**臺灣證券交易傳輸電文FPGA系統研究發展期末報告書**

**主持人: 張燕光教授**

**計算機與網路架構實驗室**

**國立成功大學資訊系**

**中 華 民 國 一 ○** 八 **年** 三 **月**

**目錄**

目錄--------------------------------------------------2

摘要--------------------------------------------------3

系統架構敘述------------------------------------------4

**系統運作流程------------------------------------------6**

**電路設計---------------------------------------------12**

**刪單交易與連線State Machine設計---------------------37**

**Stock & Warrants ID Mapping流程設計-----------------38**

FPGA資源使用量---------------------------------------44

BRAM詳細用量與未來使用估算---------------------------45

軟體測試---------------------------------------------48

結案數據---------------------------------------------52

計畫成果---------------------------------------------55

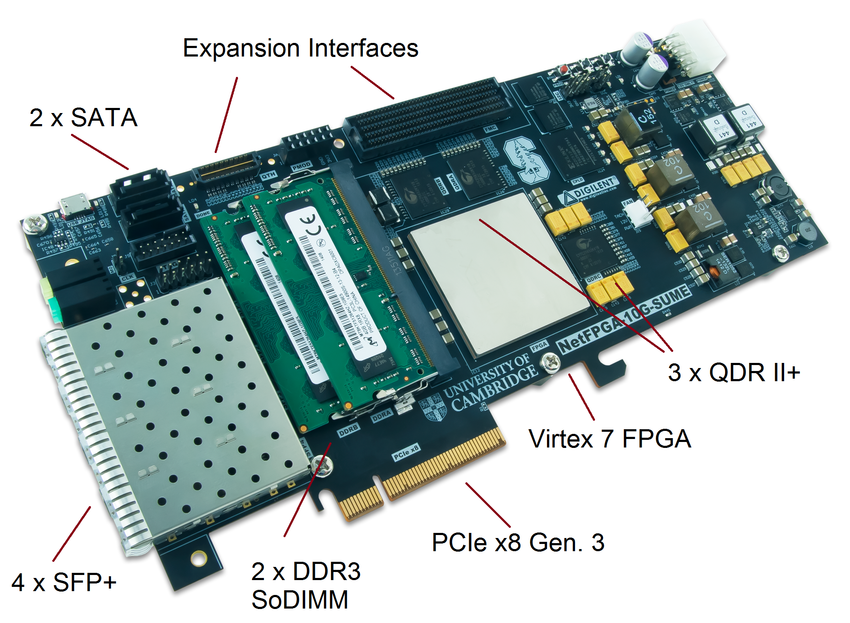
參與人員---------------------------------------------55

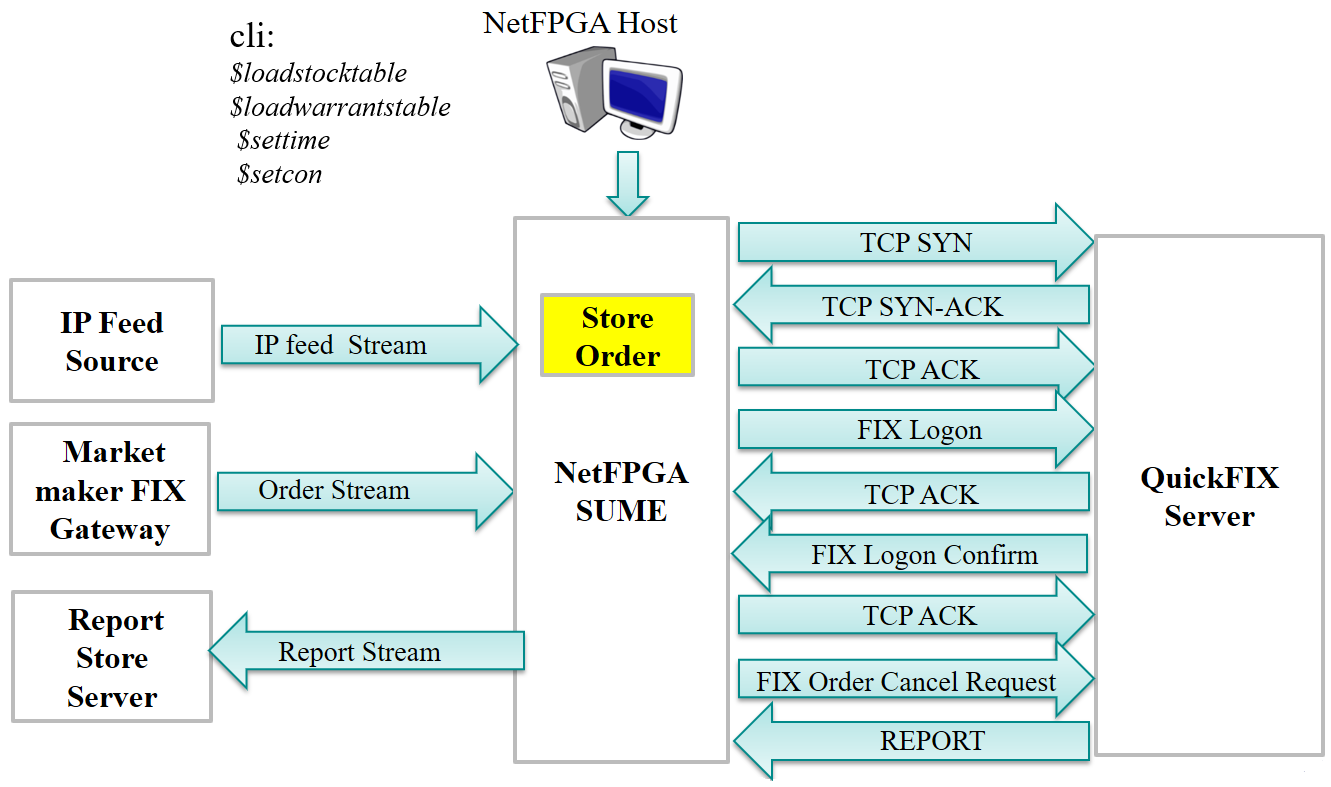
附件-------------------------------------------------56

