MINIMISASI LOGIC

LAPORAN TUGAS BESAR

Sebagai salah satu bagian dari Tugas Besar mata kuliah Pemecahan Masalah dengan C (EL2008) Kelas 1 pada Semester IV Tahun Akademik 2021/2022

oleh

Adro Anra Purnama	13220005
Surya Dharma	13220027
Fariz Iftikhar Falakh	13220029
Senggani Fatah Sedayu	13220035

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

BANDUNG

2022

Daftar Isi

Laporan Inti

A. Studi Pustaka

Minimisasi logika adalah suatu proses untuk mencari persamaan yang lebih sederhana dari suatu rangkaian logika dengan syarat tertentu. Tujuan dari minimisasi logika adalah mengurangi banyaknya gerbang logika atau sirkuit yang digunakan supaya proses lebih sederhana dan penyelesaian persamaan memakan lebih sedikit waktu.

Metode minimisasi logika yang digunakan oleh kelompok ini adalah metode tabular atau metode Quine-McCluskey. Metode ini dipilih karena pembuatan kodenya lebih mudah dibandingkan dengan metode Karnaugh Map yang lebih bergantung pada visual. Metode tabular juga lebih mudah digunakan dalam menyelesaikan masalah dengan variabel yang banyak dibandingkan dengan Karnaugh Map. Dalam minimisasi logika, metode tabular menggunakan sistem eliminasi di mana pengaplikasian algoritma dapat menggunakan rekursi tanpa mengubah fungsi yang telah ada. Dibandingkan dengan Karnaugh Map yang perlu mengubah letak urutan dari bentuk tabel tergantung dengan banyak variabel dan menjadi lebih kompleks setelah ada 4 variabel, metode tabular lebih mudah digunakan.

Langkah dalam menggunakan metode tabular adalah pertama susun minterm yang diberikan dari yang terkecil hingga yang terbesar dan buat grup berdasarkan jumlah bilangan satu yang ada dalam representasi binernya. Sehiangga, akan ada maksimal $n + 1$ grup jika ada n variabel Boolean dalam fungsi Boolean atau n bit dalam ekuivalen biner dari minterm

Kedua, bandingkan minterm yang ada pada suatu grup dengan minterm lain yang ada pada grup selanjutnya. Jika perbedaan antara minterm satu dengan yang lainnya hanya satu bit, maka gabungkan dua minterm tersebut menjadi sebuah term baru. Tempatkan simbol x atau $-$ di bit yang berbeda dan biarkan bit lainnya. Selanjutnya ulangi langkah kedua dengan term yang baru terbuat hingga kita mendapatkan semua *prime implicants*.

Ketiga, buat *prime implicant* table. Tabel tersebut terdiri dari kumpulan baris dan kolom. *Prime implicant*s disusun sebagai baris dari tabel dan minterm disusun sebagai kolom dari tabel. Tempatkan '1' di sel yang bisa direpresentasikan oleh *Prime implicant*s serta minterm pada sel tersebut.

Keempat, cari *prime implicant* penting dengan mengamati tiap kolom. Apabila minterm hanya tercakup oleh satu *prime implicant*, maka *prime implicant* tersebut penting. *Prime implicant* tersebut akan menjadi bagian dari penyederhanaan fungsi boolean.

Terakhir, Kurangi tabel *prime implicant* dengan menghilangkan baris dari tiap *prime implicant* penting dan kolom yang bersesuaian dengan minterm yang telah tercakup didalam *prime implicant* penting tersebut. Ulangi langkah keempat untuk mengurangi tabel *prime implicant*. Proses dihentikan ketika semua minterm dari fungsi boolean yang diberikan telah tercakup.

B. Spesifikasi Program

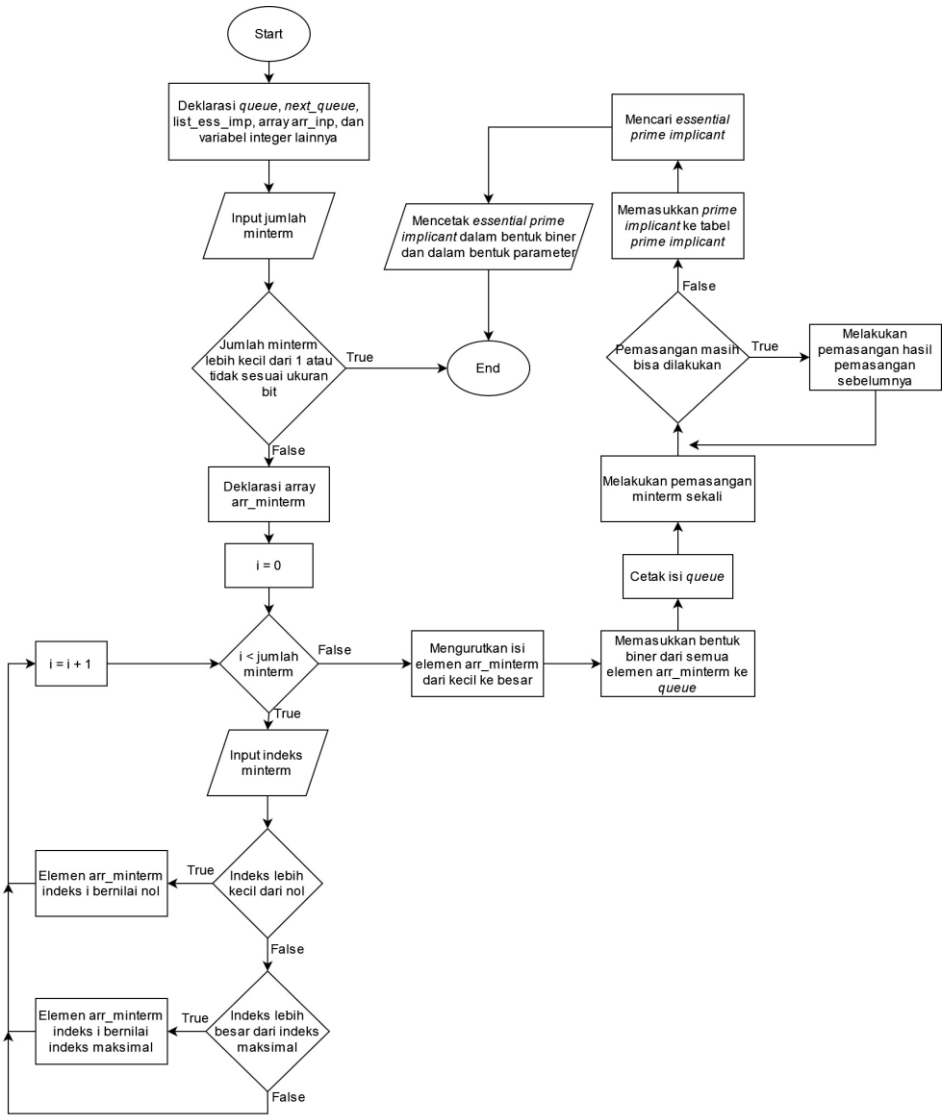
Spesifikasi program minimisasi adalah:

1. Program menerima input jumlah minterm dan indeks minterm
2. Program menampilkan hasil minimisasi berdasakran minterm yang diberikan
3. Program hanya bekerja untuk ukuran 2 – 8 bit
4. Ukuran bit hanya bisa diganti dengan mengganti definisi bitsSize pada *header* kode
5. Program tidak menerima minterm *don't care* dan menganggap tidak ada minterm *don't care*

