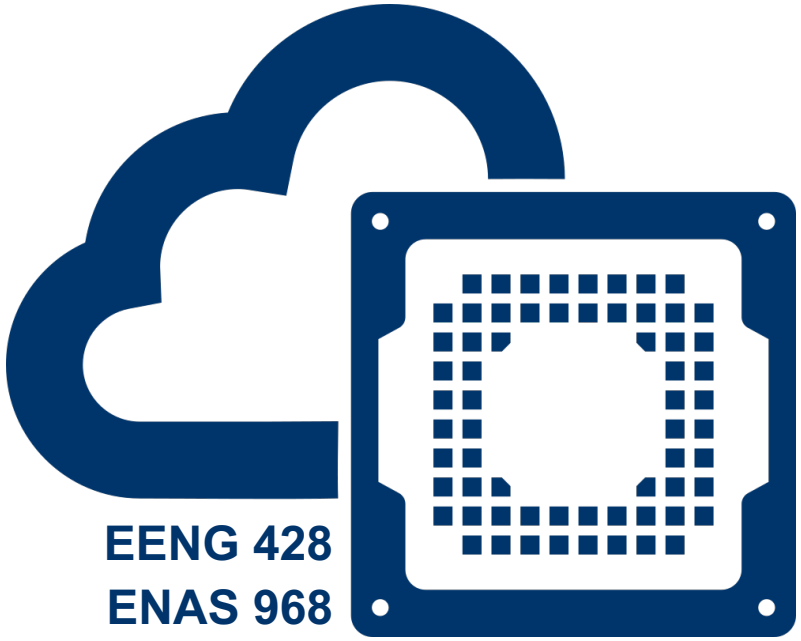
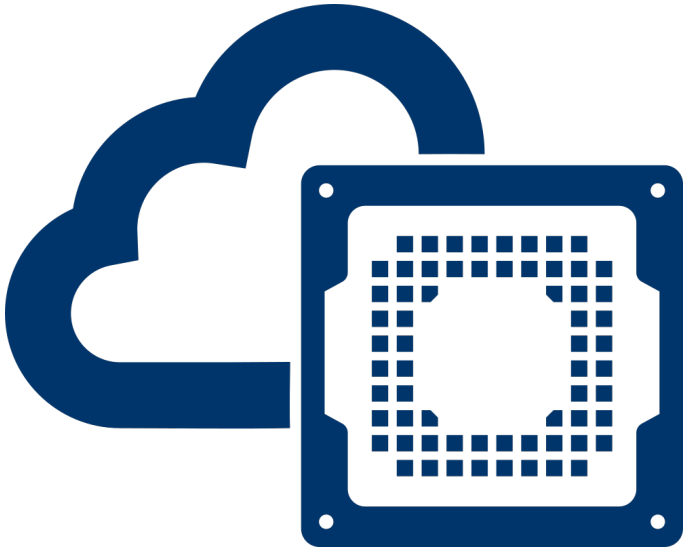
Cloud FPGA



bit.ly/cloudfpga



Lecture: Concurrent Processes

Prof. Jakub Szefer
Dept. of Electrical Engineering, Yale University

EENG 428 / ENAS 968
Cloud FPGA

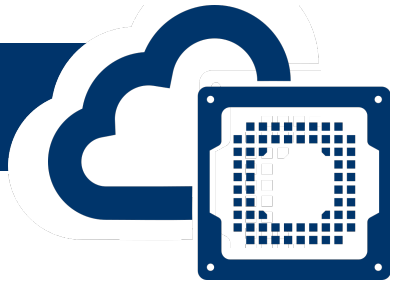
Logic Synthesis

This lecture is mostly based on contents of Chapter 4, from "The Verilog Hardware Description Language" book [1], 5th edition. Example figures and (modified) code are from the textbook unless otherwise specified.

