1 The Big Picture

Your assembler’s work is divided into two phases – the “Analysis” phase and the “Synthesis” phase. Broadly, the goal of the Analysis phase is to read and interpret input and the goal of the Synthesis phase is to produce output.

In the Analysis phase you should check the correctness of the input as you process it. Do instructions have the correct number of arguments? Are all the commas and dollar signs in the right place? Do operands like register numbers and integer literals fall in the correct ranges? If no errors are found, you can then store a simplified version of the input (with syntactic markings like commas removed) in a data structure so that it can be easily output in the Synthesis phase.

The other purpose of the Analysis phase is to construct a symbol table, which stores the values of all the labels defined in the code. In the Synthesis phase, this table is used to substitute labels with their values, or compute branch offsets using the label values.

Labels are the reason we must do two passes over the code. If we tried to combine the Analysis and Synthesis phase (outputting machine code as we read the input) we would run into problems where labels might be used before they are defined.

When constructing the symbol table, remember to check for duplicate label definitions!

The two passes over the code do not directly correspond to the Analysis and Synthesis phases – the second pass is really “finish Analysis and perform Synthesis”. During the second pass, before producing output we must perform some remaining correctness checks that could not be done in the first pass. We need to do the following checks:

- Check for uses of undefined labels.
- Check that label operands, when converted to addresses or offsets, fall in the correct ranges.

These checks require a complete symbol table, so they could not be done in the first pass.

You may think it is cleaner to do all range checking in the second pass, rather than splitting it over the two passes. However, we want to catch errors as early as possible, so it is important to do whatever range checking you can in the first pass.

As you perform these checks, you should convert the instruction data gathered into the analysis phase into binary and store it in a buffer. Once you have checked every instruction, you know that the input MIPS program is completely correct, and you can finally write the buffer to standard output!
2 Constructing a Symbol Table

Construct the symbol table for the following MIPS assembly program.

begin:
label: beq $0, $0, after
jr $4

after:
sw $31, 16($0)
lis $4
abc0: abc1: .word after

loadStore:
lw $20, 4($0)
sw $20, 28($0)

end:

Solution:

The value of a label is defined to be the number of non-null lines (lines containing an instruction) that precede the label multiplied by 4. To begin, we mark all the non-null lines in the program.

begin:
X label: beq $0, $0, after
X jr $4

after:
X sw $31, 16($0)
X lis $4
X abc0: abc1: .word after

loadStore:
X lw $20, 4($0)
X sw $20, 28($0)

end:

Now we make a list of labels in the program, and count the number of non-null lines before each label to get the label values.

begin 0
label 0
after 8
abc0 16
abc1 16
loadStore 20
end 28

This is exactly what the symbol table your program must output in A3P3 will look like. Remember to print it to standard error.

End of Solution.
Remarks on implementing the symbol table:

In C++, you can use the `std::unordered_map` class to store your symbol table. The underlying implementation is a hash table, which provides fast lookups on average.

Even though label values represent memory addresses, which are non-negative, you should store them as signed integers in C++. The reason is that to compute a branch offset, you need to do the calculation `(labelValue - instructionLocation - 4) / 4`. If `labelValue` is an unsigned type, the other values in the subtraction will be converted to unsigned, and integer underflow will occur. The variable `instructionLocation` should also be of signed type even though it represents a memory address.

Keep separate counts of “number of lines read” and “number of instructions read”. Line numbers are useful for informative error messages, but label values should be based on the number of instructions read.

3 Handling Labels as Operands

To facilitate easy output in the Synthesis phase, you will probably want to create a dedicated Instruction class that stores instruction information. Some ideas for what this class could contain:

- The location (memory address) where the instruction will appear in the final output.
- The line number on which the instruction appears in the input (useful for error messages).
- The opcode of the instruction (add, sub, etc.)
- The operands of the instruction.
- An `assemble` function that returns the integer corresponding to the instruction.

Note that operands can be numbers or labels. What type should the variable storing the operands be?

You will probably construct instances of the Instruction class during the Analysis phase, before you have a complete symbol table. Therefore, the operands should probably not be integers because you won’t know what value to put for labels.

If you make the operand variables Token type, you can easily store either numbers or labels in them. Knowing the token kind associated with each operand will also help with range checking.

Aside: what type should the opcode be? It may seem like it should obviously be a string, but note that you cannot use strings with switch statements in C++:

```cpp
// This is not allowed in C++
switch(stringVariable) {
    case "Hello": std::cout << stringVariable << std::endl; break;
    default: std::cout << "Not Hello" << std::endl; break;
}
```

If you want the convenience of being able to use switch statements, consider storing the opcode as an enum. You will need to do the initial conversion from string to enum with a series of if/else statements, but then you can use switch statements to check which instruction you are dealing with.