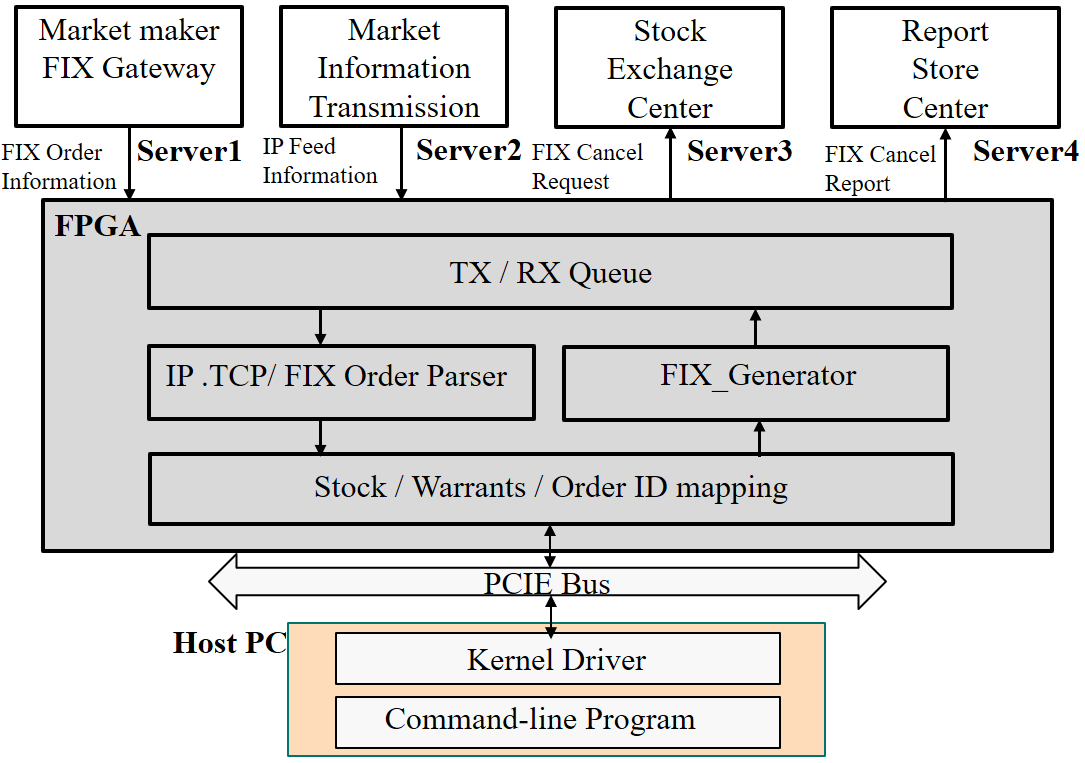
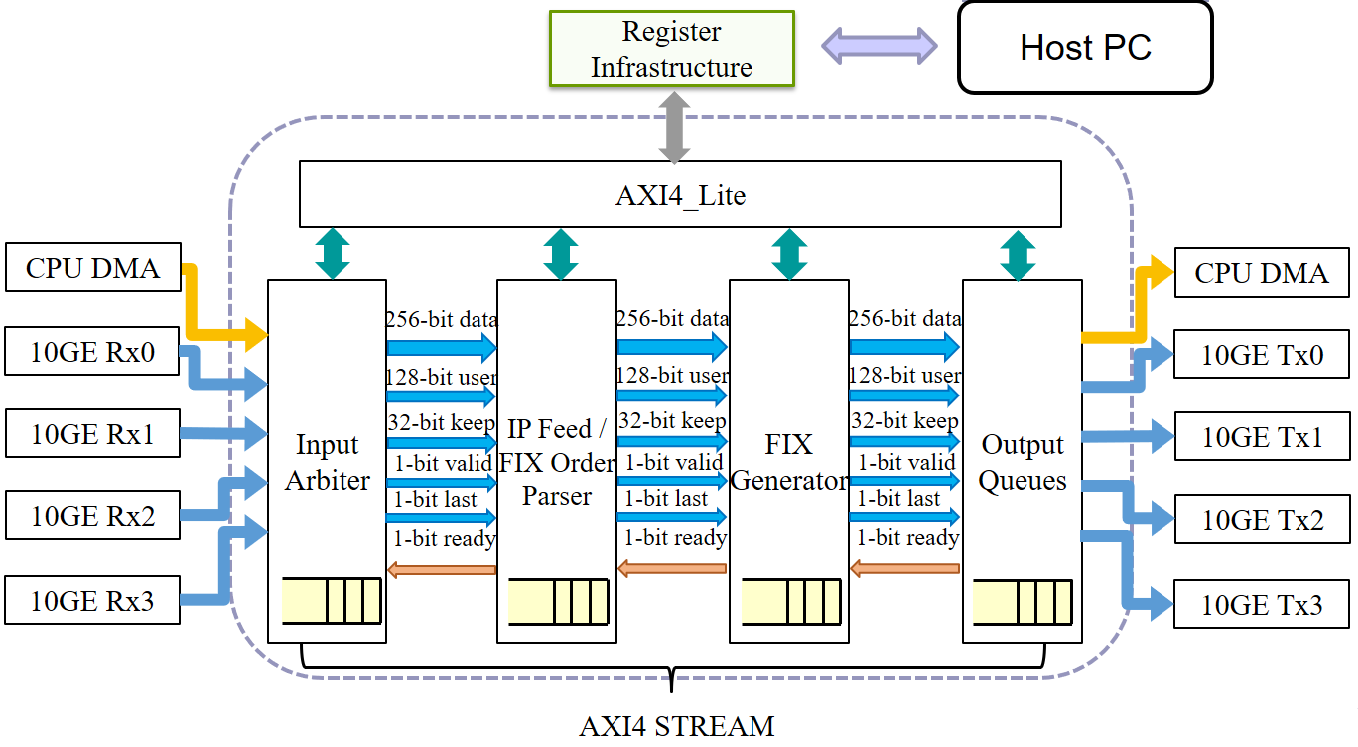
**摘要**

