Modeling HexNet in ACL2

Developing a formal specification for HexNet

Presentation by Ebele Esimai

Outline

- Introduction
- Overview of HexNet
- Modeling Approach
- Proofs
- Future works and Conclusion

Outline

- Introduction
- Overview of HexNet
- Modeling Approach
- Proofs
- Future works and Conclusion

Introduction

- Instead of a global clock, a self-timed network paces the flow of data items with local communication protocols.
- Each data item proceeds as soon as space for receiving it is available
- In the presence of congestion, reverse timing signals retard forward flow to prevent collision
- New data items safely enter into the stream of the network traffic by merging through an arbitration circuit

Objective

Proofs of correct self-timed VLSI system behavior

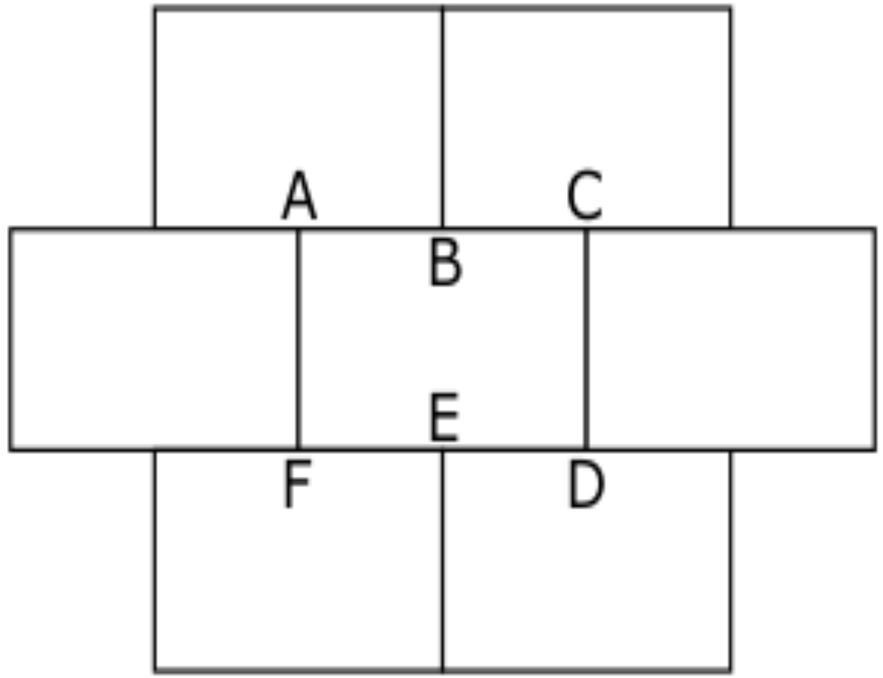
- Case : Network on Chip (NoC)
- Design of the NoC : Network plan based on hexagonal topology (proposed by Ivan Sutherland 2013)
- Goal : Explore of the practicalities of design and ensure correctness of the logic for HexNet’s subtle address calculations

Outline

- Introduction
- Overview of HexNet
- Modeling Approach
- Proofs
- Future works and Conclusion

