資料結構與程式設計

(Data Structure and Programming)

108 學年上學期複選必修課程 901 31900

Homework #6 (Due: 9:00pm, Saturday, Dec 07, 2019)

0. Objectives

1. Parsing a circuit from a description file and create the netlist as a graph.

2. Traversing the circuit graph. Report the netlist and connections.

3. Detecting floating and unused gates.

1. Problem Description

In this homework, we are going to implement a special circuit representation, called "AIG" (And-Inverter Graph), from a circuit description file. The generated executable has the following usage:

cirTest [**-File** <*dofile*>]

where the **bold words** indicate the command name or required entries, square brackets "[ ]" indicate optional arguments, and angle brackets "< >" indicate required arguments. Do not type the square or angle brackets.

This cirTest program should provide the following functionalities:

1. Parsing the circuit description file in the AIGER format (See Lecture note #09 and http://fmv.jku.at/aiger/ for details). Your program should be able to detect the following types of errors in the file (Note: both line and column numbers start from '1'):

   1. `Cannot open design "c17.aag"!!`
   2. `[ERROR] Line 1: Illegal identifier "aagg"!!`
   3. `[ERROR] Line 1, Col 5: Missing number of variables!!`
   4. `[ERROR] Line 1: Number of variables is too small (3)!!`
   5. `[ERROR] Line 1, Col 5: Extra space character is detected!!`
   6. `[ERROR] Line 6, Col 2: Missing space character!!`

```
7.  [ERROR] Line 9, Col 1: Illegal white space char(9)
    is detected!!
8.  [ERROR] Line 2: Missing PI definition!!
9.  [ERROR] Line 8: Missing AIG definition!!
10. [ERROR] Line 3, Col 1: Missing PI literal ID!!
11. [ERROR] Line 4, Col 1: Missing PO literal ID!!
12. [ERROR] Line 9: Missing "symbolic name"!!
13. [ERROR] Line 1, Col 14: A new line is expected
    here!!
14. [ERROR] Line 3: Literal "2" is redefined, previously
    defined as PI in line 1!!
15. [ERROR] Line 7: Literal "6" is redefined, previously
    defined as AIG in line 5!!
16. [ERROR] Line 3, Col 3: Literal "16" exceeds maximum
    valid ID!!
17. [ERROR] Line 3, Col 1: Cannot redefine constant
    (1)!!
18. [ERROR] Line 9: Illegal symbol index(a)!!
19. [ERROR] Line 9, Col 1: Illegal symbol type ()!!
20. [ERROR] Line 9, Col 1: Illegal symbol type (a)!!
21. [ERROR] Line 9, Col 5: Symbolic name contains un-
    printable char(27)!!
22. [ERROR] Line 10: Symbolic name for "i0" is
    redefined!!
23. [ERROR] Line 9: PI index is too big (3)!!
```

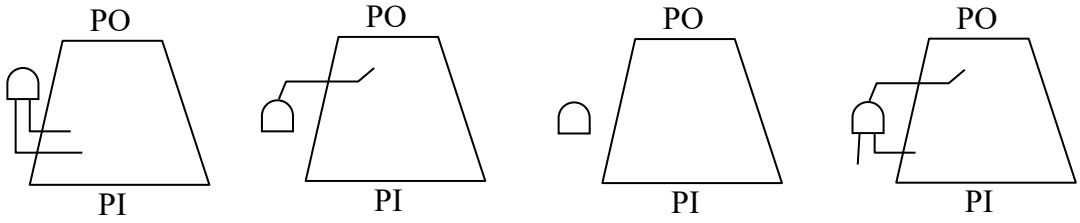
The above error messages are summarized from the 60+ testcases in the "tests.err" directory. Note that they are NOT intended to be complete. You may be able to detect different types of errors in different testcases. In addition, the error detection may depend on how you parse the circuit description file. Therefore, it is possible that you may detect different error(s) from the reference program. In our grading testcases, we will avoid the ambiguity. So even though you should try to make your error message aligned with ours, you don't need to spend the majority of your time in making your message exactly the same as ours for some minor differences. Moreover, you can stop the parsing at the first error you detect. You don't need to detect multiple errors in reading the circuit netlist. However, in case of error, please remember to clean up the partial circuit you have created.