本計畫為成大資訊系計算機與網路架構實驗室與凱基證券公司之產學合作，預計專案期間兩年。專案目標為開發能夠應用於金融交易之FPGA系統，包含翻譯台灣證券交易所之行情(證交所自訂IP行情傳輸電文)、翻譯台灣證券交易所之交易(FIX4.4 Protocol)。並能透過電腦即時移植交易策略至FPGA系統。從計畫最開始使用的NetFPGA 1G開發板，目前系統已經進展到採用最新的NetFPGA SUME開發版，透過此開發版我們可以實作高頻交易平台。我們設計的系統可以接收從台灣證券交易所送來的市場資訊，透過硬體自動比對演算法與預先載入的交易策略做比對，送出對應的刪單訊息到證交所。此系統目前採用證交所自訂IP行情傳輸電文與FIX 4.4 Protocol，並實作TCP/FIX Protocol規範的連線機制，能維持與QuickFIX Server連線，符合計畫要求。與軟體的交易平台相比，此系統可以達到較低的延遲。為了量測系統延遲，設計了相對應的測量實驗，先將預先設計的策略讀入開發版，並送出預錄的行情串流至交易系統，並接收交易系統所傳出之交易封包串流，過程為了與真實情況相符，皆實際採用10G網卡與10G SFP+網路線傳送封包，根據目前的實驗結果，平均約可在0.4~0.6µs完成一筆刪單交易。

**系統架構敘述**

* 開發平台
  + 目前採用NetFPGA SUME
    - Xilinx Virtex-7 XC7V690T FFG1761-3
    - 693,120 Logic Cells
    - 10,888 kb Distributed RAM
    - 52,920 kb Block RAM
    - 40 Gbits Ethernet port
  + 搭配PC為支援PCIe Bus主機板，OS為Ubuntu 14.04
  + 包含C driver code讓HOST與NetFPGA溝通
* 連線協定 – TCP & FIX
  + 為了與證交所能夠連線溝通，必須要實作能夠維持TCP 與 FIX Protocol的系統
  + TCP Protocol
    - 在OSI七層中屬Layer 4 – Transport Layer
    - End to End Connection
    - Dependable，需確認對方是否收到訊息，亦須確認順序是否正確
  + FIX Protocol
    - Application Layer
    - 用於金融資訊
    - 以ASCII Code表示
    - 亦須檢查對方是否收到訊息及訊息順序
* 與純軟體差異
  + 效能上透過NetFPGA處理，可以省去Packet通過Kernel的時間，可以比用純軟體處理好
  + 由於硬體資源限制，無法在NetFPGA上放太多交易策略，太複雜的策略也不易設計

**系統運作流程**

* 系統架構圖
* 系統功能
  + 實作IP / UDP stack
    - IP / UDP Checksum驗證
    - IP Version驗證
  + 實作部分TCP stack
    - TCP Checksum驗證與計算
    - Three-way handshake
    - Seq / Ack 機制
    - Timestamp紀錄
  + 實作Hardware clock
    - Load kernel clock to NetFPGA, and NetFPGA will handle a hardware clock
  + Filter設計
    - 可過濾掉不需要的訊息，只留格式六行情封包
  + Decode出行情封包內容
  + 透過Symbol ID Mapping機制搜尋交易策略表
    - 4-left Cuckoo Hash
  + 可透過command-line software修改策略表
    - 透過PCI Bus與register interface溝通
  + 送出FIX Order
    - 與FIX Server連線後，收到行情封包會比對策略表，若有中則送出對應的FIX Order
  + 查看目前FIX封包流水號、跳號重送機制
    - 記錄當前Server與Client的FIX Seq
    - 記錄上一次重送的起始序號
* 內部資料傳送- (TOP-DOWN)
* 專案系統設計
  + 開發專案

=================================================

/root/NetFPGA-SUME-live/projects/reference\_router/

=================================================

* + 專案模組設定

=================================================

參考

/root/NetFPGA-SUME-live/lib/hw/std/cores/

router\_output\_port\_lookup\_v1\_0\_0/ component.xml

/root/etFPGA-SUME-live/lib/hw/std/cores/

router\_output\_port\_lookup\_v1\_0\_0/ router\_output\_port\_lookup.tcl

=================================================

* + 專案Register definition

=================================================

/root/NetFPGA-SUME-live/lib/hw/std/cores/

router\_output\_port\_lookup\_v1\_0\_0/data/output\_port\_lookup\_regs\_defines.h

/root/NetFPGA-SUME-live/projects/reference\_router/sw/embedded/src/

sume\_register\_defines.h

=================================================

* + 實體網路connection

================================================================

[mac 0] 接收UDP格式6行情封包

[mac 1] 接收FIX Order封包

[mac 2] 送出FIX Order report封包、Order cancel reject封包、Session Reject 封包

[mac 3] 與QUICKFIX Server連結

=================================================

* + 子資料夾

================================================================

bitfiles

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

| 存放由vivado編譯完的reference router的

bitfile，把

bitfile載入

| Netfpga-SUME即可執行這個project的功能

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

hw

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

| tcl 包含各種執行vivado project編譯的腳本

| create\_ip 執行IP Core configuration

| hdl reset Netfpga-SUME

| constraints 設定timing constraint

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

lib

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

| 會用到的軟體函數庫，包含設置port的mac

address、hardware上的

| routing table、IP destination filtering table、arp table

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

sw

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

| embedded 包含xilinx SDK腳本、i2c (或稱iic、Inter-Integrated Circuit)

| 程式，可讀取netfpga-SUME板子上的power information、

| 產生156.25 MHz的clock cycle給SUME….等

| host 主要包含command line interface (cli) 控制程式，我們可對

| 原本的reference router的相關指令做修改

| ---------------------------------------------------------------------

| cli指令:

| |

| | $loadstrategy: 1. 讀取策略檔stock\_strategy\_load\_1118.txt

| | 2. 策略檔裡每一個股票代號都會先算出對應的internal ID

| | (internal ID會輸出到hash\_index\_1118.txt)

| | 3. 接下來就會把internal ID跟策略內容逐一送給

| | NetFPGA讓NetFPGA完成寫入

| | 4. NetFPGA寫入完成後會回傳寫入的address

| | 如果正確寫入這些address就會是送給NetFPGA

| | internal ID

| | 如果寫入不如預期NetFPGA會回傳0xFFFF

| | (NetFPGA回傳的address會

| | hash\_id\_mapping\_1118.txt)

