Verilog Introductions

Overview of Verilog – an HDL (Hardware Description Language)

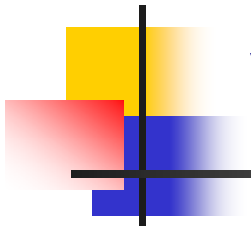
- Verilog is a *hardware description language* (HDL)
- *Verilog is used by several companies in the commercial chip design and manufacturing sector today*. It is rapidly overtaking the rival HDL called VHDL
- Verilog allows a designer to develop a complex hardware system, e.g., a VLSI chip containing millions of transistors, by defining it at *various levels of abstraction*

Register Transfer Level (RTL)

- at the (highest) *behavioral*, or *algorithmic*, *level* the design consists of C-like procedures that express functionality without regard to implementation
- at the *dataflow level* the design consist of specifying how data is processed and moved between registers

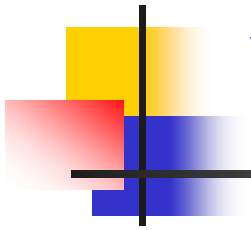
Structural Level

- at the *gate level* the structure is defined as an interconnection of logic gates
- at the (lowest) *switch level* the structure is an interconnection of transistors



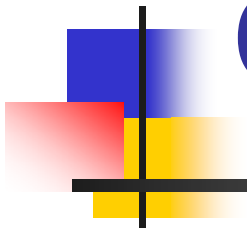
Verilog

- Verilog allows the designer to *simulate and verify* the design at each level
- EDA (*electronic design automation*) tools help the designer to move from higher to lower levels of abstraction
  - *Behavioral synthesis tools* create dataflow descriptions from a behavioral description
  - *Logic synthesis tools* convert an RTL description to a switch level interconnection of transistors, which is input to an *automatic place and route tool* that creates the chip layout
- With Verilog and EDA tools one could sit at a computer at home, design a complex chip, email the design to a *silicon foundry* in California, and receive the fabricated chip through regular mail in a few weeks!
- The Verilog environment is that of a programming language. Designers, particularly with C programming experience, find it easy to learn and work with



Verilog Resources

1. “Verilog HDL”, by Palnitkar (department library)
2. “Verilog Digital Computer Design: Algorithms to Hardware”, by Arnold (department library)
3. John Sanguinetti's Verilog Tutorial: http://www.vol.webnexus.com/
4. Gerard Blair's Verilog Tutorial: http://www.see.ed.ac.uk/~gerard/Teach/Verilog/index.html
5. ALDEC's Verilog Tutorial: http://www.aldec.com
6. DOULOS's Verilog Tutorial: http://www.doulos.com
7. Other on-line resources
8. Class website



CAD Tool Flow (참고자료)

Design decision Trade Off (참고자료)

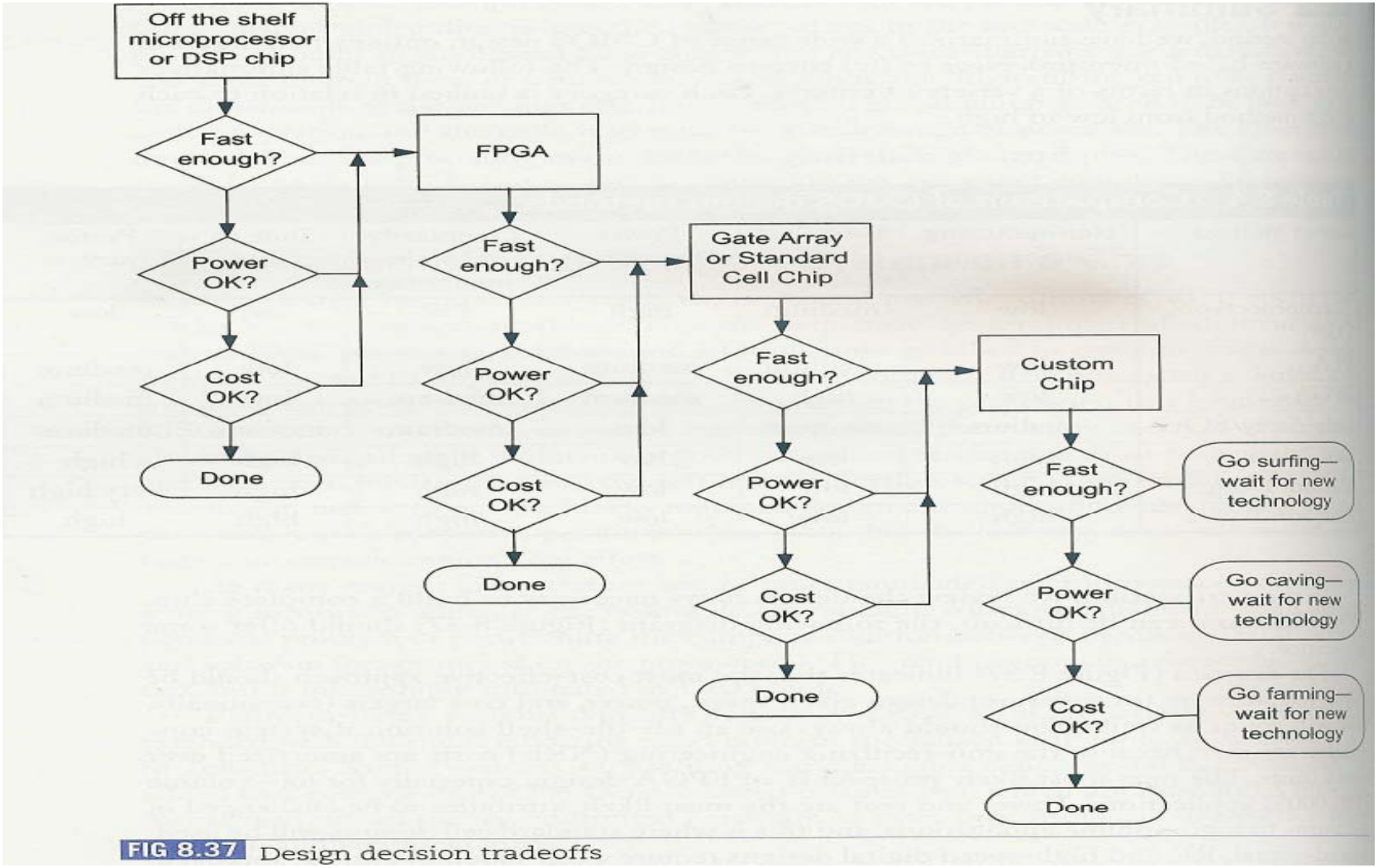


FIG 8.37 Design decision tradeoffs

Generalized Design Flow (참고자료)

