

i need for 8 bit using matrix method karnaugh's graph
create combinational circuits like addition and subtraction
multiplication in logisim give table form procedures

நல்லது — உன்னிடம் 8-பிட் காம்பினைட்டோரியல் சர்க்கியடுகள் (Addition, Subtraction, Multiplication) Logisim-ல் உருவாக்குவதற்கு தேவையான கட்டப்படுத்தப்பட்ட செட் ஏதும் — நான் இங்கே தெளிவான கட்டளைபடி (table-form) செயல்முறை, தேவையான உண்மை அட்டவணைகள், Karnaugh/Matrix யைக் எப்படி பயன்படுத்துவது, மற்றும் Logisim-ல் எவ்வாறு தொகுக்க வேண்டும் என்பதைக் கொடுக்கிறேன். உன் முன்னண்ணத்தில் தமிழ் விருப்பம் இருப்பதால் முழு விளக்கம் தமிழில் தான்.

1) அடிப்படை: Half-adder மற்றும் Full-adder

முதலில் 1-பிட் அடிப்படை பிளாக்களை சரியாக உருவாக்க வேண்டும். அதிலிருந்து 8-பிட் அமைப்பு உருவாகிறது.

Half-adder

உண்மை அட்டவணை (Truth table)

A B | Sum S | Carry C

0 0 | 0 | 0

0 1 | 1 | 0

1 0 | 1 | 0

1 1 | 0 | 1

பொருத்தமான கூறுகள்

- $S = A \oplus B$
- $C = A \cdot B$

Full-adder (A, B, Cin -> Sum, Cout)

உண்மை அட்டவணை (8 வரிசைகள்)

A B Cin | Sum S | Cout

0 0 0 | 0 | 0

0 0 1 | 1 | 0

0 1 0 | 1 | 0

0 1 1 | 0 | 1

```

1 0 0 | 1 | 0
1 0 1 | 0 | 1
1 1 0 | 0 | 1
1 1 1 | 1 | 1

```

Boolean செயல்பாடுகள் (சுருக்கமாக)

- $S = A \oplus B \oplus \text{Cin}$
- $\text{Cout} = (A \cdot B) + (B \cdot \text{Cin}) + (A \cdot \text{Cin})$

Karnaugh Map பயன்படுத்துவது (குறிப்பு)

- Sum மற்றும் Cout-க்கு K-maps இல் குழுக்களைப் பொருத்தி குறைக்கலாம். மேலுள்ள Cout உரையைக் K-map-இல் பார்க்கும்போது மூன்றில் இரண்டு பொருட்கள் மேட்ச் ஆக grouping மூலம் ஒரு சுலபமான வடிவம் கிடைக்கும். (ஆய்தமாக Sum கடத்தல் XOR மூலம் எளிமையாகப் பெறப்படுகிறது.)

2) 1-பிட் → 8-பிட் விரிவாக்கம் (Ripple-carry adder)

செயல்முறை (table-form steps):

1. **Bit-slice module:** ஒரு Full-adder ($A_i, B_i, \text{Cin}_i \rightarrow S_i, \text{Cout}_i$) 1-பிட் module உருவாக்கு. (Logisim-ல் subcircuit ஆக சேர்த்துக்கொள்ளு.)
2. **Chain:** $i = 0..7$ வரை 8 replica வைக்கு.
 - $\text{Cin}_0 = 0$ (நீ தீர்மானித்தால்) அல்லது இயல்பாகக் கூட்டீட்டில் இருந்து வரும் carry.
 - $\text{Cout}_i \rightarrow \text{Cin}_{i+1}$
3. **Buses:** 8-பிட் bus $A[7..0]$, $B[7..0]$ உருவாக்கு; ஒவ்வொரு bit-ஐ தனியே Full-adder-க்கு இணைப்பு.
4. **Out:** $S[7..0]$ என்பது 8-பிட் sum; final carry = Cout_7 (overflow flag).

Logisim Implementation checklist

- Tunnel அல்லது Splitter பயன்படுத்தி 8-bit bus manage செய்.
- ஒவ்வொரு bit-slice ஐ chain செய்யும் போது clock தேவையில்லை (combinational).
- Test vectors: $00000001 + 00000001 \rightarrow 00000010$; $11111111 + 00000001 \rightarrow 00000000$ with carry=1 (overflow).

3) 8-bit Subtraction — Two's complement method (recommended)

அடிப்படை யோசனை: $A - B = A + (\text{two's complement of } B)$

two's complement தயார் செய்யும் நடைமுறை (8-bit)

1. Bயின் bit ஒவ்வொன்றையும் **invert** செய் $\rightarrow \bar{B}$
2. அதற்குப் பிறகு **add 1** (i.e., set initial $Cin_0 = 1$)
3. $A + \bar{B} + 1 \rightarrow \text{result} = A - B$ (8-bit), final carry ignored for signed, used as borrow indicator for unsigned.

Logisim வழிமுறை (table-form)

- Subtractor circuit = 8-bit adder where B bus goes through 8 XOR gates with control = SUB (SUB=1 \rightarrow invert B).
- Initial $Cin_0 = SUB$ (0 for add, 1 for subtract).
- If SUB=1 \rightarrow B inverted and $Cin=1$ (effectively $A + (\sim B) + 1$).
- Output $S = A \pm B$, $Cout_7$ indicates unsigned overflow/borrow.

Test cases

$A=00001010$ (10), $B=00000101$ (5), SUB=1 \rightarrow result 00000101 (5).

$A=00000000$, $B=00000001$, SUB=1 \rightarrow result 11111111 (-1 in two's complement), $Cout$ indicates borrow for unsigned.

4) 8-bit Multiplication — Combinational array multiplier (8×8 \rightarrow 16-bit)

Matrix concept (partial product matrix)

உலகில் பொதுவாக பயன்படுத்தும் எளிய வழி: **Array multiplier:**

- ஒவ்வொரு $A_i \times B_j$ (AND gate) \rightarrow partial product bit at row i , column j .
- இந்த partial products-ஐ சாளரங்கள் போல (matrix) இடம் பெறுத்து, ஒவ்வொரு column-இல் உள்ள பிட்டுகளை எண்ணிக்கையால் (shifted) சேர்க்க வேண்டும்.
- முடிவாக 16-bit product $P[15..0]$ கிடைக்கும்.

Partial product matrix (உதாரணம் **4-bit**-க்கு காண்பிக்கிறேன், **8-bit** இல் மாத்திரை விரிவாக்கம்தான்)

அதாவது:

	B7	B6	B5	B4	B3	B2	B1	B0
A7	a7b7	a7b6	a7b5	a7b4	a7b3	a7b2	a7b1	a7b0

```

A6 |   a6b7 a6b6 ...
...
A0 |   a0b7 ... a0b0

```

இந்த matrix-இல், column wise add செய்யும் (shifted) போது column 0 மட்டும் a0b0, column1: a0b1 + a1b0, column2: a0b2 + a1b1 + a2b0 ... இதுபோல.

Design steps (table-form)

- 1. Generate partial products:** For i in 0..7 and j in 0..7: $PP[i][j] = A[i] \text{ AND } B[j]$. (In Logisim: 64 AND gates)
- 2. Arrange by columns (weight):** $PP[i][j]$ contributes to final column $k = i+j$.
- 3. Column addition:** ஒவ்வொரு column k -இற்கும் உள்ள bits-ஐ add செய்ய 1-bit full adders / multi-operand adder பயன்படுத்துவோம். Typical approach: use ripple of adders column by column (Wallace tree or array):
 - Simple array multiplier: use adders row by row: each PP row is shifted and added to accumulated sum using 8-bit adders (or wider as needed). This is easy but resource-heavy.
 - Optimized: Wallace tree or carry-save adders to reduce levels (பல்தடவை குறையும்).
- 4. Result bits:** $P0 = PP[0][0]$; Final highest bits from last column up to P15.

Logisim practical steps (simple array)

- Row0: take $PP[0][j]$ produce an 8-bit partial (but shifted by 0) → add to accumulator (initially 0) using 16-bit adder (you can wire as 16).
- Row1: $PP[1][j]$ shifted left by 1 (i.e., multiply by 2) → add to accumulator.
- Repeat for rows 0..7.
- Use 16-bit adder (built by chaining eight 2-bit or one 16-bit ripple adder) to add each shifted partial product.
- Final accumulator = product (16-bit).

Resource count (approx for simple method)

- AND gates: 64
- 8 additions of 16-bit (but you can reuse one adder sequentially in sequential design; for combinational, you'll need a tree of adders)
- For purely combinational array: many full adders arranged column-wise (128+ full adders depending on implementation).

Test case

$A=00000011$ (3), $B=00000101$ (5) → product = 0000000000001111 (15).

