

Introduction to Hardware Design

Basic Digital Logic: Figures and Code Snippets

Profs. Sundeep Rangan, Siddharth Garg

0.1 Data types

Synthesizable 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
```

These data types are synthesizable, meaning they can be directly mapped to hardware. Non-synthesizable data types, such as ‘int’ and ‘shortint’, cannot be directly mapped to hardware and are typically used for simulation purposes only.

```
int a; // 32-bit signed integer
shortint b; // 16-bit signed integer
unsigned int c; // 32-bit unsigned integer
unsigned shortint d; // 16-bit unsigned integer
```

0.2 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 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
```

When this module is synthesized, a potential block diagram is as shown in Figure 1.

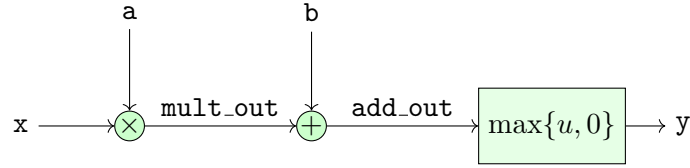


Figure 1: Fully combinational block diagram for ReLU with a linear input.

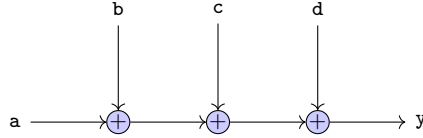


Figure 2: Serial implementation of an adder.

0.3 Counter

```

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

```

0.4 Adder

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

Next, we show a parallel implementation:

$$z_1 = a + b$$

$$z_2 = c + d$$

$$y = z_1 + z_2$$

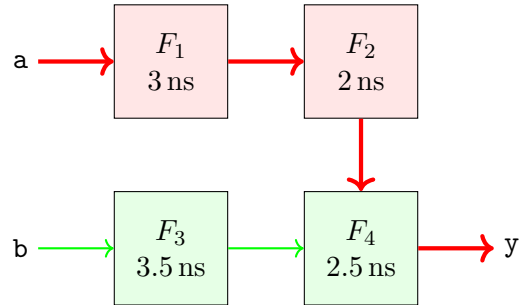


Figure 3: Critical path of a circuit. Critical path highlighted in red.

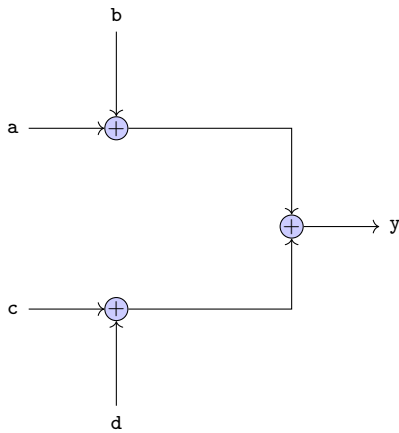


Figure 4: Parallel implementation of an adder.

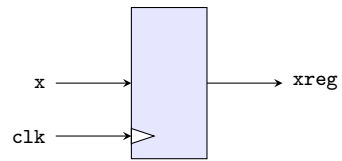


Figure 5: Basic register block diagram.

0.5 Register

Draw a basic register.

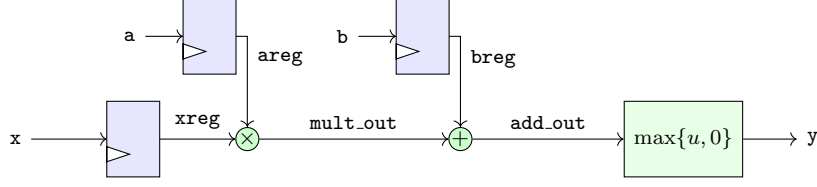


Figure 6: Synchronous version of ReLU with a linear input.

0.6 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,
    input logic signed [31:0] x,
    input logic signed [31:0] a,
    input logic signed [31:0] b,
    output logic signed [31:0] y
);
    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

endmodule

```

0.7 Quadratic Function with a Single Cycle

We wish to implement the function:

$$y = w_2x^2 + w_1x + w_0$$

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,

```

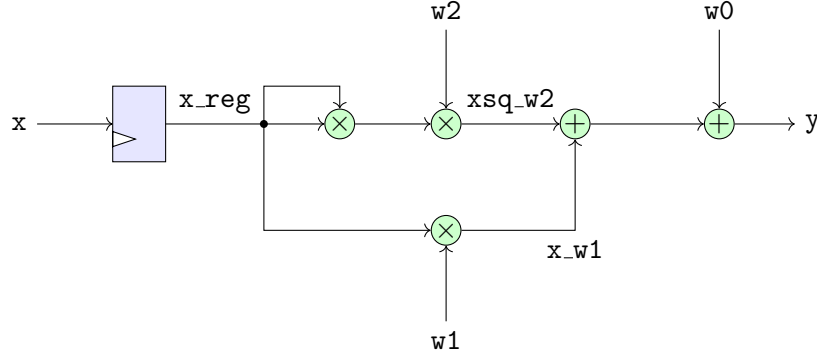


Figure 7: Single cycle implementation of a quadratic function.

```

input logic int x,
output logic int y
);
int xreg;
always_ff @(posedge clk) begin
    xreg <= x;
end
always_comb begin
    logic signed [31:0] 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 7.

0.8 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
);
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

```

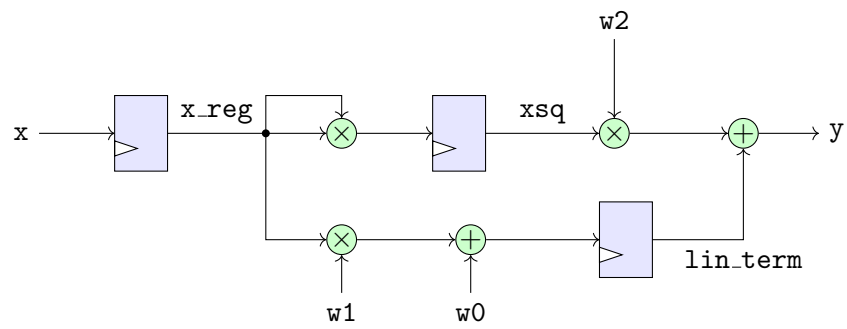


Figure 8: Two cycle implementation of a quadratic function.