Simulation of Digital Systems

- Simulation: What do you do to test a software program you write?
  - Give it some inputs, and see if it does what you expect
  - When done testing, is there any assurance the program is bug free? — NO!
  - But, to the extent possible, you have determined that the program does what you want it to do
- Simulation tests a model of the system you wish to build
  - is the design correct? Does it implement the intended function correctly? For instance, is it a UART
  - and are the setup times met?
  - Also, is it the correct design?
  - Might there be some other functions the UART could do?
- It has all the limitations of software testing
  - Have I tried all the cases?
  - is there any assurance the program is bug free? — NO!
  - e.g. are the set-up times met?
  - http://www.ece.cmu.edu/~thomas

Modern Design Methodology

- Simulation and Synthesis are components of a design methodology
- Place and Route
- Synthesize
- Gate implementation
- Synthesis

Representation: Structural Models

- Structural models: Need to model the circuit using logic gates — much as you would see in an implementation of a circuit.
  - You could describe your lab1 circuit this way
  - They describe the circuit using logic gates — much as you would see in an implementation of a circuit.
- Gate instances, wire names, delay from a to b.
  - Output f; input a, b, sel;
  - and #5 instance-name (out, in1, in2, in3, …);
  - or #5 instance-name (out, in1, in2, in3, …);
  - optional

Four-Valued Logic

- Four-Valued Logic Values
  - 1, 0, x (unknown), z (high impedance)
  - 1 — one of 1, 0, x, or in the state of change
  - 0 — one of 0, 1, x, or in the state of change
  - x — the high impedance output of a tri-state gate.

- What basis do these have in reality?
  - 0, 1, 2, … no question
  - x, z — A nand gate drives either a zero or one to the output. If it's not driven, it's a unknown (x). Tri-state gates are driven.
  - x — not a real value. There is no real gate that drives an x on to a node. A gate treated as a debugging aid. A zero the simulator can't determine the answer and so maybe you should worry!
  - BTW: some simulators keep track of more values than these. Verilog will in some situations.

Four-Valued Logic

- Logic with multi-level logic values
  - Nand anything with a 0, and you get a 1. This includes having an x on the other input. That's the nature of the nand gate.
  - Nand two x's and you get an x.

- Note: a treated as an x on input. Their rows and columns are the same.
  - If you forget to connect an input... it will be seen as an x.
  - At the start of simulation, everything is an x.

© Don Thomas, 1998, Page 1
Another view of this

- 3 chunks of verilog, one for each of:
  - STRUCTURAL is the first piece of hardware which connect DESIGN with TEST as the inputs generated go to the thing you want to test.
  - TESTBENCH is the real piece of hardware which connect DESIGN with TEST as the inputs generated go to the thing you want to test.
  - Your hardware called DESIGN

Another piece of hardware called TEST to generate interesting inputs.

Other things you can do

- More than modeling hardware
  - $monitor — give it a list of variables. When one of them changes, it prints the information. Can only have one of these active at a time.
    - $monitor ($time, "a=%b, b=%b, sum=%b, cOut=%b", a, b, sum, cOut);
  - $display() — sorts of like printf()
    - $display("Hello, world — \"$b\", hexadecimal")
  - $display contents of data item called "hexvalue" using hexadecimal digits (0,1,A-F)

### Structural vs Behavioral Models

#### Structural model
- Just specifies primitive gates and wires
- I.e., the structure of a logical netlist
- You basically know how to do this now.

#### Behavioral model
- More like a procedure in a programming language
- Specifies a procedure in Verilog with inputs and outputs
- ...more to come

Why use behavioral models
- For high-level specs to drive logic synthesis tools (Lab 2)
- You write an "abstract" Verilog description of the logic
  - For high-level specs to drive logic synthesis tools (Lab 2)
  - High-level specs to drive logic synthesis tools (Lab 2)

Two initial statements?

- initial begin
  - $a = 0; $b = 0;
  - #10 out = 0;
- initial begin
  - $a = 1; $b = 1;
  - #10 out = 1;

Things to note
- Which initial statement starts first?
- What are the values of $a, $b, and out when the simulation starts?
- Things appear to be executing concurrently (at the same time). Are they?

### What do we mean by "Synthesis"?

#### Logic synthesis
- A program that "designs" logic from abstract descriptions of the logic
- Takes constraints (e.g. size, speed)
- Uses a library (e.g. 3-input gates)

#### How?
- You write an "abstract" Verilog description of the logic
- The synthesis tool provides alternative implementations constraints

#### An example

- What's cool?
  - You type the left, synthesis gives you the gates
  - It's the test generator
  - Only one monitor can be active at any time
  - Things appear to be executing concurrently (at the same time). Are they?
- $monitor ($time, "a=%b, b=%b, sum=%b, cOut=%b", a, b, sum, cOut);

- Things appear to be executing concurrently (at the same time). Are they?
**Behavioral Statements**

**If-then-else**
- What you would expect, except that it's 4-valued logic. It's interpreted as True: 0, a, and 0 are interpreted as False.