5) Karnaugh map / Matrix method —

உபயோகப்படுத்தும் நடைமுறை (how to get gate minimization)

எந்த இடத்தில் **K-map** பயன்படுத்தலாம்?

- 1-bit Full-adder விவரத்திற்கு அல்லது carry-lookahead logic-ஐ எளிதாக்க Karnaugh பயன்படுத்தலாம்.
- Example: Full-adder Cout expression from truth table → K-map-இல் grouping செய்து simplified SOP பெறலாம். ஆனால் Sum எப்போதும் XOR தொடர்ச்சி; K-map-இன் மூலம் XOR ஐ NAND/NOR மூலம் உருவாக்கலாம்.

Carry-lookahead (Speed optimisation) — கண்ணோட்டம்

- ஒவ்வொரு bit-க்கு:
 - Generate: $G_i = A_i \cdot B_i$
 - Propagate: $P_i = A_i \oplus B_i$
- Carry equations:
 - $C_1 = G_0 + P_0 \cdot C_0$
 - $C_2 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0$
 - ... மேலும் C_n வைப்பது K-map/Boolean expansion மூலம் சுருக்க முடியும்.
- இந்த G/P கடந்தகளை பயன்படுத்தி carry computation-ஐ parallel ஆக பெறலாம் (carry-lookahead adder). K-map பயன்பாடு: சிறு bit-groups-க்கு carry function சுருக்க உபயோகப்படும்.

6) Logisim-ல் கட்டமைப்புக்கான practical checklist (step-by-step)

1. **Create subcircuits:** HalfAdder, FullAdder, RippleAdder8, Multi8x8_Array, InverterBus (for subtraction).
2. **Testbenches:** உருவாக்கிய subcircuit-க்கு test inputs (LEDs or probe) வைத்து சோதனை செய்.
3. **Use Splitters:** 8-bit bus-ஐ ஒவ்வொரு bit-க்கு பிரிக்க Splitter widget. (Logisim: splitter size $8 \rightarrow 1$ or vice versa).
4. **Labeling:** bus names A[7..0], B[7..0], SUB, CLK (not needed), OUT[15..0].
5. **Debugging:** Start with small bitwidth (e.g., 2-bit, 4-bit) verify outputs, பின்னர் 8-bit-க்கு உயர்த்து.

6. **Simulation:** apply edge cases: all ones, all zeros, max/min, overflow conditions.
7. **Optimization:** performance தேவை என்றால் Carry-Lookahead subcircuit-ஐ உருவாக்கு.

7) சுருக்கப்பட்ட உதாரணங்கள் (table snippets you can copy into Logisim docs)

Full-adder truth table (copyable)

A B Cin | Sum Cout

0 0 0 | 0 0

0 0 1 | 1 0

0 1 0 | 1 0

0 1 1 | 0 1

1 0 0 | 1 0

1 0 1 | 0 1

1 1 0 | 0 1

1 1 1 | 1 1

Carry-generate / propagate table (per bit)

A B | G (A·B) | P (A⊕B)

0 0 | 0 | 0

0 1 | 0 | 1

1 0 | 0 | 1

1 1 | 1 | 0

8) வெள்ளை அடுத்த படிகள் (practical templates)

- முதல்: Logisim-ல் HalfAdder, FullAdder subcircuits கட்டு; verify with truth table.
- இரண்டாம்: 8-bit ripple adder: 8 FullAdder chain; test.
- மூன்றாம்: Subtractor: use XOR-invert + initial carry trick. Test signed & unsigned.
- நான்காம்: Multiplier: implement small 4×4 array first; verify; then scale to 8×8. Use partial product row-addition approach.
- ஐந்தாம்: Optimize: carry-lookahead for adders or Wallace tree for multiplier if performance is important.

9) Quick tips & common pitfalls

- **Overflow vs Carry:** signed overflow detection differs from unsigned carry. Signed overflow = $\text{Cout}_7 \text{ XOR } \text{Cout}_6$ for two's complement adders. (Check carefully in tests.)
- **Bus endianness:** Logisim-ல் splitter bit ordering கவனிக்க — MSB vs LSB position.
- **Propagation delay:** ripple adder நேரம் அதிகம்; carry-lookahead குறைத்துக் கொடும்.
- **Multiplier size:** array multiplier resource-heavy; sequential shift-add multiplier uses fewer gates but needs clock (sequential design).

நீச் வேண்டுமானால்:

- நான் உனக்கு **Logisim project**-க்கான 1) Full-adder subcircuit ஆய்வுக் கோப்பு (step-by-step screenshots, connectors) 2) 8-bit ripple adder schematic 3) 8×8 array multiplier schematic — இதெல்லாம் கட்டியமைக்க ஊறாக ஒரு குறுகிய சராசரி README / step list எழுதலாம்.
- அல்லது இனி நான் நேரடியாக **4-bit** உதிரி (**worked**) **schematics** மற்றும் **K-maps**-ஐ எடுத்துச் சிக்கலான **boolean** சுருக்கங்களை வழங்க முடியும்.

படிகளைத் தொடங்க விரும்புகிறியா? நான் உடனே (இதே பதிலில்) 1-பிட் Full-adder K-map சுருக்கத்துடன் (ASCII K-map) மற்றும் 4×4 multiplier partial product matrix-ஐ காட்டி விடுவேன் — நீ சுட்டி சொல்லு.

in logisim tell me stepo by step howhic allin 8 bit adder matrix table and karnaugh to click

சரி — கீழே **Logisim**-ல் **8-bit adder** (ripple & matrix/Karnaugh உதவியுடன்) முழு click-by-click, table-format, படி-படி தெரிந்து கொடுப்பேன். விளக்கம் தமிழில், ஒவ்வொரு action-ஐ Logisim UI இல் எங்கு கிளிக் செய்ய வேண்டும் என்பதையும் சேர்த்திருக்கிறேன். நீ படி-படி செல்லவும் — அவசரம் இருந்தால் எதாவது skip செய்யாமல் தொடரு.

முன்னொட்டி (requirements)

- Logisim அல்லது Logisim-evolution திறந்து வைத்திரு.
- புதிய Project உருவாக்கியுள்ளேனில் நன்றாக இருக்கும்.
- பக்கவாட்டில் உள்ள பட்டைகள்: **Wiring, Gates, Plexers, Arithmetic, Input/Output, Wiring → Splitter**, முதலியன உண்டா என்பதைச் சரிபார்.

குறிப்பு: எது கட்டுவதற்கு முன்?

முதலில் 1-பிட் **Full Adder** subcircuit ஒன்றை உருவாக்குவோம் (சூழல்: இது உன் 8-bit adder-இன் அடிப்படை slice). அதற்குப் பிறகு அதனை 8 முறை நகலெடுத்து (copy/paste) chain செய்வோம். Karnaugh maps-ஐ Full-Adder உடன் தொடர்புடைய Cout மற்றும் Sum தன்மை சுருக்கம் பார்க்கப் பயன்படும்.

A. Full-Adder subcircuit — Click-by-click

1. Logisim → **Project** மெனுவில் → **Add Circuit...** → Name = FullAdder → OK.
2. வலது panel → **Input/Output** → Drag three Pin onto canvas. ஒவ்வொன்றையும் டாக் செய்யவும்:
 - முதன்மை Pin → பெயர் A (attributes → Label = A; Facing = West)
 - இரண்டாம் → B
 - மூன்றாம் → Cin
 (Pin size = 1-bit default)
3. **Output/Pin** இல் இருந்து two Pin (as output) இடுக: name them Sum and Cout . Facing = East.
4. லாஜிக் கேட்ஸ் சேர்க்க: வலது panel → **Gates** → drag:
 - Two XOR gates (one for $A \oplus B$, one for $(A \oplus B) \oplus \text{Cin}$)
 - Three AND gates (2-input)
 - One OR gate (3-input or use two 2-input ORs chained)
 - (Optional) NOT not needed here.
5. Wire them as follows (click wire tool → connect):
 - A → XOR1 input1; B → XOR1 input2. XOR1 output → XOR2 input1.
 - Cin → XOR2 input2. XOR2 output → Sum output pin.
 - For Cout: A → AND1 input1; B → AND1 input2.
 - XOR1 output ($A \oplus B$) → AND2 input1; Cin → AND2 input2.
 - A → AND3 input1; Cin → AND3 input2. (OR, you may skip AND3 and use standard formula; but we will use three ANDs.)

- AND1, AND2, AND3 outputs → OR gate inputs → OR output → Cout .

6. Arrange neatly; label wires if needed (wire label tool). **Save.**

(தகவல்: இந்த வடிவம் $\text{Sum} = A \oplus B \oplus C_{in}$, $\text{Cout} = AB + (A \oplus B)C_{in} + AC_{in}$ — இரண்டில் சுருக்கம் $\text{Cout} = AB + BC_{in} + AC_{in}$ ஆகும்.)

