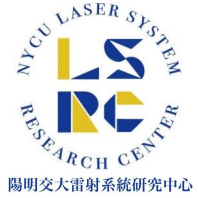
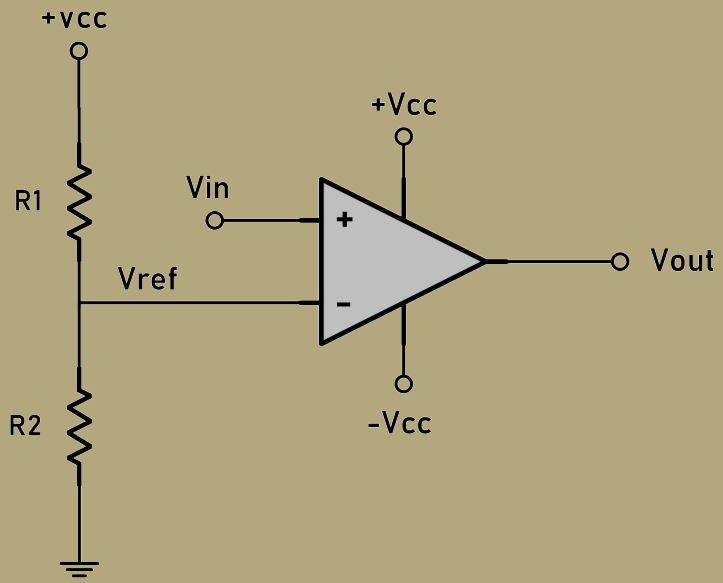
Class of Digital Design with FPGA

Comparison of FSM-based RCA/CLA

David

Nov. 22, 2025





Fixed Voltage Divider Reference

Contents

1. Adder

Study the full adder, ripple carry adder (RCA), and carry lookahead adder (CLA).

2. FSM

Compare Mealy and Moore finite state machines (FSMs) and their encoding methods.

3. Comparator

Start with gate-level building blocks and compare the 1-, 2-, and 4-bit magnitude comparator.

4. Simulation

Combine adders, comparators and FSM to analyze 8-bit and 128-bit RCA/CLA in ModelSim.

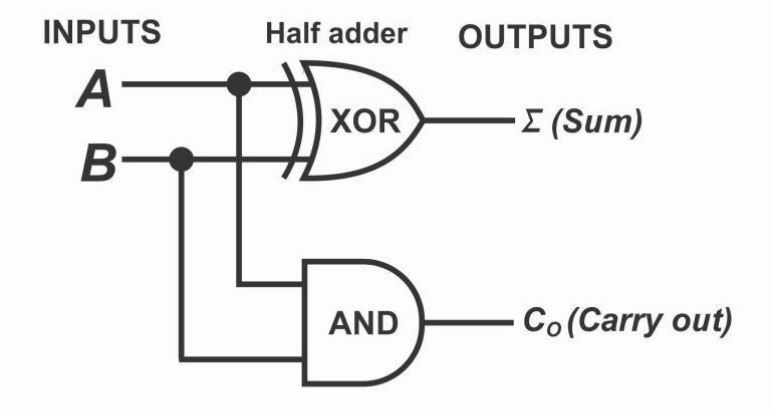
①

Adders

- ➔ half adder
- ➔ full adder
- ➔ ripple-carry adder (RCA)
- ➔ carry-lookahead adder (CLA)

Half Adder

- A half adder is a digital circuit that adds two single binary digits (bits) and produces a two-bit sum: a sum bit and a carry bit.
- Logic Gates: one XOR gate (for the sum) and one AND gate (for the carry).
- Sum (S): The result of an XOR gate
  S = A ⊕ B
- Carry (C): The result of an AND gate
  C = A • B
- A key limitation is that it cannot account for a carry-in bit from a previous addition, which is handled by a full adder.

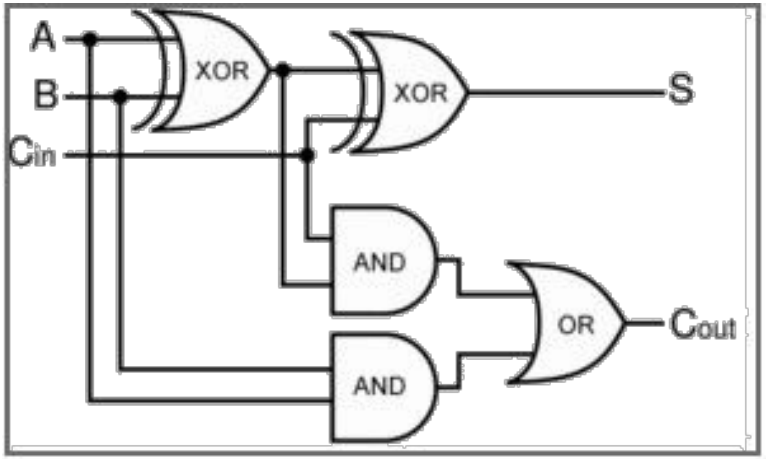


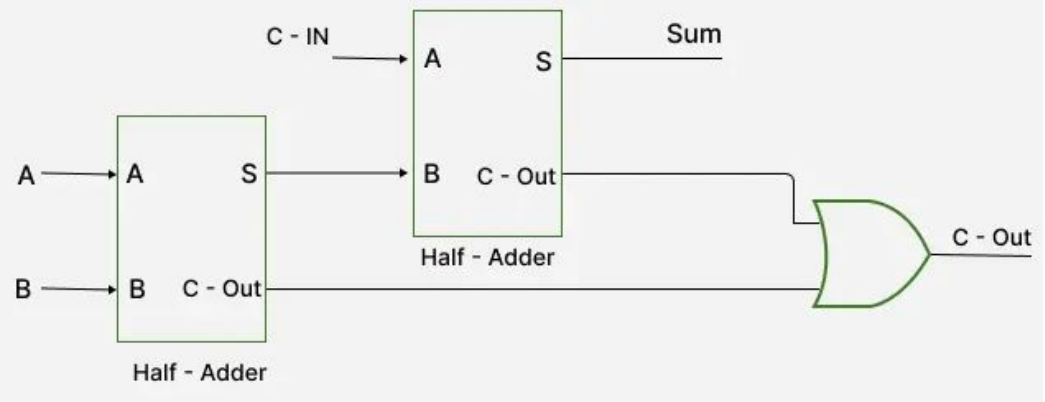
Truth Table

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

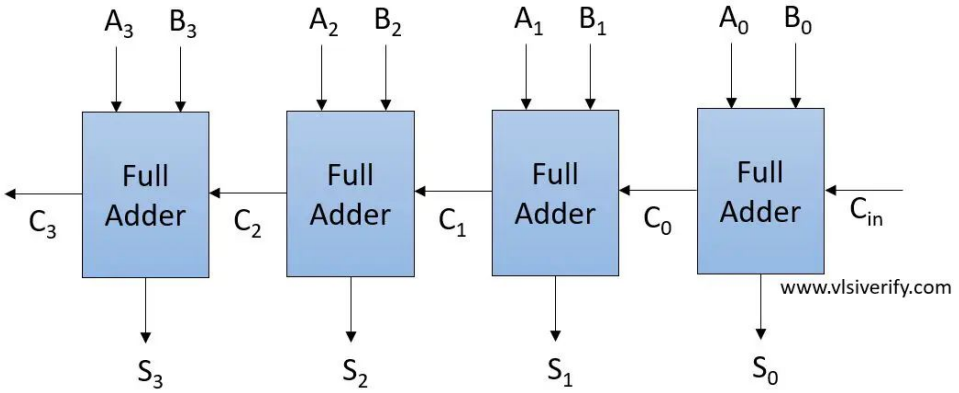
Full Adder

- A full adder is a digital circuit that adds three single-bit binary numbers and produces a two-bit output: a sum and a carry-out.
- Sum
  S = A ⊕ B ⊕ Cin
- Carry-out
  C = AB + ACin + BCin
  = AB + Cin (A ⊕ B)





Ripple-Carry Adder (RCA)



- A ripple-carry adder is a digital circuit that adds two binary numbers by **cascading** multiple **full adder** circuits.
- The carry-out of the first full adder is connected to the carry-in of the second, and so on. The carry "ripples" from the least significant bit to the most significant bit.
- **Propagation delay**: Each full adder must wait for the carry-in from the previous stage, so a significant propagation delay increases with the number of bits.

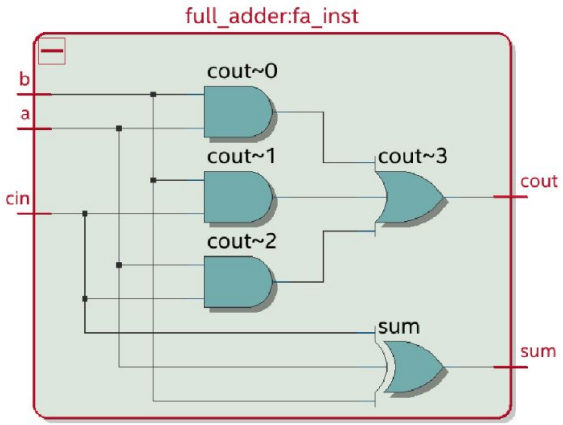
8-bit RCA: Verilog

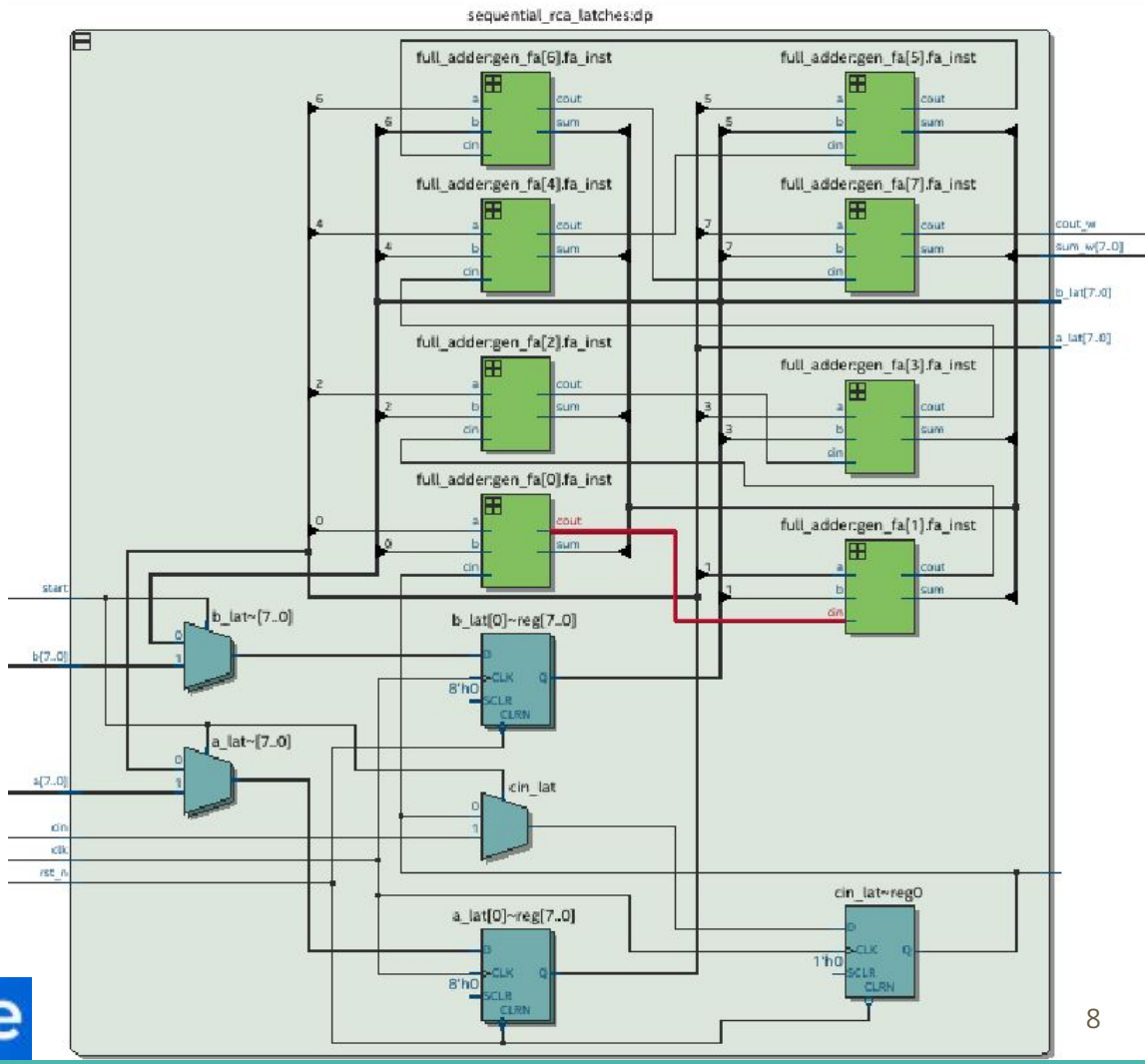
```verilog
// 1-bit full adder (combinational)
module full_adder (
    input  wire a,
    input  wire b,
    input  wire cin,
    output wire sum,
    output wire cout
);
    assign sum  = a ^ b ^ cin;
    assign cout = (a & b) | (b & cin) | (a & cin);
endmodule
```

```verilog
// sequential_rca_latches: latches inputs, instantiates RCA
module sequential_rca_latches
#( parameter integer WIDTH = 8 )
(
    input  wire                clk,
    input  wire                rst_n,
    input  wire                start,
    input  wire [WIDTH-1:0]    a,
    input  wire [WIDTH-1:0]    b,
    input  wire                cin,
    output reg  [WIDTH-1:0]    a_lat,
    output reg  [WIDTH-1:0]    b_lat,
    output reg                 cin_lat,
    output wire [WIDTH-1:0]    sum_w,
    output wire                cout_w
);
    wire [WIDTH:0]   carry_w;
    assign carry_w[0] = cin_lat;
    genvar i;
    generate
        for (i = 0; i < WIDTH; i = i + 1) begin : gen_fa
            // instantiate the same full_adder module used previously
            full_adder fa_inst (

                .a   (a_lat[i]),
                .b   (b_lat[i]),
                .cin (carry_w[i]),
                .sum (sum_w[i]),
                .cout(carry_w[i+1])
            );
        end
    endgenerate
    assign cout_w = carry_w[WIDTH];
endmodule
```

