Connor Kalina

Java Abacus Computer

SCOPE:

The following documentation outlines an implementation of an “abacus” computer in java. An abacus computer has only two functions, increment, and decrement that it can perform on it’s registers’ values. Through this project, we prove that using just these two functions, we can build much more complicated and useful methods and data structures. Implemented below are the following functions: Assignment, test for equality, addition, multiplication, less than or equal, and division. Using these functions I was able to build an associative memory data structure, and the methods required to use it.

FUNCTIONS:

- User can input code with described syntax into an input file
- Instructions are created from text file and sent to abacus computer
- Abacus computer computes instructions and displays a trace of its computation.
- Final associative memory configuration is displayed

PERFORMANCE:

Due to the nature of an associative memory data structure using exponential functions on large integers, and the abacus computer’s inherent ability to manipulate register values by only one at a time, the performance of this structure is lacking. It is recommended when using the associative memory data structure, that only small register values and arguments are used. For example, register 2 value 5, not register 5 value 9.

However, if the associative memory data structure is not used, exponentiation and manipulation of extremely large numbers is less likely to be needed, and larger numbers can be used with acceptable performance. For example, the program can generate the first 200 prime values in just a few seconds. The compute() function in the AbacusComputer class has example math functions that can easily be modified to demonstrate the effectiveness of its functions.

USAGE DIRECTIONS:

- To use the program simply compile the code with command “javac Wrapper.java“
- Make sure the input file is in the same directory as the class files
- Then, run it with the command “java Wrapper input.txt”
- The process will run the instructions as specified in the input file

- To not use the associative memory data structure, edit the compute() method in the AbacusComputer class with math tests as desired.
- Example input files are provided in the ExampleInput folder, use the file's name as the argument. Ex: java Wrapper ifRemainder.txt

Structure:

- The Wrapper class reads the argument file and creates instructions for each line. Then it creates an abacus computer and enqueues the created instructions. It then calls the compute function.
- The AbacusComputer class compute function first runs some test arithmetic problems and displays the results. Then it dequeues it's instructions and executes them. A switch statement on the instruction type determines how to compute each instruction.
- The Instruction class is constructed from a string(a line in the input file). It sets its type and parses all the needed registers from the given string into an ArrayList.
- The AssociativeMemory class contains only one value, and its getters and setters.

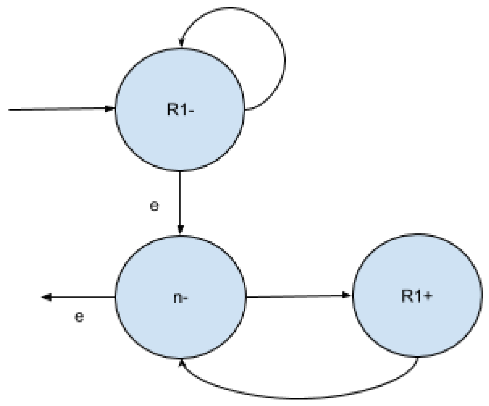
Extra Functions:

- checkIfPrime: divides parameter by every number until itself, and checks remainder, if there is no remainder, it is not prime.
- nextPrime: runs checkIfPrime on an incremented parameter register until checkIfPrime returns 1. Returns the incremented value.
- nthPrime: initialize a temporary variable and sets it to nextPrime(0). The parameter variable is decremented until it is 0 and the temporary variable is changed to the nextPrime. When the parameter variable is 0, the temporary variable is the n'th prime number.
- set(register, value): resets the given register. Then finds the register'th prime number. (register 3 = prime 5). Raises the prime number to the value given to store. Then multiplies this value into the memory.
- retrieve(register): Finds the register'th prime number. Divides the memory value by the prime number until there is a remainder. The number of times divided-1 is the value of that register.
- reset(register): Finds the register'th prime number. Divides the memory value by the prime number until there is a remainder. Set the value of memory to the last division with no remainder.

Connor Kalina

Assignment function:
Syntax : R1=n
Puts n into R1, after R1 is reset

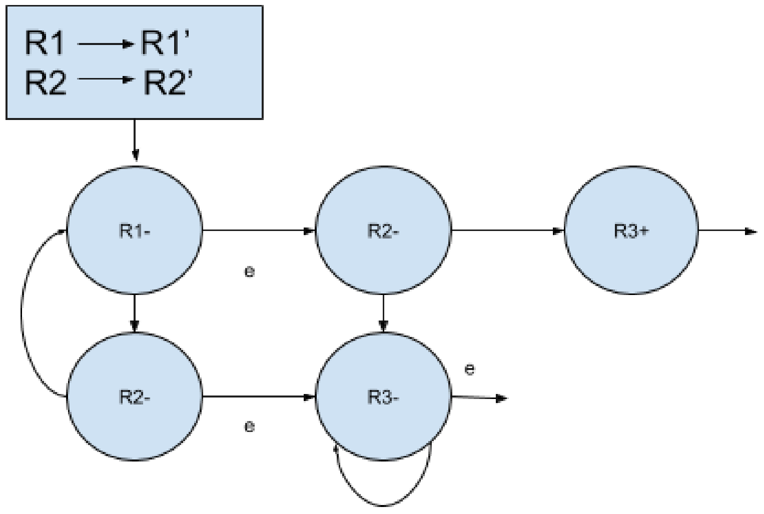
e: take path when register value is empty (0)



The assignment function resets the given register, then it decrements the given value and increments the given register until the given value is 0.

