Arch Lab

Warm Up

Chenlu Miao 09/2022

Overview

- Lab Sessions
- Vivado
- Verilog Warmup
- Verilog Event-Driven Simulation

Lab Sessions

Lab Sessions

- **Lab 01**
  - Implement a 5-staged pipelined CPU with forwarding and predicted-not-taken to support RV32I instructions.
- **Lab 02**
  - Implement handling of interrupt and exception on the pipelined CPU from Lab 01.
- **Lab 03**
  - Implement a two-way associative cache through simulation.
- **Lab 04**
  - Incorporate the two-way associative cache from Lab 03 to the pipelined CPU from Lab 02.
- **Lab 05**
  - Extend the pipelined CPU from Lab 02 to support multi-cycle operations, out-of-order execution, and hazard detection
- **Lab 06**
  - Extend the pipelined CPU from Lab 05 to support dynamic scheduling such as Scoreboarding or Tomasulo.

Vivado

Vivado

- New vivado users ? → click this
- Vivado 2020.2 and higher versions
- Our Board → **xc7k325tffg676-2L**

Verilog Warmup

Verilog Warmup

- variable
- vector
- assignment
- always block
- if-else
- case
- ROM & RAM

Verilog Practice in HDLBits: https://hdlbits.01xz.net/wiki/Main_Page

Verilog – Variable

- **wire**: Physically a wire. Can not "hold" value. Must be continuously assigned.
- **reg** : Not necessarily a register. Can "hold" value. Can be conditionaly assigned.

Verilog: wire vs. reg →

https://inst.eecs.berkeley.edu/~cs150/Documents/Nets.pdf

Verilog – Vector

- **Concatenation**

```
input [15:0] in;
output [23:0] out;
//left
assign {out[7:0], out[15:8]} = in;
//right
assign out[15:0] = {in[7:0], in[15:8]};
assign out = {in[7:0], in[15:8]};
```

Verilog – Vector

- **Concatenation**
- **Reverse**

```
module top_module(
    input [7:0] in,
    output [7:0] out
);
    // assign out[7:0] = in[0:7]; wrong
    assign out = {in[0], in[1], in[2], in[3], in[4], in[5], in[6], in[7]};
endmodule
```

Verilog – Vector

- **Concatenation**
- **Reverse**
- **Replication**

```
{5{1'b1}} // 5'b11111 (or 5'd31 or 5'h1f)
{2{a,b,c}} // The same as {a,b,c,a,b,c}
{3'd5, {2{3'd6}}} // 9'b101_110_110
```

`{num{vector}}`

This replicates vector by num times. num must be a constant

Verilog – Vector

- **Concatenation**
- **Reverse**
- **Replication**
- **Reduction**

```
& a[3:0]      // AND: a[3]&a[2]&a[1]&a[0].
Equivalent to (a[3:0] == 4'hf)
| b[3:0]      // OR:  b[3]|b[2]|b[1]|b[0].
Equivalent to (b[3:0] != 4'h0)
^ c[2:0]      // XOR: c[2]^c[1]^c[0]
```

Verilog – Assignment

- **Continuous assignment** (assign x = y;)
- **Procedural blocking assignment** (x = y;)
- **Procedural non-blocking assignment** (x <= y;)

Verilog – Assignment

- **Continuous assignment**

Example:

```
wire x;
assign x = y;
```

Requirements:

- Can only be used when not inside a procedure ("always block")
- The left-hand-side of an assign statement must be a net type (e.g., wire)

Verilog – Assignment

- **Procedural blocking assignment**

Example:

```
reg x;
always @(*) x = y;
```

Requirements:

- Can only be used inside a procedure.
- The left-hand-side of a procedural assignment (in an always block) must be a variable type (e.g., reg)

Verilog – Assignment

- **Procedural non-blocking assignment**

Example:

```
reg x;
always @(posedge clk) x <= y;
```

Requirements:

- Can only be used inside a procedure.
- The left-hand-side of a procedural assignment (in an always block) must be a variable type (e.g., reg)

Verilog – Always Block

- **Combinational**: `always @(*)`
- **Clocked**: `always @(posedge clk)`

Verilog – Always Block

- **Combinational**:

  `always @(*)`

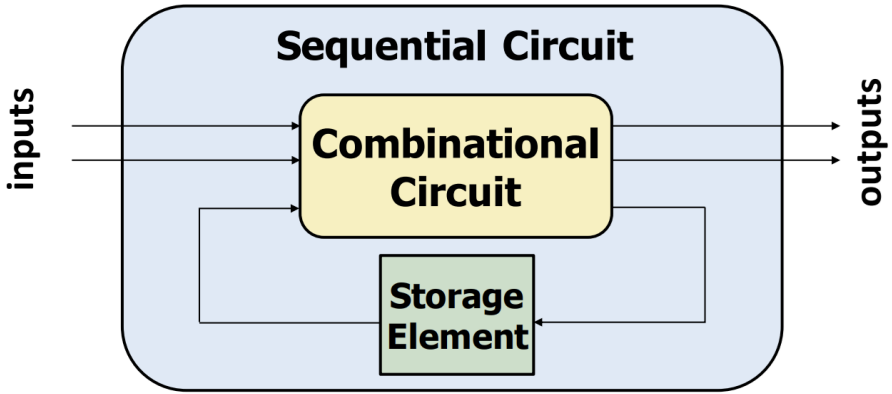
  - Equivalent to assign statements
  - Procedural blocks have a richer set of statements (e.g., if-then, case) …

```
reg [3:0] out;
always @(*) begin
    if(sel1)
        out = in1[3:0];
    else if(sel2)
        out = in2[3:0];
    else if(sel3)
        out = in3[3:0];
    else
        out = 4'b0;
end
```

```
wire [3:0] out;
assign out = ({4{sel1}} & in1[3:0])
           | ({4{sel2}} & in2[3:0])
           | ({4{sel3}} & in3[3:0]);
```

Verilog – Always Block

- Combinational: `always @(*)`
- **Clocked**: `always @(posedge clk)`




```
reg [2:0] q;

always @(posedge clk)
begin
    if(reset) q <= 0;
    else begin
        q <= q + 1;
    end
end
```

Verilog – Always Block

- Combinational: `always @(*)`
- Clocked: `always @(posedge clk)`

⚠️ Always use

blocking assignment in **combinational logic** `always @(*)`

non-blocking assignment in **sequential logic** **`always @(posedge clk)`**.

