modules are the basic
building blocks of
verilog programs

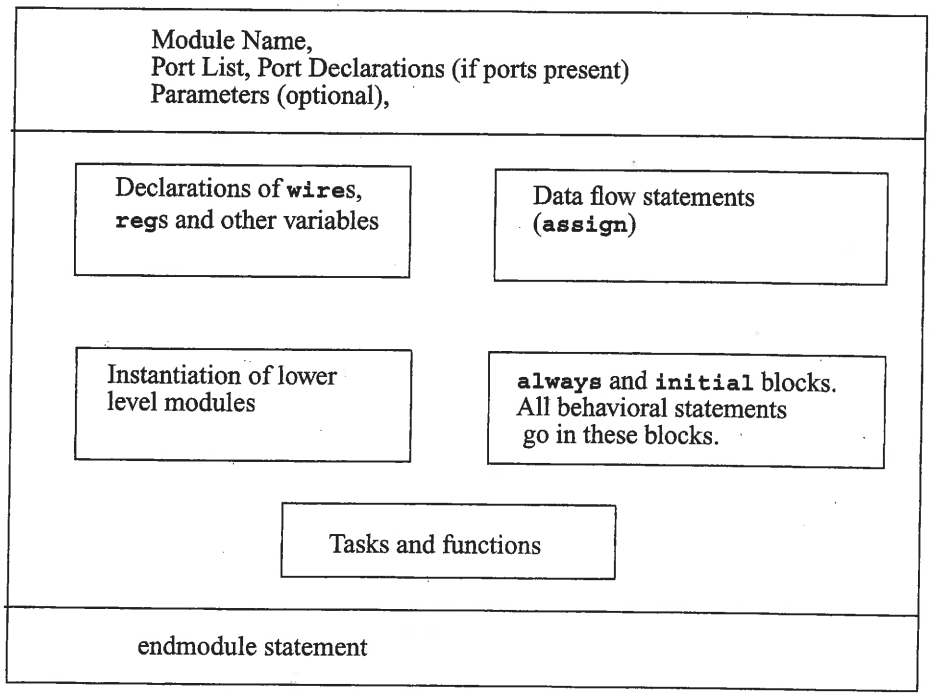


Figure 4-1 Components of a Verilog Module

Defn (instantiation)

this is the process of creating an object from a module template

Defn (an instance)

the objects created by instantiation

in C

```
float foo (float m)
{
    =
}

main (void)
{
    =
    y = foo(x);
    =
}
```

r fn mplate)

ere 'foo' s executed

in Verilog

```
module foo (....)
endmodule

module main (...)
    foo( );      // instantiatie
endmodule
```

