Universitetet i Sørøst-Norge

Fakultet for TNM

Eksamensoppgave

Programmerbare logiske kretser

Vår 2025

Gamma-code encoder

Finite state machine

Innhold

1.0 Introduksjon

I dette prosjektet skal jeg implementere en Gamma-kode-enkoder ved hjelp av en Finite State Machine (FSM) på FPGA-kort. Gamma-koden benytter et mønster av pulser med forskjellige varighet for å representere symboler som bokstaver, tall og spesialtegn. Hvert symbol kodes som en sekvens av kort (dot), middels (dash) og lang (bar) puls.

Dot er definert som fire påfølgende 0.25 sekunders pulser, dash varer i 0.75 sekunder, og bar varer 1.5 sekunder.

Systemet bruker bryterne SW_{3-0} og knappene KEY_{1-0} som innganger. Bryterne velger symboler som skal kodes, men KEY_1 starter visningen av Gamma-koden på $LEDR_0$. KEY_0 fungerer som en aktiv høy reset knapp og aktiverer $LEDR_9$.

Hensikten med prosjektet er å designe et digitalt system som kan oversette et valgt symbol til lysblink med korrekt tidsvarighet, og på den måten realisere en funksjonell og tidsstyrt enkoder. Løsningen implementeres i VHDL og verifiseres gjennom både simulasjon og testing på fysisk FPGA.

2.0 Fremgang til løsning

Prosjektet har jeg valgt å løse ved å dele koden inn i flere delmoduler, hvor hver modul har sin unike funksjon. Hensikten med denne strukturen er å få bedre oversikt over systemet, samt å gjøre det mulig å testes hver modul separat. Delmodulene for syv segment displayet, skiftregister og teller hadde jeg laget i en tidligere lab oppgave, og disse er gjenbrukt med noen tilpasning. Top-modulen kobler sammen alle delmodulene og håndterer inngangene fra bryter og knapper, samt utgangene til LED og 7-segment display.

Gamma lookup table

Denne modulen tar inn en 4-bits verdi fra bryterne og konverterer den til en 8-bit gamma kode. Hver kode representerer et symbol, tall eller bokstav og består av 2-bit sekvenser der : «00» = dot (kort pus), «01» = dash (middels puls), «10» = bar (lang puls), «11» = NULL. Gamma koden er harkodet i et case statement. Tabbelen under viser alle symbolene er kodet.

Tabell 1 Gamma LUT

SYMBOL	4-BIT FRA SW	GAMMA KODE
P	0000	01010000
B	0001	10000000
1	0010	10000100
8	0011	00100000
0	0100	01000100
-	0101	01000000
F	0110	10010000

G	0111	10000010
A	1000	00010000
OTHERS		11111111

Gamma skiftregister

Skiftregisteret lagrer den 8-bit lange gamma koden som genereres av gamma lookup tabellen, og sender ut 2-bits sekvenser èn og èn til FSM-en. Etter hvert skift fylles de nederste bittene med «00» for å opprettholde registerets lengde. Dette gjøres den ved å konkatinere de seks nederste bittene med «00». FSM-en bruker den mottatte 2-bit sekvensen til å bestemme varigheten på LED blinket.

Tabell 2 2-bit sekvens repesentasjon for puls

2-BIT KODE	PULS TYPE	VARIGHET I SENKUNDER
00	KORT PULS (DOT)	0.25 sek
01	MEDIUM PULS (DASH)	0.75 sek
10	LANG PULS (BAR)	1.50 sek
11	UGYLDIG	-

Kvart sekunds klokke

Denne komponentene bruker FPGAens 50 MHz klokke og teller opp til en konstant (k = 12 500 000) for å generere en puls(tick_out) hver 0.25 sekund. Dette fungerer som tidsbase i FSM-en, som igjen bruker tick_out til å styre både blinkvarighet og tilstandsovergangene.

Verdien K er beregnet fra formelen:

$K = (\text{klokkefrekvens} / \text{ønsket tidsintervall}) = 50\,000\,000 / 4 = 12\,500\,000$

For å beregne antall bit når nye kokke kan håndtere bruker vi formelen:

$2^N >= K$

$N = \log_2(K) = 23{,}6 \text{ bit runder opp til } 24 \text{ bit}$

Dette betyr at det minste bitte vi kan bruke er 24 bit. [1]

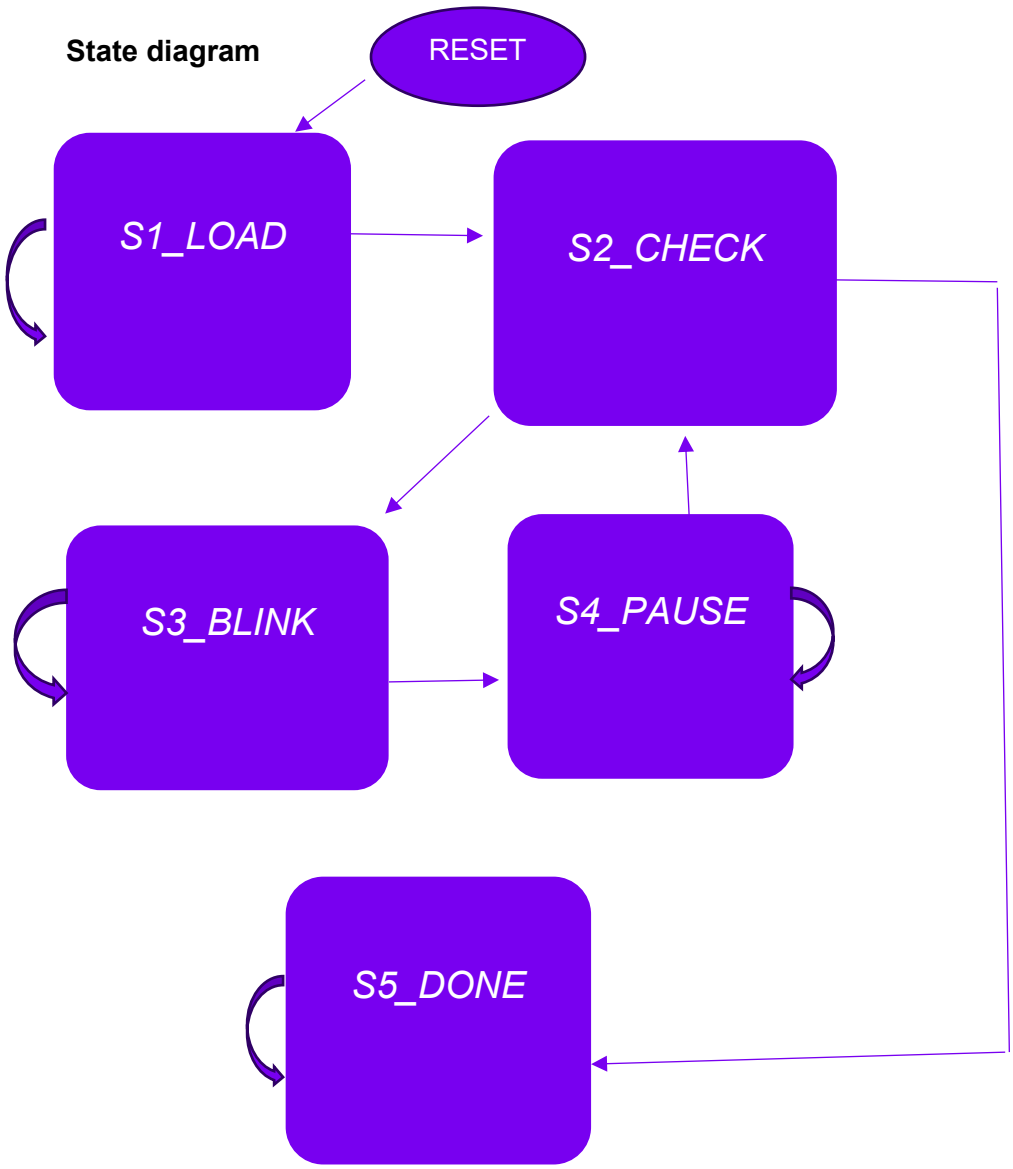
Finite State Machine (FSM)

