Sintese Lógica - otimização lógica implementando o algoritmo Quine-McCluskey

Arthur João Lourenço
Dept. de Informatica e Estatistica (INE), Universidade Federal de Santa Catarina (UFSC)
Florianópolis, Brasil
arthurjolo@gmail.com

I. Introdução

Este projeto foi realizado como trabalho prático para a matéria do PPGCC UFSC INE410133 - Electronic Design Automation. Visando aplicar os conhecimentos adquiridos durante as aulas, este trabalho teve como objetivo a implementação de uma ferramenta para otimização lógica, um dos passos de Síntese Lógica no fluxo de EDA. Para realizar a otimização, a ferramenta utiliza o algoritmo de Quine-McCluskey [1], que recebe como entrada uma soma de mintermos e, como saída, gera um and-inverter graph (AIG) [2], para ser utilizado no próximo passo da Síntese Lógica (mapeamento tecnológico).

II. Definição do Problema

A etapa de Síntese Lógica no fluxo de EDA tem como objetivo transformar uma descrição de circuito em register-transfer level (RTL) em gate-level ou transistor-level [3]. Esta etapa pode ser dividida em: otimização da função lógica, mapeamento tecnológico e otimização lógica baseada na tecnologia.

Na parte de otimização da lógica independente da tecnologia, a expressão lógica obtida pela descrição RTL é simplificada visando diminuir os componentes da netlist a ser gerada. A otimização lógica pode também ser dividida em: otimização *two-level* ou *multilevel*, onde *two-level logic* representa expressões lógicas compostas por and-or, nor-nor, nand-nand, entre outros. Já o *multilevel* pode ser representado como uma *network* de *two-level logic* [3].

Neste trabalho, focou-se na otimização *two-level*, utilizando o algoritmo Quine-McCluskey [1]. O algoritmo recebe como entrada a soma dos mintermos da expressão e os dontcares da lógica a ser simplificada. Como saída, gera uma SOP com a lógica simplificada. O algoritmo Quine-McCluskey possui dois passos essenciais:

- Gerar os implicantes primos.
- Selecionar o menor conjunto de implicantes primos que cobre todos os mintermos.

A geração dos implicantes primos se baseia em juntar exaustivamente dois cubos que possuem uma diferença em exatamente uma posição. A posição que possui a diferença pode ser desconsiderada no novo cubo. Por exemplo: c1 = [00-1] e c2 = [01-1] podem ser fundidos em um novo cubo c3 = [0–1]. Um cubo que não consegue juntar-se a nenhum outro e não é composto inteiramente por dontcares é considerado um implicante primo. Isso é feito até que nenhuma nova junção possa ser feita e, assim, todos os implicantes primos sejam encontrados. Os cubos iniciais são o conjunto de mintermos e dontcares [3].

Para a realização dessas junções, o Quine-McCluskey utiliza tabelas para armazenar os cubos. Primeiramente, os mintermos e dontcares são agrupados pelo número de 1s, gerando a tabela 0. A partir desta tabela, são feitas as junções mencionadas entre os cubos de grupos adjacentes, ou seja, cuja diferença no número de 1s seja 1, gerando a tabela 1. Como mencionado acima, as junções são feitas até que nenhuma nova junção possa ser realizada, ou seja, as tabelas são geradas até que nenhuma nova junção seja possível.

Após gerar todos os implicantes primos, será selecionado o conjunto de implicantes primos que cobre todos os mintermos. Os mintermos que são contemplados por apenas um implicante primo são chamados de mintermos essenciais, e o implicante primo que os contempla é chamado de implicante primo essencial. Estes obrigatoriamente devem fazer parte do resultado. Então, é preciso apenas selecionar os implicantes primos que cobrem os mintermos restantes.

Para se adequar às formas de representação de lógicas booleanas mais comuns na área de EDA, o resultado do algoritmo Quine-McCluskey é armazenado em um AIG. Isso possibilita que mais otimizações possam ser feitas e que o resultado possa ser enviado para uma ferramenta de mapeamento tecnológico.

O AIG é um grafo direcionado e acíclico, onde os vértices representam portas lógicas AND de duas entradas, e as arestas que conectam dois vértices representam uma conexão entre duas portas lógicas. As arestas também possuem uma marcação indicando se a aresta inverte o valor do sinal. Essa representação consegue expressar uma SOP de forma simples e de fácil manipulação para etapas futuras, além de ser uma representação muito utilizada na área de EDA.

