The Data Center Network L1 Switch Protocol

Thomas L. Rodeheffer
Microsoft Research, Silicon Valley

April 27, 2011

1 Overview

Chuck Thacker *et al.* have proposed a data center network design using FPGAs to implement the switching infrastructure [2]. Inside each switch, the design uses time slot circuit switching of data cells via a distributed crossbar in the data plane and best-effort forwarding of control messages via a high-speed ring in the control plane. The goal is to produce a system with minimum total complexity. Existing networks are extremely complex, have many different failure modes, and are difficult to scale out to the size of a large data center.

The design fits in the context of a data center built in shipping containers. Each container holds a large number of servers and L0 switches, and holds two, top-level L1 switches, called A and B. Cables are run between all the containers to form a complete interconnection graph of the A L1 switches and likewise for the B L1 switches.

This paper presents a TLA+ [1] specification of the L1 switch infrastructure.

1.1 Simplifications

This specification simplifies the full design in a number of respects.

1.1.1 No L0 switches

The full design consists of 64 containers. Each container has 1280 servers, 64 L0 switches, and two L1 switches, called A and B.

This specification concerns itself with the A L1 switches only, ignoring all the servers, L0 switches, and B L1 switches. Where there would be a connection to an L0 switch, the specification connects to a model of an end node that sources and sinks traffic, thus simulating the server nics that would be reached through the L0 switches in the full design.

1.1.2 No link frames

In the full design, data cells and control cells are sent on the links structured inside link frames. Each input unit has a deframer that extracts the cells from the input link frame and each output unit has an enframer that packs the cells into the output link frame.

Furthermore, each input unit uses a double buffer of frame data cells to reorder the data cells between the slot time in their arriving frame and the slot time in which they are sucked through the crossbar into their outgoing frame.

This specification ignores the link frames. In this specification, data cells and control cells are sent on links in any sequence. Each input unit uses a queue per slot to reorder the data cells from their arriving times to their departing times.

1.1.3 Assumed timing constraint

In the full design, worst-case timing assumptions are made in order to guarantee the following timing constraint.

During connection setup, when switch A's down port processes a setup request message and successfully allocates a local slot, two control paths emanate:

- On the first control path, the down port generates a setup answer message which returns around the ring to the upport. The up port converts the setup answer message to a setup xbar message which circles the ring to setup the connection in all slices of the distributed crossbar. Eventually, the distributed crossbar is completely setup and cells that arrive at the down port during the assigned local slot correspond to cells arriving on the new connection at the up port.

- On the second control path, the down port generates a setup request control cell that it sends downstream across the link to the downstream switch or node. If the connection is successfully setup in the downstream component, eventually it replies with a setup ack control cell. This cell arrives at switch A's down port, informing it of the value to use for nextSlot on the downstream link. At this point the down port begins forwarding cells arriving through the crossbar on the new connection.

The timing constraint is that the first control path must complete before the second control path. Otherwise, the down port could forward arbitrary garbage cells on the downstream link, and this would be bad.

The specification guarantees the timing constraint by brute force. A connection identifier is associated with all control cells and crossbar setups, and the setup ack is prevented from being received until the crossbar has been setup.

1.2 Terminology

Terminology in the data center network design document [2] tends to be incomplete and wordy. That is, not everything we need to talk about has a crisp name. So, in this specification, we adopt the following terminology.

The specification has to deal with objects, which often have internal state and actions upon them. Objects can be complicated and contain other objects. We have the following examples of objects.

- A *data cell* object, with type name DataCell (section 9). Data cell objects are sent over links, buffered at input ports, and forwarded through a switch crossbar to output ports.

- A *control cell* object, with type name CtrlCell (section 10). Control cell objects are sent over links from output ports to input ports.
- A *ring message* object, with type name RingMsg (section 12). Ring message objects are sent around the elastic ring inside a switch.
- A *switch* object, with type name Switch (section 18).
- A *port* object (part of a switch), with type name Port (section 14). A port object is called an "i/o unit" in the design document.
- A *gare* object (part of a switch), with type name Gare (section 17). A gare object manages a station on the control ring within a switch on behalf of a number of ports. ("Gare" is the French word for "train station".) A gare is called a "ring master" in the design document. Unfortunately, "ring master" and "ring message" have the same initials, so in this specification we use a different term for "ring master".
- A *node* object, with type name Node (section 11). A node object models the network interface card in a server.

In addition to objects, the specification has to deal with references to objects (or to places). In full generality, a reference is a *name*, which can be complicated and contain internal structure. We have the following examples of names.

- A *slot name*, with type name SlotName (section 4). A slot name has an internal structure that identifies the subframe and slottina within the subframe.
- A *port name* (locally within a switch), with type name PortName (section 13). A port name has an internal structure with multiple variants, one variant for a port name of a port that connects externally to a container and another for a port name of a port that connects externally to a halfrow.
- A *gare name* (locally within a switch), with type name GareName (section 15). A gare name has an internal structure which identifies the set of ports it manages.
- A *switch name*, with type name SwitchName (section 16). A switch name has an internal structure which identifies the container of the L1 A switch.

1.3 Modules

The specification is written as a collection of modules, in which each module provides definitions for a particular data type or component of the design. In this way the complexity of the overall design is factored into logical pieces that can be understood one at a time.

First we specify some useful definitions (section 2) and basic system parameters (section 3).

Then we build up basic system modules. This requires specifying the definition of a *chunk sequence number* (section 5), the definition of a *CRC* (section 6), the definition

of a *slot name* (section 4), the definition of *cell type* (section 7), the definition of *control queue type* (section 8), the format of a *data cell* (section 9), the format of a *control cell* (section 10), and the internal state and actions of an *end node* (section 11).

Then we add many modules requires to specify the components of an L1 switching network. This requires specifying the format of a *ring message* (section 12), the definition of a *port name* (section 13), the definition of a *gare name* (section 15, the definition of a *switch name* (section 16, the internal state and actions of a *port* (section 14), the internal state and actions of a *gare* (section 17), the internal state and actions of a *switch* (section 18), and finally a system to connect a number of switches and end nodes (section 19).

2 Useful definitions

MODULE $DCNL1\,Useful$

EXTENDS $Naturals$, $Sequences$, $FiniteSets$, TLC

Useful TLA constants and operators.

A model constant representing a value that is never (supposed to be) read. A field can be assigned to $NULL$ but there should be no checks to see if a field is $NULL$.

CONSTANT $NULL$

Nullable version of type T. This is used to define the type of a field that may be assigned $NULL$ in some situations.

$Nullable(T) \triangleq T \cup \{NULL\}$

Assert that x is in type T. Returns TRUE if no error.

$TypeAssert(x,\ T) \triangleq$
  IF $x \in T$ THEN TRUE ELSE
  $Assert(\text{FALSE},\ \langle \text{“type error”},\ x\rangle)$

The head of sequence s if s is non-empty, else n.

$HeadElse(s,\ n) \triangleq$ IF $s \neq \langle\rangle$ THEN $Head(s)$ ELSE n

The tail of sequence s if s is non-empty, else $\langle\rangle$.

$TailElse(s) \triangleq$ IF $s \neq \langle\rangle$ THEN $Tail(s)$ ELSE $\langle\rangle$

Choose an arbitrary sequence of “names” (the elements of a set).

In TLA+ this would more naturally be written as a submodule taking the set $Name$ as a constant and defining all these symbols as operators. The module would then be instantiated using the desired set as a parameter. Unfortunately, TLC fails to recognize that the resulting operators are level 0 constants, and it repeatedly reevaluates them during model checking.

So instead we write this as an operator that returns a record of maps. In this style, TLC does recognize that everything is a level 0 constant, and it evaluates them only once before computing the initial states.

$ArbSeq(Name) \triangleq$
LET
Number of names.
$N \triangleq Cardinality(Name)$

The index set.
$Index \triangleq 1 \,..\, N$

Get name for a given index.
$NameForIndex \triangleq$
LET
RECURSIVE $Recur(\_, \_)$
$Recur(nn, q) \triangleq$
IF $nn = \{\}$ THEN q ELSE
LET $n \triangleq$ CHOOSE $n \in nn$: TRUEIN
$Recur(nn \setminus \{n\}, Append(q, n))$
IN
$Recur(Name, \langle\rangle)$

Get index for a given name.
$IndexForName \triangleq [n \in Name \mapsto$ CHOOSE $i \in Index : NameForIndex[i] = n]$

Operations on indexes.
$FirstIndex \triangleq 1$
$IncrIndex \triangleq [i \in Index \mapsto ((i - 1 + N + 1)\%N) + 1]$ cyclic
$DecrIndex \triangleq [i \in Index \mapsto ((i - 1 + N - 1)\%N) + 1]$ cyclic
$LeqIndex \triangleq [i1, i2 \in Index \mapsto i1 \leq i2]$

Operations on names.
$FirstName \triangleq NameForIndex[FirstIndex]$
$IncrName \triangleq [n \in Name \mapsto NameForIndex[IncrIndex[IndexForName[n]]]]$
$DecrName \triangleq [n \in Name \mapsto NameForIndex[DecrIndex[IndexForName[n]]]]$
$LeqName \triangleq [n1, n2 \in Name \mapsto LeqIndex[IndexForName[n1], IndexForName[n2]]]$
IN
[
$Index \mapsto Index,$

$NameForIndex \mapsto NameForIndex,$
$IndexForName \mapsto IndexForName,$
$FirstIndex \mapsto FirstIndex,$
$IncrIndex \mapsto IncrIndex,$
$DecrIndex \mapsto DecrIndex,$
$LeqIndex \mapsto LeqIndex,$
$FirstName \mapsto FirstName,$
$IncrName \mapsto IncrName,$
$DecrName \mapsto DecrName,$
$LeqName \mapsto LeqName$
]

3 Parameters

MODULE $DCNL1Param$

EXTENDS $DCNL1Useful$

Parameterization constants. Values in the full design are listed in parenthesis.

CONSTANT $NumContainer$ number of containers (64)
CONSTANT $NumHalfrow$ number of halfrows (64)

Note that hardware implementation of the following definitions is zero-based. TLA+ likes it better for natural number ranges to start with one, so we define it the TLA+ way.

$Container \triangleq 1 \,..\, NumContainer$
$Halfrow \triangleq 1 \,..\, NumHalfrow$

$Reason \triangleq$ STRING teardown reason, opaque
$Throttle \triangleq$ BOOLEAN throttle indication

CONSTANT $Payload$ normal data cell payload (32 bytes)

For the purpose of the specification, each connection has an identification that is carried by every data cell, control cell, ring message, and status record pertaining to the connection. This identification is not present in the hardware, but is used to specify certain protocol timing assumptions that are guaranteed by the hardware.

CONSTANT $NumConnection$

$Connection \triangleq 1 \,..\, NumConnection$

4 Slot name

MODULE $DCNL1SlotName$

EXTENDS $DCNL1Useful$

Time on the links, and in the data path inside a switch, is divided into recurring frames. Each frame consists of a sequence of subframes. Each $subframe$ consists of a sequence of slottinas.

A slot is a particular $slottina$ inside a particular $subframe$.

The existence of "$subframe$" in the time hierarchy is partly an implementation artifact in that the hardware design did not want to spend the effort to carry a full bit vector of slot-available status around the ring in the "$SetupReq$" message. Note that the "$SetupReq$" message is already the biggest ring message, being the only two word message in the hardware.

Parameterization constants. Values in the full design are listed in parenthesis.

CONSTANT $NumSubframe$ number of subframes (4)
CONSTANT $NumSlottina$ number of slottinas (32)

Note that hardware implementation of the following definitions is zero-based. TLA+ likes it better for natural number ranges to start with one, so we define it the TLA+ way.

$$Subframe \triangleq 1 \,..\, NumSubframe$$
$$Slottina \triangleq 1 \,..\, NumSlottina$$

Define slot name.

$$SlotName \triangleq [$$
$$\quad subframe : Subframe,$$
$$\quad slottina \ : Slottina$$
$$]$$

Define slot index based on an arbitrary ordering of slot names.

$$FirstSlotName \triangleq ArbSeq(SlotName).FirstName$$
$$IncrSlotName \triangleq ArbSeq(SlotName).IncrName$$
$$DecrSlotName \triangleq ArbSeq(SlotName).DecrName$$
$$LeqSlotName \triangleq ArbSeq(SlotName).LeqName$$

5 Chunk sequence number

MODULE $DCNL1Chunkseqn$

EXTENDS $DCNL1Useful$

Chunk sequence number.

Parameterization constants. Values in the full design are listed in parenthesis.

CONSTANT $NumChunkseqn$ number of chunk sequence numbers ($2 * *16$)

Note that hardware implementation of the following definition is zero-based. TLA+ likes it better for natural number ranges to start with one, so we define it the TLA+ way.

$Chunkseqn \triangleq 1 \,..\, NumChunkseqn$

$FirstChunkseqn \triangleq 1$
$IncrChunkseqn(x) \triangleq (x\%NumChunkseqn) + 1$
$DecrChunkseqn(x) \triangleq \text{CHOOSE } x0 \in Chunkseqn : IncrChunkseqn(x0) = x$

6 CRC

MODULE $DCNL1CRC$

EXTENDS $DCNL1Param$

We model a CRC as a record containing the sequence of payload values it was computed from.

$CRC \triangleq [crc : Seq(Payload)]$

The initial value of a CRC.

$InitCRC \triangleq [crc \mapsto \langle\rangle]$

Update a crc by extending it with a payload.

$ExtendCRC(crc,\ p) \triangleq [crc \text{ EXCEPT } !.crc = Append(@,\ p)]$

7 Cell type

MODULE $DCNL1CellType$

$CellType\_Data \triangleq$ "CellType_Data"
$CellType\_Ctrl \triangleq$ "CellType_Ctrl"

$CellType \triangleq \{$
$\quad CellType\_Data,$
$\quad CellType\_Ctrl$
$\}$

8 Control queue type

MODULE $DCNL1CtrlQueueType$

$CtrlQueueType\_Req \triangleq$ "CtrlQueueType_Req"
$CtrlQueueType\_Ups \triangleq$ "CtrlQueueType_Ups"

$CtrlQueueType \triangleq \{$
$\quad CtrlQueueType\_Req,$
$\quad CtrlQueueType\_Ups$
$\}$

9 Data cell

MODULE $DCNL1DataCell$

EXTENDS
$\quad DCNL1Param,$
$\quad DCNL1SlotName,$
$\quad DCNL1Chunkseqn,$
$\quad DCNL1CRC$

Data cells on the link.

