Verilog

Winter ’09 CSE378 Section for January 22

Jacob Nelson

Agenda

- Verilog
- Two's complement arithmetic
- Using stack

Verilog tips and traps

=   vs.   <=

- Simple rule:
  - If you want sequential logic, use always @(posedge clk) with <=.
  - If you want combinational logic, use always @(*) with =.

HW Tools: pain in digital form?

- We should teach ideas, not tools!
  But tools help express ideas
- HW tools often kind of suck, but
  don't blame tools for bad craftsmanship,
  do good craftsmanship with bad tools.
- Patience and care will win

Datatypes

- "vector": wire/reg
  any width you want
- wire x = 1'b1;
  wire [7:0] = 7'b10110100;
- 8'd10, 8'b10, 8'o10, 8'h10
- also integer, float, time, but we'll ignore.

Comparators

- assign isZero = (a == 0);
- assign isGreater = (a > b);   // unsigned!!!
  assign isLTZ = (a < 0);   // is this ever true?
- can do signed compares if ALL signals involved are declared “signed”.

```
wire signed [7:0] a, b;    assign a = b < 0;
   or
wire [7:0] a, b;
assign a = $signed( $signed(b) < 0);
```

Concatenation and replication

- wire a = 8’b10110100;
  wire [?:0] b = { 6’b0, a, 2’b0 };

- wire x = 1’b1;
  wire [7:0] y = {7{x}};

  wire [?:0] c = { {6{a[7]}}, a, 2’b0};

Shifting

- wire [7:0] x = 7’b10110100;
- wire [?:0] y = x << 2;

  is it
  wire [7:0] x = { x[7:2], 2’b0 };
  or
  wire [9:0] x = { x[7:0], 2’b0 };

Adders/subtractors

- ```
  assign f = a + b;
  assign g = a - b;
  ```
- ```
  wire [8:0] s, t;
  wire [7:0] a, b;
  assign s = {0,a} + {0,b};  // pick up carry out
  assign t = a + b;          // equivalent
  ```
- what about multiplication? division?

Logic

- ```
  wire a = 1'b0;
  wire b = 1'b1;
  wire c = 1'b1;
  wire f = c && (a || b);
  ```
- ```
  wire [7:0] d = 8'b10110100;
  wire [7:0] e = 8'b11001100;
  wire [7:0] x = 8'b00000001;
  wire [7:0] g = x | (d & e);
  ```

Reduction operators

- wire [7:0] foo = 8'b10110100

```
wire anyFoo0 = |foo;
wire anyFoo1 = foo != 8'b0;
wire anyFoo2 = foo[7] || foo[6] || ...

wire allFoo0 = &foo;
wire allFoo1 = foo == 8'b11111111;

wire parity = ^foo;  // xor reduction
```

?:, If, Case: Muxes

```
assign f =
  s[1] ?
    (s[0] ? a : b) :
    (s[0] ? c : d);

  always @(*)
    case (s)
      2'b00 : g = a;
      2'b01 : g = b;
      2'b10 : g = c;
      default: g = d; // 2'b11
    endcase
```

```
always @(*)
  if (s == 2'b00)
     h = a;
  else if (s == 2'b01)
     h = b;
  else if (s == 2'b10)
     h = c;
  else  // s == 2'b1
     h = d;
```

Literals: 32 bits, decimal

- wire [7:0] foo = 127; // synthesis warning!
- wire [7:0] foo = 8'd127;
- wire [7:0] foo = 8'b11111111;
- wire [7:0] foo = 8'hff;
- wire [7:0] foo = 8'hFF;
- watch out: 1010 looks like 4'b1010!

