Introduction to Digital Design with Verilog HDL

This lecture introduces digital logic circuit design using the Verilog Hardware Description Language. It covers many topics at a superficial level; we will revisit each in more detail.
After this lecture you should be able to: define a module with single- and multi-bit logic inputs and outputs; write expressions using `logic` *variables and operators; use* `assign` *statements and* `always_comb` *procedural blocks to generate combinational logic; use* `always_ff` *to model D flip-flops; use* `if` *and* `case` *statements to model multiplexers and arbitrary logic functions; write Verilog numerical constants; implement an algorithm as a register-transfer datapath and controller; instantiate a module into another module and into a testbench;.*

Introduction

Most of the functionality of modern electronics is defined by its software. However, for reasons of speed, cost or power a digital system may require custom digital hardware. This course explains how to design such circuits.

The design process begins by specifying the interfaces and how the design will be verified. Solutions are then iterated until the requirements are met.

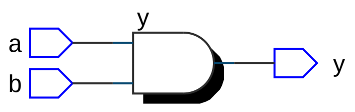
Digital circuits are mainly designed using Hardware Description Languages. In this course we will use Verilog rather than the other popular HDL, VHDL.

Combinational Logic

Consider the following Verilog description of an AND gate:

```
// AND gate in Verilog

module ex1 ( input logic a, b,
             output logic y ) ;

   assign y = a & b ;

endmodule
```

Logic synthesis software (e.g. Altera's Quartus) can map this description into the following circuit:



Verilog includes most C operators including arithmetic (`+, -, *, /, %`), bitwise (`&, |, ^, ~, <<, >>`), comparison (`>, >=, !=`, etc.), logical (`&&, ||, !`), array indexing (`[]`), and ternary conditional (`?:`). C syntax is also used for comments.

Exercise 1: What changes would result in a 3-input OR gate?

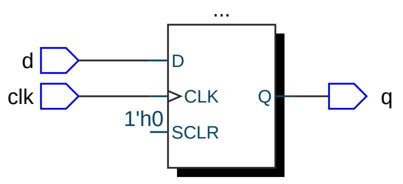
Exercise 2: What schematic would you expect if the statement was `assign y = a ? b : c ;`?

Registers

The following Verilog:

```
module ex2 (input logic d, clk,
            output logic q) ;

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

endmodule
```

synthesizes a D-flip-flop that transfers the `d` input to the `q` output on the rising (positive) edge of `clk`:

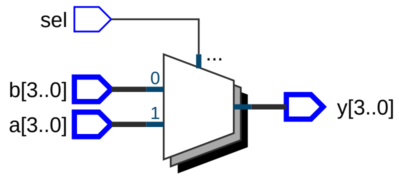


Multiplexers and Buses

Verilog's `if` statement models a multiplexer. The following example selects one of two 4-bit inputs:

```
module ex3 (input logic sel,
            input logic [3:0] a, b,
            output logic [3:0] y) ;

   always_comb begin
      if ( sel ) y <= a ;
      else y <= b ;
   end

endmodule
```

which results in:



Arrays model buses. The array declaration specifies the range of possible index values. Each index value corresponds to one bit of the bus. Index ranges are usually specified in decreasing order so that when buses represent integers the first array element is the most significant (leftmost) bit. In the example above `a[3]` is the most significant bit of the 4-bit signal `a`.

Exercise 3: What change might produce a 4-bit 4-to-1 multiplexer controlled by a 2-bit `sel` input?

Exercise 4: If the signal `i` is declared as `logic [2:0] i;`, what is the 'width' of `i`? If `i` has the value 6 (decimal), what is the value of `i[2]`? Of `i[0]`?

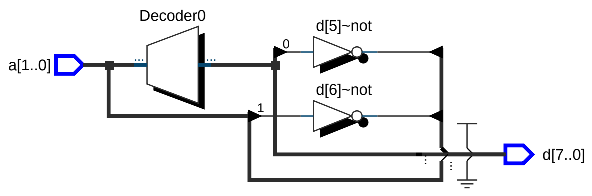
Case Statements and Numeric Constants

A Verilog case statement can model arbitrary combinational logic.

The following code describes a 2-input/8-output ROM memory (or "look-up table"):

```
module ex4 (input logic [1:0] a,
            output logic [7:0] d) ;

   always_comb begin
      unique case (a)
        0: d = 8'hc0 ;
        1: d = 8'b1111_1001 ;
        2: d = 'ha4 ;
        3: d = 176 ;
      endcase
   end

endmodule
```

which synthesizes into:



Numeric constants in Verilog are written as the number of bits (default 32), an optional quote (`'`) followed by the base (b=binary, x=hex, d=decimal), and the value. Underscore separators (_) are optional.

