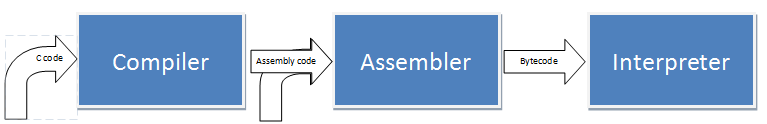
Decomposing the problem

# Overall structure

The standard build process for a C program, which my project is trying to emulate, looks like this:

* The C code is the input
* The pre-processor makes simple text replacements based on “preprocessor directives” (lines beginning with a # symbol)
* The compiler takes a pure C code file and compiles it into object code, which is almost the same as machine code but not yet executable as it represents only one C file, and it has not yet been linked to any other files (and therefore cannot find the functions they provide)
* The linker takes all of the object files produced by the compiler for the project and links them together into one functional executable
* The machine code is the output, usually in the form of a .exe file (for windows; .elf for linux though it is usually omitted).

In my compiler, I will simplify this. It would be too complicated, and beyond the reach of A level, to include the step of the linker. Instead, I will explicitly limit programs to one C file (so no linking of files is necessary) and have the compiler produce assembly code, which is then turned into “machine code” (i.e. bytecode) by the assembler later on. This method is illustrated below:



This document’s first level of decomposition is therefore based on these parts. I will start with a section called “Specification”, which defines the specification for the C code and assembly language, as well as the instruction set for the machine code and notes on memory function. I will then have a section on the compiler, one on the assembler and one on the interpreter, each of which will discuss the implementation and further breakdown of their respective parts.

I will also make note of my intention regarding iterations; in the first iteration of the programs, I will simply make them function as their native items. Then, I will return and add in the GUI and educational elements. Finally, if it is feasible, I will go back over and add in more complex features.

# Specification

In this section I will define the types of C constructs available, then describe the assembly language syntax available, and then the instruction set, bytecode format and interpreter architecture. It should be noted that this is an initial specification. When designing the memory layout and other fundamentals, I will be keeping in mind the possibility that more complex structures such as functions will be added at a later date

## C specification

As this will not be a full C compiler, I will not be able to keep it exactly to the C standard. Instead, I will describe here my specification for the subset of C I will be using to compile. I do intend this to be a strict subset – that is, any code my compiler accepts should be valid C code that any other compiler can use. I have also decided to remove a large part of the language whose implementation is much more complex, such as structures, pointers and functions. All of these are not part of the simple flow of the program, but require much more complex memory and stack management. I will likely implement these in a later iteration, but at present the full list of supported features is below.

* Output (using printf) and input (using getchar/getline/scanf) – Because of the lack of an OS-run environment or custom build command line to print to, this will require being written natively and will need a custom opcode. However, it is essential and so must be included, even if it will work differently to in real programs.
* int main() – While functions in general will not be implemented, in order to be C-compatible, this specific function must be where the code is written.
* #include statements – At least in the first iteration, this will be the only preprocessor directive, but it is necessary to import printf and other native functions.
* Variables and the basic types (char, short, int, float) – This is a very important part, as these five types of data (an 8 bit int/character, a 16-bit integer, a 32-bit integer, and a 32 bit float respectively) are fundamental parts of C that are required. The integer types will have signed and unsigned options. They will each be understood natively and have their own bytecode representations, and the assembler and interpreter will understand the amount of memory to allocate for each.
* Arithmetic operations – These are necessary for any reasonable program to do any computations. The operations are below:

|  |  |
| --- | --- |
| Add | + |
| Subtract | - |
| Multiply | \* |
| Divide | / |
| Modulus | % |

* Bit operations – These are syntactically the same as arithmetic operations, and I have chosen to add them both because they can be quite useful in low level programming, and also because if arithmetic expression parsing is built in already, these operations are trivial to add.

|  |  |
| --- | --- |
| AND | & |
| OR | | |
| XOR | ^ |
| One’s complement/NOT | ~ |
| Left shift | << |
| Right shift | >> |

