BEE 271 Spring 2017
How to simulate with ModelSim

Nicole Hamilton

Quartus includes a limited simulation feature but for more complex simulations, the answer is ModelSim. Two big advantages of ModelSim are that, compared to Quartus, compiles are lightning quick and you don't need an FPGA board to run your code.

To use ModelSim, you need to add a testbench module to your Verilog file and a few script files to your Quartus project directory. The examples shown here have been posted as `Simulation.zip` to files area in Canvas. Download and unzip it.

Here are the three examples:

1. The lab 2 adding machine
2. A 2-to-1 mux
3. A simple counter

Adding machine example

Here is the basic skeleton you're using for the lab 2 adding machine project, suitable for compiling and running on the DE1-SoC boards. `SevenSegment` is a partially specified seven segment decoder module.

```
module SevenSegment(
      input  [ 3:0 ] hexDigit,
      output [ 6:0 ] segments );

   wire b0, b1, b2, b3;
   wire [ 6:0 ] s;

   assign b0 = hexDigit[ 0 ];
   assign b1 = hexDigit[ 1 ];
   assign b2 = hexDigit[ 2 ];
   assign b3 = hexDigit[ 3 ];

   // s[ 0 ] done. TBD: Assign statements for s[ 1 ] .. s[ 6 ].
   assign s[ 0 ] = b1 & b2 | ~b1 & ~b2 & b3 | ~b0 & b3 |
                   ~b0 & ~b2 | b0 & b2 & ~b3 | b1 & ~b3;

   // Invert the outputs for active low on the DE1-SoC board.
   assign segments = ~s;

endmodule
```

The AddingMachine module is stubbed to exercise driving one of seven segment modules on the DE1-SoC with a four-bit value entered on the switches.

```
module AddingMachine(
      output [ 6:0 ] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5,
      output [ 9:0 ] LEDR,
      input  [ 9:0 ] SW );

   assign LEDR = SW;
   SevenSegment Hex0( SW[ 3:0 ], HEX0 );

endmodule
```

The list of ports has been cleaned up somewhat from a template created by SystemBuilder. Any DE1-SoC signals your module refers to are matched up by name, not order, against signals defined in AddingMachine.qsf (the Quartus Settings File).

The testbench

To simulate this, exercising the SevenSegment module, several testbenches are offered that all do the same thing but are coded slightly differently. This first one simply lists the 16 possible values, waiting 10 ps between assignments.

```
module testbench1( );

   reg  [ 3:0 ] hex;
   wire [ 6:0 ] segments;

   SevenSegment ss ( hex, segments );

   initial
      begin
      hex = 0;   #10;
      hex = 1;   #10;
      hex = 2;   #10;
      hex = 3;   #10;
      hex = 4;   #10;
      hex = 5;   #10;
      hex = 6;   #10;
      hex = 7;   #10;
      hex = 8;   #10;
      hex = 9;   #10;
      hex = 10;  #10;
      hex = 11;  #10;
      hex = 12;  #10;
      hex = 13;  #10;
      hex = 14;  #10;
      hex = 15;  #10;
      end

endmodule
```

The `initial` block tells the simulator what to do when it starts. It does not result in any circuitry when compiled to the FPGA. The `#10;` statement says wait 10 ticks = 10 ps before going to the next statement.

In this variation, each assignment itself is delayed 10 ps. (If you write #10; with nothing between the 10 and semicolon, what you get is a null statement that's delayed 10 ps.)

```
module testbench2( );

   reg  [ 3:0 ] hex;
   wire [ 6:0 ] segments;

   SevenSegment ss ( hex, segments );

   initial
      begin
          hex = 0;
      #10 hex = 1;
      #10 hex = 2;
      #10 hex = 3;
      #10 hex = 4;
      #10 hex = 5;
      #10 hex = 6;
      #10 hex = 7;
      #10 hex = 8;
      #10 hex = 9;
      #10 hex = 10;
      #10 hex = 11;
      #10 hex = 12;
      #10 hex = 13;
      #10 hex = 14;
      #10 hex = 15;
      end

endmodule
```

As delivered, this is the one that's set up to be used. In this variation, an integer i is used as a control variable in for loop to walk through the 16 states, waiting 10 ps after each.

```
module testbench( );

   reg  [ 3:0 ] hex;
   wire [ 6:0 ] segments;
   integer i;

   SevenSegment ss ( hex, segments );

   initial
      for ( i = 0;  i < 16;  i = i + 1 )
         begin
         hex = i; #10;
         end

endmodule
```

Running the simulation

These are the important files added for the simulation, based on the originals created by Professor Scott Hauck at UW Seattle. All of them are editable text.