Verilog – if-else

```
if(sel1)
    out = in1[3:0];
else if(sel2)
    out = in2[3:0];
else if(sel3)
    out = in3[3:0];
else
    out = 4'b0;
```

👍 **Prefer**

```
assign out = sel1 ? in1[3:0] :
             sel2 ? in2[3:0] :
             sel3 ? in3[3:0] :
                    4'b0;

assign out = ({4{sel1}} & in1[3:0])
           | ({4{sel2}} & in2[3:0])
           | ({4{sel3}} & in3[3:0]);
```

Verilog – case

```
always @(*) begin      // This is a combinational circuit
    case (in)
      1'b1: begin
                out = 1'b1;  // begin-end if >1 statement
            end
      1'b0: out = 1'b0;
      default: out = 1'bx;
    endcase
end
```

Verilog – ROM & RAM

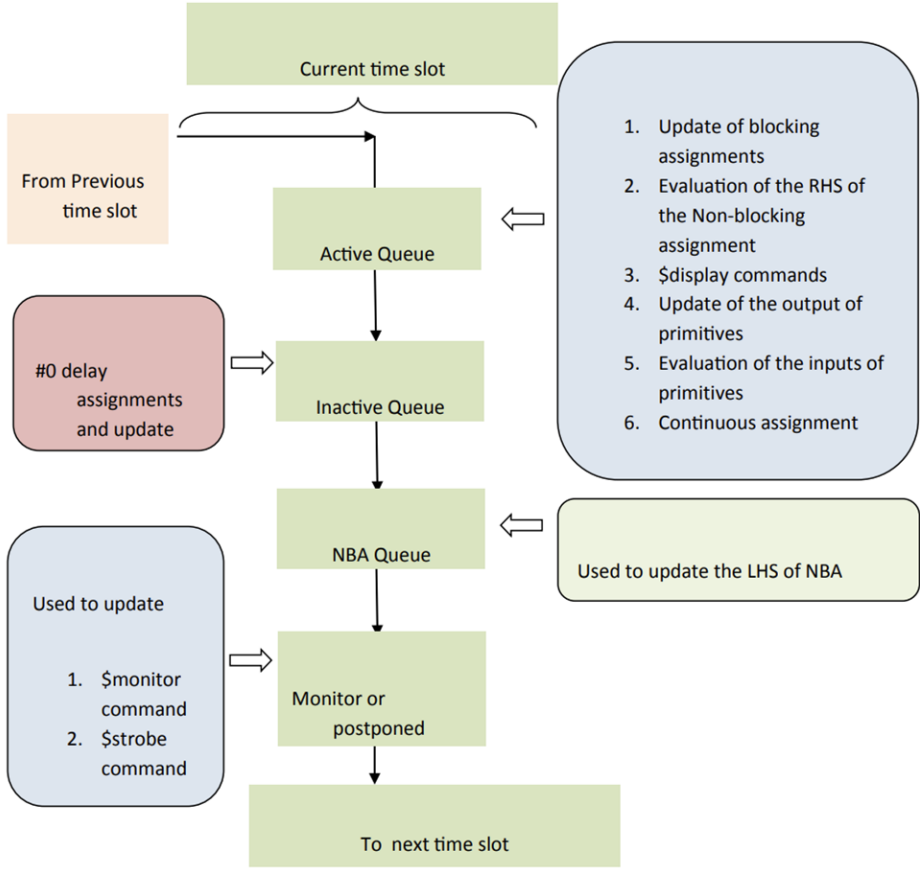
```
module ROM (
    input  [ 6:0] addr,
    output [31:0] rd_data
);
    reg [31:0] mem[127:0];

    initial begin
        $readmemh("rom.hex", mem);
    end

    assign rd_data = mem[addr];
endmodule
```

⚠ Put rom.hex in xxx\sim_1\behav\xsim dir
or use absolute path

Verilog – Event-Driven Simulation

Verilog – Stratified Event Queue

- **Active Queue**: Most of the Verilog events are scheduled in the active event queue. These events can be scheduled in any order and evaluated or updated in any order. The active queue is used to update the blocking assignments, continuous assignments, evaluation of RHS of the non-blocking assignments (LHS of NBA is not updated in the active queue), $display commands and to update the primitives.
- **Inactive Queue**: The #0delay assignments are updated in the inactive queue. Use of #0 delays in the Verilog is not good practice, and it unnecessarily complicates the event scheduling and ordering. Most of the times the designer uses the #0 delay assignments to fool the simulator to avoid the race around conditions.

