Improving the FPGA Design Process Through Determining and Applying Logical-to-Physical Design Mappings *

Paul Graham, Brad Hutchings, and Brent Nelson
Department of Electrical and Computer Engineering
Brigham Young University
459 CB, Provo, UT 84602
grahamp@ee.byu.edu, hutch@ee.byu.edu, nelson@ee.byu.edu

Abstract

In this paper we discuss several possible uses of the knowledge of how user's logical designs are mapped to physical FPGA circuits. Some of these uses include power analyses, useful feedback on physical design implementations, and direct, quick modifications of physical designs. As an example of how this knowledge can be used, we describe, in detail, how to determine the logical-to-physical mapping of Xilinx XC4000 circuits created with JHDL and how this mapping and FPGA state sampling, or readback, enables us to provide a hardware debugging environment with complete visibility of all flip-flops and LUT RAMs in executing hardware.

1 Introduction

In supporting hardware debugging in the JHDL [1, 2] design environment, we have found that knowing how design elements from the user's logical design were mapped to their counterparts in the FPGA physical implementation is quite important. With only a partial mapping from the logical to the physical, we would not be able to provide users of JHDL with a complete view of what their circuit is doing during hardware execution via FPGAs' readback mechanism [3, 4]. Beyond the applications of debugging, this same knowledge can also be used to provide the designer with a more detailed and understandable view of a circuit once physically implemented and contribute to improved and more understandable analyses of design characteristics such as dynamic power consumption and critical path identification.

*Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-1-0222. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

As a demonstration of how useful this mapping information is, this paper will describe the process of determining the logical-to-physical mappings for XC4000 circuits implemented with the Xilinx Alliance and Foundation software tools and how this mapping is applied to create a board-independent, device-specific mechanism for debugging hardware in the JHDL environment. As background, we will briefly review the role of readback in debugging FPGA circuit designs and why determining the logical-to-physical mapping is important for debugging hardware with readback. Next, we will discuss the portion of the JHDL design flow which applies to hardware debugging, namely, circuit netlist creation, the processing of Xilinx report files, and the creation of a JHDL readback symbol table. Following this, we will discuss how board models interact with and use the JHDL readback API as well as how FPGA vendor software can improve its support for third-party tools which need to determine logical-to-physical design mappings. Lastly, we will describe other possible uses of this mapping information and summarize several of the lessons learned from this work.

2 Background

After a designer describes a circuit using a hardware description language (HDL) or a schematic, the description is usually simulated to verify, to some degree, that the designed circuit operates as expected. Design verification through simulation, though, can be very time consuming, especially, when design errors may not be discovered until millions or hundreds of millions of clock cycles have been executed; the process can sometimes take hours, days, or longer. For FPGA-based configurable computing machines (CCMs), hardware debugging environments can be used to find and debug the problems much more quickly– millions or hundreds of millions of cycles may only require seconds or minutes of execution. Besides the speed bene-

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
2000		**00-00-2000 to 00-00-2000**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Improving the FPGA Design Process Through Determining and Applying Logical-to-Physical Design Mappings	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Brigham Young University,Department of Electrical and Computer Engineering,Provo,UT,84602**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**11**	

fits, hardware debugging environments can also provide a greater degree of realism during circuit validation.

One functionality found in many modern FPGAs which greatly aids the hardware debugging of CCM applications is the ability to sample, or *readback*, the state of the FPGA, including the values of flip-flops and RAM contents. For example, with the Xilinx XC4000 and Virtex families of FPGAs, the values of all user flip-flops, RAMs (LUT and block), and the combinational outputs of IOBs and CLBs are available for inspection through readback [3, 4].

Because of the excellent visibility that readback provides into the operation of circuits, many CCM platforms have either employed the FPGA readback capability in a debugging environment or provided at least an application programmer's interface (API) for accessing the readback state of the FPGAs. For instance, the Splash 2 run-time environment [5] provided both an API as well as a run-time debugging tool named *t2* which could be used to execute and debug a design. The tool provided a facility to watch the values of signals as designs' clocks were stepped—a functionality utilizing the Xilinx XC4000's readback capability. As another example, the run-time libraries provided with Annapolis Micro System's Wildforce boards [6] have several readback-related API calls for creating and manipulating readback symbol tables as well as for retrieving the readback data itself.

In our experiences using Splash 2 and Wildforce, the readback symbol table mechanisms have been very useful but also limited. At times, signal names would be modified as the Xilinx tools would optimize the designs, meaning that we would have to go searching through the known readback symbols to find the new names of the signals we needed. In other words, the users themselves had to determine the logical-to-physical mapping for some signal names due to the run-time software's incomplete knowledge of designs' logical-to-physical mappings.

