# Simulating Dataflow Graphs with C++

David Watkins Columbia University
New York, New York USA
May 10, 2017
djw2146@columbia.edu

## ABSTRACT

Compiling high level programming languages into hardware is no small task. It requires dividing the program into constituent parts that are representable by a hardware circuit and creating a proper memory management system that can fit on a single hardware circuit. Designing a memory system that can reduce contention requires analysis of the dataflow circuit generated from the high level program and can be determined using a graph coloring algorithm and using a separate memory system for each color of the graph. This will reduce memory contention and allow the system to work faster overall.

## 1. MOTIVATION

In coordination with Professor Stephen Edwards at Columbia University, the project *Hardware synthesis from a recursive functional language*[2] that compiles a subset of Haskell into a hardware description language requires a backend for compilation. In this case this means compiling both a simulated version of the dataflow network and compiling to SystemVerilog so it can be synthesized as a circuit. The difficulty with this problem is creating a memory layout for the resulting circuit so that contention between similarly typed memory elements is reduced or removed completely while still conforming to the constraints of the system. In order to determine this one needs to create the synthesized dataflow model through either a simulation of the dataflow in C++ or a simulation of the resultant HDL code using a program such as ModelSim.

## 2. LOGISTICS

The three most critical parts of the work I was able to do for this project were:

- Visualizing the dataflow graphs
- Developing an algorithm for avoiding contention in memory accesses for a dataflow network
- Developing an architecture for the generated simulator code from a given dataflow network

What would follow from this work would be to continue evaluating the effectiveness of the memory contention avoidance algorithm and finishing the compilation step to also compile to SystemVerilog. There was not enough time in the semester to complete these two tasks.

## 3. METHODS

In the next few subsections I will discuss the technologies and used in the process of the research project.

### 3.1 Background

We have described a high-level synthesis dataflow model that is the result of a transformation of a recursive function program with algebraic data types. We attempt to synthesize algorithms with irregular memory access patterns and complex control behavior. By using a limited dialect of Haskell we can take advantage of lazy evaluation and data immutability patterns in hardware to extract unique optimization that would otherwise not exist in an imperative source language.

Our dataflow model relies on a series of nodes and channels, where the nodes perform computational work as a function of the inputs and outputs from the channels which contains both data and signals to continue or stop doing work. The ready/valid signal is a large component of whether a node can send work to the next node or process work from a source node.

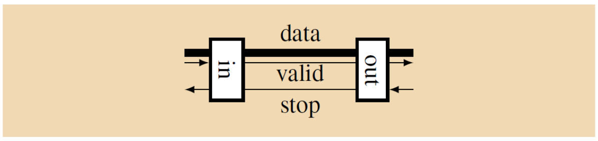

**Figure 1: A single-place buffer built from a pair of our input and output buffers. Shows the valid and stop (or ready) signals from a pair in and out nodes.** [1]

The source language, in this case Floh, would be used to translate into the dataflow language which needs to be simulated. Each of the nodes in the dataflow model can either be stateful or stateless, mostly depending on whether they need access to memory or if they need to remember which of the downstream nodes they have previously served.

### 3.2 Defining Nodes

The nodes have all been previously defined by Richard Townsend through the simulator he produced for the simulation of the dataflow programs. The nodes are listed as follows:

- merge
- mergeChoice
- source
- sink
- fork
- mux
- demux
- choice
- callLock
- func
- iconst
- dconst

- dcon
- read
- write
- add/sub/mul/div/gt/gte/lt/lte/eq/not/and/or

Each of these nodes take 1 or more inputs and have 1 or more outputs, with the exception of both sink and source in which the former has 0 outputs and the latter has 0 inputs.

#### *3.2.1 merge*

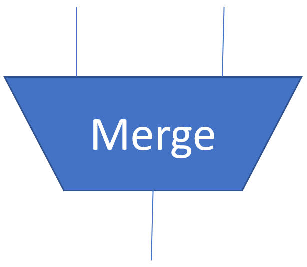

**Figure 2: A merge node with 2 inputs and 1 corresponding output**

A merge nodes takes n inputs and has 1 output. It picks the "leftmost" valid input and passes the data component of the channel through unless the downstream node is not ready. The precedence of the input nodes is determined by the

### 3.3 mergeChoice

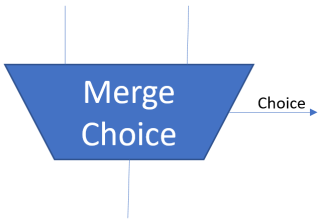

**Figure 3: A merge choice node with two inputs and two outputs**

A merge choice node functions the same as a merge node, while also outputting an additional value corresponding to the choice of the merge node as a function of the index of the input.

### 3.4 source

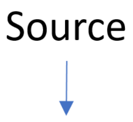

**Figure 4: A source node with one output**

A source node will always output a value based on some well defined input function. It has no inputs per se, but depending on the clock will output a value. It is effectively a theoretical model of a source of input tokens into a dataflow network, but its exact behavior depends on the implementation of the network.

### 3.5 sink

**Figure 5: A sink node with one input**

A sink node will take any input tokens and output them to some well defined behavior. Similar to a source node it is a theoretical model of an output that is undefined by the dataflow network but is defined elsewhere.

### 3.6 fork

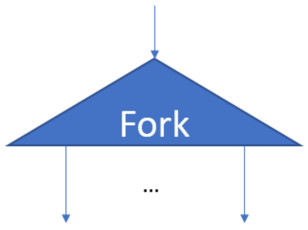

**Figure 6: A fork node with 1 input and multiple outputs**

A fork node will take in one value and reproduce the same value across multiple output channels. This is the main method of reproducing values on a network.

### 3.7 mux

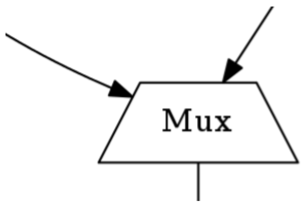

**Figure 7: A multiplexer node**

A multiplexer node will take 1 input and a choice and output that input on one of n channels depending on the choice. The choice must of type integer in order to be properly expressed.

### 3.8 demux

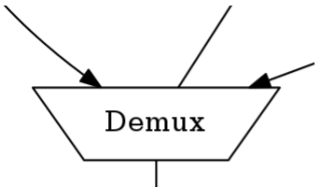

Figure 8: A demultiplexer node

A demux will do the reverse of a mux. It takes in multiple inputs, a choice, and output one of the input nodes depending on the choice. The choice must be an integer.

### 3.9 choice

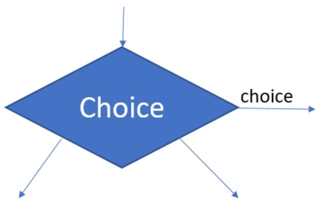

Figure 9: A choice node with a bit of information for the choice and two outputs depending on the choice.

The choice node will determine which output channel to choose and output the choice it made depending on the input value. This is a critical node for allowing conditional statements in the network.

### 3.10 func

A func node is an arbitrary node for any potential function that might be implemented on the network. This is defined using a special syntax and is implemented in the source language (in the case of this paper that is Haskell).

### 3.11 iconst

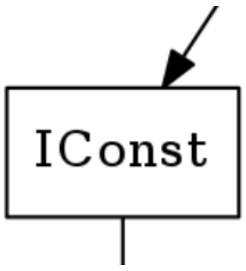

Figure 10: A constant constructor - similar to a source.

An iconst node will produce a constant value as long as the input of the node is valid. The output is always being supplied otherwise. This is only for primitive constant values.

### 3.12 dconst

A dconst, similar to an iconst, will produce a constant stream of constant values but of a custom type. These are typically defined as a sort of constructor for these objects with no input. The only input into these will dictate whether a value is produced, but not define the values inside of the constant.

### 3.13 dcons

A dcons, contrary to a dconst, does take input to define the fields of the custom type. Because of this there are often much more values that can be taken in to accomodate the custom type. This requires all values be ready for the custom type is created. A custom type is immutable upon initialization.

### 3.14 read

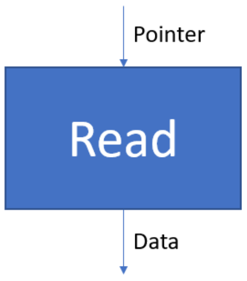

Figure 11: A read node takes in a pointer and outputs data

A read node is a special memory access node that will block until the memory is read. This is generally implemented in the form of a pipeline read to memory so as to have the fastest access time to memory. It is not possible to have as input anything other than a pointer, which means that some form of memory initialization needs to occur before a read node, whether that is in the form of a preinitialized memory block before the sink or a write node upstream in the network is dependent on the dataflow model. The data is thread safe because memory that has been read is immutable upon initialization. As long as the pointer is in flight or stored in memory the data will not be destroyed or modified.

