Simulation

Nachiket Kapre
nachiket@uwaterloo.ca
UNIVERSITY OF WATERLOO

Outline

- Understanding why order matters in parallel processing
- Verilog Execution Model
    - Model of Event-Driven Simulation
- Simulation Tools: Xsim
- Timing Diagrams (Waveforms)

Simple threaded model of parallelism in C

```c
// gcc -std=c99 poly-parallel.c -pthread
#include <pthread.h>
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#define NUM_THREADS 16
int *a, *b, *c, *x, *y;

void *poly(void *id) {
 int i = (int)(intptr_t)id; //
 y[i]=a[i]*x[i]*x[i]+b[i]*x[i]+c[i];
 printf("Thread %d\n",i); return 0;
}

int main(int argc, char *argv[]) {
 a=malloc(sizeof(int)*NUM_THREADS);
 b=malloc(sizeof(int)*NUM_THREADS);
 c=malloc(sizeof(int)*NUM_THREADS);
 x=malloc(sizeof(int)*NUM_THREADS);
 y=malloc(sizeof(int)*NUM_THREADS);

 pthread_t threads[NUM_THREADS];
 for(int id=0;id<NUM_THREADS;id++) {
  //
  pthread_create(&(threads[id]), NULL, poly,
    (void *)(intptr_t)id);
  }

 for(int id=0;id<NUM_THREADS;id++)
  pthread_join(threads[id], NULL); //
}
```

- Pthreads are a unit of parallel work
- Code execute sequentially inside a Pthread
- Different pthreads execute in parallel at **forks**
- Order of execution not known upfront $\rightarrow$ OS/Runtime/Processor
- Explicit synchronization using **joins**

Simple threaded model of parallelism in C

```c
// gcc -std=c99 poly-parallel.c -pthread
#include <pthread.h>
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#define NUM_THREADS 16
int *a, *b, *c, *x, *y;

void *poly(void *id) {
 int i = (int)(intptr_t)id; //
 y[i]=a[i]*x[i]*x[i]+b[i]*x[i]+c[i];
 printf("Thread %d\n",i); return 0;
}

int main(int argc, char *argv[]) {
 a=malloc(sizeof(int)*NUM_THREADS);
 b=m                          UM_THREADS);
                                  EADS);
                                   DS);
                                   DS);

 pthread_                 THREADS];
 for(int id=0    d<NUM_THREADS;id++) {
  //
  pthread_create(&(threads[id]), NULL, poly,
    (void *)(intptr_t)id);
  }

 for(int id=0;id<NUM_THREADS;id++)
  pthread_join(threads[id], NULL); //
}
```

Local work

Create parallel threads

Join parallel threads

- Pthreads are a unit of parallel work
- Code execute sequentially inside a Pthread
- Different pthreads execute in parallel at **forks**
- Order of execution not known upfront → OS/Runtime/Processor
- Explicit synchronization using **joins**

Parallel Event Scheduling (C Program View)

```
a = 0; b = 1;
```

```
void thread_1(int *a, int *b) {
  *a = *b + 1;
}
```

```
void thread_2(int *a, int *c) {
  *c = *a + 1;
}
```

- Two threads of C code, use same input data a and b. What are values of a and b in the end?
- Order of thread execution may result in different results
  - thread_1 runs before thread_2
  - thread_2 runs before thread_1
- Hardware circuits do not work like this! Verilog simulations should not depend on order of evaluation of concurrent statements, or always@ blocks

Parallel Event Scheduling (C Program View)

```
a = 0; b = 1;
```

```
void thread_1(int *a, int *b) {
  *a = *b + 1;
}
```

a=2

```
void thread_2(int *a, int *c) {
  *c = *a + 1;
}
```

c=3

- Two threads of C code, use same input data a and b. What are values of a and b in the end?
- Order of thread execution may result in different results
  - thread_1 runs before thread_2
  - thread_2 runs before thread_1
- Hardware circuits do not work like this! Verilog simulations should not depend on order of evaluation of concurrent statements, or always@ blocks

Parallel Event Scheduling (C Program View)

```
a = 0; b = 1;
```

```
void thread_1(int *a, int *b) {
  *a = *b + 1;
}
```

a=2

```
void thread_2(int *a, int *c) {
  *c = *a + 1;
}
```

c=1

- Two threads of C code, use same input data a and b. What are values of a and b in the end?
- Order of thread execution may result in different results
  - thread_1 runs before thread_2
  - thread_2 runs before thread_1
- Hardware circuits do not work like this! Verilog simulations should not depend on order of evaluation of concurrent statements, or always@ blocks

