CoreLoom: A Semantic Model for Fine-Grained Flexible Datapaths

Michael Pellauer

Submitted Version

Chalmers University of Technology
International Master's Program
in
Dependable Computing Systems

Abstract

Recently much research has been directed into the area of flexible or reconfigurable datapaths. However, it can be difficult for systems architects to justify the hardware costs of these extra features because, unlike traditional concepts like performance and power consumption, flexibility is not a well-defined goal. CoreLoom is an embedded language which represents a semantic model of flexible datapaths. It can be used by architects to explore the impact of design decisions on flexibility. The model can form a basis for describing the datapath to compilers, or for automatically generating control units. The CoreLoom approach is applied to example datapaths to demonstrate how a designer may add flexible features, such as transforming a standard FIR filter into a more general vector unit. Finally, the report explores issues that the model raises such as developing a useful metric of flexibility.

Acknowledgments

I would like to gratefully acknowledge the support of the FlexSoC project at Chalmers, including Magnus Sjalander, Martin Thuresson, Peter Gammie, and Lars Svensson. Particular thanks go to John Hughes and Mary Sheeran for their help and advice throughout the whole process.

I would also like to acknowledge the the work of Börje Johansson. His capable administration makes the international Master's program in Dependable Computing Systems dependable.

Contents

List of Figures

List of Tables

1 Introduction

Recently much research has been directed at the concept of flexible or reconfigurable architecture. Projects have explored all aspects of flexibility, from fine grained control of pipeline registers [11], to processors with built-in FPGA components [7, 15]. Yet, interestingly enough, the concept of *flexibility* itself remains largely ill-defined. The common denominator across recent projects seems to be the desire to add increased functionality while minimizing associated hardware overhead.

Yet the addition of extra functionality is not enough to qualify an architecture as "flexible." Consider the example of a hardware core which is designed to encrypt data with a key using the DES standard. Suppose the system architect later receives a request to add support for the newer AES standard. The architect could take advantage of the parallelism of hardware to fulfill the request - by adding new logic to calculate the AES encryption at the same time as the DES. The end result would be a unit which delivers the results of both encryptions, thus fulfilling the requirements.

Yet common sense tells us that this new dual unit is not flexible. Each clock cycle it does exactly the same thing - now it just does more of it. On the other hand, if the architect had chosen to implement the unit as one which switches back and forth between AES and DES modes, then common sense would term the unit flexible. Hence, at one level flexibility is about *exclusion*. A unit is flexible only if there is a choice to be made about which mode, or modes, to operate in.

Furthermore, this choice should not be made by the unit itself, but rather by an external agent. A unit which always operates in DES mode for 25 clock cycles followed by 25 clock cycles of AES mode is still inflexible, because there is no choice to be made. Therefore, at another level flexibility is about *interaction* between a unit and an external environment.

Flexibility is also interesting in that, unlike concepts such as performance and power consumption, it is usually not a goal in and of itself. The general-purpose processor (GPP) represents the ultimate in flexibility, yet it is not the ideal hardware solution to every computational problem. Over the last ten years regular algorithms which operate on streams of data have played an increasingly important role in many application domains. Because these computations usually do not refer to input data more than once, traditional architecture guidelines which aid GPPs such as the Principles of Locality do not apply. Additionally, these calculations sometimes must meet real-time constraints that GPPs cannot guarantee.

For these reasons systems designers often turn to custom hardware datapaths as the best solution. An entire industry has developed around the efficient development of such datapaths, their marketing as Intellectual Property Cores, and their arrangement into Systems-On-Chip which augment GPPs - or even replace them altogether.

Yet these custom datapaths can bring many disadvantages. Being tailored to one task, these units go unused when this task is not being performed, representing wasted computational potential. Their customized nature also means that they are not adaptable to future changes in standards, even if these changes represent minor alterations to the datapath.

Flexible datapaths are designed to add some of the flexibility and programmability of general-purpose processors back into hardware datapaths. Yet how can systems architects gauge the success of their results? While it is easy to say System A has 10% better performance or power consumption than System B, what does it mean to say that System B is 20% more flexible?

At the heart of the problem is the lack of any agreed-upon metric of flexibility. Such a metric would give designers a way to measure the impact of specific design decisions on flexibility. If a designer adds a new connection from an adder to a logic unit we can easily measure the impact on power consumption, critical-path length and circuit area. With the existence of such a metric we could measure whether or not the benefits to flexibility outweighed these costs.

This report describes CoreLoom, a small domain-specific embedded language in Haskell [9]. The CoreLoom framework explores the subject of flexible datapaths from the ground up, and provides systems designers with tools and methodologies to use when reasoning about (and justifying the costs of) flexible features. Additionally, this framework should be complete enough to be usable by a compiler to configure the datapaths.

Overall, this project includes:

- Defining a conceptual framework for flexible datapaths and associated terms
- Developing the CoreLoom semantic model for representing and reasoning about these datapaths
- Describing CoreLoom's features and implementation
- Modeling example datapaths in CoreLoom and adding flexibility to them.
- Exploring the issues that this model raises, including working towards a metric of flexibility.

1.1 Fine-Grained Flexible Datapaths

The project does not deal with reconfigurable computing arrays such as FPGAs. This may be confusing as such units are often referred to as offering "fine-grained control" of reconfiguration. Such units incur a high overhead for reconfiguration, and then little or no overhead while being used thereafter.

Rather, this project is limited to flexible datapaths where the route through the datapath is determined on a per-clock-cycle basis, as demonstrated by [11]. These are referred to as fine-grained because they allow designers to control datapaths on the most detailed level.

Throughout the report units which incur an initial configuration overhead followed by no overhead will be referred to as *reconfigurable*, whereas units which require control bits to be delivered on every clock cycle will be called *flexible*.

1.2 The FlexSoC Project

As demand for flexibility in hardware units increases, it stands to reason that system architects can provide efficient environments for flexible units to reside in. The FlexSoC project at Chalmers [10] aims to do just this, among other goals.

FlexSoC aims to provide a framework for System-On-Chip development including:

- Encoding and compression of configuration information on a per-application basis.
- Efficient caching and switching of configurations.
- Automatic disabling or voltage scaling of units on a per-configuration basis.
- A uniform programming interface to the system.

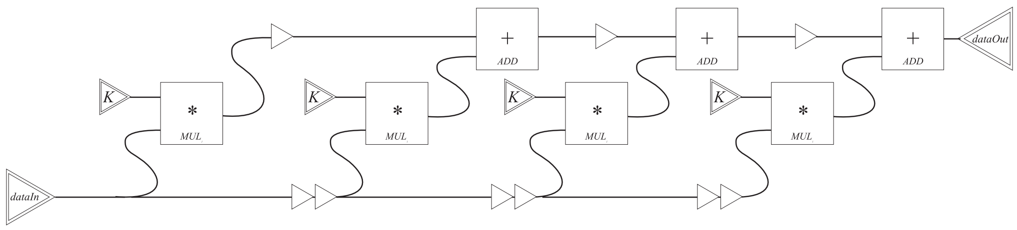
The FlexSoC approach is to expose all capabilities of all member units as a Native Instruction Set (N-ISA). This set will be quite large, yet most applications should only use a small subset for their specific task. Therefore FlexSoC allows each application to define an Application-Specific Instruction Set (AS-ISA). Special translation units handle the instruction decoding, and caching techniques are used to increase efficiency. Special control units distribute configuration information and handle communication between the units, potentially including altering clock cycle domains when appropriate.

The overall impact of projects like FlexSoC is that systems-on-a-chip designers will have access to a larger degree of flexibility. The presence of a uniform programming interface should mean that a unit's features are accessible to future applications that the designer does not necessarily anticipate, such as an application using the multipliers and adders of a FIR-Filter unit to compute vector dot products (Section 5.2). Additionally, because datapath control units like FlexSoC's should eventually be automatically inferred from datapath definitions, architects are freed from the burden of writing such units themselves.

However, the ease of implementation of flexible features is not a justification for their existence. Research shows that even small amounts of flexibility can incur large overheads [3]. Until researchers can show specific benefits from flexibility and reconfigurability they will not become widely-adapted design goals in and of themselves.

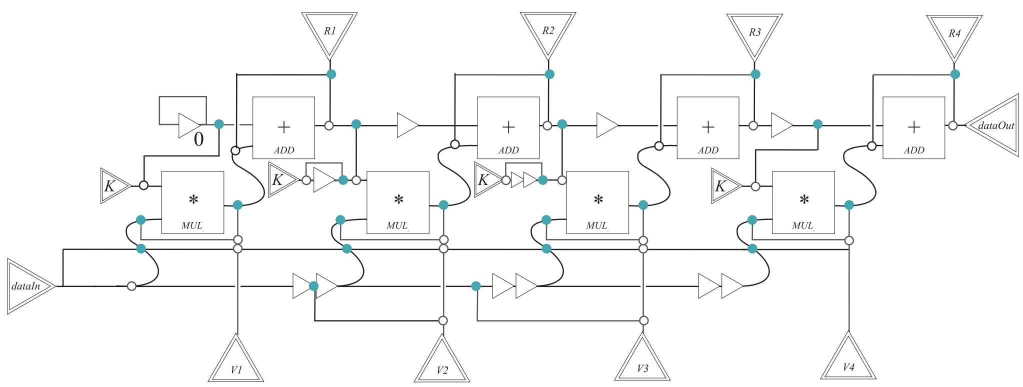
1.3 Initial Example: Flexible FIR Filter

As an initial example, let consider the FIR filter implementation shown here:



Suppose that this unit is being used as part of a system-on-chip pipeline, and that another application performs vector addition and multiplication. The system architect has determined that the FIR operations and the vector operations will never happen concurrently. In this case the architect may wish to explore whether it is practical to use the adders and multipliers of the FIR filter to perform these operations rather than adding a new core exclusively to support them.

In Section 5.2 we show how the architect could use CoreLoom to develop the following flexible FIR filter:



This unit is capable of performing FIR filtering with the same throughput and latency as the original unit, as well as vector-scalar operations, vector addition, multiplication, and dot product. This extra flexibility will certainly impose an increased cost on characteristics such as circuit area, power consumption, and critical path length. Yet perhaps the impact on the overall system of implementing the combined core will be lower than that of implementing two separate cores. Research has shown that interconnection networks play a large role in overall system-on-chip power consumption [17], and it will be up to the architect to determine whether the benefits of additional flexibility outweigh the drawbacks.

2 Conceptual Framework

This section provides a basis for the remainder of the report through the exploration of flexible datapaths at the conceptual level. It begins by defining the terms associated with flexible hardware datapaths, then uses these terms to define a problem statement that motivates the implementation of CoreLoom.

2.1 Definition of Terms

Datapath

A datapath is a finite collection of hardware functional units, latches and directed connections between them. Conceptually, data enters the datapath at inputs, walks a path through the functional units and latches, and exits from the outputs.

Flexible Datapath

A datapath which has the ability to alter the path data flows, and where this path is specified by an external environment. Conceptually this means that the datapath must have the ability to do exclusion, and that the choice of this exclusion is somehow specified through the flexible datapath's interface. This indicates that the datapath includes join points that are controlled by a control unit. Choices made by those join points are determined by control data which the control unit delivers.

An alternative approach to the concept is the idea that a flexible datapath, when combined with control data, is equivalent to a datapath. Or, in Haskell notation:

```
combine :: FlexDatapath -> Configuration -> Datapath
```

Control Unit

A unit which controls a flexible datapath by specifying the exclusion decisions, and acting as the interface for configuration changes to the external environment in which the datapath resides. Control units may handle other issues, such as stalling the pipeline or handling exceptions.

Configuration

A finite sequence of control words which the external environment provides to a flexible datapath. These will typically be expressed as tables. For example, here is an example configuration for a datapath with join points jp_0, jp_1, jp_2 :

jp_0	jp_1	jp_2
TRUE	**FALSE**	**TRUE**
FALSE	**TRUE**	**FALSE**
TRUE	**TRUE**	**TRUE**

The control words are delivered in order on each clock cycle. When the end of a sequence is reached it should repeat from the beginning.

Usage Pattern

Sequences of data corresponding to the inputs and outputs of a datapath which represents the use of that particular datapath. For example consider a datapath F with inputs P and Q, and output R. When given sequences of values for P and Q the unit produces output on R in the following manner:

CC	P	Q	R
0	x_0	y_0	
1	z_0		$x_0 + y_0 + z_0$
2	x_1	y_1	$x_1 + y_1 + z_0$
3	z_1		$x_1 + y_1 + z_1$
4	x_2	y_2	$x_2 + y_2 + z_1$
5	z_2		$x_2 + y_2 + z_2$
6	x_3	y_3	$x_3 + y_3 + z_2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

This table represents a usage pattern of unit F. Empty boxes represent unknown or irrelevant values. Encoded in a usage pattern are latency and throughput.

Latency

The number of clock cycles it takes a datapath to produce output data after the input data has been provided.

As an example consider a datapath G with inputs P and Q, and output R, which is $P \wedge Q$. If G has the following usage pattern:

CC	P	Q	R
0	x_0	y_0	
1	x_1	y_1	
2			$x_0 \wedge y_0$
3			$x_1 \wedge y_1$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Output R is calculated with a latency of 2 clock cycles. Latency is calculated with respect to each individual output of a datapath.

Throughput

A ratio of the number of clock cycles where useful output data is produced to total clock cycles, discounting any start-up latency. For example, consider if the unit G from above had the following usage pattern:

CC	P	Q	R
0	x_0	y_0	
1			
2	x_1	y_1	$x_0 \wedge y_0$
3			
4	x_2	y_2	$x_1 \wedge y_1$
5			
6	x_3	y_3	$x_2 \wedge y_2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

While G still has a latency of 2 clock cycles, it now has a throughput of 1 result every 2 clock cycles, or $\frac{1}{2}$.

Algorithm

Each configuration on a given datapath implements some algorithm - a transformation of given input data. Algorithms are independent of considerations of latency and throughput, though compilers generally try to optimize these metrics in the resulting configuration.

2.2 Note on Definitions

It should be noted that the definition of flexible datapaths is sufficiently broad that it covers many hardware units that are not generally considered flexible. For example, most ALU units could be considered to take configuration information and data - the operation and operands.

In general, this broad definition seems to be an advantage. One consequence is that the model can represent low-level functional units such as ALUs directly rather than treating them as special cases (Section 4.3). However, the model is predicated on a strict semantic separation of control from data, and thus cannot deal with datapaths where the meaning of signals shift over time.

2.3 Problem Statement

Construct and implement a semantic model to represent flexible datapaths. Given a datapath definition this model must be able to express:

- all configurations that the datapath supports

- the dataflow for specific configurations even if data values are unknown
- an indication of how flexible the datapath is
- accurate dataflows even if the exact content of functional units is unspecified.

The final requirement reflects the fact that functional units on modern datapaths can be incredibly complex, and yet the compiler often does not need to know about this complexity. Consider the example of the **S-box** tables used in AES encryption [13]. These represent nothing more than substitution tables and can be tedious to describe in a high-level language, yet they are a necessary part of the AES transformation process.

The model should still be able to reason about the unit by treating it as a "black box" functional unit (Section 4.3.2). This is acceptable because the model is required to represent the *dataflow* of the datapath, not the actual evaluation of the calculation. For example the model could show that if data values X and Y flow through the **S-Box** unit then the result flowing out will be **S-Box**(X, Y), which may be sufficient information for the architect or compiler.

3 CoreLoom Data Structures

loom
noun

> A device for making thread into cloth by weaving vertical strands, called the *warp*, together with horizontal strands, called the *weft*.

With the conceptual framework and problem statement defined we are now able to construct a model as an embedded language in Haskell. We have named this language CoreLoom, as it weaves the warp of data and the weft of control together into a flexible datapath.

3.1 Symbolic Values

Our model must represent data in the datapath, however we are not always aware what this data will be. One approach that has been used successfully in such situations is the use of symbolic values [12, 2]. Symbolic values are given a name so they may be tracked through the datapath, but their actual values need not be specified. This approach is particularly suited to compilers, which often need to apply transformations to variables without knowing the content of those variables.

CoreLoom symbolic values are represented by Haskell data type `SymbVal` (`SymbVal.hs`, Appendix):

```
data SymbVal = Var String
             | VarIx String Int
             | Unknown
             | VBool Bool
             | VInt Integer
             | FuncApp SymbFunc [SymbVal]
```

Variables

Variables (`Var`) represent unknown data flowing through the datapath. Although we do not know the value of this data, we can use the variable name to track it through the datapath. We can also distinguish equivalent values, allowing us to implement algebraic properties such as $x + y - x = y$

Additionally we will allow variables to be indexed (`VarIx`) which will help to keep track of clock cycles later.

Unknown Values

The undefined value (`Unknown`) corresponds to values which we know nothing about. This may be the uninitialized value of a unit or the a result of a circuit error, such as joined wires or combinational loops.

Additionally, users may wish to define configurations which do not make use of the entire datapath. For the unused portions rather than specifying an irrelevant choice we will use `Unknown`, referred to as `open`.

Known Values

Occasionally certain values of data may be known. Currently the model allows Boolean (`VBool`) and Integer (`VInt`) values. It would be straightforward to expand it to other data types, such as floating point values.

Symbolic Function Application

Symbolic values also include the ability to record that function application has occurred. We refer to mappings of symbolic values as symbolic functions. The Function Application (`FuncApp`) constructor can be used to record that function application has occurred to a value. Further results depend on the definition of the symbolic function itself.

For example, CoreLoom defines a symbolic version of logical **Or** called `sOr`. The definition specifies:

```
...
sOR (VBool p) (VBool q) = VBool (p || q)
sOR (VBool True) y = VBool True
...
sOR x y = FuncApp OR [x,y]
...
```

This is because **Or**(x, **True**) evaluates to **True** by Domination, but **Or**(x, y) cannot be evaluated further, and thus results in the application **Or**(x, y). Such results often turn out to be sufficient information for the compiler, which is not attempting to evaluate expressions per se, but rather to correctly map them onto the available hardware datapath.

The file `SymbVal.hs` (Appendix) contains more definitions of common symbolic functions and their simple algebraic properties.

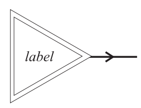
3.2 Datapath Model

CoreLoom models datapaths as directed graphs, where the nodes represent the functional units, and the edges connections between their input ports and output ports. The CoreLoom model defines five distinct types of nodes:

```
data Node = Input NodeName
          | Output NodeName
          | Delay NodeName SymbVal
          | Function NodeName SymbFunc
          | JoinPoint NodeName
```

3.2.1 Primitive Node Types

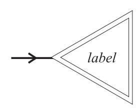
Input Nodes



Input nodes represent the entry points of data into the datapath. They are labeled with a variable marking the identity of the input.

Input nodes have no input ports, and one output port.

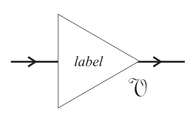
Output Nodes



Output nodes represent the exit points of data from the datapath. They are also labeled with a variable marking the data which reaches it every clock cycles.

Output nodes have one input port, and no output ports.

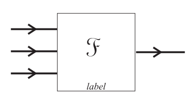
Delay Nodes



Delays represent latches which delay data one clock cycle. They are labeled, and associated with a symbolic value which is the initial value of the delay, and is output on clock cycle 0. They obtain new a value at the end of every clock cycle, and output it on the following clock cycle.

Delays have one input port and one output port.

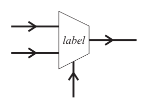
Function Nodes



Functions represent the application of symbolic functions, the parts of the datapath which transform or act upon data. Each function node has a unique label and is associated with a symbolic function f.

A function node has n input ports, where n is the arity of f, and one output port, since the result of a symbolic function application is always a single symbolic value (Section 3.1).

Join Point Nodes



Join Point nodes represent the choice between multiple data inputs. There are many ways that join points could be modeled. We could, for example, choose not to model them as a primitive node but as a Function node. This node would represent choice through a symbolic function `mux` with where:

```
mux :: SymbVal -> SymbVal -> SymbVal -> SymbVal
mux x y (VBool False) = x
mux x y (VBool True) = y
mux x y z = Unknown
```

After much debate we have decided instead to represent join points as primitive nodes although their semantics (Section 4.1) are equivalent to the above `mux` function. This reflects the special position of exclusion in our model. The benefit is that we can introduce the following convention:

> Join Point nodes with unconnected control ports are assumed to be connected to the control unit of a datapath, and the choice exposed to the Configuration.

Join points with unconnected control represent exclusion choices exposed to the external environment, thus datapaths which include such nodes represent flexible datapaths. They require decisions to be provided at every clock cycle by the external environment.

Choices between more than two elements can simply be implemented as trees of join points.

Join Point nodes have three input ports: ports 0 and 1 are the incoming data, and port 2 is the control port. By convention the control port is drawn on the top or bottom of the node. It has one output port - the result of the decision.

3.2.2 Edges

Edges in the graph represent connections from the output node of one node to the input port of another. Join points have three input ports and functions have as many as their arity. Therefore we will implement input ports as follows:

```
addNode :: Node -> Result ()
addNodes :: [Node] -> Result ()
--Add a node or nodes to the datapath
input :: NodeName -> Result ()
inputs :: [NodeName] -> Result ()
--Create input node(s) with the given label(s)
output :: NodeName -> Result ()
outputs :: [NodeName] -> Result ()
--Create output node(s) with the given label(s)
function :: SymbFunc -> NodeName -> Result ()
functions :: SymbFunc -> [NodeName] -> Result ()
--Create function node(s) with the given label(s), and the given function.
delay :: SymbVal -> NodeName -> Result ()
delays :: SymbVal -> [NodeName] -> Result ()
--Create delay(s) with the given initial value
delayN :: Int -> SymbVal -> NodeName -> Result ()
--Create N delays linked sequentially with the given initial value.
remNode :: NodeName -> Result ()
remNodes :: [NodeName] -> Result ()
--Remove node(s) from the datapath
```

Figure 3.1: CoreLoom functions for creating primitive nodes (`Datapath.hs`, Appendix)

```
type InputPort = (NodeName, Int)
```

a tuple of the node name, and the number of the port (starting from zero). Notice that no nodes have more than one output port, so they do not to be explicitly represented in the implementation. Edges in the graph then are tuples from the previous node, to the next node's input port:

```
type Connection = (NodeName, InputPort)
```

3.2.3 CoreLoom Datapath Creation Functions

Now that we have decided how to model connections we are able to implement the basic process of creating datapaths. In general a CoreLoom datapath definition first defines all of its nodes, then defines the edges between them. Figure 3.1 documents the main functions for creating primitive nodes.

Connections between nodes must be provided explicitly in the description. This increases the verbosity of description, but eases the addition or change of connections in an established datapath. Because many node types do not have more than one input port CoreLoom provides implicit versions of connection functions. By convention these connect to input port 0. CoreLoom also provides other convenience functions which cut down on the verbosity of datapath descriptions, as shown in Figure 3.2.

```
addEdge :: Connection -> Result ()
addEdges :: [Connection] -> Result ()
--Add edge(s) to the datapath
connect :: (NodeName, NodeName) -> Result ()
connections :: [(NodeName, NodeName)] -> Result ()
--connect the first node(s) to the second
--assumes input port zero
connect_n :: (NodeName, InputPort) -> Result ()
connections_n :: [(NodeName,InputPort)] -> Result ()
--Connect the first node(s) to the second
--at input port N.
connectTo :: NodeName -> [NodeName] -> Result ()
connectTo_n :: NodeName -> [InputPort] -> Result ()
--Connect the first node to all of the others.
repoint :: InputPort -> Connection -> Result ()
--Alters a connection's destination
reanchor :: NodeName-> Connection -> Result ()
--Alters a connection's origin
linkUp :: [NodeName] -> Result ()
linkUp_n :: [InputPort] -> Result()
--connects the given nodes in a sequence
```

Figure 3.2: CoreLoom functions for connecting nodes. (`Datapath.hs`, Appendix)

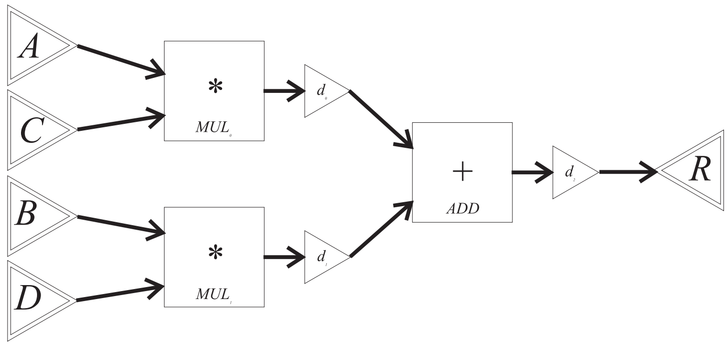
3.2.4 Graph Restrictions

In order to accurately represent the properties of hardware the following restrictions are placed on graph construction:

- All loop paths must include at least one node of Delay type - otherwise the result would be a combinational loop.
- More than one edge may originate from a output port, but multiple edges may not be connected to one input port. This represents the fact that we can split wires but not join them.
- All input and output ports of a graph must be connected, except for Join Point nodes, input port 2.
- Graphs must include at least one Output node.
- Graphs must be weakly connected. (Disjoint components would represent multiple elemental datapaths.)

