Title: Hybrid Computing on an FPGA

Introduction

When I was a freshman taking physics in 1964, I had to analyze coupled, spring-mass, harmonic oscillator data. There was one digital computer available (for the whole school!) and one analog computer. The line was long for the card punch and FORTRAN ran slowly on a machine which did about 100 multiplies/sec, so I decided to learn to use the analog computer. The analog computer was faster at the time for solving differential equations because it was fully parallel. Each separate addition, integral, and multiply happened at the same time on the machine, so results were immediately available on the attached analog pen-plotter.

Now I teach a course at Cornell University, ECE 576, which uses field-programmable gate arrays (FPGAs) to build embedded systems which may include processor cores, DSP units, I/O units, and other specialized processing. To get good performance from a FPGA, you need to execute as many operations as possible in parallel because the clock rate is relatively slow compared to a dedicated CPU. I wondered if I could simulate the parallel functions of an analog computer on the FPGA and get the advantages of parallel execution, while being able to use the extremely handy ability to implement a general-purpose CPU on the same chip to control the "analog" simulation.

The hardware and software.

Several courses at Cornell use the Altera DE2 educational board. This board has a fairly large CycloneII FPGA and a wide range of hardware interfaces including LEDs, switches, audio I/O codec, ntsc video input codec, VGA output, USB interfaces, SD card interface, ethernet controller, serial port, IRDA interface, 512 Kbyte of SRAM, 8 Mbyte of SDRAM, 4 Mbyte of flash and 80 lines of general-purpose lines arranged as two 40-pin connectors. The price is $269 for academic users.

The CycloneII FPGA on the board has about 33,000 logic elements, each of which can each be configured as an arbitrary logic function 4-input gate with output register bit, and with support for arithmetic. Logic elements are connected via switches to each other and to memory and i/o to form a specified circuit. Implementing a 32-bit CPU takes less than 10% of the logic units. In addition to the general-purpose logic units there are 105 blocks of 4Kbit memory, 35 18-bit hardware multipliers, and 4 phase-locked loops.

To control all of this hardware there is a downloadable suite of software called QuartusII for Education and NiosII for Education. QuartusII allows a student to design hardware in Verilog, VHDL or by schematic, and includes point-and-click CPU design for the NiosII soft processor. The NiosII software provides a GCC development environment for the processor you just built, including a custom C-library specific to your design. Two articles in Circuit Cellar (June/July 2004) explain how to build a Nios proessor in the FPGA and how to program the processor in GCC (see resources section).

We have found that the Verilog hardware definition language is good for advanced design because it is less bulky than VHDL, but easier for big designs than schematic capture. A module of QuartusII, called SPOC builder, generates a CPU design in Verilog based on a series of specifications which the student chooses. The specifications can include the number and bit-width of i/o ports, number of hardware timers, memory type (on chip, SRAM, SDRAM) and size to be used, cache size, and pipeline options. Multiprocessor designs are supported with a defined bus structure, shared hardware mutex structures, and shared mailboxes. It is possible to design and build a fully functional 32 bit CPU in a few minutes.

Simulating Analog Computation

Traditionally, analog computers use operational amplifiers to implement addition, subtraction and to compute time-integrals of functions. Multiplication was handled by analog multipliers. All of these operations are easy to perform on the FPGA, except for computing time integrals. However, around the time that digital circuitry was getting faster then analog computation, a device called a Digital Differential Analyzer (DDA) was invented to simulate the integral function digitally. I used a version of the DDA which was easy to implement on the FPGA and which gives speedy execution.

But first, what the problem we are trying to solve? Typically physical simulations involve the solution of differential equations. The differential equations fall directly out of Newton's F=ma because forces are often related to position (e.g. gravity or spring) or velocity (e.g. frictional drag) and acceleration is the second derivative of position. If we define a set of state variables to be the position and velocity of each particle of interest then the system behavior will be determined by a set of differential equations with state variables `v1` to `vm` and with arbitrary functions relating combinations of all the state variables to the rate of change of each state variable.

```
dv1/dt = f1(t,v1,v2,v3,...vm)
dv2/dt = f2(t,v1,v2,v3,...vm)
dv3/dt = f3(t,v1,v2,v3,...vm)
...
dvm/dt = fm(...)
```