### 3.15 write

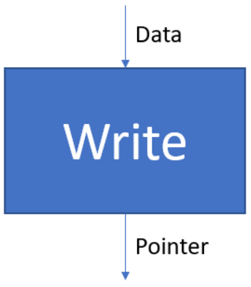

Figure 12: A write node will take as input data and output a pointer to memory where that data was stored.

A write node will take in data and write it to a memory storage device of some kind and output the pointer location of where that memory was written to. The data, once written, is immutable. Therefore it is safe to have those pointers in flight.

### 3.16 add/sub/mul/div/gt/gte/lt/lte/eq/not/and/or

Several basic primitive functions based on comparing or operating on integers or other primitive numeric types have been added to the dataflow model. These include functions such as addition, subtraction, etc. that will take some number of integers and, as long as the behavior is defined for several input values, will give the result. Subtraction, division, greater than, greater than equal, less than, less than equal, and equal are all defined for two input values. Addition, multiplication, and, and or are all defined for an arbitrary number of input values. Not is defined for only one input value.

## 4. GRAPH VISUALIZATION

In order to best represent the dataflow graphs we were producing from our compiler step we opted to use a graph visualization language GraphViz. This allowed us to define a network of nodes either by hand or as the output of our compiler step and automatically get an image of what the network looks like without drawing it by hand. An example graph visualization is here:

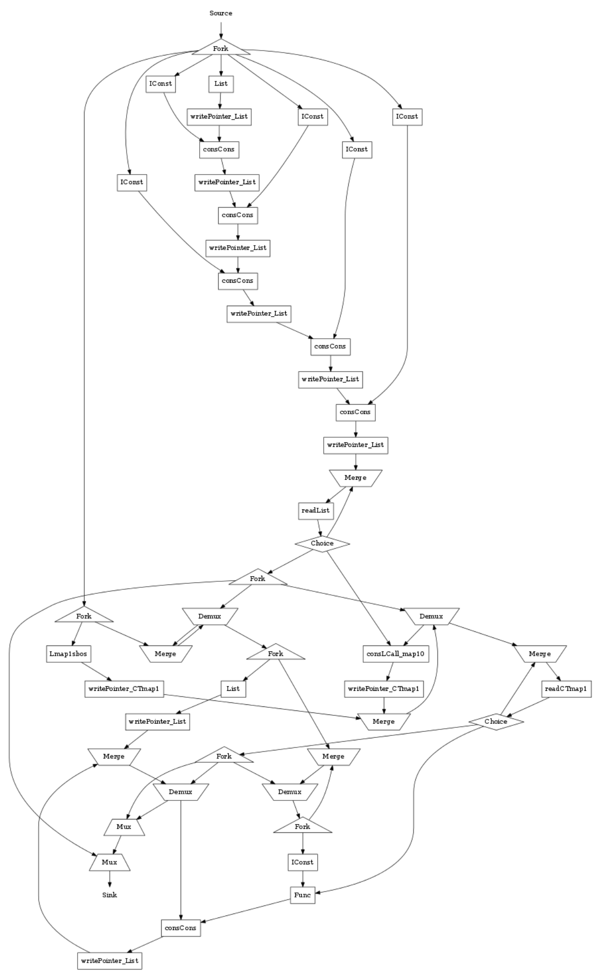

**Figure 13: A complex network generated from a series of instructions encoded in the flo language**

What is important to note from this network is that it was all auto generated from the IR of the program. This allows for more information when debugging a program within the flo programming language that otherwise wouldn't be as readily available to the programmer. A dataflow model for a programming language, similar to assembly, can be opaque to the programmer and therefore any graphical representation can assist in the development process.

## 5. GRAPH COLORING

The graph coloring algorithm is meant to separate different read and write nodes from one another to prevent memory conflicts. By coloring each identifier with a different color and having the different loads and stores parameterized by those colors, a more robust memory system arises where the memory accesses are successfully partitioned. Memory contention is one of the worst causes for latency within a dataflow network or a processing unit - and therefore any methodology to reduce the impact of these contentions is effective in speeding up an algorithm encoded in a network.

### 5.1 Overview

Some metrics for evaluating the effectiveness of a memory system include:

- Simulate a program and capture a memory trace
- Cycle accurate simulation of the dataflow graph and capture detailed contention analysis
- Abstract models of memory units and determine how much locality each portion has with one another

The issue with simulation of the program is that very frequently it does not capture the entire memory trace of a dataflow graph as the representation of a Haskell program is too high level. Cycle accurate simulation is a great way of getting results however it can be time consuming and only needs to be run once. Abstracting away a lot of the minutia of a given flo program is likely the most efficient way to determine if there will memory contention, but it can be difficult developing algorithms to abstractly compute the behavior of a network. This is when the graph coloring algorithm comes in.

### 5.2 Graph Coloring

The graph coloring algorithm is an algorithm for partitioning nodes in a graph such that nodes are given colors in a manner where no single node is touching another node of the same color. This algorithm is used for applications such as making a schedule, mobile radio frequency assignment, Sudoku solutions, and many more. The algorithm that will be used later on for coloring a complex flo graph will be the greedy implementation of the graph coloring algorithm. It follows the following format:

1. Color first vertex with first color
2. For each remaining V-1 vertices
   (a) Collect all colored vertices from current neighbors of vertex
   (b) If any color is not in the list of neighbor vertices - label it that color
   (c) Otherwise color the vertex a new color

### 5.3 Graph Coloring of a flo Dataflow Graph

The simplification of a flo graph was the trickiest step of designing this algorithm. Unfortunately there are many components in a flo graph that can cause delays or where this abstraction does not 100% apply to a given network - however this preliminary attempt at an algorithm will provide the necessary steps to providing a fairly robust sense of a valid algorithm.

First remove all nodes in a flo dataflow graph except for the memory access nodes, namely the reads and writes. Because memory in the flo language is immutable - meaning that once the data is written it will not be modified - this means that we only ever have to worry about concurrent reads/writes to different parts of memory. This means that a given memory unit - which might have 2 read-/write channels - will need to be specialized for a given memory type. The initial algorithm - once all extraneous nodes are removed - takes the write nodes and assigns to them a color given the type that they are writing.

We then take any downstream read nodes and assign them the same color as the write nodes that are upstream from them. For a given write followed by read combination - there should never be a write without a read of a corresponding type on the other end of a network. Once the preliminary coloring has been established - a memory unit is assigned to each write/read network according to the coloring of the nodes in the network. By assigning these separate memory units we ensure no contention between these units however it is possible that there are too many memory units for a more complex network. So to combat this we look for loops that only occur after some condition is met and combine the memory units in these loops with memory units that only exist outside of this loop.

### 5.4 Graph Coloring Example

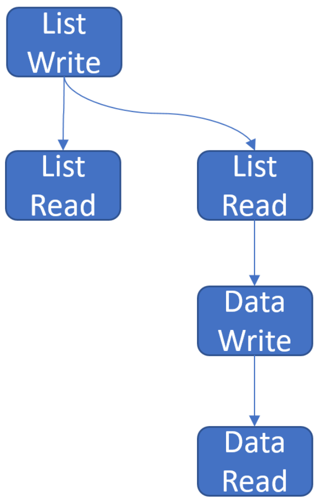

**Figure 14: An initial network where all the extraneous nodes have been removed**

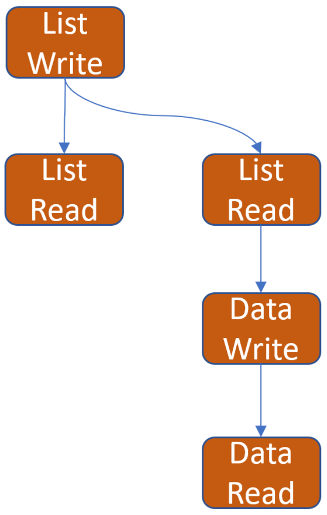

**Figure 16: After realizing the data cannot be written until the list is fully consumed - the partitioning combines the two types into one memory unit**

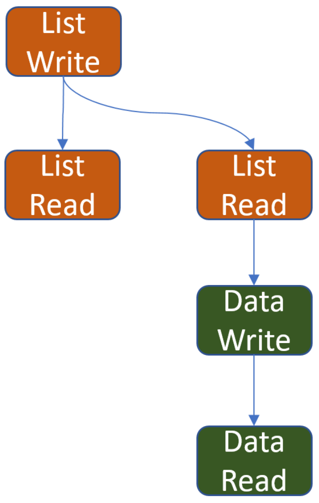

**Figure 15: An initial coloring of the memory nodes based on the type**

## 6. SIMULATION

In order to best implement a dataflow network a simulator would be necessary to test if the language describing the dataflow network made sense and if the dataflow networks behavior is well defined. Therefore a simulator creates a good debugging tool for the compiler designer to affirm that his implementation is correct. In the case of the flo language we have designed the architecture for a dataflow network featuring buffers, nodes, and memory operations. These are the basic primitive operations that define a dataflow network and with a well designed management framework will accurately simulate a dataflow network.

