PLSI: A Portable VLSI Flow

by

Palmer Dabbelt

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Krste Asanovic, Chair
Jonathan Bachrach

Spring 2017

The thesis of Palmer Dabbelt, titled PLSI: A Portable VLSI Flow, is approved:

Chair _____ Date _____

_____ Date _____

_____ Date _____

University of California, Berkeley

PLSI: A Portable VLSI Flow

Abstract

PLSI: A Portable VLSI Flow

by

Palmer Dabbelt

Master of Science in Computer Science

University of California, Berkeley

Krste Asanovic, Chair

Write an abstract

Contents

List of Figures

List of Tables

Acknowledgments

Acknowledge people

Chapter 1

Introduction

Computer architecture research has historically been handicapped by a lack of high quality baselines. Thanks to the advent of RISC-V and Rocket Chip, there are now high quality baseline implementations of multiple families of microarchitectures availiable for free and with permissive licenses, along with silicon-proven results for some benchmarks. Unfortunately there is no existing VLSI flow that allow architecture researchers to reproduce these results and to obtain numbers for their proposed ideas.

PLSI is designed to be this flow: specifically PLSI is designed to allow computer architecture researchers to quickly get VLSI numbers, allowing them to do RTL based research. To achive this, PLSI uses standard commercial VLSI tools and flows whenever possible – for this thesis only the Synopsys-based flow is demonstrated, but there are experimental flows for both Cadence and open-source tools. In addition to being portable between multiple tools vendors, PLSI is designed to be portable to multiple designs (to enable researchers to build their own cores) and to multiple processes (since researchers tend to have access to an odd set of technologies).

The specific contributions of PLSI are:

- A macro synthesis tool used to map user's memories to various flavors of ASIC SRAMs.
- A modulor, technology-independent floorplanning framework.
- Criteria for ensuring the commercial CAD tools produce reasonable results.
- Usability improvments from a good top-level Makefile that allows the entire flow to be automated and reproducable.

This thesis presents results for three Chisel-based cores (Rocket, BOOM, and Hwacha) mapped to Synopsys's educational 32nm technology via the Synopsys tools.

1.1 Sponsors and Such

I never know what to put here.

Chapter 2

Implementation

PLSI is designed to be portable to multiple technologies, CAD tool vendors, and processor generators. As such it has a modular system of addons that can be mixed and matched in order to produce a chip. Users are expected to define the following variables, which will be used to control the flow:

- `CORE_GENERATOR`: Selects the core genertor, which generates the RTL for the chip.
- `SOC_GENERATOR`: Selects the SOC injector, which adds chip-specific RTL to the core. This includes things like top-level IOs.
- `TECHNOLOGY`: The technology to map to.
- `SYNTHESIS_TOOL`: The synthesis tool to run.
- `PAR_TOOL`: The place and route tool to run.
- `FORMAL_TOOL`: The formal verification tool to run.
- `SIGNOFF_POWER_TOOL`: The tool to be used for signoff power analysis.
- `SIMULATOR`: The simulator to use when verifying the design.

It is also possible to pass configuration information to each step in the flow by setting the following variables

- `CORE_CONFIG`: This configuration is passed to the core generator and specifies things like cache sizes.
- `CORE_SIM_CONFIG`: This configures the simulations that will be run. This only exists at the core level as the rest of the steps in the flow are expected to read the core's test list and figure out duplicate tests when necessary.

- `SOC_CONFIG`: Configures the SOC generator, allowing users to set things like the interface width.
- `MAP_CONFIG`: Specifies how the SOC will be mapped to a technology, including things like the corner and target frequency.

2.1 Overview

A top-level Makefile sequences the entire chip build; which includes running all the CAD tools, testing the designs after every tool is run, and interpreting the results. A high-level overview of the PLSI flow can be seen in Figure 2.1.

While the primary user interface of PLSI is the top-level Makefile, that's not actually a particularly interesting implementation challange: everyone knows you can describe arbitrary dependency graphs in make and then execute them. The interesting part of PLSI are the tools that implement the various rules and the interchange formats that are passed between the various tools. The fundamental driving design decision behind PLSI is that computers are better than performing repetitive, arithmatic-laden tasks that humans are. When implementing PLSI I took my experience from working with a handful of tapeout teams.

This design philosophy is probably best explained with an example. Early in the design of the RISC-V ISA we flip-flopped a few times between the 64-bit ISA having 4KiB pages vs it having 8KiB pages. This resulted in our designs with a 16KiB L1 data cache moving between 2 sets of 8 KiB and 4 sets of 4 KiB. Propagating this change from the ISA spec to GDS required the intervention of a half dozen students and involved months of latency.

- The ISA designer changes the spec. This spec change is already automatically propagated to the RTL, so this step is easy.
- The RTL lead on the chip needed to trigger a merge, since this ISA change was incompatible with existing binaries. This incompatibility wasn't usually found until software failed to run, which required extensive debugging.
- A new memory macro wrapper needed to be added to the list of known memory macros. This list was duplicated multiple times, so this required talking to a handful of people.
- The verification guy's scripts needed to be modified to run simulations, as the names of the memory macros had changed.
- The floorplan needed to be updated to reflect the new macro names.
- Someone needs to produce new post-PAR power numbers, which we never actually did.

