

Diagnosis and Design of Reliable Digital Systems

Circuit Parser Tutorial

Ziming AO

Department of Electrical and Computer Engineering
University of Southern California

Fall-2022



1. Introduction
2. Input Format
3. Data Structure
4. Different function of the Parser
5. Levelization Procedure



Introduction



- Current version of readckt in C does not support xor xnor
- Current version of readckt in C does not support branch after branch while the circuit for mini-project (to be introduce later) has this feature.



Introduction

- 1 Understand the circuit format
- 2 Understand the parser
- 3 Modify the parser and add a "levelization" function

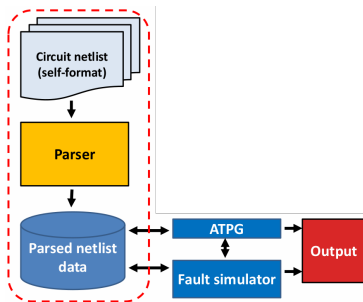


Figure 1: Outline of an ATPG system



Question: What is a parser? A parser breaks data into smaller elements, according to a set of rules that describe its structure. Most data can be decomposed to some degree. For example, a phone number consists of an area code, prefix and suffix; and a mailing address consists of a street address, city, state, country and zip code.



Introduction

- 1 This circuit format (*self format*) is based on outputs of a ISCAS 85 format translator written by Dr. Sandeep K. Gupta (USC). This parser has been developed by Chihang Chen in 1994.
- 2 The format uses only integers to represent circuit information.
- 3 The programming language is C.
- 4 You can modify it to add your function for levelization - 'lev'

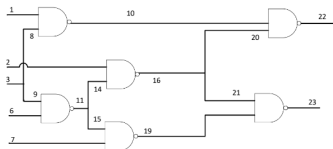


Input Format



Input Format

Example is C17.ckt:



(a) c17 gate level schematic

```
1 1 0 1 0
1 2 0 1 0
1 3 0 2 0
2 8 1 3
2 9 1 3
1 6 0 1 0
1 7 0 1 0
0 10 6 1 2 1 8
0 11 6 2 2 9 6
2 14 1 11
2 15 1 11
0 16 6 2 2 2 14
2 20 1 16
2 21 1 16
0 19 6 1 2 15 7
3 22 6 0 2 10 20
3 23 6 0 2 21 19
```

(b) CKT representation of c17



CKT Circuit Format

Important note: The CKT format is based on the nodes, and not the gates. In fact, every line is dedicated to one node, and its up-stream gate.

1	2	3	4	5	6
0 GATE	outline	0 IPT	# Fan-out	# Fan-in	inlines
		1 BRCH			
		2 XOR			
		3 OR			
		4 NOR			
		5 NOT			
		6 NAND			
		7 AND			
1 PI	outline	0	# Fan-out	0	
2 FB	outline	1 BRCH	inline		
3 PO	outline	[2-7]	0	# Fan-in	inlines

PI*: Primary Input

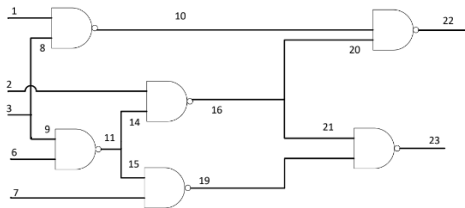
FB: Branch

PO: Primary Output



CKT format: column 2

The second column is dedicated to an integer number representing the node, just like an **ID**. Of course, all the numbers in this column are unique.

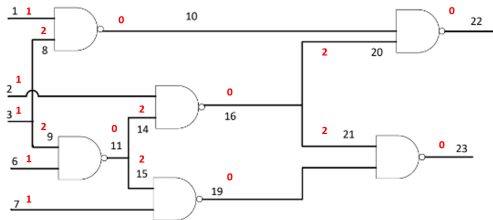


2
1
2
3
8
9
6
7
10
11
14
15
16
20
21
19
22
23



CKT format: column 1

The first column is dedicated to represent the **type of the node**. It can be an output of a gate, primary input, primary output, or fan-out of a branch.

