XAPP1163 (v1.0) January 23, 2013

Floating-Point PID Controller Design with Vivado HLS and System Generator for DSP

Author: Daniele Bagni, Duncan Mackay

Summary

This application note describes how to quickly implement and optimize floating-point Proportional-Integral-Derivative (PID) control algorithms specified in C/C++ code into an RTL design using Vivado HLS. You can use System Generator for DSP to easily analyze and verifiy the design. This enables floating-point algorithm designers to take advantage of high-performance, low cost, and power efficient Xilinx® FPGA devices.

Introduction

Floating-point algorithms are widely used in industries from analysis to control applications. Traditionally, such algorithms have been implemented on microprocessors. The primary reason for using microprocessors has been the ease with which floating-point algorithms can be captured, validated, and debugged in C/C++ code, therefore avoiding the complexity and skills required to implement them in hardware. However, implementing these algorithms on optimized and dedicated hardware provides lower cost, higher performance, and power benefits over a standard, or even optimized microprocessor.

This application note presents a new design flow enabled by the Xilinx Vivado™ Design Suite, which allows floating-point algorithms to be quickly specified in C/C++ code, optimized for performance, and implemented on Xilinx FPGA devices. This flow delivers on the cost, performance, and power benefits that have so far eluded designers who rely on traditional microprocessors for implementing floating-point algorithms.

Starting from the standard industrial application of a Proportional-Integral-Derivative (PID) control, as reported in "Vivado HLS Eases Design of Floating-Point PID Controller" [Ref 1], we will review and explain the following aspects of implementing floating-point algorithms in an FPGA:

1. Explain benefits and opportunities created by implementing floating-point designs in an FPGA device
2. Review an industry standard application, showing how the algorithm can be implemented in C/C++ code.
3. Show how the C/C++ design can transformed into a high-performance hardware implemented using Vivado high-level synthesis (HLS) and how the design can be further optimized.
4. Discuss how the performance of the RTL design can be analyzed and verified using System Generator for DSP.

The above topics 3 and 4 represent the newest contribution of this application note to the previous work already published in "VIVADO HLS Eases Design of Floating-Point PID Controller" [Ref 1], although also topic 2 has also been enhanced.

The Proportional-Integral-Derivative (PID) systems are used throughout industries, from factory control systems to car braking systems, robot controls, and medical devices. The implementation of a PID core very often requires the accuracy provided by floating-point calculations.

Figure 1 shows the block diagram for a standard PID controller adopted in a discrete closed-loop control system: the PID accepts an input reference value `w(n)` and the plant system output value `y(n)` and processes their difference, the error `e(n)`, to ensure that the output matches the desired input value. Every part of the PID contributes to a specific action:

- **Proportional stage**: Drives the controller output according to the size of the error.
- **Integral stage**: Eliminates the state-steady offset.
- **Derivative stage**: Evaluates the trend to correct the output and improve overall stability by limiting overshoot.

The result of each stage is combined to create a fast responding but stable system.

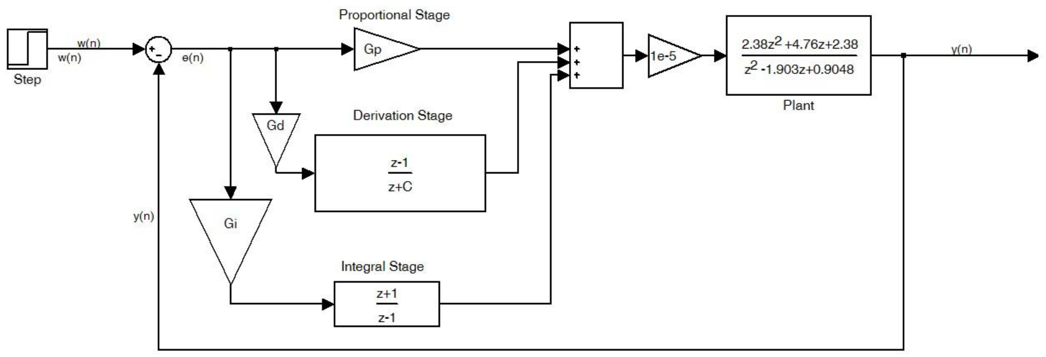


Figure 1: **PID Block Within a Closed-Loop Control System**

The entire application depends on the accuracy of calculations performed. For this reason, PID controllers have traditionally been implemented in microprocessors; they allow the algorithm designer to easily capture the algorithm and have historically had acceptable performance.

When the required performance of the system has exceeded the ability of standard microprocessors, dedicated Digital Signals Processors (DSPs) have been employed. These often require some modification to the original C algorithm but can perform floating-point calculations faster, allowing the performance requirements to be met.

However, in a lot of industrial applications, the PID is being used in more and more places and single PID loops are being replaced by cascaded loops. For instance:

- The torque is managed by the current loop PID.
- The motor speed is managed by the velocity PID cascaded with the current PID.
- The position is managed by the space PID cascaded with the velocity PID.

In these cases, the time taken for the sequential execution of the each new PID loop, which is inherent when using a microprocessor, contributes a substantial delay in the overall computation. Continuing to solve this problem using traditional micro or DSP processors simply adds more and more discrete parts to the system cost and power.

Additionally, in recent years, electrical drives and robots are placed into Networked Control Systems using the PID(s) in a loop with a communication channel, which in turn must interact with a software communication protocol stack. System architects and control engineers are often sacrificing performance to get an all-software implementation.

FPGA technology provides a much more powerful and flexible response to these challenges. For example:

- Implementing the PID in an FPGA device allows multiple PID instances to operate concurrently, due to the massive parallel resource available on an FPGA.
- FPGAs offer flexibility in adding additional PID loops if the application needs more control loops, without affecting the performance of any existing PIDs in the system or typically adding any discrete parts to the system.
- With the introduction of the new Xilinx Zynq™-7000 All Programmable SoC, there are additional advantages because the power of the FPGA can be directly exploited by the powerful ARM Cortex A9 CPU processor. Any software stack can be implemented on the on-board CPU. Here the determinism and speed of FPGA implementation is a great advantage.
- In addition to exceeding the performance of serial execution micro-processors, FPGAs offer cost and power advantages.
- FPGAs are programmable and can be updated as easily as any CPU code. The solution they provide is not all-software, but it is all programmable.

However, FPGA devices require that the design is implemented using a Register Transfer Level (RTL) design language, such as VHDL or Verilog. This has been an insurmountable hurdle for many control engineers, with a no background in hardware design, and they have been discouraged from using FPGA technology.

This application note shows how the Xilinx Vivado Design Suite, with the new Vivado High-Level Synthesis (HLS) design tool and System Generator for DSP, removes the burden of requiring the algorithm designer to also be a hardware design expert. By simply understanding the basic resources on an FPGA and standard hardware IO protocols, the designer can now implement, optimize, analyze, and verify designs on an FPGA.

The transformation from the C/C++ specification to the RTL required by an FPGA requires no more adaptations to regular C or C++ code than the implementation in a DSP, and does not represent any significant detailed knowledge of hardware design.

From PID Theory to C++ Code

Before going into detail on the specifics of the design flow, it is worth examining how the algorithm is developed. This section covers the theoretical background of the PID algorithm.

A system without feedback control is defined as an open loop, and its transfer function (the way the system's input maps into its output) must exhibit a phase shift of less than 180° at unity gain. The difference between the phase shift of the open loop and 180° at unity gain is called the "phase margin."

System gain and phase shift are commonly expressed in the analog s domain of the Laplace transform or in the discrete z domain of the Z-transform. Assuming `P(z)` and `H(z)` to be, respectively, the discrete transfer functions of the plant and of the PID controller in the z domain, the effective transfer function `T(z)` of the whole closed-loop system can be expressed as:

$$T(z) = \frac{P(z) \cdot H(z)}{1 + P(z) \cdot H(z)}$$

The values in `z` of the numerator and denominator of `T(z)` are respectively called zeros and poles. The phase shift of the PID enters into the loop and sums to the total phase; thus, a fast PID is desirable to keep the phase lag at a minimum. Ideally, the PID's response time should be immediate, as with an analog controller. Therefore, the PID computational speed is paramount.

In a closed-loop system, stability is mandatory, especially for highly sophisticated applications like robotic systems or electrical motor drives. Without stability, the control loop might respond with undesired oscillations, or by generating a sluggish response. Stability can be achieved via compensation of the poles and zeros of the PID controller in order to get the best achievable performance from the closed loop system: the gain and phase characteristics.

Without entering into the details of the mathematical derivation of the DC motor and rotor transfer functions, it is possible to express the plant as in Equation 1, showing the analog open-loop transfer function in the Laplace domain:

$$P(s) = \frac{Y(s)}{U(s)} = \frac{a}{s^2 b + c + d}$$

Open Loop Transfer Function *Equation 1*

where `a`, `b`, `c`, and `d` are the plant numerical parameters.

As a numerical example throughout this application note, the following values are assumed: `a=1`, `b=1`, `c=10`, and `d=20`.

Equation 2 illustrates the PID controller transfer function:

$$H(s) = \frac{U(s)}{E(s)} = K_P + \frac{1}{s} \cdot K_I + s \cdot K_D$$

PID Controller Transfer Function *Equation 2*

where `U(s)` and `E(s)` are the Laplace transforms of the PID output and input signals, respectively `u(n)` and `e(n)`. The terms `KP`, `KI`, and `KD` are the gains of the Proportional, Integral, and Derivative stages, respectively.

One of the methods for transforming transfer functions from the Laplace domain to the `z` domain is the Tustin approximation with trapezoidal integration: the plant (Equation 1) and PID (Equation 2) transfer functions are shown respectively in Equation 3 and Equation 4 in their digital form:

$$P(z) = 10^{-5} \cdot \frac{2.38 z^2 + 4.76 z + 2.38}{z^2 - 1.903 z + 0.9048}$$

Open Loop Transfer Function *Equation 3*

$$H(z) = K_P + K_I \frac{T_S}{2} \frac{z+1}{z-1} + \frac{K_D}{T_F + \frac{T_S}{2} \frac{z+1}{z-1}}$$

PID Controller Transfer Function *Equation 4*

where `TF` and `TS` are, respectively, the derivative filter time and the sampling time.

Equation 3 and Equation 4 can be formally rewritten as shown, respectively, in Equation 5 and Equation 6:

$$P(z) = 10^{-5} \cdot \frac{2.38 + 4.76 z^{-1} + 2.38 z^{-2}}{1 - 1.903 z^{-1} + 0.9048 z^{-2}}$$

Open Loop Transfer Function *Equation 5*

$$H(z) = \frac{U(z)}{E(z)} = G_P + G_I \frac{z+1}{z-1} + G_D \frac{z-1}{z+C} = G_P + G_I \frac{1+z^{-1}}{1-z^{-1}} + G_D \frac{1-z^{-1}}{1+Cz^{-1}}$$

PID Controller Transfer Function *Equation 6*

with

$$G_P = K_P$$
$$G_I = K_I \frac{T_S}{2}$$
$$G_D = \frac{2K_D}{\mathrm{T_S} + 2\mathrm{T_F}}$$
$$\mathrm{C} = \frac{\mathrm{T_S} - 2\mathrm{T_F}}{\mathrm{T_S} + 2\mathrm{T_F}}$$

The MathWorks' Control System toolbox of MATLAB® and Simulink® is a powerful tool to design and simulate analog and digital PID control systems. The following MATLAB script illustrates some theory of PID controllers.

Note: When possible, the same symbols shown in Equation 1, Equation 2, Equation 3, and Equation 4 are adopted in the code.

```
clear all;
close all;
clc

Ts = 1/100; % sampling period
t  = 0 : Ts : 2.56-Ts; % time basis

% (Laplace transform) transfer function of the continuous system to be
controlled
a = 1; b = 1; c = 10; d = 20;
num = a; den = [b c d];
plant_s = tf(num, den); % Plant transfer function (Laplace transform)

% clipping values
min_val = -64;
max_val =  63;

%% DISCRETE PLANT system (P(z))

% (Z transform) transfer function of the discrete system to be controlled
P = c2d(plant_s,Ts,'tustin')  % discrete plant transfer function (Zeta
transform)
figure; step(P,t); title 'Open-Loop answer to input step for a discrete
plant'; grid
```

The next commands specify a controller that will reduce the rise time, reduce the settling time and, eliminate the steady-state error.

```
%% DISCRETE PID system (H(z))
% dummy parameters to generate a PID
Kp=1; Ki=1; Kd=1; Tf=1;
HH = pid( Kp, Ki, Kd, Tf, Ts, 'IFormula','Trapezoidal',
'DFormula','Trapezoidal');
H = pidtune(P, HH);

%% Proportional Discrete Controller
H.Kp = 300;
H.Ki = 0;
H.Kd = 0;

% closed loop system:  T(z) = P(z)H(z) / ( 1 + P(z)H(z) )
T = feedback(H * P,1);
figure; step(T,t); grid
title(['Closed-Loop answer to input step with Kp=',num2str(H.Kp)]);
axis([0    2.0  0 1.5]);
```

The output signal of the plant stand-alone (that is in open-loop) to a step input signal shows that the DC gain of the plant transfer function is 1/20, so 0.05 is the final value of the output to a unit step input. This corresponds to the large steady-state error of 0.95. Furthermore, the rise time is about one second and the settling time is about 1.5 seconds, as illustrated in Figure 2.

