CMPE 230 Systems Programming Project 1

Salih Bedirhan Eker
Ece Dilara Aslan

1. <u>Global Variables</u>

In this section, we explain what global variables are used for and how they work.

map_labels <string, int>:
This map holds the label name (key) and the place of the first instruction (value) after that label. When a jump occurs, this map is used and Change is readjusted.

map_variables <string, pair<int, pair<int, int>>:
This map keeps the information about the variables. Keys are names of variables and values are pairs of int and int-int pairs. First int stores the value of that variable, second pair's first int stores address of it and second pair's second int stores the size.

set_instructions <string>:
This set contains the names of possible instructions. It helps us to check whether a string is an instruction or not.

map_registers16 <string, int>:
This map contains the names of 16-bits registers, and initialize them at the beginning to the value zero except sp (sp is initially FFFE).

map_registers8 <string, int>:
This mapcontains the names of 8-bits registers, and initialize them at the beginning to the value zero.

map_flags <string, bool>:
This map contains the names of necessary flags, and initialize them at the beginning to the value zero (false).

vector_simulation <vector<string>>:
This vector keeps instructions in the right order, every subvector is an instruction furthermore every element of a subvector is a token for an instruction. For example simulation[3] = {add, ax, var} means fourth instruction is an add instruction and elements of the subvector are add, ax ,var.

bool_isPrinted:
This boolean value checks whether error message is printed or not. It prevents that the error message is printed more than once.

bool_error:
This boolean value checks that there is an error. When it is set to true, the program prints an error message and terminates.

vector_change <string>:
This vector makes necessary register arrangement after every instruction and it is used in update() function. It keeps the connection between 8 and 16 bit registers.

int_Counter:
It decides which instruction is being executed. If any jump is not occurred, it just points to the next instruction, otherwise Counter is set the right instruction after the jump.

int_mem[]:
The memory of the program. Every information (instruction, variable, numbers, stack) is put into the memory.

2. Helper Functions

In this section, we explain the work that helper functions do, their parameters/returns and how they work.

bool isNumber (string s):
This function returns true if the parameter s is a number. s can be form of ##h, ##d, 0## or ### (# is any number). First if checks if form of s is the ##h, if it is true then checks the form is correct (first character is a digit and others are digits or a,b,c,d,e,f). Otherwise checks it starts with zero (0) (means still the number is hex). If not checks the number is decimal number (##d or ### (not starting with zero)). Still it is not true then return false. Means parameter s is not a number.

int decimal (string s):
This function assumes s is a number because it is used after isNumber function. And function returns the decimal value of s (whether it is hex format or not).

int valueOfAddress (string address):
This function returns the value of the given address. First it checks parameter points actual a memory location by a number or proper registers (bx, bp, si, di), and if it is correct returns the value of that address. If address is word size it returns the sum of mem[address]+mem[address+1]*(2^8). Otherwise, it just returns the value at that address (mem[address]).

int findAddress (string address):
This function returns what the adress is, if parameter is not appropriate returns -1, means this is not a valid address.

void update():
This function updates 8-bit and 16-bit registers after every instruction. For example if ax is changed, it updates ah and al. The bound between 16-bit registers and its high and low parts still remained because of this function. It uses change vector, the vector holds the registers that needed to be changed cleared after the necessary updates.

string toLowerCase (string line):
This function basically converts the string to lower case.

void isTrue (string s):
This checks label or variable name is valid. It checks first character of a variable or a label and after that checks the rest of the characters, if any error occurs function gives an error.

void arrange():
This function deals with offsets, characters and b/w varName and converts them to a proper state. First, offset varName's are changed to address of that variable. Also it handles b/w varName. Finally it converts characters into Ascii values.

vector<string> token1(string line) and vector<string> token2(string line):
Tokenizer functions. They remove <space> characters, tokenize and clean the input lines. After that they makes lines operable and ready to be pushed to the simulation. token1 handles lines before int20h and token2 handles lines after int20h.

3. Functions of Instructions

This section clarifies the instruction functions generally. If something needs to be explained specifically, it will be explained after general features.