Exercise 5: What are the values in decimal of the constants in the code above?

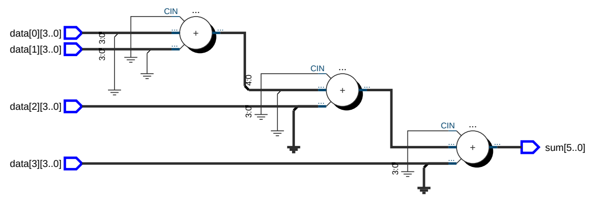
Exercise 6: What is the output in binary when the input is `a=2'b10`?

for-loops and Memory

Verilog's `for`-loops replicate combinational logic. This example:

```
module ex9 ( input logic [3:0] data [0:3],
             output logic [5:0] sum ) ;

   always_comb begin

      sum = 0 ;

      for ( int i=0 ; i<4 ; i++ ) begin
         sum += data[i] ;
      end

   end

endmodule
```

adds the four elements of an array of 4-bit numbers. However, the resulting circuit:



is not what you may have expected. A C program would add one element of the array at a time to a sum variable (register). However, the synthesized circuit has no registers and generates the result directly.

Memories can be modeled as arrays of multi-bit registers. The `data` input above is an example.

Register Transfer Level (RTL) Design

You cannot use C control structures (e.g. for, if/else) to implement algorithms. Instead, you must define registers, operations on these registers and a state machine that generates the correct sequence of these operations.

For example, the following fragment of Verilog implements a register, `x`, that can be set to 8, incremented or decremented depending on the signals `reset_n` and op.

```
always_ff @(posedge clk) begin
   if ( ! reset_n )
     x <= 4'd8 ;
   else
     case (op)
       incr: x <= x + 4'd1 ;
       decr: x <= x - 4'd1 ;
     endcase
end
```

Additional logic defines the sequence of operations. In the example below the value of the register op switches to incr(ementing) when the value of x reaches 5 and to decr(ementing) when the value of x reaches 11.

```
always_ff @(posedge clk)
   op <= next ;

always_comb begin
   if ( ! reset_n )        next = incr ;
   else if ( x == 4'd05 ) next = incr ;
   else if ( x == 4'd11 ) next = decr ;
   else                    next = op ;
end
```

We can combine these two pieces of code, called the "datapath" and "controller" respectively, into a Verilog module along with the required i/o and declaration of op and it's possible values:

```
module ex5 ( input logic reset_n, clk,
             output logic [3:0] x ) ;

   enum logic[1:0] { incr, decr } next, op ;

   // datapath code goes here
   `include "ex5a.sv"

   // controller code goes here
   `include "ex5b.sv"

endmodule
```

This generates the circuit shown in Figure 1.

The controller is a state machine shown as a rectangle while the datapath is the multiplexers that load the register `x` with 0, `x+1` or `x-1` on each rising edge of `clk`.

More complex algorithms will require more registers; both in the datapath and to control the sequencing of the operations.

Exercise 7: Label the `x+1` and `x-1` buses.

Exercise 8: Draw the state transition diagram for the op register.

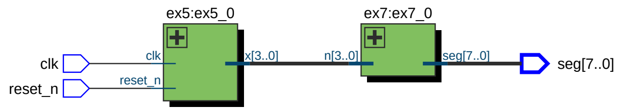
Hierarchy

Designs are divided into modules to simplify testing and allow re-use.

A module can instantiate (include) instances (copies) of other modules. For example, if we had a 7-segment LED display decoder module called `ex7` with a 4-bit input (`n`) and an 8-bit output (`seg`) we could combine it with the count-to-5 module (`ex5`) to build a display that shows digits counting from 0 to 5:

```
module ex8 ( input logic reset_n, clk,
             output logic [7:0] seg ) ;

   logic [3:0] count ;

   ex5 ex5_0 ( .*, .x(count) ) ;
   ex7 ex7_0 ( .n(count), .seg ) ;

endmodule
```

The module instantiation syntax allows mapping of the instantiated module's port names to signals in the instantiating module. The synthesized result is:



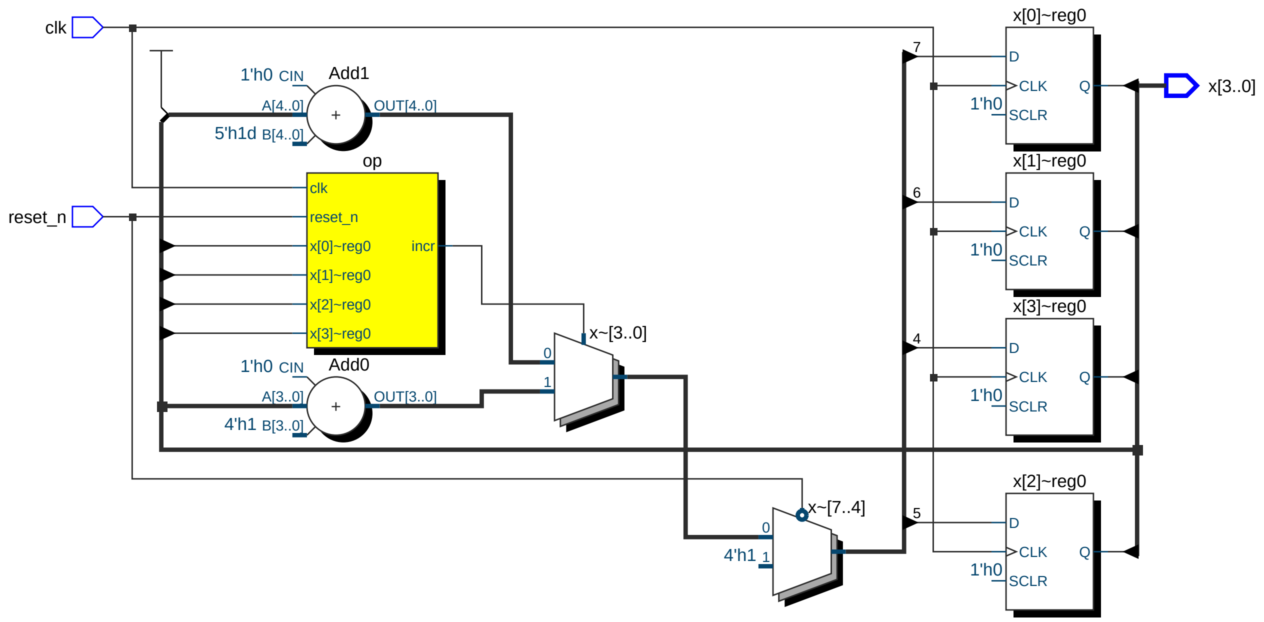


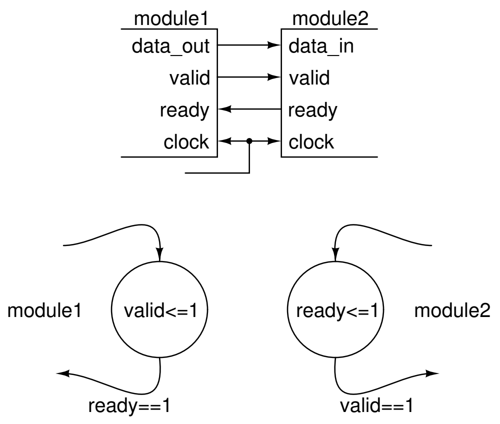
Figure 1: RTL design example.

Exercise 9: Which ports are mapped by `.*` in the instantiation of `ex5`?

Exercise 10: Write the module declaration for `ex7`.

Interfaces

A module may not be ready to accept input or its output may not be valid while it is "running" an algorithm. `ready` and `valid` signals can coordinate transfer of data between such modules:



Module 1 sets `valid` when it moves to a state where `data_out` is valid. Module 1 transfers out of this state (and possibly back to the same state) when the `ready` input is set. Module 2 sets `ready` when it transitions to a state where it can load `data_in` into a register. Module 2 transfers out of this state (and possibly back to the same state) when the `valid` input is set.

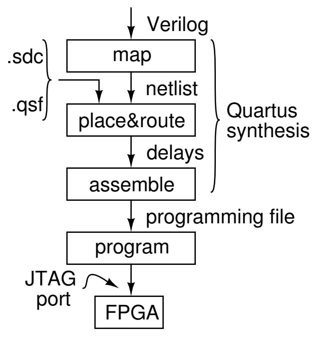
Exercise 11: What is the minimum number of clock cycles required to transfer one data word over an interface with `ready`/`valid` handshaking? Maximum?

A System on a Chip (SoC) includes a processor (CPU) plus additional logic. These interfaces have been standardized (e.g. Avalon for Altera FPGAs, AMBA for ARM processors) to allow re-use of custom logic (IP or "Intellectual Property") blocks.

These interfaces can include an address as well as data. These "memory-mapped" interfaces allow the CPU to access multiple memory locations or registers within the module.

Implementation

The process to implement a design using an FPGA (Field Programmable Gate Array) IC is shown below.



