Computer-Aided VLSI System Design

Final Report

組別: Group 34
組員: R10943018 楊欣哲
R10943117 陳昱仁

Outline

1. 架構設計

1.1 題目敘述

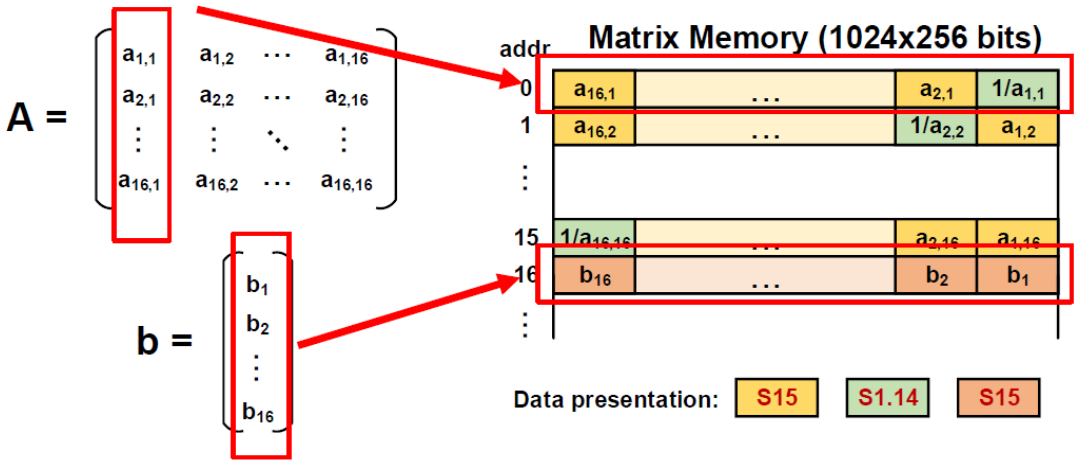
本次 CVSD Final Project 的題目為實作 Gauss-Seidel Iteration Machine。其目標是想求多元線性聯立方程式的解[x_1, ..., x_N]，如下圖所示。

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1N}x_N &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2N}x_N &= b_2 \\ &\vdots \\ a_{N1}x_1 + a_{N2}x_2 + \cdots + a_{NN}x_N &= b_N \end{aligned}$$

其具體解法是將下圖中的式子疊代數次，可將所有待求的 x 收斂在某個值，此值即為所求。在此次 Final Project 中的疊代次數為 16。

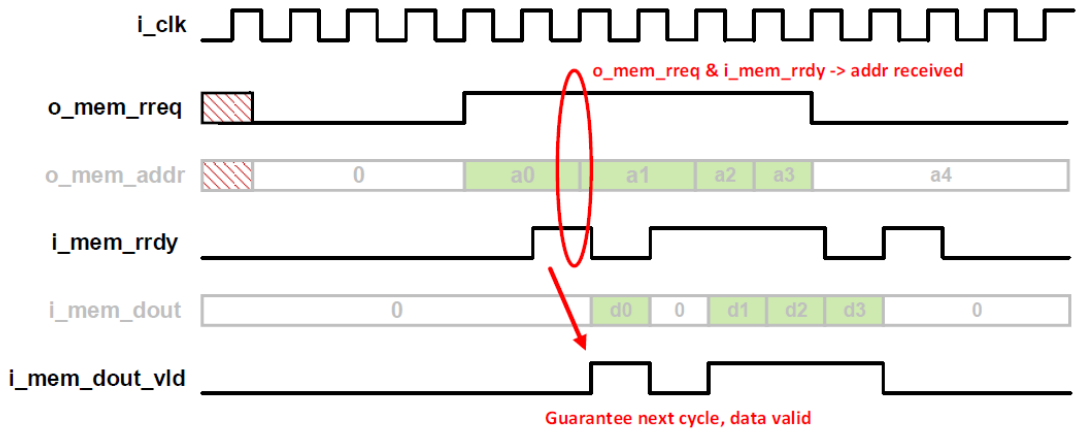
$$x_i^{k+1} = \frac{1}{a_{ii}}\left[b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k+1} - \sum_{j=i+1}^{N} a_{ij}x_j^{k} \right]$$

A Matrix 和 B Vector 的儲存方式如下圖，皆是將矩陣順時針旋轉 90 度，儲存在 Memory 中。GSIM.v 一次可以跟 Memory 索取一列資料，也就是 data bandwidth 為 256bits，或者 16 個 words。

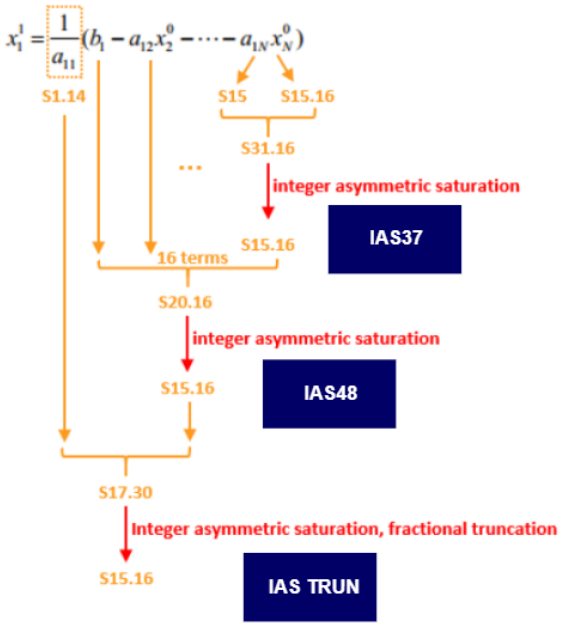


需要特別注意的是 GSIM.v 跟 Memory 索取資料，並非馬上能夠拿到。必須透過 handshake 機制，等 1bit 訊號 i_mem_rrdy 為 1 時，在下一個 cycle 方能得到正確的值，如下圖所示。在此次 Final Project 中以 50%機率為 1，50%機率為 0 的 random 訊號來生成 i_mem_rrdy 訊號。