| | $liststrategy: 1. 讀取stock\_symbol\_load\_1118.txt

| | 2. 從NetFPGA讀出策略到stock\_strategy\_list\_1118.txt

| |

| | $setstrategy: 設定特定股票的策略

| |

| | $showstrategy: 顯示特定股票策略在螢幕上

| |

| | $listprocesscount: 列出NetFPGA從燒錄後工作了幾個clock cycle

| | P.S. 工作的定義是處理封包

| | User\_Data\_Path下有一個模組會去檢查所有stage的

| | FIFO

| | $settime: 將kernel time讀入 NetFGPA

| |

| | $readtime: 顯示NetFGPA內部Clock

| |

| | $sendpkt: 與QuickFIX Server建立連線

| |

| | $readseq: 顯示目前NetFPGA與QuickFIX Server 儲存的

| | Sequence number(流水號)

| |

| | $setseq: 設定 NetFGPA的Sequence number(流水號)

| |

| | $readresend: 顯示QuickFIX Server要求重送的Sequence number(流水

| | 號)

| |

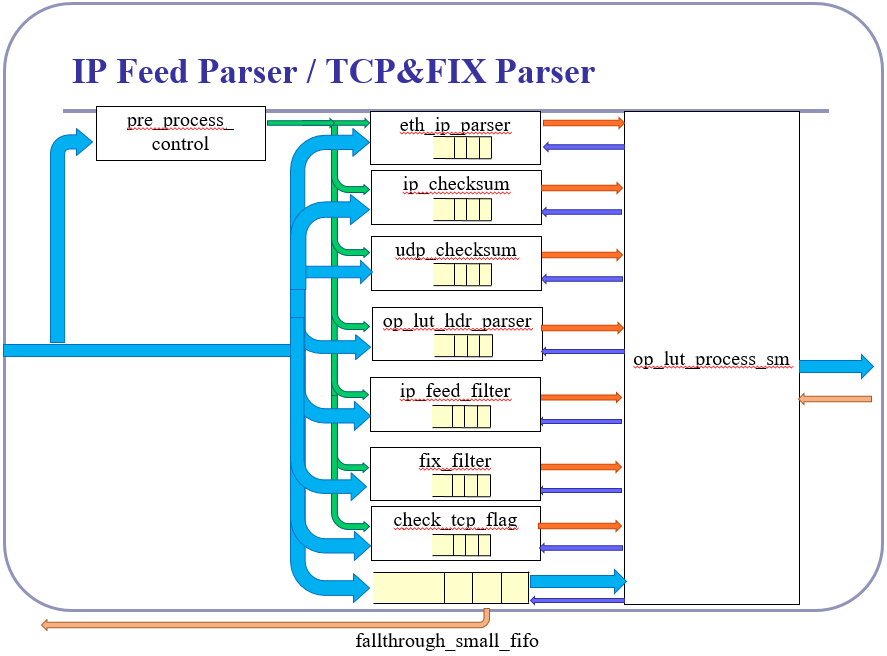
| ------------------------------------------------------------------------

|

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

**電路設計**

# IP\_Feed\_Fix\_Parser

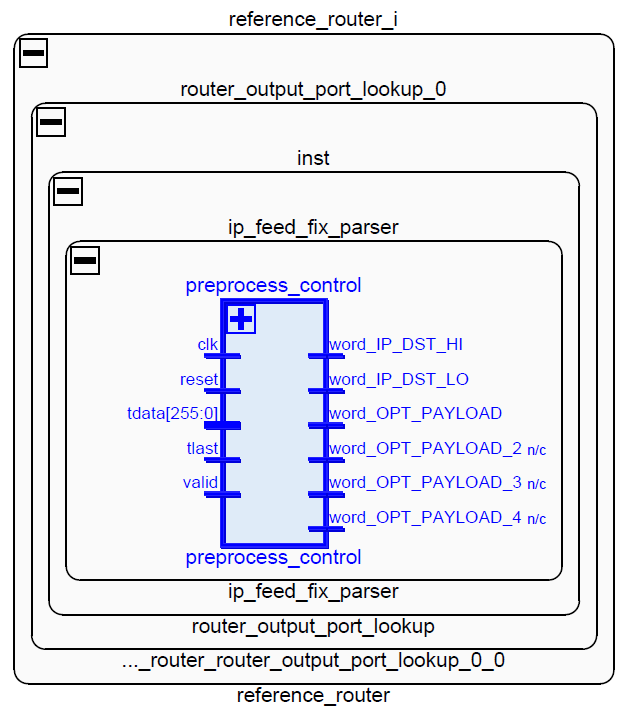


說明:

當有新的packet進入時會同時寫入fallthrough\_small\_fifo和其餘8個module(pre\_process\_control, eth\_ip\_parser, ip\_checksum, udp\_checksum, op\_lut\_hdr\_parser, ip\_feed\_filter, fix\_filter, check\_tcp\_flag)同時處理(parallel process)，待上述module處理完畢後op\_lut\_process\_sm會從fallthrough\_small\_fifo讀出packet並檢查signal(from the 8 module ouput)決定這個packet的處理方式(e.g. DROP, SEND, FIX SERVER CONNECTION)並forward到下個module(FIX Generator)。

