Constellation

Jerry Zhao

Aug 16, 2023

CONTENTS:

1 Introduction **3**
- 1.1 Setting up Constellation . . . 3
  - 1.1.1 Installing Rocketchip . . . 4
  - 1.1.2 Installing Espresso . . . 4
  - 1.1.3 Installing Minimal-Standalone . . . 4
  - 1.1.4 Installing with Chipyard . . . 4
- 1.2 Running Constellation . . . 5
  - 1.2.1 Test Configs . . . 5
  - 1.2.2 Minimal-Standalone Testing . . . 5
  - 1.2.3 Chipyard-Standalone Testing . . . 6

2 NoC Configuration **7**
- 2.1 Physical Specification . . . 8
  - 2.1.1 Topology . . . 8
    - 2.1.1.1 Terminal Router Topologies . . . 9
    - 2.1.1.2 Hierarchical Topologies . . . 10
  - 2.1.2 Channels . . . 10
    - 2.1.2.1 Virtual Channels . . . 11
    - 2.1.2.2 Channel Generation . . . 11
    - 2.1.2.3 Speedup . . . 11
    - 2.1.2.4 Clock Crossings . . . 12
  - 2.1.3 Terminals . . . 12
  - 2.1.4 Routers . . . 13
    - 2.1.4.1 Pipelining . . . 13
    - 2.1.4.2 Payload Width . . . 14
    - 2.1.4.3 Virtual Channel Allocator . . . 14
- 2.2 Flow Specification . . . 14
  - 2.2.1 Flows . . . 15
  - 2.2.2 Virtual Subnetworks . . . 15
- 2.3 Routing Configuration . . . 15
  - 2.3.1 Channel Identifier . . . 16
  - 2.3.2 Flow Identifier . . . 17
  - 2.3.3 Base Routing Relations . . . 18
  - 2.3.4 Compositional Routing Relations . . . 18
    - 2.3.4.1 Escape Channel Routing . . . 18
    - 2.3.4.2 Terminal Router Routing . . . 19
    - 2.3.4.3 Hierarchical Topology Routing . . . 19
  - 2.3.5 Virtual Subnetwork Routing Relations . . . 20

3 Protocol Support **21**

3.1 Abstract Protocol Interface . . . 21
3.2 Standalone Protocol NoC . . . 22
3.2.1 TileLink Protocol Params . . . 22
3.2.2 AXI-4 Protocol Params . . . 23
3.3 Diplomatic Protocol NoC . . . 23

4 Chipyard SoC Integration **25**
4.1 Global Shared Interconnect . . . 25

5 Evaluation Framework **27**
5.1 Running a Simple Evaluation . . . 27

6 Indices and tables **29**

Constellation is a Chisel NoC RTL generator framework designed from the ground up to support integration in a heterogeneous SoC and evaluation of highly irregular NoC architectures.

- Constellation generates **packet-switched wormhole-routed networks with virtual networks and credit-based flow control**
- Constellation supports **arbitrary directed graph network topologies**, including **irregular** and **hierarchical** network topologies
- Constellation includes a **routing algorithm verifier and routing-table compiler**, which can verify and generate deadlock-free routing tables for arbitrary topologies
- Constellation is a **protocol-independent transport layer**, yet is capable of compliant deadlock-free transport of protocols like **AXI-4** and **TileLink**
- Constellation supports drop-in **integration in Chipyard/Rocketchip SoCs**
- Constellation is **rigorously tested**, with almost 100 different tests across as many network configurations

CHAPTER

ONE

INTRODUCTION

This section will walk you through the process of generating a simple NoC configuration, generating the RTL, and simulating the network.

1.1 Setting up Constellation

Constellation can be installed and used in three modes.

- **Minimal-Standalone** operation requires a minimal installation, and enables evaluation of NoCs within simple harnesses and evaluation frameworks.
- **Chiypard-Standalone** runs Constellation as a Chipyard "subproject", enabling more detailed evaluation of standalone NoCs.
- **Chipyard-SoC** generates a complete Chipyard SoC with a Constellation NoC

A table below summarizes the differences. For most users, **Chipyard-Standalone** and **Chipyard-SoC** are the most useful operation modes. **Minimal-Standalone** should only be used in environments prohibitive to a full Chipyard installation.

Mode	Requires	Capabilities
Minimal-Standalone	• Espresso • Rocketchip	• Standalone NoC generation
CY-Standalone	• Espresso • Chipyard	• Standalone NoC generation • Waveform generation • NoC visualization
CY-SoC	• Espresso • Chipyard	• NoC integration with SoC

1.1.1 Installing Rocketchip

Note: Rocketchip should only be manually installed for the **Minimal-standalone** operation mode.

For a **Minimal-Standalone** installation, Rocketchip must be manually installed as a locally-published Scala project.

```
git clone https://github.com/chipsalliance/rocket-chip.git
cd rocket-chip
git checkout 4fbd2f238db36b2862319e94c2f96d63bd52c98b
git submodule update --init --recursive
sbt "publishLocal"
sbt "project cde; set publishArtifact := true; publishLocal"
sbt "project rocket-macros; set publishArtifact := true; publishLocal"
sbt "project hardfloat; set publishArtifact := true; publishLocal"
```

1.1.2 Installing Espresso

Espresso is a logic minimizer tool. Constellation uses Espresso to generate efficient routing decode tables.