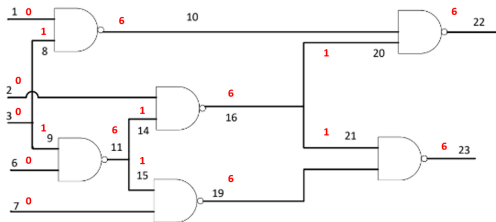


1	2
1	1
1	2
1	3
2	8
2	9
1	6
1	7
0	10
0	11
2	14
2	15
0	16
2	20
2	21
0	19
3	22
3	23



CKT format: column 3

The third column determines the **type of the up-stream gate**. As you can see, this column is redundant for PI, FB. Note that a node type can be a PO, and at the same time it represents an up-stream gate.

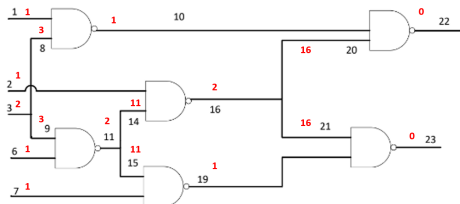


2	3
1	0
2	0
3	0
8	1
9	1
6	0
7	0
10	6
11	6
14	1
15	1
16	6
20	1
21	1
19	6
22	6
23	6



CKT format: column 4

This column represents different attributes based on the node type):
GATE and PI: number of fan-outs of this gate (usually 1, but not always!);
FB: the node-ID of the fan-in stem; PO: constant value 0.

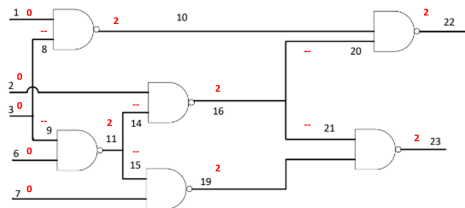


2	4
1	1
2	1
3	2
8	3
9	3
6	1
7	1
10	1
11	2
14	11
15	11
16	2
20	16
21	16
19	1
22	0
23	0



CKT format: column 5

For GATE nodes (or PO), this is the number of fan-in to the up-stream gate. For PI the constant value 0.

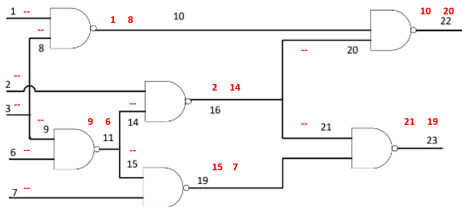


2	5
1	0
2	0
3	0
8	
9	
6	0
7	0
10	2
11	2
14	
15	
16	2
20	
21	
19	2
22	2
23	2



CKT format: column 6

For GATE nodes (or PO), this is the node ID of the fan-in to the upstream of the gate.

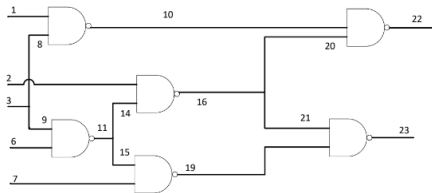


2	6 ...
1	
2	
3	
8	
9	
6	
7	
10	1 8
11	9 6
14	
15	
16	2 14
20	
21	
19	15 7
22	10 20
23	21 19



CKT format

Following is the circuit format, wherein the first column represents the node ID, followed by the six ckt formats mentioned above.



