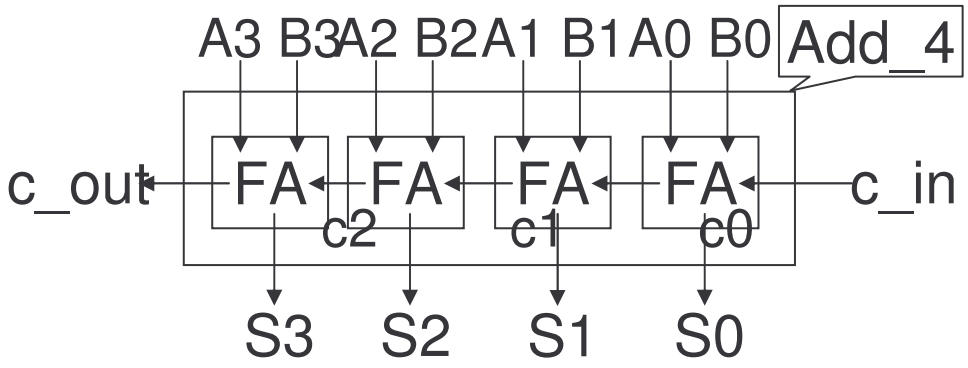
```
16'h7FEx       // 16-bit value, low order 4 bits unknown
8'bxx001100 // 8-bit value, most significant 2 bits
                   unknown.
8'hzz          // 8-bit value, all bits high impedance.
```

# Verilog Design Principles

ECGR2181

Extra Notes




*Reading:* Chapter 4

# Introduction to Hardware Design Languages

**What are Hardware Description Languages (HDL)?**

"Programming" languages that allow the characteristics of a hardware module to be described.

Many elements analogous to "Software programming" languages:

Declarations of data elements and structures

Description of the sequence of operations which must take place.

Two preeminent HDL's are **Verilog** and **VHDL**.

**Why use Hardware Design Languages?**
("My logic template works fine!")

Easier to understand design -- especially functionality.

Concise and much easier to incorporate design changes.

Text based -- easy to incorporate changes into design.

Use as input to computer aided design tools.

- Design verification
- Synthesis of logic
- Generation of "object codes" for implementing the design using ASICS or FPGA's

## Origins of Verilog HDL

Verilog is only one of many HDL's.

Developed in 1984 by Gateway Design Automation.

Verilog language structure is based on "C".

(VHDL is based on ADA.)

Has become an industry standard

2-to-1 Multiplexer

So the complete module description is:

```
// 2-to-1 Multiplexer module
module mux_2 (out, i0, i1, sel);      // header
input i0, i1, sel;                    // input & output ports
output out;
wire x1, x2, x3;                      // internal nets
or (out, x2, x3);                     // form output
and (x2, i0, x1);                     // i0 • sel'
and (x3, i1, sel);                    // i1 • sel
not (x1, sel);                        // invert sel
endmodule
```

# VHDL Example - Inverter

```
library IEEE;
use IEEE.std_logic_1164.all;

entity INV is
  port (X: in STD_LOGIC; Y: out STD_LOGIC);
end INV;

architecture INV_A of INV is
begin
  Y <= not X;
end INV_A;
```

# Verilog Language Structure

**Language conventions:**

Tokens: Verilog code contains a stream of “tokens”. Tokens can be comments, delimiters, numbers, strings, identifiers and keywords.

Case sensitivity: Verilog is case sensitive. Keywords are in lowercase.

Whitespace: Blank spaces, tabs and newlines are ignored. Exceptions are when it separates tokens or when it appears in strings.

## Language conventions:

Comments: Comments follow normal "C" conventions.

A "//" starts a one-line comment. Everything that appears after the "//" to the end of line (newline) is treated as a comment.

```
a = b && c;  // The rest of this line is a comment
```

A multiple-line comment begins with "/*" and ends with "*/". Multi-line comments cannot be nested.

```
/* This is a multiple line
        comment*/
```

**Language conventions:**

Operators: There are three types of operators:

Unary - single operand. Operator precedes operand.

Binary - two operands. Operator appears between the two operands.

Ternary - three operands. Two operators separate the three operands.

Statement terminator: Most Verilog statements end with a semicolon.

**Language conventions:** Numbers & Values

Sized numbers: *<size>'<base format><value>*

*<size>*: specifies number of bits in number (in decimal)

*<base format>*: decimal ('d, 'D); hexadecimal ('h, 'H); binary ('b, 'B); octal ('o, 'O)

*<value>*: digits (in base format) of the numeric value

```
6'b100111   // 6-bit binary number
16'h7FFE    // 16-bit hexadecimal number
8'd133      // 8-bit decimal number
```

**Language conventions:** Numbers & Values

Unsized numbers: *'<base format><value>*

Number of bits is simulator & machine dependent (≥ 32).

```
'b100111      // 32-bit number (00…100111)
'h7FFE        // 32-bit number (00007FFE)
'd2468        // 32-bit decimal number
```

Note: If *<base format>* is omitted, decimal is assumed.