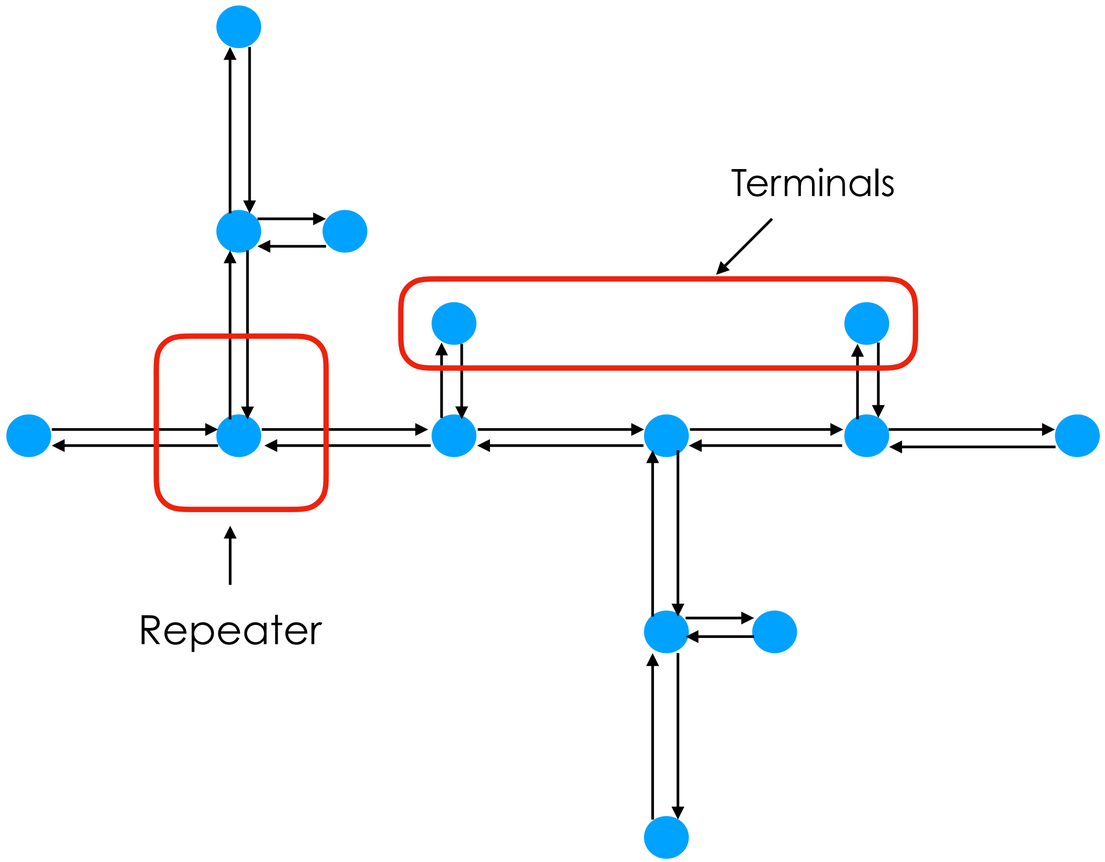
HexNet Layout

- Hexagonal layout of the network
- Two-way merging in the network
- Each node in the network is a called a JUNCTION
  - arbitration decisions
  - packet routing
  - packet information update
- Each edge is a link, storage element for packets



Junctions

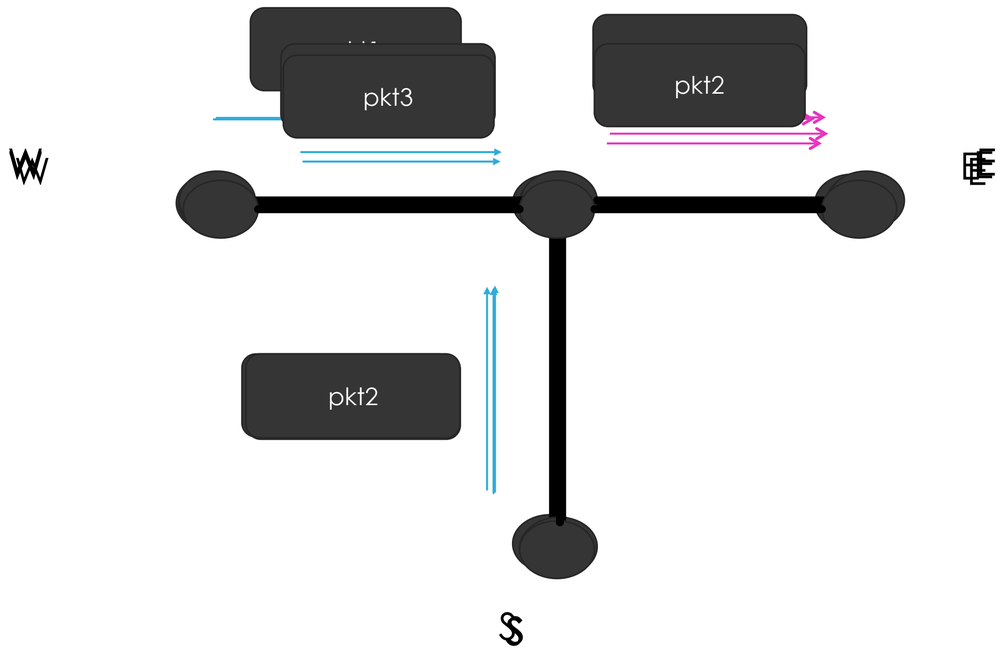
- Junctions can be of two types
  - Terminal : This is the point of packet entry or exit from the network.
  - Repeaters : This accepts packets and forwards it to another junction.



