# Assembler – Iteration 1

I started writing my code paying close attention to the Decomposing the Problem section. For this reason, the initial code I typed looked like this:

**def main**(asmfile, out\_format):  
 **with** open(asmfile, "rt") **as** file:  
 text = file.read()  
  
 # Now the text is available  
  
 # 1. PERFORM TEXT NORMALISATION  
  
 normalised\_text = normalise\_text(text)  
 print(normalised\_text)  
  
 # 2. SPLIT DOCUMENT INTO SECTIONS  
 section\_dict = split\_into\_sections(normalised\_text)  
 pprint(section\_dict)  
  
 # 3. DIVIDE LINES AND CONTEXTUALISE  
 config\_dict, instruction\_list = divide\_and\_contextualise(section\_dict)  
  
 # 4. RECORD LABELS/VARIABLES  
 mem\_table = record\_labels\_and\_variables(instruction\_list)  
  
 # 5. CONVERT EACH LINE TO BYTES  
 place\_memory\_addresses(mem\_table, instruction\_list)  
  
 bytecode = b""  
 bytecode += encode\_metadata(config\_dict)  
 bytecode += encode\_instruction\_list(instruction\_list, mem\_table)

This is an outline of the main structure of the program. At present, none of those functions exist, and the purpose of initially writing this was simply to start with an overall view of the program. In order to permit testing, as I was moving through the program I commented out lines calling functions that did not yet exist, and adding in temporary print statements to view the output of the functions.

## Text normalisation

I naturally decided to write my program from beginning to end, so I began with the text normalisation function. At the time of writing, it looks like this:

**def normalise\_text**(text):  
 *"""  
 Removes comments, unnecessary whitespace and empty lines* ***:param*** *text:* ***:return****:  
 """* # 1.1. Split text into lines  
  
 lines = text.split("\n")  
  
 **for** i, line **in** enumerate(lines):  
 # 1.2. For each line, if there is a semicolon, remove everything after the first semicolon  
 lines[i] = lines[i].split(";")[0]  
  
 # 1.3. Strip all whitespace from the start and end of every line  
 lines[i] = lines[i].strip()  
  
 # 1.4. Remove empty lines  
 lines = [line **for** line **in** lines **if** line != ""]  
  
 # 1.5. Remove duplicate whitespace  
 multiple\_whitespace = re.compile(r"\s+")  
 **for** i, line **in** enumerate(lines):  
 lines[i] = multiple\_whitespace.sub(" ", line)  
  
 # 1.6. Put the lines back together  
 normalised\_text = "\n".join(line.strip() **for** line **in** lines)  
  
 **return** normalised\_text

This is quite simple and the purpose of each part is annotated in comments. It was not difficult to write, and ad-hoc testing showed it to be for the most part successful. The only change I had to make was in the second last line, to remove the whitespace from the end of the lines before joining them together. This was to avoid a bug where there were some trailing newlines.

## Split into sections

Once again, this section (the second part of my decomposition) was quite short and dealt only with text manipulation. The code is contained within this function:

**def split\_into\_sections**(text):  
 *"""  
 Takes the normalised text and splits it into its individual sections* ***:param*** *text:* ***:return****:  
 """* # Don't bother with regular expressions for this  
 # Just split where "section." occurs  
 parts = text.split("section.")  
 parts\_with\_titles = [part.split("\n", maxsplit=1) **for** part **in** parts]  
  
 sections\_dict = {}  
 **for** title, \*other **in** parts\_with\_titles:  
 **if** title.strip() == "" **and** len(other) == 0:  
 **continue** sections\_dict[title] = (other[0] **if** len(other) > 0 **else** "").strip()  
  
 **for** section **in** ("meta", "data", "text"):  
 **if** section **not in** sections\_dict.keys():  
 **raise** AssemblyError(-1, "No {} section".format(section))  
  
 **return** sections\_dict

This section also worked quite well from the initial writing. The lines checking for the presence of all three section names was added as a result of Post-Development Test A203, as the code did not initially check that all the parts were present.

## Divide and contextualise

This is the first of the large sections. It could be considered the most important part of the program, because it does all the work of calculating the semantics of the bytecode provided, and everything after this just performs tweaking operations on the objects produced here. This took some time to write, and I will go through it here in blocks approximately similar to the way in which I wrote it. The first block of code is this:

**def divide\_and\_contextualise**(section\_dict: dict):# 4.1. Split the meta section into lines and interpret them  
 # Create the dict based on META\_CONFIG\_DEFAULT  
 config\_dict = META\_CONFIG\_DEFAULT.copy()  
  
 # Split meta section into lines  
 meta\_lines = [x.strip() **for** x **in** section\_dict["meta"].split("\n") **if** x.strip()]  
  
 # Go through the lines and split on an = sign, then act on that  
 **for** line **in** meta\_lines:  
 item, value = line.split("=")  
 config\_dict.update(\*\*{item: value})

