Lucy Hu

CSE 143 Final Report

Part 1: Explanation of my code

The programming language that I am using is c++. There are 7 functions that I implemented:

1. main()

   In the main function, I read in the input file. I read in all the inputs and outputs and set them to variables. In main, I also created a 2D array called list that keeps track of the onset/dcset products for each output. The list array elements will constantly be changing through the process due to remove_implicants. I have another 2D array called out that keeps track of if the elements in "list" is a dcset or an onset. I am also able to keep track of the number of literals and products when reading in the file. Then I open an output file and then use a for loop for each output and call minimize and write. Then the output of write will be written into the output file.

2. write ()

   The write function is how I keep track of the literals and products of my optimization. The write function will read in from the list and based on the output at the parameters, it will read in the list of onsets and dcsets. It will only write out the product if element is not an empty string.

3. minimize()

   The minimize function is what starts the logic minimization. It reads in the list array and goes in a loop. If that element is not an empty string and the equivalent element for the out array is a 1, it will start going through the element's input position from 0 - # inputs and calls expand on that position. If the expand function returns true, then that means that the string at that position can be turned into a don't care. When that happens, remove_implicants is called to remove from the list products that are valid for the new expand string and the new expand string will be updated with '-' at the position of the string in the list.

4. expand()

   The expand function is relatively simple. Given the string/product, the position that the product is trying to expand in the parameters, We will set two validity or valid functions where that position in the product is either a '0' or a '1'. For the sake of optimization and execution time, I have created two optimizations for checking validity. If the input is too large, mostly over 15 inputs, then the execution time runs too slow. So we will use the valid() function to check for validity of the product and if the input is small enough to have a fast execution time, then it will be calling the validity() function to check for validity. If both functions that check for validity where the position is a '0' and a '1' returns true, then that means that the product/string is able to expand and then expand will return true, else false.

5. validity()

The validity function is a bit special. This function is a recursive function that will be called multiple times. In this function, it will go through the list and check if the product that we are trying to check in in that function, if it is, then it will return true. If it runs through the whole list and does not find the product, then we will continue and break down the product because it's possible for a '1' or a '0' to be in a dont care '-' value of the product. So now there is a for loop that checks each position of the product. If the position is a don't care value '-', then we will check for validity once again by recursively calling validity() twice but this time with products that have '1' and '0' replacing the '-'. This will then check for if this combination of the product is in the list. If it is in the list, it will return true, and if not, it will go through the process again until there are no more dont cares in the product and return false. If both validity are true, then it will return true and stop the for loop. But if one or both are false, then we will continue to try all combinations but this time with a '-' in the position again. This is for a case where the later positions happen to have a '-'. So in the end, this function will eventually return all possible combinations of the string. If there is any chance that it is false, it will eventually return false for all validity calls. Because of the consecutive recursive calls, the execution time for large inputs will be exponentially higher. For the pla files with 20 or more inputs, it will never stop running because it will take a long time for it to run the validity check. This is where a new optimization comes into play which is the function valid().

6. valid()

   As mentioned above, this is another optimization for larger inputs because the execution time is just too large to run validity(). The only downside for this optimization is that it will not be as optimized/through as the validity function. The valid() function is fairly simple. Given the product with a large input, it will go through the list and check if it is in that list, if it is in the list, it will return true. If it is not in the list, it will always return false. This way if the exact string we are looking for is not there, then it will return false and will not expand. But if it is in the list, then that is the exact product that takes in all of the possible minterns in the onset for it to be able to expand. So then valid() will return true.

7. remove_implicants()

   The remove_implicants function is where the items in the list will be removed. So given the new product that has expanded. We will start by checking for every element in the list starting from the element that is currently being minimized to see if the position matches the new product equivalent position. To do this we need to have a flag variable that keeps track of whether it is true or false, so at each increment in the list, we start off the flag as false. So if the product's position is a '-', then we will set the flag to true, if the position of the element is equal to the position of the product, then we will also set the flag to true. Otherwise we will set the flag to false and break out of the for loop incrementing the position and go straight to the next element in the list. If it ends up being true all through the increment of position, then the flag is checked to see if it is true

and if it is, then we will clear the element in the list to be an empty string. This will make sure that the same product is not accessed over again.

