

# Final Project

## Design of a Simple Sequential Processor

### Introduction

The purpose of this lab is to design a simple sequential processor utilizing an arithmetic-logic unit (ALU) that can perform various arithmetic operations. The ALU will support four different operations: addition, multiplication, subtraction, and bitwise “AND”. For simulation and verification, the functionality will be tested sequentially on consecutive clock cycles.


### Part 0 - Preparation of logic designs

In order to be able to focus on the Simulink designs during this lab session, you need to prepare your designs on paper which will save you even more work during the lab session. The following designs and logic blocks (among others) will be used in this project. Some of the blocks were designed in previous labs or discussed in class. You may want to consult the lecture notes posted on Blackboard for design ideas of any blocks that may be unfamiliar to you.

- An 8-bit adder
- An 8-bit subtractor
- An 8-bit multiplier
- An 8-bit bitwise “AND” operator.
- A 4-to-1 multiplexer.
- An 8-bit 4-to-1 multiplexer.
- An 8-bit register

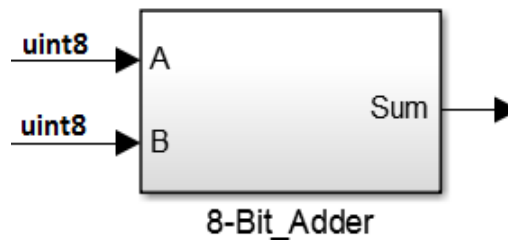
The following sections of this lab manual provide more details about the exact interface for these components, as well as some design strategies and testing. **Please read through the complete manual before starting to work on this lab.**

#### Simulink Design Settings:

1. For each of your Simulink designs, set the solver to a discrete solver. Open the configuration settings in menu entry *Simulation* → *Model Configuration Parameters*. On the solver options, select:
  - a. Type: Fixed-step
  - b. Solver: Discrete (no continuous states)
  - c. Fixed-step size: auto
2. On your scopes, click on the *Settings*  icon, click on the *Logging* tab, and ensure that the *Limit data points to last* option is unchecked.
3. Your outermost inputs and constants should be set with the appropriate **data types** (*Boolean*, *int8*, etc) and a **sampling time** of **1**. The rest of the inputs and outputs in your subsystems can either be left to inherit the data type and sampling time, or set to the appropriate data type if necessary. You can also use the *Data Type Conversion* block if needed.

### Part 1 - The 8-bit adder

Design an 8-bit adder (or use your Lab6 design) that can correctly add 8-bit positive integers (*uint8*) in Simulink. Put your adder design into subsystem *8-Bit\_Adder*, with the interfaces as shown below.



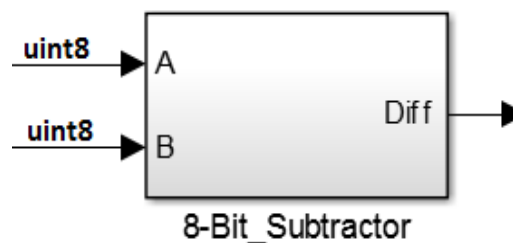
#### Assignment 1

Connect two *Constant* blocks set to *uint8* output data type and *sample time = 1* to the two inputs of your adder design, and connect a *Display* block at the output. Simulate and verify the correct behavior with various *uint8* inputs in the Constants. Assume that the sum is less than or equal to 255. You can ignore the final carry out bit.

- Include captures of sample simulations of your design in your report.
- Open the adder subsystem and capture the overall 8-bit adder design to include in your report.

### Part 2 - The 8-bit subtractor

Design an 8-bit subtractor that can correctly subtract 8-bit integers (*uint8*) in Simulink. Put your subtractor design into subsystem *8-Bit\_Subtractor*, with the interfaces as shown below.



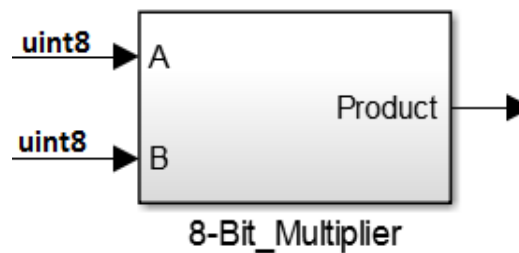
#### Assignment 2

Connect two *Constant* blocks set to *uint8* output data type and *sample time = 1* to the two inputs of your subtractor design, and connect a *Display* block at the output. Simulate and verify the correct behavior with various *uint8* inputs on the Constants.