C. Flowchart dan Data Flow Diagram

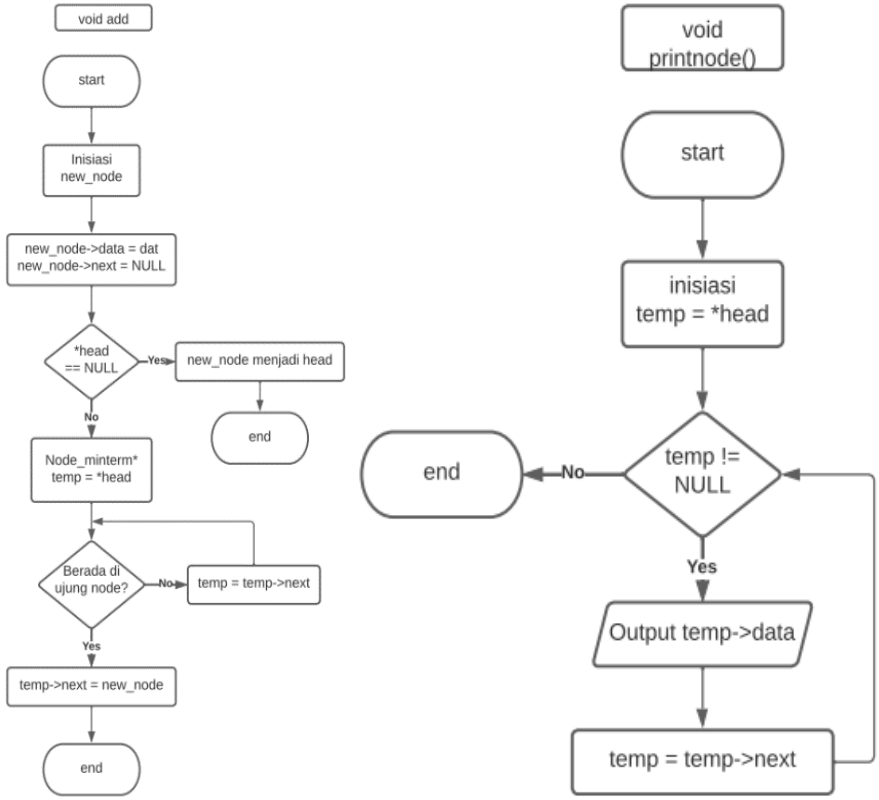
1) Flowchart

a. *Flowchart* algoritma

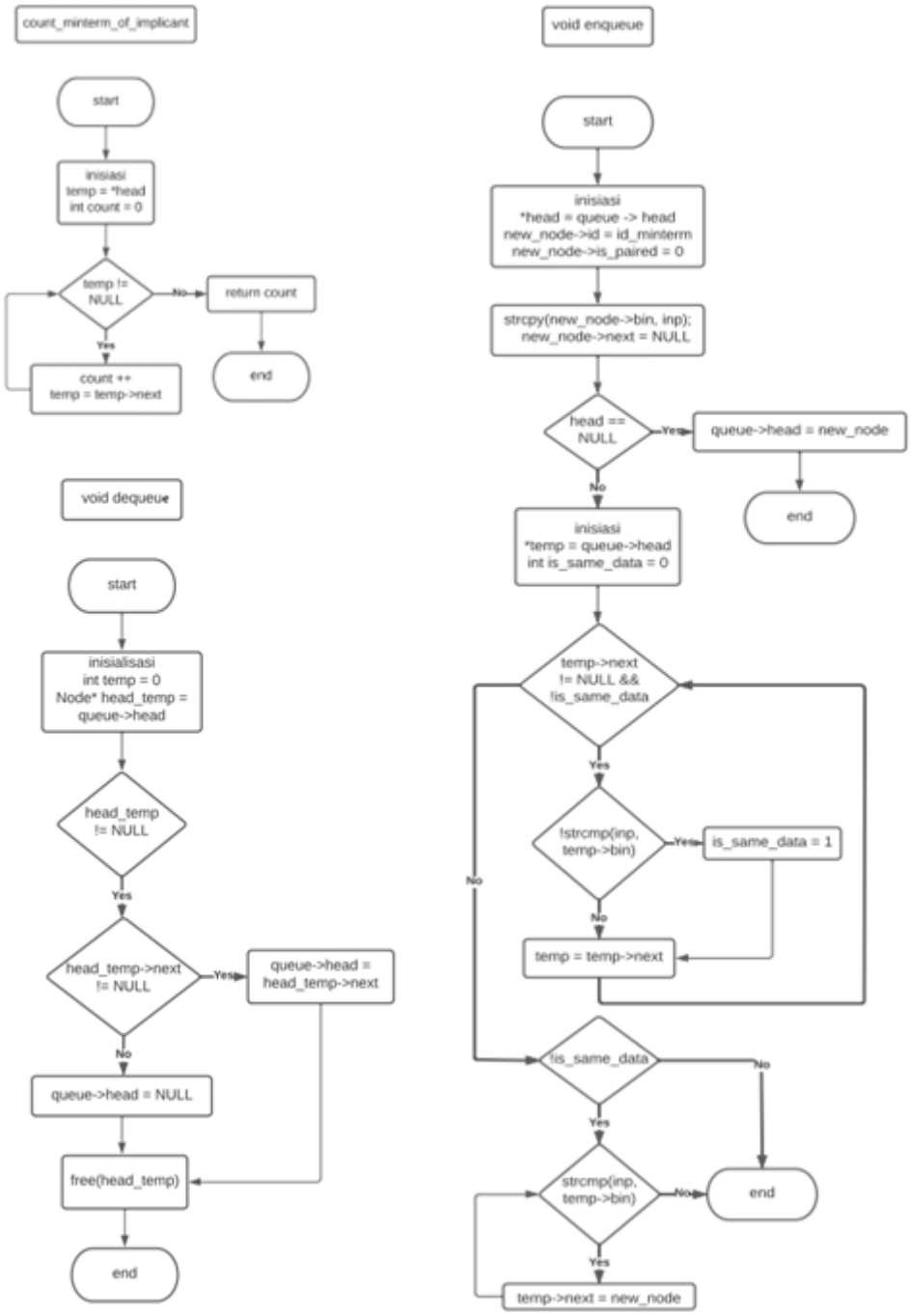


Gambar 1. *Flowchart* program secara keseluruhan

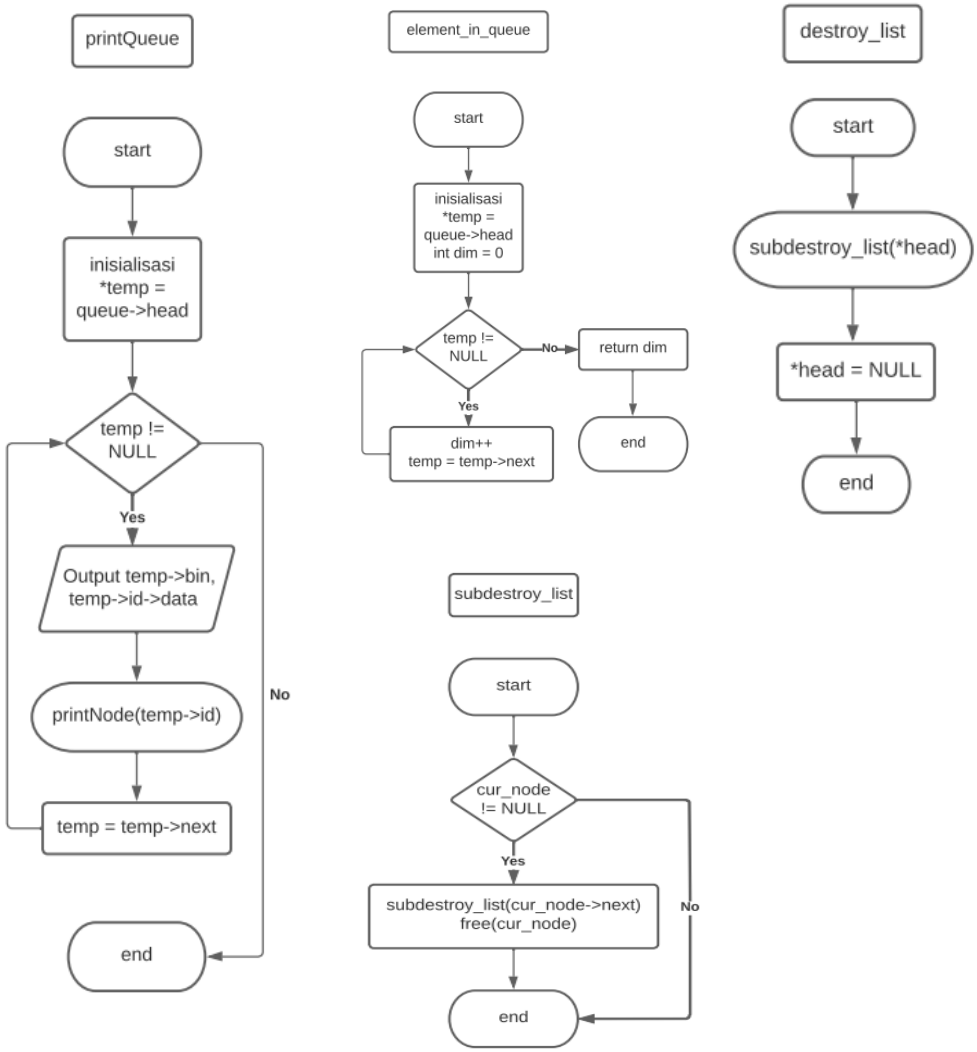