* Compound mathematical expressions – The compiler can understand the order of operations (including bitwise operations) and brackets, to understand any nested level of mathematical operations, including using literals.
* Comparison operations – These are operations which technically work in the same way as arithmetic operations, but where “a+b” evaluates to the sum of a and b, “7==7” evaluating as 1, meaning true. If false, it would evaluate to 0. The following comparison operations are available:

|  |  |
| --- | --- |
| Equals | == |
| Does not equal | != |
| Less than | < |
| Greater than | > |
| Less than or equal to | <= |
| Greater than or equal to | >= |

* Boolean logical operators – These are necessary for a complex program, and so will be included:

|  |  |
| --- | --- |
| A and B | A && B |
| A or B | A || B |
| Not A | !A |

* Loops – Both conditional iteration (while loops) and count-controlled iteration (for loops) will be available, syntactically exactly the same as they appear in C. To allow full functionality, break and continue statements will be included as well.
* Pointers – In order to create complex structures, as well as allowing arrays to work as they do in C, pointers must be available. They will work syntactically the same as in C, like:

|  |  |
| --- | --- |
| Pointer to a type | type\* |
| Get the value stored at pointer A | \*A |
| Get a pointer to the value A | &A |

* Arrays – Syntactically exactly the same as in C, arrays will have to be implemented to allow complex programs (or even in fact to make it Turing complete). Ideally, to be realistic, the assembler and interpreter should not have to deal with arrays, since they are a higher level construct than machine code. Therefore, the compiler will have to make sure that the assembly produced handles its memory allocation properly and will treat arrays correctly.
* Strings – In C, strings are just syntactic sugar for an array of chars terminated by a null.
* Memory management functions – The following functions are necessary in C to perform a number of tasks, and since the aim of this project is to be educational, it would be useless to remove dynamic memory allocation from this language subset.
  + sizeof(type) – Returns the number of bytes that an object of this type takes up
  + malloc(bytes) – Returns a pointer to the start of a new chunk of memory on the heap of the size specified (or a NULL pointer if it failed)
  + free(pointer) – Frees up the memory block pointed to by pointer. How it knows how much memory to access is discussed later.

## Assembly language

The basic syntax for the assembly language used is based on the Intel syntax. An assembly file will be split into three areas. These are the meta section, the data section and the text section. The meta section fills in a few pieces of metadata such as the size of the program’s address space. The data section is used for variables, and the text section is where instructions live (this will be the bulk of a program). An assembly file will therefore look like this:

section.meta

; Config options

section.data

; Constant definitions

section.text

; Commands

**Meta section commands**

The contents of the meta section are not actually commands that will be compiled but are metadata to place in the bytecode file’s header about the environment.

Each line in the meta section will take the following form:

item=value

The meta section is effectively a type of internal config file, giving key/value pairs to give certain instructions to the assembler. The possible “item” options are:

|  |  |  |
| --- | --- | --- |
| Name | Default | Description |
| mem\_amt | 4 | The number of Kibibytes that the interpreter should |
|  |  |  |

**Data section commands**

A line in the data section will consist of the following:

name VAR type initial

This is a simple standard. All of the variables here will have places designated for them in memory and this will be able to be accessed symbolically in the assembly file by using the variable’s name.

**Text section commands**

opcode [type] operand1[, operand2...] ; Comment

The type can be any possible type the system recognises. It tells the command how many bytes to pull in the operation, and in the case of arithmetic operations it will give information about how to interpret the number. The recognised types are the following:

|  |  |  |
| --- | --- | --- |
| Name | Num bytes | Description |
| Char | 1 | A one-byte signed (two’s complement) integer, between -128 and +127 |
| Uchar | 1 | A one-bite unsigned integer, between 0 and 255 |
| Short | 2 | A two-byte signed (two’s complement) integer, in the range of -32,768 and +32,767 |
| ushort | 2 | A two-byte unsigned integer, between 0 and 65536 |
| int | 4 | A four-byte signed (two’s complement) integer, between -2,147,483,648 and 2,147,483,647 |
| uint | 4 | A four-byte unsigned integer, between 0 and 4,294,967,296 |
| Float | 4 | A four-byte floating point number, according to IEEE 754 |

In the case of some commands, a type is not required but a size is. This can be “1B”, “2B” or “4B”, meaning the corresponding number of bytes.

