FPGA Implementations of Initial-Value Problem Solving Methods

Nick Eyre | Olin College | December 16, 2014

***Abstract*—In this project, two methods of solving initial-value problems are explored with reference to implementation on a field programmable gate array (FPGA). FPGAs allow for many digital operations to be parallelized to improve performance. Furthermore, methods of simulating gate array systems in Simulink are explored. The methods are simulated in Simulink and also written in MIPS assembly code to compare the computational efficiency of comparable algorithms on gate array and embedded microcontroller systems. In the end, both Euler's method and a modified trapezoidal Euler's method were successfuly simulated and show promise for good performance on an FPGA.**

I. Background Information

In this paper, we explore methods of solving initial value problems. Initial value problems are ordinary differential equations along with initial conditions. These types of problems are commonly used in fields such as dynamics to solve Newton's second law for a variety of input forces.

This project will use the differential equations governing the motion of a baseball thrown vertically into the air.[1] This system is controlled by a second order differential equation which can be transformed into two first-order differential equations (Equation 1). Note that these equations ignore the force due to spin, the Magnus force.

$$\begin{aligned}\frac{dy}{dt} &= v \\ \frac{dv}{dt} &= g - \frac{\rho A C_d}{2m} v|v|\end{aligned} \tag{1}$$

One important concept is that of state variables. A state variable represents and holds on to the current state of a system (a register). In this system, we have two states: v and y. An initial-value solver uses a discrete integrator to transform the continuous state into a series of approximate discrete states.

Some realistic constants for a baseball are given below. For the purpose of our simulation, we assume that the baseball is launched upward at a velocity of 100 m/s. While this is a *bit* fast for the average pitcher, it makes our math easier.

$$\begin{aligned} g &= -9.81\,\text{m/s}^2 \\ C_d &= 3 \times 10^{-1} \\ \rho &= 1.3\,\text{kg/m}^3 \\ A &= 4.2 \times 10^{-3}\,\text{m}^2 \\ m &= 1.45 \times 10^{-1}\,\text{kg} \\ v_0 &= 1.00 \times 10^{2}\,\text{m/s}\end{aligned} \tag{2}$$

[1]Equations taken from Olin College Numerical Methods Case Study I by John Geddes, Fall 2014

Scaling to Integer Math

Note that the math becomes easier to implement on an FPGA if this system can be scaled so that all math is done as integers. To do this, we will replace the SI base units of m and s with μm (micrometers) and cs (centiseconds).

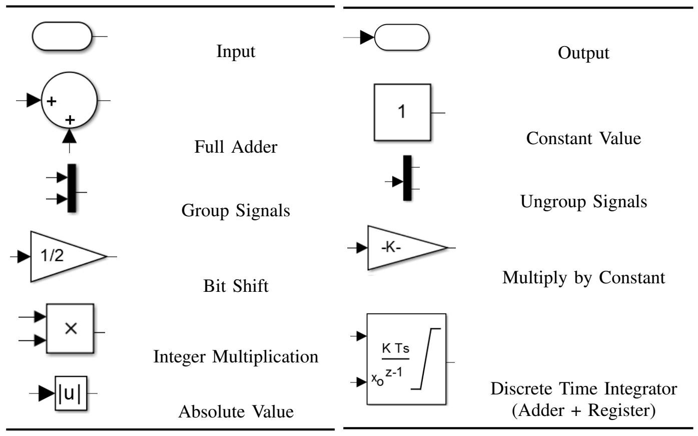
Now, our parameters are as follows:

$$\begin{aligned} g &= -9.81 \times 10^{2}\ \mu\text{m/cs}^2 \\ \frac{\rho A C_d}{2m} &= 5.9 \times 10^{-9}\ 1/\mu\text{m} \\ v_0 &= 1.000\,000 \times 10^{6}\ \mu\text{m/cs}\end{aligned} \tag{3}$$

These values, while they may seem odd, will mostly allow us to operate within the limits of our space. Note that the large initial value for velocity will not quite fit into a 32 bit signed integer. On the FPGA, this could be represented with 33 bits instead.

