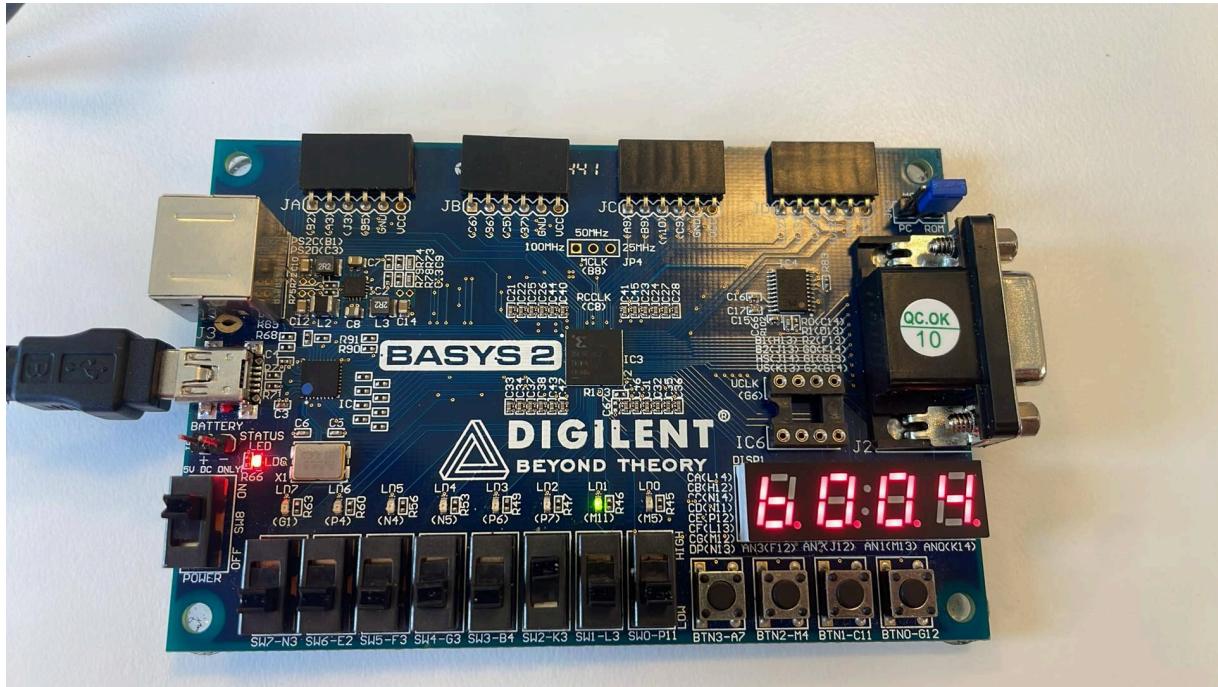


Mads Vølkers Rudolph, Andreas Skåning Jacobsen, Sigurd Hestbech Christiansen, Jonas Beck Jensen

30081 Digital engineering

Lommeregner



A head-and-shoulders portrait of a young man with short, light brown hair. He has a neutral expression and is looking directly at the camera. He is wearing a dark-colored, possibly black, t-shirt. The background is plain and light-colored.

Sigurd

Andreas



Jonas



Mads

Indholdsfortegnelse

Indledning.....	2
Kravliste.....	2
Calc_Menu.....	4
Tilstandsmaskine.....	4
Funktionalitet.....	6
Calc_Data.....	8
Div_component.....	9
Simulering af CalcTop.....	11
Konklusion.....	12
Bilag.....	15
Bilag 1. Divisions flowchart.....	15
Bilag 2. Simulering af Calc_TOP.....	16
Bilag 3. Rækkefølge af tilstade.....	16
Bilag 4. Koden.....	17
Calc_Top.....	17
Calc_Menu.....	20
Calc_Data.....	26
Div_Component.....	29
TestBench.....	32

Indledning

Dette er en rapport over vores afsluttende projekt i Digital Engineering 30081, i projektet har vi programmeret en lommeregner i VHDL.

Lommeregneren består som en helhed af 6 underkomponenter, hvoraf vi selv har designet og programmeret 2 af dem. De andre 4 underkomponenter er vi blevet tildelt, og har derfor ikke selv programmeret dem. De to komponenter som vi selv har lavet er Calc_menu lavet af Mads Rudolph & Jonas Jensen, og Calc_Data lavet af Andreas Jacobsen & Sigurd Christiansen. Der vil i rapporten være et afsnit til hver af disse komponenter med diagrammer, forklaringer og simuleringer, så du forhåbentlig kan få en god forståelse af den lommeregner som vi har lavet, håber du nyder rapporten!

Kravliste

I dette kapitel har vi lavet en kravliste som indeholder alle de krav, som er blevet stillet til vores projekt i opgaveformuleringen.

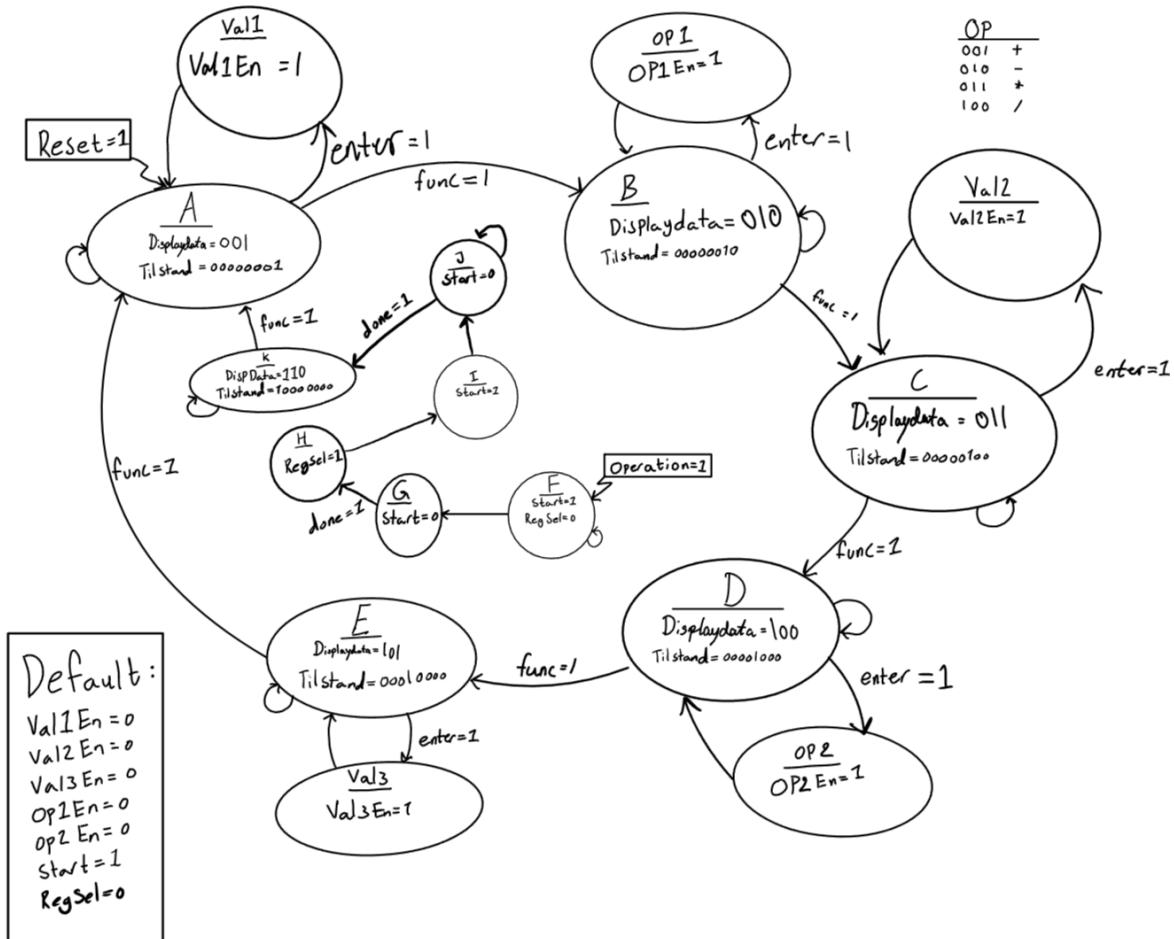
Funktionelle krav:	Kravbeskrivelse:
Krav 1:	Systemet skal kunne udføre følgende aritmetiske operationer: addition (+) , subtraktion (-) , multiplikation (*) og division (/)
Krav 2:	Systemet skal udføre beregningen i følgende rækkefølge: Resultat = (Input A Op1 Input B) Op2 Input C
Krav 3:	Systemet skal modtage tre 8-bit inputværdier (Input A, B, og C) via switche (SW) og trykknapper (BTN) på FPGA-boardet
Krav 4:	Systemet skal modtage to aritmetiske operatorer (Op1 og Op2) via switche og knapper
Krav 5:	Systemet skal benytte de fire trykknapper (BTN0–BTN3) til styring af inputfunktionalitet