To solve the equations digitally, we need to take discrete time steps. If the time step is of duration dt and we index time by n then we can use the values of the state variables at time n to compute the values at time n+1, then repeat. We will build circuitry to perform an Euler integration approximation to these equations in the form

```
v1(n+1) = v1(n) + dt*(f1(t,v1(n),v2(n),v3(n),...vm(n))
v2(n+1) = v2(n) + dt*(f2(t,v1(n),v2(n),v3(n),...vm(n))
v3(n+1) = v3(n) + dt*(f3(t,v1(n),v2(n),v3(n),...vm(n))
...
vm(n+1) = vm(n) + dt*(fm(...))
```

The number system we will use is 18-bit 2'complement, fixed point, which is compatible with the hardware multipliers and with standard addition and subtraction. Numbers are scaled so that there are 16-bits of fraction, with a sign bit and one bit of integer with a range of +1.999985 to -2.0000. Real analog computers also require amplitude scaling because they are bounded at high voltage by physical limits and at low voltage by noise.

Figure 1 shows the structure of the numerical integrator as a block diagram. Many copies of this circuit would be built on the FPGA for a real application, and all would perform their operations at the same time. Quite often, calculating F(t,V(n)) is more complicated then the integration itself.

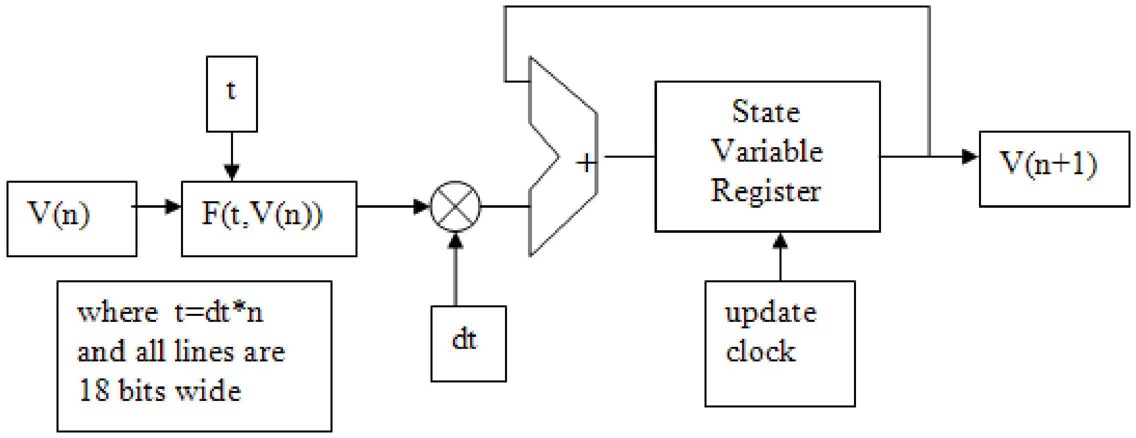


Figure 1

As an simple example, consider the linear, second-order differential equation resulting from a damped spring-mass system:

$$d^2x/dt^2 = -k/m*x-d/m*(dx/dt)$$

where k is the spring constant, d the damping coefficient, m the mass, and x the displacement of the mass. We will simulate this by converting the second-order system into a two first-order equations. If we let `v1=x` and `v2=dx/dt` then the second order equation is equivalent to

```
dv1/dt = v2
dv2/dt = -k/m*v1-d/m*v2
```

These equations can be solved by wiring together two integrators, two multipliers and an adder as shown in Figure 2. Controlling such a system to start and stop time, record variable values, and load initial conditions is easily done using a standard sequential computer. The next sections will outline the actual hardware built using Verilog to specify multiple integrators, adders, multipliers and one CPU to control it all.