Parallel Event Scheduling (Verilog Program View)

```
reg [31:0] a=32'bX;
wire [31:0] b=32'h0001; //
```

```
always @(b) begin : thread_1
 a <= b + 1; //
end
```

```
always @(a) begin : thread_2
 c <= a + 1; //
end
```

- Verilog simulation results should not depend on order of firing of blocks →
  - need a new concept of time (no advance in physical time), and
  - possibility of multiple evaluation steps at given physical time until stabilization
  - thread_1 and thread_2 run concurrently →
    - a=2 after one step, c=X after one step
    - c=3 after second step (only thread_2 is evaluated again).

Parallel Event Scheduling (Verilog Program View)

```
reg [31:0] a=32'bX;
wire [31:0] b=32'h0001; //
```

```
always @(b) begin : thread_1
 a <= b + 1; //
end
```

a=2

```
always @(a) begin : thread_2
 c <= a + 1; //
end
```

c=X

- Verilog simulation results should not depend on order of firing of blocks →
  - need a new concept of time (no advance in physical time), and
  - possibility of multiple evaluation steps at given physical time until stabilization
  - thread_1 and thread_2 run concurrently →
    - a=2 after one step, c=X after one step
    - c=3 after second step (only thread_2 is evaluated again).

Parallel Event Scheduling (Verilog Program View)

```
reg [31:0] a=32'bX;
wire [31:0] b=32'h0001; //
```

```
always @(b) begin : thread_1
 a <= b + 1; //
end
```

a=2

```
always @(a) begin : thread_2
 c <= a + 1; //
end
```

c=3

- Verilog simulation results should not depend on order of firing of blocks →
  - need a new concept of time (no advance in physical time), and
  - possibility of multiple evaluation steps at given physical time until stabilization
  - thread_1 and thread_2 run concurrently →
    - a=2 after one step, c=X after one step
    - c=3 after second step (only thread_2 is evaluated again).

Step-by-Step Execution (inside Verilog simulator)

```
always @(b) begin : thread_1
 a_new = b + 1;
end
```

a=X, a_new=2

```
 a = a_new;
```

```
always @(a) begin : thread_2
 c_new = a + 1;
end
```

c=X, c_new=X

```
 c = c_new;
```

```
always @(b) begin : thread_1
 a_new = b + 1;
end


 a = a_new;
```

```
always @(a) begin : thread_2
 c_new = a + 1;
end


 c = c_new;
```

Step-by-Step Execution (inside Verilog simulator)

```
always @(b) begin : thread_1
 a_new = b + 1;
end
```

a=X, a_new=2

```
 a = a_new;
```

a=2, a_new=2

```
always @(a) begin : thread_2
 c_new = a + 1;
end
```

c=X, c_new=X

```
 c = c_new;
```

c=X, c_new=X

```
always @(b) begin : thread_1
 a_new = b + 1;
end


 a = a_new;
```

```
always @(a) begin : thread_2
 c_new = a + 1;
end


 c = c_new;
```

Step-by-Step Execution (inside Verilog simulator)

```
always @(b) begin : thread_1
 a_new = b + 1;
end
```

a=X, a_new=2

```
 a = a_new;
```

a=2, a_new=2

```
always @(a) begin : thread_2
 c_new = a + 1;
end
```

c=X, c_new=X

```
 c = c_new;
```

c=X, c_new=X

```
always @(b) begin : thread_1
 a_new = b + 1;
end
```

Unchanged

```
 a = a_new;
```

```
always @(a) begin : thread_2
 c_new = a + 1;
end
```

c=X, c_new=3

```
 c = c_new;
```

Step-by-Step Execution (inside Verilog simulator)

```
always @(b) begin : thread_1
 a_new = b + 1;
end
```

a=X, a_new=2

```
 a = a_new;
```

a=2, a_new=2

```
always @(a) begin : thread_2
 c_new = a + 1;
end
```

c=X, c_new=X

```
 c = c_new;
```

c=X, c_new=X

```
always @(b) begin : thread_1
 a_new = b + 1;
end
```

Unchanged

```
 a = a_new;
```

Unchanged

```
always @(a) begin : thread_2
 c_new = a + 1;
end
```

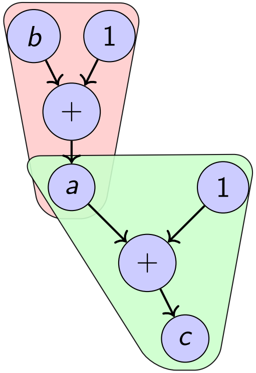
c=X, c_new=3

```
 c = c_new;
```

c=3, c_new=3

Parallel Event Scheduling (Circuit-View)

