PROGRAMMER’S GUIDE:

**W10\_560\_Machine Design**

**🡪 Overview:**

Implementation of this machine is carried out using RESOLVE/C++ however strict RESOLVE conventions are not followed, for example parameter modes are not specified with function arguments and the word “object” does not precede all object declarations. RESOLVE foundation types are used in place of C++ primitives and RESOLVE components are use as much as possible. Function declarations follow the format *firstSecond()* where the first word/abbreviation is lowercase and the second word/abbreviation, if present, has a capital first letter followed by lower case letters. Object declarations follow the same convention as functions and class names are similar however the first letter of the first word/abbreviation is capital as well. Interfaces are not used therefore it is recommended that new classes/data-structures subclass an existing one. Since RESOLVE dose not provide components to easily store 20-bit encoded data or to easily work with hexadecimal values we have decided to create our own data-structures to make it easier to work with hexadecimal values and to store 20-bit encoded data. These data-structures provide functions, where applicable, to extract and store hexadecimal and decimal values and to extract necessary information from encoded instructions/input-data.

**🡪 Data-Structures:**

*word.h*

The class *Word* is the primary data structure that is stored in memory and used for registers. This class represents a 20-bit word and provides three constructors. The default no-arg constructor initializes the 20-bit word with a value of 0(zero). The other two constructors take an *Integer* and a *Text* object respectively as their only argument; if the value passed is an *Integer* the word is initialized to the decimal representation of the *Integer* and if the argument is a *Text* object the word is initialized to the hexadecimal representation of the *Text* object. This class also provides a series of functions to manipulate and to access the data.

*void setData(Text &);*

*void setData(Integer);*

*Text toHex();*

*Integer toDecimal();*

*void increment();*

*void Clear();*

*Word operator++(Word &);*

*Word operator&=(Word &);*

*instruction.h*

The class *Instruction* is provided to extract information from the encoded hexadecimal value of a known instruction, it subclasses *Word*. This class is used when a *Word* (a known instruction) is removed from memory and data needs to be extracted from the encoded value. The removed word is placed in an *Instruction* object and then appropriate functions are called to access the data. The default constructor does nothing to the data represented within the object; the *setData()* function must be used to initialize the data. The second constructor requires a Text object representing an encoded *W10\_560\_Machine* instruction. Provided are functions to set/reset the data and to extract the various parts of the encoded instruction.

*Text getData();*

*void setData(Text &);*

*Character getOp();*

*Integer getR();*

*Integer getX();*

*Integer getS();*

*hexAddress.h*

The class *HexAddress* is used to contain a 2-hex character value which corresponds to one of the 256 memory locations. Again three constructors are provided, the default no-arg constructor initializes the 2-hex character address with a value of 0(zero). The other two constructors take an Integer and a Text object respectively as their only argument; if the value passed is an Integer the word is initialized to the decimal representation of the Integer and if the argument is a Text object the word is initialized to the hexadecimal representation of the Text object. This class also provides a series of functions to manipulate and to access the data.

void increment();

Text getAddress();

void setAddress(Text &);

void setAddress(Integer &);

Integer toDecimal();

Boolean Is\_Equal\_To(HexAddress &);

HexAddress operator++(HexAddress &);

*headerRecord.h*

The class *HeaderRecord* is used to store the header record from the executable input file and to extract the various parts of the record. The no-arg constructor does nothing to the data within the object therefore it will not be initialized and subsequent calls to extract information will fail. A call to *setData()* will need to be made to initialize the object. The provide one-arg constructor, taking a Text object, will initialize the object so that information is ready to be extracted. A call subsequent to *setData()* will overwrite any header record data that is already in the object. The following functions are provided to manipulate the object and to extract the information within.

void setData(Text &);

HexAddress getBeginAddress();

Text getName();

HexAddress getLoadAddress();

Integer length();

*textRecord.h*

The class *TextRecord* is provided to store text records from the executable input file and to extract the address and encoded instruction. The no-arg constructor does nothing to the data within the object therefore it will not be initialized and subsequent calls to extract information will fail. A call to *setData()* will need to be made to initialize the object. The provide one-arg constructor, taking a Text object, will initialize the object so that information is ready to be extracted. A call subsequent to *setData()* will overwrite any header record data that is already in the object. The following functions are provided to manipulate the object and to extract the information within.