CoreLoom uses a state monad (`STATE.hs`, Appendix) to order graph construction functions. However, this allows the user to construct graphs which do not obey these restrictions. Graph validity is checked through the `makeDatapath` function (`Datapath.hs`, Appendix).

Additionally, the `makeDatapath` function handles the application of the state monad, meaning that the data structure is available as a non-monadic type.




```
dotP :: Datapath
dotP = makeDatapath dp
 where
  dp = do
    inputs ["A","B","C","D"]
    delays open ["d_0", "d_1", "d_2"]
    functions MUL ["MUL_0", "MUL_1"]
    function ADD "ADD_0"
    output "R"
    connections_n [("A", ("MUL_0",0)), ("C", ("MUL_0",1))]
    connections_n [("B" ,("MUL_1",0)), ("D", ("MUL_1",1))]
    connections [("MUL_0", "d_0"), ("MUL_1", "d_1")]
    connections_n [("d_0", ("ADD_0", 0)), ("d_1", ("ADD_0", 1))]
    connections [("ADD_0", "d_2"), ("d_2", "R")]
```

Figure 3.3: Vector dot-product datapath in CoreLoom.

3.2.5 Example: Dot-Product Unit

Figure 3.3 shows a small example of a CoreLoom datapath. It has four inputs A, B, C, D, and one output R. Every clock cycle it calculates $R = AC + BD$, the dot product of vectors $[A\ \ B]$ and $[C\ \ D]$ - a common calculation in the domain of computer graphics. As shown by its usage pattern in Figure 3.1it performs this calculation with a latency of 2 and a throughput of 1.

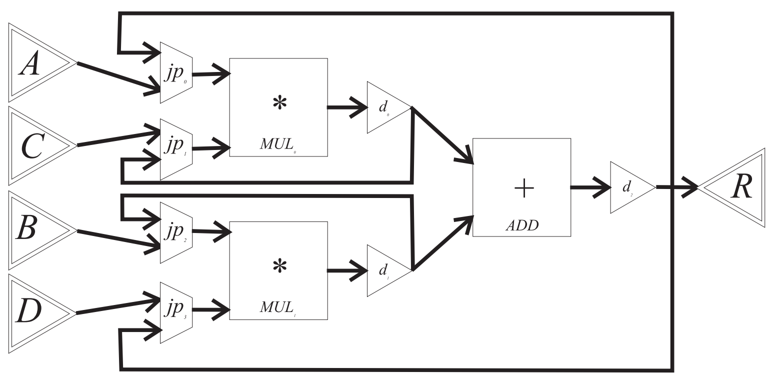
Note that it is an entirely *inflexible* datapath. However consider the altered version of the datapath shown in Figure 3.4. This datapath meets our definition of flexibility - the join points have no control, indicating that there is now an exclusion decision which the external environment must make on every clock cycle.

3.2.6 Implicit Join Points

As a convenience in CoreLoom the user does not need to explicitly add join point nodes. Rather, if the user connects more than one connection to a input node, then a join point is implicitly added before that node, and its exclusion decision exposed to the configuration. Thus the code in Figure 3.4 could be rewritten as follows:

CC	A	B	C	D	R
0	x_0	y_0	z_0	w_0	
1	x_1	y_1	z_1	w_1	
2	x_2	y_2	z_2	w_2	$x_0z_0 + y_0w_0$
3	x_3	y_3	z_3	w_3	$x_1z_1 + y_1w_1$
4	x_4	y_4	z_4	w_4	$x_2z_2 + y_2w_2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 3.1: Usage pattern for the datapath in Figure 3.3




```
flex_dotP :: Datapath
flex_dotP = makeDatapath dp
 where
  dp = do
    inputs ["A","B","C","D"]
    delays open ["d_0", "d_1", "d_2"]
    functions MUL ["MUL_0", "MUL_1"]
    function ADD "ADD_0"
    joinPoints ["jp_0","jp_1","jp_2","jp_3"]
    output "R"
    connections_n [("A", ("jp_0",1)), ("C", ("jp_1",0))]
    connections_n [("B" ,("jp_2",1)), ("D", ("jp_3",0))]
    connections_n [("jp_0", ("MUL_0",0)), ("jp_1", ("MUL_0",1))]
    connections_n [("jp_2" ,("MUL_1",0)), ("jp_3", ("MUL_1",1))]
    connections [("MUL_0", "jp_1"), ("MUL_1", "jp_2")]
    connections_n [("d_0", ("MUL_0",1)), ("d_1", ("MUL_1", 0))]
    connections_n [("d_0", ("ADD_0", 0)), ("d_1", ("ADD_0", 1))]
    connections_n [("d_0", ("ADD_0", 0)), ("d_1", ("ADD_0", 1))]
    connections [("ADD_0", "d_2"), ("d_2", "R")]
    connectTo_n "d_2" [("jp_0",0), ("jp_3",1)]
```

Figure 3.4: A flexible version of the dot-product datapath.

```
flex_dotP :: Datapath
flex_dotP = makeDatapath dp
 where
  dp = do
    inputs ["A","B","C","D"]
    delays open ["d_0", "d_1", "d_2"]
    functions MUL ["MUL_0", "MUL_1"]
    function ADD "ADD_0"
    output "R"
    connections_n [("A", ("MUL_0",0)), ("C", ("MUL_0",1))]
    connections_n [("B" ,("MUL_1",0)), ("D", ("MUL_1",1))]
    connections [("MUL_0", "d_0"), ("MUL_1", "d_1")]
    connections_n [("d_0", ("MUL_0",1)), ("d_1", ("MUL_1", 0))]
    connections_n [("d_0", ("ADD_0", 0)), ("d_1", ("ADD_0", 1))]
    connections [("ADD_0", "d_2"), ("d_2", "R")]
    connectTo_n "d_2" [("MUL_0", 0), ("MUL_1", 1)]
```

This addition is done as part of the function `makeDatapath` (`Datapath.hs`, Appendix). All existing connections are re-pointed to the join point. A new join point added before node `"X"`, input port 1 is given the name `"into_(X, 1)"`.

3.2.7 Model Assumptions

In order to accurately reflect hardware datapaths our model would have to take issues of data format into account. For example, to accurately model the dot product unit above we would need to know:

- The bit lengths of all the arguments and results
- Whether the left-most bit is most significant or least significant
- Whether the data should be in One's Complement or Two's Complement notation.

While such issues are important for completeness, they are straightforward to add and do not represent the research focus of this project. Therefore this model assumes that all such issues are resolved as a precondition. Extra hardware required to resolve these issues is not explicitly represented by the model.

Specifically, this model assumes that:

- The bit lengths of all data is appropriate.
- All data formats match in terms of signal representation.
- Input values are always available. Alternatively, if input values are not yet available then the control unit takes action such as stalling the pipeline or altering the configuration.
- Data may always be written to outputs. Thus in cases such as attempting to write into a full FIFO buffer the control unit takes appropriate action.

```
conf :: Config
conf = (["jp_0",      "jp_1",      "jp_2",      "jp_3"],
       [[VBool False, VBool False, VBool False, VBool False],
        [open,        open,        open,        open],
        [VBool True,  VBool False, VBool True,  VBool False],
        [VBool False, open,        VBool False, open],
        [open,        VBool True,  open,        VBool False]])
```

jp_0	jp_1	jp_2	jp_3
FALSE	**FALSE**	**FALSE**	**FALSE**
TRUE	**FALSE**	**TRUE**	**FALSE**
FALSE		**FALSE**	
	TRUE		**TRUE**

Table 3.2: Haskell and graphical representation of example configuration for flexible datapath from Figure 3.4.

- Hardware units that generate exceptions or interrupts are correctly handled by the control unit - these signals are not represented by the model.
- The entire datapath is within the same clock cycle domain. The model could be extended to show relationships between sections with related clock cycles as future work.

3.3 Configurations

Configurations in our model are the sequence of control words that are used by the exposed Join Point nodes each clock cycle. The control words are simply a collection of boolean values corresponding to the choice that each join point is supposed to make.

Configurations in CoreLoom are represented by the following datatype:

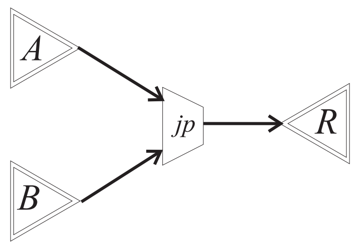
```
type Row = [SymbVal]
type Table = [Row]
type Config = ([NodeName], Table)
```

where `[NodeName]` is the list of exposed join points in the datapath, and `Table` is the sequence of exclusion decisions to make for those join points on every clock cycle. When the end of the sequence is reached it repeats from the beginning. Table 3.2 is an example of a configuration for the datapath from Figure 3.4. The **OPEN** value is represented as blank cells within the table.

Functions for working with CoreLoom configurations can be found in the file `Configuration.hs` (Appendix).

3.3.1 Representation Conventions

Lists of boolean values can be difficult for humans construct or interpret. Therefore, rather than use **True** or **False** it is easier to represent our decisions by the label of the node name whose data we wish to select. For example, with the following datapath:



the configuration:

jp
A
B
B
A

corresponds to the boolean choices:

jp
False
True
True
False

It is straightforward to translate between the two formats, therefore the rest of the report will use the node names format for readability purposes. Additionally, for convenience trees of Join Point nodes will be represented as choices between more than two nodes, as shown in Figure 3.5. These choices will be condensed in configurations as well. Table 3.3 shows a valid configuration for this datapath with and without all of the conventions.

Rather than attempt to translate between the two formats, CoreLoom is implemented with the following modifications:

- Join Points may have any number of edges connected to input port 0
- The control input of a join point is input port 1 (rather than 2).
- The control input selects from one of the nodes connected to input port 0.

This modified semantics is easier for human users to read and work with.

CoreLoom provides the following functions for extracting configuration information from a datapath definition:

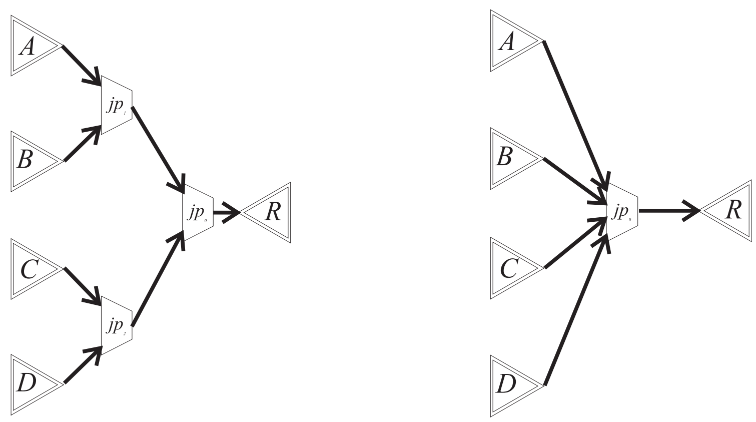


Figure 3.5: Convention for trees of join points.

jp_0
A
B
C
D

jp_0	jp_1	jp_2
False	**Open**	**False**
True	**Open**	**False**
Open	**Open**	**Open**
Open	**False**	**True**
Open	**True**	**True**

Table 3.3: Example configuration for datapath in Figure 3.5 with and without syntactic sugar conventions.

```
configChoices :: Datapath -> [NodeName]
--Returns all exposed join points of a datapath
configInfo :: Datapath -> [(NodeName,[NodeName])]
--Returns all exposed join points and their valid choices
blankConfig :: Datapath -> Int -> Config
--Creates a blank config of length n for a given dp
```

Functions for working with configurations can be found in the file `Configuration.hs` (Appendix).

3.4 Usage Patterns

As defined in Section 2.1 usage patterns are sequences which represent all information necessary for using a datapath in a given configuration, including the latency, throughput, and dataflow which reach the datapath's output nodes.

In CoreLoom, usage patterns are represented with the following data structures:

```
type Pair = (NodeName, SymbVal)
type InputPattern = [[Pair]]
type OutputPattern = [[Pair]]
```

```
type UsagePattern = (InputPattern, OutputPattern)
type Latency = Int
type Throughput = (Int, Int)
```

CoreLoom Usage Patterns are sequences of each input and how it has been used, and each output and how it has been used. `Latency` is a simple integer, but `Throughput` is a fraction `(Numerator, Denominator)` so that $\frac{1}{2}$ would be represented as `(1, 2)`.

Functions for working with usage patterns can be found in the file `UsagePattern.hs` (Appendix).

4 CoreLoom Semantic Model

The previous chapter defined representations of flexible datapaths, configurations, and usage patterns. This chapter presents the CoreLoom semantic model, which expresses the relationship between the three.

Using this model we can then discuss the more advanced features of CoreLoom, including standard graph elements, the representation of more complex functional units, properties of configurations, and modular datapaths.

4.1 Datapath Semantics

CoreLoom datapath semantics represents the following relationship:

```
run :: Datapath -> Config -> UsagePattern
```

However, this process would never terminate, as hardware datapaths never stop running as long as they have power. Instead we will limit our model to simulating a given number of clock cycles:

```
run :: Datapath -> Config -> Int -> UsagePattern
```

Because configurations are defined to be finite sequences, simulating a limited number of clock cycles is sufficient to obtain the overall usage pattern of the datapath in a configuration.

Figure 4.1 shows the overall algorithm. It takes:

- A flexible datapath model as defined in Section 3.2.4.
- A configuration `Config` which provides exclusion decisions for each unattached join point in the datapath for every clock cycle.
- The number of clock cycles to simulate.

The algorithm returns a usage pattern for input and output nodes on each clock cycle. In general, the usage pattern contains all of the information required by a compiler or scheduler. In the future, a second version will be provided which returns the overall state of every delay element on every clock cycle, which can be useful for debugging.

The algorithm constructs the usage pattern and overall state one clock cycle at a time. It uses the function `drive` (`Run.hs`, Appendix) to push data through the datapath to the delay and outputs nodes.

Figure 4.2 shows the algorithm for `drive`. On every clock cycle:

```
graphsem =
 do
  if cur_cc  <= final_cc
     then do
        semCC
        graphsem
     else do
       return usagePattern
semCC =
 do
   let dp = curDatapath
   let delays = getDelays dp
   let outputs = getOutputs dp
   mapM drive (outputs ++ delays)
   updateUsagePattern
   writeNewState
```

Figure 4.1: Overall semantics for a flexible datapath graph and configuration. For exact code see `Run.hs`, Appendix.

- Input nodes which are "driven" update the usage pattern and produce symbolic variables indexed with the current clock cycle number, which simplifies tracking latency.
- Output nodes record in the usage pattern which values reached them.
- Delay nodes calculate their new value (if it has not already been calculated) and record it in the new state. The value they produce is the value found in the current state.
- When data passes through function nodes, the associated symbolic function is applied to all inputs.
- At join point nodes, a decision is made about which value to pass through. If there is no previous vertex to a join point node's input port 1, then the decision comes from the external control, as required by convention.

For convenience, the following versions of run are also available:

```
runIO :: Datapath -> Config -> Int -> IO ()
--Displays the resulting usage pattern in a table
runHTML :: FileName -> Datapath -> Config -> Int -> IO ()
--Outputs the usage pattern as an HTML table
```

```
drive :: Node -> RunState SymbVal
drive (Input i) =
 do
   let v = VarIn i currentCC
   addUseIn (i, v)
   return v
drive (Output o) =
 do
   v <- drive (prevNode (o,0))
   addUseOut (o, v)
   return v
drive (Delay d i) =
 do
   next <- nextVal d
   case next of
     Just x -> return (curVal d)
     Nothing -> do --new value for next CC
       v <- drive (prevNode (d, 0))
       updateNextVal (d, v)
       return (curVal d)
drive (Function f fun) =
 do
   let k = nParams fun
   let pns = allPrevNodes f
   vals <- mapM drive pns
   return (symbApply fun vals)
drive (JoinPoint j) =
 do
   c <- if isConnected (j, 2)
        then drive(prevNode (j, 2))
        else getControl j
   case c of
     (VBool False) -> return drive(prevNode (j,0))
     (VBool True) -> return (drive prevNode (j,1))
     z -> return Unknown
```

Figure 4.2: Algorithm for driving nodes of a flexible datapath.

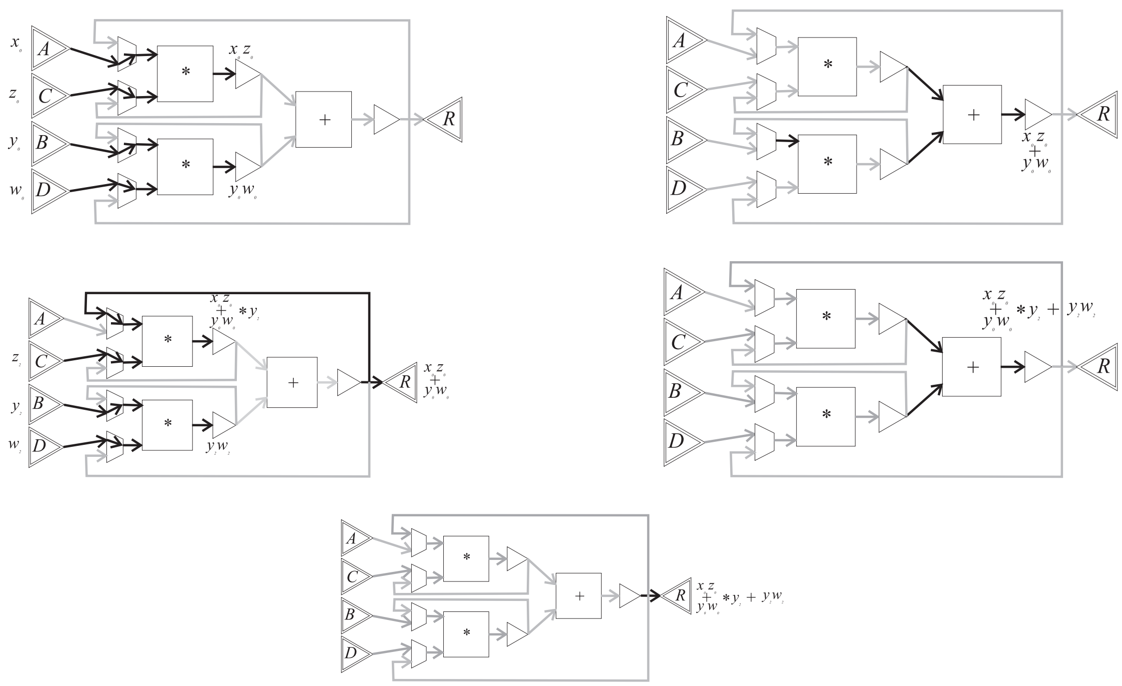


Figure 4.3: A series of dataflow decisions for the datapath from Figure 3.4

4.1.1 Semantics Example

Let us revisit the flexible dot product example from Section 3.2.5. Recall that the exclusion decisions of the join points are exposed to the external environment. We could, for example, chose to multiply the inputs, send them to the adder, send them back to the top multiplier while taking new inputs for the bottom multiplier, add these new results together, and finally output them.

This series of decisions is represented graphically by the dataflows in Figure 4.3. It corresponds to the configuration shown in Table 4.1.The problem of representing configurations and scheduling their repetition is discussed in Section 3.3.

Simulating five clock cycles in this configuration with our semantics results in the usage pattern shown in Table 4.2. Note that this unit can still calculate the original $R = AC + BD$ dot product algorithm with the same latency and throughput as the original, inflexible unit from Figure 3.3.

However, the new unit is still somewhat limited in its flexibility. Even though we have adders and multipliers, no matter what exclusion decisions we make we cannot use this datapath to calculate $R = (A+C)(B+D)$. This begins to suggest a metric of flexibility, as discussed in Section 6.2.

CC	jp_0	jp_1	jp_2	jp_3
0	TRUE	FALSE	TRUE	FALSE
1				
2	FALSE	FALSE	TRUE	FALSE
3				
4				

Table 4.1: Configuration for exclusion decisions in Figure 4.3.

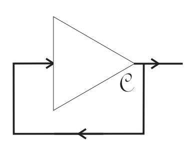
CC	A	B	C	D	R
0	x_0	y_0	z_0	w_0	
1					
2		y_2	z_2	w_2	$x_0z_0 + y_0w_0$
3					
4					$(x_0z_0 + y_0w_0)y_2 + z_2w_2$

Table 4.2: Usage pattern for Figure 4.3

4.2 Standard Graph Elements

With an understanding of how the primitive elements of the model relate to each other, we can use them to construct more complex standard elements.

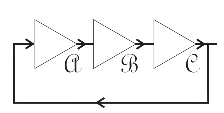
4.2.1 Constants



In a datapath a constant is a unit that emits the same value every clock cycle. A constant c can be represented as a delay unit with initial value c, connected to itself. Constants in CoreLoom are created with the following function:

```
constant :: SymbVal -> NodeName -> Result ()
--creates a constant of the given value and name
```

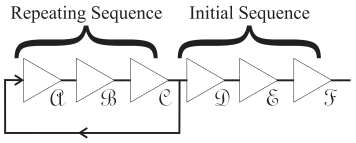
4.2.2 Constant Sequences



The concept of constants can be extended to sequences of values, which simply loop when the end of the sequence is reached. These are useful for simulation of configurations (Section 4.4).

```
constantSeq :: [SymbVal] -> NodeName -> Result ()
--creates a constant of the given sequence of values
```

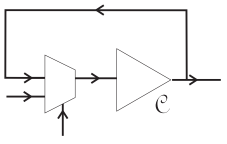
4.2.3 One-Time Constant Sequences



By combining the above constant sequence with chains of delays we can create sequences of values which are emitted once, then replaced by a new sequence which loops forever.

```
oneTimeSeq :: ([SymbVal], [SymbVal]) -> NodeName -> Result ()
--      (InitialSequence, FinalSequence)
--creates a sequence which emits InitialSequence once, then FinalSequence forever
```

4.2.4 Registers



The meaning of the term "register" is not always clear, but here we mean a latch where we can choose whether or not to replace the old value. This can be represented in our model by using a join point to choose between the old value of a register or a new incoming value.

CoreLoom provides three functions for creating and working with registers. By convention the join point for a register with name `Reg` is `Reg_load`.

```
register :: SymbVal -> NodeName -> Result ()
--creates a register with the given initial value and name
registerBefore :: NodeName -> SymbVal -> InputPort -> Result ()
registerAfter :: NodeName -> SymbVal -> NodeName -> Result ()
--create a register before or after the specified node,
--redirecting all existing connections to the new join point.
```

4.3 Functional Units in CoreLoom

Function nodes may only produce one output value. Genuine hardware functional units, on the other hand, often produce more than one value, or have more than one mode

```
half_adder :: Datapath
half_adder = makeDatapath dp
 where
  dp = do
   inputs ["P", "Q"]
   outputs ["R", "C"]
   addNode (Function "ADD" XOR)
   addNode (Function "CRY" AND)
   connections_n [("X",("ADD", 0)),("Y", ("ADD", 1)),("X", ("CRY", 0)), ("Y", ("CRY", 1))]
   connections [("ADD","R"), ("CRY", "C")]
```

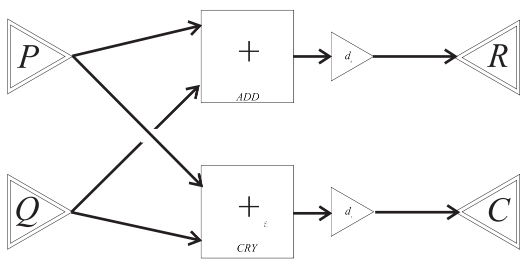


Figure 4.4: Datapath representation of a half-adder functional unit

that they operate in. We can represent such functional units simply as flexible datapaths themselves.

For example, suppose a datapath included a half-adder functional unit which took inputs P and Q and produced a result R and a carry-out bit C after 1 clock cycle. We can model this unit through the use of two functions: a symbolic implementation of addition $+$ to produce R and a new symbolic function $+_c$: the addition-carry function, which produces the carry result C. The result is the datapath graph given in Figure 4.4.

Functional units which involve selecting modes can be modeled using join point nodes. For example, if we wished to extend our half adder to be a small Arithmetic Unit capable of addition and subtraction, we could model this by adding join points controlled by new input Op, as shown in Figure 4.5. The datapath's configuration now specifies whether the unit performing addition or subtraction on any given clock cycle.

4.3.1 Modular Datapaths

CoreLoom allows users to instantiate previously defined datapaths in the definitions of new ones. This feature can be employed to reuse the definitions of functional units and build upon them modularly.

```
addDP :: Datapath -> String -> Result ()
--embeds the given datapath with the given name.
```

When a datapath is embedded it must be given a unique name. All of its nodes are then prefixed with `NAME:`. When connections are made which originate at the embedded datapath's outputs (or connect to the inputs) these outputs (respectively inputs) are

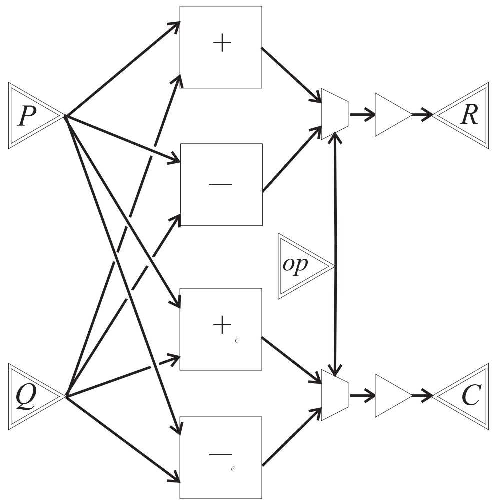


Figure 4.5: Datapath representation of a simple ALU capable of addition and subtraction

removed. Unremoved inputs and outputs are exposed to the external environment as expected.