Most of the instructions work in a same manner, first decides the case we're handling. In order to do that; if the instruction takes two operands –one of them is destination and the other one is source operand- , it decides what is the type of destination initially as is it a 16-bits register, is it an 8-bits register, is it a variable or is it a memory location. If none of them is proper type, then error is set. After the destination is determined; it is time to decide the type of source operand as is it a 16-bits register, is it an 8-bits register, is it a variable, is it a memory location or is it a number. If the type of source is not appropriate for the type of destination, for example destination and source are both memory locations, then error is set. If the instruction takes one operand, like the two-operands instructions it finds out what is the type of the operand and if it does not fit the proper types error is set.

After specifying the types, we consider the size compatibility. We check sizes of operands are compatible, as 8-bits and 8-bits/16-bits and 16-bits operands are compatible, if the instruction takes two operands. If the instruction takes one, then we check if the operand size fits the situation. For example, <pop> and <push> instructions need 16-bits operand due to the size of each element of the stack. However; <mul>, <div> and <not> may take 8-bits or 16-bits operands. <not> executes the same procedure for all of the proper types, but the operations of <mul> and <div> differ a little bit.

In addition, <rcl>, <rcr>, <shl> and <shr> operations have an exception. The second operand cannot be all of that we have mention above. It can be cl register or 8-bits number. So, we first decide the number of shifting/rotating according to source operand in shift (string dest, string source, string op) function. If the number of shifting/rotating is zero, it means there is no change in the number and any flag will be affected. So, we do nothing, just increment the Counter by one and continue to execute the program. Then, we try to find the type of the destination like we explained.

The instructions that we have written above are other than <nop>, <jmp>, conditional jumps, <int 21h> and <int 20h>. These instructions work in a different way. The only function of <nop> instruction is increasing the Counter by one to move to the next instruction. Also, it takes a space in the memory. <jmp> and conditional jumps take one operand and that operand must a label name. Otherwise, error is set. <jmp> function moves the counter to the first instruction after the label. Conditional jumps executed by jc(string label, string op) function. This function checks flags according to the type of the conditional jump. If the flag conditions are hold, it calls the jmp(string label) function. If not, Counter will be increased by one to get to the next instruction. <int 21h> instruction reads/writes a character according to the value of the ah register. For this project, the valid subfunctions in ah are 01 and 02. If there is an another subfunction, error is set. <int 20h> instruction moves the Counter to the end of the simulation vector in order to finish the program.

If the instruction affects flags, then there are some local variables at the beginning of the function of that instruction. Because we need to use the value of the result and operands of the operation. Also when a register changes because of an operation, we push_back that register and at the end of the function we call update() function to keep the high-low parts and the whole part connected. Furthermore, if we change a variable's value, we update that value in the location of variable's address in memory. And, when a memory location is an operand, we check whether this location stores an instruction or not, also if we need to use the value of the memory location, we check whether it is empty or not. Also, if a value of a memory location changes, we check whether there is a variable and if so change the value of that variable. If flags are affected, after the current operation executed flags are updated. And, at the end we increase Counter by one to get the next instruction, excluding <jmp>, conditional jumps and <int 20h>.

4. <u>Main Function</u>

In this section, main function -reading the source code of assembly, inserting labels and variables with their corresponding number, creating simulation vector in the proper situation and executing simulation vector- will be explained.

Main function starts with reading the assembly source code. It checks whether the code starts with "code segment". After that, it keeps reading the code until <int 20h> instruction. We take each line, parse them with token1(string line) function and makes the letters of the tokens which are not characters lowercase. These lines could be labels or instructions. If it is a label line, we control the syntax of label name and if it is used before. If there is no error, we insert them into the map_labels<string, int> with their corresponding instruction line. If it is an instruction, we push them into the simulation and fill the corresponding locations of memory with minus twos.

After we reached <int 20h> , we keep reading until "code ends" line (if we cannot reach <int 20h>, error is set). We take each line, parse them with token2(string line) and makes the letters of the tokens which are not characters lowercase. These lines must be variable definitions. If not, error is set. If so, we control the syntax of the variable name and if it is used before. If everything is fine, we fill the corresponding address of memory with its value and insert them into the map_variables<string, pair<int, pair<int, int>>> with their corresponding values, addresses and sizes.

After we reached "code ends", we check whether we got an error while reading the assembly source code. Then, we call arrange() function to do the adjustments and start executing the code using simulation vector. For every instruction, we call the appropriate function in our code. It continues until the Counter reaches the size of the simulation vector. In this manner, the assembly code will be executed.