FSM fungerer som kontrollenhet for systemet og styrer hvordan gamma koden tolkes og vises som LED blink. I dette prosjektet er den implementert som en Moore.maskin. Det betyr at utgangen led_out kun er avhengig av nåværende tilstand og ikke ingangssignalet. Jeg har valgt å implementere slikt for en mer stabil og forutsigbar atferd ettersom nøyaktik timing kreves. [2]

FSM-en består av fem hovedtilstander:

- S1_LOAD : Leser inn ny gamma kode fra gamma_lut og laster den inn i skiftregisteret.
- S2_CHECK : skifter ut de øveste 2-bit fra skiftregisteret og sjekker om det gjenstår mer data.
- S3_BLINK: lyser LED i en varighet bestemt av 2-bit koden (dot, dash eller bar).
- S4_PAUSE : kort pause mellom symbolene for visuell separasjon.
- S5_done: FSM avslutters etter all data er sendt, og venter på en ny startkommando.

FSM bruker qsec_tick signalet som tidsbase og en intern teller for å kontrollere hvor lenge LED skal være aktiv i blinktilstanden.



Figur 1 State transition diagram for FSM

Tabell 3 Satate transition table

Nåværende tilstand	Start	Empty	Tick	Neste tilstand	Led_out	Load_reg	Shift_en
S1_LOAD	1	X	X	S2_CHECK	0	1	0
S2_CHECK	X	0	X	S3_BLINK	0	0	1
S2_CHECK	X	1	X	S5_DONE	0	0	0
S3_BLINK	X	X	0	S3_BLINK	1	0	0
S3_BLINK	X	X	1	S4_PAUSE	1	0	0
S4_PAUSE	X	X	1	S2_CHECK	0	0	0
S5_DONE	0	X	X	S1_LOAD	0	0	0
S5_DONE	1	X	X	S5_DONE	0	0	0

3.0 VHDL PROGRAM

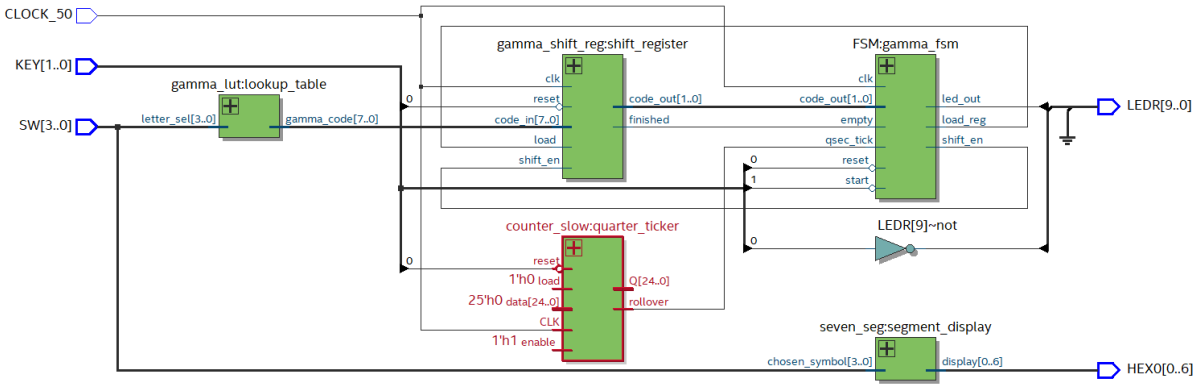
VHDL koden gamma_code_encoder kobler sammen delmodulene, samtidig som denhåndterer alle inngangene og utgangene som brytere, knappene, LED og 7-segment displayet. Koden har fem viktige steg:

1. Symbolvalg: Bruker velger et symbol ved hjelp av bryterne (SW). Denne verdien sendes til gamma_lut, som slår opp riktig 8-bit gamma kode.
2. Visning: Samtidig sendes SW verdien til seven_seg for å vise valgt symbol på display.
3. kodelagring og skifting: Når bruker trykker KEY_0 aktiveres start. FSM starter prosessen og gir kontrollsignaler til gamma_shift_reg, som skifter ut to og to bits av gamma koden.
4. Blink sekvens: FSM leser 2-bit sekvensen og bestemmer blinketid og LEDR(0) baser på symboltypen (dot, dash, bar). FSM bruker tick_qsec fra counter_slow som tidsreferanse.
5. Fullført: Når hele gamma kodener blinket, settes finished og FSM går i DONE tilstand (state 5).

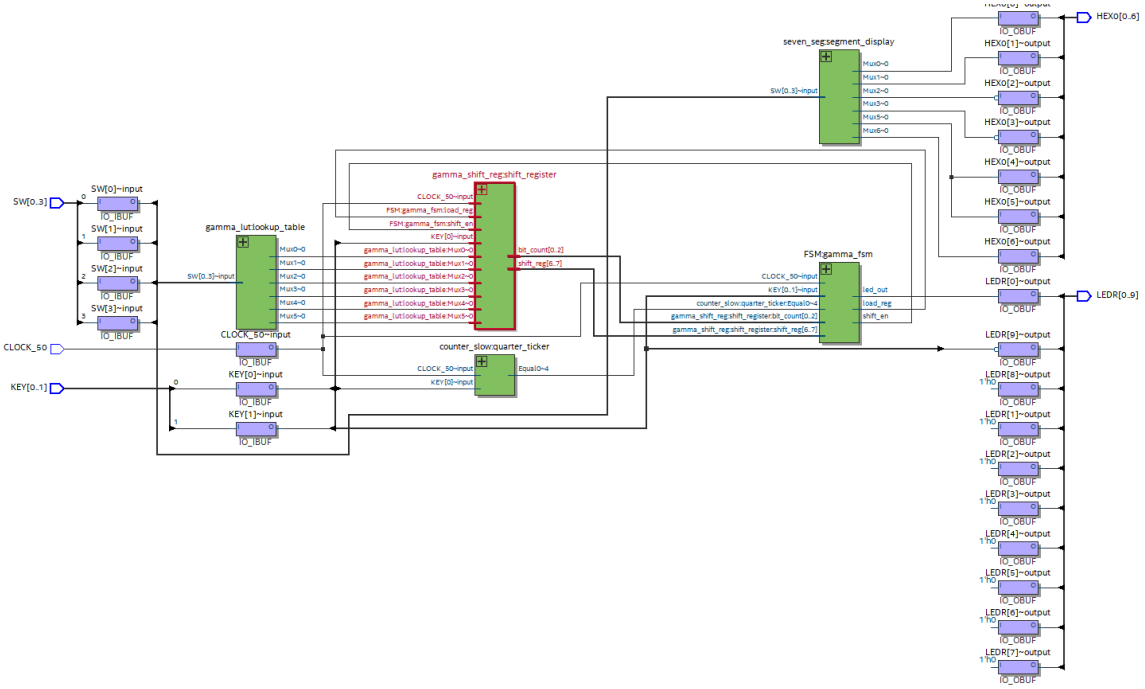
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity gamma_code_encoder is
    port (
        CLOCK_50 : in  std_logic;
        SW       : in  std_logic_vector(3 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 6)
    );
end gamma_code_encoder;

architecture structural of gamma_code_encoder is

    component counter_slow is
        generic( n : NATURAL; k : INTEGER);
        port(
            CLK, reset, load : in std_logic;
            enable           : in std_logic;
            data             : in std_logic_vector(n-1 downto 0);
            Q                : out std_logic_vector(n-1 downto 0);
            rollover         : out std_logic
        );
    end component;

    component gamma_lut is
        port (
            letter_sel : in  std_logic_vector(3 downto 0);
```

