Unit 1
Basic Digital Logic

EL-GY 9463: INTRODUCTION TO HARDWARE DESIGN

PROFS. SUNDEEP RANGAN, SIDDHARTH GARG

Learning Objectives

- ❑Write SystemVerilog descriptions for simple hardware modules
- ❑Draw high-level block diagram representations of the module implementations
- ❑Identify the critical path and its delay from the block diagram
- ❑Implement simple functions in synchronous logic
- ❑Visualize signals in a timing diagram
- ❑Write a testbench for the hardware modules
- ❑Simulate, and synthesize the SystemVerilog using Vivado tools
- ❑Extract value-change dump (VCD) files and synthesis reports and process them in python

Outline

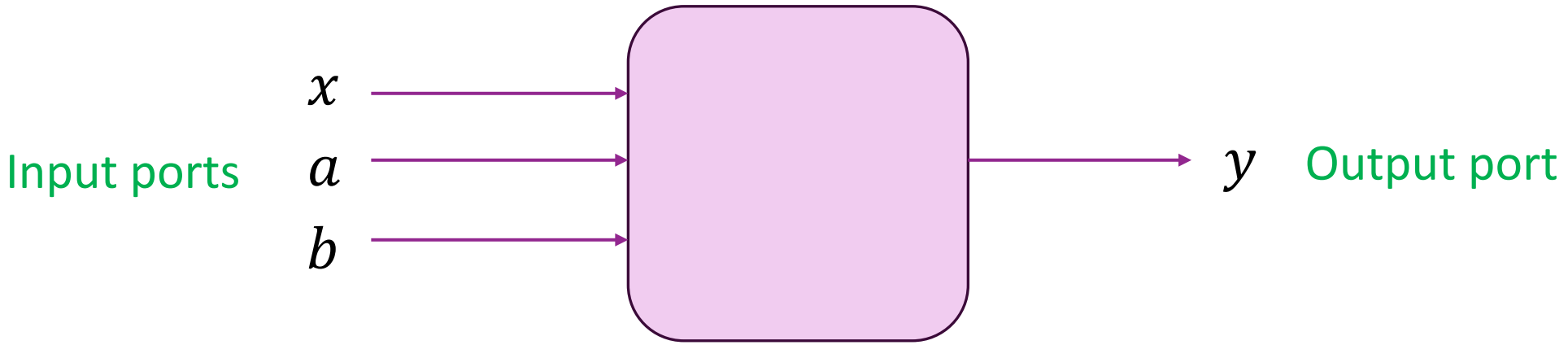
- Combinational Logic
- Timing Diagrams for Combinational Logic
- Registers, Clocks and Sequential Logic
- Simulating and Synthesizing Simple Modules in Vivado

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 int x,
  input logic int a,
  input logic int b,
  output logic int y
);
  always_comb begin
    logic int 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 int x,
  input logic int a,
  input logic int b,
  output logic int y
);
  always_comb begin
    logic int 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 (e.g., int)
- 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

Synthesis

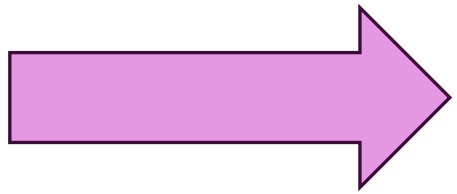
Behavioral Description

What module does?

Ex: Verilog, HLS, SV

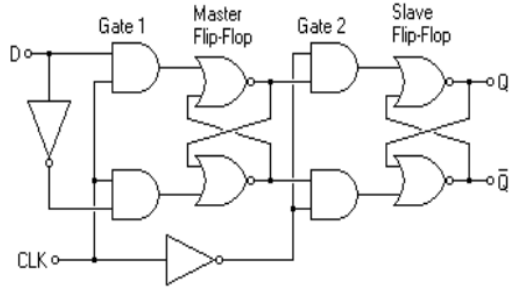
```
module relu_lin (
  input logic int x,
  input logic int a,
  input logic int b,
  output logic int y
);
  always_comb begin
    logic int mult_out, add_out;
    mult_out = x * a;
    add_out = mult_out + b;
    y = (add_out > 0) ? add_out : 0;
  end
endmodule
```

Synthesis



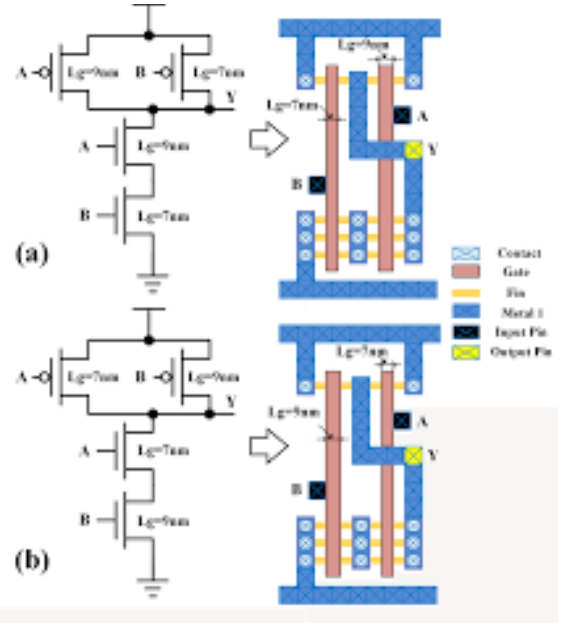
Implementation

How functions maps to hardware



Implementation Depends on the Target

- Components for implementation depend on target
- Gate-level netlist (classic implementation)
  - AND, OR, XOR, NOT gates
  - Multiplexers, adders, subtractors, flip-flops, latches
- Technology-mapped standard cells for ASICs
  - NAND2_X1, DFFR_X2, AOI21_X4, INV_X1
  - Actual cells in foundry's library
- FPGA primitives
  - DSP units, LUTS, BRAM, …

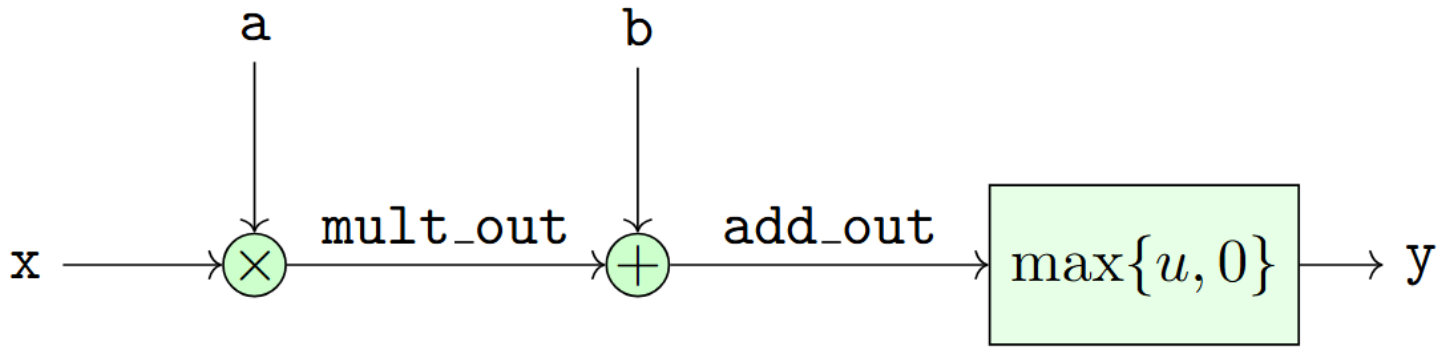


FinFET NAND gate

Cui, Tiansong, et al. "An exploration of applying gate-length-biasing techniques to deeply-scaled FinFETs operating in multiple voltage regimes." *IEEE Transactions on Emerging Topics in Computing* 6.2 (2016): 172-183.

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

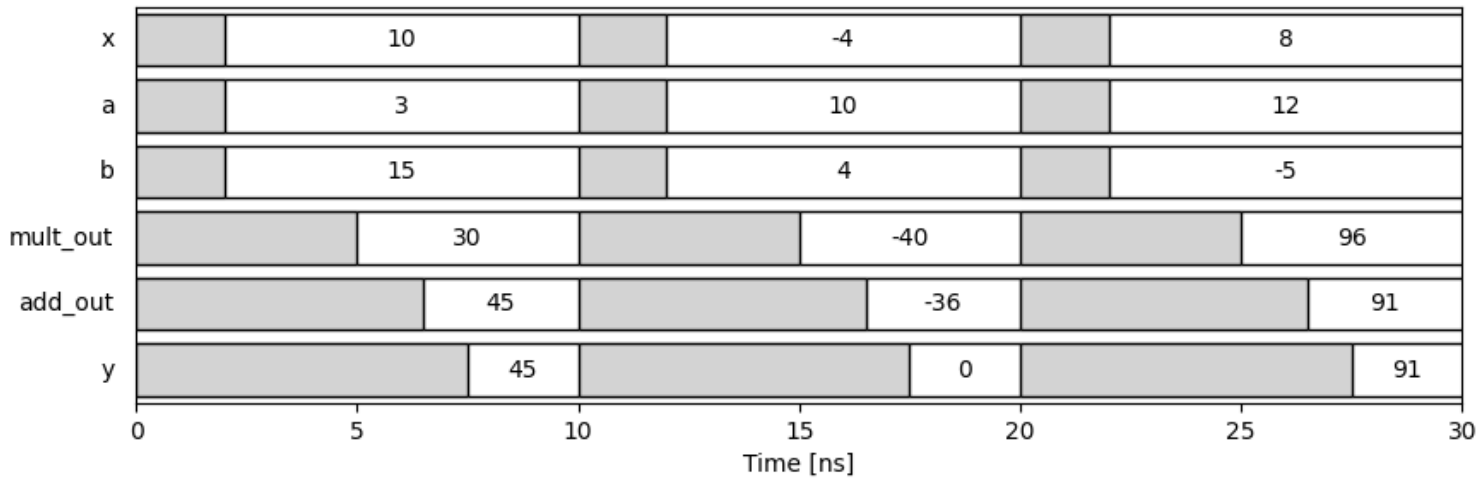


Outline

- ❑ Combinational Logic
- ❑ Timing Diagrams for Combinational Logic
- ❑ Registers, Clocks and Sequential Logic
- ❑ Simulating and Synthesizing Simple Modules in Vivado

