Implementação em FPGA do Algoritmo de Quine-McCluskey para Minimização de Funções Lógicas

Fernando Frare Vieira

4 de julho de 2025

Resumo

Este relatório detalha o projeto e a implementação de um sistema digital que executa minimização de funções booleanas utilizando o algoritmo de Quine-McCluskey. O sistema permite ao usuário inserir uma tabela-verdade de quatro variáveis através de chaves e botões, processa os dados para encontrar a expressão booleana simplificada e exibe o resultado em um display de 7 segmentos. A arquitetura do projeto é descrita em VHDL, e possui módulos dedicados para entrada de dados, processamento do algoritmo e exibição dos resultados. Este documento serve como um guia completo para o entendimento, a operação e a replicação do projeto.

Sumário

1 Introdução

O objetivo do projeto é encontrar a expressão booleana mais simples para uma dada função de uma tabela verdade. O algoritmo de Quine-McCluskey é um método tabular exato para realizar essa minimização, garantindo a obtenção das formas mínimas da função.

Este projeto transporta a teoria do algoritmo de Quine-McCluskey para uma implementação prática e interativa em hardware, utilizando uma placa FPGA DE10-Lite. O sistema foi totalmente desenvolvido em VHDL e sintetizado utilizando o software Intel Quartus Prime.

O fluxo de operação do sistema é o seguinte:

1. **Entrada de Dados:** O usuário insere os 16 bits de uma tabela-verdade (para 4 variáveis: A, B, C, D) utilizando a chave `SW0` para o valor do bit e o botão `KEY0` para carregar cada bit sequencialmente.

2. **Processamento:** Ao acionar a chave `SW7`, o sistema inicia a execução do algoritmo de Quine-McCluskey sobre a tabela-verdade fornecida.

3. **Exibição do Resultado:** Ao final do cálculo, a expressão mínima é formatada e pode ser visualizada caractere por caractere no display de 7 segmentos `HEX0`, navegando através do botão `KEY1`. A negação de uma variável é indicada pelo acendimento do ponto decimal do display.

2 Fundamentação Teórica

Para compreender a motivação e o funcionamento deste projeto, é essencial revisar os conceitos de minimização de expressões booleanas e a metodologia do algoritmo de Quine-McCluskey.

2.1 Minimização de Expressões Booleanas

Uma função booleana pode ser representada de diversas formas, sendo a mais fundamental a tabela-verdade, que enumera a saída da função para cada combinação possível de suas variáveis de entrada. A partir da tabela-verdade, pode-se derivar uma expressão algébrica, como a Soma de Produtos (SOP), que consiste na soma lógica (OR) de mintermos — produtos lógicos (AND) que resultam em '1' para uma combinação específica das entradas.

Embora funcionalmente correta, a expressão canônica derivada diretamente da tabela-verdade é frequentemente ineficiente e redundante. A minimização é o processo de encontrar uma expressão booleana logicamente equivalente que utilize o menor número possível de termos. A importância da minimização é imensa no design de hardware, pois resulta em expressões mais simples que requerem menos portas lógicas.

Métodos como os Mapas de Karnaugh são eficazes para a minimização manual de funções com poucas variáveis (até 4 ou 5). No entanto, para um número maior de variáveis ou para a automação do processo, são necessários métodos algorítmicos, como o de Quine-McCluskey.

2.2 O Algoritmo de Quine-McCluskey

O algoritmo de Quine-McCluskey é um método tabular que garante a obtenção de uma expressão minimizada para qualquer função booleana. Ele é ideal para implementação

computacional por ser sistemático e não depender de reconhecimento de padrões visuais. O processo é dividido em duas fases principais:

2.2.1 Fase 1: Encontrar todos os Implicantes Primos

O objetivo desta fase é encontrar todos os implicantes primos, que são os termos candidatos para a expressão final. Um implicante primo é um termo produto que não pode ser mais simplificado sem deixar de cobrir as saídas '1' da função original.

1. **Listar Mintermos:** Inicialmente, todos os mintermos (combinações de entrada que resultam em saída '1') são listados em sua forma binária.

2. **Agrupar e Combinar:** Os mintermos são agrupados pelo número de '1's em sua representação binária. O algoritmo então compara cada termo de um grupo com todos os termos do grupo adjacente seguinte. Se dois termos diferem em apenas um bit, eles são combinados em um novo termo, substituindo o bit diferente por um "don't care"('-'). Os dois termos originais são marcados como "usados".

3. **Iterar:** Este processo de combinação é repetido com os novos termos gerados, criando termos cada vez maiores (com mais "don't cares"), até que nenhuma combinação adicional seja possível.

4. **Coletar Implicantes Primos:** Todos os termos que não foram marcados como "usados"durante o processo são os implicantes primos da função.

2.2.2 Fase 2: Selecionar a Cobertura Mínima

Nem todos os implicantes primos são necessários na expressão final. Esta fase seleciona o menor subconjunto de implicantes primos que "cobre"todos os mintermos originais.

1. **Construir o Gráfico de Implicantes Primos:** Uma tabela é criada onde as linhas representam os implicantes primos e as colunas representam os mintermos originais. Uma marcação (X) é feita na interseção se um implicante primo cobre um determinado mintermo.

2. **Selecionar Implicantes Primos Essenciais:** O algoritmo procura por colunas que contenham apenas um 'X'. O implicante primo correspondente à linha desse 'X' é *essencial*, pois é a única maneira de cobrir aquele mintermo. Todos os implicantes primos essenciais são adicionados à solução final.

3. **Cobrir Mintermos Restantes:** Após a seleção dos essenciais, se ainda houver mintermos descobertos, o algoritmo deve escolher entre os implicantes primos restantes. A implementação neste projeto seleciona repetidamente o implicante primo que cobre o maior número de mintermos ainda não cobertos, até que todos estejam satisfeitos.

O resultado final é a soma lógica (OR) de todos os implicantes primos selecionados, formando a expressão booleana mínima.

3 Arquitetura do Sistema

O projeto foi concebido com uma arquitetura modular, onde cada responsabilidade principal é encapsulada em um componente VHDL distinto. Isso facilita o desenvolvimento, o teste e a compreensão do sistema como um todo. A entidade de topo, `Top.vhd`, integra todos os módulos.