**Case**
- What you would expect, except that it's 4-valued logic.
  - If "selector" is 2 bits, there are 4^2 possible case-items to select between.
  - There is no break statement — it's assumed.

**Funny constants?**
- Four-valued logic. The first number is the number of bits, the letter is the base of the following number that will be converted into the bits.

**Registers**
- Define storage, can be more than one bit
- Can only be changed by assigning value to them on the left-hand side of a behavioral expression.

**Wires (actually "nets")**
- Electrically connect things together
- Can be used on the right-hand side of an expression
- Thus we can define primitive gates and behavioral blocks together.

**Statements**
- Left-hand side = right-hand side
- Left-hand side must be a register
- Preceded by :=

**Concurrent Constructs**

**What Do We Want Here...?**

**Behavioral Modeling**

**Procedural statements are used**
- Statements using "always" Verilog construct
- Can specify both combinational and sequential circuits

**Normally don't think of procedural stuff as "logic"**
- They look like C: mix of ifs, case statements, assignments
- ...but there is a semantic interpretation to put on them to allow them to be used for simulation and synthesis (giving equivalent results)

**Current technology**
- You can do combinational (and later, sequential) circuit design
- Sizable designs can take hours ... days ... to run

**Points:**
- They all execute concurrently
- They contain behavioral statements like if-then-else, case, loops, functions, ...

**Statements, Registers and Wires**

**Registers**
- Define storage, can be more than one bit
- Can only be changed by assigning value to them on the left-hand side of a behavioral expression.

**Wires (actually "nets")**
- Electrically connect things together
- Can be used on the right-hand side of an expression
- Thus we can define primitive gates and behavioral blocks together.

**Statements**
- Left-hand side = right-hand side
- Left-hand side must be a register
- Preceded by :=

**Bit Selects and Part Selects**
- This expression extracts bits or ranges of bits or a wire or register
- module testgen (i);reg [3:0] i;output i;always
  - for (i = 0; i <= 15; i = i + 1)
  - mumble, mumble, blah, blah;
  - endmodule

**Behavioral Statements**

**If-then-else**
- What you would expect, except that it's 4-valued logic. It's interpreted as True: 0, a, and 0 are interpreted as False.

**Case**
- What you would expect, except that it's 4-valued logic.
  - If "selector" is 2 bits, there are 4^2 possible case-items to select between.
  - There is no break statement — it's assumed.

**Funny constants?**
- Four-valued logic. The first number is the number of bits, the letter is the base of the following number that will be converted into the bits.

**Behavioral Constructs**

**Procedural statements are used**
- Statements using "always" Verilog construct
- Can specify both combinational and sequential circuits

**Behavioral descriptions are introduced by initial and always statements**

**Behavioral Statements**

**If-then-else**
- What you would expect, except that it's 4-valued logic. It's interpreted as True: 0, a, and 0 are interpreted as False.

**Case**
- What you would expect, except that it's 4-valued logic.
  - If "selector" is 2 bits, there are 4^2 possible case-items to select between.
  - There is no break statement — it's assumed.

**Funny constants?**
- Four-valued logic. The first number is the number of bits, the letter is the base of the following number that will be converted into the bits.

**Registers**
- Define storage, can be more than one bit
- Can only be changed by assigning value to them on the left-hand side of a behavioral expression.

**Wires (actually "nets")**
- Electrically connect things together
- Can be used on the right-hand side of an expression
- Thus we can define primitive gates and behavioral blocks together.

**Statements**
- Left-hand side = right-hand side
- Left-hand side must be a register
- Preceded by :=

**Concurrent Constructs**

**What Do We Want Here...?**

**Behavioral Modeling**

**Procedural statements are used**
- Statements using "always" Verilog construct
- Can specify both combinational and sequential circuits

**Normally don't think of procedural stuff as "logic"**
- They look like C: mix of ifs, case statements, assignments
- ...but there is a semantic interpretation to put on them to allow them to be used for simulation and synthesis (giving equivalent results)

**Current technology**
- You can do combinational (and later, sequential) circuit design
- Sizable designs can take hours ... days ... to run

**Points:**
- They all execute concurrently
- They contain behavioral statements like if-then-else, case, loops, functions, ...

**Statements, Registers and Wires**

**Registers**
- Define storage, can be more than one bit
- Can only be changed by assigning value to them on the left-hand side of a behavioral expression.

**Wires (actually "nets")**
- Electrically connect things together
- Can be used on the right-hand side of an expression
- Thus we can define primitive gates and behavioral blocks together.