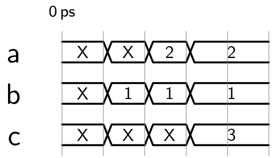
```
reg [31:0] a=32'bX;
wire [31:0] b=32'h0001; //

always @(b) begin : thread_1
 a <= b + 1; //
end

always @(a) begin : thread_2
 c <= a + 1; //
end
```



- Hardware circuits enforce dependencies between threads
- Only interested in *steady-state* values
- Verilog models the instantaneous delay of evaluation as a step
- Step is an artificial unit of time.

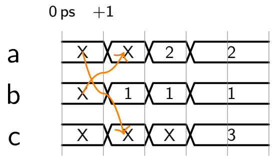
Timing Diagram/Waveform 101

```
module example32 (
  output reg [31:0] c
 );

 reg [31:0] a=32'bX;
 wire [31:0] b=32'h0001; //

 always @(b) begin : thread_1
  a <= b + 1; //
 end

 always @(a) begin : thread_2
  c <= a + 1; //
 end

endmodule
```



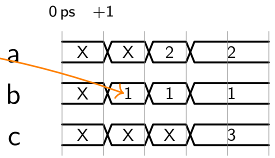
Timing Diagram/Waveform 101

```
module example32 (
  output reg [31:0] c
 );

 reg [31:0] a=32'bX;
 wire [31:0] b=32'h0001; //

 always @(b) begin : thread_1
  a <= b + 1; //
 end

 always @(a) begin : thread_2
  c <= a + 1; //
 end

endmodule
```



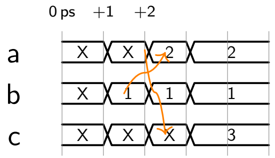
Timing Diagram/Waveform 101

```
module example32 (
  output reg [31:0] c
 );

 reg [31:0] a=32'bX;
 wire [31:0] b=32'h0001; //

 always @(b) begin : thread_1
  a <= b + 1; //
 end

 always @(a) begin : thread_2
  c <= a + 1; //
 end

endmodule
```



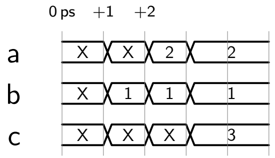
Timing Diagram/Waveform 101

```
module example32 (
  output reg [31:0] c
 );

 reg [31:0] a=32'bX;
 wire [31:0] b=32'h0001; //

 always @(b) begin : thread_1
  a <= b + 1; //
 end

 always @(a) begin : thread_2
  c <= a + 1; //
 end

endmodule
```



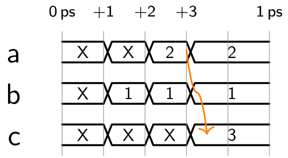
Timing Diagram/Waveform 101

```
module example32 (
  output reg [31:0] c
 );

 reg [31:0] a=32'bX;
 wire [31:0] b=32'h0001; //

 always @(b) begin : thread_1
  a <= b + 1; //
 end

 always @(a) begin : thread_2
  c <= a + 1; //
 end

endmodule
```



Timing Diagram/Waveform 101

```
module example32 (
  output reg [31:0] c
 );

 reg [31:0] a=32'bX;
 wire [31:0] b=32'h0001; //

 always @(b) begin : thread_1
  a <= b + 1; //
 end

 always @(a) begin : thread_2
  c <= a + 1; //
 end

endmodule
```



Verilog events

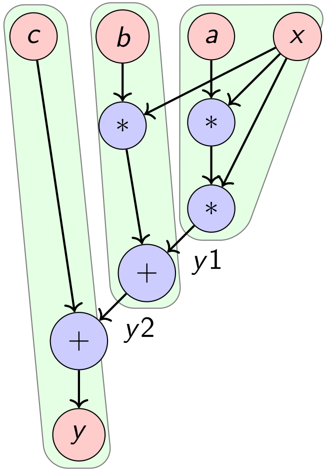
- Hardware runs in parallel
- Model instantaneous propagation of signals through a circuit $\rightarrow$ event-driven simulation model
  - (1) blocking assignments + RHS of non-blocking assignments + $display statements
  - (2) LHS of non-blocking assignments
  - Re-evaluate (1) based on results of (2)
  - Advance time when nothing left to evaluate
- To create illusion of parallel behavior, Verilog uses **events**
  - RHS of non-blocking assignments only visible

