FPGA-based Reactor Simulator

1st Li Xuan Tan
Department of Mathematics
Massachusetts Institute of Technology
Cambridge, MA, USA
lixuan@mit.edu

***Abstract*—This report presents a design to simulate the controlled operation of a simple nuclear reactor (modeled on the MIT research reactor) by using the point-kinetics equations, which model the reactor core as a point in space. The user is able to control the simulated reactor by using inputs on the FPGA board to simulate control element movements and emergency shutdowns.**

***Index Terms*—discrete, simulation, field programmable gate array.**

I. The Nuclear Reactor Model

Nuclear reactors are driven by a chain reaction in which a neutron splits apart uranium atoms to produce energy, fission products and more neutrons (which then go on to split more uranium atoms, causing a chain reaction). The *multiplication factor* k is defined as the ratio between the neutron population in two successive generations of neutrons; that is, $k = 1.5$ means the next generation of neutrons will have 1.5 times as many neutrons as the current generation. This value can be affected by the type of fuel, the amount of neutrons absorbed by non-fuel materials in the reactor, the amount of neutrons leaking out of the reactor and various other factors which allow us to safely build and operate a reactor. When $k = 1$, the reactor is in a steady state as the neutron population will not be changing over time. This is known as the reactor being *critical*, with $k > 1$ and $k < 1$ being referred to as *supercritical* and *subcritical* respectively. The state of a reactor is also reflected in its *reactivity* $\rho = \frac{k-1}{k}$, which is positive when the reactor is supercritical and negative when it is subcritical. This unit-less measure of reactivity is commonly used in nuclear science to describe the neutron behaviour in a reactor.

In the point-kinetics model, which models the entire reactor core as a single point with an associated neutron flux, the following differential equation can be derived to describe the neutron population over time $n(t)$:

$$\frac{\delta n(t)}{\delta t} = \frac{\rho - \beta}{\Lambda} n(t) + \sum_{i=1}^{6} \lambda_i C_i(t) \tag{1}$$

The point-kinetics model is generally sufficient for small, simple reactor geometries such as that of the MIT nuclear reactor. We will now define the terms used in this equation.

A. Neutron Precursors and the Delayed Neutron Fraction

The value Λ is a constant, the average lifetime of a neutron before it is absorbed or escapes the reactor. This is usually taken to be 10^{-4} seconds according to experimental measurements. With such a short neutron generation lifetime, even a value of k such as 1.001 will result in a huge multiplication factor of $1.001^{10000} \approx 22000$ in a single second if all the neutrons were produced instantly from fission (that is, they are *prompt neutrons*). This would make a reactor uncontrollable, and in fact is the driving principle behind nuclear weapons.

Controlling a reactor is only physically possible because of the presence of *delayed neutrons*, which are neutrons that do not get directly emitted from fission, but instead are released from the decay of certain fission products on a timescale of seconds to minutes after the original fission. Thus, a reactor may have a k of 1.001, but only 0.999 are prompt neutrons, with the remaining 0.002 released on a delayed timescale. In this case, the reactor is termed *prompt subcritical*, but over time it is supercritical after accounting for delayed neutrons, and this allows reactor power to increase on a longer timescale on which humans can react and control the reactor. These fission products that produce delayed neutrons are termed *neutron precursors*, and typically are grouped into six different groups depending on their half-lives. $C_i(t)$ denotes the amount of precursor i at time t, λ_i is the decay constant for precursor i, and β_i is the fraction of total fission neutrons that are emitted from the precursor i. The *delayed neutron fraction* β is the sum of β_i over all six groups of precursors, which is 0.0065 for reactors fueled with uranium-235.

The other part of the point-kinetics equations are the differential equations determining the amount of neutron precursors present in the reactor:

$$\frac{\delta C_i(t)}{\delta(t)} = -\lambda_i C_i(t) + \frac{\beta_i}{\Lambda} n(t), i = 1, 2, \cdots, 6 \tag{2}$$

We can use these equations to approximate the change in neutron flux $n(t)$ and precursor population $C_i(t)$ in a single discretized timestep, allowing us to approximately model the behaviour of the reactor over time.