Figure 4.6 shows an example of datapath which uses two half-adders from Figure 4.4. By connecting to the embedded inputs and outputs these extraneous nodes are removed, and the modularity hidden from the external environment.

Configurations for embedded datapaths may be constructed directly, or built modularly with the following functions:

```
concatConfig :: Config -> Config -> Config
concatConfigs :: [Config] -> Config
--combines two or more configurations
subConfig :: Config -> NodeName -> Config
subConfigs :: Config -> [NodeName] -> Config
--prefixes a config for use with the given module(s)
```

4.3.2 Black Box Approach

By building on this approach we can meet our requirement of defining functional units as "black boxes." For example, suppose we wished to model a datapath that included a functional unit which performed a complex encryption algorithm. Suppose that the unit was fully pipelined and calculated encryption with a latency of 12 clock cycles. We could represent the application of encryption function by defining a symbolic function `ENC` where

```
ENC key data = FuncApp ENC [key, data]
```

Then we could represent a twelve latency pipelined functional unit with the datapath shown in Figure 4.7. As the usage pattern in Table 4.3 shows the datapath clearly represents a pipelined unit with a latency of twelve.

```
embed_test :: Datapath
embed_test = makeDatapath dp
 where
  dp = do
   inputs ["X", "Y", "Z", "W"]
   outputs ["R"]
   addNode (Function "XOR_0" XOR)
   addNode (Function "XOR_1" XOR)
   addNode (Function "XOR_2" XOR)
   addDP half_adder "ADD1"
   addDP half_adder "ADD2"
   connections [("X", "ADD1:P"),("Y", "ADD1:Q"), ("Z", "ADD2:P"),("W", "ADD2:Q")]
   connections_n [("ADD1:R", ("XOR_0", 0)), ("ADD1:C", ("XOR_1", 0))]
   connections_n [("ADD2:R", ("XOR_0", 1)), ("ADD2:C", ("XOR_1", 1))]
   connections_n [("XOR_0", ("XOR_2", 0)), ("XOR_1", ("XOR_2", 1))]
   connections [("XOR_2", "R")]
```

Figure 4.6: A datapath embedding the half-adder from Figure 4.4.

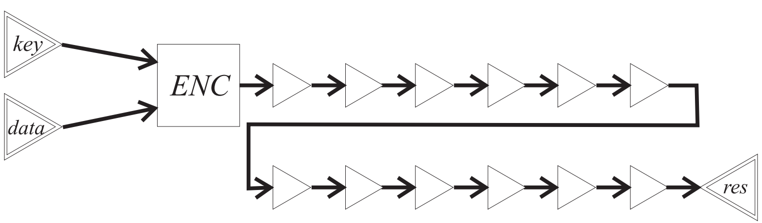


Figure 4.7: Black-box representation of 12-latency pipelined encryption unit

Further, we can model units with reduced throughput through the use of join point nodes. For example, what if the system architect later determined that it is preferable to use an un-pipelined encryption unit, which takes four clock cycles to complete an encrypt operation. We could represent this unit by the datapath shown in Figure 4.8 with usage pattern in Table 4.4. Note that because the join points' control decisions are not exposed to the external environment this graph represents an inflexible datapath. Also note that this corresponds to a unit which outputs the same result over four clock cycles. If the unit we wish to model instead output "garbage" data for three clock cycles and the result on the fourth we could represent that by changing the **False** values of the leftmost delay chain to **Open**.

This approach can be extended to apply to functional units with strange or irregular usage patterns. Consider if the encryptor unit had the usage pattern shown in Table 4.5. This unit could be modeled with the datapath in Figure 4.9.

CoreLoom provides the following functions for defining black box functional units (`UsagePattern.hs`, Appendix):

```
bbPipelined :: NodeName -> SymbFunc -> Latency -> Datapath
--Defines a pipelined datapath with TP of 1 and the given latency
```

CC	key	$data$	res
0	key_0	$data_0$	
1	key_1	$data_1$	
2	key_2	$data_2$	
3	key_3	$data_3$	
4	key_4	$data_4$	
5	key_5	$data_5$	
6	key_6	$data_6$	
7	key_7	$data_7$	
8	key_8	$data_8$	
9	key_9	$data_9$	
10	key_{10}	$data_{10}$	
11	key_{11}	$data_{11}$	
12	key_{12}	$data_{12}$	$ENC(key_0, data_0)$
13	key_{13}	$data_{13}$	$ENC(key_1, data_1)$
14	key_{14}	$data_{14}$	$ENC(key_2, data_2)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 4.3: Usage pattern for encryptor datapath in Figure 4.7.

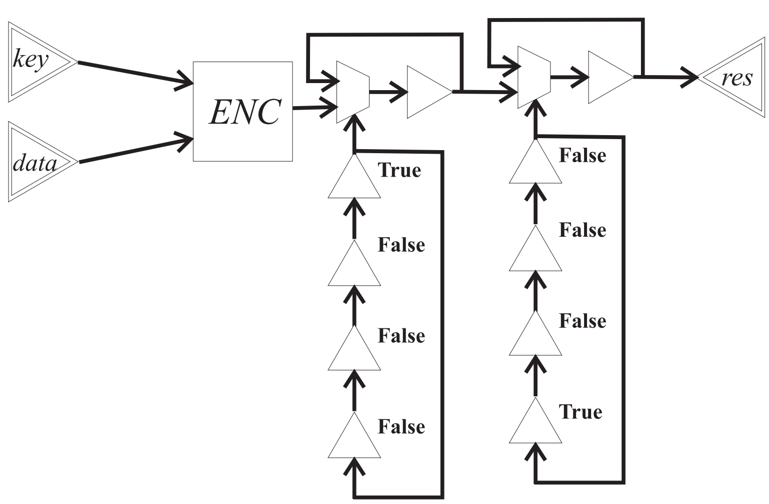


Figure 4.8: Black-box representation of un-pipelined encryption unit with throughput of 1/4.

CC	key	$data$	res
0	key_0	$data_0$	
1			
2			
3			
4	key_4	$data_4$	$ENC(key_0, data_0)$
5			$ENC(key_0, data_0)$
6			$ENC(key_0, data_0)$
7			$ENC(key_0, data_0)$
8	key_8	$data_8$	$ENC(key_4, data_4)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 4.4: Usage pattern of encryption datapath from Figure 4.8.

CC	key	$data$	res
0	key_0	$data_0$	
1	key_1	$data_1$	
2			$ENC(key_0, data_0)$
3			$ENC(key_1, data_1)$
4	key_4	$data_4$	
5	key_5	$data_5$	
6			$ENC(key_4, data_4)$
7			$ENC(key_5, data_5)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 4.5: Encryptor usage pattern which cannot be described solely in terms of throughput.

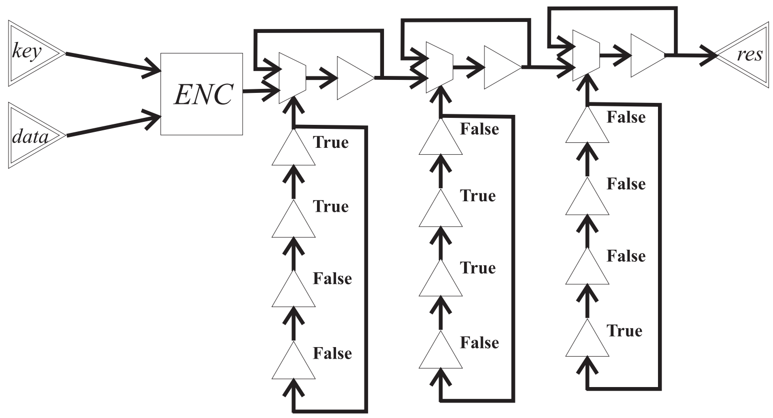


Figure 4.9: Black-box representation of usage pattern in Table 4.5.

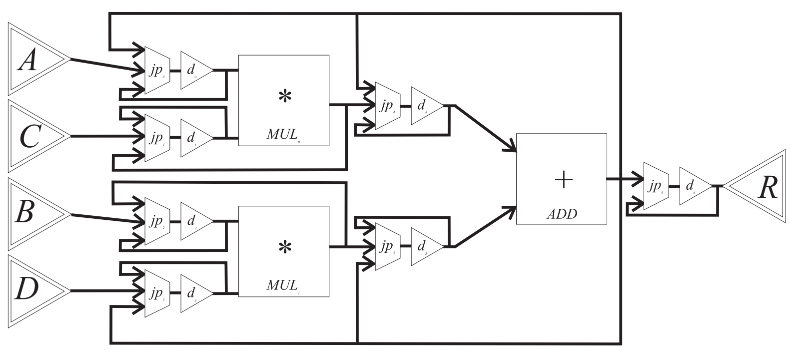


Figure 4.10: Flexible dot-product datapath with registers.

```
bbUnpipelined :: NodeName -> SymbFunc -> Latency -> TP -> Datapath
--Defines an un-pipelined datapath with given latency and TP.
--TP must be <= 1.
```

The name is used as a basis for all nodes in the generated datapath. Input and output nodes are numbered from zero.

Beyond this capability, it may be desirable to generate datapaths from arbitrary usage patterns. This seems to be a hard problem - perhaps as hard as hardware synthesis itself. Although we will not detail it here, one moderately succesful approach is outlined in the function `bbUsagePattern` (`UsagePattern.hs`, Appendix).

4.4 Configuration Scheduling

Earlier it was specified that when the end of a configuration is reached it should repeat from the beginning. However, a simple repetition often does not maximize the potential throughput of the configuration. The datapath in Figure 4.10 is an adaption of our original flexible dot product datapath from Figure 3.4. We have changed its delays to registers (Section 4.2.4) and added the ability to feed the adder output back into itself. Consider the configuration for the datapath shown in Table 4.6.

A simple repetition schedule of this configuration would result in the usage pattern given in Table 4.7, with a latency of 3 and throughput of $\frac{1}{3}$. However, it is apparent that we can do better than this simply by taking advantage of the pipelined nature of the datapath and moving the schedule up (Table 4.8). The result is a usage pattern where the latency remains 3 but the throughput is now improved to the optimal value of 1.

While we can easily schedule configurations through concatenation, there are other configurations which are most effectively scheduled by overlapping their execution. Recall the configuration from Section 4.1.1, which described a series of choices that could be made while executing our flexible dot-product unit. This configuration, translated to our new datapath, is shown in Figure 4.9.

A naive concatenation schedule results in the usage pattern given in Table 4.10, with a throughput of $\frac{1}{3}$. By overlapping the scheduling we can improve that to $\frac{1}{2}$ (Table 4.11).

In such a situation the concepts of throughput and latency become ambiguous. The first result is available on clock cycle 5. Does latency measure the number of clock cycles from the initial input of data on clock cycle 0, or from the last input of relevant data on clock cycle 2? As for throughput, we are providing input on every clock cycle but receiving results on every 2 out of every 4 clock cycles. What does this mean in terms of overall throughput?

In CoreLoom by convention the term latency refers to the number of clock cycles from the introduction of the first input to the production of the result, whereas throughput is used with respect to the output values only.

4.5 Model Conclusions

The semantic model fulfills the requirements outlined in Section 2.3. It allows us to model fine-grained flexible datapaths and to represent the choices that an external environment can make as configurations. It also allows us to reason about and optimize these configurations.

The main drawbacks to the model relate to the assumptions that are necessary, as outlined in Section 3.2.7. In particular, because it does not represent values as individual signals (as in a hardware description language such as VHDL) it cannot accurately model datapaths that, for example, split a 32-bit number into four 8-bit sections, transform the sections individually, and recombine them into a new single value.

The most significant consequence of this is that it cannot reason about configurations that are specific to bit width. For example, it cannot model a configuration that used a 64-bit XOR unit as two 32-bit XOR units, even if such a thing was possible.

With the addition of such practical features the CoreLoom model should be complete enough to serve as a means to describe core datapaths to a compiler, and as a means for the compiler to schedule algorithms. This should also make it possible to translate

jp_0	jp_1	jp_2	jp_3	jp_4	jp_5	jp_6
A	C	B	D			
				MUL_0	MUL_1	
						ADD

Table 4.6: Simple configuration for datapath from Figure 4.10.

CC	A	B	C	D	R	jp_0	jp_1	jp_2	jp_3	jp_4	jp_5	jp_6
0	A_0	B_0	C_0	D_0		A	C	B	D			
1										MUL_0	MUL_1	
2												ADD
3	A_3	B_3	C_3	D_3	$A_0C_0 + B_0D_0$	A	C	B	D			
4										MUL_0	MUL_1	
5												ADD
6					$A_3C_3 + B_3D_3$	A	C	B	D			
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 4.7: Usage pattern of a simple repetition of the configuration.

CC	A	B	C	D	R	jp_0	jp_1	jp_2	jp_3	jp_4	jp_5	jp_6
0	A_0	B_0	C_0	D_0		A	C	B	D			
1	A_1	B_1	C_1	D_1		A	C	B	D	MUL_0	MUL_1	
2	A_2	B_2	C_2	D_2		A	C	B	D	MUL_0	MUL_1	ADD
3	A_3	B_3	C_3	D_3	$A_0C_0 + B_0D_0$	A	C	B	D	MUL_0	MUL_1	ADD
4	A_4	B_4	C_4	D_4	$A_1C_1 + B_1D_1$	A	C	B	D	MUL_0	MUL_1	ADD
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 4.8: Usage pattern of optimal repetition of the configuration.

jp_0	jp_1	jp_2	jp_3	jp_4	jp_5	jp_6
A	C	B	D			
				MUL_0	MUL_1	
ADD	C	B	D			
				MUL_0	MUL_1	
						ADD

Table 4.9: Configuration for datapath 4.10 of the exclusion decisions from Figure 4.3.

CC	A	B	C	D	R	jp_0	jp_1	jp_2	jp_3	jp_4	jp_5	jp_6
0	A_0	B_0	C_0	D_0		A	C	B	D			
1										MUL_0	MUL_1	
2		B_2	C_2	D_2		ADD	C	B	D			
3	A_3	B_3	C_3	D_3		A	C	B	D	MUL_0	MUL_1	
4										MUL_0	MUL_1	ADD
5		B_5	C_5	D_5	$(A_0C_0 + B_0D_0)\,C_2$ $+B_2D_2$	ADD	C	B	D			
6	A_6	B_6	C_6	D_6		A	C	B	D	MUL_0	MUL_1	
7										MUL_0	MUL_1	ADD
8		B_8	C_8	D_8	$(A_3C_3 + B_3D_3)\,C_5$ $+B_5D_5$	ADD	C	B	D			
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 4.10: Usage pattern for a naive concatenation scheduling of this configuration.

CC	A	B	C	D	R	jp_0	jp_1	jp_2	jp_3	jp_4	jp_5	jp_6
0	A_0	B_0	C_0	D_0		A	C	B	D			
1	A_1	B_1	C_1	D_1		A	C	B	D	MUL_0	MUL_1	
2		B_2	C_2	D_2		ADD	C	B	D	MUL_0	MUL_1	
3		B_3	C_3	D_3		ADD	C	B	D	MUL_0	MUL_1	
4	A_4	B_4	C_4	D_4		A	C	B	D	MUL_0	MUL_1	ADD
5	A_5	B_5	C_5	D_5	$(A_0C_0 + B_0D_0)\,C_2$ $+B_2D_2$	A	C	B	D	MUL_0	MUL_1	ADD
6		B_6	C_6	D_6	$(A_1C_1 + B_1D_1)\,C_3$ $+B_3D_3$	ADD	C	B	D	MUL_0	MUL_1	
7		B_7	C_7	D_7		ADD	C	B	D	MUL_0	MUL_1	
8	A_8	B_8	C_8	D_8	$(A_4C_4 + B_4D_4)\,C_6$ $+B_6D_6$	A	C	B	D	MUL_0	MUL_1	ADD
9	A_9	B_9	C_9	D_9	$(A_5C_5 + B_5D_5)\,C_7$ $+B_7D_7$	A	C	B	D	MUL_0	MUL_1	ADD
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 4.11: Usage pattern of an optimal overlapping schedule of the configuration.

CoreLoom into VHDL skeletons for implementation.

Finally, for projects like FlexSoC it should be possible to automatically generate specifications for control units and communication units from a CoreLoom description. This could ease the implementation burden of using such a system.

5 Complete Examples

This chapter describes the process that architects or designers might use to model datapath cores in CoreLoom, investigate their properties, and add flexible features to them.

5.1 Combined AES Core

For the first example we have chosen to explore the open source implementation of Advanced Encryption System by Rudolf Usselmann [14]. This core has been donated to the OpenCores project [16] and thus gives us an opportunity to study something similar to a industry datapath.

Usselmann's project provides two separate cores for encryption and decryption. The datapaths contain large degree of overlapping hardware, including the key expansion and S-Box units required by the AES standard [13]. If a system architect knows that these two operations will never be performed concurrently then they may wish to investigate the possibility of combining the two cores into a flexible unit which can switch back and forth between the two modes.

The combined unit will certainly take up more circuit area than either of the original units, but may be more efficient than implementing them both. More importantly, by combining the two units the architect can minimize the number of cores connected to an interconnect network, which research indicates to be a significant source of power consumption [17].

5.1.1 Cipher Core

The general algorithm that Usselmann's encryption core implements is as follows:

```
1. Expand Key
2. Perform initial permutation
3. Repeat round permutation (10 times)
4. Perform final permutation
```

According to the documentation the core follows the timing characteristics shown in Figure X.

Figure 5.2 shows the model of this core in CoreLoom. It uses undefined functions to represent the stages of the encryption algorithm. The extraneous signals *ld* and *done* have been removed. The model results in the usage pattern given in Table 5.1, which shows that the representation successfully models both the timing and algorithm of the original core.

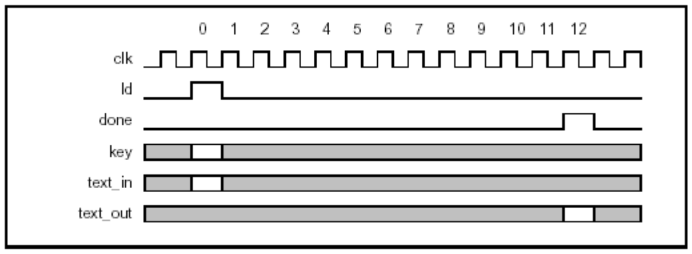


Figure 5.1: Timing characteristics of OpenCores.org AES cipher core. Source: [14]

```
aes_enc :: Datapath
aes_enc = makeDatapath dp
 where
  dp = do
   inputs ["key", "text_in"]
   outputs ["text_out"]
   function (SymbFunc "KeyExpansion" 1) "EXP"
   function (SymbFunc "InitialPermutation" 2) "INIT"
   function (SymbFunc "RoundPermutation" 2) "ROUND"
   function (SymbFunc "FinalPermutation" 2) "FINAL"
   register open "R0"
   register open "R1"
   delay open "R2"
   register open "R3"
   joinPoint "JP"
   connections [("key", "R0_load"), ("R0", "EXP"), ("text_in", "R1_load")]
   connectTo_n "EXP" [("INIT", 0), ("ROUND", 0), ("FINAL", 0)]
   connections_n [("R1", ("INIT", 1)), ("INIT", ("JP", 0)), ("JP", ("ROUND", 1))]
   connections_n [("ROUND", ("R2", 0)), ("R2", ("JP", 0)), ("R2", ("FINAL", 1))]
   connections_n [("FINAL", ("R3_load", 0)), ("R3", ("text_out", 0))]
   constantList ((Var "key") : (replicate 11 (Var "R0"))) "ZeroList"
   constantList ((Var "text_in") : (replicate 11 (Var "R1"))) "OneList"
   constantList ([Var "R2", Var "INIT"] ++ (replicate 10 (Var "R2"))) "TwoList"
   constantList ((replicate 11 open) ++ [Var "FINAL"]) "ThreeList"
   connect_n ("ZeroList_0", ("R0_load", 1))
   connect_n ("OneList_0", ("R1_load", 1))
   connect_n ("TwoList_0", ("JP", 1))
   connect_n ("ThreeList_0", ("R3_load", 1))
```

Figure 5.2: CoreLoom implementation of AES cipher core.

CC	key	$text\_in$	$text\_out$
0	key_0	$text_0$	
1			
$\vdots$	$\vdots$	$\vdots$	$\vdots$
11			
12	key_1	$text_1$	$encrypt(key_0, text_0)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

where $encrypt(key_0, text_0) =$

$$\begin{aligned}&FinalPermutation(KeyExpansion(key_0),\ RoundPermutation(KeyExpansion(key_0),\\ &RoundPermutation(KeyExpansion(key_0),\ RoundPermutation(KeyExpansion(key_0),\\ &RoundPermutation(KeyExpansion(key_0),\ RoundPermutation(KeyExpansion(key_0),\\ &RoundPermutation(KeyExpansion(key_0),\ RoundPermutation(KeyExpansion(key_0),\\ &RoundPermutation(KeyExpansion(key_0),\ RoundPermutation(KeyExpansion(key_0),\\ &RoundPermutation(KeyExpansion(key_0),\ InitialPermutation(KeyExpansion(key_0),\ text_0))))))))))))\end{aligned}$$

Table 5.1: Usage pattern for CoreLoom model of AES encryption core.

5.1.2 Inverse Cipher Core

The decryption core is slightly complicated by the fact that the key must be delivered in reverse order. This means that it must be loaded first, but then can be reused until a new key is needed. We will model this by exposing the load signal of the register that stores the key (`R0`) - analogous to the ld_key input.

The overall decryption algorithm is as follows:

```
1. Expand Key
2. Reverse Key
3. Load Text
4. Perform initial permutation
5. Repeat round permutation (10 times)
6. Perform final permutation
```

The CoreLoom model of this unit has been omitted for space considerations, but may be found in `Examples.hs` (Appendix).

5.1.3 Combined Flexible Unit

With the two individual models investigated we are now able to combine them into a single flexible unit. Figure 5.3 shows the result of this combination.

The new unit is able to perform the same encryption and decryption operations as the separate cores with identical usage patterns, as demonstrated by the cipher configuration

```
aes_comb :: Datapath
aes_comb = makeDatapath dp
 where
  dp = do
   inputs ["key", "text_in"]
   outputs ["text_out"]
   function (SymbFunc "KeyExpansion" 1) "EXP"
   function (SymbFunc "KeyReversal" 1) "REV"
   function (SymbFunc "InitialPermutation" 2) "INIT"
   function (SymbFunc "RoundPermutation" 2) "ROUND"
   function (SymbFunc "FinalPermutation" 2) "FINAL"
   function (SymbFunc "InitialInverse" 2) "INIT_I"
   function (SymbFunc "RoundInverse" 2) "ROUND_I"
   function (SymbFunc "FinalInverse" 2) "FINAL_I"
   register open "R0"
   register open "R1"
   delay open "R2"
   delay open "R3"
   register open "R4"
   joinPoints ["JP", "JP2"]
   connections [("key", "R0_load"), ("R0", "EXP")]
   connections [("EXP", "REV"), ("text_in", "R1_load")]
   connectTo_n "EXP" [("INIT", 0), ("ROUND", 0), ("FINAL", 0)]
   connectTo_n "REV" [("INIT_I", 0), ("ROUND_I", 0), ("FINAL_I", 0)]
   connections_n [("R1", ("INIT", 1)), ("INIT", ("JP", 0)), ("JP", ("ROUND", 1))]
   connections_n [("ROUND", ("R2", 0)), ("R2", ("JP", 0)), ("R2", ("FINAL", 1))]
   connections_n [("R1", ("INIT_I", 1)), ("INIT_I", ("JP2", 0))]
   connections_n [("JP2", ("ROUND_I", 1)), ("ROUND_I", ("R3", 0)), ("R3", ("JP2", 0))]
   connections_n [("R3", ("FINAL_I", 1)), ("FINAL", ("R4_load", 0))]
   connections_n [("FINAL_I", ("R4_load", 0)),("R4", ("text_out", 0))]
   constantList ([Var "R2", Var "INIT"] ++ (replicate 10 (Var "R2"))) "TwoList"
   constantList ([Var "R3", Var "INIT_I"] ++ (replicate 10 (Var "R3"))) "ThreeList"
   connect_n ("TwoList_0", ("JP", 1))
   connect_n ("ThreeList_0", ("JP2", 1))
```

Figure 5.3: Combined Encryption and Decryption core.

and usage pattern in Table 5.2. It is up to the system architect to decide whether the benefits of combined unit (potentially reduced total circuit area, fewer cores on the interconnect) outweigh the impacts on power consumption and critical path the more general core will certainly impose.

