Problem A: Learning Arithmetic Operations from Gate-Level Circuit

Chung-Han Chou, Chih-Jen (Jacky) Hsu, Chi-An (Rocky) Wu, and Kuan-Hua Tu
Cadence Design Systems, Inc.

- 2022.02.20 First Version
- 2022.03.08 Update runtime limitation
- 2022.05.12 Update cost function

1. Introduction

The ability of extracting circuit functionality from a gate-level netlist is critical in CAD tools since it can help identify malicious circuits, speed-up Engineering Change Order (ECO) process, and benefit formal verification. However, as the technologies advance rapidly, the complexity of modern IC increases incredibly and makes it difficult to extract circuit functionality just from a given gate-level netlist.

Therefore, the goal of this contest is to develop a tool to perform datapath extraction on a given gate-level circuit consisting of only primitive gates. That is, contestants are requested to write a program to "learn" and extract arithmetic operations from the given netlist and output the datapath in the RTL format.

The following is an example that demonstrates the datapath extraction. In Figure 1, the left gate-level circuit contains 28 gates. If we "abstract" the arithmetic operations from the netlist to the right RTL, the functionality of the circuit becomes clear.

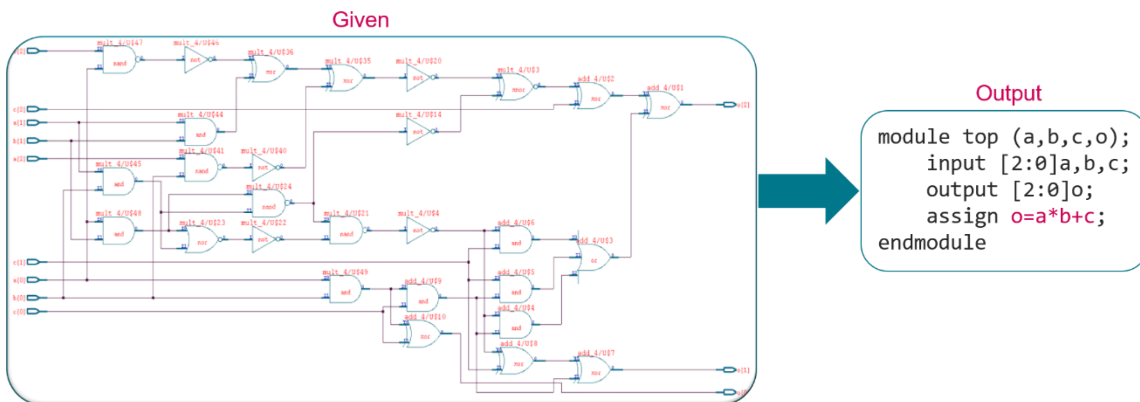

Figure 1. Example of datapath extraction.

2. Background

As the complexity of modern VLSI design increases, how to extract functionality from a given netlist has become a critical issue for many reasons. The first reason is security. Third-party resources such as fabrication services and soft/hard IP cores are widely used. Although the design effort and time to market (TTM) can be reduced by applying the third-party resources, the security issues such as hardware Trojans and IP piracy have become non-negligible threats. It is desirable for designers to have an effective technique that helps

inspect the netlist and detect hardware Trojans or malicious design changes [1][2][3][4][5][6][7][8].

Second, correctly identifying the functionality of a netlist can also benefit current design flow. For example, in formal verification, most advanced verification techniques need design information or human input. Knowing the functionality of the netlist can reduce the complexity and effort. Another example is to benefit the Engineering Change Order (ECO) process. A strong analyzer can keep the designer from locating the ECO gate in a sea of bit-level gates.

Although there are several academic research on the datapath extraction [1][2][3][4][5], learning datapath from a large circuit is still difficult due to several reasons. Different assumptions by researchers make the approaches largely *ad hoc* [9]. Aggressive optimization strategies adopted by synthesis tools makes the datapath learning even complicated.

As a result, in this contest, we formulate a datapath learning and extraction problem. We expect contestants can develop a tool to learn the arithmetic equations from a synthesized gate-level netlist.

3. Problem Formulation and Input Output Format

This section defines the input, output, and the requirement of this contest.

Given a flatten Verilog netlist which contains only primitive gates, contestants need to write a program to generate a functionally equivalent RTL module and minimize the number of operators in the output RTL by extracting the datapaths (arithmetic operations) from the netlist.

3.1 Program requirement

The requested program must be run on a Linux system. The time limit for each case is 8 hours. Parallel computation with multiple threads or processes is not allowed. The program should accept two arguments, “-input <netlist_path/name.v>” and “-output <output_path/name.v>”, for specifying the input and output files, respectively.

