Lectures 4:
Introduction to SystemVerilog

UCSD ECE 111

Prof. Farinaz Koushanfar

Fall 2024

Some slides are courtesy of Prof. Lin Prof. Turakia, or Prof. Karna ECE111 courses

Important announcements (Details in Canvas)

- **TA office hours:** are now MWF 9-11am for Fall'24
  - Zoom Meeting ID: 948 6397 0932; Passcode 004453
  - https://ucsd.zoom.us/j/94863970932?pwd=iXcQXbLYjOaAQTdOJlrmglCYMSyeir.1
- **TA Discussion session:** Wed 4-4:50PM (in person) Location: CNTRE 214
- **Oct 1:** Homework 1 was posted on Canvas
  - Due on Friday, **10/11/24**
- **Oct 8:** Homework 2 was posted on Canvas
  - Due on Wednesday, **10/16/24**
  - Cloudlabs is now up and running so you should all have cloud access to SW
  - **Late homework policy:** Max of 2 late days, with 20% grade reduction per day

Homework 2 overview

- You will learn how to:
  - Create parameterized modules, how to instantiate a module in another module, how to connect primary ports of 2 modules using explicit name based binding approach
  - Design functional behavior of SystemVerilog arithmetic and logical operators and understand hardware generated for each of the operator post synthesis.
- You will observe change in value of signals in sensitivity list of always block and how it impacts the behavior of the circuit
- There will be two parts for this homework:
  - **Homework-2a**: designing a synthesizable SystemVerilog Model of 4-bit ALU (Arithmetic Logic Unit)
  - **Homework-2b**: developing a synthesizable SystemVerilog code for 4-bit up down binary counter

SystemVerilog

Hello World!

Simple logic gate in SystemVerilog

- Module

```
module not_gate(in, out); // module name+ports
    // comments: declaring port type
   input in;
   output out;

    // Defining circuit functionality
   assign out = ~in;

endmodule
```

Synthesis

SystemVerilog:

```
module example(input  logic a, b, c,
               output logic y);
  assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b &  c;
endmodule
```

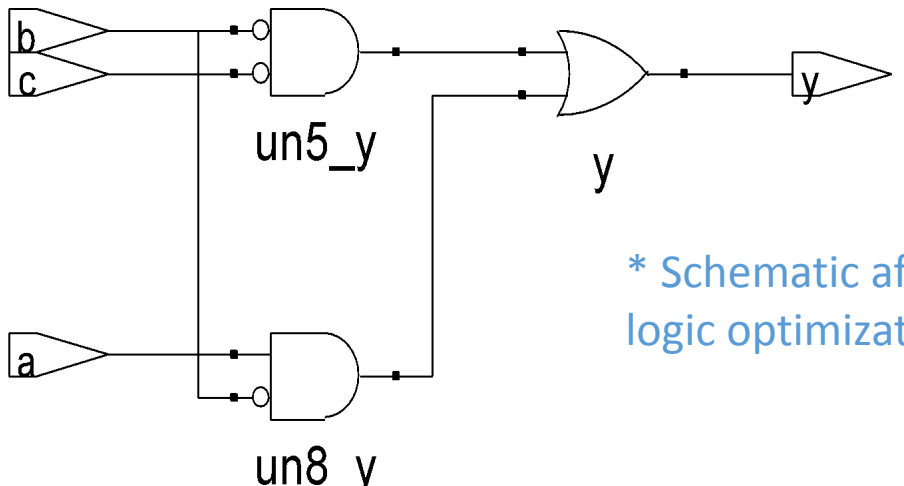
Module Abstraction:

Synthesis

SystemVerilog:

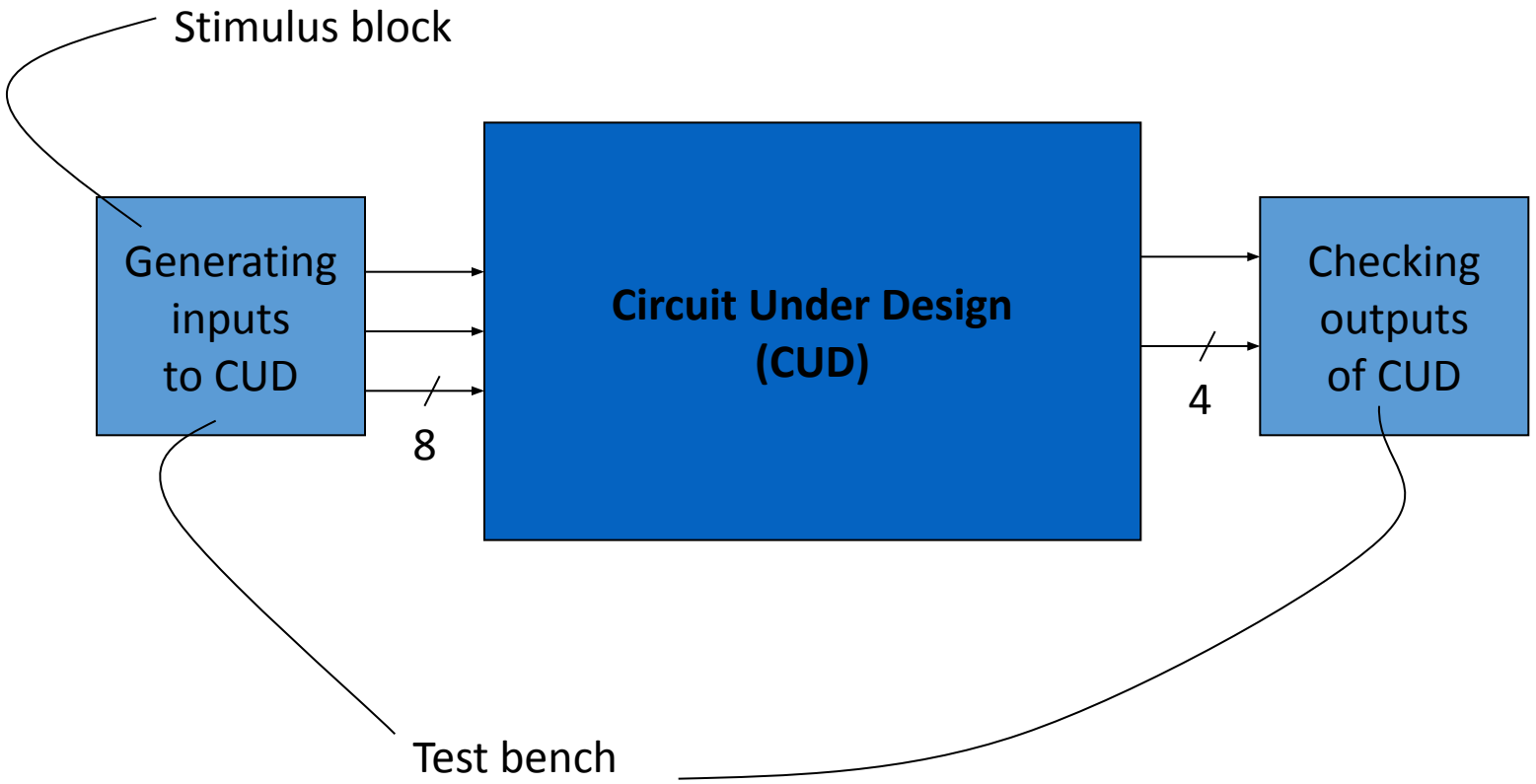
```
module example(input  logic a, b, c,
               output logic y);
  assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b &  c;
endmodule
```

Synthesis: translates into a netlist (i.e., a list of gates and flip-flops, and their wiring connections)



* Schematic after some logic optimization

Basics of digital design with HDL



Useless SystemVerilog Example

```
module useless;

    initial
        $display("Hello World!");

endmodule
```

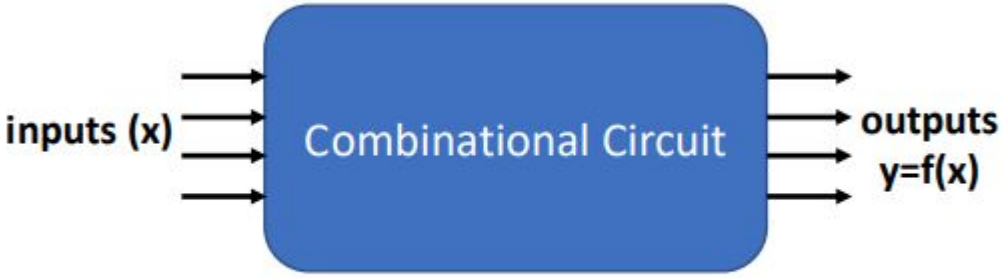
- Note the message-display statement
  - Compare to printf() in C

SystemVerilog Syntax

- Case sensitive
  - **Example:** `reset` and `Reset` are not the same signal.
- No names that start with numbers
  - **Example:** `2mux` is an invalid name
- Whitespace ignored
- Comments:
  - `// single line comment`
  - `/* multiline`
    `comment */`

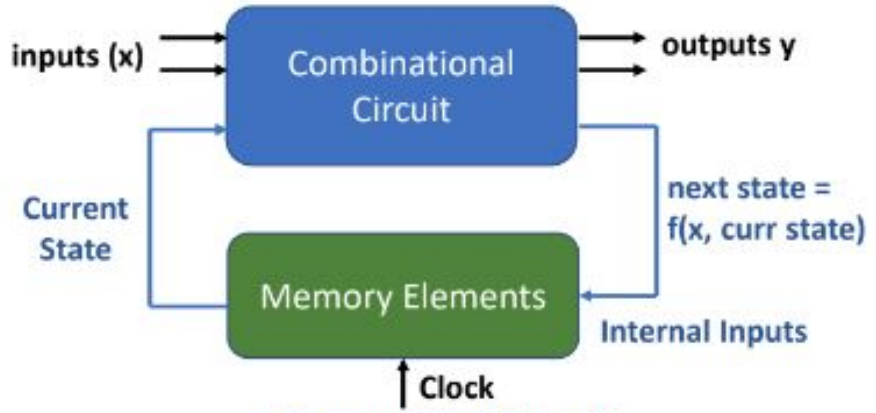
SystemVerilog Modeling Abstractions

Combinational vs Sequential Circuit



Combinational Circuit

- The output only depends on the inputs by itself at present time
- Does not store any intermediate values, therefore, no **memory elements** required
- No clock signal needed
- Behavior is described by set of output functions → **y= f(x)**
- Some examples: Full Adder, half adder, comparator, multiplexer, decoder, encoder, etc