Timing Diagrams

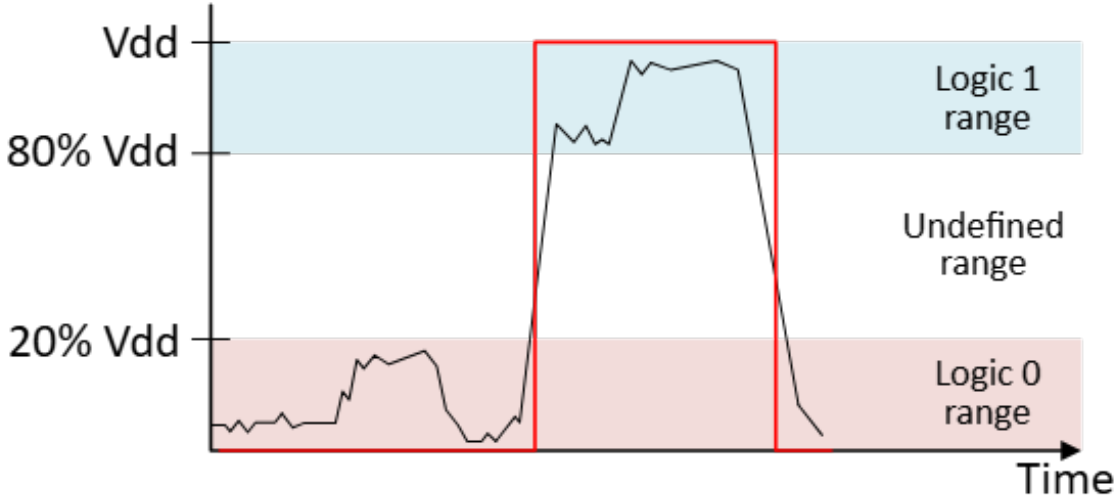
- Graphical representation of the signals in the circuit over time
- Key to analyzing circuits
- Demonstrates:
  - Sequence of operations
  - Latency in each step
  - Critical paths



Binary Signals

- Signal: Any time-varying quantity

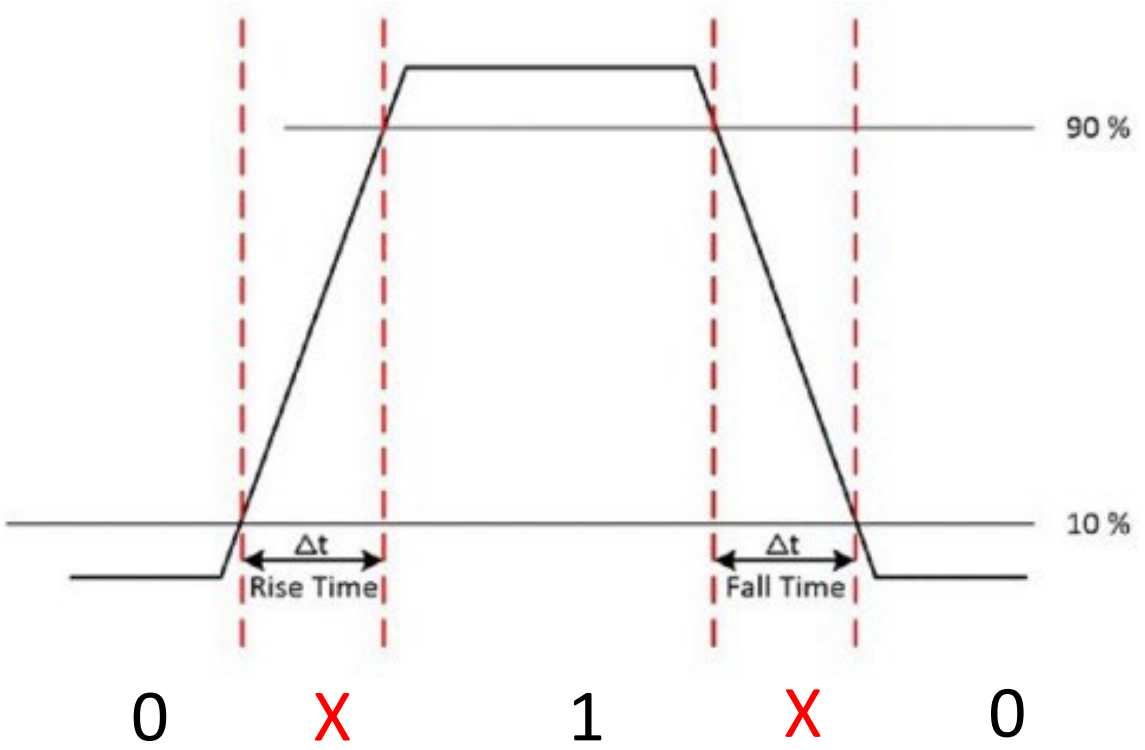
- Digital Binary Signals:
  - Two logical values: 0 or 1
  - Physically: Typically, a continuous voltage
  - Logical value by thresholding



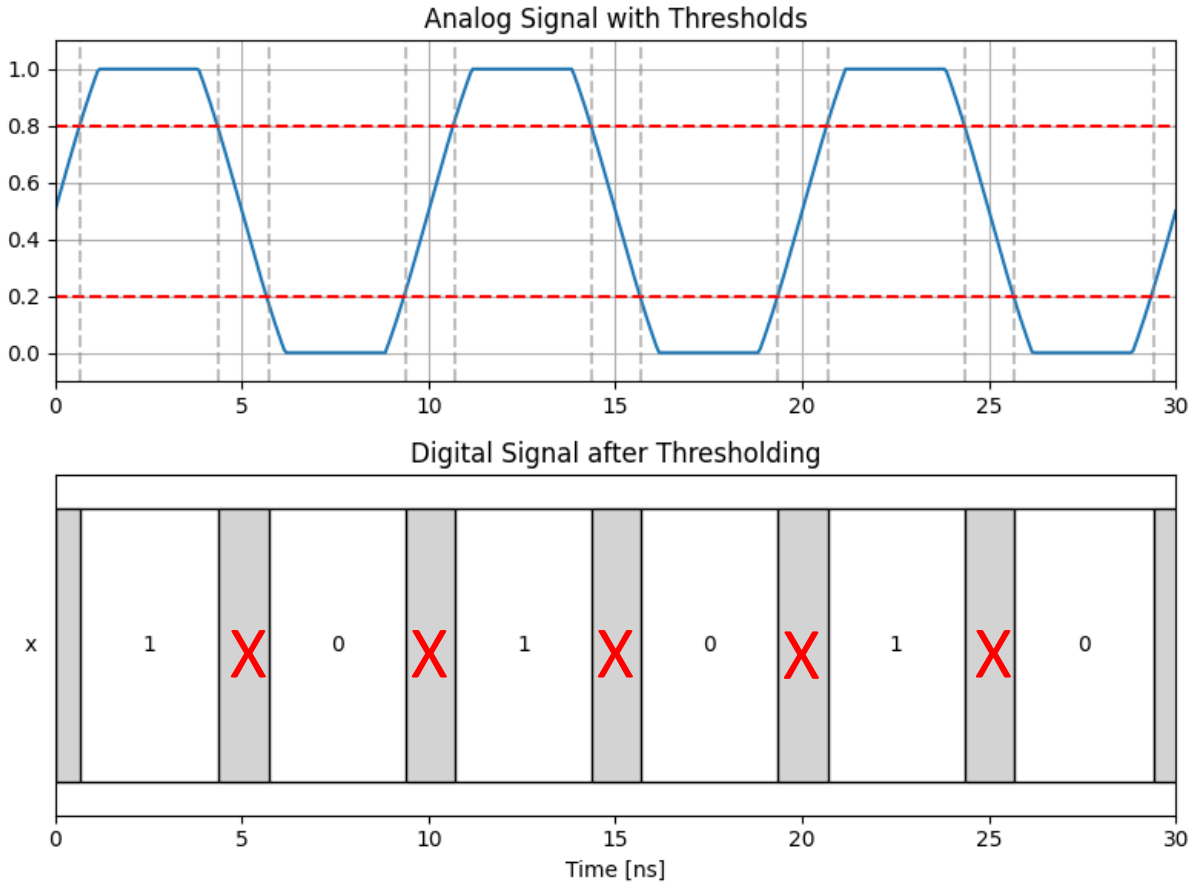
- Other signals (e.g. integer, floats) built from binary signals

Undefined Logic Value: 'X'

