CMPE 230 Systems Programming Project 1

1.Problem Description:

We were asked to implement an interpreter using C/C++ or Java for an assembly language of a hypothetical 8086- like CPU called HYP86. The input file would be given as an argument and all the scanning and printing operations should be done using console. There are several assumptions and limitations in the project and if there occurs an error the program should be terminated.

2. Implementation Details:

✓ Main Class:

This class handles all the important work; reading input, calling necessary functions stored in different classes, storing maps & arrays, writing outputs. Used datatypes and variables are explained as follows:

- ***Memory[] :*** 64 kb sized memory array.
- ***Console Scanner :*** Scanner which reads from the console.
- ***twoByteRegisters[] :*** Array which holds the values of AX, BX, CX, DI, SI and BP regs. Note that java doesn't support unsigned type it is an int type array.
- ***oneByteRegister[] :*** Array which holds the values of byte sized regs. Note that this array is a short type array.
- ***pc :*** Stands for program counter. It is incremented when new line is read.
- ***sp :*** Stands for stack pointer.
- ***ip :*** Holds the memory address of last available position after instructions.
- ***Zf, sf, cf, af, of*** in order stands for zero, sign, carry, auxiliary, overflow flag.
- ***labelNo Map :*** Holds the line number of labels.
- ***wordVariables & byteVariables Maps***: Holds the addresses of variables.
- ***commands ArrayList :*** Holds all of the instructions given as input file.
- ***Line number:*** Integer that used in for loop in second part of main. Each time a new element is read from Command arraylist, its value is increased by one.
- ***Boolean start & end:*** They are the Boolean values that decide which lines in the file will be evaluated.
- ***Splitting command line:*** In fact, this is not a method or variable, but it is explained here because this part has an important role. The brackets read in the line are

changed to " [" and "] ", semicolons are changed to" ; " , and commas are change to " , ". If there is more than one space after the line, it is trimmed to a single space. It is separated according to the spaces and in this way, each operand falls into a different index in the operands array. The fact that if the character read was blank, or was characters like '[' then we were in trouble. To prevent this, the character in the quote is replaced with "g", which is not a problem, before splitting. The character is reinstated after the split is finished.

- ***Operands:*** After the received line is modified and divided into pieces according to the space, the pieces are thrown into this array.
- **checkVariableTwoOperand:** By considering all cases, this method determines first and second operand of operation. It converts variable names into memory addresses as A86 does. If "offset" is written in front of the variable, the memory address of that variable is found, and the value found is assigned to the second operand. If offset is written in front of the first operand, it gives an error. This method also gives error when there is: "Too many arguments", "Missing comma", "Unknown operand" etc.
- **checkVariableOneOperand:** This method has similar logic with checkVariableTwoOperand. The difference is it works for instructions who has just one operand.

Also, it is helpful to specify that all the AX, BX, CL, DH, … values symbolize the index of these registers in their relevant arrays.

Main class can be separated into two main parts. First one is reading the whole text from the input file and determining labels, variables and where the HYP86 code starts & ends. Start variable becomes true when "code segment" line is read and end variable becomes false when "code ends" line is read from file. All instructions between these two lines are added into commads ArrayList, if they do not contain any syntax error. Second one is basically processing the instructions one by one. Instructions are determined by looking first operand in operation array. After that with checkVariableTwoOperand or checkVariableOneOperand first and second operands (if needed) are determined. Most instructions are executed by calling their classes. But the ones listed below are simple and are handled in this class.

- ***JMP / JZ / JE / JNZ / JNE / JA / JNBE / JAE / JNB / JNC / JB / JNAE / JC / JBE :*** If the necessary flags are in right status program is jumped into the desired line.
- ***INT 20h :*** The code simply terminates.
- ***INT 21h :*** If AH register has the value 1 in it the programs reads a character from the console into AL register, else if the value is 2 the program prints out the value stored in DL register and a copy of this value is stored in AL register.
- ***NOP :*** Nothing is done.

✓ Helper Class:

This class, like the name suggests, contains helpful in other words most common methods that are used in all of the instruction classes. All of the other classes excluding Main extends this class and often use its methods. This class extends Main.

