SAND2012-1360C

Using a Custom-Built HDL For Printed Circuit Board Design Capture

Brent Nelson, Brad Riching, and Richard Black
Department of Electrical and Computer Engineering
Brigham Young University, Provo, Utah 84602
Email: nelson@ee.byu.edu, bradriching@gmail.com, aeldstesort@gmail.com

***Abstract*—The use of Hardware Description Languages (HDL's) for printed circuit board design is presented and its advantages over schematic capture are detailed. One such HDL called PHDL is presented and its use demonstrated through a set of sample designs.**

***Index Terms*—Printed circuit board, PCB, HDL, schematic, design, EDA, CAD**

I. Introduction

The creation of Hardware Description Languages in the 1980's forever changed digital design at the gate and transistor levels by replacing schematic entry with textual design based on formally defined languages. Initially these languages were the proprietary creations of individual companies; now known as Hardware Description Languages or HDL's, they were eventually standardized by the IEEE and now bear names like Verilog and VHDL[1].

Today, virtually all integrated circuit and FPGA design is done using textual design entry in HDL's. The benefits of this are many: increased designer productivity, greater support for collaboration and design version management, greater design reuse, and the availability of a much richer CAD tool ecosystem for design.

However, the Printed Circuit Board (PCB) industry has largely failed to capitalize on the many advantages provided by HDL-based system design and continues to rely almost exclusively on graphical design entry tools. In the process, they miss out on the many advantages provided by HDL-based design entry.

This paper argues for the use of HDL's for PCB design capture and that the vast array of advantages provided far outweighs any perceived disadvantages incurred in moving away from graphical design entry. It will also introduce the PHDL language and associated CAD tool suite, an open-source CAD tool developed for PCB design capture.

PCB design consists of two main steps: (1) the use of a schematic editor to graphically draw the *connectivity* of the PCB components, and then (2) the use of a drafting tool to complete the physical PCB *layout* and prepare the required manufacturing data files. A netlist is typically used to communicate the results of the first step to the second step. This overall design process is shown in Figure 1.

This research was supported by Sandia National Laboratories under PO Contract 1125274, sponsored by Sandia Corporation (a wholly owned subsidiary of Lockheed Martin Corporation) as Operator of Sandia National Laboratories under its U.S. Department of Energy Contract No. DE-AC04-94AL85000.

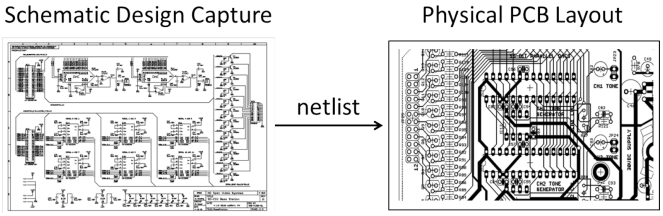

Fig. 1. The Two-Step PCB Design Process

A similar two-step process exists in conventional IC or FPGA digital design. However, the first step (RTL design capture) is done using an HDL (VHDL or Verilog) and the second step (physical design) is done using different tools (layout, placement, and routing tools). This work proposes that the first PCB design step (design entry) be completed using an HDL specially created for this purpose, while the second step (physical design) continues to use the existing PCB physical design tool flow. Thus, this paper proposes to use HDL's for only the left box in Figure 1.

The paper will begin by discussing the advantages of textual input for PC board design capture, comparing and contrasting it with current practice. It then will introduce the PHDL language and, through a series of graduated code examples, illustrate the language's features for describing PC board device connectivity. An overview of the associated PHDL CAD tool suite is then provided to show how an HDL-based CAD tool flow is used for PC board design. Finally, conclusions are given along with suggestions for future work.

II. Graphical Design Entry for PC Board Design

Schematic capture using a computer program imitates the manual drawing of schematics on paper to reflect the selection of electronic components and their interconnection using wires. For such schematics, visual inspection is the preferred mechanism for understanding connectivity and circuit function. However, such an approach is *non-scalable*, meaning that it breaks down for large designs. For example, consider the insertion of a high pin count device such as an FPGA chip into a PCB design as shown in Figure 2. The large size of the component is due to the large number of pins present around

its periphery; this large size precludes the showing of more than a few additional components and their interconnections on this particular schematic sheet.

