# Project (1)

# Digital Design 1

# Quine-McCluskey

# Report

Dana Alkhouri

Ousswa Chouchane

Mhamad Khalil

**Dana Alkhouri**

This part includes the implementation of generating and printing prime implicants using C++ Language. In which, this part included another cpp file that gets from it the minterms and the number of literals.

The main data type is pair<set<int>, string> pi. This parameter takes a pair of set of integers which is the minterms and a string that is the binary representation of the minterms. Also, set<pi> pi\_group; a data type that is a collection of prime implicants when we group the minterms together in the different groping implementations.

In the first function, map<int, pair<string, int>> BinaryRepresentation(const vector<int> &minterms, int literals), it takes the minterms and number of literals then represent the minterms as binary representation so we can group the minterms based on the distance between them. For example, if we have (1) 001 and (3) 011, the distance between them is 1.

This is compared through the function bool compare\_distance(string a, string b)in which it compares two binary representation strings. If both strings have a distance of 1, then we can group their minterms; otherwise, it won’t group them.

To group them, we change the binary representation to get a common one, discard the different bit with a dash (-), and the function used to do that is string group(string a, string b). It modifies the binary string and then returns the modified string.

Continuing with the primary function in this part, which is pi\_group Generating\_Groups(vector<pi\_group> IGroup). This function starts with initializing a parameter of type vector<pi\_group> groupedElements, which will combine the groupedElements to use it again and to combine the current binary representation and a group of minterms. In the following code, it will group the minterms and then insert the minterms and binary representation in the prime implicants if it’s not repeated.

// grouping the variables but we need to store them somewhere

string groupedBinary= group(j.second,k.second);

auto grouped\_minterms = j.first;

grouped\_minterms.insert(k.first.begin(), k.first.end());

new\_group.insert({grouped\_minterms, groupedBinary});

// since the two elements are merged, we remove them from the prime implicants set

PrimeImplicants.erase(j);

PrimeImplicants.erase(k);

Another Part was implemented is the Kmaps which uses the Essential prime implicants to present a kmap and they’re represented as 1.

**Ousswa Chouchane**

## Functions

### `generateEPI`

This function takes a `pi\_group` of prime implicants and generates a map of minterm occurrences. The map contains minterm IDs as keys and a pair of values as the associated value. The pair consists of the total occurrence count of the minterm and a set of indices of prime implicants that cover the minterm.

### `EPI\_generator`

The `EPI\_generator` function extracts essential prime implicants based on minterm occurrences. It takes the `pi\_group` of prime implicants and the map of minterm occurrences generated by `generateEPI`. It identifies minterms with a single occurrence and extracts the corresponding prime implicants. The extracted EPIs are returned as a new `pi\_group`.

### `epi\_printer`

This function takes a `pi\_group` of essential prime implicants and prints them to the console, along with their associated minterm IDs and binary representations.

### `nonEPI\_extractor`

The `nonEPI\_extractor` function extracts non-essential prime implicants from the original set of prime implicants. It takes the complete set of prime implicants (`allPI`) and the set of essential prime implicants (`epis`). Non-essential prime implicants are identified and returned as a new `pi\_group`.

### `uncoveredMinterms`

This function determines the set of uncovered minterms by comparing the minterms covered by non-essential prime implicants with those covered by essential prime implicants. It returns a set of minterms that are not covered by the essential prime implicants.

### `transformToBoolian` and `transformToBooleanVec`

These functions take a set of variables and a set of prime implicants as input and transform them into Boolean expressions. The first function, `transformToBoolian`, works with a set of prime implicants, while the second function, `transformToBooleanVec`, works with a vector of prime implicants.

- Input:

- `vars`: A set of characters representing the variables.

- `piVector`: A vector of prime implicants or a set of prime implicants.

- Output:

- A set of strings, each representing a Boolean term.

These functions iterate through the prime implicants, evaluate their binary representations, and generate corresponding Boolean terms. The output is a set of strings representing Boolean expressions for the provided prime implicants.

### `sortPrimeImplicants`

This function sorts prime implicants based on specific criteria:

- Priority is determined by the number of minterms covered.

- If two prime implicants have the same coverage, they are compared based on the total number of implicants they contain.

- If the total number of implicants is also the same, the comparison considers the number of 1's in their binary representations.

- Input:

- `nonEssential`: A group of non-essential prime implicants.

- `uncoveredMinterms`: A set of uncovered minterms.

- Output:

- A vector of pairs containing prime implicants and their coverage scores.

### `selectPrimeImplicants`

This function selects prime implicants one by one, removes covered minterms, and creates a vector of selected prime implicants. It continues this process until there are no uncovered minterms left.

- Input:

- `nonEssential`: A group of non-essential prime implicants.

- `uncoveredMinterms`: A set of uncovered minterms.

- Output:

- A vector of selected prime implicants.

This function utilizes `sortPrimeImplicants` to select prime implicants according to the specified conditions. The selected prime implicants are returned as a vector.

### . `finalBooleanFunction`

This function generates the optimized Boolean function based on essential prime implicants (EPI), non-essential prime implicants (non-EPI), and selected prime implicants.

- Input:

- `ePI`: A group of essential prime implicants.

- `nEPI`: A group of non-essential prime implicants.

- `neededPI`: A vector of selected prime implicants.

- `uncoveredMintermsSet`: A set of uncovered minterms.

- `vars`: A set of characters representing the variables.

- Output:

- An optimized Boolean expression.

The function generates an optimized Boolean expression by transforming EPI, non-EPI, and selected prime implicants into Boolean terms and combining them with logical OR operators.