```
            gamma_code : out std_logic_vector(7 downto 0)
        );
    end component;

    component seven_seg is
        port (
            chosen_symbol : in  std_logic_vector(3 downto 0);
            display       : out std_logic_vector(0 to 6)
        );
    end component;

    component gamma_shift_reg is
        port (
            clk, reset, load, shift_en : in  std_logic;
            code_in                    : in  std_logic_vector(7 downto 0);
            code_out                   : out std_logic_vector(1 downto 0);
            finished                   : out std_logic
        );
    end component;

    component FSM is
        port (
            clk       : in  std_logic;
            reset     : in  std_logic;
            start     : in  std_logic;
            qsec_tick : in  std_logic;
            code_out  : in  std_logic_vector(1 downto 0);
            empty     : in  std_logic;
            load_reg  : out std_logic;
            shift_en  : out std_logic;
            led_out   : out std_logic
        );
    end component;

    signal tick_qsec      : std_logic;
    signal gamma_code     : std_logic_vector(7 downto 0);
    signal code_out       : std_logic_vector(1 downto 0);
    signal finished       : std_logic;
    signal shift_en       : std_logic;
    signal load_reg       : std_logic;
    signal start          : std_logic;
    signal reset          : std_logic := '0';
    signal toggle_test    : std_logic := '0';
    signal tick_stretched : std_logic := '0';
    signal stretch_count  : integer range 0 to 10 := 0;

begin

    start <= not KEY(1);
    reset <= not KEY(0);

    -- 0.25s klokke
    quarter_ticker : counter_slow
                                generic map (
                                      n => 25,
                                      k => 12500000
                                )
                                port map (
                                      CLK      => CLOCK_50,
```

```
                                        reset    => reset,
                                        load     => '0',
                                        enable   => '1',
                                        data     => (others => '0'),
                                        Q        => open,
                                        rollover => tick_qsec
                                    );


    lookup_table : gamma_lut
                                port map (
                                        letter_sel => SW,
                                        gamma_code => gamma_code
                                );

    segment_display : seven_seg
                                port map (
                                        chosen_symbol => SW,
                                        display       => HEX0
                                );

    shift_register : gamma_shift_reg
                                port map (
                                        clk       => CLOCK_50,
                                        reset     => reset,
                                        load      => load_reg,
                                        shift_en  => shift_en,
                                        code_in   => gamma_code,
                                        code_out  => code_out,
                                        finished  => finished
                                );

    gamma_fsm : FSM
                        port map (
                                clk       => CLOCK_50,
                                reset     => reset,
                                start     => start,
                                qsec_tick => tick_qsec,
                                code_out  => code_out,
                                empty     => finished,
                                load_reg  => load_reg,
                                shift_en  => shift_en,
                                led_out   => LEDR(0)
                        );

LEDR(9) <= reset;

end structural;
```



Figur 2 RTL Viewer av gamma-code-encoder



Figur 3 Technology Viewer av gamma-code-encoder

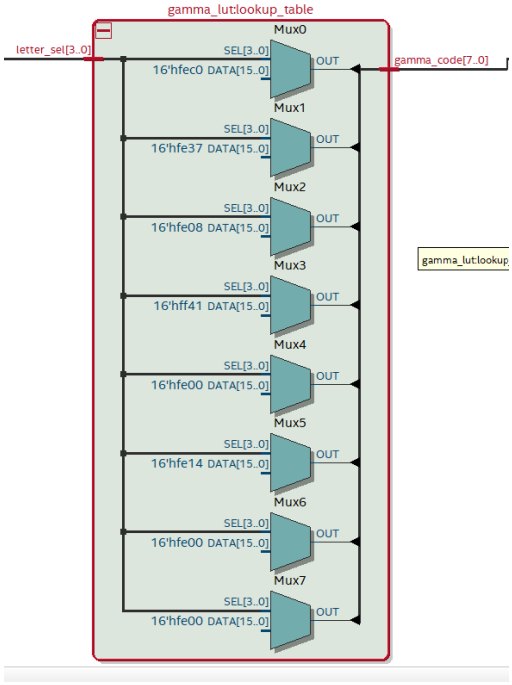
3.1 Sub-modules

3.1.1 Gamma lookuptable

Gamma_lut komponenten fungerer som enn oppslagstavle. Den tar inn en 4-bit verdi(letter_sel) fra bryterne(SW), som representerer et valgt symbol, tall eller bokstav. Basert på denne inngangen slår komponenten opp en forhåndsdefinert 8-bit gamma kode i en case struktur inne i en prosess.

Hver 8-bit gamma kode består av fire 2-bit sekvenser, der hver sekvens representerer en type puls. Som tidligere forklart representerer «00» = dot, «01» = dash og «10» = bar. Koden gjør derfor enkelt å koble et SW symbol til en bestemt sekvens av blinkemønster som sendes videre til skiftregisteret.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity gamma_lut is
    port(
        letter_sel : in  std_logic_vector(3 downto 0);
        gamma_code : out std_logic_vector(7 downto 0)
    );
end gamma_lut;

architecture behavior of gamma_lut is
begin
    process(letter_sel)
    begin
        case letter_sel is
            when "0000" => gamma_code <= "01010000"; -- P: dash dash dot
            when "0001" => gamma_code <= "01000000"; -- B: bar dot dot
            when "0010" => gamma_code <= "01000100"; -- 1: bar dot dash
            when "0011" => gamma_code <= "00100000"; -- 8: dot bar dot
            when "0100" => gamma_code <= "01000100"; -- 0: dash dot dash
            when "0101" => gamma_code <= "01000000"; -- -: dash
            when "0110" => gamma_code <= "10010000"; -- F: bar dash dot
            when "0111" => gamma_code <= "10000000"; -- G: bar dot dot bar
            when "1000" => gamma_code <= "00010000"; -- A: dot dash dot
            when others => gamma_code <= "11111111"; -- ugyldig kode
        end case;
    end process;
end behavior;
```



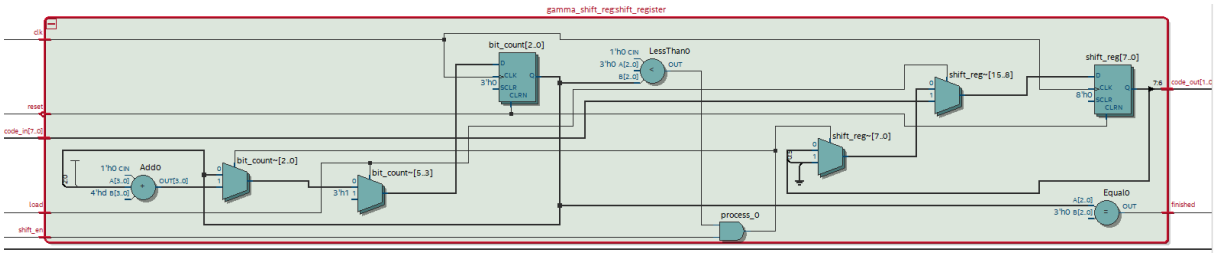
Figur 4 RTL Viewer av gamma_loopuptable

3.1.2 Gamma shift register

Gamma_shift_reg er en 8-bit skiftregister. Når load er aktiv, lagres en ny 8-bit gamma kode. Ved hver shift_en signal flyttes registere 2-bit til venstre og «00» legges inn på LSB. De to MSB (code_out) sendes videre til FSM-en for å bestemme blinkevarighet. Etter fire bit skift er hele gamma koden tømt. Da aktiveres signalet finished, som forteller FMS at sekvensen er ferdig.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity gamma_shift_reg is
    port (
        clk, reset, load, shift_en : in  std_logic;
        code_in                    : in  std_logic_vector(7 downto 0);
        code_out                   : out std_logic_vector(1 downto 0);
        finished                   : out std_logic
    );
end gamma_shift_reg;

architecture behavior of gamma_shift_reg is
    signal shift_reg : std_logic_vector(7 downto 0) := (others => '0');
    signal bit_count : integer range 0 to 4 := 0;
begin
    process(clk, reset)
    begin
        if reset = '1' then
```