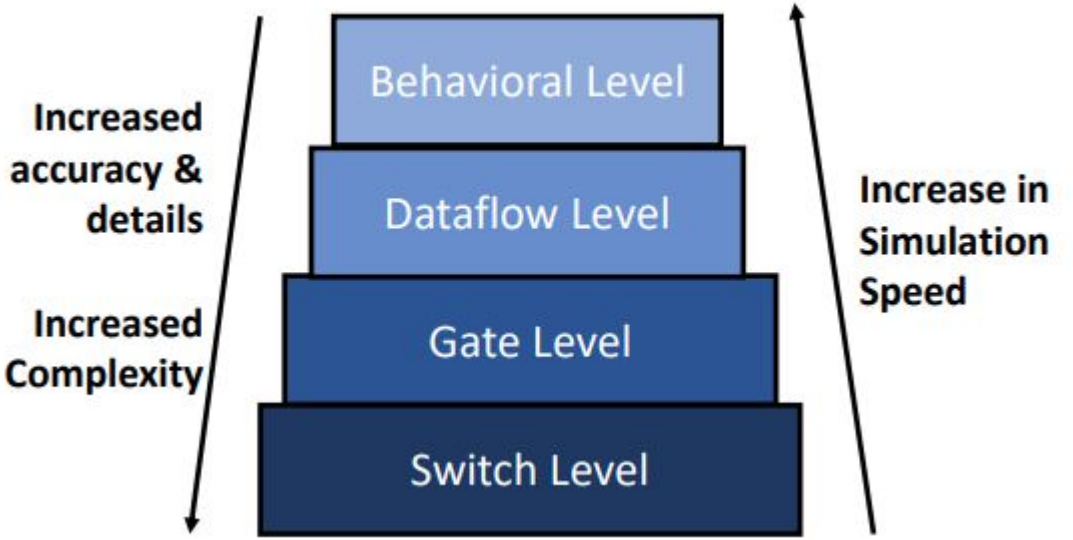
Sequential Circuit

- Present values of outputs are determined by two values, the present inputs and past state (i.e. sequence of past inputs or known as past outputs)
- Past outputs are stored in memory elements and it requires clock signal
- Behavior is described by set of output functions and set of next states functions stored in **memory**
- Examples: Flipflop, Latch, Shift Register, etc

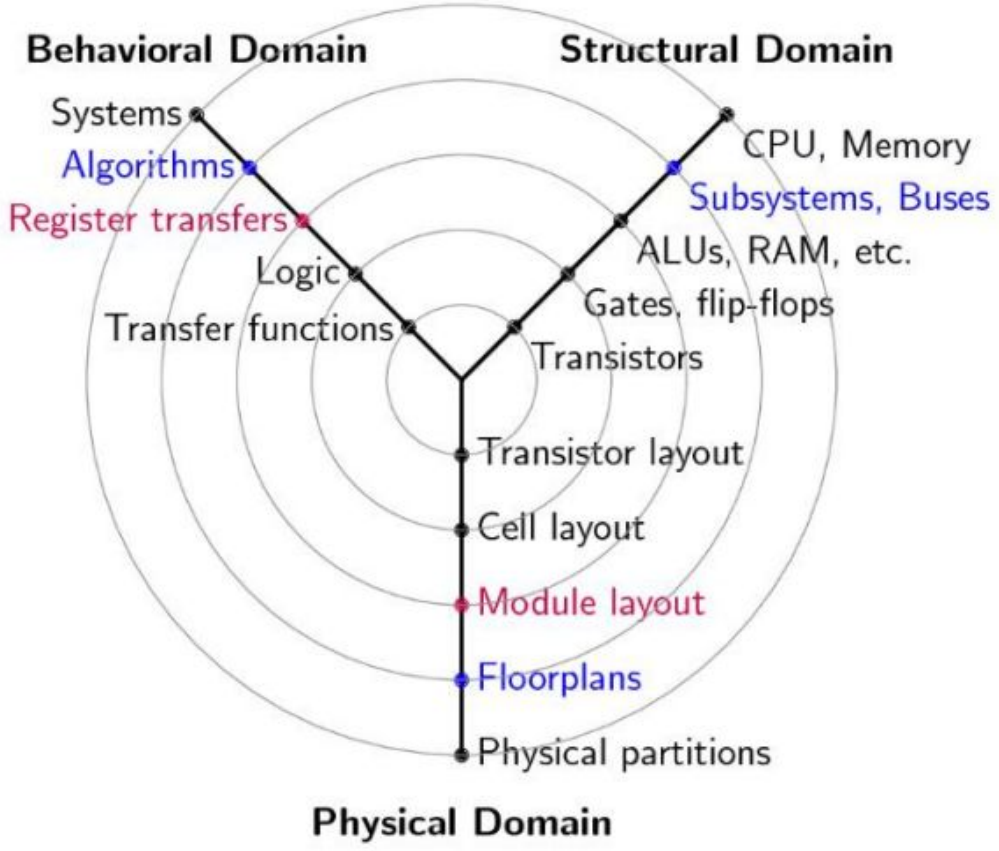
Modeling Styles in SystemVerilog

- **SystemVerilog modeling language supports three kinds of modeling styles:**
  - Behavioral level (Algorithmic or RTL level)
  - Dataflow level
  - Gate level (Structural level)
  - Switch level (Transistor level)

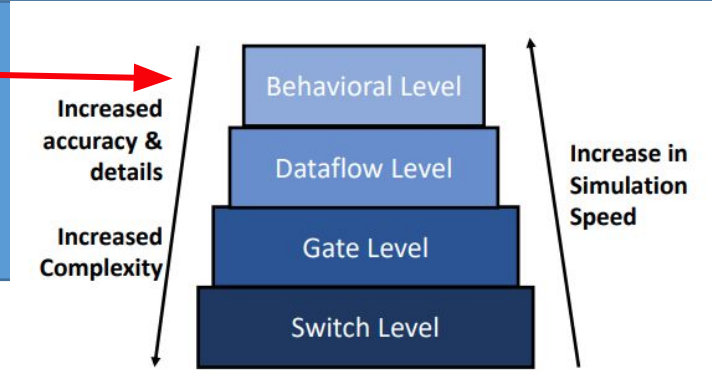
- RTL (Register Transfer Level) model utilizes combination of behavioral and dataflow styles



Design View Point Through Y-Diagram



Behavioral Level Modeling

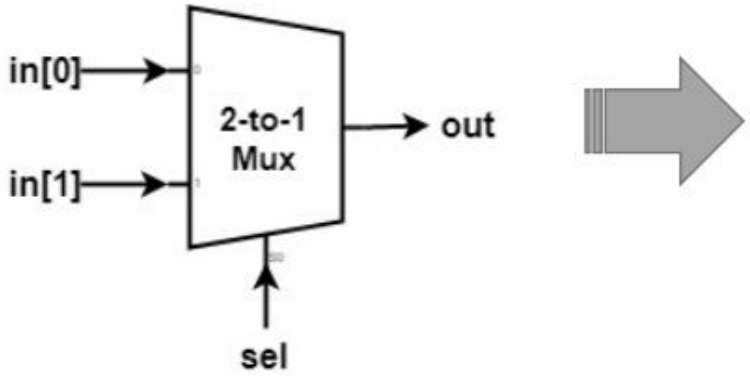


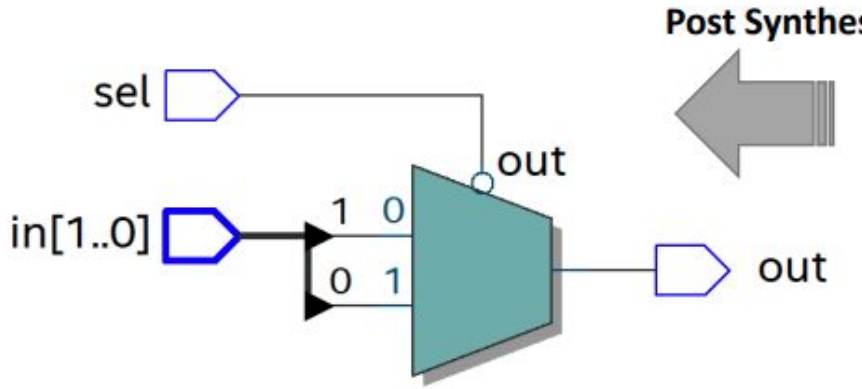
Hardware circuit is specified in terms of its expected behavior or an algorithm without concern of internal hardware implementation

- Full functionality of a complex circuit specified in **C** type natural language description
- It is **not** a cycle accurate representation of hardware circuit
- It may contain algorithms, boolean equations, truth tables (Tables of input and output values), which is also called LUT(lookup tables)
- Models at this abstraction level are also called as bus-functional or algorithmic models
- This is the **highest** level of abstraction provided by SystemVerilog HDL
- It is primarily used to model sequential circuits, but can also be used to model pure combinatorial circuits
- **Advantage :**
  - Behavioral models are faster to simulate compared to gate level models since it has less timing details

