

# Modeling in Verilog

Andrew Morton 2013

(Sukanta Pramanik 2012, Omid Ardakanian 2011)

CS 450/650 - Computer Architecture

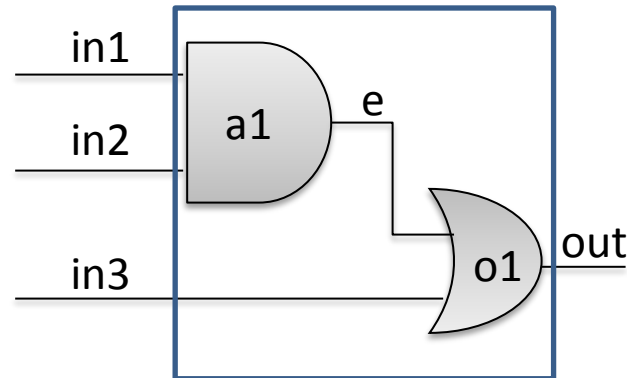
University of Waterloo

# HDL Modeling Approaches

- Modeling approaches
  - Structural **low level of abstraction**
    - could also be represented by a schematic
  - Dataflow
  - Behavioural **high level of abstraction**
    - requires tools to synthesize into circuits

# Structural Modeling

- Uses instances
  - of primitives (gates)
  - or modules
- and connectivity
  - e.g. wires
- Example:
  - and a1 (e,in1,in2)
  - or o1 (out,in3,e)



# Primitives

- predefined
  - and, or, xor, and, nor, xnor
    - 1 scalar output followed by multiple scalar inputs
    - e.g. `and a1(e, a, b, c, d)`
      - $e = a \cdot b \cdot c \cdot d$
      - identifier `a1` is optional
  - not, buf, bufif1 (tri-gate),...
    - 1 scalar output, 1 scalar input, (1 scalar control)
- user-defined
  - uses tables

# Gate-level Modeling - Example

```
module full_adder(sum, c_out, a, b, c_in);  
    output sum, c_out;  
    input a, b, c_in;  
    wire s1, c1, c2;  
  
    xor (s1, a, b);  
    and (c1, a, b);  
    xor (sum, s1, c_in);  
    and (c2, s1, c_in);  
    or (c_out, c2, c1);  
endmodule
```

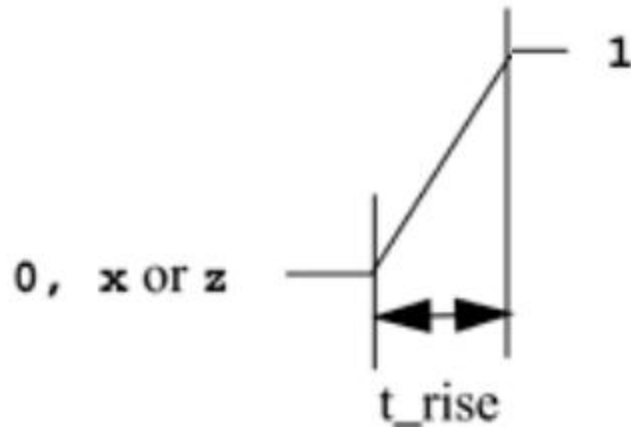
# Time Units

- can specify delays for gates and signals
- delays are dimensionless
  - can specify dimensions with compiler directive
    - e.g. ``timescale 1ns/100ps`
      - delay unit is 1ns, 0.1ns rounding precision
- delays are specified using '#'

# Transition Delays

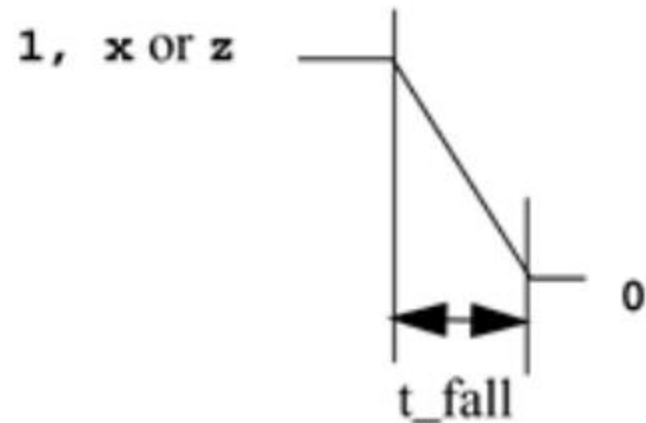
## Rise Delay

- Transition to 1 from another value (0, x, z).