So: 123456 is the same as 'd123456

**Language conventions:** Numbers & Values

In addition to normal numeric values, Verilog provides two special values, **x** and **z**.

**x** denotes an unknown or undefined value.

**z** denotes a “high impedance” value

```
16’h7FEx       // 16-bit value, low order 4 bits unknown
8’bxx001100 // 8-bit value, most significant 2 bits
                  unknown.
8’hzz          // 8-bit value, all bits high impedance.
```

**Language conventions:** Numbers & Values

Zero fill / extension: If a numeric value does not contain enough digits to fill the specified number of bits, the high order bits are filled with zeros. If the most significant bit specified is an **x** or **z**, the **x**/**z** is left extended to fill the bit field.

16'h39 ⇒ 16'h0039 ⇒ 0000 0000 0011 1001

8'hz ⇒ 8'hzz ⇒ zzzz zzzz

Negative numbers: Specified by preceding the *<size>* with a minus (negative) symbol. Values will be stored as the two's complement of the given value.

**Language conventions:** Data types

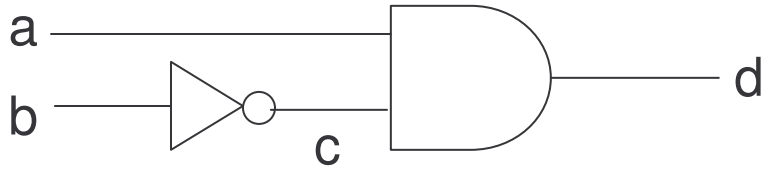
| | | |
|---|---|---|
| Logic value: | 0 | Logic zero, false condition |
| | 1 | Logic one, true condition |
| | x | Unknown / undefined value |
| | z | High impedance, floating state |

Note 1: In a physical implementation, the value x cannot exist. A logic signal will be either 0, 1, or z.

Note 2: There is no prescribed relationship between a logic state and the physical manifestation of that state.

**Language conventions:** Nets, Registers & Vectors

Nets represent connections between hardware components.




a, b, c, & d are nets.

Nets are declared by keyword **wire**.

```
wire d;          // declare output as net d.
wire b, c;       // declare two wires in same statement
```

**Language conventions:** Nets, Registers & Vectors

Nets: The logical value of a net is set by the output of its driver. If there is no driver, the net value is **z**.

"Net" is not a keyword -- it is a class of data types.

Some keywords for types of nets are: **wire**, **wand**, **wor**, **tri**, **triand**, **trior**, **trireg**, etc.

A net can be given an fixed logic value as it is declared.

Example: wire a = 1'b1;

**Language conventions:** Nets, Registers & Vectors

Registers provide data storage. In Verilog, *register* simply means a variable that can hold a value.

A Verilog register is not the same as a hardware register.

A register does not need a driver. They also do not need a clock like a hardware register does. Values are changed by assigning a new value to the register.

Confused? In Verilog, we do not explicitly declare hardware registers. The context of the description determines if a physical register exists.

**Language conventions:** Nets, Registers & Vectors

Registers are declared by the keyword **reg**. The default value for a **reg** data type is **x**.

```
reg start;  // declares register "start"

reg reset, clock;  // declares registers reset & clock
```

Vector & scalar data types: Nets and **reg** data types can be declared as vectors (multiple-bit data) or as scalars (single-bit data). If the width of a data variable is not declared, scalar is assumed.

**Language conventions:** Nets, Registers & Vectors

Vector and scalar declarations

Vectors can be specified by declaring the range of bit numbers with the variable name. The form of the declaration is: [<high#>: <low#>] <variable> ;
or [<low#>: <high#>] <variable> ;

```
wire [7:0] BYTE;      // declare 8-bit data.
reg [15:0] INFO;      // declare 16-bit register.
reg [0:11] DATA;      // declare 12-bit register
```

**Language conventions:** Nets, Registers & Vectors

Note: The bit numbers can run in either direction but the left-hand number of the pair in the brackets is always the most significant bit of the vector.

So, in the following declarations:

```
reg [15:0] INFO;      // declare 16-bit register.
reg [0:31] DATA;      // declare 32-bit register
```

bit 15 of INFO and bit 0 of DATA are the MSB's.

Note: For lecture, notes, etc. I will be assuming that bit 0 will be the Least Significant Bit. .

**Language conventions:** Nets, Registers & Vectors

Note: While bit numbering in a declaration conventionally includes bit 0, it is not required. *wire [1:10] STUFF;* is also a valid declaration.

So: For those who are “zero challenged” you could number the bits of a byte as 8:1 instead of 7:0. Just bear in mind, all class notes will use the “zero-relative” numbering.

