Mini Project: Verilog Netlist Optimizer

Submitted for: Digital Design 2

Professor: Mohamad Shalan

Names and IDs:

Ramez Moussa 900171864

Hany Moussa 900171833

Table of Contents

Objective:

This project aims to optimize Verilog Gate-Level netlists. It focuses on the aspect of cell's fanout. To reduce the fanout of the cells, two main approaches are utilized:

1. Clone the high-fanout cells to reduce the fanout of the driving cell
2. Add a buffer tree in between the driving cell and its fanout to reduce the fanout of individual cells

Moreover, adding buffers or cloning the cells usually results in a larger total delay. To solve this issue, we added the functionality to size up the high fanout cells to reduce the total delay.

Assumptions

During our development, we had to make some assumptions to for the sake of simplicity:

1. We are assuming that the gate-level netlist would include cells with only 1 output port
2. It is assumed that for all cells, the input transition is always the second value in the None Linear Delay Model (NLDM) table.
3. Whenever the load capacitance is outside the provided range in NLDM table, linear extrapolation is utilized.
4. We are assuming that the total delays of all cells are representative of the worst-case delay which might not be the case all the time.
5. Another assumption we made was that the output delay of any cell was relative to the first input pin.

Limitations

1. Unfortunately, our assumptions and the algorithms we utilized have some limitations:
2. As mentioned in the assumptions, our project might not operate correctly if cells with multiple output ports are used.
3. The sizing algorithm utilized has an exponential complexity. Consequently, it requires a lot of time and memory to provide the results. This is just a naive approach for sizing just to show its effect on the total delay.
4. The cloning algorithm can reach the maximum recursion depth if the maximum fan out is small and there is a large number of cells. To solve this, we added an iterative version of the algorithm.

Algorithms used:

Buffering

For buffering, a naive algorithm is used. Simply, loop over all the cells in the netlist and check for violations. When a violation is encountered, the number of buffers (nBuffers) to be used is obtained by the following equation:

```
if(currentFanOut%maxFanOut == 0):
    nBuffers = (int(currentFanOut/maxFanOut))
else:
    nBuffers = (int(currentFanOut/maxFanOut)) + 1
```

Then, start assigning the specified max fanout amount to each buffer of the assigned ones. In the last buffer, assign only the remaining fanouts to as it might be less than the specified maximum fanout (the case if the fanout before optimization is not divisible by the number of buffers used. After buffering, check if the number of buffers used (nBuffers) exceeds the maximum specified fanout; if so, recursively call the function to solve this issue. This results in a buffer tree.

Recursive Cloning

For the recursive cloning function, a slightly more complex algorithm than buffering is used. Similar to the buffering algorithm, loop over all the cells in the netlist and check for violations. When a violation is encountered, the number of clones of the cell (nClones) to be used is obtained by the following equation:

```
if(currentFanOut%maxFanOut == 0):
    nClones = (int(currentFanOut/maxFanOut) - 1)
else:
    nClones = (int(currentFanOut/maxFanOut))
```

After obtaining the number of clones, start assigning the specified max fanout amount to the current cell as well as each clone. In the last cloned cell, assign only the remaining fanouts as it might be less than the specified maximum fanout (the case if the fanout before optimization is not divisible by the number of clones used. After cloning, a check on the fanouts of all the inputs of the current cell must be made. If the fanout of any of the inputs to our cell (after adding the clones) exceeds the maximum specified fanout, recursively call the function with those cells to solve this issue. This is a bit problematic as in the worst case scenario, it would

result in an exponential amount of recursive calls and due to the use of python, the recursive depth is specified to 3000 calls only which might not be sufficient for cells with very large fanouts.

Iterative Cloning

This version for cloning could be used in some cases when the recursive version fails to provide results. As mentioned earlier, the recursive cloning function might have a very large amount of nested recursive calls which might lead to a **'maximum recursion depth'** in python. Alternatively, an iterative version is implemented. The algorithm is as follows:

- Loop while the cell the with highest fanout doesn't exceed the specified maximum fanout
- Apply cloning to the cell with the highest fanout

This is a purely greedy approach that is relatively inefficient. Additionally, this greedy approach follows the 'hill climbing' and therefore, only allows obtaining a local minima. Hence, sometimes, the absolute minimum fanout for the graph might not be achieved. To give an illustration, if we have a flipflop whose input is buffered and fed to itself, and both the buffer and the FF have the largest fanouts, then cloning any of them would increase the fanout of the other (so it becomes the maximum and consequently the one to be cloned) which could lead to an infinite loop.