# IP\_Feed\_Fix\_Parser內部module功能

preprocess\_control



功能：

判斷進來的tdata是封包的哪一段資料，並把相對應的訊號線拉起來給後面的module擷取資料。

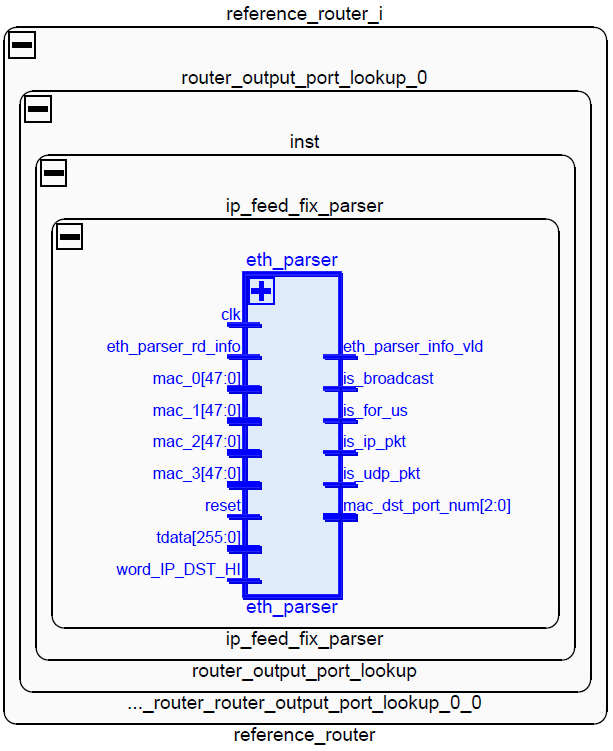
詳述：

SUME的tdata可以存放256 bits的資料，因為一個封包最多不會超過256\*6=1536個bits，所以用6個256的tdata存就可以了，word\_IP\_DST\_HI代表封包第一個256bits、word\_IP\_DST\_LO代表第二個……以此類推直到遇到eop才會回到word\_IP\_DST\_HI。假設當封包的第二個256bits進來的時候，word\_IP\_DST\_LO會被設為1，後面的module就會知道目前送到封包的第二個tdata，後面需要用到這個tdata裡面資料的module就會被trigger 。

原版差異：

原版只有word\_IP\_DST\_HI、word\_IP\_DST\_LO，KGI專案多了四條訊號線分別為word\_OPT\_PAYLOAD、word\_OPT\_PAYLOAD\_2、word\_OPT\_PAYLOAD\_3、word\_OPT\_PAYLOAD\_4，其他邏輯都一樣

eth\_parser



功能：

判斷是否為IPV4 packet、udp packet、from cpu、broadcast packet、dst port number

詳述：

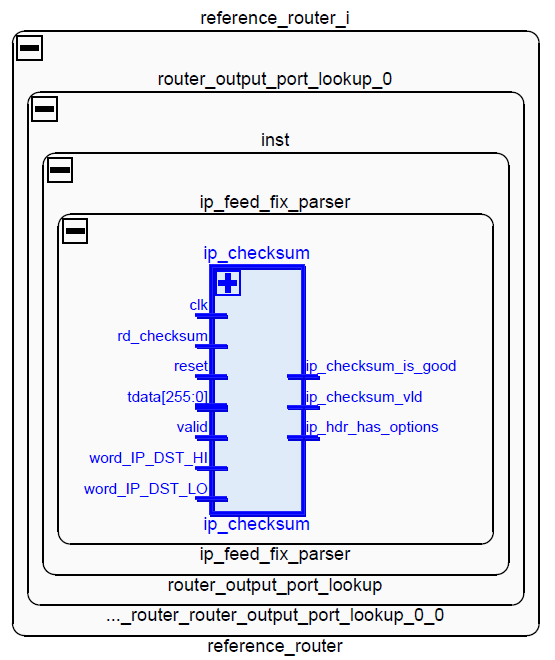
一個循序電路，當word\_IP\_DST\_HI=1存dst\_MAC、ethertype、protocol三個state，state IDLE等待packet進來，如果(!parser\_fifo\_empty=1)進到DO\_SEARCH，state DO\_SEARCH檢查destionation MAC是不是送給NetFPGA的，如果是或是broadcast代表是給我們的，然後進到FLUSH\_ENTRY，state FLUSH\_ENTRY將rd\_parser設為1(讀下一個256 bits)，然後回到IDLE

兩個fifo，fifo eth\_fifo輸出packet類型(is\_for\_us, is\_ip\_pkt, is\_udp\_pkt, is\_broadcast, mac\_dst\_port\_num)，fifo parser為data flow的暫存，輸出dst\_MAC\_fifo, ethertype\_fifo給State machine使用

差異：

拿掉is\_arp\_pkt，新增is\_udp\_packet

ip\_checksum

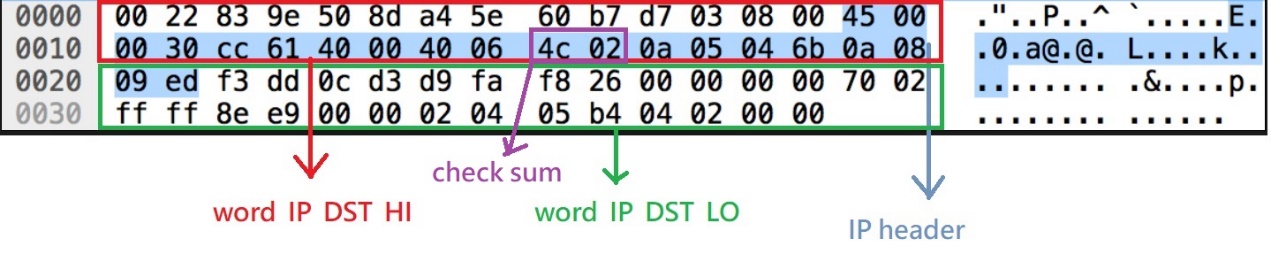


功能：

計算封包的ip header checksum（可參見RFC1936）

詳述：

當word\_IP\_DST\_HI、word\_IP\_DST\_LO被前面的module（preprocess\_control）assign為1，會去tdata相應的位置找出IP header取出來算（因為IP header剛好被分散放在兩個tdata裡面），IP checksum算法如下圖為例（本圖為一般封包）



計算IP header是不是正確的、檢查header有沒有option、decrement TTL、當TTL減1之後算出新的checksum。

**Hardware checksum算法**：用1’s complment add，然後把carry out當成carry in 加回來；以上圖為例：4500+0030+4000+4006+4c02+0a05+046b+0a08+09ed+ cc61=1FFFE=>FFFE+1=FFFF，當我們的checksum算出來為0xffff表示正確，否則錯誤。

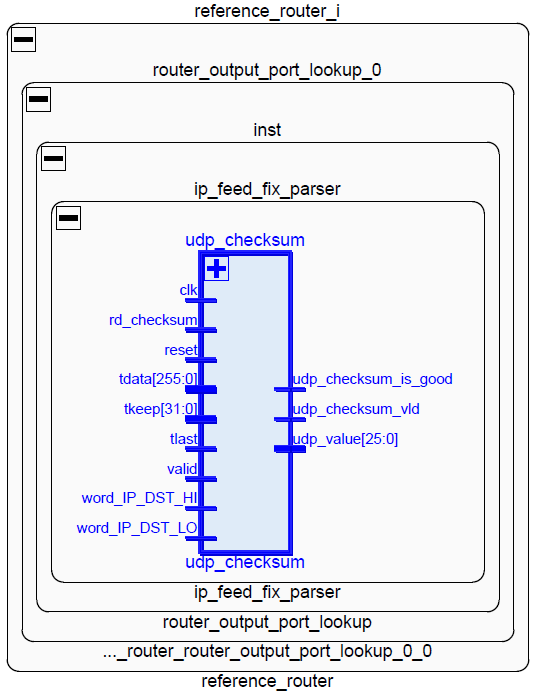
因為IP header是分散存在兩個tdata裡面，check sum 沒辦法在同一個clock cycle內算完，所以當算完checksum之後output的三條線：

