**Library Documentation**

**<Interpreter> *Interpreter.h***

The Interpreter takes the input of the program, runs error checks to make sure it is valid input then puts it into memory.

Int numbers - input is stored once it is saved as an integer.

string word - this is where the argument is stored once it is taken in as a string

Int stopCase -What that the Interpreter looks for to halt taking the programming

Int incrementor - this counts how many words get put into memory

Int wordLength – The value of the maximum word length to accept

allDigits - this is true after the first character is off and the rest of the word are digits

vmMemory\* memory = vmMemory::getInstance();- a pointer to access the memory class

maxMemory - int, the maximum amount of words that can be put into memory.

**Precondition:** Interpreter is called to start taking input commands.

**Postcondition:** All the program input has been checked for errors and input into memory.

**Functions:**

bool interpreter::parseCommand(std::string command)

returns true if the word was valid and entered into the main memory.

**<Memory> *Memory.h***

It reads and writes words from and to memory. The words manipulated are represented by integers of four digits each.

Int memSize – an integer that has the value of the size of memory, used to initialize main memory array.

Int mainMem – an array that is the main memory

**Precondition:** initialized at the start of the program for main memory. Called when reading ad writing to main memory

**Postcondition:** an array of memory has been initialized

**Functions:**

< *int write(int address, int\* data);* >

*write:* It writes a word to a designated location in memory

Returns zero

< *int read(int address, int\* data);* >

read: It reads a word from a designated location in memory.

Returns zero

< *int write\_multiple(int address, int\* data, int length);* >

write\_multiple: It writes memory addresses with integers at data address + offset

< *void dump(void);* >

dump: It prints out the entire memory array

No return

**<Executor> *Executor.h***

It uses the read and write Memory class capabilities to access the words in memory, and to extract the opcode and operand.

***Precondition:*** classes call this class when then need to interact with memory.

***Postcondition:*** Gets opcode and operand from memory calls invoker to decode instructions while not equal to 43.

***Functions:***

< *string GetOpcode();* >

GetOpcode: Reads a word from memory and extracts the instruction opcode within the word.

Returns the opcode as a string.

< *int GetOperand();* >

GetOperand: Reads a word from memory and extracts the instruction operand within the word.

Returns the operand as an integer value.

< *void CallInvoker();* >

CallInvoker: It Calls the *execute* member function in the Invoker class and uses the instruction opcode and operand to be executed by the virtual machine. After the execution of a program the dump member function in the Memory class is called to display the entire memory.

No return

***<Invoker> Invoker.h***

It executes the instructions fetched from memory.

***Precondition:***takes the opcode and operand as arguments.

***Postcondition:***the opcode and operand are sent for the execution of the code.

***Functions:***

< *int executeCommand(string code, int argument);* >

exectueCommand: takes the opcode and the operand for the execution of the respective instruction defined in the instruction set.

Returns zero if instruction is a valid instruction

Returns negative if one of the instructions is not valid

**<OpBranch> *OpBranch.h***

*Used to branch the PC counter to a different part in the program.*

***Precondition:*** The PC counter will be at a location. Takes argument for where you want the PC counter to go to.

***Postcondition:*** *The PC Counter will branch to a specified location.*

***Functions:***

*void OpBranch::execute(int argument).*

argument is specified location for the PC counter to go to.

*No return.*

**<OpBranchNeg> *OpBranchNeg.h***

*Branches to a new location of the program if the value of the accumulator has a stored value that is negative.*

***Precondition****:* When the accumulator’s value is negative.

***Postcondition:*** *T*he PC counter will branch to a specific location in the argument.

***Functions:***

*void OpBranchNeg::execute(int argument)*

*no return.*

**<OpBranchZero> *OpBranchZero.h.***

*Branches to a new location of the program if the value of the accumulator has a stored value that is zero.*

***Precondition:*** When the accumulators value is equal to zero.

***Postcondition:***The PC counter will branch to a specific location in the argument.

***Functions:***

*void OpBranchZero::execute(int argument)*

*no return.*

***<OpAdd> OpAdd.h***

For basic math of words in program. This function adds a word form a specified location in memory to the value in the accumulator, value is stored in the accumulator.

***Precondition:*** A value in the accumulator. Takes an integer as an argument which Is the location of the word to be added to accumulator.