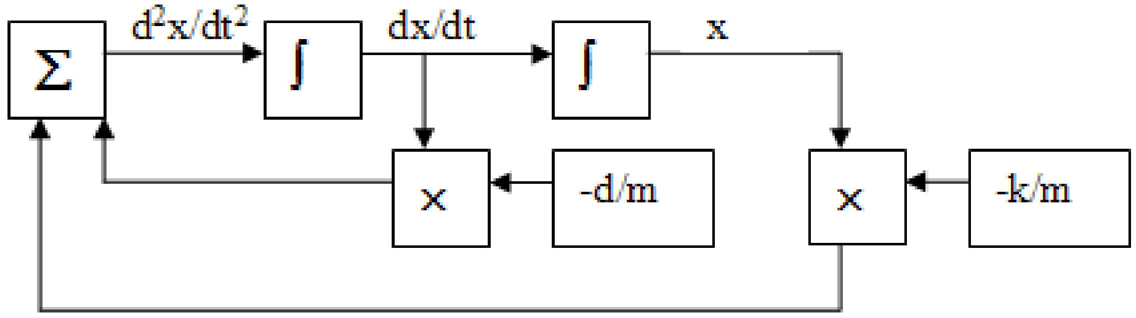


Figure 2

Verilog description of the second-order system

Verilog is a text-based, hardware description language. Superficially, it looks like a sequential programming language, but it is not. All statements in a Verilog design *execute at the same time*! You can think of each statement is defining the values on a wire or bus. Modules, which superficially resemble subroutines, cause hardware to be built and connected every time their name is used in a design.

Starting at the bottom of the DDA design, lets consider the multiplier. We are using 18-bit fixed point with the binary point between bit 15 and 16, for 16-bits of fraction and one bit of integer and a sign bit. The two 18-bit numbers result in a 36 bit result, of which bit 35 is the new sign bit, and the rest of the value is in bits 32 to 16. The Verilog is shown below. A module designed in this fashion in QuartusII will result in using one of the hardware multipliers in the FPGA, according to the Altera HDL guidelines manual. Each time this module is invoked, a new multiplier will be built.

```
module signed_mult (out, a, b);
        output          [17:0]  out;
        input   signed  [17:0]  a;
        input   signed  [17:0]  b;
        wire    signed  [17:0]  out;
        wire    signed  [35:0]  mult_out;
        assign mult_out = a * b;
        assign out = {mult_out[35], mult_out[32:16]};
endmodule
```

The integrator module updates a state variable register (v1) on the positive edge of the system clock. The multiply of the input function by dt is simplified to a shift-right. This works because the value of dt is less than one and because steps of 2 are fine enough control. The strange triple-arrow operator for shift means “shift-right-signed”.

```
module integrator(out,funct,InitialOut,dt,clk,reset);
        output [17:0] out;     //the state variable V
        input signed [17:0] funct;    //the dV/dt function
        input [3:0] dt ;       // in units of SHIFT-right
        input clk, reset;
        input signed [17:0] InitialOut; //the initial state variable V
        wire signed  [17:0] out, v1new ;
        reg signed   [17:0] v1 ;
        always @ (posedge clk)
        begin
                if (reset==0) //reset
                        v1 <= InitialOut ; //
                else
                        v1 <= v1new ;
        end
        assign v1new = v1 + (funct>>>dt) ;
        assign out = v1 ;
endmodule
```

The second-order example above can now be coded as follows. Three signed multipliers are built, and two integrators. The form 18'h0_0800 means a 18-bit hexadecimal constant with value 1/32 in the notation we are using.

```
// wire the integrators
// time step: dt = 2>>9
// v1(n+1) = v1(n) + dt*v2(n)
integrator int1(v1,v2,0,9,AnalogClock,AnalogReset);

// v2(n+1) = v2(n) + dt*(-k/m*v1(n) - d/m*v2(n))
signed_mult K_M(v1xK_M, v1, 18'h1_0000); //Mult by k/m
signed_mult D_M(v2xD_M, v2, 18'h0_0800); //Mult by d/m
//scale the input so that it does not saturate at resonance
signed_mult Sine_gain(Sinput,{sine_out[15],sine_out[15],sine_out},
{2'h0,Sgain});
integrator int2(v2,(-v1xK_M-v2xD_M+Sinput),
0,9,AnalogClock,AnalogReset);
```

The output of both the integrators (velocity and position) were wired to the audio codec (Wolfsan WM8731) stereo outputs, and displayed on a scope. The 18-bit 2's-complement notation used in the calculation was truncated to 16 bits when sent to the codec. Figure 3 shows the output. The top trace is position and the bottom is velocity. The characteristic damped sine wave of a second order system is seen. With a dt=2^{-9}, the system runs 32 times faster than real-time and had to be slowed down to use the audio codec. The resonant frequency of the simulated oscillator matched the analytical value within 0.2%.