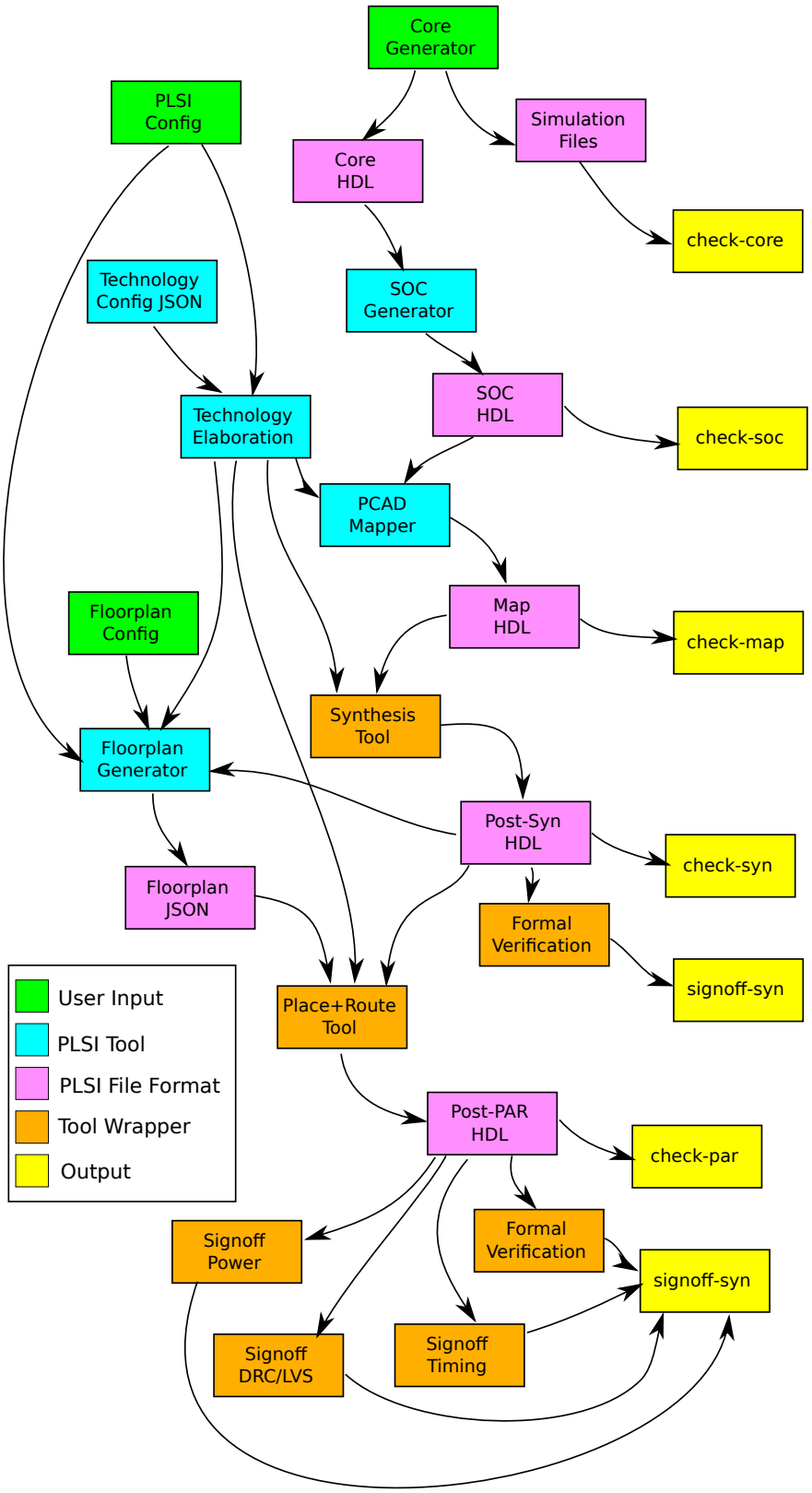

Figure 2.1: A high-level overview of PLSI

```
CORE_DIR ?= $(CORE_GENERATOR_ADDON)/src/rocket-chip

RC_CORE_ADDONS = hwacha
CORE_TOP = ExampleTop

include src/addons/core-generator/rocketchip/vars.mk
```

Figure 2.2: The core-generator addon for Hwacha

All this manual labor and communication was required in order to perform effectively no useful work: we were using 4 KiB SRAMs so we ended up having the same macros put in the same places. Contrast the old flow with the PLSI flow for performing the same change:

- The ISA designer changes the spec.
- The ISA designer runs "make" to produce post-PAR performance numbers.

This is really the key to PLSI: since everyone can run the flow, making a small change doesn't require involving everyone in the project. This allows everyone working on the project to get actual work done as opposed to spending their time propagating other people's trivial changes. This isn't a new idea: this sort of continuous integration flow is common to many software development methodologies, it's just that for some reason it doesn't seem to have propagated to how people are building chips.

2.2 Core Generators

PLSI was designed to support RTL-based research into microarchitectures by quickly providing ground-truth power numbers for a large number of core designs. Thus the whole point of PLSI is that it is easy to port to new cores. Porting PLSI to a new core generator based on Rocket Chip is extremely easy: users simply need to point PLSI at their fork of Rocket Chip. For example, Figure 2.2 shows how little code is required to port PLSI to the Hwacha vector unit.

Porting PLSI to a core generator that isn't based on Rocket Chip is a bit more involve. The most interesting part of porting to PLSI is producing the list of macros that the core requires. There is a custom FIRRTL backend in PLSI that allows Chisel based designs to easily generate PLSI macro configurations (this tool is automatically used for Rocket Chip based projects), but Verilog based cores have a bit more work to do. I haven't actually managed to get a reasonable open-source Verilog core, so I haven't ported PLSI to one yet.

2.3 SOC Generators

The SOC generation step converts a processor into an actual chip. This means inserting things like pads, clock recievers, etc. For performing evaluations of microarchitectures there is a "nop" SOC generator, which just doesn't do anything. This provides a simple way of evaluating a design without getting into the complexities of chip building. This was the only SOC generator used for this thesis.

In addition to the "nop" SOC generator, there is also a "bar-testchip" SOC generator. This SOC generator generates a controller and phy that transparently shims all top-level decoupled interfaces over a low-speed, single-ended interface that is feasible to implement using the standard digital IO pads that a popular commercial foundry provides. This is designed to provide a simple mechanism for building research-style test chips: it alleviates the need to maintain chip-specific IO harnesses. The process is completely automated: a PLSI tool reads to top-level Verilog of the target design (using a Verilog parser I wrote), infers the decoupled interfaces, generates an ASIC top-level wrapper and a test harness for both FPGAs and simulation.

2.4 Technology Mapping

The technology mapping step converts generic macro implementations to those that are specific to a particular technology. This step is important to maintaining portability of designs to multiple technologies: RTL writers should never describe a technology-specific concept in their code but should instead describe a generic version of that concept, which the mapping step then maps to a technology-specific implementation. This is really just a synthesis tool, but it handles all the things that commercial synthesis tools don't understand.

The most important example of this is mapping sequential memories to SRAMs on ASIC targets. This is something that traditional synthesis tools can't do because it's impossible to safely infer ASIC SRAMs from synthesizable Verilog. Chisel and FIRRTL handle this mismatch by having an explicit memory visible to the RTL programmer, which has been designed such that it can be mapped to ASIC SRAMs. PLSI uses this information to generate ASIC SRAM wrappers which can then be passed on to the remainder of the CAD flow.

Macro JSON Files

PCAD has standardized on JSON as the interchange format between every step in the flow because it's a well supported format. The technology mapping step expects a list of all the macros that need to be implemented and a list of all the macros availiable to the technology. Figure 2.3 shows an example of one of the branch predictor memories in one of the BOOM configurations, and Figure 2.4 shows the ASIC SRAM that will be used to implement that memory on the 32nm EDK.

```
{
  "type": "sram",
  "name": "h_table_ext",
  "depth": 32768,
  "width": 1,
  "ports": [
    {
      "clock port name": "RW0_clk",
      "mask granularity": 1,
      "output port name": "RW0_rdata",
      "input port name": "RW0_wdata",
      "address port name": "RW0_addr",
      "mask port name": "RW0_wmask",
      "chip enable port name": "RW0_en",
      "write enable port name": "RW0_wmode"
    }
  ]
},
```

Figure 2.3: An example memory that needs to be mapped

PCAD Macro Compiler

In order to map technology-agnostic macros to technology-specific macros, I implemented what is in effect a synthesis tool with a limited scope. This tool is an optimizing synthesis tool that uses some very simple optimization techniques. The only macro type I bother optimizing is the memories because they're the only macros that I can compile that have any optimizations potential.