- Signals take time to transition
  - They are outputs of physical circuits
  - Gates, computational units, …
- Creates an undefined period
  - Also called indeterminate, unknown, …
- Usually denoted as X
- So binary logic signals have three values:
  - 0, 1 and X
  - Will discuss a fourth value Z later



Undefined Regions Illustrated



- Analog physical signal
  - Typically, a voltage
  - Continuous valued
  - Has finite transition periods

- Digital logic signal
  - Three values 0, 1 and X
  - X value shown in gray

Value-Change Points

- In digital logic, signals are often described by their change points

Definition: A signal $x(t)$ has value-change points $\{(t_i, x_i), i = 0, \dots, n-1\}$ if:

$$x(t) = \begin{cases} X & t < t_0 \\ x_i & t_i \le t < t_{i+1}, \quad i = 0, \dots, n-2 \\ x_{n-1} & t \ge t_{n-1} \end{cases}$$

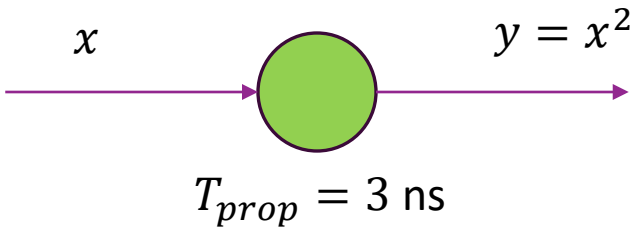
  - Value change points also called valued change events, transitions, …

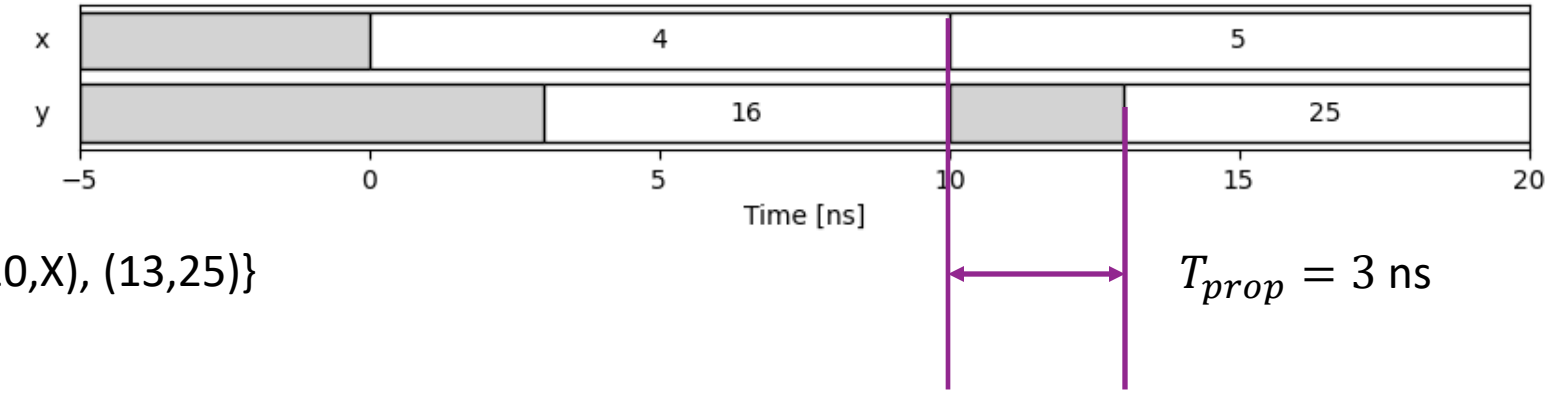
Timing Diagram

- Timing diagram: A plot of signals over time showing with value-change points
- Example with two signals:
- VC points for x : {(3,5), (20,17), (40,8)}
- VC points for y : {(12,3), (25,12), (35,0)}
- In this plot, times are in ns

Propagation Delays

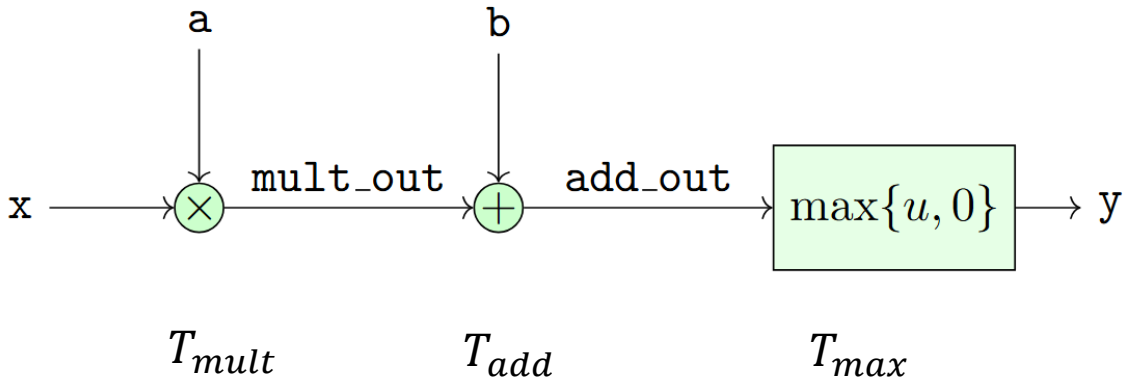
- Each circuit element incurs a propagation delay
  - Time for the output to settle after the input changes
- On transition of input
  - Output is initially unknown 'X'
  - We cannot assume the value is stable during transition
  - After propagation delay, output settles at value
- Example:
  - $y = x^2$
  - x has VC points {(0,4), (10,5)}
  - Prop delay is 3 ns
  - y has VC points {(0,X), (3,16), (10,X), (13,25)}



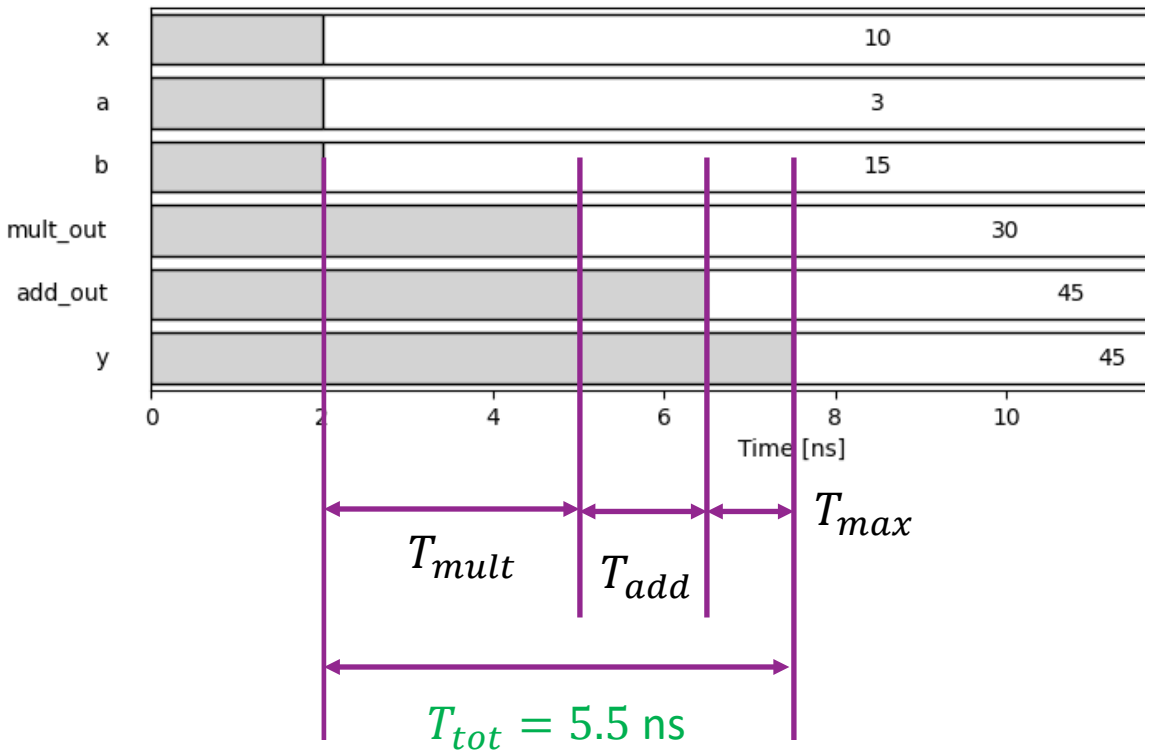


Propagation Delays of a Series

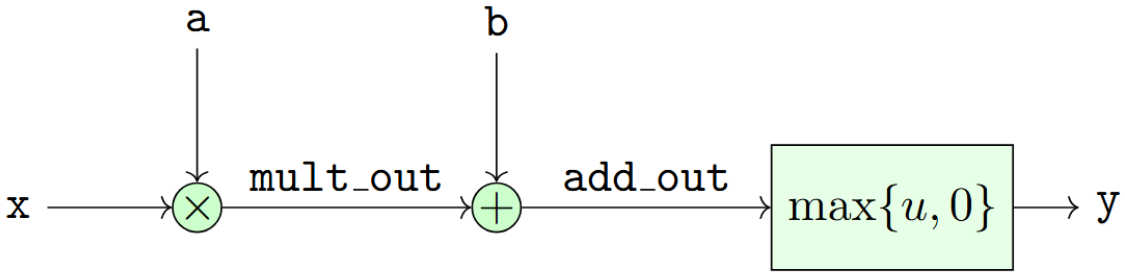
- Our example: Three propagation delays
  - Multiplier
  - Adder
  - Max operator
- Propagation delays will add
- Called the total propagation delay
  - Or total path delay
- Key property: When input x changes
  - All downstream values initially go to X