Rules when using Verilog Events

- Check if gate (always@ block) inputs have changed
  - If no input changed, output will be unchanged
  - If an input changed (and is on sensitivity list), evaluate again
- Keep doing this in a loop, until values stabilize
- If they don't stabilize you have a combinational loop in your design
  - Weakly analogous to a **infinite loop** (sometimes manifest as **stack overflow**) error in software
- Actual time increments only when all events at given point in time are processed

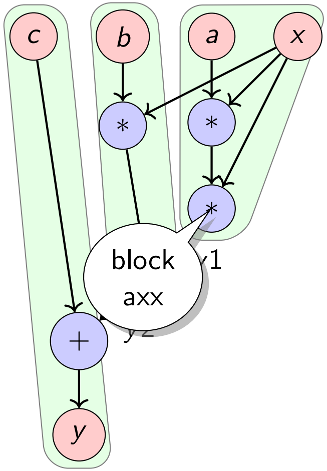
Simulation Example

```
always @(a or x) begin: axx
 $display("y1<=axx fired");
 y1 <= a * x * x; //
end

always @(y1 or b or x) begin: bx
 $display("y2<=y1+bx fired");
 y2 <= y1 + b * x; //
end

always @(y2 or c) begin: plusc
 $display("y<=y2+c fired");
 y <= y2 + c; //
end
```



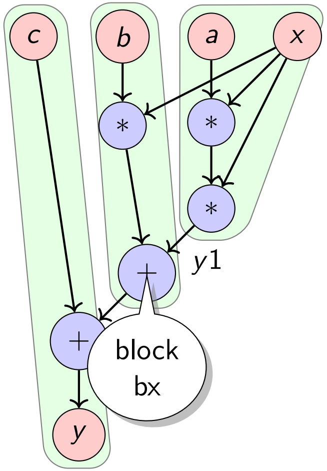
Simulation Example

```
always @(a or x) begin: axx
 $display("y1<=axx fired");
 y1 <= a * x * x; //
end

always @(y1 or b or x) begin: bx
 $display("y2<=y1+bx fired");
 y2 <= y1 + b * x; //
end

always @(y2 or c) begin: plusc
 $display("y<=y2+c fired");
 y <= y2 + c; //
end
```



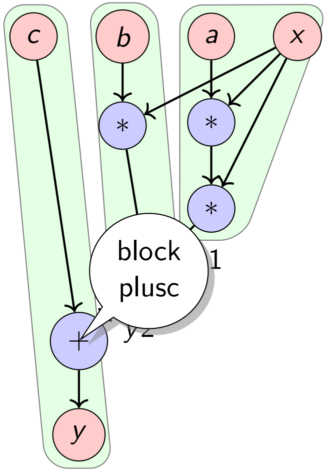
Simulation Example

```
always @(a or x) begin: axx
 $display("y1<=axx fired");
 y1 <= a * x * x; //
end

always @(y1 or b or x) begin: bx
 $display("y2<=y1+bx fired");
 y2 <= y1 + b * x; //
end

always @(y2 or c) begin: plusc
 $display("y<=y2+c fired");
 y <= y2 + c; //
end
```



Simulation Example

```
always @(a or x) begin: axx
 $display("y1<=axx fired");
 y1 <= a * x * x; //
end

always @(y1 or b or x) begin: bx
 $display("y2<=y1+bx fired");
 y2 <= y1 + b * x; //
end

always @(y2 or c) begin: plusc
 $display("y<=y2+c fired");
 y <= y2 + c; //
end
```



XSim script

```
xvlog poly_combi_tb.v poly_combi.v
xelab -debug typical poly_combi_tb -s poly_combi_tb

xsim poly_combi_tb -gui -t xsim.tcl
```

- Xsim is the preferred tool for AMD/Xilinx FPGAs

Always Block Firing Schedule (Verilog testbench)

```
initial begin
 #2 a <= 1; b <= 1; c <= 1; x <= 10; $display("x changes");
 #2 a <= 1; b <= 1; c <= 10; x <= 10; $display("c changes");
 #2 a <= 1; b <= 10; c <= 10; x <= 10; $display("b changes");
 #2 a <= 10; b <= 10; c <= 10; x <= 10; $display("a changes");
 #2 $finish;
end
```

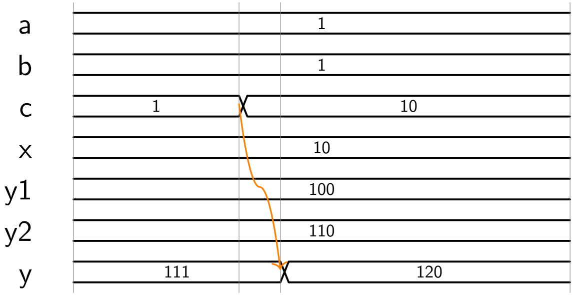
- Isolate effect of input changes – specific blocks will fire, and a certain number of times.
  - If only c changes?
  - If only x changes?
  - If only a changes?

