XLIM

An XML Language-Independent Model

document edition 1.0

ASTG Technical Memo
Xilinx DSP Division
September 18, 2007

Contents

Contents 2

1 Introduction 4
- 1.1 Scope/Context . . . 4
- 1.2 Format Overview . . . 5

2 Element Definitions 6
- 2.1 design . . . 6
- 2.2 module . . . 7
  - 2.2.1 module [kind=block] . . . 7
  - 2.2.2 module [kind=if] . . . 7
  - 2.2.3 module [kind=loop] . . . 8
  - 2.2.4 module [kind=decision] . . . 8
  - 2.2.5 module [kind=mutex] . . . 8
- 2.3 operation . . . 9
- 2.4 port . . . 10
- 2.5 PHI element . . . 10
- 2.6 stateVar . . . 11
- 2.7 initValue . . . 11
- 2.8 actor-port . . . 12
- 2.9 internal-port . . . 12
- 2.10 Finite State Machines . . . 12
- 2.11 other invariants . . . 12
  - 2.11.1 dependencies . . . 12
  - 2.11.2 document order . . . 13
  - 2.11.3 name tokens . . . 13
  - 2.11.4 graph cycles . . . 13
  - 2.11.5 scheduling . . . 13

3 XLIM Type and Structure 14
- 3.1 Type System . . . 14
- 3.2 Structure of XLIM document . . . 14

A Loop Examples **16**
A.1 Basic Loop 16
A.1.1 Loop pseudo-code 16
A.1.2 Generated XLIM 16
A.2 Example of a Nested Loop 19
A.2.1 Nested loop pseudo-code 19
A.2.2 Generated XLIM 19

B If Example **24**
B.1 If/Then/Else 24
B.1.1 Conditional pseudo-code 24
B.1.2 Generated XLIM 24

C stateVar Examples **27**
C.1 Scalar 27
C.2 List 27
C.3 Multi-dimentional List 27

Chapter 1

Introduction

1.1 Scope/Context

This memo describes XLIM, an XML format for representing a language independent model of imperative programs.

A program is *imperative* if it specifies an algorithm as a sequence of statements that modify a collection of variables and memory—as opposed to, e.g., a *functional* or *declarative* description. Imperative programs typically contain assignments and control structures such as loops and conditional statements, in addition to expressions.

When building tools that process imperative code, such as compilers that generate software and hardware implementations from it, it becomes necessary to design a representation for that code that facilitates the relevant analyses and transformations. The source code itself is usually not very useful for this purpose, because many of the common processing steps involve the dependency structure among expressions and statements in the code. This dependency structure is only implicit in the original source, and needs to be established by matching variable names according to the scoping rules of the particular language.

The abstract syntax tree (AST) form of the code, which is usually generated by a parser, shares the same limitation. Even though it is more structural than the plain text of the code, the structures in it represent the lexical structure of the program, rather than the actual dependencies between its elements.

XLIM, in addition to maintaining information about the hierarchical lexical structure explicitly provided by the user, also directly represents the dependency relation between the elements of a program. It is essentially equivalent to a program in *static single assignment* (SSA) form, in which each variable is assigned to in only one place of the source.

1.2 Format Overview

XLIM is, firstly, an XML document containing a hierarchy of elements as defined in chapter 2. The elements which make up the document define the functionality being captured. *Operation* elements define atomic operators while *modules* provide for grouping and define control structures. As with any XML document, the inclusion of additional elements, not defined in this document, is allowed. Those tools/programs which consume XLIM may impose certain additional required data to be included in the document. The inclusion of additional elements in an XLIM document does not make that document malformed and should be tolerated by any consuming applications.

XLIM documents consist of a design containing 3 main types of elements. First, the design contains an enumeration of port elements. These define the way data is transferred to and from the functionality represented. Further, internal ports may be used for communication between parallel top-level (modules) functionality (Note that state variables may also be used to communicate between parallel top-level modules). Second, the design contains an enumeration of program state (state variables). These elements may be simple scalars or larger 'addressable' units. Third, the design contains one or more units capturing the functionality of the program. Each unit is a collection of *modules* and *operations* related by their hierarchy and connectivity (defined by attributes on the operations). A design may have any number of "ports", state elements, and parallel functional blocks.