In the hardware, each data cell has a 2 byte header and a 32 byte $payload$. The header has a parity bit which we ignore.

$DataCell\_Typecode\_Null \triangleq$ "DataCell_Null"
$DataCell\_Typecode\_Normal \triangleq$ "DataCell_Normal"
$DataCell\_Typecode\_FirstMark \triangleq$ "DataCell_FirstMark"
$DataCell\_Typecode\_ChunkMark \triangleq$ "DataCell_ChunkMark"
$DataCell\_Typecode\_EndMark \triangleq$ "DataCell_EndMark"

$$
\begin{array}{l}
DataCell\_Null \triangleq [\\
\quad typecode \ : \{DataCell\_Typecode\_Null\} \\
]
\end{array}
$$

$$
\begin{array}{l}
DataCell\_Normal \triangleq [\\
\quad typecode \ : \{DataCell\_Typecode\_Normal\}, \\
\quad connect \ : Connection, \\
\quad nextSlot \ : SlotName, \\
\quad payload \ : Payload \\
]
\end{array}
$$

$$
\begin{array}{l}
DataCell\_FirstMark \triangleq [\\
\quad typecode \ : \{DataCell\_Typecode\_FirstMark\}, \\
\quad connect \ : Connection, \\
\quad nextSlot \ : SlotName, \\
\quad destC \ : Container, \\
\quad destH \ : Halfrow \\
\quad \text{this } L1 \text{ model terminates at the halfrow, hence there is no } DestN \\
]
\end{array}
$$

$$
\begin{array}{l}
DataCell\_ChunkMark \triangleq [\\
\quad typecode \ : \{DataCell\_Typecode\_ChunkMark\}, \\
\quad connect \ : Connection, \\
\quad nextSlot \ : SlotName, \\
\quad chunkseqn : Chunkseqn, \\
\quad crc \ : CRC \\
]
\end{array}
$$

$$
\begin{array}{l}
DataCell\_EndMark \triangleq [\\
\quad typecode \ : \{DataCell\_Typecode\_EndMark\}, \\
\quad connect \ : Connection, \\
\quad nextSlot \ : SlotName \\
]
\end{array}
$$

$$
\begin{array}{l}
DataCell \triangleq \{\} \\
\quad \cup DataCell\_Null \\
\quad \cup DataCell\_Normal \\
\quad \cup DataCell\_FirstMark \\
\quad \cup DataCell\_ChunkMark \\
\quad \cup DataCell\_EndMark
\end{array}
$$

$$NullDataCell \triangleq [typecode \mapsto DataCell\_Typecode\_Null]$$

$$
\begin{array}{ll}
DataCell\_Is\_Null(dc) & \triangleq \ dc.typecode = DataCell\_Typecode\_Null \\
DataCell\_Is\_Normal(dc) & \triangleq \ dc.typecode = DataCell\_Typecode\_Normal \\
DataCell\_Is\_FirstMark(dc) & \triangleq \ dc.typecode = DataCell\_Typecode\_FirstMark \\
DataCell\_Is\_ChunkMark(dc) & \triangleq \ dc.typecode = DataCell\_Typecode\_ChunkMark
\end{array}
$$

$DataCell\_Is\_EndMark(dc) \triangleq dc.typecode = DataCell\_Typecode\_EndMark$

Nice view for debugging.

$DataCell\_Debug\_Nice(dc) \triangleq$
LET
$sn \triangleq dc.nextSlot$
$nulldc \triangleq \langle$"Null"$\rangle$
$normaldc \triangleq \langle$"DC", sn, $dc.payload\rangle$
$firstmdc \triangleq \langle$"FM", $sn\rangle$
$chunkmdc \triangleq \langle$"CM", sn, $dc.chunkSeqn\rangle$
$endmdc \triangleq \langle$"EM", $sn\rangle$
IN
IF $DataCell\_Is\_Null(dc)$ THEN $nulldc$ ELSE
IF $DataCell\_Is\_Normal(dc)$ THEN $normaldc$ ELSE
IF $DataCell\_Is\_FirstMark(dc)$ THEN $firstmdc$ ELSE
IF $DataCell\_Is\_ChunkMark(dc)$ THEN $chunkmdc$ ELSE
IF $DataCell\_Is\_EndMark(dc)$ THEN $endmdc$ ELSE
$Assert$(FALSE, "type error")

10 Control cell

MODULE $DCNL1CtrlCell$

EXTENDS
$DCNL1Param$,
$DCNL1SlotName$,
$DCNL1Chunkseqn$

Control cells on the link.

In the hardware, each control cell fits in a 32 bit word and contains a parity bit. Control cells received with bad parity are ignored. We do not model the parity bit.

$CtrlCell\_Typecode\_SetupReq \triangleq$ "CtrlCell_SetupReq"
$CtrlCell\_Typecode\_SetupAck \triangleq$ "CtrlCell_SetupAck"
$CtrlCell\_Typecode\_Teardown \triangleq$ "CtrlCell_Teardown"
$CtrlCell\_Typecode\_Discard \triangleq$ "CtrlCell_Discard"
$CtrlCell\_Typecode\_DataAck \triangleq$ "CtrlCell_DataAck"
$CtrlCell\_Typecode\_Restart \triangleq$ "CtrlCell_Restart"
$CtrlCell\_Typecode\_Stop \triangleq$ "CtrlCell_Stop"
$CtrlCell\_Typecode\_Null \triangleq$ "CtrlCell_Null"

$CtrlCell\_SetupReq \triangleq [$
  $typecode : \{CtrlCell\_Typecode\_SetupReq\},$
  $connect : Connection,$
  $DestC : Container,$
  $DestH : Halfrow,$
  $prevSlot : SlotName$
  this $L1$ model terminates at the halfrow, hence there is no $DestN$
  this $L1$ model omits the $L0$ switches, hence there is no $FirstL1$
  this $L1$ model omits VLB routing
]

$CtrlCell\_SetupAck \triangleq [$
  $typecode : \{CtrlCell\_Typecode\_SetupAck\},$
  $connect : Connection,$
  $prevSlot : SlotName,$
  $nextSlot : SlotName$
]

$CtrlCell\_Teardown \triangleq [$
  $typecode : \{CtrlCell\_Typecode\_Teardown\},$
  $connect : Connection,$
  $prevSlot : SlotName,$
  $reason : Reason$
]

$CtrlCell\_Discard \triangleq [$
  $typecode : \{CtrlCell\_Typecode\_Discard\},$
  $connect : Connection,$
  $prevSlot : SlotName$
]

$CtrlCell\_DataAck \triangleq [$
  $typecode : \{CtrlCell\_Typecode\_DataAck\},$
  $connect : Connection,$
  $prevSlot : SlotName,$
  $throttle : \text{BOOLEAN},$
  $chunkseqn : Chunkseqn$
]

$CtrlCell\_Restart \triangleq [$
  $typecode : \{CtrlCell\_Typecode\_Restart\},$
  $connect : Connection,$
  $prevSlot : SlotName$
]

$CtrlCell\_Stop \triangleq [$
  $typecode : \{CtrlCell\_Typecode\_Stop\},$
  $connect : Connection,$

$$
\begin{array}{l}
\quad prevSlot \ : SlotName \\
]
\end{array}
$$

$$
\begin{array}{l}
CtrlCell\_Null \triangleq [\\
\quad typecode \ : \{CtrlCell\_Typecode\_Null\} \\
]
\end{array}
$$

$$
\begin{array}{l}
CtrlCell \triangleq \{\} \\
\quad \cup \ CtrlCell\_Null \\
\quad \cup \ CtrlCell\_SetupReq \\
\quad \cup \ CtrlCell\_SetupAck \\
\quad \cup \ CtrlCell\_Teardown \\
\quad \cup \ CtrlCell\_Discard \\
\quad \cup \ CtrlCell\_DataAck \\
\quad \cup \ CtrlCell\_Restart \\
\quad \cup \ CtrlCell\_Stop
\end{array}
$$

$$NullCtrlCell \triangleq [typecode \mapsto CtrlCell\_Typecode\_Null]$$

$$
\begin{array}{ll}
CtrlCell\_Is\_SetupReq(cc) & \triangleq \ cc.typecode = CtrlCell\_Typecode\_SetupReq \\
CtrlCell\_Is\_SetupAck(cc) & \triangleq \ cc.typecode = CtrlCell\_Typecode\_SetupAck \\
CtrlCell\_Is\_Teardown(cc) & \triangleq \ cc.typecode = CtrlCell\_Typecode\_Teardown \\
CtrlCell\_Is\_Discard(cc) & \triangleq \ cc.typecode = CtrlCell\_Typecode\_Discard \\
CtrlCell\_Is\_DataAck(cc) & \triangleq \ cc.typecode = CtrlCell\_Typecode\_DataAck \\
CtrlCell\_Is\_Restart(cc) & \triangleq \ cc.typecode = CtrlCell\_Typecode\_Restart \\
CtrlCell\_Is\_Stop(cc) & \triangleq \ cc.typecode = CtrlCell\_Typecode\_Stop \\
CtrlCell\_Is\_Null(cc) & \triangleq \ cc.typecode = CtrlCell\_Typecode\_Null
\end{array}
$$

11 End node

MODULE $DCNL1Node$

EXTENDS
$DCNL1Useful$,
$DCNL1Param$,
$DCNL1SlotName$,
$DCNL1Cell$,
$DCNL1CellType$

Emulate an endnode nic that lives at an $L1$ switch halfrow.

CONSTANT $MaxChunkCells$ maximum size of a chunk

$BugRecvWrongPacket \triangleq$ FALSE receive some arbitrary packet
$BugRecvWrongCRC \triangleq$ FALSE receive incorrect crc

$TraceRecvPacket \triangleq$ FALSE

Per slot input state.

$NodeInSlotState\_Avail \triangleq$ "Avail"
$NodeInSlotState\_WaitFirstMark \triangleq$ "WaitFirstMark"
$NodeInSlotState\_WaitChunk \triangleq$ "WaitChunk"
$NodeInSlotState\_End \triangleq$ "End"

$$
\begin{aligned}
&NodeInSlotState \triangleq \{ \\
&\quad NodeInSlotState\_Avail, \\
&\quad NodeInSlotState\_WaitFirstMark, \\
&\quad NodeInSlotState\_WaitChunk, \\
&\quad NodeInSlotState\_End \\
&\}
\end{aligned}
$$

$$
\begin{aligned}
&NodeInSlot \triangleq [\\
&\quad i\_state \quad : NodeInSlotState, \\
&\quad i\_connect \quad : Nullable(Connection), \\
&\quad i\_prevSlot \quad : Nullable(SlotName), \\
&\quad i\_chunkseqn : Chunkseqn, \\
&\quad i\_chunkcrc \quad : Nullable(CRC), \\
&\quad i\_chunk \quad : Seq(Payload), \\
&\quad i\_packet \quad : Seq(Payload) \\
&]
\end{aligned}
$$

$$
\begin{aligned}
&InitNodeInSlot \triangleq [\\
&\quad i\_state \quad \mapsto NodeInSlotState\_Avail, \\
&\quad i\_connect \quad \mapsto NULL, \\
&\quad i\_prevSlot \quad \mapsto NULL, \\
&\quad i\_chunkseqn \mapsto FirstChunkseqn, \\
&\quad i\_chunkcrc \quad \mapsto NULL, \\
&\quad i\_chunk \quad \mapsto \langle\rangle, \\
&\quad i\_packet \quad \mapsto \langle\rangle \\
&]
\end{aligned}
$$

Per slot output state.

$NodeOutSlotState\_Avail \triangleq$ "Avail"
$NodeOutSlotState\_SendSetupReq \triangleq$ "SendSetupReq"
$NodeOutSlotState\_WaitSetupAck \triangleq$ "WaitSetupAck"
$NodeOutSlotState\_SendFirstMark \triangleq$ "SendFirstMark"
$NodeOutSlotState\_SendChunkData \triangleq$ "SendChunkData"
$NodeOutSlotState\_SendChunkMark \triangleq$ "SendChunkMark"
$NodeOutSlotState\_WaitDataAck \triangleq$ "WaitDataAck"
$NodeOutSlotState\_SendEndMark \triangleq$ "SendEndMark"
$NodeOutSlotState\_WaitTeardown \triangleq$ "WaitTeardown"
$NodeOutSlotState\_Abort \triangleq$ "Abort"
$NodeOutSlotState\_End \triangleq$ "End"

$$
\begin{array}{l}
NodeOutSlotState \triangleq \{ \\
\quad NodeOutSlotState\_Avail, \\
\quad NodeOutSlotState\_SendSetupReq, \\
\quad NodeOutSlotState\_WaitSetupAck, \\
\quad NodeOutSlotState\_SendFirstMark, \\
\quad NodeOutSlotState\_SendChunkData, \\
\quad NodeOutSlotState\_SendChunkMark, \\
\quad NodeOutSlotState\_WaitDataAck, \\
\quad NodeOutSlotState\_SendEndMark, \\
\quad NodeOutSlotState\_WaitTeardown, \\
\quad NodeOutSlotState\_Abort, \\
\quad NodeOutSlotState\_End \\
\}
\end{array}
$$

$$
\begin{array}{l}
NodeOutSlot \triangleq [\\
\quad o\_state \quad : NodeOutSlotState, \\
\quad o\_connect \quad : Nullable(Connection), \\
\quad o\_destC \quad : Nullable(Container), \\
\quad o\_destH \quad : Nullable(Halfrow), \\
\quad o\_nextSlot \quad : Nullable(SlotName), \\
\quad o\_chunkseqn : Chunkseqn, \\
\quad o\_chunkcrc \quad : Nullable(CRC), \\
\quad o\_ixatchunk : Nat, \\
\quad o\_ixinchunk : Nat, \\
\quad o\_packet \quad : Seq(Payload) \\
]
\end{array}
$$

$$
\begin{array}{l}
InitNodeOutSlot \triangleq [\\
\quad o\_state \quad \mapsto NodeOutSlotState\_Avail, \\
\quad o\_connect \quad \mapsto NULL, \\
\quad o\_destC \quad \mapsto NULL, \\
\quad o\_destH \quad \mapsto NULL, \\
\quad o\_nextSlot \quad \mapsto NULL,
\end{array}
$$

$o\_chunkseqn \mapsto FirstChunkseqn,$
$o\_chunkcrc \mapsto NULL,$
$o\_ixatchunk \mapsto 1,$
$o\_ixinchunk \mapsto 1,$
$o\_packet \mapsto \langle\rangle$
]

$Node \triangleq [$
$outq : [CellType \to Seq(Cell)],$
$iss : [SlotName \to NodeInSlot],$
$oss : [SlotName \to NodeOutSlot],$
$slot : SlotName$
]

$InitNode \triangleq [$
$outq \mapsto [t \in CellType \mapsto \langle\rangle],$
$iss \mapsto [sn \in SlotName \mapsto InitNodeInSlot],$
$oss \mapsto [sn \in SlotName \mapsto InitNodeOutSlot],$
$slot \mapsto FirstSlotName$
]

Set of received packets at this moment in time.

