# 107-1 VLSI Testing

Programming Assignment #2

電子所碩一 R07943107 徐晨皓

1. **Please fill in the following table in your report.**

| circuit number | number of gates | number of total faults | number of detected faults | number of undetected faults | fault coverage | number of test vector | run time (s) |
|---|---|---|---|---|---|---|---|
| **C432** | 245 | 1110 | 149 | 961 | 13.42% | 20 | 0.01 |
| **C499** | 554 | 2390 | 2263 | 127 | 94.69% | 66 | 0.08 |
| **C880** | 545 | 2104 | 1254 | 850 | 59.60% | 65 | 0.04 |
| **C1355** | 554 | 2726 | 1702 | 1024 | 62.44% | 63 | 0.12 |
| **C2670** | 1785 | 6520 | 6278 | 242 | 96.29% | 135 | 0.13 |
| **C3540** | 2082 | 7910 | 2424 | 5486 | 30.64% | 98 | 2.08 |
| **C6288** | 4800 | 17376 | 17109 | 267 | 98.46% | 42 | 0.18 |
| **C7552** | 5679 | 19456 | 19144 | 312 | 98.40% | 289 | 0.78 |

2. **Please print out the critical parts of your code and explain it.**

(a) Find the input wire

```
// TODO find the input wire
// Hint similar to OR and NAND but different polarity
//-------------------------------- hole ---------------------------------------
    if (object_level) new_object_wire = find_hardest_control(object_wire->inode.front());
    else new_object_wire = find_easiest_control(object_wire->inode.front());
    break;
//-----------------------------------------------------------------------------
//  TODO
```

For AND and NOR gates, we find the hardest controllable input if the objective of the gate output is 1 and find the easiest controllable input if the objective of the gate output is 0.

(b) Find the input with min level

```
// TODO  find the input with min level
// Hint: similar to hardiest_control but increasing level order
//-------------------------------- hole ---------------------------------------
for (i = 0, nin = n->iwire.size(); i < nin; ++i) {
  if (n->iwire[i]->value == U) return n->iwire[i];
}
//-----------------------------------------------------------------------------
//  TODO
return(nullptr);
```

In this assignment, the easiest controllable gate input is defined as the gate input of the lowest level. Thus, we scan all the gate inputs in increasing level order. If the gate is not assigned any

value, the function will return the input immediately.

(c) Check if any fault effect reach PO

```
is_test = false;
//TODO check if any fault effect reach PO
//HINT check every PO for their value
//---------------------------------- hole ------------------------
for (i = 0, ncktout = cktout.size(); i < ncktout; ++i) {
  if ((cktout[i]->value == D) || (cktout[i]->value == B)) {
    is_test = true;
    break;
  }
}
//--------------------------------------------------------------------
//TODO
return is_test;
```

In this function, we check every PO value. If the value of a PO is D or D', the function will return *TRUE*. If none of the PO has value D or D', the function will return *FALSE*.

(d) Fault excitation

```
/* fault excitation */
//TODO fault excitation
//HINT use backward_imply function to check if fault can excite or not
//---------------------------------- hole ----------------------------------------
const int opposite_stuck_value = (fault->fault_type == STUCK0 ? 1 : 0);
pi_is_reach = (backward_imply(w, opposite_stuck_value) == CONFLICT ? CONFLICT : TRUE);
//------------------------------------------------------------------------------
//TODO
```

In this function, we confirm if the fault can be excited by PIs. We use *backward_imply* to check if the fault will cause any conflicts at PIs. Note that only if there is any conflict at PIs, the function will return *CONFLICT*; otherwise, the function will return *TRUE*.

(e) Search X-path

In this function, we check if there is any X-path from the wire *w* to POs. We use depth first search (DFS). Therefore, we maintain a stack to achieve DFS. When any output is discovered and its value is unknown, the function will return *True*. But if the stack is empty, the function will return *False* instead, which means that there is no path from the wire *w* to POs. See the bonus part for the

time complexity analysis.

```
//TODO search X-path
//HINT if w is PO, return TRUE, if not, check all its fanout
//------------------------------------ hole ------------------------------------
// 1. If w is PO, return TRUE
if (w->flag & OUTPUT) return true;
// 2. If not, check all its fanout (DFS)
forward_list<wptr> wire_stack;
unordered_set<int> hash_marked_wires;
wire_stack.push_front(w); // push to stack
hash_marked_wires.insert(w->wlist_index); // Mark discovered wires
while (!wire_stack.empty()) {
  wtemp = wire_stack.front(); // stack pop
  wire_stack.pop_front();
  for (i = wtemp->onode.size() - 1; i >= 0; --i) {
    for (wptr wout : wtemp->onode[i]->owire) {
      if (hash_marked_wires.count(wout->wlist_index) == 0
          && wout->value == U) {
        if (wout->flag & OUTPUT) return true;
        wire_stack.push_front(wout);
        hash_marked_wires.insert(wout->wlist_index);
      }
    }
  }
}
//------------------------------------------------------------------------------
return false; // X-path disappear
```

3. **(Bonus) Please analyze the complexity of function, *trace_unknown_path*. Can you implement this function with O(n) complexity where n is the number of nodes? What is the tradeoff of your implementation? Please explain what you did in your report.**
   (a) Yes. This function can be implemented with O(n) time complexity (suppose the number of the gate fanouts is a constant). Here, if we maintain a hash table, we can mark a gate as discovered with O(1) time complexity and query if a gate is marked with O(1) time complexity. If a gate is marked, it will not be pushed to the stack again, which avoids to explore the same path.
   (b) The tradeoff is between the memory usage and the runtime. The hash table will need lots of memory. But if the hash table is large enough, the query operation will be more efficient.