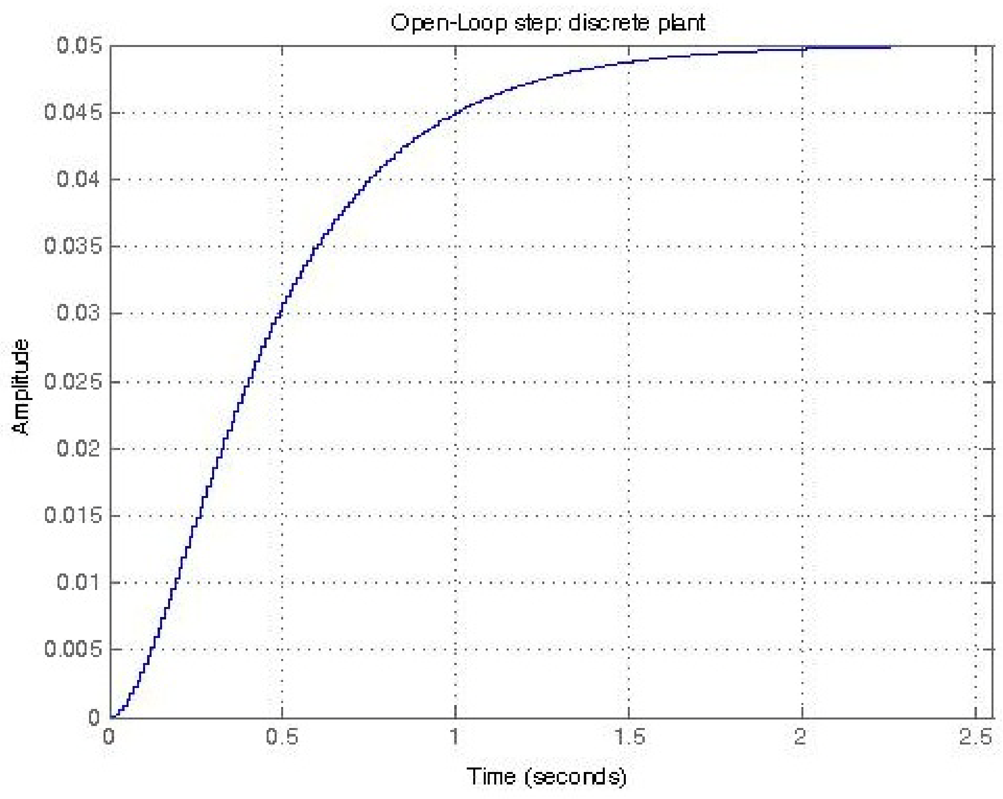


Figure 2: **Discrete Open-Loop Plant Answer to a Step Input Signal**

Figure 3 shows that the proportional (P) controller has reduced both the rise time and the steady-state error, increased the overshoot, and decreased the settling time by a small amount.

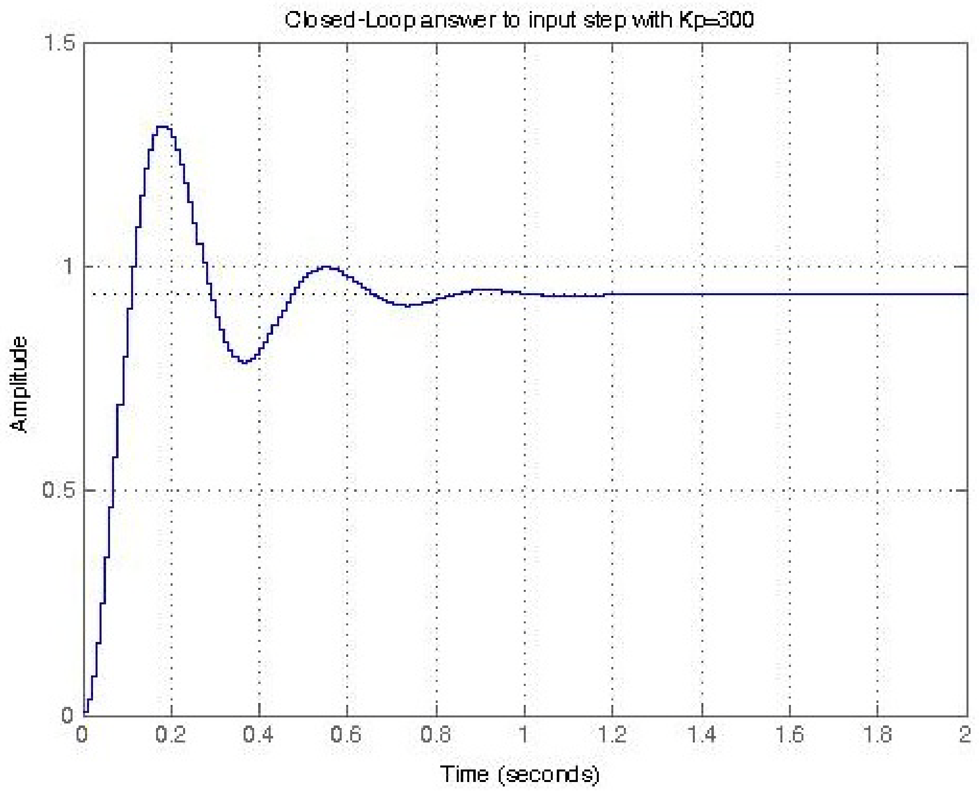


Figure 3: **Closed-Loop Plant Answer to a Step Input Signal with a P Controller Having KP=300**

On the other hand, the plot in Figure 4, page 8 confirms that the PID controller reduces both the overshoot and the settling time and has a small effect on the rise time and the steady-state error.

Finally, Figure 5, page 9 illustrates the answer to the step input signal, once the PID controller has been tuned to the plant; the closed loop system exhibits now negligible overshoot and good rise time. The optimal settings of the PID parameters are: KP=35.36, KI=102.2, and KD=0.29.

```
%% Proportional Derivative Discrete Controller
%
% The derivative controller (Kd) reduces both the overshoot and the
settling time.

H.Kp = 300;
H.Ki = 0;
H.Kd = 30;

T = feedback(H*P, 1);
figure; step(T, t);  grid;
title(['Closed-Loop step: Kp=',num2str(H.Kp), ' Kd=' num2str(H.Kd)])
axis([0    2.0   0 1.5]);
```

```
%% Proportional Integral Derivative Discrete Controller

% tuning the PID
H  = pidtune(P, HH)
Kp = H.Kp;
Ki = H.Ki;
Kd = H.Kd;
Tf = H.Tf;

% closed loop system:  T(z) = P(z)H(z) / ( 1 + P(z)H(z) )
T  = feedback(H*P, 1);
figure; step(T, t); grid
title(['Closed-Loop answer to input step: Kp=',num2str(H.Kp), ' Ki='
num2str(H.Ki), ' Kd=' num2str(H.Kd)])
axis([0    2.0  0 1.5]);
```

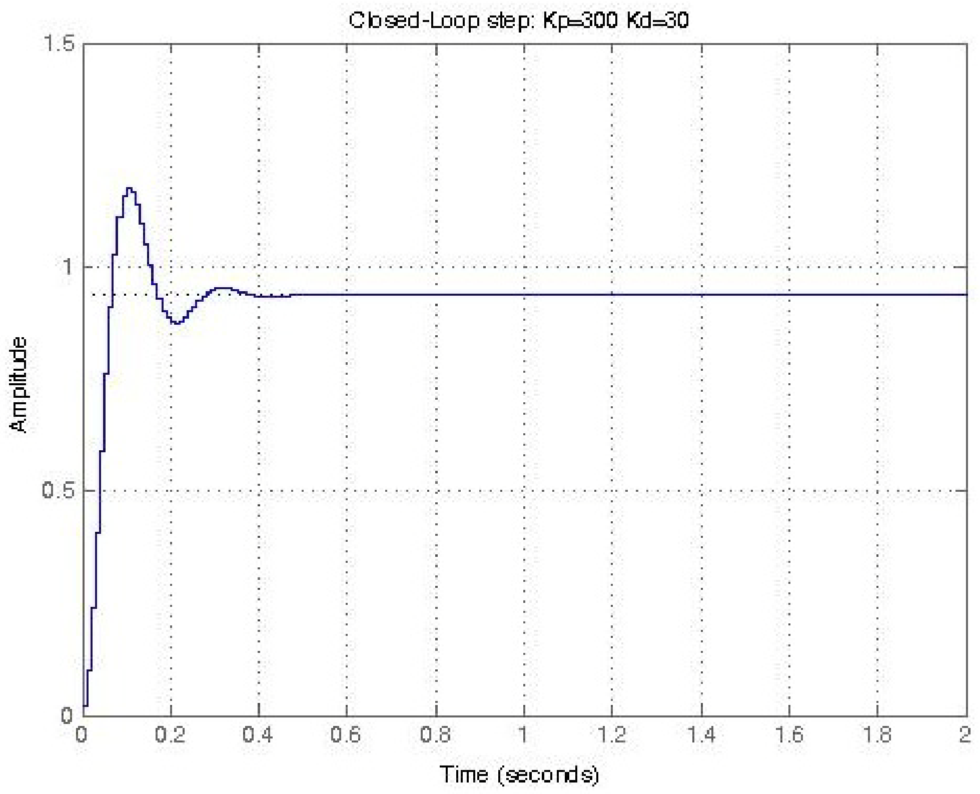


Figure 4: **Closed-Loop Plant Answer to a Step Input Signal with a Pd Having Kp=300 and Kd=30**

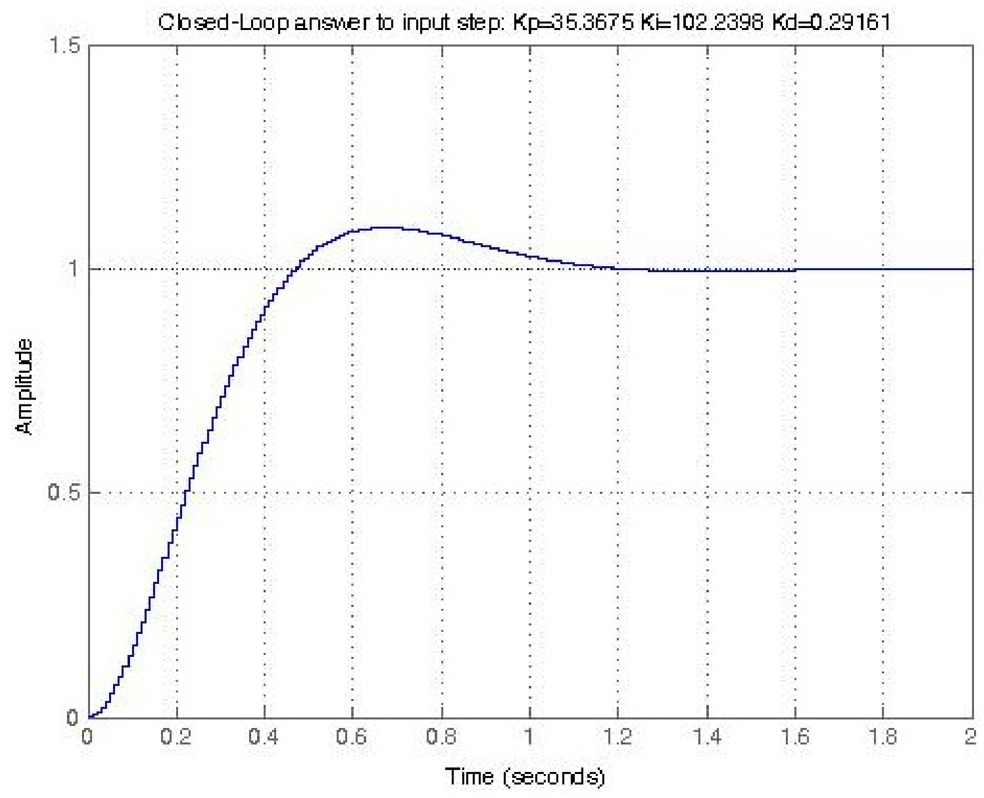


Figure 5: **Closed-Loop Plant Answer to a Step Input Signal with a PID Having Kp=35.36, Ki=102.2, and Kd=0.29**

The transfer function of the derivation stage D(z) from Equation 6 is shown here in Equation 7.