Technology JSON Files

All the technology-specific information in PLSI is described within a single file: the technology configuration file. Defining all technology-specific information in this manner is what allows PLSI backends to be portable to multiple technologies.

An example technology JSON file is shown in Figure 2.5. As you can see, technology JSON files describe where to download a technology tarball from, how to extract that tarball, and the contents of that tarball. It is possible to use scripts that operate on the post-extracted technology PDKs to generate library fragments: for example, a script reads TSMC's memory compiler documentation PDF to produce the list of SRAMs that compiler is capable generating, which can then be passed to the PLSI macro compiler in order to map arbitrary Chisel memories to TSMC SRAMs.

```
{
  "family": "1rw",
  "width": 8,
  "name": "SRAM1RW1024x8",
  "ports": [
    {
      "write enable port name": "WEB",
      "clock port polarity": "positive edge",
      "output port polarity": "active high",
      "write enable port polarity": "active low",
      "address port polarity": "active high",
      "chip enable port polarity": "active low",
      "clock port name": "CE",
      "input port name": "I",
      "output port name": "O",
      "chip enable port name": "CSB",
      "read enable port name": "OEB",
      "input port polarity": "active high",
      "address port name": "A",
      "read enable port polarity": "active low"
    }
  ],
  "type": "sram",
  "depth": 1024
},
```

Figure 2.4: An example ASIC SRAM macro description

When porting PLSI to a new technology it should only be necessary to create a new technology JSON file (presumably patterned off a similar existing technology JSON file). In practice some modifications to all the tools that consume the technology JSON files will be necessary, but the goal is that with each new technology these modifications will become simpler and simpler.

PLSI Configuration File

PLSI must be provided with a configuration file in order to configure the mapping phase. This configuration file contains the additional information needed to run the VLSI flow that is not present as part of the core generator. An example PLSI configuration file is show in Figure 2.6, which is the default configuration file for mapping Rocket Chip's `DefaultConfig` to the Synopsys 32nm educational PDK. It's important to note that while this config file is

```
{
  "name": "An example technology",
  "tarballs": [
    "path": "tech.tar",
    "homepage": "http://example.com/technology",
  ],
  "libraries": [
    {
      "nldm liberty file": "tech.tar/lib/stdcell/tt0p9v25c.lib",
      "verilog file": "tech.tar/lib/stdcell.v",
      "corner": {
        "nmos": "typical",
        "pmos": "typical",
        "temperature": "25 C"
      },
      "supplies": {
        "VDD": "0.9 V",
        "GND": "0 V"
      }
    },
    {
      "nldm liberty file": "tech.tar/lib/sram1024x8/tt0p9v25c.lib",
      "provides": [
        "type": "sram",
        "width": 8,
        "depth": 1024,
        "ports": [
          {
            "read port name": "R",
            ...
          }
        ]
      ],
      ...
    }
  ]
}
```

Figure 2.5: An example technology JSON file

```
{
  "clocks": [
    {
      "name": "clock",
      "period": "1250ps",
      "par derating": "250ps"
    }
  ],
  "scenerios": [
    {
      "corner": {
        "nmos": "typical",
        "pmos": "typical",
        "temperature": "25 C"
      },
      "supplies": {
        "VDD": "1.05 V",
        "GND": "0 V"
      }
    }
  ]
}
```

Figure 2.6: An example PLSI configuration file

specific to a technology, it is independent of the CAD tool vendors.

The PLSI configuration file is used to drive all the CAD tools that are run, including both the proprietary ones via wrappers written for PLSI and tools like the macro mapper that were written specifically for PLSI. It allows users to control chip-related settings that do not come from RTL, like the clock speed and MCMM scenarios.

2.5 Synthesis Tool Driver

In order to be portable to multiple CAD tool vendors, PLSI expects that pre-existing synthesis tools are wrapped up with a script that converts the vendor-agnostic PLSI file formats into vendor-specific formats, runs the synthesis tool, and verifies that the run completed without errors. As a study in portability, there are three synthesis tools supported: "yosys", an open-source synthesis tool; "dc", Synopsys's Design Compiler; and "genus", Cadence's GENUS synthesis tool. Only basic features are supported for "yosys" and "genus", while "dc" is the tool that was regularly used.

These wrapper scripts are pretty straight-forward. In order to avoid copyright violations, users are expected to have downloaded tarballs of the vendor's base set of scripts. PLSI then patches these scripts with patches that are generated by reading the various synthesis inputs (the post-map verilog, technology description file, and PLSI configuration file). For example, in order to produce a list of modules

2.6 Floorplanning

One of the more painful steps of producing ASIC results is floorplanning. Our floorplanning scripts have traditionally been non-portable even for similar designs on the same process, much less across different processes. In order to allow floorplanning to be portable to multiple designs, tools, and processes users write floorplan fragments in a Python DSL that was created for this project.

PLSI's floorplanning system is based on generating hierarchical groups of placeable elements, specifying hints as to how they should be arranged, and nesting these groups. The floorplanning DSL is designed to make it easy for users to produce legal first-cut floorplans and then optimize them when trying to actually build a chip. This DSL handles the following pain points for users:

- Matching Chisel-generated names to logical floorplan groups.
- Shaping logical floorplan groups.
- Ordering macros within logical floorplan groups.

SRAM Floorplanning

The vast majority of the floorplanning work has gone into handling SRAM macros for ASICs, since those are by far the most important QoR constraints when building high performance processors. The floorplanning step is semi-automated: it's designed to make it easy for users to write first-cut floorplans that apply to multiple design configurations while still allowing them to fine-tine floorplans for maximum performance on the designs that end up being interesting.

Figure 2.7 shows the floorplanning script for a single Rocket core without an L2. As you can see, users can write any Python code they want to control floorplans in as much detail as they want – even providing absolute coordinates for every floorplanable block if they so desire. The canonical way to write floorplans in PLSI is to use the demonstrated helpers to write floorplans in a more technology-agnostic way.

