Formal models of cache coherent communication fabrics: from micro-architectures to RTL designs

Hendrik De Vloed
Student # 836845696

A thesis submitted in partial fulfillment of the requirements for the degree of MSc in Computer Science.

Supervisors: Dr. Julien Schmaltz and Dr. Freek Verbeek

June 14, 2016
ACKNOWLEDGMENTS

The author would like to thank dr. Julien Schmaltz and dr. Freek Verbeek for the opportunity to work on this thesis. Dr. Schmaltz has always been a great sounding board and has patiently endured the long-winded approach to complete the thesis.

To Sebastiaan Joosten I owe a debt of gratitude to kickstart my Haskell skills beyond the mere beginner level and to immerse me thoroughly into this enigmatic programming language that has since become one of my top personal interests and challenges to master.

Further thanks are due to Bernard Van Gastel, who provided great ideas to simplify the computational complexity of the type inference algorithm used in the Verilog translator.

A final thank you is due to Harrie Passier, my mentor during the master phase of the Open University course. Although he retired from mentoring duties once the thesis phase began, his efforts to keep motivating me to maintain steady progress studying the courses of the master phase will always remain appreciated.
<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3 Simulation by translation to Verilog</td>
</tr>
<tr>
<td>3.3.1 Testbench interface</td>
</tr>
<tr>
<td>3.3.2 Packet field type specification</td>
</tr>
<tr>
<td>4 Case study: Spidergon cache coherence</td>
</tr>
<tr>
<td>4.1 Overview and top-level</td>
</tr>
<tr>
<td>4.2 Construction of the Spidergon network</td>
</tr>
<tr>
<td>4.2.1 Iteration by recursion</td>
</tr>
<tr>
<td>4.2.2 Virtual channels: locking and coherence messages</td>
</tr>
<tr>
<td>4.3 Packet routing within the Spidergon network</td>
</tr>
<tr>
<td>4.3.1 The broadcast router</td>
</tr>
<tr>
<td>4.3.2 The shortest path router</td>
</tr>
<tr>
<td>4.4 Locking algorithm design</td>
</tr>
<tr>
<td>4.5 Cache coherence algorithm design</td>
</tr>
<tr>
<td>4.6 Simulation</td>
</tr>
<tr>
<td>4.7 Conclusion</td>
</tr>
<tr>
<td>5 Discussion</td>
</tr>
<tr>
<td>5.1 Multi-input xMAS primitives</td>
</tr>
<tr>
<td>5.2 Additional xMAS primitives</td>
</tr>
<tr>
<td>5.2.1 Token semantics: Control Join</td>
</tr>
<tr>
<td>5.2.2 Optional data duplication: ForkAny</td>
</tr>
<tr>
<td>5.2.3 Joitch</td>
</tr>
<tr>
<td>5.2.4 Data duplication without consumption: Peek</td>
</tr>
<tr>
<td>5.3 Single-stage queues and pipelining</td>
</tr>
<tr>
<td>5.4 Suggested WickedXmas improvements</td>
</tr>
<tr>
<td>5.4.1 Higher-level abstractions using generalized sources</td>
</tr>
<tr>
<td>5.4.2 Hierarchical design facilities</td>
</tr>
<tr>
<td>5.4.3 Simulation integration</td>
</tr>
<tr>
<td>5.5 Ambiguities in xMAS</td>
</tr>
<tr>
<td>5.5.1 Push versus pull-based data flow</td>
</tr>
<tr>
<td>5.5.2 Source data expression syntax</td>
</tr>
<tr>
<td>5.5.3 Resolution functions</td>
</tr>
<tr>
<td>5.6 Taxonomy of data-flow equations</td>
</tr>
<tr>
<td>5.7 Intrinsic compositional deadlocks</td>
</tr>
<tr>
<td>5.7.1 Fork-merge</td>
</tr>
<tr>
<td>5.7.2 Fork-switch</td>
</tr>
<tr>
<td>5.8 Enhanced data flow control</td>
</tr>
<tr>
<td>5.9 Issues raised during simulation</td>
</tr>
<tr>
<td>5.9.1 Aspect-oriented crosscutting as a means of limiting hierarchical complexity</td>
</tr>
<tr>
<td>5.9.2 General N-port source/sink submodules</td>
</tr>
<tr>
<td>5.9.3 Combinatorial path signal drivers</td>
</tr>
<tr>
<td>5.9.4 Symbolic types in Verilog</td>
</tr>
<tr>
<td>6 Conclusion</td>
</tr>
<tr>
<td>References</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>xMAS primitives identified by Chatterjee et al.</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Mealy FSM model using xMAS primitives</td>
<td>3</td>
</tr>
<tr>
<td>1.3</td>
<td>WickedXmas graphic xMAS editor</td>
<td>5</td>
</tr>
<tr>
<td>2.1</td>
<td>Cache coherence epochs</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>MOESI states and properties (Sorin et al.)</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Directory based coherence protocol</td>
<td>11</td>
</tr>
<tr>
<td>2.4</td>
<td>Directory-based algorithm: example sequence</td>
<td>12</td>
</tr>
<tr>
<td>2.5</td>
<td>Directory based coherence protocol</td>
<td>12</td>
</tr>
<tr>
<td>2.6</td>
<td>Snooping algorithm: example sequence</td>
<td>14</td>
</tr>
<tr>
<td>2.7</td>
<td>Bus flow control examples</td>
<td>16</td>
</tr>
<tr>
<td>2.8</td>
<td>Data persistence</td>
<td>22</td>
</tr>
<tr>
<td>2.9</td>
<td>Common network topologies</td>
<td>25</td>
</tr>
<tr>
<td>2.10</td>
<td>Octagon network topology (ST Microelectronics)</td>
<td>26</td>
</tr>
<tr>
<td>2.11</td>
<td>Spidergon network topology</td>
<td>26</td>
</tr>
<tr>
<td>2.12</td>
<td>Spidergon shortest path routing</td>
<td>27</td>
</tr>
<tr>
<td>3.1</td>
<td>Recursive instantiation</td>
<td>32</td>
</tr>
<tr>
<td>4.1</td>
<td>Top-level structure of the design</td>
<td>38</td>
</tr>
<tr>
<td>4.2</td>
<td>Typical message flow of a single Spidergon node</td>
<td>39</td>
</tr>
<tr>
<td>4.3</td>
<td>Example message diagram resulting from CPU transaction</td>
<td>40</td>
</tr>
<tr>
<td>4.4</td>
<td>main.wck and spidergon.wck, the instantiation of the Spidergon network</td>
<td>40</td>
</tr>
<tr>
<td>4.5</td>
<td>Recursive instantiation of Spidergon network: single level</td>
<td>41</td>
</tr>
<tr>
<td>4.6</td>
<td>Virtual channel multiplexer/demultiplexer</td>
<td>42</td>
</tr>
<tr>
<td>4.7</td>
<td>Tagging of ingress port</td>
<td>42</td>
</tr>
<tr>
<td>4.8</td>
<td>spidergonbroadcastrouter.wck implementation</td>
<td>43</td>
</tr>
<tr>
<td>4.9</td>
<td>spidergonshortestpathrouter.wck implementation</td>
<td>44</td>
</tr>
<tr>
<td>4.10</td>
<td>Structure of the address locking algorithm</td>
<td>45</td>
</tr>
<tr>
<td>4.11</td>
<td>Integration of locking and coherence algorithm</td>
<td>46</td>
</tr>
<tr>
<td>4.12</td>
<td>spidergonlocker_coalesce.wck state machine pattern</td>
<td>48</td>
</tr>
<tr>
<td>4.13</td>
<td>Cache line lookup by external Verilog module invocation</td>
<td>49</td>
</tr>
<tr>
<td>4.14</td>
<td>Structure of the cache coherence algorithm</td>
<td>51</td>
</tr>
<tr>
<td>4.15</td>
<td>spidergonwo_algorithm.wck design</td>
<td>52</td>
</tr>
</tbody>
</table>
5.1 Generalized fork and merge primitives .................................................. 56
5.2 Token mechanism as binary semaphore ................................................. 56
5.3 Routing of mutually exclusive data: problem statement ......................... 58
5.4 Routing of mutually exclusive data: solution ........................................ 58
5.5 Join/switch design pattern ................................................................. 60
5.6 Peek: observation of queue content ....................................................... 60
5.7 Queue composition, Chatterjee et al. ...................................................... 61
5.8 Queue limit for k=1 ........................................................................ 62
5.9 Full-throughput pipeline stage ............................................................. 63
5.10 Full-throughput pipelined design ......................................................... 64
5.11 National Instruments LabView graphic design environment .................. 65
5.12 Overlapping wires on self-referential blocks ....................................... 66
5.13 Automatically generated symbol of subcircuit ...................................... 66
5.14 Intrinsic Fork-Merge deadlock ............................................................ 72
5.15 Intrinsic Fork-Switch deadlock ............................................................ 74
5.16 Additional handshake mechanism to resolve fork-switch deadlock ......... 75
5.17 Cache lookup .............................................................................. 77
Dedicated to the loving memory of my mother Juliette Achtergaele (1930 - 2012), who continually encouraged me to pursue personal growth and development.
1.1 Context

1.1.1 Cache coherence algorithms

1.1.1.1 Coherence and consistency

The advancement of multicore processors and the introduction of these processors in mainstream desktop computers and in high-end embedded systems has raised the importance of efficient multicore-capable memory subsystems (10). The level of integration, parallelism and clock speed of the processor has largely outpaced the speed at which memories can deliver the necessary instructions and data for it. A classical solution for this bottleneck is the introduction of a memory hierarchy where multiple copies of the same datum are present across several memories of various speed and size. Typically, a limited set of ultra fast processor registers is being supplied its working set data from a larger first level (L1) cache that equally resides inside the processor. This L1 cache is able to communicate at high speeds using on-chip buses. A larger but slower second-level cache (L2) is typically located on a bridge chip between the processor and the physical memory banks.

When a single in-order executing processor is being supplied with data and makes modifications to cached data items, the management of the location where the most recent data value resides is rather straightforward: the memory subsystem closest to the processor that contains a cached datum has the most recent value. Data that is not yet present is fetched from the next level cache and/or from memory. Cached items that need eviction to make room for newly requested data are written towards higher level caches and eventually update the actual physical memory for long term storage.

Nevertheless this overly simplified view quickly collapses in the presence of processors that perform out-of-order execution to gain processing speed, in case the order in which data reaches its final destination becomes significant. This is especially the case when data is
moved to memory-like actors that produce side-effects, such as memory mapped peripherals. Memory representing video buffers to be displayed on-screen or direct memory access (DMA) engines that transfer data without further processor intervention will not tolerate data lingering in intermediate caches or data reaching registers in an improper order, different from what the program writer intended. To alleviate these concerns, many processors introduce dedicated instructions and memory management units to obtain control of how the caching subsystem and memory access instructions operate internally.

One of the issues the designers of these memory subsystems are faced with is the development of algorithms (low-level communication protocols) that maintain an efficient, coherent and consistent cache of the memory subsystem for each of the cache levels and processing units.

A classification of the solutions (10) distinguishes two major categories of these protocols: snooping and directory-based.

In snooping protocols, each processor’s memory caching subsystem attempts to observe messages generated by the cache subsystems of other processors. The subsystem uses the observed messages to update memory items it has cached locally.

In directory-based protocols, a central authority provides answers to the question which of the many potential cache copies holds the currently valid copy.

Using information either obtained by snooping or from a directory, the caches are then responsible to resolve potential conflicts and maintain coherence and consistency (26):

Consistency is basically the question when an update is seen by other processors and in what order it appears in relation to updates of other memory locations.

Coherence is the ensemble of mechanisms that virtually hide the fact that the processor has a cache to the application software by ensuring only the most recent valid copy is ever in use.

A multiprocessor system that is both coherent and consistent ensures that multithreaded programs distributed over multiple processors each having local memory caches run equally correct as the same program running single-threaded or time-sliced on a single processor without cache.

1.1.2 xMAS: eXecutable Microarchitectural Specification

Chatterjee and Kishinevsky (5) propose a graphical structured modelling language to make an abstraction of the flow of data through a computing system, based on the primitive components shown in Figure 1.1. By making use of these primitive components to describe more complex algorithms and data flow diagrams, the intricacies of data passing, availability/validity and the timing aspects thereof remain contained within each primitive without complicating the algorithm described.
As an example, a “fork” component contains, next to the fact that it passes data from one input to two outputs, the fact that the data only moves when both outputs are ready to accept it. If one or more of the outputs is unable to accept the message, the input cannot deliver it and becomes blocked itself.

These strict semantics allow networks to be constructed in a graphical way, containing implied formalized semantics how and when data movement occurs. By specifying the structure of the network and the functions transforming the data, an algorithm can be constructed.

Another example in Figure 1.2 shows the construction of a Mealy finite state machine using xMAS primitives.
1.1.3 Deadlock freedom of communication networks

In (24), communication networks are generalized at a meta-level into a Generic Network-on-Chip (GeNoC) model comprised of interfaces that inject messages, possible routes they take and arbitration strategies should they come into conflict with each other. When expressed as a GeNoC, concrete networks can be modeled and properties can be proven. One major property is deadlock freedom. The property states that there never exists a sequence and combination of messages injected into the communication network such that a message cannot advance and eventually reach its intended destination. Deadlocks are a particularly important concern when designing computer algorithms distributed over different processing nodes, as the successful completion of the algorithm is halted indefinitely when a deadlock occurs, effectively rendering the algorithm useless or incorrect. From this point of view it will be clear that deadlocks will play an important role in the design of cache coherence algorithms as well.

1.2 Problem statement

1.2.1 Ongoing research

Because of the formalized description of the xMAS primitives, they are well suited for automated verification (4). A global property of a network can be proven true, regardless of the data content or timing of data flow. Such proofs can be useful in determining the correctness of an algorithm under all possible and unforeseen circumstances.

At the Open University of the Netherlands, the research group under the supervision of dr. F. Verbeek applies xMAS primitives in their larger efforts for formal automated proof generation concerning deadlock freedom of systems-on-chip communication networks (29).

WickedXmas(22), shown in Figure 1.3, is a tool previously developed at the Open University of the Netherlands that allows composing xMAS networks in a graphical way and producing various file formats suitable for integration in further analysis tools such as the ACL2 theorem prover or custom C programs to prove deadlock-related properties. The author has made small preliminary contributions to extending this program and providing file translation to other formats used in their design flow towards the formal proof generators.

1.2.2 Research questions

1. Are the xMAS primitives defined by Chatterjee sufficient to model
   - multicore SoC interconnection strategies?
   - the selected cache coherence algorithm?