A Figura 1 ilustra a interconexão entre os componentes principais. Importante destacar que esse diagrama de blocos não foi usado na execução real do projeto, ele serve só para representar as conexões entre os módulos do sistema.

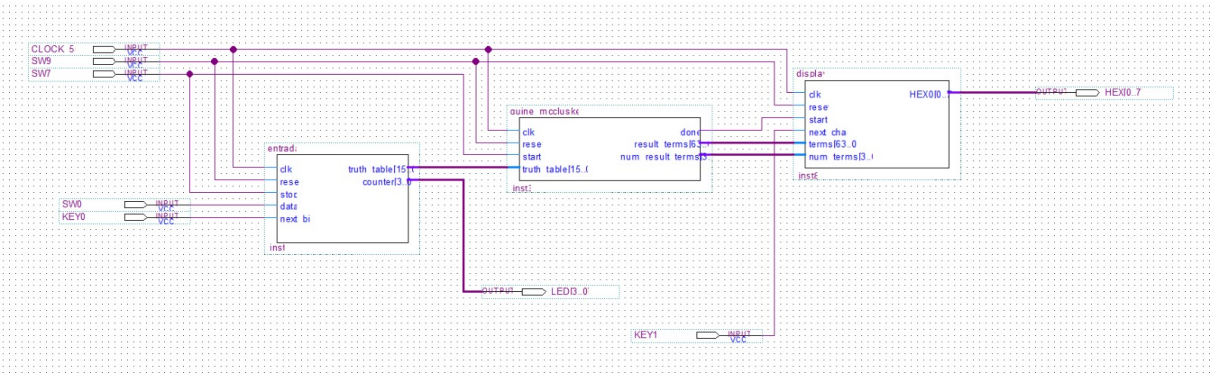

Figura 1: Diagrama de blocos da arquitetura do sistema.

4 Descrição dos Módulos VHDL

A seguir, cada um dos quatro principais arquivos VHDL do projeto é descrito em detalhe.

4.1 Top.vhd (Entidade Principal)

Este é o módulo de mais alto nível, responsável por conectar o hardware físico da placa DE10-Lite aos módulos lógicos internos. Ele instancia os outros três componentes e gerencia o fluxo de sinais entre eles. Suas principais funções são:

- Mapear as portas físicas (CLOCK_50, SW, KEY, LEDR, HEX0) para sinais internos.
- Gerar os sinais de controle globais, como o `reset` (a partir de `SW(9)`) e o sinal de início do algoritmo `s_start` (a partir de `SW(7)`).
- Controlar um sinal de "ocupado"(`s_real_busy`) para impedir a entrada de novos dados enquanto o algoritmo está em execução.
- Conectar a saída da tabela-verdade do módulo de entrada ao módulo Quine-McCluskey.
- Conectar a saída do resultado do módulo Quine-McCluskey ao módulo de display.

4.2 entrada.vhd (Módulo de Entrada)

Este componente gerencia a interface com o usuário para a inserção da tabela-verdade de 16 bits. As portas de entrada e saída foram nomeadas para refletir sua função direta:

- `clk`, `reset`: Sinais de controle síncrono.

- `stop`: Uma entrada que, quando em nível lógico '1', impede o registro de novos bits. É usada para travar a entrada enquanto o algoritmo principal está em execução.
- `data`: A entrada do bit de dado (0 ou 1) que será inserido na tabela-verdade.
- `next_bit`: Sinal vindo de um botão que, em sua borda de subida, comanda o registro do bit de `data`.
- `truth_table`: A saída de 16 bits que contém a tabela-verdade completa após a inserção.
- `counter`: Uma saída de 4 bits que indica o índice do bit que está sendo inserido (de 0 a 15), usado para feedback visual nos LEDs.

4.3 display.vhd (Módulo de Exibição)

Responsável por traduzir o resultado bruto do algoritmo em uma forma legível para o usuário no display de 7 segmentos. As portas de entrada e saída foram nomeadas para refletir sua função direta:

- `clk`, `reset`: Sinais de controle síncrono.
- `start`: Sinal que, quando ativado, inicia o processo de leitura e conversão do resultado.
- `terms`, `num_terms`: Entradas que recebem, respectivamente, o vetor de 64 bits com os termos da solução e o número total de termos.
- `next_char`: Sinal de um botão que permite ao usuário navegar pela sequência de caracteres da fórmula.
- `HEX0`: Saída de 8 bits que controla os 7 segmentos e o ponto decimal do display. A negação de uma variável é representada pelo acendimento do ponto decimal.

Assim como o módulo do algoritmo principal, este componente também opera como uma máquina de estados finitos (FSM) para gerenciar o processo de conversão e exibição do resultado. Os três estados da FSM, garantem que a tradução dos dados e a exibição ocorram na ordem correta.

`S_IDLE` Estado de espera (ocioso). O módulo permanece neste estado até que a entrada `start` seja ativada, sinalizando que o algoritmo de Quine-McCluskey terminou e os resultados estão prontos para serem processados.

`S_BUILDING` Estado de construção. Uma vez ativado pelo sinal `start`, o módulo entra neste estado para percorrer o vetor de resultado (`terms`). Ele decodifica cada termo, um char por vez, e constrói um array interno (`s_display_sequence`) com os caracteres da fórmula ('A', 'b', '+', etc.) e a informação de negação. Este processo continua, ciclo a ciclo, até que todos os termos tenham sido traduzidos para a sequência de exibição.

`S_DONE_BUILDING` Estado de conclusão. Ao terminar a construção da sequência, o módulo entra neste estado. Ele sinaliza para as outras partes do sistema que a fórmula está pronta para ser exibida pelo carrossel interativo. O sistema permanece aqui enquanto o usuário navega pelos caracteres usando o botão `next_char`. Ele só retorna ao estado `S_IDLE` quando o sinal de `start` principal for desativado, preparando o módulo para uma nova operação.