1. ip\_checksum\_vld會拉起來表示checksum已算完
2. 比對封包裡的checksum與我們的計算結果是否相符，如果相符ip\_checksum\_is\_good訊號就會拉起來表示正常
3. 如果IP header 分析出有option，則ip\_hdr\_has\_options訊號會被拉起來表示封包有option

原版差異：

KGI專案雖然仍然有對TTL做更新、計算、把新的TTL拿去算新的checksum，但是output沒有檢查TTL是否正確的相關訊號，用不到

udp\_checksum（for行情封包）

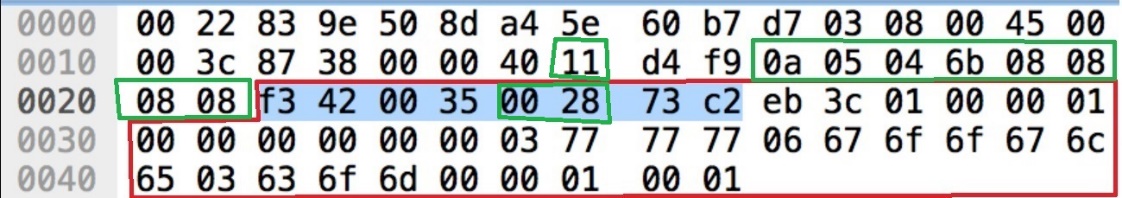


功能：

計算封包的UDP checksum（可參見RFC1936）

詳述：

算法同樣是用Hardware checksum算法，也就是1’s complment add，如果結果為0xffff表示正確，否則錯誤，UDP checksum算法如下圖為例（一般封包）：



一樣用1’s complement按照UDP的規則全部加起來

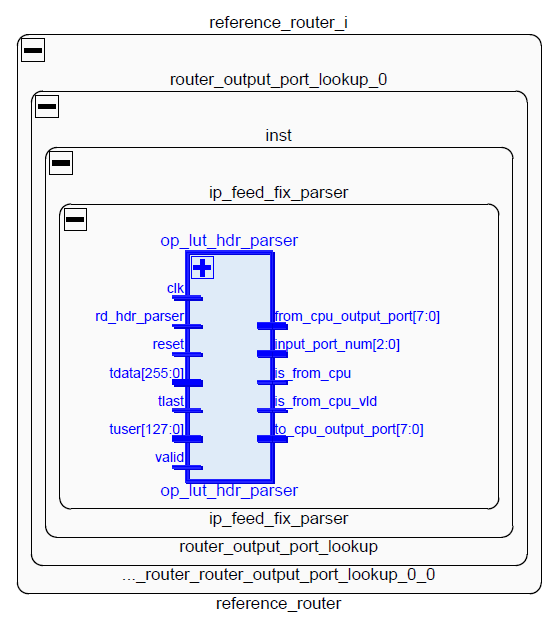
0a05+046b+0808+0808+0011+0028+f342+35+28+73c2+eb3c +100+1+377+7777+667+6f6f+676c+6503+636f+6d00+1+1=4FFFB=>FFFB+4=FFFF不過比ip checksum複雜一點的地方是因為我們算UDP checksum還需要知道封包payload的長度，也就是封包在tdata的哪個地方結尾，當tlast==1表示我們已經知道封包最後一筆tdata進來了，我們會去看tkeep哪些地方為1，然後去tdata相對應的位置把剩下的payload取出來，因為tkeep在封包結束的地方之後全部都會被設為0來補齊tkeep 32bits的長度（當tlast!=1，tkeep的32bits全部都是1，表示全部的256bits tdata的資料都有用到）。

關於output的部分：

1. udp\_checksum\_vld表示UDP checksum已算完
2. udp\_checksum\_is\_good表示UDP checksum計算結果正確