Truncation

```
wire [7:0] a = 8'hAB;
wire b;                    // oops! forgot width
wire [7:0] c;

assign b = a;      // synthesis warning if lucky.

assign c = a;
```

Logic vs. registers

```
module foo (a,b,f,g);
   input wire a, b;
   output wire f;
   output reg g;

   assign f = a && b;
   always @(*)
      g = a && b;
endmodule
```

```
module(clk, d, q);
  input wire clk, d;
  output reg q;

  always @(posedge clk)
     q <= d;
endmodule
```

reg vs. wire

- ```
  wire f;      reg g, h;

  assign f = a & b;

  always @(*)     // equivalent to above
    g = a & b;

  always @(posedge clk)
    h <= a & b;
  ```
  ```

Assign in one block

```
input wire a, b;
output reg f;

always @(posedge clk)
   if (a) f <= 1'b0; // race!

always @(posedge clk)
   if (b) f <= 1'b1; // race!
```

= vs. <=

- Simple rule:
  - If you want sequential logic, use always @(posedge clk) with <=.
  - If you want combinational logic, use always @(*) with =.

=   vs. <=

- ```
  always @(posedge clk)
    begin
      f <= a + b;
      g <= f + c;
    end
  ```

- ```
  always @(posedge clk)
    begin
      f = a + b;
      g = f + c;  // a + b + c
    end
  ```

- ```
  always@(posedge clk)
    begin
      f2 <= f1;
      f3 <= f2;

      f4 = f3;
      f5 = f4;   // f5 = f3 !!

      f7 = f6;
      f6 = f5;
    end
  ```
  ```

More specifically,

```
initial
  state = 0;

always @(posedge clk)
  begin
    if (state == 0) state = 1;
    if (state == 1) state = 2;
    if (state == 2) state = 0;
  end
```

Aargh.

