Basics of SystemVerilog

SystemVerilog is a hardware description language (HDL) used to describe Register-Transfer Level (RTL), which models digital logic as data flows between hardware registers. Synthesis tools convert RTL into a hardware implementation composed of combinational and sequential logic elements. It is an extension of Verilog, an older HDL.

Hardware variables (wires and registers)

Wires are just nets, which should have values assigned to them combinationally. Registers represent hardware registers, which have values assigned to them sequentially. In SystemVerilog, both wires and nets are defined with the `logic` keyword. Once a `logic` is defined, the compiler will figure out whether it should be a wire or a register.

```
logic x;
logic y;
```

Variables can be multi-bit as well. The syntax is:

```
logic [START_BIT:END_BIT] VARIABLE_NAME;
```

SystemVerilog does not enforce an endianness. `START_BIT` can be greater than `END_BIT` (little-endian), or `START_BIT` can be less than `END_BIT` (big-endian). However, most applications are little-endian.

Also, neither `START_BIT` nor `END_BIT` has to be `0`.

Part select

To select a single bit of a multi-bit signal, use the following syntax:

```
VARIABLE_NAME[BIT]
```

To select multiple bits of a multi-bit signal, use the following syntax:

```
VARIABLE_NAME[START_BIT:END_BIT]
```

Unfortunately, there is no easy way to flip the direction of endianness of a multi-bit signal. The easiest way to accomplish this is with a `for` loop.

Arrays

Defining combinational logic

There are two ways to write combinational logic in SystemVerilog. One way is using the `assign` keyword.

```
assign z = x + y;
```

The second way is to write a statement (or series of statements) within an `always_comb` block.

```
always_comb begin
    z = x + y;
end
```

This way, you don't need to write the `assign` keyword before all combinational logic statements.

Defining flip-flops

The `always_ff` block is used to define flip-flops. This following code creates a flip-flop that samples `d` on the positive edge of `clk` and has an output of `q`.

```
always_ff @(posedge clk)
    q <= d;
end
```