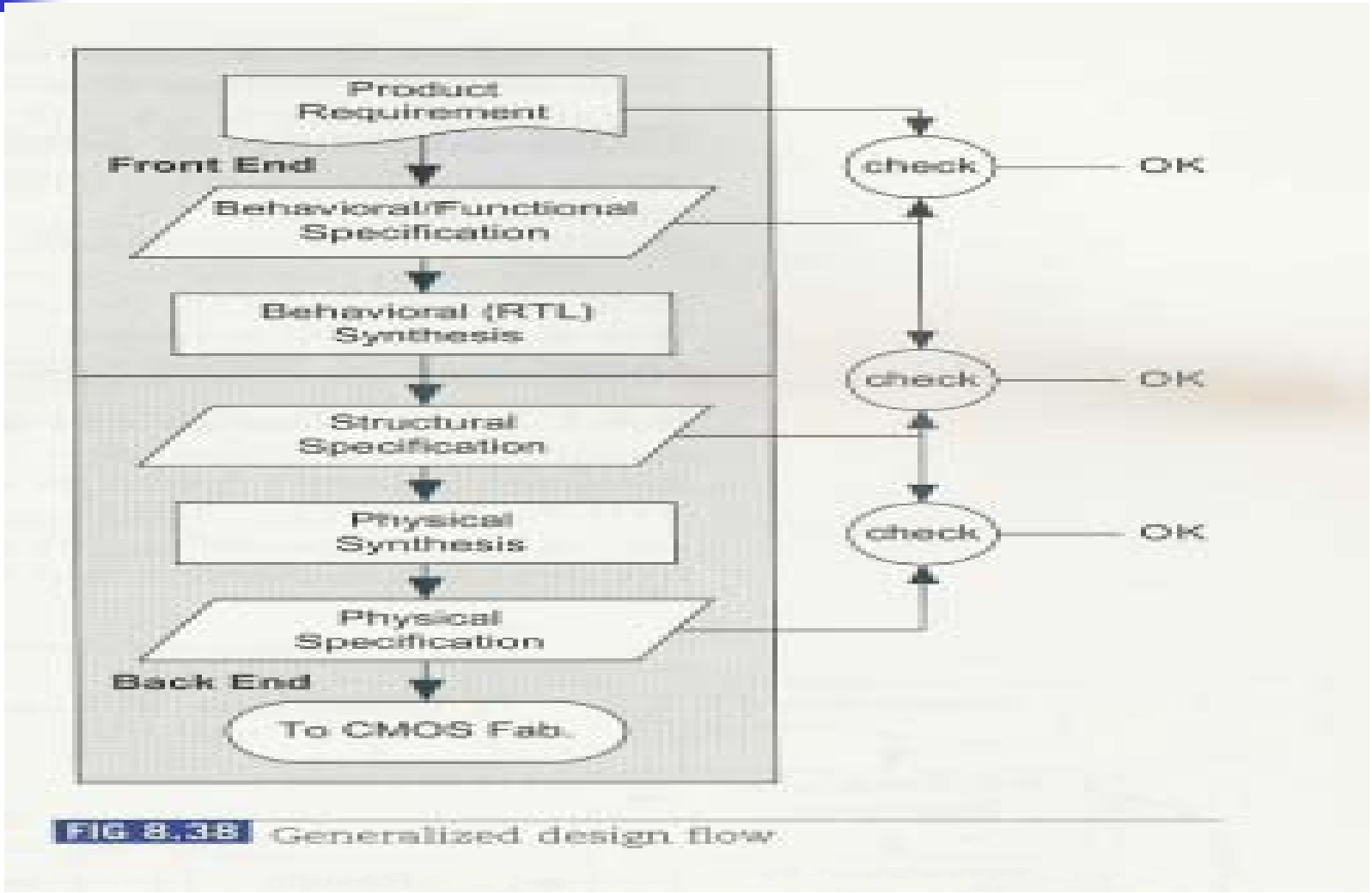


FIG 8.38 Generalized design flow

RTL Synthesis Flow (참고자료)

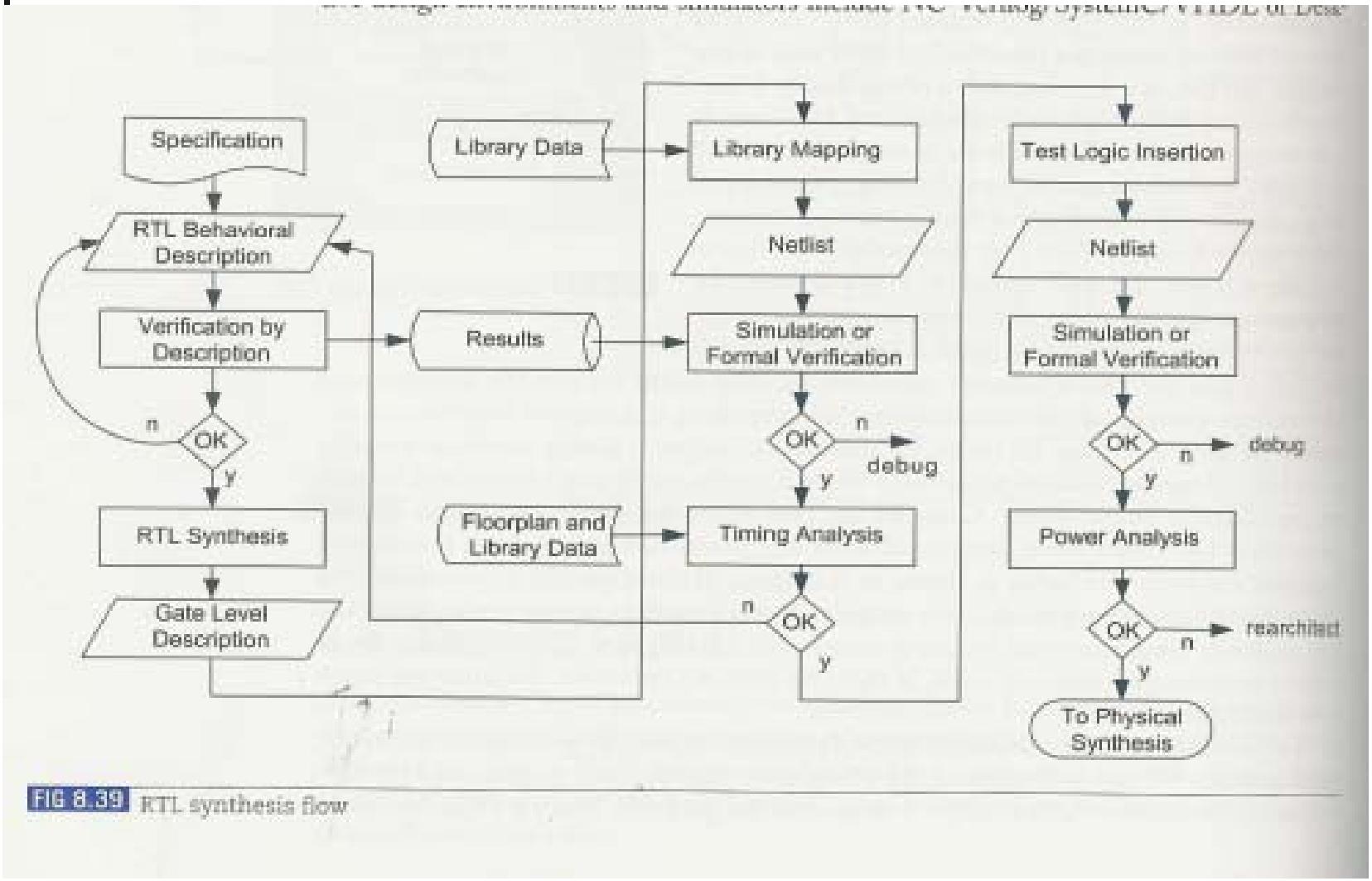
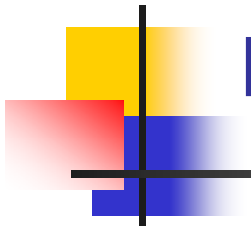
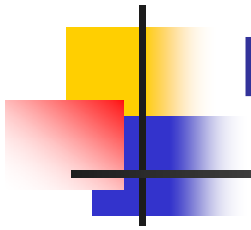