**Language conventions:** Nets, Registers & Vectors

As seen from previous examples of scalar declaration, multiple vectors can be declared on a single line. So the declaration of INFO and DATA from above could be written as:

```
reg [15:0] INFO, [31:0] DATA;
```

Likewise, the declaration of 8-bit registers A, B & C could be written as:

```
reg [7:0]A, [7:0]B, [7:0]C;
```

**Language conventions:** Nets, Registers & Vectors

When a list of variables is given in a declaration, Verilog assumes that each variable is sized according to the last size specification seen as that line is scanned. This allows a set of same-sized vectors to be declared without explicitly specifying the size of each one.

Thus the declaration of 8-bit registers A, B & C can be written as:

```
reg [7:0]A, B, C;
```

Note: While a mixed specification line such as:

```
reg [7:0] A, B, C, [15:0] DATA;
```

is allowed, it is discouraged since it could be confusing.

**Language conventions:** Nets, Registers & Vectors

Specifying parts of vectors:

Given vector declarations, it is possible to reference parts of a register (down to a single bit). The format of the reference follows the pattern <vector> [ <bit range>].

```
INFO [5]       // bit 5 of INFO
INFO [11:8]    // second nibble of INFO (bits 11-8)
DATA [7:0]     // most significant byte of DATA
```

## Language conventions:

Integer data type: A general purpose register data type for manipulating integer values. They are declared by the keyword **integer**. Values are 32 bit signed values.

These are rarely used in the description of a hardware module but are frequently useful in the Test Fixture.

Time data type: A special register which tracks simulation time. Declared with keyword **time**. A system function **$time** accesses the current simulation time.

**Module interconnections:** Ports

A digital module must have the ability to interconnect with its environment. This requires the definition of input and output **ports.**

It is through these ports that all information is passed between modules. Thus an integral part of the definition of a module is the declaration of its ports.

**Module interconnections:** Ports

There are three types of ports which can be declared in Verilog.

| | |
|---|---|
| **input** | Input only port |
| **output** | Output only port |
| **inout** | Bidirectional port |

Ports are declared with the syntax:

<port_type> { [<size>] } <name>

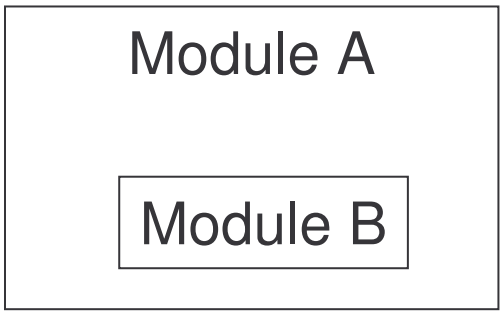
```
input clock, reset;          // single bit input signals
input [15:0] DATA;           // 16-bit input word
output data_strobe;          // single-bit output signal
output [7:0] RESULT;         // output byte
```

**Module interconnections:** Ports

Within a Verilog system model, module interconnections occur at two levels:

- Peer to peer: modules interconnect with each other:


- Hierarchical: one module incorporates the other.



**Module interconnections:** Ports

Port connections must observe a set of rules about the type of signal (both internal & external) they connect to.

Inputs: Internally ports must be a *net* (typ. Wire). The external source may be either a **reg** or a *net*.

Outputs: The internal source for an output port can be either a reg or a net. The external destination must be a net.

Inouts: The inout ports must be connected to nets both internally and externally.

Gate-level modeling is the lowest level of design description in Verilog.

(Actually there is a lower level -- transistor level.)

Verilog models at the gate level consists of directly specifying the interconnections of fundamental logic elements (AND, OR, etc.).

The available logic elements at the gate level are:
**and**, **nand**, **or**, **nor**, **xor**, **xnor**, **not**, **buf**, **notif** & **bufif**.
(All these are keywords.)

The functionality of these basic logic gates are self-explanatory with the exception of buf, notif & bufif.

**buf** is simply a non-inverting buffer gate. It is transparent from a logical sense but may be required for implementation.

**notif** & **bufif** are tri-state versions of the not & buf gates. These gates have a extra control line which enables the gate when true and places the gate into the high-impedance **z** state when false.

# Gate-Level Modeling

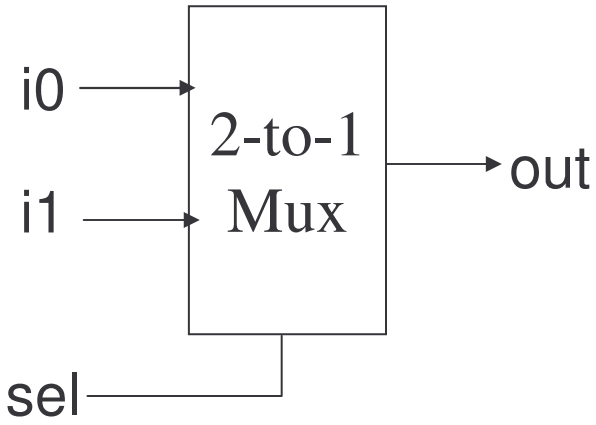
Description of a module at the gate level consists of the declarations (header, ports, variables) and a series of instantiations of the base logic elements. Through the instantiations, the wiring of the module is specified.