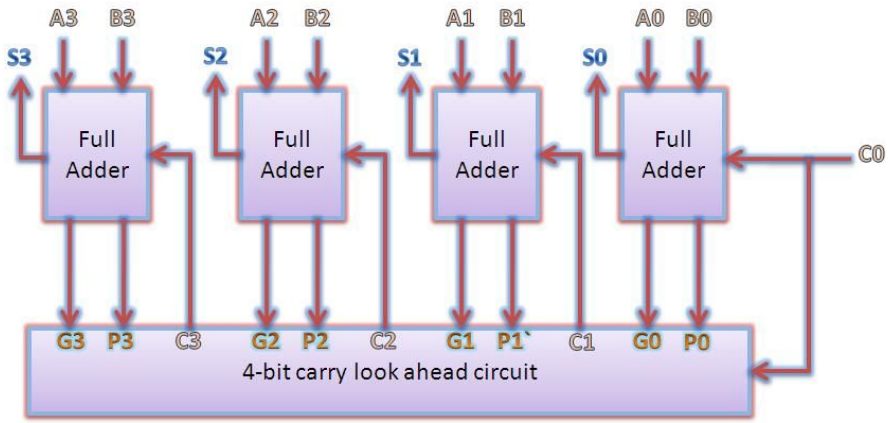
8-bit RCA: Netlist





Intel® Quartus® Prime

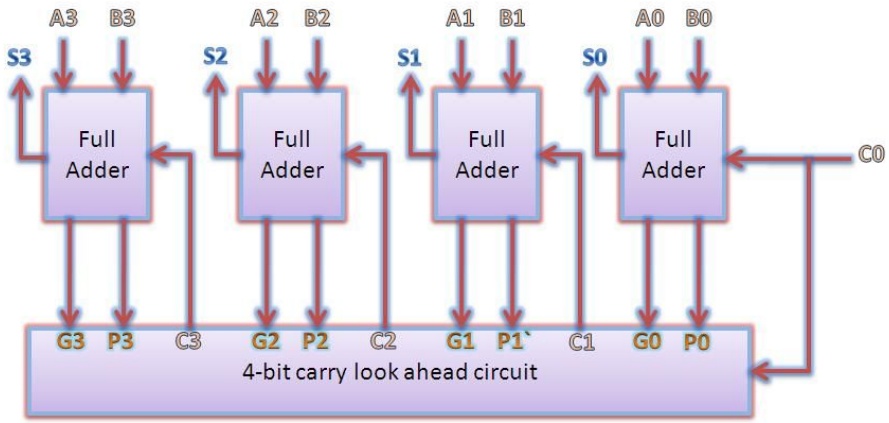
Carry-Lookahead Adder (CLA)



- A CLA uses separate logic to calculate carries in advance. It does this by creating two signals (G and P) for each bit.
- Generate G : A carry is generated if both input bits are one ($Gi = Ai \cdot Bi$).
- Propagate P : A carry is propagated if at least one of the input bits is one and the carry-in is one ($Pi = Ai \oplus Bi$).
- Benefits: much faster carry determination for wide adders; useful in high-speed ALUs and CPUs.
- **Trade-offs**: more hardware (extra gates/wires) and more complex wiring compared with simple ripple-carry adders.

Carry-Lookahead Adder (CLA)

Example: Assume inputs are A = 1011, B = 0110, Cin = 0.



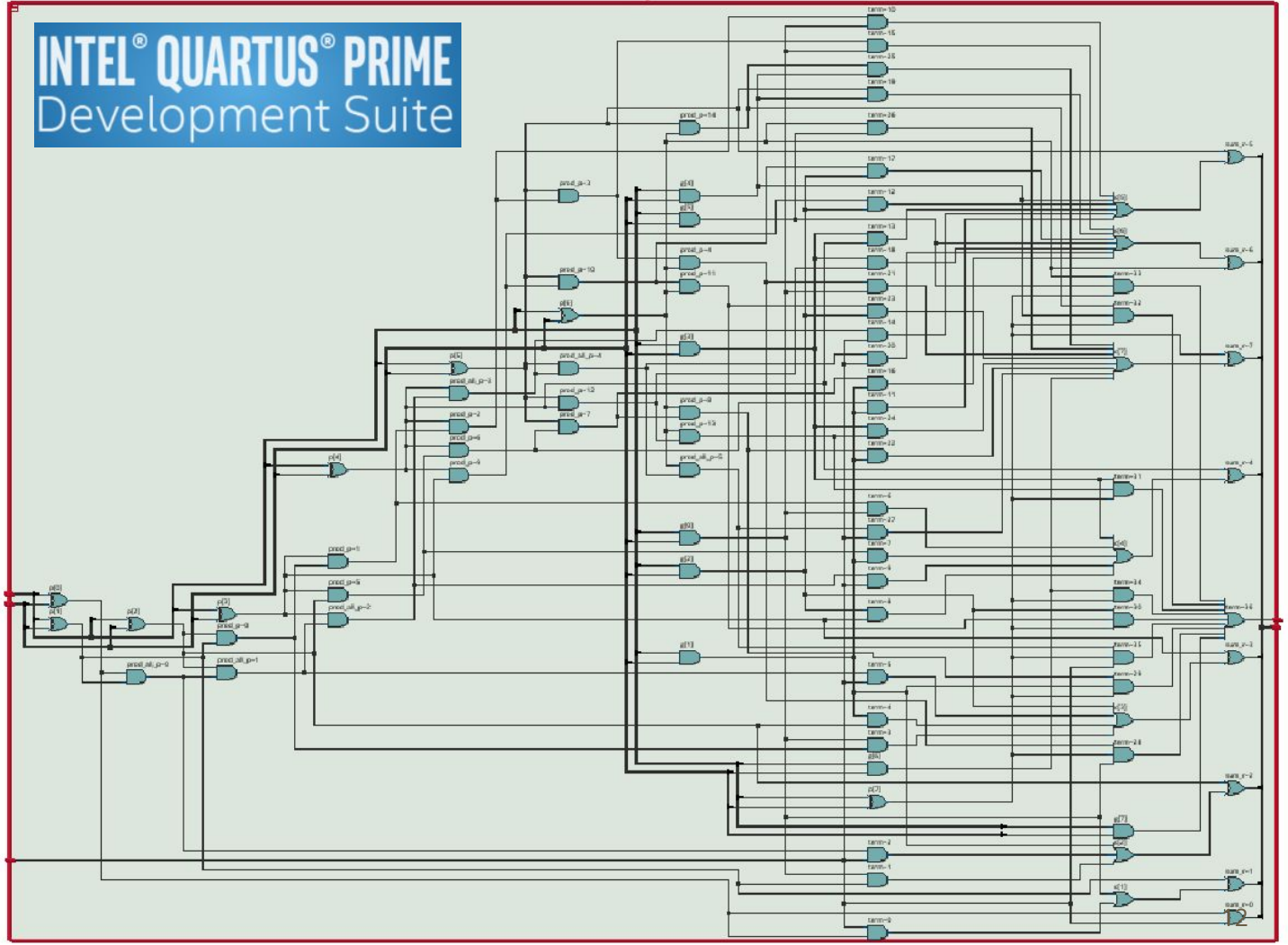
- A = $a_3a_2a_1a_0$ = 1011
- B = $b_3b_2b_1b_0$ = 0110
- Compute g_i and p_i for each bit:
  - $g_0 = a_0 \cdot b_0 = 1 \cdot 0 = 0,\ p_0 = a_0 \oplus b_0 = 1 \oplus 0 = 1$
  - $g_1 = 1 \cdot 1 = 1,\ p_1 = 1 \oplus 1 = 0$
  - $g_2 = 0 \cdot 1 = 0,\ p_2 = 0 \oplus 1 = 1$
  - $g_3 = 1 \cdot 0 = 0,\ p_3 = 1 \oplus 0 = 1$
- Compute carries using CLA formulas with $c_0 = 0$:
  - $c_1 = g_0 + p_0 \cdot c_0 = 0 + 1 \cdot 0 = 0$
  - $c_2 = g_1 + p_1g_0 + p_1p_0c_0 = 1 + 0 \cdot 0 + 0 \cdot 1 \cdot 0 = 1$
  - $c_3 = g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0 = 0 + 1 \cdot 1 + 1 \cdot 0 \cdot 0 + 1 \cdot 0 \cdot 1 \cdot 0 = 1$
  - $c_4 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_0 = 0 + 1 \cdot 0 + 1 \cdot 1 \cdot 1 + \cdots = 1$
- Compute sums:
  - $s_0 = p_0 \oplus c_0 = 1 \oplus 0 = 1$
  - $s_1 = p_1 \oplus c_1 = 0 \oplus 0 = 0$
  - $s_2 = p_2 \oplus c_2 = 1 \oplus 1 = 0$
  - $s_3 = p_3 \oplus c_3 = 1 \oplus 1 = 0$
- Sum bits $s_3s_2s_1s_0$ = 0 0 0 1 with final carry $c_4 = 1$, so the full result is 1 0001 (binary) = 17 (decimal). The CLA produced carries and sums using the generate/propagate logic rather than waiting bit-by-bit propagation

8-bit CLA: Verilog

```
3   module cla8
4   #(
5       parameter integer WIDTH = 8
6   )
7   (
8       input  wire [WIDTH-1:0] a,
9       input  wire [WIDTH-1:0] b,
10      input  wire             cin,
11      output wire [WIDTH-1:0] sum,
12      output wire             cout
13  );
14
15      // internal signals
16      // propagate p
17      wire [WIDTH-1:0] p = a ^ b;
18      // generate g
19      wire [WIDTH-1:0] g = a & b;
20      reg  [WIDTH:0]   c;
21      reg  [WIDTH-1:0] sum_r;
22
23      integer i, k, m;
24      reg term;
25      reg prod_p;
26      reg prod_all_p;
```

```
28      // combinational carry computation
29      always @* begin
30          c = { (WIDTH+1){1'b0} };
31          c[0] = cin;
32          for (i = 1; i <= WIDTH; i = i + 1) begin
33              term = 1'b0;
34              for (k = 0; k <= i-1; k = k + 1) begin
35                  prod_p = 1'b1;
36                  for (m = k+1; m <= i-1; m = m + 1)
37                      prod_p = prod_p & p[m];
38                  term = term | (prod_p & g[k]);
39              end
40              prod_all_p = 1'b1;
41              for (m = 0; m <= i-1; m = m + 1)
42                  prod_all_p = prod_all_p & p[m];
43              term = term | (prod_all_p & c[0]);
44              c[i] = term;
45          end
46
47          for (i = 0; i < WIDTH; i = i + 1)
48              sum_r[i] = p[i] ^ c[i];
49      end
50
51      assign sum  = sum_r;
52      assign cout = c[WIDTH];
53
54  endmodule
```

8-bit CLA: Netlist



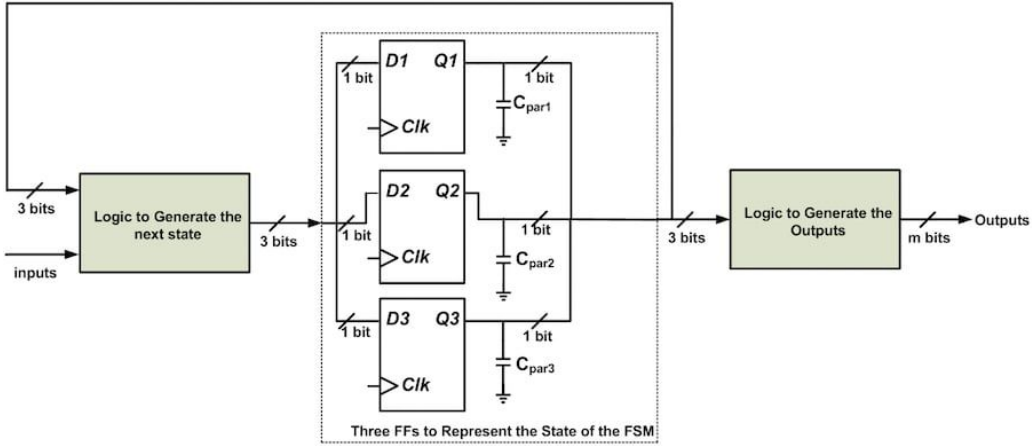
②

FSM

(Finite State Machine)

- ➔ State Encoding
- ➔ Moore Machine
- ➔ Mealy Machine

State Encoding



- **Binary encoding** assigns compact binary numbers to states.
- **Gray encoding** arranges codes so adjacent states differ by one bit, reducing simultaneous bit toggles and lowering transient hazards.
- **One-hot encoding** is a method of representing characters or words by a vector where only one element is set to one and all others are zero.

Attribute	Binary	Gray	One-hot
Flip-flops used	**Minimal**	Minimal (same width as binary)	**Many** (one per state)
Combinational logic	**Higher**	Moderate	**Lowest**
Worst-case bit toggles	Many	**1 between adjacent states**	Many (only one hot)
Timing / max frequency	Moderate	Better for transitions	**Best**
LUT/routing pressure	Moderate	Moderate	**Higher routing**
Power	Lower FF count; more switching	Lower switching for sequential changes	**Higher FF static power**
Best FPGA use cases	Dense state space, area constrained	Counters, interfaces, ADC/encoder sampling	High-speed FSMs, simple control paths

Encoding Methods in FPGA

- Choose **one-hot** when speed and simple **timing** closure matter and you can afford FFs (small state machines or performance-critical controllers).
- Use **binary** when area and FF count are constrained or the FSM is large and not timing critical.
- Pick **Gray** for **counters**, **ADC** interfaces, or sampled signals where single-bit transitions reduce **glitches** and metastability risk.