Chapter 2

Element Definitions

This section describes the XML Elements which make up a well formed XLIM document. Each element is intended to represent various facets of the structure of an imperative program such as looping, branching, and scoping constructs as well as more abstract elements such as input/output mechanisms and stateful variables. The structure of the elements defines the 'control flow' of the program as child elements are only relevent within the context of their ancestor elements.

2.1 design

The *design* element is the root level element of an XLIM document and theremust be only one per document. The *design* element contains the children which define both the interface and functionality of a design captured in XLIM. The *design* element contains two categories of elements, those which contain the logic which defines the design, potentially consisting of multiple independent control domains/tasks/routines and those elements which represent resouces which are accessible by all of the control domains/tasks/routines.

The interface of an XLIM design is defined by *actor-port*(2.8) elements. State and persistent storage by *stateVar*(2.6) elements, and logic by hierarchies of logic contained within *module*(2.2) elements.

Invariants for the *design* element are:

1. The *design* element contains only *actor-port*, *internal-port*, *stateVar*, and *module* elements.
2. There must be only one Design element per XLIM document

2.2 module

Modules are the containment structure which defines hierarchy within the XLIM design. Modules contain operations and/or other modules to create a hierarchy of arbitrary depth. Further, module structures are used to define control flow within the design. Each module has a `kind` attribute defining both its structure (number and type of child nodes) and its behavioral properties. The basic module type (module with the `kind` attribute with the value 'block') is simply a container and represents a collection of logic.

Top-level modules are those *module* elements which are direct children of the *design* element. These modules define independent control domains for the algorithm. Conceptually these are similar to independent tasks, routines, or threads of execution. Communication between top-level modules is achieved through shared resources existing at the design level (eg *stateVar*(2.6) and *internalport*(2.9) elements).

Invariants common to all modules are:

1. contains operation, *module*, and *PHI* elements
2. has a `kind` attribute with value of: [block | mutex | if | then | else | loop | decision]

2.2.1 module [kind=block]

A block is simply a collection of operations or other modules. Top-level modules are blocks and contain an attribute which defines whether they are autostarting or depend on enablement from a taskCall(2.3) operation.

Invariants for the *block* module are:

1. top level block modules contain an `autostart` attribute

2.2.2 module [kind=if]

The *if* module has the following additional invariants:

1. contains one *decision* module and one *then* module and an optional *else* module
2. contains one *PHI* element for each generated value

An *if* module with no *else* block implies a structure which is equivalent to one with an empty *else* block. An *if* with no *else* block implies a structure where the PHI for each generated value has one input from the *then* block and one from the initial value of the modified value.

A generated value is any value modified within either the *then* or *else* block of the *if* module.

2.2.3 module [kind=loop]

A *loop* module represents an iteratively executed block of functionality. The functionality is iteratively executed so long as the decision module evaluates to a true condition. The decision module is evaluated prior to the first execution of the body block and again subsequent to an iteration and prior to the execution of the following iteration.

The *loop* module has the following additional invariants:

1. contains one *decision* module and one *block* module
2. contains one *PHI* element for each modified value

Result values from the loop are taken from the output of the PHI as the PHI always represents the most recently calculated value.

One input to the PHI is the initial value of the var the other is the modified value generated in the body of the loop.

2.2.4 module [kind=decision]

The *decision* module contains logic necessary to calculate a fork in the control path (execution) of the algorithm. An attribute is used to define the specific data value responsible for the final result of the decision. The significance of the result of the decision is dependent on the context of the decision module. (ie an *if* (2.2.2) or a *loop* (2.2.3) module)
The *decision* module has the following additional invariants:

1. module has a `decision` attribute
2. contains an operation with an output port whose `source` attribute value matches the module's `decision` attribute value

2.2.5 module [kind=mutex]

The behavior and structure of a *mutex* module is the same as that of a *block* module. However, the mutex block gives additional information to the implementation that there exist no dependencies between children (*operation* or *module*) for any stateful resource. ie, two child nodes accessing the same stateful resource can execute in any arbitrary order regardless of their use or modification of the stateful resource. However two accesses to the same stateful resource which are contained within the hierarchy of one child of that mutex block do maintain dependencies based on that stateful resource.