$$D(z) = \frac{Y_D(z)}{X_1(z)} = \left(\frac{1 - z^{-1}}{1 + cz^{-1}}\right)$$

Derivation Stage Transfer Function *Equation 7*

By anti-transforming it, the discrete time equation can be obtained, with `x1(n) = GD e(n)` the effective signal input to the derivation stage:

$$y_D(n) = -c \cdot y_D(n-1) + x_1(n) - x_1(n-1)$$

Derivation Stage Discrete Time Equation *Equation 8*

Similarly, in Equation 6, the transfer function of the Integration stage `I(z)` is shown here as Equation 9:

$$I(z) = \frac{Y_I(z)}{X_2(z)} = \left(\frac{1+z^{-1}}{1-z^{-1}}\right)$$

Integration Stage Transfer Function *Equation 9*

which in the discrete time, with `x2(n) = GI e(n)` the effective signal input to the integration stage, becomes Equation 10:

$$y_I(n) = y_I(n-1) + x_2(n) + x_2(n-1)$$

Integration Stage Discrete Time Equation *Equation 10*

In conclusion, the effective time-discrete Equations modeling the PID controller and the plant are as shown in Equation 11:

$$\begin{gathered}
x_1(n) = G_D \cdot e(n) \\
x_2(n) = G_I \cdot e(n) \\
u(n) = G_P e(n) + y_D(n) + y_I(n) \\
y_D(n) = x_1(n) - x_1(n-1) - C\ y_D(n-1) \\
y_I(n) = x_2(n) + x_2(n-1) + y_I(n-1) \\
y(n) = 1.903 \cdot y(n-1) - 0.9048 \cdot y(n-2) + 10^{-5} \cdot \left(2.38 \cdot u(n) + 4.76 \cdot u(n-1) + 2.38 \cdot u(n-2)\right) \\
e(n) = w(n) - y(n)
\end{gathered}$$

PID Discrete Time Equations *Equation 11*

The C code implementation of the PID design and its functional test-bench are reported in Figure 6, page 15. The design assumes that

- The PID input and output signals `e(n)` and `U(n)` are saturated by an amount that the user can control
- The PID coefficients (`GI`, `GP`, `GD`, and `C` of Equation 6) and the maximum and minimum value for `e(n)` and `u(n)` signals are to be loaded into the PID core at function call time, so that the PID controller can even be tuned at run time in more sophisticated applications.
- To avoid the possible overflow of its accumulator, the Integrator's Equation 10 is changed into the following:

$$\begin{aligned}
T_I(n) &= x_2(n) + x_2(n-1) \\
y_I(n) &= CLIP(T_I(n) + y_I(n-1))
\end{aligned}$$

Discrete Integrator with Clipping *Equation 12*

The data types in the source code shown in Figure 6 are defined in the header file reported in Figure 7, page 15 as float types (that is 32-bit floating point) when no C preprocessing symbol is defined, which is the default configuration. Note that other data types are allowed: double types (64-bit floating point) and fixed types (25-bit fixed fractional), depending respectively on the preprocessing symbol defined, PID_DOUBLEPREC and PID_FIXEDPOINT.

```
#include "pid.h"

void PID_Controller(bool InitN, data_type coeff[6],
                    data_type din[2], data_type dout[2])
{

// previous PID states: Yd(n-1), X1(n-1), X2(n-1), Yi(n-1)
static data_type   prev_x1, prev_x2, prev_yd, prev_yi;
// current local states
data_type w, e, y, x1, x2, yd, yi;
// local variables
data_type max_lim, min_lim, Gi, Gd, C, Gp, tmp;
data_type pid_mult, pid_addsub, pid_mult2, pid_addsub2;

// get PID input coefficients
Gi = coeff[0];  Gd = coeff[1];
C = coeff[2]; Gp = coeff[3];
max_lim = coeff[4]; min_lim = coeff[5];

// get PID input signals
w = din[0]; // effective input signal
y = din[1]; // closed loop signal

if (InitN==0)
{
  prev_yi = 0; // reset Integrator stage
  prev_x1 = 0; // reset Derivative stage
  prev_yd = 0; prev_x2 = 0;
}

// compute error signal E = W - Y
pid_addsub = w - y;
e = (pid_addsub > max_lim) ? max_lim : pid_addsub;
e = (pid_addsub < min_lim) ? min_lim : e;
pid_mult  = Gd * e;
pid_mult2 = Gi * e;
x1 = pid_mult;  // input signal of the derivative stage
x2 = pid_mult2; // input signal of the integration stage

// Derivation stage
// Yd(n) = -C*Yd(n-1)+X1(n)-X1(n-1) = X1 -(prev_X1 + C*prev_Yd)
pid_mult = C * prev_yd;
pid_addsub2= x1 - prev_x1;
pid_addsub = pre_addsub2 - pid_mult;
yd         = pid_addsub;

// Integrator  stage
// Ti = X2(n) + X2(n-1)
// Yi(n) = CLIP( Yi(n-1) + Ti )
pid_addsub = prev_x2 + x2;
pid_addsub2= prev_yi + pid_addsub;
yi = (pid_addsub2 > max_lim) ? max_lim : pid_addsub2;
yi = (pid_addsub2 < min_lim) ? min_lim : yi;
```

```
// output signal U(n)
pid_mult   = Gp * e;
pid_addsub = yi + yd;
pid_addsub2= pid_addsub + pid_mult;
tmp = (pid_addsub2 > max_lim) ? max_lim : pid_addsub2;
tmp = (pid_addsub2 < min_lim) ? min_lim : tmp;
dout[0] = tmp; // PID output
dout[1] = e; // error reported as output

// update internal PID states for the next iteration
prev_x1 = x1; prev_x2 = x2;
prev_yd = yd; prev_yi = yi;

return;
}

#define N       256
#define MAX_VAL  64
const float C  = -0.7931; // derivation constant
const float Gd =  6.0324; // Derivation gain
const float Gp = 35.3675; // Proportional gain
const float Gi =  0.5112; // Integration gain
const float max_clip =  MAX_VAL-1;
const float min_clip = -MAX_VAL;

int main(void)
{

int i, ret_val;
FILE *fp_e, *fp_i, *fp_d, *fp_p, *fp_u, *fp_u2, *fp_e2;
float diff_E, diff_U, tot_diff_E, tot_diff_U;

// PID input signals
bool ResetN =1;
data_type din[2], dout[2], coeff[6];

// CLOSED LOOP SYSTEM STATES
float yd_prev = 0; float x1_prev  = 0; // derivation  states
float yi_prev = 0; float x2_prev  = 0; // integration states
float y_z1 = 0; float y_z2 = 0;
float u_z1 = 0; float u_z2 = 0;

// closed loop system states
float ref_out_prev = 0; float out_prev = 0;

// PID AND PLANT SIGNALS
float w[N], u[N], y[N], e[N];
float ref_e[N], ref_u[N], ref_y[N], ref_yi[N], ref_yd[N];
float ref_y_z1 = 0; float ref_y_z2 = 0;
float ref_u_z1 = 0; float ref_u_z2 = 0;

tot_diff_E = 0; tot_diff_U = 0;

// STEP FUNCTION
w[0] = 0;  for (i=1; i<N; i++) w[i] = 1;

// file I/O
if ( ( fp_e = fopen ( (char *) "./test_data/e_res.txt", "w" ) ) == NULL )
{
fprintf (stderr, "Cannot open input file %s\n", (char *) "e_res.txt");
exit (-1 );
}
```

```
if ( ( fp_i = fopen ( (char *) "./test_data/i_res.txt", "w" ) ) == NULL )
{
fprintf (stderr, "Cannot open input file %s\n", (char *) "i_res.txt");
exit (-1 );
}
if ( ( fp_d = fopen ( (char *) "./test_data/d_res.txt", "w" ) ) == NULL )
{
fprintf (stderr, "Cannot open input file %s\n", (char *) "d_res.txt");
exit (-1 );
}
if ( ( fp_p = fopen ( (char *) "./test_data/yp_res.txt", "w" ) ) == NULL )
{
fprintf (stderr, "Cannot open output file %s\n", (char *) "yp_res.txt");
exit (-1 );
}
if ( ( fp_u = fopen ( (char *) "./test_data/u_res.txt", "w" ) ) == NULL )
{
fprintf (stderr, "Cannot open output file %s\n", (char *) "u_res.txt");
exit (-1 );
}
if ( ( fp_u2 = fopen ( (char *) "./test_data/pid_u_res.txt", "w" ) ) == NULL
)
{
fprintf (stderr, "Cannot open output file %s\n", (char *) "pid_u_res.txt");
exit (-1 );
}
if ( ( fp_e2 = fopen ( (char *) "./test_data/pid_e_res.txt", "w" ) ) == NULL
)
{
fprintf (stderr, "Cannot open input file %s\n", (char *) "pid_e_res.txt");
exit (-1 );
}

coeff[0] = (data_type) Gi;       coeff[1] = (data_type) Gd;
coeff[2] = (data_type) C;        coeff[3] = (data_type) Gp;
coeff[4] = (data_type) max_clip; coeff[5] = (data_type) min_clip;

// TEST BENCH
for (i = 0; i<N; i++)
{
  // error between input and output signals
  ref_e[i] = w[i] - ref_out_prev; // PID IN CLOSED LOOP
  ref_e[i] = (ref_e[i] > max_clip) ? max_clip : ref_e[i];
  ref_e[i] = (ref_e[i] < min_clip) ? min_clip : ref_e[i];

  float x1 = Gd*ref_e[i];
  float x2 = Gi*ref_e[i];

  // derivation stage
  // Yd(n-1) = -C * Yd(n-1) + X1(n) - X1(n-1)
  ref_yd[i] = -C*yd_prev + x1 - x1_prev;
  yd_prev = ref_yd[i];
  x1_prev = x1;
// integration
Ti(n) = X2(n) + X2(n-1);
Yi(n) = CLIP( Yi(n-1) + Ti(n) )
float ti = x2_prev + x2;
ti = yi_prev + ti;
ti = (ti > max_clip) ? max_clip : ti;
ti = (ti < min_clip) ? min_clip : ti;
ref_yi[i] = ti;
yi_prev   = ti;
x2_prev   = x2;

// PID control output signal
ref_u[i] = ref_e[i] * Gp + ref_yd[i] + ref_yi[i];
ref_u[i] = (ref_u[i] > max_clip) ? max_clip : ref_u[i];
ref_u[i] = (ref_u[i] < min_clip) ? min_clip : ref_u[i];

// plant
ref_y[i] = 1.903*ref_y_z1 -0.9048*ref_y_z2 + 2.38e-005 *ref_u[i]
             + 4.76e-005*ref_u_z1 + 2.38e-005*ref_u_z2;
ref_y_z2 = ref_y_z1;
ref_y_z1 = ref_y[i];
ref_u_z2 = ref_u_z1;
ref_u_z1 = ref_u[i];
ref_out_prev = ref_y[i];

fprintf(fp_u, "%f\n",  ref_u[i]);
fprintf(fp_e, "%f\n",  ref_e[i]);

// EFFECTIVE Design Under Test
din[0] = (data_type) w[i];
din[1] = (data_type) out_prev;
PID_Controller(ResetN, coeff, din, dout);
u[i] = (float) dout[0];
e[i] = (float) dout[1];

// plant
y[i] = 1.903*y_z1 -0.9048*y_z2 + 2.38e-005*u[i]
            + 4.76e-005 *u_z1 + 2.38e-005*u_z2;
y_z2 = y_z1; y_z1 = y[i];
u_z2 = u_z1; u_z1 = u[i];
out_prev = y[i];

fprintf(fp_u2, "%f\n",  (float) u[i]);
fprintf(fp_e2, "%f\n",  (float) e[i]);

// CHECK RESULTS

// ERROR SIGNAL
diff_E = (float) fabs( (float) e[i] - (float) ref_e[i]);
tot_diff_E += diff_E;
// PID OUTPUT
diff_U = (float) fabs( (float) u[i] - (float) ref_u[i]);
tot_diff_U += diff_U;

}
```

```
  fclose(fp_e); fclose(fp_d); fclose(fp_i); fclose(fp_u);
  fclose(fp_p); fclose(fp_u2); fclose(fp_e2);

  fprintf(stdout, "\n TEST RESULT U=%f \n", tot_diff_U);
  fprintf(stdout, "\n TEST RESULT E=%f \n", tot_diff_E);
  tot_diff_E += tot_diff_U; // total error during simulation
  fprintf(stdout, "\n TEST RESULT TOTAL ERROR =%f \n", tot_diff_E);


  if (tot_diff_E < PID_THRESHOLD) // MY THRESHOLD
  {
    ret_val = 0;
    fprintf(stdout, "\n TEST PASSED! \n");
  }
  else
  {
    ret_val = 1;
    fprintf(stdout, "\n TEST FAILED! \n");
  }

  return ret_val;

}
```

Figure 6: **PID Controller C Code**

```
#if defined(PID_FIXEDPOINT)

// Define AP_FIXED types
#include "ap_fixed.h"
// 25-bit bit signed date, with
typedef ap_fixed<25,10> data_type;
#define PID_THRESHOLD 0.2
#elif defined(PID_DOUBLEPREC)
typedef double data_type; // 64-bit floating point
#define PID_THRESHOLD 0.01

#else

typedef float data_type; // 32-bit floating point
#define PID_THRESHOLD 0.01

#endif
```

Figure 7: **PID Controller C Header File**

C Specification to RTL Design

Once the algorithm has been captured in C++ code, the Xilinx Vivado HLS can be used to synthesize this in to an RTL implementation.

