Design of Digital Systems: Lab Assignment 4

The objective of this assignment is to implement a parameterized carry save multiplier and heavily test it using advanced techniques learned in VHDL.

Background

Multiplication, which is required by many hardware systems, is heavily studied since it is a very expensive operation that requires a large area and long critical path to implement. Carry-save multiplication is one technique that optimizes the standard ripple-carry multiplication by improving the critical path delay. Furthermore, in order to avoid any issues in the design, the unit needs to be heavily tested using a large number of test vectors which is usually generated using an external tool (such as Matlab, Python, Magma, ...). VHDL provides the `textio` library to allow reading from files which holds these test vectors.

Program Specification

The design must meet the following specifications and use all components listed at least once. Some might be used multiple times.

Full adder

The design of the full adder (name it `full_adder`) is shown in Figure 1. In the next section, all full adder figures will have their inputs/outputs as follows:

- Top input (left side) is input `a`.
- Top input (right side) is input `b`.
- Right input is input `cin`.
- Bottom output is output `sum`.
- Left output is output `cout`.

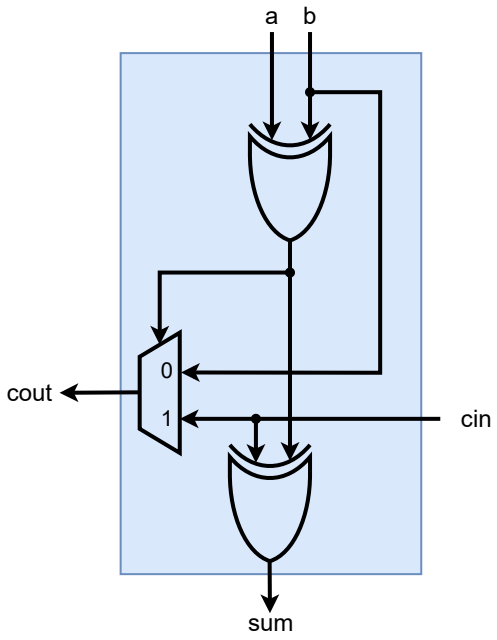

Figure 1: Full adder

Carry-save multiplier

The carry save multiplier (name it `carry_save_mult`) has the following entity:

- Parameters:
  - `n`: The size of the operands.
- Inputs:
  - `a` (`n` bits): The first operand
  - `b` (`n` bits): The second operand
- Outputs:
  - `p` (`2n` bits): The product of the two operands.

Instructions

- In order to perform multiplication, the product of every bit combination of the operands is required. The product of 1 bit by 1 bit is simply `AND`ing the two bits. Create a 2 dimensional $n \times n$ bit signal `ab` which stores the product of every bit combination of `a` and `b`. Use Figure 2 for reference. `ab(i)(j)` in VHDL will correspond to $a_i b_j$ in the figure.

- The architecture of the carry save multiplier is shown in Figure 3 (`FA` indicates full adder). It requires $n-1$ rows, each of which has n full adders. Create five $(n-1)\times n$ bit signals `FA_a`, `FA_b`, `FA_cin`, `FA_sum`, and `FA_cout` corresponding to the inputs/outputs of the full adders and wire them together.
- In order to assign values for the inputs of the `FA`s, some common patterns are needed. There are three major patterns:
  1. The inputs of the `FA`s in the first row (aka row `0`) have a common pattern.
     - `FA_a(0)` is `'0' & ab(0)(n-1 downto 1)`
     - `FA_b(0)` is `ab(1)(n-1 downto 0)`
     - `FA_cin(0)` is `ab(2)(n-2 downto 0) & '0'`
  2. The inputs of the `FA`s in the intermediate rows (aka rows `1` to `n-3`) all have common patterns
     - `FA_a(i)` is `ab(i+1)(n-1) & FA_sum(i-1)(n-1 downto 1)`
     - `FA_b(i)` is `FA_cout(i-1)(n-1 downto 0)`
     - `FA_cin(i)` is `ab(i+2)(n-2 downto 0) & '0'`
  3. The inputs of the `FA`s in the last row (aka row `n-2`) have a common pattern
     - `FA_a(n-2)` is `ab(n-1)(n-1) & FA_sum(n-3)(n-1 downto 1)`
     - `FA_b(n-2)` is `FA_cout(n-3)(n-1 downto 0)`
     - `FA_cin(n-2)` is `FA_cout(n-2)(n-2 downto 0) & '0'`
- The final product (`p`) is computed as follows:
  - Bit `0` is wired to `ab(0)(0)`
  - Bits `1 to n-2` are wired to the `sum` in the first `FA` in each row except the final row. You will need a for generate to wire them.
  - Bits `n-1 to 2n-2` are wired to the `sum` in the last row of `FA`s.
  - Bit `2n-1` are wired to the `cout` of the last `FA` in the last row.