Connor Kalina

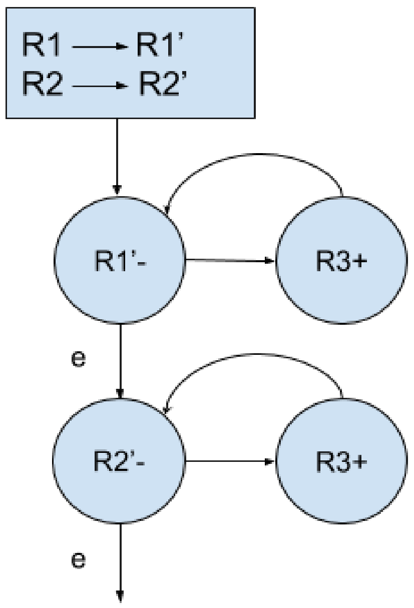
Test Equality
Syntax : R1==R2,R3
Tests if two register values are equal, sets third register value to 1 if they are, and 0 if they aren't



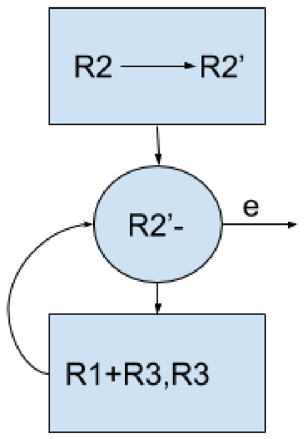
The test equality function assigns temporary registers and decrements them until one of them is empty.   If both are empty at the same time, R3 is incremented, else R3 is reset to 0.

Connor Kalina

Addition
Syntax : R1+R2,R3
Adds two register values together, stores the result in a third register.



Multiplication
Syntax : R1*R2,R3
Adds R1 to R3, R2 times

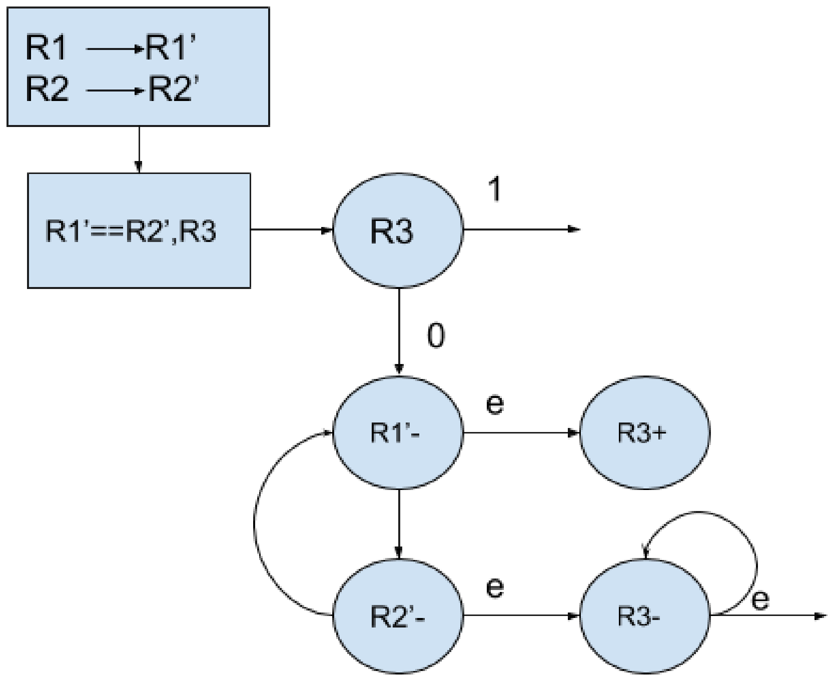


The addition algorithm puts R1 and R2 into temporary registers. Then it decrements R1' and adds one to R3 until R1' is empty. Then it decrements R2' and adds one to R3 until R2' is empty.

The multiplication algorithm uses the previously designed addition algorithm. It puts R2 into a temporary register, then decrements R2' before executing R1+R3,R3. Then it loops back to the previous step. It does this until R2' is empty.

