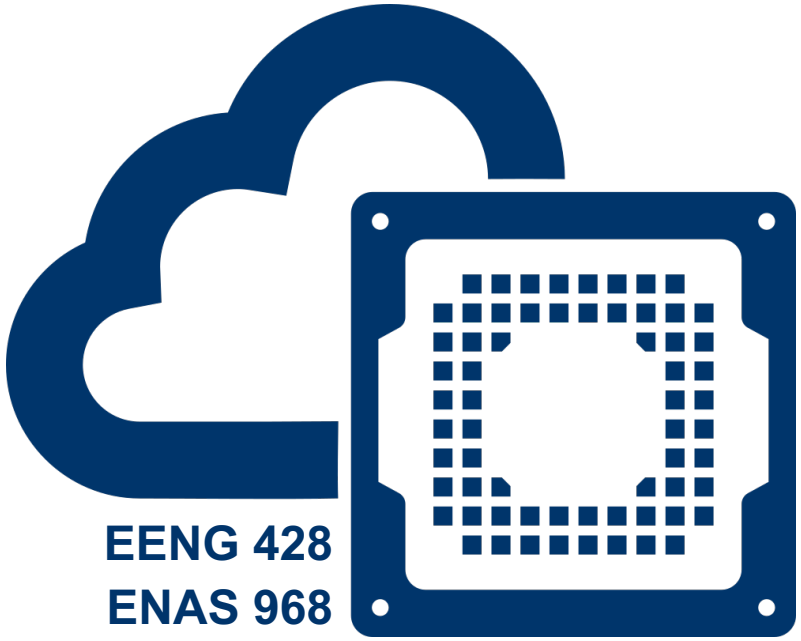
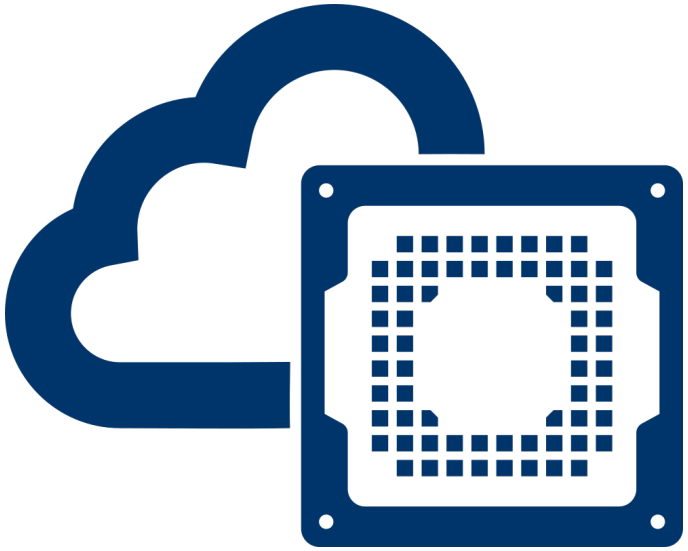
Cloud FPGA



bit.ly/cloudfpga



Lecture: Logic Level Modeling

Prof. Jakub Szefer
Dept. of Electrical Engineering, Yale University

EENG 428 / ENAS 968
Cloud FPGA

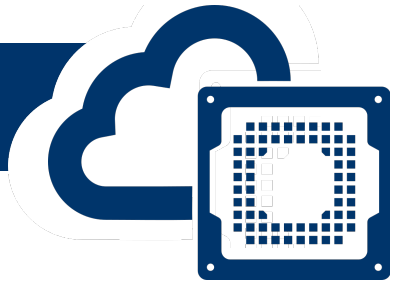
Logic Synthesis

This lecture is mostly based on contents of Chapter 6, from “The Verilog Hardware Description Language” book [1], 5th edition. Example figures and (modified) code are from the textbook unless otherwise specified.