1	2	3	4	5	6...
1	1	0	1	0	
1	2	0	1	0	
1	3	0	2	0	
2	8	1	3		
2	9	1	3		
1	6	0	1	0	
1	7	0	1	0	
0	10	6	1	2	18
0	11	6	2	2	96
2	14	1	11		
2	15	1	11		
0	16	6	2	2	2 14
2	20	1	16		
2	21	1	16		
0	19	6	1	2	15 7
3	22	6	0	2	10 20
3	23	6	0	2	21 19



Data Structure



Part 3: Data Structure

- 1 Data structure is an aggregate data type that is composed of two or more related variables (members).
- 2 For a combinational circuit, it is useful to manage various information of a node (gate) in the same data structure.
- 3 Some data structures consist of: arrays, link lists, self defined structures etc.
- 4 A data structure can consist of other data structures.



Part 3: Self defined structure

- 1 We can define our own data structure
- 2 A structure is a collection of variables under a single name. These variables can be of different types, and each has a name which is used to select it from the structure. A structure is a convenient way of grouping several pieces of related information together.
- 3 A structure can be defined as a new named type, thus extending the number of available types. It can use other structures, arrays or pointers as some of its members, though this can get complicated unless you are careful.



Self defined structure in the Parser

In this Parser the self defined structure is:

```
typedef struct n_struct {  
    unsigned indx; //node index [0, NumOfLine-1]  
    unsigned sum; //line number maybe different from indx  
    enum e_gtype type; //gate type  
    unsigned fin; //number of fanins  
    unsigned fout; //number of fanouts  
    int level; //level of the gate output  
    struct n_struct **unodes; //ptr to array of down nodes  
    struct n_struct **dnodes; //ptr to array of up nodes  
};
```



Part 3: Array

- 1 An array is a collection of variables of the same type. Individual array elements are identified by an integer index. In C the index begins at **zero** and is always written inside square brackets.
- 2 Arrays can have one or more dimensions, in which case they might be declared as:
 - `int Node[100];`
 - `int Node_2d[20][5];`
 - `int Node_3d[20][5][3];`
- 3 Each index has its own set of square brackets.



Part 3: Array in the parser

This parser uses an array to store the information regarding the logic circuit to be processed.

- First we should define an array:

```
NSTRU *Node; // Another for of array declaration
```

- Then we should reserve memory:

```
int Nnodes=100; // Maximum Number of Nodes
```

```
Node = (INSTRUC*) malloc(Nnodes × sizeof(INSTRUC));
```

Accessing a member in a node (with an example):

```
NSTRU *np;
```

```
np = & Node[node_num];
```

```
np → indx = 6;
```

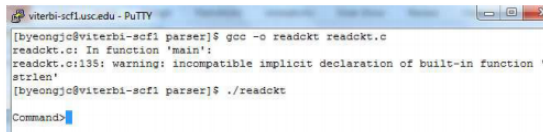


Different function of the Parser



Part 4:How to run the parser

- 1 Make sure that these two files are in your UNIX account using a FTP program, e.g., Filezilla
 - parser.c– source code for the parser
 - c17.ckt– example circuit file, written in our self-format
- 2 Compile the source code: `gcc -o readckt readckt.c`
- 3 Execute the parser with this command: `./readckt`
 - Then you see: Command>
- 4 Here you can enter the name of the function
 - For example: Command>Help



```
viterbi-scf1.usc.edu - PuTTY
[byeongjc@viterbi-scf1 parser]$ gcc -o readckt readckt.c
readckt.c: In function 'main':
readckt.c:135: warning: incompatible implicit declaration of built-in function '
strlen'
[byeongjc@viterbi-scf1 parser]$ ./readckt
Command>
```



Part 4: Different function of the parser

The given parser has 4 functions

- READ
- PC
- HELP
- QUIT



Part 4:READ

input file: circuit description file name eg. c17.ckt

output file: nothing

- 1 This function reads the circuit description file and set up all the required data structure.
- 2 It first checks if the file exists, then it sets up a mapping table, determines the number of nodes, PI's and PO's, allocates dynamic data arrays, and fills in the structural information of circuit.
- 3 To have maximal flexibility, three passes through the circuit file are required:
 - the first pass to determine the size of the mapping table
 - the second to fill the mapping table, and
 - the third to set up the circuit information.
- 4 Example: Command>READ c17.ckt