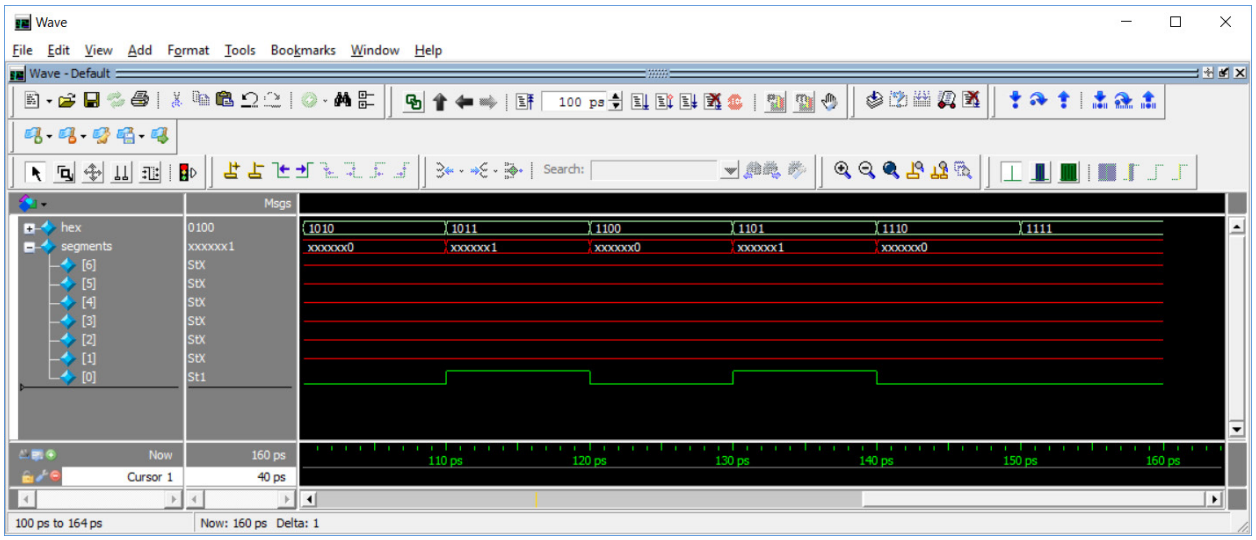
1. Launch_ModelSim.bat
2. Simulate.do
3. wave.do

To add simulation to a project of your own, copy these files into that directory, then edit them as appropriate, as shown below.

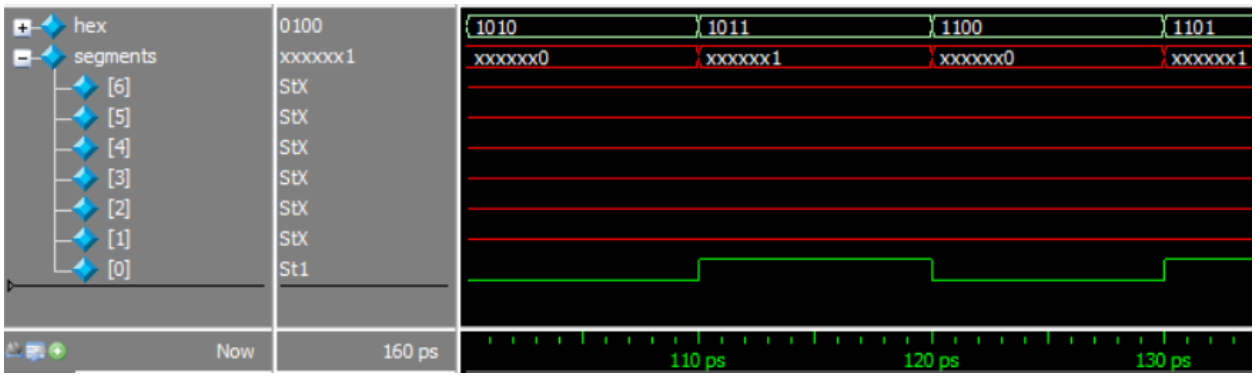
To run the simulator:

1. Start the simulator using LaunchModelSim.bat. (You may have to fix the path specified in this file.)
2. In the ModelSim Transcript window (its command line) type "do Simulate.do".

You should see it display a simulation Wave window showing the hex input changing and individual segment outputs.



Here's a zoom in on a part of it. Because the equations haven't been filled in yet for segments[6:0], they are in red, meaning their values are unknown.



You can change the format, e.g., magnifying the timescale or adding or deleting signals or how they're displayed, then save the result through File → Save Format … on the menu bar. It will suggest overwriting the wave.do file so that the next time you type "do Simulate.do", the wave display will open automatically in that new format.

The simulator files

Here are the contents of the files. Note the sections highlighted in yellow that may have to be fixed to match your system or will have to be changed if you copy them into your own project.