To use Constellation, `espresso` should be on your `PATH`.

```
git clone https://github.com/chipsalliance/espresso.git
cd espresso
mkdir -p build
cd build
cmake ../ -DBUILD_DOC=OFF -DCMAKE_INSTALL_PREFIX=/path/to/install/bin
make install
```

1.1.3 Installing Minimal-Standalone

To use Constellation in **Minimal-Standalone** mode, it is sufficient to clone the repository after installing Rocketchip and Espresso.

```
git clone https://github.com/ucb-bar/constellation.git
```

1.1.4 Installing with Chipyard

To use Constellation in **Chipyard-Standalone** or **Chipyard-SoC** mode, follow the instructions for installing Chipyard here. You must use Chipyard 1.8 or later.

After following those steps, run the following

```
make -C generators/constellation/src/main/resources/csrc/netrace netrace.o CFLAGS="-fPIC␣
↪-O3"
```

1.2 Running Constellation

This section will describe how to generate RTL for an example NoC configuration, and run a RTL simulation of the NoC under a simple test harness.

In this section, we will use pre-defined example configurationes

1.2.1 Test Configs

A large set of example configurations are described in src/main/scala/test/Configs.scala. These configs are representative of a wide set of Constellation's target design space.

These configs fall into four categories, delineated by the prefix of the Config name.

Prefix	ChiselTest Prefix	Generates...	Tests...
`TestConfig`	`NoCTest`	• NoC verilog • chisel random test harness	Functional correctness
`TLTestConfig`	`NoCTestTL`	• NoC verilog • TileLink protocol adapters • TileLink transport harness	Functional correctness for TL transport
`AXI4TestConfig`	`NoCTestAXI4`	• NoC verilog • AXI4 protocol adapters • AXI4 transport harness	Functional correctness for AXI4 transport

1.2.2 Minimal-Standalone Testing

In minimal-standalone mode, predefined testing configurations can be generated and simulated. The list of testing configurations is described in src/test/scala/constellation/NocTests.scala. Each Test configuration corresponds to a NoC configuration.

```
cd constellation
CONSTELLATION_STANDALONE=1 sbt "testOnly constellation.NoCTest00"
```

When running the above command, the NoC verilog will be generated in `test_run_dir`.

```
cd test_run_dir
cd NoC_should_pass_test_with_config_constellationtestTestConfig00
cat NoCChiselTester.sv
```

1.2.3 Chipyard-Standalone Testing

The test configurations can also be run in Chipyard-Standalone mode.

```
cd chipyard/sims/vcs
make SUB_PROJECT=constellation BINARY=none CONFIG=TestConfig00 run-binary-debug
```

After running the above command, the generated verilog will be in `generated-src`.

```
cd generated-src
cd constellation.test.TestHarness.TestConfig00
cat constellation.test.TestHarness.TestConfig00.top.v
```

A visualization of the NoC can also be generated

```
NOC_PATH=$(pwd)/constellation.test.TestHarness.TestConfig00.test.noc.
cd ~/chipyard/generators/constellation/scripts
./vis.py $NOC_PATH
```

CHAPTER

TWO

NOC CONFIGURATION

Constellation divides the NoC specification into three orthogonal concerns.

- **Physical specification** describes the topology and microarchitecture of the NoC
- **Flow specification** describes what flows the network might expect
- **Routing specification** describes how flows traverse the physical resources of the NoC

The total specification of the NoC is captured in the *constellation.noc.NoCParams* case class. The total class is depicted below. Please see the subsections for more details.

```
case class NoCParams(
  // Physical specifications
  topology: PhysicalTopology = UnidirectionalLine(1),
  channelParamGen: (Int, Int) => UserChannelParams = (_, _) => UserChannelParams(),
  ingresses: Seq[UserIngressParams] = Nil,
  egresses: Seq[UserEgressParams] = Nil,
  routerParams: Int => UserRouterParams = (i: Int) => UserRouterParams(),

  // Flow specification
  // (blocker, blockee) => bool
  // If true, then blocker must be able to proceed when blockee is blocked
  vNetBlocking: (Int, Int) => Boolean = (_, _) => true,
  flows: Seq[FlowParams] = Nil,

  // Routing specification
  routingRelation: PhysicalTopology => RoutingRelation = AllLegalRouting(),

  // other
  nocName: String = "test",
  skipValidationChecks: Boolean = false,
  hasCtrl: Boolean = false,
)
```

A specification is constructed as an instance of a `NoCParams` case class. Examples of specifications can be seen in src/main/scala/test/Configs.scala.