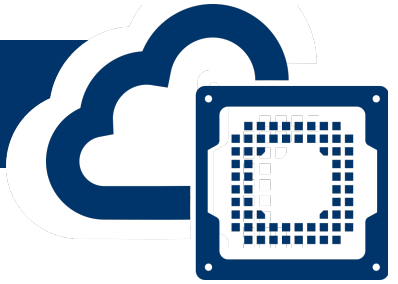
Topics covered:

- Logic gates and nets
- Four logic level values
- Continuous assignment
- Logic delay modeling
- Specifying time units

Logic Level Modeling



Logic Level Modeling



- Behavioral modeling focuses on describing behavior and functionality of a circuit
  - Behavioral code is written almost as a software-like function

- **Logic level modeling is used to model the logical structure of a design**
  - Specify ports (inputs and outputs)
  - Any submodule instances
  - Connection between the submodules
  - Logical functions

- Levels of logic modeling
  1) Gate level – describe design in terms of interconnection of logical gates
     - Use gate-level primitives
  2) Continuous assignment statements – describe designs using Boolean algebra-like expressions
     - Use **`assign`** statements
  3) Transistor switch level – describe at level of MOS and CMOS transistors

Note at the gate level, the actual implementation in FPGA will not use exactly the gates used in Verilog – all code is compiled to use Look-Up Tables (LUTs) in the FPGAs

FPGA and ASIC vendors provide modules such as `LUT`, `DFF`, etc. to let users exactly specify the hardware that will be implemented

Logic Gates and Nets

- Describing and modeling design at the logic level can be done using Verilog's built-in logic gate and switch primitives:

n_input gates	n_output gates	tristate gates	pull gates	MOS switches	bidirection-al switches
and	buf	bufif0	pullup	nmos	tran
nand	not	bufif1	pulldown	pmos	tranif0
nor		notif0		cmos	tranif1
or		notif1		rnmos	rtran
xor				rpmos	rtranif0
xnor				rcmos	rtranif1

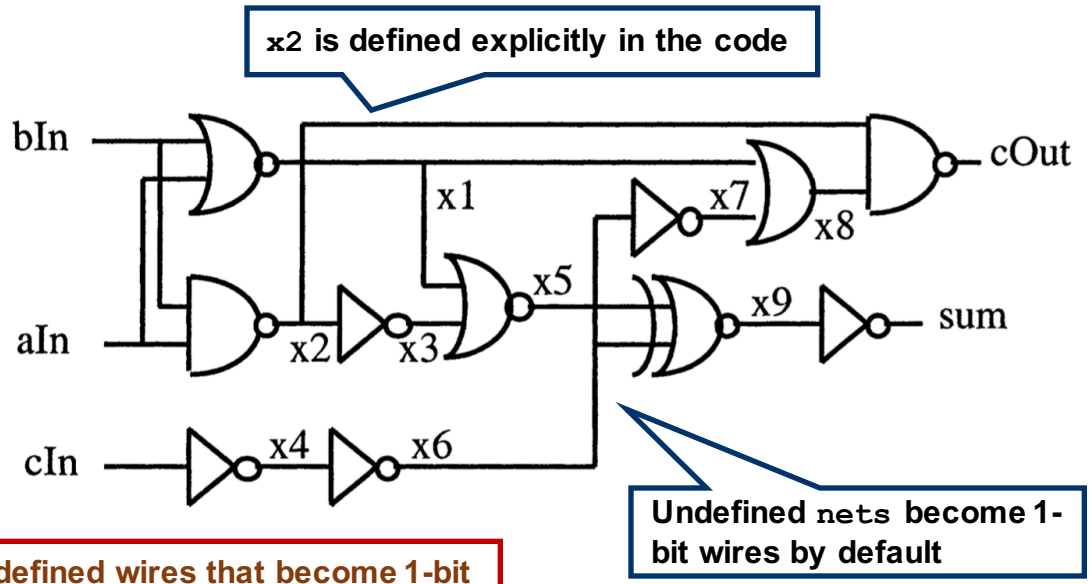
Used for logic-level modeling

Used for transistor-level modeling

- The gates (and switch level primitives) can be interconnected using **`nets`**
  - A **`net`** is a Verilog type, pretty much a representation of a physical wire
  - **`nets`** do not store values or charges, except a **`tirreg`** type of a **`net`**