This limitation is related to the amount of extra work required to resolve the logical-to-physical mappings in the VHDL or Verilog design environments for CCMs. To provide a comprehensive hardware debugging system for a CCM run-time system using traditional HDL tools, three mappings must be determined. First, the synthesis mapping of the HDL design to an FPGA vendor's library must be understood, which probably involves parsing the netlists resulting from synthesis and comparing the netlist with the HDL design. This task is not easy when the synthesis tool performs many optimizations. Second, a correlation must be made between the netlist of the synthesized design and the physical form created by the FPGA vendor's implementation tools. The final mapping from the original HDL design to the physical implementation can be derived from the first two mappings. Even without the synthesis step, the relationship between a structural user design and the physical FPGA implementation is not easy in the presence of design optimizations made by FPGA vendors' software.

Instead of determining these mappings, CCM run-time software generally uses net names as a means of relating the results of an FPGA readback to the current state of a user's design. As an example, early JHDL readback implementations used this approach. JHDL would extract net name information from the *map* report file (`.mrp` file) and the logical allocation file (`.ll` file) created by the Xilinx FPGA implementation tools. The `.mrp` file records how the circuit is mapped to FPGA resources and how it was optimized during technology mapping. The `.ll` file, on the other hand, reports the readback bitstream locations of the outputs sampled from flip-flops, CLBs, and IOBs as well as the locations of block and LUT RAM contents in the readback bitstream. Entries in the `.ll` file for the sampled outputs of CLBs, IOBs, and flip-flops are annotated with the name of the nets attached to those outputs in the physical circuit. Thus, assuming a certain net is connected to the output of a flip-flop both in the logical design and its physical implementation, a correlation between the logical and physical designs can be made. Unfortunately, using these two files alone in making the association between the logical and physical can be problematic for several reasons:

- Due to circuit optimizations and hierarchical net name flattening, it is not always simple or even possible to relate net names from the original, logical design to those found in the final, physical design based only on the information from these files. The `.mrp` file reports design changes, including net name changes, resulting from optimizations performed during technology mapping, but name flattening for nets crossing hierarchical boundaries is not reported. This makes the process of tracking net name changes difficult, if not impossible, without parsing netlists or otherwise knowing the exact structure of the original circuit.

- RAM entries in the `.ll` file are associated with physical locations on the FPGA alone and not with any net names. Though the outputs of the CLB are associated with net names, there are several cases where it is not possible to determine which output of the CLB corresponds to which LUT RAM based on the information of these two files. Thus, the net name association method fails occasionally when associating physical LUT RAMs to their logical counterparts.

- As with other uses of LUTs, LUT RAMs can have their address pins permuted by the place and route tools. This makes relating logical and physical LUT RAM contents a problem since the `.ll` and `.mrp` files have no information regarding how these inputs

to the RAMs were permuted. As far as we know, this information is only available in the placed and routed Native Circuit Description, or `.ncd`, file and a few derived files.

Considering these short-comings, it is no surprise that most CCM run-time software packages do not support the readback of LUT RAMs. Clearly, this methodology of relating logical design elements to physical circuit elements cannot be 100% accurate or complete and another approach is needed.

Another difficulty of many CCM hardware debugging environments is that the design, simulation, and execution environments are different and, potentially, unrelated. For instance, in some CCM synthesis environments, the simulator and synthesis tools are separate and provided by different CAD tool vendors. As a result of this and the limited logical-to-physical mapping used by many CCM hardware debugging environments, some correspondences between the HDL design and its physical implementation must be made manually by an experienced designer.

3 JHDL and Hardware Execution

As discussed in [2], the JHDL design, simulation, and execution environment provides a unified view of a design's simulation and hardware execution—they look the same in the various design views. During simulation, the simulator itself is responsible for keeping track of the values and state of the circuit elements in a design. To make this unified view possible in hardware mode, the sampled state of the FPGAs from readback is used to recreate the entire state of the JHDL design within the simulator.

In striving for this unified view of designs, we have developed a reasonable solution to relating both LUT RAM and flip-flop state in the readback bitstream to the user's design. This was simplified by several factors relating to both JHDL and the Xilinx FPGA implementation software. JHDL designs are structural in nature rather than synthesized; this means that at netlist time a complete description of the circuit is known. As for the Xilinx software, the instance names placed in EDIF netlists are preserved all the way through the design flow, even down to the placed-and-routed design found in the `.ncd` file. These two features allow a simple one-to-one correspondence to be made between circuit elements at the logical and physical levels.

The next few sections of the paper will describe the process of how we take advantage of these features to correlate *all* the state provided by readback to their corresponding elements in a XC4000-based design. Despite the fact that our discussion involves JHDL most, if not all, of the techniques we will describe can be used in other FPGA design methodologies.

4 Creating Designs and Netlists

As discussed in [1, 2], a JHDL user design is described in Java using libraries from the JHDL environment. The circuit can be described using hierarchy and generally involves the instantiation of design primitives which can be either from a technology-specific or technology-independent library. Technology-specific library elements directly implement the library elements found in an FPGA vendor's design libraries, while the technology-independent library elements from the `byucc.jhdl.Logic` package instance technology-specific library elements via the JHDL Techmapper API, which maps the generic library elements of the `byucc.jhdl.Logic` library to the corresponding library elements of a specific FPGA technology. In addition to library primitives, a designer can use parameterizable module generators which create optimized circuits based on library primitives; commonly used module generators include adders, multipliers, and counters.