4 Range Checking

Your assembler must perform a number of range checks. We will go over them from easiest to hardest.
4.1 Register Range Checking

Register numbers must lie between 0 and 31, inclusive. This is trivial to check.

4.2 .word Range Checking

The operand for .word can be a decimal integer, a hexadecimal integer, or a label.

If it is a decimal integer, it must lie in the range \(-2^{31}\) to \(2^{32} - 1\) (\(-2147483648\) to 4294967295).

If it is a hexadecimal integer, it cannot exceed 0xffffffff.

The specification does not say whether range checks are necessary for label operands of .word, just that the label value should be encoded as a 32-bit integer. However, it makes sense to check if the label value is in the range of a 32-bit integer before doing this encoding.

The Token::toLong function provided in the C++ starter code behaves a bit strangely with regards to hexadecimal integers. When it reads the hexadecimal integer, it will interpret it as unsigned. However, it will then try to store it in a signed 64-bit integer variable. The result is that if the unsigned value of the hexadecimal integer is larger than the signed 64-bit range, it will be capped at the maximum signed 64-bit value (\(2^{63} - 1\)). The result is that hexadecimal integers returned by Token::toLong are always non-negative.

This means decimal integers, hexadecimal integers and label values can all be handled by a single range check: \(-2^{31}\) to \(2^{32} - 1\) inclusive.

4.3 Immediate Range Checking

For lw and sw, the immediate value can be a decimal integer or hexadecimal integer. For beq and bne, labels are also allowed, and are converted to immediates via \((\text{labelValue} - \text{instructionLocation} - 4) / 4\).

If the immediate operand is a decimal integer, it must be in the range \(-2^{15}\) to \(2^{15} - 1\) (\(-32768\) to 32767).

If the immediate operand is a hexadecimal integer, it cannot exceed 0xffff.

If the immediate operand is a label, the corresponding offset must lie in the decimal integer range.

Since Token::toLong always treats hexadecimal values as unsigned, 0xffff will be interpreted as \(2^{16} - 1 = 65535\), which is outside of the decimal range. This means two separate range checks are required: one for decimal integers and labels, and one for hexadecimal integers.

The decimal integer and label range check is \(-2^{15}\) to \(2^{15} - 1\) inclusive. Remember to store both label values and instruction locations as signed integers so that the branch offset calculation \((\text{labelValue} - \text{instructionLocation} - 4) / 4\) gives the correct result.

The hexadecimal integer range check is 0 to \(2^{16} - 1\) inclusive.

5 Outputting Binary

Your assembler must output raw binary data just like cs241.binasm. For example, if the input is .word 42 it is not correct to output an ASCII string of 0 and 1 characters like “101010”. When your assembler is done, you should be able to use it just like cs241.binasm and run the machine code programs it outputs using mips.twoints or mips.array.

Write a C++ function that takes a 32-bit signed integer as a parameter and outputs it as binary data.
Solution: To output a 32-bit (4-byte) binary word in C++, we use bitwise operations to store each byte of the word in a variable of char type. When outputting a char, C++ will output the actual binary data stored in the variable (as opposed to outputting an int where it prints an ASCII string representation of the integer).

Here is the code:

```cpp
void output_word(int word) {
    char c;
    c = (word >> 24) & 0xff
    std::cout << c;
    c = (word >> 16) & 0xff
    std::cout << c;
    c = (word >> 8) & 0xff
    std::cout << c;
    c = word & 0xff
    std::cout << c;
}
```

The right shift ensures the byte we want to output gets moved to the rightmost 8 bits of the word. The bitwise and with 0xff zeroes out everything except for those rightmost 8 bits.

More concisely, we could write:

```cpp
void output_word(int word) {
    for(int shift=24; shift>=0; shift-=8) {
        char c = (word >> shift) & 0xff
        std::cout << c;
    }
}
```

End of Solution.

6 C++ Practice

Modify the asm.cc starter file to print out a count of the number of lines which contain a syntactically valid .word directive. Such a line consists of zero or more label definitions, followed by a .word directive with exactly one operand (decimal integer, hexadecimal integer, or label), and nothing after the operand (except possibly comments).

To make things simpler, you do not need to check whether the label operands have a corresponding definition. Also, you may assume all input lines can be tokenized by the given scanner – that is, a ScanningFailure exception will never be thrown.