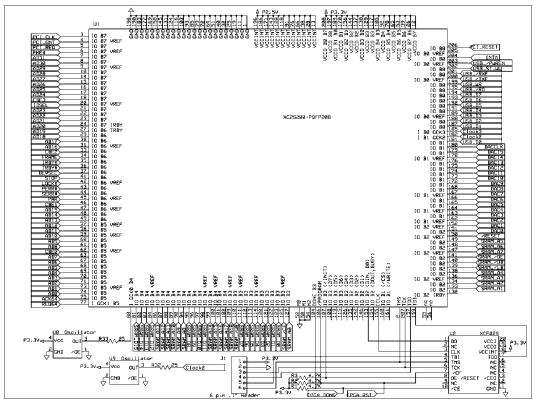

Fig. 2. A High Pin Count Device In A Schematic

Worse, consider the schematic page shown in Figure 3. In this case, the high-density mezzanine connector has so many pins its I/O banks are divided into multiple pieces, some of which are not even shown in the figure, and that span multiple sheets. As before, the complexity of the circuit component leaves little room for other relevant circuit components and their interconnections on this particular schematic page.

To interconnect pins from either of these examples to the pins of other components found on other schematic pages, the following method is typically used: (a) each pin on each component has a small wire stub attached to it and (b) each such wire stub is labelled with a signal name. Thus, the use of matching names for different wire stubs on different schematic sheets implies wire connections between the corresponding pins.

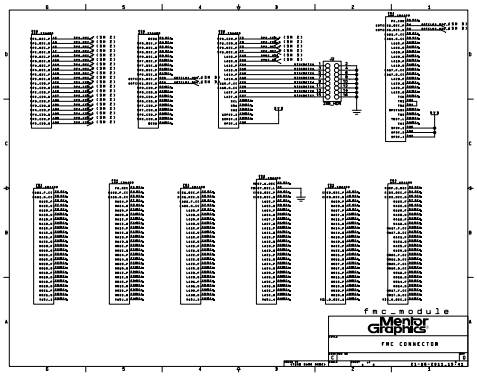

Fig. 3. A High Pin Count Device Broken Into Pieces in the Schematic

To claim that this spreading of devices across multiple schematic sheets and using name matching on wire stubs to verify connectivity is somehow *natural* and *efficient* for entering and verifying a design is a stretch — any geometric information present in a smaller schematic (which may fit onto a page or two) is completely lost for large designs. Further the act of inserting the many required wire stubs and successfully labelling them represents a huge time burden for a designer and the source of hundreds or even thousands of potential design errors.

Schematic capture approaches have a number of additional limitations. First, schematic design tools are typically proprietary and employ proprietary design database formats. This precludes third party development of any kind of add-on tools to help with the design process. Their design databases are also typically in a binary format, preventing the use of source code control systems such as SVN or CVS for managing design files and tracking changes. The same goes for the myriad of other productivity-enhancing tools present in software development environments (IDE's, etc).

III. HDL-Based PCB Design

HDL's have a number of important advantages over schematic capture for PCB design:

- HDL files are plain text and are expressed in an open and documented format — they require no sophisticated or proprietary design tools to manipulate. Simple text editing functions such as *copy-paste* and *search-and-replace* can be used during their editing and processing as well as simple command line tools such as *diff* and *grep*.
- HDL files are printable, share-able, and email-able, facilitating collaboration and communication amongst a team of designers.
- A language-based approach readily supports higher level constructs such as arrays, structures, lists, etc. for declaring and manipulating design elements as well as reference designators. This allows the designer to work at a higher level of abstraction, increasing design productivity.
- HDL files lend themselves to the use of source code control systems such as SVN or CVS, which store design files in a central repository and facilitate sharing among a group of designers. They further allow source code changes to be individually tracked and documented, allowing any version of any design file to be *checked out* for use. They support the intelligent merging of different file version content to support extensive sharing and reuse. They also allow for the tagging of *file sets* which represent logical design versions (such as all the files that make up version 1.1 of a given PCB).
- HDL files lend themselves to software development-oriented IDE's (integrated development environments) such as Eclipse. One role of IDE's is to further simplify the management of the design files using the previously-mentioned source code control systems. They also can provide automation for running the needed compilation tools to convert design source files into netlists, bill of materials, etc.
- IDE's can provide real-time syntax checking and error reporting as file editing is done. They provide syntax highlighting (coloring) to aid the programmer/designer in understanding the structure of the design code. They