b. *Flowchart* fungsi dan prosedur yang digunakan



Gambar 2. Flowchart prosedur add dan prosedur printNode

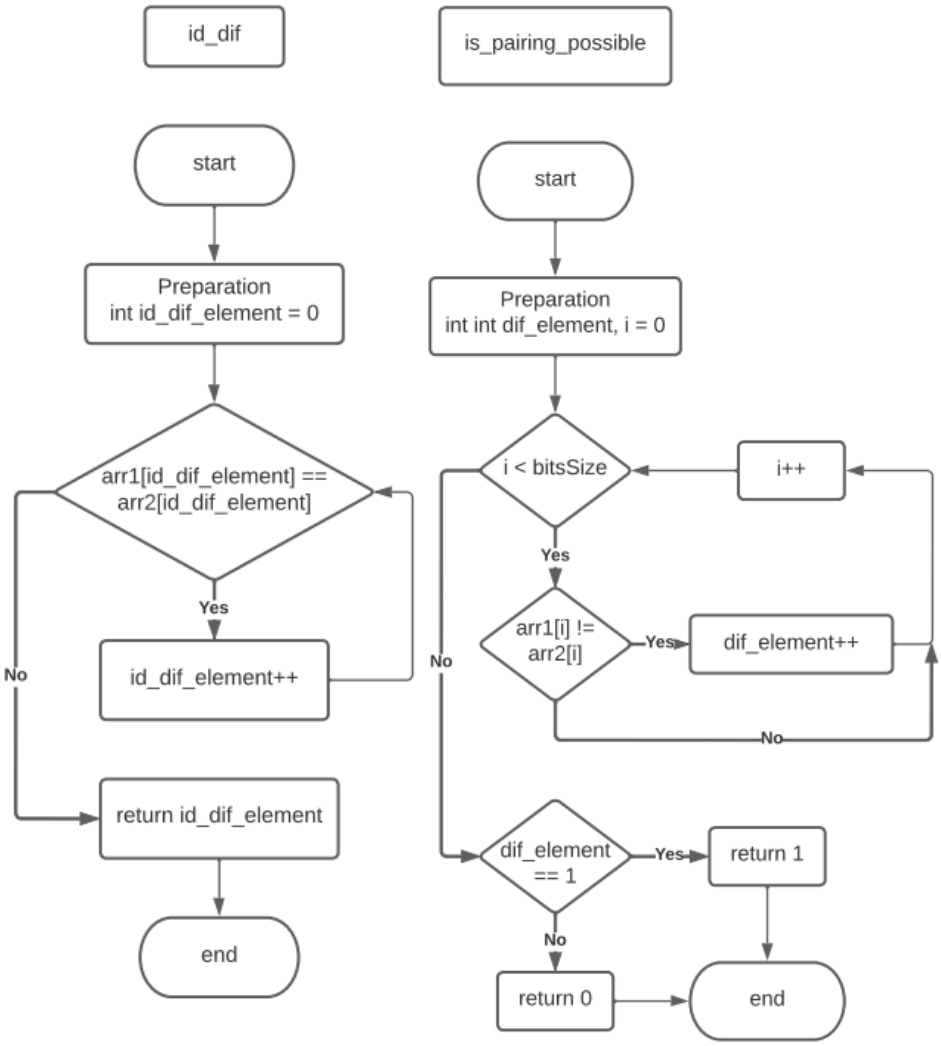


Gambar 3. Flowchart fungsi count_minterm_of_implicant, prosedur enqueue, dan prosedur dequeue