## Fall Delay

- Transition to 0 from another value (1, x, z).



## Turnoff Delay

- Transition to z from another value (0, 1, x).

# Transition Delay Examples

buf b1 (out,in) - *rise = fall = turnoff = 0*

buf #(5) (out, in) - *rise = fall = turnoff = 5*

buf #(3, 5) (out, in) - *rise = 3, fall = 5, turnoff = 3*

buf #(3, 4, 5) (out, in) - *rise = 3, fall = 4, turnoff = 5*



# Delay Settings

- Min/Typical/Max
- Example:
  - and #(2:3:4, 3:4:5, 4:5:6) a(out,in1,in2)

# Simple Example

- `d = a · b + c`  

```
module simple(d, a, b, c);  
    output d;  
    input a, b, c;  
    wire w1;  
  
    and #1 g1(w1, a, b);  
    or #2 g2(d, w1, c);  
endmodule
```

# Simple Example

- `d = a·b + c`

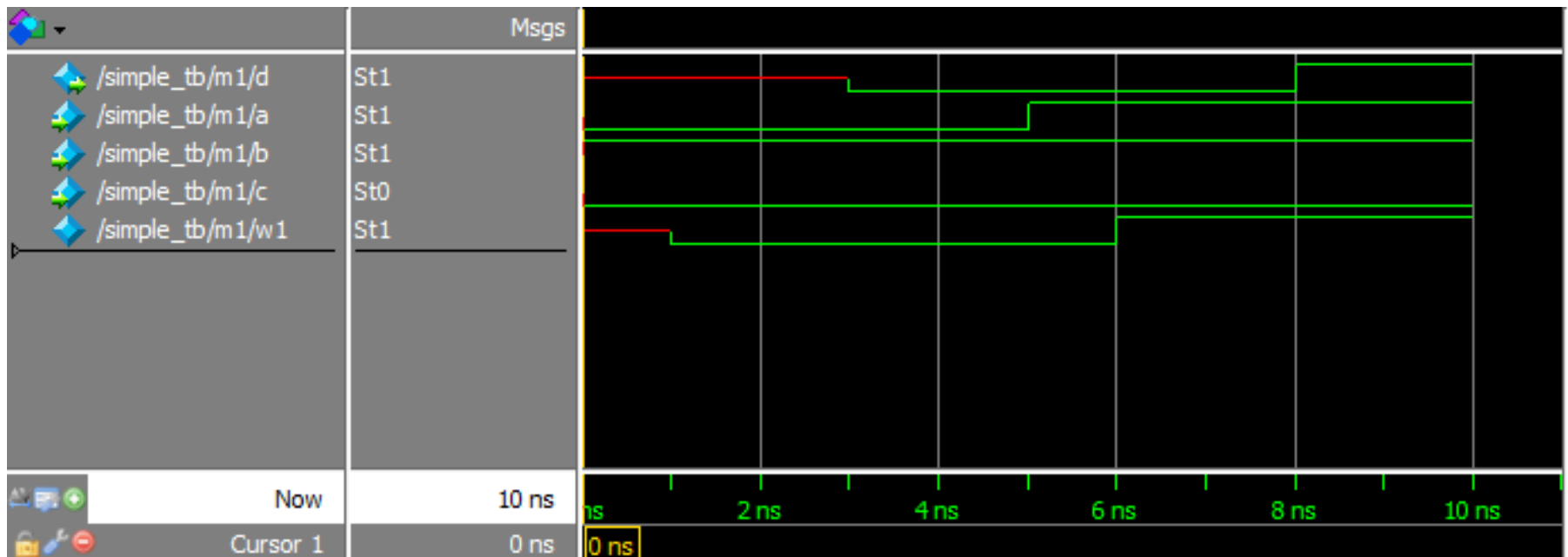
```
module simple_tb;
    reg a, b, c;
    wire d;

    simple m1(d, a, b, c);

    initial begin
        a = 1'b0; b = 1'b1; c = 1'b0;
        #5 a = 1'b1;
        #5 $stop;
    end
endmodule
```