$Node\_ReceivedPackets(n) \triangleq$
LET
slots that have finished receiving a packet
$sns \triangleq \{sn \in SlotName : n.iss[sn].i\_state = NodeInSlotState\_End\}$

packet received in slot sn
$pkt(sn) \triangleq$ IF $BugRecvWrongPacket$ THEN CHOOSE $pkt0 \in Seq(Payload)$: TRUE ELSE $n.iss[sn].i\_packet$
IN
$\{pkt(sn) : sn \in sns\}$

Determine if local slot is available outgoing / incoming.

$Node\_IsAvailOutSlot(n, sn) \triangleq n.oss[sn].o\_state = NodeOutSlotState\_Avail$
$Node\_IsAvailInSlot(n, sn) \triangleq n.iss[sn].i\_state = NodeInSlotState\_Avail$

Set of available outgoing / incoming local slots.

$Node\_AvailOutSlots(n) \triangleq \{sn \in SlotName : Node\_IsAvailOutSlot(n, sn)\}$
$Node\_AvailInSlots(n) \triangleq \{sn \in SlotName : Node\_IsAvailInSlot(n, sn)\}$

Create a packet request.

$Node\_CreateRequest(n,\ sn,\ C,\ H,\ packet,\ connect) \triangleq$
CASE $TypeAssert(n,\ Node) \rightarrow$
CASE $TypeAssert(sn,\ SlotName) \rightarrow$
CASE $TypeAssert(C,\ Container) \rightarrow$
CASE $TypeAssert(H,\ Halfrow) \rightarrow$
CASE $TypeAssert(packet,\ Seq(Payload)) \rightarrow$
CASE $TypeAssert(connect,\ Connection) \rightarrow$

IF $n.oss[sn].o\_state \neq NodeOutSlotState\_Avail$
THEN $Assert(\text{FALSE},$ "protocol error") ELSE
$[n$ EXCEPT
$!.oss[sn].o\_state \quad = NodeOutSlotState\_SendSetupReq,$
$!.oss[sn].o\_connect = connect,$
$!.oss[sn].o\_destC \quad = C,$
$!.oss[sn].o\_destH \quad = H,$
$!.oss[sn].o\_packet \quad = packet$
$]$

Update a node output slot state for slot time.

$Node\_Oss\_SlotTime(n,\ sn) \triangleq$
LET
$oss \triangleq n.oss[sn]$

Send a setup request.

Sending a control cell is not logically tied to the slot time, but checking it and queueing the cell at this time makes multiplexing our attention easier.

$SendSetupReq \triangleq$
$[n$ EXCEPT
$!.outq[CellType\_Ctrl] = Append(@,\ [$
$typecode \mapsto CtrlCell\_Typecode\_SetupReq,$
$connect \mapsto oss.o\_connect,$
$DestC \mapsto oss.o\_destC,$
$DestH \mapsto oss.o\_destH,$
$prevSlot \mapsto sn$
$]),$
$!.oss[sn].o\_state = NodeOutSlotState\_WaitSetupAck,$
$!.oss[sn].o\_ixatchunk = 1,$

```
    !.oss[sn].o_ixinchunk = 0
  ]
```

Send a first mark.

```
SendFirstMark ≜
  [n EXCEPT
   !.outq[CellType_Data] = Append(@, [
                  typecode  ↦ DataCell_Typecode_FirstMark,
                  connect   ↦ oss.o_connect,
                  nextSlot  ↦ oss.o_nextSlot,
                  destC     ↦ oss.o_destC,
                  destH     ↦ oss.o_destH
                ]),
   !.oss[sn].o_state = NodeOutSlotState_SendChunkData,
   !.oss[sn].o_chunkcrc = InitCRC,
   !.oss[sn].o_ixatchunk = 1,
   !.oss[sn].o_ixinchunk = 0,
   !.oss[sn] =
     IF ∨ @.o_ixatchunk + @.o_ixinchunk > Len(@.o_packet)
        ∨ @.o_ixinchunk ≥ MaxChunkCells
      THEN [@ EXCEPT !.o_state = NodeOutSlotState_SendChunkMark]
      ELSE @
  ]
```

Send chunk data.

```
SendChunkData ≜
  LET
    ix ≜ oss.o_ixatchunk + oss.o_ixinchunk
    py ≜ oss.o_packet[ix]
  IN
  [n EXCEPT
   !.outq[CellType_Data] = Append(@, [
                  typecode  ↦ DataCell_Typecode_Normal,
                  connect   ↦ oss.o_connect,
                  nextSlot  ↦ oss.o_nextSlot,
                  payload   ↦ py
                ]),
   !.oss[sn].o_chunkcrc = ExtendCRC(@, py),
   !.oss[sn].o_ixinchunk = @ + 1,
   !.oss[sn] =
     IF ∨ @.o_ixatchunk + @.o_ixinchunk > Len(@.o_packet)
        ∨ @.o_ixinchunk ≥ MaxChunkCells
      THEN [@ EXCEPT !.o_state = NodeOutSlotState_SendChunkMark]
      ELSE @
  ]
```

Send chunk mark.

$$
\begin{array}{l}
SendChunkMark \triangleq \\
\quad [n \text{ EXCEPT} \\
\quad\ !.outq[CellType\_Data] = Append(@, [\\
\qquad\qquad typecode \mapsto DataCell\_Typecode\_ChunkMark, \\
\qquad\qquad connect \mapsto oss.o\_connect, \\
\qquad\qquad nextSlot \mapsto oss.o\_nextSlot, \\
\qquad\qquad chunkseqn \mapsto oss.o\_chunkseqn, \\
\qquad\qquad crc \mapsto oss.o\_chunkcrc \\
\qquad\quad]), \\
\quad\ !.oss[sn].o\_state = NodeOutSlotState\_WaitDataAck, \\
\quad\ !.oss[sn].o\_chunkcrc = NULL \\
\quad]
\end{array}
$$

Send end mark.

$$
\begin{array}{l}
SendEndMark \triangleq \\
\quad [n \text{ EXCEPT} \\
\quad\ !.outq[CellType\_Data] = Append(@, [\\
\qquad\qquad typecode \mapsto DataCell\_Typecode\_EndMark, \\
\qquad\qquad connect \mapsto oss.o\_connect, \\
\qquad\qquad nextSlot \mapsto oss.o\_nextSlot \\
\qquad\quad]), \\
\quad\ !.oss[sn].o\_state = NodeOutSlotState\_WaitTeardown \\
\quad]
\end{array}
$$

Recycle output slot.

$$
\begin{array}{l}
RecycleSlot \triangleq \\
\quad [n \text{ EXCEPT} \\
\quad\ !.oss[sn] = InitNodeOutSlot \\
\quad]
\end{array}
$$

IN

CASE

$$
\begin{array}{lll}
oss.o\_state = NodeOutSlotState\_Avail & \to n & \Box \\
oss.o\_state = NodeOutSlotState\_SendSetupReq & \to SendSetupReq & \Box \\
oss.o\_state = NodeOutSlotState\_WaitSetupAck & \to n & \Box \\
oss.o\_state = NodeOutSlotState\_SendFirstMark & \to SendFirstMark & \Box \\
oss.o\_state = NodeOutSlotState\_SendChunkData & \to SendChunkData & \Box \\
oss.o\_state = NodeOutSlotState\_SendChunkMark & \to SendChunkMark & \Box \\
oss.o\_state = NodeOutSlotState\_WaitDataAck & \to n & \Box \\
oss.o\_state = NodeOutSlotState\_SendEndMark & \to SendEndMark & \Box \\
oss.o\_state = NodeOutSlotState\_WaitTeardown & \to n & \Box \\
oss.o\_state = NodeOutSlotState\_Abort & \to RecycleSlot & \Box \\
oss.o\_state = NodeOutSlotState\_End & \to RecycleSlot &
\end{array}
$$

Update a node for receiving a setup ack control cell.

```
Node_RecvCtrlCell_SetupAck(n, cc) ≜
  LET
    sn    ≜ cc.prevSlot
    oss   ≜ n.oss[sn]

    RecvSetupAck ≜
      CASE Assert(cc.connect = oss.o_connect, "mismatch connection") →
      [n EXCEPT
       !.oss[sn].o_state = NodeOutSlotState_SendFirstMark,
       !.oss[sn].o_nextSlot = cc.nextSlot,
       !.oss[sn].o_chunkseqn = FirstChunkseqn
      ]
  IN
  CASE
  oss.o_state = NodeOutSlotState_Avail              → n              □
  oss.o_state = NodeOutSlotState_SendSetupReq       → n              □
  oss.o_state = NodeOutSlotState_WaitSetupAck       → RecvSetupAck   □
  oss.o_state = NodeOutSlotState_SendFirstMark      → n              □
  oss.o_state = NodeOutSlotState_SendChunkData      → n              □
  oss.o_state = NodeOutSlotState_SendChunkMark      → n              □
  oss.o_state = NodeOutSlotState_WaitDataAck        → n              □
  oss.o_state = NodeOutSlotState_SendEndMark        → n              □
  oss.o_state = NodeOutSlotState_WaitTeardown       → n              □
  oss.o_state = NodeOutSlotState_Abort              → n              □
  oss.o_state = NodeOutSlotState_End                → n
```

Update a node for receiving a teardown control cell.

```
Node_RecvCtrlCell_Teardown(n, cc) ≜
  LET
    sn    ≜ cc.prevSlot
    oss   ≜ n.oss[sn]

    DoAbort ≜
        CASE Assert(cc.connect = oss.o_connect, "mismatch connection") →
        [n EXCEPT !.oss[sn].o_state = NodeOutSlotState_Abort]

    DoEnd   ≜
        CASE Assert(cc.connect = oss.o_connect, "mismatch connection") →
        [n EXCEPT !.oss[sn].o_state = NodeOutSlotState_End]
  IN
  CASE
  oss.o_state = NodeOutSlotState_Avail              → n              □
  oss.o_state = NodeOutSlotState_SendSetupReq       → DoAbort        □
```

```
    oss.o_state = NodeOutSlotState_WaitSetupAck    → DoAbort   □
    oss.o_state = NodeOutSlotState_SendFirstMark   → DoAbort   □
    oss.o_state = NodeOutSlotState_SendChunkData   → DoAbort   □
    oss.o_state = NodeOutSlotState_SendChunkMark   → DoAbort   □
    oss.o_state = NodeOutSlotState_WaitDataAck     → DoAbort   □
    oss.o_state = NodeOutSlotState_SendEndMark     → DoAbort   □
    oss.o_state = NodeOutSlotState_WaitTeardown    → DoEnd     □
    oss.o_state = NodeOutSlotState_Abort           → n         □
    oss.o_state = NodeOutSlotState_End             → n
```

Update a node for receiving a data ack control cell.

```
Node_RecvCtrlCell_DataAck(n, cc) ≜
  LET
    sn    ≜ cc.prevSlot
    oss   ≜ n.oss[sn]

    RecvDataAck ≜
      CASE Assert(cc.connect = oss.o_connect, "mismatch connection") →
      IF cc.chunkseqn ≠ oss.o_chunkseqn THEN n ELSE
      Got the expected data ack, go on to the next chunk.
      [n EXCEPT
       !.oss[sn].o_state = NodeOutSlotState_SendChunkData,
       !.oss[sn].o_chunkseqn = IncrChunkseqn(@),
       !.oss[sn].o_chunkcrc = InitCRC,
       !.oss[sn].o_ixatchunk = @ + oss.o_ixinchunk,
       !.oss[sn].o_ixinchunk = 0,
       !.oss[sn] =
         IF @.o_ixatchunk + @.o_ixinchunk > Len(@.o_packet)
          THEN [@ EXCEPT !.o_state = NodeOutSlotState_SendEndMark]
          ELSE @
      ]
  IN
  CASE
    oss.o_state = NodeOutSlotState_Avail          → n           □
    oss.o_state = NodeOutSlotState_SendSetupReq   → n           □
    oss.o_state = NodeOutSlotState_WaitSetupAck   → n           □
    oss.o_state = NodeOutSlotState_SendFirstMark  → n           □
    oss.o_state = NodeOutSlotState_SendChunkData  → n           □
    oss.o_state = NodeOutSlotState_SendChunkMark  → n           □
    oss.o_state = NodeOutSlotState_WaitDataAck    → RecvDataAck □
    oss.o_state = NodeOutSlotState_SendEndMark    → n           □
    oss.o_state = NodeOutSlotState_WaitTeardown   → n           □
```

$oss.o\_state = NodeOutSlotState\_Abort \quad \rightarrow n \quad \Box$
$oss.o\_state = NodeOutSlotState\_End \quad \rightarrow n$

Update a node input state for slot sn.