*void setData(Text &);*

*HexAddress getAddress();*

*Instruction getData();*

**🡪 Machine Implementation**

*w10\_560\_machine.h*

The class *W10\_560\_Machine* provides and implementation for the specified machine. The default no-arg constructor should not be used. Instead the provided 4-arg constructor should be used to pass the names of the executable input file, process input file, process output file and process trace file in that order. Once the machine object has been created the only function call that can be made is *start()*, and this can only be invoked once per object. This will load the executable and begin execution of its instructions.

*void start();*

**🡪Machine Initialization:**

Upon construction of the object, a private function *initialize()* is called. Within this function memory is initialized as a *Partial Map<Integer, Word>*. *Integer* is used to control the memory address and *Word* is used to store data. All 256 memory locations are allocated; this is the only time that the *Partial Map* member *Define()* is and should be used. All other accesses to memory should use the [] operator along with *getData()* or *setData()*, invoking these functions directly on the Word at the specified memory location(i.e. *memory[address].getData()*). Once memory allocation is complete all registers (*Word*) are given an initial value and all inputs and outputs are opened. The machine is now initialized, construction is complete and the user is free to call *start()* at their wish.

*void initialize(); (Private)*

The function *initialize()* is the process in which the W10\_560\_Machine’s memory simulation and registers are created and input/output streams are opened. As discussed previously, the memory of the machine is based off of the RESOLVE/C++ component *Partial Map*, (specifically *Partial\_Map\_Kernel\_3*). Our choice to use *Partial Map* was based primarily off of the accessibility and ease to which we could manipulate each ordered pair defined.

The function accomplishes the instantiation of our P*artial\_Map\_Of\_Memory* through the use of a *for* loop counting on integer variable *i*. Each ordered pair defined in this loop is defined uniformly as the appropriate integer address with the *Word* memory content assigned the value ‘-----’. Additionally, the registers r0, r1, r2, r3 are initialized with the same *Word* value, ‘-----‘. This consistency allows simpler error checking in other modules.

Also contained in *initialize()*, as mentioned above, the openings of inputs, *executableIn* and *processIn,* and outputs, *processOut* and *processTrace*.

**🡪Machine Execution:**

Execution begins with a single call to *start()*. If start has already been called any subsequent calls will do nothing. Within this function the machine is instructed to load the executable input file using the private function *loadExec()*. Upon successful loading to the executable a private function *run()* is called; this controls the fetch execute cycle of the machine. Within *run()* is a loop controlled by a *Boolean* object called *finished*; only a branch zero instruction can set finished to true to exit this fetch/execute loop. While inside the loop memory is accessed at the location specified by the program counter or *pc* and the contents, in hexadecimal, are used to construct an *Instruction* object. Using the function *Instruction::getOp()* the opcode is returned as a single *Character* and the appropriate op-function is called, passing the newly created *Instruction* object to the function. Once the function returns, either normally or by NO-OP, the *pc* is incremented end execution continues with the next instruction.

*Boolean loadExec(); (Private)*

The function *loadExec()* is primarily responsible for reading the executable input file and setting up the simulated memory. It also contains error checking for proper input and character usage, (see User’s Guide for proper syntax), and memory overwriting.

The function l*oadExec()* begins by reading in the header record from the executable input and storing it into a temporary *HeaderRecord* object. The begin address and name are extracted from the *HeaderRecord* object, checked for syntax errors, and stored. The text records following are then read and stored with the appropriate address provided in an “EOS” check loop. In this loop, the *TextRecord* address is checked for an overwrite error before storing. If a memory space is overwritten, the address is stored with a “no\_op” instruction to be interpreted by the *run()* function and printed to the *processTrace*.

*void incPc(); (Private)*

The function *incPc()* has the purpose of incrementing the global variable *pc*, which is the programs representation of the w10\_560 machine’s program counter. Like finalize(), writing a function to increment pc is to maintain the modular design approach.

*void Dump\_Memory(); (Private) and*

*void Dump\_Registers(); (Private)*

The functions *Dump\_Memory()* and *Dump\_Registers()* were designed to provide an aesthetically pleasing output of the complete memory structure and registers. The output can be seen in the *processTrace* output file and offer a complete record of final states for both the memory and the registers.

