**4 - BIT FULL ADDER**

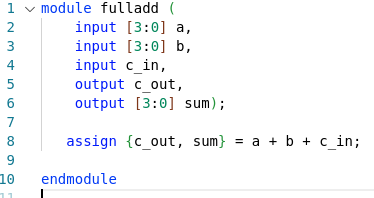
**(combinational circuit)**

We have two Verilog modules:

1. **fulladd (4-bit Full Adder)**
2. **tb\_fulladd (Testbench for fulladd)**

## **fulladd (4-bit Full Adder)**

This module is like a **calculator that adds two 4-bit numbers** along with a carry-in. It gives a 4-bit sum and a carry-out.



### **> Understanding the Code Line-by-Line:**

* We define a module named **fulladd**.
* a and b are **4-bit numbers** (like numbers between 0 to 15 in decimal).
* c\_in is a **single-bit number** (0 or 1), which is the **carry-in**.
* sum is the **4-bit result**.
* c\_out is the **carry-out** (a single-bit that tells if there's an overflow).

assign {c\_out, sum} = a + b + c\_in;

* The {c\_out, sum} notation means:
  + The result of a + b + c\_in is **5 bits long**.
  + The leftmost bit (c\_out) is the **carry-out**.
  + The remaining 4 bits are assigned to sum.

### **Example:**

If:

a = 4'b1100 (12 in decimal)

b = 4'b1011 (11 in decimal)

c\_in = 1

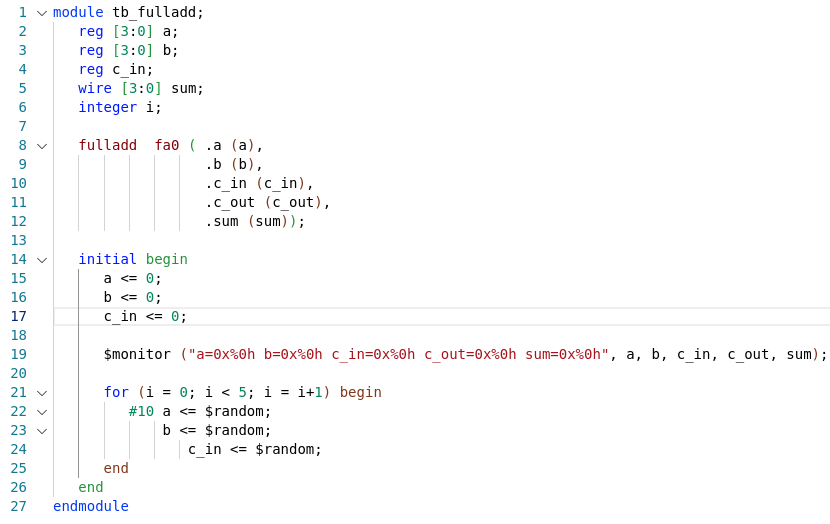
Then:

sum = 1000 (8 in decimal)

c\_out = 1 (since 12 + 11 + 1 = 24, which is 11000 in binary, and the extra '1' goes to carry-out)

## **tb\_fulladd (Testbench for fulladd)**

This is a **test script** that checks if our full adder works correctly.



### **Understanding the Code Line-by-Line:**

* This defines the **testbench** module (no inputs or outputs because it's just for testing).
* a and b: **4-bit registers** (used to store test values).
* c\_in: **Single-bit register** (for carry-in).
* sum and c\_out: **Wire connections** (outputs from fulladd).
* i: **Integer for loop iteration**.

fulladd fa0 ( .a (a),

.b (b),

.c\_in (c\_in),

.c\_out (c\_out),

.sum (sum));

* This **connects** our testbench (tb\_fulladd) to the fulladd module.
* fa0 is an instance of fulladd, meaning **we are testing the real full adder**.

### **$monitor (Displays Output in Console)**

$monitor ("a=0x%0h b=0x%0h c\_in=0x%0h c\_out=0x%0h sum=0x%0h", a, b, c\_in, c\_out, sum);

* $monitor **prints** the values of a, b, c\_in, sum, and c\_out whenever they change.
* The 0x%0h format displays numbers in **hexadecimal**.

### **Random Input Generation**

initial begin

a <= 0;

b <= 0;

c\_in <= 0;

* We **initialize** everything to 0.

for (i = 0; i < 5; i = i+1) begin

#10 a <= $random;

b <= $random;

c\_in <= $random;

end

end

endmodule

* A loop runs **5 times**, generating **random numbers** for a, b, and c\_in every **10 time units**.
* $random picks **random numbers**, helping us test the full adder with different values.

