Introduction to Verilog

(Verification and logic – Verilog)

Hardware Description Language (HDL)

Design Abstraction Ladder

<u>Informal Specification</u>
The customer specifies what the chip/design should do, how fast it should run, etc. A specification is almost always incomplete – it is a set of requirements, not a formal design description..

<u>Behaviour or algorithmic level (We will use this level)</u>
The behaviour description is much more precise than the specification. Behaviour is generally modelled as some sort of executable program. The phase “functional description” is also used. There is no universally agreed definition to distinguish between behavioural and functional descriptions. A behaviour module does not define an implementational architecture. Sometimes the functional description contains more design partitioning information i.e. building blocks.

Design Abstraction Ladder continued..

Register Transfer level RTL or Dataflow level

An architecture model of systems has been specified. The systems time behaviour is fully-specified - we know the allowed input and output values on every clock cycle - but the logic isn't specified as gates. The system is specified as Boolean functions stored in abstract memory locations.

Logic or gate level (We will use this level)

The system is defined in terms of Boolean logic gates, latches, and flip-flops.

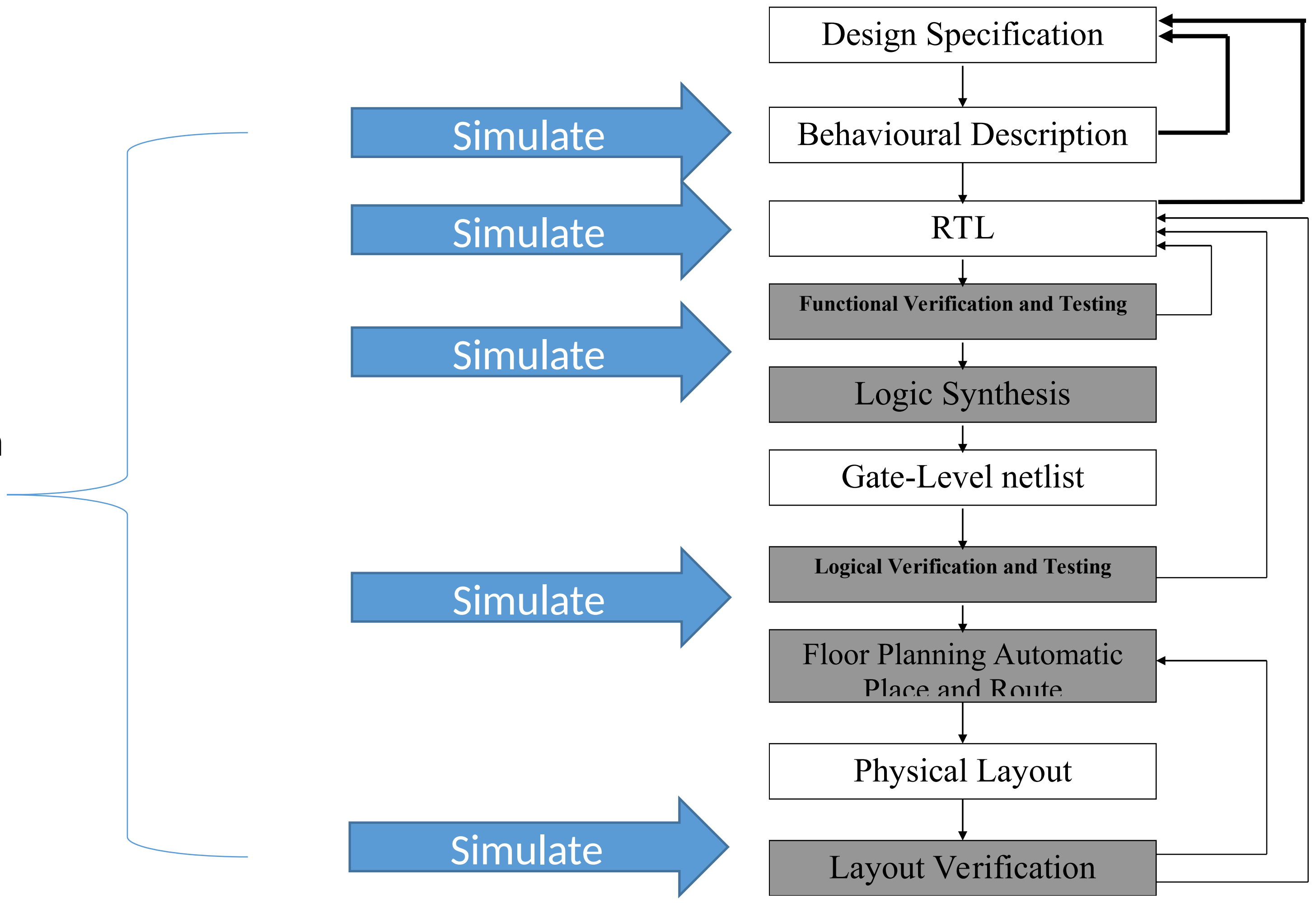
Switch level

The circuit is modelled as a series of switches that are either open or closed to make the logic function.

Circuit level

The system is implemented as transistors with impedances. A set of mathematical equations relate current and voltage i.e. a SPICE simulation model.

Typical Top Down Design Flow



Test Bench Verification file (.wav,vwf)

After Manufacture we can Test the physical system

Hardware Description Languages
HDLs

Designing with HDLs is analogous to computer programming. A textual description with comments is an easier and more efficient way of developing and debugging circuits. Gate level schematics are almost incomprehensible for very complex designs. They are also less portable than textural representations.

Two most popular languages are:

Verification and logic HDL or *Verilog* (Europe) &
Very high speed IC(Vhsic) HDL or *VHDL* (USA - DoD)

Which is best?– IT DEPENDS

See asic-world
See What's the Difference Between VHDL, Verilog, and SystemVerilog? | Electronic Design

History of Verilog

1984	Verilog-XL simulator and language developed by Gateway Design Automation Created by Phil Moorby, Prabhu Goel, Chi-Lai Huang.
1987	Synopsys introduced a Verilog based synthesis tool.
1989	Cadence Design Systems acquired Gateway, and Verilog.
1990	Cadence placed the Verilog language in the public domain.
1995	Verilog HDL became (IEEE Std 1364-1995).
1997	Verilog VCS bought by Viewlogic
1997	Viewlogic bought by Synopsys
1998	Synopsys issues Verilog VCS
2001	A significantly revised version was published in 2001.
2005	Systems Verilog-2005 additional features and OO programming

- The Verilog language roots were based on C-computer language and HILO – *first logic simulator*
- Verilog remains popular as a subset of Systems Verilog.
- System Verilog introduced features to enable easy translation between Verilog and VHDL and can support System on Chip (SoC) design.