B. Full-Adder Truth Table (matrix / table form)

இதை copy/paste செய்து subcircuit-ஐ verify செய்.

A B Cin | Sum | Cout

0 0 0 | 0 | 0

0 0 1 | 1 | 0

0 1 0 | 1 | 0

0 1 1 | 0 | 1

1 0 0 | 1 | 0

1 0 1 | 0 | 1

1 1 0 | 0 | 1

1 1 1 | 1 | 1

C. Karnaugh Map — Sum & Cout (3-variable) (நேரடியாக Logisim-

இல் கிளிக் இல்லை; K-map-ஐ வாய்ப்பாகப் பார்க்க உபயோகிக்கவும்)

Sum (S) K-map (A,B across, Cin vertical) — minterms {1,2,4,7}

கண்ணுக்கு ASCII K-map:

		AB			
Cin\ AB		00	01	11	10
0		0	1	0	1
1		1	0	1	0

- Ones at cells: (0,1)=1, (0,3)=1, (1,0)=1, (1,2)=1 → pattern shows XOR parity; **no larger grouping** reduces to simple XOR: $S = A \oplus B \oplus C_{in}$.

Cout K-map — minterms where Cout=1: {3,5,6,7}

		AB			
Cin\ AB		00	01	11	10

Cin \ AB	00	01	11	10
0	0	0	1	0
1	0	1	1	1

Groupings:

- Group 11 column (AB=11) covers (0,2) and (1,2) → gives term $A \cdot B$
- Group three others can be grouped to yield $Cin \cdot (A + B)$ (since when $Cin=1$ and (A or B)=1 $\Rightarrow Cout=1$)

Thus simplified: $Cout = A \cdot B + Cin \cdot (A + B)$ which expands to $AB + ACin + BCin$

— the common minimal SOP. (இந்தது K-map மூலம் நீங்கள் group கிளிக்-போல கணிக்கலாம்.)

D. Create 8-bit Ripple Adder — Step-by-step clicks

1. Logisim → Project → **Add Circuit...** → Name = Adder8 → OK.
2. Subcircuit reuse: From toolbar left, open **Circuits** list → choose FullAdder (the one we made).
 - Drag eight copies of the FullAdder component onto the Adder8 canvas. (If FullAdder not shown as component, use **Add Component** → choose from library → select your circuit.)
3. Place them horizontally left→right for bit 0..7 (LSB at left recommended).
4. Add two 8-bit Pin inputs for A and B:
 - From **Wiring** → Pin click, place one; double-click → set Data Bits = 8, Label = A[7..0] , Facing = West.
 - Same for B.
5. Add a single Pin for cin (1-bit) set Label = cin (this will be initial carry; choose 0 for addition).
6. Add Splitters to break the 8-bit bus to single bits:
 - **Wiring** → Splitter → click on canvas → set Bit width in = 8, Bit width out = 1, Fan out = 8, Split left to right ordering such that output 0 is LSB. (Important: Logisim splitter has ordering — set to match your bit ordering.)
 - Connect A Pin to Splitter, do same for B .
7. Connect Splitter outputs to each FullAdder:
 - A_split[0] → FullAdder0.A ; B_split[0] → FullAdder0.B ; FullAdder0.Cin → cin pin (initial).

- FullAdder0.Cout → FullAdder1.Cin ; FullAdder1.Cout → FullAdder2.Cin ... chain until FullAdder7.Cout (this is final carry).
- For each FullAdder i: its Sum output → connect to a `wire` and rejoin into a `Splitter` or use `Bit-join` (create an 8-bit output). Simpler: create 8 single output pins for S0..S7 and later use a Joiner/Splitter to make them an 8-bit bus.

8. Create Sum bus:

- Either place eight output pins (1-bit) labeled S0..S7 and then use a Splitter in reverse to form 8-bit bus, or place a Splitter configured to join 8 single bits into 8-bit output. (Easier: use 8 LED outputs for testing.)

9. Add labels: A[7..0] , B[7..0] , S[7..0] , CarryOut .

10. Save.

E. Wiring details & exact clicks for chaining carries

- Click FullAdder0 Cout wire → drag → connect to FullAdder1 Cin . Do this iteratively.
- Double-check: FullAdder i Cin must connect only to previous Cout; i.e., no short circuits.
- For initial Cin set to 0: set its Pin value property to 0 default or place a toggle input (Pin with attribute "Clock?" No — use 'Tick' or 'TickLabel?'). Better: use an **Input Pin** and set its value to 0 initially (or add a button to toggle for subtract case).

F. Test vectors (how to click to test)

1. Place an **Input** component of type `Hexadecimal` (or binary) to set A and B easily:
 - **Wiring** → `Hex Digit` or use multiple `Data Bits` pins. If no hex widget, use 8 single `Toggles` (Input pins) each representing a bit. For quick: use `Poke Tool` in Logisim to change pin values during simulation.
2. Click the **Poke Tool** (hand icon) → click each input pin to toggle 0/1.
3. Test cases to try:
 - A = 00000001, B = 00000001 → S = 00000010, Cout = 0
 - A = 11111111, B = 00000001 → S = 00000000, Cout = 1 (overflow unsigned)
 - A = 01111111 (127), B = 00000001 (1) → S = 10000000 (−128 signed overflow flag; check signed overflow separately)

G. Two's complement subtract (single click change)

- To make adder handle subtract: For B bus, place eight XOR gates feeding each B_bit with a SUB control bit:
 - Add a 1-bit Pin label SUB .
 - For each $B_split[i]$: $B_i \rightarrow XOR_i$ input1; $SUB \rightarrow XOR_i$ input2; XOR_i output $\rightarrow FullAdder_i.B$.
 - Set initial Cin to SUB pin.
- Click $SUB = 1$ to compute $A - B$ (two's complement), $SUB = 0$ for addition.

H. Carry-Lookahead hint (if you want speed)

- Create per-bit:
 - $G_i = A_i \text{ AND } B_i$
 - $P_i = A_i \text{ XOR } B_i$
- Use CLA equations:
 - $C1 = G0 + P0 \cdot C0$
 - $C2 = G1 + P1 \cdot G0 + P1 \cdot P0 \cdot C0$
 - ... continue up to $C8$.
- (Practical): Build 4-bit CLA module and chain two 4-bit CLAs to make 8-bit CLA (faster than ripple). Clicking: create subcircuit `CLA4`, implement equations with AND/OR gates and small carry generate blocks.

I. Matrix / Tabular view for 8-bit (what to prepare as documentation)

Make a table in your project README (or in Logisim notes) showing per-bit values:

Bit i	A_i	B_i	$P_i (A \oplus B)$	$G_i (A \cdot B)$	Cin_i	Sur
0					$C0$	$S0$
1					$C1$	$S1$
...						
7					$C7$	$S7$

(உனக்கு கீதகொடுப்பனில் இதை **Excel / CSV** ஆக **export** செய்து **Logisim project folder**-இல்

(உணர்வு அல்லாதவர்கள் இதை Excel/CSV ஆல் export செய்து Logisim project-ல் import செய்து save செய்யலாம்.)

J. Quick Troubleshooting (clicks to debug)

- If a Sum bit always 0: check XOR connections (click on wire to see connected endpoints).
- If multiple Cout tied: ensure each Cout only wires to next Cin (no fanout to other Cin).
- Use **Probe** tool: In Logisim toolbar, choose **Probe** → click a wire to show real-time 0/1 value. Very helpful to trace carries.

K. Short Summary — Step checklist (copy & paste)

1. Create FullAdder circuit using XOR/AND/OR.
2. Verify FullAdder truth table with Poke Tool.
3. New circuit **Adder8** : place 8 FullAdder instances.
4. Add 8-bit input pins A, B; add Splitters to break them into bits.
5. Chain Cout→next Cin across slices.
6. Collect Sum outputs into 8-bit bus (or 8 outputs).
7. Test with Poke tool using test vectors.
8. (Optional) Add SUB XORs + Cin=SUB for subtraction.
9. (Optional) Build CLA blocks from G/P signals using Karnaugh simplified terms.