Launch_ModelSim.bat

```
C:\altera\16.1\modelsim_ase\win32aloem\modelsim.exe
```

Simulate.do

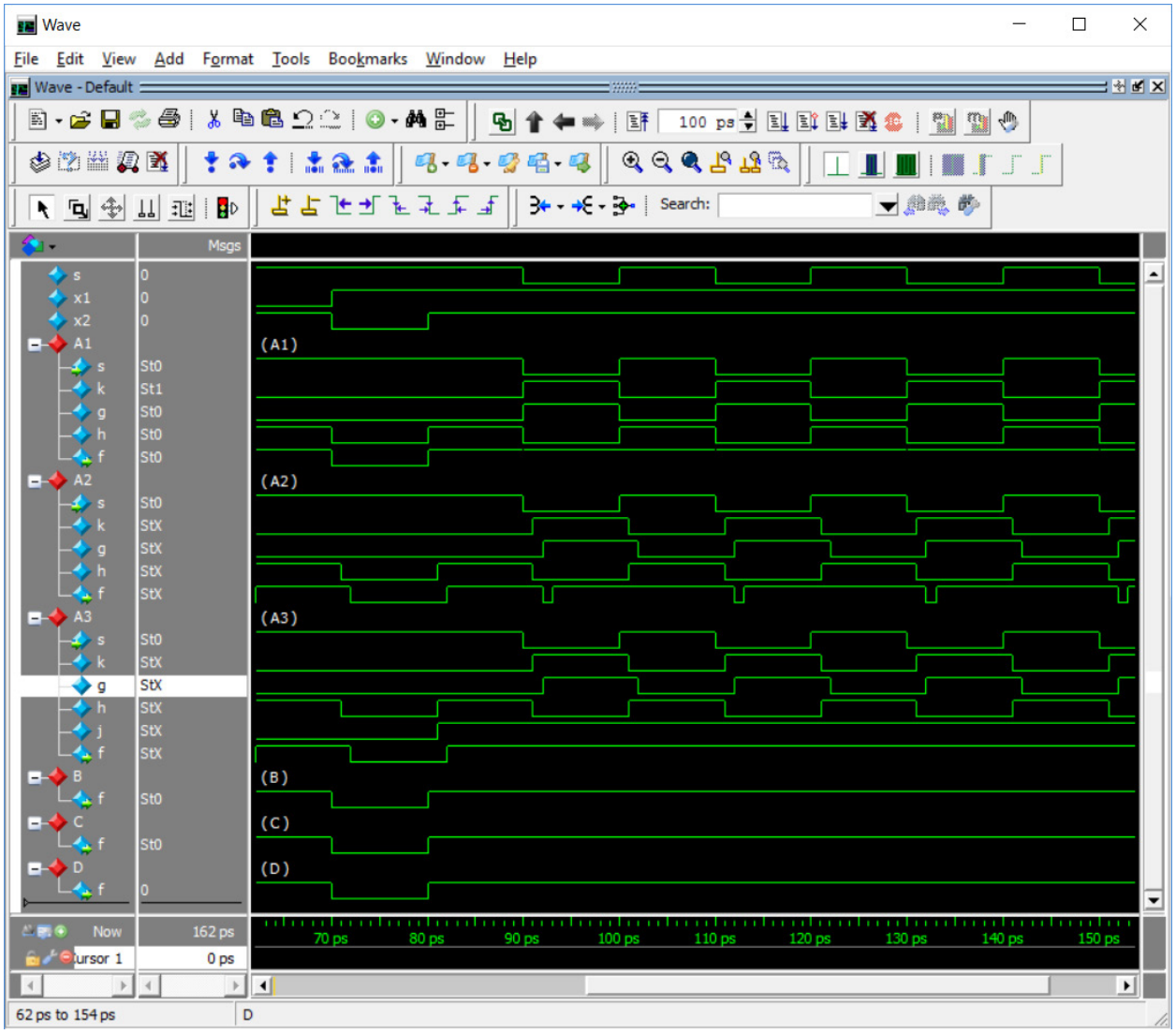
```
# Create work library
vlib work

# Compile Verilog
#     All Verilog files that are part of this design should have
#     their own "vlog" line below.
vlog "./AddingMachine.v"

# Call vsim to invoke simulator
#     Make sure the last item on the line is the name of the
#     testbench module you want to execute.
vsim -voptargs="+acc" -t 1ps -lib work testbench

# Source the wave do file
#     This should be the file that sets up the signal window for
#     the module you are testing.
do wave.do

# Set the window types
view wave
view structure
view signals

# Run the simulation
run -all

# End
```

wave.do

```
onerror {resume}
quietly WaveActivateNextPane {} 0
add wave -noupdate /testbench/hex
add wave -noupdate -expand /testbench/segments
TreeUpdate [SetDefaultTree]
WaveRestoreCursors {{Cursor 1} {40 ps} 0}
quietly wave cursor active 1
configure wave -namecolwidth 150
configure wave -valuecolwidth 100
configure wave -justifyvalue left
configure wave -signalnamewidth 1
configure wave -snapdistance 10
configure wave -datasetprefix 0
configure wave -rowmargin 4
configure wave -childrowmargin 2
configure wave -gridoffset 0
configure wave -gridperiod 300
configure wave -griddelta 40
configure wave -timeline 0
configure wave -timelineunits ps
update
WaveRestoreZoom {3 ps} {67 ps}
```

Mux2To1 example

This example shows a number of different ways of coding a 2-to-1 mux. Following the same procedure, launching ModelSim, then typing "do `Simulate.do`", you should get something approximating the following result.



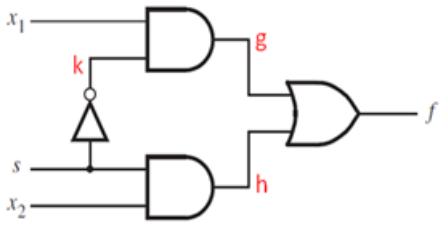
The testbed

Here's the testbed for the six different mux implementations. The first three are variations on a theme, using individual gate primitives. The next two use continuous assignments and the last uses a behavioral specification.