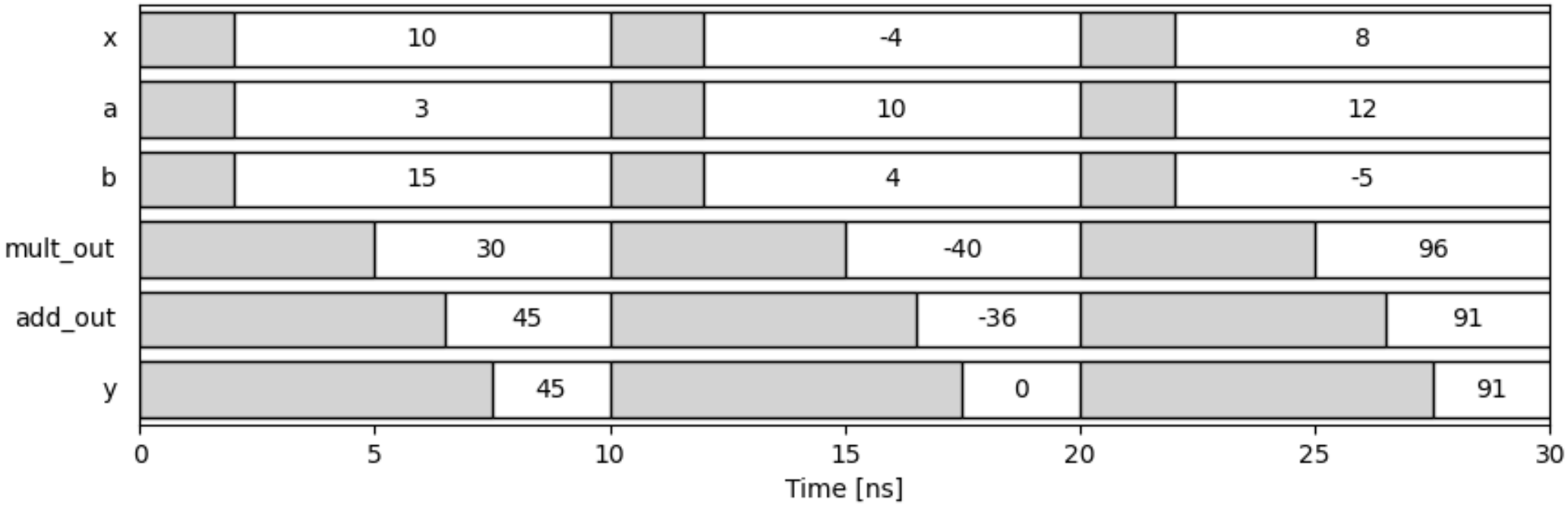
Example Total Propagation Delay



Hardware block	Example Value
Multiplier	T_{mult} =3 ns
Adder	T_{add} =1.5 ns
Max operator	T_{max} =1 ns
Total	$T_{tot} = 5.5$ ns



Example Timing Over Multiple Inputs



Typical Propagation Delays: Low-End FPGA

Hardware block	FPGA resource	Typical range
32-bit adder	LUT + dedicated carry chain	1.5–3.0 ns
16-bit multiplier	DSP48E1 block	2.5–5.0 ns
16-bit comparator	LUT logic + carry chain	1.0–2.0 ns

- Rough delays for a low-end FPGA (e.g., Pynq Z2)
- We will explain the FPGA resources more in later units
- Routing delays not included
  - May be large for more complex designs

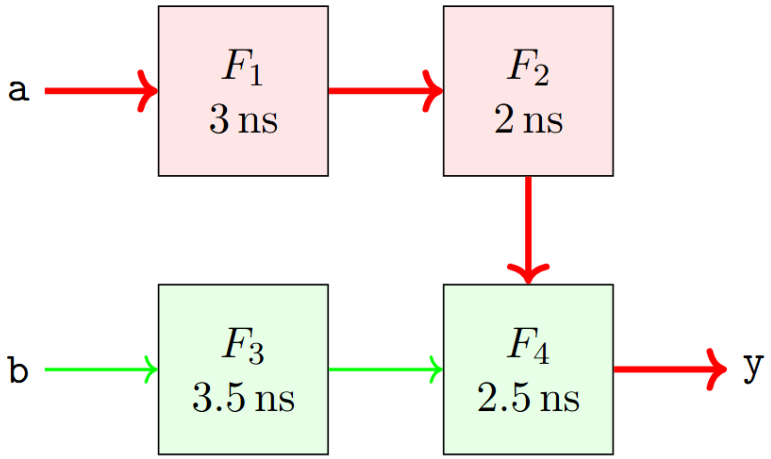
Typical Propagation Delays: ASIC

Hardware block	Typical structure	Typical range
32-bit adder	Kogge-Stone / Brent-Kung / hybrid	150-450 ps
16-bit multiplier	Wallace/Dadda tree + final adde	300-700 ps
16-bit comparator	XOR/AND/OR chain with early-out	50-300 ps

- Rough delays for TSMC 16nm
- Custom ASICs are much faster than FPGAs
  - Easily 5 to 10x faster

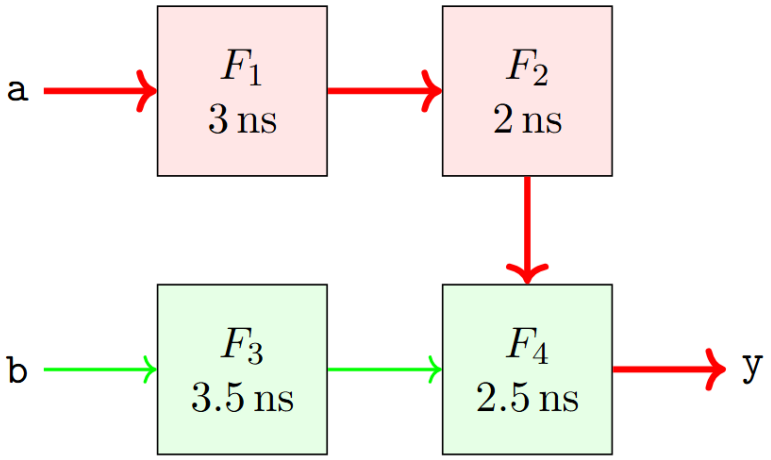
Critical Path

- Signals may need to propagate over multiple paths
- Each path accumulates different path delay
  - Total propagation time on path
- Critical path:
  - The path with the highest total propagation delay
- Critical path will determine total delay of module



Critical Path Example

- Example: For some functions $F_1, \ldots, F_4$
  - $x_1 = F_2(F_1(a))$
  - $x_2 = F_3(b)$
  - $y = F_4(x_1, x_2)$
- Path 1 ($F_1 \rightarrow F_2 \rightarrow F_4$):
  - Delay = 3+2+2.5 = 7.5 ns
  - Longest = Critical path
- Path 2 ($F_3 + F_4$):
  - Delay = 3.5 + 2.5 = 6 ns
  - Shorter = Non-critical

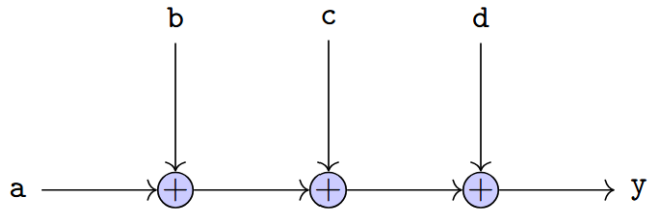


Delay Depends on Implementation

- Example: Consider $y = a + b + c + d$
- Suppose each add takes D seconds
- Synthesis tools try to find low delay implementations

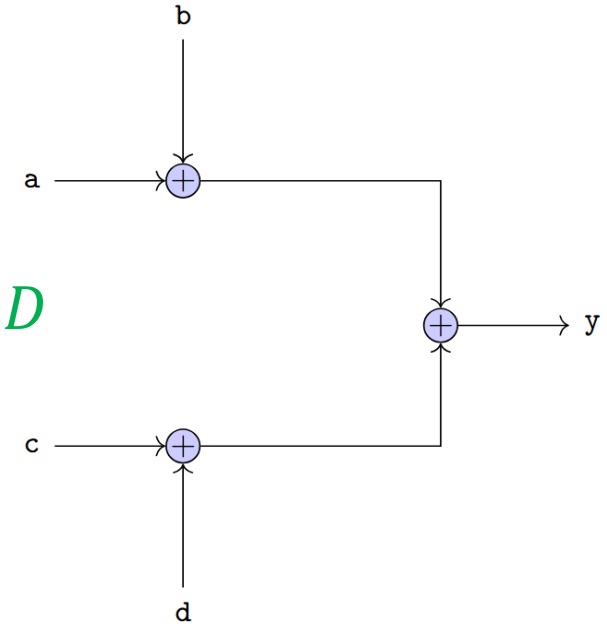
Serial implementation

Prop delay = $3D$



Parallel

Prop delay =$2D$

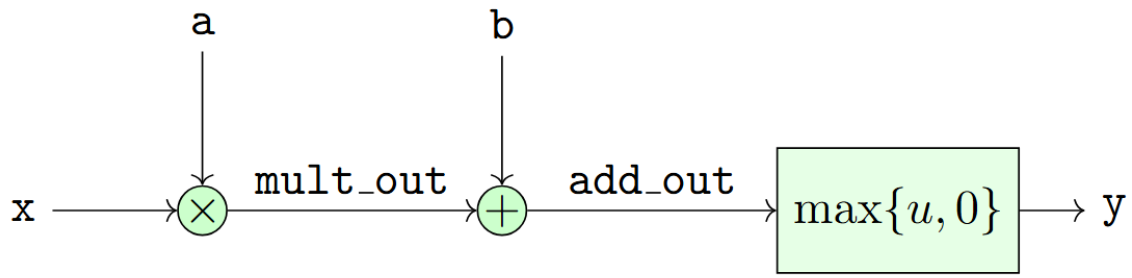


Outline

- ❑ Combinational Logic
- ❑ Timing Diagrams for Combinational Logic
- ❑ Registers, Clocks and Sequential Logic
- ❑ 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