II. Simulink for FPGA Code Design

Simulink, part of MATLAB, is a graphical programming language that can be used for modeling systems with a high degree of concurrency. Simulink has a number of blocks that are useful for modeling systems to be implemented on FPGAs and when put into discrete time mode, it can be compiled into clocked hardware description language (HDL) code such as Verilog or VHDL.



	Input		Output
	Full Adder	1	Constant Value
	Group Signals		Ungroup Signals
1/2	Bit Shift	-K-	Multiply by Constant
×	Integer Multiplication	$\frac{K\,Ts}{z-1}$, x_0	Discrete Time Integrator (Adder + Register)
|u|	Absolute Value		

TABLE I: Simulink Symbols Used

The blocks used in this project are given in Table I. All of these blocks are easily implemented in Verilog. Note that the discrete time integrator used simply starts with an initial value x_0 and adds a derivative to a register once per clock cycle, a function which would be easily implemented in Verilog.

Also notable is that although the simulation built uses multiplication by a constant, this function would be easily replaced by easier bit shift operations by scaling all equations so that multiplication is by powers of two.

Note that although Simulink was used for modeling the systems presented in this paper, the systems were not compiled into HDL code and tested on FPGAs due to a lack of access to the necessary MATLAB toolbox.

III. Building Blocks

The first step in modeling this system is to create several reusable modules in Simulink that represent core parts of the system. These modules can be used in any initial-value solver.

Although these blocks were not implemented in Verilog, they contain components that would be easy to implement.

State Integrator Block

The first block built was a state integrator (Figure 1). This block manages the states of the system and has one discrete time integrator for each of the state variables and time. The block takes in the derivatives of each of the state variables, multiplies it by the timestep and returns the values of the state registers.

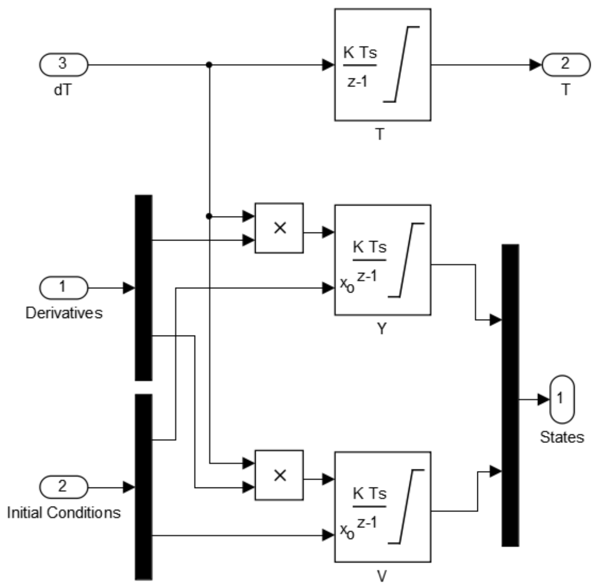


Fig. 1: State Integrator Block

Dynamic Model Block

The second block built was a block depicting the dynamic model of the system (Figure 2). This block takes the current states as an input, calculates the forces on the system based on the current states and returns the derivatives of each of the two state variables. This dynamic model accounts for the forces due to gravity and drag.

Note that drag requires squaring the value of a state. This is accomplished by putting the result of this multiplication operation into a 64-bit register and discarding the lower 32 bits when multiplying by the drag coefficient. As the scaled drag coefficient is relatively close to 2^{-32}, this does not affect the accuracy of the calculation significantly.

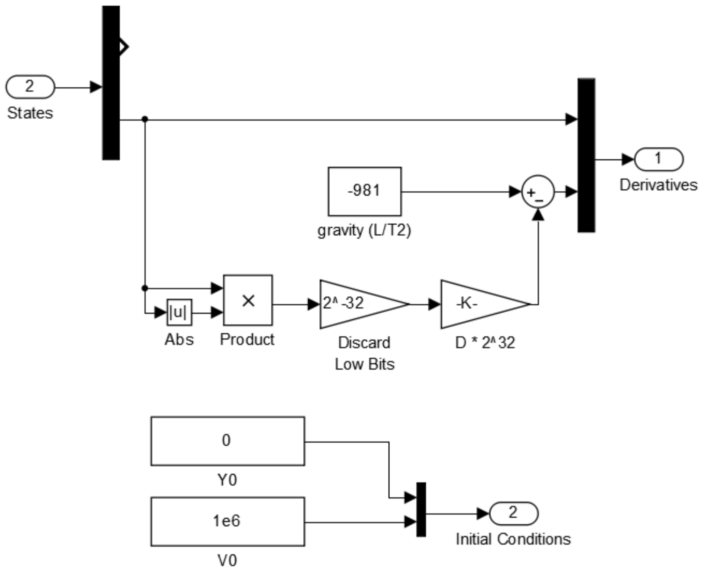


Fig. 2: Dynamic Model Block

The dynamic model block also provides the initial conditions for both of the states.

