These notes are copyrighted and are strictly for 2017-2018 courses at San Diego State University (SDSU).

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means.

02 – Verilog-I

Digital systems

- Digital systems are highly complex
- At their most detailed level, they may consist of millions of logic gates or billions of transistors
- From a more abstract viewpoint, these elements may be grouped into a number of functional components, such as arithmetic logic units, memories, and control units
- Translating block diagrams into circuit schematics is time-consuming and prone to error
- *Hardware description languages* (HDLs) have evolved to aid in the design of systems with this large number of elements and wide range of abstractions
- *Separating behavior from implementation*: A HDL allows design and debugging at a higher level of *abstraction* without detailed descriptions (compared to gates or transistors layouts) before conversion to the gate and transistor level
  - For example, a 32-bit multiplier schematic is a complicated structure
  - The designer must choose what type of multiplier architecture to use. In contrast, the multiplier can be specified with one line of HDL code

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Hardware description languages

- Describes the exact behavior of all the *components* and their *interfaces*
- Some other reasons for using HDLs instead of schematic capture
  - Greatly improves designer *productivity*
  - Improves *quality*: more time can be spent on logic verification and optimization rather than on the detailed gate-level design
  - *Design reuse*: A more concise and readable design makes it easier for the design to be reused by others
  - *Earlier design decisions*: Allows making decisions about cost, performance, power, and area earlier in the design process
- A HDL is not a software programming language
  - Software programming language can be translated into machine instructions and then executed on a computer
  - When coding in an HDL, it is important to remember that you are specifying hardware that operates in *parallel* rather than software that executes in *sequence*

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

VHDL and Verilog

- HDL is used extensively by industry and academia for modeling, implementation and verification of digital circuits
- The two most popular HDLs are *Verilog* and *VHDL*
  - They were originally intended for documentation and simulation, but are now used to synthesize gates directly from the HDL
- VHDL, which stands for VHSIC Hardware Description Language, where VHSIC in turn was a Department of Defense project on Very High Speed Integrated Circuits, was developed by committee under government sponsorship
- Verilog is less verbose and closer in syntax to C, while VHDL supports some abstractions useful for large team projects
  - May be a disadvantage as the syntax can cause beginners to assume C semantics
- Many Silicon Valley companies use Verilog while defense and telecommunications companies often use VHDL

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

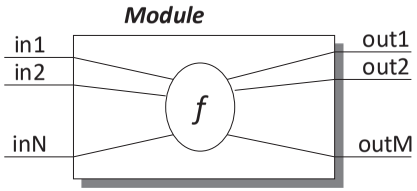
Verilog

- Verilog was developed by Advanced Integrated Design Systems (later renamed Gateway Design Automation) in 1984 as a proprietary language for logic simulation
- Gateway was acquired by Cadence in 1989 and Verilog was made an open standard in 1990
- The language became an IEEE standard in 1995 and was updated in 2001
- In 2005, it was updated again with minor clarifications; more importantly, *SystemVerilog* was introduced, which resolve some of the Verilog's ambiguity and adds high-level programming language features that have proven useful in *verification*
- Verilog is a powerful hardware description language, which provides a mechanism for system description, modeling and documentation, simulation, synthesis (implementation), testing (fault simulation, test generation) and verification

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Verilog module

- The Verilog language describes a digital system as a set of *modules*
- A module is the the basic unit of digital logic in Verilog
- Each of these modules has an interface to other modules as well as a description of its contents




```
module module_name ( port declarations);
   local variable declaration;
   functional description of f
endmodule
```

- A module definition starts with the keyword **`module`** and ends with the keyword **`endmodule`**
- **`module_name`** is a user-defined name for the model
- The port declaration defines the module's external interface
- Modules may contain local signal declarations
- The module description shows the basic *definition* of a module
- Each statement in a Verilog module ends with a semicolon, except **`endmodule`** and **`end`**

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

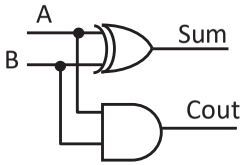
Gate-level modeling

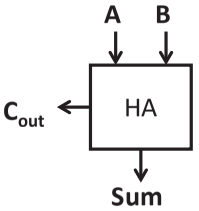
- A module represents a digital unit that can be described *at various levels of abstraction*
- For example, at the *gate-level,* we can describe a half-adder as follows:

A	B	Sum	Cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Sum = A ⊕ B=AB'+A'B

Cout = A.B






```
module halfAdder_gate (input A, B,
                       output Sum,Cout);
  xor U00 (Sum, A, B);   // instantiating primitive gates
                         // The output is always the first port in the port list
  and U01 (Cout, A, B); // all statements are executed concurrently
endmodule
```

- In Verilog, a primitive gate may be explicitly instantiated by using the primitive gates

```
gate_type [delay] instance_name (<output>, <inputs>)
```

- Verilog supports a set of primitive logic gates:
  `and,nand,or,nor,xor,xnor,not,buf`

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Verilog syntax

- Identifier
  - A ... Z, a ... Z, 0 ... 9, underscore. First char of identifier must not be a digit
  - An _ (underscore) is ignored (used to enhance readability)
- Note that Verilog is *case-sensitive*. For example, y1 and Y1 are different signals in Verilog. However, using separate signals within a module that only differ in their capitalization is confusing
- Verilog comments are just like those in C
  - Comments beginning with /* continue, possibly across multiple lines, to the next */
  - Comments beginning with // continue to the end of the line
  - Multi-line comments may not be nested. However, there may be single line comments inside a multi-line comment
- Important: Use comments to explain ports, signals, variables, or group of signals or variables, modules, functions, etc.

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Four-valued system

- Verilog supports four logic values:
  - 0: Logic-0 or FALSE
  - 1: Logic-1 or TRUE
  - x: Undefined, unknown, or don't care
  - z: High impedance, floating, unconnected (no current flowing)
- When a signal is driven to different values (**0**, **1**, or **x**) by multiple drivers simultaneously, it is in contention (**x**)
  - Hence an **x** can indicate a **0**, **1**, **z**, or in transition
- Verilog uses **x** to indicate an invalid logic level and **z** is useful for describing a tristate buffer, whose output floats when the enable is **0**
- The **x** may randomly be interpreted by the circuit as **0** or **1**, leading to unpredictable behavior
- Thus seeing **x** values in simulation is almost always an indication of a bug or bad coding practice
- Nevertheless, we can set bits to be **x** in situations where we don't care what the value is. This can help catch bugs and improve synthesis quality

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Signal data types - reg type

- Ports and the local or internal variables of a module are informally called *signals*
- A signal can belong to one of the two data types: *register* or *net*
- Register type in Verilog (i.e., "**reg**") should not be confused with hardware registers
  - While the term 'register' implies a hardware register, the name is used in Verilog to indicate a software register (i.e. a variable)
- The type **reg** simply means a variable that can hold a value (an abstract storage element) until a new value is assigned to it
- A register declaration always starts with the keyword **reg** (e.g., `reg z;`)
- The default value of register variables is '**x**'

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Signal data types - net type

- Nets are used to model an electrical connection (physical wire) in a circuit between structural entities, such as gates (gate-level modeling) and between modules (structural modeling)
- *Internal variables* are only used internal to the module and are neither inputs nor outputs
  - They are similar to local variables in programming languages
- Since nets are internal signals, they cannot be accessed by outside environment
- A **wire** is one type of net
- A net declaration always starts with the keyword **wire** (e.g., `wire x,y;`)
- Initial values may be specified in declarations: `wire wire_variable = value;`
- A **wire** does not store its value but must be driven continuously by its driver (e.g. **x = y;**)
  - A net driver can be a gate's output, output of a module, or a continuous assignment (**assign** statement)
- The default value of net types is **z**, i.e., if a net is not driven by any driver, its value is **z**

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Net declaration

- Nets must be declared before being used, except for scalar wires, which is a source of errors
- Recommendation: Declare all nets *explicitly* at the top of each module, even when an implicit declaration could be made. This improves the readability and maintainability of the Verilog code
  - Verilog-2001 provides `` `default_nettype none `` to disable default net declarations. In this case, any undeclared signals will be a syntax error, which prevents hard-to-debug wiring errors due to a mistyped name

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Vectors

- Signals can be either scalar (one bit) or vector (multiple bits)
- Vectors are represented using square brackets and the format is [index1:index2]
- A type declaration can be written as:

```
reg|wire [index1 : index2] List_of_Variables;
```

- We normally use the little endian convention index1>index2, where index1 indicates the most significant bit (MSB) and index2 indicates the least significant bit (LSB)
- Example: `wire [15:0] x; //little endian convention; 16-bit signal; x[15] MSB`
- MSB can be a smaller index than LSB

```
e.g.  Reg [0:3] x,y; //big endian convention;  //x[0] is the MSB
```

- MSB:LSB can be any integer (negative numbers too)

```
wire [15:12] addr;     //MSB:LSB may be any integer
```

- The port size can also be a range defined as **[MSB : LSB]**

```
input a,b,sel;              //3 scalar ports
output [7:0] result;        //little endian convention
```

- The default port/signal width is one bit

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Bit-selects and part-selects

- *Bit select*: selection of an individual bit: **variable_name[index]**
- *Part select*: selection of a group of bits: **variable_name[msb:lsb]**
  - Bit-selects and part-selects can be used as operands in expressions
- Part selects must address a more significant bit on the left of the colon
- Verilog-2001 supports *variable part selects* by allowing to use variables to select a group of bits from a vector

```
[base_expr +: width_expr] //positive offset
[base_expr -: width_expr] //negative offset
```

- The starting point of the part-select (**base_expr**) can be a variable
- The width of the part-select (**width_expr**) must be constant
- The offset direction indicates if the width expression is added to or subtracted from the base expression

```
reg  [63:0] word;
reg  [3:0] byte_num; //a value from 0 to 15
wire [7:0] byteN = word[byte_num*8 +: 8];
```

- In the preceding example, if **byte_num** has a value of 4, then the value of word[39:32] is assigned to **byteN**. Bit 32 of the part select is derived from the base expression, and bit 39 from the positive offset and width expression

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Ports definition

- Inputs and outputs of a module (i.e., the ports) model the signals or pins of hardware components to interface with other modules or components
- All the ports in a list of port definitions must be specified

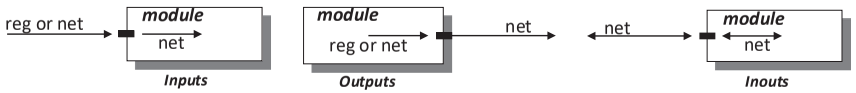
```
port_direction [port_type] [port_size] port_name1, port_name2,... ;
```

- Each port must be declared as an **input**, **output** or **inout**, which defines the *port direction*
  - An **input** port specifies the internal name for a vector or scalar that is driven by an external entity
  - An **output** port specifies the internal name for a vector or scalar which is driven by an internal entity and is available external to the module
  - An **inout** port specifies the internal name for a vector or scalar that can be driven either by an internal or external entity
- The *port type* defines the type of the port, such as **wire** or **reg**
  - The default type for port signals is **wire**

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Ports

- Ports are signals to interface with outside environment
  - **input** for input ports, can be read but cannot be written
  - **output** for output ports can be written but cannot be read
  - **inout** for bi-directional ports can be read and written
- Port connection rules:
  - **inputs** : internally must always be of type net, externally the inputs can be connected to a variable of type **reg** or net
  - **outputs** : internally can be of type net or **reg**, externally the outputs must be connected to a variable of type net
  - **inouts** : internally or externally must always be type net, can only be connected to a variable net type

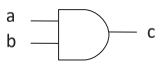


- An **input** or **inout** port cannot be declared to be of type **reg**
- An **inout** port may only be driven through a gate with high impedance capabilities (such as a **bufif0** gate)

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Logic operations

- Verilog has built-in logical operations
- Logic gates are inferred by the use of their corresponding operators
- The operations are defined for four-valued signals
- Example: The following truth table shows a truth table for an AND gate using all four possible signal values



&	1	0	X	Z
1	1	0	X	X
0	0	0	0	0
X	X	0	X	X
Z	X	0	X	X

– Note:
1 & z = x
1 & x = x
z & x = x

- Note that the gate can sometimes determine the output despite some inputs being unknown
  - For example 0 & z returns 0 because the output of an AND gate is always 0 if either input is 0. Otherwise, floating or invalid inputs cause invalid outputs, displayed as x
- An **x** output may correspond to a floating gate input (**z**) or uninitialized input (**x**) when the gate can't determine the correct output value

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Bitwise and reduction operators

- *Bitwise operators*: act on single-bit signals or on multi-bit busses and do a bit-by-bit comparison between two operands
  - **and &, or |, not ~, xor ^, xnor ~^ (or ^~)**

Operand A	Operand B	A&B	A|B	~A	A^B	A~^B
1010	0011	0010	1011	0101	1001	0110

- *Reduction operators*: operate on all the bits of an operand vector and return a single-bit value as output
- These are the unary (one argument) form of the bit-wise operators above
  - **and &, or |, xor ^, nand ~&, nor ~|, xnor ~^ (or ^~)**

Operand A	&A	~&A	|A	~|A	^A	~^A
1010	0	1	1	0	0	1

- Unary Reduction operators imply a multiple-input gate acting on a single bus

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

(2) Dataflow level modeling

- Three general styles for describing module functionality are *dataflow*, *behavioral* and *structural*
- Describing the behavior of a module using *continuous signal assignment* statements is called *dataflow modeling*
- A continuous signal assignment statement starts with the **assign** keyword
- The general form of the **assign** statement is:

```
assign [delay] List_of_Net_assignments
```

where the `List_of_Net_assignments` are in the form

```
net = expression {,net = expression};
```

- An *expression* consists of a set of operands, and one or more operators (logical, numerical, relational), literal values, and sub-expressions
- A complete command such as **`assign Sum=A^B;`** is called a *statement*
- Continuous **assign** statement specifies a value to be driven onto a net
  - The expression on the RHS of an **assign** statement may contain both "register" or "net" type variables and the LHS must be a net-type signal (i.e., a port or wire)
  - If undeclared, the left-hand side is implicitly declared as a scalar (one bit) net

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

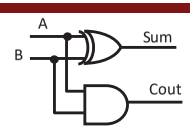
Combinational logic description using dataflow level modeling