III. Desenvolvimento

A ferramenta de otimização lógica, contendo o algoritmo Quine-McCluskey e a criação do AIG para a expressão lógica resultante, foi implementada utilizando a linguagem C++ e pode ser encontrada em: *https://github.com/arthurjolo/EDA/tree/main/trabalho_pratico*.

Fig. 1. Exemplo de arquivo de entrada para os mintermos

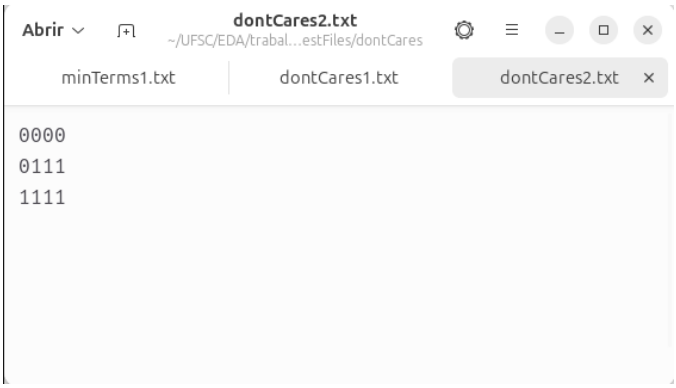

Fig. 2. Exemplo de arquivo de entrada para os dontcares

A ferramenta recebe como entrada dois arquivos plaintext. O primeiro contendo os mintermos da expressão lógica, onde cada linha contém um mintermo, ex.: Fig. 1. Já o segundo contém os dontcares, e também cada linha contém um dontcare, ex.: Fig. 2. Caso não existam dontcares, basta não passar nenhum arquivo. Tanto os mintermos quanto os dontcares devem ser representados em sua forma binária, e todos devem conter o mesmo número de bits. Estes arquivos são lidos pelo programa, armazenados em formato de string em um vetor, e é adquirido o número de entradas da expressão lógica a partir do tamanho dos mintermos e dontcares.

A. *Quine-McCluskey*

Para a implementação do Quine-McCluskey, foi implementada uma classe *QuineMcCluskey* (Fig. 3). Esta classe recebe como entrada os mintermos, os dontcares e o número de inputs da expressão lógica. A tabela do Quine-McCluskey (*std::map<int, std::vector<Implicant>> table_*) é representada por um mapa de inteiros para um vetor de cubos. Aqui, a chave é o número de 1s do cubo, e mapeia para um vetor contendo todos os cubos que possuem esse número de 1s. O cubo é representado pela classe *Implicant* (Fig. 4). A classe *Implicant* contém a expressão que representa o cubo e os mintermos que esse cubo contempla.

O principal método da classe *QuineMcCluskey* é *void createTable();*, que vai, a partir dos mintermos e dontcares,

```cpp
class QuineMcCluskey
{
    public:
    QuineMcCluskey(int nInputs,
                   std::vector<std::string> minTerms,
                   std::vector<std::string> dontCares) :
                     nInputs_(nInputs),
                     minTerms_(minTerms),
                     dontCares_(dontCares) {};
    ~QuineMcCluskey() = default;

    void run();
    Implicant bestImplicante();
    void createTable();
    void setDebuf(bool debug) { debug_ = debug; };
    std::vector<std::string> getRestuls() { return result_; };



    private:
    int nInputs_;
    bool debug_ = false;

    std::vector<std::string> minTerms_;
    std::vector<std::string> dontCares_;
    std::map<int, std::vector<Implicant>> table_;
    std::vector<Implicant> implicantePrimos_;
    std::map<int, std::vector<int>> minTerm2Implicante_;
    std::set<int> minTermContemplados_;
    std::vector<std::string> result_;
};
```

Fig. 3. Classe QuineMcCluskey

```cpp
class Implicant
{
    public:
    Implicant(std::string expression) : exp_(expression) {};
    ~Implicant() = default;

    std::string getExpression() {return exp_; };
    void setExpression(std::string expression) { exp_ = expression; };

    void addMinTerms(std::vector<int> minTerms) { minTerms_.insert(minTerms_.end(), minTerms.begin(), minTerms.end()); };
    void addMinTerm(int minTerm) { minTerms_.push_back(minTerm); };
    void removeMinTerm(int minTerm);
    std::vector<int> getMinTerms() {return minTerms_; }

    int getCoverage() { return minTerms_.size(); };

    private:
    std::string exp_;
    std::vector<int> minTerms_;
};
```

Fig. 4. Classe Implicant

criar a tabela 0, e iterativamente criar as próximas tabelas até que nenhuma junção de cubos possa ser feita. Durante a criação das tabelas, quando um cubo não realiza nenhuma junção, ele é um implicante primo. Quando eles são identificados, são armazenados no vetor de implicantes primos *std::vector<Implicant> implicantePrimos_;*.