Behavioral level Modeling

Behavioral level description of a 2-to-1 Multiplexer using *"if/else"* statement



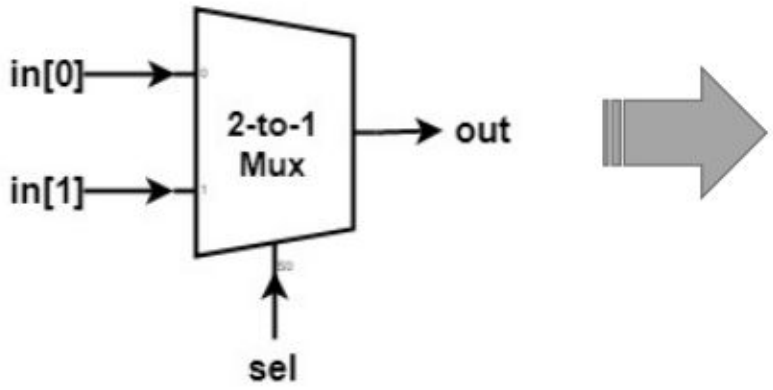


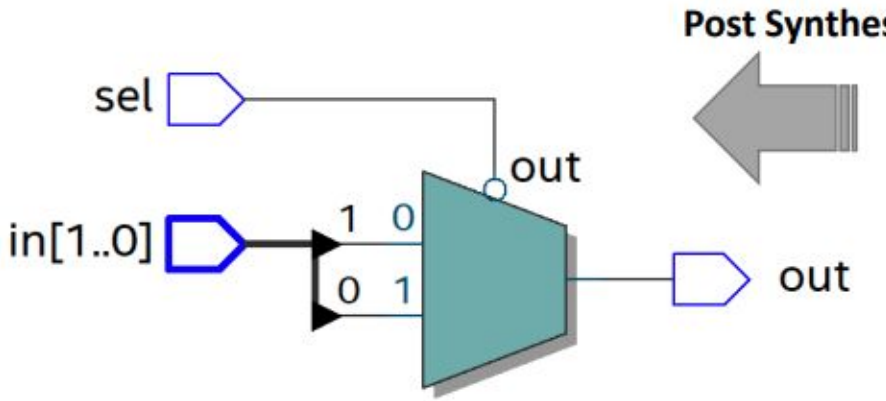

```
module mux_2x1_behavioral (
  input logic in[1:0],
  input logic sel,
  output logic out
);
  always @(sel or in)
  begin
    if(sel == 0)
      out = in[0];
    else
      out = in[1];
  end
endmodule
```

MUX implementation using if-else statement

Behavioral level Modeling

Behavioral level description of a 2-to-1 Multiplexer using *"case"* statement





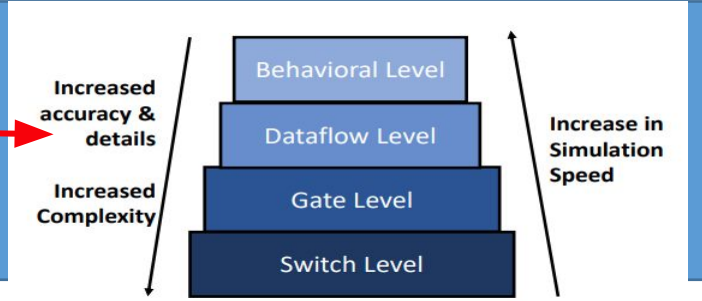

```
module mux_2x1_behavioral (
  input logic in[1:0],
  input logic sel,
  output logic out
);
  always @(sel or in)
  begin
    case(sel)
      1'b0 : out = in[0];
      1'b1 : out = in[1];
      default : out = 1'bx;
    endcase
  end
endmodule
```

MUX implementation using case statement

Behavioral Level Modeling

- How does behavioral modeling work?
  - “**initial**” and “**always**” are most common constructs.
  - All the other behavioral statements appear only inside initial and always blocks
  - What can be in a module?
    - a number of initial or always blocks,
    - one or more procedural statements.
  - Initial blocks execute only once and in the beginning (t=0)
  - Always blocks execute repeatedly (t=0 and t>0)
  - initial and always blocks execute concurrently (i.e. to model parallelism)
- Behavioral model even though mimics the intended hardware behavior but it may not be necessarily synthesizable ▪
  - It can utilize **full** set of constructs in SystemVerilog (both synthesizable and non synthesizable constructs)

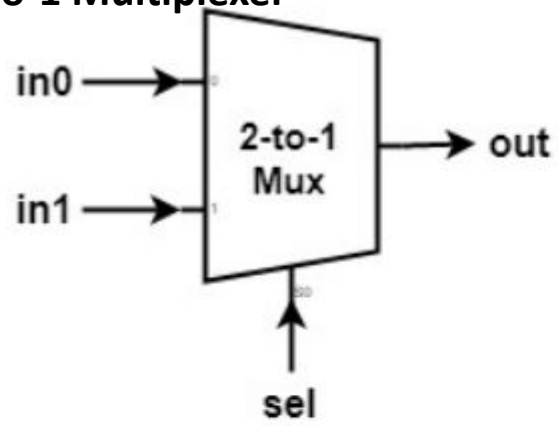
Dataflow Modeling

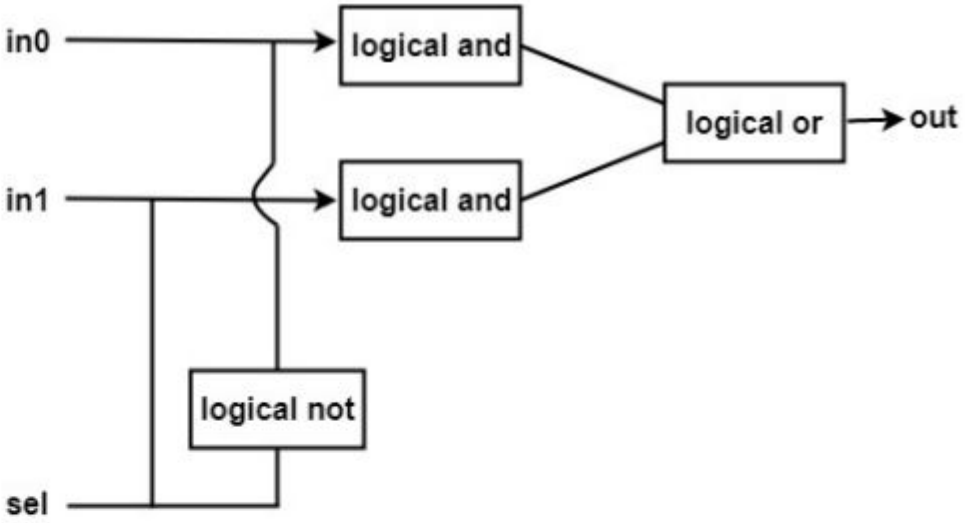


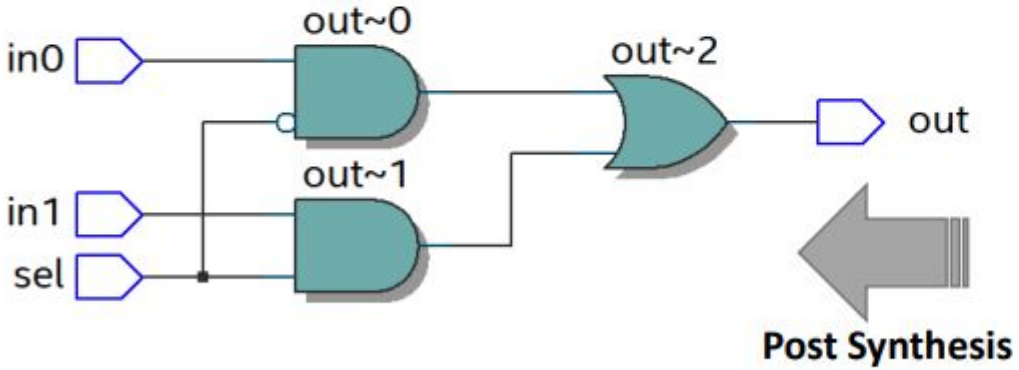
- In dataflow modeling, module is designed by describing how data flows through a circuit
  - This is a higher level of abstraction than the gate level, therefore, no hardware implementation details required
  - Using **logical equations** or **boolean expressions** for data processing.
  - Used for combinational circuits in most cases.
  - Can be translated into a gate level design through process of logic synthesis
- In dataflow modeling most of the design is implemented using continuous assignments, which are used to drive a value onto a net.
  - Continuous statements are always active statements.
  - Continuous assignments are made using the keyword assign. Syntax Format :
    - ***assign*** *[delay_specification] LHS = RHS*
    - Example : ***assign*** *#20 out = InA & InB;*
- Note : The RHS expression is evaluated whenever one of its operands changes. Then the result is assigned to the LHS, which is usually a wire/bus

Dataflow Modeling

Dataflow description of a 2-to-1 Multiplexer



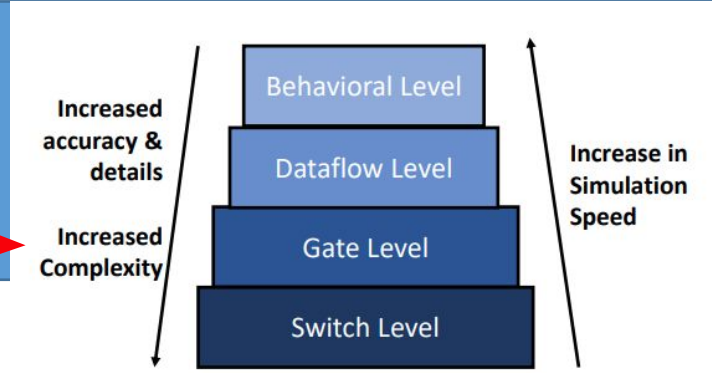





```
module mux_2x1_df (
  input logic in0,in1,sel,
  output logic out);
  assign out = (!sel && in0) || (sel && in1);
endmodule
```

Mux representation using logical expression.
No gates or hardware implementation details specified.