In addition to the C source code, Vivado HLS accepts as inputs a target clock frequency, a target device specification, and user directives (commands) which can be used to control and direct specific optimizations. The easiest way to understand the function and capabilities of Vivado HLS is to step through an example. For more information on Vivado HLS, refer to UG902, *Vivado Design Suite User Guide: High-Level Synthesis* [Ref 2].

Given the input source code shown in Figure 6, and specifying only the clock period and target device, as shown in Figure 8.

```
set_part  {xc7z010clg400-1}
create_clock -period 10
```

Figure 8: **Vivado HLS Device and Clock Targets**

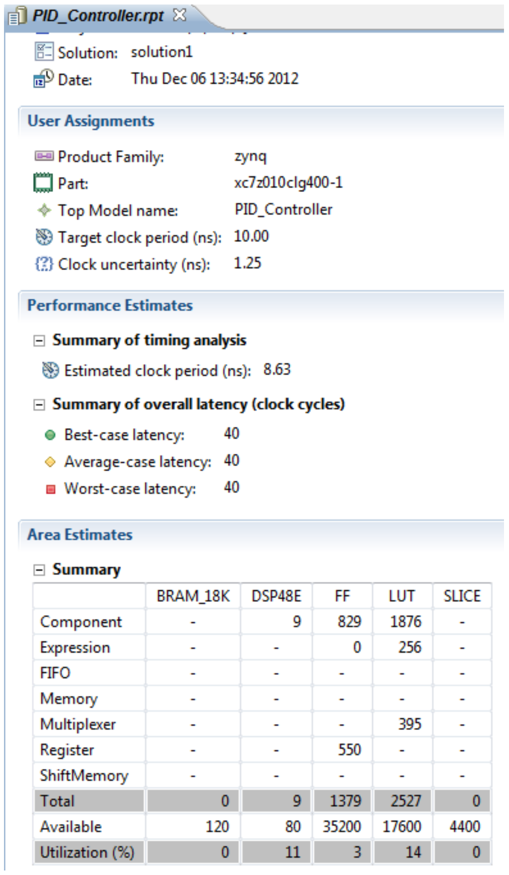
Vivado HLS:

- Transforms each of the operations in the C code into an equivalent hardware operation and schedules those operations into clock cycles. Using knowledge of the clock period and device delays, it will put as many operations as possible into a single clock cycle.
- Uses interface synthesis to automatically synchronize how the data can be brought into the hardware block and written out. For example, if data is supplied as an array it will automatically construct an interface to access a RAM block (Other IO interface options can be specified).
- Maps each of the hardware operations onto an equivalent hardware unit in the FPGA device.
- Performs any user-specified optimizations, such as pipeline or concurrent operations.
- Outputs the final design, with reports, in Verilog and VHDL for implementation in the FPGA.

The reports generated by synthesizing the code in Figure 9, page 17 can help explain its operation and capabilities and shows the initial performance characteristics (initial, because no user optimization directives have been specified: these are the default synthesis results).

Vivado HLS has analyzed all the operations in the C code and determined that it will take 40 clocks cycle to calculate this result using the specified target technology and clock period. In fact, this design could execute with a maximum clock period of 8.63 ns.

The area estimates in Figure 9 show how many resources on the FPGA the design is expected to use: 9 DSP48 resources, about 1380 Flip-Flops and 2530 LUTs (Look-Up-Tables).



PID_Controller.rpt

Solution: solution1
Date: Thu Dec 06 13:34:56 2012

User Assignments

Product Family:	zynq
Part:	xc7z010clg400-1
Top Model name:	PID_Controller
Target clock period (ns):	10.00
Clock uncertainty (ns):	1.25

Performance Estimates

Summary of timing analysis

Estimated clock period (ns): 8.63

Summary of overall latency (clock cycles)

● Best-case latency:	40
◆ Average-case latency:	40
■ Worst-case latency:	40

Area Estimates

Summary

	BRAM_18K	DSP48E	FF	LUT	SLICE
Component	-	9	829	1876	-
Expression	-	-	0	256	-
FIFO	-	-	-	-	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	395	-
Register	-	-	550	-	-
ShiftMemory	-	-	-	-	-
Total	0	9	1379	2527	0
Available	120	80	35200	17600	4400
Utilization (%)	0	11	3	14	0

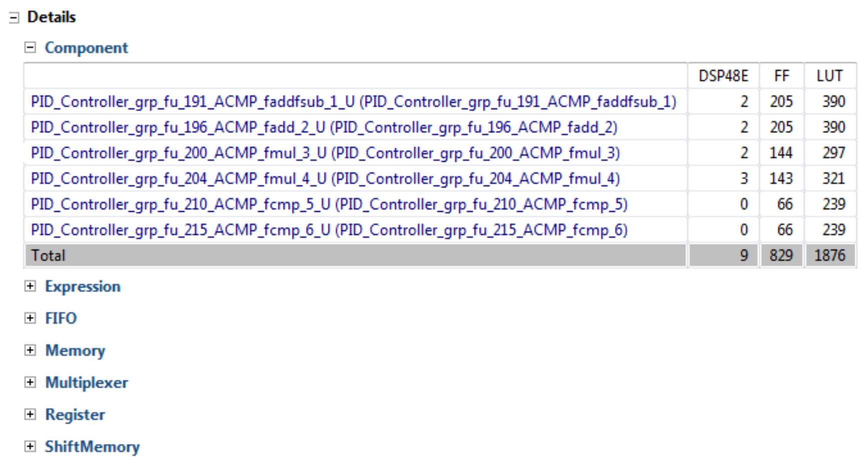
Figure 9: **Initial Performance Characteristics in 32-Bit Floating Point**

The words "estimates" and "expected" are applied here because the RTL synthesis process has still to transform this RTL code into gate level components and place and route them in the FPGA: there may be other gate level optimizations which impact the final results.

Further down the report, you can see how the design was implemented; as illustrated in Figure 10, the design uses the following floating-point components:

- Adder-subtractor
- Adder
- Multiplier
- Comparator

The report also shows how many of the basic resources (DSP48, Flip-Flop, and LUT) each of these components uses.


Details
Component

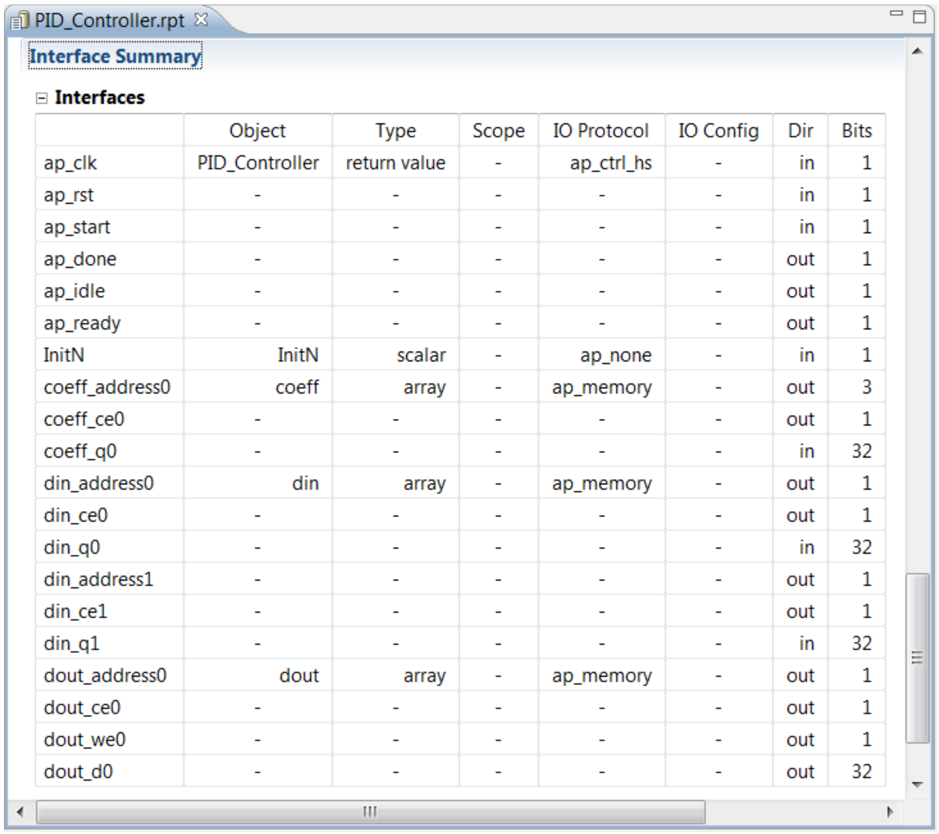
	DSP48E	FF	LUT
PID_Controller_grp_fu_191_ACMP_faddfsub_1_U (PID_Controller_grp_fu_191_ACMP_faddfsub_1)	2	205	390
PID_Controller_grp_fu_196_ACMP_fadd_2_U (PID_Controller_grp_fu_196_ACMP_fadd_2)	2	205	390
PID_Controller_grp_fu_200_ACMP_fmul_3_U (PID_Controller_grp_fu_200_ACMP_fmul_3)	2	144	297
PID_Controller_grp_fu_204_ACMP_fmul_4_U (PID_Controller_grp_fu_204_ACMP_fmul_4)	3	143	321
PID_Controller_grp_fu_210_ACMP_fcmp_5_U (PID_Controller_grp_fu_210_ACMP_fcmp_5)	0	66	239
PID_Controller_grp_fu_215_ACMP_fcmp_6_U (PID_Controller_grp_fu_215_ACMP_fcmp_6)	0	66	239
Total	9	829	1876

Figure 10: **Initial Design Details in 32-Bit Floating Point**

Finally, Figure 11, page 19 reveals how the C function arguments where transformed by interface synthesis into IO ports which can connect to other blocks in the design.

In particular, it can be noted that:

- Clock and reset signals were added to the design (`ap_clk`, `ap_rst`).
- By default, a design-level protocol is added to the design (this is optional). This allows the design to be started (`ap_start`) and indicates when it is ready for new inputs, has complete (`ap_done`), or is idle.
- The input `InitN` has been implemented as an input port with no IO protocol (`ap_none`). When the read on this port occurs, the value is read. Because there are no handshake signals to indicate when this port is being read, it should be held stable after `ap_start` is asserted and until `ap_ready` is asserted.
- Each of the array arguments have been transformed into RAM interface with the appropriate address, enable, and write signals to access a Xilinx block RAM. Additionally, Vivado HLS has automatically determined that the performance can be improved if port `din` uses a dual-port block RAM; this can be configured to a single-port BRAM if desired.



PID_Controller.rpt

Interface Summary

Interfaces

	Object	Type	Scope	IO Protocol	IO Config	Dir	Bits
ap_clk	PID_Controller	return value	-	ap_ctrl_hs	-	in	1
ap_rst	-	-	-	-	-	in	1
ap_start	-	-	-	-	-	in	1
ap_done	-	-	-	-	-	out	1
ap_idle	-	-	-	-	-	out	1
ap_ready	-	-	-	-	-	out	1
InitN	InitN	scalar	-	ap_none	-	in	1
coeff_address0	coeff	array	-	ap_memory	-	out	3
coeff_ce0	-	-	-	-	-	out	1
coeff_q0	-	-	-	-	-	in	32
din_address0	din	array	-	ap_memory	-	out	1
din_ce0	-	-	-	-	-	out	1
din_q0	-	-	-	-	-	in	32
din_address1	-	-	-	-	-	out	1
din_ce1	-	-	-	-	-	out	1
din_q1	-	-	-	-	-	in	32
dout_address0	dout	array	-	ap_memory	-	out	1
dout_ce0	-	-	-	-	-	out	1
dout_we0	-	-	-	-	-	out	1
dout_d0	-	-	-	-	-	out	32

Figure 11: **Initial RTL Ports**

Vivado HLS has created an RTL implementation where the operations in the C code and the IO operations have been automatically implemented without any requirement to know an RTL design language, such as Verilog or VHDL, or without knowing anything about RTL design in general.

The next section shows how this implementation can be made more optimal, and how in this case, some basic understanding of RTL IO interfaces and Xilinx device resources is an advantage.

