AN EXAMPLE OF A DIGITAL SYNTHESIS APPROACH TO DSP DESIGN: THE AGS TRANSVERSE DAMPER[*]

K.A.Brown, G.Smith, V.Wong
Brookhaven National Laboratory
Upton, NY 11973

1 INTRODUCTION

Using Verilog HDL [1] and Synopsys, the digital signal processing of the AGS Transverse Damper was designed and fitted to an Altera Flex 10k FPGA. Using a control point specification style in the high level description greatly simplified the design by placing the burden of specifying the controller on the digital synthesizer.[2] The basic design and low level simulation are presented as well as the design methodology.

The purpose of the AGS Transverse Damper [3] [4] is to control instabilities and injection errors that may arise in high intensity proton beams being accelerated in the AGS. The system block diagram for the DSP is shown in Figure 1. The inputs to the system come from a normalization unit. This normalization unit takes two signals as input, a sum of beam position signal plates, and a difference from the plates. The output of the normalization unit is the difference divided by the sum. This Quotient is sent to the first ALU (as Qin[11..0]). Taking differences between position measurements the system acts as a notch filter.

The Second ALU computes a running sum of the output of the first ALU. This then acts to remove any offsets in the Quotient (and thus this part acts as a high pass filter - removing any baseline components to the signal). The depth of the first FIFO (between adder and subtract units) basically determines the low pass behaviour. The multiplier serves the purpose of overall loop gain for the system (the complete system is a real-time feedback system). The FIFO on the output is used to provide the correct amount of delay for the system.

The specifications for the design of the system are enumerated below . At power up time the FIFO's are initialized and filled with zeros.

1. 12 bit system (adder is 16 bit)
2. frequency range for clocks is 3 to 5 MHz.
3. Controls are for Gain, Initialization, Mux Selects, and external excitation.
4. number systems are all 2's complement.