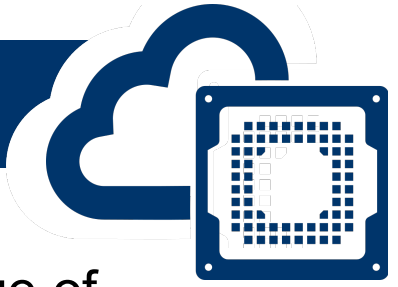
Topics covered:

- Synchronization between concurrent processes
- Events
- The `wait` statement
- Producer-consumer examples
- The `disable` statements
- Parallel blocks with `fork-join` statements

Concurrent Processes



Concurrent Processes



- A process in an “abstraction of a controller, a thread of control that evokes the change of values stored in the systems registers” [1]
- A digital system can be thought of as a set of communicating, concurrent processes that pass information among themselves
  - Each process contains state information → values eventually stored in hardware registers
  - The state is modified based on the process’ inputs and current state
  - A module contains one or more concurrent processes
  - A system is typically made of one or more modules

Recall that there can be module with no processes, just structural connection between modules

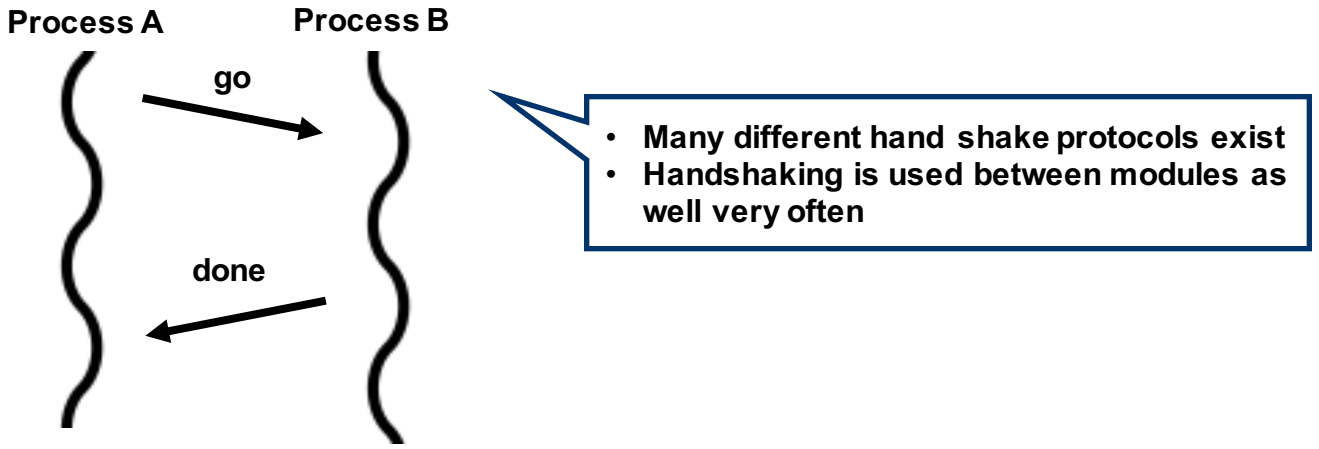
- Each process effectively represents a state machine
  - Combinatorial logic is effectively state machine with 0 states
- Example: if there two processes that have `n` and `m` states, then a combined process would have in the worst case `n * m` states

Could describe a processor using one process, but it would be very messy, so we break it into multiple processes

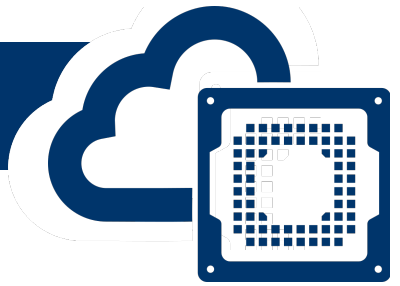
```
module computer;
  always
  begin
    powerOnInitializations;
    forever
    begin
      fetchAndExecuteInstructions;
    end
  end
endmodule
```

Synchronization Between Processes

- "When several processes exists in a system and information is to be passed among them, we must synchronize the processes to make sure that correct information is being passed" [1]
  - Each process is asynchronous with respect to each other
    - Can run on same or different clocks, but even if on same clock, one process does not know what state the other process is in → need to explicitly synchronize
- A **handshake** protocol is needed to synchronize processes



Events



- Event control statements in Verilog allow to take actions when an event happens

- **Value change events**
  - The `@` is used to specify value change events
  - Value change events, watch for changes in wires or registers
  - Stop procedure evaluation until there is an event
    - Positive edge, negative edge, or any value change
  - If previous and new value are same, then no event is triggered

```
module dEdgeFF
( output reg q,
  input clock, data
);

  always @(negedge clock)
    q <= data;

endmodule
```

