

Verslag

digitale techniek: PID regelaar

03-2023

Harm Hongerkamp

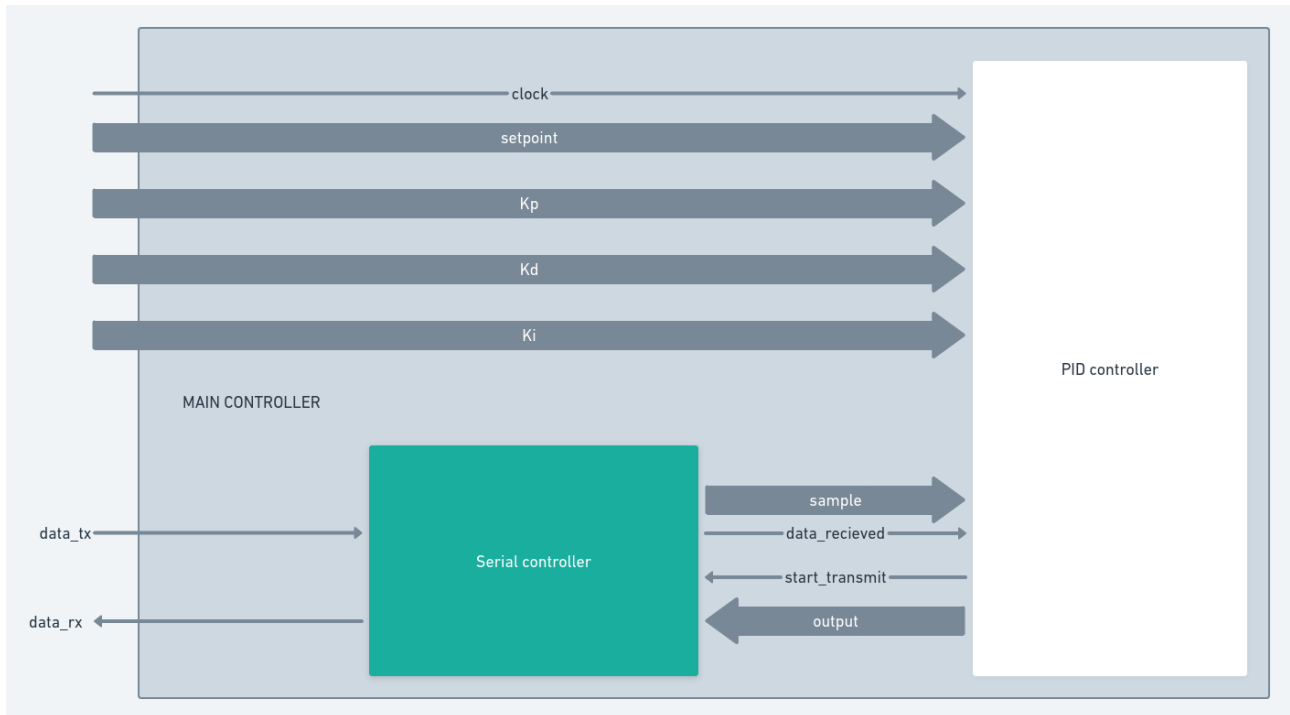
jaar 4

inhoudsopgave

1. Inleiding.....	3
2. Opbouw opdracht.....	4
2.1. PID regelaar.....	5
2.2. Serieel.....	7
2.3. Main.....	9
3. Simulatie.....	10
3.1. Transmit.....	10
3.2. Recieve.....	11
3.3. Clock.....	12
3.4. Main.....	13

1. Opbouw opdracht

De PID regelaar bestaat uit verschillende delen: De regelaar zelf, een serieel naar parallel omzetter en een parallel-serie omzetter. Er zijn in verschillende entiteiten gemaakt voor de totale controller: een voor de PID regelaar, een voor de serieele omzettingen en een waarin de 2 voorgaande bij elkaar komen.



Figuur 1: blokschema PID controller project

1.1. PID regelaar

Een PID regelaar wordt als volgt wiskundig weergegeven:

In VHDL komt dit neer op:

P-actie: $P = error * K_p$

I-actie: $I = K_i * (error(n) + error(n^{-1}))$

D-actie: $D = K_d * (error(n) - error(n^{-1}))$

Waarbij error bepaald wordt met: $error(n) = setpoint - gemeten$.

