System Verilog for FPGA Design

Scott Borisch
May 11, 2020

Overview

- System Verilog syntax overview
- Mapping of FPGA resources to System Verilog
- Recommended and Not Recommended syntax

System Verilog Syntax

- System Verilog is a superset of Verilog
- Fundamental units:
  - Modules
  - Packages
- Module:
  - Code that is converted into FPGA resources
    - Registers (aka flip-flops)
    - Lookup tables (aka, CLBs, ALMs, etc.)
    - RAMs
    - DSP blocks
    - etc.
- Packages
  - Generate constants, type definitions, and functions that can be reused in other modules/packages.
- Recommendation: One module or package per file
  - Technically not required but can make it more difficult to use tools (Intel Quartus/Xilinx Vivado)

System Verilog Modules

- A module contains code that consumes FPGA resources when synthesized + placed and routed
- Has a port: defines inputs and outputs to the module
- Can be thought of as an IC (chip) on a PCB:
  - Performs a defined function
  - Has I/O to communicate with other chips
  - Can be “instantiated” by a higher level module for reuse
- The top level module -- the port **is** the pinout of the actual FPGA.

System Verilog Modules: Example

The first part of the module is called the port:

```
module adder (  // Everything between the open and close parentheses is called the port

input logic Clk, // This is a signal declared within the port.  It is referred to as a port signal.

input logic Reset,

input logic [15:0] Addend_1,

input logic [15:0] Addend_2,

output logic [16:0] Sum

);
```

The signal in the port allow the module to interact with other modules.

System Verilog Syntax: Types and Signals

- “logic” is the fundamental data type in system verilog
  - “reg” and “wire” are legacy from verilog.
  - The difference has to do with how you drive the signal.
  - This adds complication and no value. I recommend you use “logic”, not “reg” and “wire”
- You can create single and multi-dimensional arrays of “logic” signals
  - logic [7:0]
  - logic [3:0] [15:0]

System Verilog Syntax: Types and Signals

- typedefs and typedef enums
  - A typedef enum is very useful for a state machine:

```
typedef enum logic [1:0] {

 red = 2'b_00,   green = 2'b_01,

blue = 2'b_10,   yellow = 2'b_11 }

led_state_t;
```

- typedef => defining a new type
- enum => the various values have symbolic names associated with them
- led_state_t is the name of the new type

System Verilog Syntax: Types and Signals

- typedefs and typedef structs
  - Similar to C structs
  - Useful for combining related signals

```
typedef struct {

logic [15:0] write_data; // for writes only.  read_data will be in a different struct

logic [7:0] address; // for reads and writes

logic write_strobe;

logic read_strobe;

} write_data_bus_t;
```

System Verilog Syntax: Types and Signals

- typedefs and typedef structs -- relationship to port signals
- Port signals have directionality
  - input
  - output
  - inout
- inouts are special case
  - inout can be used for top-level ports -- i.e. the actual FPGA (physical) pins
  - inout should NEVER be used for signals inside the FPGA
- A databus inherently has bidirectional behavior: data in/data out
  - This is handled by creating separate signals for input vs. output

System Verilog Syntax: Fundamentals

- All signals inside an FPGA are both written and read (otherwise they don't do anything useful)
- There are four basic ways to drive signals:
  - **always @(posedge Clk) begin** // This represents the output of registers
  - **always @* begin** // This represents the output of lookup tables (aka LUTs/ALMs/CLBs)
  - **assign xxx = yyy** // This is an alternative way to represent the output of lookup tables
  - module instantiations // This is how you build up hierarchy in the design

