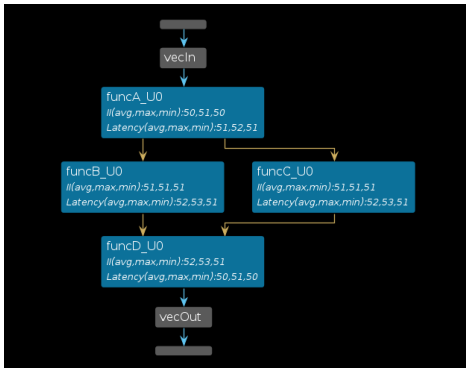
Report

110061592 許睿哲

Explain the original code/system/pragmas and how you implemented it:

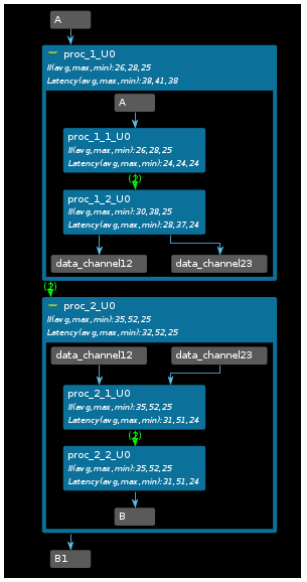


Lab1:

整個 system 其實很簡單，data flow diagram 如右。Testbench 總共讓 DUT(diamond())做 3 次運算，data flow diagram 如下，input(vecIn)進 funcA 後，生成兩個 output，分別進 funcB 及 funcC 進行平行運算，funcD 收到 funcB 與 funcC 的結果後，生成 output(vecOut)。

```
void diamond(data_t vecIn[N], data_t vecOut[N])
{
  data_t c1[N], c2[N], c3[N], c4[N];
#pragma HLS dataflow
  funcA(vecIn, c1, c2);
  funcB(c1, c3);
  funcC(c2, c4);
  funcD(c3, c4, vecOut);
}
```

使用上圖紅框的 pragma 來使用 dataflow viewer



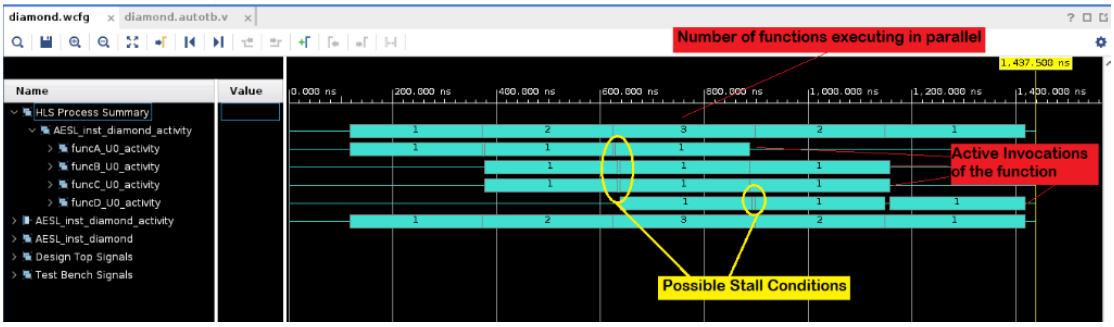
Lab2:

與 lab1 其實很像，data flow diagram 如右，kernel 的運算也都是一些簡單的運算。

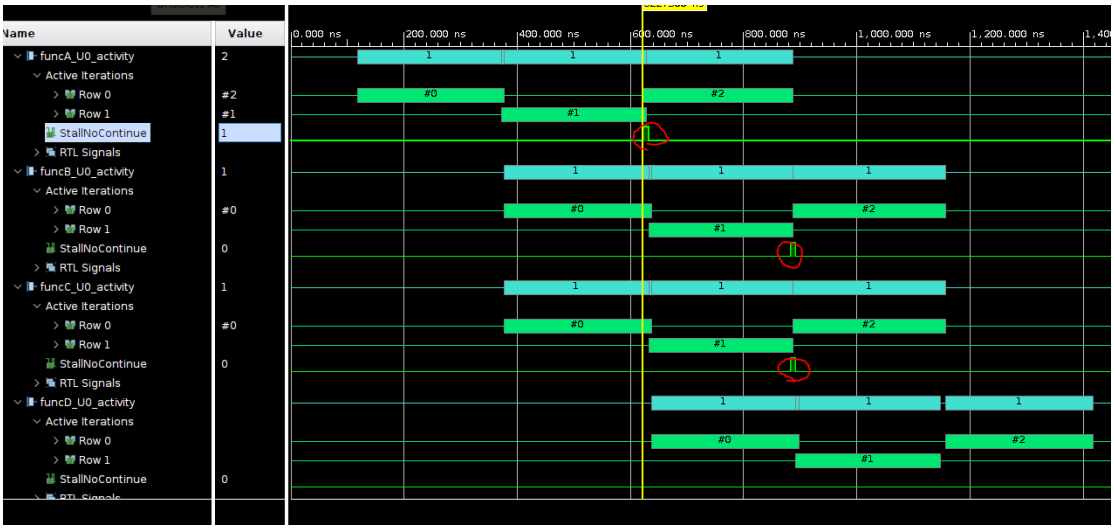
Analyze the timing/performance/utilization