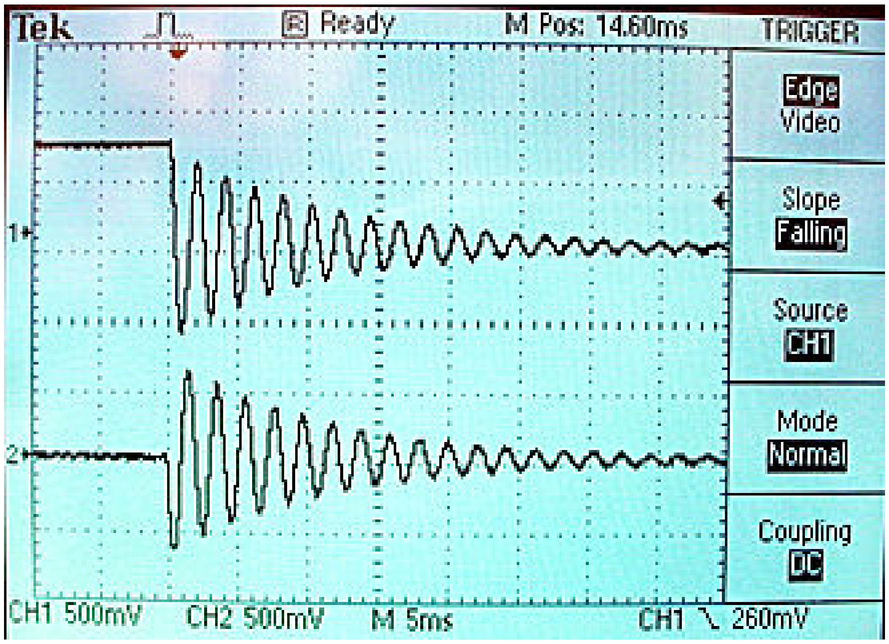
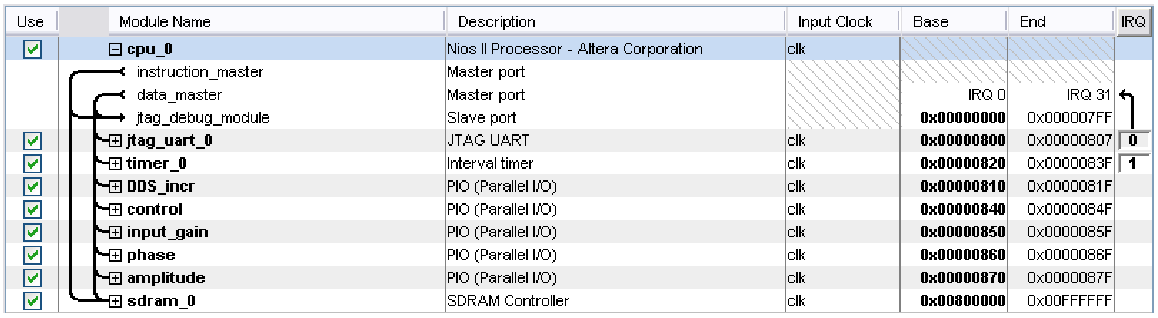


Figure 3.

Adding the NiosII for controlling the DDA subsystem

It is often useful to be able to control an analog computer using a digital computer. Such a system is called a *hybrid computer*. For example you might want to sequence through a set of input frequencies applied to the simulated system to build a Bode plot. A NiosII CPU was built on the FPGA with ports to (1) control the DDS frequency, (2) control the gain of the sine wave applied to the second order system, (3) control the analog reset and start the simulation, (4) record the amplitude and phase shift of the system under test. The top-level Verilog module contains the DDA simulation of the second order system and the NiosII cpu. The description of the NiosII CPU is shown in Figure 4 which a a screen dump of QuartusII SOPC builder for the CPU used in this design. Designing the CPU consisted of drag-and-drop operations on this table. The module name column shows the name I assigned to each i/o port. These names will be picked up by the NiosII GCC tools and used to build a C library for this configuration. This column also shows how each module is connected to data and instruction busses. The "base" and "end" columns show the address map for each module. The IRQ column shows any interrupt vectors associated with each module.