Solution:

```cpp
#include <iostream>
#include <string>
#include <vector>
#include "scanner.h"

int main() {
    std::string line;
    int validLines = 0;
```
try {
    while (getline(std::cin, line)) {
        std::vector<Token> tokenLine = scan(line);
        // Check if there are any tokens on the line.
        if(tokenLine.empty()) { continue; }
        // There are tokens.
        // Skip over all the label definitions (if any).
        int linePosition = 0;
        while(linePosition < tokenLine.size()
            && tokenLine[linePosition].getKind() == Token::LABEL) {
            linePosition ++;
        }
        // After processing labels, check if there are more tokens on the line.
        if(linePosition >= tokenLine.size()) { continue; }
        // There's at least one token after the labels.
        // Check if this token is a .word directive.
        auto token = tokenLine[linePosition];
        auto kind = token.getKind();
        if(kind == Token::WORD) {
            // Check if there are more tokens.
            linePosition ++;
            if(linePosition >= tokenLine.size()) { continue; }
            // There's at least one more token.
            // Check if this token is a valid .word operand.
            auto operand = tokenLine[linePosition];
            auto operandKind = operand.getKind();
            if(operandKind == Token::INT
                || operandKind == Token::HEXINT
                || operandKind == Token::ID) {
                // Check if there are more tokens.
                // There shouldn't be any more tokens after the operand!
                linePosition ++;
                if(linePosition < tokenLine.size()) { continue; }
                // We have a valid .word directive!
                validLines ++;
            }
        }
        // Checking that lines have the correct format like this is fine for an example,
        // but it will quickly become tedious when you have instructions with multiple
        // operands. Can you think of a more concise way to specify the correct format
        // of a line? Can you develop a helper function for checking that lines have
        // the correct format?
    }
} catch (ScanningFailure &f) {
    std::cerr << f.what() << std::endl;
    return 1;
}
std::cout << "There are " << validLines << " valid .word directive lines." << std::endl;
return 0;
}

End of Solution.
Appendix: Bitwise Operations

Here are some examples of bitwise operations on 4-bit binary numbers. Assume that the bit shifts are logical bit shifts, meaning when shifting, vacant digits are always filled with 0s (so when right shifted, negative numbers will become positive). Note that in C++, a right shift may or may not preserve the sign if the number is negative (the behavior is implementation-dependent).

1. \(3 = 0011\)
   \& 5 = 0101
   \[\begin{array}{c}
   \hline
   0001 = 1 \text{ (signed and unsigned)}
   \end{array}\]

2. \(3 = 0011\)
   \| 5 = 0101
   \[\begin{array}{c}
   \hline
   0111 = 7 \text{ (signed and unsigned)}
   \end{array}\]

3. \(3 = 0011\)
   \(<< 2\)
   \[\begin{array}{c}
   \hline
   1100 = 12 \text{ (unsigned)}
   = -4 \text{ (signed)}
   \end{array}\]

4. \(3 = 0011\)
   \(>> 2\)
   \[\begin{array}{c}
   \hline
   0000 = 0 \text{ (signed and unsigned)}
   \end{array}\]

5. \(13 = 1101\)
   \(<< 2\)
   \[\begin{array}{c}
   \hline
   0100 = 4 \text{ (signed and unsigned)}
   \end{array}\]

6. \(13 = 1101\)
   \(>> 2\)
   \[\begin{array}{c}
   \hline
   0011 = 3 \text{ (signed and unsigned)}
   \end{array}\]

What is the purpose of each bitwise operation in the context of the assembler?

Left shifting is used to move each piece of an instruction to the correct position in the final bit pattern.

Right shifting is used when outputting the bytes of a 32-bit word to shift the bytes into place.

Bitwise or is used to combine multiple bit patterns into a single pattern, after each piece of the pattern has been put in place with a left shift.

Bitwise and can be used for “bit-masking”, For example, doing a bitwise and of some 8-bit number \(n\) with the number 00001111 will “zero out” the leftmost 4 bits and preserve the rightmost 4 bits. This is used to extract a particular section of a bit pattern.

The next section demonstrates the use of all these operations more explicitly.
Appendix: Assembling Instructions

There are two instruction formats in MIPS (not counting .word): register format and immediate format.

Register-format instructions are constructed from four numbers: 5-bit unsigned integers s, t and d representing register numbers, and a 6-bit pattern f representing the function code (which tells the CPU which function to perform). They have the following structure:

\[
\begin{array}{cccccccc}
00000 & sssss & ttttt & ddddd & 00000 & ffffff \\
6 \text{ bits} & 5 \text{ bits} & 5 \text{ bits} & 5 \text{ bits} & 5 \text{ bits} & 6 \text{ bits}
\end{array}
\]

A register-format instruction can be assembled as follows:

\[
s << 21 | t << 16 | d << 11 | f
\]

Immediate-format instructions are constructed from four numbers: a 6-bit pattern o representing the opcode (which tells the CPU which operation to do), 5-bit unsigned integers s and t representing register numbers, and a 16-bit signed (two's complement) integer i. They have the following structure:

\[
oooooo & sssss & ttttt & iiiiiiiiiiiiiiiii \\
6 \text{ bits} & 5 \text{ bits} & 5 \text{ bits} & 16 \text{ bits}
\]

An immediate-format instruction can be assembled as follows:

\[
o << 26 | s << 21 | t << 16 | (i & 0xffff)
\]