"In 1976, Phil Moorby went to **Brunel University** in England to pursue a PhD research program on dynamic timing analysis. But he soon became involved in a project centred on one of the very first HDLs: Hilo. One reason: The Hilo project was funded, allowing Moorby to draw a salary. The Hilo language and simulator were subsequently sold by Genrad well into the 1980s."

www.eetimes.com/verilogs-inventor-nabs-edas-kaufman-award/

Computer Aided Design (CAD) Tools i.e. Quartus

Most CAD Tools will take input from both Verilog and VHDL design descriptions. The language war is over !

They can be used to describe a design at any abstraction level; behavioural, RTL, gate level or switch level.

CAD tools are available that will perform synthesis to aid in the generation of a lower level of abstraction from a higher level i.e. it is possible to automatically translate an RTL description into a physical chip layout without human interaction. This is not to say that a human cannot perform the task more efficiently.

CAD tools allow the simulation of the design at different levels of abstraction. It is possible to verify the design at each sage of development (abstraction level) with that of the previous stage. e.g. the same waveform file (.wvf) can be used to simulate the RTL and gate level designs.

CAD tools will allow you to specify and simulate different partitions (hierarchy) of the design at different levels of abstraction. i.e. mixed level simulation. In this way very large designs can be simulated in an acceptable time period.

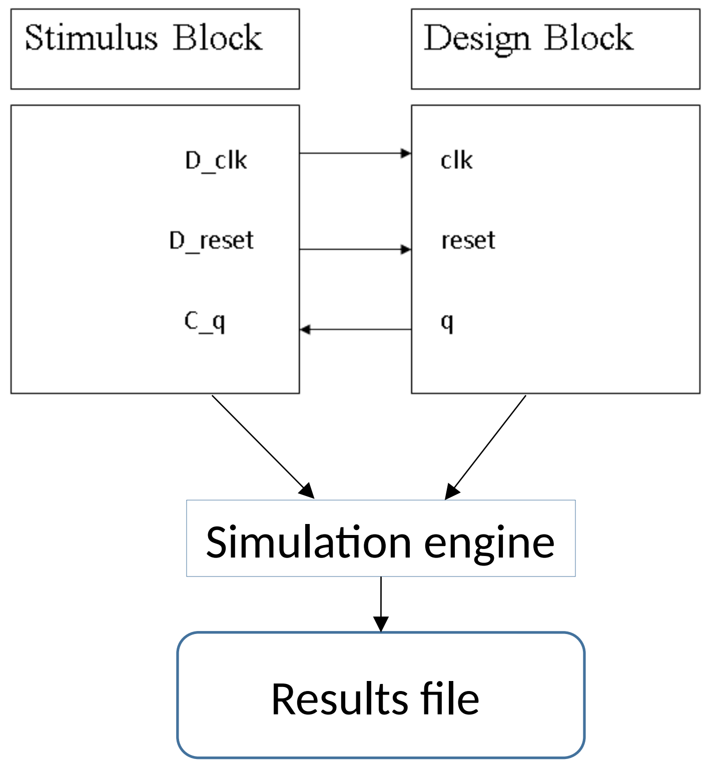
Three main components to a simulator.

- Simulation engine
  e.g Quartus simulator engine - multisim-altera

- Design description Block
  Graphical description e.g *.bdf
  Verilog description file e.g. *.v
  STD file e.g. .smf

- Stimulus description Block
  Waveform description file e.g. *.scf or .vwf
  Verilog stimulus commands (test bench) can be added to the design block (.v)
  This approach will not be discussed in this course.
  But you will see it in text books/web, look out for "# commands"

Quartus approach



Verilog Design Block – Modules

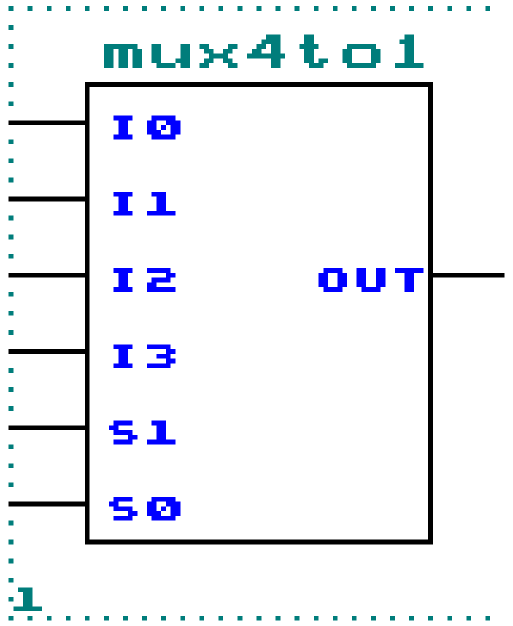
Design Block can be of any abstraction level

It has a header that defines the I/O (ports) and a body that describes the designs function.

```
module  user_defined_name ( list of ports);
//port declarations
------
-----
//Define variables and constants etc..
- - -
- - -
//logic gates or functions body
- - - -
- - - -
endmodule
```

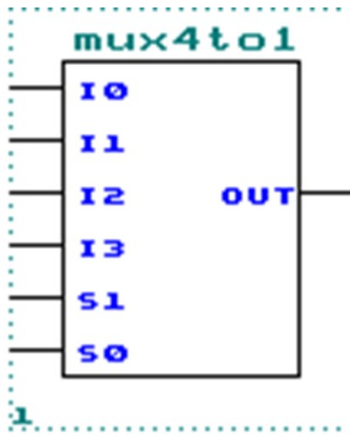
PORTS

Ports provide the interface by which a module can communicate with its environment. E.G. input and output pins of a chip or input and output terminals of a design hierarchical partition.