Logic Gates and Nets

- Textbook example of full adder module
  - Each gate is interconnected using **`nets`**
  - Inputs and outputs are wire **`nets`** by default
  - Undefined **`nets`** will be made into wires by default

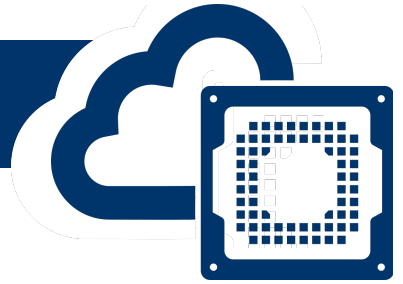


Common problem is undefined wires that become 1-bit wires by default and cause logic to work incorrectly!

Can use `` `default_nettype none `` to disable automatic definition of wire `nets`

```
module fullAdder
( output cOut, sum,
  input aIn, bIn, cIn
);

  wire x2;

  nand (x2, aIn, bIn),
       (cOut, x2, x8);

  xnor (x9, x5, x6);

  nor (x5, x1, x3),
      (x1, aIn, bIn);

  or (x8, x1, x7);

  not (sum, x9),
      (x3, x2),
      (x6, x4),
      (x4, cIn),
      (x7, x6);

endmodule
```

Logic Gate Instance Options



Gates (and modules, see later slides) can be instantiated with a number of options

- **Instance names** – are used in hierarchical design, a good practice is to assign meaningful and unique instance names to all modules and gates

- **Gate delay** – is used to quantify the number of time units form when any input changes to when output changes, this is the propagation delay

Strength and delay are not used for writing code synthesized to FPGAs (and ASICs)

```
nand (strong0, strong1) #3
  GateA (x2, aIn, bIn),
  GateB (cOut, x2, x8);
```

- **Drive (and charge) strength** – used for modeling physical behavior of wires
  - The drive strengths are: **`supply`**, **`strong`**, **`pull`**, **`weak`**, and **`highz`** (all **`nets`** except **`trireg`**)
  - The charge strengths are: **`large`**, **`medium`** and **`small`** strengths (for **`trireg`** only)
  - Higher drive strength can supply the needed current faster when switching

Drive strength details from:
http://verilog.renerta.com/source/vrg00047.htm

Four Logic Level Values

- The values that may be driven onto a net are:
  - **0** – a logic zero, or FALSE condition
  - **1** – a logic one, or TRUE condition
  - **x** – an unknown logic value (any of 0, 1, or in a state of change)
  - **z** – a high-impedance condition

- Gate truth tables with respect to the possible logic levels (see textbook's Appendix D):

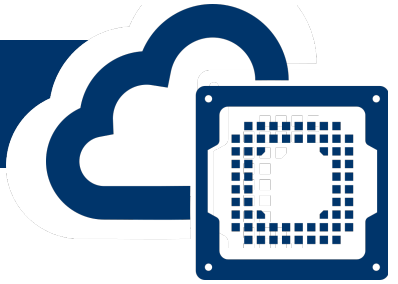
AND	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

NOT	output
0	1
1	0
x	x
z	x

		Control Input			
	Bufif0	**0**	**1**	**x**	**z**
D	**0**	0	z	L	L
A	**1**	1	z	H	H
T	**x**	x	z	x	x
A	**z**	x	z	x	x