Handshake



下圖為 x 面對 overflow 和 truncation 的機制。我們分別將不同的 overflow 和 truncation 機制命名為 IAS37，IAS48，IAS TRUN(對照 1.3)。



1.2 Dataflow

在 1.2 section 的圖中，橘色代表 row index，紫色代表 column index，以下 column 和 row 是依照 memory 的擺放為準。

1.2.1 Column-wise Dataflow

$$x_i^{k+1} = \frac{1}{a_{ii}}[b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k+1} - \sum_{j=i+1}^{N} a_{ij}x_j^k]$$

根據上式，可以衍生出 column-wise dataflow。Column-wise dataflow 依序更新 x_i^k, x_{i+1}^k(k 代表 iteration，i 代表 x 的 index)，如下圖所示。藍色代 column-wise dataflow 進行 a*x 運算，黃色代表進行倒數乘法運算($\frac{1}{a_{ii}}$)。數字代表運算的先後順序，數字相同代表同時運算。

	1	0
0	32	16,17
1	33,34	1
2	18	2
3	19	3
4	20	4
5	21	5
6	22	6
7	23	7
8	24	8
9	25	9
10	26	10
11	27	11
12	28	12
13	29	13
14	30	14
15	31	15

由於 data 的存取是橫向，採用 Column-wise Dataflow 會在 latency 有較差的表現，詳見 2.2。

1.2.2 Row-wise Dataflow

$D\vec{x}^{k+1} = \vec{b} - L\vec{x}^{k+1} - U\vec{x}^{k}$.

此形式的導出 [隱藏]

如上形式來自於高斯 - 賽德爾疊代的元素公式: 對於第m個未知量$(\vec{x}^{k+1})_m = x_m^{k+1}$, 我們可以得出

$$(D\vec{x}^{k+1})_m = (\vec{b})_m - (L\vec{x}^{k+1})_m - (U\vec{x}^{k})_m \Rightarrow a_{mm}x_m^{k+1} = b_m - \sum_{j=1}^{n}(L)_{m,j}x_j^{k+1} - \sum_{j=1}^{n}(U)_{m,j}x_j^k$$

已知$a_{mm} \neq 0$, $(L)_{m,j} = 0\,(\forall j \geq m)$ 以及 $(U)_{m,j} = 0\,(\forall j \leq m)$, 因此可以得出

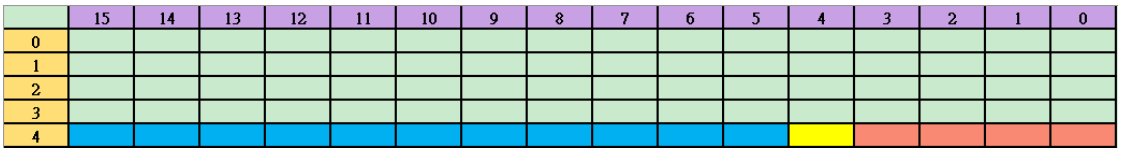
$$x_m^{k+1} = \frac{1}{a_{mm}}\left(b_m - \sum_{j=1}^{m-1}(L)_{m,j}x_j^{k+1} - \sum_{j=m+1}^{n}(U)_{m,j}x_j^k\right) = \frac{1}{a_{mm}}\left(b_m - \sum_{j=1}^{m-1}a_{mj}x_j^{k+1} - \sum_{j=m+1}^{n}a_{mj}x_j^k\right)$$

.

根據上圖的推導可以將$x_i^{k+1} = \frac{1}{a_{ii}}[b_i - \sum_{j=1}^{i-1}a_{ij}x_j^{k+1} - \sum_{j=i+1}^{N}a_{ij}x_j^k]$ 改寫成(1)式。其中 D 代表 A 的對角矩陣，L 代表 A 的下三角矩陣，U 代表 A 的上三角矩陣，但由於 A 矩陣旋轉 90 度(詳見 1.1)，因此 U 在 1.2section 的圖中會顯示在右下角，L 會顯示在左上角，特此說明。

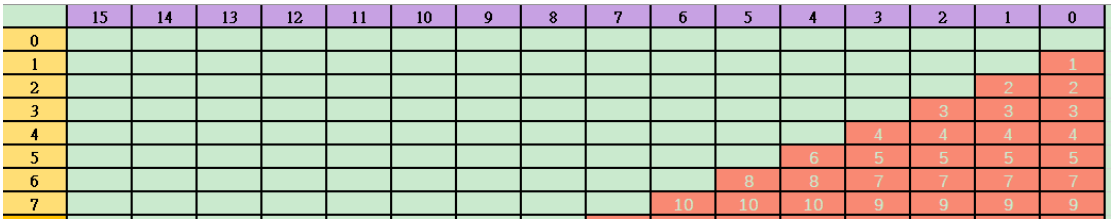
$$\begin{cases} \mathrm{D}\vec{x}^{k+1} = \vec{b} - \mathrm{L}\vec{x}^{k+1} - \mathrm{U}\vec{x}^{k} & (1) \\ \mathrm{D}\vec{x}^{k+2} = \vec{b} - \mathrm{L}\vec{x}^{k+2} - \mathrm{U}\vec{x}^{k+1} & (2) \end{cases}$$

觀察(1)式和(2)式可以發現(1)式中的L$\vec{x}^{k+1}$和(2)式中的 U$\vec{x}^{k+1}$ 共用相同的$\vec{x}^{k+1}$。如下圖所示，皮膚色帶表(2)式中的 U$\vec{x}^{k+1}$，黃色代表(1)式中的D$\vec{x}^{k+1}$倒數部分，藍色代表(1)式中的L$\vec{x}^{k+1}$。

