haystackLen);
```

On the FPGA, the user implements the application logic as a BSV module with the StrStrRequest interface. A *wrapper* is generated by the interface compiler to connect this module to the hardware FIFOs. The wrapper unmarshals messages that it receives and then invokes the appropriate method in the StrStrRequest interface. Here is the BSV code that instantiates the generated wrapper and connects it to the user's texttt{mkStrStr} module:

```
StrStrRequest strStr <- mkStrStr(...);
StrStrRequestWrapper wrapper <-
    mkStrStrRequestWrapper(strStr);
```

Figure **:ref:'Fig-msc1'_** shows how all the pieces of an application implemented using Connectal work together when hardware functionality is invoked remotely from software. Direct access to the memory mapped hardware FIFOs by the generated proxy running in user-mode is key to the efficiency of our implementation strategy.

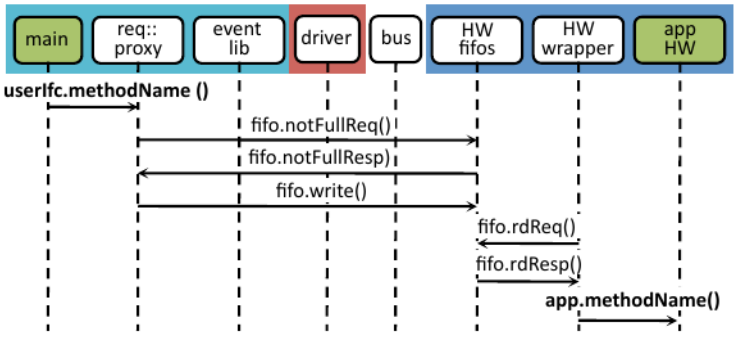


Figure 1.1: SW invokes HW: *main* and *app HW* are implemented by the user.

Invoking Software from Hardware

Invoking software from hardware takes a slightly different form, due primarily to the fact that "main" is still owned by software. Since the direction of the remote invocation is reversed, the proxy on this path will be instantiated on the FPGA and the wrapper instantiated on host side. The user implements the StrStrResponse interface in software and connects it to the generated wrapper using C++ subclasses:

```
class StrStrResponse:
  public StrStrResponseWrapper {
    ...
  void searchResult(int32_t v) {...}
}
```

The StrStrResponseWrapper constructor registers a pointer to the object with the event library which keeps track of all instantiated software wrappers. The wrapper implementation unmarshals messages sent through the hardware FIFOs and invokes the appropriate subclass interface method. To activate this path, main simply instantiates the response implementation and invokes the library event handler:

```
StrStrResponse *response =
  new StrStrResponse(...);
while(1)
  portalExec_event();
```

On the invocation side, the interface compiler generates a proxy which the application logic instantiates and invokes directly:

```
StrStrResponseProxy proxy <-
     mkStrStrRequestProxy();
StrStrRequest strStr <-
     mkStrStr(... proxy.ifc ...);
```

Figure :ref:Fig-msc0 shows how all the pieces of an application collaborate when software functionality is being invoked from hardware.

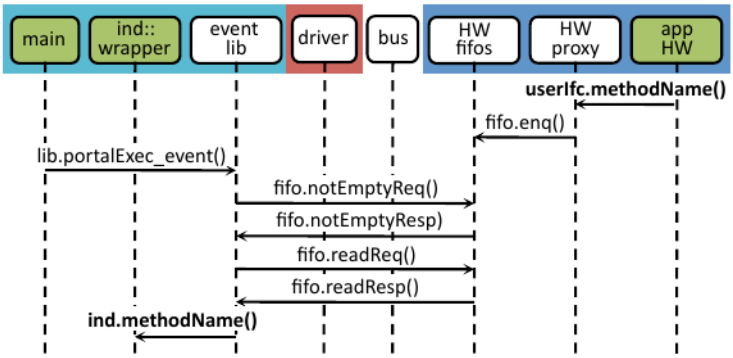


Figure 1.2: HW invokes SW: ‘main’, ‘ind::wrapper’, and ‘app HW’ are implemented by the user.

The simplest software execution environment for the string search accelerator is to have a single thread making requests and waiting for responses as follows:

```
void search(char *str){
  StrStrRequestProxy *req =
    new StrStrRequestProxy(...);
  StrStrResponse *resp =
    new StrStrResponse(...);
  while (char c = *str++)
    req->setupNeedle(c);
  // start search
  req->search(...);
  // handle responses from the HW
  while(1)
```

```
    portalExec_event();
}
```

The call to :c:func:portalExec_event() checks for a response from HW. If there is a pending response, it invokes the method corresponding to that FIFO in the wrapper class. This generated method reads out a complete message from the FIFO and unmarshals it before invoking the user-defined call-back function, which in this case would be texttt{StrStrResponse::searchResult}.

Connecting To Flash

On BlueDBM, one of our target platforms, the flash memory array is connected directly to the FPGA chip, and DDR signals are used to read/write/erase flash memory cells. The RTL required to communicate with the memory requires some commonly used functionality, such as *SerDes* and DDR controllers, both of which are included in the BSV libraries distributed as part of the Connectal framework.

Multithreading The Software

In many cases, we would like to avoid a hardware-to-software path which requires the software to poll a hardware register on the other side of a bus for relatively infrequent events. To accommodate this, the Connectal framework generates interrupts which are raised when hardware invokes software interface methods. The generic Connectal driver connects these signals to the Linux kernel and the software wrappers can exploit then by calling poll. Connectal applications often use a separate thread to execute hardware-to-software asynchronous invocations, since dedicated thread can put itself to sleep until the hardware raises an interrupt. The ‘‘main” thread is free to do other work and can communicate with the ‘‘indication” thread using a semaphore as shown below:

```
class StrStrResponse:
    public StrStrResponseWrapper {
  sem_t sem;
  int v;
  void searchResult(int32_t v) {
    this->response = v;
    sem_post(&sem);
  }
  void waitResponse(){sem_wait(&sem);}
};
StrStrResponse *resp;
StrStrRequestProxy *req;
int search(char *str){
  while (char c = *str++)
    req->setupNeedle(c);
  // start search
  req->search(...);
  // wait for response
  resp->waitResponse();
  // return result
  return resp->v;
}
```

The polling thread is started by a call to :c:func:portalExec_start(), which ultimately invokes the :c:func:portalExec_poll() function implemented in the Connectal event library. :c:func:portalExec_poll() invokes the system call textbf{poll} on the FDs corresponding to all the indication or response portals, putting itself to sleep. When an interface method is invoked in the hardware proxy, an interrupt is raised, waking the indication thread. A register is read which indicates which method is being called and the corresponding wrapper method is invoked to read/marshal the arguments and invoke the actual user-defined methods. Figure *Fig-msc2* shows this process. Multithreading often

leads to simultaneous access to shared hardware resources. If a software solution to protect these resources (such as mutex) is not available, the hardware interface can be refactored into separate portals, one for each control thread.

Each interface will generate a separate Portal which is assigned its own address space and Linux device. Using Linux devices in this way enables access control restrictions to be specified individually for each portal. This feature can be used to grant different users or processes exclusive access and prevent unauthorized access to specific pieces of hardware functionality.

Shared Access to Host Memory

In the first three refinements presented in Section *Accelerating String Search*, all communication between hardware and software takes place through register-mapped IO. The final refinement in Section *Sec-StrStrDma* is to grant hardware and software shared access to host memory. The interface to the search accelerator shown below has been updated to use direct access to system memory for the search strings:

```
interface StrstrRequest;
  method Action setup(Bit#(32) needlePtr,
                      Bit#(32) mpNextPtr,
                      Bit#(32) needleLen);
  method Action search(Bit#(32) haystackPtr,
                       Bit#(32) haystackLen,
                       Bit#(32) iterCount);
endinterface
interface StrstrIndication;
   method Action searchResult(Int#(32) v);
   method Action setupComplete();
endinterface
```

In order to share memory with hardware accelerators, it needs to be allocated using :c:func:portalAlloc(). Here is the search function updated accordingly:

```
int search(char *str){
  int size = strlen(str)+1;
  int fd = portalAlloc(size);
  char *sharedStr = portalMmap(fd, size);
  strcpy(sharedStr, str);
  // send a DMA reference to the search pattern
  req->needle(dma->reference(fd), size);
  // start search
  req->search(...);
  resp->waitResponse();
  ... unmap and free the string
  return resp->v;
}
```

The application allocates shared memory via {tt portalAlloc}, which returns a file descriptor, and then passes that file descriptor to {tt mmap}, which maps the physical pages into the application's address space. The file descriptor corresponds to a dmabufcite{dmabuf}, which is a standard Linux kernel mechanism.

To share that memory with the accelerator, the application calls {tt reference}, which sends a logical to physical address

mapping to the hardware's address translator. The call to {tt reference} returns a handle, which the application sends to

the accelerator. Connectal's BSV libraries for DMA enable the accelerator to read or write from offsets to these handles, taking care of address translation transparently.

To fully exploit the data parallelism, {tt mkStrStr} partitions the search space into p partitions. It instantiates two memory read trees from the Connectal library ({tt MemreadEngineV}, discussed in Section *Distributed Access to Memory Ports*, each with p read servers. One set is used by the search kernels to read the configuration data from the host memory, while the other is used to read the ‘‘haystack” from flash.

On supported platforms such as Zynq which provide multiple physical master connections to system memory, Connectal interleaves DMA requests over the parallel links. It does this on a per-read-client basis, rather than a per-request basis.

Alternate Portal Implementations

Connectal separates the generation of code for marshalling and unmarshaling method arguments from the transport mechanism used to transmit the messages. This separation enables ‘‘swappable” application-specific transport libraries. In light of this, a large number of transport mechanism can be considered. Switching between mechanism requires a simple directive in the project Makefile (more details are given in Section *Sec-ToolChain*).

By default, each portal is mapped to a region of address space and a memory-mapped FIFO channel is generated for each method. Though software access to all FIFO channels in a design may occur through single bus slave interface, Connectal libraries implement their multiplexing to ensure that each FIFO is independent, allowing concurrent access to different methods from multiple threads or processes.

The default portal library implements the method FIFOs in the hardware accelerator. This provides the lowest latency path between hardware and software, taking about 1 microsecond to send a message. If higher bandwidth or transaction rates are needed, FIFOs implemented as a ring buffer in DRAM can be used instead. This requires more instructions per message send and receive, but may achieve higher throughput between the CPU and hardware.

During the design exploration process, a component originally implemented on the FPGA may migrate to software running on the host processor. Remote invocations which were originally from software to hardware must be recast as software to software. Without changing the IDL specification, the transport mechanism assigned to a portal can be re-specified to implement communication between software components running either on the same host or across a network.

Connectal uses UNIX sockets or shared memory to transport messages between the application software components or the hardware simulator. In other situations, TCP or UDP can be used to transport the messages to hardware running on another machine. Viable connections to the FPGA board range from low-speed interconnects such as JTAG, SPI, to higher-speed interconnects such as USB or Aurora over multi-gigabit per second transceivers.

1.2.6 Workflow using Connectal

In this section, we give an overview of the Connectal workflow and toolchain. The complete toolchain, libraries, and many running examples may be obtained at textit{www.connectal.org} or by emailing textit{connectal@googlegroups.com}.

Top level structure of Connectal applications

The simplest Connectal application consists of 4 files:

Makefile

The top-level Makefile defines parameters for the entire application build process. In its simplest form, it specifies which Bluespec interfaces to use as portals, the hardware and software source files, and the libraries to use for the hardware and software compilation.

Application Hardware

Connectal applications typically have at least one BSV file containing declarations of the interfaces being exposed as portals, along with the implementation of the application hardware itself.

Top.bsv

In this file, the developer instantiates the application hardware modules, connecting them to the generated wrappers and proxies for the portals exported to software. To connect to the host processor bus, a parameterized standard interface is used, making it easy to synthesize the application for different CPUs or for simulation. If CPU specific interface signals are needed by the design (for example, extra clocks that are generated by the PCIe core), then an optional CPU-specific interface can also be used.

If the design uses multiple clock domains or additional pins on the FPGA, those connections are also made here by exporting a 'Pins' interface. The Bluespec compiler generates a Verilog module from the top level BSV module, in which the methods of exposed interfaces are implemented as Verilog ports. Those ports are associated to physical pins on the FPGA using a physical constraints file.

Application CPP

The software portion of a Connectal application generally consists of at least one C++ file, which instantiates the generated software portal wrapper and proxies. The application software is also responsible for implementing main.

Development cycle

After creating or editing the source code for the application, the development cycle consists of four steps: generating makefiles, compiling the interface, building the application, and running the application.

Generating Makefiles

Given the parameters specified in the application Makefile and a platform target specified at the command line, Connectal generates a target-specific Makefile to control the build process. This Makefile contains the complete dependency information for the generation of wrappers/proxies, the use of these wrappers/proxies in compiling both the software and hardware, and the collection of build artifacts into a package that can be either run locally or over a network to a remote 'device under test' machine.

Compiling the Interface

The Connectal interface compiler generates the C++ and BSV files to implement wrappers and proxies for all interfaces specified in the application Makefile. Human readable textbf{JSON} is used as an intermediate representation of portal interfaces, exposing a useful debugging window as well as a path for future support of additional languages and IDLs.

Building the Application

A target in the generated Makefile invokes GCC to compiler the software components of the application. The Bluespec compiler (bsc) is then invoked to compiler the hardware components to Verilog. A parameterized Tcl scripts is used to drive Vivado to build the Xilinx FPGA configuration bitstream for the design.

A Connectal utility called fpgamake supports specification of which Bluespec and Verilog modules should be compiled to separate netlists and to enable separate place and route of those netlists given a floor plan. Separate synthesis and floor planning in this manner can reduce build times, and to make it easier to meet timing constraints.

Another Connectal utility called buildcache speeds recompilation by caching previous compilation results and detecting cases where input files have not changed. Although similar to the better-known utility textit{ccache}, this program has no specific knowledge of the tools being executed, allowing it to be integrated into any workflow and any tool set. This utility uses the system call textbf{strace} to track which files are read and written by each build step, computing an 'input signature' of the MD5 checksum for each of these files. When the input signature matches, the output files are just refreshed from the cache, avoiding the long synthesis times for the unchanged portions of the project.

Running the Application

As part of our goal to have a fully scripted design flow, the generated Makefile includes a texttt{run} target that will program the FPGA and launch the specified application or test bench. In order to support shared target hardware resources, the developer can direct the run to a particular machines, which can be accessed over the network. For Ubuntu target machines, ssh is used to copy/run the application. For Android target machines, 'adb' is used.

Continuous Integration and Debug Support

Connectal provides a fully scripted flow in order to make it easy to automate the building and running of applications for continuous integration. Our development team builds and runs large collections of tests whenever the source code repository is updated.

Connectal also provides trace ring buffers in hardware and analysis software to trace and display the last transactions on the PCIe or AXI memory bus. This trace is useful when debugging performance or correctness problems, answering questions of the form:

- What were the last memory requests and responses?
- What was the timing of the last memory request and responses?
- What were the last hardware method invocations or indications?

1.2.7 Performance of Generated Systems

A framework is only useful if it reduces the effort required by developers to achieve the desired performance objective. Trying to gauge the relative effort is difficult since the authors implemented both the framework and the running example. On PCIE-based platforms we were able to reduce the time required to search for a fixed set of strings in a large corpus by an order of magnitude after integrating hardware acceleration using Connectal. Performance improvements on the Zynq-based platforms was even greater due to the relative processing power of the ARM CPU and scaled with the number of bus master interfaced used for DMA. In the Connectal framework, developing these applications took very little time.

Performance of Portals

The current implementation of HW/SW textbf{portal} transfers 32 bits per FPGA clock cycle. Our example designs run at 100MHz to 250MHz, depending on the complexity of the design and the speed grade of the FPGA used. Due to their intended use, the important performance metric of Portals is latency. These values are given in Figure~ref{Fig:PortalLatency}.

begin{figure} centering begin{tabular}{|c|c|c|c|c|c|c|c|c|c|}

hline & rt{KC705} & rt{VC707} & rt{ZYBO} & rt{Zedboard} & rt{ZC702} & rt{ZC706} & rt{Parallel} & rt{Mini-ITX} \

hline HW $rightarrow$ SW & 3 & 3 & X & 0.80 & 0.80 & 0.65 & X & 0.65 \ hline SW $rightarrow$ HW & 5 & 5 & X & 1.50 & 1.50 & 1.10 & X & 1.10 \ hline

end{tabular} caption{Latency (mus) of communication through portals on supported

platformslabel{Fig:PortalLatency}}

end{figure}

The Xilinx KC705 and VC707 boards connect to x86 CPUs and system memory via PCIe gen1. The default FPGA clock for those boards is 125MHz. The other platforms use AXI to connect the programmable logic to the quad-core ARM Cortex A9 and system memory. The ZYBO, Zedboard and ZC702 use a slower speed grade part on which our designs run at 100MHz. The ZC706 and Mini-ITX use a faster part on which many of our designs run at 200MHz. The lower latency measured on the ZC706 reflects the higher clock speed of the latency performance test.

Performance of Reads/Writes of System Memory

For high bandwidth transfers, we assume the developer will have the application hardware read or write system memory directly. Direct access to memory enables transfers with longer bursts, reducing memory bus protocol overhead. The framework supports transfer widths of 32 to 128 bits per cycle, depending on the interconnect used.

Our goal in the design of the library components used to read and write system memory is to ensure that a developer's application can use all bandwidth available to the FPGA when accessing system memory. DMA Bandwidth on supported platforms is listed in Figureref{Fig:DmaBandwidth}.

begin{figure} centering begin{tabular}{|c|c|c|c|c|c|c|c|c|}

hline & rt{KC705} & rt{VC707} & rt{ZYBO} & rt{Zedboard} & rt{ZC702} & rt{ZC706} & rt{Parallel} & rt{Mini-ITX} \

hline Read & 1.4 & 1.4 & X & 0.8 & 0.8 & 1.6 & X & 1.6 \ hline Write & 1.4 & 1.4 & X & 0.8 & 0.8 & 1.6 & X & 1.6 \ hline

end{tabular} caption{Maximum bandwidth (GB/s) between FPGA and host memory using

Connectal RTL libraries on supported platformslabel{Fig:DmaBandwidth}}

end{figure}

On PCIe systems, Connectal currently supports 8 lane PCIe gen1. We've measured 1.4 gigabytes per second for both reads and writes. Maximum throughput of 8 lane PCIe gen1 is 1.8GB/s, taking into account 1 header transaction per 8 data transactions, where 8 is the maximum number of data transactions per request supported by our server's chipset. The current version of the test needs some more tuning in order to reach the full bandwidth available. In addition, we are in the process of updating to 8 lane PCIe gen2 using newer Xilinx cores.

Zynq systems have four *high performance* ports for accessing system memory. Connectal enables an accelerator to use all four. In our experiments, we have been able to achieve 3.6x higher bandwidth using 4 ports than using 1 port.

1.2.8 Related Work

A number of research projects, such as Lime~cite{REL:Lime}, BCL~cite{King:2012:AGH:2150976.2151011}, HThreads~cite{DBLP:conf/fpl/PeckAASBA06}, and CatapaultC~cite{CatC:www} (to name just a few) bridge the software/hardware development gap by providing a single language for developing both the software and hardware components of the design. In addition, Altera and Xilinx have both implemented OpenCL~cite{Opencl} on FPGAs~cite{AlteraOpencl,XilinxOpencl} in an attempt to attract GPU programmers.

The computation model of software differs significantly from that of hardware, and so far none of the unified language approaches deliver the same performance as languages designed specifically for hardware or software. Connectal is intended to be used for the design of performance-critical systems. In this context we think that designers prefer a mix of languages specifically designed for their respective implementation contexts.

Infrastructures such as LEAP~cite{DBLP:conf/fpl/FlemingYAE14}, Rainbow~cite{DBLP:journals/ijrc/JozwikHETT13}, and OmpSs~cite{DBLP:conf/fpga/FilguerasGJAMLNV14} (to name just a few) use resource abstraction to enable FPGA development. We found that in their intended context, these tools were easy to use but that performance tuning in applications not foreseen by the infrastructure developers was problematic.

Some projects such as TMD-MPI~cite{DBLP:journals/trets/SaldanaPMNWCWSP10}, VFORCE/ \VSIPL++~cite{DBLP:journals/jpdc/MooreLK12}, and GASNet/GAScore~cite{DBLP:conf/fpga/WillenbergC13} target only the hardware software interface. These tools provide message passing capabilities, but rely on purely operational semantics to describe the HW/SW interface. Apart from the implementation details, Connectal distinguishes itself by using an IDL to enforce denotational interface semantics.

UIO~cite{UIO:Howto} is a user-space device driver framework for Linux. It is very similar to the Connectal's portal device driver, but it does not provide a solution to multiple device nodes per hardware device. The portal driver provides this so that different interfaces of a design may be accessed independently, providing process boundary protection, thread safety, and the ability for user processes and the kernel both to access the hardware device.

1.2.9 Conclusion

Connectal bridges the gap between software and hardware development, enabling developers to create integrated solutions rapidly. With Connectal, we take a pragmatic approach to software and hardware development in which we try to avoid any dependence on proposed solutions to open research problems.

Use of Connectal's interface compiler ensures that software and hardware remain consistent and make it easy to update the hardware/software boundary as needed in a variety of execution contexts. The generated portals permit concurrent and low-latency access to the accelerator and enable different processes or the kernel to have safe isolated access through dedicated interfaces. Support for sharing memory between software and hardware makes it easy to achieve high transfer speeds between the two environments.

Connectal supports Linux and Android operating systems running on x86 and ARM CPUs. It currently supports Xilinx FPGAs and runs on the full range of Xilinx Series 7 devices. Our fully-scripted development process enables the use of continuous integration of software and hardware development. Integrating software development early makes it easier to ensure that the complete solution actually meets design targets and customer requirements.

1.2.10 Portal Interface Structure

Connectal connects software and hardware via portals, where each portal is an interface that allows one side to invoke methods on the other side.

We generally call a portal from software to hardware to be a "request" and from hardware to software to be an "indication" interface:

```
Sequence Diagram to be drawn
{
  SW; HW
  SW -> HW [label = "request"];
  SW <- HW [label = "indication"];
}
```

A portal is conceptually a FIFO, where the arguments to a method are packaged as a message. CONNECTAL generates a "proxy" that marshalls the arguments to the method into a message and a "wrapper" that unpacks the arguments and invokes the method.

Currently, connectal Includes a library that implements portals from software to hardware via memory mapped hardware FIFOs.

Portal Device Drivers

Connectal uses a platform-specific driver to enable user-space applications to memory-map each portal used by the application and to enable the application to wait for interrupts from the hardware.

indexterm:pcieportal indexterm:zynqportal

- pcieportal.ko
- zynqportal.ko

Connectal also uses a generic driver to enable the applications to allocate DRAM that will be shared with the hardware and to send the memory mapping of that memory to the hardware.

- portalmem.ko

Portal Memory Map

Connectal currently supports up to 16 portals connected between software and hardware, for a total of 1MB of address space.

Base address	Function
0x00000	Portal 0
0x10000	Portal 1
0x20000	Portal 2
0x30000	Portal 3
0x40000	Portal 4
0x50000	Portal 5
0x60000	Portal 6
0x70000	Portal 7
0x80000	Portal 8
0x90000	Portal 9
0xa0000	Portal 10
0xb0000	Portal 11
0xc0000	Portal 12
0xd0000	Portal 13
0xe0000	Portal 14
0xf0000	Portal 15

Each portal uses 64KB of address space, consisting of a control register region and then per-method FIFOs.

Base address	Function
0x0000	Portal control regs
0x0100	Method 0 FIFO
0x0200	Method 1 FIFO
...	...

For request portals, the FIFOs are from software to hardware, and for indication portals the FIFOs are from hardware to software.

Portal FIFOs

Base address	Function
0x00	FIFO data (write request data, read indication data)
0x04	Request FIFO not full / Indication FIFO not empty

Portal Control Registers

Base address	Function	Description
0x00	Interrupt status register	1 if this portal has any messages ready, 0 otherwise
0x04	Interrupt enable register	Write 1 to enable interrupts, 0 to disable
0x08	7	Fixed value
0x0C	Ready Channel number + 1	Reads as zero if no indication channel ready
0x10	Interface Id	
0x14	Last portal	1 if this is the last portal defined
0x18	Cycle count LSW	Snapshots MSW when read
0x1C	Cycle count MSW	MSW of cycle count when LSW was read

1.2.11 What is Connectal?

Connectal provides a hardware-software interface for applications split between user mode code and custom hardware in an FPGA or ASIC.

Connectal can automaticaly build the software and hardware glue for a message based interface and also provides for configuring and using shared memory between applications and hardware. Communications between hardware and software are provided by a bidirectional flow of events and regions of memory shared between hardware and software. Events from software to hardware are called requests and events from hardware to software are called indications, but in fact they are symmetric.

Lexicon

connectal:: The name of the project, whose goal is to ease the task of building applications composed of hardware and software components. Programmers use bsv as an IDL to specify the interface between the hardware and software components. A combination of generated code and libraries coordinate the data-flow between the program modules. Because the HW and SW stacks are customized for each application, the overheads associated with communicating across the HW/SW boundary are low.

HW/SW interface :: portal

bsv:: Bluespec System Verilog. bsv is a language for describing hardware that is might higher level than verilog. See {bsvdocumentation}[BSV Documentation] and {bluespecdotcom}[Bluespec, Inc].

bluespec:: Shorthand for Bluespec System Verilog (bsv)

indexterm:portal portal:: a logical request/indication pair is referred to as a portal. current tools require their specification in the IDL to be syntactically identifiable (i.e. fooRequest/fooIndication). An application can make use of multiple portals, which may be specified independently.

request interface:: These methods are implemented by the application hardware to be invoked by application software. A bsv interface consisting of 'Action' methods. Because of the 'Action' type, data flow across this interface is unidirectional (SW -> HW).

indication interface:: The dual of a request interface, indication interfaces are 'Action' methods implemented by application software to be invoked by application hardware. As with request interfaces, the data flow across this interface is unidirectional, but in the opposite direction.

pcieportal/zynqportal:: these two loadable kernel modules implement the minimal set of driver functionality. Specifically, they expose portal HW registers to SW through mmap, and set up interrupts to notify SW that an indication method has been invoked by HW.

portalalloc:: This loadable kernel module exposes a subset of dma-buf functionality to user-space software (though a set of ioctl commands) to allocate and manage memory regions which can be shared between SW and HW processes. Maintaining coherence of the allocated buffers between processes is not automatic: ioctl commands for flush/invalidate are provided to be invoked explicitly by the users if necessary.

connectalgen:: The name of the interface compiler which takes as input the bsv interface specification along with a description of a target platform and generates logic in both HW and SW to support this interface across the communication fabric.

Example setups:

A zedboard (http://www.zedboard.org/), with Android running on the embedded ARM processors (the Processing System 7), an application running as a user process, and custom hardware configured into the Programmable Logic FPGA.

An x86 server, with Linux running on the host processor, an application running partly as a user process on the host and partly as hardware configured into an FPGA connected by PCI express (such as the Xilinx VC707 (http://www.xilinx.com/products/boards-and-kits/EK-V7-VC707-G.htm).

Background

When running part or all of an application in an FPGA, it is usually necessary to communicate between code running in user mode on the host and the hardware. Typically this has been accomplished by custom device drivers in the OS, or by shared memory mapped between the software and the hardware, or both. Shared memory has been particularly troublesome under Linux or Android, because devices frequently require contiguous memory, and the mechanisms for guaranteeing successful memory allocation often require reserving the maximum amount of memory at boot time.

Portal tries to provide convenient solutions to these problems in a portable way.

It is desirable to have

- low latency for small messages
- high bandwidth for large messages
- notification of arriving messages
- asynchronous replies to messages
- support for hardware simulation by a separate user mode process
- support for shared memory (DMA) between hardware and software

Overview

Portal is implemented as a loadable kernel module device driver for Linux/Android and a set of tools to automatically construct the hardware and software glue necessary for communications.

Short messages are handled by programmed I/O. The message interface from software to hardware (so called "requests") is defined as a bsv interface containing a number of Action methods, each with a name and typed arguments.

The interface generator creates all the software and hardware glue so that software invocations of the interface stubs flow through to, and are turned into bsv invocations of the matching hardware. The machinery does not have flow control. Software is responsible for not overrunning the hardware. There is a debug mechanism which will return the request type of a failed method, but it does not tell which invocation failed. Hardware to software interfaces (so called "indications") are likewise defined by bsv interfaces containing Action methods. Hardware invocations of these methods flow through to and cause software calls to corresponding user-supplied functions. In the current implementation there is flow control, in that the hardware will stall until there is room for a hardware to software message. There is also a mechanism for software to report a failure, and there is machinery for these failures to be returned to the hardware.

{ // edge label SW -> HW [label = "request"]; SW <- HW [label = "indication"];

Portals do not have to be structured as request/response. Hardware can send messages to software without a prior request from software.

{ // edge label SW <- HW [label = "indication"];

Incoming messages can cause host interrupts, which wake up the device driver, which can wake up the user mode application by using the select(2) or poll(2) interfaces.

Most of the time, communications between hardware and software will proceed without requiring use of the OS. User code will read and write directly to memory mapped I/O space. Library code will poll for incoming messages, and [true? eventually time out and call poll(2). Only when poll(2) or select(2) are called will the device driver enable hardware interrupts. Thus interrupts are only used to wake up software after a quiet period.

The designer specifies a set of hardware functions that can be called from software, and a set of actions that the hardware can take which result in messages to software. Portal tools take this specification and build software glue modules to translate software function calls into I/O writes to hardware registers, and to report hardware events to software.

For larger memory and OS bypass (OS bypass means letting the user mode application talk directly to the hardware without using the OS except for setup), portal implements shared memory. Portal memory objects are allocated by the user mode program, and appear as Linux file descriptors. The user can mmap(2) the file to obtain user mode access to the shared memory region. Portal does not assure that the memory is physically contiguous, but does pin it to prevent the OS from reusing the memory. An FPGA DMA controller module is provided that gives the illusion of contiguous memory to application hardware, while under the covers using a translation table of scattered addresses.

The physical addresses are provided to the user code in order to initialize the dma controller, and address "handles" are provided for the application hardware to use.

The DMA controller provides Bluespec objects that support streaming access with automatic page crossings, or random access.

An Example

An application developer will typically write the hardware part of the application in Bluespec and the software part of the application in C or C++. In a short example, there will be a bsv source file for the hardware and a cpp source file for the application.

The application developer is free to specify whatever hardware-software interface makes sense.

Refer to https://github.com/cambridgehackers/connectal

In the examples directory, see [simple](../examples/simple/). The file [Simple.bsv](../examples/simple/Simple.bsv) defines the hardware, and testsimple.cpp supplies the software part. In this case, the software part is a test framework for the hardware.

Simple.bsv declares a few *struct* and *enum* types:

```
typedef struct{
   Bit#(32) a;
   Bit#(32) b;
   } S1 deriving (Bits);

typedef struct{
   Bit#(32) a;
   Bit#(16) b;
   Bit#(7) c;
   } S2 deriving (Bits);

typedef enum {
   E1Choice1,
   E1Choice2,
   E1Choice3
   } E1 deriving (Bits,Eq);

typedef struct{
   Bit#(32) a;
   E1 e1;
   } S3 deriving (Bits);
```

Simple.bsv defines the actions (called Requests) that software can use to cause the hardware to act, and defines the notifications (called Indications) that the hardware can use to signal the software.