Gate Level Modeling

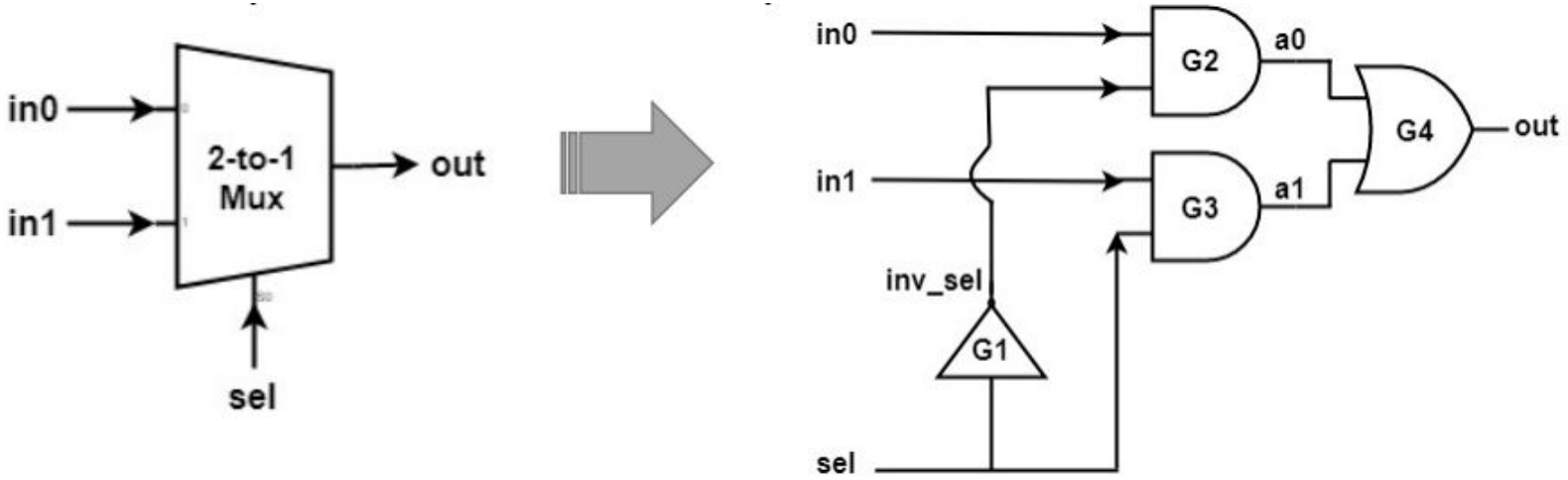


Gate level module is described with **logic gates** and **interconnections** between these gates.

- Resembles a schematic drawing with components connected with signals
- Low level of modeling abstraction
- Gate level module focus on hardware implementation details & accuracy:
  - **Actual logic gates**
  - **Timing** (gate propagation delays).

Gate level Modeling

Gate-level description of a 2-to-1 Multiplexer

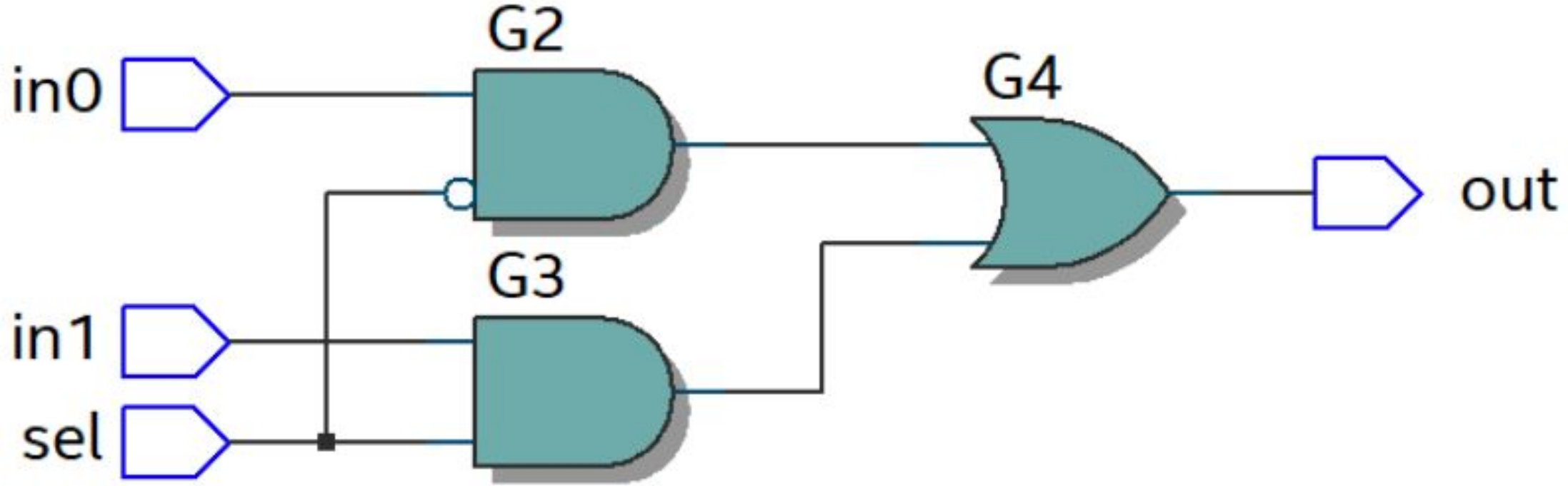



```
module mux_2x1_gatelevel(
  input logic in0, input logic in1, input logic sel, output logic out
);
  wire a0, a1, inv_sel;
  not G1(inv_sel, sel);
  and G2(a0, in0, inv_sel);
  and G3(a1, in1, sel);
  or  #1.5  G4(out, a0, a1);    // or gate defined with 1.5 time units of propagation delay
endmodule
```

Mux representation using built-in gate level primitives

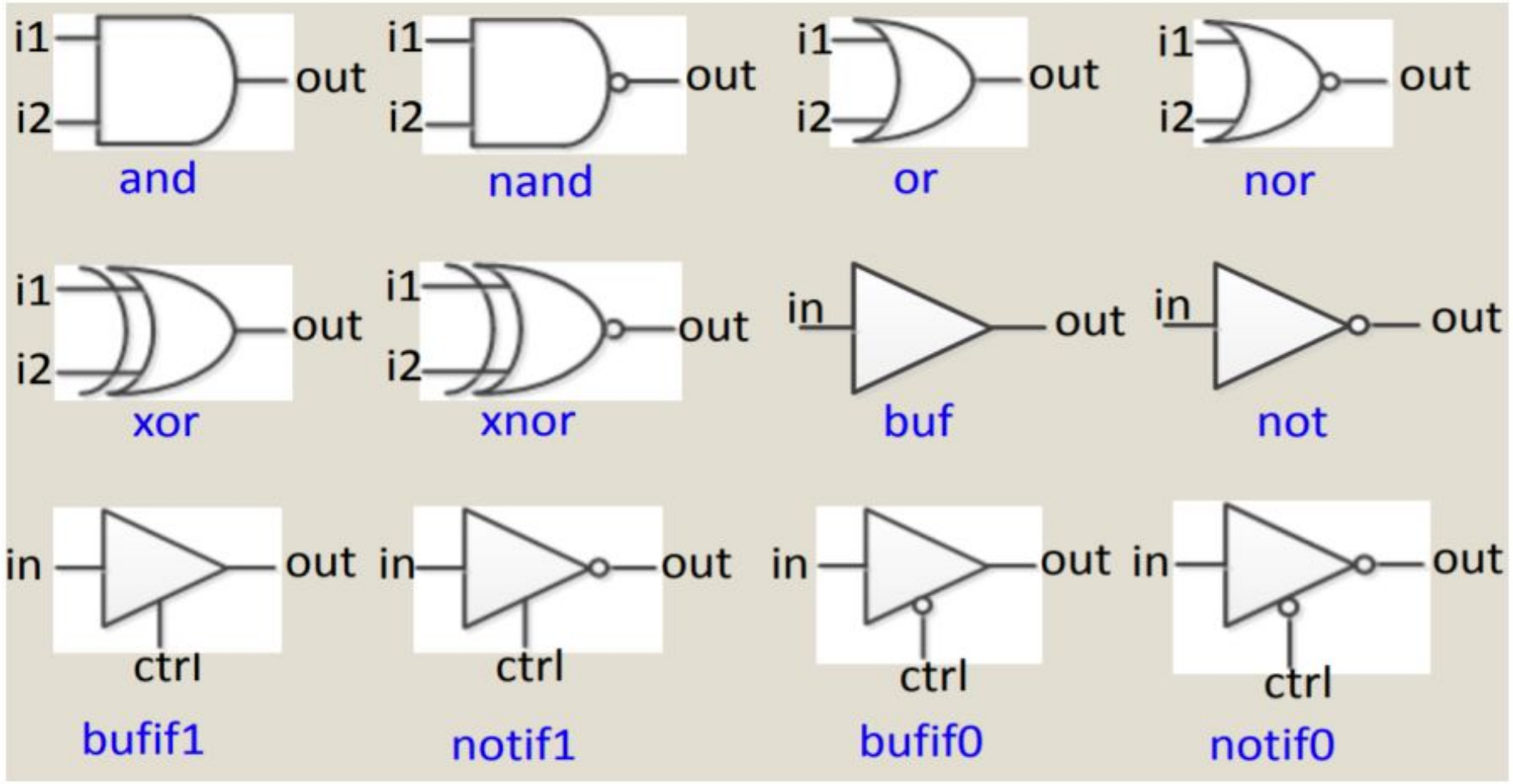
Gate level Modeling

Gate Level description of a 2-to-1 Multiplexer post synthesis schematic



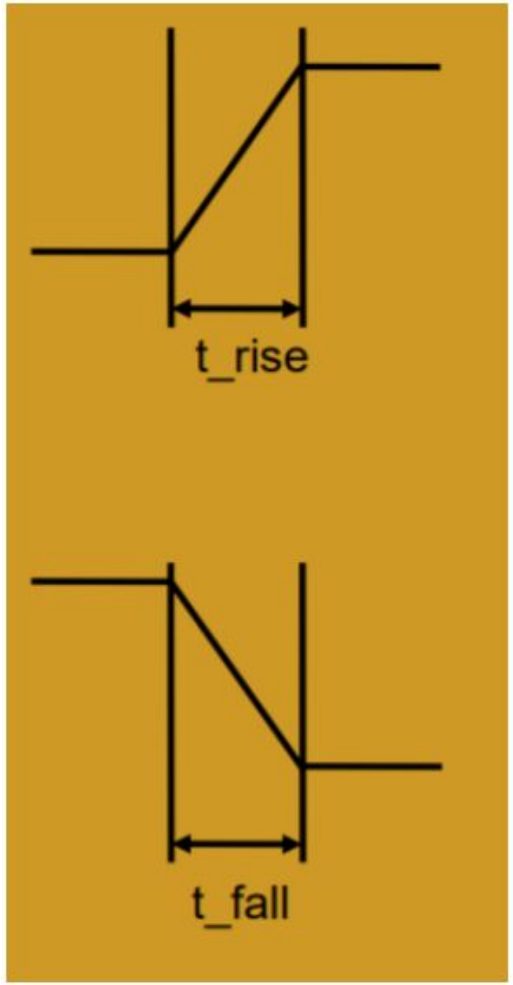
Gate level Modeling

- SystemVerilog supports modeling digital logic using built-in gate-level primitives
  - Gate level primitives can closely approximate silicon implementation
  - Gate-level models are provided by the silicon manufacturing vendor in case of ASIC or by



Gate level Modeling

- Gate level primitives can be modeled with or without propagation delays
  - If no delay is specified, output will be reflected immediately. Also called zero delay gate level model.
  - Gate-level modeling can represent the propagation delays of transistors that would be in an actual silicon with a high degree of **accuracy**.
- Types of delays which can be specified in each primitive
  - Rise delay: From (0 or X) to 1 (for a gate output)
  - Fall delay: From (0 or X) to 0 (for a gate output)
  - Turn-off delay: a gate output transition to the high impedance value **(Z)** from another value (0,1 or X).