- posedge or negedge watch for transition (0 → 1, 0 → x, or x → 1) and (1 → 0, 1 → x, or x → 0) respectively
- can OR many events to check of any of them
- order of listing the events does not matter

- **Named events**
  - Named events, watch for changes in events, not actual wires or registers in the design
  - Only really used for simulation not to write real hardware

```
module numberGen
( output reg [15:0] number = 0
);

  event ready;

  always
  begin
    #50 number = number + 1;
    #50 -> ready;
  end

endmodule
```

Declare event

Trigger event

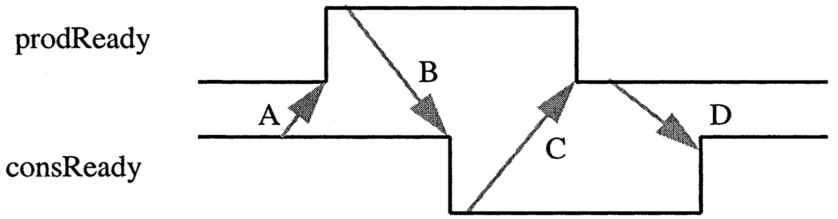
```
module fibNumCalc
( input      [15:0] startingValue,
  output reg [15:0] fibNum
);

  reg [15:0] count, oldNum, temp;

  always
  begin
    @ng.ready
      count = startingValue;

    // do some work...

  end

endmodule
```

Event change detecti on

`wait` Statements

- "The **`wait`** statement is a concurrent process statement that waits for its conditional expression to become TRUE" [1]
  - Process evaluation stops until the statement becomes true
- The **`wait`** statement can be use, for example, to make a handshake protocol:

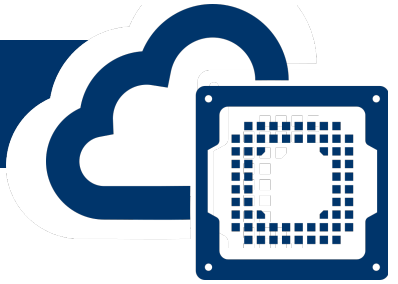
```
module ProducerConsumer;

  reg consReady, prodReady;
  reg [7:0] dataInCopy, dataOut;

  always // consumer process
  begin
    consReady = 1;

    forever
    begin

      wait (prodReady)
        dataInCopy = dataOut;

      consReady = 0;

      // do some work...

      wait (!prodReady)
        consReady = 1;

    end
  end
```




```
  always // produce process
  begin
    prodReady = 0;

    forever
    begin

      // do some work...

      wait (consReady)
        dataOut = $random;

      prodReady = 1;

      wait (!consReady)
        prodReady = 0;

    end
  end

endmodule
```

