Unit 1
Basic Digital Logic

EL-GY 9463: INTRODUCTION TO HARDWARE DESIGN

PROFS. SUNDEEP RANGAN, SIDDHARTH GARG

Learning Objectives

- ❑Write simple functions with combinational logic in SystemVerilog and draw block diagrams
- ❑Define finite state machines and identify inputs, outputs, and states
- ❑Implement FSMs with synchronous logic in SystemVerilog
- ❑Break and implement functions over multiple clock cycles
- ❑Write a testbench for the hardware modules
- ❑Simulate, and synthesize the SystemVerilog using Vivado tools

Outline

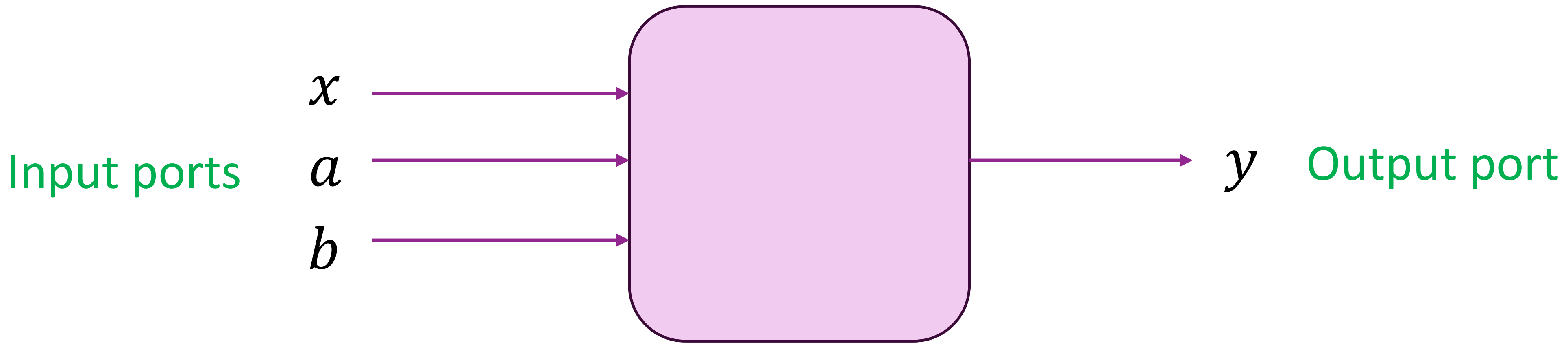
- ❑ Combinational Logic
- ❑ Registers, Clocks and Sequential Logic and Finite State Machines
- ❑ Splitting Long Computations
- ❑ Simulating and Synthesizing Simple Modules in Vivado

Simple Example: ReLU + Linear

Goal: Implement a module for the function

$$y = \max\{ax + b, 0\}$$

- Linear function + ReLU
- Used widely in machine learning



SystemVerilog Description

- SystemVerilog:
  - Language for behavioral description
  - What the module should do
- Each block is a module
  - Has inputs and outputs
  - Logical mapping from inputs to outputs
- Description is behavioral only
  - Does not say how it will be implemented
  - We discuss that later

```
module relu_lin (
  input logic signed [31:0] x,
  input logic signed [31:0] a,
  input logic signed [31:0] b,
  output logic signed [31:0] y
);
  always_comb begin
    logic signed [31:0] mult_out, add_out;
    mult_out = x * a;
    add_out = mult_out + b;
    y = (add_out > 0) ? add_out : 0;
  end
endmodule
```

Module Components in SV

```
module relu_lin (
  input logic signed [31:0] x,
  input logic signed [31:0] a,
  input logic signed [31:0] b,
  output logic signed [31:0] y
);
  always_comb begin
    logic signed [31:0] mult_out, add_out;
    mult_out = x * a;
    add_out = mult_out + b;
    y = (add_out > 0) ? add_out : 0;
  end
endmodule
```

- Module name
- Ports:
  - Defines inputs and outputs
  - Each has a type (more detail later)
- Behavioral description
  - In this case, a combinational block
  - Sequence of operations
  - From input to output

System Verilog vs Verilog

- Verilog
  - Created in 1983-84
  - Standardized in 1995. Became mainstream RTL
  - Largely used for legacy designs

- System Verilog (this unit)
  - Developed in 2002
  - Extends Verilog
  - Better abstraction, parameterization
  - Significantly improved testbench
  - Overwhelming design choice today

VERILOG

SystemVerilog

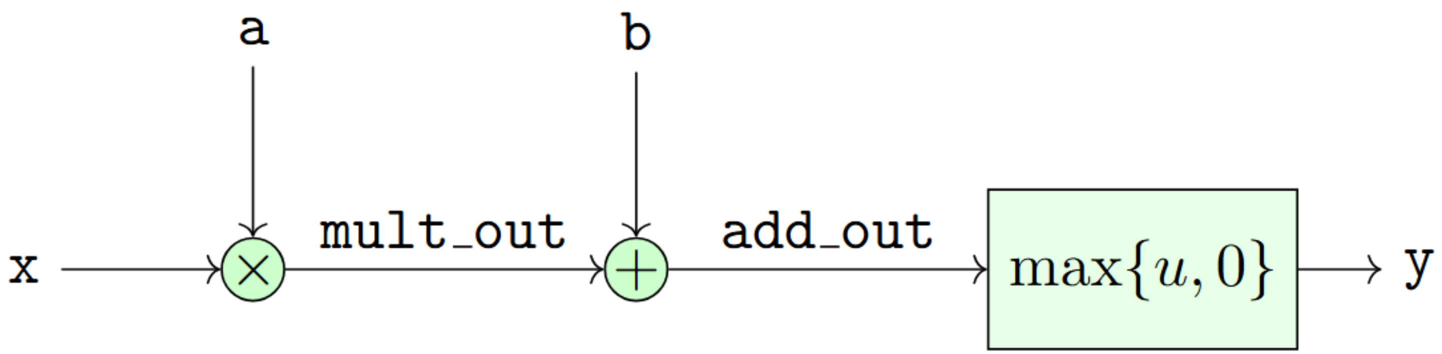
System Verilog Data Types

```
logic x;    // Single 4-state bit
logic [15:0] y;   // 16-bit unsigned integer
logic signed [31:0] s; // 32-bit signed integer
```

- Most of the modules we will write use the above three data types
- Specifies:
  - Bit width
  - Signed or unsigned
- Logic keyword:
  - Indicates that a bit can 0, 1, X or Z
  - X = unknown / undefined, Z = high impedance (disconnected)
  - We will discuss X and Z later

Block Diagram

- Function of a module typically represented by a block diagram
- Graphical representation of a potential implementation
- Typically, diagram is at high-level
  - Adders, multipliers not low-level details components like gates

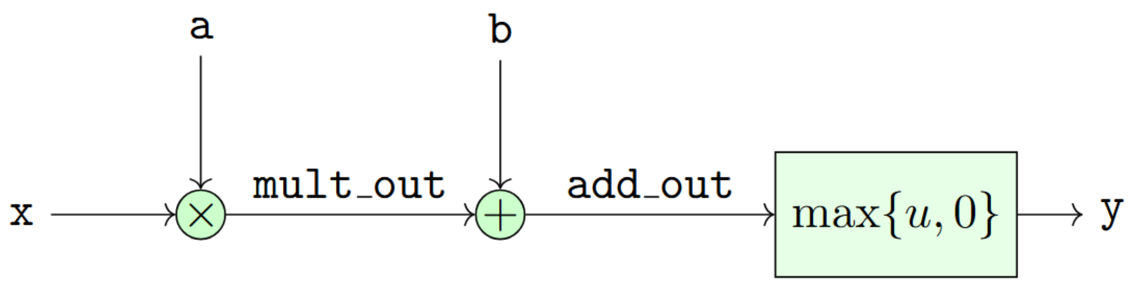


Outline

- Combinational Logic
- Registers, Clocks and Sequential Logic and Finite State Machines
- Splitting Long Computations
- Simulating and Synthesizing Simple Modules in Vivado