- Include captures of sample simulations of your design in your report.
- Open the subtractor subsystem and capture the overall 8-bit subtractor design to include in your report.

### Part 3 - The 8-bit binary multiplier

The next step is designing the 8-bit binary multiplier on Simulink. The multiplier should have the following interface:



Here are some tips for your Simulink design:

1. See design options on the attached “Combinational Multiplier” pdf document on Blackboard, in addition to the Lecture Notes on Binary Arithmetic.
2. Create a new model on Simulink and save it as multiplier.slx.
3. Copy and paste the binary adder design from the previous lab assignment, and replicate it as many times as you need for the addition of all intermediate products.
4. When inserting *In1*, *Out1*, or *Constant* components, follow the same guidelines presented in the previous section regarding the sample time and data type properties.
5. Notice that the largest number that can be produced as a result of multiplying two 8-bit numbers is a 16-bit number. However, we will only connect the lower 8-bits to the ALU output in later section, in order to match the maximum size of our ALU output. This means that the 8 most significant bits of the product will be simply discarded, and we are only interested in product terms that are equal to or less than 255 (with 8-bits).
6. Once you finish placing and connecting all components, select them all, right-click on the selection, and choose option *Create Subsystem from Selection*. Rename the subsystem as *8-bit multiplier*. Your 8-bit multiplier should have the interface similar to the figure above.
7. Simulate your design by assigning fixed values (with *constant* blocks) to its inputs, and reading the value at the output with a *Display* block. Follow the same approach presented in previous sections for the validation.

#### Assignment 3

Design the 8-bit binary multiplier following the steps presented above.

- a) Take a screenshot of your detailed design and include it in your report.
- b) Navigate to the higher-level diagram of your design, containing the *constant* blocks and the *Display*. Run a simulation, and take a screenshot that shows the output of the *Display*, as well as the value of the input constants. Add this screenshot to your report. Try with different values and verify correct multiplication operations.

### Part 4 - The bitwise “AND” operator

Design an 8-bit bitwise “AND” operator. This is the last operation supported by the ALU. This is a circuit that takes two 8-bit operands  $A$  and  $B$ , and returns a result  $Y$  in which each individual bit position is calculated as the logic product of the bits at the same positions in the source operands. This means that  $Y_0 = A_0 \cdot B_0$ ,  $Y_1 = A_1 \cdot B_1$ , etc.

This is the interface of an 8-bit bitwise “AND” operator:



Remember the following design tips introduced in previous lab assignments:

- For any input port using an *In1* component, select a sample time of 1 and a well-defined type, such as *boolean* or *uint8*. Do not leave the type set to the default *Inherit/auto* value.
- Keep the same consideration for *Constant* blocks. Adjust both sample time and data type as needed. This applies for constants used for simulation purposes, as well as for constants used to hardcode the value of circuit inputs, such as the fixed carry-in input set to “0” for the binary adder, etc.
- If you get any type incompatibility error between *boolean* and *ufix1* types, use a *Data Type Conversion*, as done in the implementation of the *BitSlice* module in a previous lab assignment. This error should not occur as long as all your *BitSlice* modules include the *Data Type Conversion* blocks in their implementation.

#### Assignment 4

Implement the bitwise “AND” logic block on Simulink.

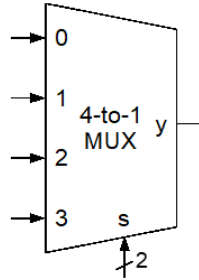
- Take a screenshot of your detailed design and include it in your report.
- Simulate your design by feeding the circuit with two 8-bit constants whose bit combinations explore all combinations of 0s and 1s in the bitwise operation. Add the appropriate constant blocks and scopes. Take a screenshot that demonstrates the correct behavior of the circuit, and add it on your report. The screenshot should show the constant values and the result of the circuit.

