By:
Michael Vestergaard Jessen
Jimmy Alison Jørgensen

1 PID controller

1.1 Introduction

The PID IP module connects to the OPB-bus through 32 bit software read/write registers. The parameters K_p, K_i and K_d are configurable through register read/writes and the width of each are specified with the DATA_WIDTH generic. Fixed point unsigned numerics are used and the bit position of the point (comma) is determined by the FIXED_POINT_POS generic. Another input is the reference input or setpoint value. This is defined as a signed integer with bit width given by the generic INPUT_WIDTH. The feedback or sensor input is the input that the PID wil try to control. This input is defined like the reference input as a signed interger with width INPUT_WIDTH.

1.2 Design discussion

What is a PID. How does work. What does it do and where is it used.
Why Fixedpoint, How fixedpoint.... Why not FLoating point, how floatingpoint. Why not choose static 32 bit width? Why combine the addr register and konst register into one?

1.2.1 A PID

A PID (Proportional Integral Control) is perhaps the most used method of controlling systems. That is keeping some system at a desired set-point.

1.2.2 Floating point vs. Fixed Point

Our implementation use fixed point arithmetics instead of floating point arithmetics. This choice was made because of the simpler representation and that fixed point arithmetic resembles integer arithmetics which are supported in VHDL. A fixed point number is represented as a binary number split by a radix point. The bits to the left of the radix point are called magnitude bits and describe the integer part of the fixed point value. The bits to the right of the radix point are fractional bits representing the fractional value of the fixed point number. Consider the fixed point binary number $abcd.efgh$ where each letter represents either 0 or 1. Equation 1 describes how to calculate the integer part and fractional part of that number.

$$integerpart = a \cdot 2^3 + b \cdot 2^2 + c \cdot 2^1 + d \tag{1}$$

$$fractionalpart = \frac{e}{2^1} + \frac{f}{2^2} + \frac{g}{2^3} + \frac{h}{2^4} \tag{2}$$

When the radix point of each fixed point number are aligned, adding and subtracting is pure integer operations. Multiplication of fixed point numbers can also be done with integer multiplication as long as the position of the radix point of the result are changed accordingly.

1.3 Design constraints

1.4 Features

- 32-bit OPB slave interface.
- Up to 8 PID driver instances.
- Programmable sample time.

1.5 Design implementation

The PID is configured and used through 4 32 bit registers.

- PIDPR - Configuration of PID internal konstants.
- PIDIN - feedback input to the pid
- PIDRIN - reference input to the pid
- PIDOUT - output of the pid.

Reading and writing from the PID. There are 5 signals that can be written and one is read-only:

- Kp - Konstant that defines the proportional gain. This is only writable and is defined as a fixed point unsigned value.
- Ki - Konstant that defines the integral gain. This is only writable and is defined as a fixed point unsigned value.
- Kd - Konstant that defines the derivative gain. This is only writable and is defined as a fixed point unsigned value.
- Reference Input - The wanted input. The PID wil minimize the difference between this input and the feedback input. This register is only writable.
- Feedback input - This input is the feedback from the system that we want to control.
- Output - This is the output of the PID and input to the system that we wish to control.

The signals Kp, Ki and Kd is changed through register . The first 2 bits in the 32 bit register determines what signal is changed.

bit	[32:2]	1	0	
	data	PR01	PR00	PIDPR
Read/Write	W	W	W	
Initial Value	0	0	0	

PRS01	PRS00	Description
0	0	NC
0	1	data is written to K_P
1	0	data is written to K_I
1	1	data is written to K_D

Table 1: Configuration of PID konstants.

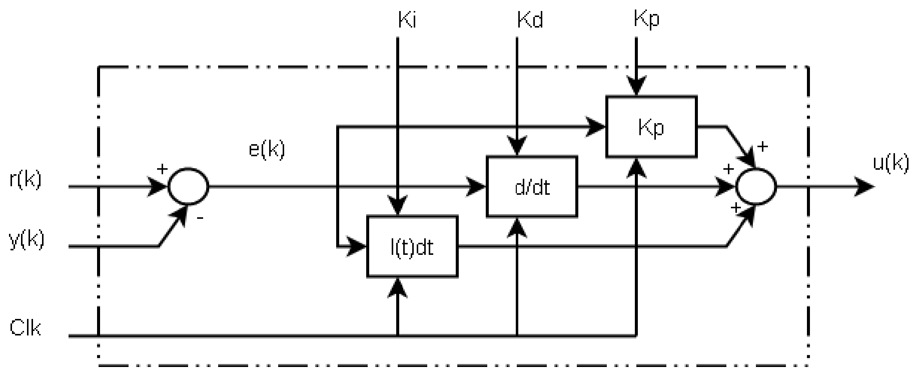

Figure 1: Diagram of the PID.

1.6 Implementation

In this section we will describe the implementation of the PID and not the OPB wrapper. The complete implementation is contained in the file ”PIDControl.vhd” with the entity name PIDControl. An instance of the entity PIDControl can control up to 32 synchronous PID instances. One output and one input signal controls the calculation and data flow.

Each PID has three inputs for the data flow: a Sample clk, a feedback data input and a reference data input. The width of the feedback and reference inputs are given by the generic C_INPUT_WIDTH.

Since multiplication is quite expensive in gates and resources the multiplication has been pipelined. Not only for each PID but for all PIDs in the PIDControl instance. Three multiplications are used for calculating the output in a PID. One PID calculation is done in 4 steps:

1. Load multiplier registers with K_p and $err(t)$.
2. Save multiplication result in p_{tmp}. Load multiplier registers with K_d and $err(t) - err(t-1)$.
3. Save multiplication result in d_{tmp}. Load multiplier registers with K_i and $integralsum$.
4. Save multiplication result in i_{tmp}. Save $p_{tmp} + d_{tmp} + i_{tmp}$ in output.

Between each step the multiplication is done. The above steps are repeated m times if m PIDs are used.

The drawback of using a pipelined model is the speed reduction. The maximum sample frequency is given by equation 3 where CLK_{pid} is the PIDControl clk and N_{inst} is the number of PID instances.

$$SampleFreq_{max} = \frac{CLK_{pid}}{4 \cdot N_{inst}} \tag{3}$$

$$output(t) = K_p * err(t) + K_i * \sum_{i=0}^{t}(err(i)) + K_d * (err(t) - err(t-1)) \tag{4}$$

Algorithm 1.1 Clamped PID calculation

```
Require: Input -
Require: RefInput - a bounding rectangle of the border
Require: Constants - Constants $K_p$, $K_i$, $K_d$.
Require: lastError - Error from last sample.
Require: errorSum - Sum of all sampled errors times $K_i$.
Require: max,min - Clamp max and min values.
1: $error \leftarrow RefInput - Input$
2: $errorSum \leftarrow errorSum + K_i \cdot error$
3: $result \leftarrow K_p \cdot error + errorSum + K_d \cdot (error - lastError)$
4: $lastError \leftarrow error$
5: if $result < min$ then
6:     $result \leftarrow min$
7: else if $result > max$ then
8:     $result \leftarrow max$
9: return $result$
```

Notice how the errorSum is accumulated. The obvious implementation of the integral gain calculation, see 1.2, multiplies the $errorSum$ with K_i. It is cheaper since multiplication is expensive and the register containing $errorSum$ is much bigger than the register containing $error$

Algorithm 1.2 Obvious integral gain calculation

```
1: $errorSumum \leftarrow errorSum + error$
2: $Result_{integral} \leftarrow K_i \cdot errorSum$
```