System Verilog Syntax: Assign Statements

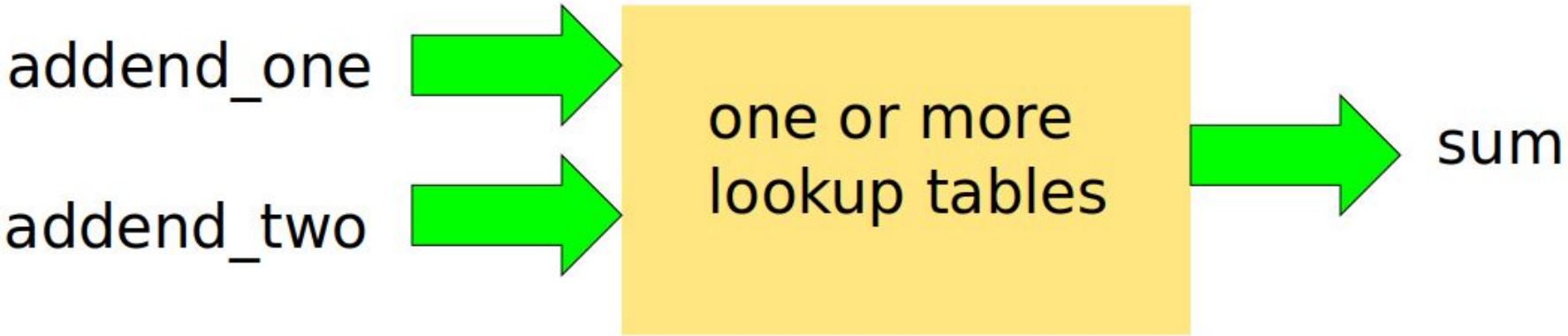
```
logic [15:0] addend_one;

logic [15:0] addend_two;

logic [16:0] sum;

assign sum = addend_one + addend_two; // system verilog “logic” behaves as an unsigned integer
```

addend_one + addend_two will be inputs to lookup table(s)

sum will be the output of lookup table(s)

System Verilog Syntax: Assign Statements

- The synthesis tool (part of Quartus/Vivado) will automatically:
  - Determine how many lookup tables are required
  - Determine how to interconnect the lookup tables
  - Determine how to program the lookup rules in each lookup table
- The place and route tool (part of Quartus/Vivado) will automatically
  - Determine the physical placement of the lookup tables within the FPGA
  - Determine the phyiscal routing lanes to connect the lookup tables within the FPGA



System Verilog Syntax: always @* Statements

- always @* statements are similar to assign statements in that they represent lookup tables
- However, they allow a much wider variety of syntax

```
logic [15:0] addend_one;

logic [15:0] addend_two;

logic [16:0] sum;

always @* begin

 sum = addend_one + addend_two; // system verilog “logic” behaves as an unsigned integer

end
```

- Note that the plus operator can be used in assign and always @*
- This is not true of all syntax

System Verilog Syntax: always @* Statements

- Common (not comprehensive) list of operators/syntax that can be used inside always @*
  - Add, subtract: + and -
    - Multiply and divide are also legal System Verilog operators, BUT
      - Most FPGAs only support integer math, not floating point, AND
      - (integer) Division in an FPGA is extremely expensive.
      - (integer) Multiplication is also expensive in lookup tables -- it is better to use dedicated DSP blocks -- we will come back to this later ...
  - Modulus operator: %
  - Equality and inequality comparisons
  - Bitwise and logical operators: or/and/not
  - Shift operators
  - If/else if/else statements
  - Case statements
  - For loops
  - Others ...

System Verilog Syntax: For loops inside always @* Statements

- What does a for loop inside an always @* statement actually mean ??
- It means that all the logical operations inside the for loop will be executed in a single clock cycle !!
  - So you need to be very careful with this ...
- For loops can be used in two very different ways
  - To use the result of the **previous iteration** in the **next iteration** -- all in one clock cycle
    - Or, at a minimum, all loops must be evaluated to determine the result.
  - To replicate the same logical operation on an array of inputs/outputs

System Verilog Syntax: For loops: Iterative logic

```
logic [7:0] channels_requesting_interrupt;
logic [2:0] winning_channel; // NOTE: 7:0 => 8 bits; log2(8) == 3; 2:0 => 3 bits
always @* begin : PriorityInterruptDetector
  integer idx; // idx is a variable
  winning_channel  = '0; // Assign a default value !!
  for (idx=0;idx<8;idx=idx+1) begin
     if (channels_requesting_interrupt[idx] === 1'b1) begin
        winning_channel = idx; // idx is an integer, winning_channel is a three bit value.  So the MSBs of idx will be truncated in the assignment.
                                //This is fine because we never assign idx >= 8.
                               // winning_channel will end up set to the value which corresponds to the
                               // highest bit position of the signal channels_requesting_interrupt with value one.
    end
  end
end : PriorityInterruptDetector
```