2.1 Physical Specification

The NoC's physical specifcation is itself broken down into five components.

```
  topology: PhysicalTopology = UnidirectionalLine(1),
  channelParamGen: (Int, Int) => UserChannelParams = (_, _) => UserChannelParams(),
  ingresses: Seq[UserIngressParams] = Nil,
  egresses: Seq[UserEgressParams] = Nil,
  routerParams: Int => UserRouterParams = (i: Int) => UserRouterParams(),
```

2.1.1 Topology

A `PhysicalTopology` is a case class which describes a directed graph, where nodes represent routers, and edges represent unidirectional channels.

```
trait PhysicalTopology {
  // Number of nodes in this physical topology
  val nNodes: Int

  /** Method that describes the particular topology represented by the concrete
    * class. Returns true if the two nodes SRC and DST can be connected via a
    * directed channel in this topology and false if they cannot.
    *
    *  @param src source point
    *  @param dst destination point
    */
  def topo(src: Int, dst: Int): Boolean

  /** Plotter from TopologyPlotters.scala.
    * Helps construct diagram of a concrete topology. */
  val plotter: PhysicalTopologyPlotter
}
```

To see how to extend his trait, consider the `UnidirectionalTorus1D` topology.

```
/** An n-node network shaped like a torus, with directed channels */
case class UnidirectionalTorus1D(n: Int) extends Torus1DLikeTopology {
  val nNodes = n
  def topo(src: Int, dest: Int) = (dest - src + nNodes) % nNodes == 1
}
```

The current list of included base topologies is described below. User can always define their own topology.

Topology	Parameters	Diagram
`UnidirectionalLine`	`nNodes`: Number of nodes	
`BidirectionalLine`	`nNodes`: Number of nodes	
`UnidirectionalTorus1D`	`nNodes`: Number of nodes	
`BidirectionalTorus1D`	`nNodes`: Number of nodes	
`Butterfly`	`kAry`: Degree of routers `nFly`: Number of stages	
`BidirectionalTree`	`height`: Height of tree `dAry` : Degree of routers	
`Mesh2D`	`nX`: Extent in X-dimension `nY`: Extent in Y-dimension	
`UnidirectionalTorus2D`	`nX`: Extent in X-dimension `nY`: Extent in Y-dimension	
`BidirectionalTorus2D`	`nX`: Extent in X-dimension `nY`: Extent in Y-dimension	

Additionally, two hierarchical compositional topology generators are provided: `TerminalRouter` and `HierarchicalTopology`.

2.1.1.1 Terminal Router Topologies

The `TerminalRouter` topology class "wraps" a base topology with a layer of terminal router nodes, which the ingresses and egresses tie to. This reduces the radix of critical routers in the network. This topology class can wrap an arbitrary base topology (including custom topologies).

For example, consider a network where the base topology is a bidirectional line. Wrapping the base `BidirectionalLine` topology class in the `TerminalRouter` class "lifts" the terminal points into separate nodes (purple), while preserving the existing topology and routing behavior on the underlying network routers.

Topology	Diagram
`BidirectionalLine(4)`	
`TerminalRouter(BidirectionalLine(4))`	

Note: The `TerminalRouter` topology must be used with the `TerminalRouting` routing relation wrapper

2.1.1.2 Hierarchical Topologies

The `HierarchicalTopology` class joins a collection of child sub-topologies using a base topology. A `HierarchicalSubTopology` describes a child sub-topology, as well as the connection to the base topology.

In the first example below, note how the first hierarchical child topology describes a channel between the 0th node on the base topology, and what would be node 2 on the child topology. Here the green nodes are from the base topology, while the blue nodes are from the child topologies.

Note: The `TerminalRouter` can be combined with `HierarchicalTopology`, as shown in the second example

Topology	Diagram

`HierarchicalTopology(`
`  base=UnidirectionalTorus1D(4),`
`  children=Seq(`
`  HierarchicalSubTopology(0,2,`
`↪BidirectionalLine(5)),`
`  HierarchicalSubTopology(2,1,`
`↪BidirectionalLine(3))`
`  )`
`)`	

`TerminalRouter(HierarchicalTopology(`
`  base=UnidirectionalTorus1D(4),`
`  children=Seq(`
`  HierarchicalSubTopology(0,2,`
`↪BidirectionalLine(5)),`
`  HierarchicalSubTopology(2,1,`
`↪BidirectionalLine(3))`
`  )`
`))`	

Note: The `HierarchicalTopology` topology must be used with the `HierarchicalRouting` routing relation wrapper

2.1.2 Channels

`channelParamGen` is a function which determines the channel parameters for each directed channel in the network. For every edge dictated by the `PhysicalTopology`, this function is called to determine the channel parameters for that edge.

This function can return a different set of parameters for each edge.

```
case class UserChannelParams(
  virtualChannelParams: Seq[UserVirtualChannelParams] =
    Seq(UserVirtualChannelParams()),
  channelGen: Parameters => ChannelOutwardNode => ChannelOutwardNode =
```

(continues on next page)

(continued from previous page)

```
    p => u => u,
  crossingType: ClockCrossingType = NoCrossing,
  useOutputQueues: Boolean = true,
  srcSpeedup: Int = 1,
  destSpeedup: Int = 1
) {
  val nVirtualChannels = virtualChannelParams.size
}

case class UserVirtualChannelParams(
  bufferSize: Int = 1
)
```

2.1.2.1 Virtual Channels

The `virtualChannelParams` field contains a list of `UserVirtualChannelParams` objects, where the each element represents one virtual channel, and the `UserVirtualChannelParams` object holds the number of buffer entries for that virtual channel.