Limitations of Combinational Logic

- No memory — cannot store past values
- Output changes immediately with input changes (no control over timing)
- Long combinational paths lead to large propagation delays
- Hard to build large systems without controlled timing boundaries
- Glitches and hazards can propagate freely
- No notion of “steps” or “cycles” — everything is continuous



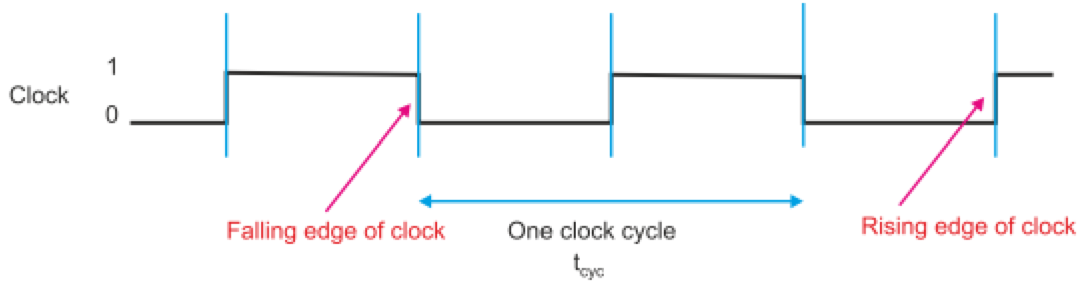
Synchronous (Sequential) Logic

- Introduces state via flip-flops or registers
- Breaks long logic into clocked stages, improving performance
- Provides predictable timing — changes only occur on clock edges
- Makes large systems modular and easier to reason about
- Eliminates most hazards/glitches from propagating across stages
- Enables pipelining, finite-state machines, and sequential algorithms
- Allows clean interfaces between modules (valid/ready, handshakes, etc.)

Synchronous logic is the dominant paradigm for design

Clock Signal

- **Clock**: An alternating binary signal
  - Visualized as a square wave
- Each period is called a clock cycle
  - Typically, between two rising edges
- Parameters:
  - Clock period T
  - Frequency $f = \frac{1}{T}$
- Typical parameters:
  - FPGA logic: 100 to 500 MHz
  - ASIC: 500 MHz to 1 GHz



Synchronous Logic Example: Counter

- Synchronous logic:
  - Updates occur on each clock cycle
- Simple example: A down-counter
- Potential specification: On clock cycle k :
  - If $start[k] = 1$ and $(\mathrm{cnt[k]} = 0)$
    $\Rightarrow cnt[k+1] \leftarrow cnt\_init[k]$
  - Else $cnt[k+1] \leftarrow \max\{0, cnt[k] - 1\}$

k	$start$	cnt_init	cnt
0	0	X	0
1	1	3	0
2	0	0	3
3	1	6	2
4	0	0	1
5	0	0	0
6	1	5	0
7	1	5	5
8	0	0	4
9	0	0	3

Key Components of a Module

- Counter example from previous slide:
  - If $start[k] = 1$ and $(\mathrm{cnt}[\mathrm{k}] = 0)$
    $\Rightarrow cnt[k+1] \leftarrow cnt\_init[k]$
  - Else $cnt[k+1] \leftarrow \max\{0, cnt[k] - 1\}$
- Inputs: $start$ and cnt_init
  - What the module takes as input
- Output: cnt
  - What the module produces
- State: cnt
  - What needs to be stored to compute the subsequent outputs given the inputs

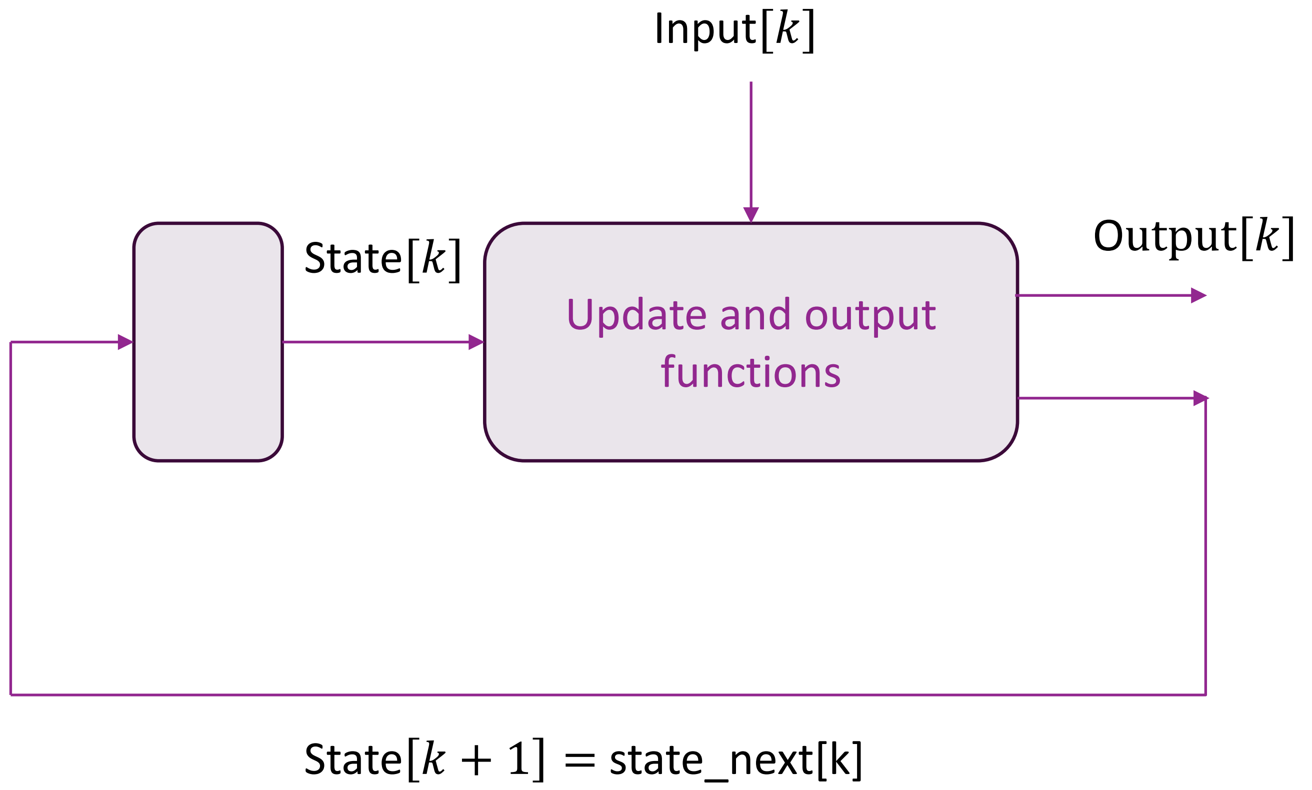
What is a State?