## 6.1 Nodes

For our preliminary results we designed nodes to be based off of C++ templated classes that contained specific methods that supported simulation. The basic primitives are Packet, Channel, Buffer, and Node. These are then used by the main simulation loop to generate a result for a given dataflow network. The program is dynamically generated from the flo compiler.

### *6.1.1 Packet*

A packet contains the messages that are passed between nodes over channels. It contains fields for both *ready* and *stop* as well as the *data* packet that is sent. This allows buffers to store the results from prior cycles and make sure the data holds between cycles. This design was chosen because it easily encapsulates a custom data type in the form of a template and also makes sure that nodes can send ready and stop signals in the form of a set message type.

```
template <typename T>
class Packet {
        public:
        bool ready;
        bool stop;
        T data;

        Packet() {
                this->data = 0;
                this->ready = false;
                this->stop = true;
        }

        Packet(bool ready, T data) {
                this->data = data;
                this->ready = ready;
                this->stop = !ready;
        }

};
```

### *6.1.2 Channel*

A channel was used primarily to allow messages to pass between nodes and buffers. This way values could be propagated throughout the network on a given cycle depending on whether the node was ready or not. The design of channels was chosen over direct message passing to allow for greater control over the debugging of the messages as well as making sure that a node was not called on more than once a cycle. The tick function is to note when a given cycle has passed and flush all of the channels.

```
template <typename T>
class Channel
{
        public:
        Channel();
        ~Channel();

        Packet<T> current;
        bool isReady();
        bool isValid();

        void setReady(bool isReady);
        void setValid(bool isValid);

        void tick() {
                isReady = false;
                isValid = false;
                packet = EmptyPacket<T>();
        }
};
```

### *6.1.3 Buffer*

Buffers are essentially memory units that allow for delays in input packets across multiple cycles. A zero buffer equates to a channel in that it directly passes the message to the next node. A buffer of size 1 will wait 1 cycle to pass the message it has to the next node. An infinite buffer will take as many packets as it is given and store them until the next node consumes all them - this is useful for testing whether or not a given network is running efficiently or not. Buffers were chosen as the ideal solution because it most closely represents what the hardware will look like when compiling the dataflow graph.

```
template<typename T>
class Buffer {
        private:
        vector<Buffer<T>> buff;
        public:
        Buffer() {
                buff = vector<Buffer<T>>(1);
        }

        Buffer(int size) {
                buff = vector<Buffer<T>>(size);
        }

        Packet getCurrent() {
                return buff[buff.size() - 1];
        }

        void clock() {
                for(int i = buff.size() - 1; i > 0; --i) {
                        buff[i] = buff[i - 1];
                }
                buff[0] = Packet<int>();
        }

        void addToBuff(Packet in) {
                buff[0] = in;
        }
};
```

### *6.1.4 Node*

We needed a flexible system for defining nodes - especially since they could perform a multitude of different functions. A Node could be any of the aforementioned nodes in section 3 where each of their behaviors are defined. This Node template is meant to serve as an interface to describe all future nodes that are implemented.

```
template <typename T>
class Node
{
        public:
        Node();
        ~Node();

        Packet<T> run();
};
```

## 6.2 Memory Operations

Memory is complex in the flo language because there is implicit garbage collection when the memory is no longer accessible. In this system each read and write node has a special implementation to share memory across the nodes by passing in a shared memory unit that can capture contention of resources. One of the most important aspects of this simulator is that it can verify whether or not contention will occur given a specific graph coloring. The given memory modules will be aware of the cycles as they occur and will trigger callbacks which will add the results of the memory to the channels when the delay for the memory unit as been met.

```
template <typename T>
class ReadNode : public Node
{
        private:
        Memory ref;
        public:
        ReadNode(Memory m) {
                ref = m;
        }

        Packet<T> readyPacket() {
                if(ref.isBusy())
                        return notReadyPacket;
                else
                        return readyPacket;
        }

        Packet<T> execute(Packet<int> readRequest) {
                Packet result = ref.request(readRequest.data);

                if(!result.ready) {
                        return notReadyPacket();

                }

        }
};

template <typename T>
class WriteNode : public Node
{
        private:
        Memory ref;
        public:
        WriteNode(Memory m) {
                ref = m;
        }

        Packet<T> readyPacket() {
                if(ref.isBusy())
                        return notReadyPacket;
                else
                        return readyPacket;
        }

        Packet<T> run(Packet<int> writeRequest) {
                Packet result =
                ref.writeRequest(writeRequest.data);

                if(!result.ready) {
                        return notReadyPacket();
                }

        }
};


template <typename T>
class Memory
{
        private:
        vector<int> pipeline;
        int pipelinesize;
        vector<T> data;
        bool accessed;
        public:
        Memory() {
                data = vector<T>();
                pipelinesize = 2;
                pipeline = vector<int>(pipelinesize);
                for(int i = 0; i < pipelinesize; ++i) {
                        pipeline[i] = -1;
                }

                accessed = true;
        }

        ~Memory();

        T read() {
                if(pipeline[pipelinesize - 1] == -1)
                return 0;
                else
                return data[]
        }

        Packet<bool> request(int address) {
                if(accessed) notReadyPacket();
                else
                pipeline[0] = address;

                return readyPacket;
        }

        bool isBusy() {
                return accessed;
        }

        void clock() {
                accessed = false;
                for(int i = pipeline.size() - 1; i > 0; --i
                        pipeline[i] = pipeline[i - 1];
                }
        }
};
```

## 6.3 Management Framework

The management framework takes all of the nodes in the current network and looks for any without any dependencies. This will likely begin with and constructors or source nodes in the network. The simulator will then propagate run functions down this queue and add additional nodes to the run queue as they have an inputted ready signal and execute them in order of observation in the network. A given node should not be run more than once within a given cycle. This effectively topologically sorts all the nodes in the network. This also allows each node to make requests to memory and to effectively determine whether or not there is memory contention within the network. A node can be determined to be ready when the input channel sends a message over to a buffer and then to a channel into a node. In the case of a zero buffer these messages will be sent instantly - but in the case of a memory buffer these messages will take as many cycles as the buffer has room for to get to the next node.

The network has the concept of quiescence, whereby if all nodes have not been able to execute work and the buffers have been unable to progress for N cycles, the entire simulation halts and the work is considered done. The result will be stored in memory and be displayed at the end of simulation.

## 6.4 Compiling from AST to Simulator

Compiling the AST to simulator has not been finished at the time of this writing. Instead a lot of discussion about how to perform this translation surrounded the idea of where to place buffers to allow the programs to run more efficiently. Placement of the buffers around memory units and any locations where they may be a cyclical dependency on results was the seemingly best heuristic - but it also seemed that placement of these buffers around two parallel pipelines would encourage a faster runtime. More experimentation would be required to find the optimal placement.

## 7. RESULTS AND ANALYSIS

### 7.1 Experiments

Not a lot of experimentation was performed in order to determine whether the simulation worked.

### 7.2 Next Steps

The next steps would be to implement the graph coloring algorithm and the simulator as described in order to get an accurate read as to whether the proposed schematic works. The biggest challenge of this system is that the flo language is very dynamic and therefore many heuristics that would work with a traditional von neumann architecture do not directly translate into a functional hardware architecture. Literature does not exist for methodologies for performing these translations and therefore more thought needs to go into this process.

## 8. CONCLUSION

In conclusion, a system for coloring memory reads and writes appears to be the best system for preventing contention as a first attempt at a heuristic. Designing a simulator to generate a dataflow network to tease out whether the methodology works in practice is the most important next step.

## 9. REFERENCES

[1] Bingyi Cao, Kenneth A. Ross, Martha A. Kim, and Stephen A. Edwards. Implementing latency-insensitive dataflow blocks. In *Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, 2015.