Hardware krav:	Kravbeskrivelse:
Krav 1:	Systemet skal implementeres på et Basys2 FPGA-board
Krav 2:	Systemet skal benytte switche (SW0-SW7) til at angive 8-bit værdier i HEX
Krav 3:	Systemet skal bruge trykknapperne (BTN0-BTN3) til inputkontrol
Krav 4:	Resultat skal kunne vises på 7-segment display
Krav 5:	Systemet skal kunne vise hvilket state den befinner sig igennem LED'erne på boardet (LD0-LD7)

Brugerinput:	Knap:	Funktion:
	BTN 3:	Reset – nulstiller alle registre og tilstande
	BTN2:	Operation – starter den aritmetiske beregning
	BTN 1:	Func – bruges til at skifte mellem de forskellige tilstænde
	BTN 0:	Enter – bekræfter indtastet værdi (Val1, Val2, Val3, Op1 og Op2)

Calc_Menu

Tilstandsmaskine



Tilstandsdiagram for Calc_Menu

Tilstandsmaskinen styrer sekventielt fem registre: Val1, Val2, Val3, Op1 og Op2. Formålet er at tillade løbende opdatering af værdier og operationer via brugerinput (I realiteten er der yderligere 5 tilstande, Val1, Val2, Val3, Op1 og Op2, men da de kun bliver brugt til at enable deres respektive registre, vil vi ikke yderligere uddybe dem).

Brugeren vælger input via en switch, og registrene aktiveres én ad gangen med enter, mens func-knappen bruges til at skifte mellem registre i en fast sekvens (A → B → C → D → E → A). Dette loop tillader kontinuerlig redigering, indtil operation = 1, hvilket uanset aktuel tilstand fører til tilstand F.

I F sættes Start = 1, hvilket sender data og operationer til beregningsmodulet (Calc_Data) via In1, In2 og Opcode.

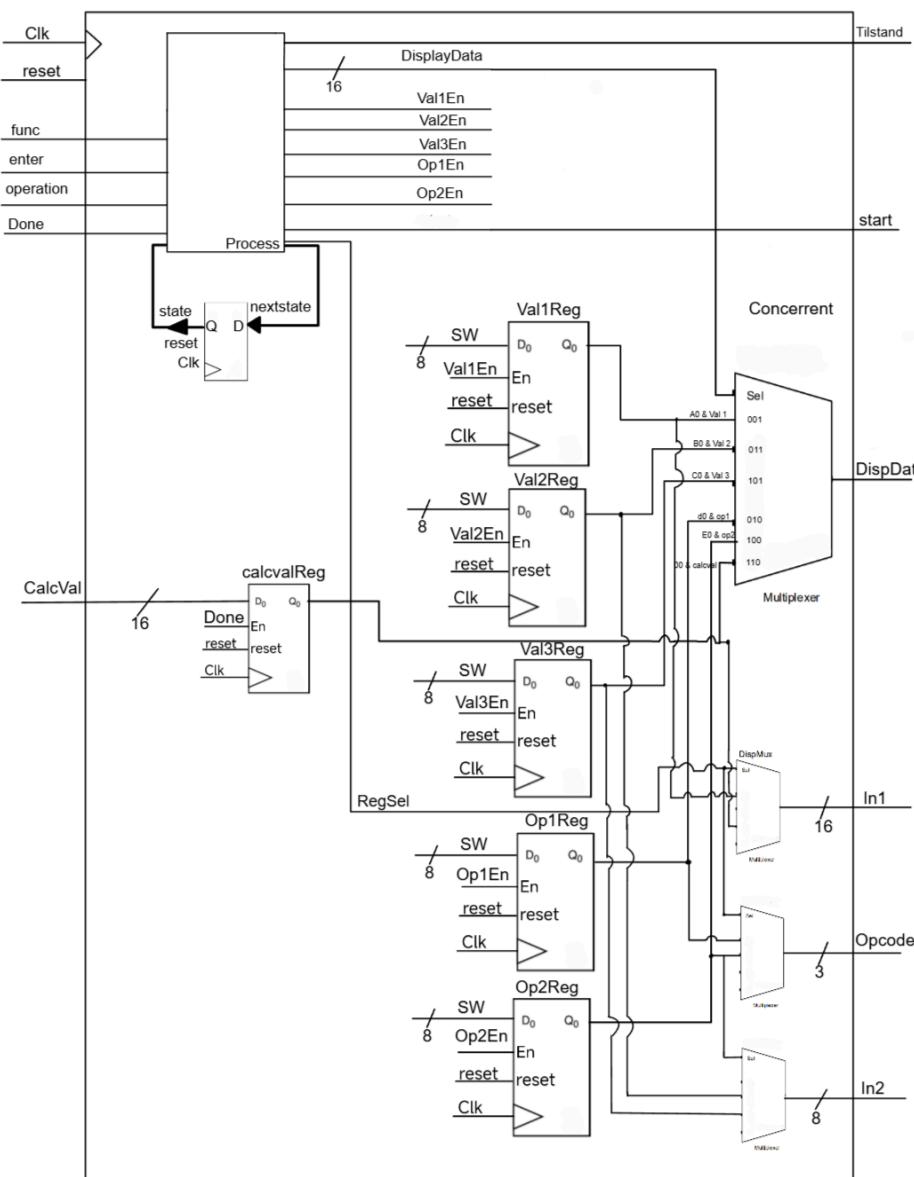
For at sikre at Start-signalen opfattes som en puls, benyttes tilstand G, hvor signalet nulstilles straks efter.

H bruges til at sætte RegSel = 1, hvilket aktiverer outputveje gennem multiplexere (MUX). For at sikre at registrene opdateres før beregning påbegyndes, er **I** indført som buffer-tilstand før J, hvor Start igen sættes til 1 og calc-modulet starter anden beregning med den nye beregnede værdi og den nye opcode.

Når Done = 1, går vi til tilstand K, hvor resultatet vises på displayet. Her forbliver systemet, indtil brugeren trykker func for at starte en ny sekvens. Reset returnerer altid systemet til starttilstand A.

Systemets default-tilstand sætter alle enable-signaler og kontrolsignaler lavt (= 0), indtil en brugerinteraktion aktiverer flowet.

Funktionalitet



Funktionsdiagram for Calc_Menu

Funktionsdiagrammet illustrerer en digital enhed opbygget omkring en sekventiel styringsstruktur, der styrer seks registre og en multiplexerbaseret datapath. Systemet styres via signaler som reset, Clk, func, enter, operation og start, der aktiverer forskellige kontrollsinaler ud fra systemets aktuelle tilstand og inputs.

Kernen i systemet er Process-blokken, som håndterer tilstandsstyring. Den genererer aktiveringssignaler (Val1En, Val2En, osv.) til de tilkoblede registre og opdaterer datapathen i takt med clock-signalen. En tilstandslagring (flip-flop) holder styr på systemets aktuelle state og næste state.