- Definition: A state is a set of variables $state[k]$ such that:
  - Given $state[k]$ and inputs $input[k], input[k+1], \ldots$
  - Outputs $output[k], output[k+1], \ldots$ are fully determined

- In our case:
  - $cnt[k]$ is the state
  - Given $cnt[k]$ and future inputs $cnt\_init[k]$ and $start[k]$
  - We can fully determine future outputs $\mathrm{cnt}[\mathrm{k}], cnt[k+1], cnt[k+2], \ldots$

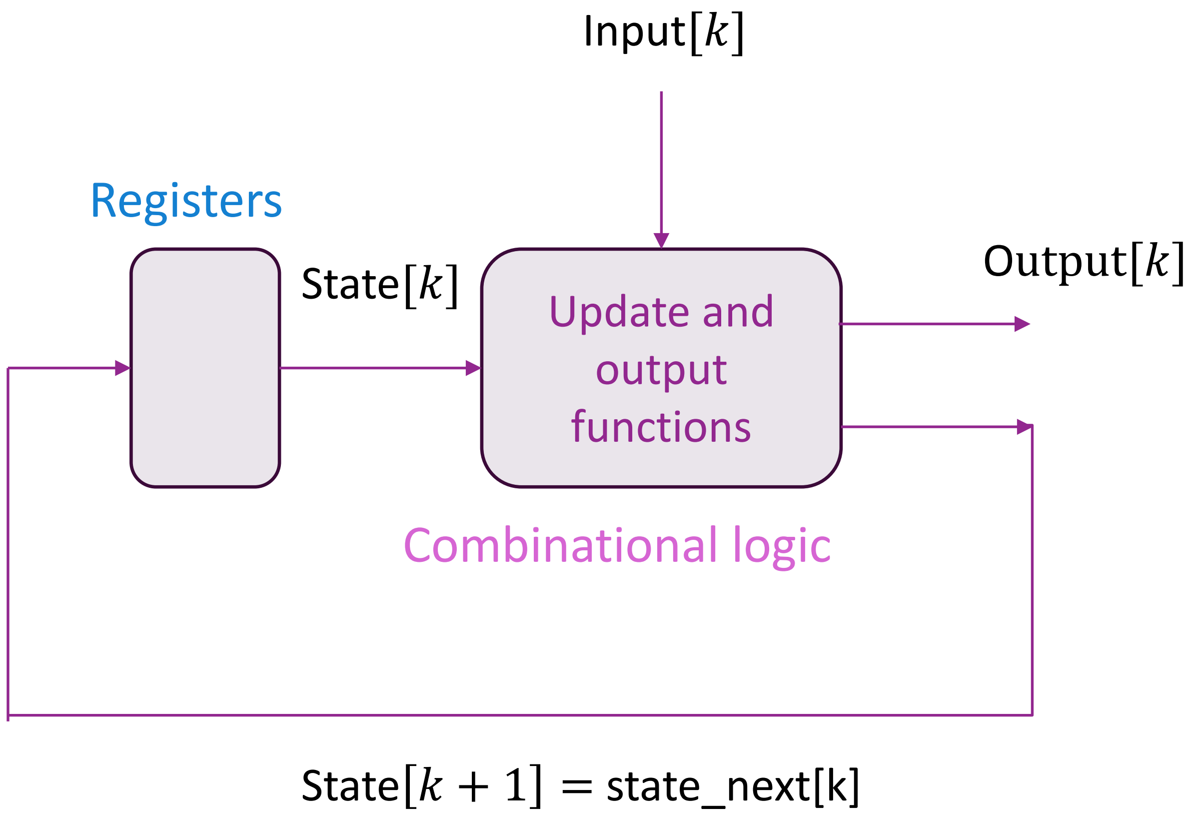
Synchronous Logic as an FSM

- General finite state machine
- Update function:
  - $state[k+1] = F(state[k], input[k])$
  - Describes how the state evolves
- Output function:
  - $output[k] = G(state[k], input[k])$



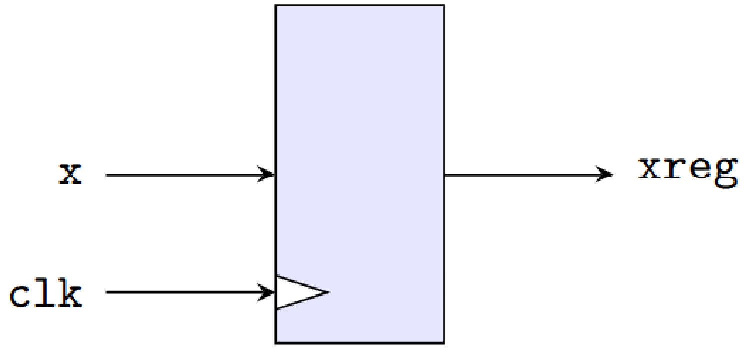
Building an FSM in Hardware

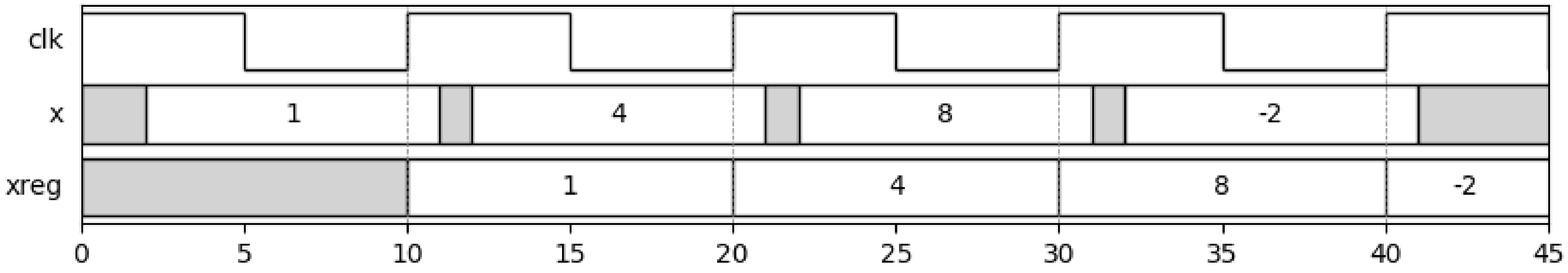
- States are stored in registers
  - Basic storage element
- Update and output functions
  - Implemented in combinational logic
  - No storage
  - Takes input and current state
  - Produces next state and output
  - Must finish within one clock cycle



Register Physical Model

- ❑Basic storage element in synchronous logic
- ❑On each rising edge of clk:
  - Assigns xreg to x
- ❑Value of xreg is maintained even if x changes
- ❑Example below: Input is “sampled” at rising edges



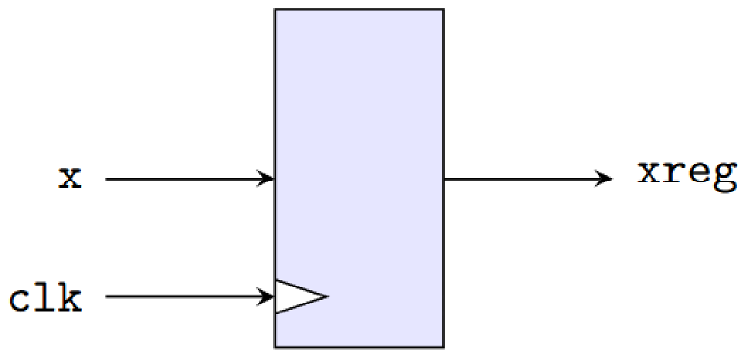


Register: Discrete-Time Model

- Suppose input has value $x[k]$ in clock cycle k
- Then register output is:

$$xreg[k+1] = x[k]$$

  - A simple delay element



- Example:
  - If $x = 4, 8, 9, 1, 2$ in clock cycles 0 to 4
  - Suppose $xreg$ is initialized $xreg = 0$ in clock cycle 0
  - Then $xreg$ = 0, 4, 8, 9, 1, 2 in clock cycles 0 to 5

