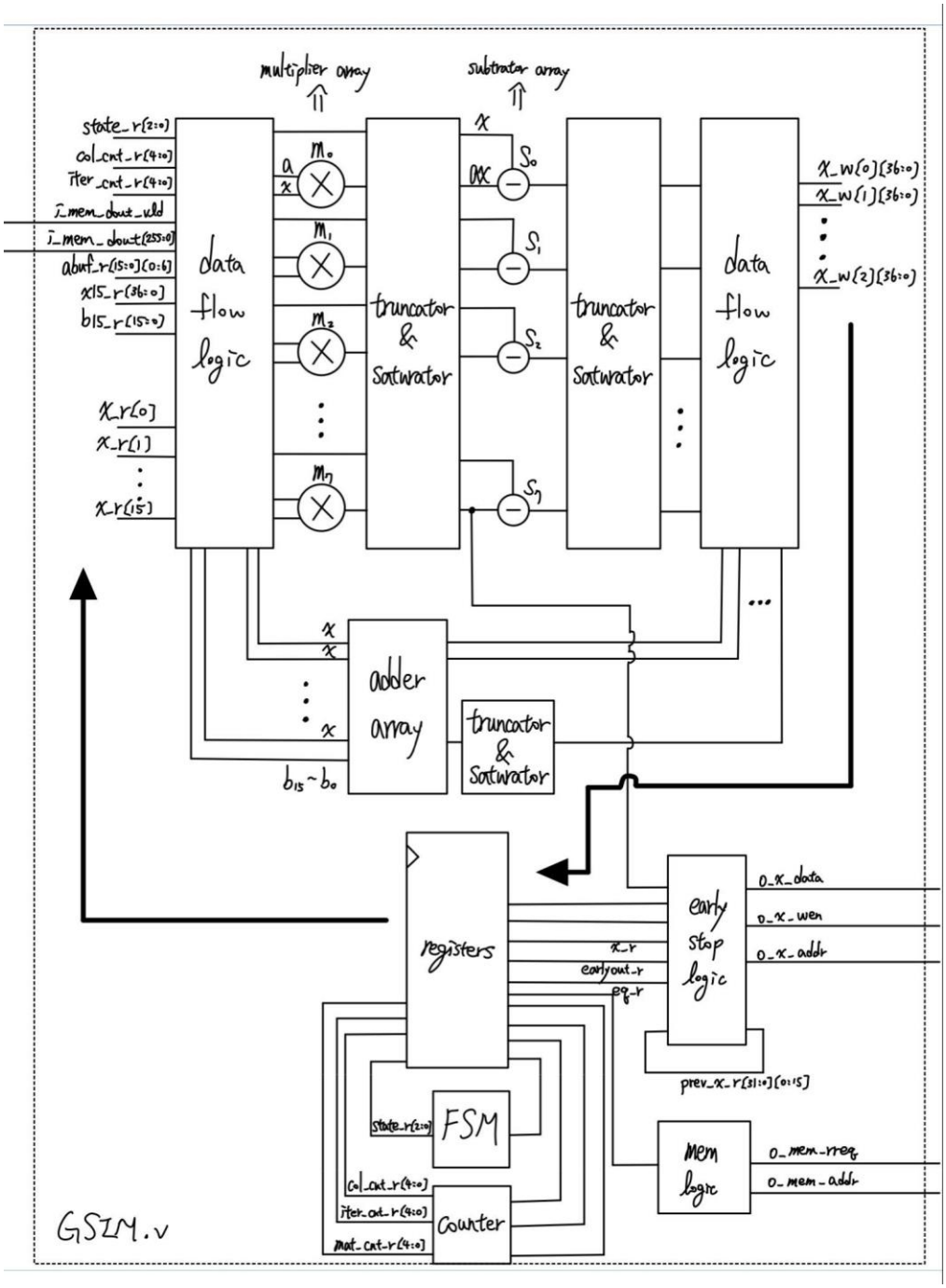
Team11 Report

b07901098 張惇宥
b07901010 范詠為

1. 架構設計
   a. Block Diagram



   i. data flow logic

處理哪筆資料要進入哪個 multipiler，和還有哪筆資料要去更新 x_w 的邏輯。這部分當時在設計時沒有經過仔細考慮，造成後續一些資源浪費，這部分在優化部分會說明。

ii. truncator & saturator

負責做 truncation 和 saturation，基本上是一些 mux 判斷有沒有 overflow。主要有三個地方，分別是每次乘法的輸出、加最後一項後和乘以最後的 1/a 後。

iii. early stop logic

優化時加入的邏輯，我們記了前一個 x_r 的值（prev_x_r），在更新 x_r 後如果沒變動，就記住相等，當16個 x_r 都相等時，就表示之後會維持穩定，即可提前輸出。這部分在優化部分也會說明。