FIG 8.39 RTL synthesis flow



Learning Verilog

- Verilog is essentially a programming language – similar to C with some Pascal-like constructs
- *The best way to learn any programming language is from live code*
- We will get you started by going through several example programs and explaining the key concepts
- *We will not try to teach you the syntax line-by-line*: pick up what you need from the books and on-line tutorials
- *Tip*: Start by copying existing programs and modifying them incrementally making sure you understand the output behavior at each step
- *Tip*: The best way to understand and remember a construct or keyword is to *experiment with it in code*, not by reading about it
- *We shall not design at the switch (transistor) level in this course* – the lowest level we shall reach is the gate level. The transistor level is more appropriate for an electronics-oriented course



Example Code

- The example code that follows is mostly from the two on-line tutorials or Palnitkar's book or written in-house. See the source code in the Examples directory for authorship information
- The transparency title shows the source file name and (in parentheses) if it is in a sub-directory of Examples
- Comments in the original source have often been deleted in the transparency and replaced with text-box annotation

helloWorld.v

A **procedural programming language** provides a programmer a means to define precisely each step in the performance of a task. The programmer knows what is to be accomplished and provides through the language step-by-step instructions on how the task is to be done. Using a procedural language, the programmer specifies language statements to perform a sequence of algorithmic steps. Procedural programming is often a better choice than simple sequential or unstructured

```
module helloWorld ;
    initial
    begin
        $display ("Hello World!!!");
        $finish;
    end
endmodule
```

Modules are the unit building-blocks (components) Verilog uses to describe an entire hardware system. Modules are (for us) of three types: *behavioral*, *dataflow*, *gate-level*. We ignore the *switch-level* in this course.

This module is behavioral. Behavioral modules contain code in *procedural* blocks.

System calls.

This is a *procedural* block.
There are two types of procedural blocks: *initial* and *always*.

More than one statement must be put in a *begin-end* group.

blocksTime1.v

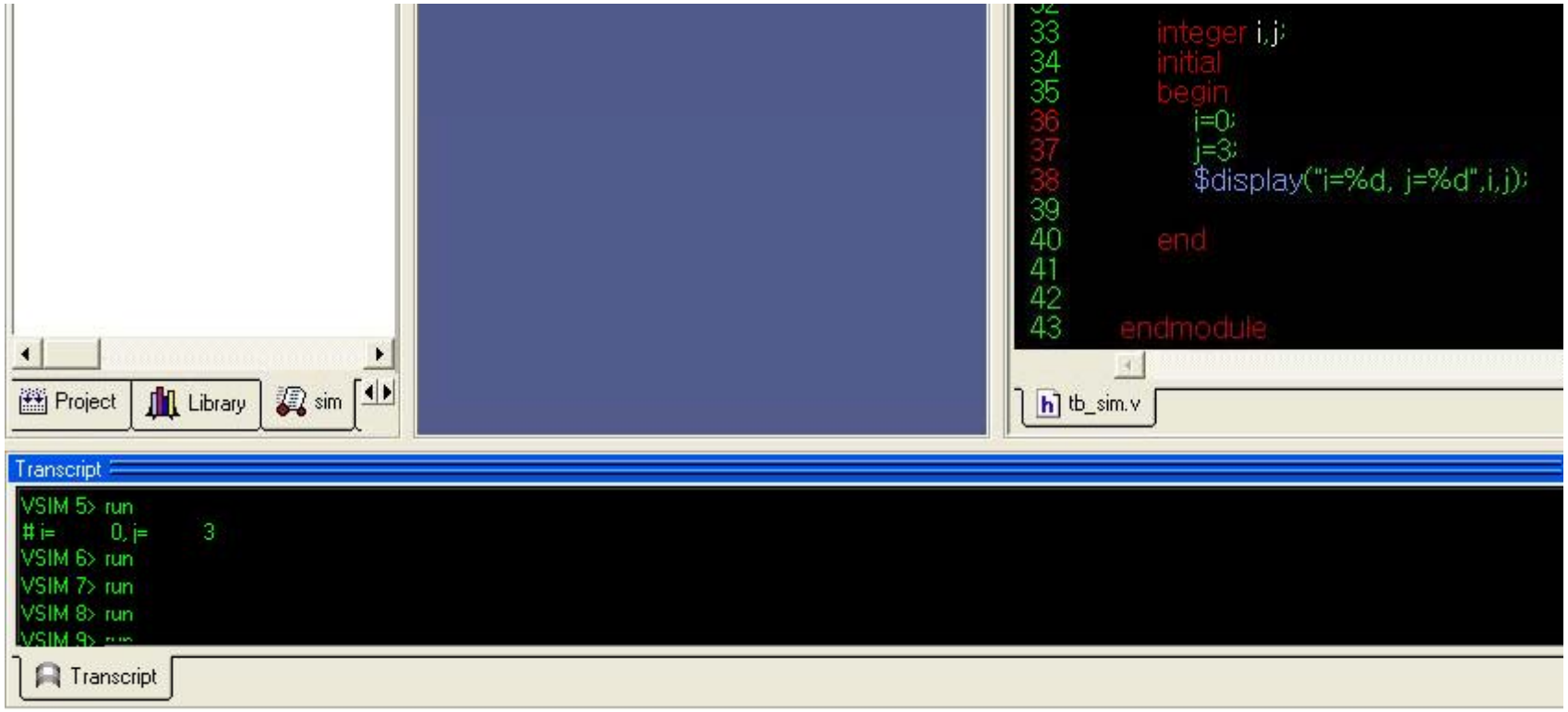
```
module blocksTime1;
   integer i, j;

   initial
      begin
      i = 0;
      j = 3;
      $display( "i = %d, j = %d", i, j );
      $finish;
      end
endmodule
```

Another behavioral module.

Integer data type: other types are time, real and realtime (same as real).

One initial procedural block.