This creates the function, which takes the dictionary of sections produced by the previous part of the algorithm. The part of the function shown here then deals with the meta section, turning it into a dictionary of configuration values. Here was where I created the first global dictionary, META\_CONFIG\_DEFAULT, which stores the default values for properties given in the meta section. I create a copy, so that the alterations made do not influence the global copy. Then any changes made in the assembly code are made to this dict. The process is fairly quick. At present, there is actually only one value that the meta section should specify, though this will grow in future iterations. Therefore, the definition of the default dictionary looks like this:

META\_CONFIG\_DEFAULT = {  
 "mem\_amt": 4  
}

This is written at the top of the file, above the functions.

The next part is also not too large. It deals with the data section, and looks like this:

# Config dict done, moving on to the big part: instructions  
instruction\_list = []  
  
# Start with the data section, adding each one as a DataInstruction instance  
data\_lines = [x.strip() **for** x **in** section\_dict["data"].split("\n") **if** x.strip()]  
**for** line **in** data\_lines:  
 # Takes the form name VAR type initial  
 name, type\_and\_initial = [x.strip() **for** x **in** line.split("VAR") **if** x.strip()]  
 dtype, initial = type\_and\_initial.split()  
  
 instruction\_list.append(DataInstruction(len(instruction\_list), name, initial, dtype))

For each line in the data section, break it up. The format of the line is name VAR type initial, and this splits the line up into those parts, then creates a DataInstruction based on it.

### Instruction and DataInstruction

In reality, when it is turned into bytes, this object will just become a MOV instruction that moves the default value into the pre-calculated memory address, so in bytecode it has no unique quality, and is just for convenience here. The DataInstruction class is a subclass of Instruction, as described by the UML diagram in Decomposing the Problem, which is defined like this:

**class Instruction**:  
 **def** \_\_init\_\_(self, instr\_num):  
 self.instruction\_num = instr\_num  
  
 **def get\_bytes\_length**(self):  
 **raise** NotImplementedError("Must only use a subclass of Instruction")  
  
 **def get\_bytes**(self, mem\_table):  
 **raise** NotImplementedError("Must only use a subclass of Instruction")

This is a short definition and direct instances of this class should not be made, because calling either get\_bytes\_length or get\_bytes will throw an error. The subclasses, of which DataInstruction and TextInstruction exist, must override these.

The definition for DataInstruction is quite a bit larger. The class definition and constructor are:

**class DataInstruction**(Instruction):  
 *"""  
 A type of instruction from the data section. It sets a variable. In its implementation, it is a MOV command,  
 moving an immediate value to a certain memory address  
 """* **def** \_\_init\_\_(self, instr\_num, name, value, dtype):  
 super().\_\_init\_\_(instr\_num)  
 self.name = name  
 self.value = value  
 self.data\_type = dtype

The constructor just calls the superclass to give it the instruction number, then files away the three additional arguments without any processing necessary. Following this constructor, I also defined \_\_eq\_\_ and \_\_repr\_\_, which tell Python how to check for equality between two instances and how to print an instance of this function out, respectively. I have, however, omitted them as they are mundane. Very small but less mundane is get\_bytes\_length:

**def get\_bytes\_length**(self):  
 # (Command (MOV) + Opcode data + Memory address = 6) + size of value  
 **return** 6 + self.\_calculate\_valsize()

The comment explains where the 6 comes from: it is the size of the command, opcode byte and memory address, which are the first three parts of the instruction. The final part is the value, and depends on the data type. The method \_calculate\_valsize finds this, and is defined as:

**def \_calculate\_valsize**(self):  
 *"""Calculate the number of bytes in the value"""* **if** self.data\_type == "float":  
 valsize = 4  
 **else**:  
 # An integer type; calculate its size  
 **if** abs(int(self.value)) < 256:  
 valsize = 1  
 **elif** abs(int(self.value)) < 65536:  
 valsize = 2  
 **else**:  
 valsize = 4  
  
 **return** valsize

This checks to see if the data type is a floating point, and if it is then the size of the value must be 4. Otherwise, it will be an integer type. The method then looks at the size of the value to determine how many bytes must be used to store it.