5.2 Flexible FIR Filter

One effect of the combined core above is that exposes more exclusion decisions to the external environment. Unfortunately, there is not much chance that a compiler will want to use the special-purpose hardware of the Rijndael cipher for other algorithms. However, in the case of cores which use more general hardware the exposure of these decisions may result in flexibility benefits to the entire system.

```
aes_comb_enc :: Config
aes_comb_enc = (["R0_load", "R1_load", "R4_load"],
               [[Var "key", Var "text_in", open],
                [Var "R0", Var "R1", open],
                ... ×9 ...
                [Var "R0", Var "R1", Var "FINAL"]])
```

CC	key	$text\_in$	$text\_out$
0	key_0	$text_0$	
1			
$\vdots$	$\vdots$	$\vdots$	$\vdots$
11			
12	key_1	$text_1$	$encrypt(key_0, text_0)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 5.2: Configuration and Usage Pattern for cipher operation for combined AES core.

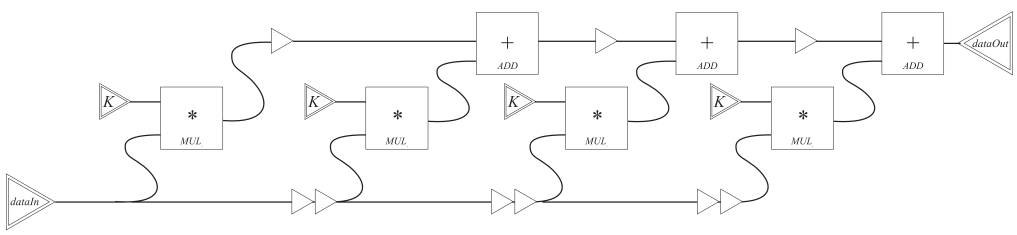


Figure 5.4: General four-coefficient FIR Filter.

One algorithm which makes use of general hardware is Finite Impulse Response. While there are many ways to implement FIR filters which are very specialized, we will consider a more general implementation as described by the RaPiD project [5, 4]. This implementation contains adders and multipliers arranged in a pipeline, as shown in Figure 5.4.

A system architect may use such a unit as part of a larger system-on-chip. They may also require the ability to do general vector addition and multiplication. If it is determined that the vector operations will not happen concurrently with the FIR operation then it is natural to explore the solution of using the FIR filter's adders and multipliers to implement these operations. Using CoreLoom we can show how the inflexible FIR filter can be expanded to support vector addition and multiplication, as well as dot product, and the flexibility to use less of the pipeline. The architect must then determine whether the flexible implementation is more suitable than using separate cores.

For space considerations we will restrict ourselves to a small FIR Filter with four coefficients, meaning that it also deals most efficiently with vectors of length four. It should be straightforward to extend the concepts to wider units.

```
fir :: Datapath
fir = makeDatapath dp
 where
  dp = do
   inputs ["Data_in", "K_0", "K_1", "K_2", "K_3"]
   outputs ["Data_out"]
   functions MUL ["M_0", "M_1", "M_2", "M_3"]
   functions ADD ["A_1", "A_2", "A_3"]
   delayN 6 open "D"
   delays open ["S_0", "S_1", "S_2"]
   connect ("Data_in", "D_5")
   connections_n [("Data_in", ("M_0", 1)), ("D_4", ("M_1", 1))]
   connections_n [("D_2", ("M_2", 1)), ("D_0", ("M_3", 1))]
   connections_n [("K_0", ("M_0", 0)), ("K_1", ("M_1", 0))]
   connections_n [("K_2", ("M_2", 0)), ("K_3", ("M_3", 0))]
   connections_n [("M_0", ("S_0", 0)), ("M_1", ("A_1", 1))]
   connections_n [("M_2", ("A_2", 1)), ("M_3", ("A_3", 1))]
   connections_n [("S_0",("A_1", 0)),("S_1", ("A_2", 0)),("S_2", ("A_3", 0))]
   connections [("A_1", "S_1"), ("A_2", "S_2"), ("A_3", "Data_out")]
```

Figure 5.5: FIR Filter implementation in CoreLoom.

5.2.1 FIR Filter in CoreLoom

The regular nature of the FIR pipeline means that we can model it concisely by using CoreLoom modular datapaths (Section 4.3.1). However, as we add flexible features the datapath will become increasingly less regular, therefore we have chosen to model it as a straightforward implementation of adders and multipliers, as shown in Figure 5.5. The usage pattern in Table 5.3 confirms that the model implements the FIR filter algorithm with a latency of 6 and a throughput of 1.

5.2.2 Implementing Vector Operations

We will begin by adding the capability to perform the following simple vector-scalar operations:

$$[x\ y\ z\ w] * n = [xn\ yn\ zn\ wn]$$
$$[x\ y\ z\ w] + n = [x{+}n\ y{+}n\ z{+}n\ w{+}n]$$

To implement multiplication we must add new inputs and outputs to deliver the vectors on each clock cycle. For the input vector we can use the FIR Coefficient inputs `K0 - K3`, as they are already directly connected to the multipliers. For the scalar input we will use the existing `Data_in` input, which now will be connected to the multipliers. To deliver the results we must add new output nodes from the multipliers, named `Mul_out`. The result is the datapath shown in Figure 5.6. It can perform vector multiplication with latency of zero and throughput of 4 (as there are 4 outputs delivering results on each clock cycle), or 1 vector of length 4 per clock cycle.

CC	*DataIn*	*DataOut*
0	$DataIn_0$	
1	$DataIn_1$	
2	$DataIn_2$	
3	$DataIn_3$	
4	$DataIn_4$	
5	$DataIn_5$	
6	$DataIn_6$	$((DataIn_3 * FIR_3K) + (DataIn_2 * FIR_2K) +$ $(DataIn_1 * FIR_1K) + (DataIn_0 * FIR_0K))$
7	$DataIn_7$	$((DataIn_4 * FIR_3K) + (DataIn_3 * FIR_2K) +$ $(DataIn_2 * FIR_1K) + (DataIn_1 * FIR_0K))$
8	$DataIn_8$	$((DataIn_5 * FIR_3K) + (DataIn_4 * FIR_2K) +$ $(DataIn_3 * FIR_1K) + (DataIn_2 * FIR_0K))$
$\vdots$	$\vdots$	$\vdots$

Table 5.3: Usage pattern for FIR Filter datapath.

```
flexFIR :: Datapath
flexFIR = makeDatapath dp
 where
  dp = do
   inputs ["Data_in", "K_0", "K_1", "K_2", "K_3"]
   outputs ["Data_out"]
   outputs["V_out_0", "V_out_1", "V_out_2", "V_out_3"]
   ...
   connections [("A_1", "S_1"), ("A_2", "S_2"), ("A_3", "Data_out")]
   connections_n [("Data_in", ("M_0", 1)),("Data_in", ("M_1", 1))]
   connections_n [("Data_in", ("M_2", 1)), ("Data_in", ("M_3", 1))]
   connections [("M_0", "V_out_0"), ("M_1", "V_out_1")]
   connections [("M_2", "V_out_2"), ("M_3", "V_out_3")]
```

Figure 5.6: FIR Filter with vector-scalar multiplication.

To implement vector addition we must add connections from the `Data_in` and `K` inputs to the adders. To improve throughput we will insert an adder at the beginning of the pipeline and add zero during normal FIR mode. These additions are summarized in Figure 5.7. The configuration will now specify whether we are performing addition or multiplication on each clock cycle.

Now we will add support for the following common vector-vector operations:

$$[x\ y\ z\ w] * [a\ b\ c\ d] = [xa\ yb\ zc\ wd]$$
$$[x\ y\ z\ w] + [a\ b\ c\ d] = [x{+}a\ y{+}b\ z{+}c\ w{+}d]$$

To add support for these operations we must add new inputs `Vector_in` and connect them to the adders and multipliers. The result is a datapath that can perform vector-vector multiplication and addition with a latency of zero and a throughput of 4. Note that if we were willing to add separate sets of inputs and outputs for addition and multiplication then we could perform the two operations simultaneously. At this point the surrounding system has access to the multipliers and adders directly and thus has access to a wide range of computational possibilities.

5.2.3 Implementing Three-Coefficient FIR

For example, the current version of the datapath is now able to use less of the pipeline to calculate FIR filter with a smaller number of coefficients - i.e. to save power. Here we will demonstrate how to use it with 3 coefficients instead of 4, but the concept can be applied to pipelines of any length.

To do so we simply take advantage of the new outputs `Vector_out` to deliver the results from earlier in the pipeline. By using the OPEN value we can specify that the final stages of the pipeline are unused, and are available to be disabled or used for another calculation. The result, as shown in Table 5.4, is a FIR calculation with throughput of 1, but with a reduced latency of 4.

5.2.4 Implementing Dot Product

Finally, we will use both the adders and multipliers to implement dot-product of vectors of length three:

$$[x\ y\ z] \bullet [a\ b\ c\ d] = xa + yb + zc + wd$$

a common calculation in the domain of computer graphics. Note that the result of this operation is a single scalar value, so we can deliver it with the output `Vector_out_2`. We will implement the operation in a pipelined manner similar to the original FIR operation. To sync up the incoming data we will add delays after the `K` input, which will be bypassed during other operations. Connections from the `Vector_in` inputs will go earlier into the pipeline so that they meet the correct incoming values. The final datapath is shown in Figure 5.8. It implements dot-product with a latency of 2 and a throughput of 1.

```
flexFIR :: Datapath
flexFIR = makeDatapath dp
 where
  dp = do
   functions ADD ["A_0", "A_1", "A_2", "A_3"]
   constant (VInt 0) "Zero"
   ...
   connections_n [("K_0", ("A_0", 0)),("K_1", ("A_1", 0))]
   connections_n [("K_2", ("A_2", 0)), ("K_3", ("A_3", 0))]
   connections_n [("Data_in", ("A_0", 1)),("Data_in", ("A_1", 1))]
   connections_n [("Data_in", ("A_2", 1)), ("Data_in", ("A_3", 1))]
   connections [("A_0", "V_out_0"), ("A_1", "V_out_1")]
   connections [("A_2", "V_out_2"), ("A_3", "V_out_3")]

--Configuration for Vector-Scalar Multiplication
cf_VSMUL :: Config
cf_VSMUL = (["into_(M_0,1)", "into_(M_1,1)", "into_(M_2,1)", "into_(M_3,1)",
             "into_(A_0,0)", "into_(A_1,0)", "into_(A_2,0)", "into_(A_3,0)",
             "into_(A_0,1)", "into_(A_1,1)", "into_(A_2,1)", "into_(A_3,1)",
             "into_(V_out_0,0)", "into_(V_out_1,0)", "into_(V_out_2,0)",
             "into_(V_out_3,0)"],
            [[Var "Data_in", Var "Data_in", Var "Data_in", Var "Data_in",
              open,      open,      open,      open,
              open,      open,      open,      open,
              Var "M_0", Var "M_1", Var "M_2", Var "M_3"]])

--Configuration for Vector-Scalar Addition
cf_VSADD :: Config
cf_VSADD = (["into_(M_0,1)", "into_(M_1,1)", "into_(M_2,1)", "into_(M_3,1)",
             "into_(A_0,0)", "into_(A_1,0)", "into_(A_2,0)", "into_(A_3,0)",
             "into_(A_0,1)", "into_(A_1,1)", "into_(A_2,1)", "into_(A_3,1)",
             "into_(V_out_0,0)", "into_(V_out_1,0)", "into_(V_out_2,0)",
             "into_(V_out_3,0)"],
            [[open,      open,      open,      open,
              Var "K_0", Var "K_1", Var "K_2", Var "K_3",
              Var "Data_in", Var "Data_in", Var "Data_in", Var "Data_in",
              Var "A_0", Var "A_1", Var "A_2", Var "A_3"]])
```

Figure 5.7: Flexible FIR filter with vector addition.

```
cf_FIR3 :: Config
cf_FIR3 = (["into_(M_0,1)", "into_(M_1,1)", "into_(M_2,1)", "into_(M_3,1)",
            "into_(A_0,0)", "into_(A_1,0)", "into_(A_2,0)", "into_(A_3,0)",
            "into_(A_0,1)", "into_(A_1,1)", "into_(A_2,1)", "into_(A_3,1)",
            "into_(V_out_0,0)", "into_(V_out_1,0)", "into_(V_out_2,0)",
            "into_(V_out_3,0)"],
           [[Var "Data_in", Var "D_4", Var "D_2", open,
             Var "Zero",    Var "S_0", Var "S_1", open,
             Var "M_0",     Var "M_1", Var "M_2", open,
             open,          open,      Var "A_2", open]])
```

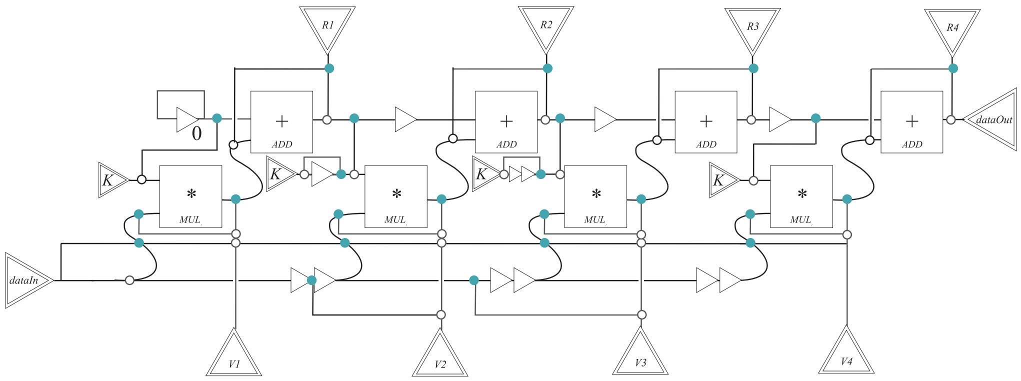
CC	*DataIn*	*DataOut*
0	$DataIn_0$	
1	$DataIn_1$	
2	$DataIn_2$	
3	$DataIn_3$	
4	$DataIn_4$	$(DataIn_2 * FIR_2K) + (DataIn_1 * FIR_1K) + (DataIn_0 * FIR_0K)$
5	$DataIn_5$	$(DataIn_3 * FIR_2K) + (DataIn_2 * FIR_1K) + (DataIn_1 * FIR_0K)$
⋮	⋮	⋮

Table 5.4: Configuration and usage pattern for three-coefficient FIR.

```
flexFIR :: Datapath
flexFIR = makeDatapath dp
 where
  dp = do
     ...
     delayN 5 open "D"
     delay open "B_0"
     delayN 1 open "C"
     ...
     connections [("K_1", "B_0"), ("K_2", "C_1")]
     connections_n [("B_0", ("M_1", 0)), ("C_0", ("M_2", 0))]
     connections [("V_in_1", "D_4"), ("V_in_2", "D_3")]
      ...
```

CC	V_in_0	V_in_1	V_in_2	K_0	K_1	K_2	V_out_2
0	$V\_in\_0_0$	$V\_in\_1_0$	$V\_in\_2_0$	K_0_0	K_1_0	K_2_0	
1	$V\_in\_0_0$	$V\_in\_1_0$	$V\_in\_2_0$	K_0_0	K_1_0	K_2_0	
2	$V\_in\_0_0$	$V\_in\_1_0$	$V\_in\_2_0$	K_0_0	K_1_0	K_2_0	$V\_in\_0_0 K\_0_0 +$ $V\_in\_1_0 K\_1_0 +$ $V\_in\_2_0 K\_2_0$
3	$V\_in\_0_0$	$V\_in\_1_0$	$V\_in\_2_0$	K_0_0	K_1_0	K_2_0	$V\_in\_0_1 K\_0_1 +$ $V\_in\_1_1 K\_1_1 +$ $V\_in\_2_1 K\_2_1$
⋮	⋮						⋮

Figure 5.8: Adding support for dot-product to flexFIR.



Configuration	Inputs	Outputs	Latency	Throughput
`cf_FIR`	$Data\_in$, K_0 - 3	$Data\_out$	6	1
`cf_VSMUL`	$Data\_in$, K_0 - 3	V_out_0 - 3	0	4
`cf_VSADD`	$Data\_in$, K_0 - 3	V_out_0 - 3	0	4
`cf_VVMUL`	V_in_0 -3, K_0 - 3	V_out_0 - 3	0	4
`cf_VVADD`	V_in_0 -3, K_0 - 3	V_out_0 - 3	0	4
`cf_FIR3`	$Data\_in$, K_0 - 2	V_out_2	4	1
`cf_DPROD`	$Data\_in$, K_0 - 2	V_out_2	2	1

The configurations may be found in the file `FlexFIR.hs` (Appendix).

Figure 5.9: Complete flexFIR datapath.

5.3 Example Conclusions

A diagram of the complete `flexFIR` filter is shown in Figure 5.9, along with a summary of its capabilities. Complete code for the datapath and configurations may be found in the file `FlexFIR.hs` (Appendix). Note that the `flexFIR` datapath is able to perform the FIR filter operation with the exact same usage characteristics as the original, inflexible datapath from Figure 5.5. With further alterations the datapath could be made to support algorithms such as vector cross-product, or matrix multiplication.

These examples demonstrate how an architect may use CoreLoom to add flexibility to an existing datapath. However, the physical characteristics of the flexible core are sure to vary widely from the original version, and may perhaps even vary from one configuration to another. The architects themselves must determine whether the combined core offers better characteristics to the overall system than separate, inflexible implementations.

6 Discussion

In this chapter we will discuss some of the interesting issues that the CoreLoom model reveals, as well as present related work and conclusions.

6.1 Representing Functional Properties of Configurations

We could think of the datapath as defining a language of configurations specific to it. However, depending on the datapath, these configurations have functional properties (representing what the datapath is calculating) but they may also have non-functional properties, such as throughput or latency.

Consider datapath shown in Figure 6.1. We could use this datapath to calculate $f(X_0)$ with the following configuration:

d_0load	d_1load
FU	d_1
	d_0

We could also use this configuration, with greater latency:

d_0load	d_1load
FU	d_1
d_0	d_1
	d_0

The second configuration has the same functional properties as the first one, but it waits one clock cycle before transferring the data between the registers, thus altering its non-functional properties. In fact, there are infinite numbers of configurations for this datapath that calculate $f(X_0)$, each one simply waiting an extra clock cycle. Schedulers must pick the configuration which both expresses the proper algorithm combined with the desired non-functional properties.

Functionally, many datapaths express *stuck* configurations, which do not successfully move data through the datapath. For example, for the above datapath the configuration:

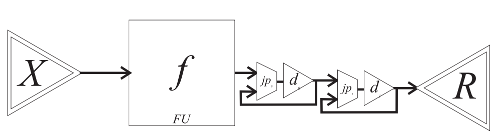
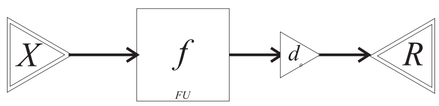


Figure 6.1: Datapath that demonstrates non-functional properties.

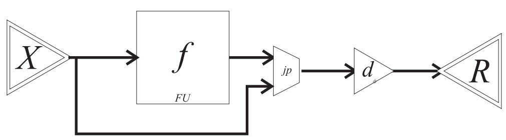
$d_0 load$	$d_1 load$
d_0	d_1

will never result in data ever successfully reaching the output node. Indeed, there are an infinite number of this type of configurations for this datapath as well.

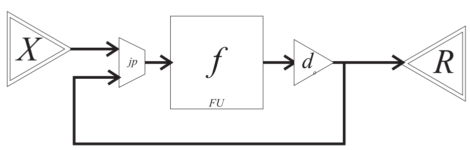
It is natural to attempt to describe the non-stuck configurations that a datapath accepts as a formal language, based on which functions the datapath can apply to which values. For example, the simple datapath:



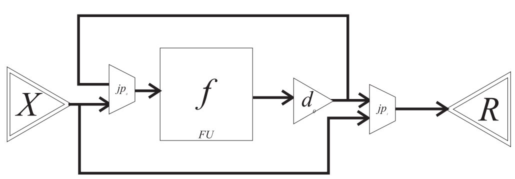
could be said to calculate $R = f(X)$. If we add flexibility in the following manner:



we would then have a choice whether or not to apply *f*, so $R = f(X) \mid X$. Alternatively, if we added flexibility as follows:



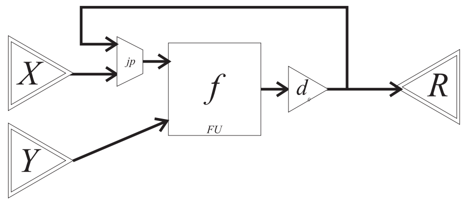
the datapath would then calculate one or more applications of *f*, which we could express as $R = f(X)^{+}$where $f(X)^2 = f(f(X))$, $f(X)^3 = f(f(f(X)))$, etc. By combining the two:



we would get $R = f(X)^{+} \mid X$, which we could naturally shorten to $R = f(X)^{*}$.

However, this regular expression approach is insufficient to deal with delay nodes. Above the delay node is just a way to avoid combinational loops. In general delay nodes present a problem because they make the representation of clock cycles explicit. Additionally, delays with the ability to feedback into themselves (registers, Section 4.2.4) present additional complications because of their ability to store calculated values and duplicate them on successive clock cycles.

Duplication of values in general means that we cannot express these configurations as a regular expression, or even a context-free language. For example, consider if the symbolic function f took two input parameters. Then we could define the following datapath:



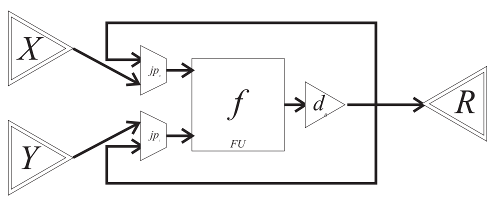
The language this datapath expresses is:

$$R = \overbrace{f(f((f(X_0, Y_0), Y_1)..., Y_n)}^{n\ applications}$$

which we could represent with a parameterized grammar:

$$\begin{aligned}
&L(n) ::= \text{“}R = \text{”}\ S(n)\\
&S(0) ::= \text{“}f(X_0, Y_0)\text{”}\\
&S(n) ::= \text{“}f(\text{”}\ S(n-1)\ \text{“}, Y_n)\text{”}
\end{aligned}$$

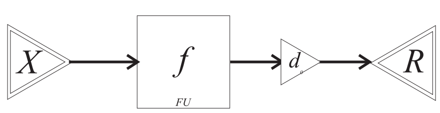
but such a representation could not deal with this datapath:



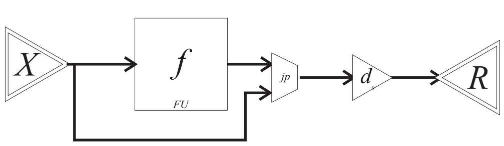
because it has the ability to duplicate the output of f at any time.

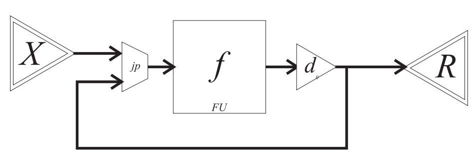
6.2 Towards a Metric of Flexibility

The reasoning in the above section begins to suggest the basis for a metric for measuring the flexibility of a datapath. The original datapath:

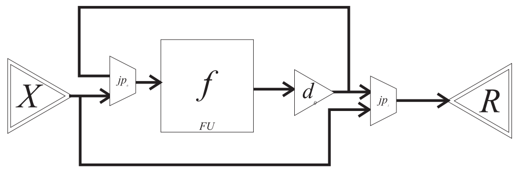


$R = f(X)$ was capable of performing only one calculation. The datapaths:

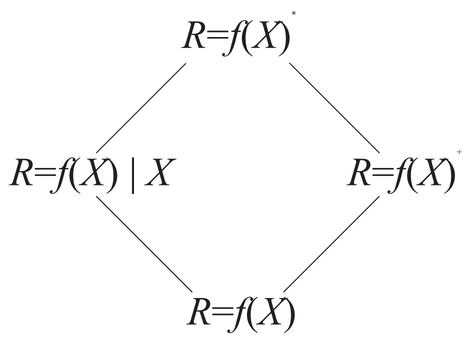




$R = f(X) \mid X$ and $R = f(X)^+$ are both capable of doing the original calculation, plus implement additional functionality. Whereas:



$R = f(X)^*$is capable of performing the operations of all of these datapaths. This is a partial order as follows:



$R = f(X) \mid X$ and $R = f(X)^+$ both express functional capability that the other cannot - we cannot judge one as more or less flexible than the other, but they are both objectively more flexible than $R = f(X)$. Trying to judge the flexibility of datapaths which have differing functional capability means that any metric of flexibility will necessarily be a partial order.

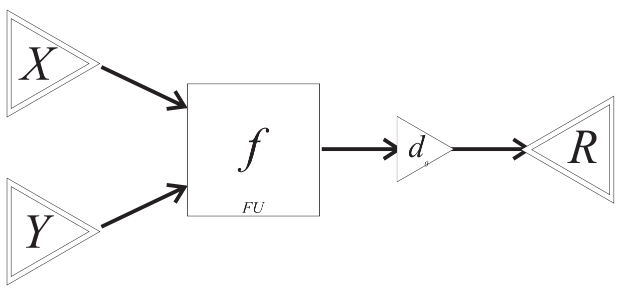
However, notice that this ordering ignores the join point nodes of the datapath. Can we really say that the $R = f(X) \mid X$ datapath is even comparable at all with $R = f(X)$ if the former includes a node that the latter does not? And yet the two are clearly functionally related.

