Hardware Description Language-Embedded Regular Expression Support for Module Iteration and Interconnection

Lionel Bening, Bryan Hornung
Hewlett-Packard Company
3000 Waterview Parkway
Richardson, TX 75080

Robert Pflederer
inSilicon Corporation
411 East Plumeria Drive
San Jose, CA 95134

Abstract

This paper describes a module iteration and interconnection support method based on regular expressions within a hardware description language elements. The method takes advantage of consistent naming practices within interconnected modules to produce orderly naming of the interconnections. It supports coherent and incremental interconnection net naming by use of naming rules. The acceptance of this method exceeded expectations in our design project work, suggesting that it may even be a future HDL replacement for the VHDL generate and Verilog for-loop

1 Introduction

Given the large numbers of parts and interconnections in computing hardware systems, designers have used automated techniques to generate the names for part instances and their interconnections.

These generating techniques originated decades before the standardization of hardware description languages. Being small and often ad-hoc programs, they are not widely discussed in the literature.

In this paper, we begin by comparing various generator in terms of the following attributes:

- explicit - implicit
- internal – external
- product – ad-hoc

We next present a user viewpoint of our preprocessor in section 3. Section 4 describes internal details of our preprocessor implementation. Section 5 suggests possible future steps we envision for wider use of this technology.

2 Comparing generating methods

2.1 Explicit vs implicit

From its beginnings, VHDL [1] supported a generate statement for iterated submodule instances and interconnections.

As Verilog became an open language and moved to be standardized [2] there were ideas [3] about iterating instances and connections, but they did not make it into the 1995 standard.

As of this writing, Section 12 of the Verilog IEEE 1364 2000(draft 5) proposed standard [4] describes new **generate**/**endgenerate**/**genvar** keywords that support iteration and connection generation.

We regard the generating techniques in both the VHDL and Verilog as *explicit*. They first create the instances and their interconnections at a module level, and then check the interconnection module ports in the port-to-net name specifications against the submodule ports being interconnected.

Implicit interconnection builds upon the generally recognized hardware design naming practice of using the same net and port names in a receiving module as in a transmitting module [5] [6]. For example, if the clock net name in the design for a clock-generating module is CK, use of CK as the port name for the corresponding nets on modules that receive that clock net makes the net function clear to anyone reading the design description.

This has prompted toolsmiths within many design shops and vendors [7][8][9][10] to develop preprocessors that automatically generate interconnect HDL modules implied by consistent name usage across submodules.

2.2 Internal vs external

Iteration/interconnection preprocessing can be internal to or external to the standard HDL's (Verilog or VHDL) language definition. External preprocessors are often written in C, or the *perl* language [11], and may invoke *perl*'s support of regular expression (regular-expression) matching. with rules supplied from another

file. Search tools and text editors have used regular-expression's for decades. Regular-expression matching tools and search library functions [12] are available free through the Free Software Foundation, Inc.

2.3 Product vs ad-hoc

Support and stability of the in-house implicit interconnection preprocessors varies from nil to tenuous to solid. With scripting languages like *perl*, rapid prototyping often results in module-specific preprocessor extensions and proliferation of preprocessor variants.

A clever way of dealing with module-specific extensions to the *perl* script is to place the script extension text within the module file as comments. Then the standard part of the *perl* script outside the module can locate the *perl* script extensions within the module comments and execute the *perl* code there.

A measure of support and stability of a preprocessor is whether designers check in the preprocessor input or preprocessor output into the release tree. With a supported and stable preprocessor, designers can check the preprocessor source into the release tree.

2.4 Our implementation

Our preprocessor implementation supports module iteration and interconnection based on implicit interconnection within the hardware description language. Compared with the current HDL for-loop and generate-loop techniques, our implementation has the advantages of:

- Improved designer productivity, built upon submodule port names that the designer has already entered.
- Rule-based generated interconnection that supports consistency across a project.
- Ease of maintenance/enhancement. When designers add ports, the port connections follow rules already entered.

Although not a product outside our laboratory, our generator implementation is of sufficient quality that we can check the Verilog preprocessor source into our design release tree.

3 Regular expression usage

3.1 Connection rules

With regular-expression extensions to Verilog, designers can enter pattern matching rules to form instance names and interconnection net names. The net names combine portions of the instance names and submodule port names. These pattern matching rules select portions of the instance names and the submodule port names for the preprocessor to combine and form connecting net names. Table 1 lists the applicability of some regular-expression match rules to instances and ports.

