A 8-bit Sequential Multiplier

Learning Goal: A Simple VHDL Design.

Requirements: FPGA4U Board. Quartus II Web Edition and ModelSim-Altera.

1 Introduction

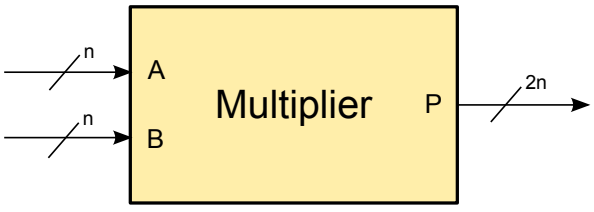
As a reintroduction to VHDL, you will implement a *sequential* multiplier using the *Shift-and-Add* algorithm.

2 Binary multiplication

Example of a 4-bit unsigned multiplication ($11 \times 9 = 99$):

Multiplicand		1011
Multiplier	×	1001
		1011
		0000
		0000
	+	1011
Product		01100011

The next figure is the generic entity of a *combinatorial* multiplier. Notice that the output bitwidth is the sum of the bitwidth of each input operand.

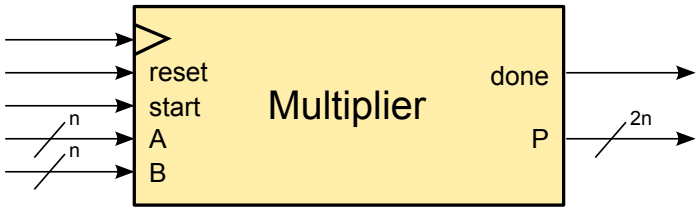


3 Sequential multiplier

A sequential multiplier requires some additional signals for synchronization purpose.

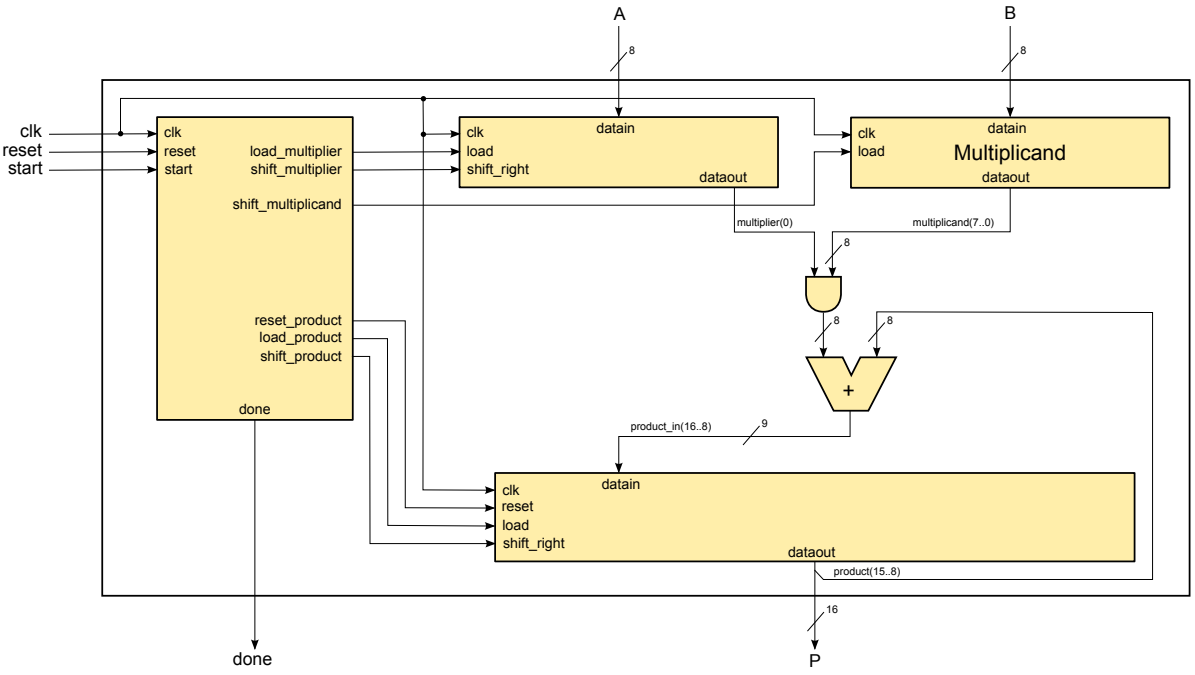
- Input **clk**: clock signal to synchronize the system.
- Input **reset**: asynchronous reset signal to initialize the system.
- Input **start**: synchronous signal that must be high to start a new operation.

- Output **done**: synchronous signal that is set during 1 cycle by the multiplier when the result of the operation is available.



To implement the sequential multiplier, we will use the *Shift-and-Add* algorithm. This algorithm is inspired from the multiplication method you learned at school: you sequentially multiply the multiplicand (operand **B** in the figure) by each digit of the multiplier (operand **A** in the figure), adding the intermediate results to the properly shifted final result.

The following figure describes the architecture of the multiplier.