As a result of this structural design methodology employed in JHDL, we know *a priori* how the design will map to device primitives. With this knowledge and the fact that instance names are preserved by the Xilinx software, net names are no longer needed as the primary means of relating readback information to the user's design. Instead, the instance names of flip-flops and RAMs themselves can be used to make the correlation between readback information and the design.

When the design is ready to be implemented using the Xilinx FPGA tools, the JHDL board models create both an EDIF netlist of the circuit and a second netlist consisting only of circuit elements whose state can be sampled. This second netlist, called an `.rbsym` file, records the instance name and the library element name for each circuit element as well as the names of its input and output ports and the nets attached to these ports. As far as port and net names are concerned, only the input ports and net names for LUT RAMs are actually needed, as we will demonstrate later.

Figure 1 provides sample `.rbsym` entries for an input IOB flip-flop (ifdx) and a synchronous 16x1 LUT RAM.

5 Creating Symbol Tables

The most difficult task in making hardware execution look like simulation to the user is the creation of a symbol table which relates a JHDL circuit element to its state information found in the readback bitstream. The

```
/PE1/IOB_Left/ifd-0/ifdx-0 byucc.jhdl.Xilinx.XC4000.ifdx  out q /PE1/IOB_Left/ifd-0/q
/PE1/LogicCore/ram16x1s-0 byucc.jhdl.Xilinx.ram16x1s  in a<0> /PE1/LogicCore/Count1BufIn<0>
  in a<1> /PE1/LogicCore/Count1BufIn<1>  in a<2> /PE1/LogicCore/Count1BufIn<2>
  in a<3> /PE1/LogicCore/Count1BufIn<3>  out o /PE1/LogicCore/RAMOut
```

Figure 1: Sample entries from an `.rbsym` file

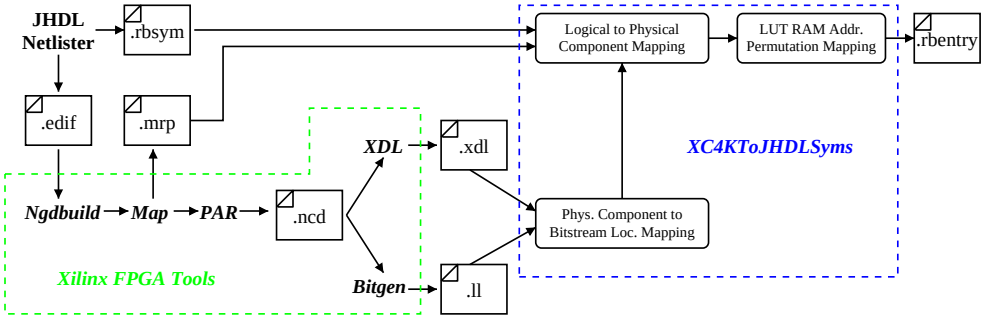

Figure 2: Process of Physical-To-Logical Mapping

`XC4KToJHDLSyms` class from the JHDL readback API performs this task using the information from the `.rbsym` file and three different types of Xilinx files: `.ll` files, `.mrp` files, and `.xdl` files. The `.xdl` file used in our methodology is a textual description of the final, physical FPGA design and is equivalent to the `.ncd` file of the placed and routed design; the file itself is written in the Xilinx Design Language (XDL). Below we will describe the important information provided by each of these files and their roles in constructing the final readback symbol table for hardware execution, or `.rbentry` file. Following these descriptions, we describe the actual process of determining the mapping between logical circuit elements and their physical implementations when generating an `.rbentry` file. For convenience, Table 1 provides a summary of the files used in the process and Figure 2 provides a graphical representation of the process.

5.1 Determining Readback Bitstream Locations

One type of information needed to create the `.rbentry` file is a mapping between a LUT or flip-flop at a given physical location on the FPGA and the offset for its state in the readback bitstream. As discussed earlier, the `.ll` file provides exactly this information. Additionally, the net names that are associated with the outputs of flip-flops, CLBs, and IOBs in the file can now be ignored since we can develop a one-to-one mapping between JHDL circuit elements and physical FPGA entities based on instance names.

Figure 3 provides some sample entries from an XC4062XL design's `.ll` file. The entries are for the output flip-flop of Pad P81's IOB, an FFX flip-flop in the CLB located at R47C12, and the *F* LUT RAM bit with address 15 from the CLB located at R48C12. The first number in each entry is the absolute offset of the state bit in the readback bitstream while the following two numbers correspond to the bit's frame number and frame offset. The absolute offset is related to the frame number and frame offset in the following manner for the XC4000 family of FPGAs:

$$\begin{aligned} offset_{abs} = \\ (frame\_size - start\_bits - stop\_bits) \\ \cdot frame\_number - offset_{frame} \end{aligned} \quad (1)$$

where $offset_{abs}$ is the absolute offset, $frame\_size$ is the size of the configuration frame in bits for the XC4000 family FPGA being used, $start\_bits$ and $stop\_bits$ correspond to the number of start bits and stop bits per configuration frame, and $frame\_number$ and $offset_{frame}$ are, respectively, the frame number and frame offset provided by the `.ll` file. For example, in the first `.ll` entry provided in Figure 3, the $frame\_size$ for a XC4062XL is 613 bits [7] while the number of frame start and stop bits are 1 and 4, respectively. Thus, the absolute offset is determined as: $(613 - 1 - 4) \cdot 1624 - 605 = 986787$.

