Logic Design for VLSI – H.W 3

Submitters:

Lidor Zino I.D: 313551186

Gal Kaner I.D: 207910738

Date: 02/02/2025

a. What is the basic code structure/high-level algorithm?

The code follows a "read–parse–flatten–encode–compare" structure. In summary, the main steps are:

1. **Command–Line and File Processing:**
   As provided in the template, the program reads command–line arguments to obtain the names of the specification and implementation Verilog files, as well as the name of the top-level cell for each design.

2. **Parsing and Flattening:**
   As given in the template, the code uses the HCM API to parse each Verilog file and then "flattens" the design into a single-level representation. This flattening produces a top–cell with explicit connections between all gates, and it also collects the "primary" ports (the inputs and outputs declared on the top–cell).

3. **Primary Ports Comparison:**
   The lists of primary inputs and outputs are extracted (and sorted) from both the specification and the implementation. If they do not match, the program immediately reports non–equivalence.

4. **CNF Generation:**
   For each circuit (both spec and imp), the code walks over each gate in the flattened netlist and encodes its behavior into CNF clauses. A helper function (using Tseytin-like transformations) is used for each gate type. For example, a buffer is encoded by two clauses enforcing the equality between its input and output, and an XOR gate is encoded by four clauses (see below).

5. **Miter Construction and Comparison:**
   After both circuits are encoded, the code constructs a "miter" by adding clauses that capture the difference between each corresponding primary output of the two circuits. (For each pair of outputs, a new variable is introduced and constrained so that it is true exactly when the two outputs differ.) In effect, the miter is satisfiable if there is an assignment to the primary inputs that makes at least one pair of outputs differ.

6. **SAT Solving and Output:**
   Finally, the CNF is written to a file and the solver's API is called (with a call to simplify() and then solve()). If the CNF is satisfiable, then a counterexample

exists and the circuits are not equivalent; if unsatisfiable, then the circuits are equivalent.

b. Provide an exact explanation of the comparison algorithm.

The comparison algorithm works by "mitering" the two CNF encodings of the circuits as follows:

- **Interface Matching:**
  First, the code verifies that the primary (top–level) inputs and outputs match between the specification and the implementation. If they do not, the circuits are immediately non–equivalent.

- **CNF Generation:**
  The internal logic of each circuit is encoded into CNF. Each gate's behavior is represented by clauses. (See below for details on constants and XOR.)

- **Miter Construction:**
  For every corresponding pair of primary outputs (one from the spec and one from the imp), the algorithm creates a fresh "difference variable" (say, *d*). The variable *d* is constrained by four clauses so that *d* is true if and only if the two outputs differ (i.e. if the XOR of the outputs is true). Once all difference variables are created, an additional clause is added that is the disjunction of all these difference variables. This clause forces that at least one difference variable must be true if there is any discrepancy between the two circuits.

- **Solving:**
  The SAT solver is then run on the miter CNF. If the formula is **satisfiable**, that means there is an assignment to the primary inputs that makes at least one pair of outputs differ (a counterexample). If the formula is **unsatisfiable**, then no such input exists and the two circuits are equivalent.

c. How are constants being handled?

Constants (i.e. the global nets "VDD" and "VSS") are handled by a helper function called getOrCreateVar. This function works as follows:

- When a signal is encountered, if its name is "VDD", the function creates a new variable (if one has not been created already) and then adds a unit clause (i.e. a clause consisting of just that literal) that forces the variable to be true.
- Similarly, if the signal is "VSS", the function forces its variable to be false.
- For all other signals, a new variable is created (or the existing one is returned).

This approach ensures that any gate connected to a constant net will "see" the proper fixed Boolean value.

d. How are XOR gates being supported? Show the detailed Tseytin development if you use it.

For a XOR gate the desired behavior is - true when exactly one of *x* or *y* is true.

A standard Tseytin encoding for XOR is to introduce a fresh variable *z* for the gate’s output and add clauses that enforce the equivalence:

$$z \leftrightarrow (x \text{ XOR } y)$$

One common CNF formulation is:

1. $(x' + y' + z')$
2. $(x + y + z')$
3. $(x + y' + z)$
4. $(x' + y + z)$

verification:

- **Case 1:** $x = 0, y = 0$
  Clause 2 becomes $(0 + 0 + z')$ which is satisfied only if z' is true (so $z = 0$).
  The other clauses are automatically satisfied.
- **Case 2:** $x = 0, y = 1$
  Clause 3 becomes $(0 + 0 + z)$ forcing $z = 1$.
  The other clauses are automatically satisfied.
- **Case 3:** $x = 1, y = 0$
  Similarly, clause 4 forces $z = 1$.
- **Case 4:** $x = 1, y = 1$
  Clause 1 forces $(0 + 0 + z')$ so that z must be 0.

In our code the encoding is written as:

```
    } else if (gateType == "xor" || gateType == "xor2") {

      // XOR: Z = A ^ B

        solver.addClause(~mkLit(varMap[inputs[0]]), ~mkLit(varMap[inputs[1]]),
~mkLit(varMap[outputs[0]]));

        solver.addClause(mkLit(varMap[inputs[0]]), mkLit(varMap[inputs[1]]),
~mkLit(varMap[outputs[0]]));

        solver.addClause(mkLit(varMap[inputs[0]]), ~mkLit(varMap[inputs[1]]),
mkLit(varMap[outputs[0]]));

        solver.addClause(~mkLit(varMap[inputs[0]]), mkLit(varMap[inputs[1]]),
mkLit(varMap[outputs[0]]));
```