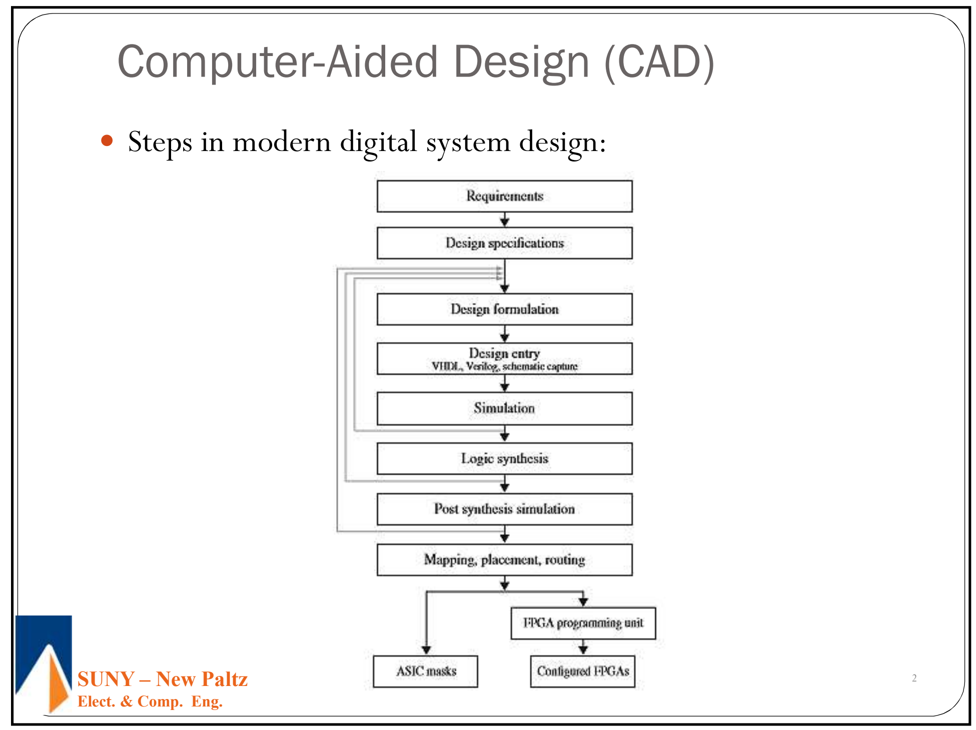
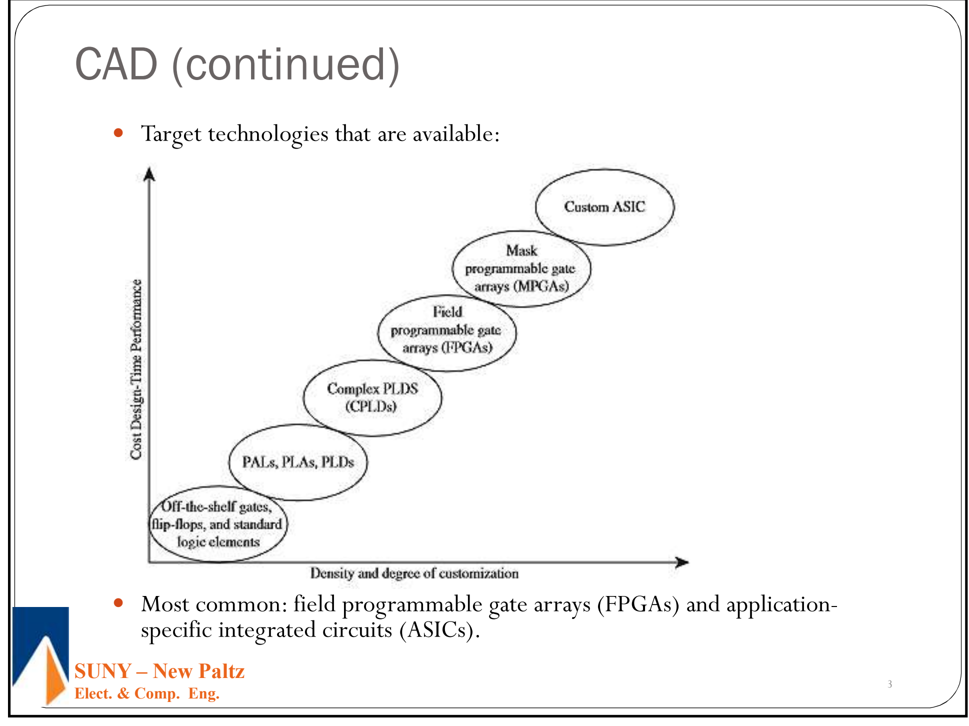
EGC 455
SOC Design &Verification
System Verilog
New Paltz
STATE UNIVERSITY OF NEW YORK
Baback Izadi
Division of Engineering Programs
bai@engr.newpaltz.edu

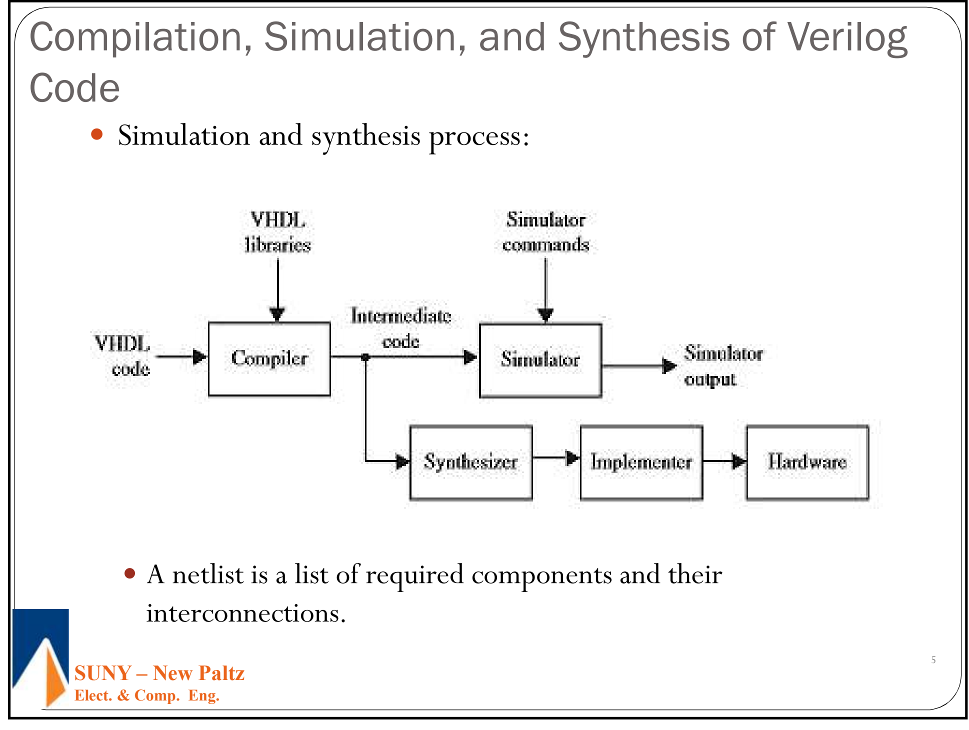

Computer-Aided Design (CAD)
Steps in modern digital system design:
Requirements
Design specifications
Design formulation
Design entry
VHDL, Verilog, schematic capture
Simulation
Logic synthesis
Post synthesis simulation
Mapping, placement, routing
FPGA programming unit
ASIC masks
Configured FPGAs
SUNY – New Paltz
Elect. & Comp. Eng.



Hardware Description Languages (HDLs)

- HDLs can describe a digital system at several different levels—behavioral, data flow, and structural.
- HDLs lead naturally to a top-down design methodology.
- Two popular HDLs—VHDL and Verilog.
- Verilog is a HDL used to describe the behavior and / or structure of digital systems.

4



History, Need, HDL vs. PL, Simulation Model, VLSI Digital Design Flow, SOC Example, References

- Session I

SUNY – New Paltz
Elect. & Comp. Eng.

6

History of Verilog Language

- Verilog HDL
  - derived from Latin word *veritas* (meaning truth), *and Greek word logos* (meaning reason, idea, word)
  - Use Verilog to express the *truth of your idea*
  - Verilog originated at Gateway Design Automation in 1983-1984 for simulation of digital circuits
  - Prabhu Goel, a founder member of Gateway Design Automation, along with Phill Moorby released Verilog HDL in 1983
  - Enhanced for ASIC design 1987–1989
  - Synopsys introduced Verilog logic synthesis 1987
  - Cadence acquired Gateway Design Automation in 1990 and puts Verilog HDL in the public domain
  - Open Verilog International (OVI) formed to maintain Verilog standards
- IEEE standardization – LRM – IEEE-1364 - 1995
  - Revision – IEEE-1364 – 2001 (many enhancements)

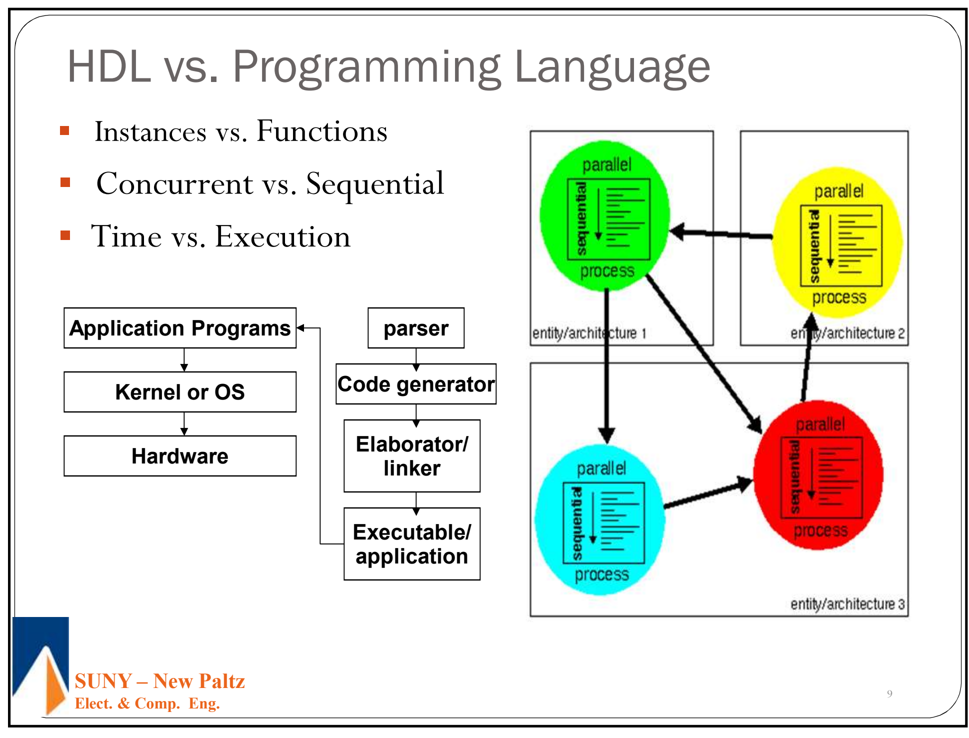
7

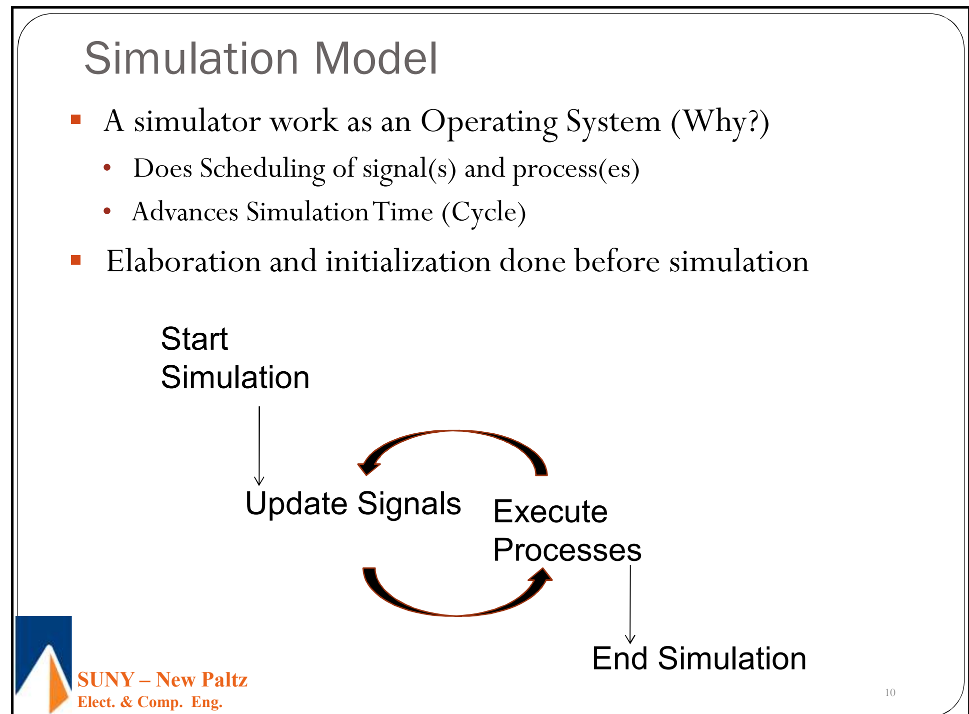
Need for Verilog Language

- Verilog HDL
  - Integral part of the Design Process for Digital Circuit Design
  - Universally available as a Public Document
  - Provide constructs that are natural to a designer
  - Flexible and easily extendable by the user
  - Support multiple levels of abstraction and modeling
  - Allow different design approaches(methodologies)
    - top – down
    - bottom - up
  - Technology and Process Independence
  - Ease of Design Exchange and Re-use
  - Primarily aimed at Designing once and using a synthesis tool to target different implémentations technologies: ASIC, FPGAs, CPLDs

SUNY – New Paltz
Elect. & Comp. Eng.

8


HDL vs. Programming Language
Instances vs. Functions
Concurrent vs. Sequential
Time vs. Execution
Application Programs
Kernel or OS
Hardware
parser
Code generator
Elaborator/ linker
Executable/ application
parallel
sequential
process
entity/architecture 1
entity/architecture 2
entity/architecture 3
SUNY – New Paltz
Elect. & Comp. Eng.
9


Simulation Model
A simulator work as an Operating System (Why?)
Does Scheduling of signal(s) and process(es)
Advances Simulation Time (Cycle)
Elaboration and initialization done before simulation
Start Simulation
Update Signals
Execute Processes
End Simulation
SUNY – New Paltz
Elect. & Comp. Eng.
10

Basics of Verilog Language
Gate Level Modeling
Dataflow Modeling

- Session II

11

- How can we model a Digital Design
  - Truth Tables
  - Schematic - Most schematic capture tools can automatically generate a Verilog netlist for simulation
  - HDL's
    - VHDL
    - Verilog
  - Tools available for Verilog HDL Simulation and Synthesis

Logic Simulation
- Verifies correctness of design functionality
- Tools
  - ModelSim : MentorGraphic
  - NC Sim : Cadence
  - VCS : Synopsys

Logic Synthesis
- Creates a logic circuit from an HDL
- Tools
  - Leanardo Spectrum : MentorGraphic
  - RC Compiler : Cadence
  - Design Compiler : Synopsys

12

Modeling styles in Verilog

- Gate Level
  - Hardware implemented in terms of logic gates with interconnections between these gates
  - Design at this level is similar to describing the design in terms of gate level logical diagram (white box)
  - mainly used to model combinational logic design
- Dataflow
  - Hardware implemented by specifying the data flow
  - Designer is aware of how the data flows between registers (black box)
  - Also used to describes data flow between registers known as RTL (Register Transfer Level) – a type of dataflow modeling used for the purpose of synthesis

13

Modeling styles (level) in Verilog

- Behavioral
  - Used to model the behavior of a design without the concern for the hardware implementation details, i.e., at the abstract level or highest level of abstraction
  - Designing at this level is very similar to C programming
  - Mainly used to model sequential logic design
  - Also used to describe RTL (Register Transfer Level) – a type of behavioral modeling used for the purpose of synthesis
- Switch Level
  - Lowest level of abstraction provided by Verilog with switches/transistors having nodes and interconnection between them
  - experienced designers use this style

14

Gate Level Modeling

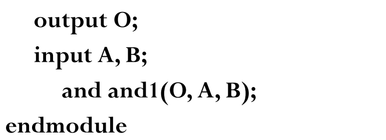
Verilog language provides basic gates as built-in primitives as shown, aka white box

Name	Description	Usage
and	$f = (a \cdot b \cdots)$	and(f, a, b, ...)
nand	$f = \overline{(a \cdot b \cdots)}$	nand(f, a, b, ...)
or	$f = (a + b + \cdots)$	or(f, a, b, ...)
nor	$f = \overline{(a + b + \cdots)}$	nor(f, a, b, ...)
xor	$f = (a \oplus b \oplus \cdots)$	xor(f, a, b, ...)
xnor	$f = (a \odot b \odot \cdots)$	xnor(f, a, b, ...)
not	$f = a$	not(f, a)
buf	$f = a$	buf(f, a)
notif0	$f = (!e ? \overline{a} : 'bz)$	notif0(f, a, e)
notif1	$f = (e ? \overline{a} : 'bz)$	notif1(f, a, e)
bufif0	$f = (!e ? a : 'bz)$	bufif0(f, a, e)
bufif1	$f = (e ? a : 'bz)$	bufif1(f, a, e)

Gate Level Modeling – single gate h/w design

Writing gate level hardware model for an and-gate

- module keyword implements a hardware
- unique name of the hardware (e.g. name of a human being)
- input, output ports or pins declarations
- by convention output ports are declared first
- body of module/hardware represents behavior
- concept of instantiation

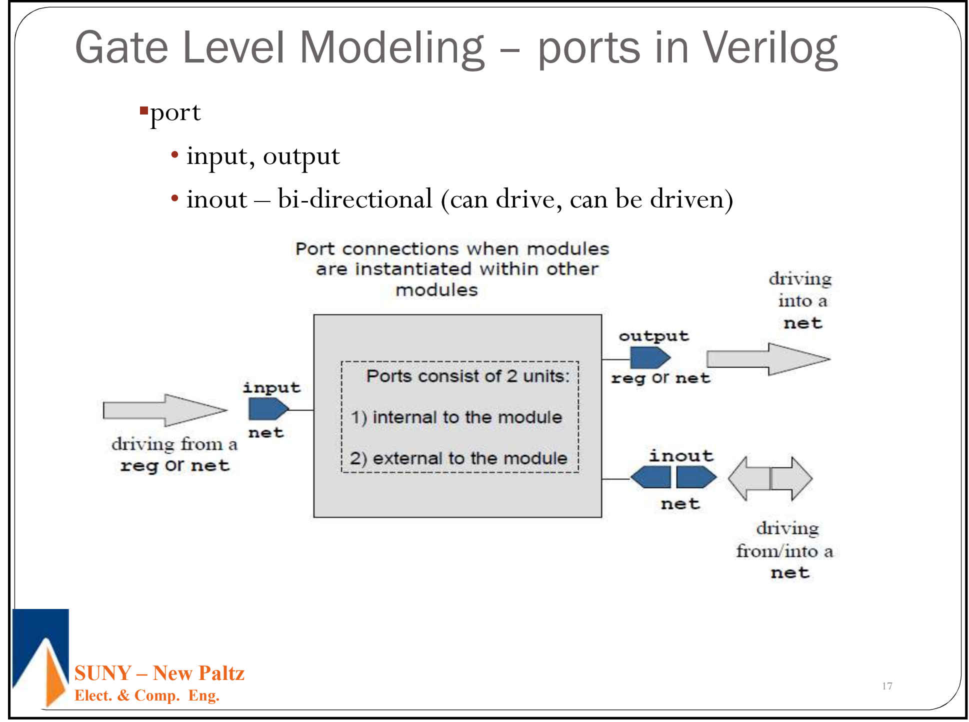


```
module and_gate_2_input(O, A, B);
   output O;
   input A, B;
      and and1(O, A, B);
endmodule
```

2-input AND gate

Input$_A$, Input$_B$ → Output

A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1



Gate Level Modeling – ports in Verilog

- provides an interface by which a module can communicate with its environment
- if output port has to hold a value then it must be declared as reg
- all port declarations are implicitly declared as wire
- input/output ports can be declared as reg
- when modules are instantiated within another
  - input must always be a wire (net), it can be externally connected to a reg or net
  - output can be net or reg, it must externally connect to a net
  - inout must be a net, it must externally connect to a net

18

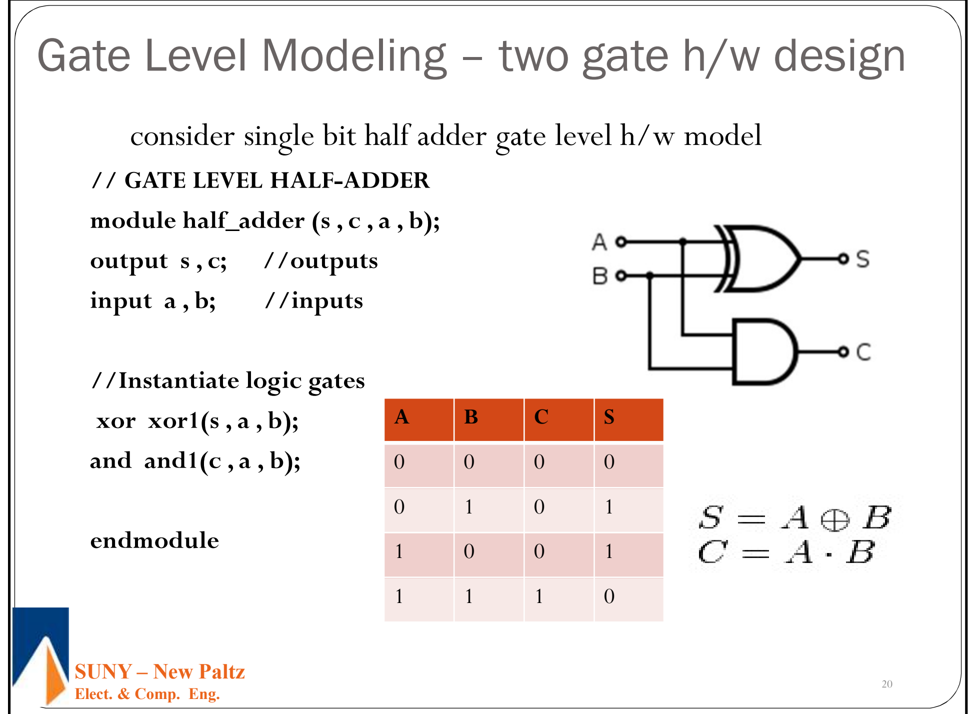
Gate Level Modeling – ports in Verilog

- port association – named vs. positional

```
module mux4_to_1(out, i0, i1, i2, i3, s1, s0); // design module
   output out;
   input i0, i1, i2, i3, s1, s0;
   ...........
endmodule
module stimulus; // stimulus module (no ports)
    reg IN0, IN1, IN2, IN3, S1, S0; // declare variables to connect
to inputs
    wire OUTPUT; // declare output wire
    // instantiate the mux - connected by position or order
     mux4_to_1 mymux1(OUTPUT, IN0, IN1, IN2, IN3, S1, S0);
    // instantiate the mux - connected by name or any order
    mux4_to_1 mymux2(.s0(S0), .s1(S1), .i3(IN3), .i2(IN2),
                          .i1(IN1), .i0(IN0), .out(OUTPUT));
    .........
endmodule
```

SUNY – New Paltz
Elect. & Comp. Eng.

19



A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Gate Level Modeling – concept of a wire

- multiple gate design
- consider single bit full adder gate level h/w model

```
// GATE LEVEL FULL-ADDER
module full_adder (Cout , S , A , B, Cin);
    output  Cout , S;  //outputs
    input  A, B, Cin;   //inputs
    wire w1, w2, w3; // wires
       xor  xor1(w1 , A, B);
       xor  xor2(S, w1, Cin);
       and  and1(w2 , w1, Cin);
       and  and2(w3 , A, B);
       or     or1(Cout, w2, w3);
endmodule
```

SUNY – New Paltz
Elect. & Comp. Eng.