Besides, to facilitate our grading, please use the error message printing function "*static bool parseError(CirParseError err)*" in "cirMgr.cpp" so that the error message can be the same as the reference program.

Although you can implement the circuit parser in C++, you can also consider using lex(flex)/yacc(bison). Other tools are **forbidden** in this homework.

Lastly, please note that handling this error message can be very tedious and time-consuming. **You are advised NOT to handle them in the beginning.** Please focus on parsing the GOOD circuits and continuing for the other commands. The error message handling is only a small portion of the total grades. Don't get stuck here!!

2. You can assume that the circuit is combinational. That is, no latch (flip-flop) is defined. Besides, you can assume that the netlist is a directed acyclic graph (DAG) and don't need to check for the loops. However, there may be floating or unused gates. That is, some gates may not be reachable either from primary inputs (PIs) or outputs (POs), or has a fanin that is floating (See the figure below). Your program should be able to report them.



(a) A gate that cannot reach any PO

(b) A gate that cannot be reached from any PI

(c) A gate that cannot be reached from PI and PO

(d) A gate with a floating fanin

In the above case (a), we call this gate "defined but not used". In case (b), we call it "a gate with floating fanins". More specifically, we call the floating fanins in (b) "referenced but not defined", and designate "UNDEF" gate type for them for netlist reporting (See examples in `CIRGate` command).

3. You should be able to report the circuit in various aspects: circuit statistics/summary, circuit status, circuit netlist in depth-first-search (DFS) order, lists of primary inputs/outputs, and list of floating gates, etc. Note that the DFS order should be constructed by recursive calls from the primary outputs in post-order traversal. That is, a gate will not be printed until all its fanins are reported. **The order of POs, as well as the order of fanins in a gate, should follow the orders as defined in the original AIGER file.** Floating gates of the cases (a), and (c) above should not be included in the DFS list (Note: cases (b) and (d) should be included) since they cannot be reached from POs. Constant gate (i.e. CONST0) should be included if it is visited during the DFS traversal. You should also provide commands to report the interconnections (i.e. lists of fanins) of a gate. Please see the corresponding commands for details.

4. Please DO NOT perform any optimization in circuit parsing, such as constant propagation, redundant gate removal, and equivalent gate merging, etc. (Note: These optimizations can be evoked by commands in the final project, though.)

5. The TODO's of this homework are all included in the *src/cir* directory. It contains a file *cirMgr.h* to define the circuit manager (i.e. `class CirMgr`) for circuit construction and reporting, etc. A global variable `CirMgr *cirMgr` as well as some member functions of `class CirMgr` should be defined in *cirMgr.cpp*. You should use this global variable in circuit-related

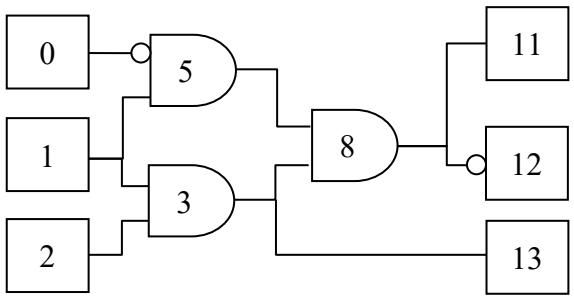
commands (defined in *cirCmd.cpp*) for all the circuit operations. The AIG gate data structure and its constructing functions should be declared and defined in the files *cirGate.h* and *cirGate.cpp*. You should at least define the `class CirGate`. It is up to you whether you want to declare inherited classes of `class CirGate` such as `class CirAigGate`, `class CirPiGate`, etc.

6. You should define the data members and member functions for the classes `CirMgr` and `CirGate` on your own. However, please follow the naming convention in other homework/classes and be cautious on the memory and runtime efficiency.

7. Each gate should have a unique gate ID, which is equivalent to the variable ID as derived in the AIGER file. That is, the IDs for PIs and AIG gates are the values of the corresponding literal IDs divided by 2.