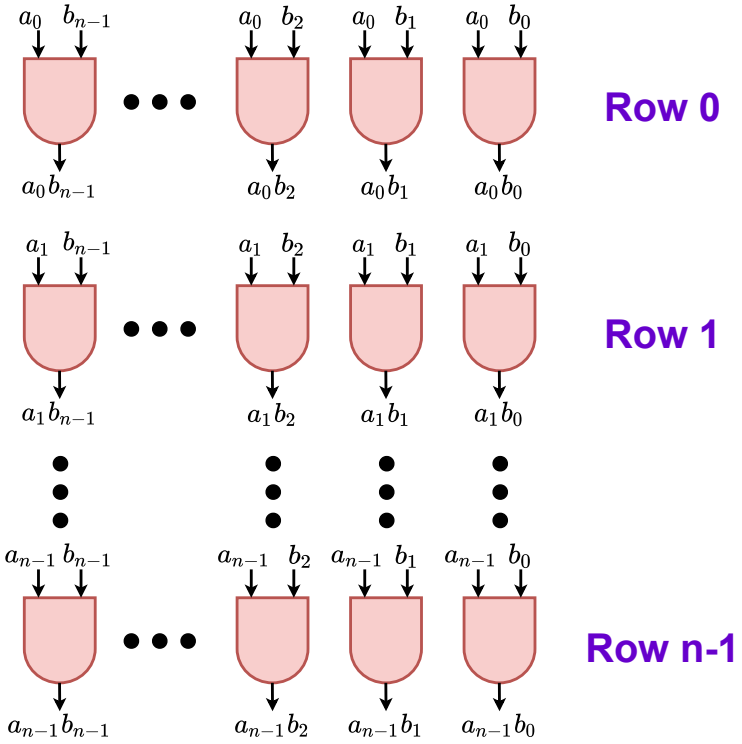

Figure 2: Bit-by-bit product needed for the multiplier

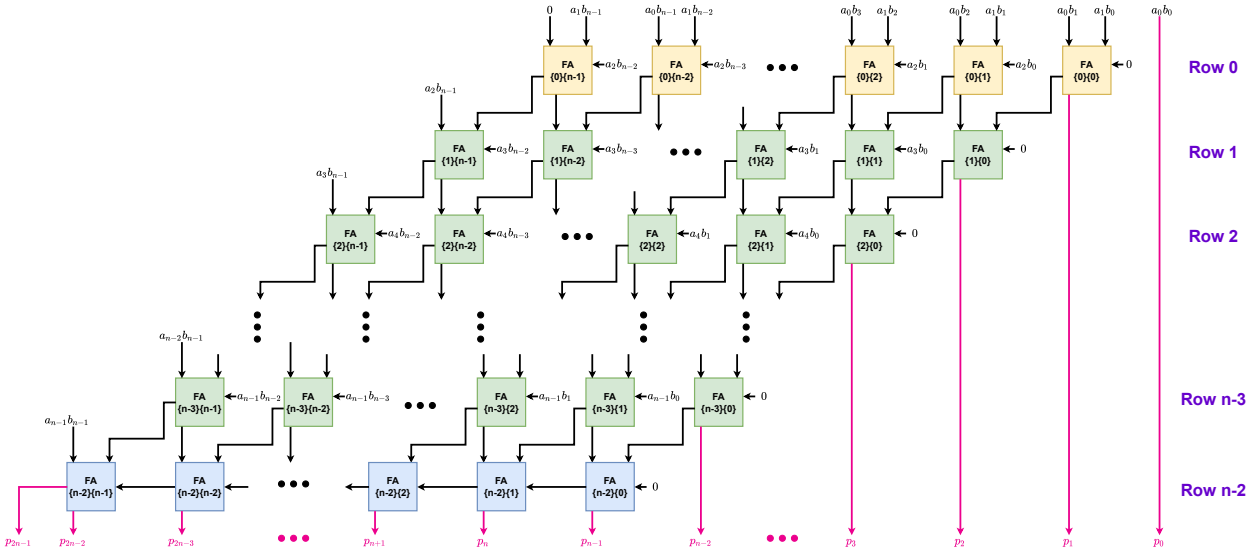

Figure 3: Carry save multiplier architecture

Multiplier wrapper

The multiplier wrapper (name it `mult`) is an 8-bit carry-save multiplier. In order to test the timing results of the multiplier, all inputs and outputs need to be pipelined by a clock as shown in Figure 4. The multiplier has the following entity:

- Inputs:
  - `clk` (1 bit): The clock signal. Flip-flops are triggered on the rising edge.
  - `a` (`8` bits): The first operand
  - `b` (`8` bits): The second operand
- Outputs:
  - `p` (`16` bits): The product of the two operands.

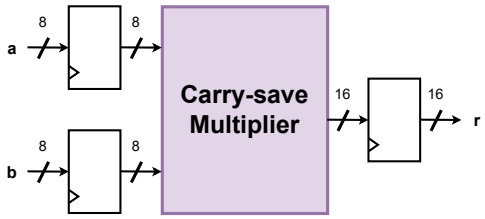

Figure 4: Multiplier wrapper

Testbench

A test vector file (`mult8x8.dat`) is provided to you and contains every possible combination of inputs `a` and `b`, as well as the expected result for each combination. You will need to use the `textio` library to read the vectors from the file. You will then need to pass the inputs of each vector to the inputs of the unit under test and compare the output of the unit under test with the expected result of the vector using the `assert` statement. Once the simulation is complete, print a line indicating that the simulation is complete. All printing statements are shown in the TCL Console in the bottom window of Vivado.

Note that to add the test vector file, do the following

1. Click `Add Sources` from the `Flow Navigator`
2. Select `Add or create simulation sources`
3. Click on `Add Files`.
4. At the bottom of the window, select `All files` for `Files of type`, then find the test vector file and add it.

Report

- For this lab, do not include waveforms. Instead report what you get in the TCL Console at the bottom window of Vivado.
- Collect the area results.
- Collect the timing results. Don't forget to add the following line in the constraints file:

```
create_clock -period 10 -name clk [get_ports clk]
```

- For the demo, test the following values
  - `a=0000_0110` and `b=0001_1011`
  - `a=1111_1011` and `b=0000_0000`
  - `a=1110_0000` and `b=0001_1000`
  - `a=1111_1111` and `b=1111_1111`

FPGA port map

IO	**FPGA port**
clk	E3
a	Switches 15-8
b	Switches 7-0
r	LEDs 15-0