LSU EE 4755 **Homework 3** Solution **Due: 28 September 2016**

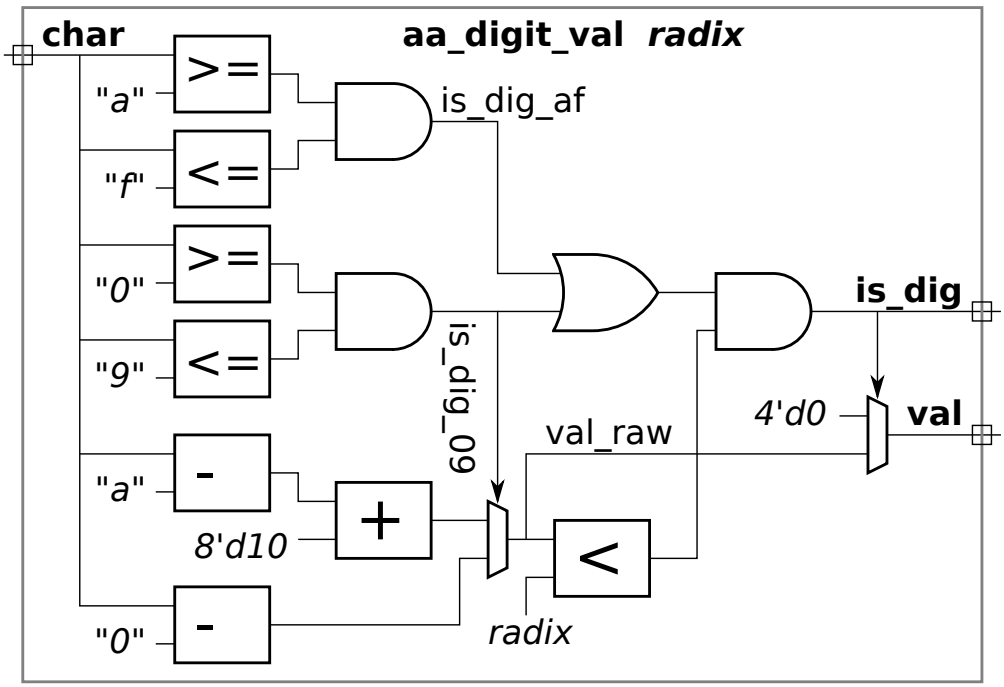
Problem 1: Module `aa_digit_val`, below, is the solution to Homework 2 Problem 1. It has an 8-bit input `char` and two outputs. Output `is_dig` is 1 iff `char` (an ASCII character) is considered a radix-R digit, where $2 \leq R \leq 16$, is the value of parameter `radix`. Output `val` is the value of that digit (in binary), or zero if it's not a digit.

```
module aa_digit_val
  #( int radix = 10 )
   ( output wire [3:0] val,   output wire is_dig,   input wire [7:0] char );

   wire  is_dig_09 = char >= "0" && char <= "9";
   wire  is_dig_af = char >= "a" && char <= "f";
   wire [3:0] val_raw = is_dig_09 ? char - "0" : char - "a" + 10;
   assign     is_dig = ( is_dig_09 || is_dig_af ) && val_raw < radix;
   assign     val = is_dig ? val_raw : 0;

endmodule
```

Provide sketches of what you expect the inferred hardware to look like for `aa_digit_val` as described below. *Hint: Some problems in the EE 4755 2014 Final Exam dealt with numbers in ASCII representation. The optimizations requested below must go beyond those found in the exam solution.*

(*a*) Show a sketch of the inferred hardware before any optimization is done.

Solution appears below. Items in *italic* are constants.

