Introduction To Verilog for Combinational Logic

- Verilog is a language used for simulation and synthesis of digital logic.
- A New Extension “System Verilog” also Supports new features including Verification of Digital Systems
- A Verilog description of a digital system can be transformed into a gate level implementation. This process is known as synthesis.

General Module Structure

module module_name [(port_name{, port_name})];
[**parameter** declarations]
[**input** declarations]
[**output** declarations]
[**inout** declarations]
[**wire** or **tri** declarations]
[**reg** or **integer** declarations]
[**function** or **task** declarations]
[**assign** continuous assignments]
[**initial** block]
[**always** blocks]
[gate instantiations]
[module instantiations]
endmodule

Figure A.1. The general form of a module.

Verilog Statements

- We will only examine a subset of the language
  - RTL – Synthesizable Portion
- Some Verilog constructs:
  - Signal Assignment: `assign A = B;`
  - Comparisons `==` (equal), `>` (greater than), `<` (less than), etc.
  - Boolean operations `&` (AND), `|` (OR), `~` (NOT), `^` (XOR)
  - Concurrent statements
    - Gate Instantiations: `and u1 (X, A, B);`
    - Continuous Assignments `assign S = X & Y;`
  - Procedural (Sequential) statements
    - Evaluated in order in which written
    - Must be Contained in an `always` or `initial` block

Some Verilog Syntax

- Approximately 100 keywords (lowercase)
  - Verilog IS case-sensitive
  - Predefined identifiers Used for Basic Language Constructs
- Comments are:
  - `//` to end of line
  - `/* comment here */`
- Simulator Directives
  - Technically not part of language, but Standard
  - Begin with a `$`
  - Example `$finish;`
- Simulator Directives not used for:
  - Documentation
  - Synthesis

Verilog Keywords (part 1)

always	ifnone	rnmos
and	incdir	rpmos
assign	include	rtran
automatic	initial	rtranif0
begin	inout	rtranif1
buf	input	scalared
bufif0	instance	showcancelled
bufif1	integer	signed
case	join	small
casex	large	specify
casez	liblist	specparam
cell	library	strong0
cmos	localparam	strong1
config	macromodule	supply0
deassign	medium	supply1
default	module	table
defparam	nand	task
design	negedge	time
disable	nmos	tran
edge	nor	tranif0
else	noshowcancelled	tranif1

*excerpt from IEEE 1364-2005 standard for academic use only

Verilog Keywords (part 2)

end	not	tri
endcase	notif0	tri0
endconfig	notif1	tri1
endfunction	or	triand
endgenerate	output	trior
endmodule	parameter	trireg
endprimitive	pmos	unsigned[1]
endspecify	posedge	use
endtable	primitive	uwire
endtask	pull0	vectored
event	pull1	wait
for	pulldown	wand
force	pullup	weak0
forever	pulsestyle_onevent	weak1
fork	pulsestyle_ondetect	while
function	rcmos	wire
generate	real	wor
genvar	realtime	xnor
highz0	reg	xor
highz1	release	
if	repeat	

[1] **unsigned** is reserved for possible future usage.

*excerpt from IEEE 1364-2005 standard for academic use only

Some Verilog Operators

Reduction Operators

`~`	negation
`&`	bitwise AND
`|`	bitwise OR
`~&`	bitwise NAND
`~|`	bitwise NOR
`^`	bitwise XOR
`~^`	bitwise XNOR
`^~`	bitwise XNOR

Arithmetic Operators

`+`	unary (sign) plus
`-`	unary (sign) minus
`+`	binary plus (add)
`-`	binary minus (sub)
`*`	multiply
`/`	divide
`%`	modulus

Logical Operators

`!`	logical negation
`==`	logical equality
`!=`	logical inequality
`&&`	logical AND
`||`	logical OR

Verilog Values and Constants

Four Basic Values

`0`	logic-0 or false
`1`	logic-1 or true
`x`	unknown value
`z`	high-impedance (open)

(`z` at input usually treated as `x`)

Constants

integers
reals
strings

_ can be embedded

Specifying Values

Simple Decimal
int, real
Base Format Notation
int
Scientific
real
Double Quotes
strings

