UNIVERSITEIT GENT

Ingebedde systemen: project - controller voor servomotoren

Francis Begyn en Heleen Cauwels

2017

Begeleider: Dimitri Van Cauwelaert

FACULTEIT INGENIEURSWETENSCHAPPEN EN ARCHITECTUUR

Inhoudstabel

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 meegeven 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 an 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 usigned variabelen, dus 637,5 en 892,5 worden afgerond naar respectievelijk 637 en 892.

Elke controller heeft een adress waarop die moet luisteren. Als er opdrachten gestuurd worden die vooraf gegaan worden door het correcte address, dn 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 adress 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 werk met 3 state logic, dit betekend dat het op een bus aangesloten kan worden. Hierdoor kunnen meerdere controllers gebruik maken van nzelfde 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;
```

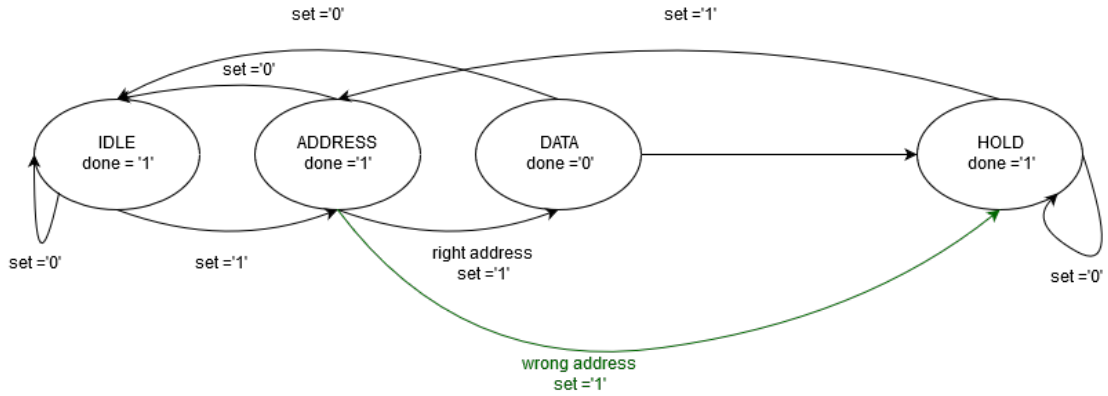
2.2 Architecture

In het begin van de archtectuur definieren we de benodigde signalen. Op basis van cnt en pwmi kunnen we gemakkelijk een PWM singaal laten generen (verder meer details over hoe). Daarna worden de nodige types en signalen aangemaakt om een statenmachine 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 pwmi : unsigned(9 downto 0) := (others => '0');

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

state_trans beschrijft de transities 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 =>
      if set = 'H' then
        nextState <= addr_rd;
      else
        nextState <= idle;
      end if;
    when addr_rd =>
        if set = 'H' then
          if (unsigned(data) = address or unsigned(data) = to_unsigned(255,8))
              then
            nextState <= data_rd;
          else
            nextState <= hold;
          end if;
```

```
            else
              nextState <= idle;
            end if;
        when data_rd =>
          nextState <= hold;
        when hold =>
          if set ='H' then
            nextState <= addr_rd;
          else
            nextState <= hold;
          end if;
        when others =>
          nextState <= idle;
        end case;

    end process state_trans;
```

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

```
    -- Transition is the actual transition beteen states
    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 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 gedfinieerd omdat er een oneliner gebruikt wordt om dit aan de uitgang te koppelen (zie verder).

```
    -- set_output determines which output correspont 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';
```

```
        end if;
      when data_rd =>
        done <= 'L';
      when hold =>
        done <= 'H';