provide syntax auto-completion to minimize the required typing to enter a design.
- IDE's can provide sophisticated cross-probing capabilities such as moving the cursor to the original declaration of a net which is connected to a specific component pin. Another example would be the user selecting a pin on a device instance and asking the IDE to highlight *all* other pins in the source tree wired to that same pin or generating a text listing with line numbers of those pins.
- Finally, open language specifications allow for the creation of a variety of design checking tools such as DRC, ERC, and even linting tools as are found in most software development environments.

In light of these significant advantages, the use of HDL's for PCB design has a number of clear advantages over schematic entry tools. These advantages were recognized long ago in the ASIC and FPGA design communities, and they will only continue to grow in importance as designs grow in complexity and as device pin counts increase.

IV. Introduction to PHDL

The Printed Circuit Board Hardware Description Language (PHDL) is a domain specific language specially designed for PCB design capture (the left half of Figure 1). PHDL exhibits all the above-mentioned benefits of a text-based design tool and addresses additional problems that arise in the use of graphical tools. Its greatest strength lies in its simplicity of syntax and data entry.

A. PHDL Syntax and Language Structure

The ***design*** construct is used in PHDL to describe the overall PCB — it simply lists the device instances in the design along with the wires (nets) that interconnect them. Since a designer will typically create device declarations before starting to specify a design, we will start by describing PHDL device declarations.

B. PHDL Device Declarations

PHDL ***device*** declarations are used to describe the library components that will be placed onto the board. A typical device declaration is shown in Program IV.1.

Program IV.1 A Basic Device Declaration

```
device seven_seg {
  attr library = "led";
  attr package = "ss4061";
  attr refPrefix = "DSP";
  pin common = {1};
  pin[1:7] segments = {2,3,4,5,6,7,8};
  pin dp = {9};
}
```

The device declaration includes the device's name. The body of the declaration then defines a list of attributes associated with the device. In the example given, the attributes specify the name of the layout footprint library the device is a part of as well as the package type for the device. It also provides a reference designator (refdes) prefix string which will be used for all devices of this type instantiated on the board (with unique instance numbers being appended to this prefix by the compilation tools). The set of attributes that can be declared for a device is limitless — the attribute names are arbitrary identifiers and the attribute values are arbitrary strings. Thus, different devices will have different attributes associated with them such as 'manufacturer', 'cost', etc. All of the specified attributes ultimately will appear in the generated Bill of Materials for the design.

The device declaration also includes definitions of the names of the device's pins. The advantage of naming pins this way is that they can later be referred to by name rather than by pin number. This also makes it possible to refer to a multi-bit pin by its name. In this example the pin named *common* is mapped to physical pin *1* on the device, the *segments* pin is mapped to physical pins *2-8* on the device, and the *dp* pin is mapped to physical pin *9* on the device. These pin numbers would presumably be taken from the data sheet or physical footprint found in the layout database.

Note that in this example the *segments* pin is declared as a bus with bits numbered *1* to *7* when viewed from left to right. Any two indices can be used in this notation to specify the range for a multi-bit pin — the indices can increase from left to right as in this example or they could decrease from left to right as in:

```
pin[6:0] segments = {2,3,4,5,6,7,8};
```

Also, note that a comma-separated list is used to specify the set of physical pins they map to on the right side of the equals sign. Any combination of letters, numbers and limited special characters (+, -, $, etc.) may be used to represent pin mappings.

C. PHDL Design Blocks

A ***design*** block is a collection of device instances along with the nets that interconnect them. In PHDL, the top-level design ***is*** the final PCB. An example design declaration is given in Program IV.2.

This design contains three nets (*vcc*, *gnd*, and *lr_net*) and three device instantiations. Nets are declared inside of the design block in a similar fashion to pins but using the ***net*** keyword, either as individual bits or as a multi-bit net using a range specifier.