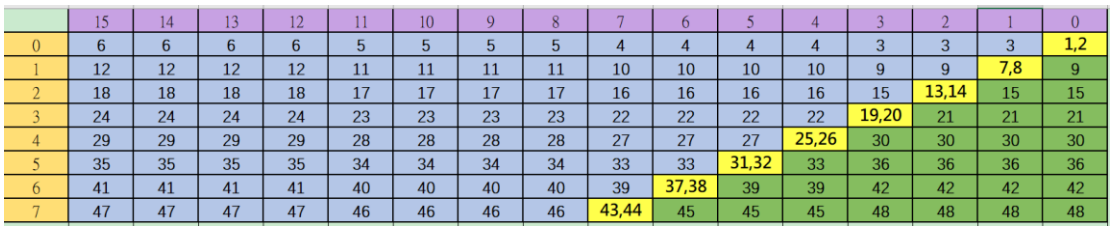


根據以上推導，我們發展出 Row-wise Dataflow，也就是橫的方向來運算。由於 data 的存取也是橫向，採用 Row-wise Dataflow 相比 Column-wise Dataflow 在 latency 會減少一半，詳見 2.2。因為 Row-wise Dataflow 在沒有額外 overhead 的情況下，能在 latency 減少一半，因此我們決定採取 Row-wise Dataflow。

以下是 Row-wise Dataflow 的詳細過程。Row-wise Dataflow 依序執行 top-triangle phase，row-wise phase，bottom-triangle phase。Top-triangle phase 只會在第一個 iteration 執行，dataflow 如下圖演示。這邊以 4 顆乘法器為例，數字代表運算的先後順序，數字相同代表同時運算。
皮膚色代表 a∗x 運算，對應到(1)式為 k = 0 的 U$\vec{x}^{k}$。



Top-triangle phase 結束後轉至 Row-wise phase，會執行 15 iteration。dataflow 如下圖演示。這邊以 4 顆乘法器為例，數字代表運算的先後順序，數字相同代表同時運算。淺藍色代表 a∗x 運算，對應到(1)式的 L$\vec{x}^{k+1}$。黃色代表倒數運算，對應到(1)式的D$\vec{x}^{k+1}$。綠色代表 a∗x 運算，對應到(2)式的U$\vec{x}^{k+1}$。

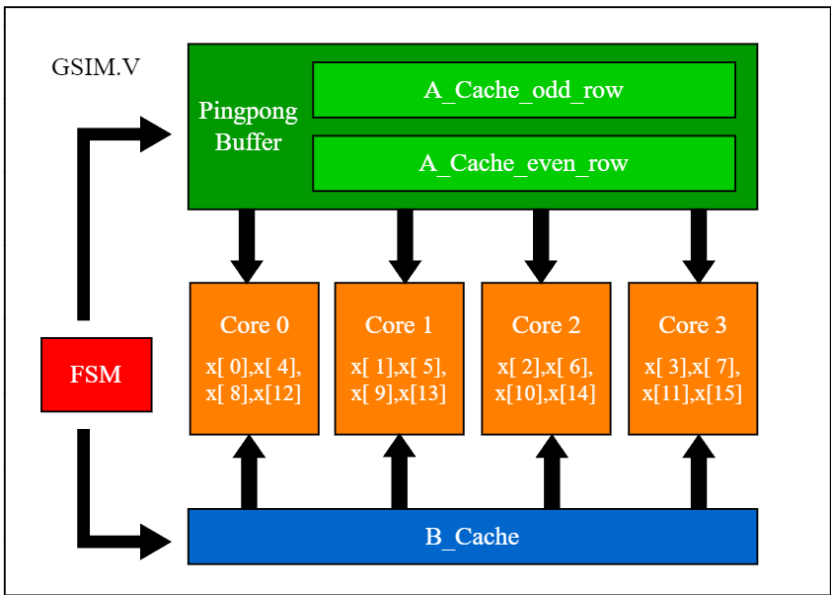


Row-wise phase 結束後轉至 bottom-triangle phase，只會在最後的 iteration 執行。dataflow 如下圖演示。這邊以 4 顆乘法器為例，數字代表運算的先後順序，數字相同代表同時運算。黃色代表進行倒數運算，對應到(1)式為 k = 16 的D$\vec{x}^{16}$。藍色代表 a∗x 運算，對應到(1)式為 k = 16 的L$\vec{x}^{16}$。

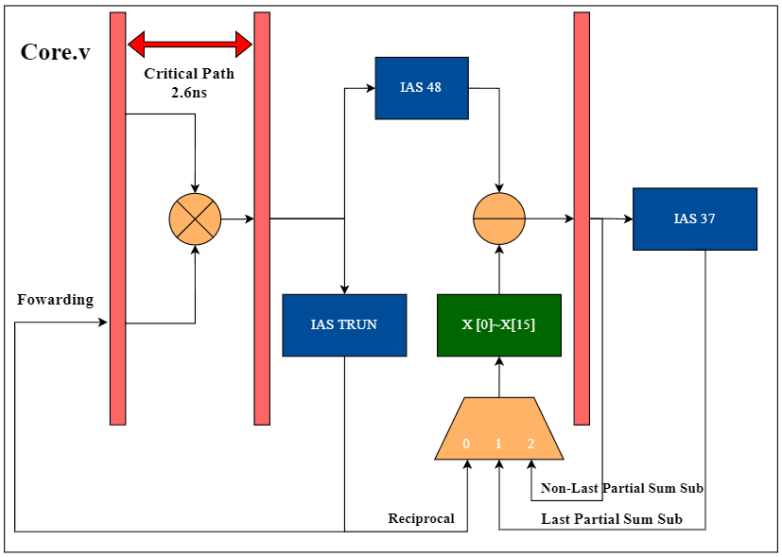
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	6	6	6	6	5	5	5	5	4	4	4	4	3	3	3	1,2
1	12	12	12	12	11	11	11	11	10	10	10	10	9	9	7,8	
2	18	18	18	18	17	17	17	17	16	16	16	16	15	13,14		
3	23	23	23	23	22	22	22	22	21	21	21	21	19,20			
4	28	28	28	28	27	27	27	27	26	26	26	24,25				
5	33	33	33	33	32	32	32	32	31	31	29,30					
6	38	38	38	38	37	37	37	37	36	34,35						
7	42	42	42	42	41	41	41	41	39,40							