Datapathen består af:

- 3 værdiregistre (Val1Reg, Val2Reg, Val3Reg)
- 2 operandregistre (Op1Reg, Op3Reg)
- 1 outputregister til beregnet værdi (calcvalReg)
- 4 multiplexere til at vælge mellem de forskellige værdier der skal sendes til Calc_data, og 7seg_disp via henholdsvis, DispData, In1,In2 og Opcode

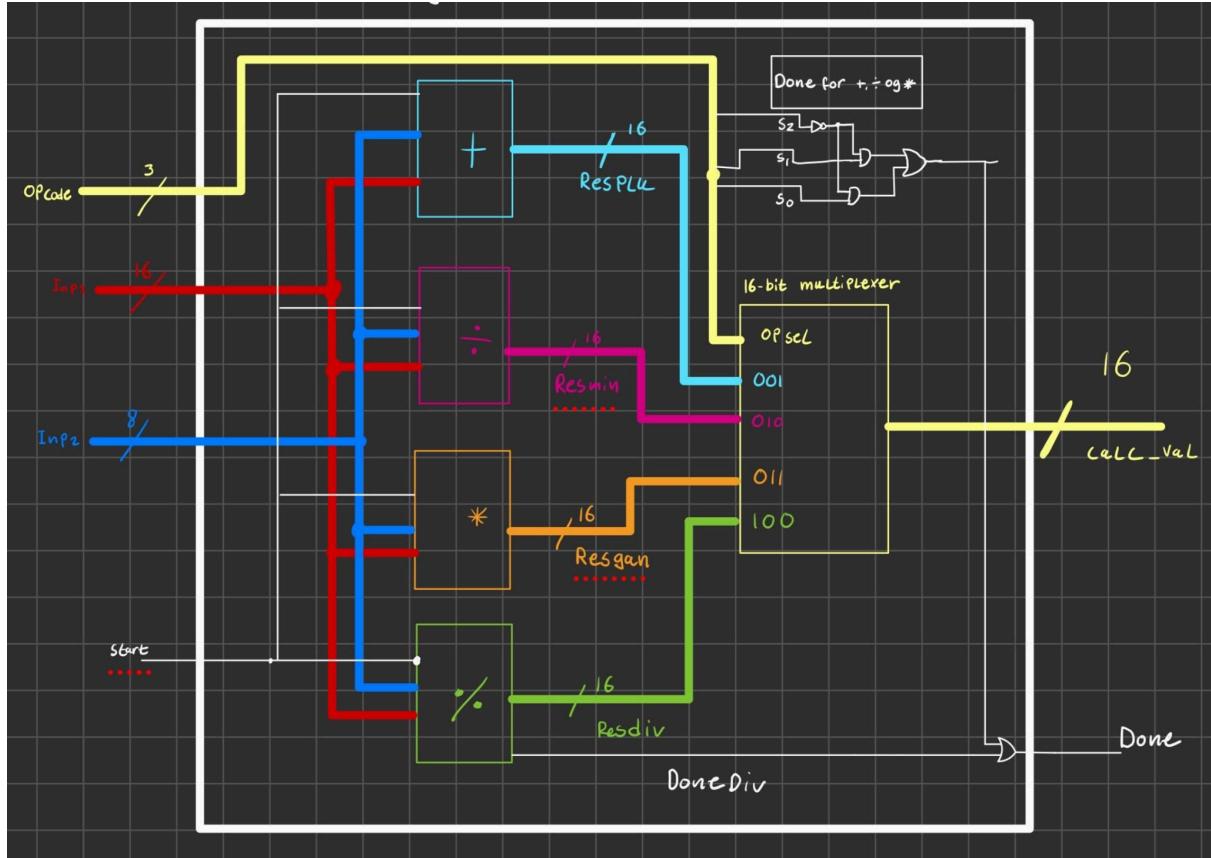
Alle registre kan nulstilles og aktiveres individuelt via deres respektive En-signaler. Dataflowet styres af en multiplexer, som vælger hvilke registre der skal sende data videre afhængigt af signalet RegSel. Multiplexeren har værdierne fra Val1, Val2 og Op1, når RegSel = 0, og værdierne for Val3, CalcVal og Op3, når RegSel = 1. En særskilt DispMux sammensætter visningsdata (DisplayData) til output baseret på valgte inputs.

I tilstand A sender DispMux data fra Val1 registret, og i Tilstand B sender den data fra Op1 registret videre til 7Seg_disp. Det samme sker i de næste tilstande på nær regne tilstandene F→G→H→I. I tilstand K sendes data fra CalcValReg, der indeholder den endelig regnede værdi.

Vi har klistret en Hex værdi foran hver værdi i Disp mux, så der på 7seg_disp vises tilstanden og register værdien. Tilstandene følger rækkefølgen, der fremgår i bilag 3.

```
with DispSel select
    DispData <=
        (others => '0')           when "0000000",
        X"A0" & Val1S            when "0000001",
        X"B0" & Op1s             when "0000010",
        X"C0" & Val2S            when "0000011",
        X"d0" & Op2S             when "0000100",
        X"E0" & Val3s             when "0000101",
        CalcVals                  when "0000110",
        (others => '0')           when others;
```

Calc_Data



Blokdiagram for Calc_Data

Calc_data er den komponent, hvis opgave er at udføre alle de aritmetiske regneoperationer på de input, som bliver modtaget af Calc_menu. Addition, subtraktion og multiplikation operationerne kan realiseres direkte ved de inbyggede operationer i FPGA'en, disse tre operationer udføres parallelt, og resultatet vælges via en multiplexer styret af en op-code.

Da addition, subtraktion og multiplikation kan udføres på en enkelt clock-cyklus, sendes et 'done'-signal umiddelbart efter, at 'start'-signalet er modtaget. Done signalet indikerer at den værdi der er på Calc_Val er det endelige resultat. Done signalet for divisionen opfører sig lidt anderledes og kan ses i næste afsnit under Div_component.

Med de indbyggede operatorer kan de første tre regne operationer skrives med ganske få linjer kode som ses nedenfor:

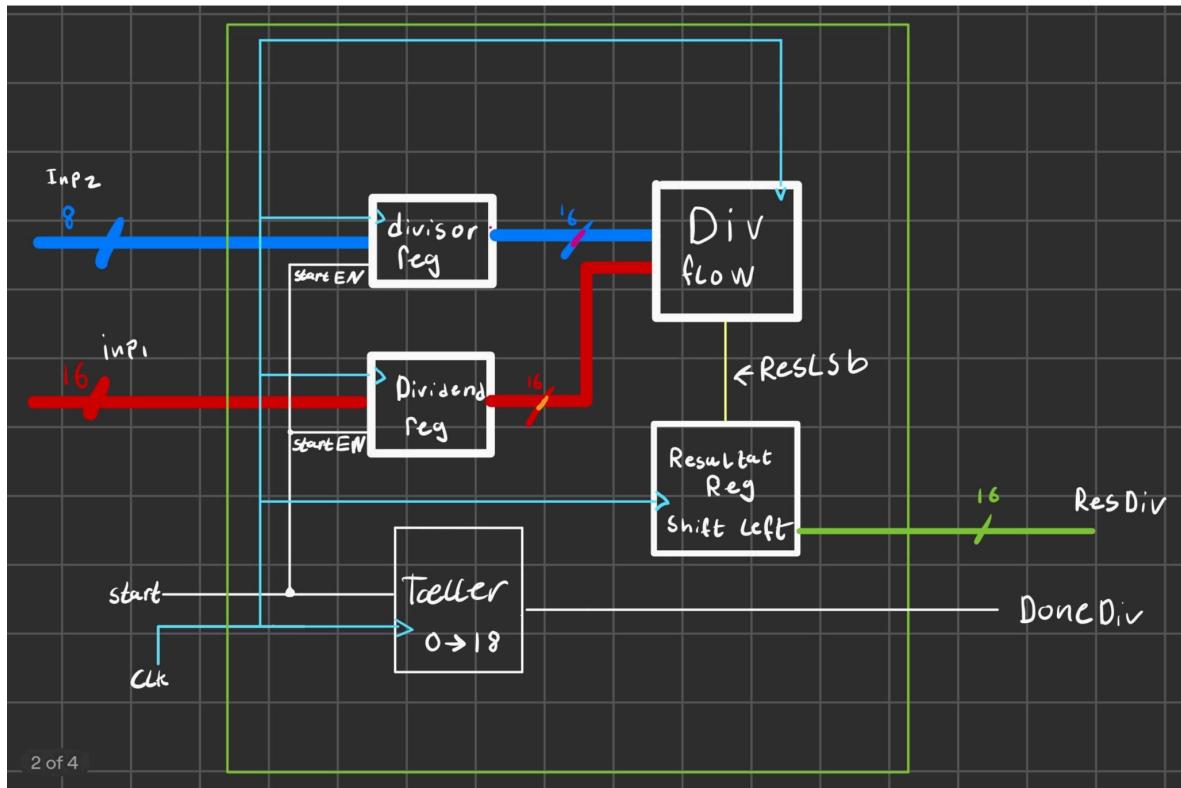
```

process(Clk)
begin
if rising_edge(Clk) then
  if start = '1' then
    ResPlu <= std_logic_vector(unsigned(Inp1) + unsigned("00000000" & Inp2));
    ResMin <= std_logic_vector(unsigned(Inp1) - unsigned("00000000" & Inp2));
    ResGan <= std_logic_vector(resize(unsigned(Inp1) * unsigned("00000000" & Inp2), 16));
  end if;
end if;
end process;