The final, largest, part of the definition for DataInstruction is the get\_bytes function, whose 28 lines look like this:

**def get\_bytes**(self, mem\_table):  
 # The instruction byte  
 instr = struct.pack(">B", OPCODES["MOV\_{}B".format(self.\_calculate\_valsize())])  
  
 valsize = self.\_calculate\_valsize()  
 # The byte to describe the operands and the format string for how to turn the immediate value into binary  
 **if** valsize == 1:  
 operand\_num = b"R" # 0x52  
 val\_fmt\_str = ">B"  
 **elif** valsize == 2:  
 operand\_num = b"S" # 0x53  
 val\_fmt\_str = ">H"  
 **elif** valsize == 4:  
 operand\_num = b"T" # 0x54  
 val\_fmt\_str = ">I"  
 **else**:  
 **raise** ValueError("Illegal value size: {}".format(valsize))  
  
 # The memory address  
 mem\_addr = struct.pack(">I", mem\_table[self.name])  
  
 # The initial value  
 **if** self.data\_type == "float":  
 value\_bytes = struct.pack(">F", float(self.value))  
 **else**:  
 value\_bytes = struct.pack(val\_fmt\_str, int(self.value))  
  
 # Put them all together and return  
 **return** instr + operand\_num + mem\_addr + value\_bytes

The first line in the function immediately creates the opcode byte. It is a MOV command, but the byte to encode it as depends on the size of the data, so this performs that check. The next if statement checks the size of the value, and decides what byte should define the operands based on this. The hexadecimal value is in a comment to the right of each, with the 5-nibble referring to the memory address and 2, 3 or 4 meaning an 8-, 16- or 32-bit operand. The memory address is then packed into a 4-byte integer, along with the value, then all of them added together is returned.

### Processing the text section

After a DataInstruction has been made for each variable definition, then the text section must be processed. The start looks like this:

text\_lines = [x.strip() **for** x **in** section\_dict["text"].split("\n") **if** x.strip()]  
**for** line **in** text\_lines:  
 # Split into basic tokens  
 parts = line.split()

The first action is to split the text section into lines, then to split each line into individual tokens (by splitting it at the spaces). Then, the first of these parts can either be the opcode mnemonic or alternatively a label. Next, check if it is a label:

# See if the first part is in the list of opcodes. If not then it is likely a label  
label = ""  
**if not** parts[0].upper() **in** OPCODE\_NAMES:  
 # The first must be a label; check that the second is the opcode  
 label = parts[0]  
 **del** parts[0] # Remove the label from the list

This records the label if there was one, and removes it from the parts (so we can be confident that parts[0] should be the mnemonic).

# Now we can assume parts[0] is the opcode  
# Next is the type. parts[1] could be a data type or it could not be  
dtype = 0  
**if** parts[1].lower() **in** ("char", "uchar", "short", "ushort", "int", "uint", "float"):  
 dtype = parts[1].lower()  
 **del** parts[1]  
**elif** parts[1].upper() == "1B":  
 dtype = "char"  
**elif** parts[1].upper() == "2B":  
 dtype = "short"  
**elif** parts[1].upper() == "4B":  
 dtype = "int"

The comments explain what this part does: it sees if the part after the mnemonic is a data type. If so then record that data type. If not, an assumption will have to be made later.

mnemonic = parts[0]  
**del** parts[0]

This just writes the mnemonic in its own variable.

# The list of parts should now just consist of the operands, separated by a space  
**if** len(parts) == 0:  
 # No operands  
 operand1 = **None** operand2 = **None  
elif** len(parts) == 1:  
 operand1 = interpret\_operand(parts[0])  
 operand2 = **None  
elif** len(parts) == 2:  
 operand1 = interpret\_operand(parts[0])  
 operand2 = interpret\_operand(parts[1])  
**else**:  
 **raise** Exception("Invalid length of {}".format(len(parts)))

This part works out how many operands are specified, and writes each of them to a variable. It uses the interpret\_operand function to turn the string format and turn it into an Operand object.

### Interpreting operands

The function to interpret an operand is itself fairly large. The start of it looks like this:

string = string.strip() # Just to make sure  
  
**if not** string:  
 **raise** ValueError("Cannot interpret an empty operand")  
  
# Is it a register name?  
**if** string.lower() **in** REGISTERS.keys():  
 # Yes it is  
 **return** RegisterOperand(string)

This removes whitespace from the ends of the string just in case, then raises a ValueError if it is empty (since something must have gone wrong). It then performs a check to see if the text is in the list of register names. REGISTERS is another global dictionary that maps register names to their numerical versions, but I will not print it here as it is quite large and contains the same information as the Bytecode section of Decomposing the Problem.

# Is it an immediate value?  
**try**:  
 val = int(string)  
 **return** ImmediateOperand(val)  
**except** ValueError:  
 # it could be an immediate float  
 **try**:  
 val = float(string)  
 **return** ImmediateOperand(val)  
 **except** ValueError:  
 **pass** # Not an immediate value

The next check is to see if it is an immediate value (i.e. a number). Initially during writing, I forgot the possibility of the number being a floating point, so this was added in later.

# Is it an address label/variable?  
**if** string.isalnum():  
 **return** AddressOperand(string)

The next check is a short one and just assumes that if the operand is alphanumeric then it must be a reference to a label/variable.

