ECE 4514
Digital Design II
Spring 2007

Lecture 3: Verilog Bread and Butter

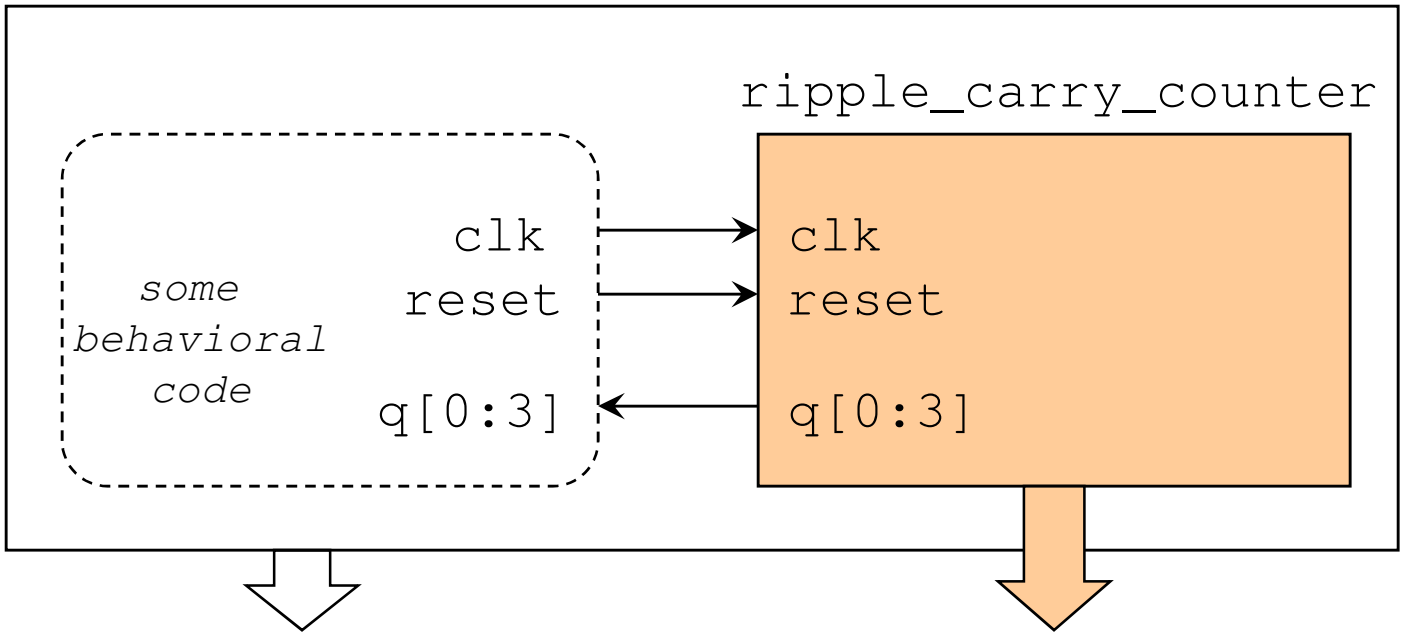
Patrick Schaumont

Verilog

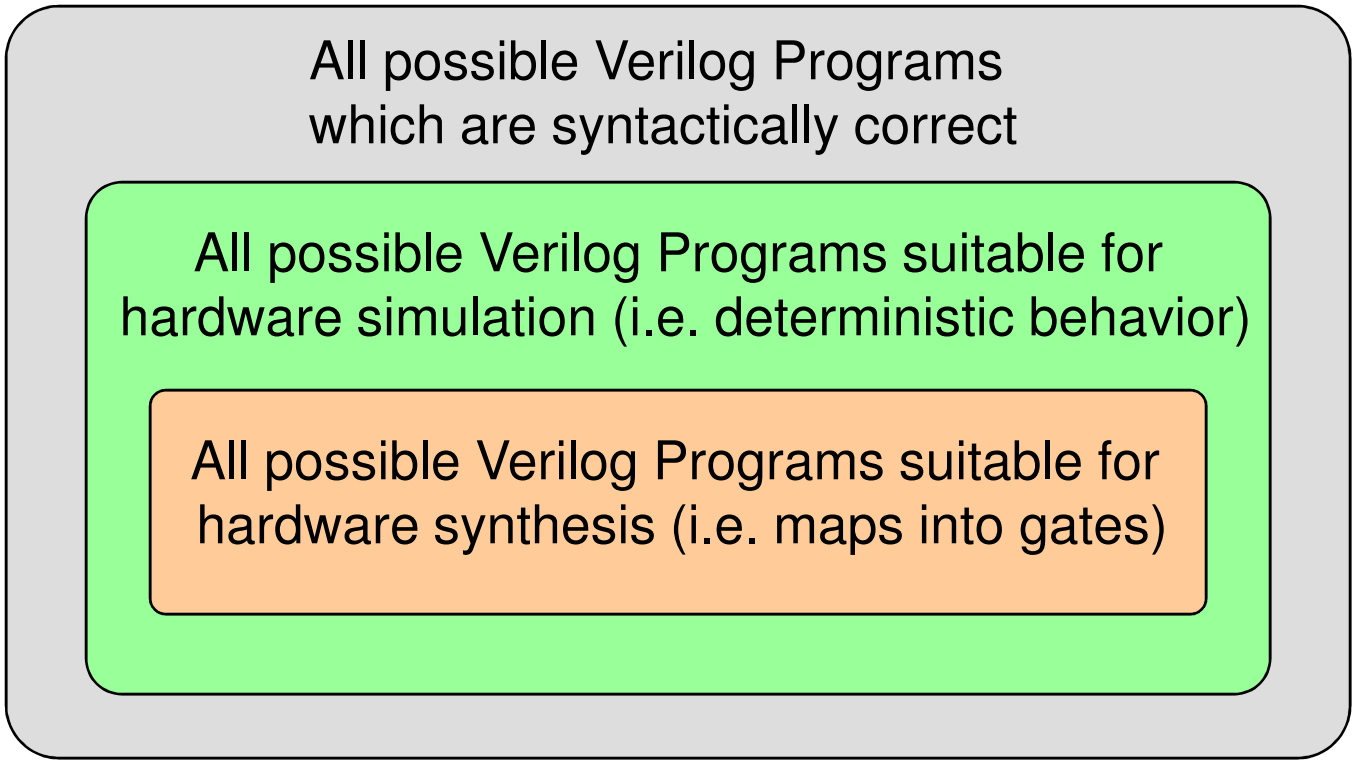
- Difference between synthesis and simulation
- Modules, module declarations and instantiation
- Constants
  - Numbers
- Data types
  - Value Levels
  - Regs
  - Vectors
  - Arrays

Synthesis and Simulation

- To simulate a 4-bit counter module we used a testbench.



Synthesis and Simulation



Synthesis and Simulation

```
module syntst;
reg b;

initial
begin
  b = 0;
end

initial
begin
  b = 1;
end

endmodule;
```

This will simulate, but it's non-deterministic.

meaning: if you use this code on a different Verilog simulator, the simulation may be different

Synthesis and Simulation

```
module syntst(clk);
init clk;
reg b;

initial
begin
    b = 0;
#10 b = 1;
end

endmodule;
```

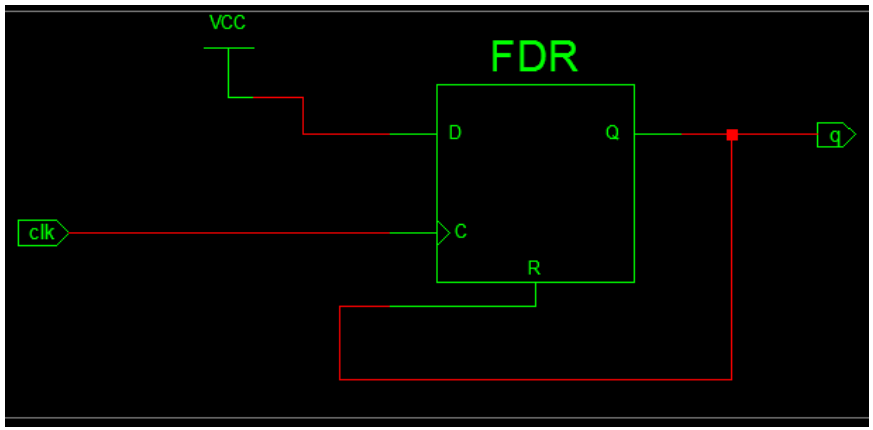
This will simulate, but it's not synthesizable.

```
Analyzing top module <syntst>.
WARNING:Xst:916 - "syntst.v" line 12: Delay is ignored for synthesis.
Module <syntst> is correct for synthesis.
 Synthesizing Unit <syntst>.
    Related source file is "syntst.v".
WARNING:Xst:647 - Input <clk> is never used.
WARNING:Xst:653 - Signal <b> is used but never assigned. Tied to value 1.
Unit <syntst> synthesized.
```

Synthesis and Simulation

```
module syntst(clk, q);
   input clk;
   output q;
   reg b;

always @(posedge clk)
begin
   b = ~b;
end

assign q = b;

endmodule
```

This will synthesize



Synthesis and Simulation

- For now, we are focusing on simulation and writing correct Verilog code
- Keep in mind: A correct simulation does not mean a correct *implementation ...*

Modules

```
module name(portlist);
    port declarations;
    parameter declarations;

    wire declarations;
    reg declarations;
    variable declarations;

    module instantations;
    dataflow statements;
    always blocks;
    initial blocks;

    tasks and functions;

endmodule
```

Modules

```
module name(portlist);
  port declarations;
  parameter declarations;

  wire declarations;
  reg declarations;
  variable declarations;

  module instantations;
  dataflow statements;
  always blocks;
  initial blocks;

  tasks and functions;

endmodule
```

- Direction of ports
- Module variants
  (module muffin;
  parameter banana_nut;)

Modules

```
module name(portlist);
    port declarations;
    parameter declarations;

    wire declarations;
    reg declarations;
    variable declarations;

    module instantations;
    dataflow statements;
    always blocks;
    initial blocks;

    tasks and functions;

endmodule
```

- local communication
- local storage
- local storage (testbench)

within the confines
of this module and this
level of the design hierarchy

Modules

```
module name(portlist);
    port declarations;
    parameter declarations;

    wire declarations;
    reg declarations;
    variable declarations;

    module instantations;          - structural
    dataflow statements;           - dataflow
    always blocks;                 - behavioral
    initial blocks;                - behavioral

    tasks and functions;           - behavioral

endmodule
```

We will next talk about

```
module name(portlist);
    port declarations;
    parameter declarations;

    wire declarations;
    reg declarations;
    variable declarations;

    module instantations;
    dataflow statements;
    always blocks;
    initial blocks;

    tasks and functions;

endmodule
```

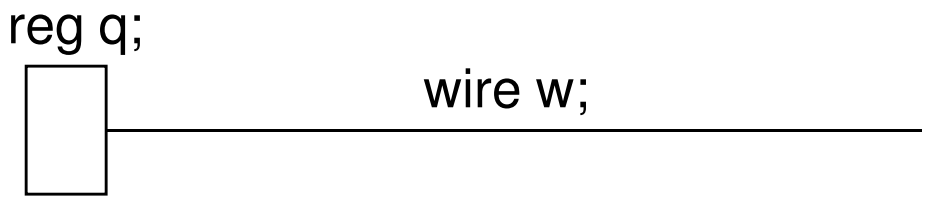
Port Lists
Port Connections

Wire and Reg Declarations
Constants

Other variables

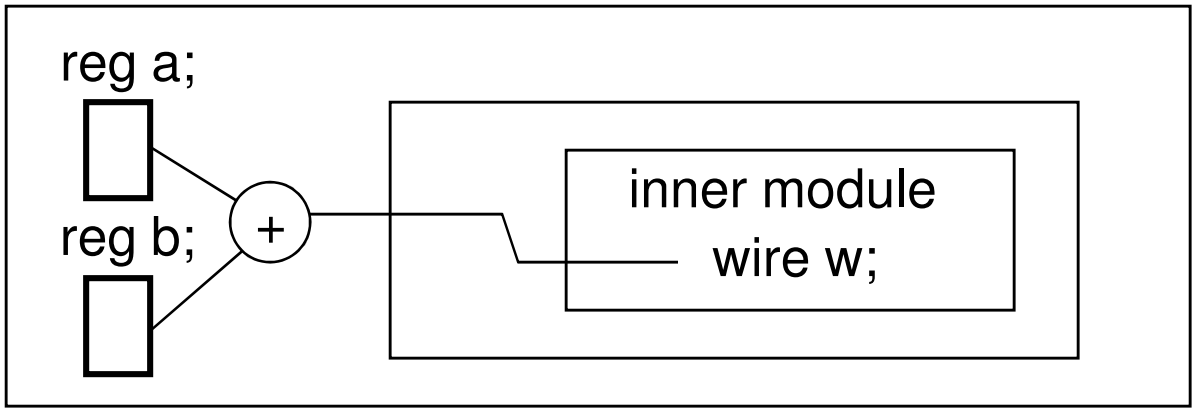
always and initial blocks

Wires and Regs



- A wire and a reg carry information around in a module
- A reg is a variable with storage
  - Assign value v at time t_1, then the value will remain v for $t > t_1$ until reg is re-assigned
- A wire does not have storage, but it reflects the value of a driver
  - Assign value v at time t_1, then the value is not valid for any other time then $t = t_1$

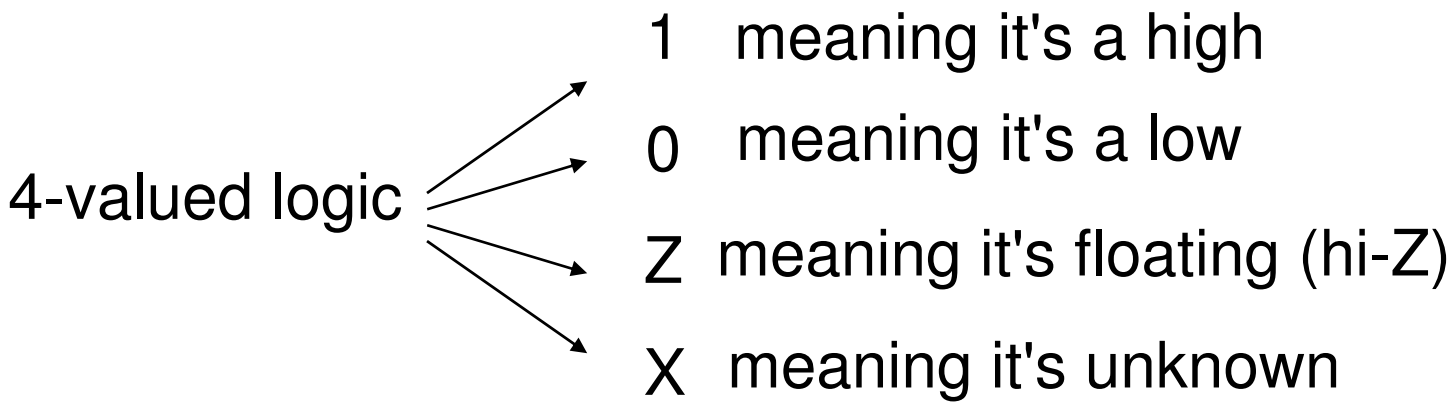
Wires and Regs



- A driver of a wire can be defined through expressions and multiple levels of hierarchy
- Eventually, all wires need a driver, otherwise their value is undefined

Value Levels

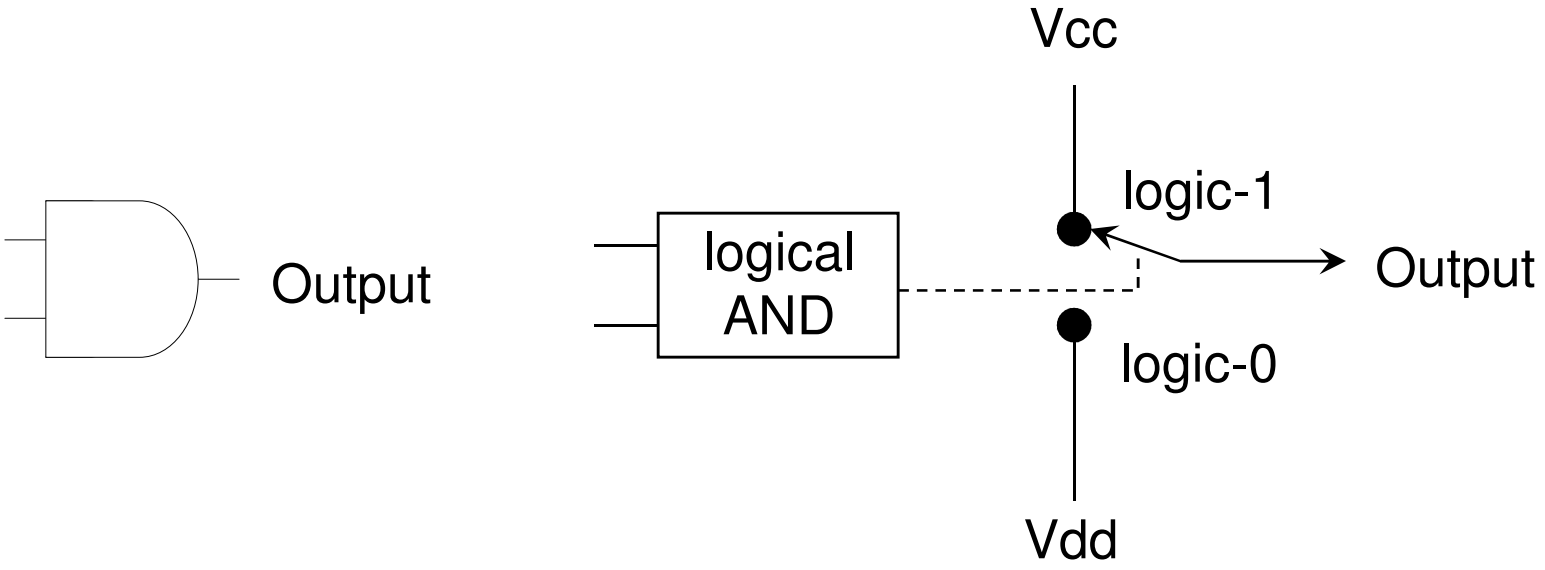
- Each wire can be at 1 of 4 logic levels



- Initially, variables start out at X
- X is contagious. E.g. logic operation (1 or X) = X
- Z is contagious. E.g. logic operation (1 or Z) = X