Verilog built-in function to generate random numbers

`wait` Statements vs. `while` Loops vs. Events

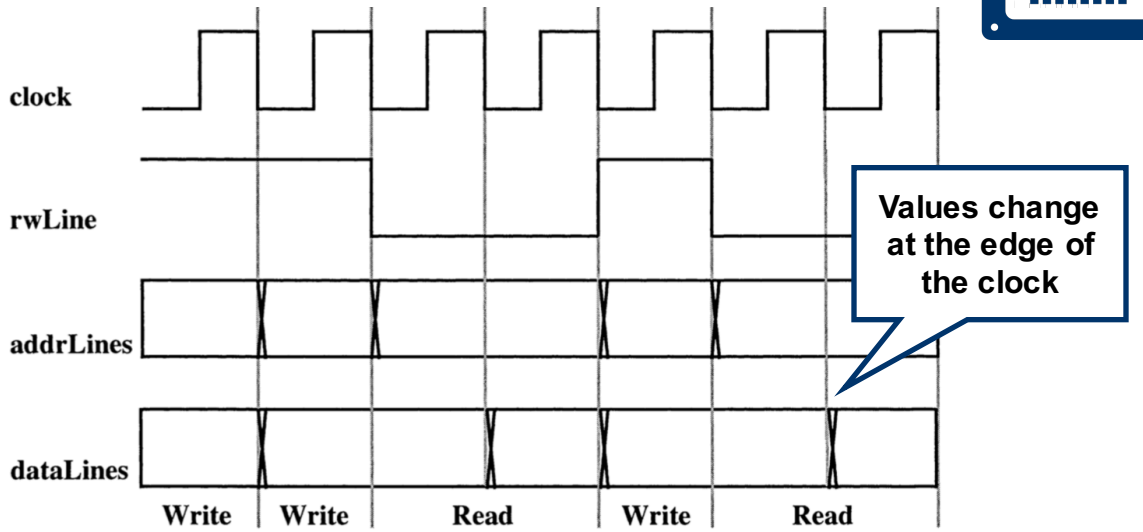


- The main difference between **`wait`** statements **`while`** loops:
  - The **`wait`** statement stops a process evaluation
  - The **`while`** loop keeps executing, does not stop process evaluation
    - The loop does not let stop the process, can get stuck in the process (other processes don't get to be evaluated so no progress can be made

- The main difference between **`wait`** statements and events:
  - Both check for situation or changed generated by other processes
  - Events are edge triggered
  - The **`wait`** statement is level triggered, once a wait is TRUE, it will stay so
    - Need another **`wait`** statement that is triggered when condition is FALSE, i.e. ! condition is TRUE

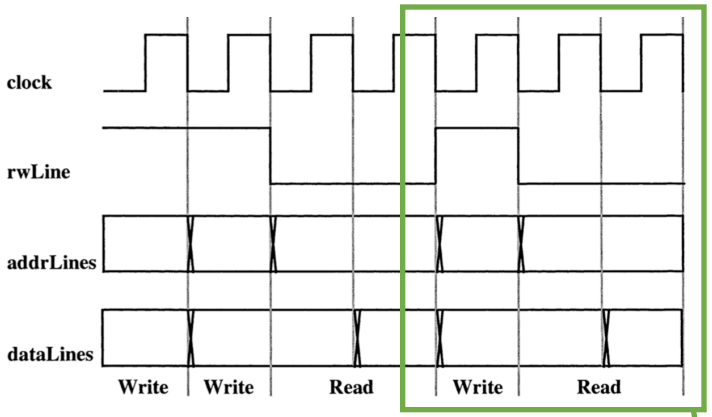
Producer-Consumer Example

- Textbook example of synchronous bus protocol
  - Master module controls rwLine and addrLine
    - rwLine == 1 means write, else it's a read
    - addrLine specifies address
  - The data line is driven by master for writes and by the slave for reads



Share: bit.ly/cloudfpga

Producer-Consumer Example



Simulation output:

```
rw=x, data=   x, addr= x at time    0
rw=0, data=   x, addr= 2 at time    1   > Read
rw=0, data=  29, addr= 2 at time   40
rw=0, data=  29, addr= 3 at time   80   > Read
rw=0, data=  28, addr= 3 at time  120
rw=1, data=   5, addr= 2 at time  160   — Write
rw=1, data=   7, addr= 3 at time  200   — Write
rw=0, data=   7, addr= 2 at time  240   > Read
rw=0, data=   5, addr= 2 at time  280
rw=0, data=   5, addr= 3 at time  320   > Read
rw=0, data=   7, addr= 3 at time  360
```

```
`define READ 0
`define WRITE 1

module sbus;
  parameter tClock = 20;

  reg clock;
  reg [15:0] m [0:31];
  reg [15:0] data

  reg rwLine
  reg [4:0] addressLines;
  reg [15:0] dataLines;

  initial
  begin
    $readmemh ("memory.data", m);
    clock = 0;
    $monitor ("rw=%d, data=%d, addr=%d at time %d",
              rwLine, dataLines, addressLines, $time);
  end

  always
    #tClock clock != clock;

  initial
  begin
    #1
    wiggleBusLines(`READ, 2, data);
    wiggleBusLines(`READ, 3, data);
    data = 5;
    wiggleBusLines(`WRITE, 2, data);
    data = 7;
    wiggleBusLines(`WRITE, 3, data);
    wiggleBusLines(`READ, 2, data);
    wiggleBusLines(`READ, 3, data);
    $finish;
  end
```

Macro defines

Module parameter

Read data from file into memory array

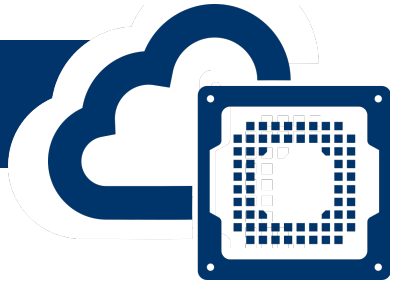
Forever toggle clock to simulate a periodic clock

