System Manual - Pullet16 Assembler

Group 5

Sean A Wiig, Mayank D Patel, Tsung Wei Wu, Mark McMurtury Jr, Samyuktha M Comandur

Homework 6B

6 December 2018

Table of Contents

The Assembler

This program takes a text file containing Pullet16 source code and generates corresponding machine code, which is then dumped to a text file, and is converted to binary and outputted to a binary file.

CodeLine Class

Each line of Pullet16 source code follows a specific format, and the CodeLine class provides a smooth implementation to keep track of each piece of one line of code. The correct format for one line of source code is shown below.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
L	**L**	**L**		**M**	**M**	**M**		**A**		**S**	**S**	**S**		**+/-**	**H**	**H**	**H**	**H**		**C**

Positions 0 - 2 (**LLL**): Label. An optional alphanumeric field that must be less than or equal to three characters long. Must begin with an alphabetic character.

Positions 4 - 6 (**MMM**): Mnemonic opcode. This opcode must be one of the following: BAN, SUB, STC, AND, ADD, LD, BR, RD, STP, WRT or one of the following pseudo-op instructions for the assembler: DS, HEX, ORG, END.

Position 8 (**A**): Addressing flag. An asterisk indicates direct addressing, a blank space here indicates indirect addressing. See page 6 for more information on direct vs. indirect addressing.

Positions 10 - 12 (**SSS**): Symbolic operand. This field follows the same restrictions as the label (Field must be less than or equal to three characters long. Must begin with an alphabetic character.) and is implemented using the Symbol class (discussed on page 3).

Positions 14 - 18 (**+/-HHHH**): Hex operand. +/- indicates that the first bit of the hex operand must indicate whether the operand is positive or negative, and the other four are hexadecimal digits and must be valid hexadecimal values (numbers 0 - 9, capital letters A - F). This validity check is implemented using the Hex class (discussed on page 4).

Positions 20 and above: (**C…**): Comments. Any comments are denoted by an asterisk and continue for the remainder of the line, meaning that this field has a variable length, unlike any other field of the code line.

In order to store each of these lines, a vector of CodeLines is created in the assembler, and each CodeLine consists of the attributes listed above (label, mnemonic, addressing, symbol, hex, comments), the program counter that indicates address of the next line of memory to be executed, the machine code that corresponds to it (denoted by the variable code_), and any error messages associated with that line. The latter two attributes, code and error messages, are populated with values following the first and second pass of the assembler. All of the attributes have accessors associated with them, and the program counter, error message log, and machine code have mutators. These functions will be used in the assembler to access and change aspects of the CodeLine object while it is in the vector.

Symbol Class

As mentioned in the CodeLine class section, a CodeLine can have an optional symbol operand. A symbol is initialized using two values: the textual 3 character label of the symbol (which must be less than or equal to three characters long, and must begin with an alphabetic character), and the location of the symbol (the program counter). These attributes have accessors and mutators associated with them, and this class is used to create the Symbol table in the first pass of the assembler by creating a map of Symbol objects, where the symbol's label is its key. The Symbol class also has a function called CheckInvalid, which ensures that the label of the symbol follows the alphanumeric stipulations listed previously.

Hex Class

The CodeLine can have Hex objects associated with it, and this class checks each character of those hex objects for validity. Note that a Hex object is initialized using a five character string, where the first character is the Hex object's sign (+ or -), and the remaining four characters are valid hexadecimal values (numbers 0 - 9, capital letters A - F). The function ParseHexOperand handles this error checking, ensuring that the hexadecimal values do follow this constraint, and then converts the text to a decimal value for a properly formatted operand. ParseHexOperand is called in the initialization step, and this ensures that only correctly formatted operands have decimal values associated with them. In addition to this function, the class also has accessors for the sign of the operand, its decimal value, its string of text that was used to initialize the operand, and functions that determine whether the operand has an error or if it is null. These functions are used in the assembly step to determine validity of Hex operands in the source code.

OneMemoryWord Class

The OneMemoryWord class is used to store the machine code that is generated from the assembler and ensure the formatting of each word is correct. The class has accessors for each segment of the memory word (opcode, addressing bits, and target address), and a ToString function for pretty printing of each memory word. The accessors allow for clean access to each subsection of the address, which is cleaner and simpler than taking substrings of an ASCII string.

