**Introduction.**

The SIMMAC is a simulated virtual machine that will run programs in its own programming dialect. This design will use Java to create and execute the SIMMAC. The memory of this virtual machine will have a 512 piece expression, registers, an arithmetic logical unit, operands, and opcodes. The operand will perform a function, while the opcode will hold values or addresses. The purpose of this exercise is to handle multiple processes. Processes will be fed to the SIMMAC via its programming dialect in a text file. These processes are validated and executed. If everything checks out the contents of all the memory and registers are printed to screen. A queue is established for process management in a round robin architecture. While a process is running its clock is counting. If the clock exceeds the time quantum value it will end the process and send it to the back of the queue and load the next process in the queue. Processes are stored in an ArrayList(). Using the .remove(0) method in the array list that holds your process will cause the process at index 1 to now become process at index 0. This effectively moves your next process to be the current process. If the process that was removed is not terminated yet it will be added back the process queue array list with the .add(process) method. Effectively adding the process to the end of the queue. This completes the round robin architecture of this system.

**Design.**

**Programming language:** Java v1.8.0\_191

**Classes: Main, Instructions, SIMMAC, OperatingSystem, Process.**

|  |  |
| --- | --- |
| **Main.Java** | |
| |  | | --- | | **Methods** | |  | | **Run(args);**  Only method loaded in main, starts **SIMMAC, CPU, and Operating System**. Executes **getFile()** method. Loads programs to operating system, and runs operating system. This is the entry point to our program with many levels of abstraction of moving parts and input validation taking place. | | **getFile(String file);**  Gets passed in path of file to be loaded. It then uses abstraction by calling the **Instruction.parseOperand()** method to validate that the input text files have correct instructions that are readable to the operating system. It then returns these instructions as an array of integers back to be referenced in the **run()** method that are then loaded and ran in the operating system via  **OperatingSystem.loadProgram()** and **OperatingSystem.run().** | | |  | | --- | | **Variables** | | **Scanner scanner**  Scanners get file path and decision making if more than 1 file will be loaded. | | **Int quantum**  Integer value of how much time the virtual CPU should spend on a process. Gets passed into initialization of operating system. | | **SIMMAC cpu**  Creates instance of virtualized CPU. Gets passed into initialization of operating system. | | **OpeatingSystem OperatingSystem**  Creates a new instance of an operating system getting passed a virtual time quantum and CPU. | | **Boolean loadFileDone**  Initializes as false and while loading the file is false (Not done.) It performs logic to load files. When this is True, loading files will end. | | **ArrayList<String> filenames**  Passes files names as an array list of strings to the getFile() method. This is the list of file paths. | | **String choice**  This string allows you to choose Y or N to load 1 or more files. | | **Int[] program**  This integer Array is used to store the numeric values of the programs to be ran. | | **ArrayList<Integer> instructions**  This ArrayList of integers that hold instructions after validation from the Instruction class. | | **String line**  Represents current starting line to read instructions on. Splits them up line by line to be loaded by operating system if valid. | | **Int nline**  Stands for next line and holds value of next line to be read in program instructions. Starts at 1 because 0 in the array would be the initial value and not the next line. | | **Integer inst**  This is where the loaded files contents are validated for us and returned in machine readable integer value or returns null | | **Int[] instr**  This Integer array is fed the inst Integers we know are valid and sends them to the instructions to be loaded to the program. | |

|  |  |
| --- | --- |
| **Instruction.Java** | |
| |  | | --- | | **Methods** | |  | | **getOpcode(String word)**  This method get passed in the three character string representation of the opCode and uses a case switching validation to return the integer value instructions of the opcode to be read and executed by the virtual machine. | | **parseOperand(int opcode,String operand,String filename,int nline)**  This method validates that the operand is correct and if true returns the Integer value of data to read and stored by the virtual machine. | | **parseInstruction(String line,String filename,int nline)**  This method verifies the instruction in the files are valid.  It has validation to check to see if the syntax is correct when reading the input text fiels. | | |  | | --- | | **Variables** | | **All opCodes public static final int (Opcode name) = (opcode value.)**  DW = 0x0000  ADD = 0x0001  SUB = 0x0002  LDA = 0x0003  LDI = 0x0004  STR = 0x0005  BRH = 0x0006  CBR = 0x0007  HLT = 0x0008  These are the true integer values of each opcode to be read by the operating system. | | **String opcode**  Takes in string value of opcode and turns to uppercase to be passed to switch case. | | **Int val**  Used to set numerical value operand and check validity of each operand. | | **String[] parts**  Splits program into a string array to be read and validated and verified is correct instructions for operating system. | | **Int opc**  Set to the returned integer value of validated opcode from getOpcode() method. | | **Int op**  Set to the returned integer value of validated operand from parseOperand() method. | |