2.3 operation

An operation is the atomic unit of functionality in XLIM. The specific types and behavior of operations is implementation dependent with the exception of the standard operations listed below. The interface to an operation is defined by port elements that it contains. The execution of an operation is dependent on availability of its data and enablement based on its context (ancestor module hierarchy). There are no specific requirements on the specific implementation details of an operation (eg timing, latency, throughput, instruction count, etc). The implementing backend manages these considerations during compilation.

operation's have the following invariants:

1. contain only port elements as children
2. contain a `kind` attribute which defines its function
3. document order of child port elements may be significant based on the definintion of the operation's kind

The set of pre-defined operation element kinds for XLIM are:

- pinRead – Contain a `portname` attribute, the value of which is one of the actor-port or internal-port elements of the design. This operation has no input ports and one output port which carries the current value of the specified actor-port or internal-port. The pinRead operation immediately retrieves one valid (as defined by the implementing backend) value from the specified port and returns it via its output port.
- pinWrite – Contains a `portname` attribute, the value of which is one of the actor-port or internal-port elements of the design. This operation has one input port and no output ports. The input port is a value which is sent immediately to the specified actor-port or internal-port. The pinWrite operation immediately and reliably sends one value to the specified output port.
- assign – Contains a `target` attribute which is the name of one of the stateVar(2.6) elements of the design. There are two forms of the assign. If the stateVar is of scalar type (stateVar element contains a single initValue element) then there is a single input port which supplies the value to be assigned to the stateVar. If the stateVar is of complex type then the first input port (document order) is the address port and the second input port is the data port.
- taskCall – Contains a `target` attribute which is the name of one top level module (modules which are immediate children of the *design* module). The taskCall contains no input ports and no output ports. Functionally,

the taskCall activates the execution of a given top-level module. Scheduling against a taskCall is implementation dependent, but must ensure that shared resources are appropriately arbitrated among any simultaneously executing modules.

2.4 port

port elements are the endpoints of data dependency relationships within the graph. A port has a specific direction (in, out) to indicate the producer and consumer relationship. All ports are children of operation elements and have a `source` attribute. The `source` attribute of an output port must be unique across all output ports and defines the static single assignment (SSA) to that name. Any number of input ports may depend on that value by using the same name as the value for their `source` attribute.

The invariants of a port element are:

1. ports have a specific direction specified by their `dir` attribute
2. ports have exactly one `source` attribute specification.
3. output ports define the production of a value tied to the name specified in its `source` attribute.
4. there is exactly one output port or stateVar (2.6)for each name used in port `source` attribute fields
5. zero or more input ports may depend on a value by specifying the name of that value in their `source` attribute
6. there are no many-to-one relationships with ports

2.5 PHI element

The PHI element resolves multiple generated values to a single named token. Consequently, the PHI is used to merge many-to-one dependencies for ports such as occur when braches merge (eg from *if* module or feedback in a *loop* module). The behavior of a PHI is to select the most recently modified value on its input ports.

Invariants for the PHI are:

1. A PHI contains exactly 2 input ports and 1 output port as child elements
2. The PHI output is the most recently modified value of its two input ports
3. document order of the ports is insignificant

4. subsequent invocation of the PHI cannot happen until the current invocation is completed (output token dispatched).

2.6 stateVar

A stateVar represents persistent storage for the design that is accessible from any portion of the design. As such it is a shared resource available to the logic contained within any top-level module and accesses must be arbitrated accordingly by the impelementation. The actual implementation of a stateVar may be implementation dependent. stateVar elements have a specified initial value which is modified by execution of an assign operations (2.3). References to the stored value are achieved through symbolic reference to the name of the stateVar by an input port (2.4) via its `source` attribute.

Invariants for the stateVar are:

1. exists only as child of *design*
2. has a single initValue child (which may be scalar or complex)

2.7 initValue

initValue elements serve to specify exact numerical values for stateVars. The initValue element has a `typeName` attribute which is used to specify the type (type system is defined by the actual implementation of an XLIM backend) of the value. The "int" type is specified to indicate a scalar value. The "list" type is pre-defined in XLIM to indicate an ordered collection of distinctly addressable values. The actual storage layout and mechanism for addressing is implementation dependent, however within a given *list* the addressing is defined to be document order. *list* type initValues may be nested to arbitrary depth, however all 'leaf' elements must be of non-list type and must resolve to a numerical value.
No provision is made at this point for symbolic or 'pointer' type initial values.
Invariants for the initValue element are:

1. exist only as child of stateVar or other initValue elements
2. non-compound types have a specified `value` attribute
3. *list* types have no `value` attribute
4. compound types have 0 or more children

2.8 actor-port

An actor-port defines one element of the interface to the design. This is a conceptual port and may be implemented as simple I/O or a port requiring a more complex protocol. Accesses to the port are achieved by pinRead(2.3) and pinWrite(2.3) operations which are reliable operation (values are valid and not lost).