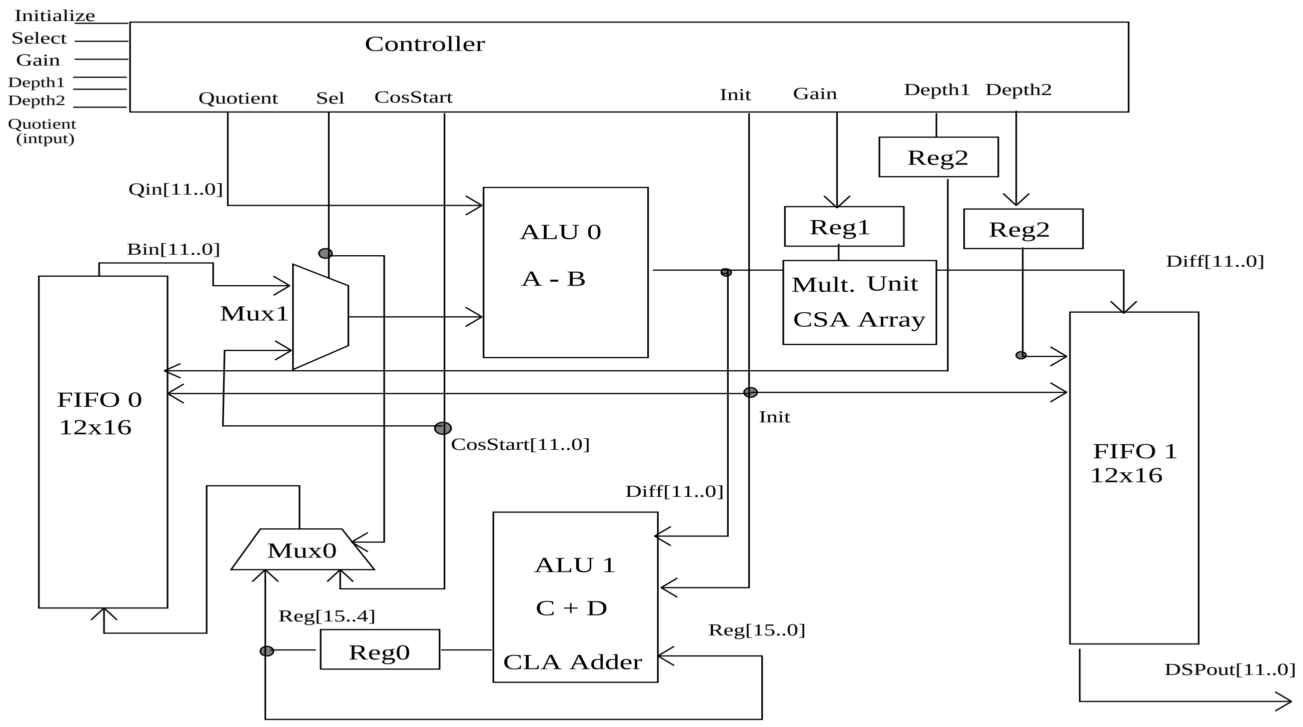


Figure 1: Block Diagram for AGS Damper DSP

[*] Work performed under the auspices of the U.S. Dept. of Energy

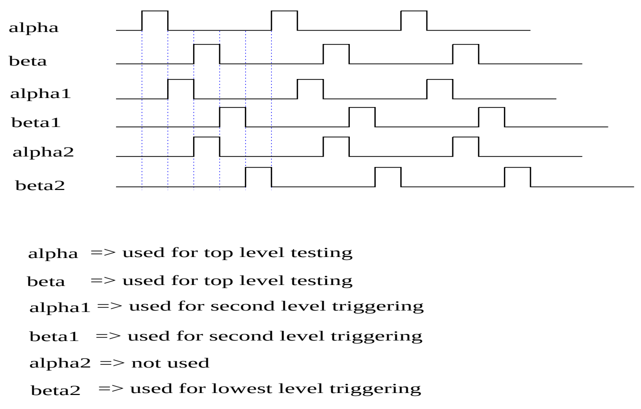


Figure 2: Multiphase timing for two level hierarchy

The design of the system is a two level hierarchy. So the triggering of the system is multiple phase. Figure 2 shows the timing sequences for the system.

2 DISCUSSION

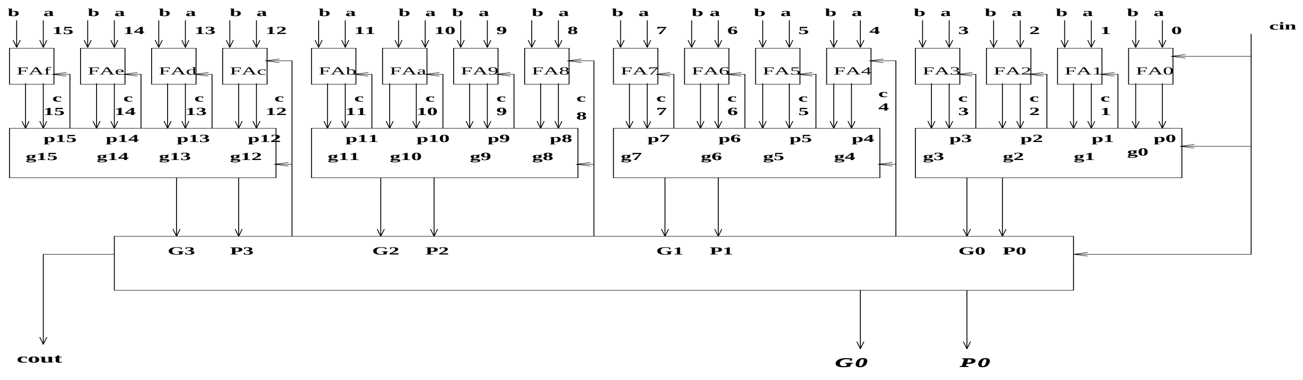
The control point specification style requires a very structured approach to the design specification.[2] The basic idea behind this style is the designer never goes into the details of building or writing specifications for the system state diagram. They need only specifiy the state tables for the state diagram and these become the control points, the inputs and outputs of a black box. This black box the synthesizer constructs based on the state tables. The benefit of this approach is seen in figure 1. The controller unit is the black box. The inputs and outputs of the controller unit are specified, but the specification of the controller itself is left up to the synthesizer. We then only need to design the specifications for the individual submodules. To illustrate this a portion of the verilog specification is shown below:

```
module damper( //input parameters );
// input/output declarations put here
...
// control points
...
wire [ws+3:0] Sout; // ALU 1 Output
reg [ws+3:0] Rout; // Reg.0 Output
...
wire [ws-1:0] Dout; // ALU 0 Output
...
// asynchronous assignments
...
// instantiate submodules
mux #(1) TestSel(SEL, 1'b0, Select, testsel);
...
// Control points are passed to the submodules as so:
// this example is a carry lookahead adder, 16 bit.

add_clh16 #(ws+4) Add_1(alpha1,
Dout[0], Dout[1], Dout[2], Dout[3], Dout[4], Dout[5], Dout[6], Dout[7],
Dout[8],Dout[9], Dout[10],Dout[11], Dout[11],Dout[11],Dout[11],Dout[11],
Rout[0], Rout[1], Rout[2], Rout[3], Rout[4], Rout[5], Rout[6], Rout[7],
Rout[8], Rout[9], Rout[10], Rout[11], Rout[12], Rout[13], Rout[14], Rout[15],
Carryin, Cout,
Sout_0, Sout_1, Sout_2, Sout_3, Sout_4, Sout_5, Sout_6, Sout_7, Sout_8,
Sout_9, Sout_a, Sout_b, Sout_c, Sout_d, Sout_e, Sout_f);

// now do the work ...
always @(posedge alpha1 or posedge beta1 or posedge init) begin
...
```