- The continuous assignment is always active (driving a **0**, **1**, **x**, or **z**), regardless of any state sequence in the circuit
- Any time the operand or operands on the right hand side (RHS) expression of an **assign** statement change, the output value is assigned to the left hand side (LHS) net
  - Thus the RHS expression is continuously evaluated as a function of changing inputs
- If any input to the **assign** statement changes at any time, the assign statement will be reevaluated and the output will be propagated
  - This is a characteristic of combinational logic
  - Thus continuous assignments provide a means to abstractly model combinational hardware driving values onto nets
  - Use continuous assignments to describe combinational logic that can easily be described using a straightforward expression
- A Verilog module can contain any number of continuous assignment statements
- All continuous signal assignment statements execute *concurrently*
  - *Thus concurrent assignment statements can be written in any order*
  - This is different from conventional programming languages like C or Java in which statements are evaluated in the order they are written

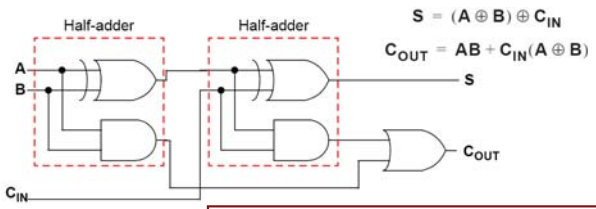
These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Dataflow level modeling

```
module halfAdder_dataflow (input A, B,
                           output Sum, Cout);
  assign Sum  = A ^ B;  //^ denotes logical XOR
  assign Cout = A & B;  //& denotes logical AND

endmodule
```



Inputs		Carry-in	Sum	Carry-out
A	B	C_{IN}	S	C_{OUT}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



$$S = (A \oplus B) \oplus C_{IN}$$

$$C_{OUT} = AB + C_{IN}(A \oplus B)$$

```
module fullAdder(input a,b,cin,
                 output sum,cout);

  assign cout = a & b | cin & (a ^ b);
  assign sum = cin ^ a ^ b;
endmodule
```

- Often, it is convenient to break a complex function into intermediate steps

```
module fullAdder(input a, b, cin,
                 output s, cout);
wire p, g;
      assign p = a ^ b;
      assign g = a & b;
      assign s = p ^ cin;
      assign cout = g | (p & cin); // do we need parenthesis here?
                                   // No, and has higher precedence than or
endmodule
```

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Examples: N-input gates examples

```
module or8(input [7:0] a,
           output y);
assign y = |a;  // using the reduction operator
// |a is much easier to write than
// assign y = a[7] | a[6] | a[5] | a[4] | a[3] | a[2] | a[1] |  a[0];
endmodule
```

```
// 3-input gates
module gates3b (input wire [2:0] x ,
                output wire and3 ,
                output wire nor3 ,
                output wire xor3);
   assign and3 = &x,
          nor3 = ~|x, // reduction NOR
          xor3 = ^x;
endmodule
```

```
assign z = x[1] & x[2] & ... & x[n]; //n-input and gate
assign z = &x; //it is much easier to describe an n-input and gate
and A00 (z,x[1],x[2],...,x[n]); //n-input and gate

assign z = ~(x[1] & x[2] & ... & x[n]); //n-input nand gate
assign z = ~&x; //n-input nand gate
nand N00 (z,x[1],x[2],...,x[n]); //n-input nand gate

assign z = ~(x[1] ^ x[2] ^ ... ^ x[n]); //n-input xnor gate
assign z = ~^x; //n-input xnor gate
xnor X00 (z,x[1],x[2],...,x[n]); //n-input xnor gate
```

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Examples

```
module gates (input [3:0] a, b,
              output [3:0] y1, y2, y3, y4, y5);
   /* Five different two-input logic gates
   acting on 4 bit busses */
   assign y1 = a & b;    // AND
   assign y2 = a | b;    // OR
   assign y3 = a ^ b;    // XOR
   assign y4 = ~(a & b); // NAND
   assign y5 = ~(a | b); // NOR
endmodule
```

```
module gates2  (input a, b,
                output and_,
                output nand_,
                output nor_,
                output or_,
                output xnor_,
                output xor_) ;
   assign and_ = b & a,
          nand_ = ~(b & a),
          or_ = b | a,
          nor_ = ~(b | a),
          xor_ = b ^ a,
          xnor_ = ~(b ^ a);
endmodule
```

```
module generate_mux (input [0:7] data,
                     input [0:2] select,
                     output out);
  assign out = data[select];
endmodule
```

Non-constant index in expression on RHS generates a MUX

```
module generate_decoder (input in,
                         input [0:1] select,
                         output [0:3] out);
  assign out[select] = in;
endmodule
```

Non-constant index in expression on LHS generates a decoder/demux

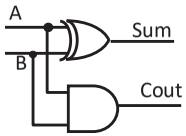
These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

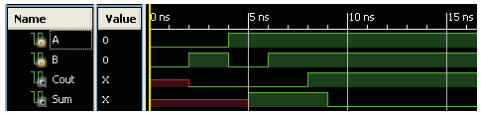
Delay in assign statements

- The default delay is zero
- In this example, the delay specifies the delay involved in the exclusive-or, not in the wire drivers

```
module modXor (output [7:0] AXorB,
               input [7:0] a, b);
   assign #5 AXorB = a ^ b;
endmodule
```

```
module HalfAdder (input A, B,
                  output Sum, Cout);
  assign #2 Cout = A & B; //& denotes logical AND
  //after 2 time units assign A&B to Cout
  assign #3 Sum = A ^ B; //^ denotes logical XOR
  //after 5 time units assign A^B to Sum
endmodule
```





- Verilog synthesizer ignores the delays specified in a procedural assignment statement
- May lead to functional mismatch between the design model and the synthesized netlist

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Delay in net declaration

- If we declare the wire and exclusive-or separately, we could assign a separate delay to the wire driver

```
wire [7:0] #10 AXorB;
assign #5 AXorB = a ^ b;
```

- When a delay is given in a net declaration, the delay is added to any driver that drives the net
- So if **a** or **b** changes, **AXorB** receives the result after 15 time units
- The combined use of a net specification and continuous assign is formally specified with the following descriptions of a net_declaration:

```
net_declaration := | net_type [drive strength] [ vectored | scalared ] [signed] [delay]
```

- In this example, we have defined a **wand** net with delay of 10
  - Two assign statements drive the net
  - One assign statement has delay 5 and the other has delay 3
  - When input a changes, there will be a delay of fifteen before its change is reflected at the inputs that c connects to
  - When input b changes, there will be a delay of thirteen

```
module wandOfAssigns (input a, b,
                      output c);
  wand #10 c;
  assign #5 c = ~a;
  assign #3 c = ~b;
endmodule
```

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Logical operators

- Verilog supports three logical operators: and **&&**, or **||**, not **!**
- Logical operators are typically used in conditional (**if ... else**) statements since they work with expressions
  - e.g., `if ((x==y) && z) a=1; //if x equals y and z is non-zero`
- They can work on expressions, integers or groups of bits
- The logical operators treat their operands as Boolean quantities
  - A non-zero operand is considered true (**1'b1**)
  - A zero operand is considered false (**1'b0**)
  - An ambiguous value (i.e. one that could be true or false, such as **4'bXX00**) is unknown (**1'bX**)
- They return a one-bit result **1** (true), **0** (false), or **x**

Operand A	Operand B	A&&B	A||B	!A	!B
1010	00	0	1	0	1
1010	011	1	1	0	0

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Numbers

- Specifying a constant value may be specified in either the *sized* or the *un-sized* form
- Syntax for sized constant values: **<size>'<base><value>**
  - **size** *(optional)* is the exact *number of bits* to represent the number
  - **'base** *(optional)* represents the radix. The default base is decimal
  - <**value**> defines the value
- Numbers can be specified in a variety of bases

Base	*Symbol*	*Legal Values*
binary	**b** or **B**	0, 1, x, X, z, Z, ?, _
octal	**o** or **O**	0-7, x, X, z, Z, ?, _
decimal	**d** or **D**	0-9, _
hexadecimal	**h** or **H**	0-9, a-f, A-F, x, X, z, Z, ?, _

- Note that the base letters, hexadecimal digits, **X** and **Z** are not case sensitive

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Truncation and padding

- If the size is less than the number of bits specified, the number is truncated from the left (the upper bits)
- If the size is greater than the number of bits specified, the number is padded on the left with **0**s
- However, constants starting with **Z** or **X** are padded with leading **Z**s or **X**s to reach their full length when necessary
  - Verilog automatically extend a logic **Z** or **X** to the full width of the left-hand side

```
data = 'bz; //data will be 'hzzzzzzzzzzzzzzzz
```

- If the size is not given, the number is assumed to have as many bits as the expression in which it is being used
  - It is better practice to explicitly give the size
- The characters **Z** and **?** are equivalent in numbers

10	unsized	decimal	0...01010 **(32-bits)**
'o7	unsized	octal	0...00111 **(32-bits)**
1'b1	1 bit	binary	1
8'hc5	8 bits	hex	11000101
6'hF0	6 bits	hex	110000 **(truncated)**
6'hF	6 bits	hex	001111 **(zero filled)**
6'hZ	6 bits	hex	ZZZZZZ **(Z filled)**

These notes

More examples

```
-253          // A signed decimal number
'haf          // An unsized hex number
'h12          // hex number 12 (18 decimal number)
'o12          // octal number 12 (10 decimal number)
'b1001        // binary number 1001 (9 decimal number)
8'b10010011   // 8-bit binary number
8'h12         // hex number 12 taking 8 bits
6'o67         // A 6 bit octal number
8'd12         // decimal number 12 taking 8 bits
8'b1010_0011  // 10100011
8'b1          // binary number 00000001 (zero filled)
16'hf1        // 0000000011110001 (zero filled)
8'bx          // An 8 bit unknown number (8'bxxxx_xxxx)
4'b100Z       // 100Z
4'bz1         // All but the lsb are Z (4'bzzz1) (Z filled)
8'h1?         // 0001ZZZZ
2'b1?         // 1Z
4'b10XX       // 10XX
```

- An underscore is not allowed as the first character of a number (this would be a valid identifier)
  - Underscores may be included for readability, and are ignored

Numbers	Bits	Base	Val	Stored
3'b101	3	2	5	101
'b11	?	2	3	000...0011
8'b11	8	2	3	00000011
8'b1010_1011	8	2	171	10101011
3'd6	3	10	6	110
6'o42	6	8	34	100010
8'hAB	8	16	171	10101011
42	?	10	42	00...0101010
'1	?	n/a		11...111

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part

Arithmetic operators

- Addition +, subtraction -, multiplication, / division, and modulo %
- Verilog-2001 adds a power operator, represented by an ** token
  - It will return a **real** number if either operand is a real value, and an **integer** value if both operands are integer values

```
// 8-bit adder
module adder( input [7:0] op1, op2,
              output [7:0] sum );
    assign sum = op1 + op2;
endmodule
```

```
// unsigned 8-bit multiplier
module umul_8bit (input [7:0] A,
                  input [3:0] B;
                  output [11:0] RES);
    assign RES = A * B;
endmodule
```

```
module uadder_8bit(input [7:0] A, B,
                   output [7:0] SUM,
                   output CO);
wire [8:0] tmp; // internal wires are local variables
    assign tmp = A + B;
    assign SUM = tmp [7:0];
    assign CO  = tmp[8];
endmodule
```

- The presence of a '**z**' or '**x**' in a **reg** or **wire** being used in an arithmetic expression results in the whole expression being unknown ('**x**')

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Inference

- Synthesis tools generally are able to *infer* arithmetic operators for the target technology
- Typically there are several implementations in the tool's library for each operator
  - For example, carry-look-ahead adders, ripple-carry adders, Booth multipliers, and Wallace Tree multipliers
- Synthesis tool decides what is the best architectural implementation for a given operator
- Synthesis tool analyzes the netlist of each operator for area and speed, then chooses the best one based upon which implementation is the smallest, yet can still meet the timing goals
- If no timing constraints are specified the tool chooses smallest architectural implementation

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Negative numbers

- Negative numbers are represented in two's complement form
- An integer number with <u>no radix</u> specified (e.g. -12) is considered a signed value
- An integer with a <u>radix</u> specified (e.g. -8'd12) is considered an unsigned value

```
-12/3;          // evaluates to -4
-8'd12 / 3;     // i becomes 244/3=81 (i.e. 8'b11110100/3)
```

- Net and **reg** types are *unsigned* by default

```
reg [15:0] regA; // reg is sized
regA = -16'd12;  // stored as 1111_1111_1111_0100= 2^16-12= 65524
regA/3           //evaluates to 21861
```

  - In this example, **reg** is assigned a negative value but it is considered as an unsigned
- Nets and **reg**s may be declared as signed

```
reg signed [63:0] data;
wire signed [7:0] vector;
input signed [31:0] a;
```

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Sized and unsized numbers

- Sized numbers are considered unsigned

```
16'hC501  //an unsigned 16-bit hex value
-8'd12   // stored as 11110100 unsigned
```

- Sized numbers may be signed: the letter '**s**' can be combined with the radix to indicate that the sized value is signed
- Syntax for sized signed constant values: **<size>'<S><base><value>**

```
16'shC501 //a signed 16-bit hex value
```

- When a sized number specified without letter "s" or a register is used in an expression, its value is always treated as an unsigned number
- Verilog uses 2's complement arithmetic for signed operands and values

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Signed and unsigned system functions

- In addition to being able to declare **signed** data types and values, Verilog adds two new *system functions*, **$signed** and **$unsigned**
  - These functions can be used to convert an unsigned value to signed, or vice-versa
- Casting using **$unsigned(signal_name)** will zero fill the input
- Casting using **$signed(signal_name)** will sign-extend the input
- If the sign bit is **x** or **z** the value will be sign extended using **x** or **z**, respectively
- Assigning to a smaller bit width signal will simply truncate the necessary MSB's as usual
- Casting to the same bit width will have no effect
- Therefore, the casting operators, **$unsigned** and **$signed**, only have effect when casting a smaller bit width to a larger bit

```
// signed multiply
input [7:0] a, b;
output [15:0] z;
wire signed [15:0] z_sgn;
assign z_sgn = $signed(a) * $signed(b);
assign z = $unsigned(z_sgn);
```

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Signed and unsigned arithmetic

```
module sadd (input signed [7:0] A,
             input signed [7:0] B,
             output signed [8:0] Sum);
   assign Sum = A + B; // A and B are implicitly sign extended
endmodule

module sadder (input signed [7:0] A, B,
               output signed [7:0] SUM);
   assign SUM = A + B;
endmodule
```

```
//signed multiplier
module mult_signed
   (input signed [2:0] a,
    input signed [2:0] b,
    output signed [5:0] prod);
   assign prod = a*b;
endmodule
```

```
//signed multiply
input signed [7:0] a;
output signed [15:0] z1, z2;
// cast constant into signed
assign z1 = a * $signed(4'b1011);
// mark constant as signed
assign z2 = a * 4'sb1011;
```