B. Neutron Poisons

Some fission products (most prominently xenon-135) are extremely strong neutron absorbers and will take away neutrons necessary for the nuclear reaction to proceed; these are known as *neutron poisons*. As neutron poisons are built up in the reactor core, their poisoning effect is measured by a corresponding decrease in the reactivity ρ. The xenon population can be tracked similarly to that of neutron precursors; it is

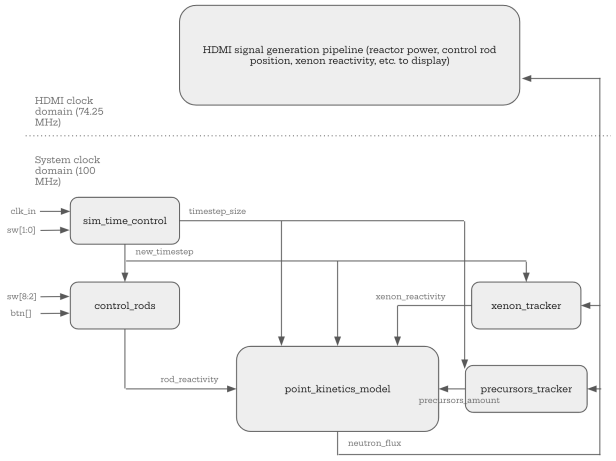
produced at a rate proportional to the neutron flux (and hence number of reactions) in the core, and is removed not by decay but by "burnup" from absorbing a neutron. In practice, xenon buildup means that operators periodically have to use control elements to add reactivity to compensate for the xenon and keep the reactor at a steady power level.

C. Control Element Reactivity Worth

Control elements usually take the form of rods or blades infused with neutron-absorbing elements like cadmium or boron. By lowering them into the reactor core, they will absorb many of the neutrons (similar to xenon) and starve the reaction of necessary neutrons. Also similar to xenon, the effect of control rod movements are measured by a corresponding change in the reactivity ρ, and these measurements (known as "control blade reactivity worths") are recalibrated periodically. Using FPGA button and switch inputs to manipulate control rods to add or remove reactivity, the user can put the reactor into a supercritical or subcritical state to change power levels. All reactors also have an emergency shutdown mechanism known as a *scram*, which often involves rapidly dropping all the control elements into the core, inserting a large negative reactivity and stopping the reaction.

II. System Design

The proposed block diagram of the reactor simulator is included below.



A. Time Control

The `sim_time_control` module controls the ratio of simulation time to real time by varying the length of a simulated timestep based on FPGA switch input from the user. At a system clock of 100 MHz, a timestep length of 10^{-8} seconds corresponds to real time simulation, but varying the timestep length δt allows us to either speed up or slow down the simulation since the system clock is constant. Varying δt also affects the accuracy of the neutron flux calculated from (1), allowing for either a faster but less accurate simulation or a slower, accurate simulation.

We can also choose to make timesteps longer than a single cycle (to enable pipelining of complex calculations), in which case the time control model will maintain a counter that rolls over at the desired timestep length in cycles, and triggers a single-cycle high `new_timestep` output to be used to control other modules. Timestep length in this case would then be the system clock period multiplied by the cycle count.

B. Point Kinetics Solver

The `point_kinetics_model` module is responsible for storing the current value of the neutron flux and updating it at each timestep. The inputs it takes are net reactor reactivity ρ (the sum of control rod reactivity and neutron poison reactivity), the timestep size δt and the neutron precursor populations $C_i(t)$. The values of Λ and λ_i are constants stored in the FPGA. Using these inputs and (1), the module computes $\delta n(t)$ and uses that to update the stored value of the neutron flux $n(t)$.

Since this module involves as many as 9 multiplications (including a multiply by δt to compute $\delta n(t)$), it may be necessary to pipeline the computation into stages with one multiplication each. This can be done using an FSM and the `new_timestep` output from the time control module, which would use a counter to initiate a new timestep every 10 cycles or so, instead of having a new timestep every cycle. The `new_timestep` output then triggers the solver to start a computation with the currently-stored value of $n(t)$ and perform one of the multiplication steps each cycle.