**Statements**
- Left-hand side = right-hand side
- Left-hand side must be a register
- Preceded by :=

**Bit Selects and Part Selects**
- This expression extracts bits or ranges of bits or a wire or register
- module testgen (i);reg [3:0] i;output i;always
  - for (i = 0; i <= 15; i = i + 1)
  - mumble, mumble, blah, blah;
  - endmodule

**Behavioral Statements**

**If-then-else**
- What you would expect, except that it's 4-valued logic. It's interpreted as True: 0, a, and 0 are interpreted as False.

**Case**
- What you would expect, except that it's 4-valued logic.
  - If "selector" is 2 bits, there are 4^2 possible case-items to select between.
  - There is no break statement — it's assumed.

**Funny constants?**
- Four-valued logic. The first number is the number of bits, the letter is the base of the following number that will be converted into the bits.

**Behavioral Constructs**

**Procedural statements are used**
- Statements using "always" Verilog construct
- Can specify both combinational and sequential circuits

**Behavioral descriptions are introduced by initial and always statements**

**Behavioral Statements**

**If-then-else**
- What you would expect, except that it's 4-valued logic. It's interpreted as True: 0, a, and 0 are interpreted as False.

**Case**
- What you would expect, except that it's 4-valued logic.
  - If "selector" is 2 bits, there are 4^2 possible case-items to select between.
  - There is no break statement — it's assumed.
A Combinational Circuit

Is it really correct?

Problem?

- Where's the register?
- Logic for a simple MUX is specified procedurally here
- Using behavioral constructs
- This example is synthesizable

Using procedural statements in Verilog

Logic is specified in "always" statements ("Initial" statements are not allowed)

Typical Style

- Your Verilog for combination stuff will look like this:
- Assigning in every control path
  - if the function is complex, you don’t know if you assigned to the
  - outputs in every control path
  - if any input changes, then the output is re-evaluated
  - That’s the definition of combinational logic.

- There are some restrictions on specification
  - Input set of an “always” statement — the set of all variables that are used on the right-hand side of procedural assignments or in conditions, i.e. anything “assumed”
  - Sensitivity list of an “always” statement — the set of all names that appear in the event ("@") list

- A control path of an “always” statement — a sequence of operations performed when executing the always statement
- Assigning in every control path
- The combinational output must be assigned in every control path

- If you don’t follow the rules...? ... you’re dead meat
  - Verilog assumes you are trying to do something clever with the timing
  - It’s legal, but it won’t be combinational
  - The rules for what it does make sense — but not yet for us

- module mux (f, sel, b, c);
  - output f;
  - input sel, b, c;
  - always @ (sel or b or c) begin
    - if (sel == 1)
      - f = b;
    - else
      - f = c;
  - end
endmodule
Using a case statement

- Truth table method
  - List each input combination
  - Assign to output(s) in each row

- Concatenation
  - (a, b, c) concatenates a, b, and c together, considering them as a single item

Example:

\[ a = \{011\} \]
\[ b = \{01\} \] \( \text{xor} \) \[ c = \{01\} \]
\[ \text{then} \ (a, b, c) = \{011\} \] \( \text{xor} \) \[ \{01\} \] \( \text{and} \) \[ \{01\} \] \( = \{011\} \] \( \text{and} \) \[ \{01\} \]

Module feed (f, a, b, c):

```vhdl
module feed (f, a, b, c);
    input a, b, c;
    output f;
    reg f;
    always @ (a or b or c)
        begin
            3'b000: f = 1'b0;
            3'b001: f = 1'b1;
            3'b010: f = 1'b1;
            3'b011: f = 1'b1;
            3'b100: f = 1'b1;
            3'b101: f = 1'b0;
            3'b110: f = 1'b0;
            default: f = 1'b1;
        endcase
    endmodule
```

Here's another version...

Module fred (f, a, b, c):

```vhdl
module fred (f, a, b, c);
    output f;
    input a, b, c;
    reg f;
    always @ (a or b or c)
        begin
            3'b000: f = 1'b0;
            3'b001: f = 1'b1;
            3'b010: f = 1'b1;
            3'b011: f = 1'b1;
            3'b100: f = 1'b1;
            3'b101: f = 1'b0;
            3'b110: f = 1'b0;
            default: f = 1'b1;
        endcase
    endmodule
```

Rules

- You can't say `z` has meaning in simulation, but not in synthesis.
- However, an unknown `x` on the right-hand side will be interpreted as a don't care.

Check the rules...

- These aren't quite equivalent to the previous slide... why?