1.3 架構圖

下圖是我們的整體架構圖。



總共有 4 個 core，每個 core 包含一個乘法與減法以及 truncation module。選擇 4 個 core 的理由是可以最小化 AT，詳見 2.3。每個 core 固定更新 4 個 x。A cache 是儲存 A 矩陣的 buffer。我們選擇實作兩個 256 bits ping-pong buffer，理由是可以降低 latency，詳見 2.2。另外還有一個 256 bits B_cache，負責儲存 B vector。



2. 硬體優化方法

2.1 Critical Path Optimization – Pipeline

使用 Pipeline 技巧，使得 critical path 縮減至 2.6ns。面對 data dependent hazard，使用 forwarding 使運算倒數時減少 50% bubble。

使用 pipeline 技巧的好處可使 critical path 縮短，進而縮短 cycle time。使用 pipeline 技巧帶來的 overhead 為當前後運算有 dependency 時，會產生 hazard。常用來解決 hazard 的方法有兩種。一種是 insert bubble，但會使系統 utilization 降低，且 pipeline 的級數越多，bubble 就越多，嚴重影響 performance。另一種是 forwarding，但需要較複雜的控制，且可能會使 critical path 變長。

在此次 Final Project 的架構(詳見 1.3)中，critical path 為 core.v 中的對 x 更新的部分，包括一個乘法，一個減法，以及 overflow 和 truncation 的判斷電路。使用 pipeline，加上合成時 retiming，可使得 Critical path 縮減至 2.6ns。

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	6	6	6	6	5	5	5	5	4	4	4	4	3	3	3	1,2
1	12	12	12	12	11	11	11	11	10	10	10	10	9	9	7,8	9
2	18	18	18	18	17	17	17	17	16	16	16	16	15	13,14	15	15
3	24	24	24	24	23	23	23	23	22	22	22	22	19,20	21	21	21
4	29	29	29	29	28	28	28	28	27	27	27	25,26	30	30	30	30
5	35	35	35	35	34	34	34	34	33	33	31,32	33	36	36	36	36
6	41	41	41	41	40	40	40	40	39	37,38	39	39	42	42	42	42
7	47	47	47	47	46	46	46	46	43,44	45	45	45	48	48	48	48

如上圖所示，黃色為倒數運算，綠色和淺藍色為乘減運算。在此次 Final Project 的 dataflow 中，乘倒數的運算(黃色)與之後的乘減運算(淺藍色、綠色)有 dependency，因此會產生 hazard。如果使用 insert bubble 的方法，會產生 2cycle 的 bubble。

面對 hazard，我們使用 forwarding 的技巧，在不讓 critical path 變長的前提下，使得原本會產生 2cycle 的 bubble 減少為 1 個，有效提升系統 utilization rate。

令$Cycle_{reciprocal\ per\ row}$，$Cycle_{reciprocal\ per\ col}$包含每行列需運算一次倒數需要運算的 1 cycle，以及因為 pipeline 產生的 1 bubble cycle。

$$\boldsymbol{Cycle_{reciprocal\ per\ row}} = \boldsymbol{Cycle_{reciprocal\ per\ col}} = 2$$

2.2 Latency Optimization – Row-wise dataflow & Ping-Pong buffer

Dataflow 的選擇可以使 latency 減少 50%。而 buffer 的選擇可使 latency 再額外減少 25%。

在此次 Final Project 中，取得每行資料需要透過 handshake 機制，每 cycle 只有 50%能取得 valid data。令 E[hit]為取得一列正確資料的期望值 cycle 數。

$\mathbf{P(Miss)} = \mathbf{P(Hit)} = 0.5$

$\mathbf{E[hit]} = \sum_1^\infty P(Miss)^{i-1} * P(Hit)^1 * i$

$= \sum_1^\infty \frac{n}{2^n} = \frac{1}{2} + \frac{(1+1)}{2^2} + \frac{(1+2)}{2^3} \ldots = \left[\frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} \ldots\right] + \frac{1}{2}\left[\frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} \ldots\right]$

$= \sum_1^\infty \frac{1}{2^n} + \frac{1}{2} * \sum_1^\infty \frac{n}{2^n} = \frac{\frac{1}{2}}{1-\frac{1}{2}} + \frac{1}{2} * \sum_1^\infty \frac{n}{2^n} = 2$

作為對照組的 Column-wise dataflow，因為 data dependency 的緣故，一次 mem hit 只能取 16word 中的 1 word 來進行運算。這樣有兩大壞處，包括平行度上限為 1，以及 long latency。平行度上限的部分，由於每次 mem hit 只能拿到 1 word，且 E[hit] = 2 > T_{ax} = 1，令T_{ax}為運算一個 a* x需要的 cycle 數。因此在不把整個矩陣存下來的前提下，最多只能有一個乘法器保持高 utilization，因此最高平行度為 1。第二個之後的乘法器在 Column-wise dataflow 中 utilization 永遠是 0。平行度上限太小，會嚴重影響到 AT performance，詳見 2.3 latency 的分析。