Features of HexNet

Arbitration

- Correct decision behavior of the network when packets arrive on two input links at nearly the same time.
- A mechanism for fairness is necessary. The model stores the arbitration decision made at the previous cycle.
- Arbitration is based on the output direction.
- One of the packets is chosen to proceed while the second waits.
- In the following cycle, the packet waiting gets to proceed.



Features of HexNet

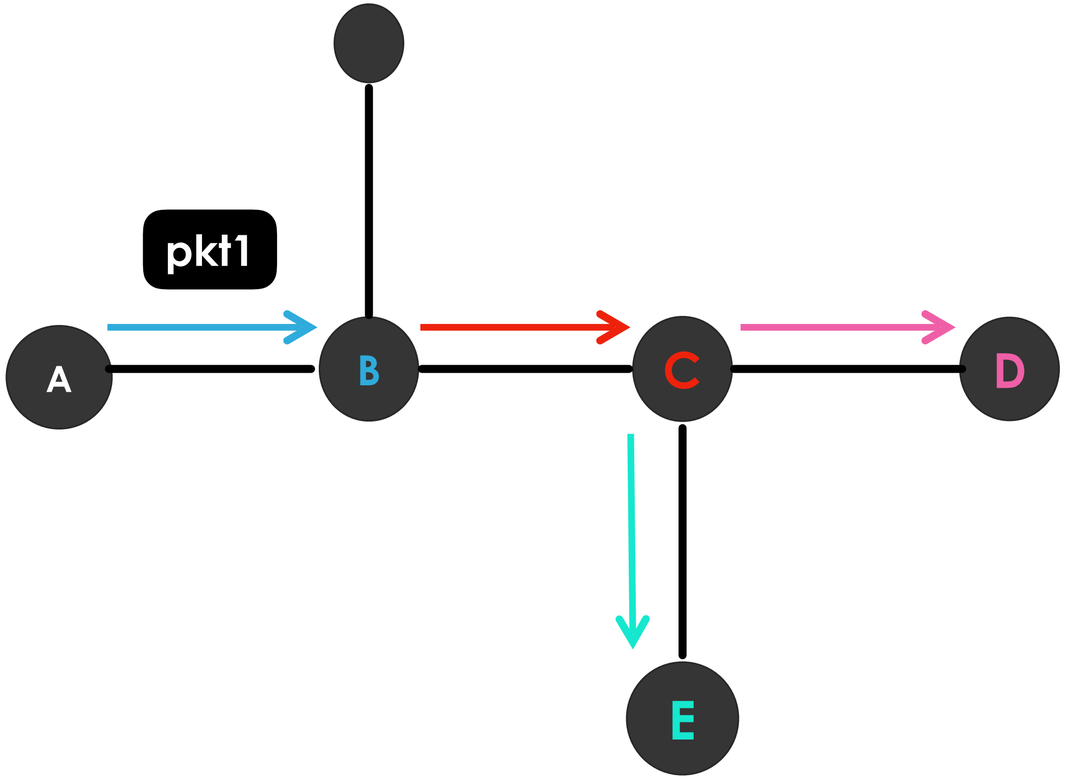
Packet Routing

- The network uses **Advanced Address Decoding**
- This means that the routing function calculates the next direction, i.e. turn-signal, a step in advance. This means before it actually takes a turn.
- The routing function decides the packet direction by comparing the final destination stored in the packet with the address of next junction in its path.
- Let the current turn_signal of pkt1 be East, thus, the packet would continue to junction C.