8. For each PO, you should define a distinct PO gate whose single fanin is the AIG gate as defined in the AIGER file. See the following example:

```
aag 10 2 0 3 3
2
4
16
17
6
6 2 4
10 1 2
16 10 6
```



This allows an AIG gate to fanout to multiple POs (e.g. Gate 8) potentially with inverted phases, or to other AIG gate(s) (e.g. Gate 3). The IDs for POs are numbered from the maximum variable IDs (as defined in the header line, not the max AIG gate ID) plus 1 (i.e. 11 in this case). Note that they are different from the IDs defined in the PO section of the AIGER file. (See 3. for the order of POs)

9. There should be a mechanism (e.g. flags) to control the dependency of the circuit operations/commands. For example, the circuit netlist should be constructed before any other circuit operations.

2. Supported Commands

In this homework, you should support these new commands:

```
CIRRead:    read in a circuit and construct the netlist
CIRPrint:   print circuit
```

```
CIRGate:     report a gate
CIRWrite:    write the netlist to an ASCII AIG file (.aag)
```

Please refer to Homework #3 and #4 for the lexicographic notations.

2.1 Command "CIRRead"

Usage: **CIRRead <(string fileName)> [-Replace]**

Description: Read the circuit from the file "*fileName*" and store it in the global variable *cirMgr*. If the circuit has been read, issue an error "`Error: circuit already exists!!`" unless the option "-Replace" is issued. In the latter case (even with the same filename), delete the original circuit and discard all the circuit related data, create a new one, and print out the message "`Note: original circuit is replaced...`". Note that with "-Replace" option the original circuit will be deleted even if the new circuit construction fails. In such case, *cirMgr* should be reset to NULL (0). No warning or error message is needed if "-Replace" is issued when *cirMgr* == 0.

Example:

```
cir> cirread test.cir      // read in the circuit description file "test.cir"
cir> cirr -rep kk.cir      // read in the circuit description file "kk.cir"
                           // and replace the original circuit
```

2.2 Command "CIRPrint"

Usage: **CIRPrint [-Summary | -Netlist | -PI | -PO | -FLoating]**

Description: Report circuit information in different ways. If the circuit hasn't been constructed, print out an error message "`Error: circuit has not been read!!`". When "-Summary" is specified, print out the circuit statistics, and when "-Netlist" option is used, list all the gates in the topological (depth-first-search, DFS) order (Note: For the gates in DFS list only, those unreachable from POs won't be printed, even they are PIs). The "-PI", "-PO" and "-FLoating" options will report the lists of primary inputs (PIs), primary outputs (POs) and floating gates, respectively.

Example:

```
cirtest> cirprint              // print out circuit summary
Circuit Statistics
==================
  PI           20
  PO           12
  AIG         128
------------------
  Total       160
```

Note that the statistics of gates should be in the order: { PI, PO, AIG }, no other gate types. The numbers should be consistent with the numbers of PI, PO, and AIG gates in the circuit. If there is no gate for a specific type, print a '0' and do not skip this line. Note that when the circuit is read in, the numbers of PI, PO, and AIG gates must be the same as the numbers declared in the header of the AIG file.

cirtest> cirprint –netlist // output the netlist in depth-first-search order

```
[0] PI  1 (1GAT)
[1] CONST0
[2] AIG 10 1 0
[3] PI  2 (2GAT)
[4] PI  6 (6GAT)
[5] AIG 11 !1 6
[6] AIG 16 2 !11
[7] AIG 22 !10 !16
[8] PO  24 !22 (22GAT$PO)
[9] PI  7 (7GAT)
[10] AIG 19 *!15 7
[11] AIG 23 !16 !19
[12] PO  25 !23 (23GAT$PO)
```

Please pay attention to the alignment of the gate types and IDs. The IDs shown here are the variable IDs as derived from the original AIGER file. The '!' sign means the input is inverted. If the input is undefined (floating), we should put a '*' sign right before '!' and gate ID (e.g. `AIG 19 *!15 7`). If the names of the PIs/POs are specified in the "symbol section", print out the names at the end of the lines.