- Note that many operations such as addition, subtraction, and Boolean logic are identical whether a number is signed or unsigned
- However, magnitude comparison, multiplication and arithmetic right shifts are performed differently for signed numbers

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

(2) Structural modeling

- The *behavioral* description specifies what a particular module does while *structural* coding style specifies how a module is composed of other modules or primitive gates to achieve a particular behavior
- A Verilog module can be described by specifying its internal logical *structure* — for instance describing the actual logic gates or other modules it is comprised of
- A structural model of a digital system uses Verilog module instances to describe other components composed of other modules and gate primitives
- Verilog modules, described at the structural level, behavioral level, dataflow level, or any mix of these, can be interconnected with nets, allowing them to communicate

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Structural description

- To describe the functionality of a digital system, we can partition the design into modules which can then be further divided until the design is simple enough to be described accurately
- This *hierarchical* description allows a designer to control the complexity of the design through the well-known *divide-and-conquer* approach to large engineering design
- To allow the creation of a *hierarchy* in a Verilog description, a module can be *instantiated* within another module
  - Note that module *definition* is different than module *instantiation*
  - *We define a module by specifying the functionality of that module*
  - This module may then be instantiated in other modules as many times
- Each of these instantiations are called *instances of the module*
- Multiple instances of the same module are distinguished by distinct instance names
  - Instance names can be used for debugging
- Note that a module cannot contain definitions of other modules

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Structural description

- *Structural* modeling describes a module in terms of how it is composed of sub-components
- Each unique copy of a module or a primitive gate is called an *instance*
- In a top-down description, each of these sub-components can be described *structurally* from its building blocks recursively until the pieces are simple enough to be described *behaviorally*
- By using module instances to describe complex modules, the designer can better manage the complexity of a design
- In a bottom-up description, once a module has been designed (mostly at the behavioral level), it can be used by (*instantiated* in) other modules, which creates a *module hierarchy*
- In this case, the behavior of a design can be described structurally, by making instances of primitive gates and other modules, connecting them together with *nets*

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Module instantiation format

- Module instantiation format:

```
<module_name> <instance_name> (wire_connected_to_port1,
                               wire_connected_to_port2,
                                  . . .
                              wire_connected_to_portn);
```

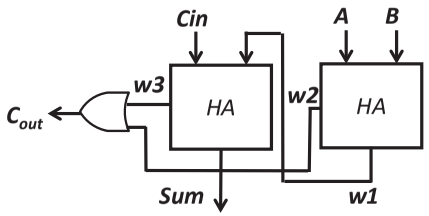
- For connecting the ports of the instantiated module to the nets in the top-level module, we can use a *positional association*
- In a positional (ordered) port connection, the first net connects to the first port of the component, the second net to the second port, etc.
- Recall that when you instantiate Verilog primitives, positional association is used

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Structural modeling of a full-adder

- If you have already described halfadder, you can instantiate it in another module to create a full adder

```
module halfAdder(input in1, in2,
                 output S,C);
  xor U00 (S, in1, in2);
  and U01 (C, in1, in2);
endmodule
```




```
module fullAdder (input A, B, Cin,
                  output Sum, Cout);
wire [1:3] w; //wire [msb:lsb] wire1, wire2,...

  halfAdder HA0(A, B, w[1], w[2]);     //instantiation of HAs
  halfAdder HA1(w[1], Cin, Sum, w[3]); //half_adder is defined (previously)
                                       //in another module
```

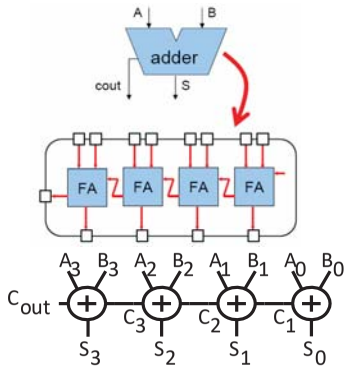
Module name *Instance name*

```
  assign Cout = w[2] || w[3]; //the same as or g1(Cout,w[2],w[3]);
endmodule
```

- In this example, **`fullAdder`** is called the *top-level module*
- Note that a module may contain a combination of *behavioral* modeling statements (**`always`** statements), *continuous assignment* statements (**`assign`** statements), or module *instantiations* referencing other modules or gate level primitives

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Structural modeling example



A 4-bit binary adder can be formed with four full-adders

```
module 4bitAdder (input [3:0] A,B,
                  output [3:0] S,
                  output Cout);

wire [3:1] C;  //Intermediate carry signals

   // instantiate an array of instances
   fullAdder  FA0 (A[0],B[0],, S[0],C[1]),
              FA1 (A[1],B[1],C[1], S[1],C[2]),
              FA2 (A[2],B[2],C[2], S[2],C[3]),
              FA3 (A[3],B[3],C[3], S[3],Cout);

endmodule
```

- In this example, **`4bitAdder`** is the top-level module
- As can be seen in this example, module ports may be left unconnected in a positional port connection list by omitting an expression, leaving two adjacent commas

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Array of instances

- Consider this module example:
- This definition of the **`xor8`** module is rather tedious because each **`XOR`** instance had to be individually numbered with the appropriate bit
- Verilog has a shorthand method of specifying an array of instances where the bit numbering of each successive instance differ in a controlled way

```
module xor8 (output [1:8] xout,
             input [1:8] xin1, xin2);
  xor (xout[8], xinl [8], xin2[8]),
      (xout[7], xinl [7], xin2[7]),
      (xout[6], xinl [6], xin2[6]),
      (xout[5], xinl [5], xin2[5]),
      (xout[4], xinl [4], xin2[4]),
      (xout[3], xinl [3], xin2[3]),
      (xout[2], xinl [2], xin2[2]),
      (xout[l], xinl [l], xin2[l]);
endmodule
```

```
module xor8 (output [1:8] xout,
             input [1:8] xin1, xin2);
  xor a[l:8] (xout, xinl, xin2);
endmodule
```

- The second module is the equivalent redefinition of module **`xor8`** using arrays of instances
- The array of instances specification uses the optional *range specifier* to provide the numbering of the instance names
- There are no requirements on the absolute values or the relationship of the msb or lsb of the range specifier — both must be integers and one is not required to be larger than the other
- Indeed, they can be equal in which case only one instance will be generated
- Given msb and lsb, 1 + abs(msb-lsb) instances will be generated

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Array of instances

- This example showed the case where each instance was connected to a bit-select of the outputs and inputs
- When the instances are generated and the connections are made, there must be an equal number of bits provided by the terminals (ports, wires, registers) and needed by the instances
- In this example, eight instances needed eight bits in each of the output and input ports (It is an error if the numbers are not equal.)
- <u>Instances are not limited to bit-select connections</u>
- If a terminal has only one bit (it is scalar) but there are *n* instances, then each instance will be connected to the one-bit terminal
- This example shows D flip flops connected to form a register
- The equivalent module with the instances expanded is shown at the bottom
- Note that **`CLK`** and **`RST`**, being one-bit scalars, are connected to each instance

```
module register (output [7:0] q,
                 input [7:0] d,
                 input CLK, RST);
   dff r[7:0] (q, d, RST, CLK);
endmodule

module registerExpanded (output [7:0] q,
                         input [7:0] d,
                         input CLK, RST);
  dff r7 (q[7], d[7], RST, CLK),
      r6 (q[6], d[6], RST, CLK),
      r5 (q[5], d[5], RST, CLK),
      r4 (q[4], d[4], RST, CLK),
      r3 (q[3], d[3], RST, CLK),
      r2 (q[2], d[2], RST, CLK),
      r1 (q[1], d[l], RST, CLK),
      r0 (q[0], d[0], RST, CLK);
endmodule
```

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Name association

- It is easy to swap two ports accidentally in an ordered list
- If the ports are both the same width and direction, then such errors can be difficult to debug
- Verilog allows to connect to a module's ports by naming the port and giving its connection
- It is recommended to use *named port connections* to avoid this problem and improve readability
- In a name association, the module's port names are used and the order of connections is irrelevant
- The period (".") introduces the port name as defined in the module being instantiated

```
<module_name> <instance_name> (.port1    (wire_connected_to_port1),
                               .port2    (wire_connected_to_port2),
                                       . . .
                               .portn    (wire_connected_to_portn) );
```

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Name association

```
module halfAdder(input in1, in2,
                 output S,C);
  xor U00 (S, in1, in2);
  and U01 (C, in1, in2);
endmodule
```

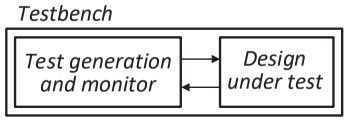
```
module fullAdder (input A, B, Cin,
                  output Sum, Cout);
wire [1:3] w; //wire [msb:lsb] wire1, wire2,...

  halfAdder HA0(.in1(A), .in2(B), .S(w[1]), .C(w[2]));
  halfAdder HA1(.in1(w[1]), .in2(Cin), .S(Sum), .C(w[3]));

  assign Cout = w[2] || w[3]; //the same as or g1(Cout,w[2],w[3]);

endmodule
```

- Given that both port names and connection names are specified together, the connections may be listed in any order
- Ports may be left unconnected in a named port connection list either by omitting the name and expression altogether, or by leaving the expression blank in the parenthesis

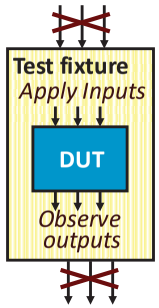
These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Functional simulation using testbenches

- The idea of simulating a Verilog module is similar to an engineer's workbench where the system being designed is wired to a test generator that is going to provide inputs to the design under test and monitor the outputs as they change
- A *testbench* (or test fixture) is an HDL module that passes test vectors (or *stimuli*) to another module, called the design (module, unit) under test (DUT)



- There are no input or output ports for the testbench
- In order to simulate a module, the test vectors need to be known so as to generate known output signals
- The test vectors (inputs to the DUT) are defined as **reg** and the outputs are defined as **wire**
- Simulator can apply testvectors to the DUT at some specified time, which could be in the future rather than immediately
- Test benches are not synthesizable so it can use all Verilog commands



These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Initial statement

- One approach to apply stimuli to the DUT is to use **initial** statements
- An **initial** statement contains a statement or block of statements

```
initial [begin]
... Procedural statements ...
[end]
```

- An initial statement executes the statements in its body at simulation time zero (at the beginning of simulation) only once and finish when the last statement executes
- The **initial** statement starts with the first procedural statement and continues executing until it finds the delay on the next statement
- The **initial** block is then *suspended* and scheduled to wake up at certain time
- Therefore, an **initial** statement is typically used in testbenches to initialize variables and uses procedural statements and delays to apply the test vectors in the appropriate order
- **Initial** statements are not synthesizable and should only be used in testbenches for simulation, not in modules intended to be synthesized into actual hardware
- The test vectors can be specified using one or more **initial** statements

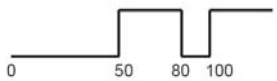
These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Delayed assignment

- A timing control #Δ*t*; is commonly used in testbenches to delay or schedule execution of statements
- A timing (delay) control before a statement causes the execution of the immediately following statement to be delayed

```
#Δt Procedural_statement;
```

```
initial begin
  a = 0;
  #50 a = 1;
  #30 a = 0;
  #20 a = 1;
end
```



Each assignment is blocked by its previous one

- Δ*t* is the number of time units, specifies the delay time units before a statement is executed (during simulation)
- It means that Δ*t* time units has to pass before the RHS statement is executed and the result is assigned to the LHS
- The expression on the RHS of a procedural assignment is evaluated when the assignment is executed
- When no delay is specified, the default is zero
- Note that when a delay time of zero is specified (#0), it forces the statement to the end of the list of statements to be evaluated at the current simulation time

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Compiler directives

- Compiler directives are instructions to the Verilog compiler
- Compiler directives start with a back tic ` and executed prior to simulation time zero and synthesis
- The effect of a compiler directive starts from the place where it appears in the source code, and continues through all modules synthesized after the directive, whether in the same file, or in files that are synthesized separately, to the point where the directive is reset, until the next appearance of the directive, or the end of the last module to be synthesized, or until
  - A compiler directive can be used with different values in different modules
- Example:
  - `**`include** file_name // include source code from another file
  - `**`define** macro_name macro_code // substitute macro_code for macro_name
  - `**`define** macro_name(par1, par2,...) macro_code // parameterized macro
  - `**`undef** macro_name // undefine a macro
  - `**`timescale** 1ns/1ns // units/precision for time e.g. for %t
  - `**`default_nettype** net_type // sets the default net type for implicit net declarations, net_type is one of:
    // wire, tri, tri0, tri1, triand, trior, trireg, wand, wor, none
  - `**`ifdef** macro_name1 // include source lines1 if macro_name1 is defined
    <source lines1> // the source lines1
    `**`elsif** macro_name2 // any number of elsif clauses, the first defined
    <source lines2> // macro_name includes the source lines
    `**`else** // include source lines3 when no prior macro_name defined
    <source lines3> // the source lines 3
    `**`endif** // end the construct
  - `**`ifndef** macro_name // like `ifdef except logic is reversed, true if macro_name is undefined
  - `**`resetsall** resets all compiler directives to default values

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

`**timescale** compiler directive

- Simulation times have been described in terms of "time units"
- Any design that includes **#**delays relies on the accuracy of the specified time units and precisions set by the compiler directive '**timescale**
- The `**timescale** compiler directive is used to define time units of any delay operator (#) and the precision to which time calculations will be rounded

```
`timescale TimeUnit / PrecisionUnit
```

where `TimeUnit` and `PrecsionUnit` are in **TU** format:

```
T = {1, 10, 100}
U = {s, ms, us, ns, ps, fs}
```

- Precision is the maximum number of decimal places used in time values
  - Hence, the precision unit defines how delay values are to be rounded off during simulation and all delays are rounded to the nearest precision unit
- Precision unit must be less than or equal to the time unit
  - Note that only 1, 10 or 100 are valid integers for specifying time units and valid time units include s, ms, us (µs), ns, ps, fs
- The default is 1ns with precision of 100ps

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Example

- If the compiler directive , `**timescale** 1ns/100ps was placed before a module definition, then all delay operators in that module and any module that followed it would be in units of nanoseconds and any time calculations would be internally rounded to the nearest one hundred picoseconds (1/10ns)
  - For example 10.512ns is interpreted as 10.5ns
- Example: `**timescale 10ps/1ps**