அவ்வளவுதான் — மாற்றமில்லாமல் நிச்சயமாக உன்னிடம் **Logisim** இருக்கிறதா? (நான் கேள்வி கேட்கக்கூடாது — ஆனால் உனக்கு நீங்க சொல்லாம போயிருக்கலாம்). நீ இதை படி-படி செய்யு; நான் இங்கே இன்னும் கேட்காமல் 2 விஷயங்கள் உடனே கொடுப்பேன்:

1. **Full-Adder K-map image (ASCII கொடுத்தது மேலே)** — அதேபோல, நீ **Karnaugh grouping**-ஐ கையால் செய்ய நீண்டதொரு நேரம் வேண்டாம்; Cout için சுருக்கம் $AB + Cin(A + B)$ என்று இருக்கு — இந்த expression-ஐ Logisim-ல் 2 ANDs + 1 OR சாலவாக உருவாக்கலாம் (மூன்று-term expansion பயன்படுத்த வேண்டாம்).
2. **If you want**, நான் நேரடியாக ஒரு **downloadable Logisim .circ file** உருவாக்கி தர முடியாது (இங்கு **file generation** பக்கத்துக்கு அனுமதி இல்லை). ஆனாலும், நீ சொல்லு — நான் உனக்கு circuit-இன் **textual step-by-step XML snippet** அல்லது **4-bit working example XML** கொடுக்கலாம், நீ அதைப் **paste** செய்து Logisim-இல் **import** செய்யலாம். அகை செய்யவா?

நீ இதுவரை படி-படி செய்து பார்த்தா என்ன பிரச்சனை வந்தது, எந்த **step**-இல் திணறுகிறாய் என்று சொல்லு — நான் அதற்கேற்ற சரியான **wire-level clicks/patch** கொடுப்பேன்.

in table form

8-பிட் Adder — Table form (தமிழில்)

கீழே Logisim-ல் 8-bit adder (ripple), அதற்கான 1-bit Full-Adder, Karnaugh சுருக்கங்கள், கட்டுமான படிகள் (click-by-click) அனைத்தும் **எ**அட்டவணை வடிவில் தருகிறேன் — நீ அதை 그대로 copy செய்து படியே படியே செய்யலாம்.

1) Full-Adder — Truth table (1-bit)

A	B	Cin	Sum (S)	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

2) Karnaugh / Simplified expressions (குறுகிய விதம்)

Function	Minterms (1s)	K-map result (short)	Simplified Boolean
Sum (S)	{1,2,4,7}	parity → no grouping beyond XOR	$S = A \oplus B \oplus \text{Cin}$

Cout

{3,5,6,7}

groups → AB and
Cin(A + B)

$$\begin{aligned} \text{Cout} &= A \cdot B + \text{Cin} \cdot (A + B) = \\ &AB + AC_{\text{in}} + BC_{\text{in}} \end{aligned}$$

3) Required Components (Logisim palette)

Category	Component	Quantity (min)	Purpose
Wiring	Input Pin (8-bit)	2	A[7..0], B[7..0]
Wiring	Input Pin (1-bit)	2	Cin (initial), SUB (optional)
Wiring	Output Pin (8-bit)	1	Sum S[7..0]
Wiring	Output Pin (1-bit)	1	CarryOut
Wiring	Splitter	2	Break 8-bit buses to bits
Gates	XOR (2-input)	16 (for 8 FA: 2 each)	Sum logic
Gates	AND (2-input)	24 (3 per FA)	For Cout, etc.
Gates	OR (2/3-input)	8+	Combine ANDs for Cout
Subcircuit	FullAdder (made by you)	8	bit-slices

4) Build steps (click-by-click) — condensed table

Step	What to click / set in Logisim	Result / note
1	Project → Add Circuit... → name FullAdder	New subcircuit canvas
2	Palette → Wiring → place 3 Input Pins ; label A , B , Cin (Facing West)	Inputs ready
3	Palette → Wiring → place 2 Output Pins ; label Sum , Cout (Facing East)	Outputs ready
4	Palette → Gates → place XOR, XOR, AND, AND, AND, OR	Logic elements on canvas

5	Wire tool → connect $A, B \rightarrow \text{XOR1}$; $\text{XOR1} \rightarrow \text{XOR2}$ with $\text{cin} \rightarrow \text{Sum}$	$\text{Sum} = A \oplus B \oplus \text{Cin}$
6	Wire tool → connect $A \cdot B$ (AND1), $(A \oplus B) \cdot \text{Cin}$ (AND2), $A \cdot \text{Cin}$ (AND3) → OR → Cout	$\text{Cout} = AB + (A \oplus B)\text{Cin} + A\text{Cin} \rightarrow$ simplified $AB + \text{Cin}(A+B)$
7	save → return to Project main → Add Circuit... → Adder8	New 8-bit adder canvas
8	Palette → Wiring → place Input Pin (8 bits) ; set Data bits=8, label $A[7..0]$	A bus
9	Palette → Wiring → place Input Pin (8 bits) ; set Data bits=8, label $B[7..0]$	B bus
10	Palette → Wiring → place Splitter ; double- click → Bit width in=8, Bit width out=1, Fan out=8, set bit order LSB → output0	Split A to $A0..A7$
11	Repeat step 10 for B	Split B to $B0..B7$
12	Palette → Project → drag FullAdder component → place 8 copies (or copy/paste)	8 bit-slices
13	Wire: $A_split[0] \rightarrow \text{FA0.A}$; $B_split[0] \rightarrow \text{FA0.B}$; $\text{FA0.Sum} \rightarrow S0$ output	LSB slice done
14	Wire: $\text{FA0.Cout} \rightarrow \text{FA1.Cin}$; chain Cout → next Cin for $\text{FA0}.. \text{FA7}$	Ripple carry chain
15	Repeat wiring for bits 1..7: $A_split[i] \rightarrow \text{FAi.A}$; $B_split[i] \rightarrow \text{FAi.B}$; $\text{FAi.Sum} \rightarrow Si$	All bits connected
16	Place Output Pin (8-bit) or 8 single Output pins and join via Splitter → label $S[7..0]$	Sum bus ready
17	Place 1-bit Output → connect to $\text{FA7.Cout} \rightarrow$ label CarryOut	Final carry shown
18	Use Poke Tool → toggle input bits → verify test vectors	Debug & test

5) Bit-to-slice wiring table (exact mapping)

Slice index i	Connect $A_split[i]$ →	Connect $B_split[i]$ →	$FA_i.Sum$ →	$FA_i.Cout$ →
0 (LSB)	FA0.A	FA0.B	S0	FA1.Cin
1	FA1.A	FA1.B	S1	FA2.Cin
2	FA2.A	FA2.B	S2	FA3.Cin
3	FA3.A	FA3.B	S3	FA4.Cin
4	FA4.A	FA4.B	S4	FA5.Cin
5	FA5.A	FA5.B	S5	FA6.Cin
6	FA6.A	FA6.B	S6	FA7.Cin
7 (MSB)	FA7.A	FA7.B	S7	Final CarryOut

6) Subtraction (two's complement) — table

Action	Clicks in Logisim	Effect
Add SUB control	Place 1-bit Input Pin → label SUB	SUB=0: add; SUB=1: subtract
Invert B when SUB=1	For each $B_split[i]$ → place XOR gate: $B_i \oplus SUB$ → feed $FA_i.B$	If SUB=1, B bits inverted
Set initial carry	Connect SUB pin → to FA0.Cin (initial Cin)	SUB=1 adds the +1 for two's complement
Test example	Poke: SUB=1, A=00000101(5), B=00000011(3)	Result S = 00000010 (2) →

7) Useful test vectors (click to set via Poke tool)

Test #	A (bin)	B (bin)	SUB	Expected S (bin)	Carry
--------	---------	---------	-----	------------------	-------

Step	A (8-bit)	B (8-bit)	Cin	Sum (8-bit)	Cout
1	00000001	00000001	0	00000010	
2	11111111	00000001	0	00000000	
3	01111111	00000001	0	10000000	Cout=0 (sign overflow: check S)
4 (sub)	00000110 (6)	00000011 (3)	1	00000011 (3)	
5 (sub borrow)	00000000	00000001	1	11111111 (two's complement -1)	Cout indicates borrow (unsigned)

8) Debug checklist (table)

Symptom	What to click/check	Fix
Some Sum bits always 0	Click wire; probe XOR inputs	Rewire XOR inputs A/B/Cin
All Cout tied together	Click each Cout wire	Ensure Cout of FA _i → only to FA _(i+1) .Cin
Output ordering reversed	Click Splitter → check bit order	Flip splitter bit order LSB↔MSB
Incorrect subtraction	Click XOR on each B bit & SUB wiring	Ensure SUB → XOR second input; SUB → initial Cin

பக்கவாட்டு குறிப்புகள்:

- Logisim-இல் **Splitter** bit-order கவனம் மிக அவசியம் — **LSB index 0** என்பதை சரிபார்.
- Karnaugh தொகுப்புகள் (Sum/Cout) மேலே கொடுக்கப்பட்ட **simplified forms**-ஐ Logisim-ல் பயன்படுத்தி **gate** எண்ணிக்கையை குறைக்கலாம் ($Cout = AB + Cin(A+B)$).

