Implementation of Quine-McCluskey

수업명 : 디지털논리신호

과제 이름 : Implementation of Quine McCluskey

담당 교수님 : 유지현 학번 : 2021202033

이름 : 윤성민

A. Problem statement (15)

콰인-맥클러스키 알고리즘은 부울식을 단순화하는 알고리즘이다. 카르노맵 알고리즘에 비해 조금 복잡하지만, 더 많은 변수를 다룰 수 있다.

구현은 크게 PI(Prime Implicant)를 찾고, 그 중에서 EPI(Essential PI)를 찾고, 구한 SOP의 트랜지스터 개수를 구하는 세 단계로 나뉘어진다.

첫 번째는 PI를 찾는 것인데, Implicant Table을 활용하여 구할 수 있다. 먼저 주어진 입력들을 1의 개수를 통해 분류하여 그룹짓는다. 그리고 각 그룹에서 해밍거리가 1만큼 차이나는 것이 있는지 체크하고, 만약 1만큼 차이난다면 해당 자리에 '로 표시하여 두 숫자를 하나로 묶는다. 이를 반복하였을 때 서로 묶이지 않는 것이 PI이다.

두 번째로 EPI를 찾기 위해서, PI들과 True인 minterm들을 각각 table의 행과 열의 헤더로 활용한다. PI가 커버할 수 있는 minterm들에 대해서 교점을 찍듯이 표시하면 EPI를 고를 준비가 된 것이다. 만약 PI가 유일한 열이 있다면, 다시 말해 해당 minterm을 표현할 PI가 유일하다면, 그 PI를 EPI로 선택하고 해당 PI로 표현할 수 있는 minterm들을 추가로 지운다. 이를 반복하면 EPI를 구할 수 있는데, 문제는 유일한 열이 없을 때 발생한다. 이를 해결하기 위해서는 부울 논리를 활용한 패드릭의 방법이 있고, 나는 그리디한 방법을 사용하였다.

세 번째로 구한 SOP에 필요한 트랜지스터를 구하는 것은 2단계의 논리회로라는 점과 inverter, AND, OR 세 가지 게이트만 활용한다는 제한이 있어 단순해졌다. 인버터는 2개가 필요한데, 각 변수별로 필요한지 아닌지 세면 된다. NAND와 NOR의 경우 CMOS를 통해 인풋의 두배만큼의 트랜지스터를 사용해 구현할 수 있으니 AND와 OR은 인버터를 추가하여 구현할 수 있다. 따라서 AND의 경우 대시로 표시하지 않은 1과 0의 값만큼 input이 필요한데, 인풋의 두배에 2를 더한 만큼 사용하여 구할 수 있다. OR의 경우 EPI의 개수가 인풋의 개수이다.