module templa

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

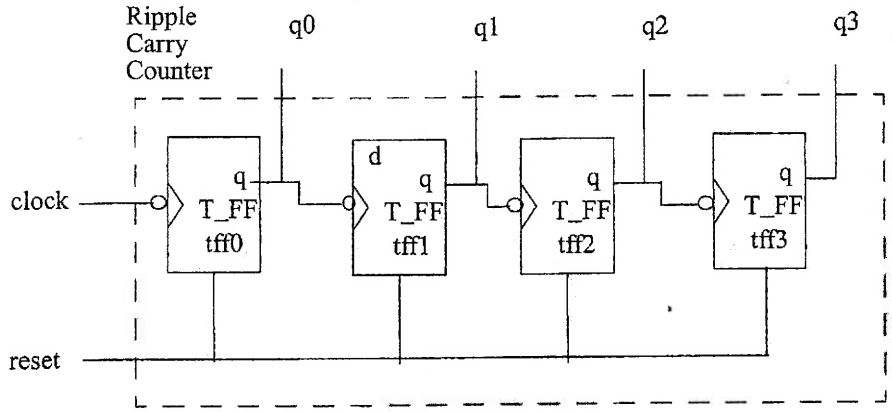


Figure 2-3 Ripple Carry Counter

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

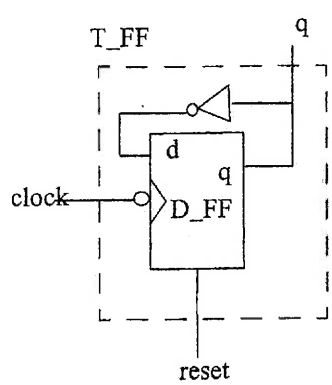


Figure 2-4 T-flipflop

example "2"


```
// 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

Defn (ports)

these are the interface between the module and an external 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.

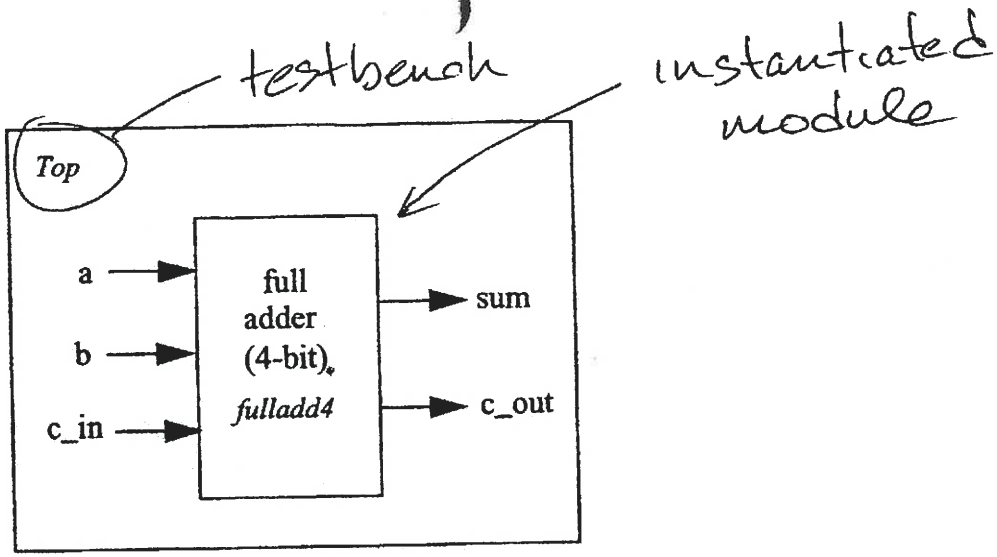


Figure 4-3 I/O Ports for Top and Full Adder

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
```

'output' declares a net type variable

e.g.

```
module Foo (a, c);
  input a;      // net
  output c;     // net
    :
endmodule
```

if you want c to be a reg variable
you must re-delare it

Example 4-4 *Port Declarations for DFF*

```
module DFF(q, d, clk, reset);
output q;
reg q; // Output port q holds value; therefore it is declared as reg.
input d, clk, reset;
...
...
endmodule
```

Ports of the type **input** and **inout** cannot be declared as **reg** because **reg** variables store values and input ports should not store values but simply reflect the changes in the external signals they are connected to.

Note that the module *fulladd4* in Example 4-3 can be declared using an ANSI C style syntax to specify the ports of that module. Each declared port provides the complete information about the port. Example 4-5 shows this alternate syntax. This syntax avoids the duplication of naming the ports in both the module definition statement and the module port list definitions. If a port is declared but no data type is specified, then, under specific circumstances, the signal will default to a *wire* data type.

Example 4-5 *ANSI C Style Port Declaration Syntax*

```
module fulladd4(output reg [3:0] sum,
                output reg c_out,
                input [3:0] a, b, //wire by default
                input c_in); //wire by default
...
<module internals>
...
endmodule
```

output [3:0] su
reg [3:0] sum

Port Connection Rules

One can visualize a port as consisting of two units, one unit that is *internal* to the module and another that is *external* to the module. The internal and external units are connected. There are rules governing port connections when modules are instantiated within other modules. The Verilog simulator complains if any port connection rules are violated. These rules are summarized in Figure 4-4.

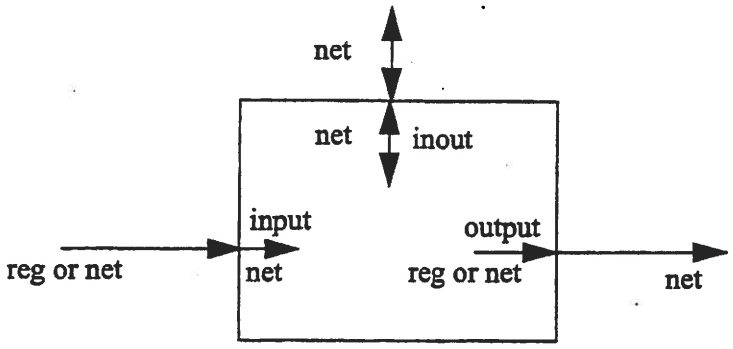


Figure 4-4 Port Connection Rules

Inputs

Internally, input ports must always be of the type *net*. Externally, the inputs can be connected to a variable which is a `reg` or a *net*.

