

# Ingebedde systemen: project - controller voor servomotoren

Francis Begyn en Heleen Cauwels

2017

Begeleider: Dimitri Van Cauwelaert

# Inhoudstabel

Inhoudstabel	2
Lijst met figuren	3
Lijst met tabellen	4
Lijst met afkortingen	5
Inleiding	6
1 Vereisten en afspraken van de controller	6
2 Verslag code	7
2.1 Entity . . . . .	7
2.2 Architecture . . . . .	7
2.3 Testbench . . . . .	11

## Lijst met figuren

## Lijst met tabellen



## Inleiding

De opdracht is het maken van een controller voor een servomotor in VHDL en een testbench om deze te kunnen uittesten. Hier is het ook belangrijk om rekening te houden met de vereisten van de klant en indien nodig het apparaat aan te passen.

Een servomotor is een motor die door middel van Pulse Width Modulation (PWM) aangestuurd kan worden naar een gewenste positie en deze in normale operatie zal behouden. De snelheid waarmee deze behouden wordt, is afhankelijk van de kloksnelheid die aan de controller meegegeven wordt.

## 1 Vereisten en afspraken van de controller

De controller moet in staat om een servomotor over 180 aan te sturen met PWM signalen tussen 1,25ms en 1,75ms. De reset verloopt synchroon met de klok, dit is om te verhinderen dat het PWM signaal vervormt zou worden door de slechte timing van een reset. Moet men met een asynchrone reset werken, zou het PWM signaal onbedoeld langer kunnen worden dan de maximum pulsduur (1,75ms).

De controller werkt met een klok van 50Hz en een servoklok van 510kHz. Hiermee rekening houdende blijkt dat de minimum pulsduur aan 1,25ms overeen komt met 637,5 servoklok ticks, het midden van 1,5ms komt overeen met 765 servoklok ticks en de maximum pulsduur van 1,75ms komt overeen met 892,5 servoklok ticks. In VHDL wordt gewerkt met unsigned variabelen, dus 637,5 en 892,5 worden afgerond naar respectievelijk 637 en 892.

Elke controller heeft een adres waarop die moet luisteren. Als er opdrachten gestuurd worden die vooraf gegaan worden door het correcte address, dan moet de controller deze instructie uitvoeren. Indien een ander address vermeld wordt dan mag de controller de opdracht negeren.

Ook is er een broadcast adres waarop alle controller moeten luisteren, namelijk 255. Als een signaal uitgestuurd wordt met het broadcast adres dan moeten alle aangesloten controller naar dat signaal luisteren.

Onder normale operatie zal de controller de positie behouden, maar als er iets fout loopt zal de controller de servomotor terugzetten naar zijn neutrale positie. De neutrale positie wordt hier gedefinieerd als op 0 radialen, en komt overeen met een pulsduur van 1,5ms. De controller keert ook naar deze neutrale positie terug indien een reset opgeroepen wordt.

Het Done signaal werkt met 3 state logic, dit betekend dat het op een bus aangesloten kan worden. Hierdoor kunnen meerdere controllers gebruik maken van dezelfde bus, waardoor er bespaard kan worden op het aantal kabels dat nodig is.

Als de aansturende microcontroller tijdens het aansturen van de controller beslist om een taak van hogere prioriteit uit te voeren, moet de controller dit kunnen afhandelen en naar de neutrale positie gaan.

## 2 Verslag code

### 2.1 Entity

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

entity servocontrol is
  generic (
    address : unsigned(7 downto 0)
  );
  port (
    clk      : in  std_logic;
    rst      : in  std_logic;
    sclk     : in  std_logic;
    set      : in  std_logic;
    data     : in  std_logic_vector(7 downto 0);
    done     : out std_logic;
    pwm      : out std_logic
  );
end entity;
```

### 2.2 Architecture

In het begin van de architectuur definiëren we de benodigde signalen. Op basis van cnt en pwmi kunnen we gemakkelijk een PWM signaal laten generen (verder meer details over hoe). Daarna worden de nodige types en signalen aangemaakt om een statemachine te kunnen gebruiken. Eerst worden de verschillende staten gedefinieerd waarna de signalen voor het bijhouden van de staat ook aangemaakt worden.

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

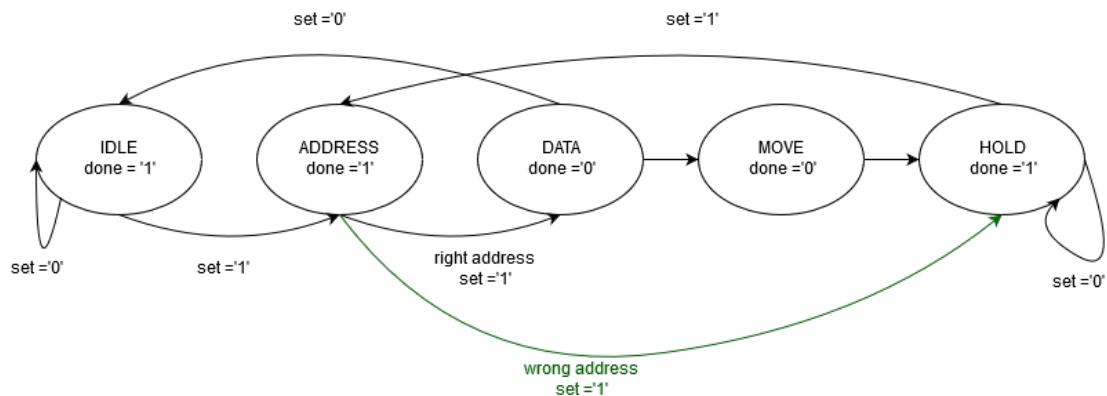
architecture control of servocontrol is

  signal cnt : unsigned(9 downto 0) := (others => '0');
```

```
signal pwm_i : unsigned(9 downto 0) := (others => '0');
```

```
--signal pwm_gen : std_logic;
type state is (idle,addr_rd,data_rd,move,hold);
signal currentState : state:= idle;
signal nextState: state:= idle;
```

state\_trans beschrijft de transitie tussen verschillende staten. Hierbij hoort volgende flowchart.