Two inputs, Three outputs

Module freg (f, a, b, c):

```vhdl
module freg (f, a, b, c);
    input a, b, c;
    reg f;
    always @ (a or b or c)
        begin
            3'b000: f = 1'b0;
            3'b001: f = 1'b1;
            3'b010: f = 1'b1;
            3'b011: f = 1'b1;
            3'b100: f = 1'b1;
            3'b101: f = 1'b0;
            3'b110: f = 1'b0;
            default: f = 1'b1;
        endcase
    endmodule
```

Alternatively...

Module fred1 (f, a, b, c):

```vhdl
module fred1 (f, a, b, c);
    output f;
    input a, b, c;
    reg f;
    always @(a or b or c)
        begin
            3'b000: f = 1'b0;
            3'b001: f = 1'b1;
            3'b010: f = 1'b1;
            3'b011: f = 1'b1;
            3'b100: f = 1'b1;
            3'b101: f = 1'b0;
            3'b110: f = 1'b0;
            default: f = 1'b1;
        endcase
    endmodule
```

Model Organization

- Here's an always block for a combinational function.
- What Boolean functions can it model?
- Can I have more than one of these always blocks in a module?
- Can two separate always calculate function?

Module input (f, sel, b, c):

```vhdl
module input (f, sel, b, c);
    output f;
    input sel, b, c;
    reg f;
    always @ (sel or b or c)
        begin
            0: f = 1'b0;
            1: f = 1'b1;
        endcase
    endmodule
```

Model Organization Trade-Off

- Module partitioning can affect logic optimizations
- Module feed (input, output):
- Module fred1 (input, output):
- Module fred2 (input, output):
- Module fred3 (input, output):
- Module fuse (input, output):
- Module box (input, output):
- Module block (input, output):
- Module block2 (input, output):

© Don Thomas, 1998, Page 5
Verilog Overview

Verilog is a concurrent language
- Always at modeling hardware — optimized for it!
- Typical of hardware description languages (HDLs), it:
  - provides for the specification of concurrent activities
  - stands on its head to make the activities look like they happened at the same time
- Why?
  - allows for intricate timing specifications

A concurrent language allows for:
- Multiple concurrent "elements"
- An event in one element to cause activity in another. (An event is an output or state change at a given time)
  - based on interconnection of the element's ports
- Further execution to be delayed
  - until a specific event occurs

Discrete Event Simulation

Quick example
- Gate A changes its output. This causes gates B and C to execute
  - But as we'll see, A doesn't call B and C (as in a function call)
- Rather, they execute because they're connected

Observation
- The elements in the diagram don't need to be logic gates
- SimCity is a discrete event simulator, Verilog too

Discrete Event Simulation

Events — changes in state — occur at discrete times. These cause other events to occur.
- Time advances in discrete (not continuous) steps
- Could you do logic circuits that way too?
  - e.g. analog circuits, numerical integration ...
  - differential equations to solve

Approach to Simulating a System

Basic models — things not found in C
- Gate level — built-in models for AND, OR, ...
  - When an input to one of these changes, the model executes to see if its output should change
- Behavioral level — sort-of C-like programs but with a few extra operators
  - Executes until it blocks for one of three reasons — wait, wait(n), or delay — when the reason for blocking is resolved, it continues executing
  - Does C have any notion of these?
- Gate and behavioral models can advance time

How does it keep track of time?

Explicitly
- Events are stored in an event list (actually a 2-D list) ordered by time
- Events execute at a time and possibly schedule their output to change at a later time (a new event)
- When no more events for the current time, move to the next
- Events within a time are executed in arbitrary order

Verilog Levels of Abstraction

Gate modeling
- The system is represented in terms of primitive gates and their interconnections
- NAND, NOR

Behavioral modeling
- The system is represented by a program-like language

Mixing Levels

Generally there is a mix of levels in a model
- e.g. part of the system is at the gate level and another part is at the behavioral level

Why?
- Early in design process you might not have fully-detailed models — you don’t actually know all the gate implementations of the multipliers, adders, register files
  - You might want to think of the design at a conceptual level before doing all the work to obtain the gate implementations
- There might be a family of implementations planned
- Levels — switch, gate, functional block (e.g. ALU), register-transfer, behavioral

A Gate Level Model

A Verilog description of an SR latch
A Gate Level Model

A Verilog description of an SR latch

Things to note:
- It doesn’t appear “executable” — no for loops, if-then-else, etc.
- It’s not in a programming sense, rather it describes the interconnection of elements.