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

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, moet deze code aangepast worden zodat de juiste waarden gebruikt worden.

```
-- 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
    when idle =>
      pwmi <= to_unsigned(766,10); -- values according to 510kHz servo clock.
    when data_rd =>
      if data > std_logic_vector(to_unsigned(255,8)) then
        pwmi <= to_unsigned(892,10);
      else
        pwmi <= unsigned('0' & data) + to_unsigned(638,10);
      end if;

    when others =>
  end case;
end process pwm_data;
```

gen_pwm is het proces dat de eigenlijk puls genereert. Het aantal servoklok ticks wordt bijgehouden en vergelijken met de benodigde aantallen die in pwm_data bepaald zijn. cnt moet gerest wordne bij elke klok tick.
Nadien wordt met een one line process eigenlijk het pwm signaal gegenereert door een klein procesje dat reageert op de veranderingen in gen_pwm.

```
-- gen_pwm is a combination of the proces below and the one-line process below
    it.
-- gen_pwm counts the amount of ticks according to sclk and resets every clk
```

```
  gen_pwm: process(clk, sclk) begin
    if rising_edge(clk) then
      cnt <= (others => '0');
    elsif rising_edge(sclk) then
      if (cnt < 1023) then
        cnt <= cnt + 1;
      end if;
    end if;
  end process gen_pwm;

  -- one-line process generates the output signal
  pwm <= '1' when (cnt < pwmi) else '0'; -- this holds the pwm signal at all times
```

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.

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

entity tb_servocontroller is
end;

architecture test of tb_servocontroller is


--inputs dut
signal clk: std_logic;
signal rst: std_logic;
signal sclk: std_logic;
signal set: std_logic;
signal data: std_logic_vector(7 downto 0);
--outputs dut
signal done: std_logic;
signal pwm: std_logic;
--timing
constant clkPeriod: time:= 20 ms;
constant sclkPeriod: time:= 1.960784 us; --aan te passen
constant dutyCycle: real := 0.5;
```

```
constant idle_time: time:= 1.5 ms;
constant min_time: time:= 1.25 ms;
constant tol: real:= 0.5;
--end of simulation
signal EndOfSim: boolean:= false;


signal plaats: unsigned(8 downto 0):=(others => '0');
signal pwm_start: time;
signal pwm_stop: time;
```

```
begin


dut: entity work.servocontrol
generic map( address => to_unsigned(1,8)
)
port map(
  clk => clk,
  rst => rst,
  sclk => sclk,
  set => set,
  data => data,
  done => done,
  pwm => pwm
);
```

```
clk_gen: process
begin
  clk <= '0';
  wait for (1.0-dutyCycle)*clkPeriod;
  clk <= '1';
  wait for dutyCycle*clkPeriod;
  if EndOfSim then
    wait;
  end if;
end process clk_gen;
```

```
servoclk_gen: process
begin
  sclk <= '0';
  wait for (1.0-dutyCycle)*sclkPeriod;
  sclk <= '1';
```

```
  wait for dutyCycle*sclkPeriod;
  if EndOfSim then
    wait;
  end if;
end process servoclk_gen;
```

```
input_gen: process
begin
  rst<='1';
  wait until falling_edge(clk);
  rst<='0';
  wait until rising_edge(clk);
-- normale werking
  plaats <= (others => '0');
  while (plaats < 256) loop
    report "normale werking, plaats is "&integer'image(to_integer(plaats));
    wait until rising_edge(clk);
    set <='H';
    data <=std_logic_vector(to_unsigned(1,data'length)); -- address sturen
    wait until falling_edge(clk);
    assert(done = 'H')
    report "Done is "&std_logic'image(done)&", verwacht 'H'"
    severity error;
    wait until rising_edge(clk);
    data <= std_logic_vector(plaats(7 downto 0)); -- positie sturen
    wait until falling_edge(clk);
    assert(done ='L')
    report "Done is "&std_logic'image(done)&", verwacht 'L'"
    severity error;
    wait until rising_edge(clk);
    set <= 'L';
    wait until falling_edge(clk);
    assert( done ='L')
    report "Done is "&std_logic'image(done)&", verwacht 'L'"
    severity error;
    wait until rising_edge(pwm);
    pwm_start<=now;
    wait until falling_edge(pwm);
    pwm_stop<=now;
    wait until falling_edge(clk);
    --assert pwmsignaal juist
    assert(abs(((pwm_stop-pwm_start)-(min_time+ to_integer(plaats)*sclkPeriod)))
        <sclkPeriod*tol)
```

```
    report "Fout PWM signaal: " &time'image(pwm_stop-pwm_start)&" /= "&time'
        image(min_time+to_integer(plaats)*sclkPeriod)
    severity error;
    wait until falling_edge(clk);
    assert(done ='H')
    report "Done is "&std_logic'image(done)&", verwacht 'H'"
    severity error;
    plaats <= plaats+32;
    wait for 1 ms;
end loop;
```