# Simple Example

- $d = a \cdot b + c$
- #0  $a=0, b=1, c=0$
- #5  $a=1, b=1, c=0$



# Modules

- instantiation
  - modules instances must have identifiers
  - port mapping
    - implicitly in order
      - e.g. `full_adder fa0(x[0], y[0], cin, z[0], carry[1]);`
    - named (the better way)
      - e.g. `full_adder fa0(.a(x[0]), .b(y[0]), .cin(cin), .sum(z[0]), .cout(carry[1]));`
- primitive and module instantiations are concurrent
  - think in terms of circuit schematics, not a software program

# Dataflow Modelling

- Continuous Assignment
  - drives a value onto a net
  - e.g. `assign w1 = a && b;`
- Operators
  - logical: ! (not), && (and), || (or)
  - arithmetic: +, -, \*, /, %
  - relational: <, >, <=, >=
  - equality: ==, !=
  - bit-wise: ~ (negate), & (and), | (or), ^ (xor), ^~ (xnor)
  - shift: <<, >>
  - concatenation: {, }
  - replication: {n{a}} – a is replicated n times
  - conditional: ? :

# Dataflow Modelling

- Example:

- assign out = in1 & in2;
- assign #10 out = in1 & in2;
- assign out = sel ? in1 : in2;
- assign {b[3:0],b[7:4]} = a[7:0];
  - nibble-swap
- assign b[15:0] =  
{{8{a[7]}},a[7:0]};
  - sign extend from 8 to 16 bits

# Dataflow Modeling - Example

```
module adder_4_bit(sum, c_out, a, b, c_in);  
    output [3:0] sum;  
    output c_out;  
    input [3:0] a, b;  
    input c_in;  
  
    assign {c_out, sum} = a + b + c_in;  
endmodule
```



# Behavioural Modeling - always vs. initial

- procedural statements start with a control construct
  - initial, or
  - always
- sequential blocks
  - begin, end
- parallel blocks
  - fork, join
  - contents run concurrently

# Behavioural Modeling - always vs. initial

## initial

- Starts at  $t=0$ , runs once

initial

```
begin
  #1 $display ("Line 1");
  $display ("Line 2");
end
```

Sequential Block

## always

- Starts at  $t=0$ , runs continuously

always

```
begin
  #1 $display ("Line 1");
  $display ("Line 2");
end
```

# Behavioural Modeling - always vs. initial

## initial

- Starts at  $t=0$ , runs once

initial

```
fork
  #1 $display ("Line 1");
  $display ("Line 2");
join
```

Parallel Block

## always

- Starts at  $t=0$ , runs continuously

# Behavioural Modeling - always vs. initial

## **initial**

- Starts at t=0, runs once

## **always**

- Starts at t=0, runs continuously

initial

clock = 1'b0;

always

#10 clock = ~clock;

initial

#1000 \$finish;

- initial and always blocks execute concurrently

# Behavioural Modeling – Assignment

- Blocking assignment (=)
  - Execute in the order they are specified
  - Block execution of statements that follow in a sequential block

```
initial begin
  i = 3;
  j = 4;
  #10 a = i + j;
  i = a + 5;
  j = a;
end
```

i = 12, j = 7

# Behavioural Modeling – Assignment

- Blocking assignment (=)
  - Execute in the order they are specified
  - Block execution of statements that follow in a sequential block

```
initial begin
  i = 3;
  j = 4;
  #10 a = i + j;
  i = a + 5;
  j = a;
end
```

$i = 12, j = 7$

```
initial begin
  i = 3;
  j = 4;
  fork
    i = j;
    j = i;
  join
end
```

race condition

# Behavioural Modeling – Assignment

- Blocking assignment (=)
  - Execute in the order they are specified
  - Block execution of statements that follow in a sequential block

```
initial begin
  i = 3;
  j = 4;
  #10 a = i + j;
  i = a + 5;
  j = a;
end
```

i = 12, j = 7

```
initial begin
  i = 3;
  j = 4;
  fork
    i = j;
    j = i;
  join
end
```

race condition

```
initial begin
  i = 3;
  j = 4;
  fork
    i = #1 j;
    j = #1 i;
  join
end
```

i = 4, j = 3

# Behavioural Modeling – Assignment

- Non-blocking assignments ( $\leq$ )
  - Allow scheduling of the following assignments

```
initial begin
  i = 3;
  j = 4;
  begin
    i = #1 j;
    j = #1 i;
  end
end
```

i = 4, j = 4

```
initial begin
  i = 3;
  j = 4;
  begin
    i <= #1 j;
    j <= #1 i;
  end
end
```

i = 4, j = 3



# Behavioural Modeling – Timing Control

- Event-based
  - always @(sensitivity list)
    - runs whenever a signal in the sensitivity list changes
  - e.g.

```
always @(a, b) begin
    x <= a & b;
    y <= a | b;
end
```
  - **rule of thumb**: use non-blocking assignment in always blocks