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

- The `**timescale** directive can have a huge impact on the performance of simulators. It is a common new-user mistake to select a time precision of 1ps
- Adding a 1ps precision to a model that is adequately modeled using either 1ns or 100ps time precisions can increase simulation time by more than 100% and simulation memory usage by more than 150%

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Full-adder and its testbench

```
module fullAdder(input a, b, cin,
                 output sum, cout);
   assign cout = a & b | cin & (a ^ b);
   assign sum = cin ^ a ^ b;
endmodule
```

DUT

```
`timescale 1ns / 1ns
module tb_fullAdder;
wire  s,co; //outputs are defined as wires
reg   a,b,ci; // stimuli are defined as regs

 // Instantiate the design under test (DUT) in the testbench
 fullAdder FA00 (.a(a),.b(b),.cin(ci),.cout(co),.sum(s));

 initial  begin
        a=1'b0; b=1'b0; ci=1'b0; // time = 0
  #10;  ci=1'b1;                 // time = 10.
                                 // Note that only ci is changed at time=10
  #10;  b=1;//the same as b=1'b1;// time = 20
  #10;  ci=1'b0; a=1'b1;         // time = 30
  #1    $finish;                 // Verilog built-in system function
 end
endmodule
```

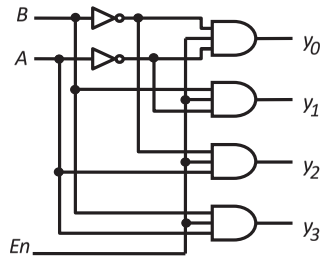
- Use tb_modulename for naming your testbenches
- List the variables only when their values change
- $**finish** is a control task that exits the simulator and the control returns back to the host operating system

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

2-to-4 decoder

```
module decoder2to4 (input A,B,En,
                    output [0:3] y);
wire Anot,Bnot;
   not // multiple gates of the
       // same type are separated by ,
      U0 (Anot,A),
      U1 (Bnot,B);
   and
      U2 (y[0],Anot,Bnot,En),
      U3 (y[1],Anot,B,En),
      U4 (y[2],A,Bnot,En),
      U5 (y[3],A,B,En);
endmodule
```

Array of instances




```
module decoder2to4 (input A, B, En,
                    output [0:3] y);
   assign y[0] = ~A & ~B & En,
          y[1] = ~A & B & En,
          y[2] = A & ~B & En,
          y[3] = A & B & En;
endmodule
```

Dataflow model

```
module tb_decoder2to4;
wire  [0:3] y; //outputs are defined as wires
reg   A, B; // stimuli are defined as regs
reg   EN;

    // Instantiate the decoder (DUT)
    decoder2to4 UUT (.A(A), .B(B), .En(EN), .y(y));

    initial  begin
      A  = 1'b0;  B  = 1'b0; EN = 1'b0; // time = 0
      #10; EN = 1'b1;                   // time = 10
      #10; A  = 1'b0;  B  = 1'b1;       // time = 20
      #10; A  = 1'b1;  B  = 1'b0;       // time = 30
      #10; A  = 1'b1;  B  = 1'b1;       // time = 40
      #5;  EN = 1'b0;                   // time = 45
      #5;
    end
endmodule
```

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Concatenation operator

- The *concatenation operator* {op1, op2, ..} combines two or more operands to form a larger vector

```
reg a;
reg [2:0] b, c;
a = 1'b1;  b = 3'b010;
x = {a, b};   // x = 1010
y = {b, 2'b11, a};  // y = 010_11_1 //underscore is just for readability
z = {b, 1};// incorrect. The operands must be sized
```

- Often, it is necessary to operate on a subset of a bus or to concatenate, i.e., join together, signals to form busses

```
reg [15:0] a,b;
{b[7:0],b[15:8]}= {a[15:8],a[7:0]}; // byte swap

wire [7:0] A,B;  wire [3:0] C;  wire [11:0] D;
assign {A,B} = {C,D};
assign {A,B} = X; // split X into A and B. The width of A and B are defined
```

```
wire [7:0] A1,B1,A2,B2,A3,B3;
wire [15:0] X = 16'b0000001111001100;
assign {A1,B1} = X; // split X into A and B
wire [19:0] Y = 20'b00000011110011001111;
assign {A2,B2} = Y;
wire [11:0] Z= 12'b001111001100;
assign {A3,B3} = Z;
```

X[15:0]	0000001111001100	0000001111001100
A1[7:0]	00000011	00000011
B1[7:0]	11001100	11001100
Y[19:0]	00000011110011001111	00000011110011001111
A2[7:0]	00111100	00111100
B2[7:0]	11001111	11001111
Z[11:0]	001111001100	001111001100
A3[7:0]	00000011	00000011
B3[7:0]	11001100	11001100

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Replication operator

- The *replication operator* **{N{item}}** makes **N** fold replication of item
  - **N** is a constant value

```
a = 1'b1; b = 3'b101;
r = {3{a}, b}; // r = 111_101
//{4{4'b1001,1'b0}} // 10010100101001010010
assign x = {2{1'b0}, a}; // 001
```

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Some tricky points

- A sized negative number is not sign extended when assigned to a register

```
reg [7:0] byte;
reg [3:0] nibble;
initial begin    //Initial statements are used in testbenches
  nibble = -1;    // i.e. 4'b1111
  byte = nibble;  // byte becomes 8'b0000_1111
end
```

- To perform signed arithmetic, all operands in the expression must be signed
  - *If any operand in an expression is unsigned, the operation is considered to be unsigned*

- Consider this addition example
- Adding two values that are n-bits wide will produce a n+1 bit result
  - In general adding an m-bit and an n-bit numbers require max(m,n)+1 bit for results
- In this example, A and B and cin are unsigned and hence the addition is unsigned

```
module uadd (input [2:0] A,
             input [2:0] B,
             input cin,
             output [3:0] Sum);
  assign Sum = {A[2],A} + {B[2],B} + cin;
endmodule
```

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Some tricky points

- Consider this addition module
- This code is incorrect
  - If **cin = 1**, then the **$signed** operator sign extends the **cin** so it now equals **4'b1111** and we would have been subtracting 1 instead of adding 1

```
module sadd (input signed [2:0] A,
             input signed [2:0] B,
             input cin,
             output signed [3:0] Sum);
  assign Sum = A + B + $signed(cin);
endmodule
```

- A similar functional error occurs if we declare **cin** to be a signed input
- We can use a concatenation operator to solve the issue

```
module sadd (input signed [2:0] A,
             input signed [2:0] B,
             input cin,
             output signed [3:0] Sum);
     assign Sum = A + B + $signed({1'b0,cin});
endmodule
```

- Note that concatenation results are unsigned, regardless of the operands

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Splitting the output

- We can use concatenation operator on the left hand side of a statement to split an output into pieces
- This example shows how to split an output into two pieces, SUM and Cout

```
module adder4bits (input [3:0] A,B, // little endian convention
                   input Cin,
                   output [3:0] SUM,
                   output Cout);
  assign {Cout,SUM} = A + B + Cin;
   // {} is the concatenation operator
   // the 4 least significant bits of A+B+Cin will be stored in in SUM
   // and the most significant bit will be stored in Cout
endmodule
```

- This example shows how to sign extend a 16-bit number to 32 bits by copying the most significant bit into the upper 16 positions

```
module mul (input [7:0] a, b,
            output [7:0] upper, lower);
   assign {upper, lower} = a*b;
endmodule
```

```
module signextend (input [15:0] a,
                   output [31:0] y);
  assign y = {{16{a[15]}}, a[15:0]};
endmodule
```

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Some tricky points - Sign of part selects

- Part-select results are unsigned, regardless of the operands, <u>even if part-select specifies the entire vector</u>

```
//Functionally incorrect
input signed [7:0] a, b;
output signed [15:0] z1, z2;
assign z1 = a[7:0];     // a[7:0] is unsigned -> zero-extended
assign z2 = a[6:0] * b; // a[6:0] is unsigned -> unsigned multiply
```

```
//Functionally correct
input signed [7:0] a, b;
output signed [15:0] z1, z2;

assign z1 = a; // a is signed -> sign-extended
assign z2 = $signed(a[6:0]) * b; // cast a[6:0] to signed -> signed multiply
```

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Some tricky points - Mixing signed and unsigned

- Do not mix **unsigned** and **signed** types in one expression
  - This results in functional incorrectness because Verilog interprets the entire expression as unsigned if one operand is unsigned
- The synthesizer generates a warning message when unsigned-to-signed/signed-to-unsigned conversions occurs (Check for warnings about implicit conversions/assignments)

```
//unsigned multiply
input [7:0] a;
input signed [7:0] b;
output signed [15:0] z;
   assign z = a * b;
```

```
// unsigned multiply
input signed [7:0] a;
output signed [11:0] z;
// constant is unsigned
assign z = a * 4'b1011;
```

```
//signed multiply
input [7:0] a;
input signed [7:0] b;
output signed [15:0] z;
assign z = $signed(a) * b;
```

```
//signed multiply
input signed [7:0] a, b;
output signed [15:0] z;
assign z = a * b;
```

- If we multiply –3 (**3'b101**) by 2 (**3'b010**) with the following code we get 10 (**6'b001010**)
  - The reason for this is that since we mixed signed with unsigned we actually multiplied 5 by 2 and got 10 since the operation is considered unsigned

```
module mult (input signed [2:0] a,
             input [2:0] b,
             output signed [5:0] prod);
  assign prod = a*b;
endmodule
```

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Relational and equality operators

- The relational operators typically used in conditional expressions
- These include **>** (greater than), **>=** (greater than or equal), **==** (equal), and **!=** (not equal)
- Relational operators compare two operands and indicate whether the comparison is true or false
  - They evaluate to a Boolean *false*, which is equivalent to one bit 1'b0 and Boolean *true*, which is equivalent to 1'b1
- These operators synthesize into comparators
- Note that since **reg** and **wire** types are unsigned, the synthesized comparators will be unsigned
- If the comparison is ambiguous, the result is unknown (**1'bX**)

```
2'b10 > 2'b0X // is true (1'b1)
2'b11 > 2'b1X // is unknown (1'bX)
```

- So a comparison operator may return **0**, **1** or **x**
- The rules about unknown and ambiguous comparisons using **== != < > <= >=** are not followed closely by all simulators. Be careful!

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Logical and case equality and inequlity operators

- The *logical equality* **==** and inequality **!=** operators will return an **X** if any bit of an operand is **X** or **Z**
- However, the *case equality* operator (**===**) and inequality operator (**!==**) can be used to specify that individual unknown or high impedance bits are to take part in the comparison
  - That is, a 4-valued logic comparison is done where the value of each bit being compared, including the unknowns and high impedances, must be equal
  - Therefore, case equality **===** and inequality **!==** operators: **X** and **Z** values are considered in comparison
- While the case equality operators (**===**, **!==**) and the bitwise operators treat the individual bits of their operands separately, the case equality operators are not generally synthesizable

Operand A	Operand B	===	!==	==	!=
0110	0110	1	0	1	0
0110	0XX0	0	1	X	X
0XZ0	0XZ0	1	0	X	X

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Summary

Bitwise

~a	NOT
a & b	AND
a | b	OR
a ^ b	XOR
a ~^ b	XNOR

Logical

!a	NOT
a && b	AND
a || b	OR

Reduction

&	AND
~&	NAND
|	OR
~|	NOR
^	XOR

Note distinction between ~a and !a

Comparison

a < b a > b a <= b a >= b	Relational
a == b a != b	[in]equality returns x when x or z in bits. Else returns 0 or 1
a === b a !== b	case [in]equality returns 0 or 1 based on bit by bit comparison

- Note the distinction between the *unary reduction operators* and the *bitwise logic operators,* which look the same. The meaning depends on the context, and brackets may be needed to force a particular interpretation

```
module mulTest (output [63:0] y,
                input  [31:0] a, b);
  assign y = a * b;    //defaults to wire, width of port y
  assign eq = (a == b);// defaults to 1-bit wire in Verilog-2001
                       // ERROR in Verilog 1995: 'eq' is not
                       // declared
endmodule
```

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Shift operators

- *Logical shift* operators >> and << shift the first operand by the number of bits specified by the second operand
- Result is the same size as first operand, always zero filled from the left or right

```
a = 4'b1010;
d = a >> 2;      // shift right: d = 0010
c = a << 1;      // shift left: c = 0100
```

- Vacated positions are filled with zeros for both left and right shifts (There is no sign extension)
- *Arithmetic shift* operators **<<<** and **>>>** have been added to Verilog-2001
- An arithmetic right-shift operation **>>>** maintains the sign of a value, by filling with the sign-bit value as it shifts
  - For example, if **d=8'b10100111** is an 8-bit signed variable, then

```
d >> 3 //logical shift yields 8'b00010100
d >>> 3 //arithmetic shift yields 8'b11110100
```

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

The conditional operator

- The conditional operator (?:) can be used in place of the if statement when one of two values is to be selected for assignment
- The general form of the conditional operator is:

signal ::= conditional_expression ? true_expression : false_expression

- If the **conditional_expression** is TRUE (or nonzero), then the operator chooses the value of the **true_expression** to be assigned to **signal**. Otherwise the value of **false_expression** will be assigned to **signal**
- ?: is also called a *ternary operator* because it takes three inputs
- ?: is especially useful for describing a multiplexer
- The conditional operators are synthesizable as multiplexers or tri-states
- Multiplexer is a combinational circuit where an input is chosen by a select signal

```
module generate_set_of_MUX (input [0:3] a, b,
                            input sel,
                            output [0:3] f);
  assign f = sel ? a : b;
endmodule
```

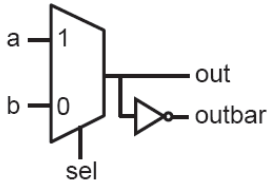
Conditional operator generates a MUX

- Note that a two-input mux is actually a three-input device (a,b,sel): **out =a** if **sel =1** and **out= b** if **sel =0**

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Multiplexer examples

```
module mux2x1 (input a, b, sel;
               output out, outbar);
wire w;
   assign w = sel ? a : b;
   assign out = w;
   assign outbar = ~w; //~ denotes not
endmodule
```



