Lab03

Design of Combinational Multiplier

1 Purpose

- This HW is a pre-requisite for Lab1 which will be covered in another document
- The objectives of this homework are:
    - RTL implementation of a 4x4 multiplier for unsigned numbers using combinational logic
    - RTL implementation of the conversion of binary numbers to binary-coded-decimal (BCD) encoder (binary to BCD conversion) using the "shift-add-3" algorithm
    - RTL implementation of the a seven-segment decoder (SSD) using combinational logic.
    - Integration of the above three blocks as shown in the figure below:

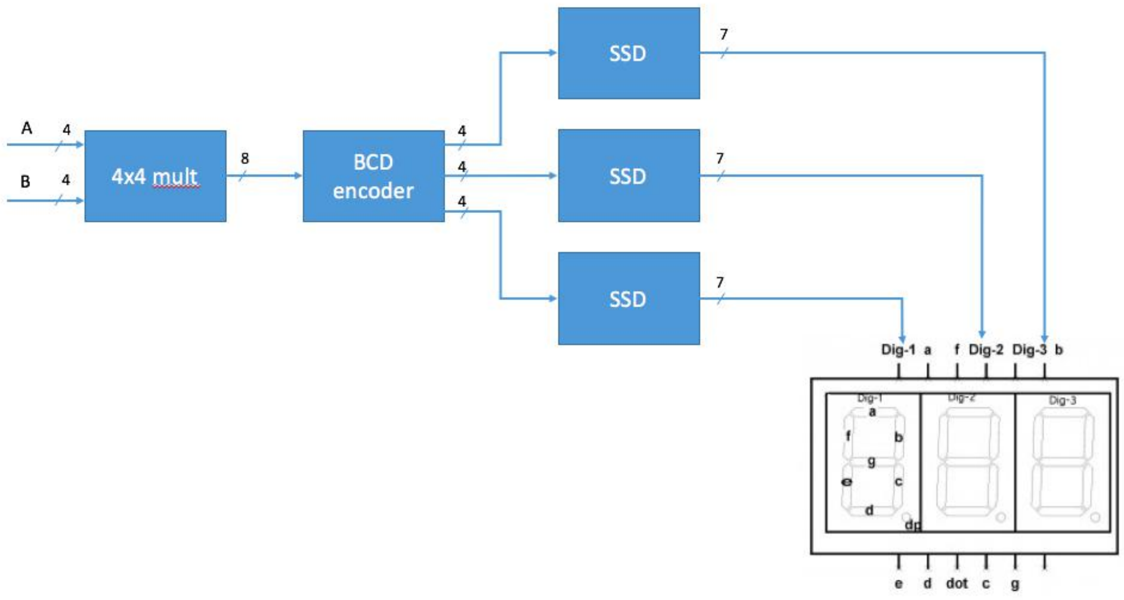


Figure 1: Block diagram for the integrated multiplier

- Validation of the design using testbenches and simulation

2 Required tools

Notepad++ (source code editing)
ModelSim PE student editions (simulation)

3 Multiplier

The multiplication of a 4-bit number by a 3-bit number is shown in the figure below.

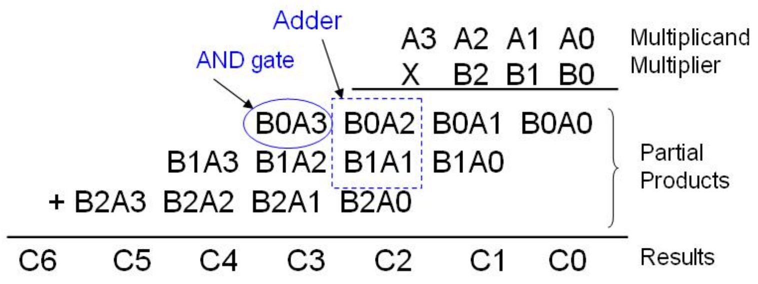


Figure 4: multiplication example

The implementation below follow one-to-one correspondence with the manual multiplication shown above.