Writing the Counter in System Verilog

- The counter is a module
- The module definition describes:
  - Inputs and outputs
  - Must have a clock input
  - Typically, also a reset
- Does not need the internal state
  - Unless it happens to be an output

```
module counter (
  input logic clk,
  input logic rst,
  input logic start,
  input logic [3:0] cnt_init,
  output logic [3:0] cnt
);
```

Counter Main Body

Combinational block (always_comb)

- Takes current state and input
- Describes next state and output

always_ff

- Registers values for next state

```
);
  logic [3:0] cnt_next;
  always_comb begin
    cnt_next = cnt;
    if ((start) && (cnt == 0)) begin
      cnt_next = cnt_init;
    end else if (cnt > 0) begin
      cnt_next = cnt - 1;
    end else begin
      cnt_next = 0;
    end
  end
```

```
always_ff @(posedge clk) begin
  if (rst) begin
    cnt <= 0;
  end else begin
    cnt <= cnt_next;
  end
end
```

Putting it Together

- We now have the FSM description
- Three parts
- Module declaration
  - Inputs, outputs
- Always_comb block
  - Describes next state
- Always_ff block
  - Updates state from next_state

```
module counter (
  input logic clk,
  input logic rst,
  input logic start,
  input logic [3:0] cnt_init,
  output logic [3:0] cnt
);
  logic [3:0] cnt_next;
  always_comb begin
    cnt_next = cnt;
    if ((start) && (cnt == 0)) begin
      cnt_next = cnt_init;
    end else if (cnt > 0) begin
      cnt_next = cnt - 1;
    end else begin
      cnt_next = 0;
    end
  end
  always_ff @(posedge clk) begin
    if (rst) begin
      cnt <= 0;
    end else begin
      cnt <= cnt_next;
    end
  end
endmodule
```

Blocking vs. Non-Blocking

❑ Blocking assignment

- Use = in SV
- Performed sequentially
- Assigns temporary (i.e., non-state variables)
- Used in always_comb or always_ff block

```
);
logic [3:0] cnt_next;
always_comb begin
  cnt_next = cnt;
  if ((start) && (cnt == 0)) begin
    cnt_next = cnt_init;
  end else if (cnt > 0) begin
    cnt_next = cnt - 1;
  end else begin
    cnt_next = 0;
  end
end
```

❑ Non-blocking assignment

- Use <= in SV
- Assigns registers / states
- Only in always_ff block
- Always update in parallel

```
always_ff @(posedge clk) begin
  if (rst) begin
    cnt <= 0;
  end else begin
    cnt <= cnt_next;
  end
end
```

Example Problem

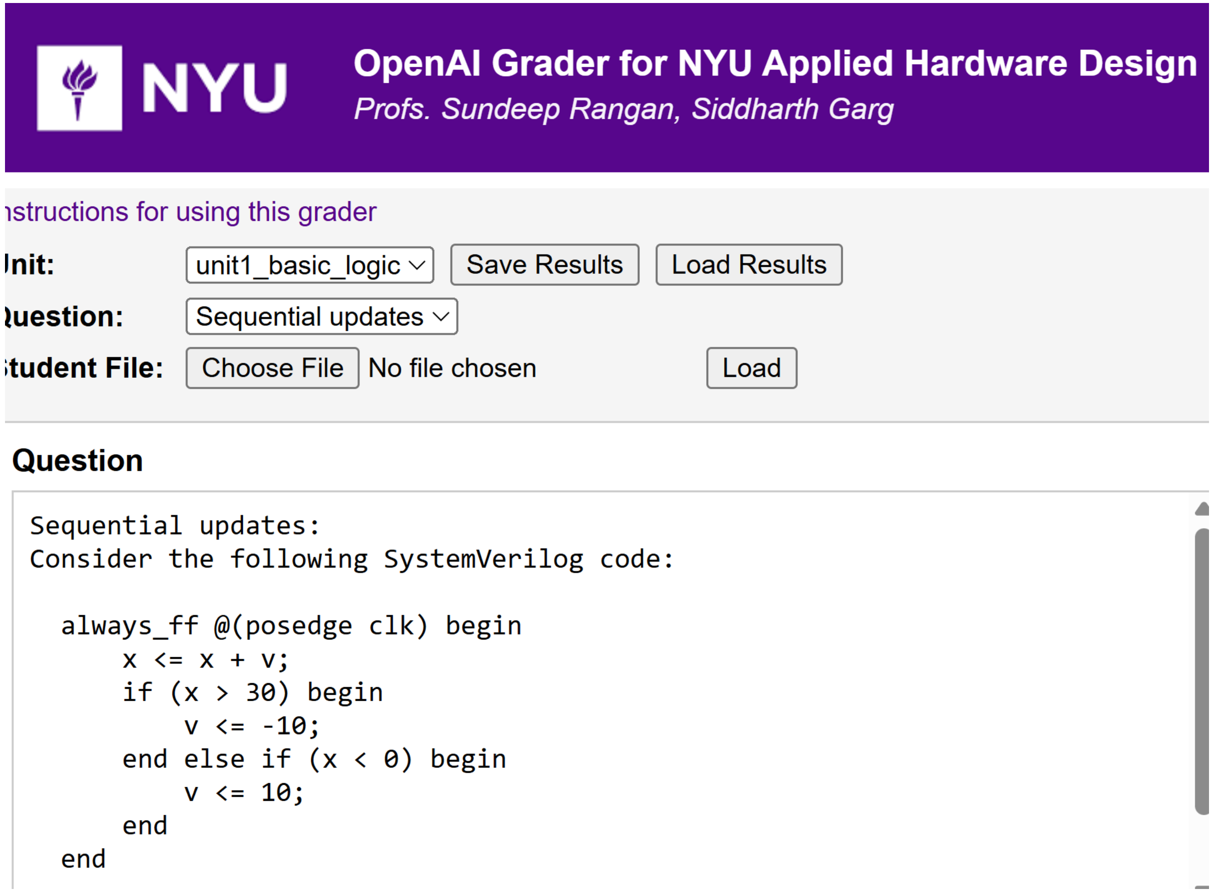
Sequential updates: Consider the following SystemVerilog code:

```
always_ff @(posedge clk) begin
    x <= x + v;
    if (x > 30) begin
        v <= -10;
    end else if (x < 0) begin
        v <= 10;
    end
end
```

Starting from $(x, v) = (15, 10)$, what are the values of (x, v) for the next 5 clock cycles?

Try Answer on LLM Auto-Grader

- Use AI to get instant feedback
- Enter your answer
- Hit Grade



Solution: FSM Model

k	$x[k]$	$v[k]$
0	15	10
1	25	10
2	35	10
3	45	-10
4	35	-10
5	25	-10

- Write the FSM model
- Remember: Updates in always_ff are in parallel
  - Test $x > 30$ is based on current cycle value, not next cycle

```
always_ff @(posedge clk) begin
    x <= x + v;
    if (x > 30) begin
        v <= -10;
    end else if (x < 0) begin
        v <= 10;
    end
end
```