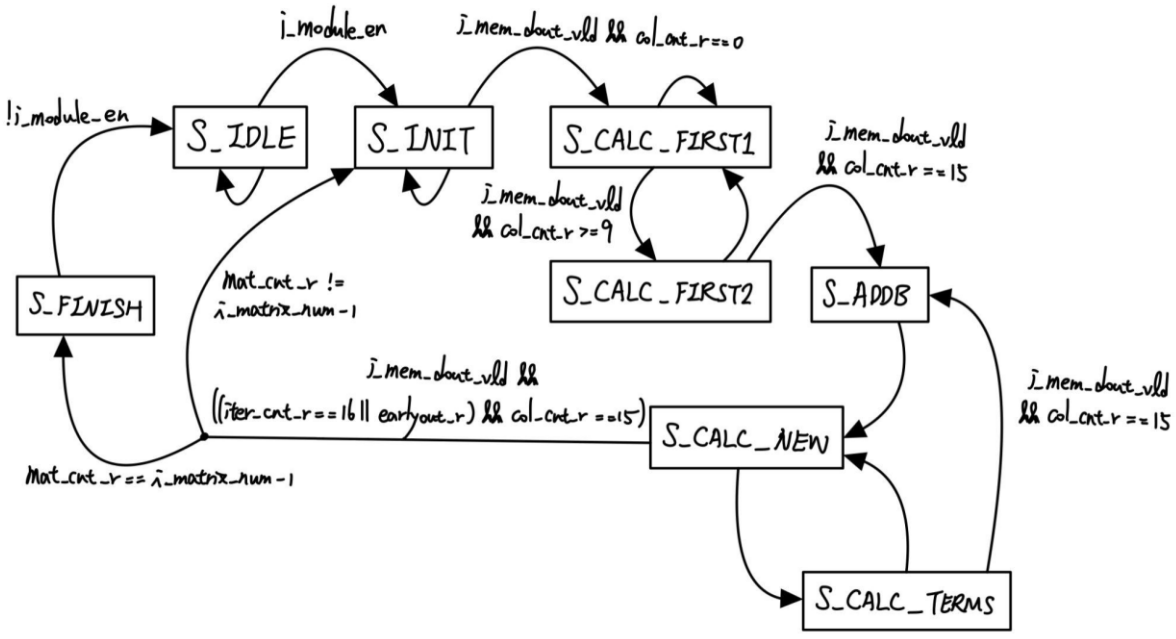
iv. mem logic

用當前更新後的 counter 去給 matrix memory 位址，我們將 request 一直拉成1，因為每個 cycle 都要跟其拿資料。（但這部分因為 memory 50% idle 的特性會造成一些浪費，在優化部分會討論）

v. counter

主要有三個 counter，col_cnt_r 紀錄現在執行到的 column、iter_cnt_r 紀錄做了幾個 iteration、mat_cnt_r 紀錄做到第幾個 matrix。

b. FSM



i. S_IDLE

在 i_module_en 還未拉高前的 state。

ii. S_INIT

初始化 x_r[0]~x_r[15]，先將 b0~b15 放入 x_r，再花十六個 cycle 分別讀取 1/a 並與 x_r[i] 相乘。

iii. S_CALC_FIRST1 & S_CALC_FIRST2

初始化完後，開始算第一個 iteration，因為這個 iteration 較為特別，一定要獨立做，且我們只使用8個 multipliers，因此分兩個 state 做。詳見 d. 部分。

iv. S_ADDB

每做一個 iteration，會有一個 cycle 用來更新 x_r，也就是將每個 x_r[i] 加上 bi，因為我們最後改成沒有記 b_r 的版本，b0~b15 這邊要讀進來。

v. S_CALC_NEW

這部分使用一個 multiplier 更新一個 x_r 的值，將之乘以從 matrix memory 讀出來的 1/a，另外七個 multipliers 將其他在 S_CALC_TERMS 因為只有八個成法器而沒做完的剩下七個減乘做完，使用的是 buffer 住的 a。