For example, the floorplan demonstrated in Figure 2.7 should work on any of the technologies I have had access to, and should work for any reasonable configuration of Rocket. This floorplan puts the data RAMs for the instruction and data caches on opposite sides of the chip and puts the metadata RAMs in between, limiting the width of the metadata

```
class RocketTilePlacer:
  def __init__(self, config):
    self.top = None
    self.l1dd = []
    self.l1dm = []
    self.l1id = []
    self.l1im = []

  def insert(self, macro):
    if macro.matches(config.rtl_top):
      self.top = TopMacro(macro.name, macro.width, macro.height)
    elif macro.matches("coreplex/rocketTiles/dcache/data"):
      self.l1dd.append(macro)
    elif macro.matches("coreplex/rocketTiles/dcache/MetadataArray"):
      self.l1dm.append(macro)
    elif macro.matches("coreplex/rocketTiles/frontend/icache/u"):
      self.l1id.append(macro)
    elif macro.matches("coreplex/rocketTiles/frontend/icache/tag_array/tag_array"):
      self.l1im.append(macro)
    else:
      print("%s cannot be matched" % macro.name)
      return False
    return True

  def list_constraints(self):
    l1dd = TopLeftPlacer   (self.top, self.top.tl(), sorted(self.l1dd))
    l1dm = TopLeftPlacer   (self.top, l1dd.bl(),     sorted(self.l1dm), self.top.width *
    l1id = BottomLeftPlacer(self.top, self.top.bl(), sorted(self.l1id))
    l1im = BottomLeftPlacer(self.top, l1id.tl(),     sorted(self.l1im), self.top.width *
    return l1dd.place() + l1dm.place() + l1id.place() + l1im.place()
```

Figure 2.7: Example Rocket Floorplan

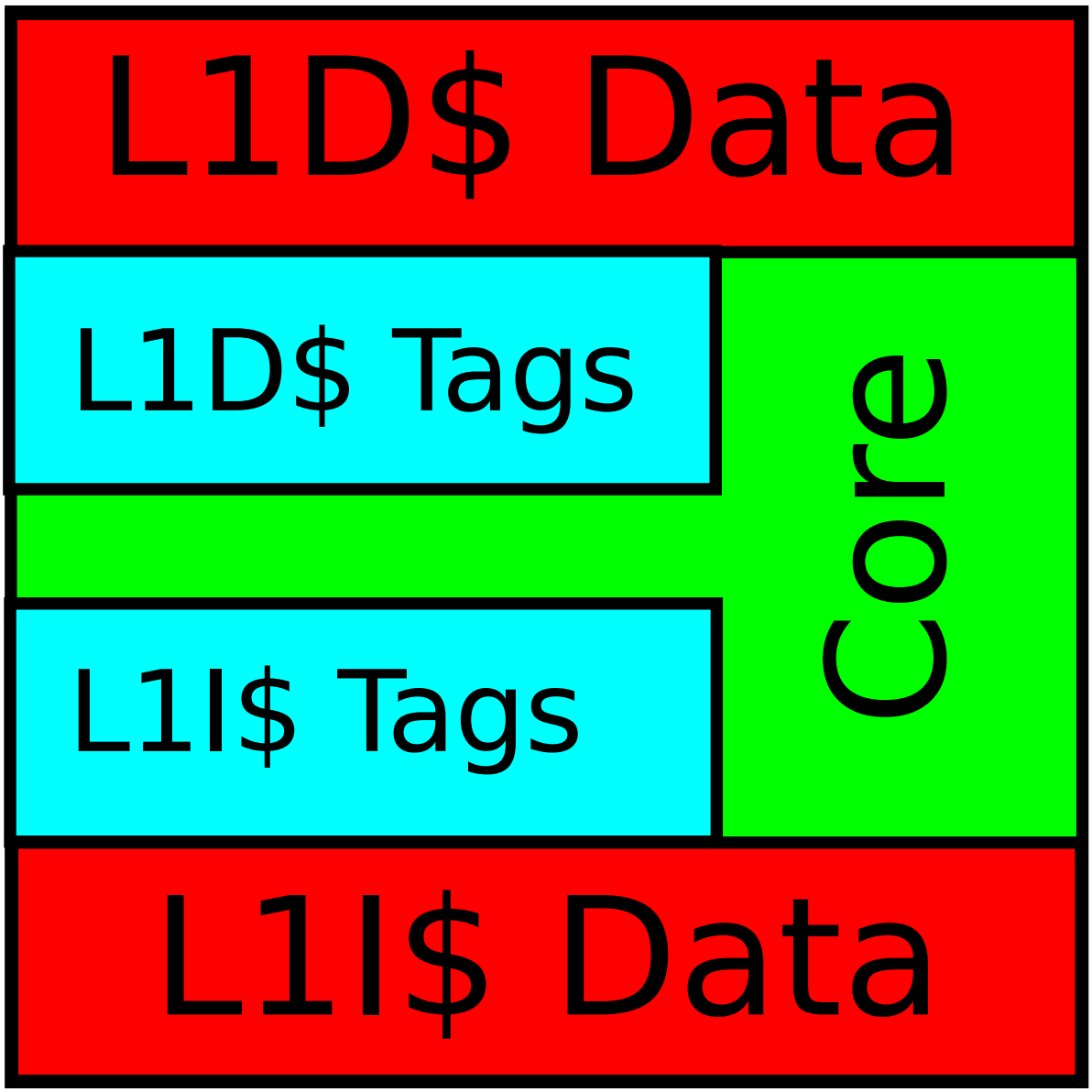

Figure 2.8: Generalized version of the example floorplan from Figure 2.7

blocks to 75% of the total chip width in order to avoid entirely blocking off the data RAMs on pathological technologies (like SAED32) where Rocket's L1 metadata maps very poorly to the available SRAMs.

Figure 2.9: Concrete version of the example floorplan from Figure 2.7, on SAED32

Power Floorplanning

Almost no work has gone into power floorplanning in PLSI: you just get a single power domain with rings around the outside of the chip's routeable area and a grid inside the rings on the highest metal layers available for regular routing. This power floorplan should be sufficient to evaluate microarchitectures and to build simple chips, but isn't enough to build a big chip.

2.7 Place and Route Tool Driver

Much like the synthesis tool driver, the place and route tool driver takes in various vendor-agnostic file formats, converts them to vendor-specific formats when necessary, runs the vendor's synthesis tool, checks to see that there are no errors, and then produces the results for other steps to use.

The only PAR tool driver I bothered implementing was for ICC, since it is the only one that works with the 32nm Synopsys educational PDK.

Chapter 3

Results

PLSI has been implemented and works for multiple Rocket Chip based designs. This section presents the results for Rocket, BOOM, and Hwacha on Synopsys' 32nm Educational PDK produced via the Synopsys tools. In addition to presenting the results, there is a discussion of the validity of the results.

These results were all run using the latest version of Rocket Chip that supported the various configurations when this thesis was being written. The excat versions of Rocket Chip differ between the various targets because all the forks aren't up to date at all times, but they're all tagged as git submodules in the thesis repository. Newer versions of Rocket Chip have removed support for the L2 cache, which is why the Hwacha configuration (which is based on an older version of Rocket Chip) is the only configuration that has an L2 cache attached.