The opcode is the only item above which is required (unless the line is just whitespace), and the possible values for it are listed in the specification table below. How many opcodes there are, and what they represent, depends on the opcode. The assembler can infer what type of operand is a literal, what is a register and what is a memory address, and will make sure to produce the appropriate bytecode representation. Line comments are possible, beginning with a semicolon. Everything after the semicolon will be stripped and ignored.

### Operand syntax

Operands can be registers, memory addresses or literals. Each of these has its own syntax, and there is some additional syntax for other techniques that may be required. Below is a list of things that can be used for an operand.

**Literal** – A literal is simply an integer. It looks like:

5

There is no additional syntax necessary to specify a literal. Character literals are also possible, using single quotes, such as:

‘a’

However, since memory is entirely made of numbers, this is simply for convenience and will be treated as a number. The example above will generate exactly the same bytecode as 97 would, as 97 is the ASCII character code for a lower-case letter a.

**Register** – The syntax for a register is to simply say its name. For example:

eax

There is a full list of registers in the “Interpreter specification” section. A register name can be used as a source or destination, or any other operand in an instruction.

**Memory location** – A memory location has to be specified by the register currently storing its address. If the address of a location in memory is currently in the ebx register, then this:

[ebx]

means “the value stored at the location given in ebx”. This can also be made more complex, with arithmetic expressions inside the brackets. For example, [eax+ebx] would be “add together the values in eax and ebx and get the value stored at that address”. One of those can be multiplied by 2, 4 or 8, like [eax+4\*ebx]. This might be useful, for example, if eax contains the starting memory address of a list of 4-byte integers, and ebx contains the index of the item you want to access.

The following table is a comprehensive list of opcodes and syntax, with the instruction’s name and a description.

|  |  |  |
| --- | --- | --- |
| Operation name | Syntax | Description |
| **Data management** | | |
| Move | MOV [type] dest, src | Moves the content of src to dest. dest can be a register or memory address, and src can be those or a literal. src will be unchanged. |
| Load effective address | LEA dest, src | src should be a memory address, and lea will place this address (not the address’s contents) into dest. |
| **Arithmetic** | | |
| Add | ADD [type] dest, src | Performs the operation “dest = dest + src”, assuming dest and src to be integers unless specified otherwise |
| Subtract | SUB [type] dest, src | Performs the operation “dest = dest - src”, assuming dest and src to be integers unless specified otherwise |
| Multiply | MUL [type] dest, src | Performs the operation “dest = dest \* src”, assuming dest and src to be integers unless specified otherwise |
| Integer division | IDIV [type] dest, src | Performs the operation “dest = int(dest / src)”, assuming dest and src to be integers unless specified otherwise |
| Integer modulus | MOD [type] dest, src | Puts the remainder of dividing dest by src into dest |
| Exact division | EDIV [type] dest, src | Divides dest by src yielding a float. |
| Bitwise AND | AND [type] dest, src | Performs a binary AND operation on dest and src, storing the result in dest. type tells the command the size to use. |
| Bitwise OR | OR [type] dest, src | Performs a binary OR operation on dest and src, storing the result in dest. type tells the command the size to use. |
| Bitwise XOR | XOR [type] dest, src | Performs a binary XOR operation on dest and src, storing the result in dest. type tells the command the size to use. |
| Bitwise NOT | NOT [type] val | Flips every bit in val. type tells the command the size to use. |
| Left shift | LSH [type] val, num | Shifts the bits in val to the left by num places, losing bits to the left and creating zeroes to the right. type tells the command the size to use. |
| Right shift | RSH [type] val, num | Shifts the bits in val to the right by num places, losing bits to the right and creating zeroes to the left. type tells the command the size to use. |
| **Control flow** | | |
| Halt | HLT | Stops the program. |
| Compare | CMP [type] v1, v2 | Performs a comparison: performs “v1 - v2” and remembers information about the result for a jump expression to use. |
| Jump always | JMP label | Regardless of any comparisons, jumps to the given label. Arguments for jump instructions use textual labels as mnemonics which will be replaced with memory addresses by the assembler. |
| Jump if equal | JE label | Jumps to label if the comparison expression found v1 and v2 to be equal |
| Jump if not equal | JNE label | Jumps to label if the comparison expression found that v1 != v2 |
| Jump if less than | JLT label | Jumps to label if the comparison expression found that v1 < v2 |
| Jump if less than or equal to | JLE label | Jumps to label if the comparison expression found that v1 <= v2 |
| Jump if greater than | JGT label | Jumps to label if the comparison expression found that v1 > v2 |
| Jump if greater than or equal to | JGE label | Jumps to label if the comparison expression found that v1 >= v2 |