Finite State Machine (FSM)

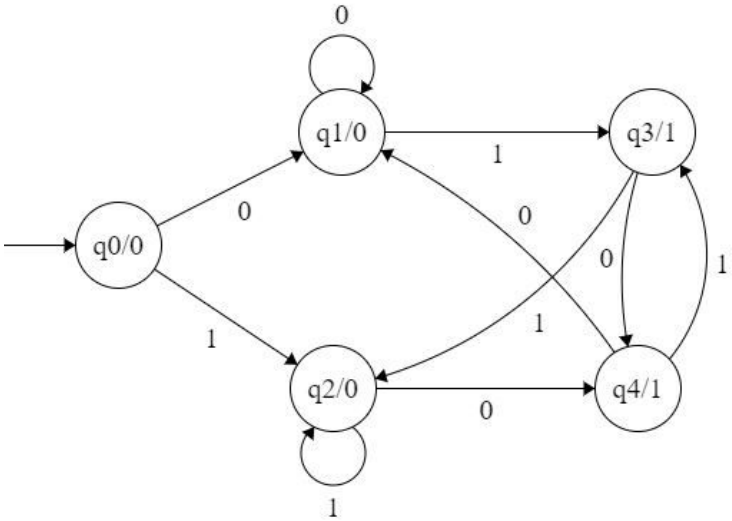
- A finite state machine is an abstract machine that can be in exactly one of a finite number of states at any time.
- The machine changes state in response to inputs; those changes are called transitions.
- An FSM is fully specified by its set of **states**, its **initial state**, the **input alphabet (events)**, and the **transition** rules that map state+input to the next state.

Types of FSMs

- **Deterministic finite automaton (DFA)** — for each state and input there is exactly one defined next state.
- **Nondeterministic finite automaton (NFA)** — a state+input may allow multiple possible next states.
- **Mealy and Moore machines** — hardware/software variants that differ in whether outputs depend on state only (Moore) or on state and input (Mealy).

Moore Machine

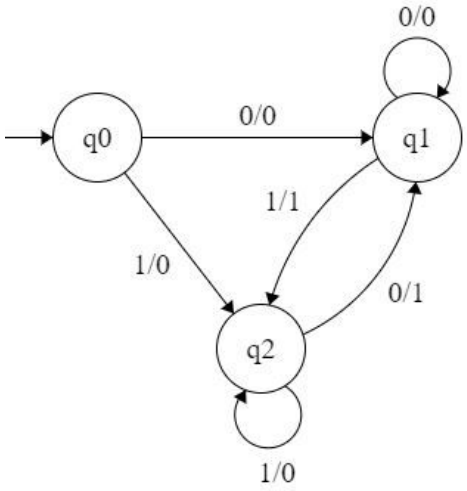
A Moore machine whose output depends on the **present state** is defined by 6-tuples (Q, Σ, Δ, δ, λ, q0):
Q is a finite set of states; **q0** is the initial state.
∑ is the input alphabet; **Δ** is the output alphabet.
δ is the transition function which maps **Q×∑ → Q**.
λ is the output function which maps **Q → Δ**.



(Example) Input: 1,1
Transition: δ (q0,1,1) → δ(q2,1) → q2
Output : 0, 0, 0

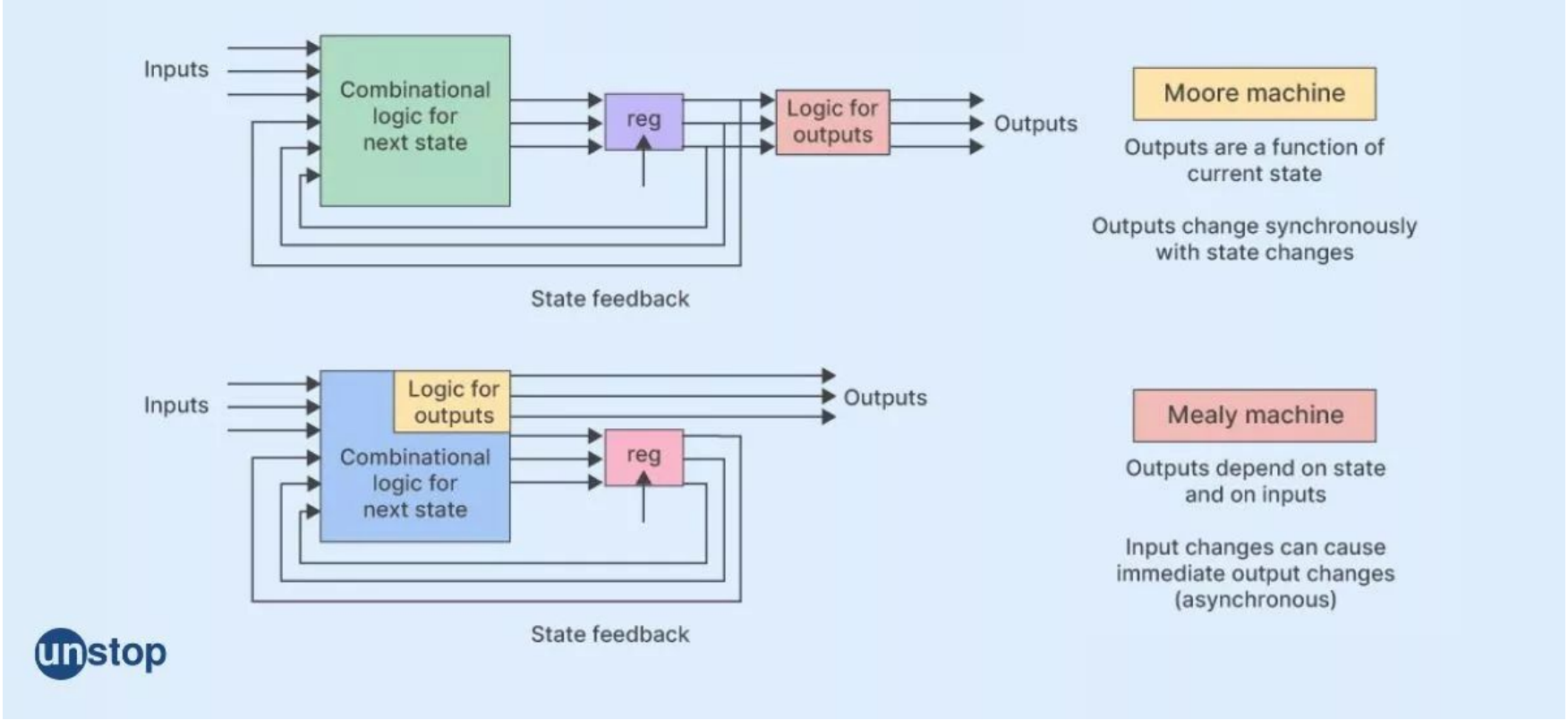
Mealy Machine

A Mealy machine whose output depends on the **present state** and **current input symbol** is defined by 6-tuples (Q, Σ, Δ, δ, λ, q0):
Q is a finite set of states; **q0** is the initial state.
∑ is the input alphabet; Δ is the output alphabet.
δ is the transition function which maps **Q×∑ → Q**.
λ is the output function that maps **Q×∑→ Δ**.



(Example) Input: 1,1
Transition: δ (q0,1,1) → δ(q2,1) → q2
Output : 0, 0

Moore & Mealy Machines



Source: https://unstop.com/blog/difference-between-mealy-and-moore-machine

Moore & Mealy Machines

Mealy Machine	Moore Machine
The output depends both on the present state and the present input.	The output depends only on the present state.
It generally has fewer states than Moore's machine.	Generally, a Moore machine has more states.
The hardware requirement is comparatively more for circuit implementation.	Less hardware requirement for circuit implementation.
They react faster to input.	Moore machines react slower to input.
The output generation is asynchronous.	The output and state of the change of the device are synchronous with the clock edge.
Comparatively difficult to design.	The output and state of the change of the device are synchronous with the clock edge.

Source: https://www.naukri.com/code360/library/moore-and-mealy-machine

Verilog Example

Moore Machine

- The output **detect** is driven solely from the current state (**detect_combinational = (mstate == M_DET)**).
- States
  - **M0** — idle / expecting `0`.
  - **M1** — saw `0`, now expecting `1`.
  - **M_DET** — detection state; **assert** `detect = 1` while in this state.
- State transition table

State	Meaning	data_in = 0 → next	data_in = 1 → next	detect (output)
M0	idle / expecting 0	M1	M0	0
M1	saw 0, expect 1	M1	M_DET	0
M_DET	detection state	M1	M0	1

Mealy Machine

- The output **detect_next** is driven inside the combinational **always @*** based on both the current state and the current input (**S1** and **data_in == 1'b1**),
- States
  - **S0** — idle / waiting for a `0`.
  - **S1** — saw a `0`, waiting for a `1` to detect the pattern `01`.
- State transition table

Current State	Input (data_in)	Next State	detect_next (combinational)	detect (registered next clock)
S0	0	S1	0	0
S0	1	S0	0	0
S1	0	S1	0	0
S1	1	S0	1	1

Moore Machine

Verilog Example

Mealy Machine

```
input  wire clk,
input  wire rst_n,
input  wire data_in,
output reg  detect
```

```
16      // State type and encoding
17      typedef enum logic [1:0] {
18          M0      = 2'b00, // idle/expecting 0
19          M1      = 2'b01, // saw 0, now expecting 1
20          M_DET   = 2'b10  // detection state: assert detect = 1
21      } mstate_t;
22
23      // Current and next state signals
24      mstate_t mstate;
25      mstate_t next_state;
26
27      // combinational next-state and output
28      logic detect_combinational;
```

```
16      // State type and symbolic names
17      typedef enum logic [1:0] { S0 = 2'b00, S1 = 2'b01 } state_t;
18      state_t state;       // current state
19      state_t next_state;  // next state computed combinationally
20
21      // Combinational signal for the Mealy output; will be registe
22      logic detect_next;
```

Moore Machine

```
27        // combinational next-state and output
28        logic detect_combinational;
29        always @(*) begin
30            next_state = mstate;
31            detect_combinational = 1'b0;
32            case (mstate)
33                M0:    next_state = (data_in==1'b0) ? M1 : M0;
34                M1:    next_state = (data_in==1'b1) ? M_DET : M1;
35                M_DET: next_state = (data_in==1'b0) ? M1 : M0;
36                default: next_state = M0;
37            endcase
38            detect_combinational = (mstate == M_DET);
39        end
40
41        // registered state and output
42        always_ff @(posedge clk or negedge rst_n) begin
43            if (!rst_n) begin
44                mstate <= M0;
45                detect  <= 1'b0;
46            end else begin
47                mstate <= next_state;
48                detect  <= detect_combinational; // keeps detect synchronous
49            end
50        end
51    endmodule
```

Verilog Example

```
input  wire clk,
input  wire rst_n,
input  wire data_in,
output reg  detect
```

Mealy Machine

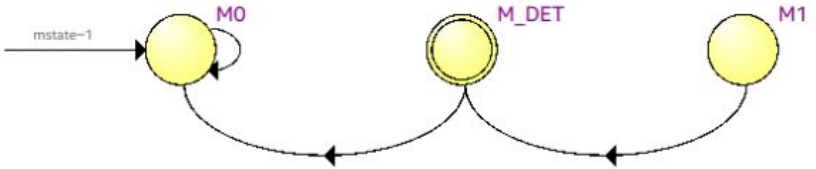
```
29        always @* begin
30            // Default values
31            next_state  = state;
32            detect_next = 1'b0;
33
34            case (state)
35                S0: begin
36                    // S0: idle/waiting for a 0 to start the 0->1 detection
37                    if (data_in == 1'b0) begin
38                        next_state = S1; // saw 0 -> transition to S1 to wait for 1
39                    end else begin
40                        next_state = S0; // stay in S0 if input is 1
41                    end
42                end
43
44                S1: begin
45                    // S1: previously saw a 0; a 1 now produces a detection pulse
46                    if (data_in == 1'b1) begin
47                        detect_next = 1'b1; // Mealy immediate output (one-cycle pulse)
48                        next_state  = S0;   // go back to S0 after detection
49                    end else begin
50                        next_state  = S1;   // remain in S1 while zeros persist
51                    end
52                end
53
54                default: begin
55                    // Safe recovery for X/unknown states during simulation
56                    next_state  = S0;
57                    detect_next = 1'b0;
58                end
59            endcase
60        end
67        always @(posedge clk or negedge rst_n) begin
68            if (!rst_n) begin
69                state  <= S0;       // reset to idle state
70                detect <= 1'b0;     // clear output on reset
71            end else begin
72                state  <= next_state;
73                detect <= detect_next;
74            end
75        end
```

Verilog Example

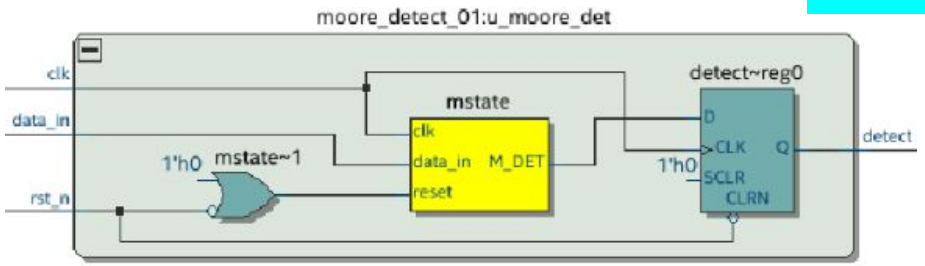
Moore Machine

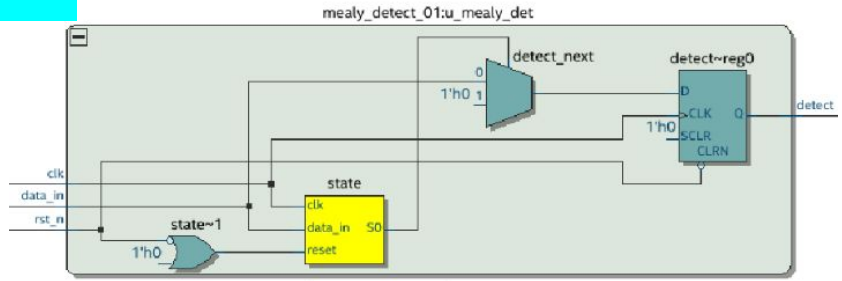
FSM

Mealy Machine



Netlist





Verilog Analysis

Moore Machine

- **Registers (flip-flops)**: 3
  - 2 bits for the state register (**mstate[1:0]**)
  - 1 bit for the registered output (**detect**).
- **Combinational logic (LUTs / logic elements)**: typically ~3 LUTs because the **always @(*)** block:
  - computes **next_state** (2 bits) from **mstate** and **data_in**, and
  - computes **detect_combinational = (mstate == M_DET)**.