- ***determineWordRegister & determineByteRegister :*** Checks whether given string is a register, returns the index if it is a register, returns -1 if it is not.
- ***convertInteger :*** Since integer values can be given in different bases this method handles the conversion of any integer into decimal base and returns it. Else if second operand is sp, it returns value of sp. Else Boolean isNumber becomes false and it returns -1 .
- ***determineRegisterAddress :*** If a register is used in the format of indirect address this method returns the address value stored in that register. Else Boolean registerIndirectAddressing becomes false and it returns -1.
- ***determineAddress :*** Like in the convertInteger method integer values can be in different base, this method converts any address(register indirect, direct addressing) into decimal base and returns it. Else Boolean isAddress becomes false and returns -1.
- ***updateChild :*** Word registers and byte registers are stored in different arrays so if an instruction changes the value of either AX, BX, CX or DX requisite registers are updated.
- ***updateParent :*** Similar procedure in updateChild, if any of the registers which has a parent register is changed necessary register is updated accordingly.

✓ DefineVariable Class:

This class is used when declaring variables in the first part of the execution. Bp register is used for finding the next available location in the memory. If the given variable is in word sized ***defineWord*** is called and the next two memory locations are updated, else the given variable is byte sized and ***defineByte*** is called and the next one memory location is updated.

✓ Add Class:

If the current instruction is ADD then the matching one of the following method is executed.

- ***addToWordReg :*** In main class the word sized destination register is found and passed as “destination” parameter. After checking possible variants for secondOperand the addition is done, if secondOperand doesn’t match any of these we terminate, also if the given register has sub-registers they are updated at the end of the method.

- ***addWordToSp :*** Same procedure in above method is done. This method is written because sp register is not stored in twoByteRegisters array.
- ***addToByteReg :*** Byte version of addToWordReg method.
- ***addToWordMemory :*** is called if the prefix of the memory is W or no prefix is given. It checks the type of secondOperand, if it is a byte register then addToByteMemory is called with the same destination and secondOperand parameters. If it is not the case value of the secondOperand is found and added into the required memory location.
- ***addToByteMemory :*** Byte version of addToWordMemoy method.

✓ BooleanOperation Class:

If the current instruction is one of XOR, OR, AND, NOT one of the following method is called, to clarify the ambuigity type 0 is XOR, type 1 is OR and type 2 is AND.

- ***wordToReg :*** If the destination operand is a word sized register then this method takes place. It determines the type and value of the secondOperand and performs the necessary operation depending on the type. updateChild is called if destination has sub-registers.
- ***byteToReg :*** Byte version of the above method.
- ***byteToMemory :*** This method is directly executed from the main if firstOperand has the 'b' prefix, or wortToMemory method can call this if firstOperand doesn't contain prefix and secondOperand isa byte sized register. It performs the operation based on the types and stores the result in memory.
- ***wordToMemory :*** is called if the prefix of the memory is W or no prefix is given. It checks the type of secondOperand, if it is a byte register then toByteMemory is called with the same destination and secondOperand parameters. If it is not the case value of the secondOperand is found and necessary operation is done and related memory locations are updated.
- ***notOperation :*** Since not instruction only takes one operand this instruction is distinguished from the other three. It converts 0's to 1 and vice versa.

✓ Compare Class:

This class handles CMP instruction. It is much similar to Subtraction Class. It subtracts secondOperand from firstOperand but firstOperand is not updated; only the flags are changed according to the result.

✓ Divide Class:

If the instruction is DIV, one of the first 5 method is called.

- ***divideByWordReg :*** Concatenates DX register with AX register, divides this with given word sized register and stores the quotient in AX register and remainder in DX register.

- ***divideByWordMemory :*** Determines the value by concatenating the required memory locations and follows the same procedure done in above method.
- ***divideByByteReg :*** Divides AX register with the given one byte sized register and stores the quotient in AL register and remainder in AH register.
- ***divideByByteMemory :*** Value stored in the given address is found and same procedure is done in above method.
- ***divideByImmediate :*** Value of the sp is checked and divideByByteNumber is called if it is in range of byte or divideByWordNumber is called.
- ***divideByByteNumber :*** Divides AX register with value of sp and stores the quotient in AL register and remainder in AH register.
- ***divideByWordNumber :*** Divides DX : AX duple with value of sp and stores the quotient in AX register and remainder in DX register.

✓ Move Class:

If the instruction is MOV, one of the matching method is called.