2.9 internal-port

An internal-port defines a communication mechanism between two top-level modules. The internal-port specifies the communication channel and is accessed by pinRead(2.3) and pinWrite(2.3) operations, both of which are reliable (values are valid and not lost). The specific implementation of an inernal-port is impelementation dependent but must ensure that data/messages transmitted are never lost and that the listener always receives a valid data token/message.

2.10 Finite State Machines

This section and XLIM structure is TBD...

It is anticipated that XLIM will be augmented with an FSM structure in the near future. The exact syntax and content of the FSM element will be defined in a future revision of this document.

2.11 other invariants

There are a number of specific details which help to define the structure and functionality represented by an XLIM document but which cannot be directly tied to a specific element. These are detailed here.

2.11.1 dependencies

The data dependencies represented by an output port and input port pair (or stateVar/input port pair) sharing the same value for their `source` attribute are conceptually implemented by a queue or FIFO. Data produced by the output port is sent to the input port in an ordered manner and that data leaves the queue only through consumption by the input port (eg via execution of the operation). However, this does not restrict the backend consumer to a particular method of implementation. Generally it is the case that these dependencies resolve to simple (non-stateful) data communication minimal control/overhead to ensure the values are valid and not lost.

2.11.2 document order

In general, document order does not have any effect on the functionality specified in XLIM except as noted for specific elements above. eg, the *then* and *else* block of *if* module may occur in either order, howeve the port order of a subtract operation may be significant.

2.11.3 name tokens

Name tokens are globally unique. Name tokens may appear as the `name` attribute of a *module*, stateVar, actor-port, or internal-port. Similarly, name tokens specified as the source of an output port are also globally unique. Name tokens should be considered to be case sensitive.

2.11.4 graph cycles

The only place cycles are allowed is in the path from a loop body to the input of a PHI.

2.11.5 scheduling

The way in which the functionality of an algorithm specified in XLIM is implemented is largely impelementation dependent. In particular, the way in which the operations are scheduled (ordered for execution) can be achieved in many ways. However, the following invariants (derived from the invariants of elements above) must hold:

1. The graph is acyclic except for cycles that are broken by merges at PHI elements.
2. At runtime a next token may not be received for processing at a PHI port until the current token has been processed and dispatched.

These two conditions (intuitively) indicate that XLIM is scheduling invariant. Meaning that regardless of how it is scheduled the resulting functionality will be the same.

Chapter 3

XLIM Type and Structure

3.1 Type System

XLIM allows for a flexible type system to be used by allowing attributes for typename (eg int or bool) and size (precision) on each port element within the graph. These attributes are arbitrary and in their absence it is up to the particular XLIM consumer implementation to define a type system.

In one implementation, the default type system is as follows. Integer values are represented with a type int. This type is a signed (two's complement) value whose precision is determined by the size attribute thus ranging in value from -2^{size-1} to $2^{size-1}-1$. Unsigned values are implemented by increasing the size by one bit to allow for a leading 0. Boolean values are represented with a type bool. This type is unsigned and allowable values are 1 (true) and 0 (false).

For consistency of the graph it is recommended that all operation output ports have an explicitly defined type and size. For edges that connect ports of different type and/or size it is recommended that a cast operation be used to explicitly define the transition in type or size.

3.2 Structure of XLIM document

```
<design>
<design attribute/>
<actorPort/>
...
<internalPort/>
...
<stateVar>
<initValue/>
</stateVar>
...
<module>
```

```
<operation>
<port/>
<operation/>
...

<module>
<port/>
<operation/>
...
</module>
...

</module>
...
</design>
```

Appendix A

Loop Examples

A.1 Basic Loop

Note that this loop contains all 3 methods of variable consuption/production:

- Variable consumed, not modified (variable `a`)
- Variable consumed and modified (variable `i`)
- Variable consumed, modified, and used again outside loop (variable `b`)

Take the following example:

A.1.1 Loop pseudo-code

```
int a = actorPortRead;
int b = 9;
int i = 0;
while ( i < 7 )
{
  i = i + a;
  b++;
}
actorPortWrite(b);
```