## 

## **How Does This Work Together?**

* tb\_fulladd generates **random test values**.
* These values are **fed into** fulladd.
* The full adder **computes the sum and carry-out**.
* $monitor **prints** the results for checking.

### **Example Output:**

a=0x3 b=0x4 c\_in=0x0 c\_out=0x0 sum=0x7

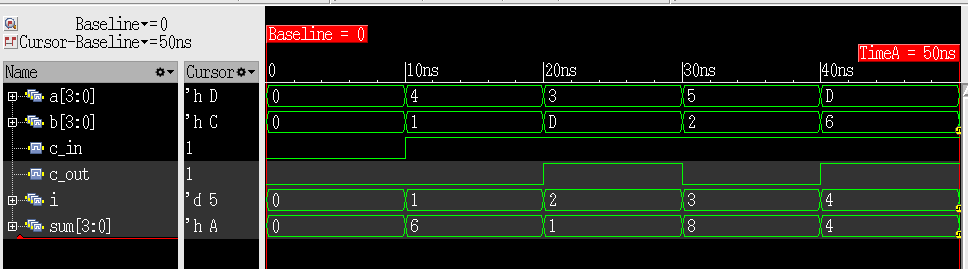
a=0xA b=0xC c\_in=0x1 c\_out=0x1 sum=0x7

a=0x5 b=0xF c\_in=0x1 c\_out=0x1 sum=0x5

## **TL;DR (Too Long; Didn't Read)**

🔹 **fulladd**: A 4-bit adder that takes two 4-bit inputs and a carry-in, then outputs a 4-bit sum and carry-out.  
 🔹 **tb\_fulladd**: A testbench that **randomly tests** the full adder by providing random inputs and printing results.  
 🔹 **Key Line:** assign {c\_out, sum} = a + b + c\_in; → Handles both sum and carry-out.  
 🔹 **Key Debug Tool:** $monitor(...) → Prints values every time they change.

**RESULTS OF CADENCE NC LAUNCH WAVEFORM WINDOW**



### **What Are We Looking At?**

This is a **simulation waveform** for the **4-bit full adder** (fulladd module) tested using the tb\_fulladd. It shows how inputs a, b, and c\_in change over time and how the outputs sum and c\_out respond.

### **Understanding Each Signal:**

| **Signal** | **Description** |
| --- | --- |
| **a[3:0]** | 4-bit input A (changes every 10ns) |
| **b[3:0]** | 4-bit input B (changes every 10ns) |
| **c\_in** | Carry-in (changes occasionally) |
| **c\_out** | Carry-out (output) |
| **sum[3:0]** | 4-bit sum output |
| **i** | Loop iteration counter (not relevant for addition, used for generating values) |

## **Time-by-Time Analysis**

**At Time = 0ns**

* a = 0x0 (0000)
* b = 0x0 (0000)
* c\_in = 1
* **Computation:** 0 + 0 + 1 = 1
* sum = 0x1 (0001)
* c\_out = 0 (No overflow)
* **Why?** Since a and b are 0, only c\_in contributes to the sum.

### **At Time = 10ns**

* a = 0x4 (0100)
* b = 0xC (1100)
* c\_in = 1
* **Computation:** 4 + 12 + 1 = 17 (10001 in binary)
* sum = 0x1 (0001)
* c\_out = 1 (Because the result exceeded 4 bits)
* **Why?** The sum exceeds 4 bits, so the **leftmost bit goes into c\_out**.

### **At Time = 20ns**

* a = 0x8 (1000)
* b = 0xD (1101)
* c\_in = 0
* **Computation:** 8 + 13 + 0 = 21 (10101 in binary)
* sum = 0x5 (0101)
* c\_out = 1
* **Why?** 10101 is **5 bits** long. The leftmost 1 is the carry-out.

### **At Time = 30ns**

* a = 0x6 (0110)
* b = 0x2 (0010)
* c\_in = 0
* **Computation:** 6 + 2 + 0 = 8 (1000 in binary)
* sum = 0x8 (1000)
* c\_out = 0
* **Why?** The result fits within 4 bits, so c\_out stays 0.

