Methods for RNS to Binary Conversion: Classic Logic and Neural Networks

Author: Sam McCauley
Advising Professor: Dr. Ahmad Salehi
University of Kentucky College of Engineering
Spring 2020

The Residue Number System

The Residue Number System (RNS) is a novel method used to represent integer numbers: an alternative to binary. RNS works by selecting a list of *relatively* prime numbers such as [8,7,5] or [11,7,5,3]. To convert an integer to RNS, you take the integer and apply the modulus operation with each of the relatively prime numbers acting as the moduli. For example, to convert the integer 78 using the moduli [8,7,5]:

```
78 % 8 = 6 = 0b110
78 % 7 = 1 = 0b001
78 % 5 = 3 = 0b011
```

The final RNS representation is the modulus bits concatenated together: `0b110001011`.

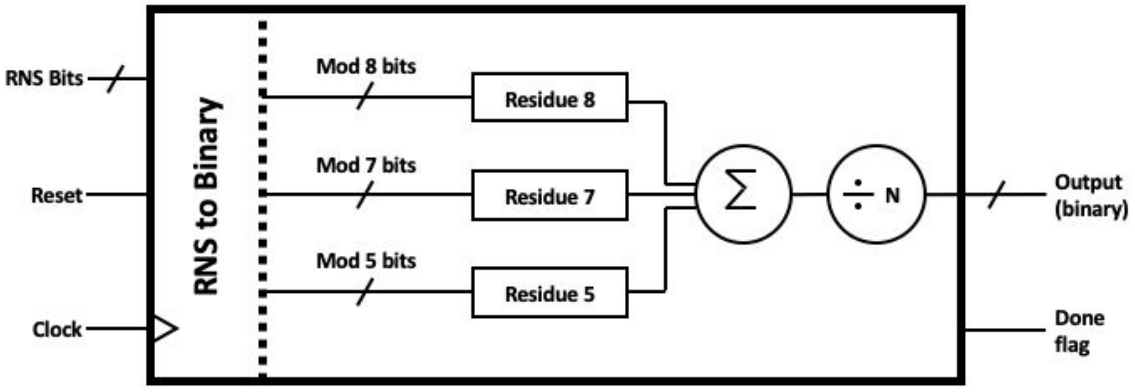
RNS is most useful for performing arithmetic on very large numbers. When adding, subtracting, or multiplying two RNS numbers, you just add/subtract/multiply each residue together to form the result. For example, if you are performing addition of two numbers using moduli [8,7,5], you would add the two (mod 8) residues, two (mod 7) residues, and the two (mod 5) residues all independently. Ultimately, this allows for a lot of parallelization within the RNS arithmetic, something that classic binary integer representation cannot do.

Converting RNS to Binary Classically

Going from binary to RNS is pretty simple, as you saw in the example above. Going from RNS to binary is a little more complicated. The typical approach is to use the Chinese Remainder Theorem, an algorithm that finds the unique solution that satisfies all of the moduli conditions. It would be quite verbose for me to actually explain the Chinese Remainder Theorem in this paper, so if you're interested you can watch this great video on it. The example used in the video will be the same example I will use throughout this paper (in fact, I already have): converting moduli [8,7,5] and residues [6,1,3] to binary integer 78.

Chinese Remainder Theorem

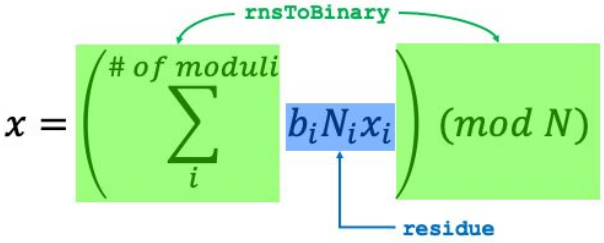
$$x = \left(\sum_{i}^{\#\ of\ moduli} b_i N_i x_i \right) (mod\ N)$$



Above are two representations of the Chinese Remainder Theorem. The first is the actual mathematical formula, where b_i is the residue, N is the product of all moduli, and x_i is the modular inverse of the residue. The second image is the diagram for a circuit implementation of the conversion using verilog, which can be found in `rnsToBinary.v` and can be run in this online verilog simulator.