2. What features should a good environment have in order to allow formal modelling of coherence protocols in interconnection networks?
1.2.3 Supporting questions

1. How can the functional correctness of xMAS-designed algorithms be tested and debugged?

2. How can xMAS-designed networks be converted to other representations for further formal analysis?

3. How can atomicity concerns be solved while modeling cache coherence algorithms?

1.2.4 Contributions of this thesis

In order to address the research questions stated above, the following contributions were made:

- A model was made for a non-trivial cache coherence algorithm, first as a rough draft in an existing graphic editor LTSpice, then in the WickedXmas editor. Although operational correctness of the model was not formally proven, the resulting design is a valid real-world application of the xMAS primitive set and is therefore a realistic use case.

- During development of the model, additional non-critical xMAS primitives were identified to improve the ease of design.

- During development of the model, additional critical xMAS primitives were identified, necessary to complete the model.

- Minor modifications were made to WickedXmas graphic editor to support the additional primitives identified.
A simulation program was made to allow functional simulation of the generated model.

The simulation program was repurposed as a translation program to the industry standard Verilog language, to allow simulation of the generated model including modelisation of the surrounding environment. Apart from simulation, the Verilog language also provides a link to the remaining tools in the xMAS verification toolchain developed at Open Universiteit Nederland.

Additional shortcomings, including fundamental deadlocks, of the set of xMAS primitives were identified during simulation. Solutions were proposed.

Shortcomings for the large-scale use of the WickedXmas graphic editor were identified and solutions were proposed.

1.2.5 Outline

The remainder of this document is structured as follows.

In Chapter 2, the underlying terminology and necessary background information will be provided. A brief primer on cache coherence will allow the reader to comprehend the example algorithm implemented. A primer on xMAS will describe the original set of primitives defined by Chatterjee et al. The topological structure of the example design, an 8-core processor design with a specific interconnection network, will be defined.

In Chapter 3, the supporting tools developed to allow xMAS simulation will be described.

In Chapter 4, the Write-Once cache coherence model in the Spigergon topology, developed as a realistic case study for use of the xMAS primitives, will be analyzed.

Chapter 5, Discussion, will elaborate in detail on the findings and shortcomings discovered during the design of the case study and propose solutions to improve xMAS for larger real-world designs.

An executive summary of the lessons learned during the course of this thesis can be found in Chapter 6.
Chapter 2

Background

2.1 Cache coherence algorithms

Throughout the history of microprocessors, caches have been used to speed up access to slow devices and memory. For example, in 1968 the IBM System/360 used a data cache (“buffer storage”) mechanism (14) running at the 80ns cycle time of the main processing unit.

As a side-effect of introducing caches, however, consistency and coherence issues arise in multi-processor systems. As different actors (processors) maintain their local cached value of a datum, its validity may expire when other actors make modifications to their cached value of the same datum. Sorin and Hill (25) specify a set of invariants that must hold in order to maintain a globally coherent memory model:

**Single writer, multiple reader (SWMR) invariant** For any memory location \( A \), at any given (logical) time, there exists only a single core that may write to \( A \) (and can also read it) or some number of cores that may only read \( A \).

**Data-Value invariant** The value of the memory location at the start of an epoch is the same as the value of the memory location at the end of its last read-write epoch.

![Figure 2.1: Cache coherence epochs](image)

In this definition, the lifetime of all cached copies of a memory location is divided in contiguous temporal epochs (Figure 2.1). The SWMR invariant states that in each epoch, there is at most one cached copy with write access or an unlimited number of cached copies with read-only access.

7
A change in the cache access of any core ends an epoch and moves to the next epoch. At that time, the second invariant imposes that any modified value is propagated to all observers in the next epoch, so that they may share a coherent view of the memory location as if no caching was present.

The methods used by the different competing processors to ensure these invariants hold are called cache coherence algorithms.

It is important to note the distinction between cache coherence and memory consistency. The latter is concerned with the ordering of read and write accesses to different addresses throughout the processor-to-memory hierarchy, as seen by multiple parties performing concurrent accesses. Consistency is outside the scope of this thesis, which focuses on the management of cached copies of a single address location.

### 2.1.1 Taxonomies

Cache coherence algorithms can be classified in two main categories (21). This dichotomy is based on the location of the knowledge where the most up-to-date cached information is present. When there is a centralized keeper of information (by analogy to a telephone book called a directory, relating memory addresses and the owner(s) of the current epoch’s data values), the algorithm is called directory-based. When there is no central repository and the information is distributed across the system, the individual parties are forced to inspect transactions, negotiating with other parties. Such algorithms are called snooping.

In snooping algorithms, all parties typically observe a common interconnection bus, searching for transactions involving addresses that are locally cached. By observing the address and the access type, as well as some out-of-band information which depends on the coherence algorithm, each party can either establish the correct owner or can relinquish ownership to another party in order to maintain the SWMR and Data Value invariants. As an optimization to snooping, Lawrence (19) defines snarfing algorithms. Next to observing the address, access type and any meta-information, the data content of the memory access is also observed. This allows faster establishment of the current epoch’s data value.

Snooping and directory-based algorithms form the two principal branches of a particular taxonomy style. Another taxonomy style focuses on the different states any cached address location can have. We can identify at least five common state types, not all of which are used in every algorithm.

**Invalid** The most common state type, indicating that a particular cache has no copy of the memory address.

**Shared** A common state indicating that a copy of the data value is present in the cache, with the additional information that other parties may also hold their own copies at this time. By the SWMR invariant it can be concluded that these copies are all read-only.
Modified  The third common state indicating that the current party's cache holds the data value of the current epoch and that it is not equal to the value of the previous epoch. Again by the SWMR invariant, this means all other parties must consider their copy read-only or invalid.

These three principal states are sufficient to model the invariants and construct a coherence algorithm, whether directory-based or snooping. Such an algorithm will be classified in the taxonomy as MSI, by the initials of the states.

More complex algorithms introduce extra states such as

Owned  The current party's cache is the conceptual owner of the memory location and, being owner, might have modified the data value so that it is more recent than the value in main memory. Other caches might have read-only copies.

Exclusive The current party's cache contains an up-to-date read-only copy of the memory location and is the only party that caches the location. Although quite similar to the Shared state, the information of being the only party can improve performance should the access need to be converted from read-only to read-write.

Forward  Similar to the Shared state, but the current party is considered responsible for responding to data value queries from other parties that try to establish their own copy. It is assumed that copies can created faster by inter-cache traffic than by requesting the data value from memory.

This list is not exhaustive, as each specialized algorithm could introduce additional states.

The order of the state initials in the algorithm taxonomy carries no meaning. The order MOESIF prevails throughout the literature.

The aforementioned states can be further broken into their constituent independent Boolean properties, which will here be denoted without capital to distinguish them from states. Any state can be

dirty if the data value of the current epoch differs from the value at the start of the epoch, i.e. (per the second invariant) the value at the end of the previous read-write epoch has been overwritten locally.

exclusive if the current cache is the only party containing a cached value, as in the Exclusive state.

valid if the current cache contains a cached value at all, regardless of the access mode.

Not all combinations of these Boolean properties are independent, as shown in Figure 2.2: it only makes sense to consider dirtiness and exclusivity for valid states.
2.1.2 Example of a directory based coherence algorithm

A directory based algorithm relies on a single entity (“directory”) to manage information about the ownership and the existence of cached copies of memory regions. This does not imply there is only one directory for the system: for example, if the memory is distributed across many nodes, it makes sense to have the node containing a segment of memory be the manager of the cached copies for that segment.

Figure 2.3 shows the three states and their transitions. Note that, although conceptually identical (invalid, read, read/write), the directory has different state names than the nodes. An example access sequence is given in Figure 2.4 where two CPU nodes request read access to a location followed by a third node requesting write access to the same location. When the first node requests read access to an address, the CPU read miss results in a read message being sent to the directory. The directory upgrades the state of the cached address from Uncached to Shared, sends the data to the requesting CPU#1 and takes note that CPU#1 now has a cached copy.

Once the second node requests read access, the directory remains in the Shared state, sends the data to CPU#2 and adds it to its directory.

When the third node tries to obtain write access and sends a write miss notification to the directory, the directory first notifies the two first CPU nodes to invalidate their cache. The directory transitions to the Exclusive state, sends the current data to CPU#3 and takes note that it is now the exclusive owner of the cacheable location. The node’s state indicating write ownership is Modified.

Further state transitions all follow the same access pattern: the requesting node addresses the directory. The directory uses its own state and its knowledge of the states of the different nodes to deduce what other nodes to inform of the pending state transition.
2.1.3 Example of a snooping coherence algorithm

A snooping algorithm relies on cooperation and communication between nodes to derive information about the ownership and the existence of cached copies of memory regions. There is no central authority. Figure 2.5 shows a specific type of snooping protocol, the Write Once protocol proposed by Goodman (8, 1).

The protocol contains four states:

**Invalid** indicates the address is not present in the current cache.

**Valid** indicates the address is present in the current cache, with read-only semantics. There is no write access allowed as more than one node may be in this state simultaneously.

**Reserved** indicates the current cache has exclusive write access to the memory location. Particular to the write-once protocol is that while the cache has been written to, the
Figure 2.4: Directory-based algorithm: example sequence

Figure 2.5: Write-Once snooping protocol
write has also been propagated through to the memory subsystem. This has important implications that improve performance:

- Although the current cache has gained exclusive read/write access to the location, the cached value itself is not dirty, and may later be evicted at no further cost.
- If another cache later requests read access to the location, the Reserved state may be downgraded to Valid at no extra cost as the data is consistent with the value the other cache will fetch from memory.
- The write-through operation acts as a notification to the other caches so that they can infer the state transition and invalidate any of their copies.

Dirty follows the Reserved state once multiple writes occur in sequence. Only the first write to a location results in the write-through access to memory associated with the Reserved state. Further writes will upgrade the state to Dirty and will not result in additional write-through memory accesses. This allows gaining the performance benefits associated with a write-back memory strategy. It has the implication that once an eviction is required or when another cache requests read access, a final write-back operation to memory must be inserted before another cache can complete their read transaction.

An example of the Write-Once algorithm is given in Figure 2.6. At the start we assume only CPU2 has a valid read-only cached copy.

When CPU1 requests its own read-only cached copy, the read operation is snooped by both other caches. Because the access is a read access, the read-only copy of CPU remains valid. CPU1 receives the data from the memory subsystem and also establishes its valid copy.

When CPU1 writes to the cached location, the local cache is upgraded to the Reserved state. The write operation also generates a write bus cycle through to memory which is observed by the other caches. CPU2 now invalidates its read-only copy as it has become stale.

If CPU3 now reads the location, CPU1 observes the transaction and can simply downgrade its copy from Reserved to Valid, as it knows its own cache is not dirty.

If CPU1 performs multiple write transactions to the same location, the first transaction will again upgrade the Valid copy to Reserved and invalidate CPU3’s copy. Subsequent writes remain confined within the cache subsystem of CPU1 and are not written back to memory nor visible on the bus. The state of CPU1 is now Dirty.

When another cache such as CPU2 now requests read access to the same location, the bus transaction causes CPU1 to flush its dirty value to memory, from which it can be retrieved to complete CPU2’s request. This requires a mechanism where CPU1 can overrule or stall the memory transaction that would otherwise have yielded a stale memory value to CPU2.

In the remainder of this thesis, the Write-Once algorithm will be used as a case study.
2.2 xMAS: eXecutable Microarchitectural Specification

Historically, the realisation of digital circuits has evolved from low-level individual transistor based-design to ever increasing hierarchical abstractions. In the interest of having a more efficient design flow compared to designing every individual transistor circuit, early application-specific integrated circuits (ASICs) were based on a two-dimensional array of fixed logic gates. For example, a freely interconnectable array (“gate array”) of either NAND or NOR gates can be used to construct arbitrary Boolean functions, including memory circuits such as flip-flops and latches. By using a customizable interconnection layer (metallization layer), a gate array designer can realize arbitrary logic functions without having to resort to individually drawing three-dimensional layers forming the transistor wells.

As every higher level logic function (AND, OR, flip-flop) was generated using location-specific interconnections between a fixed set of gates, functionally simulating correctness of individual logic functions was still required. Later design methodologies improved on this requirement by providing a customizable library of derived logic functions as basic primitives. An extended set of non-primitive gates such as exclusive-OR gates, flip-flops, multiplexers
were made available as a library of three-dimensional cells ("standard cells"). Because these have a fully characterized layer build-up, individual logic functions became identical and independent of their location on the die. Therefore their timing and electrical characteristics became proven and reproducible, allowing simulation to be abstracted to the timing delays of the cells and the interconnection channels (routing distances across the die).

Starting from the latter part of the 1980s, the design methodology has similarly abstracted away from drawing graphic diagrams representing the logic gates to be realized. Textual design languages (VHDL, Verilog, VHDL-AMS) allow creating iterated structures including buses or width-parameterized circuits such as a n-bit adder. These languages also allow defining models that simulate the external environment and provide stimuli to the circuits under test.

Even these languages are becoming supplanted by ever-increasing conceptual levels. Languages for behavioural synthesis such as Xilinx AccelDSP or BlueSpec SystemVerilog are trying to generate implementations based on specification.

One of the possible higher abstraction levels was proposed by Chatterjee et al. In this abstraction level, the motion of data across interconnection networks is key. The abstraction also targets the creation of provable properties (invariants) to perform automated functional correctness checking.

2.2.1 Flow control

Traditional VHDL or Verilog-based designs perform data processing by writing custom Boolean conditions that govern the consuming and producing of data. Often, the presence or absence of meaningful data involves additional book-keeping in the logic functions. For example, an additional Boolean signal often accompanies a datum, indicating whether or not during a particular clock cycle a bus carrying a datum is meaningful. An analogy of such a conditionally present datum \( d \) in conventional programming languages would be C#'s `Nullable<Datum> d;` or Haskell's `d :: Maybe Datum`.

Similarly, the consumption of valid data can also be conditional and needs to be incorporated in the logic equations that control the production of more data.

Informally, a design pattern with basic invariants has been recognized:

- Data can only be consumed when it is available (valid).
- It is only desirable to produce new data when already available data is being consumed, otherwise data would be lost.

Although this design pattern is key to the flow control of xMAS, it is not new. For example, the ARM AXI4 Stream bus (30, 2) defines active-high signals \( TVALID \) and \( TREADY \). As shown in Figure 2.7a, any produced data is qualified as \( TVALID \) by the originator and is consumed when the destination asserts \( TREADY \).
Similarly, the Peripheral Component Interconnect (PCI) bus standard (27) (Figure 2.7b) uses a flow control mechanism based on active-low DEVSEL# (device selection), IRDY# (initiator ready) and TRDY# (target ready) signals:

“Write data is stable and valid when IRDY# is asserted; read data is stable and valid when TRDY# is asserted. Data is transferred during those clocks where both IRDY# and TRDY# are asserted.”

On the example of Figure 2.7b, the data transfer occurs at the fifth clock edge.