2.1.2.2 Channel Generation

Currently, the `channelGen` field is only used to specify adding additional pipeline buffers in a channel.

For example, the following segment sets a two-deep buffer on the channel.

```
channelGen = (u) => {
  implicit val p: Parameters = u
  ChannelBuffer(2) := _
}
```

Note: In the future more functionality can be added through this interface.

2.1.2.3 Speedup

`srcSpeedup` indicates the number of flits that may **enter** a channel in a cycle. For `srcSpeedup` > 1, the generator will effectively increase the input bandwidth of the channel.

`destSpeedup` indicates the number of flits that may **exit** a channel in a cycle. Increasing this pressures the routing resources and switch of the destination router.

Note: Setting `srcSpeedup` > `destSpeedup` is an unusual design point.

2.1.2.4 Clock Crossings

Currently unsupported. One day this will allow auto insertion of clock crossings between routers of the network on different clock domains.

2.1.3 Terminals

Constellation decouples the parameterization of ingress and egress channels from the base routing topology to allow for more flexibility in the described networks. Any router in the network can support many or no ingress and egress channels. The `UserIngressParams` and `UserEgressParams` case classes specify where the ingress and egress terminals are.

```
ingresses=Seq(UserIngressParams(0),
              UserIngressParams(1),
              UserIngressParams(1)),
egresses=Seq(UserEgressParams(0),
             UserEgressParams(3),
             UserEgressParams(3))
```

Explicit payload widths should also be specified. Usually, the payload width specified here would match the payload width of the router the terminal connects to.

If the terminal payload width is a multiple or factor of the router payload width, Constellation will auto-generate width converters to either further segment or merge flits.

```
ingresses = Seq(UserIngressParams(0),␣
↪payloadBits=128)),
egresses  = Seq( UserEgressParams(0),␣
↪payloadBits=128)),
routers   = (i) =>␣
↪UserRouterParams(payloadBits=64),
```

Note: The common use case for `payloadWidth` is to set the same width for all terminals.

Warning: Be wary of using payload width converters liberally. For example, a 3-flit packet of 64 bits per flit, if up-scaled to a 2-flit packet of 128 bits per flit, will be down-scaled into a 4-flit packet of 64 bits per flit.

2.1.4 Routers

Constellation's router generators have several microarchitectural configuration knobs. The standard router microarchitecture follows the standard design pattern. A brief description of the operation of a router follows:

1. The head flit of the packet queries the router's `RouteComputer` to determine the next set candidate virtual channels it may allocate. This is the **RC** stage.
2. The head flit of the packet queries the router's `VirtualChannelAllocator` to allocate a virtual channel from the candidate set of next virtual channels, in the **VA** stage
3. Flits ask the `SwitchAllocator` for access to the crossbar switch in the router in the **SA** stage. This stage also checks that the next virtual channel has an empty buffer slot to accomodate this flit.
4. A flit traverses the crossbar switch in the **ST** stage.

Flow-control is credit based. Matching `InputUnit` and `OutputUnit` pairs exchange credits over a separate narrow channel to indicate the availability of buffer entries. A flit departing a `InputUnit` sends a credit backwards to the `OutputUnit` on its source router.

Note: Constellation treats terminal Ingress and Egress points as special instances of `InputUnits` and `OutputUnits`.

Fig. 1: The standard router micro-architecture

The `router` field of the `NoCParams` case class returns per-router parameters, enabling heterogeneous designs with different router configurations.

2.1.4.1 Pipelining

The standard pipeline microarchitecture is a 4-hop router, with RC, VA, SA, and ST on separate stages. Two flags exist in the base router generator to reduce the hop count.

- `combineRCVA` performs route-compute and virtual-channel-allocation in the same cycle. Routers which implement near-trivial routing policies may benefit from this setting.
- `combineSAST` performs switch allocation and switch traversal in the same cycle. This is useful for low-radix routers.

`combineRCVA = false` `combineSAST = false`	`combineRCVA = true` `combineSAST = true`

In the base microarchitecture, stalls can occur due to a delay in reallocating an output virtual channel to a new packet. In the left diagram below, observe that no flit traverses the switch in cycle 4, due to the delay for the virtual channel to be freed in cycle 3.

When `coupleSAVA` is enabled, the freed virtual channel is immediately made available on the same cycle. However `coupleSAVA` can introduce long combinational paths on high-radix routers.

coupleSAVA = false	coupleSAVA = true

2.1.4.2 Payload Width

A router can specify its internal `payloadWidth`. When routers with different payload widths are connected by a channel, Constellation will autogenerate width-adapters on the channels if the widths are multiples of each other.

```
NoCParams(
  topology = Mesh2D(2, 2)
  routerParams = (i) =>␣
↪UserRouterParams(payloadWidth =
    if (i == 1 or i == 2) 128 else 64),
)
```

2.1.4.3 Virtual Channel Allocator

The has of the Virtual Channel Allocator has significant implications on the resulting performance of the NoC. Currently, there are two categories of allocators implemented

- **Single** VC allocators allocate only a single VC per cycle. These are useful in networks where most packets are multi-flit, as only the head flit needs to query the allocator
- **Multi** VC allocators attempt to allocate multiple VCs per cycle.