Outputs

Internally, outputs ports can be of the type `reg` or *net*. Externally, outputs must always be connected to a *net*. They cannot be connected to a `reg`.

Inouts

Internally, inout ports must always be of the type *net*. Externally, inout ports must always be connected to a *net*.

Width matching

It is legal to connect internal and external items of different sizes when making inter-module port connections. However, a warning is typically issued that the widths do not match.

when redeclaring, you must match sizes

e.g. (wrong)

```
module temp (A,B);
output [4:0]A;
reg A;
.
.
.
```

e.g. (right)

```
module temp (A,B);
output [4:0]A;
reg [4:0]A;
```

Example 4-6 *Illegal Port Connection*

```
module Top;

//Declare connection variables
reg [3:0]A,B;
reg C_IN;
reg [3:0] SUM;
wire C_OUT;

    //Instantiate fulladd4, call it fa0
    fulladd4 fa0(SUM, C_OUT, A, B, C_IN);
    //Illegal connection because output port sum in module fulladd4
    //is connected to a register variable SUM in module Top.
    .
    .
    <stimulus>
    .
    .
endmodule
```

This problem is rectified if the variable *SUM* is declared as a *net* (**wire**).

how do I connect two modules?

Ans: instantiation and then thru the port list

you connect signals 2 ways

- by position
- by name

Example 4-7 *Connection by Ordered List* (or conn by position)

```
module Top;

//Declare connection variables
reg [3:0]A,B;
reg C_IN;
wire [3:0] SUM;
wire C_OUT;

    //Instantiate fulladd4, call it fa_ordered.
    //Signals are connected to ports in order (by position)
    fulladd4 fa_ordered(SUM, C_OUT, A, B, C_IN);
    ...
    <stimulus>
    ...
endmodule


module fulladd4(sum, c_out, a, b, c_in);
output [3:0] sum;
output ~~c_cout;~~ c_out;
input [3:0] a, b;
input c_in;
    ...
    <module internals>
    ...
endmodule
```

module template

Connection by name (preferred)

```
module fulladd4(sum, c_out, a, b, c_in);
output[3:0] sum;
output c_cout;
input [3:0] a, b;
input c_in;
    ...
    <module internals>
    ...
endmodule
```

mistake

```
// Instantiate module fa_byname and connect signals to ports by name
fulladd4 fa_byname(.c_out(C_OUT), .sum(SUM), .b(B), .c_in(C_IN), .a(A),);
```

order doesn't matter!!

Defn (abstraction)

a description that hides unimportant information to make the description less complex

e.g. Suppose I want to describe a digital design. How could I do it?

method #1	method #2	method #3
schematic	boolean eqns	truth table

more abstract →

Defn (abstraction)

a description that hides information to present a more concise form

e.g. Suppose I want you to build a digital circuit for me. How do I describe what is needed?

method #1	method #2	method #3
Schematic	Boolean equations	truth table

abstraction $\longrightarrow$

$\longleftarrow$ details

in Verilog, code can be written in 3 Levels of abstraction

abstraction

(Lowest) — gate-Level description

— dataflow *

(highest) — behavioral *

Notes:

1) modules can contain multiple Levels of abstraction

2) modules that use only * are ~~ta~~ said to be RTL coded modules

Verilog has 3 levels of abstraction

abstraction

(lowest) — gate level

— dataflow *

(highest) — behavioral *

Notes:

1) a module may contain multiple levels of abstraction

2) any module written solely in * is called

RTL coding

gate Level modeling

Basic building block of Verilog programs is a

module

Defn (primitive gate)

a ~~predefing~~

predefined Logic gate in Verilog

```
and      or      xor
nand     nor     xnor
```

The corresponding logic symbols for these gates are shown in Figure 5-1. We consider gates with two inputs. The output terminal is denoted by *out*. Input terminals are denoted by *i1* and *i2*.