An initial attempt then, given a datapath D, might be to define a poset $\langle S, R \rangle$where S is constructed by adding a join point node before every (non-join point) input port in the graph, then selecting every valid connection scheme. R then would be an ordering relation of graph completeness, as shown in Figure 6.2.

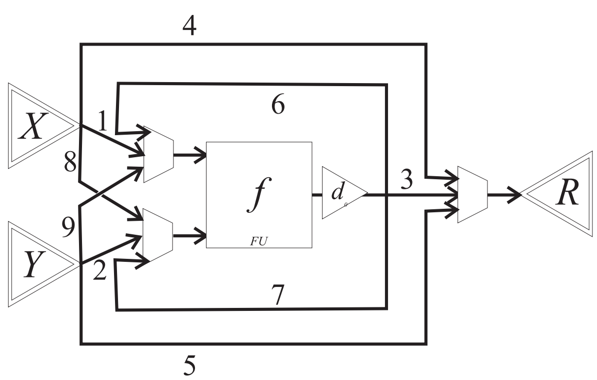
However, this approach cannot deal with the complication of delay elements. Initially delay elements seem to have no impact on the functional capabilities of a datapath. However the ability of delays - particularly registers which can store data an arbitrary length of time - to act as a memory bank to remember the result of previous calculations actually can impact the flexibility of the datapath.

Additionally, it is not clear how such a metric would judge interconnection networks in datapaths. For example, Figure 6.3 shows an altered version of the datapath from Figure 6.2. It has a join point "interconnect" which can feed into certain input ports. In this case, this datapath is functionally equivalent to position (1,2,3,4,6) on the ordering, yet it is not included in the poset. Yet only one source may drive the interconnect at a time, whereas two unrelated direct connections may be used simultaneously - thus exact positioning on a metric of flexibility remains unclear.

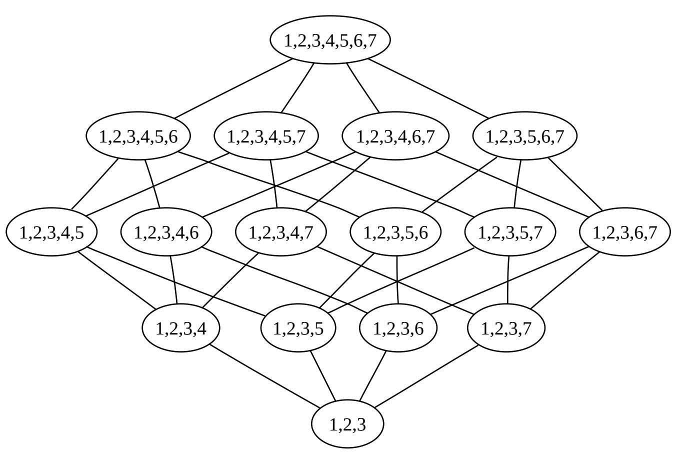
Original datapath:



Complete datapath:



Partial order of flexibility based on inclusion of connections:



(For space considerations connections 8 and 9 have been omitted.)

Figure 6.2: Complete example of a ordering of datapath flexibility.

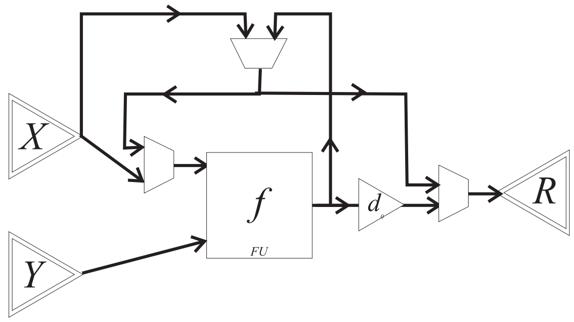


Figure 6.3: Efficient flexibility using an interconnect.

One approach that may overcome both of these limitations comes from model checking. This would involve enumerating all possible states that the datapath can be in, and judging the flexibility based on the transition relation between those states. Such an approach may also solve the problem from the previous section of representing all algorithms that a datapath can calculate, and requires in-depth exploration in the future.

6.3 Difference of Control Words in Configurations

Another issue which we can use the model to explore is how greatly control words of a configuration differ from one clock cycle to the next. Recent projects like FlexSoC (Section 1.2) seek to efficiently encode and translate configurations. The choice of compression technique will depend on the underlying features of the configurations of a typical program.

While it is too early to define the characteristics of a typical FlexSoC program we can use the CoreLoom model to perform an initial investigation of the example datapaths defined for this project. We will categorize configurations by their *width* - how many join points they control - their *length* - how many clock cycles they span - and their *variance* - how many control words are unique and how many are reused.

For the example combined AES core (Section 5.1) there are two main configurations to investigate: encoding, and decoding. These configurations do not vary greatly, as they reuse one of their units ten times to translate the data. Because of this no scheduling will improve their throughput. The encoding configuration has 12 control words, 10 of which are repeats, for a total of 3 unique control words. The decoding configuration has 24 control words, but only 5 unique control words. Overall, these configurations are long and narrow, yet do not vary greatly.

In contrast, the flexible FIR Filter from Section 5.2 has configurations of a different complexion. The configurations created for the examples in this report take advantage of that datapath's regularity - they all consist of one single control word, although those control words are the widest in the report. A contrast between the FIR and AES configurations is given in Table 6.1.

Configuration	Width	Length	Unique Control Words
`cf_aes_enc`	3	12	3
`cf_aes_dec`	3	24	5
`cf_fir_FIR`	16	1	1
`cf_fir_VSMUL`	16	1	1
`cf_fir_VSADD`	16	1	1
`cf_fir_VVMUL`	16	1	1
`cf_fir_VVADD`	16	1	1
`cf_fir_FIR3`	16	1	1
`cf_dpfir_DPROD`	20	1	1

Table 6.1: Charactestics of AES and FIR configurations.

Finally, we can explore the toy examples created for this report. The previous examples can not benefit from CoreLoom's configuration scheduling. An unscheduled configuration presented as the user would write it seems to vary greatly. The configuration in Table 4.6, for example, has 3 control words, all unique. After the configuration has been scheduled, as in Table 4.8, there still seem to remain 3 unique control words. However, closer analysis shows that the user has specified the `open` value for some choices, specifying that they are irrelevant. With careful construction we can take advantage of this fact to reduce the three unique control words to one, as demonstrated below:

jp_0	jp_1	jp_2	jp_3	jp_4	jp_5	jp_6
A	C	B	D			
A	C	B	D	MUL_0	MUL_1	
A	C	B	D	MUL_0	MUL_1	ADD
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

$$\Updownarrow$$

jp_0	jp_1	jp_2	jp_3	jp_4	jp_5	jp_6
A	C	B	D	MUL_0	MUL_1	ADD
A	C	B	D	MUL_0	MUL_1	ADD
A	C	B	D	MUL_0	MUL_1	ADD
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

$$\Updownarrow$$

jp_0	jp_1	jp_2	jp_3	jp_4	jp_5	jp_6
A	C	B	D	MUL_0	MUL_1	ADD
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Beyond repitition scheduling, CoreLoom's overlapping scheduling also can have an effect. The configuration in Table 4.9 initially has 5 control words, 3 of which are unique. However, after the configuration has been scheduled by CoreLoom (Table 4.11) it has much less variance - 4 control words, with only 2 uniques. In general the toy examples have greatly differing characteristics before and after scheduling, as shown in Table 6.2. The results indicate that configurations of this type should be compressed after being scheduled, if possible.

After investigating all of the examples created for this project (`Examples.hs` and `FlexFir.hs`, Appendix) we found three general types of configurations. First, narrow configurations that span multiple clock cycles represent unpipelined datapaths with minimal hardware that reuse their functional units to perform complex calculations. Second, wide configurations that last only one clock cycle represent long, pipelined datapaths which add hardware to increase throughput. Third, general configurations that are improved by scheduling, which seem to represent datapaths being used for configurations that they were not originally designed to support. Any project such as FlexSoC which wishes to perform compression of configurations must determine which type of configurations are most common in order to determine the most suitable compression technique.

6.4 Related Work

As a domain-specific embedded language in Haskell, CoreLoom owes much to Lava [2]. Although the visible syntax of the languages are very different, the underlying semantics are often the same. The main difference is that while Lava can describe circuits with symbolic values, it does not simulate their dataflow directly but rather uses external formal verification tools such as model checkers.

Hawk [12] is another embedded language in Haskell which is meant to describe hardware. The focus of Hawk is in describing microprocessor pipelines at a high level through transactions, and has not yet been used to describe reconfigurable datapaths.

CoreLoom also bears resemblance to the language Lustre [6]. Both describe hardware datapaths with implicit clocks that operate on streams of values. One difference is that CoreLoom's semantics generate input values themselves in the form of symbolic values, whereas in Lustre inputs are generally supplied by the user. Overall, CoreLoom sacrifices Lustre's generality for a strict semantic separation between data and configuration streams. CoreLoom's main contribution is the ability to extract a formal usage pattern, with characteristics such as throughput and latency, from a datapath definition and a configuration.

BlueSpec [1] is a Haskell-like language that uses the Term Rewriting System as its underlying semantic model. BlueSpec's approach is to generate a hardware description from a user-created specification of states and transitions. CoreLoom, on the other hand, can be thought of as extracting relevant state information (usage patterns and configurations) from a user-created hardware description. However the Term Rewriting System seems to have good potential as a general-purpose semantic model of hardware for both description and synthesis [8].

Original Configuration	Width	Length	Unique Words
`dp0_flat`	2	2	2
`dp0_loop`	2	4	3
`dp1_flat`	3	4	3
`dp1_sched`	3	7	4
`dotP_dotP`	4	1	1
`dotP_back`	4	3	3
`dotP_stuck`	4	1	1
`reg_dotP_flat`	7	3	3
`reg_dotP_rep`	7	5	5
`reg_dotP_over`	7	5	4
`reg_dotP_nosched`	7	5	5
Average:	4.5	3.6	3

Scheduled Configuration	Length (Change)	Unique Words (Change)
`dp0_flat'`	1 (-1)	1 (-1)
`dp0_loop'`	3 (-1)	2 (-1)
`dp1_flat'`	1 (-3)	1 (-2)
`dp1_sched'`	6 (-1)	3 (-1)
`dotP_dotP'`	1 (0)	1 (0)
`dotP_back'`	4 (+1)	2 (-1)
`dotP_stuck'`	1 (0)	1 (0)
`reg_dotP_flat'`	1 (-2)	1 (-2)
`reg_dotP_rep'`	3 (-2)	3 (-2)
`reg_dotP_over'`	4 (-1)	2 (-2)
`reg_dotP_nosched'`	5 (0)	5 (0)
Average:	2.7 (-0.91)	2 (-1.1)

(Configurations may be found in `Examples.hs`, Appendix)

Table 6.2: Configuration characteristics before and after scheduling.

6.5 Conclusions

Overall, CoreLoom represents an interesting but limited approach to fine-grained flexible datapaths. It has good potential as a tool for design-space exploration, and encourages architects to rethink the traditional boundaries between cores. However, as an embedded language it can be quite verbose, and sometimes not obvious as to the programmer's intent. In the process of making the examples for this report typos were frequent and often resulted in misconnections.

The CoreLoom model is much more suited to implementation as a graphical language. With such an implementation the potential for misconnections would be eliminated, and the development environment itself could check for valid graph constructions rather than performing such a check as part of the semantics. The diagram of a datapath would then become its main source of definition, rather than a convenient way of expressing its structure.

However, even a model with CoreLoom's limitations has exposed research problems for the future, including:

- Expressing all algorithms that a datapath is capable of calculating
- Automatic translation of configurations between datapaths
- A metric of flexibility that can deal with state and interconnects
- Creating an optimal datapath from an arbitrary usage pattern
- Expanding the model to handle non-determinism or asynchronous hardware

As architectures become more and more complex, formal models of their behavior will become necessary. Such models have large potential to aid in the design, verification, and compiler re-targeting of individual architectures and system-on-chips. However, with semantic models there is always a balance between the amount of practical information to include. Characteristics such as bit width and signal representation are important hardware details, yet such details are often irrelevant to higher-level design, and too much detail can reduce usability. In CoreLoom we have opted for one extreme - to ignore such considerations altogether. In the future elegant ways to incorporate them need to be found.

Bibliography

[1] L. Augustsson, J. Schwartz, and R. Nihkil. Bluespec language definition, 2002.

[2] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. In *International Conference on Functional Programming*, pages 174–184, 1998.

[3] W. Dougherty, D. Pursley, and D. Thomas. Instruction subsetting: Trading power for programmability. *IEEE Wshop on VLSI*, 1998.

[4] C. Ebeling, D. C. Cronquist, P. Franklin, J. Secosky, and S. G. Berg. Mapping applications to the RaPiD configurable architecture. In *Field-Programmable Custom Computing Machine*, 1997.

[5] C. Ebeling D. C. Green and P. Franklin. RaPiD – reconfigurable pipelined datapath. In R. W. Hartenstein and M. Glesner, editors, *Field-Programmable Logic: Smart Applications, New Paradigms, and Compilers. 6th International Workshop on Field-Programmable Logic and Applications*, pages 126–135, Darmstadt, Germany, 1996. Springer-Verlag.

[6] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[7] Scott Hauck, Thomas W. Fry, Matthew M. Hosler, and Jeffery P. Kao. The Chimaera reconfigurable functional unit. In Kenneth L. Pocek and Jeffery Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 87–96. IEEE Computer Society Press, 1997.

[8] J.C. Hoe and Arvind. Hardware synthesis from term rewriting systems. *Proceedings of X IFIP International Conference on VLSI, Lisbon, Portugal, December*, 1999.

[9] Paul Hudak. Modular domain specific languages and tools. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

[10] J. Hughes, K. Jeppson, P. Larsson-Edefors, M. Sheeran, P. Stenstrom, and L. Svensson. FlexSoC: combining flexibility and efficiency in SoC designs. *21st Norchip conference, Riga, Latvia, November*, 2003.

[11] S. Kim, C. H. Ziesler, and M. C. Papaefthymiou. Fine-grain real-time reconfigurable pipelining. *IBM J. RES. & DEV.*, 47, 2003.

[12] John Matthews, Byron Cook, and John Launchbury. Microprocessor specification in Hawk. In *International Conference on Computer Languages*, pages 90–101, 1998.

[13] National Institute of Standards and Technology. Federal information processing standards publication 197: Announcing the Advanced Encryption Standard (AES), csrc.nist.gov/publications/fips/fips197/fips-197.pdf, 2001.

[14] Rudolf Usselmann. Advanced Encryption Standard / Rijndael IP Core, www.opencores.org/cgi-bin/cvsget.cgi/aes_core/doc/aes.pdf, 2002.

[15] M. Wirthlin and B. Hutchings. A dynamic instruction set computer. In Peter Athanas and Kenneth L. Pocek, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 99–107, Los Alamitos, CA, 1995. IEEE Computer Society Press.

[16] www.OpenCores.org. Opencores.org: Free open-source IP cores and chip design, 2004.

[17] Lin Zhong and Niraj K. Jha. Interconnect-aware high-level synthesis for low power. *IEEE Int. Conf. on Computer-Aided Design, Nov. 2002*, 2002.

Appendix: Complete Haskell Code

SymbVal.hs

```
module SymbVal where

import List

data SymbFunc =
  SymbFunc String Int --user-defined function, # parameters
  | AND --built-in functions
  | OR
  | XOR
  | INV
  | ADD
  | SUB
  | MUL
  | NEG
  | ID
  | EQUALS
  deriving (Eq)
instance Show SymbFunc
 where
  show (SymbFunc s n) = s
  show AND = "AND"
  show OR = "OR"
  show XOR = "XOR"
  show ID = "ID"
  show INV = "INV"
  show ADD = "ADD"
  show SUB = "SUB"
  show MUL = "MUL"
  show NEG = "NEG"
  show EQUALS = "EQUALS"

data SymbVal =
  FuncApp SymbFunc [SymbVal]
  | VBool Bool
  | VInt Integer
  | Var String
  | VarIx String Int
  | Unknown
```

```
 deriving (Eq)
instance Show SymbVal
 where
  show Unknown = "@"
  show (FuncApp f fs) = (show f) ++ "(" ++ (concat (intersperse ", " (map show fs))) ++ ")"
  show (VBool True) = "1"
  show (VBool False) = "0"
  show (VInt n) = show n
  show (Var x) = x
  show (VarIx x n) = x ++ "_" ++ (show n)

showHTML :: SymbVal -> String
showHTML Unknown = ""
showHTML (FuncApp ADD fs) = "(" ++ (concat (intersperse "+" (map showHTML fs))) ++ ")"
showHTML (FuncApp MUL fs) = "(" ++ concatMap showHTML fs ++ ")"
showHTML (FuncApp s fs) = (show s) ++ "(" ++
         (concat (intersperse ", " (map showHTML fs))) ++ ")"
showHTML (VBool True) = "1"
showHTML (VBool False) = "0"
showHTML (VInt n) = show n
showHTML (Var x) = x
showHTML (VarIx x n) = x ++ "<SUB>" ++ (show n) ++ "</SUB>"

open :: SymbVal
open = Unknown

nParams :: SymbFunc -> Int
nParams (ID) = 1
nParams (INV) = 1
nParams (NEG) = 1
nParams (SymbFunc s n) = n
nParams z = 2

combine :: SymbVal -> SymbVal -> SymbVal
combine Unknown y = y
combine x Unknown = x
combine x y = Unknown

combineM :: SymbVal -> SymbVal -> Maybe SymbVal
combineM Unknown y = Just y
combineM x Unknown = Just x
combineM x y = Nothing

sOR :: SymbVal -> SymbVal -> SymbVal
sOR (VInt n) y = Unknown
sOR x (VInt n) = Unknown
sOR (VBool p) (VBool q) = VBool (p || q)
sOR (VBool True) y = VBool True
sOR x (VBool True) = VBool True
```

```
sOR (VBool False) y = y
sOR x (VBool False) = x
sOR (FuncApp OR xs) (FuncApp OR ys) = FuncApp OR (xs ++ ys)
sOR x (FuncApp OR vs) = FuncApp OR (x:vs)
sOR (FuncApp OR vs) y = FuncApp OR (y:vs)
sOR x y = FuncApp OR [x,y]

sAND :: SymbVal -> SymbVal -> SymbVal
sAND (VInt n) y = Unknown
sAND x (VInt n) = Unknown
sAND (VBool p) (VBool q) = VBool (p && q)
sAND (VBool True) y = y
sAND x (VBool True) = x
sAND (VBool False) y = VBool False
sAND x (VBool False) = VBool False
sAND (FuncApp AND xs) (FuncApp AND ys) = FuncApp AND (xs ++ ys)
sAND x (FuncApp AND vs) = FuncApp AND (x:vs)
sAND (FuncApp AND vs) y = FuncApp AND (y:vs)
sAND x y = FuncApp AND [x,y]

sXOR :: SymbVal -> SymbVal -> SymbVal
sXOR (VInt n) y = Unknown
sXOR x (VInt n) = Unknown
sXOR (VBool True) (VBool False) = VBool True
sXOR (VBool False) (VBool True) = VBool True
sXOR (VBool True) (VBool True) = VBool False
sXOR (VBool False) (VBool False) = VBool False
sXOR (VBool True) y = sEQUALS y (VBool False)
sXOR x (VBool True) = sEQUALS x (VBool False)
sXOR (VBool False) y = sEQUALS y (VBool True)
sXOR x (VBool False) = sEQUALS x (VBool True)
sXOR (FuncApp XOR xs) (FuncApp XOR ys) = FuncApp XOR (xs ++ ys)
sXOR x (FuncApp XOR vs) = FuncApp XOR (x:vs)
sXOR (FuncApp XOR vs) y = FuncApp XOR (y:vs)
sXOR x y = FuncApp XOR [x,y]

sINV :: SymbVal -> SymbVal
sINV (VInt x) = Unknown
sINV (VBool x) = VBool (not x)
sINV (FuncApp INV vs) = FuncApp ID vs
sINV z = FuncApp INV [z]

sID :: SymbVal -> SymbVal
sID x = x

sEQUALS :: SymbVal -> SymbVal-> SymbVal
sEQUALS (VInt p) (VInt q) = VBool (p == q)
sEQUALS (VBool p) (VBool q) = VBool (p == q)
sEQUALS (Var p) (Var q) = VBool (p == q)
```

```
sEQUALS (VarIx p n) (VarIx q m) = VBool ((p == q) && (n == m))
sEQUALS (FuncApp EQUALS xs) (FuncApp EQUALS ys) = FuncApp EQUALS (xs ++ ys)
sEQUALS x (FuncApp EQUALS vs) = FuncApp EQUALS (x:vs)
sEQUALS (FuncApp EQUALS vs) y = FuncApp EQUALS (y:vs)
sEQUALS x y = FuncApp EQUALS [x,y]

sADD :: SymbVal -> SymbVal -> SymbVal
sADD (VBool p) y = Unknown
sADD x (VBool q) = Unknown
sADD x (VInt 0) = x
sADD (VInt 0) y = y
sADD (VInt p) (VInt q) = VInt (p + q)
sADD (FuncApp ADD xs) (FuncApp ADD ys) = FuncApp ADD (xs ++ ys)
sADD x (FuncApp ADD vs) = FuncApp ADD (x:vs)
sADD (FuncApp ADD vs) y = FuncApp ADD (vs ++ [y])
sADD x y = FuncApp ADD [x,y]

sSUB :: SymbVal -> SymbVal -> SymbVal
sSUB (VBool p) y = Unknown
sSUB x (VBool q) = Unknown
sSUB x (VInt 0) = x
sSUB (VInt 0) y = sNEG y
sSUB (VInt p) (VInt q) = VInt (p - q)
sSUB (Var x) (Var y) | x == y = VInt 0
sSUB (VarIx x n) (VarIx y m) | ((x == y) && (m == n)) = VInt 0
sSUB x y = FuncApp SUB [x,y]

sMUL :: SymbVal -> SymbVal -> SymbVal
sMUL (VBool p) y = Unknown
sMUL x (VBool q) = Unknown
sMUL x (VInt 0) = VInt 0
sMUL (VInt 0) y = VInt 0
sMUL x (VInt 1) = x
sMUL (VInt 1) y = y
sMUL (VInt p) (VInt q) = VInt (p * q)
sMUL (FuncApp MUL xs) (FuncApp MUL ys) = FuncApp MUL (xs ++ ys)
sMUL x (FuncApp MUL vs) = FuncApp MUL (x:vs)
sMUL (FuncApp MUL vs) y = FuncApp MUL (vs ++ [y])
sMUL x y = FuncApp MUL [x,y]

sNEG :: SymbVal -> SymbVal
sNEG (VBool x) = Unknown
sNEG (VInt x) = VInt (negate x)
sNEG (FuncApp NEG vs) = FuncApp ID vs
sNEG z = FuncApp NEG [z]

lift1 :: (SymbVal -> SymbVal) -> SymbVal -> SymbVal
lift1 f Unknown = Unknown
lift1 f x = f x
```

```
lift2 :: (SymbVal -> SymbVal -> SymbVal) -> SymbVal -> SymbVal -> SymbVal
lift2 f Unknown y = Unknown
lift2 f x Unknown = Unknown
lift2 f x y = f x y

flatten :: SymbVal -> SymbVal
flatten (FuncApp f vs) = symbApply f (map flatten vs)
flatten z = z

symbApply :: SymbFunc -> [SymbVal] -> SymbVal
symbApply f vals =
  if elem Unknown vals
    then Unknown
    else if elem open vals
      then open
      else app f vals
 where
  app OR vals = foldl (lift2 sOR) (VBool False) vals
  app AND vals = foldl (lift2 sAND) (VBool True) vals
  app XOR (v:vals) = foldl (lift2 sXOR) v vals
  app INV vals = (lift1 sINV) (head vals)
  app ID vals = (lift1 sID) (head vals)
  app EQUALS (v:vals) = foldl (lift2 sEQUALS) v vals
  app ADD vals = foldl (lift2 sADD) (VInt 0) vals
  app SUB (v:vals) = foldl (lift2 sSUB) v vals
  app MUL vals = foldl (lift2 sMUL) (VInt 1) vals
  app NEG vals = (lift1 sNEG) (head vals)
  app f vals = FuncApp f vals

equalNoIx :: SymbVal -> SymbVal -> Bool
equalNoIx (VarIx x n) (VarIx y m) = x == y
equalNoIx x y = x == y

getIxs :: SymbVal -> [(String, Int)]
getIxs (VarIx x n) = [(x,n)]
getIxs (FuncApp f vs) = concat (mapM getIxs vs)
getIxs z = []

groupIxs :: [(String, Int)] -> [(String, [Int])]
groupIxs [] = []
groupIxs ((s,n):ns) = (s, nub (n : (map snd (filter (fstEq s) ns)))) : (groupIxs ns')
 where
  ns' = filter (fstNeq s) ns
  fstEq x (y,s) = x == y
  fstNeq x (y,s) = x /= y
```