நீ எதைத் தேர்ந்தெடுப்பாய் — நான் இத்தேய்தான் 1) இதைப் பெற்று **paste** செய்யும் **XML snippet** (FullAdder subcircuit) கொடுத்து தரலாமா? அல்லது 2) உன்னுக்காக 4-bit .circ XML கொடுக்கலாமா? சொல்லாமல் இருக்கிறேன் — நீ இன்று இந்த **table** படி செய்யும் போது பிரச்சனை வந்தால் நேரடியாக அந்த **step**-இன் பெயரை (e.g., Step 12) க்கு மட்டுமே சொல்லு; நான் அதற்கான **wire-level** குறிப்பு மற்றும் **quick-fix** உடனே கொடுத்துவிடுவேன்.

) Full-Adder — Truth table (1-bit)

A B Cin Sum (S) Cout

0 0 0 0 0 which of these are input and output

Full-Adder-ல் எது **Input?** எது **Output?** — புரிய மிக எளிய விளக்கம் 🙌

Full Adder (1-bit)

Inputs (3)

Symbol	Meaning
A	First input bit
B	Second input bit
Cin	Carry-in from previous bit

🙌 இவை மூன்றும் **inputs**

- Logisim-ல் Input Pin வைத்து, label: **A, B, Cin**

Outputs (2)

Symbol	Meaning
Sum (S)	$A + B + \text{Cin}$ இன் least significant bit
Cout	Output carry (next bit-க்கு செல்லும் carry)

🙌 இவை இரண்டு **outputs**

- Logisim-ல் Output Pin வைத்து, label: **Sum, Cout**

Truth Table — Input / Output mapping (final)

Inputs →			Outputs →	
A	B	Cin	Sum (S)	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Summary (super easy)

- Inputs:** A, B, Cin
- Outputs:** Sum, Cout

நீ Full Adder வரைந்து காட்ட சொல்ல வேண்டுமா? அல்லது 8-bit adder-க்கு wiring table வேணுமா?

THIS IS FOR SUM " $\sim A \sim B \text{ Cin} + \sim A B \sim \text{Cin} + A \sim B \sim \text{Cin} + A B \text{ Cin}$ " AND COUT " $B \text{ Cin} + A \text{ Cin} + A B$ " SO TELL IS THIS CORRECT

Short answer: **Yes — both are correct.** I'll show why and give the simpler/suggested

forms you should use in Logisim.

1) Your Sum expression

You wrote:

$$\sim A \sim B \text{ Cin} + \sim A B \sim \text{Cin} + A \sim B \sim \text{Cin} + A B \text{ Cin}$$

These are the minterms for $(A,B,\text{Cin}) = 001, 010, 100, 111$ — exactly the rows where $\text{Sum} = 1$ in the full-adder truth table.

So it is a correct SOP for **Sum**.

Cleaner form:

$$\text{Sum} = A \oplus B \oplus \text{Cin}$$

(Implement with two 2-input XOR gates in Logisim: $X1 = A \oplus B$, $\text{Sum} = X1 \oplus \text{Cin}$.)

2) Your Cout expression

You wrote:

$$B \text{ Cin} + A \text{ Cin} + A B$$

That is $AB + A \cdot \text{Cin} + B \cdot \text{Cin}$. This matches the carry truth table. So **Cout is correct**.

Equivalent, common simplified forms:

- $\text{Cout} = AB + \text{Cin} \cdot (A + B)$
- also $\text{Cout} = AB + (A \oplus B) \cdot \text{Cin}$ (useful for gate implementation)

Logisim-friendly implementation:

- Compute $X = A \oplus B$ (reuse the XOR from Sum stage).
 - $\text{Cout} = \text{OR}(\text{AND}(A,B), \text{AND}(X, \text{Cin}))$ — i.e., two ANDs and one OR.
-

3) Quick mapping (for your confidence)

(A B Cin)	Sum (your terms)	Cout (your terms)
0 0 0	0	0
0 0 1	1 ($\sim A \sim B \text{ Cin}$)	0
0 1 0	1 ($\sim A B \sim \text{Cin}$)	0

0 1 1	0	1 ($B \cdot C_{in}$)
1 0 0	1 ($A \sim B \sim C_{in}$)	0
1 0 1	0	1 ($A \cdot C_{in}$)
1 1 0	0	1 ($A \cdot B$)
1 1 1	1 ($A \cdot B \cdot C_{in}$)	1 ($A \cdot B$ and others)

4) Implementation tips (Logisim)

- Build FullAdder with:
 - $XOR1 = A \oplus B$
 - $Sum = XOR1 \oplus C_{in}$
 - $Cout = OR(AND(A, B), AND(XOR1, C_{in}))$
- This uses fewer gates than directly implementing the 4-term SOP for Sum and the 3-term SOP for Cout.

If you want, I can give the exact Logisim component layout (which gates to place and how to wire them) in a tiny table you can follow click-by-click. Want that?

IS THIS CORRECT OR NOT IN TWELL YES OR NO

Yes — correct.

FOR SUBTRACTION 16 BIT

குறுகிய பதில் — **16-bit subtraction** செய்வது ஆம், இது சரி; கீழே Logisim-ல் படி-படி (தொகுத்து அட்டவணை வடிவில்) எப்படி செய்ய வேண்டும் என்று கொடுத்திருக்கிறேன் —

two's complement முறை பரநதுரைக்கப்பட்டது (எளியது, உவகமானது).

1) முறை — **Two's complement** (அமல்படுத்த இது பயன்படுத்தவும்)

விக்கியம்: $A - B = A + (\text{two's complement of } B) = A + (\sim B) + 1$

2) சுருக்கப்பட்ட தேவையான கூறுகள் (Table)

உருப்படி	எவ்வளவு	நோக்கம்
Input Pin (16-bit)	2	A[15..0], B[15..0]
Input Pin (1-bit)	1	SUB (0=add, 1=sub)
Splitter	2	A to 16 bits, B to 16 bits
XOR gates (2-input)	16	$B_i \oplus SUB \rightarrow$ inverts B when SUB=1
FullAdder slice (1-bit)	16	Each does Sum _i , Cout _i (use FullAdder subcircuit)
Output Pin (16-bit)	1	S[15..0] (result)
Output Pin (1-bit)	1	CarryOut (final carry)
(Optional) Gate set	few	Overflow detection logic

3) கட்டமைப்புக் கட்டளைகள் (click-by-click table)

படி	என்ன கிலிக் செய்ய வேண்டும் / செய்யும் காரியம்	குறிப்பு
1	Project → Add Circuit → Name = FullAdder (இல்லையெனில் reuse)	1-bit full adder subcircuit
2	New circuit → Add Circuit → Name = Sub16	16-bit subtract/add circuit
3	Wiring → Place Input Pin (set Data bits=16) → Label A[15..0]	A bus
4	Wiring → Place Input Pin (Data bits=16) → Label B[15..0]	B bus

5	Wiring → Place Input Pin (Data bits=1) → Label SUB	SUB=0 add, SUB=1 subtract
6	Wiring → Place Splitter → double-click → Bit width in=16, Bit width out=1, Fan out=16, set order LSB→output0	Connect A bus to A_split
7	Repeat Step 6 for B bus	B_split[0..15]
8	Gates → Place 16 XOR gates (2-input)	For i=0..15: connect B_split[i] and SUB → XOR_i output → FullAdder_i.B
9	Place 16 FullAdder instances horizontally (FA0..FA15)	Or use one FA subcircuit and copy/paste
10	Wire: A_split[i] → FAi.A ; XOR_i output → FAi.B	For all i 0..15
11	Set initial carry: connect SUB pin → FA0.Cin	This provides the +1 when SUB=1
12	Chain carries: FAi.Cout → FA(i+1).Cin for i=0..14	Ripple carry chain
13	FAi.Sum outputs → collect into 16-bit output bus S[15..0] via Splitter join	Use Splitter in reverse or place 16 single Output pins and join
14	FA15.Cout → connect to 1-bit Output → label CarryOut	For unsigned borrow check
15	(Optional) Overflow detection: place gates to compute overflow flag	See next table
16	Save → Use Poke tool to set A,B,SUB and verify	Test vectors below

4) Overflow / Borrow detection (table)

Flag	How to compute	Meaning
Unsigned borrow	If final CarryOut = 0 after	For unsigned numbers: borrow = 1 means A<B

(borrow occurred)	$A + (\sim B) + 1 \rightarrow$ borrow occurred. So borrow = $\neg \text{CarryOut}$	
Signed overflow (two's complement)	$\text{Overflow} = (A_{15} \& \sim B_{15} \& \sim S_{15}) \text{ OR } (\sim A_{15} \& B_{15} \& S_{15})$ OR equivalently $\text{Overflow} = (C_{15} \text{ xor } C_{16})$ when using carry into and out of MSB in certain implementations	

Logisim wiring for signed overflow (practical):

- Let A_{15} = MSB of A, B_{15} = MSB of B, S_{15} = MSB of S.
- Implement $\text{Overflow} = (A_{15} \& \sim B_{15} \& \sim S_{15}) \text{ OR } (\sim A_{15} \& B_{15} \& S_{15})$ using AND/NOT/OR gates.

