1 Summary

1. Symbol tables
2. Outputting bytes
3. Assembly errors
4. Outputting instructions

2 Problems

1. 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.

```
begin
label
after
abc0
abc1
loadStore
end
```

Finally, we count the number of non-null lines before each label to get the values.
This is exactly what the symbol table your program should output in A3P4 will look like. Remember to print it to standard error.

We solved this problem by examining the whole program at once. When writing code that constructs the symbol table, you will have to approach this problem a little differently. The code will probably scan through the program line by line and keep a counter of the number of non-null lines. When the program sees the label, it can compute the label's value using the counter and then add the label to the symbol table.

2. The code will look like this:

```c
output_word(int word) {
    output_byte((word >> 24) & 0xff);
    output_byte((word >> 16) & 0xff);
    output_byte((word >> 8) & 0xff);
    output_byte(word & 0xff);
    return;
}
```

Explanation: Suppose the input word is 0xabcd1234. In binary, this is:

```
1010 1011 1100 1101 0001 0010 0011 0100
a b c d 1 2 3 4
```

We want to output all four bytes of this word from left to right: first 0xab, then 0xcd, then 0x12, then 0x34. But output_byte can only output one byte at a time.

We start by figuring out how to output 0xab. We need to manipulate the bits of “word” to get the following binary value:

```
0000 0000 0000 0000 0000 0000 1010 1011
0 0 0 0 0 0 0 a b
```

This value fits into 8 bits, so we can print it with output_byte.

To move the upper 8 bits into the right position, we can use a right bitwise shift by 24 bits, which gives us:

```
1111 1111 1111 1111 1111 1111 1010 1011
f f f f f f f a b
```

Why all the 1s? Since 0xabcd1234 begins with a “1” bit, it is a negative two’s complement number. The bitwise shift will preserve the sign, so the number gets padded with 1s instead of 0s. If we had used an unsigned int rather than an int to store the number being shifted the number would always get padded with 0s.

Now we need to get rid of all the 1s. We can do this by doing a bitwise and with the following number:

```
0000 0000 0000 0000 0000 0000 1111 1111
0 0 0 0 0 0 0 f f
```
Wherever there is a 0 bit in the bottom number, the result will have a 0. Whenever there is a 1 bit, the result will have a copy of the bit in the top number. So this bitwise and operation will zero out everything but the last 8 bits, which it leaves alone.

After this we can print the byte. So to print the first byte, we do:

```c
output_byte((word >> 24) & 0xff);
```

To print the second and third bytes, we basically do the same thing, but we shift by different amounts. To print the last byte, we don’t need to shift at all; we just need to zero out everything but the last 8 bits. Note that `output_byte` only considers the last 8 bits so zeroing out the other bits is not needed.

To assemble the `.word after`, we would look up the label “after” in the symbol table and pass the integer we get (which is 8) into `output_word`.

```c
output_word(table_lookup("after"));
// outputs 0x00000008, value of the label "after"
```

3. 1 label: label: .word label
   2 .word ; 0
   3 .word aaaaa
   4 .word 1 2 3
   5 .word 2147483648 abcde:
   6 .word ,

Line 1: Duplicate definition of “label”.

Line 2: No operand for the `.word`. A `.word` must have exactly one operand.

Line 3: The label “aaaaa” is used here, but it was never defined.

Line 4: Too many operands for the `.word`.

Line 5: Label definitions must appear before instructions on a line.

Line 6: The operand must be an identifier (label name), a decimal integer or a hex integer. A comma is none of those.

Note that there is no “out of range” error on line 5. The number here is outside the range of 32-bit two's complement values, but it is within the unsigned range, and the `.word` operand can be an unsigned or two’s complement integer.

4. `assemble_add(int s, int t, int d) {
   return (s << 21) | (t << 16) | (d << 11) | 32;
}

Explanation: If we look at the MIPS reference sheet, add is encoded as follows:

```
000000 sssss ttttt ddddd 00000 100000
```

In binary, our parameters look like:

```
000000 00000 00000 00000 00000 0ssss
000000 00000 00000 00000 00000 0ttttt
000000 00000 00000 00000 00000 0dddd
```
Notice that to get the bits of s in the right position, we have to shift it to the left by 21 bits. Similarly, we have to shift t by 16 bits. Finally, we have to shift d by 11 bits.

\[
\begin{align*}
000000 & \quad \text{sssss} \quad 00000 \quad 00000 \quad 000000 \quad s \ll 21 \\
000000 & \quad 00000 \quad \text{ttttt} \quad 00000 \quad 000000 \quad t \ll 16 \\
000000 & \quad 00000 \quad 00000 \quad \text{ddddd} \quad 00000 \quad 000000 \quad d \ll 11 \\
\end{align*}
\]

Now we combine them with bitwise or. When we do a bitwise or, any time there is a 1 in a column, the result will contain a 1 in that column. This combines the three words into one.

\[
000000 \quad \text{sssss} \quad \text{ttttt} \quad \text{ddddd} \quad 00000 \quad 00000
\]

Now to complete the assembly of the instruction, we just need to add in the function bits, which tell the MIPS machine that this is an add. We can do this by bitwise-or-ing what we have with 32, the integer that corresponds to the function bits for add.

\[
\begin{align*}
000000 & \quad \text{sssss} \quad \text{ttttt} \quad \text{ddddd} \quad 00000 \quad 000000 \quad (s\ll21)|(t\ll16)|(d\ll11) \\
\text{OR} & \quad 000000 \quad 00000 \quad 00000 \quad 00000 \quad 00000 \quad 100000 \quad 32 \\
\hline
000000 & \quad \text{sssss} \quad \text{ttttt} \quad \text{ddddd} \quad 00000 \quad 100000
\end{align*}
\]

Then we simply return the assembled integer.

Yes, we can easily generalize the \texttt{assemble\_add} function to all R-type instructions. We can do this by adding a fourth parameter representing the f-bits of the R-type instructions and passing the correct value for each R-type instruction in the parameter. This fourth parameter will take the place of the 32 in \texttt{assemble\_add}. However, we need to ensure that this number does not exceed the value 32. e.g.

\[
\text{assemble\_rtype}(3, 4, 5, 32); \quad // \text{Generating the binary for an add}
\]