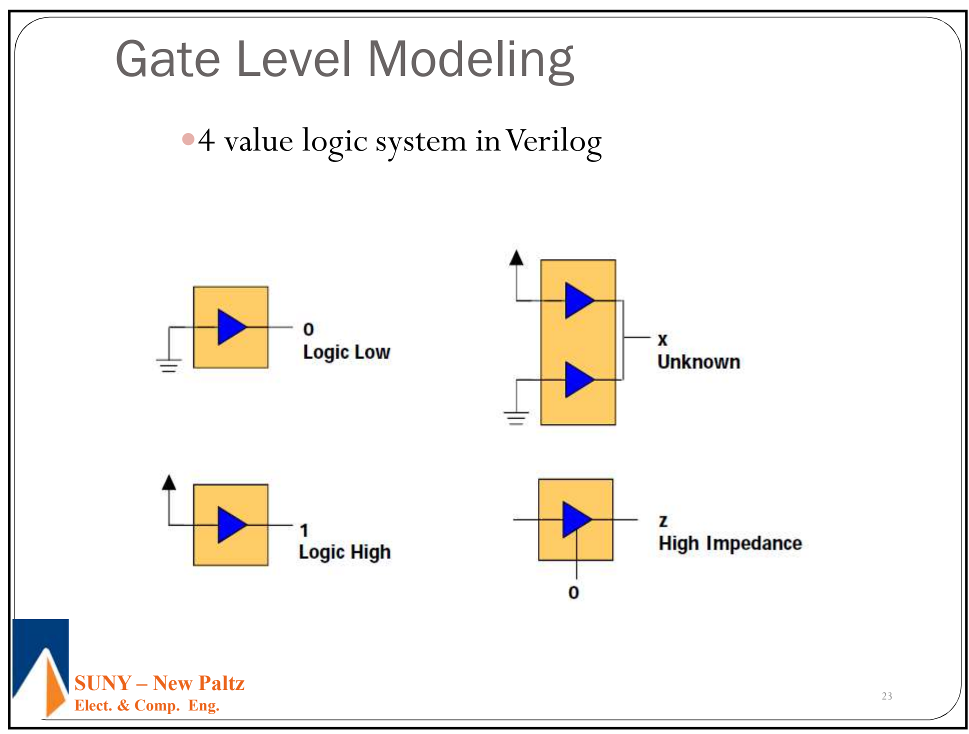
21

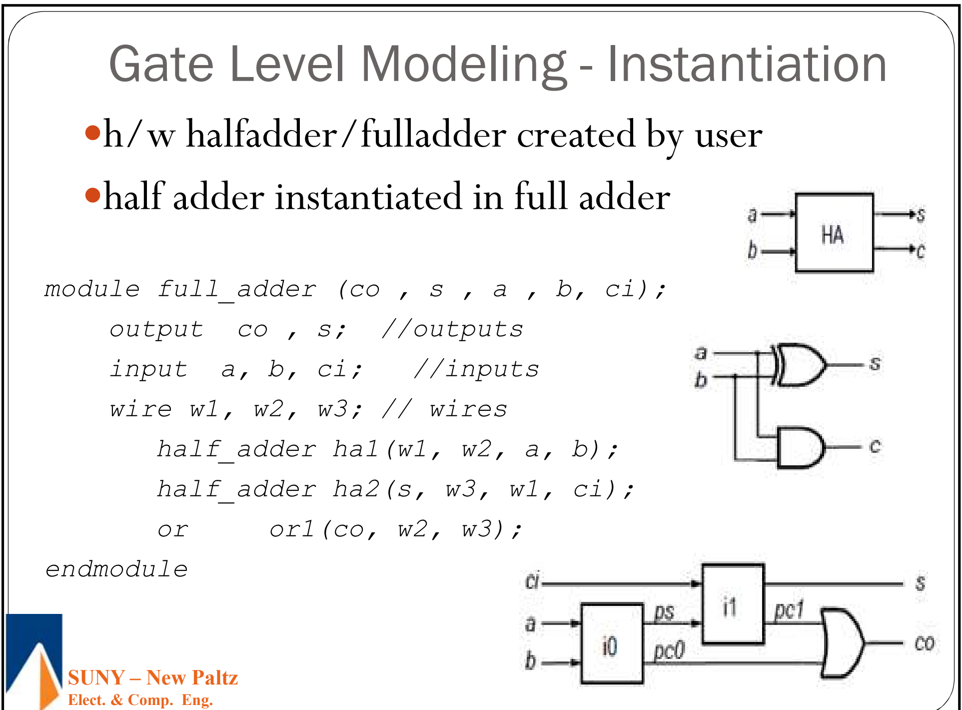
Gate Level Modeling

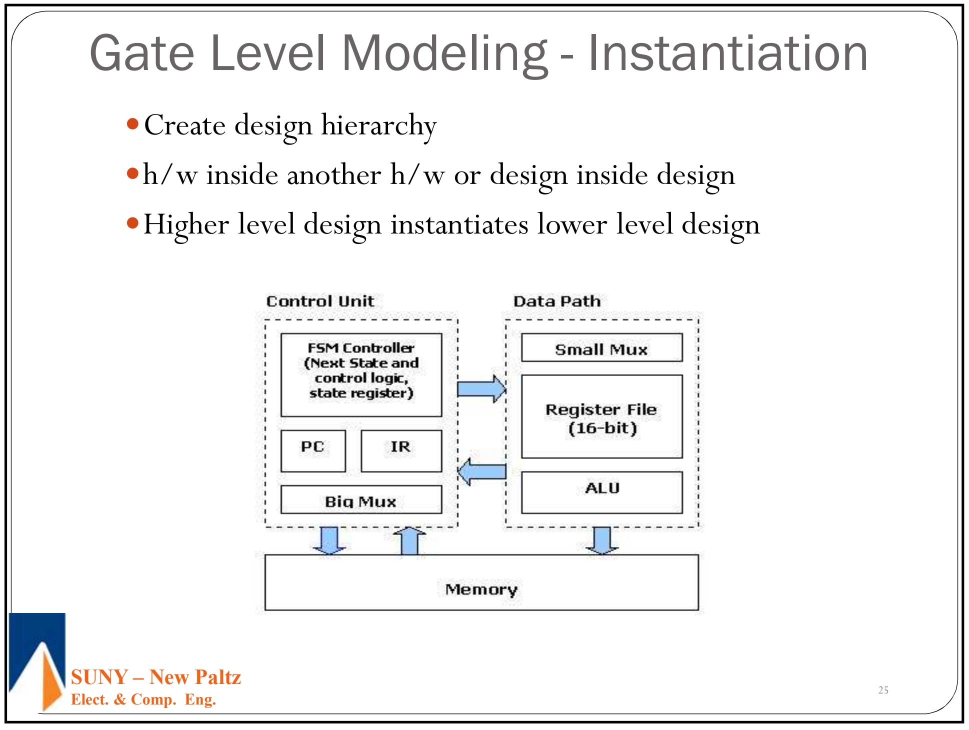
About wire

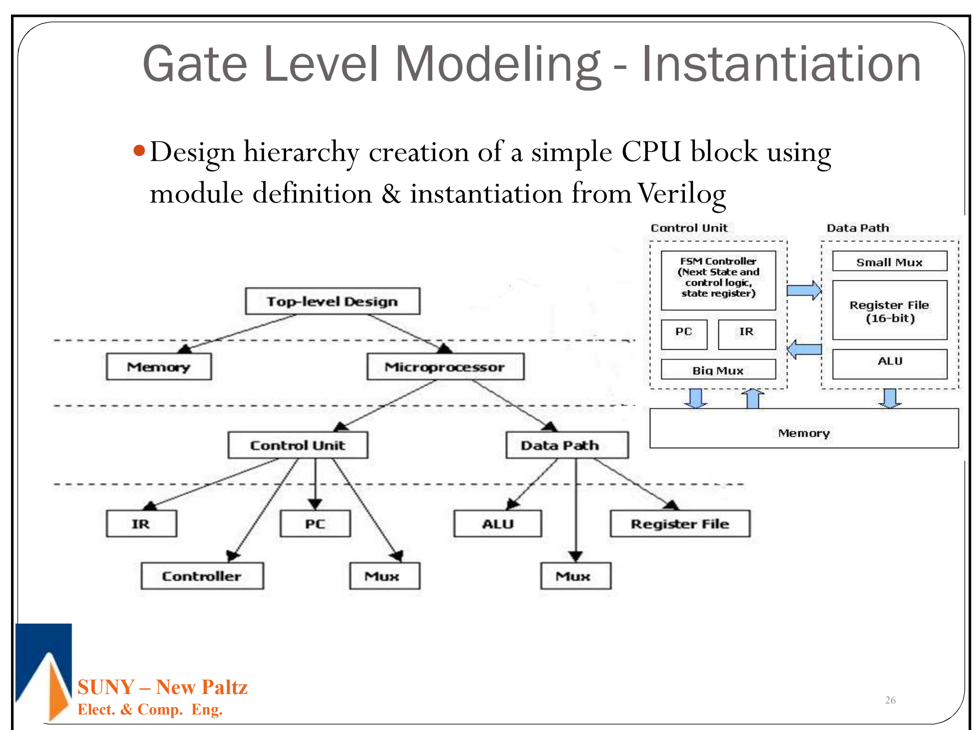
- Electrically connect things together i.e. act as an interconnection mechanism between different hardware elements
- Can be used on the right-hand side of an expression i.e. it is said to be driven
- Can have multiple drivers "Fan-Out"
- Multiple drivers are resolved using resolution table
- Used to tie gates and behavioral blocks together defaults to a logic value 'z' – high impedance state if no driver or if not connected

22


Gate Level Modeling
•4 value logic system in Verilog
0
Logic Low
x
Unknown
1
Logic High
z
High Impedance
0
SUNY – New Paltz
Elect. & Comp. Eng.
23


Gate Level Modeling - Instantiation
•h/w halfadder/fulladder created by user
•half adder instantiated in full adder
module full_adder (co , s , a , b, ci);
output co , s; //outputs
input a, b, ci; //inputs
wire w1, w2, w3; // wires
half_adder ha1(w1, w2, a, b);
half_adder ha2(s, w3, w1, ci);
or or1(co, w2, w3);
endmodule
HA
a
b
s
c
ci
i0
i1
ps
pc0
pc1
co
SUNY – New Paltz
Elect. & Comp. Eng.


Gate Level Modeling - Instantiation
•Create design hierarchy
•h/w inside another h/w or design inside design
•Higher level design instantiates lower level design
Control Unit
Data Path
FSM Controller
(Next State and
control logic,
state register)
PC
IR
Big Mux
Small Mux
Register File
(16-bit)
ALU
Memory
SUNY – New Paltz
Elect. & Comp. Eng.
25


Gate Level Modeling - Instantiation
•Design hierarchy creation of a simple CPU block using
module definition & instantiation from Verilog
Top-level Design
Memory
Microprocessor
Control Unit
Data Path
IR
PC
ALU
Register File
Controller
Mux
Mux
FSM Controller
(Next State and
control logic,
state register)
Big Mux
Small Mux
Register File
(16-bit)
SUNY – New Paltz
Elect. & Comp. Eng.
26

Gate Level Modeling – writing a test bench

- No input/output port declarations
- h/w to be tested is instantiated
- Input declared as reg, output declared as wire
- Stimulus is written i.e. input values are changed
- Output values are monitored whenever input values change, **monitor matches printf in "C" and it's called a system task represented with the first character as $**
- Initial block is written to change initial values at different simulation times and to monitor them
- Regular or absolute delays are used with # as first character
- reg assignment and initialization using bit format creation
- Bit format is <length>'<format><value> e.g. 2'b11

27

Gate Level Modeling – test bench components

reg (registers – storage element)

- Define storage, but not necessarily a data latch
- Can only be changed by assigning value to them
- defaults to logic value 'x' – unknown

System task

- $monitor("time=%d, s=%b, c=%b, a=%b, b=%b", $time, s, c, a, b);
- %d means decimal format, %b means binary format
- $monitor is a system tasks for monitoring value changes in s, c, a, b, $time represents the current simulation time
- try the testbench without the $ sign and see what happens
  - monitor will be treated as any other name
  - it will be an elaboration error

28

Gate Level Modeling – test bench example

```
module tb_half_adder;
    wire  s, c;  //outputs
    reg  a, b;   //inputs
      half_adder ha1(s, c, a, b);
    initial
    begin
      $monitor("time = %d, s = %b, c = %b, a = %b, b =
  %b", $time, s, c, a, b);
    end
    initial begin
      #10 a = 1'b0; b = 1'b0;
      #10 a = 1'b0; b = 1'b1;
      #10 a = 1'b1; b = 1'b0;
      #10 a = 1'b1; b = 1'b1;
    end
endmodule
```

A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

SUNY – New Paltz
Elect. & Comp. Eng.

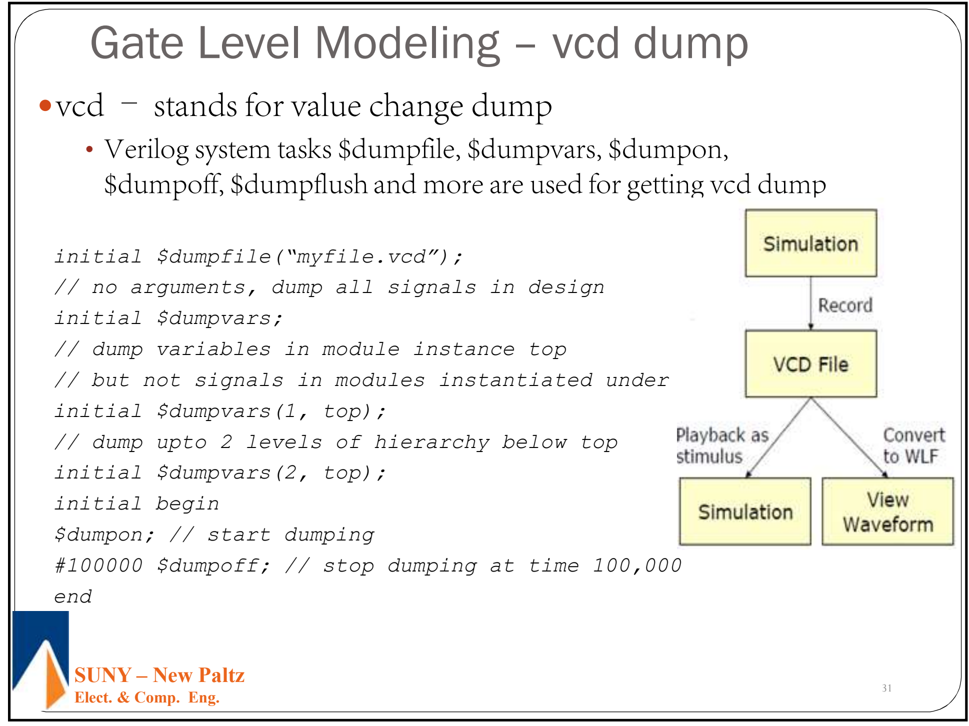
29

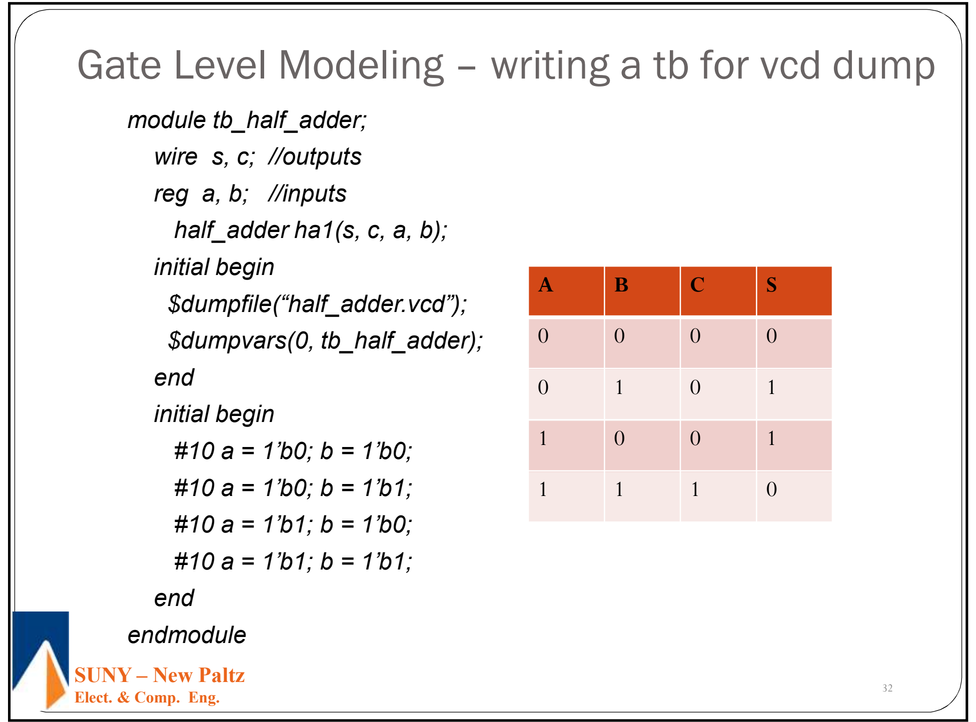
Gate Level Modeling – test bench components

- initial block
  - Used for initializing values
  - Initial block is executed once and only once at the start of simulation i.e. at 0 time
  - Different initial block executes concurrently
  - keywords `begin` and `end` are used to enclose multiple statements
- Regular delay, format, assignment
  - #10 a = 1'b0; b = 1'b0;
  - #10 represents regular delay (in the unit of simulation time)
  - 1'b0 bit format, b means binary and 1 means 1 bit
  - b = 1'b0 means assignment where = is assignment operator

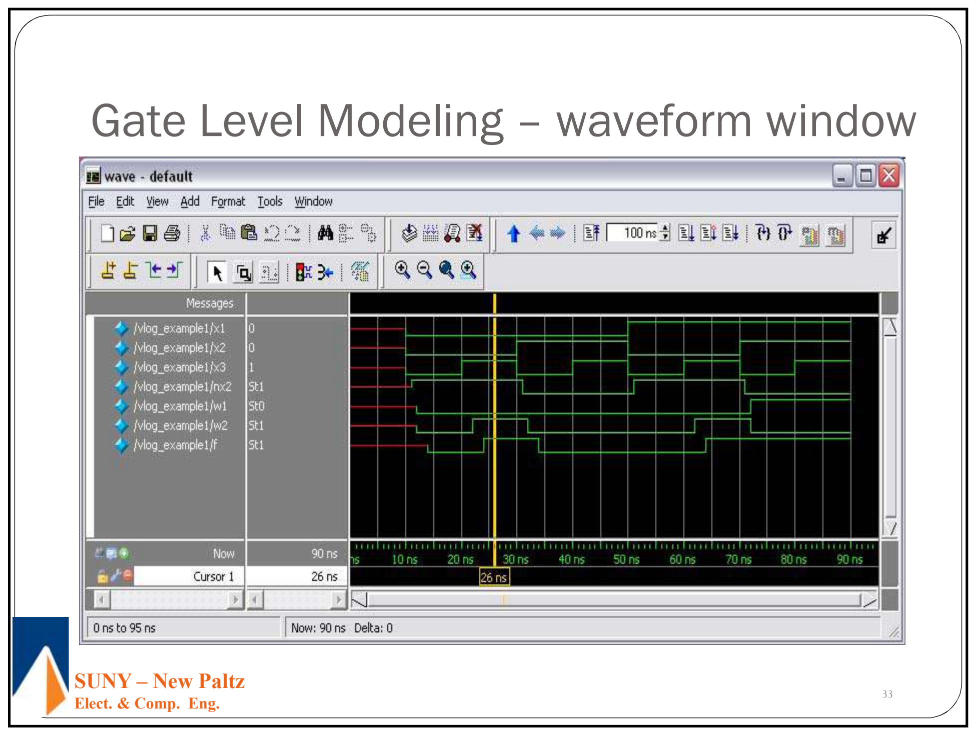
SUNY – New Paltz
Elect. & Comp. Eng.

30


Gate Level Modeling – vcd dump
vcd – stands for value change dump
Verilog system tasks $dumpfile, $dumpvars, $dumpon, $dumpoff, $dumpflush and more are used for getting vcd dump
initial $dumpfile("myfile.vcd");
// no arguments, dump all signals in design
initial $dumpvars;
// dump variables in module instance top
// but not signals in modules instantiated under
initial $dumpvars(1, top);
// dump upto 2 levels of hierarchy below top
initial $dumpvars(2, top);
initial begin
$dumpon; // start dumping
#100000 $dumpoff; // stop dumping at time 100,000
end
Simulation
Record
VCD File
Playback as stimulus
Convert to WLF
Simulation
View Waveform
SUNY – New Paltz
Elect. & Comp. Eng.
31


Gate Level Modeling – writing a tb for vcd dump
module tb_half_adder;
wire s, c; //outputs
reg a, b; //inputs
half_adder ha1(s, c, a, b);
initial begin
$dumpfile("half_adder.vcd");
$dumpvars(0, tb_half_adder);
end
initial begin
#10 a = 1'b0; b = 1'b0;
#10 a = 1'b0; b = 1'b1;
#10 a = 1'b1; b = 1'b0;
#10 a = 1'b1; b = 1'b1;
end
endmodule
A B C S
0 0 0 0
0 1 0 1
1 0 0 1
1 1 1 0
SUNY – New Paltz
Elect. & Comp. Eng.
32

Gate Level Modeling – waveform window



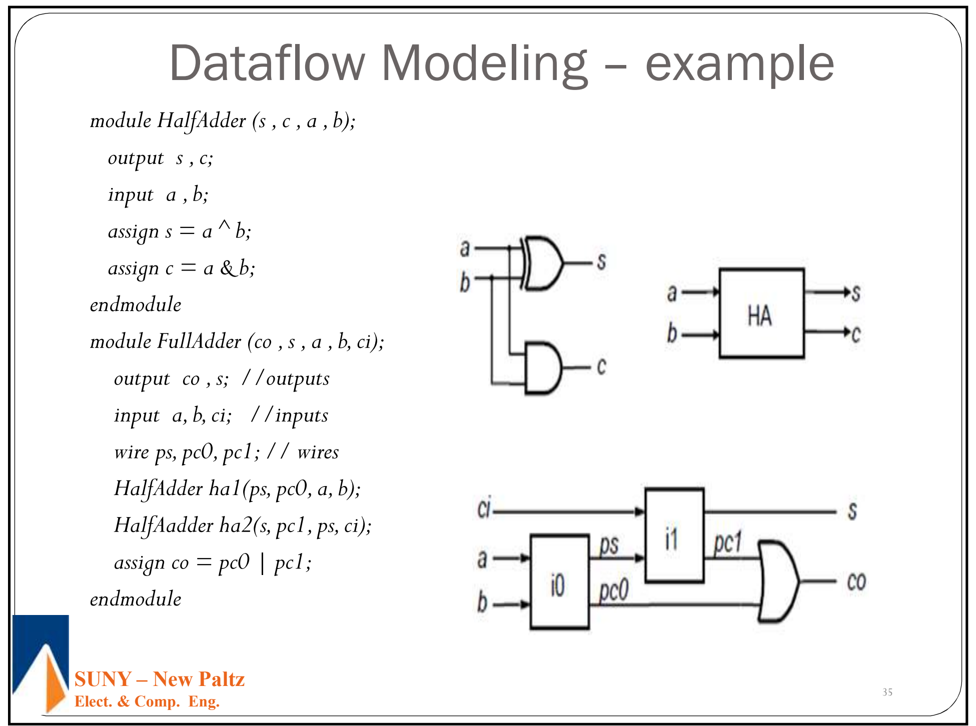
SUNY – New Paltz
Elect. & Comp. Eng.
33

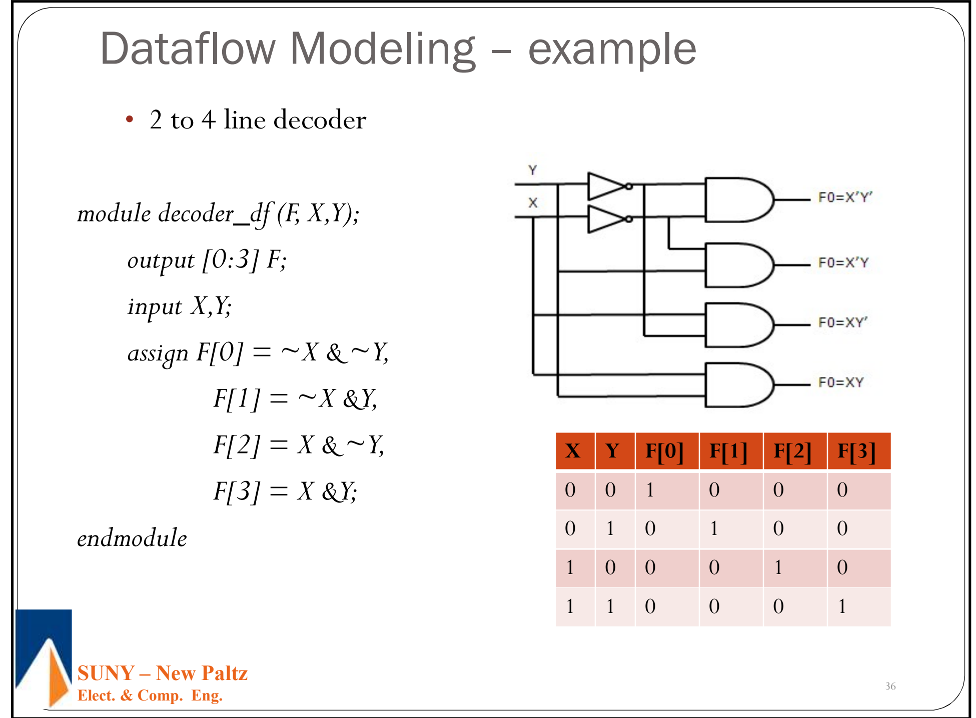
Dataflow Modeling

- black box where bitwise operators are used for data flow between registers
  - keyword **assign** is used for dataflow modeling
  - usage of assignment operator = results in continuous assignment
  - bitwise operators

Operator	Operation	Examples a = 3'b101, b = 3'b110, c = 3'b01x
~	invert each bit	~a is 3'b010
&	and each bit	a & b is 3'b100, b & c is 3'b010
|	or each bit	a | b is 3'b111
^	xor each bit	a ^ b is 3'b011
~^ or ^~	xnor each bit	a ^~ b = 3'b100

SUNY – New Paltz
Elect. & Comp. Eng.
34





X	Y	F[0]	F[1]	F[2]	F[3]
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Basics of Verilog Language
Insight into the Language

- Session III

37

Insight into Verilog language – keywords - 1995

always	endmodule	large	reg	
and	endprimitive	macromodule	release	tranif0
assign	endspecify	nand	repeat	tranif1
attribute	endtable	negedge	rnmos	tri
begin	endtask	nmos	rpmos	tri0
buf	event	nor	rtran	tri1
bufif0	for	not	rtranif0	triand
bufif1	force	notif0	rtranif1	trior
case	forever	notif1	scalared	trireg
casex	fork	or	signed	unsigned
casez	function	output	small	vectored
cmos	highz0	parameter	specify	wait
deassign	highz1	pmos	specparam	wand
default	if	posedge	strength	weak0
defparam	ifnone	primitive	strong0	weak1
disable	initial	pull0	strong1	while
edge	inout	pull1	supply0	wire
else	input	pulldown	supply1	wor
end	integer	pullup	table	xnor
endattribute	join	rcmos	task	xor
endcase	medium	real	time	
endfunction	module	realtime	tran	

SUNY – New Paltz
Elect. & Comp. Eng.

38

Insight into Verilog language – keywords 2001

automatic	cell
config	design
endconfig	endgenerate
generate	genvar
incdir	include
instance	library
uwire	use

39

Insight into Verilog language – bnf, syntax, semantics – example module definition

```
module module name [(port name{, port name})];
        [parameter declaration]
        [input declaration]
        [output declaration]
        [inout declaration]
        [net declaration]
        [reg declaration]
        [time declaration]
        [integer declaration]
        [real declaration]
        [event declaration]
        [gate declaration]
        [UDP instantiation]
        [function or task]
        [continuous assign]
        [specify block]
        [initial statement]
        [always statement]
        [module instantiation]
endmodule
```

SUNY – New Paltz
Elect. & Comp. Eng.

40

Insight into Verilog language – bnf, syntax, semantics

- Insight into Verilog language – bnf, syntax, semantics
  - Backus Naur Form
    - defines the grammar for the full language
  - Syntax
    - defines the way you can write HDL as a text file
    - If not followed appropriately errors will get reported at compile time
  - Semantics
    - defines the meaning of language constructs
    - defines the multiple meaning of language constructs in different contexts
  - ? represents a 'z' value

41

Insight into Verilog language – etc.

- The basic lexical convention used by Verilog HDL are similar to those in C programming
- All keywords must be in LOWER case i.e. the language is case sensitive
- White spaces makes code more readable but are ignored by compiler
  - Blank space(\b) , tabs(\t) , newline(\n) are ignored by the compiler
- White spaces are not ignored by the compiler in strings
- Comments
  - // single line comment style
  - /* multi line
    comment style */
  - Nesting of comments not allowed

SUNY – New Paltz
Elect. & Comp. Eng.

42

Insight into Verilog language – identifiers

- Naming convention
  - objects are given different names, names are identifiers
  - Identifiers can be build using a combination of letters [A-Z], [a-z], digits [0-9] (can't be used as the first character in the identifier), underscore _, $ character (can't be used as the first character in the identifier), max 1024 characters allowed in an identifier
  - white spaces not allowed
  - statements are terminated by ;
  - e.g. myid, m_y_id, _myid, myid3 are valid
  - $myid, 3my_id are invalid
- Escaped identifiers
  - They start with a \ and end with a white space
  - They can include printable ASCII characters
  - E.g. \ 546 , \ .*.& , \ {***} , \ a+b-c , \Gate#3

43

Insight into Verilog language – numbers

- Representation
  - Decimal d or D
  - Hexadecimal h or H
  - Octal o or O
  - Binary b or B
- Value format
  - <size>'<radix><format>
  - When size not specified default value is 32 bits e.g. 'bz, 'h9
  - When radix not specified default decimal is taken e.g. 3
  - 2'b10, 3'd6, 6'o57, 3'O4, 8'H2d, 32'haA19, 5'B110x0, 6'ozz, 12'hZXb
  - For negative number use – sign e.g. -6'd3, -3'b11
  - Underscore can be used to enhance readability e.g. 12'o07_24, 12'b000_111_010_100

SUNY – New Paltz
Elect. & Comp. Eng.

44

Insight into Verilog language – numbers

- Value format when used in string format for printing
  - %h or %H for hexadecimal
  - %d or %D for decimal
  - %o or %O for octal
  - %c or %C for an ASCII character
  - %m or %M for hierarchical name
  - %v or %V for net signal value and strength
  - %s or %S for string
  - %t or %T for time
  - %f or %F for real value in decimal format (floating point)
  - %g or %G real value in either exponential or decimal form whichever is short
  - %e or %E for real value in exponent form

SUNY – New Paltz
Elect. & Comp. Eng.