A new module made up of other modules has been defined:

```verilog
module nandLatch (q, qBar, set, reset);
    output q, qBar;
    input set, reset;
    `gate nand (g1 (qBar, q, reset), g2 (qBar, q, reset));
endmodule
```

Things to note:
- Software engineering aspect — we can hide detail
- Timing model — how time is advanced, what triggers new output

So we can hide detail:

```verilog
module d_type_FF (q, clock, data);
    output q;
    input clock, data;
    reg q;
    always @ (posedge clock) q = #10 data;
endmodule
```

```
58© Don Thomas, 1998,
```

Comparison
- These two models are interchangeable — either could have been instantiated into a register
- ports in same order
- same delay from clock to q
- one is abstract, clear
- one is specific
- there are subtle differences

```
59© Don Thomas, 1998,
```

Quick example
- Gate A changes its output, gates B and C are evaluated to see if their outputs will change, if so, their fanouts are also followed...The behavioral model will only execute if it was waiting for a change on the A input.

```
60© Don Thomas, 1998,
```

Behavioral Modeling

Why not describe a module’s function and delay using a language like C?
- Sounds fun, here goes

```
61© Don Thomas, 1998,
```

At first look, it is a lot like C
- Most of the operators are the same as C
- makes it easy to read
- But there are major differences (quick list, we’ll get to these)
  - statements like delay (event, wait/level)
  - the language is concurrent — can specify many things that can happen at the same time.
  - four-valued logic (1, 0, x, z) and the operators to go with them
  - arbitrary bit width specification
  - there are a couple of procedural assignments (=, <=) with subtle differences
  - a different timing model

Behavioral Timing Model (not fully detailed here)

How does the behavioral model advance time?
- # — delaying a specific amount of time
- @ — delaying until an event occurs ("posedge", "negedge", or any change)
- this is edge-sensitive behavior
- wait — delaying until an event occurs ("wait (f == 0)")
- this is level sensitive behavior

What is a behavioral model sensitive to?
- any change on any input?
- any event that follows, say, a "posedge" keyword
- e.g. @posedge clock
- Actually "or" here too, not always

What are behavioral models sensitive to?
- Gate A changes its output, gates B and C are evaluated to see if their outputs will change, if so, their fanouts are also followed.
- The behavioral model will only execute if it was waiting for a change on the A input.

```
62© Don Thomas, 1998,
```

Order of Execution

In what order do these models execute?
- Assume A changes. Is B, C, or the behavioral model executed first?
- Answer: the order is defined to be arbitrary
- All events that are to occur at a certain time will execute in an arbitrary order.
- The simulator will try to make them look like they all occur at the same time — but we know better.
### Some more gate level examples

#### An adder

```verilog
module adder (carryOut, sum, aInput, bInput, carryIn);
output carryOut, sum;
input aInput, bInput, carryIn;
assign sum = aInput ^ bInput ^ carryIn;
assign carryOut = (aInput & bInput) | (bInput & carryIn) |
                            (aInput & carryIn);
endmodule
```

### Adder with delays

```verilog
module adder (carryOut, sum, aInput, bInput, carryIn);
output carryOut, sum;
input aInput, bInput, carryIn;
assign #(3, 5) sum = aInput ^ bInput ^ carryIn;
assign #(4, 8) carryOut = (aInput & bInput) | (bInput & carryIn) |
                              (aInput & carryIn);
endmodule
```

### Continuous assignment assigns continuously

- Delays can be specified (same format as for gates) on whole equation.
- No instance names — nothing is being instantiated. Given the same delays in this and the gate-level model of an adder, there is no functional difference between the models.

### Gate level timing model

- **Execution model**
  - **Execution model** — how time advances and new values are created
  - A fundamental concept in any language.

- **Gate level timing model**
  - Applies to both primitive instantiations and continuous assigns.

- **Definition**
  - When an input changes, the simulator will evaluate the primitive or continuous assign statement, calculating a new output.
  - No registers are latched/loaded, no need to know about a clock event
  - Anything on the left-hand side of the "=" in a continuous assign.
  - Outputs on this "side" of the language are all shadows.
  - No registers are latched/loaded, no need to know about a clock event.
  - The left-hand side of the "=" is propagated to other primitives and again as new assigns.

### Event-Driven Simulation

- How does the simulator execute a gate-level model?
- Event-driven simulation.
  - What is an event?
  - A value change occurs at a given time.
  - The event-driven simulator only executes models when events occur.
  - Some simulators execute every model every time unit.
**Events**

- Two types of events
  - Evaluation events — evaluate, or execute, a gate model or continuous assign
    - produce update events
    - if the output changes, schedule an update event
  - Update events — propagate new values along a fanout.
    - produce evaluation events
    - for each element on the fanout, schedule an evaluation event

- We’ll treat these as separate types of events
  - gate level simulators generally combine them for efficiency
    - i.e. when an output is updated, instead of scheduling an evaluation, just do the evaluation and schedule any updates resulting from it.
  - We’ll keep them separate for now — it will help in the later discussion of behavioral models