令 E[cycle per column]為將 1 column 的資料完全運算完成需要的期望值 cycle

$\mathbf{E[cycle\ per\ column]_{Column-wise\ dataflow}}$

$= 16 * \mathrm{E[hit]} + Cycle_{reciprocal\ per\ col}$

$= 16 * 2 + 2 = 34$

對於 Column-wise dataflow，另一個選擇是將整個矩陣存下來。這樣可以將 E[hit]降為 1，但是需要付出龐大的面積為代價，估算約為 20 萬 um^2。因此不考慮此方案。

作為實驗組的 Row-wise dataflow 能有效運用一次 mem hit 中的所有資料-16 word 來進行運算。相較於 Column-wise dataflow，Row-wise dataflow 搭配 16word a-cache 可以享有平行度上限提高為 16，以及 shorter latency 的好處。提高平行度上限的好處詳見 2.3。

令 E[cycle per row]為將 1 row 的資料完全運算完成需要的期望值 cycle。n 為乘法器平行度，也就是乘法器個數。

$$\mathbf{E[cycle\ per\ row]_{Row-wise\ dataflow\ with\ a\ 16\ word\ cache}}$$
$$=1 * \mathrm{E[hit]} + Cycle_{a*x\ calulation\ per\ row} + Cycle_{reciprocal\ per\ row}$$
$$= 2 + \frac{16}{n} + 2 = \frac{16}{n} + 4$$

在 Row-wise dataflow 的基礎上，我們發現將一個 16word a-cache 換成 ping-pong buffer，也就是 two 16word a-cache，可以再進一步將低 latency。Ping-pong buffer 一個負責儲存 current row data，一個負責儲存 next row data， 在一行結束進入下一行前，如果 next row buffer 尚未 mem hit，則持續等待直到 next row buffer mem hit。這樣可以保證每行開始時 current row buffer 都必定有資料，可以直接開始進行運算。在運算的過程中，可以跟 memory 要 next row data。如果幸運在此行運算結束前取得 next row data，則可直接進入下行。否則如上所述，等待直到 next row buffer mem hit。令 E[cycle per row]為將 1 row 的資料完全運算完成需要的 cycle 的期望值。

$$\mathbf{E[cycle\ per\ row]_{Row-wise\ dataflow\ with\ ping-pong\ buffer}}$$
$$= P[Next\ row\ mem\ hit\ before\ finishing\ this\ row\ calulation\ ] * \left(Cycle_{a*x\ cal\ of\ this\ row} + Cycle_{reciprocal}\right) + P[Next\ row\ mem\ all\ miss\ before\ finishing\ this\ row\ calulation * \left(Cycle_{a*x\ cal\ of\ this\ row} + Cycle_{reciprocal} + Cycle_{Extra\ time\ next\ row\ hit}\right)$$
$$= \left[\frac{2^{\frac{16}{n}}-1}{2^{\frac{16}{n}}} * \left(\frac{16}{n}+2\right) + \frac{1}{2^{\frac{16}{n}}} * \left(\frac{16}{n}+2+\sum_1^\infty \frac{n}{2^n}\right)\right]$$
$$= \frac{16}{n} + 2 + \frac{1}{2^{\frac{16}{n}}-1}$$

Column-wise dataflow，Row – wise dataflow with a 16 word buffer，Row – wise dataflow with ping-pong buffer 三者運算一個 row 或 column 總 cycle 的期望值比較如下圖表。

n	1	2	4	8	16
Column-wise dataflow	34	x	x	x	x
Row-wise dataflow with a 16 word buffer	20	12	8	6	5
Row-wise dataflow with ping-pong buffer	18	10	6.03	4.125	3.25

可以看到，latency 在 n = 1 時 Row-wise dataflow 相較 Column-wise dataflow 可以減少 50%。而 ping-pong buffer 比起單一 buffer 可以進步 2cycle。當 n 越大，ping-pong buffer 的成效越明顯。以我們最後選擇的 n=4 來看，ping-pong buffer 使 latency 減少 25%。

2.3 AT Optimization – 平行度分析

我們發現平行度，也就是乘法器的個數對於 AT 的表現有重要的影響，以下是我們的推導。

$$\boldsymbol{AT} = (A_{mult\ total} + A_{const}) * (E[Cycle\ \ per\ \ row] * Num\ \ of\ \ row\ \ per\ \ task\ )$$

$\boldsymbol{n}$: 平行度(乘法器的個數)

$$\boldsymbol{A_{mult\ \ total}} = n * A_{mult\ \ single}$$

$\boldsymbol{A_{mult\ \ single}}$ 指一個 core.v 的面積，包含一個乘法器，一個減法器，以及 truncation/overflow module。$A_{mult\ \ single}$ 在 cycle time 固定下為常數。

$\boldsymbol{A_{const}}$ 為乘法器個數無關的面積。在此 design 中包含 x register，a_cache_odd register， a_cache_even register，b_cache register， FSM，control circuit 等。A_{const} 在 cycle time 固定下為常數。