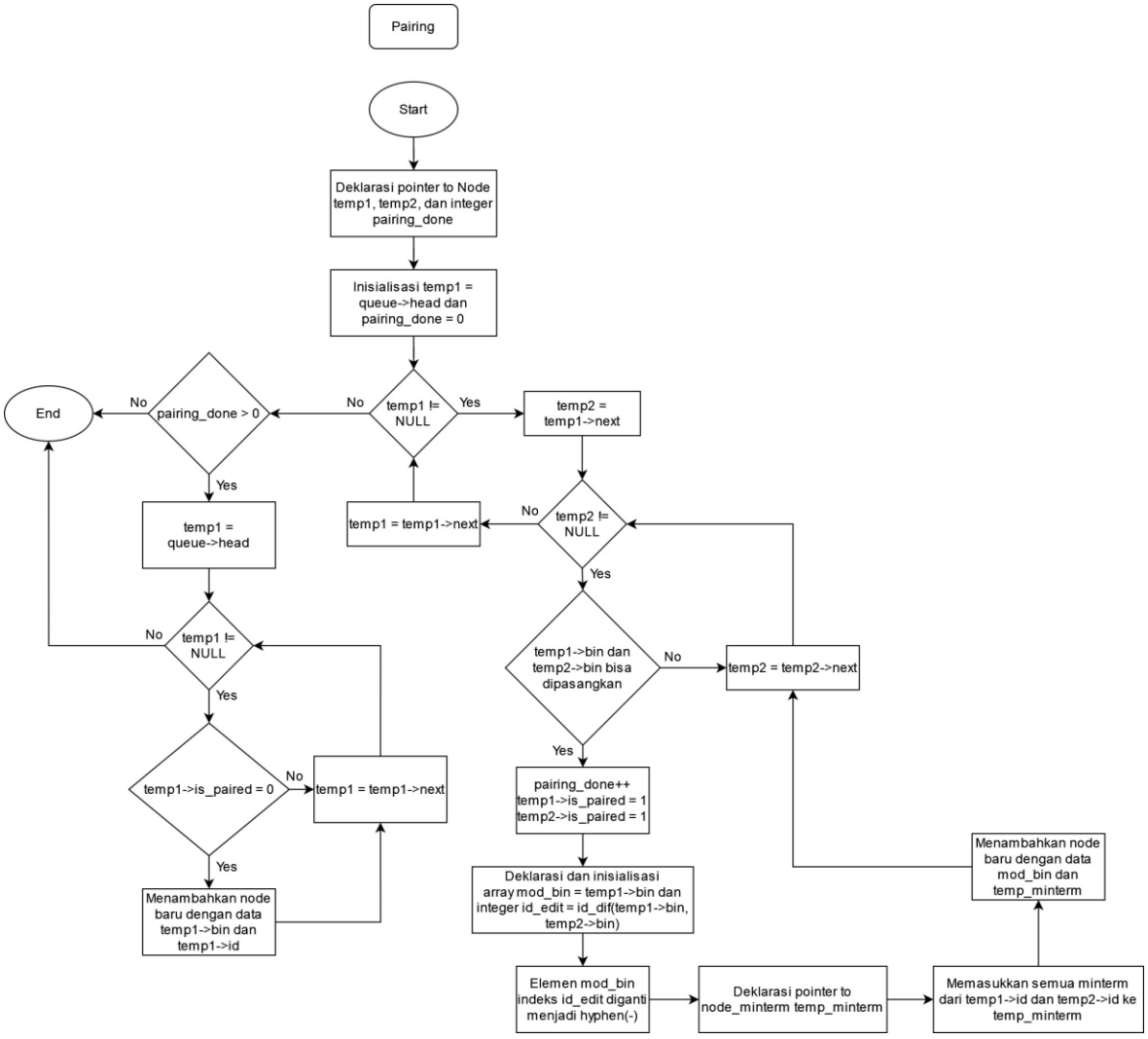


Gambar 4. Flowchart prosedur printQueue, fungsi element_in_queue, prosedur subdestroy_list, dan prosedur destroy_list

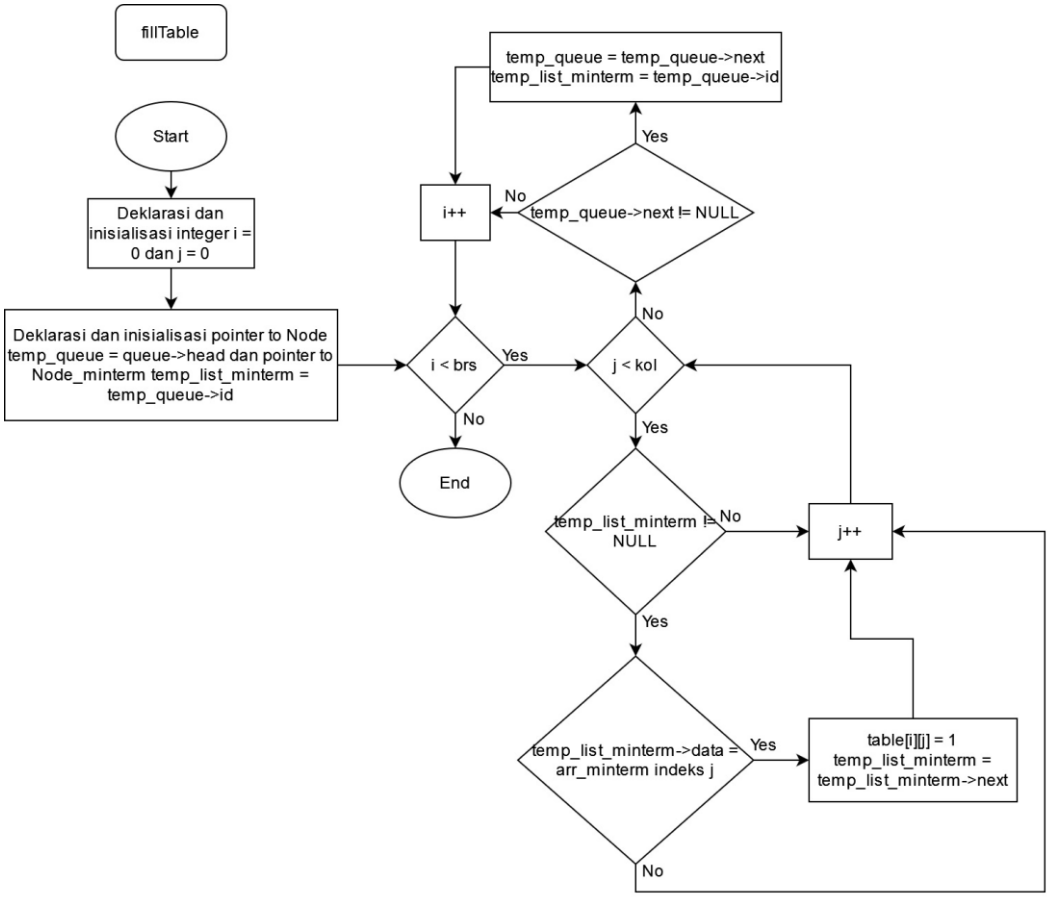
Gambar 5. *Flowchart* prosedur destroy_implicant_list dan prosedur dec2bin



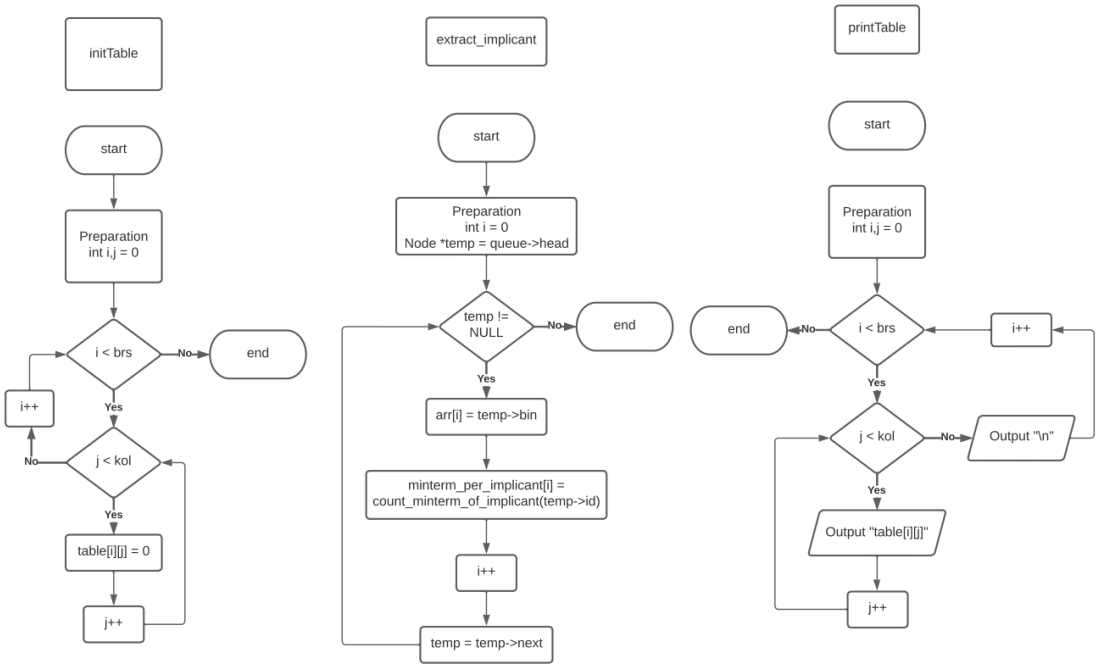
Gambar 6. Flowchart fungsi id_dif dan fungsi is_pairing_possible



