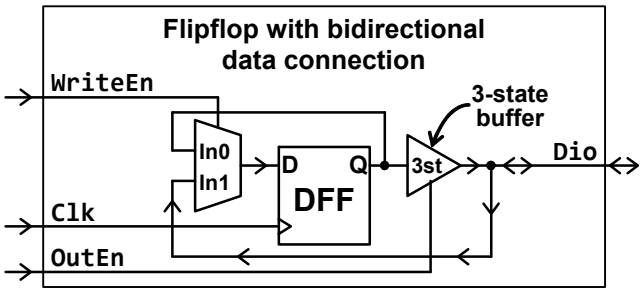
EE342 - Digital System Design

Laboratory Experiment - 7
Bidirectional Data Transfer

A data line can be used for input and output by using 3-state (or tri-state) output buffers. As a typical example, the data flipflop in the following circuit has a single data connection that can transmit data in both directions.



Reading from flipflop: `OutEn` input is set to **1** to enable the 3-state buffer. Flipflop output drives `Dio` line, and `Dio` can be an input for other circuits.

Writing to flipflop: The 3-state buffer is disabled when `OutEn` input is **0**. Another circuit can drive `Dio` line. If `WriteEn` input is set to **1** then the flipflop stores the data on `Dio` at the rising `Clk` edge.

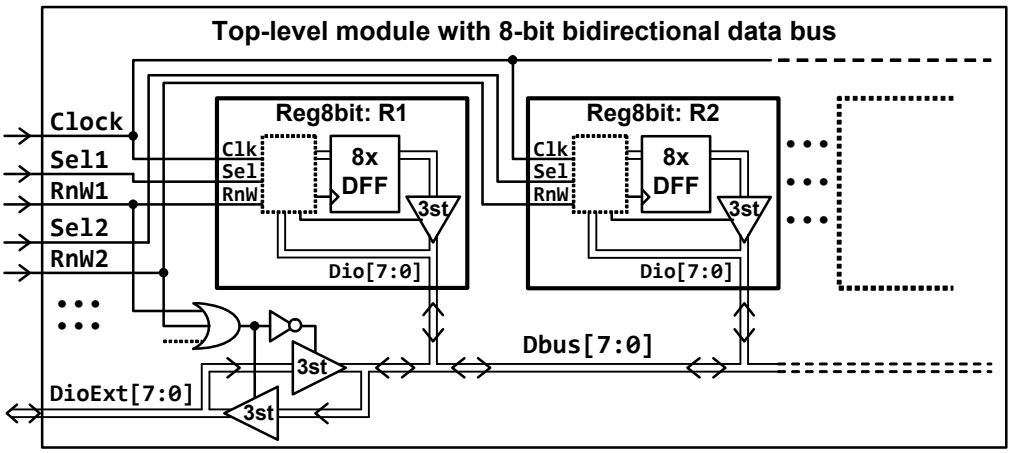
A Verilog implementation of this circuit is given by the following module.

```
module DFF3st(Clk, WriteEn, OutEn, Dio);
input Clk;      // clock input to trigger register storage
input WriteEn;  // enables write operation
input OutEn;    // enables 3-state output buffer for read operation
inout Dio;      // connection for data input and output

reg  FFstore;   // storage register

always @(posedge Clk)
  if (WriteEn == 1'b1)
    FFstore <= Dio;
  else
    FFstore <= FFstore;

assign Dio = (OutEn == 1'b1) ? FFstore : 1'bZ;

endmodule
```

`Dio` is the bidirectional data connection in the port list, and it is declared with the `inout` keyword. The 3-state output buffer is inferred by the conditional assignment that sets `Dio` to `1'bZ` when `OutEn` is **0**. The constant value of `1'bZ` means ***high-Z*** (i.e. *high-impedance*) output, indicating that `Dio` output is disabled when `OutEn` is **0**.

An 8-bit bidirectional data bus will be implemented in this experiment as shown in the following block diagram. The top-level module contains several register modules that can be accessed through the 8-bit internal data bus (`Dbus[7:0]`).

Bidirectional data pins (**DioExt[7:0]**) provide external access to **Dbus[7:0]** to read and write register contents. Either **DioExt[7:0]** pins or one of the register modules can drive **Dbus[7:0]**. The 3-state buffers are enabled according to **RnW…** control signals.



The Verilog code of the top-level module is given below. This code is also provided as a separate file labeled as **Exp7_Top.v**.

```
module Exp7Top(Clock, Sel1, RnW1, Sel2, RnW2, Sel3, RnW3, DioExt);
input Clock;  // clock input for all modules
input Sel1, Sel2, Sel3;  // module select inputs
input RnW1, RnW2, RnW3;  // module read/write control inputs
inout [7:0] DioExt; // I/O connection to external data pins

tri  [7:0] Dbus;  // internal data bus connecting all modules and
                  // DioExt[7:0] external data pins
Reg8bit R1(Clock, Sel1, RnW1, Dbus);  // register modules
Reg8bit R2(Clock, Sel2, RnW2, Dbus);
Reg8bit R3(Clock, Sel3, RnW3, Dbus);

// DioExt[7:0] drive Dbus[7:0] while writing to a register module.
assign Dbus[7:0] = ( (RnW1 | RnW2 | RnW3) == 1'b0 ) ?
                   DioExt[7:0] : 8'bZ;
// Dbus[7:0] drive DioExt[7:0] while reading from a register module.
assign DioExt[7:0] = ( (RnW1 | RnW2 | RnW3) == 1'b1 ) ?
                     Dbus[7:0] : 8'bZ;
endmodule
```

A **tri** net cannot be simultaneously driven by more than one buffer. Therefore, only one of **RnW1**, **RnW2**, and **RnW3** control inputs can be high at any time. **Logic contentions** occur when a buffer drives a signal node to high level while another buffer is pulling the same node to low level. If there is a logic contention then the signal level is undefined, and the simulator gives warning messages reporting these events.