L indicates 0 or z; H indicates 1 or z

Different `net` types



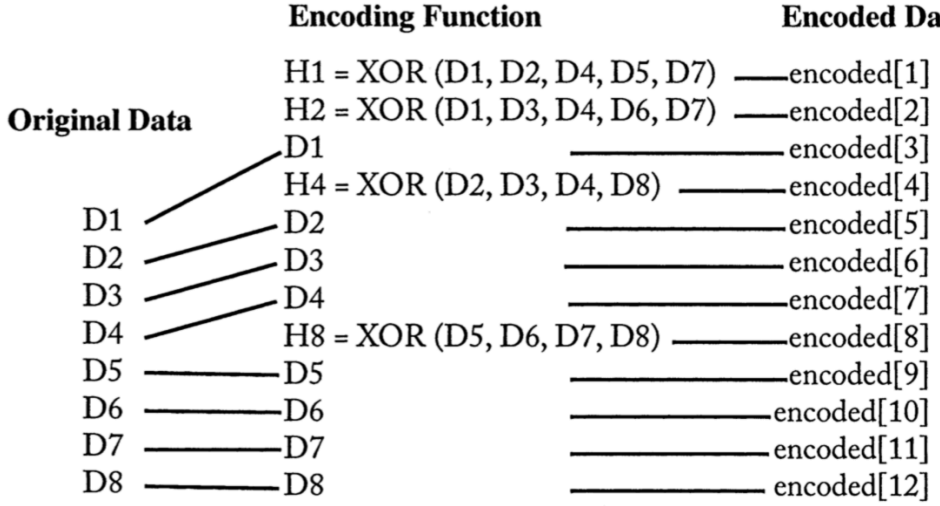
- **`nets`** are used to model electrical connections
  - **`nets`** store no charges and are just a connection
  - Except **`trireg`** that models wires as capacitors that store charge
- Many net types are supported in Verilog:
  - **`wire`**, **`tri`**, **`tri1`**, **`supply0`**, **`wand`**, **`triand`**, **`tri0`**, **`supply1`**, **`wor`**, and **`trior`**

Main type used for any synthesizable code

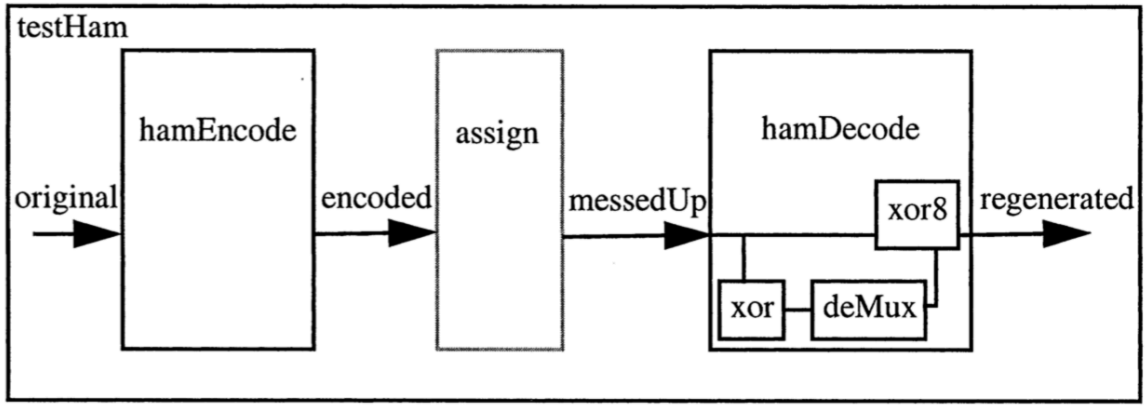
Net Type	Modeling Usage
wire and tri	Used to model connections with no logic function. Only difference is in the name. Use appropriate name for readability.
wand, wor, triand, trior	Used to model the wired logic functions. Only difference between wire and tri version of the same logic function is in the name.
tri0, tri1	Used to model connections with a resistive pull to the given supply
supply0, supply1	Used to model the connection to a power supply
trireg	Used to model charge storage on a net. See Chapter 10.