A.1.2 Generated XLIM

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<design name="Loop">
  <actor-port dir="in" name="A" size="32" typeName="int"/>
  <actor-port dir="out" name="B" size="32" typeName="int"/>
  <module kind="block" name="actionAtLine_4">
```

```
<operation kind="pinRead" portName="A" removable="no">
  <port dir="out" size="32" source="0x_5b1b1" typeName="int"/>
</operation>
<operation kind="$literal_Integer" value="9">
  <port dir="out" size="5" source="1x_3b3b3" typeName="int"/>
</operation>
<operation kind="cast">
  <port dir="in" source="1x_3b3b3"/>
  <port dir="out" size="32" source="0x_5b5" typeName="int"/>
</operation>
<operation kind="$literal_Integer" value="0">
  <port dir="out" size="1" source="1x_3b5b3" typeName="int"/>
</operation>
<operation kind="cast">
  <port dir="in" source="1x_3b5b3"/>
  <port dir="out" size="32" source="0x_5b7" typeName="int"/>
</operation>
<module kind="loop">
  <PHI>
    <port dir="in" source="0x_5b7"/>
    <port dir="in" source="0x_5b9b3b1"/>
    <port dir="out" size="32" source="0x_5b9$PHI$0x_5b7" typeName="int"/>
  </PHI>
  <PHI>
    <port dir="in" source="0x_5b5"/>
    <port dir="in" source="0x_5b9b3b3"/>
    <port dir="out" size="32" source="0x_5b9$PHI$0x_5b5" typeName="int"/>
  </PHI>
  <module decision="1x_3b7b1" kind="test">
    <operation kind="noop">
      <port dir="in" source="0x_5b9$PHI$0x_5b7"/>
      <port dir="out" size="32" source="0x_5b9b1b1" typeName="int"/>
    </operation>
    <operation kind="$literal_Integer" value="7">
      <port dir="out" size="4" source="1x_3b7b1b3b3" typeName="int"/>
    </operation>
    <operation kind="$lt">
      <port dir="in" source="0x_5b9b1b1"/>
      <port dir="in" source="1x_3b7b1b3b3"/>
      <port dir="out" size="1" source="1x_3b7b1" typeName="bool"/>
    </operation>
  </module>
  <module kind="block">
    <operation kind="noop">
      <port dir="in" source="0x_5b9$PHI$0x_5b7"/>
      <port dir="out" size="32" source="0x_5b9b3b1b1b1" typeName="int"/>
```

```
        </operation>
        <operation kind="noop">
          <port dir="in" source="0x_5b1b1"/>
          <port dir="out" size="32" source="0x_5b9b3b1b1b5" typeName="int"/>
        </operation>
        <operation kind="$add">
          <port dir="in" source="0x_5b9b3b1b1b1"/>
          <port dir="in" source="0x_5b9b3b1b1b5"/>
          <port dir="out" size="33" source="1x_3b7b3b1b1" typeName="int"/>
        </operation>
        <operation kind="noop">
          <port dir="in" source="1x_3b7b3b1b1"/>
          <port dir="out" size="32" source="0x_5b9b3b1" typeName="int"/>
        </operation>
        <operation kind="noop">
          <port dir="in" source="0x_5b9$PHI$0x_5b5"/>
          <port dir="out" size="32" source="0x_5b9b3b3b1b1" typeName="int"/>
        </operation>
        <operation kind="$literal_Integer" value="1">
          <port dir="out" size="2" source="1x_3b7b3b3b1b3b3" typeName="int"/>
        </operation>
        <operation kind="$add">
          <port dir="in" source="0x_5b9b3b3b1b1"/>
          <port dir="in" source="1x_3b7b3b3b1b3b3"/>
          <port dir="out" size="33" source="1x_3b7b3b3b1" typeName="int"/>
        </operation>
        <operation kind="noop">
          <port dir="in" source="1x_3b7b3b3b1"/>
          <port dir="out" size="32" source="0x_5b9b3b3" typeName="int"/>
        </operation>
      </module>
    </module>
    <operation kind="noop">
      <port dir="in" source="0x_5b9$PHI$0x_5b5"/>
      <port dir="out" size="32" source="0x_5b3b1" typeName="int"/>
    </operation>
    <operation kind="pinWrite" style="simple" portName="B">
      <port dir="in" source="0x_5b3b1"/>
    </operation>
  </module>
</design>
```

A.2 Example of a Nested Loop

Loop elements may be nested to arbitrary depth. However, references to symbols generated in loop elements from outside any loop hierarchy must refer to a symbol defined at the highest level of the loop hierarchy. Take the following example:

A.2.1 Nested loop pseudo-code