|  |  |
| --- | --- |
| **Process.Java** | |
| |  | | --- | | **Methods** | | **Process(int address, int size, int processID)**  This method gets passed the processID, and sets the Accumlator and Primary Storage Instruction Address Register counters to 0. Memory base is set to the address that is passed in and the memory limit is set to that address plus the size that is also passed in. | | |  | | --- | | **Variables** | | **Int processID**  Creates an integer value for a process ID that will later be used to establish a position in the ready queue. | | **Int ACC**  The Accumulator counter is set to zero to be later used as processes are running. | | **Int PSIAR**  The Primary Storage Instruction Address Register counter is set to zero to be later used as processes are running. | | **Int memoryBase**  Integer value of the address of program. | | **Int memoryLimit**  Address of program plus the size of the program will tell you last available memory address. | |

­­

|  |  |
| --- | --- |
| **SIMMAC.Java** | |
| |  | | --- | | **Methods** | | **SIMMAC()**  Constructor method that creates memory of 512, sets the CSIAR, PSIAR, and ACC to 0 but sets the memory limit to the 512 mentioned above. | | **read()**  Validates there is something to read from SAR and Memory base and if so then that it does not exceed the memory limit constraint of 512. If so, SDR is equal to the integer stored in the memory array where the index equals SAR + memory base. | | **write()**  Validates there is something to read from SAR and Memory base and if so then that it does not exceed the memory limit constraint of 512. If so, the integer stored in the memory array where the index equals SAR + memory base equals the integer in the SDR.(Opposite of Read().) | | **Operational methods provided in assignment.**  fetch()  add()  sub()  load()  store()  branch()  conditionalBranch()  loadImmediate()  Please see code notes for individual method. | | **Dump()**  This method dumps the contents of everything in Memory looping through all 512 units and printing its contents of the 7 registers and the accumulator. | | **executeTheInstructions()**  This method executes the fetch() method to get the CSIAR and run a case switch depending on returned micro-instruction register and runs that corresponding MIRs method in that chosen case. | | |  | | --- | | **Variables** | | **MEM\_SIZE**  **Sets standards to 512** | | **All Registers integers.**  int ACC  int PSIAR  int SAR  int SDR  int TMPR  int CSIAR  int MIR | |

|  |  |
| --- | --- |
| **OperatingSystem.Java** | |
| |  | | --- | | **Methods** | | **OperatingSystem(SIMMAC cpu, int quantum)**  Class constructor that creates virtual operating system with a SIMMAC cpu and time quantum value from its specific instances. The last loaded address, ready queue, current process, are initialized. Also a clock is created to contest against the passed in time quantum value. This is what delegates whether too much time has been allocated for the process and will switch processes if the clock exceeds the time quantum value. | | **printProcesses()**  loops through the ready que and prints process ID’s. | | **Switch\_process()**  If the current process is not null it saves the current register states and then adds the current process to the end of the ready queue to be executed later. If it is null it gets the next process by removing the zero index in the queue array making the next process inline the new index of 0, and setting that new process as our current process. It then loads the state of the registers and gets the memory base and limits for this new process. The clock is also restarted so this process can compete against the quantum time value. | | **Run()**  **Sets current process to null and runs switch\_process() method to get next process in ready queue array. While the program is not terminated the clock is counting. If the clock exceeds the quantum value and is not terminated it switches to the next process and adds current process to end of queue round robin style.** | | **loadProgram(int[] program)**  **Loads program from last load address. It then validates that it won’t exceed cpu memory size. If everything is good it will create a new process to be added to ready queue.** | | |  | | --- | | **Variables** | | **ArrayList<Process> rdyQue**  Creates an array of stored process in the queue. | | **Process current\_process**  This Process is the current process running. | | **Int lastLoadAddress**  When loading the process the start address will become the last start address. | | **Int clock**  While a process is running and not terminated the clock is counting, if it exceeds quantum time it switches processes. | |