As Eq. 1 demonstrates, the absolute offset does not account for the start and stop bits for frames. Further, through trial and error, we determined that the offset also ignores any header information before the first frame. So, when a readback bitstream is obtained, we must account for non-data bits in the bitstream when given an absolute offset from the `.ll` file. The following equation provides this

Name	Creator	Description
`.rbsym`	JHDL	Netlist of logical circuit elements for readback
`.ll`	Xilinx, `bitgen -l`	Lists the locations of sampled FPGA state in the readback bitstream
`.mrp`	Xilinx, `map`	Reports design's mapping to FPGA resources and optimizations
`.ncd`	Xilinx, `par`	Binary, closed description of a design's FPGA physical implementation
`.xdl`	Xilinx, `xdl`	Textual, open description of a design's FPGA physical implementation
`.rbentry`	JHDL, `XC4KToJHDLSyms`	Relates logical design elements to locations in bitstream

Table 1: Summary of Files Used for Creating Readback Symbol Tables

```
...
Bit   986787   1624    605 Block=P81 Latch=OQ Net=PE_Right_Out<8>
...
Bit  1062819   1749    573 Block=CLB_R47C12 Latch=XQ Net=LogicCore/Count1BufIn<0>
...
Bit  1071314   1763    590 Block=CLB_R48C12 Ram=F:15
...
```

Figure 3: Sample Entries from an `.ll` File

mapping:

$$\begin{aligned} offset_{actual} = offset_{abs} \\ +(\frac{offset_{abs}}{frame\_size - start\_bits - stop\_bits}) \\ \cdot(start\_bits + stop\_bits) \\ +start\_bits + header\_bits \end{aligned} \quad (2)$$

where $offset_{actual}$ is the bit offset used with a readback bitstream accounting for non-data bits and $header\_bits$ are the number of bits before the start bit of the first bitstream frame—all other variables are the same as before.

Several additional tidbits of information relating to readback bitstream offsets are required to handle the mapping of logical to physical LUT RAM address bits. The RAM bit addresses provided in the `.ll` file are the physical addresses of the LUTs assuming that `F1` and `G1` are the MSBs for addresses of *F* and *G* LUTs, respectively, and that the LSBs of the addresses are, of course, `F4` and `G4`. For 32x1 LUT RAMs, the *F* LUT contains the first 16 RAM bits (addresses 0–15) and the *G* LUT contains the second 16 bits (addresses 16–31); 32x1 LUT RAMs are denoted using "RAM=M:<addr>" instead of using the LUT name as in the example above.

When `XC4KToJHDLSyms` parses the `.ll` file, it records useful readback bitstream entries in one of two hash tables based on what they represent—RAMs are recorded in what we will call the `LLRAMHash` while all flip-flops are placed in the `LLBlockHash`. The class enters the readback bitstream locations for individual flip-flops into the `LLBlockHash`, using the physical locations of flip-flop outputs as hash keys. Similarly, it places the 16 or 32 bitstream locations for a LUT RAM in a single `LLRAMHash` entry, using the RAM's physical location and RAM depth as a hash key. These hash tables are used later to relate physical circuit elements to their location in the readback bitstream.

5.2 Handling Design Optimizations

Since the instance names of a design's circuit elements can be used to perform the mapping between its logical and physical forms, the `.mrp` file is not needed to perform the primary logical-to-physical mapping between these forms as with the mapping method using net names alone. But, since we still need to be able to take a logical LUT RAM address from a user's design and map it to the physical address in the final, physical circuit, we are still interested in the net names connected to the address pins of LUT RAMs and how their names are modified due to optimization. Thus, the `.mrp` file is still important.

The "Merged Signals(s):" subsection of the `.mrp` file's "Removed Logic" section is the only subsection needed for keeping track of how net names are modified due to circuit optimizations. A typical entry in this subsection of the file is:

```
The signal "PE_Right_Out<0>" was merged into
  signal "LogicCore/Count1BufIn<0>".
```