## CPU, bytecode and memory

The interpreter will take bytecode created by the assembler, and run it as designed. To accomplish this, the interpreter has a memory region, containing a large number of bytes that can be referenced and modified by the instructions, and a set of registers. The environment, then the methods of execution, will be laid out in detail here.

### Registers

Following the layout of the Intel 80386 processor, there will be 8 general purpose 32-bit registers available for the programmer to use, as well as the single 32-bit instruction pointer (which can only be changed by jump instructions).

The general registers are called: eax, ebx, ecx, edx, esi, edi, ebp and esp. This is a strange naming convention, and it stems from the origins of the Intel x86 series, where each was intended to perform a specific function (for example, the “a” in eax stands for “accumulator”, and “SI” and “SI” stand for “source index” and “destination index”). These purposes have now mostly been lost to history and you can use whatever register you want for whatever purpose.

It is notable, however, that the “e” in each stands for “extended” – that is, 32-bit. If you omit the e, and use simply “ax”, “bx”, etc. then these are treated as their own registers, except only 16 bits in size. However, keeping in accordance with the actual processors, you cannot use “eax” and “ax” at the same time as in hardware “ax” is actually just the upper 16 bits of “eax”. This leaves the first 16 bits unused, so they are now called “al” for the lower 8 bits and “ah” for the higher 8 bits of this remaining space\*. However, these divisions can only be done for the first 4 registers. Below is an illustration of this complicated system, with the full 32-bit register name written on the left and how it is divided on the right:

|  |  |  |  |
| --- | --- | --- | --- |
| EAX | AH | AL | AX |
| EBX | BH | BL | BX |
| ECX | CH | CL | CX |
| EDX | DH | DL | DX |
| ESI |  | | |
| EDI |  | | |
| EBP |  | | |
| ESP |  | | |

These registers each have their own ID in the bytecode (which will be described in its section). These registers can be used to store whatever values you want them to, and they are intended as the primary form of temporary storage. The nature of how data types work is a matter for the bytecode specification to discuss, and these registers simply store a sequence of bytes without concern for their meaning.

Another thing that in real-life hardware is implemented with registers is flags (i.e. purpose-built booleans). In this CPU, there are three flags: the equality flag, the negative flag and the positive flag. They are used for comparisons but cannot be directly accessed by the programmer.

### Memory

The memory is a contiguous series of bytes, starting at memory address 0 and working upwards to a set maximum that the metadata of the bytecode will specify. Certain portions of memory will have designated purposes, based on the standard layout for C programs.

\* In the actual Intel processors, the AH and AL are the higher and lower 8 bytes of AX, not of the upper half of EAX. This means in the real processors, you must either utilise AH/AL or AX, but not both at the same time. In this system it is different.

At the lowest address is placed the “text” section, which is the sequence of instructions. The program counter will start at address 0, and will make its way up until it reaches an instruction to halt (a byte of zeroes). This will terminate the program.

Above this will be the data section, which stores all of the variables initialised in the assembly code. By the time the interpreter is running, the assembler will already have turned these variables into simple memory addresses, and their values will be initialised at the beginning of the program.

Above that, the remainder of the memory will be designated for the heap, which stores objects which use dynamically allocated memory. It will work upwards from the top of the data section, and will all be initialised to a value of 0. It will be possible in a further iteration for the stack to be implemented, which will work down from the top of this same section, but this is not a necessary initial feature\*.