- Could have we defined out as **inout** instead of using **wire w**?
- Two different description of the same module (multiplexer):

```
module mux(output f,
           input a,b,sel)
  assign f=(a&~sel) | (b&sel);
endmodule
```

```
module mux(output f,
           input a,b,sel);
wire nsel,d0,d1;
  not g1 (nsel, sel);
  and g2 (d0, nsel, a);
  and g3 (d1, sel, b);
  or  g4 (f, do,d1);
endmodule
```

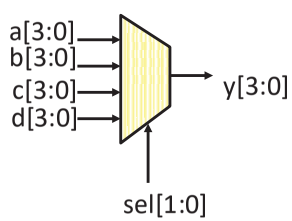
These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

A 4-to-1 multiplexer

```
//A 4:1 multiplexer can select one of four inputs using nested conditional
//operators.
module mux4(input [3:0] d0, d1, d2, d3,
            input [1:0] s,
            output [3:0] y);
   assign y = s[1] ? (s[0] ? d3 : d2) : (s[0] ? d1 : d0);
endmodule
```

- If s[1] = 1, then the multiplexer chooses the first expression, (s[0] ? d3 : d2). This expression in turn chooses either d3 or d2 based on s[0] (y = d3 if s[0] = 1 and d2 if s[0] = 0). If s[1] = 0, then the multiplexer similarly chooses the second expression, which gives either d1 or d0 based on s[0]

```
module mux_4bits(input [3:0] a, b, c, d,
                 input [1:0] sel,
                 output [3:0] y);
  assign y =
    (sel == 0) ? a :
    (sel == 1) ? b :
    (sel == 2) ? c :
    (sel == 3) ? d : 4'bx;
endmodule
```



These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Operator precedence

- The operator precedence specifies the order of evaluation
- This table shows the precedence of operators from highest to lowest
  - Operators on the same level evaluate from left to right
- When an expression is evaluated, the operator with higher precedence is evaluated first

Operator	Name
[ ]	bit-select or part-select
( )	parenthesis
!, ~	logical and bit-wise NOT
&, |, ~&, ~|, ^, ~^, ^~	reduction AND, OR, NAND, NOR, XOR, XNOR; If X=3'B101 and Y=3'B110, then X&Y=3'B100, X^Y=3'B011;
+, -	unary (sign) plus, minus; +17, -7
{ }	concatenation; {3'B101, 3'B110} = 6'B101110;
{{ }}	replication; {3{3'B110}} = 9'B110110110
*, /, %	multiply, divide, modulus; */ and % not be supported for synthesis*
+, -	binary add, subtract.
<<, >>	shift left, shift right; X<<2 is multiply by 4

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Operator precedence – Cont'd

- For example, in the **a + b >> 1** expression, **a + b** is evaluated first and then **>> 1** is evaluated
- We can use parentheses to alter the precedence, as in **a + (b >> 1)**
- It is a good practice to use parentheses to make an expression clearer, as in **(a + b) >> 1**, even when they are not required

<, <=, >, >=	comparisons. Reg and wire variables are taken as positive numbers.
= =, !=	logical equality, logical inequality
= = =, != =	case equality, case inequality; not synthesizable
&	bit-wise AND; AND together all the bits in a word
^, ~^, ^~	bit-wise XOR, bit-wise XNOR
|	bit-wise OR; AND together all the bits in a word
&&,	logical AND. Treat all variables as False (zero) or True (nonzero). logical OR. (7||0) is (T||F) = 1, (2||-3) is (T||T) =1, (3&&0) is (T&&F) = 0.
||	
? :	conditional. x=(cond)? T : F;

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Summary of operators

- Arithmetic operators
  `m+n  m-n  -m  m*n  m/n  m%n (modulo), ** (in verilog 2001 only)`
- Bitwise operators
  `~m  m&n  m|n  m^n  m~^n  m^~n`
- Unary reduction operators
  `&m  ~&m  |m  ~|m  ^m  ~^m  ^~m`
- Logical operators
  `!m  m&&n  m||n`
- Equality operators (compares logic values of **0** and **1**)
  `m==n  m!=n`
- Identity operators (compares logic values of **0**, **1**, **X** and **Z**)
  `m===n  m!==n`
- Relational operators
  `m<n m>n  m<=n  m>=n`
- Logical shift operators
  `m<<n  m>>n`
- Miscellaneous operators
  `sel?m:n  {m,n}  {n{m}}`

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

(3) Behavioral level

- We learned that the behavior of a module can be described using *continuous signal assignment statements*
- A *behavioral* model of a module is an abstraction of how the module works (describing the function of a module behaviorally) without directly specifying how the module is implemented in terms of structural logic gates
- In this case, the behavior of logic is described similar to a programming language at the higher-level of abstraction than gate-level modelling and dataflow level modelling
- In this way, the designer can focus on developing the design that works correctly and has the intended behavior
- Behavioral models are useful at the early stages of the design cycle where a designer is more concerned with simulating the system's intended behavior
- Synthesis tools read a behavioral description of a circuit and automatically design a gate level structural version of the circuit
- The behavioral model can be synthesized to several alternate structural implementations of the behavior

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Behavioral level

- The basic Verilog statement for describing the behavior is an **always** block
- An **always** statement starts off with an *event control* statement
- An *event* occurs when a signal (a net or register) changes its value

```
always [event_control]
begin [: name_for_block]
[variable declaration]
... Procedural statements ...
end
```

- An *event control statement* always starts with symbol @ and has the format of @(sensitivity list)
- The general form of the event control statement is:
- **event_control::= @ event_identifier |@ (event_expression)|@* |@(*) |(<pos|neg>edge <signal>)**
- Event control statements provide a means of watching for a change in a value

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Sensitivity list

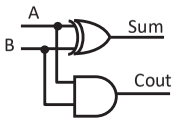
- The *sensitivity list* of an **always** block is the list of names appearing in the event control statement
- Sensitivity list specifies events on which signals activating **always** blocks
- An event control is triggered when any one of the events in the sensitivity list occurs
  - Any number of events can be expressed in the event control statement such that the occurrence of any one of them will trigger the execution of the statement
- These event identifiers are separated with **or** (in Verilog 1995) or commas (in Verilog 2001) , which allows us to wait for any of several events
- Sensitivity list can be used to describe both combinational and sequential logic
  - Sensitivity list for the combinational logic has the format of **always @(list_of_sensitive_signals)|@*|@(*)**
  - Sensitivity list for the sequential logic has the format of **always @ (<pos|neg>edge <signal>)**
- Essentially, **@(*)** is shorthand for "all the signals on the right-hand side of the statement or in a conditional expression"

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Examples

- The **always** block can be used to infer combinational logic and sequential logic

```
module halfAdder_behavioral(input A, B,
                            output reg Sum, Cout);
always @(A,B) begin // the same as always @*
    Sum  = A ^ B;
    Cout = A & B;
  end
endmodule
```



- **always** block reevaluates the statements inside the **always** statement any time any of the signals in the sensitivity list change
- In this example, the **always** statement states that the simulator should suspend execution of this **always** block until an event (change) occurs on A or B
- Every **always** starts executing at the start of simulation
- When a change occurs on any one (or more) of these, then execution will continue with the statements in the **begin … end** block
- If, while waiting for the event, a new value for the expression is generated that happens to be the same as the old value, then no event occurs
- 2:1 multiplexer at the behavioral level:

```
module mux(output reg f,
           input a,b,sel)
  always @*
    f=(a&~sel) | (b&sel);
endmodule
```

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Reg type in procedural statements

- A register type must be used when the signal is on the left-hand side of a procedural assignment (e.g., target variables in the body of an **always** statement)
- Why only **reg** type variables can be assigned within an **always** block?
  - The sequential **always** block executes only when the event expression triggers
  - At other times the block is in the suspended mode
  - A signal being assigned to must therefore retain the last value assigned (not continuously driven)
- Register variable can be referenced anywhere in module, but they can be assigned only with procedural statements
  - Anything assigned in an always block must be declared as type **reg**
- Register variable cannot be **input** or **inout**

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Nets vs. regs

- Nets are used to <u>model connections</u> between continuous assignments & instantiations
- Nets must be continuously driven by primitive, continuous assignment, module ports
- Net variable can be referenced anywhere in module, but they may not be assigned within procedural blocks. Exception: **force** … **release**
  - Hence, in a procedural blocks, the value of nets can be read but cannot be assigned
- The 'reg' declaration explicitly specifies the size
  - **reg** x, y; // single-bit register variables
  - **reg** [15:0] bus; // 16-bit bus, bus[15] MSB

LHS must be a reg type

```
reg [8:0] sum;
always @(a or b)
   sum = a + b;
```

LHS must be a net type

```
wire [8:0] sum;
assign sum = a + b;
```

```
wire y;
and g1(y, c, d);
```

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

While TRUE

- An always block uses *procedural statements* to model a design at a higher level of abstraction than the other levels
- An **always** containing more than one procedural statement must enclose the statements in a **begin-end** block (i.e., a compound statement)
  - **begin** … **end** block statements are used to group several statements
- The **always** statement, essentially a "while (TRUE)" statement, includes one or more procedural statements that are repeatedly executed
  - An **always** with no event control will loop forever
- All statements within the always statement are executed *sequentially* once when one or more than one event (i.e., change in the logical value of a signal) occur on any signal in the sensitivity
- The always continuously repeats its statement(s), never exiting or stopping
- The execution of the process containing the event control is *suspended* until the change occurs
- Thus, the value must be changed by a separate process
- A behavioral module can be described using one or more **always** blocks
- Several **always** statements are executed continuously and concurrently

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Combinational logic using **always**

- A combinational logic will be inferred if the sensitivity list is written to respond to changes in all of the inputs and the body prescribes the output value for every possible input combination
- This follows from the very definition of combinational logic — *any* change of *any* input value may have an immediate effect on the resulting output
- Thus when describing combinational logic using procedural statements, every element of the always block's input set must appear in the sensitivity list of the event statement without any edge specifiers (i.e., **posedge** and **nededge**)
- For a combinational logic, the list must specify only level changes and must contain all the variables appearing in the right-hand-side of statements in the always
- If an element of the input set is not in the sensitivity list, or only one edge-change is specified, then it cannot have an immediate effect, which is not true of combinational circuits

```
module shift (input [3:0] data,
              output reg [3:0] q1, q2);
parameter B = 2;
 always @* begin
  q1 = data << B; // logical shift left
  q2 = data >> B; //logical shift right
 end
endmodule
```

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

@*

- **always** @(complete event list) synthesizes to a combinational logic
- A common error in specifying combinational circuits with procedural statements is to incorrectly specify the sensitivity list
- If the intent is to describe a combinational circuit using an always block, the explicit sensitivity list can replaced with a **@(*)** or **@*** constructs

```
module logical (input [3:0] A, B;
                output reg Q[1:6]);
 always @* begin
   Q[1] = A > B;  //greater than
   Q[2] = A < B;  //less than
   Q[3] = A >= B; //greater than equal to
   Q[4] = A <= B  //less than equal to
   Q[5] = A == B;  //equality
   Q[6] = (A != B) //inequality
end
endmodule
```

```
module arithmetic (input [3:0] A, B,
                   output reg [4:0] Q1,
                   output reg [3:0] Q2, Q3,
                   output reg [7:0] Q4);
  always @* begin
      Q1 = A + B; //addition
      Q2 = A - B; //subtraction
      Q3 = A / 2; //division
      Q4 = A * B; //multiplication
  end
endmodule
```

- The **@*** token indicates that the simulator or synthesis tool should automatically be sensitive to changes on any values which are read in the body of **always** statement

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Integer *and* **regs**

- Register types supported for synthesis are **reg** and **integer**
- For **integer**, it takes the default size, usually 32-bits, and the synthesizer tries to determine the size
- Note that in arithmetic expressions, an '**integer**' is treated as a 2's complement signed integer but a **reg** is treated as an unsigned quantity
- General rule of thumb: **reg** used to model actual hardware registers such as counters, accumulator, etc., however, '**integer**' used for situations like loop counting
- When **integer** is used, the synthesis tool often carries out a data flow analysis of the model to determine its actual size

```
wire [1:10] A, B;
integer C;
C = A + B;
```

What is the size of C?

- No ranges or arrays supported (is this correct?)

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Local variables in an **always** statement

- You can define local variables inside an **always** statement to limit the scope of variables
- It can include register, integer and parameter declarations
- The local variable is only visible inside the block. So you can use another variable with the same name outside of this scope
- In general, if a **begin-end** block has local declarations, it must be named (i.e. it must have a label)

```
always [event_control_statement] begin [: name_for_block]
[variable declaration]
   ... Procedural statements ...
end
```

- A behavioral model may contain one or more **always** statements
- All **always** blocks in a module execute simultaneously
- This is very unlike conventional programming languages, in which all statements execute sequentially
- Note that modules may not be instantiated inside procedural blocks, such as **always** statements

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Recommendations

- Several modules may be described in one file (not recommended)
- It is better to name the file and module the same name. This avoids any confusion while compiling the files and during the synthesis
- When the question asks you to describe a module at a particular level, you should model your design in that style only and you are not allowed to use any other style
  - For example, if the question asks you to model a design at the behavioral level, then you should describe your module using **always** statements only. Using gates (gate-level modeling), **assign** statements (dataflow level modeling), and module instantiation (structural level modeling) are not allowed
- However, if the question does not specify a particular modeling style, then you can describe your module at any level
- Low level gates or Boolean level constructs (Verilog primitives) constrain the synthesis tool
  - Don't spend a lot of time trying to force synthesis tool to implement a gate-level solution by describing the module with primitive gates
  - The synthesis tool takes Boolean expressions and gate level instantiations and translate them to an optimized description. You almost never get exactly the gate-level implementation

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Parameterized modules

- Module definitions may be expressed using parameters
- A *parameter* is a constant with a name that is local to a module
- Each instance of a module may redefine the parameters to be unique to that instance
- Since parameters can be overridden, they allow customization of a module during instantiation
- Thus, you can build modules that are parameterized and specify the value of the parameter at each instantiation of the module
- For *parameterized modules*, parameter declarations typically precede the port declarations
- The list of parameters is introduced and declared before the port list so that some of the port specifications can be parameterized

```
module multiplier #(parameter SizeA = 8,
                            SizeB = 4)
                   (output [SizeA+SizeB-1:0] mult,
                    input [SizeA-1:0] a,
                    input [SizeB-1:0] b);
   assign mult = a * b;
endmodule
```

Verilog 2001

```
module multiplier (mult, a, b);
parameter SizeA = 8,
          SizeB = 4;
output [SizeA+SizeB-1:0] mult;
input [Sizea-1:0] a
input [Sizeb-1:0] b;
   assign mult = a * b;
endmodule
```