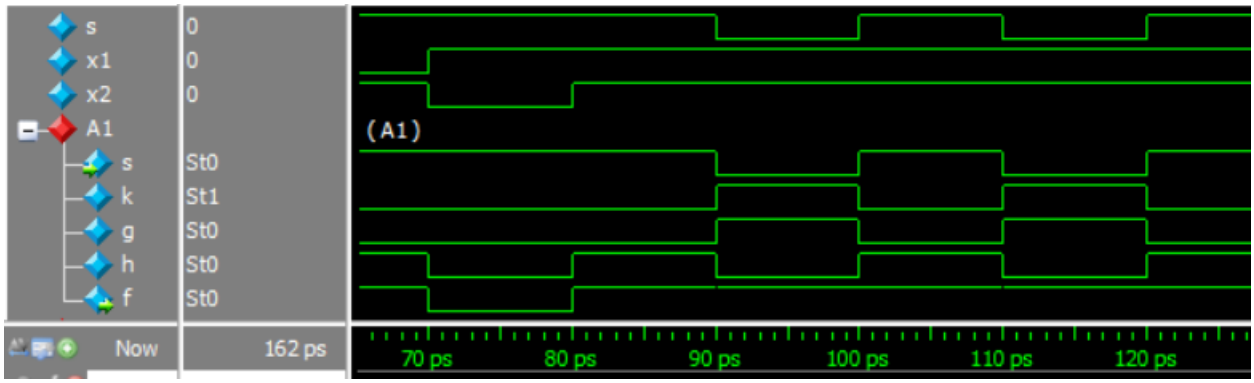
```
module testbench( );

   reg     s, x1, x2;
   wire    fA1, fA2, fA3, fB, fC, FD;
   integer i;

   Mux2To1A1 A1( x1, x2, s, fA1 );
   Mux2To1A2 A2( x1, x2, s, fA2 );
   Mux2To1A3 A3( x1, x2, s, fA3 );
   Mux2To1B  B ( x1, x2, s, fB  );
   Mux2To1C  C ( x1, x2, s, fC  );
   Mux2To1D  D ( x1, x2, s, fD  );

   initial
      begin
      // test with all possible inputs
      { s, x1, x2 } = 0;
      for ( i = 0;  i < 8;  i = i + 1 )
         #10 { s, x1, x2 } = i;

      // static-1 hazard test
      #10 { s, x1, x2 } = 'b011;
      for ( i = 1;  i < 8;  i = i + 1 )
         #10 s = ~s;
      end

endmodule
```

As you already know from lab 1, module A1, shown here, contains a static 1 hazard: When both x1 and x2 equal 1 and s switches from 1 → 0, the output f briefly glitches to 0.

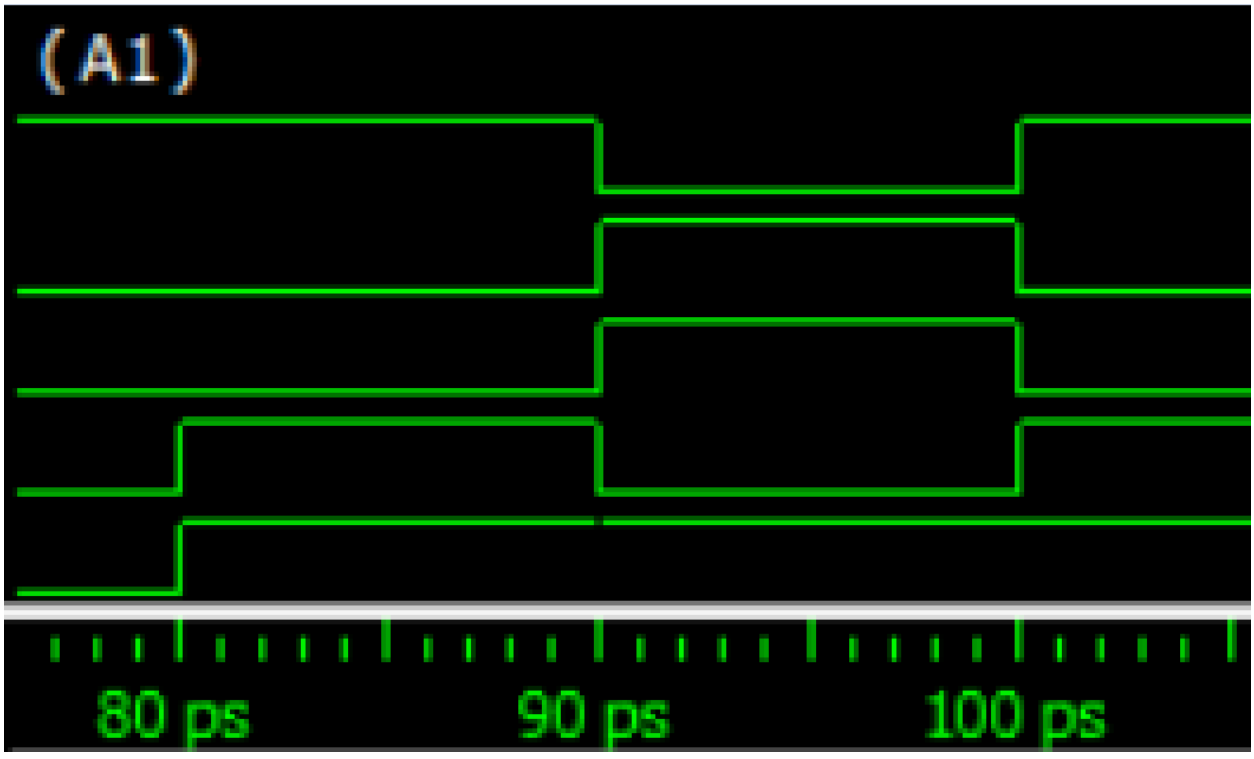
```
module Mux2To1A1(
      input  x1, x2, s,
      output f );

   wire g, h, k;
   not ( k, s );
   and ( g, k, x1 );
   and ( h, s, x2 );
   or  ( f, g, h );

endmodule
```