Datapath.hs

```
module Datapath where

import List
import STATE
import ErrM
import SymbVal

type Datapath = ([Node], [Connection])

type NodeName = String

data Node =
 Function
 {
   nodename :: NodeName,
   func :: SymbFunc
 }
 | JoinPoint
 {
   nodename :: NodeName
 }
 | Input
 {
   nodename :: NodeName
 }
 | Output
 {
   nodename :: NodeName
 }
 | Delay
 {
   nodename :: NodeName,
   value :: SymbVal
 }
instance Eq Node where
 n == m = (nodename n) == (nodename m)
instance Show Node where
 show (Delay s v) = "|" ++ s ++ "|" ++ (show v) ++ ">"
 show (Output s) = "//" ++ s
 show (Input s) = s ++ "\\\\"
 show (JoinPoint s) = s ++ "}"
 show (Function s f) =  "{" ++ s ++ ":" ++ (show f) ++ "}"

type InputPort = (NodeName, Int)

type Connection = (NodeName, InputPort)
     --           (from,(to, port#))
```

```
type Result = STATE Datapath

--Node and Edge creation functions
addEdge :: Connection -> Result ()
addEdge e@(f, (t,n)) =
 do
   dp@(ns, es) <- readSTATE
   if elem e es
     then fail "Tried to add a duplicate connection!"
     else do
       to <- lookupNodeM t
       from <- lookupNodeM f
       case (from,to) of --for modular DPs
         (Output _, Input _) -> do --replace with a direct connection
           let prv = getPrevNode_n dp (f, 0)
           let nexts = getNextNodes_n dp t
           remNode t
           remNode f
           let zips = zip (replicate (length nexts) prv) nexts
           mapM connect_n zips
           return ()
         (_, Input _) -> do
           let nexts = getNextCons dp t
           mapM (reanchor f) nexts
           remNode t
         (Output _, _) -> do
           let prv = getPrevCon_n dp (f, 0)
           repoint (t,n) prv
           remNode f
         z -> writeSTATE (ns, e:es) --normal add

addEdges es =
 do
   mapM addEdge es
   return ()

connect :: (NodeName, NodeName) -> Result ()
connect (f, t) =
 do
   t' <- lookupNodeM t
   case t' of
     (Function _ func) -> if nParams func > 1
        then fail "Use connect_n to connect to multi-parameter functions."
        else addEdge (f, (t, 0))
     z -> addEdge (f, (t, 0))

connect_n :: (NodeName, InputPort) -> Result ()
```

```
connect_n (f, t) =
 do
   addEdge (f, t)

connections :: [(NodeName, NodeName)] -> Result ()
connections zs =
 do
   mapM connect zs
   return ()

connections_n :: [(NodeName,InputPort)] -> Result ()
connections_n zs =
 do
   mapM connect_n zs
   return ()

--connects a node to all elements of the list
connectTo :: NodeName -> [NodeName] -> Result ()
connectTo from tos =
 do
   let zips = zip (replicate (length tos) from) tos
   connections zips

connectTo_n :: NodeName -> [InputPort] -> Result ()
connectTo_n from tos =
 do
   let zips = zip (replicate (length tos) from) tos
   connections_n zips

addNode :: Node -> Result ()
addNode n =
 do
   (ns, es) <- readSTATE
   if elem n ns
     then fail ("Non-unique node name: " ++ (nodename n))
     else writeSTATE (n:ns, es)

addNodes ns =
 do
   mapM addNode ns
   return ()

input :: NodeName -> Result ()
input = addNode . Input

inputs :: [NodeName] -> Result ()
inputs ns =
 do
    let is = map Input ns
```

```
    addNodes is

output :: NodeName -> Result ()
output = addNode . Output

outputs :: [NodeName] -> Result ()
outputs ns =
 do
    let is = map Output ns
    addNodes is

delay :: SymbVal -> NodeName -> Result ()
delay v n = addNode (Delay n v)
delays :: SymbVal -> [NodeName] -> Result ()
delays v ns =
 do
   let is = map (flip Delay v) ns
   addNodes is

--N delays with the same value, linked in a line
delayN :: Int -> SymbVal -> NodeName -> Result ()
delayN 0 _ _ = return ()
delayN 1 v nm =
 do
   delay v (nm ++ "_0")
delayN n v nm =
 do
   let ns = map show (range (0, n - 1))
   let nms = map ((nm ++ "_") ++) ns
   delays v nms
   linkUp (reverse nms)

--Creates a linked line of delays with the given initial values
delayList :: [SymbVal] -> NodeName -> Result ()
delayList [] nm = return ()
delayList [v] nm =
 do
   delay v (nm ++ "_0")
delayList vs nm =
 do
   let ns = map show (range (0, (length vs) - 1))
   let nms = map ((nm++"_")++) ns
   let zips = zip vs nms
   mapM (uncurry delay) zips
   linkUp (reverse nms)

function :: SymbFunc -> NodeName -> Result ()
function f n = addNode (Function n f)
```

```
functions :: SymbFunc -> [NodeName] -> Result ()
functions f ns =
 do
   let is = map (flip Function f) ns
   addNodes is

joinPoint :: NodeName -> Result ()
joinPoint n = addNode (JoinPoint n)

joinPoints :: [NodeName] -> Result ()
joinPoints ns =
 do
   let is = map (JoinPoint) ns
   addNodes is

remEdge :: Connection -> Result ()
remEdge e =
 do
   (ns, es) <- readSTATE
   writeSTATE (ns, es \\ [e])

remEdges es =
 do
   mapM remEdge es
   return ()

disconnect :: NodeName -> NodeName -> Result ()
disconnect f t =
 do
   t' <- lookupNodeM t
   case t' of
     (Function _ func) -> if nParams func > 1
        then fail "Use disconnect_n to disconnect from multi-parameter functions."
        else remEdge (f, (t, 0))
     z -> remEdge (f, (t, 0))

disconnect_n :: NodeName -> InputPort -> Result ()
disconnect_n f t =
 do
   remEdge (f, t)

remNode :: NodeName -> Result ()
remNode node =
 do
   n <- lookupNodeM node
   dp <- readSTATE
   let cs = getEdges dp (nodename n)
   remEdges cs
   (ns, es) <- readSTATE
```

```
   writeSTATE (ns \\ [n], es)

remNodes ns =
 do
   mapM remNode ns
   return ()

--Non-monadic data-gathering helper functions

getEdges :: Datapath -> NodeName -> [Connection]
getEdges dp n = next ++ prvs
 where
   prvs = getPrevCons dp n
   next = getNextCons dp n

getNextCons :: Datapath -> NodeName -> [Connection]
getNextCons (_, es) n = nexts
 where
  nexts = filter (isPrev n) es
getNextNodes :: Datapath -> NodeName -> [NodeName]
getNextNodes dp n = map (fst . snd) cs
 where
   cs = getNextCons dp n

getNextNodes_n :: Datapath -> NodeName -> [InputPort]
getNextNodes_n dp n = map snd (getNextCons dp n)

getPrevCons :: Datapath -> NodeName -> [Connection]
getPrevCons dp@(ns, es) node =
 case lookupNode dp node of
    (Function f func) -> concatMap (getPrev' f es) (range (0, (nParams func) - 1))
    (JoinPoint f) -> concatMap (getPrev' f es) [0, 1]
    n -> filter (isNext (node,0)) es
  where
   getPrev' f es n = filter (isNext (f,n)) es

getPrevNodes :: Datapath -> NodeName -> [NodeName]
getPrevNodes dp node = map fst cs
 where
  cs = getPrevCons dp node

getPrevCon_n :: Datapath -> InputPort -> Connection
getPrevCon_n dp@(ns, es) (node, n) =
 case prvs of
    [] -> error ("No previous node for node " ++ node ++ ", input " ++ (show n))
    (x:[]) -> x
    xs -> error ("Somehow there is more than one node connected to "
                 ++ node ++ ", input " ++ (show n))
  where
```

```
   prvs = filter (isNext (node,n)) es

getPrevCons_n :: Datapath -> InputPort -> [Connection]
getPrevCons_n (_, es) (node, n) = filter (isNext (node,n)) es

getPrevNode_n :: Datapath -> InputPort -> NodeName
getPrevNode_n dp p = fst (getPrevCon_n dp p)

getPrevNodes_n :: Datapath -> InputPort -> [NodeName]
getPrevNodes_n (_, es) (node, n) = map fst prvs
  where
    prvs = filter (isNext (node,n)) es

lookupNode :: Datapath -> NodeName -> Node
lookupNode (ns, _) s =
   case find (\x -> nodename x == s) ns of
     Just x -> x
     Nothing -> error ("Unable to find a node named " ++ s)

lookupNodeM :: NodeName -> Result Node
lookupNodeM s =
 do
   (ns, _) <- readSTATE
   case find (\x -> nodename x == s) ns of
     Just x -> return x
     Nothing -> fail ("Unable to find a node named " ++ s)


isNext :: InputPort -> Connection -> Bool
isNext (s,k) (f,(t,n)) = t == s && k == n

isPrev :: NodeName -> Connection -> Bool
isPrev s (f,(t,n)) = f == s

isConnected :: Datapath -> InputPort -> Bool
isConnected (_, es) c = prvs /= []
 where
   prvs = filter (isNext c) es

isDelay :: Node -> Bool
isDelay (Delay _ _) = True
isDelay _ = False

isInput :: Node -> Bool
isInput (Input _) = True
isInput _ = False

isOutput :: Node -> Bool
isOutput (Output _) = True
```

```
isOutput _ = False

isFunction :: Node -> Bool
isFunction (Function _ _) = True
isFunction _ = False

isJoinPoint :: Node -> Bool
isJoinPoint (JoinPoint _) = True
isJoinPoint _ = False

getNodes :: Datapath -> [NodeName]
getNodes (ns, _) = map nodename ns

getDelays :: Datapath -> [Node]
getDelays (ns, _) = filter isDelay ns

getInputs :: Datapath -> [Node]
getInputs (ns, _) = filter isInput ns

getOutputs :: Datapath -> [Node]
getOutputs (ns, _) = filter isOutput ns

getFunctions :: Datapath -> [Node]
getFunctions (ns, _) = filter isOutput ns

getJoinPoints :: Datapath -> [Node]
getJoinPoints (ns, _) = filter isJoinPoint ns

getInitialState :: Datapath -> [(NodeName, SymbVal)]
getInitialState dp = map getIV (getDelays dp)
 where
  getIV (Delay n v) = (n, v)

--Returns all non-JP nodes with more than one incoming connection per port
--Used to add implicit JPs
getMultipleCons :: Datapath -> [InputPort]
getMultipleCons (ns, es) = concatMap hasMCs ns
 where
  hasMCs (JoinPoint s) = []
  hasMCs (Function s f) = concatMap checkPrev (zip (replicate k s) (range (0, k - 1)))
   where
    k = nParams f
  hasMCs n = checkPrev (nodename n, 0)
  checkPrev p = if length (getPrevCons_n (ns, es) p) > 1
              then [p]
              else []

--Datapath altering helper functions
```

```
--alters the destination of a connection
repoint :: InputPort -> Connection -> Result ()
repoint to e@(from, t) =
 do
   (ns,es) <- readSTATE
   let es' = es \\ [e]
   writeSTATE (ns, (from, to):es')

--alters the origin of a connection
reanchor :: NodeName-> Connection -> Result ()
reanchor from e@(f, to) =
 do
   (ns,es) <- readSTATE
   let es' = es \\ [e]
   writeSTATE (ns, (from, to):es')

--creates Delay D and JoinPoint D_load, linked
register :: SymbVal -> NodeName -> Result ()
register v r =
 do
   let load = r ++ "_load"
   addNodes [JoinPoint load, Delay r v]
   addEdges [(load,(r,0)), (r, (load, 0))]

--adds a register before the specified port
--incoming connections are re-pointed
registerBefore :: SymbVal -> NodeName -> InputPort -> Result ()
registerBefore v r (to, n) =
 do
   let load = r ++ "_load"
   dp <- readSTATE
   let cs = getPrevCons_n dp (to, n)
   addNodes [JoinPoint load, Delay r v]
   addEdges [(load,(r,0)), (r, (load, 0)), (r, (to, n))]
   mapM_ (repoint (load, 0)) cs

--adds a register after the specified node
--outgoing connections are re-anchored
registerAfter :: SymbVal -> NodeName -> NodeName -> Result ()
registerAfter v r from =
 do
   let load = r ++ "_load"
   dp <- readSTATE
   let cs = getNextCons dp from
   addNodes [JoinPoint load, Delay r v]
   addEdges [(load,(r,0)), (r, (load, 0)), (from, (r, 0))]
   mapM_ (reanchor r) cs

--Links up the nodes in the list
```

```
linkUp :: [NodeName] -> Result ()
linkUp [] = return ()
linkUp [x] = return ()
linkUp (x:y:xs) =
 do
   addEdge (x,(y,0))
   linkUp (y:xs)

linkUp_n :: [InputPort] -> Result ()
linkUp_n [] = return ()
linkUp_n [x] = return ()
linkUp_n ((x,_):(y,n):xs) =
 do
   addEdge (x,(y,n))
   linkUp_n ((y,n):xs)

--Creates a Delay linked to itself
constant :: SymbVal -> NodeName -> Result ()
constant v n =
 do
   addNode (Delay n v)
   addEdge (n, (n, 0))

--Creates a list of delays linked to the beginning
constantList :: [SymbVal] -> NodeName -> Result ()
constantList [] _ = return ()
constantList [x] s = constant x (s ++ "_0")
constantList vs nm =
 do
   let nm' = nm ++ "_"
   let k = (length vs) - 1
   let ns = map show (range (0,k))
   let nms = map (nm'++) ns
   let zips = zip nms vs
   let ds = map (uncurry Delay) zips
   addNodes ds
   linkUp (reverse nms)
   addEdge (nm' ++ "0", (nm' ++ (show k), 0))

--Creates two lists, linked, the second of which repeats
oneTimeList :: ([SymbVal], [SymbVal]) -> NodeName -> Result ()
oneTimeList (is, fs) nm =
 do
   constantList fs (nm ++ "_init")
   delayList is nm
   connect ((nm ++ "_init_0"), (nm ++ "_" ++ (show ((length is) - 1))))

--renames nodes, edges for modular datapaths
prefix :: String -> String -> String
```

```
prefix s n = s ++ ":" ++ n

nodePrefix :: String -> Node -> Node
nodePrefix s (Delay nm (Var v)) = (Delay (prefix s nm) (Var (prefix s v)))
nodePrefix s (Delay nm (VarIx v x)) = (Delay (prefix s nm) (VarIx (prefix s v) x))
nodePrefix s n = n {nodename = (prefix s (nodename n))}

edgePrefix :: String -> Connection -> Connection
edgePrefix s (f, (t, n)) = (prefix s f, (prefix s t, n))

--Embeds a previously defined DP
addDP :: Datapath -> String -> Result ()
addDP (ns,es) nm =
 do --rename all units, fix connections, add.
   let ns' = map (nodePrefix nm) ns
   let es' = map (edgePrefix nm) es
   mapM addNode ns'
   mapM addEdge es'
   return ()

--Adds one JP and repoints incoming connections to it.
addJoinPoint :: InputPort -> Result ()
addJoinPoint (to, n) =
 do
   dp <- readSTATE
   let prvs = getPrevCons_n dp (to, n)
   let s = "into_(" ++ to ++ "," ++ (show n) ++ ")"
   addNode (JoinPoint s)
   mapM (repoint (s, 0)) prvs
   connect_n (s, (to, n))
   return ()

--Adds implicit JPs
addJoinPoints :: Result ()
addJoinPoints =
 do
   dp <- readSTATE
   let ns = getMultipleCons dp
   mapM addJoinPoint ns
   return ()

makeDatapath dp = case appSTATE (dp >> checkDatapath) ([],[]) of Ok (x, y) -> x

checkDatapath :: Result Datapath
checkDatapath =
 do
   addJoinPoints
   checkOutputs
   checkCons
```

```
   checkGraph
   dp <- readSTATE
   return dp

checkOutputs :: Result ()
checkOutputs =
 do
  dp <- readSTATE
  case getOutputs dp of
   [] -> error "Datapath must include at least one Output node!"
   _ -> return ()

--All ports must be connected, except JP control ports
--More than one edge may originate from a output port,
--but multiple edges may not be connected to one input port.
checkCons :: Result ()
checkCons =
 do
   (ns, es) <- readSTATE
   mapM_ checkCon ns
checkCon :: Node -> Result ()
checkCon (Input nm) =
  do
   dp <- readSTATE
   case getNextCons dp nm of
    [] -> error ("No outgoing connection for Input " ++ nm ++ "!")
    _ -> return ()
checkCon (Output nm) =
  do
   dp <- readSTATE
   case getPrevCon_n dp (nm, 0) of --this function does the input checking
    c -> return ()
checkCon (Delay nm val) =
  do
   dp <- readSTATE
   case getPrevCon_n dp (nm, 0) of
    c -> case getNextCons dp nm of
     [] -> error ("No outgoing connection for Delay " ++ nm ++ "!")
     _ -> return ()
checkCon (JoinPoint nm) =
  do
   dp <- readSTATE
   case getPrevCons_n dp (nm, 0) of
    [] -> error ("No incoming connections to Join Point " ++ nm ++ " input port 0!")
    c -> case getNextCons dp nm of
     [] -> error ("No outgoing connection for Join Point " ++ nm ++ "!")
     _ -> return ()
checkCon (Function nm f) =
  do
```

```
   dp <- readSTATE
   let k = nParams f
   let zips = zip (replicate k nm) (range (0, k - 1))
   mapM (checkFun dp) zips
   case getNextCons dp nm of
     [] -> error ("No outgoing connection for Function " ++ nm ++ "!")
     _ -> return ()
  where
   checkFun dp port = case getPrevCon_n dp port of
                        c -> return ()

--check that the datapath is weakly connected,
--IE no disjoint components. Also checks for combinational loops.
--A straightforward fixed-point iteration from output nodes
checkGraph :: Result ()
checkGraph =
 do
   dp <- readSTATE
   let outs = map nodename (getOutputs dp) --guaranteed non-empty
   coverage <- expand outs
   if coverage `setEquals` (getNodes dp)
     then return ()
     else error ("Disjoint component found including nodes: " ++
                (unwords coverage))

expand :: [NodeName] -> Result [NodeName]
expand ns =
 do
  dp <- readSTATE
  let prvs = concatMap (getPrevNodes dp) ns
  let is = intersect prvs ns
  mapM checkLoops is
  let nexts = union ns prvs
  if nexts `setEquals` ns
    then return nexts
    else expand nexts

--Checks for a possible combinational loop.
checkLoops :: NodeName -> Result ()
checkLoops n =
 do
   dp <- readSTATE
   node <- lookupNodeM n
   case node of
    (Delay _ _) -> return ()
    _ -> do
      let prvs = getPrevNodes dp n
      mapM (checkPath n []) prvs
      return ()
```

```haskell
--All loop paths must include at least one Delay node
checkPath :: NodeName -> [NodeName] -> NodeName -> Result ()
checkPath orig visited n =
 do
  dp <- readSTATE
  node <- lookupNodeM n
  case node of
    (Delay _ _) -> return ()
    _ -> if elem n visited
      then return ()
      else do
        if orig == n
          then error ("Combinational Loop found from node " ++ orig ++ " to " ++ n ++ "!")
          else do
            let prvs = getPrevNodes dp n
            mapM_ (checkPath orig (n:visited)) prvs


setEquals :: Ord a => [a] -> [a] -> Bool
setEquals x y = sort x == sort y
```

Configuration.hs

```haskell
module Configuration where

import SymbVal
import Datapath
import List
import Monad

type Pair = (NodeName, SymbVal)
type Row = [SymbVal]

type Column = (NodeName, [SymbVal])

type Table = [Row]

type Config = ([NodeName], Table)

type ConfigInfo = [(NodeName, [NodeName])]

flat :: Config
flat = ([], [])

--Table Functions
```

```
--attempts to combine two tables without increasing their length.
--Uses the open value
combineTables :: Table -> Table -> Maybe Table
combineTables c1s c2s =
 do
   let r1 = length (head c1s)
   let r2 = length (head c2s)
   when (r1 /= r2) Nothing
   let l1 = length c1s
   let l2 = length c2s
   let c1s' = if l2 > l1 then c1s ++ (blankTable (l2 - l1, r1)) else c1s
   let c2s' = if l1 > l2 then c2s ++ (blankTable (l1 - l2, r2)) else c2s
   let zips = zip c1s' c2s'
   rs <- mapM (uncurry combineRows) zips
   return rs

combineRows :: Row -> Row -> Maybe Row
combineRows xs ys =
 do
   let zips = zip xs ys
   r <- mapM (uncurry combineM) zips
   return r

--Combines tables by pushing one down until it fits
addTables :: Table -> Table -> Table
addTables x y = case combineTables x y of
   Just z -> z
   Nothing -> addTables x (pushDown y)

--Adds tables and returns how many rows it was pushed down
addTablesN :: Table -> Table -> (Table, Int)
addTablesN x y = addTablesN' x y 0
 where
  addTablesN' x y n = case combineTables x y of
   Just z -> (z, n)
   Nothing -> addTablesN' x (pushDown y) (n+1)

--Repeats a table a number of times equal to its length
--and returns how many rows larger the result is.
{- IE  1                 1
        2          ==>   12
         3               123   & 3
                         123
                          23
                           3
-}
repTableL :: Table -> (Table, Int)
repTableL x = repTableL' x x (length x)
```

```
 where
  repTableL' x y 1 = addTablesN x y
  repTableL' x y n = repTableL' (addTables x y) y (n-1)
   where

--Performs concatenation and overlapping scheduling
scheduleTable :: Table -> Table
scheduleTable x = scheduleTable' x x
 where
  scheduleTable' x y = if n < ((length y) - 2)
                        then scheduleTable' x' y
                        else x
   where
    (x', n) = addTablesN x y

--Schedules a table and then compresses the columns.
{-   Original     Repeated  Scheduled  Compressed
        1           1
         2    ==>   12   ==>  123   ==> 123
        1 3         123       143        4
         4          143
                     4

-}
optimize :: Config -> [Column]
optimize (ns, t) = final
 where
  sched = scheduleTable t
  (t', n1) = repTableL sched
  (t'',n2) = addTablesN t' sched
  mid = drop n1 (take n2 t'')
  final = map compressCol (configToCols (ns, mid))


--Compresses columns
compressCol :: Column -> Column
compressCol (n, (c:cs)) = case filter (c /=) cs of
                            [] -> (n,[c]) --search for longer patterns in future
                            z -> (n, (c:cs))

--Creates a blank table of the given dimensons
blankTable :: (Int, Int) -> Table
blankTable (x,y) = replicate x (blankRow y)

blankRow :: Int -> Row
blankRow y = replicate y Unknown

pushDown :: Table -> Table
pushDown [] = []
```

```
pushDown (x:xs) = (blankRow (length x)):x:xs

tableWidth :: Table -> Int
tableWidth [] = 0
tableWidth (x:xs) = length x

dupTable :: Table -> Table
dupTable t = addTables t t

repeatTable :: Int -> Table -> Table
repeatTable 0 t = []
repeatTable 1 t = t
repeatTable k t = addTables (repeatTable (k-1) t) t

--Configuration Functions
configToCols :: Config -> [Column]
configToCols (ns, rs) = zip ns (transpose rs)
colsToConfig :: [Column] -> Config
colsToConfig cs = (headings, rows)
 where
   headings = map fst cs
   rows = transpose (map snd cs)

pairsToCols :: [[Pair]] -> [Column]
pairsToCols [] = []
pairsToCols pss = map (makeCol pss) (getVarRows pss [])
 where
  getVarRows [] vs = vs
  getVarRows (ps:pss) vs = getVarRows pss (getVars ps vs)
  getVars [] vs = vs
  getVars ((p,v):ps) vs = if elem p vs
                    then getVars ps vs
                    else getVars ps (p:vs)
  makeCol ps v = (v, map makeCol' (map (lookup v) ps))
  makeCol' (Just x) = x
  makeCol' Nothing = Unknown

concatConfig :: Config -> Config -> Config
concatConfig (ns1, bs1) (ns2, bs2) = (ns1 ++ ns2, bs')
 where
  bs' = map (uncurry (++)) zips
  zips = zip bs1 bs2

concatConfigs :: [Config] -> Config
concatConfigs [] = ([],[])
concatConfigs [x] = x
concatConfigs (x:xs) = foldl concatConfig x xs

--gives the increase in size from adding two tables
```

```
sizeIncrease :: Table -> Table -> Int
sizeIncrease t1 t2 = snd (addTablesN t1 t2)

prefixVal :: String -> SymbVal -> SymbVal
prefixVal p (Var s) = Var (p ++ ":" ++ s)
prefixVal p (VarIx s n) = VarIx (p ++ ":" ++ s) n
prefixVal p z = z

subConfig :: Config -> NodeName -> Config
subConfig (ns, bs) m = (ns', bs')
 where
  ns' = map (\x -> m ++ ":" ++ x) ns
  bs' = map (map (prefixVal m)) bs

subConfigs cf ms = concatConfigs cfs
 where
  cfs = map (subConfig cf) ms

--Returns all exposed join points and their associated choices
configInfo :: Datapath -> ConfigInfo
configInfo (ns, es) = zips
 where
  jps = map nodename (filter isExposedJoinPoint ns)
  prvs = map getPrevs jps
  getPrevs jp = map fst (filter (isNext (jp, 0)) es)
  zips = zip jps prvs
  isExposedJoinPoint n = (isJoinPoint n) && ((filter (isNext (nodename n, 1)) es) == [])

--Returns all exposed join points
configChoices :: Datapath -> [NodeName]
configChoices dp = map fst (configInfo dp)

--Creates a blank config of length n for a given dp
blankConfig :: Datapath -> Int -> Config
blankConfig (ns, es) n = (jps, blank)
 where
  jps = map nodename (filter isJoinPoint ns)
  blank = blankTable (n,length jps)

-- IO Functions

displayCols :: [Column] -> IO ()
displayCols cs = displayConfig (colsToConfig cs)
displayConfig :: Config -> IO ()
displayConfig (headings, bt) =
 do
   let h = concat (intersperse "\t" headings)
   putStrLn h
   putStrLn (replicate 80 '=')
```

```
    mapM printRow bt
    return ()
  where
   printRow r = putStrLn $ concat (intersperse "\t" (map show r))

displayConfigN :: Int -> Config -> IO ()
displayConfigN n (headings, bt) =
 do
   let h = concat (intersperse "\t" headings)
   putStr "CC\t|\t"
   putStrLn h
   putStr "--------+"
   putStrLn (replicate (8 * ((length headings)+1)) '-')
   printRows bt 0 n
   return ()
 where
  printRows [] _ _ = return ()
  printRows _ _ 0 = return ()
  printRows (r:rs) c n =
   do
    putStr (show c)
    putStr "\t|\t"
    putStrLn $ concat (intersperse "\t" (map show r))
    printRows rs (c+1) (n-1)
```