This program facilitates the simplification and optimization of Boolean functions by minimizing the number of required terms while considering coverage and other criteria. It allows for a more efficient representation of Boolean expressions.

**### Example Usage**

**Here's an example of how you can use these functions:**

| // Define your set of prime implicants pi\_group primeImplicants;  // Generate a map of minterm occurrences map<**int**, pair<**int**, set<**int**>>> mintermOccurrences = generateEPI(primeImplicants);  // Extract essential prime implicants pi\_group epis = EPI\_generator(primeImplicants, mintermOccurrences);  // Print the essential prime implicants epi\_printer(epis);  // Extract non-essential prime implicants pi\_group nonEssentialPI = nonEPI\_extractor(primeImplicants, epis);  // Determine uncovered minterms set<**int**> uncovered = uncoveredMinterms(nonEssentialPI, epis);  // Transform prime implicants into Boolean terms set<string> booleanTerms = transformToBoolian(vars, primeImplicants); set<string>nepii = transformToBooleanVec(vars, neededPI);  //coverage chart of NEPI vector<pi> selectedPIs = selectPrimeImplicants(nonEssentialPrimeImplicants, uncoveredMintermsSet);  //final boolian expression cout << finalBoolianFunction(essentialPrimeImplicants, nonEssentialPrimeImplicants, selectedPIs, uncoveredMintermsSet, vars); |
| --- |

**Mhamad Khalil:**

Circuit.cpp file:

This part include 2 main points of the project which are:

* Read in (and validate) a Boolean function given as
* Print the truth table of the function as well as the canonical SoP and PoS.

**isNegated(int index, const string& str):**

This function checks if a character in the product term is negated, meaning it is followed by a single quote ('). It returns true if the character at the specified index is followed by a single quote and false otherwise.

**create\_script(vector<string> sop):**

This function takes a vector of SOP product terms (sop) and generates a JSON-like script that represents these terms in a format understandable by WaveDrom.

It constructs the script in a way that WaveDrom can visualize the logical connections between the variables and their negations using AND, OR, and NOT gates.

The script generation consists of constructing nested arrays and objects based on the SOP product terms.

Input.cpp

This C++ code appears to be part of a program for processing and validating Boolean logic expressions. It defines functions to handle Boolean expressions and determine their form (Sum of Products - SoP, or Product of Sums - PoS). Here's an explanation of the code:

**variables**

- This function extracts variables from the global string `f`.

- It removes spaces from the string using `remove\_if` and `erase`.

- It also removes double single quotes (`''`) from the string to handle negation (e.g., `'A''` is converted to `'A'`).

- The function then uses regular expressions to identify and extract variables from the expression, adding them to the `vars` set.

**SoP**

- This function checks if the expression is in Sum of Products (SoP) form.

- It uses a regular expression pattern to match expressions that are in SoP form.

- If the expression is in SoP form, it calls the `variables` function to populate the `vars` set and returns `true` (1). Otherwise, it returns `false` (0).

**PoS**

- This function checks if the expression is in Product of Sums (PoS) form.

- It uses a regular expression pattern to match expressions that are in PoS form.

- If the expression is in PoS form, it calls the `variables` function to populate the `vars` set and returns `true` (1). Otherwise, it returns `false` (0).

**checkValidity**

- This function determines the validity and form of the input expression.

- It first checks if the expression is in SoP form using the `SoP` function.

- If it's in SoP form, it returns "SoP" and prints a message.

- If it's not in SoP form, it checks if the expression is in PoS form using the `PoS` function.

- If it's in PoS form, it returns "PoS" and prints a message.

- If the expression doesn't match either form, it returns "0" and prints an "Invalid Input" message.

**int binToDecimal(string bin)**

- This function converts a binary string to its decimal equivalent.

- It iterates through the binary string from right to left and adds the corresponding powers of 2 to the decimal value when a '1' is encountered.

- The decimal value is returned.

**generateCanonical(vector<vector<bool>> table)**

- This function generates canonical SoP and PoS expressions from the truth table.

- It initializes empty strings `SoP` and `PoS` to store the canonical forms.

- It also maintains a vector `minterms` to store the minterms encountered during the generation process.

- The function iterates through all possible combinations of variable values (truth table rows).

- For rows with '1' in the output column, it constructs the SoP form and updates `minterms`.

- For rows with '0' in the output column, it constructs the PoS form.

- The final SoP and PoS expressions are stored in `SoP` and `PoS`, respectively.

- The minterms are printed to the console, followed by the canonical SoP and PoS forms.

**printTT(vector<vector<bool>> table)**

- This function prints the truth table, which includes variable columns and an output column ('f').

- It iterates through the truth table rows and columns and prints the values.

**fillSoP(bool flag)**

- This function fills in the values in the `values` map for the SoP form.

- It extracts individual terms from the expression and evaluates them against the truth table.

- The results are stored in the `values` map, including the 'f' values.

- The `flag` parameter is used to specify whether to negate the 'f' values, allowing for PoS generation.

**fillPoS()**

- This function converts the given expression into PoS form for evaluation.

- It first removes any parentheses and replaces '+' with ' characters.

- The expression is then passed to `fillSoP` with the `flag` set to 1, indicating that 'f' values should be negated.

**generateTT(string flag)**

- This function generates the truth table, given a flag indicating whether to evaluate in SoP or PoS form.

- It creates an empty truth table and initializes the values for each variable.

- The truth table rows are generated with all possible combinations of variable values.

- The 'f' values are determined based on the provided form (SoP or PoS).

- The truth table is printed using the `printTT` function.

- The canonical forms are generated using the `generateCanonical` function.