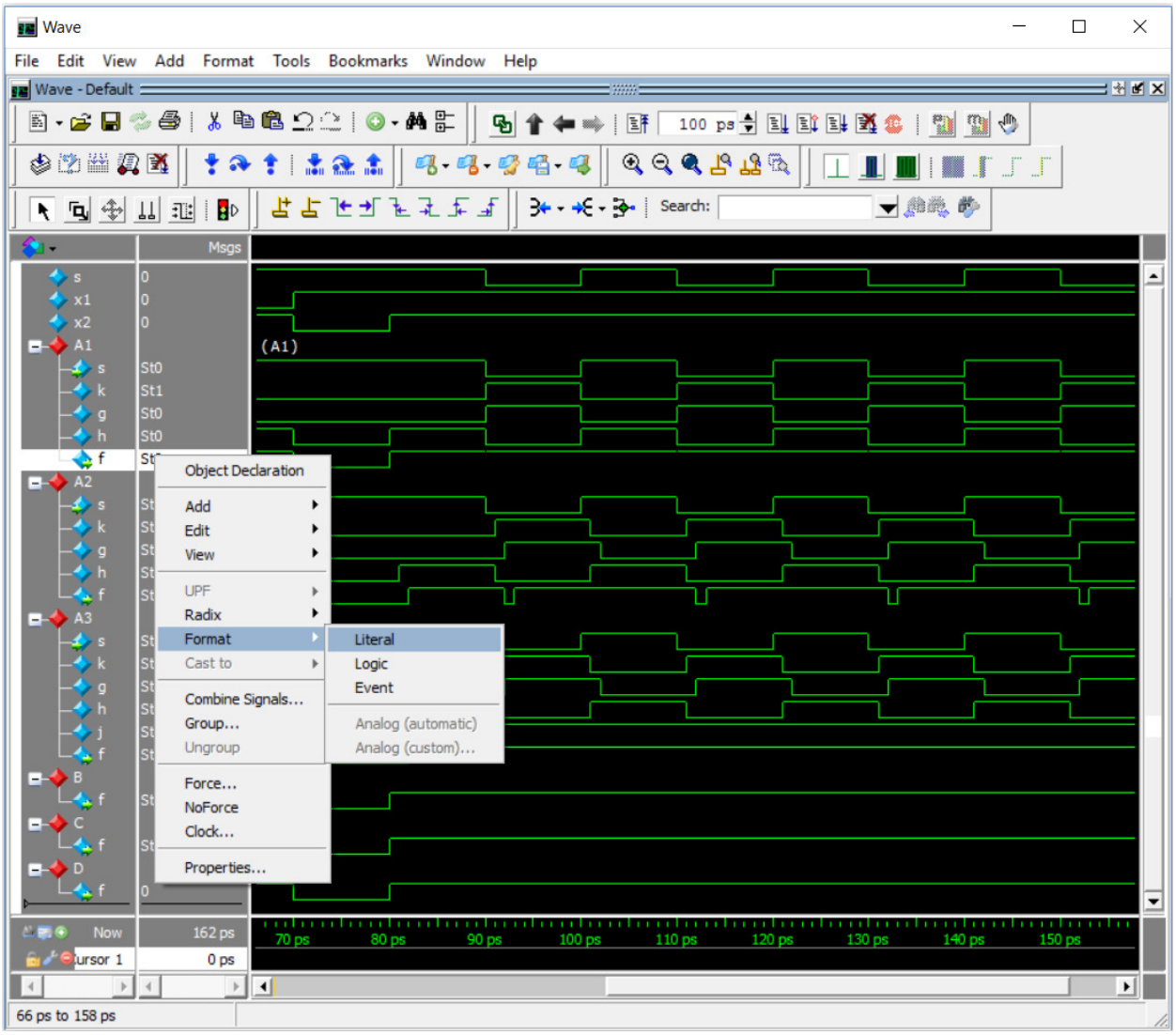
You can’t easily see the glitch in the output of A1 because it’s just one missing pixel in the green trace.



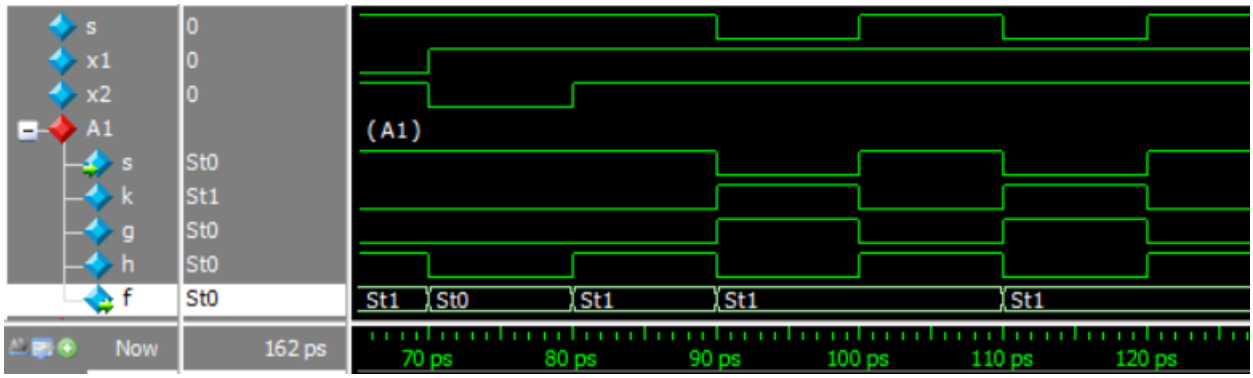
Here it is magnified.



To see what's happening more clearly, right-click on the name of the signal, f, and choose Format → Literal to make it clearer what's happening.