Optimized RTL

The initial design created by Vivado HLS can be made more optimal. The optimizations shown here are only a few of the possible optimizations that can be made to the design. For more details on the optimizations provided by Vivado HLS, refer to UG902, *Vivado Design Suite User Guide: High-Level Synthesis* [Ref 2].

The first thing to be done is to decide that instead of using a RAM interface for the data signals `din`, `coeff`, and `dout`, a FIFO interface can be adopted. This is possible because the input data and output data are always read and written sequentially. A FIFO interface requires no address, so there is no need of logic required to calculate the address.

The directives to achieve these IO optimizations are shown in Figure 12.

```
set_directive_interface -mode ap_fifo "PID_Controller" din
set_directive_interface -mode ap_fifo "PID_Controller" dout
set_directive_interface -mode ap_fifo "PID_Controller" coeff
```

Figure 12: **FIFO IO Protocols**

Finally, in the initial performance results, we noticed that the design was using two floating-point multipliers and both a floating-point `addsub` unit and `adder` unit. Here, some knowledge of the Xilinx FPGA device is useful. Floating-point operations consume more resources than standard integer or bit operations. For example, these floating-point adders use two DSP48s and approximately 400 flip-flops and LUTs each: a standard integer adder only requires a few LUTs.

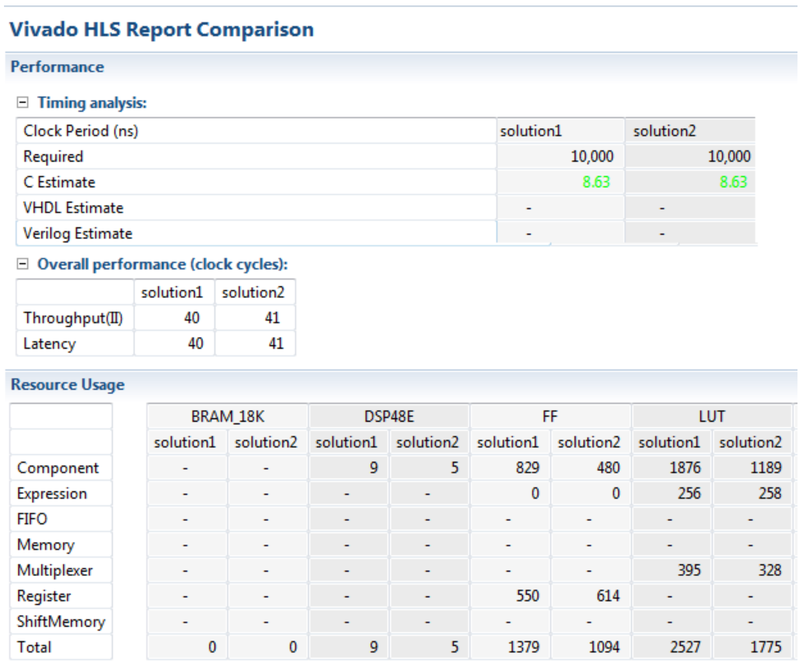
Therefore, Vivado HLS is directed to limit these area-expensive resources, by adopting only a single addsub and multiplier component. These directives are shown in Figure 13.

```
set_directive_allocation -limit 1 -type core "PID_Controller" fAddSub
set_directive_allocation -limit 1 -type core "PID_Controller" fMul
```

Figure 13: **Directive to Limit Floating Point Resources**

You can re-synthesize the design with these optimization directives to create a new solution.

Figure 14 reports the comparison of the two solutions. As expected in the new solution2, there are fewer resources being used. However, the design now takes 41 clock cycles to complete instead of the 40 clock cycles of solution1. For a design that takes one clock cycle more, we have 44% less DSP48, 20% fewer flip-flops, and 30% less LUTs.



Vivado HLS Report Comparison

Performance

Timing analysis:

Clock Period (ns)	solution1	solution2
Required	10,000	10,000
C Estimate	8.63	8.63
VHDL Estimate	-	-
Verilog Estimate	-	-

Overall performance (clock cycles):

	solution1	solution2
Throughput(II)	40	41
Latency	40	41

Resource Usage

	BRAM_18K		DSP48E		FF		LUT	
	solution1	solution2	solution1	solution2	solution1	solution2	solution1	solution2
Component	-	-	9	5	829	480	1876	1189
Expression	-	-	-	-	0	0	256	258
FIFO	-	-	-	-	-	-	-	-
Memory	-	-	-	-	-	-	-	-
Multiplexer	-	-	-	-	-	-	395	328
Register	-	-	-	-	550	614	-	-
ShiftMemory	-	-	-	-	-	-	-	-
Total	0	0	9	5	1379	1094	2527	1775

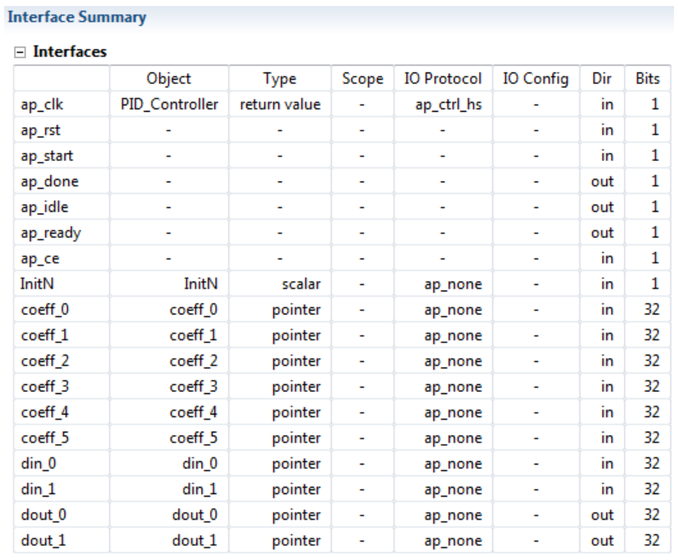
Figure 14: **Design Comparison Between 32-Bit Floating Point Solutions 1 and 2**

The estimated clock period of 8.63 ns means that we have an output data rate of 2.51 Millions of Samples Per Second (MSPS), corresponding to a signal period of 41x8.63 ns; that is, 354 ns.

We can do a further architectural analysis: we can tell Vivado HLS to not implement any FIFO, by breaking each input signal into multiple discrete ports, through the directives shown in Figure 15, in order to generate the IO ports reported in Figure 16, page 21.

```
set_directive_array_partition -type complete -dim 1 "PID_Controller" coeff
set_directive_array_partition -type complete -dim 1 "PID_Controller" din
set_directive_array_partition -type complete -dim 1 "PID_Controller" dout
set_directive_interface -mode ap_ctrl_hs "PID_Controller"
set_directive_interface -mode ap_none "PID_Controller" dout
```

Figure 15: **New IO Protocols**



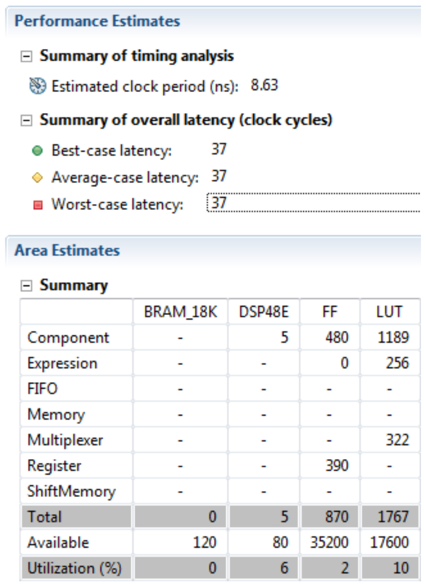
Interface Summary

Interfaces

	Object	Type	Scope	IO Protocol	IO Config	Dir	Bits
ap_clk	PID_Controller	return value	-	ap_ctrl_hs	-	in	1
ap_rst	-	-	-	-	-	in	1
ap_start	-	-	-	-	-	in	1
ap_done	-	-	-	-	-	out	1
ap_idle	-	-	-	-	-	out	1
ap_ready	-	-	-	-	-	out	1
ap_ce	-	-	-	-	-	in	1
InitN	InitN	scalar	-	ap_none	-	in	1
coeff_0	coeff_0	pointer	-	ap_none	-	in	32
coeff_1	coeff_1	pointer	-	ap_none	-	in	32
coeff_2	coeff_2	pointer	-	ap_none	-	in	32
coeff_3	coeff_3	pointer	-	ap_none	-	in	32
coeff_4	coeff_4	pointer	-	ap_none	-	in	32
coeff_5	coeff_5	pointer	-	ap_none	-	in	32
din_0	din_0	pointer	-	ap_none	-	in	32
din_1	din_1	pointer	-	ap_none	-	in	32
dout_0	dout_0	pointer	-	ap_none	-	out	32
dout_1	dout_1	pointer	-	ap_none	-	out	32

Figure 16: **Rtl Ports of the Best Performance 32-Bit Floating Point Solution**

As shown in Figure 17, this new solution3 takes even fewer clock cycles and Flip-Flop and LUT resources than the previous solutions, in detail: 37 clock cycles (corresponding to a data rate of 3.1 MSPS), 5 DSP48, 870 FFs and 1770 LUTs. This is the best performance design we will export in System Generator for DSP, as described in the Section related to it.



Performance Estimates

Summary of timing analysis

Estimated clock period (ns): 8.63

Summary of overall latency (clock cycles)

● Best-case latency:	37
◆ Average-case latency:	37
■ Worst-case latency:	37

Area Estimates

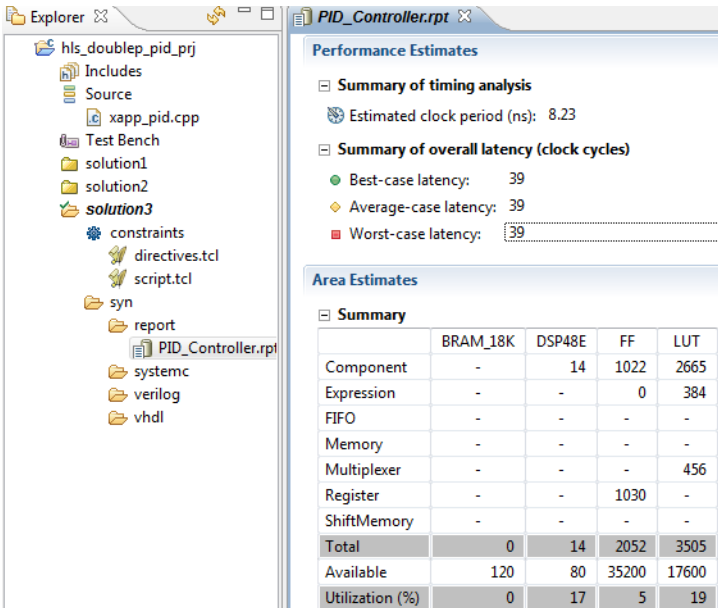
Summary

	BRAM_18K	DSP48E	FF	LUT
Component	-	5	480	1189
Expression	-	-	0	256
FIFO	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	322
Register	-	-	390	-
ShiftMemory	-	-	-	-
Total	0	5	870	1767
Available	120	80	35200	17600
Utilization (%)	0	6	2	10

Figure 17: **Best Performance Design in 32-Bit Floating Point**

The key point here is that with Vivado HLS, you can easily create a new more optimal RTL design. The knowledge required to make use of the optimization directives is an understanding of standard IO protocols and an appreciation of Xilinx FPGA resources.

Finally, just as a curiosity, if we enable the C preprocessing symbol PID_DOUBLEPREC, we get a 64-bit floating point design (C double data types). Adopting the same directives we used in the previous design (illustrated in Figure 12, page 19, Figure 13, page 20 and Figure 15, page 20), we obtain a performance of 39 clock cycles latency, 14 DSP48, 2052 FFs and 3505 LUTs, which is 2-3 times more resources than the 32-bit floating point design, as shown in Figure 18, page 22.



Performance Estimates

Summary of timing analysis

Estimated clock period (ns): 8.23

Summary of overall latency (clock cycles)

Best-case latency:	39
Average-case latency:	39
Worst-case latency:	39

Area Estimates

Summary

	BRAM_18K	DSP48E	FF	LUT
Component	-	14	1022	2665
Expression	-	-	0	384
FIFO	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	456
Register	-	-	1030	-
ShiftMemory	-	-	-	-
Total	0	14	2052	3505
Available	120	80	35200	17600
Utilization (%)	0	17	5	19

Figure 18: **Best Performance Design in 64-Bit Floating Point**

Optimization for System Architects: 25 bits fixed-point

When code is written to execute on a CPU, coding optimizations are sometimes made in order to improve performance. As noted earlier, when code is written to for a DSP, coding optimizations are often required to meet performance. It's no different for a CPU target or for an FPGA.