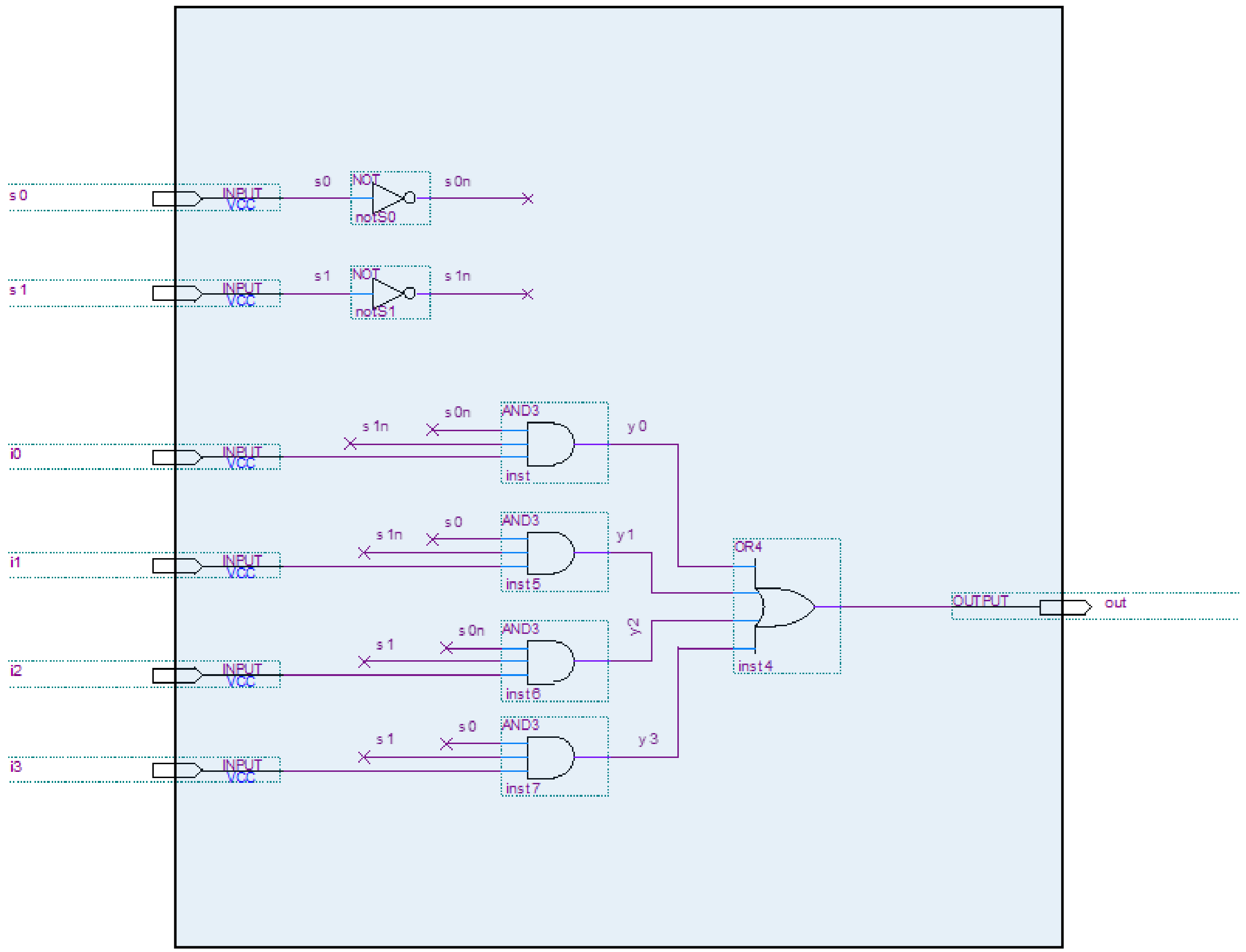


s1	s0	out
0	0	i0
0	1	i1
1	0	i2
1	1	i3

```
module mux4to1(out,I0,I1,I2,I3,S1,S0);
input I0,I1,I2,I3,S1,S0;
output out;
----
----
----
endmodule
```



```
module mux4to1(out,I0,I1,I2,I3,S1,S0);
input I0,I1,I2,I3,S1,S0;
output out;
----
----
----
endmodule
```



Port Declarations (Example)
(non –ANSI Style)

Types of Port : input, output, inout (i.e. tri-state)

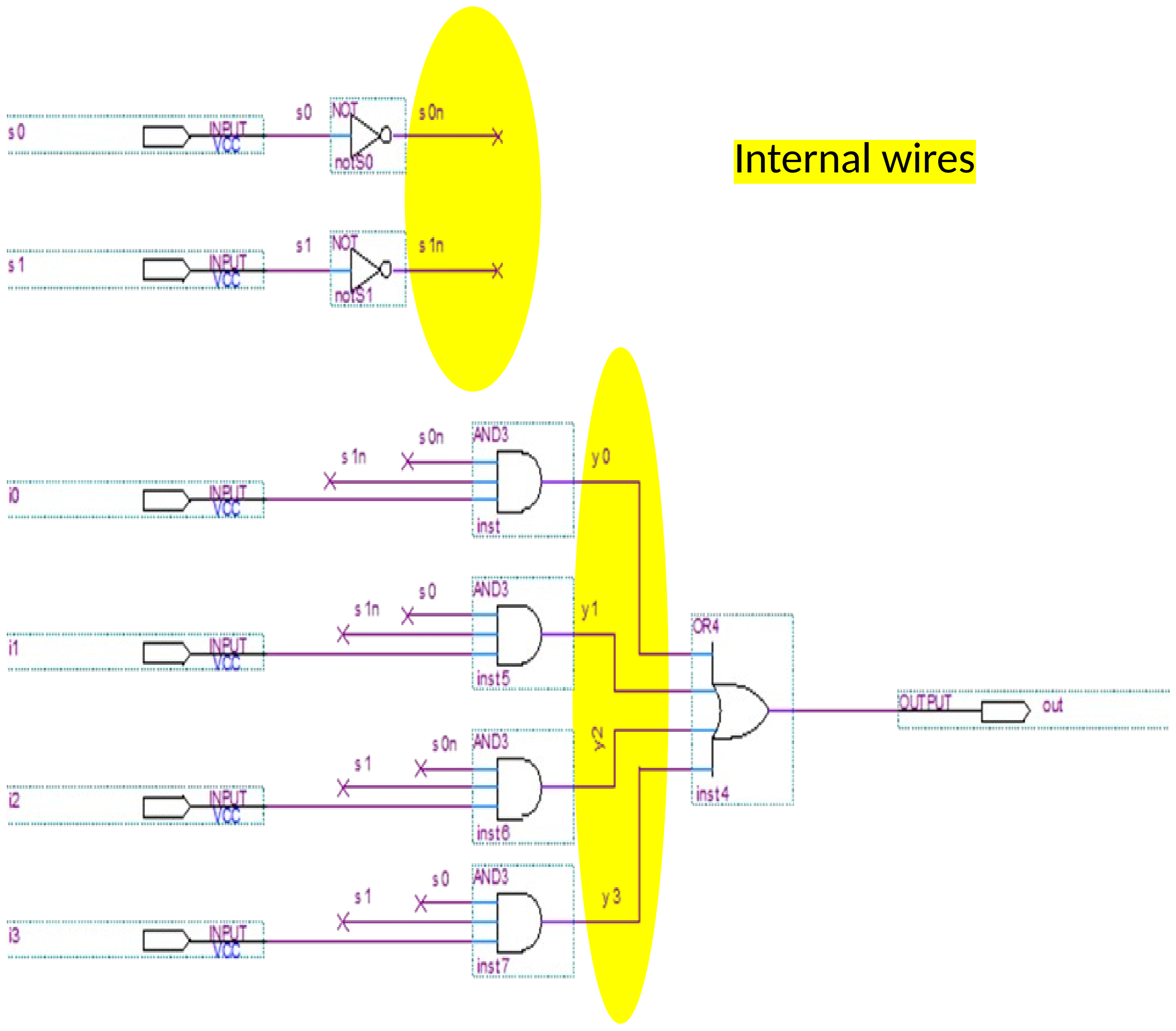
```
module mux4to1(out, i0,i1,i2,i3,s0,s1);

//begin port declarations
input i0,i1,i2,i3,s0,s1;  /* case sensitive */
output out;
//end port declarations


//Define the internal nets and constants etc..
-    - - -
-    - - -
//module internal description logic gates or functions
-    - - - -
-    - - - -

endmodule
```