5) Bit mapping table (how bits connect)

Bit i	$A_split[i] \rightarrow$	$B_split[i] \rightarrow$	$\text{XOR } (B_i \oplus \text{SUB}) \rightarrow$	$FA_i.A$	$FA_i.B$	
0 (LSB)	A0	B0	$X0 = B0 \oplus \text{SUB}$	$FA0.A \leftarrow A0$	$FA0.B \leftarrow X0$	F_0
1	A1	B1	$X1 = B1 \oplus \text{SUB}$	$FA1.A \leftarrow A1$	$FA1.B \leftarrow X1$	F_1
...
15 (MSB)	A15	B15	$X15 = B15 \oplus \text{SUB}$	$FA15.A \leftarrow A15$	$FA15.B \leftarrow X15$	FA_{15}

6) Test vectors (click with Poke tool)

#	SUB	A (hex/bin)	B (hex/bin)	Expected result (hex/bin)	Notes
1	1	0000 0000 0000 0110 (6)	0000 0000 0000 0011 (3)	0000 0000 0000 0011 (3)	simple subtract
2	1	0000 0000	0000 0000	1111 1111	borrow:

		0000 0000 (0)	0000 0001 (1)	1111 1111 (-1 two's comp)	CarryOut=0
3	1	0111 1111 1111 1111 (max pos)	1111 1111 1111 1111 (-1)	check signed overflow flag	demonstrates signed overflow
4	0	any	any	addition works as normal	SUB=0 acts as adder

7) Practical tips & pitfalls (table)

Issue	Why it happens	How to fix
Result wrong for subtraction	Forgot to connect SUB → FA0.Cin (so +1 not added)	Ensure SUB pin wired to FA0.Cin
B not inverted on subtract	XOR gate missing or SUB not connected to XOR	For each bit connect B _i and SUB to XOR, output to FA _i .B
Bit order reversed	Splitter bit order set incorrectly	Double-click splitter → set bit order LSB→output0 (or adjust)
Wrong borrow interpretation	Misread CarryOut meaning	For subtraction borrow = ¬CarryOut for unsigned; document it
Signed overflow misdetected	Using CarryOut alone	Use overflow formula above (A15,B15,S15)

8) Quick summary (one-line)

- **Implement 16-bit subtract** in Logisim by: split buses → XOR each B bit with SUB → connect SUB to initial Cin → chain 16 full adders → interpret CarryOut and Overflow correctly.

நீ இதை இன்னும் சுருக்கமாக **XML snippet** அல்லது **4-bit** உதாரணம் வேண்டும் என்று சொன்னால் நான் உடன் ஒரு **4-bit .circ**-இற்கான **XML copy-paste-able snippet** (அதனை Logisim-இல் Import செய்யலாம்) தருகிறேன். இப்போது சொல்றதா?

GIVE TABLE FORM

Below is the **cleanest possible TABLE FORM** for **16-bit subtraction using a full-adder chain** (two's-complement method).
Only tables — no paragraphs.

✓ **TABLE 1 — Inputs & Outputs**

Signal	Width	Direction	Meaning
A	16-bit	Input	Minuend
B	16-bit	Input	Subtrahend
SUB	1-bit	Input	1 = Subtract, 0 = Add
S	16-bit	Output	Result
CarryOut	1-bit	Output	Unsigned borrow indicator (borrow = NOT CarryOut)
Overflow	1-bit	Output	Signed overflow flag

✓ **TABLE 2 — Bit-by-bit connections (for 0..15 bits)**

Bit i	A(i)	B(i)	XOR Out: B(i) ⊕ SUB	FullAdder.A	FullAdder.B	Fu
0	A0	B0	X0	A0	X0	SL
1	A1	B1	X1	A1	X1	C1
2	A2	B2	X2	A2	X2	C2

3	A3	B3	X3	A3	X3	C3
...
14	A14	B14	X14	A14	X14	C1
15	A15	B15	X15	A15	X15	C1

Note: $X(i) = B(i) \text{ XOR } SUB$

Cin(0) = SUB

 **TABLE 3 — Logic equations (needed for circuit)**

Signal	Equation
X(i)	$B(i) \oplus SUB$
Sum(i)	$A(i) \oplus X(i) \oplus Cin(i)$
Cout(i)	$A(i) \cdot X(i) + A(i) \cdot Cin(i) + X(i) \cdot Cin(i)$
Borrow (unsigned)	$\neg CarryOut$
Overflow	$(A15 \& \sim B15 \& \sim S15) + (\sim A15 \& B15 \& S15)$

 **TABLE 4 — Required components count**

Component	Quantity	Purpose
Input Pins	3	A(16), B(16), SUB
Output Pins	3	S(16), CarryOut, Overflow
Splitters	2	Break A & B buses
XOR Gates	16	Invert B when SUB=1
Full Adders	16	Ripple chain
AND/OR/NOT	~6	Overflow logic

✓ TABLE 5 — Test Vectors

SUB	A (hex)	B (hex)	Operation	Expected Result S	CarryOut	Bor
1	0006	0003	6 − 3	0003	1	0
1	0000	0001	0 − 1	FFFF	0	1
1	7FFF	FFFF	32767 − (−1)	Overflow	—	—
0	0004	0002	4 + 2	0006	1	—

If you want, I can give:

- ✓ Full 16-bit adder–subtractor table in CSV
- ✓ Table for Karnaugh map simplification (per bit)
- ✓ Logisim step-by-step clicking table

Just tell me **which table** you need next.

GIVE IN TRUTH TABLE

A **16-bit** subtraction truth table is **impossible** to write (it would need 2^{33} rows = **8,589,934,592** rows**).

Truth tables are only possible for **1-bit** or **2-bit** logical units.

So I will give you the **correct truth table for 1-bit subtractor (using A − B − BorrowIn)**. This is the base cell that repeats 16 times.

✓ 1-BIT SUBTRACTOR TRUTH TABLE (standard)

Inputs & Outputs

inputs → Outputs

A, B, Bin → Difference, Bout

A	B	Bin	Difference	BorrowOut (Bout)
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

✓ Equations derived from the truth table

Difference = $A \oplus B \oplus \text{Bin}$

BorrowOut = $(\sim A \& B) + (B \& \text{Bin}) + (\sim A \& \text{Bin})$

These equations are correct and used in Logisim and in hardware.

! If you want 16-bit subtraction:

You simply **repeat this 1-bit truth table** for all 16 bits with borrow chaining:

Bit0: A0, B0, Bin = SUB → Diff0, Borrow1

Bit1: A1, B1, Borrow1 → Diff1, Borrow2

...

Bit15: A15, B15, Borrow15 → Diff15, BorrowOut

If you want:

✓ Full 1-bit adder-subtractor truth table

✓ 2-bit subtractor truth table

✓ Truth table expanded with XOR(B,SUB)

✓ Truth table for “ $A - B = A + (\sim B) + 1$ ”

I can provide those too.

Which one do you want next?

If you want 16-bit subtraction:

You simply repeat this 1-bit truth table for all 16 bits with borrow chaining: GIVE INPUTS AND IUTPUTS TO GIVE IN MATRICX IN LOGISIM

அற்புதம் — கீழே **Logisim-இல் 16-bit subtract (two's-complement / XOR+adder)** மாதிரியான மாட்ரிக்ஸ் (**matrix**) — ஒவ்வொரு bit-slice இற்கும் எந்த **INPUTS**, எந்த **INTERMEDIATE node (XOR)** மற்றும் எந்த **OUTPUTS** என்பதைக் காட்டும். இதில் SUB = 1 → *subtract*, SUB = 0 → *add*. எல்லா வரிசைகளும் பொதுவான சூத்திரங்களை மட்டும் காட்டுகிறது (பிட்-விலக்குகள் அல்ல).

குறிப்பு: Logisim-ல் A, B என்பதை 16-bit Input Pins ஊடாக கொண்டு, அவை Splitter உபயோகித்து தனி A0..A15, B0..B15 ஆக பிரிக்கப்பட வேண்டும். SUB என்பது 1-bit Input Pin. Cin0 = SUB என்று அமைக்கவும்.