$Node\_Iss\_SlotTime(n,\ sn) \triangleq$
LET
$iss \triangleq n.iss[sn]$

A convenient time to take the finished received packet away and reset the state of the connection.

$AtEnd \triangleq$
CASE IF $TraceRecvPacket$ THEN $PrintT(\langle$"recv pkt", $iss.i\_packet\rangle)$ ELSE TRUE $\rightarrow$
$[n$ EXCEPT $!.iss[sn] = InitNodeInSlot]$

IN
CASE
$iss.i\_state = NodeInSlotState\_Avail \quad \rightarrow n \quad \Box$
$iss.i\_state = NodeInSlotState\_WaitFirstMark \rightarrow n \quad \Box$
$iss.i\_state = NodeInSlotState\_WaitChunk \quad \rightarrow n \quad \Box$
$iss.i\_state = NodeInSlotState\_End \quad \rightarrow AtEnd$

Advance the slot in a node

$Node\_AdvanceSlot(n) \triangleq$
$[[n \mapsto n]$ EXCEPT
$!.n = Node\_Iss\_SlotTime(@,\ @.slot),$
$!.n = Node\_Oss\_SlotTime(@,\ @.slot),$
$!.n.slot = IncrSlotName[@]$
$].n$

Receive a setup request control cell into node n.

$Node\_RecvCtrlCell\_SetupReq(n,\ cc) \triangleq$
LET
$prevSlot \triangleq cc.prevSlot$

$sns \triangleq Node\_AvailInSlots(n)$
$sn \triangleq$ CHOOSE $sn1 \in sns : \forall\, sn2 \in sns : LeqSlotName[sn1,\ sn2]$

```
    iss      ≜ n.iss[sn]

    DoNak ≜ [n EXCEPT
              !.outq[CellType_Ctrl] = Append(@, [
                                        typecode ↦ CtrlCell_Typecode_Teardown,
                                        connect  ↦ cc.connect,
                                        prevSlot ↦ prevSlot,
                                        reason   ↦ "dest full"
                                      ])
            ]

    DoAck ≜ [n EXCEPT
              !.outq[CellType_Ctrl] = Append(@, [
                                        typecode ↦ CtrlCell_Typecode_SetupAck,
                                        connect  ↦ cc.connect,
                                        prevSlot ↦ prevSlot,
                                        nextSlot ↦ sn
                                      ]),
              !.iss[sn].i_state = NodeInSlotState_WaitFirstMark,
              !.iss[sn].i_connect = cc.connect,
              !.iss[sn].i_prevSlot = prevSlot,
              !.iss[sn].i_packet = ⟨⟩
            ]
  IN
  IF sns = {} THEN DoNak ELSE DoAck
```

Receive a first mark data cell into node n.

```
Node_RecvDataCell_FirstMark(n, dc) ≜
  LET
    sn    ≜ dc.nextSlot
    iss   ≜ n.iss[sn]

    If we are waiting for a first mark . . .
    ok_iss   ≜ [iss EXCEPT
                 !.i_state = NodeInSlotState_WaitChunk,
                 !.i_chunkseqn = FirstChunkseqn,
                 !.i_chunkcrc = InitCRC,
                 !.i_chunk = ⟨⟩,
                 !.i_packet = ⟨⟩
               ]
    ok_node ≜
      CASE Assert(dc.connect = iss.i_connect, "mismatch connection") →
      [n EXCEPT !.iss[sn] = ok_iss]
  IN
```

IF $iss.i\_state \neq NodeInSlotState\_WaitFirstMark$ THEN n ELSE ok_node

Receive a normal data cell into node n.

$Node\_RecvDataCell\_Normal(n,\ dc) \triangleq$
LET
$sn \triangleq dc.nextSlot$
$iss \triangleq n.iss[sn]$

If we are waiting for a normal data cell . . .

$ok\_iss \triangleq [iss$ EXCEPT
$!.i\_chunkcrc = ExtendCRC(@,\ dc.payload),$
$!.i\_chunk = Append(@,\ dc.payload)$
$]$
$ok\_node \triangleq$
CASE $Assert(dc.connect = iss.i\_connect,$ "mismatch connection") $\rightarrow$
$[n$ EXCEPT $!.iss[sn] = ok\_iss]$

IN
IF $iss.i\_state \neq NodeInSlotState\_WaitChunk$ THEN n ELSE ok_node

Receive a chunk mark data cell into node n.

$Node\_RecvDataCell\_ChunkMark(n,\ dc) \triangleq$
LET
$sn \triangleq dc.nextSlot$
$iss \triangleq n.iss[sn]$

Send data ack control cell corresponding to this chunk mark.

$SendAck(n0) \triangleq$
$[n0$ EXCEPT
$!.outq[CellType\_Ctrl] = Append(@, [$
$typecode \mapsto CtrlCell\_Typecode\_DataAck,$
$connect \mapsto iss.i\_connect,$
$prevSlot \mapsto iss.i\_prevSlot,$
$throttle \mapsto$ FALSE,
$chunkseqn \mapsto dc.chunkseqn$
$])$
$]$

Accept the current chunk. Append it to the packet being received and increment the chunk sequence number.

$AcceptChunk(n0) \triangleq$
$[n0$ EXCEPT
$!.iss[sn].i\_packet = @ \circ iss.i\_chunk,$

```
      !.iss[sn].i_chunkseqn = IncrChunkseqn(@)
    ]

  Start receiving the next chunk.
  StartChunk(n0) ≜
    [n0 EXCEPT
      !.iss[sn].i_chunkcrc = IF BugRecvWrongCRC THEN @ ELSE
                             InitCRC,
      !.iss[sn].i_chunk = ⟨⟩
    ]
IN
IF iss.i_state ≠ NodeInSlotState_WaitChunk THEN n ELSE

CASE
  (iss.i_chunkseqn = dc.chunkseqn ∧ iss.i_chunkcrc ≠ dc.crc)
   ⇒ PrintT("crc error")
→

CASE Assert(dc.connect = iss.i_connect, "mismatch connection") →

IF iss.i_chunkseqn = dc.chunkseqn ∧ iss.i_chunkcrc = dc.crc
 THEN
  Chunk mark has the sequence number of the next chunk we are expecting and the crc is correct.
  [[n ↦ n] EXCEPT
   !.n = SendAck(@),
   !.n = AcceptChunk(@),
   !.n = StartChunk(@)
  ].n
 ELSE
IF iss.i_chunkseqn = IncrChunkseqn(dc.chunkseqn)
 THEN
  Chunk mark repeats the sequence number of the last chunk we accepted.
  [[n ↦ n] EXCEPT
   !.n = SendAck(@),
   !.n = StartChunk(@)
  ].n
 ELSE
  Something else.
  [[n ↦ n] EXCEPT
   !.n = StartChunk(@)
  ].n


Receive an end mark data cell into node n.
```

```
Node_RecvDataCell_EndMark(n, dc) ≜
  LET
    sn     ≜ dc.nextSlot
    iss    ≜ n.iss[sn]

    endcc  ≜ [typecode  ↦ CtrlCell_Typecode_Teardown,
              connect   ↦ iss.i_connect,
              prevSlot  ↦ iss.i_prevSlot,
              reason    ↦ "end mark"
             ]

    If we accept the end mark . . .
    ok_iss   ≜ [iss EXCEPT
                  !.i_state  = NodeInSlotState_End,
                  !.i_chunk = ⟨⟩,
                  !.i_packet = @
                ]
    ok_node  ≜
      CASE Assert(dc.connect = iss.i_connect, "mismatch connection") →
      [n EXCEPT
       !.outq[CellType_Ctrl] = Append(@, endcc),
       !.iss[sn] = ok_iss
      ]
  IN
  IF iss.i_state ≠ NodeInSlotState_WaitChunk THEN n ELSE ok_node
```

Receive a data cell from the link into node n.

```
Node_RecvDataCell(n, dc) ≜
  CASE
  DataCell_Is_FirstMark(dc)  → Node_RecvDataCell_FirstMark(n, dc)  □
  DataCell_Is_Normal(dc)     → Node_RecvDataCell_Normal(n, dc)     □
  DataCell_Is_ChunkMark(dc)  → Node_RecvDataCell_ChunkMark(n, dc)  □
  DataCell_Is_EndMark(dc)    → Node_RecvDataCell_EndMark(n, dc)    □
  DataCell_Is_Null(dc)       → n
```

Receive a control cell from the link into node n.

```
Node_RecvCtrlCell(n, cc) ≜
  CASE
  CtrlCell_Is_SetupReq(cc)  → Node_RecvCtrlCell_SetupReq(n, cc)  □
  CtrlCell_Is_SetupAck(cc)  → Node_RecvCtrlCell_SetupAck(n, cc)  □
  CtrlCell_Is_DataAck(cc)   → Node_RecvCtrlCell_DataAck(n, cc)   □
  CtrlCell_Is_Teardown(cc)  → Node_RecvCtrlCell_Teardown(n, cc)  □
  CtrlCell_Is_Null(cc)      → n
```

Interface to link.

Receive a cell of the indicated type from the link into node n.

$Node\_RecvCell(n,\ t,\ cell)\ \triangleq$
LET
$ready\ \triangleq$ TRUE
$update\ \triangleq$
CASE
$t = CellType\_Data \rightarrow Node\_RecvDataCell(n,\ cell)\ \Box$
$t = CellType\_Ctrl\ \rightarrow Node\_RecvCtrlCell(n,\ cell)$
IN
IF $\neg ready$ THEN $[ready \mapsto ready]$ ELSE
[
$ready\ \mapsto ready,$
$update \mapsto update$
]

Extract a cell of the indicated type onto the link from node n.

$Node\_TakeCell(n,\ t)\ \triangleq$
LET
$ready\ \triangleq\ n.outq[t] \neq \langle\rangle$
$cell\ \triangleq\ Head(n.outq[t])$
$update\ \triangleq\ [n$ EXCEPT $!.outq[t] = Tail(@)]$
IN
IF $\neg ready$ THEN $[ready \mapsto ready]$ ELSE
[
$ready\ \mapsto ready,$
$cell\ \mapsto cell,$
$update \mapsto update$
]

12 Ring message

MODULE $DCNL1RingMsg$

EXTENDS
$DCNL1Param,$

$DCNL1SlotName$,
$DCNL1Chunkseqn$,
$DCNL1PortName$

Messages on the ring.

In the hardware, each ring message except $SetupReq$ fits in one 40 bit word. The $SetupReq$ message takes two 40 bit words. However, we ignore this difference.

$RingMsg\_Typecode\_RedSetupReq \triangleq$ "RingMsg_RedSetupReq" up port to down port
$RingMsg\_Typecode\_RedSetupAns \triangleq$ "RingMsg_RedSetupAns" down port to up port
$RingMsg\_Typecode\_RedSetupXbar \triangleq$ "RingMsg_RedSetupXbar" up port to up port
$RingMsg\_Typecode\_RedNull \triangleq$ "RingMsg_RedNull"

$RingMsg\_Typecode\_GreenTeardown \triangleq$ "RingMsg_GreenTeardown" down port to up port
$RingMsg\_Typecode\_GreenGCClear \triangleq$ "RingMsg_GreenGCClear" down port to up port
$RingMsg\_Typecode\_GreenDiscard \triangleq$ "RingMsg_GreenDiscard" down port to up port
$RingMsg\_Typecode\_GreenDataAck \triangleq$ "RingMsg_GreenDataAck" down port to up port
$RingMsg\_Typecode\_GreenRestart \triangleq$ "RingMsg_GreenRestart" down port to up port
$RingMsg\_Typecode\_GreenStop \triangleq$ "RingMsg_GreenStop" down port to up port
$RingMsg\_Typecode\_GreenNull \triangleq$ "RingMsg_GreenNull"

$RingMsg\_RedSetupReq \triangleq [$
  $typecode$: $\{RingMsg\_Typecode\_RedSetupReq\}$,
  $connect$: $Connection$,
  $DestC$: $Container$,
  $DestH$: $Halfrow$,
  $prevSlot$: $SlotName$,
  $upPort$: $PortName$,
  $downPort$: $PortName$,
  $subframe$: $Subframe$,
  $occupied$: $[Slottina \rightarrow \text{BOOLEAN}]$
  this $L1$ model terminates at the halfrow, hence there is no $DestN$
  this $L1$ model omits the $L0$ switches, hence there is no $FirstL1$
  this $L1$ model omits VLB routing
]

$RingMsg\_RedSetupAns \triangleq [$
  The hardware encodes a setup answer ring message as a setup request with processed $=$ TRUE. For the specification it is cleaner to use a separate ring message for the setup answer.
  $typecode$: $\{RingMsg\_Typecode\_RedSetupAns\}$,
  $connect$: $Connection$,
  $DestC$: $Container$,
  $DestH$: $Halfrow$,
  $prevSlot$: $SlotName$,
  $upPort$: $PortName$,
  $downPort$: $PortName$,

$subframe \quad : Subframe,$
$occupied \quad : [Slottina \rightarrow \text{BOOLEAN}\,]$
]

$RingMsg\_RedSetupXbar \triangleq [$

The hardware encodes a setup xbar ring message as a setup request with processed $=$ TRUE and $xDone =$ TRUE. For the specification it is cleaner to use a separate ring message for the setup xbar.