Schedule order: c changes

```
  #2 a <= 1; b <= 1; c <= 10; x <= 10; $display("c changes");
# c changes
# y<=y2+c fired
# time=4 --> a=1,b=1,c=10,x=10,y=120
```

- Only `plusc` block will fire
- No other process sensitivity lists have any input change
- Will need single evaluation of `plusc`.

Timing Diagram c changes



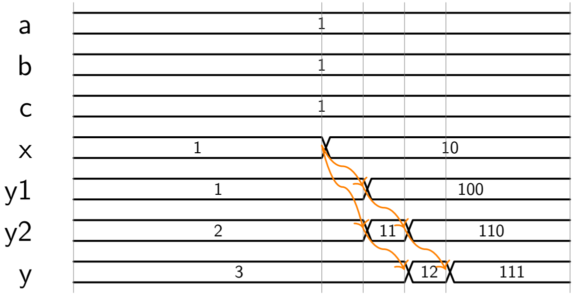
Schedule order: x changes

```
  #2 a <= 1; b <= 1; c <= 1; x <= 10; $display("x changes");
```

```
# x changes
# y1<=axx fired
# y2<=y1+bx fired
# y<=y2+c fired
# y2<=y1+bx fired
# y<=y2+c fired
# time=2 --> a=1,b=1,c=1,x=10,y=111
```

- All blocks fired
- bx + plusc fired twice, axx fired once
- Why did some blocks fire multiple times? Can we do better? (Yes: Later, RTL Simulation)

Timing Diagram x changes



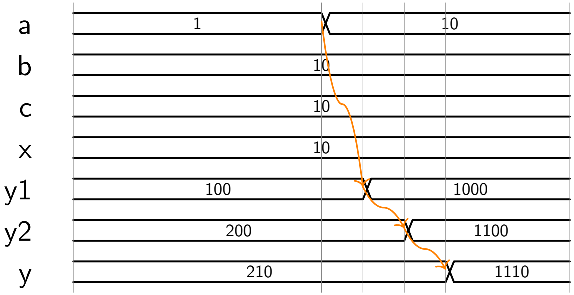
Schedule order: a changes

```
  #2 a <= 10; b <= 10; c <= 10; x <= 10; $display("a changes");
```

```
# a changes
# y1<=axx fired
# y2<=y1+bx fired
# y<=y2+c fired
# time=8 --> a=10,b=10,c=10,x=10,y=1110
```

- All blocks fired exactly once → signal propagation obeyed **topological** order of evaluation
- RTL Simulation does exactly this to avoid repeated firings

Timing Diagram a changes



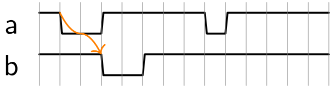
Inertial and Transport Delays

- Modeling hardware behavior requires two kinds of delay modeling
- `http://www.sunburst-design.com/papers/CummingsHDLCON1999_BehavioralDelays_Rev1_1.pdf`
- **Inertial Delay**: Captures inertia of a hardware signal in changing its value. Output will get value after delay, and ignore input pulses shorter than delay.
- **Transport Delay**: Models the delays where **all** input transitions are faithfully propagated to the output irrespective of pulse width.

Inertial Delay

- Models gate input behavior where input must persist for long enough to charge input capacitance of the gate.

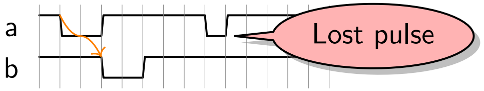
-
  ```
  assign #2 b = a;
  ```



- In this example, the small pulse does not make it to the output as it fails to charge the input capacitance of whatever the output is connected to.
- Timing models of gates use this delay model during simulation. Verilog simulation with timing models is a timing simulation. It is useful to verify that your circuit will work in hardware with considering models of physical delays.