Devices previously defined using the device construct are instantiated inside the design using the ***inst*** keyword. Device instances consist of an instance name and the name of a previously declared device. The body of the inst construct then contains a set of pin assignments.

A single copy of the *battery_9V* device is first instantiated and its *pos* pin is tied to design net named *vcc* and its *neg* pin is tied to design net *gnd* (both of these nets must have been previously declared).

Next, an array of 8 *led_red* devices is instantiated using array-like syntax to specify that 8 such devices are desired.

Program IV.2 A Simple Design

```
design led_circuit {
  net vcc, gnd;
  net[1:8] lr_net;

  inst pwrsup of battery_9V {
    pos = vcc;
    neg = gnd;
  }
  inst(1:8) led of led_red {
    a = vcc;
    combine(k) = lr_net;
  }
  inst(1:8) res of resistor {
    combine(a) =lr_net;
    b = gnd;
    value = "330";
  }
}
```

Program IV.3 Slicing and Dicing Indices

```
// Tie all 8 'a' pins to signal 'vcc'
a = vcc;

// Combine the 'k' pins from all 8 instances
// into an 8-bit bus and tie it to the 8-bit
// net named 'lr_net'
combine(k) = lr_net;

// Tie the 'a' pin of instance number 3
// to net 'gnd'
this(3).a = gnd;

// Tie the 'a' pins of the odd numbered
// instances to the right half of 'lr_net'
this(1,3,5,7).a = lr_net[5:8];

// Tie the 'a' pins of the even numbered
// instances to the other half of 'lr_net'
this(2,4,6,8).a = lr_net[3:0];
```

Each one of the 8 LED's *a* pins is then wired to design net *vcc*. This shows the ability to map all of the *a* pins on these 8 devices to a single wire, something commonly done in designs. Next, all of the *k* pins of the 8 different LED's are lined up into an 8-bit bus (using the ***combine*** keyword) and these are then connected to the 8-bit net called *lr_net*. The semantics of this are that the *k* pin of instance 1 is tied to bit *1* of *lr_net* and so on.

Finally, 8 *resistor* devices are instanced. The new construct shown here is that the attribute *value* is over-ridden from whatever default value it might have been given in its device declaration and set to the value "330" here, illustrating the ability to over-ride attribute values on an instance by instance basis.

D. More On PHDL Device Instantiation

As previously shown, either single device instantiations can be made or an array of multiple instantiations can be made, all with the ***inst*** keyword. In the case of a single device instantiation, the syntax to map its pins to design nets is straightforward. The same goes for changing its attributes as shown in Program IV.2.

However, when instantiating an array of devices, PHDL provides great flexibility in wiring instance pins to nets and over-riding instance attributes. A variety of mechanisms for doing so are shown in Program IV.3. All of of the statements in the code assume that an array of eight *led_red* devices has just been instanced.

Note that when a subset of the instances in an array of instances is to be accessed, it is done using $this(range)$. where $range$ can be a single number, two numbers separated by a colon, or a comma-separated list of numbers. This same flexibility in specifying ranges also extends to the selection of the bits of *lr_net* on the right hand side of each statement.

The last statement in the code of Program IV.3 illustrates how the order of the index range is significant, and that by reversing the range a permutation can be effected. In the very last statement, instance 2's *a* pin is assigned to $lr\_net[3]$, instance 4's *a* pin is mapped to $lr\_net[2]$ and so forth. It is crucial to realize that the assignment rule always treats aggregate bits of *pins* spanning possibly multiple device instances on the left (and similarly aggregate bits of *nets* on the right) as groups lined up in the order specified by the range expression. These two groups are then assigned one-to-one, in a left-to-right fashion. Assignments of mismatching aggregate sizes on the left and right hand side of an assignment statement is not supported.

On the right hand side of assignment statements, a concatenation operator can be used to create new multi-bit nets as in:

```
this(1,3,5,7).a = vcc & lr_net[2] & gnd & gnd;
```

If no selector notation is used at all in an assignment statement, all the instances are modified as they were in Program IV.2 where all of the LED's *a* pins were connected to *vcc* together.