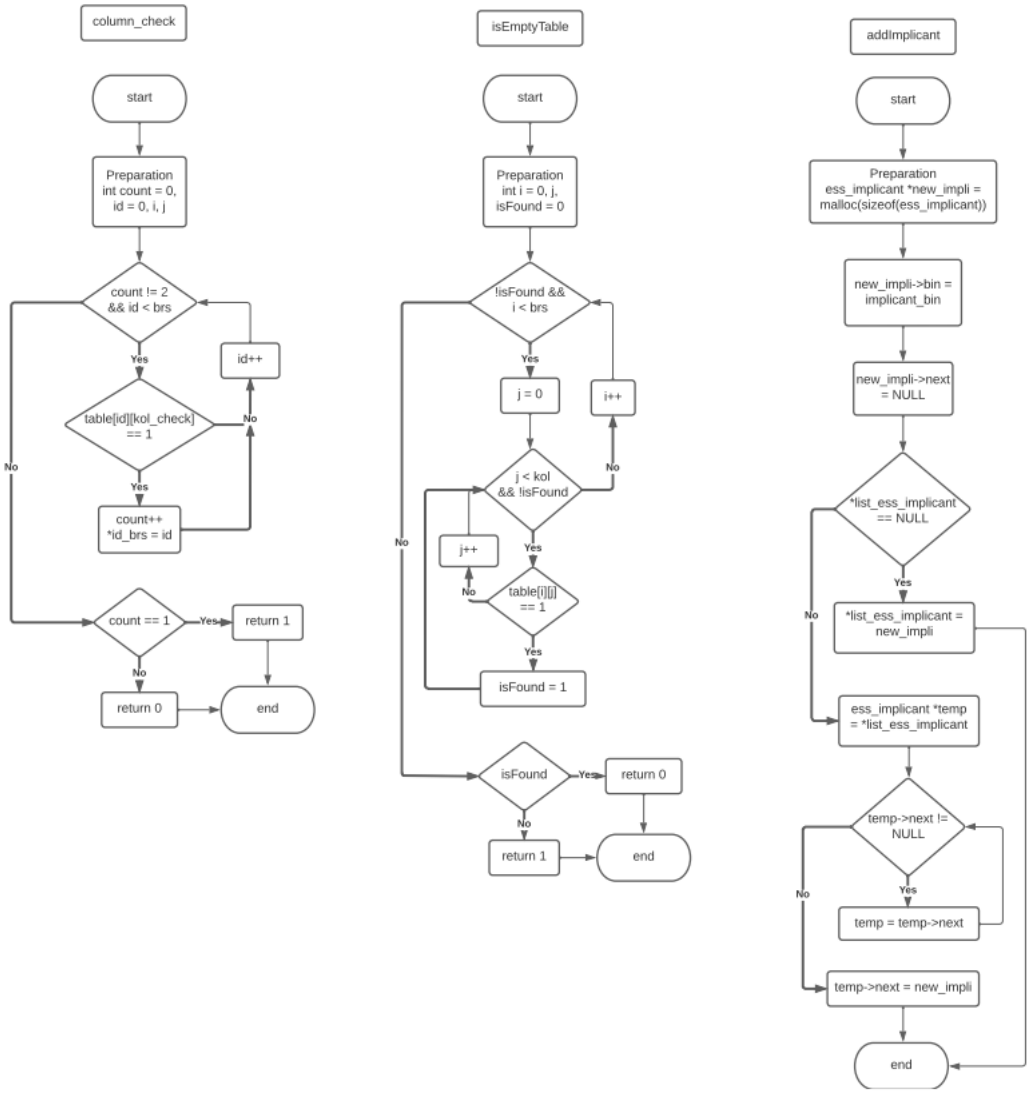
Gambar 7. *Flowchart* prosedur pairing



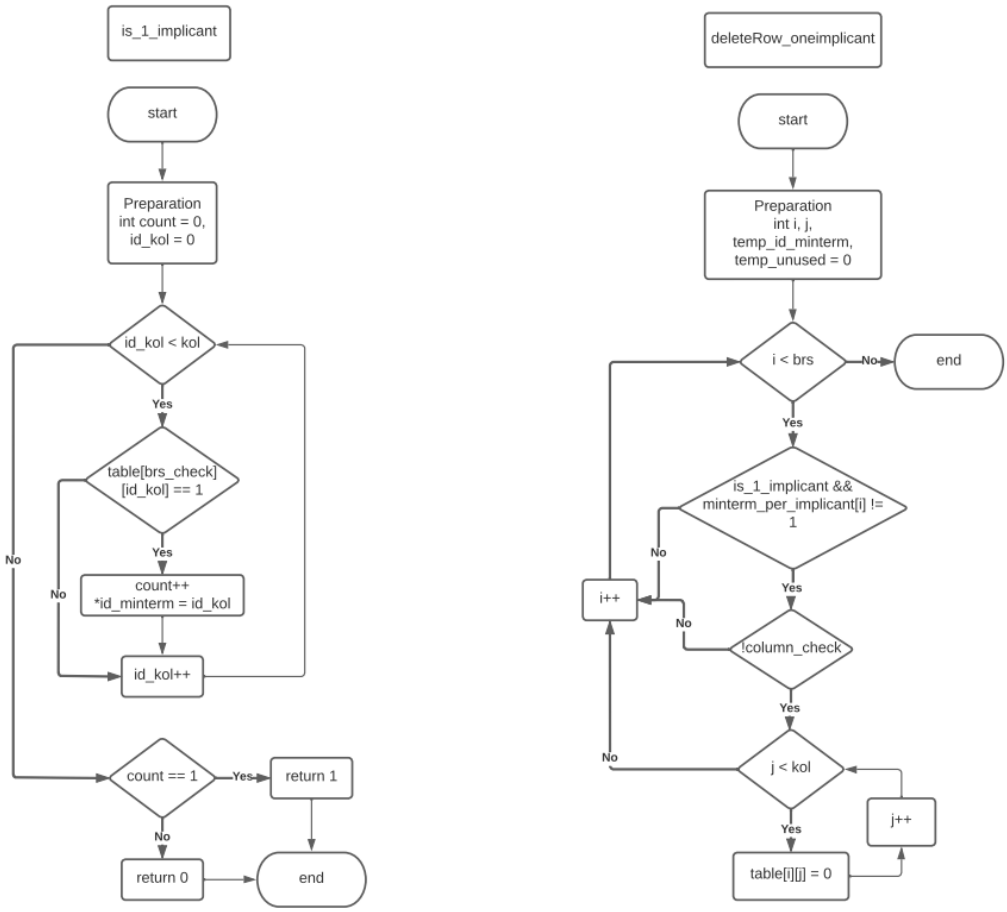
Gambar 8. Flowchart prosedur fillTable



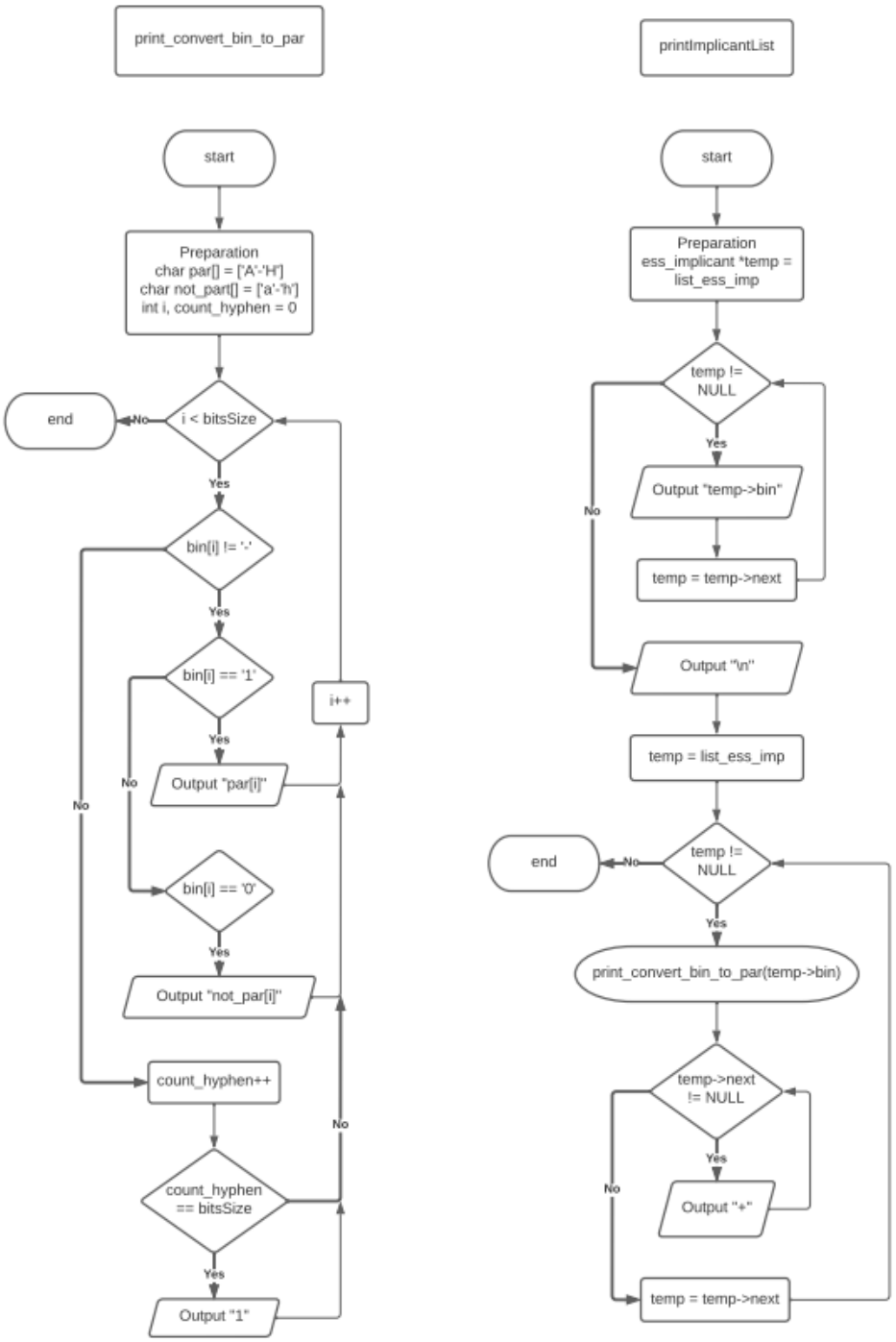
Gambar 9. Flowchart prosedur initTable, prosedur extract_implicant, dan prosedur printTable



Gambar 10. Flowchart fungsi column_check, fungsi isEmptyTable, dan prosedur addImplicant

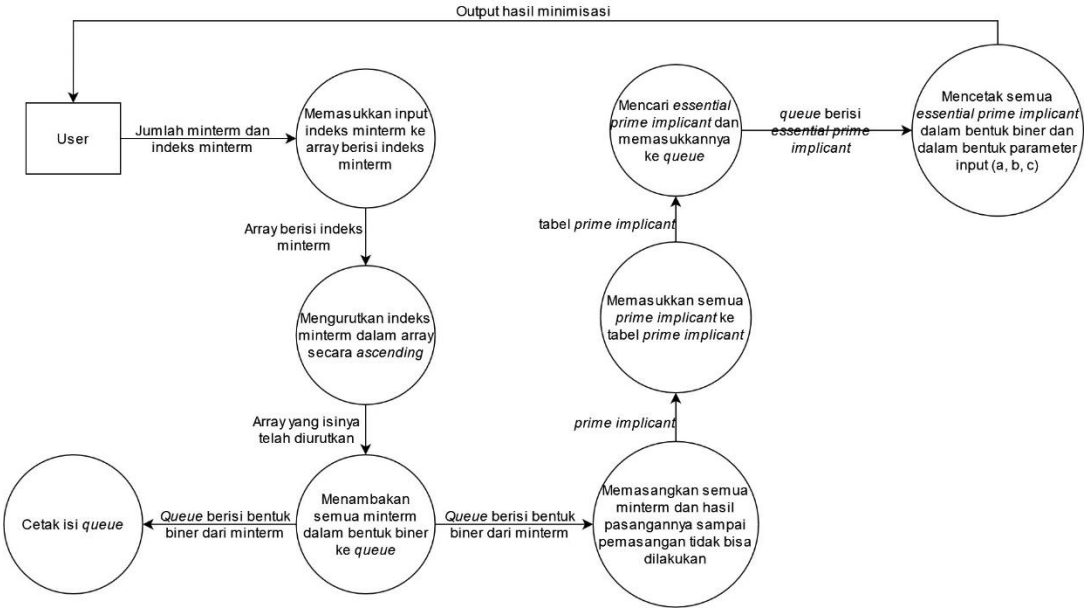


Gambar 11. Flowchart fungsi is_1_implicant dan prosedur deleteRow_oneimplicant



Gambar 12. Flowchart prosedur print_convert_bin_to_par dan prosedur printImplicantList

2) Data Flow Diagram



Gambar 13. Data Flow Diagram program minimisasi

D. Source Code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#define bitsSize 4

// Struct untuk implicant
typedef struct node
{
    struct minterm *id;
    char bin[bitsSize + 1];
    int is_paired;
    struct node *next;
}Node;

// Struct untuk jumlah minterm setiap implicant
typedef struct minterm
{
    int data;
    struct minterm *next;
}Node_minterm;

// Struct untuk prime implicant
typedef struct implicant
{
    char bin[bitsSize + 1];
    struct implicant *next;
}ess_implicant;

// Queue untuk menampung implicant
typedef struct Queue
{
    Node* head;
}Queue;

void add(Node_minterm **head, int dat)
// Prosedur menambah node implicant ke head
{
    Node_minterm* new_node = malloc(sizeof(Node_minterm));
    new_node->data = dat;
    new_node->next = NULL;
    if(*head == NULL){
        *head = new_node;
    }
    else {
        Node_minterm* temp = *head;
        while(temp->next != NULL){
            temp = temp->next;
        }