```

Div_component

Divisionen er noget mere kompliceret, vi har lavet en hel under komponent kaldet Div_component, hvis opgave kun er at udføre sekventiel binær division på et 16bit binært tal. **blokdiagrammet for denne ses nedenfor:**



Funktionsdiagram for Div_component

Div_component indeholder 4 registre; Divisor register, dividend register, tæller register og et resultat register.

dividend og **divisor** registrene holder på input værdierne, og fører dem videre over i divisionsblokken. Samtidig udvider disse også dividenden med 16 bits på venstre side og 16 bits på højre side af divisoren, som er påkrævet for at vi kan lave den sekventielle binære division.

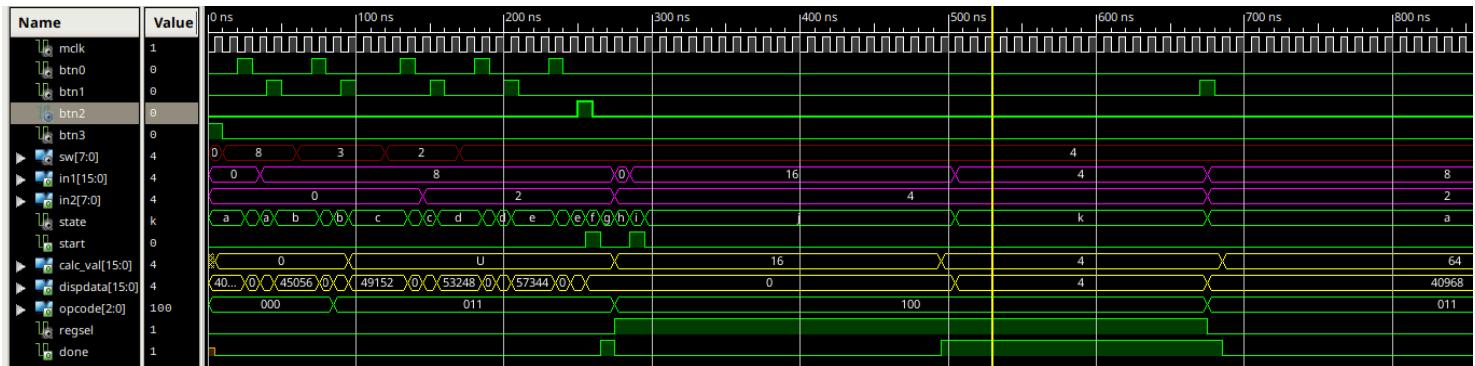
Tæller Registeret tæller fra 1 til 18. Af disse bruges tællingerne 2 til 17 (i alt 16 tællinger) til selve divisionsprocessen. Tælling 1 anvendes til at loade Dividend og Divisor registrene med input værdier, og tælling 18 bruges til at loade sidste bit ind i resultatregistret. Ydermere genererer tæller registeret sit eget done signal, (**doneDiv**) som i modsætning til addition, subtraktion og multiplikation operationernes done signal, først sendes efter der er gået 18 clock cykluser for at sikre at divisionen er færdig og at calc_Val har den rigtige værdi.

Div Flow blokken, som ses øverst til højre i blokdiagrammet ovenfor, laver operationer som følger et flowchart, som blev oplagt for vores hold på femte undervisningsgang af Digital Engineering 30088 kurset. Flowchartet kan ses i **bilag 1** og illustrerer hvordan den sekventielle division fungerer.

Div Flow fungerer i grove træk ved at dividenden er udvidet med 16 nuller på venstre side og divisoren er udvidet med 16 nuller på højre side. Ydermere vil divisoren rykkes en plads mod højre for hver clock, samtidig sammenlignes dividend med divisor. Hvis dividenden er større end eller lig med divisoren, sættes ResLSB til '1' og 'dividend - divisor' udføres hvorefter resultatet gemmes i dividend registret. Hvis dividenden er mindre end divisoren, sættes ResLSB til '0'. Denne proces gentages 16 gange indtil divisoren har rykket tilbage til sin originale størrelse som før bit udvidelsen.

Resultat registret opdateres ved hvert clock-signal med en enkelt bit fra **ResLSB**, som er outputtet fra **Div_Flow** blokken. Dette bit skubbes ind i resultatregistret ved et venstreskift. Når tællingen når 18 er det sidste bit blevet skiftet ind i resultatregisteret og kommer ud af registeret som ResDiv, som føres videre i Calc_data og gennem multiplexerne som Calc_Val.

Simulering af CalcTop



Simuleringen er vedhæftet i fuld størrelse

I vores simulering udføres regnestykket $(8 \times 2)/4$. Det ses tydeligt, at hvert register bliver indstillet korrekt ved hjælp af SW og Enter-knappen (BTN0). Efter hver indstilling skiftes der til næste tilstand med func-knappen (BTN1).

Når alle registrene har fået de rigtige værdier, trykkes der på Operation-knappen (BTN2). Det sender tilstandsmaskinen videre til tilstand F, hvor start-signalet sættes høj, og det fortæller Calc_Data, at den nu skal gå i gang med beregningen.

Herefter kører tilstandsmaskinen igennem regnetilstandene, indtil den når til tilstand G. I denne tilstand venter den på, at done-signalet bliver sat til 1, før den går videre til H. Der går kun én clock-cyklus, da multiplikationen kun tager én cyklus at udføre.

I tilstand H bliver RegSel sat høj, hvilket opdaterer alle registre. FSM'en fortsætter så igennem regnetilstandene igen, til den når til tilstand J. Her venter den endnu en gang på, at done bliver sat høj, og så skifter den til tilstand K – sluttilstanden, hvor resultatet vises, indtil der igen trykkes på func.

Det endelige resultat er 4 – som forventet.

Det er værd at bemærke, at debounce-modulet er udkommenteret i denne simulering, da det gav store timingproblemer i simuleringen. Dette problem opstår ikke på det fysiske board, hvor debounce fungerer som forventet.

Konklusion

Projektets mål var at udvikle en funktionsdygtig aritmetisk lommeregner implementeret i VHDL på Basys2 FPGA-platformen. Systemet er blevet opdelt i klart definerede underkomponenter, hvor Calc_Menu og Calc_Data blev designet og programmeret fra bunden af gruppen selv. Gennem en struktureret Finite State Machine og en præcis datapath-arkitektur er det lykkedes at opfylde samtlige krav til funktionalitet og brugerinteraktion.

Lommeregneren understøtter de fire grundlæggende operationer (addition, subtraktion, multiplikation og division) samt kompleks rækkefølgebehandling af tre inputværdier og to operatorer. Divisionen blev realiseret gennem en specialdesignet komponent, som anvender sekventiel binær division.

Testbench-simuleringer samt funktionstest på hardware bekræfter, at systemet opfører sig som forventet, og alle tilstade samtlige resultater vises korrekt på display og LED'er. Lommeregneren lever således op til alle opstillede krav og fungerer stabilt under brug.

