*Ezgi A. Batı - 2018400207*
*Z. Didem Aytaç - 2018400045*

# CMPE 230 SYSTEMS PROGRAMMING
# PROJECT DOCUMENTATION

This documentation expresses our approach to implement an interpreter for an assembly language of a hypothetical 8086- like CPU called HYP86, using C++.

## *General Structure / Approach*

We tried to implement greater amount of values as global variables so that they are reachable and editable also inside the functions. We have registers, flags and pointers as global variables.

*map<string,*reg> sixteenBitRegs / eightBitRegs* : two maps for two sizes of registers, keys holding the name of the register and values hold the a pointer to the corresponding register.

*vector<string> lines* : holds the instructions read from file.

*map<string,int> labels*: key is the label name as string and the value stands for the starting index of the label.

*map<string, pair<int,unsigned short> vars* : a construct to save the variables of the program, key is the variable name as string, and corresponding values hold the size and address of the variable.

*unsigned char memory [2<<15]* : The memory as an array. The head of the array holds the instructions, the tail holds the stack. We use the remaining part for holding memory values.

We implemented registers as a class, for details see the next section.

We implemented instructions as void functions. For most of the instructions, we have more than one function. This is because instructions mostly take different kinds of parameters, and we find it more stable to implement those separately. These functions may be specified in the following sections of the document.

We also have some helper functions, which will also be discussed later.

The main function first does the insertion of maps and updating only nonzero register, which is sp. Then comes the file reading part. The first reading aims to read the input file properly. The second reading executes the instructions. The main function will be discussed in detail, in the preceding section.

## *The reg Class*

The implementation consists of a class named reg, which stands for register, because registers have many fields. We implemented fields of the reg class as:

*int regType* (to hold the size of the register),

*unsigned char eightValue* (to hold the values of 8 bit registers),

*unsigned short sixteenValue* (to hold the values of 16 bit registers),

*unsigned char low / high* ( to hold low and high values for ax, bx, cx and dx),

*string name*.

Functions belong to this class are as following:

*reg(int type, string name)* : The constructor, which initializes type and name to given values in the parameters and others to 0.

*void update_8 (unsigned char val) / void update_16 (unsigned short val)* : two update functions, special for two different type of registers. The update_8 function changes the value of the 8 bit register, and updates corresponding 16 bit registers value, if it exists. Likewise, update_16 function changes the value of the 16 bit register, as well as low and high fields of that register and also updates corresponding 8 bit registers values, if exists.

### *The Main Function*

As explained before, the main function first assigns the true initial default value to the stack pointer register. Then does the insertions to the maps sixteenBitRegs and eightBitRegs. These maps help us to make searches among registers by name and then allow us to reach to a pointer of that register.

After these, we open the input file which is given as an argument to the program. If the file is reachable, we start the first reading process. In this step, we take the input and declare the variables. We trim leading whitespaces, and count number of instructions (ignoring labels) to be able to fill the head of our memory array truly. We handle the variable declarations at this step and also when we encounter a *label*, we insert it to the vector named labels.

After first reading is completed, comes the second reading process. This part aims to execute the instructions given as input. We start reading the input line by line from the lines vector we have created. Then we check the condition with the *int* instructions. We have two possibilities of int instructions in the language, either 20h or 21h, and the program performs necessary actions here.

The next possible instructions we check are *jump* and *conditional jumps*.

We proceed with checking *operations* that take one operand, two operands and those that can take more than two operands, respectively. In all tree of the possibilities we first extract the operand(s) and check the type of the operand (see findOperandType function in the Helper Functions section below) After that, for possible combinations of operand types, we check what the instruction is. And finally call the appropriate instruction with the corresponding parameter type(s).

### *Helper Functions*

*unsigned concat (unsigned char a, unsigned char b)* : simply concatenates binary values of two unsigned char values into an unsigned short value and returns the result. The function is mostly used when reading an unsigned short value, divided into consecutive places of the memory.

*unsigned long hex2dec (string hex)* : function returns the decimal value of a given number in hexadecimal representation. This function eases the procedure when we encounter with hexadecimal values.

*pair<int,unsigned int> realAdressFinder (string address)* : a function used when an instruction has a memory location as its operand. The addresses can be given in many different ways in the Assembly language. To simplify the process in the program, when a memory operand is read, this function evaluates all different kinds of representations into one unique form. The unique form is a pair<int,unsigned int>, where the first element is the size of the value (8 bit or 16 bit) and the second element is the address in decimal representation. The function first deletes the brackets and finds the plain dress between the brackets. If this string is a register name, the value of the register is extracted. If not, it is a decimal or hexadecimal number. The functions takes all possibilities into account and returns the appropriate pair.