Base Format Notation Examples

Format is: `[size (in bits)]'base value`

`5'O37`	5-bit octal
`4'D2`	4-bit decimal
`4'B1x_01`	4-bit binary (underscores "_" ignored)
`7'Hx`	7-bit `x` (`x` extended) `xxxxxxx`
`4'hZ`	4-bit `z` (`z` extended) `zzzz`
`4'd-4`	ILLEGAL: value cannot be negative
`8 'h 2A`	spaces allowed between size and ' and between base and value
`'o721`	9-bit octal
`'hAF`	8-bit hex
`10'b10`	10-bit padded to left `0000000010`
`10'bx0x1`	10-bit padded to left `xxxxxxx0x1`
`3'b1001_0011`	same as `3'b011`
`5'H0FFF`	same as `5'H1F`

Verilog Data Types

- Net Types (eg. **`wire`**)
  - Represents Physical Connection Between Structural Elements
  - Value is Determined from Value of Drivers
    - Continuous **`assign`** Statement
    - Output of Gate or UDP (User Defined Primitive)
    - If no Driver is Present, Defaults to value of **`z`**
- Register Type (eg. **`reg`**)
  - Abstract Data Storage Element
  - Assigned Values Only within **`always`** or **`initial`** statement
  - Does not ALWAYS synthesize to storage device
  - Value is Saved from one Assignment to the Next
  - Default value is **`x`**

Verilog Data Types

- Net Types

wire, tri	- most common, default is z
wor, trior	- emulates wired-OR with mult. drivers (ECL NOR)
wand, triand	- emulates wired-AND with mult. drivers (OC-TTL NAND)
trireg	- stores a value like **reg** for modeling capacitive nets
tri1, tri0	- default values are 1(0)
supply0,supply1	- used to model power connections for 0 and 1 values

- Register Type

reg	- most common, default is **x**
integer	- used for storing integers, typical use in behavioral model
time	- used for storing/manipulating time values
real	- used storing reals, typical use in behavioral model
realtime	- same as real

4-Valued Net Fanin Tables

wire/ tri	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	z

- Two Names for Same Data type

Wired Nets

wand/ triand	0	1	x	z
0	0	0	0	0
1	0	1	x	1
x	0	x	x	x
z	0	1	x	z

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

- Used to Model Wired Logic
- Assumes Equal Strength Drivers

Driver Strength

- Not Used in this Class for Synthesis Purposes
- Created for Accurate Modeling of Devices Such as:
  - signal contention
  - bidirectional pass gates
  - resistive MOS
  - dynamic MOS
  - charge sharing
- strength0 part of net:

  `supply0 strong0 pull0 weak0 highz0`
- strength1 part of net:

  `supply1 strong1 pull1 weak1 highz1`
- In contrast to 9-val. logic IEEE std 1164 as used in VHDL

Busses and Multi-bit Registers

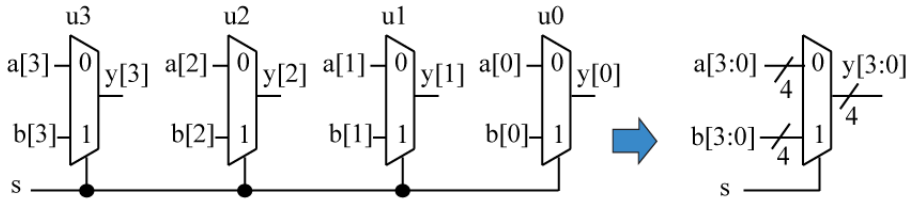
- Can use “array-type” Notation
- Examples:

```
wire [2:0] Bname            // A 3-bit bus called Bname

reg [7:0] Accumulator       // An 8-bit register named Accumulator
```

- Suggestions
  - Always number from MSb to LSb
  - Matches the Radix Power in Radix Polynomial
  - Consistency Helps to Avoid Bugs

Busses and Instantiation



(a) Four-bit 2-to-1 multiplexer using four of the previously defined one-bit 2-to-1 modules named `mux2to1`

```
module mux2to1_4bit(s,a,b,y);
input  s;
input  [3:0] a,b;
output [3:0] y;

mux2to1 u3 (.a(a[3]), .b(b[3]), .s(s),
.y(y[3]));
mux2to1 u2 (.a(a[2]), .b(b[2]), .s(s),
.y(y[2]));
mux2to1 u1 (.a(a[1]), .b(b[1]), .s(s),
.y(y[1]));
mux2to1 u0 (.a(a[0]), .b(b[0]), .s(s),
.y(y[0]));

endmodule
```

Port *a* of instance *u3* connects to port *a[3]*

User-defined instance name, must be unique within this module.

