Problem 1

Below is the assembly code to copy the string. This implementation copies the null terminator to the data register of the UART, just as `strcpy(b,a)` in the C standard library copies the null terminator to the new pointer.

```
ADDIU $t0,$0,1
LUI $t1,$0,0x1000
ORI $t1,$t1,0x1234
LUI $t2,$0,0xffff

LOOP:  LB  $t3,0($t1)
ADDIU $t1,$t1,$1

CHECK: LB  $t4,8($t2)
ADDU $r0,$r0,$r0  #NOP
BNE $t4,$t0,CHECK
SB  $t3,12($t2)
BNE $t3,$0,LOOP
```

If a load delay were exposed to the compiler, the assembly would be modified to wait before checking the loaded contents of the control register.

```
ADDIU $t0,$0,1
LUI $t1,$0,0x1000
ORI $t1,$t1,0x1234
LUI $t2,$0,0xffff

LOOP:  LB  $t3,0($t1)
ADDIU $t1,$t1,$1

CHECK: LB  $t4,8($t2)
ADDU $r0,$r0,$r0  #NOP
BNE $t4,$t0,CHECK
SB  $t3,12($t2)
BNE $t3,$0,LOOP
```
Problem 3

This problem may most easily be solved by hierarchical definition of sub-components. There is some ambiguity about off-by-one timing issues with the count: if the selected input (T1 or T0) is zero, it is unlikely that the timer would be specified to asynchronously output ‘DING’ high immediately. Therefore, in this implementation, an input of ‘0’ to the selected input will cause the timer to output ‘DING’ high on the following clock cycle.

This choice has the consequence that the following input state on cycle 0 will cause a high output on cycle 14, not cycle 13:

\[
\begin{align*}
START &= 1'b1 \\
SEL &= 1'b0 \\
T0 &= 11'b1101
\end{align*}
\]

An implementation that outputs ‘DING’ high once cycle sooner would still be correct, however. Here is a top level diagram that includes an 11-bit counter and an 11-bit binary equality comparator.

In order to implement the 11-bit counter, a bit-slice counter is implemented by letting each bit generate the enable signal for the next-most significant bit based on its current value and its current enable status.
Eleven of these bitslices may be combined to form a full 11-bit counter.
Finally, the 11-bit binary equality comparator can be created at a gate level by feeding bitwise XNOR outputs into an 11-input AND gate (among several other implementations).
Problem 3

The second input on each FF below D is an enable.
Problem 4

a.) The serial device will use two shift registers from (3). The controller needs to control the shift and load signals for each; it also generates the valid signals and receives the ready signals.

For the transmit shift register, the start and stop bits need to be copied in when ‘LD’ is asserted; however, the datapath input from the producer of data is only 8 bits. Therefore, the control module produces the MSB and LSB of the parallel input of the shift register – these are the signals ‘TX.0’ and ‘RX.0’ in this diagram. However, RX_SH is always low, and may safely be tied to zero. Below is a sketch of the datapath:

![Diagram of serial device](image)

b.) Using the depiction of the datapath from (a), it is possible to figure out what signals the controller must produce to allow the datapath to function. The controller takes in the serial input so it can see when a serial byte is incoming. ‘TXShift.Bit0’ and ‘TXShift9’ are the signals the controller sends to the parallel input of the transmit shift register to load the start and stop bits. Again, RX_SH is always low, and may safely be tied to zero (and may then be omitted from the diagram).
c.) On the following page is one version of an FSM transition diagram designed to control the diagram.

This FSM relies on two auxiliary FSM components: the timer created in problem 2 and a counter to count the number of symbols received in the transmission so far. The counter is controlled by the ‘Sym_count_inc’ signal and counts from 0 to 9 before overflowing.

For the timing intervals, the ‘T0’ input is tied to 1023 and the ‘T1’ input is tied to 1535 (one clock cycle is spent actually sampling each bit). This machine is very much a Mealy machine; outputs change asynchronously based on the inputs in several state.
This FSM, however, has discrete states for waiting and shifting in bits. Since the FSM moves to the state where a bit is shifted when ‘DING’ is asserted and remains in that state for exactly one clock cycle, it might be easier to simply combine the waiting and shifting states using ‘DING’ to select whether to set the shift control high or low and to start the timer.

By eliminating even more of the synchronous output functions and making the output to the shift register and the counter depend on the input ‘DING’, this FSM diagram becomes much simpler than the previous one. Most of the logic depends on ‘Serial_in’ and ‘DING’. This diagram uses a counter that increments from 0-9 before overflowing; it increments when its ‘inc’ input is high and outputs a 1 on its ‘done’ output when the internal count is equal to 9, further simplifying the logic.
Below is a diagram of a circuit implementing this reduced FSM (all unmarked widths 1b wide). The output and state transition logic is multiplexed by the 2-bit state signal.

This uses the counter depicted on the following page, which counts modulo 10 and has its ‘done’ output high if and only if the count is 9.
The control signals are the ALU source select, the ALU operation select, and the write enables for the product storage, sum storage (power accumulator), and output registers. The output is a fixed-point representation that is proportional to (but not exactly equivalent to) the power, which is the best possible without units specified.