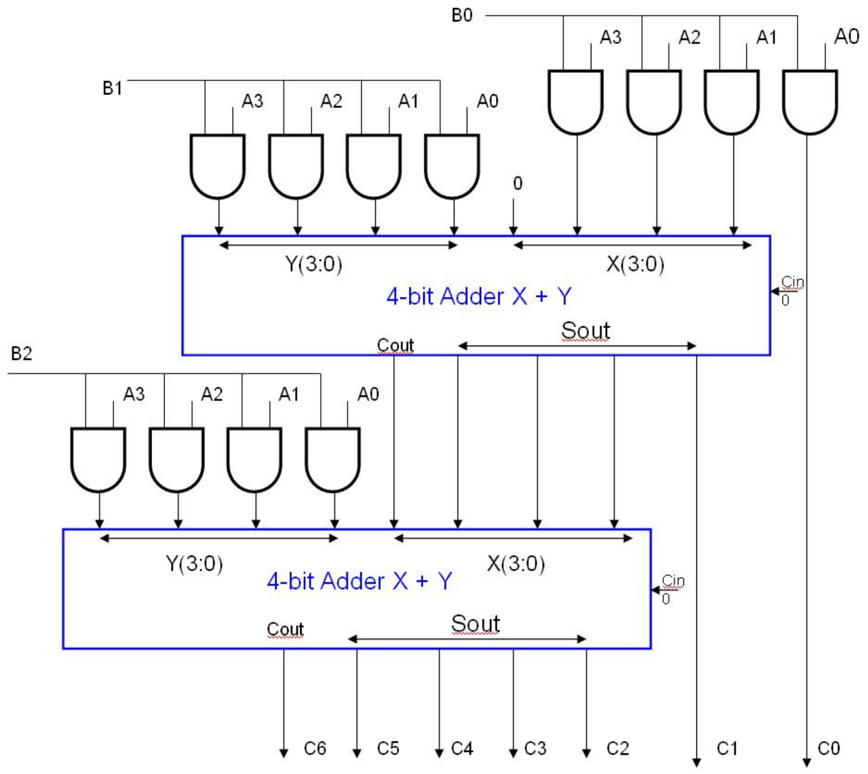


Figure 5: 4x3 multiplier implementation

For 4x4 multiplier, you will need to extend the above block diagram by adding another stage for B3 in a manner similar to B2 and B1.

4 BCD Encoder using the "shift-add-3" algorithm

Another part of this HW is the binary to BCD conversion. This will be implemented using shift-add-3 algorithm. Suppose we have a four-bit binary number that we want to convert to BCD. If it is less than 10, we don't need to change anything. If it is larger than 10, however, we need to do a conversion. We can subtract 10 from the original number to get the ones digit. Then, putting a 1 in the tens digit in BCD can be thought of as adding $0001\ 0000_2=16_{10}$ to the original number. For example,

Start with 12_{10}=1100	0000 1100
Subtract 10_{10}=1010	0000 0010
Add 16_{10}=10000	$0001\ 0010_{BCD}$

If we combine these operations into one, we see that to convert a four-digit number from binary to BCD, just add 6 or 0110_2 if the number is greater than or equal to 10. This works well for four digit numbers, but what if we have more than that? In this case, we can shift the input, one bit at a time, through sets of four bits, and at each step add 0110_2 to a digit if the number is 10 or greater. Here is $22=10110_2$ as an example:

Table 1: Example for "shift-add-6" algorithm

Operation	**Tens**	**Ones**	**Input**
Binary Input			10110
Shift Left (and check if ≥10?)		1	0110
Shift Left (and check):		10	110
Shift Left (and check):		101	10
Shift Left ("Ones" is >10:add 6)		**1101**	0
After adding 6_{10}=0110	**1**	**0001**	0
Shift Left	10	0010	
BCD Output	2	2	

After each shift, one needs to check if the number in the "Ones" column is equal or greater than 10_{10}. If it is, one add 6 to it before shifting again. Notice that when we added 0110_2, the carry bit was carried into the next digit. This means we would need five output bits at each step rather than four. However, there is a shortcut we can use to avoid this. What does shifting a binary number one spot to the left do to the number mathematically? It has the effect of doubling a number (try it out!). If we switch the order of the operations, we only need four bits of output. Specifically, we can add 3 and double instead of doubling and adding 6, because both give the same result in the end. This is where the algorithm gets its name. The new rule then is to shift the input, one bit at a time, through sets of four bits, and if a digit is five or greater, then we add 3 before shifting again. Here is the same example again, using the "shift-add-3" algorithm:

Table 2: Example for "shift-add-3" algorithm

Operation	Tens	Ones	Input
Binary Input			10110
Shift Left (and check if ≥5?)		1	0110
Shift Left (and check):		10	110
Shift Left (and check):		**101**	10
After adding 3_{10}=0011		**1000**	10
Shift Left (and check):	1	0001	0
Shift Left	10	0010	
BCD Output	2	2	

The same procedure can be applied for any number of bits. You will keep shifting (and conditionally adding 3) until all input bits have been shifted (e.g. for an 8 bit you will need to shift 8 times).