Equivalent FSM model

$x[k+1] = x[k] + v[k]$

if $x[k] > 30$

$\quad v[k+1] = -10$

else if $(x[k] < 0)$

$\quad v[k+1] = 10$

$v[k+1] = v[k]$

In-Class Exercise

5. *Bouncing ball*: We simulate a ball moving in one-dimensional space between two walls at positions 0 and 100. The ball has a position x and a velocity v. At each time step, the ball first moves according to its velocity:

$$x \leftarrow x + v.$$

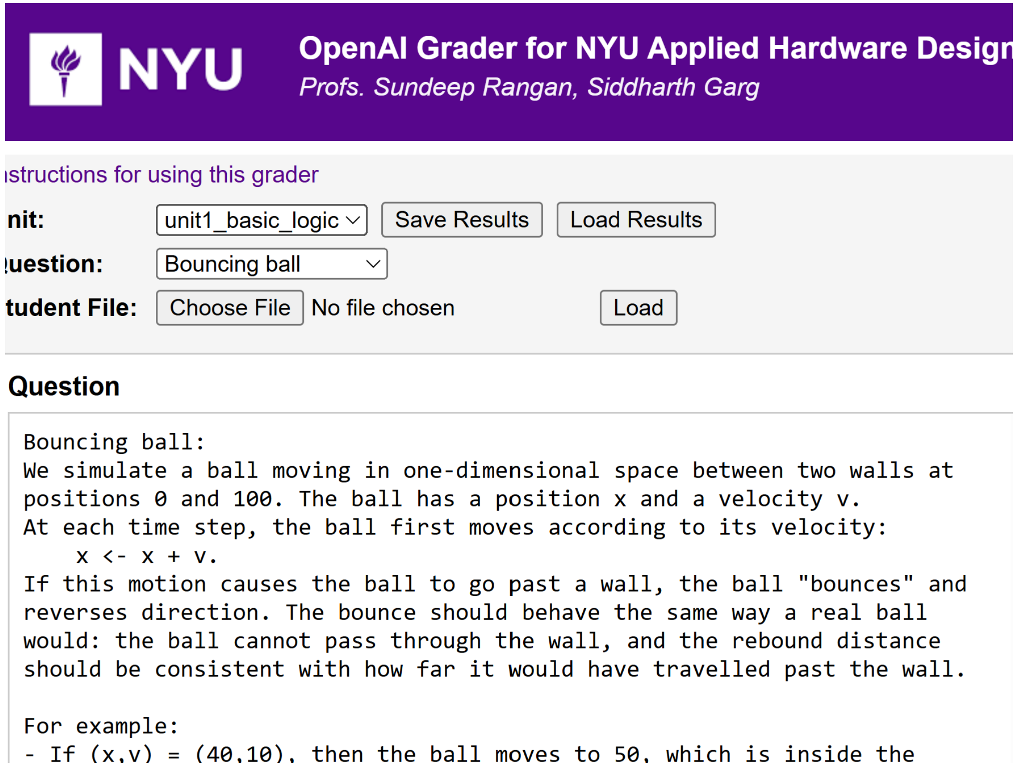
If this motion causes the ball to go past a wall, the ball "bounces" and reverses direction. The bounce should behave the same way a real ball would: the ball cannot pass through the wall, and the rebound distance should be consistent with how far it would have travelled past the wall.

For example:

- If $(x, v) = (40, 10)$, then the ball moves to 50, which is inside the interval, so the next state is $(50, 10)$.
- If $(x, v) = (96, 10)$, then the ball would move to 106, which is past the right wall at 100. After bouncing, the ball ends up at position 94 with velocity -10.
- If $(x, v) = (3, -10)$, then the ball would move to -7, which is past the left wall at 0. After bouncing, the ball ends up at position 7 with velocity 10.

Write the SystemVerilog code for the updates for x and v. You do not need to include the module declaration, just the `always_ff` and `always_comb` blocks.

In-Class Exercise



- ❑ Try answering on the auto-grader

Outline

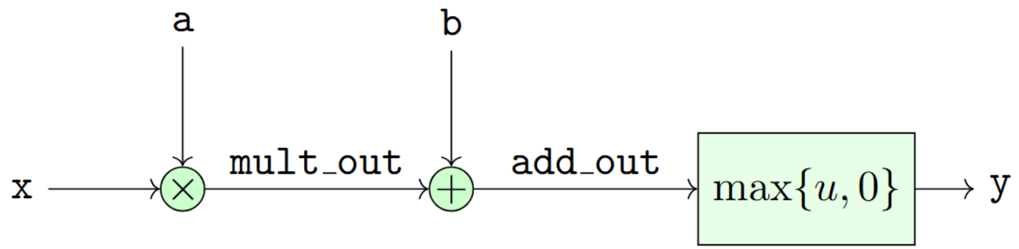
- Combinational Logic
- Registers, Clocks and Sequential Logic and Finite State Machines
- Splitting Long Computations
- Simulating and Synthesizing Modules in Vivado

Synchronous Version of ReLU + Linear

- Recall early combinational example

$$y = \max\{ax + b, 0\}$$

- A potential synchronous version could work as follows
- On each clock cycle:
  - Register input x to a state $xreg$
- Output $y[k]$ combinationally from $xreg[k]$
- Why?
  - Output computation has a full clock cycle
  - Output is in a known state by the end of each clock cycle



Synchronous Module Declaration

- Making module synchronous is easy
- First add two inputs to ports
  - Clock signal
  - Reset

```
module relu_lin (
  input logic clk,
  input logic rst,
  input logic signed [31:0] x,
  input logic signed [31:0] a,
  input logic signed [31:0] b,
  output logic signed [31:0] y
);
```

Clock

Reset

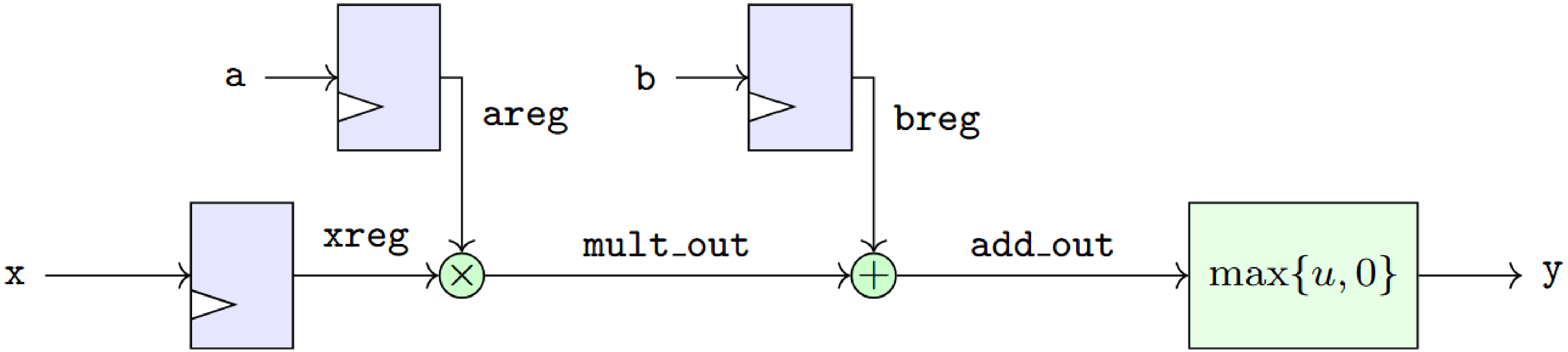
System Verilog Main Body

- Synchronous logic has two parts
- `always_ff`:
  - Assigns register outputs
  - All operations are in parallel (non-blocking)
  - Our example: Saves inputs to registers
- `always_comb`:
  - Assigns signals via combinational logic
  - Operations are sequential (blocking)
  - Our example: Computes output from registered inputs

```
logic signed [31:0] x_reg, a_reg, b_reg;

always_comb begin
  logic signed [31:0] mult_out, add_out;
  mult_out = x_reg * a_reg;
  add_out = mult_out + b_reg;
  y = (add_out > 0) ? add_out : 0;
end


always_ff @(posedge clk) begin
  if (rst) begin
    x_reg <= 0;
    a_reg <= 0;
    b_reg <= 0;
  end else begin
    x_reg <= x;
    a_reg <= a;
    b_reg <= b;
  end
end
```