RE	MATCH RULE	APPLICABILITY	
[a-d0-4]	matches any character of set	port	instance
string1 | string2 | string3	matches string1 or string2 or string3	port	instance
.	matches any [a-zA-Z0-9_] character	port	
.*	matches 0 or more [a-zA-Z0-9_] characters	port	
$	matches the end-of-string	port	
(RE)	remembers the match for later reference	port	instance

Table 1. Regular Expression (RE) Application

3.2 Generating and using instance names

For instance names containing regular-expression patterns, the regular-expression processor uses the regular-expression to generate all instance names within a specified range. For example,

```
mtype minst_([a-b])([0-3]) ( ...
```

Example 1. Instances Specified in regular-expressions.

generates eight instances of the form:

```
mtype minst_a0 ( ...
mtype minst_a1 ( ...
mtype minst_a2 ( ...
mtype minst_a3 ( ...
mtype minst_b0 ( ...
mtype minst_b1 ( ...
mtype minst_b2 ( ...
mtype minst_b3 ( ...
```

Example 2. Instances Implied by regular-expressions.

Designers can reference the matching strings in net names by variables of the form $n, where n is an integer corresponding to the order in which the regular-expressions appear in the Verilog text. The numbering restarts at $1 with each new instance.

3.3 Matching and using port names

For port names in submodules that match patterns specified by designers, the regular-expression preprocessing can combine the module instance with portions of the submodule port names to produce connection net names. For each port within an instance, regular-

expression numbering restarts at one plus the last number of patterns in the instance name.

Example 3(a) shows two instances of a part **qdata** in which the regular-expression preprocessing generates net names based on the generated instance names and patterns that it finds in the submodule port.

```
qdata        qd([01]) (
  .rd0_ecc                  ( rd0_$1_ecc ),
  .(bt|rd|sc)([01])_(.*)$   ( $2$3_$1_$4 ),
  .hg_qd_(.*)$              ( hg_qd$1_$2 ),
  .hg_ecc_ind                    ( 6'b0 ),
  .qdctl(.*)$                ( qdctl$1$2 ),
  .op_s2c_(.*)$            ( op$1_s2c_$2 ),
  .dp_(.*)$                  ( dp$1_$2 ));
     ...
```

(a) Interconnect module with instance, ports and nets as regular expressions

```
module qdata( ... );
    input bt0_fnd_ff;
    input bt1_fnd_ff;
    input rd0_ecc;
    input [287:0] rd0dat2qd;
    input [5:0] rd0_wr_ind;
    input rd0_wr_stt;
    input rd1_ecc;
    input [287:0] rd1dat2qd;
    input [5:0] rd1_wr_ind;
    input rd1_wr_stt;
    input [2:0] hg_qd_adl;
    input [5:0] hg_qd_ind;
    input hg_qd_ind_vld;
    input hg_qd_len;
    input [5:0] hg_ecc_ind;
    input [6:4] op_s2c_qd_adl;
    input [5:0] op_s2c_qd_ind;
    input [3:0] op_s2c_qd_pt;
    input op_s2c_tv;
    input [5:0] sc0_wr_ind;
    input sc0_wr_trns;
    input [5:0] sc1_wr_ind;
    input sc1_wr_trns;
    input qdctl_ecc;
    output [287:0] dp_cdpdat;
    output [287:0] dpdat2mdp;
     ...
endmodule
```

(b) Submodule ports to be interconnected

Example 3. Interconnect module and submodule.

Except for the regular-expression text added in, this is standard Verilog.

- **qdata** is the submodule type name
- **qd([01])** is the instance name
- **rd0_ecc** is a port name on **qdata**
- **rd0_$1_ecc** is a net name connected to **rd0_ecc**
- the **.rd0_e** connection is an exception to the connection rul on the next line.

The key ideas in this paper shown in this example that support iteration and interconnection are:

- the regular-expressions within parenthesis in instance and port names,
- the **$** denoting the end of the port name, and
- the **$n** in the net name.

The **([01])** added in the instance name indicates that the designer wants two instances of **qdata**: **qd0** and **qd1**. Enclosing the range of values within parenthesis means that the range element (**0** or **1**, depending on the instance) can be referenced from scalar **$1** to form part of a net name, as shown in the net **rd0_$1_ecc**.