Port Declarations (Alternative method)
(ANSI Style)

Types of Port : input, output, inout

```
module mux4to1(
output out,  // output port
input i0,      //  use comment to explain ports
input i1,      //  data input i1
input i2,
input i3,
input s0,       // select or control input port
input s1);

//Define the internal nets and constants etc..
-    - - -
-    - - -
//module internal description logic gates or functions
-    - - - -
-    - - - -

endmodule
```

```
module mux4to1gate (out,i0,i1,i2,i3,s1,s0);
//gate level Verilog module

// ports
output out;
input i0, i1, i2, i3; // data – can use two lines
input s1, s0;          // control

//internal connections
wire s1n, s0n;         //can all be on one line
wire y0,y1, y2, y3; //if all the same type

//logic gates
-----
-----
-----
endmodule
```



Internal wires

Some Primitive Logic Gate Types

- <gate_type> ::= and | nand | or | nor | xor | xnor | buf | not
- <tri_state_gate> ::= bufif0 | bufif1 | notif0 | notif1

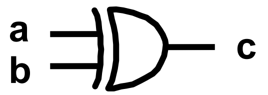
<gate_type>(out, in1, in2, …, inm)

<tri_state_gate>(out, in, enable)

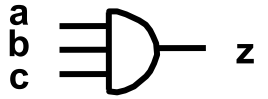
Examples, Logic Values: - 0, 1, X, Z

Note: for a primate logic gate the First parameter is the output. All primitive gates have a single output

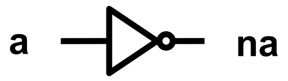
xor(c, a, b);



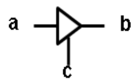
and(z, a, b, c);



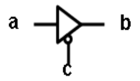
not(na, a);



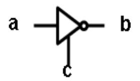
bufif1(b, a, c);



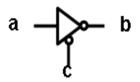
bufif0(b, a, c);



notif1(b, a, c);

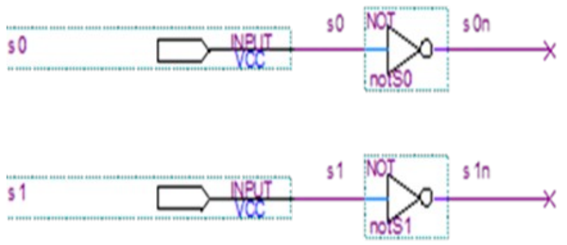


notif0(b, a, c);

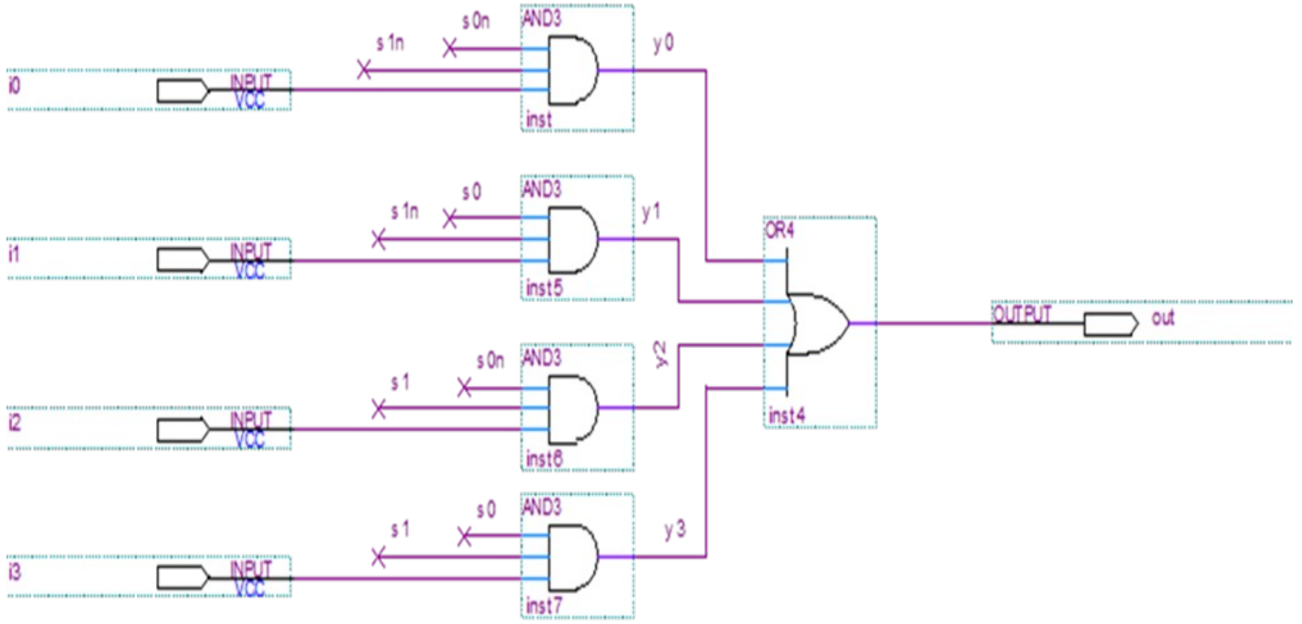


```
module mux4to1gate (out,i0,i1,i2,i3,s1,s0);
//gate level module

// ports
output out;
input i0,i1,i2,i3, s1, s0;

//internal connections
wire s1n, s0n;
wire y0,y1, y2, y3;

//logic gates
not (s1n, s1);
not (s0n, s0);
and (y0, i0, s1n, s0n);
and (y1, i1, s1n, s0);
and (y2, i2,s1, s0n);
and (y3, i3, s1, s0);
or (out, y0, y1, y2, y3);

endmodule
```

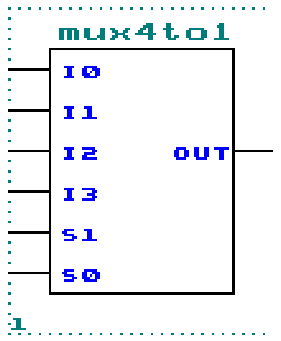
Home work: try to simulate the Verilog design!