Finally, the same selection mechanism used to assign pins to nets can also be used to over-ride a subset of the instances' attribute values. For example, to make half the resistors one value and half another, the following could be used:

```
this(1,2,3,4).value = "100";
this(5:8).value = "330";
```

E. Using Hierarchy in PHDL Designs

Like all HDL's, PHDL supports the use of hierarchy in design creation. First, the ***subdesign*** keyword is used to create a sub-design as shown in Program IV.4. A sub-design is identical to a design in all respects except two (which have been highlighted in the code). First, it is declared using the ***subdesign*** keyword. Second, it must have one or more ***port*** declarations. The purpose of a port is to provide something to wire the sub-design to when it is instantiated. In this case, there are two port declarations which use similar syntax to net declarations. While these ports are both one-bit wide, in

general they can be of any width. Like a design, this sub-design then instances a number of devices.

Program IV.4 A Sub-Design Declaration

```
subdesign led_block {
  port vcc, gnd;
  net lr_net;

  inst led of led_red {
    a = vcc;
    k = lr_net;
  }

  inst res of resistor {
    a = lr_net;
    b = gnd;
    value = "330";
  }
}
```

A sub-design can then be instanced inside another sub-design or design, similar to how devices are instanced. This is shown in Program IV.5. Comparing it to Program IV.2, it can be seen that the arrays of 8 LEDs and 8 resistors have been replaced with an array of 8 sub-designs (using the ***subinst*** keyword), where each ***subinst*** contains an LED and resistor sub-circuit.

Program IV.5 Instancing a Sub-Design

```
design led_circuit {
  net vcc, gnd;

  inst pwrsup of battery_9V {
    pos = vcc;
    net = gnd;
  }

  subinst(1:8) leds of led_block {
    vcc = vcc;
    gnd = gnd;
  }
}
```

We saw previously that the attributes of device instances can be over-ridden using the $this$ syntax. A similar capability exists for arrays of sub-designs as shown in Program IV.6.

In the highlighted statements of this program, dot notation has been used to reach inside the various sub-designs and modify the *value* attribute of their *res* instances. Use of dot notation to reach down into the design hierarchy and modify device attributes is a powerful capability of PHDL. If two collections of circuitry have different attributes on their constituent devices but otherwise have identical connectivity, they can be converted to a single sub-design, instanced multiple times, and then dot notation used to customize each sub-design instance's device attributes as needed. Further, this dot notation can be used with arbitrary levels of hierarchy in a design. Finally, it is important to realize that PHDL does not allow modification of *connectivity* across levels of hierarchy as this would likely open the door to a plethora of hard-to-find

Program IV.6 Instancing a Sub-Design

```
design led_circuit {
  net vcc, gnd;

  inst pwrsup of battery_9V {
    pos = vcc;
    net = gnd;
  }

  subinst(1:8) leds of led_block {
    vcc = vcc;
    gnd = gnd;
    this(1,3,5,7).res.value = "330";
    this(2,4,6,8).res.value = "660";
  }
}
```

connectivity design errors.

The preceding examples provided show the ease with which arbitrary arrays of device and sub-design instances can be inserted into a design, interconnected, and customized. Repeated or regular structures can be created in just a line or two of code, compared to the hundreds of steps required in a graphical *zoomIn-select-zoomOut-pan-zoomIn-select-...* design tool.

At first glance, some may argue that PHDL is just another netlisting language. However, PHDL provides an environment much more capable than simply expressing one-to-one correspondences between declared primitives and generated connectivity. With all the examples of aggregation, slicing, concatentation, and hierarchy previously shown, PHDL actually infers structure based on implications in the source text. As such, we believe PHDL more closely parallels a structural HDL by providing a higher level of abstraction that can be used to model all aspects of a PCB.

F. The PHDL CAD Tool Flow

The PHDL CAD tool is a compiler, taking PHDL source files as input and producing a netlist and other files for use by back-end physical layout tools. Current formats which can be produced by the compiler include netlists for Mentor Graphics PADS and EAGLE PCB. Additionally, the compiler outputs ancillary files such as a Bill of Materials, a comma-separated Reference Designator Mapping Document, and a Layout Supplementary Information Document.

The Bill of Materials summarizes all device instances used in the design along with any attributes attached to those device instances. The format of the Bill of Materials is a 2D table with one row per device and one column for each unique attribute.