45

Insight into Verilog language – strings, data types

- Strings
  - A string is a sequence of characters enclosed by double quotes
  - Must be contained on a single i.e. without carriage return
  - Strings are treated as a sequence of one-byte ASCII values
  - e.g. string_val = “Hello Verilog”;
- Data types
  - Net (physical connectivity) – default value z
    - wire, wand, wor
    - tri, triand, trior
    - tri0, tri1, trireg
    - supply0 (GND), supply1 VDD (VCC)
  - Registers (physical storage elements) – default value x
    - reg, integer, real
    - time, realtime (time shown in real number format)

46

Insight into Verilog language – data types - net

- "wor" performs "or" operation on multiple driver logic
- "wand" performs "and" operation on multiple driver logic
- "trior" and "triand" perform the same function as "wor" and "wand", but model outputs with resistive loads

net type	modeling usage
wire tri	Net with single driver Net with multiple driver
wand, triand wor, trior	Model wired logic function at gate level
tri0, tri1	Pulls up or down the net at gate level
trireg	Stores the value at previous level (gate)
supply0 supply1	Constant logic 0 at switch level Constant logic 1 at switch level

47

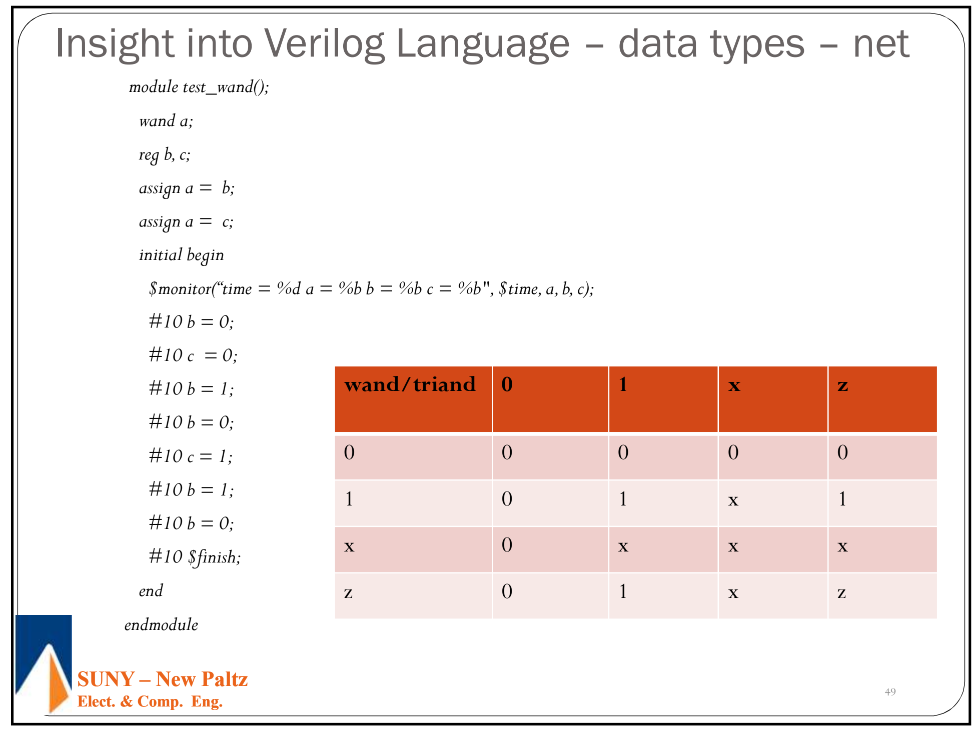
Insight into Verilog Language – data types – net

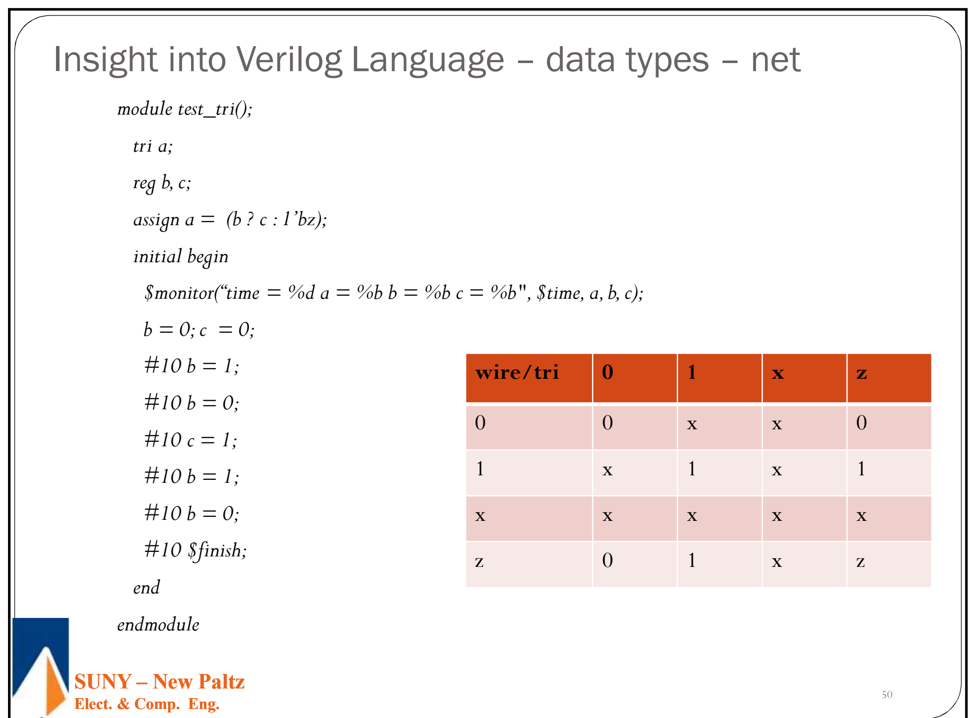
```
module test_wor();
 wor a;
 reg b, c;
 assign a = b;
 assign a = c;
 initial begin
  $monitor("time = %d a = %b b = %b c = %b", $time, a, b, c);
  #10 b = 0;
  #10 c = 0;
  #10 b = 1;
  #10 b = 0;
  #10 c = 1;
  #10 b = 1;
  #10 b = 0;
  #10 $finish;
 end
endmodule
```

wor/trior	0	1	x	z
0	0	1	x	0
1	1	1	1	1
x	x	1	x	x
z	0	1	x	z

SUNY – New Paltz
Elect. & Comp. Eng.

48


Insight into Verilog Language – data types – net
module test_wand();
wand a;
reg b, c;
assign a = b;
assign a = c;
initial begin
$monitor("time = %d a = %b b = %b c = %b", $time, a, b, c);
#10 b = 0;
#10 c = 0;
#10 b = 1;
#10 b = 0;
#10 c = 1;
#10 b = 1;
#10 b = 0;
#10 $finish;
end
endmodule
wand/triand | 0 | 1 | x | z
0 | 0 | 0 | 0 | 0
1 | 0 | 1 | x | 1
x | 0 | x | x | x
z | 0 | 1 | x | z
SUNY – New Paltz
Elect. & Comp. Eng.
49


Insight into Verilog Language – data types – net
module test_tri();
tri a;
reg b, c;
assign a = (b ? c : 1'bz);
initial begin
$monitor("time = %d a = %b b = %b c = %b", $time, a, b, c);
b = 0; c = 0;
#10 b = 1;
#10 b = 0;
#10 c = 1;
#10 b = 1;
#10 b = 0;
#10 $finish;
end
endmodule
wire/tri | 0 | 1 | x | z
0 | 0 | x | x | 0
1 | x | 1 | x | 1
x | x | x | x | x
z | 0 | 1 | x | z
SUNY – New Paltz
Elect. & Comp. Eng.
50

Insight into Verilog language – strength

- Data types – net data type strengths
  - Verilog allows signals to have logic values and strength values
  - Logic values are 0 , 1 , x , z
  - Strength values are used to resolve combinations of multiple signals and to represent behavior of actual hardware elements as accurately as possible
  - Driving strengths are used for signal values that are driven on a net
  - Storage strengths are used to model charge storage in trireg type nets
  - Strength values can be used to resolve signal contention on nets that have multiple drivers
  - There are many rules applicable to resolution of contention

Insight into Verilog language – strength

- Data types – net data type strengths
  - In case of signal contention , value is resolved using logic strength

Strength Level	Strength Name	Abbreviation	Strength Level	Strength Type
7	supply1	su1	strongest1	driving
6	strong1	st1	strongest1	driving
5	pull1	pu1	strongest1	driving
4	large1	la1	strongest1	storage
3	weak1	we1	strongest1	driving
2	medium1	me1	strongest1	storage
1	small1	sm1	strongest1	storage
0	highz1	hiz1	weakest1	High impedance

SUNY – New Paltz
Elect. & Comp. Eng.

Insight into Verilog language – strength

- Data types – net data type strengths
  - In case of signal contention , value is resolved using logic strength

Strength Level	Strength Name	Abbreviation	Strength Level	Strength Type
7	supply0	su0	strongest0	driving
6	strong0	st0	strongest0	driving
5	pull0	pu0	strongest0	driving
4	large0	la0	strongest0	storage
3	weak0	we0	strongest0	driving
2	medium0	me0	strongest0	storage
1	small0	sm0	strongest0	storage
0	highz0	hiz0	weakest0	High impedance

SUNY – New Paltz
Elect. & Comp. Eng.

53

Insight into Verilog language – strength

- Data types – net data type strengths
  - Multiple signals with same values and different strength
    - If two signals with same known value and different strength drive the same net, the signal with higher strength wins
    - If two signals with unequal strengths drive the net then signal with stronger strength prevails

Su1
Pu1
Su1

    - When two signals with opposite value and same strength combine, the resulting value is "unknown – x"

Pu1
Pu0
Pu X

SUNY – New Paltz
Elect. & Comp. Eng.

54

Insight into Verilog language – vectors & arrays

- Vectors
  - wire [lsb : msb] wire1, wire2,…..;
  - reg [lsb:msb] reg1, reg2, ………;
  - wire [15:0] clear ; // 16 bit bus or wire
  - reg [3:0] cla ; // 4 bit register , can be assigned cla = 0;
- Arrays – memories – allowed only for reg, integer, time
  - reg [lsb:msb] mem [upper: lower]
  - reg [3:0] mem [31:0] ; // memory consisting of 32, 4 bit registers
  - reg mema [7:0] ; // array of 8 – 1 bit registers
  - Can't be assigned mem = 0 ; // illegal must use index
- Arrays – multi dimensional arrays (mda's)
  - wire [3:0] mda_wire [7:0][15:0];
  - reg [7:0] mda_rega [7:0][7:0][7:0];

55

Insight into Verilog language – operators

- Bitwise
  - Operates on each bit of operand
  - Result is in the size of the largest operand
  - Left extended if the sizes are different

Operator	Operation	Examples a = 3'b101, b = 3'b110, c = 3'b01x
~	invert each bit	~a is 3'b010
&	and each bit	a & b is 3'b100, b & c is 3'b010
|	or each bit	a | b is 3'b111
^	xor each bit	a ^ b is 3'b011
~^ or ^~	xnor each bit	a ^~ b = 3'b100

56

Insight into Verilog language – operators

- Arithmetic
  - If any operand contains z or x the result is unknown
  - If result and operand are of same size then carry is lost
  - Treats vectors as a whole value

Operator	Operation	Examples a = 5, b = 10, c = 2'b01
+	add	b + c = 11, +b = 10 (unary form)
-	subtract	b - c = 9, -b = -10 (unary form)
*	multiply	a * b = 50
/	divide	b / a = 2
%	modulo	b % a = 0
**	power	b ** a = 10000, a ** c = 5

SUNY – New Paltz
Elect. & Comp. Eng.

57

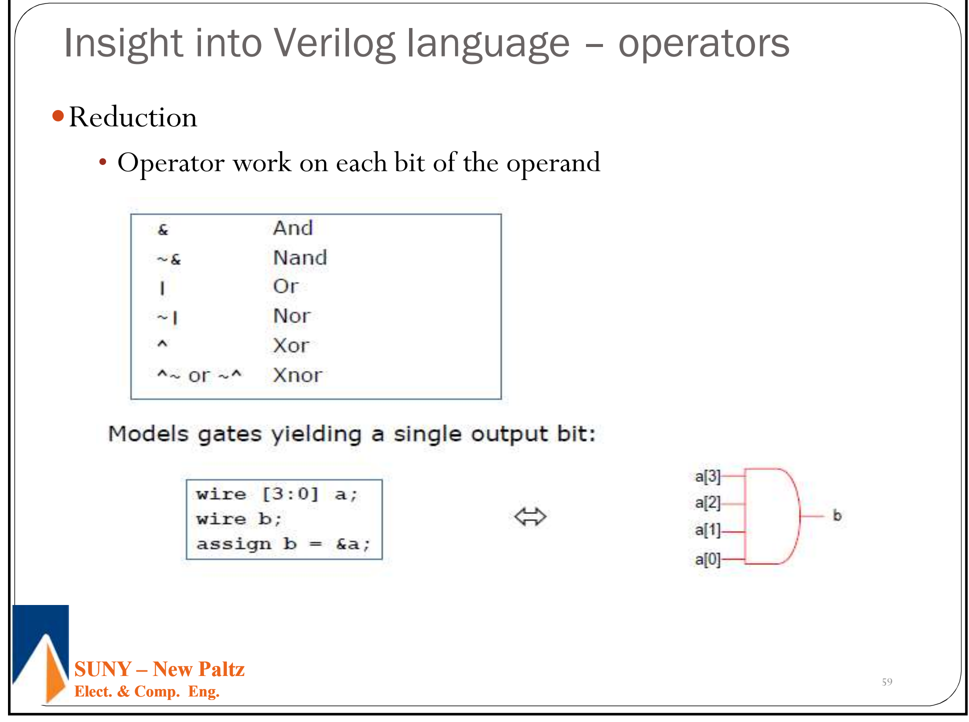
Insight into Verilog language – operators

- Logical
  - Can evaluate to 1, 0, x values
  - The results is either true (1) or false (0)

Operator	Operation	Examples a = 4, b = 0, c = 2'b0x
&&	logical and	a && b = 0, a && c = x, b && c = 0
||	logical or	a || b = 1, a || c = 1, b || c = x
!	arithmetic not	!a = 0, !b = 1, !c = x

SUNY – New Paltz
Elect. & Comp. Eng.

58



Insight into Verilog language – operators

- Shift
  - Shifts the bit of a vector left or right
  - Shifted bits are lost
  - Arithmetic shift right fills the shifted bits with sign bit
  - All others fill the shifted bits by zero

Operator	Operation	Examples ain = 4'b1010, bin = 4'b10X0
>>	logical shift right	bin >> 1 = 4'b010X
<<	logical shift left	ain << 2 = 4'b1000
>>>	arithmetic shift right	ain >>> 2 = 4'b1110
<<<	arithmetic shift left	ain <<< 2 = 4'b1000

SUNY – New Paltz
Elect. & Comp. Eng.

60

Insight into Verilog language – operators

- Relational
  - Evaluates to 1, 0, x
  - Result in x if any operand bit is z or x

Operator	Operation	Examples `a = 4, b = 0, c = 2'b0x`
>	greater than	`(a > b) = 1, (b > a) = 0`
>=	greater than or equal to	`(a >= 4) = 1, (b >= c) = x`
<	less than	`(a > b) = 0, (b > a) = 1`
<=	less than or equal to	`(b <= a) = 1, (b <= 0) = 1`

SUNY – New Paltz
Elect. & Comp. Eng.

61

Insight into Verilog language – operators

- Equality
  - assign WriteMe = (wr == 1) &&
    ((a >= 16'h7000) && (a < 16'h8000));

Operator	Operation	Result
==	logical equality	1 if operands are equal, 0 if operands are not equal, x if x or z in either operand
!=	logical inequality	1 if operands are not equal, 0 if operands are equal, x if x or z in either operand
===	case equality	1 if operands are equal, *including x and z,* else 0
!==	case inequality	1 if operands are not equal, *including x and z,* else 0

SUNY – New Paltz
Elect. & Comp. Eng.

62

Insight into Verilog language – operators

- Concatenation {}
  - wire [3:0] apart;
  - wire abit;
  - wire [7:0] cbus;
  - assign cbus = {apart, 3’b000, abit};
- Replication (repeat by a number)
  - assign cbus = {4{abit}, 4’b0000};
- Conditional ? :
  - assign y = (sel ? a : b); // mux
  - assign y = (s1 ? (s0 ? i1 : i2) : (s0 ? i3 : i4)); // can be nested

63

Insight into Verilog language – operators

- Concatenation – 4 bit adder
- Conditional – 8 bit comparator (>=)

```
module binary_adder (S, Cout, A , B , Cin);
  output [3:0] S;
  output Cout;
  input [3:0] A, B;
  input Cin;
  assign {Cout , S} = A + B + Cin;
endmodule

module comparator(Comp, A, B);
  output Comp;
  input [7:0] A, B;
  assign Comp = ((A >= B) ? 1’b1 : 1’b0);
endmodule
```

SUNY – New Paltz
Elect. & Comp. Eng.

64

Insight into Verilog language – operators

- Operator precedence

+ , - , ! , ~ (unary) Highest Priority
*, / , %, **
+ , - (Binary)
<< , <<<, >>, >>>
< , > , <= , >=
== , !=, ===, !==
& , ~&
^ , ^~ or ~^
| , ~|
&&
| |
? : (ternary) Lowest Priority

Highest Priority

Lowest Priority

- Parenthesis can be used to override the precedence

65

Insight into Verilog language – bit-part select

- Bit or part select
  - reg [15:0] Areg; // 16 bit register
  - wire Abit; // a single bit wire
  - wire [3:0] Apart; // 4 bit wire
  - assign Abit = Areg[1]; // bit select – select one bit
  - assign Apart = Areg[7:4]; // select parts of Areg – 4 bits
- Width mismatch: What happens in an assignment when LHS is wider than RHS
  - If RHS is unsigned and left most bit is 1 or 0, then LHS is extended with 0
  - If left most bit of RHS is Z, then LHS is extended with Z
  - If left most bit of RHS is X, then LHS is extended with X
  - If RHS is signed, then LHS is sign extended

66

Insight into Verilog language – width mismatch

- Width mismatch examples
  - wire [15:0] y;
  - wire [7:0] a = 'hff; // assigns b'11111111
  - wire signed [7:0] b = -1 ; // assigns b'11111111
  - wire [7:0] c = 8'bx1010101;
  - wire [7:0] d = 8'bz;
  - assign y = a; // fills with 'b0000000011111111
  - assign y = b; // fills with 'b1111111111111111
  - assign y = c; // fills with 'bxxxxxxxxx1010101;
  - assign y = d; // fills with 'bzzzzzzzzzzzzzzzz;

67

Insight into Verilog language – all about delays

- Regular delays
  - Delay execution of assignment by specified delay
  - assign #d1 y = a & b;
- Intra - assignment delays
  - Evaluate expression now and delay assignment by specified delay
  - assign y = #d1 a & b;
- Mix of regular and intra-assignment delays
  - assign #d1 y = #d2 a & b;
- Gate & Net delays
  - input x1, x2; wire #5 y; and #3 and1(y, x1, x2);

68

Insight into Verilog language – delay example

- Insight into Verilog language – delay example

```
module delaytb;

reg a = 1, b = 0, c = 0, d = 0;

initial begin // stimulus block
        #15 a = 0;
        #20 $finish;
end

initial begin
        /* Regular delay control:
         evaluate expression at time = 10 */
        #10 c = a | b;

        /* Intra-assignment delay control:
         evaluate expression now then wait 10 time units
         to assign value to d */
        d = #10 a | b;
end

endmodule
```

SUNY – New Paltz
Elect. & Comp. Eng.

69

Insight into Verilog language – all about delays

- Gate or propagation delays
  - Rise delay – transition from 0, x, z to 1
  - Fall delay – transition from 1, x, z to 0
  - Turn off delay – transition from 0, 1, x to z
  - Minimum (min), typical (typ), maximum (max) gate delays

<component_name> #(Rise, Fall, Turnoff) <instance_name> (port_list);

```
and    #(2)       u1(co, a, b);         // Delay of 2 for all transitions
and    #(1, 3)    u2(co, a, b);         // Rise = 1, Fall = 3
bufif0 #(1, 2, 3) u3(out, in, enable); // Rise = 1, Fall = 2, Turn-off = 3
```

#(Min:Typ:Max, Min:Typ:Max, Min:Typ:Max)

```
and    #(1:2:3)                u1 (co, a, b);
and    #(1:2:3, 1:2:3)         u2 (co, a, b);
bufif0  #(2:3:4, 2:3:4, 3:4:5)    u3 (out, in, enable) ;
```

SUNY – New Paltz
Elect. & Comp. Eng.

70

Insight into Verilog language – Gate delay example

```
module fa (cout, s, A, B, Cin);
   input A, B, Cin;
   output cout, s;
   wire w1, w2, w3;

   xor #(5, 4) a1(w1, A, B);
   xor #(10) x1(s, w1, Cin);
   and #(5, 4) a2(w2, A, B);
   and #(5, 4) a3(w3, Cin, w1);
   or #(5:6:7) o1(cout, w2, w3);
endmodule
```

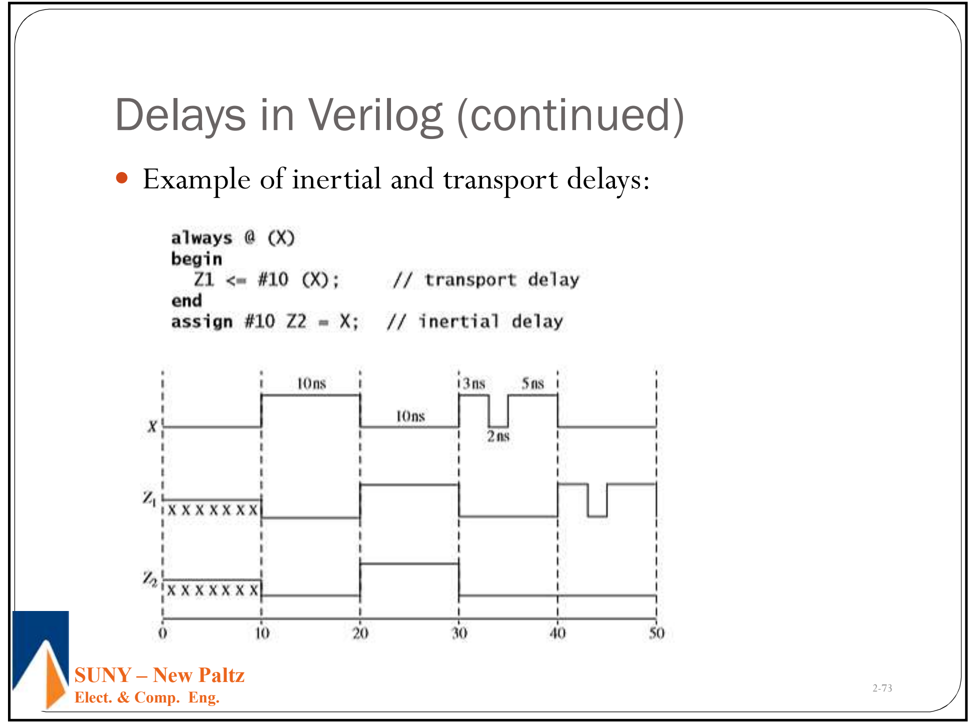
SUNY – New Paltz
Elect. & Comp. Eng.

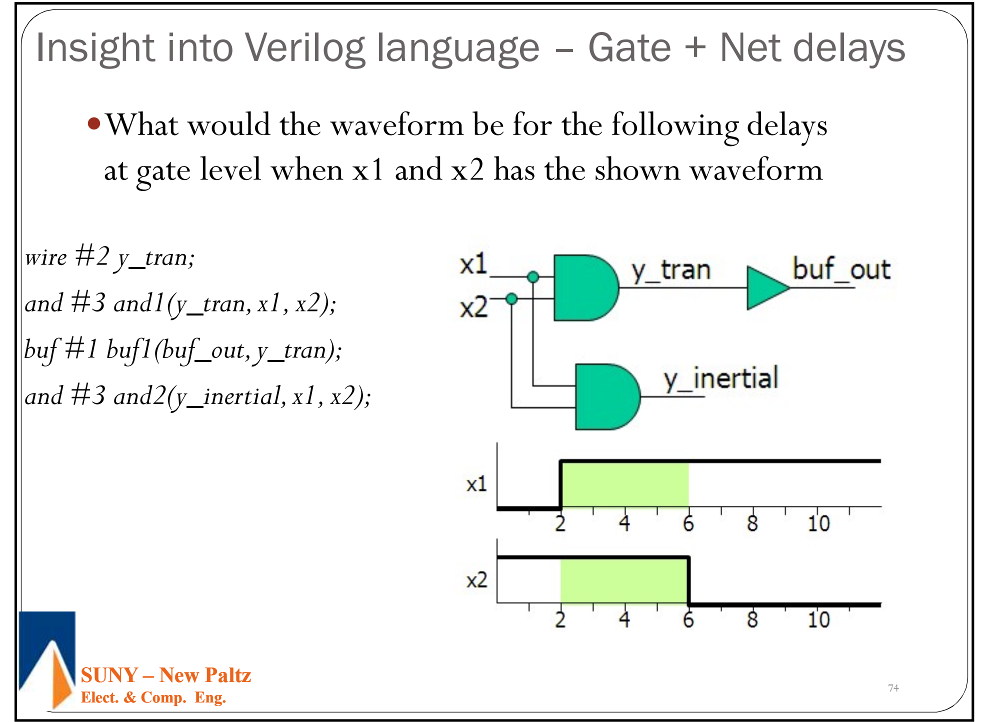
71

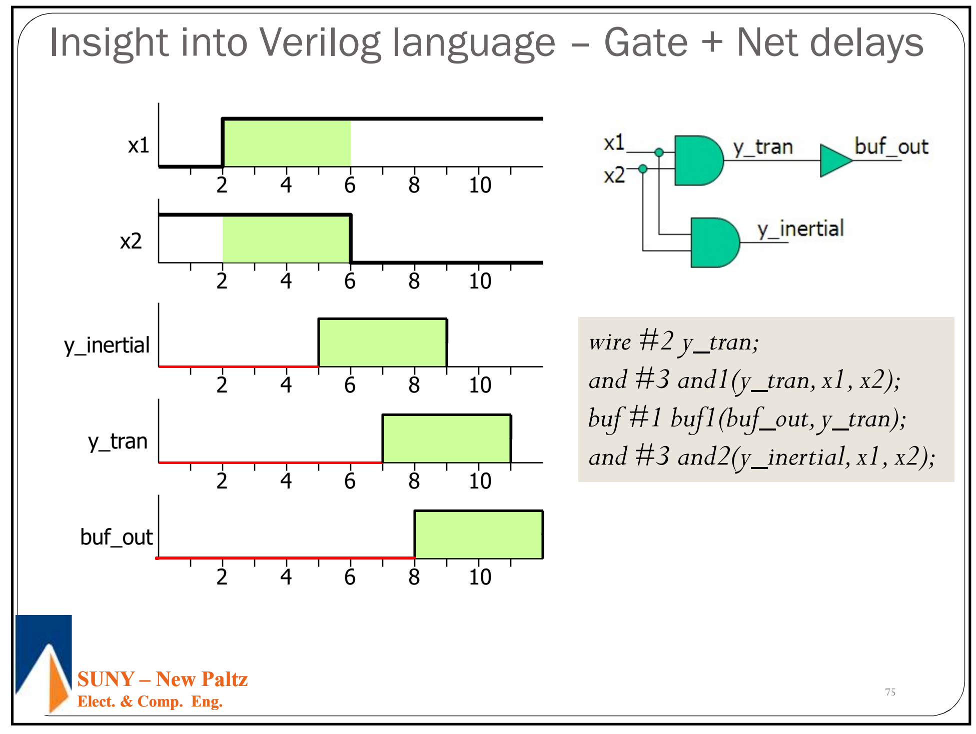
Delays in Verilog

- Types of Delays:
  - Inertial delay: intended to model gates and other devices that do not propagate short pulses from the input to the output.
  - Transport delay: is intended to model the delay introduced by wiring; it simply delays an input signal by the specified delay time.
  - Net delay: the time it takes from any driver on the net to change value to the time when the net value is updated and propagated further.

2-72


Delays in Verilog (continued)
Example of inertial and transport delays:
always @ (X)
begin
Z1 <= #10 (X); // transport delay
end
assign #10 Z2 = X; // inertial delay
10ns
10ns
3ns
5ns
2ns
X
Z1
Z2
0
10
20
30
40
50
SUNY – New Paltz
Elect. & Comp. Eng.