The following allocator implementations are provided.

- `PIMMultiVCAllocator` implements parallel-iterative-matching for a separable allocator
- `ISLIPMultiVCAllocator` implements the ISLIP policy for a separable allocator
- `RotatingSingleVCAllocator` rotates across incoming requests
- `PrioritizingSingleVCAllocator` prioritizes certain VCs over others, according to the priorities given by the routing relation

2.2 Flow Specification

The NoC's flow specification consits of two components.

```
// (blocker, blockee) => bool
// If true, then blocker must be able to proceed when blockee is blocked
vNetBlocking: (Int, Int) => Boolean = (_, _) => true,
flows: Seq[FlowParams] = Nil,
```

2.2.1 Flows

The `flows` field of `NoCParams` is a list of `FlowParams` case classes, where each flow uniquely identifies its source ingress terminal, destination egress terminal, and the virtual subnetwork identifier.

The `flows` parameter is motivated by the observation that the actually supported traffic patterns in a NoC is a subset of all possible ingress-egress pairs. Constellation can optimize the generated RTL for only the flows the network is expected to handle at runtime.

```
case class FlowParams(
  ingressId: Int,
  egressId: Int,
  vNetId: Int,
  fifo: Boolean = false
)
```

The `ingressId` and `egressId` fields index the specified ingresses and egresses of the NoC. That is, the ingress/egress indices decouple the flow specification from the underlying physical implementation of the network

The `vNetId` field can be used to specify a virtual subnetwork identifier to the flow. Virtual subnetworks are used to delineate between different channels of a actual messaging protocol, and is necessary for avoiding protocol-deadlock.

2.2.2 Virtual Subnetworks

The `vNetBlocking` function indicates which virtual subnetworks must make forwards progress while some other virtual subnetwork is blocked. If `vNetBlocking(x, y) == true`, then packets from subnetwork `x` must make forwards progress while packets of subnetwork `y` are stalled.

2.3 Routing Configuration

In Constellation, the desired routing relation is specified by the user as an abstract Scala function, instead of an RTL implementation of a routing table. Constellation will use this function to "compile" a hardware implementation of the desired routing algorithm.

```
abstract class RoutingRelation(topo: PhysicalTopology) {
  // Child classes must implement these
  def rel       (srcC: ChannelRoutingInfo,
                 nxtC: ChannelRoutingInfo,
                 flow: FlowRoutingInfo): Boolean

  def isEscape  (c: ChannelRoutingInfo,
                 vNetId: Int): Boolean = true

  def getNPrios (src: ChannelRoutingInfo): Int = 1

  def getPrio   (srcC: ChannelRoutingInfo,
                 nxtC: ChannelRoutingInfo,
                 flow: FlowRoutingInfo): Int = 0
```

2.3.1 Channel Identifier

The `ChannelRoutingInfo` case class uniquely identifies a virtual channel, ingress channel, or egress channel in the system.

```
case class ChannelRoutingInfo(
  src: Int,
  dst: Int,
  vc: Int,
  n_vc: Int
) {
```

The fields of this case class are:

- `src` is the source physical node of the channel. If this is an ingress channel, this value is -1.
- `dst` is the destination physical node of the channel. If this is an egress channel, this value is -1.
- `vc` is the virtual channel index within the channel.
- `n_vc` is the number of available virtual channels in this physical channel.

Consider a toy network depicted below with bidirectional physical channels, and two virtual channels on each physical channel. The list of all possible `ChannelRoutingInfo` in this network is shown

```
// Ingresses
ChannelRoutingInfo(src=-1, dst=0, vc=0, n_
↪vc=1)
ChannelRoutingInfo(src=-1, dst=1, vc=0, n_
↪vc=1)

// Egresses
ChannelRoutingInfo(src=0, dst=0, vc=0, n_
↪vc=1)
ChannelRoutingInfo(src=0, dst=3, vc=0, n_
↪vc=1)

// Routing channels
ChannelRoutingInfo(src=0, dst=1, vc=0, n_
↪vc=2)
ChannelRoutingInfo(src=0, dst=1, vc=1, n_
↪vc=2)
ChannelRoutingInfo(src=1, dst=0, vc=0, n_
↪vc=2)
ChannelRoutingInfo(src=1, dst=0, vc=1, n_
↪vc=2)
...
```

Note: In the current implementations, packets arriving at the egress physical node are always directed to the egress. Thus, `ChannelRoutingInfo` for the egress channels are not used. Additionally, this limitation prevents the implementation of deflection routing algorithms

2.3.2 Flow Identifier

The `FlowRoutingInfo` case class uniquely identifies a potential flow, or packet that might traverse the NoC.

```
case class FlowRoutingInfo(
  ingressId: Int,
  egressId: Int,
  vNetId: Int,
  ingressNode: Int,
  ingressNodeId: Int,
  egressNode: Int,
  egressNodeId: Int,
  fifo: Boolean
) {
```

The fields of this case class are:

- `ingressId` is the ingress index of the flow.
- `ingressNode` is the physical node of the ingress of this flow.
- `ingressNodeId` is the index of the ingress within all ingresses at the physical node.
- `egressId` is the egress index of the flow
- `egressNode` is the physical node of the egress of this flow
- `egressNodeId` is the physical node of the egress within all egresses at the physical node
- `vNetId` is the virtual subnetwork identifier of this flow

As an example, consider the same topology with two virtual subnetworks, for odd and even ingress and egresses.

```
                      egressId
         ingressNodeId  |  egressNode
        ingressNode  |  |  | ␣
↪egressNodeId
       ingressId  |  |  |  |  |  vNetId
                  |  |  |  |  |  |  |
FlowRoutingInfo(0, 0, 0, 0, 0, 0, 0)
FlowRoutingInfo(0, 0, 0, 2, 3, 1, 0)
FlowRoutingInfo(0, 0, 0, 2, 3, 1, 0)
FlowRoutingInfo(2, 1, 1, 0, 0, 0, 0)
FlowRoutingInfo(2, 1, 1, 2, 3, 1, 0)
FlowRoutingInfo(2, 1, 1, 2, 3, 1, 0)

FlowRoutingInfo(1, 1, 1, 1, 3, 1, 1)
FlowRoutingInfo(1, 1, 1, 1, 3, 1, 1)
FlowRoutingInfo(1, 1, 1, 1, 3, 1, 1)
```

2.3.3 Base Routing Relations

Numerous builtin routing relations are provided. These provide deadlock-free routing for the included topology generators.

RoutingRelation	Description
`AllLegalRouting`	Allows any packet to transition to any VC. Only useful for trivial networks
`UnidirectionalLineRouting`	Routing for a `UnidirectionalLine` topology.
`BidirectionalLineRouting`	Routing for a `BidirectionalLine` topology.
`UnidirectionalTorus1DDatelineRouting`	Routing for a `UnidirectionalTorus1D` topology. Uses a dateline to avoid deadlock. Requires at least 2 VCs per physical channel
`BidirectionalTorus1DRandomRouting`	Routing for a `BidirectionalTorus1D` topology. Uses a dateline to avoid deadlock. Packets randomly choose a direction at ingress. Requires at least 2 VCs per physical channel
`BidirectionalTorus1DShortestRouting`	Routing for a `BidirectionalTorus1D` topology. Uses a dateline to avoid deadlock. Packets choose direction at ingress which minimizes hops. Requires at least 2 VCs per physical channel.
`ButterflyRouting`	Routing for a `Butterfly` topology with arbitrary `kAry` and `nFly`.
`BidirectionalTreeRouting`	Routing for a `BidirectionalTree` topology.
`Mesh2DDimensionOrderedRouting`	Routing for a `Mesh2D` topology. Routes in one dimension first, than the other to prevent deadlock
`Mesh2DWestFirstRouting`	Routing for a `Mesh2D` topology. Routes in the westward dimension first to prevent deadlock
`MEsh2DNorthLastRouting`	Routing for a `Mesh2D` topology. Routes in the northward dimension last to prevent deadlock.
`Mesh2DEscapeRouting`	Escape-channel based adaptive minimal routing for `Mesh2D` topology. Escape channel routes using dimension ordered routing, while other channels route any minimal path.
`DimensionOrderedUnidirectionalTorus2DRouting`	Routing for a `UnidirectionalTorus2D` topology. Routes in the X dimension first.
`DimensionOrderedBidirectionalTorus2DRouting`	Routing for a `BidirectionalTorus2D` topology. Routes in the X dimension first.

2.3.4 Compositional Routing Relations

Several compositional routing relation classes are provided.

2.3.4.1 Escape Channel Routing

Escape-channel based routing can be used to build deadlock-free adaptive routing policies. When the routing policy on the escape virtual channels is deadlock-free, and packets in the escape channels cannot route on the non-escape virtual channels, the resulting policy is deadlock-free, even if circular dependencies exist among the non-escape virtual channels.

The `EscapeChannelRouting` class has three parameters:

- `escapeRouter`: the base deadlock-free routing algorithm that will be used on the escape VCs.
- `normalRouter`: the routing algorithm for all non-escape VCs.
- `nEscapeChannels`: the number of VCs to dedicate for escape channels on each physical channel

For example, a `Mesh2DEscapeRouting` algorithm with two escape channels using dimension-orderd routing can be described as:

```
EscapeChannelRouting(
  escapeRouter    = Mesh2DDimensionOrderedRouting(),
  normalRouter    = Mesh2DMinimalRouting(),
  nEscapeChannels = 2
)
```

2.3.4.2 Terminal Router Routing

`TerminalRouter` topologies require the `TerminalRouterRouting` routing relation. The `baseRouting` field is the base routing relation used to route among the non-terminal routers in the topology.

`topolog = TerminalRouter(BidirectionalLine(4))` `routing = TerminalRouterRouting(BidirectionalLineRouting())`	

2.3.4.3 Hierarchical Topology Routing

`Hierarchical` toplogies require the `HierarchicalRouting` routing relation. The `baseRouting` field is the routing relation for the base topology, while the `childRouting` field is alist of routing relations for the sub-topologies.