Her er en video af programmet i brug:

 CalcProjekt (8*2/4)



Her er en opsummering af kravlisten med status på hvert enkelt krav:

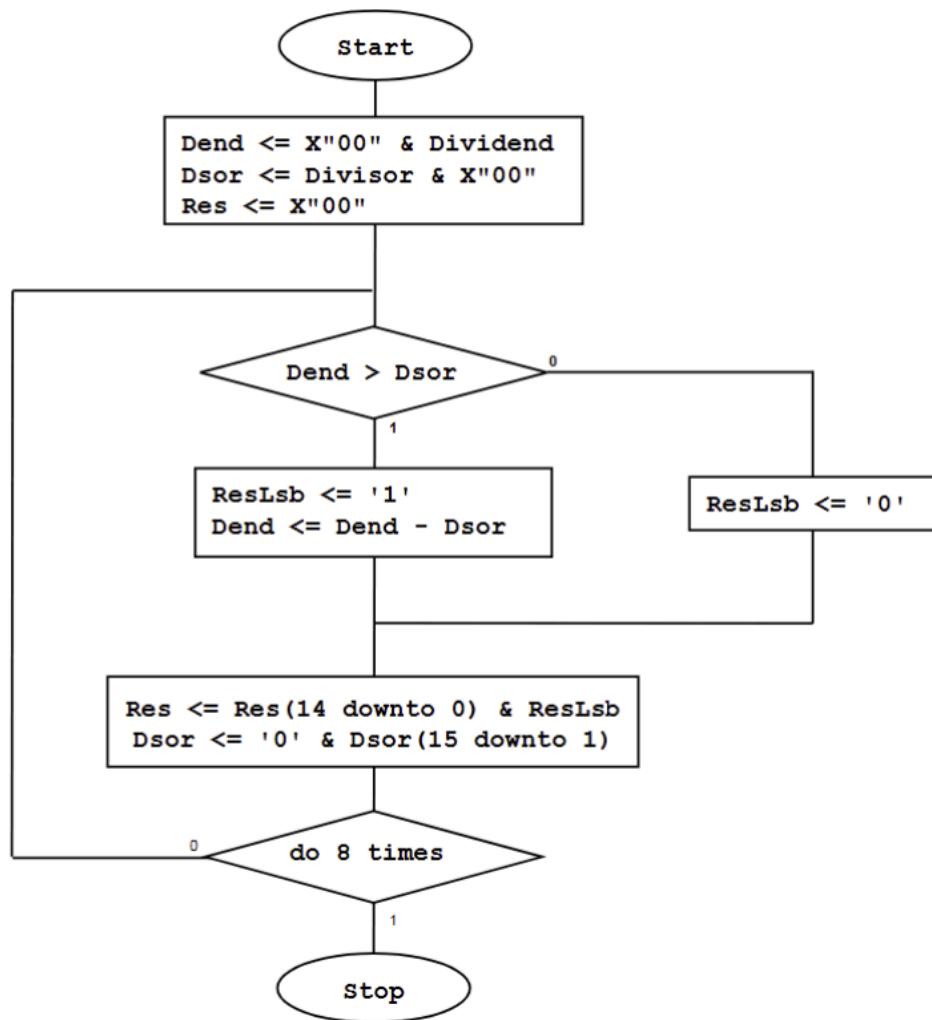
Funktionelle krav:	Kravbeskrivelse:	Status:
Krav 1:	Systemet skal kunne udføre følgende aritmetiske operationer: addition (+), subtraktion (-), multiplikation (*) og division (/)	Opfyldt!
Krav 2:	Systemet skal udføre beregningen i følgende rækkefølge: Resultat = (Input A Op1 Input B) Op2 Input C	Opfyldt!
Krav 3:	Systemet skal modtage tre 8-bit inputværdier (Input A, B, og C) via switche (SW) og trykknapper (BTN) på FPGA-boardet	Opfyldt!
Krav 4:	Systemet skal modtage to aritmetiske operatorer (Op1 og Op2) via switche og knapper	Opfyldt!
Krav 5:	Systemet skal benytte de fire trykknapper (BTN0-BTN3) til styring af inputfunktionalitet, som beskrevet i tabel nedenfor	Opfyldt!

Hardware krav:	Kravbeskrivelse:	Status:
Krav 1:	Systemet skal implementeres på et Basys2 FPGA-board	Opfyldt!
Krav 2:	Systemet skal benytte switcher (SW0-SW7) til at angive 8-bit værdier i HEX	Opfyldt!
Krav 3:	Systemet skal bruge trykknapperne (BTN0-BTN3) til inputkontrol	Opfyldt!
Krav 4:	Resultat skal kunne vises på 7-segment display	Opfyldt!
Krav 5:	Systemet skal kunne vise hvilket state den befinner sig igennem LED'erne på boardet (LDO-LD7)	Opfyldt!

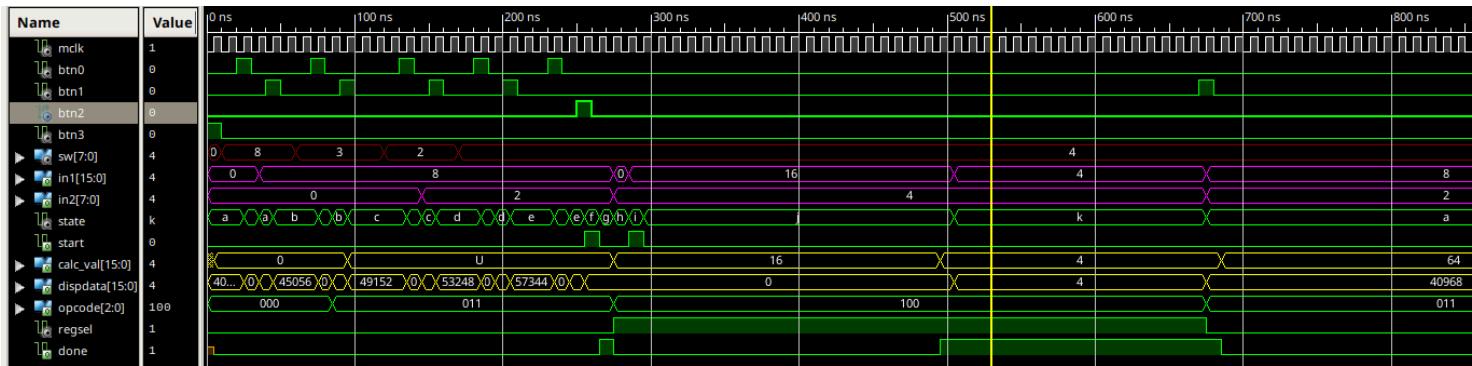
Bilag

Bilag 1. Divisions flowchart

Division Flow

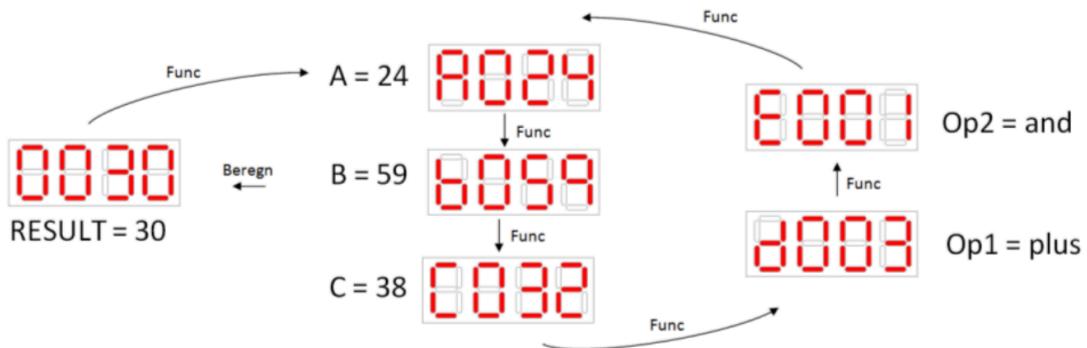


Bilag 2. Simulering af Calc TOP



*Se vedhæftet bilag for fuld størrelse

Bilag 3. Rækkefølge af tilstænde



Bilag 4. Koden

Calc_Top

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

entity Calc_Top is
    Port (
        BTN3 : in STD_LOGIC;                      -- Reset
        MClk : in STD_LOGIC;                      -- Master Clock
        BTN0 : in STD_LOGIC;                      -- Enter button
        BTN1 : in STD_LOGIC;                      -- Func button
        BTN2 : in STD_LOGIC;                      -- Operation button
        SW   : in STD_LOGIC_VECTOR (7 downto 0);  -- Switches
        An   : out STD_LOGIC_VECTOR (3 downto 0); -- 7-Segment anodes
        Cat  : out STD_LOGIC_VECTOR (6 downto 0); -- 7-Segment cathodes
        ld   : out STD_LOGIC_VECTOR (7 downto 0)  -- FSM state output to LEDs
    );
end Calc_Top;