[2] Kuangya Zhai, Richard Townsend, Lianne Lairmore, Martha A. Kim, and Stephen A. Edwards. Hardware synthesis from a recursive functional language. In *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis*, CODES '15, pages 83–93, Piscataway, NJ, USA, 2015. IEEE Press.

# 10. CODE LISTING

## 10.1 Graph Visualization Translation Code

```haskell
{-
Module : Fhw.Pass.Dataflow.NodeTypes
Description : The abstract syntax of our dataflow language.

These type definitions form the abstract syntax for our dataflow
graphs. These graphs bridge the gap between a restricted Core program
and SystemVerilog.

A dataflow network is specified with a list of algebraic type definitions
and a list of nodes. The former encompasses the non-integer types that
data tokens may take, while the latter describes the topology and
functionality of the network.

Nodes are primitive functional units. Each node exposes
its connectivity to other nodes as well as the types of tokens it outputs.

At this level of abstraction, tokens passing between nodes
may be of two types: integers and algebraic types.

-}
module Fhw.Pass.Dataflow.NodeTypes (
  -- * Abstract Dataflow Syntax
  Dataflow(..),
  Node(..),
  Type(..),
  Op(..),
  Tydef(..),
  Codef(..),
  Func(..),

  -- * Pretty Printing
  pdataflow,

  -- * Dot Output
  toDotDataflow,

  -- * Correctness checker
  verify
) where

import Text.PrettyPrint
import Data.Char (toLower)
import Control.Monad
import Control.Applicative ((<$>))
import Data.List
import Data.Maybe
import Fhw.Core.Core

---- | Import dot library
--import Data.Graph.Inductive
--import Data.GraphViz
--import Data.GraphViz.Attributes.Complete
--import Data.GraphViz.Types.Generalised as G
--import Data.GraphViz.Types.Monadic
--import Data.Text.Lazy as L
--import Data.Word
--import WriteRunDot

-- | Types and topology of a dataflow network
data Dataflow = Dataflow [Tydef] [Node]

-- | A general node in our dataflow networks
data Node = Node [Id]         --input port identifiers String
                 Op           --Node type
                 [Id]         --output port destinations
  deriving (Eq,Ord)

type Id = String

-- | Different nodes and the types of their data tokens
```

```haskell
data Op = Merge Int Type
        | Fork Int Type
        | MergeChoice Int Type
        | Mux Type [Dcon]
        | Demux Type [Dcon]
        | IConst Int Int
        | DConst String
        | Func Func
        | Source Type
        | Sink Type
        | CallLock [Type]
        | ChoiceHold [Dcon]
        | Choice [(Dcon,[Int])]

  deriving (Eq,Ord)


-- | Primitive functions
data Func = Add | Sub  | Mul | Div
                | Lt   | Gt  | Eq
                | And  | Or  | Not
                | Read  Type Type --Memory operations specify input and output types
                | Write Type Type
                | Dcons String --name of constructor (arg types are inferred)
  deriving (Eq,Ord,Show)

-- Type definition representation mirrors the representation
-- used in FHW.Core (without type variables)
data Tydef = Tydef Tcon [Codef] deriving (Eq,Ord)
data Codef = Codef Dcon [Type]  deriving (Eq,Ord)


-- | Data token types
data Type = Int' Int    --Sized integer
          | Tycon String --Name of user-defined type
  deriving (Eq,Ord)

instance Show Dataflow where
  showsPrec _ d = shows (pdataflow d)

instance Show Tydef where
  showsPrec _ t = shows (ptdef t)

instance Show Codef where
  showsPrec _ c = shows (pcdef c)

instance Show Type where
  showsPrec _ t = shows (pTy t)

instance Show Node where
  showsPrec _ node = shows (pnode node)

instance Show Op where
  showsPrec _ op = shows (pOp op)

-- | Print dot representation of Dataflow instance
toDotDataflow :: Dataflow -> String
toDotDataflow (Dataflow _ nodes) = do
  nodesWithNames <- [getNodeNamesFrom nodes 0]
  "digraph Dataflow {\n" ++ (getEdgesFrom nodesWithNames nodesWithNames) ++ (getVerticesFrom nodesWithNames) ++ "\n

getNodeNamesFrom :: [Node] -> Int -> [(Node, String)]
getNodeNamesFrom [] _ = []
getNodeNamesFrom (node:nodes) count = [(node, (dotnodename node count))] ++ (getNodeNamesFrom nodes (count+1))

getEdgesFrom :: [(Node, String)] -> [(Node, String)] -> String
getEdgesFrom [] _ = ""
getEdgesFrom (((Node _ _ outputs), curNodename):xs) allNodes =
  (foldl (\s ((Node inputs _ _), nodename) -> s ++ (if (length $ intersect inputs outputs) > 0 then curNodename ++
  (getEdgesFrom xs allNodes)

getVerticesFrom :: [(Node, String)] -> String
getVerticesFrom [] = ""
getVerticesFrom (((Node _ op _), nodename):nodes) = (dotOp nodename op) ++ (getVerticesFrom nodes)

dotnodename :: Node -> Int -> String
dotnodename (Node _ (Merge _ _) _)         count = "Merge" ++ (show count)
dotnodename (Node _ (Fork  _ _) _)         count = "Fork" ++ (show count)
dotnodename (Node _ (MergeChoice _ _) _)   count = "MergeChoice" ++ (show count)
dotnodename (Node _ (Source _) _)          count = "Source" ++ (show count)
dotnodename (Node _ (Sink _) _)            count = "Sink" ++ (show count)
dotnodename (Node _ (Mux _ _) _)           count = "Mux" ++ (show count)
dotnodename (Node _ (Demux _ _) _)         count = "Demux" ++ (show count)
dotnodename (Node _ (IConst _ _) _)        count = "IConst" ++ (show count)
dotnodename (Node _ (DConst name) _)       count = "DConst" ++ name ++ (show count)
dotnodename (Node _ (Func (Add)) _)        count = "Add" ++ (show count)
dotnodename (Node _ (Func (Sub)) _)        count = "Sub" ++ (show count)
dotnodename (Node _ (Func (Mul)) _)        count = "Mul" ++ (show count)
dotnodename (Node _ (Func (Div)) _)        count = "Div" ++ (show count)
dotnodename (Node _ (Func (Lt))  _)        count = "Lt" ++ (show count)
dotnodename (Node _ (Func (Gt))  _)        count = "Gt" ++ (show count)
dotnodename (Node _ (Func (Eq))  _)        count = "Eq" ++ (show count)
dotnodename (Node _ (Func (And)) _)        count = "And" ++ (show count)
dotnodename (Node _ (Func (Or))  _)        count = "Or" ++ (show count)
dotnodename (Node _ (Func (Not)) _)        count = "Not" ++ (show count)
dotnodename (Node _ (Func (Read _ ty)) _)  count = "Read" ++ (show count) ++ (dottts ty)
dotnodename (Node _ (Func (Write _ ty)) _) count = "Write" ++ (show count) ++ (dottts ty)
dotnodename (Node _ (Func (Dcons ty)) _)   count = "Cons" ++ (show count) ++ ty
dotnodename (Node _ (CallLock _) _)        count = "CallLock" ++ (show count)
dotnodename (Node _ (ChoiceHold _) _)      count = "ChoiceHold" ++ (show count)
dotnodename (Node _ (Choice _) _)          count = "Choice" ++ (show count)

dottts :: Type -> String
dottts (Tycon name) = name
dottts (Int' size) = "Int_" ++ (show size)

dotOp :: String -> Op -> String
dotOp nodename (Merge _ _)          = nodename ++ "[label=\"Merge\",shape=invtrapezium];\n"
dotOp nodename (Fork  _ _)          = nodename ++ "[label=\"Fork\",shape=triangle];\n"
dotOp nodename (MergeChoice _ _)    = nodename ++ "[label=\"MergeChoice\",shape=trapezium];\n"
dotOp nodename (Source _)           = nodename ++ "[label=\"Source\",shape=plaintext];\n"
dotOp nodename (Sink _)             = nodename ++ "[label=\"Sink\",shape=plaintext];\n"
dotOp nodename (Mux _ _)            = nodename ++ "[label=\"Mux\",shape=trapezium];\n"
dotOp nodename (Demux _ _)          = nodename ++ "[label=\"Demux\",shape=invtrapezium];\n"
dotOp nodename (IConst _ _)         = nodename ++ "[label=\"IConst\",shape=rectangle];\n"
dotOp nodename (DConst name)        = nodename ++ "[label=\"" ++ name ++ "\",shape=rectangle];\n"
dotOp nodename (Func (Dcons ty))    = nodename ++ "[label=\"cons" ++ ty ++ "\",shape=rectangle];\n"
dotOp nodename (Func (Read _ ty))   = nodename ++ "[label=\"read" ++ (dottts ty) ++ "\",shape=rectangle];\n"
dotOp nodename (Func (Write _ ty))  = nodename ++ "[label=\"write" ++ (dottts ty) ++ "\",shape=rectangle];\n"
dotOp nodename (Func _)             = nodename ++ "[label=\"Func\",shape=rectangle];\n"
dotOp nodename (CallLock _)         = nodename ++ "[label=\"CallLock\",shape=rectangle];\n"
dotOp nodename (ChoiceHold _)       = nodename ++ "[label=\"ChoiceHold\",shape=diamond];\n"
dotOp nodename (Choice _)           = nodename ++ "[label=\"Choice\",shape=diamond];\n"

-- | Pretty print a Dataflow instance
pdataflow :: Dataflow -> Doc
pdataflow (Dataflow tdefs nodes) = text "Dataflow" $$
                                   braces (vcat (map ptdef tdefs)) $$
                                   braces (vcat (map pnode nodes))

ptdef :: Tydef -> Doc
ptdef (Tydef tcon cdefs) =
  text "data" <+> text tcon <+> char '='
  $$ indent (vcat $ punctuate (space <> char '|') $ map pcdef cdefs)

pcdef :: Codef -> Doc
pcdef (Codef dcon tys)  =
  text dcon <+> sep (map pTy tys)

pTy :: Type -> Doc
pTy (Tycon name) = text name
pTy (Int' size) = text "Int" <> char '_' <> int size

pnode :: Node -> Doc
pnode (Node inputs op outputs) =
  pPorts inputs <> char '.' <> pOp op <+>
  char '-' <> char '>' <+> pPorts outputs

pPorts :: [Id] -> Doc
pPorts = tupify

pOp :: Op -> Doc
pOp (Merge _ ty)             = text "Merge" <+> tySig ty
pOp (Fork  _ ty)             = text "Fork"  <+> tySig ty
pOp (MergeChoice _ ty)       = text "Mergechoice" <+> tySig ty
pOp (Source ty)              = text "Source" <+> tySig ty
pOp (Sink ty)                = text "Sink"   <+> tySig ty
pOp (Mux ty dcons)           = text "Mux"    <> tupify dcons <+> tySig ty
pOp (Demux ty dcons)         = text "Demux"  <> tupify dcons <+> tySig ty
pOp (IConst num size)        = text "Iconst" <> parens (int num)