Turn_signal (pkt1) = East

- But before leaving junction B, it calculates the value of the next turn_signal, that is, the direction it should take when it leaves junction C.

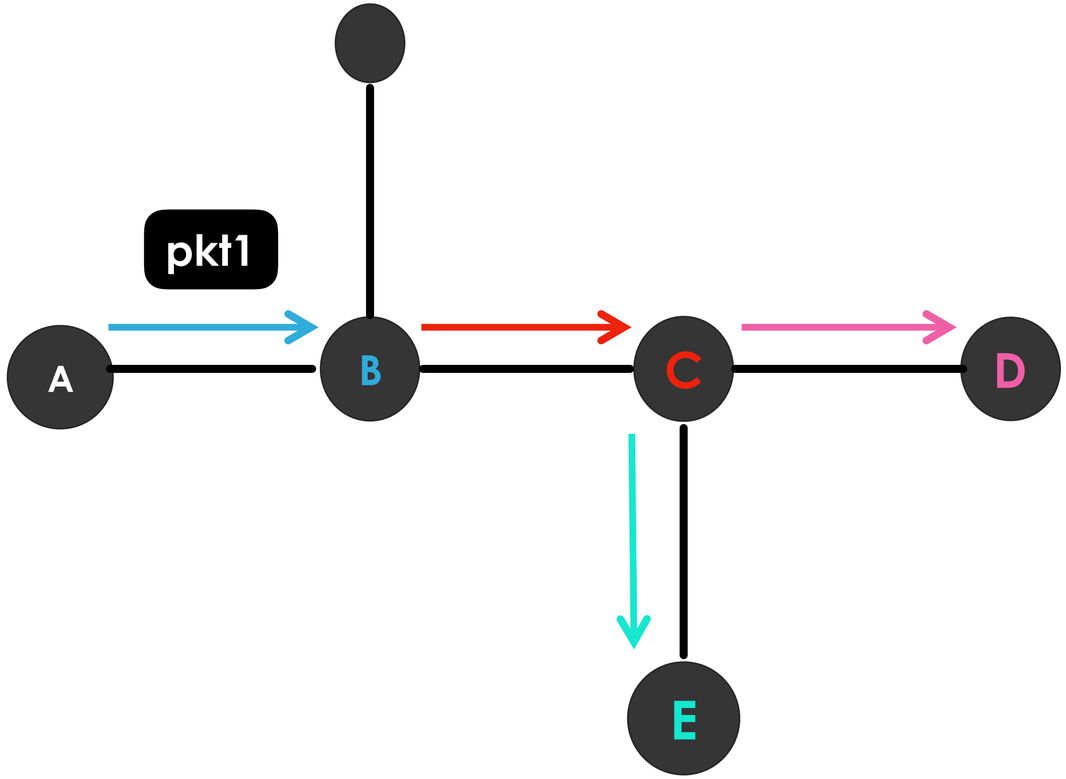
(compare_addresses junctC final_destination) => East or South



Packet Routing

Routing Algorithm

- Get the next possible junctions from current junction.
- Check if either junction is the final destination address
- If not, check if either junction is a terminal. If one is a terminal, return the direction of the repeater.
- If either junction is a repeater, compare the current junction address with the final destination address.
  - return the direction of repeater which is closest to the final destination
  - vertical movement is preferred



- Next possible junctions from **C** are **D** and **E**

Outline

- Introduction
- Overview of HexNet
- Modeling Approach
- Proofs
- Future works and Conclusion

Modeling Approach

Incremental extension and refinement of the model to capture HexNet functionalities and behavior.

1. Network represented as an association list with the keys as arbitrary names of junctions and value as a list of pairs of connected junctions and associated undirected links. Also, packets have stored predefined paths and all junctions are processed at each cycle of execution.