Logic Level Modeling Example

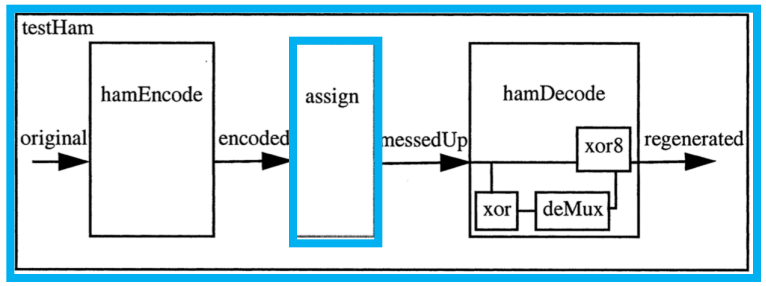
- Textbook of Hamming encoder and decoder and a simple testbench
  - Hamming codes can be used to detect and correct errors in transmitted data
  - Example uses code that can correct 1 error



- Example test module:

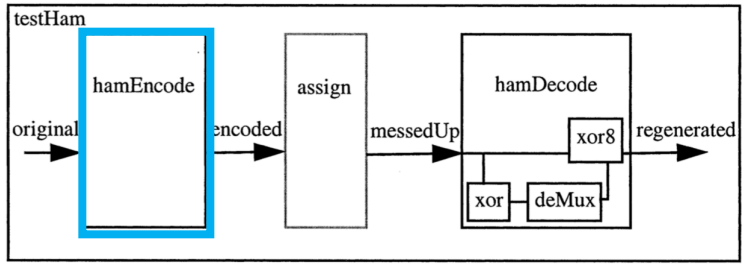


Logic Level Modeling Example




```
module testHam;

  reg [1:8] original;
  wire [1:8] regenerated;
  wire [1:12] encoded, messedUp;

  integer seed;
  initial
  begin

    seed = 1;

    forever
    begin
      original = $random (seed);
      #1
      $display ("original=%h, encoded=%h, messed=%h,regen=%h",
                original, encoded, messedUp, regenerated);
    end
  end

  hamEncode hIn (original, encoded);

  hamDecode hOut (messedUp, regenerated);

  assign messedUp = encoded ^ 12'b 0000_0010_0000;

endmodule
```

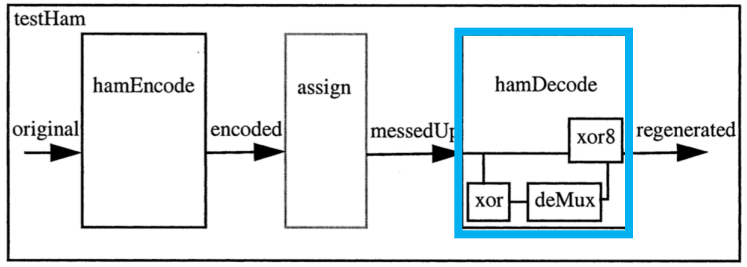
Logic Level Modeling Example




```
module hamEncode
( input [1:8] vIn,
  output [1:12] valueOut
);

  wire h1, h2, h4, h8;
  xor (h1, vIn[1], vIn[2], vIn[4], vIn[5], vIn[7]),
      (h2, vIn[1], vIn[3], vIn[4], vIn[6], vIn[7]),
      (h4, vIn[2], vIn[3], vIn[4], vIn[8]),
      (h8, vIn[5], vIn[6], vIn[7], vIn[8]);

  assign valueOut = {h1, h2, vIn[1], h4, vIn[2:4], h8, vIn[5:8]};

endmodule
```