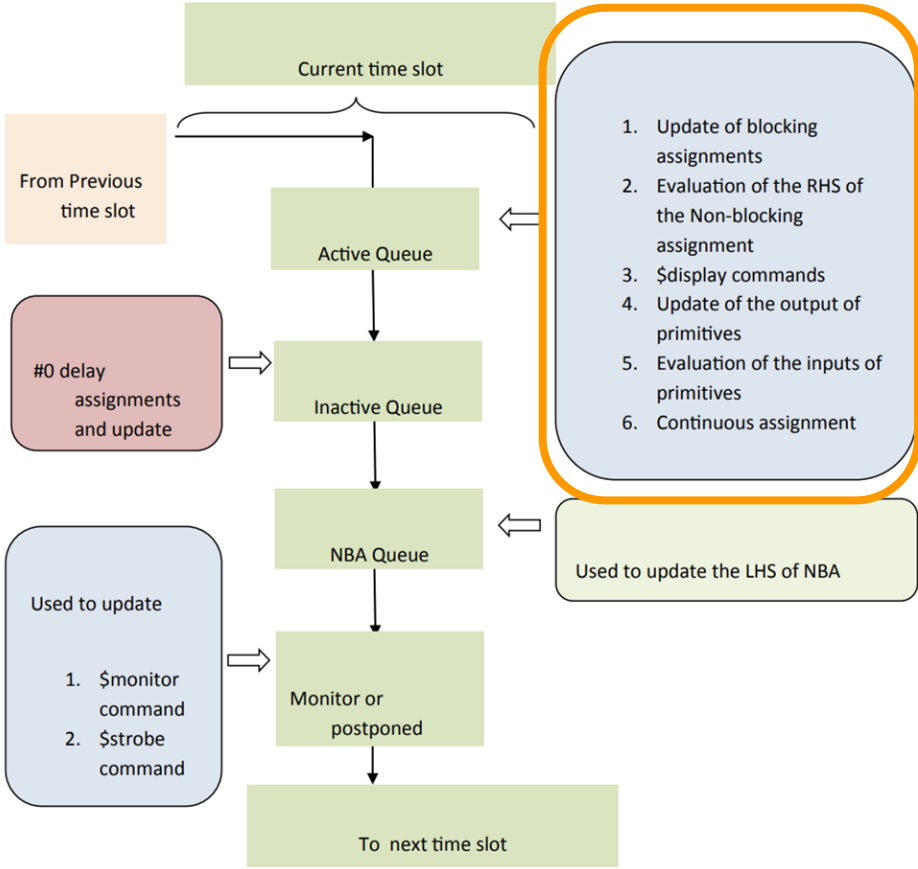


Verilog – Stratified Event Queue

```
while (there are events) {
    if (no active events) {
        if (there are inactive events) {
            activate all inactive events;
        } else if (there are nonblocking assign update events) {
            activate all nonblocking assign update events;
        } else if (there are monitor events) {
         activate all monitor events;
        } else {
            advance T to the next event time;
            activate all inactive events for time T;
        }
    }
    E = any active event;
    if (E is an update event) {
        update the modified object;
        add evaluation events for sensitive processes to event queue;
    } else { /* shall be an evaluation event */
        evaluate the process;
        add update events to the event queue;
    }
}
```

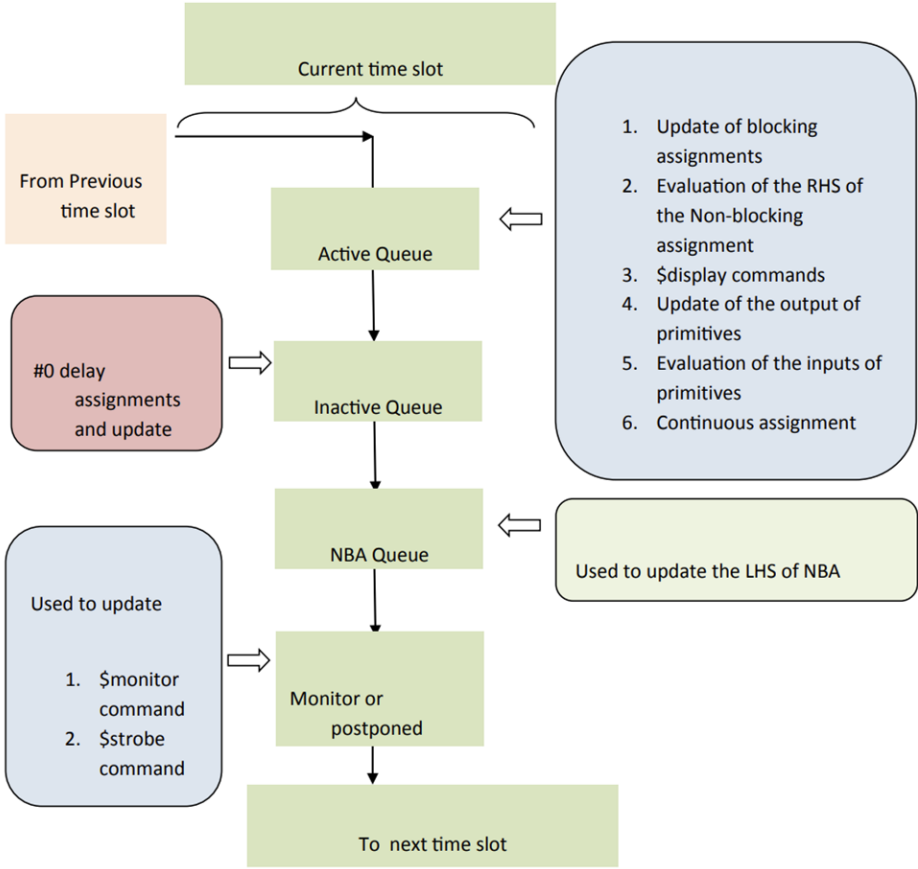
Verilog – Stratified Event Queue

- **Active Queue**: Most of the Verilog events are scheduled in the active event queue. These events can be scheduled in any order and evaluated or updated in any order. The active queue is used to update the blocking assignments, continuous assignments, evaluation of RHS of the non-blocking assignments (LHS of NBA is not updated in the active queue), $display commands and to update the primitives.
- **Inactive Queue**: The #0delay assignments are updated in the inactive queue. Use of #0 delays in the Verilog is not good practice, and it unnecessarily complicates the event scheduling and ordering. Most of the times the designer uses the #0 delay assignments to fool the simulator to avoid the race around conditions.