The xMAS flow control uses identically-named but active-high iRDY and tRDY signals. Note that the PCI analogy cannot be extended further:

- The data is not a fixed 32-bit bus, but is essentially unlimited and can vary according to each individual connection.
- There is no address phase or device selection signals (IDSEL and DEVSEL#): all xMAS connections are point-to-point.

Figure 2.7: Bus flow control examples
• There are no bursts or transaction boundaries: each valid data transfer cycle is independent.

• All signals are continuously driven: tri-stating of signals does not occur. During cycles where no valid data is present, the data bus is undefined.

• Apart from \texttt{irdy} and \texttt{trdy} there are no ancillary signals such as \texttt{BE\#} ("byte lane enable", qualifying the validity of individual bytes within the 32-bit data): Should the data only be partially meaningful, such a qualification should be part of the data being transferred.

• Transactions cannot be aborted: once the initiator commits to sending data, the data remains constant and available until (if ever) \texttt{trdy} consumes it.

The latter of these conditions is not formally part of the original xMAS specification, although the definition of a \texttt{source} primitive can be interpreted in that way. This will be discussed in Section 5.5.1.

2.2.2 Basic primitives

Chatterjee et al. define eight primitives from which networks are constructed. We will examine the behaviour of these in detail in a structured manner using a tabular format:

<table>
<thead>
<tr>
<th>Graphic symbol</th>
<th>(Name of primitive)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>(List of input pin names)</td>
</tr>
<tr>
<td></td>
<td>For each of the inputs, equations describing the behaviour of the \texttt{trdy} signals generated by this primitive</td>
</tr>
<tr>
<td>Outputs</td>
<td>(List of output pin names)</td>
</tr>
<tr>
<td></td>
<td>For each of the outputs, equations describing the behaviour of the \texttt{irdy} and \texttt{data} signals generated by this primitive</td>
</tr>
<tr>
<td>Dual to</td>
<td>other primitive(s) that have a behaviour dual to the current primitive</td>
</tr>
</tbody>
</table>

2.2.2.1 Source

\[
\begin{array}{|c|c|}
\hline
\text{Source} & \\
\hline
\text{Outputs} & o \\
\hline
\text{o.data} & := \text{expression} \\
\text{o.irdy} & := \text{(ready to generate) \lor z^{-1}(o.irdy} \land \neg o.trdy) \\
\hline
\text{Dual to} & \text{Sink} \\
\hline
\end{array}
\]
Here, $z^{-1}$ represents the temporal delay operator, i.e. last clock cycle's expression. The $trdy$ signal is therefore generated when either a new datum becomes available, or when last time there was already a datum available ($irdy$) that was unable to be consumed ($\neg trdy$).

A source represents a generator of data. The expression governing the actual data content, data type and the condition that governs when data becomes available are all outside the scope of the formal xMAS specification.

### 2.2.2.2 Sink

<table>
<thead>
<tr>
<th>Sink</th>
<th>i.trdy := (ready to consume) $\lor$ $z^{-1}.(i.trdy \land \neg i.irdy)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>i.trdy := a.trdy $\land$ b.trdy</td>
</tr>
<tr>
<td>Outputs</td>
<td>a, b</td>
</tr>
<tr>
<td>a.data := i.data</td>
<td>b.data := i.data</td>
</tr>
<tr>
<td>a.irdy := i.irdy $\land$ b.trdy</td>
<td>b.irdy := i.irdy $\land$ a.trdy</td>
</tr>
</tbody>
</table>

A sink represents a consumer of data. The condition that governs when the sink is able to consume a datum is outside the scope of the formal xMAS specification.

### 2.2.2.3 Fork

<table>
<thead>
<tr>
<th>Fork</th>
<th>i.trdy := a.trdy $\land$ b.trdy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>i.trdy := a.trdy $\land$ b.trdy</td>
</tr>
<tr>
<td>Outputs</td>
<td>a, b</td>
</tr>
<tr>
<td>a.data := i.data</td>
<td>b.data := i.data</td>
</tr>
<tr>
<td>a.irdy := i.irdy $\land$ b.trdy</td>
<td>b.irdy := i.irdy $\land$ a.trdy</td>
</tr>
</tbody>
</table>

A fork is able to move data from its input i to both of its outputs a and b if and only if both outputs are ready to accept the data. The data is duplicated verbatim.
2.2.2.4 Join

<table>
<thead>
<tr>
<th>Inputs</th>
<th>a, b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.trdy := o.trdy ∧ b.irdy</td>
<td></td>
</tr>
<tr>
<td>b.trdy := o.trdy ∧ a.irdy</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Outputs</th>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td>o.data := aggregate(a.data, b.data)</td>
<td></td>
</tr>
<tr>
<td>o.irdy := a.irdy ∧ b.irdy</td>
<td></td>
</tr>
</tbody>
</table>

Dual to Fork

A join is able to simultaneously produce data on its output o if and only if both inputs are ready to deliver data. The data produced is a combination of both inputs. The formal xMAS specification allows defining a function to aggregate the data content of both inputs.

2.2.2.5 Switch

<table>
<thead>
<tr>
<th>Inputs</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>i.trdy := (a.irdy ∧ a.trdy) ∨ (b.irdy ∧ b.trdy)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Outputs</th>
<th>a, b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.data := i.data</td>
<td></td>
</tr>
<tr>
<td>b.data := i.data</td>
<td></td>
</tr>
<tr>
<td>a.irdy := i.irdy ∧ s(i.data)</td>
<td></td>
</tr>
<tr>
<td>b.irdy := i.irdy ∧ ¬ s(i.data)</td>
<td></td>
</tr>
</tbody>
</table>

A switch function inspects the input data using a user-supplied function s that inspects the data content and determines to which outputs the data should be routed.
2.2.2.6 Merge

<table>
<thead>
<tr>
<th>Inputs</th>
<th>a, b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.trdy := arbitrated ∧ o.trdy ∧ a.irdy</td>
<td></td>
</tr>
<tr>
<td>b.trdy := ¬ arbitrated ∧ o.trdy ∧ b.irdy</td>
<td></td>
</tr>
<tr>
<td>arbitrated := fairness algorithm</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Outputs</th>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td>o.data := arbitrated ? a.data: b.data</td>
<td></td>
</tr>
<tr>
<td>o.irdy := a.irdy ∨ b.irdy</td>
<td></td>
</tr>
</tbody>
</table>

A merge takes data from either of the two inputs and attempts to route each individual datum to the output. Should there be congestion because both inputs are presenting data simultaneously, a fair arbitration mechanism decides the input selection order.

The fairness algorithm proposed by Chatterjee et al. can be summarized as:

- If the previous datum has not yet been consumed, maintain the existing arbitration
- If the previous datum was consumed,
  - If there is only one input bearing valid data (irdy), arbitrate that input
  - If no inputs have valid data, the most recent arbitration is remembered for use in the next case:
  - If both inputs have valid data, arbitrate the opposite input to ensure fairness and avoid starvation

2.2.2.7 Queue

<table>
<thead>
<tr>
<th>Inputs</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>i.trdy := ¬ fifo.empty</td>
<td></td>
</tr>
<tr>
<td>fifo.write := i.irdy ∧ ¬ fifo.full</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Outputs</th>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td>o.irdy := ¬ full</td>
<td></td>
</tr>
<tr>
<td>fifo.read := o.trdy ∧ ¬ fifo.empty</td>
<td></td>
</tr>
</tbody>
</table>
The first-in-first-out queue is parameterizable for depths $\geq 1$. Contrary to intuition, a queue depth of 1 does not equal a simple pipeline register stage. By observing the condition that a full queue cannot be written and that an empty queue cannot be read, it can be shown that a transfer of a single datum through a queue of size 1 needs to take at least two clock cycles: one to write the empty register, one to read it back out.

### 2.2.2.8 Function

<table>
<thead>
<tr>
<th>Inputs</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>i.trdy</td>
<td>:= o.trdy</td>
</tr>
<tr>
<td>Outputs</td>
<td>o</td>
</tr>
<tr>
<td>o.data</td>
<td>:= function(i.data)</td>
</tr>
<tr>
<td>o.irdy</td>
<td>:= i.irdy</td>
</tr>
</tbody>
</table>

A function does not influence the flow control and only alters the data content and type.

### 2.2.3 Data path

Let us assume the source in Figure 2.8 is always ready to generate a datum, i.e. it is an eager source. Assume the content of the datum produced is equal to the current simulation cycle. If we attach a consumer that is not always ready to consume data, e.g. a queue of depth 1 (which is only able to consume data once every two cycles) or a sink that is not eager, the source will only be able to transfer valid data at the rate of the queue or sink, as indicated by the three transfers (T1, T2, T3) highlighted.

If we focus on the data content, we can distinguish two possible interpretations:

- A non-persistent source can generate new data content at every possible simulation cycle. The data content can therefore change while there is no transfer.

- A persistent source only generates new data content at the moment the generation “oracle” function determines a new datum can be generated. The generation oracle function is evaluated either when there was no data yet ($\neg$irdy) or when a transfer successfully consumed the previous data (irdy$\land$trdy).

The xMAS specification does not specify the content of the data itself, nor does it formally specify if a source should behave in a persistent or non-persistent manner. As demonstrated in this example, the assumption of data persistence has an influence on the data content transferred.
2.2.4 Generalization of the number of I/O ports

The primitives presented by Chatterjee et al. are limited to a maximum of two inputs or outputs. A straightforward improvement is the generalization to an arbitrary number of inputs or outputs while preserving functional equivalence. Depending on the primitive, one of the following generalisation categories can be distinguished:

2.2.4.1 Endpoints

For the endpoints source and sink, a generalisation to multiple ports is not very meaningful and is equivalent to either

- instantiating multiple instances in parallel in case the ports should have independent oracles.
- instantiating a multi-output fork after a source or a multi-input join before a sink in case of a common oracle.

2.2.4.2 Channel transformers

Primitives such as queue and function are intended to be attached to a single channel and would not gain from extension to multiple inputs and outputs.
2.2.4.3 Fork and join

The generalization of the dual fork and join primitives is straightforward by extending the two ports `a` and `b` to an indexed notation.

<table>
<thead>
<tr>
<th>Original two-port fork</th>
<th>Generalized to n + 1 ports</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Original two-port fork diagram" /></td>
<td><img src="image2.png" alt="Generalized to n + 1 ports diagram" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Original two-port join</th>
<th>Generalized to n + 1 ports</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image3.png" alt="Original two-port join diagram" /></td>
<td><img src="image4.png" alt="Generalized to n + 1 ports diagram" /></td>
</tr>
</tbody>
</table>
For the join primitive, the $\otimes$ symbol represents the resolution function by which the data fields of multiple inputs are propagated to the output.

### 2.2.4.4 Switch

Generalizing the switch primitive involves a more important modification. In the 2-input version, the result of the switching function $s$ are the Boolean values $\bot$ or $\top$, representing the first and second output respectively. Extending the switching function to $n + 1$ multiple outputs requires replacing it by an integer-typed function returning the range $[0 \ldots n]$.

<table>
<thead>
<tr>
<th>Original two-port switch</th>
<th>Generalized to $n + 1$ ports</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Original switch diagram" /></td>
<td><img src="image2" alt="Generalized switch diagram" /></td>
</tr>
</tbody>
</table>

### 2.2.4.5 Merge

Similar to the switch primitive, the merge primitive contains a selector $u$ (the currently or most recently arbitrated input) that requires generalization from a Boolean to an integer range.
2.3 The Spidergon network topology

The topology of a graph is determined by the arrangement of channels interconnecting its nodes. Common topologies in computer networks are shown in Figure 2.9. For each of these networks, some key parameters can be determined as a function of the number of nodes (size) of the network:

- The number of interconnecting channels
- The maximum distance between two arbitrary nodes (“network diameter”)

---

**Figure 2.9**: Common network topologies: (a) Bus, (b) Star, (c) Fully connected, (d) Ring, (e) Tree
In on-chip interconnection networks, the designer wants to strike a balance between maximizing the available communication bandwidth and minimizing the amount of hardware resources required to implement the actual channel wiring. One particular design choice was made by ST Microelectronics for use in OC-768 network processors (17). The Octagon topology (Figure 2.10) is based on a ring topology of 8 nodes where each node also has a cross-connection to its opposite node. This allows communication to bypass half of the ring in 1 hop and achieve a maximum routing distance of 2 hops to reach any node.

The Octagon was further generalized by lifting the 8-node restriction. The Spidergon (6) topology has $2N$ nodes $0 \ldots 2N - 1$ that form a ring structure and have cross-connections from node $n$ to node $(n + N) \mod 2N$. Figure 2.11a illustrates a network of size $2N = 10$ featuring the cross-connections prominently. The ring-like diagram can be reorganized as two linear structures that are more suitable for an iterative construction process. Figure 2.11b shows how a pair consisting of a top and bottom node $n$ and $n + N$ form an inductive base case that can be extended to arbitrary $N$.

The Spidergon of size $2N$ has $3N$ edges and allows reaching any node in at most $\left\lfloor \frac{N}{2} \right\rfloor$ hops. These properties can be easily shown:
• All edges can be counted by considering each of the $N$ top and bottom pairs of Figure 2.11b in isolation: for each pair there are 2 unique leftward edges and 1 edge between the nodes of the pair.

• If we take node 0 as an arbitrary starting position in Figure 2.12, we can distinguish four quadrants: two of them have a size $\left\lceil \frac{2N}{4} \right\rceil = \left\lceil \frac{N}{2} \right\rceil$ and are reached through shortest paths going counterclockwise and clockwise along the ring. The two other quadrants are reached across the Spidergon and share a node $N$. As the network is pointsymmetrical (Figure 2.11a), this shortest path routing strategy holds for any starting node, as formally proven by Schmaltz and Borrione (23).
3.1 xMAS expression syntax

To support the design of xMAS networks and the formal verification of deadlock freedom in these networks, a dedicated graphic editor WickedXmas(16) has previously been developed. The editor allows free-form strings to be associated with particular xMAS primitives, e.g.

- A generator expression (“oracle”) for source and sink
- An expression describing the data content for source
- The output selection condition of a switch
- The transformation function of function

These free-form strings have no formal xMAS language specification. For example, in (16) these are set to code fragments in the C language that get merged into an executable C model of the network, that in turn can be translated to Verilog for deadlock verification.

The need for a more formal definition of the content of these expression was recognized by Van Gastel and Verbeek (28). To accomplish this, they identified two categories of expressions:

Matching expressions compare data content and result in a Boolean value. These expressions can be used in a switch primitive to determine the output or in a source to describe the valid content fields as an expression yielding a true result.

Modifying expressions manipulate the data content to generate or transform it. These expressions are applicable in a function.
In order to minimize the divergence from the Van Gastel and Verbeek toolchain, their expression syntax was used as a strong guideline. Deviations were made only when unavoidable, e.g. where the expression of the output selection of a multi-output switch requires an integer selector.