# always blocks

- use continuous assignment for combinational logic
- use always event blocks for sequential logic
  - the *lhs* of assignments should be a variable
    - usually `reg`
    - could also be `integer`, `time`, `real`

# Procedural Select

- `if ( expression ) statement else statement`

```
module mux_2x1(d, i0, i1, sel);  
    output reg d;  
    input i0, i1, sel;  
  
    always @(*) // always @(i0, i1, sel)  
    if (sel == 1'b0)  
        d <= i0;  
    else  
        d <= i1;  
endmodule
```

# Procedural Select

- `case ( expression ) {expr:stmt}+ {default:stmt}? endcase`

```
module mux_2x1(d, i0, i1, sel);  
    output reg d;  
    input i0, i1, sel;  
  
    always @(*) case(sel)  
        1'b0: d <= i0;  
        1'b1: d <= i1;  
        default: d <= 1'b0;  
    endcase  
endmodule
```

# Synchronous Designs

- use a clock

- in testbench

- ```
initial clk = 1'b0;
```

- ```
always #10 clk = ~clk;
```

- or

- ```
initial begin
```

- ```
    clk = 0; forever #10 clk = ~clk;
```

- ```
end
```

# Synchronous Design

- sensitivity list includes an edge event
  - posedge *signal* or negedge *signal*

```
always @(posedge clk)
```
- asynchronous reset?
  - synthesis tools can't combine edge events and level sensitive events
  - use a synchronous reset

```
always @(posedge clk or posedge reset)
```

# Synchronous with Reset

```
module counter(val, en, clk, reset);  
    output reg [7:0] val;  
    input en, clk, reset;  
  
    always @(posedge clk, posedge reset)  
        if(reset)  
            val <= 0;  
        else  
            if(en) val <= val + 1;  
            else val <= val;  
endmodule
```

# Constants

- parameter keyword

```
parameter n = 16, lsb = 0,  
        msb = 15;
```

```
parameter [1:0] // functional units  
        fs_null = 2'b00, // no unit  
        fs_alu   = 2'b01, // alu  
        fs_bra   = 2'b10, // branch unit  
        fs_ldst  = 2'b11; // ld/st unit
```



# Arrays

- arrays of scalar or vector wires or regs

```
reg [15:0] a; // 16-bit register
```

```
reg [15:0] b [0:7]; // eight 16-bit  
registers
```

- indexing

```
input [2:0] sel;
```

```
output [15:0] x;
```

```
assign x = b[sel];
```

# Other Statements

- `wait ( expression ) statement`
  - simulate waits for *expression* to become true before executing *statement*
  - not synthesizable

```
wait ( (mem_wr_addr==16'hffff) &&  
      (mem_wr_data==16'h0001) )  
      $stop;
```

# Other Statements

- generate

- instantiates multiple copies of a module

```
genvar i;
```

```
generate
```

```
    for(i=0; i<4; i=i+1)
```

```
        full_adder fa(x[i], y[i],
```

```
            c_in[i], sum[i], c_out[i+1]));
```

```
endgenerate
```

# System Tasks

- `$display`, `$write`
  - output to console (`$display` appends newline)
- `$monitor`
  - display changes to signal list
- `$fopen`, `$fwrite`, `$fclose`
  - file I/O
- `$readmemb`, `$readmemh`
  - read data into a memory array