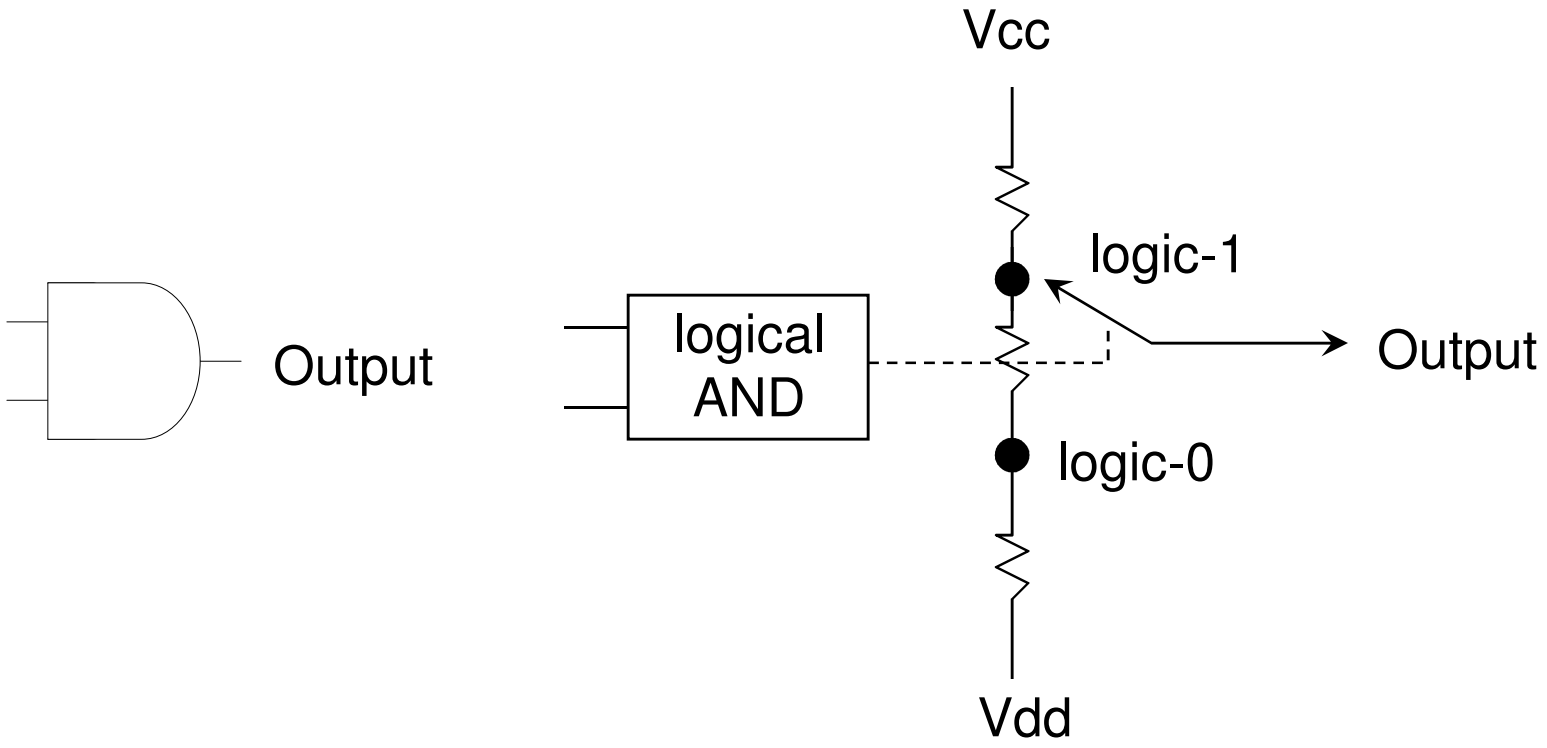
Driving Levels

- Ideally, a driver is a switch



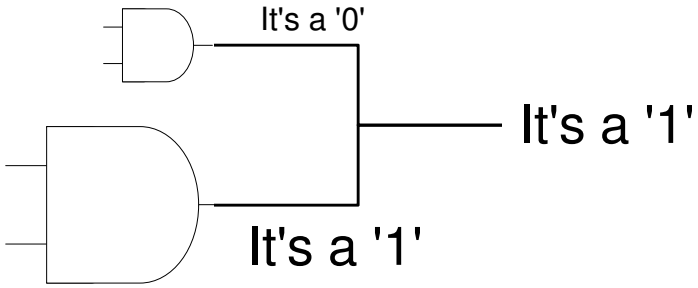
Driving Levels

- In reality, a driver has different strengths



Driving Levels

- A driving level enables to simulate the electrical effect of different driving strengths
  - Can simulate the effect of signal contention
  - Can implement wired-AND and wired-OR
  - Can simulate tri-state buses
  - Can simulate large gates and small gates



- Verilog offers support for this type of simulation
  - We will stick to wire, reg and a single driving level for now

Vectors

```
wire [7:0] b,c; // two 8-bit vectors
                // msb is b[7] and c[7]
reg [0:20] a;   // 21-bit reg, msb is a[0]

b[7]    // bit 7 of the b bit-vector
c[6:5]  // bit 6 and 5 of the c bit-vector
a[3+:4] // bit 3, 4, 5, 6 of a
```

Constants

```
<size> <base>' <number>
```

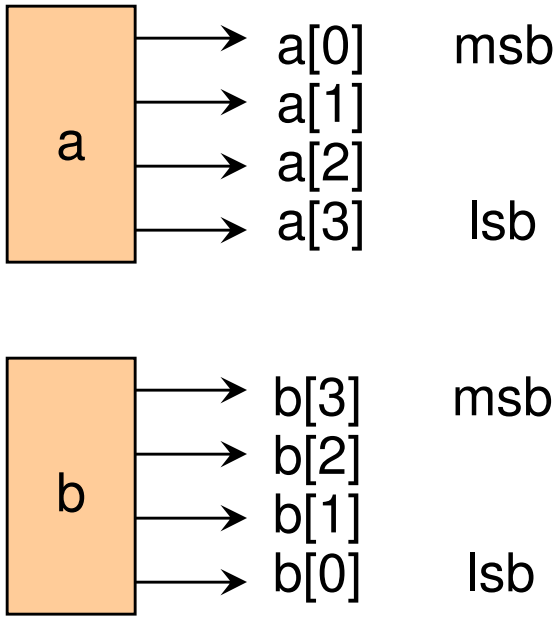
number of *bits*	b or B Binary	number in selected base
	o or O Octal	may use _ as a spacer
	h or H Hex	0-extended with 0 or 1 msb
	d or D Decimal	x-extended with x msb
		z-extended with z msb

For example ...

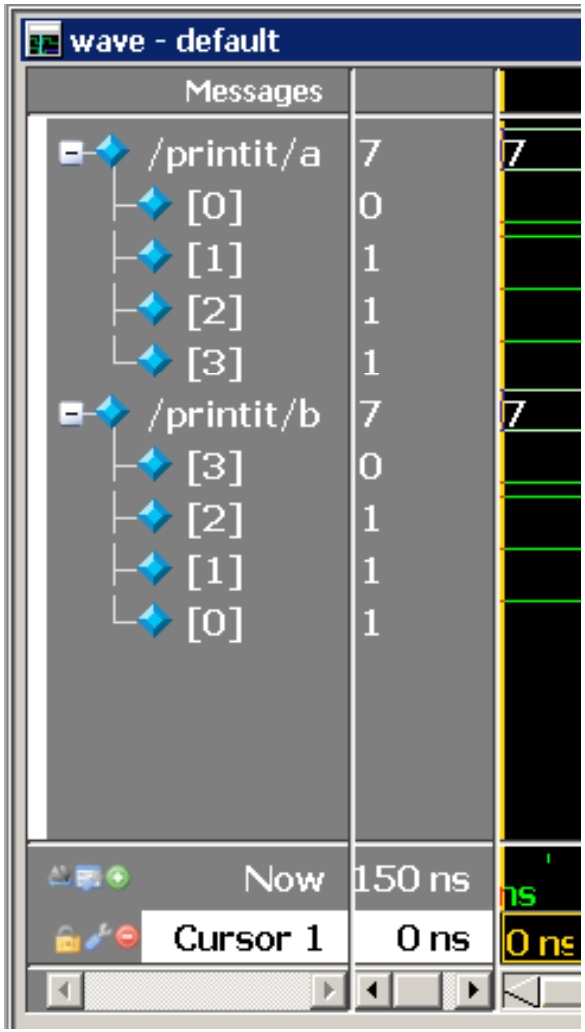
```
<size> <base>' <number>

6'b010_111    gives 010111
8'b0110       gives 00000110
8'b1110       gives 00001110
4'bx01        gives xx01
16'H3AB       gives 0000001110101011
24            gives 0…0011000
5'O36         gives 11100
16'Hx         gives xxxxxxxxxxxxxxxx
8'hz          gives zzzzzzzz
```

Let's go through an example

```
module printit;

    reg [0:3] a;
    reg [3:0] b;

    initial
    begin

    // some statements here ..

    end
endmodule
```

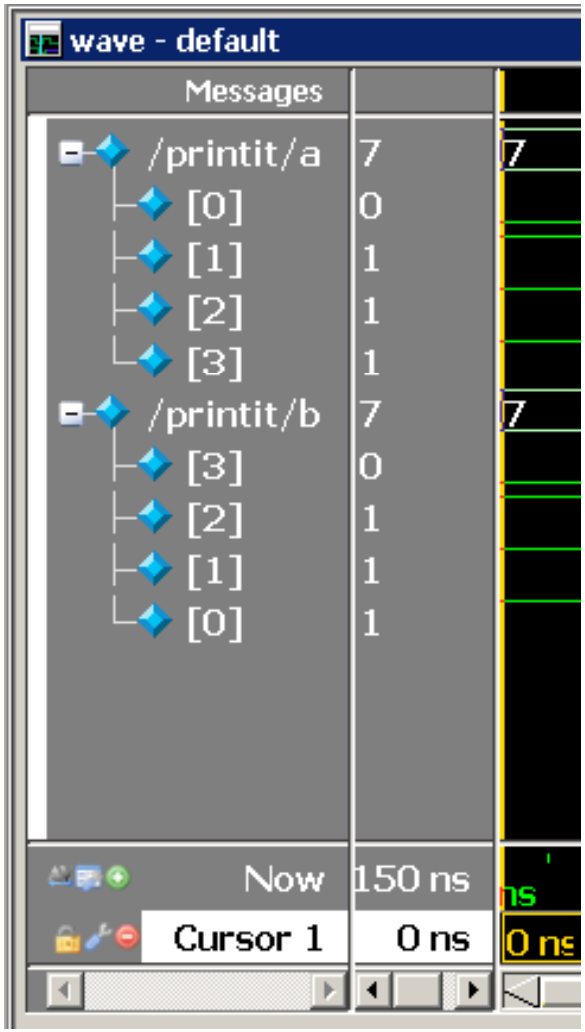


We initialize as follows:




```
module printit;

    reg [0:3] a;
    reg [3:0] b;

    initial
    begin

    a = 7;
    b = 7;

    end
endmodule
```

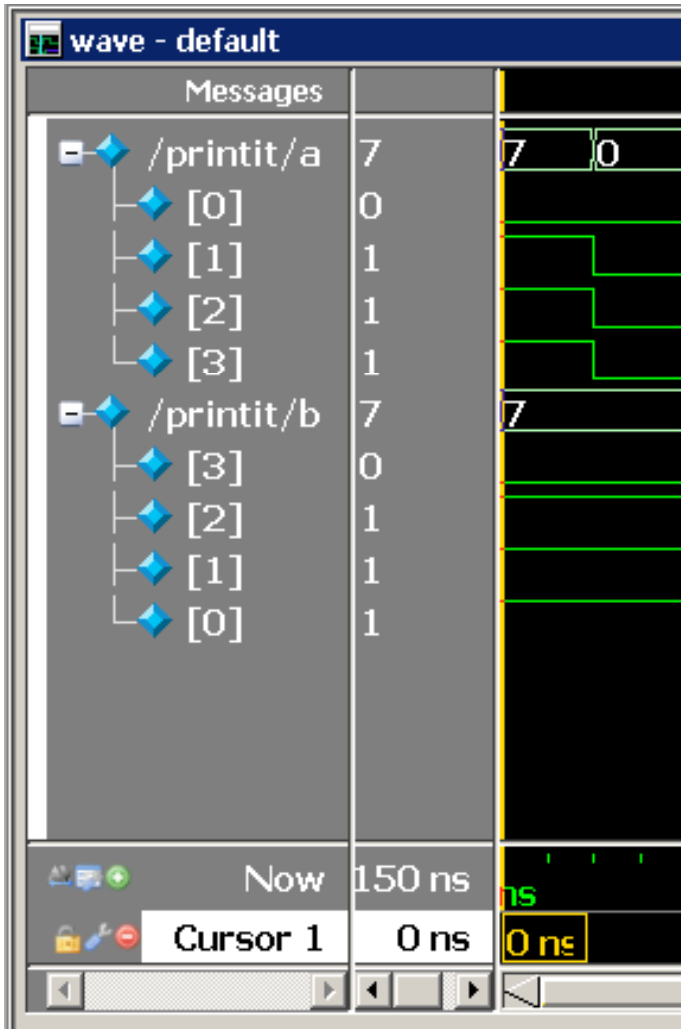
What would happen after:




```
module printit;

    reg [0:3] a;
    reg [3:0] b;

    initial
    begin

    a = 7;
    b = 7;
    #10 a = 16;

    end
endmodule
```

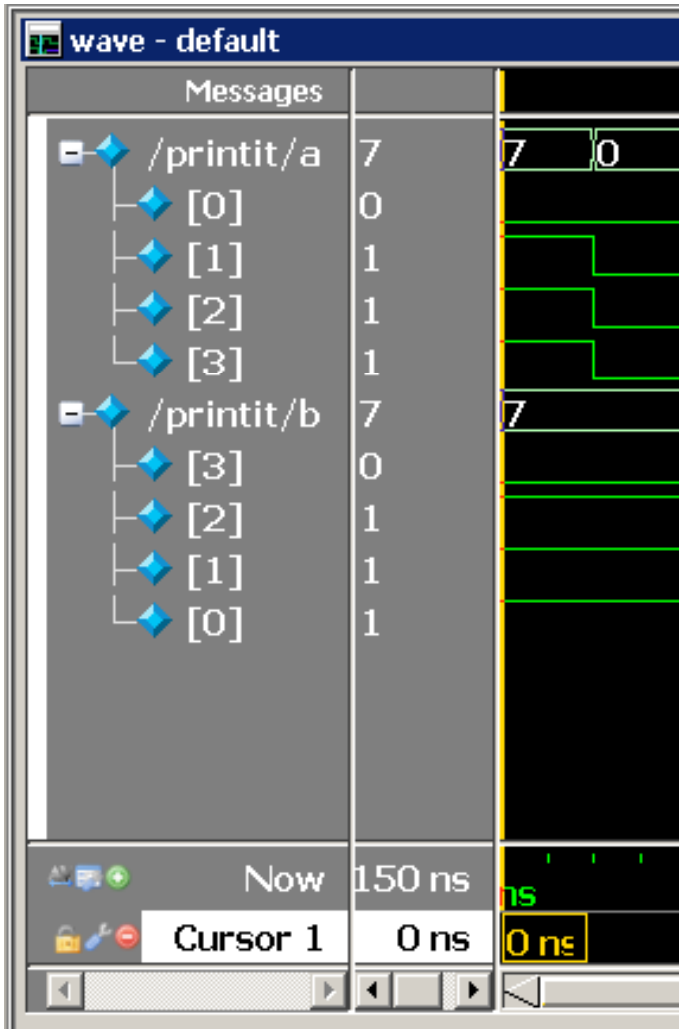
What would happen after:

a = 0




```
module printit;

    reg [0:3] a;
    reg [3:0] b;

    initial
    begin

    a = 7;
    b = 7;
    #10 a = 16;

    end
endmodule
```

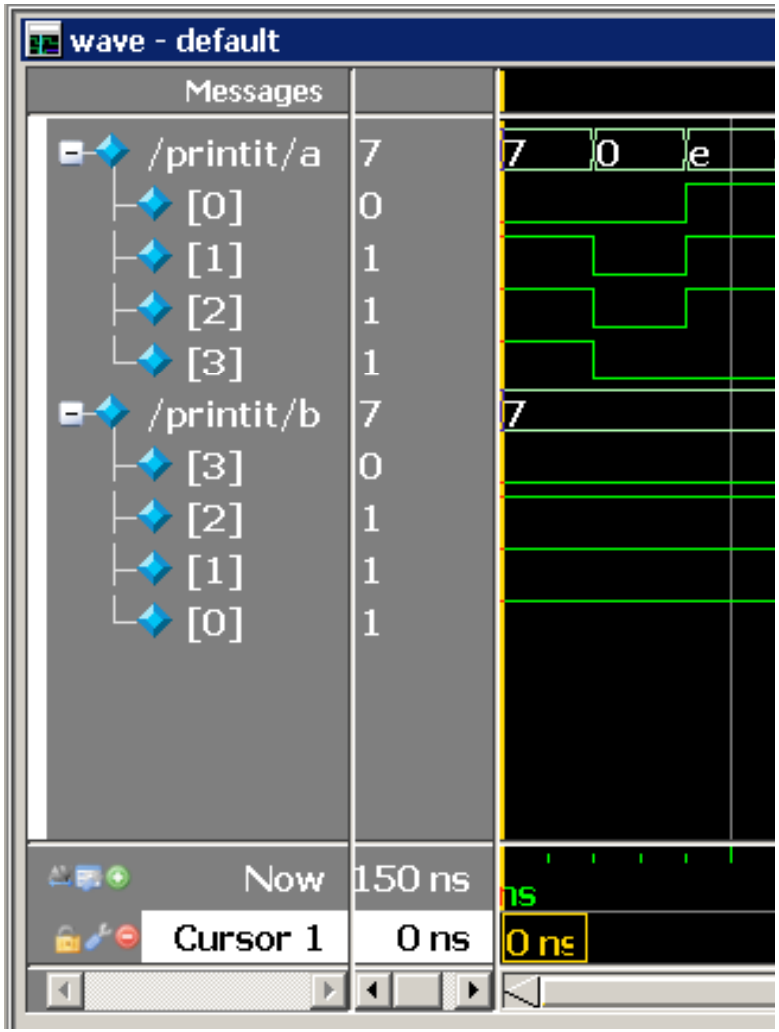
What would happen after:




```
module printit;

    reg [0:3] a;
    reg [3:0] b;

    initial
    begin

    a = 7;
    b = 7;
    #10 a = 16;
    #10 a = 4'b1110;

    end
endmodule
```