IV. Euler's Method

Euler's method is the most basic method for solving ordinary differential equations. In Euler's method, the derivatives of the state variables at each state are integrated for the full timestep. This is also known as a forward tangent approximation. Because the tangent at one state is projected forward for the next timestep without consideration for the tangent at the future point, this method has substantial error (Figure 3).

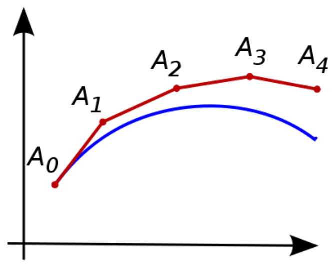


Fig. 3: Euler's Method Approximation[2]

This method was implemented in Simulink by connecting the state integrator block directly with the dynamic model block (Figure 4). This feeds the derivative of the states into the integrator to integrate it for the duration of the next timestep.

The results from this simulation are given in Figure 6 for a timestep ΔT of 32 cs.

[2]From http://en.wikipedia.org/wiki/Euler_method

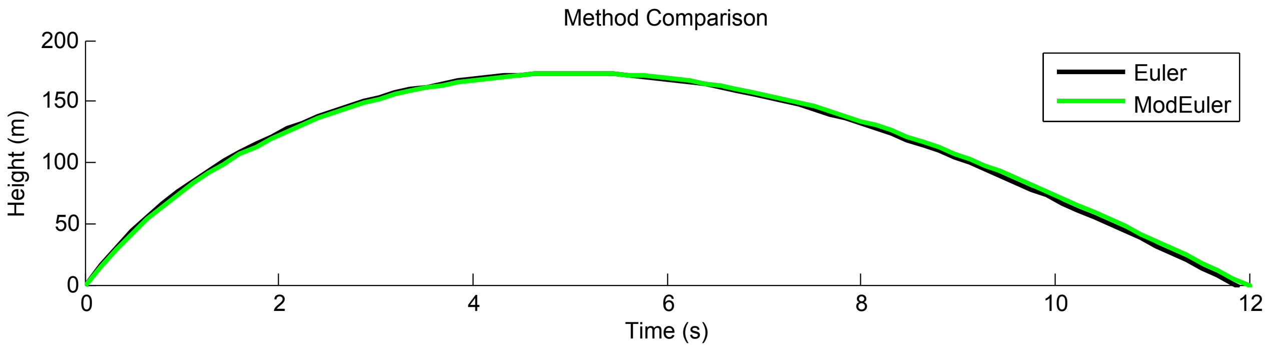


Fig. 5: Comparison of results from two methods.

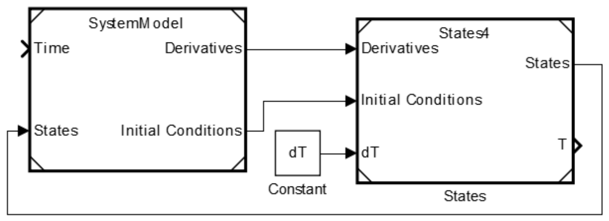


Fig. 4: Euler's Method Simulation

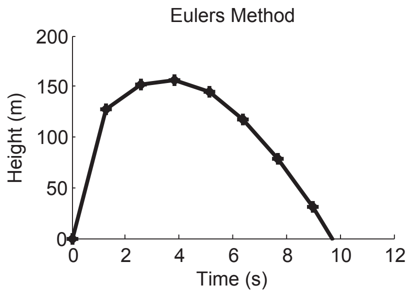


Fig. 6: Euler's Method Results

Euler's method was also implemented in MIPS assembly code to compare the performance of this Simulink model with that which could be expected on an embedded microcontroller. Obviously, the huge difference between clock speeds of our FPGA-based system and a personal computer would give a speed edge to the computer — a microcontroller is a more fair comparison. The code is given in Appendix A.