The aag file for the above case can be found at "…/tests.fraig/cirp-n.aag".

cir> cirp -pi // print out the PIs

```
PIs of the circuit: 1 2 7 6 3
```

The PIs are printed out as the same order as specified in the AIGER file. The numbers are their variable IDs. The POs should be printed by the "cirp -po" command similarly. Print "`PIs/POs`" even if there is zero or only one PI/PO.

cir> cirp -fl // report the floating gates

```
Gates with floating fanin(s): 7
Gates defined but not used  : 9 10
```

Print "`Gates`" even if there is zero or only one floating gate.

We report two types of floating gates:

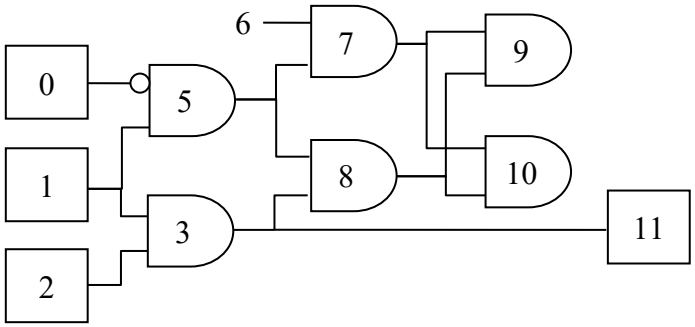
(1) Gates with floating fanin(s): referring to the gates with one or multiple undefined fanins. Cases (b), (c) and (d) in the floating-gate definition in Section 1.2 belong to this type. A PO with the fanin undefined should also be reported in this case.

(2) Gates defined but not used: referring to the gates that is defined in the AIGER file but are never used by other gates. In other words, it has no fanout. Cases (a) and (c) in the floating-gate definition in Section 1.2 belong to this type. A PI with no fanout should also be reported in this case.

Note that case (c) belongs to both types, so it should be reported in both places. Besides, to ease the checking of the reporting correctness, *please print out the gates of above two cases (for -Floating) in ascending order of their IDs*.

The netlist for the above example is:

```
aag 10 2 0 1 6
2
4
6
6 2 4
10 1 2
14 12 10
16 10 6
18 14 16
20 14 16
```



If we do "CIRPtint -Netlist", we will get:

cir> cirp -n

[0] PI 1
[1] PI 2
[2] AIG 3 1 2
[3] PO 11 3

2.3 Command "CIRGate"

Usage: **CIRGate <<(int gateId)> [<-FANIn | -FANOut><(int level)>]>**

Description: Report the gates in the circuit. If the "*gateId*" is not defined in the circuit, report "Error: Gate(*gateId*) not found!!". If the option "-FANIn" or "-FANOut" is NOT specified, print out the gate information including ID, name (for PI and PO), gate type, and line number (as appeared in the AIGER file). Otherwise, print out its fanin/fanout cone with the followed option (int level). Put proper indentations for different levels of fanin/fanout printing. If there is an inversion between a gate and its fanin/fanout, put a '!' before the fanin/fanout. If a gate whose fanin(s)/fanout(s) have been reported in previous lines, put a "(*)" after it is printed; do not repeatedly report its fanin(s)/fanout(s). If there is any undefined (i.e. floating) fanin, print "UNDEF" for its gate type.

Example:

cir> cirg 25 // print out gate (25)

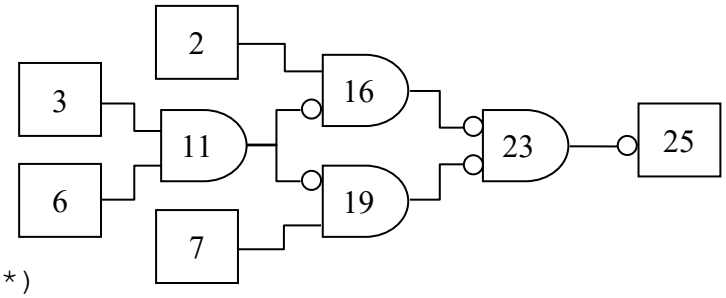
```
==================================================
= PO(25)"23GAT$PO", line 9                       =
==================================================
```

cir> cirg 19

```
==================================================
= AIG(19), line 12                               =
==================================================
```

cir> cirg 25 -fanin 4

```
PO 25
  !AIG 23
    !AIG 16
      PI 2
      !AIG 11
        PI 3
        PI 6
    !AIG 19
      !AIG 11 (*)
      PI 7
```



The above command reports the partial circuit as shown on the right-hand side. Please note that if a gate (e.g. AIG 11) whose fanin(s)/fanout(s) have been reported in previous lines, put an "(*)" after it and do not report its fanins/fanouts again in the later encounters.