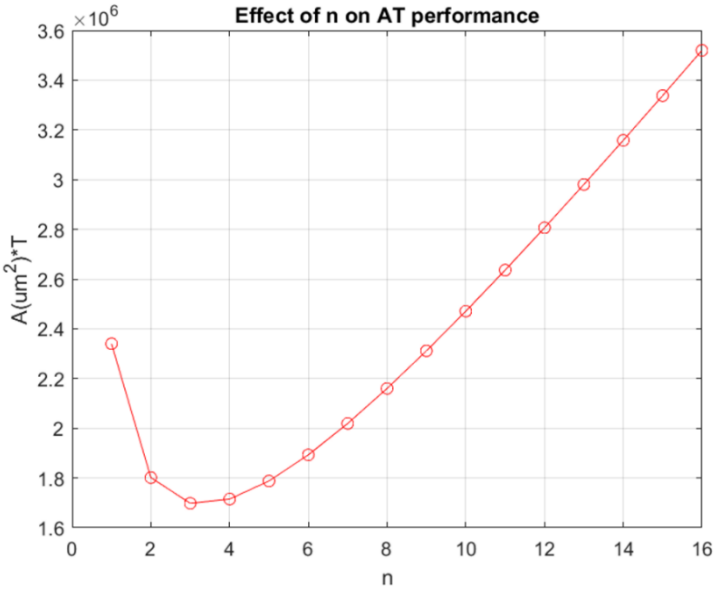
$$\boldsymbol{E[Cycle\ \ per\ \ row]} = \frac{16}{n} + 2 + \frac{1}{2^{\frac{16}{n}-1}}$$ (詳見 2.2)

$\boldsymbol{Num\ \ of\ \ row\ \ per\ \ task}$: 與 n 無關的常數

我們發現 AT 可以寫成以 n 為變數的 function

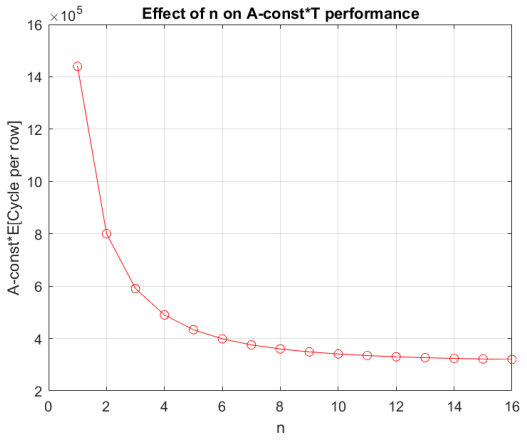
$$AT = (A_{mult\ \ total} + A_{const}) * (E[Cycle\ \ per\ \ row] * Num\ \ of\ \ row\ \ per\ \ task\ )$$
$$= \left(n * A_{mult\ \ single} + A_{const}\right) * \left(\frac{16}{n} + 2 + \frac{1}{2^{\frac{16}{n}-1}}\right) * Num\ \ of\ \ row\ \ per\ \ task$$

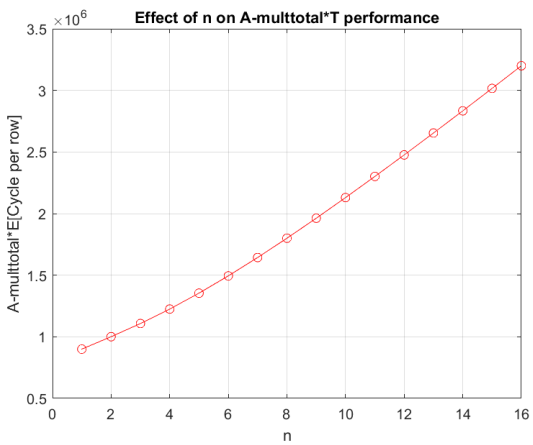
經過實驗，cycle time 為 3ns 時 $A_{mult\ \ single} = 50000um^2$，$A_{const} = 80000um^2$代入上式畫圖可以得到下圖結果。可以發現在 2 的冪次方的選擇中，n = 4 在 AT 上有最好的表現。



詳細分析為什麼 n 太大太小，AT 的表現都比較差，可以分別對$A_{mult\ total} * E[Cycle\ per\ row]$和$A_{const} * E[Cycle\ per\ row]$作圖。由以下兩張圖交叉比對可以發現當 n 太小，$E[Cycle\ per\ row]$太大，進而導致 $A_{const} * E[Cycle\ per\ row]$過大而影響整體 AT performance，而當 n 越大，乘法器因為 next row mem miss 沒有進行 valid 運算的機率就越高。乘法器 utilization 與 n 呈現負相關，因此當 n 越大， $A_{mult\ total} * E[Cycle\ per\ row]$會越大，進而影響整體 AT。

	優點	缺點
n 大	$A_{const} * E[Cycle\ per\ row]$很小	utilization 與 n 呈現負相關
n 小	乘法器 utilization 很高	$E[Cycle\ per\ row]$太大





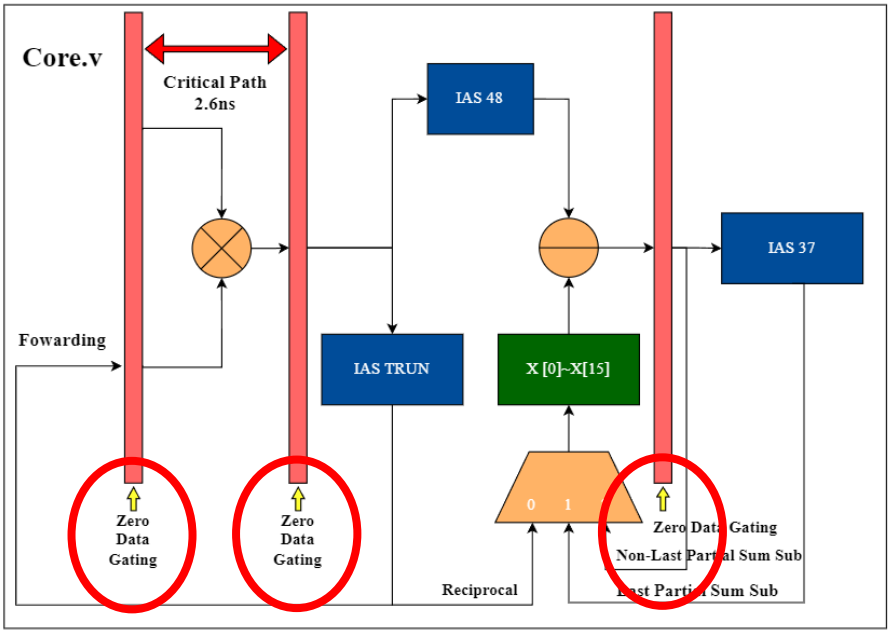
結論來說，n=4 的 AT 相較於 n=16 減少 50 %，相較於 n=1 減少 25%。選擇適當的平行度對於 AT performance 有深刻的影響。