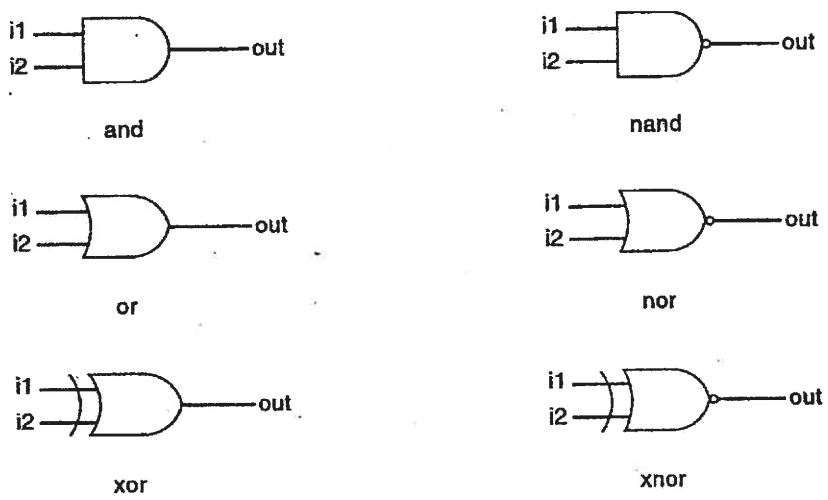


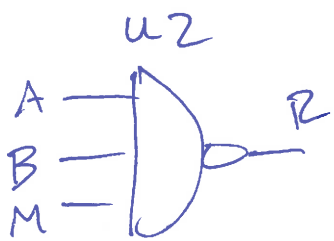
Figure 5-1 Basic Gates

format:

gate-type gate-identifier ($\underbrace{\text{out-name}}_{\substack{\text{output} \\ \text{name} \\ (1\ \text{only})}}$, $\underbrace{\text{in0, in1, in2, ..., inK}}_{\substack{\text{inputs} \\ (\text{as many as} \\ \text{you want})}}$);

and
nor
.
.
.

e.g.



nand u2 (R, A, M, B);

Example 5-1 *Gate Instantiation of And/Or Gates*

```
wire OUT, IN1, IN2;

// basic gate instantiations.
and a1(OUT, IN1, IN2);
nand na1(OUT, IN1, IN2);
or or1(OUT, IN1, IN2);
nor nor1(OUT, IN1, IN2);
xor x1(OUT, IN1, IN2);
xnor nx1(OUT, IN1, IN2);

// More than two inputs; 3 input nand gate
nand na1_3inp(OUT, IN1, IN2, IN3);

// gate instantiation without instance name (not recommended)
and (OUT, IN1, IN2); // legal gate instantiation
```

Suppose I need 3 instances of a nand gate

(1) could ~~declare~~ instantiate 3 separate times

e.g.

```
nand G1(Y1, A, B);
nand G2(Y2, A, B);
nand G3(Y3, A, B);
```

(2) could instantiate on 1 line

```
nand G1(Y1, A, B), G2(Y2, A, B), G3(Y3, A, B);
```

Table 5-1 Truth Tables for And/Or Gates

and	i1: 0	1	x	z
i2: 0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

nand	i1: 0	1	x	z
i2: 0	1	1	1	1
1	1	0	x	x
x	1	x	x	x
z	1	x	x	x

or	i1: 0	1	x	z
i2: 0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

nor	i1: 0	1	x	z
i2: 0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	x	x

xor	i1: 0	1	x	z
i2: 0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

xnor	i1: 0	1	x	z
i2: 0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

The symbols for these logic gates are shown in Figure 5-2.

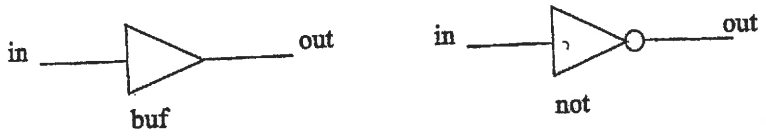


Figure 5-2 Buf and Not Gates

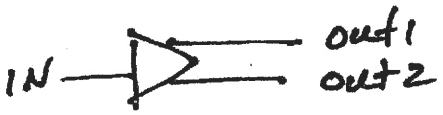
These gates are instantiated in Verilog as shown Example 5-2. Notice that these gates can have multiple outputs but exactly one input, which is the last terminal in the port list.

Example 5-2 Gate Instantiations of Buf/Not Gates

```
// basic gate instantiations.
buf b1(OUT1, IN);
not n1(OUT1, IN);

// More than two outputs
buf b1_2out(OUT1, OUT2, IN);

// gate instantiation without instance name (not recommended)
not (OUT1, IN); // legal gate instantiation
```



Bufif/notif

Gates with an additional control signal on **buf** and **not** gates are also available.

```
bufif1        notif1
bufif0        notif0
```

These gates propagate only if their control signal is asserted. They propagate **z** if their control signal is deasserted. Symbols for *bufif/notif* are shown in Figure 5-3.