```
            shift_reg <= (others => '0');
            bit_count <= 0;
        elsif rising_edge(clk) then
            if load = '1' then
                shift_reg <= code_in;
                bit_count <= 4;
            elsif shift_en = '1' and bit_count > 0 then
                shift_reg <= shift_reg(5 downto 0) & "00";
                bit_count <= bit_count - 1;
            end if;
        end if;
    end process;

    code_out <= shift_reg(7 downto 6);
    finished <= '1' when bit_count = 0 else '0';
end behavior;
```



Figur 5 RTL Viewer av gamma shift register

3.1.3 FSM

FSM-en fungerer som en kontrollenheten for hele systemet.[1] Den styrer når en nye kode skal lastes, når LED skal blinke, og når en pause skal holdes mellom symbolene. Koden tar i mort 2-bit sekvensene fra skiftregisteret og styrer varigheten på blink ut fra disse.

FMS har fem tilstander som er inndelt i:

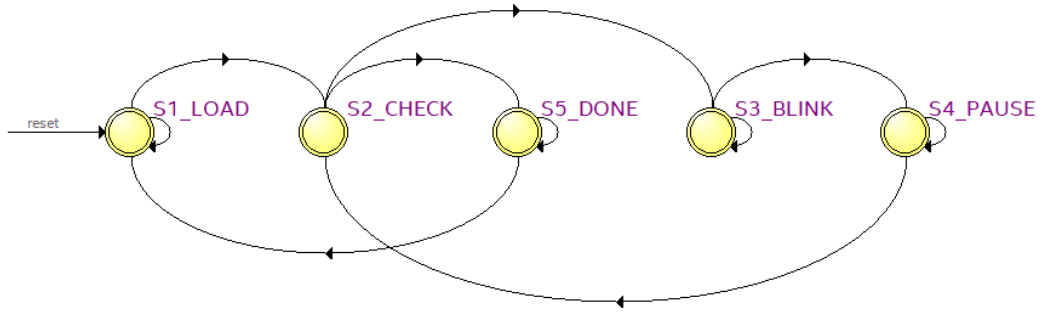
- S1 LOAD: leser inn ny gamma kode
- S2 CHECK: leser og skifter til nesten sekvens
- S3 BLINK: aktiverer LED basert på code_out
- S4 PAUSE: kort pause mellom blink
- S5 DONE: Avslutter blinksekvens

Den bruker tick_qsec som en 0.25s tidsbase og signalene shift_en, load_reg og led_out for kontroll og utdata.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity FSM is
    port (
        clk       : in  std_logic;
        reset     : in  std_logic;
        start     : in  std_logic;
        qsec_tick : in  std_logic;
        code_out  : in  std_logic_vector(1 downto 0);
        empty     : in  std_logic;
        load_reg  : out std_logic;
        shift_en  : out std_logic;
        led_out   : out std_logic
    );
end FSM;

architecture Behavioral of FSM is
    type state_type is (S1_LOAD, S2_CHECK, S3_BLINK, S4_PAUSE, S5_DONE);
    signal state, next_state : state_type;

    signal led_timer : integer range 0 to 6 := 0;
    signal duration  : integer range 1 to 6 := 1;

begin

    -- State register
    process(clk, reset)
    begin
        if reset = '1' then
            state <= S1_LOAD;
        elsif rising_edge(clk) then
            state <= next_state;
        end if;
    end process;

    -- Next state logic
    process(state, start, qsec_tick, empty, led_timer)
    begin
        case state is
            when S1_LOAD =>
                if start = '1' then
                    next_state <= S2_CHECK;
                else
                    next_state <= S1_LOAD;
                end if;

            when S2_CHECK =>
                if empty = '0' then
                    next_state <= S3_BLINK;
                else
                    next_state <= S5_DONE;
                end if;
```

```
            when S3_BLINK =>
                if qsec_tick = '1' and led_timer = duration then
                    next_state <= S4_PAUSE;
                else
                    next_state <= S3_BLINK;
                end if;

            when S4_PAUSE =>
                if qsec_tick = '1' then
                    next_state <= S2_CHECK;
                else
                    next_state <= S4_PAUSE;
                end if;

            when S5_DONE =>
                if start = '0' then
                    next_state <= S1_LOAD;
                else
                    next_state <= S5_DONE;
                end if;
        end case;
    end process;

    -- Output logic and duration handling
    process(clk, reset)
    begin
        if reset = '1' then
            load_reg  <= '0';
            shift_en  <= '0';
            led_out   <= '0';
            led_timer <= 0;
            duration  <= 1;
        elsif rising_edge(clk) then
            case state is

                when S1_LOAD =>
                    load_reg <= '1';
                    shift_en <= '0';
                    led_out  <= '0';

                when S2_CHECK =>
                    load_reg <= '0';
                    shift_en <= '1';
                    led_timer <= 0;

                    case code_out is
                        when "00" => duration <= 1; -- dot
                        when "01" => duration <= 3; -- dash
                        when "10" => duration <= 6; -- bar
                        when others => duration <= 1;
                    end case;

                when S3_BLINK =>
                    load_reg <= '0';
                    shift_en <= '0';
                    led_out  <= '1';
                    if qsec_tick = '1' then
                        led_timer <= led_timer + 1;
                    end if;
```

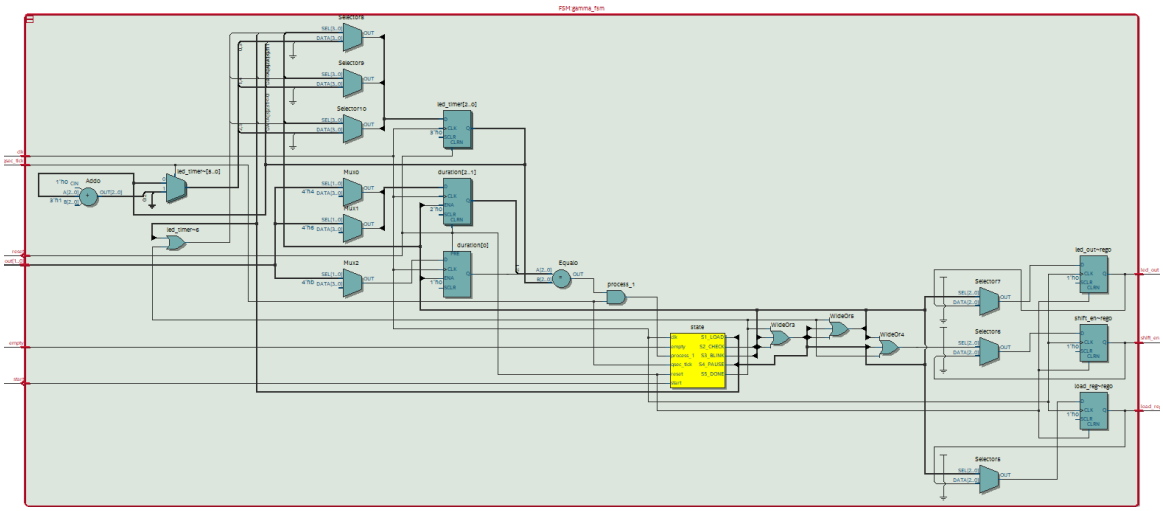
```
                when S4_PAUSE =>
                    led_out <= '0';
                    if qsec_tick = '1' then
                        led_timer <= 0;
                    end if;

                when S5_DONE =>
                    led_out  <= '0';
                    shift_en <= '0';
                    load_reg <= '0';

            end case;
        end if;
    end process;

end Behavioral;
```