The port regular-expression

```
.( bt|rd|sc)([01])_(.*)$
```

provides the rule that matches any port on submodule type **qdata** that matches the 8 combinations of the first two regular-expressions and suffixed with any character.

Given the preceding input of the regular-expression Verilog shown in Example 3, the regular-expression preprocessing generates the Verilog instances and connections shown in Example 4.

```
qdata qd0 (
      .rd0_ecc             ( rd0_0_ecc ),
      .bt0_fnd_ff        ( bt0_0_fnd_ff ),
      .bt1_fnd_ff        ( bt1_0_fnd_ff ),
      .rd0_wr_ind        ( rd0_0_wr_ind ),
      .rd0_wr_stt        ( rd0_0_wr_stt ),
      .rd1_ecc             ( rd1_0_ecc ),
      .rd1_wr_ind        ( rd1_0_wr_ind ),
      .rd1_wr_stt        ( rd1_0_wr_stt ),
      .sc0_wr_ind        ( sc0_0_wr_ind ),
      .sc0_wr_trns      ( sc0_0_wr_trns ),
      .sc1_wr_ind        ( sc1_0_wr_ind ),
      .sc1_wr_trns      ( sc1_0_wr_trns ),
      .hg_qd_adl           ( hg_qd0_adl ),
      .hg_qd_ind           ( hg_qd0_ind ),
      .hg_qd_ind_vld    ( hg_qd0_ind_vld ),
      .hg_qd_len           ( hg_qd0_len ),
      .hg_ecc_ind              ( 6'h00 ),
      .qdctl_ecc           ( qdctl0_ecc ),
      .op_s2c_qd_adl    ( op0_s2c_qd_adl ),
      .op_s2c_qd_ind    ( op0_s2c_qd_ind ),
      .op_s2c_qd_pt      ( op0_s2c_qd_pt ),
      .op_s2c_tv           ( op0_s2c_tv ),
      .dp_cdpdat           ( dp0_cdpdat ),
      .dpdat2mdp            ( dpdat2mdp ),
      .rd0dat2qd            ( rd0dat2qd ),
      .rd1dat2qd           ( rd1dat2qd ));
qdata qd1 (
      .rd0_ecc             ( rd0_1_ecc ),
      .bt0_fnd_ff        ( bt0_1_fnd_ff ),
      .bt1_fnd_ff        ( bt1_1_fnd_ff ),
      .rd0_wr_ind        ( rd0_1_wr_ind ),
      .rd0_wr_stt        ( rd0_1_wr_stt ),
      .rd1_ecc             ( rd1_1_ecc ),
      .rd1_wr_ind        ( rd1_1_wr_ind ),
      .rd1_wr_stt        ( rd1_1_wr_stt ),
      .sc0_wr_ind        ( sc0_1_wr_ind ),
      .sc0_wr_trns      ( sc0_1_wr_trns ),
      .sc1_wr_ind        ( sc1_1_wr_ind ),
      .sc1_wr_trns      ( sc1_1_wr_trns ),
      .hg_qd_adl           ( hg_qd1_adl ),
      .hg_qd_ind           ( hg_qd1_ind ),
      .hg_qd_ind_vld    ( hg_qd1_ind_vld ),
      .hg_qd_len           ( hg_qd1_len ),
      .hg_ecc_ind              ( 6'h00 ),
      .qdctl_ecc           ( qdctl1_ecc ),
      .op_s2c_qd_adl    ( op1_s2c_qd_adl ),
      .op_s2c_qd_ind    ( op1_s2c_qd_ind ),
      .op_s2c_qd_pt      ( op1_s2c_qd_pt ),
      .op_s2c_tv           ( op1_s2c_tv ),
      .dp_cdpdat           ( dp1_cdpdat ),
      .dpdat2mdp            ( dpdat2mdp ),
      .rd0dat2qd            ( rd0dat2qd ),
      .rd1dat2qd           ( rd1dat2qd ));
```

Example 4. Preprocessor-generated interconnect

3.4 Bit Slice Arithmetic

In some designs, successive instances of a given type operate on different bit slices of a bus. Our preprocessor supports integer addition + subtraction - and multiplication * operations on numeric $n variables.