### **At Time = 40ns**

* a = 0xD (1101)
* b = 0x6 (0110)
* c\_in = 0
* **Computation:** 13 + 6 + 0 = 19 (10011 in binary)
* sum = 0x3 (0011)
* c\_out = 1
* **Why?** The sum exceeds 4 bits (10011), so the leftmost 1 becomes the carry-out.

## **Transition Breakdown (Why Does the Signal Move?)**

1. **Why does a change every 10ns?**
   * Because $random generates new values every 10ns in the testbench.
2. **Why does b change every 10ns?**
   * Same reason as a, $random updates it.
3. **Why does c\_in stay mostly 1 at first and then 0?**
   * $random is generating it randomly; it just happened to be 1 at first.
4. **Why does sum jump up and down?**
   * Because new inputs cause new results. Simple addition.
5. **Why does c\_out sometimes go 1 and sometimes 0?**
   * If the addition result **exceeds 4 bits**, c\_out turns 1.

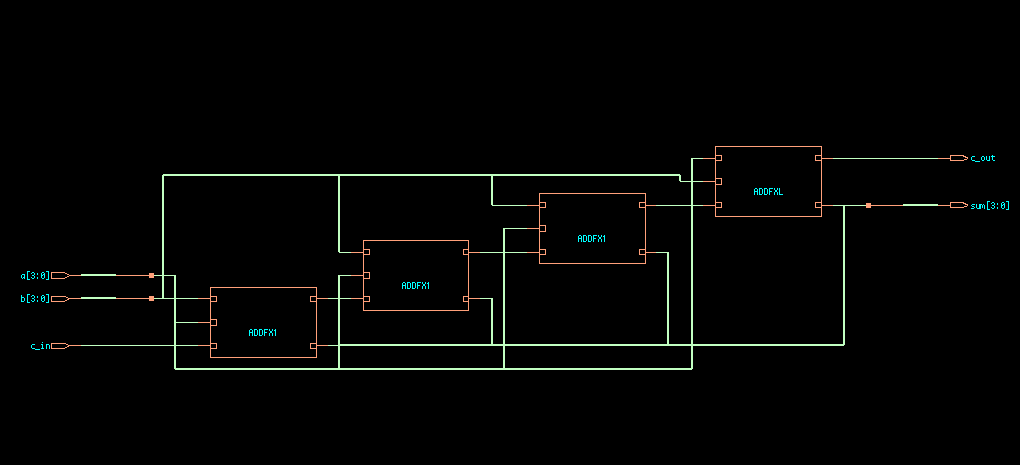
* "Why does it go up?" → Because we added bigger numbers.
* "Why does it go down?" → Because new values are smaller.
* "Why is there a carry?" → Because the sum exceeded 4 bits.
* "Why does everything change at every 10ns?" → Because $random updates the inputs in the testbench every 10ns.

**TCL SCRIPT**



1. **set\_db library /home/user/cadence/FOUNDRY/digital/90nm/dig/lib/slow.lib**
   * Sets the standard cell library (90nm technology) for synthesis.
2. **read\_hdl fulladd.v**
   * Reads the Verilog HDL file (fulladd.v) for synthesis.
3. **elaborate**
   * Converts the HDL design into a structural representation (initial synthesis step).
4. **syn\_generic**
   * Performs technology-independent logic optimization.
5. **syn\_map**
   * Maps the design to the standard cell library (90nm in this case).
6. **syn\_opt**
   * Optimizes the mapped design for timing, power, and area.
7. **write\_hdl > fulladd\_netlist.v**
   * Saves the synthesized Verilog netlist.
8. **write\_sdc > fulladd\_output.sdc**
   * Writes out the constraints file in SDC (Synopsys Design Constraints) format.
9. **gui\_show**
   * Opens the graphical interface to inspect the design.
10. **report timing > fulladd\_timing.rpt**
    * Generates a timing report.
11. **report power > fulladd\_power.rpt**
    * Generates a power report.
12. **report area > fulladd\_cell.rpt**
    * Reports the area usage of the design.
13. **report gates > fulladd\_gates.rpt**
    * Reports the gate count.
14. **clean**
    * Cleans up temporary synthesis files.