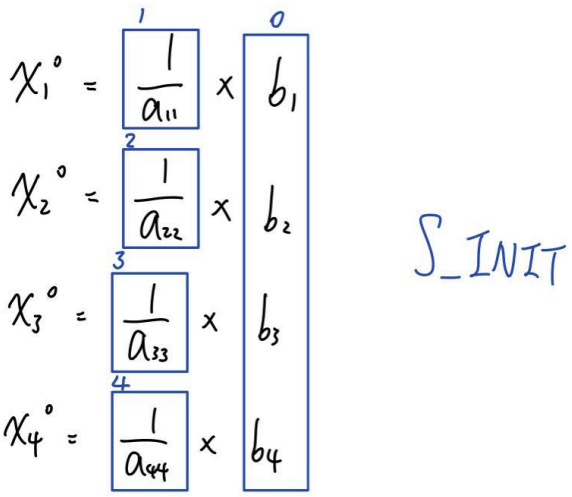
vi. S_CALC_TERMS

基本上與 S_CALC_NEW 互相來回。這個 state 平行用八個 multipliers 乘上從 matrix memory 讀出來的 a 的 row，因 data 存放的特性，用上這條 row 的 a 算的乘法可以平行。然而因為只有八個乘法器，剩下七個要減乘的運算要把 a buffer 住，並等下一個 cycle 也就是進入 S_CALC_NEW 時算。

vii. S_FINISH

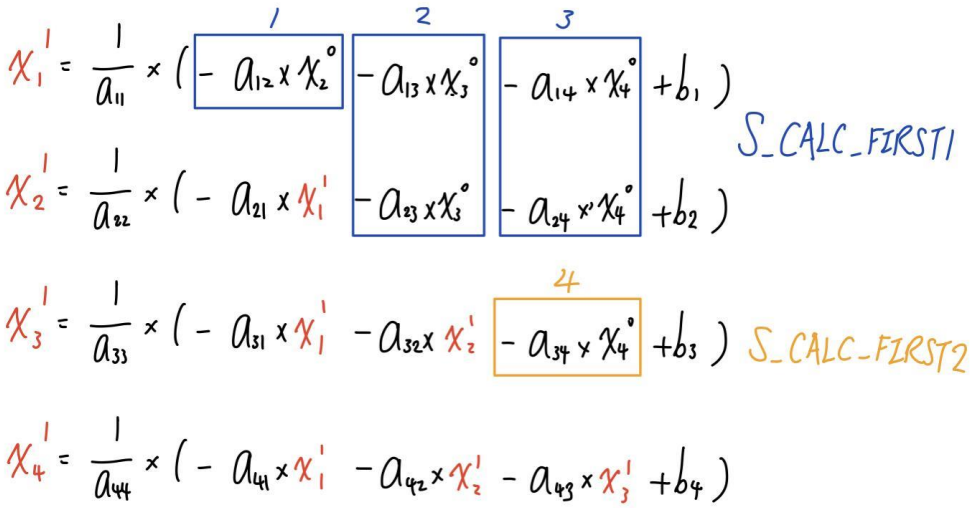
當 iter_cnt_r 為16時，表示已經做完一個 Ax=b，這時會從 S_CALC_NEW 跳回 S_INIT 用下一個矩陣重新初始化，而當做完最後一個 Ax=b 後，即進入 S_FINISH，並拉高 o_prod_done，待 i_module_en 拉下來時回到 S_IDLE，並拉低 o_prod_done。

c. Scheduling & Allocation - S_INIT



以4x4矩陣為例，初始化先將 b1~4 讀進 x_r，接著花四次讀入 1/aii 後與 x_r[i] 相乘，每次運算只使用一個乘法器。

d. Scheduling & Allocation - S_CALC_FIRST1, 2



以4x4矩陣為例，我們如果只用兩個乘法器，就必須如圖，當越減越多項時要分兩次才能完成。同理16x16矩陣，用八個乘法器也是如此。

e. Scheduling & Allocation - S_ADDB

$$x_1' = \frac{1}{a_{11}} \times (-a_{12} \times x_2^{\circ} - a_{13} \times x_3^{\circ} - a_{14} \times x_4^{\circ} + b_1)$$

$$x_2' = \frac{1}{a_{22}} \times (-a_{21} \times x_1' - a_{23} \times x_3^{\circ} - a_{24} \times x_4^{\circ} + b_2)$$