Rocket

```
{
  "clocks": [
    {
      "name": "clock",
      "period": "1250ps",
      "par derating": "250ps"
    }
  ],
  "scenarios": [
    {
      "corner": {
        "nmos": "typical",
        "pmos": "typical",
        "temperature": "25 C"
      },
      "supplies": {
        "VDD": "1.05 V",
        "GND": "0 V"
      }
    }
  ]
}
```

Figure 3.1: PLSI CAD Configuration for DefaultConfig

```
class RocketTilePlacer:
  def __init__(self, config):
    self.top = None
    self.l1dd = []
    self.l1dm = []
    self.l1id = []
    self.l1im = []

  def insert(self, macro):
    if macro.matches(config.rtl_top):
      self.top = TopMacro(macro.name, macro.width, macro.height)
    # Version Break
    elif macro.matches("coreplex/RocketTile/DCache/data"):
      self.l1dd.append(macro)
    elif macro.matches("coreplex/RocketTile/DCache/MetadataArray"):
      self.l1dm.append(macro)
    elif macro.matches("coreplex/RocketTile/icache/icache"):
      self.l1id.append(macro)
    elif macro.matches("coreplex/RocketTile/icache/icache/tag_array/tag_array"):
      self.l1im.append(macro)
    # Version Break
    elif macro.matches("coreplex/rocketTiles/dcache/data"):
      self.l1dd.append(macro)
    elif macro.matches("coreplex/rocketTiles/dcache/MetadataArray"):
      self.l1dm.append(macro)
    elif macro.matches("coreplex/rocketTiles/frontend/icache"):
      self.l1id.append(macro)
    elif macro.matches("coreplex/rocketTiles/frontend/icache/u"):
      self.l1id.append(macro)
    elif macro.matches("coreplex/rocketTiles/frontend/icache/tag_array/tag_array"):
      self.l1im.append(macro)
    else:
      print("%s cannot be matched" % macro.name)
      return False
    return True

  def list_constraints(self):
    l1dd = TopLeftPlacer   (self.top, self.top.tlf(), sorted(self.l1dd), True)
    l1dm = TopLeftPlacer   (self.top, l1dd.bl(),      sorted(self.l1dm), False)
    l1id = BottomLeftPlacer(self.top, self.top.blf(), sorted(self.l1id), True)
    l1im = BottomLeftPlacer(self.top, l1id.tl(),      sorted(self.l1im), False)
    return l1dd.place() + l1dm.place() + l1id.place() + l1im.place()
```

Figure 3.2: PLSI Floorplan for DefaultConfig

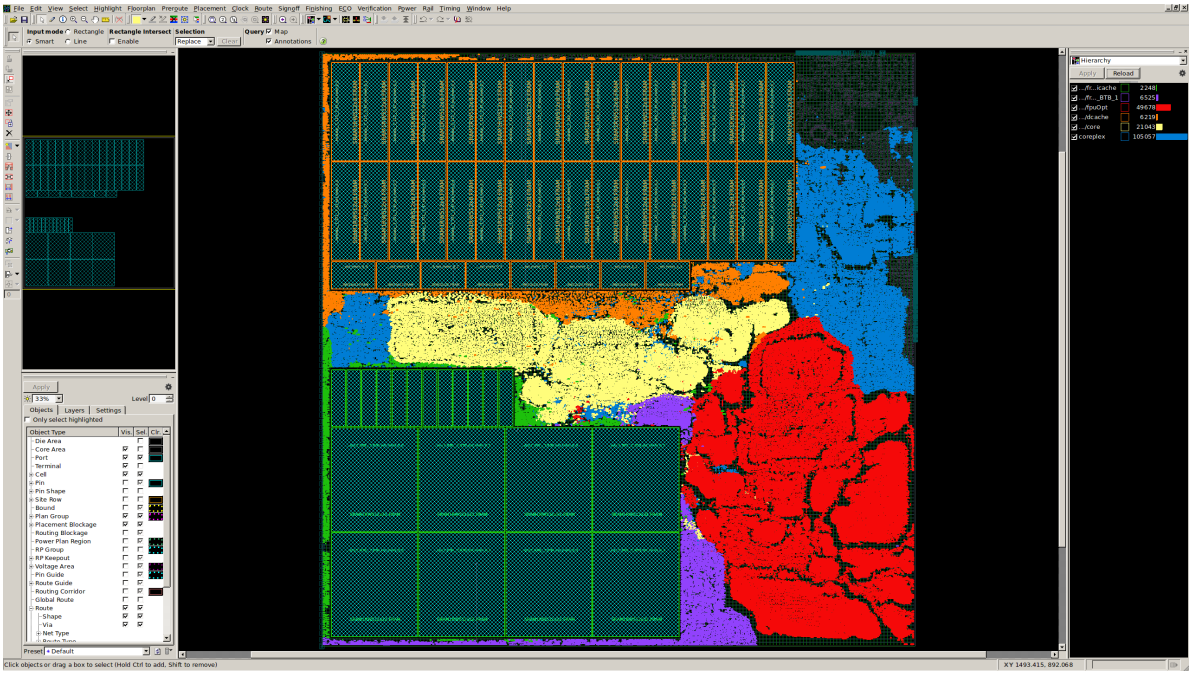

Figure 3.3: ICC Floorplan for DefaultConfig

```
Timing Path Group 'clock'
-----------------------------------
Levels of Logic:              34.00
Critical Path Length:          1.52
Critical Path Slack:          -0.16
Critical Path Clk Period:      1.35
Total Negative Slack:        -18.73
No. of Violating Paths:      988.00
Worst Hold Violation:          0.00
Total Hold Violation:          0.00
No. of Hold Violations:        0.00
-----------------------------------

Cell Count
-----------------------------------
Hierarchical Cell Count:        609
Hierarchical Port Count:      18116
Leaf Cell Count:             115987
Buf/Inv Cell Count:           16638
Buf Cell Count:                7235
Inv Cell Count:                9403
CT Buf/Inv Cell Count:          689
Combinational Cell Count:     94266
Sequential Cell Count:        21721
Macro Count:                     60
-----------------------------------
```

```
Area
-----------------------------------
Combinational Area:   258124.149879
Noncombinational Area:
                      144030.016922
Buf/Inv Area:          48915.350153
Total Buffer Area:         31126.03
Total Inverter Area:       17789.32
Macro/Black Box Area:
                     1009378.968750
Net Area:             381408.759989
Net XLength        :     2435133.75
Net YLength        :     2365051.00
-----------------------------------
Cell Area:           1411533.135552
Design Area:         1792941.895541
Net Length         :      4800185.00


Design Rules
-----------------------------------
Total Number of Nets:         125786
Nets With Violations:            316
Max Trans Violations:             14
Max Cap Violations:              308
```