**Machine Instructions:**

These machine instructions, implemented as a single function for each instruction, are the primary source of execution of the executable input file. Listed by opcode, these functions are called within the fetch/execute cycle. To each function is passed an *Instruction* object which is created by accessing memory at the specific location specified by the *pc* register. The object passed contains the values of R, S and X which are referred to in the following function explanations.

*void op0(Instruction &); //LD R,S(X)*

The function *op0()* is used to load the contents at memory location S(X) into register R. This function works by first getting the decimal value of S and adding to it the decimal value of register X if X is not zero. This value represents the memory address that will be accessed, if this value is outside the limits of a valid memory address (i.e. value>255) then the function returns and this instruction is considered a *NO\_OP*. The contents at this address are then copied into register R without changing the contents at that memory address.

*void op1(Instruction &); //LDI R,S(X)*

The function *op1()* is used to immediately load the value of S(X) into register R. This function works by first getting the decimal value of S and adding to it the decimal value of register X if X is not zero. This value is then stored into register *R*.

*void op2(Instruction &); //ST R,S(X)*

The function *op2()* is used to store the value at register *R* into memory location *S(X)*. This function works by first getting the decimal value of *S* and adding to it the decimal value of register *X* *if X* is not zero. This value represents the memory address that will be accessed, if this value is outside the limits of a valid memory address (i.e. value>255) then the function returns and this instruction is considered a *NO-OP*. The contents of register *R* are then copied into the specified memory location without changing the contents of register *R*.

*void op3(Instruction &); //ADD R,S(X)*

The function *op3()* is used to perform an arithmetic addition of the contents at memory location *S(X)* and the value of register *R* and store the result in register *R*. This function works by first getting the decimal value of *S* and adding to it the decimal value of register *X* if *X* is not zero. This value represents the memory address that will be accessed, if this value is outside the limits of a valid memory address (i.e. value>255) then the function returns and this instruction is considered a *NO-OP*. The contents at this address are then added to the value of register *R* and the resulting value is stored in register *R*; the contents in memory remain the same.

*void op4(Instruction &); //SUB R,S(X)*

The function *op4()* is used to perform an arithmetic subtraction of the contents at memory location *S(X)* from the value of register *R* and store the result in register *R*. This function works by first getting the decimal value of *S* and adding to it the decimal value of register *X* if *X* is not zero. This value represents the memory address that will be accessed, if this value is outside the limits of a valid memory address (i.e. value>255) then the function returns and this instruction is considered a *NO-OP*. The contents at this address are then subtracted from the value of register *R* and the resulting value is stored in register *R*; the contents in memory remain the same.

*void op5(Instruction &); //MUL R,S(X)*

The function *op5()* is used to perform an arithmetic multiplication of the contents at memory location *S(X)* and the value of register *R* and store the result in register *R*. This function works by first getting the decimal value of *S* and adding to it the decimal value of register *X* if *X* is not zero. This value represents the memory address that will be accessed, if this value is outside the limits of a valid memory address (i.e. value>255) then the function returns and this instruction is considered a *NO-OP*. The contents at this address are then multiplied to the value of register *R* and the resulting value is stored in register *R*; the contents in memory remain the same.

*void op6(Instruction &); //DIV R,S(X)*

The function *op6()* is used to perform an arithmetic division of the value of register *R* and the contents at memory location *S(X)* and store the result in register *R*. This function works by first getting the decimal value of S and adding to it the decimal value of register X if X is not zero. This value represents the memory address that will be accessed, if this value is outside the limits of a valid memory address (i.e. value>255) then the function returns and this instruction is considered a *NO-OP*. The value of register *R* is then divided by the contents at this address and the resulting value is stored in register *R*; the contents in memory remain the same.

*void op7(Instruction &); //OR R,S(X)*

The input of this function is simply just an instruction. The overall purpose of this operation is to take the chosen register along with the memory at memory address *X* (given in the instruction), and to compare bit by bit and ‘OR’ the bits together. What it means to ‘OR’ the bits together, is to compare them and one of four results will happen. If both bits in the register and memory are equal to 0, then the result will be 0. If both bits are equal to 1, then the result will be 1. If the register bit equals 1 and the memory bit equals 0 (or vice versa) then the resulting bit will be 1. The bits must be in the same position in order to be compared, so the first bit in the register must be compared with the first bit in the memory, not the last bit.