The verilog code has one main module called `rnsToBinary` which takes in the RNS bits as well as a reset wire, for when the RNS bits change, and lastly a clock wire. This module instantiates three other modules to calculate the individual products within the sum function: these `residue` modules take the b_i residue, N_i computed as (N / modulus), calculate the modular inverse x_i, and finally multiply them all together. Once each individual residue module finishes its partial computation, the `rnsToBinary` module completes the RNS to binary conversion by summing up the partial results from the each modulus' `residue` module and then dividing that sum by N.

In other words, in relation to the mathematical formula, each `residue` module carries out one of the inner products $b_i N_i x_i$ and `rnsToBinary` completes the calculation by summing and dividing it out.



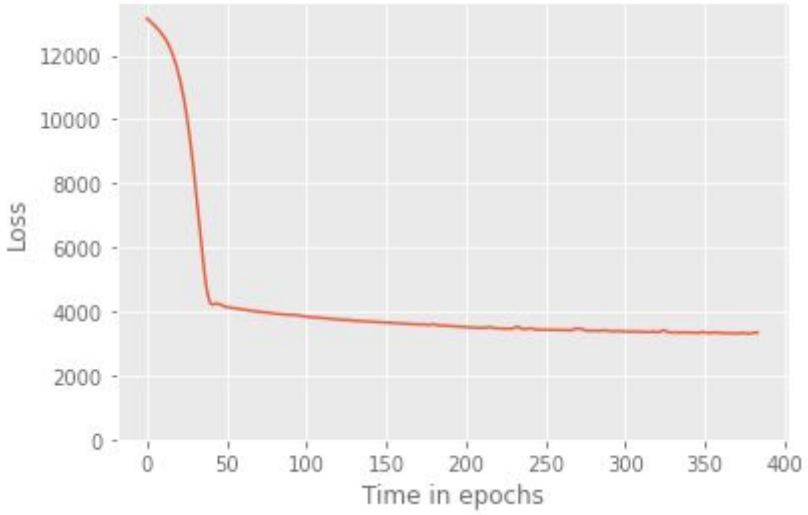
The verilog synthesis for this conversion is efficient, flexible, and modular. The RNS moduli can be changed with minor code edits.

Converting RNS to Binary Using Neural Networks

Beyond the classical logic implementation of the RNS to binary conversion, I also tried using neural networks to handle the conversion. Neural networks are good at handling nonlinear data, which RNS most definitely is. The hope was that I could find a neural network architecture with relatively low width (3-4 hidden layers) so that a conversion would only require a couple multiplication operations in series to find the output. If the nodes were to be implemented in silicon, each layer could be computed in parallel. Though it would require a lot of area to implement, the idea was that a neural network implementation could be potentially faster than the classic logic implementation. Three architectures were attempted. All code and results can be viewed and experimented with in the `rns_to_binary.ipynb` file.