As already seen in the previous section, 32- and 64-bit floating-point calculations can be performed on an FPGA with relative ease. However, floating-point operators require more FPGA hardware resources than those required for fixed-point data types. Vivado HLS provides a class library to model fixed-point numbers which in some cases can be used in place of floating-point types. Fixed-point types do not have the same dynamic range as float or double types but if the range of the application can be modeled with fixed-point types, they do provide much more optimal hardware.

The `ap_fixed` C++ class can be used to specify fixed-point numbers of any arbitrary bit-width: the number of integer bits, fractional bits, quantization and overflow modes are configurable by the user. A fixed-point data type is specified in the C++ code as:

```
ap_[u]fixed<W,I,Q,O,N> my_variable
```

Table 1 shows the type specifiers.

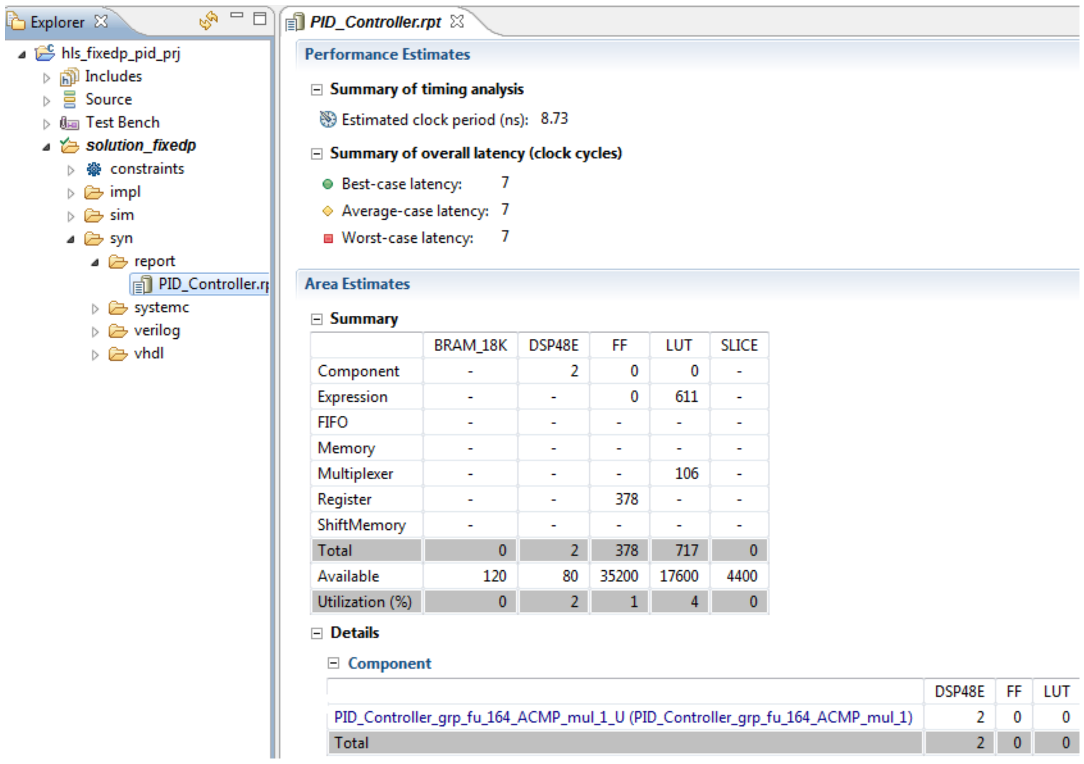
Table 1: Fixed-Point Types

Type Specifier	Description
W	Total Bit-width==7
I	Number of integers bits (number of fractional bits is W-I)
Q	Quantization mode: AP_RND Rounding to plus infinity AP_RND_ZERO Rounding to zero AP_RND_MIN_INF Rounding to minus infinity AP_RND_INF Rounding to infinity AP_RND_CONV Convergent rounding AP_TRN Truncation to minus infinity AP_TRN_ZERO Truncation to zero (default)
O	Overflow mode: AP_SAT Saturation AP_SAT_ZERO Saturation to zero AP_SAT_SYM Symmetrical saturation AP_WRAP Wrap around (default) AP_WRAP_SM Sign magnitude wrap around
N	Selects the number of bits to use in overflow modes AP_WRAP and AP_WRAP_SM

To move from the existing 32-bit floating-point example to a fixed-point one, the data_type specified in the header file can be updated to a 25-bit type with 10 integer bits if we enable the C preprocessing symbol PID_FIXEDPOINT, previously shown in Figure 7, page 15.

If the synthesis is done with this new `ap_fixed` type, the result is a design which takes fewer clock cycles to complete and with much less resources, as illustrated in Figure 19, page 24. In particular:

- There are only 7 clock cycles of latency; therefore the output data rate is 16.6 MSPS.
- The estimated resources are 2 DSP48 units, 380 FFs, and 720 LUTs, which is almost 50% the 32-bit floating-point resources utilization.



Performance Estimates

Summary of timing analysis

Estimated clock period (ns): 8.73

Summary of overall latency (clock cycles)

● Best-case latency:	7
◇ Average-case latency:	7
■ Worst-case latency:	7

Area Estimates

Summary

	BRAM_18K	DSP48E	FF	LUT	SLICE
Component	-	2	0	0	-
Expression	-	-	0	611	-
FIFO	-	-	-	-	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	106	-
Register	-	-	378	-	-
ShiftMemory	-	-	-	-	-
Total	0	2	378	717	0
Available	120	80	35200	17600	4400
Utilization (%)	0	2	1	4	0

Details

Component

	DSP48E	FF	LUT
PID_Controller_grp_fu_164_ACMP_mul_1_U (PID_Controller_grp_fu_164_ACMP_mul_1)	2	0	0
Total	2	0	0

Figure 19: **25-bit Fixed-Point Results**

The optimization directives suitable for this fixed point version of the design are shown in Figure 20.

```
set_directive_allocation -limit 1 -type operation "PID_Controller" mul
set_directive_array_partition -type complete -dim 1 "PID_Controller" coeff
set_directive_array_partition -type complete -dim 1 "PID_Controller" din
set_directive_array_partition -type complete -dim 1 "PID_Controller" dout
set_directive_interface -mode ap_ctrl_hs "PID_Controller"
set_directive_interface -mode ap_none "PID_Controller" dout
```

Figure 20: **Fixed-Point Optimization Directives**

Whether or not this is a better design depends on whether the fixed-point types can satisfy the accuracy the application requires. With the input stimuli applied in the current testbench, the effective accuracy is very close to the 32-bit floating point one, but it would be safer to do a more exhaustive numerical analysis.

Analysis with System Generator for DSP

System Generator is a DSP design tool from Xilinx that enables the use of the MathWorks model-based Simulink design environment for FPGA design. Similar to Vivado HLS, previous experience with Xilinx FPGAs or RTL design methodologies are not required when using System Generator. Designs are captured in the DSP friendly Simulink modeling environment using a Xilinx specific block set. All of the downstream FPGA implementation steps, including synthesis and place and route, are automatically performed to generate an FPGA programming file. More details on this design tool are provided in UG640, *System Generator for DSP User Guide* [Ref 3].

Using System Generator for DSP (Sysgen) is so easy and intuitive that you can design and validate the PID controller in just few hours, with the 32-bit floating-point basic blocks. The Derivative and Integrator stages of Equation 8 and Equation 12, respectively, are mapped straightforwardly in Simulink and Sysgen, as illustrated in Figure 21 and Figure 22, page 26. The PID controller top level is shown in Figure 23, page 26.

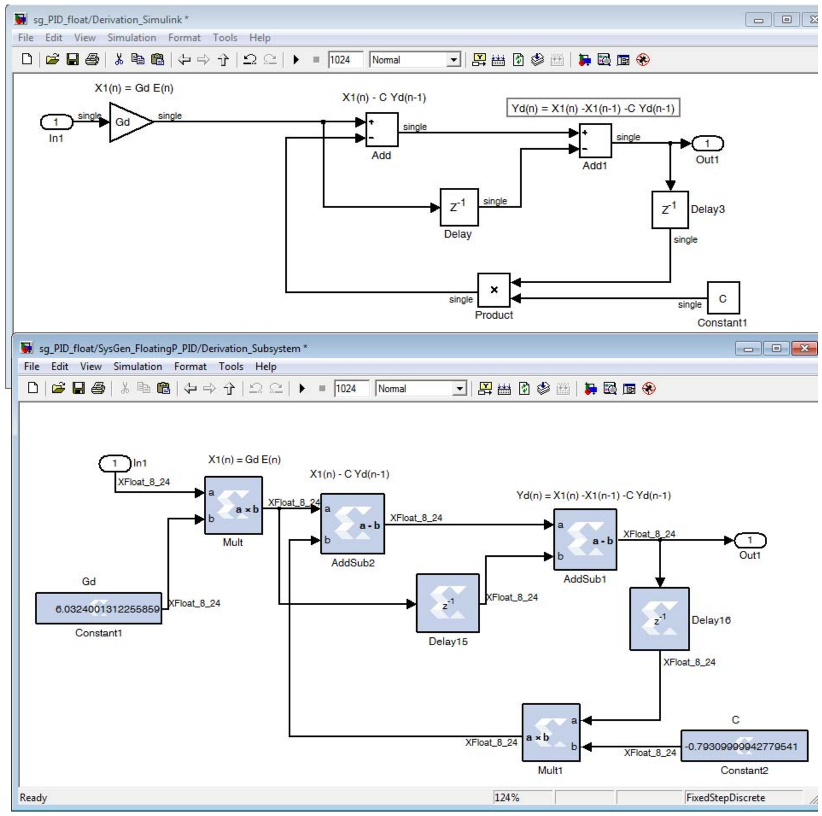


Figure 21: **Simulink (top) and Sysgen (bottom) Derivation Stage of the PID Controller**

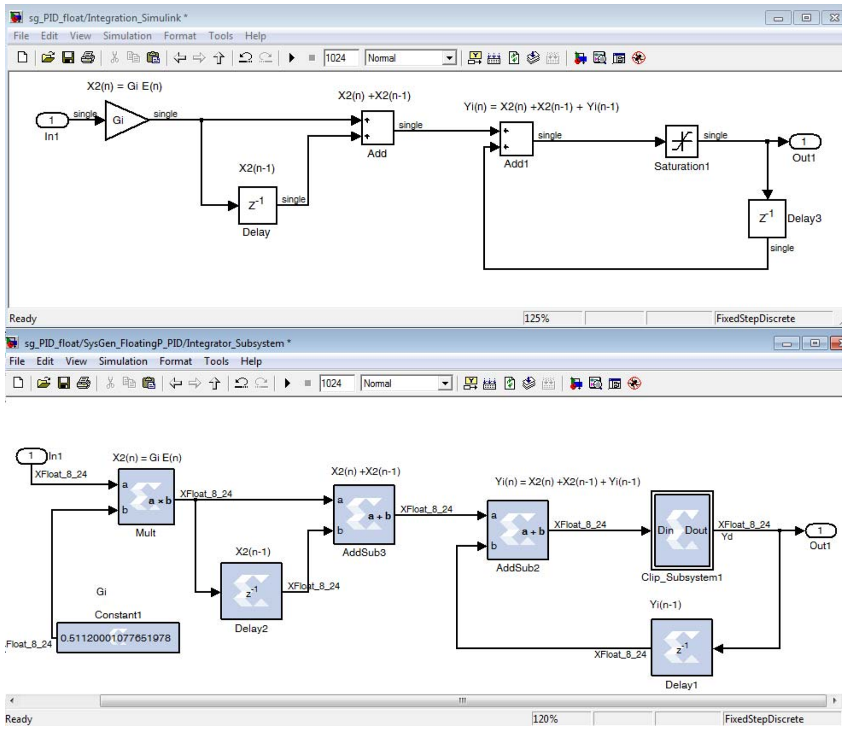


Figure 22: **Simulink (top) and Sysgen (bottom) Integrator Stage of the PID Controller**