Example 5 is based on example 8 in Section 12 of the IEEE 1364 2000(draft 5) proposed Verilog standard [4]. As well as regular-expression-based instances and net names, this example presents regular-expression-based addition and multiplication applied to the instance number to specify successive bit slices.

```
sms_16b216t0 p([0-3]) (
   .dqi    (data[15+16*$1:16*$1]),
   .csb                    (csx),
   .ba                   (ba[0]),
   .addr               (adr[10:0]),
   .rasb                   (rasx),
   .casb                   (casx),
   .web                     (wex),
   .udqm             (dqm[2*$1+1]),
   .ldqm               (dqm[2*$1]),
   .dev_id           (dev_id3[4:0])
   );
```

Example 5. Instances, Interconnections and Bit Slices Specified in regular-expressions.

Example 6 is the resultant Verilog from the preceding preprocessor input shown in Example 5. The highlight/underlining relates the components of the original text with the generated instance and bit numbering.

```
sms_16b216t0 p0 (
   .dqi              (data[15:0]),
   .clk                    (clk),
   .csb                    (csx),
   .cke                    (cke),
   .ba                   (ba[0]),
   .addr               (adr[10:0]),
   .rasb                   (rasx),
   .casb                   (casx),
   .web                     (wex),
   .udqm                 (dqm[1]),
   .ldqm                 (dqm[0]),
   .dev_id           (dev_id3[4:0])
   );
sms_16b216t0 p1 (
   .dqi              (data[31:16]),
   .clk                    (clk),
   .csb                    (csx),
   .cke                    (cke),
   .ba                   (ba[0]),
   .addr               (adr[10:0]),
   .rasb                   (rasx),
   .casb                   (casx),
   .web                     (wex),
   .udqm                 (dqm[3]),
   .ldqm                 (dqm[2]),
   .dev_id           (dev_id3[4:0])
   );
```

```
sms_16b216t0 p2 (
    .dqi            (data[47:32]),
    .clk              (clk),
    .csb              (csx),
    .cke              (cke),
    .ba             (ba[0]),
    .addr         (adr[10:0]),
    .rasb             (rasx),
    .casb             (casx),
    .web              (wex),
    .udqm           (dqm[5]),
    .ldqm           (dqm[4]),
    .dev_id       (dev_id3[4:0])
    );
sms_16b216t0 p3 (
    .dqi            (data[63:48]),
    .clk              (clk),
    .csb              (csx),
    .cke              (cke),
    .ba             (ba[0]),
    .addr         (adr[10:0]),
    .rasb             (rasx),
    .casb             (casx),
    .web              (wex),
    .udqm           (dqm[7]),
    .ldqm           (dqm[6]),
    .dev_id       (dev_id3[4:0])
    );
```

Example 6. Resultant Instances, Interconnections and Bit Slices Derived from Regular-Rxpressions.

3.5 Instance-specific constants

To "personalize" multiple instances of a module, inputs can be tied off to an instance-specific constant value. For the example:

```
            …
kpi_pi4       pi([01]) (
 .pi_bus_num ( 1'b$1 ),
            …
```

Example 7. Instance-specific constant In regular-expression.

the preprocessor-generated Verilog is:

```
            …
kpi_pi4       pi0 (
 .pi_bus_num ( 1'b0 ),
            …
kpi_pi4       pi1 (
 .pi_bus_num( 1'b1 ),
            …
```

Example 8. Resultant Instance-Specific Constant

3.6 Defining shared interconnection rules

Example 9 illustrates macro definitions for regular interconnection expression rules and their shared usage across multiple module instances.

```
`define FOUR_INST mbt|mdprd|mdpwr|mcs|mdel
`define TWO_INSTnoe|mopim|mopc|mrqc|mdrd|mdrc
`define RULE_D(`FOUR_INST)_(.*)$ ($3$1$2_$4)
`define RULE_E(`TWO_INST)_(.*)$ ($3$1_$4)
      …
kmsc_mem_sched4 msc([01])([01]) (
              `RULE_D,
              `RULE_E);
kmpd_mpd_inout4 mdp([01])([01]) (
              `RULE_D,
              `RULE_F);
```

Example 9. Define rule-based interconnection.

4 Implementation

We implemented the regular-expression-based iteration and interconnection by combining the Free Software Foundation regular-expression pattern matching and search library [12] with our in-house Verilog parser and elaborator. Our parser supports the RTL subset described in [6] for our in-house Verilog-based applications.

We extended our RTL Verilog parser to recognize

- regular-expressions in instances and ports,
- scalars in connection net names and bit expressions.