                               <+> tySig (Int' size)
pOp (DConst name)            = text "Dconst" <> parens (text name)
pOp (Func (Dcons name))      = text "Func" <> parens (text name)
pOp (Func (Read tIn tOut))   = text "Func" <> memOpTy "read" tIn tOut
pOp (Func (Write tIn tOut))  = text "Func" <> memOpTy "write" tIn tOut
pOp (Func f)                 = text "Func" <> parens (text $ map toLower $ show f)
pOp (CallLock inputTys)      = text "CallLock" <>
                               parens (commaList $ map tySig inputTys)
pOp (ChoiceHold dcons)       = text "ChoiceHold" <> tupify dcons
pOp (Choice dconFields)      = text "Choice" <>
                               parens (commaList $ map pFields dconFields)
  where
    pFields (dcon,nums) = text dcon <> brackets (cat $
                                       punctuate comma $
                                       map int nums)

-- Generate a type signature for a Read or Write node
memOpTy :: String -> Type -> Type -> Doc
memOpTy version tIn tOut = parens (text version <+> tySig tIn
                                   <+> char '-' <> char '>' <+> pTy tOut)

tySig :: Type -> Doc
tySig ty = colon <> colon <+> pTy ty

commaList :: [Doc] -> Doc
commaList = cat . punctuate comma

tupify :: [String] -> Doc
tupify = parens . commaList . map text

indent :: Doc -> Doc
indent = nest 2


-- | Sanity check a dataflow network. If all checks pass return the netework,
-- otherwise return the first failed check.
verify :: Dataflow -> Either String Dataflow
verify dflow@(Dataflow types nodes) = do
  uniquePorts nodes                 --Every port name is used exactly twice
  mapM_ (typeExistence types) nodes --All data token types have been defined
  mapM_ (opCheck types) nodes       --Verify individual node properties
  --typeTopology types nodes  --TODO: Typecheck ports
  return dflow

-- | Each port name should appear exactly twice in a Dataflow network:
-- once as an output port, and once as an input port for a different
-- node (we don't currently allow direct feedback loops).
uniquePorts :: [Node] -> Either String ()
uniquePorts nodes
  | isJust doubleIn  = Left $ showJust "Port appears twice as input: " doubleIn
  | isJust doubleOut = Left $ showJust "Port appears twice as output: " doubleOut
  | isJust feedbacks = Left $ showJust "Feedback node: " feedbacks
  | isJust mismatch  = Left $ showJust "Not specified as input and output " mismatch
  | otherwise = Right ()
  where doubleIn  = finder (/=1) ins
        doubleOut = finder (/=1) outs
        mismatch  = finder (==1) (ins ++ outs)
        finder f  = find (f . length) . group . sort
        feedbacks = find (\(Node is _ os) -> not $ null $ intersect is os) nodes
        ins = concatMap (\(Node is _ _) -> is) nodes
```

```haskell
        outs = concatMap (\(Node _ _ os) -> os) nodes
        showJust str val = str ++ show (fromJust val)

-- | Ensure that every type and data constructor has a definition,
-- and that sets of data constructors all come from the same definition
typeExistence :: [Tydef] -> Node -> Either String ()
typeExistence types (Node _ op _) = case op of
  Merge _ ty          -> isDefined ty
  Fork _ ty           -> isDefined ty
  MergeChoice _ ty    -> isDefined ty
  Source ty           -> isDefined ty
  Sink ty             -> isDefined ty
  ChoiceHold constrs -> foldM_ sameTy Nothing constrs
  Mux ty constrs      -> isDefined ty >> foldM_ sameTy Nothing constrs
  Demux ty constrs    -> isDefined ty >> foldM_ sameTy Nothing constrs
  CallLock tys        -> mapM_ isDefined tys
  Choice patterns     -> foldM_ sameTy Nothing (map fst patterns)
  DConst dcon         -> void $ findDef dcon types
  Func (Dcons dcon)   -> void $ findDef dcon types
  Func (Read t1 t2)   -> mapM_ isDefined [t1,t2]
  Func (Write t1 t2) -> mapM_ isDefined [t1,t2]
  _                   -> return () -- IConst and other Func variants
  where
    tyConstrs = map (\(Tydef tcon _) -> tcon) types

    --Check if type constructor is defined
    isDefined (Int' _   )     = return ()
    isDefined (Tycon tcon)
      | tcon `elem` tyConstrs = return ()
      | otherwise             = Left $ "Type '" ++ tcon ++ "' undefined"

    --Check if a set of data constructors belong to same type definition
    sameTy Nothing     dcon = Just <$> findDef dcon types
    sameTy (Just tcon) dcon = do
      tcon' <- findDef dcon types
      if tcon == tcon'
        then return $ Just tcon
        else Left $ "Constructor " ++ dcon ++
                    " undefined in type "  ++ tcon

    -- Find the type containing a defintion for dcon
    findDef dcon [] = Left $ "Constructor " ++ dcon ++ " undefined"
    findDef dcon (Tydef tcon cdefs : rest) =
      if dcon `elem` map dcDefs cdefs
       then return tcon
       else findDef dcon rest
      where dcDefs (Codef dc _) = dc

-- | Check each node's individual properties for consistency
opCheck :: [Tydef] -> Node -> Either String ()
opCheck types node@(Node ins op outs) = case op of
  -- One output, inP or outP field counts number of inputs
  Merge inP _       -> check "Merge" node (portSizes outs ins inP)
  Fork outP _       -> check "Fork"  node (portSizes ins outs outP)

  -- One output, number of inputs is one more than number of choices
  Mux _ constrs     -> check "Mux" node (length ins == length constrs + 1
                                         && singlePort outs)
  -- Two inputs, number of outputs equals number of choices
  Demux _ constrs   -> check "Demux" node (length outs == length constrs
                                           && length ins == 2)

  -- Two outputs, inP field counts number of inputs
  MergeChoice inP _ -> check "MergeChoice" node (length outs == 2
                                                 && length ins == inP)

  -- Two inputs, one output
  ChoiceHold _      -> check "ChoiceHold" node (length ins == 2 &&
                                                singlePort outs)

  -- One input, one output, enough bits to express constant
  IConst num bits   -> check "IConst" node $ bitWidth num bits &&
                                             singlePort ins    &&
                                                  singlePort outs
  -- One input, one output
  DConst _          -> check "DConst" node $ singlePort ins && singlePort outs

  -- No inputs, one output
  Source _          -> check "Source" node $ null ins && singlePort outs
  -- No outputs, one input
  Sink _            -> check "Sink"   node $ null outs && singlePort ins

  -- One more input than number of types (each type is an input to the locked
  -- function, but we also have a polymorphic input as an unlock signal),
  -- number of outputs equals number of types
  CallLock tys      -> check "CallLock" node (length ins == length tys + 1 &&
                                              length outs == length tys)

  -- one output, input number matches function
  Func func         -> check "Func" node (singlePort outs) >>
                       funcCheck func node

  -- One input, the number of outputs is one more than
  -- total fields specified, no pattern index is out of bounds
  Choice patterns   -> check "Case" node $ and $ singlePort ins :
                       (length outs == 1 + totalOutFields patterns) :
                       map (patternBounds $ getCDefs $ map fst patterns) patterns
  where
    check name n bl = if bl
                        then Right ()
                        else Left $ errMsg (name ++ ": " ++ show n)
    errMsg n = "A " ++ n ++ " node's definition is inconsistent"

    portSizes single many size = singlePort single && length many == size
    singlePort ports = length ports == 1
    bitWidth val bits = val <= 2 ^ bits - 1

    totalOutFields patterns = sum $ map (length . snd) patterns
    patternBounds cdefs (dcon,fields) =
      let (Codef _ tys) = fromJust $ find ((==dcon) . getDcon) cdefs
          highField  = maximum fields
          lowField   = minimum fields
      in null fields ||                --No fields used
         lowField >= 0 &&              --Lowest index is non-negative
         length tys >= (highField + 1) --Highest index is within bound

    getCDefs [] = error "Case node with no patterns"
    getCDefs (dcon:_) = fromJust $ find (elem dcon . map getDcon) $
                        map getVariants types

    getVariants (Tydef _ cdefs) = cdefs
    getDcon (Codef dcon _) = dcon

    funcCheck Add          n = twoInCheck "Add"   n
    funcCheck Sub          n = twoInCheck "Sub"   n
    funcCheck Mul          n = twoInCheck "Mul"   n
    funcCheck Div          n = twoInCheck "Div"   n
    funcCheck Lt           n = twoInCheck "Lt"    n
    funcCheck Gt           n = twoInCheck "Gt"    n
    funcCheck Eq           n = twoInCheck "Eq"    n
    funcCheck And          n = twoInCheck "And"   n
    funcCheck Or           n = twoInCheck "Or"    n
    funcCheck (Read _ _)   n = oneInCheck "Read"  n
    funcCheck (Write _ _)  n = oneInCheck "Write" n
    funcCheck Not          n = oneInCheck "Not"   n
    funcCheck (Dcons dcon) n =
     let (Just (Codef _ tys)) = find ((== dcon) . getDcon) $ getCDefs [dcon]
     in check "Constructor" n $ not (null ins) && --no constant constructors
                                length tys == length ins

    twoInCheck str n = check str n $ length ins == 2
    oneInCheck str n = check str n $ singlePort ins

--typeTopology = undefined
```