**RESULTS OF CADENCE GENUS**

****

This schematic represents a **4-bit Ripple Carry Adder (RCA)**. Here’s how it works:

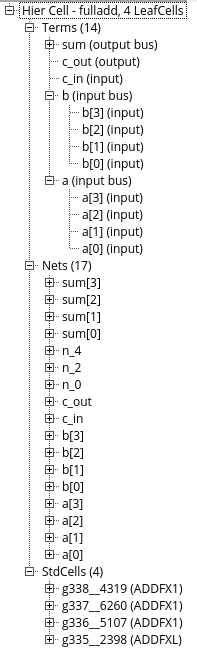
### **1. Blocks & Connections**

* The circuit takes **two 4-bit binary numbers (a[3:0] & b[3:0])** and an **input carry (c\_in)** as inputs.
* It consists of **four 1-bit full adder (FA) modules** labeled **"ADDFA1"**.
* Each **full adder takes one bit** from a and b, along with a carry-in, to compute a sum and a carry-out.

### **2. Working of the Ripple Carry Adder**

* The **first adder** adds the **LSB bits (a[0], b[0])** and c\_in.
* Its **carry-out** is fed into the **next adder** as a carry-in.
* This process continues through all four stages, propagating the carry forward.
* The **final carry-out (c\_out)** represents the carry from the last stage.
* The **sum bits (sum[3:0])** form the final 4-bit result.

### **3. Key Takeaways**

* This is a **combinational circuit** (no clock, operates instantly).
* The **carry ripples from one stage to the next**, which can cause delay.
* It's used in **basic arithmetic operations inside processors and ALUs**.

### **Explanation of the Hierarchy**

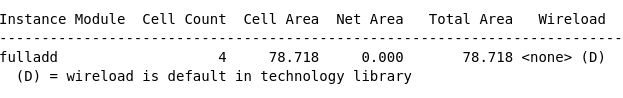
1. **Module Name:** fulladd\_4  
   * Represents a **4-bit Full Adder** circuit.
2. **Inputs & Outputs:**
   * **Inputs:**
     + a[3:0] → 4-bit input bus
     + b[3:0] → 4-bit input bus
     + c\_in → Carry-in
   * **Outputs:**
     + sum[3:0] → 4-bit sum output
     + c\_out → Final carry-out
3. **Nets (Connections between components):**
   * sum[3:0], c\_out, c\_in, a[3:0], b[3:0] are primary nets.
   * n\_4, n\_2, n\_0 → Internal signals for carry propagation.
4. **Standard Cells (Logic Gates used in Design):**
   * Uses **4 full adders (ADDFX1, ADDFXL)**, likely from a **standard cell library**.
   * These are different variations of **full adder cells** with different drive strengths.

### **Summary:**

* This is a **4-bit Ripple Carry Adder** implemented using **standard cell-based full adders** (ADDFX1, ADDFXL).
* It connects **input buses (a, b), a carry-in (c\_in)**, and generates **sum (sum[3:0]) and final carry (c\_out)**.
* The internal nets handle **carry propagation** between the adders.

### **Area Report:**

#### **1. Cell Area Report**



* **Instance Module**: The name of the synthesized module (here, fulladd).
* **Cell Count**: Number of standard cells used in the design (4 cells).
* **Cell Area**: The total area occupied by the logic cells (78.718).
* **Net Area**: The area occupied by wiring (0.000, meaning negligible).
* **Total Area**: The sum of cell and net areas (78.718).
* **Wireload**: Wire load model used, set by the technology library (D means default).