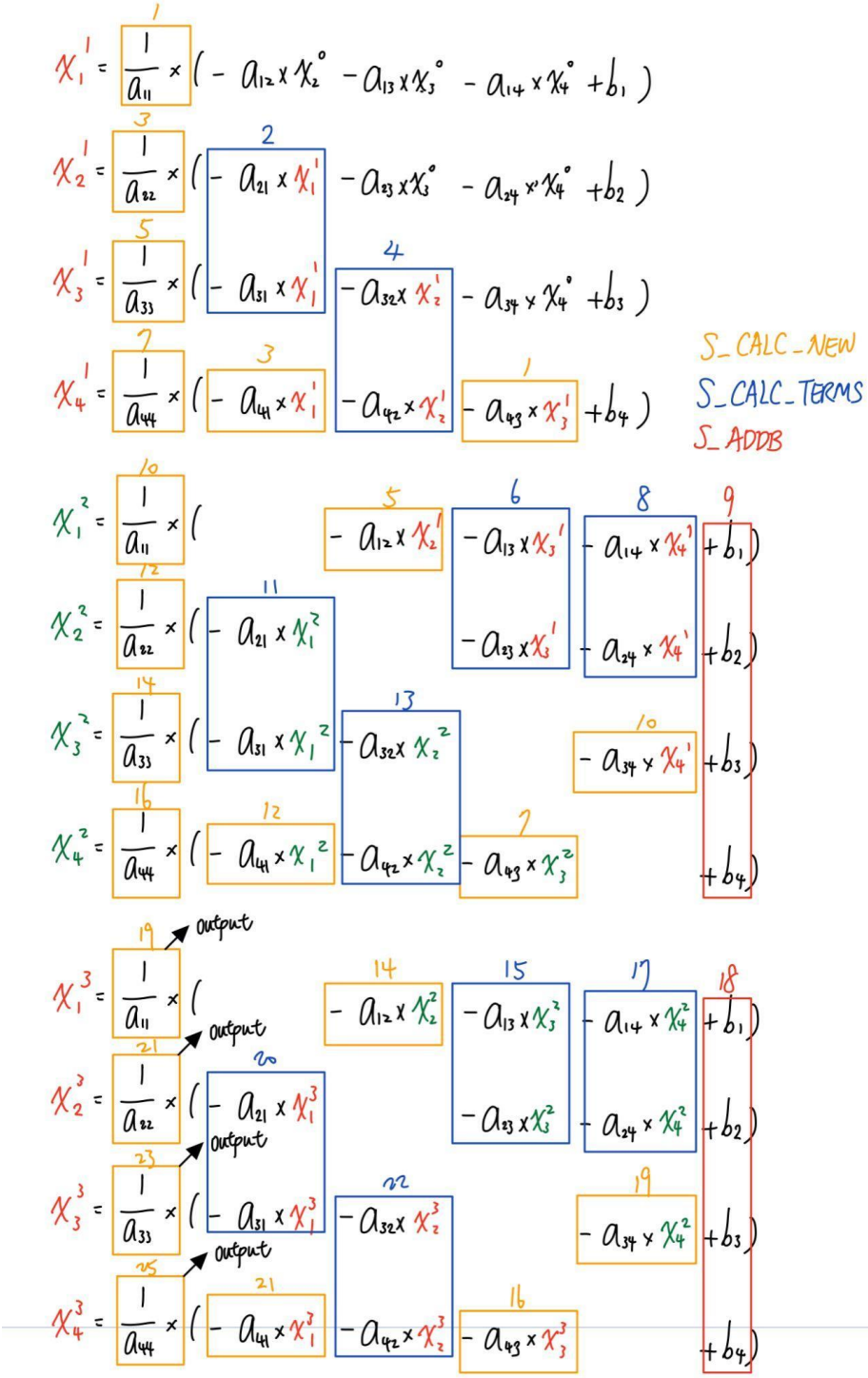
$$x_3' = \frac{1}{a_{33}} \times (-a_{31} \times x_1' - a_{32} \times x_2' - a_{34} \times x_4^{\circ} + b_3)$$

$$x_4' = \frac{1}{a_{44}} \times (-a_{41} \times x_1' - a_{42} \times x_2' - a_{43} \times x_3' + b_4)$$