***Postcondition:***Word from a location in memory added to the value in the accumulator and left in the accumulator.

***Functions:***

*void OpAdd::execute(int argument)*

Adds the specific location memory value to the accumulator value. Value left in accumulator. No return.

**<OpDivide> *OpDivide.h***

For basic math of words in program. This function divides a word form a specified location in memory to the value in the accumulator, value is stored in the accumulator.

***Precondition:*** A value in the accumulator. Takes an integer as an argument which Is the location of the word to be divided to accumulator.

***Postcondition:***The value stored in the accumulator will be divided by a word from a location in memory and left in the accumulator.

***Functions:***

void OpDivide::execute(int argument)

Divides the specific location memory value to the accumulator value. Value left in accumulator. No return.

***<OpMultiply> OpMultiply.h***

For basic math of words in program. This function multiplies a word form a specified location in memory to the value in the accumulator, value is stored in the accumulator.

***Precondition:*** A value stored in the accumulator. Takes an integer as an argument which Is the location of the word to be multiplied to accumulator.

***Postcondition:***The value stored in the accumulator will be multiplied by a word from a location in memory and left in the accumulator.

***Functions:***

void OpMultiply::execute(int argument)

Multiplies the specific location memory value to the accumulator value. Value left in accumulator. No return.

***<OpLoad> OpLoad.h***

This function loads a word from a specified location in memory and loads it into the accumulator.

***Precondition:*** *A word in a location of memory that needs to be put into the accumulator. Takes an integer as an argument which is the location in memory of the word to be uploaded into the accumulator.*

***Postcondition:*** *The word in the specified location will be in the accumulator.*

***Functions:***

*void OpLoad::execute(int argument)*

*loads the value in the specified location(the value of the argument) into the accumulator. no return.*

***<OpStore> OpStore.h***

This function is used to store the value of the accumulator into a specified location in memory.

***Precondition****:* A word in the accumulator that needs to be stored in memory. Takes an integer as an argument which is the location in memory the word in the accumulator will be stored in.

***Postcondition:*** *Stores the word in the accumulator into a location of memory.*

***Functions:***

*void OpStore::execute(int argument)*

The value of the accumulator will be but into the location of the argument. No return.

**<OpSubtract> *OpSubtract.h***

For basic math of words in program. This function subtracts a word form a specified location in memory to the value in the accumulator, value is stored in the accumulator.

***Precondition:*** A value in the accumulator Takes an integer as an argument which Is the location of the word to be subtracted to accumulator.

***Postcondition:*** *Subtract a word form a location in memory from the value in the accumulator and left in the accumulator.*

***Functions:***

*void OpSubtract::execute(int argument)*

Subtracts the specific location memory value to the accumulator value. Value left in accumulator. No return.

***<OpRead> OpRead.h***

This function is called to read a word from use input and stored into a location in memory.

***Precondition:*** To store words into memory. This class takes a integer and a string as an argument. Integer is the location in memory the string will be but into.

***Postcondition:***The word will be but in memory at the specified location.

***Functions:***

*this->memory->write(arguement, &inputWord);*

Puts the input word into the location of memory at the specified location(argument).

Returns “”.

***<OpWrite> OpWrite.h***

*This is called when writing a word from a location in memory to the screen.*

***Precondition:*** *A word in a location of memory that needs to be printed to the screen. Takes an integer as an argument which is the location of the word to be printed.*

***Postcondition:*** *Word from a specific location in memory printed screen.*

***Functions:***

*void OpWrite::execute(int argument)*

*returns string, with the value of the location of in memory of the integer argument.*

***< BranchOpCode> BranchOpCode.h***

This is an abstract class, it defines the opcode classes to be used by the invoker.

***Precondition****: clasess are need to be defined for the invoker.*

***Postcondition:*** *all the opcode classes are defined for the invoker.*

***Functions:***

vmMemory\* memory – a pointer to memory to read from memory

*virtual std::string getCode()*

***<BranchInvoker> BranchInvoker.h***

It executes the instructions fetched from memory, if not found in branch invoker, it calls LastInvoker.

***Precondition:***takes the opcode and operand as arguments.

***Postcondition:***the opcode and operand are sent for the execution of the code. If not found then it will send opcode and operand to LastInvoker.