Matrix — per-bit connections (i = 0..15)

Bit i	Inputs	Intermediate	FullAdder Inputs	FullAdder Operation	
i	A(i), B(i), SUB (global)	$X(i) = B(i) \oplus$ SUB	FA_A = A(i) ; FA_B = X(i) ; FA_Cin = Cin(i)	FA computes: $Sum(i) = A(i) \oplus X(i)$ $\oplus Cin(i)$; Cout(i) = $A \cdot X + A \cdot Cin + X \cdot Cin$	S

ವ್ಯಾಖ್ಯೆ (expanded row meaning):

- $Cin(0) = SUB$
- For each i : $X(i) = B(i) \text{ XOR } SUB$ (this inverts B when $SUB=1$)
- Connect $A(i) \rightarrow \text{FullAdder A input}$, $X(i) \rightarrow \text{FullAdder B input}$, $Cin(i) \rightarrow \text{FullAdder Cin input}$
- FullAdder outputs: $S(i) \rightarrow \text{result bit } i$, $Cout(i) \rightarrow \text{next stage } Cin(i+1)$
- Final $Cout(15) \rightarrow \text{CarryOut}$ (unsigned borrow = NOT CarryOut).

Large table (explicit columns) — copy-paste friendly

Bit	A(i)	B(i)	SUB	X(i)=B(i)⊕SUB	Cin(i)
0	A0	B0	SUB	X0	Cin0 = SUB
1	A1	B1	SUB	X1	Cin1 = C1
2	A2	B2	SUB	X2	Cin2 = C2
3	A3	B3	SUB	X3	Cin3 = C3
4	A4	B4	SUB	X4	Cin4 = C4
5	A5	B5	SUB	X5	Cin5 = C5
6	A6	B6	SUB	X6	Cin6 = C6
7	A7	B7	SUB	X7	Cin7 = C7
8	A8	B8	SUB	X8	Cin8 = C8
9	A9	B9	SUB	X9	Cin9 = C9
10	A10	B10	SUB	X10	Cin10 = C10
11	A11	B11	SUB	X11	Cin11 = C11
12	A12	B12	SUB	X12	Cin12 = C12

...					
13	A13	B13	SUB	X13	Cin13 = C13
14	A14	B14	SUB	X14	Cin14 = C14
15	A15	B15	SUB	X15	Cin15 = C15

Final outputs (matrix row)

Output signal	Definition / source
S[15..0]	Concatenate S15...S0 from each FullAdder
CarryOut	Cout(15) from bit 15
Borrow (unsigned)	Borrow = NOT(CarryOut)
Overflow (signed)	$\text{Overflow} = (A_{15} \ \& \ \sim B_{15} \ \& \ \sim S_{15}) + (\sim A_{15} \ \& \ B_{15} \ \& \ S_{15})$

Quick Logisim wiring checklist (table)

Step	Click/action
1	Place 16-bit Input Pin → label A[15..0]
2	Place 16-bit Input Pin → label B[15..0]
3	Place 1-bit Input Pin → label SUB
4	Place Splitter (in=16, out=1, fanout=16) → connect A and B to splitters
5	For i=0..15: place XOR (2-input): connect B_i and SUB → XOR_i output = X_i
6	Place 16 FullAdder subcircuits → connect A_i, X_i, Cin_i as matrix above

7	Connect Cin0 = SUB ; chain Cout(i) → Cin(i+1)
8	Collect S0..S15 → join into 16-bit Output Pin S[15..0]
9	Connect Cout(15) → CarryOut output ; implement Overflow logic separately

இதை 그대로 Logisim-இல் follow பண்ணு — உனக்கு தேவையானால் நான் இக்குறிப்பில் இருந்து **CSV / Excel** மாதிரியாக S(sheet) உருவாக்கி தரலாம் (அதன் மூலம் **Splitter bit order, pin indices** என்று நீ நேரடியாக paste செய்யலாம்).

இதுதான் உன் கேள்விக்கு தேவையான **INPUTS & OUTPUTS matrix**. இன்னும் எந்த வடிவம் (CSV, Excel, XML snippet, அல்லது ஒரு small 4-bit .circ example) வேண்டும்?

Bit	A(i)	B(i)	SUB	X(i)=B(i)⊕SUB	Cin(i)	FullAdder
Inputs (A,B,Cin) S(i) = A⊕X⊕Cin Cout(i)						
--	--	--	--	-----	-----	-----
-----	-----	-----	-----	-----	-----	-----
0	A0	B0	SUB	X0	Cin0 = SUB	(A0, X0, SUB) S0 C1
1	A1	B1	SUB	X1	Cin1 = C1	(A1, X1, C1) S1 C2
2	A2	B2	SUB	X2	Cin2 = C2	(A2, X2, C2) S2 C3
3	A3	B3	SUB	X3	Cin3 = C3	(A3, X3, C3) S3 C4
4	A4	B4	SUB	X4	Cin4 = C4	(A4, X4, C4) S4 C5
5	A5	B5	SUB	X5	Cin5 = C5	(A5, X5, C5) S5 C6
6	A6	B6	SUB	X6	Cin6 = C6	(A6, X6, C6) S6 C7
7	A7	B7	SUB	X7	Cin7 = C7	(A7, X7, C7) S7 C8
8	A8	B8	SUB	X8	Cin8 = C8	(A8, X8, C8) S8 C9
9	A9	B9	SUB	X9	Cin9 = C9	(A9, X9, C9) S9 C10
10	A10	B10	SUB	X10	Cin10 = C10	(A10, X10, C10) S10 C11
11	A11	B11	SUB	X11	Cin11 = C11	(A11, X11, C11) S11 C12
12	A12	B12	SUB	X12	Cin12 = C12	(A12, X12, C12) S12 C13
13	A13	B13	SUB	X13	Cin13 = C13	(A13, X13, C13) S13 C14
14	A14	B14	SUB	X14	Cin14 = C14	(A14, X14, C14) S14 C15
15	A15	B15	SUB	X15	Cin15 = C15	(A15, X15, C15) S15 Cout

```

| 9 | A9 | B9 | SUB | X9 | Cin9 = C9 | (A9, X9,
C9) | S9 | C10 |
| 10 | A10 | B10 | SUB | X10 | Cin10 = C10 | (A10,
X10, C10) | S10 | C11 |
| 11 | A11 | B11 | SUB | X11 | Cin11 = C11 | (A11,
X11, C11) | S11 | C12 |
| 12 | A12 | B12 | SUB | X12 | Cin12 = C12 | (A12,
X12, C12) | S12 | C13 |
| 13 | A13 | B13 | SUB | X13 | Cin13 = C13 | (A13,
X13, C13) | S13 | C14 |
| 14 | A14 | B14 | SUB | X14 | Cin14 = C14 | (A14,
X14, C14) | S14 | C15 |
| 15 | A15 | B15 | SUB | X15 | Cin15 = C15 | (A15,
X15, C15) | S15 | Cout(15) → CarryOut | WHICH
ARE INPUTS AND OUTPUTS

```

Here is the clear answer in **YES / NO style**:

Below table — **which are INPUTS and which are OUTPUTS** — for your 16-bit subtract/adder.

✓ INPUT SIGNALS (Logisim)

These are given **from outside**:

Global Inputs

- **A[15:0]** → 16-bit input
- **B[15:0]** → 16-bit input
- **SUB** → 1-bit input (0 = ADD, 1 = SUBTRACT)

Per-bit Inputs (from splitters)

- **A(i)** → input
- **B(i)** → input
- **SUB** → input
- **Cin(i)** → input for FullAdder i
 - Note: **Cin(0) = SUB** (this is input for bit0)

🧩 INTERNAL SIGNALS (NOT inputs, NOT outputs — internal wires)

These are inside circuit:

- $X(i) = B(i) \oplus SUB$
- **FullAdder Inputs** = $(A(i), X(i), Cin(i))$

OUTPUT SIGNALS

These come from the FullAdder outputs:

- **S(i)** → output bit
- **Cout(i)** → output (also used as next carry input)
- **Final result S[15:0]** → main output
- **CarryOut = Cout(15)** → main output
- **Borrow = NOT(CarryOut)** (optional output)

✓ Marking each column: INPUT or OUTPUT

Column	INPUT?	OUTPUT?	INTERNAL?	Notes
Bit	No	No	No	Only index
A(i)	✓ Yes	No	No	Input from 16-bit A bus
B(i)	✓ Yes	No	No	Input from 16-bit B bus
SUB	✓ Yes	No	No	Global subtract signal
$X(i) = B(i) \oplus SUB$	No	No	✓ Internal	XOR output
Cin(i)	No	No	⚠ Internal/ Input to FA	Cin(0)=SUB (input), others internal chain
FullAdder Inputs (A,B,Cin)	No	No	Internal	fed into FA
S(i)	No	✓ Yes	No	Output bit
Cout(i)	No	⚠ Yes (for i=15)	Internal for 0– 14	For bit 15 → external output

FINAL SUMMARY (VERY EASY)

External Inputs

- $A(i)$
- $B(i)$
- SUB
- $Cin(0) = SUB$

External Outputs

- $S(i)$
- $Cout(15)$

Internal Signals

- $X(i)$
 - $Cin(1..15)$
 - $Cout(0..14)$
-

If you want, I can convert this into **CSV**, **Excel-table**, or **Logisim circuit diagram (.circ)**.