Synchronous Block Diagram

- Inputs are registered
- Output is combinational from input



FSM Description

- States:
  - $xreg[k], areg[k], breg[k]$
- Combinational statements:
  - $mult\_out[k] = x\_reg[k] * a\_reg[k]$
  - $add\_out[k] = x\_reg[k] * a\_reg[k] + b\_reg[k]$
  - $y[k] = \max\{0,\ x\_reg[k] * a\_reg[k] + b\_reg[k]\}$
- State updates:
  - $x\_reg[k+1] = x[k], a\_reg[k+1] = a[k], \ldots$
- Putting it together:

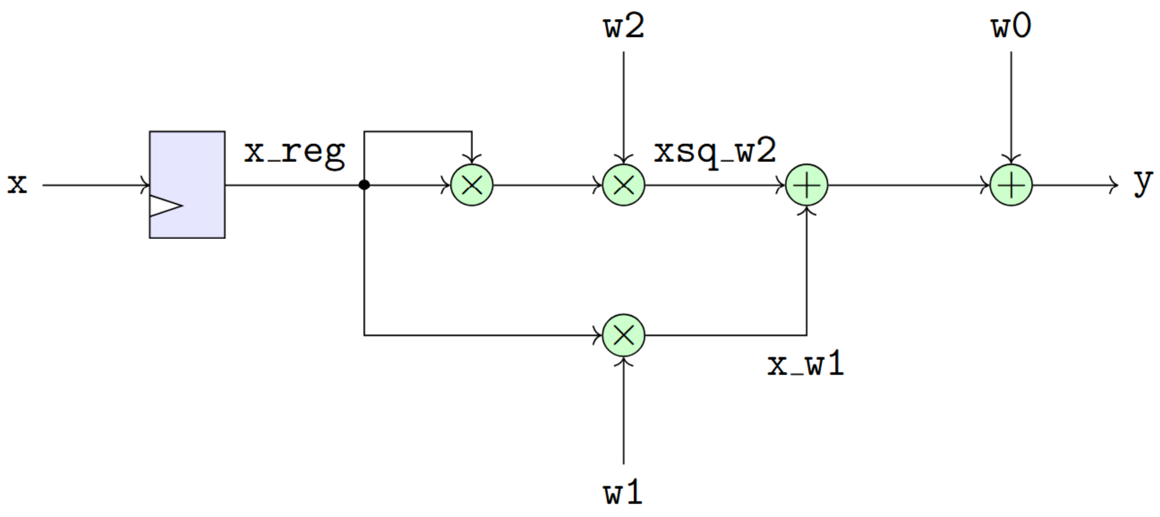
$$y[k+1] = \max\{0, a[k]x[k] + b[k]\}$$

  - Desired output with one cycle delay

```
logic signed [31:0] x_reg, a_reg, b_reg;

always_comb begin
  logic signed [31:0] mult_out, add_out;
  mult_out = x_reg * a_reg;
  add_out = mult_out + b_reg;
  y = (add_out > 0) ? add_out : 0;
end


always_ff @(posedge clk) begin
  if (rst) begin
    x_reg <= 0;
    a_reg <= 0;
    b_reg <= 0;
  end else begin
    x_reg <= x;
    a_reg <= a;
    b_reg <= b;
  end
end
```

A More Complex Example

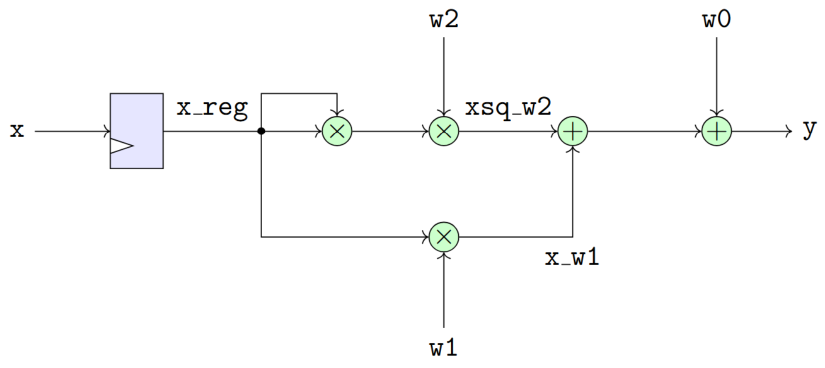
- Consider a quadratic function: $y = w_2 x^2 + w_1 x + w_0$
- For simplicity, assume w_0, w_1, w_2 are fixed parameters
- Single cycle implementation to the right




```
module quad_func #(
  parameter int w2 = 0,
  parameter int w1 = 0,
  parameter int w0 = 0
) (
  input logic clk,
  input logic int x,
  output logic int y
);
  int xreg;
  always_ff @(posedge clk) begin
    xreg <= x;
  end
  always_comb begin
    logic int xsq, xsq_w2, x_w1;
    xsq = xreg * xreg;
    xsq_w2 = w2 * xsq;
    x_w1 = w1 * xreg;
    y = xsq_w2 + x_w1 + w0;
  end
```

Problems with Single Cycle Implementation

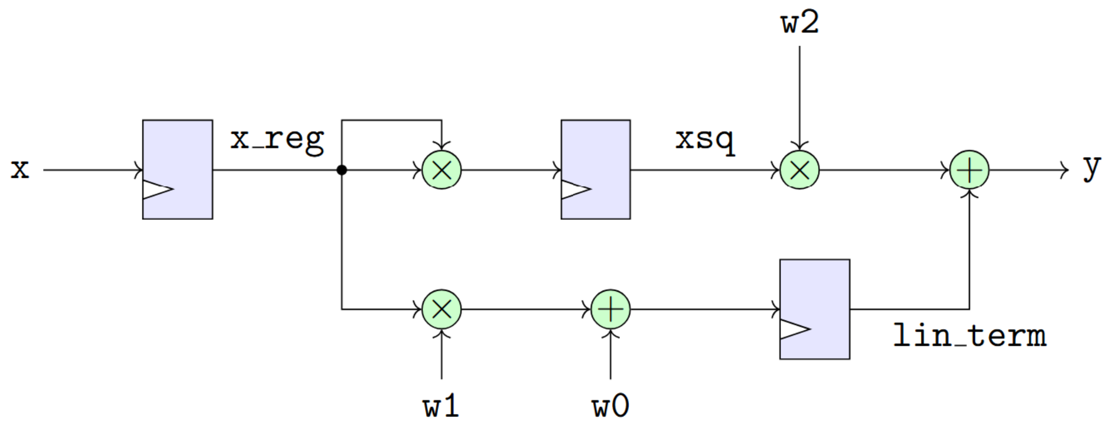
- Problem: Each clock cycle has a "lot" to do
- Longest path from $x\_reg \rightarrow y$ has:
  - Compute xsq : One multiply
  - Compute xsq_w2 : One multiply
  - Add $w0$: One addition
  - We will discuss finding longest paths in detail later
- In general, two multiplications in one clock cycle is hard
- Typically, we limit to one multiplication
  - More on this later




```
always_comb begin
  logic int xsq, xsq_w2, x_w1;
  xsq = xreg * xreg;
  xsq_w2 = w2 * xsq;
  x_w1 = w1 * xreg;
  y = xsq_w2 + x_w1 + w0;
end
```