Inertial Delay

- Models gate input behavior where input must persist for long enough to charge input capacitance of the gate.
- `assign #2 b = a;`

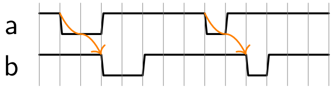


- In this example, the small pulse does not make it to the output as it fails to charge the input capacitance of whatever the output is connected to.
- Timing models of gates use this delay model during simulation. Verilog simulation with timing models is a timing simulation. It is useful to verify that your circuit will work in hardware with considering models of physical delays.

Transport Delay

- Models the delays where **all** input transitions are faithfully propagated to the output irrespective of pulse width.

-
  ```
  always @(a) begin
    b <= #2 a;
  end
  ```

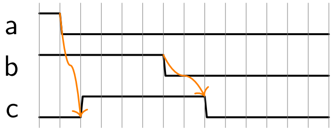


- In this example, both pulses get faithfully delayed to the output.

Pin-to-Pin Inertial Delays

- For large blocks too expensive to do fine grained delay modeling.
- Pin-to-pin delays allow top-level delays to be accurately modeled

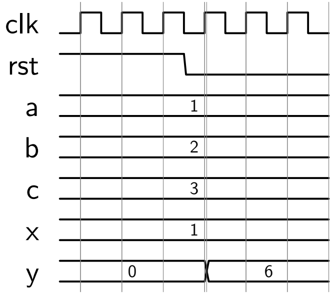
-
  ```
  assign c = a ^ b; // xor

  specify
    (a => c) = (1);
    (b => c) = (2);
  endspecify
  ```



- Inertial delays attached to outputs from specific inputs, and specific conditions

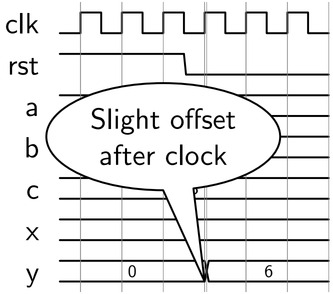
Timing Diagram for Clocked Verilog

```
always @(posedge clk) begin
  if(rst=1)
    y < =0;
  else
    y <= a*x*x + b*x + c;
  end;
end;
```



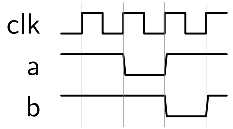
Timing Diagram for Clocked Verilog

```
always @(posedge clk) begin
  if(rst=1)
    y < =0;
  else
    y <= a*x*x + b*x + c;
  end;
end;
```

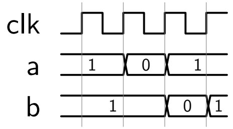


Offset after Clock Edge

- Signals updated on **posedge** `clk` require an extra evaluation step before assigned values are visible
- We show output updates shifted by a small amount after the clock edge has passed
- For boolean values use sloping lines `b<=a;`



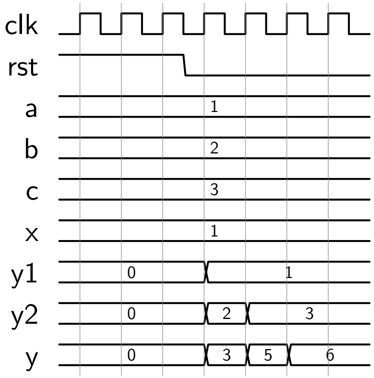
- For data busses delay the crossover `b<=a;`



- Important when signals change on each clock cycle $\rightarrow$ you must disambiguate which values to use when performing calculations

Timing Diagram for Clocked Verilog

```
always @(posedge clk) begin
  if(rst=1)
    y1 <= 0;
    y2 <= 0;
    y  <= 0;
  else
    y1 <= a*x*x;
    y2 <= y1+b*x;
    y <= y2+c;
  end;
end;
```



Simulator Algorithm (Bulk Synchronous Version)

```
while(time<FINISH_TIME) {
  while(!values_stable()) {
    for(block: blocks) { // check everything all the time
      if(sensitive(block,value)) {
        blocking_statements(block);
        continuous_statements(block);
        RHS_of_non_blocking_statements(block);
        display_statements(block);
      }
    }
    LHS_of_non_blocking_statements(block);
  }
  monitor_statements(block);
  time = time + stepsize;
}
```

- Outer loop advances time in the simulation
- Inner loop evaluates deltas, until the values have stabilized
- Conceptually simple, gates/blocks keep re-evaluating at a give time, steady-state detected

Simulator Algorithm (Bulk Synchronous Version)

```
while(time<FINISH_TIME) {
  while(!values_stable()) {
    for(block: blocks) { // check everyt
      if(sensitive(block,value)) {
        blocking_statements(block);
        continuous_statements(block);
        RHS_of_non_blocking_statements(block);
        display_statements(block);
      }
    }
    LHS_of_non_blocking_statements(block);
  }
  monitor_statements(block);
  time = time + stepsize;
}
```