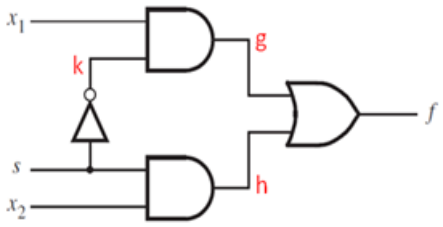


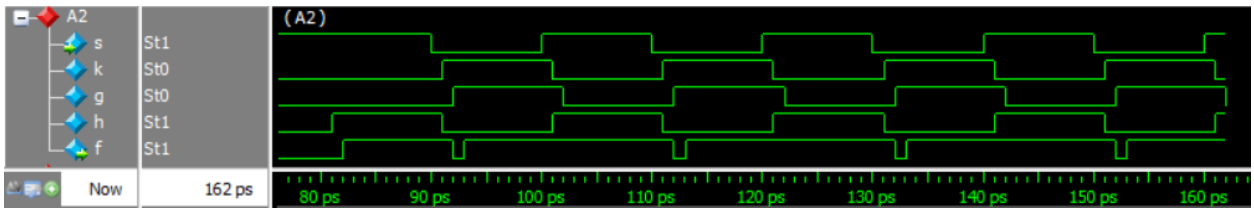
In literal format as shown here, the X shows where the output glitches at 90 ps even though both x1 and x2 are 1.



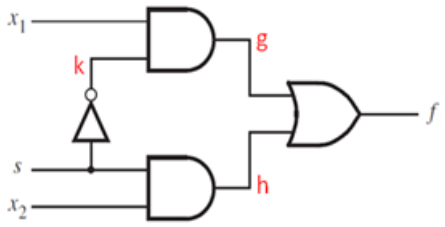
In A2, the gates are simulated with 1-tick = 1 ps delays. Again, assume x1 and x2 are 1. Now the glitch caused by the static 1 hazard is clearly visible.

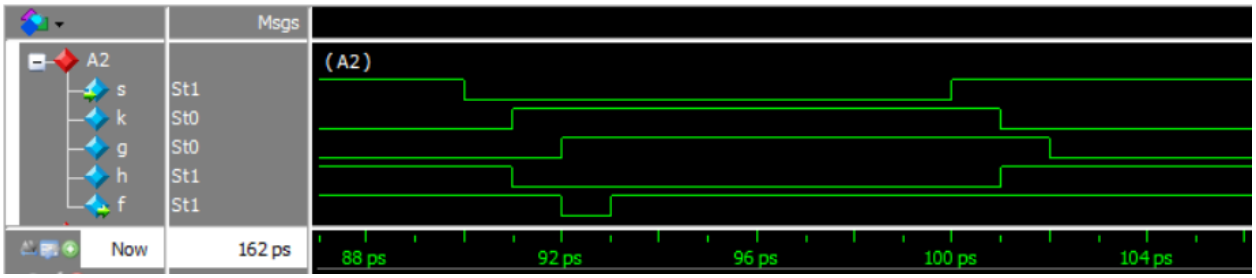
```
module Mux2To1A2(
      input  x1, x2, s,
      output f );

   wire g, h, k;
   not #1 ( k, s );
   and #1 ( g, k, x1 );
   and #1 ( h, s, x2 );
   or  #1 ( f, g, h );

endmodule
```





Zooming in, k and h switch 1 ps after s transitions from 1 to 0. g switches 2 ps after s, creating a 1 ps window when both g and h are 0, creating the glitch in f one gate delay = 1 ps later.





In A3, the hazard is eliminated by adding back the non-essential prime implicant x1 x2 shown in green in the Karnaugh map. When both x1 and x2 are 1, it doesn't matter which is selected. Adding this term back eliminates the hazard caused by switching between adjacent 1s in the Karnaugh map that aren't in the same prime implicant.

```
module Mux2To1A3(
       input  x1, x2, s,
       output f );

   wire g, h, j, k;
   not #1 ( k, s );
   and #1 ( g, k, x1 );
   and #1 ( h, s, x2 );
   and #1 ( j, x1, x2 ); // Eliminate hazard
   or  #1 ( f, g, h, j );

endmodule
```

		x1 x2			
		00	01	11	10
s	0	0	0	1	1
	1	0	1	1	0

All of the remaining mux implementations are hazard-free because the Verilog compiler detects the hazard and automatically adds in the extra term.

```
module Mux2To1B(
      input  x1, x2, s,
      output f );

   assign f = s ? x2 : x1;

endmodule


module Mux2To1C(
      input  x1, x2, s,
      output f );

   assign f = ~s & x1 | s & x2;

endmodule


module Mux2To1D(
      input x1, x2, s,
      output reg f );

   always @( * )
      if ( s )
         f = x2;
      else
         f = x1;

endmodule
```

The simulator files

Here are the contents of the files. As before, sections highlighted in yellow that may have to be fixed to match your system or will have to be changed if you copy them into your own project.