Sizing

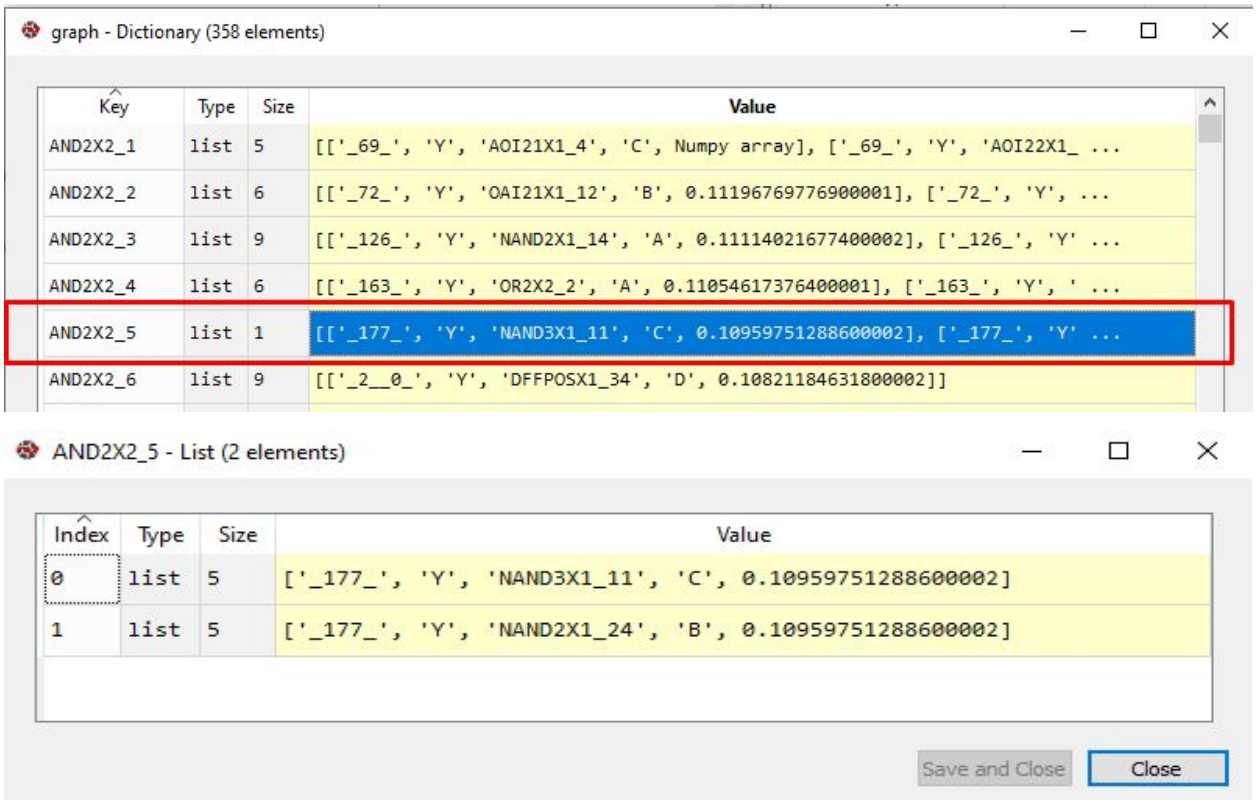
A greedy bruteforce algorithm was used for the reduction of the total timing delay by sizing. The algorithm utilizes the idea of 'hill climbing'. Starting from the first instance in the instances dictionary:

- Check if the cell has a higher fanout than the one specified by the user; if so:
    - Replace the cell with a higher size
    - Calculate the new total delay
    - If the delay is shorter than the previous delay:
        - If so: keep the change
        - Else: reverse the change
    - Else: go to the next cell

The Design:

Graph

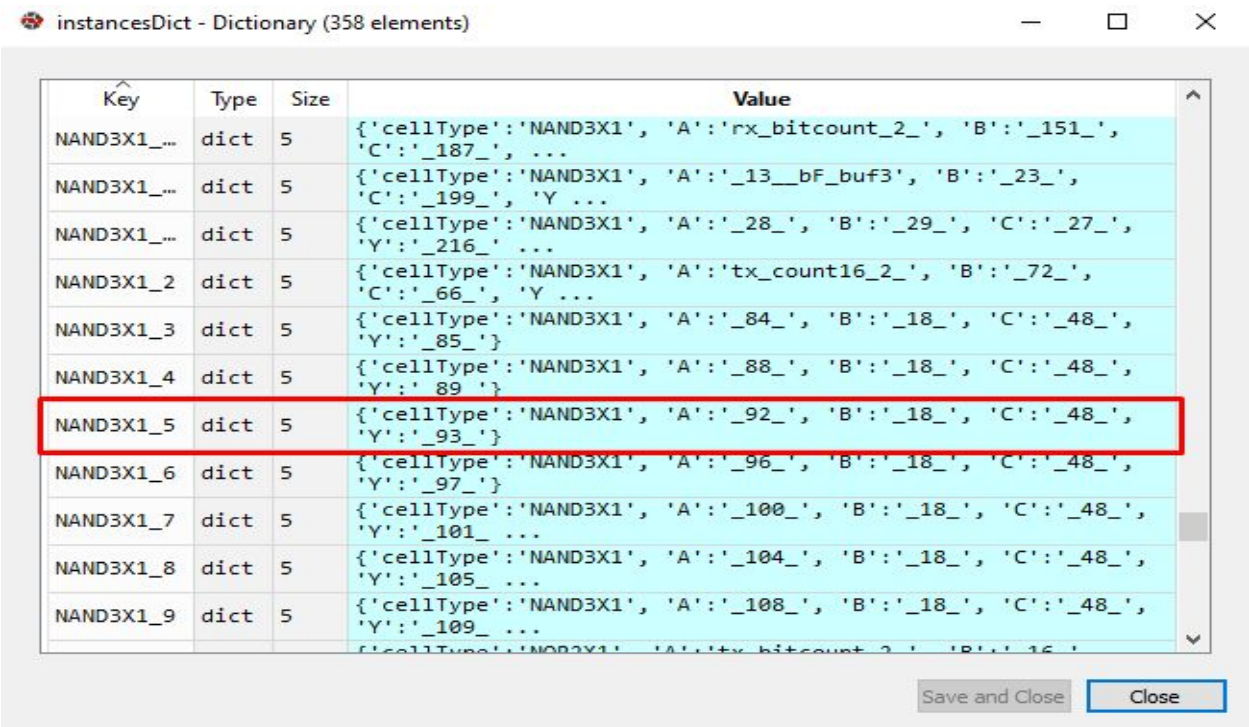
In this project, we represented the netlist as a Directed Acyclic Graph (DAG). Using pyhton, this was implemented as a dictionary of lists, called graph. An instance in the dictionary is accessed by the instance name (BUFX4_6 for example). Each instance accesses a number of lists that give data about all the instances our cell is connected to (including which pin) and the delay to all the fanout cells.

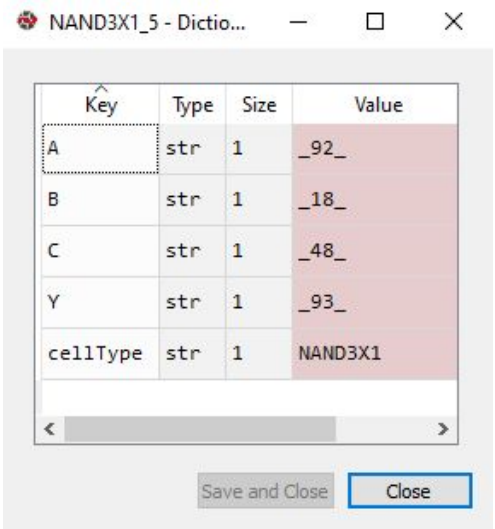


In this example, an illustration of the graph data structure that was implemented. The example shows an instantiated cell called **AND2X2_5**. This cell has a fanout of 2 and its pin **'Y'** is connected using wire **_177_** to pin **'C'** in **NAND3X1_11** and pin **'B'** in **NAND2X1_24.**

Instances Dictionary

InstancesDict is a data structure for all the netlist details for all the instantiated cells. It is implemented as a dictionary of dictionaries. InstancesDict is accessed using the instance name. It provides information about the cell type (i.e. AND2X2, OR2X2, etc…), all the pins in a cell and the wires connected to each one of them.