Part 4: PC

input file: nothing

output file: nothing

- 1 This function prints out the circuit description from previous READ command.
- 2 Example: Command>PC
- 3 The output of the parser for circuit c17 is as follow:

Node	Type	In	Out
1	PI		10
2	PI		16
3	PI		8 9
8	BRANCH	3	10
9	BRANCH	3	11
6	PI		11
7	PI		19
10	NAND	1 8	22
11	NAND	9 6	14 15
14	BRANCH	11	16
15	BRANCH	11	19
16	NAND	2 14	20 21
20	BRANCH	16	22
21	BRANCH	16	23
19	NAND	15 7	23
22	NAND	10 20	
23	NAND	21 19	

Primary inputs: 1 2 3 6 7

Primary outputs: 22 23

Number of nodes = 17

Number of primary inputs = 5

Number of primary outputs = 2



Part 4: HELP

input file: nothing

output file: nothing

- 1 This function prints out help information for each command.
- 2 Example: Command>Help
- 3 Then you see the following info:

```
Command>help  
READ filename - read in circuit file and creat all data structures  
PC - print circuit information  
HELP - print this help information  
QUIT - stop and exit  
Command>
```



Part 4: QUIT

input file: nothing
output file: nothing

- 1 This function terminates the program.



Part 4: new function

- 1 You may write your program as a subroutine under main().
- 2 The following is an example to add another command 'lev' under main():

```
enum e_com{READ, PC, HELP, QUIT, LEV};
#define NUMFUNCS 5
int cread(), pc(), quit(), lev();
struct cmdstruc command[NUMFUNCS] = {
    {"READ", cread, EXEC},
    {"PC", pc, CKTLD},
    {"HELP", help, EXEC},
    {"QUIT", quit, EXEC},
    {"LEV", lev, CKTLD},
};

lev(){
    // add some code here!
}
```



Levelization Procedure



Part 5: Levelization Procedure

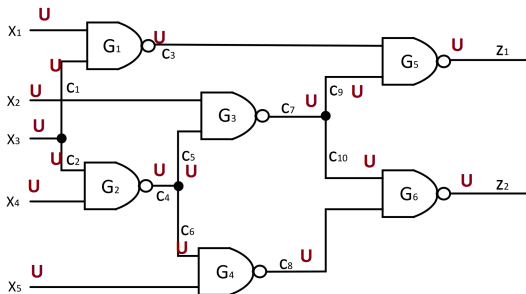
Procedure [InputLevelize()]

- 1 Initialization: For each circuit line c , $lev_{inp}(c_i) = \text{undefined}$
- 2 For each primary input x_i , $lev_{inp}(x_i) = 0$
- 3 While there exist one or more logic elements such that (i) lev_{inp} is defined for each of the element's inputs, and (ii) lev_{inp} is undefined for any of its outputs, select one such element.
 - If the selected element is a gate with inputs $c_{i1}, c_{i2}, \dots, c_{i\alpha}$ and output c_j then assign:
$$lev_{inp}(c_j) = \max[lev_{inp}(c_{i1}), lev_{inp}(c_{i2}), \dots, lev_{inp}(c_{i\alpha})]$$
 - If the selected element is the fanout system with stem c_i and branches $c_{j1}, c_{j2}, \dots, c_{j\beta}$, then for each output c_{jl} , where $l = 1, 2, \dots, \beta$, assign:
$$lev_{inp}(c_{jl}) = lev_{inp}(c_i) + 1$$



Part 5: Example

Example C17 (ISCAS 85)