Gate level Modeling

- **If one delay value is specified – its value is used for all gate's transitions**

// Delay is equal to trans_delay (or transport delay) for all transitions

nand #(3) g1 (out, in1, in2) ; → **3 time units delay**

- **If two delay values are specified – they refer to gate's rise and fall delay values respectively**

// Rise and Fall delays are specified

and #(2, 3) g2 (out, in1, in2) ; → **rise=2, fall=3 time units**

- **If three delay values are specified – they refer to gate's rise, fall and turn-off delay values respectively**

// Rise, Fall and Turn-off delays (or trans_delay) are specified

bufif0 #(2, 3, 4) b1(out, in, control) ; → **rise=2, fall=3, turn_off=4 time units**

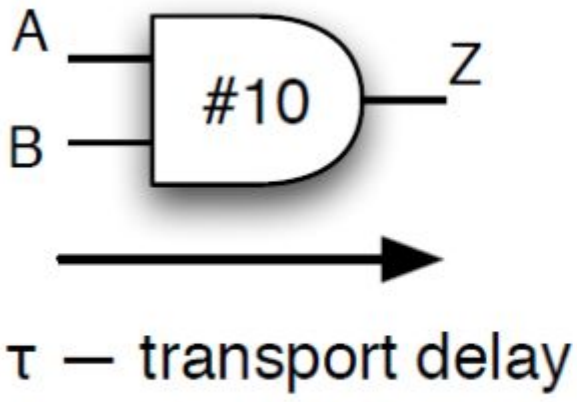
- **Default gate delay value is zero**

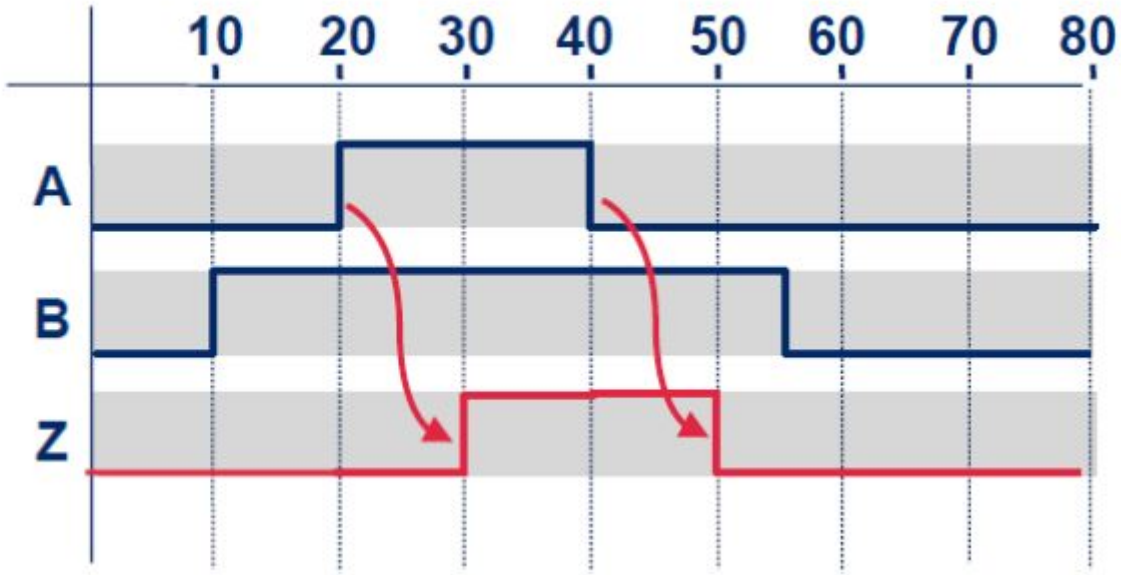
// Zero delay

buf b1(a, b) ; → **rise=0, fall=0, turn_off=0 time units**

Gate level Modeling

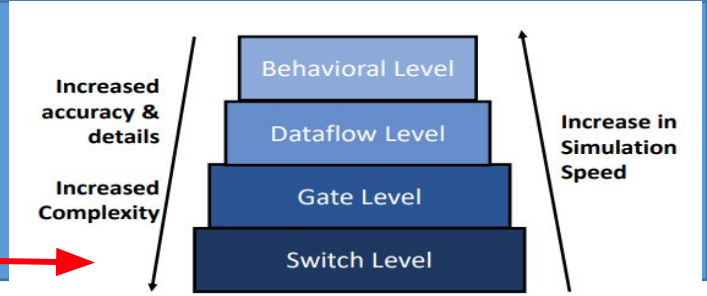
- Example: AND gate
  - The output follows the inputs according to the AND function after a certain delay.
  - Delay (#10 here) is the input to output("transport") delay





- **The delay is defined in time units, could be any time length,** e.g. 1ps, 100ns, 1ms, etc

Switch Level Modeling



- **Switch level modeling consists transistors, switches, storage nodes, and the interconnections between them**
  - Lowest modeling level of abstraction provided by SystemVerilog
  - Describe the detail interconnection inside of gates
  - Can represent actual silicon implementation → the most accurate representation of design
  - At this level designer requires knowledge of transistor-level implementation details

- **For switch-level modeling, SystemVerilog language provides switch primitives, (such as pmos, cmos and nmos), resistive switch primitives (such as rpmos, rnmos, rcmos) and capacitive nets.**
  - Only **digital systems** can be modeled at switch level using Standard SystemVerilog
  - Digital simulators does not accurately reflect transistor behavior
  - Slowest to simulate compared to other model levels
  - Typically **not used** in ASIC and FPGA design flows with SystemVerilog
    - Used only in some cases such as leaf level modeling

Switch Level Modeling

- **SystemVerilog supports two types of switch primitives :**
  - **Ideal switch :**
    - When switch is closed(ON), there is zero resistance, hence no signal degradation
  - **Resistive switch :**
    - When switch is closed(ON), there is low resistance, hence signal degradation
    - When signal passes resistive switch signal strength decreases
    - Resistive switch primitives in SystemVerilog starts with "**r**" such as **r**nmos, **r**pmos
- **Some of the Switch primitives supported in SystemVerilog are mentioned in table below:**

Switch Type	SystemVerilog Switch Primitives
Ideal MOS switches	pmos, nmos, cmos
Resistive MOS switches	rpmos, rnmos, rcmos
Ideal Bidirectional switches	tran, tranifo0, tranif1
Resistive Bidirectional switches	**r**tran, **r**tanif0, **r**tranif1
Power and Ground nets	supply1 and supply0
Pullup and Pulldown	pullup and pulldown

Switch Level Modeling

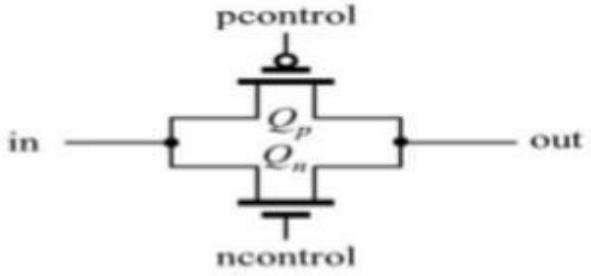
- nmos, pmos and cmos switches

nmos		control			
		0	1	x	z
in	0	z	0	L	L
	1	z	1	H	H
	x	z	x	x	x
	z	z	z	z	z

(a) nMOS switch

pmos		control			
		0	1	x	z
in	0	0	z	L	L
	1	1	z	H	H
	x	x	z	x	x
	z	z	z	z	z

(b) pMOS switch



(a) Symbol

control		data			
n	p	0	1	x	z
0	0	0	1	x	z
0	1	z	z	z	z
0	x	L	H	x	z
0	z	L	H	x	z
1	0	0	1	x	z
1	1	0	1	x	z
1	x	0	1	x	z
1	z	0	1	x	z
x	0	0	1	x	z
x	1	L	H	x	z
x	x	L	H	x	z
x	z	L	H	x	z
z	0	0	1	x	z
z	1	L	H	x	z
z	x	L	H	x	z
z	z	L	H	x	z

Switch Instance Synax :

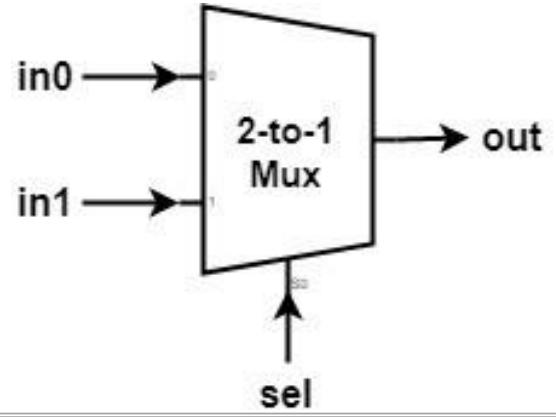
nmos <instance name> (out, in, ncontrol);