## 10.2 Simulator Defined Code

```cpp
#include <iostream>
#include <vector>

using namespace std;

template <typename T>
class Packet {
public:
        bool ready;
        bool stop;
        T data;

        Packet() {
                this->data = 0;
                this->ready = false;
                this->stop = true;
        }

        Packet(bool ready, T data) {
                this->data = data;
                this->ready = ready;
                this->stop = !ready;
        }

        Packet(const Packet &obj) {
                this->data = obj.data;
                this->ready = obj.ready;
                this->stop = obj.stop;
        }
};

class Channel
{
private:
        bool isReady;
        bool isValid;
public:
        Channel();
        ~Channel();

        Packet<T> current;
        bool isReady();
        bool isValid();

        void setReady(bool isReady);
        void setValid(bool isValid);

        void tick() {
                isReady = false;
                isValid = false;
        }
};

class Buffer {
        private:
                vector<Buffer> buff;
        public:
                Buffer() {
                        buff = vector<Buffer>(1);
                }

                Buffer(int size) {
                        buff = vector<Buffer>(size);
                }

                Packet getCurrent() {
                        return buff[buff.size() - 1];
                }

                void clock() {
                        for(int i = buff.size() - 1; i > 0; --i) {
                                buff[i] = buff[i - 1];
                        }
                        buff[0] = Packet<int>();
                }

                void addToBuff(Packet in) {
                        buff[0] = in;
                }

                //Need to capture no change
};

class Add1
{
private:

public:
        Add1() {}
        ~Add1(){}

        Packet<T> isReady() {
                return readyPacket;
        }

        Packet execute(Packet<int> in) {
                return Packet<int>(in.ready, in.data + 1);
        }
};

template <typename T>
class Memory
{
private:
        vector<int> pipeline;
        int pipelinesize;
        vector<T> data;
        bool accessed;
public:
        Memory() {
                data = vector<T>();
                pipelinesize = 2;
                pipeline = vector<int>(pipelinesize);

                for(int i = 0; i < pipelinesize; ++i) {
                        pipeline[i] = -1;
                }

                accessed = true;
        }

        ~Memory();

        T read() {
                if(pipeline[pipelinesize - 1] == -1)
                        return 0;
                else
                        return data[]
        }

        Packet<bool> request(int address) {
                if(accessed) 1; //Contention  return notReadyPacket?
                else
                        pipeline[0] = address;

                return readyPacket;
        }

        bool isBusy() {
                return accessed;
        }

        void clock() {
                accessed = false;
                for(int i = pipeline.size() - 1; i > 0; --i) {
                        pipeline[i] = pipeline[i - 1];
                }
        }
};

template <typename T>
class ReadNode
{
private:
        Memory ref;
public:
        ReadNode(Memory m) {
                ref = m;
        }

        Packet<T> readyPacket() {
            if(ref.isBusy())
                    return notReadyPacket;
            else
                    return readyPacket;
        }

        Packet<T> execute(Packet<int> readRequest) {
                Packet result = ref.request(readRequest.data);

                if(!result.ready) {
                        return notReadyPacket();

                }

        }
};

template <typename T>
class Merge
{
private:

public:

        Packet<T> isReady() {
                return readyPacket;
        }

        Packet<T> execute(vector<Packet<T>> inputs) {
                for(Packet<T> & p : inputs) {
                        if(p.ready) {
                                return p;
                        }
                }
                return Packet<T>();
        }
};

class Merge
{
private:
        vector<Channel<T>> upstreams;
        Channel<T> output;
public:
        Merge();
        ~Merge();

        bool canInput() {

        }

        bool canOutput() {


        }

        bool canExecute() {
                bool isInputReady = false;
                for (std::vector<T>::iterator i = upstreams.begin(); i != upstreams.end(); ++i)
                {
                        isInputReady |= i.get().hasToken();
                }

                bool isOutputReady = out.canReceiveToken();

                return isInputReady && isOutputReady;
        }

        void execute() {

        }
};

class Demux
{

private:
        vector<Channel<T>> downstreams;
        Channel<T> upstream;
        Channel<int> select;
public:
        Demux();
        ~Demux();

        bool canExecute() {
                bool canSelect = select.hasToken();
                Channel<T> & downstreamChannel = downstreams[select.getData()];

        }
};

class Fork
{
private:
        vector<Channel<T>> neighbors;
        Channel<T> upstream;
public:
        Fork();
        ~Fork();

        bool readyToExecute() {

                return upstream.hasToken();
        }

        void execute() {
                Packet<T> in = upstream.current();

                if(!in.ready) {
                        neighbors[i].put(notReadyPacket);
                } else {
                        for (int i = 0; i < neighbors.size; ++i)
                        {
                                neighbors[i].put(upstream.current());
                                /* code */
                        }
                }


        }
};

int main(int argc, char** argv) {
        Add1 n;
        Buffer b;

        b.addToBuff(Packet<int>(true, 1));


        while(not quiescefd) {
                if(n.isReady()) {
```

```
                                in = b.getCurrent();
                                out = n.execute(in);

                                b.addToBuff(out);
                        } else {
                                //Pass
                        }

                        for(buf b : buffers) {
                                b.clock();

                        }

                        for(Memory m : memories) {
                                m.clock();
                        }
        }


        //Fork node with 3 CHANNELS
        //

        Packet<int> out = n.execute(b.getCurrent());

        //Initialize network

        //Propogate readys down (checking if the node is ready to receive as well)

        //Calculate values until all nodes are not ready

        //Clock all buffers and memorys

        //Repeat calculation until there is no change


        cout << "Done" << endl;
}

template <typename T>
class Node
{
public:
        Node();
        ~Node();


};

template <typename T>
class Fork : public Node
{
private:
        std::vector<Channel<T>> downstreams;
        Channel<T> upstream;
        std::vector<bool> haveWritten;
public:
        Fork() {

        }
        ~Fork();


};



/* EXAMPLE CODE */


// In the case of Fork
//         pass not ready up if not ready but send data to the people downstream that are ready
//         Remember who you sent it to to send it down
```