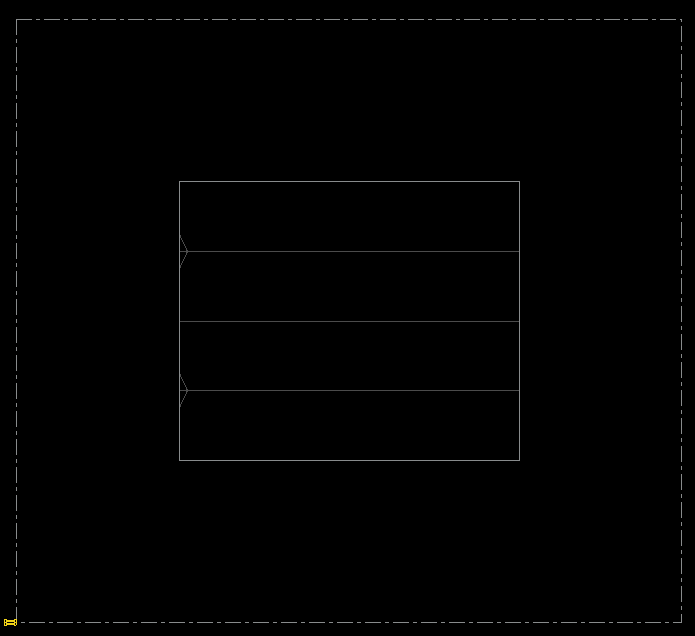
#### 

* **Generated by**: Tool used (Cadence Genus).
* **Operating Conditions**: Library conditions used (slow corner, meaning worst-case timing).
* **Wireload Mode**: Defines how wire capacitance is estimated (enclosed means a predefined model).
* **Area Mode**: Specifies whether optimization prioritizes area (timing library suggests timing-driven synthesis).
* **Gate Instances**:
  + **ADDFX1**: Full-adder cell with fast performance (3 instances).
  + **ADDFXL**: Low-power full-adder cell (1 instance).
* **Type Breakdown**:
  + **Logic**: Area occupied by logic cells (100%).
  + **Physical Cells**: Extra non-logic cells (here, none are used).

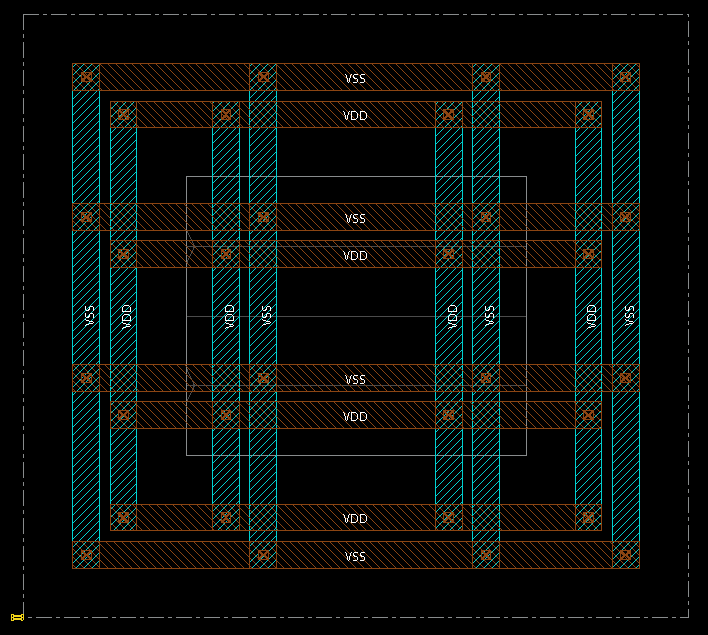
#### **3. Power Report**

* **Power Unit**: Measured in Watts.
* **Categories**:
  + **Memory/Register/Latch**: Not used in this design.
  + **Logic**: The only contributor to power consumption.
* **Power Components**:
  + **Leakage Power**: Power lost due to subthreshold conduction (13.52%).
  + **Internal Power**: Power dissipated internally in transistors (59.49%).
  + **Switching Power**: Power used during logic transitions (27.00%).
* **Total Power**: Sum of leakage, internal, and switching power.