Use	Module Name	Description	Input Clock	Base	End	IRQ
✓	**cpu_0**	Nios II Processor - Altera Corporation	clk			
	instruction_master	Master port				
	data_master	Master port		IRQ 0	IRQ 31	
	jtag_debug_module	Slave port		**0x00000000**	0x000007FF	
✓	**jtag_uart_0**	JTAG UART	clk	**0x00000800**	0x00000807	0
✓	**timer_0**	Interval timer	clk	**0x00000820**	0x0000083F	1
✓	**DDS_incr**	PIO (Parallel I/O)	clk	**0x00000810**	0x0000081F	
✓	**control**	PIO (Parallel I/O)	clk	**0x00000840**	0x0000084F	
✓	**input_gain**	PIO (Parallel I/O)	clk	**0x00000850**	0x0000085F	
✓	**phase**	PIO (Parallel I/O)	clk	**0x00000860**	0x0000086F	
✓	**amplitude**	PIO (Parallel I/O)	clk	**0x00000870**	0x0000087F	
✓	**sdram_0**	SDRAM Controller	clk	**0x00800000**	0x00FFFFFF	

Figure 4

The CPU module generated by SOPC builder is completely defined by a Verilog description which is instantiated in the same module which defines the DDA hardware. There are nine control ports for the SDRAM, as well as the clock, reset, and the i/o ports I defined.

```
module hybrid_cntl (
            // 1) global signals:
             clk,reset_n, out_port_from_the_DDS_incr, in_port_to_the_amplitude,
             out_port_from_the_control, out_port_from_the_input_gain, in_port_to_the_phase,
            // the_sdram_0
             zs_addr_from_the_sdram_0,
             zs_ba_from_the_sdram_0,
             zs_cas_n_from_the_sdram_0,
             zs_cke_from_the_sdram_0,
             zs_cs_n_from_the_sdram_0,
             zs_dq_to_and_from_the_sdram_0,
             zs_dqm_from_the_sdram_0,
             zs_ras_n_from_the_sdram_0,
             zs_we_n_from_the_sdram_0
            )
```

The GCC program running on the NiosII:

1. Holds the DDA in reset, and initializes the test frequency
2. Releases the DDA reset
3. Waits until the DDA output phase and magnitude reach steady-state
4. Prints the phase and magnitude through the JTAG UART
5. Increments the frequency by a small factor
6. Repeats the above steps for a number of frequencies

A set of C macros generated by the NiosII development environment provides interfaces to the hardware. For instance to write a zero to the i/o port named "control" in the SOPC builder (Figure 4) you could use the following statement. The Avalon notation here refers to the bus structure used by the NiosII CPU. The CONTROL_BASE is the first address of the address map of the "control" port.

```
IOWR_ALTERA_AVALON_PIO_DATA(CONTROL_BASE,0);
```

The resulting phase/magnitude Bode plots in Figure 5 were produced with a small Matlab script which read the text file generated by the NiosII and also superimposed the analytical solution generated by Matlab on a PC. The top plot is log-amplitude versus log-frequency and the bottom is phase versus log-frequency. You can see that the match to the analytical solution (red curves) is good. The resonant frequency can be easily seen and the analytical value is marked with a vertical red line. You can also see some amplitude quantization error at low amplitudes on both ends of the curve.

