CLA4 carry

$c_0 = c_{in}$

$c_1 = p_0c_0 + g_0$

$c_2 = p_1c_1 + g_1$

$c_3 = p_2c_2 + g_2$

$c_4 = p_3c_3 + g_3$

CLA Equations

$S = A \oplus B \oplus C_{in}$

$C_{out} = a'bc_{in} + b'ac_{in} + abc_{in}' + abc_{in}$

$= (a'b + ab')c_{in} + ab(c_{in} + c_{in}')$

$= (a \oplus b)c_{in} + ab$

$= pc_{in} + g$

$p = a \oplus b$

$g = ab$

Clock Ladder

Input Clock Frequency = F_{CLK}

$F_{y0} = \frac{fclk}{2}$

$F_{y1} = \frac{fclk}{4}$

$F_{y2} = \frac{fclk}{8}$ $F_{yn} = \frac{fclk}{2^{n+1}}$

$F_{y3} = \frac{fclk}{16}$

Sync a clock to an async input like a switch can lead to multiple reads per clock pulse, or unpredictable behavior by the switch. Solutions:

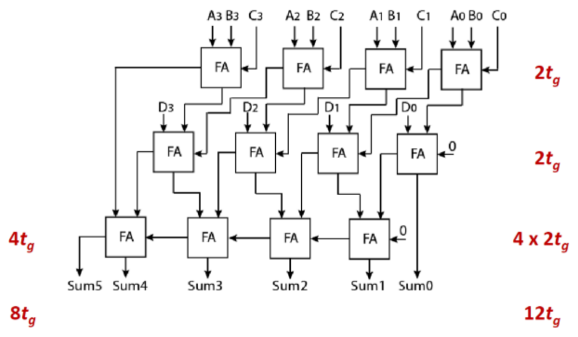
- Slow the clock speed to control unit
- Add transition states
- Use edge detection on switch

Overflow creates the wrong sign bit.

1. When adding two (+) numbers that yield a (-) number
2. When adding two (-) numbers that yield a (+) number

i.e. $7 + 7 \neq -2$ $-8 + -8 \neq 0$

In a 4x4 keypad scanning algo. Upon pressing any of the buttons on a given row, this one will send a signal to the Keypad Scanner & Encoder. Then columns are scanned. Having both the row and column, the specific button that was pressed can be determined. Code corresponding to button is outputted, along w a signal that tell whether code is ready.



$2t_g$

$2t_g$

$4t_g$ $4 \times 2t_g$

$8t_g$ $12t_g$

```
module Register #(parameter N=4)
(
    input LOAD, CLR,
    input [N-1:0] ABCD,
    output logic [N-1:0] Q
);
    always_ff @ (posedge LOAD, negedge CLR) begin
        if(CLR == 1'b0) Q <= 0;
        else
        if(LOAD == 1'b1) Q <= ABCD;
    end
endmodule
```

For 4 chained CLAs

Add Time = $4t_g + 4t_g + 4t_g + 4t_g$

One CLA4: takes $1t_g$ to compute propagate and generate; takes $2t_g$ to compute C_{out} (2 logic gates needed); and computing sum takes $1t_g$.

1 CLA4 Add Time = $4t_g$

$$[50MHz \to 5\ Hz] = \left[\frac{50*10^6}{1000} = 50,000\ Hz\right] \to \left[\frac{50,000}{100} = 500\ Hz\right] \to \left[\frac{500}{10} = 50\ Hz\right] \to \left[\frac{50}{10} = 5\ Hz\right]$$

* **Data Arrival Time** (**DAT**): the time it takes for the data to arrive at the destination register input.

- Max
- Min

* **Clock Arrival Time** (**Tclk**) : the time it takes for the clock to arrive at the destination register

- Max
- Min

* **Data Required Time** (**setup**) = Clock Arrival Time – Setup Time

- Max
- Min

* **Data Required Time** (**hold**): = Clock Arrival Time + Hold Time

- Max
- Min

* **Setup slack** = clock period + minimum data required time – max data arrival time

Dependent on frequency!

* **Hold slack** = minimum data arrival time – max data required time

NOT dependent on frequency!

* **Max clock frequency**

new period = clock period + |setup slack|

$T_{MIN} = DAT_{MAX} + \text{Setup Time} - Tclk_{MIN}$

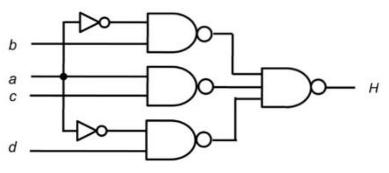
$F_{MAX} = \frac{1}{Tmin}$

* If setup slack < 0, then logic design is rather slow. Fix by increasing the period, or, in other words, decreasing frequency.