```
interface SimpleIndication;
    method Action heard1(Bit#(32) v);
    method Action heard2(Bit#(16) a, Bit#(16) b);
    method Action heard3(S1 v);
    method Action heard4(S2 v);
    method Action heard5(Bit#(32) a, Bit#(64) b, Bit#(32) c);
    method Action heard6(Bit#(32) a, Bit#(40) b, Bit#(32) c);
    method Action heard7(Bit#(32) a, E1 e1);
endinterface

interface SimpleRequest;
    method Action say1(Bit#(32) v);
    method Action say2(Bit#(16) a, Bit#(16) b);
    method Action say3(S1 v);
    method Action say4(S2 v);
    method Action say5(Bit#(32)a, Bit#(64) b, Bit#(32) c);
    method Action say6(Bit#(32)a, Bit#(40) b, Bit#(32) c);
    method Action say7(S3 v);
endinterface
```

Software can start the hardware working via say, say2, ... Hardware signals back to software with heard and heard2 and so fort. In the case of this example, say and say2 merely echo their arguments back to software.

The definitions in the bsv file are used by the connectal infrastructure (a python program) to automatically create corresponding c++ interfaces.:

```
../../connectalgen -Bbluesim -p bluesim -x mkBsimTop \
     -s2h SimpleRequest \
     -h2s SimpleIndication \
     -s testsimple.cpp \
     -t ../../bsv/BsimTop.bsv  Simple.bsv Top.bsv
```

The tools have to be told which interface records should be used for Software to Hardware messages and which should be used for Hardware to Software messages. These interfaces are given on the command line for genxpprojfrombsv

connectalgen constructs all the hardware and software modules needed to wire up portals. This is sort of like an RPC compiler for the hardware-software interface. However, unlike an RPC each method is asynchronous.

The user must also create a toplevel bsv module Top.bsv, which instantiates the user portals, the standard hardware environment, and any additional hardware modules.

Rather than constructing the *makefilegen* command line from scratch, the examples in connectal use include *Makefile.connectal* and define some *make* variables.

Here is the Makefile for the *simple* example:

```
CONNECTALDIR?=../..
INTERFACES = SimpleRequest SimpleIndication
BSVFILES = Simple.bsv Top.bsv
CPPFILES=testsimple.cpp

include $(CONNECTALDIR)/Makefile.connectal
```

Designs outside the connectal directory using *connectal* may also include *Makefile.connectal*:

```
CONNECTALDIR?=/scratch/connectal
INTERFACES = ...
BSVFILES = ...
CPPFILES = ...
include $(CONNECTALDIR)/Makefile.connectal
```

simple/Top.bsv

Each CONNECTAL design implements [Top.bsv](../examples/simple/Top.bsv) with some standard components.

It defines the *IfcNames* enum, for use in identifying the portals between software and hardware:

```
typedef enum {SimpleIndication, SimpleRequest} IfcNames deriving (Eq,Bits);
```

It defines *mkConnectalTop*, which instantiates the wrappers, proxies, and the design itself:

```
module mkConnectalTop(StdConnectalTop#(addrWidth));
```

:bsv:module:StdConnectalTop is parameterized by *addrWidth* because Zynq and x86 have different width addressing. *StdConnectalTop* is a typedef:

```
typedef ConnectalTop#(addrWidth,64,Empty)     StdConnectalTop#(numeric type addrWidth);
```

The "64" specifies the data width and *Empty* specifies the empty interface is exposed as pins from the design. In designs using HDMI, for example, *Empty* is replaced by *HDMI*. On some platforms, the design may be able to use different data widths, such as 128 bits on x86/PCIe.

Next, *mkConnectalTop* instantiates user portals:

```
// instantiate user portals
   SimpleIndicationProxy simpleIndicationProxy <- mkSimpleIndicationProxy(SimpleIndication);
```

Instantiate the design:

```
SimpleRequest simpleRequest <- mkSimpleRequest(simpleIndicationProxy.ifc);
```

Instantiate the wrapper for the design:

```
SimpleRequestWrapper simpleRequestWrapper <- mkSimpleRequestWrapper(SimpleRequest,simpleRequest);
```

Collect the portals into a vector:

```
Vector#(2,StdPortal) portals;
portals[0] = simpleRequestWrapper.portalIfc;
portals[1] = simpleIndicationProxy.portalIfc;
```

Create an interrupt multiplexer from the vector of portals:

```
let interrupt_mux <- mkInterruptMux(portals);
```

Create the system directory, which is used by software to locate each portal via the *IfcNames* enum:

```
// instantiate system directory
StdDirectory dir <- mkStdDirectory(portals);
let ctrl_mux <- mkAxiSlaveMux(dir,portals);
```

The following generic interfaces are used by the platform specific top BSV module:

```
   interface interrupt = interrupt_mux;
   interface ctrl = ctrl_mux;
   interface m_axi = null_axi_master;
   interface leds = echoRequestInternal.leds;

endmodule : mkConnectalTop
```

simple/testsimple.cpp

CONNECTAL generates header files declaring wrappers for hardware-to-software interfaces and proxies for software-to-hardware interfaces. These will be in the "jni/" subdirectory of the project directory.

```
#include "SimpleIndication.h"
#include "SimpleRequest.h"
```

It also declares software equivalents for structs and enums declared in the processed BSV files:

```
#include "GeneratedTypes.h"
```

CONNECTAL generates abstract virtual base classes for each Indication interface.

```
class SimpleIndicationWrapper : public Portal {

public:
    ...
    SimpleIndicationWrapper(int id, PortalPoller *poller = 0);
    virtual void heard1 ( const uint32_t v )= 0;
    ...
};
```

Implement subclasses of the wrapper in order to define the callbacks:

```
class SimpleIndication : public SimpleIndicationWrapper
{
public:
  ...
    virtual void heard1(uint32_t a) {
      fprintf(stderr, "heard1(%d)\n", a);
      assert(a == v1a);
      incr_cnt();
    }
    ...
};
```

To connect these classes to the hardware, instantiate them using the *IfcNames* enum identifiers. CONNECTAL prepends the name of the type because C++ does not support overloading of enum tags.

```
SimpleIndication *indication = new SimpleIndication(IfcNames_SimpleIndication);
SimpleRequestProxy *device = new SimpleRequestProxy(IfcNames_SimpleRequest);
```

Create a thread for handling notifications from hardware:

```
pthread_t tid;
if(pthread_create(&tid, NULL,  portalExec, NULL)){
  exit(1);
}
```

Now the software invokes hardware methods via the proxy:

```
device->say1(v1a);

device->say2(v2a,v2b);
```

Simple Example Design Structure

The *simple* example consists of the following files:

```
Simple.bsv
Makefile
Top.bsv
testsimple.cpp
```

After running *make BOARD=zedboard verilog* in the *simple* directory, the *zedboard* project directory is created, populated by the generated files.

A top level *Makefile* is created:

```
zedboard/Makefile
```

makefilegen generates wrappers for software-to-hardware interfaces and proxies for hardware-to-software interfaces:

```
zedboard/sources/mkzynqtop/SimpleIndicationProxy.bsv
zedboard/sources/mkzynqtop/SimpleRequestWrapper.bsv
```

Connectal supports Android on Zynq platforms, so connectalgen generates *jni/Android.mk* for *ndk-build*:

```
zedboard/jni/Android.mk
zedboard/jni/Application.mk
```

Connectal generates *jni/Makefile* to compile the software for PCIe platforms (vc707 and kc705):

    zedboard/jni/Makefile

CONNECTAL generates software proxies for software-to-hardware interfaces and software wrappers for hardware-to-software interfaces:

```
zedboard/jni/SimpleIndication.h
zedboard/jni/SimpleIndication.cpp
zedboard/jni/SimpleRequest.cpp
zedboard/jni/SimpleRequest.h
```

CONNECTAL also generates *GeneratedTypes.h* for struct and enum types in the processed BSV source files:

```
zedboard/jni/GeneratedTypes.h
```

CONNECTAL copies in standard and specified constraints files:

```
zedboard/constraints/design_1_processing_system7_1_0.xdc
zedboard/constraints/zedboard.xdc
```

CONNECTAL generates several TCL files to run *vivado*.

The *board.tcl* file specifies *partname*, *boardname*, and *connectaldir* for the other TCL scripts.:

```
zedboard/board.tcl
```

To generate an FPGA bit file, run *make bits*. This runs vivado with the *mkzynqtop-impl.tcl* script.:

```
zedboard/mkzynqtop-impl.tcl
```

make verilog

Compiling to verilog results in the following verilog files:

```
zedboard/verilog/top/mkSimpleIndicationProxySynth.v
zedboard/verilog/top/mkZynqTop.v
```

Verilog library files referenced in the design are copied for use in synthesis.:

```
zedboard/verilog/top/FIFO1.v
...
```

make bits

Running *make bits* in the zedboard directory results in timing reports:

```
zedboard/bin/mkzynqtop_post_place_timing_summary.rpt
zedboard/bin/mkzynqtop_post_route_timing_summary.rpt
zedboard/bin/mkzynqtop_post_route_timing.rpt
```

and some design checkpoints:

```
zedboard/hw/mkzynqtop_post_synth.dcp
zedboard/hw/mkzynqtop_post_place.dcp
zedboard/hw/mkzynqtop_post_route.dcp
```

and the FPGA configuration file in .bit and .bin formats:

```
zedboard/hw/mkZynqTop.bit
zedboard/hw/mkZynqTop.bin
```

make android_exe

CONNECTAL supports Android 4.0 on Zynq platforms. It generates *jni/Android.mk* which is used by *ndk-build* to create a native Android executable.:

```
make android_exe
```

This produces the ARM elf executable:

```
libs/armeabi/android_exe
```

make run

For Zynq platforms:

```
make run
```

will copy the Android executable and FPGA configuration file to the target device, program the FPGA, and run the executable. See [run.zedboard](../scripts/run.zedboard) for details.

It uses *checkip* to determine the IP address of the device via a USB console connection to the device (it is built/installed on the host machine from the git repo cambridgehackers/consolable). If the target is not connected to the build machine via USB, specify the IP address of the target manually:

```
make RUNPARAM=ipaddr run
```

For PCIe platforms, *make run* programs the FPGA via USB and runs the software locally.

For bluesim, *make run* invokes bluesim on the design and runs the software locally.

Shared Memory

Shared Memory Hardware

In order to use shared memory, the hardware design instantiates a DMA module in Top.bsv:

```
AxiDmaServer#(addrWidth,64) dma <- mkAxiDmaServer(dmaIndicationProxy.ifc, readClients, writeClients),
```

The AxiDmaServer multiplexes read and write requests from the clients, translates DMA addresses to physical addresses, initiates bus transactions to memory, and delivers responses to the clients.

DMA requests are specified with respect to "portal" memory allocated by software and identified by a *pointer*.

Requests and responses are tagged in order to enable pipelining:

```
typedef struct {
   SGLId pointer;
   Bit#(MemOffsetSize) offset;
   Bit#(8) burstLen;
   Bit#(6)  tag;
   } MemRequest deriving (Bits);

typedef struct {
   Bit#(dsz) data;
   Bit#(6) tag;
   } MemData#(numeric type dsz) deriving (Bits);
```

Read clients implement the *MemReadClient* interface. On response to the read, *burstLen MemData* items will be put to the *readData* interface. The design must be ready to consume the data when it is delivered from the memory bus or the system may hang:

```
interface MemReadClient#(numeric type dsz);
   interface GetF#(MemRequest)    readReq;
   interface PutF#(MemData#(dsz)) readData;
endinterface
```

Write clients implement *MemWriteClient*. To complete the transaction, *burstLen* data items will be consumed from the *writeData* interace. Upon completion of the request, the specified tag will be put to the *writeDone* interface. The data must be available when the write request is issued to the memory bus or the system may hang:

```
interface MemWriteClient#(numeric type dsz);
   interface GetF#(MemRequest)    writeReq;
   interface GetF#(MemData#(dsz)) writeData;
   interface PutF#(Bit#(6))       writeDone;
endinterface
```

A design may implement *MemReadClient* and *MemWriteClient* interfaces directly, or it may instantiate DmaReadBuffer or DmaWriteBuffer.

The *AxiDmaServer* is configured with physical address translations for each region of memory identified by a *pointer*. A design using DMA must export the *DmaConfig* and *DmaIndication* interfaces of the DMA server.

Here are the DMA components of [memread_nobuff/Top.bsv](../examples/memread_nobuff/Top.bsv):

Instantiate the design and its interface wrappers and proxies:

```
MemreadIndicationProxy memreadIndicationProxy <- mkMemreadIndicationProxy(MemreadIndication);
Memread memread <- mkMemread(memreadIndicationProxy.ifc);
MemreadRequestWrapper memreadRequestWrapper <- mkMemreadRequestWrapper(MemreadRequest,memread.request
```

Collect the read and write clients:

```
Vector#(1, MemReadClient#(64)) readClients = cons(memread.dmaClient, nil);
Vector#(0, MemReadClient#(64)) writeClients = nil;
```

Instantiate the DMA server and its wrapper and proxy:

```
DmaIndicationProxy dmaIndicationProxy <- mkDmaIndicationProxy(DmaIndication);
AxiDmaServer#(addrWidth,64) dma <- mkAxiDmaServer(dmaIndicationProxy.ifc, readClients, writeClients);
DmaConfigWrapper dmaConfigWrapper <- mkDmaConfigWrapper(DmaConfig,dma.request);
```

Include *DmaConfig* and *DmaIndication* in the portals of the design:

```
Vector#(4,StdPortal) portals;
portals[0] = memreadRequestWrapper.portalIfc;
portals[1] = memreadIndicationProxy.portalIfc;
portals[2] = dmaConfigWrapper.portalIfc;
portals[3] = dmaIndicationProxy.portalIfc;
```

The code generation tools will then produce the software glue necessary for the shared memory support libraries to initialize the DMA "library module" included in the hardware.

Shared Memory Software

The software side instantiates the DmaConfig proxy and the DmaIndication wrapper:

```
dma = new DmaConfigProxy(IfcNames_DmaConfig);
dmaIndication = new DmaIndication(dma, IfcNames_DmaIndication);
```

Call *dma->alloc()* to allocate DMA memory. Each chunk of portal memory is identified by a file descriptor. Portal memory may be shared with other processes. Portal memory is reference counted according to the number of file descriptors associated with it:

```
PortalAlloc *srcAlloc;
dma->alloc(alloc_sz, &srcAlloc);
```

Memory map it to make it accessible to software:

```
srcBuffer = (unsigned int *)mmap(0, alloc_sz, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_SHARED, srcAlloc->h
```

Connectal is currently using non-snooped interfaces, so the cache must be flushed and invalidated before hardware accesses portal memory:

```
dma->dCacheFlushInval(srcAlloc, srcBuffer);
```

Call *dma->reference()* to get a pointer that may be passed to hardware:

```
unsigned int ref_srcAlloc = dma->reference(srcAlloc);
```

This also transfers the DMA-to-physical address translation information to the hardware via the *DmaConfig* interface:

```
device->startRead(ref_srcAlloc, numWords, burstLen, iterCnt);
```

Notes

stewart notes

Currently there are no valid bits and no protections against bursts crossing page boundaries

There needs to be a way to synchronize Request actions and DMA reads, and to synchronize DMA writes with Indications, so that the writes complete to the coherence point before the indication is delivered to software. One could imagine an absurdly buffered memory interface and a rather direct path for I/O reads that could get out of order.

1.2.12 Interface Declarations

1.2.13 Flow Control

1.2.14 Host Interface

1.3 Connectal Developer's Guide

1.3.1 Connectal Rationale

Are you happy with the connection between software development and hardware development on your projects? Do you wish you had more flexible tools for developing test benches, drivers, and applications using your hardware? Do you wish you didn't need a kernel driver?

Do you wish that you could use a common flow to drive your hardware development, across simulation, timing verification, FPGA implementation, and even the ASIC back end?

Do you wish you could use a better programming language to implement the hardware? Do you miss declared interfaces and structural types?

Do you wish you had better tools for debugging the hardware? For reflecting conditions encountered on the hardware back into the simulation environment where you can drill down into the problem?

These are the kinds of problems we encountered which inspired us to develop the Connectal framework. With Connectal, we are trying to bring the productivity of software engineering practices to hardware development without compromising on the performance of the hardware.

Connectal provides:

- Seamless development environment with test benches and apps written in C/C++
- Streamlined application structure
- Faster builds and greater visibility simulating hardware in bluesim/modelsim/xsim
- Support for capture traces on interfaces and replaying them in test benches

1.3.2 Connectal Project Structure

The set of files composing the input to the Connectal toolchain is referred to as a project. A collection of out-of-tree example projects is available at https://github.com/connectal-examples. To illustrate the structure of a project, this chapter uses the example https://github.com/connectal-examples/leds, which can be executed on the Bluesim or Zynq target platforms.

Project Makefile

The top-level Makefile in the project (in our example, https://github.com/connectal-examples/leds/blob/master/Makefile) defines parameters building and executing the project. In its simplest form, it specifies the hardware and software source files, which Bluespec interfaces to use as interfaces (portals) between them, and the libraries to use for the hardware and software compilation:

```
INTERFACES = LedControllerRequest
BSVFILES = LedController.bsv Top.bsv
CPPFILES= testleds.cpp
NUMBER_OF_MASTERS =0
include \$(CONNECTALDIR)/Makefile.connectal
```

`BSVFILES` is a list of bsv files containing interface definitions used to generate portals and module definitions used to generate HW components. Connectal bsv libraries can be used without being listed explicitly.

`CPPFILES` is a list of C/C++ files containing software components and `main`. The Connectal C/C++ libraries can be used without being listed explicitly.

`INTERFACES` is a list of names of BSV interfaces which may be used to communicate between the HW and SW componentsy. In addition to user-defined interfaces, there are a wide variety of interfaces defined in Connectal libraries which may be included in this list.

`NUMBER_OF_MASTERS` is used to designate the number of host bus masters the hardware components will instantiate. For PCIe-based platforms, this value can be set to 0 or 1, while on Zynq-based platforms values from 0 to 4 are valid.

`CONNECTALDIR` must be set so that the top-level Connectal makefile can be included, defining the makefile build targets for the project. This brings in the default definitions of all project build parameters as well as the Connectal hardware and software libraries. When running the toolchain on AWS, this varible is set automatically in the build environment. (See *Compiling and Running Connectal Project*)

Project Source

Interface Definitions

label{interface_definitions}

When generating portals, the Connectal interface compiler searches the Connectal bsv libraries and the files listed in `BSVFILES` for definitions of the interfaces listed in `INTERFACES`. If the definition of a listed interfaces is not found, an error is reported and the compilation aborts. The interfaces in this list must be composed exclusively of

`Action` methods. Supported method argument types are `Bit\#(n)`, `Bool`, `Int\#(32)`, `UInt\#(32)`, `Float`, `Vector\#(t)`, `enum`, and `struct`.

Software

The software in a Connectal project consists of at least one C++ file which instantiates the generated portal wrappers and proxies and implements `main()`. The following source defines the SW component of the example, which simply toggles LEDs on the Zedboard (url{https://github.com/connectal-examples/leds/blob/master/testleds.cpp}):

```
#include <unistd.h>
#include "LedControllerRequest.h"
#include "GeneratedTypes.h"
int main(int argc, const char **argv)
{
  LedControllerRequestProxy *device =
    new LedControllerRequestProxy(IfcNames_LedControllerRequest);
  for (int i = 0; i < 20; i++) {
    device->setLeds(10, 10000);
    sleep(1);
    device->setLeds(5, 10000);
    sleep(1);
  }
}
```

The makefile listed `LedControllerRequest` as the only communication interface. The generated proxies and wrappers for this interface are in `LedControllerRequest.h` which is included, along with C++ implementations of all additional interface types in `GeneratedTypes.h`. Line 9 instantiates the proxy through which the software invokes the hardware methods (See also *Flow Control*)

To support projects that will execute software inside the linux kernel (for example, to work in conjunction with filesystems or the network stack), connectal project software can also be written in C.

Hardware

Connectal projects typically have at least one BSV file containing interface declarations and module definitions. The implementation of the interfaces and all supporting infrastructure is standard BSV. Interfaces being used as portals are subject to the type restrictions described earlier (See also *Interface Declarations*)

Top.bsv

In Top.bsv (https://github.com/connectal-examples/leds/blob/master/Top.bsv), the developer instantiates all hardware modules explicitly. Interfaces which can be invoked through portals need to be connected to the generated wrappers and proxies. To connect to the host processor bus, a parameterized standard interface is used, making it easy to synthesize the application for different CPUs or for simulation:

```
// Connectal Libraries
import CtrlMux::*;
import Portal::*;
import Leds::*;
import MemTypes::*;
import MemPortal::*;
import HostInterface::*;
import LedControllerRequest::*;
import LedController::*;
```

```
typedef enum {LedControllerRequestPortal} IfcNames deriving (Eq,Bits);

module mkConnectalTop(StdConnectalTop#(PhysAddrWidth));
   LedController ledController <- mkLedControllerRequest();
   LedControllerRequestWrapper ledControllerRequestWrapper <-
      mkLedControllerRequestWrapper(LedControllerRequestPortal,
      ledController.request);