- ***moveWordToReg :*** Determines the type and availability of the secondOperand and copies the value of it into the given word register. If the destination register has sub-registers they are modified.
- ***moveByteToReg :*** Byte version of moveWortToReg.
- ***moveWordToMemory :*** If the destination register has sub-registers then move operation is done between value of the low memory address and _L register & high memory address and _H register. If it doesn't contain sub-registers first concatenation is done and modification is done using substrings.
- ***moveByteToMemory :*** Simply copies the value of the given memory address into the destination byte register.
- ***moveWordToSp :*** Same procedure in moveWordToReg method is done. This method is written because sp register is not stored in twoByteRegisters array.

✓ Multiply Class:

If the instruction is MUL, one of the first 5 method is called.

- ***multiplyByWordReg :*** Multiplies the given word sized register with AX register and stores the 16 right-most digits in AX register and the rest of the digits in DX register.
- ***multiplyByWordMemory :*** Using the given memory address word sized value is obtained and multiplied with AX register. Same as in above method 16 right-most digits are stored in AX register while the rest is stored in DX register.
- ***multiplyByByteReg :*** Multiplies the given byte sized register with AL register and store the result in AX register.
- ***multiplyByByteMemory :*** The value stored in the given address is multiplied with AL register and result is stored in AX register.

- ***multiplyByImmediate :*** Value of the sp is checked and multiplyByByteNumber is called if it is in range of byte or multiplyByWordNumber is called.
- ***multiplyByByteNumber :*** Multiplies AL register with value of sp and stores the result in AX register.
- ***multiplyByWordNumber :*** Multiplies AX register with value of sp and stores the 16 right-most digits in DX register and rest in AX register.

✓ Rotate Class:

This class contains methods for left rotate (RCL) with carry and right rotate (RCR) with carry, if the passed parameter called "type" is true right rotate is done, left rotate is done otherwise. Note that the number of rotations has 2 alternatives which are an integer in range [0, 31] and CL register.

- ***bigReg :*** Given word sized register is rotated accordingly after binding carry flag to the register and obtaining the 17-Bit representation, updateChild is called if necessary.
- ***smallReg :*** Given byte sized register is rotated accordingly after binding carry flag to the register and obtaining the 9-Bit representation, updateParent is called if necessary.
- ***memory :*** Determines the type of the memory, creates the 17-Bit representation if it is word and 9-Bit representation if it is byte. Performs the rotation and updates the memory locations.

✓ Shift Class:

This class contains methods for left shift (SHL) and right shift (SHR), if the passed parameter called "type" is true right shift is done, left shift is done otherwise. Note that the number of shifts has 2 alternatives which are an integer in range [0, 31] and CL register.

- ***bigReg :*** Given word sized register is shifted accordingly, updateChild is called if necessary.
- ***smallReg :*** Given byte sized register is shifted accordingly, updateParent is called if necessary.
- ***memory :*** If the type is word concatenation is done and shifting is made after that, otherwise shifting is directly done to the memory address.

✓ StackController Class:

This class includes instructions for POP and PUSH.

- ***push :*** Checks if the given operation is word sized, if it is not program terminates. After this confirmation given word value is assigned into the next empty location in the stack part of the memory. 2 is subtracted from sp afterwards.

- ***pop :*** Likewise confirmation of the word value is done. The most recent pushed value in the stack part of the memory is moved to the destination. 2 is added to sp before these steps.

✓ Subtract Class:

If the current instruction is SUB then the matching one of the following method is executed. Note that when source value is greater than destination value the result is negative therefore 65536 (2^{16}) is added for word situation and 256 (2^{8}) is added for byte.

- ***subtractWordToReg :*** Checks if secondOperand is valid and if so subtract its value from the given word register.
- ***subtractWordFromSp :*** Since sp is not stored in twoByteRegisters array this method is called if and only if firstOperand is sp and secondOperand value is subtracted.
- ***subtractByteToReg :*** Byte version of subtractWortToReg.
- ***subtractWordToMemory :*** Firstly subtrahend is formed using the given memory address. Secondly secondOperand is checked whether it is eligible or not. Lastly subtraction is done ad memory addresses are updated.
- ***subtractByteToMemory :*** Validity of the secondOperand is checked and subtraction is done afterwards.

3. Conclusion:

The program prints out the exact match for all the given tests. It contains all features which are specified in description and it also has some extra features that we haven't had to implement such as giving the type of the error, stating which line the error is occurred etc. We also consider the cases where w/b is not written in front of the variables and memory addresses. Additionally, we tried to name the variables and methods of each class to be comprehensible so it would be easy to grasp how the program works.

Project is implemented by :

Hakan BALIK & İrem Zeynep ALAGÖZ

2018400177 - 2018400063