Figur 6 FSM state trantion diagram

Tabell 4 FSM state transtion table

	Source State	Destination State	Condition
1	S1_LOAD	S1_LOAD	(!start)
2	S1_LOAD	S2_CHECK	(start)
3	S2_CHECK	S5_DONE	(empty)
4	S2_CHECK	S3_BLINK	(!empty)
5	S3_BLINK	S3_BLINK	(!process_1)
6	S3_BLINK	S4_PAUSE	(process_1)
7	S4_PAUSE	S2_CHECK	(qsec_tick)
8	S4_PAUSE	S4_PAUSE	(!qsec_tick)
9	S5_DONE	S5_DONE	(start)
10	S5_DONE	S1_LOAD	(!start)



Figur 7 Technology Viewer av FSM

3.1.4 Seven segment display

VHDL koden seven_seg konverterer en 4-bit inngangsverdi til passende binærmønster for å vise en bokstav eller et symbol på et 7-segment display. Dette gir visuell bekreftelse på hvilket symbol som er valgt fra bryterne.

```
library ieee;
use ieee.std_logic_1164.all;

entity seven_seg is
    port (
        chosen_symbol : in  std_logic_vector(3 downto 0);
        display       : out std_logic_vector(0 to 6)
    );
end seven_seg;

architecture behavior of seven_seg is
```

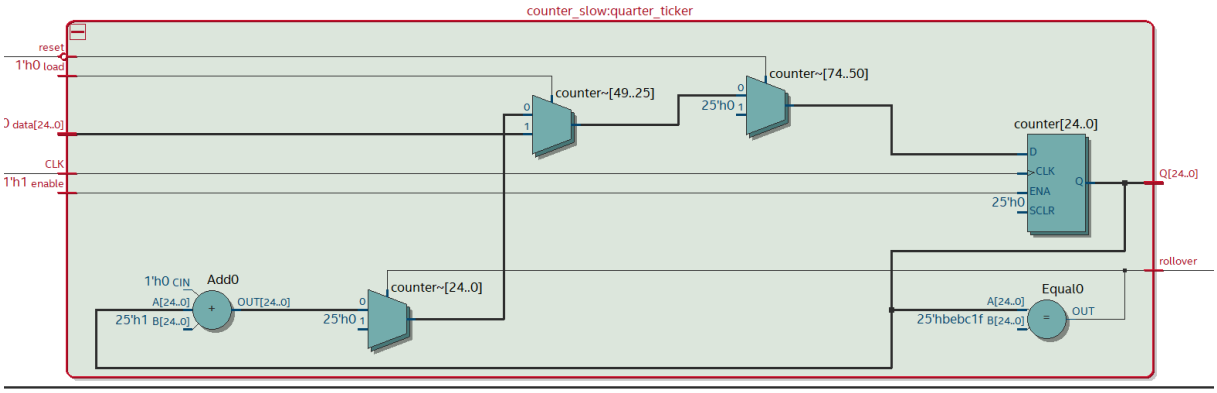
```
begin
    process(chosen_symbol)
    begin
        case chosen_symbol is
            when "0000" => display <= "0011000"; -- P
            when "0001" => display <= "1100000"; -- B
            when "0010" => display <= "1001111"; -- 1
            when "0011" => display <= "0000000"; -- 8
            when "0100" => display <= "0000001"; -- 0
            when "0101" => display <= "1111110"; -- -
            when "0110" => display <= "0111000"; -- F
            when "0111" => display <= "0100001"; -- G
            when "1000" => display <= "0001000"; -- A
            when others => display <= "1111111"; -- blank
        end case;
    end process;
end behavior;
```

3.1.4 Quarter second counter

VHDL koden counter_slow en er generisk klokke som brukes til å lage tidsbaserte signaler. Den bruker FPGA-enes 50 MHz klokke og teller opp til en spesifisert verdi k. Når dette tallet er nådd, sendes et rollover signal so tikker en gang. I dette prosjektet brukes den til å gi et tick_qsec signal hver 0.25 sekund som brukes av FSM-en til å styre blinkvarighet for LED.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity counter_slow is
    generic (
        n : natural := 25;
        k : integer := 50_000_000  -- 1 sekund ved 50 MHz
    );
    port (
        CLK      : in  std_logic;
        reset    : in  std_logic;
        load     : in  std_logic;
        enable   : in  std_logic;
        data     : in  std_logic_vector(n-1 downto 0);
        Q        : out std_logic_vector(n-1 downto 0);
        rollover : out std_logic
    );
end counter_slow;

architecture behavior of counter_slow is
    signal counter : unsigned(n-1 downto 0) := (others => '0');
    signal limit   : unsigned(n-1 downto 0) := to_unsigned(k - 1, n);
begin

    process(CLK)
    begin
        if rising_edge(CLK) then
```

```
            if enable = '1' then
                if reset = '1' then
                    counter <= (others => '0');
                elsif load = '1' then
                    counter <= unsigned(data);
                elsif counter = limit then
                    counter <= (others => '0');
                else
                    counter <= counter + 1;
                end if;
            end if;
        end if;
    end process;

    Q <= std_logic_vector(counter);
    rollover <= '1' when counter = limit else '0';
end behavior;
```



Figur 8 Technology Viewer av Uarter second counter

4.0 Testbench and waveforms

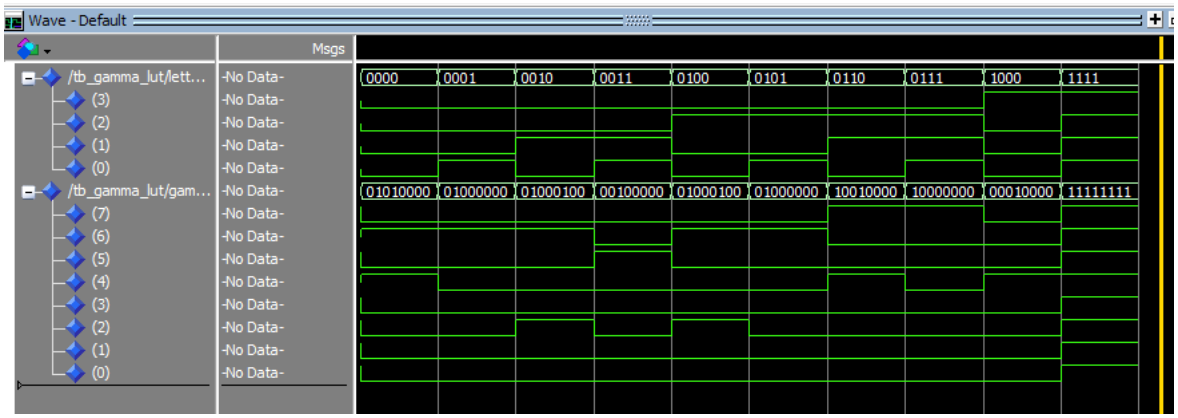
For å teste delmodulene har jeg utviklet egne testbencher. Disse simulerer inngangssignaler og overvåker utgangene for å sikret korrekt reaksjon. Noen av testbenchene er gjenbrukt og tilpasset fra tidligere laboppgaver, noe som har effektivisert utviklingsprosessen.

4.1 Gamma lookuptable

I denne testbenchen simuleres gamma_lut modulen ved å sende alle gyldige 4-bit verdier via letter_sel. I wavewormen som generes av Medelsim observerer vi at gamma_code gir riktig 8-bit kode for hver input. Tilslutt tester vi en ugyldig verdi «1111» og observerer det blir håndtert riktig. I waveformen under observerer vi at den 8-bit koden som genereres samsvarer med verdiene i gamma oppslagstabellen.