### Part 5 - The (1-bit) 4-to-1 multiplexer

A sub-component of the ALU is a 1-bit 4-to-1 multiplexer, or simply 4-to-1 MUX. This component will help us select the result of one specific arithmetic operator, based on an operation code encoded as a binary value.

The 4-to-1 MUX has 4 data inputs  $x_0 \dots x_3$  (simply labeled by their indexes on the diagram below), a 2-bit selector input  $s$ , and a 1-bit output  $y$ . The multiplexer acts as a configurable switch, where the binary

value provided in the selector, ranging between 0 and 3, indicates the index of the input that will be internally connected into the output. The 4-to-1 MUX has the following interface:



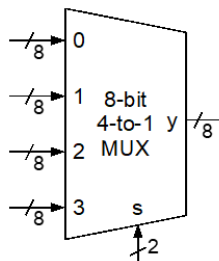
### Assignment 5

Design the 4-to-1 MUX on Simulink.

- Take a screenshot of your detailed design and include it in your report.
- Simulate your design by assigning different constant values to the inputs, and a scope at the output. Run two simulations with two different values for the selector  $s$ , one that makes output  $y$  take a value of 0, and one that makes  $y$  take a value of 1. Take a screenshot for each of them and add them to your report. The screenshots should capture the constant values, as well as the value shown in the scope. Explain the values you observe.

### Part 6 - The 8-bit 4-to-1 multiplexer

The 4-to-1 MUX designed before is limited to the selection of multiple 1-bit data values. However, the operands and the result of our ALU are 8 bits wide. We need to extend our multiplexer to build an 8-bit 4-to-1 MUX. The operation of this device is similar to the basic 4-to-1 MUX, but each data input  $x_i$  is composed of 8 bits instead of just 1. The output  $y$  is also composed of 8 bits. The 2-bit input  $s$  behaves as before, indicating the index of the 8-bit input that should be driven into the 8-bit output of the MUX. Here is the logic block for an 8-bit 4-to-1 MUX:



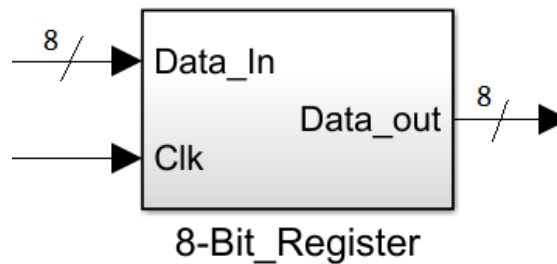
### Assignment 6

Design the 8-bit 4-to-1 MUX on Simulink. You can create this module by combining eight 1-bit 4-to-1 MUXs, as designed in the previous section. At each of these design steps, remember to create a “sealed” logic block, or using Simulink's terminology, a subsystem with a clearly defined interface.

- Take a screenshot of your detailed design and include it in your report.
- Simulate your design by assigning different 8-bit constant values to the inputs, and a scope at the output. Run two simulations with two different values for the selector  $s$ . Take a screenshot for each of them and add them to your report. The screenshots should capture the constant values, as well as the value shown in the scope. Explain the values you observe.

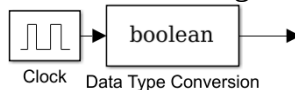
### Part 7 - The 8-bit register

A register is a configuration of  $n$  flip-flops used to store an  $n$ -bit number. In this section, we will design an 8-bit parallel register to store our output value after each operation. Since parallel registers are designed with flip-flops synchronized to the same clock, the 8-bit register will need a clock input as shown in the figure below. For parallel registers, data is loaded in parallel on the rising edge of the clock or the falling edge of the clock, and accessed in parallel at any time. The *initial value* (default value before you store anything) of the register is automatically set to '0'.



Here are some tips for your Simulink design:

1. On the Simulink library, search for “flip flop” and use the ‘*D Flip-Flop*’. Each flip flop stores 1 bit.
2. Connect all clock inputs to an *In1* input component and rename it *Clk*.
3. Connect all clear inputs to one *Constant* block set to *Boolean* output data type and *Sample time* = 1. Set the value of the Constant to 1. The clear input is *active low*, so a value of 1 will not clear the register.
4. Connect the 8-bit output from the Flip-Flop’s output *Q* using a *Bit Concat* block as done in previous designs. Ignore output *!Q* (we do not need it for this design).
5. Complete the design with any other necessary blocks (e.g. *Bit Slice*) and put it in a subsystem with the interface shown above.
6. Connect a *Constant* block set to *uint8* output data type and *Sample time* = 1 to the *Data\_In* input.
7. Connect a clock signal with a *period* of 1 to the *Clk* input through a *Data Type Conversion* block.