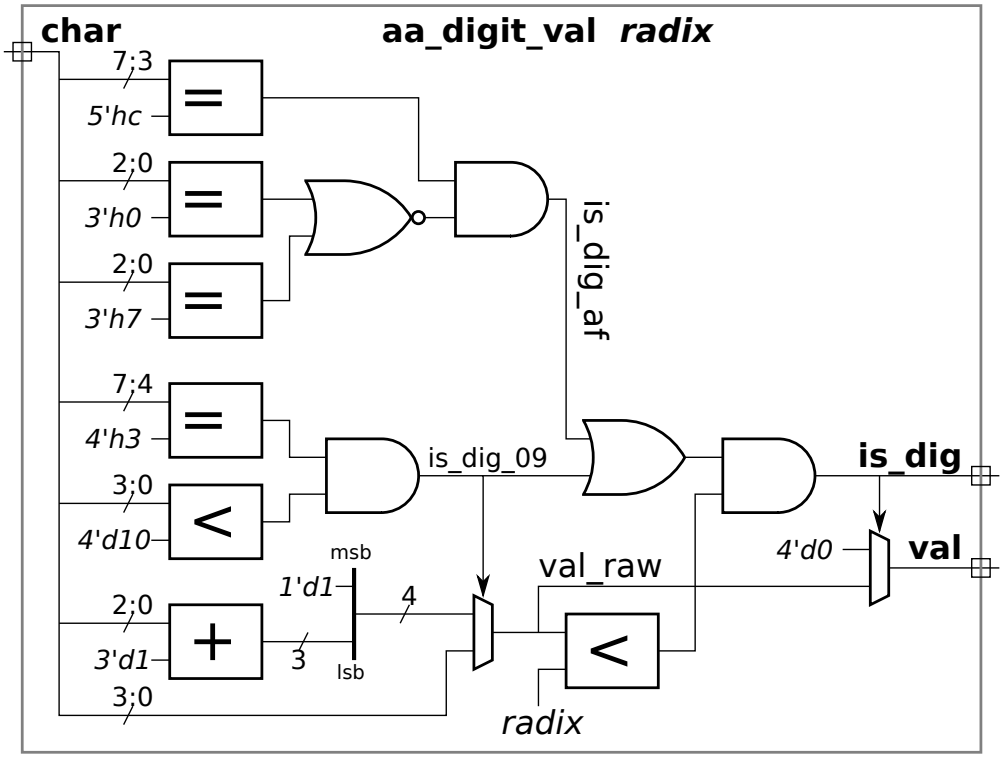


(*b*) Show a sketch of the inferred hardware after some optimization has been performed.

- The sketches must show the product of human thought (in particular, the human who's name is on the submission), not a synthesis program.
- When considering the optimizations for the logic generating `is_dig` (including the logic for `is_dig_09` and `is_dig_af`) recall that in general the cost of logic computing `a==b` is less than the cost of logic computing `a>b`.
- When considering the optimizations for the logic generating `val` think about the subtraction operations and what they actually do when `is_dig` is true. If necessary, work out examples of the subtraction by hand in hexadecimal.

Solution appears below. The optimization to avoid some magnitude comparison when computing `is_dig_09` is based on the fact that the ASCII values of characters "0" to "9" are `0x30` to `0x39` and so one can check whether the most-significant four bits are equal to `0x3` and only do a single magnitude comparison on the lower four bits. Similarly, the optimization of `is_dig_af` is based on the fact that the ASCII values of "a" to "f" are `0x61` to `0x67`, and so one can check whether the five most significant bits are 01100_2 and whether the low three bits are neither 000_2 nor 111_2.

The logic computing the value of "0" to "9" just takes the low four bits of `char`, no arithmetic is performed. The logic computing the value of "a" to "f" adds 1 to the low three bits of `char` and puts a 1 in the MSB position to make a four-bit quantity.



There is another problem on the next page!

Problem 2: Module `aa_full_adder` from Homework 2, Problem 2 adds together two digits of a radix-`R` number represented in ASCII plus a carry in. The module description from the solution appears below.

```
module aa_full_adder
  #( int radix = 10 )
   ( output wire [7:0] sum, output wire carry_out, output wire is_dig_out,
     input wire [7:0] a, b, input wire carry_in, input wire is_dig_in);

   wire [3:0] val_a, val_b;
   wire       is_dig_a, is_dig_b;

   aa_digit_val #(radix) dva(val_a, is_dig_a, a);
   aa_digit_val #(radix) dvb(val_b, is_dig_b, b);

   assign is_dig_out = is_dig_in && ( carry_in || is_dig_a || is_dig_b );
   wire [4:0] sum_val = carry_in + val_a + val_b;
   assign     carry_out = sum_val >= radix;
   wire [3:0] sum_dig_val = carry_out ? sum_val - radix : sum_val;
   assign sum = !is_dig_out ? " " :
          sum_dig_val < 10  ? "0" + sum_dig_val : "a" + sum_dig_val - 10;

endmodule
```