The reader is referred to (28) for a more formal BNF specification of the syntax. For clarity, some examples will be given with their typical use.

1. \( \text{color in } \{\text{red, green}\} \, \top \equiv (\text{hue} \equiv 0 \land \text{hue} \leq 255) : (\text{intensity} \equiv 0 \land \text{intensity} \leq 128) \)
2. \( \text{data not in } [3..9] \)
3. \( \text{year} \equiv (\text{month} \equiv 12) \equiv (\text{15} \equiv (6 \equiv 12)) \land (\text{year} \equiv 1) \)

**Listing 3.1: Matching expressions**

In Listing 3.1, three independent matching expressions are shown. Only a single expression can be used in a xMAS primitive. If used in a source expression, line 1 defines the existence of a field \textit{color} with a symbolic data type containing possible values \textit{red} and \textit{green}. There are also numeric fields \textit{hue} and \textit{intensity} with the specified dynamic range. The ternary comparison operator could be used by Van Gastel and Verbeek to infer constraints on possible legal field value combinations, but will be ignored in source primitives in the tools of this work.

When interpreted in the context of a \textit{switch} primitive, the same switching expression of line 1 would be a condition to inspect the fields in an incoming packet and determine the output to route it to. The if-then-else ternary operator \(?\): is interpreted as it would be in C. If the Boolean result of the expression is \(\top\), the first output is selected, otherwise the second output is selected.

The two other example expressions in Listing 3.1 show the use of the \(\text{not}\) in operator with integer ranges and the use of general numeric equations with Boolean operators.

1. \( \text{color} := \text{color with } \{\text{red:green, _:red}\} \)
2. \( \text{hue} := \text{intensity} \equiv 2 \)
3. \( \text{intensity} := 128 - (\text{hue} / 2) \)

**Listing 3.2: Modifying expression**

In Listing 3.2, a single multi-line modifying expression is given, as modifying expressions allow multiple assignments to be concatenated with comma separators. Each field assignment statement := has a left hand side with a destination field and a right hand side that allows constructing a numeric or symbolic value. To distinguish symbolic literals from field names, a field with \{substitution-clause(s)\} construct is introduced. Here, an existing field is inspected and the value is compared to the specified possibilities. If a match is found, an alternate value is substituted. An equivalent Haskell statement describing the expression of Listing 3.2 would be

\begin{verbatim}
  expression (Packet color hue intensity) = Packet color' hue' intensity'
  where
    color' = case color of
      Red -> Green
      _   -> Red
    hue' = intensity + 2
    intensity' = 128 - hue / 2
\end{verbatim}
Note the use of _ as "else" in a case match, as common in Haskell. Using a where with a dummy selector (e.g. integer 0) and a catch-all _ match allows us to assign symbolic literals, e.g. color := 0 with { _ : red }.

### 3.2 xMAS network simulation

Independent of the expression syntax described above, the WickedXmas program allows designing arbitrarily complex hierarchical xMAS networks. Prior to introducing the modifying/matching expression syntax described above, a first attempt at simulating xMAS networks drawn in WickedXmas was made.

The xhas simulator was developed by the author as a precursor to this thesis and is written in the Haskell programming language. It reads a WickedXmas-generated .wck design and outputs an industry standard Value Change Dump (.vcd) format waveform. It makes use of a number of supporting packages, the most important of which are:

- **aeson** for parsing the JSON file format used in WCK files
- **parsec** for parsing function strings
- **vcd** for writing the VCD file format

In the simulator, a XHASChannel represents an interconnection between xMAS primitives, i.e. a wire on the graphic editor.

```haskell
data XHASChannel = XHASChannel {
  fromComp :: Int, -- pos in IBISNetwork.components
  fromOutp :: Int, -- pos in IBISComponent.outps (if applicable) or 0
  toComp :: Int, -- pos in IBISNetwork.components
  toInp :: Int, -- pos in IBISComponent.inps (if applicable) or 0
} deriving Show
```

A XHASComponent represents a xMAS primitive.

```haskell
data XHASComponent =
  Source { nid :: String, outps, state, generator }
  | Sink  { nid :: String, inp, available, oracle }
```

During simulation, each channel carries a live value that is composed of irdy and data. This value is represented as a Maybe XHASPacket, where Nothing means the irdy is deasserted and therefore the data is meaningless. The trdy channel signal flowing in the opposite direction is represented as a simple Bool value.

```haskell
data XHASState = XHASState {
  pktState :: MaybeXHASPacket,
  trdyState :: Bool
} deriving Show
```
The simulator uses the `deriveChannelTypes` function (Listing 3.3) to complete the given network with data type information and with the packet field lengths specified in an ASCII `.pkt` file. The function backtracks all interconnecting channels to source primitives and uses the fields that are mentioned in each source’s expression string as fields for the channel emanating from the source. While tracing back to sources, a list is kept with all visited channels in order to break possible loops in cyclic designs. While tracing through multi-input primitives, the fields existing in all inputs are combined as output fields. A notable exception is the `ctrljoin` extended primitive, which will be described later in Section 5.2.1, for which only the data channel (the last input) contributes to the output data. The process of tracing back to source primitives and breaking loops is done using the auxiliary `deriveChannelType` function, which carries an `[Int]` list along, representing the IDs of the primitives already visited. Haskell’s lazy evaluation mechanism makes it legal that, while backtracking to the sources, each intermediate primitive visited can derive its output types from its input types as if these are already known.

```
deriveChannelTypes :: XHASNetwork -> XHASPacketType -> XHASNetwork
deriveChannelType :: (XHASChannel, [Int]) -> (XHASPacketType, [Int])
declareStateTrace :: XHASComponent -> IO (VCDComponent)
declareChannelTrace :: (XHASChannel, XHASPacketType) -> IO (VCDChannel)
simulationStep :: XHASNetwork -> VCHandle -> VCDTraces
                 -> { tick :: Int } -> { ticks :: Int } -> VCDTraces
                 -> IO ()
```

Listing 3.3: Declarations of key xhas functions

Once the channel types have been derived, the variables that will record the VCD output are defined using `declareStateTrace` and `declareChannelTrace`. Declaring VCD variables generates VCD file output (a side effect), therefore these functions return a monadic type `IO (VCDComponent)` and `IO (VCDChannel)`. The contained types again contain monadic higher order functions: whenever a value is written to one of these trace variables, a corresponding IO action occurs in the VCD file.

```
data VCDComponent = VCDComponent {
  stateVar :: XHASComponent -> IO ()
}
data VCDChannel = VCDChannel {
  dataVar :: MaybeXHASPacket -> IO (),
  trdyVar :: Bool -> IO (),
  irdyVar :: Bool -> IO ()
}
data VCDTraces = VCDTraces {
  stateTraces :: [VCDComponent], -- one trace for each IHasComponentState
  channelTraces :: [VCDChannel]  -- one trace for each IHasChannel
}
```

After declaring the VCD variables, one for each channel and component in the simulation netlist, the VCD file is stepped to the first clock tick and the `simulationStep` function is called in a tail-recursive loop until the required number of ticks is reached.

Each simulation step performs 2 basic actions:

- write the network state into the corresponding VCD trace variables, i.e. both components and channels,
update the state of components and channels using the functions

\[
\text{advanceTime} :: \text{XHASNetwork} \rightarrow \text{Int} \rightarrow \text{XHASNetwork} \\
\text{updateNetworkState} :: \text{XHASNetwork} \rightarrow \text{XHASNetwork}
\]

The former is responsible for moving the component state forward in time, i.e. given the network at \( t - 1 \), calculate the component states at \( t \). The latter then updates the channel states that are produced by the components at time \( t \).

To achieve \( O(N) \) instead of \( O(N^2) \) complexity in the evaluation of the channel states, a lazy list of channel states is used as suggested by Sebastiaan Joosten(15). Haskell lazy evaluation can identify and “cache” the channel states already calculated, avoiding duplicate calculations during recursion, thus preserving linear \( O(N) \) performance.

The channel state calculations for each of the components and the wires they drive are performed using the \( \text{pkt} \) and \( \text{trdy} \) functions, and their \( \text{irdy}' \), \( \text{trdy}' \) variants using pattern matching to select the proper xMAS primitive.

### 3.3 Simulation by translation to Verilog

Although the xhas simulator is able to simulate xMAS networks as intended, its major shortcoming is the lack of expressivity in source expressions to model input data. An implicit variable \( t \) was added to allow generating time-dependent data, but proved still insufficient to model more complex algorithms.

In order to alleviate this shortcoming, it was observed that a standard VHDL or Verilog simulator such as Icarus or Modelsim would be able to perform an identical xMAS network simulation but would allow the inclusion of arbitrarily complex submodules and testbenches. The Verilog language was selected because of the compatibility with pre-existing toolchains that perform deadlock freedom verification.

The WickedXmas editor supports structural recursion and hierarchical designs. A direct translation to Verilog would require generating parametric Verilog instances. Even then, the inferred data types for some hierarchical instances would possibly be non-uniform.

![Recursive instantiation: top, instance \( n = 1 \), instance \( n = 0 \)](image)

Figure 3.1 shows a two-level recursive submodule instantiation leading from a source (top level) through a function (module instance \( n = 1 \)) to a base case module instance \( n = 0 \) containing a sink (right). Assume the source generates a datum of type \( d \) and that the function contains an equation transforming type \( d \) to type \( e \). It is clear that, although a recursive instantiation using a parameter \( n \) is technically possible in Verilog, there is no valid Verilog way to express the changing data type across multiple instances of a parameterized module.
To avoid issues such as these, the translation process uses the flattened .fjson format generated by the WickedXmas program, where all instances have been made unique and individual types can be inferred for each interconnection.

For the same reason, each individual xMAS primitive is implemented in its own Verilog module. Primitives that share the same input and output data types and transformation behaviour (switch condition, function, join field resolution, ...) could be implemented using a single Verilog module. The additional complexity to create such module sharing is not offset by any tangible gain for simulation, therefore each primitive instance has its own module implementation.

The actual implementation of a primitive instance is a direct translation of the logic equations governing the irdy and trdy signals for inputs and outputs. The data fields are transformed if necessary, by the Verilog-translated equivalents of the modifying expression syntax.

```
module FUN$n1_d_to_e (clk, rst,
    .i0$trdy, .i0$irdy, .i0$data$d,
    .o0$trdy, .o0$irdy, .o0$data$e);
    input clk, rst;
    input i0$irdy;
    output i0$trdy;
    input [7:0] i0$data$d;
    output o0$irdy;
    input o0$trdy;
    output [7:0] o0$data$e;

// Function
assign i0$trdy = o0$trdy;
assign o0$irdy = i0$irdy;

// Equations:
// d := null,
// e := d
assign o0$data$e = i0$data$d;

// Deleted fields:
// d
endmodule
```

As an example, the generated code for the function of Figure 3.1 is shown in Section 3.3. The module name is composed of the abbreviated xMAS primitive type FUN and the unique instance name assigned by the flattening in WickedXmas. A $ character is legal in Verilog identifiers and is used as a separator.

All primitives share two global implicit signals rst and clk. Simple combinatorial primitives, such as function or fork, ignore these, whereas primitives that carry internal state, such as queue or merge, can use these to initialize their initial state and clock the next state.

Each input and output is declared and suffixed with an index starting at 0, followed by its constituent irdy, trdy and data signals. In turn, data is composed of the individual fields as separate signals. The fields are determined individually for each interconnection, using a customized version of the the type inference algorithm described by Van Gastel(28):

> The type inference algorithm (see Algorithm 1) is based on iteratively propagating a symbolic packet from a channel to the next channel, connected by
a primitive. Propagation continues until a fix point has been reached, where no new inference can be performed.

**Algorithm 1** Basic propagation algorithm

1: inject source types into channels
2: while not all types in the network are marked as propagated do
3: for all channels in the network do
4: normalise types in channel
5: for all types in the channel do
6: if type is not marked as propagated then
7: propagate
8: mark type as propagated
9: end if
10: end for
11: end for
12: end while

For the intents and purposes of wck2v, the normalisation of the types remains limited to the removal of duplicate fields that occur when multiple inputs carry the same field name.

Once the data types of all channels have been determined, the Verilog module for each primitive can be generated. Finally, the top level Verilog module is generated tying all modules and channels together.

### 3.3.1 Testbench interface

One of the driving forces behind the need for a simulation in Verilog was the lack of expressivity of matching expressions in sources. Even by adding an implicit temporal variable $t$, no internal state can be kept. Instead of trying to create a full-fledged language for sources expressions, the wck2v program allows converting selected sources to input ports of the top-level Verilog module. By symmetry, selected sinks can be converted to top-level outputs.

Still referring to the example design of Figure 3.1, the sources can be converted to a top-level interface by adding the `-c` option and specifying a regular expression that uniquely identifies the source instance name. In our example, the name `s` is matched by the `^s$` regular expression.

```plaintext
./wck2v.exe -t recurse -o recurse.v -c '^s$' recurse.json
```

```verilog
// // Primitive declarations
#include "recurse-definitions.v"
module $q (clk, rst, i0$trdy, i0$irdy, i0$data$d, o0$trdy, o0$irdy, o0$data$d);

//

module FUN$n1_d_to_e (clk, rst, i0$trdy, i0$irdy, i0$data$d, o0$trdy, o0$irdy, o0$data$d);

//

module SNK$n1_n0_s (clk, rst, t, i0$trdy, i0$irdy, i0$data$d);
```

Lines 27–29 show the added top-level signals. Lines 58–60 show that instead of instantiating the source module, the top-level signals are tied into the remainder of the converted xMAS network.

In order to allow the user to add Verilog `define clauses for the bitwise representation of symbolic types, a `include "recurse-definitions.v" statement is provided on line 4.

The user can now write a testbench using the expressivity of the entire Verilog language, which includes access to external files, to provide the signals emanating from source \( s \) in the xMAS design. By using regular expressions to selectively replace source and sink instances, hierarchical designs avoid the need to modify a design and laboriously propagate deeply nested source-driven signals to I/O pins of the top level.

### 3.3.2 Packet field type specification

The inference of channel types by propagating the fields present in source primitives allows each channel to carry the strict minimum number of fields necessary. Nevertheless the type inference algorithm is only concerned with the field names contained in a packet and is not
capable to infer the dynamic range of the individual fields. Implementing such automated inference is part of formal proof derivation and described in Van Gastel’s paper (28) but remains outside the scope of this thesis. To allow the dynamic range of each field to be specified, a simple list of field names and associated bit widths can be provided by the user. For example, the 8-bit field widths for \(d\) and \(e\) in Listing 3.4 can be obtained by providing the packet type specification:

\[
d\ 8 \\
e\ 8 
\]

This list can be provided in the packet type information window available in WickedXmas or can be provided as a separate file using the \(-p\) \texttt{pktfile.pkt} command line option.
In order to assess the capability of xMAS primitives to model non-trivial networks, a case study was proposed. The Write-Once snooping protocol (Section 2.1.3) in conjunction with a Spidergon network topology (Section 2.3) was considered to be a sufficiently complex design choice to exercise the xMAS primitive capabilities to their limits.