**Event-Driven Simulation**

- while something in time-ordered event list {
  - advance simulation time to top event’s timeretrieve all events for this time
  - For each event in arbitrary order
    - if it’s an update event
      Update the value specified.
      Follow fanout and evaluate gate models.
      Schedule any new updates from gates.
      Schedule eval events for behavioral models
    - else // it’s an evaluation event
      evaluate the model
      schedule resulting update events
  }

**Gate level timing model**

- What if an update event is already scheduled for an output?
  - if the value being scheduled is different, the currently scheduled value is removed from the event list; the new is not scheduled
  - thus, any input pulse shorter than the propagation delay will not be seen (inertial delay)

**Scheduling and event list management**

- Can think of the event list as a 2-D linked list
  - One dimension links all the events for a given time
  - The second dimension links these lists in ascending order

- Problem
  - inefficient — most events are near in time to the current one, thus lots of linked list bashing

**Asides**

- Can a gate model be executed several times in a time step?
- Does the order of execution of the gates in a combinational circuit matter?

**Summary on gate evaluation**

- Timing model
  - timing-execution model:
    - new time is advanced and new values created
  - Any gate input or assign right-hand-side change causes the model to be evaluated during the time step
  - this is not the case for behavioral models — they have a different timing model
  - Fanout list is static — design never changes

- Gate level modeling
  - detailed timing

- Continuous assignment
  - abstract
  - What if you don’t like these models?
    - e.g., inertial delays?
    - use behavioral models

© Don Thomas, 1998, Page 9
**Review Stuff**

- **Update Events**
  - A new value appears at some simulated time
- **Evaluation Events**
  - A model is evaluated (proceeded) at some simulated time
- **Event List**
  - A time-ordered list of events
- **Simulation scheduler**
  - Software program that manages the event list by scheduling update and evaluation events, tracing fanouts to propagate values, and manages simulated time

**Behavioral Timing Model**

- **How does the behavioral model advance time?**
  - $a$ — delaying until an event occurs — e.g., $@v$
  - this is edge-sensitive behavior
  - When the event is encountered, the value $v$ is sampled.
  - $a = --$ delaying until an event occurs ($"a = --"$)
    - this is level sensitive behavior
  - While one model is waiting for one of the above reasons, other models execute — time marches on
- **Semantics**
  - $@$ — delaying until an event occurs — e.g., $@v$
  - A model is executed (evaluated) at some simulated time
  - wait (expression) statement;
    - - while $a$ becomes 35, it resumes with $q = q + 4$
  - $@$ and $#$ always “block” the process from continuing
  - Partial comparison to $@$ and $#$

**Events — @something**

- **Action**
  - when first encountered, sample the expression
  - wait for expression to change in the indicated fashion
- **Examples**
  - always begin
    - @posedge clk; $q = d$;
  - @hello or goodbye; $a = b$
  - always begin
    - @hello; $a = b$
  - @posedge hello or negedge goodbye; $a = b$

**Sensitivity Lists**

- **In the gate level timing model...**
  - model execution was sensitive to any change on any of the inputs at any time
  - sensitivity list — a list of inputs that a model is sensitive to
    - a change on any of them will cause execution of the model
  - In a structural timing model, the lists don’t change.
  - @posedge hello or negedge goodbye is an event
  - Examples in procedural models...
    - the sensitivity list changes as a function of time and execution

**Fanout Lists**

- Outputs of things are connected to inputs of other things
  - No surprise
  - The simulator maintains a list of inputs driven by each “output”
  - Why?
    - When the output changes, it’s easy to figure out what other models need to be evaluated
  - What’s an “output” in the above sense?
  - Because of procedural models ...
    - Fanout lists ↔ Sensitivity lists
Behavioral Timing Model

- What is the behavioral model sensitive to?
  - The behavioral statements execute in sequence (one then the next)
  - i.e., it is only sensitive to what it is currently waiting for
  - time, edge, level — (offs, wait)
  - The model is not sensitive to a change on y or w.

Scheduling #, @, and Wait

- How are #, @, and wait tied into the event list?
  - # delayed: schedule the resumption of the process — put it in the event queue
  - @ delayed: schedule the resurrection of the process
  - When suspended for an @v, the behavioral model is put on the fanout list(s)
  - When an update event for v occurs, (e.g. posedge), then the behavioral model is scheduled to resume at the current time — an evaluation event.
  - Wait (exp)
    - If exp is TRUE, don’t stop
    - If exp is FALSE, then the behavioral model is put on the fanout list(s)
  - All right-hand sides are evaluated before any left-hand sides are updated
  - All assignments guarded by the edge happen concurrently
  - Concurrent Assignment — primary use of <=
    - All assignments guarded by the edge happen concurrently
    - Example — intra-assignment time delay
      - a <= #10 b + c;
      - Values after the clock edge ([t]) — calculated in response to the clock edge, using values at the clock edge
      - Values at the clock edge (At t)

Concurrent Assignment — guarded by an edge

Example — intra-assignment time delay

Non-blocking assignments (<=)

- Non-blocking assignments use "="
  - a <= #10 b + c;

The Important Aspect ...