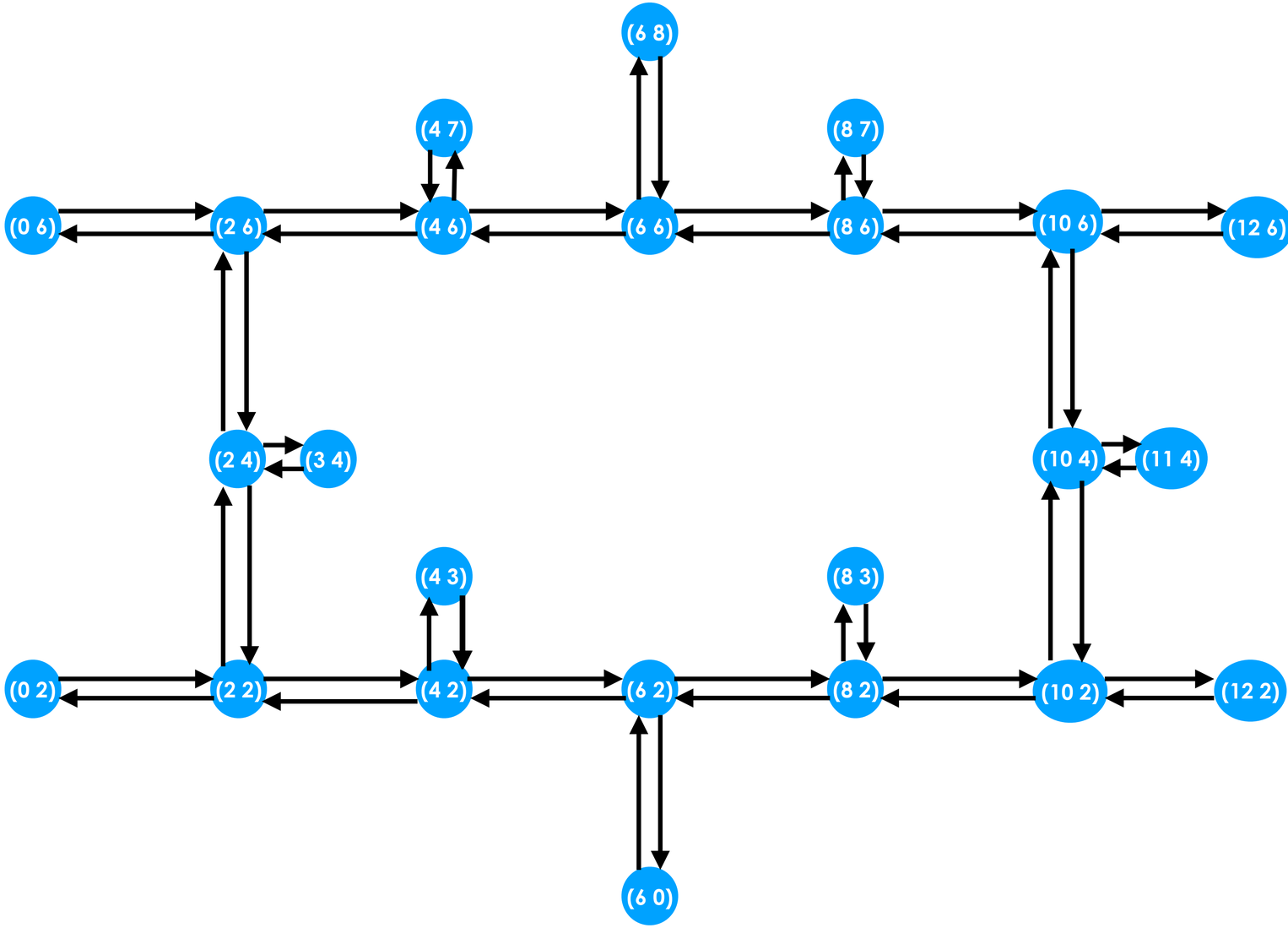
2. Undirected link is replaced with two directed links for concurrent network traffic. Arbitration of input links. Restriction of process list to junctions with packets on their input links.

3. Junctions are named by their x- and y- coordinates. Address decoding and packet routing. Packet structure defined as a list of a turn signal, a final destination address, an origin address and data item.