```
--reset test
report "Reset test";
wait until rising_edge(clk);
set <='H';
data <=std_logic_vector(to_unsigned(1,data'length)); -- address sturen
wait until falling_edge(clk);
assert(done = 'H')
report "Done is "&std_logic'image(done)&", verwacht 'H'"
severity error;
wait for 5 ms;
rst <='1';
wait for 1 ms;
rst <='0';
wait until rising_edge(clk);
data <= std_logic_vector(plaats(7 downto 0)); -- positie sturen
wait until falling_edge(clk);
assert(done ='H')
report "Done is "&std_logic'image(done)&", verwacht 'H'"
severity error;
wait until rising_edge(clk);
set <= 'L';
wait until falling_edge(clk);
assert( done ='H')
report "Done is "&std_logic'image(done)&", verwacht 'H'"
severity error;
wait until rising_edge(pwm);
pwm_start<=now;
wait until falling_edge(pwm);
pwm_stop<=now;
wait until falling_edge(clk);
--assert pwmsignaal juist
assert(abs((pwm_stop-pwm_start)-idle_time)<sclkPeriod*tol)
```

```
report "Fout PWM signaal: "& time'image(pwm_stop-pwm_start)&" /= "&time'
    image(idle_time)
severity error;
wait until falling_edge(clk);
assert(done ='H')
report "Done is "&std_logic'image(done)&", verwacht 'H'"
severity error;
```

```
--Test Broadcast address (plaats =224)
plaats <= to_unsigned(224,9);
wait for 1 ms;
report "Test broadcast adres, plaats ="&integer'image(to_integer(plaats));
wait until rising_edge(clk);
set <='H';
data <=std_logic_vector(to_unsigned(255,data'length)); -- address sturen
wait until rising_edge(clk);
data <= std_logic_vector(plaats(7 downto 0)); -- positie sturen
wait until rising_edge(clk);
set <= 'L';
wait until rising_edge(clk);
wait until rising_edge(pwm);
pwm_start<=now;
wait until falling_edge(pwm);
pwm_stop<=now;
wait until falling_edge(clk);
--assert pwmsignaal juist
assert(abs((pwm_stop-pwm_start)-(min_time+ to_integer(plaats)*sclkPeriod))<
    sclkPeriod*tol)
report "Fout PWM signaal: "& time'image(pwm_stop-pwm_start)&" /= "&time'
    image(min_time+to_integer(plaats)*sclkPeriod)
severity error;
wait until falling_edge(clk);
assert(done ='H')
report "Done is "&std_logic'image(done)&", verwacht 'H'"
severity error;
```

```
--wrong address (huidige hold plaats= 224 )
plaats <= to_unsigned(32,9);
wait for 1 ms;
report "Geef fout adres mee";
wait until rising_edge(clk);
set <='H';
data <=std_logic_vector(to_unsigned(2,data'length)); -- address sturen
```

```
wait until falling_edge(clk);
assert(done = 'H')
report "Done is "&std_logic'image(done)&", verwacht 'H'"
severity error;
wait until rising_edge(clk);
data <= std_logic_vector(plaats(7 downto 0)); -- positie sturen
wait until falling_edge(clk);
assert(done ='H')
report "Done is "&std_logic'image(done)&", verwacht 'H'"
severity error;
wait until rising_edge(clk);
set <= 'L';
wait until falling_edge(clk);
assert( done ='H')
report "Done is "&std_logic'image(done)&", verwacht 'H'"
severity error;
wait until rising_edge(pwm);
pwm_start<=now;
wait until falling_edge(pwm);
pwm_stop<=now;
wait until falling_edge(clk);
--assert pwmsignaal juist (vorige positie)
assert(abs((pwm_stop-pwm_start)-(min_time+ 224*sclkPeriod))<sclkPeriod*tol)
report "Fout PWM signaal: "& time'image(pwm_stop-pwm_start)&" /= "&time'
    image(min_time+224*sclkPeriod)
severity error;
wait until falling_edge(clk);
assert(done ='H')
report "Done is "&std_logic'image(done)&", verwacht 'H'"
severity error;
wait for 1 ms;
```