## 10.3 Python Simulator Attempt Main

```python
class Node:
    """docstring for Node"""
    def __init__(self):
        pass

    def enqueue(data, ready, port):
        pass

    def process():
        pass

    def update_output():
        pass

    def get_output():
        pass

    def isReady():
        pass

class Memory:
    def __init__(self):
        self.contentions = []
        self.nextPtr = 0
        self.data = []
        self.

    def addContention(self, ptr):
        self.contentions.append(data)

    def readValue(self, ptr):
        return self.data[ptr]

    def writeValue(self, data):
        self.data.append(data)
        self.nextPtr += 1
        return self.nextPtr - 1

class ReadNode(Node):

    def __init__(self, memory):
        super(Node, self).__init__()
        self.memory = memory
        self.ready = True
        self.ptr = None
        self.queue = None
        self.nextOutput = None

    def enqueue(ptr, ready, port):
        if self.ready:
            self.queue = ptr
        else:
            self.memory.addContention(ptr)
        self.ready = False

    def processData(self):
        self.nextOutput =
        ↪  self.memory.readValue(self.ptr)
        self.ready = True

    def update_output(self):
        self.output = self.nextOutput

    def isReady(self):
        return self.ready

    def get_output(self):
        return self.output

nodes = []

def setup_layout():

```

```

def main():
    setup_layout()
    while simulation_has_not_quiesced():
        run_simulation()
    print_contentions()
```

## 10.4 Python MapReturn Example

```
DELAY = 2

class Node():
    def __init__(self):
        self.ready = False
        self.stop = False

    def isReady(self):
        return self.ready

    def stop(self):
        return self.stop

    def hasQuiesced(self):
        return True

class Stack():
    C0 = 0
    C1 = 1

    def __init__(self, type, ):


class Memory():
    def __init__(self, type, delay=2):
        self.type = type
        self.busy = False
        self.delay = delay

        self.pipeline = {}
        self.pipelineSize = delay

        self.memory = []
        self.ptr = 0

    def read(self, ptr):
        if ptr in self.pipeline:



    def write(self, data):
        self.memory.append(data)
        self.ptr += 1
        return self.ptr

    def clock(self):
        for ptr, remaining in
        ↪  self.pipeline.copy().items():



#Initialize Nodes

stackMemory = Memory("StackMemory", Stack)
listMemory = Memory("ListMemory", List)

g = In("g", g)
lp = In("lp", ListPointer)

#Map Nodes
C0 = Produce("C0", Stack.C0)
stackWrite1 = Write("stackWrite1", stackMemory)
stackWrite2 = Write("stackWrite2", stackMemory)
spMerge = Merge("spMerge")

```

```python
demuxConsNil = Demux("demuxConsNil")
C1 = Produce("C1", Stack.C1)

gFork = Fork("gFork", 2)
gMerge = Merge("gMerge")

demuxConsNilG("demuxConsNilG")

nil = Produce("Nil", List.Nil)
listWrite1 = Write("listWrite1", listMemory)

lpMerge = Merge("lpMerge")
listRead = Read("listRead", listMemory)
nilConsChoice = Choice("nilConsChoice", List)
choiceFork = Fork("choiceFork", 2)

#Cont nodes
lpMerge2 = Merge("lpMerge2")
spMerge2 = Merge("spMerge2")

stackRead = Read("stackRead", stackMemory)
C0C1Choice = Choice("C0C1Choice", Stack)

f = Func("f", fdef)
C1C0Demux = Demux("C1C0Demux")
listCons = Cons("listCons", List.C1)
listWrite2 = Write("listWrite2", listMemory)

#Out nodes
result = Out("result")

#Link Nodes
graph = Graph()
graph.clink(g, gFork)
graph.clink(gFork, [C0, gMerge])
graph.clink(lp, lpMerge)
graph.clink()

g.setOutputs([C0, gMerge])
lp.setOutputs([lpMerge])

C0.setInputs([g])
C0.setOutputs([stackWrite1])

stackWrite1.setInputs([C0])




#Initialize Graph
#Set initial values on the transitions
#Retrieve all outputs and pass until quiescense
↪  is reached
#Where would buffering go?
#What would a valid pipeline look like?
```

## 10.5 Python Logic Node Definitions

```python
"""

   A basic digital logic simulator.

   The circuits passed to the simulator may use 4
↪  primitives: and,
   or, not, and flipFlop.

   This program is an exercise to help me figure
↪  out how to simulate
   our dataflow graphs.

"""

#-------------------------------------------------
#-- Shallow embedding used by the simulator
```

```python
#-----------------------------------------------
#-- | Given an initial state and a stream of inputs, delay the stream
#--   by a cycle.
def flipFlop(init, inputs):
    return inputs.insert(0, init)

# def traffic(reset):
#     red = flipFlop(True, [Or(x,y) for x,y in zip(reset, yellow)])
#     notreset = map(lambda x:not x, reset)
#     yellow = flipFlop(False, [And(x,y) for x,y in zip(notreset, green)])
#     green = flipFlop(False, [And(x,y) for x,y in zip(red, notreset)])
#     return zip(red, yellow, green)

#----------------------------------------------
#-- Deep embedding for input network
#----------------------------------------------

#-- Each node type specifies the source of its input. FlipFlop
#-- also has a boolean indicating its initial output. We include a Var
#-- constructor so we can name the sources of the network.

class Packet():
    def __init__(self, ready=False, data=None):
        self.ready = ready
        self.data = data

    def getData(self):
        return self.data

    def isReady(self):
        return self.ready

class Node():
    def __init__(self):
        pass

    def execute(self):
        return None

class And(Node):
    def __init__(self, input1, input2, graph):
        super(And, self).__init__()
        self.input1 = input1
        self.input2 = input2
        self.graph = graph

    def execute(self):
        input1 = self.graph.getOutputFor(self.input1)
        input2 = self.graph.getOutputFor(self.input2)
        return input1 and input2

class Or(Node):
    def __init__(self, input1, input2, graph):
        super(Or, self).__init__()
        self.input1 = input1
        self.input2 = input2
        self.graph = graph

    def execute(self):
        input1 = self.graph.getOutputFor(self.input1)
        input2 = self.graph.getOutputFor(self.input2)
        return input1 or input2
```

```python

class Not(Node):
    def __init__(self, input1, graph):
        super(Not, self).__init__()
        self.input1 = input1
        self.graph = graph

    def execute(self):
        output =
        ↪  self.graph.getOutputFor(self.input1)
        return not output

# class FlipFlop(Node):
#     def __init__(self, input1, graph):
#         super(FlipFlop, self).__init__()
#         self.input1 = input1
#         self.previousOutput = False
#         self.graph = graph
#
#     def execute(self):
#         output =
↪  self.graph.getOutputFor(self.input1)
#         o

class Var(Node):
    def __init__(self, name, initialValue,
    ↪  graph):
        super(Var, self).__init__()
        self.name = name
        self.initialValue = initialValue
        self.graph = graph

    def setInput(self, initialValue):
        self.initialValue = initialValue

    def execute(self):
        return self.initialValue

    def getName(self):
        return self.name

class Graph():
    def __init__(self):
        self.nodes = {}
        self.inputs = {}
        self.outputs = {}

    def addNode(self, index, node,
    ↪  initialOutput=False):
        self.nodes[index] = node
        self.outputs[index] = initialOutput

    def getOutputsFor(self, index):
        return self.outputs[index]

    def executeAll(self):
        for index, node in self.nodes.items():
            self.outputs[index] = node.execute()

    def addInput(self, index, node):
        if not isinstance(node, Var):
            raise "Ya done fucked up"
        self.nodes[index] = node
        self.inputs[node.getName()] = node

    def setInputValue(self, name, value):
        self.inputs[name].setInput(value)


def initHalfAdder():

    halfAdder = Graph()

    A = Var("A", True, halfAdder)
    B = Var("B", True, halfAdder)

    halfAdder.addNode(1, Or(2, 3, halfAdder))
    halfAdder.addNode(2, And(4, 5, halfAdder))
    halfAdder.addNode(3, And(6, 7, halfAdder))
    halfAdder.addNode(4, A)
    halfAdder.addNode(5, Not(7, halfAdder))
    halfAdder.addNode(6, Not(4, halfAdder))
    halfAdder.addNode(7, B)

    return halfAdder

#Given a set of input values and a netlist, simulate a cycle of the netlist's
#operation on those inputs.

def simulate(values, graph):
    for name, val in values:
        graph.setInputValue(name, val)

    graph.executeAll()



#
#
#
#     initValues =
# simulate :: [(String,Bool)] -> Graph -> [(Int,Node)]
# simulate values (Graph end nodes) = startGraph
#   where
#     initValues = catMaybes £ setInputs values
#     startGraph = filter (\x -> fst x `notElem` map fst initValues) nodes
#     --outputs = walkGraph initValue startGraph
#     --Replace each Var node with it's initial boolean value
#     setInputs [] = []
#     setInputs ((name,val):rest) = let node = find ((==) (Var name) . snd) nodes
#                                   in fmap (\(num,_) -> (num,val)) node : setInputs rest
```