Verilog 1995

- Verilog 2001 has a ANSI-C style syntax for the parameter definition

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Parameterized modules

- The parameters can be specified right after the **module** keyword and name

```
parameter_declaration := parameter [ signed ] [ range ] list_of_param_assignments;
                       | parameter integer list_of_param_assignments;
                       | parameter real list_of_param_assignments;
                       | parameter realtime list_of_param_assignments;
                       | parameter time list_of_param_assignments;
```

- Parameters can be sized and typed
  - The types of parameters that can be specified include signed, sized (with a range) parameters, as well as parameter types integer, real, realtime, and time.
  - The size of the parameter is decided from the constant itself (32-bits if nothing is specified). Example: `parameter HI = 25, LO = 5; parameter up = 2b'00;`
- Not only does this allow us to reuse the same module definition in more situations, but it allows us to define generic information about the module that can be overridden when the module is instantiated
- Thus the parameterized modules can now be used (instantiated) for multipliers with different widths, simply by changing the default value of the parameters in the calling module

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Overriding parameters by position

- A parameter can optionally be redefined on an instance-by-instance basis and each module instance can have different parameter values
- To override the values of parameters, you can use the # syntax in a module instantiation `module_name #(parameter_values) instance_name(port_connection_list);`
- Parameter re-definition by position

```
multiplier #(8,8) U00(mult, a, b);
```

- The main problem with the positional parameter redefinition is that the parameters must be redefined in the order that they appear in the module definition. For a module with a few parameters, this is error prone
- Also if a module instantiation has to pass only one new value for one of the parameters, all parameter values up to and including all values that are changed, must be listed in the instantiation

```
// illegal parameter passing example
// the module cannot be instantiated with a series
//of commas followed by the new value for one of the parameters
myModule #(,,8) r1 (.q(q), .d(d), .clk(clk), .rst_n(rst_n));
```

```
// the first two parameters must be
// explicitly passed even though the
// values did not change
myModule #(1,1,8) r1 (.q(q), .d(d), .clk(clk), .rst_n(rst_n));
```

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Parametric module example

- This example presents an 8-bit **XOR** module that instantiates eight **XOR** primitives and wires them to the external ports
- The ports are 8-bit scalars; bit-selects are used to connect each primitive
- A parameterized version of this module is shown
- First, we replace the eight XOR gate instantiations with a single assign statement, making this module more generally useful with the parameter specification
- Here we specify two parameters, the width of the module (4) and its delay (10)

```
module xorx # (parameter width = 4,
                         delay = 10)
      (output [l:width] xout,
       input [1:width] xinl, xin2);
 assign #(delay) xout = xin1 ^ xin2;
endmodule
```

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Parametric module example

- When module **xorx** is instantiated, the values specified in the parameter declaration are used. This is a *generic* instantiation of the module
- However, an instantiation of this module may override these parameters
- The **"#(4, 0)"** specifies that the value of the first parameter (width) is 4 for this instantiation, and the value of the second (delay) is 0
- If the **"#(4, 0)"** was omitted, then the values specified in the module definition would be used instead. That is, we are able to override the parameter values on a per module-instance basis

```
module overriddenParameters(input [3:0] b1,c1,b2,c2,
                            output [3:0] a1, a2);
  xorx #(4, 0) a(a1, b1, c1),
               b(a2, b2, c2);
endmodule
```

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Overriding parameters by name

- In positional association, the order of the overriding values follows the order of the parameter specification in the module's definition
- However, the parameters can also be explicitly overridden by naming the parameter at the instantiation site
- Parameter re-definition by name allows inline parameter values to be listed in any order

```
// Generic 2-to-1 MUX using a parameter
module mux2g  #(parameter N = 4)
                (input [N-1:0] a,
                 input [N-1:0] b,
                 input s,
                 output reg [N-1:0] y);
   always @*
         if(s == 0) y = a;
         else y = b;
endmodule
```

```
// 8-bit 2-to-1 MUX using a parameter
module mux2_8bits (input [7:0] a,
                   input [7:0] b,
                   input s,
                   output [7:0] y);
    mux2g #(.N(8)) M8 (.a(a),
                       .b(b),
                       .s(s),
                       .y(y));
endmodule
```

- Or for the previous example we can write:

```
xorx #(.width(4), .delay(0)
    a(a1, b1, c1),
    b(a2, b2, c2);
```

- With the explicit approach, the parameters can be listed in any order
- Those not listed at the instantiation will retain their generic values

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

defparam

- Another approach to overriding the parameters in a module definition is to use the **defparam** statement to re-define parameters values by name
- **defparam** uses *hierarchical naming* conventions to affect the change
- The parameters may be respecified on an individual basis
- The general form of the defparam statement is:

```
parameter_override := defparam list_of_param_assignments;
```

- It is illegal to modify parameter values during simulation
- **defparam** overrides the default parameter values at compile (synthesis) time
- Also, parameter values can be changed at compile time when a module containing parameters is instanced
- The **defparam** statement can be placed before the instance, after the instance or anywhere else in the file
- In the case of multiple **defparams** for a single parameter, the parameter takes the value of the last **defparam** statement encountered in the source text

```
module TestShifter;
  ...
   defparam U1.NBits = 10;
   Shifter U1 (...)
   // the same as Shifter #(10) U1 (…)
endmodule
```

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

defparam *or using module instance method*

- The choice of using the **defparam** or module instance method of modifying parameters is a matter of personal style and modeling needs
- Using the module instance method makes it clear at the instantiation site that new values are overriding defaults
- Using the **defparam** method allows for grouping the respecifications of parameters in one place within the description
- Indeed, the **defparams** can be collected in a separate file and compiled with the rest of the simulation model
- The system can be changed by compiling with a different **defparam** file rather than by reediting the entire description
- Further, a separate program could generate the **defparam** file for back annotation of delays

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

localparam

- Local parameters have a similar declaration style except that the **localparam** keyword is used instead of **parameter**

```
local_parameter_declaration:= localparam [ signed ] [ range ] list_of_param_assignments;
                           | localparam integer list_of_param_assignments;
                           | localparam real list_of_param_assignments;
                           | localparam realtime list_of_param_assignments;
                           | localparam time list_of_param_assignments;
```

- Unlike a **parameter**, a **localparam** cannot be modified by parameter redefinition nor can a **localparam** be redefined by a **defparam** statement
- Since **localparam**s cannot be directly overridden, they are typically used for defining constants within a module
- However, the **localparam** can be defined in terms of **parameters** or **defparam** statements
- Thus The idea behind the **localparam** is to permit generation of some local parameter values based on other **parameters** while protecting the **localparams** from accidental or incorrect redefinition by an end-user
- Since a local parameter assignment expression can contain a parameter (which can be overridden), it can be indirectly overridden

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

localparam *example*

```
module ram1 #(parameter ASIZE=10, DSIZE=8)
              (inout [DSIZE-1:0] data,
               input [ASIZE-1:0] addr,
               input en, rw_n);
// Memory depth equals 2**(ASIZE)
localparam MEM_DEPTH = 1<<ASIZE;
reg [DSIZE-1:0] mem [0:MEM_DEPTH-1];
  assign data = (rw_n && en) ? mem[addr] : {DSIZE{1'bz}};
  always @(addr, data, rw_n, en)
    if (~rw_n && en) mem[addr] = data;
endmodule
```

- The memory depth-size **MEM_DEPTH** is "protected" from incorrect settings by placing the **MEM_DEPTH** in a **localparam** declaration
- The **MEM_DEPTH** parameter will only change if the **ASIZE** parameter is modified

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

`define compiler directive

- `define directive defines a *macro*
- A macro is an identifier that represents a string of text

```
`define <macroName> <textString>
```

- **macroName** will be substituted with **textString** in the first phase of compilation (similar to parameters), at the beginning of synthesis and simulation
- This improves the readability and maintainability of the Verilog code
- Note that a macro definition does not end with a semicolon
- A macro can be invoked with the quoted macro name
- For example, if define **BUS** as `` `define BUS reg [31:0], `` we can use it in the de declaration part as `` `BUS data; ``
- Verilog has the **`undef** compiler directive to remove a macro definition created with the **`define** compiler directive

```
`define BUS_WIDTH 16
 reg [ `BUS_WIDTH - 1 : 0 ] System_Bus;
   …
`undef BUS_WIDTH //`undef removes the previously defined directive
```

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Parameters vs. macros

- A *parameter*, after it is declared, is referenced using the parameter name, however, a **`define** *macro* definition, after it is defined, is referenced using the macro name with a preceding back-tic character
- Parameter declarations can only be made inside of module boundaries, however, macro definitions can exist either inside or outside of a module declaration, and both are treated the same
- Since macros are defined for all files read after the macro definition, using macro definitions generally makes compiling order dependent
- A typical problem associated with using macro definitions is that another file might also make a macro definition to the same macro name
- If the same macro name has been given multiple definitions in a design, only the last definition will be available
- Macro definitions, like all compiler directives, are active from the point of definition and remain active across all files read after the macro definition is made until overridden by a subsequent **`define**, **`undef** or **`resetall** directive

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

More on macros

- You can place all macro definitions in your top-level module so that they are globally available to all files compiled in the design
- You can place all macro definitions into one “macro.vh" file and read the file first when compiling the design (using **`include** compiler directive)
- Only use macro definitions for identifiers that clearly require global definition of an identifier that will not be modified elsewhere in the design
- Do not use macro definitions to define constants that are local to a module
- For example, clock cycles are a fundamental constant of a design

```
`define CYCLE 10
module tb_cycle;
// ...
initial begin
  clk = 1'b0;
  forever #(`CYCLE/2) clk = ~clk;
end
// ...
endmodule
```

- A macro can be defined with arguments
- When invoked, the actual argument expressions will be used

```
`define add(a,b) a + b
f = `add(1,2); // f = 1 + 2;
```

- Macros with arguments are not supported by all synthesis tools

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

`include compiler directive

- **Include** compiler directive is used to include the contents of a text file at the point in the current file where the include directive is

```
`include “filename”;
```

- You can write it anywhere in the code

```
`include “my_macros.vh”
module top
   ...
endmodule
```

- Since the defines in **my_macros.vh** are put into a global namespace, it makes sense never to include or redefine those again

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

ifdef compiler directive

- Verilog-1995 supports conditional compilation, using the **`ifdef**, **`else** and **`endif** compiler directives
- **`ifdef** conditionally compiles Verilog code, depending on whether or not a specified macro is defined

```
`define behavioralModel
 module Test;
        ...
  `ifdef behavioralModel
     MyDesign_behavioral UUT (...);
   `else
     MyDesign_RTL UUT (...);
   `endif
endmodule
```

- If the macro name has been defined using **`define**, only the first block of Verilog code is compiled and if an **`else** directive is present, the second block only is compiled
- Therefore, **`ifdef** can be used to switch between alternative implementations of a module, or to selectively turn on the writing of diagnostic messages
- These directives may be nested
- Any code that is not compiled must still be valid Verilog code

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

`ifndef

- Verilog-2001 adds more extensive conditional compilation control, with **`ifndef** and **`elsif** compiler directives
- The **`ifndef/`endif** clause prevents redefinition (or inclusion) of the file's contents (if this same file was already included earlier)

```
`ifndef _my_macro_vh_
// If we have not included file before, this symbol _my_macro_vh_ is not defined
`define _my_macro_vh_
  `include "my_macros.vh"
`endif //_my_macro_vh_
```

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Process model

- Typically, the dataflow model using continuous assignment statements is used when a combinational function can be described using a few simple assign statements
- More complex combinational functions are typically easier to describe with a combinational always statement at the behavioral level
- The basic essence of a behavioral model is the *process*
- We represent the behavior of digital systems as a set of these independent, but communicating processes
- A process can be thought of as an independent thread of control, which may be quite simple or very complex
- In the *initialization* phase, each signal is given its initial value, simulation time is set to 0, the simulation cycle enters the suspended state and waits for events
- When some events occur on one or more signals, each process that was suspended waiting on a signal event enters the *execution* (activate) state, which usually involves scheduling transactions on signals for later times
- At the end of always statement, the process enters the *suspended* state

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Procedural blocks

- A *procedural block* defines a region of code containing *sequential* statements and the statements execute in the order they are written
- Two types of procedural blocks in Verilog
- An **always** statement is an infinite loop that never terminates
  - When the statement is completed, it returns to the beginning and starts over (if there is not sensitivity list)
- The **initial** block is similar to the **always** statement except that it is executed only once at the beginning of the simulation
  - When it is completed, it does not repeat; rather it becomes inactive
  - The **initial** provides a means of initiating input waveforms and initializing simulation variables before the actual description begins simulation
- Although it is possible to mix the description of behavior between the **always** and **initial** statement, it is more appropriate to describe the behavior of the hardware in the **always**, and describe initialization for the simulation in the **initial**
  - The "**initial**" block is not synthesizable and is commonly used in testbenches for applying stimuli to the DUT
  - **always** statements model the <u>continuous operation of hardware</u>

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Compound statements

- A module can have multiple **`initial`** and **`always`** statements, but they cannot be nested
- All "**`always`**" blocks execute concurrently
  - Thus, concurrent/overlapping behavior is modeled
- An **`always`** or an **`initial`** block may consist of a single statement or a *block statement*
- A block statement begins with **`begin`** and ends with **`end`**
- Statements within a block statement execute sequentially
- Even though the statements in an **`always`** or **`initial`** block are executed in order, it is possible that statements from other **`always`** or **`initial`** blocks will be interleaved with them
  - When an **`always`** or **`initial`** block is waiting to continue (due to @, #, or **`wait`** statement), other **`always`** or **`initial`** blocks, gate primitives, and continuous **`assign`** statements can execute
  - When using **`always`** or **`initial`** statements, we should be thinking conceptually of concurrently active processes that will interact with each other

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Procedural assignment statements – blocking assignments

- HDL supports two procedural assignment statements that can be used in "**`always`**" and "**`initial`**" blocks: *blocking* and *nonblocking assignments*
- The blocking assignment operator is an equal sign "=" and the nonblocking assignment operator is "<="
- A blocking assignment "blocks" trailing assignments in the same **`always`** block from occurring until after the current assignment has been completed
  - Hence, the left hand side (LHS) operand of a blocking assignment gets updated before the next sequential statement in the procedural block is executed
- Execution of blocking assignments is a one-step process: Evaluate the RHS (right-hand side argument) and update the LHS of the blocking assignment without interruption from any other Verilog statement
- A group of blocking assignments are evaluated in the order they appear in the code

```
module fulladder (input a, b, cin,
                  output reg s, cout);
reg p, g;
always @ (*) begin
  p    = a ^ b; // blocking
  g    = a & b; // blocking
  s    = p ^ cin; // blocking
  cout = g | (p & cin); // blocking
end
endmodule
```

