# Introduction to Hardware Design
# Basic Digital Logic: Figures and Code Snippets

Profs. Sundeep Rangan, Siddharth Garg

## 0.1 Combinational Example: ReLU with a linear input

We consider implementing the ReLU function:

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

where $x$, $a$, and $b$ are integer inputs. This function arises commonly in machine learning. We first consider a purely combinational implementation of this function. A possible SystemVerilog implementation is as follows:

```
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
```

When this module is synthesized, a potential block diagram is as shown in Figure **??**.

## 0.2 Adder

Implementation of $y = a + b + c + d$. First, we show a serial implementation:

Next, we show a parallel implementation:

$$z_1 = a + b$$
$$z_2 = c + d$$
$$y = z_1 + z_2$$

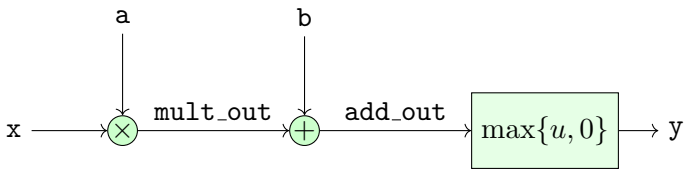


Figure 1: Fully combinational block diagram for ReLU with a linear input.

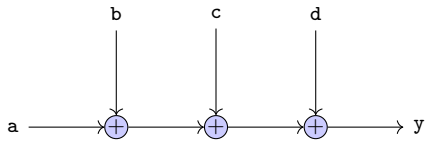


Figure 2: Serial implementation of an adder.

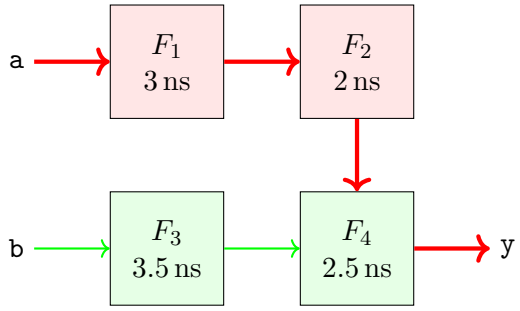


Figure 3: Critical path of a circuit. Critical path highlighted in red.

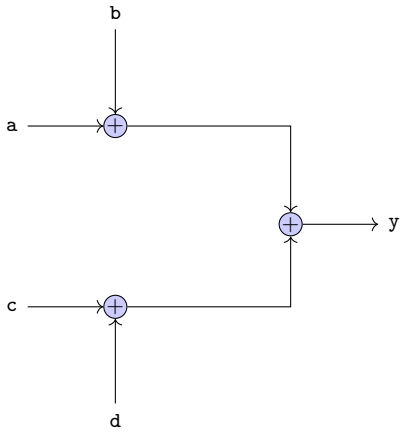


Figure 4: Parallel implementation of an adder.

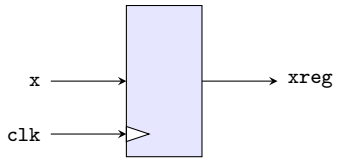


Figure 5: Basic register block diagram.

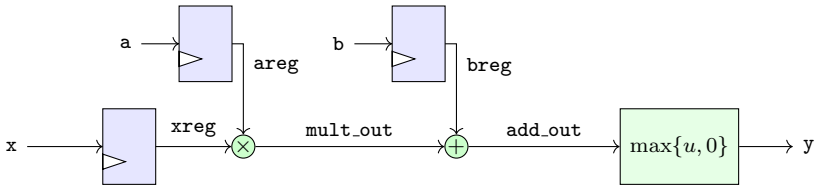


Figure 6: Synchronous version of ReLU with a linear input.

## 0.3 Register

Draw a basic register.

## 0.4 Sequential ReLU with a linear input

The sequential implementation of the ReLU with a linear input is as follows:

```
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
);
  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
```

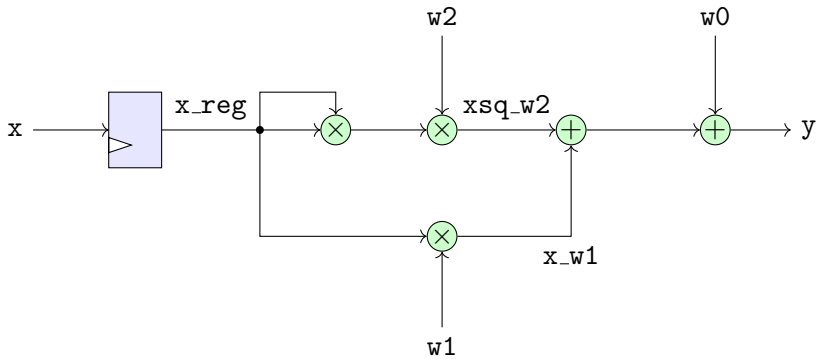


Figure 7: Single cycle implementation of a quadratic function.

## 0.5 Quadratic Function with a Single Cycle

We wish to implement the function:

$$y = w_2 x^2 + w_1 x + w_0$$

A single cycle implementation is as follows treating $w_i$ as fixed parameters is:

```
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
```

The block diagram is as shown in Figure **??**.

## 0.6 Quadratic Function with a Multi-Cycle Pipeline

A single cycle implementation is as follows treating $w_i$ as fixed parameters is:

```
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
);
```

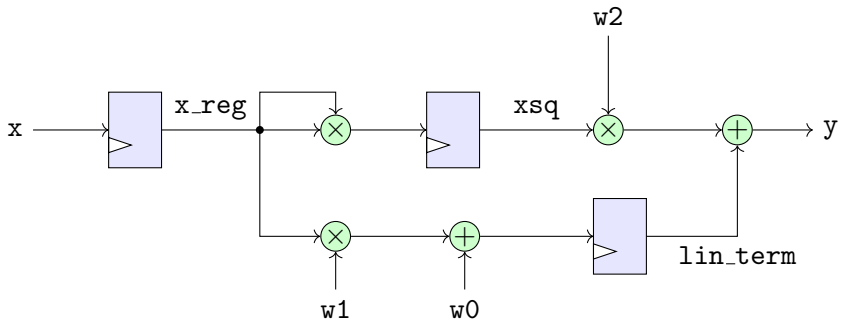


Figure 8: Two cycle implementation of a quadratic function.

```
  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
endmodule
```