udp\_value把udp checksum的數值傳出來

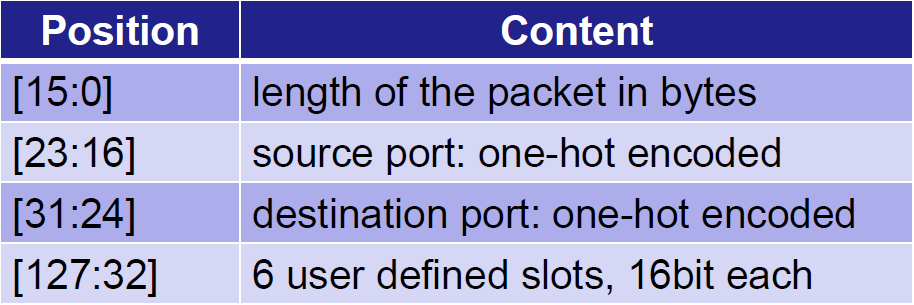
op\_lut\_hdr\_parser



功能：

檢查封包是不是來自CPU。

詳述：



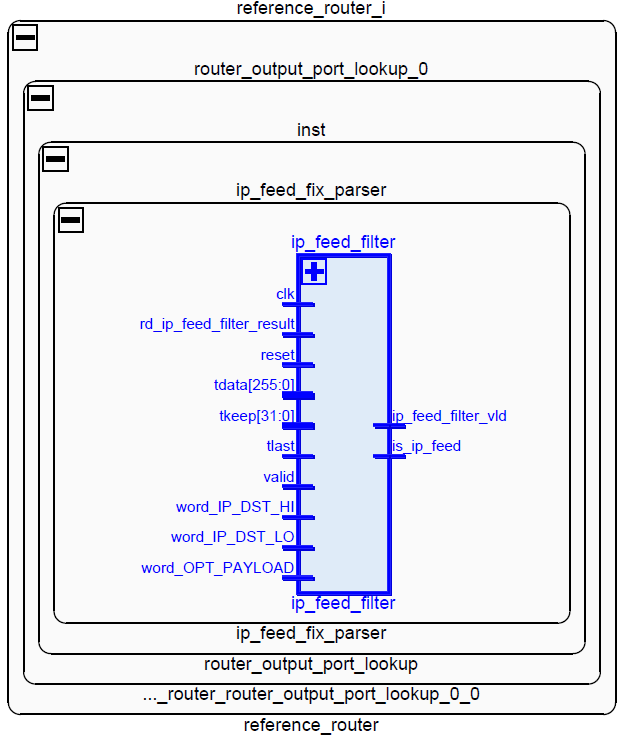
這個module只會用到tuser的資料，這裡值得注意的是我們會去tuser[23:16]抓source port，one-hot encoded的意思是8個bits裡面一次只會有一個1，例如00010000表示資料是從第0個port～第7個port其中的第4個port進來。這裡不管封包是不是從CPU來的，他from\_cpu\_output\_port、to\_cpu\_output\_port都會先算出來，等這些output傳進下一個module＂op\_lut\_process\_sm＂，他會依據現在is\_from\_cpu是不是1，如果是，dst\_port=from\_cpu\_output\_port，否則另外判斷dst\_port（詳細情形會在op\_lut\_process\_sm說明）

Output部分：

1. is\_from\_cpu表示封包是不是從CPU來的（是：1、不是：0）
2. to\_cpu\_output\_port表示這個封包的資料要往CPU的哪一個port送
3. from\_cpu\_output\_port表示這個來自CPU port的封包該往哪個port送
4. input\_port\_num表示這個封包來自哪一個port

is\_from\_cpu\_vld表示這個module的封包來源判斷已經處理完畢

ip\_feed\_filter



功能：

判斷是否為格式六封包，判斷依據為檢查udp、ESC-CODE、傳輸格式代碼、TERMINAL-CODE等欄位

詳述：

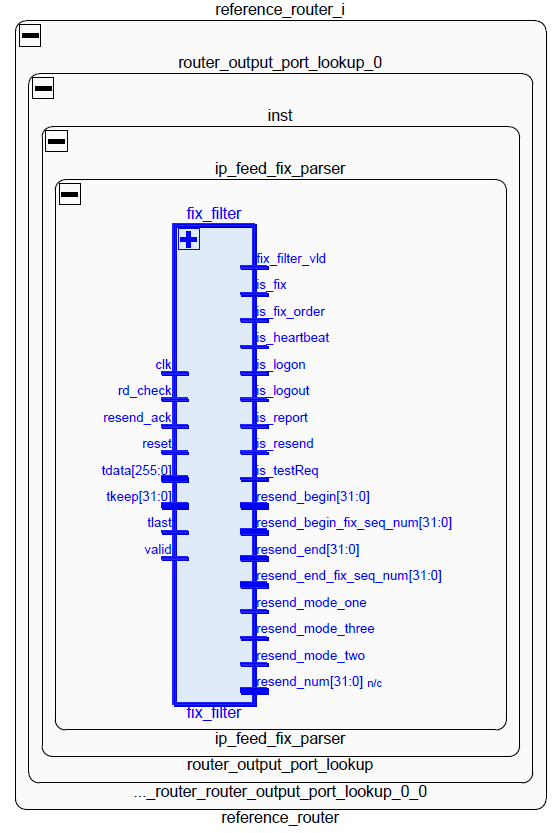
word\_IP\_DST\_HI=1時檢查是否為udp封包

word\_IP\_DST\_LO=1時檢查ESC-CODE和傳輸格式代碼是否分別為1b和06

藉由tkeep判斷封包結尾最後16bits是否為0d0a(TERMINAL-CODE)

(tdata\_reg只需存8個bit，wr\_en不需要)

fix\_filter



功能：

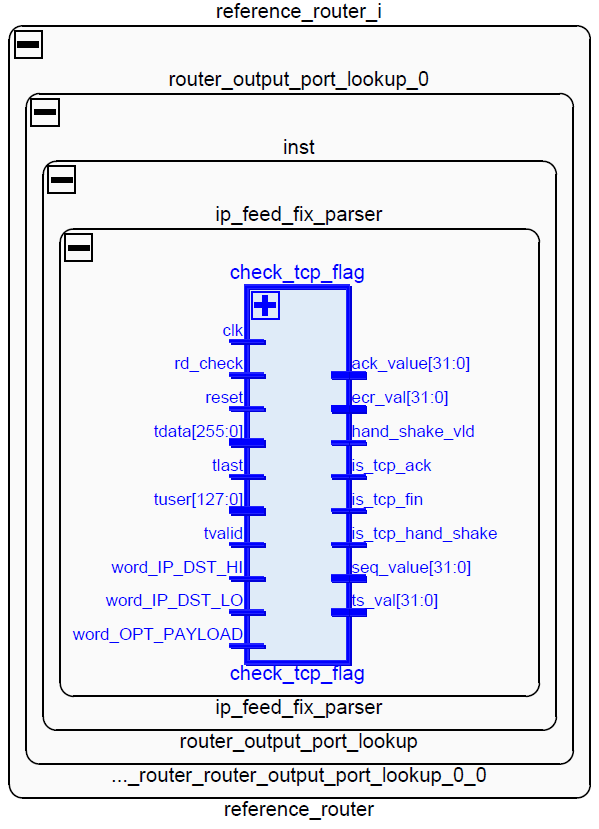
偵測fix packet並輸出交談層訊息(Session Level Messages)、應用類訊息(Application Messages)，如果是Resend Request則parse出BeginSeqNo和EndSeqNo以及輸出重送請求訊息的三種mode。

詳述：

counter表示現在是第幾個256bits，每個clk cycle將counter+1

counter=0時判斷是否為tcp packet，counter=1且為tcp packet時判斷destionation port決定是否為fix packet，因為讓fix的封包固定走59140這個port，counter=2且為fix packet時判斷MsgType並拉起對應的訊號線，counter=4且為resend request時parse出BeginSeqNo，counter=5時parse出EndSeqNo並決定重送請求訊息的mode，最後拉起check\_done讓fifo輸出結果，拉起end\_state重設。

check\_tcp\_flag



功能：

偵測tcp syn, ack packet

詳述：

當word\_IP\_DST\_HI=1時判斷是否為tcp packet，透過tuser儲存packet的長度

當word\_IP\_DST\_LO=1時判斷tcp的flags欄位並拉起對應訊號線(ack\_check後面未使用)