*unsigned long long int decToBinary(unsigned long long int n)* : converts decimal number into binary representation and returns the result. This function aims to do necessary checks during implementation.

*char findOperantType(string op)*: a function called inside the main function. After an instruction is read, we read the operand(s). These operand can be a memory location, a register, a variable or a constant. The function aims to decide which one and returns the beginning letter of the type as a char.

*unsigned char MSB_16(unsigned short temp), unsigned char MSB_8(unsigned char temp), unsigned char LSB_16(unsigned short temp), unsigned char LSB_8(unsigned char temp)* : returns most or least significant bits of 8 or 16 bit values

### *Functions of Instructions*

***MOV:*** The mov instruction copies the data item referred to by its second operand (i.e. register contents, memory contents, or a constant value) into the location referred to by its first operand (i.e. a register or memory). 5 separate mov functions stand for 5 different possibilities of source and destination combinations. All the functions get the value in the second parameter and assign the first parameters value to that. Program terminates if the two parameters do not have the same size.

*void mov_opr(reg* reg1, reg* reg2)* mov <reg>,<reg>
*void mov_opr(reg* reg1, pair<int, unsigned short> realAddress)* : mov <reg>,<mem>
*void mov_opr(pair<int, unsigned short> realAddress, reg* reg1)* : mov <mem>, <reg>
*void mov_opr(reg* reg1, unsigned constant, char constType)* : mov <reg>,<const>
*void mov_opr(pair<int, unsigned short> realAddress, unsigned constant, char constType*):mov <mem>, <const>

**CMP:** The comparison is performed by subtracting the second operand from the first operand and then setting the status flags in the same manner as the SUB instruction. 5 separate cmp functions stand for 5 different possibilities of operand combinations. Program terminates if the two parameters do not have the same size.

*void cmp_opr(reg* reg1, reg* reg2):* cmp<reg><reg>
*void cmp_opr (reg* reg, pair<int,unsigned short> mem)* : cmp <reg><mem>
*void cmp_opr(pair<int,unsigned short>mem, reg reg)* : cmp<mem><reg>
*void cmp_opr(reg* reg, unsigned short con)* : cmp <reg><con>
*void cmp_opr(pair<int,unsigned short>mem, unsigned short con)* : cmp <mem><con>

**STACK OPERATIONS:** SP points to the top free location on the stack. The stack grows backwards and push and pop operations update the value of the sp register.
The push instruction places its operand onto the top of the hardware supported stack in memory.

*void push_opr(reg* reg)* : push <reg>
*void push_opr(pair<int, unsigned short> realAddress)* :push <mem>
void push_opr(unsigned constant,char constType) :push <con>

The pop instruction removes the data element from the top of the hardware-supported stack into the specified operand

*void pop_opr(reg* reg)* : pop <reg>
*void pop_opr(pair<int, unsigned short> realAddress)* : pop <mem>

**MATH OPERATIONS**: Perform necessary math operations properly and does error checks and necessary assignments. AF, CF, OF, ZF SF flags are set accordingly. For add and sub the destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. For inc, dec, mul and div the single operand can be a register or a memory location.

*void add_opr(reg* reg1, reg* reg2)* :add<reg><reg>
*void add_opr(reg* reg, pair<int,unsigned short> address)* :add <reg><mem>
*void add_opr(pair<int,unsigned short> address,reg* reg)* :add <mem><reg>
*void add_opr(reg* reg, unsigned short constant)* :add <reg><con>
*void add_opr(pair<int,unsigned short> address, unsigned short constant)* :add<mem><con>
*void sub_opr(reg* reg1, reg* reg2)* : sub<reg><reg>
*void sub_opr(reg* reg, pair<int,unsigned short> mem)* :sub <reg><mem>
*void sub_opr(pair<int,unsigned short>mem, reg* reg)* :sub<mem><reg>
*void sub_opr(reg* reg, unsigned short con)*
*void sub_opr(pair<int,unsigned short>mem, unsigned short con)*
*void inc_opr(reg* reg)*
*void inc_opr(pair<int,unsigned short> address)*
*void dec_opr(reg* reg)*
*void dec_opr(pair<int,unsigned short> address)*
*void mul_opr(reg* source)* :mul<reg>
*void mul_opr(pair<int,unsigned short>address)* :mul<mem>
*void div_opr(reg* operand)*
*void div_opr(pair<int,unsigned short>address)*

**LOGICAL OPERATIONS:** All of and, or and xor functions take two operands and work the same except the operation they perform among the parameters: bitwise and, bitwise or and bitwise xor. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location.The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined. 5 separate functions stand for 5 different possibilities of source and destination combinations, for each instruction. Program terminates if the two parameters do not have the same size.