Verilog – Stratified Event Queue

- **NBA (Non-Blocking Assign Update) Queue**: The LHS of the non-blocking assignments updates in this queue.
- **Monitor events**: It is used to evaluate and update the $monitor and $strobe commands. The updates of all the variables are during the current simulation time.



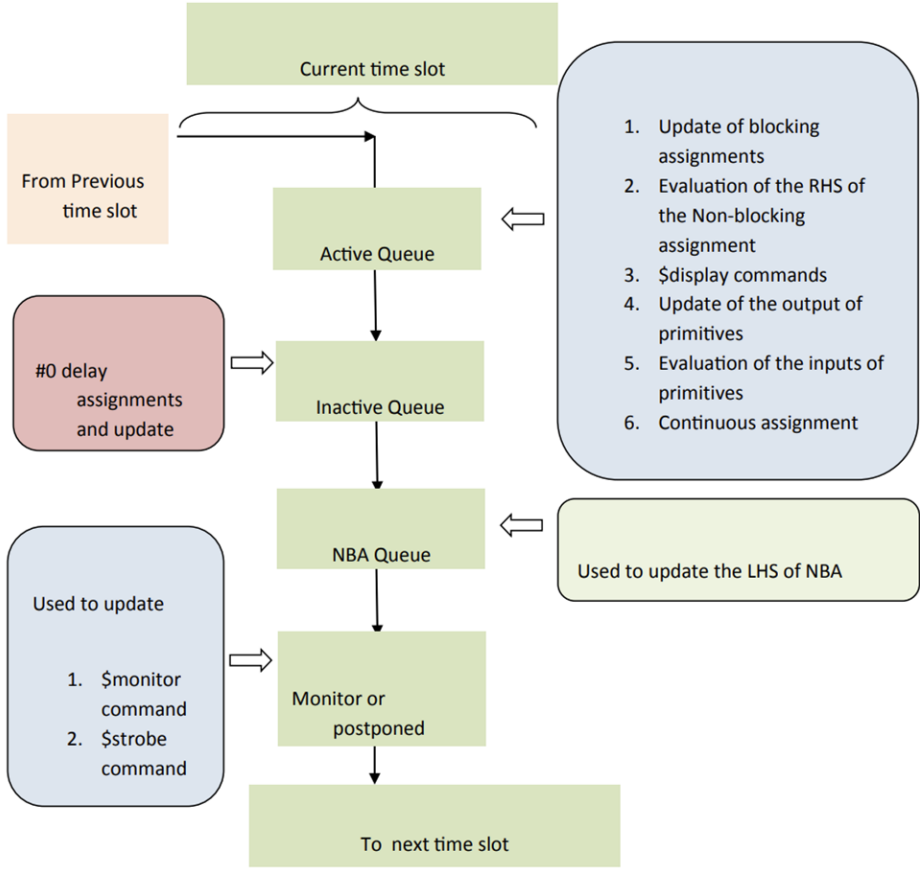
Verilog – Stratified Event Queue

- **Active Queue**:

```
always @(*)
    f = a & b;
always @(*)
    f = a | b;
```

execute in any order!

f = ?



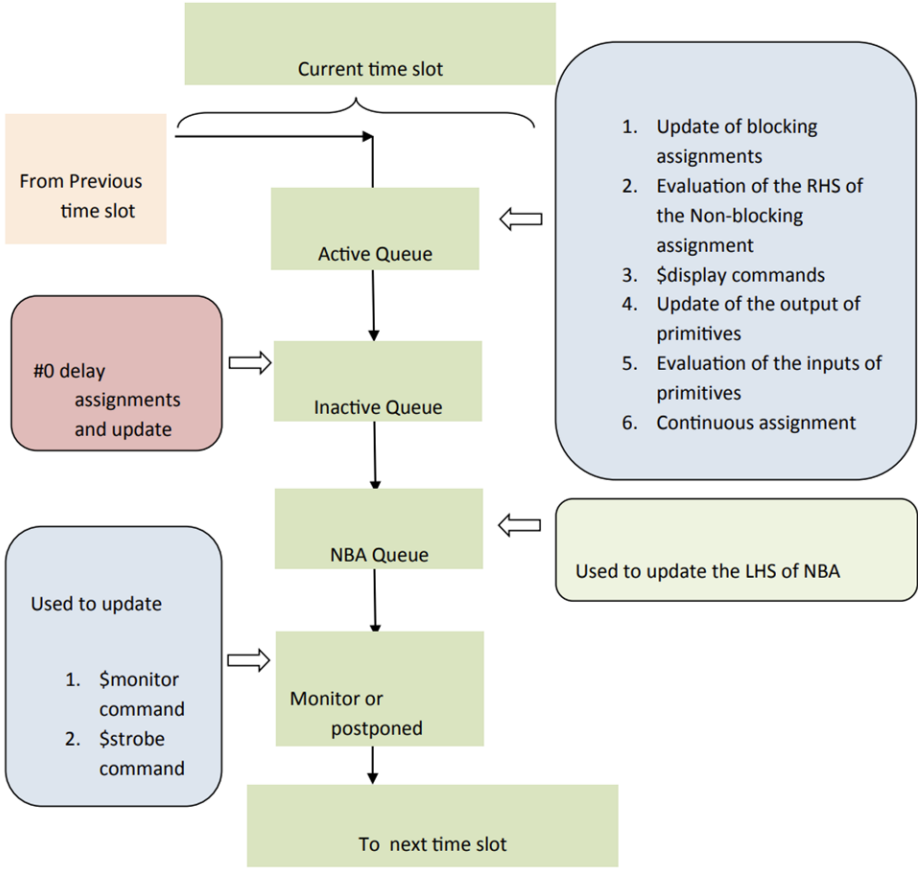
Verilog – Stratified Event Queue

- **Active Queue**:

```
always @(*)
begin
    f = a & b;
    f = a | b;
end
```

execute in order!