   Vector#(1,StdPortal) portals;
   portals[0] = ledControllerRequestWrapper.portalIfc;
   let ctrl_mux <- mkSlaveMux(portals);

   interface interrupt = getInterruptVector(portals);
   interface slave = ctrl_mux;
   interface masters = nil;
   interface leds = ledController.leds;
endmodule
```

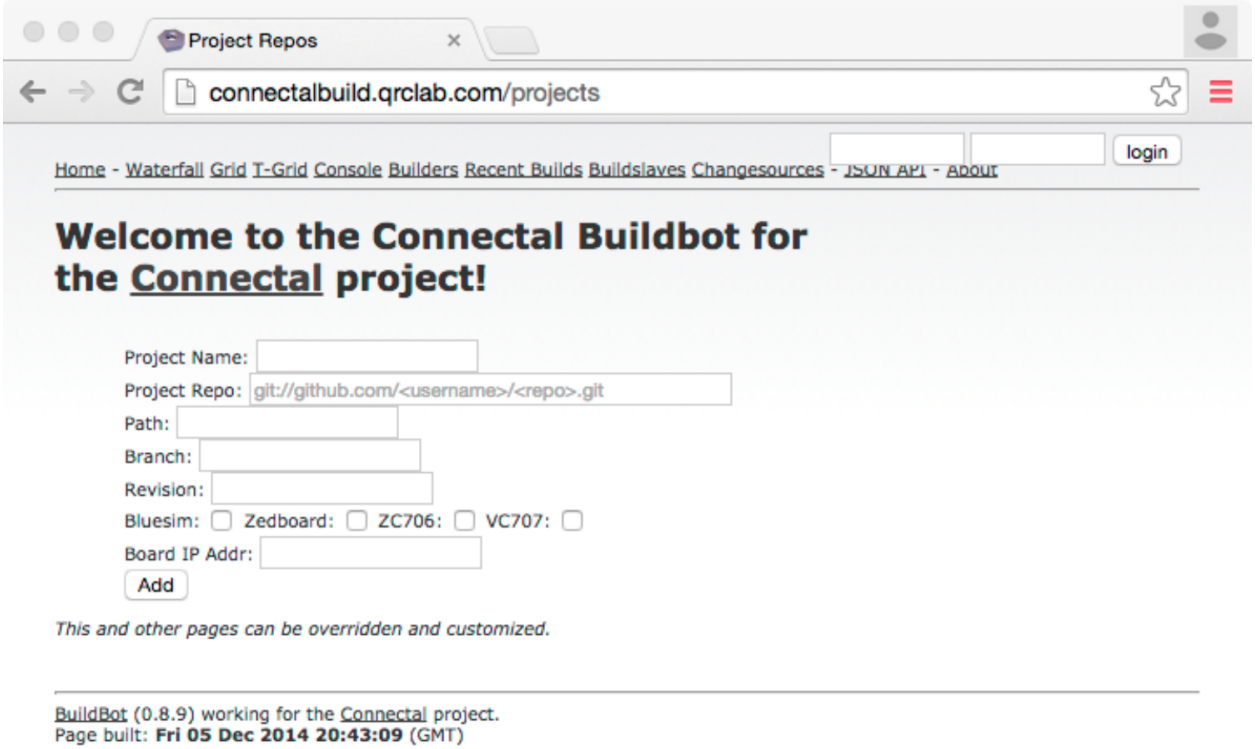
Like the SW components, the HW begins by importing the generated wrappers and proxies corresponding to the interfaces listed in the project Makefile. The user-defined implementation of the LedControllerRequest interface is instantiated on line 14, and wrapped on line 15. This wrapped interface is connected to the bus using the library module `mkSlaveMux` on line 21 so it can be invoked from the software. At the end of the module definition, the top-level interface elements must be connected. A board-specific top-level module will include this file, instantiate `mkConnectalTop` and connect the interfaces to the actual peripherals. The module `mkConnectalTop` must be defined in a file named `Top.bsv` in the user project.

The Bluespec compiler generates a Verilog module from the top level BSV module, in which the methods of exposed interfaces are implemented as Verilog ports. Those ports are bound to physical pins on the FPGA using a physical constraints file. If CPU specific interface signals are needed by the design (for example, extra clocks that are generated by the PCIe core), then an optional CPU-specific HostInterface parameter to `mkConnectalTop` can also be used. If the design uses external pins on the FPGA, those connections are also made here by exporting a 'Pins' interface (hyperref[host_interface]{Section~ref{host_interface}}) and providing bindings in the constraints file.

1.3.3 Compiling and Running Connectal Project

Compiling on ConnectalBuild

The Connectal toolchain can be run on ConnectalBuild using the following Buildbot web interface: http://connectalbuild.qrclab.com/projects.



Before submitting a project, you must sign in using your github credentials. We do not store credentials, but pass them through to github. Next, enter a name for the project, which will be used for subsequent build requests through the Buildbot web interface. The project must be in a publicly accessible git-hub repository, whose Repo location is entered beginning with git:// as follows git://github.com/connectal-examples/leds.git. If the project makefile is not in the root directory of the repository, enter its relative path in the 'Path' field of the form. If a particular branch or revision number are desired, enter these as well. Check the button to select the build target. If you have selected a zynq-based platform and would like the tool-chain to automatically program the device and execute the design as its final step, then enter the IP address of your board. This works only because `adb` doesn't require authentication. SSH keys required to run on PCIe-based platforms are not currently supported. Finally, don't forget to click 'Add'. If the project name has already been used, you will be prompted to enter a new one at this point.

Compiling Locally

Before compiling a project locally, you will need to install the toolchain. After setting the `CONNECTALDIR` to the root of the connectal source tree, enter the command `make`

Running the Design

1.4 Connectal BSV Libraries

1.4.1 Address Generator

One of the common patterns that leads to long critical paths in designs on the FPGA are counters and comparisons against counters. This package contains a module for generating the sequence of addresses used by a memory read or write burst, along with a field indicating the last beat of the burst.

struct AddressGenerator::**AddrBeat#(numeric type addrWidth)**

addr → Bit#(addrWidth)
The address for this beat of the request.

bc → Bit#(BurstLenSize)

tag → Bit#(MemTagSize)

last → Bool

interface AddressGenerator::**AddressGenerator**(*numeric type addrWidth*, *numeric type dataWidth*)

request → Put#(PhysMemRequest#(addrWidth))
The interface for requesting a sequence of addresses.

addrBeat → Get#(AddrBeat#(addrWidth))
The interface for getting the address beats of the burst. There is one pipeline cycle from the reuqest to the first address beat.

module AddressGenerator::**mkAddressGenerator** → (AddressGenerator#(addrWidth, dataWidth)
Instantiates an address generator.

1.4.2 Arith Package

The Arith package implements some functions that correspond to infix operators.

function Arith::**booland**(*Bool x1*, *Bool x2*) → Bool
Returns logical "and" of inputs. Named to avoid conflict with the Verilog keyword "and".

function Arith::**boolor**(*Bool x1*, *Bool x2*) → Bool
Returns logical "or" of inputs. Named to avoid conflict with the Verilog keyword "or".

function Arith::**eq**(*a x1*, *a x2)*) → Bool

function Arith::**add**(*a x1*, *a x2*) → a
Returns sum of inputs. Requires Arith#(a).

function Arith::**mul**(*a x1*, *a x2*) → a
Returns product of inputs. Requires Arith#(a).

function Arith::**rshift**(*Bit#(b) x1*, *Integer i*) → Bit#(b)
Returns input right shifted by i bits.

function Arith::**a) vadd**(*Vector#(n, a) x1*, *Vector#(n, a) x2*) → Vector#(n,
Returns sum of input vectors.

function Arith::**a) vmul**(*Vector#(n, a) x1*, *Vector#(n, a) x2*) → Vector#(n,
Returns element-wise product of input vectors.

function Arith::**Bit#**(*b)) vrshift(Vector#(n, Bit#(b)) x1*, *Integer i*) → Vector#(n,
Right shifts the elements of the input vector by i bits.

1.4.3 CtrlMux Package

module CtrlMux::**mkInterruptMux**(*Vector#(numPortals, MemPortal#(aw, dataWidth)) portals*) → (ReadOnly#(Bool)
Used by BsimTop, PcieTop, and ZynqTop. Takes a vector of MemPortals and returns a boolean indicating whether any of the portals has indication method data available.

module `CtrlMux::`**`mkSlaveMux`**(*Vector#(numPortals, MemPortal#(aw, dataWidth)) portals*) → (PhysMemSlave#(addrWidth,dataWidth)
Takes a vector of MemPortals and returns a PhysMemSlave combining them.

1.4.4 HostInterface Package

The HostInterface package provides host-specific typedefs and interfaces.

Host-Specific Constants

typedef `HostInterface::`**`DataBusWidth`**
Width in bits of the data bus connected to host shared memory.

typedef `HostInterface::`**`PhysAddrWidth`**
Width in bits of physical addresses on the data bus connected to host shared memory.

typedef `HostInterface::`**`NumberOfMasters`**
Number of memory interfaces used for connecting to host shared memory.

Host-Specific Interfaces

interface `HostInterface::`**`BsimHost`**
Host interface for the bluesim platform

interface `HostInterface::`**`PcieHost`**
Host interface for PCIe-attached FPGAs such as vc707 and kc705

interface `HostInterface::`**`ZynqHost`**
Host interface for Zynq FPGAs such as zedboard, zc702, zc706, and zybo.

The Zc706 is a ZynqHost even when it is plugged into a PCIe slot.

1.4.5 Leds Package

interface `Leds::`**`LEDS`**

typedef `Leds::`**`LedsWidth`**
Defined to be the number of default LEDs on the FPGA board.

The Zedboard has 8, Zc706 has 4, ...

`Leds::`**`leds`** → Bit#(LedsWidth)

1.4.6 MemPortal Package

mkMemPortal Module

module `MemPortal::`**`mkMemPortal`**(*Bit#(slaveDataWidth) ifcId, PipePortal#(numRequests, numIndications, slaveDataWidth) portal*) → (MemPortal#(slaveAddrWidth, slaveDataWidth)
Takes an interface identifier and a PipePortal and returns a MemPortal.

1.4.7 MemreadEngine Package

module MemreadEngine::**mkMemreadEngine**(**MemreadEngineV**(*dataWidth*, *cmdQDepth*, *numServers*)

Creates a MemreadEngine with default 256 bytes of buffer per server.

module MemreadEngine::**mkMemreadEngineBuff**(*Integer bufferSizeBytes*) → (MemreadEngineV#(dataWidth, cmdQDepth, numServers)

Creates a MemreadEngine with the specified buffer size.

1.4.8 MemTypes Package

Constants

typedef MemTypes::**Bit#(32) SGLId**

typedef MemTypes::**44 MemOffsetSize**

typedef MemTypes::**6 MemTagSize**

typedef MemTypes::**8 BurstLenSize**

typedef MemTypes::**32 MemServerTags**

Data Types

struct MemTypes::**PhysMemRequest#(numeric type addrWidth)**

A memory request containing a physical memory address

addr → Bit#(addrWidth)

Physical address to read or write

burstLen → Bit#(BurstLenSize)

Length of read or write burst, in bytes. The number of beats of the request will be the burst length divided by the physical width of the memory interface.

tag → Bit#(MemTagSize)

struct MemTypes::**MemRequest**