Flow Status	Successful - Sun Nov 23 16:32:47 2025
Quartus Prime Version	24.1std.0 Build 1077 03/04/2025 SC Lite Edition
Revision Name	top_level
Top-level Entity Name	moore_detect_01
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	3 / 114,480 (< 1 %)
Total registers	3
Total pins	4 / 529 (< 1 %)

Mealy Machine

- **Registers**
  - State register: 1 FF (optimized) or 2 FFs (literal) — holds current state.
  - Output register (**detect**): 1 FF
  - Total FFs: 2 (typical optimized) or 3 (literal).
- **Combinational logic**
  - Next-state logic: implements **if data_in==0 then S1 else S0** from S0 and **if data_in==1 then S0 else S1** from S1. This boolean function maps to 1 LUT if packed efficiently.
  - Detect logic: **detect_next = state==S1 && data_in==1** — one boolean expression; typically 1 LUT or merged into the next-state LUT.
  - Estimated LUT/LE count: 1–4 depending on synthesis optimizations.

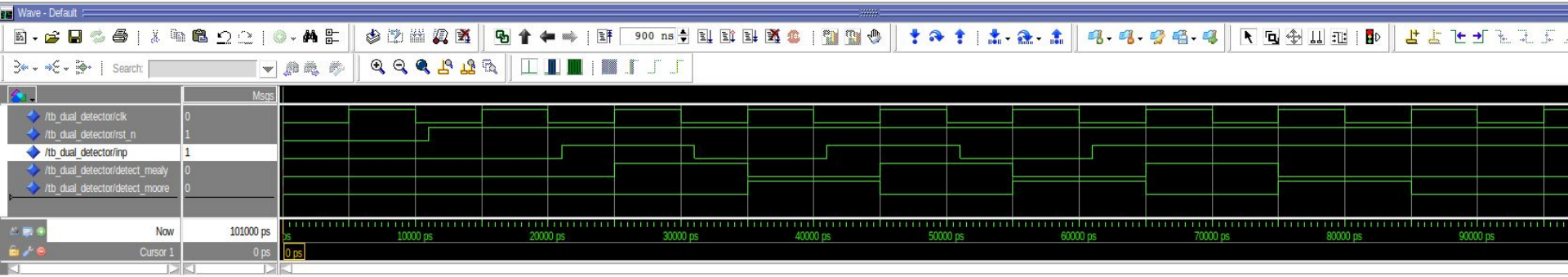
Flow Status	Successful - Sun Nov 23 16:34:48 2025
Quartus Prime Version	24.1std.0 Build 1077 03/04/2025 SC Lite Edition
Revision Name	top_level
Top-level Entity Name	mealy_detect_01
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	2 / 114,480 (< 1 %)
Total registers	2
Total pins	4 / 529 (< 1 %)

FSM: Moore VS Mealy

Verilog Simulation

ModelSim

Intel® Quartus® Prime



Discussion:

The Moore machine output (**detect_moore**) are synchronous with clock. It changes only with state transition at clock edge. The Mealy machine output (**detect_mealy**) are asynchronous. They can change immediately with input (**inp**) change, independent of the clock. So we can say Moore machine is not as "fast" as Mealy.

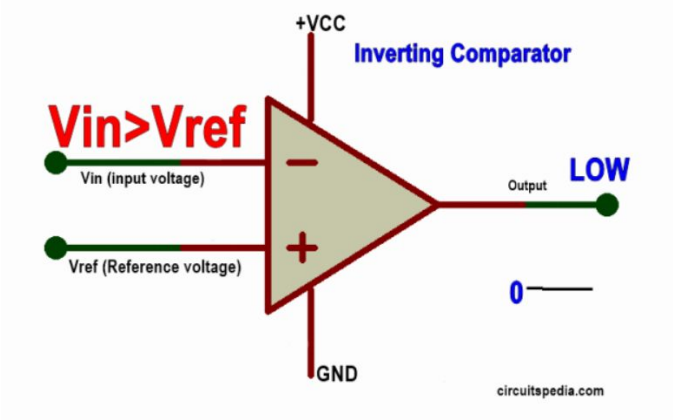
③

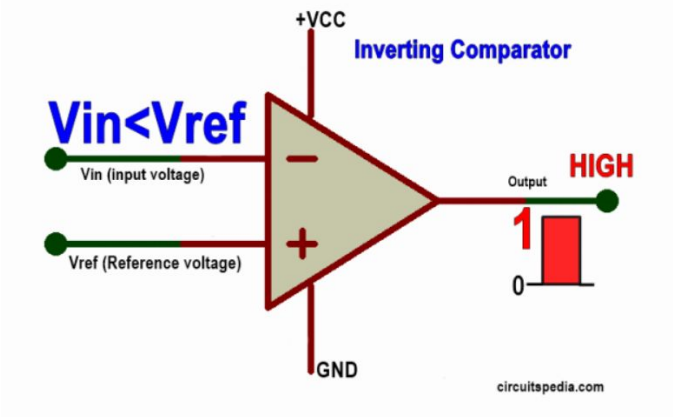
Comparator

- Op-amp Comparator
- 1-bit Comparator
- 2-bit Comparator
- 4-bit Comparator

Op-amp Comparator

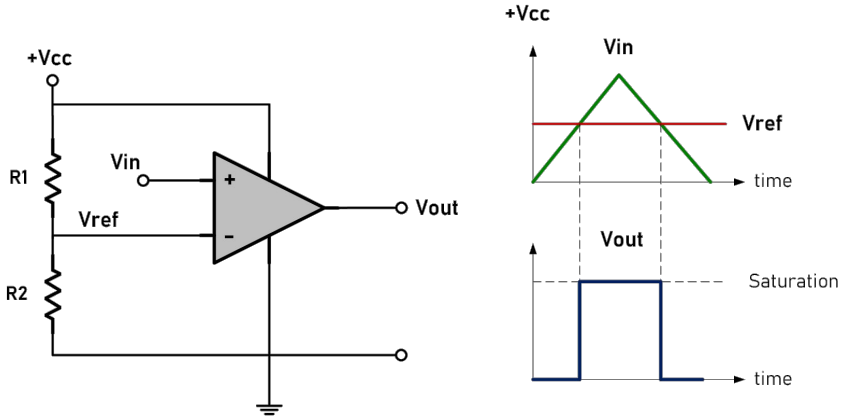
- Op-amp comparators are circuits in an open-loop configuration to compare two input voltages.
- The op-amp has two inputs:
  - **Non-inverting terminal (+)**: Receives the signal (Vin) that will be compared.
  - **Inverting terminal (-)**: Typically set to a reference voltage (Vref) to establish the threshold.





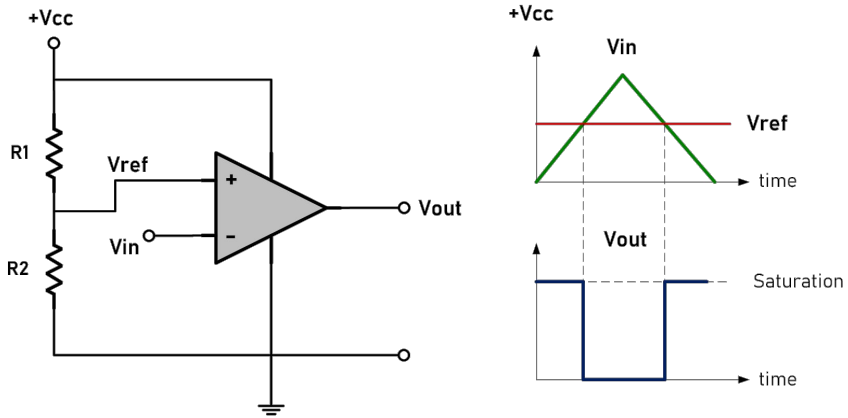
Positive Comparator

(Non-inverting Op-amp Comparator)



Negative Comparator

(Inverting Op-amp Comparator)



Source:
https://anitocircuits.com/op-amp-comparator-configurations-and-applications-in-circuits/

Digital Comparator

- A **digital comparator** is to compare a set of variables, for example A (A1, A2, A3, …. An, etc) against that of a constant value such as B (B1, B2, B3, …. Bn, etc) and produce an output flag depending upon the result of the comparison. There are two main types of Digital Comparator:
  - An **Identity Comparator** is a digital comparator with only one output terminal for when A = B, either A = B = 1 (HIGH) or A = B = 0 (LOW)
  - A **Magnitude Comparator** is a digital comparator which has three output terminals, one each for equality, A = B greater than, A > B and less than A < B

1-bit Comparator

- The result of “A = B” can be computed by XNOR

$$\overline{A \oplus B}$$

Where $\oplus$ represents XOR

- The result of A < B can be computed by

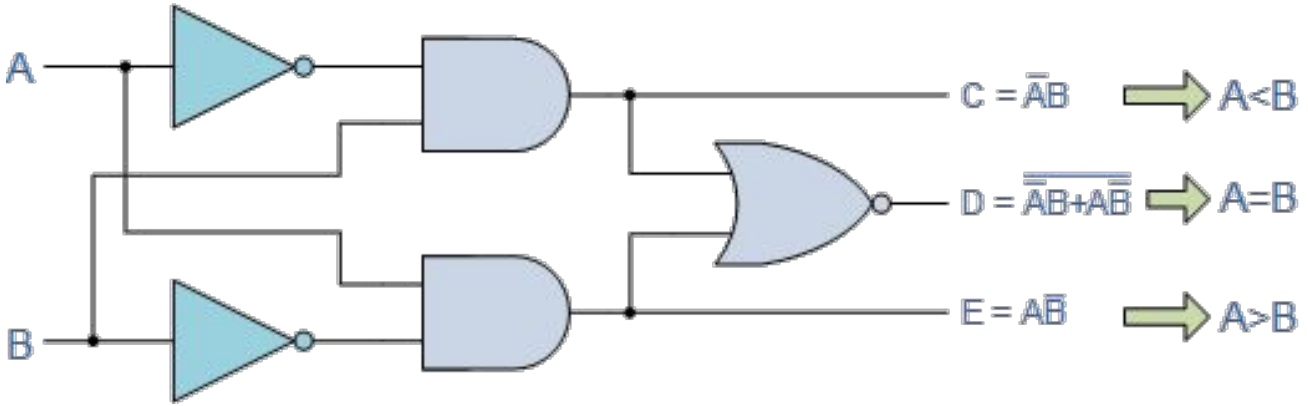
$$\overline{A} \wedge B$$

- The result of A > B can be computed by

$$A \wedge \overline{B}$$

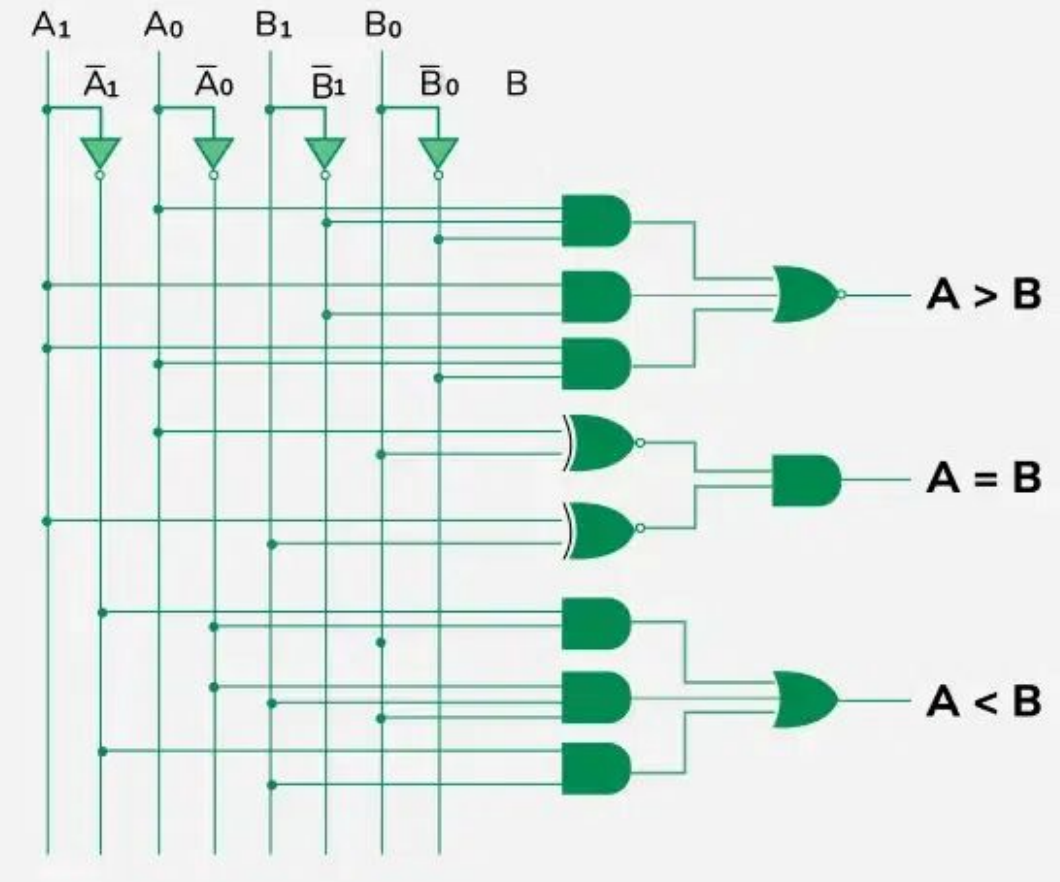
1-bit Comparator Circuit

Inputs		Outputs		
B	A	A > B	A = B	A < B
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0