All of this is worth pointing out, but it is not a fundamental part of the interpreter. The first few bytes of the bytecode file tell the interpreter how much memory to lay out, and the interpreter will simply move all of the instructions from the bytecode directly into the start of the memory and start running. Any initialisation of variables or the heap should be decided on by the assembler and written as the first instructions in the program.

### Bytecode design

To store instructions, a bytecode design must be formulated. It has to be capable of storing all of the instructions that the assembly language can encode, and be as close as possible to its design to simplify the assembly process. It must also handle the trade-off between memory efficiency and speed efficiency. I have outlined below the nature of the machine code language’s design.

A bytecode instruction starts with an opcode byte. It has a number that represents the instruction to execute, analogous to an assembly instruction. This will always be followed by a byte that describes the nature of the arguments that will come next. For example, if the instruction has been called with two registers, or a memory address and an immediate value. Based on the contents of this, the interpreter will then read the correct number of argument bytes and execute the instruction.

There are, in effect, four types of operands: registers, immediate values, memory addresses and arithmetic addresses. The operand byte is split into two nibbles of four bits, each describing one of the operands. The possible values, and corresponding encoding of the operands, are listed below.

* None – Designated 0 – There is just no operand.
* Registers – Designated 1 – This will require 1 byte to be read. This will hold a number corresponding to this table:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| EAX | 0xA0 | ESI | 0xE1 | AX | 0xA1 | AH | 0xA2 | AL | 0xA3 |
| EBX | 0xB0 | EDI | 0xE2 | BX | 0xB1 | BH | 0xB2 | BL | 0xB3 |
| ECX | 0xC0 | EBP | 0xE3 | CX | 0xC1 | CH | 0xC2 | CL | 0xC3 |
| EDX | 0xD0 | ESP | 0xE4 | DX | 0xD1 | DH | 0xD2 | DL | 0xD3 |

Note here that all of the lower values are untouched, which will be useful later. In addition to these, there are also 0xF0 and 0xF1, which are used for output and input respectively.

* Immediate value – Designated 2 for an 8-bit value, 3 for a 16-bit value and 4 for a 32-bit value - requires the corresponding number of bytes to be read.
* Memory addresses – Designated 5 – requires 4. Memory address operands will only have come from variables and labels in the bytecode.
* Arithmetic addresses – Designated 6, 7, 8, 9 or 10 - These effectively require a sequence of instructions. There are five possible forms of instruction: a (which is 6), a\*b (7), a+b (8), a\*b+c (9) and a+b\*c (10). The first requires one byte, the next two need two bytes and the last needs three. There is no need to distinguish between whether each of the three bytes will refer to an immediate value or a register, because the numbers designated for the meanings of registers never go below 0xA0, so if the value of the byte is below that then consider it to be immediate.

Below is a table showing the opcode byte and how the different assembly commands are arranged, followed by a table listing the assembly commands, corresponding opcodes and implementation details.

\* The assembler and interpreter have been designed so they should be fully capable of handling a call stack. All of the necessary features are in place in these so that to implement all of the later features in a second iteration, all that has to happen is for the compiler to be changed to generate the right assembly code.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
| 0 | HLT | CMP char | CMP uchar | CMP short | CMP ushort | CMP int | CMP uint | CMP float | JMP | JE | JNE | JLT | JLE | JGT | JGE |  |
| 1 | MOV 1B | MOV 2B | MOV 4B |  | LEA |  |  |  |  |  |  |  |  |  |  |  |
| 2 | ADD char | ADD uchar | ADD short | ADD ushort | ADD int | ADD uint | ADD float |  | SUB char | SUB uchar | SUB short | SUB ushort | SUB int | SUB uint | SUB float |  |
| 3 | MUL char | MUL uchar | MUL short | MUL ushort | MUL int | MUL uint | MUL float |  | IDIV char | IDIV uchar | IDIV short | IDIV ushort | IDIV int | IDIV uint | IDIV float |  |
| 4 | MOD char | MOD uchar | MOD short | MOD ushort | MOD int | MOD uint | MOD float |  | EDIV char | EDIV uchar | EDIV short | EDIV ushort | EDIV int | EDIV uint | EDIV float |  |
| 5 | AND 1B | AND 2B | AND 4B |  | OR 1B | OR 2B | OR 4B |  | XOR 1B | XOR 2B | XOR 4B |  | NOT 1B | NOT 2B | NOT 4B |  |
| 6 | LSH 1B | LSH 2B | LSH 4B |  | RSH 1B | LSH 2B | LSH 4B |  |  |  |  |  |  |  |  |  |
| 7 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 8 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 9 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| A |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| B |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| C |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| D |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| E |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| F |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