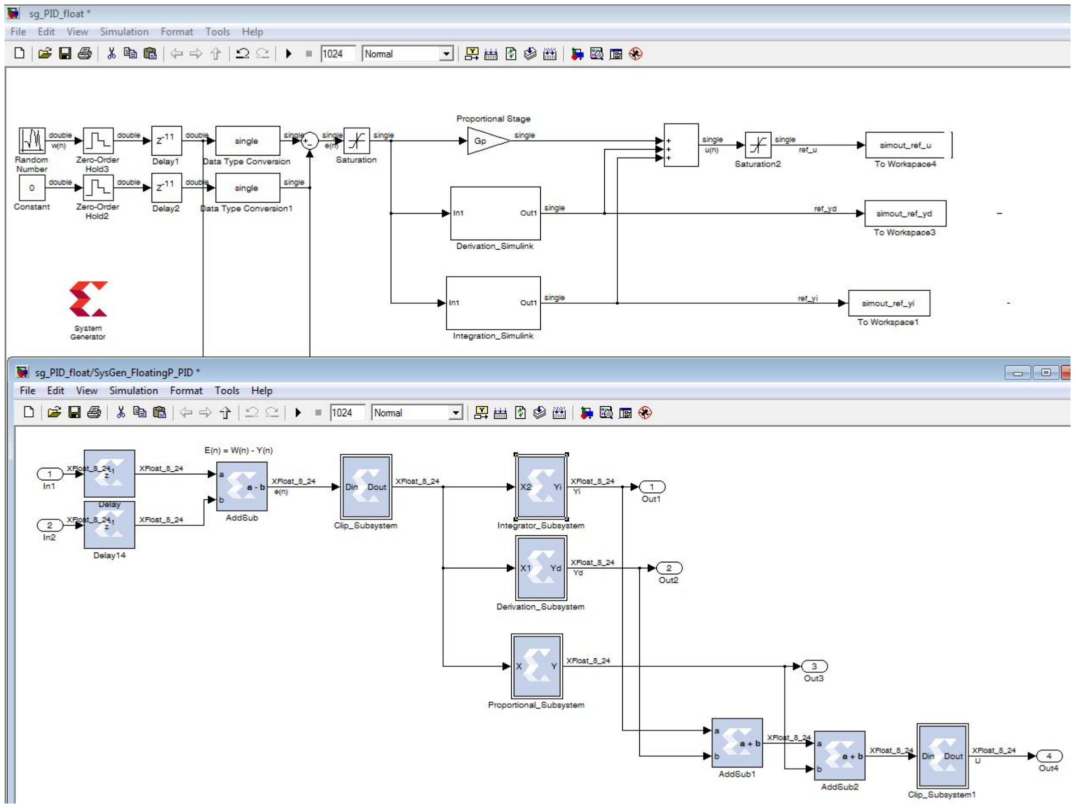
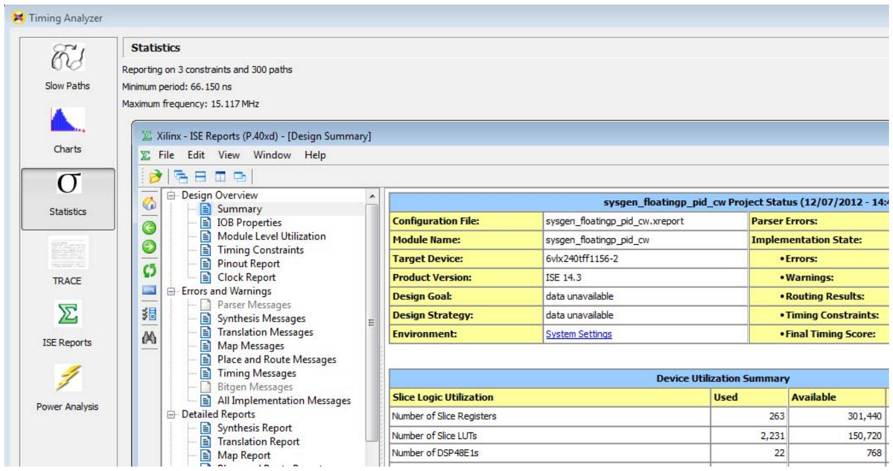


Figure 23: **Simulink (top) and Sysgen (bottom) Top Level of the PID Controller**

The numerical accuracy of the Sysgen model is excellent: the Integrator and Derivation results match perfectly the ones of the Simulink ideal subsystems, while the whole PID result is accurate within a very small error in the order of 6e-5.

Note that in floating point arithmetic, numbers expected to be equal (for example when calculating the same result through different correct methods) often differ slightly, but as long as this difference remains small, it can usually be ignored, as in our effective case. For more details about floating point arithmetic features and caveats, refer to “Floating-Point Design with Xilinx's Vivado HLS," [Ref 4].

From an FPGA performance point of view, the PID controller you implemented in Sysgen is not optimized at all in terms of area occupation, because we have applied all parallel resources. In fact, after place and route, the resources utilization is: 263 FFs, 2231 LUTs, 22 DSP48 slices, and 15 MSPS data rate (equal to the FPGA clock frequency of 15 MHz), as reported in the Timing Analysis report (see Figure 24). However, sharing in time-division the same multiplication block or the same addition/subtraction block to reduce both LUT and DSP48 units in the design would have required much longer development time (estimated in 2-4 days), because of no automation (that is, the designer must do such optimization by hand). On the other hand, as explained in the previous section, Vivado HLS dramatically shortens the development time and makes architectural exploration very easy and effective. Furthermore, we have implemented such a PID controller in Sysgen only to validate the design that was done with Vivado HLS and to show a different alternative design.



Statistics

Reporting on 3 constraints and 300 paths
Minimum period: 66.150 ns
Maximum frequency: 15.117 MHz

sysgen_floatingp_pid_cw Project Status (12/07/2012 - 14:

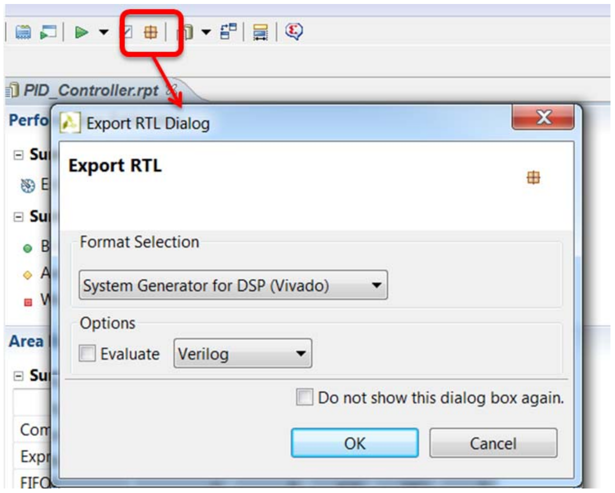
Configuration File:	sysgen_floatingp_pid_cw.xreport	Parser Errors:
Module Name:	sysgen_floatingp_pid_cw	Implementation State:
Target Device:	6vlx240tff1156-2	•Errors:
Product Version:	ISE 14.3	•Warnings:
Design Goal:	data unavailable	•Routing Results:
Design Strategy:	data unavailable	•Timing Constraints:
Environment:	System Settings	•Final Timing Score:

Device Utilization Summary

Slice Logic Utilization	Used	Available
Number of Slice Registers	263	301,440
Number of Slice LUTs	2,231	150,720
Number of DSP48E1s	22	768

Figure 24: **Timing Analysis and ISE Report for the Sysgen PID Controller Standalone**

You can export the RTL design created by Vivado HLS to System Generator for DSP. From the Vivado HLS interface menu, click the **Export RTL** toolbar button, select **System Generator for DSP**, and export the IP (see Figure 25). For information about integrating HLS subsystems into Sysgen designs, refer to UG871, *Vivado Design Suite Tutorial: High-Level Synthesis* [Ref 5].



Format Selection: System Generator for DSP (Vivado)

Options: ☐ Evaluate Verilog

☐ Do not show this dialog box again.

Figure 25: **Export the RTL as IP**

Figure 26 illustrates the top-level model containing the 3 PID subsystems at open-loop: from the top to the bottom there are the Simulink, Sysgen, and HLS PID subsystems; their output signals are respectively named `ref_u`, `sg_u`, and `hls_u`. Similarly, the partial results of the derivative and integrator stages are named `ref_yd`, `sg_yd`, `hls_yd`, and `ref_yi`, `sg_yi`, and `hls_yi`. The Sysgen cycle accurate simulation generates signals that can be viewed either as waveforms (see Figure 27), or as vectors to be analyzed with a MATLAB script (see Figure 28, page 29).

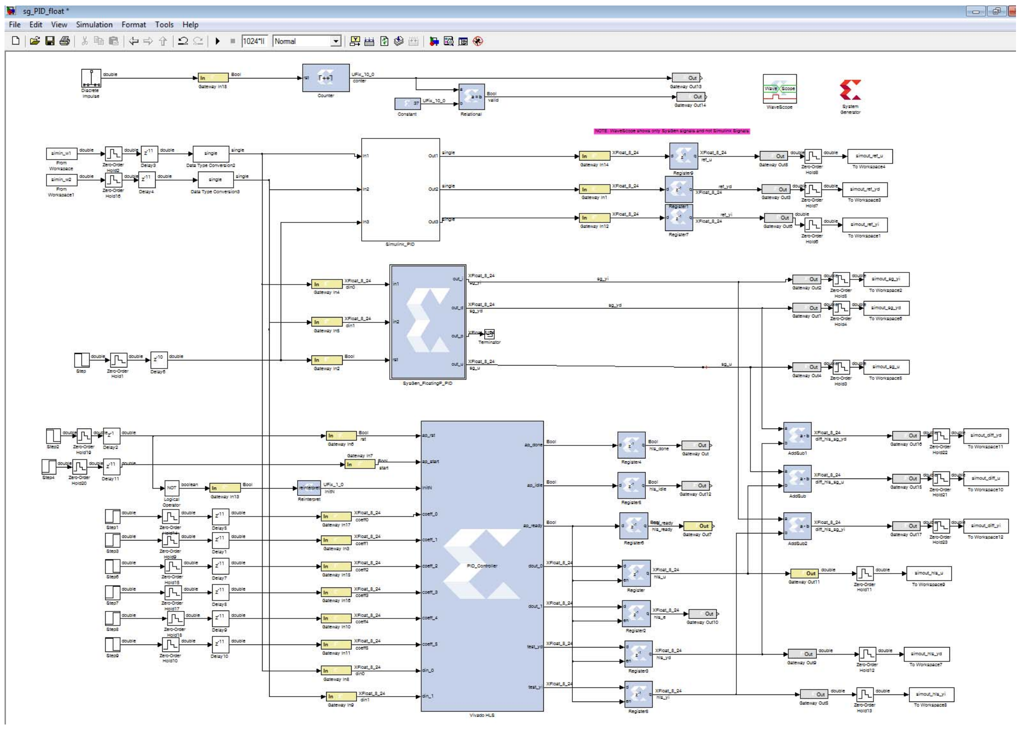


Figure 26: **Open-Loop PID Subsystems in Simulink (top), Sysgen (centre) and HLS (bottom)**

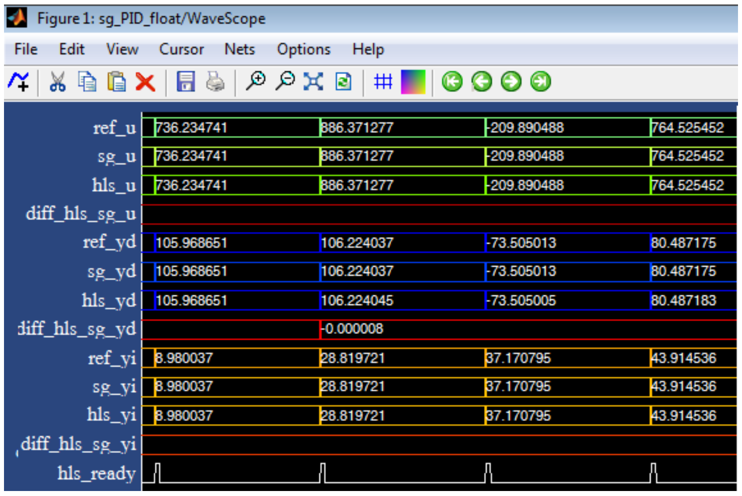
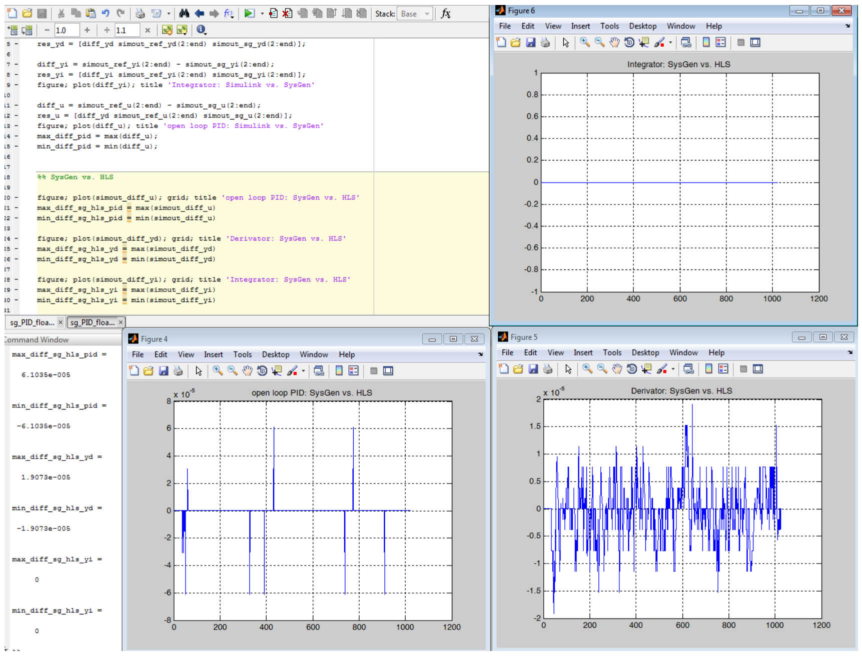