Figure 3.4: ICC QoR Report for DefaultConfig

```
{
  "clocks": [
    {
      "name": "clock",
      "period": "1600ps",
      "par derating": "400ps"
    }
  ],
  "scenerios": [
    {
      "corner": {
        "nmos": "typical",
        "pmos": "typical",
        "temperature": "25 C"
      },
      "supplies": {
        "VDD": "1.05 V",
        "GND": "0 V"
      }
    }
  ]
}
```

Figure 3.5: PLSI CAD Configuration for SmallBOOMConfig

BOOM

```
class BoomTilePlacer(RocketTilePlacer):
  def __init__(self, config):
    self.top = None
    self.l1dd = []
    self.l1dm = []
    self.l1id = []
    self.l1im = []
    self.l1ht = []
    self.l1pt = []
    self.l1ei = []

  def insert(self, macro):
    if macro.matches(config.rtl_top):
      self.top = TopMacro(macro.name, macro.width, macro.height)
    # Version Break
    elif macro.matches("coreplex/BOOMTile/DCache/data"):
      self.l1dd.append(macro)
    elif macro.matches("coreplex/BOOMTile/DCache/MetadataArray"):
      self.l1dm.append(macro)
    elif macro.matches("coreplex/BOOMTile/HellaCache/MetadataArray"):
      self.l1dm.append(macro)
    elif macro.matches("coreplex/BOOMTile/HellaCache/meta"):
      self.l1dm.append(macro)
    elif macro.matches("coreplex/BOOMTile/icache/icache"):
      self.l1id.append(macro)
    elif macro.matches("coreplex/BOOMTile/icache/icache/u"):
      self.l1id.append(macro)
    elif macro.matches("coreplex/BOOMTile/HellaCache/data"):
      self.l1dd.append(macro)
    elif macro.matches("coreplex/BOOMTile/icache/icache/tag_array/tag_array"):
      self.l1im.append(macro)
    elif macro.matches("coreplex/BOOMTile/core/bpd_stage/br_predictor/counters/p_table/p_table/p_table"):
      self.l1pt.append(macro)
    elif macro.matches("coreplex/BOOMTile/core/bpd_stage/br_predictor/counters/h_table/h_table/h_table"):
      self.l1ht.append(macro)
    elif macro.matches("coreplex/BOOMTile/core/bpd_stage/br_predictor/brob/entries_info/entries_info"):
      self.l1ei.append(macro)
    else:
      print("%s cannot be matched" % macro.name)
      return False
    return True

  def list_constraints(self):
    l1dd = TopLeftPlacer   (self.top, self.top.tlf(), sorted(self.l1dd), True)
    l1dm = TopLeftPlacer   (self.top, l1dd.bl(),      sorted(self.l1dm), False)
    l1ht = BottomLeftPlacer(self.top, self.top.blf(), sorted(self.l1ht), True)
    l1pt = BottomLeftPlacer(self.top, l1ht.tl(),      sorted(self.l1pt), False)
    l1id = BottomLeftPlacer(self.top, l1pt.tl(),      sorted(self.l1id), False)
    l1im = BottomLeftPlacer(self.top, l1id.tl(),      sorted(self.l1im), False)
    l1ei = BottomLeftPlacer(self.top, l1im.tl(),      sorted(self.l1ei), False)
    return l1dd.place() + l1dm.place()  + l1ht.place() + l1pt.place() + l1id.place() + l1im.place() + l1ei.place()
```

Figure 3.6: PLSI Floorplan for SmallBOOMConfig

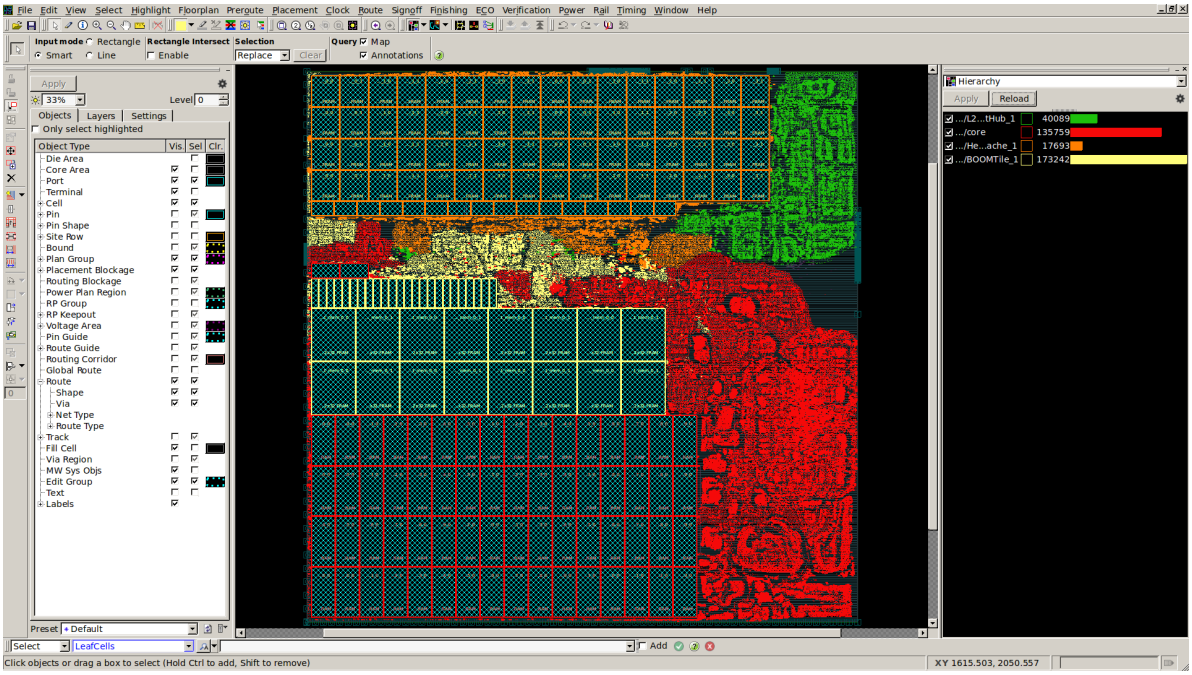

Figure 3.7: ICC Floorplan for SmallBOOMConfig

```
Timing Path Group 'clock'
-----------------------------------
Levels of Logic:                53.00
Critical Path Length:            3.56
Critical Path Slack:            -1.56
Critical Path Clk Period:        2.00
Total Negative Slack:        -1282.67
No. of Violating Paths:      10636.00
Worst Hold Violation:            0.00
Total Hold Violation:            0.00
No. of Hold Violations:          0.00
-----------------------------------

Cell Count
-----------------------------------
Hierarchical Cell Count:         1165
Hierarchical Port Count:        76737
Leaf Cell Count:               226528
Buf/Inv Cell Count:             39474
Buf Cell Count:                 20757
Inv Cell Count:                 18717
CT Buf/Inv Cell Count:           2089
Combinational Cell Count:      187351
Sequential Cell Count:          39177
Macro Count:                      186
-----------------------------------

Area
-----------------------------------
Combinational Area:   548249.859481
Noncombinational Area:
                      260366.723512
Buf/Inv Area:         141541.690599
Total Buffer Area:         94002.53
Total Inverter Area:       47539.16
Macro/Black Box Area:
                     3912562.714844
Net Area:            1230432.321193
Net XLength        :     7458008.00
Net YLength        :     7564863.50
-----------------------------------
Cell Area:           4721179.297837
Design Area:         5951611.619030
Net Length         :    15022872.00
```