4.4 quine_mccluskey.vhd (Núcleo do Algoritmo)

Este é o componente mais complexo, implementando o algoritmo de Quine-McCluskey através de uma extensa máquina de estados finitos (FSM). Cada passo do algoritmo é cuidadosamente executado em um ou mais ciclos de clock. A seção seguinte detalha cada estado.

`IDLE` Estado inicial de espera. O sistema permanece aqui até que o sinal de entrada `start` seja ativado pelo usuário, iniciando o processo de minimização.

Fase 1: Encontrar os Implicantes Primos

`IE_INIT` **Inicialização da Extração Inicial.** Prepara o sistema para a primeira fase, zerando todos os contadores (como `num_pi`, `num_minterms`) e limpando as tabelas de armazenamento de implicantes (`pi_table`, `final_pi_list`, etc.).

`IE_LOOP` **Loop da Extração Inicial.** Este estado itera sobre os 16 bits da `truth_table` de entrada. A cada ciclo, ele verifica um bit. Se o bit for '1', o índice correspondente é registrado como um mintermo inicial (um implicante com 0 "don't cares") e adicionado à `pi_table`. Ao final dos 16 ciclos, o sistema avança para a próxima fase, a menos que a função seja trivial (toda '0' ou toda '1').

`C_PASS_INIT` **Inicialização do Passo de Combinação.** Prepara as variáveis para um passe de combinação. Zera os contadores de laço (`i`, `j`, `k`), limpa a tabela que armazenará os novos implicantes combinados (`next_pi_table`) e zera as flags de controle (`pi_used_flags`, `combination_made`).

`C_LOOP_I` **Loop Externo de Combinação.** Inicia o laço externo ('for i...'), que seleciona o primeiro implicante para a comparação.

`C_LOOP_J` **Loop Interno de Combinação.** Inicia o laço interno ('for j...'), que seleciona o segundo implicante para a comparação com o implicante 'i'.

`C_CHECK_COMBINE` **Verificação de Combinação.** Com dois implicantes selecionados ('i' e 'j'), este estado chama a função `can_combine`. Se eles puderem ser combinados (diferem por apenas um bit), ele avança para adicionar o novo termo; caso contrário, continua o laço interno.

`C_ADD_NEW_PI` **Adição do Novo Implicante.** Se uma combinação for válida, o novo termo combinado é gerado. Este estado então verifica se este novo termo já existe na `next_pi_table` para evitar duplicatas. Se for único, ele é adicionado. Em ambos os casos, o laço interno continua.

`C_PASS_FINISH` **Finalização do Passo de Combinação.** Ocorre quando os laços de comparação terminam. Apenas prepara para a próxima etapa.

`COLLECT_AND_DECIDE` **Coleta e Decisão.** O sistema percorre os implicantes do passe recém-concluído. Aqueles que não foram usados em nenhuma combinação (`pi_used_flags = '0'`) são Implicantes Primos e são movidos para a lista final (`final_pi_list`). Ao final, o sistema decide: se alguma combinação foi feita neste passe, ele inicia um novo passe com os termos recém-gerados; se não, todos os Implicantes Primos foram encontrados e ele avança para a construção da tabela de cobertura.

Fase 2: Encontrar a Cobertura Mínima

`BCT_INIT` **Inicialização da Construção da Tabela de Cobertura.** Zera os contadores para preparar a construção da matriz `coverage_table`.

`BCT_LOOP` **Loop de Construção da Tabela de Cobertura.** Através de um laço aninhado, este estado preenche a tabela. Para cada Implicante Primo final e cada mintermo original, ele chama a função `covers_minterm` e marca '1' na tabela se o implicante cobrir o mintermo.

`SEP_INIT` **Inicialização da Seleção de Essenciais.** Prepara para a busca por implicantes primos essenciais.

`SEP_FIND_OUTER_LOOP` **Loop Externo da Busca por Essenciais.** Itera sobre cada mintermo original.

`SEP_FIND_INNER_LOOP` **Loop Interno da Busca por Essenciais.** Para um dado mintermo, este estado itera sobre todos os Implicantes Primos, contando quantos deles cobrem este mintermo.

`SEP_FIND_EVALUATE` **Avaliação da Busca por Essenciais.** Ao final do loop interno, o sistema avalia a contagem. Se um mintermo é coberto por apenas um Implicante Primo (contagem = 1), este implicante é "essencial"e é marcado para inclusão na solução final (`pi_is_in_solution`).

`SEP_MARK_INIT, SEP_MARK_OUTER_LOOP, SEP_MARK_INNER_LOOP` **Marcação dos Mintermos Cobertos.** Após identificar todos os essenciais, estes estados percorrem a tabela de cobertura novamente. Todos os mintermos que são cobertos pelos implicantes essenciais são marcados como "resolvidos"no vetor `minterms_covered`.

Fase 3: Cobertura Final por Heurística (se necessário)

`CFC_CHECK_INIT, CFC_CHECK_LOOP` **Verificação da Cobertura Final.** Estes estados verificam se todos os mintermos já foram cobertos. Se sim, o algoritmo pula para a finalização. Se não, ele inicia a fase de cobertura por heurística.

`CFC_FIND_BEST_OUTER_INIT, CFC_FIND_BEST_OUTER_LOOP, CFC_FIND_BEST_INNER_LOOP, CFC_FIN`
Busca pelo Melhor Implicante (Heurística). Este conjunto de estados implementa uma heurística "gulosa". Ele analisa todos os Implicantes Primos ainda

não escolhidos e, para cada um, conta quantos mintermos *ainda não cobertos* ele consegue cobrir. Ao final, o estado `...EVALUATE` identifica o implicante "mais eficiente"(aquele com a maior contagem).

`CFC_UPDATE_INIT, CFC_UPDATE_LOOP` **Atualização da Cobertura.** O melhor implicante encontrado na fase anterior é adicionado à solução. Em seguida, os mintermos que ele cobre são marcados como "resolvidos". O sistema então retorna ao estado `CFC_CHECK_INIT` para verificar se o trabalho terminou ou se outra iteração da heurística é necessária.