Global Inputs

$A[15:0] \rightarrow$ 16-bit input

$B[15:0] \rightarrow$ 16-bit input

SUB \rightarrow 1-bit input (0 = ADD, 1 = SUBTRACT) GIVE AS
CODES IMN LOGISIM TO ASUTOMATICALLY CREATER
THE TRUTH TABLEAND MATRIOX

Done — I generated two files you can download and open:

- [Download one_bit_truth_table.csv](#) — 1-bit subtractor truth table (A , B , $Bin \rightarrow$ Difference, BorrowOut).

- [Download per_bit_matrix_16.csv](#) — 16-row symbolic matrix for the per-bit wiring of the 16-bit subtractor ($A(i)$, $B(i)$, SUB, $X(i)$, $Cin(i)$, FA inputs, $S(i)$, $Cout(i)$).

How to use them in your workflow:

- Open the CSV in Excel/LibreOffice/Google Sheets to view or edit the matrix and truth table.
- Use the one-bit CSV to automatically generate test vectors for each bit slice.
- Use the per-bit CSV as a copy-paste checklist to wire Splitter outputs to XORs and FullAdder inputs in Logisim.

If you want, I can now:

- Produce a downloadable Logisim .circ XML that constructs the 16-bit subtractor (ripple) automatically, or
- Produce a CSV containing expanded test vectors for the full 16-bit adder-subtractor for a limited set of inputs (since full 2^{33} table is infeasible).

Which next?

```
A(i) B(i) SUB  X(i)=B(i)⊕SUB  Cin(i)
FullAdderInputs(A,B,Cin)  S(i)=A⊕X⊕Cin  Cout(i) GIVE
IN LOGISIM SDYNTAXC
```

Nice — below I give the exact **Logisim-style wiring syntax / naming convention** you can copy into your project when building the 16-bit adder/subtractor.

Two parts:

1. a **template** (how to name components / signals in Logisim)
2. a **full 16-line mapping** (bit 0..15) you can paste into a text editor and follow exactly when wiring in Logisim.

Use the following conventions in Logisim:

- Create two **16-bit Input Pins** named **A** and **B**.
- Create a **1-bit Input Pin** named **SUB**.
- Create a **Splitter** for each bus: set Bit width in = 16, Bit width out = 1, Fan out = 16. **Set bit order so output 0 = A0 (LSB)**. Do same for **B**.
- For each gate or signal, use the **Label** (or Tunnel) with the exact name shown below to make wiring easy.

Template / Logisim syntax for one bit (use for each $i = 0..15$)

```
# Splitter outputs (from Splitter A and Splitter B)
A[i]          // Splitter A output index i - use label A0, A1, ... A15
B[i]          // Splitter B output index i - use label B0, B1, ... B15

# SUB (global 1-bit input)
SUB           // global input pin name

# XOR to invert B when SUB=1
X[i] = XOR(B[i], SUB)           // place a 2-input XOR; label its output "Xi

# Carry-in for this bit
Cin0 = SUB                       // for i=0 only
Cin[i] = C[i] (for i>0)         // i>0: Cin is previous Cout (label previous

# Full Adder instance naming & connections
Place FullAdder component instance named FA[i]
Connect:
    FA[i].A    <- A[i]
    FA[i].B    <- X[i]
    FA[i].Cin  <- Cin[i]

# Full Adder outputs
FA[i].Sum  -> label S[i]           // S0..S15 are output bits
FA[i].Cout -> label C[i+1]         // Cout from FA[i] name as C1..C16; C16 is i

# Final outputs
S[15..0]   // join S0..S15 with a splitter set to 16-bit output
CarryOut = C16
Borrow = NOT(CarryOut) // optional: label as BORROW
Overflow = (A15 & ~B15 & ~S15) | (~A15 & B15 & S15) // optional gates to c
```

Full explicit mapping (copyable, one row per bit)

(Replace i with $0..15$. These are the exact label names to use in Logisim.)

```
Bit 0:
    A0    = Splitter_A.out[0]
    B0    = Splitter_B.out[0]
```

```

X0    = XOR(B0, SUB)           // label XOR output "X0"
Cin0  = SUB
FA0 inputs: (A0, X0, Cin0)
FA0 outputs: Sum->S0, Cout->C1

```

Bit 1:

```

A1    = Splitter_A.out[1]
B1    = Splitter_B.out[1]
X1    = XOR(B1, SUB)           // label "X1"
Cin1  = C1
FA1 inputs: (A1, X1, C1)
FA1 outputs: Sum->S1, Cout->C2

```

Bit 2:

```

A2    = Splitter_A.out[2]
B2    = Splitter_B.out[2]
X2    = XOR(B2, SUB)           // label "X2"
Cin2  = C2
FA2 inputs: (A2, X2, C2)
FA2 outputs: Sum->S2, Cout->C3

```

...

(repeat pattern up to bit 14)

Bit 14:

```

A14   = Splitter_A.out[14]
B14   = Splitter_B.out[14]
X14   = XOR(B14, SUB)          // label "X14"
Cin14 = C14
FA14 inputs: (A14, X14, C14)
FA14 outputs: Sum->S14, Cout->C15

```

Bit 15:

```

A15   = Splitter_A.out[15]
B15   = Splitter_B.out[15]
X15   = XOR(B15, SUB)          // label "X15"
Cin15 = C15
FA15 inputs: (A15, X15, C15)
FA15 outputs: Sum->S15, Cout->C16 // C16 = CarryOut

```

Every signal needs to have a label / Tunnel is required

Exact signal names to place as Labels / Tunnels in Logisim

Place labels/tunnels with these names so wires are easily joined:

- A0, A1, ..., A15 (Splitter A outputs)
- B0, B1, ..., B15 (Splitter B outputs)
- SUB
- X0, X1, ..., X15 (XOR outputs)
- C1, C2, ..., C16 (carry chain; note C1 is Cout from bit0)
- S0, S1, ..., S15 (sum outputs)
- CarryOut (alias of C16)
- Borrow (optional = NOT(CarryOut))
- Overflow (optional expression as gate output)

How to implement quickly in Logisim (step checklist — exact clicks)

1. Place **Input Pin**: name A , Data bits = 16.
2. Place **Input Pin**: name B , Data bits = 16.
3. Place **Input Pin**: name SUB , Data bits = 1.
4. Place **Splitter**: configure in=16 , out=1 , fanout=16 , set bit order so out0 = LSB. Connect A → Splitter_A. Label outputs A0..A15 (use label tool or tunnels). Repeat for B .
5. For i = 0..15: place a **2-input XOR**, connect B<i> and SUB → label its output X<i> .
6. For i = 0..15: place **FullAdder** (or subcircuit you made) named FAi . Connect:
A<i> → FAi.A, X<i> → FAi.B, Cin<i> → FAi.Cin (use Cin0 = SUB , otherwise wire C<i> from previous FA). Label FAi.Sum S<i> and FAi.Cout C<i+1> .
7. Use a **Splitter** in reverse (or join) to form 16-bit output S from S0..S15. Place an Output pin S , Data bits = 16.
8. Place an Output pin CarryOut and wire to C16 . Optionally create Borrow = NOT(CarryOut) and Overflow logic.

Example: exact wiring lines you can follow for bit 0 in Logisim GUI

- Click Splitter_A.out[0] → select wire → right-click → Add Label → type A0 .
- Click Splitter_B.out[0] → wire label B0 .
- Place XOR, connect B0 and SUB → label XOR output X0 .
- Place FullAdder FA0, connect A0 → FA0.A, X0 → FA0.B, SUB → FA0.Cin.

- Label FA0.Sum \rightarrow s0 . Label FA0.Cout \rightarrow c1 .

Repeat for bits 1..15, replacing indices and for Cin use previous c# .

If you want, I can now:

- produce a **ready-to-import Logisim .circ XML** that uses these exact labels and constructs the full 16-bit subtractor (I can generate and give you the file), **or**
- produce a **copyable CSV** with the exact label names (A0..A15, B0..B15, X0..X15, C1..C16, S0..S15) so you can paste into a wiring checklist.

Which do you want next?