Pipelining of an Arithmetic Unit

Learning Goal: Pipelining.

Requirements: Quartus, ModelSim, FPGA4U Board.

1   Introduction

In this lab, you will design a single circuit for two different arithmetic functions. The data flow graph of one of the functions is given and you should modify it to implement the second function. First of all, you will design a fully combinatorial circuit and then you will progressively pipeline the circuit.

2   The Arithmetic Expression

The two different arithmetic functions that you will design, are given below.

$$f_0 = (ab + c) \cdot (5a + b) \quad \{a, b, c \in \mathbb{N}\}$$

$$f_1 = a^2 c^2 + b^2 + 2a \quad \{a, b, c \in \mathbb{N}\}$$

Both functions have three inputs: a, b and c. Each input is an unsigned 8-bit variable. The following figure illustrates the *data flow graph* (DFG) of the first function, f_0. The circuit of the first function can be easily implemented by using the given DFG.
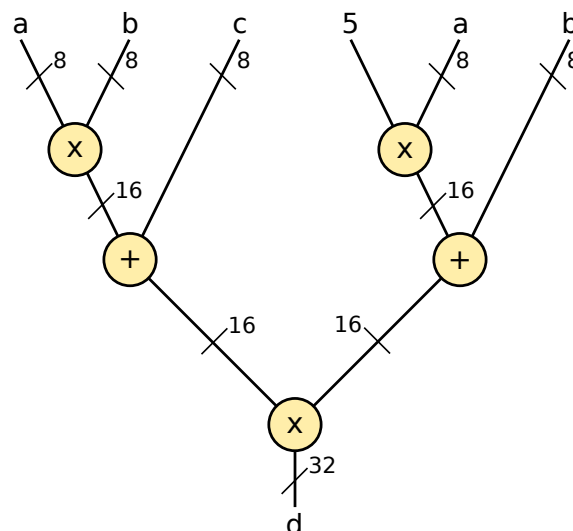
2.1 Exercise

- In the given DFG, do we need to keep the carry-outs of the 16-bit adders? If your answer is yes, what is wrong with the bit-widths given in the DFG? What is the greatest value that can be returned by $ab + c$?

- Draw the DFG of the second function f_1 similarly to the DFG given for the first function f_0. Then, try to merge the two DFGs using the minimum number of resources. The goal is to have one DFG which can be configured to implement either one of the two functions. This is possible by inserting some multiplexers in the circuit to select the appropriate function that is going to be executed. The new circuit has only one output and the same inputs (a, b and c) plus a 1-bit input, sel used to select which function is going to be executed (f_0 when $sel = 0$ and f_1 when $sel = 1$). This new input represents the select signal of every multiplexer that you have in the new DFG.

- *Your final DFG should have three multipliers, three adders and seven 2-1 multiplexers. Shift operations can be used. The output is 32 bits wide. No feedback is allowed in the DFG.*

3 Multiplier

The conventional way to implement a multiplier is to develop the partial products of the two multiplicands, a and b, and then to compute the result by summing the partial products. This is a multi-input addition, which is called *adder tree*. Here, a and b are 8-bit operands, therefore we have to sum 8 partial products:

$$a \cdot b = a_0 \cdot b + ((a_1 \cdot b) \ll 1) + ((a_2 \cdot b) \ll 2) + ... + ((a_7 \cdot b) \ll 7)$$

An efficient way to implement the adder tree is to build a balanced binary tree structure, shown in the following figure. To sum the partial products, first we need to shift some of them to the left. In the following figure, you can see the shifting modules. The shift-left operation can be modelled by concatenating zero values in the least significant bits of the partial products. The number of zeros is equal to the shift value. Therefore, the shift-left operation can be done implicitly without using any shifting module. For each $a_n b$ partial product, you can follow the path to the last adder, and verify that it accumulates a displacement of n bits to the left.

3.1 Exercise

- Download the project template from the web page of the course.

  Note that due to the high amount of arithmetic operations needed in this lab, we have modified the entities of the various components to use the standard VHDL type `unsigned` instead of the `std_logic_vector` type we have used until now.

  The `unsigned` type corresponds to a bit vector of predefined width (similarly to `std_logic_vector`), but has the advantage of supporting standard arithmetic operations such as addition and subtraction (defined through the `ieee.numeric_std` library.)