The first signal in the entry is the original name of the signal while the second is its new name. To make net-name resolution a bit more complicated, the new signal may itself be merged into other nets, causing the signal to receive yet another name. For instance, net *A* may be merged into net *B*, while net *B* is merged into net *C*; thus, net *A* from the original design has been merged into both *B* and *C*, but only net *C* exists in the final physical design. So, several

```
Find the newName in the MergedSignalsHash;
Find the newName in the MergedToHash;
Find the oldName in the MergedSignalsHash;
if(newName has not been merged into other nets, i.e. it is not in the MergedToHash) {
  if(entry for newName doesn't exist in MergedSignalsHash) {
    add entry for newName in MergedSignalsHash;
    add oldName to the oldNames list in newName's MergedSignals object in the
      MergedSignalsHash;
  }
  else {
    add oldName to the oldNames list in newName's MergedSignals object in the
      MergedSignalsHash;
  }
  if(other signals have been merged into the oldName, i.e., oldName has an entry in
     the MergedSignalsHash) {
    add the oldNames list from oldName's MergedSignal object to the oldNames list in
      newName's MergedSignal object in the MergedSignalHash;
    delete oldName's MergedSignalHash entry;
  }
  add a MergedTo entry into the MergedToHash for the oldName;
}
else { /* aliases exist for the new name */
  set mergedIntoNet to the name of the net into which the newName was merged using the
    MergedToHash entry for newName;
  while(mergedIntoNet also has an entry in the MergedToHash) {
    set mergedIntoNet to the name of the net into which the last mergedIntoNet was merged;
  }
  add the old name to the oldNames list in the mergedIntoNet's MergedSignal object in
    the MergedSignalsHash;
  if(other signals have been merged into the oldName, i.e., oldName has an entry in
     the MergedSignalsHash) {
    add the oldNames list from oldName's MergedSignal object to the oldNames list in
      mergedIntoNet's MergedSignal object in the MergedSignalHash;
    delete oldName's MergedSignalHash entry;
  }
  add a MergeTo entry into the MergedToHash for oldName referring to mergedIntoNet as
    the new name;
}
```

Figure 4: Pseudo-code for Resolving Net Name Aliases

levels of merging must be handled to determine what some nets' final names are.

The `XC4KToJHDLSyms` class performs a single-pass algorithm to determine the final name of nets which have been merged one or more times. The algorithm uses two hash tables to quickly perform the name resolution. The `MergedSignalsHash` has entries keyed on the new name for the nets. Each entry is a `MergedSignals` object which associates a new name to a list of old (original) net names since several nets can be merged into a single net. The second hash table, `MergedToHash`, has entries keyed on the old names of nets. Each entry is a `MergedTo` object which records the new net name for each old net name. If a signal has been merged into another signal, it has a corresponding entry in the `MergedToHash`. Pseudo-code for the name resolution algorithm is provided in Figure 4.

When all of the merged signal entries from the `.mrp` file have been added into the `MergedSignalsHash`, multiple levels of net name aliases have been resolved, if needed, and this hash table can be used to quickly find the names of the nets into which other signals have been merged. Specifically, once the `MergedSignals` object for a net is found in the `MergedSignalsHash`, the `oldNames` list of that object can be used to discover what nets have been merged into the net represented by the object.

5.3 Associating Physical and JHDL Elements

The last set of information needed to create a readback symbol table is the final placement, logical names, and interconnection of the readback entities (RAMs and flip-flops) in the physical design implemented by the Xilinx FPGA tools. This information is contained in the placed-and-routed `.ncd` file for the design. The format of the

.ncd file is not publicly known, but Xilinx provides a tool named xdl to convert the .ncd file into a published, textual file format called the Xilinx Design Language (XDL) which retains all of the information of the .ncd file. In other words, the file describes the configuration of everything on an FPGA in a textual format. To reduce the size of XDL files, all the information about programmable interconnection points (PIPs) can be turned off since, in our case, the actual routes taken by nets is not needed to create our readback symbol table.

As we mentioned earlier, the Xilinx FPGA tools preserve the design's instance names for each RAM and flip-flop used in the design, providing an easy, one-to-one mapping between the physical and logical designs. Since these names are preserved, XC4KToJHDLSyms records the physical location and both the physical and logical names of these state elements as it parses the .xdl representation of the physical design. In addition, the program must also record the configuration of the *H1* multiplexers for CLBs, the net names for LUT RAM inputs, and the configuration of the *I1MUX* and *I2MUX* multiplexers of IOBs. The configuration for each CLB's *H1* multiplexer is used to determine the net name for the MSB of a 32x1 LUT RAM's address. The rest of the input net names for LUT RAMs are identified by their connections to the *F1-F4* pins on *F* LUTs and the *G1-G4* pins on *G* LUTs; the net names for RAMs are recorded so the permutation of the RAM address bits can be determined when trying to map the values of logical design RAM bits to their corresponding locations in the physical design's readback bitstream. The configuration of the *I1MUX* and *I2MUX* IOB multiplexers are used to determine which IOB output is used for the output of the IOB's input flip-flop (*INFF*). This is important since readback provides the values of the *I1* and *I2* IOB outputs and not the value of the *INFF* itself.

To simplify the parsing process, XC4KToJHDLSyms does several things. First, it ignores the information provided in XDL module definitions since the instance statements in the physical design's description completely describe the circuit. Second, since there is no guarantee based on the published XDL grammar that net statements will always follow block instances, the program parses the file twice, recording the configuration of CLBs and IOBs the first time and then recording the names of the input nets to LUT RAMs the second time. This allows us to only record the nets that we are interested in and to associate them immediately with the CLB containing the LUT RAMs. When XC4KToJHDLSyms has fully processed the .xdl file, it has entered all of the FPGA blocks (IOBs and CLBs) which have either flip-flops or LUT RAMs into an array, which we will call the XDLBlockArray; XC4KToJHDLSyms will use this array during the final stages of the symbol table creation process.

