LSU EE 4755 Homework 6 Solution Due: 2 December 2015

The solution code has been placed in /home/faculty/koppel/pub/ee4755/hw/2015f/hw06/hw06.v and an htmlized version is at `http://www.ece.lsu.edu/koppel/v/2015/hw06sol.v.html`, the original code in htmlized form can be found at `http://www.ece.lsu.edu/koppel/v/2015/hw06.v.html`.

Problem 0: The homework Verilog file, `hw06.v`, contains something similar to the integer compression modules presented in class. (Follow the homework workflow instructions on the course procedures page to get a copy of the assignment package.) These modules compress an ASCII character stream by substituting a binary-encoded integer for a string of ASCII digits. These modules were based on 2014 Homework 4. Feel free to look at that assignment an solution for help.

Module `icomp_none` is a version of the module that does no compression at all. It does though implement the handshaking protocol so that characters can be passed from input to output. This module can be studied to help understand how the others work.

Module `icomp_2cyc` is one of the compression modules covered in class. It computes the encoded value in stage 0, and checks for overflow in stage 1. Don't modify this module, save if for reference. Module `icomp_sol` is initially identical to `icomp_2cyc`, but it should be modified as part of this assignment.

The testbench is set to simulate `icomp_sol` on a sample test string. At the end it will report the amount of compression and whether there was any errors. The testbench also prints out a trace showing some module inputs and outputs and the status of internal signals. Examine the testbench code to see how this is done and feel free to modify it to add signals of your own. A more detailed trace of execution can be obtained using the SimVision gui. To start that use the command `irun hw06.v -gui`. See `http://www.ece.lsu.edu/koppel/v/v/s/SimVisionIntro.pdf` for documentation. (On campus access only without password.)

The synthesis script will synthesize the modules `icomp_2cyc` and `icomp_sol`. Use the synthesis script to make sure that your designs are synthesizable and to determine their cost and performance.

(There is nothing to turn in for this assignment.)

Problem 1: In module `icomp_sol` there is a declaration of a variable named `val_encode_size_1`, but no uses of that variable. Add code to that module so that `val_encode_size_1` is set to the number of bytes that are needed for the number currently in the register `val_encode_1`. For example, if `val_encode_1` has a 0, then `val_encode_size_1` should be 0. If `val_encode_1` has a 123 then `val_encode_size_1` should be 1 (one byte), if `val_encode_1` has a 300 then `val_encode_size_1` should be 2 (for 2 bytes), etc.

To help with your solution add code to the testbench to show the value of this variable.

The solution appears below. The idea is to check each byte of `val_encode_1`, from least significant to most significant. If the byte is non-zero tentatively set `val_encode_size_1` to the byte position (starting at one for the least-significant byte). Note that `val_encode_1` is declared as a two-dimensional packed array, and so the expression `val_encode_1[i]` evaluates to the value of byte number `i` (with 0 being least significant, see the declaration).

```
logic [max_chars:0][7:0] val_encode_1;
logic [mc_bits:0] val_encode_size_1;
always_comb begin
   val_encode_size_1 = 0;
   for ( int i=0; i<max_chars; i++ )
     if ( val_encode_1[i] ) val_encode_size_1 = i + 1;
end
```

Problem 2: Modify module `icomp_sol` so that a group of ASCII digits is compressed into the smallest number of bytes needed, up to `max_chars`. For example, if `max_chars` is 4 then just use one byte to compress 200, two bytes for 4000, and for 1234567890123 use a four-byte integer (for 1234567890) followed by a one byte integer (for 123).

Precede the compressed integer by the character 128 plus the number of bytes in the compressed number. For example, if the compressed value takes two bytes then where the first character of the uncompressed value would go emit a 130, then the next two characters should be the compressed number. (See how `char_out` is assigned in the unmodified code.)

To solve this problem you'll need to understand how the existing code works, how to interpret the trace output provided by the simulator, and how to use the SimVision waveform viewer. Random guesses based on a vague understanding will get you nowhere.