  The `unsigned` type will help avoid having to excessively wrap all signals with `std_logic_vector(unsigned(...) + unsigned(..))`.

- Open the ModelSim project.

- In the `multiplier.vhd` file, implement the multiplier according to its description.

- Run the `multiplier_tb.vhd` testbench to verify your design.

  - Start a simulation of `multiplier_tb.vhd`.
  - Add the signals of the **Multiplier** to the wave.

- Type `run -all` in the command line.
- A message will tell you if your design is correct or not.
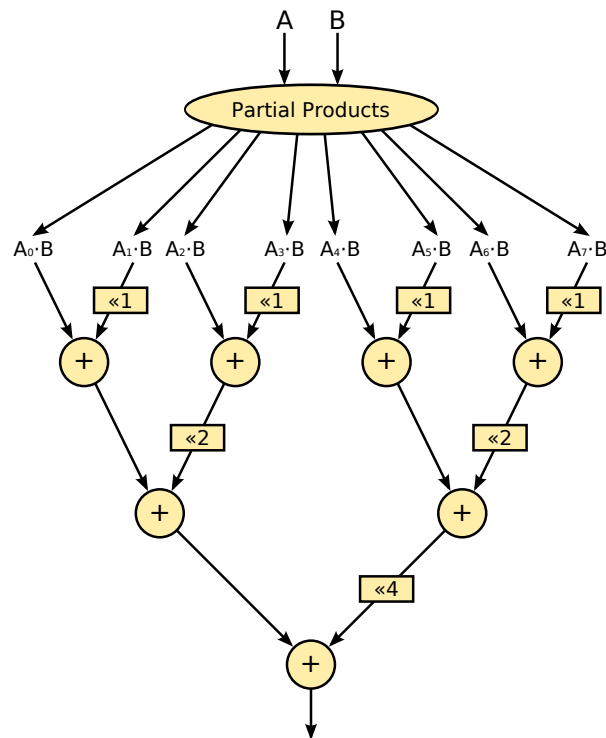
4   The Arithmetic Unit

In this part, you should implement the circuit of the merged DFG that you designed in Section 2. The following figure represents the entity of this unit.
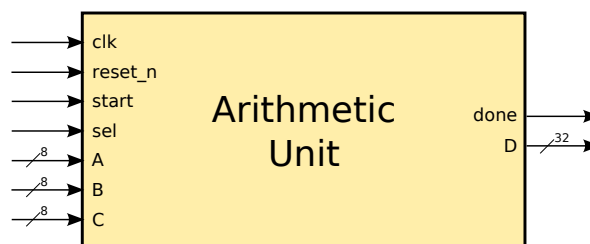
- The **clk** input signal synchronizes the system. It is used to clock the pipeline registers.

- The **reset_n** input signal initializes the pipeline registers of the arithmetic unit (active low).

- The **start** input signal indicates when the arithmetic unit has valid inputs.

- The **sel** input signal indicates the configuration mode of the arithmetic unit. When it is 0, the unit should be configured to implement f_0 and when it is 1, the unit should be configured to implement f_1.

- The **A**, **B** and **C** input signals are the a, b and c input variables of the functions f_0 and f_1.

- The **done** output signal is set to 1 when **D** holds a valid result.

- The **D** output signal is the result of the arithmetic expression d.

4.1 Exercise

4.1.1 Combinatorial

Implement a combinatorial version of the arithmetic unit.

- Open the Quartus project.

- Open the `arith_unit.vhd` file. The given entity has the interface of the arithmetic unit previously described.

- In the `arith_unit.vhd` file, create a `combinatorial` architecture. In the architecture body, implement the combinatorial version of the arithmetic unit. For the 16-bit multiplier, you can either use four 8-bit multipliers or implement a new multiplier with a larger *adder tree*. As this implementation is combinatorial, you must ignore the **clk** and **reset_n** inputs, and connect the **start** input directly to the **done** output.

- Set the output clock frequency of the PLL to 10 `MHz` (see the Section 5 for instructions).

- Verify that your system is working (see the Section 6 for instructions).

- The timing analysis guarantees that your design will work under a certain maximum frequency, but it might still work if you overclock it.