A logical memory read or write request. The linear offset of the request will be translated by an MMU according to the specified scatter-gather list.

sglId → SGLId

Indicates which scatter-gather list the MMU should use when translating the address

offset → Bit#(MemOffsetSize)

Linear byte offset to read or write.

burstLen → Bit#(BurstLenSize)

Length of read or write burst, in bytes. The number of beats of the request will be the burst length divided by the physical width of the memory interface.

tag → Bit#(MemTagSize)

struct MemTypes::**MemData#(numeric type dsz)**

One beat of the payload of a physical or logical memory read or write request.

data → Bit#(dsz)

One data beat worth of data.

tag → Bit#(MemTagSize)
Indicates to which request this beat belongs.

last → Bool
Indicates that this is the last beat of a burst.

Physical Memory Clients and Servers

interface `MemTypes::`**`PhysMemSlave`** (*numeric type addrWidth*, *numeric type dataWidth*)

read_server → PhysMemReadServer#(addrWidth, dataWidth)

write_server → PhysMemWriteServer#(addrWidth, dataWidth)

interface `MemTypes::`**`PhysMemMaster`** (*numeric type addrWidth*, *numeric type dataWidth*)

read_client → PhysMemReadClient#(addrWidth, dataWidth)

write_client → PhysMemWriteClient#(addrWidth, dataWidth)

interface `MemTypes::`**`PhysMemReadClient`** (*numeric type asz*, *numeric type dsz*)

readReq → Get#(PhysMemRequest#(asz))

readData → Put#(MemData#(dsz))

interface `MemTypes::`**`PhysMemWriteClient`** (*numeric type asz*, *numeric type dsz*)

writeReq → Get#(PhysMemRequest#(asz))

writeData → Get#(MemData#(dsz))

writeDone → Put#(Bit#(MemTagSize))

interface `MemTypes::`**`PhysMemReadServer`** (*numeric type asz*, *numeric type dsz*)

readReq → Put#(PhysMemRequest#(asz))

readData → Get#(MemData#(dsz))

interface `MemTypes::`**`PhysMemWriteServer`** (*numeric type asz*, *numeric type dsz*)

writeReq → Put#(PhysMemRequest#(asz))

writeData → Put#(MemData#(dsz))

writeDone → Get#(Bit#(MemTagSize))

Memory Clients and Servers

interface `MemTypes::`**`MemReadClient`** (*numeric type dsz*)

readReq → Get#(MemRequest)

readData → Put#(MemData#(dsz))

interface `MemTypes::MemWriteClient` (*numeric type dsz*)

`writeReq` → Get#(MemRequest)

`writeData` → Get#(MemData#(dsz))

`writeDone` → Put#(Bit#(MemTagSize))

interface `MemTypes::MemReadServer` (*numeric type dsz*)

`readReq` → Put#(MemRequest)

`readData` → Get#(MemData#(dsz))

interface `MemTypes::MemWriteServer` (*numeric type dsz*)

`writeReq` → Put#(MemRequest)

`writeData` → Put#(MemData#(dsz))

`writeDone` → Get#(Bit#(MemTagSize))

Memory Engine Types

struct `MemTypes::MemengineCmd`

A read or write request for a MemreadEngine or a MemwriteEngine. Memread and Memwrite engines will issue one or more burst requests to satisfy the overall length of the request.

`sglId` → SGLId
Which scatter gather list the MMU should use to translate the addresses

`base` → Bit#(MemOffsetSize)
Logical base address of the request, as a byte offset

`burstLen` → Bit#(BurstLenSize)
Maximum burst length, in bytes.

`len` → Bit#(32)
Number of bytes to transfer. Must be a multiple of the data bus width.

`tag` → Bit#(MemTagSize)
Identifier for this request.

Memory Engine Interfaces

interface `MemTypes::MemwriteServer` (*numeric type dataWidth*)

`cmdServer` → Server#(MemengineCmd,Bool)

`dataPipe` → PipeIn#(Bit#(dataWidth))

interface `MemTypes::MemwriteEngineV` (*numeric type dataWidth*, *numeric type cmdQDepth*, *numeric type numServers*)

`dmaClient` → MemWriteClient#(dataWidth)

`writeServers` → Vector#(numServers, Server#(MemengineCmd,Bool))

`dataPipes` → Vector#(numServers, PipeIn#(Bit#(dataWidth)))

write_servers → Vector#(numServers, MemwriteServer#(dataWidth))

typedef MemTypes::**MemwriteEngineV#(dataWidth,cmdQDepth,1) MemwriteEngine#(numeric type dataWi**

interface MemTypes::**MemreadServer** (*numeric type dataWidth*)

cmdServer → Server#(MemengineCmd,Bool)

dataPipe → PipeOut#(Bit#(dataWidth))

interface MemTypes::**MemreadEngineV** (*numeric type dataWidth*, *numeric type cmdQDepth*, *numeric type numServers*)

dmaClient → MemReadClient#(dataWidth)

readServers → Vector#(numServers, Server#(MemengineCmd,Bool))

dataPipes → Vector#(numServers, PipeOut#(Bit#(dataWidth)))

read_servers → Vector#(numServers, MemreadServer#(dataWidth))

typedef MemTypes::**MemreadEngineV#(dataWidth,cmdQDepth,1) MemreadEngine#(numeric type dataWidt**

Memory Traffic Interfaces

interface MemTypes::**DmaDbg**

getMemoryTraffic → ActionValue#(Bit#(64))

dbg → ActionValue#(DmaDbgRec)

Connectable Instances

instance MemTypes::**Connectable** (*MemReadClient#(dsz)*, *MemReadServer#(dsz)*)

instance MemTypes::**Connectable** (*MemWriteClient#(dsz)*, *MemWriteServer#(dsz)*)

instance MemTypes::**Connectable** (*PhysMemMaster#(addrWidth*, *busWidth)*, *PhysMemSlave#(addrWidth*, *busWidth)*)

instance MemTypes::**Connectable** (*PhysMemMaster#(32*, *busWidth)*, *PhysMemSlave#(40*, *busWidth)*)

1.4.9 MMU Package

typedef MMU::**32 MaxNumSGLists**

typedef MMU::**Bit#(TLog#(MaxNumSGLists)) SGListId**

typedef MMU::**12 SGListPageShift0**

typedef MMU::**16 SGListPageShift4**

typedef MMU::**20 SGListPageShift8**

typedef MMU::**Bit#(TLog#(MaxNumSGLists)) RegionsIdx**

typedef MMU::**8 IndexWidth**

Address Translation

struct MMU::**ReqTup**
Address translation request type

id → SGListId
Which SGList to use.

off → Bit#(MemOffsetSize)
The address to translate.

interface MMU::**MMU** (*numeric type addrWidth*)
An address translator

request → MMURequest
The interface of the MMU that is exposed to software as a portal.

addr → Vector#(2,Server#(ReqTup,Bit#(addrWidth)))
The address translation servers

module MMU::**mkMMU** (*Integer iid*, *Bool bsimMMap*, *MMUIndication mmuIndication*) → (MMU#(addrWidth)
Instantiates an address translator that stores a scatter-gather list to define the logical to physical address mapping.

Parameter iid is the portal identifier of the MMURequest interface.

Parameter bsimMMAP ??

Multiple Address Translators

interface MMU::**MMUAddrServer** (*numeric type addrWidth*, *numeric type numServers*)
Used by mkMemServer to share an MMU among multiple memory interfaces.

interface Vector (*(numServers*, *Server#(ReqTup*, *Bit#(addrWidth))) servers*)
The vector of address translators.

module MMU::**mkMMUAddrServer** (*Server#(ReqTup*, *Bit#(addrWidth))* *server*) → (MMUAddrServer#(addrWidth,numServers)
Instantiates an MMUAddrServer that shares the input server among multiple clients.

1.4.10 Pipe Package

The Pipe package is modeled on Bluespec, Inc's PAClib package. It provides functions and modules for composing pipelines of operations.

Pipe Interfaces

interface Pipe::**PipeIn** (*type a*)
Corresponds to the input interface of a FIFOF.

enq (*a v*) → Action

notFull → Bool

interface Pipe::**PipeOut** (*type a*)
Corresponds to the output interface of a FIFOF.

first → a

deq → Action

notEmpty → Bool

typeclass Pipe::**ToPipeIn** (*type a*, *type b*)

function toPipeIn (*b in*) → PipeIn#(a)
Returns a PipeIn to the object "in" with no additional buffering.

typeclass Pipe::**ToPipeOut** (*type a*, *type b*)

function toPipeOut (*b in*) → PipeOut#(a)
Returns a PipeOut from the object "in" with no additional buffering.

typeclass Pipe::**MkPipeIn** (*type a*, *type b*)

module mkPipeIn (*b in*) → (PipeIn#(a)
Instantiates a module whose interface is a PipeIn to the input parameter "in". Includes a FIFO buffering stage.

typeclass Pipe::**MkPipeOut** (*type a*, *type b*)

module mkPipeOut (*b in*) → (PipeOut#(a)
Instantiates a module whose interface is PipeOut from the input parameter "in". Includes a FIFO buffering stage.

instance Pipe::**ToPipeIn** (*a*, *FIFOF#(a)*)
Converts a FIFOF to a PipeIn.

instance Pipe::**ToPipeOut** (*a*, *function a pipefn()*)
Converts a function to a PipeOut.

instance Pipe::**ToPipeOut** (*a*, *Reg#(a)*)
Converts a register to a PipeOut.

instance Pipe::**ToPipeIn** (*Vector#(m, a)*, *Gearbox#(m, n, a)*)
Converts a Gearbox to a PipeOut.

instance Pipe::**ToPipeOut** (*a*, *FIFOF#(a)*)
Converts a FIFOF to a PipeOut.

instance Pipe::**ToPipeOut** (*Vector#(n, a)*, *MIMO#(k, n, sz, a)*)
Converts a MIMO to a PipeOut.

instance Pipe::**ToPipeOut** (*Vector#(n, a)*, *Gearbox#(m, n, a)*)
Converts a Gearbox to a PipeOut.

instance Pipe::**MkPipeOut** (*a*, *Get#(a)*)
Instantiates a pipelined PipeOut from a Get interface.

instance Pipe::**MkPipeIn** (*a*, *Put#(a)*)