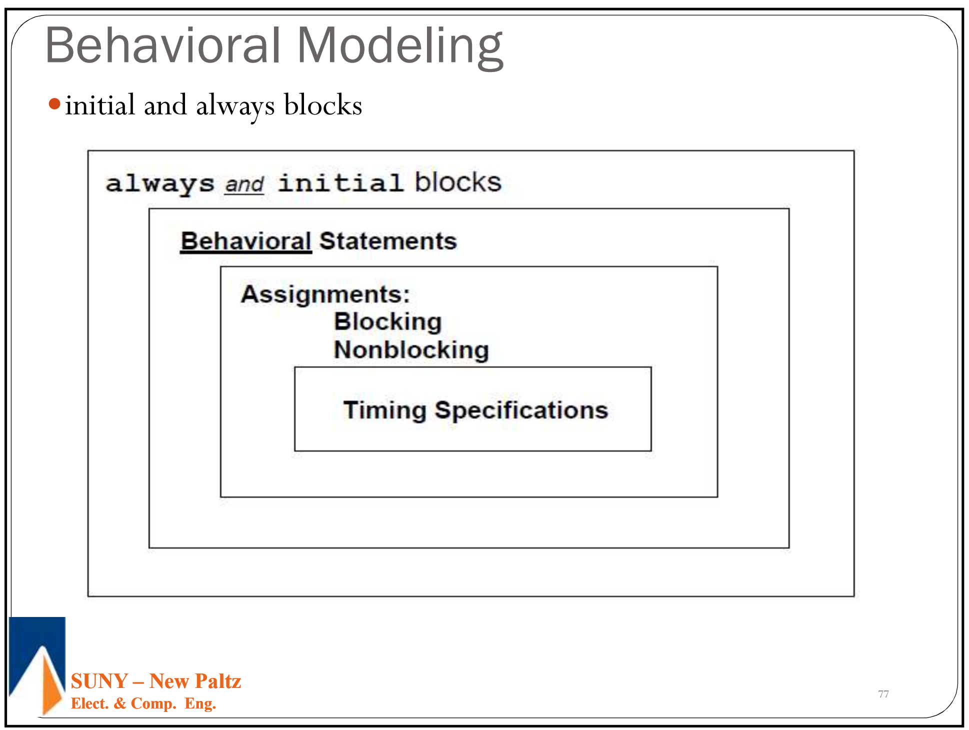

Insight into Verilog language – Gate + Net delays
What would the waveform be for the following delays at gate level when x1 and x2 has the shown waveform
wire #2 y_tran;
and #3 and1(y_tran, x1, x2);
buf #1 buf1(buf_out, y_tran);
and #3 and2(y_inertial, x1, x2);
x1
x2
y_tran
buf_out
y_inertial
2
4
6
8
10
SUNY – New Paltz
Elect. & Comp. Eng.

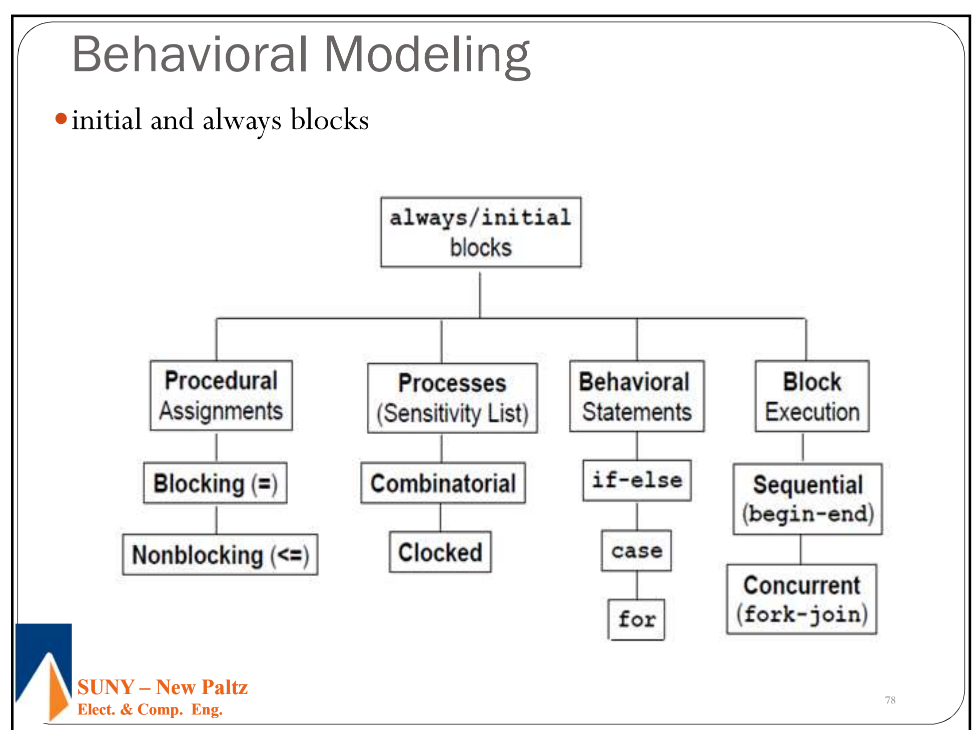


Behavioral Modeling

- initial and always blocks
- **initial block -** used to initialize behavioral statements
- **always block -** used to describe the functionality using behavioral statements
- assign values to register data types **(reg, integer, real, time)**
- each **always** and **initial** block represents a separate process
- processes run in parallel and start at simulation time 0
- statements inside a process execute either
  - sequentially **(begin - end)**
  - concurrently **(fork - join)**
- **always and initial blocks cannot be nested**

76


Behavioral Modeling
initial and always blocks
always and initial blocks
Behavioral Statements
Assignments:
Blocking
Nonblocking
Timing Specifications
SUNY – New Paltz
Elect. & Comp. Eng.
77


Behavioral Modeling
initial and always blocks
always/initial blocks
Procedural Assignments
Blocking (=)
Nonblocking (<=)
Processes (Sensitivity List)
Combinatorial
Clocked
Behavioral Statements
if-else
case
for
Block Execution
Sequential (begin-end)
Concurrent (fork-join)
SUNY – New Paltz
Elect. & Comp. Eng.
78

Behavioral Modeling

- initial block
  - initial blocks execute only once during simulation, starting at time 0, does't execute again
  - Multiple behavioral statement inside an initial block must be grouped using begin-end or fork-join
  - Multiple initial blocks execute concurrently
  - initial blocks are generally not synthesizable – mostly used in test-benches to apply stimulus sequences
  - Synthesizable only when used to initialize ROM/RAM contents
  - Assignment statements within an initial block
    - between begin and end execute sequentially – statement order matters
    - between fork and join execute concurrently – statement order doesn't matter

SUNY – New Paltz
Elect. & Comp. Eng.

79

Behavioral Modeling

- initial block example

```
module system;
reg a, b, c, d;

// single statement
initial
        a = 1'b0;

/* multiple statements
   need to be grouped */
initial
    begin
          b = 1'b1;
       #5  c = 1'b0;
       #10 d = 1'b0;
    end

initial
    #20  $finish;

endmodule
```

Time	Statement Executed
0	a = 1'b0; b = 1'b1;
5	c = 1'b0;
15	d = 1'b0;
20	$finish;

SUNY – New Paltz
Elect. & Comp. Eng.

80

Behavioral Modeling

- initial block – statements in procedural blocks can be grouped together to execute either sequentially or in parallel
- sequential execution
  - statements are enclosed within the keywords begin, end
  - statements are processed in the order they are specified
  - delays specified are additive

```
reg x, y;
 initial
 begin
   x = 1'b0;     //execute at t = 0
   y = 1'b1;
   #10 x = 1'b1;   //executes at t = 10
   #15 y = 1'b0;  //executes at t = 25
 end
```

SUNY – New Paltz
Elect. & Comp. Eng.

81

Behavioral Modeling

- initial block – parallel execution
  - statements are enclosed within the keyword fork, join
  - all statements are executed concurrently
  - ordering of statements is controlled by delay or event control assigned to each statement
  - if delay is specified, it is relative to the time the block started

```
reg x, y;
 initial
 fork
   x = 1'b0; y = 1'b0;
   #10 x = 1'b1;
   #20 y = 1'b0;
   #20 x = 1'b0;
   #40 y = 1'b1;
 join
```

SUNY – New Paltz
Elect. & Comp. Eng.

82

Behavioral Modeling

- always block
  - Used to model a process that is repeated continuously in a digital circuit
  - An **always** block starts at time 0 and executes the behavioral statements continuously in an event based looping fashion
  - Multiple behavioral statements inside an **always** block must be grouped using **begin-end** or **fork-join**
  - Assignment statements within an **always** block
    - between **begin** and **end** execute sequentially – statement order matters (synthesizable)
    - between **fork** and **join** execute concurrently – statement order doesn't matter (not synthesizable)

83

Behavioral Modeling

- always block example

```
module clock_gen (output reg clk);

parameter period=50, duty_cycle=50;

initial
   clk = 1'b0;

always
   #(duty_cycle*period/100)  clk = ~clk;

initial
   #100  $finish;

endmodule
```

Time	Statement Executed
0	clk = 1'b0
25	clk = ~clk // clk=1
50	clk = ~clk // clk=0
75	clk = ~clk // clk=1
100	$finish;

SUNY – New Paltz
Elect. & Comp. Eng.

84

Behavioral Modeling

- different methods of generating a clock

```
module clkgen(output reg clock);
initial begin
  #5 clock = 1;
  forever  #50 clock = ~clock;
end
endmodule
```

```
module clkgen_25_75_dc;
reg clk;
initial begin
   clk = 0;
end
always begin
   #25 clk = 0;
   #75 clk = 1;
end
endmodule
```

```
module clkgen_forever;
reg clk;
initial begin
   clk = 0;
end
always begin
   #10 clk = ~clk;
end
endmodule
```

SUNY – New Paltz
Elect. & Comp. Eng.

85

Behavioral Modeling

- Blocking and non blocking assignments in Verilog
- Blocking assignment "=“
  - Statement order matter
  - Completes the assignment in hand before moving on to the next statement meaning "Blocks" the other assignments until the current assignment completes
- Non blocking assignment "<=“
  - Concurrent assignment i.e. it does not "Block" execution of assignments in other statements
  - Evaluates the RHS at the beginning of a time step
  - Schedules the LHS update for the end of the time step
  - Results less dependent on order of assignments
  - If there are multiple non-blocking assignments to same variable in same behavior, latter overwrites previous
  - Not allowed in continuous assignments

SUNY – New Paltz
Elect. & Comp. Eng.

86

Behavioral Modeling

- Blocking and non blocking assignments in Verilog - example

```
initial begin
    a = 1;
    b = 0;
    a = b;   // a = 0;
    b = a;   // b = 0;
end
initial begin
    a = 1;
    b = 0;
    a <= b;   // a = 0;
    b <= a;   // b = 1;
end
```

87

Behavioral Modeling

- Execution control in procedural blocks
  - initial block executes once starting at time 0
  - always block executes continuously starting at time 0
- Within procedural blocks how does time advance?
  - Delays using # i.e. delay assignment by a specific amount of time (not synthesizable) using regular, intra-assignment, zero delays
  - Event control using @ i.e. delay assignment until specific event occurs (synthesizable)
    - edge triggering (sensitive)
      - 1⟶0 or 0⟶1 transition, or *edge on signal*
      - "posedge" 0⟶1 only
      - "negedge" 1⟶0 only
  - level triggering (sensitive)
    - use "wait" to delay execution until condition is true e.g. wait (f == 0);
    - Not synthesizable – used only in test-benches

88

Behavioral Modeling

- Execution control in procedural blocks
  - Event control using @ i.e. delay assignment until specific event occurs (synthesizable)
    - edge triggering (sensitive)
      - 1⟶0 or 0⟶1 transition, or *edge on signal*
      - "posedge" 0⟶1 only
      - "negedge" 1⟶0 only

```
@ (clk) q=d;   // statement executed whenever signal clk changes value
@ (posedge clk) q=d;  // executed whenever signal clk does a positive
                      //  transition (0 to 1, x or z, x to 1, z to 1)
q=@(posedge clk)d;  // d is evaluated immediately and assigned to q
                    // at positive edge of clk
```

SUNY – New Paltz
Elect. & Comp. Eng.

89

Behavioral Modeling

- Execution control in procedural blocks
- Event control using @ - combinational logic forms

```
always @(a) y = ~a;          // equivalent to assign y = ~a;

always @(a or b) y = a & b; // equivalent to assign y = a & b;

always @(a) y = a & b;       // what happens here?

// Verilog 2001 comma-separated event list
always @(s, a, b) y = s ? a : b;

// Verilog 2001 * operator makes block sensitive to all
// changes on any input - a, b, c, and d
always @(*) begin
     y1 = a + b;
     y2 = c - d;
end

module parity (input [31:0] in, output p);
   always @(in) p = ^in; // event list can include vectors
endmodule
```

SUNY – New Paltz
Elect. & Comp. Eng.

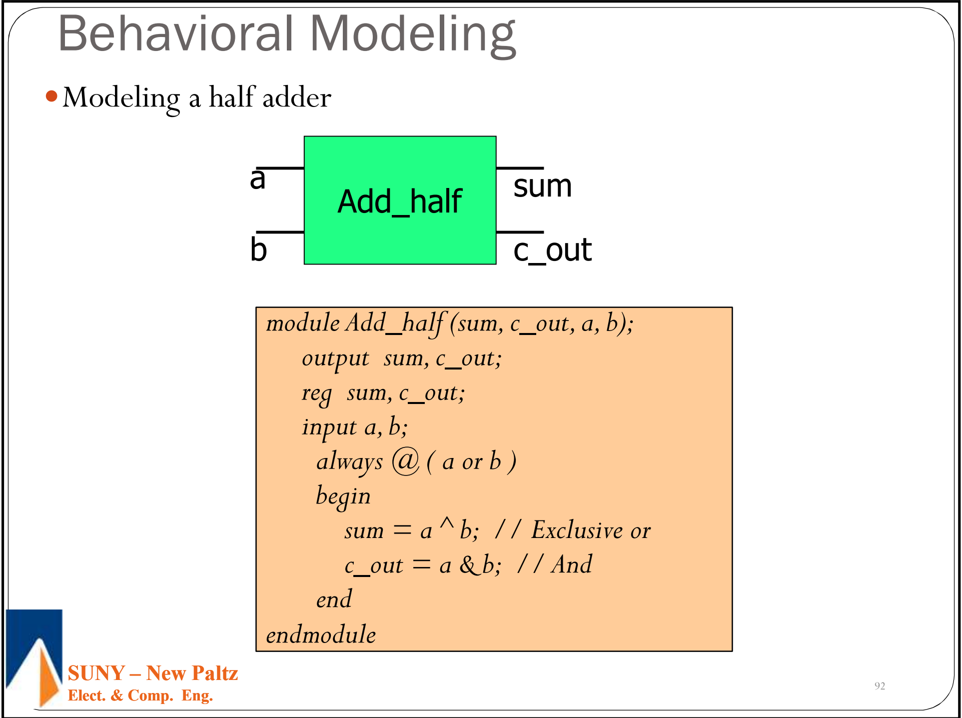
90

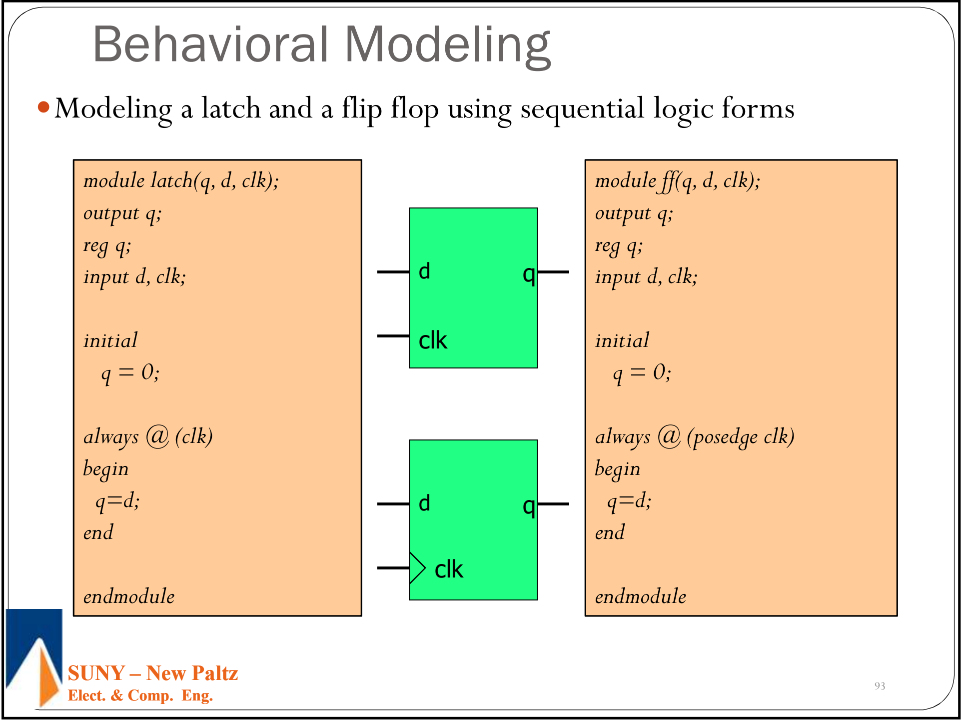
Behavioral Modeling

- Execution control in procedural blocks
- Event control using @ - sequential logic forms

```
// q = d executed whenever clock changes
always @(clock) q = d;

// q = d executed on rising edge (0->1) of clk
always @(posedge clk) q = d;

// q = d executed on falling edge (1->0) of clk
always @(negedge clk) q = d;

// flip-flop with asynchronous reset
always @(posedge clock, negedge reset_n)
   if (!reset_n) q = 0;
   else          q = d;
```

SUNY – New Paltz
Elect. & Comp. Eng.
91





Behavioral Modeling

- Execution control in procedural blocks - level triggering ("wait")
  - Provides level sensitive timing control
  - Activity flow is suspended if *expression* is false
  - It resumes when the *expression* is true
  - Other processes keep going

```
module modA (…);
…
    always begin
        …
        wait ( enable ) ra = rb;
        …
    end
endmodule
```

begin wait (!enable) #10 a = b; end
If the value of enable is 1 when block is entered, the wait statement will delay the evaluation of the statement (#10 a = b;) until the value of enable changes to 0. If enable is already 0 when the block is entered, then the assignment "a = b;" is evaluated after a delay of 10 and no additional delay occurs.

SUNY – New Paltz
Elect. & Comp. Eng.
94

Behavioral Modeling

- Blocking vs. non blocking assignments – Race condition
- When blocking assignments in two or more always blocks are scheduled to execute in the same time step , order of execution is indeterminated and it can result in a race condition

```
always @ (posedge clk)
    a = b;
always @ (posedge clk)
    b = a;
```

- Race condition (blocking statements)
- whether a = b or b = a ??
- **Recommended : use blocking assignments for modeling combinational logic in procedural blocks**
- Non blocking statements can be used to eliminate the race condition

SUNY – New Paltz
Elect. & Comp. Eng.

95

•Behavioral Modeling

- Blocking vs. non-blocking – Non-blocking statements can be used to eliminate the race condition
- At positive edge of clock, the values of all R.H.S variables are "read" and R.H.S expressions are evaluated and stored in temporary variables
- During "write" operation, the values stored in temporary variables are assigned to L.H.S variables
- Separating the read/write operations ensures that the values of registers a and b are swapped correctly

```
always @ (posedge clk)
    a <= b;
always @ (posedge clk)
    b <= a;
```

```
always @(posedge clk) begin
//read operation
temp_a = a;
temp_b = b;
//write operation
a = temp_b;
b = temp_a;
end
```

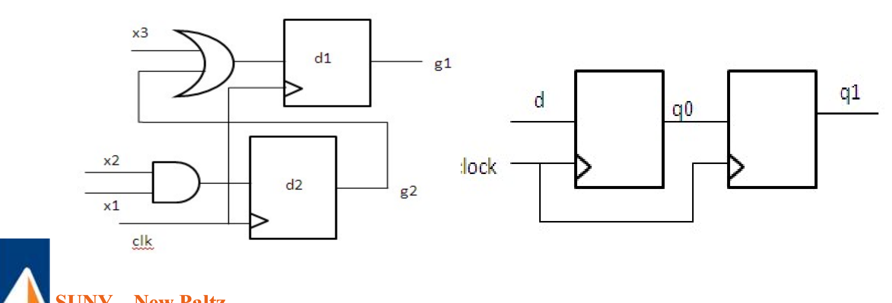
- **Recommended : use non blocking assignments for modeling clocked processes in sequential logic in procedural blocks**

SUNY – New Paltz
Elect. & Comp. Eng.

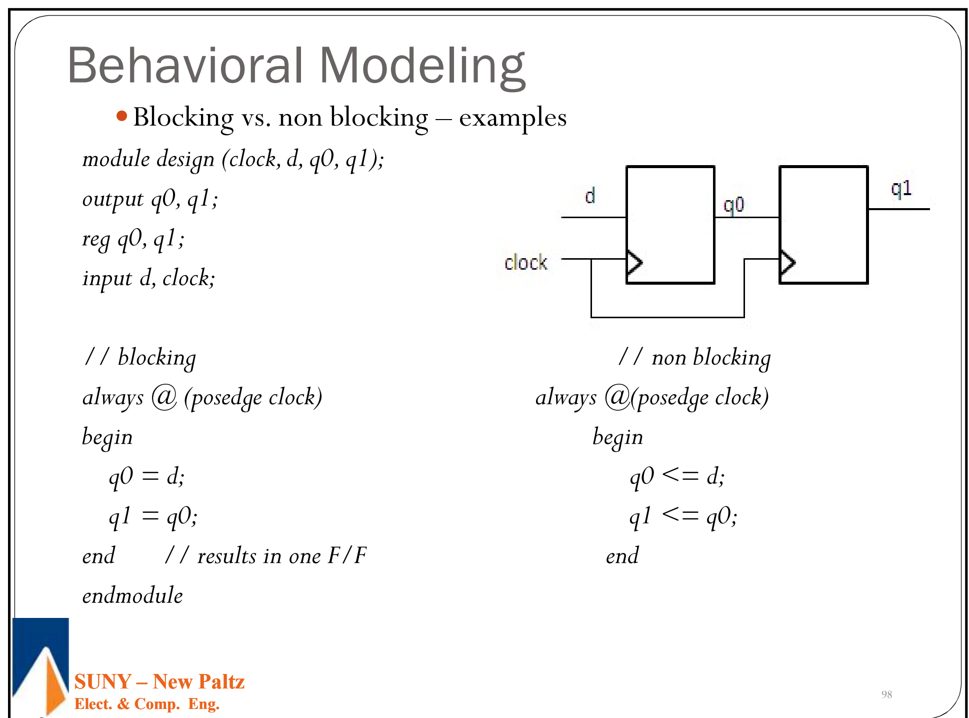
96

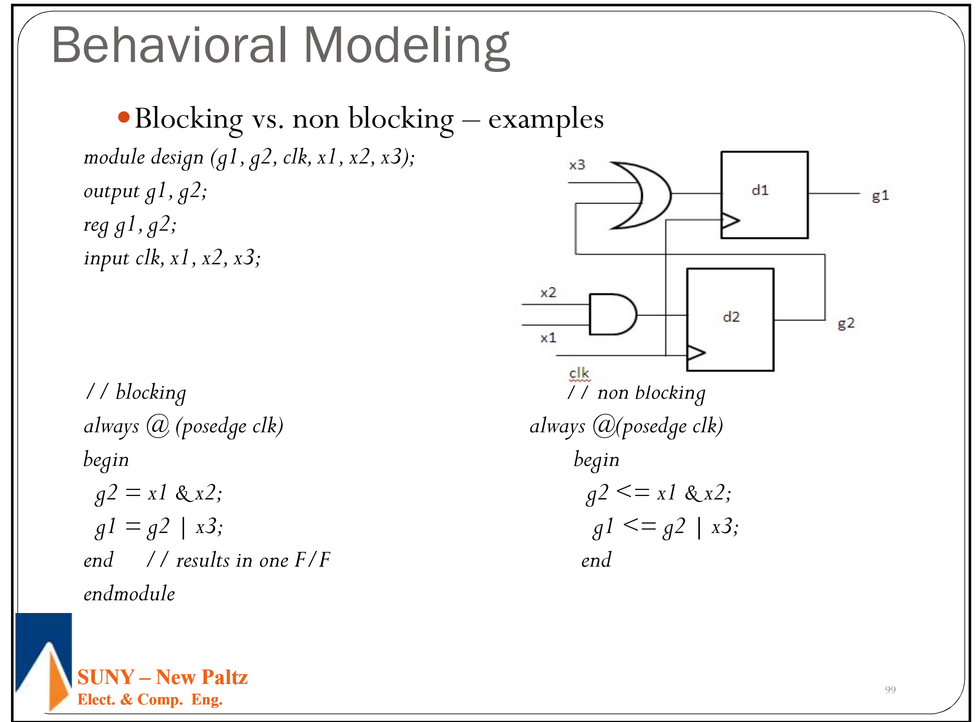
•Behavioral Modeling

- Blocking vs. non blocking – examples
- Write Verilog code for the following circuits using blocking and non blocking assignments
- Test these circuits by changing the order of the assignments
- Use a synthesis tool to showcase that assignments order does not matter in non blocking assignments whereas it does in blocking assignments



SUNY – New Paltz
Elect. & Comp. Eng.
97





Behavioral Modeling

- Procedural statements – if , if else if
- if-else statement conditionally controls a logic operation

```
always @*
    if (Ctl)
        Z = A & B;
    else
        Z = A | B;
```

- if-else implies multiplexing logic like conditional operator (? :)

```
reg Y;
always @*
   if (Sel)
      y = A;
   else
      Y = B;
```

⇔

```
wire Y;
assign Y = Sel ? A : B
```

SUNY – New Paltz
Elect. & Comp. Eng.

100

Behavioral Modeling

```
reg Y;
always @(Sel or A)
   if (Sel)
      Y = A;
```

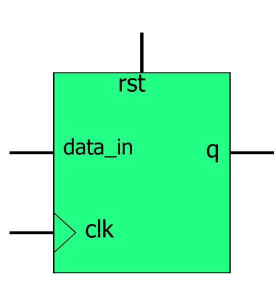
- Procedural statements – if , if else if
- if without else implies storage
- What is the value of Y when Sel = ‘0’?
- Verilog register variables hold their value when not driven, therefore hardware storage elements are required
- From synthesis point of view, it is a good design practice for every `if` to have an `else` unless you need storage
- A continuous assignment establishes static binding for net variables
- A procedural continuous assignment “assign – deassign” establishes a dynamic binding for register variables

101

Behavioral Modeling

- Modeling a flip flop – if else - example

```
module Flip_flop (q, data_in, clk, rst );
   output  q;
   reg  q;
   input  data_in, clk, rst;
   always @ ( posedge clk )
   begin
      if ( rst == 1)   // synchronous rst
         q = 0;
      else
         q = data_in;
   end
endmodule
```



SUNY – New Paltz
Elect. & Comp. Eng.

102

Behavioral Modeling

- Procedural statements – if else – example
- Modeling a comparator2

```
module comparator2 (c, a, b);
  output c;
  reg c;
  input  a;
  input  b;
  always @ (a or b)
    if (a == b)
      c = 1'b1;
    else
      c = 1'b0;
 endmodule
```

SUNY – New Paltz
Elect. & Comp. Eng.

103

Behavioral Modeling

- Procedural statements – nested if else – example
- Modeling an 8 to 3 bit priority encoder

```
module priority_encoder(code, sel);
   output reg [2:0] code;
    input [7:0] sel;
   always @(sel) begin
     if (sel[0]) code <= 3'b000;
     else if (sel[1]) code <= 3'b001;
     else if (sel[2]) code <= 3'b010;
     else if (sel[3]) code <= 3'b011;
     else if (sel[4]) code <= 3'b100;
     else if (sel[5]) code <= 3'b101;
     else if (sel[6]) code <= 3'b110;
     else if (sel[7]) code <= 3'b111;
     else code <= 3'bxxx;
   end
endmodule
```

SUNY – New Paltz
Elect. & Comp. Eng.

104

Behavioral Modeling

- Procedural statements – nested if else – example
- Modeling a 2x1, 4x1 mux

```
module mux2x1(f, s, w0, w1);
output f;
reg f;
input w0, w1, s;

always @ (*)
begin
  if (s == 0)
    f = w0;
  else
    f = w1;
end
endmodule
```

```
module mux4x1(f, s, w0, w1, w2, w3);
  output reg f;
  input w0, w1, w2, w3;
  input [1:0] s;
  always @ (*)
   if (s == 2'b00)
     f = w0;
   else if (s == 2'b01)
     f = w1;
   else if (s == 2'b10)
     f = w2;
   else if (s == 2'b11)
     f = w3;
   else
     f = 1'bx;
endmodule
```

SUNY – New Paltz
Elect. & Comp. Eng.

105

Behavioral Modeling

- Procedural statements – if else if - Modeling an up counter

```
// 8 bit up counter
module up_counter(out, enable, clk, reset );
  output [7:0] out;
  input enable, clk, reset;
  reg [7:0] out;
  always @(posedge clk)
   if (reset) begin
     out <= 8'b0 ;
   end else if (enable) begin
     out <= out + 1;
   end
endmodule
```

SUNY – New Paltz
Elect. & Comp. Eng.