Fase 4: Finalização

`FINALIZE_INIT, FINALIZE_LOOP, FINALIZE_WRITE` **Finalização e Formatação da Saída.** Estes estados coletam todos os implicantes marcados em `pi_is_in_solution`, os formatam no vetor de 64 bits `result_terms`, e contam o número final de termos. Também tratam os casos especiais onde a função é constante '0' ou '1'.

`DONE_STATE` **Concluído.** O estado final. O sinal de saída `done` é ativado para '1', informando aos outros módulos que o resultado está pronto. O sistema aguarda o sinal de `start` ser desativado para retornar ao estado `IDLE`.

5 Operação e Resultados

Para operar o sistema na placa DE10-Lite, é só seguir os seguintes passos:

1. Mantenha a chave `SW9` em '0' (modo de operação normal). Para reiniciar, mova para '1' e retorne a '0'.
2. As chaves `SW7` e `SW0` devem estar em '0'.
3. Para cada um dos 16 bits da tabela verdade (do mintermo 0 ao 15):
   - Coloque o valor do bit na chave `SW0` ('0' para baixo, '1' para cima).
   - Pressione e solte o botão `KEY0`.
   - Os LEDs `LEDR(3 downto 0)` indicarão o índice do próximo bit a ser inserido.
4. Após inserir os 16 bits, levante a chave `SW7` para a posição '1'.
5. O LED `LEDR(8)` acenderá, indicando que o algoritmo está em execução.
6. Ao final do processo, o LED `LEDR(9)` acenderá por um instante e o LED `LEDR(8)` apagará. O resultado estará pronto.
7. O display `HEX0` mostrará o primeiro caractere da expressão mínima.
8. Pressione o botão `KEY1` repetidamente para navegar pelos caracteres da expressão. Um ponto decimal aceso indica uma variável negada (ex: `A.` significa A').

6 Conclusão

O projeto demonstrou com sucesso a viabilidade da implementação de algoritmos lógicos complexos, como o de Quine-McCluskey, diretamente em hardware reconfigurável. A arquitetura modular provou ser eficiente para o desenvolvimento e depuração. O sistema final é robusto, funcional e oferece uma interface de usuário intuitiva.

Durante a fase de testes práticos na placa DE10-Lite, foi observado um comportamento problemático na ativação do algoritmo. O uso da chave SW7 como gatilho direto para o sinal de start mostrou-se, em certas ocasiões, pouco confiável. Isso ocorre porque a atuação de uma chave mecânica pode não ser interpretada como um evento único e limpo pelo sistema, levando a falhas na ativação. Como uma melhoria futura, propõe-se a alteração desta lógica: uma abordagem mais robusta seria utilizar a chave SW7 apenas para indicar a intenção de iniciar o cálculo. O disparo efetivo do algoritmo ocorreria então através de um evento discreto e confiável, como o pressionamento de um botão (KEY1, por exemplo). Esta implementação aumentaria consideravelmente a previsibilidade e a robustez da operação.

Como trabalho futuro, o projeto poderia ser modificado para suportar um número variável de entradas, permitindo que o sistema minimize funções booleanas com o número de varíaveis que o usuário escolher, embora isso traga desafios significativos relacionados ao consumo exponencial de recursos de hardware.

A Apêndice: Códigos-Fonte VHDL

A.1 Top.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Top is
     port (
          CLOCK_50 : in  std_logic;
          SW       : in  std_logic_vector(9 downto 0);
          KEY      : in  std_logic_vector(1 downto 0);
          LEDR     : out std_logic_vector(9 downto 0);
          HEX0     : out std_logic_vector(0 to 7);
          HEX1     : out std_logic_vector(0 to 7);
          HEX2     : out std_logic_vector(0 to 7);
          HEX3     : out std_logic_vector(0 to 7);
          HEX4     : out std_logic_vector(0 to 7);
          HEX5     : out std_logic_vector(0 to 7)
     );
end entity Top;

architecture structural of Top is

    component entrada is
         port (
            clk         : in  std_logic;
            reset       : in  std_logic;
            stop        : in  std_logic;
            data        : in  std_logic;
            next_bit    : in  std_logic;
```

```
            truth_table : out std_logic_vector(15 downto 0);
            counter     : out std_logic_vector(3 downto 0)
        );
    end component entrada;

    component quine_mccluskey is
         port (
            clk              : in  std_logic;
            reset            : in  std_logic;
            start            : in  std_logic;
            truth_table      : in  std_logic_vector(15 downto 0);
            done             : out std_logic;
            result_terms     : out std_logic_vector(63 downto 0);
            num_result_terms : out std_logic_vector(3 downto 0)
        );
    end component quine_mccluskey;

    component display is
         port (
            clk       : in  std_logic;
            reset     : in  std_logic;
            start     : in  std_logic;
            next_char : in  std_logic;
            terms     : in  std_logic_vector(63 downto 0);
            num_terms : in  std_logic_vector(3 downto 0);
            HEX0      : out std_logic_vector(0 to 7)
        );
    end component;

    signal s_reset             : std_logic;
    signal s_start             : std_logic;
    signal s_done              : std_logic;
    signal s_truth_table       : std_logic_vector(15 downto 0);
    signal s_result_terms      : std_logic_vector(63 downto 0);
    signal s_qm_num_terms      : std_logic_vector(3 downto 0);
    signal s_current_bit_index : std_logic_vector(3 downto 0);
    signal s_real_busy         : std_logic;

begin

    s_reset <= SW(9);
    s_start <= SW(7);

    LEDR(9)            <= s_done;
    LEDR(8)            <= s_real_busy;
    LEDR(7 downto 4)   <= (others => '0');
    LEDR(3 downto 0)   <= s_current_bit_index;

    process(CLOCK_50, s_reset)
    begin
        if s_reset = '1' then
            s_real_busy <= '0';
        elsif rising_edge(CLOCK_50) then
            if s_start = '1' then
                s_real_busy <= '1';
            elsif s_done = '1' then
                s_real_busy <= '0';
            end if;
```