Dit is vrij makkelijk in een process in te voeren:

```
1  error: process(clock)
2  begin
3      if rising_edge(clock) then
4          case state is
5              when IDLE =>
6                  if data_recieved = '1' then --wait until data is recieved
7                      state <= CALCULATE_ERROR;
8                      start_transmitting <= '0';
9                  end if;
10             when CALCULATE_ERROR =>
11                 prev_err <= current_err;
12                 current_err <= to_integer(setpoint)- to_integer(sample);
13                 state <= DETERMINE_ACTIONS;
14             when DETERMINE_ACTIONS =>
15                 P_action <= to_integer(Kp)* current_err;
16                 I_action <= to_integer(Ki)*(current_err+prev_err);
17                 D_action <= to_integer(kd)*(current_err-prev_err);
18                 state <= CALCULATE_ACTION;
19             when CALCULATE_ACTION =>
20                 total_action <= to_signed((P_action+I_action+D_action), 8);
21                 state <= SET_OUTPUT;
22             when SET_OUTPUT =>
23                 if transmit_ready = '1' then --wait for transmit logic te be ready
24                     start_transmitting <= '1';
25                     output <= sample +total_action;
26                     state <= IDLE;
27                 end if;
28             when OTHERS =>
29                 state <= IDLE;
30             end case;
31         end if;
32     end process;
```

Figuur 2: PID process

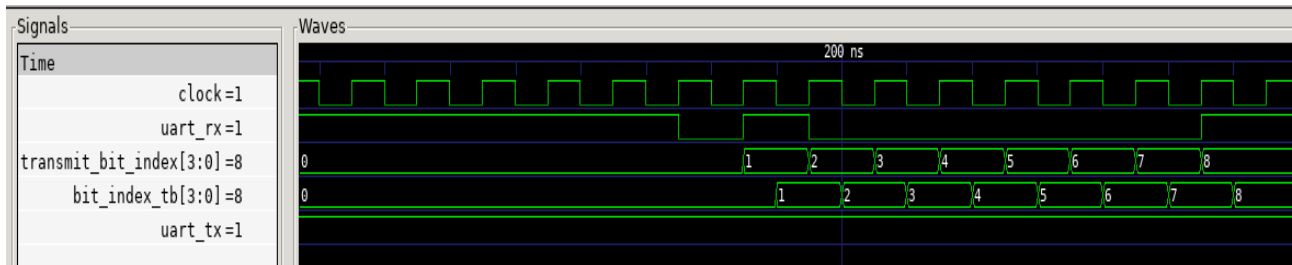
Ook hier is gekozen om een state-machine te maken om te zorgen dat alle berekeningen op het juiste moment gebeuren.

Wanneer een nieuw sample is binnengekomen, *data_recieved* wordt hoog gemaakt door de seriele ontvanger, wordt de nieuwe error waarde berekend.

Vanuit de errors kunnen vervolgens de P, I en D acties berekent worden, al deze acties worden vervolgens bij elkaar opgeteld en de som vormt de uitgangswaarde. Wanneer deze waarde klaar staat zet de PID controller het signaal *data_ready* hoog zodat de parallel naar seriele omzetter de output kan versturen.

1.2. Serieel

Voor het seriele protocol is gekozen voor een variatie van USART, dit staat voor: Universal Synchronous/Asynchronous Receiver/Transmitter. Het protocol is hetzelfde als UART met als verschil dat er een klok signaal is toegevoegd, dit maakt het ontwerp van de blokken een stuk makkelijker.



Figuur 3: waveform dataprotocol

Zoals te zien is in Figuur 3 zijn de data signalen hoog in rust. Wanneer er data klaarstaat om verzonden te worden, wordt een start conditie verzonden door de tx datalijn (gezien vanuit de zender) op de opgaande flank van de klok op 0 te zetten. Het *transmitting* signaal is hoog om de tijd aan te geven dat er data vanuit de testbench de uart entity in gestuurd wordt.