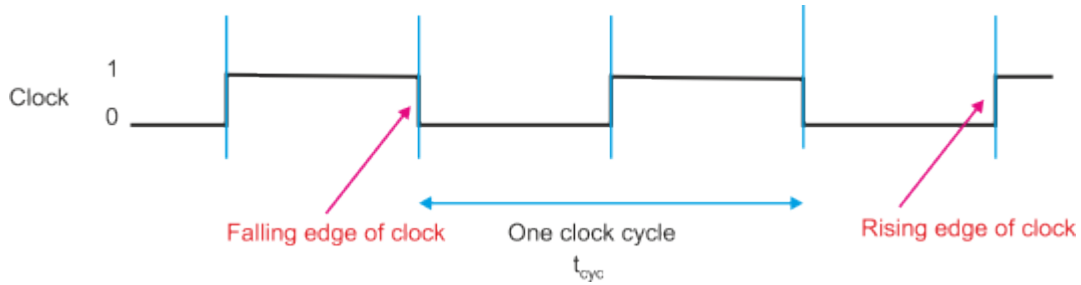
Synchronous (Sequential) Logic

- 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

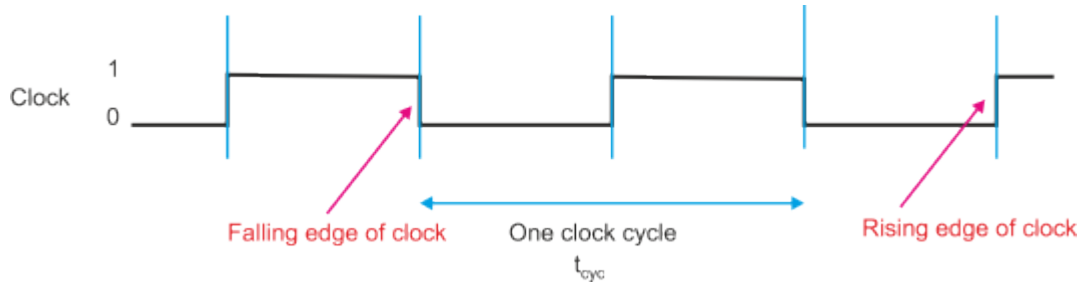
Clock Signal

- 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



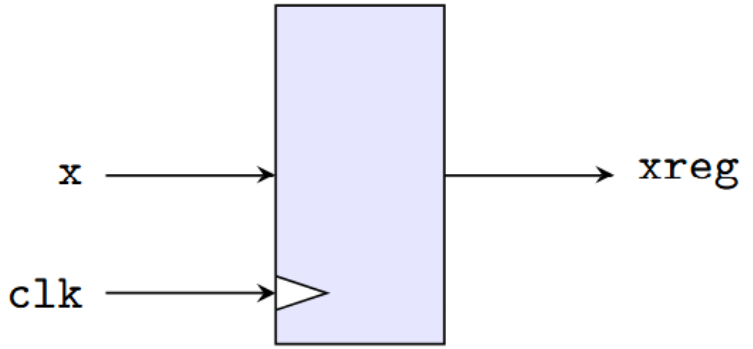
Synchronous Logic

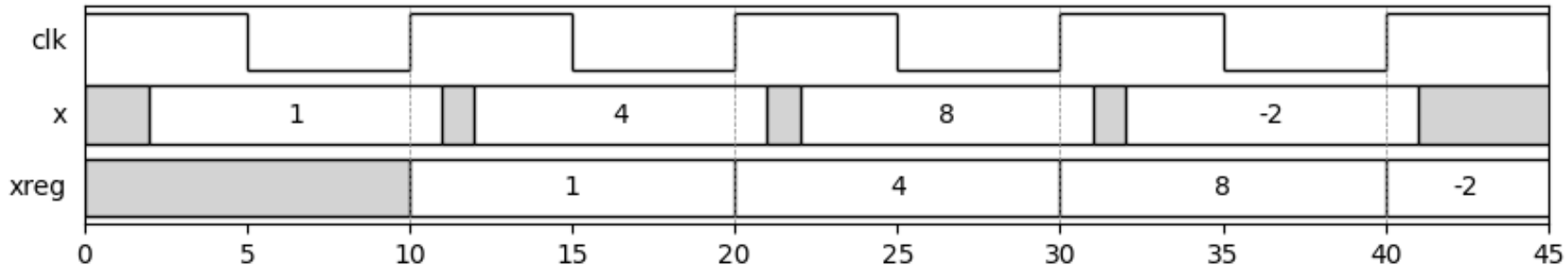
- Key idea: Save system state on each rising edge
- Save state in a storage element
  - This unit we will use registers (next slide)
- Brings system to a known state in each cycle
- Computation can be performed in known steps



Register

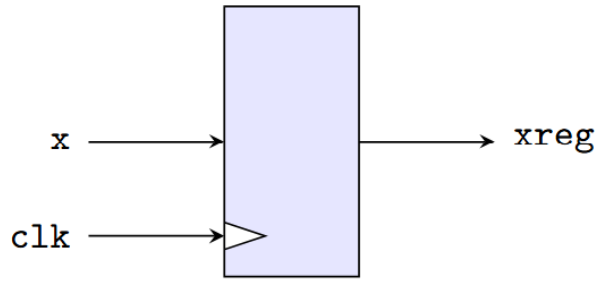
- Basic storage element in synchronous logic
- On each rising edge of clk:
  - Assigns xreg to x
- Value of xreg is maintained even if x changes
- Example below: Input is “sampled” at rising edges
  - Unknown periods not propagated

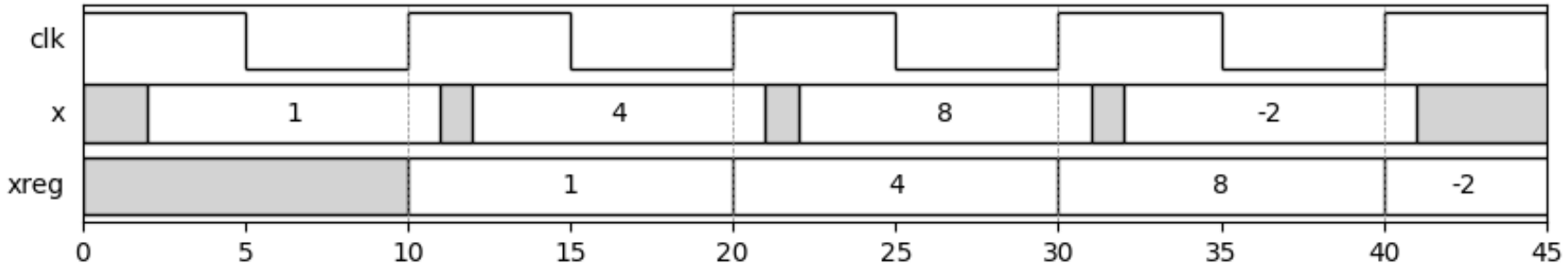




Register

- Samples input at each rising edge
- Input may have X values
  - Not propagated assuming input is vald on positive edges





Making the Module Synchronous

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

```
module relu_lin (
  input logic clk,
  input logic rst_n,
  input logic int x,
  input logic int a,
  input logic int b,
  output logic int y
);
```