blocksTime1.v – Simulation Result



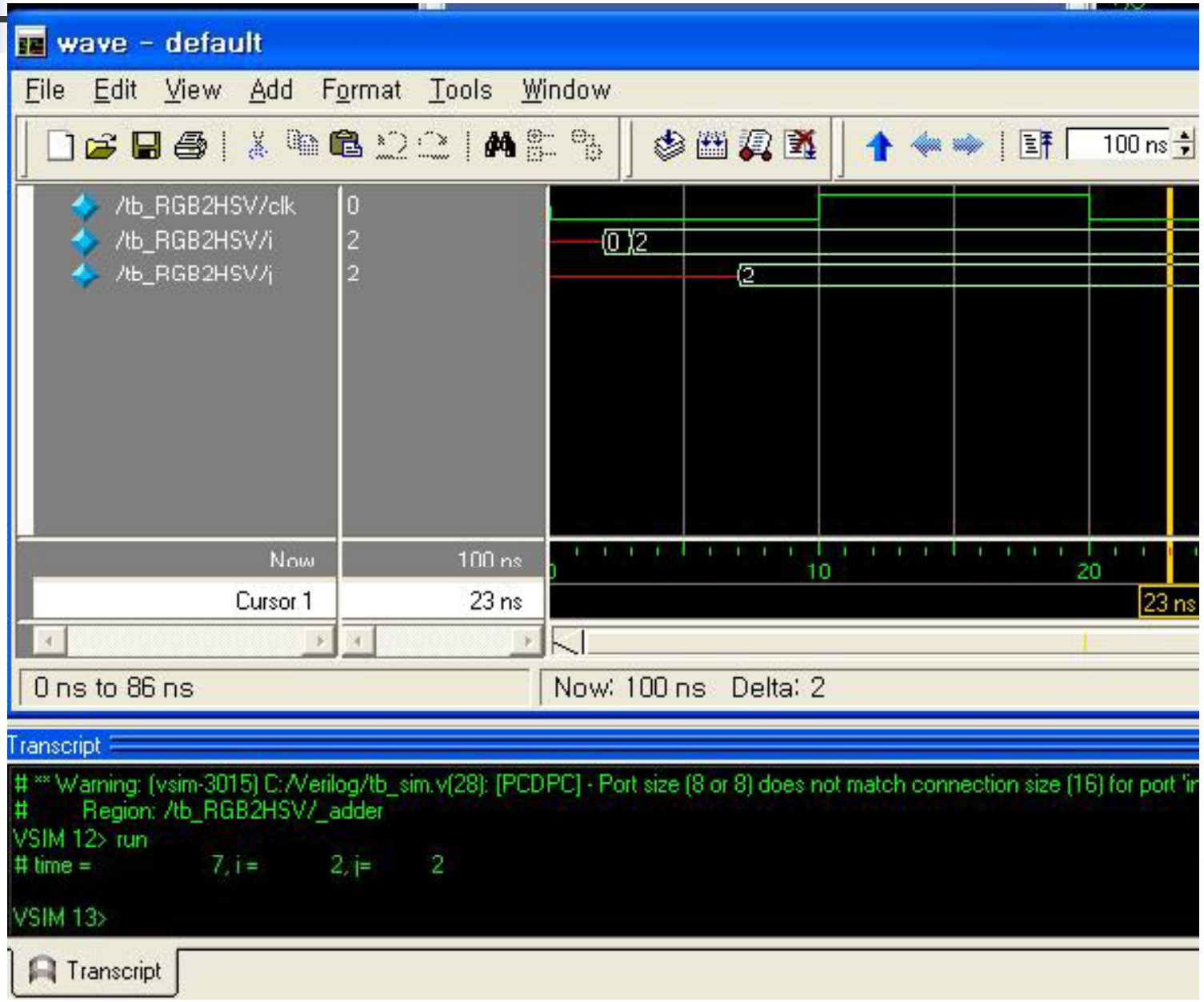
blocksTime2.v

```
module blocksTime2;
   integer i, j;

   initial
      begin
      #2 i = 0;
      #5 j = i;
      $display( "time = %d, i = %d, j = %d", $time, i, j );
      end

   initial
      #3 i = 2;

   initial
      #10 $finish;
endmodule
```

Time *delay* models signal propagation delay in a circuit.

Multiple initial blocks.
Delays add within each block, but different initial blocks all start at time $time = 0 and run in *parallel* (i.e., *concurrently*).

blocksTime2.v – Simulation Result



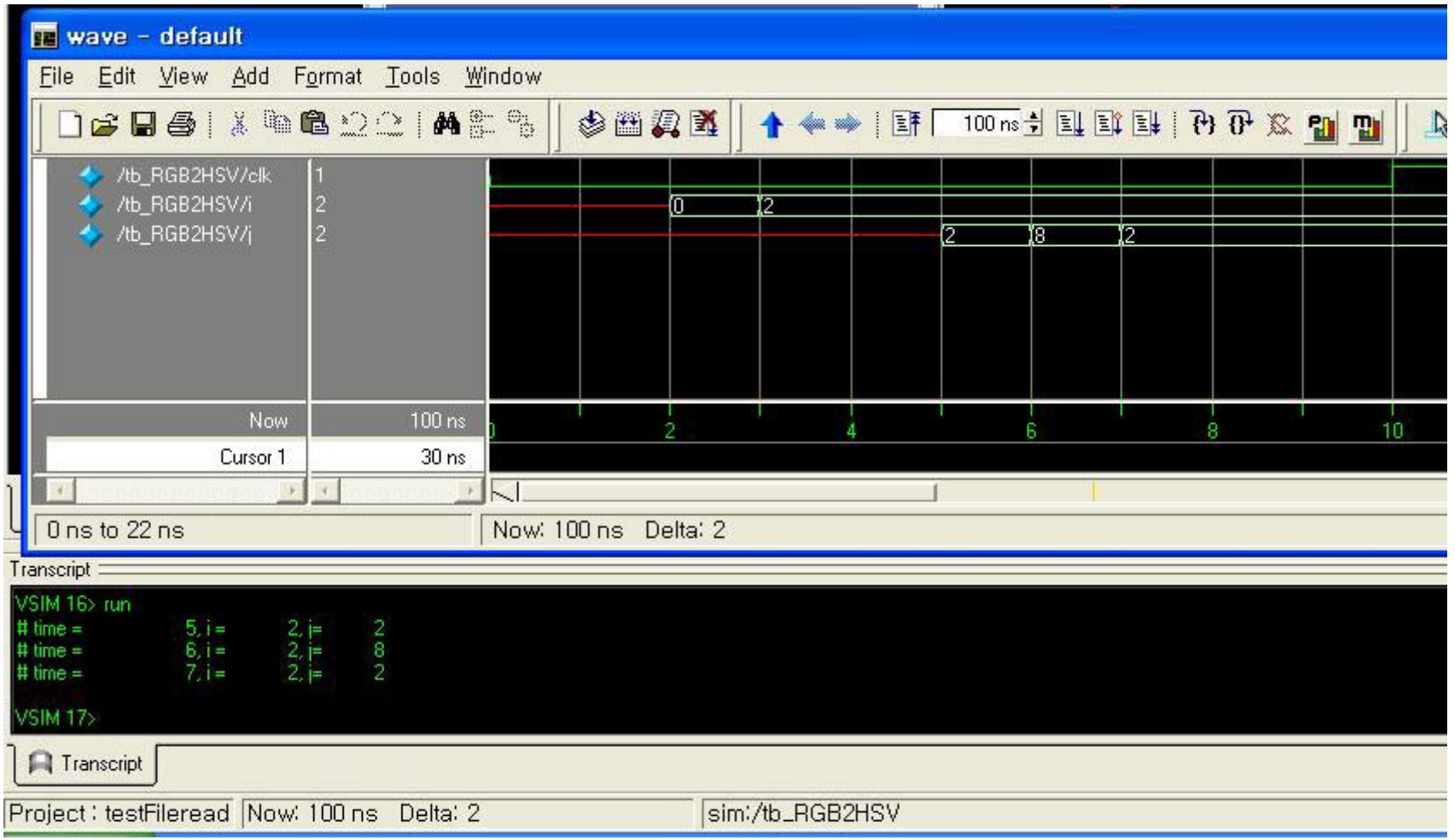
blocksTime3.v

```
module blocksTime3;
   integer i, j;

   initial
      begin
      #2 i = 0;
      #5 j = i;
      $display( "time = %d, i = %d, j = %d", $time, i, j );
      end

   initial
      begin
      #3 i = 2;
      #2 j = i;
      $display( "time = %d, i = %d, j = %d", $time, i, j );
      #1 j = 8;
      $display( "time = %d, i = %d, j = %d", $time, i, j );
      end

   initial
      #10 $finish;
endmodule
```

Important Verilog is a **discrete event simulator**: events are executed in a time-ordered queue.