5.4 Putting it all together

With the data from the .rbsym file created by the JHDL environment at netlist time and the data from the three Xilinx files (.ll, .mrp, .xdl), we now have enough information to create a comprehensive readback symbol table. The construction of the readback symbol table takes three additional steps: associating readback bitstream locations to each entry in the XDLBlockArray, associating the information from the XDLBlockArray with the logical JHDL design elements from the .rbsym file, and permuting the readback bitstream locations for LUT RAM bits. After these tasks have been performed, the symbol table is written to a .rbentry file for use with JHDL hardware execution.

To perform the first association, XC4KToJHDLSyms traverses each entry of the XDLBlockArray. During this traversal, each flip-flop and LUT RAM in each XDLBlockArray entry is then found in either the LLBlockHash or LLRAMHash to determine the readback bitstream location(s) corresponding to that physical element on the FPGA. For CLB flip-flops and IOB output flip-flops (*OUTFF*s), this is a simple lookup into the LLBlockHash. For a *INFF*, the configuration of the *I1MUX* and *I2MUX* has to be tested in the XDLBlockArray entry to determine which IOB output should be looked up in the LLBlockHash. For RAMs, the LLRAMHash is used for determining the readback bitstream locations; since all 16 or 32 locations are recorded in a single entry, only one lookup is required. At the end of the XDLBlockArray traversal, each IOB or CLB entry of the array contains the bitstream locations for the state of their flip-flops or RAMs.

Next, the XDLBlockArray is again traversed, but this time XC4KToJHDLSyms associates each RAM or flip-flop in each entry with a logical JHDL circuit element from the .rbsym file. When the .rbsym file is read in, a hash table of logical JHDL readback symbols, which we will call the RBSymHash, is created, each entry being keyed on the instance name of the JHDL design elements. For each flip-flop or LUT RAM contained in an XDLBlockArray entry, the corresponding logical JHDL symbol is located in the RBSymHash based on the instance name recorded in the .xdl file. When found, the bitstream location corresponding to each flip-flop is then recorded in the RBSymHash entry. For LUT RAMs, the physical permutation of the address bits is recorded in the RBSymHash entries in addition to the readback bitstream locations. Since the logical net names recorded in the RBSymHash entries and the physical net names recorded in XDLBlockArray entries are often different due to circuit optimizations, the MergedSignalsHash is used to check for

net-name equivalences between these two representations when recording address bit permutations. Once the `XDLBlockArray` has been traversed, all flip-flops and LUT RAMs which exist in the physical implementation of the JHDL design have been associated with their logical JHDL counterpart.

As the last process before writing out the `.rbentry` file, `XC4KToJHDLSyms` permutes the readback bitstream locations for RAM bits according to the address permutations recorded in the RAM entries of the `RBSymHash`. When computing the permutations for XC4000 family parts, we had to keep in mind several facts about the addressing of LUT RAMs. First, for 32x1 LUT RAMs, the input into the *H1MUX* is the most significant address bit—it is not permuted with any other address lines by the current Xilinx FPGA tools. Further, the permutations for the address pins of the *F* and *G* LUTs are identical both in 32x1 RAMs and dual-ported 16x1 RAMs. Additionally, the first 16 physical LUT RAM addresses in a 32x1 RAM are in the *F* LUT while the next 16 are in the *G* LUT. Lastly, *F1* and *G1* are the MSBs of physical LUT RAMs' addresses when performing the permutations. Though not taken advantage of in our tools, we also discovered that the addresses for dual-ported 16x1 LUT RAMs (ram16x1d) are not permuted at all by the Xilinx FPGA tools.

With the RAM address permutations handled, the `RBSymHash` entries are written out in an `.rbentry` file for use by the JHDL hardware execution environment. For each entry, the type of the symbol (flip-flop or RAM), the symbol's JHDL instance name, and the readback bitstream location of the symbol's state is recorded in the file. If a certain logical JHDL flip-flop or RAM has been optimized away (i.e., no bitstream locations were associated with the `RBSymHash` entry), the entry is still written out, but an indication is made that it was optimized away by the Xilinx software. At this point, the `.rbentry` file we have created provides a complete mapping of every flip-flop and LUT RAM found in the design's physical implementation to a corresponding design element in the logical JHDL circuit description.

6 Hardware Execution and Readback

With the ability to generate complete readback symbol tables, we created a CCM platform-independent, device-specific readback API for JHDL. Because of its platform independence, the API can easily be integrated into JHDL board models and other tools. The key feature which makes this possible is that the API directly manipulates "raw" FPGA readback bitstreams, which are only dependent on the FPGA type and not the platform. Though most FPGA-based platforms have their own APIs for handling readback, many also provide access to the "raw" readback bitstreams, making our readback mechanism possible.