106

Behavioral Modeling

- Procedural statements – case, casex, casez
- case construct in Verilog gives another way to organize a conditional expression with many alternatives
- behaves like nested if - else statement where case items are examined in order
- simulation is capable of comparing ‘x’s and ‘z’s explicitly
- casez treats ‘z’ as don’t cares
- casex treats both ‘x’ and ‘z’ as don’t cares
- exact match between case expression and case item

107

Behavioral Modeling

- Procedural statements – case, casex, casez

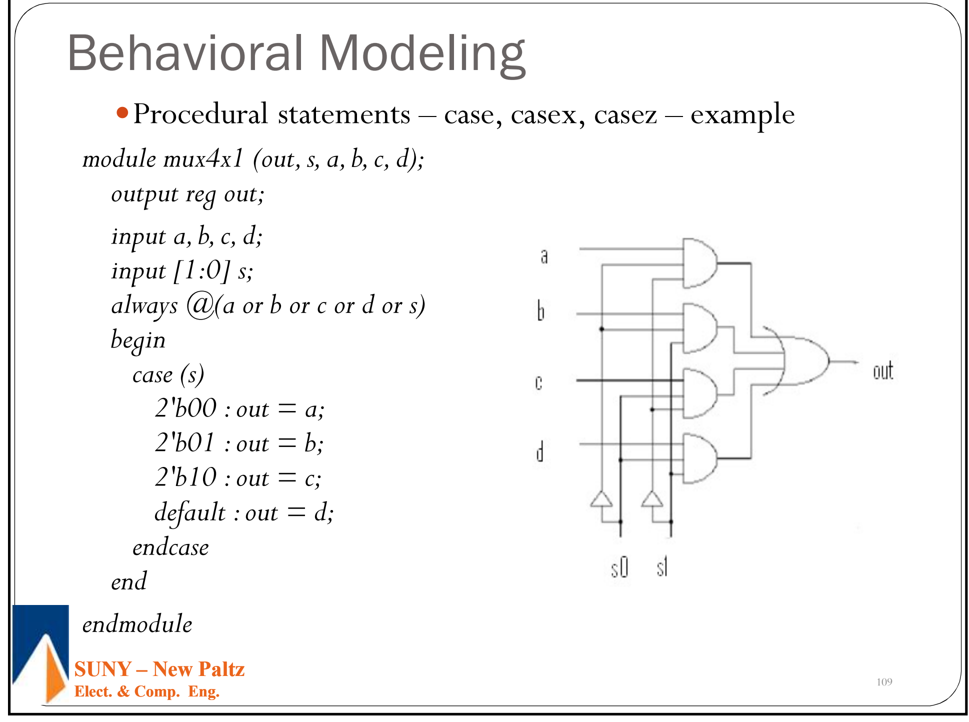
```
case (case_expression)
   case_item1 : statement1;
   case_item2 : statement2;
   case_item3 : statement3;
   case_item4 : statement4;
   default    : statement5;
endcase
```

Equivalent to

```
if      (case_expression === case_item1) statement1;
else if (case_expression === case_item2) statement2;
else if (case_expression === case_item3) statement3;
else if (case_expression === case_item4) statement4;
else                                     statement5;
```

SUNY – New Paltz
Elect. & Comp. Eng.

108



Behavioral Modeling

- Procedural statements – case, casex, casez – example

```
module decoder2to4 (y, w, en);
   output reg [3:0] y;
   input en;
   input [1:0] w;
   always @(*) begin
      if (en == 0)
         y = 4'b0000;
      else
       case (w)
          2'b00 : y = 4'b1000;
          2'b01 : y = 4'b0100;
          2'b10 : y = 4'b0010;
          2'b11 : y = 4'b0001;
       endcase
   end
endmodule
```

SUNY – New Paltz
Elect. & Comp. Eng.

110

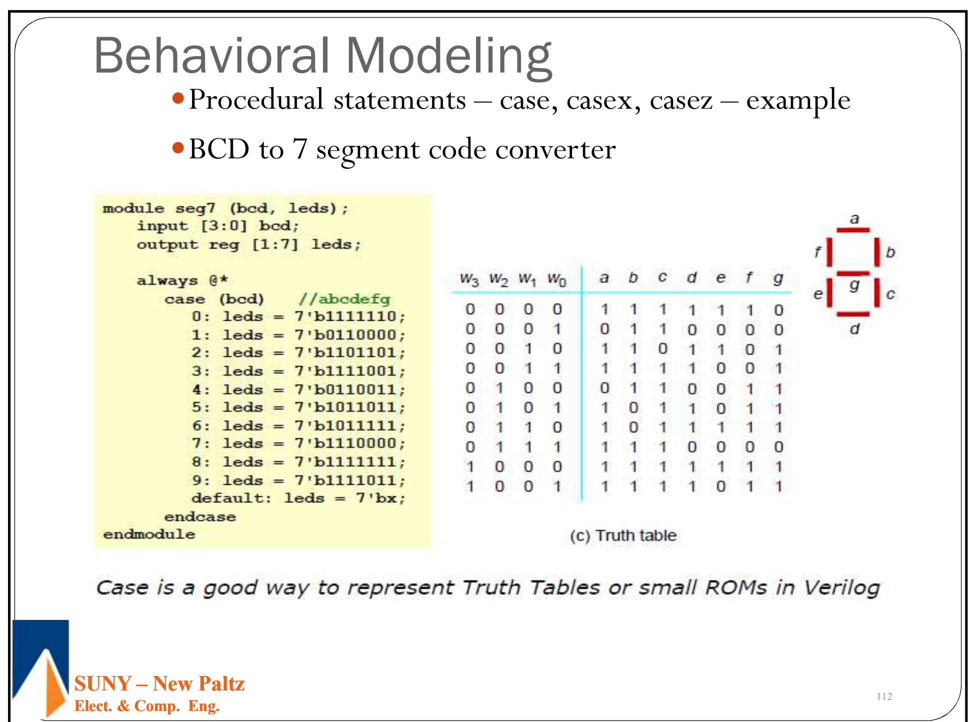
Behavioral Modeling

- Procedural statements – case, casex, casez – example
- Only case-based decoder (previous example re-written)

```
module decoder2to4 (y, w, en);
    output reg [3:0]y;
    input en;
    input [1:0] w;
    always @(*) begin
        case ({en, w})
            3'b100 : y = 4'b1000;
            3'b101 : y = 4'b0100;
            3'b110 : y = 4'b0010;
            3'b111 : y = 4'b0001;
            default : y = 4'b0000;
        endcase
    end
endmodule
```

SUNY – New Paltz
Elect. & Comp. Eng.

111



Behavioral Modeling

- Procedural statements – case, casex, casez – example
- BCD to 7 segment code converter

```
module seg7 (bcd, leds);
   input [3:0] bcd;
   output reg [1:7] leds;

   always @*
      case (bcd)    //abcdefg
         0: leds = 7'b1111110;
         1: leds = 7'b0110000;
         2: leds = 7'b1101101;
         3: leds = 7'b1111001;
         4: leds = 7'b0110011;
         5: leds = 7'b1011011;
         6: leds = 7'b1011111;
         7: leds = 7'b1110000;
         8: leds = 7'b1111111;
         9: leds = 7'b1111011;
         default: leds = 7'bx;
      endcase
endmodule
```

w_3	w_2	w_1	w_0	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

(c) Truth table

Case is a good way to represent Truth Tables or small ROMs in Verilog

SUNY – New Paltz
Elect. & Comp. Eng.

112

Behavioral Modeling

- Procedural statements – case, casex, casez – example
- Looking at designing arithmetic blocks – Function codes for 74381 TTL ALU

Operation	Inputs (S2 S1 S0)	Functional output
CLEAR	000	0000
B - A	001	B - A
A - B	010	A - B
ADD	011	A + B
XOR	100	A xor B
OR	101	A or B
AND	110	A and B
PRESET	111	1111

113

Behavioral Modeling

- Procedural statements – case, casex, casez – example
- Looking at designing arithmetic blocks

```
module alu(s, A, B, F);
    input [2:0] s;
    input [3:0] A, B;
    output reg [3:0] F;

    always @(*)
      case (s)
        0: F = 4'b0000;
        1: F = B - A;
        2: F = A - B;
        3: F = A + B;
        4: F = A ^ B;
        5: F = A | B;
        6: F = A & B;
        7: F = 4'b1111;
      endcase
endmodule
```

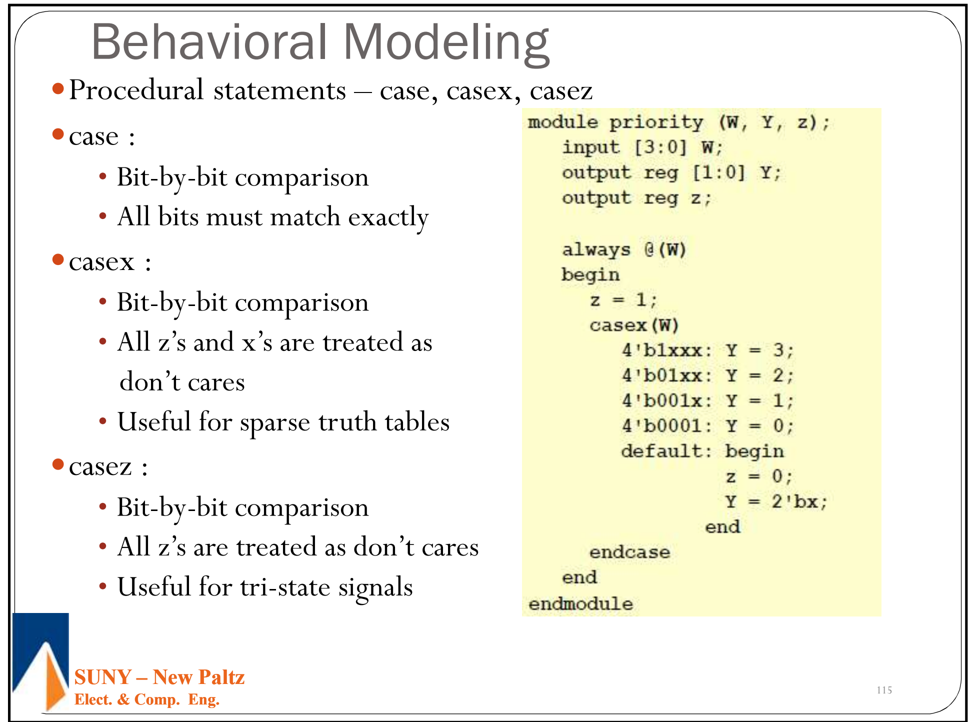
Verilog Arithmetic Operators (cases 1–3)

Verilog Logical Operators (cases 4–6)

No 'default' case needed because all possibilities are enumerated

SUNY – New Paltz
Elect. & Comp. Eng.

114



Behavioral Modeling

- Procedural statements – case, casex, casez – example

```
module case_compare;
reg sel;
initial begin
  #10 $display("\n  Driving 0");
  sel = 0;
  #10 $display("\n  Driving 1");
  sel = 1;
  #10 $display("\n  Driving x");
  sel = 1'bx;
  #10 $display("\n  Driving z");
  sel = 1'bz;
  #10 $finish;
end
```

```
always @ (sel)
 case (sel)
   1'b0 : $display("Normal : Logic 0 on sel");
   1'b1 : $display("Normal : Logic 1 on sel");
   1'bx : $display("Normal : Logic x on sel");
   1'bz : $display("Normal : Logix z on sel");
  endcase
```

SUNY – New Paltz
Elect. & Comp. Eng.

116

•Behavioral Modeling

- Procedural statements – case, casex, casez – example

```
always @ (sel)
  casex (sel)
    1'b0 : $display("CASEX : Logic 0 on sel");
    1'b1 : $display("CASEX : Logic 1 on sel");
    1'bx : $display("CASEX : Logic x on sel");
    1'bz : $display("CASEX : Logix z on sel");
  endcase
always @ (sel)
  casez (sel)
    1'b0 : $display("CASEZ : Logic 0 on sel");
    1'b1 : $display("CASEZ : Logic 1 on sel");
    1'bx : $display("CASEZ : Logic x on sel");
    1'bz : $display("CASEZ : Logix z on sel");
  endcase
endmodule
```

SUNY – New Paltz
Elect. & Comp. Eng.

117

Behavioral Modeling

- Procedural statements – for, while, repeat, forever
- Looping statements appear inside procedural blocks only
- forever
  - The forever loop executes continuously i.e. the loop never ends
  - Normally we use forever statements in initial blocks for clock generation and synchronization with other hardware in test-benches
  - One should be very careful in using a forever statement because if no timing construct is present in the forever statement the simulation could hang

SUNY – New Paltz
Elect. & Comp. Eng.

118

Behavioral Modeling

- Procedural statements – forever – example – named block

```
parameter half_cycle = 50;

initial
   begin : clock_loop
      clock = 0;
      forever begin
         #half_cycle clock = 1;
         #half_cycle clock = 0;
      end
   end

initial
   #350 disable clock_loop;
```

SUNY – New Paltz
Elect. & Comp. Eng.

119

Behavioral Modeling

- Procedural statements – repeat – example
- Executes the loop a fixed number of times

```
integer count;
initial begin
  count = 0;
  // repeat the block 128 times
  repeat (128)
  begin
    $display("count = %d \n", count);
    count = count + '1';
  end
end
```

```
...
w_address = 0;
repeat ( memory_size )
    begin
        memory [w_address] = 0;
        w_address = w_address+1;
    end
...
```

SUNY – New Paltz
Elect. & Comp. Eng.

120

Behavioral Modeling

- Procedural statements – while – example
- Executes the loop as long as the condition evaluates to true

```
module while_example();
reg [5:0] loc;
reg [7:0] data;
always @ (data or loc) begin
  loc = 0;
 //data=0, loc=32 (invalid value)
  if (data == 0) begin
    loc = 32;
  end else begin
    while (data[0] == 0) begin
      loc = loc + 1;
      data = data >> 1;
    end
  end
 $display ("DATA = %b, LOCATION = %d", data, loc);
end
initial begin
  #10 data = 8'b11;
  #10 data = 8'b100;
  #10 data = 8'b1000;
  #10 data = 8'b1000_0000;
  #10 data = 8'b0;
  #10 $finish;
end
endmodule
```

121

Behavioral Modeling

- Procedural statements – for – example

```
module for_example();
integer i;
reg [7:0] ram [0:255];
initial begin
 for (i = 0; i < 256; i = i + 1) begin
   #10 $display("Address = %g  Data = %h", i, ram[i]);
   ram[i] <= 0; // Initialize the RAM with 0
   #10 $display("Address = %g  Data = %h", i, ram[i]);
 end
 #10 $finish;
end
endmodule
```

SUNY – New Paltz
Elect. & Comp. Eng.

122

Insight into Verilog language – parameters

- parameter work like a constant in a module or its port list
- they are not variables i.e. they assign value to a symbolic name
- help in parameterization of a module
- can be changed using defparam at module instantiation
- instead, if localparams are defined they can not be changed

```
/* n – bit counter*/
module countern (clk, reset, load, en, d_in, value);
  input clk, reset, load, en;
  input [WIDTH-1:0] d_in;
  output reg [WIDTH-1:0] value;
  parameter WIDTH = 8;
   initial value <= 0;
   always @ (posedge clk or negedge reset)
      if (reset)  value <= 0;  //Asynchronous reset
      else begin
          if (load) value <= d_in;
          else if (en) value <= value + 1;
      end
endmodule
```

Elect. & Comp. Eng.

123

Insight into Verilog language – parameters

```
module callcountern(clk, reset, receive,, data, frame_cnt);
   input clk, reset, receive;
   output [11:0] frame_cnt;
   countern datac(.clk(clk), .reset(reset), .load(~receive), .en(receive), .d_in(data),
   .value(frame_cnt));
   defparam datac.WIDTH = 12;
endmodule
/* 8 bit parity checker */
module parity (y, in);
  parameter size = 8;
  input [size-1:0] in;
  output y;
 assign y = ^in;
endmodule
```

SUNY – New Paltz
Elect. & Comp. Eng.

124

Insight into Verilog language – parameters

- Can be included in module declaration before module port list

```
module adder #(parameter MSB = 32, LSB = 0)
  (output reg [MSB:LSB] sum,
   output reg co,
   input wire [MSB:LSB] a, b,
   input wire ci, ……
  )
  …
endmodule
```

```
/* 8 bit parity checker */
module parity (y, in);
  parameter size = 8;
  input [size-1:0] in;
  output y;
 assign y = ^in;
endmodule

module tb_parity;
  reg [15:0] data;
  wire y;
  parity #(16) io0(y, data);
  parity #(.size(16)) io1(y, data);
endmodule
```

125

Behavioral Modeling

- Procedural statements – function and task
- Verilog provides tasks and functions to break up large behavioral designs into smaller pieces
- Tasks and functions allow the designer to abstract Verilog code that is used at many places in design reducing code repetition
- Tasks are used in all programming languages, generally known as procedures or subroutines and they include Verilog behavioral code only enclosed in **task**…..**endtask** keywords
- Usually, tasks are defined in the module (local) in which they are used
- It is possible to define a task in a separate file also and use the compile directive **'include** to include the task in the file in which it is instantiated
- Tasks have any number of input, output and inout arguments
- The order of declaration within the task defines how the arguments are passed to the task by the caller

SUNY – New Paltz
Elect. & Comp. Eng.

126

Behavioral Modeling

- Procedural statements – task
- The variables declared within the task are local to that task
- tasks can include timing delay e.g. posedge, negedge, #delay & wait
- tasks can take, drive and source global variables, when no local variables are used
- When local variables are used, basically output is assigned only at the end of task execution
- tasks can call another task or function
- tasks can be used for modeling combinational & sequential logic
- A task must be specifically called with a statement, it cannot be used within an expression as a function can

```
task task_name;
    input, output, and inout declarations
    local variable declarations
    procedural_statement or statement_group
  endtask
```

SUNY – New Paltz
Elect. & Comp. Eng.

127

Behavioral Modeling

- Procedural statements – task - example

```
task convert;
input [7:0] temp_in;
output [7:0] temp_out;
begin
 temp_out = (9/5)*( temp_in + 32);
end
endtask
```

```
// task with globals
reg [7:0] temp_out;
reg [7:0] temp_in;
task convert;
begin
 temp_out = (9/5)*( temp_in + 32);
end
endtask
```

```
module callingtask (temp_a, temp_b, temp_c, temp_d);
  input [7:0] temp_a, temp_c;
  output [7:0] temp_b, temp_d;
  reg [7:0] temp_b, temp_d;
  `include "mytask.v"
always @ (temp_a) begin
  convert (temp_a, temp_b);
end
always @ (temp_c) begin
  convert (temp_c, temp_d);
end
endmodule
```

SUNY – New Paltz
Elect. & Comp. Eng.

128

Behavioral Modeling

- Procedural statements – task - example

```
module bit_counter (data, count);
  input [7:0] data;
  output [3:0] count; reg [3:0] count;

  always @(data) t(data, count);

  task t;
    input [7:0] a; output [3:0] c; reg [3:0] c; reg [7:0] tmp;
    begin c = 0; tmp = a;
      while (tmp) begin
              c = c + tmp[0];
              tmp = tmp >> 1;
      end
    end
  endtask
endmodule
```

SUNY – New Paltz
Elect. & Comp. Eng.

129

•Behavioral Modeling

- Procedural statements – function
- A Verilog HDL function behaves the same as a task, but with differences
- functions are defined in the module (local) in which they are used
- It is possible to define functions in separate files and use compile directive **'include** to include the function in the file in which it is instantiated
- functions **can not include timing delays** e.g. posedge, negedge, # delay, that means functions should be executed in "zero" time delay i.e. functions can be used for **modeling combinational logic** only
- functions can **call other functions, but can not call tasks**
- functions can have any number of inputs but only one output
- The variables declared within the function are local to that function
- The order of declaration within the function defines how the variables are passed to the function by the caller

SUNY – New Paltz
Elect. & Comp. Eng.

130

Behavioral Modeling

- Procedural statements – function
- functions can take, drive, and source global variables, when no local variables are used
- When local variables are used, basically output is assigned only at the end of function execution
- A function return the value that is assigned to function name

```
function [size_or_type] function_name;
    input declarations
    local variable declarations
      procedural_statement or statement_group
  endfunction
```

SUNY – New Paltz
Elect. & Comp. Eng.

131

Behavioral Modeling

- Procedural statements – function - example

```
// function with locals
function myfunction;
input a, b, c, d;
begin
  myfunction = ((a+b) + (c-d));
end
endfunction
```

```
module  functiocalling(a, b, c, d, e, f);
input a, b, c, d, e ;
output f;
wire f;
`include "myfunction.v"
assign f =  (myfunction (a,b,c,d)) ? e : 0;
endmodulen_
```

132

Behavioral Modeling

- Procedural statements – function - example

```
module word_aligner (w_in, w_out);
  input [7:0] w_in;
  output [7:0] w_out;
  assign w_out = align (w_in);
  function [7:0] align;
    input [7:0] word;
    begin
      align = word;
      if (align != 0)
        while (align[7] == 0)
          align = align << 1;
    end
  endfunction
endmodule
```

SUNY – New Paltz
Elect. & Comp. Eng.

133

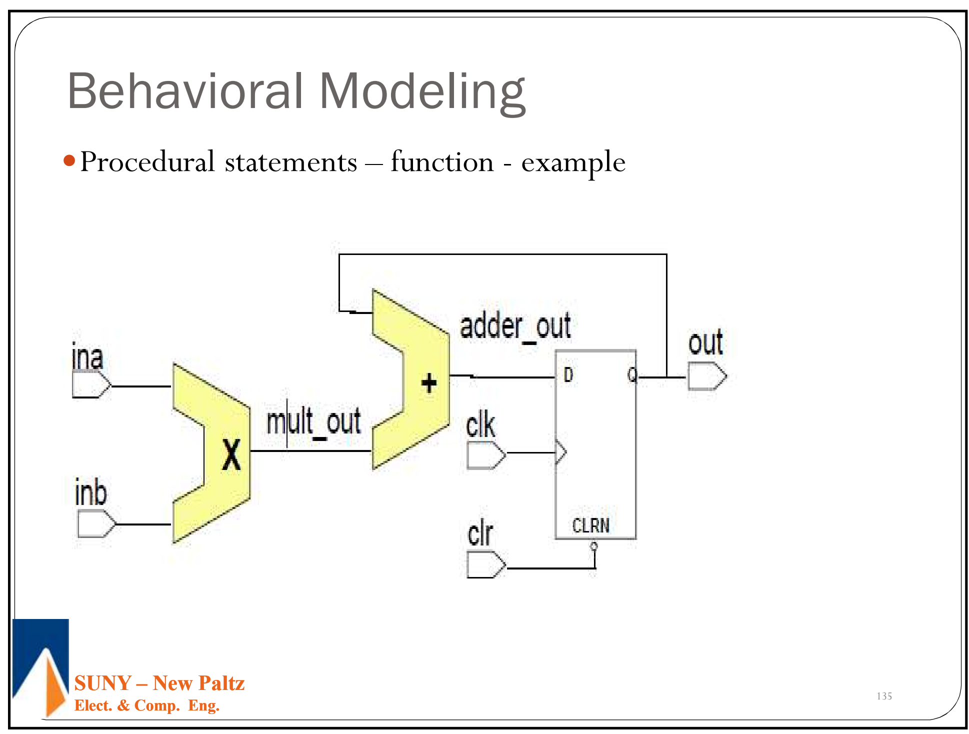
Behavioral Modeling

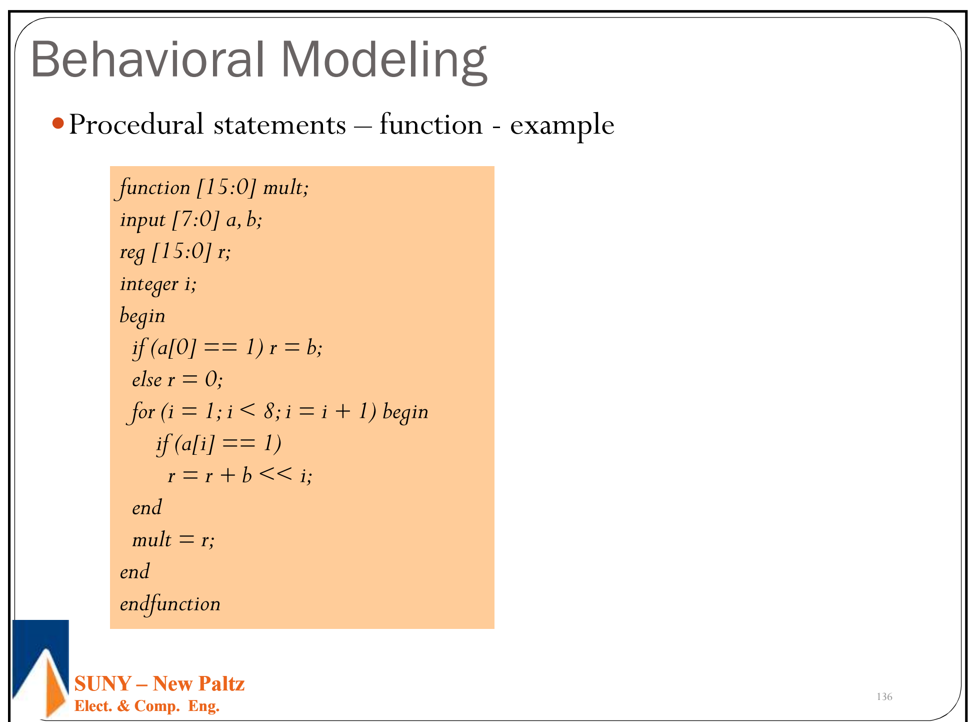
- Procedural statements – function - example

```
function [31:0] factorial;
input [3:0] operand;
reg [3:0] index;
begin
 factorial = operand ? 1 : 0;
 for (index = 2; index <= operand; index = index+1)
   factorial = index * factorial;
end
endfunction
...
assign res = n * factorial(n);
```

SUNY – New Paltz
Elect. & Comp. Eng.

134


Behavioral Modeling
Procedural statements – function - example
ina
inb
X
mult_out
+
adder_out
clk
clr
D
Q
CLRN
out
SUNY – New Paltz
Elect. & Comp. Eng.
135


Behavioral Modeling
Procedural statements – function - example
function [15:0] mult;
input [7:0] a, b;
reg [15:0] r;
integer i;
begin
if (a[0] == 1) r = b;
else r = 0;
for (i = 1; i < 8; i = i + 1) begin
if (a[i] == 1)
r = r + b << i;
end
mult = r;
end
endfunction
SUNY – New Paltz
Elect. & Comp. Eng.
136

Generate - for

- Work as a parallel block therefore allows following
  - instance creation, continuous assignment
  - initial and always blocks, local variable declaration
  - nesting of generate is allowed for all – generate - for, if, case

```
module RCAdder (a, b, ci, s);
   parameter N = 64;
   input [N-1:0] a, b;
   input ci;
   output [N-1:0] s;
   wire [N:0] pc;
   genvar i;
   generate for (i=0; i<N; i=i+1) begin : u1
      fulladder inst[i](a[i], b[i], pc[i], s[i], pc[i+1]);
   end
endmodule
```

SUNY – New Paltz
Elect. & Comp. Eng.
137

Conditional Generate - if

- Select at most one generate block from a set of alternative generate blocks based on constant expression

```
module multiplier(a,b,product);
   parameter a_width = 8, b_width = 8;
   localparam product_width = a_width + b_width;
   input [a_width-1:0] a;
   input [b_width-1:0] b;
   output [product_width-1:0] product;
   generate
     if ((a_width < 8 || b_width <8) )
    begin : mult
       cla_multiplier #(a_width, b_width) u1(a,b,product);
     end
     else begin : mult
       wallace_multiplier #(a_width, b_width) u1(a,b,product);
     end
   endgenerate
endmodule
```

SUNY – New Paltz
Elect. & Comp. Eng.
138

Conditional Generate - case

- Select at most one generate block from a set of alternative generate blocks based on constant expression

```
generate
    case (WIDTH)
      1: begin : adder
           adder_1bit x1(co, sum, a, b, ci);
         end
      2: begin : adder
           adder_2bit x1(co, sum, a, b, ci);
         end
      default: begin : adder
                 adder_cla  #(WIDTH) x1(co, sum, a, b, ci);
               end
    endcase
  endgenerate
```

SUNY – New Paltz
Elect. & Comp. Eng.