Multiple initial blocks. **Predict output before you run!**

blocksTime3.v – Simulation Result



blocksTime4.v

```
module blocksTime4;
   integer i, j;

   initial
      begin
      i = 0;
      j = 3;
      end

   initial
      #10 $finish;

   always
      begin
      #1
      i = i + 1;
      j = j + 1;
      $display( "i = %d, j = %d", i, j );
      end
endmodule
```

Always block is an infinite loop. Following are same:

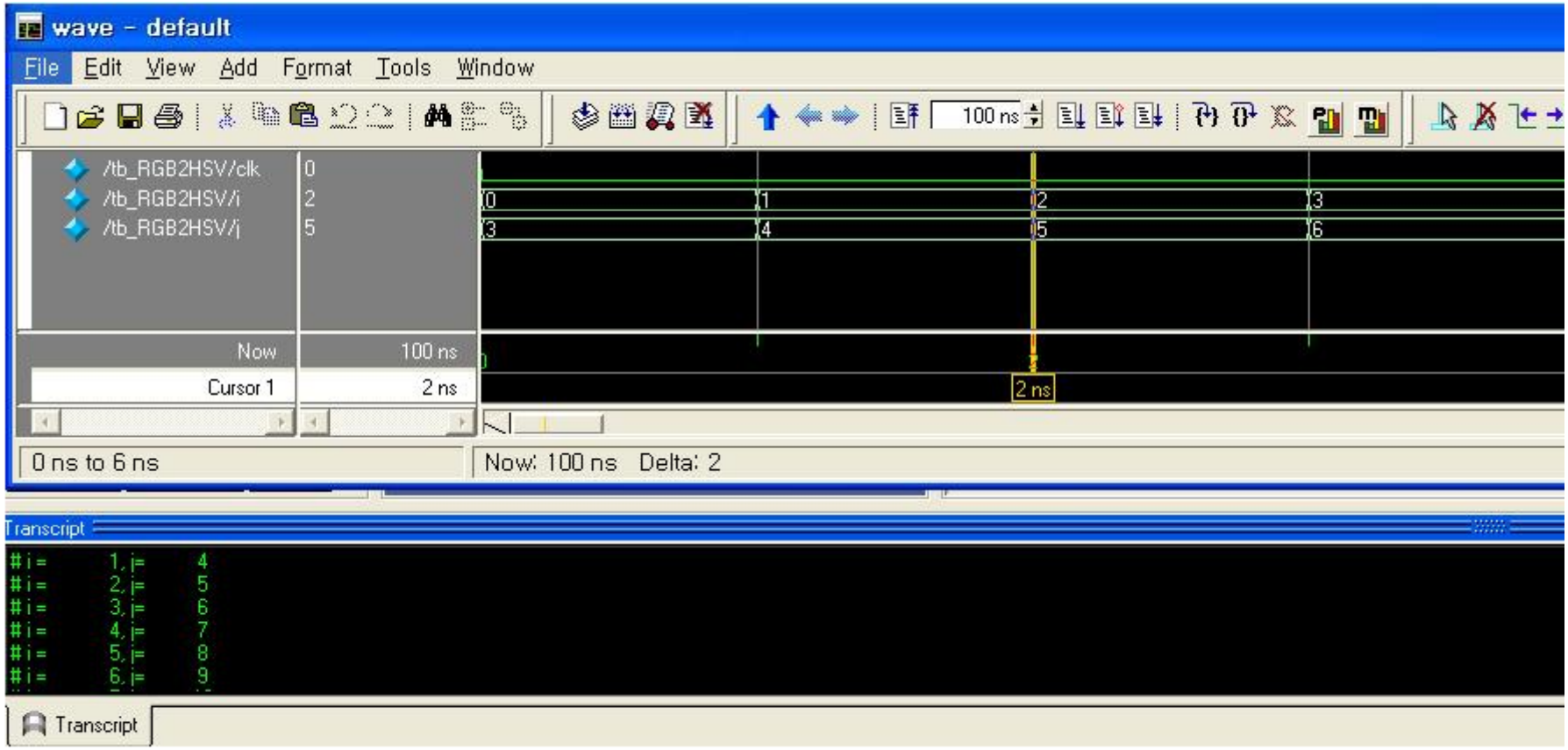
```
always
  begin
  …
  end
```

```
initial
  begin
   while(1)
    begin
      …
    end
  end
```

```
initial
  begin
   forever
     begin
       …
     end
  end
```

Comment out this delay. Run. Explain the problem!

blocksTime4.v



clockGenerator.v

```
module clockGenerator(clk);
   output clk;
   reg clk;

   initial
      begin
      clk = 0;
      end

   always
      #5 clk = ~clk;
endmodule
```

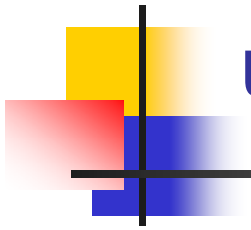
Port list. Ports can be of three types: *input*, *output*, *inout*. Each must be declared.

Internal register.

Register reg data type can have one of four values: 0, 1, x, z. Registers *store* a value till the next assignment. Registers are assigned values in procedural blocks.

If this module is run stand-alone make sure to add a $finish statement here or simulation will never complete!

The delay is half the clock period.



useClock.v

Compile with the clockGenerator.v module.

```
module useClock(clk);
   input clk;
   clockGenerator cg(clk);

   initial
      #50 $finish;

   always @(posedge cg.clk) // Bug!! Should work with "clk" only instead of
                            // "cg.clk“ - Version 9 of Verilogger Pro fixes the bug
      $display("Time = %d, Clock up!", $time);

   always @(negedge cg.clk) //
      $display("Time = %d, Clock down!", $time);
endmodule
```

systemCalls.v

```
module systemCalls(clk);
   input clk;
   clockGenerator cg(clk);

   initial
      begin
      #25 $stop;
      #50 $finish;
      end

   initial
      begin
      $write("$write does not ");
      $write("add a new line₩n");

      $display("$display does");
      $display("add a new line");

      $monitor("Clock = %d", cg.clk); end
endmodule
```

Compile with the clockGenerator.v module.

Suspends simulation – enters interactive mode.

Terminates simulation.

Similar output calls except $display adds a new line.

$monitor produces output each time a variable changes value.

blocksTime5.v

```
// Ordering processes without advancing time
module blockTime5;
   integer i, j;

   initial
      #0
      $display( "time = %d, i = %d, j = %d", $time, i, j );

   initial
      begin
      i = 0;
      j = 5;
      end

   initial
      #10 $finish;
endmodule
```

#0 delay causes the statement to execute after other processes scheduled at that time instant have completed. $time does not advance till after the statement completes.

Comment out the delay. Run. Explain what happens!