# Is it an arithmetic expression?  
**if** string[0] == "[" **and** string[-1] == "]":  
 **return** ArithmeticOperand(string[1: -1])

This final check sees if it is an arithmetic expression, which would be designated by the presence of square brackets. Originally, I wrote string[1: -2] but this cut off the last character inside the brackets, whereas the substring operation is intended only to remove the brackets.

# If it was none of those then it is invalid  
**raise** ValueError("Invalid operand: {}".format(string))

The final block raises an error if none of these were found. Its format must be invalid.

### Operand objects

Operand objects are objects that describe operands. There are four types: immediate, register, memory and arithmetic. All are represented by a subclass of Operand, whose definition is this:

**class Operand**:  
 **def** \_\_init\_\_(self):  
 self.\_bit\_designation = -1  
 self.\_required\_length = -1  
  
 **def get\_bit\_designation**(self):  
 **return** self.\_bit\_designation  
  
 **def get\_required\_length**(self):  
 **return** self.\_required\_length  
  
 **def get\_bytes**(self):  
 **raise** NotImplementedError("Must use a subclass of Operand")

This should not have instances created directly. Of the three methods, only one will raise an error if called, but in its basic state here it is useless nonetheless. The two attributes declared here are \_bit\_designation, which will be used to store the number describing this type of operand (a number from 0-10, whose meanings are described in Decomposing the Problem), and \_required\_length, which records the number of bytes the operand will take up. These attributes are used to prevent the subclasses from needing to override the corresponding functions.

The first subclass of this is RegisterOperand:

**class RegisterOperand**(Operand):  
 **def** \_\_init\_\_(self, regname):  
 super().\_\_init\_\_()  
 self.name = regname.lower()  
 self.numerical = REGISTERS[regname.lower()]  
  
 self.\_bit\_designation = 1  
 self.\_required\_length = 1

# ... Omitted some internal python methods ...  
  
 **def get\_bytes**(self):  
 # Basically turn the numerical register number into a byte  
 **return** struct.pack(">B", self.numerical)

Again, I have omitted some objects that tell Python how to perform certain processes on the object. Importantly, the constructor stores the register name and also looks up the numerical format for this register, as well as setting the pieces of information required by Operand. The get\_bytes method then simply packs the numerical format into a byte and returns it.

**class ImmediateOperand**(Operand):  
 **def** \_\_init\_\_(self, value):  
 super().\_\_init\_\_()  
 **try**:  
 self.value = int(value)  
 **except** ValueError:  
 self.value = float(value)  
  
 **if** isinstance(self.value, float) **or** self.value < -32768 **or** self.value > 65536:  
 self.size = 4  
 self.\_bit\_designation = 4  
 **elif** self.value < -128 **or** self.value > 255:  
 self.size = 2  
 self.\_bit\_designation = 3  
 **else**:  
 self.size = 1  
 self.\_bit\_designation = 2  
  
 self.\_required\_length = self.size

**return** str(self.value)

# ... Omitted some internal python methods ...  
  
 **def \_get\_value\_format\_string**(self):  
 # Find the formatting string to use to turn it into bytes  
 **if** isinstance(self.value, float):  
 **return** ">f"  
 **elif** self.value < -32768:  
 **return** ">i"  
 **elif** self.value < -128:  
 **return** ">h"  
 **elif** self.value < 0:  
 **return** ">b"  
 **elif** self.value < 256:  
 **return** ">B"  
 **elif** self.value < 65536:  
 **return** ">H"  
 **else**:  
 **return** ">I"  
  
 **def get\_bytes**(self):  
 **return** struct.pack(self.\_get\_value\_format\_string(), self.value)

The ImmediateOperand class is more sophisticated. The constructor takes a value and then goes through an algorithm to determine the data type is has. \_get\_value\_format\_string is used by the once again simple get\_bytes() to tell the Python struct module what type of data to write this as.

**class AddressOperand**(Operand):  
 **def** \_\_init\_\_(self, address):  
 super().\_\_init\_\_()  
 self.addr = address  
 self.\_bit\_designation = 5  
 self.\_required\_length = 4  
  
# ... Omitted some internal python methods ...  
  
 **def get\_bytes**(self):  
 **return** struct.pack(">I", self.addr)

The AddressOperand type is once again quite simple, just storing the given address in an attribute, writing down its size and bit designation, and having a simple method to make the bytes. One point of note here is that when it is initially made, the address attribute will have a string object in it, which would fail on conversion to bytes. It relies on a later part of the algorithm replacing this with the correct integer memory address.

By far the most complex type is the arithmetic operand. The structure of this type of operand is intrinsically more complicated than the others. This necessitates the following, admittedly quite horrific, regular expressions.