2.1 *Project Submodules*

Each block inside figure 1 corresponds to a submodule, such as the carry lookahead adder whose instantiation is shown above. It is beyond the scope of this paper to describe each one. It should suffice to explain the basic organization of one. For this example we will continue with the carry lookahead module, which is the basic module for adders and multipliers (which can be built with trees of carry save adders and end with a final full adder). This basic idea can be extended to be used for subtration and division, although these algorithms are more complicated.

Figure 3 shows the block diagram for a 16 bit carry lookahead adder. For 16 bit addition a simple ripple carry adder would not work since it needed more than 100 nsec to finished a calculation. So going to something faster was needed. Below portions of the specification are shown. It is intended only to provide a flavor for how the specificaions can easily be made to follow the flow of the block diagram.


b a 15 b a 14 b a 13 b a 12 b a 11 b a 10 b a 9 b a 8 b a 7 b a 6 b a 5 b a 4 b a 3 b a 2 b a 1 b a 0
cin
FAf FAe FAd FAc FAb FAa FA9 FA8 FA7 FA6 FA5 FA4 FA3 FA2 FA1 FA0
c 15 c 14 c 13 c 12 c 11 c 10 c 9 c 8 c 7 c 6 c 5 c 4 c 3 c 2 c 1
p15 p14 p13 p12 p11 p10 p9 p8 p7 p6 p5 p4 p3 p2 p1 p0
g15 g14 g13 g12 g11 g10 g9 g8 g7 g6 g5 g4 g3 g2 g1 g0
G3 P3 G2 P2 G1 P1 G0 P0
cout
G0
P0

G0 = g3 | (p3 & g2) | (p3 & p2 & g1) | (p3 & p2 & p1 & g0)
P0 = p3 & p2 & p1 & p0
c1 = g0 | (p0 & cin)
c2 = g1 | (p1 & g0) | (p1 & p0 & cin)
c3 = g2 | (p2 & g1) | (p2 & p1 & g0) | (p2 & p1 & p0 & cin)
c4 = G0 | (P0 & cin)

