connectal Documentation

Release 14.12.6

Jamey Hicks, Myron King, John Ankcorn

February 04, 2015

CHAPTER

ONE

CONTENTS

1.1 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.

1.1.1 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

1.1.2 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 Connectal Developer's Guide

1.2.1 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

1.2.2 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 using the Bluesim or Zynq target platforms.

Project Makefile

The top-level Makefile (https://github.com/connectal-examples/leds/blob/master/Makefile) defines parameters building and executing the project. 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:

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

`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.

`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.

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. 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 all the interfaces listed in INTERFACES. If an the definition of a listed interfaces is not found, an error is reported the 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*)

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_definitions*)

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
```

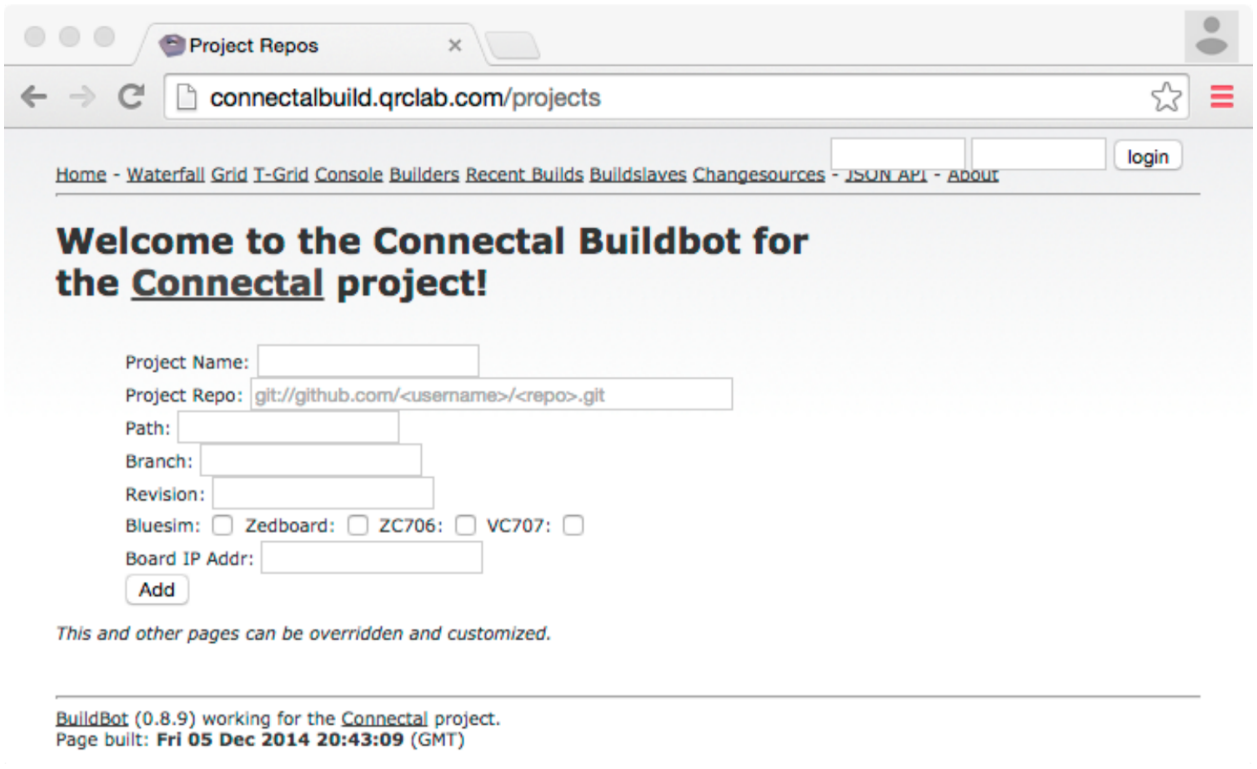
Like the SW components, the HW begins by importing the generated wrappers and proxies corresponding to the interfaces listed in the project Makefile. The user-defined implementation of the LedControllerRequest interface is instantiated on line 14, and wrapped on line 15. This wrapped interface is connected 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 name of the file must be `Top.bsv` and the name of the module must be `mkConnectalTop`.

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. 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 (hyperref[host_interface]{Section~ref{host_interface}}).

1.2.3 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.


Project Repos
connectalbuild.qrclab.com/projects
Home - Waterfall Grid T-Grid Console Builders Recent Builds Buildslaves Changesources - JSON API - About
login

Welcome to the Connectal Buildbot for the Connectal project!

Project Name:
Project Repo: git://github.com/<username>/<repo>.git
Path:
Branch:
Revision:
Bluesim: ☐ Zedboard: ☐ ZC706: ☐ VC707: ☐
Board IP Addr:
Add

This and other pages can be overridden and customized.

BuildBot (0.8.9) working for the Connectal project.
Page built: **Fri 05 Dec 2014 20:43:09** (GMT)

Before submitting a project, you must sign in using your github credentials. We do not store credentials, but pass them througth 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

1.3 Connectal BSV Libraries

1.3.1 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.

1.3.2 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.

1.3.3 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.

1.3.4 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.

1.3.5 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)

1.3.6 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.

1.3.7 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.

1.3.8 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)*)

1.3.9 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.

1.3.10 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.

1.3.11 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.

1.4 Connectal Examples

1.4.1 Simple Example

1.5 Indices and tables

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

BSV PACKAGE INDEX