```
int a = actorPortRead;
int b = 9;
int i = 0;
int j = 0;
while ( i < 7 )
{
  i = i + a;
  j = 0;
  while (j < 3)
  {
    j++;
    b++;
  }
}
actorPortWrite(b);
```

A.2.2 Generated XLIM

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<design name="NestedLoop">
  <actor-port dir="in" name="A" size="32" typeName="int"/>
  <actor-port dir="out" name="B" size="32" typeName="int"/>
  <module kind="block" name="actionAtLine_4">
    <operation kind="pinRead" portName="A" removable="no">
      <port dir="out" size="32" source="0x_7b1b1" typeName="int"/>
    </operation>
    <operation kind="$literal_Integer" value="9">
      <port dir="out" size="5" source="1x_4b3b3" typeName="int"/>
    </operation>
    <operation kind="cast">
      <port dir="in" source="1x_4b3b3"/>
      <port dir="out" size="32" source="0x_7b5" typeName="int"/>
    </operation>
    <operation kind="$literal_Integer" value="0">
      <port dir="out" size="1" source="1x_4b5b3" typeName="int"/>
    </operation>
    <operation kind="cast">
```

```
  <port dir="in" source="1x_4b5b3"/>
  <port dir="out" size="32" source="0x_7b7" typeName="int"/>
</operation>
<operation kind="$literal_Integer" value="0">
  <port dir="out" size="1" source="1x_4b7b3" typeName="int"/>
</operation>
<operation kind="cast">
  <port dir="in" source="1x_4b7b3"/>
  <port dir="out" size="32" source="0x_7b9" typeName="int"/>
</operation>
<module kind="loop">
  <PHI>
    <port dir="in" source="0x_7b7"/>
    <port dir="in" source="y_b3b1"/>
    <port dir="out" size="32" source="y_$PHI$0x_7b7" typeName="int"/>
  </PHI>
  <PHI>
    <port dir="in" source="0x_7b9"/>
    <port dir="in" source="y_b3b5$PHI$0x_7b9"/>
    <port dir="out" size="32" source="y_$PHI$0x_7b9" typeName="int"/>
  </PHI>
  <PHI>
    <port dir="in" source="0x_7b5"/>
    <port dir="in" source="y_b3b5$PHI$0x_7b5"/>
    <port dir="out" size="32" source="y_$PHI$0x_7b5" typeName="int"/>
  </PHI>
  <module decision="1x_4b9b1" kind="test">
    <operation kind="noop">
      <port dir="in" source="y_$PHI$0x_7b7"/>
      <port dir="out" size="32" source="y_b1b1" typeName="int"/>
    </operation>
    <operation kind="$literal_Integer" value="7">
      <port dir="out" size="4" source="1x_4b9b1b3b3" typeName="int"/>
    </operation>
    <operation kind="$lt">
      <port dir="in" source="y_b1b1"/>
      <port dir="in" source="1x_4b9b1b3b3"/>
      <port dir="out" size="1" source="1x_4b9b1" typeName="bool"/>
    </operation>
  </module>
  <module kind="block">
    <operation kind="noop">
      <port dir="in" source="y_$PHI$0x_7b7"/>
      <port dir="out" size="32" source="y_b3b1b1b1" typeName="int"/>
    </operation>
    <operation kind="noop">
```

```
  <port dir="in" source="0x_7b1b1"/>
  <port dir="out" size="32" source="y_b3b1b1b5" typeName="int"/>
</operation>
<operation kind="$add">
  <port dir="in" source="y_b3b1b1b1"/>
  <port dir="in" source="y_b3b1b1b5"/>
  <port dir="out" size="33" source="1x_4b9b3b1b1" typeName="int"/>
</operation>
<operation kind="noop">
  <port dir="in" source="1x_4b9b3b1b1"/>
  <port dir="out" size="32" source="y_b3b1" typeName="int"/>