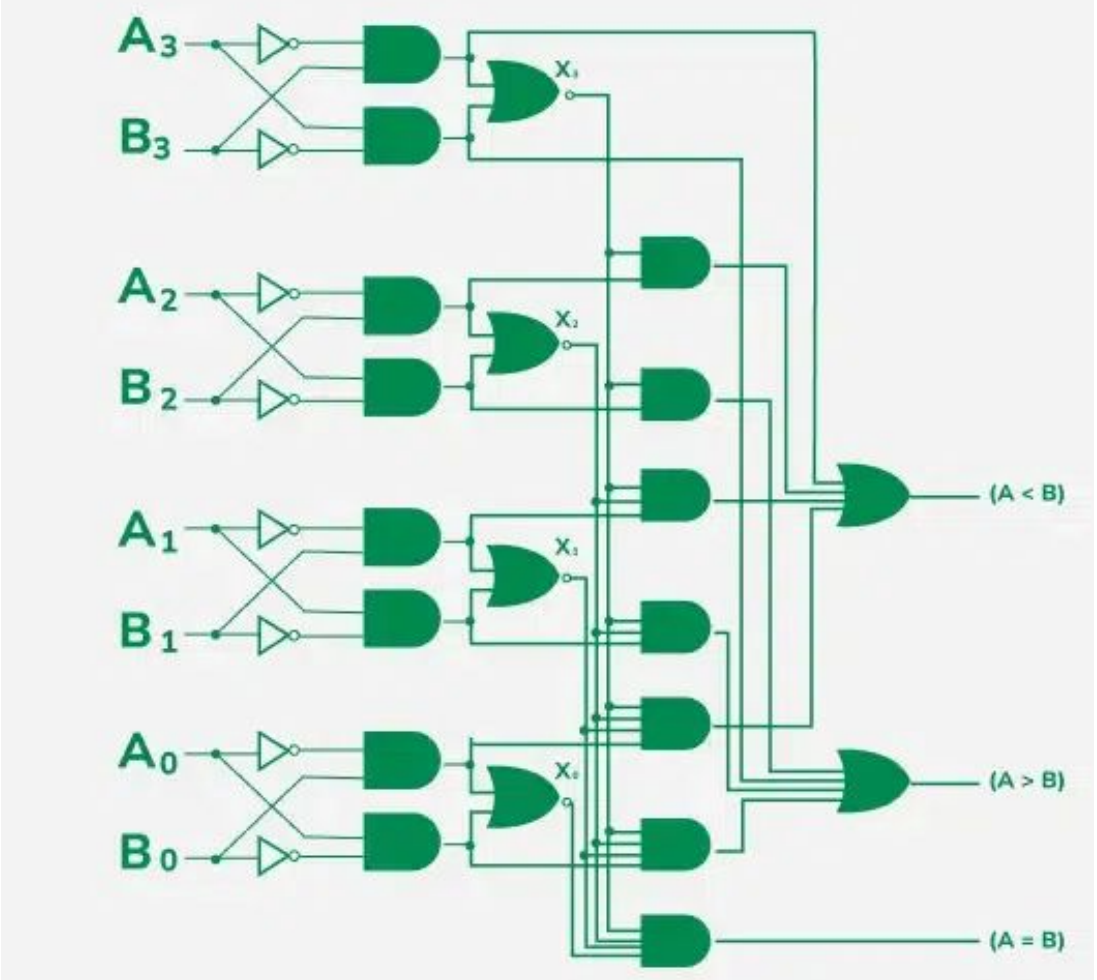
Source:
https://anitocircuits.com/op-amp-comparator-configurations-and-applications-in-circuits/

2-bit Comparator



Source:
https://www.geeksforgeeks.org/digital-logic/magnitude-comparator-in-digital-logic/

4-bit Comparator



Source:
https://www.geeksforgeeks.org/digital-logic/magnitude-comparator-in-digital-logic/

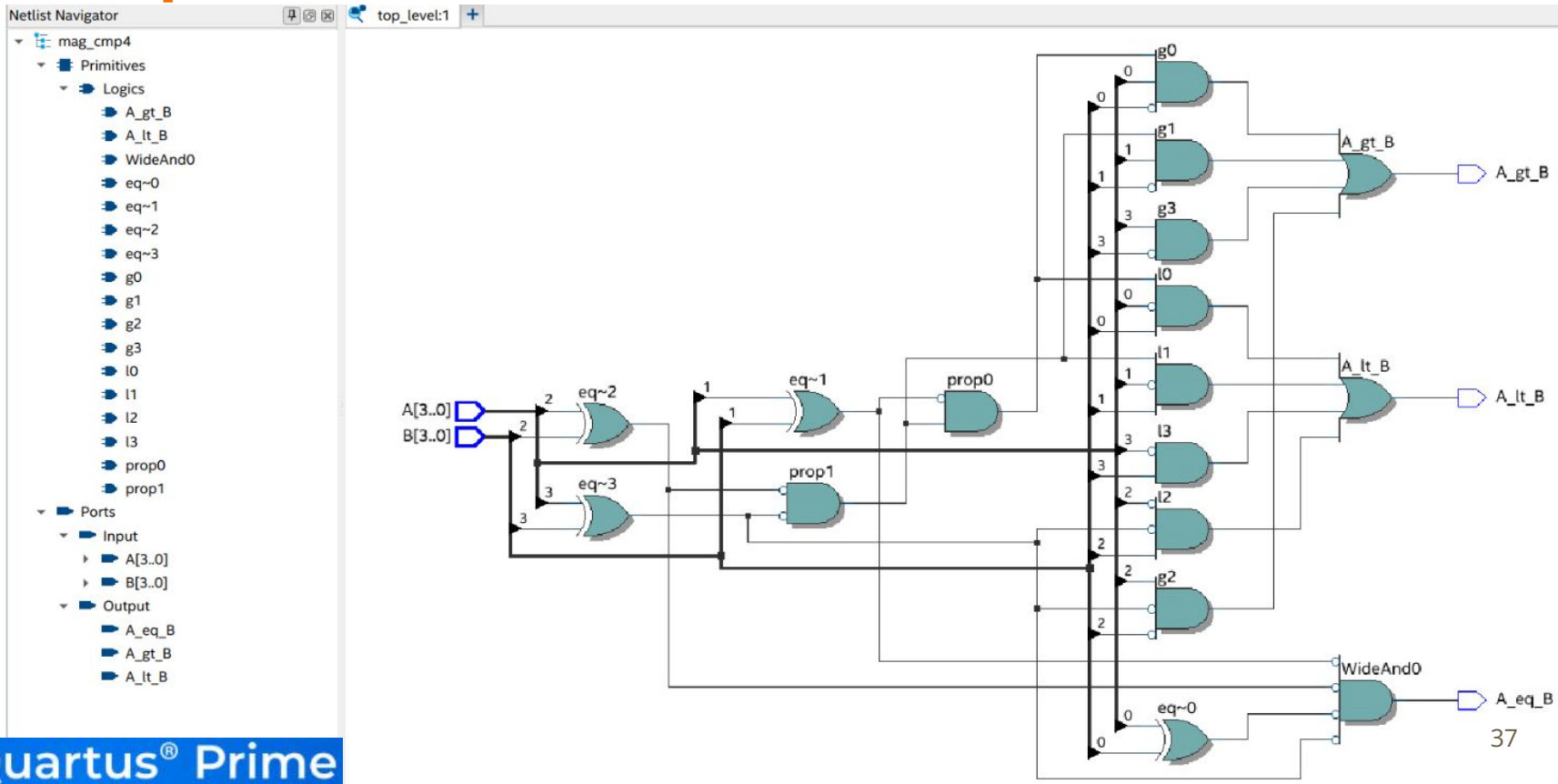
4-bit Comparator

Verilog Example

```
 8        input  [3:0] A,
 9        input  [3:0] B,
10        output       A_gt_B,
11        output       A_lt_B,
12        output       A_eq_B
```

```
15        // Per-bit equality: eq[i] == 1 when A[i] equals B[i]
16        wire [3:0] eq;
17        assign eq = ~(A ^ B);
18
19        // A_eq_B is true when all per-bit equalities are true
20        assign A_eq_B = &eq; // reduction AND
21
22        // Propagate signals: prefix AND of higher significant eq bits
23        // prop[i] == 1 means all bits more significant than i are equal
24        wire prop3 = 1'b1;                    // no higher bits above MSB
25        wire prop2 = eq[3];
26        wire prop1 = eq[3] & eq[2];
27        wire prop0 = eq[3] & eq[2] & eq[1];
28
29        // Generate signals: g[i] is true if bit i determines A > B
30        wire g3 = A[3] & ~B[3];
31        wire g2 = A[2] & ~B[2] & prop2;
32        wire g1 = A[1] & ~B[1] & prop1;
33        wire g0 = A[0] & ~B[0] & prop0;
34        assign A_gt_B = g3 | g2 | g1 | g0;
35
36        // Similarly for A < B (l[i] true if bit i determines A < B)
37        wire l3 = ~A[3] & B[3];
38        wire l2 = ~A[2] & B[2] & prop2;
39        wire l1 = ~A[1] & B[1] & prop1;
40        wire l0 = ~A[0] & B[0] & prop0;
41        assign A_lt_B = l3 | l2 | l1 | l0;
```

4-bit Comparator: Netlist

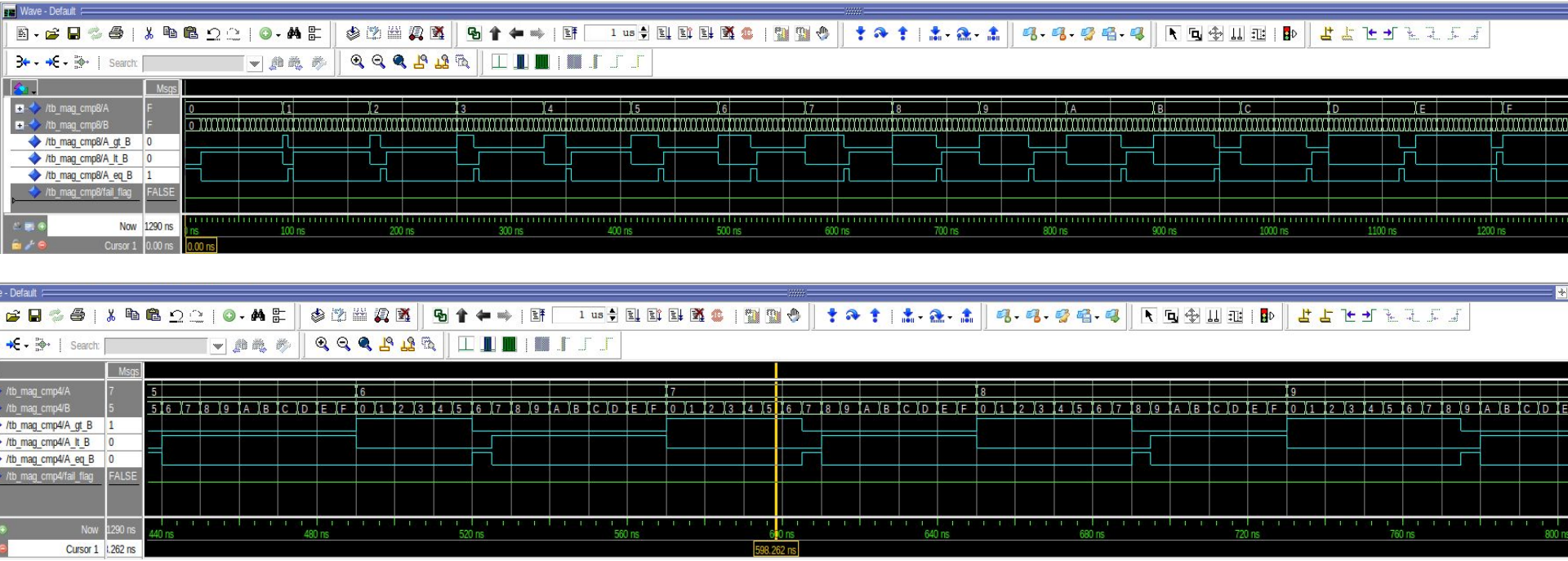


4-bit Comparator

Verilog Simulation

ModelSim

Intel® Quartus® Prime



↑ Example at time = 598 ns:
Assume **A** = 7 and **B** = 5.
Because **A** > **B** , the output **A_gt_B** becomes one!