Pullet16 Assembler Class

Pass One

The goal of the first pass is to create a symbol table that keeps track all of the symbols by noting their labels and their program counter values in the Symbol table map. Also during this process, the CodeLine vector is populated. If the line starts with an asterisk, the CodeLine object is set as a comments only line. Otherwise, the new CodeLine object will be fed with all attributes of the line: the label (if one exists), mnemonic, addressing mode, symbolic operand (if one exists), hex operand (if one exists), the comment (if one exists), and finally the line number. The line number is incremented during the loop and will be used for logging purposes and has no effect on the assembled code. Finally, the CodeLine object is also assigned a memory address (represented by the "pc_" variable) but this is a dummy value. This is because determining the memory address is beyond the scope of this part of the code and cannot be determined until the first pass of the assembler begins.

If the line starts with an asterisk, the CodeLine object is set as a comments-only line. Otherwise, the new CodeLine object will be fed with all attributes of the line: the label (if one exists), mnemonic, addressing mode (direct or indirect), symbolic operand (if one exists), hex operand (if one exists), the comment (if one exists), the line number, and the memory address (which is called pc_). The line number is incremented during the loop. The memory address is incremented unless the line is comments-only, or may change as the result of an ORG or DS pseudo-instruction. Finally, if there is a label, then it will be added to the symbol table along with the memory address of the label. If the same label is encountered twice, then no new entry is added to the symbol table and the existing entry will be given the “MULTIPLY” flag, which signifies that a symbol is multiply defined.

Pass Two

The purpose of the second pass is to generate machine code for the given source code. The machine code is made of 16 bits and follows the pattern shown below.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
OP	OP	OP	ADD	T	T	T	T	T	T	T	T	T	T	T	T

Positions 0 - 2 (**OP**): These three bits will represent the opcode. The opcodes are mapped using the following the protocol listed below.

Opcode	Bits in Position 0 - 2
BAN	000
SUB	001
STC	010
AND	011
ADD	100
LD	101
BR	110
RD, STP, or WRT	111

Position 3 (**ADD**): This indicates direct or indirect addressing. If this is a 0, this means direct addressing, and if it is a 1, that means indirect addressing. Direct addressing means that the contents at the memory location (T) are the value of a particular hex operand (i.e., if T = 0000 0001 0010, indirect addressing, the value 18 would be loaded to the accumulator for whatever operation is conducted). Indirect addressing means that the contents at the memory location (T) refer to another address where the real value of the hex operand is found (i.e., if T = 0000 0001 0010, in indirect addressing, the value at location 18 would be loaded to the accumulator for whatever operation is conducted).

Position 4 - 15 (**T**): The target address means two different things, depending on what the opcode is. If the opcode is 111, the target denotes whether the machine code represents a RD (0000 0000 0001), STP (0000 0000 0010), or WRT (0000 0000 0011). Otherwise, the target address refers to the memory address in hexadecimal where the data is to be taken from, either using direct addressing or indirect addressing.

Machine code that follows this pattern is generated in the second pass by checking each CodeLine object's attributes. First, the type of opcode is determined (RD/WRT/STP style machine code, or one of the other types that involve direct/indirect addressing and hex objects). If the opcode is RD, WRT, or STP, the machine code is generated accordingly using the patterns listed above.

If the opcode is one of the other mapped codes from the opcode table above, then its opcode bits are generated and added to the machine code string; otherwise the line is flagged with an error. Following this, the addressing flag of the CodeLine object is accessed to determine whether or not the CodeLine uses direct or indirect addressing, and this is denoted with a 0 (for direct) or 1 (for indirect) appended to the machine code string, which should now be 4 bits. Finally, to determine the last hex object, the symbol in the CodeLine is looked up in the symbol table, and if it is found, that value is converted to binary and appended to the machine code string, which should now be 16 bits. If any errors occur during this process, such as a symbol undefined in the symbol table being referenced, the error is flagged accordingly. The machine code string is used to initialize a OneMemoryWord instance for each CodeLine and is added to a vector of OneMemoryWord objects.

From this vector of OneMemoryWord objects, each string of ASCII zeroes and ones is 1) dumped to a text file and 2) converted to binary, which is used to create a binary file that will be used in the Interpreter step.

Binary to ASCII conversion

In the assembler, the binary files containing the machine code that were previously dumped after the Pass Two is now being read back. This is to ensure that the machine code dumped in the binary files are identical to the ASCII zeroes and ones as found in the adotout text files.