```
87          end if;
88      end process;
89
90      input_ctrl : entrada
91          port map (
92              clk         => CLOCK_50,
93              reset       => s_reset,
94              stop        => s_real_busy,
95              data        => SW(0),
96              next_bit    => not KEY(0),
97              truth_table => s_truth_table,
98              counter     => s_current_bit_index
99          );
100
101     qm_inst : quine_mccluskey
102         port map (
103             clk              => CLOCK_50,
104             reset            => s_reset,
105             start            => s_start,
106             truth_table      => s_truth_table,
107             done             => s_done,
108             result_terms     => s_result_terms,
109             num_result_terms => s_qm_num_terms
110         );
111
112     display_inst : display
113         port map (
114             clk       => CLOCK_50,
115             reset     => s_reset,
116             start     => s_done,
117             next_char => not KEY(1),
118             terms     => s_result_terms,
119             num_terms => s_qm_num_terms,
120             HEX0      => HEX0
121         );
122
123     HEX1 <= (others => '1'); HEX2 <= (others => '1'); HEX3 <= (others =>
        '1');
124     HEX4 <= (others => '1'); HEX5 <= (others => '1');
125
126 end architecture structural;
```

A.2 entrada.vhd

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity entrada is
6      port (
7          clk         : in  std_logic;
8          reset       : in  std_logic;
9          stop        : in  std_logic;
10         data        : in  std_logic;
11         next_bit    : in  std_logic;
12
13         truth_table : out std_logic_vector(15 downto 0);
14         counter     : out std_logic_vector(3 downto 0)
```

```
    );
end entity entrada;

architecture behavioral of entrada is
    signal s_truth_table_reg : std_logic_vector(15 downto 0) := (others
   => '0');
    signal s_bit_index_reg   : integer range 0 to 15 := 0;
    signal s_prev_next       : std_logic := '0';
    signal s_prev_load_start_btn : std_logic := '0';

begin
    truth_table <= s_truth_table_reg;
    counter <= std_logic_vector(to_unsigned(s_bit_index_reg, 4));

    process(clk, reset)
    begin
        if reset = '1' then
            s_truth_table_reg <= (others => '0');
            s_bit_index_reg   <= 0;
            s_prev_next       <= '0';
        elsif rising_edge(clk) then
            s_prev_next <= next_bit;

            if stop = '0' then
                if next_bit = '1' and s_prev_next = '0' then
                    s_truth_table_reg(s_bit_index_reg) <= data;
                    if s_bit_index_reg = 15 then
                        s_bit_index_reg <= 0;
                    else
                        s_bit_index_reg <= s_bit_index_reg + 1;
                    end if;
                end if;
            end if;
        end if;
    end process;
end architecture behavioral;
```

A.3 quine_mccluskey.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity quine_mccluskey is
    port (
        clk              : in  std_logic;
        reset            : in  std_logic;
        start            : in  std_logic;
        truth_table      : in  std_logic_vector(15 downto 0);
        done             : out std_logic;
        result_terms     : out std_logic_vector(63 downto 0);
        num_result_terms : out std_logic_vector(3 downto 0)
    );
end entity;

architecture behavioral_sequential of quine_mccluskey is

    type b_number_type is record
```

```
        number : std_logic_vector(3 downto 0);
        dashes : std_logic_vector(3 downto 0);
    end record;

    type prime_implicants_array is array (0 to 31) of b_number_type;
    type minterms_array is array (0 to 15) of integer range 0 to 15;
    type coverage_table_type is array (0 to 31, 0 to 15) of std_logic;

    type state_type is (
        IDLE,

        IE_INIT, IE_LOOP,

        C_PASS_INIT, C_LOOP_I, C_LOOP_J, C_CHECK_COMBINE, C_ADD_NEW_PI,
    C_PASS_FINISH,
        COLLECT_AND_DECIDE,

        BCT_INIT, BCT_LOOP,

        SEP_INIT, SEP_FIND_OUTER_LOOP, SEP_FIND_INNER_LOOP,
    SEP_FIND_EVALUATE,
        SEP_MARK_INIT, SEP_MARK_OUTER_LOOP, SEP_MARK_INNER_LOOP,

        CFC_CHECK_INIT, CFC_CHECK_LOOP,
        CFC_FIND_BEST_OUTER_INIT, CFC_FIND_BEST_OUTER_LOOP,
    CFC_FIND_BEST_INNER_LOOP, CFC_FIND_BEST_EVALUATE,
        CFC_UPDATE_INIT, CFC_UPDATE_LOOP,

        FINALIZE_INIT, FINALIZE_LOOP, FINALIZE_WRITE,
        DONE_STATE
    );

    signal current_state    : state_type := IDLE;
    signal pi_table         : prime_implicants_array;
    signal next_pi_table    : prime_implicants_array;
    signal final_pi_list    : prime_implicants_array;
    signal pi_used_flags    : std_logic_vector(31 downto 0);
    signal num_pi           : integer range 0 to 32 := 0;
    signal num_next_pi      : integer range 0 to 32 := 0;
    signal num_final_pi     : integer range 0 to 32 := 0;

    signal minterms_list    : minterms_array;
    signal num_minterms     : integer range 0 to 16 := 0;

    signal coverage_table   : coverage_table_type;
    signal minterms_covered : std_logic_vector(15 downto 0);
    signal pi_is_in_solution: std_logic_vector(31 downto 0);

    signal i, j, k          : integer range 0 to 32;
    signal combination_made : std_logic;
    signal temp_pi          : b_number_type;

    signal best_pi_idx      : integer range -1 to 31;
    signal max_coverage     : integer range 0 to 16;

    signal s_temp_count         : integer range 0 to 32;
    signal s_temp_essential_idx : integer range -1 to 31;
    signal s_current_coverage   : integer range 0 to 16;
```