```
library ieee;
use ieee.std_logic_1164.all;

entity tb_gamma_lut is
end tb_gamma_lut;

architecture sim of tb_gamma_lut is

    component gamma_lut is
        port (
            letter_sel : in  std_logic_vector(3 downto 0);
            gamma_code : out std_logic_vector(7 downto 0)
        );
    end component;

    signal letter_sel : std_logic_vector(3 downto 0);
    signal gamma_code : std_logic_vector(7 downto 0);

begin

    uut: gamma_lut
        port map (
            letter_sel => letter_sel,
            gamma_code => gamma_code
        );

    stim_proc: process
    begin
        -- Test each valid symbol
        letter_sel <= "0000"; wait for 20 ns; -- P
        letter_sel <= "0001"; wait for 20 ns; -- B
        letter_sel <= "0010"; wait for 20 ns; -- 1
        letter_sel <= "0011"; wait for 20 ns; -- 8
        letter_sel <= "0100"; wait for 20 ns; -- 0
        letter_sel <= "0101"; wait for 20 ns; -- -
```

```
        letter_sel <= "0110"; wait for 20 ns; -- F
        letter_sel <= "0111"; wait for 20 ns; -- G
        letter_sel <= "1000"; wait for 20 ns; -- A
        letter_sel <= "1111"; wait for 20 ns; -- invalid

        wait;
    end process;

end architecture;
```



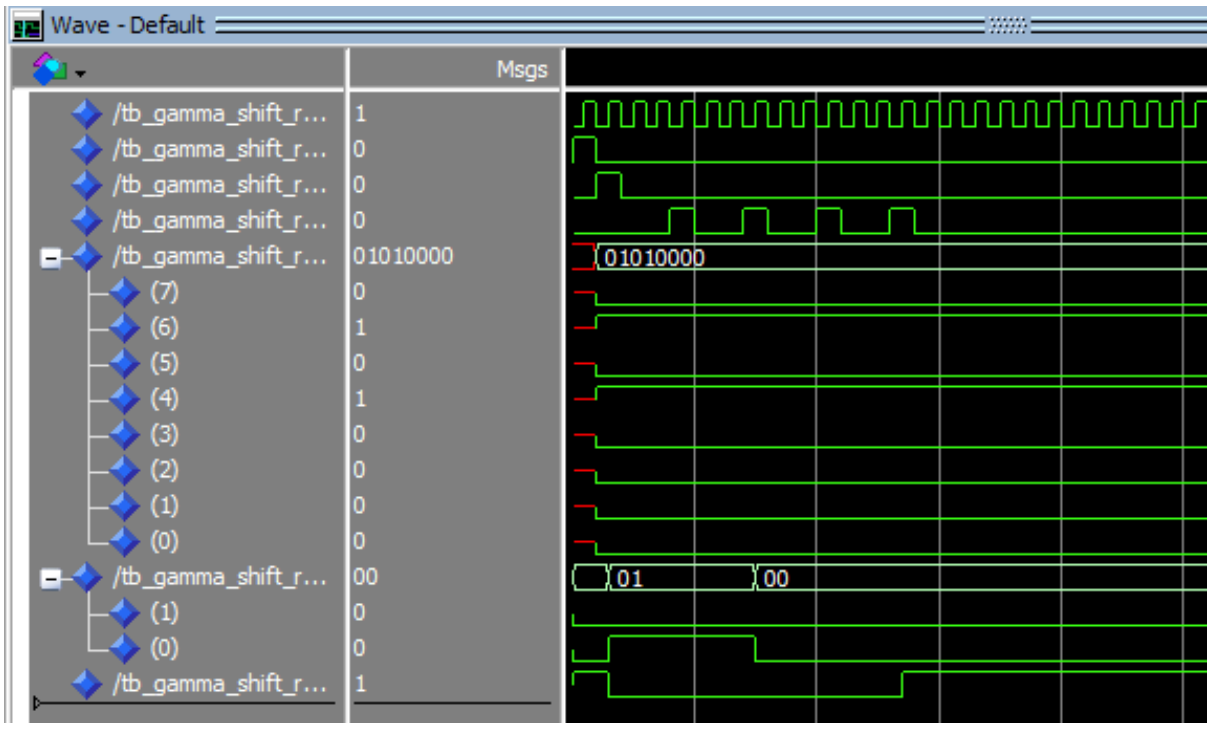
Figur 9 Waveform Gamma lookuptable

4.2 Gamma shift register

I denne testbenchen verifiseres funksjonaliteten til gamma_shift_reg modulen. Testbenchen simulerer lasting av 8-bit gamma kode og deretter forskyvning av to og to bit ved hjelp av shift_en signalet. I waveformen under ser vi at ved hver positive klokke flanke og aktive shift_en, forskyves registeret og legger inn «00» i bunnen. Under simuleringen skal code_out observes og kontrolleres at riktig 2-bit sekvenser sendes ut til FSM-en. Finished signalet settes i tillegg høyt etter fire skift, noe som bekrefter at all data er sendt.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity tb_gamma_shift_reg is
end entity;

architecture sim of tb_gamma_shift_reg is
    signal clk       : std_logic := '0';
    signal reset     : std_logic := '0';
    signal load      : std_logic := '0';
    signal shift_en  : std_logic := '0';
    signal code_in   : std_logic_vector(7 downto 0);
    signal code_out  : std_logic_vector(1 downto 0);
```

```
    signal finished  : std_logic;

    constant clk_period : time := 20 ns;

begin

    uut: entity work.gamma_shift_reg
        port map (
            clk       => clk,
            reset     => reset,
            load      => load,
            shift_en  => shift_en,
            code_in   => code_in,
            code_out  => code_out,
            finished  => finished
        );


    clk_process : process
    begin
        while now < 1000 ns loop
            clk <= '0';
            wait for clk_period / 2;
            clk <= '1';
            wait for clk_period / 2;
        end loop;
        wait;
    end process;


    stim_proc: process
    begin
        -- Reset
        reset <= '1';
        wait for clk_period;
        reset <= '0';

        -- Load a code value (e.g., P: dash dash dot = 01010000)
        code_in <= "01010000";
        load <= '1';
        wait for clk_period;
        load <= '0';

        wait for clk_period * 2;

        -- Perform 4 shifts
        for i in 0 to 3 loop
            shift_en <= '1';
            wait for clk_period;
            shift_en <= '0';
            wait for clk_period * 2;
        end loop;

        wait;
    end process;

end architecture;
```



Figur 10 Waveform Gamma shift register

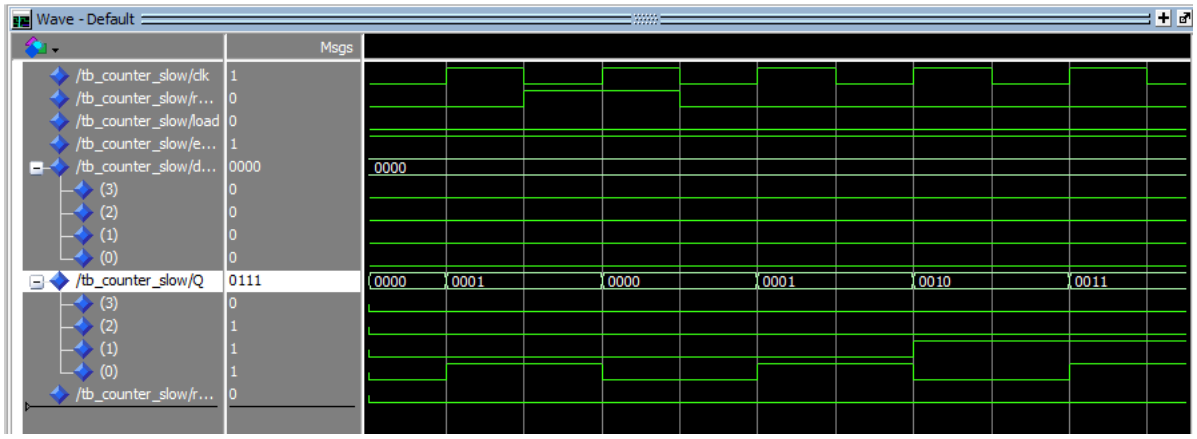
4.3 Quarter second counter

I denne testbenchen testes counter_slow module som er en generell klokketeller designet for å generere et rollover signal etter å ha telt til et spesifisert verdi k.Testbenchen simulerer klokkepulser og aktiverer enable signalert for å kontrollere at telleren øker korrekt. Når telleren når verdien k -1, forventes at rollover settes til høyt i en klokkesyklus og at telleren nullstilles. I wavecharten under ser vi at rollover signalet utløses presist i linje 2 og at Q øker sekvensielt.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity tb_counter_slow is
end entity;