Lab1:

透過 dataflow viewer 比較難看出執行各個 function 平行化與重疊的程度，因此需要透過 waveform 來觀察。



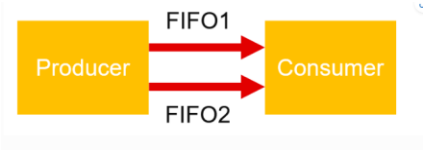
HLS process summary 整理了各個 function 的時間，下面的 AESL_inst_diamond 顯示現在有幾個 iteration 在執行。黃色圈起來的地方表示那個地方有可能有 stall。



展開下方的 AESL_inst_diamond 可以查看更詳細的 waveform，例如這個 sub-function 開始與結束的時間。

特別是 StallNoContinue，以這張圖為例，可以看到 funcB 與 funcC，因為 funcD 還沒計算完的關係而 stall(back pressure)；funcA 因為 funcB 與 funcC 還沒計算完的關係而 stall。

Lab2:



以此圖為例，若 Producer 依序寫入 FIFO1 十次，再寫入 FIFO2 十次，Consumer 依序從 FIFO1, FIFO2, FIFO1, FIFO2 讀。那 FIFO1 的 depth 必須大於 10。這個 lab 的 data_channel1 就等於這個範例的 FIFO1。

If possible, share how you optimize the design and the trade-off you make:

在這個例子，Tutorial 的 channel 是預設的(PIPO)，因為 data 是照固定順序傳輸的，因此使用 FIFO 會比較快，例如 funcA 一有 data 可以用，funcB. funcC 就可以馬上執行了；相對的因為 FIFO 的 protocol 較複雜，電路可能面積會較大。PIPO 則是如果資料是隨機順序會比較適合。

Explain what you observed and learned :

Lab1:

學會如何透過 dataflow viewer 來分析整個系統，例如:

在 The Dataflow Graph Pane:

Ports(灰色框框)。

Node(藍色框框):

functions(有可能是 compiler 自己產生的(_proc)，而不是我們自己定義的)或 loop。

Edge:

藍色: function 之間的 datadependencies

綠色: function 之間的 FIFO channels

黃色: function 之間的 PIPO channels

Process Channel

Name	Cosim Category	Cosim Stalling Time	FIFO EMPTY	FIFO FULL	Cosim Stall No Start	Cosim Stall No Continue	Cosim AVG II	Cosim Max II	Cosim Min II	Cosim AVG Latency	Cos
funcA_U0	none	0.76%	0.00%	0.00%	0.00%	0.76%	50	51	50	51	52
funcB_U0	none	0.76%	0.00%	0.00%	0.38%	0.76%	51	51	51	52	53
funcC_U0	none	0.76%	0.00%	0.00%	0.38%	0.76%	51	51	51	52	53
funcD_U0	none	1.15%	0.00%	0.00%	1.15%	0.00%	52	53	51	50	51

Process table 可以查看上面各個 function 的資訊來查看那些地方可以 optimize，例如 read/write 的 block 的 percentage 特別多的 function，就可以判斷是前面還是他後面的地方可以 optimize，或有可能導致 deadlock。

Cosim Stalling Time: 因為這個 process stall 所花的時間百分比。

Cosim Read Block Time or Cosim Write Block Time 代表了 read/write 進 channel 被 block 的時間百分比。

Cosim Stall No Start and Cosim Stall No Continue indicates forward and back pressure respectively:

- Forward pressure: 上一個 function 還沒準備好。
- Back pressure: 下一個 function 還沒準備好。

Channel table 也可以查看他的 type, depth(), maximum depth achieved。

Lab2:

如何修改 FIFO 的 depth，並新增 pragma 至 source code。

總共有三種方法，第一種是 Manual FIFO Sizing，手動修改，一步一步查看各個 channel 的 depth 要設多少，有點類似 trial and error；第二種是 Global FIFO Sizing，一次設定全部 channel 的 depth，再看各自 channel 跑 co-sim 會用到最大

的 depth 是多少調整；第三種 Automated FIFO Sizing 最簡單，需要一些時間來重複跑 co-sim，直到找到最佳的 FIFO depth，但是也有可能這個演算法會永遠找不到最佳的 depth。

Automated FIFO Sizing 還會依照 performance 來推薦 channel 的 depth，左圖為推薦的，右圖為

Process Channel FIFO Sizing

Name	Suggest Depth	Type
data_channel1	10	Stream
data_channel2	2	Stream
data_channel1	11	Stream
data_channel2	2	Stream
data_channel1	10	Stream
data_channel2	2	Stream

Name	Cosim Category	FIFO EMPTY	FIFO FULL	Cosim Max Depth	Depth
data_channel1	read_block	10.45%	0.00%	10	10
data_channel2	read_block	22.39%	0.00%	1	1
data_channel1	none	0.00%	0.00%	10	10
data_channel2	read_block	11.94%	0.00%	1	1
data_channel1	read_block	25.37%	0.00%	10	10
data_channel2	read_block	64.18%	0.00%	1	1

Submit your work to GitHub:

ray3210ray3210/2022HLS (github.com)