Connor Kalina

Less than or Equal
Syntax : R1<=R2,R3
Sets R3 to 1 if R1 is less than or equal to R2, otherwise 0



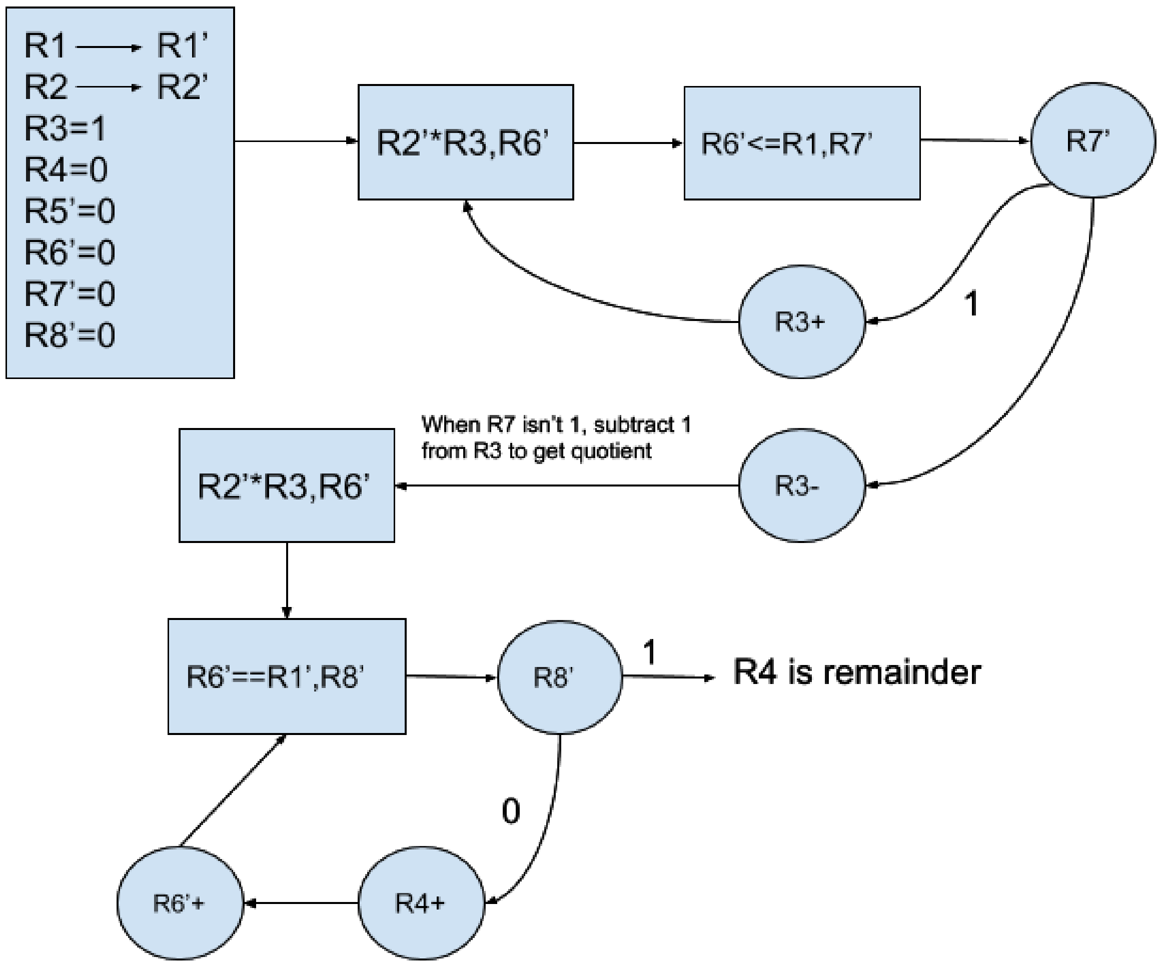
The Less than or Equal function assigns temporary registers and then checks if the given R1 and R2 are equal and sets R3 to one if they are. If they aren't equal, R1 and R2 are decremented until either is empty. If R1' becomes empty before R2, R3 is incremented
(R1 < R2). Otherwise R3 is reset to 0.

Connor Kalina

Division

Syntax : R1/R2,R3,R4

Divides R1 by R2, and sets the quotient in R3 and remainder in R4



The division algorithm first assigns temporary registers and sets them to 0, except for R3 which is set to 1. It then multiplies R2' by R3 and stores the result in R6'. If R6 is less than or equal to R1, R3 is incremented we go back to the multiplication step. This is done until R7 is 0, meaning R6 is greater than R1. R3 is then decremented, and is now equal to the quotient. Then R2' is multiplied with R3 and put in R6. R6 is tested for being equal to R1, if they are equal, R4 is the remainder. Else, R4 is incremented, R6 is incremented and the equality step is performed again. When R6' and R1' are equal, R4 is the remainder.