architecture Behavioral of Calc_Top is

    -- Internal signals
    signal DispClk    : STD_LOGIC;              -- Clock for display
    signal BTN2d      : STD_LOGIC;              -- Debounced BTN2
    signal BTN1d      : STD_LOGIC;              -- Debounced BTN1
    signal start       : STD_LOGIC;              -- Start signal for calculation
    signal done        : STD_LOGIC;              -- Done signal from datapath
    signal DispData   : STD_LOGIC_VECTOR (15 downto 0); -- Displayed data
    signal In1         : STD_LOGIC_VECTOR (15 downto 0); -- Input 1
    signal CalcVal    : STD_LOGIC_VECTOR (15 downto 0); -- Result from datapath
    signal In2         : STD_LOGIC_VECTOR (7 downto 0);  -- Input 2
    signal OpCode     : STD_LOGIC_VECTOR (2 downto 0);  -- Operation code

begin

    -- Clock Divider to generate display-friendly clock
    U1: entity work.DivClk
    port map (
        Reset  => BTN3,
        Clk    => MClk,
        TimeP  => 50e3,
        Clk1   => DispClk
    );


```

```

--7-Segment Display Controller
U2: entity work.SevenSeg4
port map (
    Reset => BTN3,
    Clk    => DispClk,
    Data   => DispData,
    Cat    => Cat,
    An     => An
);

-----
-- Debounce modules (should be commented out in simulation/testbench)

U3: entity work.Debounce
port map (
    Reset => BTN3,
    Clk   => MClk,
    BTN   => BTN2,
    BTNd  => BTN2d
);

U4: entity work.Debounce
port map (
    Reset => BTN3,
    Clk   => MClk,
    BTN   => BTN1,
    BTNd  => BTN1d
);

-----
-- Calc_Menu Module: FSM and input handling

-- For simulation:
-- Use this version (with raw button inputs)
--
-- U5: entity work.Calc_Menu
port map (
    Reset      => BTN3,
    CLK        => MCLK,
    Enter     => BTN0,
    Operation  => BTN2,
    Func       => BTN1,
    SW         => SW,
    CalcVal   => CalcVal,
    DispData  => DispData,
    Tilstand  => ld,
    Start     => Start,
    Done      => Done,
    OpCode    => OpCode,
    In1       => In1,

```

```

--          In2      => In2
--      );
-- For hardware implementation:
-- Use this version (with debounced button inputs)

U5: entity work.Calc_Menu
port map (
    Reset      => BTN3,
    Clk        => MC1k,
    Enter      => BTN0,
    Operation  => BTN2d,
    Func       => BTN1d,
    SW         => SW,
    CalcVal    => CalcVal,
    DispData   => DispData,
    Tilstand   => ld,
    Start      => Start,
    Done       => Done,
    OpCode     => OpCode,
    In1        => In1,
    In2        => In2
);
-- Datapath Module: Performs the actual calculations
U6: entity work.Calc_Data_topmodule
port map (
    Clk      => MClk,
    Calc_Val => CalcVal,
    Start    => Start,
    Done     => Done,
    OpCode  => OpCode,
    Inp1    => In1,
    Inp2    => In2
);
end Behavioral;

```

Calc_Menu

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

entity Calc_Menu is
    Port ( Reset : in STD_LOGIC;
           Clk : in STD_LOGIC;
           Enter : in STD_LOGIC;
           Operation : in STD_LOGIC;
           Func : in STD_LOGIC;
           CalcVal : in STD_LOGIC_VECTOR (15 downto 0);
           DispData : out STD_LOGIC_VECTOR (15 downto 0);
           Start : out STD_LOGIC;
           Done : in STD_LOGIC;
           OpCode : out STD_LOGIC_VECTOR (2 downto 0);
           In1 : out STD_LOGIC_VECTOR (15 downto 0);
           In2 : out STD_LOGIC_VECTOR (7 downto 0);
           SW : in STD_LOGIC_VECTOR (7 downto 0);
           Tilstand: out std_logic_vector (7 downto 0) -- State output for LED
visualization
    );
end Calc_Menu;

architecture Behavioral of Calc_Menu is

    -- Define state type for the FSM
    type Statetype is (A, B, C, D, E, F, G, H, I, J, K, Val1, Val2, Val3,
Op1, Op2);

    -- Signals for current and next state
    signal state, nextstate : Statetype;

    -- Internal signals for calculations and register enables
    signal Calcvls : std_logic_vector (15 downto 0);
    signal Op1S, Op2S, Val1S, Val3S, DispSel, Val2S : STD_LOGIC_VECTOR (7
downto 0);
    signal Val1En, Val2En, Val3En, Op1En, Op2En, RegSel : STD_LOGIC;

begin

    -- Sequential process for updating the state
    Statereg: process(reset, clk)
    begin
        if reset = '1' then
            state <= A; -- Reset state to A
        elsif clk'event and clk = '1' then
            state <= nextstate; -- Update state on clock edge
        end if;
    end process;
```

```

-- Next state logic and register enable control
StateDec: process (state, func, enter, operation, done)
begin
    -- Default values for register enables and outputs
    Val1En <= '0';
    Val2En <= '0';
    Val3En <= '0';
    Op1En <= '0';
    Op2En <= '0';
    Start <= '0';
    RegSel <= '0';
    DispSel <= "00000000";
    Tilstand <= "00000000"; -- Default state (no LEDs lit)

    case state is
        -- State A: Display Selection and Function Check
        when A =>
            DispSel <= "00000001"; -- Display selection for state A
            Tilstand <= "00000001"; -- Light up LEDs to show state A (binary 00000001)
            if func = '1' then
                nextstate <= B; -- Transition to state B if function is pressed
            elsif enter = '1' then
                nextstate <= Val1; -- Transition to Val1 state on Enter
            elsif operation = '1' then
                nextstate <= F; -- Transition to state F if operation is pressed
            else
                nextstate <= A; -- Stay in state A
            end if;

            -- State Val1: Enable Value 1 Register
            when Val1 =>
                Val1En <= '1'; -- Enable Val1 register
                nextstate <= A; -- Return to state A

        -- State B: Display Selection and Function Check
        when B =>
            DispSel <= "00000010"; -- Display selection for state B
            Tilstand <= "00000010"; -- Light up LEDs to show state B (binary 00000010)
            if func = '1' then
                nextstate <= C; -- Transition to state C if function is pressed
            elsif enter = '1' then
                nextstate <= Op1; -- Transition to Op1 state on Enter
            elsif operation = '1' then
                nextstate <= F; -- Transition to state F if operation is pressed
            else
                nextstate <= B; -- Stay in state B
            end if;

```

```

-- State Op1: Enable Op1 Register
when Op1 =>
Op1En <= '1'; -- Enable Op1 register
    nextstate <= B; -- Return to state B

-- State C: Display Selection and Function Check
when C =>
DispSel <= "00000011"; -- Display selection for state C
Tilstand <= "00000100"; -- Light up LEDs to show state C (binary 00000100)
    if func = '1' then
        nextstate <= D; -- Transition to state D if function is pressed
    elsif enter = '1' then
        nextstate <= Val2; -- Transition to Val2 state on Enter
    elsif operation = '1' then
        nextstate <= F; -- Transition to state F if operation is pressed
    else
        nextstate <= C; -- Stay in state C
    end if;

-- State Val2: Enable Value 2 Register
when Val2 =>
Val2En <= '1'; -- Enable Val2 register
    nextstate <= C; -- Return to state C

-- State D: Display Selection and Function Check
when D =>
DispSel <= "00000100"; -- Display selection for state D
Tilstand <= "00001000"; -- Light up LEDs to show state D (binary 00001000)
    if func = '1' then
        nextstate <= E; -- Transition to state E if function is pressed
    elsif enter = '1' then
        nextstate <= Op2; -- Transition to Op2 state on Enter
    elsif operation = '1' then
        nextstate <= F; -- Transition to state F if operation is pressed
    else
        nextstate <= D; -- Stay in state D
    end if;

-- State Op2: Enable Op2 Register
when Op2 =>
Op2En <= '1'; -- Enable Op2 register
    nextstate <= D; -- Return to state D

-- State E: Display Selection and Function Check
when E =>
DispSel <= "00000101"; -- Display selection for state E
Tilstand <= "00010000"; -- Light up LEDs to show state E (binary 00010000)