|  |  |
| --- | --- |
|  | Control flow |
|  | Data management |
|  | Arithmetic |
|  | Unused/reserved |

|  |  |  |
| --- | --- | --- |
| Assembly instruction | Num opcode | Description |
| **Control flow** | | |
| Halt (HLT) | 0x00 | This is a single byte instruction, a full byte of zeroes. It causes the execution to stop. |
| Compare (CMP) | 0x01-0x07 | These are the comparison instructions; which exact opcode is used tells the interpreter the size of the data to read and the way to interpret it. Once the numerical operands have been found, a subtraction of the second away from the first is performed. The positive flag, negative flag and equality flag are set based on this. |
| Jump (JMP) | 0x08 | This will only take one operand, and the operand will be a memory address. The program counter will be set to this address. |
| Jump if equal (JE) | 0x09 | Takes one operand and sets the PC to equal it if the equality flag is set to true. |
| Jump if not equal (JNE) | 0x0A | Takes one operand and sets the PC to equal it if the equality flag is set to false. |
| Jump if less than (JLT) | 0x0B | Takes one operand and sets the PC to equal it if the negative flag is true |
| Jump if less than or equal to (JLE) | 0x0C | Takes one operand and sets the PC to equal it if either the negative flag or the equality flag is set to true |
| Jump if greater than (JGT) | 0x0D | Takes one operand and sets the PC to equal it if the positive flag is set to true |
| Jump if greater than or equal to (JGE) | 0x0E | Takes one operand and sets the PC to equal it if either the positive flag or the equality flag is set to true |
| **Data management** | | |
| Move (MOV) | 0x10-0x12 | Takes two operands. Based on the second byte of the opcode (if it is 0, 1, or 2) decide if this is the 1, 2 or 4 byte version. |
| Load effective address (LEA) | 0x14 | Takes two operands. Take the second operand and computes it – that is, do not load the value of that memory address, but if it is something like [EBP\*4+EAX] then compute this. Move this value, which must be a memory address, to the location defined by the first operand. |
| **Arithmetic** | | |
| Add (ADD) | 0x20-0x26 | Loads the value determined by the second operand (retrieving it from memory/registers) and increase the value stored at the address in operand 1 by this amount. Which opcode is used determines the number of bytes and method of decoding to use. |
| Subtract (SUB) | 0x28-0x2E | Does the same as above, but performs subtraction |
| Multiply (MUL) | 0x30-0x36 | Does the same as above, but performs multiplication |
| Integer division (IDIV) | 0x38-0x3E | Does the same as above, but performs floored division |
| Modulus (MOD) | 0x40-0x46 | Does the same as above, but performs a modulus operation (remainder of a division) |
| Exact division (EDIV) | 0x48-0x4E | This will load the values stored at the locations specified in opcode 1 and opcode 2, decoding as instructed by which opcode is used. It divides in an exact way, producing a float. It will place this float in the position of opcode 1. |
| Bitwise AND (AND) | 0x50-0x52 | This will determine the number of bytes based on which opcode is used, and then take the values from the locations specified by the opcodes. It will perform a bitwise AND operation, placing the result in the location of opcode 1. |
| Bitwise OR (OR) | 0x54-0x56 | Does the same as above, but performs and OR operation |
| Bitwise XOR (XOR) | 0x58-0x5A | Does the same as above, but performs and XOR operation |
| Bitwise NOT (NOT) | 0x5C-0x5E | Takes only one operand. Goes to this memory address and flips all of the bits at that location (with the length given by which opcode was used). |
| Left shift (LSH) | 0x60-0x62 | Reads the bytes at the location of the one operand given, and performs a left binary shift, adding zeroes to the right and chopping off values shifted past the left hand side |
| Right shift (RSH) | 0x64-0x66 | The same as above but in the other direction. |