The Reference Designator Mapping Document lists all of the device instance names and their respective reference designators for layout-driven manual back-annotation or debugging.

A part of the syntax of the PHDL language allows for the designer to add comments or informational statements about the various structures in the design (designs, sub-designs, instances, and nets). The Layout Supplementary Information Document is a text document that compiles all these comments and information statements into a readable form for the layout engineer.

Figure 4 shows the PHDL tool flow. The front end (lexer, parser, AST, and tree parser) is built on the ANTLR framework (a compiler creation framework) [2][3]. The back end (analyzer/generator) is custom-written Java program code. In addition to creating a netlist and other files for transmission to the physical layout tools, the compiler can also be instructed via command line switches to create an XML representation of the parsed PHDL data structure for processing by custom tools.

```
$java phdl.Compile <file_name>.phdl [switches]
```

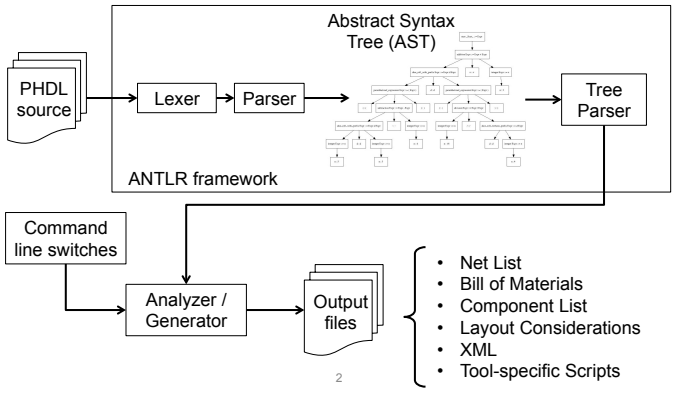

Fig. 4. The PHDL CAD Tool Flow

G. PHDL and High Pin-Count FPGA Devices

FPGAs present many challenges to PCB design for a number of reasons. First, they typically have hundreds of pins which must be declared in a device declaration and then mapped to wires when the FPGA is instantiated. Second, most of their pins are reconfigurable and may frequently change during the design process, requiring both device declarations as well as instantiations to be changed frequently until the design stabilizes.

As previously mentioned, it is difficult to express an instance of an FPGA on a single schematic page while maintaining meaningful circuit context with surrounding hardware. Breaking up a graphical instantiation of an FPGA device to span multiple schematic pages further exacerbates the problem. However, the equivalent FPGA instantiation written in PHDL is extremely compact, especially if many of the pins are declared as bit vectors. Program IV.7 shows an actual FPGA instantiation which is surprisingly compact given it contains some 98 user pins in addition to 46 power, ground, and configuration interface pins.

Additionally, FPGA tools already generate many report files that describe how an HDL FPGA design is synthesized, placed and routed, and how its inputs and outputs are constrained to particular I/O sites [4]. Perhaps less widely known in the PCB community is that these reports contain configuration-specific information about every pin on the FPGA, including how all of the top-level design ports (and their corresponding wires) are bound to specific I/O sites.

Program IV.7 An FPGA Instance in PHDL

```
inst my_fpga of fpga {
  // map all of the power and ground pins to signals
  // angle brackets replicate a one-bit-wide net
  vcco = <vcco_net>;
  vccaux = <vcc_aux_net>;
  vccint = <vcc_int_net>;
  gnd = <gnd_net>;

  // map all of the configuration pins
  cclk = cclk_net;
  done = done_net;
  tck = tck_net;
  tms = tms_net;
  tdi = tdi_net;
  tdo = tdo_net;
  prog_b = prog_b_net;
  m0 = m_net[0];
  m1 = m_net[1];
  m2 = m_net[3];
  hswap_en = hswap_en_net;

  // map all of the user (reconfigurable) pins
  clk_100 = clk_100;
  out_data[31:0] = data_out_bus[31:0];
  in_data[31:0] = data_in_bus[31:0];
  w_en = w_en;
  r_en = r_en;
  leds[7:0] = led_net[7:0];
  switches[7:0] = sw_net[7:0];
  buttons[3:0] = btn_net[3:0];
  segments[6:0] = seg_net[6:0];
  digit_sel[3:0] = anode_net[3:0];
  dp[3:0] = dp_net[3:0];
}
```

The information in these report files can be used to automatically generate a device declaration for a given FPGA and an instantiation template, both of which greatly reduce the typing required by the designer. As the FPGA design changes or I/O pins are moved to different locations, the device declaration and instantiation template can be re-generated and an updated netlist immediately forwarded to the the physical layout tools, often without manual intervention or modifying the PHDL source. Program IV.8 shows a small portion of the generated pinout file from the Xilinx ISE tool flow.