$typecode \quad : \{RingMsg\_Typecode\_RedSetupXbar\},$
$connect \quad : Connection,$
$DestC \quad : Container,$
$DestH \quad : Halfrow,$
$prevSlot \quad : SlotName,$
$upPort \quad : PortName,$
$downPort \quad : PortName,$
$subframe \quad : Subframe,$
$occupied \quad : [Slottina \rightarrow \text{BOOLEAN}\,]$
]

$RingMsg\_RedNull \triangleq [$
$typecode \quad : \{RingMsg\_Typecode\_RedNull\}$
]

$RingMsg\_GreenTeardown \triangleq [$
$typecode : \{RingMsg\_Typecode\_GreenTeardown\},$
$connect \quad : Connection,$
$prevSlot : SlotName,$
$localSlot : SlotName,$
$upPort \quad : PortName,$
$reason \quad : Reason$
]

$RingMsg\_GreenGCClear \triangleq [$
$typecode : \{RingMsg\_Typecode\_GreenGCClear\},$
$connect \quad : Connection,$
$upPort \quad : PortName,$
$localSlot : SlotName$
]

$RingMsg\_GreenDiscard \triangleq [$
$typecode \quad : \{RingMsg\_Typecode\_GreenDiscard\},$
$connect \quad : Connection,$
$prevSlot \quad : SlotName,$
$upPort \quad : PortName$
]

$RingMsg\_GreenDataAck \triangleq [$
$typecode \quad : \{RingMsg\_Typecode\_GreenDataAck\},$

$$
\begin{array}{l}
\quad connect \quad : Connection, \\
\quad prevSlot \quad : SlotName, \\
\quad upPort \quad : PortName, \\
\quad chunkseqn : Chunkseqn, \\
\quad throttle \quad : Throttle \\
]
\end{array}
$$

$$
\begin{array}{l}
RingMsg\_GreenRestart \triangleq [\\
\quad typecode \ : \{RingMsg\_Typecode\_GreenRestart\}, \\
\quad connect \ : Connection, \\
\quad prevSlot \ : SlotName, \\
\quad upPort \ : PortName \\
]
\end{array}
$$

$$
\begin{array}{l}
RingMsg\_GreenStop \triangleq [\\
\quad typecode \ : \{RingMsg\_Typecode\_GreenStop\}, \\
\quad connect \ : Connection, \\
\quad prevSlot \ : SlotName, \\
\quad upPort \ : PortName \\
]
\end{array}
$$

$$
\begin{array}{l}
RingMsg\_GreenNull \triangleq [\\
\quad typecode \ : \{RingMsg\_Typecode\_GreenNull\} \\
]
\end{array}
$$

$$
\begin{array}{l}
RingMsg \triangleq \{\} \\
\quad \cup RingMsg\_RedSetupReq \\
\quad \cup RingMsg\_RedSetupAns \\
\quad \cup RingMsg\_RedSetupXbar \\
\quad \cup RingMsg\_RedNull \\
\quad \cup RingMsg\_GreenTeardown \\
\quad \cup RingMsg\_GreenGCClear \\
\quad \cup RingMsg\_GreenDiscard \\
\quad \cup RingMsg\_GreenDataAck \\
\quad \cup RingMsg\_GreenRestart \\
\quad \cup RingMsg\_GreenStop \\
\quad \cup RingMsg\_GreenNull
\end{array}
$$

$$
\begin{array}{l}
RedNullRingMsg \quad \triangleq [typecode \mapsto RingMsg\_Typecode\_RedNull] \\
GreenNullRingMsg \triangleq [typecode \mapsto RingMsg\_Typecode\_GreenNull]
\end{array}
$$

$$
\begin{array}{ll}
RingMsg\_Is\_RedSetupReq(rm) & \triangleq rm.typecode = RingMsg\_Typecode\_RedSetupReq \\
RingMsg\_Is\_RedSetupAns(rm) & \triangleq rm.typecode = RingMsg\_Typecode\_RedSetupAns \\
RingMsg\_Is\_RedSetupXbar(rm) & \triangleq rm.typecode = RingMsg\_Typecode\_RedSetupXbar \\
RingMsg\_Is\_RedNull(rm) & \triangleq rm.typecode = RingMsg\_Typecode\_RedNull \\
RingMsg\_Is\_GreenTeardown(rm) & \triangleq rm.typecode = RingMsg\_Typecode\_GreenTeardown
\end{array}
$$

$RingMsg\_Is\_GreenGCClear(rm) \triangleq rm.typecode = RingMsg\_Typecode\_GreenGCClear$
$RingMsg\_Is\_GreenDiscard(rm) \triangleq rm.typecode = RingMsg\_Typecode\_GreenDiscard$
$RingMsg\_Is\_GreenDataAck(rm) \triangleq rm.typecode = RingMsg\_Typecode\_GreenDataAck$
$RingMsg\_Is\_GreenRestart(rm) \triangleq rm.typecode = RingMsg\_Typecode\_GreenRestart$
$RingMsg\_Is\_GreenStop(rm) \triangleq rm.typecode = RingMsg\_Typecode\_GreenStop$
$RingMsg\_Is\_GreenNull(rm) \triangleq rm.typecode = RingMsg\_Typecode\_GreenNull$

The setup request, setup answer, and setup xbar messages all use a *subframe* number and occupied map that need to be evaluated in the same way, to see if there is a *localSlot* available and if so, what the least such slot is. Here is how we compute this.

$RingMsg\_Eval\_Occupied(subframe,\ occupied) \triangleq$
LET
$sns \triangleq \{sn \in SlotName :$
$\land\ sn.subframe = subframe$
$\land\ \neg occupied[sn.slottina]$
$\}$
$success \triangleq sns \neq \{\}$
$localSlot \triangleq$ CHOOSE $sn1 \in sns : \forall\, sn2 \in sns : LeqSlotName[sn1,\ sn2]$
IN
IF $\neg success$ THEN $[success \mapsto success]$ ELSE
$[$
$success \mapsto success,$
$localSlot \mapsto localSlot$
$]$

13 Port name

MODULE $DCNL1PortName$

EXTENDS $DCNL1Param$

Port names inside an $L1$ switch.

$PortName\_Typecode\_Halfrow \triangleq$ "PortName_Halfrow"
$PortName\_Typecode\_Container \triangleq$ "PortName_Container"

$PortName\_Halfrow \triangleq [$
$typecode : \{PortName\_Typecode\_Halfrow\},$

$to \quad : Halfrow$
]

$PortName\_Container \triangleq [$
$typecode : \{PortName\_Typecode\_Container\},$
$to \quad : Container$
]

$PortName \triangleq \{\}$
$\cup\ PortName\_Halfrow$
$\cup\ PortName\_Container$

$PortName\_Is\_Halfrow(pn) \triangleq pn.typecode = PortName\_Typecode\_Halfrow$
$PortName\_Is\_Container(pn) \triangleq pn.typecode = PortName\_Typecode\_Container$

Define an arbitrary ordering of port names.

$FirstPortName \triangleq ArbSeq(PortName).FirstName$
$IncrPortName \triangleq ArbSeq(PortName).IncrName$
$DecrPortName \triangleq ArbSeq(PortName).DecrName$
$LeqPortName \triangleq ArbSeq(PortName).LeqName$

14 Port

MODULE $DCNL1Port$

EXTENDS
$DCNL1Useful$,
$DCNL1Param$,
$DCNL1Cell$,
$DCNL1CellType$,
$DCNL1DataCell$,
$DCNL1CtrlCell$,
$DCNL1RingMsg$,
$DCNL1SwitchName$,
$DCNL1CtrlQueueType$

Definition of the state maintained in a port.

The input state per slot of a port.

$PortInSlot \triangleq [$
i_assign : BOOLEAN , the connection is assigned
$i\_connect$: $Nullable(Connection)$, connection identifier (spec only)
i_buf : $Seq(DataCell)$ arriving data cells
]

$InitPortInSlot \triangleq [$
$i\_assign \mapsto$ FALSE,
$i\_connect \mapsto NULL$,
$i\_buf \mapsto \langle\rangle$
]

The output state per slot of a port.

Some of the output related fields are declared as $Nullable$ so that they can accept a $NULL$ value. When the field is set to $NULL$ it should never be read. Using a $NULL$ value where it is not expected should lead to a type error during model checking.

$PortOutSlot \triangleq [$
o_assign : BOOLEAN , the connection is assigned
$o\_connect$: $Nullable(Connection)$, connection identifier (spec only)
$o\_nextset$: BOOLEAN , the next slot is set up
$o\_prevSlot$: $Nullable(SlotName)$, upstream switch's *localSlot*
$o\_nextSlot$: $Nullable(SlotName)$, downstream switch's *localSlot*
o_upPort : $Nullable(PortName)$ upstream port in this switch
]

$InitPortOutSlot \triangleq [$
$o\_assign \mapsto$ FALSE,
$o\_connect \mapsto NULL$,
$o\_nextset \mapsto$ FALSE,
$o\_prevSlot \mapsto NULL$,
$o\_nextSlot \mapsto NULL$,
$o\_upPort \mapsto NULL$
]

State of a port.

$Port \triangleq [$
$switchname$: $SwitchName$,
$portname$: $PortName$,

$link\_outq$: $[CellType \rightarrow Seq(Cell)]$, departing cells by type

$locked$: $[Subframe \rightarrow$ BOOLEAN $]$, which subframes are locked
$locked\_connect$: $[Subframe \rightarrow Nullable(Connection)]$, for what connection

$ccinq$: $[CtrlQueueType \to Seq(CtrlCell)]$, ctrl cells in from link

iss : $[SlotName \to PortInSlot]$,
oss : $[SlotName \to PortOutSlot]$,

$toxbar\_in$: $DataCell$ most recent cell sent into crossbar
]

$InitPort(swn,\ pn) \triangleq$ [
$switchname$ $\mapsto swn$,
$portname$ $\mapsto pn$,

$link\_outq$ $\mapsto [t \in CellType \mapsto \langle\rangle]$,

$locked$ $\mapsto [sf \in Subframe \mapsto \text{FALSE}]$,
$locked\_connect$ $\mapsto [sf \in Subframe \mapsto NULL]$,

$ccinq$ $\mapsto [t \in CtrlQueueType \mapsto \langle\rangle]$,

iss $\mapsto [sn \in SlotName \mapsto InitPortInSlot]$,
oss $\mapsto [sn \in SlotName \mapsto InitPortOutSlot]$,

$toxbar\_in$ $\mapsto NullDataCell$
]

Get the container for port p.

$Port\_MyContainer(p) \triangleq p.switchname.container$

Interface to link.

Receive a data cell from the link into port p. Compute the updated state of the port.

$Port\_RecvDataCell(p,\ dc) \triangleq$
CASE $TypeAssert(dc,\ DataCell) \to$

IF $dc.typecode = DataCell\_Typecode\_Null$
THEN
ignore arriving null data cell
p
ELSE
LET
$sn \triangleq dc.nextSlot$
$iss \triangleq p.iss[sn]$
IN

```
IF ¬iss.i_assign THEN
  ignore arriving data cell whose next slot is not assigned for a connection
  p
ELSE
  enqueue arriving data cell based on its next slot
  CASE Assert(dc.connect = iss.i_connect,
              ⟨"connection mismatch", dc, p.portname, iss⟩) →
  [p EXCEPT !.iss[sn].i_buf = Append(@, dc)]
```

Receive a control cell from the link into port p. Compute the updated state of the port.

```
Port_RecvCtrlCell(p, cc) ≜
  LET
    pack ≜ LET
             sn ≜ cc.prevSlot
             oss ≜ p.oss[sn]
           IN
           CASE Assert(cc.connect = oss.o_connect, "connection mismatch") →
           IF ¬oss.o_assign THEN p ELSE
           [p EXCEPT
            !.oss[sn].o_nextSlot = cc.nextSlot,
            !.oss[sn].o_nextset = TRUE
           ]
    pinq(t) ≜ [p EXCEPT !.ccinq[t] = Append(@, cc)]
  IN
  CASE
  CtrlCell_Is_SetupReq(cc)  → pinq(CtrlQueueType_Req) □
  CtrlCell_Is_SetupAck(cc)  → pack                    □
  CtrlCell_Is_Teardown(cc)  → pinq(CtrlQueueType_Ups) □
  CtrlCell_Is_Discard(cc)   → pinq(CtrlQueueType_Ups) □
  CtrlCell_Is_DataAck(cc)   → pinq(CtrlQueueType_Ups) □
  CtrlCell_Is_Restart(cc)   → pinq(CtrlQueueType_Ups) □
  CtrlCell_Is_Stop(cc)      → pinq(CtrlQueueType_Ups) □
  CtrlCell_Is_Null(cc)      → p
```

Receive a cell of the indicated type.

```
Port_RecvCell(p, t, cell) ≜
  LET
    ready ≜ TRUE
    update ≜
      CASE
      t = CellType_Data → Port_RecvDataCell(p, cell) □
```

```
      t = CellType_Ctrl  → Port_RecvCtrlCell(p, cell)
  IN
  IF ¬ready THEN [ready ↦ ready] ELSE
  [
    ready  ↦ ready,
    update ↦ update
  ]
```

Extract a cell of the indicated type onto the link from port p.

```
Port_TakeCell(p, t) ≜
  LET
    ready  ≜ p.link_outq[t] ≠ ⟨⟩
    cell   ≜ Head(p.link_outq[t])
    update ≜ [p EXCEPT !.link_outq[t] = Tail(@)]
  IN
  IF ¬ready THEN [ready ↦ ready] ELSE
  [
    ready  ↦ ready,
    cell   ↦ cell,
    update ↦ update
  ]
```

Send data cell dc from port p. Compute the updated state of the port.

Each port sends one data cell per slot time. In the hardware, this feeds into the framer which constructs frames for transmission on the link.

We do not model link frames in the specification, so we just queue data cells for transmission on the link.

```
Port_SendDataCell(p, dc) ≜
  CASE TypeAssert(dc, DataCell) →

  [p EXCEPT !.link_outq[CellType_Data] = Append(@, dc)]
```

Send ctrl cell cc from port p. Compute the updated state of the port.

In the hardware, this feeds into a queue which the framer draws upon to construct frames for transmission on the link. In the hardware, there are 128 data cells and 64 control cells in each frame.

We do not model link frames in the specification, so we just queue control cells for transmission on the link.

```
Port_SendCtrlCell(p, cc) ≜
  CASE TypeAssert(cc, CtrlCell) →
```

$[p \text{ EXCEPT } !.link\_outq[CellType\_Ctrl] = Append(@,\ cc)]$

Interface to crossbar.