Cyclone series FPGAs manufactured by Altera has 3-state drivers for external pins, but the internal FPGA hardware does not support 3-state connections. Quartus II synthesis tool replaces internal 3-state buffers with the logic operations that implement **tri** nets according to the buffer enable signals. Verilog compiler will give warning messages indicating the implementation method of **Dbus[7:0]** nets when the above top-level module is compiled.

Preliminary Work

1. Read the section titled "**Tri-state Drivers**" in the **Logic Circuits Review** lecture notes.

2. Write a Verilog module that implements an 8-bit register with bidirectional data connection. The module header and connections are as follows.

```
module Reg8bit(Clk, Sel, RnW, Dio);
input Clk;  // clock input to trigger register storage
input Sel;  // select input that enables read/write operations
input RnW;  // read/write control input;
            // 0=> store data input at Dio[7:0] if Sel is 1
            // 1=> enable output to Dio[7:0] if Sel is 1
inout [7:0] Dio;  // connection for data input and output
```

Procedure

1. Create a new project for this experiment, and compile the provided top-level module with the register module you wrote in the preliminary work.

2. Create a simulation waveform file as follows.
- Set "**End Time**" of the waveform file to **2 µs**.
- Set "**Grid Size**" of the waveform file to **25 ns**.
- Include all input/output pins and all module registers.
- Generate a **10 MHz** clock as the **`Clock`** input.
- Select hexadecimal radix for all 8-bit waveforms (right-click on the waveform name and select **Properties**).
- Set the top-level module inputs to perform an operation in each clock cycle:
  `op1:` write **`0x11`** from **`DioExt`** to **`R1`** register (**`R1`** instance of **`Reg8bit`**)
  `op2:` write **`0x22`** from **`DioExt`** to **`R2`** register
  `op3:` write **`0x33`** from **`DioExt`** to **`R3`** register
- Save the simulation waveform file.

3. Open simulation tool and select the created waveform file as the simulation input.
- Check the "**Overwrite simulation input file with simulation results**" option.
- Start the simulation and display updated waveform file with simulation results.
- Check the simulation messages for "**logic contention**" warnings. Correct the input waveforms or debug your program to fix the logic contentions.

Quartus Note: Simulator adds the output waveforms for bidirectional I/O pins labeling each output with "**`~result`**" extension. **`DioExt`** waveform is used to enter input values, and **`DioExt[.]~result`** waveforms display the status of output drivers. "**`~result`**" waveforms can be grouped as a single hexadecimal waveform to observe the outputs easily on the waveform display:

- Select all eight **`DioExt[..]~result`** nodes in the waveform file
- Right-click and select "**Grouping**" option.
- Enter "**`DioExtOut`**" as the group name and select **hexadecimal** radix.
- Right-click again on the "**`DioExtOut`**" label and select "**LSB on Top, MSB on Bottom**" option for "**Group and Bus Bit Order**"..

4. Add the following operations to the simulation waveform file after the write operations entered in **step-2**.

`op4:` read **`R1`** register at **`DioExtOut`** pins
`op5:` read **`R2`** register at **`DioExtOut`** pins
`op6:` read **`R3`** register at **`DioExtOut`** pins
`op7:` copy **`R3`** register contents to **`R1`** register
`op8:` copy **`R2`** register contents to **`R3`** register

Verify the operations by checking "**`DioExtOut`**" signals and module registers.

5. Write an ***accumulator*** module that has the same connections as the **`Reg8bit`** module you wrote before. The accumulator read/write operations are as follows.

`RnW=0 :` add data at **`Dio[7:0]`** to the accumulator register
`RnW=1 :` enable output from accumulator register to **`Dio[7:0]`**, and reset the accumulator register in the next clock cycle.

Instantiate the accumulator module in the top-level module, add the required control inputs, and modify the 3-state buffer statements as necessary. Compile the project and debug the code, if necessary.

6. Include the accumulator control inputs in the simulation waveform file. Clear the register operations added in **step-4**, replacing them with the following operations:

`op4:` add **`R2`** register contents (**`0x22`**) to the accumulator
`op5:` add **`R3`** register contents (**`0x33`**) to the accumulator
`op6:` add **`0x44`** from **`Dio[7:0]`** to the accumulator
`op7:` copy accumulator contents to **`R1`** register

Verify the accumulator operations (the accumulator should be reset after the sum, **`0x99`**, is copied to **`R1`** register).

7. Add the following operation right after reading accumulator register in **`op7`**.

`op8:` add **`0x88`** from **`Dio[7:0]`** to the accumulator

This operation attempts to add a number while the accumulator is in the reset cycle. Modify the accumulator module so that it will work properly in both cases, whether it resets or adds a number right after a read operation.