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 afkortingen

DUT Device Under Test.

FSM Finite State Machine.

PWM Pulse Width Modulation.

VHDL Very High Speed Integrated Circuit Hardware Description Language.

Inleiding

De opdracht is om in Very High Speed Integrated Circuit Hardware Description Language (VHDL) een controller voor een servomoter te ontwerpen. Er moet ook een testbench ontwikkeld worden, om de controller te testen. Het is belangrijk om rekening te houden met de vereisten van de klant (zie sectie 1) 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 1,25 ms en 1,75 ms. 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. In het geval van asynchrone reset, zou het signaal onbedoeld langer kunnen worden dan de maximum pulsduur (1,75 ms).

De controller werkt met een klok van 50 Hz en een servoklok van 510 kHz. Hiermee rekening houdend blijkt dat de minimum pulsduur van 1,25 ms overeen komt met 637,5 servoklok ticks, het midden van 1.5 ms komt overeen met 765 servoklok ticks en de maximum pulsduur van 1,75 ms 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 servocontroller heeft een adres. Als er een positie gestuurd wordt die vooraf gegaan wordt door het correcte adres, dan moet de controller deze instructie uitvoeren. Indien een ander adres vermeld wordt dan moet de controller de opdracht negeren.
Ook is er een *broadcast* adres waar alle controllers naar moeten luisteren, namelijk 255. Als een signaal uitgestuurd wordt met het *broadcast* adres dan moeten alle aangesloten controllers hetzelfde PWM signaal generen.

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 90°, 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 *tri-state logic*, dit betekend dat het op een bus aangesloten kan worden. Hierdoor kunnen meerdere controllers gebruik maken van éénzelfde 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

De *entity* van de servocontroller beschrijft de poortmapping. De servocontroller neemt clk (clock), rst (reset), sclk (servoclock), set en data als inputs. De servocontroller zal het *done* signaal en het aanstuursignaal (pwm) als uitput genereren. Er is voor gekozen om een generische variabele adres toe te voegen, zodat de code kan hergebruikt worden voor verschillende adressen.

```
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 de hulpsignalen cnt en pwmi kunnen we een PWM signaal generen (zie gen_pwm). Daarna worden de nodige types en signalen aangemaakt om een FSM te kunnen gebruiken. Eerst worden de verschillende staten gedefinieerd waarna de signalen voor het bijhouden van de staat (currentState en nextState) 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. De transities worden beschreven in volgend flowchart.

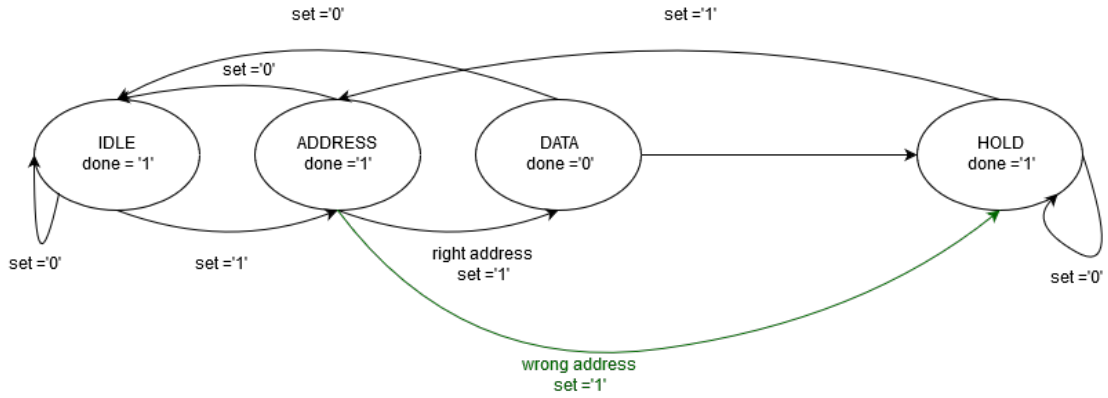

Figure 1: Transitiediagram voor de FSM van de servocontroller

Voordat de controller het PWM signaal aanmaakt, moet er gecontroleerd worden of het doorgestuurde adres enerzijds het *broadcast* adres is, of anderzijds het adres is van de servomotor. Anders mag de controller zijn output niet aanpassen. Indien het set-signaal onderbroken wordt, voordat er een nieuwe positie is doorgestuurd, moet de controller naar de neutrale positie (idle) 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 het proces dat de overgangen van de staten dicteert. Hier wordt er ook voor gezorgd dat bij een reset de controller altijd terug gaat naar neutrale (idle) positie, ongeacht de status van de andere signalen.