architecture sim of tb_counter_slow is

    constant n : natural := 4;         -- Lav verdi for rask simulering
    constant k : integer := 10;

    component counter_slow is
        generic (
            n : natural := 4;
            k : integer := 10
        );
```

```
        port (
            CLK      : in  std_logic;
            reset    : in  std_logic;
            load     : in  std_logic;
            enable   : in  std_logic;
            data     : in  std_logic_vector(n-1 downto 0);
            Q        : out std_logic_vector(n-1 downto 0);
            rollover : out std_logic
        );
    end component;

    signal clk      : std_logic := '0';
    signal reset    : std_logic := '0';
    signal load     : std_logic := '0';
    signal enable   : std_logic := '1';
    signal data     : std_logic_vector(n-1 downto 0) := (others => '0');
    signal Q        : std_logic_vector(n-1 downto 0);
    signal rollover : std_logic;

    -- clock generation
    constant clk_period : time := 10 ns;
begin

    clk_process : process
    begin
        while now < 200 ns loop
            clk <= '0';
            wait for clk_period / 2;
            clk <= '1';
            wait for clk_period / 2;
        end loop;
        wait;
    end process;

    uut: counter_slow
        generic map (
            n => n,
            k => k
        )
        port map (
            CLK      => clk,
            reset    => reset,
            load     => load,
            enable   => enable,
            data     => data,
            Q        => Q,
            rollover => rollover
        );

    stimulus : process
    begin
        wait for 10 ns;
        reset <= '1';
        wait for clk_period;
        reset <= '0';

        wait for 150 ns;

        -- test done
```

```
        wait;
    end process;

end architecture;
```



Figur 11 Waveform Quarter second counter

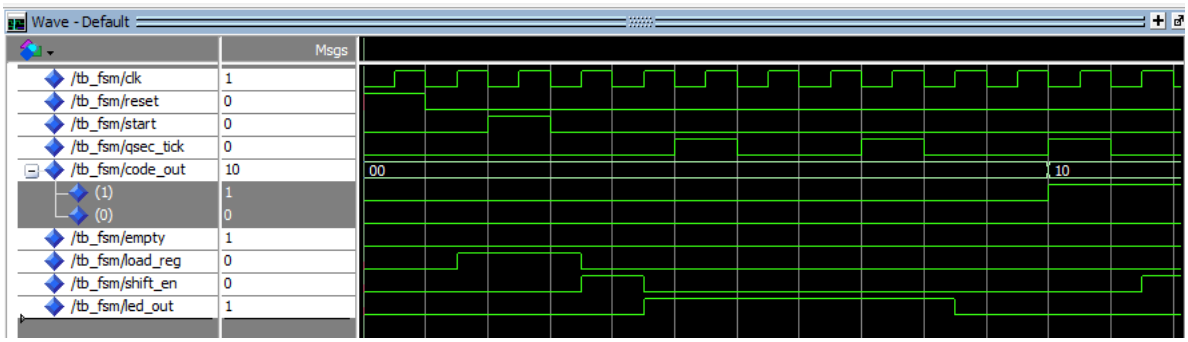
4.4 FSM

I denne testbenchen tester vi tilstandsmaskinen (FSM) for korrekt sekvensiell oppførsl. Ved å simulere kontrollsignaler som start, reset og qsec_tick observerer hvordan FSM beverseg gjennom de ulike tilstandene. I wavecharten sjekker vi at LED lyser i riktig varighet ut i fra code_out, signalene shift_en og load_reg aktiveres i riktig state. Tilslutt sjekker vi at FSM går i DONE state etter at alle bits er sendt. Dette vertifiseres ved at signalet emty blir høy.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity tb_FSM is
end entity;

architecture sim of tb_FSM is

    component FSM is
        port (
            clk       : in  std_logic;
            reset     : in  std_logic;
            start     : in  std_logic;
            qsec_tick : in  std_logic;
            code_out  : in  std_logic_vector(1 downto 0);
            empty     : in  std_logic;
            load_reg  : out std_logic;
            shift_en  : out std_logic;
            led_out   : out std_logic
        );
```

```
    end component;

    signal clk       : std_logic := '0';
    signal reset     : std_logic := '0';
    signal start     : std_logic := '0';
    signal qsec_tick : std_logic := '0';
    signal code_out  : std_logic_vector(1 downto 0) := "00";
    signal empty     : std_logic := '0';
    signal load_reg  : std_logic;
    signal shift_en  : std_logic;
    signal led_out   : std_logic;

    constant clk_period : time := 20 ns;

begin

    uut: FSM
        port map (
            clk       => clk,
            reset     => reset,
            start     => start,
            qsec_tick => qsec_tick,
            code_out  => code_out,
            empty     => empty,
            load_reg  => load_reg,
            shift_en  => shift_en,
            led_out   => led_out
        );

    -- Clock generation
    clk_process : process
    begin
        while now < 1000 ns loop
            clk <= '0';
            wait for clk_period / 2;
            clk <= '1';
            wait for clk_period / 2;
        end loop;
        wait;
    end process;

    -- Stimulus
    stim_proc: process
    begin
        -- Reset
        reset <= '1'; wait for clk_period;
        reset <= '0'; wait for clk_period;

        -- Start signal
        start <= '1'; wait for clk_period;
        start <= '0'; wait for clk_period * 2;

        -- Simuler kode for "dot" og flere qsec_tick
        code_out <= "00";
        empty <= '0';

        qsec_tick <= '1'; wait for clk_period;
        qsec_tick <= '0'; wait for clk_period * 2;
```

```
        qsec_tick <= '1'; wait for clk_period;
        qsec_tick <= '0'; wait for clk_period * 2;

        -- Neste symbol
        code_out <= "10";

        qsec_tick <= '1'; wait for clk_period;
        qsec_tick <= '0'; wait for clk_period * 2;

        -- Ferdig
        empty <= '1';
        qsec_tick <= '1'; wait for clk_period;
        qsec_tick <= '0'; wait;

    end process;

end architecture;
```