### 4.1 Overview and top-level

The structure of the design is shown in Figure 4.1. The design is composed of many hierarchical levels, the most important of which are

- The construction of the Spidergon topology for $2N$ nodes, each containing a CPU, the cache coherence subsystem (subject of the case study) and a distributed RAM. Section 4.2 will describe this part of the hierarchy in more detail.

- The Spidergon packet routing subsystem responsible for inter-node communication of cache- and lock-related messages. A more detailed description can be found in Section 4.3.

- The address locking subsystem responsible for creating an atomic transaction across the entire Spidergon network, as described in Section 4.4.

- The Write-Once algorithm responsible for maintaining cache coherence, as described in Section 4.5.

The principal structure of a Spidergon node is contained in the *spidergonwriteonce.wck* design file and will be described in more detail in Section 4.4. The three central subsystems
Figure 4.1: Top-level structure of the design
are interconnected as depicted in Figure 4.2. A typical message routing is shown in Figures 4.2 and 4.3. It involves the CPU transaction being duplicated into a lock request and broadcast to all other nodes (1), awaiting a series of successful responses from other nodes (2), confirming they are not currently involved in performing a CPU transaction of their own at the same address. Once the locking subsystem confirms the address is not in use, the actual Write-Once algorithm continues processing the transaction (3). Nodes receive and snoop the transactions and reply with their view of the cached address (4). Finally, the received snooping responses are coalesced and the CPU transaction is completed (5).

4.2 Construction of the Spidergon network

4.2.1 Iteration by recursion

Figure 2.11b illustrates the linear representation of a Spidergon network of size $2N$ with $N$ stages that can be chained iteratively. The WickedXmas program does not feature an iterative loop mechanism to generate arbitrary-length structures, but contains a recursion feature (Figure 3.1). This allows generating a nested structure of spidergon.wck designs as in Figure 4.4 where the parameter $n \in [N \ldots 0]$ instantiates the remaining stages. The base case $n = 0$ terminates the recursion. The top-level design main.wck instantiates the recursive structure for the required depth and completes the loops of the clockwise and counterclockwise rings.
Figure 4.3: Example message diagram resulting from CPU transaction

Figure 4.4: main.wck and spidergon.wck, the instantiation of the Spidergon network
Figure 4.5 shows the detailed implementation of the inductive case of the recursion. The UP and DOWN nodes are instantiated and supplied with generic parameters that can be used inside each node to customize circuit behaviour:

$p$ contains the total number of nodes in the Spidergon network, i.e. $2N$.

$n$ is the node number, ranging from 0 to $N - 1$ for the UP nodes and from $N$ to $2N - 1$ for the DOWN nodes.

On each channel, a xMAS queue is present, located at each node’s input for the channel.

### 4.2.2 Virtual channels: locking and coherence messages

As described in Section 2.3, the Spidergon topology consists of bidirectional links between nodes. On these links two kinds of messages must be exchanged: those related to the locking algorithm (Section 4.4) and those related to cache coherence (Section 4.5). To be able to distinguish between these messages, a field `data.kind` is introduced with symbolic values `lockmsg` and `cachemsg`. Each node contains a virtual channel multiplexer/demultiplexer (Figure 4.6) that inspects the input channels and triages the messages according to the `data.kind` field. The field is then removed. Similarly, the field is added on the outputs before the two kinds of messages are merged onto the outgoing channels.

Also visible are the `cpureq` and `cpuresp` source and sink primitives that will be replaced in simulation by a Verilog model representing each node’s CPU.
4.3 Packet routing within the Spidergon network

Many of the routing decisions to be made by the locking and cache coherence algorithms depend on the originating node, the destination node and the port by which messages were received. To keep track of this information, a number of bookkeeping fields are present in the data packet. On the interconnections between Spidergon nodes, such fields are:

**data.kind** As mentioned in Section 4.2.2, this field classifies the message as a message related to the cache coherence algorithm (cachemsg), a message related to the address locking subsystem (lockmsg) or a message from the local node’s CPU subsystem (cpumsg). The latter kind is only for internal use and will never propagate on the Spidergon ring.

**data.type** Depending on the data.kind, the data.type field indicates the meaning of the message. There are three types of messages used to communicate snoopy transactions throughout the network: rdreq, wrreq and reply. For locking messages, there are also three types of messages related to the address locking algorithm: req, ack and nak. There are two types of CPU requests: rd and wr.
**data.origin** The identifier ($n$) of the node at which the packet originated. It is relevant in snoopable transactions and in address lock requests to provide return path information for replies, in case one or more nodes want to reply.

**data.dest** The identifier ($n$) of the node for which the packet is intended. It is relevant in reply messages, as these have a specific destination, i.e. back to the origin. Snoopable transactions and address lock requests are broadcast across the entire Spidergon network, so this field has no meaning for them.

**data.TTL** The time-to-live counter of broadcast messages. The counter is decremented at each hop. Once the counter reaches 0, the receiving node will no longer propagate the message further.

Internal in each node, an additional field **data.ingressport** is added according to the direction (cw, ccw or acr) from which a message was received (Figure 4.7).

### 4.3.1 The broadcast router

Figure 4.8 shows spidergonbroadcastrouter.wck in detail. A broadcast message is queued and its time-to-live is decremented preemptively, to anticipate the transmission on one or more of the three links that will follow. When the message is new, originating in this node, the message is triplicated and sent in all directions by the 3-way broadcast fork. For odd values of $N$, the TTL on the connection across the network is additionally reduced by 1 in order to ensure the furthest nodes will not be reached twice, unlike Figure 2.12b.

If the message originated elsewhere, two possibilities exist.

- If the message came in via the across port (determined by ingress2), it is duplicated by fromacross and retransmitted to the clockwise and counterclockwise directions.
- If the message came in on the clockwise or counterclockwise rings, as determined by the ingress01 switch, it is not duplicated but relayed in the same direction.
A `data.weight` field is added to reflect the implicit double weight of a broadcast message going across the link, as it will be duplicated in the opposite node on arrival.

### 4.3.2 The shortest path router

Figure 4.9 shows `spidergonshortestpathrouter.wck` in detail. The distance function converts the absolute message destination into a relative address \( \text{relad} := (\text{dest} - \$n) \mod \$p \) with respect to the local node position \$n, where \( \text{relad} = 0 \) represents the current node and counts up in a clockwise direction. The subsequent quadrant function again converts the relative address relative to the current node into a relative address relative to a starting node located at the maximum TTL when going across, then counterclockwise. \( \text{relad} := (\text{relad} + (\$p/2) + \text{maxttl} - 1) \mod \$p \) where \( \text{maxttl} = \lceil \frac{\$p}{4} \rceil = \lfloor \frac{\$p+3}{4} \rfloor \).

The `relad` pre-calculation allows a simple comparison operation in the subsequent switches:

- **isacross** if \( \text{relad} \leq 2(\text{maxttl} - 1) \), the destination should be reached by going across, then either clock- or counterclockwise.

- **iscw** if not going across, to determine if the counterclockwise direction is reachable, it suffices to compare the same relative address with \( \text{maxttl} \) beyond the highest node number that was reachable via the across link: \( \text{relad} \leq 2(\text{maxttl} - 1) + \text{maxttl} \), i.e. \( \text{relad} \leq 3\text{maxttl} - 2 \).

Finally, three `noreadr...` functions remove the temporary fields used for relative address calculation.

### 4.4 Locking algorithm design

As described in Section 2.1.3 and shown in Figure 2.6, the Write Once snooping protocol assumes it can observe the bus transactions of other cores. As there is no shared bus in the Spidergon topology, using this class of cache coherence algorithm would require:
Figure 4.10: Structure of the address locking algorithm
- A dedicated interconnection bus containing transaction addresses and coherence messages. The data responses could then flow along the Spidergon routing network. This would essentially defeat the selection of Spidergon as a network topology.

- A timestamping mechanism to create a total temporal ordering in the network, which could be exploited by each core to process messages in the same predictable order. The overhead incurred in terms of decoding and re-ordering logic was deemed too big to warrant further investigation. It would require a significant effort to model this kind of circuitry in xMAS.

- A locking mechanism that creates an atomic transaction for a specific memory access by “reserving” a specific cache address across the Spidergon network.

The latter choice was considered to be a reasonable compromise that is still sufficiently non-trivial to implement.

The locking algorithm is intended to be used as a synchronization primitive (semaphore) to surround a snoopy bus transaction, as in Algorithm 2. Figure 4.11 shows the interaction between the locking algorithm (locker block) and the cache coherence algorithm (woalg block). The incoming CPU request’s asklock fork can only proceed when the locking algorithm is ready to initiate a lock attempt. The incoming CPU request is consumed for processing by the locking algorithm while a copy resides in a single-token queue waitlock.

**Algorithm 2** Snoopy transaction to address $a$

```plaintext
repeat
  Lock memory access to address $a$ across Spidergon network
until Lock succeeded
Broadcast memory transaction to address $a$
All nodes of the network snoop and reply to indicate they observed the transaction.
while # replies < # nodes in the network do
  Coalesce replies
end while
Broadcast unlocking of address $a$ across Spidergon network
```

**Figure 4.11:** Integration of locking and coherence algorithm
Once the locking algorithm has performed its duties, either the lockfailed or the islocked join proceeds, consuming the CPU request from waitlock.

In case of a successful lock, the CPU request proceeds to be processed by the Write-Once cache coherence algorithm. Both the completion of the algorithm and a failed lock request result in a response back to the CPU. This response also causes an unlock request to be returned to the locking algorithm, freeing the pending lock.

Figure 4.10 shows the constituent parts of the algorithm.

**Algorithm 3** Locking of address $a$ at initiator node $n$

- Set current lock address := $a$
  - for each direction (clockwise, counterclockwise, across) do
    - Send lock request data.origin := $n$, data.addr := $a$, data.TTL := $N/2$
  - end for
- ok := true
- while #replies < 4 (quadrants) do
  - ok := ok ∧ reply=ack
- end while
- Successful lock ⇔ ok

**Algorithm 4** Processing of lock messages

- if message is request then
  - data.TTL := data.TTL − 1
  - if data.addr = this node’s current lock address then
    - Stop propagating request
    - Reply nack to node data.origin (shortest path routing strategy)
  - else if data.TTL > 0 then
    - Propagate request (broadcast routing strategy)
  - else if data.TTL = 0 then
    - Stop propagating request
    - Reply ack to node data.origin (shortest path routing strategy)
  - end if
- else if message is response for other node then
  - Propagate response (shortest path routing strategy)
- else if message is response for this node then
  - See Algorithm 3
- end if

Algorithm 3 shows the locking mechanism from the point of view of the originating node. Broadcast messages are sent to the four quadrants of the network along the paths shown in Figure 2.12a. Three messages are sent from the originating node: one counterclockwise, one clockwise and one across. The message sent on the node across is further duplicated into a clockwise and counterclockwise message. This results in a total of four messages, one per quadrant of the Spidergon. Each message carries a time-to-live counter (TTL) that is decremented by 1 on each hop. The value is chosen so each message can reach the furthest node in each quadrant: TTL := $N/2$.

The construction of the while-loop coalescing the responses is shown in Figure 4.12. It forms a state machine pattern that will recur in the Write-Once algorithm: an initial state initval
is injected when a token input (start) permits it. Either the initial state or the ongoing state (held in the state queue) is joined with the input to be processed (ack or nak). Meanwhile, a single-position queue processing blocks any further input requests while the state machine is operating.

Whenever input data to be processed arrives, a resolution function and subsequent switch determine whether or not to move to the next state or to terminate the state machine. The done join removes the state and the waiting token, freeing the state machine for future start requests.

At the output of the state machine, a final function is used to remove (assign null) internal state fields so the type inference algorithm will not propagate them outside the state machine.

Algorithm 4 shows the processing and message propagation of the broadcast messages at each of the nodes. Upon reception of a lock request, each node checks if it is not currently trying to lock the same address. If this is not the case, the node approves the lock request and propagates the message in the same direction (clockwise, counterclockwise or both in case the message came from across the Spidergon). If the message has no more TTL remaining, the entire quadrant has been verified and the request is not propagated further but transformed into a positive response (ack). In the case the address is involved in a lock attempt by the current node, the request will be immediately turned into a negative response (nack) without further propagation.

The routing method to return the response to the originating node uses a shortest path algorithm.

Algorithm 3 will coalesce all four quadrants’ responses. If all of them are successful, the address a remains locked by the originating node and is available for use in the forthcoming...
cache coherence transaction. If any negative acknowledgement is received, the address lock attempt has failed. This indicates there is a possible contention between nodes for updating the same cache line state. The algorithm is conservative and will fail all potential simultaneous lockers. The respective node’s processing units are notified of the failure to obtain exclusive access and are responsible for retrying the access, using a random back-off scheme in order to avoid starvation.

### 4.5 Cache coherence algorithm design

Next to the bus locking problem, a second issue open for interpretation was the location of the memory subsystem with respect to the Spidergon topology. Possible choices were:

- A dedicated additional node added to the topology containing the memory subsystem. Again this would subvert the topology choice by creating a special status for the memory alongside the regular node structure.

- A centralized memory located at or in one of the nodes. The memory could either replace the regular node behaviour (memory node instead of processor node) or it could be co-located at a specific location of the network (node 0 containing both processor and memory subsystem). This is a reasonable choice that would nevertheless create traffic congestion when scaled up to large amounts of processor traffic.

- A distributed memory system where each node hosts part of the total memory subsystem.

The latter choice was selected as it would be more representative of modern Uniform Memory multiprocessor systems and would not create asymmetric access patterns.

Because each node needs to reply to a broadcast message before the initiating node can proceed, any latency-related race conditions are avoided. In a network of $2^N$ nodes, there are $2^N - 1$ nodes that need to reply. The memory subsystem will also reply with the current value, resulting in a total of $2^N$ replies.

**Figure 4.13:** Cache line lookup by external Verilog module invocation

The constituent parts of the Write-Once algorithm are shown in Figure 4.14.