        temp->next = new_node;
    }
}

void printNode(Node_minterm *head)
// Prosedur mencetak seluruh minterm dari sebuah implicant
{
    Node_minterm *temp = head;
    while(temp != NULL){
```

```
        printf("%d ", temp->data);
        temp = temp->next;
    }
}

int count_minterm_of_implicant(Node_minterm *head)
// Fungsi yang menghitung jumlah minterm setiap implicant
{
    Node_minterm *temp = head;
    int count = 0;
    while(temp != NULL){
        count++;
        temp = temp->next;
    }
    return count;
}

void enqueue(Queue *queue, char inp[], Node_minterm *id_minterm)
// Prosedur menambah node baru dengan data (bentuk biner dari implicant dan
mintermnya) ke queue
{
    Node *head = queue->head;
    Node* new_node = malloc(sizeof(Node));
    new_node->id = id_minterm;
    new_node->is_paired = 0;
    strcpy(new_node->bin, inp);
    new_node->next = NULL;

    if(head == NULL){
        queue->head = new_node;
    }
    else {
        Node* temp = queue->head;
        int is_same_data = 0;
        while(temp->next != NULL && !is_same_data){
            if(!strcmp(inp, temp->bin)){
                is_same_data = 1;
            }
            temp = temp->next;
        }

        if(!is_same_data){
            if(strcmp(inp, temp->bin)){
                temp->next = new_node;
            }
        }
    }
}

void dequeue(Queue *queue)
// Prosedur menghapus node pada queue
{
    int temp = 0;
    Node* head_temp = queue->head;
    if(head_temp != NULL){
        if(head_temp->next != NULL){
            queue->head = head_temp->next;
        }
        else {
            queue->head = NULL;
        }
```

