**CMPE-250 Laboratory Exercise Nine**

**Serial I/0 Driver**

By submitting this report, I attest that its contents are wholly my individual writing about this exercise and that they reflect the submitted code. I further acknowledge that permitted collaboration for this exercise consists only of discussions of concepts with course staff and fellow students; however, other than code provided by the instructor for this exercise, all code was developed by me.

Joel Yuhas

Performed 4/24/16

Submitted 5/5/2016

Lab Section L2

Instructor: Shaaban, Muhammad

TA: Peter Muller

Stephen Moore

Connor Goss

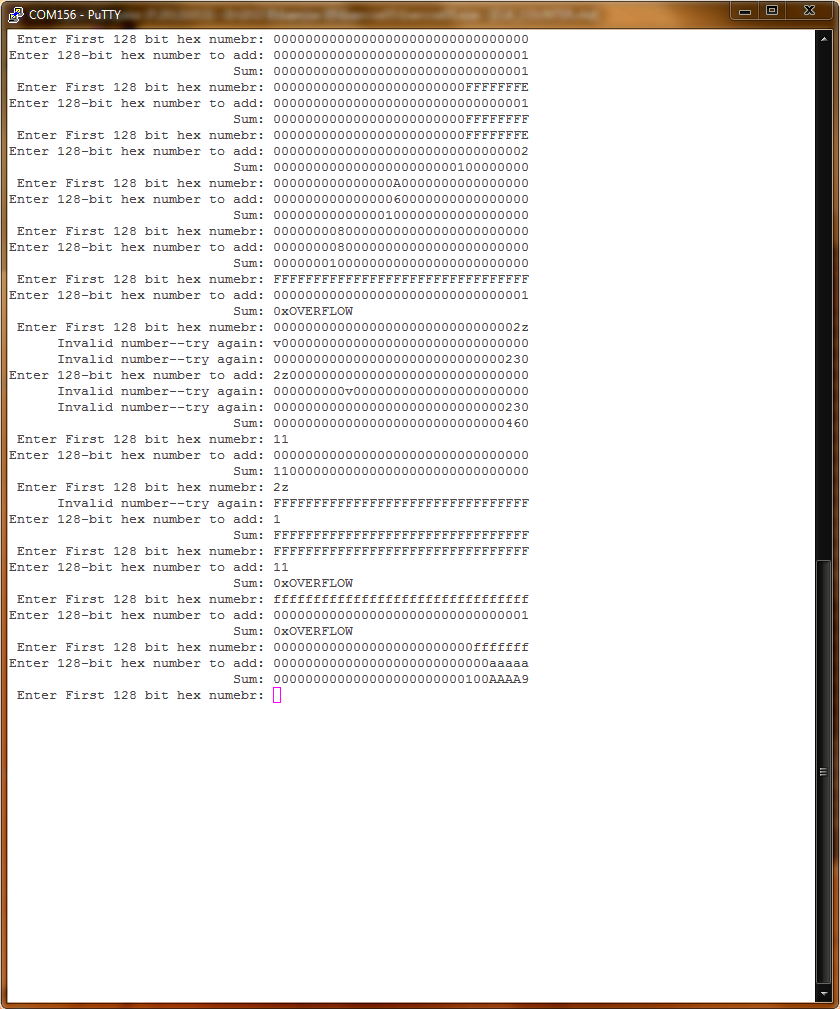
Lecture Section 01

Professor: Sarra, Alessandro

**Screen Capture:**

A screen capture was needed that displayed the output from step 8 along with an explanation of how the screen capture verifies the multiprecision input, addition, and outputs.

Figure 1 shows the screen capture of the terminal window.



**Figure 1: Terminal Window**

The first five sets of inputs displayed on the terminal window from figure 1 test the programs basic addition properties. This was done by ensuring the carry function worked, which was verified using FFFFFFE + 1 and FFFFFFE + 2, which produced correct results. Inputs such as 00000000800000000000000000000000 were used to test the program for proper addition across words, since the inputs and outputs were being stored in a number that was 4 words long.

The sixth addition, with FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF + 1 was used to test the programs overflow abilities as well as other operations such as FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF + FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF.

The next batch of inputs tested to ensure that the program declined to accept letters that weren’t of hex value, and would prompt the user for another input.

Finally, the extra credit was tested. Note that the extra credit, where the program received lowercase as well as uppercase values was fulfilled, as seen in the last summation. However, the other extra credit, where the program would receive a number that was less than 32 bytes’ long only worked partially. As can be seen in the terminal window, the program successfully declined input values that were less than 32 bytes in size and not valid hex digits. However, instead of adding leading zeros, the program would only add trailing zeros to numbers entered that were less than 32 bytes.

**Memory Map:**

The memory map section had to include the starting address of AddIntMultiU, Dequeue, Enqueue, GetHexIntMulti (), GetStringSB, main (), PutHexIntMulti (), PutStringSB, Init\_UART0\_IRQ, and UART\_ISR as well as a discussion of the placement of c code and assembly code in rom.

These values can be seen in Table 1.

Table 1: Functions Starting Address

|  |  |
| --- | --- |
| **Function** | **Starting Address** |
| AddIntMultiU | 0x00000417 |
| Dequeue | NOT LISTED |
| Enqueue | NOT LISTED |
| GetHexIntMulti () | 0x00000759 |
| GetStringSB | 0x0000052F |
| main () | 0x000009E1 |
| PutHexIntMulti () | 0x000002CD |
| PutStringSB | 0x000005A5 |
| Init\_UART0\_IRQ | 0x000005FD |
| UART0\_ISR | 0x00000758 |

Memory Map discussion:

The memory map shows the address location of almost all elements in the code. This memory map in particular is different from previous exercises because it is the first time using C along with assembly. In this case, the C code, which was specific to GetHexIntMulti (), PutHexIntMulti () and main() is stored right alongside the assembly code. For example, the starting address for PutHexIntMulti () is 0x000002CD, which is C code, is in an address that is less than PutStringSB, which is at 0x000005A5. However, GetHexIntMulti () is stored at 0x00000759, which is at a higher value than both, meaning that the C code and assembly code all have their own location in memory individual to each function and not based on whether or not the code is in Assembly or C.

Also note, that there was no listed address for either Enqueue or Dequeue subroutines.