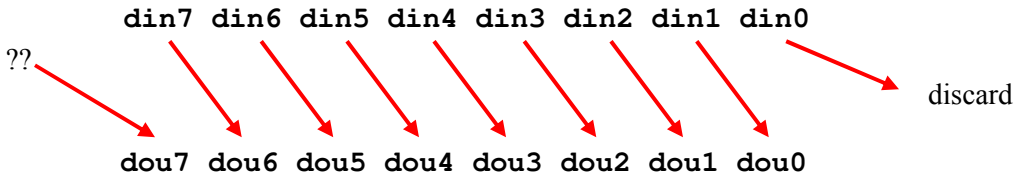
Lab 5 – Combinational Shift Logic: Up/ Down/Arithmetic/Circular/Barrel

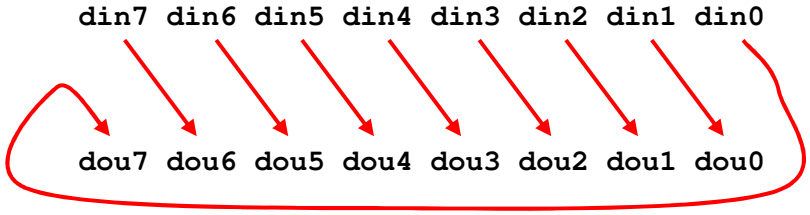
Introduction

Shifting operations are commonly used in digital design. Just as shifting in base 10 is multiplication/division by 10, shifting in base 2 (binary) is multiplication/division by 2. Also the shift operation is important when converting data between parallel and serial, for transmission of multi-bit data serially over a single channel, or a single wire bus. While you will be designing static shift operations (combinational logic) in this lab, these operations are typically implemented with registers, triggered by clock signals, i.e., one shift per clock cycle.



When shifting up or down, what is placed in the vacated bit position is important. The simplest operation is to replace the vacated bit with zero. When shifting up, this becomes a multiplication by 2 – and the result is obviously an even number. When shifting down and replacement with zero, we have division by 2 – but only for an unsigned number. If the number is signed, then the most significant bit is 0 or 1 depending on the sign. The vacated bit must keep the value that was shifted out of the bit. This sign replacement operation, also known as sign extension, is called an arithmetic shift. I will call this a linear shift operation.

In contrast to a linear shift, a circular shift replaces the vacated bit with the bit that would have been discarded in the shift operation, i.e., a rotation. Circular shifting an n-bit word by n shifts leaves the word unchanged. Sometimes the circular shift has an advantage when transmitting a data word serially by shifting the bits in a storage register, as when the transmission is complete (n-shifts) the original data is still intact.



Barrel shifting is shifting by more than one bit. While this seems trivial, when we start looking at synchronous or clocked operations, in some applications it is important to be able to shift by more than one bit in a single clock cycle. Barrel shifting can be either circular or linear.

The Verilog implementation of the shift operations can be one of several:

(some examples below using: `logic [7:0] data;`)

- explicit bit manipulations using the concatenation operator: `{1'b0,data[7:1]}`
- using the unsigned shift operator: `data >> 1`
- using the signed shift operator: `data >>> 1`
- using divide by or multiply by 2^n operator: `data/2` (be careful of the signed/unsigned declarations and defaults – there are several Verilog/SystemVerilog rules to consider)

Detailed Specification:

In this lab you will design a number of 8-bit shifter operations input data defined by switches 0-7. The functionality of the particular shift operation is controlled by the top push button (***key0***) and switches 8-9 and on the DE10 board. The results of the operation will be displayed on the 8 LEDs (0-7).

The shift operations are defined in the table below:

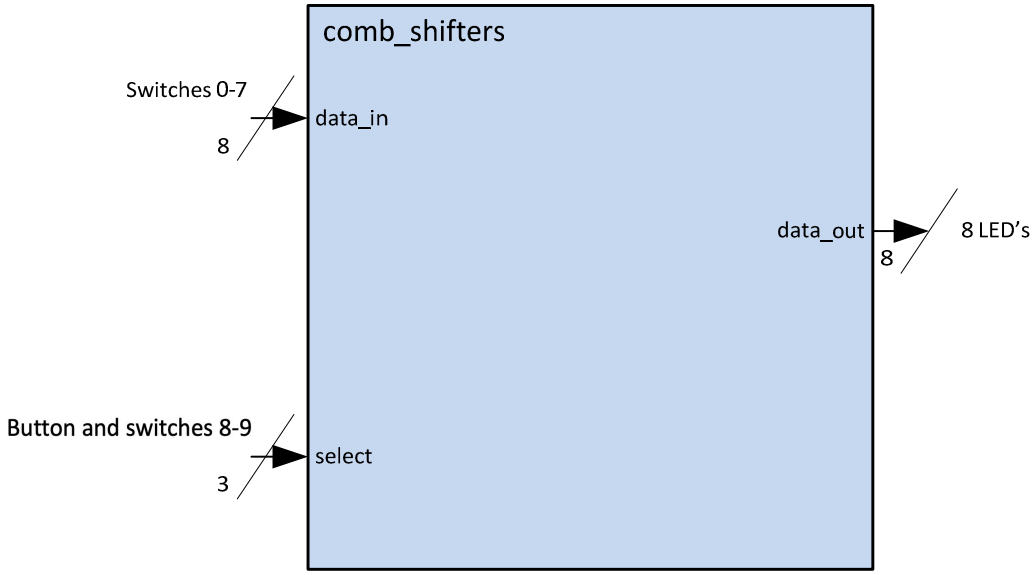
{sw[9:8], key0}	Operation
3'b000	no operation, output = input
3'b001	shift 1 bit left, pad with 0
3'b010	shift 1 bit right, pad with 0
3'b011	circular shift 1 bit left
3'b100	circular shift 1 bit right
3'b101	arithmetic shift 1 bit right, pad with sign
3'b110	barrel circular left shift by 3 bits
3'b111	barrel arithmetic right shift by 5 bits