Algorithm 5 and Figure 4.15 show the Write-Once algorithm implemented in the woalg block of Figure 4.11 (spidergonwo_algorithm.wck in Figure 4.14). It is responsible for providing a suitable response to CPU requests of the current node. Prior to invocation of the algorithm, the lookup block (Figure 5.17, spidergonwo_cachelookup.wck) adds a field data.state containing the local node’s cache tag (Invalid, Valid, Reserved, Dirty).
Algorithm 5 Write-once algorithm at initiator node

Look address up in local cache, retrieve state
if CPU write request then
  if Cache hit then
    if State is Valid then
      Broadcast write request, await end
      State := Reserved
      Update data in cache
    else if State is Reserved then
      State := Dirty
      Update data in cache
  end if
else if Cache miss then
  Broadcast read request, await end
  Broadcast write request, await end
  State := Reserved
  Update data in cache
end if
else if CPU read request then
  if Cache hit then
    Return data in cache
  else if Cache miss then
    Broadcast read request, await response
    State := Valid
    Update data in cache
    Return data in cache
  end if
end if

The remaining portion of the Write-Once cache coherence protocol, the snooping algorithm for transactions made by remote nodes, is shown as Algorithm 6 (spidergonwo_snoopal.g.wck). Observing a transaction changes the state of the local node's cache line, if present, according to the semantics of the Write-Once protocol. For each transaction observed, a response is sent back to the originating node. This allows the originating node to count responses and know that all nodes have performed snooping.

The responses typically have no further meaning and carry a Invalid data.state field. There is one exception: in order to perform eviction of a dirty cache line, the state Dirty is sent back to the originating node. This allows a direct cache-to-cache update of the most recent memory content. This is a minor performance-driven modification of the Write-Once protocol, to avoid the bus locking mechanism (last transaction of Figure 2.6) to prevent the memory subsystem from providing a response.

In the originating node, all responses are coalesced. The response of the memory subsystem is preempted by the Dirty response of the dirty cache.
Chapter 4 Case study: Spidergon cache coherence

4.6 Simulation

Simulating the resulting design can be performed using common Verilog-2001 capable simulators such as Icarus Verilog or Mentor Graphics Modelsim.

When using Icarus Verilog, the compilation command for a design main.wck converted to a top-level Verilog file main.v and accompanying testbench main.tb.v is

```bash
export WFMFORMAT=xvt2 # or alternatively: vcd
export VLOGOPTS="-Wall -Winfloop -g2001 -tvvp"
iverilog $VLOGOPTS -o main.vvp main.tb.v main.v
vvp main.vvp -$WFMFORMAT +wfmyfile=main.$WFMFORMAT
```
Algorithm 6 Processing of coherence messages

Look address up in local cache, retrieve state
if Cache hit then
  if Message type = Read request then
    if State = Dirty then
      State := Valid
      Response := Dirty
    else if State = Reserved then
      State := Valid
      Response := Invalid
    else if State = Valid then
      Response := Invalid
    end if
  else if Message type = Write request then
    State := Invalid
    Response := Invalid
  end if
else if Cache miss then
  Response := Invalid
end if
Send response to originating node (shortest path)
This will produce a simulation result waveform in the more compact LXT2 waveform format, readable in GTKWave. In order to actually create the waveform file and record all internal simulation signals into it, the testbench needs to include a piece of Verilog code

```verilog
initial begin
  if (!$value$plusargs("wfmf=", wmf)) begin
    $dumpfile(wfm);
    $dumpvars;
  end
end
```

Compiling and simulating the design in Mentor Graphics is done by invoking

```
vlog -reportprogress 300 -work work main.v main.tb.v -novopt
vsim -novopt -voptargs="acc -00" work.test
```

The -novopt switches can be omitted to enable optimization at the cost of longer compilation times. A distinct advantage of using the Modelsim simulator is the iteractivity by which waveforms can be logged and inspected. The environment contains a TCL scripting engine that allows creating user-defined procedures to visualise and inspect signals of interest, e.g.

```
proc wavecpu {n} {
  add wave -noupdate -group cpu$n -p /cpu$n/*
}
proc wavecpus {{cpus {1 2 3 4 5 6 7 8}}} {
  foreach cpu $cpus {wavecpu $cpu}
}
proc waverings {} {
  for {set ring 0} {($ring<8)} {incr ring} {
    set from $ring
    set cw [expr ($ring+1)%8]
    set ccw [expr ($ring+7)%8]
    set acr [expr ($ring+4)%8]
    foreach {dir dest} {cw $cw ccw $ccw acr $acr} {
      getsenv [eval ring.$dir$queue $dest]
      waveports $from$dir i
    }
  }
}
proc waveports {{group {}} {sides {i o}}} {
  if {(![string length $group])} {set group [env]}
  foreach side $sides {
    foreach port {0 1 2 3} {
      set data [join [list "*$side$port" "$data"] ""]
      set control [join [list "*$side$port" "$?rdy"] ""]
      puts "$group $data $control"
      catch {add wave -noupdate -group $group -radix unsigned -p $data}
      catch {add wave -noupdate -group $group -radix binary -p $control}
    }
  }
}
```

### 4.7 Conclusion

The case study based on the Write-Once algorithm in the Spidergon topology is a non-trivial medium-scale design that presented several design challenges. In the next chapter, these will be discussed in detail:
• The extension of xMAS primitives to more than two inputs or outputs was a convenience feature to visually simplify the design. It will be discussed in Section 5.1.

• The algorithms implemented have revealed the need for certain kinds of data flow for which no xMAS primitive existed. These will be discussed in Section 5.2.

• The behaviour of the queue of depth 1 is not equivalent to a typical pipeline stage found in digital hardware designs, as will be discussed in Section 5.3

• Section 5.4 will identify several potential improvements to the graphic editor.

• The semantics of several existing xMAS primitives were revealed to contain room for interpretation. Section 5.5 will discuss possible resolutions.

• Several observations were made regarding the data flow conditions of existing xMAS primitives. Whereas Section 5.6 will merely note their incompleteness, Section 5.8 will show that they can actually produce deadlocks that are not obvious to the designer. Section 5.8 extends on this topic to propose a resolution beyond xMAS.

• Finally, Section 5.9 will raise some simulation issues and will make suggestions to address them.
The principal research question of this thesis was to assess the suitability of the xMAS primitives for modelling complex algorithms. During development of the Write-Once algorithm several shortcomings were found, some of which were addressed by extending the set of xMAS primitives. Other modifications were mere convenience improvements. Some fundamental issues remain. In this section we will discuss the observations in detail.

5.1 Multi-input xMAS primitives

The extension of xMAS primitives to an arbitrary number of inputs and outputs, as described in Section 2.2.4, can be categorized as “syntactic sugar”: it allows compactifying multi-level tree-like structures to make the algorithms more visually appealing and concise.

The equivalence of the generalisation of fork, join and primitives to their tree-equivalents is shown trivially by the associativity and the commutativity of the ∧ and ∨ operators that govern their irdy and trdy equations. Similarly, the generalization of a switch is equivalent to the transformation of a 4-way decision switch\(\{\text{case a: A; case b: B; case c: C; case d: D;}\}\) to a pairwise decision tree if\(s=\text{a or } s=\text{b}\) ... else ... (Figure 5.1a)

In the case of the merge primitive, there is a minor semantic difference due to the internal state. The merge deprecates the most recently arbitrated input in order to try and maintain fairness. If we consider a 4-input primitive where input 0 was arbitrated most recently (Figure 5.1b, “prev”) and inputs 1 and 3 are ready during the current cycle, there is no specific preference to select either. If we assume the 4-input merge uses an ascending round-robin scheme to maintain fairness, it is likely that input 1 (the input following the most recently selected input) will be preferred over input 3.

Should the same construct be implemented as a cascading 2-input merge, it is likely that the fairness algorithm of the final merge2c instance will deprecate the most recently used input from merge2a and favor the input from the merge2b primitive, resulting in a selection of input 3.
In general, fairness will still be ensured, although the arbitration order will be different.

5.2 Additional xMAS primitives

5.2.1 Token semantics: Control Join

The use of an access token (binary semaphore, mutex) to arbitrate access to a resource is easily reproduced in xMAS, e.g. in Figure 5.2. Here, an opaque submodule (processing, center) consumes input and produces output. If we assume the submodule can only process a single input and must be protected against a steady stream of inputs while it is operating, a token mechanism such as the one depicted can be used. The mechanism is composed of an initialization stage (top left dashed area). It uses a single-position queue (once) attached
to a dead sink. This queue will block indefinitely after it receives a single token from the bottom output of the \texttt{pass1} \texttt{fork}. Meanwhile, the \texttt{pass1} top output has initialized the token queue.

The presence of the token in the token queue indicates processing is ready for operation. If the token can be consumed by the \texttt{alloctoken join}, the resulting token+data combination will start operation and \texttt{alloctoken} will block any other input for lack of a new token.

When the operation completes, the result is duplicated by the \texttt{done} \texttt{fork} towards the output and towards a mechanism to inject a replacement token into the queue.

If we analyze the data type requirements of the token as observed in the queue, it becomes clear that it is of a null (void) type. There is no need for any data content, only the presence indication \texttt{irdy} suffices.

Nevertheless, the type inference mechanism (Section 3.3, Algorithm 1) will eventually propagate all data fields present in the output stream through \texttt{done}, \texttt{freetoken}, \texttt{initial} to the token queue. In order to avoid this behaviour, it would be necessary to insert a function into the top (token-only) path after the \texttt{done join} and explicitly nullify all known fields. Although this is a valid solution, it would require manual type inference by the designer when inserting the function, an undesirable action.

It was observed that a similar distinction (token-only channels versus data carrying channels) arose during formal deadlock freedom verification by Joosten et al.\cite{Joosten}, more specifically during type coalescing of \texttt{join} primitives. It proved to be more efficient to assume that a single input of the \texttt{join} is a data channel, whereas the other input channel(s) represent \texttt{irdy}-tokens. A semantic convention was introduced to have the output of the \texttt{join} carry the data type of a specific input only.

A formalisation of this assumption was made in order to solve both of the issues raised above, essentially introducing a token semantic as a distinct primitive in \texttt{xMAS}. By explicitly differentiating the \texttt{ctrljoin} from the original \texttt{join}, the intent of carrying a single data input becomes unambiguous. In the primitive, a slanted line connecting the data-carrying input to the output highlights this data path, typically the bottommost input. All other inputs are consumed without any data content propagating to the output. Formalizing this as a primitive:
Referring again to Figure 5.2, the use of ctrljoin primitives for alloctoken and freetoken allows the designer to unambiguously break the type-inference loop between the domain carrying meaningful data into and out of the processing sub-module and the token-processing loop. Note that freetoken's topmost input is used as a “token” input: the channel's data is not observed or necessary: only the presence of a datum is relevant. The bottom input of freetoken represents the new data-type that is to be propagated to the output. In this case it is void because the data itself intentionally represents a typeless token.

### 5.2.2 Optional data duplication: ForkAny

Consider the design of Figure 5.3. A request is processed in a complex sub-module processing. During this time, a copy of the original request is stored in a queue. The outcome of the processing is either a success or a failure token. Once the outcome is known, the waiting request must be passed on, depending on the outcome, via different paths.
Although expressing these semantics lies well within the capabilities of the basic xMAS primitives, as is shown in Figure 5.4, the solution converts the mutually exclusive token semantics into a single-bit data field \texttt{data.ok}. After adding this field to the request data type using a \texttt{join}, it can be used to route the waiting request to the proper branch of a \texttt{switch}. Afterwards, the now meaningless \texttt{data.ok} must be removed. The solution is functional but neither compact nor elegant.

If we return to Figure 5.3 and assume a primitive connecting \texttt{waiting to success} and \texttt{failed}, we can deduce that its required behaviour is in fact quite simple. As the outputs of the processing block are mutually exclusive, only one \texttt{ctrljoin} will become ready at a time. This implies that the \texttt{waiting queue} can signal its readiness to both \texttt{ctrljoin} instances, as if a common channel tied the output of the queue to both \texttt{ctrljoin} instances at once.

The data from the \texttt{queue} must be consumed if at least one \texttt{ctrljoin} instance consumes data. This indicates a logical $\lor$ function connecting the \texttt{trdy} signals.

We can now define a new primitive according to these semantics and first identify its closest sibling primitive. One could argue that the existing \texttt{fork} primitive performs a duplication to all outputs at once, whereas the new primitive performs the same functionality to any ready output. Hence the name \texttt{forkany} was chosen. As a symbol, the right vertical bar from which the outputs stem was broken into pieces, symbolizing the independence of each output from the others.

\begin{center}
\begin{tabular}{|c|c|}
\hline
\textbf{Inputs} & i \\
\hline
\texttt{i.trdy} := \texttt{a.trdy} $\lor$ \texttt{b.trdy} \\
\hline
\textbf{Outputs} & a, b \\
\hline
\texttt{a.data} := \texttt{i.data} \\
\texttt{b.data} := \texttt{i.data} \\
\texttt{a.irdy} := \texttt{i.irdy} \\
\texttt{b.irdy} := \texttt{i.irdy} \\
\hline
\end{tabular}
\end{center}

\textbf{5.2.3 Joitch}

The basic solution to the routing problem of Figure 5.4 contains a design pattern that was observed repeatedly: a \texttt{join} with the express purpose of adding a field on which a subsequent \texttt{switch} can take a decision. Frequently, the field is only intended for the switch and is no longer needed or desirable in the outgoing data.

Such a \texttt{join-switch} primitive has been generalized to \texttt{joitch}. The output data is uniquely dependent on the bottommost input, whereas the switching condition can make use of both
inputs. As in a join, both inputs are consumed simultaneously. The graphic symbol shows the two outputs originating from the bottommost input.

<table>
<thead>
<tr>
<th>Joitch</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
</tr>
<tr>
<td>i0, i1</td>
</tr>
<tr>
<td>i0.trdy := o0.irdy (\lor) o1.irdy</td>
</tr>
<tr>
<td><strong>Outputs</strong></td>
</tr>
<tr>
<td>o0, o1</td>
</tr>
<tr>
<td>o0.data := i1.data</td>
</tr>
<tr>
<td>o1.data := i1.data</td>
</tr>
<tr>
<td>o0.irdy := i0.irdy (\land) i1.irdy (\land) (s(i0.data, i1.data))</td>
</tr>
<tr>
<td>o1.irdy := i0.irdy (\land) i1.irdy (\land) (\neg s(i0.data, i1.data))</td>
</tr>
</tbody>
</table>

Although the utility of this primitive seems limited, formal deadlock verification can make use of the fact that only the bottommost input’s data is propagated.

### 5.2.4 Data duplication without consumption: Peek

All xMAS primitives encountered so far share the same data flow semantic: data is presented at the output(s) of a primitive and, if the attached primitive(s) is/are ready, immediately consumed. During development of the locking algorithm (Section 4.4), a situation arose that did not fit this data flow assumption.

Figure 5.6 shows a simplified version of the lock algorithm. An address is locked by the current node, as symbolized by a lock address residing in a single-element queue lockaddr.
When an unlock token allows it, the lock address is consumed from the queue by the sink unlocked. While a lock is being held, the remaining part of the algorithm needs to process incoming lock requests by other nodes and reply whether or not their requested lock addresses match the currently locked address of the local node. This requires inspecting the queue content without consuming it.