out = (~s1 & s0& i0) + (~s1 & s0& i1) + (s1 & ~s0& i2) + (s1 & s0& i3)



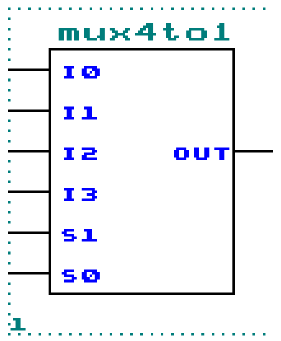
Slightly Higher Level of Abstraction
Data Flow modelling



s1	s0	out
0	0	i0
0	1	i1
1	0	i2
1	1	i3

```
module mux4to1logic (out,i0,i1,i2,i3,s1,s0);
// data flow example 1 primitive gates are replaces by
//functional discrete operators.

//ports
output out;
input i0,i1,i2,i3, s1, s0;

//logic equations are in  first canonical form
//assign used for continuous combinational statements
assign out =          (~s1 & ~s0 & i0) |
                      (~s1 & s0 & i1) |
                      (s1 & ~s0 & i2) |
                      (s1 & s0  & i3) ;
endmodule

// Note : ~  =  "not" , & = "and", pipe | = "or"  logic operators
```

Data Flow modelling for combination logic



s1	s0	out
0	0	i0
0	1	i1
1	0	i2
1	1	i3

```
module mux4to1cond (out,i0,i1,i2,i3,s1,s0);
// data flow example 2 functional operators
//ports
output out;
input i0,i1,i2,i3, s1, s0;


//Use nested conditional operator
assign out = s1 ? ( s0 ? i3 : i2) : (s0 ? i1 : i0);

//explanation of conditional operator
// if (s1 == 1) then
//          if (s0 == 1) then     out=i3
//                       else     out=i2
//          else
//          if (s0 == 1) then     out=i1
//                       else     out=i0

endmodule
```

Note
Read " ==" as "is equal too"
Read "=" as " becomes equal too"

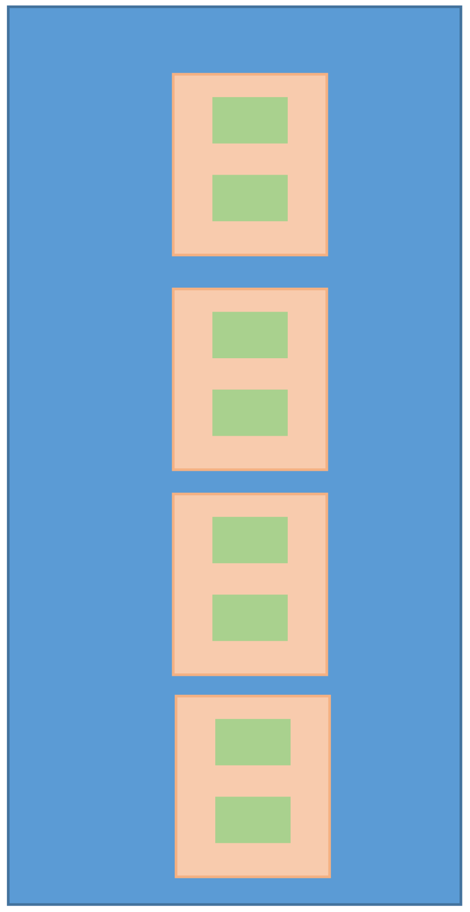
Even Higher Level of Abstraction Behavioural/ Functional modelling

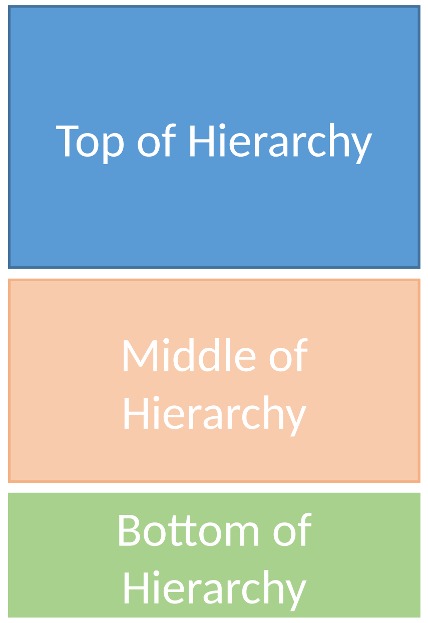
EXPLAINED IN DETAIL NEXT LECTURE!

s1	s0	out
0	0	i0
0	1	i1
1	0	i2
1	1	i3

```
module mux4to1case (out,i0,i1,i2,i3,s1,s0);
// Behavioural/Functional  Modelling
//ports
output out;
input i0,i1,i2,i3,s1, s0;
//variables
 reg out;  // you only have reg definitions at functional level

/*the always statement will execute the case statement if any input changes
functional Verilog needs always statements to define events for simulation.
it is NOT a continuous statement*/

always @(s0 or s1 or i0 or i1 or i2 or i3);
        // switch based on control signals s0, s1
case({s1,s0})
            2'd0        :out = i0;// 2'd0  = 2 bit decimal 0   i.e.  s1=0, s0=0
            2'd1        :out = i1;
            2'd2        :out = i2;// 2'd0  = 2 bit decimal 2   i.e. s1=1, s0=0
            2'd3        :out = i3;
            default:    out=1'bx;// 1'b0  = 1 bit binary don't care  i.e. out=x
endcase

endmodule
```