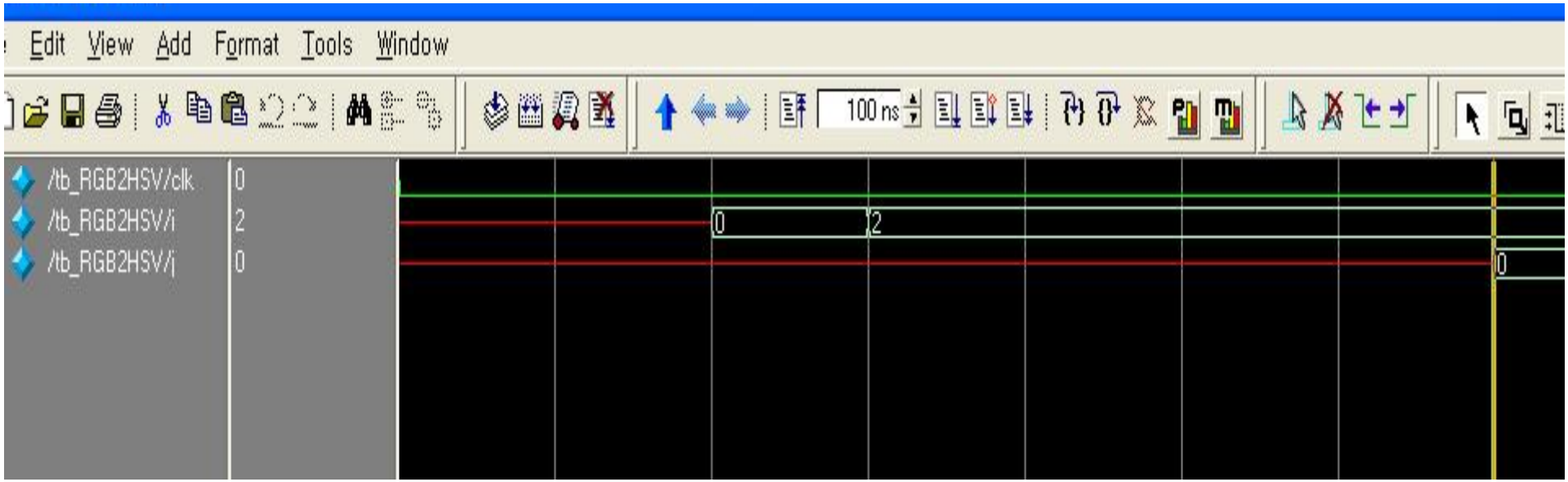
blocksTime6.v

```
module blocksTime6;
   integer i, j;

   initial
      begin
      #2 i = 0;
      j = #5 i;
      $display( "time = %d, i = %d, j = %d", $time, i, j );
      end

   initial
      #3 i = 2;

   initial
      #10 $finish;
endmodule
```

Intra-assignment delay: RHS is computed and stored in a temporary (transparent to user) and LHS is assigned the temporary after the delay.

Compare output with blocksTime2.v.

blocksTime6.v – Simulation Result



numbers.v

```
module numbers;
   integer i, j;
   reg[3:0] x, y;

   initial
      begin
      i = 'b1101;
      $display( "decimal i = %d, binary i = %b", i, i );
      $display( "octal i = %o, hex i = %h", i, i );

      j = -1;
      $display( "decimal j = %d, binary j = %b", j, j );
      $display( "octal j = %o, hex j = %h", j, j );

      x = 4'b1011;
      $display( "decimal x = %d, binary x = %b", x, x );
      $display( "octal x = %o, hex x = %h", x, x );

      y = 4'd7;
      $display( "decimal y = %d, binary y = %b", y, y );
      $display( "octal y = %o, hex y = %h", y, y );

      $finish;
      end
endmodule
```

Register array.

'<base>: base can be d, b, o, h

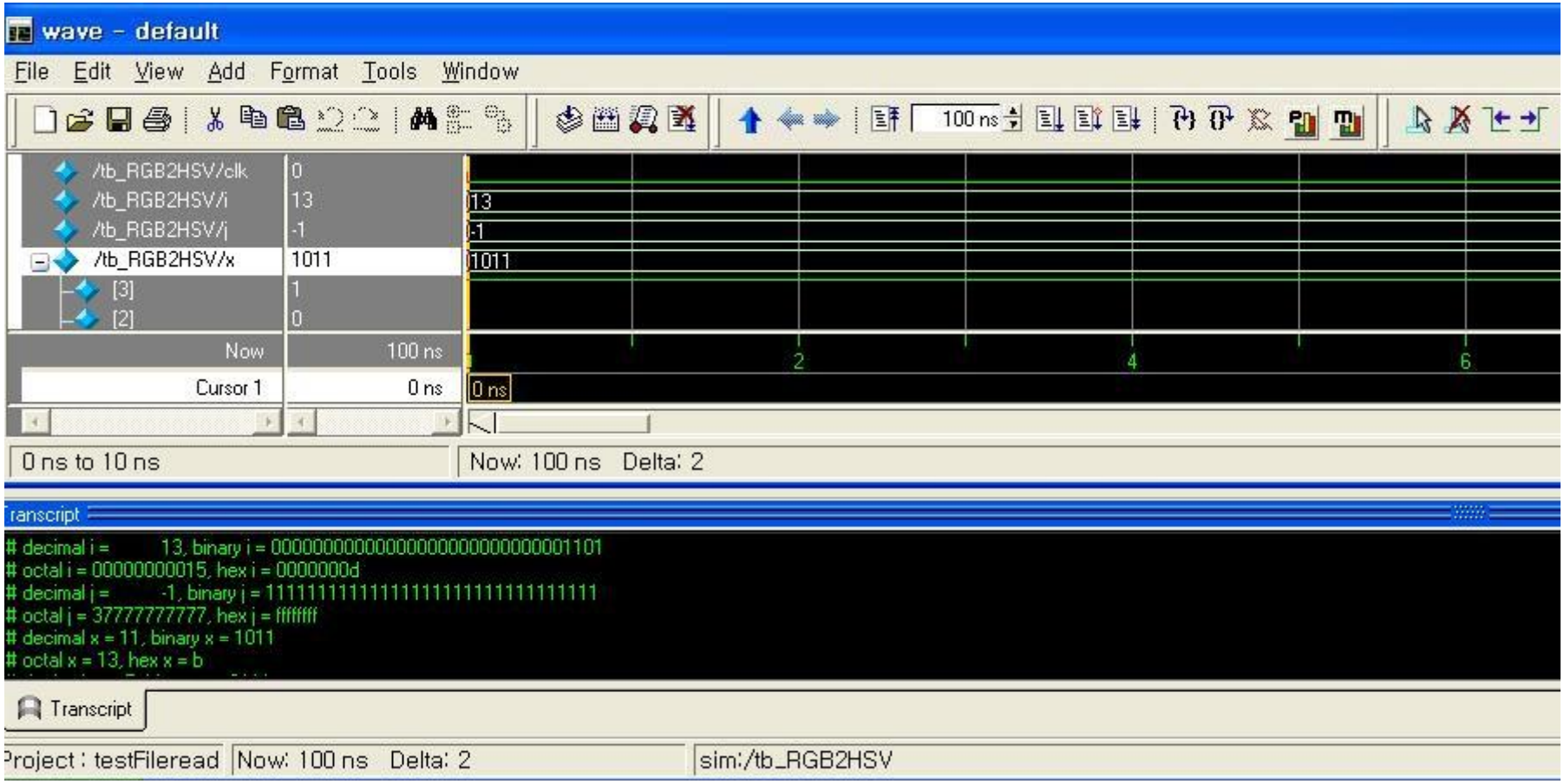
Default base: d

Typical format: <size>'<base><number>
size is a *decimal* value that specifies the size of the number in *bits*.

Array of register arrays simulate memory. Example memory declaration with 1K 32-bit words:
reg[31:0] smallMem[0:1023];

Negative numbers are stored in two's complement form.

numbers.v – Simulation Result



simpleBehavioral.v

Sensitivity trigger: when any of a, b or c changes. Replace this statement with “initial”. Output?!

```
module aOrNotbOrc(d, a, b, c);
   output d;
   input a, b, c;
   reg d, p;

   always @(a or b or c)
      begin
      p = a || ~b;
      d = p || c;
      end
endmodule
```

Modules are of three types: *behavioral*, *dataflow*, *gate-level*. Behavioral modules contain code in procedural blocks.

Statements in a procedural block *cannot be re-ordered* without affecting the program as these statements are executed sequentially, exactly like in a conventional programming language such as C.