Vervolgens worden op de volgende 8 opgaande klokflanken de bits verstuurd. Dit gaat van de minst belangrijke bit naar de meest belangrijke bit. Op de neergaande klokflank word aan de ontvanger kant de inkomende bits gelezen (te zien aan de verandering van het *bit_index_tb* signaal) en op de juiste plek in het ontvangst register gezet.

De zender stuurt vervolgens een stop bit door de data lijn hoog te zetten voor 1 klokcyclus.

Wanneer een byte verzonden is wordt de *ready_for_transmit* signaal hoog gezet zodat de controller weet wanneer er nieuwe data verzonden kan worden. De ontvanger heeft eenzelfde signaal genaamd *data_recieved*, dat aan de controller aangeeft dat er een byte ontvangen is.

```

1  recieve: process(clock)
2  begin
3      --process to recieve date, note: process is dependent on cloksignal that has the same frequency as the baudrate.
4      if falling_edge(clock) then
5          case recieve_state is
6              when IDLE =>
7                  recieved_byte <= x"00";
8                  bit_index <= x"0";
9                  data_recieved <= '0';
10                 if rx = '0' then --start condition detected;
11                     recieve_state <= RECIEVE_BYTE;
12                 end if;
13             when RECIEVE_BYTE => --recieve byte
14                 if bit_index /= x"7" then
15                     recieved_byte(to_integer(bit_index)) <= rx;
16                     bit_index <= bit_index +1;
17                 else
18                     recieve_state <= SEND_RECIEVED;
19                 end if;
20             when SEND_RECIEVED =>
21                 D_out <= recieved_byte;
22                 recieve_state <= IDLE;
23                 data_recieved <= '1';
24             when others =>
25                 recieve_state <= IDLE;
26         end case;
27     end if;
28 end process;
29
30 transmit: process(clock)
31 begin
32     if rising_edge(clock) then
33         case transmit_state is
34             when IDLE =>
35                 tx <= '1';
36                 ready_for_transmit <= '1';
37                 transmit_bit_index <= x"0";
38                 if start_transmit = '1' then --wait for start command
39                     transmit_byte <= D_in;
40                     transmit_state <= SEND_BYTE;
41                     tx <= '0'; --send start condition
42                     ready_for_transmit <= '0';
43                 end if;
44             when SEND_BYTE =>
45                 if transmit_bit_index /= x"7" then
46                     tx <= transmit_byte(to_integer(transmit_bit_index));
47                     transmit_bit_index <= transmit_bit_index+ x"1";
48                 else
49                     tx <= '1'; --transmit stop bit before going idle
50                     transmit_state <= IDLE;
51                 end if;
52             when others =>
53                 transmit_state <= IDLE;
54         end case;
55     end if;
56 end process;
57

```

Figuur 4: data ontvang en verzend processen

1.3. Main

De main entity brengt de 2 voorgaande entities samen.

Hierdoor bestaat de architectuur voornamelijk uit port maps:

```
1  entity hardware is
2      port(
3          rx : in std_logic;
4          tx : out std_logic;
5          setpoint: in signed (7 downto 0);
6          Ki,Kp,Kd: in signed(7 downto 0);
7          clock: in std_logic
8      );
9  end entity;
10
11 architecture main of hardware is
12     component PID is
13     port(
14         sample : in signed(7 downto 0);
15         setpoint: in signed (7 downto 0);
16         output: out signed(7 downto 0) := x"00";
17         Ki, Kd, Kp: in signed (7 downto 0);
18         clock :in std_logic;
19         transmit_ready: in std_logic;
20         data recieved: in std_logic;
21         start_transmitting: out std_logic := '0'
22     );
23 end component;
24 for PID0: PID use entity work.PID;
25
26     component uart is
27     port(
28         rx: in std_logic;
29         D_out: out std_logic_vector(7 downto 0):=x"00";
30         clock: in std_logic;
31         D_in: in std_logic_vector(7 downto 0);
32         tx: out std_logic:= '1';
33         data recieved: out std_logic;
34         start_transmit: in std_logic;
35         ready_for_transmit: out std_logic
36     );
37 end component;
38 for uart0: uart use entity work.uart;
39 signal DATA1, DATA2: std_logic_vector(7 downto 0) := x"00";
40 signal data_recieved1 : std_logic;
41 signal ready_for_transmit1 : std_logic;
42 signal start_transmission1: std_logic;
43 begin
44     PID0: PID port map(sample => signed(DATA1), std_logic_vector(output) => DATA2, clock => clock, setpoint => setpoint,
45                       Ki => Ki, Kp=>Kp, Kd => Kd, data_recieved=>data_recieved1,transmit_ready => ready_for_transmit1, start_transmitting => start_transmission1);
46     uart0: uart port map(rx => rx, tx=>tx, D_out => DATA1, D_in => DATA2, clock => clock, data_recieved => data_recieved1,
47                       ready_for_transmit=> ready_for_transmit1, start_transmit => start_transmission1);
48 end architecture;
```