Although only 3 inputs are used in this problem, others can be added to investigate additional arithmetic operations.

Design/Module

In this design there is only one design module that you need to develop, ***comb_shifters***. As illustrated in the block diagram below, it has 11 inputs: 3-bit ***select*** (wired to the top push button and switches 8-10), 8-bit ***data_in*** (wired to switches 0-7), and 8-bit ***data_out*** (wired to LEDs 0-7).

This module is instantiated in the top level i/o wrapper, **DE10_LITE_Temple_Top.sv**, for implementation. It is also instantiated in the top level test bench, **tb_comb_shifters.sv**, for simulation/verification. The test vectors for this design are contained in **tb_comb_shifters.txt**, with 2048 vectors.



As usual, the Quartus Settings file (**lab5_top.qsf**) contains the mapping and signal naming between the pins on the FPGA and the signal names used at the top level i/o wrapper (**DE10_LITE_Temple_Top.sv**). Do not modify these files for this lab.

Design step:

- Complete the design for ***comb_shifters*** based on the above specification.

Simulation/Verification

The testbench, **tb_comb_shifters.sv**, will read the text file, **tb_comb_shifters.txt**. Neither of these files should be edited.

Verification step:

- There is one simulation command file (**comb_shifters.m_sim**). Verify your ***comb_shifters*** design by right clicking on the command file and Run. Modify your design as needed.

This simulation will test all 2048 possible input combinations. Your design **MUST** pass the simulation, as you will not have time to go through the entire 2048 button/switch combinations in

hardware. The instructor will be simulating and checking your design for mismatches after your report is due.

Synthesis

The top level i/o wrapper is **DE10_LITE_Temple_Top**. Generate a binary file as you did in past labs
(right-click and Run on: **lab5_top.qsf**). When successful, you will generate the file: **lab5_top.sof**. Copy it to your shared folder to load and run on your DE10 board. Select some random input data values on the 8 switches and observe the binary result displayed on the LED's. Then change the button and select switches see if the shifting operation makes sense as per the specifications in the previous table. Use both positive and negative binary patterns (msbit). Include the results of your selected patterns in your lab report. (You would need 2048 data patterns to verify all of them!)

Extras to try

a) Try different shifting operations by modifying your design, i.e., barrel shifting of different shift values.
b) For practice on instantiation, wire the least significant 4 LED bits to drive one of the digits of the seven segment decoder (recall last lab).

Useful Verlog rules and constructs for signed operation

In version *Verilog 2001* many signed operations were added to the language syntax. Some of those are listed below.

- The default of a **`reg`** or **`wire`** is unsigned. To make the signal bus signed, add the **`signed`** keyword. Example:

```
wire signed [7:0] data_in;
```

- **The default arithmetic operations are unsigned**. However, when working with mixed signed and unsigned signals, the rule is **that if any operand in an expression is unsigned the full operation is considered to be unsigned**. So to get a signed operation by using the operators described below, **all** operands must be signed. This can be a source of errors.
- The **default** base of literal numbers is **unsigned decimal**. Thus, they are zero extended to the size needed. To override/control this, use the Verilog prefixes. The syntax is:

```
<+ or -><size>'<signed><base><value>
```

`<+ or ->`	(optional) the sign of the number
`<size>`	(optional) number of bits (default=32)
`<signed>`	(optional) s for signed, nothing for unsigned
`<base>`	**`b, o, d`** or **`h`** for binary, octal, decimal or hexadecimal, respectively

Example: **`assign data_in = -8'sd98;`**

- *Type casting* operators are available: **`$signed()`** or **`$unsigned()`**. Example, if **`data_out`** is signed and **`data_in`** is unsigned:

```
assign data_out = $signed(data_in);
```

- For signed right shifting, a new operator was introduced: **`>>>`** Thus, **`>>`** does an unsigned shift, and **`>>>`** does the signed (sign extension) shift. While signed and unsigned left shifting are identical (0 fill), a signed left shift operator was added for consistency: **`<<<`** Example (remember, for this operation to work **both** data_out and data_in must be declared as ***signed***):

```
assign data_out = data_in >>> 3;   // signed divide by 8
```