Problem F: Verilog Simulation Optimization via Instruction Reduction

Topic Chairs: Kyle Ko and Terry Liu

Synopsys, Inc.

1. Introduction

In IC design flow, validating functional correctness of a circuit design is an indispensable and crucial step. However, since the size and complexity of an SOC keep growing exponentially to satisfy various applications, the verification time is also significantly lengthened. In the past few decades, functional verification gradually becomes the most time-consuming part in the whole design cycle [1]. How to perform functional verification in a stringent time-to-market window is undoubtedly a critical problem in EDA field.

Simulator is an essential tool for functional verification and widely used in the whole design flow from RTL to gate-level. Its performance thus directly related to the IC development progress. Unfortunately, when the design size exceeds a certain amount, e.g., millions of gates, which is not uncommon in current IC design, the simulation time becomes unacceptable. For example, IC designers may spend couple of hours for design with millions of gates in RTL simulation. The number may even reach to several weeks in gate-level simulation [2]. In other words, the simulator performance enhancement is beneficial for the whole industry.

The objective of this contest topic, Verilog Simulation Optimization via Instruction Reduction, is to enhance the performance of simulator from the algorithm point of view. The contestants are asked to write a program performing Verilog-to-Verilog transformation to minimize instructions/assignments for performance consideration. The generated Verilog file must be syntactically correct, inviolate the given limitations, and functional equivalent to the input file.

2. Background

Hardware simulation with CPU is basically a compiler topic. In digital IC design flow, hardware designs are described in a hardware description language (HDL), such as Verilog. HDL simulators accept HDL designs as their input, and then generate binary codes for a specific platform (usually Intel 80x86). The IC designers execute those binary codes on a compatible high-performance workstation to verify the design functionality. Such compilation-based simulation happens repeatedly during IC design flow [3]. Therefore, minimizing the simulation time in every cycle can significantly shorten the development time. Most industry-level simulators (e.g., Synopsys VCS) consider the CPU time as the major optimization objective and leverage various compiler technologies to shorten the runtime [4].

Like traditional C/C++ compilers, an HDL simulator consists of several stages and can be roughly divided as three parts: parsing and design resolution (front-end), IR optimization (middle-end), and code generation (back-end). The front-end part analyzes the given source codes and builds the data structure representing the whole circuit (named intermediate representations, IR). It usually does not perform any optimization but focuses on depict the design with IR faithfully. The second part then optimizes the IR generated by the previous part with various algorithms. Since algorithms are applied with global perspective, this part may dominate the effectiveness of overall optimization. Additionally, the IR reflects the circuit design, optimizing it with algorithms considering circuit property is always very effective. Finally, the last part generates the compatible machine codes for the specific platform from the optimized IR. In contrast with the previous stage, local optimizations and platform-related optimizations are usually carried out.

In this topic, we focus on the IR optimization part and consider the instruction count as the optimization objective. Minimizing instructions is beneficial to reduce runtime due to less code being executed and data locality. In HDL code, the number of certain constructs, such as assignment and select, is proportional to the final instruction count. Thus, we define the optimization metric based on the two constructs and ask the contestants to minimize the metric.

3. Objective

This topic aims at facilitating Verilog simulation while keeping the correct outcome. The simulation efficiency is fundamentally equivalent to how much CPU time spent on instruction execution. For simplicity, we only focus on the CPU instruction count, which can be further interpreted as how many and what kinds of Verilog constructs are generated.

In this contest, contestants need to convert a given Verilog design into an optimized Verilog design (so called “source-to-source” transformation). With firm principle of simulation correctness, the optimized design is expected to have better simulation performance than the original one. Furthermore, only continuous assignments, wire type signals, bit operations, and bit/part-selects of vectors are considered in the given Verilog design for optimization.

The objective is to minimize the total number of continuous assignments and vector bit/part-selects in the Verilog design. We look forward to any innovative optimization strategies suitable for Verilog simulation.

4. Problem Formulation and Input/Output Format

The input of the problem is a Verilog file. Contestants need to develop a program to optimize the Verilog file. Evaluation metrics include the correctness and the number of reduced continuous assignments and vector bit/part-selects in the Verilog file. Additionally, there are some restrictions which cannot be violated during the optimization. The correctness of the optimized Verilog file will be verified via simulation, and testcases containing 2-state input values will be provided.

4.1. Program Requirement

The requested program `verilogopt` takes in a Verilog design “`original.v`” and outputs the optimized Verilog design “`optimized.v`”.

```
verilogopt original.v optimized.v
```

Contestants are required to implement a simple Verilog parser. Note that any 3rd party open-source Verilog parsers or simulators, e.g., `abc`, `iverilog`, `verilater`, are NOT allowed for this problem. However, the Verilog file `original.v` uses very restrictive Verilog syntax in a hope to relieve the contestants' burden on the parsing work. Basically, C library function `strcmp` along with some other basic character string processing APIs should suffice for the parsing work.