```
    }
    free(head_temp);
}

void printQueue(Queue *queue)
// Prosedur mencetak isi queue
{
    Node* temp = queue->head;
    while(temp != NULL){
        printf("%s ", temp->bin, temp->id->data);
        printNode(temp->id);
        temp = temp->next;
        printf("\n");
    }
    printf("\n");
}

int element_in_queue(Queue *queue)
// Fungsi yang mencetak jumlah node pada queue
{
    Node* temp = queue->head;
    int dim = 0;
    while(temp != NULL){
        dim++;
        temp = temp->next;
    }
    return dim;
}

void subdestroy_list(Node* cur_node)
// Fungsi untuk membebaskan semua node setelah head
{
    if(cur_node != NULL){
        subdestroy_list(cur_node->next);
        free(cur_node);
    }
}

void destroy_list(Node **head)
//Fungsi untuk membebaskan semua node selain head dan head itu sendiri
{
    subdestroy_list(*head);
    *head = NULL;
}

void destroy_implicant_list(ess_implicant *list_implicant)
{
    if(list_implicant != NULL){
        destroy_implicant_list(list_implicant->next);
        free(list_implicant);
    }
}

void dec2bin(int bil, char hasil[])
// Prosedur yang mengubah indeks minterm ke bentuk binernya
{
    for(int i = 0; i < bitsSize; ++i){
        if(bil >= pow(2, bitsSize - 1 - i)){
            hasil[i] = '1';
            bil -= pow(2, bitsSize - 1 - i);
        }
```

```
        else{
            hasil[i] = '0';
        }
    }
}

int isPairingPossible(char arr1[], char arr2[])
// Fungsi yang mengecek apakah arr1 bisa dipasangkan dengan arr2
{
    int dif_element = 0, i;
    for(i = 0; i < bitsSize; ++i){
        if(arr1[i] != arr2[i]){
            dif_element++;
        }
    }

    if(dif_element == 1){
        return 1;
    }
    else{
        return 0;
    }
}

int id_dif(char arr1[], char arr2[])
// Fungsi yang menghasilkan indeks dari elemen kedua array yang berbeda
{
    int id_dif_element = 0;
    while(arr1[id_dif_element] == arr2[id_dif_element]){
        id_dif_element++;
    }
    return id_dif_element;
}

void pairing(Queue *queue, Queue *next_queue)
// Prosedur memasangkan minterm
{
    Node *temp1 = queue->head;
    Node *temp2;
    int pairing_done = 0;

    while(temp1 != NULL){
        temp2 = temp1->next;
        while(temp2 != NULL){
            if(isPairingPossible(temp1->bin, temp2->bin)){
                pairing_done++;
                temp1->is_paired = 1;
                temp2->is_paired = 1;

                char mod_bin[bitsSize + 1];
                int id_edit = id_dif(temp1->bin, temp2->bin);

                strcpy(mod_bin, temp1->bin);
                mod_bin[id_edit] = '-';

                Node_minterm *temp_minterm = NULL;
                Node_minterm *id_temp1 = temp1->id;
                Node_minterm *id_temp2 = temp2->id;
                while(id_temp1 != NULL){
                    add(&temp_minterm, id_temp1->data);
                    id_temp1 = id_temp1->next;
```

```
                }
                while(id_temp2 != NULL){
                    add(&temp_minterm, id_temp2->data);
                    id_temp2 = id_temp2->next;
                }

                enqueue(next_queue, mod_bin, temp_minterm);
            }
            temp2 = temp2->next;
        }
        temp1 = temp1->next;
    }

    if(pairing_done > 0){
        // Kasus ada minterm yang belum dipasangkan
        temp1 = queue->head;
        while(temp1 != NULL){
            if(temp1->is_paired == 0){
                enqueue(next_queue, temp1->bin, temp1->id);
            }
            temp1 = temp1->next;
        }
    }
}

void initTable(int brs, int kol, int table[brs][kol])
// Prosedur untuk inisiasi tabel dengan nilai nol
{
    int i, j;
    for(i = 0; i < brs; ++i){
        for(j = 0; j < kol; ++j){
            table[i][j] = 0;
        }
    }
}

void extract_implicant(Queue *queue, char arr[][bitsSize + 1], int
minterm_per_implicant[])
// Prosedur memasukkan semua implicant ke arr dan memasukkan jumlah minterm
setiap implicant ke array minterm_per_implicant
{
    int i = 0;
    Node *temp = queue->head;
    while(temp != NULL){
        strcpy(arr[i], temp->bin);
        minterm_per_implicant[i] = count_minterm_of_implicant(temp->id);
        i++;
        temp = temp->next;
    }
}

void fillTable(Queue *queue, int arr_minterm[], int brs, int kol, int
table[brs][kol])
// Prosedur mengisi tabel essential prime implicant
{
    int i, j;
    Node *temp_queue = queue->head;
    Node_minterm *temp_list_minterm = temp_queue->id;
    for(i = 0; i < brs; ++i){
        for(j = 0; j < kol; ++j){
            if(temp_list_minterm != NULL){
```

```
                if(temp_list_minterm->data == arr_minterm[j]){
                    table[i][j] = 1;
                    temp_list_minterm = temp_list_minterm->next;
                }
            }
        }

        if(temp_queue->next != NULL){
            temp_queue = temp_queue->next;
            temp_list_minterm = temp_queue->id;
        }
    }
}

void printTable(int brs, int kol, int table[brs][kol])
// Prosedur mencetak tabel essential prime implicant
{
    int i, j;
    for(i = 0; i < brs; ++i){
        for(j = 0; j < kol; ++j){
            printf("%d ", table[i][j]);
        }
        printf("\n");
    }
}

int column_check(int brs, int kol, int kol_check, int table[brs][kol], int
*id_brs)
/* Fungsi untuk mengecek apakah suatu kolom hanya terdiri dari satu elemen
bernilai 1
Pointer id_brs digunakan untuk menyimpan indeks baris/prime implicant dari
elemen bernilai 1 tersebut */
{
    int count = 0, id = 0, i, j;

    while(count != 2 && id < brs){
        if(table[id][kol_check] == 1){
            count++;
            *id_brs = id;
        }
        id++;
    }

    if(count == 1){
        return 1;
    }
    else{
        return 0;
    }
}

int isEmptyTable(int brs, int kol, int table[brs][kol])
// Fungsi untuk mengecek apakah tabel kosong
{
    int i = 0, j, isFound = 0;

    while(!isFound && i < brs){
        j = 0;
        while(j < kol && !isFound){
            if(table[i][j] == 1){
                isFound = 1;
```

```
            }
            else{
                j++;
            }
        }
        i++;
    }

    if(isFound){
        return 0;
    }
    else{
        return 1;
    }
}

void addImplicant(ess_implicant **list_ess_implicant, char implicant_bin[])
// Prosedur menambah prime implicant ke node list_ess_implicant
{
    ess_implicant *new_impli = malloc(sizeof(ess_implicant));
    strcpy(new_impli->bin, implicant_bin);
    new_impli->next = NULL;

    if(*list_ess_implicant == NULL){
        *list_ess_implicant = new_impli;
    }
    else{
        ess_implicant *temp = *list_ess_implicant;
        while(temp->next != NULL){
            temp = temp->next;
        }
        temp->next = new_impli;
    }

}

int is_1_implicant(int brs, int kol, int brs_check, int table[brs][kol],
int *id_minterm)
/* Fungsi untuk mengecek apakah suatu baris memiliki satu elemen bernilai 1
Pointer id_minterm digunakan untuk menyimpan indeks minterm dari elemen
bernilai 1 tersebut */
{
    int count = 0, id_kol = 0;
    while(id_kol < kol){
        if(table[brs_check][id_kol] == 1){
            count++;
            *id_minterm = id_kol;
        }
        id_kol++;
    }

    if(count == 1){
        return 1;
    }
    else{
        return 0;
    }
}

void deleteRow_oneimplicant(int brs, int kol, int table[brs][kol], int
minterm_per_implicant[])
```