Após gerar todos os implicantes primos, é feito o mapeamento de mintermos para os implicantes primos que os contemplam, para facilitar a seleção do conjunto resultado. Então, os mintermos que são contemplados por apenas um implicante primo são identificados e selecionados para o resultado (Fig. 5).

A seleção dos implicantes primos restantes é feita através de uma busca gulosa, baseada no número de novos mintermos que um implicante primo cobre. A cada iteração da busca, o implicante primo que cobre mais novos mintermos é selecionado, repetindo até que todos os mintermos sejam cobertos. Essa abordagem pode não encontrar a cobertura mínima, já que a busca gulosa tende a não encontrar o mínimo global, porém ela encontra mínimos locais com uma velocidade maior do que buscar todas as possibilidades de cobertura (Fig. 5).

```
for(auto el : minTerm2Implicante_) {
    if(el.second.size() == 1) {
        Implicant implicante = implicantePrimos_[el.second[0]];
        result_.push_back(implicante.getExpression());
        for(int minTerm : implicante.getMinTerms()) {
            minTermContemplados_.insert(minTerm);
        }
    }
}

for(int minTerm : minTermContemplados_) {
    for(int imp : minTerm2Implicante_[minTerm]) {
        implicantePrimos_[imp].removeMinTerm(minTerm);
    }
}

// busca gulosa pelo restante de implicantes para cobrir todos os mintermos
while (minTermContemplados_.size() != minTerms_.size()) {
    Implicant escolhido = bestImplicante();  // implicante que cobre maior número de novos mintermos

    result_.push_back(escolhido.getExpression());
    for(int minTerm : escolhido.getMinTerms()) {
        minTermContemplados_.insert(minTerm);
        for(int imp : minTerm2Implicante_[minTerm]) {
            implicantePrimos_[imp].removeMinTerm(minTerm);
        }
    }
}
```

Fig. 5. Seleção dos implicantes primos que envolvem todos os mintermos da função

O resultado do Quine-McCluskey, portanto, é uma lista de cubos, em formato de string, que minimizam os mintermos de entrada.

B. AIG

Com o resultado do Quine-McCluskey, um grafo AIG é construído. Neste trabalho, os vértices do grafo são representados pela classe *GraphNode*, como é mostrado na Figura 6, e possuem apenas um *id*, um nome e o valor booleano armazenado no nodo (utilizado para fazer o teste da expressão minimizada). As arestas são representadas pela classe *Edge*, que possui um destino e um peso, onde 0 indica que a aresta é negada e 1 que não é negada. O grafo é composto, então, por uma lista de nodos e um mapa de arestas, onde a chave é o índice do nodo fonte da aresta e mapeia para um vetor de *Edge*s.

A construção do grafo a partir da SOP minimizada segue os seguintes passos:

- Criar os nodos de Input e Output.
- Criar os nodos que fazem os ANDs da SOP.
- Criar os nodos que fazem o OR da SOP.

Para a criação dos nodos AND, basta utilizar os cubos e fazer as operações AND de 2 em 2 para os termos, negando as arestas quando o input é negado. Já para as operações OR, são utilizadas as saídas das operações AND e juntadas de 2 em 2 também. Porém, para representar a operação OR no AIG, é preciso inverter as entradas e saídas do nodo. Portanto, as arestas de entrada e saída dos nodos OR têm os pesos invertidos. Um exemplo de AIG construído para uma SOP é mostrado na Figura 7.