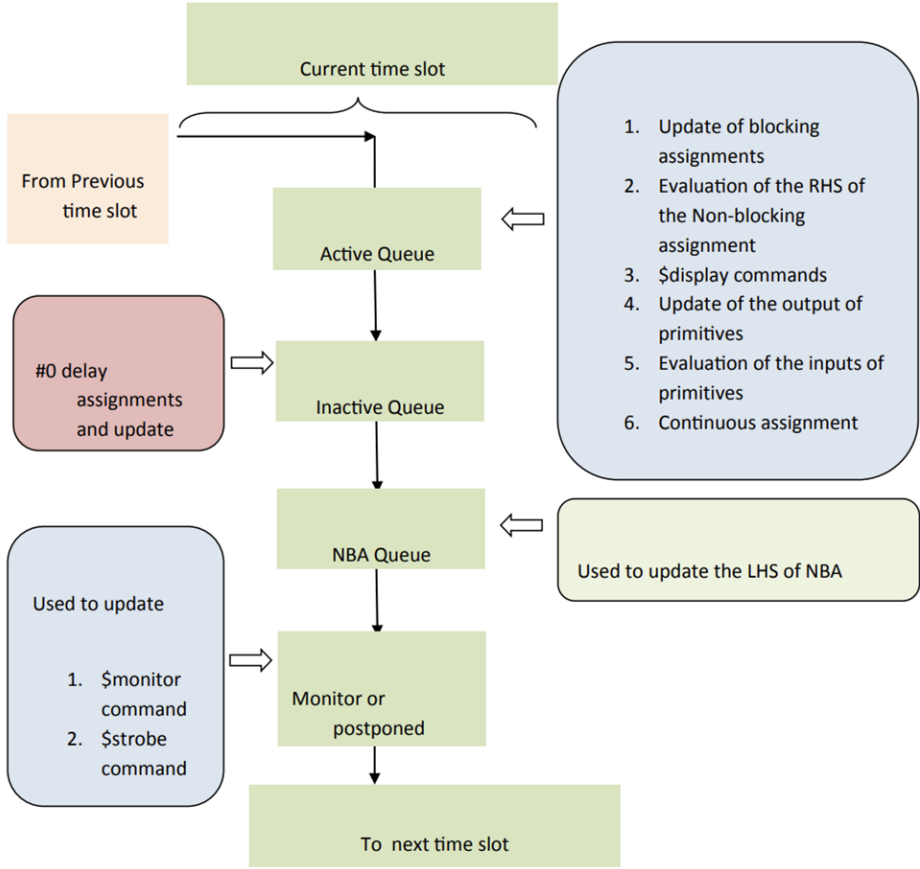
1. f = a & b
2. f = a | b



Verilog – Stratified Event Queue

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

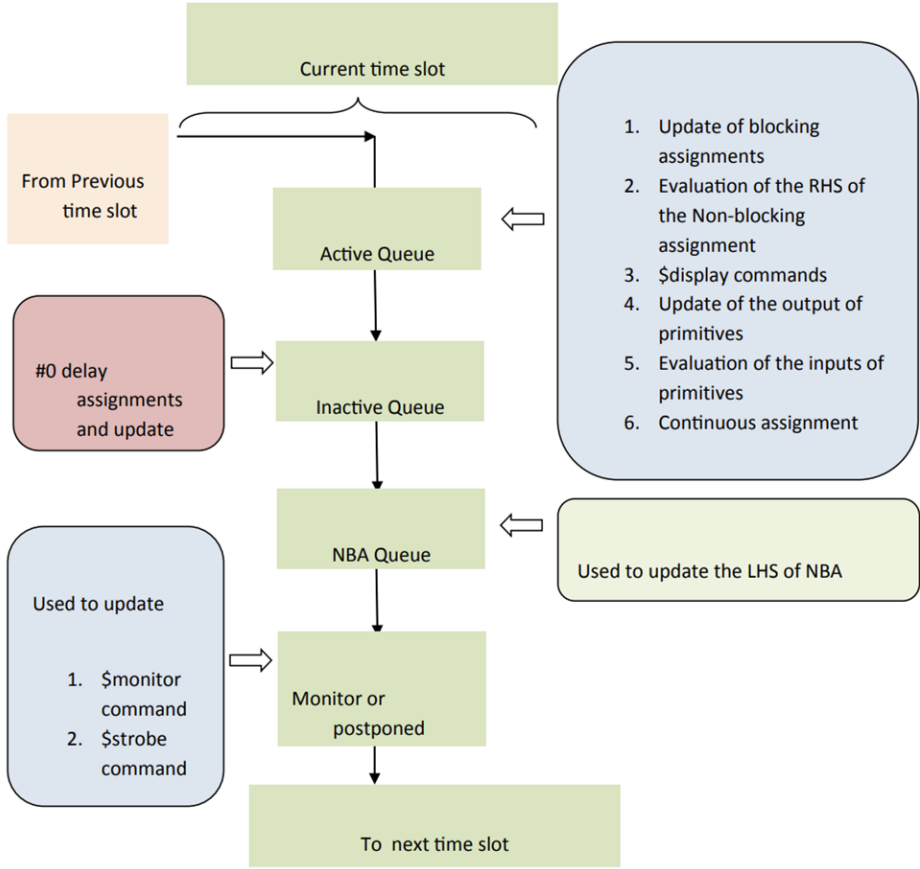
Active Queue: RHS Evaluation



Verilog – Stratified Event Queue

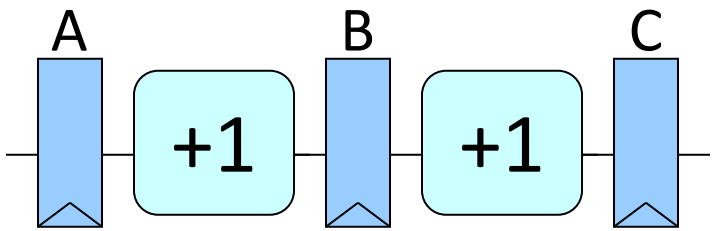
```
always @(posedge clk)
  f <= a & b;
```

NBA Queue: LHS Update



Verilog – Stratified Event Queue

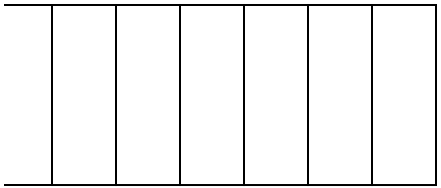
```
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @( posedge clk )
  A_out = A_in;

assign B_in = A_out + 1;

always @( posedge clk )
  B_out = B_in;

assign C_in = B_out + 1;

always @( posedge clk )
  C_out = C_in;
```



Verilog – Stratified Event Queue

```
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @( posedge clk )