System Verilog Syntax: For loops: Iterative logic

- Comments on code on previous slide:
  - ALL signals driven in an always @* begin block should be assigned a value, for all possible values on input signals
  - Because winning_channel is driven inside an if statement, it will not be driven if channels_requesting_interrupt **is all zeros.**
  - Therefore it is assigned a default value.
    - winning_channel = ‘0; // ‘0 means assign all bits to zero.
  - It is good style to assign a default value for ALL signals driven in an always @* block, even if they unconditionally driven (overridden) further down in the always @* block
    - This is because this is easy to overlook this in complex (lengthy) blocks.
  - If you don’t assign a value to a signal in all cases, you will get an “inferred latch”. This means the signal will retain the value from the previous clock cycle.
  - Combinational logic is not supposed to do this. It means the code is easy to misunderstand.
  - Synthesis tools will warn about an inferred latches. You should always fix your code to remove them.

System Verilog Syntax: For loops: Parallel logic

```
logic [3:0][15:0] addend_one;

logic [3:0][15:0] addend_two;

logic [3:0][16:0] sum;


always @* begin
  integer idx;
  for (idx=0;idx<=3;idx=idx+1) begin
    sum[idx] = addend_one[idx] + addend_two[idx]; // Each iteration of the logic involves unrelated inputs and unrelated outputs
  end
end
```

Generate Statements

- Two types:
  - Conditional
  - Loops
- Conditional Loops: allows you to add or remove entire sections of logic:
  - **always @* begin** blocks
  - Module instantiations
  - etc.
- Loops
  - Allows for replicating logic N times

For Generate Statements

```
genvar idx; // add extra genvars here to allow for nested generate loops
generate
  for (idx=0;idx<=3;idx=idx+1) begin // you can have multiple nested for loops if desired
    // always @* begin logic ...
    // always @(posedge Clk) begin logic ...
    // module instantiations ...
  end
endgenerate
```

If Generate Statements

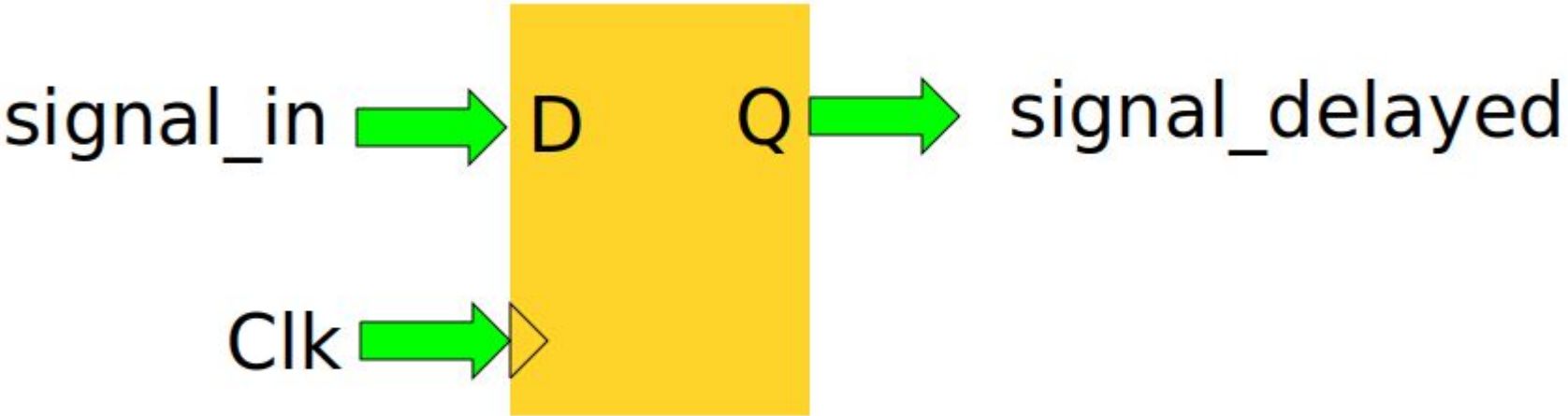
```
localparam logic INCLUDE_LOGIC = 1'b1;
// no genvar for if generate
generate
  if (INCLUDE_LOGIC == 1'b1) begin
    // always @* begin logic ...
    // always @(posedge Clk) begin logic ...
    // module instantiations ...
  end
  else begin
    // always @* begin logic ...
    // always @(posedge Clk) begin logic ...
    // module instantiations ...
  end
endgenerate
```