A Two Cycle Design

- We can reduce the computation per cycle by breaking operation in two cycles
- Register two intermediate terms:
  - `xsq = xreg * xreg`
  - `lin_term = w1*xreg + w0`




```
int xreg;
always_ff @(posedge clk) begin
  xsq <= xreg * xreg;
  lin_term <= w1 * xreg + w0;
  xreg <= x;
end
always_comb begin
  y = w2*xsq + lin_term;
end
```

Analysis

- Registers:
  - $xsq[k], lin\_term[k], xreg[k]$
- Updates:
  - $y[k] = w2 * xsq[k] + lin\_term[k]$
  - $xsq[k+1] = xreg[k]^2$
  - $lin\_term[k+1] = w1\, xreg[k] + w0$
- Substituting in values:
  - $y[k] = w2\ * xreg[k-1]^2 + w1 * xreg[k-1] + w0$
  - $y[k] = w2\ * x[k-2]^2 + w1 * x[k-2] + w0$
- So, output matches desired equation
  - But at a two cycle delay

```
int xreg;
always_ff @(posedge clk) begin
  xsq <= xreg * xreg;
  lin_term <= w1 * xreg + w0;
  xreg <= x;
end
always_comb begin
  y = w2*xsq + lin_term;
end
```

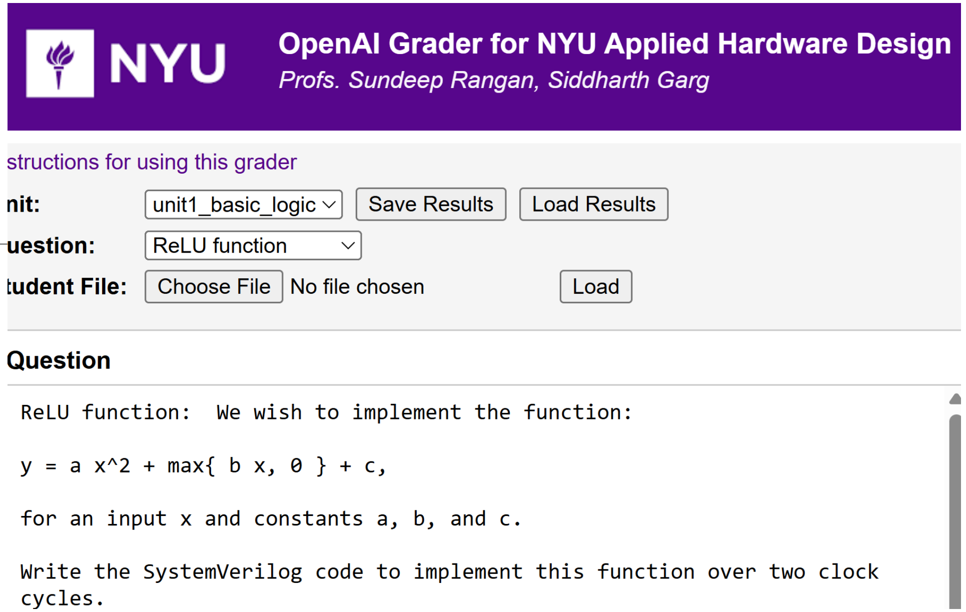
In Class Problem

ReLU function: We wish to implement the function:

$$y = ax^2 + \max\{bx, 0\} + c,$$

for an input x and constants a, b, and c.

Write the SystemVerilog code to implement this function over two clock cycles. Specifically, the input x should be registered in the first clock cycle, and the output y should be produced in the second clock cycle. Make sure that no clock cycle requires two or more multiplications that cannot be parallelized.

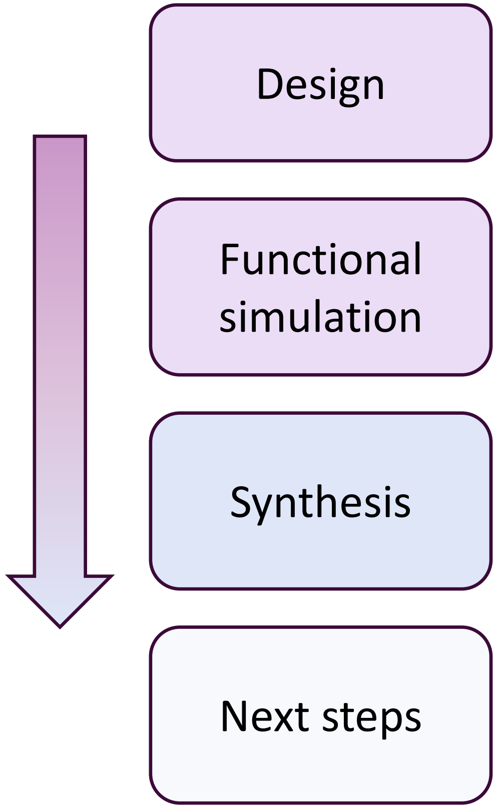


- Hint:
  - Write the computation graph for $x \rightarrow y$
  - Add registers at input and intermediate variables

Outline

- ❑Combinational Logic
- ❑Registers, Clocks and Sequential Logic and Finite State Machines
- ❑Splitting Long Computations
- ❑Simulating and Synthesizing Modules in Vivado

Building the Hardware Roadmap



- **Design**:
  - Write a behavioral description (e.g., in SV)
  - What should the module should do exactly?
- **Functional simulation**
  - Validate module does what it is supposed to do?
  - Logical mapping from inputs to outputs

This unit

- **Synthesis**
  - Map the description into real hardware
  - Gates, FPGA components, …
- **Next steps**
  - Place route, timing analysis, bitstreams, …

Later units…

Linear+ReLU Example

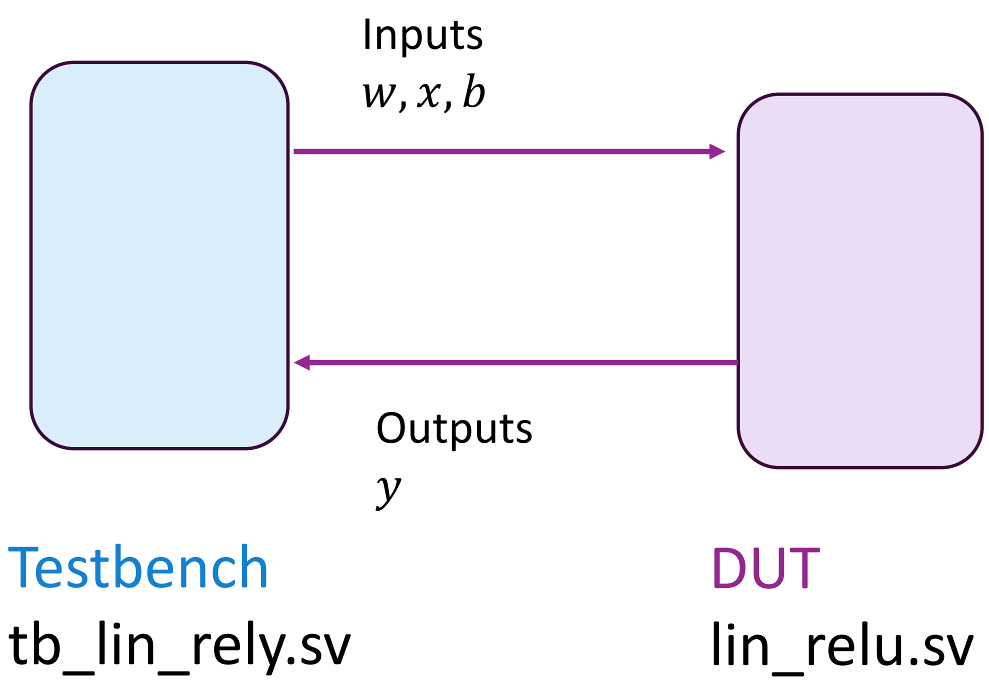
- All files in github repo: hwdesign/demos/fsm
- Define counter in file counter.sv

```
module lin_relu #(
    parameter WIDTH = 16
)(
    input  logic                    clk,
    input  logic                    rst,    // sync
    input  logic signed [WIDTH-1:0]  w_in,
    input  logic signed [WIDTH-1:0]  b_in,
    input  logic signed [WIDTH-1:0]  x_in,
    output logic signed [WIDTH-1:0]  y_out
);
```

```
// Registered inputs
logic signed [WIDTH-1:0] w_reg, b_reg, x_reg;

// Intermediate value
logic signed [WIDTH-1:0] u;

// Register the inputs
always_ff @(posedge clk) begin
    if (rst) begin
        w_reg <= '0;
        b_reg <= '0;
        x_reg <= '0;
    end else begin
        w_reg <= w_in;
        b_reg <= b_in;
        x_reg <= x_in;
    end
end

// Combinational output
always_comb begin
    u = w_reg * x_reg + b_reg;
    y_out = (u > 0) ? u : 0;
end
```