139

System Tasks

- These tasks are used for generating input, output during simulation and they start with a $ sign
- For displaying text on screen during simulation
  - $display – display text only once whenever this is executed
  - $strobe – display text only once every time this is executed at the very end of the current simulation time
  - $monitor – executes and displays every time if any of its parameter changes
- For displaying current simulation time
  - $time – as a 64 bit integer
  - $stime – as a 32 bit integer
  - $realtime – as a real number
- For generating random numbers
  - $random – a seed may be given otherwise seed is derived from system clock y

SUNY – New Paltz
Elect. & Comp. Eng.

140

System Tasks

- Controlling the simulation
  - $reset – resets the simulation back to time 0
  - $stop – halt the simulation and put it in interactive mode for user to enter the commands
  - $finish – exit the simulation completely and return to OS
- Scoping in case of hierarchical design
  - $scope – set the current hierarchical scope to the one provided as an argument to this command, if not then top scope is taken by default – simulator specific
  - $showscope – this is again simulator specific and may show list of all the modules, tasks, blocks in the current scope
- For enabling value change dump (vcd)
  - $dumpfile, $dumpvars, $dumpon, $dumpoff, $dumpall
- For file input, output
  - $fopen, $fdisplay, $fmonitor, $fwrite, $fstrobe

141

Compiler Directives

- Are mainly used for controlling the compilation of Verilog code
- A directive is effective from the point it is declared till the point another directive declaration arrives which overrides it
- `include – this directive is to support in-lining of Verilog code from another file in the place where it is declared
- `define – for defining text Macros
- `undef – for removing or disabling the text macros created by `define or the +define+ command line option
- `ifdef – for optionally including lines of Verilog code in the source code i.e. if a macro has been defined then the Verilog code within `ifdef will get compiled
- `else, `elseif – if the `ifdef condition is false then the Verilog code in `else or `elseif will get compiled
- `endif – for ending the `ifdef, `else combination in Verilog code

SUNY – New Paltz
Elect. & Comp. Eng.

142

Compiler Directives - example

```
module ifdef ();

initial begin
`ifdef FIRST
   $display("First code is compiled");
`else
  `ifdef SECOND
    $display("Second code is compiled");
  `else
    $display("Default code is compiled");
  `endif
`endif
 $finish;
end
endmodule
```

SUNY – New Paltz
Elect. & Comp. Eng.

143

Compiler Directives – `timescale

- `timescale <time_unit> / <time_precision>
- time_unit - the time multiplier for time values
- time_precision - minimum step size during simulation - determines rounding of numerical values
- Allowed unit/precision values:
  {1| 10 | 100, s | ms | us | ns | ps}
- Example: precision 10 ps / 1 ps

```
`timescale 10 ps / 1ps
nor #3.57 (z, x1, x2);
nor delay used = 3.57 x 10 ps = 35.7 ps => 36 ps
```

- Different timescales can be used for different sequences of modules
- The smallest time precision determines the precision of the simulation

SUNY – New Paltz
Elect. & Comp. Eng.

144

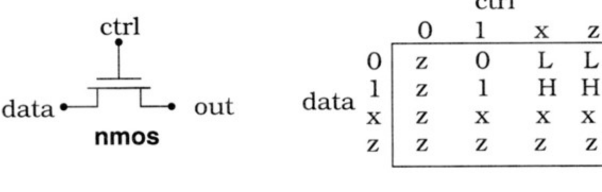
Switch Level Modeling

- MOS switches
  - nmos, pmos
  - cmos
- Buffer and Not, Tristated gates
  - buf, not
  - bufif0, bufif1, notif0, notif1
- Pullup and Pulldown gates
  - pullup
  - pulldown

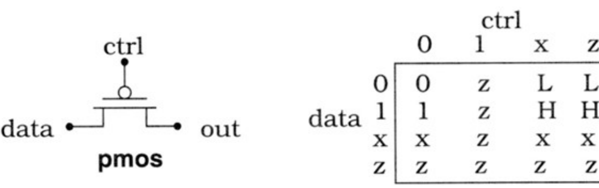
145

Switch Level Modeling

- MOS switches
  - nmos, pmos

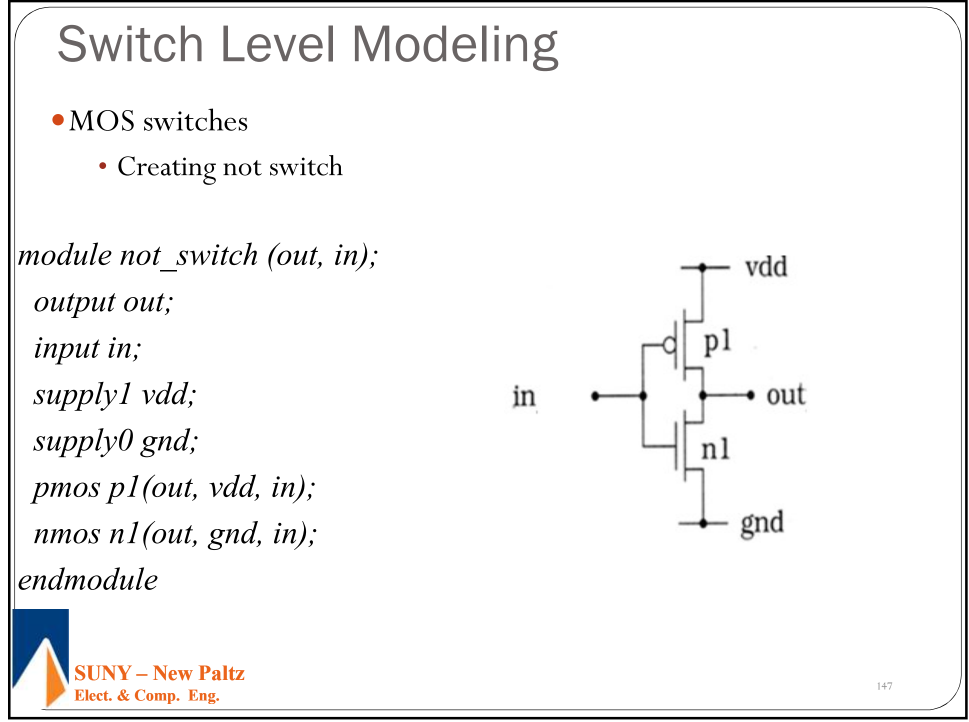


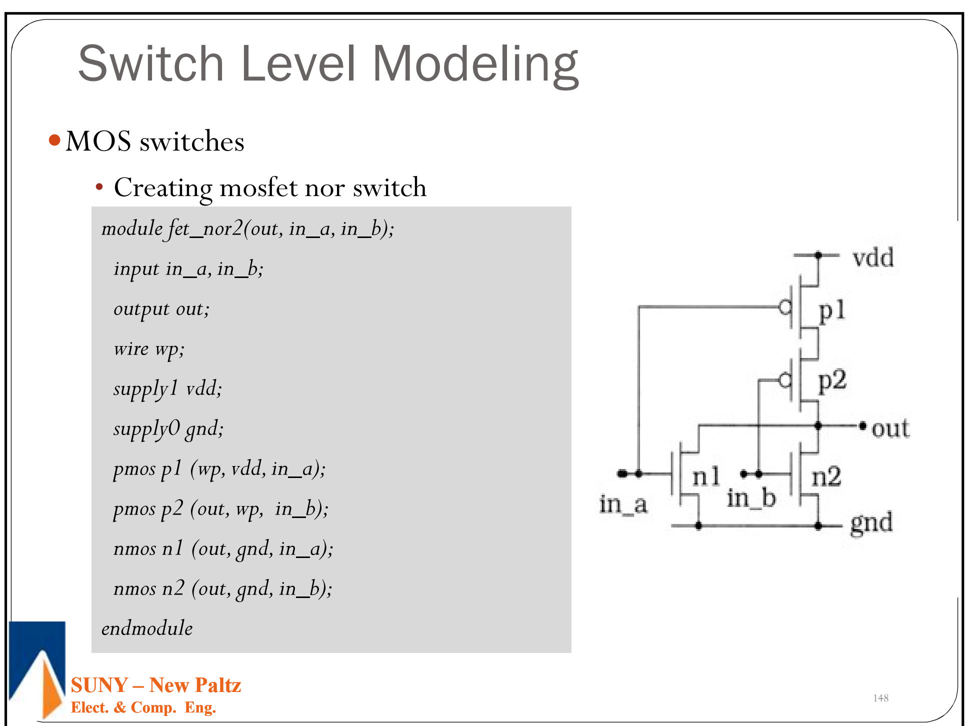
data \ ctrl	0	1	x	z
0	z	0	L	L
1	z	1	H	H
x	z	x	x	x
z	z	z	z	z

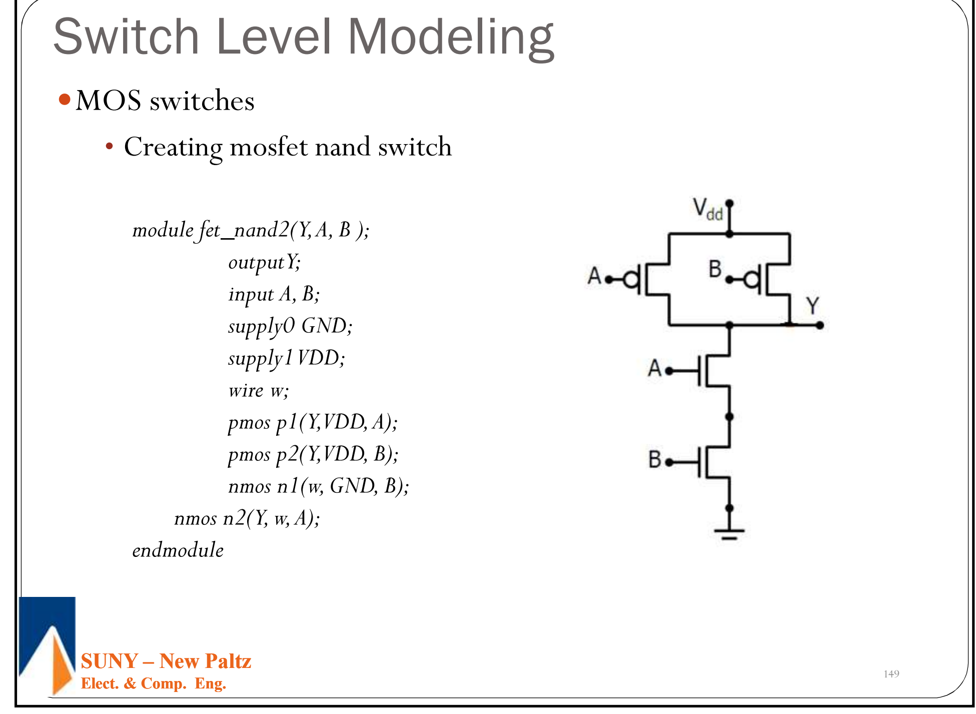


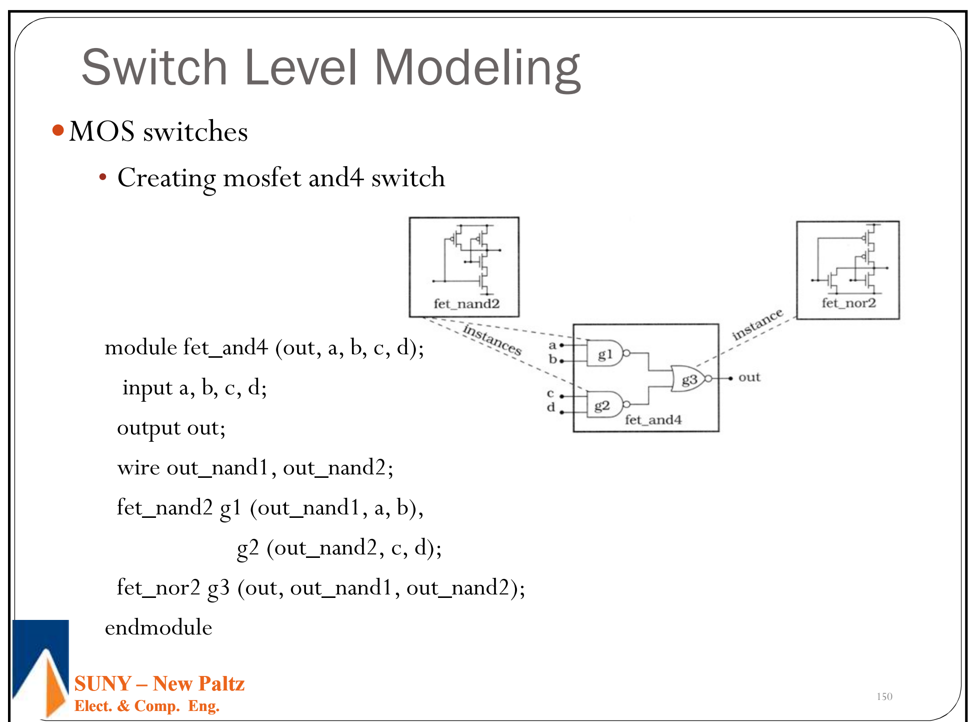
data \ ctrl	0	1	x	z
0	0	z	L	L
1	1	z	H	H
x	x	z	x	x
z	z	z	z	z

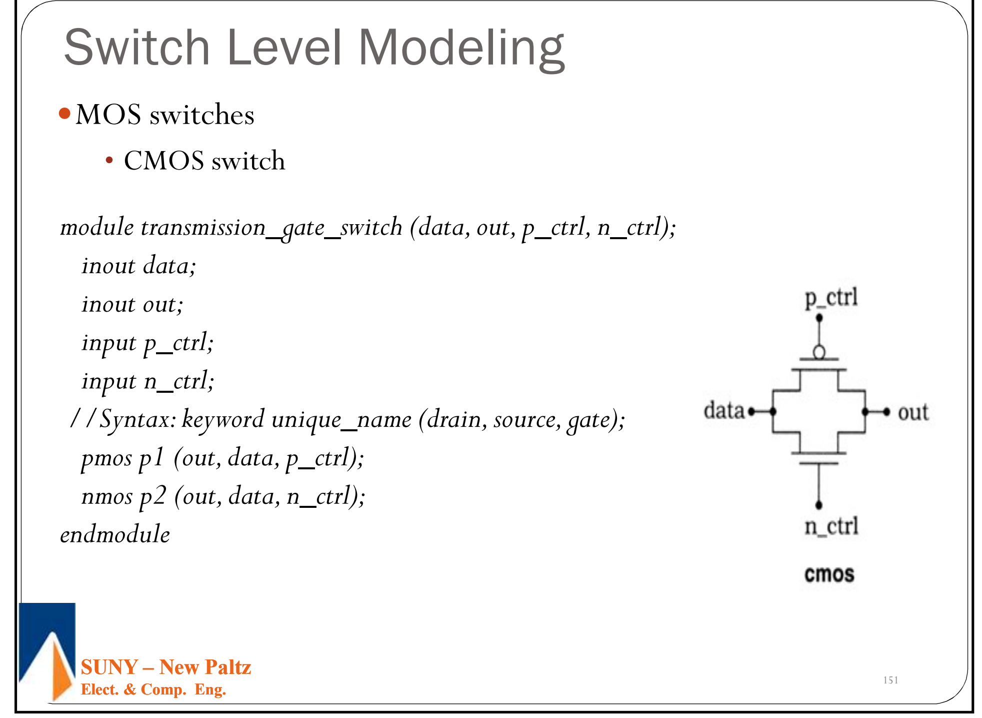
146

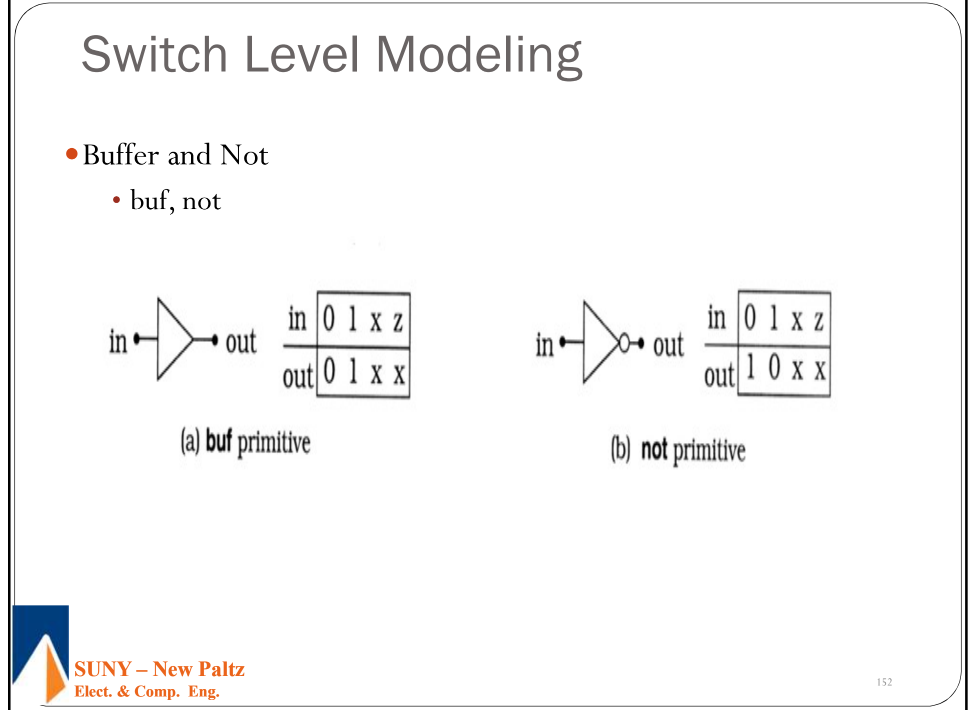

Switch Level Modeling
MOS switches
Creating not switch
module not_switch (out, in);
output out;
input in;
supply1 vdd;
supply0 gnd;
pmos p1(out, vdd, in);
nmos n1(out, gnd, in);
endmodule
vdd
p1
in
out
n1
gnd
SUNY – New Paltz
Elect. & Comp. Eng.
147


Switch Level Modeling
MOS switches
Creating mosfet nor switch
module fet_nor2(out, in_a, in_b);
input in_a, in_b;
output out;
wire wp;
supply1 vdd;
supply0 gnd;
pmos p1 (wp, vdd, in_a);
pmos p2 (out, wp, in_b);
nmos n1 (out, gnd, in_a);
nmos n2 (out, gnd, in_b);
endmodule
vdd
p1
p2
out
n1
in_b
n2
in_a
gnd
SUNY – New Paltz
Elect. & Comp. Eng.
148


Switch Level Modeling
• MOS switches
• Creating mosfet nand switch
module fet_nand2(Y, A, B);
output Y;
input A, B;
supply0 GND;
supply1 VDD;
wire w;
pmos p1(Y, VDD, A);
pmos p2(Y, VDD, B);
nmos n1(w, GND, B);
nmos n2(Y, w, A);
endmodule
Vdd
A
B
Y
A
B
SUNY – New Paltz
Elect. & Comp. Eng.
149

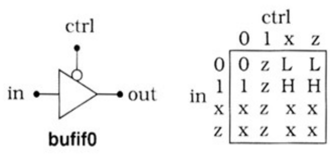

Switch Level Modeling
• MOS switches
• Creating mosfet and4 switch
module fet_and4 (out, a, b, c, d);
input a, b, c, d;
output out;
wire out_nand1, out_nand2;
fet_nand2 g1 (out_nand1, a, b),
g2 (out_nand2, c, d);
fet_nor2 g3 (out, out_nand1, out_nand2);
endmodule
fet_nand2
fet_nor2
instances
instance
a
b
c
d
g1
g2
g3
out
fet_and4
SUNY – New Paltz
Elect. & Comp. Eng.
150


Switch Level Modeling
MOS switches
CMOS switch
module transmission_gate_switch (data, out, p_ctrl, n_ctrl);
inout data;
inout out;
input p_ctrl;
input n_ctrl;
//Syntax: keyword unique_name (drain, source, gate);
pmos p1 (out, data, p_ctrl);
nmos p2 (out, data, n_ctrl);
endmodule
p_ctrl
data
out
n_ctrl
cmos
SUNY – New Paltz
Elect. & Comp. Eng.
151


Switch Level Modeling
Buffer and Not
buf, not
in
out
in 0 1 x z
out 0 1 x x
(a) buf primitive
in
out
in 0 1 x z
out 1 0 x x
(b) not primitive
SUNY – New Paltz
Elect. & Comp. Eng.
152

Switch Level Modeling

- Buffer and Not - tristated gates
  - bufif0, bufif1, notif0, notif1



bufif0 (rows: in, columns: ctrl)

in \ ctrl	0	1	x	z
0	0	z	L	L
1	1	z	H	H
x	x	z	x	x
z	x	z	x	x

bufif1 (rows: in, columns: ctrl)

in \ ctrl	0	1	x	z
0	z	0	L	L
1	z	1	H	H
x	z	x	x	x
z	z	x	x	x

notif0 (rows: in, columns: ctrl)

in \ ctrl	0	1	x	z
0	1	z	H	H
1	0	z	L	L
x	x	z	x	x
z	x	z	x	x

notif1 (rows: in, columns: ctrl)

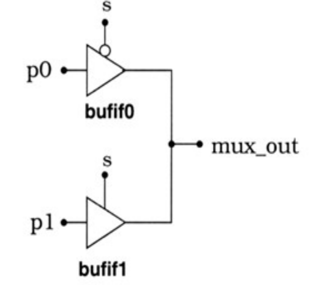
in \ ctrl	0	1	x	z
0	z	1	H	H
1	z	0	L	L
x	z	x	x	x
z	z	x	x	x

153

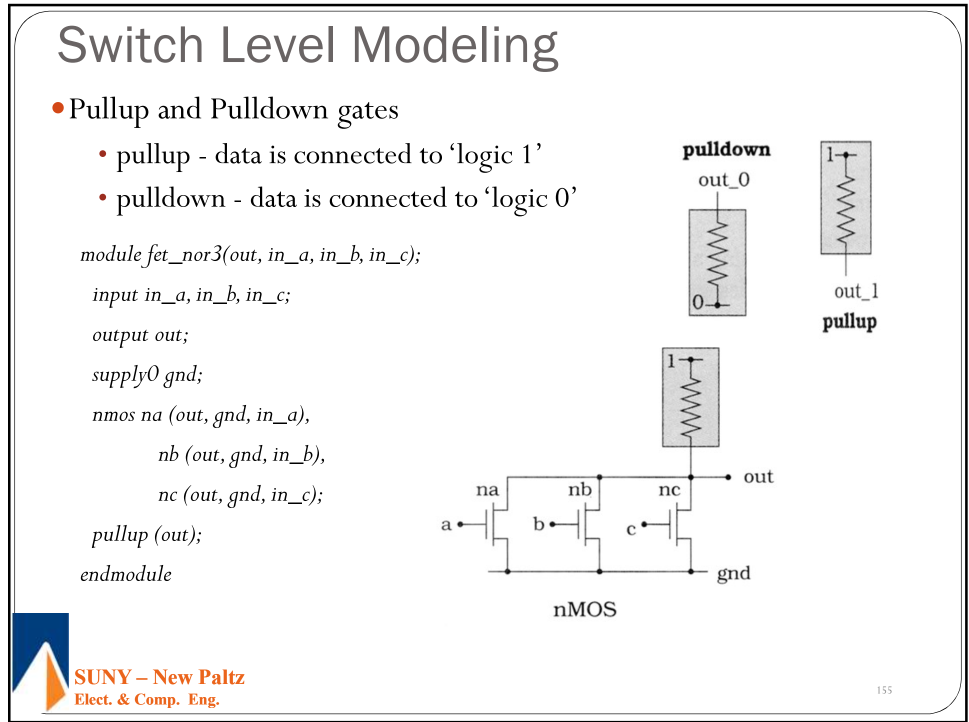
Switch Level Modeling

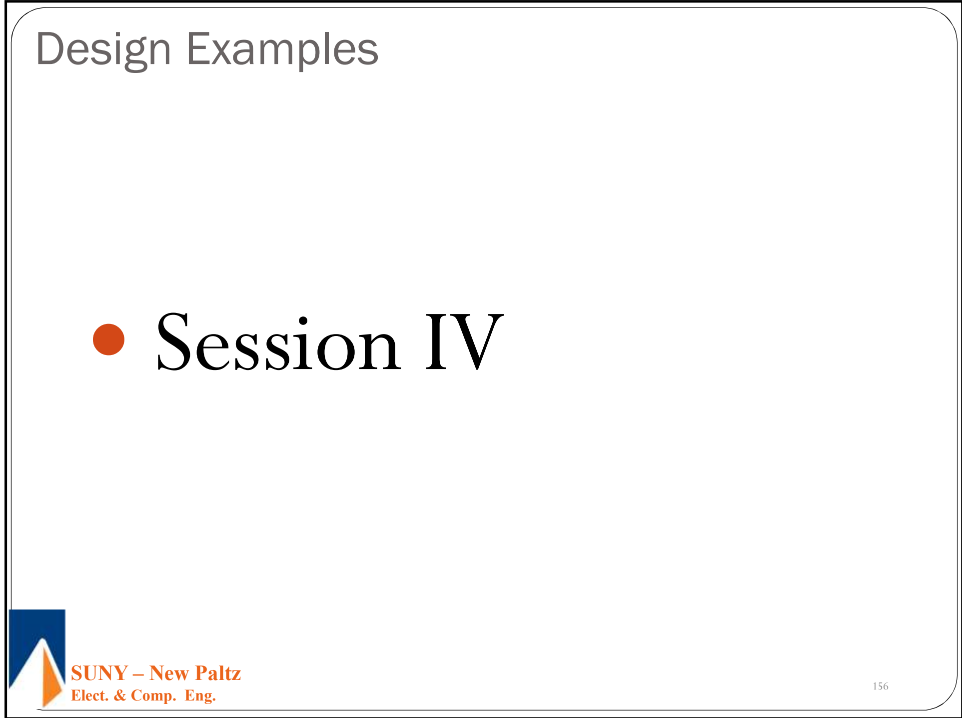
- Buffer and Not - tristated gates
  - Creating a 2x1 mux using tristated buffers

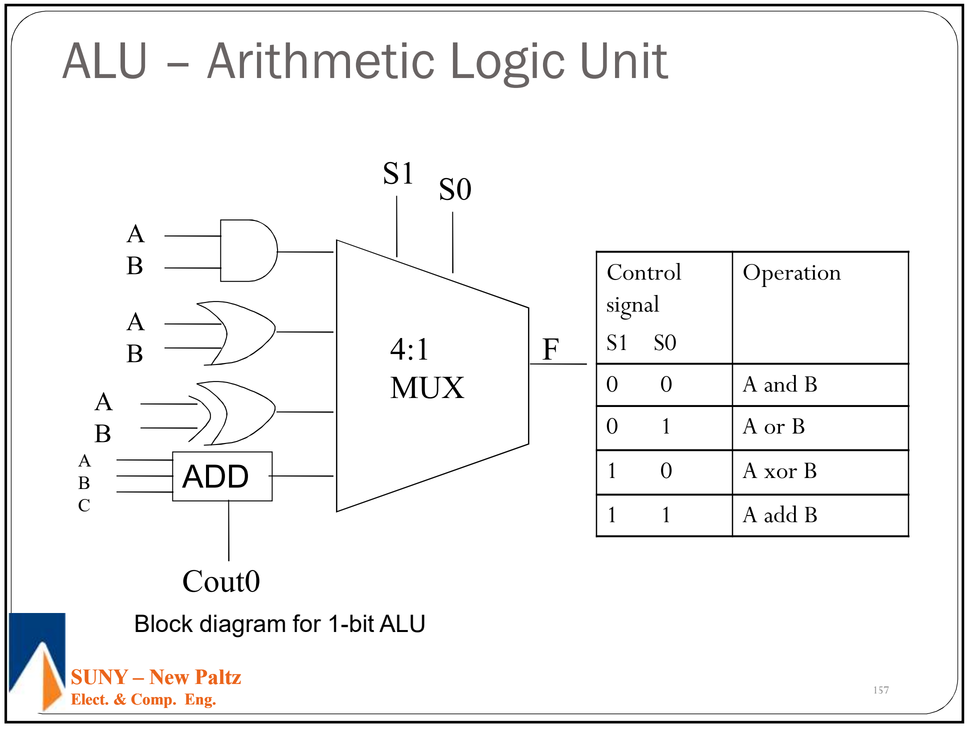
```
module mux_2_1 (mux_out, p0, p1, s);
  input p0, p1, s;
  output mux_out;

  bufif0 bf0(mux_out, p0, s);
  bufif1 bf1(mux_out, p1, s);

endmodule
```