- Manually, find the maximum frequency supported by your system (make sure you keep this value for further comparisons). As a first attempt, you can double the clock frequency.

4.1.2 1-Stage Pipeline

- In the `arith_unit.vhd` file, create a `one_stage_pipeline` architecture.

- Copy the VHDL code of the `combinatorial` architecture, and modify it by adding a register stage.

- In the `pipeline_lab.vhd` file, on lines 75-78, make sure the `one_stage_pipeline` architecture is selected (comment out all the others).

      ArithmeticUnit : entity work.arith_unit(*architecture name*) port map.

- Find the maximum frequency supported by your design.

- Compute the latency of your arithmetic unit in *ns*.

- Compare your results with the other architectures.

4.1.3 2-Stage Pipeline I

- In the `arith_unit.vhd` file, create a `two_stage_pipeline_1` architecture.

- Copy the VHDL code of the `combinatorial` architecture and modify it by adding two register stages.

- For this first version of the 2-stage pipelined arithmetic unit, do not insert registers inside the multipliers.

- Modify the `pipeline_lab.vhd` file to load the new architecture.

- Find the maximum frequency supported by your design.

- Compute the latency of your arithmetic unit in *ns*.

- Compare your results with the other architectures.

4.1.4   2-Stage Pipeline II

- In the `arith_unit.vhd` file, create a `two_stage_pipeline_2` architecture.

- Copy the VHDL code of the `combinatorial` architecture and modify it by adding two register stages.

- For this second version, you are allowed to place your register stages inside your multipliers. See if you can improve the speed of your design.

- Modify the `pipeline_lab.vhd` file to load the new architecture.

- Find the maximum frequency supported by your design.

- Compute the latency of your arithmetic unit in *ns*.

- Compare your results with the other architectures.

5   Setting the Output Clock Frequency of the PLL

In this section, we use a PLL (*Phase-Locked Loop*) to set the clock frequency of the arithmetic unit. The PLL can be configured to multiply or to divide the frequency of a clock signal. Modify the output clock frequency using the following steps.

- In Quartus, select Tools >IP Catalog.

- Enter `ALTPLL` in the IP Catalog search box and double-click on it. The configuration window of the PLL should appear.

- Enter `../vhdl/PLL.vhd` as the IP variation file name, then press on **OK**. You will have to overwrite the current PLL file each time you want to modify its settings.

- Choose the **Parameter Settings** >**General/Modes** tab.

    - Set the frequency of the **inclk0** input to 50 `MHz`.

- Choose the **Parameter Settings** >**Inputs/Lock** tab.

    - Uncheck the **Create an 'areset' input to asynchronously reset the PLL** checkbox.
    - Uncheck the **Create 'locked' output** checkbox.

- Choose the **Output Clocks** >**clk c0** tab.

    - Set the desired frequency in the field **Enter output clock frequency**.

    [*Hint: If the frequency that you entered is red, then the program cannot set the multiplier or the divider to get the desired frequency. In this case, copy the frequency value that the program finds in the **Actual Settings** field and use it in the **Requested Settings** field.*]

- Press **Finish** twice.

- When asked to add the Quartus II IP file to the project, press **No**, as it is already done in the template we provide.

- Recompile your project.

6   Verify Your Design

A verification system is provided to verify your design. It sends some input values to your arithmetic unit, writes in memory the returned results, and then compares these results with the expected results.

- To run the verification, compile your Quartus project and program your FPGA.

- You should see the text `RDY` on the LEDs.

- First, the verification system sends some values to the arithmetic unit and stores the results in memory.

- Press **Button 0** to proceed (i.e., the push button on the right).

- If this step finished correctly, you should see the text `OK` on the LEDs.

- Next, the verification system compares the results of the arithmetic unit with the expected results.

- Press **Button 1** to proceed (i.e., the push button next to **Button 0**).

- If all the results are correct, a check mark ✓ appears on the LEDs; otherwise, a cross × is displayed. Note: Incorrect results could be due to a fast clock (try reducing the clock frequency).

7   Submission

Submit all designed multipliers in a single `multiplier.vhd` file, as well as the `arith_unit.vhd` file with the four architectures from Section 4.