SystemVerilog Tutorial

Introduction

SystemVerilog is a hardware description language that improves upon and extends Verilog. For the purposes of this course, consider SystemVerilog a superset of Verilog. That is, **every Verilog file you've ever written is also a valid SystemVerilog file**, meaning all documentation and tutorials pertaining to Verilog can be applied to SystemVerilog as well. The file extension for SystemVerilog files is `.sv`, as opposed to Verilog's `.v`. Both Quartus Prime and Modelsim support the language.

The purpose of this document is to acquaint you with *three* language features of SystemVerilog that make noticeable improvements over Verilog in terms of ease of coding and debugging your hardware. The last section contains example source code for a complete module that utilizes all three features, for your reference.

1 Enumerations

One of the more annoying and error-prone parts of creating Finite State Machines in Verilog was defining the encodings for each state. For example:

```
localparam
  S_FIRST = 2'd0,
  S_SECOND = 2'd1,
  S_THIRD = 2'd2,
  S_FOURTH = 2'd3;
reg [1:0] state, nextstate;
```

You had to do these three things, or else your FSM would not work:

- Each encoding had to be unique
- The state registers had to have enough bits for all the states
- The encodings also had to have this many bits

If you wanted to add a fifth state in the example above, nearly every line of code would need to change. SystemVerilog solves this problem by introducing enumerations, which are similar to `enum`s in C/C++. Here's the same example using SystemVerilog enumerations:

```
enum int unsigned
{
  S_FIRST,
  S_SECOND,
  S_THIRD,
  S_FOURTH
} state, nextstate;
```

Note the lack of encodings or widths. They are handled automatically. The syntax also allows us to simultaneously define two variables (the current state and the next state) of the enum type we just defined, which is a convenient shorthand. Another benefit is that the actual state names (like `S_FIRST`) show up in ModelSim during simulation, which makes debugging easier.

2 New `always` blocks

SystemVerilog introduces two new kinds of `always` blocks that help you keep your code error-free.

2.1 `always_comb`

Another common problem with writing Verilog for FPGAs is the accidental creation of latches. This happens when the behavior specified in an `always @*` block requires one or more `reg` variables to maintain its old value. Without a clock, the only possible hardware to realize this is a latch, which is unreliable when compiling for the FPGA and behaves differently in simulation.

The `always_comb` construct is a drop-in replacement for `always @*`. It forces the creation of purely combinational logic. If a latch is accidentally inferred, it generates a *compile-time error* rather than an easily-overlooked warning. Figure 1 shows the difference between the two when compiling in Quartus Prime.