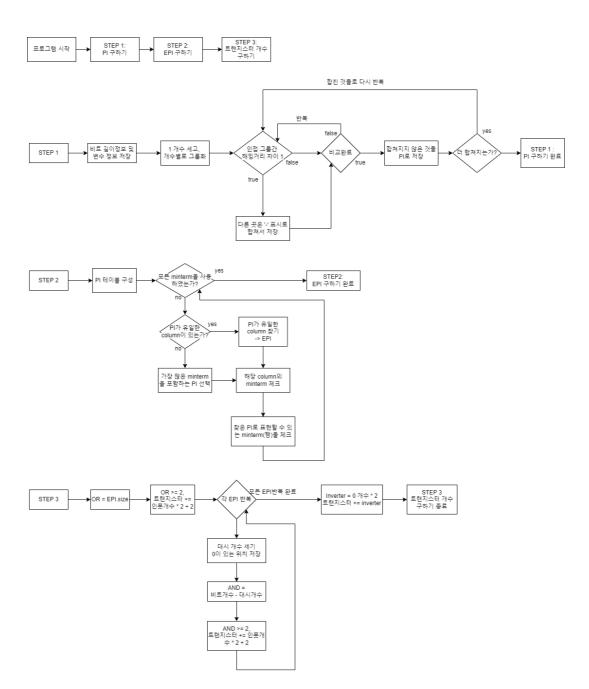
B. Your algorithm with pseudo code and flow chart (30)

```
class minTerm {
public :
 char value;
 string binary;
 bool isCmbnd;
int bincmp(string &bin1, string &bin2) {
 for (int i = 0; i < bin1.length(); i++) {
   if (bin1[i] != bin2[i]) {
     cnt++; // 다른 비트 개수
     pos = i; // 위치 저장
   }
 return (cnt == 1) ? pos : -1; // 다른 게 하나면 위치 반환, or -1 반환
bool binMatch(string &bin1, string &bin2) {
 for (int i = 0; i < bin1.length(); i++) {
  if (bin1[i] == '-' || bin2[i] == '-') continue; // '-'는 0과 1 돌다 포함, match
   if (bin1[i] != bin2[i]) return false; // 비트가 다르면 false
 return true;
}
int getNumTrans(set<string> &EPIs, int b_len) {
 int trans_num = 0; // 트랜지스터 개수
 int Or = EPIs.size(); // EPI의 개수가 OR의 input 개수
 if (0r > 1) // 인풋 개수는 항상 2보다 큼
   trans_num += (0r << 1) + 2; // (NOR(input * 2) + inverter(2) => OR)
 for (auto &k : EPIs) {
  int cnt_dash = 0; // '-' 대시 하나마다 AND input 하나씩 감소
   int pos_zero = 0; // 인버터를 위해 0이 있는 자리 계산
   for (int i = 0; i < b_len; i++) {
     if (k[i] == '-') cnt_dash++;
     if (k[i] == '0') pos_zero += (1 << i); // 0의 위치 정보를 2진법에서 10진법으로 전환
   And = b_len - cnt_dash;
   if (And > 1) // 인풋 개수는 항상 2보다 큼
     trans_num += (And << 1) + 2; // (NAND(input * 2) + inverter(2) => AND)
   Not |= pos_zero; // 0이 있는 위치들을 합집합연산(각 인덱스별로 0이 있는지 없는지)
 while (Not) {
   if (Not % 2 == 1) trans_num += 2; // 찾은 0 한개마다 인버터 하나씩(트랜지스터 두개씩)
   Not >>= 1;
```

```
return trans_num;
int main(int argc, char *argv[]) {
 int b_len; // 비트 길이
  fin >> b_len;
  vector<minTerm> MT[b_len+1]; // minterms
 vector<string> PIs; // Prime Implicants
 vector<string> trueMT; // true minterms
 minTerm init; int ones; // mt 벡터 초기화
 while (!fin.eof()) {
   fin >> init.value >> init.binary;
   if (init.value == 'm') trueMT.push_back(init.binary); // true minterm 저장
   ones = count(init.binary.begin(), init.binary.end(), '1'); // 1 개수 세기
   MT[ones].push_back(init); // 개수별로 정리
 while (2021202033) {
   bool brk = 1; // 루프 탈출 플래그. 더 이상 합칠 게 없으면 1로 유지. 합쳐지면 0.
    for (int i = 0; i < b_{len}; i++) {
      for (auto &k : MT[i]) \{
       for (auto &k2 : MT[i+1]) { // 해밍디스턴스 1인 그룹
         int pos;
         if ((pos = bincmp(k.binary, k2.binary)) != -1) { // 비트가 하나만 다를 때 위치(인덱스) 반환, or -1
           brk = 0; // 루프 한번 더
            k.isCmbnd = true; k2.isCmbnd = true; // 사용 표시
           string combined(k.binary);
combined[pos] = '-'; // 겹치는 부분 '-' 표시 -> 합치기
           uniq.emplace(combined); // 합친 표현 저장
       }
     }
    for (int i = 0; i < b_{len+1}; i++) {
     for (auto &k : MT[i]) {
  if (k.isCmbnd == false)
         PIs.push_back(k.binary); // PI 저장
     for (auto &k : uniq) // 저장해놓은 비트들
       if (count(k.begin(), k.end(), '1') == i) MT[i].push_back(k); // 새로 만든 것들 다시 1개수 순으로 그룹화하여 저장
    if (brk) break;
   b_len--;
  set<string> EPIs; // Essential PIs
  vector<int> table; // Essential PI를 찾기 위한 표. int형 정수를 통해 비트로 표현.(메모리, 성능)
  for (int i = 0; i < trueMT.size(); i++) { // 열을 먼저 확인하기 위함
   int col = 0;
   int cnt = 0; // 열에 PI가 얼마나 있는지 count
   int pos = -1; // PI가 하나일 때 EPI로 사용할 수 있게 위치 저장
   for (int j = 0; j < PIs.size(); j++) {
     if (binMatch(trueMT[i], PIs[j])) {
       col |= (1 << j); // 해당 비트에 1로 표시하기
       cnt++;
       pos = j;
   if (cnt == 1) { // 열에 PI 하나 -> Essential PI
     EPIs.emplace(PIs[pos]);
      trueMT.erase(trueMT.begin()+i--); // 해당 true minterm은 필요 없어짐.
      for (int t = 0; t < i; t++) {
       if (table[t] & (1 << pos)) {
   table.erase(table.begin() + t); // 방금 선택한 PI가 포함하는 범위. 해당 열 삭제
         trueMT.erase(trueMT.begin()+t--); i--;
     }
   else table.push_back(col);
  while(!trueMT.empty()) { // EPI 다 못찾은 경우
   int maxMt = 0; // 한 행당 minterm의 최대 개수
   int maxpos = -1; // minterm이 가장 많은 행의 위치
   if (trueMT.size() < 2) {
     if (trueMT.empty()) break; // 다 찾은 경우 for (int i = 0; i < PIs.size(); i++) { // 하나만 남은 경우
       if (table[0] & (1 << i)) {
         EPIs.emplace(PIs[i]);
         break;
```

```
for (int i = 0; i < PIs.size(); i++) { // maxMT, maxpos 27
    int cnt = 0;
for (int j = 0; j < trueMT.size(); j++) {</pre>
     if (table[j] & (1 << i)) cnt++; // MT가 있으면 카운트
    if (maxMt < cnt) {
      maxMt = cnt;
      maxpos = i;
  -
EPIs.emplace(PIs[maxpos]); // 찾은 행 EPI로 저장
  for (int i = 0; i < trueMT.size(); i++) { // 해당 PI가 커버하는 minterm들 삭제
   if (table[i] & (1 << maxpos)) {
     table.erase(table.begin() + i);
     trueMT.erase(trueMT.begin()+i--);
 }
  if (trueMT.size() < 2) {
    if (trueMT.empty()) break; // 다 찾은 경우
for (int i = 0; i < PIs.size(); i++) { // 하나만 남은 경우
if (table[0] & (1 << i)) {
        EPIs.emplace(PIs[i]);
        break;
     }
  if (trueMT.empty()) break; // minterm 개수가 최대인 행 접근으로 해결된 경우
  for (int i = 0; i < trueMT.size(); i++) { // 다시 한 번 유일 minterm이 있는 열 확인
    int cnt = 0;
    int pos = -1;
    for (int j = 0; j < PIs.size(); j++) {
  if (table[i] & (1 << j)) {
        cnt++;
        pos = i;
     }
    if (cnt == 1) {
      EPIs.emplace(PIs[pos]);
      trueMT.erase(trueMT.begin()+i--); // 해당 true minterm은 필요 없어짐.
      if (!trueMT.empty()) {
        for (int t = 0; t < trueMT.size(); t++) {
          if (table[t] & (1 << pos)) {
table[t] = 0; // 방금 선택한 PI가 포함하는 범위. 해당 열 삭제
            trueMT.erase(trueMT.begin()+t--); i--;
       }
     }
   }
 }
for (auto &k : EPIs)
    fout << k << '\n';
fout << \ "\nCost(\# of \ transistors): \ " << \ getNumTrans(EPIs, \ PIs[0].length());
fin.close();
fout.close()
```