I did not end up creating a reduce function but the way I set up the design allows me to only check for validity of the products only once if it is actually valid. This is done because the list is only checked starting from the row that is being minimized at remove_implicants(), validity() and valid(). This ensures that after a product has been minimized, it will not be considered in any future minimizations of other products.

Part2: SIS1.3

Using alu.pla

a) Free_dc

Write_slif response:

```
.model alu.pla;
.inputs m s1 s0 ci ai bi;
.outputs fi co;
.type input_arrival %f %f ;
.type output_required %f %f ;
.type input_drive %f %f ;
.type output_load %f ;
[4] = m s1 s0 ci ai' bi;
[5] = m s1 s0' ci ai bi;
[6] = m s0' ci ai' bi';
[7] = m s0 ci ai bi';
[8] = m s1' s0' ci ai';
[9] = m s1' s0 ci ai;
[10] = s1 s0' ci' ai' bi;
[11] = m' s1 s0' ai' bi;
[12] = s1 s0 ci' ai bi;
[13] = m' s1 s0 ai bi;
[14] = s0 ci' ai' bi';
[15] = m' s0 ai' bi';
[16] = s0' ci' ai bi';
[17] = m' s0' ai bi';
[18] = s1' s0 ci' ai';
[19] = m' s1' s0 ai';
[20] = s1' s0' ci' ai;
[21] = m' s1' s0' ai;
[22] = s1 ci bi;
[23] = s0 ai' bi;
[24] = s0' ai bi;
[25] = s0 ci ai';
```

[26] = s0' ci ai;
[27] = [4]' [5]' [6]' [7]' [8]' [9]' [10]' [11]' [12]' [13]' [14]' [15]' [16]'
[17]' [18]' [19]' [20]' [21]';
fi = [27]';
[29] = [22]' [23]' [24]' [25]' [26]';
co = [29]';
.endmodel alu.pla;

Before espresso optimization print_stats :

alu.pla pi= 6 po= 2 node= 2 latch= 0 lits(sop)= 154 lits(ff)= 47

After espresso optimization print_stats :

alu.pla pi= 6 po= 2 node= 27 latch= 0 lits(sop)= 124 lits(ff)= 124

b) Mapping using library nand-nor genlib:

Print stats before espresso

Total Area = 63.00
Gate Count = 36
Buffer Count = 0
Inverter Count = 10
Most Negative Slack = -16.00
Sum of Negative Slacks = -27.30
Number of Critical PO = 2

After espresso

Print statistics

Total Area = 145.00
Gate Count = 48
Buffer Count = 0
Inverter Count = 6
Most Negative Slack = -20.70
Sum of Negative Slacks = -35.30
Number of Critical PO = 2

So the value of the most negative slack is a lot higher after espresso. I am not quite sure why this happened or if the way I use SIS was faulty. From my understanding, to be more optimized, it will take less time and that means that the most negative slack should be lower. But based on the stats above, I got a higher negative slack after using espresso.

c) Check on the delay of output “co” and “fi” . I am not using espresso for this part.

{fi} : arrival=(11.30 11.30) required=(0.00 0.00) slack=(-11.30 -11.30)
{co} : arrival=(16.00 16.00) required=(0.00 0.00) slack=(-16.00 -16.00)

I read in alu.pla again and printed stats which returned this

alu.pla pi= 6 po= 2 node= 2 latch= 0 lits(sop)= 154 lits(ff)= 47

Checked print_map_stats and it was

Total Area = 63.00
Gate Count = 36
Buffer Count = 0
Inverter Count = 10
Most Negative Slack = -16.00
Sum of Negative Slacks = -27.30
Number of Critical PO = 2

Use speed_up and Buffer_opt

Speed up and then print stats:

alu.pla pi= 6 po= 2 node= 25 latch= 0 lits(sop)= 50 lits(ff)= 50

Buffer_opt

Total Area = 54.00
Gate Count = 29
Buffer Count = 0
Inverter Count = 7
Most Negative Slack = -12.00
Sum of Negative Slacks = -20.80
Number of Critical PO = 2

So the speed up was able to decrease the total area from 63 to 54. This means that the optimization is a lot better/faster since the total area is smaller.