Figure 3.8: ICC QoR Report for SmallBOOMConfig

```
{
  "clocks": [
    {
      "name": "clock",
      "period": "2500ps",
      "par derating": "2500ps"
    }
  ],
  "scenerios": [
    {
      "corner": {
        "nmos": "typical",
        "pmos": "typical",
        "temperature": "25 C"
      },
      "supplies": {
        "VDD": "1.05 V",
        "GND": "0 V"
      }
    }
  ]
}
```

Figure 3.9: PLSI CAD Configuration for EOS24Config

Hwacha (with L2)

3.1 Discussion

The results in this section and meant to be suitable for doing computer architecture research, not for building an actual chip. As such they skip the signoff sections of the flow that, from my experience when building chips, don't have meaningful effects on QoR but take a lot of time to get right.

The simplest configuration run for this thesis was the default Rocket configuration, DefaultConfig. This contains Rocket, 64-bit a 5 stage in-order core, with 32 KiB split L1 caches and a floating-point unit. This configuration and floorplan gets very good performance (1.5GHz) on commercial 28nm technologies, but it appears that the 32nm Synopsys EDK is significantly slower.

The PLSI configuration used to generate DefaultConfig is shown in Figure 3.1 and is pretty boring: there's a little bit of derating for PAR, but that's just about as good as I'm able to do for anything. The post-PAR results show that it's close to meeting timing, and

```
class HwachaTilePlacer(RocketTilePlacer):
  def __init__(self, config):
    self.top = None
    self.l1dd = []
    self.l1dm = []
    self.l1id = []
    self.l1im = []
    self.hid = []
    self.him = []
    self.hrf = []
    self.l2d = []
    self.l2m = []

  def insert(self, macro):
    if macro.matches(config.rtl_top):
      self.top = TopMacro(macro.name, macro.width, macro.height)
    elif macro.matches("DefaultCoreplex/tiles/HellaCache/data"):
      self.l1dd.append(macro)
    elif macro.matches("DefaultCoreplex/tiles/HellaCache/meta"):
      self.l1dm.append(macro)
    elif macro.matches("DefaultCoreplex/tiles/icache/icache"):
      self.l1id.append(macro)
    elif macro.matches("DefaultCoreplex/tiles/icache/icache/u"):
      self.l1id.append(macro)
    elif macro.matches("DefaultCoreplex/tiles/icache/icache/tag_array/tag_array"):
      self.l1im.append(macro)
    elif macro.matches("DefaultCoreplex/tiles/Hwacha/icache/icache"):
      self.hid.append(macro)
    elif macro.matches("DefaultCoreplex/tiles/Hwacha/icache/icache/u"):
      self.hid.append(macro)
    elif macro.matches("DefaultCoreplex/tiles/Hwacha/icache/icache/tag_array/tag_array"):
      self.him.append(macro)
    elif macro.matches("DefaultCoreplex/tiles/Hwacha/vus/vxuInst/laneInst/bankInst/rfInst/HwSRAMRF/HwSRAMR"):
      self.hrf.append(macro)
    elif macro.matches("DefaultCoreplex/tiles/Hwacha/vus/vxuInst/laneInst/bankInst/rfInst/HwSRAMRF/HwSRAMRF"):
      self.hrf.append(macro)
    elif macro.matches("DefaultCoreplex/L2HellaCacheBank/data_array/array"):
      self.l2d.append(macro)
    elif macro.matches("DefaultCoreplex/L2HellaCacheBank/meta/meta"):
      self.l2m.append(macro)
    else:
      print("%s cannot be matched" % macro.name)
      return False
    return True

  def list_constraints(self):
    l1dd = TopLeftPlacer   (self.top, self.top.tlf(), sorted(self.l1dd), True,  0.2)
    l1dm = TopLeftPlacer   (self.top, l1dd.bl(),      sorted(self.l1dm), False, 0.2)
    l1id = TopLeftPlacer   (self.top, l1dm.bl(),      sorted(self.l1id), False, 0.2)
    l1im = TopLeftPlacer   (self.top, l1id.bl(),      sorted(self.l1im), False, 0.2)
    hrf  = BottomLeftPlacer(self.top, self.top.blf(), sorted(self.hrf),  True,  0.2)
    hid  = BottomLeftPlacer(self.top, hrf.tl(),       sorted(self.hid),  False, 0.2)
    him  = BottomLeftPlacer(self.top, hid.tl(),       sorted(self.him),  False, 0.2)
    l2d  = TopRightPlacer  (self.top, self.top.trf(), sorted(self.l2d),  True,  0.8)
    l2m  = TopRightPlacer  (self.top, l2d.br(),       sorted(self.l2m),  False, 0.8)
    return l1dd.place() + l1dm.place() + l1id.place() + l1im.place() + hrf.place() + hid.place() + him.place() + l2d.place() + l
```