In order to accomplish ‘ORing’ the register and memory at address *X*, a few things needed to be done. First the memory at address *X* is taken from the given instruction and converted to decimal. Next the contents of the appropriate register (designated by the instruction) are taken and converted to decimal. Now since you need these values to be in binary for a bitwise comparison, both are then converted to binary and stored as 1’s and 0’s in a text file. Then the bits, or characters, are compared and the resulting bit is stored in the register according to the requirements as described above. Once all the bits are compared (‘ORed’) and the result is in the register, then the resulting register is converted back from binary to decimal and stored as an integer. Once back in decimal form the result is simply set back into the register. The memory at address *X* is preserved, and the changes are made only to the designated register.

*void op8(Instruction &); //AND R,S(X****)***

The input for this operation is an instruction. With the given instruction, the purpose of this operation is to ‘AND’ the appropriate register and the memory together. To ‘AND’ is to do a bitwise comparison. There are two possible results of ANDing bits together. If both bits are ‘1’ then the result is a ‘1’. If only one or neither of the bits are ‘1’ then the resulting bit is a ‘0’.

In order to ‘AND’ the register and memory together the first thing which was done was taking the hexadecimal value of memory and converting it to decimal. Once it is in decimal, it is then converted to 20-bit binary. Next the register value is extracted and converted from hexadecimal to base ten decimal, and then to binary. This is done in the same way as the memory. Once the register and memory are converted to binary, they are compared bit by bit (bits in the same relative position). As described above, the result will be a ‘1’ if both bits are ‘1’ and the result will be ‘0’ otherwise. The result bit is stored in the result at the same position as the compared bit before moving on. Once all of the bits have been ‘ANDed’ together, the resulting register value is converted from binary back to decimal. Once in decimal it is set back to the original register, overwriting its value. The memory is preserved and the changes are only made to the designated register.

*void op9(Instruction &); //SHL R,S(X)*

The function *op9()* is used to perform an arithmetic shift left on specified register *R*. In the sense of binary numbers, the arithmetic shift function moves each bit of the number one bit to the left. The most significant bits are discarded upon the shift and the new bits introduced to the least significant bit spot are always 0. In the sense of decimal numbers the shift left operation is used to multiply the given number by 2. Therefore the equation for a number being shifted by *n* number of times will be:

New\_Number = Orig\_Number \* 2*n*

For the implementation of *op9()* there were two main algorithms considered. The first was to get the contents of the address *S(X)* and then to convert to a binary number and manually move each bit over as many times as the shift specified. The second, and much quicker, algorithm was to use a while loop to multiply the original number by two and then reassign it to the same data structure, then iterate this operation as many times as was needed. We chose the second algorithm due to performance and ease of writing code. The *op9()* function begins by retrieving the S field from the give Instruction *inst.* If the *S* field is not 0 then it is indexed by adding the register corresponding with the given *X* to the *S* field. The contents is then removed from this address and placed into an integer value *shift* implemented by *shift = memory[s].toDecimal().* Next the operation retrieves the integer value of register *X* and places into the integer object *val*. The arithmetic shift is now performed by using the formula *val \*= 2*, which multiplies *val* by 2 then reassigns the value to *val.* This formula is iterated *shift* number of times to give the new number. Finally the operation places the final *val* into register *R* by the line *r#.setData(val)* with the *#* being the register number. In this operation there are two possible error messages that can arise. The first is if the address S is above 255, then it is out of range and produces the error message *“Address Range Error”,* which is written to the *processOut* file. The second is if the value of *shift* is above 19 or less then 0, which means that the operation is trying to shift the register more than 19 bits. The error message *“Error: Cannot shift more than 19 bits”* is then written to the *processOut* file. Neither of these errors is fatal and should result in a *NO-OP.*

*void opA(Instruction &); //SHR R,S(X)*

The function *opA* is used to perform an arithmetic shift right on a specified register *R.* In the singed binary sense, the operation shifts each bit one bit to the right. The least significant bit in the binary number is discarder upon the shift. The most significant bit is then moved to the left but at the same time the value of the most significant bit does not change. In the decimal sense the *opA* function is used to divide a number by two. The equation for the *opA* function to be shifted *n* times is:

New\_Number = Orig\_Number / 2n

Since this operation was written after the *op9* function was written, the algorithm of choice was very obvious. We chose to divide the number the value of the register by 2 and iterate this over the number of times the shift calls for. As for the implementation of *opA* it is almost identical to *op9*. The process of retrieving the contents of *S(X),* retrieving the value of register *R*, and storing the new value after the shift are exactly the same, so all that information can be referred to in the above section for *op9.* Only new and unique implementations will be discussed in this section. As far as new implementations there is only one, and that is in the while loop to perform the shift. The shift will be performed by dividing the number retrieved from the register *R*, *val,* by 2 each time and reassigning the produced value to *val*. The line of code to perform this is *val \*= 2*. This will be iterated *shift* number of times, with *shift* being the contents at the address *S(X)*. The errors and error messages for this operation are the same as for *op9* and both are again not fatal.

*void opB(Instruction &); //IO R,S(X)*

The *opB* function is used to produce input and output for the w10\_560\_machine. There are different modes it can run, 2 input and 2 output. These modes are specified by what the value of the *R* field of the Instruction *inst* is. Since the *R* field is used to specify the mode of input or output, the *X* field of *inst* is used to find the register. Also since each mode of the machine is so different from the others, each mode will be discussed separately. The program is broken down into 4 main if-else-if statements, each of which uses the *inst.getR()* command to retrieve the *R* value of *inst*, and then this is used to test which integer it is equal to, 0-3.

**R = 0 mode**

This mode uses *opB* as an input function. The operation first consumes all the separator bytes at the beginning of the input stream *processIn*. The separator bytes are the ASCII values 0-32 and 127-160. The operation will continue to read these until the first “printable” byte appears. The operation then continues to read and consume printable bytes until it reaches the next separator byte or end of the stream. The operation will then check to see if the sequence of bytes is an integer that can be represented by 20-bit 2’s complement encoding. 20-bit 2’s complement encoding means that the integer must be between 0 and 1048576, since the number of total possible numbers that can be represented by a 20 bit number is 220 = 1048576. If the number can be represented by this then the value at the address of *S(X)* is cleared and the new value is placed in. If not then there is no change to memory or the registers.

The implementation of this mode follows a straightforward algorithm of getting the input then performing the processing and checks. The operation starts by reading in the *processIn* stream until the first non separator byte (ASCII 32 < b < 127) is read in or the stream ends. The operation now continues to read in non separator bytes and place them in ascending order into the word object *ins*. The operation will stop reading in bytes once the stream ends or another separator token appears. The first character of *ins* is now checked to see if it is either a “+” or “-“ (ASCII 43 and 45 respectively), and if it is, the character is removed. The text string *ins* is now checked to see if it can be converted to an integer using the procedure *Can\_Convert\_To\_Integer(ins)* and setting this equal to a Boolean object *truth*. If *truth* is false, then the operation will end. If true, the operation now converts *ins* to the integer *num* and checks to see I num can be represented by 20-bit 2’s complement binary notation (0 <= num <= 1048576). If true, then the operation will get the *S* field from *inst (inst.getS())* and then index it according to the *x* value. Finally the operation will clear the contents at *S(X)* and replace it with *num (memory[s].setData(num))*. There is one error that will be generated within this operation and that is the address range error (s < 0 or 255 < s). It will output the message *“Address Range Error”* to the *processOut* file.

**R = 1 mode**

This mode uses *opB* as an input function. The operation just consumes the first byte of the input stream *processIn.* The operation then obtains the values of register *X*. Then the 8 least significant bits are removed from the value obtained from register *X* and replaced with the byte that was read in from *processIn*. This new value is then placed back into the register *X*.

*R=1* mode will be implemented by first getting the register number by using the *inst.getX()* command and comparing it to either 0,1,2,3 to find the appropriate register. It will then create an integer *i* and text *t* which will hold the decimal and hex values of register *X*. The operation will now read the first byte of *processIn* into the character *byte*. The operation will now take off the 8 LSB’s of *t* by removing the last 2 characters. Now the operation will convert these two characters into a one bit decimal digit. Now this decimal digit is now subtracted from *i* to remove the last 8 bits. The operation now converts *byte* into an integer and adds it to *i*. Finally the data is put back into the register by using *r#.setData(i)* with *#* being the register number. There are no error messages that will be produced inside the function body.