Figure 5 illustrates how the JHDL readback API fits in the JHDL hardware execution environment. The API interacts directly with a JHDL board model (or other tool) as well as a Java Native Interface to the native board API—an API often written in either C or C++.

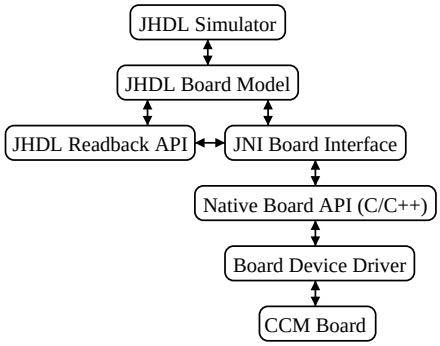

Figure 5: JHDL Hardware Environment Interactions

There are three aspects with integrating the readback functionality into a board model or other design tool: the Java Native Interface (JNI) code for the board, the readback management code, and an API call for generating the `.rbsym` file for a design. We will discuss each of these aspects in turn.

The JNI code for interfacing JHDL with the board's native API must implement two main functions:

int nativeReadBackPE(int peNum, byte [] bitstream)
: This function loads the readback bitstream into the `bitstream` byte array for the processing element (PE) specified by `peNum`. An integer indicating failure or success is returned.

int nativePEType(int peNum, ReadBackPEType peType)
: This function loads information about the PE's type into the `peType` object for the PE specified by `peNum`. The information provided includes the FPGA type (XC4062XL, XC4036EX, etc.); the length of the bitstream's header, i.e., the number of bits before the first start bit of the first configuration frame; and, an indication of whether the MSBs of the bytes in the array have the lowest or the highest bitstream offset number within the byte. An integer indicating failure or success is returned.

These are the only board-specific methods which the readback API requires of the board JNI code for readback operation. As Figure 5 suggests, the readback API calls these JNI functions directly.

The JHDL readback API provides the `ReadBackManager` class for managing readback data. The class provides methods for:

- adding and removing readback symbols for a specific PE on the board,
- enabling and disabling readback for a specific PE,
- actually performing readback on the board's PEs,
- and creating data structures of the readback values so they can be loaded into the JHDL simulator.

Those who write board models or other JHDL-based tools do not have to worry about manipulating bitstreams or making the associations between the JHDL simulation models and the readback state of the hardware; this is all performed by the `ReadBackManager`, its supporting classes, and `XC4KToJHDLSyms`.

To create the `.rbsym` netlist of JHDL circuit elements used by `XC4KToJHDLSyms`, the JHDL readback API provides the `ReadBackSymbolWriter` class. When a board model creates an EDIF netlist for a circuit, it should also instance a `ReadBackSymbolWriter` object and call the `writeRBSymInfo` method for the same circuit to create a `.rbsym` file. Nothing additional is required to create these netlists.

Currently, we have been using this readback API in our models for the Annapolis Micro Systems Wildforce family of CCM boards as well as ISI's SLAAC1 family of boards. From our experience, the JHDL readback API performs readback with little additional performance overhead when compared with using just the readback APIs provided with the boards. Additionally, the memory overhead for JHDL's readback API has been on the order of only a few megabytes for designs with tens of thousands of readback values. The symbolic readback API currently provided with the Wildforce boards suffers from a large amount of memory overhead, requiring as much as 250 MB of RAM for handling an equivalent number of values—the API was clearly not intended for large numbers of readback symbols.

7 Improving FPGA Implementation Tools' Support for Determining Logical-to-Physical Mappings

Having wrestled with the task of determining logical-to-physical mappings for FPGA designs, we have several suggestions that FPGA vendors can follow which can greatly ease this process:

1. *Instance names for nets as well as flip-flops, RAMs, and other state elements should be preserved as long as possible during the design implementation process (netlist conversion, technology mapping, design placement and routing, etc.).* The fact that the XDL representation of the placed-and-routed circuit still reflected the instance names of these design elements made their correspondence to the JHDL design almost trivial.

2. *If the design is modified in any way, report it.* When trying to support the readback of LUT RAMs on XC4000 parts, we discovered that when net names are flattened across a design's hierarchical boundaries by the Xilinx tools, these changes were not reported anywhere, making it hard to track how net names were changed in the design. As a result, this made it especially hard to determine the ordering of the address pins on LUT RAMs since we had no simple way of relating the names of the nets in the physical implementation with those of the JHDL design when optimizations and net name flattening were performed by the Xilinx tools. Our work-around for this problem is to place buffers just before the address pins of the LUT RAMs to prevent the net names from being completely flattened. Xilinx's *map* tool will optimize away the buffers and change the net names for the address signals, but *these changes are reported*, making it easier to associate the physical net names attached to the LUTs' address pins to nets in the JHDL design.[1]

3. *When the state of RAMs can be sampled, the FPGA vendor tools should provide some method of either constraining the ordering of the address pins or reporting how the address pins were permuted.* This addition alone would have greatly simplified the generation of readback symbol tables since the net names of the address signals would no longer be needed. The second option is probably preferred since it allows the place and route tools to better optimize the physical design for speed.

