CS 429 Homework 4

Name: _____________________ Section #: ___________________

Instructions: As usual, you may collaborate with your classmates and ask for assistance from the TA. But don’t copy anyone else’s answer. Submit any code in a hw4.c file and pipe the output to a separate file that you also submit. Don’t forget to include your name in your files.

1. Show the machine code corresponding to the Y86 version of the sumInts program from Slideset 6. Assume that the code starts at location 0x050. For each statement, list address: code bytes. Follow the model on the slide labeled “Sample Program Machine Code” (around slide 21).

2. Annotate the following Y86 code (add comments to each line) explaining what the line accomplishes. The two arguments are passed in %rdi and %rsi, respectively. Use meaningful/useful comments.

   proc:
   irmovq $0, %rax
   irmovq $1, %r9
   rrmovq %rdi, %r8
   andq %r8, %r8
   je done
   loop:
   addq %rsi, %rax
   subq %r9, %r8
   andq %r8, %r8
   jne loop
   done:
   ret

3. Explain what function is computed by the code above. Are there any constraints on the parameters? (Will it work for arbitrary integer inputs?)

4. Write a C function that might compile into the above code.

5. Assuming that proc is at location 0x100, determine the Y86 byte encoding for the Y86 program. For each statement, list address: code bytes.
6. Consider the circuit above. First, write a C expression $E$ that describes the output $Q$ in terms of the three inputs, $A$, $B$, and $C$. Then, write, compile, and execute a C function that prints the truth table for this circuit, including the outputs of each of the intermediate gates (call them $W$, $X$, $Y$, and $Z$ from top to bottom, then left to right). *Use your expression $E$ to generate the outputs; i.e., don’t precompute them and just print the constants.* Loop through the input value; i.e., don’t just call the expression 8 times. In fact, $E$ should only appear once in your code, inside the loop body. Submit your code and the output, which should be similar to the following:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>W</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
7. Consider the following C code, similar to what we considered in slideset 6.

```c
long array[] = {0xd, 0xc0, 0xb00, 0xa000, 0};

/* Count elements in null-terminated list */
long len1( long a[] )
{
    long len;
    for (len = 0; a[len]; len++ );
    return len;
}

main()
{
    long ans;
    ans = len1( array );
    printf( "Answer: %ld\n", ans );
}
```

(a) Assemble this code to generate x86-64 assembly language with minimal optimization (-Og). Please delete any assembler directives, other than .pos, .quad, .string, and labels. Annotate the assembly with comments. Now, using the code examples from the slides and book, rewrite this into assembly code for the Y86. For this part, you don’t have to do `main`, just `len1`. What substantive differences do you see between the Y86 and x86-64 code.

BTW: whenever you’re asked to annotate assembler, point out what’s going on, but not trivia. “Move 1 to %rax” is not a useful comment.

(b) Now assemble it again with higher optimization (-O2). Compare the two compilations. What differences do you observe when the optimization is more advanced? Again concentrate on `len1`. 