- Non-Blocking Concurrent transfers
  - Across the whole design, all right-hand sides are evaluated before any left-hand sides are updated.
  - Thus, the order of #c’s evaluated and #c’s updated can be arbitrary (but separate)
- This allows us to ...
  - handle concurrent specification in major systems
  - reduce the complexity of our description
  - attach lots of actions to one event — the clock

Alternates — not all equivalent

- Area different animal?
- The same?

Edges in time — concurrent assignment

Intra-Assignment Non-blocking Example

- What’s the difference?
  - Assume c = 1
  - a2 = #5 b & c;

The Important Aspect ...

- Two important aspects to these
  - an intra-assignment time delay doesn’t stop them (they’re non-blocking)
  - they implement a concurrent assignment

Example — intra-assignment time delay

Non-blocking assignments use "="

What happens?

© Don Thomas, 1998, Page 11
A State Change

- Find all of your "state" variables
  - Not just FSM state, but registers in a datapath too
  - They're probably all keyed to an edge of a clock
  - Use <= to assign to them at the edge
  - You're guaranteed they'll all be sampled before any of them are updated.

A check: in many cases, the only #delay operator you need is in the clock (for functional specification).

Event List: We told a fib

This is what we told you before:

- In a concurrent language, there are some very dirty issues regarding the "arbitrary order" of execution.
- In software, such issues are handled by synchronization primitives - Some of you have probably seen semaphores in the OS or real-time (embedded systems) course - They only allow other concurrent parts of a system to see full state changes, not partial. State changes appear "atomic" - These provide a very clean way to enforce order (actually, mutual exclusion) within "zero time".

Follow the Execution

module fsm (Q1, Q0, in, clock);
output Q1, Q0;
input clock, in;
reg Q1, Q0;
always @(posedge clock) begin
  Q1 <= in & Q0;
  Q0 <= in | Q1;
end
endmodule

always #10 clock = ~clock;

module dff (Q, D, clock);
output Q;
input clock, D;
reg Q;
always @(posedge clock) begin
  Q <= D;
end
always #10 clock = ~clock;

Follow the Execution

module fsm (Q1, Q0, in, clock);
output Q1, Q0;
input clock, in;
reg Q1, Q0;
always @(posedge clock) begin
  Q1 <= in & Q0;
  Q0 <= in | Q1;
end
endmodule

always #10 clock = ~clock;

Follow the Execution

module dff (Q, D, clock);
output Q;
input clock, D;
reg Q;
always @(posedge clock) begin
  Q <= D;
end
always #10 clock = ~clock;

Follow the Execution

module dff (Q, D, clock);
output Q;
input clock, D;
reg Q;
always @(posedge clock) begin
  Q <= D;
end
always #10 clock = ~clock;

More Scheduling

<table>
<thead>
<tr>
<th>Time</th>
<th>Q1</th>
<th>Q0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

More Scheduling

<table>
<thead>
<tr>
<th>Time</th>
<th>U</th>
<th>V</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

More Scheduling

<table>
<thead>
<tr>
<th>Time</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Other strange things you can do

A 4-stage pipelined multiplier
- every clock edge, the a and b inputs are read and their product is scheduled to appear three clock periods later

```verilog
module pipeMult (product, a, b, ck);
    input [9:0] a, b;
    input ck;
    output [19:0] product;
    reg [19:0] product;
    always @ (posedge ck)
        begin
            a = 3 + 4;
            b = 5;
            c = a * b;
            product = repeat (3) @ (posedge ck) a * b;
        end
endmodule
```

Some ugly ramifications

You need to be careful when mixing blocking and non-blocking assignments
- blocking — you can read it like regular C language assignments.
- non-blocking — you can read it like regular C language assignments.

```verilog
if (there are no events for the current time)
    do blocking,
else
    do non-blocking
```

Delay Models

Common use delay:
- transport delay — input to output delay
- inertial delay — how long must an input spike be to be seen?
- in Verilog, inertial = transport

Kinds of delays

Definitions
- zero delay models — functional testing
- unit delay models — all gates have delay 1. OK for feedback
- transport delay — input to output delay
- inertial delay — too small, no output change

Delay Models

Other factors
- delay can be a function of output transition
- need a number for each of the arrows

Verilog example

```verilog
module delayModel (a, b, c);
    input a, b;
    output c;
    wire [9:0] delay;
    wire [9:0] inertialDelay;
    reg [9:0] delay;
    reg [9:0] inertialDelay;
    always @ (posedge a)
        begin
            delay = 3;
            inertialDelay = 0;
        end
endmodule
```