Hierarchical routing can be composed with terminal router routing.

```
topology = HierarchicalTopology(
  base = UnidirectionalTorus1D(4),
  children = Seq(
    HierarchicalSubTopology(0,2,
↪BidirectionalLine(5)),
    HierarchicalSubTopology(2,1,
↪BidirectionalLine(3))
  )
)
routing = HierarchicalRouting(
  baseRouting =␣
↪UnidirectionalTorus1DDatelineRouting(),
  childRouting = Seq(
    BidirectionalLineRouting(),
    BidirectionalLineRouting()
  )
)
```

```
topology =␣
↪TerminalRouter(HierarchicalTopology(
  base = UnidirectionalTorus1D(4),
  children = Seq(
    HierarchicalSubTopology(0,2,
↪BidirectionalLine(5)),
    HierarchicalSubTopology(2,1,
↪BidirectionalLine(3))
  )
))
routing =␣
↪TerminalRouter(HierarchicalRouting(
  baseRouting =␣
↪UnidirectionalTorus1DDatelineRouting(),
  childRouting = Seq(
    BidirectionalLineRouting(),
    BidirectionalLineRouting()
  )
)
```

2.3.5 Virtual Subnetwork Routing Relations

A special class of compositional routing relations can enforce forwards progress on networks with multiple virtual subnetworks.

- The `NonblockingVirtualSubnetworksRouting` policy guarantees all virtual subnetworks on the network can always make forwards progress by dedicating some number of virtual channels to each subnetwork.
- The `BlockingVirtualSubnetworksRouting` policy guarantees that lower-indexed virtual subnetworks can always make forwards progress while higher-indexed virtual subnetworks are blocked.

CHAPTER

THREE

PROTOCOL SUPPORT

Constellation supports an interface to automatically wrap a multi-channel communication protocol on top of the NoC. This section describes the abstract interface for layering any potential protocol onto the NoC, as well as the implementations which provide AXI-4 and TileLink compliant interconnects.

3.1 Abstract Protocol Interface

To layer a protocol on top of the NoC, an instance of a case class extending `ProtocolParams` must be constructed.

```
trait ProtocolParams {
  val minPayloadWidth: Int
  val ingressNodes: Seq[Int]
  val egressNodes: Seq[Int]
  val nVirtualNetworks: Int
  val vNetBlocking: (Int, Int) => Boolean
  val flows: Seq[FlowParams]
  def genIO()(implicit p: Parameters): Data
  def interface(
    terminals: NoCTerminalIO,
    ingressOffset: Int,
    egressOffset: Int,
    protocol: Data)(implicit p: Parameters)
}
```

The required fields of `ProtocolParams` must be provided as follows:

- `minPayloadWidth` is the required minimum payload width for flits to transport this protocol. It is the minimum payload width at the ingress and egress terminals.
- `ingressNodes` is an ordered list of physical node destination for all ingress terminals
- `egressNodes` is an ordered list of physical node sources for all egress terminals
- `nVirtualNetworks` is the number of virtual subnets in this protocol, often this is the number of protocol channels.
- `vNetBlocking` describes the blocking/nonblocking relationships between virtual subnetworks in this protocol
- `flows` is a list of possible flows for the protocol
- `genIO` returns the protocol-level IO for the entire interconnect
- `interface` provides the interface for the NoC through `terminals: NoCTerminalIO` and the interface for the protocol through `protocol: Data`. This function should instantiate the user-defined protocol adapters to connect the two interfaces.

3.2 Standalone Protocol NoC

A NoC with protocol support can be elaborated in Chisel using the `ProtocolNoC` generator. The `ProtocolNoCParams` case class parameterizes both the network (through `nocParams`), as well as the protocol interface (though a list of `ProtocolParams`.

```
case class ProtocolNoCParams(
  nocParams: NoCParams,
  protocolParams: Seq[ProtocolParams]
)
class ProtocolNoC(params: ProtocolNoCParams)(implicit p: Parameters) extends Module {
  val io = IO(new Bundle {
    val ctrl = if (params.nocParams.hasCtrl) Vec(params.nocParams.topology.nNodes, new␣
↪RouterCtrlBundle) else Nil
    val protocol = MixedVec(params.protocolParams.map { u => u.genIO() })
  })
```

Note: Multiple protocols can be supported on a shared interconnect by passing multiple `ProtocolParams` to `protocolParams`

Note: Harnesses for testing standalone non-diplomatic protocol-capable NoCs are currently not provided. At the moment, Diplomatic integration is recommended.

3.2.1 TileLink Protocol Params

The `TileLinkProtocolParams` case class describes an instance of `ProtocolParams` for TileLink-C. `edgesIn` and `edgesOut` are ordered list of inwards and outwards TileLink edges (from masters and slaves, respectively). `edgeInNodes` and `edgeOutNodes` map the masters and slaves to physical node indices. The definition of the `TLEdge` parameter class can be found in Rocketchip.

```
case class TileLinkProtocolParams(
  edgesIn: Seq[TLEdge],
  edgesOut: Seq[TLEdge],
  edgeInNodes: Seq[Int],
  edgeOutNodes: Seq[Int]
) extends ProtocolParams with TLFieldHelper {
```

Note: TL-C is a superset of TL-UL and TL-UH, so this implementation supports all potential TileLink implementations.