Component’s module name

(b) Four-bit 2-to-1 multiplexer using a single `assign` statement

```
module mux2to1_4bit(s,a,b,y);
input  s;
input  [3:0] a,b;
output [3:0] y;

assign y=(b & s)|(a & ~s);

endmodule
```

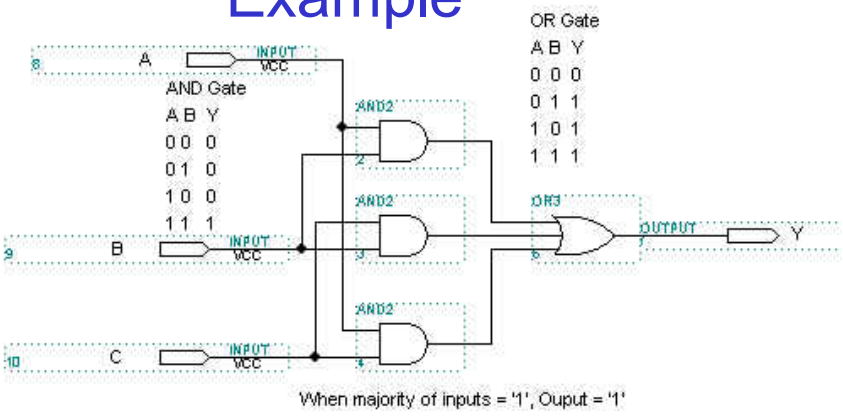
Bus definition for a 4-bit bus; most significant bit is *y[3]* and least significant bit is *y[0]*.

Verilog Logic Gate Primitives

Name	Description	Usage
and	$f = (a \cdot b \cdots)$	**and** (f,a,b,…)
nand	$f = \overline{(a \cdot b \cdots)}$	**nand** (f,a,b,…)
or	$f = (a + b + \cdots)$	**or** (f,a,b,…)
nor	$f = \overline{(a + b + \cdots)}$	**nor** (f,a,b,…)
xor	$f = (a \oplus b \oplus \cdots)$	**xor** (f,a,b,…)
xnor	$f = (a \odot b \odot \cdots)$	**xnor** (f,a,b,…)
not	$f = a$	**not** (f,a)
buf	$f = a$	**buf** (f,a)
notif0	$f = (!e?\overline{a} : 'b\overline{z})$	**notif0** (f,a,e)
notif1	$f = (e?a : 'bz)$	**notif1** (f,a,e)
buf0	$f = (!e?a : 'bz)$	**buf0** (f,a,e)
buf1	$f = (e?a : 'bz)$	**buf1** (f,a,e)

Table A.2. Verilog gates.

Example




```
// Combinational Logic Circuit
module maj_circ(Y, A, B, C);
   input A, B, C;
   output Y;
   and U1 (x1,A,B);
   and U2 (x2,A,C);
   and U3 (x3,B,C);
   or  U4 (Y,x1,x2,x3);
endmodule
```

User Defined Primitives (UDPs)

- Keywords **`and`**, **`or`**, **`not`**, **`xor`**, etc. are System Primitives
- Can Define your Own Primitives (UDPs)
- Can do this in a variety of ways including Truth Tables
- Instead of **`module`**/**`endmodule`** use the keywords **`primitive`**/**`endprimitive`**
- Only one output and must be listed first
- Keywords **`table`** and **`endtable`** used
- Input values listed in order with a **`:`**
- Output is always last entry followed by **`;`**

UDP Verilog Example

```
// User defined primitive (UDP)
primitive crctp (x, A, B, C);
  output x;
  input  A, B, C;
// Truth table for x(A,B,C)=Σ(0,2,4,6,7)
  table
//    A  B  C  :  x  (note: this is a comment)
      0  0  0  :  1;
      0  0  1  :  0;
      0  1  0  :  1;
      0  1  1  :  0;
      1  0  0  :  1;
      1  0  1  :  0;
      1  1  0  :  1;
      1  1  1  :  1;
  endtable
endprimitive

// Instantiate primitive
module declare_crctp;
        reg   x,y,z;
        wire  w;
        crctp (w, x, y, z);
endmodule
```