C. Neutron Precursors

The `precursors_tracker` module stores and updates the population of each of the six neutron precursor groups, and passes it to `point_kinetics_model` for use. It takes the neutron flux $n(t)$ and the timestep size δt as inputs to update the populations $C_i(t)$ based on (2). A similar model is used for tracking the xenon population, and the negative reactivity worth of xenon can be assumed to scale linearly based on the amount of xenon. Since this module also requires several multiplications, it can be pipelined similarly to the point-kinetics solver.

D. User Control

The `control_rods` module stores the position of several control rods, which can be updated by the user by using FPGA switches and buttons to select, raise or lower a particular rod. We can model the control rods as being cut into 2^n segments for some n, and the position will be stored as an n-bit integer representing the number of segments that are currently inserted. At each new timestep when a particular rod is selected and moved, its position will be changed by some (potentially adjustable) speed increment that determines how fast the rod moves. Reactivity worth of a partially-inserted rod can also be assumed to scale linearly based on the fraction of the rod that is inserted into the core, and the net reactivity of the control rods is then passed to `point_kinetics_model` to compute the neutron flux. In addition, a scram can be implemented by using a FSM to place the system into a "scrammed" state, in which the control rods will drop rapidly into the core and not be able to be removed.

At the MIT reactor, a scram is required to take under 1 second by regulation, and is typically below 700 milliseconds, so a suitable "scram speed" (in segments per timestep) of the rods can be calculated from the desired drop time, the number of control rod segments and timestep length.

This module was implemented as a parametrizable module to allow for control rods to have different reactivity worths, ranges of motion and movement speed. The scram was simply implemented as a "set all rods to full insertion" loop. To simulate the response time from an actual reactor (and also to make it easier to handle the movement speed time-stepping), the `control_rod` module is passed an input `new_second` produced by the `sim_time_control` module. This triggers the control rods to move approximately once per second, and in discrete increments instead of continuous.

III. Results

To avoid having to deal with floating-point multiplication, the code used a lot of bitshifts/adds to approximate the product. This turned out to be somewhat inaccurate especially for the smaller fractions like λ_i, and resulted in the simulation not exactly mirroring expected behaviour - I had to tweak the reactivity calculations a little to have it match my experience better. Although quantitatively the results aren't great, the simulation does display the expected qualitative properties, which makes me think that the inaccuracy is likely to just be down to bad approximations. For example (as noted in my demo video), the simulation replicates the "prompt drop" that occurs upon a sudden insertion or removal of reactivity, like a scram. This is a scenario in which the power spikes or dips sharply, then the graph shows an "elbow" as the reactor returns to the exponential growth or decay that's closer to expectation. The prompt drop is predicted by reactor theory and can be derived from the point kinetics equations, but this scenario doesn't explicitly appear in my code so it does seem that the simulation is working as it should, since it was able to reproduce this.

In terms of performance, I ran into some issues meeting timing even with the slightly more lenient 74.25 MHz HDMI clock that I had originally planned to use. I put pipelines in a number of places that didn't seem to help much, and in the end it seems that the problem was at least partially caused by having too many large additions in one statement? After splitting those up by making an FSM for the `point_kinetics_solver` module, the code was able to meet timing so it seems like that's it, which is interesting because I had been significantly more worried about my multiplies. Otherwise, the computation is fairly simple and not very resource-intensive - only 4 DSP blocks were used even though I made some slightly suboptimal choices (the main reason I avoided using them much was because there were a lot of floating-point multiplies that I just approximated with bitshifts). Considering there's so much idle time in between timesteps (thousands of cycles apart, whereas even the most complicated pipelined computation in my design was 5 cycles), there's a lot of room to increase the accuracy of the simulation by shortening the timesteps, but that runs into the problem that we need to multiply by ever smaller fractional values to get the change in neutron flux. Approximating those small fractional values can get inaccurate or need to use up more bitshifts/adds, which might be pushing timing limits? Definitely room for a few orders of magnitude improvement though.

IV. Code