This description of the full specification should be thorough, and anything not specified here should simply be considered an implementation detail. I considered it important to have all of this information written out fully before the beginning of the actual decomposition of the parts, so that in that section I already had a clear aim of exactly where I was heading so I know how to get there. It is also simply important in principle to have the exact specification laid out, so that a person can easily use and understand the system.

# Compiler

The job of the compiler is to take C code and turn it into assembly code.

TBC

# Assembler

The assembler will take in assembly code and produce bytecode. Compared to the process of compilation, this is quite simple, because by the design of assembly code it is already very low level and an approximate 1-to-1 representation of the opcodes that will be produced. The format is quite strict, which makes the text processing much easier.

Below is a diagram decomposing the overall task of “turn assembly into bytecode” into several smaller steps:

I will below take each of these sections and further break them down. Each section, generally, will be split up into a table giving the steps and justifications. Lines that have a slight blue background will be broken up further.

## Perform text normalisation

Three things have to happen to “normalise” the text: comments must be removed, empty lines must be removed, and unnecessary (i.e. repeated) whitespace must be removed. This will require no extra level of decomposition, and the process will simply be the following sequence, with notes about why (or why in this order) provided:

|  |  |  |
| --- | --- | --- |
| No. | Action | Notes |
| 1 | Split the text into lines | Every action to be taken is to do with lines so this as a first step is logical |
| 2 | For each line, if there is a semicolon, remove everything after the first semicolon | A semicolon makes everything after it, including other semicolons, into a comment |
| 3 | Strip all whitespace from the start and end of every line | Doing this now will guarantee lines with only whitespace are empty, so these are included in the next step |
| 4 | Remove any empty lines | Now every line will have an instruction |
| 5 | Remove duplicate whitespace | Possibly not necessary but a natural step for normalisation. Use regular expressions to replace any sequence of tabs and spaces with a single space. |
| 6 | Add all the lines back together | This stage was only intended to normalise the text, not actually split it up for context. We put the text back together and from now on treat that text as the assembly. |

## Split document into its sections

Once again, this process should be short and linear. In fact, it cannot really be split into smaller sections. The assembler must just split the document whenever one of section.meta, section.data and section.text appears. Split the text into four groups on the appearance of those characters, recording which section is which. If any of the sections are not present in the original assembly, then put an empty string into their place to remove special cases further down the line.