Ports are of three types: *input*, *output*, *inout*. Each must be declared. Each port also has a data type: either *reg* or *wire* (*net*). Default is wire. Inputs and inouts are always wire. Output ports that hold their value are reg, otherwise wire. More later…

One port register, one internal register.

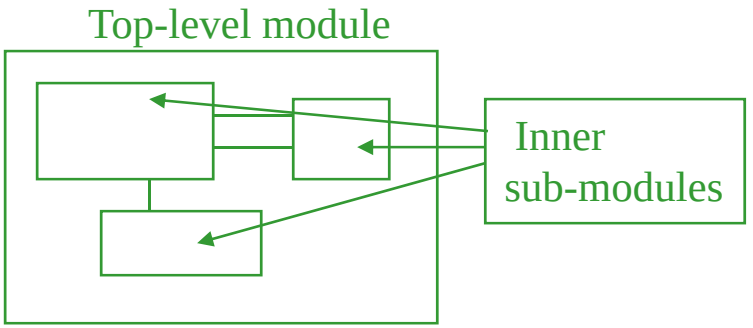
Wires are part of the more general class of nets. However, the only nets we shall design with are wires.

simpleBehavioral.v (cont.)
Top-level stimulus module

```
module stimulus;
   integer i, j, k;
   reg a, b, c;
   aOrNotbOrc X(d, a, b, c);

   initial
      begin
      for ( i=0; i<=1; i=i+1 )
         for ( j=0; j<=1; j=j+1 )
            for ( k=0; k<=1; k=k+1 )
             begin
             a = i;
             b = j;
             c = k;
             #1 $display("a = %d b = %d, c = %d, d = %d", a, b, c, d)
             end
      $finish;
      end
endmodule
```

Instantiation.

Verilog Good Design Principle There is one top-level module, typically called system or stimulus, which is *uninstantiated* and has no ports. This module contains *instantiations* of lower-level (inner) sub-modules. Typical picture below.



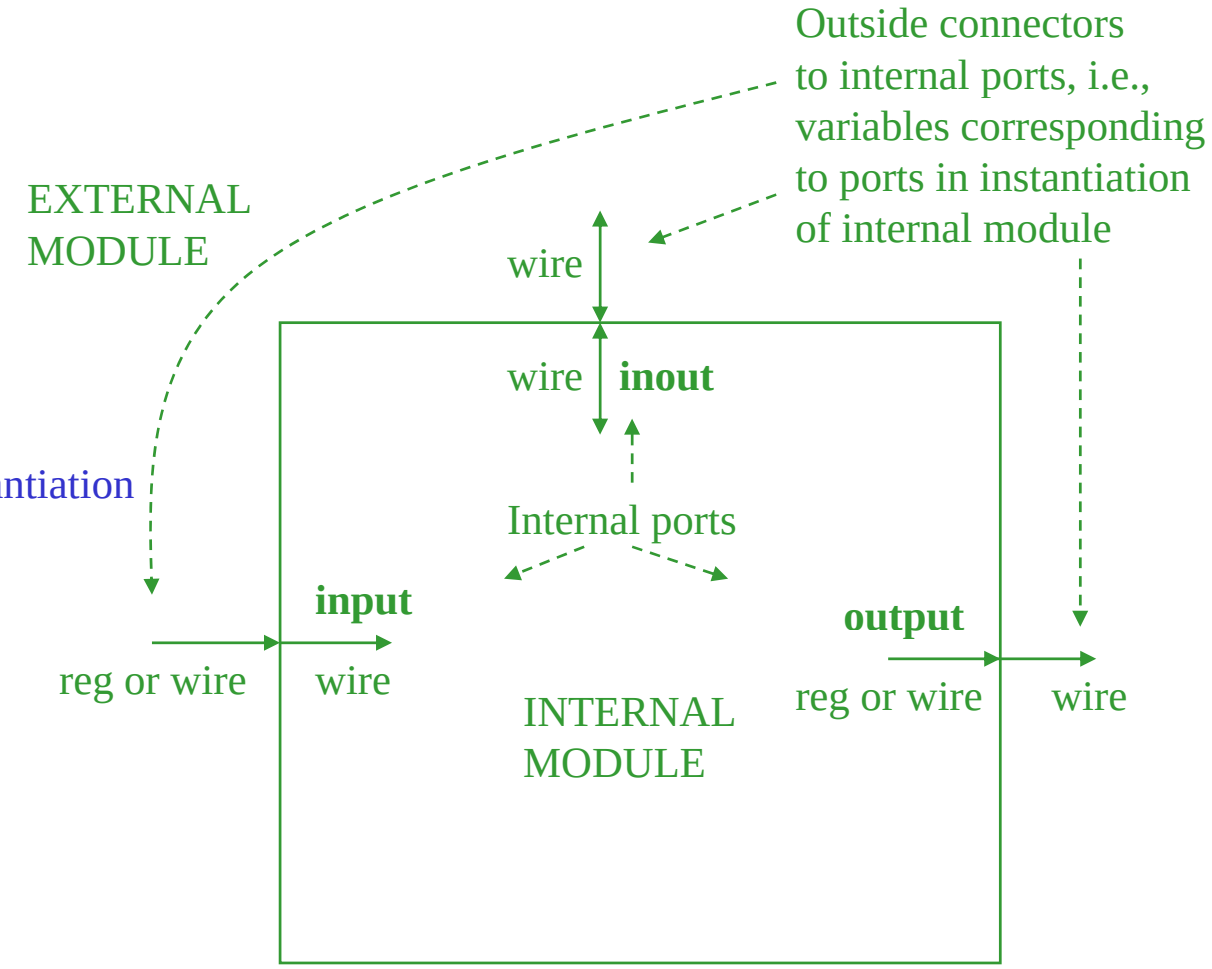
Remove the #1 delay. Run. Explain!

Port Rules Diagram

Example:

```
module external
reg a;
wire b;
internal in(a, b); //instantiation
…
endmodule

module internal(x, y)
input x;
output y;
wire x;
reg y;
…
endmodule
```

port-connector



General rule (with few exceptions) Ports in all modules except for the stimulus module should be wire. Stimulus module has registers to set data for internal modules and wire ports only to read data from internal modules.

simpleDataflow.v

```
module aOrNotbOrc(d, a, b, c);
   output d;
   input a, b, c;
   wire p, q;

   assign q = ~b;
   assign p = a || q;
   assign d = p || c;
endmodule
```

A *dataflow* module does not contain procedures.

Statements in a dataflow module *can be re-ordered* without affecting the program as they simply describe a *set of data manipulations and movements* rather than a *sequence of actions* as in behavioral code. In this regard dataflow code is very similar to gate-level code.

Continuous assignment statements: any change in the RHS causes instantaneous update of the wire on the LHS, unless there is a programmed delay.

Use stimulus module from behavioral code.

simpleGate.v

```
module aOrNotbOrc(d, a, b, c);
   output d;
   input a, b, c;
   wire p, q;

   not(q, b);
   or(p, a, q);
   or(d, p, c);
endmodule
```

A *gate-level* module does not contain procedures.

Statements in a gate-level module *can be re-ordered* without affecting the program as they simply describe a *set of connections* rather than a *sequence of actions* as in behavioral code. A gate-levelmodule is equivalent to a *combinational circuit*.

Wire data type can have one of four values: 0, 1, x, z. Wires *cannot store* values – they are continuously *driven*.

Primitive gates. Verilog provides several such, e.g., *and*, *or*, *nand*, *nor*, *not*, *buf*, etc.