S_ADDB

以4x4矩陣為例，因為 b1~4 可以一次讀進來，我們用四個加法器一次加完。同理16x16矩陣，用十六加法器也是如此，但會顯得有點浪費，這部分優化部分會探討。

f. Scheduling & Allocation - S_CALC_NEW, TERMS



以4x4矩陣，共三次 iteration 為例，每次的操作如上圖所示，如 FSM 部分所述，橘色部分更新最終 x_r[i] 值，其他乘法器拿來做上一個 cycle 沒做完的 buffer 住 a 的乘法。而藍色部分則是用全部的乘法器做運算，並 buffer 住剩下的 a 待下次橘色部分時算。有一個需要注意的地方是，因為像是在上圖中第8個 cycle，算完後要進入 S_ADDB，這時 x4 會被更新，導致在進入10時用的 x4 會不是原本的，而且當之後reset 完 x4 後，原本 x4 有加上的 b4 會被歸零。基於上述兩個原因，在16x16 下，我們多記了 x15 和 b15。

2. 硬體優化方法

a. parallel processing

最初的想法是，不用平行運算的話，雖然只使用一個乘法器和減法器，但時間會被拉很長，要差不多256個 cycle 才做得完。但因為 data 在 matrix 擺放位置的關係，沒辦法在同一個 x_r[i] 內用內積的維度做平行，所以最後選擇在不同 x_r[i] 間做平行如上面 scheduling & allocation 的圖所示。但最原本使用了15個乘法器而非八個。

b. break boundaries between iterations

接著因為下三角要等 x_r 更新後的值才能算，平行只能平行在上三角，但這樣一來乘法器的 utilization rate 會降低，因為會慢慢的越用越少。因此我們就想到說可以將不同 iteration 之間的邊界打破，這樣除了第一個 iteration 之外，剩下的就可以都用滿乘法器。

c. dump output right after computation finish

在輸出的時候，為了節省時間，我們不是將全部運算算完然後一次用16個 cycle 輸出16筆，而是在某 x_r[i] 更新完16次後馬上輸出，這部分可以省下 16*matrix_num 左右的 cycle 數，儘管並不多。

d. folding (16x to 8x, increase utilization rate)

但即使將不同 iteration 並在一起，即 b. 部分的方式，因為不想讓一個 cycle 內的 critical 是兩個乘法運算，乘以 1/a 這一步勢必要分開來算，也就是多花一個 cycle，造成會有一個 cycle 用滿乘法器沒錯，但另一個 cycle 只用一個乘法器，這樣乘法器的 utilization rate 仍舊只有 50% 左右，於是我們使用 folding 的技巧，cycle 數不變的條件下同時將乘法器變為原來的一半，也就是八個，而將沒算完的另外七個運算 buffer 到下一個 cycle 跟乘以 1/a 的這個操作一起算，這樣乘法器的 utilization rate 就可以提升至近 100%。代價是花了一些 buffer 的面積（半條a的大小），但因為新架構我們同時拿掉了原本有存的 b_r，改成用讀的方式，這部分反而 register 數也變小。但可惜的是在合成時發現總面積與十六個乘法器的初版差不多，推測原因是 data flow logic 太複雜，這部分也寫在後面的 unfinished optimization part。

e. early stop

因為此演算法收斂速度快，所以我們如果發現前後兩個iteration如果數值都相同的話，就可以停止運算，不一定要做到16個iterations，代價是一條 prev_x_r，但是是完全值得的，因為收斂的速度足夠快。

實際結果:

我們可以發現到，在一般的matrix之中大概會有26%的進步，在sparse martix的進步更加明顯，高達40%，但是如果發生overflow就不會有甚麼進步。

testbench	with early stop	without early stop	improvement
tb0	12648ns	17122ns	26%
tb1	19858ns	33189ns	40%
tb2	8523ns	8625ns	1%
tb3	24891ns	33291ns	25%
tb4	2474ns	3201ns	23%

f. other unfinished optimization thought

i. pipeline between multiplier / subtractor and saturator

因為這次combination的比例很高，而且現在是一個cycle之中就做乘，saturate和減，如果可以做成pipeline架構，能提高throughput，讓AT值的成績更好。(因為如果是用parallel，即便我們能讓timing進步一倍，但面積也一樣增加一倍，所以整體來看效果似乎不顯著)

ii. input register

因為本次的input有1ns的delay，因此如果加上input registers可以節省這1ns，而且在合成以及apr時的critical path也發生在input2reg，因此如果加上之後會有可預期的效果。

iii. use only 4x and buffer data to reduce the impact of 50% memory idle issue

因為這次final的memory有50%的機率才能讀的到，因此理論上應該要增加每一筆的資料運算的cycle，以減少因為memory造成之idle的比例來克服這個問題，所以說我們有想說如果只開四個乘法器，這樣就能讓每筆data運算的cycle提高到四個，減少random產生的影響。

iv. clock gating

因為在我們的架構底下，combinational circuit的比例遠大於sequential，所以clock的效果不明顯，還可能因為fanout導致timing問題，而且增加判斷的邏輯電路，因此最後不採用clock gating。