Part 3: Comparison of PLA files

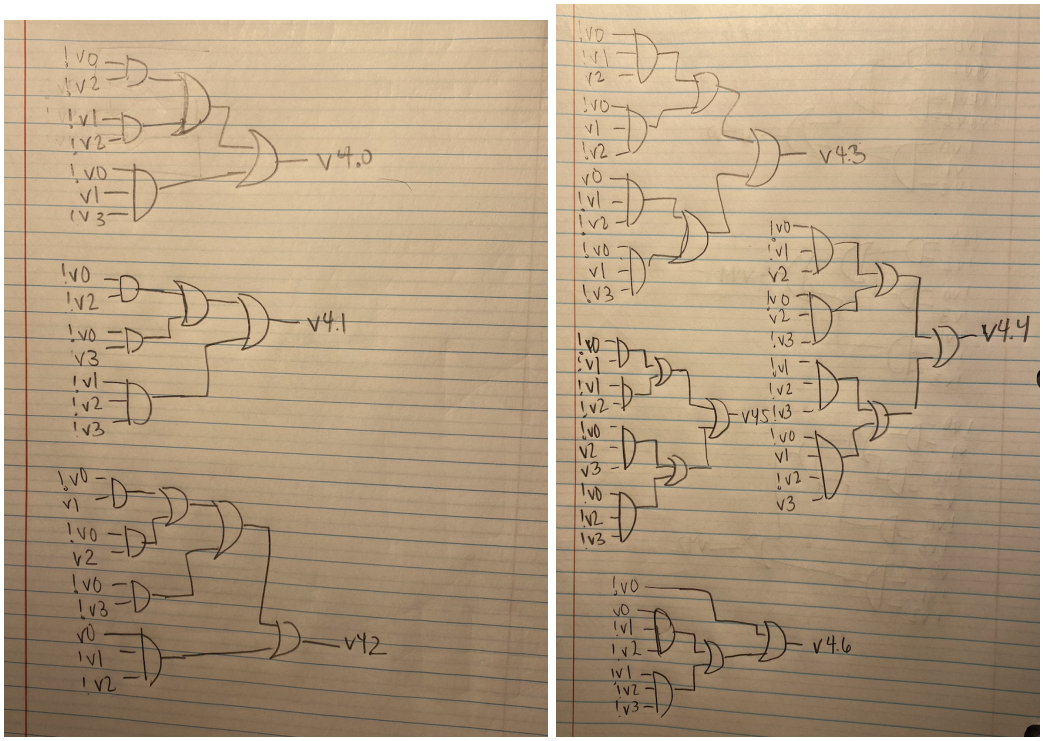
Pla file	Initial products	SIS products	My optimized products
pla1NEW	31	24	26
pla2NEW	576	76	85
pla3NEW	630	520	573
pla4NEW	1577	1436	2256
pla5NEW	78	82	78
pla6NEW	251	262	194
pla7NEW	1886	1522	3208
pla8NEW	833	764	775

Pla1 of SIS optimization:

Equation:

v4.0 = !v0!v2 + !v1!v2 + !v0v1!v3
v4.1 = !v0v2 + !v0v3 + !v1!v2!v3
v4.2 = !v0v1 + !v0v2 + !v0!v3 + v0!v1!v2
v4.3 = !v0!v1v2 + !v0v1!v2 + v0!v1!v2 + !v0v1!v3
v4.4 = !v0!v1v2 + !v0v2!v3 + !v1!v2!v3 + !v0v1!v2v3
v4.5 = !v0!v1 + !v1!v2 + !v0v2v3 + !v0!v2!v3
v4.6 = !v0 + !v1!v2

Schematics:



How many AND/OR gates will be needed? (Assuming 2-input gates)