C. Verificação do resultado

Após a criação, o AIG é utilizado para testar se o resultado do Quine-McCluskey é equivalente à soma de mintermos de entrada. Para isso, todas as combinações de entradas são geradas e o AIG é percorrido para essas entradas, realizando as operações AND e inversão conforme o grafo. Assim, encontram-se todas as entradas que levam a expressão ao 1

```
struct Edge
{
    int endPoint;
    int wight;
};

class GraphNode
{
    public:
    GraphNode(int id, std::string name) : id_(id), name_(name) {};
    ~GraphNode() = default;
    std::string getName() { return name_; };

    void setCurrValue(int value) { currValue_ = value; };
    int getCurrValue() { return currValue_; };
    int getId() { return id_; };

    private:
    int id_;
    std::string name_;
    int currValue_;
};
```

Fig. 6. Estruturas do grafo

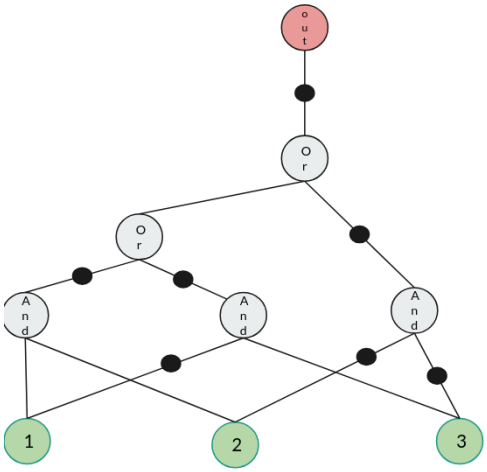

Fig. 7. Exemplo de um AIG para a SOP 11- + 0-1 + -00

lógico e verifica-se se essas entradas são as mesmas que estão contidas na soma de mintermos ou dontcares.

Caso as duas expressões sejam equivalentes, a ferramenta mostra uma mensagem de sucesso: *QuineMcCluskey logic is equivalent to the input*. Caso as lógicas não sejam equivalentes, a ferramenta mostra uma mensagem de erro: *ERROR: QuineMcCluskey logic is NOT equivalent to the input!*.

IV. Resultados

A. Testes

Para testar, foram utilizados 5 exemplos, apresentados na Tabela II. Os tamanhos das entradas testadas foram 4, 5, 6 e 14. Os exemplos foram tirados dos exercícios de aula, da página da Wikipédia [4] e de um exemplo disponível no GitHub da ferramenta ABC. Foi feito um Makefile para compilar e rodar todos os testes. Para compilar o código, basta rodar o comando *make*. Para rodar os testes, basta rodar o comando *make run_tests*.

TABLE I
RESULTADOS

Teste	Entrada	SOP de saida
test1	$\sum m = (2, 4, 6, 8, 910, 12, 13, 15)$	1-0- + 11-1 + 0-10 + -010 + 01-0
test2	$\sum m = (4, 5, 6, 8, 910, 13) + \sum d = (0, 7, 15)$	01– + 10-0 + 1-01
test3	$\sum m = (4, 5, 6, 8, 910, 13, 18, 22, 23, 24, 25, 26, 27, 30, 31)$	1-11- + –010 + 01-0- + 110– + 001-0 + 011-1
test4	$\sum m = (0, 4, 6, 12, 16, 17, 19, 26, 27, 28, 29, 30, 32, 37, 39, 40, 45, 46, 47, 50, 51, 52, 53, 54, 60, 61)$	0001-0 + -1110- + 10-1-1 + 10-000 + 10111- + 0-0000 + 0100-1 + 01101- + 11001- + 1101-0 + 00-100 + 011-10 + 1–101
test5	*olhar nos testcases*	0—0—0–0– + 0—0—10-1– + 0—0—-1-0– + 0—1—0-0— + 0—1—101— + 0—1—-10—

TABLE II
DESCRIÇÃO DOS TESTES

Teste	# entradas	# Mintermos + dontcares
test1	4	9
test2	4	10
test3	5	18
test4	6	26
test5	12	5632

TABLE III
RUNTIMES

Teste	runtime (s)
test1	0.000159
test2	0.000144
test3	0.000227
test4	0.000253
test5	670.546

Caso alguém deseje adicionar um teste novo, é preciso adicionar um arquivo contendo os mintermos da função na pasta do projeto *testFiles/minTerms* e um arquivo para os dontcares, caso existam, na pasta *testFiles/dontCares*. Por fim, adicionar um arquivo na pasta *testFiles* com o nome *testX.txt*. Neste arquivo, a primeira linha deve conter o path para o arquivo dos mintermos e a segunda linha o path para o arquivo dos dontcares, caso exista. Dessa forma, ao rodar *make run_tests*, o novo teste também será executado.