- The module should be written for arbitrary values of `max_chars`.
- Make sure that the testbench is not reporting errors.
- Make sure that your module is compressing the string.

In the original module integers were encoded into `max_chars` bytes. So that the module can now encode integers into sizes from 1 up to `max_chars` bytes the following must be changed:

Encoding Acceptance: The hardware that decides whether to accept an encoded integer must now compare the ASCII length (`ascii_int_len`) to the actual encoded size (`val_encode_size_1`), not to `max_chars`. See `Changed Line` below.

```
wire use_encoding = end_encoding
     && ( ascii_int_len > 2 )               /// Changed Line
     && ( !val_wait_full || end_draining );
```

Tail Changes: The position for writing incoming characters into storage is `tail`. Ordinarily `tail` is incremented each time a character is read. But because an encoded integer takes less space than the ASCII version `tail` must be adjusted after the last character of an encoded integer is encountered. In the original code the adjusted tail value is found by adding the starting point of the ASCII string (`tail_at_enc_start_1`) to `max_chars` plus a possible overflow adjustment. In the solution `max_chars` is replaced by `val_encode_size_1`. (The overflow adjustment adds an extra one to the tail because the tail is being updated one cycle late.)

```
wire [size_lg:1] tail_adj =
    tail_at_enc_start_1 + val_encode_size_1 + overflow_1;
```

Head/Char Out Changes: Module output `char_out` can connect to either `storage` (where the ASCII characters are stored), the escape character (a constant value in the original module), or `val_wait` (the encoded integer). In the original code control logic would connect `char_out` to `val_wait` until `max_chars` characters were read. In the modified module it connects `char_out` to `val_wait` until `val_encode_size_1` characters were read.

In the original code each element of array `esc_here` was one bit, indicating that the corresponding ASCII character in `storage` was the start of an ASCII string that should be replaced by an encoded integer. In the solution each element of `esc_here` indicates how many bytes are in the encoded integer (a 0 means that an encoded integer does not start here). The solution excerpt below shows the new declaration for `esc_here` and how `esc_here` gets written:

```
/// SOLUTION HIGHLIGHTS -- SURROUNDING CODE REMOVED

/// SOLUTION -- Problem 2
//  Increase the size of the "escape here" marker from 1 bit to
//  mc_bits. Its value now indicates the size of the encoded
```

```
//  integer in bytes.
logic [mc_bits-1:0] esc_here [size];

   /// SOLUTION -- Problem 2
   //
   //  Write the size of the encoded integer into the esc_here array.
   //  (Previously we just wrote a 1, to indicate that an encoded
   //  integer starts at this position.)
   //
always_ff @( posedge clk )
   if ( use_encoding ) esc_here[tail_at_enc_start_1] <= val_encode_size_1;
```

Head/Char Out Changes continued: The variable `drain_idx` indicates the byte position in `val_wait` that should be sent to `char_out`. In the original code it was initialized to `max_chars-1`, an elaboration-time constant. In the solution it is set to `esc_here[head]-1`, see the first excerpt below. The final change is to change the escape character. In earlier classroom examples the escape character was a constant, `Char_escape`. In this assignment the escape character should be set to the sum of `Char_escape` and the size of the encoded integer. In the original code, that's still a constant because both `Char_escape` and `max_chars` are elaboration-time constants. But in the solution the encoded size can vary, so we must add the actual encoded size, `esc_here[head]` to `Char_escape`, that appears in the second excerpt below.

```
   /// SOLUTION -- Problem 2
   //
   //  Initialize drain_idx with one minus the size of
   //  the encoded integer, rather than max_chars - 1.
   //
   drain_idx <= start_draining  ?  esc_here[head] - 1 :
                drain_idx > 0   ?  drain_idx - 1 :
                                   0;

/// SOLUTION -- Problem 2
//
//  When we reach an encoded integer output the escape character
//  plus the size of the encoded integer.
//
assign char_out =
       start_draining  ?  Char_escape + esc_here[head] :
       draining        ?  val_wait[drain_idx] :
                          storage[head];
```