System Verilog Syntax: always @(posedge Clk) Statements

- always @(posedge Clk) statements represent the outputs of registers (flip-flops)
- They can optionally represent BOTH registers and lookup tables

```
logic Clk;

logic signal_in;

logic signal_delayed;

always @(posedge Clk) begin

 signal_delayed <= signal_in; // signal_in is the input to the register.  signal_out is the output of the register

end
```

System Verilog Syntax: always @(posedge Clk) Statements



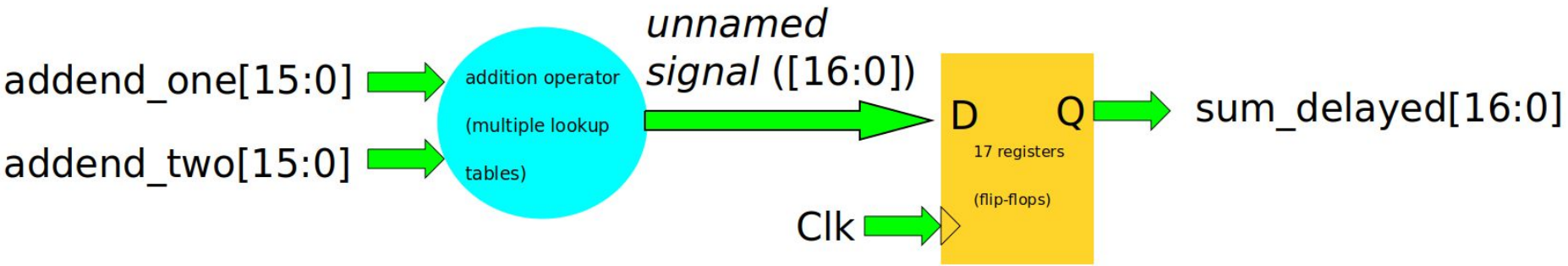
System Verilog Syntax: always @(posedge Clk) Statements

- Example showing **always @(posedge Clk)** representing BOTH registers and lookup tables

```
logic Clk;
logic [15:0] addend_one;
logic [15:0] addend_two;
logic [16:0] sum_delayed;
always @(posedge Clk) begin
 sum_delayed <= addend_one + addend_two ; // the input to the registers are implicit -- they exist but have no System Verilog name !!
end
```

System Verilog Syntax: always @(posedge Clk) Statements

- Example showing **always @(posedge Clk)** representing BOTH registers and lookup tables



System Verilog Syntax: always @(posedge Clk) Statements

- Lookup tables are referred to as combinational logic
- Registers are referred to as registered logic
- So, **always @(posedge Clk)** can represent registered logic ONLY, or they can represent combinational AND registered logic
- Combining combinational and registered logic may seem appealing due to its brevity.
- However there are multiple reasons I recommend against this. They all have to do with debugging.
    - You cannot view the result of the logical operation in a simulator waveform viewer. Thus, you cannot determine if an unexpected value is due to a mistake in the combinational logic, or due to unexpected behavior with the clock (or reset) signal.

System Verilog Syntax: Resets

- Synchronous resets only matter on the clock edge, the same as the “D” input to the register
- Asynchronous resets take effect immediately, without regard to the clock edge
- Asynchronous resets are more complex to use properly, so they should be avoided unless there is a good reason to use them
    - Really outside the scope of an introductory presentation
    - Just be aware they exist

System Verilog Syntax: Synchronous Resets

- Now let's add a synchronous reset to the example

```
logic Clk;
logic Reset;
logic [15:0] addend_one;
logic [15:0] addend_two;
logic [16:0] sum_delayed;
always @(posedge Clk) begin
  if (Reset == 1'b1) begin // This is a synchronous reset
     sum_delayed <= '0 ;
  end
  else
    sum_delayed <= addend_one + addened_two;
  end
end
```

System Verilog Syntax: Asynchronous Resets : DON’T

- Now let’s add an asynchronous reset to the example

```
logic Clk;
logic aReset;
logic [15:0] addend_one;
logic [15:0] addend_two;
logic [16:0] sum_delayed;
always @(posedge Clk or posedge aReset) begin
  if (aReset == 1’b1) begin // This is an asynchronous reset
     sum_delayed <= ‘0 ;
  end
  else
    sum_delayed <= addend_one + addened_two;
  end
end
```

System Verilog Syntax: Modules and Module Instantiations

- Modules can be code you/a co-worker writes.
- Or that you buy from another company …
- Or get open source …
- They can also represent certain resource types inside the FPGA
  - RAMs, FIFOs, PCIe endpoints, etc.
  - In this case, the module is a “black box”
  - You can’t see the code inside of it
  - You only know the ports of the module, and get documentation that describes the behavior.