B. Escalabilidade e runtime

O algoritmo Quine-McCluskey é limitado pelo número de entradas da função minimizada, dado que o número de implicantes primos para uma função com n entradas pode chegar a $\frac{3^n}{\sqrt{n}}$ [4]. Portanto, a etapa de encontrar os primos implicantes pode ser bem custosa. A segunda etapa, de encontrar a cobertura mínima, também é custosa, dado que tenta resolver o problema da cobertura de conjuntos, que é NP-completo [5].

Podemos ver isso na prática ao analisar o runtime da ferramenta deste projeto rodando o algoritmo Quine-McCluskey, mostrado na Tabela III. Cada teste foi executado 10 vezes e o runtime foi definido como a média das 10 execuções. Podemos ver que o runtime cresce conforme o número de entradas e de mintermos mais dontcares aumenta. O maior teste, test5, que possui um pouco mais que o dobro de entradas que o segundo maior teste, teve um runtime aproximadamente 10^6 vezes maior. Porém, com um comportamento atípico, onde o test3 teve um runtime menor que o test4. Isso pode ser por dois fatores: no test3 foram necessárias menos junções para gerar todos os implicantes primos ou a busca gulosa pela cobertura dos mintermos foi mais rápida para o test3.

C. Qualidade dos resultados

Para todos os testes, a ferramenta gerou uma expressão lógica minimizada equivalente à expressão de entrada. A Tabela I mostra as entradas e saídas para os testes test1 ao test5, a entrada para o test 5 não é mostrada por ser muito grande.

Os resultados da ferramenta, apesar de equivalentes e menores que a entrada, não são o resultado ótimo para todos os casos. No test1, que foi visto em sala, o resultado encontrado pela ferramenta foi: $1-0-+11-1+0-10+-010+01-0$. Porém, como vimos em sala, é possível chegar em: $0-0-+-010+01-0+11-1$. Isso se dá pelo fato de que a busca gulosa pelo resultado não necessariamente encontra o ótimo global.

V. CONCLUSÃO

Neste trabalho foi desenvolvida uma ferramenta que utiliza o algoritmo Quine-McCluskey para realizar a minimização de expressões lógicas, armazenando o resultado em um AIG para se adequar às ferramentas mais comuns da área. A ferramenta gera resultados válidos e minimizados em relação à entrada.

A principal limitação deste projeto é o formato dos arquivos de entrada. Foi utilizado um formato criado apenas para este projeto, onde são necessários dois arquivos: um contendo os mintermos e outro contendo os dontcares. Outra limitação é que a ferramenta consegue otimizar apenas uma saída por vez, portanto, para cada saída é preciso fornecer os mintermos e dontcares e rodar a ferramenta uma vez por saída. Por fim, o resultado poderia ser mais otimizado caso não fosse utilizada uma busca gulosa pela cobertura dos mintermos, com o custo extra de runtime para utilizar o backtracking.

Como trabalho futuro, para aprimorar a ferramenta, seria adequado ajustar as entradas para formatos mais comuns na área, como arquivos .pla, adicionando também suporte para tratar múltiplas saídas para as mesmas entradas.

REFERENCES

[1] N. N. Biswas, "Minimization of boolean functions," *IEEE Transactions on Computers*, vol. 100, no. 8, pp. 925–929, 2006.

[2] A. Mishchenko, S. Chatterjee, and R. Brayton, "Dag-aware aig rewriting a fresh look at combinational logic synthesis," in *Proceedings of the 43rd Annual Design Automation Conference*, ser. DAC '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 532–535. [Online]. Available: https://doi.org/10.1145/1146909.1147048

[3] J.-H. R. Jiang and S. Devadas, "Chapter 6 - logic synthesis in a nutshell," in *Electronic Design Automation*, L.-T. Wang, Y.-W. Chang, and K.-T. T. Cheng, Eds. Boston: Morgan Kaufmann, 2009, pp. 299–404. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780123743640500138
[4] W. contributors. (2024) Quine-mccluskey algorithm. Accessed: 2025-07-14. [Online]. Available: https://en.wikipedia.org/wiki/Quine-McCluskey_algorithm
[5] ——. (2024) Set cover problem. Accessed: 2025-07-14. [Online]. Available: https://en.wikipedia.org/wiki/Set$_cover_problem$