```
75      signal s_all_covered        : std_logic;
76      signal s_result             : std_logic_vector(63 downto 0);
77      signal s_count_terms        : integer range 0 to 8;
78
79      constant NULL_B_NUMBER  : b_number_type := (number => (others=>'0'),
        dashes => (others=>'0'));
80
81      function count_ones(number : std_logic_vector(3 downto 0)) return
        integer is
82          variable count : integer := 0;
83      begin
84          for i in number'range loop
85              if number(i) = '1' then
86                  count := count + 1;
87              end if;
88          end loop;
89          return count;
90      end function;
91
92      function can_combine(term1, term2 : b_number_type) return boolean is
93          variable diff_mask : std_logic_vector(3 downto 0);
94      begin
95          if term1.dashes /= term2.dashes then return false; end if;
96          diff_mask := (term1.number xor term2.number) and (not term1.
        dashes);
97          return (count_ones(diff_mask) = 1);
98      end function;
99
100     function covers_minterm(prime_impl : b_number_type; minterm :
        integer) return boolean is
101         variable minterm_vector : std_logic_vector(3 downto 0);
102     begin
103         minterm_vector := std_logic_vector(to_unsigned(minterm, 4));
104         return ((prime_impl.number and not prime_impl.dashes) = (
        minterm_vector and not prime_impl.dashes));
105     end function;
106
107 begin
108
109     done <= '1' when current_state = DONE_STATE else '0';
110
111     process(clk, reset)
112     begin
113         if reset = '1' then
114             current_state <= IDLE;
115             num_result_terms <= (others => '0');
116             result_terms <= (others => '0');
117             i <= 0; j <= 0; k <= 0;
118
119         elsif rising_edge(clk) then
120             case current_state is
121
122                 when IDLE =>
123                     if start = '1' then
124                         current_state <= IE_INIT;
125                     end if;
126
127                 when IE_INIT =>
```

```
128                     i <= 0;
129                     num_pi <= 0;
130                     num_minterms <= 0;
131                     pi_table <= (others => NULL_B_NUMBER);
132                     minterms_list <= (others => 0);
133                     num_final_pi <= 0;
134                     final_pi_list <= (others => NULL_B_NUMBER);
135                     current_state <= IE_LOOP;
136 
137                 when IE_LOOP =>
138                     if i < 16 then
139                         if truth_table(i) = '1' then
140                             pi_table(num_pi).number <= std_logic_vector(
    to_unsigned(i, 4));
141                             pi_table(num_pi).dashes <= "0000";
142                             minterms_list(num_minterms) <= i;
143                             num_pi <= num_pi + 1;
144                             num_minterms <= num_minterms + 1;
145                         end if;
146                         i <= i + 1;
147                     else
148                         if num_pi = 0 or num_pi = 16 then
149                             current_state <= FINALIZE_INIT;
150                         else
151                             current_state <= C_PASS_INIT;
152                         end if;
153                     end if;
154 
155 
156                 when C_PASS_INIT =>
157                     i <= 0; j <= 1; k <= 0;
158                     num_next_pi <= 0;
159                     next_pi_table <= (others => NULL_B_NUMBER);
160                     pi_used_flags <= (others => '0');
161                     combination_made <= '0';
162                     current_state <= C_LOOP_I;
163 
164                 when C_LOOP_I =>
165                     if i < num_pi - 1 then
166                         j <= i + 1;
167                         current_state <= C_LOOP_J;
168                     else
169                         current_state <= C_PASS_FINISH;
170                     end if;
171 
172                 when C_LOOP_J =>
173                     if j < num_pi then
174                         current_state <= C_CHECK_COMBINE;
175                     else
176                         i <= i + 1;
177                         current_state <= C_LOOP_I;
178                     end if;
179 
180                 when C_CHECK_COMBINE =>
181                     if can_combine(pi_table(i), pi_table(j)) then
182                         pi_used_flags(i) <= '1';
183                         pi_used_flags(j) <= '1';
184                         combination_made <= '1';
```

```
185                         temp_pi.number <= pi_table(i).number and
    pi_table(j).number;
186                         temp_pi.dashes <= pi_table(i).dashes or (
    pi_table(i).number xor pi_table(j).number);
187                         k <= 0;
188                         current_state <= C_ADD_NEW_PI;
189                     else
190                         j <= j + 1;
191                         current_state <= C_LOOP_J;
192                     end if;
193
194                 when C_ADD_NEW_PI =>
195                     if k < num_next_pi then
196                         if next_pi_table(k).number = temp_pi.number and
    next_pi_table(k).dashes = temp_pi.dashes then
197                             j <= j + 1;
198                             current_state <= C_LOOP_J;
199                         else
200                             k <= k + 1;
201                         end if;
202                     else
203                         next_pi_table(num_next_pi) <= temp_pi;
204                         num_next_pi <= num_next_pi + 1;
205                         j <= j + 1;
206                         current_state <= C_LOOP_J;
207                     end if;
208
209                 when C_PASS_FINISH =>
210                     k <= 0;
211                     current_state <= COLLECT_AND_DECIDE;
212
213                 when COLLECT_AND_DECIDE =>
214                     if combination_made = '1' then
215                         if k < num_pi then
216                             if pi_used_flags(k) = '0' then
217                                 final_pi_list(num_final_pi) <= pi_table(
    k);
218                                 num_final_pi <= num_final_pi + 1;
219                             end if;
220                             k <= k + 1;
221                         else
222                             pi_table <= next_pi_table;
223                             num_pi <= num_next_pi;
224                             current_state <= C_PASS_INIT;
225                         end if;
226                     else
227                         if k < num_pi then
228                             final_pi_list(num_final_pi) <= pi_table(k);
229                             num_final_pi <= num_final_pi + 1;
230                             k <= k + 1;
231                         else
232                             current_state <= BCT_INIT;
233                         end if;
234                     end if;
235
236
237                 when BCT_INIT =>
238                     i <= 0;
```