```
-- 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 het *done* signaal zich gedraagt in elke staat. In deze setup heeft done een *tri-state logic*. Het PWM signaal wordt hier niet gedefinieerd omdat er een one-line process gebruikt wordt om dit aan de uitgang te koppelen (zie gen_pwm).

```
-- 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 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 eigenlijke puls genereert. Het aantal servoklok ticks wordt bijgehouden en vergeleken met de benodigde aantallen die in pwm_data bepaald zijn. Cnt moet gereset worden bij elke klok tick (50 Hz).
Nadien wordt met een one-line process het pwm signaal aan de output gekoppeld.

```
-- 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 voor het uittesten van de geschreven architectuur. De vereiste van de testbench is dat deze de klok en de servoklok genereert. De servocontroller moet uitgetest worden in stappen van 32, hierbij moet het PWM signaal gecontrollerd worden. De servoklok mag niet gebruikt worden om het PWM signaal te controleren.

De testbench heeft zelf geen input en *outputs*, daarom is de *entity* van de testbench leeg. In de architectuur van de testbench moeten alle in- of outputsignalen van de servocontroller aangemaakt worden, zodat deze later met het Device Under Test (DUT) kunnen verbonden worden.

Verder worden ook enkele constanten aangemaakt die betrekking hebben tot de timing van de testbench. Enerzijds moeten er twee kloksignalen gegenereerd worden: De algemene klok en de servoklok. Hiertoe wordt een klokperiode (clkPeriod) en een servoklokperiode (sclkPeriod) gedefinieerd. Anderzijds worden min_time en idle_time gedefinieerd. Dit zijn respectievelijk de minimum duur van het PWM signaal en de duur van het PWM signaal in neutrale positie.

Tot slot worden er nog een aantal hulpsignalen aangemaakt. EndOfSimulation is het signaal dat true wordt wanneer de simulatie klaar is. Plaats is het signaal dat gebruikt wordt om een positie aan de servocontroller door te geven. Pwm_start en pwm_stop zijn hulpsignalen om de timing van 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;
constant dutyCycle: real := 0.5;
constant idle_time: time:= 1.5 ms;
constant min_time: time:= 1.25 ms;
constant tol: real:= 1.0;
--end of simulation
signal EndOfSim: boolean:= false;


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

Na het definiëren van de signalen, begint de eigenlijke architectuur van de testbench. Als eerste stap wordt het DUT op de juiste manier verbonden. Er is voor gekozen om de te testen servocontroller het adres 1 te geven.

```
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
);
```

In een volgende stap worden de twee kloksignalen gegenereerd, die nodig zijn om de servocontroller aan te sturen.

```
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;
```

Het laatste proces, zal inputsignalen sturen naar de servocontroller en de output controleren. Dit proces kan in verschillende delen onderverdeeld worden.
Eerst wordt de normale werking getest. Posities van 0 tot 255 worden aangelegd, in stappen van 32. Tijdens de opdracht wordt de staat van het *done* signaal gecontroleerd. Na een opdracht wordt er gecontroleerd of het PWM signaal correct is. Het verwachte gedrag wordt geïllustreerd in figuur 2.