```

```

if func = '1' then
    nextstate <= A; -- Transition to state A if function is pressed
elsif enter = '1' then
    nextstate <= Val3; -- Transition to Val3 state on Enter
elsif operation = '1' then
    nextstate <= F; -- Transition to state F if operation is pressed
else
    nextstate <= E; -- Stay in state E
end if;

-- State Val3: Enable Value 3 Register
when Val3 =>
Val3En <= '1'; -- Enable Val3 register
nextstate <= E; -- Return to state E

-- State F: Start Operation
when F =>
Start <= '1'; -- Set Start signal to 1 to begin operation
RegSel <= '0'; -- Disable Register selection
nextstate <= G; -- Transition to state G

-- State G: Check for Done Signal
when G =>
Tilstand <= "00100000"; -- Light up LEDs to show state G (binary 00100000)
Start <= '0'; -- Reset Start signal
if done = '1' then
    nextstate <= H; -- Transition to state H if done signal is received
else
    nextstate <= G; -- Stay in state G
end if;

-- State H: Enable Register Selection
when H =>
RegSel <= '1'; -- Enable Register selection
nextstate <= I; -- Transition to state I

-- State I: Start Second Operation
when I =>
RegSel <= '1'; -- Keep Register selection enabled
Start <= '1'; -- Set Start signal for operation
nextstate <= J; -- Transition to state J

```

```

-- State J: Check for Done Signal
when J =>
  RegSel <= '1'; -- Keep Register selection enabled
  Tilstand <= "01000000"; -- Light up LEDs to show state J (binary 01000000)
  Start <= '0'; -- Reset Start signal
    if done = '1' then
      nextstate <= K; -- Transition to state K if done signal is received
    else
      nextstate <= J; -- Stay in state J
    end if;

-- State K: Final State and Display Selection
when K =>
  RegSel <= '1'; -- Keep Register selection enabled
  Tilstand <= "10000000"; -- Light up LEDs to show state K (binary 10000000)
  DispSel <= "00000110"; -- Final display selection (Show Result)
    if func = '1' then
      nextstate <= A; -- Return to state A if function is pressed
    else
      nextstate <= K; -- Stay in state K
    end if;
  end case;
end process;

-- Display data assignment based on DispSel value
with DispSel select
  DispData <=
    (others => '0')           when "00000000",
    X"A0" & Val1S             when "00000001",
    X"B0" & Op1S              when "00000010",
    X"C0" & Val2S              when "00000011",
    X"D0" & Op2S              when "00000100",
    X"E0" & Val3S              when "00000101",
    Calcvals                  when "00000110",
    others                     when others;

-- Register mappings for 8-bit registers (Val1, Val2, Val3, Op1, Op2)
Val1Reg: entity work.std_8bit_reg
port map (
  Reset => reset,
  Clk   => clk,
  Enable => Val1En,
  Data_in => SW,
  Data_out => Val1S
);

```

```

Val2Reg: entity work.std_8bit_reg
port map (
    Reset => reset,
    Clk    => clk,
    Enable => Val2En,
    Data_in => SW,
    Data_out => Val2S
);

Val3Reg: entity work.std_8bit_reg
port map (
    Reset => reset,
    Clk    => clk,
    Enable => Val3En,
    Data_in => SW,
    Data_out => Val3S
);

Op1Reg: entity work.std_8bit_reg
port map (
    Reset => reset,
    Clk    => clk,
    Enable => Op1En,
    Data_in => SW,
    Data_out => Op1S
);

Op2Reg: entity work.std_8bit_reg
port map (
    Reset => reset,
    Clk    => clk,
    Enable => Op2En,
    Data_in => SW,
    Data_out => Op2S
);

-- 16-bit register for storing calculation results
CalcValReg: entity work.std_16bit_reg
port map (
    Reset => reset,
    Clk    => clk,
    Enable => RegSel,
    Data_in => CalcVal,
    Data_out => CalcVals
);

end Behavioral;

```

Calc_Data

```
library IEEE;
--Library numeric_std;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following Library declaration if using
-- arithmetic functions with Signed or Unsigned values

-- Uncomment the following Library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Calc_data_topmudule is
    Port (
        Clk      : in STD_LOGIC;
        Opcode   : in STD_LOGIC_VECTOR (2 downto 0);
        Inp1    : in STD_LOGIC_VECTOR (15 downto 0);
        Inp2    : in STD_LOGIC_VECTOR (7 downto 0);
        start   : in STD_LOGIC;
        Calc_val : out STD_LOGIC_VECTOR (15 downto 0);
        done    : out STD_LOGIC
    );
end Calc_data_topmudule;

architecture Behavioral of Calc_data_topmudule is

    -- Component for division
    component div_component is
        Port (
            Inp1   : in STD_LOGIC_VECTOR (15 downto 0); -- dividend
            Inp2   : in STD_LOGIC_VECTOR (7 downto 0); -- divisor
            clk    : in STD_LOGIC;
            start  : in STD_LOGIC;                      -- start signal
            Resdiv : out STD_LOGIC_VECTOR (15 downto 0); -- result (number of
times divisor fits in dividend)
            doneDiv : out STD_LOGIC-- goes high when division is done
        );
    end component;

    -- Signals
    signal ResPlu   : std_logic_vector(15 downto 0);
    signal ResMin   : std_logic_vector(15 downto 0);
    signal ResGan   : std_logic_vector(15 downto 0);
    signal Resdiv   : std_logic_vector(15 downto 0);
    signal doneDiv  : STD_Logic;
```

```

begin

    u1 : div_component
    port map(
        Inp1    => Inp1,
        Inp2    => Inp2,
        start   => start,
        clk     => Clk,
        Resdiv  => Resdiv,
        doneDiv => doneDiv
    );

    -- Calculate +, -, *
    process(Clk)
    begin
        if rising_edge(Clk) then
            if start = '1' then
                ResPlu <= std_logic_vector(unsigned(Inp1) + unsigned("00000000" &
Inp2)); -- Inp1 + Inp2 is loaded into ResPlu
                ResMin <= std_logic_vector(unsigned(Inp1) - unsigned("00000000" &
Inp2)); -- Inp1 - Inp2 is loaded into ResMin
                ResGan <= std_logic_vector(resize(unsigned(Inp1) *
unsigned("00000000" & Inp2), 16)); -- resize to ensure the result doesn't get
too big
            end if;
        end if;
    end process;

    -- Result is loaded into Calc_val
    process(clk)
    begin
        if rising_edge(clk) then
-- Inp1/Inp2 is loaded into Calc_val when doneDiv (division) is finished and
opcode is "100"
            if Opcode = "100" then
                if doneDiv = '1' then
                    Calc_val <= Resdiv;
                end if;
                else
                    case Opcode is
when "001" => Calc_val <= ResPlu; -- Calc_val is assigned the value of ResPlu
when "010" => Calc_val <= ResMin; -- Calc_val is assigned the value of ResMin
when "011" => Calc_val <= ResGan; -- Calc_val is assigned the value of ResGan
when others  => Calc_val <= (others => '0'); -- to cover all cases, Calc_val
is set to '0' if the opcode is invalid
                    end case;
                end if;
            end if;
        end process;

```

```

-- Below, the done signal is set in a clocked process because otherwise
it would arrive too quickly compared to Calc_val
process(clk)
begin
if rising_edge(clk) then
    if Opcode = "100" and doneDiv = '1' then
        done <= '1';
    elsif Opcode /= "100" and start = '1' then
        done <= '1';
    else
        done <= '0';
    end if;
end if;
end process;

end Behavioral;