UsagePattern.hs

```
module UsagePattern where

import SymbVal
import Datapath
import Configuration
import STATE
import List

type UsagePattern = ([[Pair]], [[Pair]])

type Latency = Int
type Throughput = (Int, Int)

type UsageEnv = [(SymbVal, NodeName)]

--Creates a pipelined version of a function
--with given latency
funcPipelined :: NodeName -> SymbFunc -> Latency -> Result ()
funcPipelined nm f 0 =
```

```
 do
   function f nm
funcPipelined nm f l =
 do
   let nm' = nm ++ "_d"
   function f nm
   delayN l open nm'
   addEdge (nm, (nm' ++ "_" ++ (show (l - 1)), 0))

--Creates an unpipelined version of a function
--with given latency and TP <= 1
funcUnpipelined :: NodeName -> SymbFunc -> Latency -> Throughput -> Result ()
funcUnpipelined nm f l (n, d) =
 do
   let nm' = nm ++ "_d"
   function f nm
   delayN l open nm'
   addEdge (nm, (nm' ++ "_" ++ (show (l - 1)), 0))
   let last = nm' ++ "0"
   addEdge (last, (last,0))
   let v1 = replicate n (Var (nm' ++ "1"))
   let v2 = replicate (d - n) (Var last)
   let con = nm ++ "_control"
   constantList (v1 ++ v2) con
   addEdge (con ++ "0", ("FOO", 0))

--An attempt to generate a datapath from an arbitrary usage pattern
bbUsagePattern :: UsagePattern -> Datapath
bbUsagePattern (ins, outs) = makeDatapath res
 where
  outcols = pairsToCols outs
  incols = pairsToCols ins
  res =  buildDP (incols, outcols)

buildDP :: ([Column], [Column]) -> Result ()
buildDP (incols, outcols) =
 do
  let env = concatMap buildEnv incols
  let ins = (map fst incols)
  mapM_ input ins
  mapM_ inputGuard incols
  mapM_ (addPaths env) outcols
  mapM_ outputGuard outcols

buildEnv :: Column -> UsageEnv
buildEnv (node, vs) = map (\v -> (v, node)) vs
addPaths :: UsageEnv -> Column -> Result ()
addPaths env (node, vs) =
 do
```

```
  output node
  let zips = zip vs (range (0, length vs))
  mapM_ (addPath env (node, 0)) zips


addPath :: UsageEnv -> InputPort -> (SymbVal, Int) -> Result ()
addPath env (node, p) (Unknown, _) = return ()
addPath env (node, p) (FuncApp f vs, n) =
 do
   let func = (node ++ "_" ++ (show n))
   function f func
   connect_n (func, (node, p))
   let k = nParams f
   let zs = zip (replicate k func) (range (0, k))
   let zs' = zip vs (replicate k n)
   let zips = zip zs zs'
   mapM_ (uncurry (addPath env)) zips
addPath env (node, p) (v@(VarIx var ix), n) = --constant or input
 do
   case lookup v env of
     Nothing -> do --constant value
       let c = (node ++ "_" ++ (show n))
       constant v c
       connect_n (c, (node, p))
     Just x -> do
       if n - ix > 0 --latency test
         then do
           let latname = node ++ "_" ++ var ++ "_" ++ (show ix) ++ "_lat_" ++ (show n)
           delayN (n - ix) open latname
           connect (x ++ "_guard", latname ++ "_" ++ (show (n - ix - 1)))
           connect_n (latname ++ "_0", (node, p))
         else do
           connect_n (x ++ "_guard", (node, p))

inputGuard (i, vs) =
 do
   let vs' = map (\x -> if x == Unknown then Unknown else Var i) vs
   joinPoint (i ++ "_guard")
   constantList vs' (i ++ "_glist")
   connect_n (i ++ "_glist_0", (i ++ "_guard",1))
   connect (i, i ++ "_guard")

outputGuard (o, vs) =
 do
   let vs' = makeVals vs 0
   joinPoint (o ++ "_guard")
   constantList vs' (o ++ "_glist")
   connect_n  (o ++ "_glist_0", (o ++ "_guard", 1))
   dp <- readSTATE
```

```
   let prvs = getPrevCons dp o
   mapM_ (repoint (o ++ "_guard", 0)) prvs
   connect (o ++ "_guard", o)
  where
   makeVals [] _ = []
   makeVals (Unknown:vs) n = Unknown : (makeVals vs (n+1))
   makeVals (v:vs) n = (Var (o ++ "_" ++ (show n))) : (makeVals vs (n+1))

--IO Functions

displayUP_n :: Int -> UsagePattern -> IO ()
displayUP_n n (ins, outs) =
 do
   let icols = pairsToCols ins
   let ocols = pairsToCols outs
   let (ihs, idata) = colsToConfig icols
   let (ohs, odata) = colsToConfig ocols
   let headings = ihs ++ ohs
   let bt =  transpose ((transpose idata) ++ (transpose odata))
   let h = concat (intersperse "\t" headings)
   putStr "CC\t|\t"
   putStrLn h
   putStr "--------+"
   putStrLn (replicate (8 * ((length headings)+1)) '-')
   printRows bt 0 n
   return ()
 where
  printRows [] _ _ = return ()
  printRows _ _ 0 = return ()
  printRows (r:rs) c n =
   do
    putStr (show c)
    putStr "\t|\t"
    putStrLn $ concat (intersperse "\t" (map show r))
    printRows rs (c+1) (n-1)

upToHTML_n :: Int -> UsagePattern -> String
upToHTML_n n (ins, outs) = header ++ (pRows bt 0 n) ++ footer
 where
   icols = pairsToCols ins
   ocols = pairsToCols outs
   (ihs, idata) = colsToConfig icols
   (ohs, odata) = colsToConfig ocols
   headings = ihs ++ ohs
   bt = transpose ((transpose idata) ++ (transpose odata))
   h = concat (intersperse "</TD><TD>" headings)
   header = "<HTML>\n<BODY>\n<TABLE BORDER = \"1\">\n<TR><TD>CC</TD><TD>"
            ++ h ++ "</TD></TR>\n"
   footer = "</TABLE>\n</BODY>\n</HTML>"
```

```
   pRows [] _ _ = []
   pRows _ _ 0 = []
   pRows (r:rs) c n = row ++ (pRows rs (c+1) (n-1))
    where
     row = "<TR><TD>" ++ (show c) ++ ("</TD><TD>") ++ datum ++ "</TD></TR>\n"
     datum = concat (intersperse "</TD><TD>" (map showHTML r))
```

Run.hs

```
module Run where

import Datapath
import Configuration
import SymbVal
import UsagePattern
import STATE
import ErrM
import List

type DelayState = [Pair]

--State during simulation
data DPState = DPState
 {
   datapath    :: Datapath,
   last_cc     :: Int,
   cur_cc      :: Int,
   cur_st      :: DelayState,
   next_st     :: DelayState,
   next_up      :: ([Pair], [Pair]),
   up          :: UsagePattern,
   st          :: [DelayState]
 } deriving (Show, Eq)

type RunState = STATE DPState

currentCC :: RunState Int
currentCC =
 do
   st <- readSTATE
   return (cur_cc st)

currentDP :: RunState Datapath
currentDP =
 do
   st <- readSTATE
   return (datapath st)
```

```
prevNode :: InputPort -> RunState Node
prevNode p =
 do
   dp <- currentDP
   return (lookupNode dp (getPrevNode_n dp p))

addUseIn :: Pair -> RunState ()
addUseIn p =
 do
   st <- readSTATE
   let (ins, outs) = next_up st
   if elem p ins
    then return ()
    else do
      writeSTATE st {next_up = (p : ins, outs)}
      return ()

addUseOut :: Pair -> RunState ()
addUseOut p =
 do
   st <- readSTATE
   let (ins, outs) = next_up st
   writeSTATE st {next_up = (ins, p : outs)}
   return ()

nextVal :: NodeName -> RunState (Maybe SymbVal)
nextVal n =
 do
   st <- readSTATE
   case lookup n (next_st st) of
     Just x -> return (Just x)
     Nothing -> do --this stops infinite loops on delays
       updateNextVal (n, Unknown)
       return Nothing

updateNextVal :: Pair -> RunState ()
updateNextVal (n, v) =
 do
   st <- readSTATE
   case lookup n (next_st st) of
     Just x ->
       writeSTATE st {next_st = (n,v) : ((next_st st) \\ [(n,x)])}
     Nothing -> do
       writeSTATE st {next_st = (n,v) : (next_st st)}
       return ()

curVal :: NodeName -> RunState SymbVal
curVal n =
```

```
 do
   st <- readSTATE
   case lookup n (cur_st st) of
     Nothing -> fail ("Missing state for node " ++ n)
     Just c -> return c

drive :: Node -> RunState SymbVal
drive (Input i) =
 do
   cc <- currentCC
   let v = VarIx i cc
   addUseIn (i, v)
   return v
drive (Output o) =
 do
   pred <- prevNode (o,0)
   v <- drive pred
   addUseOut (o, v)
   return v
drive (Delay d i) =
 do
   next <- nextVal d
   case next of
     Just x -> curVal d
     Nothing -> do --calculate new value for next CC
       pred <- prevNode (d, 0)
       v <- drive pred
       updateNextVal (d, v)
       curVal d
drive (Function f fun) =
 do
   dp <- currentDP
   let k = nParams fun
   let prvs = getPrevNodes dp f
   let pns = map (lookupNode dp) prvs
   vals <- mapM drive pns
   return (symbApply fun vals)
{- This version is hard for humans to use
drive (JoinPoint j) =
 do
   con <- prevNode (j, 2)
   c <- drive con
   case c of
     Unknown -> return Unknown
     (VBool False) -> do
       pred <- prevNode (j,0)
       return (drive pred)
     (VBool True) -> do
       pred <- prevNode (j,1)
```

```
       return (drive pred)
     z -> return Unknown -}
drive (JoinPoint j) =
 do
   con <- prevNode (j, 1)
   c <- drive con
   case c of
     Unknown -> return Unknown
     (Var s) -> do
       dp <- currentDP
       let pns = getPrevNodes_n dp (j,0)
       if elem s pns then do
         let pn = lookupNode dp s
         drive pn
        else fail ("Node " ++ s ++ " is not connected to Join Point " ++ j)
     z -> return Unknown

semCC :: RunState ()
semCC =
 do
   st <- readSTATE
   let dp = datapath st
   let delays = getDelays dp
   let outputs = getOutputs dp
   mapM_ drive (outputs ++ delays)
   st' <- readSTATE
   let (ins, outs) = up st'
   let (is, os) = next_up st'
   let up' = (is : ins, os : outs)
   writeSTATE st' {cur_cc = (cur_cc st) + 1, next_up = ([],[]), up = up', cur_st = next_st st', next_st = []}

graphsem :: RunState UsagePattern
graphsem =
 do
  st <- readSTATE
  if (cur_cc st) <= (last_cc st)
     then do
        semCC
        graphsem
     else do
       let (ins, outs) = up st
       return (reverse ins, reverse outs)

run :: Datapath -> Config -> Int -> UsagePattern
run dp cf k =
 case appSTATE graphsem newstate of
    Ok (x,y) -> x
  where
   newstate = DPState dp' k 0 initial [] ([],[]) ([],[]) []
```

```
   dp' = case appSTATE (addController cf) dp of
     Ok (x,y) -> x
   initial = getInitialState dp'

runIO :: Datapath -> Config -> Int -> IO ()
runIO dp c n=
 do
   let res = run dp c n
   displayUP_n (n+1) res

runHTML :: String -> Datapath -> Config -> Int -> IO ()
runHTML file dp c n =
 do
   let res = run dp c n
   let html = upToHTML_n (n+1) res
   writeFile file html

col2Delays :: Column -> Result ()
col2Delays (nm, vs) =
 do
  let k = (length vs) - 1
  let nm' = "control:" ++ nm ++ "_d"
  constantList vs nm'

addController :: Config -> Result Datapath
addController (ns, bs) =
 do
   let k = length ns
   let cols = optimize (ns, bs)
   mapM col2Delays cols
   let heads = map (\x -> "control:" ++ x ++ "_d_0") ns
   let zips = zip heads (zip ns (replicate k 1))
   mapM connect_n zips
   dp <- readSTATE
   return dp

test dp f =
  case appSTATE f dp of
    (Ok a) -> case a of
      (x,y) -> do
        putStr "Result:"
        putStrLn (show x)
        putStrLn "State:"
        putStrLn (show y)
```

CoreLoom.hs

```
module CoreLoom
```

```
 (
  module Datapath,
  module Configuration,
  module SymbVal,
  module UsagePattern,
  module Run
 ) where
import Datapath
import Configuration
import SymbVal
import UsagePattern
import Run
```