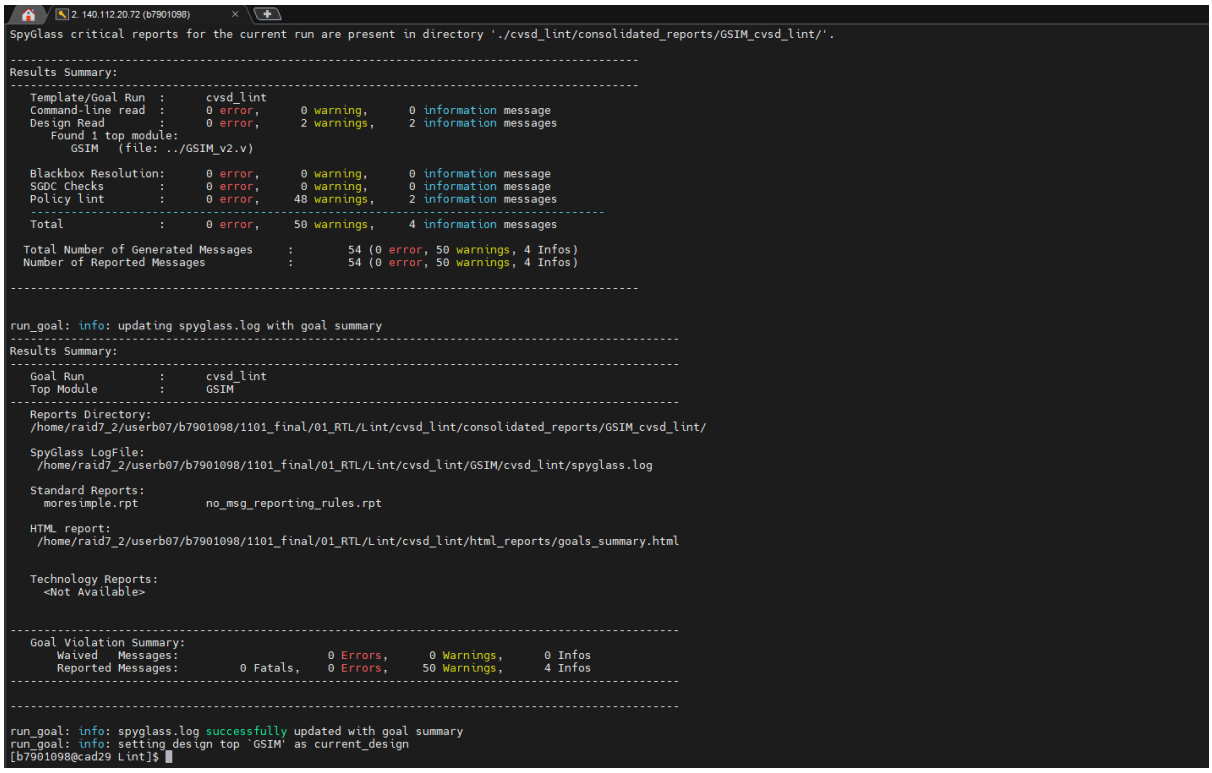
v. reuse subtractor for replacing adder array

因為一個加法可以視為減負的一個數，因此如果有把b存在reg當中，就可以在initial結束之後，逐步地將b改成它的2's complement，如此一來就可以讓減法器reuse，不需要多開加法器。