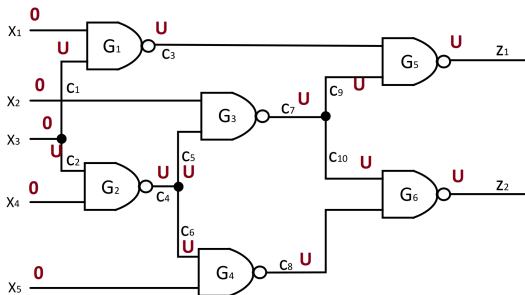


U=Undefined



Part 5: Example

Level of PI is zero

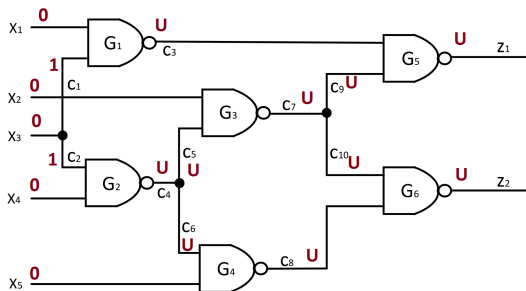


U=Undefined



Part 5: Example

Next step we select C1 and C2

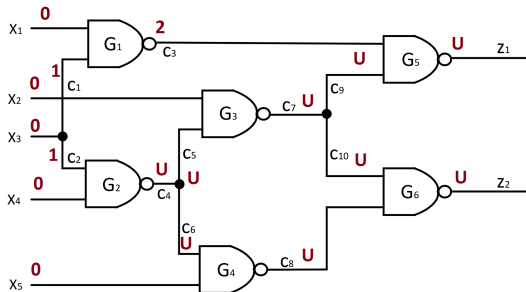


U=Undefined



Part 5: Example

Next step we select C3

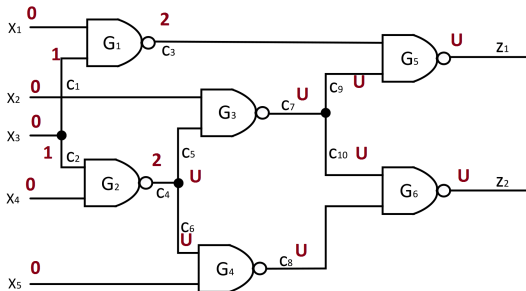


U=Undefined



Part 5: Example

Next step we select C4

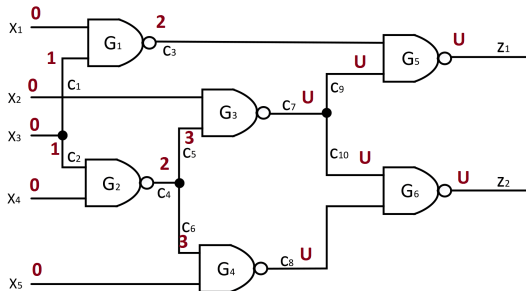


U=Undefined



Part 5: Example

Next step we select C5,C6

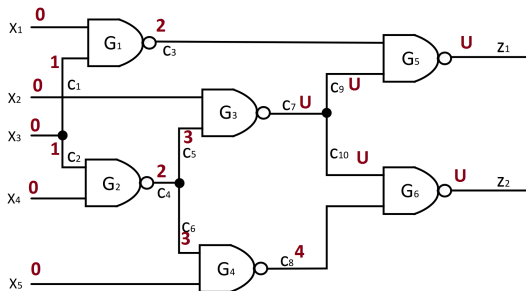


U=Undefined



Part 5: Example

Next step we select C8

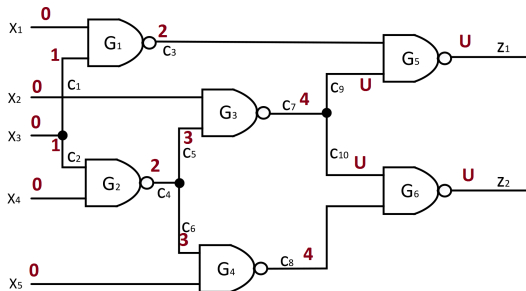


U=Undefined



Part 5: Example

Next step we select C7

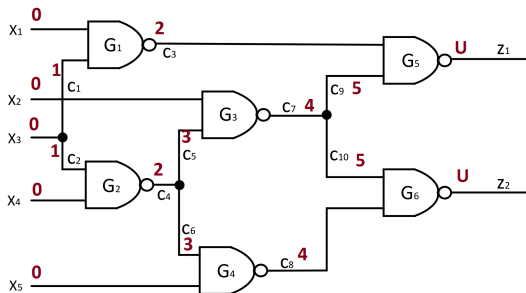


U=Undefined



Part 5: Example

Next step we select C9 and C10

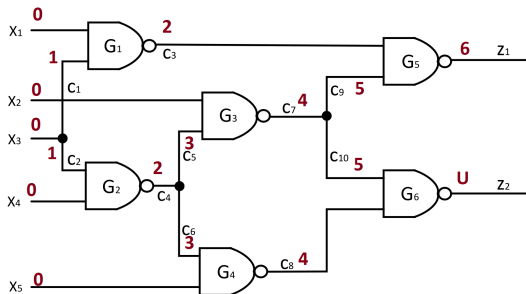


U=Undefined



Part 5: Example

Next step we select Z1

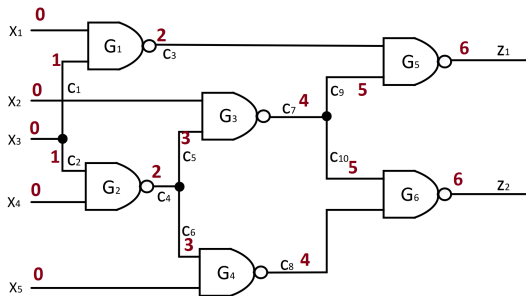


U=Undefined



Part 5: Example

Next step we select Z2



Levelization

- This kind of animation requires a JavaScript-supporting PDF viewer, such as Acrobat Reader.



What you need to do?

- The code should be compiled and run on **viterbi-scf** servers. This is a must! We use the same version of compiler as in the viterbi server.
- The programming language is either C or C++. In case you want to use C++, you need to make a few changes to the readckt.c file.
- Add a new function to the given code that does the levelization for a circuit.
- The command to run this function is: *lev output-filename*.
- The output-filename is an arbitrary string.
- We assume there are no loops in the input circuits.



Levelization output format