Gate-Level Modeling

Need to model the gate's:
- function
- delay
- Generally, HDLs have built-in gate-level primitives
- Verilog has NAND, NOR, AND, OR, XOR, XXNOR, BUF, NOT, and some others
- The gates operate on input values producing an output value
- typical Verilog gate instantiation:

```verilog
NAND 01xz
```

Logic Values

Verilog Logic Values
- 1, 0, x (unknown), z (high impedance)
- a — one of: 1, 0, x, or in the state of change
- z — the high impedance output of a tri-state gate. Generally treated as an x on an input

Procedural
delay (out, in1, in2, in3, …)
- multi-level logic used in some models to represent:
  - values, edges, unknowns, high impedances, ...

Logic with multi-level logic values
- note: z treated as an x on input
- logic values (Verilog does)

Types of delays

Definitions
- Zero delay models — functional testing
- there’s no delay, not cool for circuits with feedback!
- Unit delay models — all gates have delay 1. OK for feedback
- Transport delay — input to output delay
- Inertial delay — how long must an input spike be to be seen?
- in Verilog, Inertial = Transport

Delay Models

Other factors
- delay can be a function of output transition
- need a number for each of the arrows

Verilog example

```verilog
module delayModel (a, b, c);
    input a, b;
    output c;
    wire [9:0] delay;
    wire [9:0] inertialDelay;
    reg [9:0] delay;
    reg [9:0] inertialDelay;
    always @ (posedge a)
        begin
            delay = 3;
            inertialDelay = 0;
        end
endmodule
```

Delay Models

Unknown Delays — different simulators do different things
- d = randomize (min, max, distribution)
- delay is determined per gate at simulator startup time, same time used for gate throughout
- might model TTL chips, but not gates on an IC
- why?
- d = min, typical, max
- delay to be used is determined by simulator command at simulator startup time (i.e. one is selected)
- for Verilog, each of the three timing values can be replaced by a triple (min:typ:max)
- note # (3, 4, 5, 6, 7, 8, 9) (sbar, s)
Delays on Wires

1. How do you drive wires?
   - Gate outputs can drive wires
   - Gate outputs implicitly define wires
   - Wires can also be defined — with or without delay

2. Model Evaluation
   - Gate evaluation
     - The design is made up of primitive gates and behaviors
     - We're only considering primitive gates

Approach #1 — Input counting method

- The delay on a wire is added to any delay in the gate(s) that drive the wire

Approach #2 — Input counting method

- The evaluation function:
  ```
  if (v == c) return (c \oplus i)
  if (v == x) x_val = TRUE
  if (c_count > 0) return c
  if (a_count > 0) return a
  return (c' \oplus i)
  ```

Simulation: Model Evaluation

- Approach #3: Input counting
  - An update event keeps count of various features
  - When 1 \to 0 on AND gate, increment c_count
  - When 0 \to x on AND gate, decrement c_count, increment x_count

- an update event becomes
  ```
  if (c_count > 0) return c
  if (x_count > 0) return x
  ```

- Can you make this work with XORs?

Tying Behavior and Gate Models together

- A real design consists of a mixture of behavior and gate models

- An alternate version
  ```
  ```

© Don Thomas, 1998, Page 14
Names of things

- Thus far, we’ve seen names of:
  - registers, variables, inputs, outputs, instances, integers
  - Their scope is the begin-end block within which they were defined
  - Their scope is the begin-end block within which they were defined
  - task — endtask
  - function — endfunction
  - begin name — end
  - … nothing else within that scope may already have that name

Types of references

- Forward referenced — Identifiers for modules, tasks, functions, named-blocks may be used before being defined
- Not Forward referenced — must be defined before use
- wires and registers
- Hierarchical references — named through the instantiation hierarchy
  - “a.b” references identifier b in namespace a

Identifiers

- Forward referenced
  - Identifiers of modules, tasks, functions, named-blocks
  - Hierarchical search tree defined by module instantiation
  - Identifiers within each higher scope are known
  - After all instantiations are known, search upward for the first identifier
    - a.b.c.d
    - When found go down through the rest of the name

- Non-Forward referenced
  - Identifiers for registers and wires (non-hierarchical)
  - Hierarchical search tree defined by nested procedural blocks
    - rooted in module
    - Search doesn’t cross module instantiation boundaries

- Hierarchical — registers and wires
  - These are forward referenced — see above

Scope of functions and tasks

- Where defined
  - functions and tasks are defined within modules

- Scope
  - As with other names, the scope of the functions and tasks is the begin-end block (module-endmodule) within which they are defined
  - They can also be accessed hierarchically
    - define “global” functions and tasks in the “top” module
    - they’ll be accessible from any (recursively) instantiated module.