슈도코드



플로우차트

프로그램 적용 화면

100비트, 1000 line으로 테스트한 결과

C. Verification stratagy & corresponding examples with explanation (50)

- PI 구하기
 - minterm 클래스를 만들어 값을 저장하고, merge 여부를 체크했다. merge되지 않은 것은 PI로 추가하였다. 예를 들어 0000
 과 0001을 비교한다면 서로 다른 비트가 하나기 때문에 isCmbnd 변수를 true로 저장해주고, 모든 경우의 수를 반복한 뒤 isCmbnd 변수가 false인 것을 PI로 추가하였다.
- EPI 구하기
 - 테이블을 인트형 배열로 사용하였고, 인트형 정수를 32비트 숫자로 생각하여 가로로 최대 32개의 너비를 가지는 2차원 배열을 구성하였다.
 - 행 by 행으로 입력받는 것보다 열 by 열로 입력받으면서, 동시에 해당 열의 PI의 cnt와 위치정보를 기록하였다. 그렇게 하여 유일한 PI가 발견되면 바로 EPI로 저장하고, 해당 minterm과 해당 PI가 포함하는, 그 이전까지 나온 것들을 삭제하였다.
 - 。 강의자료의 케이스를 활용해 예시를 들면 다음과 같다.

PI \ minterm	0100	0101	0110	1001	1010
PI1. 1010					
PI2. 0-00	х				
PI3. 1-01					
PI4. 01	х	х	x		
PI51-1		x			

0100과 0101의 경우 유일한 minterm을 가지고 있기 때문에 계속 입력받고, 0110에서 minterm이 유일한 것을 발견하였다.
 따라서 0110을 처리하고, 해당 PI로 표현할 수 있는 0101과 0100도 뒤로 돌아가 처리해준다.