- Assignments made using the *blocking assignment* ("=") take effect immediately and the value written to the left-hand side of the = is available for use in the next statement

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Procedural assignment statements – Nonblocking assignments

- The nonblocking assignment does not block trailing Verilog statements from being evaluated
- Evaluation of nonblocking assignments can be viewed as a two-step process:
  - 1. Evaluate the right-hand side (RHS) argument of all nonblocking statements concurrently *at the beginning* of the current simulation cycle (time step)
  - 2. Update the LHS of nonblocking statements *at the end* of the current time step
- Nonblocking assignments will be executed in the order that are written in an **`always`** block
- When assigning multiple values to same variable using nonblocking assignments, the last nonblocking assignment wins!
- In this example, when this block is executed, there will be two events added to the signal driver's queue at time step zero. At the end of time-step 0, the variable "a" will be assigned 0 and then 1

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

The order of nonblocking assignment statements is important only when assigning to the same signal

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Blocking and non-blocking assignments

- It is most efficient to use *blocking* assignments in **`always`** blocks that are written to generate *combinational logic*
- It is most efficient to use *nonblocking* assignments in **`always`** blocks that are written to generate *sequential and latching logic*
- Ignoring the above guidelines can still infer the correct synthesized logic, but the pre-synthesis simulation might not match the behavior of the synthesized circuit
- Note that *continuous signal assignment statements* using the **`assign`** keyword are used outside **`always`** statements
- Comparing to concurrent signal assignment statements, *concurrent assignment* statements change the value of the target *net* whenever the right-hand-side operands change value
- However, a *procedural assignment* changes the target **`reg`** only when the assignment is executed according to the sequence of operations
- **`assign`** `y <= a + b;` has a syntax error
- **`assign`** `out = a <= b;` is a correct statement

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Events and event control

- An event is a change in a variable and the change may be a positive edge, a negative edge, or a level change
- The event control can be described using the name of signals (representing level change) or described using the *edge specifiers*: **posedge** or **negedge** of a signal
- An event control (**@posedge** or **@negedge**) before a statement causes the execution of the immediately following statement to be delayed (similar to timing control using #)

```
@(CLK) Q = D;          // assignment will be performed whenever
                          signal CLK changes to its value
@(posedge CLK) Q = D;  // assignment will be performed whenever
                          signal CLK has a rising edge (0→1, 0→X,
                          0→Z, X→1, Z→1)
@(negedge CLK) Q = D;  // assignment will be performed whenever
                          signal CLK has a falling edge (1→0, 1→X,
                          1→Z, X→0, Z→0)
```

- Note that the event control expression may take on unknown values

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Always statement

- The always block is triggered to execute by the level or the edge (transition) of one or more signals

```
always @ (posedge variable or negedge variable) statement;
always @ (variable, variable, . . .) statement;
```

- Example:

```
always @(a or b)       // level-triggered; if a or b changes levels
always @(posedge CLK)  // edge-triggered: on positive edge of CLK
```

- The **always** @ (<signal>) or @* or @(*) synthesizes to a combinational logic or to a sequential logic
- The **always** @ (<pos|neg>edge <signal>) synthesizes to a sequential logic

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Procedural statements: **if** statement

- Conditional statements are used in a sequential behavior description to alter the flow of control
- The **if** statement and its variations are common examples of conditional statements
- An if-then-else statement is used to conditionally execute sequential statements based on the value a Boolean expression

```
if (expression) begin
... Procedural Statements ...
end
else if (expression) begin
... Procedural Statements ...
end
...more else if blocks ...
else begin
... Procedural Statements ...
end
```

- Statements associated with the true condition are then executed and the rest of the statement is ignored
- If more than one statement is required to be executed in either the **if** or the **else** branch, the statements must be enclosed in a **begin-end** block
  - If there is one statement in a block, then the **begin ... end** statements may be omitted
- Both the **else if** and **else** statements are optional
- There can be as many **else if** statements as required, but only one **if** block and one **else** block

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

else association

- An **else** is associated with the immediately preceding **if**, unless an appropriate **begin-end** is present
- In this example, the **begin-end** block in the first **if** statement causes the **else** to be paired with the first **if** rather than the second
  - When in doubt about where the **else** will be attached, use **begin-end** pairs to make it clear

```
if (expressionA)
 if (expressionB)
   a = a + b;
 else
   q = r + s;
```

```
if (expressionA) begin
 if (expressionB)
   a = a + b;
end
else
q = r + s;
```

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

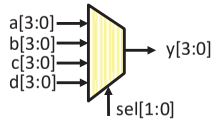
Example: `if` statement

```
module mux_2x1(input a, b, sel,
               output reg out);
   always @* begin
         if (sel == 1) //if (sel)
              out = a;
         else out = b;
   end
endmodule
```

Assignments within if statements generally synthesize to multiplexers

```
module mux4x1_1bit(output reg out,
                   input [3:0] in,
                   input [1:0] sel);
always @*
        if (sel == 0)      out = in[0];
        else if (sel == 1) out = in[1];
        else if (sel == 2) out = in[2];
        else out = in[3];
endmodule
```

```
module mux4x1_4bits
(input [3:0] a, b, c, d,
 input [1:0] sel,
 output reg [3:0] y);
  always @*
     if (sel == 0) y = a;
     else if (sel == 1) y = b;
     else if (sel == 2) y = c;
     else if (sel == 3) y = d;
     else y = 4'bx; --sel can be X or Z
endmodule
```



These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Some notes

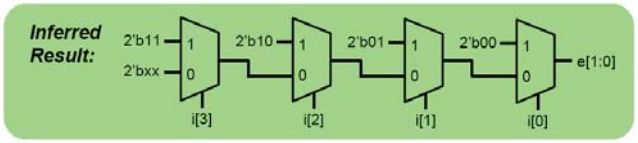
- Note that comparison with an unknown (**x**) or high impedance (**z**) may produce a result that is either unknown or high impedance; these are interpreted as FALSE
  - Thus the expression is considered to be true if it is non-zero, and false if it is zero, **X** or **Z**
- If the conditional expression evaluates to false, then the statements in the **`else`** block, if present, are executed
- Note that the conditional operator may appear in an expression that is either part of a <u>procedural</u> assignment in the behavioral modeling or in <u>continuous assignment statements in the dataflow level modeling</u>

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

4-to-2 binary encoder

- Beware of unintended priority logic when using if statements
- A set of nested **`if-else`** statements can be used to give *priority* to the conditions
- Will this code be synthesized to a 4-to-2 priority encoder?
- Each condition of the **`if-then-else`** statement is checked in order against that value until a true condition is found
- If **`i[0]`** is **1**, the result is **00** regardless of the other inputs. So **`i[0]`** takes the highest priority

```
module binary_encoder (input [3:0] i,
                       output reg [1:0] e);
 always @* begin
  if (i[0]) e=2'b00;
  else if (i[1]) e=2'b01;
  else if (i[2]) e=2'b10;
  else if (i[3]) e=2'b11;
  else e=2'bxx;
 end
endmodule
```



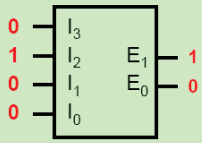
- Priority encoded muxes can impact timing due to their cascading structural nature
  - A priority mux can also be used to speed up a design by placing the late arriving operand and its condition at the top of the **`if`** statement – that way it travels through the least number of levels of logic

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Binary encoder

- If mutually-exclusive conditions are chosen for each branch, then the synthesis tool can generate a simpler circuit that evaluates the branches in parallel

4-to-2 Binary Encoder

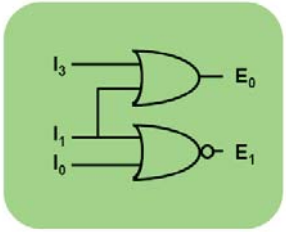


I_3 I_2 I_1 I_0	E_1 E_0
0 0 0 1	0 0
0 0 1 0	0 1
0 1 0 0	1 0
1 0 0 0	1 1
all others	X X

```
module binary_encoder (input [3:0] i,
                       output reg [1:0] e);
 always @* begin
  if (i==4'b0001) e=2'b00;
  else if (i==4'b0010) e=2'b01;
  else if (i ==4'b0100) e=2'b10;
  else if (i ==4'b1000) e=2'b11;
  else e=2'bxx;
 end
endmodule
```

Assuming only one of the bits of i is 1, these are mutually-exclusive conditions

Minimized result



These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

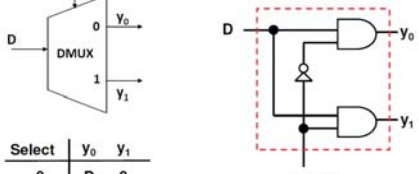
Rules for synthesizing combinational circuits

- Within an **`always`** statement, we define a *control path* to be a sequence of operations performed when executing an **`always`** loop
- There may be many different control paths in an always block due to the fact that conditional statements (e.g. **`if`**) may be used
- **(1)** To produce a combinational circuit using procedural statements, the output of the combinational function must be assigned in each and every one of the different control paths
  - Thus, for every input change, the combinational output will be calculated
- Thus in a combinational circuit behavioral description, a LHS variable must be assigned a value at least once in every execution of the **`always`** loop
- **(2)** Make sure that all inputs to your combinational function are listed in the control event's sensitivity list (the comma-separated list of names)
  - Therefore, if one of them changes, the output is re-evaluated
  - The need for this requirement stems from the definition of a purely combinational circuit
  - The output of a combinational circuit is a function of the current inputs; if one changes, the output should be re-evaluated

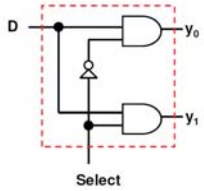
These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Example: Demultiplexer

```
module demux (input D, select,
              output y0,y1);
   assign y0 = (~select) & D;
   assign y1 = select & D;
endmodule
```



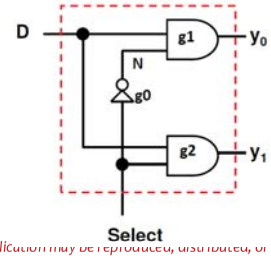
Select	y_0	y_1
0	D	0
1	0	D




```
module demux (input D, select,
              output reg y0,y1);
   always @( D or select ) begin
      if( select == 1'b0) begin
          y0 = D;
          y1 = 1'b0;
      end
      else begin
          y0 = 1'b0;
          y1 = D;
      end
   end
endmodule
```

Note that a latch will be inferred if a variable is not assigned to for all the possible branch conditions

```
module demux (input D, select,
              output y0,y1)
wire N;
   and g1(y0, D, N);
   and g2(y1, D, Select);
   not g0(N, Select);
endmodule
```



These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Latch inferrence

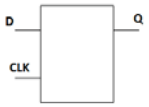
- If the **`always`** block is executed and no value is assigned to the output, the circuit needs to remember the previous value
  - Thus, the output is a function of the current inputs *and* the previous output
  - This is a fundamental characteristic of a *sequential circuit*, not a combinational one
  - A synthesized version of such a circuit will have storage elements to implement the sequential nature of the description
- Therefore, if there exists a control path that does not assign to the output, then the previous output value needs to be remembered
  - This is not a characteristic of combinational hardware. Rather it is indicative of a sequential system where the previous state is remembered in a latch when the inputs specify this control path
  - This causes *latch inference*
- Assuming that we are trying to describe a sequential element, leaving the output variable unassigned in at least one control path will cause a latch to be inferred

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

D-Latch

- Latches are *level-sensitive* storage devices
- Recall that a *D* latch is transparent when the clock is high, allowing data to flow from input to output. The latch becomes opaque when the clock is low, retaining its old state

```
module dLatch (input CLK, D,
               output reg Q);
   always @(D or CLK) // or always @*
       if (CLK)
           Q = D;
endmodule
```



- This code infers a latch, because the output, **`Q`**, is not assigned under all possible conditions
- To prevent syntheiszer from inferring unintentional latches for these examples, you should make a default assignment to **`Q`** outside the **`if`** statement or add an **`else`** branch to the **`if`** statement

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Inferring a latch

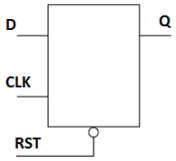
- Thus to infer a latch, two situations must exist in the **always** statement: at least one control path must exist that does not assign to an output, and the sensitivity list must not contain any edge-sensitive specifications
- The first gives rise to the fact that the previous output value needs to be remembered
- The second leads to the use of *level-sensitive* latches as opposed to *edge-sensitive* flip flops
- A logic synthesis tool will recognize this situation and infer that a latch is needed in the circuit

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

D-latch with other control signals

- Note that set and/or reset inputs may change the flip flop state either synchronously or asynchronously with respect to the clock
- The tests for the set and reset conditions are done first in the **always** statement using **if** constructs
- After all of the set and resets are specified, the final statement specifies the action that occurs on the latch is transparent

```
module dlatchwReset (input RST, CLK, D,
                     output reg Q);
   always @ (RST or CLK or D)
     if (~RST)
        Q = 1'b0;
     else if (CLK)
        Q = D;
endmodule
```



D-latch with asynchronous reset

```
module latchwPreset (input CLK, PRE,
                     input [3:0] D,
                     output reg [3:0] Q);
  always @(CLK or D or PRE) begin
     if (PRE) Q = 4'b1111;
     else if (~CLK) Q = D;
  end
endmodule
```

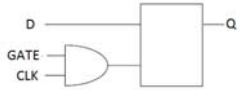
//4-bit latch with active-low clock and asynchronous preset

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Compound latches

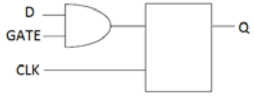
- The synthesized latch does not need to be a simple D-latch; other functionality can be included

```
module dlatch (input CLK, GATE, D,
               output reg Q);

   always @ (CLK or D or GATE)
         if (CLK & GATE)
            Q = D;
endmodule
```



D-latch with gated enable

```
module dlatch (input CLK, GATE, D,
               output reg Q);

   always @(CLK or D or GATE)
   //the same as always@(CLK, D, GATE)
         if (CLK)   Q = (D & GATE);
endmodule
```