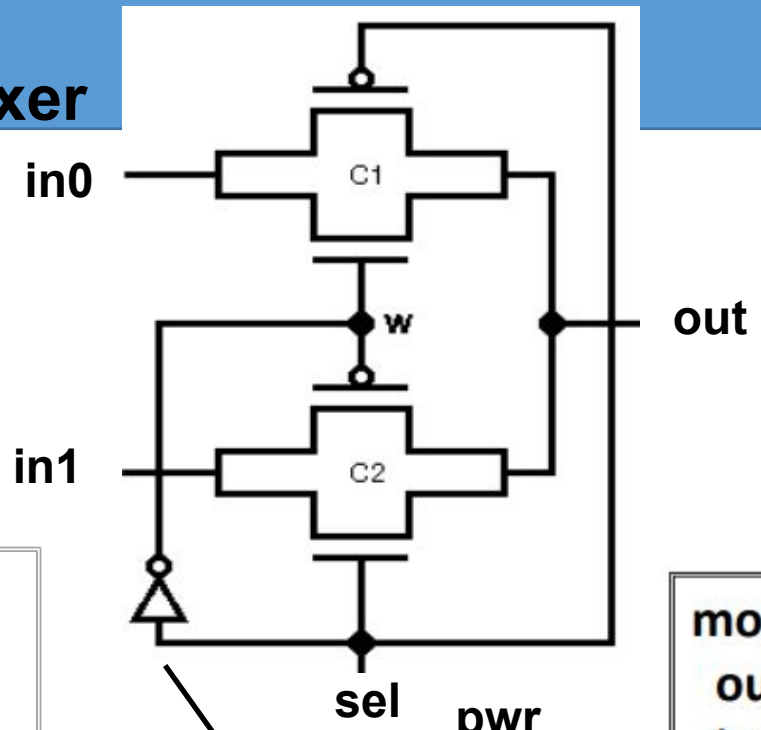
pmos <instance name> (out, in, ncontrol);

cmos <instance name> (out, in, ncontrol);

Switch Level Modeling

Switch level description of a 2-to-1 Multiplexer

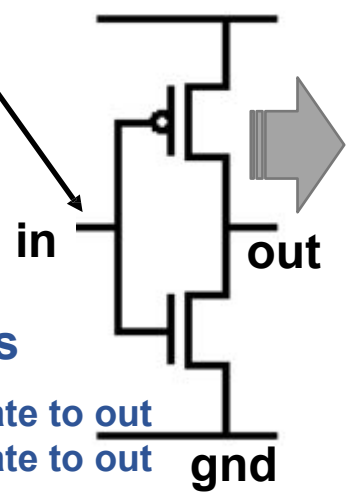





```
module mux_2x1_switchlevel (
  input in0, input in1, input sel, output out
);
  wire w;
  inv G1(w, sel);// invertor switch model instantiated
  cmos  C1(out,  in0,  w,  sel);
cmos  C2(out,  in1,  sel,  w);
endmodule
```

SV Built-in switch primitives instantiated

Inverter modeled using switch primitives

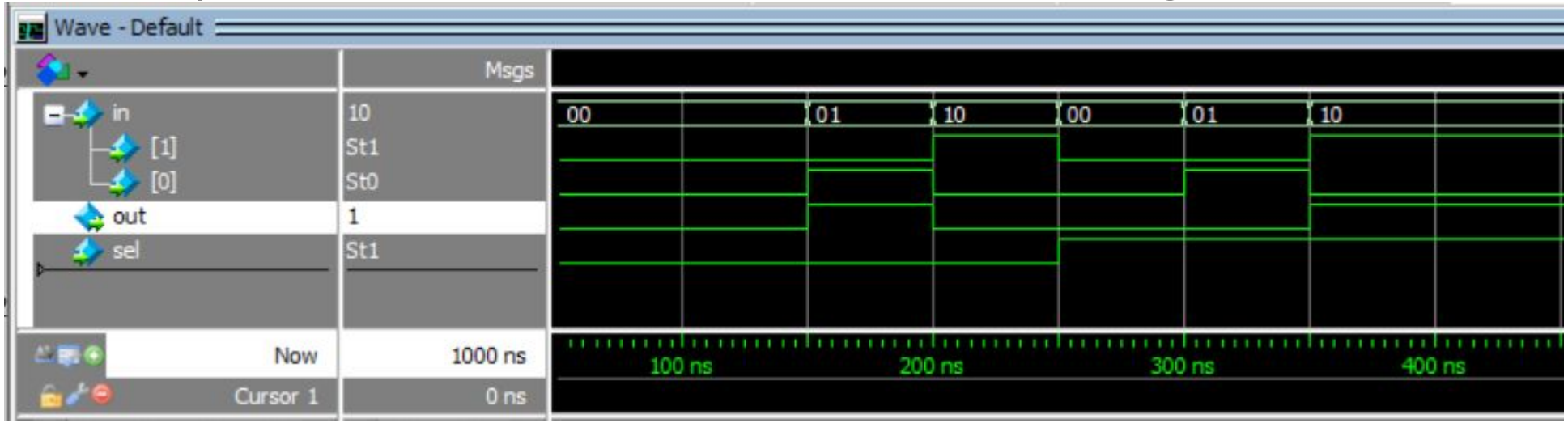



```
module inv(out, in);
  output out;
  input in;
  supply1 pwr;
  supply0 gnd;
  pmos (out, pwr, in);
  nmos (out, gnd, in);
endmodule
```

- When sel=0 then w = 1 and cmos C1 is ON and cmos C2 is OFF hence in0 will propagate to out
- When sel=1 then w = 0 and cmos C1 is OFF and cmos C2 is ON hence in1 will propagate to out

2to1 Mux Simulation Result

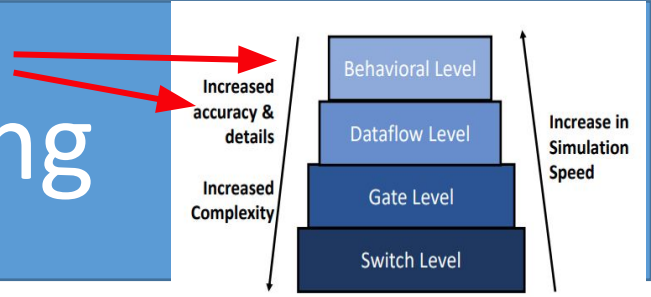
- **Waveform (Same results for behavioral, dataflow and gatelevel model simulation)**



- **Simulation output log**

```
Vsim > run 500ns
# time=0   , in=00 sel=0 out=0
# time=150, in=01 sel=0 out=1
# time=200, in=10 sel=0 out=0
# time=250, in=00 sel=1 out=0
# time=300, in=01 sel=1 out=0
# time=350, in=10 sel=1 out=1
```

RTL (Register Transfer Level) Modeling



- RTL modeling utilizes **combination of behavioral and dataflow modeling**
  - **Register:** Storage element like Flipflop, latches
  - **Transfer:** Transfer data between input, output and register using combinational logic.
  - **Level:** Level of Abstraction modeled using HDL
- RTL is a **Cycle accurate** model unlike behavioral model
- Does **not contain low level hardware** implementation details
- Complex and larger designs can be **quickly and concisely modeled** using RTL modeling technique compared to gate level modeling
- RTL models do simulate **faster** than gate-level and **switch-level** models, making it possible to verify larger and more complex designs
- Two primary constructs for RTL modeling :
  - continuous assignments (*"assign"*) and always procedural blocks (*"always"*)
  - RTL code is **easy to read** and interpret compared to gate-level code
- RTL model **supports vectors** (multi-bit signal) unlike gate-level and switch-level modeling which works on scalar signals in SystemVerilog

Behavioral vs RTL Level Modeling

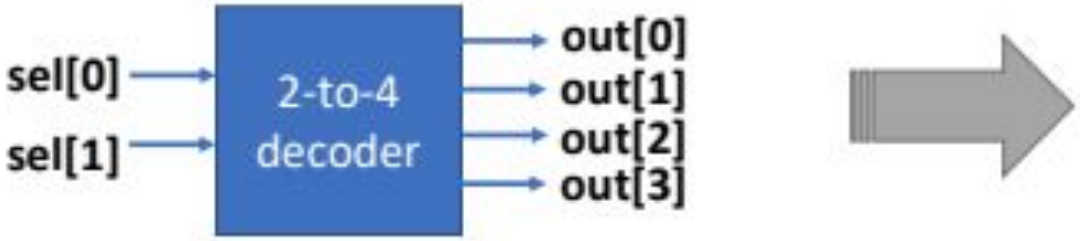
- **Behavioral model describes what intended design does. An RTL model describes how it does it.**
  - For example, an RTL half adder must describe the registers for the operands, the carry method, and a clock. It may be done in various ways to balance speed and accuracy
  - For a behavioral adder, A + B = C is sufficient. A + B = C also simulates a lot faster than all those gates and registers in the RTL model
- **Behavioral model are similar to RTL model since both uses** *always* **procedural blocks, however they differ :**
  - RTL model executes its programming statements in **a single clock cycle**, or in zero cycles if combinational logic. While behavioral block can take **arbitrary number of clock cycles** to execute is statements.

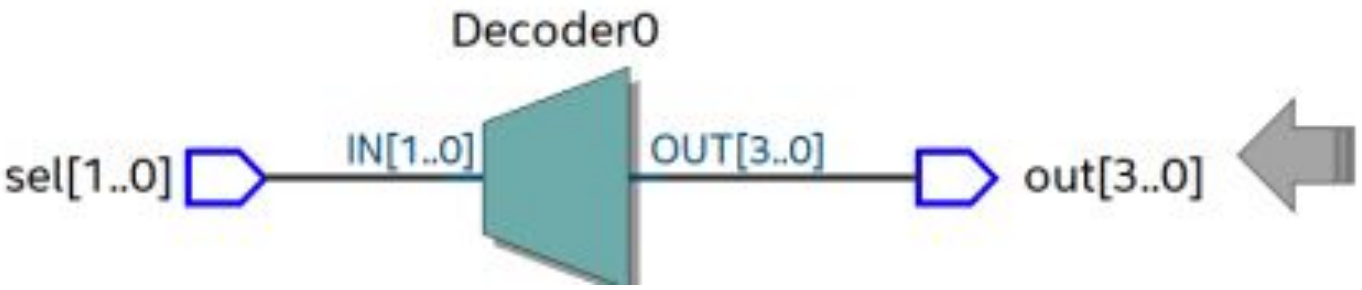
Behavioral vs RTL Level Modeling

- **RTL (Register-Transfer-Level) model is synthesizable.**
  - Hardware functionality is described using ***always*** blocks and ***assign*** statements that are synthesizable (can be translated into gate level).
  - RTL model follows some constraints that are understandable by RTL synthesis compilers or tools.
  - It's also called synchronous finite state machine(FSM) description
- **Behavioral though mimics the desired functionality of the hardware but not necessarily synthesizable.**
  - Most of the behavioral modeling is done using two important constructs: ***initial*** and ***always***
  - There is no strict rules as long as the code generates the desired behavior
  - A behavioral model can utilize full SystemVerilog Language and may not be synthesizable.