Figure 3.10: PLSI Floorplan for EOS24Config

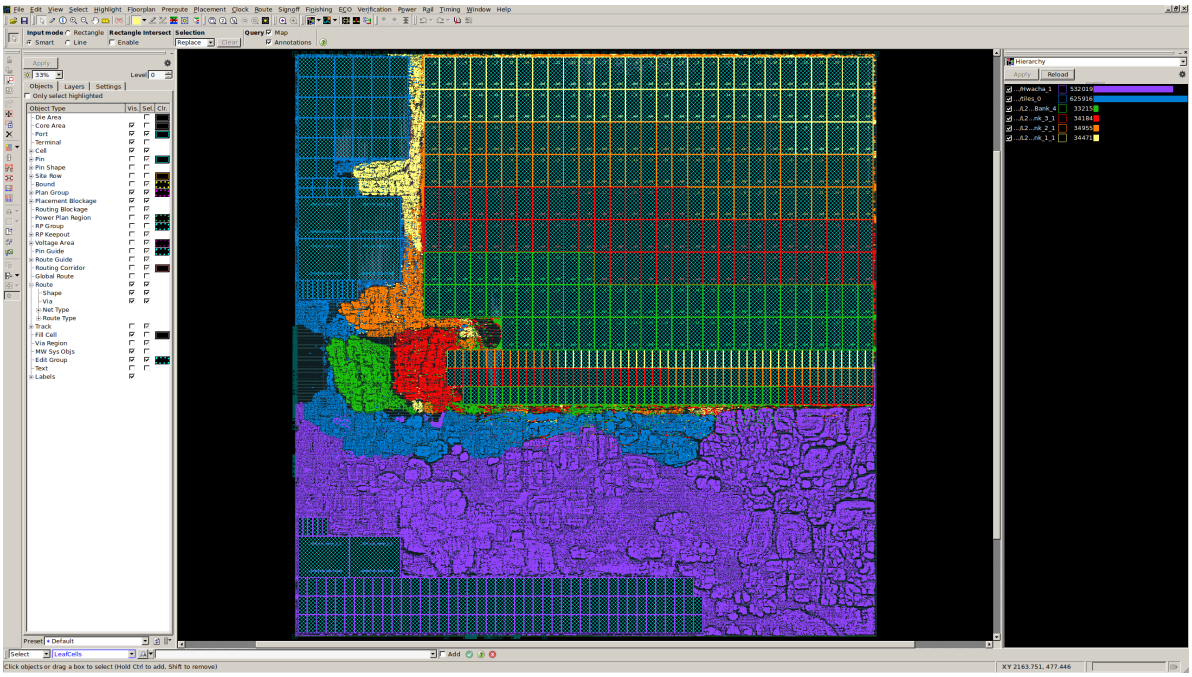

Figure 3.11: ICC Floorplan for EOS24Config

that there aren't too many DRC errors so the results are probably believable.

The Hwacha results are, however, a whole different story. As you can see from the PLSI configuration file, the physical design here gets very bad QoR – there's a 50% clock rate difference between the post-PAR and post-synthesis results. While some of this can be attributed to Rocket Chip's memories not mapping well to the 32nm EDK's SRAMs (which have to write masks), a significant part of this is probably due to the poor floorplan quality. From looking at Figure 3.11, you can see how the floorplan generated from Figure **??** is quite bad: you can see how the L2 is too wide and too short, which results in there being no space for the actual core logic.

```
Timing Path Group 'clock'
-----------------------------------
Levels of Logic:              29.00
Critical Path Length:          5.63
Critical Path Slack:          -0.73
Critical Path Clk Period:      5.00
Total Negative Slack:         -2.61
No. of Violating Paths:       27.00
Worst Hold Violation:          0.00
Total Hold Violation:          0.00
No. of Hold Violations:        0.00
-----------------------------------

Cell Count
-----------------------------------
Hierarchical Cell Count:       3517
Hierarchical Port Count:     145207
Leaf Cell Count:             786039
Buf/Inv Cell Count:          144033
Buf Cell Count:               65811
Inv Cell Count:               78222
CT Buf/Inv Cell Count:         3501
Combinational Cell Count:    686589
Sequential Cell Count:        99450
Macro Count:                    608
-----------------------------------
```

```
Area
-----------------------------------
Combinational Area:  1875823.402210
Noncombinational Area:
                      659483.367816
Buf/Inv Area:         463893.370644
Total Buffer Area:        283467.90
Total Inverter Area:      180425.47
Macro/Black Box Area:
                    10255909.371094
Net Area:            3823554.904260
Net XLength        :    28303776.00
Net YLength        :    25156786.00
-----------------------------------
Cell Area:          12791216.141120
Design Area:        16614771.045380
Net Length         :     53460560.00


Design Rules
-----------------------------------
Total Number of Nets:         850635
Nets With Violations:          13080
Max Trans Violations:           1133
Max Cap Violations:            12164
```

Figure 3.12: ICC QoR Report for EOS24Config

Chapter 4

Conclusion

The goal of producing PLSI was to make it easy to generate VLSI results for computer architecture research, and I think it's at least gotten part of the way there. Right now the biggest obstacle in using PLSI is that the semi-automated floorplanning flow hasn't been pushed on enough to be solid.

4.1 Future Work

A Useable Educational Technology

I was limited to using the Synopsys 32nm EDK to evaluate PLSI for this thesis. Unfortunately, optimizing designs for this technology results in design that don't match any real technology I've seen because the cost of some key structures doesn't match real technologies.

In order to actually be able to do RTL-based computer architecture research without relying on the benevolence of other research groups, we need an educational technology that produces results that match those produced by real designs. Since we're using this for computer architecture research instead of VLSI or CAD research, this doesn't actually need things like a realistic DRC deck.

While I think this the most important future work for PLSI, I have no idea how to go about solving this.

Benchmarking and Power

If you've been paying attention then you'll probably notice that there aren't any results from actually simulating the designs that were pushed through the tools. This was mostly for time reasons (as far as I know PrimeTime Power works and has been used by other people, but Donggyu fixed it after I'd ran out of time to run results), but also because it appears that the 32nm Synopsys EDK's power models wouldn't be good enough to get any reasonable information from. There's two problems here: the lack of write masks means many more

SRAM macros are emitted than is reasonable, and I get a lot of warnings from the tools about SRAMs not having power models.

DRC and LVS Fixing

I was hoping to have my designs LVS clean and set a low threshold for DRC errors, but it looks like the 32nm Synopsys EDK has some problems that prevent you from producing DRC clean results using it – I specifically ran into a lot of nets with redundant via errors that looked like some sort of tech file vs DRC deck mismatch. While I tried a few ways to fix this, I didn't have enough time to get it fixed properly so just gave up.

We've had these sorts of problems on real technologies as well, so I think this actually warrants writing a proper tool. This tool would read DRC decks, technology files, and a set of DRC errors. It would then modify the technology file in order to elide these DRC errors.

Clock Estimation

Even when using the PLSI flow, users still have to manually write clock constraints for every clock in their design. This is an impediment when doing RTL-based computer architecture research as you can't know if each of your design points are producing reasonable QoR without performing some manual optimization on each of them.

Based on my experience trying to get good QoR from our designs, it should be possible to apply some simple hueristics to our RTL inputs in order to determine the achievable clock peroid of a design to with 25%, which is good enough that the CAD tools won't blow up when asked to meet that timing constraint.

Trace Based SRAM Selection

Given a well designed microprocessor, the most important constraint should be how it maps to SRAMs. Even when using PLSI this process still requires manual intervention: while PLSI might make the selection of SRAMs automatic, the cost function still requires an engineer to intuit how memories should be mapped given their design constraints. For PLSI this was easy: we just select the fastest SRAM every time – now that we're past the end of Dennard Scaling this isn't the right thing to do.

The right way to fix this is to use simulation traces to drive the cost function for SRAM selection, which will actually select the SRAM configuration the user desires. This is really just the same as trace-based synthesis, but for SRAMs.

An Accuracury vs Effort Curve for Physical Design

While some might consider GDS the holy grail of accuracy when evaluating designs, getting accurate results actually requires significant effort. It would be interesting to attempt to

optimize extant designs to see how accurate their results actually are, and to compare these results with those produced by methodologies that requires less effort.