Figuur 5: code main architecture

2. Simulatie

Voor de simulatie is een testbench geschreven. Deze bestaat uit meerdere processen:

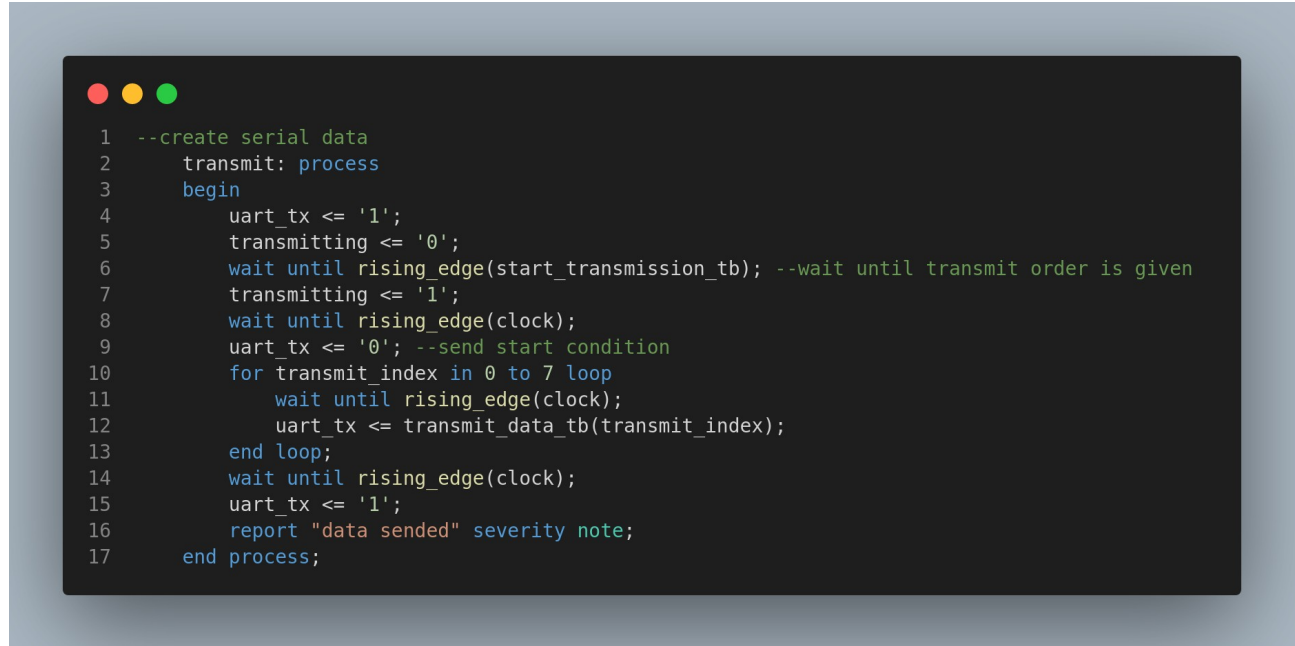
- Clock.
- Recieve.
- Transmit.
- Main.

Dit is gedaan om bepaalde stukken te automatiseren zodat de simulatie makkelijk gaat.