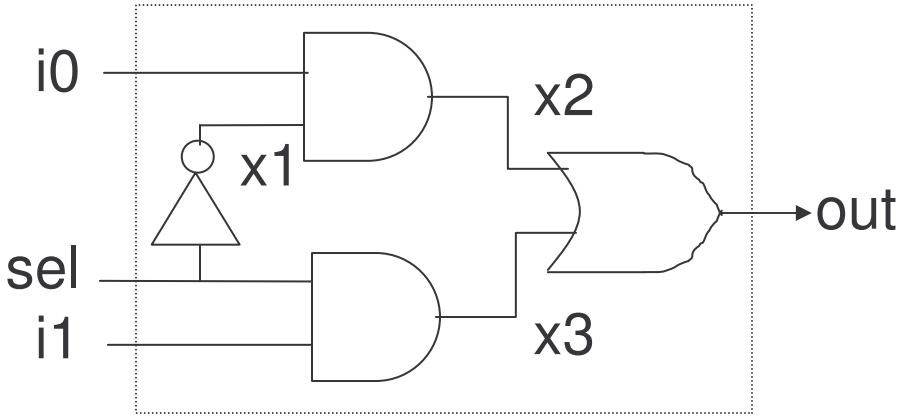
The format of a gate-level instantiation is:

<gate_type> <i_name> (<out_name>, <in_name_list);

```
and q1 (q1_out, q1_in1, q1_in2);                 // 2-input AND
or   q2 (q2_out, q2_in1, q2_in2, q2_in3);       // 3-input OR
not  q3 (q3_out, q3_in);                         // inverter
notif q4 (q4_out, q4_in, control);               // tri-state inverter
```

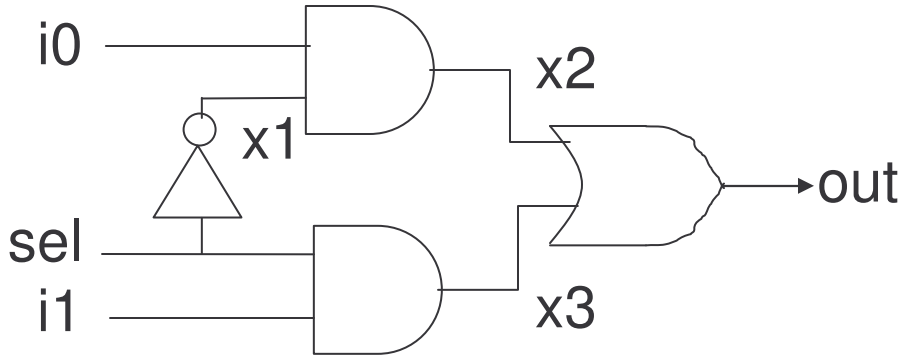
2-to-1 Multiplexer




```
// 2-to-1 Multiplexer module
module mux_2 (out, i0, i1, sel);        // header
input i0, i1, sel;                      // input & output ports
output out;
wire x1, x2, x3;                        // internal nets
```

2-to-1 Multiplexer




```
or (out, x2, x3);                    // form output
        and (x2, i0, x1);                    // i0 • sel'
and (x3, i1, sel);                   // i1 • sel
not (x1, sel);                       // invert sel

endmodule
```

2-to-1 Multiplexer

So the complete module description is:

```
// 2-to-1 Multiplexer module
module mux_2 (out, i0, i1, sel);      // header
input i0, i1, sel;                    // input & output ports
output out;
wire x1, x2, x3;                      // internal nets
or (out, x2, x3);                     // form output
and (x2, i0, x1);                     // i0 • sel'
and (x3, i1, sel);                    // i1 • sel
not (x1, sel);                        // invert sel
endmodule
```

```
library IEEE;
use IEEE.std_logic_1164.all;

entity parity9 is
    port (
       I: in STD_LOGIC_VECTOR
     (1 to 9);
       EVEN, ODD: out
     STD_LOGIC
    );
end parity9;
```

```
architecture parity9p of parity9 is
begin
process (I)
  variable p : STD_LOGIC;
  variable j : INTEGER;
  begin
    p := I(1);
    for j in 2 to 9 loop
      if I(j) = '1' then p := not p; end if;
    end loop;
    ODD <= p;
    EVEN <= not p;
  end process;
end parity9p;
```