4.2. Input – Original Verilog Design

The original design, in the Verilog file named "`original.v`", contains only 2 modules: `dut` and `tb`. The module `dut` is the main module to be optimized and the module `tb` is merely an auxiliary module for checking the simulation correctness. Thus, contestants should put all their efforts on the module `dut`.

There are only 2 ports, `out` and `in`, declared for module `dut` as follows:

```
module dut (out, in);
  output[SIZEOUT:0] out;
  input[SIZEIN:0] in;
```

Both of them are packed vector ports, where `SIZEOUT` and `SIZEIN` are positive integer constants. Please note that Verilog ports are by default 2-state wires, where a value can be `0 or 1`.

In fact, an SOC module with only two wide ports is not quite common. It is entirely for the sake of this contest problem formulation. Industrial real designs will be appropriately adapted to conform to the problem formulation, e.g., consolidating all output ports into a single wide one.

In addition to the ports, there are also a bunch of 1-bit wire signals declared locally in the module `dut`. These local wires serve as temporary variables, which are common in real designs, as follows:

```
wire origtmp1;
wire origtmp2;
…
wire origtmpN;
```

The following expressions of ports and local wires are all continuous assignments.

```
assign <LHS> = <RHS>;
```

Here are the rules about `<LHS>` and `<RHS>`. To avoid confusion, **Backus-Naur Form (BNF)** is adopted here.

4.2.1.

`<LHS>` is either a bit-select on port `out` or a temporary wire: `out[#]` or `origtmp#` (where # denotes constant).

```
<LHS> ::=
        out[#]
        | origtmp#
```

4.2.2.

<RHS> is either one item, one item with unary operation, or a binary operation with two items.

```
<RHS> ::=
        <ITEM>
        | <UNARY_OP> <ITEM>
        | <ITEM> <BINARY_OP> <ITEM>
```

4.2.3.

<ITEM> can be either a 2-state constant, a bit-select on port `out`, a bit-select on port `in`, or a temporary wire.

```
<ITEM> ::=
         <1-bit 2-state constant>
         | out[#]
         | in[#]
         | origtmp#
```

4.2.4.

<UNARY_OP> is the bit-wise negation operator.

```
<UNARY_OP> ::=
              ~     (bit negation)
```

4.2.5.

<BINARY_OP> is one of the following binary bit-wise operators.

```
<BINARY_OP> ::=
               &     (bit and)
               | |   (bit or)
               | ^   (bit xor)
```

4.2.6.

Here is an example of module `dut`.

```
module dut (out, in);
  output[3:0] out;
  input[15:0] in;
  wire origtmp1;
  assign origtmp1 = 1'b0;
  assign out[2] = in[0] | origtmp1;
  assign out[3] = origtmp1 | 1'b0;
endmodule
```

As for the module `tb`, it is merely used to assist the validation of simulation correctness. Contestants only need to copy it to the optimized Verilog file without any modification.

```
module tb();
  reg[3:0] results[1];
  reg[15:0] data[1];
  dut duttest(results[0], data[0]);
  initial begin
    $readmemb("data.txt", data);
    $display("data = [%16b]", data[0]);
    #1
    $display("results = [%4b]", results[0]);
    $writememb("results.txt", results);
  end
endmodule
```

4.3. Output – Optimized Verilog Design

The optimized Verilog design, named "`optimized.v`", contains 2 modules as well: `dut` and `tb`. The module `dut` contains the optimized design achieved by contestants, and the module `tb` requires to be identical to that in `original.v`.

In order to explore potential optimization opportunities, the optimized design allows a little more flexible Verilog syntax. Here are the rules about the optimized design.

4.3.1.

In addition to the local temporary wires `origtmp#`, declaration of extra temporary wires is permitted, named `xformtmp#`. Each `xformtmp#` can be a single or multiple-bit wide. Multiple bits need to be one dimensional and packed only. Furthermore, multiple bits can be declared either ascending or descending indices. For example,

```
// 1-bit → OK
wire      xformtmp1;
// packed 4-bit, descending indices from 3 to 0 → OK
wire[3:0] xformtmp2;
// packed 7-bit, ascending indices from 0 to 6 → OK
wire[0:6] xformtmp3;
// unpacked 2-bit → NOT allowed
wire      xformtmp4[1:0];
// packed 2 or more dimensions → NOT allowed
wire[7:0][1:0] xformtmp5;
```

4.3.2.