4. *Make report files so that they can be easily parsed by computer software.* Making report files available in human-readable format is important, but providing some command option for making the reports reasonable to parse using a grammar would ease the process of determining logical-to-physical design mappings. For instance, the `.mrp` file breaks lines at a set location to make the files more readable in a text editor or other viewer, but the line breaks make the file very

[1] We consider this approach to be a bit of a "hack".

difficult to describe with a grammar for compiler tools such as *JavaCC* or *lex* and *yacc*.

5. *Vendors should provide designers with the ability to fully determine how the FPGA was configured using an open file format.* We found the XDL representations of XC4000-based designs to be quite important for creating symbol tables. Without this information, it would have been hard to determine which IOB output was used for the output of the *INFF*, which net was attached to the MSB of a 32x1 LUT RAM's address pins, or how the addresses of LUT RAMs were permuted. This degree of visibility into the physical design was crucial for determining the logical-to-physical mapping of the design and, thus, for creating a complete and accurate readback symbol table.

8 Other Uses of Logical-to-Physical Mappings

Having gone through the exercise of determining how logical designs are mapped to physical FPGA circuits, we have come to realize how useful this information really is. With the information, the design environment can help the designer better understand how designs were optimized as well as the characteristics of their FPGA implementations. For instance, beyond the ability to support hardware debugging in a simulation-like environment, this information can be used to help estimate the power consumption of circuits from within the simulation environment—we know which nets exist in the physical circuit and can monitor their toggle rates in the simulator. Further, we might be able to extract capacitance or path length estimates from the physical design to provide more accurate power models.

Understanding these mappings also has the potential to allow quick modifications of designs without having to go through the complete ASIC-style design cycle over and over again. As an example, several designs we have worked on at BYU required only the modification of ROM contents to change the circuit's operational parameters. Through the direct modification of bitstreams via JBits [8] or an XDL representation of the circuit, these design modifications would take only minutes not hours to complete, as can happen when the entire Xilinx FPGA design flow is used. For that matter, the run-time reconfiguration or modification of these same circuits becomes easier to perform and manage with this information.

9 Summary and Future Work

We have discussed the utility of knowing the logical-to-physical mappings of FPGA circuit designs in the context of hardware debugging and readback. As an illustration of their utility, we have discussed a methodology for providing platform-independent, device-specific support for readback in JHDL board models and execution environments. This API has been successfully used to support board models for both the Wildforce and SLAAC1 families of CCMs, providing good performance with little memory overhead. Since the readback software for JHDL can completely determine the logical-to-physical mappings of JHDL designs, it can provide developers of JHDL board environments with a simple way of adding readback capabilities without large amounts of additional coding. The JHDL readback API depends on the ability of CCM board APIs to provide "raw" readback bitstreams from FPGA processing elements to make platform independence possible.

In the course of the paper, we have also discussed in detail how to determine the logical-to-physical mappings of all flip-flops and LUT RAMs in optimized, placed-and-routed XC4000 designs. Several key factors made this possible, including the preservation of designs' instance names by the Xilinx FPGA tools and the ability to view the complete configuration of the FPGA using XDL. The knowledge of how JHDL designs are mapped to Xilinx design primitives at netlist time also simplified the process.

In the near future, we plan to support these same logical-to-physical mapping capabilities for the Xilinx Virtex family of FPGAs in the JHDL readback API as well as provide JHDL designers with feedback concerning how their designs were optimized by the Xilinx tools. We also plan to explore other possible applications of these mappings, including modeling power in FPGAs, quick design modifications, and controlling low-level design implementation.

References

[1] P. Bellows and B. L. Hutchings, "JHDL - an HDL for reconfigurable systems," in *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines* (J. M. Arnold and K. L. Pocek, eds.), (Napa, CA), pp. 175–184, Apr. 1998.

[2] B. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, and M. Rytting, "A cad suite for high-performance fpga design," in *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines* (K. L. Pocek and J. M. Arnold, eds.), (Napa, CA), p. n/a, IEEE Computer Society, IEEE, April 1999.

[3] W. Hölfich, "Using the XC4000 readback capability," Application Note XAPP 015, Xilinx, XC4000, San Jose, CA, 1994.

[4] C. Carmichael, "VIRTEX configuration and readback," Application Note XAPP 138, Xilinx, San Jose, CA, March 1999.

[5] J. M. Arnold, "The Splash 2 software environment," in *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines* (D. A. Buell and K. L. Pocek, eds.), (Napa, CA), pp. 88–93, Apr. 1993.

[6] B. K. Fross, R. L. Donaldson, and D. J. Palmer, "Pci-based WILDFIRE reconfigurable computing engines," in *Proceedings of SPIE—The International Society for Optical Engineering*, vol. 2914, (Bellingham, WA), pp. 170–179, SPIE, SPIE, November 1996.

[7] Xilinx, San Jose, CA, *The Programmable Logic Data Book*, 1999.

[8] S. A. Guccione, D. Levi, and P. Sundararajan, "JBits: A Java-based interface for reconfigurable computing," in *Second Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, (Laurel, MD), September 1999.