```

Div_Component

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

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity div_component is
    Port (
        Inp1 : in STD_LOGIC_VECTOR (15 downto 0); -- dividend
        Inp2 : in STD_LOGIC_VECTOR (7 downto 0); -- divisor
        clk : in STD_LOGIC;
        start : in STD_LOGIC; -- the start signal
        Resdiv : out STD_LOGIC_VECTOR (15 downto 0); -- result (number of times
the divisor fits into the dividend)
        doneDiv : out STD_LOGIC -- sent when division is finished
    );
end div_component;

architecture Behavioral of div_component is

    signal ResLSB : std_logic;
    signal counter : integer range 0 to 18; -- Counter for clock cycles
set to 16+1 so there's one clock to load values
    signal Dividend : std_logic_vector(31 downto 0);
    signal Divisor : std_logic_vector(23 downto 0);
    signal Div_reg : std_logic_vector(31 downto 0);
    signal Dsor_reg : std_logic_vector(23 downto 0);
    signal Resultat : STD_LOGIC_VECTOR(15 downto 0);
    signal intdone : std_logic; -- internal done signal
    signal aktiv : std_logic; -- used to keep the division
component active until it's done

begin

    -- dividend register
    process(clk)
    begin
        if rising_edge(clk) then
            if start = '1' then
                Div_reg <= (31 downto 16 => '0') & Inp1;
            elsif start = '0' then
                Div_reg <= Div_reg;
            end if;
        end if;
    end process;
```

```

-- divisor register
process(clk)
begin
if rising_edge(clk) then
    if start = '1' then
        Dsor_reg <= Inp2 & (15 downto 0 => '0'); -- merges 16 lower bits on
the right side of input 2 (the divisor)
    elsif start = '0' then
        Dsor_reg <= Dsor_reg; -- if start is low, the register should just
stay the same (can optionally be set to 0 for reset)
    end if;
end if;
end process;

-- Division block
process(clk)
begin
if rising_edge(clk) then
    if counter = 0 then
        Dividend <= Div_reg; -- Div register value is loaded into dividend
        Divisor <= Dsor_reg; -- Dsor register value is loaded into divisor

    elsif counter <= 18 then -- until the counter reaches 18, the below runs
        Divisor <= '0' & Divisor(23 downto 1); -- right shift
    if unsigned(Dividend) < unsigned(Divisor) then
        ResLSB <= '0'; -- if dividend is less than divisor, ResLSB goes low and
this bit is left-shifted into result
    else
        Dividend <= std_logic_vector(unsigned(Dividend) - unsigned(Divisor)); --
if dividend >= divisor, subtract divisor from dividend
        ResLSB <= '1'; -- ResLSB is set high and ready to be shifted into result
    end if;
    else
        ResLSB <= '0'; -- fallback when counter > 18, safety for 'elsif'
    counter <= 18 then'
    end if;
end if;
end process;

-- result block
process(clk)
begin
if rising_edge(clk) then
    if aktiv = '1' then -- if active is high, this process runs
        if ResLSB = '1' then
            Resultat <= Resultat(14 downto 0) & '1'; -- if ResLSB is
high, merge a '1' bit to the right of result
        elsif ResLSB = '0' then
            Resultat <= Resultat(14 downto 0) & '0'; -- if ResLSB is

```

```

Low, merge a '0' bit to the right of result
    else
        Resultat <= (15 downto 0 => '0');
    end if;
    end if;
end if;
end process;

-- counter block
process(clk)
begin
if rising_edge(clk) then
    if start = '1' then
        counter <= 0;          -- counter initialized to 0
        intdone <= '0';       -- intdone initialized to 0
        aktiv <= '1';         -- active signal set when start goes high, keeps
count going
        elsif aktiv = '1' then -- counting happens if active is high
            if counter < 18 then -- set to 18 for buffer, to allow overflow, and
to Load/send values
                counter <= counter + 1;
            else
                intdone <= '1'; -- internal done signal sent
                aktiv <= '0'; -- finished
            end if;
        end if;
    end if;
end process;

Resdiv <= Resultat; -- we connect result to Resdiv which is passed on to
the top module
doneDiv <= intdone; -- the internal done signal is connected to doneDiv
which is passed to the top module

end Behavioral;

```

TestBench

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

entity Calc_Top_Uden_Debounce_TB is
end Calc_Top_Uden_Debounce_TB;

architecture behavior of Calc_Top_Uden_Debounce_TB is

    -- Component declaration
    component Calc_Top
    Port(
        BTN3 : IN std_logic;
        MClk : IN std_logic;
        BTN0 : IN std_logic;
        BTN1 : IN std_logic;
        BTN2 : IN std_logic;
        SW   : IN std_logic_vector(7 downto 0);
        An   : OUT std_logic_vector(3 downto 0);
        Cat  : OUT std_logic_vector(6 downto 0);
        Id   : OUT std_logic_vector(7 downto 0)
    );
    end component;

    -- Input signals
    signal BTN3 : std_logic := '0';
    signal MClk : std_logic := '0';
    signal BTN0 : std_logic := '0';
    signal BTN1 : std_logic := '0';
    signal BTN2 : std_logic := '0';
    signal SW   : std_logic_vector(7 downto 0) := (others => '0');

    -- Output signals
    signal An  : std_logic_vector(3 downto 0);
    signal Cat : std_logic_vector(6 downto 0);
    signal Id  : std_logic_vector(7 downto 0);

    -- Clock period definition
    constant MClk_period : time := 10 ns;

begin

    -- Instantiate the Unit Under Test (UUT)
    uut: Calc_Top
    port map (
        BTN3 => BTN3,
        MClk => MClk,
        BTN0 => BTN0,
        BTN1 => BTN1,
```

```

        BTN2 => BTN2,
        SW    => SW,
        An    => An,
        Cat   => Cat,
        ld    => ld
    );

-- Clock generation process
MClk_process : process
begin
MClk <= '0';
wait for MClk_period/2;
MClk <= '1';
wait for MClk_period/2;
end process;

-- Stimulus process: run through all FSM states, doing the calculation of
((8*2)/4)
stim_proc: process
begin
-- Reset
BTN3 <= '1';
wait for MClk_period;
BTN3 <= '0';

-- State A - Load Val1
SW <= "00001000";--8
wait for MClk_period;
BTN0 <= '1';
wait for MClk_period;
BTN0 <= '0';
wait for MClk_period;

-- Move to State B
BTN1 <= '1';
wait for MClk_period;
BTN1 <= '0';
wait for MClk_period;

-- State B - Load Op1
SW <= "00000011";--*
wait for MClk_period;
BTN0 <= '1';
wait for MClk_period;
BTN0 <= '0';
wait for MClk_period;

-- Move to State C
BTN1 <= '1';
wait for MClk_period;

```

```

BTN1 <= '0';
wait for 20 ns;

-- State C - Load Val2
SW <= "00000010";--2
wait for MClk_period;
BTN0 <= '1';
wait for MClk_period;
BTN0 <= '0';
wait for MClk_period;

-- Move to State D
BTN1 <= '1';
wait for MClk_period;
BTN1 <= '0';
wait for MClk_period;

-- State D - Load Op2
SW <= "00000100";--/
wait for MClk_period;
BTN0 <= '1';
wait for MClk_period;
BTN0 <= '0';
wait for MClk_period;

-- Move to State E - Load Val3
BTN1 <= '1';
wait for MClk_period;
BTN1 <= '0';
wait for MClk_period;

SW <= "00000100";--4
wait for MClk_period;
BTN0 <= '1';
wait for MClk_period;
BTN0 <= '0';
wait for MClk_period;

-- Move to State F - Execute
BTN2 <= '1';
wait for MClk_period;
BTN2 <= '0';
wait for MClk_period;

-- Wait for operation to complete
wait for 40 * MClk_period;

-- Return to State A
BTN1 <= '1';
wait for MClk_period;

```

```
    BTN1 <= '0';
    wait for MClk_period;

    wait;
    end process;

end behavior;
```