Odin: A Verilog RTL Synthesis Tool for FPGA Research - User Manual

Peter Jamieson

March 28, 2005

1 Introduction

This document is intended to help researchers install, use, understand, and develop our software tool, Odin. We will cover these details in a number of sections. Section 2 covers the basics about setting up the system. Section 3 discusses how to use Odin, and what inputs and outputs files Odin uses and generates. Section three provides some understanding of the design of Odin, and Section four

provides some strategies for debugging and developing new code for Odin. Sectionfinal describes some things that remain to be done, caveats, and concludes this document.

2 Setting up Odin

Odin is a Verilog back-end synthesis tool targeting FPGAs that currently uses Icarus [1] as front-end Verilog parser. The version of Odin, which you have downloaded, includes the Icarus parser and the Odin source code. To find the Odin software look for "tgt-odin", which is the top level for the Odin tool.

Before getting Odin setup you need to make sure you have the proper libraries and software. Each of these software packages was downloaded and compiled from the various sources, and I added their bin directories to my path. Additionally, the libraries that are used are included in the Makefiles for both Odin and Icarus need to be properly specified. Here is the list of software that I had to install to get Odin working:

- binutils - BFD version 2.13.2
- GCC 3.2 - gcc version 3.2
- gdb - GNU gdb 5.3
- gperf - gperf-2.7.2
- libxml2 - libxml2-2.6.2

I'm not sure how Odin compiles if you use different libraries, so I provide these as a basis to help you figure out what libraries and Linux tools are needed. I have also compiled Odin on Cygwin with the equivalent or newer libraries.

Icarus is designed a front-end parser that has the ability to target many back-ends. To do this Icarus dynamically links a library at run-time and executes the back-end through a call to target_design(). Odin, however, is intended as a research tool, and to help debug Odin it is better to join the front-end with the back-end statically. To do this I have made a few changes to Icarus. Here are the files that have been modified (all files are listed off the top Icarus directory normally called verilog-xxxx):

- BIN/lib/ivl/odin-paj_vpr.inc has been added, and this file specifies the parameters that get passed when odin is targeted.

- t-dll.c is the point where Icarus does dynamic linking. We have changed a few lines to statically link Odin with Icarus.
- Makefile has been changed to link some of the libraries Odin depends on.

You don't have to change any of these files to get Odin compiled and running except the Makefile. The main issue with getting Odin to compile and run is correctly updating the paths. Here are some stages to setup and compile Odin.

1. tgt-odin/Makefile - change directories for -I, -L and prefix to the directories where you are installing Odin, and where the appropriate libraries are located
2. BIN/lib/ivl/odin-paj_vpr.conf needs proper paths for the directories where you are installing Odin
3. peter_peter_compile_script_and_config_local.bash and peter_compile_script_local.bash need to have directories where you are installing Odin
4. run peter_peter_compile_script_and_config_local.bash - this won't compile successfully, but will setup the Makefiles
5. Change line in Icarus Makefile from: "$(CXX) $(LDFLAGS) -o ivl $O $(dllib)" to: "$(CXX) $(CXXFLAGS) $(LDFLAGS) -finstrument-functions -o ivl $O $(dllib) -Ltgt-odin -lodin -Ltgt-odin/PETER_LIB/ -lpeters -lm -L/nfs/eecg/q/grads10/jamieson/PROGRAMMING_TOOLS/libxml2-2.6.2/PETER_BIN/lib -lxml2 -lz -lpthread" where: "-L/nfs/eecg/q/grads10/jamieson/PROGRAMMING_TOOLS/libxml2-2.6.2/PETER_BIN/lib" should be defined as the location of your libxml2 library.
6. cd tgt-odin/ && make clean && make && cd .. && make clean && ./peter_compile_script_local.bash
7. Test setup by reading the directions in tgt-odin/SAMPLE_FILES

At this point Odin is setup on your system. Good luck.

3 Using Odin

In the previous Section I described how to setup Odin. The final step in the setup actually describes the location of a file that describes the basic usage of Odin. In

this Section I will discuss some more details about running Odin. This includes a description of the input and output files, a description of some useful scripts, and a brief description of what happens during system execution.