Design Hierarchy - Partitions and Building blocks





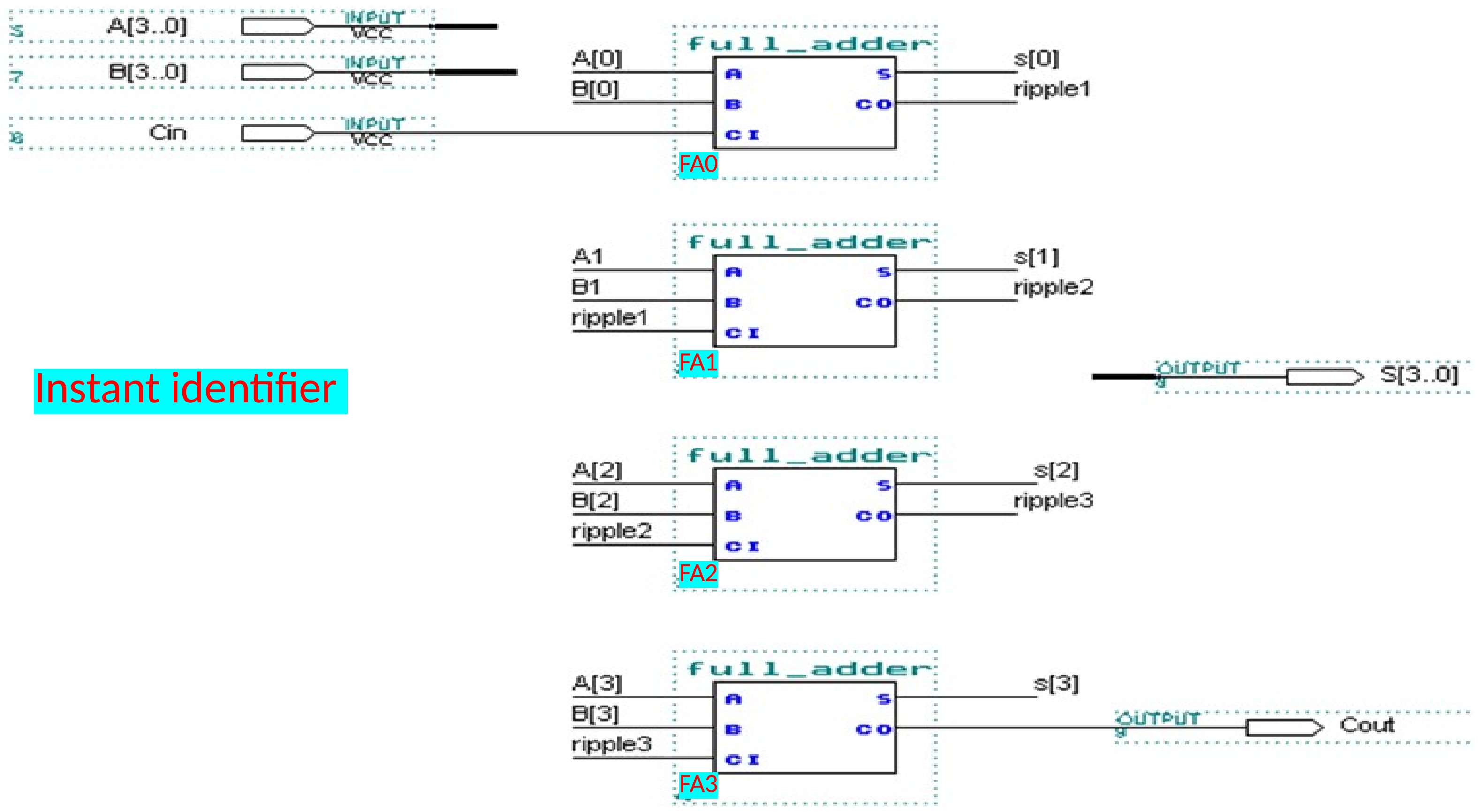
Let's build a GATE level Verilog module of a 4 bit Adder.

We will use multiple Verilog modules to partition the design and use a hierarchical approach.

4 – bit Adder example

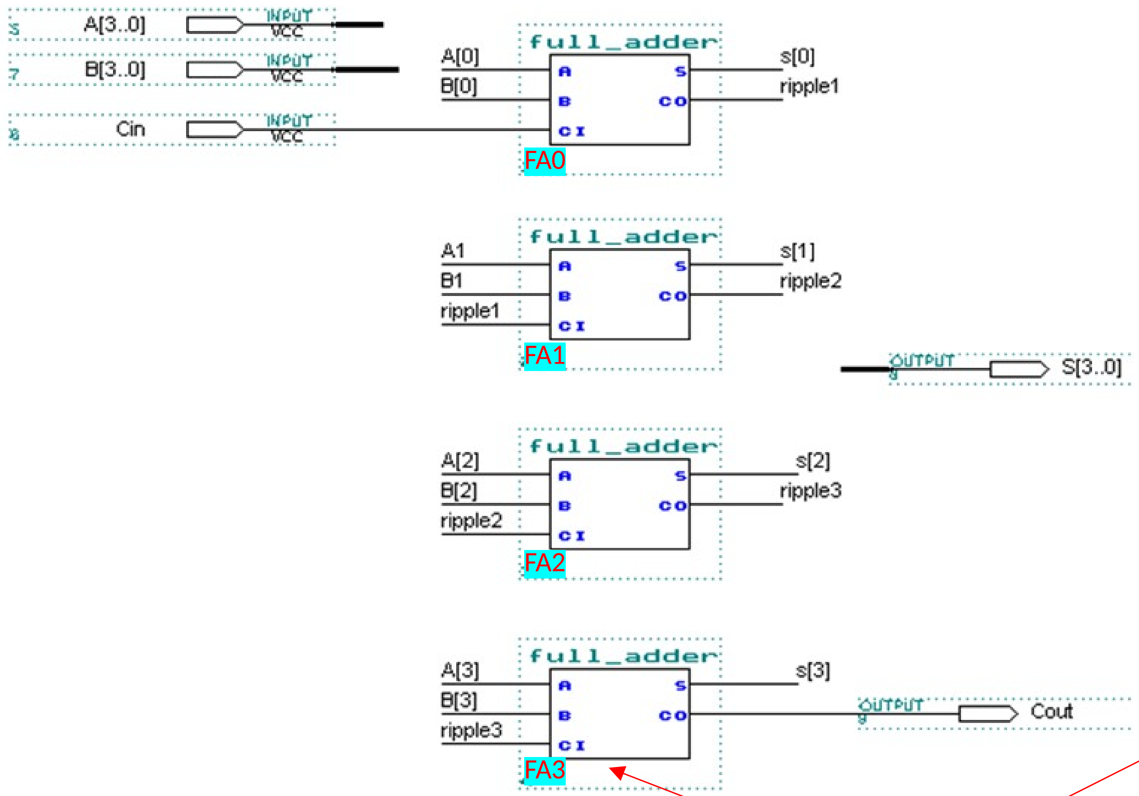
Adder
Top of Hierarchy

Design Hierarchy - Partitions and Building blocks
4 bit binary added made up from 4 full adders :- level 1 lab?
Each partition or block is at a gate level of abstraction



“Adder”
Top of Hierarchy

Design Hierarchy - Partitions and Building blocks
4 bit binary added made up from 4 full adders :- level 1 lab?