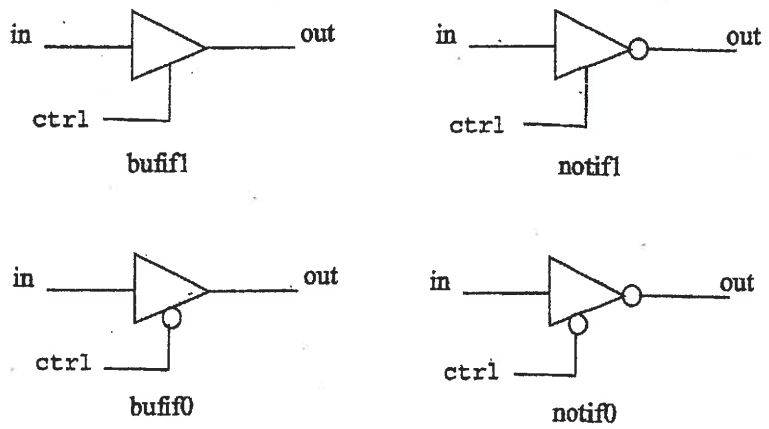
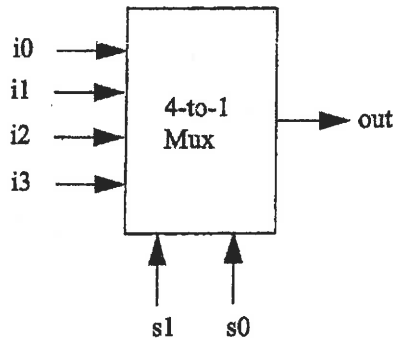


Figure 5-3 Gates Bufif and Notif

Example 5-3 Gate Instantiations of Bufif/Notif Gates

```
//Instantiation of bufif gates.
bufif1 b1 (out, in, ctrl);
bufif0 b0 (out, in, ctrl);

//Instantiation of notif gates
notif1 n1 (out, in, ctrl);
notif0 n0 (out, in, ctrl);
```



s1	s0	out
0	0	I0
0	1	I1
1	0	I2
1	1	I3

Figure 5-4 4-to-1 Multiplexer

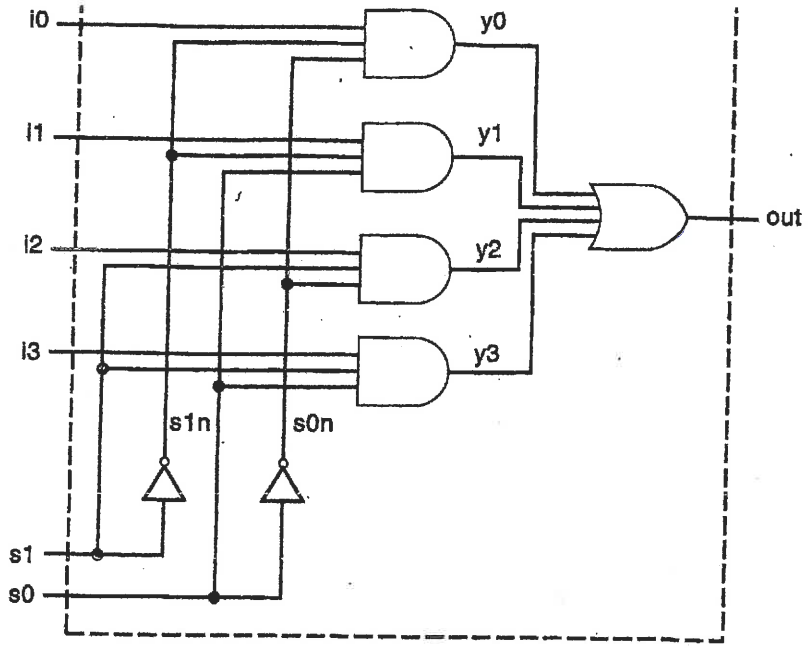


Figure 5-5 Logic Diagram for Multiplexer

Example 5-5 Verilog Description of Multiplexer

```
// Module 4-to-1 multiplexer. Port list is taken exactly from
// the I/O diagram.
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;  // data inputs
input s1, s0;          // select lines

// Internal wire declarations
wire s1n, s0n;
wire y0, y1, y2, y3;

// Gate instantiations

// Create s1n and s0n signals.
not (s1n, s1);
not (s0n, s0);

// 3-input and gates instantiated
and (y0, i0, s1n, s0n);
and (y1, i1, s1n, s0);
and (y2, i2, s1, s0n);
and (y3, i3, s1, s0);

// 4-input or gate instantiated
or (out, y0, y1, y2, y3);

endmodule
```