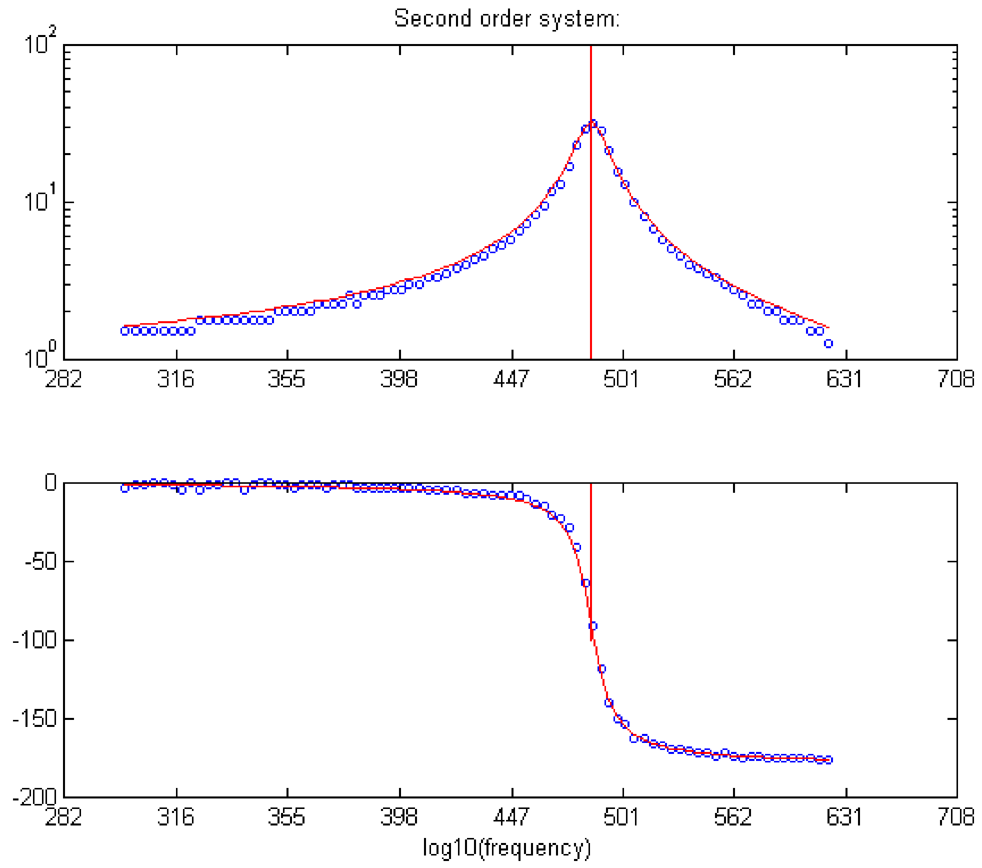


Figure 5

Conclusions

Each integrator takes about 2% of the circuitry on the FPGA and each multiplier takes 3% of the multipliers. You could thus expect to build about 50 integrators and about 30 multipliers in a larger design. If you filled the FPGA with integrators and multipliers and ran them at 50 MHz, you would expect to compute around four billion 18-bit operations/second. Of course, putting a control CPU on the FPGA would drop the number of integrators you could build.

Let's compare the experience of programming an analog computer by plugging in resistors and wires in 1964 and using Verilog and QuartusII to build one on an CycloneII

FPGA in 2006. The learning and setup time for my first design on both systems was about the same (a few days). The setup time for the second project on the FPGA was much shorter. Adding integrators in a text file is just faster than running wires. The FPGA version has a simulation bandwidth at least 500 times higher, computed as (number of integrators)*(integrator bandwidth). The analog computer ran in "continuous time" and thus had no time quantization error, but had no better than 1.0 % (7-bit) amplitude accuracy. The FPGA simulation seems to have an accuracy of 18-bits, but it is actually somewhat lower, depending on the size of dt. Picking a small dt decreases time quantization error, but increases round-off error. Extending the numerical precision, or using a higher-order integrator would help soften this tradeoff.

Resources.

Altera DE2 board.
http://altera.com/education/univ/materials/boards/unv-dev-edu-boards.html

Altera educational software page
http://altera.com/education/univ/software/unv-software.html

Altera Recommended HDL style
http://www.altera.com/literature/hb/qts/qts_qii51007.pdf

Digital Differential Analyzer: Full hybrid design details are available at
http://instruct1.cit.cornell.edu/courses/ece576/DDA/index.htm

ECE576 webpage
http://instruct1.cit.cornell.edu/courses/ece576/

Nios and GCC
(1) Circuit Cellar ISSUE 167 June 2004, p. 72; Designing with the Nios (Part 1): Second-Order, Closed-Loop Servo Control, by George Martin.
(2) Circuit Cellar ISSUE 168 June 2004, p. 36; Designing with the Nios (Part 2): System Enhancement, by George Martin.