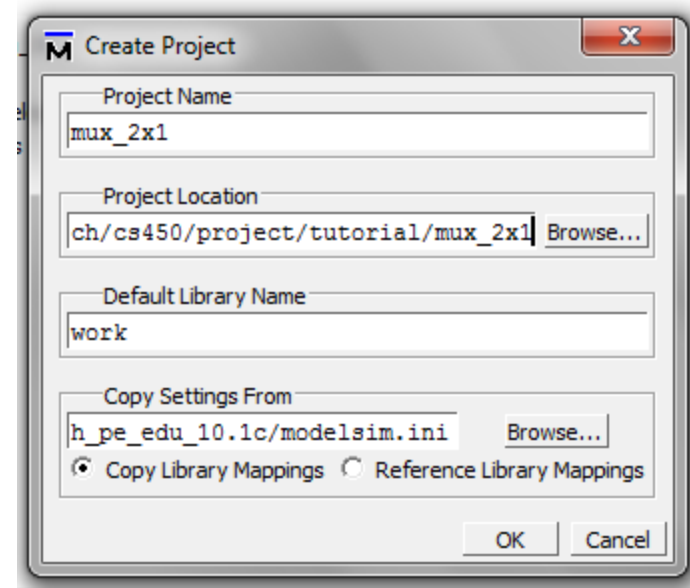
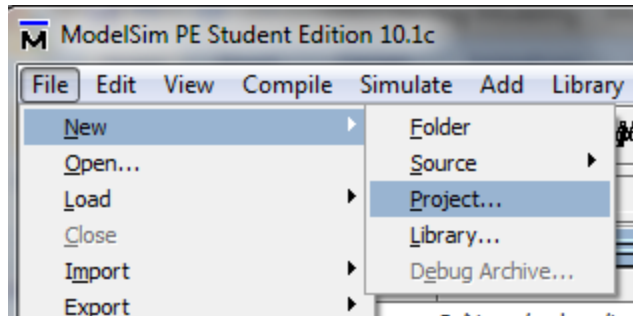
# System Tasks

- \$time
  - current simulation time
- \$stop
  - suspend execution
- \$finish
  - end execution

# ModelSim

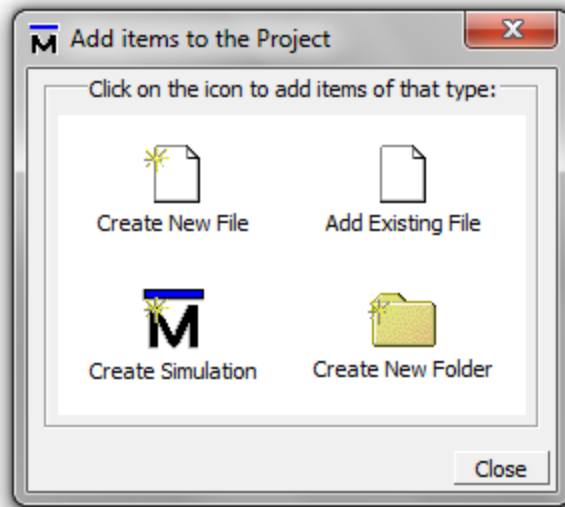
- Workflow
  - create project
  - add files
  - compile
  - simulate