Share: bit.ly/cloudfpga

Logic Level Modeling Example

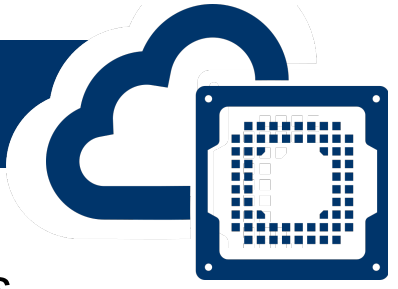



```
module deMux
( output [1:8] outVector,
  input A, B, C, D, enable);

  and v (m12, D, C, ~B, ~A, enable),
      h (m11, D, ~C, B, A, enable),
      d (m10, D, ~C, B, ~A, enable),
      l (m9, D, ~C, ~B, A, enable),
      s (m7, ~D, C, B, A, enable),
      u (m6, ~D, C, B, ~A, enable),
      c (m5, ~D, C, ~B, A, enable),
      ks (m3, ~D, ~C, B, A, enable);

assign outVector = {m3, m5, m6, m7, m9, m10, m11, m12};

endmodule


module xor8
( output [1:8] xout,
  input [1:8] xin1, xin2
);

  xor a[1:8] (xout, xin1, xin2);

endmodule
```

```
module hamDecode
( input [1:12] vIn,
  output [1:8] valueOut);

  wire c1, c2, c4, c8;

  wire [1:8] bitFlippers;

  xor (c1, vIn[1], vIn[3], vIn[5], vIn[7], vIn[9], vIn[11]),
      (c2, vIn[2], vIn[3], vIn[6], vIn[7], vIn[10], vIn[11]),
      (c4, vIn[4], vIn[5], vIn[6], vIn[7], vIn[12]),
      (c8, vIn[8], vIn[9], vIn[10], vIn[11], vIn[12]);

  deMux mux1 (bitFlippers, c1, c2, c4, c8, 1'b1);

  xor8 x1 (valueOut, bitFlippers, {vIn[3], vIn[5], vIn[6],
           vIn[7], vIn[9], vIn[10], vIn[11], vIn[12]});

endmodule
```


Share: bit.ly/cloudfpga

Continuous Assignment



- A different way to describe logic is using **`assign`** continuous assignment statements

```
module oneBitFullAdder
( output cOut, sum,
  input aIn, bIn, cIn
);

  assign sum = aIn ^ bIn ^ cIn,
         cOut = (aIn & bIn) | (bIn & cIn) | (aIn & cIn);

endmodule
```

Boolean algebra-like expression for each of two assign statements

- Continuous assignment can specify delays

```
module combiningDelays
( input a, b,
  output c
);

  wire #10 c;

  assign #5 c = ~a;

  assign #3 c = ~b;

endmodule
```

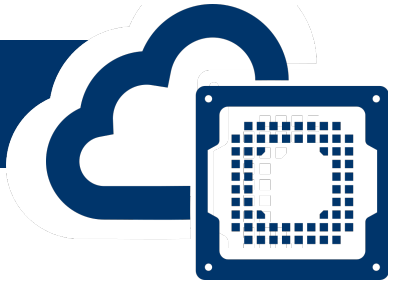
Add up delays of any wires and `assigns` when computing actual delay

```
module multiplexor
( input a, b, c, d,
  input [1:0] select,
  output e
);

  assign e = mux (a, b, c, d, select);

  function mux
  ( input a, b, c, d,
    input [1:0]select
  );

  case (select)
    2'b00: mux = a;
    2'b01: mux = b;
    2'b10: mux = c;
    2'b11: mux = d;
    default: mux = 'bx;
  endcase

  endfunction

endmodule
```

Functions, but not tasks, can be used in `assign` statements

Use default case to capture possible `x` on the inputs (not needed for synthesizable code, no `x`)