It is possible to construct a complex recirculation mechanism where the queue is constantly being read out and a duplicate is made using a forkany primitive for use in the comparison. After the duplicate is made, the original is put back into the queue under the condition that it is not consumed by an unlock operation. As a 1-entry queue cannot be read and written in the same clock cycle, this recirculation would require at least two clock cycles to complete.

If we consider that on the underlying hardware level, the output of a one-stage queue is a trivial set of data and irdy wires, it must be clear that any complex construction to achieve the same goal highlights the lack of a primitive that is able to observe a channel without actually consuming its data. A new primitive has therefore been defined as peek, symbolized by a split wire. The straight wire section indicates the path along which the irdy/trdy flow control mechanisms operate unchanged. It is equivalent to the identity function. The smaller slanted wire section represents the observed data without trdy flow control.

5.3 Single-stage queues and pipelining
Chatterjee et al. (5) introduce the queue as the elementary data storage element (Figure 5.7). They specify its behaviour using an underlying FIFO storage mechanism with associated full and empty signals:

“A synchronous FIFO queue with a write port (on the left) and a read port (on the right). The queue can store $k$ data elements. In each clock cycle, if the queue is not full, a new element may be inserted; and if the queue is not empty, the oldest element may be removed. read data exposes the oldest element if the queue is not empty. There is no bypass: if the queue is empty, the incoming data appears at the output one cycle later."

The necessary glue logic to connect empty and full signals is then absorbed into the queue primitive (Figure 5.7) and renamed as irdy and trdy.

Let us consider the physical implementation of such a queue element in terms of conventional digital logic circuits used in application-specific integrated circuits (ASIC) and field-programmable gate array (FPGA) technology.

Figure 5.8 shows possible hardware implementations of the limit case where $k=1$, where the queue reduces to a single register stage. As a basic memory element, the flip-flop with enable has been used. When the Enable (En) signal is found to be asserted at the rising edge
of the clock (not shown but represented as the triangle at the bottom of the flip-flop), the data at the \( D \) input is stored in the flip-flop and appears at the \( Q \) output immediately until the next clock edge. If the \( En \) input is found to be deasserted at the rising edge of the clock, the \( D \) input is ignored and the previous \( Q \) output persists for another clock cycle. A flip-flop drawn without an Enable pin behaves as if \( En \) is permanently asserted.

In Figure 5.8a the quoted queue definition using \textit{empty} and \textit{full} has been followed rigorously:

- \( o.\text{data} \) is replaced by a new element from the input if and only if the queue is not full (top flip-flop).
- \( o.\text{irdy} \) reflects the validity of \( o.\text{data} \) and is updated each clock cycle according to both clauses of the strict definition above:
  - if the condition for updating \( o.\text{data} \) is present, i.e. the queue is not full, the validity \( o.\text{irdy} \) of the updated \( o.\text{data} \) reflects the validity of the input data (\( i.\text{irdy} \))
  - if there is valid output data, i.e. the queue is not empty, the validity of the next clock cycle depends on whether or not the output is consumed. If it is not consumed, it remains valid. If it is consumed, the queue becomes empty.

By collapsing the identities \( \textit{full} = \neg \textit{empty} = o.\text{irdy} \), the circuit may be reduced to its minimal form shown in Figure 5.8b.

![Figure 5.9: Full-throughput pipeline stage](image)

This circuit is not equivalent to a primitive pipeline stage encountered in typical digital hardware designs. Because the loading of new data is conditional to the pipeline stage being empty (\( \neg o.\text{irdy} \)), the maximum possible throughput is one datum every two clock cycles. A pipeline typically encountered in hardware would allow re-filling a new datum while the current datum is being consumed. Figure 5.9 shows a possible implementation and its simpler logic conditions for flow control. An example use case is given in Figure 5.10a where data is being transformed using functions \( f[] \) inbetween pipeline stages represented as \( k = 1 \) queue. This is a realistic use case if we assume the processing function \( f[] \) is complex and takes a significant portion (> 50%) of the clock cycle time to produce its result. Performing two successive function applications during the same clock cycle would violate the timing budget and result in a forcibly reduced clock speed for the entire design. The obvious solution to this problem is to insert a queue stage after each function to create a
pipeline capable of running at full speed at the cost of taking one more clock cycle of latency to produce the final result.

A disadvantage of the pipelining mechanism of Figure 5.10a is that the trdy flow control line contains an asynchronous accumulation \( \bigvee_{\text{stages}} \emptyset \). This is required to allow pipeline “bubbles” (individual stages with no valid data, i.e. \( \neg \text{irdy} \)) to be overwritten with valid incoming data, filling the entire pipeline with valid data. As the number of stages increases, the time to perform the \( \bigvee \) operation will eventually become another clock frequency limit. A solution is to periodically insert a \( k \geq 2 \) queue that does provide timing isolation. This strategy is e.g. reflected in the Xilinx 'AXI4-Stream Register Slice' (a.k.a. 'skid buffer') (30) building block, a \( k = 2 \) queue.

A different pipelining strategy that avoids this \( \bigvee \) operation altogether is a simple shift-register pipeline, as shown in Figure 5.10b. Here, the trdy stage is fully controlled by the end-consumer and the data will shift gradually through the pipeline without collapsing “bubbles”. If the producer feeding the input of the pipeline does not supply a valid datum (\( \neg \text{irdy} \)) during the cycle the output of the pipeline is consumed, a bubble is inserted in the pipeline which will eventually propagate to the output after a fixed number of trdy-qualified cycles. The advantage of the pipeline scheme is clear: its clock frequency is independent of the number of stages.

The two proposed pipeline strategies cannot be represented using the queue semantics, although their data flow semantics are compatible to the xMAS rules, even including the additional data path persistence constraint. It follows that the queue primitive is not sufficient as the basic sequental primitive and additional pipelining primitives should be defined. A formalism to generalize the behaviour of pipelines can be found in the work of Cortadella, Kishinevski and Grundmann (7): their SELF protocol is interchangeable with the xMAS protocol as the VALID and STOP signals correspond to irdy and \( \neg \text{trdy} \), respectively.
5.4 Suggested WickedXmas improvements

5.4.1 Higher-level abstractions using generalized sources

When defining complex algorithms, a graphical design flow becomes cumbersome. With increasing design size, it becomes rapidly difficult to maintain an overview of the interaction of primitives in the presence of scores of interconnecting wires, even in specialized environments such as LabView (Figure 5.11). The principles of structured design and software architecture call out for code re-use. This requires the creation of stand-alone submodules that can be re-used and instantiated multiple times to form a complex design.

The WickedXmas design program already allows defining a hierarchical block as a separate design file and instantiating it as a parameterizable subcircuit in another design. During the development of the cache coherence algorithm, this feature was used extensively. The inputs and outputs of a module are represented as special symbols (green and red dots as used in Figure 5.5) that are semantically equal to sources and sinks.

5.4.2 Hierarchical design facilities

The instantiation of a submodule involves the automatic generation of a block-like symbol containing connectable pins representing the aforementioned inputs and outputs, e.g. as shown in Figure 4.11 on Page 46.

The generation of the symbol is done in a pseudo-arbitrary order, currently the order in which the inputs and outputs appear in the design file of the subcircuit. This order in itself
is currently dependent on the moment the input or output symbol is drawn. On the block-like symbol, inputs are added from top to bottom on the left edge of the subcircuit, outputs on the right edge. This results in a tall and narrow block that has several shortcomings:

![Figure 5.12: Overlapping wires on self-referential blocks](image)

- The width of the symbol is fixed. In order to conserve precious space where it is instantiated, the default width is rather small. If the names of the inputs and outputs are long enough, they can collide into each other, making the interface difficult to read. A larger default width would waste space in case of trivial sub-blocks.

- The routing of input and output wires connected to a subcircuit instance is automatic (as is all wire routing in WickedXmas). This often results in overlapping wire segments making it difficult to read the origin and destination of a wire without selecting the originating pin and visually searching for the corresponding selection of the destination pin. Particularly wires that are attached to pins of the same instance are impossible to read. An example is shown in Figure 5.12, the top-level instantiation of the Spider-gon ring of Figure 2.11b on Page 26. Here, four looping wires, corresponding to the counterclockwise and clockwise channels of the top and bottom half-Spidergon, are drawn on top of each other and over the block itself, rendering it totally illegible.

- When abstracting low-level functionality, it is often desirable to have an immediate indication of the functionality of a subcircuit without requiring internal inspection. For example, a dead source can be instantiated directly by placing a source primitive with an oracle function of false and a descriptive name dead. However, in the interest of drawing attention to the fact that the source is dead, it may be interesting to isolate it as a distinct sub-circuit (Figure 5.13a). It is unfortunate that the instantiation (Figure 5.13b) does not reflect the functionality of the subcircuit at a glance and even...
looks visually similar for all one-output subcircuits. A source symbol that is clearly crossed out would be a better visualisation of such a subcircuit instance.

- The order of input and output ports on the symbol is created in the same order as the placement of input and output primitives on the subcircuit. This has important consequences: if an input or output is deleted and immediately re-added, the order of the pin on the symbol changes. Wires attached to the symbol follow the name and remain correct, however the visual layout is damaged.

Some minor issues with room for improvement include the lack of a placement grid necessitating hairline movements to achieve straight lines, the necessity to use the Open dialog to open a subcircuit and the single document interface that does not allow to return to higher design levels after drilling down.

In conclusion, the following features are recommended for inclusion in future versions of WickedXmas:

- Drawing or importing custom symbols for subcircuits
- User-defined placement of output pins on such symbols
- The ability to guide the channel auto-routing by adding intermediate waypoints
- Grid-based placement and routing
- Hierarchical drilling, i.e. entering a subcircuit (e.g. by double-clicking) and returning to the higher level
- Annotations with free-text comments to document certain primitives or channels

### 5.4.3 Simulation integration

During simulation of a flattened netlist, the resulting signals are mangled into unique identifiers by concatenating the hierarchical path. For example, the instance

```
J0I$ring.n.n.Down.writeonce.lookup.getresp
  i2551$J0I$ring.n.n.Down.writeonce.lookup.getresp(.clk(clk), .rst(rst)
    .i0$irdy(sig2548$0$irdy)
    .i1$irdy(sig2550$0$irdy)
    ...
    .o0$data$addr(sig2551$0$data$addr)
);
```

shows a `join` Verilog module instance corresponding to the third recursive instance (`n_n_n`) of the top-level Spidergon ring. In that instance, the `DOWN` node contains a `writeonce` subcircuit, containing a `lookup` subcircuit. The final `getresp` name corresponds to the label of the `join`. The instance name prefixes a unique instance number `i2551` generated during flattening of the netlist.
This instance number can also be used to trace the signal names, e.g. \texttt{sig2551$o0$data$addr}.

For easy signal tracing, each signal is prefixed with the module instance number driving the signal and follows its output indexing scheme. The remaining part of the signal name is composed of the irdy, trdy and data.field parts.

Going back and forth between the simulation and the schematic involves significant overhead to identify the hierarchical path instance and to locate the relevant .wck file.

As a potential improvement, the WickedXmas program could provide hooks that allow integration with the TCL programming language. This would allow TCL-capable programs such as the ModelSim simulator to create scripts that facilitate cross-probing between waveforms and design.

5.5 Ambiguities in xMAS

5.5.1 Push versus pull-based data flow

The definition of a source by Chatterjee et al. (5) imposes only a data persistence requirement on sources and sinks. The generation or consumption of data is governed by a non-deterministic oracle function, e.g. for a source 

\[
\text{o.irdy := oracle or pre(o.irdy and not o.trdy)}
\]

In practice, the conditions to generate new data are dependent on externalities such as the current time. In case the source represents an interface to the outside world, the source is likely to indirectly use signals derived from other sources or sinks. This is a situation that opens up additional possibilities for deadlock. Even for a stand-alone source that observes nothing but its own trdy signal, waiting for the trdy to be asserted before asserting irdy is a source for deadlock, should the attached xMAS primitive insist on seeing irdy asserted before asserting its own trdy.

This potential for deadlock was addressed in the ARM AXI4 Stream bus standard (2), during the definition of the transfer handshake signalling mechanism. The ARM TVALID signal corresponds to irdy in xMAS, while ARM TREADY matches trdy.

\begin{quote}
1. A master is not permitted to wait until TREADY is asserted before asserting TVALID.
2. Once TVALID is asserted it must remain asserted until the handshake occurs.
3. A slave is permitted to wait for TVALID to be asserted before asserting the corresponding TREADY.
4. If a slave asserts TREADY, it is permitted to deassert TREADY before TVALID is asserted.
\end{quote}
The second criterium corresponds to data persistence. The third criterium basically lifts all restrictions on the TREADY assertion. The fourth criterium adds a persistence requirement on the TREADY signal, once it is asserted and the initiator has equally indicated its readiness. Criteria 2 and 3 match the xMAS flow control mechanism with the assumption of data persistence.

A strict (“if and only if”) interpretation of the clause “before TVALID is asserted” in the fourth criterium would imply that once TVALID is asserted, TREADY is no longer allowed to be deasserted. It seems superfluous to add this persistence requirement: assuming the TVALID is already persistent, any combinatorially derived signal (including TREADY) would therefore also become persistent.

The first criterium imposes an additional restriction that can prevent the possible deadlock situation described before.

When applied to the xMAS terminology and extended to multi-port primitives, the ARM AXI4-Stream criteria would translate to:

1. The assertion of \texttt{irdy} of an xMAS output port cannot depend on the assertion of the \texttt{trdy} input signal corresponding to that output port.
2. The assertion of \texttt{irdy} of an xMAS output port must persist until the corresponding \texttt{trdy} input signal assertion (and the resulting data transfer) is observed.
3. The \texttt{trdy} output signal of an xMAS input is allowed (but not required) to wait until the corresponding \texttt{irdy} input signal is asserted.
4. The \texttt{trdy} output signal of an xMAS input can be asserted and deasserted without restriction if and only if the corresponding \texttt{irdy} input signal has not been asserted yet. Once asserted, the \texttt{trdy} output can be asserted but must remain so until the resulting data transfer has taken place.

When observing a network of xMAS primitives, the presence or absence of the first criterium would translate into a push-, respectively a pull-based traffic flow control pattern. If the initiator waits for the target to indicate its readiness before asserting \texttt{irdy}, the flow control is pull-based. If the initiator advertises readiness of new data (as per the original xMAS source semantics), the data is push-based. Deadlocks are likely to occur on the interfaces between network regions that are intrinsically pull-based and regions that are intrinsically push-based, without an intermediate queue to provide the necessary “impedance adaptation”. Examples of non-obvious mismatches by subtle interactions between push- and pull-based flow control in multi-input or multi-output primitives will be discussed further in Section 5.8.