```
239                     j <= 0;
240                     coverage_table <= (others => (others => '0'));
241                     current_state <= BCT_LOOP;
242 
243                 when BCT_LOOP =>
244                     if i < num_final_pi then
245                         if j < num_minterms then
246                             if covers_minterm(final_pi_list(i),
    minterms_list(j)) then
247                                 coverage_table(i,j) <= '1';
248                             end if;
249                             j <= j + 1;
250                         else
251                             j <= 0;
252                             i <= i + 1;
253                         end if;
254                     else
255                         pi_is_in_solution <= (others => '0');
256                         minterms_covered <= (others => '0');
257                         current_state <= SEP_INIT;
258                     end if;
259 
260 
261                 when SEP_INIT =>
262                     i <= 0;
263                     current_state <= SEP_FIND_OUTER_LOOP;
264 
265                 when SEP_FIND_OUTER_LOOP =>
266                     if i < num_minterms then
267                         j <= 0;
268                         s_temp_count <= 0;
269                         s_temp_essential_idx <= -1;
270                         current_state <= SEP_FIND_INNER_LOOP;
271                     else
272                         current_state <= SEP_MARK_INIT;
273                     end if;
274 
275                 when SEP_FIND_INNER_LOOP =>
276                     if j < num_final_pi then
277                         if coverage_table(j, i) = '1' then
278                             s_temp_count <= s_temp_count + 1;
279                             s_temp_essential_idx <= j;
280                         end if;
281                         j <= j + 1;
282                     else
283                         current_state <= SEP_FIND_EVALUATE;
284                     end if;
285 
286                 when SEP_FIND_EVALUATE =>
287                     if s_temp_count = 1 then
288                         pi_is_in_solution(s_temp_essential_idx) <= '1';
289                     end if;
290                     i <= i + 1;
291                     current_state <= SEP_FIND_OUTER_LOOP;
292 
293                 when SEP_MARK_INIT =>
294                     i <= 0;
295                     current_state <= SEP_MARK_OUTER_LOOP;
```

```
296
297                 when SEP_MARK_OUTER_LOOP =>
298                     if i < num_final_pi then
299                         if pi_is_in_solution(i) = '1' then
300                             j <= 0;
301                             current_state <= SEP_MARK_INNER_LOOP;
302                         else
303                             i <= i + 1;
304                         end if;
305                     else
306                         current_state <= CFC_CHECK_INIT;
307                     end if;
308
309                 when SEP_MARK_INNER_LOOP =>
310                     if j < num_minterms then
311                         if coverage_table(i, minterms_list(j)) = '1'
    then
312                             minterms_covered(minterms_list(j)) <= '1';
313                         end if;
314                         j <= j + 1;
315                     else
316                         i <= i + 1;
317                         current_state <= SEP_MARK_OUTER_LOOP;
318                     end if;
319
320                 when CFC_CHECK_INIT =>
321                     i <= 0;
322                     s_all_covered <= '1';
323                     current_state <= CFC_CHECK_LOOP;
324
325                 when CFC_CHECK_LOOP =>
326                     if i < num_minterms then
327                         if minterms_covered(minterms_list(i)) = '0' then
328                             s_all_covered <= '0';
329                         end if;
330                         i <= i + 1;
331                     else
332                         if s_all_covered = '1' then
333                             current_state <= FINALIZE_INIT;
334                         else
335                             current_state <= CFC_FIND_BEST_OUTER_INIT;
336                         end if;
337                     end if;
338
339                 when CFC_FIND_BEST_OUTER_INIT =>
340                     i <= 0;
341                     max_coverage <= 0;
342                     best_pi_idx <= -1;
343                     current_state <= CFC_FIND_BEST_OUTER_LOOP;
344
345                 when CFC_FIND_BEST_OUTER_LOOP =>
346                     if i < num_final_pi then
347                         if pi_is_in_solution(i) = '0' then
348                            j <= 0;
349                            s_current_coverage <= 0;
350                            current_state <= CFC_FIND_BEST_INNER_LOOP;
351                         else
352                            i <= i + 1;
```

```
353                         end if;
354                     else
355                         current_state <= CFC_FIND_BEST_EVALUATE;
356                     end if;
357 
358                 when CFC_FIND_BEST_INNER_LOOP =>
359                     if j < num_minterms then
360                         if minterms_covered(minterms_list(j)) = '0' and
    coverage_table(i, j) = '1' then
361                            s_current_coverage <= s_current_coverage + 1;
362                         end if;
363                         j <= j + 1;
364                     else
365                         if s_current_coverage > max_coverage then
366                             max_coverage <= s_current_coverage;
367                             best_pi_idx <= i;
368                         end if;
369                         i <= i + 1;
370                         current_state <= CFC_FIND_BEST_OUTER_LOOP;
371                     end if;
372 
373                 when CFC_FIND_BEST_EVALUATE =>
374                      if best_pi_idx = -1 then
375                         current_state <= FINALIZE_INIT;
376                      else
377                         pi_is_in_solution(best_pi_idx) <= '1';
378                         current_state <= CFC_UPDATE_INIT;
379                      end if;
380 
381                 when CFC_UPDATE_INIT =>
382                     i <= 0;
383                     current_state <= CFC_UPDATE_LOOP;
384 
385                 when CFC_UPDATE_LOOP =>
386                     if best_pi_idx /= -1 and i < num_minterms then
387                         if coverage_table(best_pi_idx, i) = '1' then
388                            minterms_covered(minterms_list(i)) <= '1';
389                         end if;
390                         i <= i + 1;
391                     else
392                         current_state <= CFC_CHECK_INIT;
393                     end if;
394 
395 
396                 when FINALIZE_INIT =>
397                     s_result <= (others => '0');
398                     s_count_terms <= 0;
399                     i <= 0;
400                     if truth_table = "1111111111111111" then
401                          s_result(7 downto 0) <= "00001111";
402                          s_count_terms <= 1;
403                          current_state <= FINALIZE_WRITE;
404                     elsif truth_table = "0000000000000000" then
405                          current_state <= FINALIZE_WRITE;
406                     else
407                          current_state <= FINALIZE_LOOP;
408                     end if;
409 
```