We added a "regular-expression" phase in the elaboration that traverses the data structures representing the Verilog, transforming the data structures containing regular expressions and scalars into the expanded Verilog data structures, without the expressions and scalars. This phase runs in the following steps:

1. Iterate instances based on regular-expressions within instance names.
2. Form scalar lists based upon iterated instance names.
3. Search submodule ports for names corresponding to regular-expressions within the ports Use the *re_compile_pattern* and *re_match* functions from the regex.c [12] library.
4. Form scalars based upon regular-expression matches within port names.
5. Substitute scalar values within net names and bit positions.

5 Status and future directions

Acceptance of the HDL regular-expression technology among our designers has been good. Before we had this technology, about 10% of all our structural Verilog modules used automated interconnection based on implicit connections of matched port names.

Since we combined the implicit interconnection with the HDL regular-expression technology, 70% of our modules use automated interconnection.

5.1 Language considerations

We can only speak from our own regular-expression-based iteration and interconnect experience with Verilog. Someone else will have to look at the applicability of this technology in VHDL.

In Verilog, the use of regular-expression dollarsign-prefixed-scalars in net names conflicts with Verilog's allowing dollarsign characters in names. From the very beginning of our use of Verilog, we ruled the $ character out of our Verilog style[6]. This enabled us to use the $ to prefix scalar numbers in Verilog, just as in *perl*.

Even though we were able to parse and elaborate an RTL Verilog subset containing regular-expression iteration and interconnect, it remains to be determined whether parsing and elaboration problems will be encountered when this technology faces with the entire Verilog language.

5.2 Future of HDL Regular-Expressions

As of this writing, we are using this technology on one project in one laboratory in our company. Although sharing the method is fine, sharing the software itself beyond our laboratory is not a part to our main activity: designing logic chips and systems. We can consider sharing this software with other projects and labs within our company. However, delivery and support of EDA software outside our company is highly unlikely.

Because the HDL-embedded regular-expression's for iteration and interconnection technology seemed simple, powerful and unique, we filed for a patent on it. The patent and this paper serve to publicize and support the wider acceptance and use of this technology.

The vision that we have is to license this technology:

- Free, if it can be supported through a standardization group, or the Free Software Foundation.
- At low cost to commercial vendors, or in exchange for a cost discount to us on vendor products in exchange the vendor's use of this technology.
- At low cost to competitors who want to implement this technology in their in-house tools.

6 Conclusion

We have presented a HDL-embedded iteration and interconnection method that utilizes well-known regular-expression technology to build consistent rule-based net names. These net names build upon the iterated instance name extensions and the submodule port names, resulting in a high degree of consistency. As a result, designers from one project to the next increased their automatic interconnect generator usage on their structural modules 7X. We patented and publicized this technology with the vision that it might find widespread acceptance and support.

7 References

[1] IEEE Standard 1076-1987, *IEEE VHDL Language Reference Manual* IEEE, Inc. New York, NY, USA, 1988.

[2] IEEE Standard 1364-1995 *IEEE Standard Hardware Description Language Based on the Verilog Hardware description Language*, IEEE, Inc. New York, NY, USA, 1996.

[3] B. Davis and T. Mudge, "A Verilog Preprocessor for Representing Datapath Components," *Proceedings of the 4th International Verilog Conference* March, 1995, pp. 90-98.

[4] IEEE Proposed Standard 1364-2000 (Draft 5) *IEEE Standard Hardware Description Language Based on the Verilog Hardware description Language*, IEEE, Inc. New York, NY, USA, March, 2000.

[5] M. Keating and P. Bricaud, *Reuse Methodology Manual*, Kluwer Academic Publishers, 1999.

[6] L. Bening and H. Foster, *Verifiable RTL Design*, Kluwer Academic Publishers, Feb, 2000.

[7] I. Chayut, *MKTREE - Auto-Connects Verilog Version 4.0* http://www.verilog.net/mktree/

[8] B. J. Rosen *HDLmaker* http://www.polybus.com/hdlmaker

[9] M. McNamara *Mac's Verilog Mode for EMACs* http://www.verilog.com/verilog-mode.html

[10] *Vstruct* http://www.prime-tec.com

[11] L. Wall, T. Christiansen, J. Orwant, *Programming perl, 3rd Edition* O'Reilly & Associates, July, 2000, pp. 75-83.

[12] regex.c extended regular expression matching and search library, version 0.12, available from the Free Software Foundation, Inc http://www.gnu.org/