Network #1: Scalar Input Regression

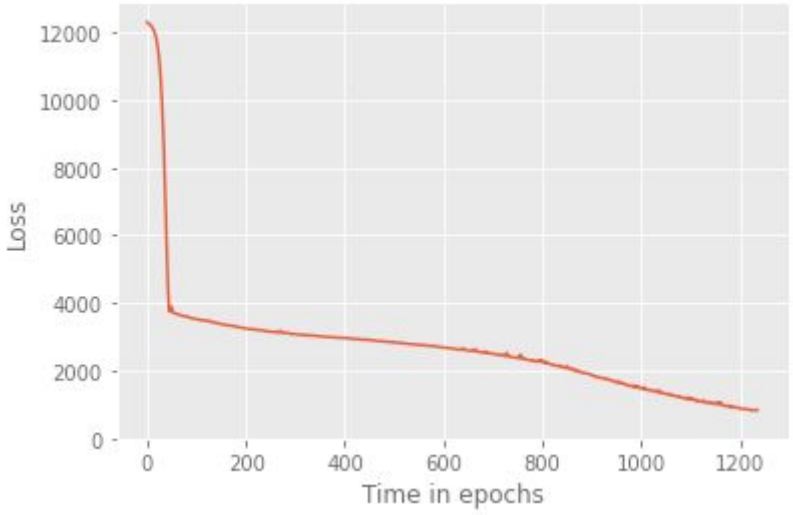
The first neural network is a regression with scalar inputs. For example, if our RNS moduli are [8,7,5] and the residues are [6,1,3] respectively, the inputs to the network are [6,1,3] and the output should be 78. The model is trained on 80% of every possible conversion in the RNS system. Unfortunately, this model doesn't even come close to converging, as you can see below. Currently, the model is configured for hidden layers of (32,32,32,32). Other configurations were tried but don't seem to aid in convergence.



Network #2: Bitwise Input Regression

The second neural network is a regression with binary inputs. For example, if our RNS moduli are [8,7,5] and the RNS representation is [6,1,3], then the actual input to the network will be the residues in their binary form: [1,1,0,0,0,1,0,1,1]. The output should still be 78.

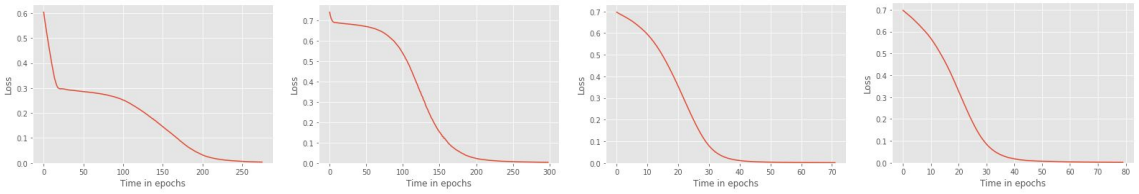
Like the previous, this model is trained on 80% of every possible number in the RNS system with hidden layers of (32,32,32,32). While this model usually comes *close* (much closer than the previous network) to converging, it still never does. It's predictions are still wildly off.



Network #3: Bitwise Input Classification

For this last neural network we use the same bitwise input structure, but instead of trying to do a regression output I created a new neural network classifier for *each binary digit of the output*. Therefore, if the max possible output of the RNS in decimal form has 8 binary digits, then we will need 8 models: one for each of the binary digits:

model for bit n-1 *model for bit n-2* ... *model for bit 1* *model for bit 0*



The models were trained on 100% of the training set, thus the neural network was used in more of a memory based architecture. With hidden layers of (32,32,32,32), each model was able to predict the output bits with 100% accuracy. On slightly larger systems with moduli [11,7,5,3], the model(s) were still largely accurate and are generally able to convert most (~98%) RNS numbers successfully. However, this does not scale well at all. Training is very cumbersome for very large RNS moduli sets and accuracy is either greatly diminished (~60%) or the models can't converge at all with sets like [17,13,11,7,5,3]. While this method had some success, it isn't really viable. In fact, for the small sets it works with it might even be more efficient to just store a lookup table for the conversions.

Summary

Ultimately, the Residue Number System is a novel number system with some pretty unique advantages. The biggest disadvantage, however, is the complexity required to convert it back to the standard binary representation. The easiest way to do this conversion is with some circuit logic that requires a series of summing, multiplying, dividing, and modular inversing. While a neural network approach to this same problem was worth exploring, it proved to be unsuccessful and could not match the relatively minimal computational area and scalability of the circuit implementation of the conversion module.