Figure 3: Carry Lookahead Adder

```
module add_clh16(clka, // parameters );
//declarations, inputs/outputs, wires, etc.
  assign g0 = A[0]&B[0], g1 = A[1]&B[1],
         g2 = A[2]&B[2], g3 = A[3]&B[3];
  assign g4 = A[4]&B[4], g5 = A[5]&B[5],
         g6 = A[6]&B[6], g7 = A[7]&B[7];
...
  assign p0 = A[0]ˆB[0], p1 = A[1]ˆB[1],
         p2 = A[2]ˆB[2], p3 = A[3]ˆB[3];
...
  assign P0 = p3&p2&p1&p0;
  assign P1 = p7&p6&p5&p4&P0;
...
  assign G0 = g4|(p4&g3)|(p4&p3&g2)|
                 (p4&p3&p2&g1)|(p4&p3&p2&p1&g0);
  assign G1 = g8|(p8&g7)|
                 (p8&p7&g6)|
                 (p8&p7&p6&g5)|
                 (p8&p7&p6&p5&g4)|
                 (p8&p7&p6&p5&p4&g3)|
                 (p8&p7&p6&p5&p4&p3&g2)|
                 (p8&p7&p6&p5&p4&p3&p2&g1)|
                 (p8&p7&p6&p5&p4&p3&p2&p1&g0);
...
always @ (posedge clka )
  begin
     sum <= A ˆ B;
carry[0] <= (g0)|((p0)&cin);
carry[1] <= (g1)|((p1)&(g0))|((p1)&(p0)&cin);
carry[2] <= (g2)|((p2)&(g1))|
                 ((p2)&(p1)&(g0))|
                 ((p2)&(p1)&(p0)&cin);
carry[3] <= (g3)|((p3)&(g2))|
                 ((p3)&(p2)&(g1))|
                 ((p3)&(p2)&(p1)&(g0))|
                 ((p3)&(p2)&(p1)&(p0)&cin);
...
  end
endmodule
```

3 CONCLUSIONS

These techniques allowed the design and simulation of a complex system to be done relatively quickly. The synthesizer is often much better at optimizations than most human beings, and the use of implicit specifications gives the synthesizer the freedom to build a circuit which fits the design requirements. Of course the designer must learn how to define the specifications such that the desired result is obtained. The real benefit of this approach is it allows for the design process to become more dynamic. It is not difficult nor very time consuming to alter the design or even change specifications. It allows the designer to design and simulate as realistically as possible the circuits they are going to build. The amount of time actually spent testing the hardware on the bench is reduced considerably. It certainly has drawbacks. The engineer now needs to learn a complicated language and become proficient in that language. The software needed to do this is expensive and needs to be configured and maintained. It also requires alot of computing power and resources in order to run well. For a large production company building DSP's or microprocessors, they simply cannot exist without software tools such as these, and necessarily make the investment. For a small lab it is understandably difficult to consider such an investment.

Results of the simulation are shown in figure 4.

4 ACKNOWLEGEMENTS

This work was done as a class project in a graduate course on Digital System Synthesis, taught by Prof. David Smith of the Dept. Computer Science, SUNY at Stony Brook. A number of students did work in relation to this work. In particular Karpagavinayagam Ramesh built a Wallace Tree adder, which was experimented with in this design.

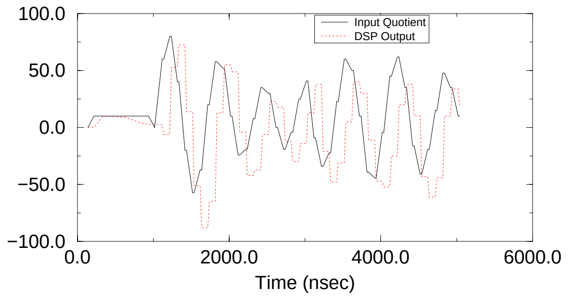


Figure 4: Simulation Results

5 REFERENCES

[1] Verilog HDL; A Guide to Digital Design and Synthesis by Samir Palnitkar, SunSoft Press, 1996.

[2] Dr. D. Smith, SUNY at Stony Brook Dept. Computer Science, Class notes from Graduate course on Digital Systems Synthesis, Fall 1996.

[3] G.A. Smith, T. Roser, R. Witkover, V. Wong, "Transverse Beam Dampers for the Brookhaven AGS" AIP Conference Proceedings 319, Beam Instrumention Workshop, Santa Fe, NM 1993, pp309-318

[4] G.A.Smith, V. Castillo, T. Roser, W. Van Asselt, R. Witkover, and V. Wong, "Digital Transverse Beam Dampers for the Brookhaven AGS", Proceedings of the 1995 Particle Accelerator Conference and International Conference on High-Energy Accelerators, pp2678-2680