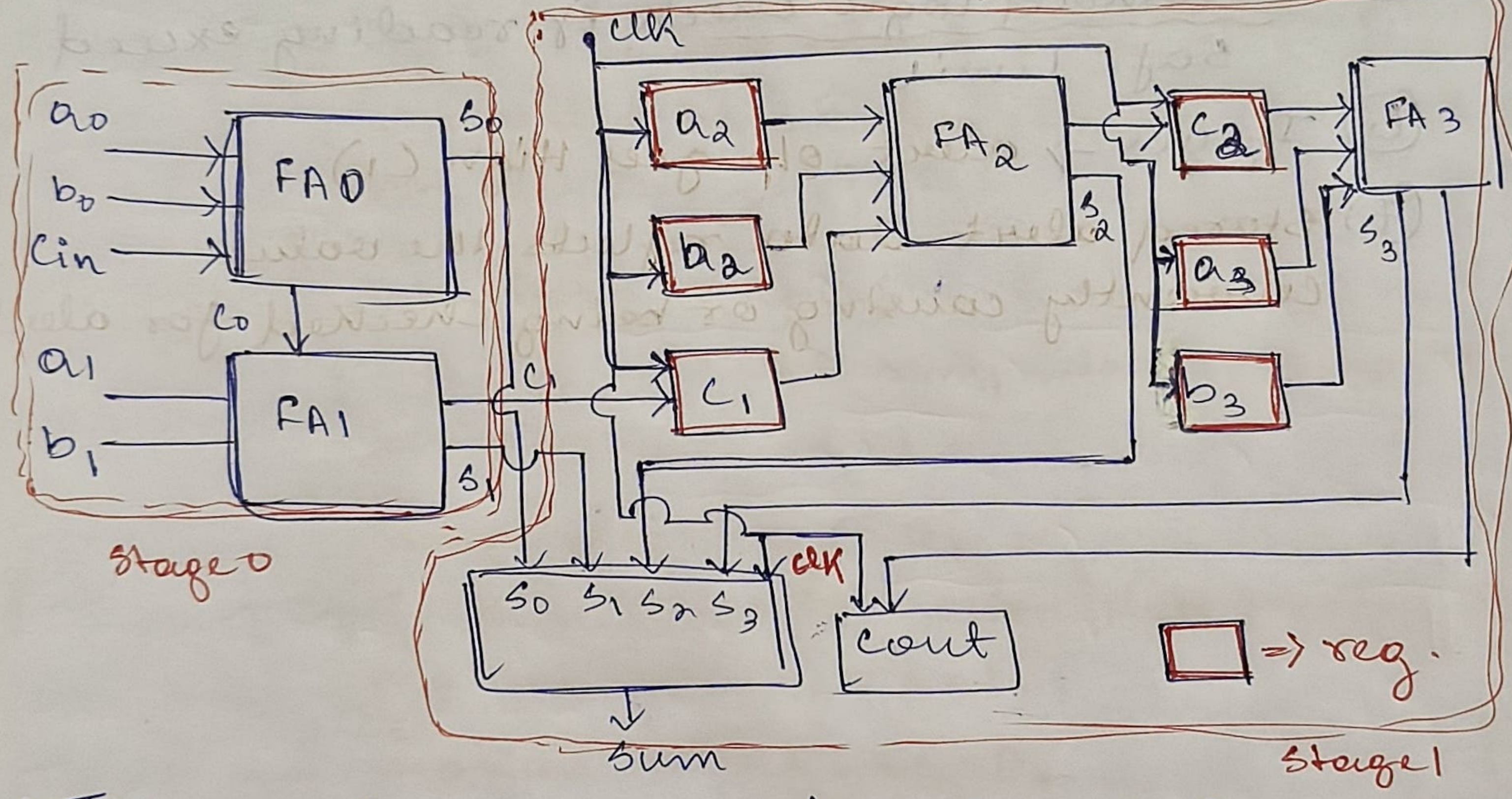
2-Stage Pipelined 4-bit Ripple carry Adder (RCA)

- It implements a 4-bit adder using 4-full adder, but split into 2 pipeline stages

⇒ This pipelining is done in order to "increase clock frequency (reduce the max. combinational path)"



