LSU EE 4755 **Homework 4** Solution **Due: 24 November 2014**

Problem 0: Copy the code package from `/home/faculty/koppel/pub/ee4755/hw/2014f/hw04`. Verify that everything is working by running the simulation on the unmodified file. It should report that there is correct output but no compression:

```
Correct output, strings match. But no compression!
In size         117 bytes, out size         117 bytes.
```

Problem 1: Module `asc_to_bin` is to filter a stream of ASCII characters so that ASCII decimal numbers are replaced by binary numbers preceded by an escape character. The idea is to reduce the size of data streams that contain lots of large numbers. For example, consider the sentence, "There are 31536000 seconds in a year." The module `asc_to_bin` should replace that sequence of eight ASCII characters 31536000 with an escape character and an integer encoding of the number.

The module has an 8-bit input and output for the character, `char_in` and `char_out`. There is a 1-bit input `can_insert` which is true when the module can read a character from `char_in`. If input `insert_req` is asserted when `can_insert` is true then the character on `char_in` will be read.

There is a 1-bit output `can_remove` which is true when the character on `char_out` is valid. (It would not be valid if the module does not contain any characters and for other reasons.) If input `remove_req` is set to `1` and `can_remove` is true then the character at `char_out` will change to the next character or, if that's the last available character, `can_remove` will go to zero.

There is also a 1-bit input `reset`. If `reset` is high at the positive edge of the clock then the module should reset itself.

Initially in the homework package, module `asc_to_bin` passes through characters unchanged. Modify it so that it converts ASCII decimal numbers to binary as described above.

At the end of the simulation the testbench will indicate whether the output string is correct, and the original and compressed sizes. For example, the output using the unmodified code package will be:

```
Correct output, strings match. But no compression!
In size         117 bytes, out size         117 bytes.
```

The testbench also provides a trace showing some information each time a character is removed. For the unmodified code,

```
ncsim> run
c  79 = O   tail  1 head  0
c 110 = n   tail  3 head  1
c 101 = e   tail  4 head  2
c  32 =     tail  7 head  3
c  49 = 1   tail  8 head  4
```

The character removed is shown as a decimal number and as a character, for example 110 and "n" for the second line. Also shown are the values of two objects in the `asc_to_int` module, `tail` and `head`. Feel free to add your own variables to the list. Search for "Trace execution" to find the code that prints this trace.

The parameter `max_chars` indicates the maximum size of the integer that should be created. Currently the testbench expects all integers to be of this size.

Keep the following in mind:

- Do not convert a number to binary if it would take more space than the original.
- The module must be synthesizable.
- The synthesized hardware must be reasonably efficient.

For extra credit, modify both the `asc_to_bin` module and the testbench so that `asc_to_bin` can compress a string of ASCII digits to the smallest integer (in multiple of bytes) that can hold the integer. (The current behavior is to use one size integer, determined by parameter `max_chars`.)

The complete solution can be found at `/home/faculty/koppel/pub/ee4755/hw/2014f/hw04/hw04-sol.v` and on the Web at `http://www.ece.lsu.edu/koppel/v/2014/hw04-sol.v.html`.

Encoding the incoming ASCII characters as an integer is straightforward. The tricky part is sending the encoded integer and escape character to the module outputs at the correct time. Remember that characters are removed from the module only when the external device requests them (asserts `remove_req`) so one can't assume that something that has just been read can immediately be sent to the output.

The following approach is used in the solution. Two registers hold the encoded binary values. The design encodes incoming digits as they arrive into register `val_encode`, a second register `val_wait`, holds completed integers that are worth using (the ASCII version is not too short).

Let's suppose the value of `tail` was 7 when the first ASCII digit of a suitable string of digits arrived. Normally, the first ASCII character of this string would be sent to the module output when `head` reaches 7. What we want instead is that the escape character be sent to the output, followed by the bytes of the binary number in subsequent cycles. The solution uses a new array, `esc_here`, to indicate that an encoded integer starts here. If `esc_here[head]` is 1 the module will output an escape character and will switch to using `val_wait` as the source of module outputs for the next `max_chars` characters. It then returns to using `storage` as the source of characters.

Array `esc_here[tail]` is set to zero each time a character is read. When the start of a string of digits is detected (see `start_encoding`) the tail location is saved in `tail_at_enc_start`. When encoding is to end (either due to a non-digit character or overflow, see `end_encoding`) if the encoded number can be used (ASCII number not too short, and `val_wait` is not being used, see `use_encoding`) then we set `esc_here[tail_at_enc_start]=1`.

The updating of the `head` pointer has not been modified: it's incremented at each `remove_req`. However `tail` is adjusted whenever an encoded value is to be used. If the encoding reduces the number of characters by x then x is subtracted from `tail`.

An alternative to using `val_wait` would be to write the escape character and encoded integer into storage. This would simplify the design by removing the multiplexor at the character output and the associated "drain" logic (see the solution code), but it would require a second write port for storage.

Problem 2: Synthesize your module.

(a) Indicate the cost and performance with and without timing optimization. (With timing optimization means using `define_clock`.)

See the table below. The column headed "Timing Constr" indicates the kind of timing optimization. *None* means that no timing constraints were specified and so there was no timing optimization. *Reg -¿ Reg* means that the Encounter `define_clock` command was used, and so timing was optimized from register outputs to register inputs. However the timing of paths starting at module inputs or leading to module outputs was ignored. For the column headed *In, Reg -¿ Reg, Out* the `define_clock` command was used and `external_delay` was also used to indicate the assumption that module inputs are available at the beginning of the clock cycle and that module outputs are expected to be available at the end of the clock cycle.

Without timing optimization the module is 20% cheaper but five times slower. The optimizations were performed with effort set to medium.

Module Name	Area	Clock Period	Timing Constr
asc_to_bin_sol 4 4	206728	20084	None
asc_to_bin_sol 4 4	255460	3844	Reg -> Reg
asc_to_bin_sol 4 4	251736	3687	In, Reg -> Reg, Out

(b) Even if `define_clock` is used, the synthesis program won't optimize all paths, only those with both ends affected by the clock. Show how to use the Encounter `external_delay` command to get the proper timing optimization.

The first command below tells Encounter that all inputs are assumed to be available at the beginning of the clock period for `my_clk` (which needs to have been defined with `define_clock`). The second tells encounter that all outputs are expected to be stable at the end of the clock period. The stuff in the square brackets returns the list of ports, the port names could also have been typed by hand.

```
external_delay -clock my_clk -output 0 [find /designs/*/ports_out/ -port *]
external_delay -clock my_clk -input 0 [find /designs/*/ports_in/ -port *]
```