Send a data cell to the crossbar from port p in slot name sn. Compute a record with the following fields:

$p =$ the updated state of the port

$dc =$ the data cell sent into the crossbar

$Port\_Xbar\_TakeSlot(p,\ sn) \triangleq$
  LET
    $dc \triangleq HeadElse(p.iss[sn].i\_buf,\ NullDataCell)$
  IN
  [
    $p \mapsto [p \text{ EXCEPT}$
          $!.toxbar\_in = dc,$
          $!.iss[sn].i\_buf = TailElse(@)$
        $],$
    $dc \mapsto dc$
  ]

Receive a data cell from the crossbar into port p during slot name sn. Compute the updated state of the port.

$Port\_Xbar\_RecvSlot(p,\ sn,\ dc) \triangleq$
  LET
    $oss \triangleq p.oss[sn]$
  IN
  IF $dc.typecode = DataCell\_Typecode\_Null$ THEN
    a null data cell coming out of the crossbar – ignore it
    p
  ELSE
  IF
    $\lor \neg oss.o\_assign$
    $\lor \neg oss.o\_nextset$
  THEN
    this slot is not assigned or we do not yet know the next slot – ignore the data cell
    p
  ELSE

this slot is connected – twiddle the $nextSlot$ in the data cell and send it

$Port\_SendDataCell(p,\ [dc\ \text{EXCEPT}\ !.nextSlot = oss.o\_nextSlot])$

15 Gare name

MODULE $DCNL1GareName$

EXTENDS
$DCNL1Param$,
$DCNL1PortName$

Index set and assignments of the port gares in an $L1$ switch.

CONSTANT $MpxPortGare$ how many ports are multiplexed to each port gare

Construct a set of partitions of the set of port names. The partitions are disjoint subsets that when combined include all port names. As many partitions as possible must be of size $MpxPortGare$.

$PortNamePartition \triangleq$
LET
RECURSIVE $Recur(\_,\ \_,\ \_)$

$Recur(parts,\ mkpart,\ pns) \triangleq$
IF
$\lor\ Cardinality(mkpart) = MpxPortGare$ subset is required size
$\lor\ mkpart \neq \{\} \land pns = \{\}$ non-empty subset and done
THEN
Include $\{mkpart\}$ as a subset in the partition.
$Recur(parts \cup \{mkpart\},\ \{\},\ pns)$
ELSE
IF $pns = \{\}$
THEN
Construction of the partition is complete.
$parts$
ELSE
Take an arbitrary port name from pns and include it in the subset being constructed.
LET
$pn \triangleq$ CHOOSE $pn \in pns$: TRUE
IN
$Recur(parts,\ mkpart \cup \{pn\},\ pns \setminus \{pn\})$
IN

$Recur(\{\}, \{\}, PortName)$

Gare names in an $L1$ switch.

$$
\begin{array}{l}
GareName \triangleq [\\
\quad part \quad : PortNamePartition \\
]
\end{array}
$$

$$
\begin{array}{lll}
FirstGareName & \triangleq & ArbSeq(GareName).FirstName \\
IncrGareName & \triangleq & ArbSeq(GareName).IncrName \\
DecrGareName & \triangleq & ArbSeq(GareName).DecrName \\
GareIndexForName & \triangleq & ArbSeq(GareName).IndexForName
\end{array}
$$

Gare name assigned to a given port name.

$$
\begin{array}{l}
GareName\_ForPortName \triangleq \\
\quad [pn \in PortName \mapsto \text{CHOOSE } gn \in GareName : pn \in gn.part]
\end{array}
$$

16 Switch name

MODULE $DCNL1SwitchName$

EXTENDS $DCNL1Param$

Switch names.

$$
\begin{array}{l}
SwitchName \triangleq [\\
\quad container : Container \\
]
\end{array}
$$

$$
\begin{array}{l}
SwitchName\_ForContainer(c) \triangleq [\\
\quad container \mapsto c \\
]
\end{array}
$$

17 Gare

MODULE $DCNL1Gare$

EXTENDS
$DCNL1Param$,
$DCNL1GareName$,
$DCNL1Port$,
$DCNL1RingMsg$

Gare.

$$
\begin{array}{l}
Gare \triangleq [\\
\quad garename \ : GareName, \\
\quad ring\_out \ : RingMsg, \\
\\
\quad port \ : [PortName \to Nullable(Port)], \\
\quad priopn \ : [CtrlQueueType \to PortName] \\
]
\end{array}
$$

$$
\begin{array}{l}
InitGare(swn,\ gn) \triangleq [\\
\quad garename \ \mapsto gn, \\
\quad ring\_out \ \mapsto \text{LET } x \triangleq GareIndexForName[gn]\%2 \text{IN} \\
\qquad\qquad \text{CASE} \\
\qquad\qquad x = 0 \to GreenNullRingMsg \Box \\
\qquad\qquad x = 1 \to RedNullRingMsg, \\
\\
\quad port \ \mapsto [pn \in PortName \mapsto \\
\qquad\qquad \text{IF } pn \notin gn.part \text{ THEN } NULL \text{ ELSE } InitPort(swn,\ pn) \\
\qquad\qquad], \\
\quad priopn \ \mapsto [t \in CtrlQueueType \mapsto FirstPortName] \\
]
\end{array}
$$

Set of port names assigned to gare g.

$$
\begin{array}{l}
Gare\_AssignedPortNames(g) \triangleq \\
\quad g.garename.part
\end{array}
$$

Update a gare to send a ring message.

```
Gare_SendRingMsg(g, rm) ≜
  CASE TypeAssert(g, Gare) →
  CASE TypeAssert(rm, RingMsg) →

  [g EXCEPT !.ring_out = rm]
```

Process a request control cell in port name pn at gare g. Compute the updated gare.

```
Gare_Work_PortRequest(g, pn) ≜
  LET
    cc ≜ Head(g.port[pn].ccinq[CtrlQueueType_Req])
    p  ≜ [g.port[pn] EXCEPT !.ccinq[CtrlQueueType_Req] = Tail(@)]

    Exec(downPort, DestC, DestH, prevSlot) ≜
      LET
        set of unlocked subframes for port p
        usfs ≜ {sf ∈ Subframe : ¬p.locked[sf]}

        number of free slots in subframe sf
        cntf(sf) ≜ Cardinality({sn ∈ SlotName :
                                  ∧ sn.subframe = sf
                                  ∧ ¬p.iss[sn].i_assign})

        unlocked subframe with most number of free slots
        subframe ≜ CHOOSE sf1 ∈ usfs :
                   ∀ sf2 ∈ usfs :
                   cntf(sf1) ≥ cntf(sf2)

        map of which slottinas are currently occupied in that subframe
        occupied ≜ [sti ∈ Slottina ↦
                     LET
                       sn ≜ [subframe ↦ subframe, slottina ↦ sti]
                     IN
                     p.iss[sn].i_assign
                    ]

        launch the setup request ring message
        SendG ≜ [Gare_SendRingMsg(g, [
                   typecode  ↦ RingMsg_Typecode_RedSetupReq,
                   connect   ↦ cc.connect,
                   DestC     ↦ DestC,
```

```
                    DestH    ↦ DestH,
                    prevSlot ↦ prevSlot,
                    upPort   ↦ pn,
                    downPort ↦ downPort,
                    subframe ↦ subframe,
                    occupied ↦ occupied
                  ]) EXCEPT

                  !.priopn[CtrlQueueType_Req] = IncrPortName[pn],    rotate the priority

                  !.port[pn] = [p EXCEPT
                    !.locked[subframe] = TRUE,                  lock the subframe
                    !.locked_connect[subframe] = cc.connect     for what connection
                  ]
                 ]

  the request cannot be serviced now
  WaitG ≜ g

  fail this connection attempt
  FailG ≜ [g EXCEPT

                  !.priopn[CtrlQueueType_Req] = IncrPortName[pn],    rotate the priority

                  !.port[pn] = Port_SendCtrlCell(p, [
                    typecode ↦ CtrlCell_Typecode_Teardown,
                    connect  ↦ cc.connect,
                    prevSlot ↦ prevSlot,
                    reason   ↦ "full"
                  ])
                 ]
  IN
  IF ∀ sf ∈ Subframe : cntf(sf) = 0 THEN FailG ELSE
  IF usfs ≠ {} ∧ cntf(subframe) > 0 THEN SendG ELSE
  WaitG



downPortToC(DestC) ≜ [
  typecode ↦ PortName_Typecode_Container,
  to       ↦ DestC
]

downPortToH(DestH) ≜ [
  typecode ↦ PortName_Typecode_Halfrow,
  to       ↦ DestH
]
```

```
IN

IF cc.DestC ≠ Port_MyContainer(p)
 THEN
  make a setup request to connect to container DestC
  Exec(downPortToC(cc.DestC), cc.DestC, cc.DestH, cc.prevSlot)
 ELSE
  make a setup request to connect to halfrow DestH
  Exec(downPortToH(cc.DestH), cc.DestC, cc.DestH, cc.prevSlot)
```

Process an upstream control cell in port name pn at gare g. Compute the updated gare.

```
Gare_Work_PortUpstream(g, pn) ≜
 LET
   cc ≜ Head(g.port[pn].ccinq[CtrlQueueType_Ups])
   p  ≜ [g.port[pn] EXCEPT !.ccinq[CtrlQueueType_Ups] = Tail(@)]

   sn ≜ cc.prevSlot
   oss ≜ p.oss[sn]

   UsePermitG ≜ [g EXCEPT
                  !.priopn[CtrlQueueType_Ups] = IncrPortName[pn],
                  !.port[pn] = p
                ]

   UpstreamG(rm) ≜ Gare_SendRingMsg(UsePermitG, rm)

   matchconn ≜ Assert(cc.connect = oss.o_connect,
                      ⟨"connection mismatch", cc, pn, oss⟩)

   TeardownG ≜
     CASE matchconn →
     IF ¬oss.o_assign THEN UsePermitG ELSE
     [UpstreamG([
          typecode ↦ RingMsg_Typecode_GreenTeardown,
          connect  ↦ oss.o_connect,
          localSlot ↦ sn,
          prevSlot ↦ oss.o_prevSlot,
          upPort   ↦ oss.o_upPort,
          reason   ↦ cc.reason
        ]) EXCEPT
      !.port[pn].oss[sn] = InitPortOutSlot
     ]
```

```
  DiscardG ≜
    CASE matchconn →
    IF ¬oss.o_assign THEN UsePermitG ELSE
    UpstreamG([
          typecode ↦ RingMsg_Typecode_GreenDiscard,
          connect  ↦ oss.o_connect,
          prevSlot ↦ oss.o_prevSlot,
          upPort   ↦ oss.o_upPort
        ])

  DataAckG ≜
    CASE matchconn →
    IF ¬oss.o_assign THEN UsePermitG ELSE
    UpstreamG([
          typecode   ↦ RingMsg_Typecode_GreenDataAck,
          connect    ↦ oss.o_connect,
          prevSlot   ↦ oss.o_prevSlot,
          upPort     ↦ oss.o_upPort,
          chunkseqn  ↦ cc.chunkseqn,
          throttle   ↦ cc.throttle
        ])

  RestartG ≜
    CASE matchconn →
    IF ¬oss.o_assign THEN UsePermitG ELSE
    UpstreamG([
          typecode ↦ RingMsg_Typecode_GreenRestart,
          connect  ↦ oss.o_connect,
          prevSlot ↦ oss.o_prevSlot,
          upPort   ↦ oss.o_upPort
        ])

  StopG ≜
    CASE matchconn →
    IF ¬oss.o_assign THEN UsePermitG ELSE
    UpstreamG([
          typecode ↦ RingMsg_Typecode_GreenStop,
          connect  ↦ oss.o_connect,
          prevSlot ↦ oss.o_prevSlot,
          upPort   ↦ oss.o_upPort
        ])
IN
CASE
CtrlCell_Is_Teardown(cc) → TeardownG□
CtrlCell_Is_Discard(cc)  → DiscardG  □
CtrlCell_Is_DataAck(cc)  → DataAckG  □
CtrlCell_Is_Restart(cc)  → RestartG  □
```

$CtrlCell\_Is\_Stop(cc) \quad \rightarrow StopG$

Process a control cell from the indicated type queue of one of our assigned input ports, and generate a ring message. Compute the updated gare.

$Gare\_Work(g,\ t) \triangleq$
LET
Given a set of port names, find the one that is least higher than $priopn$. Compute a record with fields $ok =$ there is a highest priority port name, $pn =$ that port name.
$Prio(pns,\ priopn) \triangleq$
LET
$elevpns \triangleq \{pn \in pns : LeqPortName[priopn,\ pn]\}$
$priopns \triangleq$ IF $elevpns \neq \{\}$ THEN $elevpns$ ELSE pns
$pn \triangleq$ CHOOSE $pn1 \in priopns : \forall\, pn2 \in priopns : LeqPortName[pn1,\ pn2]$
$ok \triangleq priopns \neq \{\}$
IN
IF $\neg ok$ THEN $[ok \mapsto ok]$ ELSE
[
$ok \mapsto ok,$
$pn \mapsto pn$
]

$pns \quad \triangleq Gare\_AssignedPortNames(g)$

$q\_pns \triangleq \{pn \in pns : g.port[pn].ccinq[t] \neq \langle\rangle\}$
$q\_prio \triangleq Prio(q\_pns,\ g.priopn[t])$
IN
IF $q\_prio.ok$ THEN
CASE
$t = CtrlQueueType\_Req \rightarrow Gare\_Work\_PortRequest(g,\ q\_prio.pn)$ □
$t = CtrlQueueType\_Ups \rightarrow Gare\_Work\_PortUpstream(g,\ q\_prio.pn)$
ELSE
g

Compute the updated state of a gare after it processes a setup request ring message.