**class ArithmeticOperand**(Operand):  
 type\_6 = r"(?P<a>[a-zA-Z0-9]+)" # a  
 type\_7 = r"(?P<a>[a-zA-Z0-9]+)\\*(?P<b>[a-zA-Z0-9]+)" # a\*b  
 type\_8 = r"(?P<a>[a-zA-Z0-9]+)\+(?P<b>[a-zA-Z0-9]+)" # a+b  
 type\_9 = r"(?P<a>[a-zA-Z0-9]+)\\*(?P<b>[a-zA-Z0-9]+)\+(?P<c>[a-zA-Z0-9]+)" # a\*b+c  
 type\_10 = r"(?P<a>[a-zA-Z0-9]+)\+(?P<b>[a-zA-Z0-9]+)\\*(?P<c>[a-zA-Z0-9]+)" # a+b\*c

I did not write each of these independently, but produced the first and then copied and pasted to form the rest. The first type, labelled type\_6 (because 6 is the number that will go into the operand byte), just represents any alphanumeric sequence with one or more characters. This will mean an integer value or a variable name. This regular expression also names this as group a, which will help processing later. Also, note the comments to the right hand side which show what each type represents.

The other types are simply repeats of this sequence with the \* and + characters between them in all the necessary permutations. Unfortunately, the ugly code only starts with those regular expressions. The uncomfortably large and repetitive constructor looks like this:

**def** \_\_init\_\_(self, asm\_str):  
 super().\_\_init\_\_()  
 self.asm\_str = asm\_str  
  
 **if not** asm\_str.strip():  
 **raise** ValueError("Cannot process empty operand")  
  
 # See which type of operand this is and set everything accordingly  
 **if** re.search(self.type\_6, asm\_str) **is not None**:  
 m = re.match(self.type\_6, asm\_str)  
 self.\_bit\_designation = 6  
 self.\_required\_length = 1  
 self.a = m.group("a")  
 self.b = **None**; self.c = **None  
 elif** re.search(self.type\_7, asm\_str) **is not None**:  
 m = re.match(self.type\_7, asm\_str)  
 self.\_bit\_designation = 7  
 self.\_required\_length = 2  
 self.a = m.group("a")  
 self.b = m.group("b")  
 self.c = **None  
 elif** re.search(self.type\_8, asm\_str) **is not None**:  
 m = re.match(self.type\_8, asm\_str)  
 self.\_bit\_designation = 8  
 self.\_required\_length = 2  
 self.a = m.group("a")  
 self.b = m.group("b")  
 self.c = **None  
 elif** re.search(self.type\_9, asm\_str) **is not None**:  
 m = re.match(self.type\_9, asm\_str)  
 self.\_bit\_designation = 9  
 self.\_required\_length = 3  
 self.a = m.group("a")  
 self.b = m.group("b")  
 self.c = m.group("c")  
 **elif** re.search(self.type\_10, asm\_str) **is not None**:  
 m = re.match(self.type\_10, asm\_str)  
 self.\_bit\_designation = 10  
 self.\_required\_length = 3  
 self.a = m.group("a")  
 self.b = m.group("b")  
 self.c = m.group("c")  
 **else**:  
 **raise** AssemblyError(-1, "Incorrect format for arithmetic operand: {}".format(asm\_str))

This stores how this operand appears as a string, then moves on to working out which, if any, of the sequences corresponds to the given string. While some simplification and refactoring is most likely possible, the subtle differences between each block of code prevent a simple solution using a utility function. The sequence of if/elif statements checks for a match against each one of the sequences. If it finds a match, it pulls out each part of it and assigns them to attributes, giving a default of None if that part is not needed for this type. It also specifies the number of bytes necessary and the bit designation to use. If none of the types matches the string, then an error is raised.

**def \_interpret\_value**(self, value) -> int:  
 # The values in arithmetic expressions can be immediate or a register name. Either way it has to be encoded.  
 **if** isinstance(value, str):  
 # A register  
 **try**:  
 **return** REGISTERS[value]  
 **except** KeyError **as** err:  
 **raise** ValueError("Invalid register name: {}".format(value)) **from** err  
  
 # It is just an immediate value  
 numval = int(value)  
 **if** numval **in** (2, 4, 8):  
 **return** numval  
 **else**:  
 **raise** ValueError("Only 2, 4 and 8 are permitted for multiplication in arithmetic operands")

This method turns the provided value into an integer format. Register names have corresponding integer values, which are returned if this is one of those. Alternatively, if it is the number 2, 4 or 8 (the only ones permitted for multiplication in this way), then that is returned. This complies with the Decomposing the Problem bytecode definition and unambiguously stores both registers and immediate values in a single byte.

**def get\_bytes**(self):  
 bytes\_ = b""  
 **for** val **in** (self.a, self.b, self.c):  
 **if** val **is not None**:  
 bytes\_ += struct.pack(">B", self.\_interpret\_value(val))  
  
 **assert** len(bytes\_) == self.\_required\_length, "Required length and calculated byte length do not match"  
 **return** bytes\_

This is once again quite a simple function. It turns each of the a, b and c values into their numeric format, then turns each of them into a byte (if they are present). This is then returned.