Behavioral Model of a Decoder

- Behavioral level description of a 2-to-4 decoder






```
module decoder_2x4_behav(
 input logic[1:0] sel,  output
 logic[3:0] out);  always @(sel or
 out)  begin
   case (sel)
    2'b00  : out = 4'b0001;
    2'b01  : out = 4'b0010;
    2'b10  : out = 4'b0100;
    2'b11  : out = 4'b1000;
    default : out = 4'b0000;
   endcase  end
endmodule
```

Dataflow Model of a 2-to-4 decoder

Truth Table

sel[0]	sel[1]	out[3]	out[2]	out[1]	out[0]
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

```
module decoder_2x4_dataflow(
 input  logic[1:0] sel,
 output logic[3:0] out);
   assign out[0] = (!sel[0]) && (!sel[1]);
   assign out[1] = (sel[0]) && (!sel[1]);
   assign out[2] = (!sel[0]) && (sel[1]);
   assign out[3] = (sel[0]) && (sel[1]);
endmodule
```

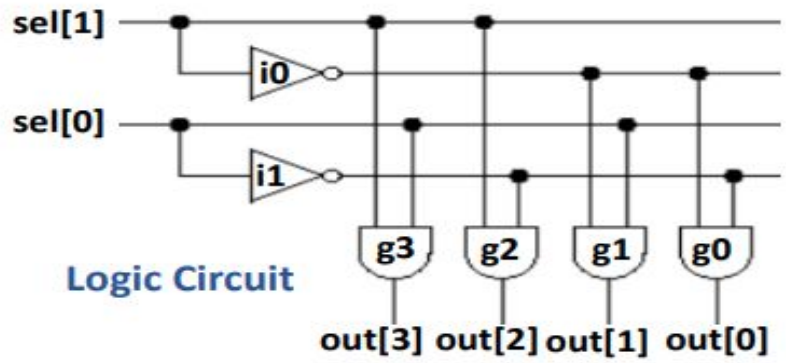
$out[0] = \overline{sel[0]}.\overline{sel[1]}$

$out[1] = \overline{sel[0]}.sel[1]$

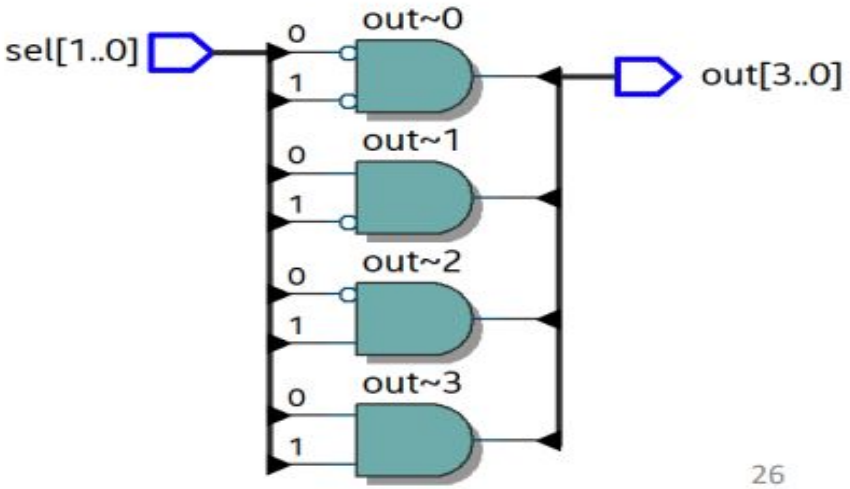
$out[2] = sel[0].\overline{sel[1]}$

$out[3] = sel[0].sel[1]$

Boolean Expression



Logic Circuit



Gate level Model of a 2-to-4 decoder

- Gate level description of a 2-to-4 decoder

Truth Table

sel[0]	sel[1]	out[3]	out[2]	out[1]	out[0]
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

```
module decoder_2x4_gate(
 input  logic[1:0] sel,
 output logic[3:0] out);
 wire w0, w1;
   not i0(w0, sel[0]);
   not i1(w1, sel[1]);
   and g0(out[0], w1, w0);
   and g1(out[1], w1, sel[0]);
   and g2(out[2], sel[1], w0);
   and g3(out[3], sel[1], sel[0]);
endmodule
```

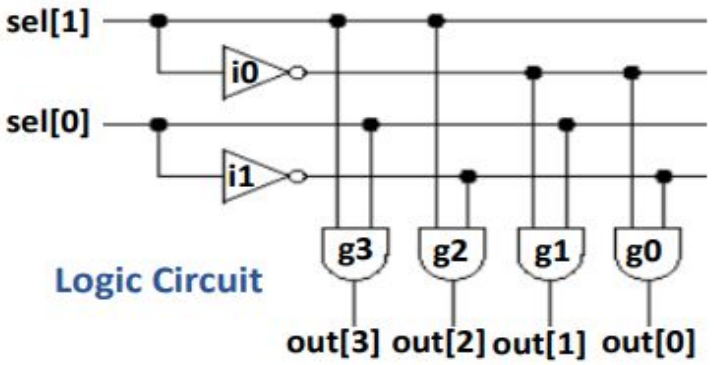
$out[0] = \overline{sel[0]}.\overline{sel[1]}$

$out[1] = \overline{sel[0]}.sel[1]$

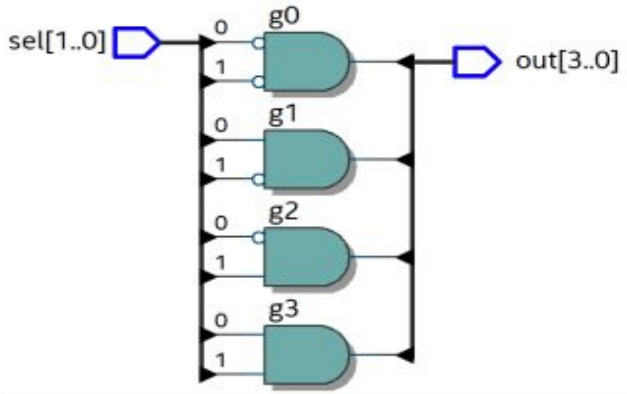
$out[2] = sel[0].\overline{sel[1]}$

$out[3] = sel[0].sel[1]$

Boolean Expression

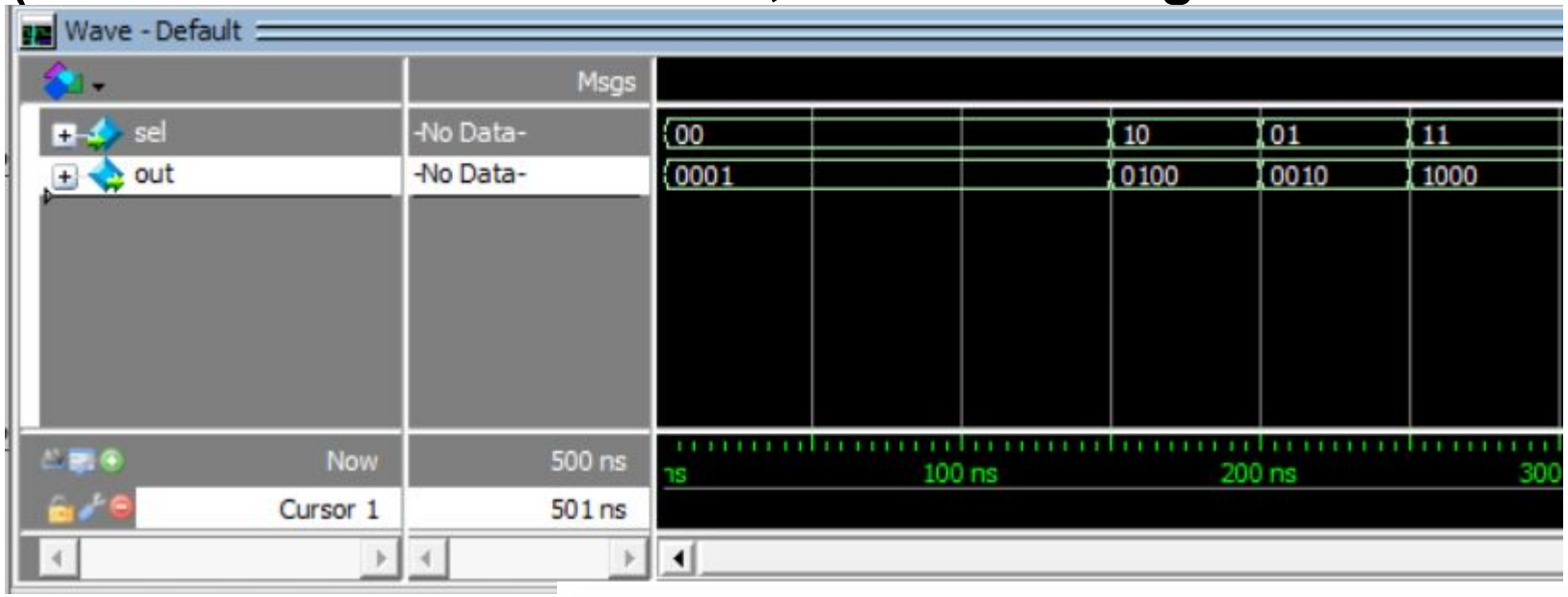


Logic Circuit



2-to-4 Decoder Simulation Result

- **Waveform (Same results for behavioral, dataflow and gatelevel model simulation)**