System Verilog Syntax: Modules and Module Instantiations

- A simple fifo example:

```
module fifo (
  input logic            Clk,
  input logic [15:0]   data_in, // When push == 1, the value on data_in is copied into the fifo buffer
  output logic [15:0] data_out,  // When empty == 0, contains the data at the head of the fifo buffer
  input logic            push,  // When == 1, the value on data_in is copied into the tail of the fifo buffer
  input logic            pop,  // When == 1, the value on data_out is replaced with the “head - 1” element in the fifo buffer
  output logic          empty, // When == 1, fifo is empty and data_out valid is invalid.  When == 0, data_out is the entry at the fifo head.
                               // Driving pop == 1 when empty == 1 results in an underflow
  output logic          full, // When == 1, all storage locations in the fifo is full.  Driving push == 1 when full == 1 results in overflow.
  output logic [3:0]  words,  // The number of words in the fifo
);
```

- In this example, the port signal words indicates the number of 16-bit **words** stored in the fifo
- Since **words** is 4-bit, the maximum possible value is 15.
- Most likely, this fifo can either store 16 words (in which case **full** == 1 implies **words** should be ignored), OR can store 8 words.
- You need to check the documentation to know which is the case !

System Verilog Syntax: Modules and Module Instantiations

- Now we can instantiate the fifo:

```
fifo my_fifo ( // mf_fifo is the instance name of an instantiation of module fifo.
  .Clk (Clk),
  .data_in (my_data_in),
  .data_out (my_data_out),
  .push (my_push),
  .pop (my_pop),
  .empty (my_empty),
  .full (my_full),
  .words (my_words)
);
```

- my_data_in, my_data_out, etc. are signals declared in the local module -- the one instantiating my_fifo.
- Inputs to my_fifo are driven from the local module -- this could be from either a register output (**always @(posedge Clk) begin**) or a combinational output (***assign*** or ***always @* begin***)
    - Driving from a register would be best due to static timing (topic for the future)

Static Timing

- All paths in the FPGA must run fast enough to complete within 1 clock cycles.
- A path starts at a register (flip-flop) output, passes through 0 or more lookup tables and ends at the input of another (or the same) register
- Lookup tables have delay
- Routes have delay
  - Register output => lookup table input route
  - Lookup table output => lookup table input route
  - Lookup table output => register input route

Static Timing

```
logic [31:0] my_address, some_other_address;

always @* begin
  case(state)
    IDLE : begin
      // something
    End
    COMPARE_ADDRESS : begin
      if (my_address >= some_other_address) begin // This requires first doing a state decode and then doing an inequality comparison in 1 clock cycle
       state_next = SOME_OTHER_STATE
      end
    end
  endcase
end
```

Static Timing

```
logic [31:0]   my_address, some_other_address;

logic address_compare_next; address_compare; // address_compare_next is register input; address_compare is register output

always @* begin

  address_compare_next = (my_address >= some_other_address);

  case(state)

    IDLE : begin

      // something

    End

    COMPARE_ADDRESS : begin

      if (address_compare == 1'b1) begin // This requires first doing a state decode and then doing a single bit compare only.

                                                  // Inequality comparison was moved to a different pipeline stage

        state_next = SOME_OTHER_STATE

       end

     end

  endcase

end
```

Summary of Recommendations

1. One module per file
2. One package per file
3. Filename matches module/package name
4. Don’t use **reg** and **wire**; use **logic**
5. No async resets
6. **inouts** only at top level
7. structs can’t combine inputs and outputs
8. **always @*** / **@(posedge Clk)** code completes in one clock cycle
9. Initialize all LHS signals in **always @***
10. **always @*** for combinational logic
11. **always @(posedge Clk)** for registered logic
12. Use **if generate** to include/exclude logic

Questions ?