HW 2 – Dry Part:

1) We used 2 "queue" data structures to hold the elements in the netlist.
   By using HCM properties we added a "Value" field to the "hcmNode" structure and "Visited" field to the "hcmInstance" structure.
   Every time that node's value gets updated, the logical gate that is connected to that value through "IN" port is inserted to "gate_queue". In a same way, when a logical gate's output is updated, the node that is connected to the relevant "OUT" port is inserted to the "node_queue".
   *Within the queue there is no priority on which element should be dealt first, since all of the events (or gates) that are held in a the same queue are modeled to happen simultaneously.
   We used "vector" of ranked instances to calculate initiative state of the model.
   To support DFF we used a separate "vector" data structure which held the DFF instances, ordered by descending rank.

2) The algorithm of schedule control works in a follow way:
   a) Applying input vectors on input nodes (as "Value" parameter).
   b) Push to "node_queue" all the nodes that signal's value got changed.
   c) Go through all DFF primitive gates (if there are any) and push their output if clock=1. If DFF output changed – add the relevant node to "node_queue"
   d) Pop all the elements of "node_queue" one by one, for each element (node) find all instances that are connected to this node through "IN" port and add those instances to the "gate_queue". By end of this step "node_queue" is supposed to be empty.
   e) Pop all the elements of "gate_queue" one by one, for each element (instance) perform logical gate evaluation and if the output signal of that instance got changed, push a node that is connected to the instance's "OUT" port to the "node_queue". By the end of this step "gate_queue" is supposed to be empty.
   f) If "node_queue" is not empty go to "d" step, else take next input vector, increase the time and jump to "a" step. If it was the last input vector – finished.

      The algorithm is looped in 2 loops, the internal loop will break only if all the signals are stabilized, hence both queues are empty. The external loop will follow until the program reads all the input vectors.

Pseudo-code: (for simplification, in pseudo-code we are assuming that each element in queue is unique, in actual code we are implementing it)

```
for  (current_vector != NULL ) {
    for each signal in vector {
            if (signal changed) {
                    add connected node to node_queue
    for each dff instance  {
            calculate new output
            if output changed {
                    add connected node to node_queue
            }
    }
    for each node in node_queue {
            add connected instances to gate_queue
            remove node from node_queue

            for each instance in gate_queue {

                    calculate new gate's output

                    if output changed {

                            add connected node to node_queue

                    }
            }
    }
Increase time
}
```

3) In our model we are assuming that the time is progressing by one time unit each time new vector is applied, meaning that internal logic (combinatorial and procedural) is based on zero delay mode. Therefore, there is no need to model any time within a single cycle.

4) We can only simulate cells that are included in "stdcell.v" or are built from them.

   We can't have loops in a simulation, meaning that output signal goes back to input, this will lead to the endless loop simulation.

5) We are initiating our circuits values at first to ensure DFF are 0 at start, and that circuit is at valid state:

```
initiate all nodes value to 0
rank all instances
calculate outputs for all gates

for  (current_vector != NULL ) {
    for each signal in vector {
            if (signal changed) {
                    add connected node to node_queue
    for each dff instance in dff vector {
            calculate new output
            if output changed {
                    add connected node to node_queue
            }
    }
    for each node in node_queue {
            add connected instances to gate_queue
            remove node from node_queue

            for each instance in gate_queue {

                    calculate new gate's output

                    if output changed {

                            add connected node to node_queue

                    }
            }
    }
Increase time
}
```

After learning we need to support primitive DFF simulation with our code, we added a separate approach to DFF gates to evaluate them only once at each cycle, before evaluating other lower-level gates. This included adding a "vector" data structure that is ordered in descending rank of the dff gates, and a loop to iterate through the dff vector and update their output if needed.
To support the gate level DFF implementation (using other primitive gates) we didn't have to apply any changes to our approach.

6.1 Our complexity per n gates in the circuit is described below (assuming gates are also limiter of signals, nodes. Also, disregarding input vector length effect on complexity):

initiate all nodes value to 0 – O(n)
rank all instances – O(n*log(n))
calculate outputs for all gates O(n)

in each input vector:
update all signals nodes – O(n)
update dff ouptputs O(n)

loop through node_queue and gate_queue until stabilized – $O(n^2)$:
assuming in worst case we have one gate's new output, causing changes in all the rest of the circuit's gates - calculate all the gates (n) for n times.

In total our code is running with complexity of $O(n^2)$.

We could improve our complexity, for example, by iterating gate_queue per the gates' rank , ensuring that we don't calculate the same gate twice in same time progression.

6.2 For simulating each gate, we are using "eval_gate" function.

It iterates on the gate's ports :

If port's direction is IN : re-evaluate the output value, per the type of gate (orX, andX, DFF etc.).

If port's direction is OUT : save the out port.

Once we iterated through all ports, we can store the calculated output value to the node connected to out port.

6.3 We did not use an event queue, but have implemented "node_queue" and "gate_queue" that are used to update each other in any simulation time progression.

For each time progression we have the node_queue holding any nodes affected by input signal changes, and we add the nodes' gates to gate_queue. If the gates have changed their output – we add their output port's connected nodes to node_queue, and repat the process until node_queue is empty – meaning the simulation stabilized for the current time simulated.