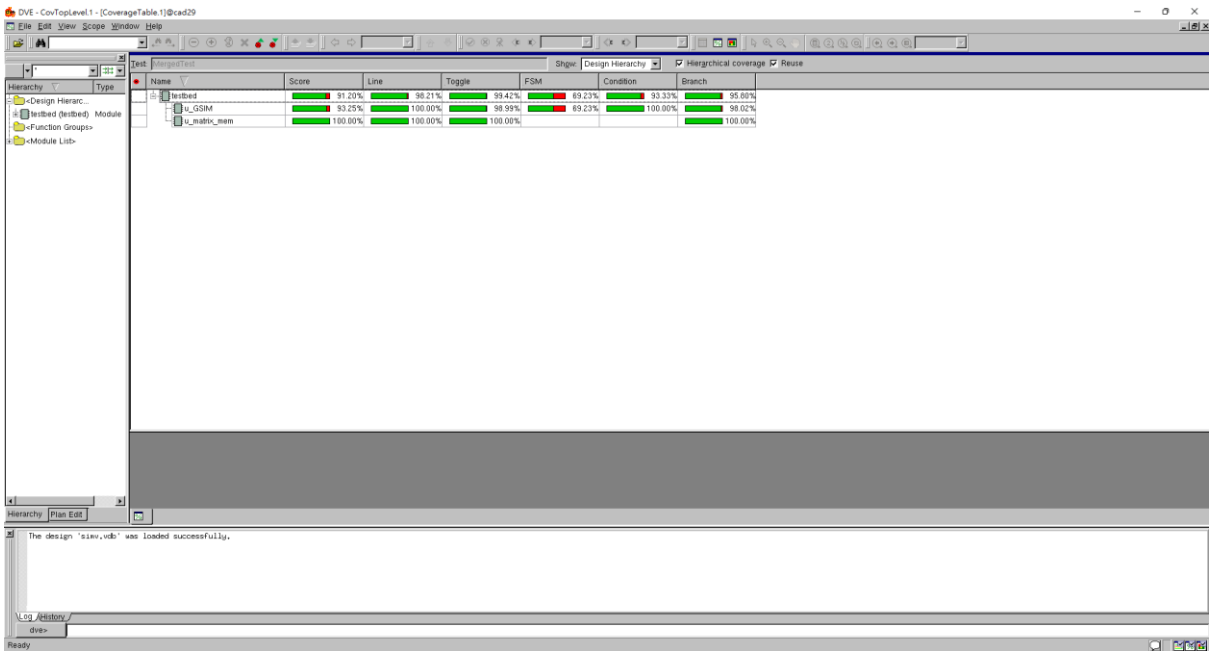
vi. simplify data flow logic

我們在第二個版版本之中，雖然乘法器的數量由16個減到8個，然而面積卻沒有顯著的下降，原因有可能是出在我們的控制器有點複雜，可能可以思考要如何簡化控制器。

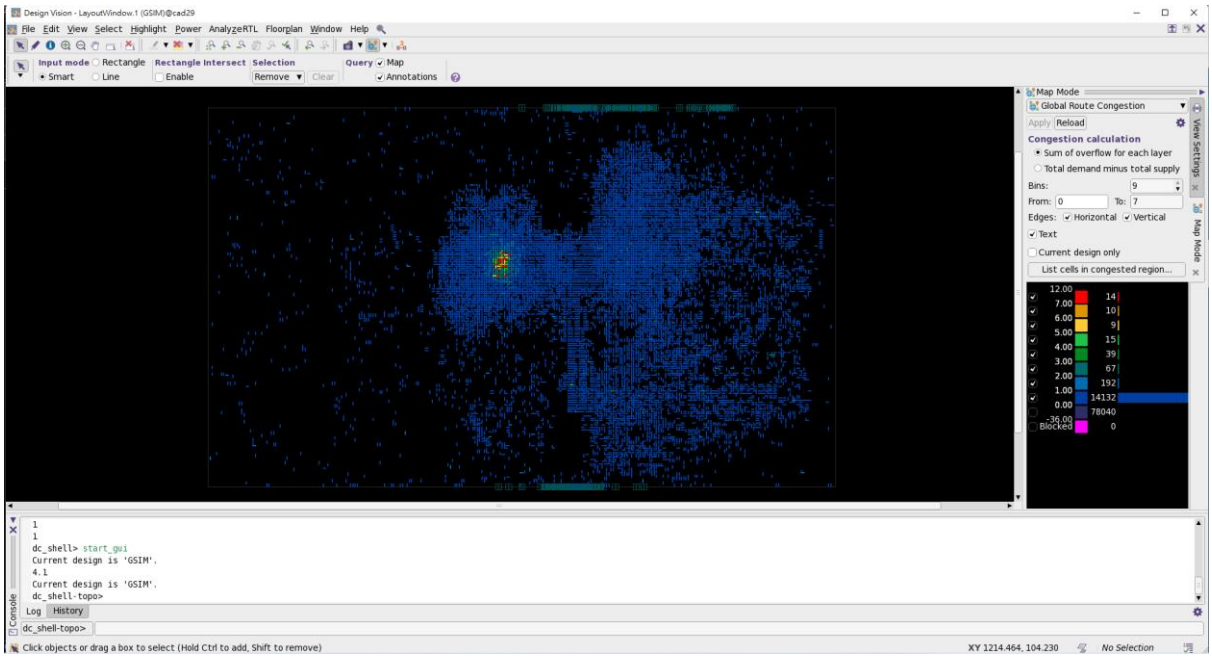
3. nLint report with 0 errors



4. Coverage result



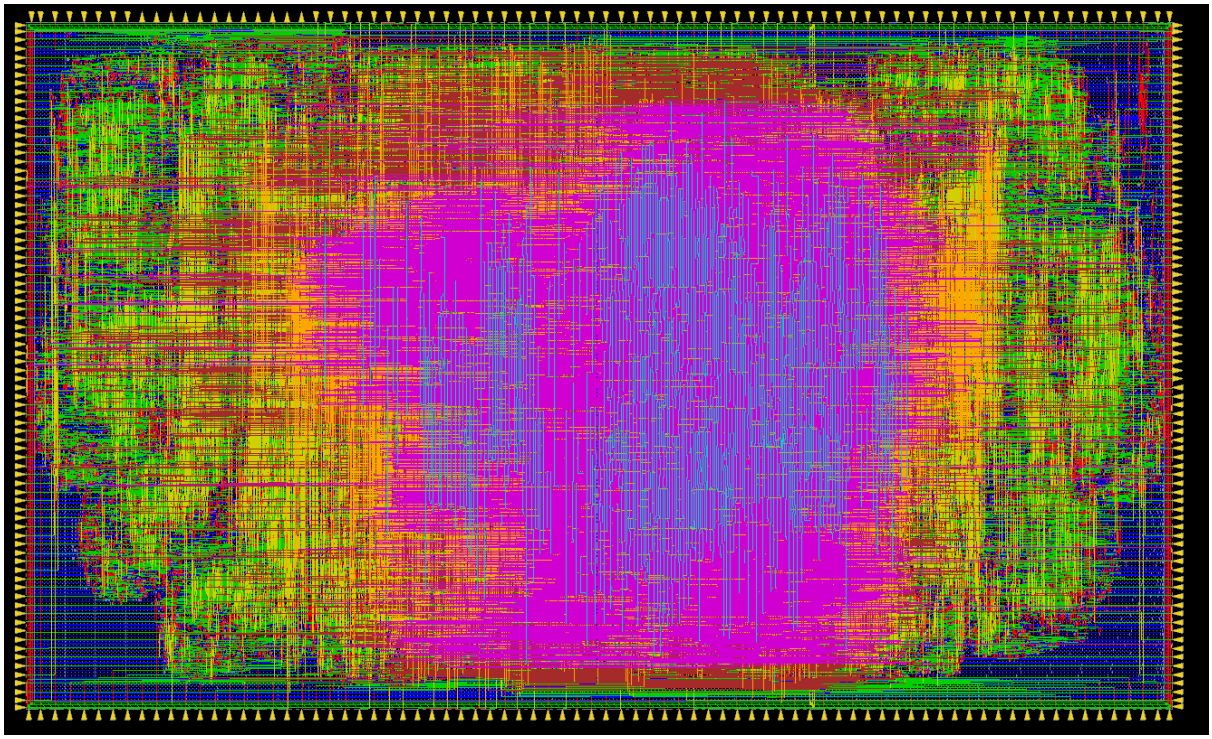
5. Congestion map



6. Prime time power report (Gate-level)

```
(A) Idle power:                0.002393
(B) Idle_after_active power: 0.002373
(C) Active power:              0.006737
***** Assume (A) & (B) time window are the same *****
(A) & (B) diff ratio:          0.84%
=====================================================
========== Power diff ratio result PASS!!! ==========
=====================================================
```

7. Layout



8. Performance 表格

Physical category		
Design Stage	Description	Value
Gate-level Simulation	Cycle time for Gate-level Simulation (ex. 10ns)	9.5ns
P&R	Number of DRC violation (ex: 0) (Verify -> Verify Geometry…)	0
	Number of LVS violation (ex: 0) (Verify -> Verify Connectivity…)	0
	Core area (um^2)	709214
	Die area (um^2)	738150
Post-layout Simulation	Cycle time for Post-layout Simulation (ex. 10ns)	9.5ns