### Compiling the instruction list

# We now have both operands; that's everything  
 instruction\_list.append(TextInstruction(instr\_num=len(instruction\_list),  
 opcode=mnemonic,  
 dtype=dtype,  
 label=label,  
 op1=operand1,  
 op2=operand2))  
   
**return** config\_dict, instruction\_list

At the end of the divide\_and\_contextualise function is these lines. Still within the loop is a command to make a TextInstruction object and add it to the list. Then, when the loop finishes, the configuration dictionary and instruction list are returned.

### TextInstruction

More complex than the DataInstruction objects made by the data section is the TextInstruction object. Its constructor is:

**class TextInstruction**(Instruction):  
 **def** \_\_init\_\_(self, instr\_num, opcode: str, dtype: str, op1, op2, label=""):  
 super().\_\_init\_\_(instr\_num)  
 self.opcode\_mnemonic = opcode  
 self.data\_type = dtype  
 self.operand1 = op1  
 self.operand2 = op2  
 self.label = label  
  
 **if** dtype == 0:  
 # None was specified, so try to work it out based on the operands  
 op1\_size = op1.get\_required\_length() **if** op1 **is not None else** 0  
 op2\_size = op2.get\_required\_length() **if** op2 **is not None else** 0  
  
 max\_op\_size = max(op1\_size, op2\_size)  
  
 **if** max\_op\_size == 1:  
 self.data\_type = "char"  
 **elif** max\_op\_size == 2:  
 self.data\_type = "short"  
 **elif** max\_op\_size == 4:  
 self.data\_type = "int"

This takes in all of the possible pieces of information about an instruction in the text section. It simply writes down the opcode mnemonic, data type, operands and label. The rest of the code is used to specify a data type if one was not given, based on the operands. I have omitted a few internal python methods, so the next method is:

**def get\_bytes\_length**(self):  
 # (Opcode byte + operand byte = 2) + operand 1 + operand 2  
 length = 2  
 **if** self.operand1 **is not None**:  
 length += self.operand1.get\_required\_length()  
  
 **if** self.operand2 **is not None**:  
 length += self.operand2.get\_required\_length()  
  
 **return** length

This starts with an assumption of length 2, 1 for the opcode and 1 for the registers. It then checks each operand, makes sure it is not None, and adds the length of that operand to the length of this.

The get\_bytes function is once again quite complicated, but it starts like this:

**def get\_bytes**(self, mem\_table):  
 # First find the opcode byte  
  
 # Check the opcode is valid  
 **if** self.opcode\_mnemonic **not in** OPCODE\_NAMES:  
 **raise** ValueError("Unsupported opcode mnemonic: {}".format(self.opcode\_mnemonic))  
  
 # Is this the type of opcode that has no subtypes?  
 **if** self.opcode\_mnemonic **in** OPCODES.keys():  
 opcode\_num = OPCODES[self.opcode\_mnemonic]

This will find the numeric form of the opcode if it is a command that does not depend on the data type. If a data type or size is necessary for the opcode, then the next part of the code finds that:

# Is it based on a data type?  
**elif** (self.opcode\_mnemonic + "\_" + self.data\_type) **in** OPCODES.keys():  
 opcode\_num = OPCODES[self.opcode\_mnemonic + "\_" + self.data\_type]  
  
# If neither of those, then it might be size based  
**elif** (self.data\_type **in** ("char", "uchar")) **and** (self.opcode\_mnemonic + "\_1B" **in** OPCODES.keys()):  
 opcode\_num = OPCODES[self.opcode\_mnemonic + "\_1B"]  
**elif** (self.data\_type **in** ("short", "ushort")) **and** (self.opcode\_mnemonic + "\_2B" **in** OPCODES.keys()):  
 opcode\_num = OPCODES[self.opcode\_mnemonic + "\_2B"]  
**elif** (self.data\_type **in** ("int", "uint", "float")) **and** (self.opcode\_mnemonic + "\_4B" **in** OPCODES.keys()):  
 opcode\_num = OPCODES[self.opcode\_mnemonic + "\_4B"]  
  
**else**:  
 **raise** ValueError("Mismatch between opcode {} and data type {}".format(self.opcode\_mnemonic, self.data\_type))

This checks in the dictionary of registers to see if the mnemonic + a data type has a number, and remembers that so. If not, it checks the possible size-based opcodes. If none of them work, then just raise an error. By the end of this section, if an error has not been thrown, then opcode\_num will contain the numeric form of the opcode, ready for turning into binary. It is turned into a byte immediately:

# Turn the opcode number into an opcode byte  
opcode\_byte = struct.pack(">B", opcode\_num)

The next job is to construct the operand byte, i.e. the description of the operands. The code to find it is this:

# Next, find the operand byte  
operand\_num = self.operand1.get\_bit\_designation() **if** self.operand1 **is not None else** 0  
operand\_num = operand\_num << 4 # Shift the bits to the left to make space for the second  
operand\_num += self.operand2.get\_bit\_designation() **if** self.operand2 **is not None else** 0  
operand\_byte = struct.pack(“>B”, operand\_num)

This asks each of the operands for their bit designations (which fit in a nibble size), or default to 0 if there is no operand. It gets the first operand, a 4-bit number, then shifts it 4 bits to the left and adds the bit designation for the second operand. Given this number, it then turns it into a byte.