Here is a brief description of the input files that Odin uses:

- (design_name).v - this is the Verilog design which Odin will synthesize.
- config_file.txt - this really isn't an input file to Odin, but this is an example of a file that is passed into ICARUS that describes the arguments to ICARUS. However, this file is very important since the arguments in this file will eventually get passed to Odin. These passed arguments describe the location of input files, the target FPGA, and so on. Take a look at tgt-odin/SAMPLE_FILES/config_file.txt to get a feel for the this file. The most important argument in this file is the flag:arch=? where ? can be "vpr", "stratix", and soon "virtex". This parameter essentially describes the architecture you want to target.
- dynamic_debug_file.xml - this file is used for a little debugging technique I use in Odin. Essentially, Odin's internal data structure is a large graph that represents the circuit netlist. Sometimes when I want to stop GDB on a specific node or another data structure when it is created, I can specify a unique number in this file (part of all data structures in Odin *m_id*) that is associated with the structure. Odin then breaks in GDB when this node is created. Similarly, I allow this to be done for items that are being freed. I suggest looking at ou_malloc and onu_node_free in Odin's code to get a better understanding of these techniques. A sample of this file can be found in tgt-odin/dynamic_debug_file.xml. This xml file is parsed in odin_xml_parser_config_files.c.
- optimization_file.xml - this file is used to turn different optimization techniques in Odin on or off. The comments in the file describe each of these techniques. A sample of this file can be found in tgt-odin/optimization_file.xml, and this file is also parsed in by odin_xml_parser_config_files.c.
- tech_lib.xml - this file describes the architecture that is being targeted. I haven't formalized how these files are created and inputted into Odin, and currently, describing an FPGA architecture is spread out all over. This file essentially is used to describe DSP-block structure on the Stratix FPGAs.

Here is a brief description of the output files that Odin generates:

- (synthesize_design_name).v or blif - this is the outputted Verilog file of the synthesized design to a structural netlist. The actual name of the output file is specified in the config_file.txt described earlier.
- ?_log_file - this file contains any messages Icarus or Odin sends out.
- ?_stats_file - this file contains the information of any statistics that Odin has collected about the design.

The execution of Odin is as follows: First, a verilog file, the configuration file that passes the argument, and the other inputs are passed into Icarus/Odin. The verilog design is read and converted into an intermediate representation in Icarus. All this information including the original arguments are passed to Odin. Odin reads in all the input files into data-structures, and processes the intermediate representation. Odin does its thing, and all the outputs are generated.

For additional information about Icarus [1] check out the web-page and documentation.

4 A Description of Odin

The following is taken from a paper we are writing on Odin. The best place to see how Odin performs all these stages is from the top-level *odin.c*.

Figure 1 shows the major stages of the Odin tool. First, a front-end parser, Icarus [1], parses the Verilog design and generates a hierarchical representation of the design.

Second, Odin has an elaboration stage that traverses the intermediate representation of the design to create a flat netlist that consists of structures including logic blocks, memory blocks, if *and* case *blocks, arithmetic operations, and registers. Each of these structures within the netlist we refer to as a node in the netlist.*

Third, some simple synthesis and mapping is performed on this netlist. This includes examining adders and multipliers for constants, collapsing multiplexers, and detecting and re-encoding finite state machines to one-hot encoding. These mappings are discussed in more detail in the next Section.

Fourth, an inferencing stage searches for structures in the design that could be mapped to hard circuits on the target Field-Programmable Gate Array (FPGA). These structures are connected sub-graphs of nodes that exist in the design netlist. One of the inputs to the Odin tool that allow it to flexibly target this and like

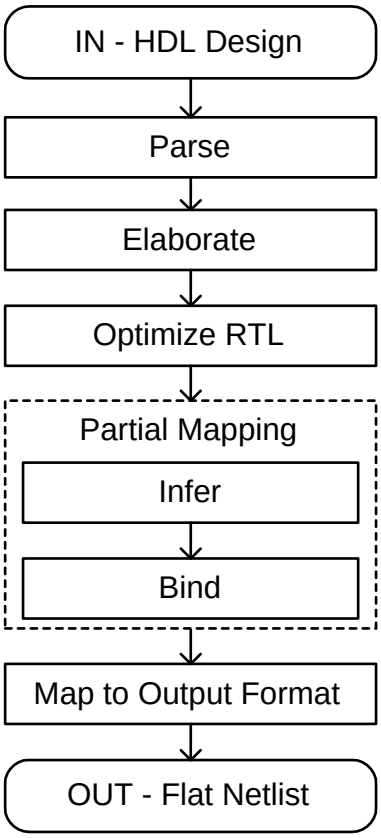

Figure 1: This is a general flow to convert Verilog designs to logic netlists.

structures, is a text file that describes the node subgraph to search for this kind of hard circuit mapping. We search for these structures in the netlist using a matching algorithm.

Fifth, a binding stage guides how each node in the netlist will be implemented. This is done by mapping nodes in the netlist to either hard circuits, soft programmable logic, or a mixture of hard and soft structures. One way to do this is to map structures to Library Parametrized Modules (LPMs), which later stages of the industrial Computer Aided Design (CAD) flow will bind to an implementation on the FPGA whether that be a hard or soft implementation.

Finally all nodes are converted into a flat netlist consisting of connected complex logic structures and primitive gates.

To understand the details within Odin, it is best to look at the actual code. My recommendation is start with *odin.c* and *odin_types.c*. The first C file is the top-level of Odin, and you can proceed from there to see all the little pieces of execution. The second C file contains most of the datasturctures used in Odin. Look at the major datastructures to get an understanding of how I build the netlist of the design.