  A_out = A_in;

assign B_in = A_out + 1;

always @( posedge clk )
  B_out = B_in;

assign C_in = B_out + 1;

always @( posedge clk )
  C_out = C_in;
```

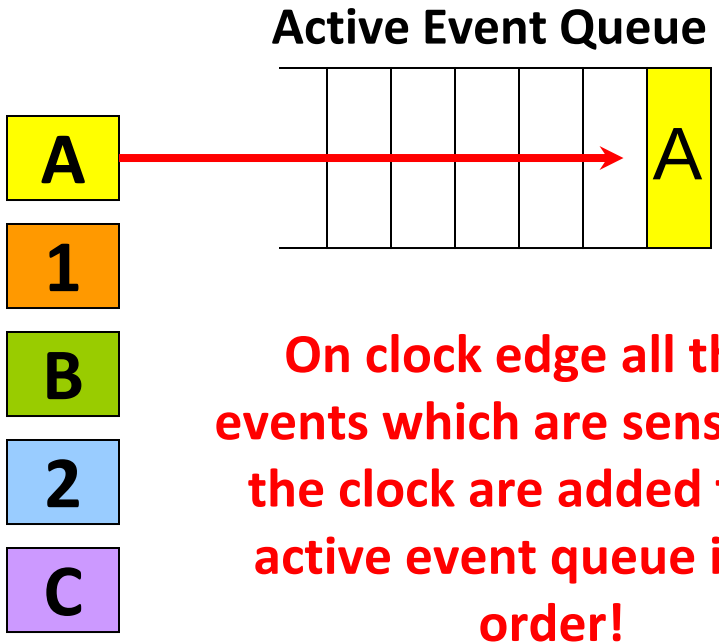
Active Event Queue



On clock edge all those events which are sensitive to the clock are added to the active event queue in any order!

Verilog – Stratified Event Queue

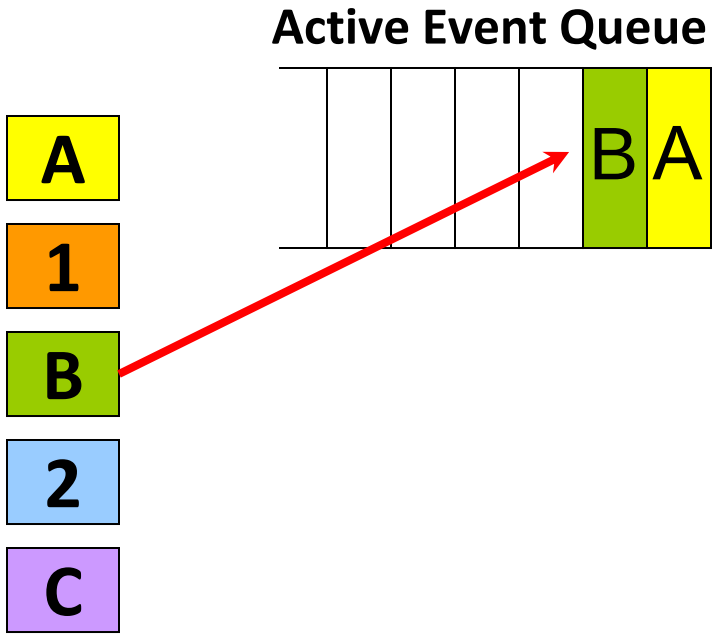
```
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @( posedge clk )
  A_out = A_in;

assign B_in = A_out + 1;

always @( posedge clk )
  B_out = B_in;

assign C_in = B_out + 1;

always @( posedge clk )
  C_out = C_in;
```



On clock edge all those events which are sensitive to the clock are added to the active event queue in any order!