# Get the operand bytes  
op1\_bytes = self.operand1.get\_bytes() **if** self.operand1 **is not None else** b""  
op2\_bytes = self.operand2.get\_bytes() **if** self.operand2 **is not None else** b""

The next job is to ask the operands to turn themselves into bytes. It checks to make sure they are present, and if not then simply adds nothing, but if they do exist then it stores their byte format in a variable.

# Add them together  
instr\_bytes = opcode\_byte + operand\_byte + op1\_bytes + op2\_bytes  
  
**assert** len(instr\_bytes) == self.get\_bytes\_length()  
  
**return** instr\_bytes

The final piece of this function is to add all of the parts together, check that it is the length that was expected (this is absolutely necessary as if the length it returns is different to the length the object stated it had, memory addresses will be wrong). The bytes are then returned.

## Record labels and variables

This stage is far smaller than the previous. It goes through, records all of the variables (with their data types and how far into the data section of memory they will be) and all of the labels (with the number of the instruction it refers to). This is then all turned into a final memory address table.

The start of the function is:

**def record\_labels\_and\_variables**(instruction\_list):  
 # Create the empty tables  
 label\_table = {}  
 var\_table = {}  
 mem\_table = {}

This creates the function and makes the empty dictionaries. The first part of the function, making the label and variable tables, is this loop:

# Loop through the instructions  
for instruction in instruction\_list:  
 # If the instruction is a DataInstruction, add it to the variable table with its relative position  
 if isinstance(instruction, DataInstruction):  
 var\_table[instruction.name] = (calculate\_var\_table\_size(var\_table), instruction.data\_type)  
  
 # If the instruction is a TextInstruction and has a label, add it to the label table  
 elif isinstance(instruction, TextInstruction):  
 if instruction.label != "":  
 label\_table[instruction.label] = instruction.instruction\_num  
  
 # If it was neither of these types, what's it doing here?  
 else:  
 raise ValueError("Item in instruction list is neither \  
 DataInstruction nor TextInstruction: {}".format(instruction))

It runs through each instruction in the instruction list. If it is a data instruction, then it must be creating a variable. We store the starting byte and the data type. The starting byte is just the current size of the variable table. For example, if all that is in the variable table is a single 4-byte integer, then this will take up the spaces 0-3, so the next variable made will start at address 4. The code for calculate\_var\_table\_size is quite simple:

def calculate\_var\_table\_size(var\_table):  
 size = 0  
 for (var, (\_, dtype)) in var\_table.items():  
 size += DTYPE\_META[dtype].size  
  
 return size

It loops through the variable table, and adds up the total length of all the data types contained within it. The global dictionary DTYPE\_META contains instances of DataTypeMetadata and the code making it is:

DataTypeMetadata = namedtuple("DataTypeMetadata", ["size"])  
  
DTYPE\_META = {  
 "char": DataTypeMetadata(1),  
 "uchar": DataTypeMetadata(1),  
 "short": DataTypeMetadata(2),  
 "ushort": DataTypeMetadata(2),  
 "int": DataTypeMetadata(4),  
 "uint": DataTypeMetadata(4),  
 "float": DataTypeMetadata(4),  
 "1B": DataTypeMetadata(1),  
 "2B": DataTypeMetadata(2),  
 "4B": DataTypeMetadata(4)  
}

This stores a record of every data type and how big it is, which is used to calculate the size of the variable table.

The next part of the loop checks to see if the instruction is a TextInstruction. The relevant part here is whether the instruction has a label. If not, it can be ignored, but if so, then record an entry in the table matching the label name to the instruction number. This also checks and throws an error if somehow something other than an Instruction ended up in the instruction list.

After the loop has finished, both of these tables must be combined to make a memory address table. The first step is to calculate the total size of the text section:

# Calculate the total size of the text section  
total\_text\_section\_size = sum(instr.get\_bytes\_length() for instr in instruction\_list)

This is important because currently the variable table is storing how many bytes into the data section of memory each variable is. However, to turn this into an actual memory address (since the data section is arranged after the text section), we have to add the total size of the text section.