```
410                 when FINALIZE_LOOP =>
411                     if i < num_final_pi then
412                         if pi_is_in_solution(i) = '1' and s_count_terms
    < 8 then
413                             s_result(s_count_terms*8 + 7 downto
    s_count_terms*8 + 4) <= final_pi_list(i).number;
414                             s_result(s_count_terms*8 + 3 downto
    s_count_terms*8)     <= final_pi_list(i).dashes;
415                             s_count_terms <= s_count_terms + 1;
416                         end if;
417                         i <= i + 1;
418                     else
419                         current_state <= FINALIZE_WRITE;
420                     end if;
421
422                 when FINALIZE_WRITE =>
423                     result_terms <= s_result;
424                     num_result_terms <= std_logic_vector(to_unsigned(
    s_count_terms, 4));
425                     current_state <= DONE_STATE;
426
427                 when DONE_STATE =>
428                     if start = '0' then
429                         current_state <= IDLE;
430                     end if;
431
432                 when others =>
433                     current_state <= IDLE;
434
435             end case;
436         end if;
437     end process;
438
439 end architecture;
```

A.4 display.vhd

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity display is
6      port (
7          clk       : in  std_logic;
8          reset     : in  std_logic;
9          start     : in  std_logic;
10         next_char : in  std_logic;
11         terms     : in  std_logic_vector(63 downto 0);
12         num_terms : in  std_logic_vector(3 downto 0);
13         HEX0      : out std_logic_vector(0 to 7)
14     );
15 end entity display;
16
17 architecture behavioral of display is
18
19     type T_BUILD_STATE is (S_IDLE, S_BUILDING, S_DONE_BUILDING);
20
21     type display_char_type is record
```

```
        char    : character;
        is_neg  : boolean;
    end record;

    type display_sequence_type is array (0 to 39) of display_char_type;

    signal s_build_state      : T_BUILD_STATE := S_IDLE;
    signal s_display_sequence : display_sequence_type;
    signal s_display_idx      : integer range 0 to 39 := 0;
    signal s_sequence_len     : integer range 0 to 40 := 0;

    signal s_build_term_idx   : integer range 0 to 8 := 0;
    signal s_build_literal_idx: integer range 0 to 4 := 0;
    signal s_build_seq_idx    : integer range 0 to 40 := 0;

    signal s_prev_next_char   : std_logic := '0';

begin

    process(clk, reset)
        variable term_vec       : std_logic_vector(7 downto 0);
        variable term_num       : std_logic_vector(3 downto 0);
        variable term_dash      : std_logic_vector(3 downto 0);
        variable new_char       : display_char_type;
        variable literal_char   : character;
    begin
        if reset = '1' then
            s_build_state <= S_IDLE;
            s_build_term_idx <= 0;
            s_build_literal_idx <= 0;
            s_build_seq_idx <= 0;
            s_sequence_len <= 0;
        elsif rising_edge(clk) then
            case s_build_state is
                when S_IDLE =>
                    if start = '1' then
                        s_build_state <= S_BUILDING;
                        s_build_term_idx <= 0;
                        s_build_literal_idx <= 0;
                        s_build_seq_idx <= 0;
                    end if;

                when S_BUILDING =>
                    if s_build_term_idx < to_integer(unsigned(num_terms)
) then
                        term_vec  := terms(s_build_term_idx*8 + 7 downto
 s_build_term_idx*8);
                        term_num  := term_vec(7 downto 4);
                        term_dash := term_vec(3 downto 0);

                        if s_build_literal_idx = 4 then
                            new_char := (char => '+', is_neg => false);
                            s_display_sequence(s_build_seq_idx) <=
new_char;
                            s_build_seq_idx <= s_build_seq_idx + 1;
                            s_build_term_idx <= s_build_term_idx + 1;
                            s_build_literal_idx <= 0;
                        else
```

```
                            case s_build_literal_idx is
                                when 0 => literal_char := 'A';
                                when 1 => literal_char := 'b';
                                when 2 => literal_char := 'C';
                                when 3 => literal_char := 'd';
                                when others => literal_char := ' ';
                            end case;

                            if term_dash(3 - s_build_literal_idx) = '0' then
                                new_char := (char => literal_char, is_neg => (term_num(3 - s_build_literal_idx) = '0'));
                                s_display_sequence(s_build_seq_idx) <= new_char;
                                s_build_seq_idx <= s_build_seq_idx + 1;
                            end if;

                            s_build_literal_idx <= s_build_literal_idx + 1;
                        end if;
                    else
                        s_build_state <= S_DONE_BUILDING;
                        s_sequence_len <= s_build_seq_idx;
                    end if;

                when S_DONE_BUILDING =>
                    if start = '0' then
                        s_build_state <= S_IDLE;
                    end if;
            end case;
        end if;
    end process;

    process(clk, reset)
    begin
        if reset = '1' then
            s_display_idx <= 0;
            s_prev_next_char <= '0';
        elsif rising_edge(clk) then
            s_prev_next_char <= next_char;
            if s_build_state = S_DONE_BUILDING then
                if next_char = '1' and s_prev_next_char = '0' then
                    if s_sequence_len > 0 then
                        if s_display_idx < s_sequence_len - 1 then
                            s_display_idx <= s_display_idx + 1;
                        else
                            s_display_idx <= 0;
                        end if;
                    end if;
                end if;
            else
                s_display_idx <= 0;
            end if;
        end if;
    end process;

    process(s_display_sequence, s_display_idx, s_build_state, num_terms, terms)
```

```
130     variable current_display_char : display_char_type;
131     variable char_pattern : std_logic_vector(0 to 6);
132     begin
133          if s_build_state /= S_DONE_BUILDING then
134               HEX0 <= (others => '1');
135          else
136               if to_integer(unsigned(num_terms)) = 0 then
137                    char_pattern := "0000001";
138                    HEX0(0 to 6) <= char_pattern;
139                    HEX0(7) <= '1';
140
141               elsif to_integer(unsigned(num_terms)) = 1 and terms(3
    downto 0) = "1111" then
142                    char_pattern := "1001111";
143                    HEX0(0 to 6) <= char_pattern;
144                    HEX0(7) <= '1';
145
146               else
147                    current_display_char := s_display_sequence(
    s_display_idx);
148
149                    case current_display_char.char is
150                         when 'A'    => char_pattern := "0001000";
151                         when 'b'    => char_pattern := "1100000";
152                         when 'C'    => char_pattern := "0110001";
153                         when 'd'    => char_pattern := "1000010";
154                         when '+'    => char_pattern := "1000001";
155                         when others => char_pattern := "1111111";
156                    end case;
157
158                    HEX0(0 to 6) <= char_pattern;
159
160                    if current_display_char.is_neg then
161                         HEX0(7) <= '0';
162                    else
163                         HEX0(7) <= '1';
164                    end if;
165               end if;
166          end if;
167     end process;
168
169 end architecture behavioral;
```