The portion of the file given shows a total of eleven pins, eight of which are assigned to LEDs. The other pins are the FPGA clock, a ground pin, and an unused pin (*P58*).

To leverage this information, a utility called $csv2phdl$ has been developed that reads this file, and automates the process of creating the FPGA device declaration. A portion of the corresponding output from the $csv2phdl$ tool is shown in the code fragment of Program IV.9. Note that the package and library attributes are set to a placeholder value, so that the user may customize them to reflect actual device libraries and footprints in the layout tool of their choice. Also, only one *GND* pin was shown in the CSV file, but $csv2phdl$ accounted for all occurrences in the entire file and consequently generated a *GND* bit vector in the declaration. When this FPGA is instanced, all of the pins in the *GND* pin array may be assigned to the ground net using the *replication* syntax in Program IV.7,

Program IV.8 Partial Pinout From ISE

```
          ⋮
#INPUT FILE:       top_bldc_map.ncd
#OUTPUT FILE:      top_bldc_pad.csv
#PART TYPE:        xc3s400
#SPEED GRADE:      -4
#PACKAGE:          tq144
          ⋮
Pin Number,Signal Name,Pin Usage,Pin Name,...
          ⋮
P50,leds<7>,IOB,IO_L31P_5/D5,OUTPUT...
P51,leds<6>,IOB,IO_L31N_5/D4,OUTPUT...
P52,clk,IOB,IO_L32P_5/GCLK2,INPUT...
P53,leds<5>,IOB,IO_L32N_5/GCLK3,OUTPUT...
P54,,,GND,,,2,,,,,2.50,,,,
P55,leds<4>,IOB,IO_L32P_4/GCLK0,OUTPUT...
P56,leds<3>,IOB,IO_L32N_4/GCLK1,OUTPUT...
P57,leds<2>,IOB,IO_L31P_4/DOUT/BUSY,OUTPUT...
P58,,DIFFS,IO_L31N_4/INIT_B,UNUSED...
P59,leds<1>,IOB,IO_L30P_4/D3,OUTPUT...
P60,leds<0>,IOB,IO_L30N_4/D2,OUTPUT...
          ⋮
```

greatly simplifying how connectivity is expressed for an otherwise repetitive task.

Program IV.9 Partial FPGA Device Declaration

```
device top_tq144 {
  attr refPrefix = "U";
  attr package = "myPkg";
  attr library = "myLib";
  ⋮
  pin[7:0] led = {P50,P51,P53,P55,P56,P57,P59,P60};
  ⋮
  pin[15:0] GND = {P54, ...};
  ⋮
}
```

In addition to the initial generation of the PHDL device declaration as shown, changes made in the HDL can be propagated from the embedded FPGA HDL code to the netlist, and PCB layout. This is important because HDL ports frequently need to be remapped to other pins as the design evolves. To illustrate the advantages of this refactoring process, a small data bus port of the FPGA device declaration is shown in Program IV.10.

As shown, the 4-bit data pins in PHDL were initially assigned to the comma-separated list of pins in the FPGA device, which were created in Xilinx PlanAhead. To alleviate routing congestion in the board layout immediately surrounding the FPGA, several pins were later swapped inside the FPGA I/O constraints file with Xilinx PlanAhead. The $csv2phdl$ utility completely automated the process of generating a new FPGA device declaration to propagate the changes through the PHDL with an updated netlist, and into the layout tool. The resulting new device declaration for the FPGA is shown in

Program IV.10 FPGA Device Declaration

```
device top_tq144 {
  attr refPrefix = "U";
  attr package = "tq144";
  attr mfgr = "XILINX";
  attr partNumber = "xc3s400-4tq144";

  // User I/O pins.
  pin[3:0] data = {P18,P17,P15,P13};
  ...
}
```

Program IV.11.