2.4 Power Optimization – Zero Data Gating

運用 Zero Data Gating 能使 power 降低 3~7%。

Sparsity，也就是 A 矩陣中資料為 0 的比例，能提供額外的優化機會。如果 A 矩陣中某元素 a 為 0，則不論對應的 x 的值，a*x 必為 0。利用這樣的特性，我們使用 Zero Data Gating 來節省 power。

Zero Data Gating 的具體作法參考下圖。當我們在 core.v 外部檢測 A 矩陣中元素 a 為 0，我們會傳遞額外訊號通知 core.v，以啟動 Zero Data Gating。由於 a*x=0，因此這筆資料不須要更新 x，也能保持正確性。因此當 Zero Data Gating 啟動，我們會將 pipeline register 的值保持為上個 cycle 的值，以減少 switching activity，進而減少 dynamic power。使用 Zero Data Gating 根據 tb 的 sparsity 程度不同，能將 power 降低 3~7%。



2.5 Latency Optimization –Zero Cycle Skipping

於 2.4 中，我們利用資料 sparsity 的特性，使用 Zero Data Gating 能將 power 降低 3~7%。然而，如果改使用 Zero Cycle Skipping，能大幅縮減 latency。

當 a 為 0，由於 a*x=0，因此這筆資料不須要更新 x，也能保持正確性。Zero Data Gating 對於 a 為 0 的情形仍然花費相同的時間來運算，只是將 register gated 來節省 power。而 Zero Cycle Skipping 則更進一步，只要 a 為 0 就跳過不運算，只運算 a 不為 0 的運算。如此以來可以節省 latency。

然而要實現 Zero Cycle Skipping，對於資料需要較多預處理。因為時間關係，我們未能在此次 final project 實作出 Zero Cycle Skipping 功能。期待未來的 project 中，有機會能用上此技巧。

3. Screenshot

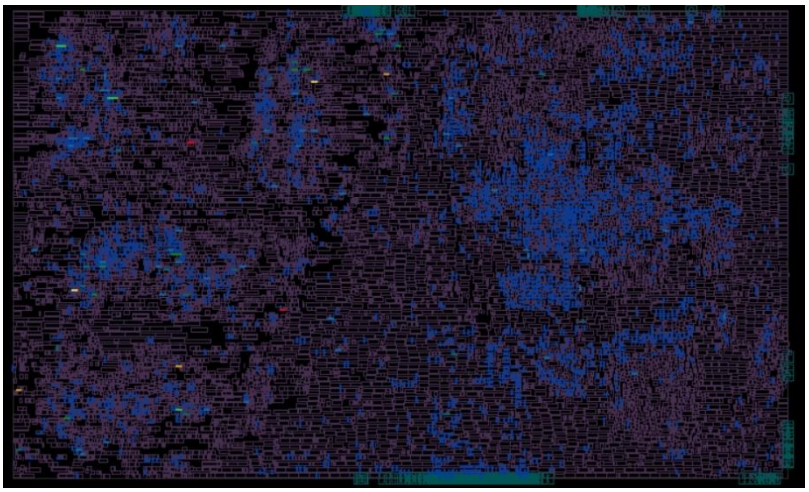
3.1 NLint Report

```
------------------------------------------------------------------------------------------------
Results Summary:
------------------------------------------------------------------------------------------------
   Goal Run           :      cvsd_lint
   Top Module         :      GSIM
------------------------------------------------------------------------------------------------
   Reports Directory:
   /home/raid7_2/user10/r10117/clean/01_RTL/Lint/cvsd_lint/consolidated_reports/GSIM_cvsd_lint/

   SpyGlass LogFile:
    /home/raid7_2/user10/r10117/clean/01_RTL/Lint/cvsd_lint/GSIM/cvsd_lint/spyglass.log

   Standard Reports:
     moresimple.rpt          no_msg_reporting_rules.rpt

   HTML report:
    /home/raid7_2/user10/r10117/clean/01_RTL/Lint/cvsd_lint/html_reports/goals_summary.html


   Technology Reports:
     <Not Available>


------------------------------------------------------------------------------------------------
   Goal Violation Summary:
       Waived   Messages:                    0 Errors,      0 Warnings,      0 Infos
       Reported Messages:         0 Fatals,    0 Errors,     42 Warnings,      4 Infos
------------------------------------------------------------------------------------------------


------------------------------------------------------------------------------------------------

run_goal: info: spyglass.log successfully updated with goal summary
run_goal: info: setting design top `GSIM' as current_design
[r10117@cad29 Lint]$
```