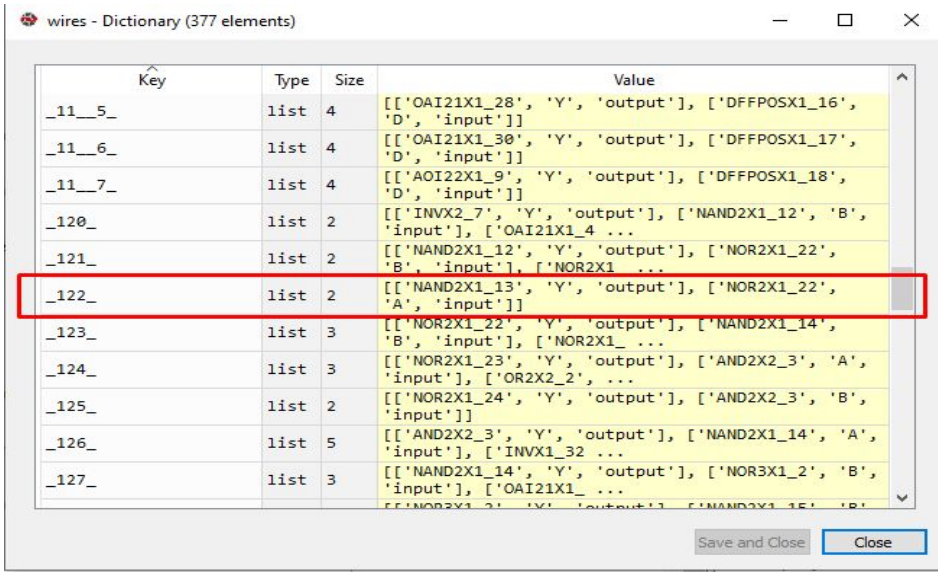


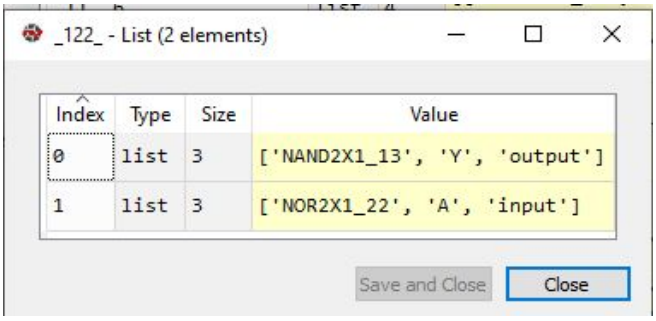
This illustration shows how the instancesDict data structure that is implemented and an example of one of its entries called **NAND3X1_5.**

This cell is of type **‘NAND3X1’** and it has 4 pins: **‘A’** is connected to the wire **_92_** , **‘B’** is connected to the wire **_18_** , **‘C’** is connected to the wire **_48_** , and **‘Y’** is connected to the wire **_93_.**

Wires

Wires is a data structure that stores information about all the wires in the netlist. It is implemented as a dictionary of lists. The details of a wire are accessed by the wire name. An instance in the wires dictionary gives details regarding the source and destination cells of this wire and the pins they are connected to.





The screenshots above show an illustration of the wires data structure that was implemented. The example shows a wire named **_122_**. The source of this wire is pin **‘Y’** in the cell called **NAND2X1_13** and the destination is pin **‘A’** in the cell called **NOR2X1_22**. Please note that there could be multiple destinations (lists that are labelled ‘input’) but there’s always only one source (lists that are labelled ‘output’). **‘Input’** simply means that this wire is an input to that cell (the wire’s destination. **‘Output’** simply means that this wire is the output of that cell (the wire’s source).