$Gare\_ProcessSetupReq(g,\ rm) \triangleq$
CASE $TypeAssert(g,\ Gare) \rightarrow$
CASE $TypeAssert(rm,\ RingMsg\_RedSetupReq) \rightarrow$

LET

```
  occupied ≜ LET dp ≜ g.port[rm.downPort]IN
             [sti ∈ Slottina ↦
              ∨ rm.occupied[sti]
              ∨ LET sn ≜ [slottina ↦ sti, subframe ↦ rm.subframe]IN
                 dp.oss[sn].o_assign
             ]

  eval     ≜ RingMsg_Eval_Occupied(rm.subframe, occupied)

  OutG     ≜ Gare_SendRingMsg(g, [
               typecode   ↦ RingMsg_Typecode_RedSetupAns,
               connect    ↦ rm.connect,
               DestC      ↦ rm.DestC,
               DestH      ↦ rm.DestH,
               prevSlot   ↦ rm.prevSlot,
               upPort     ↦ rm.upPort,
               downPort   ↦ rm.downPort,
               subframe   ↦ rm.subframe,
               occupied   ↦ occupied
             ])
IN
IF ¬eval.success THEN OutG ELSE
[OutG EXCEPT !.port[rm.downPort] =
 [Port_SendCtrlCell(@, [
    typecode   ↦ CtrlCell_Typecode_SetupReq,
    connect    ↦ rm.connect,
    DestC      ↦ rm.DestC,
    DestH      ↦ rm.DestH,
    prevSlot   ↦ eval.localSlot
  ])
  EXCEPT !.oss[eval.localSlot] =
  [
    o_assign   ↦ TRUE,
    o_connect  ↦ rm.connect,
    o_nextset  ↦ FALSE,
    o_prevSlot ↦ rm.prevSlot,
    o_nextSlot ↦ NULL,
    o_upPort   ↦ rm.upPort
  ]
 ]
]
```

```
Compute the updated state of a gare after it processes a setup answer ring message.
Gare_ProcessSetupAns(g, rm) ≜
  CASE TypeAssert(g, Gare) →
  CASE TypeAssert(rm, RingMsg_RedSetupAns) →

  LET
    eval  ≜  RingMsg_Eval_Occupied(rm.subframe, rm.occupied)

    Send output ring message and unlock the up port.
    UnlkG  ≜  [Gare_SendRingMsg(g, [
                typecode   ↦ RingMsg_Typecode_RedSetupXbar,
                connect    ↦ rm.connect,
                DestC      ↦ rm.DestC,
                DestH      ↦ rm.DestH,
                prevSlot   ↦ rm.prevSlot,
                upPort     ↦ rm.upPort,
                downPort   ↦ rm.downPort,
                subframe   ↦ rm.subframe,
                occupied   ↦ rm.occupied
              ]) EXCEPT

              !.port[rm.upPort] = [@ EXCEPT
                !.locked[rm.subframe] =
                  CASE Assert(@, "already unlocked subframe on setup ans") →
                  FALSE,
                !.locked_connect[rm.subframe] =
                  CASE Assert(@ = rm.connect, "connection mismatch") →
                  NULL
              ]
            ]

    FailG  ≜  [UnlkG EXCEPT
              !.port[rm.upPort] = Port_SendCtrlCell(@, [
                typecode ↦ CtrlCell_Typecode_Teardown,
                connect  ↦ rm.connect,
                prevSlot ↦ rm.prevSlot,
                reason   ↦ "setup fail"
              ])
            ]

    AssnG  ≜  [UnlkG EXCEPT
              !.port[rm.upPort].iss[eval.localSlot] = [@ EXCEPT
                !.i_assign =
                  CASE Assert(¬@, "already assigned slot on setup ans") →
                  TRUE,
                !.i_connect = rm.connect
```

```
                ]
               ]
  PassG  ≜  [AssnG EXCEPT
               !.port[rm.upPort] = Port_SendCtrlCell(@, [
                 typecode ↦ CtrlCell_Typecode_SetupAck,
                 connect  ↦ rm.connect,
                 prevSlot ↦ rm.prevSlot,
                 nextSlot ↦ eval.localSlot
               ])
              ]
IN
IF ¬eval.success THEN FailG ELSE  PassG
```

Receive ring message rm at gare g. Compute the new state of the gare.

```
Gare_RecvRingMsg(g, rm) ≜
  CASE TypeAssert(g, Gare) →
  CASE TypeAssert(rm, RingMsg) →

  LET
    MyU  ≜  rm.upPort ∈ Gare_AssignedPortNames(g)
    MyD  ≜  rm.downPort ∈ Gare_AssignedPortNames(g)

    OutCopy      ≜  Gare_SendRingMsg(g, rm)
    OutRedNull   ≜  Gare_SendRingMsg(g, RedNullRingMsg)
    OutGreenNull ≜  Gare_SendRingMsg(g, GreenNullRingMsg)

    Upstream(cc) ≜  [OutGreenNull EXCEPT
                      !.port[rm.upPort] = Port_SendCtrlCell(@, cc)
                    ]

    UpInit(g1)   ≜  [g1 EXCEPT !.port[rm.upPort] =
                      CASE Assert(@.iss[rm.localSlot].i_assign, "protocol error") →
                      [@ EXCEPT !.iss[rm.localSlot] = InitPortInSlot]
                    ]

    matchconn    ≜  LET
                      iss  ≜  g.port[rm.upPort].iss
                    IN
                    Assert(rm.connect = iss[rm.localSlot].i_connect,
                      ⟨"up port connection mismatch", rm, iss⟩)

    DoTeardown ≜
                  CASE matchconn →
                  UpInit(Upstream([
```

```
    typecode ↦ CtrlCell_Typecode_Teardown,
    connect  ↦ rm.connect,
    prevSlot ↦ rm.prevSlot,
    reason   ↦ rm.reason
  ]))

DoGCClear ≜
  CASE matchconn →
  UpInit(OutGreenNull)

DoDiscard ≜
  cannot matchconn as no localSlot in a discard ring message
  CASE matchconn →
  Upstream([
    typecode ↦ CtrlCell_Typecode_Discard,
    connect  ↦ rm.connect,
    prevSlot ↦ rm.prevSlot
  ])

DoDataAck ≜
  cannot matchconn as no localSlot in a data ack ring message
  CASE matchconn →
  Upstream([
    typecode   ↦ CtrlCell_Typecode_DataAck,
    connect    ↦ rm.connect,
    prevSlot   ↦ rm.prevSlot,
    chunkseqn  ↦ rm.chunkseqn,
    throttle   ↦ rm.throttle
  ])

DoRestart ≜
  cannot matchconn as no localSlot in a restart ring message
  CASE matchconn →
  Upstream([
    typecode  ↦ CtrlCell_Typecode_Restart,
    connect   ↦ rm.connect,
    prevSlot  ↦ rm.prevSlot
  ])

DoStop ≜
  cannot matchconn as no localSlot in a stop ring message
  CASE matchconn →
  Upstream([
    typecode  ↦ CtrlCell_Typecode_Stop,
    connect   ↦ rm.connect,
    prevSlot  ↦ rm.prevSlot
  ])
```

IN
CASE
$RingMsg\_Is\_RedSetupReq(rm) \quad \wedge MyD \rightarrow Gare\_ProcessSetupReq(g,\ rm) \quad \Box$
$RingMsg\_Is\_RedSetupAns(rm) \quad \wedge MyU \rightarrow Gare\_ProcessSetupAns(g,\ rm) \quad \Box$
$RingMsg\_Is\_RedSetupXbar(rm) \quad \wedge MyU \rightarrow OutRedNull \quad \Box$
$RingMsg\_Is\_GreenTeardown(rm) \wedge MyU \rightarrow DoTeardown \quad \Box$
$RingMsg\_Is\_GreenGCClear(rm) \quad \wedge MyU \rightarrow DoGCClear \quad \Box$
$RingMsg\_Is\_GreenDiscard(rm) \quad \wedge MyU \rightarrow DoDiscard \quad \Box$
$RingMsg\_Is\_GreenDataAck(rm) \quad \wedge MyU \rightarrow DoDataAck \quad \Box$
$RingMsg\_Is\_GreenRestart(rm) \quad \wedge MyU \rightarrow DoRestart \quad \Box$
$RingMsg\_Is\_GreenStop(rm) \quad \wedge MyU \rightarrow DoStop \quad \Box$
$RingMsg\_Is\_RedNull(rm) \quad \rightarrow Gare\_Work(OutCopy,\ CtrlQueueType\_Req) \quad \Box$
$RingMsg\_Is\_GreenNull(rm) \quad \rightarrow Gare\_Work(OutCopy,\ CtrlQueueType\_Ups) \quad \Box$
OTHER $\rightarrow OutCopy$

18 Switch

MODULE $DCNL1Switch$

EXTENDS
$DCNL1Param$,
$DCNL1SlotName$,
$DCNL1CtrlCell$,
$DCNL1DataCell$,
$DCNL1RingMsg$,
$DCNL1SwitchName$,
$DCNL1Port$,
$DCNL1Gare$

$BugNoTimingAssumption \triangleq$ FALSE

State of a $L1$ switch.

$SwitchXbarInfo \triangleq [$
$upPort : PortName,$
$connect : Nullable(Connection)$
$]$

$InitSwitchXbarInfo \triangleq [$

$$
\begin{array}{l}
\quad upPort \;\mapsto FirstPortName, \\
\quad connect \mapsto NULL \\
]
\end{array}
$$

$$
\begin{array}{l}
Switch \triangleq [\\
\quad switchname : SwitchName, \\
\quad gare \quad\quad\;\; : [GareName \to Gare], \\
\quad xbarinfo \quad : [SlotName \;\; \to [PortName \to SwitchXbarInfo]], \\
\quad slot \quad\quad\;\; : SlotName \\
]
\end{array}
$$

$$
\begin{array}{l}
InitSwitch(swn) \triangleq [\\
\quad switchname \mapsto swn, \\
\quad gare \quad\quad\;\; \mapsto [gn \in GareName \mapsto InitGare(swn,\, gn)], \\
\quad xbarinfo \quad \mapsto [sn \in SlotName \;\; \mapsto [pn \in PortName \mapsto InitSwitchXbarInfo]], \\
\quad slot \quad\quad\;\; \mapsto FirstSlotName \\
]
\end{array}
$$

Get a port of a switch.

$$
\begin{array}{l}
Switch\_Port(sw,\, pn) \triangleq \\
\quad \text{LET} \\
\quad\quad gn \triangleq GareName\_ForPortName[pn] \\
\quad \text{IN} \\
\quad sw.gare[gn].port[pn]
\end{array}
$$

Update a port of a switch.

$$
\begin{array}{l}
Switch\_Port\_Update(sw,\, p) \triangleq \\
\quad \text{LET} \\
\quad\quad pn \triangleq p.portname \\
\quad\quad gn \triangleq GareName\_ForPortName[pn] \\
\quad \text{IN} \\
\quad [sw \text{ EXCEPT } !.gare[gn].port[pn] = p]
\end{array}
$$

In switch sw, on port name pn, determine if the switch is ready to receive data cell dc.

$Switch\_Port\_RecvCell\_Ready(sw,\ pn,\ t,\ cell)\ \triangleq$

CASE

$t = CellType\_Data \rightarrow$ TRUE

$\Box$

$t = CellType\_Ctrl \rightarrow$

IF $BugNoTimingAssumption$ THEN TRUE ELSE

In switch sw, on port name pn, determine if the switch is ready to receive a given control cell. There is a timing requirement that the crossbar must be set up for a connection before a setup ack control cell for that connection arrives at the down port input unit.

LET

$cc \triangleq cell$

$match(sn) \triangleq sw.xbarinfo[sn][pn].connect = cc.connect$

IN

CASE

$CtrlCell\_Is\_SetupReq(cc) \rightarrow$ TRUE $\Box$

$CtrlCell\_Is\_SetupAck(cc) \rightarrow match(cc.prevSlot) \Box$

$CtrlCell\_Is\_Teardown(cc) \rightarrow$ TRUE $\Box$

$CtrlCell\_Is\_Discard(cc) \rightarrow$ TRUE $\Box$

$CtrlCell\_Is\_DataAck(cc) \rightarrow$ TRUE $\Box$

$CtrlCell\_Is\_Restart(cc) \rightarrow$ TRUE $\Box$

$CtrlCell\_Is\_Stop(cc) \rightarrow$ TRUE $\Box$

$CtrlCell\_Is\_Null(cc) \rightarrow$ TRUE

In switch sw, on port name pn, receive a given cell.

$Switch\_Port\_RecvCell(sw,\ pn,\ t,\ cell)\ \triangleq$

LET

$p \triangleq Switch\_Port(sw,\ pn)$

$recv \triangleq Port\_RecvCell(p,\ t,\ cell)$

$ready \triangleq Switch\_Port\_RecvCell\_Ready(sw,\ pn,\ t,\ cell) \wedge recv.ready$

$update \triangleq Switch\_Port\_Update(sw,\ recv.update)$

IN

IF $\neg ready$ THEN $[ready \mapsto ready]$ ELSE

$[$

$ready \mapsto ready,$

$update \mapsto update$

$]$

In switch sw, on port name pn, extract a cell.