Macro cell – Verilog module name is full_adder

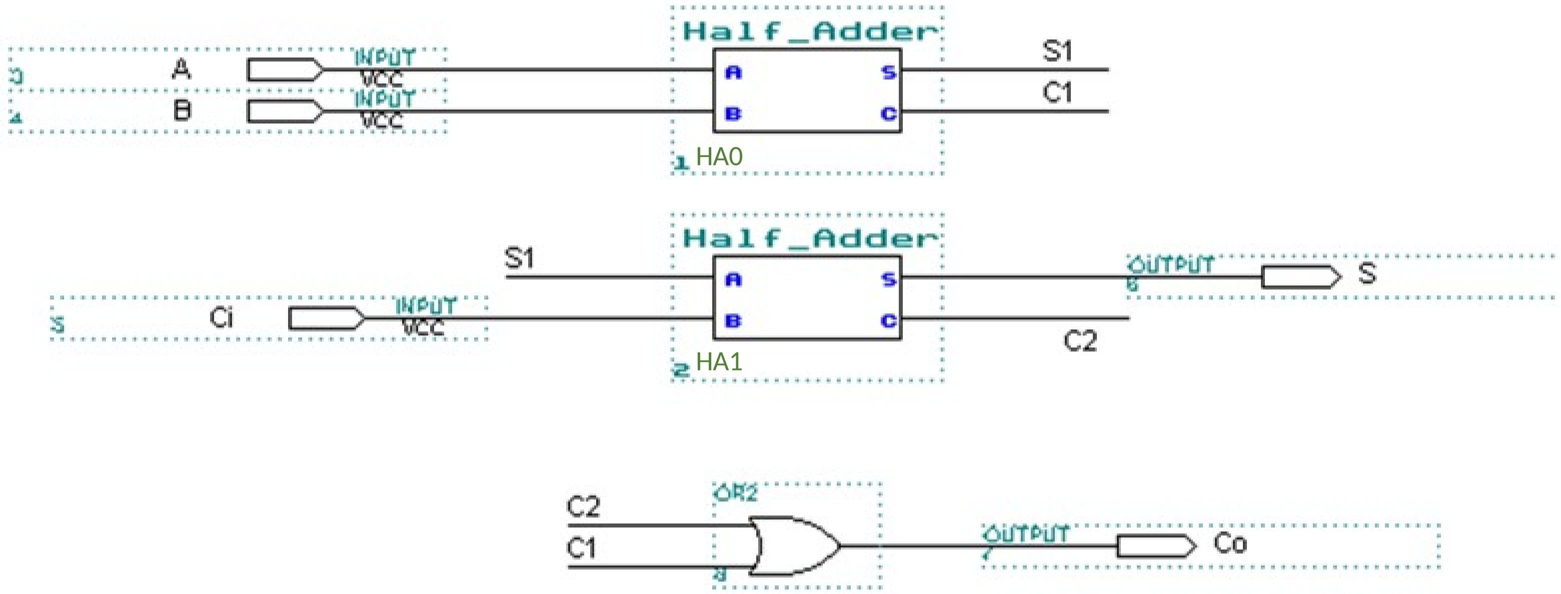
```
module Adder(S, Cout, A, B, Cin);
// Top of Hierarchy

//ports
input [3:0] A, B;  //  4 bit bus
input Cin;
output [3:0] S;
output Cout;

//internal connectors
wire ripple1,ripple2,ripple3;

//module instantiations
//I will use Positional Association – simple..

full_adder FA0 (S[0], ripple1, A[0], B[0], Cin);
full_adder FA1 (S[1], ripple2, A[1], B[1], ripple1);
full_adder FA2 (S[2], ripple3, A[2], B[2], ripple2);
full_adder FA3 (S[3], Cout, A[3], B[3], ripple3);

endmodule
```

Note: An instantiation name is needed for user defined macro cells

“Adder”
Top of Hierarchy

Need to generate a module for the “full adder”
A “full added” is made out of 2 half adders

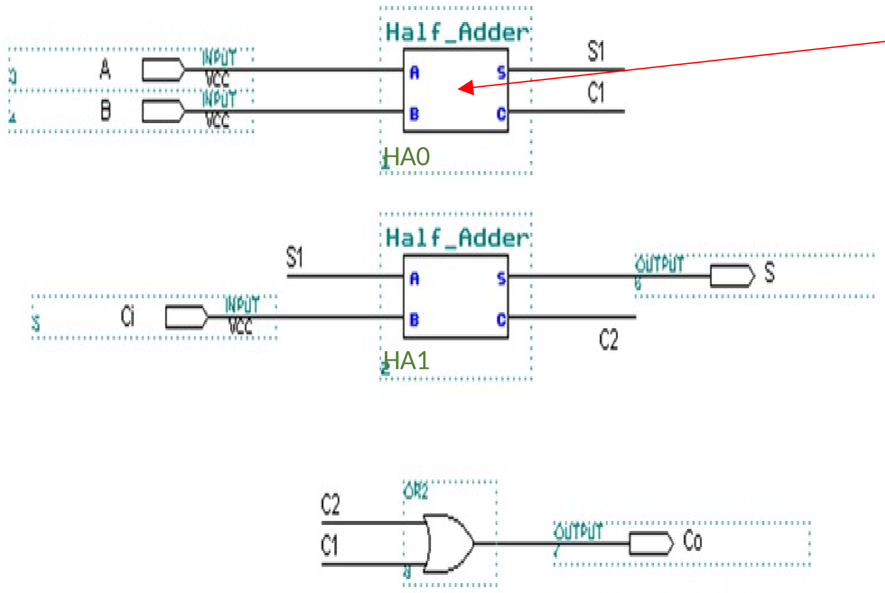
“Full adder”
Middle of
Hierarchy



"Full adder" Middle of Hierarchy

Need to generate a module for the full adder
A full added is made out of 2 half adders

Macro cell name



```
module full_adder(S, Co, A, B, Ci);
/*Middle of hierarchy, it is called four times to
generate a four bit adder.*/
//ports
        input A, B, Ci;
        output S, Co;
//internal connectors
        wire S1, C1, C2;
        //gates & module instantiations
                Half_Adder HA0(S1, C1, A, B);
                Half_Adder HA1(S, C2, S1, Ci);
                or(Co, C1, C2);
        endmodule
```

Note: An instantiation name is needed for macro cells

Note: An instantiation name is NOT needed for primitive cell

“Adder”
Top of Hierarchy

Finally need to define a module for the “Half_Added”

“Full adder”
Middle of
Hierarchy

“Half_Adder”
Bottom of Hierarchy

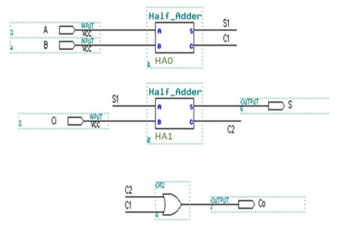
Draw logic diagram

```
module Half_Adder(S, C, A, B);
/*lowest level Hierarchy, it is called twice for every
full_adder i.e 8 times for 4 bit adder*/
//ports
        input A, B;
        output S, C;
//primitive gates
        xor (S, A, B);
        and (C, A, B);
endmodule
```