PI \ minterm	0100	0101	0110	1001	1010
PI1. 1010					x
PI2. 0-00					
PI3. 1-01				x	
PI4. 01 V	V	V	V		
PI51-1					

- 。 그런 다음 1001과 1010을 입력받는데, 둘 모두 유일한 PI를 가지므로 체크한다. 따라서 PI1,3,4를 최종적으로 EPI로 선택하 게된다.
- EPI 구하기 PI가 유일한 열이 없을 때
 - PI가 유일한 열이 없다는 것은, 무엇을 골라도 성립하기는 한다는 것이다. 그러나 SOP의 2단계 논리회로를 구성해야 하므로, 트랜지스터의 개수를 줄이기 위해 가장 간단한 SOP를 찾아야 한다.

- 경우의 수가 많지 않으니 백트래킹으로 모든 경우의 수를 비교하려고 하다가, 속도 면에서 보너스 점수를 노리기 위해 그리디 한 접근 방법을 떠올렸다.
- 그 방법은 minterm을 가장 많이 포함하는 행(PI)을 선택하는 것이다. minterm을 포함하는 개수가 모든 PI에서 같다면 첫번째로 나온 것을 선택한다. minterm을 많이 포함하는 것은 '-'(대시)의 숫자가 많을 확률이 높고, 그렇다면 SOP에서 AND의 인풋 개수도 감소한다.
- 。 행을 통한 접근을 한 뒤엔, 유일한 minterm을 가진 열이 생겼는지 확인하여 열을 통해 접근할지 행을 통해 접근할지 결정한다.
- 아래에 제시할 TC를 통해 예시를 들어보면 다음과 같다. 해당 예시에서는 모든 PI의 대시 개수가 같으므로 대시 개수는 고려하지 않는다.

$$F(a,b,c) = \sum m(0,1,2,5,6,7)$$

PI \ minterm	0001	0010	0101	0111	1001	1111
PI1. 00-	x	x				
PI2. 0-0	x		x			
PI301		x		x		
PI410			x		x	
PI5. 1-1				x		x
PI6. 11-					x	х

- ∘ 해당 케이스의 PI를 구하여 table에 옮기면 위 표와 같이 되는데, 유일한 열이 하나도 없다. 그렇다고 차례대로 선택하게 되면 PI를 위에서부터 총 다섯 개 사용하게 된다.
- 위에서 언급한대로 행을 통해 접근한다면, PI1번에서 minterm의 개수가 최대가 된 뒤에 유지가 되므로, PI1번을 선택한다. 그런 다음 PI1이 포함하는 minterm 0001, 0010을 소거한다. 그럼 다음과 같은 상태가 될 것이다.

PI \ minterm	0001	0010	0101	0111	1001	1111
PI1. 00- V	V	V				
PI2. 0-0			x			
PI301				x		
PI410			x		x	
PI5. 1-1				x		х
PI6. 11-					x	x

- 。 유일한 minterm의 열이 없기 때문에 다시 한 번 행을 통해 접근한다.
- 。 만약 순차적으로 접근했다면 여기서 PI2를 선택했겠지만, PI2는 현재 포함하는 minterm의 개수가 한 개이고, PI3도 마찬가지 이므로, PI4를 선택하게 된다. 선택하고 나면 다음과 같은 상태가 된다.

PI \ minterm	0001	0010	0101	0111	1001	1111
PI1. 00- V	V	V				
PI2. 0-0						
PI301				х		
PI410 V			V		V	
PI5. 1-1				x		х
PI6. 11-						x

。 이제 다음 번에는 minterm이 두개로 가장 많은 PI5를 EPI로 선택하게 되고 모든 minterm을 골랐으니 다음 단계로 넘어가게 된다.

• 트랜지스터 구하기

OR과 AND의 경우 인풋개수가 1이면 성립하지 않으므로 추가하지 않도록 하였다. 예를 들어 ---1는 D이므로 AND연산할 필요가 없다.

0이 있는 곳의 위치를 비트로 표시하였다. EPI 각각의 비트를 돌며 비트마스킹을 하여 0의 위치를 찾았고, 이를 Not 인자에 합집합 연산하면서 추가하였다. 예를 들어 00--, 0--1, 1--0이 있다면 0의 위치를 1101과 같이 표시하여 inverter가 총 세 개의 변수에 사용된다는 것을 표현한다.

D. A testcase that you think it is very hard to solve (5)

• multiple EPI case
3
m 000
m 001
m 010
m 101

m 110 m 111

결과 :

-01

0-0

11-

Cost(# of transistors): 32

• 큰 수(100비트, 1000line testcase)

100

```
m
d
m
```

```
d
d
m
m
```

```
m
d
m
```

```
d
m
d
d
d
```

```
m
d
```

```
m
d
```

```
m
d
d
```

```
d
m
```

```
d
```

```
m
m
```

```
m
m
m
d
```

m

```
m
```

```
m
m
d
```

```
d
```

```
m
d
m
```

```
m
m
```

```
d
d
```

```
m
d
m
```

결과 :

Cost(# of transistors): 72958