154


Switch Level Modeling
Pullup and Pulldown gates
pullup - data is connected to 'logic 1'
pulldown - data is connected to 'logic 0'
module fet_nor3(out, in_a, in_b, in_c);
input in_a, in_b, in_c;
output out;
supply0 gnd;
nmos na (out, gnd, in_a),
nb (out, gnd, in_b),
nc (out, gnd, in_c);
pullup (out);
endmodule
pulldown
out_0
1
0
out_1
pullup
na
nb
nc
a
b
c
out
gnd
nMOS
SUNY – New Paltz
Elect. & Comp. Eng.
155


Design Examples
Session IV
SUNY – New Paltz
Elect. & Comp. Eng.
156



Design Examples

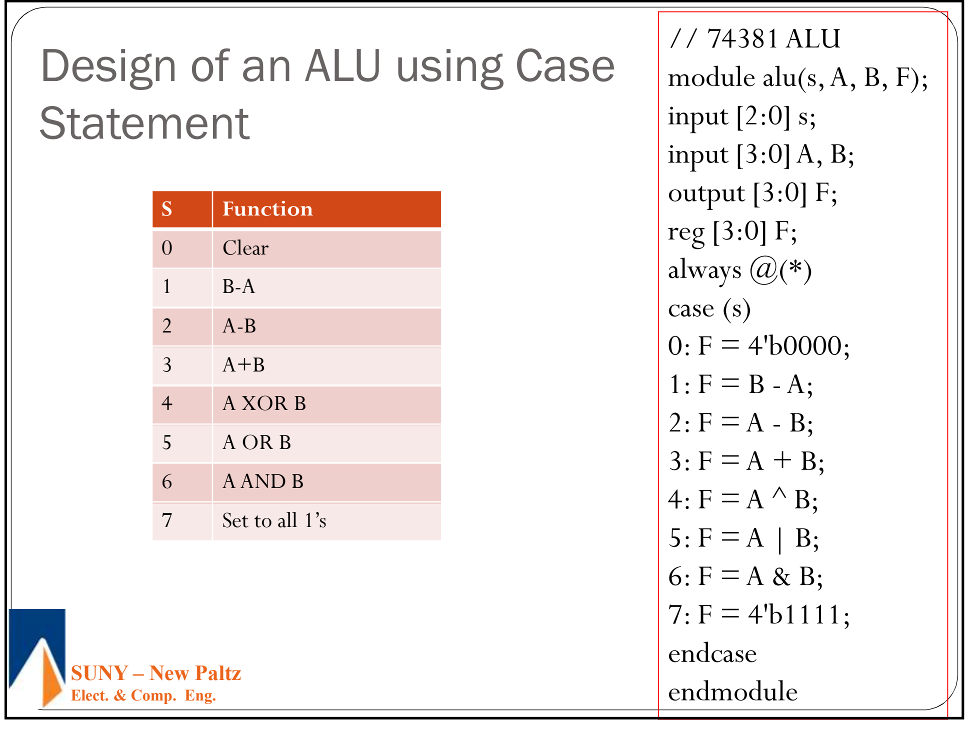
- ALU – Arithmetic Logic Unit

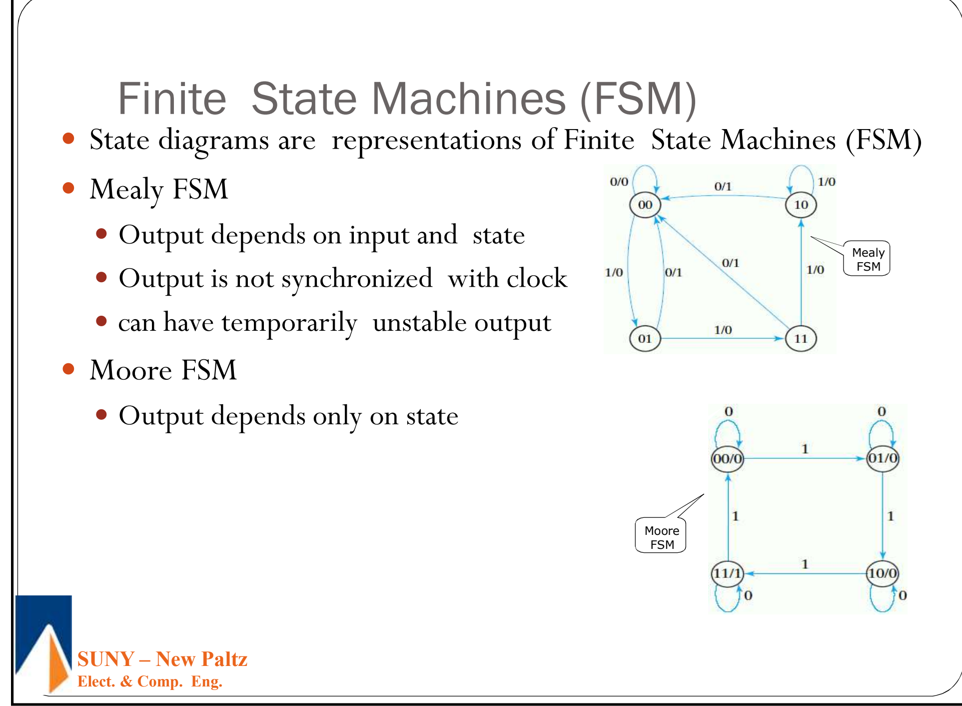
```
// 1-bit alu
module mux(f, s0, s1, w, x, y, z);
output f;
input s0, s1, w, x, y, z;
assign f = s1 ? (s0 ? x : y) : (s0 ? w :
z);
endmodule

module fulladder(z, c0, a, b, c);
output z,c0;
input a,b,c;
assign z = a ^ b ^ c;
assign c0 = (a&b) | (b&c) | (c&a);
endmodule
```

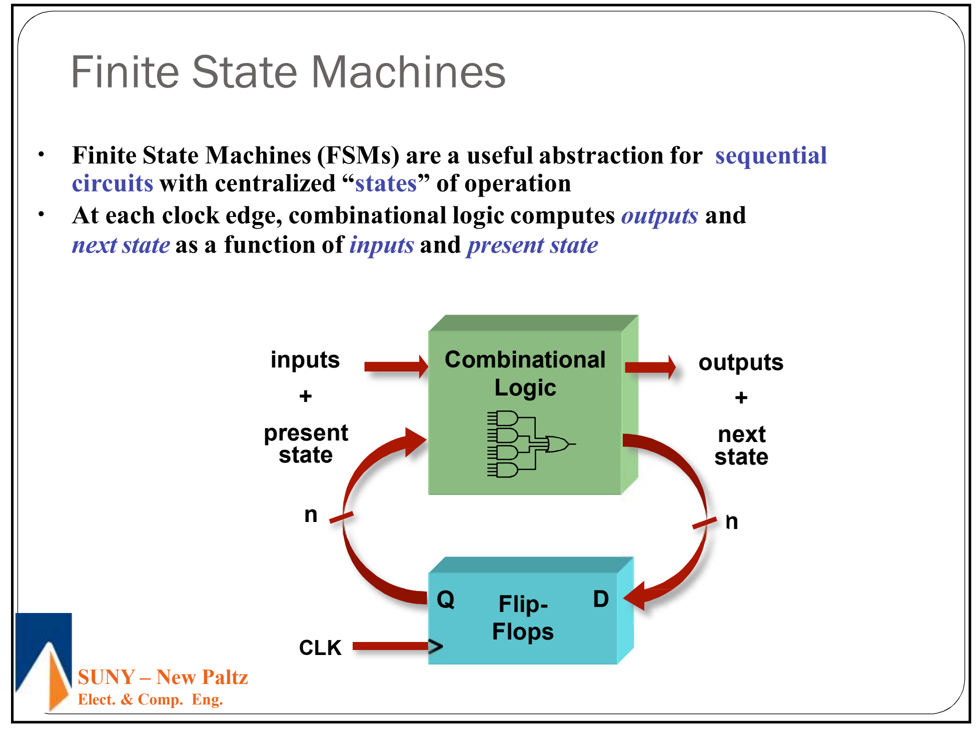
```
// 1-bit alu
module alu(F, CO, S0, S1, A, B, C);
output F, CO;
input S0, S1, A, B, C;
wire w, x, y, z;
and a1(w, A, B);
or o1(x, A, B);
xor x1(y, A, B);
fulladder f1(z, CO, A, B, C);
mux m1(F, S0, S1, w, x, y, z);
endmodule
```

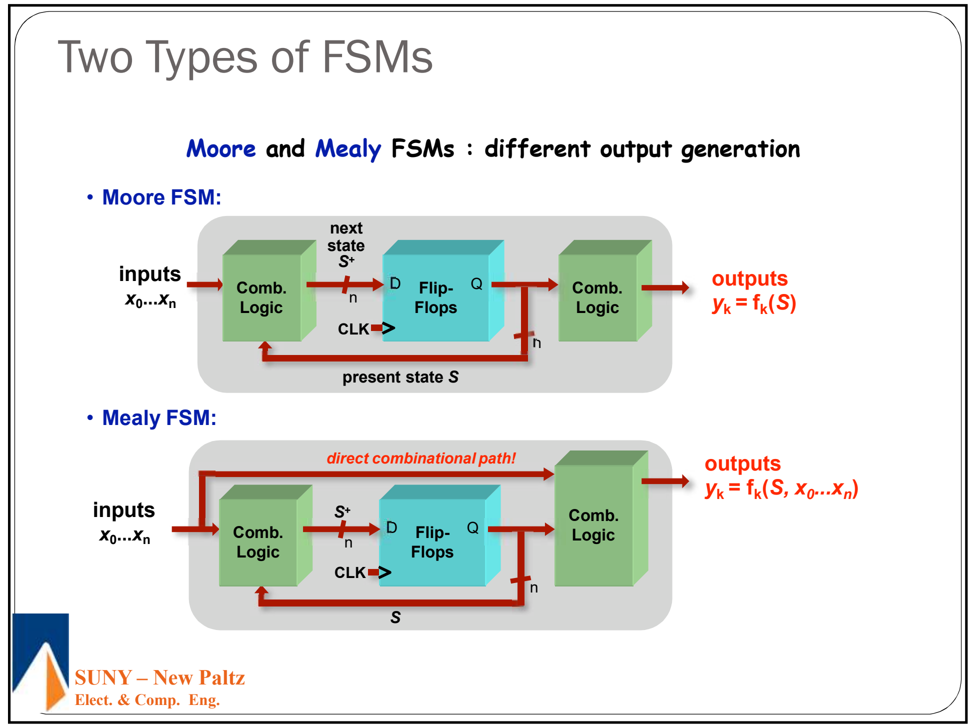
SUNY – New Paltz
Elect. & Comp. Eng.

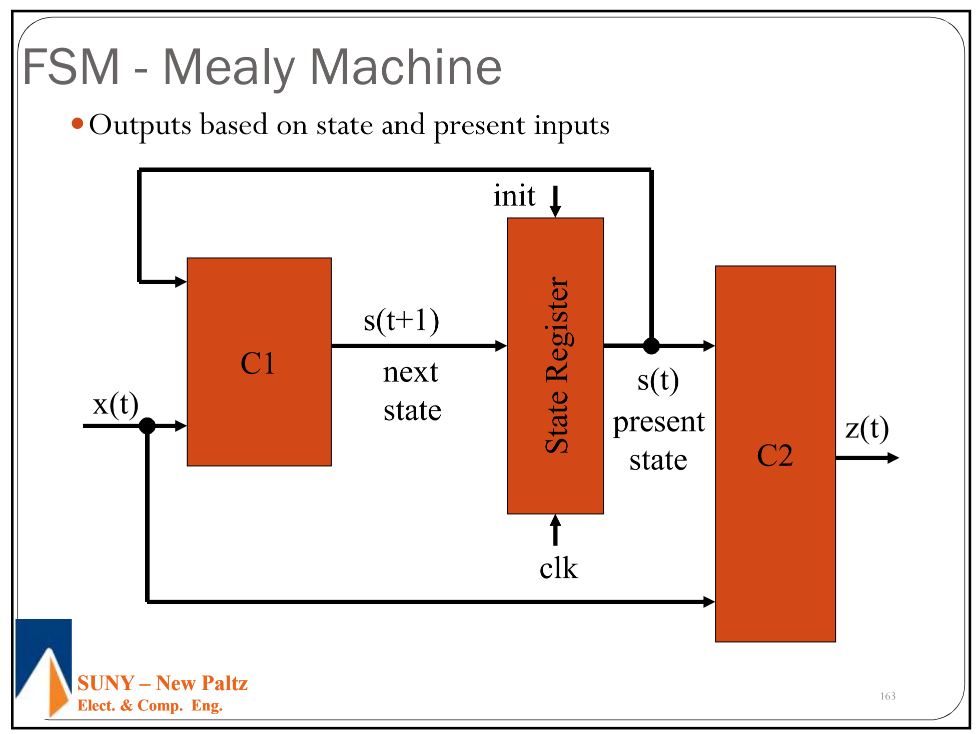
158

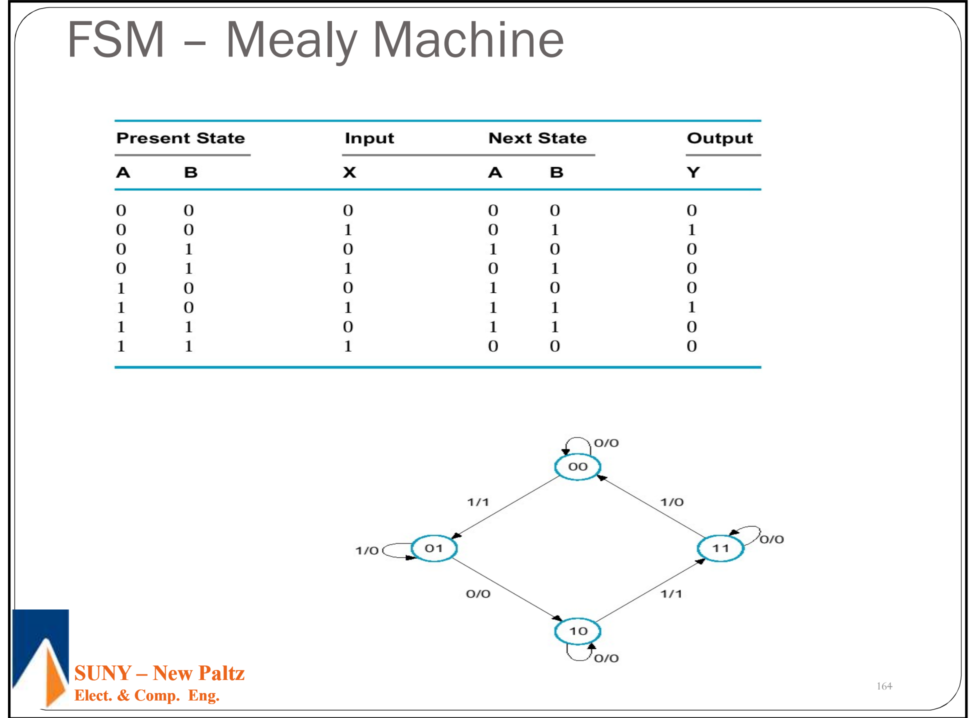

Design of an ALU using Case Statement
S | Function
0 | Clear
1 | B-A
2 | A-B
3 | A+B
4 | A XOR B
5 | A OR B
6 | A AND B
7 | Set to all 1's
// 74381 ALU
module alu(s, A, B, F);
input [2:0] s;
input [3:0] A, B;
output [3:0] F;
reg [3:0] F;
always @(*)
case (s)
0: F = 4'b0000;
1: F = B - A;
2: F = A - B;
3: F = A + B;
4: F = A ^ B;
5: F = A | B;
6: F = A & B;
7: F = 4'b1111;
endcase
endmodule
SUNY – New Paltz
Elect. & Comp. Eng.


Finite State Machines (FSM)
State diagrams are representations of Finite State Machines (FSM)
Mealy FSM
Output depends on input and state
Output is not synchronized with clock
can have temporarily unstable output
Moore FSM
Output depends only on state
Mealy FSM
Moore FSM
00/0
01/0
10/0
11/1
SUNY – New Paltz
Elect. & Comp. Eng.


Finite State Machines
Finite State Machines (FSMs) are a useful abstraction for sequential circuits with centralized "states" of operation
At each clock edge, combinational logic computes outputs and next state as a function of inputs and present state
inputs
+
present
state
Combinational
Logic
outputs
+
next
state
n
n
Q
Flip-
Flops
D
CLK
SUNY – New Paltz
Elect. & Comp. Eng.


Two Types of FSMs
Moore and Mealy FSMs : different output generation
Moore FSM:
inputs
x0...xn
Comb.
Logic
next
state
S+
n
D
Flip-
Flops
Q
CLK
n
present state S
Comb.
Logic
outputs
yk = fk(S)
Mealy FSM:
direct combinational path!
inputs
x0...xn
Comb.
Logic
S+
n
D
Flip-
Flops
Q
CLK
n
S
Comb.
Logic
outputs
yk = fk(S, x0...xn)
SUNY – New Paltz
Elect. & Comp. Eng.





Present State		Input	Next State		Output
A	B	X	A	B	Y
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	1	1	1
1	1	0	1	1	0
1	1	1	0	0	0

FSM – Mealy Machine

- Designing with DFF's
- Lets assume that following is given

$DA(A,B,X) = \Sigma\, m(2,4,5,6)$

$DB(A,B,X) = \Sigma\, m(1,3,5,6)$

$Y(A,B,X) = \Sigma\, m(1,5)$

165

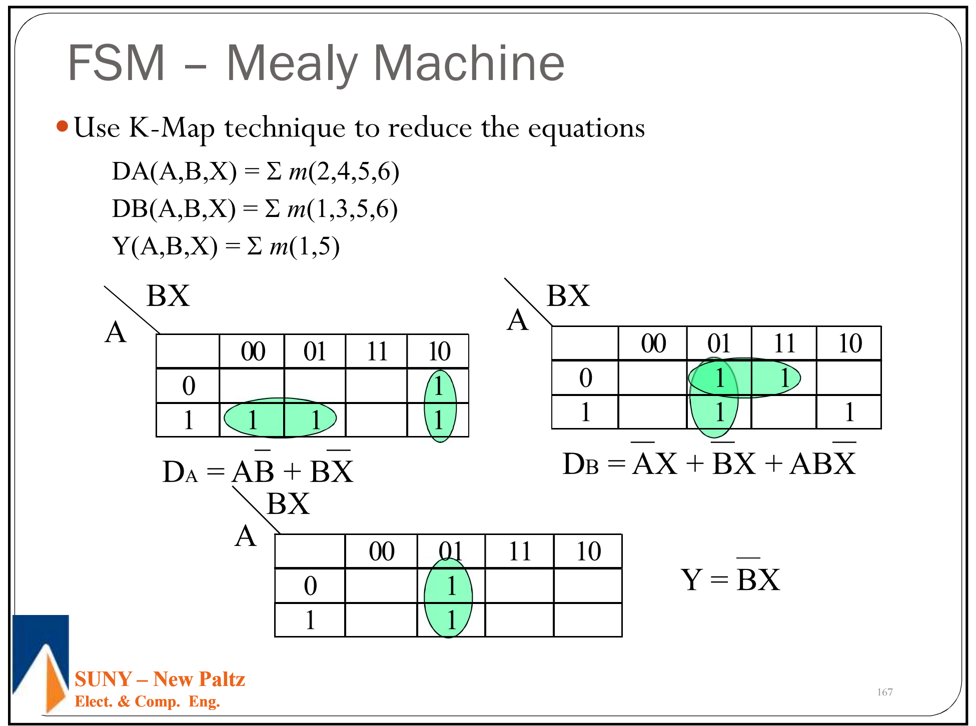
FSM – Mealy Machine

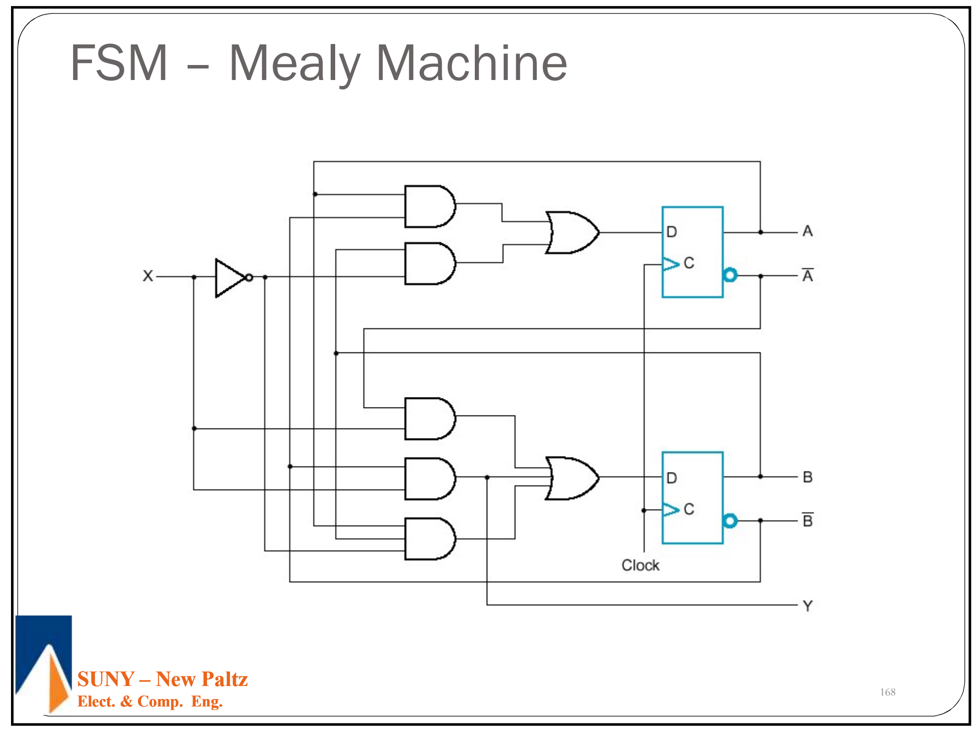
$DA(A,B,X) = \Sigma\, m(2,4,5,6)$
$DB(A,B,X) = \Sigma\, m(1,3,5,6)$
$Y(A,B,X) = \Sigma\, m(1,5)$

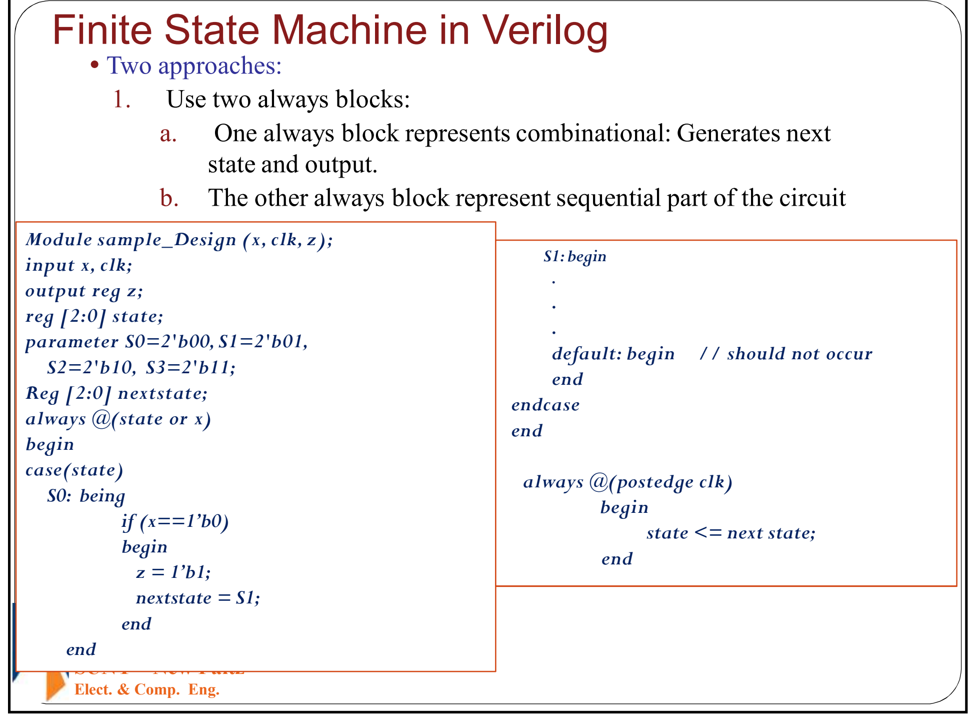
Present State		Input	Next State		Output
A	B	X	A	B	Y
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	1	1	1
1	1	0	1	1	0
1	1	1	0	0	0

SUNY – New Paltz
Elect. & Comp. Eng.

166


FSM – Mealy Machine
Use K-Map technique to reduce the equations
DA(A,B,X) = Σ m(2,4,5,6)
DB(A,B,X) = Σ m(1,3,5,6)
Y(A,B,X) = Σ m(1,5)
BX
A
00 01 11 10
0
1
DA = AB̄ + BX̄
DB = ĀX + B̄X + ABX̄
Y = B̄X
SUNY – New Paltz
Elect. & Comp. Eng.
167


FSM – Mealy Machine
X
D
C
A
Ā
B
B̄
Clock
Y
SUNY – New Paltz
Elect. & Comp. Eng.
168



Finite State Machine in Verilog

2. Use a single always block.

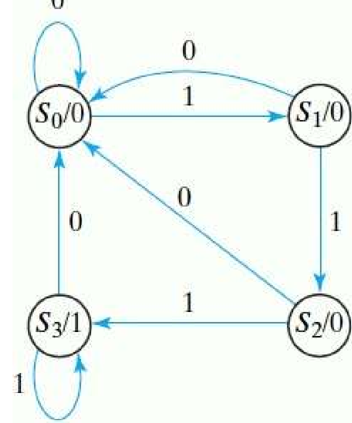
```
Module sample_Design (x, clk, z);
   input x, clk;
output reg z;
reg [2:0] state;
parameter S0=2'b00, S1=2'b01,
   S2=2'b10, S3=2'b11;
always @(postedge clk)
begin
case(state)
   S0: being
              if (x==1'b0)
                 state <= S1;
              else
                 statet <=S2;
       end
   S1: begin
.
```

```
      .
      .
      default: begin     // should not occur
      end
endcase
end
  Assign z = (state == S0 && x == 1'b0) || …

  .
  endmodule
```

SUNY – New Paltz
Elect. & Comp. Eng.