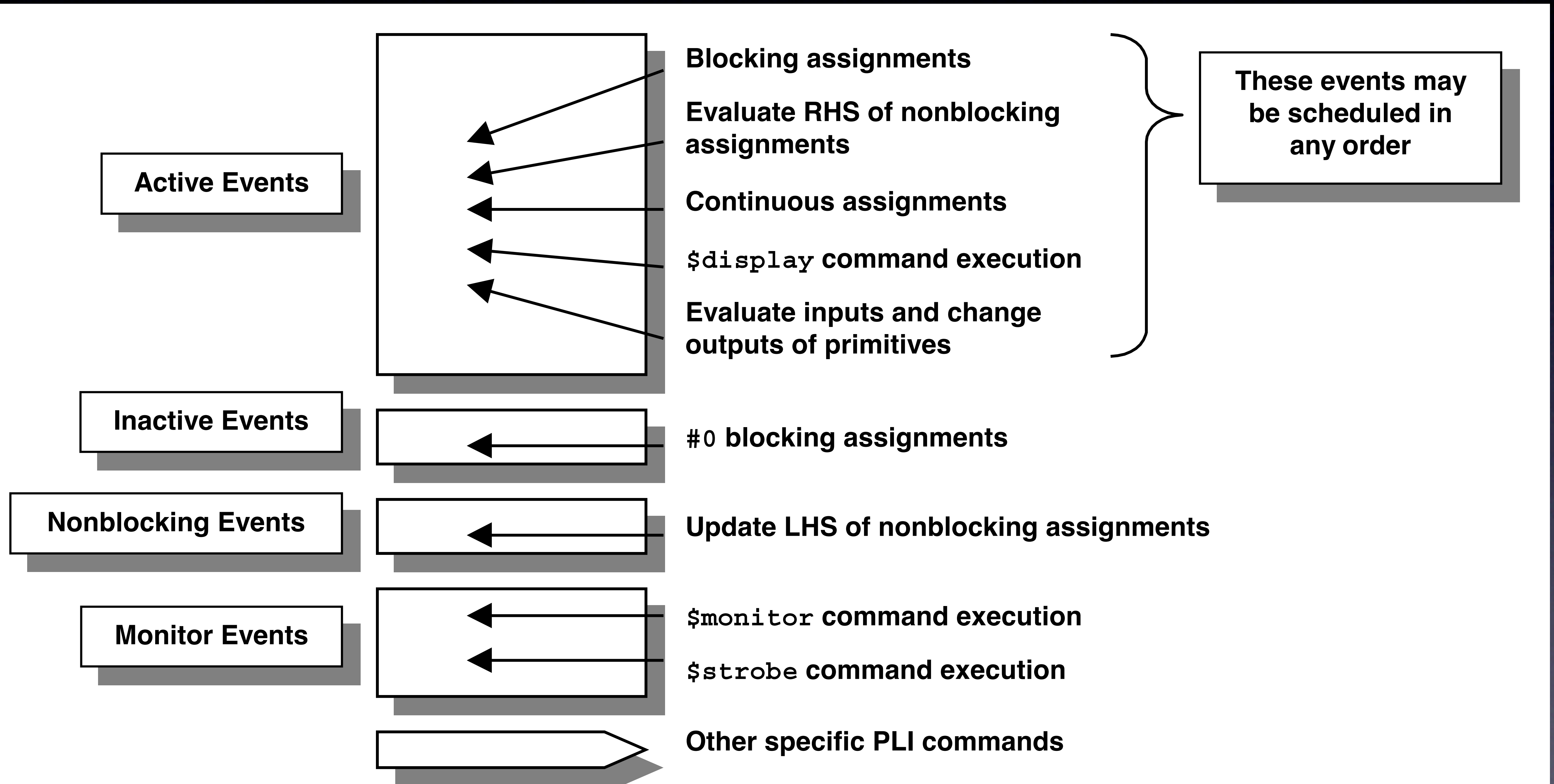

Figure 1 - Verilog "stratified event queue"

from Cliff Cummings' "Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!"

Incomplete sensitivity lists

- always @(a or b) // it's or, not ||
  f = a & b;

- always @(a)
  f = a & b;

- always
  f = a & b;

- Just use always@(*) for combinational logic

Enables and Latches

- always @(posedge clk)
  ```
  if (a == 1)
    f <= 1;
  else if (a == 2)
    f <= 2;
  else if (a == 3)
    f <= 3;
  ```
- implicitly:
  ```
  else
    f <= f;
  ```

- always @(*)
  ```
  if (a == 1)
    f = 1;
  else if (a == 2)
    f = 2;
  else if (a == 3)
    f = 3;
  ```
- implicitly:
  ```
  else
    f = f;
  ```
  this is memory!

=   vs.   <=

- Simple rule:
  - If you want sequential logic, use always @(posedge clk) with <=.
  - If you want combinational logic, use always @(*) with =.

Combinational and Sequential

```
input wire a, b, s;
output reg f, g, h;

always @(posedge clk)
  begin

    f <= (a & ~s) | (b & s);

    g <= s ? a : b;

    if (s)
      h <= a;
    else
      h <= b;

  end
```

Displaying things

- works for most stuff:
  ```
  $display("the answer is %h.", ans);
  ```
- for nonblocking assignments, you may sometimes want:
  ```
  $strobe("the answer is %h.", ans);
  ```
  (see Aargh. for reason)

X's

- X's are for undefined values:
  ```
  wire a;
  $display(a);   // prints an X
  ```
- Pins that aren't hooked up will be X's: Often, 32'hxxxxxxf4 indicates an Active-HDL bus with default width.
- 1'b1 & 1'bX yields 1'bX
  1'b1 + 1'bX yields 1'bX

Z's

- Z's are for bus sharing. You won't need this.
- a <= 1'bZ;   b <= 1'bZ;
  a <= 2'b0;   b <= 1'b1;
  // a will be 0 and b will be 1
- Z's turn into X's sometimes:
  1'b1 & 1'bZ yields 1'bX.
  1'b1 + 1'bZ yields 1'bX.

Initial values

- Synthesis sometimes ignores (!?!), so better include a reset line.
- Maybe:
```
reg foo = 1'b1;
```
- Maybe:
```
initial begin
  foo = 1'b1;
end
```
- module fooReg;
```
module fooReg;
  input wire newFoo;
  output reg foo;

  initial #0 foo = 1'b0;

  always @(posedge clk)
    if (reset == 1'b1)
      foo <= 1'b0;
  else
      foo <= newFoo;
endmodule
```

Whew.

Questions?