# Create Project

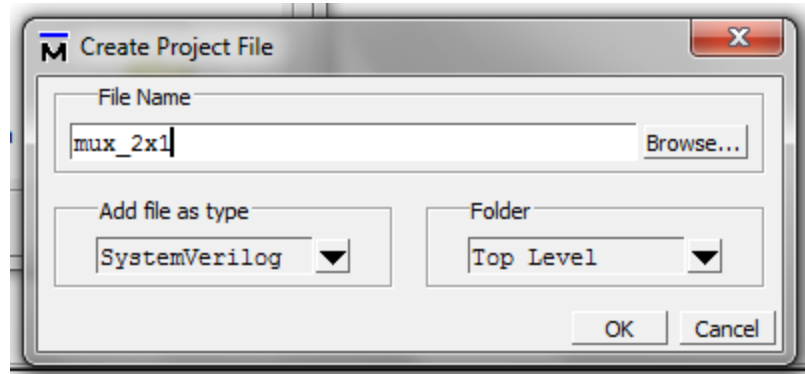


# Add Items

- Can import existing files or create new

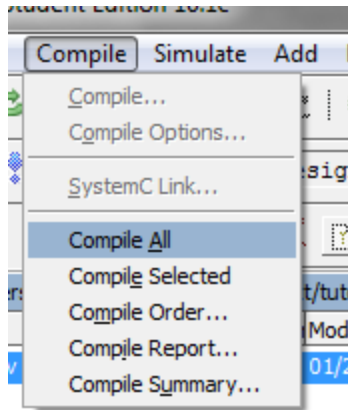


- files should be of type SystemVerilog





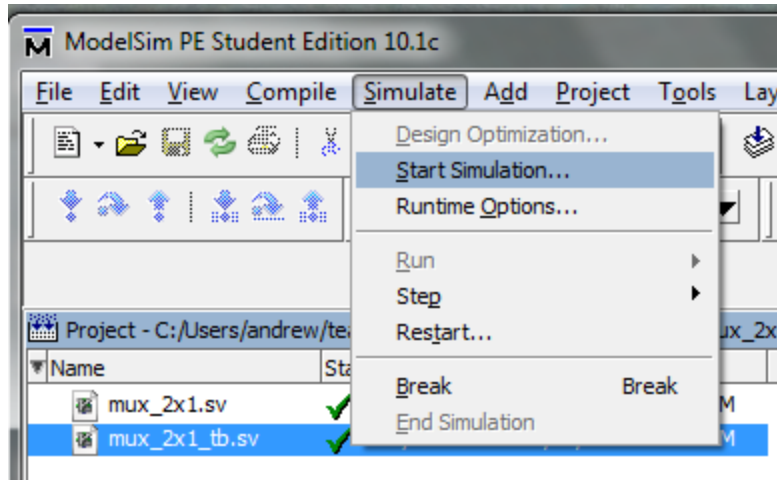
# Compile

A screenshot of the 'Transcript' window in ModelSim. It displays the output of a compilation process. The text is as follows:

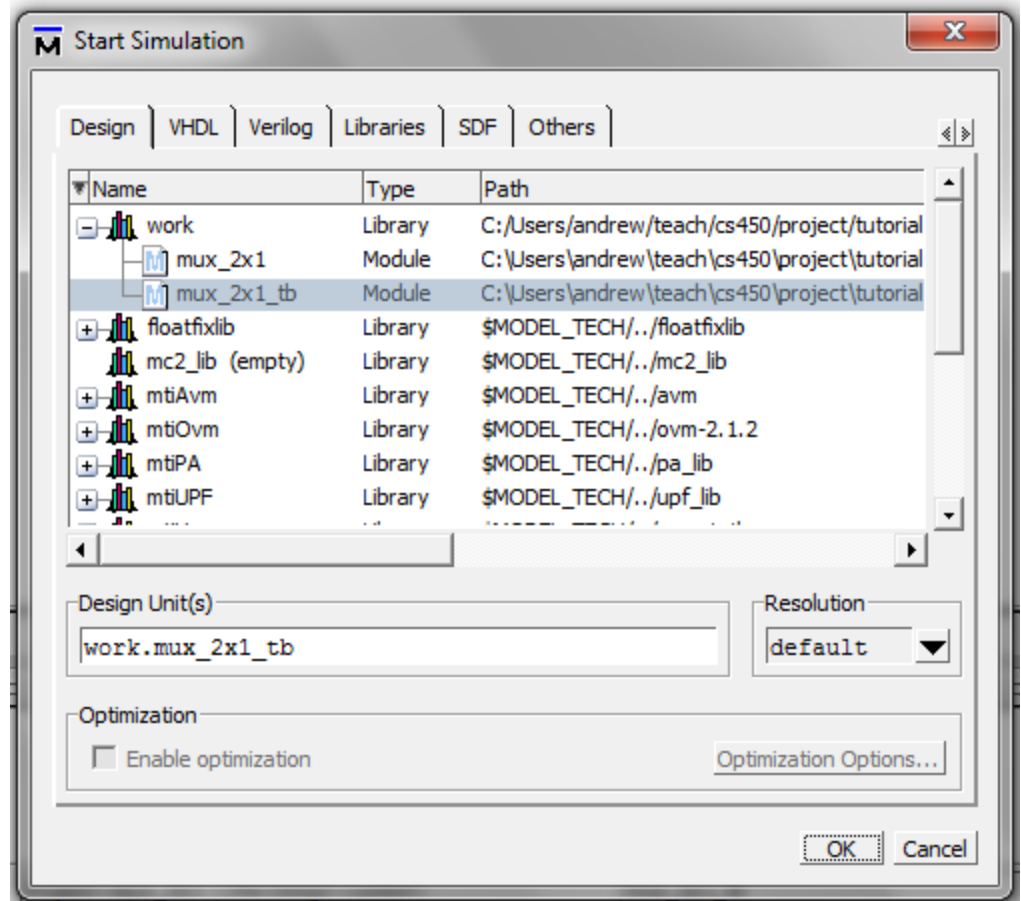
```
ModelSim>  
# Compile of mux_2x1.sv was successful.  
# Compile of mux_2x1_tb.sv failed with 1 errors.  
# 2 compiles, 1 failed with 1 error.
```

