3) numbers

syntax: <size>'<base><value>

(in bits) b, B

h, H

d, D

e.g.

4'b1111

16'h1111

for binary strings, there are 4 ~~possibles~~ possible choices

'0', '1', 'x', 'z'

e.g. 4'b11x0

e.g. the following is ok: 20'b00xz_1111_1...._0

4) "hello world" is supported

5) identifiers (i.e. variables) naming

rules: ① can only start with

_ or a letter
(underbar)

② '$' is reserved
(don't use it)

NOTE: with identifiers, Verilog
is case sensitive

6) Keywords

variables names that are reserved

e.g. module
begin
end
parameter
include
⋮

Note: Keywords are <u>always</u> lower case

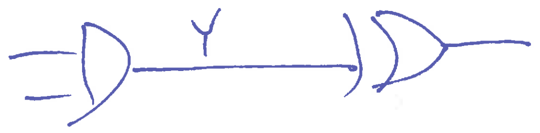
Data types

values: '0', '1', 'x', 'z'

Defn (net type variable)

an interconnection between hardwire elements

e.g.



Keywords used to declare net variables

- wire ✓
- wand
- woR
- input ✓
- output ✓
- ⋮

syntax

```
wire a, y; // 'a' & 'y' declared
              as net type variables
```

Defn: (register Variable)

a variable that can hold a value

Keyword : reg

notes: 1) can be multiple bit values
(and BTW so can net variables)

2) <u>not</u> a D-FF !!!

it means its a variable that can updated with assignment statements

e.g.

```
c = 22;
  :
c = 49;
```

e.g.

```
reg reset;
initial
     begin
          reset = 1'b1;
     #100 reset = 1'b0;
end
```

Want an 8-bit bus

wire [7:0] my-bus ;

most significant bit notation

Least significant bit notation

You get

my-bus_0

my-bus_1

⋮

my-bus_7

Always Remember

the purpose of writing a program is <u>not</u> to get something you can simulate

But something you can synthesize

```
reg A;
reg [7:0] B;
reg [2:0] c;
```

e.g.

```
              b7   b4 b3 b2  b0
              ↓    ↓  ↓  ↓   ↓
B = 8'b110 11100;

c = B[4:2];   // c = 3'b111
```

Some textbooks

```
reg [0:7] A;
```

Declaring memory

$$\text{reg } \underbrace{[7:0]}_{\text{bits/loc.}} \text{ m } \underbrace{[4095:0]}_{\text{\# of locs}} \text{ ; // 4K} \times \text{8 memory}$$

Keyword : parameter

allows you to define constants at compile time

Note: code cannot change them

e.g.

```
parameter byte=8;
parameter     K4 = 4095;

reg [byte-1:0] sally [K4:0];
```

Note: supports module re-use

<u>Compiler directive</u>

there are instructions to the compiler
(no executable code)

<u>in C</u>

```
#include <file.h>
```

<u>in Verilog</u>

```
`include <file.h>
```

$\nearrow$

backwards tick mark

in C

```
#define A 6 ;
```

in Verlog

```
`define A 6;
```

~~we saw par~~

```
wire ['A:0] W;
```

we say

```
parameter A =6;
   ⋮
wire [A:0] W;
```

Defn (instantiation)

the process of creating an object from a module template

Defn (instance)

the objects created by instantiation

in C

```
            float foo (int m)
template →  {
                :
            }

            main (void)
            {
                :
where foo   → y = foo(2);
is executed     :
            }
```

in Verilog

```
module foo (. . . .)    } module template
endmodule

module main (. . . )    } instantiation
  foo (. . .) ;   ← instance of the module template
endmodule
```

Example 'L'

```
// Define the top-level module called ripple carry
// counter. It instantiates 4 T-flipflops. Interconnections are
// shown in Section 2.2, 4-bit Ripple Carry Counter.
module ripple_carry_counter(q, clk, reset);

output [3:0] q; //I/O signals and vector declarations
              //will be explained later.
input clk, reset; //I/O signals will be explained later.

//Four instances of the module T_FF are created. Each has a unique
//name.Each instance is passed a set of signals. Notice, that
//each instance is a copy of the module T_FF.
T_FF tff0(q[0],clk, reset);
T_FF tff1(q[1],q[0], reset);
T_FF tff2(q[2],q[1], reset);
T_FF tff3(q[3],q[2], reset);

endmodule

// Define the module T_FF. It instantiates a D-flipflop. We assumed
// that module D-flipflop is defined elsewhere in the design. Refer
// to Figure 2-4 for interconnections.
module T_FF(q, clk, reset);

//Declarations to be explained later
output q;
input clk, reset;
wire d;

D_FF dff0(q, d, clk, reset); // Instantiate D_FF. Call it dff0.
not n1(d, q); // not gate is a Verilog primitive. Explained later.

endmodule
```

instan.

4 instances

template

To illustrate these hierarchical modeling concepts, let us consider the design of a negative edge-triggered 4-bit ripple carry counter described in Section 2.2, `4-bit Ripple Carry Counter`.