D-Latch with gated data

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Some important points

- The values computed can be held in a 'wire', a 'flip-flop' (edge-triggered storage cell) or a 'latch' (level-sensitive storage cell)
- A variable in Verilog can be of
  - 'net data type: Maps to a 'wire' during synthesis
  - 'register' data type: Maps either to a 'wire' or to a 'storage cell' depending on the context under which a value is assigned
- Synthesis tools usually infer latches and flip-flops from **always** blocks, but not from continuous assignments
- Incompletely specified **if** statements cause synthesis tool to infer latches
- Here an adder/subtractor capable of adding and subtracting is synthesized with an output latch

```
module addSub #(parameter Width = 4)
               (output reg [Width-l:0] out
                input [Width-l:0] a, b,
                input EN, addsub);
  always @(*)
    if (EN) begin
      if (addsub) out= a+b;
      else out = a-b;
    end
endmodule
```

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Some important points

- For combinational logic and latches, the sensitivity list must be the input set and contain no edge-sensitive specifiers
- *Be aware of incomplete sensitivity lists*: Synthesis tool may issue warnings for signals that are read in an **always** block but are not listed in the sensitivity list
- In this example, the signal **EN** is read, but it is not in the sensitivity list
- Assuming that **RST** is stable at 0, a change in **EN** from **0** to **1** does not trigger the **always** block, so the value of **D** does not get latched onto **Q**

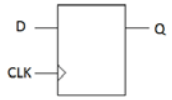
```
always @(D or RST)
  if (RST)
    Q = 1'b0
  else if (EN)
    Q = D;
```

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Flip-flops

- Flip flops are *edge-triggered* storage devices
  - Their behavior is controlled by a positive or negative edge that occurs on a special input, called the *clock*
- The main characteristic of a flip flop description is that the event expression in the **always** statement specifies an edge
- Edge-triggers are specified by **posedge** and **negedge** keywords
- When the edge event occurs, the input **data** is passed to the output

```
module dff (input D, CLK,
            output reg Q);

  always @(posedge CLK)
     Q <= D;
endmodule
```



- **@(posedge CLK) Q <= D;** This procedural event control statement watches for the positive transition of **CLK** and then assigns the value of **D** to **Q**
- The value assigned to **Q** is the value of **D** <u>just before the positive edge of the clock</u>
- Since we are describing a D flip-flop, a change on **D** will not change the flip flop state
  - So the **D** input is not included in the sensitivity list

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

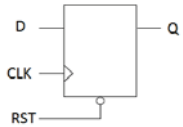
D flip-flops with reset and preset

- Typically flip flops include reset signals to initialize their state at system start-up
- If a negative edge was specified, then the test should be:

  if (~reset) … or if (reset == 1'b0) …
- If a positive edge was specified, then the test should be:

  if (set)… or if (set == l'b1) …

```
module dff_sync_rst (input D, CLK, RST,
                     output reg Q);

  always @(posedge CLK)
     if (~RST)  Q <= 1'b0;
     else       Q <= D;
endmodule
```



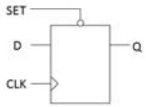
dff with synchronous reset

- Note that the sensitivity list of the always block includes <u>only the edges for the clock, reset and preset conditions</u>
- These are the only inputs that can cause a state change

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

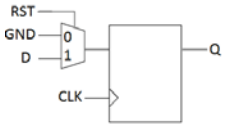
Examples

```
module dff_sync_pre (input D, CLK, SET,
                     output reg Q);

   always @(posedge CLK)
        if (!SET)  Q <= 1'b1;
        else       Q <= D;
endmodule
```



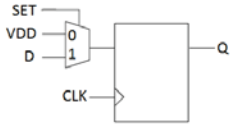
flip-flop with synchronous preset

```
module dff_sync_rst (input D, CLK, RST,
                     output reg Q);
   always @(posedge CLK)
     if (!RST) Q <= 1'b0;
     else      Q <= D;
endmodule
```



flip-flop with synchronous reset

```
module dff_sync_pre (input D, CLK, SET,
                     output reg Q);
   always @ (posedge CLK)
     if (!SET) Q <= 1'b1; //active-low preset
     else      Q <= D;
endmodule
```

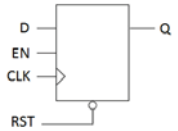


flip-flop with synchronous preset

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

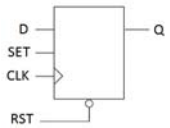
D flip-flops with asynchronous reset and preset

```
module dff_async_nrst (input D, CLK, RST, EN,
                       output reg q);

   always @ (posedge CLK or negedge RST)
        if (!RST)  Q <= 0;
        else if (EN) Q <= D;
endmodule
```



flip-flop with asynchronous reset and clock enable

```
module dff_async_nrst (input CLK, RST, SET, D,
                       output reg Q);

   always @ (posedge CLK or negedge RST or posedge SET)
     if (!RST)     Q <= 0;
     else if (SET) Q <= 1;
     else          Q <= D;
endmodule
```



flip-flop with asynchronous reset and preset

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Some important points

- The expressions for set and reset cannot be indexed; they must be one-bit variables
- Make sure that your HDL does not imply any unintended latches
- Many synthesis tools warn you if a latch is created; if you didn't expect one, debug your design
- For **posedge** and **negedge**, only the least significant bit of the expression is tested
- For synthesis, one cannot combine level and edge changes in the same list
- Single flipflop modules will work with blocking assignments – Bad habit - better to consistently code sequential logic with nonblocking assignments
- `always @(CLK or D or GATE)` is the same as `always@(CLK, D, GATE)`

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Synchronous or asynchronous?

- If you use asynchronous reset, the main issue is not when the asynchronous reset goes active (since the circuit is going to reset anyway), but rather when it goes inactive
- As soon as the asynchronous reset goes inactive, the flip-flop is free to change its state when the next clock edge occurs
- If the reset signal is distributed throughout your design without care to how long the delay is on the reset network, you can have parts of the chip working in active mode while other parts of the chip are still in reset mode
- You have to treat the asynchronous reset signal similar to a clock signal and balance it so that the signal goes inactive within the same clock cycle throughout the entire chip

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

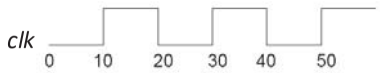
Synchronous testbench

- Synchronous testbenches are used for cycle based simulations of synchronous logic, which do not use any delays smaller than a clock cycle
- Every clock cycle, a test vector is applied to the DUT
- How to generate a repetitive clock signal CLK?

```
wire CLK;
assign #10 CLK = ~ CLK;
```

- Why this does not work?
  - This is because the initial value of **CLK** (**wire** data type) is **z** (**~z = x** and **~x = x**)
  - CLK has to be defined as type **reg** in order to be used in an **initial** statement. **CLK** of data type **wire**, cannot be used in an **initial** statement

```
reg clk =0;
  always begin
    #10 clk = 1;
    #10 clk = 0;
  end
```



CLK signal with 20ns clock cycle

- This code enters an **always** loop, where it initializes the clock to 1 at time 10 and thereafter toggles its value every 10 time units

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

forever statement

- The **forever** statement loops forever

```
reg CLK, RST, EN;
reg [7:0] data;
parameter HalfPeriod = 5; // parameter defines a local constant value
```

```
initial begin : ClockGenerator
  CLK = 0;
  forever // execute one or more statements indefinitely
    #(HalfPeriod) CLK = ~CLK;
end

initial begin
  EN = 1'b0; RST = 1'b1; // activating the reset RST signal
  #(2*HalfPeriod) RST  = 1'b0; EN = 1'b1;
   //releasing reset after 10 time units
  #(2*HalfPeriod) data = 8'haa
  #(2*HalfPeriod)
 disable ClockGenerator; //disable can be called in another
                         //initial statement
end
```

- A forever loop can include a **disable** statement to disable itself
- Note that **disable** task is similar to the C break statement except it can terminate any loop, not just the one in which it appears

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

disable system task

- Generally, a loop statement is written to execute to a "normal" exit; the **for** loop counter is exhausted or the **while** expression is no longer TRUE
- However, any of the loop statements may be exited through use of the **disable** statement
- **disable** system task terminates any named blocks (using **begin** and **end**), tasks, modules, or any loop statements and passes control to the next statement following the block

```
disable block_name;
```

- A **disable** task can only be used with named **begin-end** blocks
- **begin-end** blocks can be named by placing the name of the block after a colon following the **begin** keyword

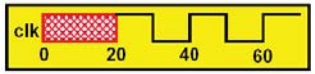
```
begin
i = 0;
forever begin : continue_block
  if (i==a) disable continue_block;
  #1 i = i + 1;
  end
end
```

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Clock generation

- Symmetric clock with delayed startup

```
//clock generation with variable start time
reg clk;
  initial begin
    #20 clk = 1;
    forever begin
      #10 clk = 0;
      #10 clk = 1;
    end
  end
```




```
forever [begin]
... Procedural statements ...
[end]
```

- The **forever** statement causes one or more statements to be executed in an indefinite loop

```
// Declare a constant clock period
parameter ClockPeriod = 10;
reg CLK;

initial CLK = 0; // initialize the Clock signal
always #(ClockPeriod / 2) CLK = ~CLK;
```

```
localparam T=10; // clock period
//reset for the first half cycle
initial begin
  reset = 1 'bl;
  #(T/2);
  reset = l'bO;
end
always begin
 clk = l'bl;
 #(T/2) ;
 clk = l'bO;
 #(T/2) ;
end
```

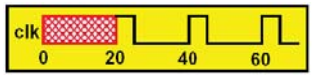
- Not including any delay control or event control in an **always** may cause infinite loop in the simulator

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Clock generation using forever statement

- Asymmetric clock with delayed startup
- This code initializes the clock to 1 at time 20 and thereafter toggles its value in a forever loop with a 5/15 duty cycle

```
reg clk;
  initial begin
    #20 clk = 1;
    forever begin
      #5 clk = 0;
      #15 clk = 1;
    end
end
```




```
initial begin
   forever
     CLK = #(ClockPeriod / 2) ~ CLK;
end
```

- The **forever** statement is not generally synthesizable
- To avoid combinational feedback during synthesis, a **forever** loop must be controlled with an @(**posedge/negedge** clock) statement

```
forever begin
  @(posedge clk);
  a = a + 1;
end
```

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

for statement

- Iterative sequential behavior is described with looping statements: **repeat**, **for**, **while**, and **forever**
- **For** loops are used to repeatedly execute a statement or block of statements

```
for (index = init; index </<=/>/>= limit; index = index +/- step)
[begin]
    ... Procedural statements ...
[end]
```

- The **for** loop is highly structured and very similar in function to for loops in the C programming language
- The first assignment, which is the initialization of an index (counter) variable, is executed once at the beginning of the loop
- The second expression is executed before the body of the loop to determine if we are to stay in the loop. Execution stays in the loop while the expression is TRUE
- The comparison for end of loop may be <, >, <=, or >=
- The step variable and the value of the loop count expression (`limit`) are determined once at the beginning of the execution of the loop
- Then the loop is executed the given number of times

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

for statement

- The loop counter is updated after every execution of the body of the loop and before the next check for the end of the loop
- Note that step size **step** need not be one
- The index must either start with a low limit and step up to a high limit, or start with a high limit and step down to a low limit
- If the loop contains only one statement, the **begin ... end** statements may be omitted
- It is not possible to exit the loop execution by changing the loop count variable
- The **disable** task allows for early loop exits (will be discussed)

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

For statement example

- A large N:2^N decoder is cumbersome to specify with case statements, but easy using parameterized code that simply sets the appropriate output bit to 1
- Specifically, the decoder sets all the bits to 0, and then changes the appropriate bit to 1

```
module decoder #(parameter N = 3)
                (input [N-1:0] a,
                 output reg [2**N-1:0] y);
  always @* begin
     y = 0;
     y[a] = 1;
  end
endmodule
```

```
module decoder_index #(parameter N = 8,
                       log2N = 3)
                 (input [log2N-1:0] in1,
                 output reg [N-1:0] out1);
always @(in1) begin
    out1 = 0;
    out1[in1] = 1'b1;
  end
endmodule
```

Decoder using indexing

```
module decoder38_loop
        #(parameter N = 8,
                    log2N = 3)
         (input [log2N-1:0] in1,
          output reg [N-1:0] out1);
integer i;
  always @(in1) begin
    for(i=0;i<N;i=i+1)
      out1[i] = (in1 == i);
    end
endmodule
```

Decoder using for loop

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Applying stimuli from loops

```
module loop_tb;
wire [7:0] response;
reg [7:0] stimulus;
reg clk;
integer i;

  DUT u1 (response, stimulus, clk);

  initial clk = 0;
  always begin
    #10 clk = 1;
    #10 clk = 0;
  end

  initial begin
    for (i = 0; i <= 255; i = i + 1)
      @(posedge clk) stimulus = i;
    #20 $finish; //specifies the end of simulation
  end

endmodule
```

- Using a for loop, the testbench is more compact
- For each iteration a new stimulus vector is applied after a time delay

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Describing combinational logic using **for** loops

- The **for** loop in Verilog may be used for repetitive specification of a combinational logic
- Synthesis tool unrolls the for loops and implement repeated hardware structures, provided the loop bounds are fixed

```
for (init_assignment; cond; step_assignment)
        procedural_statements;
```

```
module Xor8 (output reg [1:8] xout,
             input [1:8] xin1, xin2);
reg [1:8] i;
always @*
  for (i = 1; i <= 8; i = i + 1)
    xout[i] = xin1[i] ^ xin2[i];
endmodule
```

- In this example, each iteration of the loop specifies a different logic element indexed by the loop variable **i**
- Thus, eight **xor** gates are connected between the inputs and the outputs
- Since this is a specification of combinational logic, **i** does not appear as a register in the final implementation

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.

Examples

```
module paritygen (InP, par);
`define input_width 8
input [`input_width - 1:0] InP;
output reg par;
integer J;
  always @(InP) begin
   par = 0;
   for ( J=0; J < `input_width; J=J+1)
     par = par ^ InP[J];
  end
endmodule
```

```
module parity #(parameter WIDTH = 2)
               (input [WIDTH-1 : 0] in,
                output reg p);
  always @(in) begin: loop
    integer i;
    reg parity = 0;
    for (i = 0; i < WIDTH; i = i + 1)
      parity = parity ^ in[i];
    p = parity;
  end
endmodule
…
reg [3:0] word;
wire parity;
parity #(.WIDTH(4)) ecc (.in(word),.p(parity));
```

- A simple approach: **assign** `p = ^in;`
- The above are more general approaches

These notes are copyrighted and are strictly for 2017 courses at SDSU. No part of this publication may be reproduced, distributed, or transmitted.