④

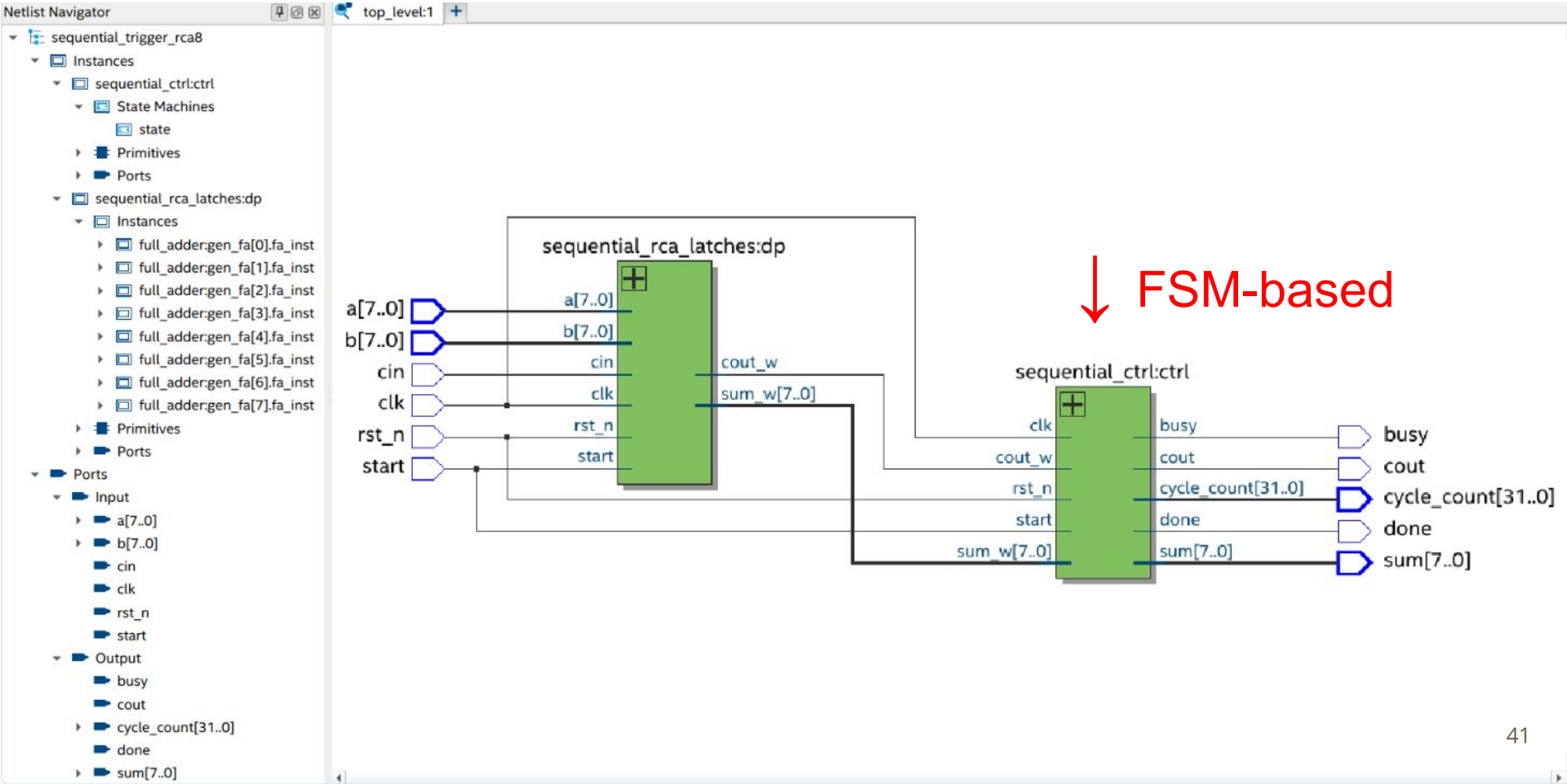
Simulation

- ➔ FSM-based Adder
  - RCA
  - CLA
- ➔ 8-bit Comparator
- ➔ ModelSim

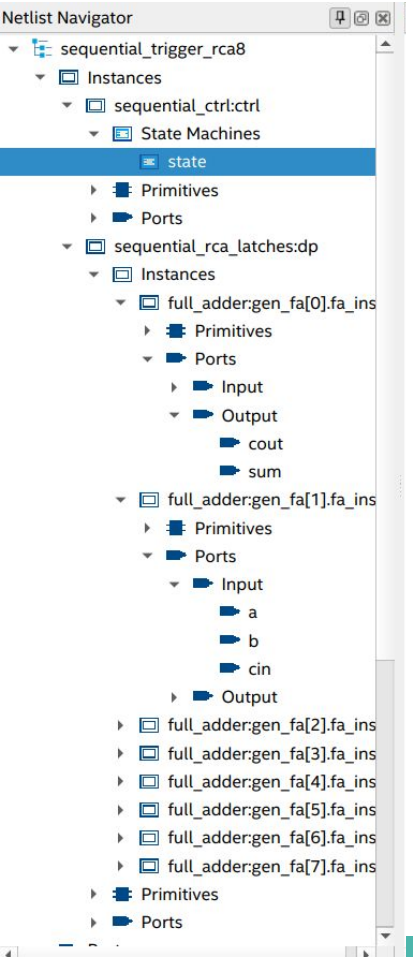
States & State Transition in FSM (Mealy)

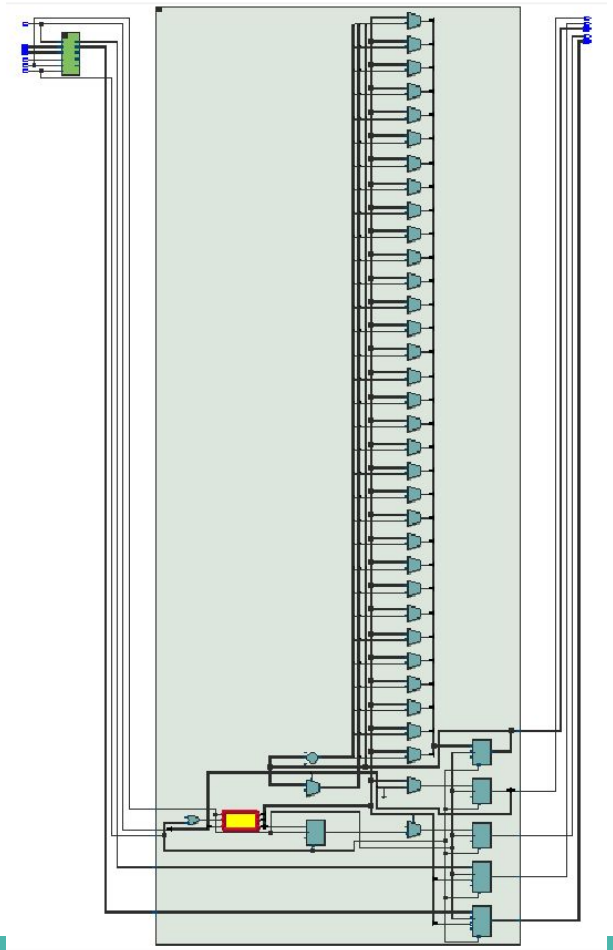
State	Meaning	Entry condition	Key outputs / registers driven	Next state
STATE_IDLE	Waiting for a start	`start == 1` while idle	`busy = 0`; on start: latch `a,b,cin`; `carry <= cin`; `bit_idx <= 0`; `busy <= 1`	`STATE_RUN`
STATE_RUN	Process one bit per clock	entered from IDLE or self (iterating)	each cycle: `sum[bit_idx] <= bit_sum`; `carry <= bit_cout`; `cycle_count++`; if last bit: `cout <= bit_cout`; `busy <= 0`; `done <= 1`	`STATE_DONE` (if last bit) or remain to process next bit (`bit_idx+1`)
STATE_DONE	One-cycle completion pulse	entered after last bit in RUN	`done` is cleared (ensures one-cycle pulse); `state <= STATE_IDLE`	`STATE_IDLE`

FSM-based 8-bit Adder

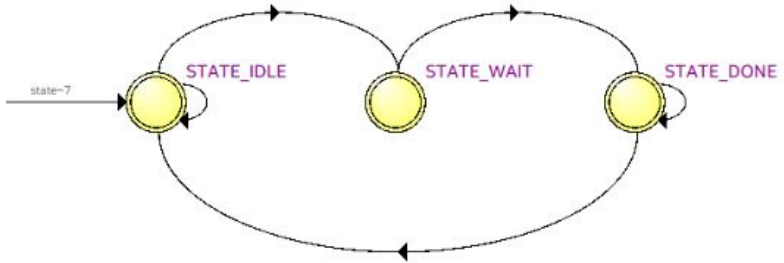


FSM-based 8-bit Adder





Mealy Machine



State Table

	Name	STATE_IDLE	STATE_DONE	STATE_WAIT
1	STATE_IDLE	0	0	0
2	STATE_WAIT	1	0	1
3	STATE_DONE	1	1	0

Transitions | Encoding

State Table

	Source State	Destination State	Condition
1	STATE_DONE	STATE_IDLE	(!start)
2	STATE_DONE	STATE_DONE	(start)
3	STATE_IDLE	STATE_IDLE	(!start)
4	STATE_IDLE	STATE_WAIT	(start)
5	STATE_WAIT	STATE_DONE	