However, do not put "(*)" for a gate if its fanin(s)/fanout(s) haven't been report. For the above example (AIG 11):

cir> cirg 25 -fanin 3

```
PO 25
  !AIG 23
    !AIG 16
      PI 2
      !AIG 11
    !AIG 19
      !AIG 11
      PI 7
```

If a floating (undefined) fanin is reported in certain gate's fanins, print "UNDEF" for its gate type.

cir> cirg 7 -fanin 1 // Please refer to the netlist in "CIRPrint" command

```
AIG 7
  UNDEF 6
  AIG 5
```

Optionally, if a floating fanin ID is issued by this command, you can either report "Error: Gate(*gateId*) not found!!", or print out as follows:

```
cir> cirg 6     // Please refer to the netlist in "CIRPrint" command
==================================================
= UNDEF(6), line 0                               =
==================================================
```

[Orders of fanins/fanouts] The order of fanins should be kept as the same as they are defined in the original file. However, since the order of fanouts depends on how you parse the circuit, we don't set the restriction here. Our grading testcases will try to avoid the ambiguity in checking the "CIRGate -fanout" command.

[Known Problem] If a gate *g* has two inverted fanins *f* and *!f*, then reporting fanouts from f will not be able to distinguish !g from g. This is a limitation in the reference program, and you DO NOT need to handle this.

2.4 Command "CIRWrite"

Usage: **CIRWrite [-Output (string aagFile)]**

Description: Write the netlist to an ASCII format AIGER file. The "-Output" option specifies the file name. If it is not specified, print out the AIGER output on the standard output (i.e. the monitor). In the header line, M(Max #vars), I(#PIs), L(#Latches), and O(#POs) should be the same as the original file, while A(#AIGs) should be set as the number of AIG gates in the DFS list. PI and PO definitions in the output file should be exactly the same as the original file, even if some of the PIs are unused or some of the POs are floating. The AIG section, however, contains only the AIG gates in the DFS list of the circuit and should be printed in that order. That is, the gates that cannot be reached from POs will not be included in the output file. The symbol section should print out the symbolic names of PIs and POs in the orders as in the PI and PO lists, even though these orders are different from those in the original file. Comment section is optional. You can leave it blank or write whatever you want.

Example:

cir> cirwrite -o C17-opt.aag   // write out the net list to "C17-opt.aag"

3. What you should do?

You are encouraged to follow the steps below for this homework assignment:

1. Read the specification carefully and make sure you understand the requirements.

2. Think first how you are going to write the program, assuming you don't have the reference code.

3. Type "make linux" or "make mac" to switch to a proper platform. The default is linux platform.

4. Understand the AIGER specification.

5. Define the circuit data structure.

6. Implement the circuit reader/parser which should include the following steps: (1) Parsing the netlist description file and constructing the circuit netlist, (2) Generating DFS list of the gates, (3) Checking floating gates, and (4) Checking unused gates. DO NOT handle parsing error message in the beginning.

7. Finish the circuit and gate reporting functions.

8. Test your implementation with the provided testcases. You are also encouraged to create more testcases on your own.

9. If you have time, handle the parsing errors.

10. [Note] The cir command interfaces are already given. You don't need to implement them. Actually, you are NOT allowed to modify cirCmd.{h,cpp}. They are included in the MustRemove.txt.

4. Grading

We will test your submitted programs with various testcases and compare the outputs with our reference program. Minor differences due to printing alignment, spacing, error message, etc can be tolerated. However, to assist TAs for easier grading work, please try to match your output with ours (especially the printed order of gates).