Figure 1: sg_PID_float/WaveScope
File Edit View Cursor Nets Options Help
ref_u 736.234741 886.371277 -209.890488 764.525452
sg_u 736.234741 886.371277 -209.890488 764.525452
hls_u 736.234741 886.371277 -209.890488 764.525452
diff_hls_sg_u
ref_yd 105.968651 106.224037 -73.505013 80.487175
sg_yd 105.968651 106.224037 -73.505013 80.487175
hls_yd 105.968651 106.224045 -73.505005 80.487183
diff_hls_sg_yd -0.000008
ref_yi 8.980037 28.819721 37.170795 43.914536
sg_yi 8.980037 28.819721 37.170795 43.914536
hls_yi 8.980037 28.819721 37.170795 43.914536
diff_hls_sg_yi
hls_ready

Figure 27: **Sysgen Partial Simulation Results Analyzed with WaveScope Tool**


```
    res_yd = [diff_yd simout_ref_yd(2:end) simout_sg_yd(2:end)];

    diff_yi = simout_ref_yi(2:end) - simout_sg_yi(2:end);
    res_yi = [diff_yi simout_ref_yi(2:end) simout_sg_yi(2:end)];
    figure; plot(diff_yi); title 'Integrator: Simulink vs. SysGen'

    diff_u = simout_ref_u(2:end) - simout_sg_u(2:end);
    res_u = [diff_yd simout_ref_u(2:end) simout_sg_u(2:end)];
    figure; plot(diff_u); title 'open loop PID: Simulink vs. SysGen'
    max_diff_pid = max(diff_u);
    min_diff_pid = min(diff_u);


    %% SysGen vs. HLS

    figure; plot(simout_diff_u); grid; title 'open loop PID: SysGen vs. HLS'
    max_diff_sg_hls_pid = max(simout_diff_u)
    min_diff_sg_hls_pid = min(simout_diff_u)

    figure; plot(simout_diff_yd); grid; title 'Derivator: SysGen vs. HLS'
    max_diff_sg_hls_yd = max(simout_diff_yd)
    min_diff_sg_hls_yd = min(simout_diff_yd)

    figure; plot(simout_diff_yi); grid; title 'Integrator: SysGen vs. HLS'
    max_diff_sg_hls_yi = max(simout_diff_yi)
    min_diff_sg_hls_yi = min(simout_diff_yi)
```

```
max_diff_sg_hls_pid =

  6.1035e-005


min_diff_sg_hls_pid =

 -6.1035e-005


max_diff_sg_hls_yd =

  1.9073e-005


min_diff_sg_hls_yd =

 -1.9073e-005


max_diff_sg_hls_yi =

     0


min_diff_sg_hls_yi =

     0
```
Figure 6
Integrator: SysGen vs. HLS
Figure 4
open loop PID: SysGen vs. HLS
x 10^-5
Figure 5
Derivator: SysGen vs. HLS
x 10^-5
Command Window
sg_PID_floa...
File Edit View Insert Tools Desktop Window Help

Figure 28: **MATLAB Script Check-Up on Sysgen Simulation Results Accuracy**

Source Files and Project Directories

The Xilinx design tools here adopted are Vivado HLS release 2012.3 and System Generator for DSP and ISE releases 14.3.

The source code of this Application Note is organized as illustrated in Figure 29, page 30. There are three main folders:

./xapp_pid/M

./xapp_pid/hls

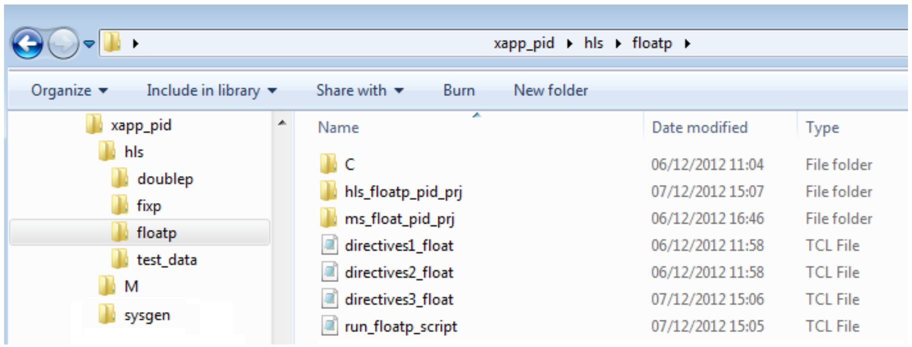
./xapp_pid/sysgen

The folder ./xapp_pid/M contains the MATLAB script xapp_pid_script.m to generate the reference results for the C program and to design the PID controller parameters. The reference results are text files placed in ./xapp_pid/hls/test_data. The script validate_pid.m can be applied to further check that the MATLAB and C generated results are matching within a certain threshold.

The folder ./xapp_pid/hls contains the C code and Tcl script for Vivado HLS. There are three included folders containing the optimizations and synthesis results: doublep (64-bit floating point design), float (32-bit floating point design), and fixp (25-bit fixed point design).

The folder ./xapp_pid/sysgen contains the Sysgen schemes:

- simulink_ClosedLoopPID.mdl, a purely ideal Simulink model of the PID and the plant subsystems in a closed loop, adopted in the next Sysgen schemes as the reference PID subsystem.
- sg_PID_float.mdl to test that PID (Simulink, Sysgen, and HLS) designs at open loop, with random input signals. It has two function callbacks: at the opening of the model in Simulink, sg_PID_float_PreLoadFcn.m is launched to initialize the MATLAB workspace, while sg_PID_float_StopFcn.m is called at the end of the Simulink simulation, to check the numerical accuracy and validity of the results.
- sg_ClosedLoop_PID_float, to test PID HLS design at closed-loop with the plant subsystem in Simulink. It has two function callbacks: sg_ClosedLoop_PID_float_PreLoadFcn.m and sg_ClosedLoop_PID_float_StopFcn.m.



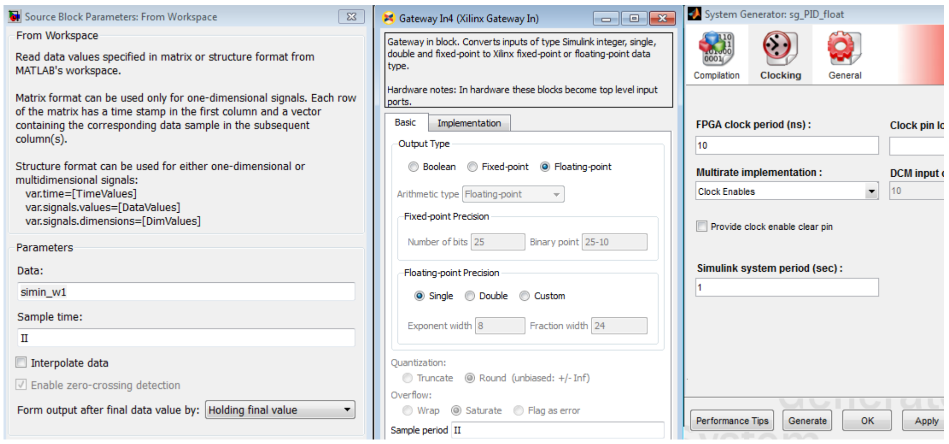
Name	Date modified	Type
C	06/12/2012 11:04	File folder
hls_floatp_pid_prj	07/12/2012 15:07	File folder
ms_float_pid_prj	06/12/2012 16:46	File folder
directives1_float	06/12/2012 11:58	TCL File
directives2_float	06/12/2012 11:58	TCL File
directives3_float	07/12/2012 15:06	TCL File
run_floatp_script	07/12/2012 15:05	TCL File

Figure 29: **Source Code Project Folders**

The procedure to build the HLS project is as follows:

1. You launch the script xapp_pid_script.m from the MATLAB command shell to create the input stimuli and the reference results in ./xapp_pid/M.
2. In ./xapp_pid/hls/floatp, you execute the following Tcl command: **vivado_hls -f run_floatp_script.tcl** from the Vivado HLS Command prompt. The Vivado HLS design tool opens; you can see the results of the various solutions in the 32-bit floating point project. The results of the Design Under Test are validated by the C++ testbench but can also be verified by running the MATLAB script **./xapp_pid/M/validate_pid.m**.

3. In `./xapp_pid/hls/double`, you execute the following Tcl command: **`vivado_hls -f run_double_script.tcl`** from the Vivado HLS command prompt. The Vivado HLS design tool opens; and you can see the results of the various solutions in the 64-bit floating point project.
4. In `./xapp_pid/hls/fixp`, you execute the following Tcl command: **`vivado_hls -f run_pid_fixp_script.tcl`** from the Vivado HLS command prompt. The Vivado HLS design tool opens; you can see the results of the various solutions in the 25-bit fixed point project.
5. The following settings are needed to run the simulations in Simulink: **Sample Time = II** for all the source blocks (either the two **From Workspace** blocks, the various **Step** blocks, or the Xilinx **Gateway In** blocks). II is the Initialization interval value declared by the Vivado HLS synthesis report of Figure 17, page 21, incremented by 1, that is 38; in fact, the input signal has a data rate that is 38 times slower than the FPGA clock period. The **From WorkSpace** blocks also need the following settings: **Interpolate data = off** and **Form output after final data value** by **Holding Final Value**. Figure 30 illustrates those settings.



Source Block Parameters: From Workspace

From Workspace

Read data values specified in matrix or structure format from MATLAB's workspace.

Matrix format can be used only for one-dimensional signals. Each row of the matrix has a time stamp in the first column and a vector containing the corresponding data sample in the subsequent column(s).

Structure format can be used for either one-dimensional or multidimensional signals:
var.time=[TimeValues]
var.signals.values=[DataValues]
var.signals.dimensions=[DimValues]

Parameters

Data: simin_w1

Sample time: II

[ ] Interpolate data

[✓] Enable zero-crossing detection

Form output after final data value by: Holding final value

Gateway In4 (Xilinx Gateway In)

Gateway in block. Converts inputs of type Simulink integer, single, double and fixed-point to Xilinx fixed-point or floating-point data type.

Hardware notes: In hardware these blocks become top level input ports.

Basic | Implementation

Output Type: Boolean Fixed-point [✓ Floating-point]

Arithmetic type: Floating-point

Fixed-point Precision: Number of bits 25 Binary point 25-10

Floating-point Precision: [✓ Single] Double Custom

Exponent width 8 Fraction width 24

Quantization: Truncate [✓ Round (unbiased: +/- Inf)]

Overflow: Wrap [✓ Saturate] Flag as error

Sample period II

System Generator: sg_PID_float

FPGA clock period (ns) : 10

Clock pin lo

Multirate implementation : Clock Enables

DCM input 10

[ ] Provide clock enable clear pin

Simulink system period (sec) : 1

Figure 30: **Simulink Simulation Settings**

6. Note that all of the above mentioned Tcl scripts contain the command `config_interface -clock_enable` to generate clock enables needed by Sysgen. This is a mandatory step and must be done before exporting the VIVADO HLS design to System Generator for DSP.

Conclusion

Floating-point designs written in C or C++ can now be quickly and easily implemented on FPGA devices, taking advantage of their massively parallel performance, low power, and low cost. As with other C/C++ flows, a full and complete tool chain allows performance trade-offs to be made throughout the flow and the results to be comprehensively analyzed. As a driver application, we took a PID controller that we optimized for either 32- or 64-bit floating point accuracy (we have even implemented it in 25-bit fixed point) using a Vivado HLS design. The design effort was really negligible. We have also adopted System Generator for DSP to validate the design in a MATLAB/Simulink testbench, due to the highly sophisticated synergy and interoperability between these two tools.

References

The current versions of the following documents are referenced in this document:

1. "Vivado HLS Eases Design of Floating-Point PID Controller," D. Bagni, G. Corradi, Xcell Journal, Fourth Quarter 2012. Available online at http://issuu.com/xcelljournal/docs/xcell_journal_issue_81/38.
2. UG902, *Vivado Design Suite User Guide: High-Level Synthesis*
3. UG640, *System Generator for DSP User Guide*
4. “Floating-Point Design with Xilinx's Vivado HLS," J. Hrica, Xcell Journal, Fourth Quarter 2012. Available online at http://issuu.com/xcelljournal/docs/xcell_journal_issue_81/28.
5. UG871, *Vivado Design Suite Tutorial: High-Level Synthesis*

Additional Information

The design files for this document are located at https://secure.xilinx.com/webreg/clickthrough.do?cid=201672.