4. Packet entry into and exit from the network.

Current Model in ACL2

- The network is modeled as a graph.
- Addressing is based on a cartesian grid, as such, each junction is list of its x coordinate and its y coordinate
- Packet movement in the network is bidirectional.
- Packet entry and exit from the network is through the terminals



Packets

Pkt1

```
'(pkt1 E 0 (8 . 7) (0 . 2) 'data)
```

- A packet is modeled as a list that contains
  - A packet id
  - A turn signal :- the next direction the packet should take
  - A processed bit :- this ensures that a packet moves at most once in each ``cycle'' of execution of the system
  - Final terminal :- the address of final destination of the packet
  - Source terminal :- the address of the origin of the packet
  - The information to be passed along

- The packets are stored on the links.
- A link-table shows the current state of all the links in the network at a given time.
- A link can either be empty or full.
- In the link-table, each link is depicted as either empty or with the packet stored on it.

```
(defconst *linktable*
    '((S2 (pkt1 E 0 (12 . 2) (0 . 2) data))
      (S1)
     (S4)
     (S3)
     …
    (T19 (pkt2 E 0 (12 . 2) (6 . 8) data))
     …))
```

Running the system

Multi-step Function : runs multiple iterations of the system.

- Terms :
  - **`input-stream`** : A stream of inputs with some packet information - packet id, a final destination address, a source address and information to be passed along
  - **`hist`** : The global history of all delivered packets
  - **`lt`** : State of the system
  - **`imap`** : inverse graph
  - **`ast`** : arbiter state
  - **`g`** : hexnet network
  - **`MS`** : model state
- How multi-step works :
  - removes any delivered packets
  - get the list of junctions with current inputs
  - run the step junction
  - add new packets
  - repeat

```
(defun multi-step (input-stream hist lt imap ast g MS)
 (if MS
    (mv MS nil)
   (if (endp input-stream)
       (mv lt hist)
    (b* (((mv trans-lt1 new-hist)
          (remove-delivered-packets hist (strip-cars lt) lt g))
         (junct-list (get-junct-list trans-lt1 imap g))
         ((mv new-MS trans-lt2 new-ast)
           (Single-step junct-list trans-lt1 ast g MS))
         (trans-lt3 (reset_process_bit (strip-cars trans-lt2)
                                       trans-lt2 g))
         (new-lt (add-new-packets (car input-stream)
                                  trans-lt3 g)))
      (multi-step (cdr input-stream) new-hist
                  new-lt imap new-ast g new-MS)))))
```

Single-step Function : a single cycle of execution in the system. All packets in the system take at most one step.

- Terms :
  - **junct-list** : The list of junctions that have packets on its inputs links.
  - **lt** : State of the system
  - **ast** : arbiter state
  - **g** : hexnet network
  - **MS** : model state

```
(defun Single-step (junct-list lt ast g MS)
 (if MS
    (mv MS lt ast)
   (if (endp junct-list)
     (mv MS lt ast)
    (b* ((junct (car junct-list))
         (neighbors (sources junct g))
         ((mv new-MS new-lt new-ast)
          (process-junct junct neighbors lt ast g MS)))
      (Single-step (cdr junct-list) new-lt new-ast g
                   new-MS)))))
```

Update Link function : The link table shows the current state of the system, thus, to model **packet movement**, the link table is updated as long as some packet can move.

- Terms :
  - **`inputs`** : The list of input links into junct with packets on them. The arbiter returns one of these links.
  - **`lt`** : State of the system
  - **`ast`** : arbiter state
  - **`g`** : hexnet network
  - **`MS`** : model state

Algorithm:

- Check if the receiving link (outlink) is empty. If full, no update is made.
- Otherwise, the arbiter gives a chosen input link. Check if the packet on the input link is valid.
- If the packet is valid, the routing function calculates and returns the packet's next ``turn_signal'' direction.
- If the ``turn_signal'' is valid, repack the packet and update the link table by adding it to output link. Then, update the link table to remove the old packet from the input link.
- Otherwise, make no changes to the link table.

```
(defun update-link (junct dest lt ast g MS)
  (if MS
      (mv MS lt ast)
    (let* ((outlink (get-output-link junct dest g)))
      (if (get-packet outlink lt g)
          (mv MS lt ast)
        (let* ((stack (remove-equal dest (sources junct g)))
               (inputs (current-inputs junct dest stack lt g)))
          (if (endp inputs)
              (mv MS lt ast)
            (mv-let (input new-ast) (arbiter outlink inputs ast g)
              (if (null (get-packet input lt g))
                  (mv "Invalid input, problem from the Arbiter" lt ast)
                (let* ((pkt (get-packet input lt g))
                       (final (get-final-dest pkt g))
                       (turn_signal (routing junct dest final g)))
                  (if (turn_signalp turn_signal)
                      (let* ((new-pkt (list* (car pkt) turn_signal 1
                                             (cdddr pkt)))
                             (new-link-state (update-alist outlink
                                                           (list new-pkt) lt))
                             (new-lt (update-alist input nil new-link-state)))
                        (mv MS new-lt new-ast))
                    (mv turn_signal lt new-ast)))))))))))
```

Adding new packets to the network

Packets are added to the network through the terminals. Such arriving packets are viewed as a stream of input that is then split into valid packets.

- Terms :
  - **input** : A stream of inputs with some packet information - a packet id, a final destination address, a source address and information to be passed along.
  - **lt** : State of the system
  - **g** : hexnet network
- For each input in the stream, determine if the junction at the origin address is a terminal.
- If so, calculate the first direction of the packet using the routing algorithm.
- Get the output link name, where the packet will stored upon entry into the network
- If the link is not full, fill the link

```
(defun add-new-packets (input lt g)
  (if (endp input)
      lt
    (let* ((entry (car input))
           (pktid (car entry))
           (final (cadr entry))
           (source (caddr entry))
           (data (cadddr entry)))
      (if (isTerminal source g)
          (let* ((junctB (car (sources source g)))
                 (turn_signal (routing source junctB final g)))
            (if (turn_signalp turn_signal)
                (let* ((pkt (list pktid turn_signal 0 final source data))
                       (outlink (get-output-link source junctB g))
                       (new-lt (update-alist outlink (list pkt) lt)))
                  (add-new-packets (cdr input) new-lt g))
              (add-new-packets (cdr input) lt g)))
        lt))))
```

Removing delivered packets from the network

Packets are delivered to their final destination, terminals. To track the output from the network, we maintain a global history of all delivered packets.

- Terms :
  - **`hist`** : A list of delivered packets
  - **`lt`** : State of the system
  - **`g`** : hexnet network
- Maintain a global history for delivered packets
- Walk through the link table before each cycle of the system
- Check if packet present on the link has the final destination address as the current destination of link
- If so, update the global history and empty the link

```
(defun remove-delivered-packets (hist lnk-lst lt g)
  (if (atom lnk-lst)
      (mv lt hist)
    (if (get-packet (car lnk-lst) lt g)
      (let* ((pkt (get-packet (car lnk-lst) lt g))
             (turn_signal (get-turn-signal pkt g)))
        (if (equal turn_signal 'Done)
           (let
            ((new-hist (cons (get-packet (car lnk-lst) lt g)
                             hist))
             (new-lt (update-alist (car lnk-lst) nil lt)))
            (remove-delivered-packets new-hist
                                      (cdr lnk-lst) new-lt g))
          (remove-delivered-packets hist (cdr lnk-lst)
                                    lt g)))
      (remove-delivered-packets hist (cdr lnk-lst) lt g))))
```

Modeling Choices

- High level abstraction focused on global system function, data and control flow, thus no demonstration of asynchrony in current model
- Junction-focused model versus Link-focused model : Though the links are the storage elements and they describe the state of the system, flow of control decisions are made at the junctions.
- Packets along the same path arrive at their destination in order
- A packet would not change the course of its path even if the next link on its path is full
- Each packet has a processed bit that ensures a packet moves at most once in each cycle of execution.
- The path taking by the routing algorithm is not always the shortest path because of vertical first preference

Outline

- Introduction
- Overview of HexNet
- Modeling Approach
- Proofs
- Future works and Conclusion

Verifying the model

There are some properties we desire of the model include

- The network is a connected graph such that every source can send a packet to any destination.
- Some packet makes progress on its path to its destination. We would like to have a strong property that the network is deadlock free but we can not make that claim yet.
- Packets are delivered to their actual destinations.