Examples.hs

```
import CoreLoom

--A simple R = f(X) datapath
--With feedback and externally controlled registers
dp0 :: Datapath
dp0 = makeDatapath dp
 where
  dp = do
   input "X"
   function (SymbFunc "f" 1) "F"
   output "R"
   connections [("X", "F"), ("F", "R"), ("F", "F")]
   registerBefore open "R1" ("F", 0)
   registerBefore open "R2" ("R", 0)

--Straight through the dp
dp0_flat :: Config
dp0_flat =
  (["R1_load",      "R2_load"],
   [[Var "X",         open],
    [open,            Var "F"]])

--an example of looping with reduced throughput
dp0_loop :: Config
dp0_loop =
  (["R1_load",      "R2_load"],
   [[Var "X",        open],
    [Var "F",        open],
    [Var "F",        open],
    [open,           Var "F"]])
```

```
-- R = f(X, Y) with feedback, registers
-- Extra latency added
dp1 :: Datapath
dp1 = makeDatapath dp
 where
  dp = do
   inputs ["X", "Y"]
   function (SymbFunc "f" 2) "F"
   output "R"
   delay open "D1"
   delay open "D2"
   connections_n [("X", ("F", 0)), ("Y", ("F", 1)), ("D2", ("F", 0)), ("D2", ("F", 1))]
   connections [("F", "D1"), ("D1", "D2"), ("D2", "R"),("X", "R"), ("Y", "R")]
   registerBefore open "R0" ("F", 0)
   registerBefore open "R1" ("F", 1)
   registerBefore open "R2" ("R", 0)

--Straight through
dp1_flat :: Config
dp1_flat =
  (["R0_load",     "R1_load",      "R2_load"],
   [[Var "X",       Var "Y",        open],
   [open,           open,           open],
   [open,           open,           open],
   [open,           open,           Var "D2"]])

--An example of how scheduling improves throughput.
cf_dp1_sched :: Config
cf_dp1_sched =
  (["R0_load",      "R1_load",      "R2_load"],
  [[Var "X",        Var "Y",        open],
  [open,            open,           open],
  [open,            open,           open],
  [Var "X",         Var "D2",       open],
  [open,            open,           open],
  [open,            open,           open],
  [open,            open,           Var "D2"]])

--Demonstrates implicit join points
--  Main> configInfo dp2
--   [("into_(R,0)",["A","B","C","D","E"])]
dp2 :: Datapath
dp2 = makeDatapath dp
 where
  dp = do
   inputs ["A","B","C","D", "E"]
   outputs ["R"]
   connections [("A","R"),("B","R"), ("C", "R"), ("D", "R"), ("E", "R")]
--full adder example
```

```
fadd :: Datapath
fadd = makeDatapath dp
 where
  dp = do
   inputs ["X", "Y"]
   outputs ["R", "C"]
   addNode (Function "ADD" XOR)
   addNode (Function "CRY" AND)
   connections_n [("X",("ADD", 0)),("Y", ("ADD", 1)),("X", ("CRY", 0)), ("Y", ("CRY", 1))]
   connections [("ADD","R"), ("CRY", "C")]

--demonstrates embedded modules
embed_test :: Datapath
embed_test = makeDatapath dp
 where
  dp = do
   inputs ["X", "Y", "Z", "W"]
   outputs ["R"]
   addNode (Function "F" XOR)
   addNode (Function "F2" XOR)
   addNode (Function "F3" XOR)
   addDP fadd "ADD1"
   addDP fadd "ADD2"
   connections [("X", "ADD1:X"),("Y", "ADD1:Y"), ("Z", "ADD2:X"),("W", "ADD2:Y")]
   connections_n [("ADD1:R", ("F", 0)), ("ADD1:C", ("F2", 0))]
   connections_n [("ADD2:R", ("F", 1)), ("ADD2:C", ("F2", 1))]
   connections_n [("F", ("F3", 0)), ("F2", ("F3", 1))]
   connections [("F3", "R")]

--Calculates Dot Product R=AC+BD inflexibly
dotP :: Datapath
dotP = makeDatapath dp
 where
  dp = do
    inputs ["A","B","C","D"]
    delays open ["d_0", "d_1", "d_2"]
    functions MUL ["MUL_0", "MUL_1"]
    function ADD "ADD_0"
    output "R"
    connections_n [("A", ("MUL_0",0)), ("C", ("MUL_0",1))]
    connections_n [("B" ,("MUL_1",0)), ("D", ("MUL_1",1))]
    connections [("MUL_0", "d_0"), ("MUL_1", "d_1")]
    connections_n [("d_0", ("ADD_0", 0)), ("d_1", ("ADD_0", 1))]
    connections [("ADD_0", "d_2"), ("d_2", "R")]

--Dot Product with added flexibility
flex_dotP :: Datapath
flex_dotP = makeDatapath dp
 where
```

```
  dp = do
    inputs ["A","B","C","D"]
    delays open ["d_0", "d_1", "d_2"]
    functions MUL ["MUL_0", "MUL_1"]
    function ADD "ADD_0"
    output "R"
    connections_n [("A", ("MUL_0",0)), ("C", ("MUL_0",1))]
    connections_n [("B" ,("MUL_1",0)), ("D", ("MUL_1",1))]
    connections [("MUL_0", "d_0"), ("MUL_1", "d_1")]
    connections_n [("d_0", ("MUL_0",1)), ("d_1", ("MUL_1", 0))]
    connections_n [("d_0", ("ADD_0", 0)), ("d_1", ("ADD_0", 1))]
    connections [("ADD_0", "d_2"), ("d_2", "R")]
    connectTo_n "d_2" [("MUL_0", 0), ("MUL_1", 1)]

--Flat AC+BD
dotP_dotP :: Config
dotP_dotP =
  (["into_(MUL_0,0)","into_(MUL_0,1)","into_(MUL_1,0)","into_(MUL_1,1)"],
   [[Var "A",         Var "C",         Var "B",         Var "D"]])

--An example of feedback
dotP_back :: Config
dotP_back =
  (["into_(MUL_0,0)","into_(MUL_0,1)","into_(MUL_1,0)","into_(MUL_1,1)"],
   [[Var "A",         Var "C",         Var "B",         Var "D"],
    [open,            open,            open,            open],
    [Var "d_2",       Var "C",         Var "B",         Var "D"]])

--An example of a "stuck" configuration
dotP_stuck :: Config
dotP_stuck = (["into_(MUL_0,0)","into_(MUL_0,1)","into_(MUL_1,0)","into_(MUL_1,1)"],
             [[Var "d_2",   Var "d_0",   Var "d_1",   Var "d_2"]])

--Dot Product with registers
reg_dotP :: Datapath
reg_dotP = makeDatapath dp
 where
  dp = do
    inputs ["A","B","C","D"]
    functions MUL ["MUL_0", "MUL_1"]
    function ADD "ADD_0"
    output "R"
    connections_n [("A", ("MUL_0",0)), ("C", ("MUL_0",1))]
    connections_n [("B" ,("MUL_1",0)), ("D", ("MUL_1",1))]
    connections_n [("MUL_0", ("ADD_0", 0)), ("MUL_1", ("ADD_0", 1))]
    connections_n [("MUL_0", ("MUL_0",1)), ("MUL_1", ("MUL_1", 0))]
    connectTo_n "ADD_0" [("R", 0), ("MUL_0", 0), ("MUL_1", 1)]
    connectTo_n "ADD_0" [("ADD_0", 0), ("ADD_0", 1)]
    registerBefore open "d_0" ("MUL_0", 0)
```

```
    registerBefore open "d_1" ("MUL_0", 1)
    registerBefore open "d_2" ("MUL_1", 0)
    registerBefore open "d_3" ("MUL_1", 1)
    registerBefore open "d_4" ("ADD_0", 0)
    registerBefore open "d_5" ("ADD_0", 1)
    registerBefore open "d_6" ("R", 0)

--AC+BD
reg_dotP_flat :: Config
reg_dotP_flat =
  (["d_0_load", "d_1_load", "d_2_load", "d_3_load", "d_4_load", "d_5_load", "d_6_load"],
  [[Var "A",     Var "C",    Var "B",    Var "D",    open,       open,       open],
   [open,        open,       open,       open,      Var "MUL_0", Var "MUL_1",open],
   [open,        open,       open,       open,      open,        open,       Var "ADD_0"]])

--demonstrates repeated scheduling
reg_dotP_rep :: Config
reg_dotP_rep =
  (["d_0_load", "d_1_load", "d_2_load", "d_3_load", "d_4_load", "d_5_load", "d_6_load"],
  [[Var "A",     Var "C",    Var "B",    Var "D",    open,       open,       open],
   [Var "A",     Var "MUL_0",Var "MUL_1",Var "D",   Var "MUL_0", Var "MUL_1",open],
   [Var "ADD_0", Var "C",    Var "B",   Var "ADD_0",Var "MUL_0", Var "MUL_1",Var "ADD_0"],
   [open,        open,       open,      open,       Var "MUL_0", Var "MUL_1",Var "ADD_0"],
   [open,        open,       open,      open,       open,        open,       Var "ADD_0"]])

--demonstrates overlapping scheduling
reg_dotP_over :: Config
reg_dotP_over =
  (["d_0_load", "d_1_load", "d_2_load", "d_3_load", "d_4_load", "d_5_load", "d_6_load"],
  [[Var "A",     Var "C",    Var "B",    Var "D",    open,       open,       open],
   [open,        open,       open,       open,       Var "MUL_0",Var "MUL_1",open],
   [Var "ADD_0", Var "C",    Var "B",    Var "D",    open,       open,       open],
   [open,        open,       open,       open,       Var "MUL_0",Var "MUL_1",open],
   [open,        open,       open,       open,       open,       open,       Var "ADD_0"]])

--A configuration which is impossible to schedule
reg_dotP_nosched :: Config
reg_dotP_nosched =
  (["d_0_load", "d_1_load", "d_2_load", "d_3_load", "d_4_load", "d_5_load", "d_6_load"],
   [[Var "A",    Var "C",    Var "B",    Var "D",    open,       open,       open],
    [open,       open,       open,       open,       Var "MUL_0",Var "MUL_1",open],
    [Var "ADD_0",Var "C",    Var "B",    Var "ADD_0",open,       open,       open],
    [open,       open,       open,       open,       Var "MUL_0",Var "MUL_1",open],
    [Var "ADD_0",Var "MUL_0",Var "MUL_1",Var "ADD_0",Var "d_4",  Var "d_5",  Var "ADD_0"]])

--An example of a Black-Box datapath
--Represents an inflexible encryptor, unpipelined
--latency = 4, TP = 1/4
exampleF :: Datapath
```

```
exampleF = makeDatapath dp
 where
  dp = do
   inputs ["Key", "Data"]
   outputs ["Cipher"]
   function (SymbFunc "encrypt" 2) "ENC"
   connections_n [("Key", ("ENC", 0)), ("Data", ("ENC", 1))]
   connect ("ENC", "Cipher")
   registerBefore open "R0" ("Cipher", 0)
   registerBefore open "R1" ("Cipher", 0)
   constantList [Var "ENC", Var "R0", Var "R0", Var "R0"] "OneList"
   constantList [Var "R1", Var "R1", Var "R1", Var "R0"] "TwoList"
   connect_n ("OneList_0", ("R0_load", 1))
   connect_n ("TwoList_0", ("R1_load", 1))
--An example of a black-box datapath
--represents an arbitrary usage pattern
exampleG :: Datapath
exampleG = makeDatapath dp
 where
  dp = do
   inputs ["Key", "Data"]
   outputs ["Cipher"]
   function (SymbFunc "encrypt" 2) "ENC"
   connections_n [("Key", ("ENC", 0)), ("Data", ("ENC", 1))]
   connect ("ENC", "Cipher")
   registerBefore open "R0" ("Cipher", 0)
   registerBefore open "R1" ("Cipher", 0)
   registerBefore open "R2" ("Cipher", 0)
   constantList [Var "ENC", Var "ENC", Var "R0", Var "R0"] "OneList"
   constantList [Var "R1", Var "R0", Var "R0", Var "R1"] "TwoList"
   constantList [open, open, Var "R1", Var "R1"] "ThreeList"
   connect_n ("OneList_0", ("R0_load", 1))
   connect_n ("TwoList_0", ("R1_load", 1))
   connect_n ("ThreeList_0", ("R2_load", 1))

--A model of OpenCore.org's AES core encryptor
--Latency = 12 TP = 1/12
aes_enc :: Datapath
aes_enc = makeDatapath dp
 where
  dp = do
   inputs ["key", "text_in"]
   outputs ["text_out"]
   function (SymbFunc "KeyExpansion" 1) "EXP"
   function (SymbFunc "InitialPermutation" 2) "INIT"
   function (SymbFunc "RoundPermutation" 2) "ROUND"
   function (SymbFunc "FinalPermutation" 2) "FINAL"
   register open "R0"
   register open "R1"
```

```
   delay open "R2"
   register open "R3"
   joinPoint "JP"
   connections [("key", "R0_load"), ("R0", "EXP"), ("text_in", "R1_load")]
   connectTo_n "EXP" [("INIT", 0), ("ROUND", 0), ("FINAL", 0)]
   connections_n [("R1", ("INIT", 1)), ("INIT", ("JP", 0)), ("JP", ("ROUND", 1))]
   connections_n [("ROUND", ("R2", 0)), ("R2", ("JP", 0)), ("R2", ("FINAL", 1))]
   connections_n [("FINAL", ("R3_load", 0)), ("R3", ("text_out", 0))]
   constantList ((Var "key") : (replicate 11 (Var "R0"))) "ZeroList"
   constantList ((Var "text_in") : (replicate 11 (Var "R1"))) "OneList"
   constantList ([Var "R2", Var "INIT"] ++ (replicate 10 (Var "R2"))) "TwoList"
   constantList ((replicate 11 open) ++ [Var "FINAL"]) "ThreeList"
   connect_n ("ZeroList_0", ("R0_load", 1))
   connect_n ("OneList_0", ("R1_load", 1))
   connect_n ("TwoList_0", ("JP", 1))
   connect_n ("ThreeList_0", ("R3_load", 1))

--A model of OpenCore.org's decryptor
--The key is loaded before the text
--Latency = 24, TP = 1/24
aes_dec :: Datapath
aes_dec = makeDatapath dp
 where
  dp = do
   inputs ["key", "text_in"]
   outputs ["text_out"]
   function (SymbFunc "KeyExpansion" 1) "EXP"
   function (SymbFunc "KeyReversal" 1) "REV"
   function (SymbFunc "InitialInverse" 2) "INIT"
   function (SymbFunc "RoundInverse" 2) "ROUND"
   function (SymbFunc "FinalInverse" 2) "FINAL"
   register open "R0"
   register open "R1"
   delay open "R2"
   register open "R3"
   joinPoint "JP"
   connections [("key", "R0_load"), ("R0", "EXP"),
                ("EXP", "REV"), ("text_in", "R1_load")]
   connectTo_n "REV" [("INIT", 0), ("ROUND", 0), ("FINAL", 0)]
   connections_n [("R1", ("INIT", 1)), ("INIT", ("JP", 0)), ("JP", ("ROUND", 1))]
   connections_n [("ROUND", ("R2", 0)), ("R2", ("JP", 0)), ("R2", ("FINAL", 1))]
   connections_n [("FINAL", ("R3_load", 0)), ("R3", ("text_out", 0))]
   oneTimeList ([Var "key"], [Var "R0"]) "ZeroList"
   constantList ((replicate 12 (Var "R1")) ++ [Var "text_in"]
                 ++ (replicate 11 (Var "R1"))) "OneList"
   constantList ((replicate 13 (Var "R2")) ++ [Var "INIT"]
                 ++ (replicate 10 (Var "R2"))) "TwoList"
   constantList ((replicate 23 open) ++ [Var "FINAL"]) "ThreeList"
   connect_n ("ZeroList_0", ("R0_load", 1))
```

```
   connect_n ("OneList_0", ("R1_load", 1))
   connect_n ("TwoList_0", ("JP", 1))
   connect_n ("ThreeList_0", ("R3_load", 1))

--A combination of the encryption and decryption cores
aes_comb :: Datapath
aes_comb = makeDatapath dp
 where
  dp = do
   inputs ["key", "text_in"]
   outputs ["text_out"]
   function (SymbFunc "KeyExpansion" 1) "EXP"
   function (SymbFunc "KeyReversal" 1) "REV"
   function (SymbFunc "InitialPermutation" 2) "INIT"
   function (SymbFunc "RoundPermutation" 2) "ROUND"
   function (SymbFunc "FinalPermutation" 2) "FINAL"
   function (SymbFunc "InitialInverse" 2) "INIT_I"
   function (SymbFunc "RoundInverse" 2) "ROUND_I"
   function (SymbFunc "FinalInverse" 2) "FINAL_I"
   register open "R0"
   register open "R1"
   delay open "R2"
   delay open "R3"
   register open "R4"
   joinPoints ["JP", "JP2"]
   connections [("key", "R0_load"), ("R0", "EXP"), ("EXP", "REV"), ("text_in", "R1_load")]
   connectTo_n "EXP" [("INIT", 0), ("ROUND", 0), ("FINAL", 0)]
   connectTo_n "REV" [("INIT_I", 0), ("ROUND_I", 0), ("FINAL_I", 0)]
   connections_n [("R1", ("INIT", 1)), ("INIT", ("JP", 0)),
                  ("JP", ("ROUND", 1))]
   connections_n [("ROUND", ("R2", 0)), ("R2", ("JP", 0)),
                  ("R2", ("FINAL", 1))]
   connections_n [("R1", ("INIT_I", 1)), ("INIT_I", ("JP2", 0)),
                  ("JP2", ("ROUND_I", 1))]
   connections_n [("ROUND_I", ("R3", 0)), ("R3", ("JP2", 0)),
                  ("R3", ("FINAL_I", 1))]
   connections_n [("FINAL", ("R4_load", 0)), ("FINAL_I", ("R4_load", 0)),
                  ("R4", ("text_out", 0))]
   constantList ([Var "R2", Var "INIT"]
                 ++ (replicate 10 (Var "R2"))) "TwoList"
   constantList ([Var "R3", Var "INIT_I"]
                 ++ (replicate 10 (Var "R3"))) "ThreeList"
   connect_n ("TwoList_0", ("JP", 1))
   connect_n ("ThreeList_0", ("JP2", 1))

--Configuration for encryption
aes_comb_enc :: Config
aes_comb_enc = (["R0_load", "R1_load", "R4_load"],
               [[Var "key", Var "text_in", open],
```

```
                [Var "R0", Var "R1", open],
                [Var "R0", Var "R1", open],
                [Var "R0", Var "R1", open],
                [Var "R0", Var "R1", open],
                [Var "R0", Var "R1", open],
                [Var "R0", Var "R1", open],
                [Var "R0", Var "R1", open],
                [Var "R0", Var "R1", open],
                [Var "R0", Var "R1", open],
                [Var "R0", Var "R1", open],
                [Var "R0", Var "R1", Var "FINAL"]])

--Configuration for decryption
aes_comb_dec :: Config
aes_comb_dec = (["R0_load", "R1_load", "R4_load"],
               [[Var "key", Var "R1", open],
                [Var "R0", Var "R1", open],
                [Var "R0", Var "R1", open],
                [Var "R0", Var "R1", open],
                [Var "R0", Var "R1", open],
                [Var "R0", Var "R1", open],
                [Var "R0", Var "R1", open],
                [Var "R0", Var "R1", open],
                [Var "R0", Var "R1", open],
                [Var "R0", Var "R1", open],
                [Var "R0", Var "R1", open],
                [Var "R0", Var "R1", open],
                [Var "R0", Var "text_in", open],
                [Var "R0", Var "R1", open],
                [Var "R0", Var "R1", open],
                [Var "R0", Var "R1", open],
                [Var "R0", Var "R1", open],
                [Var "R0", Var "R1", open],
                [Var "R0", Var "R1", open],
                [Var "R0", Var "R1", open],
                [Var "R0", Var "R1", open],
                [Var "R0", Var "R1", open],
                [Var "R0", Var "R1", open],
                [Var "R0", Var "R1", Var "FINAL_I"]])

--Demonstrates "Black Box" functions from UsagePattern.hs
testBB :: Datapath
testBB = makeDatapath dp
 where
  dp = do
   inputs ["X", "Y"]
   outputs ["R", "S"]
   funcPipelined "AND_0" AND 4
   funcPipelined "AND_1" AND 3
```

```
   connections_n [("X", ("AND_0", 0)), ("X", ("AND_1", 0))]
   connections_n [("Y", ("AND_0", 1)), ("X", ("AND_1", 1))]
   connections [("AND_0_d0", "R"), ("AND_0_d1", "S")]

--BAD EXAMPLES

--Examples of invalid datapaths that demonstrate the checkDatapath function
--All are variants of dp0

bad_NoOutputs :: Datapath
bad_NoOutputs = makeDatapath dp
 where
  dp = do
   input "X"
   function (SymbFunc "f" 1) "F"
   connections [("X", "F"), ("F", "F")]
   registerBefore open "R1" ("F", 0)

bad_notComplete :: Datapath
bad_notComplete = makeDatapath dp
 where
  dp = do
   input "X"
   function (SymbFunc "f" 1) "F"
   output "R"
   connections [("X", "F"), ("X", "R")]
   registerBefore open "R1" ("F", 0)
   registerBefore open "R2" ("R", 0)

bad_Loop :: Datapath
bad_Loop = makeDatapath dp
 where
  dp = do
   input "X"
   function (SymbFunc "f" 1) "F"
   output "R"
   connections [("X", "F"), ("F", "R"), ("F", "F")]
   registerBefore open "R2" ("R", 0)

bad_Disjoint :: Datapath
bad_Disjoint = makeDatapath dp
 where
  dp = do
   input "X"
   function (SymbFunc "f" 1) "F"
   output "R"
   connections [("X", "F"), ("F", "F")]
   registerBefore open "R1" ("F", 0)
   registerBefore open "R2" ("R", 0)
```

FlexFIR.hs

```
module FlexFIR where

import CoreLoom

fir :: Datapath
fir = makeDatapath dp
 where
  dp = do
     inputs ["Data_in", "K_0", "K_1", "K_2", "K_3"]
     outputs ["Data_out"]
     functions MUL ["M_0", "M_1", "M_2", "M_3"]
     functions ADD ["A_1", "A_2", "A_3"]
     delayN 6 open "D"
     delays open ["S_0", "S_1", "S_2"]
     connect ("Data_in", "D_5")
     connections_n [("Data_in", ("M_0", 1)), ("D_4", ("M_1", 1))]
     connections_n [("D_2", ("M_2", 1)), ("D_0", ("M_3", 1))]
     connections_n [("K_0", ("M_0", 0)), ("K_1", ("M_1", 0))]
     connections_n [("K_2", ("M_2", 0)), ("K_3", ("M_3", 0))]
     connections_n [("M_0", ("S_0", 0)), ("M_1", ("A_1", 1))]
     connections_n [("M_2", ("A_2", 1)), ("M_3", ("A_3", 1))]
     connections_n [("S_0", ("A_1", 0)), ("S_1", ("A_2", 0))]
     connections_n [("S_2", ("A_3", 0))]
     connections [("A_1", "S_1"), ("A_2", "S_2"), ("A_3", "Data_out")]

flexFIR :: Datapath
flexFIR = makeDatapath dp
 where
  dp = do
     inputs ["Data_in", "K_0", "K_1", "K_2", "K_3"]
     inputs ["V_in_0", "V_in_1", "V_in_2", "V_in_3"]
     outputs ["Data_out"]
     outputs ["V_out_0", "V_out_1", "V_out_2", "V_out_3"]
     functions MUL ["M_0", "M_1", "M_2", "M_3"]
     functions ADD ["A_0", "A_1", "A_2", "A_3"]
     constant (VInt 0) "Zero"
     delayN 6 open "D"
     delays open ["S_0", "S_1", "S_2"]
     connect ("Data_in", "D_5")
     connections_n [("Data_in", ("M_0", 1)), ("D_4", ("M_1", 1))]
     connections_n [("D_2", ("M_2", 1)), ("D_0", ("M_3", 1))]
     connections_n [("K_0", ("M_0", 0)), ("K_1", ("M_1", 0))]
     connections_n [("K_2", ("M_2", 0)), ("K_3", ("M_3", 0))]
     connections_n [("K_0", ("A_0", 0)), ("K_1", ("A_1", 0))]
     connections_n [("K_2", ("A_2", 0)), ("K_3", ("A_3", 0))]
     connections_n [("Data_in", ("M_1", 1)), ("Data_in", ("M_2", 1))]
     connections_n [("Data_in", ("M_3", 1))]
```

```
     connections_n [("Data_in", ("A_0", 1)), ("Data_in", ("A_1", 1))]
     connections_n [("Data_in", ("A_2", 1)), ("Data_in", ("A_3", 1))]
     connections_n [("V_in_0", ("M_0", 1)), ("V_in_1", ("M_1", 1))]
     connections_n [("V_in_2", ("M_2", 1)), ("V_in_3", ("M_3", 1))]
     connections_n [("M_0", ("A_0", 1)), ("M_1", ("A_1", 1))]
     connections_n [("M_2", ("A_2", 1)), ("M_3", ("A_3", 1))]
     connections_n [("V_in_0", ("A_0", 1)), ("V_in_1", ("A_1", 1))]
     connections_n [("V_in_2", ("A_2", 1)), ("V_in_3", ("A_3", 1))]
     connections_n [("Zero", ("A_0", 0)), ("S_0", ("A_1", 0))]
     connections_n [("S_1", ("A_2", 0)), ("S_2", ("A_3", 0))]
     connections [("A_0", "S_0"), ("A_1", "S_1"), ("A_2", "S_2"), ("A_3", "Data_out")]
     connections [("M_0", "V_out_0"), ("M_1", "V_out_1")]
     connections [("M_2", "V_out_2"), ("M_3", "V_out_3")]
     connections [("A_0", "V_out_0"), ("A_1", "V_out_1")]
     connections [("A_2", "V_out_2"), ("A_3", "V_out_3")]

cf_FIR :: Config
cf_FIR = (["into_(M_0,1)", "into_(M_1,1)", "into_(M_2,1)", "into_(M_3,1)",
           "into_(A_0,0)", "into_(A_1,0)", "into_(A_2,0)", "into_(A_3,0)",
           "into_(A_0,1)", "into_(A_1,1)", "into_(A_2,1)", "into_(A_3,1)",
           "into_(V_out_0,0)", "into_(V_out_1,0)", "into_(V_out_2,0)", "into_(V_out_3,0)"],
          [[Var "Data_in",  Var "D_4",      Var "D_2",      Var "D_0",
            Var "Zero",     Var "S_0",      Var "S_1",      Var "S_2",
            Var "M_0",      Var "M_1",      Var "M_2",      Var "M_3",
            open,           open,           open,           open]])

cf_VSMUL :: Config
cf_VSMUL = (["into_(M_0,1)", "into_(M_1,1)", "into_(M_2,1)", "into_(M_3,1)",
             "into_(A_0,0)", "into_(A_1,0)", "into_(A_2,0)", "into_(A_3,0)",
             "into_(A_0,1)", "into_(A_1,1)", "into_(A_2,1)", "into_(A_3,1)",
             "into_(V_out_0,0)", "into_(V_out_1,0)", "into_(V_out_2,0)", "into_(V_out_3,0)"],
            [[Var "Data_in",  Var "Data_in",   Var "Data_in", Var "Data_in",
              open,           open,            open,          open,
              open,           open,            open,          open,
              Var "M_0",      Var "M_1",       Var "M_2",     Var "M_3"]])

cf_VSADD :: Config
cf_VSADD = (["into_(M_0,1)", "into_(M_1,1)", "into_(M_2,1)", "into_(M_3,1)",
             "into_(A_0,0)", "into_(A_1,0)", "into_(A_2,0)", "into_(A_3,0)",
             "into_(A_0,1)", "into_(A_1,1)", "into_(A_2,1)", "into_(A_3,1)",
             "into_(V_out_0,0)", "into_(V_out_1,0)", "into_(V_out_2,0)", "into_(V_out_3,0)"],
            [[open,           open,           open,           open,
              Var "K_0",      Var "K_1",      Var "K_2",      Var "K_3",
              Var "Data_in",  Var "Data_in",  Var "Data_in",  Var "Data_in",
              Var "A_0",      Var "A_1",      Var "A_2",      Var "A_3"]])

cf_VVMUL :: Config
cf_VVMUL = (["into_(M_0,1)", "into_(M_1,1)", "into_(M_2,1)", "into_(M_3,1)",
             "into_(A_0,0)", "into_(A_1,0)", "into_(A_2,0)", "into_(A_3,0)",
             "into_(A_0,1)", "into_(A_1,1)", "into_(A_2,1)", "into_(A_3,1)",
             "into_(V_out_0,0)", "into_(V_out_1,0)", "into_(V_out_2,0)", "into_(V_out_3,0)"],
            [[Var "V_in_0",   Var "V_in_1",   Var "V_in_2",   Var "V_in_3",
              open,           open,           open,           open,
```

```
              open,           open,           open,           open,
              Var "M_0",      Var "M_1",      Var "M_2",      Var "M_3"]])

cf_VVADD :: Config
cf_VVADD = (["into_(M_0,1)", "into_(M_1,1)", "into_(M_2,1)", "into_(M_3,1)",
             "into_(A_0,0)", "into_(A_1,0)", "into_(A_2,0)", "into_(A_3,0)",
             "into_(A_0,1)", "into_(A_1,1)", "into_(A_2,1)", "into_(A_3,1)",
             "into_(V_out_0,0)", "into_(V_out_1,0)", "into_(V_out_2,0)", "into_(V_out_3,0)"],
            [[open,           open,           open,           open,
              Var "K_0",      Var "K_1",      Var "K_2",      Var "K_3",
              Var "V_in_0",   Var "V_in_1",   Var "V_in_2",   Var "V_in_3",
              Var "A_0",      Var "A_1",      Var "A_2",      Var "A_3"]])
cf_FIR3 :: Config
cf_FIR3 = (["into_(M_0,1)", "into_(M_1,1)", "into_(M_2,1)", "into_(M_3,1)",
            "into_(A_0,0)", "into_(A_1,0)", "into_(A_2,0)", "into_(A_3,0)",
            "into_(A_0,1)", "into_(A_1,1)", "into_(A_2,1)", "into_(A_3,1)",
            "into_(V_out_0,0)", "into_(V_out_1,0)", "into_(V_out_2,0)", "into_(V_out_3,0)"],
           [[Var "Data_in",  Var "D_4",      Var "D_2",      open,
             Var "Zero",     Var "S_0",      Var "S_1",      open,
             Var "M_0",      Var "M_1",      Var "M_2",      open,
             open,           open,           Var "A_2",      open]])

flexFIR_DP :: Datapath
flexFIR_DP = makeDatapath dp
 where
  dp = do
     inputs ["Data_in", "K_0", "K_1", "K_2", "K_3"]
     inputs ["V_in_0", "V_in_1", "V_in_2", "V_in_3"]
     outputs ["Data_out"]
     outputs ["V_out_0", "V_out_1", "V_out_2", "V_out_3"]
     functions MUL ["M_0", "M_1", "M_2", "M_3"]
     functions ADD ["A_0", "A_1", "A_2", "A_3"]
     constant (VInt 0) "Zero"
     delayN 6 open "D"
     delay open "B_0"
     delayN 2 open "C"
     delays open ["S_0", "S_1", "S_2"]
     connect ("Data_in", "D_5")
     connections_n [("Data_in", ("M_0", 1)), ("D_4", ("M_1", 1))]
     connections_n [("D_2", ("M_2", 1)), ("D_0", ("M_3", 1))]
     connections_n [("K_0", ("M_0", 0)), ("K_1", ("M_1", 0))]
     connections_n [("K_2", ("M_2", 0)), ("K_3", ("M_3", 0))]
     connections [("K_1", "B_0"), ("K_2", "C_1")]
     connections_n [("B_0", ("M_1", 0)), ("C_0", ("M_2", 0))]
     connections_n [("K_0", ("A_0", 0)), ("K_1", ("A_1", 0))]
     connections_n [("K_2", ("A_2", 0)), ("K_3", ("A_3", 0))]
     connections_n [("Data_in", ("M_1", 1)), ("Data_in", ("M_2", 1))]
     connections_n [("Data_in", ("M_3", 1))]
     connections_n [("Data_in", ("A_0", 1)), ("Data_in", ("A_1", 1))]
     connections_n [("Data_in", ("A_2", 1)), ("Data_in", ("A_3", 1))]
     connections_n [("V_in_0", ("M_0", 1)), ("V_in_1", ("M_1", 1))]
     connections_n [("V_in_2", ("M_2", 1)), ("V_in_3", ("M_3", 1))]
     connections_n [("M_0", ("A_0", 1)), ("M_1", ("A_1", 1))]
```

```
     connections_n [("M_2", ("A_2", 1)), ("M_3", ("A_3", 1))]
     connections_n [("V_in_0", ("A_0", 1)), ("V_in_1", ("A_1", 1))]
     connections_n [("V_in_2", ("A_2", 1)), ("V_in_3", ("A_3", 1))]
     connections [("V_in_1", "D_4"), ("V_in_2", "D_3")]
     connections_n [("Zero", ("A_0", 0)), ("S_0", ("A_1", 0))]
     connections_n [("S_1", ("A_2", 0)), ("S_2", ("A_3", 0))]
     connections [("A_0", "S_0"), ("A_1", "S_1")]
     connections [("A_2", "S_2"), ("A_3", "Data_out")]
     connections [("M_0", "V_out_0"), ("M_1", "V_out_1")]
     connections [("M_2", "V_out_2"), ("M_3", "V_out_3")]
     connections [("A_0", "V_out_0"), ("A_1", "V_out_1")]
     connections [("A_2", "V_out_2"), ("A_3", "V_out_3")]

cf_DP_FIR :: Config
cf_DP_FIR = (["into_(M_0,1)", "into_(M_1,1)", "into_(M_2,1)", "into_(M_3,1)",
              "into_(A_0,0)", "into_(A_1,0)", "into_(A_2,0)", "into_(A_3,0)",
              "into_(A_0,1)", "into_(A_1,1)", "into_(A_2,1)", "into_(A_3,1)",
              "into_(V_out_0,0)", "into_(V_out_1,0)", "into_(V_out_2,0)", "into_(V_out_3,0)",
              "into_(D_4,0)", "into_(D_3,0)", "into_(M_1,0)", "into_(M_2,0)"],
            [[Var "Data_in",  Var "D_4",       Var "D_2",      Var "D_0",
              Var "Zero",     Var "S_0",       Var "S_1",      Var "S_2",
              Var "M_0",      Var "M_1",       Var "M_2",      Var "M_3",
              open,           open,            open,           open,
              Var "D_5",      Var "D_4",       Var "K_1",      Var "K_2"]])

cf_DP_DPROD :: Config
cf_DP_DPROD = (["into_(M_0,1)", "into_(M_1,1)", "into_(M_2,1)", "into_(M_3,1)",
                "into_(A_0,0)", "into_(A_1,0)", "into_(A_2,0)", "into_(A_3,0)",
                "into_(A_0,1)", "into_(A_1,1)", "into_(A_2,1)", "into_(A_3,1)",
                "into_(V_out_0,0)", "into_(V_out_1,0)", "into_(V_out_2,0)", "into_(V_out_3,0)",
                "into_(D_4,0)", "into_(D_3,0)", "into_(M_1,0)", "into_(M_2,0)"],
              [[Var "V_in_0",    Var "D_4",      Var "D_2",      open,
                Var "Zero",      Var "S_0",      Var "S_1",      open,
                Var "M_0",       Var "M_1",      Var "M_2",      open,
                open,            open,           Var "A_2",      open,
                Var "V_in_1",    Var "V_in_2",   Var "B_0",      Var "C_0"]])
```