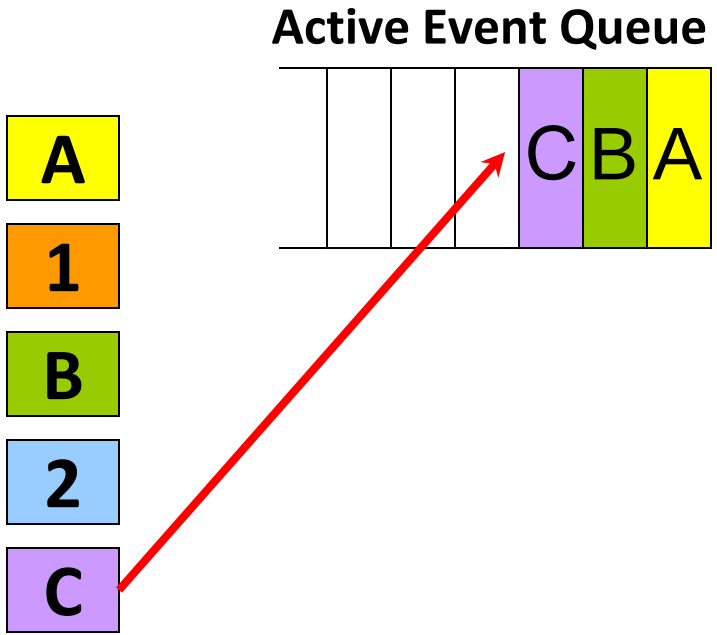
Verilog – Stratified Event Queue

```
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @( posedge clk )
  A_out = A_in;

assign B_in = A_out + 1;

always @( posedge clk )
  B_out = B_in;

assign C_in = B_out + 1;

always @( posedge clk )
  C_out = C_in;
```



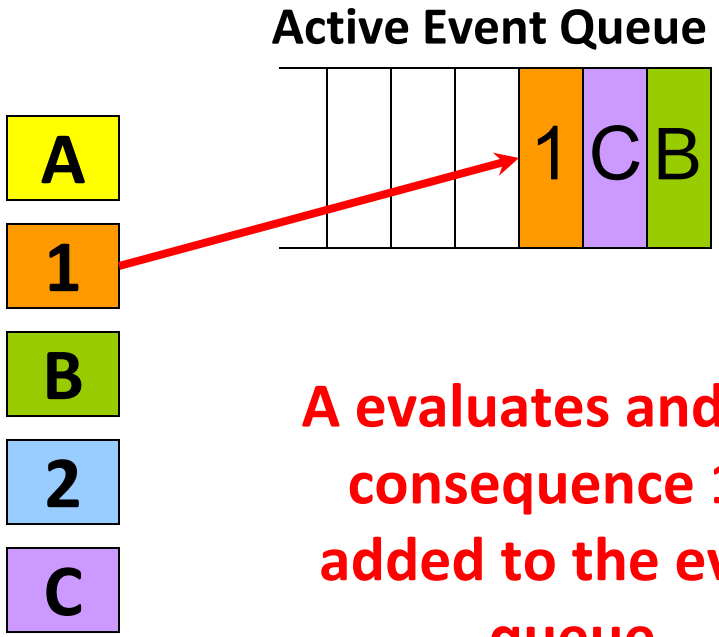
Verilog – Stratified Event Queue

```
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @( posedge clk )
  A_out = A_in;

assign B_in = A_out + 1;

always @( posedge clk )
  B_out = B_in;

assign C_in = B_out + 1;

always @( posedge clk )
  C_out = C_in;
```



Verilog – Stratified Event Queue

```
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @( posedge clk )
  A_out = A_in;

assign B_in = A_out + 1;

always @( posedge clk )
  B_out = B_in;

assign C_in = B_out + 1;

always @( posedge clk )
  C_out = C_in;
```

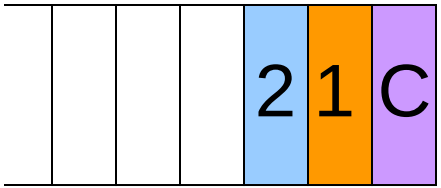


A evaluates and as a consequence 1 is added to the event queue

Verilog – Stratified Event Queue

```
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @( posedge clk )
  A_out = A_in;

assign B_in = A_out + 1;

always @( posedge clk )
  B_out = B_in;

assign C_in = B_out + 1;

always @( posedge clk )
  C_out = C_in;
```

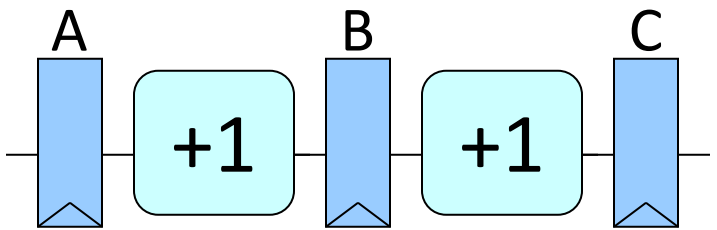
Active Event Queue

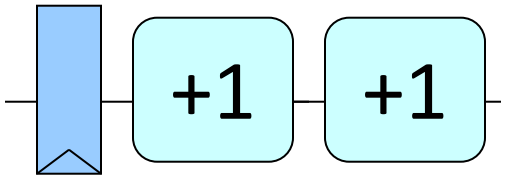


Event queue is emptied before we go to next clock cycle

Verilog – Stratified Event Queue

```
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @( posedge clk )
  A_out = A_in;

assign B_in = A_out + 1;

always @( posedge clk )
  B_out = B_in;

assign C_in = B_out + 1;

always @( posedge clk )
  C_out = C_in;
```