**R= 2 mode**

This mode uses *opB* as an output function. The operation takes the contents at the address *S(X)* and checks to see if it is in the well-formed representation of a 20-bit 2’s complement number. This means that the integer must be between 0 and 1048576 because the number of total possible numbers that can be represented by a 20 bit number is 220 = 1048576. So by this representation, any integer below 524288 will be a positive number and any integer equal to or above will be negative. If this check is true then the operation will output to the *processOut* file. If not then the operation will output nothing to the *processOut* file. There is one error that will be generated within this operation and that is the address range error (s < 0 or 255 < s). It will output the message *“Address Range Error”* to the *processOut* file.

The implementation of *R=2* mode begins by using *inst.getS()* function to get the value of S then indexing using the *inst.getX()* function. The operation will then get the by setting integer *s* equal to *memory[s].toDecimal().* The operation now determines whether the value is positive or negative by checking if it is below or above 524828, respectively and writing output to the *processOut* file.

**R=3 mode**

This mode uses *opB* as an output function. The operation takes the 8 most significant bits of register *X* according the simple binary scheme. To do this it must read in the first two hex characters from the register and convert them to a single integer byte. This will then be output to the *processOut* file without affect the *S* field of *inst.*

The R=3 mode will be implemented by first using the *inst.getX()* function to get the appropriate register number and setting text *t* equal to the hex value of that registers contents. The operation now removes the 8 most significant bits of ­­­­*t* and placing them into characters *ch1* and *ch2*. The operation will now convert the two hex numbers into one decimal number and output this integer to the *processOut* file.

*void opC(Instruction &); //BR R,S(X)*

This operation takes only an instruction as input. The overall goal of this operation is to provide and unconditional branch, or GOTO, to somewhere else in the program. The branch to another part of the program depends largely on what *R* is equal to, or which register is selected. The first thing that is done is the memory is extracted from the instruction and stored in a decimal to be used later on. Next, the *R,* or register, is determined.

If the register is register *0*, then the action depends on what *X* is equal to. If *X* equals 0 (dec), then nothing is done. This is called being “quiet”. If *X* equals 1 (dec), then a dump of all memory is done. This does not mean the memory is erased, it simply means that all memory is displayed, or output. If *X* is equal to 2 (dec) then another dump is done, but instead of the output being the memory, it is the registers and the program counter. In the last case, if the *X* is 3 then the memory as well as the registers and program counter are all dumped.

If the register is register 1, then the action is simply to dump everything and branch to address specified by memory. This was done by outputting the memory, the registers, and then the program counter. Then, since we already took out the value of memory, we simply store that in the program counter.

If the register is register 2, then the action is to branch to the address in memory of the program counter plus the value of memory. Since we took out the value of memory already, we simply add the value of the pc (after extracting it in decimal form) to the value of memory, and then set that data back into the program counter.

If the register is register 3, then the action is to simply branch to the address in memory. Since we already took the value of memory out as a decimal value, the only action needed is to set it to the program counter.

*void opD(Instruction &); //BRZ R,S(X)*

If this operation is called, then the action is simply to “Branch if Zero”. This means if the value of the register is equal to 0 then the program should branch to the address specified by memory. This is done by extracting the value of the register in decimal notation. The value is then checked, and if the value is 0 then the program counter is set to the value of memory which we took out before. If the register has any other value nothing is done.

*void opE(Instruction &); //BRN R,S(X****)***

If this operation is called, then the action is simply to “Branch if Negative”. This means if the value of the register is less than zero (in two’s complement) then the program should branch to the address specified by memory. This is done by first extracting the value of the register in decimal form. Then, since in two’s complement, any value above 127 (dec) will be negative that is the number the value is compared to. If the value of the register is 128 (dec) or higher then it is negative, and the program counter is set to the value of memory which we took out before. If the register has any other value then nothing is done.

*void opF(Instruction &); //BRS R,S(X)*

If this operation is called, then the goal is to “Branch to Subroutine”. The program counter is stored in the register and then the program branches to memory at the specified address. First, the value of the program counter is extracted as a decimal and stored in the register which is specified by the instruction. Next, we take out the value of memory, in decimal format. We then take that decimal and put it into the pc, which will then make the program counter choose the next instruction from memory at the address from the previous instruction.

**Machine Termination:**

*void finalize(); (Private)*

The function of *finalize()* is to close input, *execIn* and *processIn,* and outputs, *processOut* and *processTrace*. These relatively simple actions were written as a separate function to maintain our modular design approach.