Boolean Expressions in Verilog

- Use the Continuous Assignment Statement
  - Keyword is **`assign`**
  - Boolean Operators (normal precedence):
    - **`&`** - AND
    - **`|`** - OR
    - **`~`** - NOT (invert)
  - When in Doubt about Precedence Use Parentheses
- Previous Example as Expression:

```
assign x = (A & B) | (~C);
```


Verilog Assignment Statements

(a) Single `assign` statement using boolean operators

```
module mux2to1(s,a,b,y);
input  s,a,b;
output y;

//this is a comment
assign y=(b & s)|(a & ~s);


endmodule
```

(b) Multiple `assign` statements using Boolean operations and intermediate values

```
module mux2to1(s,a,b,y);
input  s,a,b;
output y;

wire na, nb;

//this is a comment
assign nb = b & s;
assign na = a & ~s;
assign y = na | nb;

endmodule
```

(c) Single `assign` statement using a conditional operator

```
module mux2to1(s,a,b,y);
input  s,a,b;
output y;

//conditional statement
//            s=1 s=0
assign y = s ? b : a ;


endmodule
```

Assignment Statement Ordering

(a) assignment statement
ordering does matter in an HLL

```
a = 1; b = 0; s = 0;
na = 0; nb = 0;
```

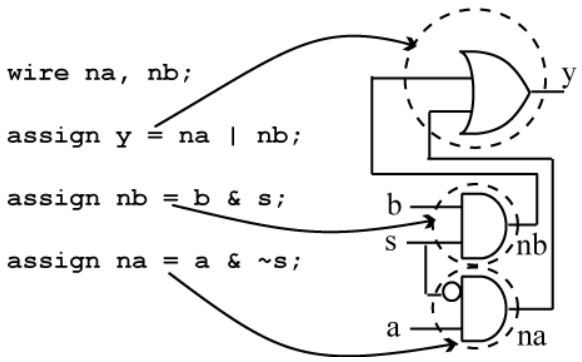
```
y = na | nb;          nb = b & s;
nb = b & s;           na = a & ~s;
na = a & ~s;          y = na | nb;
```

Final y value is 0. Final y value is 1.

Sequential Language
- Program Counter or Instruction Pointer
- Static Execution

(b) `assign` statement
ordering does not matter in Verilog

```
wire na, nb;
assign y = na | nb;
assign nb = b & s;
assign na = a & ~s;
```

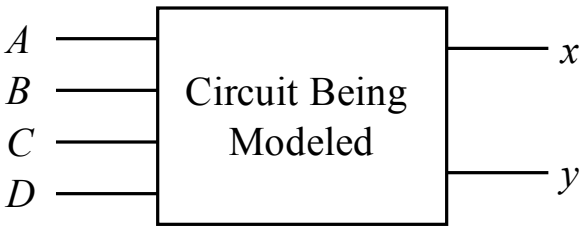


Event Driven Language
- Event Queue
- Dynamic Execution

Verilog Example

$$x = A + BC + \overline{B}D$$

$$y = \overline{B}C + B\overline{C}\,\overline{D}$$




```
// Circuit specified with Boolean expressions
module circuit_bln (x, y, A, B, C, D);
      input A,B,C,D;
      output x,y;
      assign x = A | (B & C) | (~B & D);
      assign y = (~B & C) | (B & ~C & ~D);
endmodule
```

Assignment Statement Problem

(a) assignment statements in an HLL can target the same variable

```
a = 1; b = 0; s = 0;
na = 0; nb = 0;

na = b & s;

na = a & ~s;
```

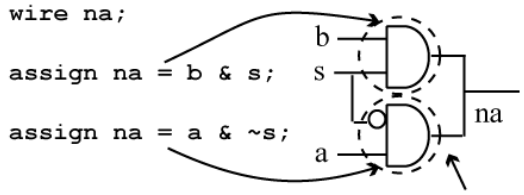
The `na` variable is assigned twice; the final value of `na` is the last assignment.

(b) illegal use of `assign` statements

```
wire na;

assign na = b & s;

assign na = a & ~s;
```



Gate outputs are shorted together!