Sequence Detector

- Circuit specification:
  - Design a circuit that outputs a 1 when three consecutive 1's have been received as input and 0 otherwise.
- FSM type
  - Moore or Mealy FSM?
  - » Both possible
  - » Chose Moore to simplify diagram
- State diagram:
  - » State S0: zero 1s detected
  - » State S1: one 1 detected
  - » State S2: two 1s detected
  - » State S3: three 1s detected

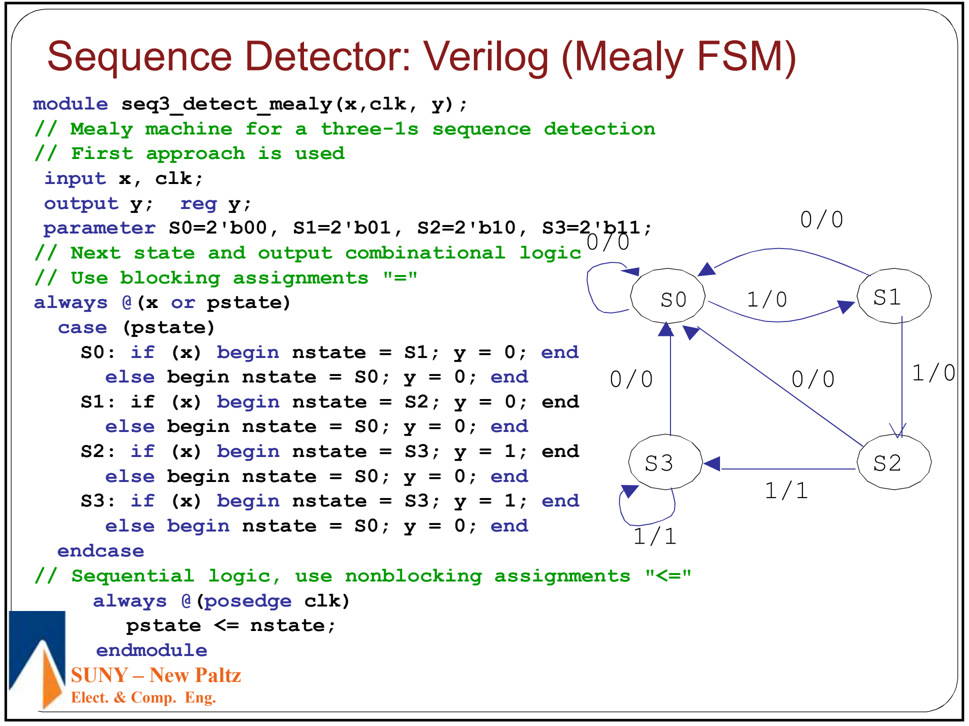


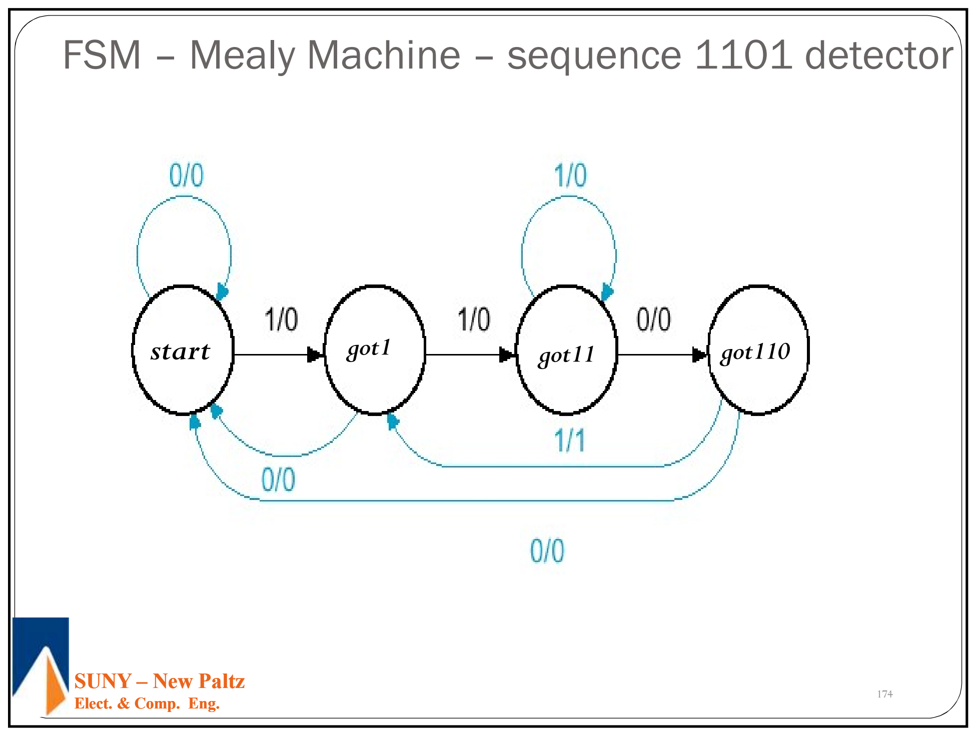
Sequence Detector: Verilog (Moore FSM)

```
module seq3_detect_moore(x,clk, y);
// Moore machine for a three-1s sequence detection
// second approach is used
  input x, clk;
  output y;
  reg [1:0] state;
  parameter S0=2'b00, S1=2'b01, S2=2'b1
// Define the sequential block
always @(posedge clk)
    case (state)
        S0: if (x) state <= S1;
            else    state <= S0;
        S1: if (x) state <= S2;
            else    state <= S0;
        S2: if (x) state <= S3;
            else    state <= S0;
        S3: if (x) state <= S3;
            else    state <= S0;
    endcase
// Define output during S3
assign y = (state == S3);
endmodule
```

$S_0/0$ $S_1/0$ $S_2/0$ $S_3/1$

SUNY – New Paltz
Elect. & Comp. Eng.


Sequence Detector: Verilog (Mealy FSM)
module seq3_detect_mealy(x,clk, y);
// Mealy machine for a three-1s sequence detection
// First approach is used
input x, clk;
output y; reg y;
parameter S0=2'b00, S1=2'b01, S2=2'b10, S3=2'b11;
// Next state and output combinational logic
// Use blocking assignments "="
always @(x or pstate)
case (pstate)
S0: if (x) begin nstate = S1; y = 0; end
else begin nstate = S0; y = 0; end
S1: if (x) begin nstate = S2; y = 0; end
else begin nstate = S0; y = 0; end
S2: if (x) begin nstate = S3; y = 1; end
else begin nstate = S0; y = 0; end
S3: if (x) begin nstate = S3; y = 1; end
else begin nstate = S0; y = 0; end
endcase
// Sequential logic, use nonblocking assignments "<="
always @(posedge clk)
pstate <= nstate;
endmodule
0/0
S0
1/0
S1
0/0
1/0
0/0
S3
S2
1/1
1/1
SUNY – New Paltz
Elect. & Comp. Eng.


FSM – Mealy Machine – sequence 1101 detector
0/0
1/0
start
1/0
got1
1/0
got11
0/0
got110
1/1
0/0
0/0
SUNY – New Paltz
Elect. & Comp. Eng.
174

FSM – Mealy Machine – sequence 1101 detector

```
module seq1101_mealy(y, x, RESET);
output y; reg y;
input x, RESET;
parameter start = 2'b00, got1 = 2'b01,
             got11 = 2'b11, got110 = 2'b10;
reg [1:0] Q, D;   // state, next state logic
reg CLK;
initial CLK <= 0;
always
  #10 CLK <= ~CLK;
always @(x or Q)
begin
 y <= 0;
 case(Q)
   start: D <= x ? got1  : start;
   got1:  D <= x ? got11 : start;
   got11: D <= x ? got11 : got110;
```

```
    got110 : if (x) begin
                 D <= got1; y <= 1;
                end
               else begin
                 D <= start; y <= 0;
               end
    default : D <= 2'bxx;
  endcase
end
always @(posedge CLK or negedge RESET)
begin
  if (~RESET)
    Q <= 0;
  else
    Q <= D;
end
endmodule
```

SUNY – New Paltz
Elect. & Comp. Eng.

175

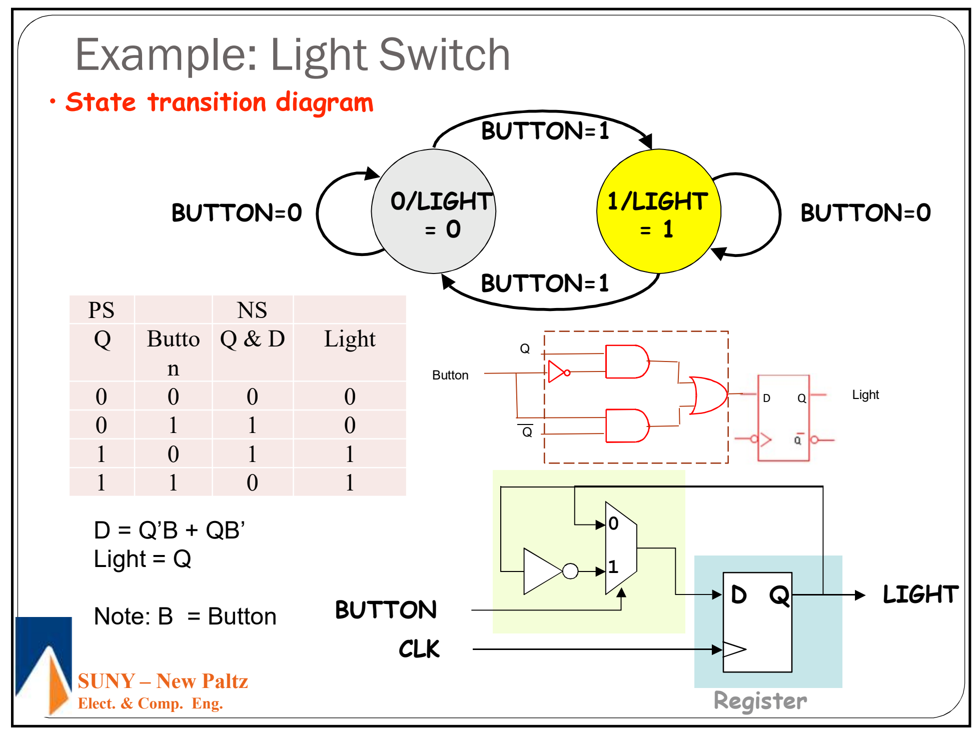
Design Examples

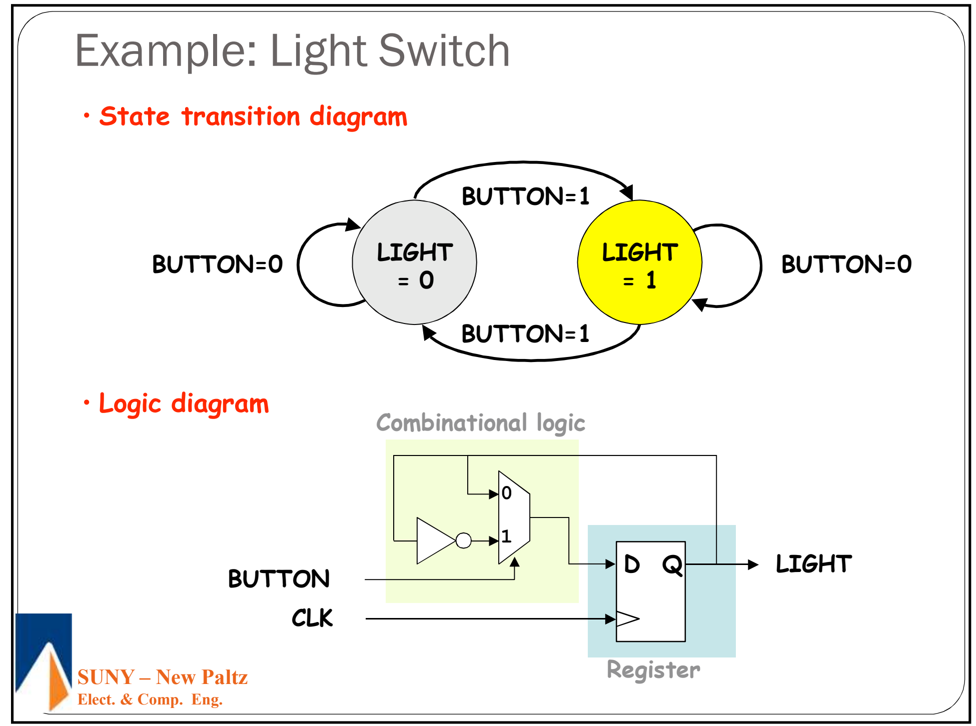
```
module seq1101_moore(y, x, CLK, RESET);
output y;
reg y;
input x, CLK, RESET;
parameter start = 3'b000, got1 = 3'b001,
got11 = 3'b011, got110 = 3'b010;
got1101 = 3'b110;
reg [2:0] Q;   // state variables
reg [2:0] D;   // next state logic output
// next state logic
always @(x or Q) begin
  case(Q)
    start: D = x ? got1   : start;
    got1:  D = x ? got11  : start;
    got11: D = x ? got11  : got110;
    got110: D = x ? got1101 : start;
```

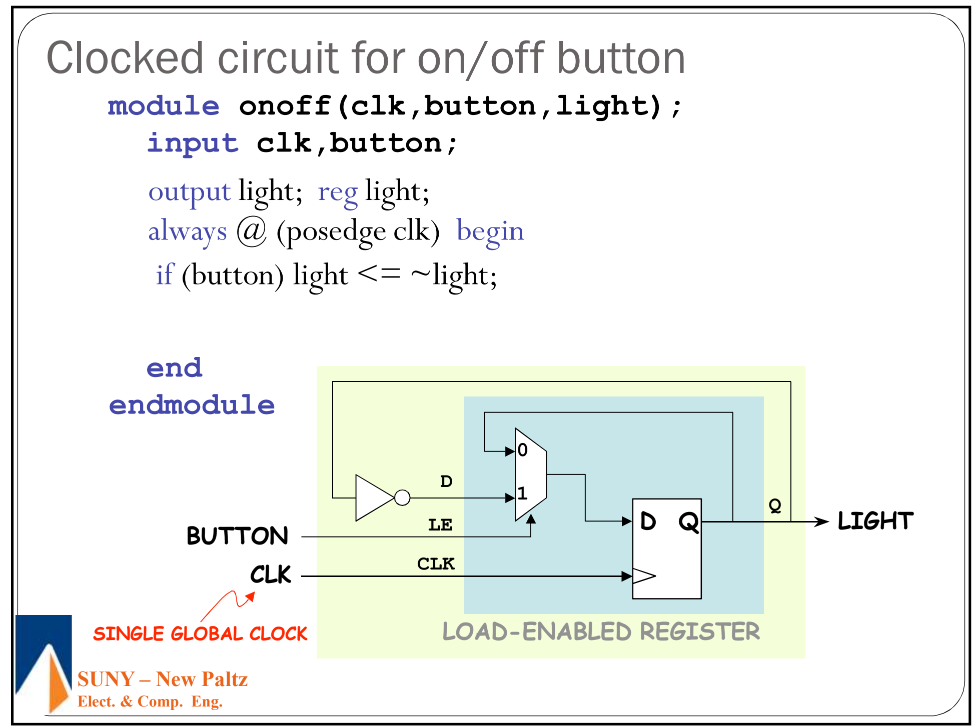
```
  got1101: D = x ? got11 : start;
  default : D = 3'bxxx;
 end
// state variables
always @(posedge CLK or
               negedge RESET)
begin
  if (~RESET)
    Q = 0;
  else
    Q = D;
end
// output logic
always @ (Q)
  y = Q[2];
endmodule
```

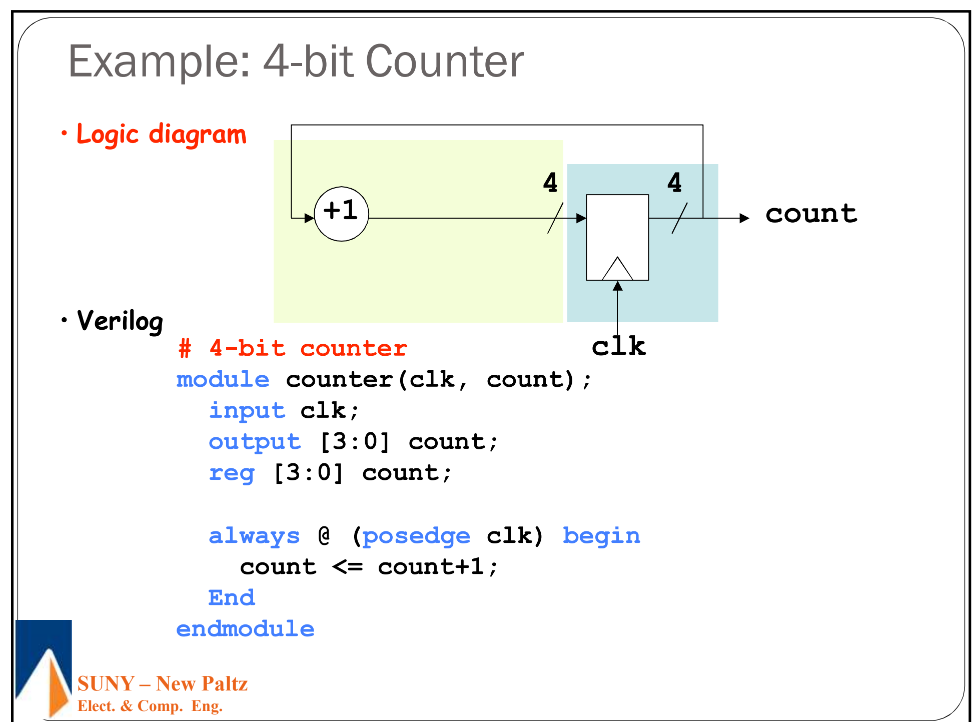
SUNY – New Paltz
Elect. & Comp. Eng.

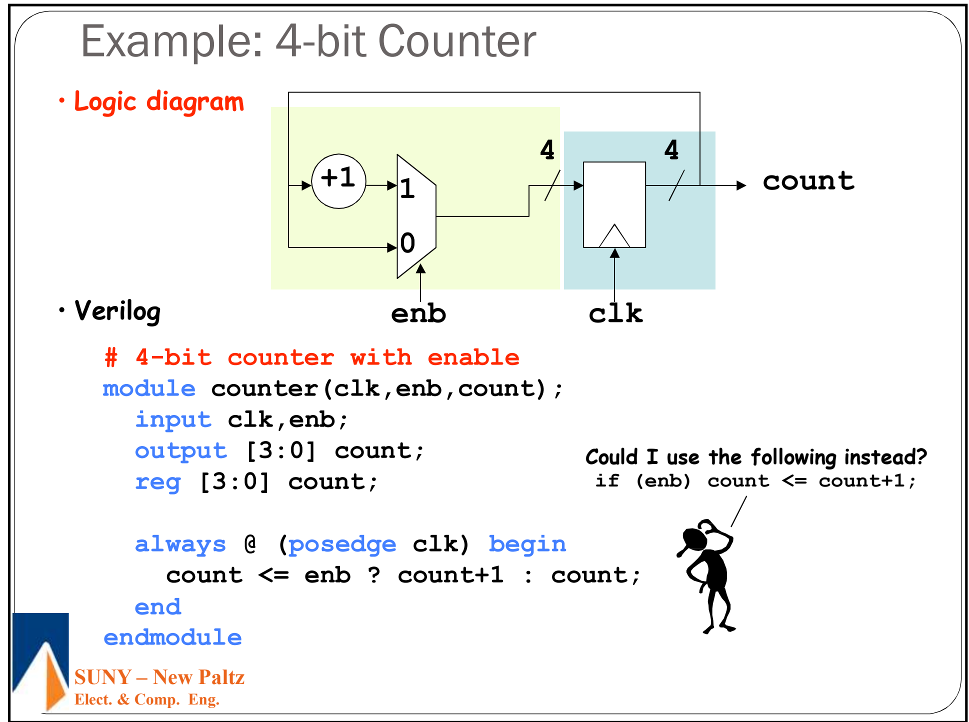
176

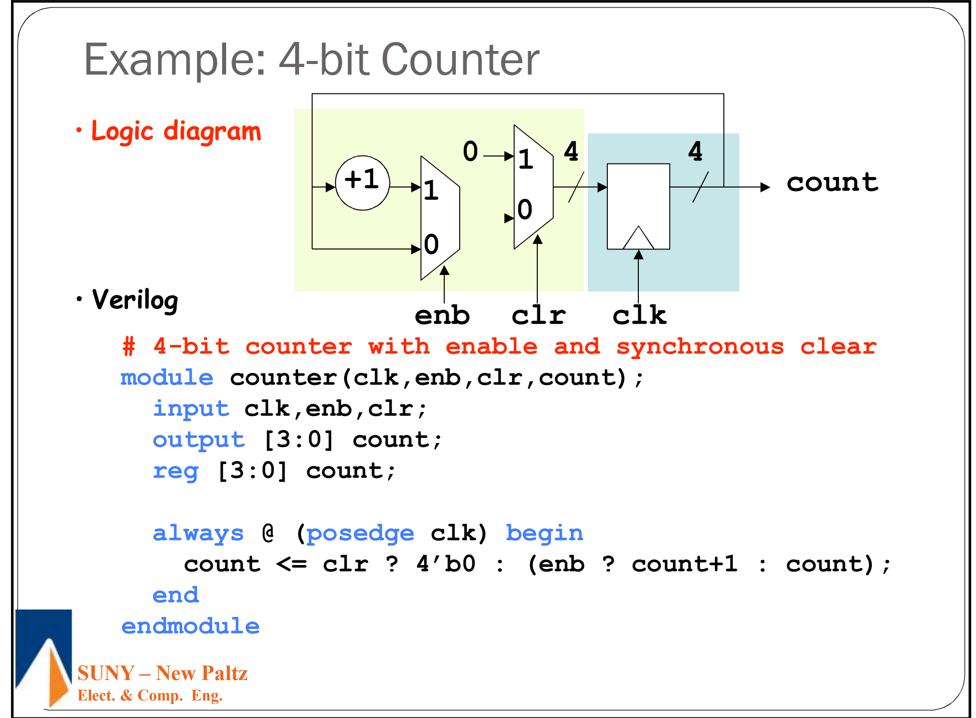

Example: Light Switch
• State transition diagram
BUTTON=1
BUTTON=0
0/LIGHT = 0
1/LIGHT = 1
BUTTON=0
BUTTON=1
PS Q
Button
NS Q & D
Light
0 0 0 0
0 1 1 0
1 0 1 1
1 1 0 1
D = Q'B + QB'
Light = Q
Note: B = Button
Button
Light
BUTTON
CLK
LIGHT
Register
SUNY – New Paltz
Elect. & Comp. Eng.


Example: Light Switch
• State transition diagram
BUTTON=1
BUTTON=0
LIGHT = 0
LIGHT = 1
BUTTON=0
BUTTON=1
• Logic diagram
Combinational logic
BUTTON
CLK
LIGHT
Register
SUNY – New Paltz
Elect. & Comp. Eng.


Clocked circuit for on/off button
module onoff(clk,button,light);
input clk,button;
output light; reg light;
always @ (posedge clk) begin
if (button) light <= ~light;
end
endmodule
D
0
1
LE
CLK
D Q
Q
LIGHT
BUTTON
CLK
SINGLE GLOBAL CLOCK
LOAD-ENABLED REGISTER
SUNY – New Paltz
Elect. & Comp. Eng.


Example: 4-bit Counter
• Logic diagram
+1
4
4
count
clk
• Verilog
# 4-bit counter
module counter(clk, count);
input clk;
output [3:0] count;
reg [3:0] count;
always @ (posedge clk) begin
count <= count+1;
End
endmodule
SUNY – New Paltz
Elect. & Comp. Eng.


Example: 4-bit Counter
• Logic diagram
+1
1
0
4
4
count
enb
clk
• Verilog
# 4-bit counter with enable
module counter(clk,enb,count);
input clk,enb;
output [3:0] count;
reg [3:0] count;
always @ (posedge clk) begin
count <= enb ? count+1 : count;
end
endmodule
Could I use the following instead?
if (enb) count <= count+1;
SUNY – New Paltz
Elect. & Comp. Eng.


Example: 4-bit Counter
• Logic diagram
+1
1
0
0
1
0
4
4
count
enb
clr
clk
• Verilog
# 4-bit counter with enable and synchronous clear
module counter(clk,enb,clr,count);
input clk,enb,clr;
output [3:0] count;
reg [3:0] count;
always @ (posedge clk) begin
count <= clr ? 4'b0 : (enb ? count+1 : count);
end
endmodule
SUNY – New Paltz
Elect. & Comp. Eng.

4-bit Shift Register with Reset

```
module srg_4_r_v (CLK, RESET, SI, Q,SO);
input CLK, RESET, SI;
output [3:0] Q;
output SO;
reg [3:0] Q;
assign SO = Q[3];
always@(posedge CLK or posedge RESET)  begin
 if (RESET)
   Q <= 4'b0000;
 else
   Q <= {Q[2:0], SI};
   end
endmodule
```

SUNY – New Paltz
Elect. & Comp. Eng.

4-bit Binary Counter with Reset

```
module count_4_r_v (CLK, RESET, EN, Q, CO);
input CLK, RESET, EN;
output [3:0] Q;
output CO;
reg [3:0] Q;
assign CO = (count == 4'b1111 && EN == 1'b1) ? 1 : 0;
always@(posedge CLK or posedge RESET)
  begin
  if (RESET)
   Q <= 4'b0000;
  else if (EN)
   Q <= Q + 4'b0001;
  end
endmodule
```

SUNY – New Paltz
Elect. & Comp. Eng.

Advanced features in Verilog

- Session V

185

User Defined Primitives

- Verilog has built-in gates, transmission gates and switches
- They are Verilog provided built in primitives
- This is a rather small number of primitives
- If we need more such simple or complex primitives then Verilog provides facility for writing them as UDP's or simply User Defined Primitives
- The UDP's are self contained and are instantiated like gate level primitives
- One can write combinational as well as sequential UDP's
- UDPs cannot be defined inside the modules i.e. can only be instantiated inside the module
- One of their applications is to model primitives of ASIC libraries
- UDP's begin with keyword **primitive** and end with **endprimitive**
- Ports declaration follow primitive keyword just like module

SUNY – New Paltz
Elect. & Comp. Eng.

186

User Defined Primitives – when to use UDP's

- UDPs description is technology independent
- UDPs cannot model timing parameters, hence functionalities that need to model timing parameters should be modeled as module
- UDP is a lookup table, hence as the number of inputs increase table entries grows exponentially which in turn increases the memory requirement, therefore do not design UDP's with large number of inputs
- UDP state table should be specified as completely as possible because if certain combination of inputs is not specified, the default output for that combination will be 'x'

```
primitive udp_syntax (a, b, c, d);
output a;
input b,c,d;
// UDP function code here
endprimitive
```

User Defined Primitives – some rules to follow

- UDPs can take only scalar input terminals (1 bit)
- Multiple inputs are permitted
- Can have only 1-bit scalar output
- Output terminal must appear first in terminal list
- Multiple output terminals are not allowed
- Inputs/Outputs are declared with keyword **input, output**
- In sequential UDPs output is declared as **reg**
- Do not support inout ports
- The state in a sequential UDP can be initialized with an initial statement otherwise it is optional
- State table defines the state of output under different input conditions
- State table entries can contain values '0' , '1' or 'x'
- 'z' values passed to UDP are treated as x

SUNY – New Paltz
Elect. & Comp. Eng.

188

User Defined Primitives – Symbols

- Symbols used for level, edge transition specification in UDP table

Symbol	Meaning	Explanation
?	0 or 1 or x	Cannot be specified in an output field
b	0 or 1	Cannot be specified in an output field
-	no change in state value	Can be specified only in an output field of a sequential UDP
r	(01)	Rising edge of an input signal
f	(10)	Falling edge of an input signal
p	(01) or (0x) or (x1) or (1z) or (z1)	Potential rising edge of a signal
n	(10) or (1x) or (x0) or (0z) or (z0)	Potential falling edge of a signal
*	(??)	Any value change in signal

SUNY – New Paltz
Elect. & Comp. Eng.

189

User Defined Primitives – example combinational

```
primitive mux_21_udp(out, sel, i0, i1);
output out;
input sel, i0, i1;
table
  // sel i0 i1   out
     0  0  ?  :0 ; // 1
     0  1  ?  :1 ; // 2
     1  ?  0  :0 ; // 3
     1  ?  1  :1 ; // 4
     ?  0  0  :0 ; // 5
     ?  1  1  :1 ; // 6
endtable
endprimitive
```

SUNY – New Paltz
Elect. & Comp. Eng.

190

User Defined Primitives – example sequential

```
// Latch with active low clock
primitive latch_udp(q, clock, data) ;
output q; reg q ;
input clock, data;
table
  // clock data   q   q+
     0    1    :? : 1 ;
     0    0    :? : 0 ;
     1    ?    :? : - ; // - = no change
endtable
endprimitive
```

SUNY – New Paltz
Elect. & Comp. Eng.

191

User Defined Primitives – example sequential

```
// Flip flop with rising clock
primitive dff_udp (q, clk, d);
  output q; reg q;
  input clk, d;
  table
    // clk d : q : q+
      r  0 : ? : 0 ;
      r  1 : ? : 1 ;
      f  ? : ? : - ;
      ?  * : ? : - ;
  endtable
endprimitive
```

SUNY – New Paltz
Elect. & Comp. Eng.

192

User Defined Primitives – example sequential

```
// Flip flop with rising clock and reset
primitive dff_reset_udp (q, d, clk, rst);
   output q; reg q;
   input clk, rst, d;
   initial q = 0;    // powers up in reset state
   table
   // d  clk  rst  :  q    : q+
      ?   ?    0   :  ?    : 0;
      0   r    1   :  ?    : 0;
      1   r    1   :  ?    : 1;
      ?   n    1   :  ?    : -;
      *   ?    1   :  ?    : -;
      ?   ?    p   :  ?    : -;
   endtable
endprimitive
```

SUNY – New Paltz
Elect. & Comp. Eng.

193