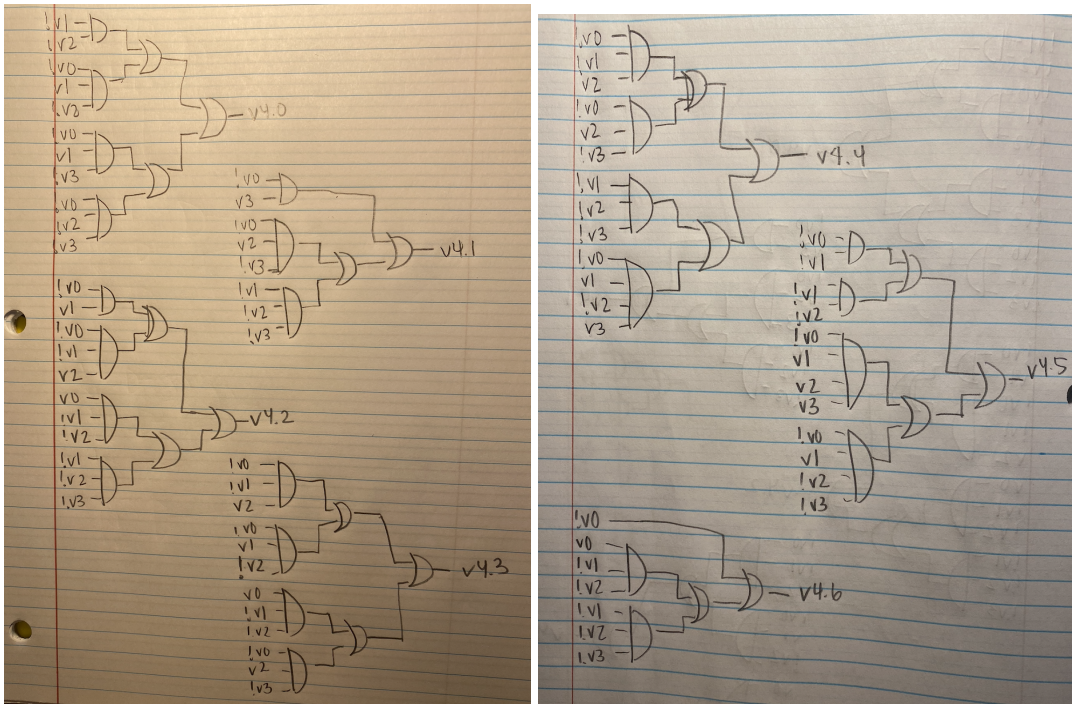
17 or gates + 37 and gates = 54 gates

Pla1 of my optimization:

Equation:

v4.0 = !v1!v2 + !v0v1!v2 + !v0v1!v3 + !v0!v2!v3
v4.1 = !v0v3 + !v0v2!v3 + !v1!v2!v3
v4.2 = !v0v1 + !v0!v1v2 + v0!v1!v2 + !v1!v2!v3
v4.3 = !v0!v1v2 + !v0v1!v2 + v0!v1!v2 + !v0v2!v3
v4.4 = !v0!v1v2 + !v0v2!v3 + !v1!v2!v3 + !v0v1!v2v3
v4.5 = !v0!v1 + !v1!v2 + !v0v1v2v3 + !v0v1!v2!v3
v4.6 = !v0 + v0!v1!v2 + !v1!v2!v3

Schematics:



How many AND/OR gates will be needed? (Assuming all 2-input gates)

19 (OR) + 48(AND) = 67 AND/OR gates.

The comparison between my optimization and the SIS optimization may have a few issues. I was not able to get it as optimized as the SIS but I was able to get it to be more optimized than the original. For pla5, I did not see any changes at all between my optimization and the original. I believe my optimization may think that it is already fully optimized. As for pla4 and pla7, I'm not quite sure why, but it seems like it has more products than the original. I believe the issue comes from pla4 and pla7 is reading in dont cares or onsets that are not in its output list but I do not know why that is happening when all other tests are reading just fine and I double checked that all lists should have the right inputs for those tests and they did. I tried debugging the issue but I still could not figure out the problem. But for all other tests, I believe that they are more optimized than the original but may be less optimized than the SIS.

Executables:

1. Make main
2. ./main filename e.g.(./main pla1NEW)
3. The outputs will be in the files label with the input name_output e.g.(pla1NEW_output)
    - The output has the written equation, initial products/literals, optimized products/literals, execution time.