After the design is mapped to gates and flip-flops it must be fit into a specific device. Additional information the fitter needs to "place and route" the design is supplied in two files. The `.qsf` (Quartus settings) file device contains, among other things, the device type (part number) and the pin assignments. For example:

```
set_global_assignment -name DEVICE EP4CE22F17C6
set_location_assignment PIN_A15 -to x[0]
  ...
set_location_assignment PIN_E1 -to reset_n
```

Timing constraints such as clock frequencies and external device setup/hold times are defined in a `.sdc` (Synopsis Design Constraint) file. For example, the following statement requires that the design meet setup and hold requirements with a 50 MHz (20 ns period) clock signal called `CLOCK_50`:

```
create_clock -period 20ns CLOCK_50
```

Finally, the placed and routed design is "assembled" to a file that can program the FPGA, typically over a dedicated "JTAG" programming/diagnostic interface port on the FPGA.

Timing

Reliable operation requires that the setup and hold time requirements of all flip-flops in the design be met. The place and route operation generates a netlist "annotated" with the delays between flip-flop outputs and inputs. Timing analysis software subtracts these delays from the clock period to check for "slack" in meeting the timing requirements. If negative, then mapping and/or P&R can be repeated with this additional information until there is timing "closure."

The timing requirements of off-chip components such as memories and interface ICs should also be included in the timing analysis.

When signals are exchanged between circuits with independent clocks ("clock domain crossing") it's not possible to ensure that the timing requirements will be met. This results in a non-zero probability that the timing requirements will violated and that the circuit will fail. Various strategies can reduce, but not eliminate, the likelihood and consequences of such "metastable" behaviour.

Exercise 12: A circuit's clock has a period of 20 ns. The propagation delay between flip-flop outputs and inputs is a minimum of 5 ns and maximum of 18 ns. What are the minimum setup and hold times?

Testbenches and Simulation

A circuit can be tested by simulating its operation. In this example an executable Verilog module called a "testbench" applies inputs to the module being tested:

```
// synthesis translate_off
module ex6 ;

   logic clk=0, reset_n=0 ;
   logic [3:0] x ;

   ex5 ex5_0 (.*);

   initial begin
      $dumpfile("ex6.vcd");
      $dumpvars ;
      @(negedge clk) reset_n = 1 ;
      repeat (22) @(posedge clk) ;
      $finish ;
   end

   always begin
      #500ns clk = ~clk ;
   end

endmodule
// synthesis translate_on
```

This testbench de-asserts the reset signal at the clock's first falling edge and waits 30 clock cycles before terminating. It also generates a 1 MHz clock. In this example the waveforms are written to the file `ex6.vcd` for viewing. Testbenches can also check the outputs themselves.

The testbench is not synthesizable because it includes "system tasks" (`$dumpfile` and `$finish`)

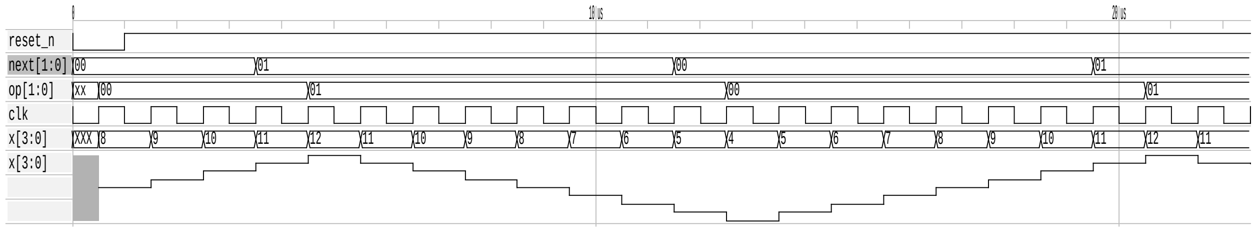


Figure 2: Simulation results.

and delays (`#500ns`), that cannot be implemented in hardware. The `translate_on/_off` "pragmas" in the comments disable processing of the testbench by synthesis software.

Figure 2 shows the waveforms written to file `ex6.vcd` when the simulation was run.

Exercise 13: Where in the code is the Device Under Test (DUT) instantiated?

Exercise 14: The simulation results show the values of `x` ranging from 4 to 12. But the limits in the `ex5` code are 5 and 11. Why?

Other simulators read a file containing "test vectors" – test inputs and the expected outputs. The simulator compares the module's simulation output to the expected values.

Simulations using netlists without delay annotations are called "functional" simulations because they verify that the design is correct. Simulations that include delays, called "timing" simulations, verify that the design will work properly after being placed and routed and at the specified clock rate.