3.2 Coverage Result

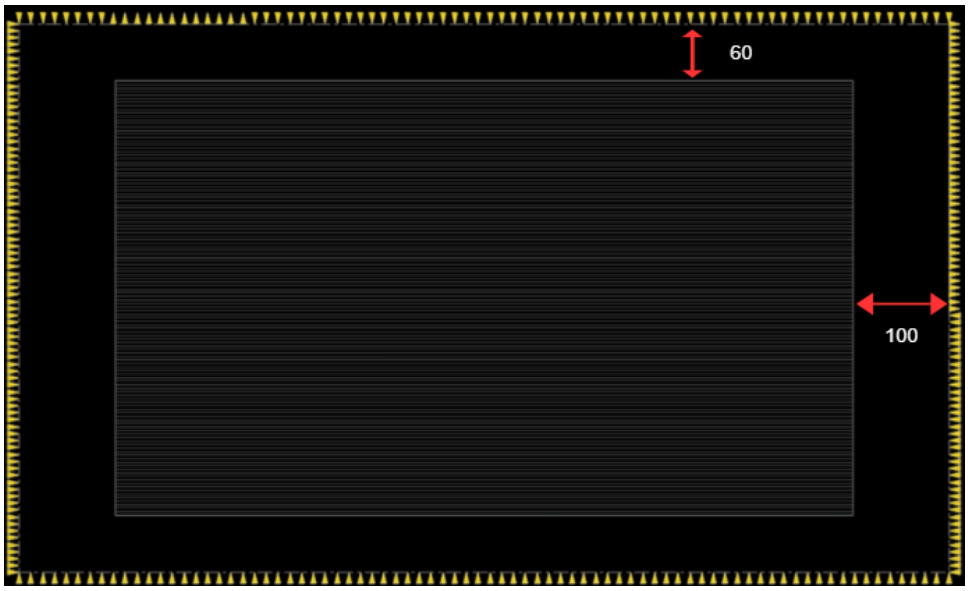
Name	Score	Line	Toggle
testbed	89.85%	98.76%	99.84%
u_GSIM	90.38%	100.00%	99.88%
u_matrix_mem	100.00%	100.00%	100.00%

3.3 Congestion Map

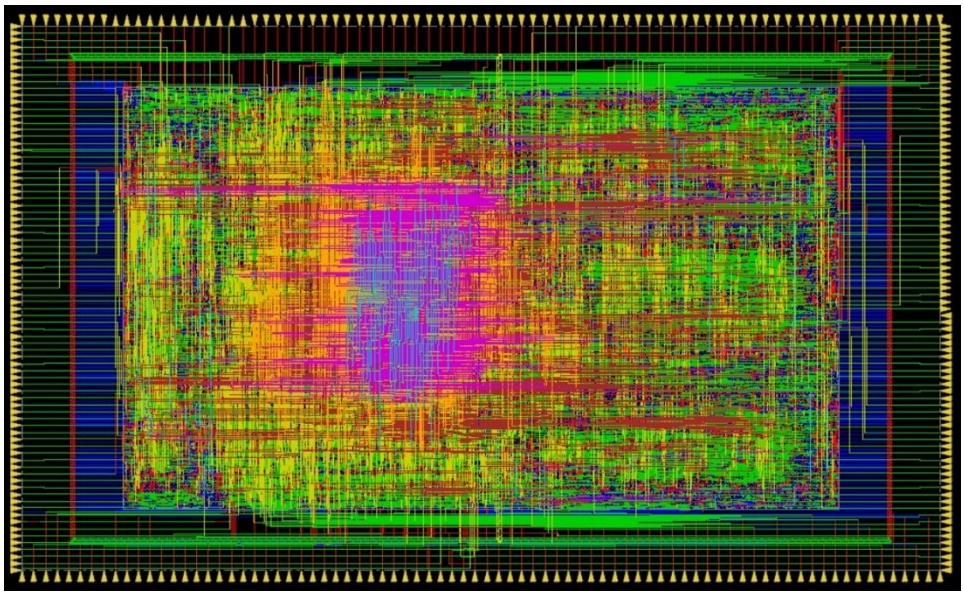


上圖中，紅點表示 timing 較緊的點，以我們的 RTL 來說是乘法器，以及 retiming 完的後段減法電路。屬於預期之中比較緊的部分，所以沒有回去修改 RTL。

3.4 Layout



在 APR 的 **Specify Floorplan**，由於 ratio 設定 0.6，可以從圖中看出左右兩側 I/O pin 相較於上下的 I/O pin 較密。所以我們給左右間距較大，讓較密的一測比較好繞線，左右以及上下分別設定為 100、60。



上圖為最後 APR 結果，直的 stripe 只打一條

4. Performance

4.1 Primetime Power Report

```
(A) Idle power:                 0.0152
(B) Idle_after_active power:    0.0154
(C) Active power:               0.0318
***** Assume (A) & (B) time window are the same *****
(A) & (B) diff ratio:           1.32%
=====================================================
========== Power diff ratio result PASS!!! ==========
=====================================================
[1]    Done                              nWave gsim.fsdb
```

	Power(W)	Time(ns)	Energy($W * ns$)
Active	0.0318	15152.8	481.85

4.2 Area result

```
************************* Analyze Floorplan **************************
    Die Area(um^2)              : 487833.33
    Core Area(um^2)             : 294806.13
    Chip Density (Counting Std Cells and MACROs and IOs): 60.432%
    Core Density (Counting Std Cells and MACROs): 100.000%
    Average utilization         : 100.000%
    Number of instance(s)       : 26736
    Number of Macro(s)          : 0
    Number of IO Pin(s)         : 320
    Number of Power Domain(s) : 0
************************* Estimation Results *************************
**********************************************************************
```

4.3 AT result

	tb0	tb1	tb2	tb3	Total(ns)
Time(ns)	90459.1	173497.3	46303.3	173497.3	483757

Total (ns)	Area (um^2)	AT ($ns * um^2$)
483757	486833	2.36e+11

4.4 Performance Table

Physical category		
Design Stage	Description	Value
Gate-level Simulation	Cycle time for Gate-level Simulation	3.1(ns)
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)	294806.13
	Die area (um^2)	487833.33
Post-layout Simulation	Cycle time for Post-layout Simulation	3.4 (ns)