3.2 Input file format

- Input file is a flatten netlist in Verilog format without hierarchy (one top module only). The netlist is synthesized from RTL which is composed of addition, subtraction, multiplication, conditioning, and/or, bit selection and concatenation.
- The netlist is composed of:
    1. primitive gates (and, or, nand, nor, not, buf, xor, xnor)
    2. wires
    3. constant values (1’b1, 1’b0)
- Each primary input and primary output can be either a scalar or a vector:
    - input [7:0] in1; // an input vector
    - input in2; // an input scalar
    - output [7:0] out1; // an output vector
    - output out2; // an output scalar

3.3 Output file format and requirements

- Output file should be in Verilog format.
- DO NOT change the name of the top module.
- DO NOT change the name and declaration for primary inputs and primary outputs.
- Allowed Verilog operators are shown in Appendix A
- Allowed Verilog keywords are shown in Appendix B

4. Evaluation Criteria

- Correctness is necessary. If the generated RTL is not functionally equivalent to the given netlist, the contestants will get zero points for the case.
- ***cost*** for each case:
  1. ***module_cost*** for each module:
     - The contribution to the ***module_cost*** for each operator can be found in Appendix A.
     - The contribution to the ***module_cost*** for each keyword can be found in Appendix B.
     - Each submodule instantiation contributes the ***module_cost*** of the submodule.
     - Instantiation of a primitive gate contributes 1.
  2. ***cost*** = ***module_cost*** of the `top` module.
- For each case, ***reduction_rate*** MUST be larger than or equal to 70%; otherwise, 0 points are obtained for the case.

$$\boldsymbol{reduction_rate} = \left(1 - \frac{\boldsymbol{cost}}{Gate\ count\ in\ input\ netlist}\right) \times 100\%$$

- ***point*** for each case which has a ***reduction_rate*** >= 70%:

$$\boldsymbol{point} = \frac{\min(\boldsymbol{cost}\ of\ the\ case\ from\ all\ teams)}{\boldsymbol{cost}\ of\ the\ case} \times 100\%$$

- ***final_score*** is the sum of ***point*** of all cases.
- The team having the largest ***final_score*** wins.

5. Examples

This section demonstrates the calculation of ***cost*** and the evaluation metric.

Figure 2 (a)-(d) show four Verilog RTL examples, which have the same functionality. Let us consider their ***cost***. In (a), the top module contains 2 equality operators ("=="), 3 multiplication operators ("*"), 3 addition operators ("+"), and 2 conditional operators ("? :"). As a result, the ***cost*** of (a) is 2+3+3+2=10. In (b), the numbers of equality operators, addition operators, and multiplication operators are 2, 3, and 3, respectively. Furthermore, there are 2 "if" keywords, which totally cost 2. As a result, the ***cost*** of (b) is 2+3+3+2=10 as well. In (c), there are 2 case items, which totally cost 4. As a result, the ***cost*** is still 10. In (d), Multiply-Accumulate (MAC) are extracted as a submodule (named `my_mac`). According to the cost definition, the ***module_cost*** of `my_mac` is 2. Since there 3 instantiations of `my_mac` in the `top` module, the ***cost*** of (d) is 2x3+2+2=10.

```
module top(a, b, c, d, e, s, o);
  input  [3:0] a, b, c, d, e;
  input  [1:0] s;
  output [3:0] o;
  assign o=
    (s==2'b00)?a*b+e:
    (s==2'b01)?a*c+e:
               a*d+e;
endmodule
```

(a)

```
module top(a, b, c, d, s, o);
  input  [3:0] a, b, c, d;
  input  [1:0] s;
  output reg [3:0] o;
  always @ (*) begin
    if      (s==2'b00) o<=a*b+e;
    else if(s==2'b01) o<=a*c+e;
    else               o<=a*d+e;
  end
endmodule
```

(b)

```
module top(a, b, c, d, s, o);
  input  [3:0] a, b, c, d;
  input  [1:0] s;
  output reg [3:0] o;
  always @ (*) begin
    case(s)
      2'b00:   o<=a*b+e;
      2'b01:   o<=a*c+e;
      default: o<=a*d+e;
    endcase
  end
endmodule
```

(c)

```
module top(a, b, c, d, e, s, o);
  input  [3:0] a, b, c, d, e;
  input  [1:0] s;
  output [3:0] o;
  wire   [3:0] r1, r2, r3;
  my_mac M1(a, b, e, r1);
  my_mac M1(a, c, e, r2);
  my_mac M1(a, d, e, r3);
  assign o=
    (s==2'b00)?r1:
    (s==2'b01)?r2:
               r3;
endmodule
module my_mac(a, b, c, o);
  input [3:0] a, b, c;
  output[3:0] o;
  assign o=a*b+c;
endmodule
```

(d)

Figure 2. Demonstrations of ***cost*** calculation. (a), (b), (c) and (d) show four different, but functionally-equivalent Verilog RTL modules.