Continuous Assignment



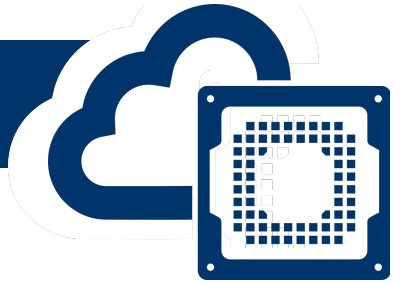
- Continuous assignment can be also used with **`inouts`** to specify high-impedance output

```
module Memory_64Kx8
( inout [7:0] dataBus,
  input [15:0] addrBus,
  input we, re, clock
);

  reg [7:0] out;
  reg [7:0] Mem [65535:0];

  assign dataBus = (~re) ? out: 16'bz;

  always @(negedge re or addrBus)
    out = Mem[addrBus];

  always @(posedge clock)
    if (we == 0)
      Mem[addrBus] <= dataBus;

endmodule
```

Use `inout` to model a shared data bus where master can drive the bus (writes) or reads the bus (reads)

Assign output to high-impedance when not sending data from memory

Mixing Behavioral and Structural Description



- Most of Verilog code written uses both structural and behavioral descriptions mixed together to describe the design:
  - Use structural for module interconnection, gate-level modeling of certain modules that need specific gate-level implementation
  - Use behavioral for describing modules where specific hardware implementation is less important

Note again, that even with gate-level specification, actual hardware will be different (e.g. using LUTs on FPGAs)

Structural description

Behavioral description

```
module sbus;

  parameter Tclock = 20,
            Asize = 5,
            Dsize = 16,
            Msize = 32;

  reg clock;
  wire rw;
  wire [Asize-1:0] addr;
  wire [Dsize-1:0] data;

  master #(Asize, Dsize) m1 (rw, addr, data, clock);

  slave #(Asize, Dsize, Msize) s1 (rw, addr, data, clock);

  initial
  begin
    clock = 0;
    $monitor ("rw=%d, data=%d, addr=%d at time %d",
    rw, data, addr, $time);
  end

  always
    #Tclock clock = !clock;

endmodule
```

Logic Delay Modeling

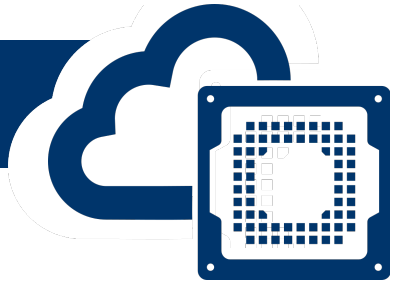
- Gate and net specification can include information about delays
  - Gates can has specified propagation delay for transition to 1, transition to 0, and transition to z, resulting in up to three delay values used with **#** operator:

```
#(d1, d2, d3)
```

  - Delays are used by simulators to determine when to update the values on output of the gates

From value	To value	2 Delays specified	3 Delays specified
0	1	d1	d1
0	x	min(d1, d2)	min(d1, d2, d3)
0	z	min(d1, d2)	d3
1	0	d2	d2
1	x	min(d1, d2)	min(d1, d2, d3)
1	z	min(d1, d2)	d3
x	0	d2	d2
x	1	d1	d1
x	z	min(d1, d2)	d3
z	0	d2	d2
z	1	d1	d1
z	x	min(d1, d2)	min(d1, d2, d3)

Minimum, Maximum, and Average Delays



- Verilog allows for three values to be specified for each of the rising, falling, and turn-off delays
  - A three-valued delay specification:

```
#(d1, d2, d3)
```

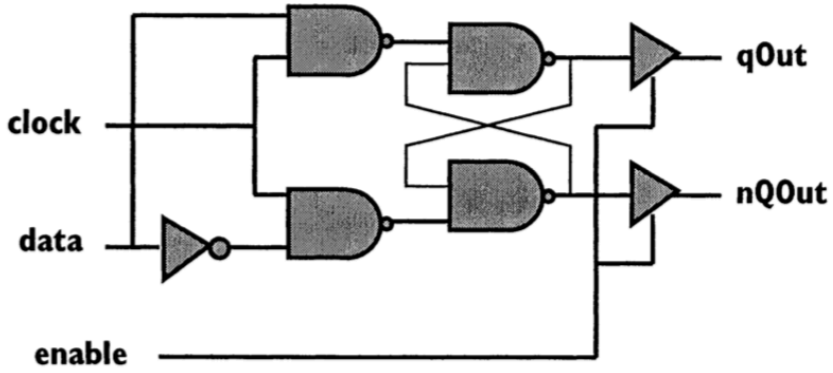
  - Can be expanded to:

```
#(dl_min: dl_typ: dl_max, d2_min: d2_typ: d2_max, d3_min: d3_typ, d3_max))
```