```
// Prosedur menghapus/mengosongkan baris pada tabel prime implicant
{
    int i, j;
    int temp_id_minterm = 0, temp_unused = 0;
    for(i = 0; i < brs; ++i){
        if(is_1_implicant(brs, kol, i, table, &temp_id_minterm) &&
minterm_per_implicant[i] != 1){
                if(!column_check(brs, kol, temp_id_minterm, table,
&temp_unused)){
                    for(j = 0; j < kol; ++j){
                        table[i][j] = 0;
                }
            }
        }
    }
}

void print_convert_bin_to_par(char bin[])
// Prosedur mencetak hasil minimisasi dalam bentuk parameter input
{
    char par[] = {'A', 'B', 'C', 'D', 'E', 'F' , 'G', 'H'};
    char not_par[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'};
    int i, count_hyphen = 0;

    for(i = 0; i < bitsSize; ++i){
        if(bin[i] != '-'){
            if(bin[i] == '1'){
                printf("%c", par[i]);
            }
            else if(bin[i] == '0'){
                printf("%c", not_par[i]);
            }
        }
        else{
            count_hyphen++;
            if(count_hyphen == bitsSize){
                printf("1");
            }
        }
    }

}

void printImplicantList(ess_implicant *list_ess_imp)
// Prosedur mencetak hasil minimisasi dalam bentuk biner dan diikuti oleh
bentuk parameter input
{
    ess_implicant *temp = list_ess_imp;
    while(temp != NULL){
        printf("%s ", temp->bin);
        temp = temp->next;
    }

    printf("\n");

    temp = list_ess_imp;
    while(temp != NULL){
        print_convert_bin_to_par(temp->bin);
        if(temp->next != NULL){
            printf(" + ");
        }
```

```
        temp = temp->next;
    }
}

int main(){
    // Deklarasi variabel
    Queue *queue = malloc(sizeof(queue));
    Queue *next_queue = malloc(sizeof(queue));
    ess_implicant *list_ess_imp = NULL;
    queue->head = NULL;
    next_queue->head = NULL;
    int jml_mint, id_mint, i, j, n_iter = 1, temp_sort;
    char arr_inp[bitsSize + 1];
    arr_inp[bitsSize] = '\0';

    // Menerima input jumlah minterm
    printf("Input : ");
    scanf("%d", &jml_mint);
    if(jml_mint < 1 || jml_mint > pow(2, bitsSize)){
        printf("Input jumlah minterm tidak valid!");
        free(queue);
        free(next_queue);
        exit(1);
    }
    int arr_minterm[jml_mint];

    // Menerima input indeks minterm
    for(i = 0; i < jml_mint; ++i){
        printf("Minterm ke : ");
        scanf("%d", &arr_minterm[i]);
        if(arr_minterm[i] < 0){
            printf("\n!Warning! Karena tidak ada minterm negatif, maka
dipilih minterm 0\n\n");
            arr_minterm[i] = 0;
        }
        else if(arr_minterm[i] > pow(2, bitsSize) - 1){
            printf("\n!Warning! Karena indeks minterm melebihi minterm
maksimal, maka dipilih minterm maksimal sesuai ukuran bit\n\n");
            arr_minterm[i] = pow(2, bitsSize) - 1;
        }
    }

    // Sort indeks minterm
    for(i = 0; i < jml_mint; ++i){
        for(j = 0; j < jml_mint - 1; ++j){
            if(arr_minterm[j] > arr_minterm[j + 1]){
                temp_sort = arr_minterm[j + 1];
                arr_minterm[j + 1] = arr_minterm[j];
                arr_minterm[j] = temp_sort;
            }
        }
    }

    // Konversi minterm ke biner dan masukkan ke queue
    for(i = 0; i < jml_mint; ++i){
        dec2bin(arr_minterm[i], arr_inp);
        Node_minterm *temp = NULL;
        add(&temp, arr_minterm[i]);
        enqueue(queue, arr_inp, temp);
    }
```

```
    // Cetak list minterm input
    printf("\nList minterm input :\n");
    printQueue(queue);

    // Melakukan algoritma Quine-McCluskey sekali
    pairing(queue, next_queue);

    // Melakukan algoritma Quine-McCluskey sampai tidak bisa dilakukan lagi
    while(next_queue->head != NULL){
        printf("Iterasi %d :\n", n_iter);
        n_iter++;
        printQueue(next_queue);
        queue->head = next_queue->head;
        next_queue->head = NULL;
        pairing(queue, next_queue);
    }

    // Deklarasi variabel untuk tabel essential prime implicant
    int n_imp = element_in_queue(queue);
    int EprimeImp[n_imp][jml_mint];
    char list_implicant[n_imp][bitsSize + 1];
    int minterm_hold_by_implicant[n_imp];

    // Inisiasi tabel
    initTable(n_imp, jml_mint, EprimeImp);

    // Memasukkan semua prime implicant ke list_implicant dan jumlah
minterm setiap implicant ke minterm_hold_by_implicant
    extract_implicant(queue, list_implicant, minterm_hold_by_implicant);

    // Mengisi tabel essential prime implicant
    fillTable(queue, arr_minterm, n_imp, jml_mint, EprimeImp);

    // Mencari essential prime implicant sampai tabel kosong
    while(!isEmptyTable(n_imp, jml_mint, EprimeImp)){
        id_mint = 0;
        int id_implicant;
        while(!column_check(n_imp, jml_mint, id_mint, EprimeImp,
&id_implicant) && id_mint < jml_mint){
            id_mint++;
        }
        if(id_mint != jml_mint){
            addImplicant(&list_ess_imp, list_implicant[id_implicant]);
            for(i = 0; i < jml_mint; ++i){
                if(EprimeImp[id_implicant][i] == 1){
                    for(j = 0; j < n_imp; ++j){
                        EprimeImp[j][i] = 0;
                    }
                    EprimeImp[id_implicant][i] = 0;
                }
            }
        }

        deleteRow_oneimplicant(n_imp, jml_mint, EprimeImp,
minterm_hold_by_implicant);
    }

    // Mencetak essential prime implicant
    printf("Hasil minimisasi (not A = a) : \n");
    printImplicantList(list_ess_imp);
```

```
    destroy_list(&(queue->head));
    destroy_list(&(next_queue->head));
    destroy_implicant_list(list_ess_imp);
    free(queue);
    free(next_queue);
    return 0;
}
```

Kesimpulan dan Lesson Learned

- Minimisasi merupakan sebuah proses penyederhanaan sebuah fungsi boolean yang sangat bermanfaat, karena dapat mengurangi *cost* serta kompleksitas dari sebuah rangkaian
- Quine-McCluskey merupakan salah satu metode minimisasi rangkaian yang mudah untuk diimplementasikan ke dalam suatu program, karena di dalamnya menggunakan sistem eliminasi yang berulang, sehingga dapat diselesaikan secara rekursif. Selain itu metode ini juga lebih efektif untuk menyederhanakan suatu rangkaian yang memiliki banyak variabel.
- Deklarasi array sebagai isi suatu *structure* dengan panjang yang bergantung dengan nilai variabel di luar *struct* tidak bisa dilakukan, kecuali dengan *header define*

Pembagian Tugas dalam Kelompok

Nama Anggota Kelompok	Kontribusi
Adro Anra Purnama	Membuat flowchart fungsi dan prosedur
Surya Dharma	Membuat *source code* program, *data flow diagram*, flowchart algoritma
Fariz Iftikhar Falakh	Melengkapi laporan, membuat file presentasi
Senggani Fatah Sedayu	Melengkapi laporan

Link Repository

https://github.com/DentAlpha/Minimisasi-Quine-McCluskey

Referensi

[1] Brown S. D. dan Vranesic Z. G, *Fundamentals of digital logic with VHDL design*, 2000.