Finally, we demonstrate the calculation of ***reduction rate*** with the example shown in Figure 1 . The ***cost*** of the output RTL is 2 (one "+" and one "*"), and the gate count of the given gate-level netlist is 28. According to the definition, $\boldsymbol{reduction_rate} = (1 - 2/28) \times 100\% = 91.75\%$, which is much larger than 70%.

6. Appendix

6.1 Appendix A: Allowed Verilog operators and the corresponding cost

Operator	Description	Cost
{} {{}}	Concatenation, replication	1/per symbol
[] [:]	Bit-select, part-select	1
+ - * / **	Arithmetic	1
%	Modulus	1
> >= < <=	Relational	1
!	Logical negation	1
&&	Logical and	1
||	Logical or	1
==	Logical equality	1
!=	Logical inequality	1
===	Case equality	1
!==	Case inequality	1
~	Bit-wise negation	1/per bit
&	Bit-wise and	1/per bit
|	Bit-wise inclusive or	1/per bit
^	Bit-wise exclusive or	1/per bit
^~ or ~^	Bit-wise exclusive nor	1/per bit
&	Reduction and	1
~&	Reduction nand	1
|	Reduction or	1
~|	Reduction nor	1
^	Reduction xor	1
~^ or ^~	Reduction xnor	1
<<	Logical left shift	1
>>	Logical right shift	1
<<<	Arithmetic left shift	1
>>>	Arithmetic right shift	1
? :	Conditional	1

※ For example, the cost of the following assignment is 7. This assignment is composed by a 4-symbol concatenation and a 3 bit(part)-selections.

```
assign x[9:3] = {a[7:4], b[3], c, 1’b0} ;
```

6.2 Appendix B: Allowed Verilog keywords and the corresponding cost

keyword	cost	keyword	cost
always	0	input	0
and	1 (primitive)	integer	0
assign	0	module	0
begin	0	nand	1 (primitive)
buf	1 (primitive)	nor	1 (primitive)
case	2 per item (default is excluded)	not	1 (primitive)
casex		or	1 (primitive)
casez		output	0
default	0	parameter	0
else	0	reg	0
end	0	signed	0
endcase	0	unsigned	0
endmodule	0	wire	0
for	0	xnor	1 (primitive)
if	1	xor	1 (primitive)

7. References

[1] T. Meade, S. Zhang, and Y. Jin, "Netlist Reverse Engineering for High-Level Functionality Reconstruction," in *ASP-DAC*, 2016, pp. 655–660.

[2] P. Subramanyan, N. Tsiskaridze, K. Pasricha, D. Reisman, A. Susnea, and S. Malik, "Reverse engineering digital circuits using functional analysis," in the *Proc. of IEEE/ACM Design Automation and Test in Europe*, 2013.

[3] W. Li, Z. Wasson, and S. A. Seshia, "Reverse engineering circuits using behavioral pattern mining," in the *Proc. of IEEE International Symposium on Hardware-Oriented Security and Trust*, pp. 83–88, 2012.

[4] W. Li, A. Gascon, P. Subramanyan, W. Y. Tan, A. Tiwari, S. Malik, N. Shankar, and S. A. Seshia, "WordRev: Finding word-level structures in a sea of bit-level gates," in *Hardware-Oriented Security and Trust*, 2013, pp. 67–74.

[5] M. Fyrbiak, S. Wallat, P. Swierczynski, M. Hoffmann, S. Hoppach, M. Wilhelm, T. Weidlich, R. Tessier, and C. Paar, "HAL—The missing piece of the puzzle for hardware reverse engineering, Trojan detection and insertion," *IEEE Trans. Depend. Secure Comput.* 16, 3 (2018), 498–510.

[6] Z. Huang, Q. Wang, Y. Chen, and X. Jiang, "A survey on machine learning against hardware trojan attacks: Recent advances and challenges," *IEEE Access*, vol. 8, pp. 10796–10826, 2020.

[7] Y. Yang, J. Ye, Y. Cao, J. Zhang, X. Li, H. Li, and Y. Hu, "Survey: Hardware Trojan Detection for Netlist," in *IEEE 29th Asian Test Symposium*, pp. 1-6. IEEE, 2020.

[8] C. Yu, X. Zhang, D. Liu, M. Ciesielski, and D. Holcomb, "Incremental SAT-based reverse engineering of camouflaged logic circuits," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 36, no. 10, pp. 1647–1659, Oct. 2017.

[9] M. Rostami, F. Koushanfar, and R. Karri, "A primer on hardware security: Models, methods, and metrics," *Proc. IEEE*, vol. 102, no. 8, pp. 1283–1295, Aug. 2014.