- check compile status in Transcript window
- double-click error to see details

# Simulate

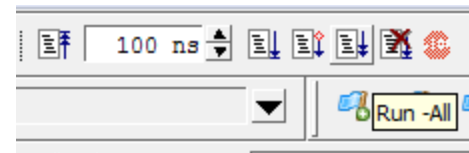
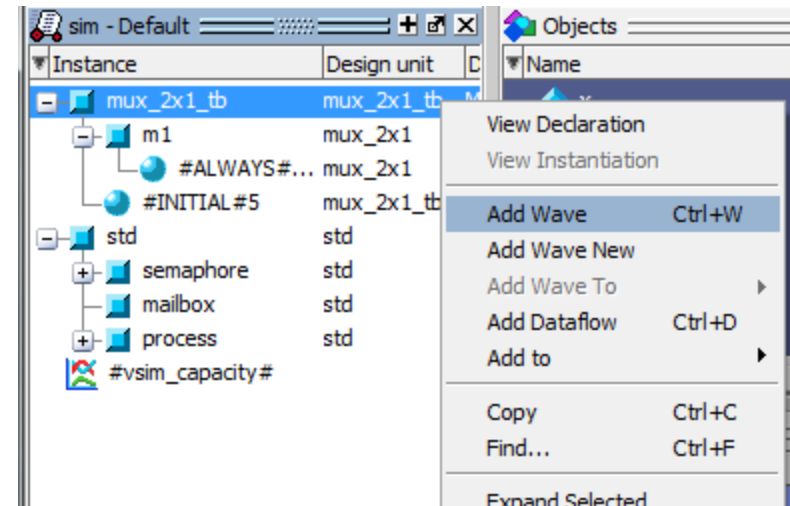


- select testbench from work directory

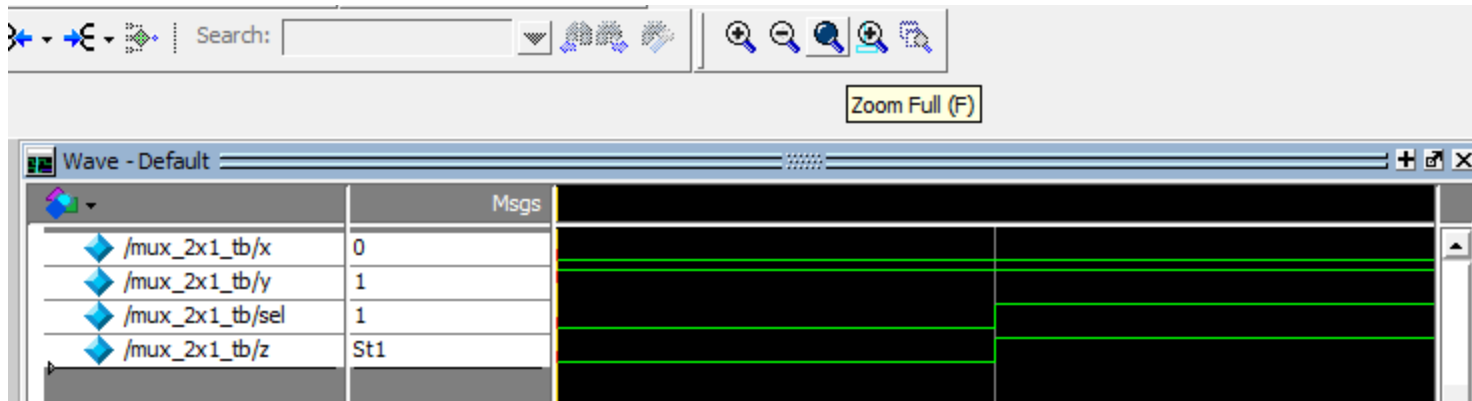


# Simulate

- add waves
  - can expand instances
  - can add individual or add all
- run options:
  - reset
  - for specified time
  - continue
  - run all



# Simulate



- zoom interface
  - in (I)
  - out (O)
  - full (F)