`<LHS>` allows 2 more types: `xformtmp#` and `out[MSB:LSB]`. In Verilog syntax, `myvect[MSB:LSB]` denotes 'part select' of vector `myvect`, which means consecutive bits from index MSB to index LSB (width equals to `MSB-LSB+1`). Both MSB and LSB are non-negative integers. MSB can be either greater than (descending), less than (ascending), or equal to LSB. According to IEEE standard, though ascending or descending indices are both fine, every use must be consistent with its declaration.

```
// declared descending
output[3:0] out;
assign out[0:1] = …    // used ascending → Syntax Error!
// declared ascending
wire[0:15] xformtmp3;
assign xformtmp3[8:12] = …    // used ascending → ok
assign xformtmp3[4:1] = …     // used descending → Syntax Error!
```

```
<LHS> ::=
        out
        | out[#]
        | out[MSB:LSB]
        | origtmp#
        | xformtmp#
        | xformtmp#[#]
        | xformtmp#[MSB:LSB]
```

4.3.3.

<RHS> is the same as input Verilog.

4.3.4.

<ITEM> allows four extra types: `out[MSB:LSB]`, `in[MSB:LSB]`, `xformtmp#`, and `xformtmp#[MSB:LSB]`. Additionally, it could be a N-bit 2-state constant.

```
<ITEM> ::= <1 or N-bit 2-state constant>
         | out[#]
         | in[#]
         | origtmp#
         | out[MSB:LSB]
         | in[MSB:LSB]
         | xformtmp#
         | xformtmp#[MSB:LSB]
```

4.3.5.

<UNARY_OP> is the same as input Verilog.

4.3.6.

<BINARY_OP> is the same as input Verilog.

4.3.7.

Here is an example of optimized module `dut` in `optimized.v`.

```
module dut (out, in);
  output[3:0] out;
  input[15:0] in;
  wire origtmp1;
  wire[1:0] xformtmp1;
  assign origtmp1 = 1’b0;
  assign xformtmp1[0] = in[1] & 1’b0;
  assign xformtmp1[1] = origtmp1;
  // xformtmp1 below is same as xformtmp1[1:0]
  assign out[3:2] = xformtmp1;
endmodule
```

4.4. Verification of Simulation Correctness

There is another input file named ‘`data.txt`’, which is used for verifying simulation correctness of `optimized.v`. It is the input memory data read by system function `$readmemb` in module `tb`. The content of `data.txt` contains one-line text string of characters: `0`, `1`. For example (suppose that the data to read in is for ‘`reg[15:0] data[1]`’),

```
0111100100011001
```

Simulation results, i.e., value of `out` port vector, are dumped via `$writememb`. Simulation correctness is checked by comparing the `$writememb` results of `original.v` and `optimized.v`.

4.5. Examples

Suppose we have the following `original.v`.

```
module dut (out, in);
  output[3:0] out;
  input[15:0] in;
  wire origtmp1;
  assign origtmp1 = 1’b0;
  assign out[2] = in[0] | origtmp1;
  assign out[3] = in[1] | 1’b0;
endmodule
```

Then we may derive the following 3 versions of valid `optimized.v`.

4.5.1. Version 1

```
module dut (out, in);
  output[3:0] out;
  input[15:0] in;
  assign out[2] = in[0] | 1’b0;
  assign out[3] = in[1] | 1’b0;
endmodule
```

4.5.2. Version 2

```
module dut (out, in);
  output[3:0] out;
  input[15:0] in;
  wire[1:0] xformtmp1;
  assign xformtmp1 = 2’b0;
  assign out[3:2] = in[1:0] | xformtmp1;
endmodule
```

4.5.3. Version 3

```
module dut (out, in);
  output[3:0] out;
  input[15:0] in;
  assign out[3:2] = in[1:0];
endmodule
```

5.

6.

7. Evaluation

There are three evaluation metrics: simulation correctness, optimization regulation, and efficiency. Simulation correctness and optimization regulation are priority requirements. If any of them is violated, efficiency will NOT be evaluated any further.

7.1. Simulation Correctness

Verilog simulation results are generated in module `tb` by dumping to memory files through `$writememb("results.txt", results)`. The dumped results from `optimized.v` and `original.v` must be identical. For contestants’ ease of debugging, associated `$display()` calls are also added.