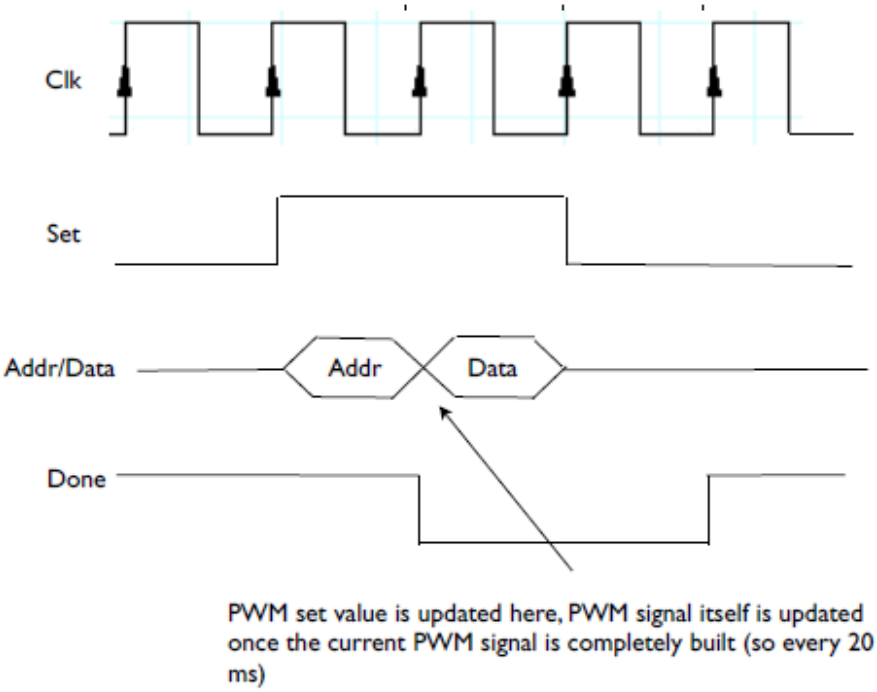

Figure 2: Timing diagram voor de servocontroller

De timing wordt getest met behulp van de *assert/report/severity commands* van VHDL. Bij een stijgende flank van de klok wordt *set* hoog gezet, en wordt er naar data het adres 1 gestuurd. Op de dalende flank van de klok wordt gecontroleerd of *done* hoog staat, anders wordt een error gerapporteerd. Na de volgende stijgende klokflank wordt de gewenste plaats naar data gestuurd. *done* moet nu laag zijn, dit wordt opnieuw getest. Op de volgende stijgende flank moet *set* terug op laag gezet worden. Er wordt opnieuw gecontroleerd dat *done* laag is. Het PWMsignaal moet nog gecontroleerd worden. Pwm_start is de timestamp van de eerstvolgende stijgende flank van het PWMsignaal, pwm_stop is de timestamp van de eerstvolgende dalende flank. Door het verschil te berekenen tussen deze twee momenten, en te controleren met de verwachte duur van het signaal, wordt de juistheid van het PWMsignaal bepaald. Een verschil van de servoklokperiode wordt getolereerd als fout, omdat de servoklok de maximale resolutie bepaald. Op dit moment wordt *done* voor de laatste keer getest, *done* moet nu hoog zijn. Dit proces wordt herhaald voor de verschillende posities.

```
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;
```

Na de normale werking getest te hebben, moet er gecontroleerd worden of de servocon-

troller correct reageert op een *reset*. Nadat het *set* signaal hoog is gezet en het juiste adres is aangelegd, wordt het *reset* signaal korte tijd hoog gezet. Er wordt een plaats aangelegd, maar de controller mag niet naar deze plaats bewegen, in plaats daarvan moet de controller naar de neutrale (*idle*) positie gaan.

```
--reset test (plaats =224)
plaats <= to_unsigned(224,9);
wait for 1 ms;
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;
```

Vervolgens wordt getest of de servocontroller ook luistert naar het *broadcastaddress*. De structuur is identiek zoals in de normale werking test, maar het adres is nu 255.

```
--Test Broadcast address (plaats =224)
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;
```

Een volgende stap is het testen wat er gebeurt als de servocontroller een adres ontvangt dat niet zijn adres is, m.a.w. als de positiewijziging voor een andere servomotor bedoeld is. Het adres dat aangelegd wordt, is 2, de positie is 32°. Er wordt gecontroleerd dat de servomotor nog hetzelfde PWMsignaal geeft, nl. het signaal dat in de vorige stap (test broadcastadres) bepaald is.

```
--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;
```

Er kan een situatie ontstaan, waar de servocontroller wel een adres krijgt, maar geen positie. In dit geval zal *set* vroegtijdig laag worden. De servomotor moet dan terugvallen naar zijn reset positie. Er wordt getest wat er gebeurd als *set* laag wordt op de volgende stijgende klokflank: *done* zal even laag worden, totdat de servocontroller beseft dat er geen verdere instructie komt. Op dat moment wordt de positie naar neutraal gebracht. Wanneer *set* vroeger laag wordt, zal de servocontroller bij de volgende stijgende klokflank onmiddellijk zien dat er geen verdere informatie komt en naar de neutrale positie gaan.

```
--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;
```

Tot slot wordt er getest of de servocontroller nog steeds correct wordt wanneer er een nieuwe opdracht wordt doorgegeven onmiddellijk na de vorige positiewijziging is afgerond. Omdat er in de vorige testen, telkens gecontroleerd werd na een verandering, kunnen er zo fouten in de reactietijd verborgen worden. Enkel de laatste positie wordt gecontroleerd.
Op het einde van de simulatie wordt EndOfSim true.

```
  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

Er werd een servocontroller ontworpen in VHDL. Deze controller moest voldoen aan verschillende specificaties, zowel specificaties uit de oorspronkelijke opdracht, als verdere specificaties besproken met de begeleider: maximale stappen van 32 zetten, resetten wanneer er zich fouten voordoen in de communicatie, de huidige positie behouden wanneer er geen communicatie is en er is een *broadcast* adres beschikbaar. Er werd ook een *testbench* voor de servocontroller geschreven die bovenstaande specificaties test.

Door de testbench een beetje aan te passen, kan men aan tonen dat de controller ook in staat is om zelf stappen van 2 toe zetten. Men kan de controller dus ook gebruiken om een grotere precisie te bekomen dan de vereisten oplegden.