Clock

Reset

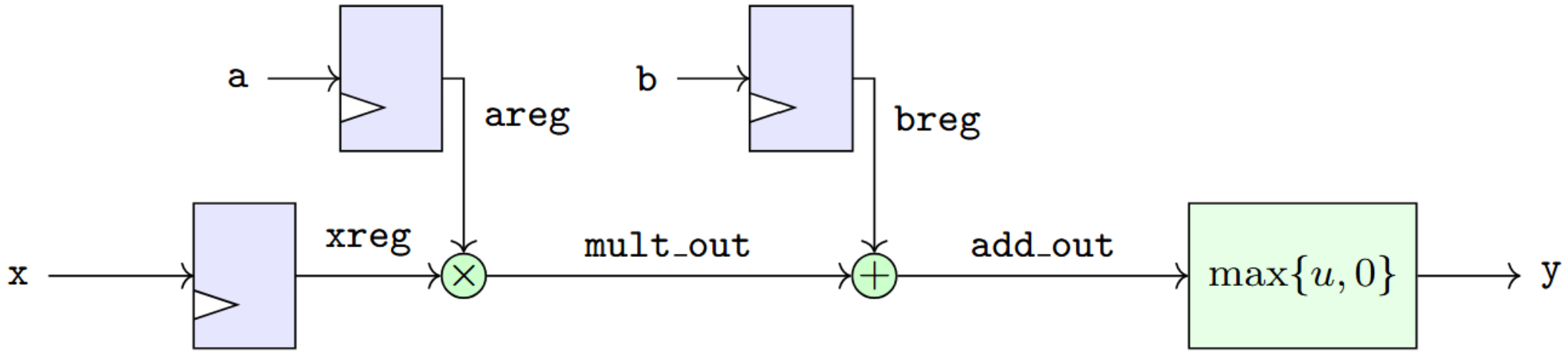
System Verilog Main Body

- 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

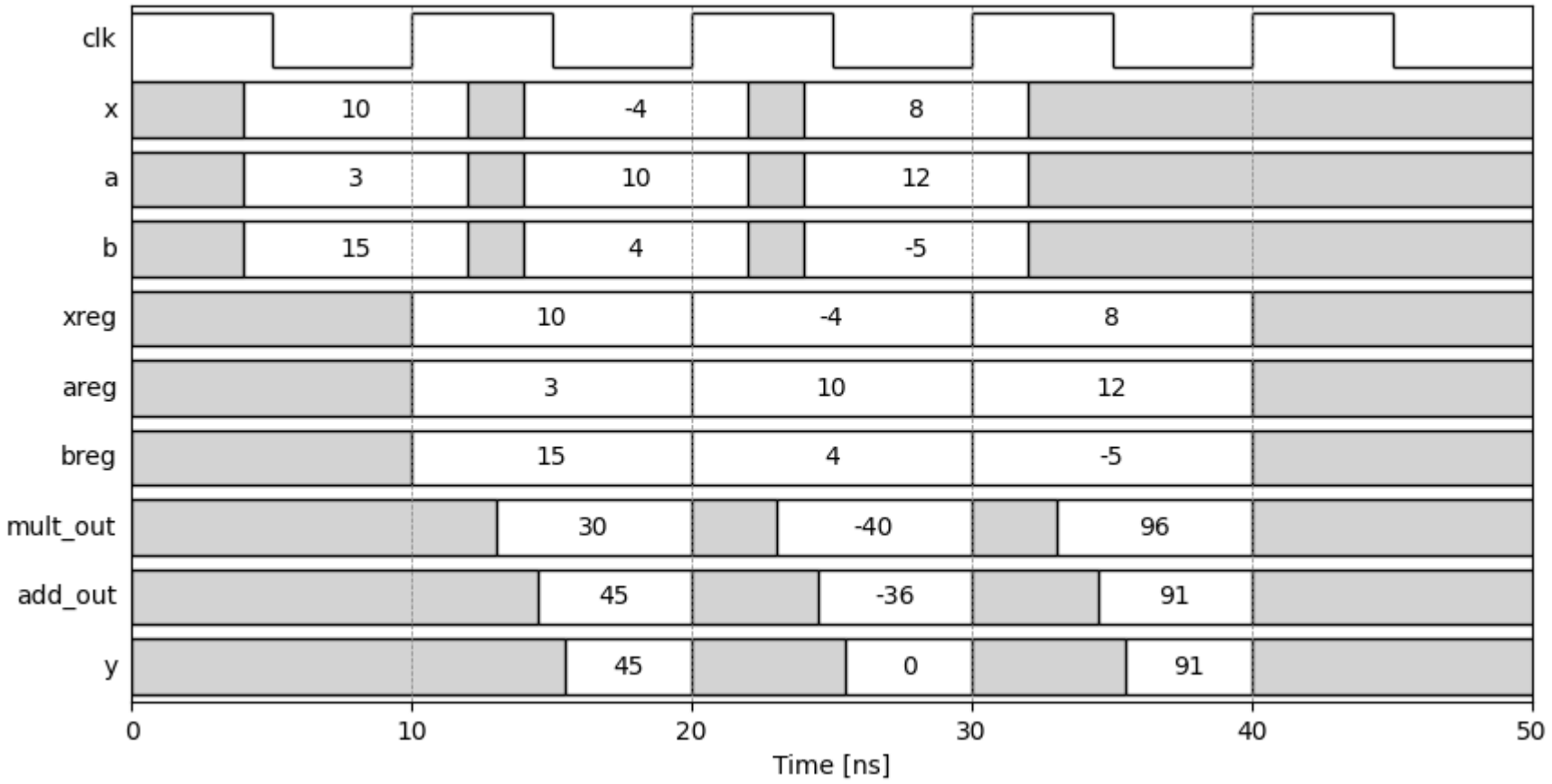
```
  int x_reg, a_reg, b_reg;

  always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
      x_reg <= 0;
      a_reg <= 0;
      b_reg <= 0;
    end else begin
      x_reg <= x;
      a_reg <= a;
      b_reg <= b;
    end
  end
  always_comb begin
    logic int mult_out, add_out;
    mult_out = x_reg * a_reg;
    add_out = mult_out + b_reg;
    y = (add_out > 0) ? add_out : 0;
  end
endmodule
```

Synchronous Block Diagram

- Inputs are registered
- Output is combinational from input



Synchronous Timing Diagram

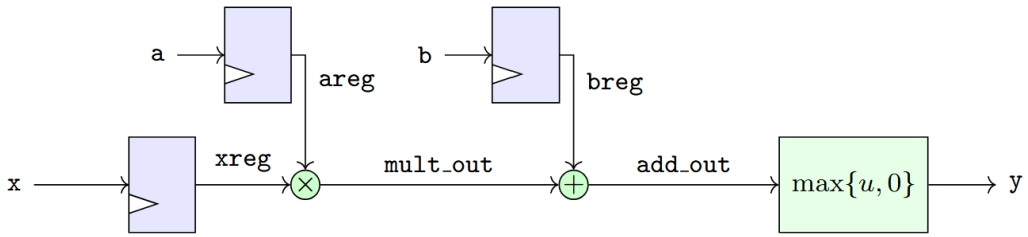


- Clk period = 10 ns
- Clk frequency = 100 MHz

Maximum Clock Frequency

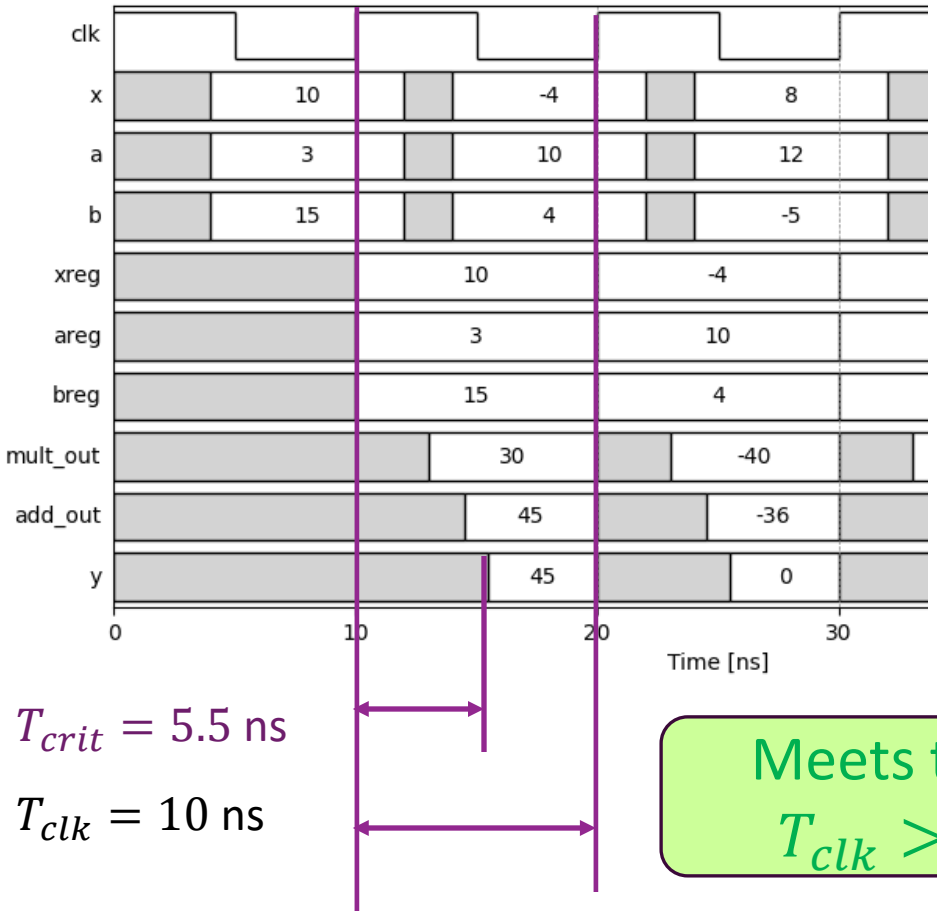
- Consider combinational path
  - xreg, areg, breg → y
- This path must complete within one clock cycle
- Otherwise, next input to next stage not correctly sampled

To meet timing: $T_{clk} > T_{crit}$



  - T_{crit} = delay of critical path, T_{clk} = clock period
- $T_{clk} > T_{crit}$: System meets timing
- $T_{clk} > T_{crit}$: System fails timing / timing violation