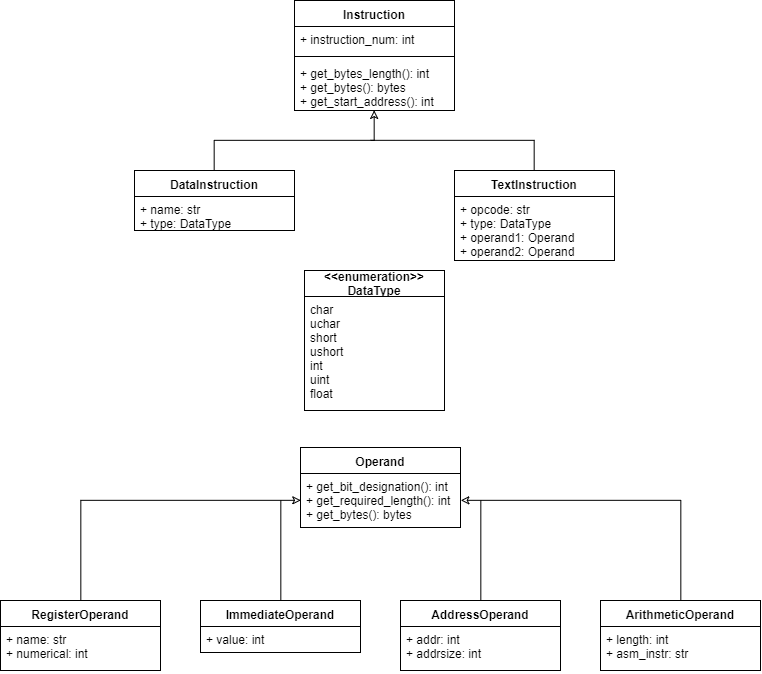
## Divide lines and contextualise

Firstly, split the meta section into lines and interpret them, recording the settings they give (as action 1). For the data and text sections however, do this:

|  |  |  |
| --- | --- | --- |
| No. | Action | Notes |
| 2 | Split the text into lines | Every action to be taken is to do with lines so this is a logical first step |
| 3 | Split the line by whitespace | This separates individual tokens from each other |
| 4 | Convert each line into a program-friendly format (i.e. a custom object) which attaches context to each of the tokens | This means all logic can be done without messing around with text, as everything is correctly packaged |

### Convert each line into a program-friendly format

In my implementation, this program-friendly format will be in the form of an object of type Instruction or a subclass of it. These objects will record data types as values of the enumeration DataType and records operands as instances of the Operand class. The relationships between these are shown in the UML diagram below.



Once this step has been completed, the program will be storing a list of Instruction objects, and the rest of the work will be done on those. Most instructions will be of the TextInstruction form, which is a general command. The DataInstruction type is for variable declarations of the form “name VAR type initial”, such as can only exist in the data section.

Also, when a variable or a label is in an operand’s place, it will be stored there as a string for now.

## Record labels/variables

Labels and variables are two things which are going to be replaced with a memory address eventually. In this step, we are not performing any replacement, but simply scanning through the commands and creating two tables: a table to record the instruction number that each label means, and a table to record the memory location in the data section of each variable. We then combine these into one memory address table. The breakdown is as follows:

|  |  |  |
| --- | --- | --- |
| No. | Action | Notes |
| 1 | Create a str->(int, DataType) table to store the variables, a str->int table for the labels and a str->int table to hold final memory addresses |  |
| 2 | Loop through every instruction in the list |  |
| 3 | If the instruction is a DataInstruction, then add a value in the variable table with the key being the variable’s name. Calculate from the types of the variables already in the table what byte it must be at and record that. | This will lead to a full table mapping the variable names to their location in the data section. |
| 4 | If the instruction is a TextInstruction, and it has a label, then add the number of this instruction to the labels table. | This will yield a table that points the textual labels to the number of the instruction it points to – though not yet its memory address. |
| 5 | Calculate the total size of the text section by adding up the sizes of each instruction | This will be used to turn the offset values in the variable table into absolute memory addresses |
| 6 | For each value in the variable table, add the text section size to it and add this to the memory address table | This memory address is what will be used in the final replacement |
| 6 | For each value in the label table, calculate the total size of all the instructions until the one referenced, and add this to the memory address table |  |

## Convert each line to bytes

Each instruction must now be turned into bytes individually. This is handled in an object-oriented way – that is, each instruction is simply asked to turn itself into bytes.

|  |  |  |
| --- | --- | --- |
| No. | Action | Notes |
| 1 | Create an empty byte string and add to it an encoding of the configuration from the meta section | This means each instruction will have everything it needs to turn itself into its binary string. This is the header of the file that will not be added to the memory but will be separately read in order to properly configure the interpreter |
| 2 | Replace all references to variables and labels with their memory addresses (AddressOperand objects) |  |
| 3 | Loop through the instructions |  |
| 4 | For each instruction object, run get\_bytes() and add it to the string | This is the climax of the program |

Once this is done, that binary string should be the final bytecode to output. It will be displayed on the screen, or written to a file. It should immediately be able to be run by the interpreter.

### 5.1. Encode config data

The encoding of data from the meta section works similarly to encoding of data in a URL. Each config data item is encoded as “key=value” with an & character separating them all (and a final one at the end). This is then followed by 4 null bytes, then the code begins.