Instantiates a pipelined PipeIn to a Put interface.

Get and Put Pipes

instance Pipe::**ToGet** (*PipeOut #(a)*, *a*)

instance Pipe::**ToPut** (*PipeIn #(a)*, *a*)

Connectable Pipes

instance `Pipe::Connectable` (*PipeOut#(a)*, *Put#(a)*)

instance `Pipe::Connectable` (*PipeOut#(a)*, *PipeIn#(a)*)

Mapping over Pipes

function `Pipe::toCountedPipeOut` (*Reg#(Bit#(n)) r*, *PipeOut#(a) pipe*) → PipeOut#(a)

function `Pipe::zipPipeOut` (*PipeOut#(a) ina*, *PipeOut#(b) inb*) → PipeOut#(Tuple2#(a,b))
Returns a PipeOut whose elements are 2-tuples of the elements of the input pipes.

function `Pipe::mapPipe` (*function b f(a av)*, *PipeOut#(a) apipe*) → PipeOut#(b)
Returns a PipeOut that maps the function f to each element of the input pipes with no buffering.

module `Pipe::mkMapPipe` (*function b f(a av)*, *PipeOut#(a) apipe*) → (PipeOut#(b)
Instantiates a PipeOut that maps the function f to each element of the input pipes using a FIFOF for buffering.

function `Pipe::mapPipeIn` (*function b f(a av)*, *PipeIn#(b) apipe*) → PipeIn#(a)
Returns a PipeIn applies the function f to each value that is enqueued.

Reducing Pipes

Functions on Pipes of Vectors

function `Pipe::unvectorPipeOut` (*PipeOut#(Vector#(1, a)) in*) → PipeOut#(a)

Funneling and Unfunneling

module `Pipe::mkFunnel` (*PipeOut#(Vector#(mk, a)) in*) → (PipeOut#(Vector#(m, a))
Returns k Vectors of m elements for each Vector#(mk,a) element of the input pipe.

module `Pipe::mkFunnel1` (*PipeOut#(Vector#(k, a)) in*) → (PipeOut#(a)
Sames as mkFunnel, but returns k singleton elements for each vector element of the input pipe.

module `Pipe::mkFunnelGB1` (*Clock slowClock*, *Reset slowReset*, *Clock fastClock*, *Reset fastReset*, *PipeOut#(Vector#(k, a)) in*) → (PipeOut#(a)
Same as mkFunnel1, but uses a Gearbox with a 1 to k ratio.

module `Pipe::mkUnfunnel` (*PipeOut#(Vector#(m, a)) in*) → (PipeOut#(Vector#(mk, a))
The dual of mkFunnel. Consumes k elements from the input pipe, each of which is an m-element vector, and returns an mk-element vector.

module `Pipe::mkUnfunnelGB` (*Clock slowClock*, *Reset slowReset*, *Clock fastClock*, *Reset fastReset*, *PipeOut#(Vector#(1, a)) in*) → (PipeOut#(Vector#(k, a))
The same as mkUnfunnel, but uses a Gearbox with a 1-to-k.

module `Pipe::mkRepeat` (*UInt#(n) repetitions*, *PipeOut#(a) inpipe*) → (PipeOut#(a)
Returns a PipeOut which repeats each element of the input pipe the specified number of times.

Fork and Join

Fork and Join with limited scalability

module `Pipe::mkForkVector` (*PipeOut#(a) inpipe*) → (Vector#(n, PipeOut#(a))
Replicates each element of the input pipe to each of the output pipes. It uses a FIFOF per output pipe.

module `Pipe::mkSizedForkVector` (*Integer size*, *PipeOut#(a) inpipe*) → (Vector#(n, PipeOut#(a))
Used a SizedFIFOF for each of the output pipes.

module `Pipe::mkJoin` (*function c f(a av*, *b bv)*, *PipeOut#(a) apipe*, *PipeOut#(b) bpipe*) → (PipeOut#(c)
Returns a PipeOut that applies the function f to the elements of the input pipes, with no buffering.

module `Pipe::mkJoinBuffered` (*function c f(a av*, *b bv)*, *PipeOut#(a) apipe*, *PipeOut#(b) bpipe*) → (PipeOut#(c)
Returns a PipeOut that applies the function f to the elements of the input pipes, using a FIFOF to buffer the output.

module `Pipe::mkJoinVector` (*function b f(Vector#(n, a) av)*, *Vector#(n, PipeOut#(a)) apipes*) → (PipeOut#(b)
Same as mkJoin, but operates on a vector of PipeOut as input.

Funnel Pipes

Fork and Join with tree-based fanout and fanin for scalability.

These are used by MemreadEngine and MemwriteEngine.

typedef `Pipe::Vector#(j,PipeOut#(a))  FunnelPipe#(numeric type j, numeric type k, type a, nu`

typedef `Pipe::Vector#(k,PipeOut#(a)) UnFunnelPipe#(numeric type j, numeric type k, type a, nu`

typeclass `Pipe::FunnelPipesPipelined` (*numeric type j*, *numeric type k*, *type a*, *numeric type bpc*)

module `mkFunnelPipesPipelined` (*Vector#(k, PipeOut#(a)) in*) → (FunnelPipe#(j,k,a,bpc)

module `mkFunnelPipesPipelinedRR` (*Vector#(k, PipeOut#(a)) in*, *Integer c*) → (FunnelPipe#(j,k,a,bpc)

module `mkUnFunnelPipesPipelined` (*Vector#(j, PipeOut#(Tuple2#(Bit#(TLog#(k)), a))) in*) → (UnFunnelPipe#(j,k,a,bpc)

module `mkUnFunnelPipesPipelinedRR` (*Vector#(j, PipeOut#(a)) in*, *Integer c*) → (UnFunnelPipe#(j,k,a,bpc)

instance `Pipe::FunnelPipesPipelined` (*1*, *1*, *a*, *bpc*)

instance `Pipe::FunnelPipesPipelined` (*1*, *k*, *a*, *bpc*)

module `Pipe::mkUnFunnelPipesPipelinedInternal` (*Vector#(1, PipeOut#(Tuple2#(Bit#(TLog#(k)), a))) in*) → (UnFunnelPipe#(1,k,a,bpc)

module `Pipe::mkFunnelPipes` (*Vector#(mk, PipeOut#(a)) ins*) → (Vector#(m, PipeOut#(a))

module `Pipe::mkFunnelPipes1` (*Vector#(k, PipeOut#(a)) ins*) → (PipeOut#(a)

module `Pipe::mkUnfunnelPipes` (*Vector#(m, PipeOut#(a)) ins*) → (Vector#(mk, PipeOut#(a))

module `Pipe::mkPipelinedForkVector` (*PipeOut#(a) inpipe*, *Integer id*) → (UnFunnelPipe#(1,k,a,bpc)

Delimited Pipes

interface `Pipe::FirstLastPipe` (*type a*)
A pipe whose elements two-tuples of boolean values indicating first and last in a series. The ttype a indicates the type of the counter used.

pipe → PipeOut#(Tuple2#(Bool,Bool))
The pipe of delimited elements

start (*a count*) → Action
Starts the series of count elements

module `Pipe::`**mkFirstLastPipe** → (FirstLastPipe#(a)
Creates a FirstLastPipe.

struct `Pipe::`**RangeConfig#(type a)**
The base, limit and step for mkRangePipeOut.

xbase → a

xlimit → a

xstep → a

interface `Pipe::`**RangePipeIfc** (*type a*)

pipe → PipeOut#(a)

isFirst → Bool

isLast → Bool

start (*RangeConfig#(a) cfg*) → Action

module `Pipe::`**mkRangePipeOut** → (RangePipeIfc#(a)
Creates a Pipe of values from xbase to xlimit by xstep. Used by Memread.

1.4.11 Portal Package

PipePortal Interface

interface `Portal::`**PipePortal** (*numeric type numRequests*, *numeric type numIndications*, *numeric type slaveDataWidth*)

messageSize (*Bit#(16) methodNumber*) → Bit#(16)
Returns the message size of the methodNumber method of the portal.

requests → Vector#(numRequests, PipeIn#(Bit#(slaveDataWidth)))

indications → Vector#(numIndications, PipeOut#(Bit#(slaveDataWidth)))

MemPortal Interface

interface `Portal::`**MemPortal** (*numeric type slaveAddrWidth*, *numeric type slaveDataWidth*)

slave → PhysMemSlave#(slaveAddrWidth,slaveDataWidth)

interrupt → ReadOnly#(Bool)

top → WriteOnly#(Bool)

function `Portal::`**getSlave** (*MemPortal#(_a, _d) p*) → PhysMemSlave(_a,_d)

function `Portal::`**getInterrupt** (*MemPortal#(_a, _d) p*) → ReadOnly#(Bool)

function `Portal::`**getInterruptVector** (*Vector#(numPortals, MemPortal#(_a, _d)) portals*) → Vector#(16,ReadOnly#(Bool))

ShareMemoryPortal Interface

interface `Portal::`**`SharedMemoryPortal`**(*numeric type dataBusWidth*)
Should be in SharedMemoryPortal.bsv

`readClient` → MemReadClient(dataBusWidth)

`writeClient` → MemWriteClient#(dataBusWidth)

`cfg` → SharedMemoryPortalConfig

`interrupt` → ReadOnly#(Bool)

ConnectalTop Interface

interface `Portal::`**`ConnectalTop`**(*numeric type addrWidth*, *numeric type dataWidth*, *type pins*, *numeric type numMasters*)
Interface ConnectalTop is the interface exposed by the top module of a Connectal hardware design.

`slave` → PhysMemSlave#(32,32)

`masters` → Vector#(numMasters,PhysMemMaster#(addrWidth, dataWidth))

`interrupt` → Vector#(16,ReadOnly#(Bool))

`leds` → LEDS

`pins` → pins

StdConnectalTop Typedef

typedef `Portal::`**`StdConnectalTop`**(numeric type addrWidth) → ConnectalTop#(addrWidth,64,Empty,0)
Type StdConnectalTop indicates a Connectal hardware design with no user defined pins and no user of host shared memory. The "pins" interface is Empty and the number of masters is 0.

typedef `Portal::`**`StdConnectalDmaTop`**(numeric type addrWidth) → ConnectalTop#(addrWidth,64,Empty,1)
Type StdConnectalDmaTop indicates a Connectal hardware design with no user defined pins and a single client of host shared memory. The "pins" interface is Empty and the number of masters is 1.

1.5 Connectal C/C++ Libraries

1.5.1 C/C++ Portal

void *__x*)
Polls the registered portals and invokes their callback **`handlers.`**()

void **`portalExec_start`**()

void **`portalExec_poll`**()

int **`portalAlloc`**(args)

1.6 Connectal Examples

1.6.1 Simple Example

1.7 Indices and tables

- *genindex*
- *modindex*
- *search*

a

c

h

l

m

p

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

P

R

S

T

U

V

Z