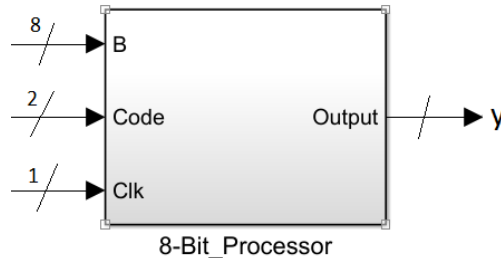
### Assignment 7

Design the 8-bit register with the tips above.

- a) Take a screenshot of your detailed design and include it in your report.
- b) Simulate your design by assigning different constant values (0-255) to the Data input, and a scope at the output. Run two simulations for 10 cycles with two different input values. Take a screenshot for one of them add it to your report. The screenshot should capture the scope showing the clock input, as well as the output value. Verify that the default output is '0' for the first cycle, and your input value for the rest of the cycles.

## Part 8 – Putting it all together: The 8-bit Processor

The last step in our hardware design consists in building the complete 8-bit Processor, based on all logic blocks created in this and the previous labs, including the 8-bit adder, 8-bit subtractor, 8-bit multiplier, 8-bit bitwise “AND”, 8-bit 4-to-1 MUX, and the 8-bit register. The following logic block shows the interface for the 8-bit Processor:



As you can see, the 8-bit Processor takes one 8-bit input  $B$ , and provides its result in output  $y$ . A 2-bit input code determines the operation performed by the ALU, as follows:

- If code is 00, the ALU performs an addition of the current output with input  $B$  i.e.  $(y + B)$ .
- If code is 01, the ALU performs a multiplication of the current output with input  $B$  i.e.  $(y * B)$ .
- If code is 10, the ALU subtracts  $B$  from current output i.e.  $(y - B)$ .
- If code is 11, the ALU performs a bitwise “AND” operation  $(y \& B)$ .

Design the system with the logic blocks discussed above and any other necessary Simulink blocks.

### Design Verification:

1. Connect a *Constant* block set to *uint8* output data type and *Sample time* = 1 to the  $B$  input. Set the *value* to 11 for the first test case.
2. Connect a clock signal with a *period* of 1 to the  $Clk$  input through a *Data Type Conversion* block as shown in Lab 10.7 step 7.
3. The 2-bit code will be generated by an HDL Counter set to *Free running* and with *Word length* = 2. The counter should be able to count from 0 to 3 generating the desired codes above. When the counter gets to 3 it then resets to 0 and starts again.
4. Connect a Scope with 3 inputs connected to: the clock input, HDL Counter output (generated code), and the processor output.

### Assignment 8

Running the simulations in Simulink.

- a) Take a screenshot of your detailed design and include it in your report.
- b) Simulate your design for 10 Simulink cycles. The counter should count from 0 to 3 two and a half times generating the following code sequence: 00, 01, 10, 11, 00, 01, 10, 11, 00, 01. With the initial default output value of '0' and the given input  $B$ , verify that your design performs the correct sequence of operations as shown on the output on the Scope.

Run several simulations with different input  $B$  values and verify the correct behavior. For each simulation, take a screenshot that clearly shows the values of the input  $B$  and the results for at least 2 test cases. Add the screenshots to your report, and explain the output.

**Final Report Submission**

- **Submit a zipped folder with your lab report (PDF) and your final Simulink design.**

You should follow the lab report outline provided on Blackboard. Your report should be developed on a word processor (e.g., OpenOffice), and should include graphics when trying to present a large amount of data. Your report should include all your responses to the questions asked, screen captures of all your subsystems and the contents of the subsystems, and explain any settings for the various blocks wherever necessary. Submit all these in a zipped folder (one per team).