ReLU+Linear Example

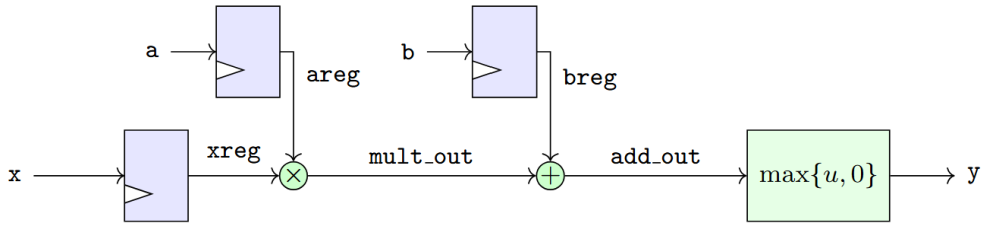


$T_{crit} = 5.5$ ns

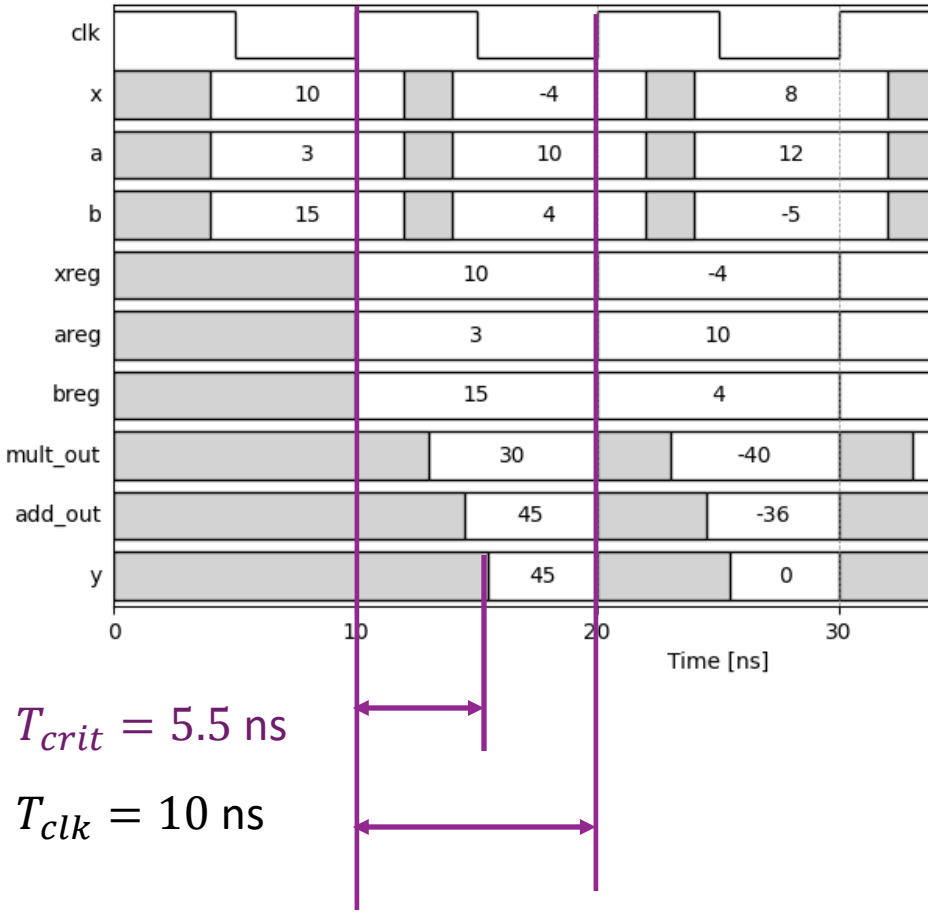
$T_{clk} = 10$ ns

Meets timing:
$T_{clk} > T_{crit}$

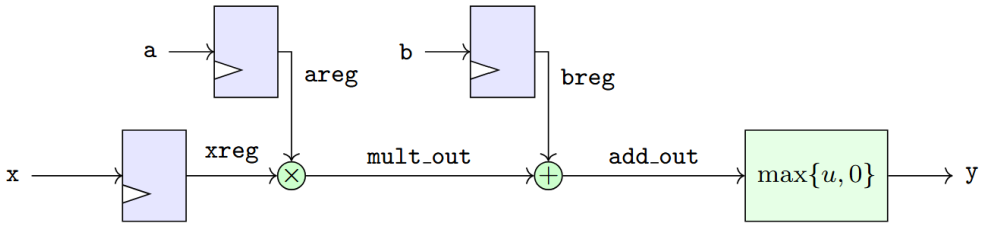
Hardware block	Example Value
Multiplier	T_{mult} =3 ns
Adder	T_{add} =1.5 ns
Max operator	T_{max} =1 ns
Total =critical path	$T_{crit} = 5.5$ ns



Max Clock Frequency Example

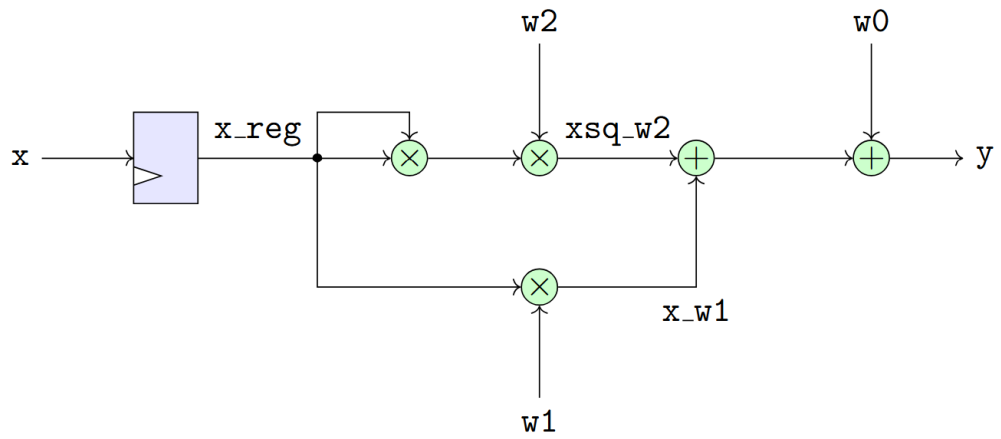


- Max clock frequency in our example:
  - Min $T_{clk} = T_{crit} = 5.5$ ns
  - Max clock frequency = $\frac{1}{5.5} \cong 180$ MHz



A More Complex Example

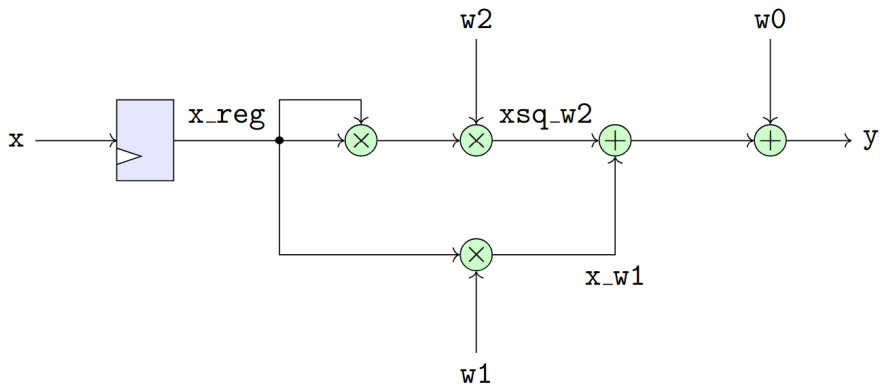
- Consider a quadratic function: $y = w_2x^2 + w_1x + w_0$
- For simplicity, assume w_0, w_1, w_2 are fixed parameters
- Single cycle implementation to the right




```
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
```

Finding the Critical Path

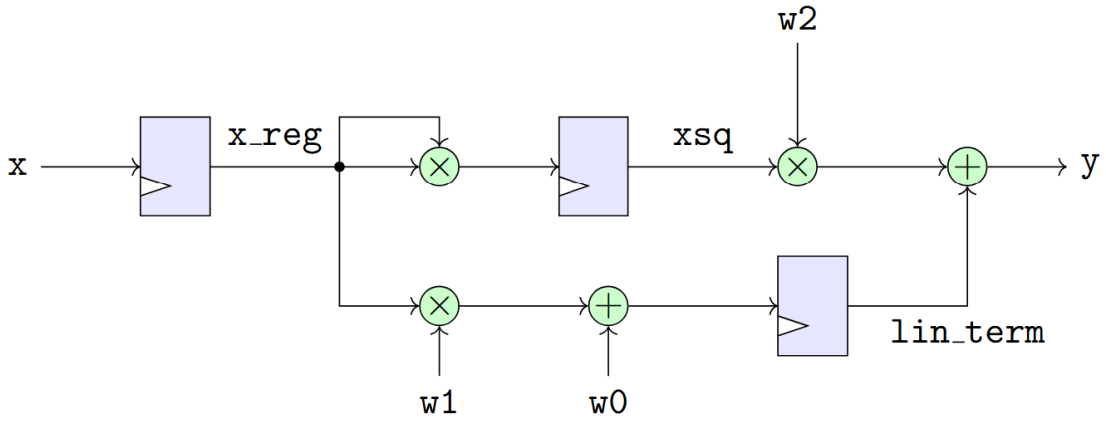
Hardware block	Example Value
Register	T_{reg}= 1 ns
Multiplier	T_{mult} = 4 ns
Adder	T_{add} =1.5 ns
Path 1	1+4+4+1.5+1.5 = 12
Path 2	1+4+1.5+1.5=8