2.2 4-bit Ripple Carry Counter

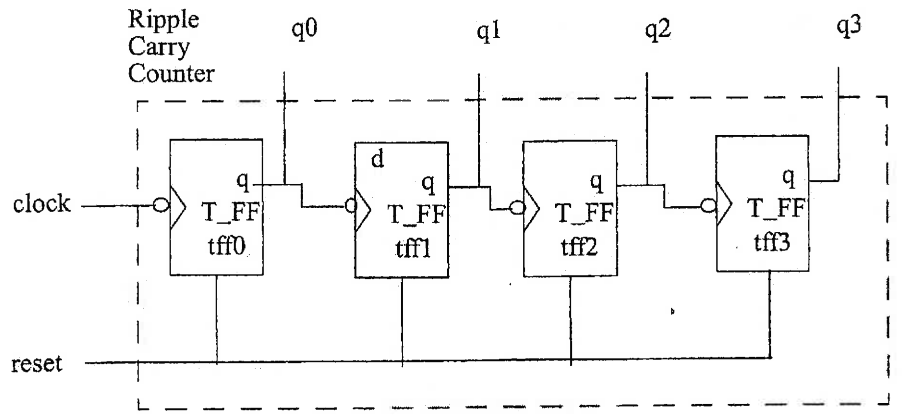


Figure 2-3 Ripple Carry Counter

The ripple carry counter shown in Figure 2-3 is made up of negative edge-triggered toggle flipflops (`T_FF`). Each of the *T_FFs* can be made up from negative edge-triggered D-flipflops (*D_FF*) and `inverters` (assuming *q_bar* output is not available on the *D_FF*), as shown in Figure 2-4.

reset	q_n	q_{n+1}
1	1	0
1	0	0
0	0	1
0	1	0
0	0	0

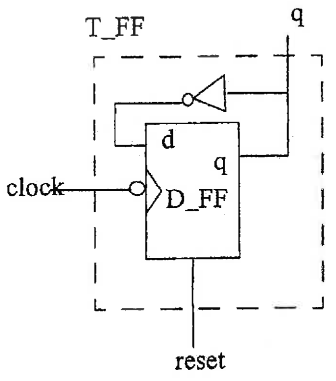


Figure 2-4 T-flipflop

Defn (port)

are the interface between a module and its environment

4.2.2 Port Declaration

All ports in the list of ports must be declared in the module. Ports can be declared as follows:

Verilog Keyword	Type of Port
`input`	Input port
`output`	Output port
`inout`	Bidirectional port

} all net variable declarations

Each port in the port list is defined as **`input`**, **`output`**, or **`inout`**, based on the direction of the port signal. Thus, for the example of the *fulladd4* in Example 4-2, the port declarations will be as shown in Example 4-3.

Example 4-3 Port Declarations

```
module fulladd4(sum, c_out, a, b, c_in);

//Begin port declarations section
output[3:0] sum;
output c_cout;

input [3:0] a, b;
input c_in;
//End port declarations section
...
<module internals>
...
endmodule
```

Note that all port declarations are implicitly declared as **`wire`** in Verilog. Thus, if a port is intended to be a **`wire`**, it is sufficient to declare it as **`output`**, **`input`**, or **`inout`**. **`Input`** or **`inout`** ports are normally declared as **`wires`**. However, if **`output`** ports hold their value, they must be declared as **`reg`**. For example, in the definition of *DFF*, in Example 2-5, we wanted the output *q* to retain its value until the next clock edge. The port declarations for *DFF* will look as shown in Example 4-4.

[redacted]

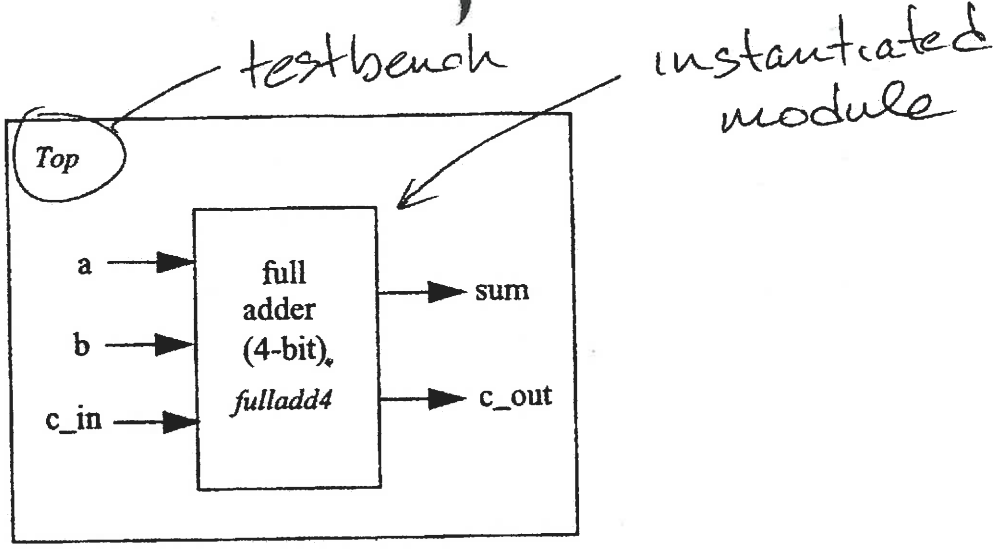


Figure 4-3 *I/O Ports for Top and Full Adder*

Example 4-2 *List of Ports*

```
module fulladd4(sum, c_out, a, b, c_in); //Module with a list of ports
module Top; // No list of ports, top-level module in simulation
```