Challenges

- Modeling decision of the layout of the network - number of destinations, shape of the network, completely connected network
- Defining a simple, yet sufficient, routing algorithm for HexNet’s subtle address calculations.
- Creating the right measure value for proofs using the routing algorithm.
- Approach to proposed theorems, global system view or packet view.

Outline

- Introduction
- Overview of HexNet
- Modeling Approach
- Proofs
- Future works and Conclusion

Conclusion

- A complete model of HexNet in ACL2 with verified guards
- Almost complete reachability theorem
- Other proofs in progress

Future Work

- Complete proofs on desired properties
- Create a new abstraction model of the network with asynchrony
- Create mapping between the new model and present model if possible
- Enhance the model to depict all functionalities of HexNet

References

- Ivan Sutherland, Marly Roncken, Navaneeth Jamadagni, Chris Cowan, and Swetha Mettala Gilla: The Weaver, an 8x8 Crossbar Experiment.

- Marly Roncken, Ivan Sutherland and Chris Chen: Latches in a Network.

```
(defun arbiter (outlink inputs ast g)
  (let* ((entry (assoc-equal outlink ast)))
    (cond
     ((endp inputs)  (mv nil ast))
     ((endp (cdr inputs))  (mv (car inputs) ast))
     (t (if entry
            (let* ((last-choice (cdr entry))
                   (new-choice (car (remove1-equal last-choice
                                                   inputs)))
                   (new-ast (update-alist outlink new-choice ast)))
              (mv new-choice new-ast))
          (let* ((choice (car inputs))
                 (new-ast (acons outlink choice ast)))
            (mv choice new-ast)))))))
```

```
(defun Routing (past curr final g)
  (declare (xargs :guard (routing-guard past curr final g)))
  (let ((dests (remove-equal past (sources curr g))))
    (cond
      ;; Case 1: If there is no junction in the list Dests, then the packet is at its destination
      ;; or there is an error
     ((atom dests) (if (equal curr final)
                       'Done
                     (cw "Bad route")))
    ;; Case 2: If there is only one junction in the list Dests, then return the direction to the junction
     ((atom (cdr dests)) (get-direction curr (car dests) g))
```

```
((atom (cddr dests))
         (let* ((dest1 (car dests))                          ;; Case 3: If there are two junctions, determine which junction
                (dest2 (cadr dests))                         ;; will take the packet closest to the final destination address
                (dir1 (get-direction curr dest1 g))
                (dir2 (get-direction curr dest2 g)))
       (cond
        ((isTerminal dest1 g)  (if (equal dest1 final)       dir1
                                   (if (isTerminal dest2 g)   (if (equal dest2 final)  dir2  (cw "Bad route"))
                                     dir2)))
        ((isTerminal dest2 g)  (if (equal dest2 final) dir2  dir1))
        (t    (let ((curr-x (car curr))      (curr-y (cdr curr))
                    (final-x (car final))    (final-y (cdr final))
                    (dest1-x (car dest1))    (dest1-y (cdr dest1))
                    (dest2-x (car dest2))    (dest2-y (cdr dest2)))
              (cond    ((< curr-y final-y) (if (= dest1-y dest2-y)
                                               (if (< (abs (- dest1-x final-x)) (abs (- dest2-x final-x)))  dir1 dir2)
                                             (if (< dest1-y dest2-y)  dir2  dir1)))
                       ((> curr-y final-y) (if (= dest1-y dest2-y)
                                               (if (< (abs (- dest1-x final-x))(abs (- dest2-x final-x))) dir1 dir2)
                                             (if (< dest1-y dest2-y) dir1 dir2)))
                       ((< curr-x final-x) (if (< dest1-x dest2-x) dir2 dir1))
                       ((> curr-x final-x) (if (< dest1-x dest2-x) dir1 dir2))))))))
                       (t nil))))
```