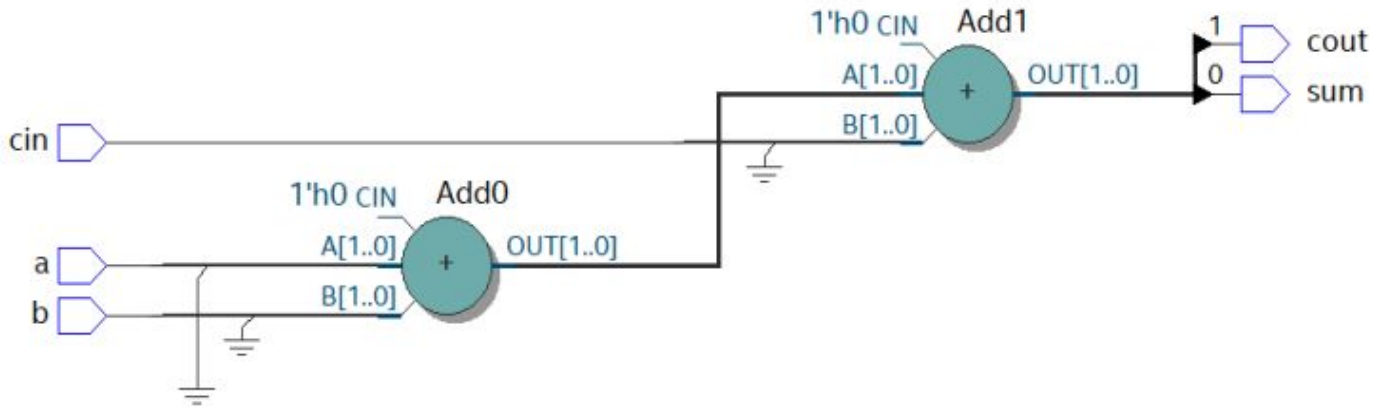
- **Simulation output log**

```
Vsim > run 500ns
# time=0   , sel=00  out=0001
# time=150, sel=10  out=0100
# time=200, sel=01  out=0010
# time=250, sel=11  out=1000
```

Behavioral Model of Full Adder

- Behavioral level description of Full Adder


```
module fulladder_behav(
  input logic a, b, cin,
  output logic sum, cout
);
  assign {cout, sum} = a + b + cin;
endmodule
```



Dataflow level Model of Full Adder

- Dataflow level description of Full Adder

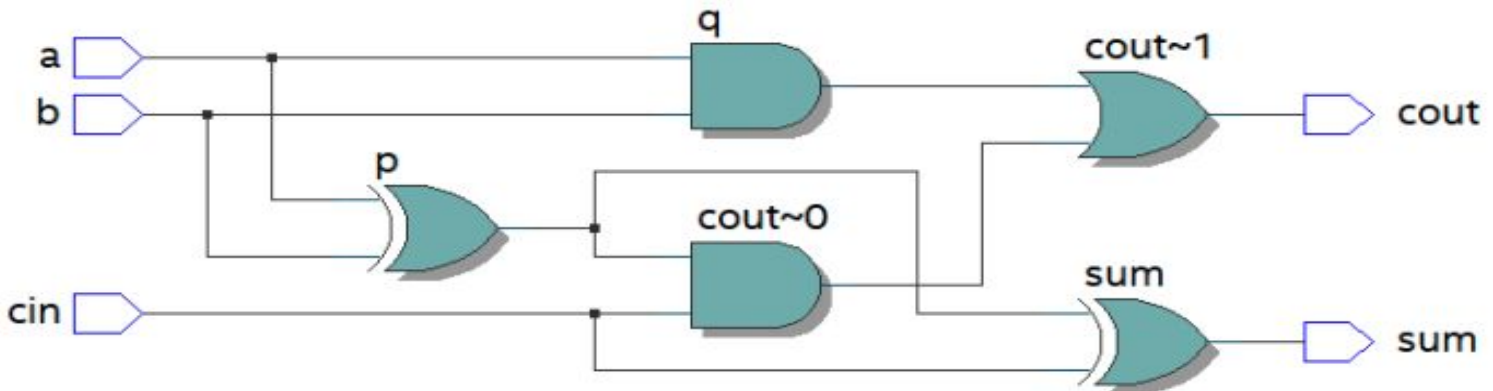
Full Adder Truth Table

inputs			outputs	
a	b	cin	sum	cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Boolean Expression for Full Adder based on Truth Table

sum = ((a xor b) xor cin);
cout = (a and b) or ((a xor b) and cin)

```
module fulladder_dataflow(
  input logic a, b, cin,
  output logic sum, cout
);
  logic p, q;
  assign p = a ^ b;
  assign q = a & b;
  assign sum = p ^ cin;
  assign cout = q | (p & cin);
endmodule
```



Gatelevel Model of Full Adder

- Gate level description of Full Adder

Full Adder Truth Table

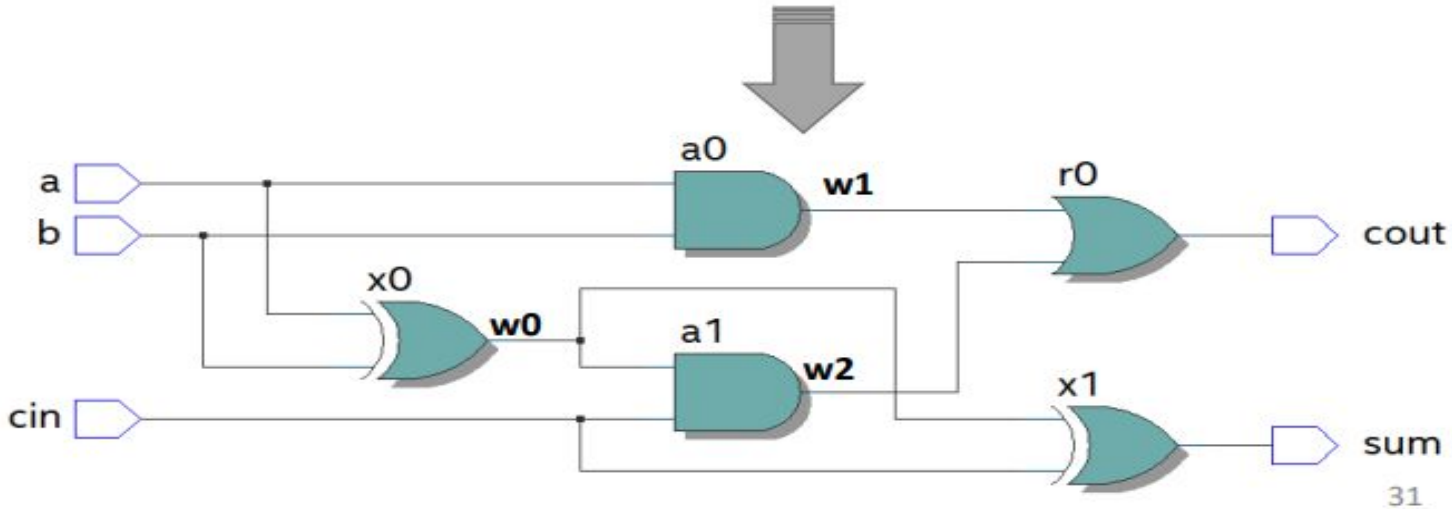
inputs			outputs	
a	b	cin	sum	cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Boolean Expression for Full Adder based on Truth Table

sum = ((a xor b) xor cin);

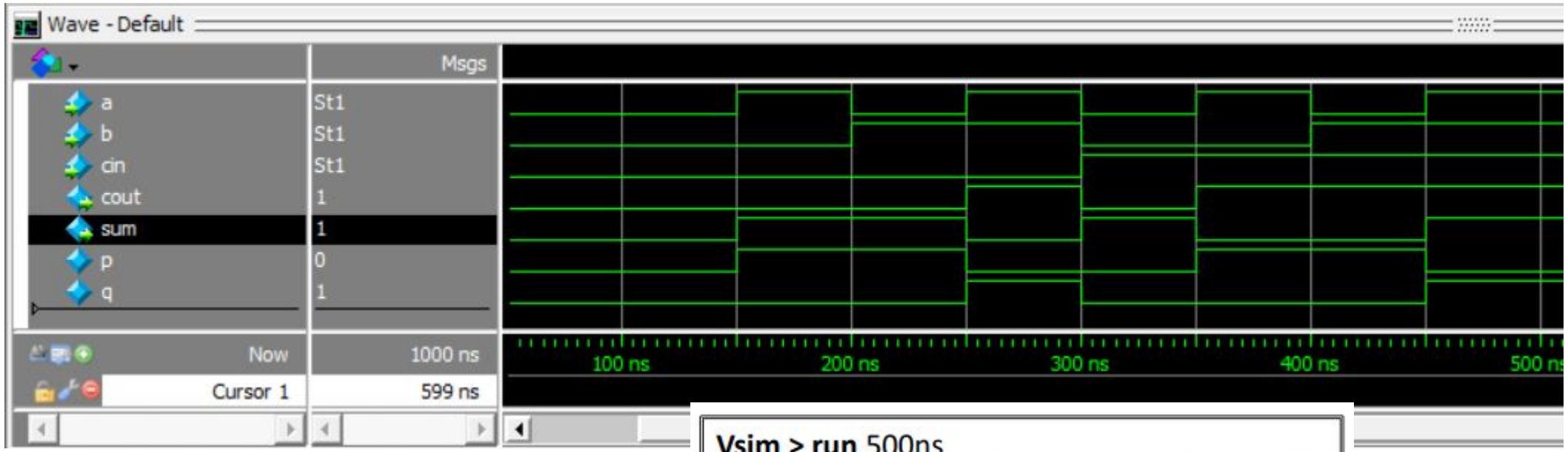
cout = (a and b) or ((a xor b) and cin)

```
module fulladder_gatelevel(
  input logic a, b, cin,
  output logic sum, cout
);
   wire w0, w1, w2;
   xor x0(w0, b, a);
   and a0(w1, b, a);
   and a1(w2, w0, cin);
   or r0(cout, w2, w1);
   xor x1(sum, w0, cin);
endmodule
```



FullAdder Simulation Result

- **Waveform (Same results for behavioral, dataflow and gatelevel model simulation)**



- **Simulation output log**

```
Vsim > run 500ns
# time=0     a=0  b=0  c=0  sum=0  cout=0
# time=150   a=1  b=0  c=0  sum=1  cout=0
# time=200   a=0  b=1  c=0  sum=1  cout=0
# time=250   a=1  b=1  c=0  sum=0  cout=1
# time=300   a=0  b=0  c=1  sum=1  cout=0
# time=350   a=1  b=0  c=1  sum=0  cout=1
# time=400   a=0  b=1  c=1  sum=0  cout=1
# time=450   a=1  b=1  c=1  sum=1  cout=1
```