```
--no position set after address sent (set goes to 0 on next clk pulse)
report "no position sent after address sent (set ='0' op volgende klokperiode)";
wait until rising_edge(clk);
set <='H';
data <=std_logic_vector(to_unsigned(1,data'length)); -- address sturen
wait until falling_edge(clk);
assert(done = 'H')
report "Done is "&std_logic'image(done)&", verwacht 'H'"
severity error;
wait until rising_edge(clk);
set <='L';
```

```
wait until falling_edge(clk);
assert(done ='L')
report "Done is "&std_logic'image(done)&", verwacht 'L'"
severity error;
wait until rising_edge(clk);
wait until falling_edge(clk);
assert( done ='H')
report "Done is "&std_logic'image(done)&", verwacht 'H'"
severity error;
wait until rising_edge(pwm);
pwm_start<=now;
wait until falling_edge(pwm);
pwm_stop<=now;
wait until falling_edge(clk);
--assert pwmsignaal juist (idle positie)
assert(abs((pwm_stop-pwm_start)-idle_time)<sclkPeriod*tol)
report "Fout PWM signaal: "& time'image(pwm_stop-pwm_start)&" /= "&time'
    image(idle_time)
severity error;
wait until falling_edge(clk);
assert(done ='H')
report "Done is "&std_logic'image(done)&", verwacht 'H'"
severity error;
wait for 1 ms;


--set =0 voor volgende puls
report "no position sent after address sent (set ='0' voor volgende klokperiode)";
wait until rising_edge(clk);
set <='H';
data <=std_logic_vector(to_unsigned(1,data'length)); -- address sturen
wait until falling_edge(clk);
assert(done = 'H')
report "Done is "&std_logic'image(done)&", verwacht 'H'"
severity error;
wait for 5 ms;
set <='L';
wait until falling_edge(clk);
assert(done ='H')
report "Done is "&std_logic'image(done)&", verwacht 'H'"
severity error;
wait until rising_edge(clk);
wait until falling_edge(clk);
```

```
  assert( done ='H')
  report "Done is "&std_logic'image(done)&", verwacht 'H'"
  severity error;
  wait until rising_edge(pwm);
  pwm_start<=now;
  wait until falling_edge(pwm);
  pwm_stop<=now;
  wait until falling_edge(clk);
  --assert pwmsignaal juist (idle positie)
  assert(abs((pwm_stop-pwm_start)-idle_time)<sclkPeriod*tol)
  report "Fout PWM signaal: "& time'image(pwm_stop-pwm_start)&" /= "&time'
      image(idle_time)
  severity error;
  wait until falling_edge(clk);
  assert(done ='H')
  report "Done is "&std_logic'image(done)&", verwacht 'H'"
  severity error;
  wait for 1 ms;
```

```
  rst<='1';
  wait until falling_edge(clk);
  rst<='0';
  wait until rising_edge(clk);
-- normale werking
  plaats <= (others => '0');
  wait for 1 ms;
  while (plaats < 60) loop
    report "normale werking zonder testen, plaats is "&integer'image(to_integer(
        plaats));
    wait until rising_edge(clk);
    set <='H';
    data <=std_logic_vector(to_unsigned(1,data'length)); -- address sturen
    wait until rising_edge(clk);
    data <= std_logic_vector(plaats(7 downto 0)); -- positie sturen
    wait until rising_edge(clk);
    set <= 'L';
    wait until falling_edge(clk);
    plaats <= plaats+27;

    wait for 1 ms;
  end loop;
  wait until rising_edge(pwm);
  pwm_start<=now;
```

```
    wait until falling_edge(pwm);
    pwm_stop<=now;
    wait until falling_edge(clk);
    --assert pwmsignaal juist (54)
    assert(abs((pwm_stop-pwm_start)-(min_time+ 54*sclkPeriod))<sclkPeriod*tol)
    report "Fout PWM signaal: "& time'image(pwm_stop-pwm_start)&" /= "&time'
        image(min_time+54*sclkPeriod)
    severity error;
    wait for 1 ms;

    EndOfSim <= true;
    wait;
end process;


end architecture;
```

Conclusie