Program IV.11 Refactored FPGA Device Declaration

```
device top_tq144 {
  attr refPrefix = "U";
  attr package = "tq144";
  attr mfgr = "XILINX";
  attr partNumber = "xc3s400-4tq144";

  // User I/O pins.
  pin[3:0] data = {P13,P15,P17,P18};
  ...
}
```

H. PHDL in a Realistic Design

As an example of the use of PHDL, a proof of concept board design was captured using the PHDL language, a netlist generated, and the physical design then done using EAGLE PCB. The design consists of a multi-axis 32-bit precision motor controller based on a Xilinx Spartan-3 400K FPGA. It has 1MB (512Kb x 16) of on-board RAM and a 2MB flash PROM to store the FPGA configuration. In addition to H-Bridge drivers for brush-motor control, the design also contains circuitry for 2 brushless motor drives. All axes can operate under fully parameterized closed loop Proportional-Integral-Derivative control using high resolution quadrature encoder feedback. The concept board communicates with a host application developed in C# using simple RS232 commands. The top layers of the finished board are shown in Figure 5.

As with any FPGA-centric board, a significant amount of time was spent choosing appropriate FPGA I/O pin constraints to satisfy routing congestion and keep the overall board size to a minimum. With the aid of the $csv2phdl$ utility to automatically generate PHDL device declarations from the Xilinx tool flow, several I/O pin location arrangements were easily tested to find one that minimized board level routing congestion.

The finished board was approximately 4×6 inches in size, had 4 layers, and contained about 200 components. The first batch manufactured were assembled, tested and found to be fully functional and are currently in use at Brigham Young University.

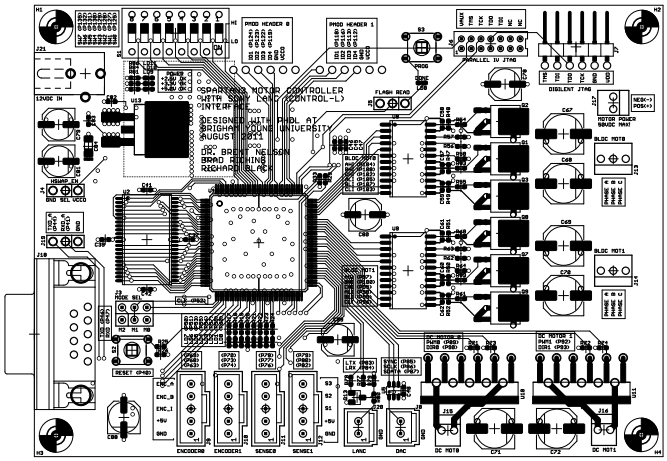

Fig. 5. Motor Controller Board Created Using PHDL

V. Conclusion

This paper has presented the use of an HDL for PCB design, argued and also illustrated by way of example the many advantages an HDL has over graphical schematic entry. In addition, it has introduced PHDL, a new special-purpose HDL for doing PCB design and given some examples of its syntax and structure. A proof of concept board design using PHDL was also described.

The PHDL compiler has been released open source at "http://phdl.sourceforge.net" where either the complete source code or a pre-compiled version of the tool can be downloaded. As it is written in Java, the PHDL compiler will run on any Java-compatible computer. The $csv2phdl$ tool, a tutorial, and set of sample designs are also available at the SourceForge site.

The current version of the PHDL does not contain all of the features described as advantages of HDL's for PCB design in Section III. In particular, future work to be completed includes:

- The creation of an Eclipse plug-in to take advantage of all the IDE capabilities described above (cross probing, content assist, syntax highlighting, real-time syntax checking).
- The creation of DRC, ERC, and linting tools.
- Adding support for additional physical layout tool netlist formats.
- Adding support for back-annotating design changes made by the layout engineer back into the PHDL source code.
- The creation of additional analysis and automation tools for PHDL designs.

References

[1] Hardware Description Language. [Online]. Available: http://en.wikipedia.org/wiki/Hardware_description_language

[2] T. Parr, *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[3] ——. What is ANTLR? [Online]. Available: http://www.antlr.org/

[4] Xilinx. Verifying Pinout. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ise_c_report_pinout.htm