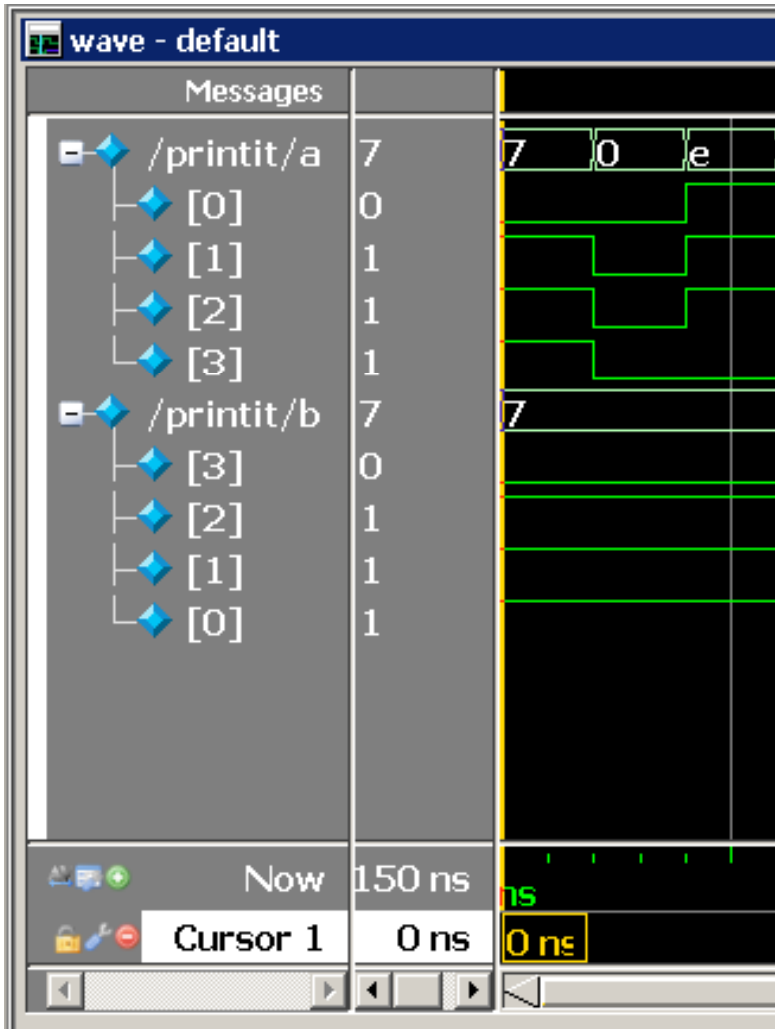
What would happen after:

a = e




```
module printit;

    reg [0:3] a;
    reg [3:0] b;

    initial
    begin

    a = 7;
    b = 7;
    #10 a = 16;
    #10 a = 4'b1110;

    end
endmodule
```

What would happen after:

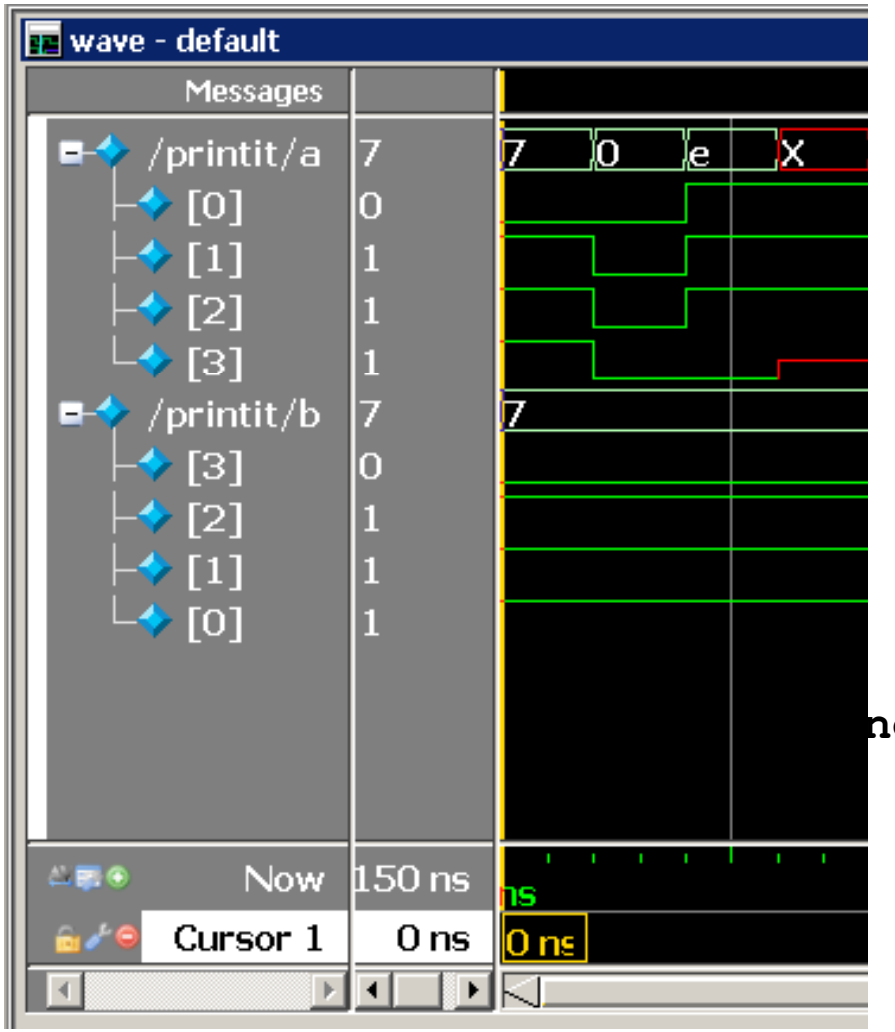



```
module printit;

    reg [0:3] a;
    reg [3:0] b;

    initial
    begin

    a = 7;
    b = 7;
    #10 a = 16;
    #10 a = 4'b1110;
    #10 a = 4'b111x;

    end
endmodule
```

What would happen after:

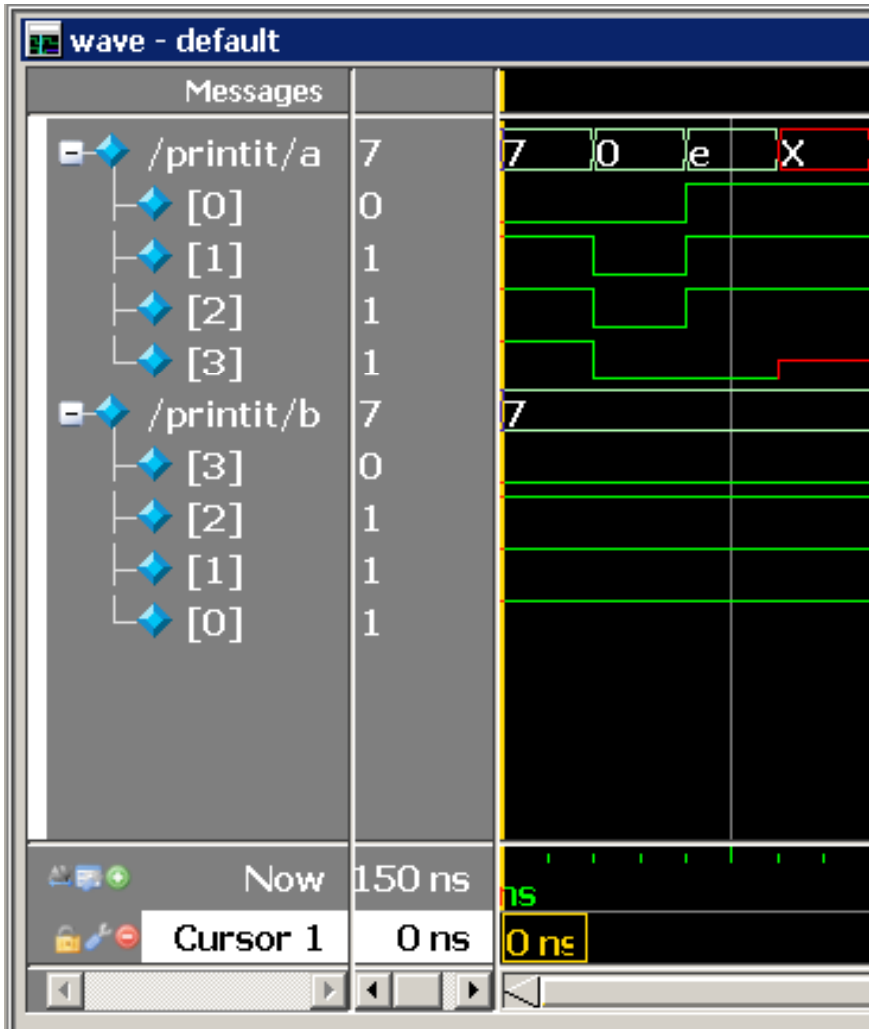
```
module printit;

  reg [0:3] a;
  reg [3:0] b;

  initial
  begin

  a = 7;
  b = 7;
  #10 a = 16;
  #10 a = 4'b1110;
  #10 a = 4'b111x;

  end
endmodule
```

a = X



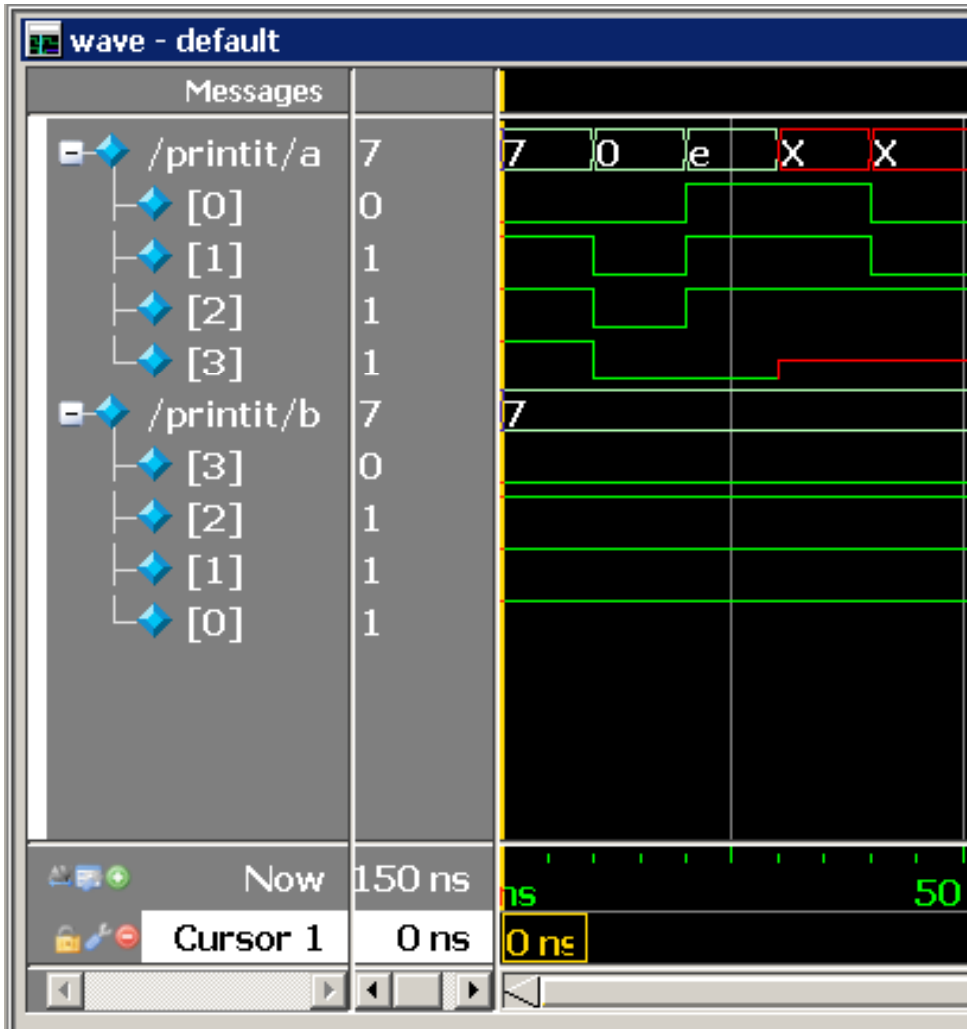
What would happen after:

```
#10 a = 4'b1x;
```



What would happen after:

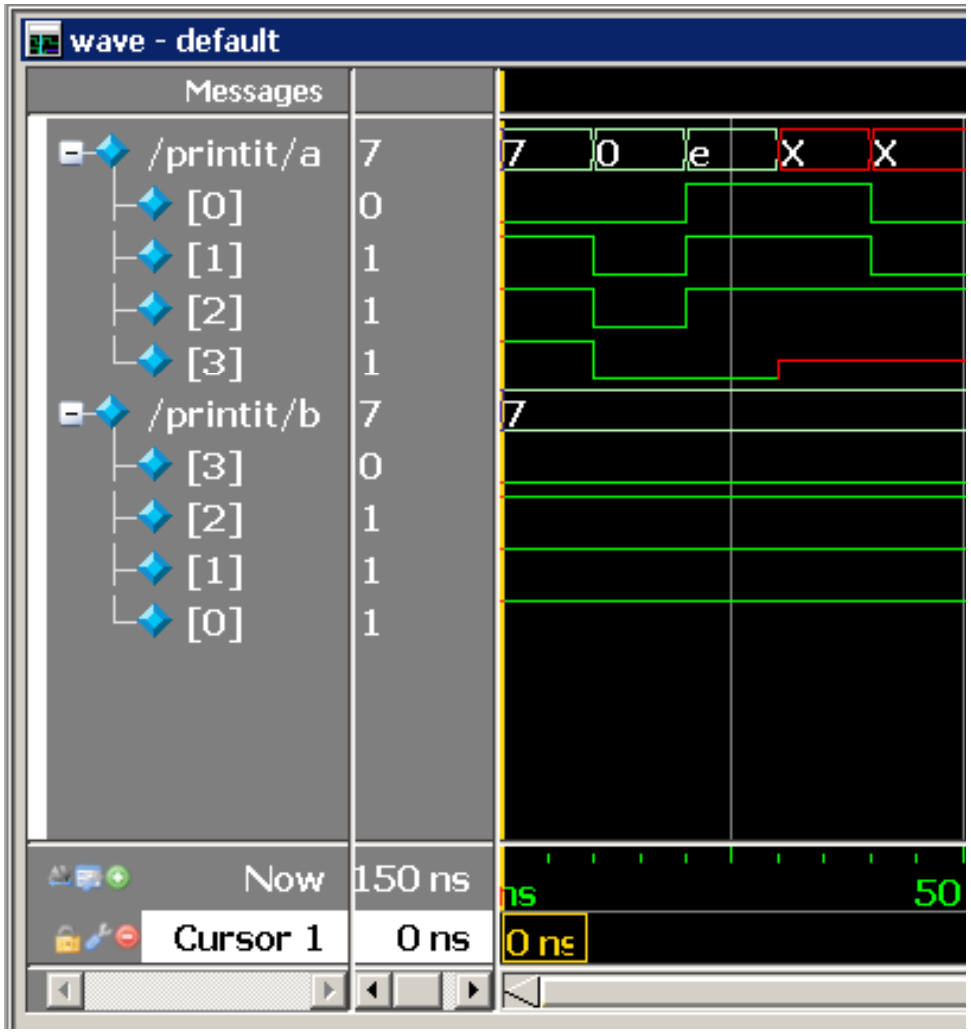
```
#10 a = 4'b1x;
```



a = X

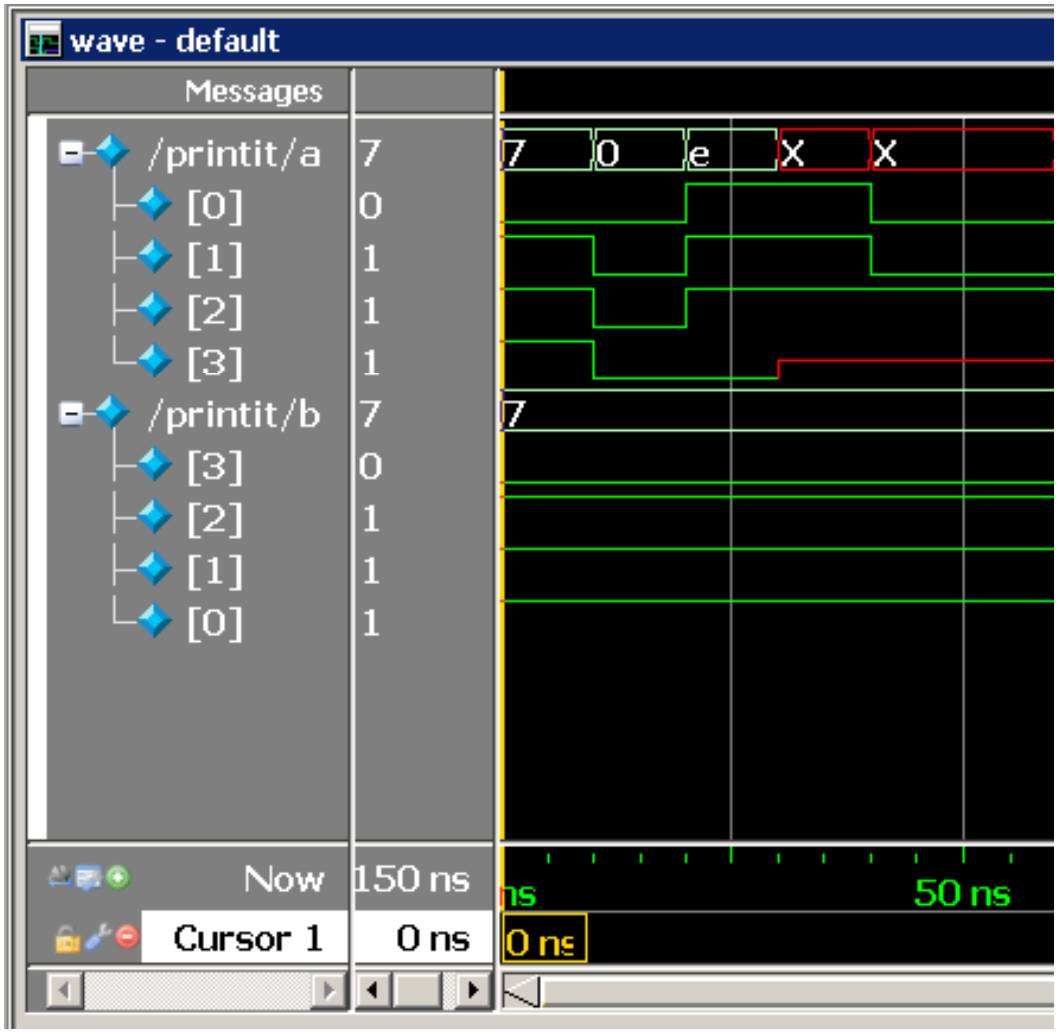
What would happen after:

```
#10 a = 6'b1x;
```



What would happen after:

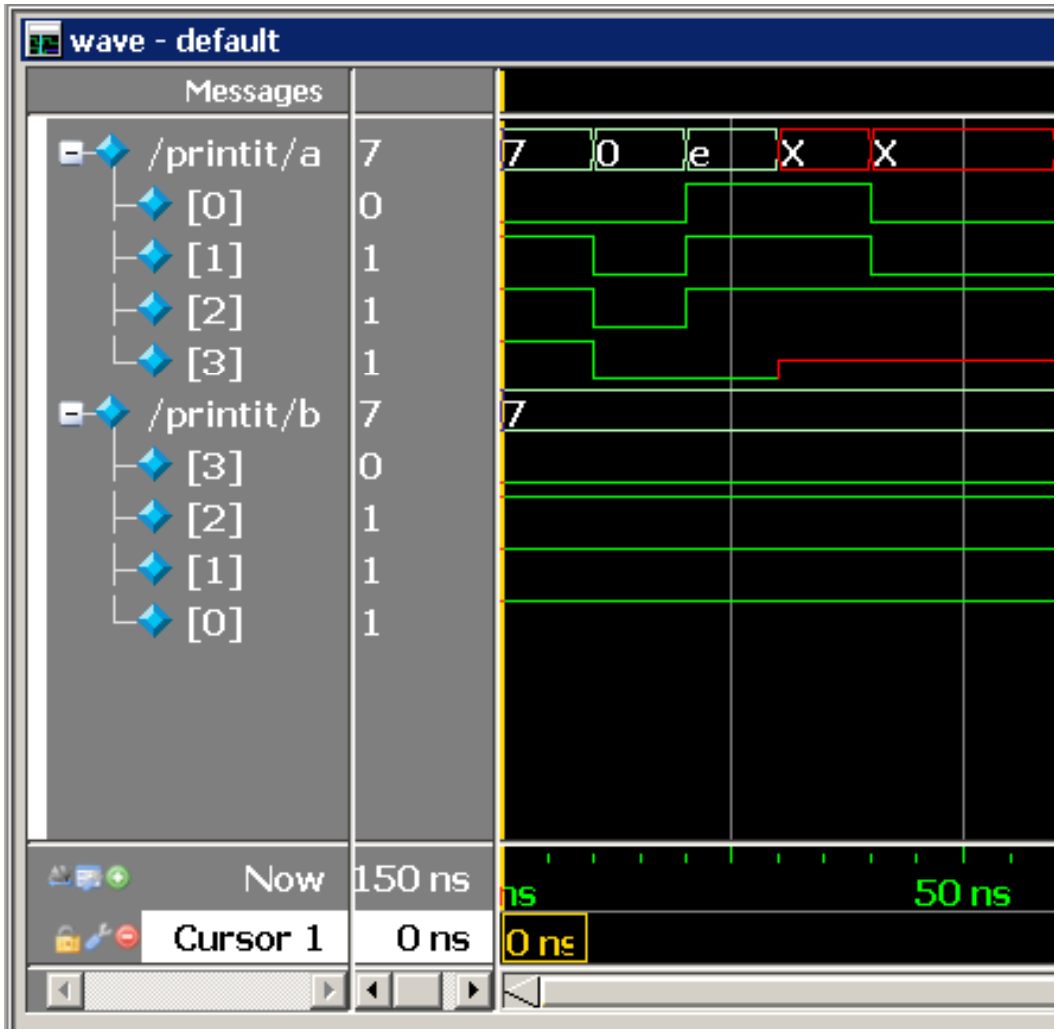
```
#10 a = 6'b1x;
```



a = X

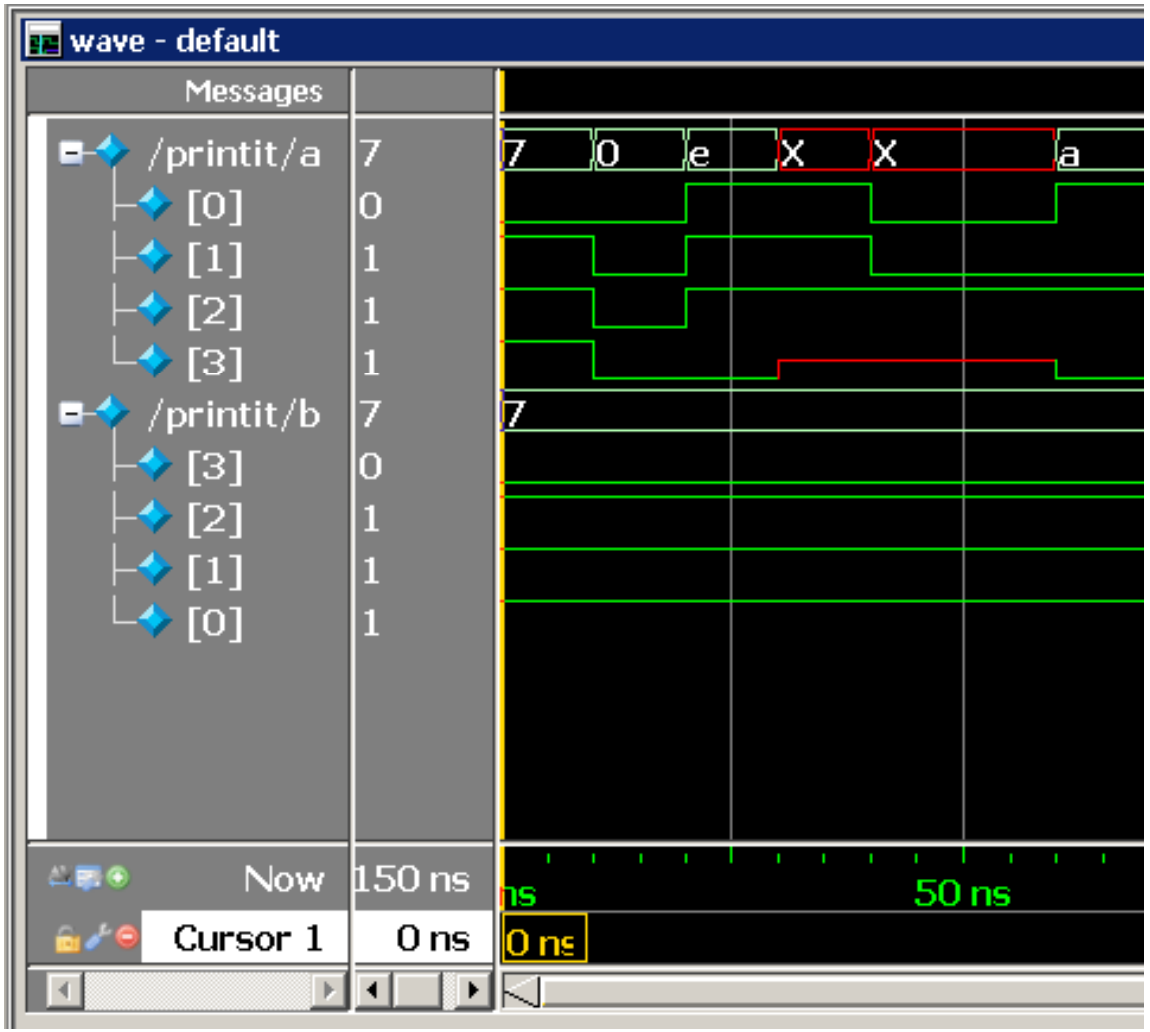
What would happen after:

```
#10 a = 6'b101010;
```



What would happen after:

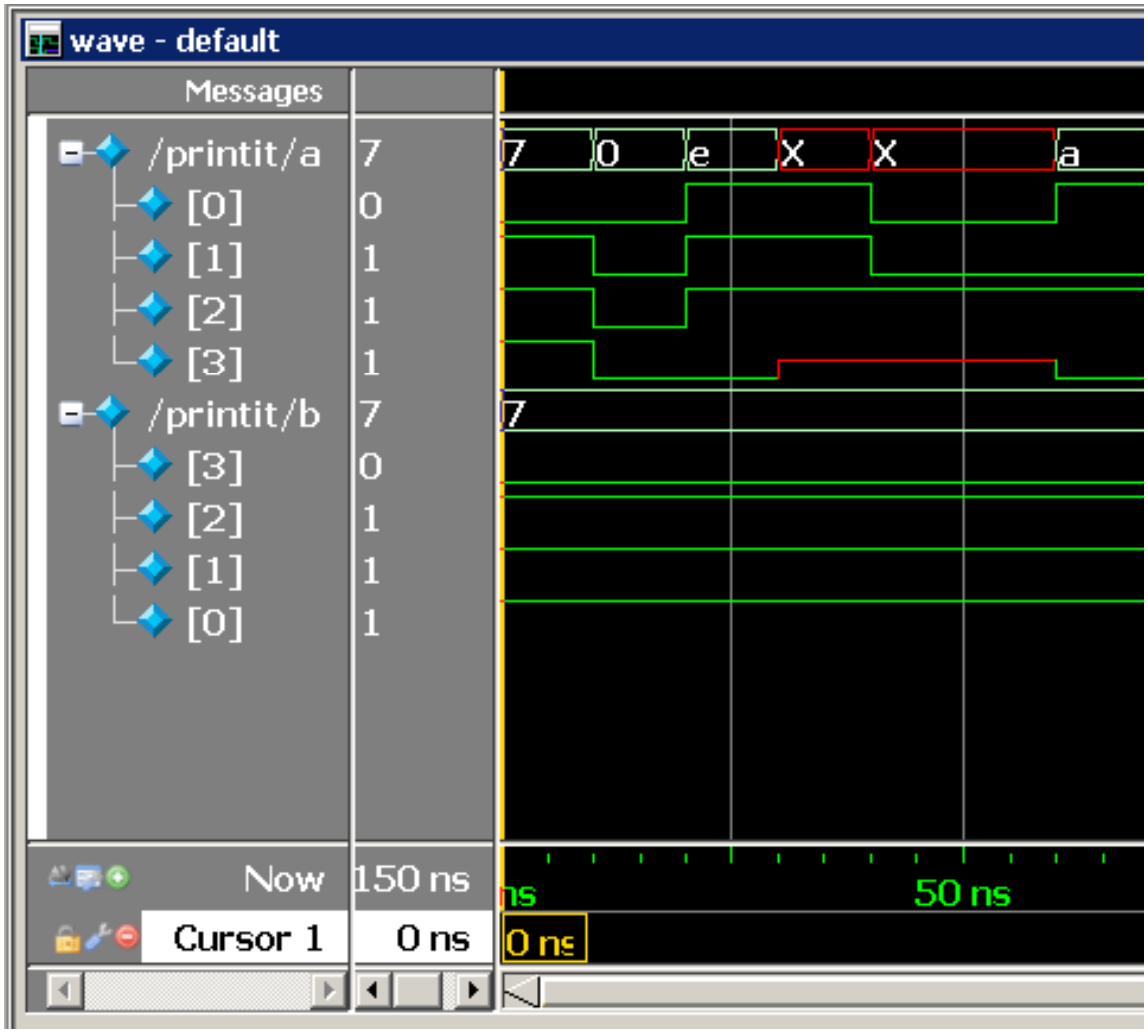
```
#10 a = 6'b101010;
```



a = A

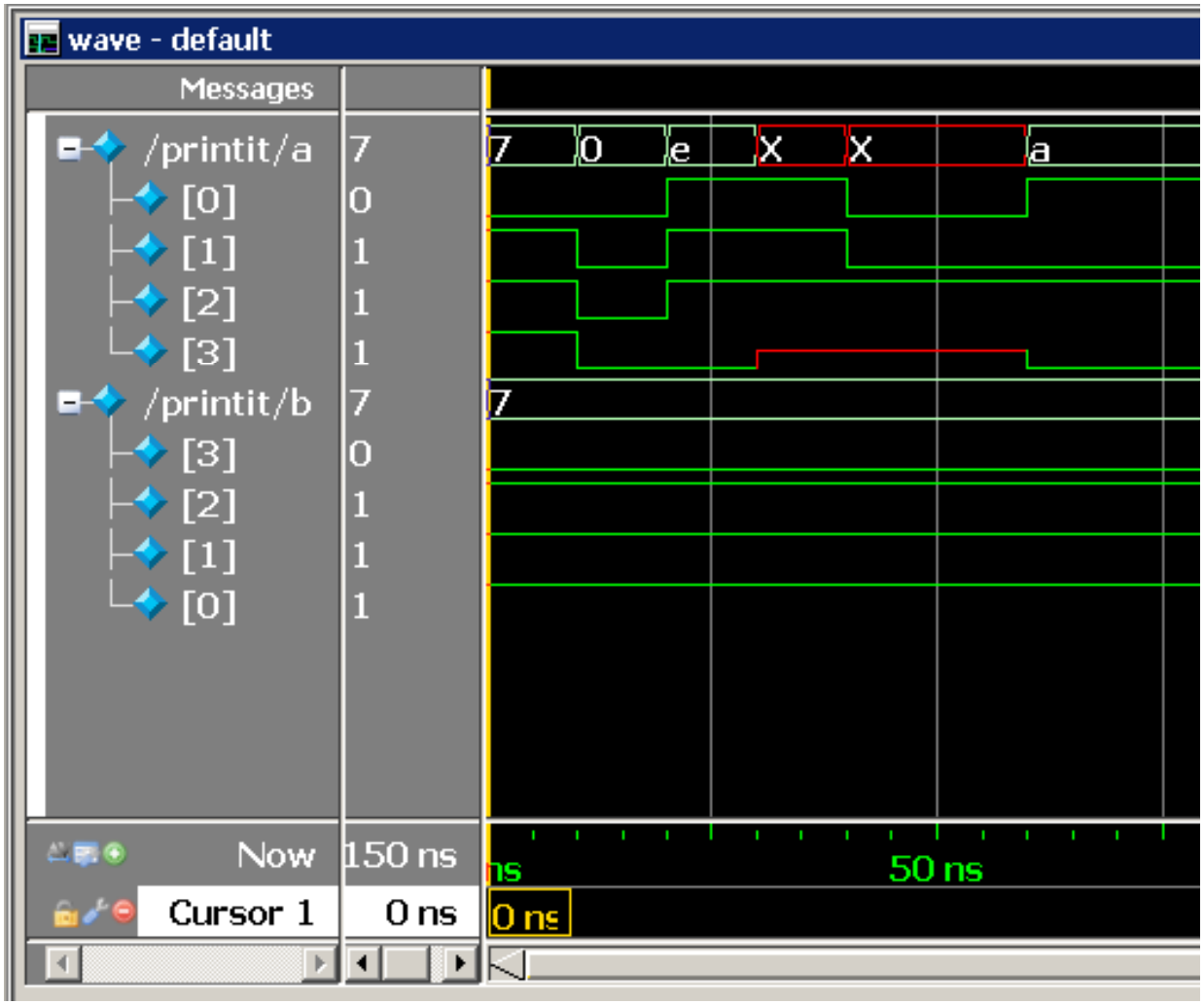
What would happen after:

```
#10 a = 4'ha;
```



What would happen after:

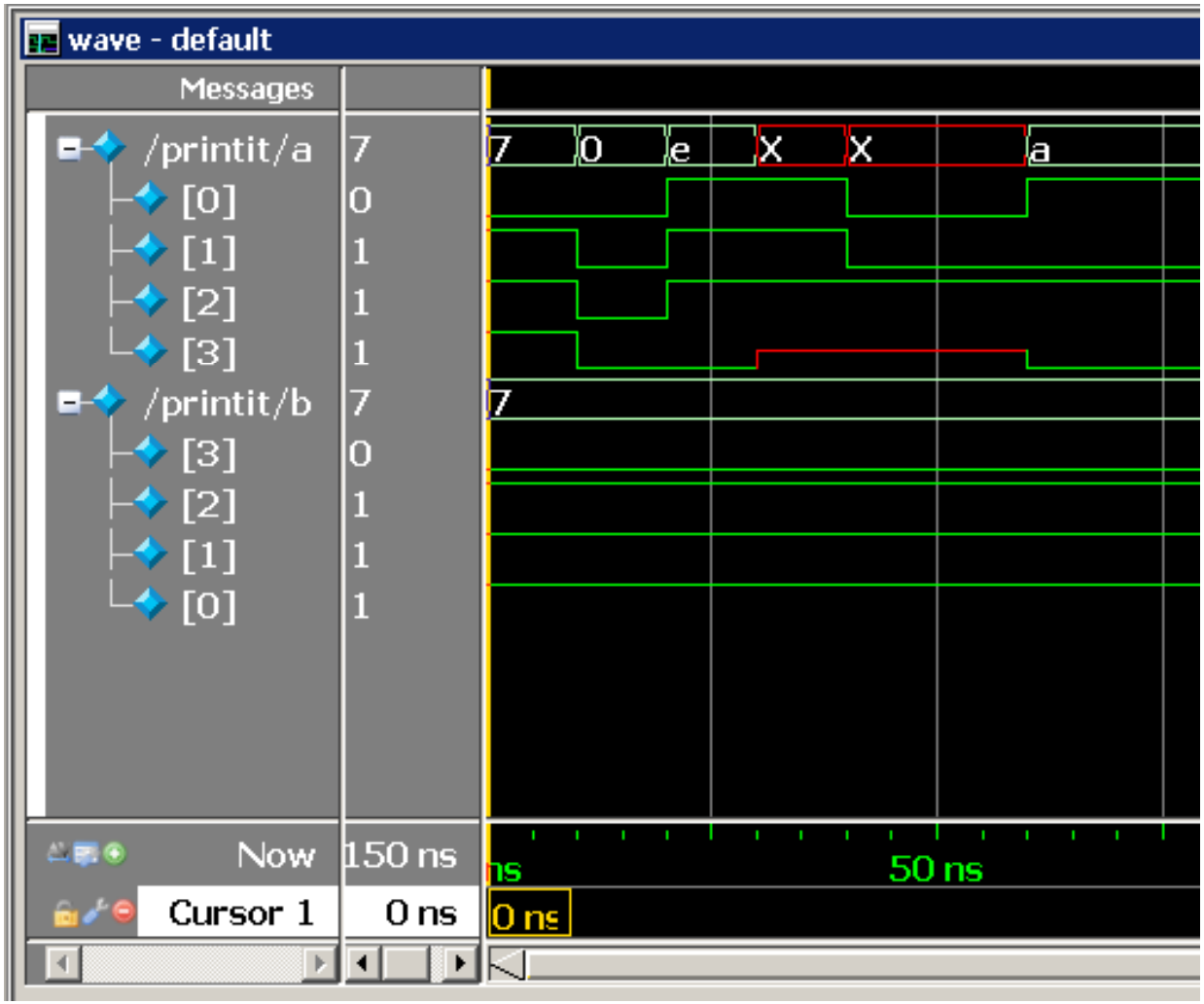
```
#10 a = 4'ha;
```



a = A

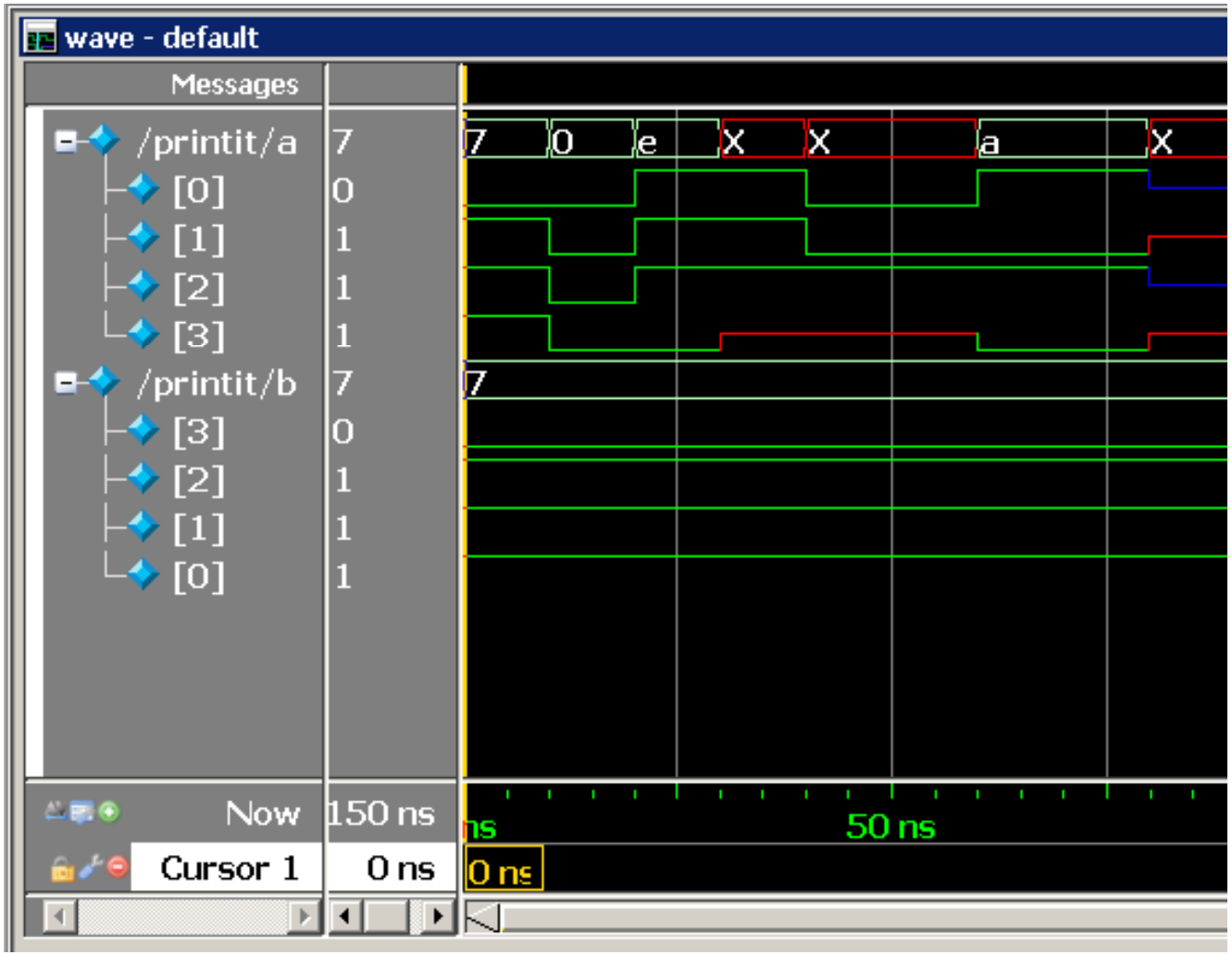
What would happen after:

```
#10 a = 4'bzxzx;
```



What would happen after:

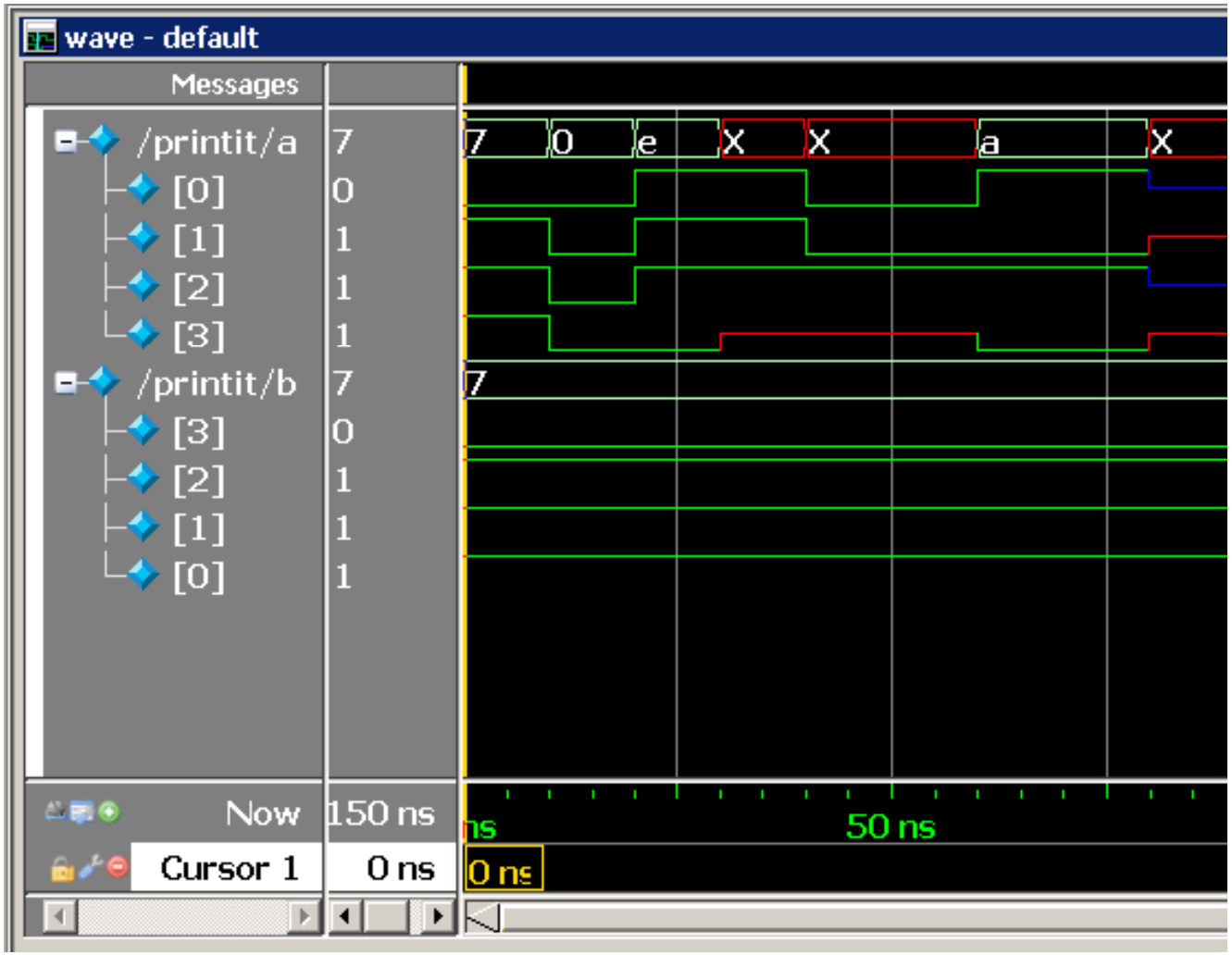
```
#10 a = 4'bzxzx;
```



a = X

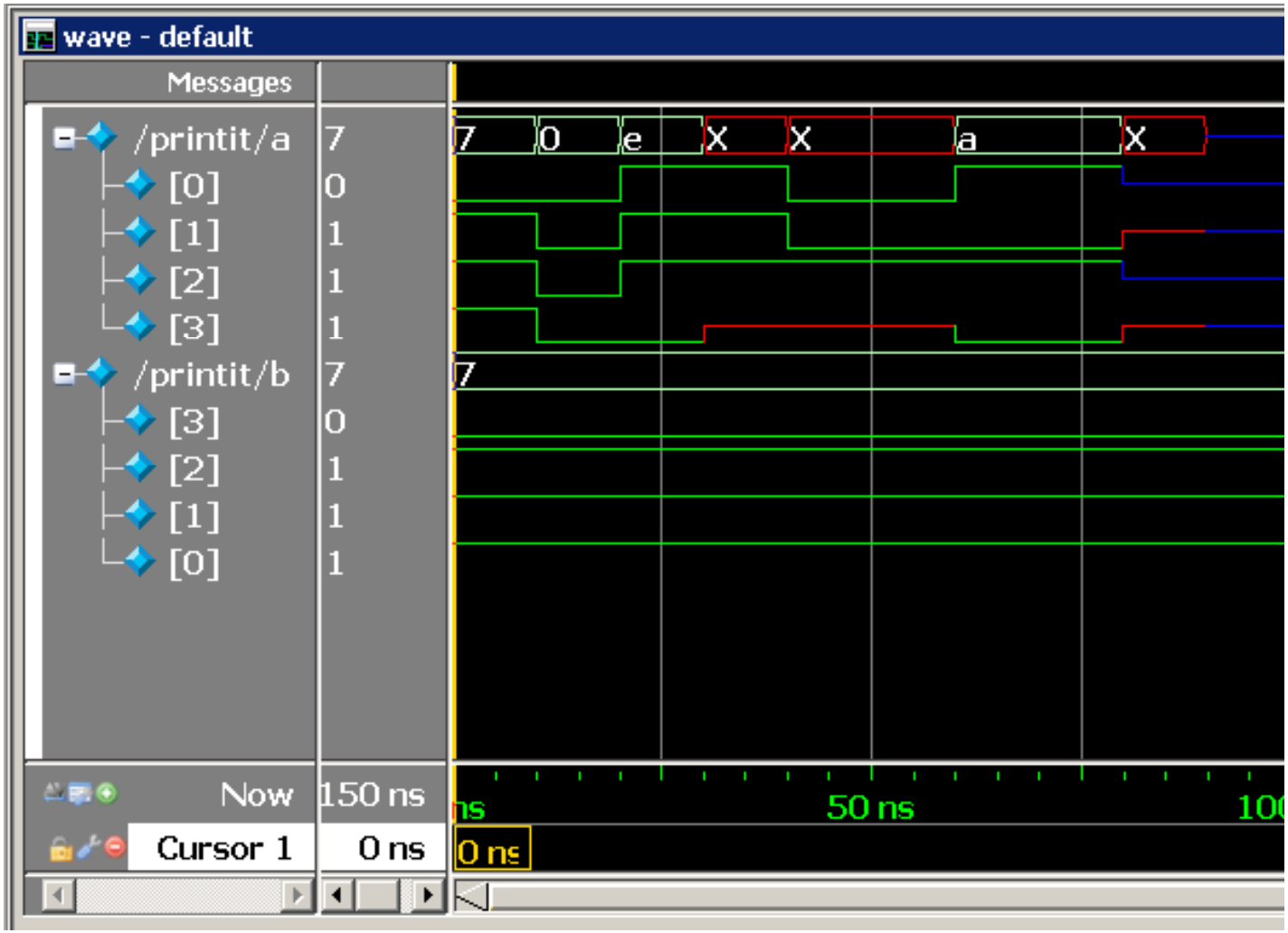
What would happen after:

```
#10 a = 8'hxz;
```



What would happen after:

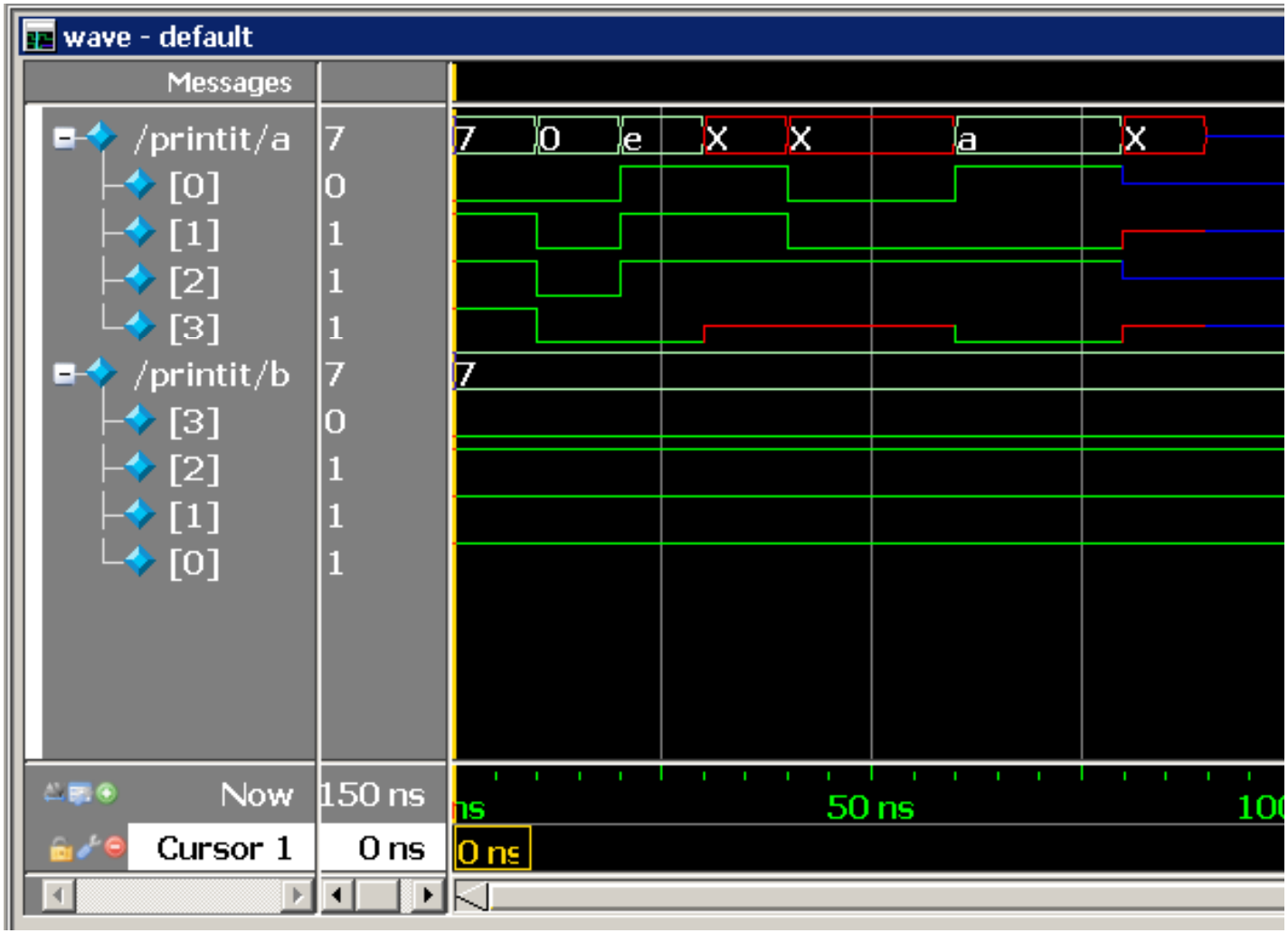
```
#10 a = 8'hxz;
```



a = Z

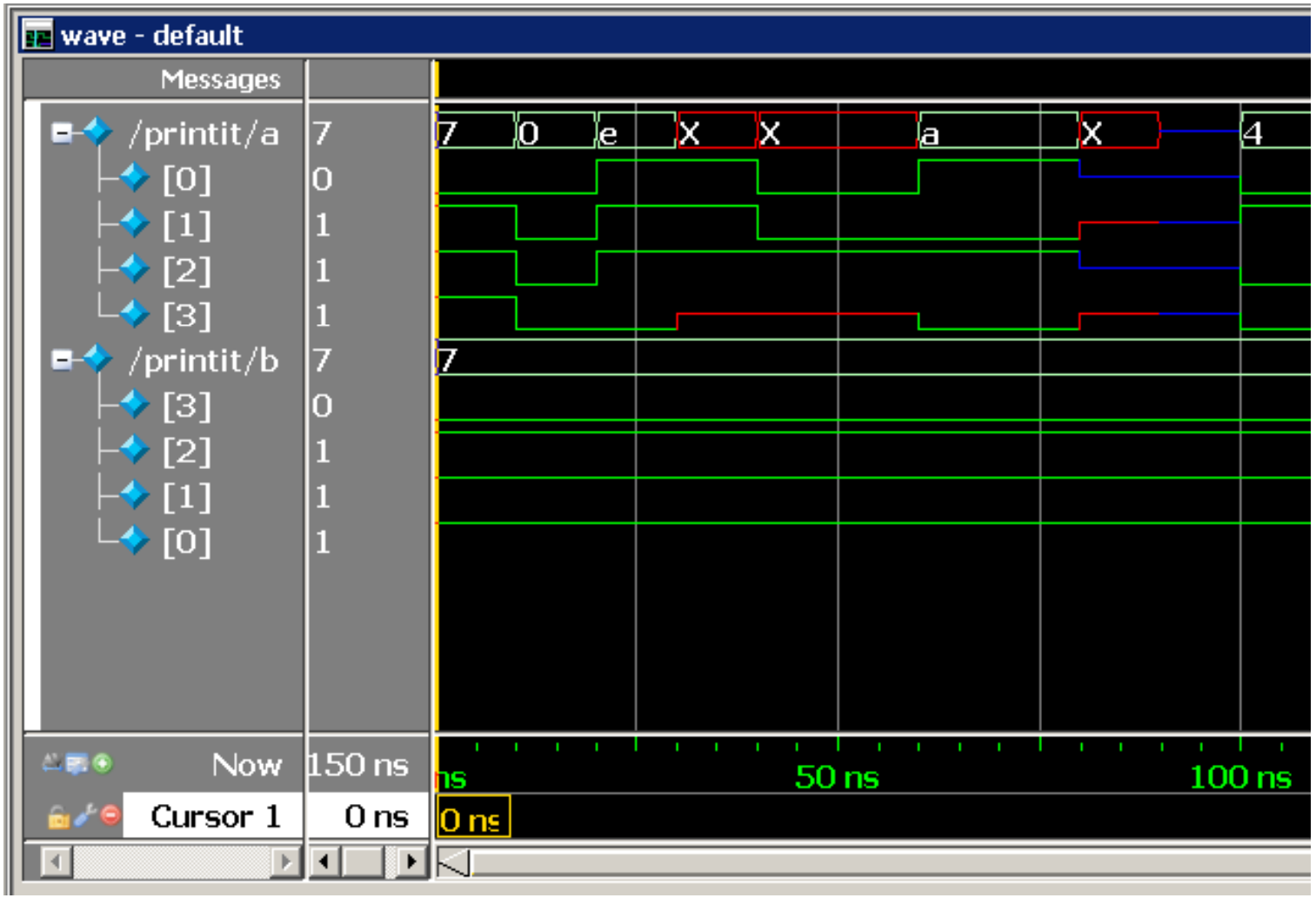
What would happen after:

```
#10 a = 0;
    a[1] = 1'b1;
```



What would happen after:

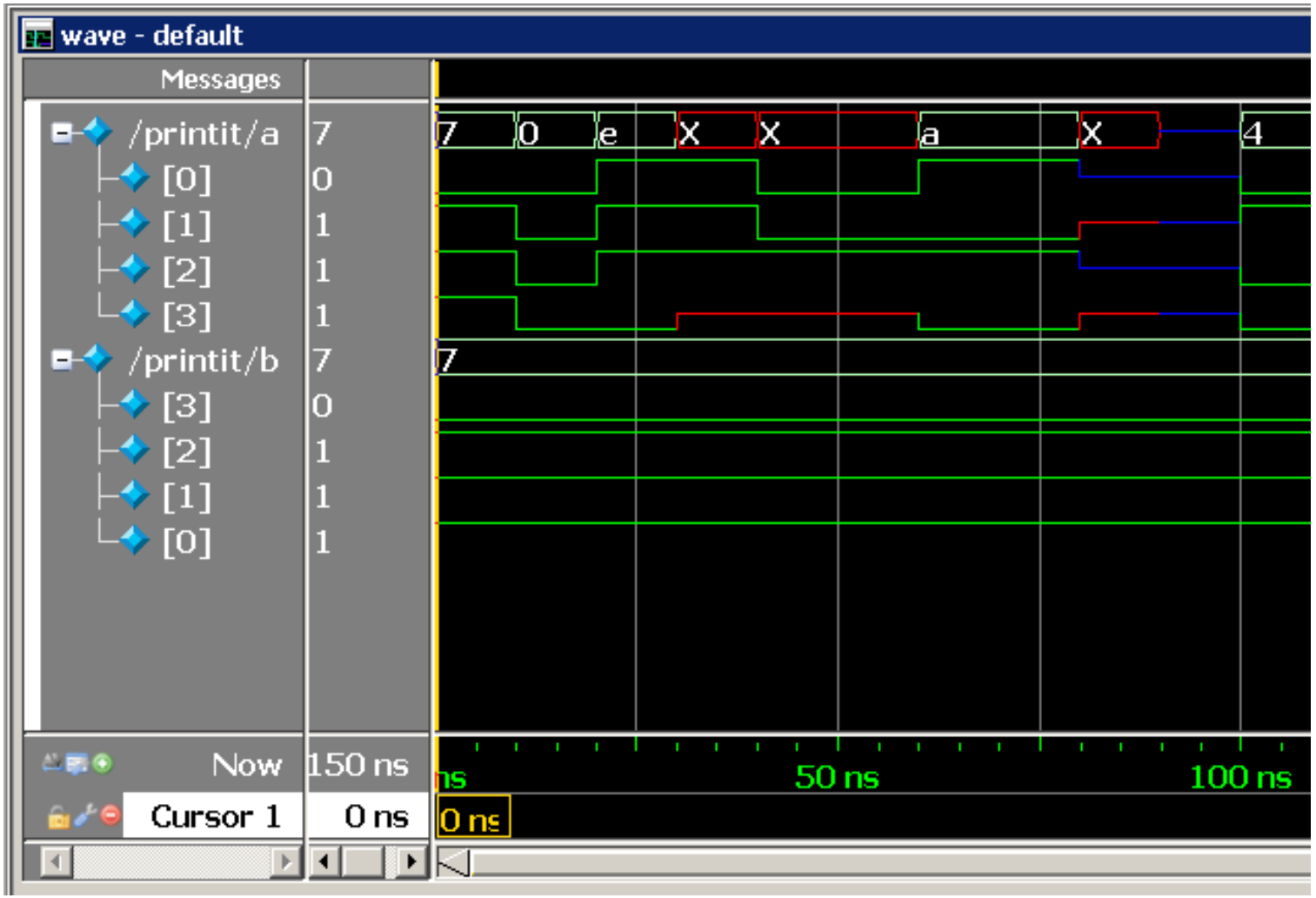
```
#10 a = 0;
    a[1] = 1'b1;
```



a = 4

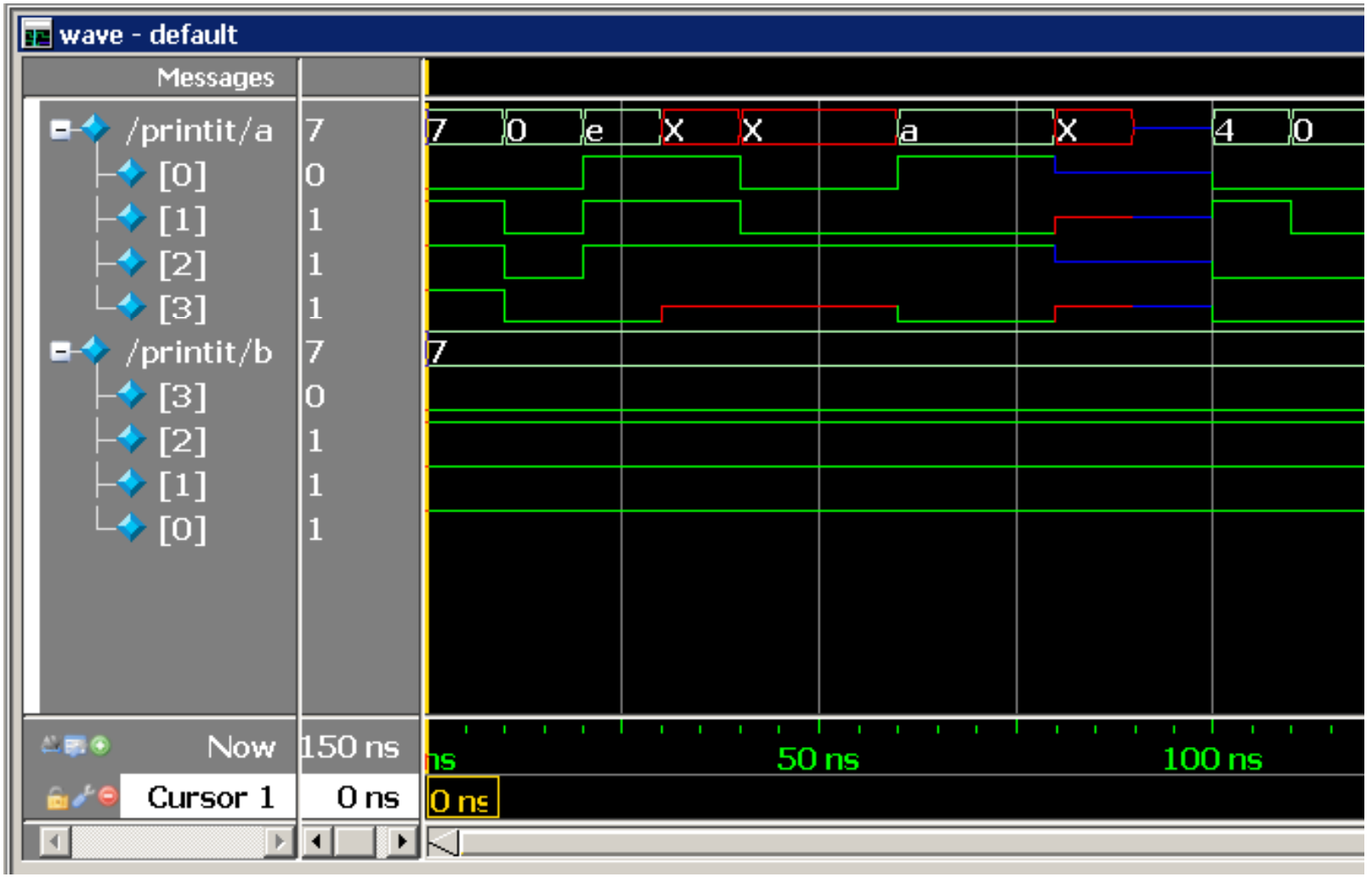
What would happen after:

```
#10 a[1] = b[3];
```



What would happen after:

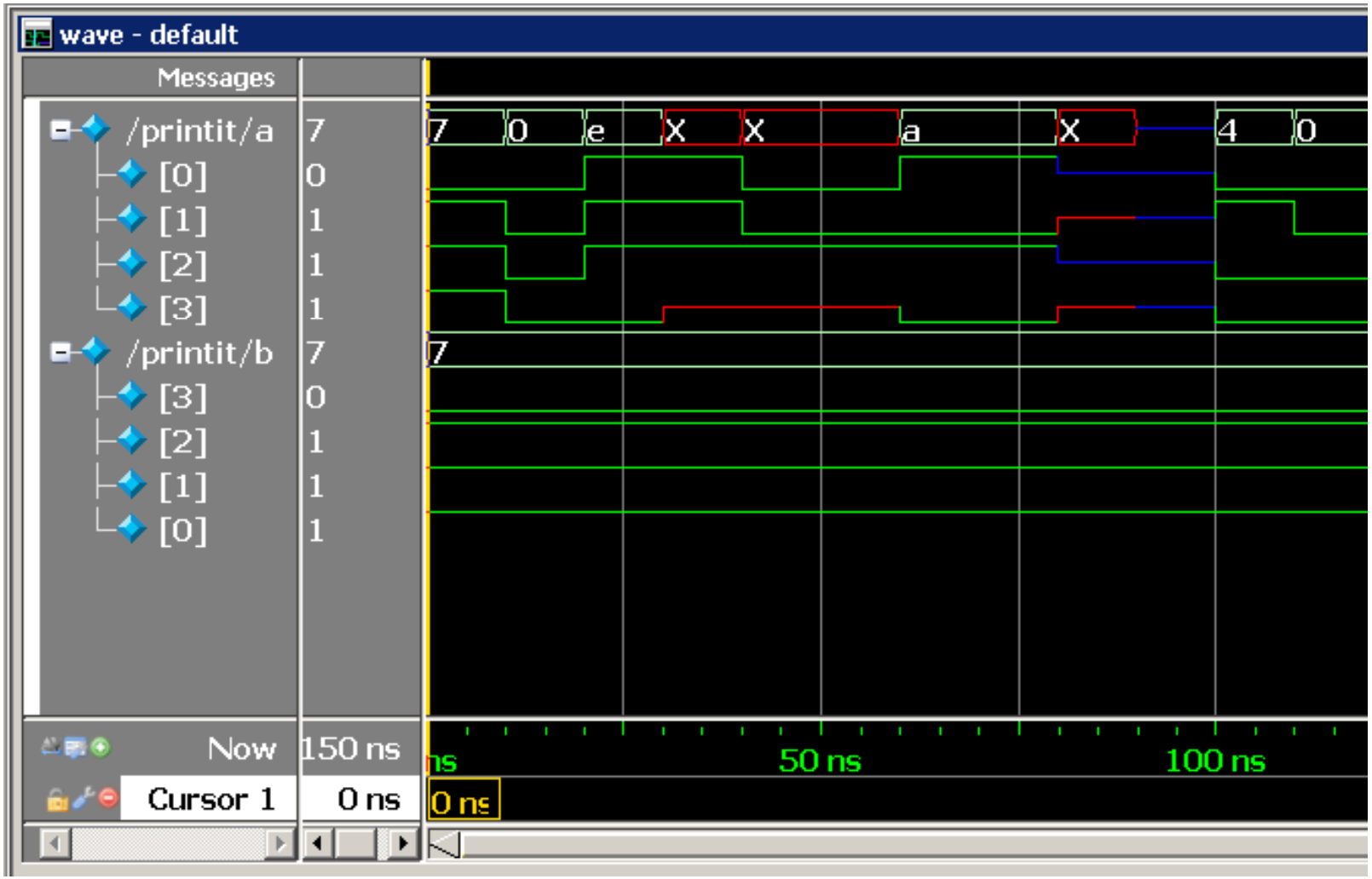
```
#10 a[1] = b[3];
```



a = 0

What would happen after:

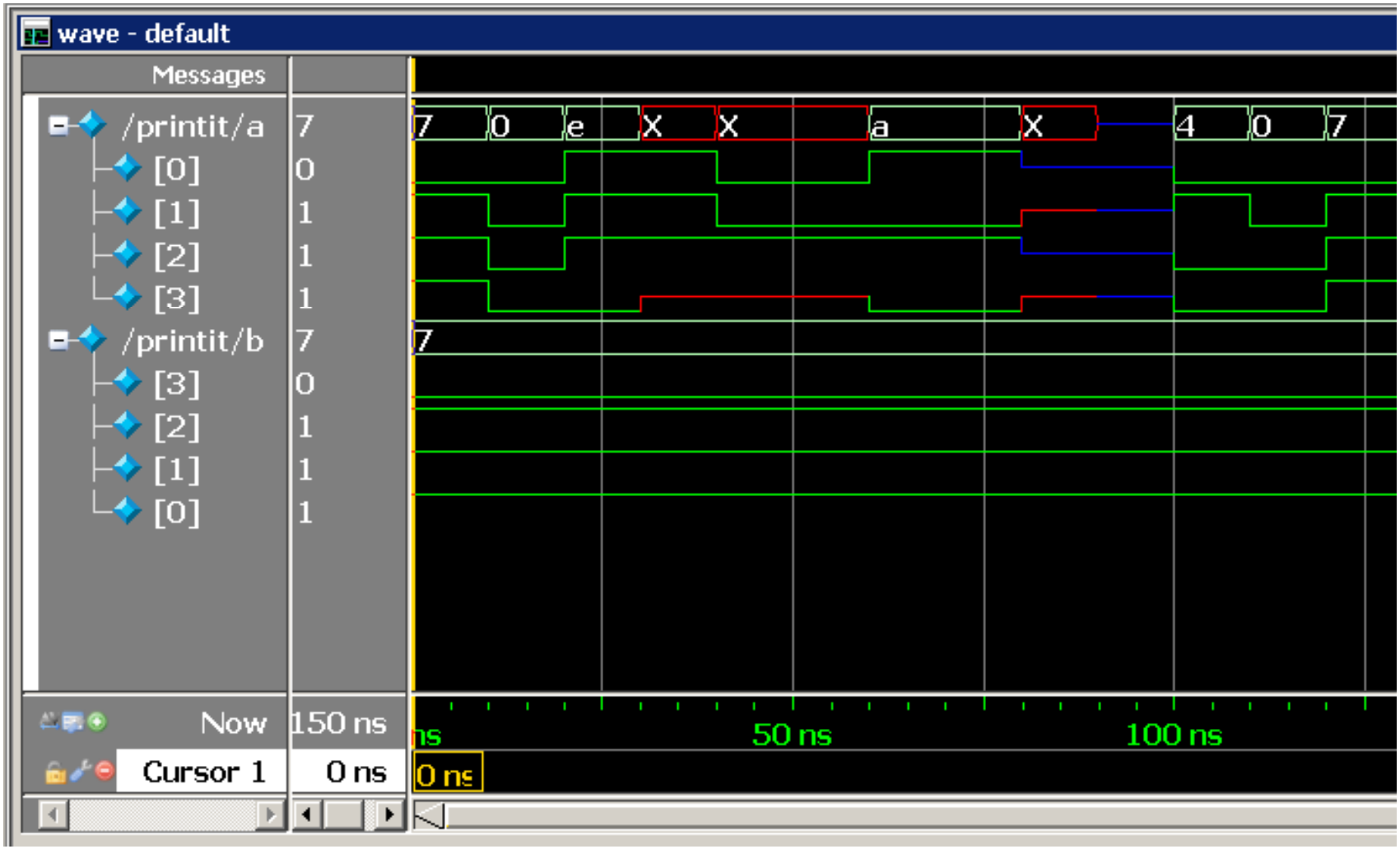
```
#10 a = 0;
    a[0:3] = b[3:0];
```



What would happen after:

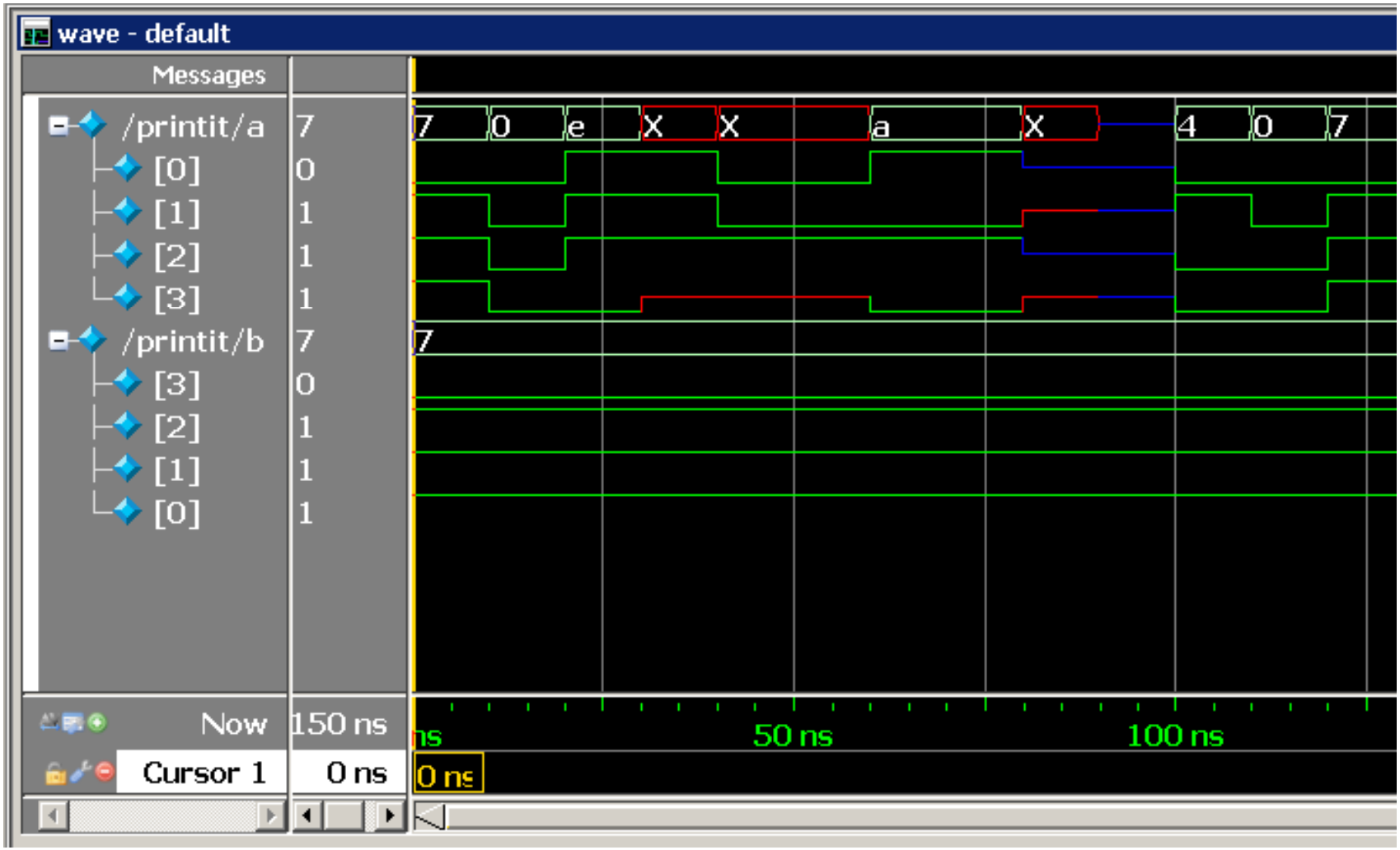
```
#10 a = 0;
    a[0:3] = b[3:0];
```

a = 7



What would happen after:

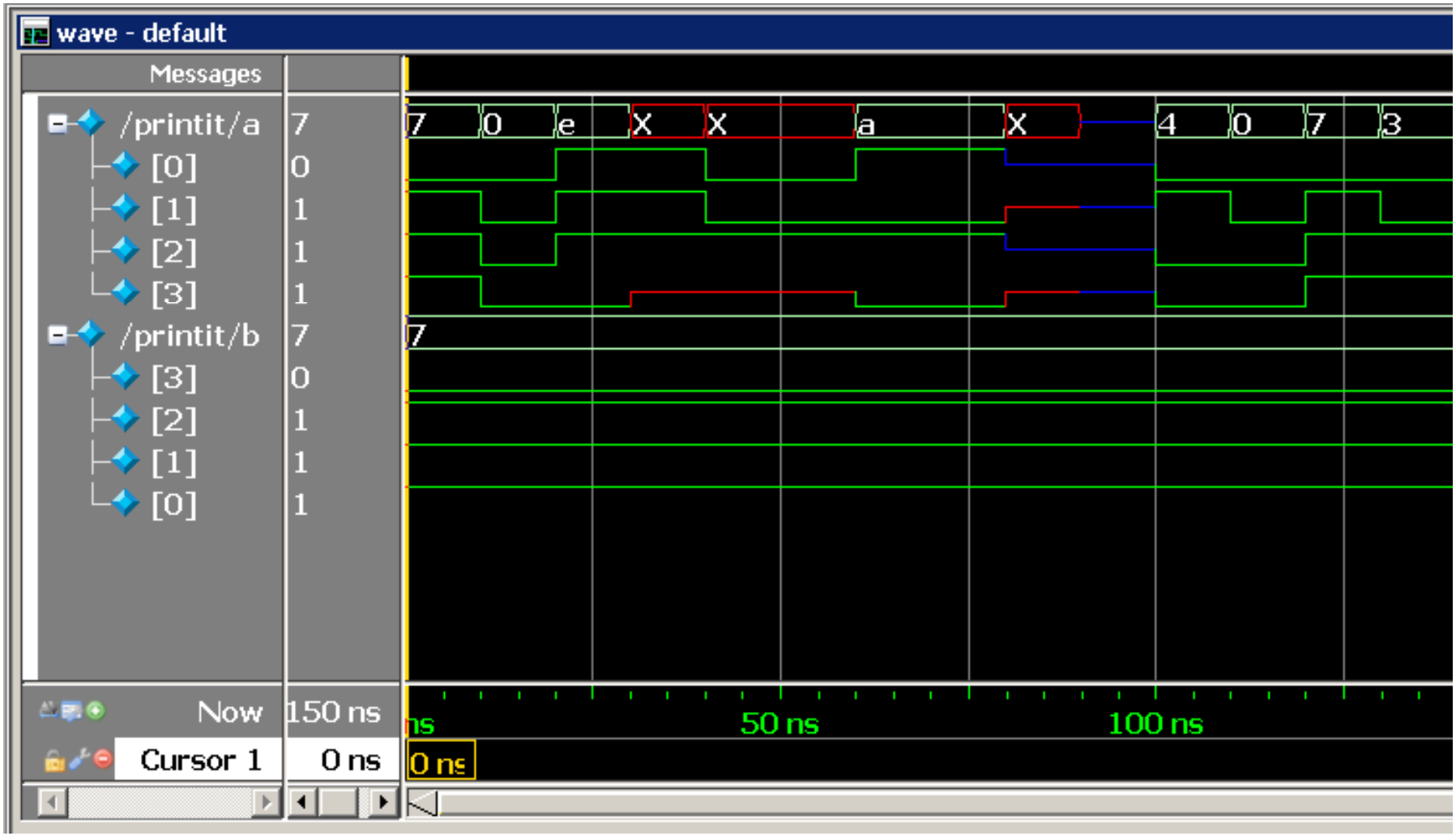
```
#10 a = 0;
    a[1:3] = b[3:1];
```



What would happen after:

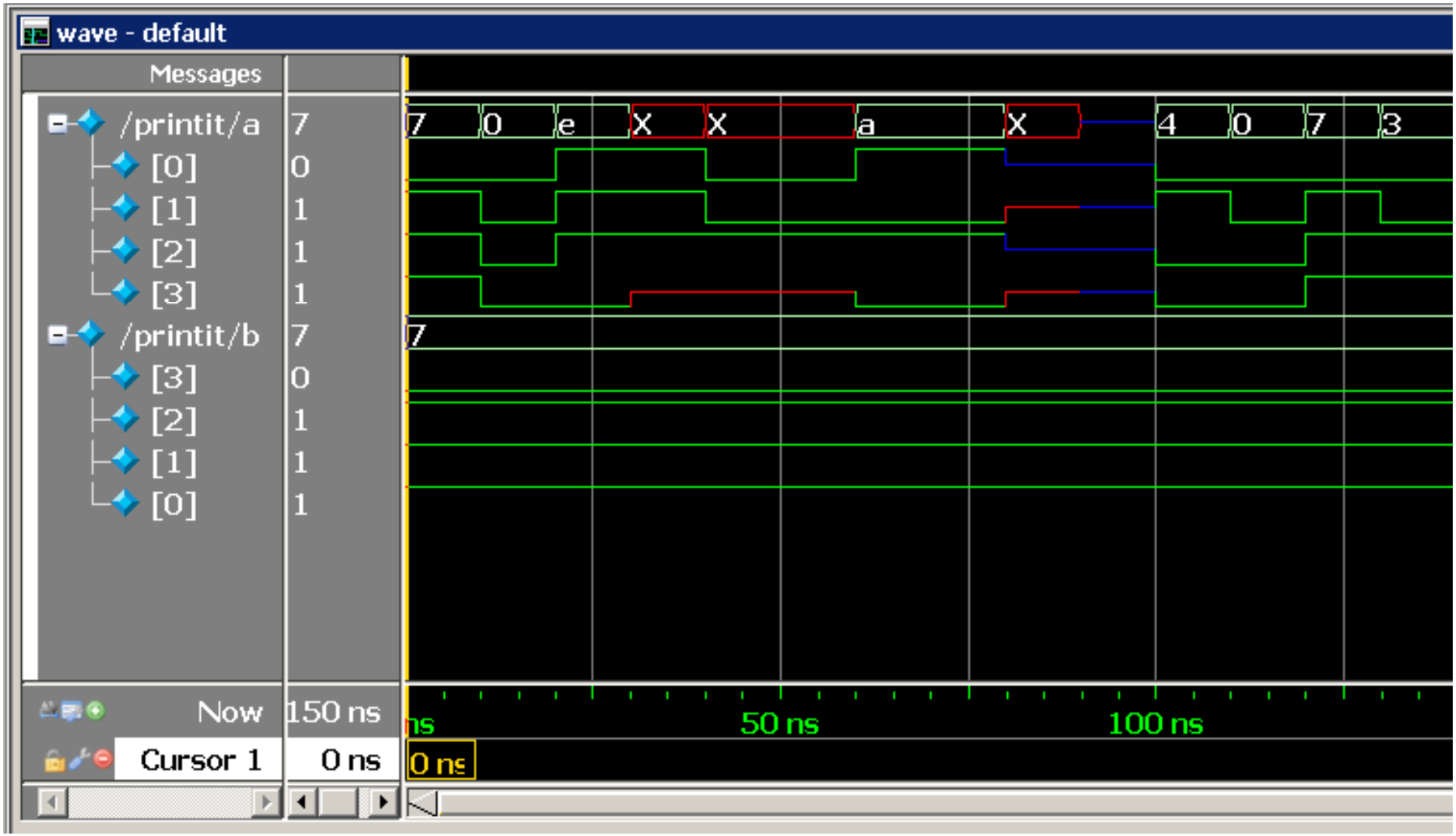
```
#10 a = 0;
    a[1:3] = b[3:1];
```

a = 3



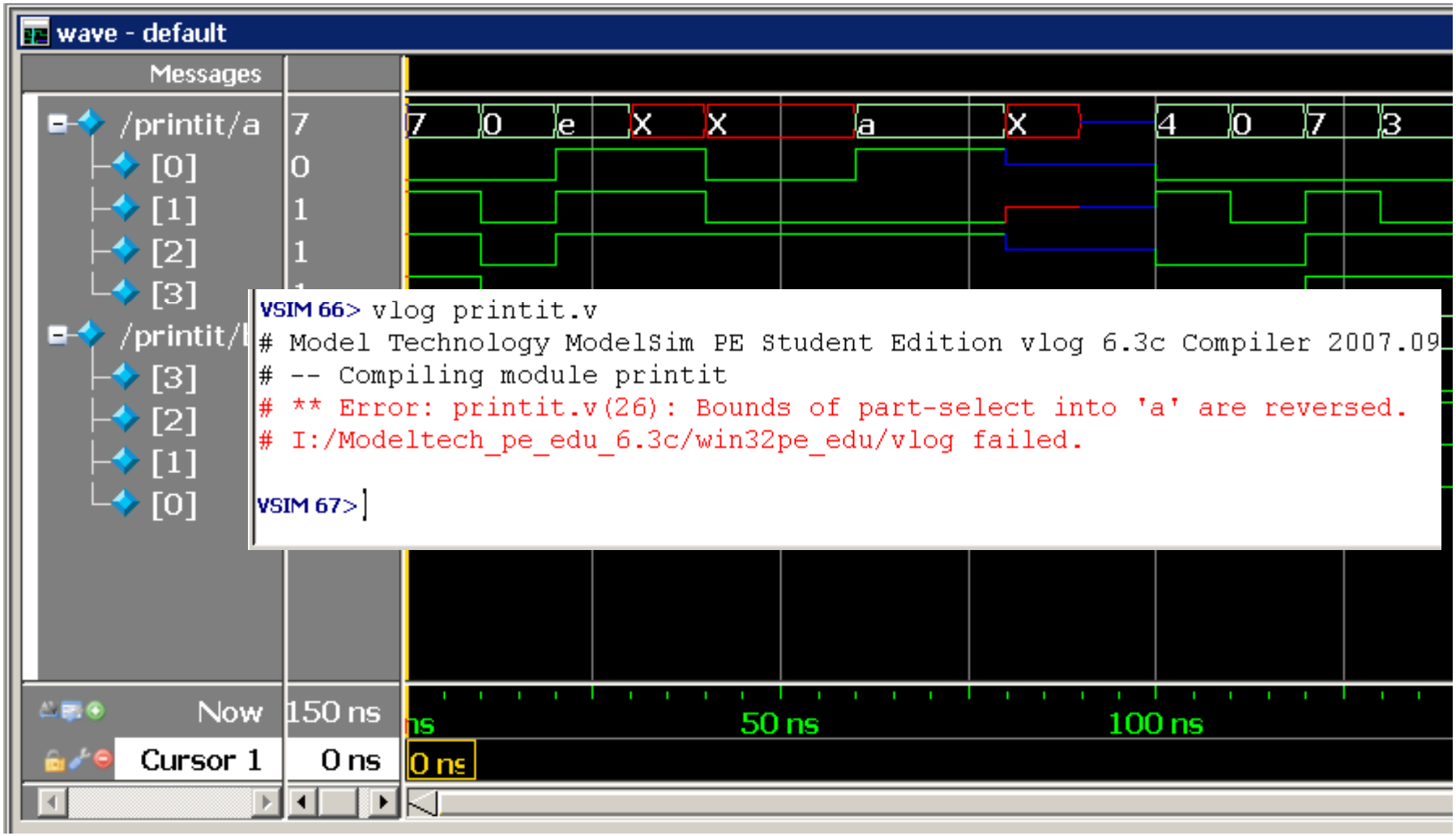
What would happen after:

```
#10 a[3:1] = b[2:0];
```



What would happen after:

```
#10 a[3:1] = b[2:0];
```



We will next talk about

```
module name(portlist);
    port declarations;
    parameter declarations;

    wire declarations;
    reg declarations;
    variable declarations;

    module instantations;
    dataflow statements;
    always blocks;
    initial blocks;

    tasks and functions;

endmodule
```

Port Lists
Port Connections

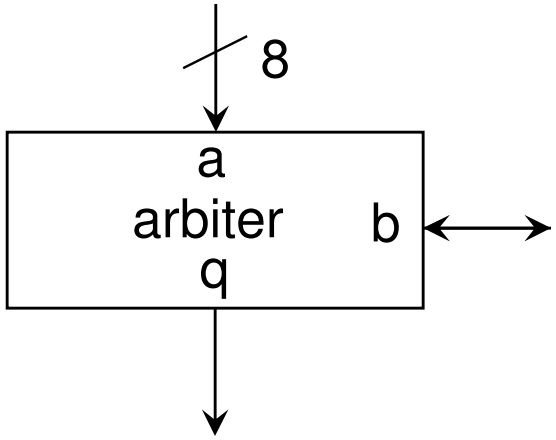
Wire and Reg Declarations
Constants

Other variables

always and initial blocks

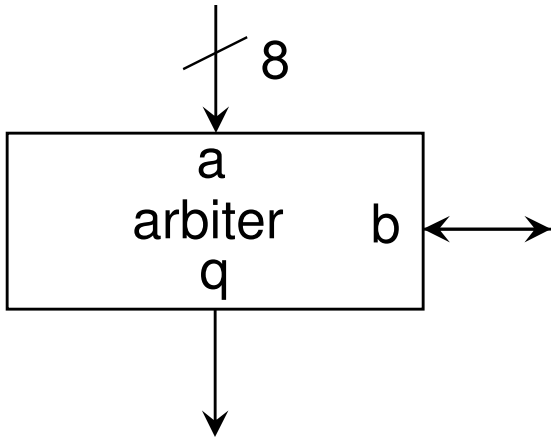
Port List

```
module arbiter(q, a, b);
  output [7:0] q;
  input a;
  inout b;

endmodule
```



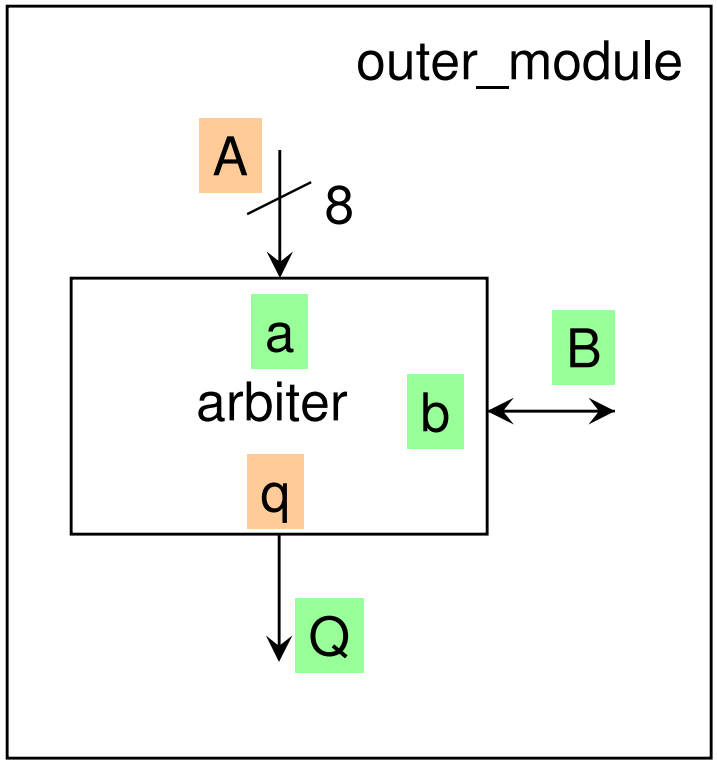
Port List

```
module arbiter(q, a, b);
   output [7:0] q;
   reg    [7:0] q;
   input a;
   inout b;

endmodule
```



- inputs, outputs, inouts are wires by default
- output ports can also be reg

Port Reg Rules



A and q MAY be a reg

a, b, B, Q MUST be a wire

Why is this? Hierarchy is a imaginary boundary.
In other words, q and Q, b and B, a and A are the same value.
Hence, we can have only a single storage location for a single value.

Module Instantation and Port Matching

```
module outer_module;
  wire [7:0] q;
  wire a, b;

  arbiter A1(q, , b);
endmodule
```

Positional Matching

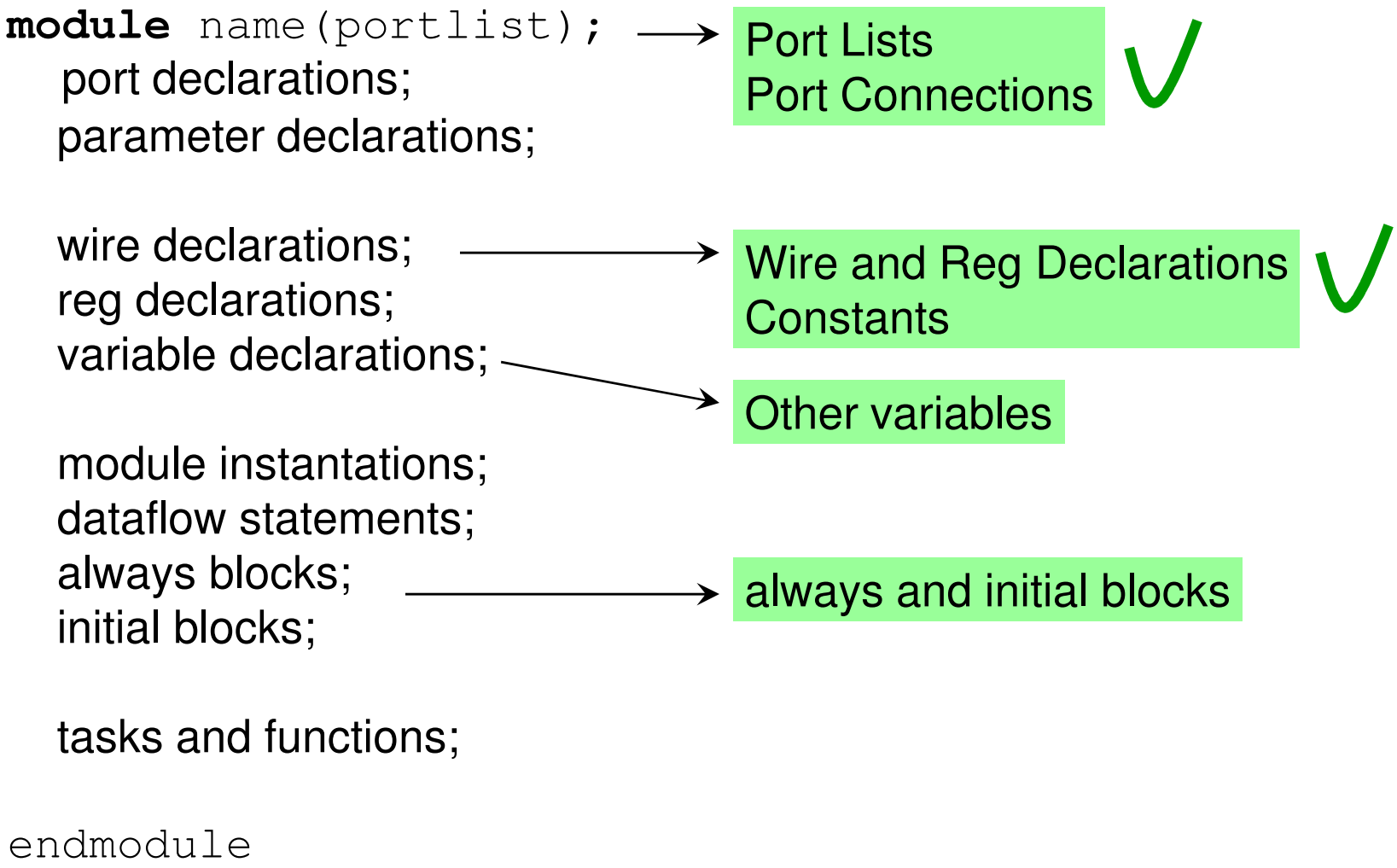
unconnected port

```
module outer_module;
  wire [7:0] Q;
  wire A, B;

  arbiter A1(.q(Q),
             .b(B));
endmodule
```