5 Development and Debugging Strategies with Odin

5.0.1 Developing

With an understanding of Odin, you're now at the stage where you can add new code, target different structures, and build new features for Odin. There's lots to improve in Odin, and we believe there are many ideas at the RTL synthesis stage that will affect the overall area, speed, and power results of designs mapped to FPGAs. This section gives some suggestions on how to develop new features in Odin and common debugging tactics that I use.

First of all, one of the major flaws that currently exists in Odin is the existence of two major data structures. This flaw occurred for historical reasons, and weak brain power. I have a tough time wrapping my head around the data structures that Icarus generates, and I essentially hacked my way through converting this structure into a netlist which I could deal with. I then take this data-structure and convert it into a flattened netlist built of somewhat reasonable structure. I then went back and found ways to make some of the original parsing of the file directly to the second data structure, but had neither the time or success to convert everything.

Historical flaws aside, Odin still needs to be re-factored to be better at creating the flattened netlist, but I don't plan on doing this in the near future. Therefore, if you're going to add things try to deal with the second data structure; this is the *node_t* structure. This is not always possible, and sometimes you need to identify structures or functionality at the conversion stage. Do the identification when it has to be done, but perform optimizations at the flattened stage if possible.

One of the standard features that I do with Odin, is attempt to recognize and implement a new structure. I have done this in a variety of ways with varying success in each case. For example, odin_finite_state_machine.c I identify and remap finite state machines. Other examples include inferring complex DSP-block structures, and identifying resets for registers. I have never found a clean way to do identification. Once I get the identification information, I pass this information to later stages. odin_soft_mapping.c is a very interesting file since it is the stage where I convert larger nodes to logic implementations. When you make changes here, one of the standard problems I have is not remembering to map all inputs/outputs properly. This can be painful to debug, but the dynamic debug technique helps, and my asserts tend to capture many errors you'll make.

I can't give much more advice on adding and changing features. Read Code Reading the Open Source Perspective by Diomidis Spinellis for some good ideas on how to read software. I wish I read this before I started.

5.0.2 Debugging

As for debugging, there are lots of little pieces of advice I can give since I've spent lots of time decoding my software.

The *dynamic_debug_file.xml* is a very useful entry point in gdb to catch the creation and destruction of data structures. The big advantage is you don't need to recompile Odin to setup these breakpoints (if only I had though of this idea much earlier). I use this technique to find the creation points of datastructures, and then I can watch the data structure to see how it is used and changed as execution continues. This is usually sufficient to find out at what point I do something incorrect with the structure. Similarly, I use this technique at destruction points so I can find out why something was destroyed and why this destruction is a factor in the bug I'm trying to find.

Another technique that I use is *watch *((struct ?*)address)* to put watch breakpoints on structures in gdb. This little snippet of debugging commands helps significantly for finding out when dynamic datastructures are modified.

I still believe the biggest thing Odin is missing is a graphical view of the

netlist that is quickly traversable. Since many of the problems in Odin come from incorrectly changing and forming parts of the netlist, half the time in debugging spent performing textual traverses of portions of the netlist. Maybe a dynamic text traversing technique would be beneficial.

I'll add more specific debugging cases over time.

6 Miscellaneous and Conclusion

6.0.3 The missing and the got to do

Odin is by no means a complete back-end for synthesizing Verilog circuits to netlists that target various technologies. Some of these challenges deal with the short-comings of Icarus the front-end, and most of these I work around by rewriting the Verilog benchmark. In this section I will list some of the to do's and problems with Odin.

- Common subexpression elimination not handled - can rewrite Verilog
- Functions handling is defunct - can rewrite Verilog
- Not all ways of writing a state machine will be identified by Odin - can rewrite Verilog
- Inputs for technology target is spread out over many files
- Matching algorithm in partial-mapping is a mess, but works so far
- Parsing of Icarus intermediate representation complicated and confusing
- Many Verilog structures not handled. Generate loops, parametrized modules, divider, casex, ...
- Verification for a limited number of benchmarks, and use random vectors = weak verification
- Identification of sensitivity signals and their usage is messy and seems like it could be better
- Optimizations done blindly without any timing model
- Constant propagation could be better for more complex structures

- Timing issues described in Verilog not handled
- Inferencing sign arithmetic operations not done

I have been working on building Odin for two years, and I'm satisfied I've got as far as I have. This document will help you get Odin up and running so you too can do some interesting things with Verilog. Please provide input, criticism, and contributions/improvements as I hope to improve the quality of this software for others to use. Contact me at: jamieson.peter@gmail.com. I will reply as soon as I can, and I plan to make a mail list so I can keep you folks up to date with changes I make.

Peter Jamieson - created Mar 10th, 2005

References

[1] ICARUS Verilog at www.icarus.com/eda/verilog/.