</operation>
<operation kind="$literal_Integer" value="0">
  <port dir="out" size="1" source="1x_4b9b3b3b1" typeName="int"/>
</operation>
<operation kind="noop">
  <port dir="in" source="1x_4b9b3b3b1"/>
  <port dir="out" size="32" source="y_b3b3" typeName="int"/>
</operation>
<module kind="loop">
  <PHI>
    <port dir="in" source="y_b3b3"/>
    <port dir="in" source="y_z_b1"/>
    <port dir="out" size="32" source="y_b3b5$PHI$0x_7b9" typeName="int"/>
  </PHI>
  <PHI>
    <port dir="in" source="y_$PHI$0x_7b5"/>
    <port dir="in" source="y_z_b3"/>
    <port dir="out" size="32" source="y_b3b5$PHI$0x_7b5" typeName="int"/>
  </PHI>
  <module decision="1x_4b9b3b5b1" kind="test">
    <operation kind="noop">
      <port dir="in" source="y_b3b5$PHI$0x_7b9"/>
      <port dir="out" size="32" source="y_b3b5b1b1" typeName="int"/>
    </operation>
    <operation kind="$literal_Integer" value="3">
      <port dir="out" size="3" source="1x_4b9b3b5b1b3b3" typeName="int"/>
    </operation>
    <operation kind="$lt">
      <port dir="in" source="y_b3b5b1b1"/>
      <port dir="in" source="1x_4b9b3b5b1b3b3"/>
      <port dir="out" size="1" source="1x_4b9b3b5b1" typeName="bool"/>
    </operation>
  </module>
  <module kind="block">
    <operation kind="noop">
```

```
            <port dir="in" source="y_b3b5$PHI$0x_7b9"/>
            <port dir="out" size="32" source="y_z_b1b1b1" typeName="int"/>
          </operation>
          <operation kind="$literal_Integer" value="1">
            <port dir="out" size="2" source="1x_4b9z_b1b1b3b3" typeName="int"/>
          </operation>
          <operation kind="$add">
            <port dir="in" source="y_z_b1b1b1"/>
            <port dir="in" source="1x_4b9z_b1b1b3b3"/>
            <port dir="out" size="33" source="1x_4b9z_b1b1" typeName="int"/>
          </operation>
          <operation kind="noop">
            <port dir="in" source="1x_4b9z_b1b1"/>
            <port dir="out" size="32" source="y_z_b1" typeName="int"/>
          </operation>
          <operation kind="noop">
            <port dir="in" source="y_b3b5$PHI$0x_7b5"/>
            <port dir="out" size="32" source="y_z_b3b1b1" typeName="int"/>
          </operation>
          <operation kind="$literal_Integer" value="1">
            <port dir="out" size="2" source="1x_4b9z_b3b1b3b3" typeName="int"/>
          </operation>
          <operation kind="$add">
            <port dir="in" source="y_z_b3b1b1"/>
            <port dir="in" source="1x_4b9z_b3b1b3b3"/>
            <port dir="out" size="33" source="1x_4b9z_b3b1" typeName="int"/>
          </operation>
          <operation kind="noop">
            <port dir="in" source="1x_4b9z_b3b1"/>
            <port dir="out" size="32" source="y_z_b3" typeName="int"/>
          </operation>
        </module>
      </module>
    </module>
  </module>

  <operation kind="noop">
    <port dir="in" source="y_$PHI$0x_7b5"/>
    <port dir="out" size="32" source="0x_7b3b1" typeName="int"/>
  </operation>
  <operation kind="pinWrite" style="simple" portName="B">
    <port dir="in" source="0x_7b3b1"/>
  </operation>
 </module>
</design>
```