2.1. Transmit

Het transmit proces heeft dezelfde functie als de seriële controller in van hoofdstuk 2.2.

Het zet dat de datavectors die de samples bevatten omgezet worden in seriële data dat naar de PID controller gaat.



```
1  --create serial data
2  transmit: process
3  begin
4      uart_tx <= '1';
5      transmitting <= '0';
6      wait until rising_edge(start_transmission_tb); --wait until transmit order is given
7      transmitting <= '1';
8      wait until rising_edge(clock);
9      uart_tx <= '0'; --send start condition
10     for transmit_index in 0 to 7 loop
11         wait until rising_edge(clock);
12         uart_tx <= transmit_data_tb(transmit_index);
13     end loop;
14     wait until rising_edge(clock);
15     uart_tx <= '1';
16     report "data sended" severity note;
17 end process;
```

Figuur 6: transmit sample process

2.2. Recieve

Het recieve proces ontvangt de output data van de PID controller en plaatst het in een vector.

Hierdoor kan de seriële data die de PID controller uitstuurt makkelijk op het scherm weergegeven worden of bewerkt worden zodat een systeem gesimuleerd kan worden.

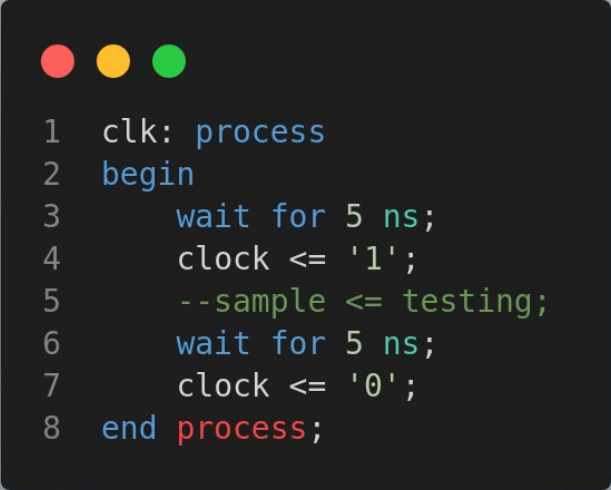
De uitvoering van de reciever is precies hetzelfde als beschreven in hoofdstuk 2.2.

```
1  recieve: process(clock)
2      begin
3          if falling_edge(clock) then
4              case recieve_state_tb is
5                  when IDLE =>
6                      recieved_byte_tb <= x"00";
7                      bit_index_tb <= x"0";
8                      if uart_rx = '0' then --start condition detected;
9                          recieve_state_tb <= RECIEVE_BYTE;
10                         data_recieved_tb <= '0';
11                     end if;
12                 when RECIEVE_BYTE => --recieve byte
13                     if bit_index_tb /= x"7" then
14                         recieved_byte_tb(to_integer(bit_index_tb)) <= uart_rx;
15                         bit_index_tb <= bit_index_tb +1;
16                     else
17                         recieve_state_tb <= SEND_RECIEVED;
18                     end if;
19                 when SEND_RECIEVED =>
20                     data_recieved_tb <= '1';
21                     recieve_data_tb <= signed(recieved_byte_tb);
22                     report "data recieved" severity note;
23                     recieve_state_tb <= IDLE;
24                 when others =>
25                     recieve_state_tb <= IDLE;
26             end case;
27         end if;
28     end process;
```

Figuur 7: VHDL code serieel onvangst process

2.3. Clock

Het clockprocess zorgt ervoor dat er een kloksignaal loopt. Op basis van deze klok wordt onder andere data verstuurt.



```
1 clk: process
2 begin
3     wait for 5 ns;
4     clock <= '1';
5     --sample <= testing;
6     wait for 5 ns;
7     clock <= '0';
8 end process;
```

Figuur 8: klokprocess

2.4. Main

In het main process word de daadwerkelijke test uitgevoerd.

Als eerste word de setpoint bepaalt. Daarna worden de P-, I- en D-acties vastgezet, vervolgens wordt een sample gestuurd.

Daarna wordt in een FOR loop voor 100 samples een systeem gesimuleerd.

In dit geval is het een simpel systeem waarvan de output 5 samples achterloop op de ingang.