Transitions | Encoding

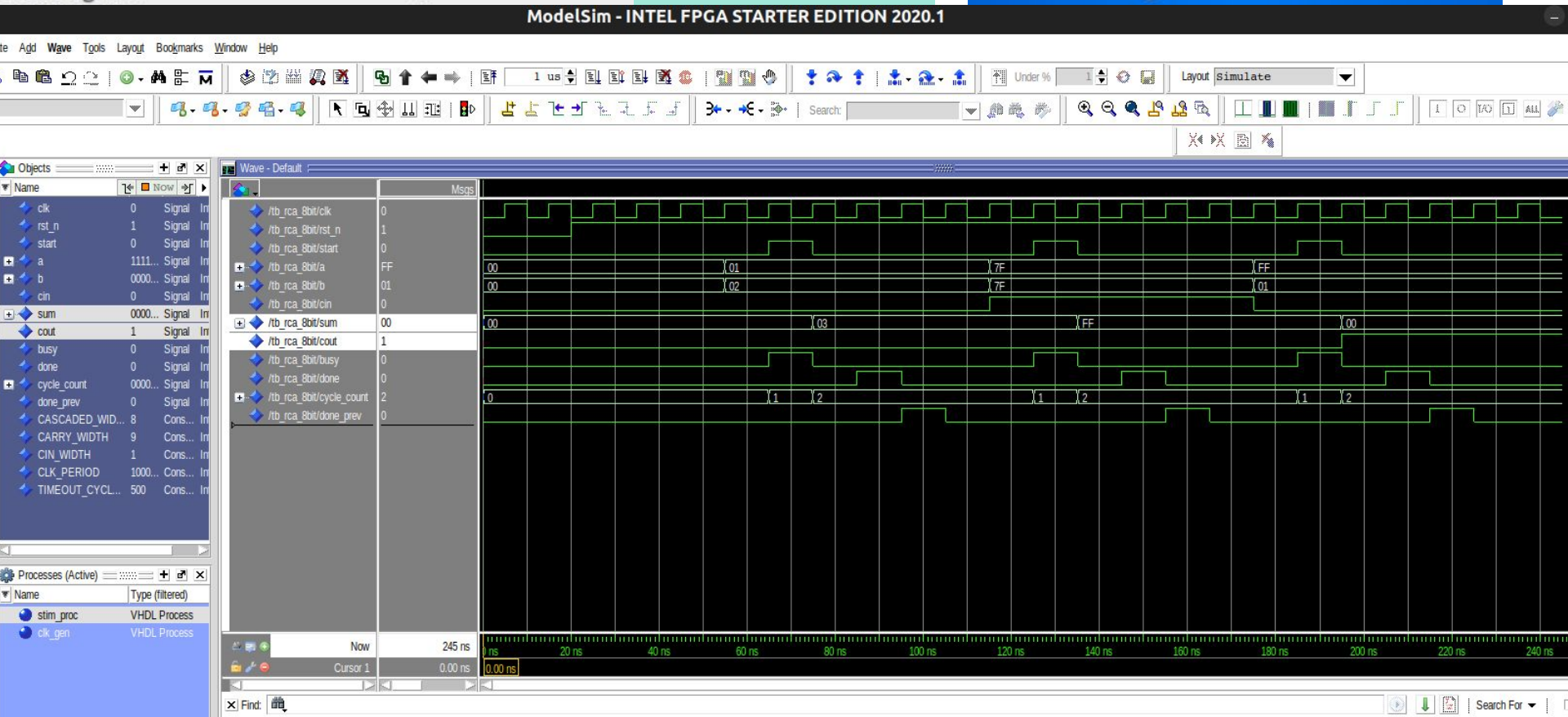
FSM-based 8-bit RCA

Verilog Example

ModelSim

Intel® Quartus® Prime

Total logic elements	67 / 114,480 (< 1 %)
Total registers	64



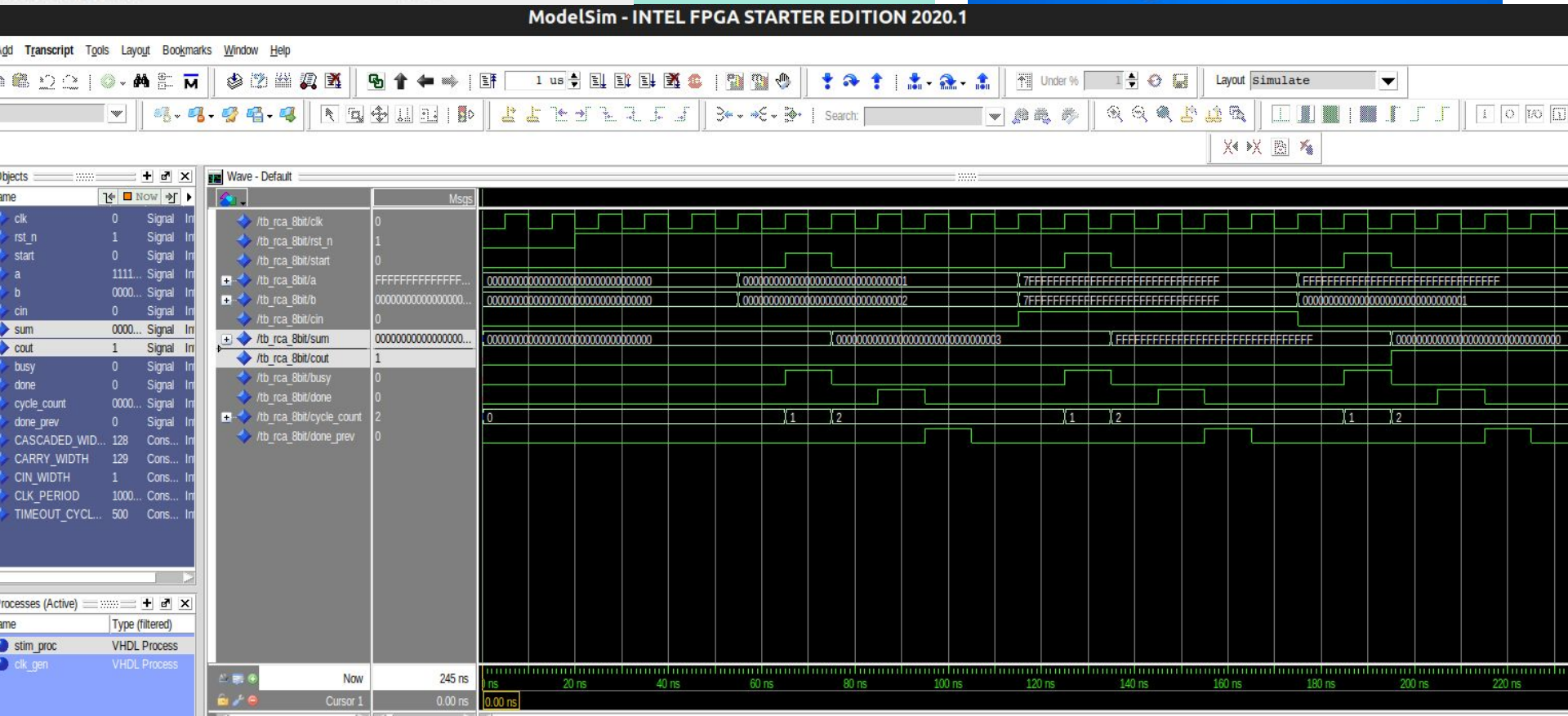
FSM-based 128-bit RCA

Verilog Example

ModelSim

Intel® Quartus® Prime

Total logic elements	463 / 114,480 (< 1 %)
Total registers	424



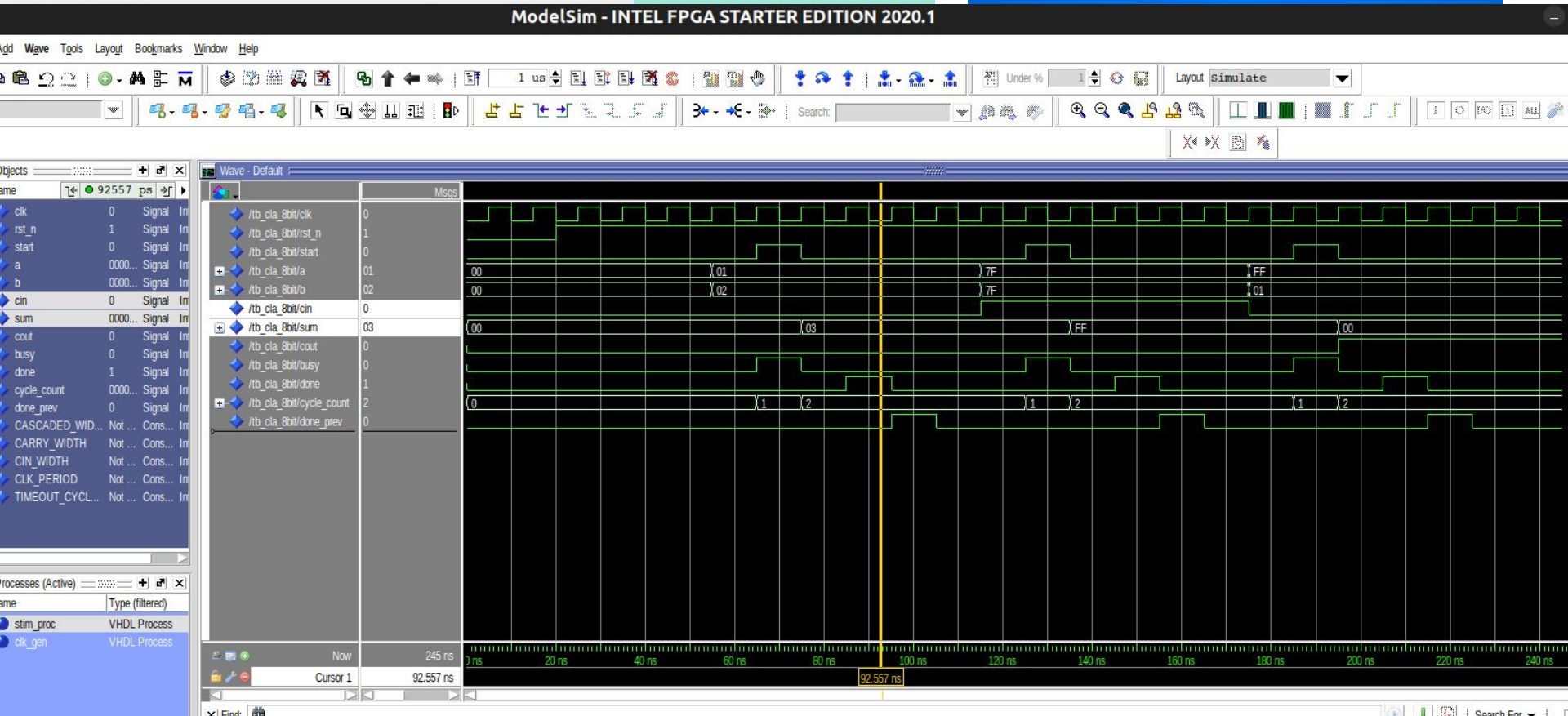
FSM-based 8-bit CLA

Verilog Example

ModelSim

Intel® Quartus® Prime

Total logic elements	84 / 114,480 (< 1 %)
Total registers	64



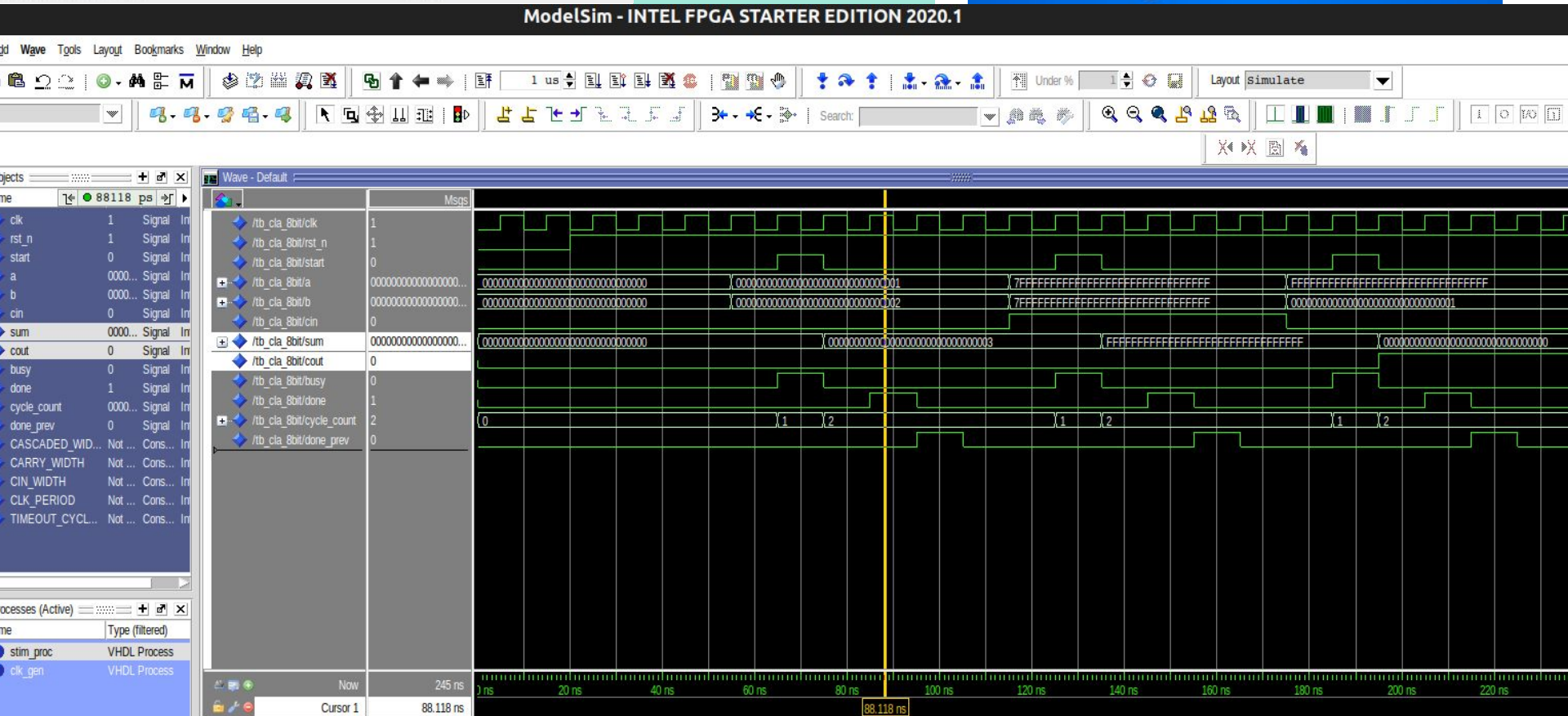
FSM-based 128-bit CLA

Verilog Example

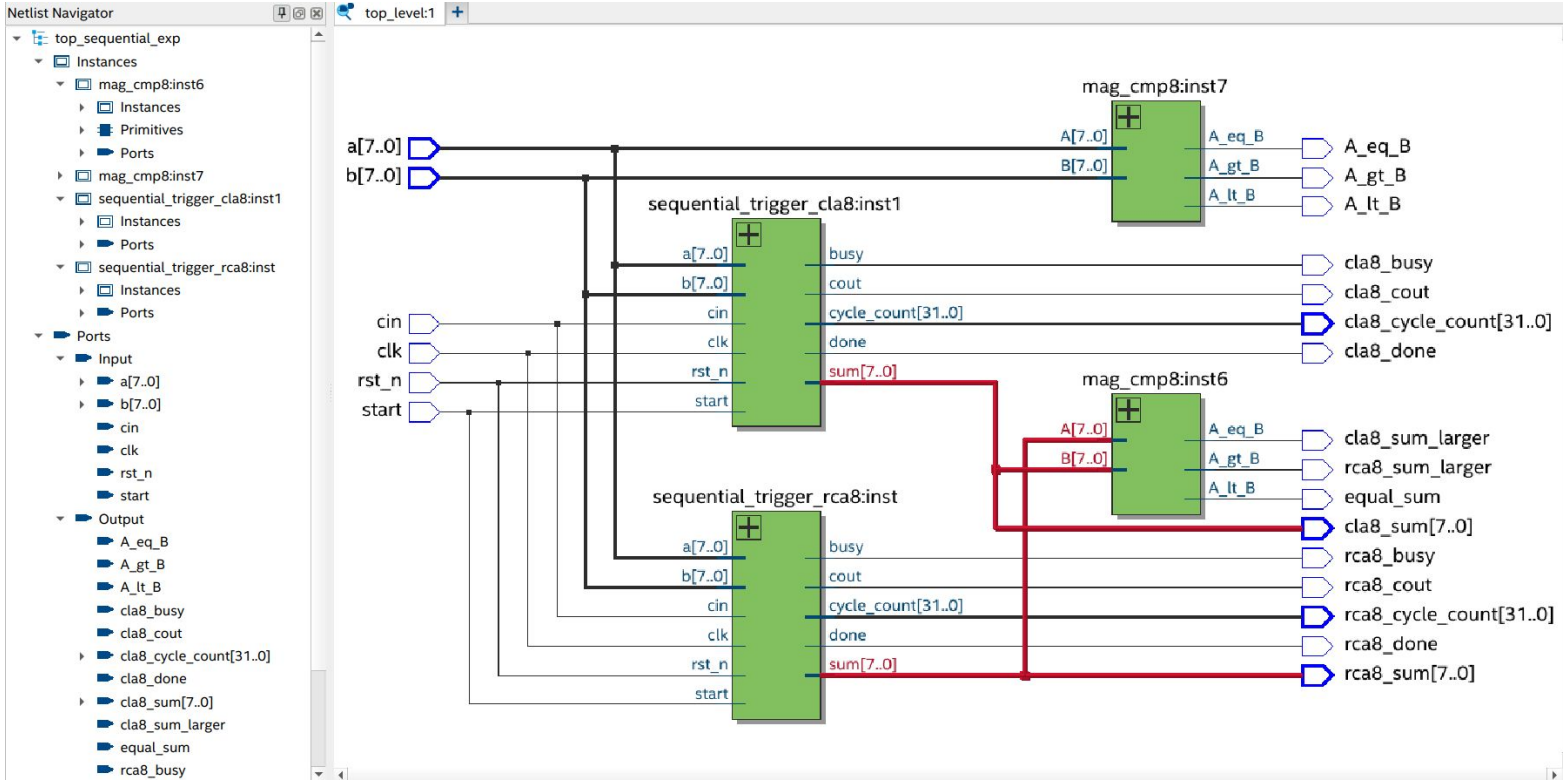
ModelSim

Intel® Quartus® Prime

Total logic elements	4,935 / 114,480 (4 %)
Total registers	424



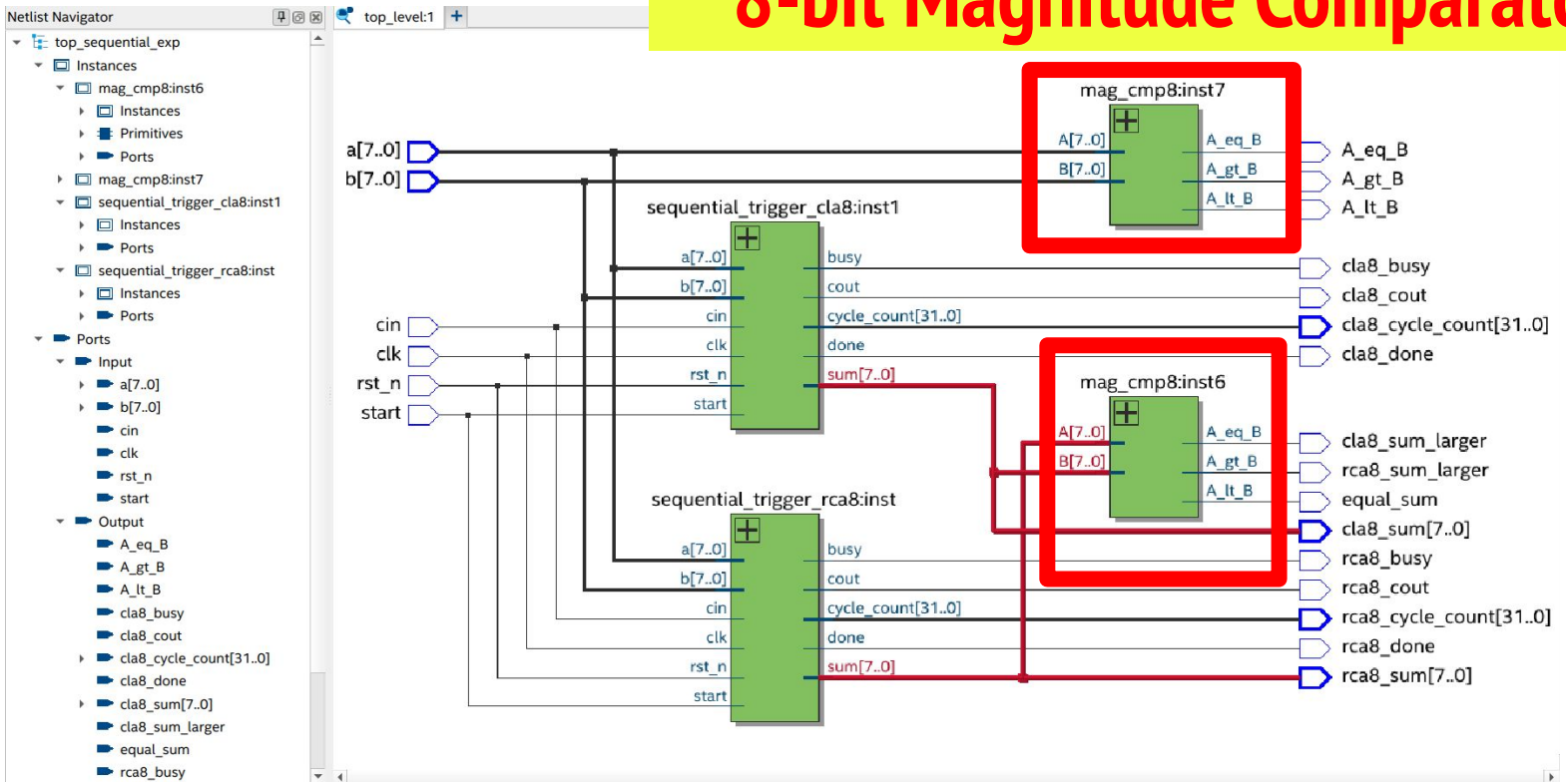
Project: Comparison of FSM-based RCA/CLA



Intel® Quartus® Prime ModelSim®

Project: Comparison of FSM-based RCA/CLA

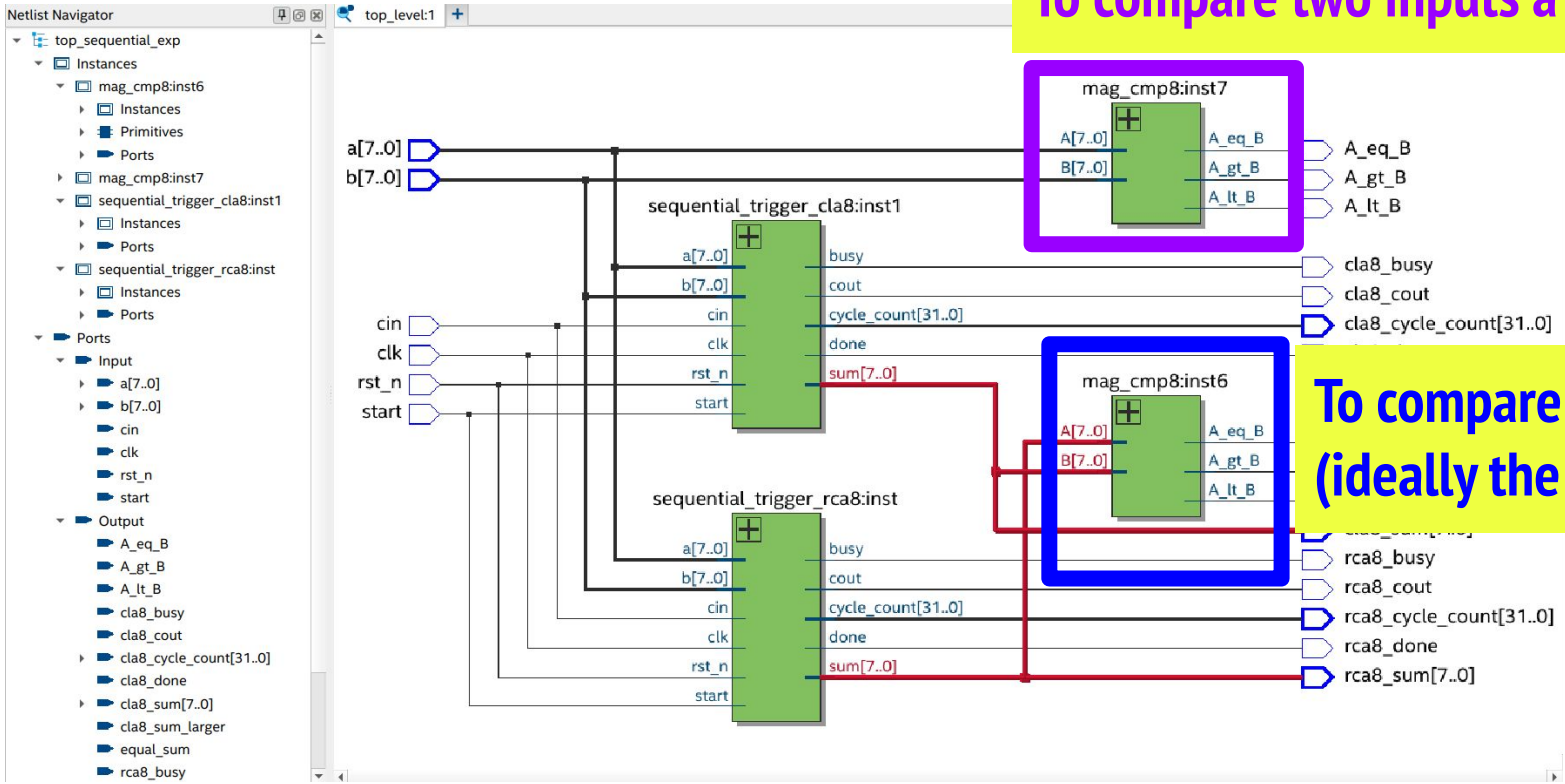
8-bit Magnitude Comparator



Intel® Quartus® Prime ModelSim®

Project: Comparison of FSM-based RCA/CLA

To compare two inputs a and b



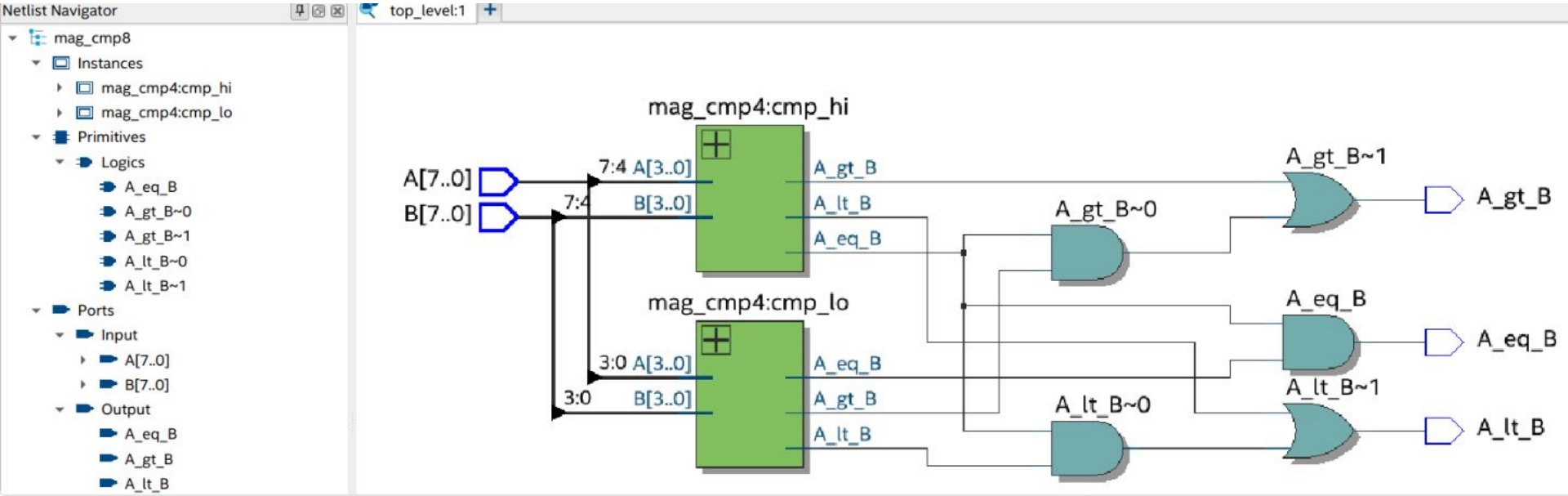
To compare sum (ideally the same)

Intel® Quartus® Prime ModelSim®

Project: Comparison of FSM-based RCA/CLA

How to implement an 8-bit comparator?