Launch_ModelSim.bat

```
C:\altera\16.1\modelsim_ase\win32aloem\modelsim.exe
```

Simulate.do

```
# Create work library
vlib work

# Compile Verilog
#     All Verilog files that are part of this design should have
#     their own "vlog" line below.
vlog "./Mux2To1.v"

# Call vsim to invoke simulator
#     Make sure the last item on the line is the name of the
#     testbench module you want to execute.
vsim -voptargs="+acc" -t 1ps -lib work testbench

# Source the wave do file
#     This should be the file that sets up the signal window for
#     the module you are testing.
do wave.do

# Set the window types
view wave
view structure
view signals

# Run the simulation
run -all

# End
```

wave.do

```
onerror {resume}
quietly WaveActivateNextPane {}
add wave -noupdate /testbench/s
add wave -noupdate /testbench/x1
add wave -noupdate /testbench/x2
add wave -noupdate -expand -group A1 /testbench/A1/s
add wave -noupdate -expand -group A1 /testbench/A1/k
add wave -noupdate -expand -group A1 /testbench/A1/g
add wave -noupdate -expand -group A1 /testbench/A1/h
add wave -noupdate -expand -group A1 /testbench/A1/f
add wave -noupdate -expand -group A2 /testbench/A2/s
add wave -noupdate -expand -group A2 /testbench/A2/k
add wave -noupdate -expand -group A2 /testbench/A2/g
add wave -noupdate -expand -group A2 /testbench/A2/h
add wave -noupdate -expand -group A2 /testbench/A2/f
add wave -noupdate -expand -group A3 /testbench/A3/s
add wave -noupdate -expand -group A3 /testbench/A3/k
add wave -noupdate -expand -group A3 /testbench/A3/g
add wave -noupdate -expand -group A3 /testbench/A3/h
add wave -noupdate -expand -group A3 /testbench/A3/j
add wave -noupdate -expand -group A3 /testbench/A3/f
add wave -noupdate -expand -group B /testbench/B/f
add wave -noupdate -expand -group C /testbench/C/f
add wave -noupdate -expand -group D /testbench/D/f
TreeUpdate [SetDefaultTree]
WaveRestoreCursors
quietly wave cursor active 0
configure wave -namecolwidth 85
configure wave -valuecolwidth 81
configure wave -justifyvalue left
configure wave -signalnamewidth 1
configure wave -snapdistance 10
configure wave -datasetprefix 0
configure wave -rowmargin 4
configure wave -childrowmargin 2
configure wave -gridoffset 0
configure wave -gridperiod 300
configure wave -griddelta 40
configure wave -timeline 0
configure wave -timelineunits ps
update
WaveRestoreZoom {84 ps} {175 ps}
```

Simple counter example

This example shows how a clocked sequential circuit might be simulated, where it's necessary to provide a clock signal.

Here are two simple variations on a counter, one with a synchronous reset and the other with an asynchronous reset. Notice the use of initial values for the reg variables.

```
module CounterA(
      input clock, reset,
      input [ 31:0 ] resetValue,
      output reg [ 31:0 ] count = 0 );

   // Synchronous reset (synchronized to the clock)
   always @( posedge clock )
      count <= reset ? resetValue : count + 1;

endmodule


module CounterB(
      input clock, reset,
      input [ 31:0 ] resetValue,
      output reg [ 31:0 ] count = 0 );

   // Asynchronous reset (not synchronized to the clock)
   always @( posedge reset, posedge clock )
      count <= reset ? resetValue : count + 1;

endmodule
```

Here is the simple wrapper for the DE1-SoC board, tying the high-order 10 bits of the counter to the LEDs, the reset value to the switches and the reset button to KEY[3]. As written here, it uses the counter with the synchronous reset. (To a human observer, there's no difference.)

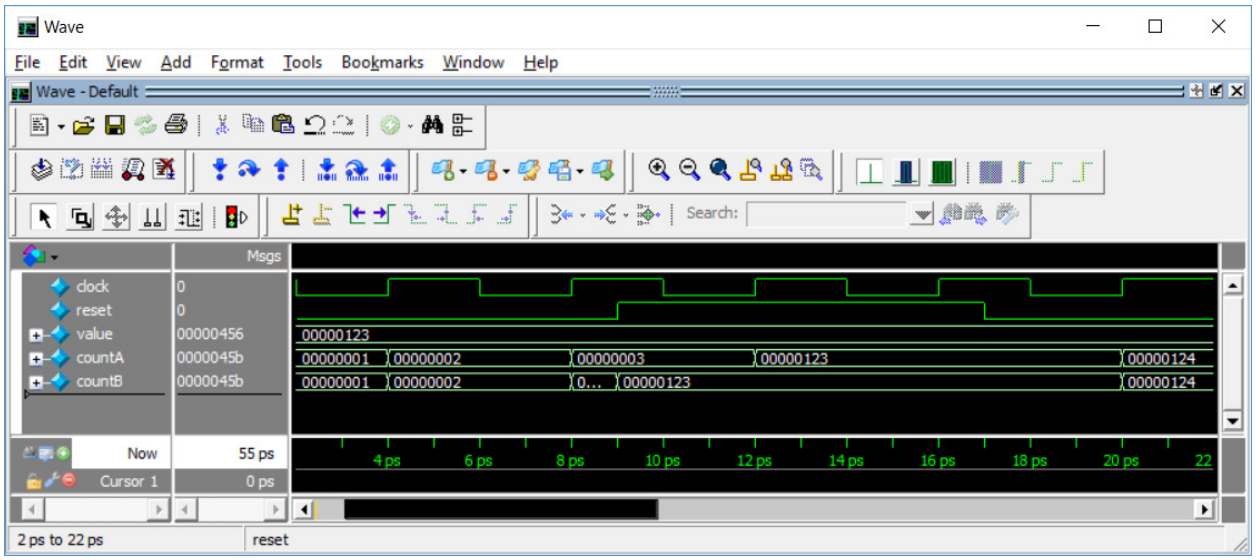
```
module SimpleCounter(
      input  CLOCK_50,
      input  [ 3:0 ] KEY,
      output [ 9:0 ] LEDR,
      input  [ 9:0 ] SW );

   wire reset = ~KEY[ 3 ];
   wire [ 31:0 ] count;

   assign LEDR = count[ 31:22 ];
   CounterA c ( CLOCK_50, reset, { SW, 22'b0 }, count );

endmodule
```

This example can also be used to demonstrate the SignalTap II logic analyzer for debugging with the DE1-SoC.