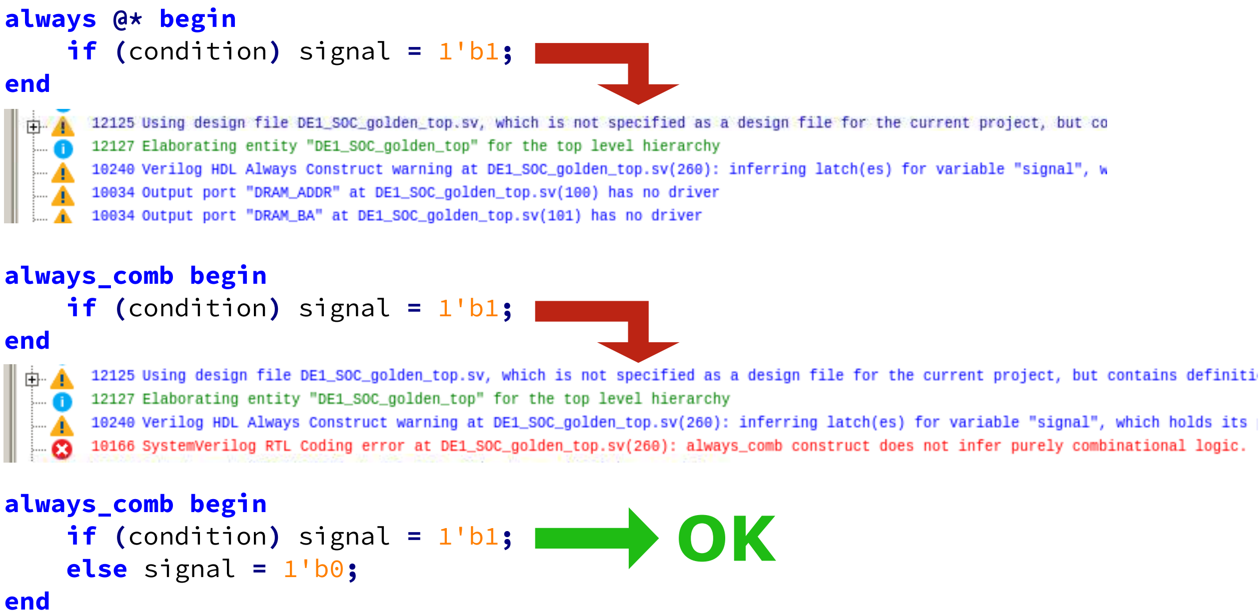

Figure 1: `always @*` versus `always_comb`

2.2 `always_ff`

The `always_ff` block is another way to make clocked `always` blocks for creating sequential logic (registers). It works exactly the same as using the `always` block in Verilog with `posedge` or `negedge` specifiers, except that it will throw a compiler error if `posedge/negedge` are missing:

```
// makes registers - works in Verilog and SystemVerilog
always @ (posedge clock) begin
  ...
end

// also makes registers - SV only
always_ff @ (posedge clock) begin
  ...
end
```

```
// Forgot 'posedge' - compiles but accidentally makes combinational logic or latches
always @(clock) begin
  ...
end

// Compiler error - can't infer registers without pos/negedge
always_ff @(clock) begin
  ...
end
```

3 The `logic` datatype

In Verilog, there are two kinds of signals, and you have to choose carefully the one to use based on what kind of construct writes to it:

- **wire** : Represents the output of combinational logic. Can be `assign`'ed to, but can't be written to from an `always` block.
- **reg** : Represents the output of either combinational logic, a register, or a latch (if you messed up). Can be written to from an `always` block, but not an `assign` statement.

This is inconvenient, since you might need to change the type of a signal from `wire` to `reg` or vice-versa while coding your design. SystemVerilog still supports `wire` and `reg`, but also introduces a new datatype:

- **logic** : Can be written to by either an `always` block (of any kind), or by an `assign` statement.

It makes sense to declare all your signals as `logic` for maximum flexibility. Note that a module's `input`s and `output`s are, by default, of the `wire` type (as they are in Verilog too), so if you want to write to an output from an `always`, `always_ff`, or `always_comb` block, you need to declare it as `output logic` (similarly to how you had to write `output reg` in Verilog).

There is an exception where using `wire` might make sense over using `logic`: `wire` variables can be initialized upon declaration, saving the need for an `assign` statement, but `logic` cannot:

```
// Good
wire x;
assign x = a & b;

// Good - shorthand
wire x = a & b;

// Good
logic x;
assign x = a & b;

// Bad - compiles but doesn't actually perform the assignment
logic x = a & b;
```

Full Example

Shown on the next page is an example module implementing a state machine. On the left is the Verilog version (or equivalently, a SystemVerilog version that only uses a strictly-Verilog subset of the language). On the right is the same module making use of the three SystemVerilog language features introduced in this document.

Verilog

```
module example
(
    input clk,
    input reset,

    input i_go,
    output reg o_ack,
    output reg o_ctrl_output,
    input i_ctrl_busy
);

// Derived signal using assign statement
wire ctrl_success;
assign ctrl_success = o_ctrl_output && !i_ctrl_busy;

// Define states and create state/nextstate variables
localparam
    S_IDLE = 2'd0,
    S_DO_THING = 2'd1,
    S_ACK = 2'd2;

reg [1:0] state, nextstate;

// Creates flip-flops for 'state'
always @ (posedge clk or posedge reset) begin
    if (reset) state <= S_IDLE;
    else state <= nextstate;
end

// Determine outputs and next state
always @* begin
    // Defaults
    nextstate = state;
    o_ack = 1'b0;
    o_ctrl_output = 1'b0;

    case (state)
        S_IDLE: begin
            if (i_go) nextstate = S_DO_THING;
        end

        S_DO_THING: begin
            o_ctrl_output = 1'b1;
            if (ctrl_success) nextstate = S_ACK;
        end

        S_ACK: begin
            o_ack = 1'b1;
            if (!i_go) nextstate = S_IDLE;
        end
    endcase
end


endmodule
```

SystemVerilog

```
module example
(
    input clk,
    input reset,

    input i_go,
    output logic o_ack,
    output logic o_ctrl_output,
    input i_ctrl_busy
);

// Derived signal using assign statement
logic ctrl_success;
assign ctrl_success = o_ctrl_output && !i_ctrl_busy;

// Define states and create state/nextstate variables
enum int unsigned
{
    S_IDLE,
    S_DO_THING,
    S_ACK
} state, nextstate;

// Creates flip-flops for 'state'
always_ff @ (posedge clk or posedge reset) begin
    if (reset) state <= S_IDLE;
    else state <= nextstate;
end

// Determine outputs and next state
always_comb begin
    // Defaults
    nextstate = state;
    o_ack = 1'b0;
    o_ctrl_output = 1'b0;

    case (state)
        S_IDLE: begin
            if (i_go) nextstate = S_DO_THING;
        end

        S_DO_THING: begin
            o_ctrl_output = 1'b1;
            if (ctrl_success) nextstate = S_ACK;
        end

        S_ACK: begin
            o_ack = 1'b1;
            if (!i_go) nextstate = S_IDLE;
        end
    endcase
end


endmodule
```