Now, the question is how to implement this in your circuit. First, we need a macro which has a four-bit input and four-bit output and which performs the operation "add 3 if the input is 5 or greater". This is up to you to design however you want. Then, the converter can be arranged in the following manner:

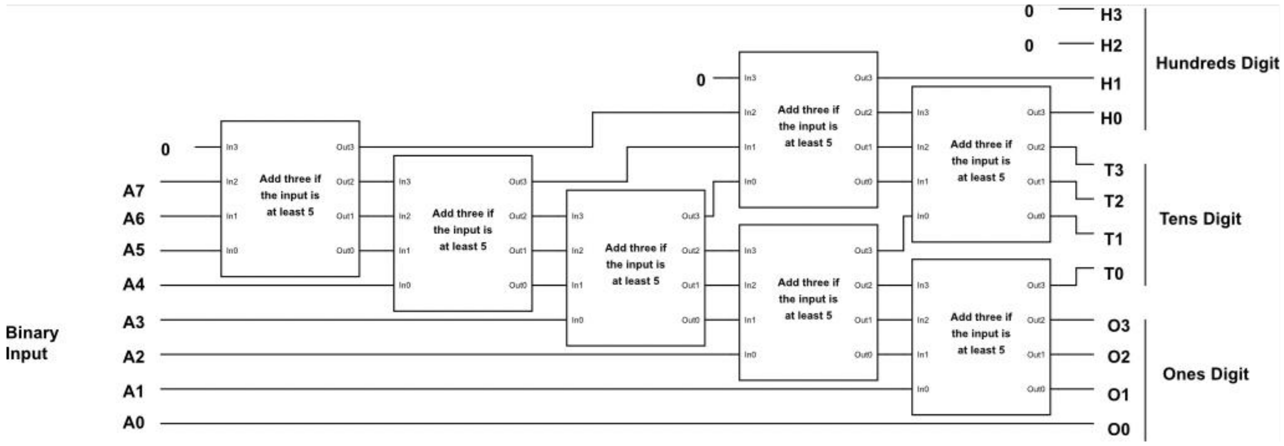
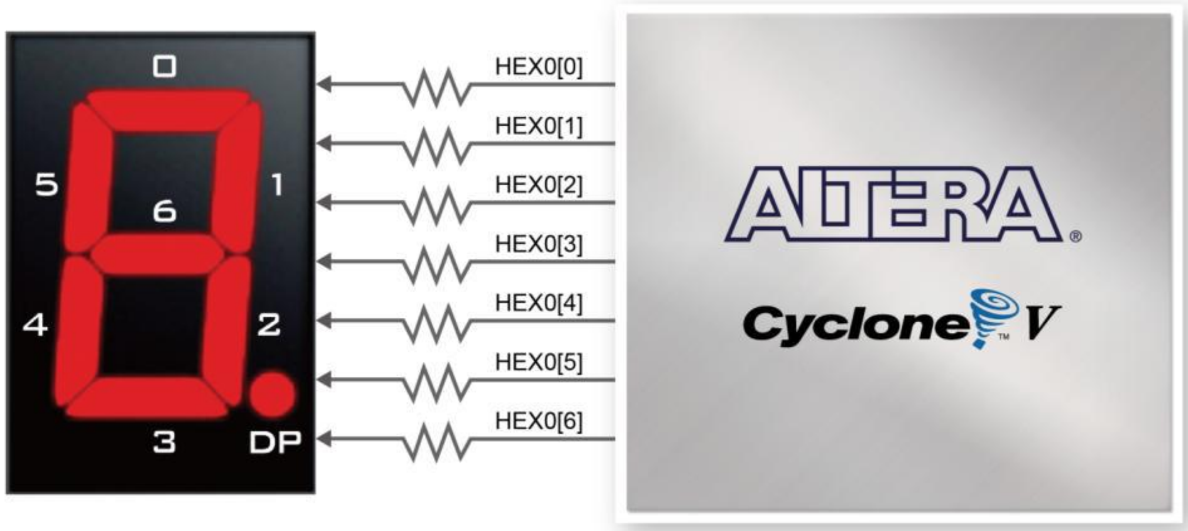


Figure 6: Binary to BCD Converter

5 Seven-segment Decoder (SSD)

7-segment displays are paired to display numbers in various sizes. The figure below shows the connection of seven segments (common anode) to pins on Cyclone V FPGA. The segment can be turned on or off by applying a low logic level or high logic level from the FPGA, respectively. Develop the truth table for converting BCD to the 7-segment display decoder. For example, BCD=0000 will correspond to SSD=100000, BCD=0001 will correspond to SSD=1111001, etc.



6 Submission

1. Verilog RTL code for the different blocks of Figure 1.
   a. 4x4 multiplier (mult4x4.v)
   b. BCD encoder (bcd.v)
   c. SSD decoder (ssd.v)
   d. Integrated multiplier (mult_integrated.v)
2. Write testbench (TB) that verifies the different RTL blocks
   a. TB for 4x4 multiplier (mult4x4_tb.v)
   b. TB for BCD encoder (bcd_tb.v)
   c. TB for SSD decoder (ssd_tb.v)
   d. TB for Integrated multiplier (mult_integrated_tb.v)
3. Simulation captures indicating successful implementation of the different blocks as well as the integrated block.