All of the logic in the Simulink model could be performed in one clock cycle on an FPGA for each time step (assuming proper support for integer multiplication). However, the assembly code written compiles to $7 + 17n$ lines of machine code where n is the number of time steps. This is much more clock cycles than required on the FPGA. If the clock speed of the FPGA could be raised high enough, this would give quite the advantage to the parallelism offered by the FPGA. Unfortunately, as this code was not put onto an FPGA, the clock speed was not able to be computed.

V. Trapezoidal Euler's Method

A modified trapezoidal Euler's method was next implemented as an attempt to increase the accuracy of the simulation. This method calculates the derivative at the current state and temporarily updates the state with this derivative. The derivative is then calculated at this new state and the two derivatives are avearged. This has the effect of calculating the derivative in both direction at every point and applying it in both directions. This is alike to the trapezoidal method of integral approximation

The method was implemented in Simulink by connecting one system model block to another to calculate the derivatives at both ends of the curve. Both derivatives are then averaged to get the derivative to be applied to the states (Figure 7).

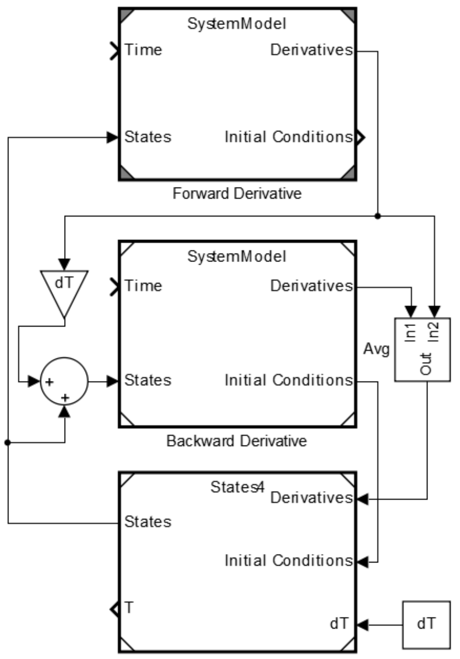


Fig. 7: Trapezoidal Euler's Method Simulation

The results of this simulation when compared to the Euler's method simulation is shown in Figure 5 for ΔT of 16 cs. As shown, for this relatively large number of time steps, the results match closely for the two methods. With a larger ΔT, the difference between the two cases would be larger.

The trapezoidal Euler's method was also implemented in MIPS assembly code as before. All of the logic in the Simulink model can still be performed in one clock cycle on an FPGA, albeit likely with a slower clock speed due to the linear chaining of computation blocks. The assembly code compiled to $7 + 33n$ clock cycles where n is the number of time steps. This is significantly more clock cycles than the FPGA which is able to compute much of the math simultaneously.

VI. Conclusions

In conclusion, two solvers were successfully designed and simulated for implemenation on an FPGA. Although these solvers were not actually synthesized onto an FPGA, they are built such that they could be, either by using the appropriate MATLAB HDL Coding toolbox or by converting the code to Verilog. Furthermore, the code was modularized such that more complicated solving algorithms such as one of the Range-Kutta methods could be implemented with relative ease.

The heavily-parallelized nature of an FPGA proves to be an advantageous way of solving initial value problems which rely on performing a large number of computations at once and then integrating these derivatives. This structure allows for more computations to happen in a given clock cycle and more things to happen at once than on a traditional computer. Although this code when synthesized will probably not support clock speeds anywhere near those offered by modern desktop computers, an FPGA could be a compelling alternative to performing dynamic simulations on a small embedded system.

However, there are still multiple items which must occur in series, especially with higher order solvers. FPGA-based initial-value problem solvers could prove even more advantageous in systems where more forces are involved and must be calculated indendently. Furthermore, difficulties may arise in dealing with more complicated systems that make heavy use of division or more complicated mathematical functions that are harder to synthesize onto an FPGA.