Note that the pinWrite element is using the variable b. The value for this variable is calculated in the inner loop. However, to obtain the correct value for b, the symbolic reference must be to the merged value (the PHI output which is `y_$PHI$0x_7b5`).

Appendix B

If Example

B.1 If/Then/Else

The example below shows an if/then/else structure.

B.1.1 Conditional pseudo-code

```
int a = portRead(A);
int b = portRead(B);
int x = a;
int y = b;
if (a > b)
{
  x = a + 1;
  y = y * 2;
}
else
{
  x = a / 2;
}
portWrite(O, x - y);
```

Note that in the example the variable `y` is only modified in the *then* branch of the statement. To account for the unmodified value of `y` through this block, a `noop` operation is inserted into the *else* branch in order to carry the unmodified value of `y`. Thus the PHI element which produces the result of `y` merges the output of the multiply and the noop.

B.1.2 Generated XLIM

```
<?xml version="1.0" encoding="UTF-8"?>
<design name="IfTest">
```

```
<actor-port dir="in" name="A" size="32" typeName="int"/>
<actor-port dir="in" name="B" size="32" typeName="int"/>
<actor-port dir="out" name="O" size="32" typeName="int"/>
<module autostart="false" kind="action" name="actionAtLine_3">
  <operation kind="pinRead" portName="A" removable="no">
    <port dir="out" size="32" source="z7b1b1" typeName="int"/>
  </operation>
  <operation kind="pinRead" portName="B" removable="no">
    <port dir="out" size="32" source="z7b3b1" typeName="int"/>
  </operation>

  <module kind="if">
    <module decision="1$id$y5b1" kind="test">
      <operation kind="$gt">
        <port dir="in" source="z7b1b1"/>
        <port dir="in" source="z7b3b1"/>
        <port dir="out" size="1" source="1$id$y5b1" typeName="bool"/>
      </operation>
    </module>
    <module kind="then">
      <operation kind="$literal_Integer" value="1">
        <port dir="out" size="2" source="1$id$y5b3b1b1b3b3" typeName="int"/>
      </operation>
      <operation kind="$add">
        <port dir="in" source="z7b1b1"/>
        <port dir="in" source="1$id$y5b3b1b1b3b3"/>
        <port dir="out" size="33" source="1$id$y5b3b1b1" typeName="int"/>
      </operation>
      <operation kind="$literal_Integer" value="2">
        <port dir="out" size="3" source="1$id$y5b3b3b1b3b3" typeName="int"/>
      </operation>
      <operation kind="$mul">
        <port dir="in" source="z7b3b1"/>
        <port dir="in" source="1$id$y5b3b3b1b3b3"/>
        <port dir="out" size="35" source="1$id$y5b3b3b1" typeName="int"/>
      </operation>
    </module>
    <module kind="else">
      <operation kind="$literal_Integer" value="2">
        <port dir="out" size="3" source="1$id$y5b5b1b1b3b3" typeName="int"/>
      </operation>
      <operation kind="$div">
        <port dir="in" source="z7b1b1"/>
        <port dir="in" source="1$id$y5b5b1b1b3b3"/>
        <port dir="out" size="32" source="1$id$y5b5b1b1" typeName="int"/>
      </operation>
```

```
        <operation kind="noop">
          <port dir="in" source="z7b3b1"/>
          <port dir="out" size="32" source="z7c17$ELSE$z7b9" typeName="int"/>
        </operation>
      </module>
      <PHI>
        <port dir="in" qualifier="then" source="1$id$y5b3b1b1"/>
        <port dir="in" qualifier="else" source="1$id$y5b5b1b1"/>
        <port dir="out" size="32" source="z7c17$PHI$z7b7" typeName="int"/>
      </PHI>
      <PHI>
        <port dir="in" qualifier="then" source="1$id$y5b3b3b1"/>
        <port dir="in" qualifier="else" source="z7c17$ELSE$z7b9"/>
        <port dir="out" size="32" source="z7c17$PHI$z7b9" typeName="int"/>
      </PHI>
    </module>
    <operation kind="$sub">
      <port dir="in" source="z7c17$PHI$z7b7"/>
      <port dir="in" source="z7c17$PHI$z7b9"/>
      <port dir="out" size="33" source="1$id$y7b1" typeName="int"/>
    </operation>
    <operation kind="pinWrite" portName="O" style="simple">
      <port dir="in" source="1$id$y7b1"/>
    </operation>
  </module>

</design>
```

Appendix C

stateVar Examples

C.1 Scalar

Scalar type variable

```
<stateVar name="myScalar">
  <initValue type="int" size="12" value="0"/>
</stateVar>
```

C.2 List

List type variable

```
<stateVar name="myList">
  <initValue type="int" size="12" value="0"/>
  <initValue type="int" size="12" value="1"/>
  <initValue type="int" size="12" value="2"/>
  <initValue type="int" size="12" value="3"/>
</stateVar>
```

C.3 Multi-dimentional List

2-d list type variable

```
<stateVar name="myList">
  <initValue type="list">
    <initValue type="int" size="12" value="1"/>
    <initValue type="int" size="12" value="2"/>
  </initValue>
  <initValue type="list">
    <initValue type="int" size="12" value="10"/>
```

```
    <initValue type="int" size="12" value="20"/>
  </initValue>
</stateVar>
```