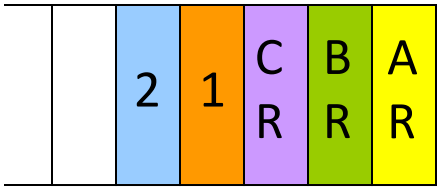




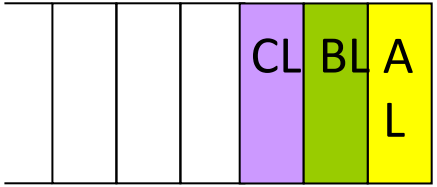
Verilog – Stratified Event Queue

```
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @( posedge clk )
  A_out <= A_in;

assign B_in = A_out + 1;

always @( posedge clk )
  B_out <= B_in;

assign C_in = B_out + 1;

always @( posedge clk )
  C_out <= C_in;
```

Active Event Queue



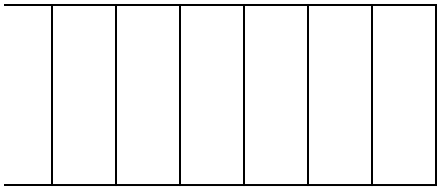
Non Blocking Queue



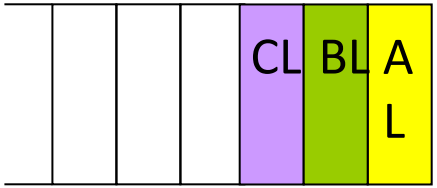
Verilog – Stratified Event Queue

```
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @( posedge clk )
  A_out <= A_in;

assign B_in = A_out + 1;

always @( posedge clk )
  B_out <= B_in;

assign C_in = B_out + 1;

always @( posedge clk )
  C_out <= C_in;
```

Active Event Queue



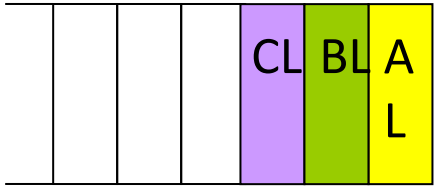
Non Blocking Queue



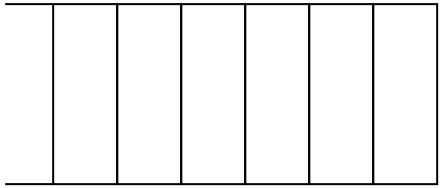
Verilog – Stratified Event Queue

```
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @( posedge clk )
  A_out <= A_in;

assign B_in = A_out + 1;

always @( posedge clk )
  B_out <= B_in;

assign C_in = B_out + 1;

always @( posedge clk )
  C_out <= C_in;
```

Active Event Queue

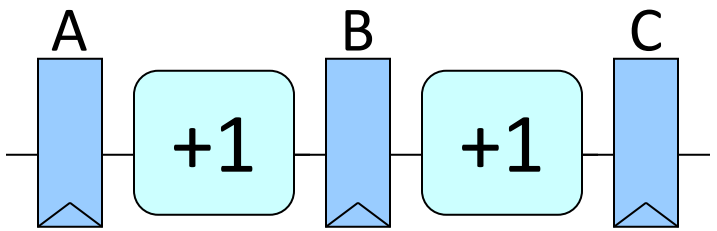


Non Blocking Queue



Verilog – Stratified Event Queue

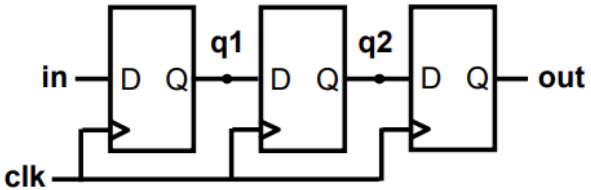
```
wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @( posedge clk )
  A_out <= A_in;

assign B_in = A_out + 1;

always @( posedge clk )
  B_out <= B_in;

assign C_in = B_out + 1;

always @( posedge clk )
  C_out <= C_in;
```



Verilog – Stratified Event Queue

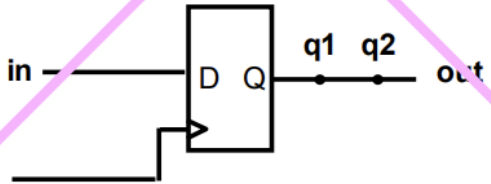
```
always @ (posedge clk)
begin
  q1 <= in;
  q2 <= q1;
  out <= q2;
end
```

"At each rising clock edge, *q1*, *q2*, and *out* simultaneously receive the old values of *in*, *q1*, and *q2*."




```
always @ (posedge clk)
begin
  q1 = in;
  q2 = q1;
  out = q2;
end
```

"At each rising clock edge, *q1* = *in*. After that, *q2* = *q1* = *in*; After that, *out* = *q2* = *q1* = *in*; Finally *out* = *in*."



- **Blocking assignments do not reflect the intrinsic behavior of multi-stage sequential logic**

https://courses.csail.mit.edu/6.111/f2007/handouts/L06.pdf

References

- Verilog General Guide
- https://safari.ethz.ch/digitaltechnik/spring2021/lib/exe/fetch.php?media=onur-digitaldesign_comparch-2021-lecture7-hdl-verilog-afterlecture.pdf
- https://safari.ethz.ch/digitaltechnik/spring2021/lib/exe/fetch.php?media=onur-digitaldesign_comparch-2021-lecture6-sequential-logic-updated-beforelecture.pdf
- HDLBits
- 从仿真器的角度理解Verilog语言
- RISC-V CPU处理器设计
- Digital Logic Design Using Verilog Coding and RTL Synthesis Second Edition
- https://cseweb.ucsd.edu/classes/sp09/cse141L/Slides/02-Verilog2.pdf
- https://electronics.stackexchange.com/questions/443641/why-we-need-non-blocking-assignments-in-verilog
- https://courses.csail.mit.edu/6.111/f2007/handouts/L06.pdf
- IEEE Standard for Verilog Hardware Description Language