Functions

parseNetlist

A function that takes a file name for a gatelevel netlist and parses it. tI initializes the instances dictionary with all the instances created (accessed by instance name). Each entry in the instantiations dictionary has all the pins of the cell and the wires connected to them. It also initializes the wires dictionary; itis accessed using the wire name and gives information about all the cells it is connected to and whether it is the cell's output or input. Finally it stores the upper section of the gatelevel netlist so that it is printed later to the final .v file.

getPinCapacitance

This function gets the capacitance of a given pin in a cell. It uses the liberty file to access the desired cell. Then, it uses the specified pin name to return its capacitance.

getColumnDelay

Get the column from the characterization table containing the delay of a cell. We use the second column in the table (as we are assuming the input transition to have the middle value, for simplicity)

displayAsAnInstantiation

Display a cell from the instances dictionary as an instantiation. It uses the same format of the gatelevel netlist. This function is used to output the optimized verilog netlist when the optimization process is over.

getDelay

This function takes the capacitance array and the delays array. Based on the given inputs, it interpolates/extrapolates to obtain the delay corresponding to a target capacitance. The function uses the ‘scipy.interpolate.interp1d’ function. Interpolation in only 1 dimension is used as an input transition is assumed for all the cells.

getTotalDelay

This cell iterates over all the cells (using the graph) and sums the delays of all the cells.

**Please note that the delay of a cell is only considered once even if it has a fanout larger than 1. Only the maximum value is considered.

printNumberOfCellsOfEachType

This function counts the number of cells of each type and prints them. It simply iterates over all the instances in the instancesDict and keeps count of how many times a cellType (such as ‘AND2X2’) has been used.

constructGraph

This function uses the wires list to construct the graph. While doing so, it takes into consideration the fanout of each cell in order to calculate the delays of the cell. The delay of a cell to another cell in the path is represented using an edge in the graph.

fixByCloningSingle

This function solves the violation of a single high fanout cell using cloning. It assigns up to the maximum legal fanout to each cell and clone until all the fanouts have been divided between the clones in such a way that none of them cause violations. However, in doing so, adding clones undoubtedly increases the fanout of the previous cells driving the current cell. That being said, the function is only concerned with the current cell so no checking for violations caused is necessary.

fixByCloning

This function solves the violation of a single high fanout cell using cloning. It assigns up to the maximum legal fanout to each cell and clone until all the fanouts have been divided between the clones in such a way that none of them cause violations. In doing so, adding clones undoubtedly increases the fanout of the previous cells driving the current cell. Unlike fixByCloningSingle, this function is recursively called to fix any violations it might have caused.

removeViolationsByCloning

This is the generalized function that removes violation via cloning. It simply loops over all cells using the graph data structure and calls the fixByCloning function to fix individual violations (if exists).

fixByBuffering

A function that fixes the fanout of a single cell by adding buffers. It creates a specific number of buffers in such a way that the fanout of the current cell could be divided between them equally causing no violations. Then, the function checks if the number of buffers added exceeds the maximum legal fanout.. If so, the function is recursively called to fix any violations it might have caused .

removeViolationsByBuffering

This is the generalized function that removes violation by adding buffers. It simply loops over all cells using the graph data structure and calls the fixByBuffering function to fix individual violations (if exists).

constructAndDisplay

Constructs the graph and displays the number of cells of each type.

getCurrentMaxFanOut

Gets the maximum fanout in a given netlist.

writeToFile

Writes the current netlist (potentially optimized) to a the OptimizedNetlist.v file.

displayGraph

Displays all the nodes in our graph (cells) and the edges (connections between cells) and their direction, weight, and related pins.

displayMenu

Display the main menu with all the options to allow the user pick their desired operation

reopenMenu

Ask the user if they want to reopen the menu or terminate the program.

updateSizing

A function that uses a bruteforce algorithm to try to size every single cell and check the new delay. If it gets improved, then keep the new size. Else, return the cell to its original size.

replaceCell

A function that replaces a specific instance with a different size. It is used to replace the small cells with larger sizes.

updateWire

A function that updates a wire by replacing the old instance name by the new one .