3.2.2 AXI-4 Protocol Params

The `AXI4ProtocolParams` case class describes an instance of `ProtocolParams` for AXI4. `edgesIn` and `edgesOut` are ordered list of inwards and outwards AXI-4 edges (from masters and slaves, respectively). `edgeInNodes` and `edgeOutNodes` map the masters and slaves to physical node indices. The definition of the `AXI4EdgeParameters` class can be found in Rocketchip.

```
case class AXI4ProtocolParams(
  edgesIn: Seq[AXI4EdgeParameters],
  edgesOut: Seq[AXI4EdgeParameters],
  edgeInNodes: Seq[Int],
  edgeOutNodes: Seq[Int],
  awQueueDepth: Int
) extends ProtocolParams {
```

3.3 Diplomatic Protocol NoC

The common approach for generating interconnects within a Chipyard/Rocketchip-based SoC is to use the Diplomatic wrapper for a protocol-capable interconnet. Diplomacy's parameter negotiation will automatically construct a `ProtocolParams` case class to construct the `TileLinkProtocolParams` or `AXI4ProtocolParams` case classes from the Diplomatic graph.

These modules can replace existing `TLXbar()` and `AXI4Xbar()` Diplomatic modules in any Chipyard/Rocketchip design.

- `nocParams`: `NoCParams` describes the physical topology and routing relation of the network
- `nodeMappings`: `DiplmaticNetworkNodeMapping` contains mappings from `String` to a `Int` to map Diplomatic edges to physical node identifiers

```
case class AXI4NoCParams(
  nodeMappings: DiplomaticNetworkNodeMapping,
  nocParams: NoCParams,
  awQueueDepth: Int = 2
)
class AXI4NoC(params: AXI4NoCParams, name: String = "test")(implicit p: Parameters)␣
↪extends LazyModule {
```

```
case class TLNoCParams(
  nodeMappings: DiplomaticNetworkNodeMapping,
  nocParams: NoCParams = NoCParams()
)
class TLNoC(params: TLNoCParams, name: String = "test")(implicit p: Parameters) extends␣
↪TLNoCLike {
```

```
case class DiplomaticNetworkNodeMapping(
  inNodeMapping: ListMap[String, Int] = ListMap[String, Int](),
  outNodeMapping: ListMap[String, Int] = ListMap[String, Int]()
) {
```

CHAPTER

FOUR

CHIPYARD SOC INTEGRATION

Constellation NoCs can replace any TileLink crossbar in a Chipyard/Rocketchip SoC. The most common use case is for generating the interconnect for the SystemBus or the MemoryBus.

Examples of NoC integration can be found in the `NoCConfigs.scala` file in Chipyard, in the `MultiNoCConfig`.

4.1 Global Shared Interconnect

While the default approach for integrating NoCs generates an independent network for each logical interconnect (i.e. the SystemBus and MemoryBus are wholly independent strucutres), Constellation also supports a shared global interconnect that is still deadlock-free. Configs pursuing this style of integration should set the `GlobalNoCParams` field with `constellation.soc.WithGlobalNoC`.

An example of a global shared NoC config is the `SharedNoCConfig` in `NoCConfigs.scala` in Chipyard.

CHAPTER

FIVE

EVALUATION FRAMEWORK

Constellation includes a lightweight C++ evaluation framework, designed to support future advanced traffic models. Currently, the framework supports measuring per-flow latency and bandwidth using two traffic models.

- An injection-rate based model, with per-flow injection rates
- A trace-driven model, which loads Netrace trace files

5.1 Running a Simple Evaluation

Traffic evaluations are best run in Chipyard-standalone mode.

```
cd chipyard/sims/vcs
make SUB_PROJECT=constellation BINARY=none CONFIG=EvalTestConfig00 MODEL=EvalHarness run-
↪binary-debug
```

A generated `noceval.cfg` file will be generated in the `generated-src` directory. Copy this file elsewhere and modify it to adjust the simulation parameters. The important fields of this config file are:

- `warmup`: Number of cycles to spend in the warmup phase, to bring the network to steady-state
- `measurement`: Number of cycles after warmup in which throughput is measured
- `drain`: Number of cycles after measurement to wait for the network to drain. If the network does not drain in this many cycles, an assertion is fired
- `flits_per_packet`: Packet size
- `required_XXX`: Required throughput, median latency, max latency. IF measurement exceeds these, an assertion fires.
- `netrace_enable`: Use Netrace trace file as traffic model
- `netrace_trace`: Path to Netrace trace file
- `netrace_region`: Netrace region to begin trace replay at
- `flow x y z`: Specifies injection rate `z` for flow from ingress index `x` to egress index `y`

After modifying a `noceval.cfg` flag, the simulation can be rerun with:

```
cd chipyard/sims/vcs
make SUB_PROJECT=constellation BINARY=none CONFIG=EvalTestConfig00 MODEL=EvalHarness run-
↪binary-debug EXTRA_SIM_FLAGS="+eval_params=path/to/noceval.cfg"
```

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search