5.5.2 Source data expression syntax

Chatterjee et al. (5) omit a rigid specification of the data produced by an \texttt{source}:
A source is a primitive which is parameterized by a constant expression $e : \alpha$. Each cycle, it non-deterministically attempts to send a packet $e$ through its output port. $o.data := e$

The definition could be interpreted in a very restrictive way so that the data packet produced is immutable in itself: the “constant expression $e$” of data type $\alpha$. This kind of interpretation would restrict the expressivity of a source and is unlikely to be the authors’ intent. It has been assumed the expression $e$ itself is constant (a fixed string representing a computer program) but is able to produce different packet data each time it is evaluated. In this interpretation, $e$ is an impure function written in any suitable language.

As this leaves the interpretation of the expression $e$ outside the scope of the formal xMAS syntax, the interaction between xMAS-based tools is therefore inherently fragmented: each tool tends to define its own domain-specific syntax, e.g. based on C-language fragments for simulation or a custom BNF-defined syntax (Van Gastel, Verbeek and Schmaltz (28)) for data type inference.

The latter approach was adopted in this thesis for simulation purposes because it provides a more stringent definition of the legal expressions that are allowed in the translation to Verilog.

During implementation of the BNF parsing, it was discovered that there the BNF syntax is still incomplete, e.g. the integer comparison operation only defines a comparison between a field and a constant, not two fields.

```
integer-match ::= variable
  | variable compare-op constant
  | variable 'in' '[' constant '..' constant ']
  | variable 'not' 'in' '[' constant '..' constant ']
```

### 5.5.3 Resolution functions

Chatterjee et al. define implicit functions in two primitives: the `fork` and `join`. In case of the `fork` these implicit functions transform the data types of the outputs $a$ and $b$. The choice to allow different data types on both outputs is superfluous: should the designer wish to change the data type of an output, a function primitive can be inserted. Another argument against the implicit `fork` function is given by the `switch` primitive which doesn’t feature a similar transformation function. For this reason, the possibility of having implicit transformation functions in a `fork` was disabled in the supporting tools.

The same arguments cannot be made for a `join`, however. Here, a single output needs to be created from multiple inputs. It is a distinct possibility that inputs share a similar data field. In this case, a specific resolution must be made to identify which input’s field has precedence. The downside of requiring a resolution function in any `join` would be the need for a manual data type inference by the designer, to identify and resolve all data fields involved in each particular `join`. To avoid this burden, a set of implicit defaults was assumed, which should eventually be added to the xMAS formalism in order to make xMAS designs universally unambiguous.
1. The default resolution function is the union of all fields of all inputs.

2. The presence of a specific resolution function augments the default resolution function, it does not supplant it. Even when present, the union of all fields is the starting point for the specific resolution function, which can override individual fields.

3. In case fields are present on multiple inputs, the input with the highest index provides the field value.

4. The specific resolution function can delete fields by assigning a null value.

This set of rules defines the semantics but does not impose a particular syntax for the resolution function itself. The naming of the inputs (e.g. \( i[0], i[1] \) or \( a, b \)) is still dependent on the language used (e.g. Van Gastel et al. (28) as in this thesis).

### 5.6 Taxonomy of data-flow equations

During the development of the Write-Once algorithm, several missing primitives were identified, as described in Section 5.2. One common observation in these new primitives was the need for a data flow control equation that differed fundamentally from the ones available in the existing xMAS primitives. For example, the `peek` primitive ignores the second output's `trdy` signal altogether, a feature that could not otherwise be achieved.

Extending this observation, one could argue that there are many other combinations of Boolean equations that are still unexplored and could lead to the introduction of more primitives. For example, if we consider the `fork`, its data flow equation required a logical `∧` operator on both output's `trdy` signals. The `forkany` extended primitive introduced the complementary operation `∨`. By using the exclusive-or `⊕` operator, we could define another primitive, whose semantics would allow data to propagate if and only if exactly one output is ready to receive data.

A more formal treatment of this generalisation mechanism should also analyze which of these primitives would negate the data persistence assumption.

### 5.7 Intrinsic compositional deadlocks

#### 5.7.1 Fork-merge

During development of the example cache coherence algorithm, simulation deadlocks were encountered for which the reason was not immediately obvious. Figure 5.14a shows a seemingly functional circuit where data from a source `s2` is forked. Each of the outputs is merged with other data and is consumed by sinks. If we assume the `merges` have fair arbitration, the casual observer would not identify possible deadlocks. Although the `fork` could temporarily be unable to proceed due to the one or both `merges` not being arbitrated to
(a) xMAS

(b) Flow control logic

Figure 5.14: Intrinsic Fork-Merge deadlock
corresponding input, the assumption of fair arbitration would guarantee that eventually the fork would be able to proceed.

In Figure 5.14b the equations of Section 2.2 have been used to explicitly draw the logic circuits governing the irdy and trdy signals surrounding the fork and merges. Let us assume all sources s1, s2 and s3 have been idle, the sinks are eager and s2 asserts irdy. One would expect the fork to become ready after gaining arbitration from both merges. When we trace the logic circuit for the f1.i.trdy signal we first observe the logical AND gate U1. For f1 to assert its input’s trdy, the trdy signals of both of its outputs must be asserted. When we focus on either of these (e.g. f1.a.trdy driven by the AND gate U6, we can continue tracing a dependency through U3 to the opposite f1.b.trdy signal. In turn, in order for this signal to be asserted, it requires assertion of the outputs of U7 and U2. To assert U2, the original signal we considered, f1.a.trdy, needs to be asserted.

Although an assertion of all these signals is therefore a stable situation representing the fork ready and delivering its data to both sinks, it will not occur spontaneously due to the mutually dependent combinatorial loop formed by U2, U3, U6 and U7 being deadlocked.

5.7.2 Fork-switch

A similar situation can occur when we replace the merge primitives by switches. Figure 5.15b shows the expansion of the xMAS design of Figure 5.15a using the original definitions of the switch equations. As in Figure 5.14b, a combinatorial loop is formed consisting of U3, U8, U6, U4, U2, U12, U10 and U5.

5.8 Enhanced data flow control

Investigating the reason how intrinsic combinatorial deadlocks can exist, we need to observe that the irdy signal is asserted if and only if the originator is actively producing data. This is a side-effect of the persistence assumption (Figure 2.8 on Page 22). Should we ignore data persistence, the irdy signal would have the semantics:

- The producer is offering data to be consumed to the consumer.
- The producer can retract its offer should it desire to do so.

The data transfer condition remains unchanged: only when both the consumer and producer agree and assert irdy and trdy in unison, data is actually transferred. The principal difference with a persistent assertion of irdy is that there is now room for “negotation” between producer and consumer(s).

The trdy signal on the other hand is not bound to persistence requirements and is the combination of two different conditions, depending on the trdy assertion:

- The consumer could consume data if it were presented with data
Figure 5.15: Intrinsinc Fork-Switch deadlock
• The consumer is consuming the data it is being presented with

Observing these different signal semantics, we can conclude a multi-way handshake could resolve the combinatorial deadlock.

• The producer indicates it is willing to produce data but doesn’t commit to do so.

• The consumer indicates it is willing to consume data, assuming the the consumer is offering it.

• The producer commits to its offer and persists the classical irdy signal

Such a multi-way ireq-tack-irdy handshake between the fork and join primitives is depicted in Figure 5.16. Adding these additional handshake signals could resolve the deadlock problem but would create a new xMAS semantics. When generalized to all sequences of xMAS primitives, adding speculative ireq-tack handshakes would also open up a potential for livelocks by combinatorial oscillations. Such a loop could be caused by
• a tentative ireq becoming revoked by conditions based on the resulting (lack of) \texttt{tack}(s) of other channels

• different propagation delays of different paths along the xMAS network, resulting in temporary assertions/retractions of the handshake signals

The currently best known solution to the intrinsic deadlocks presented in is the insertion of queues into at least one of the branches involved. This breaks any potential for combinatorial loops. The designer does need to take the added latency through the queue into account for the functionality of the design.

5.9 Issues raised during simulation

5.9.1 Aspect-oriented crosscutting as a means of limiting hierarchical complexity

During the design of the Write-Once algorithm, several cross-cutting concerns became prominent. The concept of cross-cutting in aspect-oriented programming (18) involves the need to gather common concerns (and implement them in a modular fashion) that are spread across the program in a way that would normally defeat modularization attempts. In the Write-Once design, a concern is the cache subsystem which is modelled outside of xMAS. In several hierarchically different locations (\texttt{spidergonwriteonce.wck}, \texttt{spidergonwosnoopalg.wck}) cache lookup operations are made. In \texttt{spidergonwriteonce.wck} a cache update is made. These are all referring to the same underlying cache and would therefore require a common implementation located at the highest common hierarchical level. This would involve running channels across hierarchical levels for no other reason than to implement the cache subsystem, without adding any conceptual meaning.

Because of the crosscutting mechanism implemented in wck2v, where regular expressions allow identifying \texttt{source} and \texttt{sink} primitives to be replaced by top-level I/O ports, all cache-related ports can be isolated regardless of what hierarchical design level they occur on. In the top-level testbench, these can all be connected to the cache subsystem implementation without incurring overhead or making structural modifications to the hierarchy.

5.9.2 General N-port source/sink submodules

Orthogonal with but related to the cross-cutting concern, it was identified that many of the \texttt{sources} and \texttt{sinks} extracted using the cross-cutting mechanism occur in pairs. For example, the cache lookup operation implemented in Figure 5.17a consists of a \texttt{sink} in which the address is consumed coupled to an \texttt{source} from which the cache state and optional data (if present) is returned. In case the original request or other non-cache related data needs to be preserved, a bypass queue is required so the response data can be coalesced with data from the original request.
The inherent coupling of source and sink is lost during flattening and replacement by a top-level port in wck2v. Instead of lifting source and sink individually, it would be more convenient to lift the cache lookup interface out of the design at the block level (Figure 5.17b). This would require the possibility to mark sub-modules as black boxes in WickedXmas that don’t get flattened and can be automatically lifted to top level interfaces in wck2v.

### 5.9.3 Combinatorial path signal drivers

A major issue during Verilog verification of the Write-Once algorithm was the propagation of undefined states. Whereas xMAS defines the irdy and trdy flow control signals as having Boolean levels, the Verilog language uses a set of four possible logic values: 0, 1, x and z. Of these four, the latter (z, representing a floating tri-stated signal) is not used at all. The third value (x, an undefined value or a tri-state conflict) can occur in the xMAS simulation when initial states are not properly reset or when data fields that have no meaning at a particular point in time are being inspected and (possibly erroneously) used in logic equations.

Although the global reset signal inferred in the Verilog implementations of all primitives takes care of undefined initialization states, some unexpected combinatorial loops such as the ones described in Section 5.8 can persist after the reset procedure. In turn, these undefined logic levels will eventually “poison” the subsequent stateful queue and merge primitives, resulting in the entire network becoming undefined.

In practice, tracing the undefined levels back to their origins proved to be difficult, especially as they formed many self-sustaining combinatorial loops from which the originating condition was no longer apparent.

A potential improvement would therefore be to add a set of monitor tasks to the Verilog models of the xMAS primitives. Assertions, warning the user about driving outputs to undefined signal levels, would identify problems at their originating nodes in a chronological order.
5.9.4 Symbolic types in Verilog

The Verilog language does not contain the concept of enumerated types. In order to translate symbolic xMAS field values into Verilog, a pragmatic approach was taken: the symbol names are translated into Verilog preprocessor symbols such as `lockmsg references. The actual implementation of these symbols can be defined by the user in a common include file to make use of any suitable encoding for the corresponding hardware signals representing a field. Common encodings are binary with wires of $\log_2(\text{# of symbols})$ bits and one-hot with as many bits as there are symbols.

Although this representation is sufficient, it needs to be noted that type information is lost during translation to Verilog. As shown in the example definitions above, the symbol values for cachemsg and ack can be used interchangeably, as they both map to an integer value. The VHDL hardware description language does include user-defined enumerated types and will not allow the accidental intermixing of enumerated type values. A possible improvement to the wck2v program would be to introduce either the Haskell netlist (9) package or the Haskell C\lambda\ash package (3), both defining abstract syntax tree (AST) representations. By generating an AST instead of direct Verilog statements, either a VHDL or Verilog representation can be produced by package-specific back-ends. Neither of the aforementioned packages currently has provisions for generating enumerated types, although these could be added without significant modifications.
If we refer back to the research questions of Section 1.2.2, it has been demonstrated that the xMAS primitives form a suitable basis for modelling interconnection networks, even non-trivial structures such as the Spidergon network. By implementing iterative and/or recursive modelling features in the supporting toolset (WickedXmas), a network can be constructed that remains generic in the number of nodes. Using a formally specified syntax for the equations in source, switch, function etc., the necessary packet routing decisions can be defined unambiguously. There remains a small amount of ambiguity in the expression language that should be addressed. Assumptions about the tacit resolution of identical data fields for multi-input primitives should be formalized in order to avoid different interpretations across toolchains.

In conjunction with the generation of a flattened netlist and an automated translation to a simulation-capable language such as Verilog, the network can be verified.

The generalisation of xMAS primitives to their n-input or -output equivalents is straightforward, with only minor remarks for the arbitration of n-input merges.

A particular area where xMAS is lacking expressivity is pipelining. Using the $k \geq 2$-deep queue primitives, data processing stages can be formed that are locally decoupled from each other using elastic buffers (FIFOs). When extending the notion of a queue to $k = 1$, however, pipelining becomes severely limited in bandwidth and cannot be used to model the typical register stages used in manually crafted digital logic. Combining the xMAS formalism with the SELF protocol used by Cortadella et al. could address this shortcoming.

The xMAS primitives are not semantically complete and are lacking some expressivity that would be available when modelling directly in an underlying hardware description language. Shortcomings in expressivity that presented themselves during the design of the Write-Once example algorithm were addressed by the introduction of new primitives (ctrljoin, forkany, joitch, peek). Although not all of these will adhere to the persistence assumption, their existence is indicative that a more rigorous exploration of possible Boolean equations for control and data flow will give rise to additional useful primitives.
Further study is required to characterize conditions where seemingly innocuous interconnections of xMAS primitives give rise to unexpected combinatorial loops that cause intrinsic deadlocks. Again, it is assumed that a more rigorous exploration will allow identifying these loops from a theoretical point of view, before they become obvious in simulation.

In order to allow proper simulation of xMAS networks using mainstream Verilog simulation tools, the need for interfacing with non-xMAS-based testbenches and external models was identified. A solution was implemented using a crosscutting mechanism to replace specific source and sink primitives, although further improvements using hierarchical non-xMAS black boxes are possible. The data type mapping relies on manual definitions for encoding symbolic types. Name mangling allows tracing flattened Verilog signal names back to the xMAS hierarchical design. An extension to the VHDL language is feasible in the current translator framework.
REFERENCES