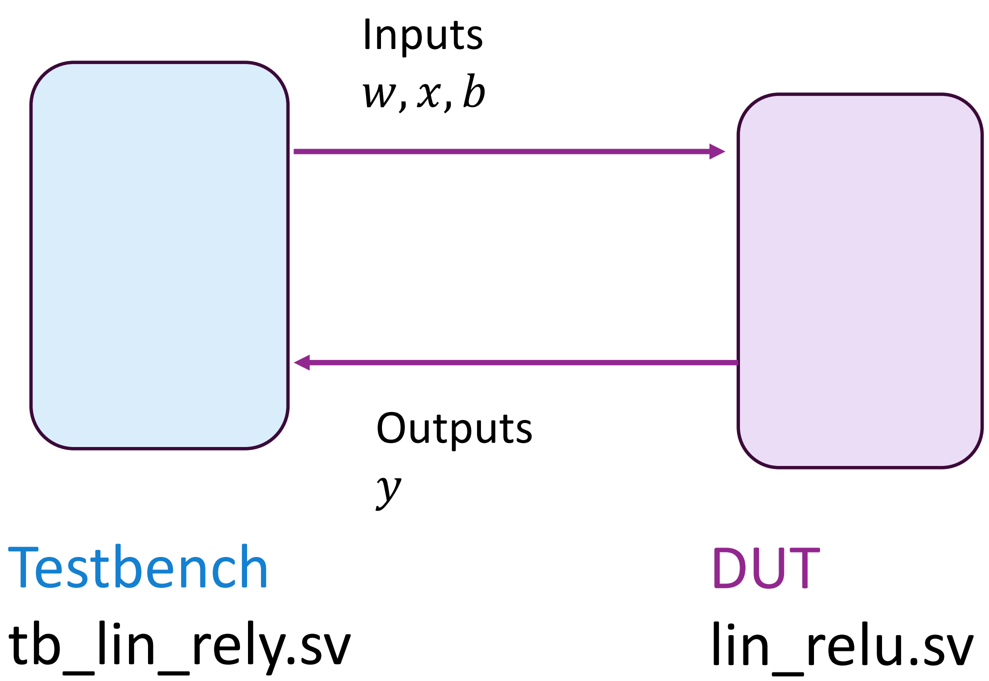
Next Step: Build a Testbench

- **Testbench**
  - A second SV module
  - Connects to device under test (i.e, module we want to test)
  - TB provides DUT inputs
  - TB verifies DUT outputs are correct



Testbenches are not Synthesized

- Testbenches are for simulation only
- They will not be synthesized
- Can include un-synthesizable constructs
  - Print statements
  - File read and write
  - Complex data structures
- Modern trend:
  - Write TB in higher level language
  - E.g., python



Testbench Declaration

- ❑ Testbench: a module to test the module
  - Will not be synthesized
  - Can have un-synthesizable constructs
  - Ex: Read / write from file

```
module tb_lin_relu;

    localparam WIDTH = 16;
    localparam time CLK_PERIOD = 10ns;

    logic clk = 0;
    logic rst = 1;
    logic signed [WIDTH-1:0] x_in;
    logic signed [WIDTH-1:0] w_in;
    logic signed [WIDTH-1:0] b_in;
    logic signed [WIDTH-1:0] y_out;
```

- ❑ Create a clock

- ❑ Instantiate an instance of module to test
  - Called the DUT = device under test
  - Connect DUT to testbench signals

```
// Clock generation
always #(CLK_PERIOD/2) clk = ~clk;

// Instantiate DUT
lin_relu #(
    .WIDTH(WIDTH)
) dut (
    .clk(clk),
    .rst(rst),
    .w_in(w_in),
    .b_in(b_in),
    .x_in(x_in),
    .y_out(y_out)
);
```

Test Vectors

- ❑Create a set of test inputs

```
// Define test vectors
test_vector_t test_vectors[] = '{
    '{x: 10,  w: 3,  b: 15},
    '{x: -4, w: 10,  b: 4},
    '{x: 8, w: 12,  b: -5}
};
```

- ❑Loop through inputs
- ❑Put one new input each clock cycle
- ❑Testbench uses an initial construct
  - Enables sequential set of events

```
for (int i = 0; i < test_vectors.size(); i++) begin

    #(0.1*CLK_PERIOD) // hold time before changing input
    x_in = 'x;  // initial intedeterminate value
    w_in = 'x;
    b_in = 'x;
    #(0.15*CLK_PERIOD);  // Small delay for propagation time
    x_in = test_vectors[i].x;
    w_in = test_vectors[i].w;
    b_in = test_vectors[i].b;

    // Clock cylce
    @(posedge clk);

    // Compute expected value for verification (optional)
    x  = test_vectors[i].x;
    w  = test_vectors[i].w;
    b  = test_vectors[i].b;
    y_exp = (w * x + b > 0) ? (w * x + b) : 0;

    $display("Test %0d: x_in=%0d, y_out=%0d, y_exp=%0d",
             i, x, y_out, y_exp);
end
```

Initial Block

- ❑ Runs once at time 0 of the simulation
  - (or at its scheduled start time if delayed).
- ❑ Statements inside run sequentially
  - like a small program.
- ❑ Not synthesizable for hardware
  - Simulation-only construct.
  - Used for test benches
- ❑ Concurrency:
  - initial blocks and always blocks.

```
initial begin
    // Define test vectors
    test_vector_t test_vectors[] = '{
        '{x: 10,  w: 3,  b: 15},
        '{x: -4, w: 10,  b: 4},
        '{x: 8, w: 12,  b: -5}
    };

    // Local temporaries for expected value computation
    logic signed [WIDTH-1:0] x;
    logic signed [WIDTH-1:0] w;
    logic signed [WIDTH-1:0] b;
    logic signed [WIDTH-1:0] y_exp;

    // Reset for a few cycles
    repeat (1) @(posedge clk);
    rst = 0;


    for (int i = 0; i < test_vectors.size(); i++) begin

        #(0.1*CLK_PERIOD) // hold time before changing inp
        x_in = 'x;  // initial intedeterminate value
        w_in = 'x;
        b_in = 'x;
        #(0.15*CLK_PERIOD);  // Small delay for propagatio
```

Test Results

- ❑Test results outputs
- ❑Match expected values

```
Test 0: x_in=10, y_out=45, y_exp=45
Test 1: x_in=-4, y_out=0, y_exp=0
Test 2: x_in=8, y_out=91, y_exp=91
```