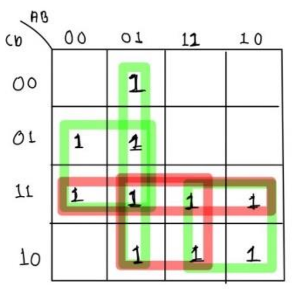
```
logic clk1;

// 2^10 = 1024
// Able to hold 1000
// input CLK = 50 MHz
divideXN #(1000, 10) name  // 50*10^6
)                          // -------  = 50,000
    .CLK(CLK)              //   1000
    .CLEAR(CLR),
    .OUt(Clk1),            // clk1 = 50,000 kHz
);
```

```
module CLK_ladder #(parameter N=4)
(
    input CLK, CLR,
    output logic [N-1:0] ladder
)
    always_ff @(posedge CLK, negedge CLR) begin
        if(CLR == 1'b0)
            ladder <= 0;
        else
            ladder <= ladder + 1'b1;
    end
endmodule
```

```
module Neg_EdgeDetect
(
    input IN, CLK,
    output logic OUT
)
    logic IN_delay

    always_ff @(negedge CLK)
        IN_delay <= in;

    assign out = ~IN & IN_delay
endmodule
```



$H = AC + \bar{A}B + \bar{A}D$



CD \ AB	00	01	11	10
00		1		
01	1	1		
11	1	1	1	1
10		1	1	1

$H = AC + \bar{A}B + \bar{A}D + BC + CD$

Three Static 1 hazards

0111 ⟷ 1111

0110 ⟷ 1110

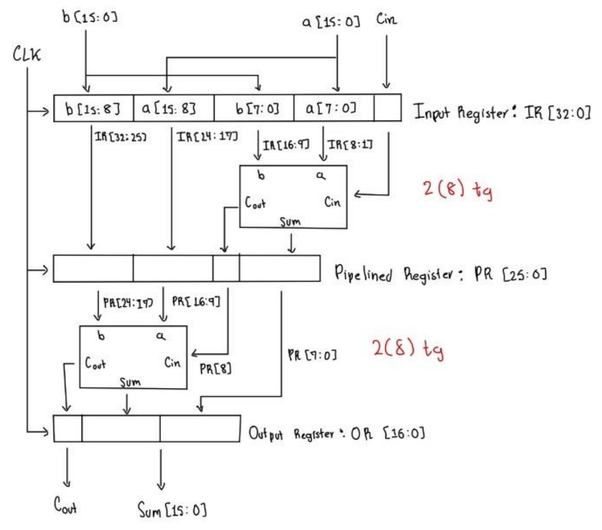
0011 ⟷ 1011

Multiplexed displays Advantages:

- fewer pin assignments required
- Simpler and shorter code
- easier to implement
- separate Anodes as supposed to single
- less wiring than non-multiplexed
- more efficient

Disadvantages:

- clock rate for cycling thru digits must be set, otherwise flickering
- dimmer than non-multiplexed displays



For an n-bit ripple carry adder, the add time is $2*n*t_g$

```
// Test Bench
timescale 1ns/100ps
module muxTester (output out);
   logic [2:0] count;
   logic muxOut;

   mux UUT (muxOut, count[2], count[1], count[0]);

   initial begin
      $monitor ($time, "a b sel = %b, muxOut = %b", count, muxOut);

      for (count = 0; count != 3'b111; count ++)
         #10;

      #10 $finish;          // After test is done wait 10 ns

   end
endmodule : muxTester

// Module to be tested
module mux
   (output logic f,
   input logic a, b, sel);

   and #2    g1 (f1, a, n_sel),
             g2 (f2, b, sel);
   or  #2    g3 (f, f1, f2);
   not       g4 (n_sel, sel);
endmodule: mux
```

```
module RCAddSubReg
(
   input AddSub,
   input clock,
   input [3:0] A, B,
   output [3:0] S_out,
   output Cout
);
   logic [3:0] reg_A,reg_B,reg_S;
   logic reg_C;

   logic [4:0] C;
   logic [3:0] S;

   assign C[0] = AddSub;
   assign Cout = reg_C;
   assign S_out = reg_S;

   always_ff @(negedge clock) begin
      reg_A <= A; reg_B <= B;        //load input registers
      reg_S <= S; reg_C <= C[4];     //load output registers
   end

   FAbehavSV s0 (reg_A[0], AddSub^reg_B[0], C[0], S[0], C[1]);
   FAbehavSV s1 (reg_A[1], AddSub^reg_B[1], C[1], S[1], C[2]);
   FAbehavSV s2 (reg_A[2], AddSub^reg_B[2], C[2], S[2], C[3]);
   FAbehavSV s3 (reg_A[3], AddSub^reg_B[3], C[3], S[3], C[4]);
endmodule
```

```
module CirBuff
(
   input CLK, CLR,
   output logic [15:0] data2disp,
   output logic [0:63] msg_out
);

   parameter msg = 64'h0123456789ABCDEF;

   always_ff @(negedge CLK, negedge CLR) begin
      if(CLR == 1'b0)
         msg_out <= message;
      else begin
         msg_out [0:59] <= msg_out [4:63];
         msg_out [60:63] <= msg_out [0:3];
      end
   end

   assign data2disp = msg_out [0:15];
endmodule
```