- It takes 2 4bit operands like 4-bit RCA along with Cin and produces Sum and Cout
- Arrangements :- FA0 → FA1 ⇒ $\boxed{\text{Pipeline}}$ ⇒ FA2 → FA3 → o/p

* chain is pipelined betn bit1 and bit2 so the carry prop. for upper half is handled in next clk stage

```
module RCA_2_stage (
  input clk,
  input wire [3:0] a,b,
  input cin,
  Output reg [3:0] sum,
  Output reg cout );
```

• stage 0 comb. wires

```
    wire s0_w, s1_w;
    wire c0_w, c1_w;
```

• pipelined reg (stage 0 → 1)

```
  reg a2_reg, b2_reg, a3_reg, b3_reg;
  reg c1_reg;
  reg s0_reg, s1_reg;
```

note

⇒ After FA0, FA1 produce their result, the design registers - (1) partial sum s0 and s1
(2) The carry c1 (from FA1)
(3) The upper operand bits a[2], b[2], a[3], b[3]

⇒ All the above are reg. and values stored in it and to be noted all of them controlled using clk signal

⇒ And all these values feed to FA2 and FA3

- Stage1 combinational wires

```
wire s2_w , s3_w ;
wire c2_w , c3_w ;
```

- Stage 0 combinational - LSB half

```
full_adder FA0 (.a(a[0]), .b(b[0]), .cin(cin)
                 .sum(s0_w), .cout(c0_w));
full_adder FA1 (.a(a[1]), .b(b[1]), .cin(c0_w),
                 .sum(s1_w), .cout(c1_w));
```

c1_w is carry that must be driven to upper half and it is registered before use.

- Stage1 combinational - MSB half

```
full_adder FA2(.a(a2_reg), .b(b2_reg), .cin(c1_reg),
                 .sum(s2_w), .cout(c2_w));
full_adder FA3(.a(a3_reg), .b(b3_reg), .cin(c2_w)
                 .sum(s3_w), .cout(c3_w));
```

FA2 and FA3 implement the upper half using the reg. high bits and registered carry c1_reg
Those reg. were loaded at the prev. clk edge so FA2 and FA3 evaluate on fresh, stable signal

• pipeline and o/p reg block

```
always @ (posedge clk) begin
    s0_reg <= s0_w;     } stage 0 result into
    s1_reg <= s1_w;     }   the pipeline reg.
    c1_reg <= c1_w;     } then used in stage 1

    a2_reg <= a[2];     } Reg. upper operand bits
    b2_reg <= b[2];     } so that stage 1 FAs will
    a3_reg <= a[3];     } see stable operand in
    b3_reg <= b[3];     } stage 1

    sum[0] <= s0_reg;   } assigned from reg and
    sum[1] <= s1_reg;   } will be delayed from s0_w
                          and s1_w due to clk.
    sum[2] <= s2_w;     } assigned from wire from
    sum[3] <= s3_w;     }  comb. ckt
    cout   <= c3_w;     } cout from c3_w
end
```

• full_adder module

```
module full_adder (
    input a, b, cin,
    output sum, cout );
  assign sum = a ^ b ^ cin;
  assign cout = (a & b) | (b & cin) | (cin & A);
end
```

• Importance and Behaviour of ckt

① Pipeline stages :-

Stage 0 ⇒ FA0 & FA1 compute LSB result & C1

Stage 1 ⇒ FA2 & FA3 compute MSB result using reg. operands & reg carry from stage 0

② Increased Max. clk freq. :-

Bcz the longest comb. path is split into 2 shorter paths. leading to critical path redⁿ allowing higher clk freq.

③ Throughput :- (amt of signal passing thro. system)

Once pipeline is filled, the design can accept a new pair of operands each clk and produce one result / clk (steady state throughput = 1 result / clk)

④ Latency :- i/p (a[3:0], b[3:0], cin) presented at time T are fully reflected at sum [3:0], Cout after two-clk edges.

✱ { ✱ (pipeline latency = 2 cycles) }

Reason :- Stage 0 result into Reg. ⇒ 1 clk
Stage 1 o/p is sampled / propagated ⇒ 1clk