<solution> **Then combine two 4-bit comparators.**

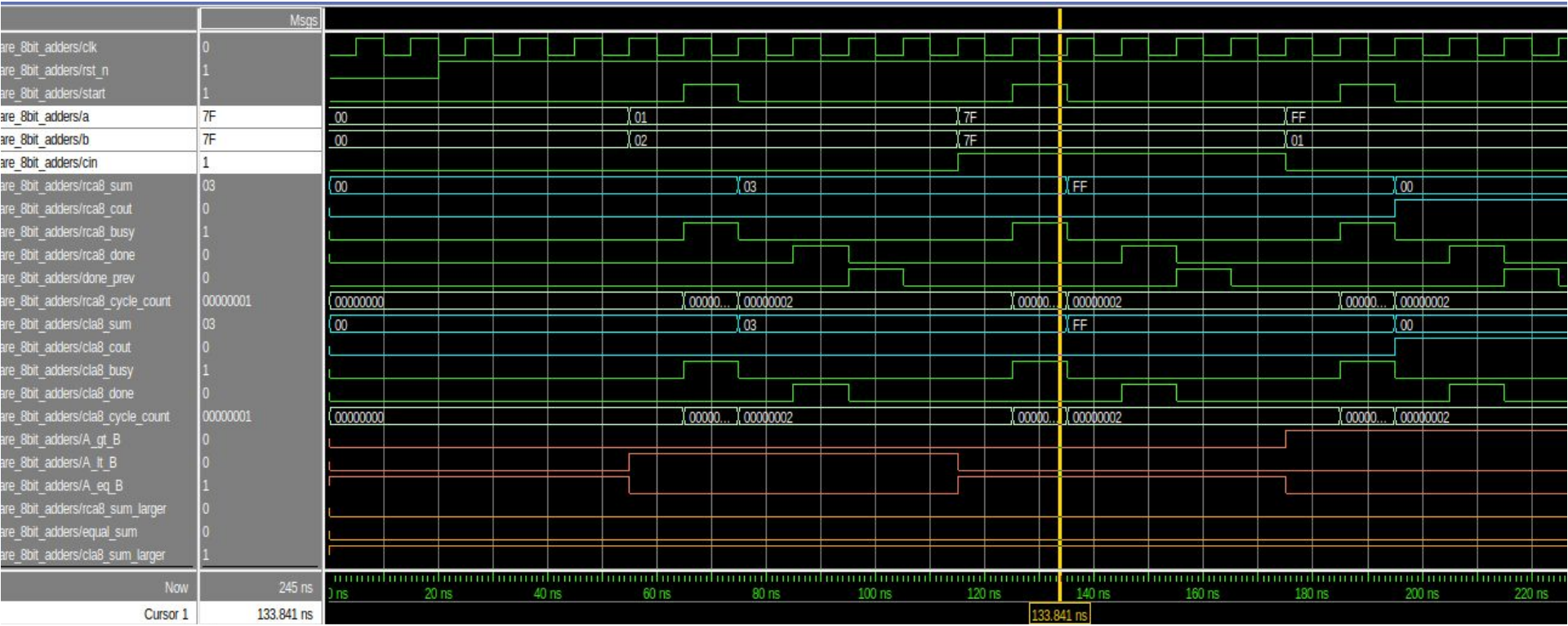


FSM-based RCA/CLA

Verilog Project

ModelSim

Intel® Quartus® Prime



Comparison of FSM-based RCA/CLA

Conclusions

- ripple-carry adder (RCA)
- carry-lookahead adder (CLA)
- 8-bit magnitude comparator
- ModelSim

Conclusions

- Either 8-bit/128-bit RCA or CLA computes the sum and cout correctly.
- More logic elements (LEs) and registers are used in CLA than those in RCA.
- More LEs and registers are used when the adder requires more bits.
- The simulation outputs from 8-bit magnitude comparators match the results in the truth table.

Discussion (and Future Work)

- Only 1 clock cycle is taken to compute the sum with either one of the algorithms (8-bit RCA, 128-bit RCA, 8-bit CLA, and 128-bit CLA).
- No differences of propagation delay can be found in this report.
  - ModelSim results may not match the real-world phenomenon.
  - The sampling mechanism of the clock in the Mealy FSM could be modified when an adder output is computed.

If you have any feedback, please contact me at

twwang97@gmail.com

twwang97

Electronics and AI Training Program (3rd session in 2025)

Class of Digital Design with FPGA

Thanks for your attention

NYCU
國立陽明交通大學

陽明交大雷射系統研究中心