Hierbij moet rekening gehouden worden met een aantal dingen. Zo is er een reset met hoogste prioriteit, deze heeft altijd voorrang. Moet er gekeken worden of het address juist is of het address het broadcast address is. Indien de set onderbroken wordt, moet de controller naar de neutrale positie gaan.

```
-- State_trans describes the transitions of the states.
state_trans : process(currentState,set)
begin
  case currentState is
    when idle =>
      -- report "state: idle";
      if set = '1' then
        nextState <= addr_rd;
      else
        nextState <= idle;
      end if;
    when addr_rd =>
      -- report "state: addr_rd";
      if set = '1' then
        -- report "address is "& integer'image(to_integer(address));
        -- report "data (address) is "& integer'image(to_integer(unsigned(data)))
        ;
      end if;
    when data_rd =>
      -- report "state: data_rd";
      if set = '1' then
        nextState <= move;
      else
        nextState <= data_rd;
      end if;
    when move =>
      -- report "state: move";
      if set = '1' then
        nextState <= hold;
      else
        nextState <= move;
      end if;
    when hold =>
      -- report "state: hold";
      if set = '1' then
        -- report "wrong address";
        nextState <= addr_rd;
      else
        nextState <= hold;
      end if;
  end case;
end process;
```



```

        if unsigned(data) = address then
            nextState <= data_rd;
        else
            nextState <= hold;
        end if;
    else
        nextState <= idle;
    end if;
when data_rd =>
    -- report "state: data_rd";
    nextState <= move;
when move =>
    -- report "state: move";
    nextState <= hold;
when hold =>
    -- report "state: hold";
    if set ='1' then
        nextState <= addr_rd;
    elsif set ='0' then
        nextState <= hold;
    end if;
end process state_trans;

```

transition is gewoon het synchroon proces dat de overgangen van de staten dicteert.

```

        when others =>
            nextState <= idle;
        end case;

    end process state_trans;

```

set\_output dicteert hoe de output signalen zich gedragen in elke staat. In deze setup stelt zich dit zo dat Done een 3 state logic heeft. Het PWM signaal wordt hier niet gedefinieerd omdat er een oneliner gebruikt wordt om dit aan de uitgang te koppelen (zie verder).

```

transition: process(rst,clk) begin
    if rst = '1' then
        currentState <= idle;
    elsif falling_edge(clk) then
        currentState <= nextState;
    end if;
end process transition;

-- set_output determines which output correspond with a state
-- The done is defined so it works on bus structure, 3 state logic
set_output: process(currentState,clk, nextState) begin

```

```

if (rising_edge(clk)) then
  case currentState is
    when idle =>
      done <= 'H';
    when addr_rd =>
      if (nextState=data_rd) then
        done <= 'L';
      else
        done <= 'H';

```

pwm\_data is het proces dat de berekeningen en lengte van de puls bepaalt. Op basis van de servoklok wordt berekend welke waarden nodig zijn om de gewenste pulsen te bekomen. In de commentaar zijn de berekende waarden voor een servoklok van 510kHz te vinden. Indien de servoklok aangepast wordt, met deze code aangepast worden zodat de juiste waarden gebruikt worden.

```

    when data_rd =>
      done <= 'L';
    when move =>
      done <= 'H';
    when hold =>
      done <= 'H';

    when others =>
      -- done <= '-';
  end case;
end if;
end process set_output;

-- pwm_data sets the amount of ticks needed to generate a correct duration with
  servo clock = 510kHz
-- 1.25ms = 637 ticks, 1.5ms = 765 ticks, 1.75ms = 892 ticks
pwm_data: process(currentState,clk) begin
  case currentState is

```

gen\_pwm is het proces dat de eigenlijk puls genereert. Het aantal servoklok ticks wordt bijgehouden en vergeleken met de benodigde aantallen die in pwm\_data bepaald zijn. cnt moet gerest worden bij elke klok tick.

Nadien wordt met een oneliner eigenlijk het pwm signaal gegenereert door een klein procesje dat reageert op de veranderingen in gen\_pwm.

```

  pwmi <= to_unsigned(766,10); -- values according to 510kHz servo clock.
when move =>
  -- report "setting value";
  if data > std_logic_vector(to_unsigned(255,8)) then

```

```

        -- report "255";
        pwmi <= to_unsigned(892,10);
    else
        -- report "setting ...";
        pwmi <= unsigned('0' & data) + to_unsigned(638,10);
        -- report " "&integer'image(to_integer(pwmi));
    end if;

    when others =>
    end case;

```

### 2.3 Testbench

De testbench is verantwoordelijk om de geschreven architecture uit te testen. De vereiste van de testbench is dat deze een klok en servoklok genereert, de servocontroller uittest in stappen van 32, de positie van de servomotor opvraagt en controleert. De servoklok mag niet gebruikt worden om het PWM signaal te controleren.

## Conclusie