## 10.6 Python MapReturnLib

```python
# LANGUAGE DeriveDataTypeable
import NodeLib
import Memory

#INS/OUTS
caseList_names = ["_c1","_c0"]
caseList_Ins = [[1,2]]

#Cons has an AND-firing rule on its inputs
cons_Ins = [[1, 2]]

#caseList doesn't have any output sets with AND-firing rules
caseList_Outs = []

caseCont_names = ["_x","_sp","_c0c1"]
#Set of outputs, keyed by index, with an AND-firing rule
caseCont_Outs = [[1,2,3]]

#What the original data types were
# data Cont = C2 | C1 Int (Pointer Cont) deriving (Show,Eq,Typeable)
# data List = Empty | Cons Int (Pointer List) deriving (Show,Eq,Typeable)

#Define Cont type
```

```
class Cont():
    C1, C2 = 0, 1

    def __init__(self, type, data=None):
        self.type = type
        self.data = data

    def getType(self):
        return self.type

    def getData(self):
        return self.data

class Pointer():
    def __init__(self, pntr):
        self.pntr = pntr

    def getPntr(self):
        return self.pntr

##################
# NODE FUNCTIONS #
##################

def diffMaybes(a, b):
    return (a is None and b is not None) or (a is not None and b is None)

#Construct a continuation
# buildCont val k = C1 val (Pointer k)
def buildCont(val, k):
    return Cont(Cont.C1, (val, k))

# cons :: Maybe Int -> Maybe (Pointer List) -> Bool -> (Maybe List,Bool,Bool)
# cons value pntr stop = (dOut,stop1,stop2)
#   where
#     dOut = pure Cons <*> value <*> pntr
#     stop1 = stop || isNothing pntr
#     stop2 = stop || isNothing value
def cons(value, pntr, stop):
    dOut = pure(Cons(value, pntr))
    stop1 = stop or pntr is None
    stop2 = stop or value is None
    return dOut, stop1, stop2


###What was get input
# -- | Construct input for the mapReturn function
# getInput size = Pointer £ foldr buildCont C2 [size,(size-1)..1]
# def getInput(size):
#     return Pointer(reduce(buildCont, ))
# All this effectively does is initialize memory
def initializeMapReturnMemory(memory, size):
    pntr = memory.addItems(range(size, 1))
    return pntr

# caseList :: Maybe (Pointer List) -> Maybe Cont -> Bool -> Bool ->
#             (Maybe (Pointer List),Maybe (Pointer List), Bool,Bool)
# caseList ptr select stopIn1 stopIn2 = (c1,c2,ptrStop,selectStop)
#   where
#     (c1,c2) = caseListData ptr select
#     ptrStop = genPtrStop select stopIn1 stopIn2
#     selectStop = genSelectStop ptr select stopIn1 stopIn2
#
#     genPtrStop Nothing         _     _     = True
#     genPtrStop (Just (C2    )) _     True  = True
#     genPtrStop (Just (C1 _ _)) True  _      =
↪  True
#     genPtrStop _               _     _      =
↪  False
#
#     genSelectStop Nothing _               _
↪  _     = True
#     genSelectStop _       (Just (C2    )) _
↪  True  = True
#     genSelectStop _       (Just (C1 _ _)) True
↪  _     = True
#     genSelectStop _       _               _
↪  _     = False
#
#
#     caseListData (Just _) (Just (C1 _ _)) =
↪  (ptr,Nothing)
#     caseListData (Just _) (Just (C2    )) =
↪  (Nothing,ptr)
#     caseListData _        _               =
↪  (Nothing,Nothing)
def caseList(ptr, select, stopIn1, stopIn2):
    def genPtrStop(select, stopIn1, stopIn2):
        return (select is None) or \
               (isinstance(select, Cont) and
               ↪  select.getType() == Cont.C2
               ↪  and stopIn2) or \
               (isinstance(select, Cont) and
               ↪  select.getType() == Cont.C1
               ↪  and stopIn1)

    def genSelectStop(ptr, select, stopIn1,
    ↪  stopIn2):
        return (ptr is None) or \
               (isinstance(select, Cont) and
               ↪  select.getType() == Cont.C2
               ↪  and stopIn2) or \
               (isinstance(select, Cont) and
               ↪  select.getType() == Cont.C1
               ↪  and stopIn1)

    def caseListData(ptr, select):
        if ptr is not None and isinstance(select,
        ↪  Cont) and select.getType() ==
        ↪  Cont.C2:
            return (ptr, None)
        elif ptr is not None and
        ↪  isinstance(select, Cont) and
        ↪  select.getType() == Cont.C1:
            return (None, ptr)
        else:
            return (None, None)

    c1, c2 = caseListData(ptr, select)
    ptrStop = genPtrStop(select, stopIn1,
    ↪  stopIn2)
    selectStop = genSelectStop(ptr, select,
    ↪  stopIn1, stopIn2)
    return c1, c2, ptrStop, selectStop



# fNode :: Maybe Int -> Bool -> (Maybe Int,Bool)
# fNode input stop = (fmap (*2) input, stop)
def fNode(input, stop):
    return map(lambda x: x * 2, input), stop

# caseCont :: Maybe Cont -> Bool -> Bool -> Bool
↪  -> Maybe [Bool]
#           -> (Maybe Int, Maybe (Pointer Cont),
↪  Maybe Cont,Bool,Maybe [Bool])
# caseCont val dataStop ptrStop choiceStop st =
↪  (dOut,spOut,choice,stopOut,nextState)
```

```
#   where
#     dOut   = makeOut 0 £ caseData val
#     spOut  = makeOut 1 £ casePtr  val
#     choice = makeOut 2 val
#
#     makeOut index v = if not (bits !! index)
#                       then v else Nothing
#
#     --previous state bits
#     bits@[b1,b2,b3] = if isNothing st
#                         then replicate 3 False
#                         else fromJust st
#
#     stopOut = if isNothing st
#                 then dataStop   || ptrStop ||
↪   choiceStop
#                 else dataStop   && not b1 ||
#                      ptrStop    && not b2 ||
#                      choiceStop && not b3
#
#     caseData val@(Just (C1 x _)) = Just x
#     caseData _ = Nothing
#
#     casePtr val@(Just (C1 _ k)) = Just k
#     casePtr _ = Nothing
#
#     nextState = let bits = [isJust dOut   &&
↪   not dataStop   || b1
#                            ,isJust spOut  &&
↪   not ptrStop    || b2
#                            ,isJust choice &&
↪   not choiceStop || b3]
#                 in if isNothing st || and bits
#                      then Just £ replicate 3
↪   False
#                      else Just bits

def caseCont(val, dataStop, ptrStop, choiceStop,
↪   st):
    def makeOut(index, v):
        return v if not (bits[index]) else None

    dOut = makeOut(0, caseData(val))
    spOut = makeOut(1, casePtr(val))
    choice = makeOut(2, val)
```