VII. Future Work

Several minor issues were faced over the course of this project. Most notably, I ran into an error in Simulink where the solver was unable to compute the algebraic loop in the trapezoidal solver for a timestep ΔT greater than 16 cs. I think I probably could have figured out this issue given more time. Furthermore, I tried to implement an adaptive-timestep solver using the difference between the two methods as the error but failed to implement this.

Possible Next Steps

Future work on this project could be directed into one of a number of areas:

- Extend the solvers to use floating point math. This would allow for more precise calculations and would elminate the need for scaling into strange unit systems to simplify calculations.
- Convert the Simulink code presented into synthesizable HDL code in Verilog or VHDL. This would allow for a more complete comparison of this system's performance relative to a microcontroller.
- Use the MATLAB HDL Coder Toolbox to synthesize this code to a FPGA and test the systems presented.
- Implement this system to solve equations with a greater number of state variables. This may require normalizing vectors which may require a square root function.
- Implement trigonometric functions which would allow for systems with rotation to be more easily solved.
- Implement an adaptive-timestep solver or one of the higher-order Range-Kutta methods. This would allow for more accurate calculations and may make better use of the FPGA's capabilities.

Code, Schematics & Build Instructions

The assembly code provided in the appendix uses several of the MIPS macros provided the MARS MIPS Simulator.[3] The code should be able to be run as is in MARS or compiled from MARS to Machine Code.

The images of Simulink models in this document are complete and can be implemented exactly as shown. To run these models, be sure to configure the Simulink model to run in discrete time instead of the default continuous time.

Simulink for HDL Development

Simulink proved to be a very valuable tool for development and simulation of systems to be implemented in FPGA code. I would highly recommend that this tool be used in future Computer Architecture classes as it provides a nice way to go straght from block diagrams to working systems. Verilog could still be used to implement lower level structural functions. I think the two tools could be used in parallel to improve the labs in the course.

[3]http://courses.missouristate.edu/KenVollmar/MARS/

Appendix A
Euler's Method Assembly Code

```
# Euler Projectile Simulation
# Uses 7+17n Clock Cycles


# Set Initial Conditions
li $t0,0       # Y
li $t1,1000000 # V
li $t4,0       # Step Count
li $t5,0       # T
li $t6,15      # dT
li $t7,100     # Steps



MAINLOOP:

# While T < T_End
bge $t4,$t7,ENDPROGRAM

# dYdT
mul $a0,$t1,$t6
add $t0,$t0,$a0

# dVydT
abs $a0,$t1
mult $a0,$t1
mfhi $a0
mul $a0,$a0,-23
addi $a0,$a0,-981
mul $a0,$a0,$t6
add $t1,$t1,$a0

# Increment Time Counter
add $t5,$t5,$t6
addi $t4,$t4,1

j MAINLOOP

ENDPROGRAM:
```

Appendix B
Trapezoidal Euler's Method Assembly Code

```
# Modified Euler Projectile Simulation
# Uses 7+33n Clock Cycles

# Set Initial Conditions
li $t0,0       # Y
li $t1,1000000 # V
li $t4,0       # Step Count
li $t5,0       # T
li $t6,15      # dT
li $t7,100     # Steps

MAINLOOP:

# While T < T_End
bge $t4,$t7,ENDPROGRAM


# CALCULATE FORWARD VALUE

# dYdT
mul $a0,$t1,$t6
add $t2,$t0,$a0

# dVydT
abs $a0,$t1
mult $a0,$t1
mfhi $a0
mul $a0,$a0,-23
addi $a0,$a0,-981
mul $a0,$a0,$t6
add $t3,$t1,$a0

# CALCULATE BACKWARD VALUE & AVERAGE

# dYdT
mul $a0,$t3,$t6
add $t0,$t0,$a0
add $t0,$t0,$t2
sra $t0,$t0,1

# dVydT
abs $a0,$t3
mult $a0,$t3
mfhi $a0
mul $a0,$a0,-23
addi $a0,$a0,-981
mul $a0,$a0,$t6
add $t1,$t1,$a0
add $t1,$t1,$t3
sra $t1,$t1,1

# Increment Time Counter
add $t5,$t5,$t6
addi $t4,$t4,1

j MAINLOOP

ENDPROGRAM:
```