Use stimulus module from behavioral code.

4valuedLogic.v

```
module fourValues( a , b, c, d );
   output a, b, c, d ;

   assign a = 1;
   assign b = 0;
   assign c = a;
   assign c = b;
endmodule

module stimulus;
   fourValues X(a, b, c, d);

   initial
      begin
      #1 $display("a = %d b = %d, c = %d, d = %d", a, b, c, d);
      $finish;
      end
endmodule
```

Conflict or *race* condition. Remember this is not a procedural (i.e., sequential) block! These are continuous assign-ments.

4-valued logic:
0 – low
1 – high
x – unknown
z – undriven wire
Now explain output!

blockingVSnba1.v

```
module blockingVSnba1;
   integer i, j, k, l;

   initial
      begin
      #1 i = 3;
      #1 i = i + 1;
      j = i +1;
      #1 $display( "i = %d, j = %d", i, j );

      #1 i = 3;
      #1 i <= i + 1;
      j <= i + 1;
      #1 $display( "i = %d, j = %d", i, j );

      $finish;
      end
endmodule
```

Blocking (procedural) assignment: the whole statement must execute before control is released, as in traditional programming languages.

Non-blocking (procedural) assignment: *all* the RHSs for the current time instant are evaluated (and stored transparently in temporaries) first and, subsequently, the LHSs are updated at the end of the time instant.

blockingVSnba2.v

```
module blockingVSnba2(clk);
   input clk;
   clockGenerator cg(clk);
   integer i, j;

   initial
      begin
      i = 10;
      #50 $finish;
      end

   always @(posedge clk)
      i = i + 1;     // i <= i + 1;
   always @(posedge clk)
      j = i;    // j <= i;

   always @(negedge clk)
      $display("i = %d, j = %d", i, j);
endmodule
```

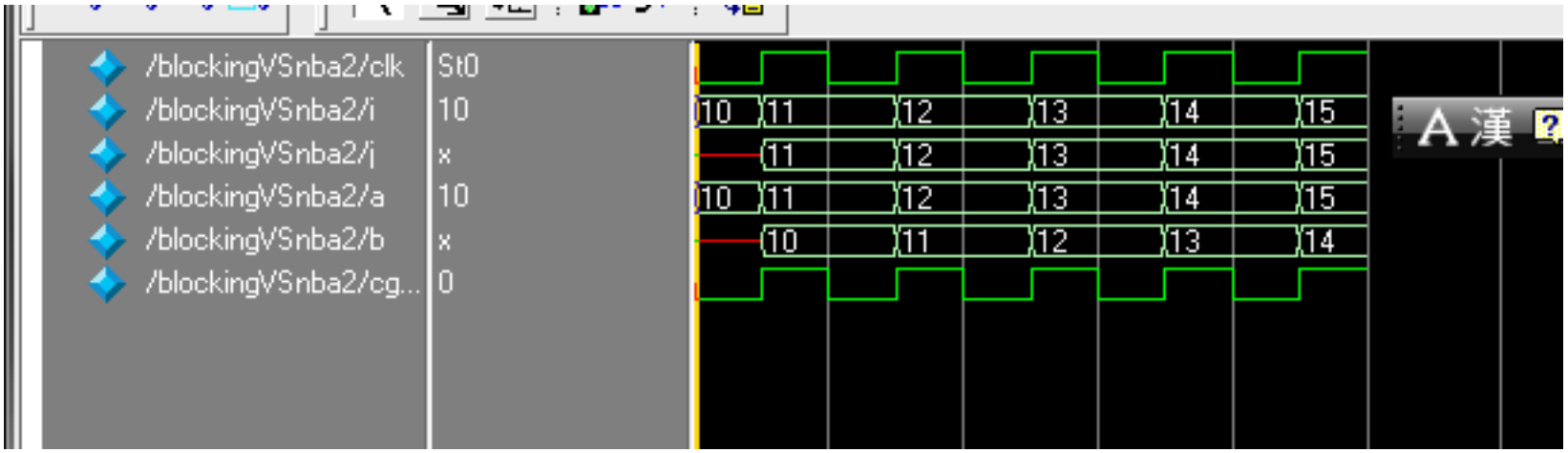
Compile with clockGenerator.v.

An application of non-blocking assignments to solve a *race* problem.

With blocking assignments we get different output depending on the order these two statements are executed by the simulator, though they are both supposed to execute “simultaneously” at posedge clk - race problem.

Race problem is solved if the non-blocking assignments (after the comments) are used instead - output is unique.

blockingVSnba2.v – Simulation Result



blockingVSnba2.v – Simulation Result

```
begin
     i = 10;
     #50 $finish;
     end

  always @(posedge clk)
     i = i + 1;
  always @(posedge clk)
     j = i;
```

```
# i =      10, j =      x
# i =      11, j =      11
# i =      12, j =      12
# i =      13, j =      13
# i =      14, j =      14
```

```
begin
     a = 10;
     #50 $finish;
     end

  always @(posedge clk)
     a <= a+ 1;
  always @(posedge clk)
     b <= a;
```

```
# a =      10, b =      x
# a =      11, b =      10
# a =      12, b =      11
# a =      13, b =      12
# a =      14, b =      13
```

blockingVSnba3.v

```
module blockingVSnba3;
   reg[7:0] dataBuf, dataCache, instrBuf, instrCache;

   initial
      begin
      dataCache = 8'b11010011;
      instrCache = 8'b10010010;

      #20;
      $display("Time = %d, dataBuf = %b, instrBuf = %b", $time, dataBuf, instrBuf);
      dataBuf <= #1 dataCache;
      instrBuf <= #1 instrCache;
      #1 $display("Time = %d, dataBuf = %b, instrBuf = %b", $time, dataBuf, instrBuf);

      $finish;
      end
endmodule
```

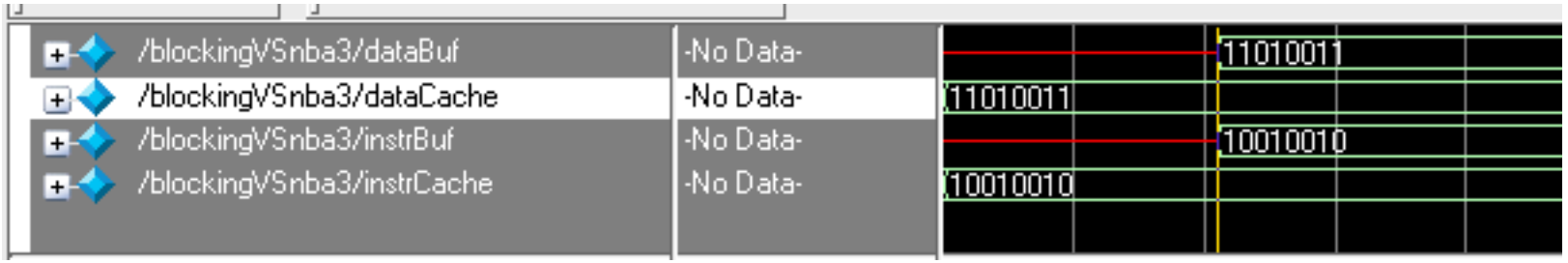
The most important application of non-blocking assignments is to model concurrency in hardware systems at the behavioral level.

Both loads from *dataCache* to *dataBuf* and *instrCache* to *instrBuf* happen concurrently in the 20-21 clock cycle.

Replace non-blocking with blocking assignments and observe.

blockingVSnba3.v – Simulation Result

nonBlock



Block