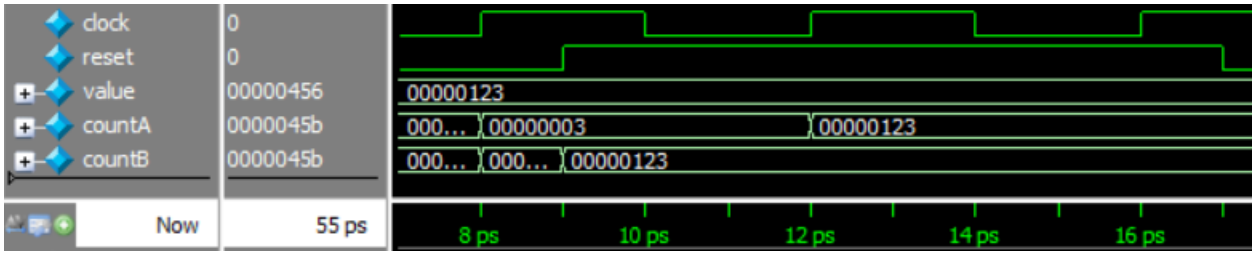
Here is the testbench. Notice the use of multiple `initial` blocks. They all start when the simulation starts, then run independently in no guaranteed order of execution. When `$stop;` is encountered inside any block, the entire simulation ends.

```
module testbench( );

   reg  clock = 1, reset = 0;
   reg  [ 31:0 ] value;
   wire [ 31:0 ] countA, countB;

   CounterA A ( clock, reset, value, countA );
   CounterB B ( clock, reset, value, countB );

   // Create a clock

   parameter clockPeriod = 4;
   initial
      forever
         #( clockPeriod / 2 ) clock <= ~clock;

   // Simulate some resets

   initial
      begin
          value = 'h123;       // @ t =  0
      # 9 reset = 1;           // @ t =  9
      # 8 reset = 0;           // @ t =  9 +  8 = 17

      // No delay at the next statement but the assignment
      // happens when the RHS is available 12 ticks later

          value <= #12 'h456; // @ t = 17 + 12 = 29

      #14 reset = 1;           // @ t = 17 + 14 = 31
      # 4 reset = 0;           // @ t = 31 +  4 = 35
      #20;                     // @ t = 35 + 20 = 55

      $stop;                   // @ t = 55
      end

endmodule
```

Launching ModelSim and typing “do Simulate.do” as before, you should get something like this.



Zooming in:



CounterA

```
always @( posedge clock )
   count <= reset ?
      resetValue : count + 1;
```

CounterB

```
always @( posedge reset, posedge clock )
   count <= reset ?
      resetValue : count + 1;
```

Notice how the resets behave differently based on how the `always` blocks have been written. In CounterA, the reset doesn’t happen until the next clock edge in CounterA because it doesn’t enter the `always` until then. This is a *synchronous* reset because it happens *with* the clock.

In CounterB, where the reset was listed in the `always` sensitivity list, the reset happens immediately. This is an *asynchronous* reset because it’s not synchronized to (i.e., made to happen at the same time as) the clock.

The simulator files

Here are the contents of the files. As before, sections highlighted in yellow that may have to be fixed to match your system or will have to be changed if you copy them into your own project.

Launch_ModelSim.bat

```
C:\altera\16.1\modelsim_ase\win32aloem\modelsim.exe
```

Simulate.do

```
# Create work library
vlib work

# Compile Verilog
#     All Verilog files that are part of this design should have
#     their own "vlog" line below.
vlog "./SimpleCounter.sv"

# Call vsim to invoke simulator
#     Make sure the last item on the line is the name of the
#     testbench module you want to execute.
vsim -voptargs="+acc" -t 1ps -lib work testbench

# Source the wave do file
#     This should be the file that sets up the signal window for
#     the module you are testing.
do wave.do

# Set the window types
view wave
view structure
view signals

# Run the simulation
run -all

# End
```

wave.do

```
onerror {resume}
quietly WaveActivateNextPane {} 0
add wave -noupdate /testbench/clock
add wave -noupdate /testbench/reset
add wave -noupdate -radix hexadecimal /testbench/value
add wave -noupdate -radix hexadecimal /testbench/countA
add wave -noupdate -radix hexadecimal /testbench/countB
TreeUpdate [SetDefaultTree]
WaveRestoreCursors {{Cursor 1} {0 ps} 0}
quietly wave cursor active 0
configure wave -namecolwidth 107
configure wave -valuecolwidth 81
configure wave -justifyvalue left
configure wave -signalnamewidth 1
configure wave -snapdistance 10
configure wave -datasetprefix 0
configure wave -rowmargin 4
configure wave -childrowmargin 2
configure wave -gridoffset 0
configure wave -gridperiod 300
configure wave -griddelta 40
configure wave -timeline 0
configure wave -timelineunits ps
update
WaveRestoreZoom {0 ps} {20 ps}
```