$Switch\_Port\_TakeCell(sw,\ pn,\ t)\ \triangleq$

LET

```
    p      ≜ Switch_Port(sw, pn)
    take   ≜ Port_TakeCell(p, t)

    ready  ≜ take.ready
    cell   ≜ take.cell
    update ≜ Switch_Port_Update(sw, take.update)
  IN
  IF ¬ready THEN [ready ↦ ready] ELSE
  [
    ready  ↦ ready,
    cell   ↦ cell,
    update ↦ update
  ]
```

Get a sequence of port outputs (debugging).

```
Switch_Debug_LinkOut(sw) ≜
  [pn ∈ PortName ↦ Switch_Port(sw, pn).dcoutq]
```

Advance a crossbar slot time. Compute the updated state of the switch.

```
Switch_AdvanceSlot(sw) ≜
  LET
    pmap0     ≜ [pn ∈ PortName ↦ Switch_Port(sw, pn)]
    sn        ≜ sw.slot
    xbar(pn)  ≜ sw.xbarinfo[sn][pn].upPort

    from(p)   ≜ pmap0[xbar(p.portname)]      port sending to p

    ip(p)     ≜ Port_Xbar_TakeSlot(p, sn).p      updated input port p
    dc(p)     ≜ Port_Xbar_TakeSlot(p, sn).dc     data cell sent to xbar by p
    xb(p)     ≜ Port_Xbar_RecvSlot(ip(p), sn, dc(from(p)))   full update of p

    pmap1     ≜ [pn ∈ PortName ↦ xb(pmap0[pn])]
  IN
  [sw EXCEPT
   !.gare = [gn ∈ GareName ↦
             [@[gn] EXCEPT
              !.port = [pn ∈ PortName ↦
                        IF pn ∉ gn.part THEN NULL ELSE pmap1[pn]
                       ]
             ]
```

```
            ],
 !.slot  = IncrSlotName[@]
]
```

Compute the updated state of the switch after pushing the out ring message from each *gare* into the next *gare* around the ring.

```
Switch_AdvanceRing(sw) ≜
  LET
    ring message entering gare gn
    rmin(gn) ≜ sw.gare[DecrGareName[gn]].ring_out
  IN
  [sw EXCEPT
    Update each gare acording to the ring message entering it.
   !.gare = [gn ∈ GareName ↦ Gare_RecvRingMsg(@[gn], rmin(gn))],

    Update xbar info according to any setup xbar ring message entering the first gare. In the actual hardware, each gare handles a bit slice of the crossbar, and the crossbar setup happens for each bit slice as the setup xbar ring message enters the respective gare. We simplify this by doing all the crossbar setup at the first gare.
   !.xbarinfo =
     LET
       rm   ≜ rmin(FirstGareName)
       eval ≜ RingMsg_Eval_Occupied(rm.subframe, rm.occupied)
     IN
     IF ¬RingMsg_Is_RedSetupXbar(rm) THEN @ ELSE
     IF ¬eval.success THEN @ ELSE
     [@ EXCEPT ![eval.localSlot][rm.downPort] =
      [@ EXCEPT
       !.upPort = rm.upPort,
       !.connect = rm.connect
      ]
     ]
  ]
```

19 Full L1 system

MODULE $DCNL1$

EXTENDS

$DCNL1Param$,
$DCNL1SlotName$,
$DCNL1CtrlCell$,
$DCNL1DataCell$,
$DCNL1RingMsg$,
$DCNL1SwitchName$,
$DCNL1Port$,
$DCNL1Gare$,
$DCNL1Switch$,
$DCNL1Node$

Specification of the $L1$ switch infrastructure in Chuck $Thacker$'s Data Center Network version 4.4.

CONSTANT $MaxPacketCells$
CONSTANT $MinPacketCells$

State of the system.

$$NodeName \triangleq [$$
$$\quad swn : SwitchName,$$
$$\quad hr \ : Halfrow$$
$$]$$

$$Request \triangleq [$$
$$\quad connect : Connection,$$
$$\quad srcx \ : NodeName,$$
$$\quad dstx \ : NodeName,$$
$$\quad pkt \ : Seq(Payload)$$
$$]$$

$$State \triangleq [$$
$$\quad node : [NodeName \to Node],$$ nodes in the system
$$\quad switch : [SwitchName \to Switch],$$ switches in the system
$$\quad request\_cnt : Nat,$$ number of requests launched
$$\quad request\_set : \text{SUBSET } Request,$$ set of requests lauched
$$\quad debug : Any$$ debugging info
$$]$$

$$InitState \triangleq [$$
$$\quad node \mapsto [nn \in NodeName \mapsto InitNode],$$

$switch \mapsto [swn \in SwitchName \mapsto InitSwitch(swn)],$
$request\_cnt \mapsto 0,$
$request\_set \mapsto \{\},$
$debug \mapsto NULL$
]

Link definitions

A link end is either a node or a port.

$LinkEnd\_Typecode\_Node \triangleq$ "LinkEnd_Node"
$LinkEnd\_Typecode\_Port \triangleq$ "LinkEnd_Port"

$LinkEnd\_Node \triangleq [$
$typecode : \{LinkEnd\_Typecode\_Node\},$
$nn : NodeName$
]

$LinkEnd\_Port \triangleq [$
$typecode : \{LinkEnd\_Typecode\_Port\},$
$swn : SwitchName,$
$pn : PortName$
]

$LinkEnd \triangleq \{\}$
$\cup\ LinkEnd\_Node$
$\cup\ LinkEnd\_Port$

The set of all port-to-port links.

$Link\_PortPort \triangleq$
LET
$slkes \triangleq$
$\{lke \in LinkEnd\_Port :$
$PortName\_Is\_Container(lke.pn)$
$\}$
$dlke(slke) \triangleq$
CHOOSE $lke \in LinkEnd\_Port :$
LET
$spn \triangleq slke.pn$
$dpn \triangleq lke.pn$
IN
$\wedge\ PortName\_Is\_Container(dpn)$

```
        ∧ dpn.to = slke.swn.container
        ∧ lke.swn.container = spn.to
    IN
    {[srcx ↦ slke, dstx ↦ dlke(slke)] : slke ∈ slkes}
```

The set of all port-to-node and node-to-port links.

```
Link_PortNode ≜
  LET
    slkes ≜
      {lke ∈ LinkEnd_Port :
       PortName_Is_Halfrow(lke.pn)
      }
    dlke(slke) ≜
      CHOOSE lke ∈ LinkEnd_Node :
      LET
        spn ≜ slke.pn
        dnn ≜ lke.nn
      IN
       ∧ dnn.swn = slke.swn
       ∧ dnn.hr = spn.to
  IN
  {}
   ∪ {[srcx ↦ slke, dstx ↦ dlke(slke)] : slke ∈ slkes}
   ∪ {[dstx ↦ slke, srcx ↦ dlke(slke)] : slke ∈ slkes}
```

The set of all links.

```
Link ≜ {}
  ∪ Link_PortPort
  ∪ Link_PortNode
```

In state s, on link lk, receive a cell of the indicated type.

The receiver might not be ready for this particular cell yet. The hardware avoids this situation by using timing assumptions that are carefully checked in the hardware design. In the specification, we model it explicitly.

```
State_Link_RecvCell(s, lk, t, cell) ≜
  LET
    DoNode ≜
      LET
        nn   ≜ lk.dstx.nn
        recv ≜ Node_RecvCell(s.node[nn], t, cell)
```

```
        ready  ≜ recv.ready
        update ≜ [s EXCEPT !.node[nn] = recv.update]
      IN
      IF ¬ready THEN [ready ↦ ready] ELSE
      [
        ready  ↦ ready,
        update ↦ update
      ]

    DoPort ≜
      LET
        swn    ≜ lk.dstx.swn
        pn     ≜ lk.dstx.pn
        recv   ≜ Switch_Port_RecvCell(s.switch[swn], pn, t, cell)

        ready  ≜ recv.ready
        update ≜ [s EXCEPT !.switch[swn] = recv.update]
      IN
      IF ¬ready THEN [ready ↦ ready] ELSE
      [
        ready  ↦ ready,
        update ↦ update
      ]
  IN
  CASE
  lk.dstx.typecode = LinkEnd_Typecode_Node → DoNode□
  lk.dstx.typecode = LinkEnd_Typecode_Port → DoPort
```

In state s, on link lk, extract a cell of the indicated type from the sender.

```
State_Link_TakeCell(s, lk, t) ≜
  LET
    DoNode ≜
      LET
        nn     ≜ lk.srcx.nn
        take   ≜ Node_TakeCell(s.node[nn], t)

        ready  ≜ take.ready
        cell   ≜ take.cell
        update ≜ [s EXCEPT !.node[nn] = take.update]
      IN
      IF ¬ready THEN [ready ↦ ready] ELSE
      [
        ready  ↦ ready,
        cell   ↦ cell,
```

```
      update ↦ update
    ]

  DoPort ≜
    LET
      swn    ≜ lk.srcx.swn
      pn     ≜ lk.srcx.pn
      take   ≜ Switch_Port_TakeCell(s.switch[swn], pn, t)

      ready  ≜ take.ready
      cell   ≜ take.cell
      update ≜ [s EXCEPT !.switch[swn] = take.update]
    IN
    IF ¬ready THEN [ready ↦ ready] ELSE
    [
      ready  ↦ ready,
      cell   ↦ cell,
      update ↦ update
    ]
IN
CASE
lk.srcx.typecode = LinkEnd_Typecode_Node → DoNode□
lk.srcx.typecode = LinkEnd_Typecode_Port → DoPort
```

VARIABLE *state*

NEXT STATE RELATIONS

Advance the slot time everywhere.

```
NextSlot ≜
  state′ = [state EXCEPT
             !.node = [nn ∈ NodeName ↦ Node_AdvanceSlot(@[nn])],
             !.switch = [swn ∈ SwitchName ↦ Switch_AdvanceSlot(@[swn])],
             !.debug = ⟨"adv slot"⟩
           ]
```

Advance the ring in switch *swn*.

```
NextRingSwitch(swn) ≜
  LET
    sw0 ≜ state.switch[swn]
```

```
    sw1 ≜ Switch_AdvanceRing(sw0)
  IN
  ∧ state' = [state EXCEPT
                !.switch[swn] = sw1,
                !.debug = ⟨"sw adv ring", swn⟩
              ]
```

Move a cell of the indicated type across link lk.

```
NextLinkCell(lk, t) ≜
  LET
    take ≜ State_Link_TakeCell(state, lk, t)
    recv ≜ State_Link_RecvCell(take.update, lk, t, take.cell)
  IN
  ∧ take.ready
  ∧ recv.ready
  ∧ state' = [recv.update EXCEPT
                !.debug = ⟨"link", t, lk.srcx, lk.dstx, take.cell⟩
              ]
```

Create a request to send a packet. With back-to-back nodes, the destination is always the other node than the source. However, we still have to pick $destV$, $destC$, and $destH$ for the setup request control cell.

```
NextCreateRequest ≜
  ∃ srcx ∈ NodeName :
  LET
    cnt  ≜ state.request_cnt + 1
    srcn ≜ state.node[srcx]                    node to send the packet
  IN
  ∧ cnt ≤ NumConnection                        limit on total packets
  ∧
    ∃ s   ∈ Node_AvailOutSlots(srcn) :         outgoing local slot in node n
    ∃ C ∈ Container :                          destination container
    ∃ H ∈ Halfrow :                            destination halfrow
    ∃ len ∈ MinPacketCells .. MaxPacketCells : length of the packet
    ∃ pkt ∈ [1 .. len → Payload] :             the packet
    LET
      dstx ≜ [swn ↦ SwitchName_ForContainer(C), hr ↦ H]
    IN
    state' =
      [state EXCEPT
       !.node[srcx] = Node_CreateRequest(@, s, C, H, pkt, cnt),
       !.request_cnt = cnt,
```

$$
\begin{array}{l}
\quad !.request\_set \;=\; @ \cup \{[\\
\qquad\qquad\qquad\qquad\qquad\quad connect \mapsto cnt, \\
\qquad\qquad\qquad\qquad\qquad\quad srcx \quad\;\; \mapsto srcx, \\
\qquad\qquad\qquad\qquad\qquad\quad dstx \quad\;\; \mapsto dstx, \\
\qquad\qquad\qquad\qquad\qquad\quad pkt \quad\;\; \mapsto pkt \\
\qquad\qquad\qquad\qquad\qquad]\}, \\
\quad !.debug = \langle \text{“create req”}, \; srcx, \; dstx, \; pkt \rangle \\
]
\end{array}
$$

INVARIANTS

The state must always be of the proper type.

$$
\begin{array}{l}
InvType \;\triangleq \\
\quad \wedge \; state \in State
\end{array}
$$

Any packets received must have been sent.

$$
\begin{array}{l}
InvRecvPacket \;\triangleq \\
\quad \forall \, dstx \in NodeName : \\
\quad \text{LET} \\
\qquad dstn \;\triangleq\; state.node[dstx] \\
\qquad pkts \;\triangleq\; Node\_ReceivedPackets(dstn) \\
\quad \text{IN} \\
\quad \forall \, pkt \in pkts : \\
\quad \exists \, r \in state.request\_set : \\
\quad \wedge \; r.dstx = dstx \\
\quad \wedge \; r.pkt = pkt
\end{array}
$$

TEMPORAL PROPERTIES

All node output slots always eventually available.

$$
\begin{array}{l}
AllNodeOutSlotEvAvail \;\triangleq \\
\quad \forall \, nn \in NodeName : \\
\quad \forall \, sn \in SlotName \;\; :
\end{array}
$$

$\Box\Diamond Node\_IsAvailOutSlot(state.node[nn],\ sn)$

All node input slots always eventually available.

$AllNodeInSlotEvAvail \triangleq$
$\forall\, nn \in NodeName :$
$\forall\, sn \in SlotName :$
$\Box\Diamond Node\_IsAvailInSlot(state.node[nn],\ sn)$

All links are eventually quiet.

$AllLinkEvQuiet \triangleq$
$\forall\, lk \in Link :$
$\forall\, t \in CellType :$
$\Box\Diamond\neg State\_Link\_TakeCell(state,\ lk,\ t).ready$

Initial state.

$Init \triangleq$
$\land\ FirstGareName \neq NULL$
$\land\ state = InitState$

Next state relation.

$Next \triangleq$
$\lor\ \exists\, swn \in SwitchName : NextRingSwitch(swn)$

$\lor\ \exists\, lk \in Link :$
$\quad \exists\, t \in CellType :$
$\quad NextLinkCell(lk,\ t)$

$\lor\ NextSlot$

$\lor\ NextCreateRequest$

Liveness.

$Liveness \triangleq$
$\land\ \forall\, swn \in SwitchName :$
$\quad \mathbf{WF}_{state}(NextRingSwitch(swn))$

$\land\ \forall\, lk \in Link :$
$\quad \forall\, t \in CellType :$
$\quad \mathbf{WF}_{state}(NextLinkCell(lk,\ t))$

$\land \text{WF}_{state}(NextSlot)$

System specification.

$Spec \triangleq$
$\land Init$
$\land \Box[Next]_{state}$
$\land Liveness$

View removing debugging info.

$View \triangleq [state \text{ EXCEPT } !.debug = NULL]$

References

[1] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[2] C. Thacker, A. Nowatzyk, T. Rodeheffer, and F. Yu. A data center network using FPGAs (v4.5), April 2011. Unpublished.