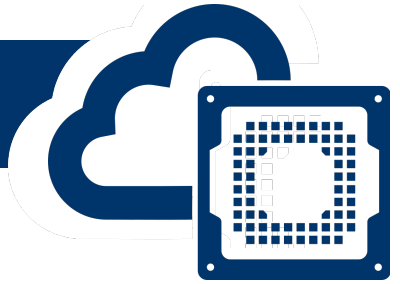
Logic Delay Modeling

Textbook example of modeling logic gate delays in a tri-state latch:

```
module triStateLatch
( output qOut, nQOut,
  input clock, data, enable
);

  tri qOut, nQOut;

  not #5 (ndata, data);

  nand #(3,5) d(wa, data, clock),
              nd(wb, ndata, clock);

  nand #(12, 15) qQ(q, nq, wa),
                 nQ(nq, q, wb);

  bufif1 #(3, 7, 13) qDrive (qOut, q, enable),
                     nQDrive(nQOut, nq, enable);

endmodule
```



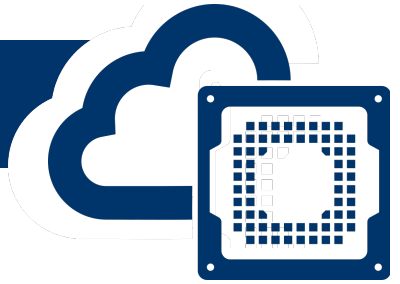
Delays Across Modules



- Verilog allows for modeling delays across a whole module using the **`specify`** block
  - **`source => destination = (delays)`** – specifies the delays, same order as for wires: transition to 1, transition to 0
  - **`(a, b *> c, d) = (delays)`** – combines multiple delay specifications into one: a to c, a to d, b to c, and c to d

```
module dEdgeFF
( input clock, d, clear, preset,
  output q
);

  specify
    specparam tRiseClkQ = 100,
              tFallClkQ = 120,
              tRiseCtlQ = 50,
              tFallCtlQ = 60;

    (clock => q) = (tRiseClkQ, tFallClkQ);

    (clear, preset *> q) = (tRiseCtlQ, tFallCtlQ);

  endspecify

  // Code of the module goes here...

endmodule
```

Specifying Time Units



- Verilog simulator works in term of time units, each # delay is in time units
  - To assign specific time unit magnitude to delays use:
    `timescale <time_unit> / <time_precision>
  - Example:
    `timescale 10 ns / 1 ns

Unit of Measurement	Abbreviation
seconds	s
milliseconds	ms
microseconds	us
nanoseconds	ns
picoseconds	ps
femtoseconds	fs

- The precision is used to quantify how fine-grained the simulator keeps track of time

Unit / precision	Delay specification	Time delayed	Comments
10 ns / 1 ns	#7	70 ns	The delay is 7 * time_unit, or 70 ns
10 ns / 1 ns	#7.748	77 ns	7.748 is rounded to one decimal place (due to the difference between 10 ns and 1 ns) and multiplied by the time_unit
10 ns / 100 ps	#7.748	77.5 ns	7.748 is rounded to two decimal places and multiplied by the time_unit
10 ns / 1 ns	#7.5	75	7.5 is rounded to one decimal place and multiplied by 10
10 ns / 10 ns	#7.5	80	7.5 is rounded to the nearest integer (no decimal places) and multiplied by 10

References

1. Donald E. Thomas and Philip R. Moorby. " The Verilog Hardware Description Language, Fifth Edition." Springer. 2002