- The result of the output file is a text file with the following format:
 - line 1: **circuit-name** (without .ckt suffix, e.g. c17)
 - line 2: **#PI: number-of-PI**
 - line 3: **#PO: number-of-PO**
 - line 4: **#Nodes: number-of-nodes**
 - line 5: **#Gates: number-of-gates**
 - line X: **node-ID level-number** (separated by a single space, every node and level pair in a separate line)
- The order of the nodes is not important in this assignment.
- The golden output file is provided for c17.
- Possible extra line at the end of the is OK!



What to submit:

Upload **ONLY TWO FILES** (in a zipped folder) on the D2L:

- A **readckt.c** or **readckt.cpp** source file
- A text file "**compile.txt**" with one single line, which is the command that you use to compile your code on the viterbi server (example: *gcc -o parser readckt.c*)
- The name of your application (executable file) should be *parser*.
- Your code will be tested with many different circuits.
- Deadline: Saturday, October 15th, 2022.



How we grade you code:

- Grading is completely automatic via scripts. It is important that your output format exactly matches the golden output. **Be cautious** about possible text issues such as extra new lines, extra commas or spaces, etc.
- We will run the codes at the end of each day and will notify you about the results.

These arguments are **NOT** valid:

- My code compiles on my own computer but not on the server
- My output file is exactly the same as golden, but on my Windows
- The issue of my output is only a single extra space
- I used a comma to make my output more readable



Some additional notes

- In some CKT format circuits, basic gates have only one input. Please consider these gates as buffer.