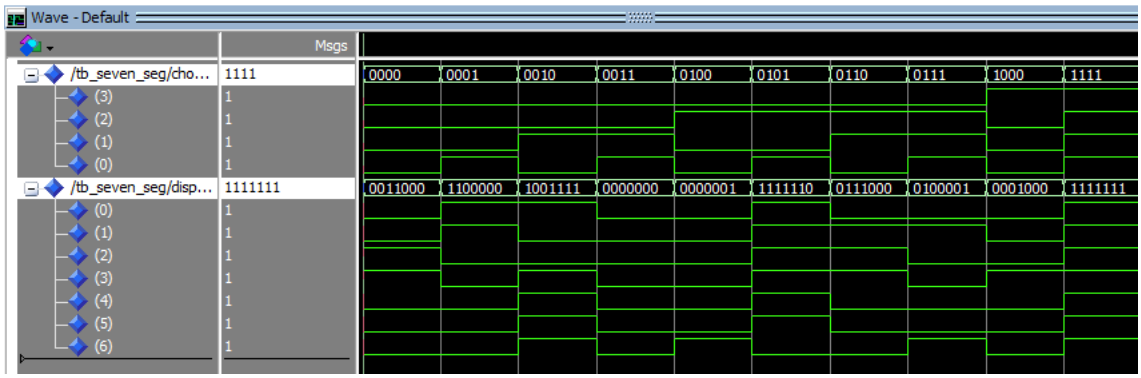
Figur 12 Waveform FSM

4.5 Seven segment display

Denne testbenchen vertifiserer vi at seven_seg modulen oversetter et 4-bit binært symbol til riktig 7-segment kode. Hver gyldig symbol testes i tilegg til ugyldig verdig. Wavecharten under bekrefter at riktig segmentmønster for våre symboler vises. I tillegg viser den at ugyldig verdi «1111» vises som «1111111» som slår av alle segmentene.

```
library ieee;
use ieee.std_logic_1164.all;

entity tb_seven_seg is
end tb_seven_seg;

architecture sim of tb_seven_seg is

    component seven_seg is
        port (
            chosen_symbol : in  std_logic_vector(3 downto 0);
```

```
            display       : out std_logic_vector(0 to 6)
        );
    end component;

    signal chosen_symbol : std_logic_vector(3 downto 0);
    signal display       : std_logic_vector(0 to 6);

begin

    uut: seven_seg
        port map (
            chosen_symbol => chosen_symbol,
            display       => display
        );

    stim_proc: process
    begin
        -- Test valid symbols
        chosen_symbol <= "0000"; wait for 20 ns; -- P
        chosen_symbol <= "0001"; wait for 20 ns; -- B
        chosen_symbol <= "0010"; wait for 20 ns; -- 1
        chosen_symbol <= "0011"; wait for 20 ns; -- 8
        chosen_symbol <= "0100"; wait for 20 ns; -- 0
        chosen_symbol <= "0101"; wait for 20 ns; -- -
        chosen_symbol <= "0110"; wait for 20 ns; -- F
        chosen_symbol <= "0111"; wait for 20 ns; -- G
        chosen_symbol <= "1000"; wait for 20 ns; -- A
        chosen_symbol <= "1111"; wait for 20 ns; -- invalid

        wait;
    end process;

end architecture;
```



Figur 13 Waveform 7-segment display

5.0 Resultat og diskusjon

Prosjektets mål var å implementere en fungerende gamma-kode-encoder på FPGA, der input fra bryter og knapper styrer et blinkemønster som vises på LEDR, og valgt symbol vises på 7-segmentdisplayet. Systemet består av flere delmoduler, og en FSM som styrer logikken.

Systemet fungere som forventet. Når KEY_0 trykkes blinker $LEDR_0$ i mønster som tilsvarer valgt gamma symbol. Gamma kodene genereres korrekt fra gamma_lut, og shift_reg sender ut 2-bit segmenter til FSM. FSM tolker koden riktig og styrer LED blink i henhold til lengde (kort, medium og lang). 7-segment displayet viser korrekt bokstav, tall, og regn som tilhører SW verdiene. Counter_slow generer tick_qsec hver 0.25 sekund og gir rytme til blinkemønsteret.

5.1 Utfordringer og debugging

Prosjektet med i utgangspunktet utviklet i Quartus, fordi mange av delmodulene slikt som counter_slow, seven_seg, shift_register allerede var laget i tidligere laboppgaver. Dette sparte noe tid på gjenbruk, men førte også til flere runder med feilsøking og debugging som kunne vært unngått med tidlig simulering i modelsim med testbencher.

Noen av utfordringene jeg møtte på inskluderer:

- **Tick_qsec virket ikke:** klokken ble først implementert med en annen form med harkoding quarter_sec_counter istedenfor den gemeriske klokken som slutt prosjektet inneholder. Den tidligere klokken ga ingen synlig puls. Etter mange forsøk med debugging valgt jeg å erstatte den men en tidligere delmodul som jeg har utviklet i en tidligere laboppgave. Denne versjonen ga en stabil tikkerpuls etter grundig test og justering av n og k.
- **FSM startet ikke som forventet:** prosjektet startet med flere tilstander, noe som skapte uønsked venting og dobbeltrykk på KEY_1 før $LERD_0$ begynte å blinke. Løsningen ble å gjerne vente tilstanden og starte direkte på S1_load.
- **$LEDR_0$ blinket ikke først:** FSM måtte omstruktureres slik at LED skrus på og av basert på qsec_tick og riktig blinkvarighet. Flere forsøk ble gjort for å synkronisere med shift_reg.
- **Debugging med LEDR:** $LEDR_{7-0}$ ble brukt aktivt for å vise interne signaler som tick_qsec, shift_en, load_reg, finished, noe som gjorde lette å forstå FSM-ens atferd og isolere hvor probleme befant seg.

Når testbencher senere ble laget i ModelSim, ble feil lettere identifisere og forstå. Simuleringen ga tydelig innsyn i bølgediagrammer og gjorde det mulig å validere submodulene isolert før de ble koblet sammen. Om jeg hadde startet med testbencher som anbefalt ville jeg spart mye tid debugging.

Til senere prosjekter er det tre erfaringer jeg vil ta med meg fra dette prosjektet.

- **Start i ModelSim:** Det er eksrem viktig å starte med testbencher og simulere alle delmodulene før enn går til Quartus og fysisk programerer. Dette gjør det lettere å isolere feil og forstår kompleks oppførsel. Kompilering i ModelSim er også raskere.
- **Bruk tydelig logikk for FSM :** Fra start kunne jeg unngått å bruke så mange unødvendige tilstander. Systemet kunne starter på S1_LOAD ettersom den initieres med et tasketrykk.
- **Visuell debug signaler:** Forsete å bruke LEDR aktivt til å feil søke. Det gir rask og intuitiv tilbakemelding fra FPGA.

6.0 Konklusjon

I dette prosjektet har jeg utviklet en fungerende gamma-kode-encoder på FPGA, styrt av en Finite State Machine(FSM). Systemet tolker inngang fra brytere og knapper, og blinker ut et mønster på LED som representerer en tilhørende gamma koden. Prosjektet kombinerer flere VHDL komponenter, inkludert klokketeller, skiftregister og lookuo table, som alle ble koblet sammen i en strukturert hoved modul.

Etter flere runder med feilsøking og testing fungerer systemet som ønsket. $LEDR_0$ blinker i riktig i tiktig mønster for valgt symbol, og både FSM og delkomponentene reagerer korrekt på signalet fra bruker og klokke.

Jeg har fått dypere innsikt i hvordan VHDL kan brukes til å modellere og simulere digitale systemer, spesielt hvordan en FSM koordinerer komplekse prosesser. Jeg har også lært viktigheten av å simulere i ModelSim før implementering på FPGA. Å starte rett i Quartus uten testbencher førte til mye tid brukt på debugging.

Til framtidige prosjekter vil jeg prioritere simulering tidlig i utviklingen og lage testbencher for alle delmoduler før integrasjon. Det gir bedre kontroll og forståelse gjennom hele prosessen. Systemet kan videreutvikles ved å støtte flere symboler, lydutgang, eller visuell representasjon på display.

Litteraturliste

[1] D. *M. Harris and S. L. harris, Digital Deign and Computer Architecture: RISC-V Edition*, 1st ed. Cambridge, MA, USA: Morgan Kaufmann, 2022.

[2] B. Cohen, *Free Range VHDL: No Strings Attached!*, 2nd ed. Los Alamitos, CA, USA: VhdlCohen Publishing, 2013.