Simulation of `original.v` and `optimized.v` will be conducted by the industrial Verilog simulator Synopsys VCS (denoted `<sim>`). The result is PASS if no text diff is observed; otherwise, it is FAIL.

```
<sim> original.v
// “results.txt” will be renamed as “simorig.txt” by Topic Chairs

<sim> optimized.v

diff simorig.txt results.txt
```

```
CSCORE = 1 (if no diffs)
CSCORE = 0 (if any diffs found)
```

7.2. Optimization Regulation

The output `optimized.v` has to follow the rules regulated in Section 4.3.1 through 4.3.7.

```
RSCORE = 1 (if all regulations followed)
RSCORE = 0 (if any regulation violated)
```

7.3. Execution Time Limit

The program `verilogopt` developed by each contestant needs to execute for a reasonable time. For this, there is an execution time limit 600 seconds that cannot be exceeded for each testcase.

```
TSCORE = 1 (if ‘verilogopt’ executes 600sec or less)
TSCORE = 0 (if ‘verilogopt’ executes longer than 600sec)
```

7.4. Simulation Efficiency

Simulation efficiency is measured with the total number of two types of assignments in the module `dut` in `optimized.v` as follows:

- continuous assignments (denoted as <COUNT_ASGN>), and
- bit/part-selects appearing in all continuous assignments (denoted as <COUNT_SELS>)

```
ESCORE =
(<OUT_PORT_WIDTH> + <IN_PORT_WIDTH>) / (<COUNT_ASGN> + <COUNT_SELS>)
```

For each `original.v`, the number `(<OUT_PORT_WIDTH> + <IN_PORT_WIDTH>)` is fixed. The lower number of `(<COUNT_ASGN> + <COUNT_SELS>)` indicates fewer CPU instructions needed, and therefore leads to the higher `EFFICIENCY_SCORE`.

7.5. Scoring for Single Test Case

For each `optimized.v`, a final score will be concluded from simulation correctness, optimization regulation, and efficiency.

```
FSCORE = CSCORE * RSCORE * TSCORE * ESCORE
```

7.6. Examples

Take the 3 versions of `optimized.v` from Section 4.5 for example, here is the respective score calculation.

7.6.1. Version 1

<COUNT_ASGN> = 2 *(demonstrated below)*

```
    assign out[2] = in[0] | 1'b0;
    assign out[3] = in[1] | 1'b0;
```

<COUNT_SELS> = 4 *(demonstrated below)*

```
    assign out[2] = in[0] | 1'b0;
    assign out[3] = in[1] | 1'b0;

    (4 bit/part selects: out[2], in[0], out[3], in[1])
```

<COUNT_ASGN> + <COUNT_SELS> = 2 + 4 = 6

7.6.2. Version 2

<COUNT_ASGN> = 2 *(demonstrated below)*

```
    assign xformtmp1 = 2'b0;
    assign out[3:2] = in[1:0] | xformtmp1;
```

<COUNT_SELS> = 2 *(demonstrated below)*

```
    assign xformtmp1 = 2'b0;
    assign out[3:2] = in[1:0] | xformtmp1;

    (2 bit/part selects: out[3:2], in[1:0])
```

<COUNT_ASGN> + <COUNT_SELS> = 2 + 2 = 4

7.6.3. Version 3

<COUNT_ASGN> = 1 *(demonstrated below)*

```
    assign out[3:2] = in[1:0];
```

<COUNT_SELS> = 2 *(demonstrated below)*

```
    assign out[3:2] = in[1:0];

    (2 bit/part selects: out[3:2], in[1:0])
```

`<COUNT_ASGN> + <COUNT_SELS> = 1 + 2 = 3`

7.6.4. *Summary of the Efficiency Examples*

Here is the summary of the 3 versions and respective FSCORE achieved. Basically, the sum of port widths is identical for all 3 optimization results:

`(<OUT_PORT_WIDTH> + <IN_PORT_WIDTH> = 4 + 16 = 20`

Version 3 of `optimized.v` leads to the smallest `(<COUNT_ASGN> + <COUNT_SELS>)` and highest FSCORE (6.67), which implies the best performance for demanding the least CPU instructions on simulating assignments and selects.

	VERSION 1	**VERSION 2**	**VERSION 3**
<COUNT_ASGN> + <COUNT_SLES>	6	4	3
FSCORE	20 / 6 = 3.33	20 / 4 = 5	20 / 3 = 6.67

7.7. Scoring for Single Contestant

Several test cases (including public and hidden ones) will be used to evaluate a contestant's program, `verilogopt`. The ranking of the contest is based on the summation of FSCORE acquired from each test case.

8. Test Case

Some test cases, including "`original.v`" and "`data.txt`", will be announced soon.

9. Reference

[1] A. Evans et al., "Functional verification of large ASICs," Proceedings 1998 Design and Automation Conference. 35th DAC. (Cat. No.98CH36175), San Francisco, CA, USA, 1998, pp. 650-655, doi: 10.1145/277044.277210.

[2] Y. Deng, "GPU Accelerated VLSI Design Verification," 2010 10th IEEE International Conference on Computer and Information Technology, Bradford, 2010, pp. 1213-1218, doi: 10.1109/CIT.2010.219.

[3] D. Page, "Hardware design using verilog", in A practical introduction to computer architecture, Springer Science & Business Media, 2009.

[4] VCS, Synopsys. https://www.synopsys.com/verification/simulation/vcs.html.