*void and_opr(reg* reg1, reg* reg2)* :and <reg>,<reg>
*void and_opr(reg* reg1, pair<int, unsigned short> realAddress)* : and <reg>,<mem>
*void and_opr(pair<int, unsigned short> realAddress, reg* reg1*) : and <mem>, <reg>
*void and_opr(reg* reg1, unsigned constant, char constType)* : and <reg>,<const>
*void and_opr(pair<int, unsigned short> realAddress, unsigned constant, char constType*) :and <mem>, <const>
*void or_opr(reg* reg1, reg* reg2)* : or <reg>,<reg>
*void or_opr(reg* reg1, pair<int, unsigned short> realAddress)* : or <reg>,<mem>
*void or_opr(pair<int, unsigned short> realAddress, reg* reg1) :* or <mem>, <reg>
*void or_opr(reg* reg1, unsigned constant, char constType)* : or <reg>,<const>
*void or_opr(pair<int, unsigned short> realAddress, unsigned constant, char constType)* : or <mem>, <const>
*void xor_opr(reg* reg1, reg* reg2)* : xor <reg>,<reg>
*void xor_opr(reg* reg1, pair<int, unsigned short> realAddress)* : xor <reg>,<mem>
*void xor_opr(pair<int, unsigned short> realAddress, reg* reg1) :* xor <mem>, <reg>
*void xor_opr(reg* reg1, unsigned constant, char constType)* : xor <reg>,<const>
*void xor_opr(pair<int, unsigned short> realAddress, unsigned constant, char constType)* : xor <mem>, <const>

The not operation, takes only one parameter and logically negates the operand contents.

*void not_opr(reg* reg1)* :not <reg>
*void not_opr(pair<int, unsigned short> realAddress)* :not <mem>

**SHIFT AND ROTATE OPERATIONS:** SHR and SHL instructions shift the bits in their first operand's contents left and right, padding the resulting empty bit positions with zeros.RCL and RCR instructions rotate the bits of the first operand (destination operand) the number of bit positions specified in the second operand (count operand) and store the result in the destination operand.

*void shift_right_opr(reg* reg1, unsigned constant)* :shr <reg>,<con>
*void shift_right_opr(pair<int, unsigned short> realAddress, unsigned constant)* :shr <mem>,<con>
*void shift_right_opr(reg* reg1, reg* reg2)* :shr <reg>,<cl>
*void shift_right_opr(pair<int, unsigned short> realAddress, reg* reg2)* :shr <mem>,<cl>
*void shift_left_opr(reg* reg1, unsigned constant)* :shl <reg>,<con>
*void shift_left_opr(pair<int, unsigned short> realAddress, unsigned constant)* :shl <mem>,<con>
*void shift_left_opr(reg* reg1, reg* reg2)* :shl <reg>,<cl>
*void shift_left_opr(pair<int, unsigned short> realAddress, reg* reg2) :*shl <mem>,<cl>
*void rotate_left_opr(reg* reg1, unsigned constant)* : rcl <reg>,<con>
*void rotate_left_opr(pair<int, unsigned short> realAddress, unsigned constant)* :rcl <mem>,<con>
*void rotate_left_opr(reg* reg1, reg* reg2)* : rcl <reg>,<cl>
*void rotate_left_opr(pair<int, unsigned short> realAddress, reg* reg2) :* rcl <mem>,<cl>
*void rotate_right_opr(reg* reg1, unsigned constant)* : rcr <reg>,<con>
*void rotate_right_opr(pair<int, unsigned short> realAddress, unsigned constant)* :rcr <mem>,<con>
*void rotate_right_opr(reg* reg1, reg* reg2)* : rcr <reg>,<cl>
*void rotate_right_opr(pair<int, unsigned short> realAddress, reg* reg2) :* rcr <mem>,<cl>

### *Conclusion*

Our project reads and executes the instructions properly, and gives the same output when the input is given to the real HYP86 interpreter. Besides from output, the flag, register and memory assignments are also done properly. Error conditions and edge cases are taken into account. Details can be found in the documentation or in the comments of the code. Our program passes all 25 of the test cases provided, gives the proper outputs. In conclusion, we have properly implemented an interpreter for an assembly language, using C++.