De waarde van de samples die de controller ingaan en die controller uitstuurt worden vervolgens op het scherm geprint waardoor controle vrij makkelijk kan verlopen.

```
1  main: process
2  begin
3      --start values
4      Ki <= x"00";
5      Kp <= x"00";
6      Kd <= x"01";
7      setpoint <= x"0f";
8      wait for 10 ns;
9      transmit_data_tb <= x"56";
10     start_transmission_tb <= '1';
11     wait until rising_Edge (transmitting);
12     start_transmission_tb <= '0';
13     for loop_tb in 0 to 100 loop
14         wait until rising_edge(data_recieved_tb);
15         report "serial transmitted: " & integer'image(to_integer(transmit_data_tb)) severity note;
16         report "setpoint: " & integer'image(to_integer(setpoint)) severity note;
17         report "serial recieved: " & integer'image(to_integer(recieve_data_tb)) severity note;
18         transmit_data_tb <= delay5;
19         delay5<=delay4;
20         delay4<=delay3;
21         delay3<=delay2;
22         delay2<=delay1;
23         delay1<=recieve_data_tb;
24         start_transmission_tb <= '1';
25         wait until rising_Edge (transmitting);
26         start_transmission_tb <= '0';
27     end loop;
28     report "end of sim" severity failure;
29 end process;
```

Figuur 9: main test process

3. resultaten

Draaien van de testbench gaf de volgende output (deze is terug te vinden in het log.txt bestand):

```
6  main.vhdl:163:13:@280ns:(report note): serial transmitted: 86
7  main.vhdl:164:13:@280ns:(report note): setpoint: 15
8  main.vhdl:165:13:@280ns:(report note): serial recieved: -127
9  main.vhdl:163:13:@550ns:(report note): serial transmitted: 0
10 main.vhdl:164:13:@550ns:(report note): setpoint: 15
11 main.vhdl:165:13:@550ns:(report note): serial recieved: 45
12 main.vhdl:163:13:@820ns:(report note): serial transmitted: 0
13 main.vhdl:164:13:@820ns:(report note): setpoint: 15
14 main.vhdl:165:13:@820ns:(report note): serial recieved: 45
15 main.vhdl:163:13:@1090ns:(report note): serial transmitted: 0
16 main.vhdl:164:13:@1090ns:(report note): setpoint: 15
17 main.vhdl:165:13:@1090ns:(report note): serial recieved: 45
18 main.vhdl:163:13:@1360ns:(report note): serial transmitted: 0
19 main.vhdl:164:13:@1360ns:(report note): setpoint: 15
20 main.vhdl:165:13:@1360ns:(report note): serial recieved: 45
21 main.vhdl:163:13:@1630ns:(report note): serial transmitted: 0
22 main.vhdl:164:13:@1630ns:(report note): setpoint: 15
23 main.vhdl:165:13:@1630ns:(report note): serial recieved: 45
```

Figuur 10: deel van de simulatie output

In Figuur 10 is een deel van de output weergegeven.

Transmitted betekend dat de data van de testbench naar de controller gestuurd is, recieved betekent dat de data van de controller naar de testbench gaat. De verdere beschrijving is vanuit het oogpunt van de testbench.

Op regelnummer 5 van Figuur 10 wordt de eerste sample verstuurd. Deze heeft de waarde van 86, dit geeft dan een error waarde van $15 - 86 = -71$. Aangezien dit de eerste meting is is de voorgaande error 0.

hierdoor komen alle acties op dezelfde waarde uit namelijk: -71. dit maakt de totale error -213. wanneer dit van de originele sample afgetrokken wordt komt dit neer op -127. wat er daadwerkelijk ontvangen wordt op regel 8 van Figuur 10.

De volgende sample is 0 wat de error 15 maakt. Dit maakt:

$P = 15$, $I = (15 + -71) = -56$, $D = (15 - -71) = 86$, de totale actie is dan: 45. omdat de sample 0 is word de output dus ook 45. Dit is te zien op regel 11.

Met deze voorbeelden is dus te zien dat alle onderdelen van de regelaar naar behoren werkt.