Compute RHS values and stash away

Assign RHS values to LHS

- Outer loop advances time in the simulation
- Inner loop evaluates deltas, until the values have stabilized
- Conceptually simple, gates/blocks keep re-evaluating at a give time, steady-state detected

Simulator Algorithm (Bulk Synchronous Version)

```
while(time<FINISH_TIME) {
  while(!values_stable()) {
    for(block: blocks) { // check everything all the time
      if(sensitive(block,value)) {
        blocking_statements(block);
         ontinuous_statements(block);
            non_blocking_statements(block);
                  ents(block);



    LHS_of_non_blocking_statements(block);
  }
  monitor_statements(block);
  time = time + stepsize;
}
```

Non-deterministic evaluation

- Outer loop advances time in the simulation
- Inner loop evaluates deltas, until the values have stabilized
- Conceptually simple, gates/blocks keep re-evaluating at a give time, steady-state detected

Recommended Programming Practice

- `www.sunburst-design.com/papers/CummingsSNUG2000SJ_NBA_rev1_2.pdf`
  - Use non-blocking assignments for **sequential** (clocked) logic.
  - Use blocking assignments for **combinational** logic.
- Non-blocking assignments capture how hardware works.
- Blocking assignments can speedup Verilog simulations $\rightarrow$ overrated.

Recommended Programming Practice

Non-blocking everywhere!

- `www.sunburst-design.com/papers/Cumming`[illegible]`BA_rev1_2.pdf`
  - Use non-blocking assignments for **sequential** (clocked) logic.
  - Use blocking assignments for **combinational** logic.
- Non-blocking assignments capture how hardware works.
- Blocking assignments can speedup Verilog simulations $\rightarrow$ overrated.

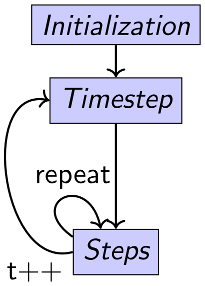
NON-BLOCKING ASSIGNMENTS
EVERYWHERE!!
imgflip.com

Simulator Algorithm (Event-Driven Version)

```
while(time<FINISH_TIME) {
 while(events_pending_in_active_queue()) {
  for(event: list_of_blocking_events()) {
    evaluate(event);
  }
  for(event: list_of_non_blocking_events()) {
    evaluate(event);
    schedule_new_event_in_active_queue();
  }
 }
 for(event: list_of_postponed_events()) {
   evaluate(event);
 }
 time = time + earliest_next_event(); // fast-forward
}
```

- Same as before, with the notion of scheduled events
- Ability to schedule events at future deltas (or even time)
- Conceptually simple, gates/blocks keep re-evaluating at a give time, steady-state detected
- Computationally efficient $\rightarrow$ can fast-forward simulation

Algorithm details



- Statements are either (1) **blocking**, (2) **continuous**, (3) **non-blocking**.
- *Initialization*: All signals start off with `'X'` values unless they have assignments in **`initial`** blocks.
- *Timestep*: Identify all blocks waiting at given time. Evaluate them until stabilization of signal values in the *Step* phase.
- *Step*: This evaluates blocking + continuous + RHS of non-blocking + display statements. Then it assigns values to LHS of non-blocking. The *Step* is repeated if required.
- VHDL has a more elegant **delta** cycle approach for handling ordering.

Stack Overflow

```
void thread_1(int *a, int *b,
int *c) {
  *a = *b + 1;
  thread_2(a,c);
}
```

```
void thread_2(int *a, int *c,
int *b) {
  *c = *a + 1;
   thread_1(a,b);
}
```

- Two functions call each other. Recursion has no bound, infinite jumps $\rightarrow$ stack overflow.
- Can hardware behave like this?
- **Yes** Unstable circuits, oscillators!

Stack Overflow

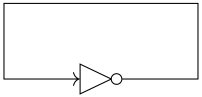
```
void thread_1(int *a, int *b,
int *c) {
  *a = *b + 1;
  thread_2(a,c);
}
```

```
void thread_2(int *a, int *c,
int *b) {
  *c = *a + 1;
   thread_1(a,b);
}
```

- Two functions call each other. Recursion has no bound, infinite jumps $\rightarrow$ stack overflow.
- Can hardware behave like this?
- **Yes** Unstable circuits, oscillators!