# Add all the variables to the memory address table  
for name, (offset, \_) in var\_table.items():  
 mem\_table[name] = total\_text\_section\_size + offset  
  
# Add all the labels to the memory address table  
for name, instr\_num in label\_table.items():  
 mem\_table[name] = calculate\_instr\_start\_address(instr\_num, instruction\_list)

The comments above explain what each of these two blocks of code do. The last line contains a reference to another utility function, which is defined as:

def calculate\_instr\_start\_address(instr\_num, instruction\_list):  
 total = 0  
 for instr in instruction\_list[:instr\_num]:  
 total += instr.get\_bytes\_length()  
 return total

This is similar to calculate\_var\_table\_size, but it works through the instruction list and cuts off just before the desired index. This is because the start address of an instruction will be equal to the sum of the lengths of all those before, for similar reasons to how variable offset was calculated.

# That's done, so return it  
return mem\_table

The memory table is complete.

## Convert each line to bytes

This is the last step of the process. Almost everything is now in place. The first action to be taken is to go through the instructions and replace all references to variables and labels with their memory addresses that have just been calculated, as instigated by the line in main:

place\_memory\_addresses(mem\_table, instruction\_list)

This function is responsible for the conversion. Its definition is:

def place\_memory\_addresses(mem\_table, instruction\_list):  
 for instr in instruction\_list:  
 if isinstance(instr, DataInstruction):  
 continue # These don't have operands  
  
 if isinstance(instr.operand1, AddressOperand) and isinstance(instr.operand1.addr, str):  
 instr.operand1 = AddressOperand(mem\_table[instr.operand1.addr])  
  
 if isinstance(instr.operand2, AddressOperand) and isinstance(instr.operand2.addr, str):  
 instr.operand2 = AddressOperand(mem\_table[instr.operand2.addr])

For each instruction, it first checks that this is not a data instruction, because these do not have operands that could possibly be replaced. Then, for each of the first and second operands, it checks to see if the operand is an AddressOperand (and this also doubles as a check for the operand's existence). Then it checks to see if the operand's value is a string, because if it is then it needs replacing with a memory address, so the function does this.

The next lines of the main function are arguably the climax of the program:

bytecode = b""  
bytecode += encode\_metadata(config\_dict)  
bytecode += encode\_instruction\_list(instruction\_list, mem\_table)

These are the lines that do the final work of making the bytecode, now that everything is in place. The first function here encodes the metadata from the configuration dictionary, and looks like this:

def encode\_metadata(config\_dict):  
 encoded = b""  
 for key, value in config\_dict.items():  
 encoded += key.encode() + b"=" + str(value).encode() + b"&"  
 encoded += b"\x00\x00\x00\x00"  
  
 return encoded

It loops through the config dictionary, and for each key/value pair it adds them in the form "key=value&" to the bytecode. At the end it adds 4 null bytes, which will never be a sequence to appear in the metadata just encoded, so it can be used as a buffer to reliably separate the metadata from the instructions. The next function is this:

def encode\_instruction\_list(instruction\_list, memory\_table):  
 encoded = b""  
 for instr in instruction\_list:  
 b = instr.get\_bytes(memory\_table)  
 encoded += b  
 return encoded

Due to my use of Object Oriented Programming, this is extremely short and offloads all of the difficult work onto the instructions themselves. It loops through the instruction list and tells each to turn itself into bytes, giving it the memory table in case that should be necessary. It totals all of these together and returns them.

The bytecode has now been created in full. Of course, it must now be output, and this can happen in four ways, and is controlled by an argument to the function:

# Output it as the user wanted  
if out\_format == "hex":  
 print\_bytes\_as\_hex(bytecode, 16)  
elif out\_format == "binstr":  
 print(bytecode)  
elif out\_format == "return":  
 return bytecode  
elif out\_format == "file":  
 fname = input("Name of output file: ")  
 with open(fname, "wb") as file:  
 file.write(bytecode)

hex means the program will convert the bytecode to a hexadecimal format and print it. The function for this is:

def print\_bytes\_as\_hex(bytes\_, rowlen):  
 for i, byte in enumerate(bytes\_):  
 print(format(byte, ">02X") + " ", end="")  
 if (i+1) % rowlen == 0:  
 print()

This turns each byte into a hexadecimal format and prints it with a space to act as a divider, with no newline at the end. It only prints the newline on a regular number of bytes, so the output takes a table-style format. The code in main sets this to 16.

The next output format is binstr. This just prints the object as Python does, so it will be in the format of b'Bytecode here'. It can return the bytecode, and this feature is intended for use if another module imports this one and uses the bytecode, since this would be invisible when the program is run directly. The final option is to output the data into a file, for which the program will ask for a filename and then write the output.

This has been the first iteration of the assembler. It will likely be the largest, as it was responsible for setting up the backbone of the program. Once completed and some ad-hoc testing (as has been mentioned) was done, it correctly takes assembly code and outputs bytecode.