The components description is listed below.

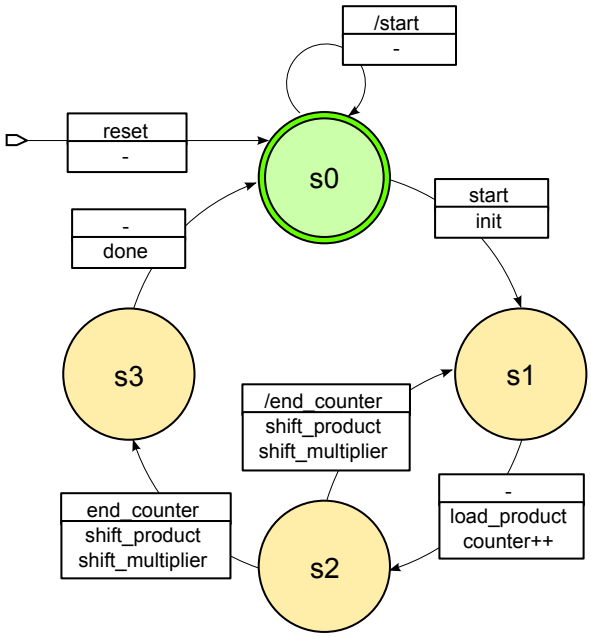
- The 8-bit register **Multiplicand**:
  - Operand **B** is loaded when **load** is high.
- The 8-bit shift-register **Multiplier**:
  - Operand **A** is loaded when **load** is high.
  - It shifts to the right when **shift_right** is high.
  - The **dataout** output is the least significant bit of the register.
- The 17-bit shift-register **Product**:
  - It's initialized to 0 when **reset** is high.

  - It loads the **Add** result in its most significant bits when **load** is high.
  - It shifts to the right when **shift** is high.
  - The **dataout** output bits are its 16 least significant bits.

- The **AND** gates:
  - It performs the AND logical operation on each **Multiplicand** bits with the least significant bit of the **Multiplier**.
- The 8-bit **Adder**:
  - It adds the result of the **AND** gates with the 8 most significant bits of the **Product** output.
- The system **Controller**:
  - It contains a state machine, and it sets the control signals of all the components.

4 State machine

The controller contains the following state machine. It includes a 3-bit saturating counter used to count the 8 steps of the addition algorithm.



- The **reset** signal resets the state machine to state **S0**.
- State **S0**: We wait until the **start** signal is active on the rising edge of the clock before going into state **S1** and initializing the registers where:
  - The **Multiplicand** and **Multiplier** registers load the input values.
  - The **Product** register and the **loop counter** are initialized to 0.
- State **S1**: The next state is always **S2**.
  - The ALU result is loaded in the **Product** register.

  - The **loop counter** is incremented.

- State **S2**: If the **loop counter** saturates (overflow after an increment) we continue to state **S3**, otherwise we return to state **S1**.

  - In each cases, the **Multiplier** and the **Product** registers are shifted.

- State **S3**: The Multiplication is complete and the next state is **S0**.

  - The **done** signal is set.

5 Exercise

Download the project template. In the **quartus** folder you will find the Quartus project. Open it and take a look at the `muliplier_8bits.bdf` schematic file. The structure of the system is already given.

For the **start** and **reset** signals we are going to use the push buttons of the FPGA4U. These buttons send a a logical 0 when they are pressed and a logical 1 when there are released. We want the opposite behavior on those buttons. In the schematic file, inverters have been placed on the **reset_n** and **start_n** inputs.

From the schematic file, you can edit the VHDL code of any instance by double clicking on it. You can also open project files from the **Project Navigator** in the **File** tab.

5.1 The Registers and the Combinatorial Circuits

- Complete the VHDL code for the registers and the combinatorial circuits (i.e., every module except the **Controller**).

- Simulate each component separately in ModelSim.

For the simulations, create a Modelsim project in the **modelsim** folder. Add each VHDL file to it. Compile, correct potential errors, and start the simulations. Some simulation scripts (i.e., `*.do` files) are already present in the **modelsim** folder. You are free to modify them or create new ones.

5.2 The Entire System

ModelSim does not support the `*.bdf` file format. To simulate the entire system, you should convert the schematic file to VHDL:

- Open the `muliplier_8bits.bdf` schematic file.

- In the menu, select File > Create / Update > Create HDL Design File for Current File.

- Click **OK**.

This will generate a `multiplier_8bits.vhd` file in your **quartus** folder. Move it into your **vhdl** folder, and add it to your ModelSim project.

- Complete the **Controller** VHDL code.

- Simulate the entire system and verify that its behavior is correct.