Ring Oscillator

```
always @(a) : invert
begin
        a <= not(a);
end;
```



- Ring Oscillator – circuit with its output fed back to its input
- Above circuit is valid! But will not simulate
- If feedback wire is initially `1'b0`, after inversion its value will become `1'b1`
- If feedback wire is initially `1'b1`, after inversion its value will become `1'b0`
- This results in a simulation failure! Why?

Ring Oscillator Testbench

```
module ring_osc (
  output wire y
 );

 reg a=1'b0; // initial value

 always @(a) begin: invert
  $display("Time=%0d, a=%0d", $time, a);
  a <= ~(a);
 end

 assign y = a;

endmodule
```

```
module ring_osc_tb();

 wire y;

 ring_osc ring_osc_inst(.y(y));

 initial begin
  $dumpfile("ring_osc.vcd");
  $dumpvars(0,ring_osc_tb);
 end

 always begin
  #10 $finish();
 end

endmodule
```

Ring Oscillator Simulation result

```
# vsim ring_osc_tb
# Start time: 17:07:05 on Apr 17,2019
# Loading work.ring_osc_tb
# Loading work.ring_osc
# Time=0, a=0
# Time=0, a=1
# Time=0, a=0
# Time=0, a=1
# Time=0, a=0
# Time=0, a=1
# Time=0, a=0
# Time=0, a=1
...
# Time=0, a=0
# ** Error (suppressible): (vsim-3601) Iteration limit 5000 reached at time 0 ps.
```

Ring Oscillator Testbench (Correct)

```verilog
module ring_osc (
  output wire y
 );

 reg a=1'b0; // initial value

 always @(a) begin: invert
  $display("Time=%0d, a=%0d", $time, a);
  a <= #1 ~(a);
 end

 assign y = a;

endmodule
```

```verilog
module ring_osc_tb();

 wire y;

 ring_osc ring_osc_inst(.y(y));

 initial begin
  $dumpfile("ring_osc.vcd");
  $dumpvars(0,ring_osc_tb);
 end

 always begin
  #10 $finish();
 end

endmodule
```

Ring Oscillator Simulation result

```
# vsim ring_osc_tb
# Start time: 17:08:19 on Apr 17,2019
# Loading work.ring_osc_tb
# Loading work.ring_osc
# Time=0, a=0
# Time=1, a=1
# Time=2, a=0
# Time=3, a=1
# Time=4, a=0
# Time=5, a=1
# Time=6, a=0
# Time=7, a=1
# Time=8, a=0
# Time=9, a=1
# ** Note: $finish    : ring_osc_tb.v(14)
#    Time: 10 ps  Iteration: 0  Instance: /ring_osc_tb
# End time: 17:08:20 on Apr 17,2019, Elapsed time: 0:00:01
# Errors: 0, Warnings: 0
```

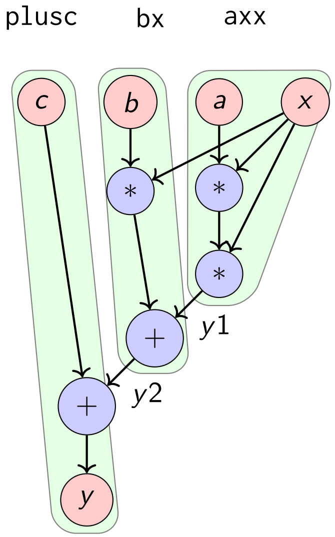
Dealing with combinational loops

- First ask do you really need it? Answer is almost always no
- Not all combinational loops cause oscillatory behavior *e.g.* two inverters in series. Feedback from second is input to first.
- Ring oscillator only valid for **odd** number of inverters in chain
- Latches can implement combinational loops, but delays must be carefully calibrated to avoid race conditions $\rightarrow$ Vivado discourages use of latches for FPGAs

RTL Simulation Example

```
always @(a or x) begin: axx
 $display("y1<=axx fired");
 y1 <= a * x * x; //
end

always @(y1 or b or x) begin: bx
 $display("y2<=y1+bx fired");
 y2 <= y1 + b * x; //
end

always @(y2 or c) begin: plusc
 $display("y<=y2+c fired");
 y <= y2 + c; //
end

endmodule
```

- In RTL simulation, blocks are topologically sorted upfront: axx → bx → plusc



Wrapup

- Verilog simulation model crucial to understanding concurrency/parallelism in hardware
- Event-driven simulation enforces ordering of parallel events to deliver consistent results
- Combinational loops defeat simulation (but are bad hardware design practice anyway)
- *If you understand concurrency, you understand how hardware works*