- Consider implementation with times in table
- Two paths:
- Path 1: x, x_reg, xsq_w2, y
  - 1 register, 2 mults, 2 adders = 12 ns
- Path 2: x, x_reg, x_w1, y
  - 1 register, 1 mult, 2 adders = 8 ns
- Path 1 is critical
- Max clock frequency $= \frac{1}{12\ ns} = 83$ MHz
- Would fail timing at 100 MHz

A Two Cycle Design

- We can improve throughput by breaking computation over 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
```

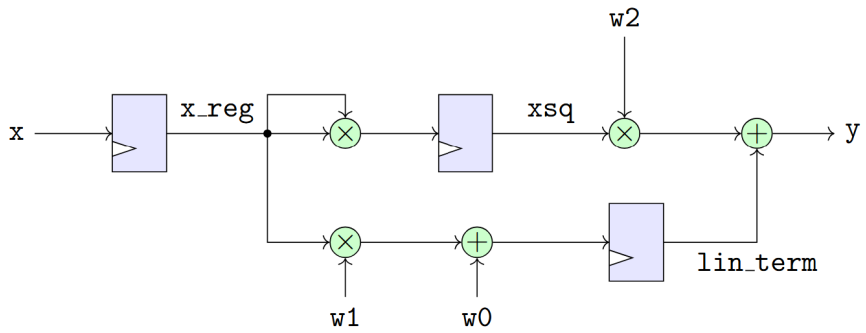
Three Stages

- Designs is processed in three stages = 2 cycles
- Stage 0
  - Register input x to `xreg`
- Stage 1: Compute and register:
  - `xsq = xreg * xreg`
  - `lin_term = w1*xreg + w0`
- Stage 2: Compute output
  - `y = w2*xsq + lin_term`

```
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
```

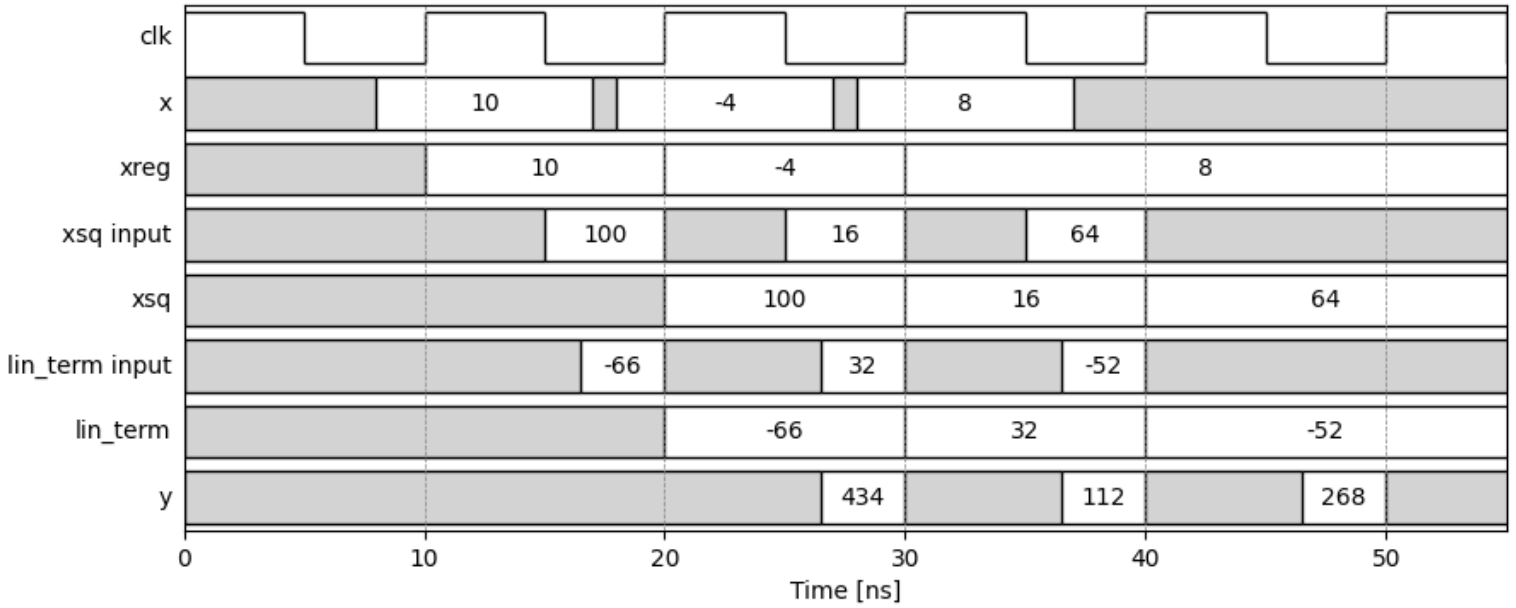
Critical Path with Three Stages

Hardware block	Example Value
Register	T_{reg}= 1 ns
Multiplier	T_{mult} = 4 ns
Adder	T_{add} =1.5 ns
Path 1	1+4=5
Path 2	1+4+1.5=6.5
Path 3	1+4+1.5=6.5



- Measure paths from register to register
- Path 1: `x, x_reg, xsq` input
  - 1 register, 1 mult = 5 ns
- Path 2: `x, x_reg, w1_x, lin_x` input
  - 1 register, 1 mult, 1 add = 6.5 ns
- Path 3: `xsq_reg, lin_x, w2_xsq, y`
  - 1 register, 1 mult, 1 add = 6.5 ns
- Critical path delay = 6.5 ns
- Max freq = $\frac{1}{6.5} \approx 153$ MHz

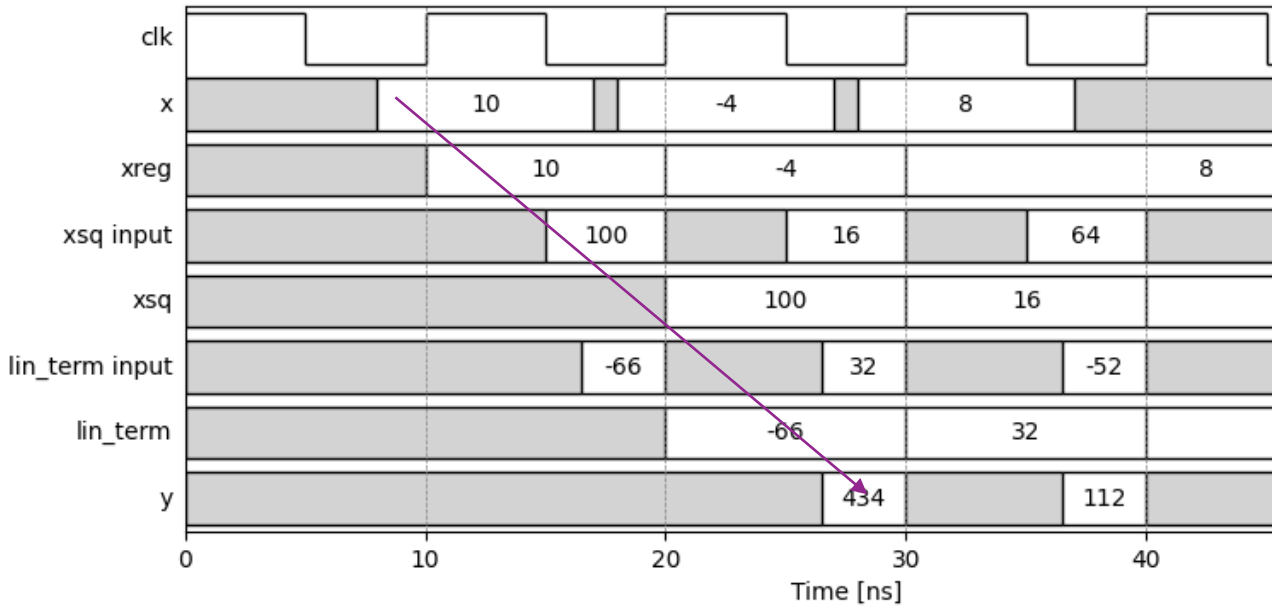
Two Cycle Timing Diagram



- Example with
  - $w = [4, -7, 5]$
- x inputs 10, -4, 8

Latency, Throughput and Pipelining

- Suppose that we run at our design at 100 MHz clock
- Latency:
  - Suppose input is valid on rising edge of clock n
  - Then output is valid before rising edge of clock $n + 2$
  - Latency = 2 clock cycles = 20 ns
- Throughput:
  - Circuit can take a new input each cycle
  - Throughput = 1 input / cycle = 100 MHz
- Process is called a pipeline
  - Circuit is processing multiple inputs at the same time



Performance Comparison

	Single Cycle Design	Two Cycle Design
Critical path	12 ns	6.5 ns
Max clock frequency	≈ 80 MHz	≈ 150 MHz
Max throughput	80 MHz	150 MHz
Latency	1 cycle = 12 ns	2 cycles = 13 ns

Pipelining benefit:
For small increase in latency, we dramatically improve throughput

Outline

- ❑ Combinational Logic
- ❑ Timing Diagrams for Combinational Logic
- ❑ Registers, Clocks and Sequential Logic
- ❑ Simulating and Synthesizing Simple Modules in Vivado