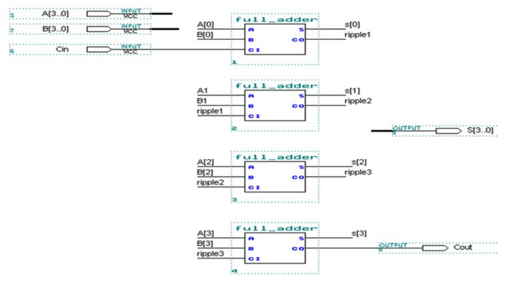
Note: no instantiation name is needed for primitives

Cellular Adder – Hierarchical Design

```
//*************************************
module Half_Adder(S, C, A, B);
//lowest level Hierarchy, it is called by full_adder
//Two times
//ports
      input A, B;
      output S, C;
//primitive gates
      xor(S, A, B);
      and(C, A, B);
endmodule

//*************************************
module full_adder(S, Co, A, B, Ci);
//Middle of hierarchy, it is called four time to
//generate a four bit adder.
//ports
      input A, B, Ci;
      output S, Co;
//internal connectors
      wire S1, C1, C2;
//gates & module instantiations
      Half_Adder HA0(S1, C1, A, B);
      Half_Adder HA1(S, C2, S1, Ci);
      or(Co, C1, C2);
endmodule

//*************************************
```

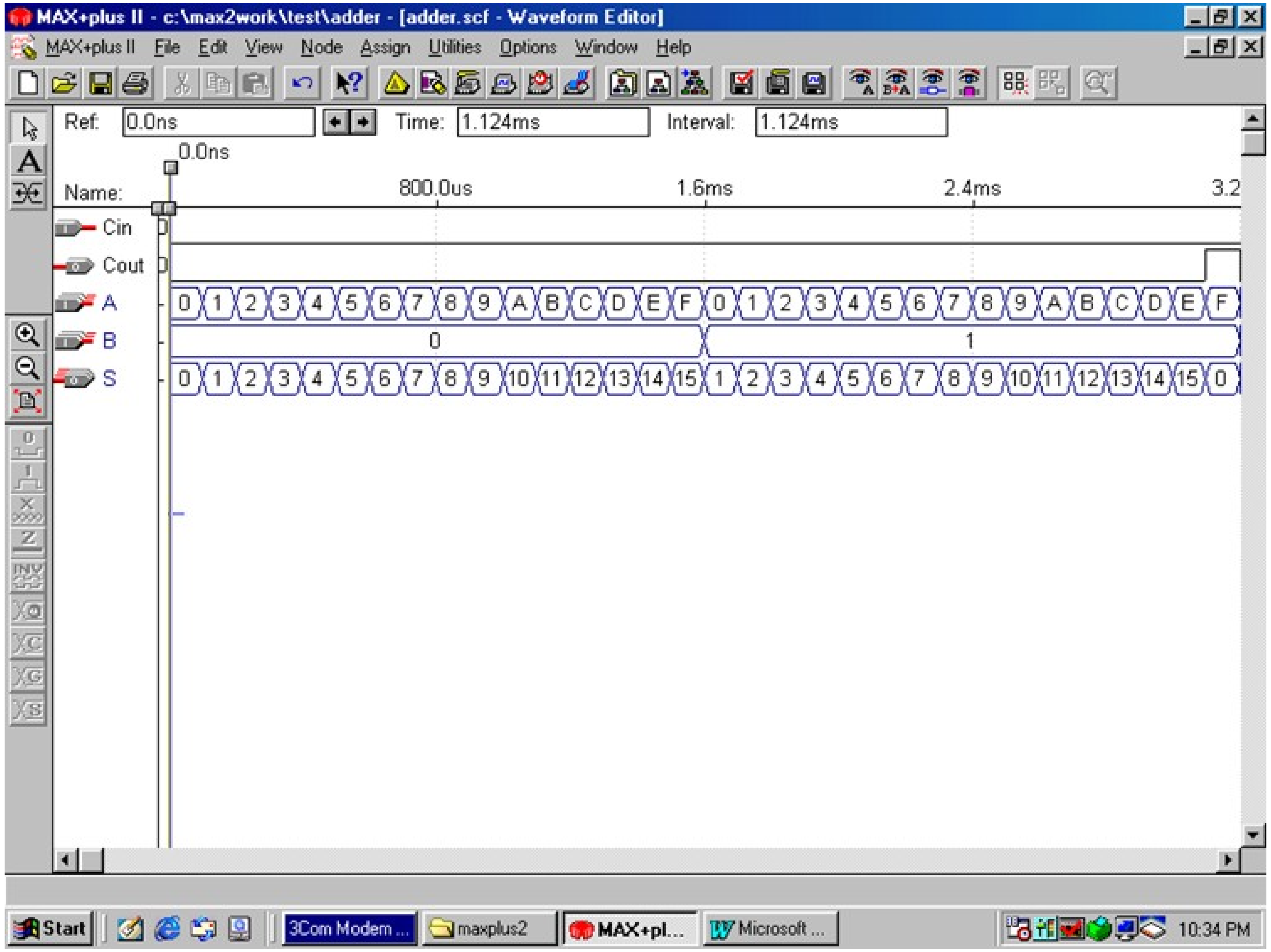


```
//*************************************

module Adder(S, Cout, A, B, Cin);
// Top of Hierarchy

//ports
input [3:0] A, B;  //  4 bit bus
input Cin;
output [3:0] S;
output Cout;

//internal connectors
wire ripple1,ripple2,ripple3;

//module instantiations
full_adder FA0(S[0], ripple1, A[0], B[0], Cin);
full_adder FA1(S[1], ripple2, A[1], B[1], ripple1);
full_adder FA2(S[2], ripple3, A[2], B[2], ripple2);
full_adder FA3(S[3], Cout, A[3], B[3], ripple3);

endmodule
//*************************************
```



The above modules can all be written into one Verilog .v file. The file name must be the same as the module at the Top of the Hierarchy.
e.g.   Adder.v

Demo of 4 bit adder gate level Verilog design



MAX+plus II - c:\max2work\test\adder - [adder.scf - Waveform Editor]
MAX+plus II File Edit View Node Assign Utilities Options Window Help
Start 3Com Modem ... maxplus2 MAX+pl... Microsoft ... 10:34 PM

Ref: 0.0ns Time: 1.124ms Interval: 1.124ms

0.0ns

Name: 800.0us 1.6ms 2.4ms 3.2

```
Cin   0
Cout  0
A     0 1 2 3 4 5 6 7 8 9 A B C D E F 0 1 2 3 4 5 6 7 8 9 A B C D E F
B     0                               1
S     0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0
```