Name-based Matching

unconnected port
remains unnamed

We will next talk about



Initial and Always Block Statement

```
module initalways;
reg a, b;

initial
begin
   #10 a = 1;
   #10 a = 0;
end

always
begin
   #10 b = 1;
   #10 b = 0;
end

endmodule;
```

The begin .. end is a block statement
Statements within this block execute sequentially

The initial block is a behavioral construct
It executes a single time when t = 0

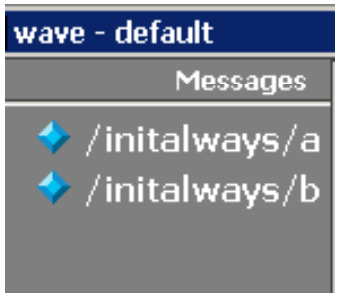
The always block is a behavioral construct
It executes repeatedly

Execution of always can be controlled through sensitivity list (see Thursday)

Initial and Always Block Statement

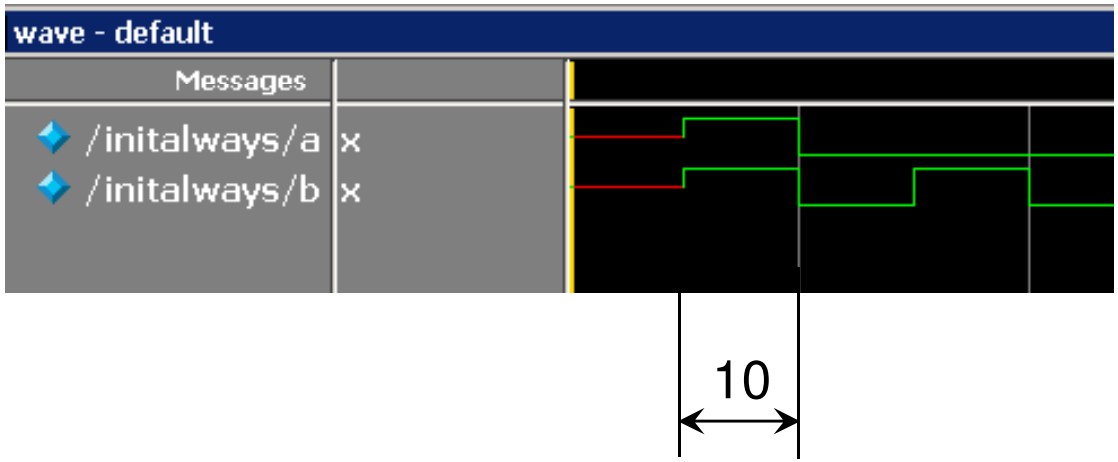
```
module initalways;
reg a, b;

initial
begin
   #10 a = 1;
   #10 a = 0;
end

always
begin
   #10 b = 1;
   #10 b = 0;
end

endmodule;
```

What does this code do ?



Initial and Always Block Statement

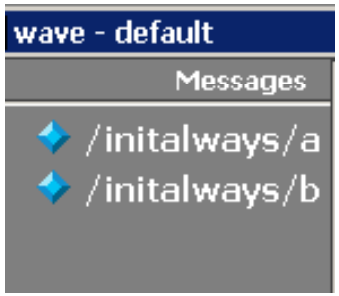
```
module initalways;
reg a, b;

initial
begin
   #10 a = 1;
   #10 a = 0;
end

always
begin
   #10 b = 1;
   #10 b = 0;
end

endmodule;
```



Initial and Always Block Statement

```
module initalways;
reg a, b;

initial
begin
   b = 1;
end

always
begin
   #10 b = ~b;
       a =  b;
end

endmodule;
```

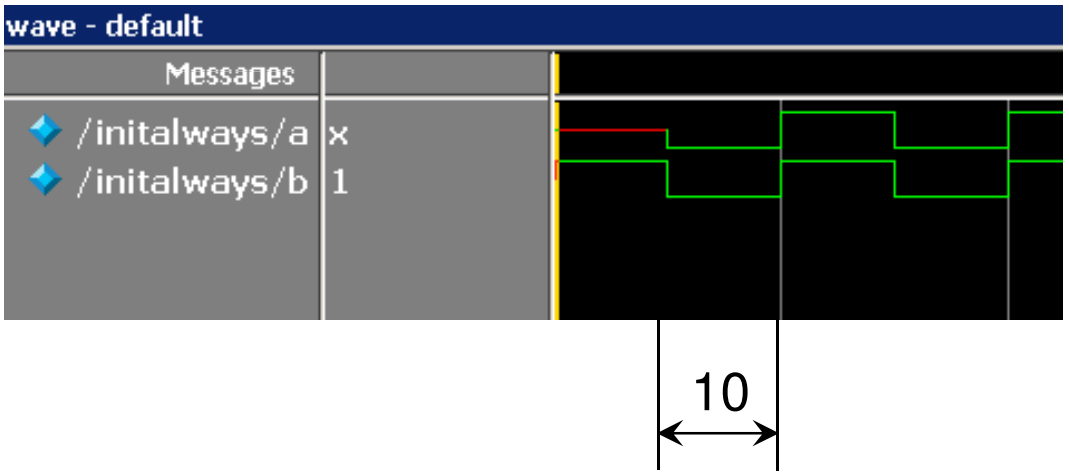
What does this code do ?



'not' operator

Initial and Always Block Statement

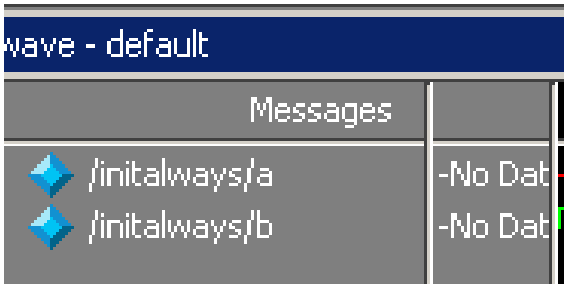
```
module initalways;
reg a, b;

initial
begin
   b = 1;
end

always
begin
   #10 b = ~b;
       a =  b;
end

endmodule;
```



Initial and Always Block Statement

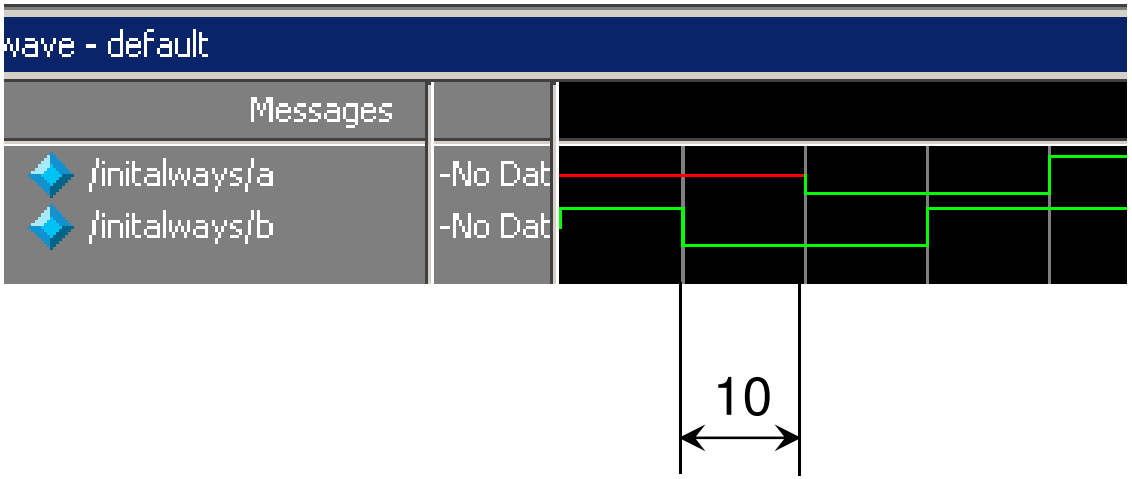
```
module initalways;
reg a, b;

initial
begin
   b = 1;
end

always
begin
   #10 b = ~b;
   #10 a =  b;
end

endmodule;
```

What does this code do ?



Initial and Always Block Statement

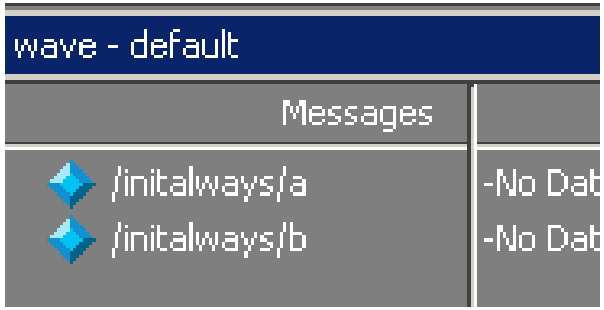
```
module initalways;
reg a, b;

initial
begin
   b = 1;
end

always
begin
   #10 b = ~b;
   #10 a =  b;
end

endmodule;
```



Initial and Always Block Statement

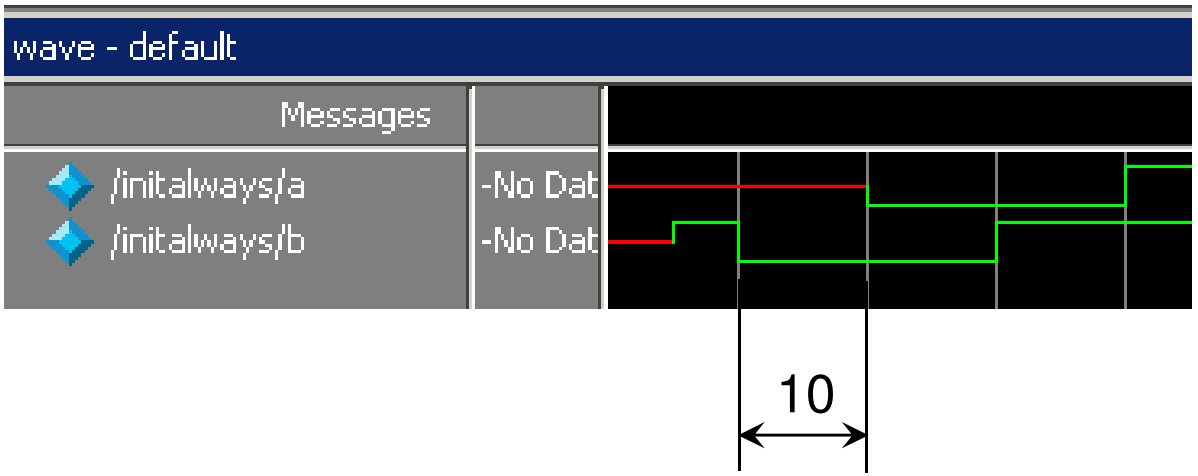
```
module initalways;
reg a, b;

initial
begin
   #5 b = 1;
end

always
begin
   #10 b = ~b;
   #10 a =  b;
end

endmodule;
```

What does this code do ?



Initial and Always Block Statement

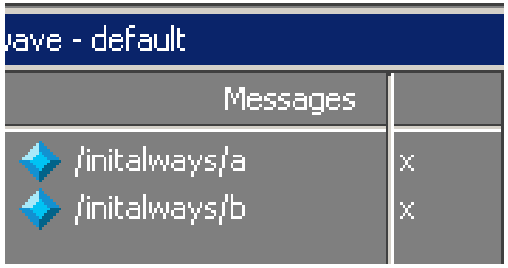
```
module initalways;
reg a, b;

initial
begin
   #5 b = 1;
end

always
begin
   #10 b = ~b;
   #10 a =  b;
end

endmodule;
```



Initial and Always Block Statement

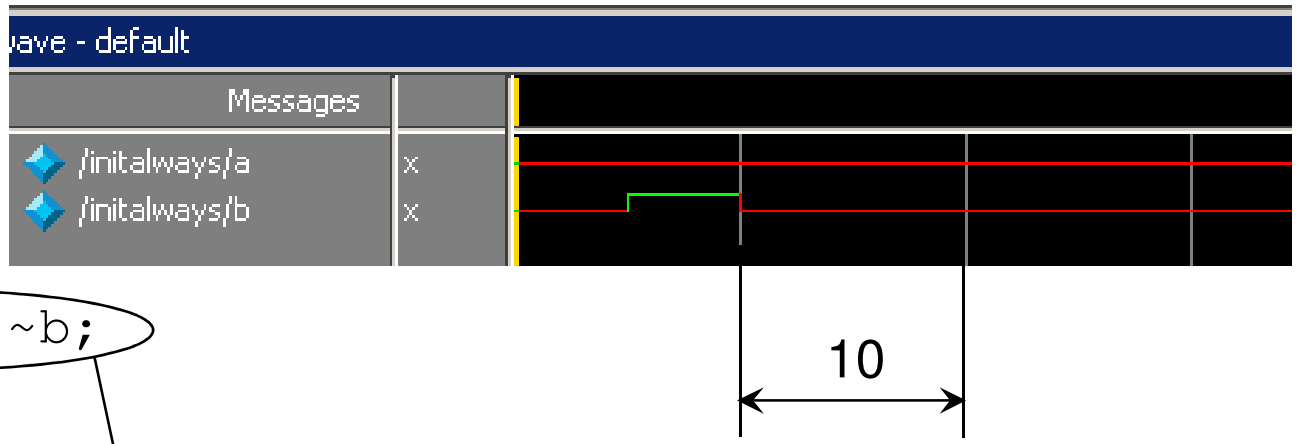
```
module initalways;
reg a, b;

initial
begin
   #5 b = 1;
end

always
begin
        b = #10 ~b;
   #10 a = b;
end

endmodule;
```

What does this code do ?



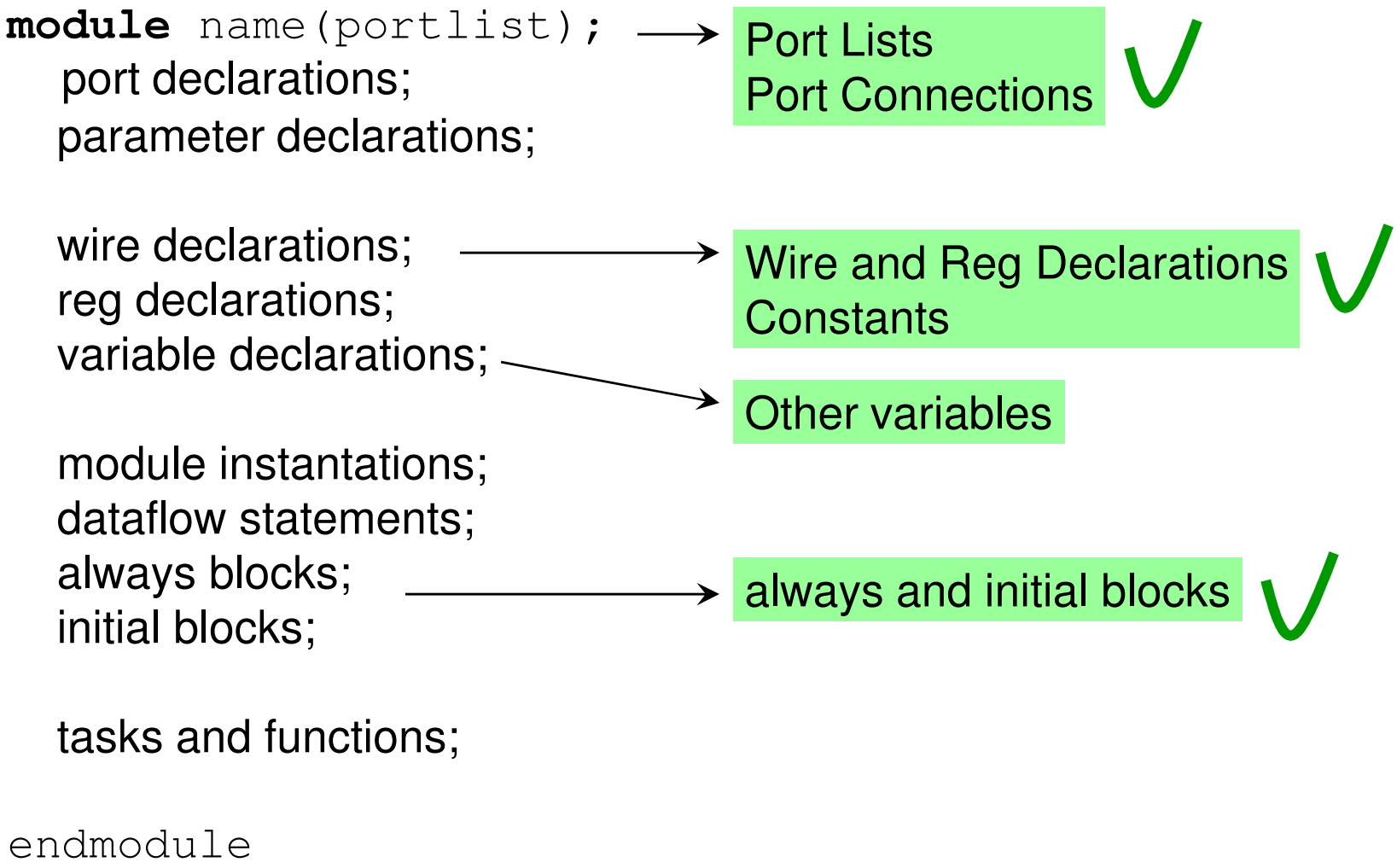
Initial and Always Block Statement

```
module initalways;
reg a, b;

initial
begin
   #5 b = 1;
end

always
begin
      b = #10 ~b;
   #10 a = b;
end

endmodule;
```



Evaluate ~b immediately, then wait 10, and the assign the result to b

We will next talk about



Summary

- Verilog code for Simulation is different from Verilog code for Synthesis
- Verilog module structure
  - Port lists, module instantiation and connection
  - Behavioral constructs (initial and always)
  - Dataflow assignments
  - Structural description
- Verilog data types
  - reg and wire, vectors of reg and wire
- Verilog 4-valued logic and constants