Explicit simulation finish

```
  task wiggleBusLines
  ( input readWrite,
    input [5:0] addr,
    inout [15:0] data
  );
  begin
    rwLine <= readWrite;
    if (readWrite)
    begin
      addressLines <= addr;
      dataLines <= data;
    end
    else
    begin
      addressLines <= addr;
      @ (negedge clock);
    end
    @ (negedge clock);
    if (~readWrite)
      data <= dataLines;
  end
  endtask

  always
  begin
    @ (negedge clock);
    if (~rwLine)
    begin
      dataLines <= m[addressLines];
      @ (negedge clock);
    end
    else
      m[addressLines] <= dataLines;
  end
endmodule
```


Share: bit.ly/cloudfpga

Simple Processor Example



- Very simplified Mark-1 processor example from the textbook
  - Uses events instead of wait statements
  - More similar to typical synthesizable logic writing style

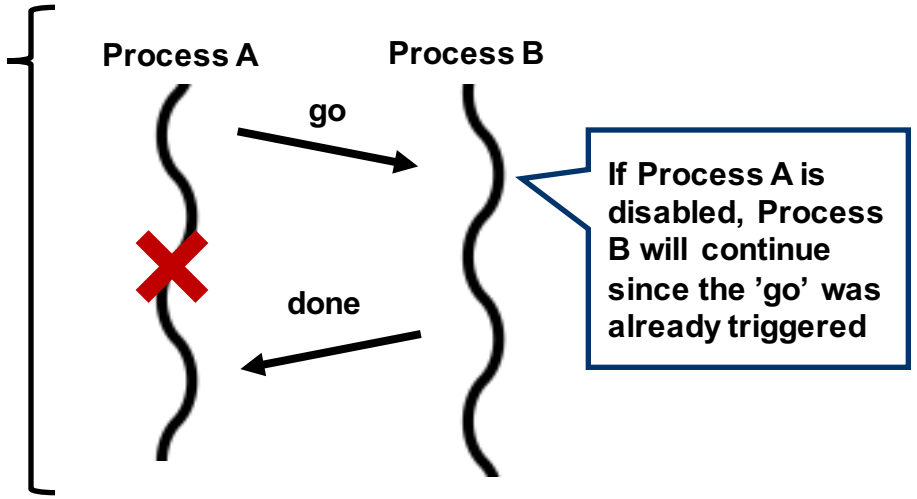
Each register can only be written in one always block

```
module mark1PipeStage;

  reg [15:0] signed m [0:8191];
  reg [15:0] signed pc;
  reg [15:0] signed acc;
  reg [15:0] ir
  reg ck, skip;

  always @ (posedge ck) // Fetch instructions
  begin
    if (skip)
      pc <= pctemp;
      ir <= m[pc];
      pc <= pc + 1;
  end

  always @ (posedge ck) // Execute instructions
  begin
    if (skip)
      skip <= 0;
    else
      case (ir[15:13])
        3'b000: begin
                  pctemp <= m[ir[12:0]];
                  skip <= 1;
                end
        3'b001: begin
                  pctemp <= pc + m[ir[12:0]];
                  skip <= 1;
                end
        3'b010: acc <= -m[ir[12:0]];
        3'b011: m[ir[12:0]] <= acc;
        3'b100,
        3'b101: acc <= acc - m[ir[12:0]];
        3'b110: if (acc < 0) begin
                  pctemp <= pc + 1;
                  skip <= 1;
                end
      endcase
  end
endmodule
```

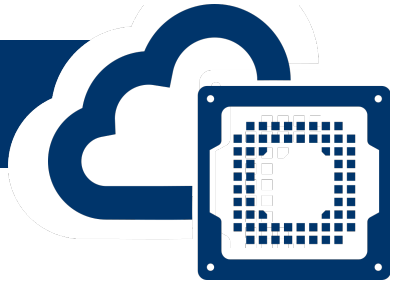

Share: bit.ly/cloudfpga

Disabling Named Blocks

- The **`disable`** statement can be used to break out of a loop statement
- The **`disable`** statement can also be used in concurrent processes
  - Stop any named begin-end block
  - Stop any functions or tasks called from the block
  - Execution continues with the next statement following the end of the block
  - If another process was triggered by the stopped block, it will continue