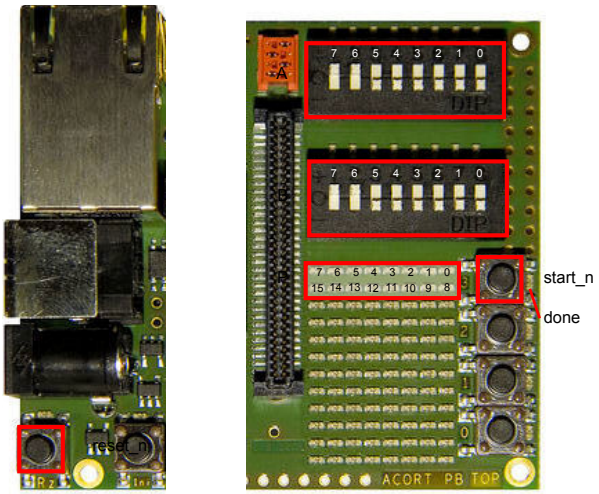
5.3 Programing the design onto the FPGA

If the design works properly in the simulation, you are ready to download the design onto the board. The pin assignment is already set.

- Return to Quartus.
- Compile the project.

It may happen that Quartus sees errors and warnings that ModelSim ignores (like when a signal is set in two or more processes). In that case, correct all new errors until compilation is successful, and simulate again with ModelSim, if necessary.

- Plug your FPGA4U board.
- Open the **Programmer** and download the design onto the FPGA.

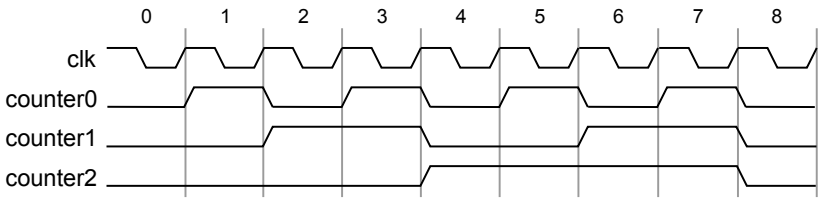


The clock signal has a frequency of 24MHz. Multiplication takes less than 20 cycles and therefore is done in less than 1μs. The steps of the multiplication happen too fast for you to observe something with your eyes.

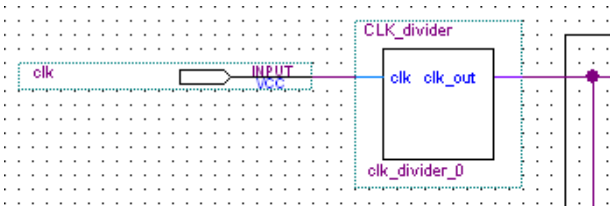
5.4 Clock Divider

Here, you implement a simple clock divider to slow down the execution of the multiplier so that you can observe each step of the multiplication.

An easy way to divide the clock signal frequency is to use a counter. Each counter bit pulses at the clock frequency divided by a power of 2: the bit 0 of the counter will pulse at half the frequency of the input clock, bit 1 at half the frequency of bit 0, bit 2 at half the frequency of bit 1, etc. The following timing diagram gives an example of a 3-bit counter.



- Knowing that the counter is clocked at 24MHz. Find the counter bitwidth needed to achieve approximately one pulse per second on its most significant bit.
- Implement the counter in a new VHDL file. It will take the system clock as input and the most significant bit of the counter as output.
- Generate the counter **Symbol File** and add it in your system to look like the following figure.
- Compile, download the new design onto your board, and observe the result.

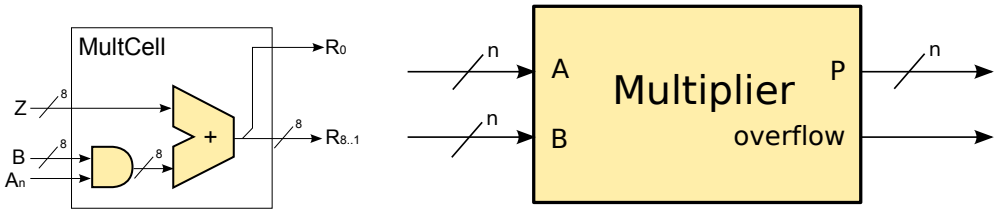


5.5 Combinational Multiplier

We now want to design a combinational multiplier that have the same specifications as the **mul** instruction in *nios 2*, except that we chose to use 8 bits instead of 32 and add an overflow signal:

- Input **A**: 8 bits unsigned integer.
- Input **B**: 8 bits unsigned integer.
- Output **P**: 8 bits product of the multiplication of A by B.
- Output **overflow**: signal who's high when an overflow has been detected.

Implement a structural combinational multiplier in `Combinational_MultCell.vhd`. First, in VHDL, implement a unit containing the logical AND gates and the adder. Then, in `Combinational_Multiplier.vhd`, add 8 instances of this unit (using the example in comment), and connect them in a chain. Finally, connect the overflow signal. Notice that there is an overflow signal because the product has the same bitwidth than the inputs.



6 Submission

Submit all vhdl files related to the exercise in sections 5.1, 5.2 and 5.5 (`Multiplier_8bits.vhd`, `Add.vhd`, `And1x8.vhd`, `Controller.vhd`, `Multiplicand.vhd`, `Multiplier.vhd`, `Product.vhd`, `Combinational_MultCell.vhd` and `Combinational_Multiplier.vhd`)