***Functions:***

*int execute (string code, int argument);*

exectueCommand: takes the opcode and the operand for the execution of the respective instruction defined in the instruction set. Returns zero if instruction is a valid instruction, Returns negative if one of the instructions is not valid

int BranchInvoker::executeCommand(std::string code, int argument)

returns 0 if executed, if not it calls LastInvoker to find command.

***<fecade> fecade.h***

Used for the graphical user interface to interact with the rest of the program.

***Precondition:*** This class is called by the GUI

***Postcondition:*** The GUI talks and gets what it needs from the

***Functions:***

*std::string Facade::opWrite()*

*-for the GUI to write to the screen*

*void Facade::opRead(std::string input)*

*-for the GUI to read to memory*

*int Facade::getCurrentOperand()*

*-returns the current operand*

*std::string Facade::getCurrentOpCode()*

*-returns the current opcode*

*void Facade::incrementPC(int increment = 1)*

*-*stores local variable for PC counter than increments the PC counter by 1.

*bool Facade::addCommand(std::string command)*

*-returns true if command was valid and entered into memory.*

*void Facade::execute()*

*-calls the invoker*

*std::string Facade::getMemory()*

*-returns memory dump of the main memory*

***<Form1> Form1.h***

Used for the Graphical User Interface.

***Precondition:*** The form is compiled and blank ready for user commands. Help button will display the read me file for assistance.

***Postcondition:***  The memory content display is showing the current content of the memory array and the commands are shown on the left side.

***Functions:***

***Button1:*** Help button, displays read me file for user assistance

***TextField:*** Used for taking command from the user to put into memory.

***Memory\_content:*** Displays the content in main memory.

***<LastInvoker> LastInvoker.h***

It executes the instructions fetched from memory, gets called if BranchInvoker didn’t find code.

***Precondition:***takes the opcode and operand as arguments.

***Postcondition:***the opcode and operand are sent for the execution of the code.

***Functions:***

int LastInvoker::executeCommand(std::string code, int argument)

returns 0 if executed, returns -1 if command was not executed.

***<MathInvoker> MathInvoker.h***

It executes the instructions fetched from memory, if not found in math invoker, it calls BranchIvoker .

***Precondition:***takes the opcode and operand as arguments.

***Postcondition:***the opcode and operand are sent for the execution of the code. If not found then it will send opcode and operand to BranchInvoker.

***Functions:***

*int execute (string code, int argument);*

exectueCommand: takes the opcode and the operand for the execution of the respective instruction defined in the instruction set. Returns zero if instruction is a valid instruction, Returns negative if one of the instructions is not valid

int MathInvoker::executeCommand(std::string code, int argument)

returns 0 if executed, if not it calls LastInvoker to find command.

***<MemoryInvoker> MemoryInvoker.h***

It executes the instructions fetched from memory, if not found in memory, it calls MathtInvoker.

***Precondition:***takes the opcode and operand as arguments.

***Postcondition:***the opcode and operand are sent for the execution of the code. If not found then it will send opcode and operand to MathInvoker.

***Functions:***

*int execute (string code, int argument);*

exectueCommand: takes the opcode and the operand for the execution of the respective instruction defined in the instruction set. Returns zero if instruction is a valid instruction, Returns negative if one of the instructions is not valid

int MemoryInvoker::executeCommand(std::string code, int argument)

returns 0 if executed, if not it calls MathInvoker to find command.

***<GUIOpCode> GUIOpCode.h***

Abstract class, defines opcode classes to be used by invoker

***Precondition:*** classes need opcode classes to be used by the invoker.

***Postcondition:*** takes an input (some string for the execute to read) returns its output if it has one.

***Functions:***

virtual std::string getCode()

***<MathOpCode>MathOpCode.h***

Abstract class, defines opcode classes to be used by invoker

***Precondition:*** classes need opcode classes to be used by the invoker.

***Postcondition:*** takes an input (some string for the execute to read) returns its output if it has one.

***Functions:***

virtual std::string getCode()

***<MemoryOpCode>MemoryOpCode.h***

Abstract class, defines opcode classes to be used by invoker

***Precondition:*** classes need opcode classes to be used by the invoker.

***Postcondition:*** takes an input (some string for the execute to read) returns its output if it has one.

***Functions:***

virtual std::string getCode()