Disabling Named Blocks Example



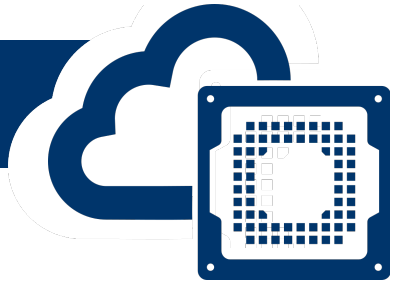
- Textbook example of disabling named blocks

```
module simpleTutorialWithRest
( input clock, reset,
  output reg [7:0] y, x
);

  initial
    forever
    begin
      @(negedge reset)
        disable main;
    end

  always
  begin: main

    wait (reset);

    @(posedge clock)
      x <= 0;

    i = 0;

    while (i <= 10)
    begin
      @(posedge clock);
      x <= x + y;
      i = i + 1;
    end

    @(posedge clock);
    if (x < 0)
      y <= 0;
    else
      x <= 0;

  end

endmodule
```

Trigger disable of main block

- Normally wait will get activated once when reset is TRUE
- Because of the disable, the main will restart and activate wait again

Intra-Assignment Control and Timing Events



- Most of the time the timing or event control is specified to occur before the action or assignment occurs
- But can also have intra-assignment control and timing
  - Works for blocking and non-blocking assignments
- Not used to describe synthesizable logic
  - Can be used for simulation

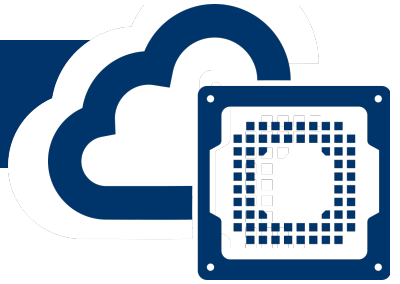
Statements using intra-assignment constructs	Equivalent statements without intra-assignments

a = #25 b;	begin
	bTemp = b;
	#25 a = bTemp;
	end

q = @(posedge w) r;	begin
	rTemp = r;
	@(posedge w)
	q = rTemp;
	end

w = repeat (2)
@(posedge clock) t;
	begin
	tTemp = t;
	repeat (2)
	@(posedge clock);
	w = tTemp;
	end

Procedural Continuous Assignment



- Continuous assignment is typically done with the **`assign`** statement
- Can use the assignment statements in procedural specification

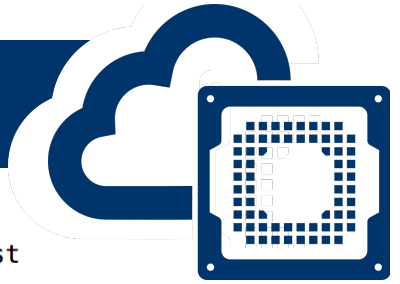
- Again, not used to synthesize real hardware because of the **#** delays

```
module dFlop
( input preset, clear,
  output reg q,
  input clock, d
);

  always
    @(clear, preset)
    begin
      if (!clear)
        #10 assign q = 0;
      else if (!preset)
        #10 assign q = 1;
      else
        #10 deassign q;
    end

  always
    @(negedge clock)
      q = #10 d;

endmodule
```

Set the value

Revert to value before last `assign`

Parallel Blocks with `fork-join` Statements



- Each statement in the **`fork-join`** block is a separate process that begins when control is passed to the **`fork`**
- The **`join`** waits for all of the processes to complete before continuing with the next statement beyond block

```
module microprocessor;

  always
  begin

    resetSequece;

    fork: mainWork

      forever
        fetchAndExecute Instructions;

      @(posedge reset)
        disable mainWork;

    join
  end

endmodule
```

```
module simpleTutorialWithRest
( input clock, reset,
  output reg [7:0] y, x
);

  reg [7:0] i;

  always
    fork: main

      @(negedge reset)
        disable main;

      begin
        wait (reset);

        @(posedge clock)
          x <= 0;

        i = 0;

        while (i <= 10)
        begin
          @(posedge clock);
          x <= x + y;
          i = i + 1;
        end

        @(posedge clock);

        if (x < 0)
          y <= 0;
        else
          x <= 0;

      end

    join

endmodule
```

References

1. Donald E. Thomas and Philip R. Moorby. " The Verilog Hardware Description Language, Fifth Edition." Springer. 2002