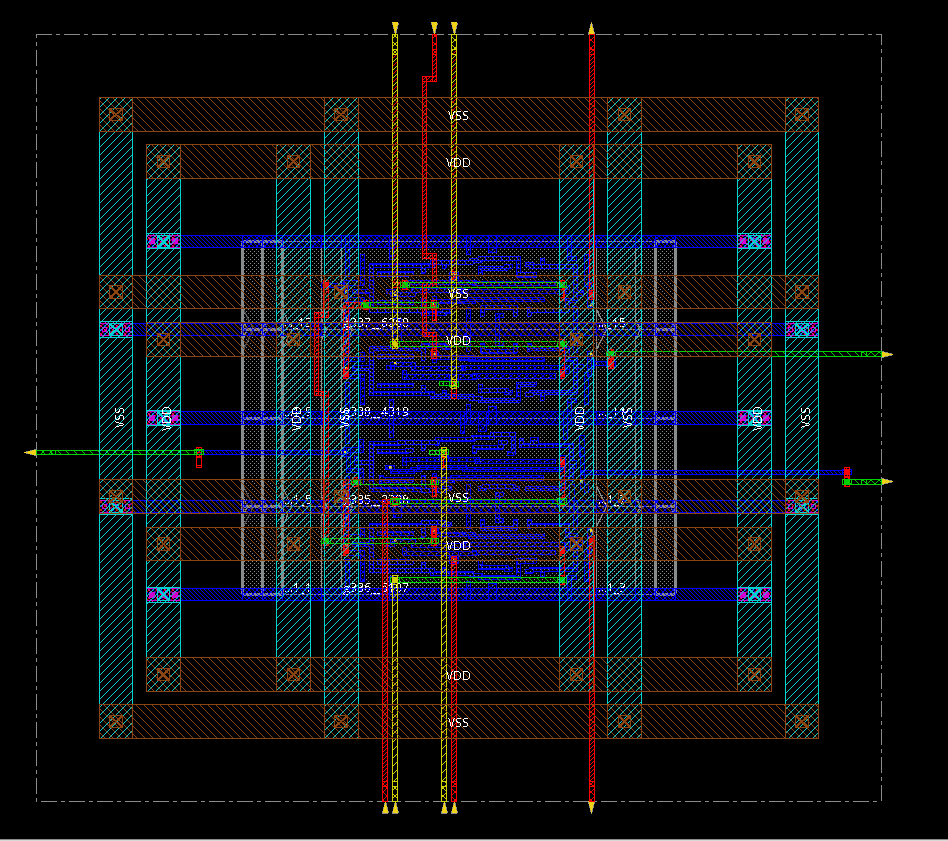
**FLOORPLANNING :**

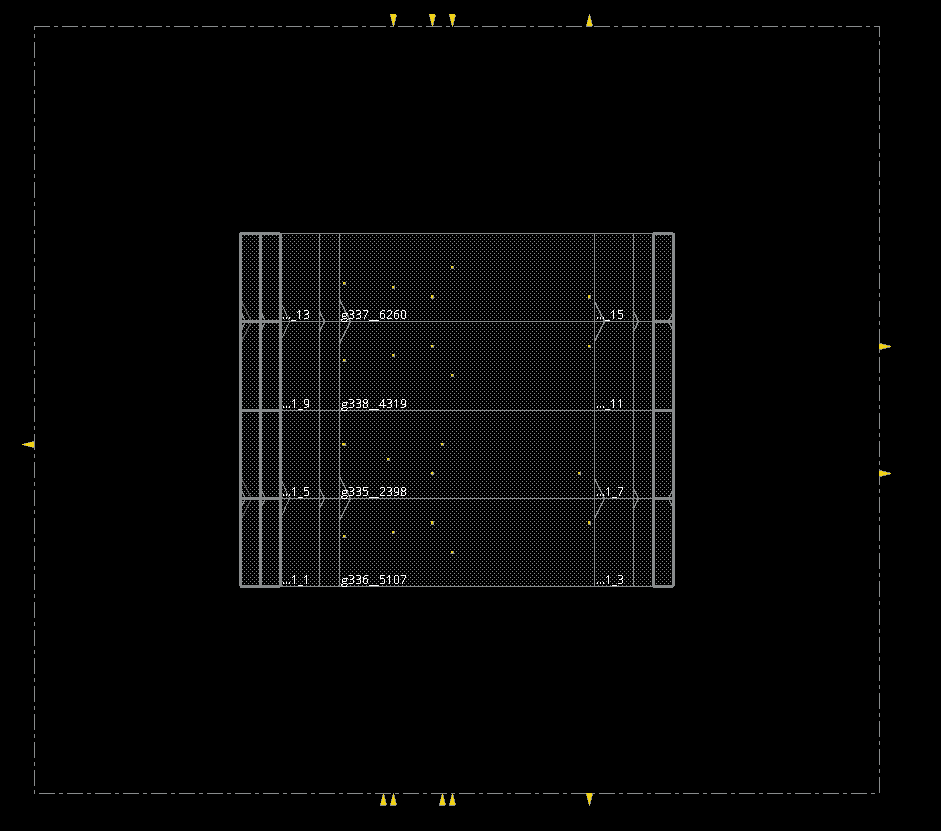


**POWERPLANNING :**

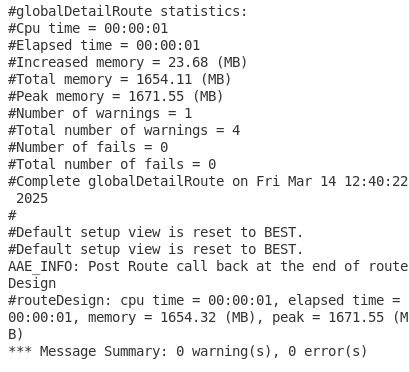


**PLACEMENT :**





**ROUTING :**



**DRC :**