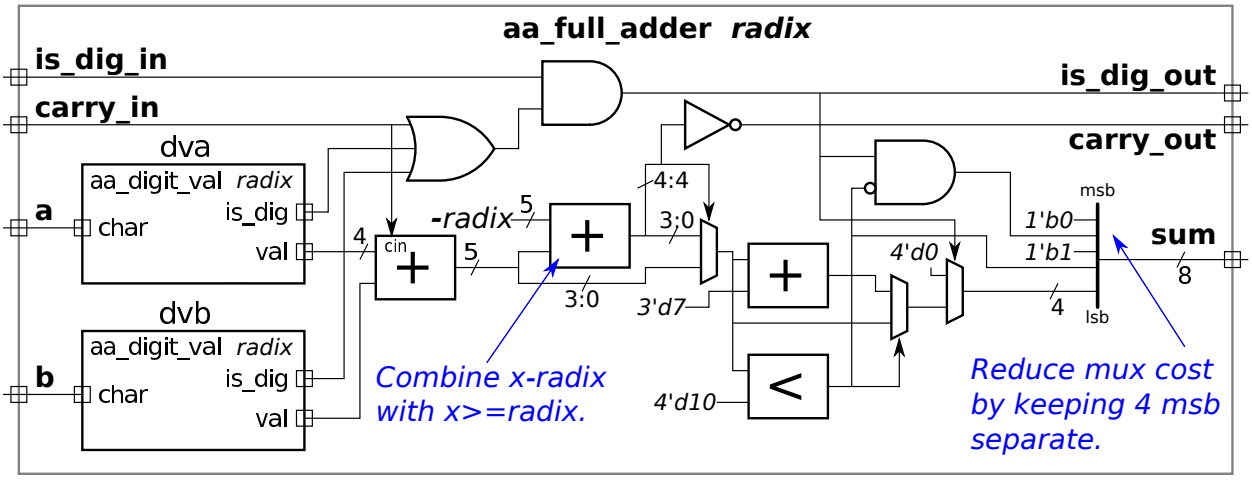
An obvious objection to an ASCII-coded radix-R adder is that it uses 8 bits to represent a digit that can be represented using only $\lceil \lg R \rceil$ bits.

(a) Show the hardware that might be synthesized for the module `aa_full_adder` based on the description above. This should be the inferred hardware with some optimizations applied. Take care to show the number of bits at the inputs and output of units like adders and comparison logic.

Solution appears below. Several optimizations were applied. The logic computing `sum_val >= radix` was eliminated, instead the logic computing `sum_val - radix` was widened to five bits, if the difference is positive then `sum_val >= radix` is true. If the radix is a power of two this logic would not be needed at all, an overflow can be detected by examining one bit position and `sum_val - radix` would simply be the least significant $\lg R$ bits, where R is the radix.

To save multiplexor cost, the 4 LSB of `sum` were computed separately from the 4 MSB. Note that the four possible values for the 4 MSB, `0x2` (for a space), `0x3` (for digits 0-9), and `0x6` (for digits a-f), can be easily be constructed from `is_dig_out` and `sum_dig_val<10`.

Note that only one adder is needed to compute the sum of the two digit values and the carry in, that's because the module's carry in value can go in to the adder's carry in input. It would be very wasteful to show a second adder just to add the carry in.



(b) Compare the cost of a d-digit ASCII-coded radix-16 adder to a $4d$-bit ripple adder. (Note that both adders can add numbers in the range of 0 to $2^{4d} - 1$.) Do so by estimating the cost in terms of the number of gates, and state any assumptions, such as the number of gates needed for an x-bit comparison unit.

The following cost model will be used. All x-input AND and OR gates have a cost of $x - 1$. Inverted inputs and outputs (those little circles) are free! Inverters are also free. A 2-input XOR cost 3 units and a 3-input XOR cost 5 units.

Based on those costs, a binary full adder cost 10 units and a n-bit ripple adder cost $10n$ units. A comparison unit can be made from a ripple adder by eliminating the sum bits, and would cost $5n$ units. An equality unit made from an XOR and an AND costs $4n$ for n bits. (The difference in cost between equality and magnitude is larger for lower-delay designs.) A w-bit, 2-input multiplexor cost $3w$ units.

In many of the adder, equality, and comparison units one of the inputs is constant. That has a big impact on cost. The cost of an n-bit ripple adder drops to $4n$ units (the BFA has a 2-input XOR and a 2-input AND gate to propagate the carry). With one input constant n-bit magnitude comparison and equality drop in cost to just n units.

When `radix` is 16, the `aa_digit_val` module will be simplified further. The `val_raw<radix` comparison is no longer necessary. Based on that the cost is `(+ 4 4 1 3 5 1 1 0 16 12 0 4); = 51` units. The `aa_full_adder` module instantiates two of these and has plenty of logic of its own. The cost including the instantiated modules is `(+ 51 51 2 1 40 4 12 12 4 1 4); = 182` units. (Figuring out the LISP syntax and attaching the costs to parts is left as an exercise to the reader.)

Based on this, the cost of a d-digit ASCII adder is $182d$ units. The equivalent ripple adder costs just $40d$ units. Sure, we expected the ASCII adder to cost more, but over $4\times$ more? Notice that a big part of the ASCII adder's cost are the two `aa_digit_val` modules, $112d$ units.