如果為tcp packet儲存seq\_num、ack\_num、ts\_num、ecr\_num

同時seq\_num在這裡會做更新，如果is\_ack或is\_fin將seq\_num+1，否則將seq\_num+pkt\_len-66

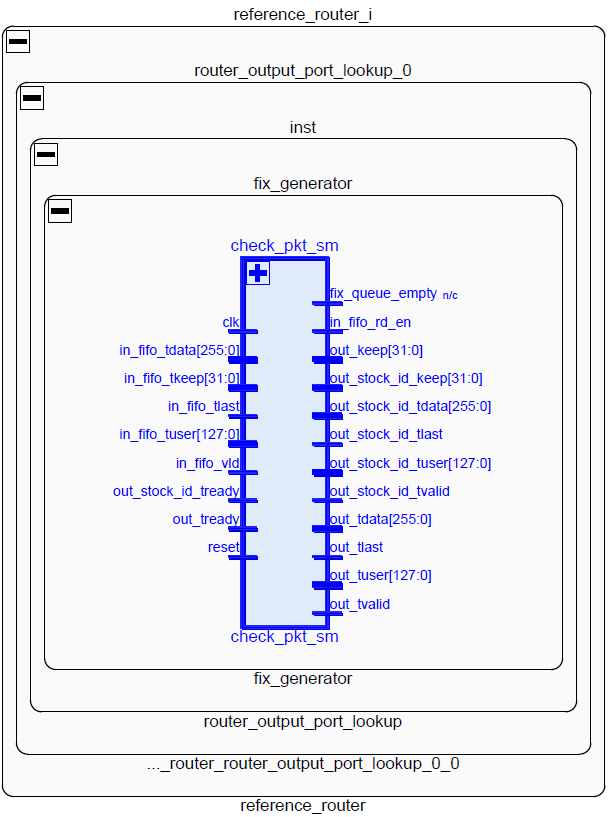
(is\_eop\_delay、is\_eop、ctrl\_prev\_is\_0未使用)

# FIX Generator – Decision Executor

說明：

* check\_pkt\_sm判斷從ip\_feed\_parser傳來的是什麼類型的封包，如果是屬於行情(UDP 格式六)或FIX Order則會送到Stock\_ID Mapping處理，否則為一般地連線封包傳至Packet Arbiter的Connect fifo，
* 送到stock\_id\_mapping處理的封包如果是FIX order，warrant ID和content會被parse出來存到fifo中，再由Warrant\_ID\_Mapping從fifo讀出資料並用Symbol(warrant ID)做cuckoo hashing後的值到warrant ID table和order content table查找，如果找到對應的ID(hash後的值)，則檢查在order content table中對應addr的資料裡的valid bit，如果為0代表content沒被存入，則將前面parse出的order content寫入order content table並將valid bit設為1
* 如果為IP feed封包，stock\_id\_mapping會parse出Stock ID並存在fifo中，內部的state machine會從fifo讀出資料並到stock id table查詢，如果match則match的data裡的begin和end傳到warrant\_id\_mapping，start到end即為warrant ID table裡面要查找index的範圍，數量為end-start+1，同時會到order content找對應的資料，並檢查valid bit是否為1，如果為1則將order content table裡的資料傳回fifo，送至FIX\_formatter產生FIX格式的cancel request封包
* pkt\_arbiter每次會檢查兩個fifo裡是否有資料有的話會把資料取出並計算checksum後傳到output queues，優先權order cancel request較高

# FIX Generator內部module功能

* check\_pkt\_sm

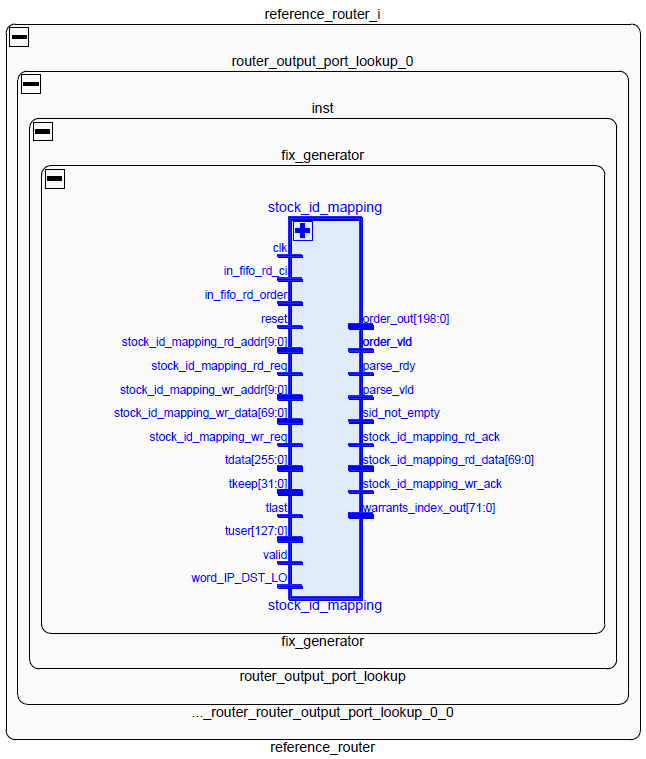
功能：

決定從ip\_feed\_fix\_parser傳來的封包要送往何處：

1. UDP格式六封包：送到Stock\_ID Mapping
2. FIX Order封包：送到Stock\_ID Mapping
3. 一般的連線封包：送到Packet Arbiter的Connect fifo
4. FIX的report封包：送到Packet Arbiter的Connect fifo

詳述：

狀態WAIT\_PREPROCESS\_RDY等待fifo有封包傳入，接著判斷tuser裡的資料決定下個狀態(這裡是看output port)，狀態PASS\_UDP\_FORMAT\_6\_PKT、PASS\_CONNECT\_PKT、PASS\_STOCK\_ID\_PKT、PASS\_REPORT\_PKT都是從fifo讀資料後傳出，差別只在送往的fifo不同

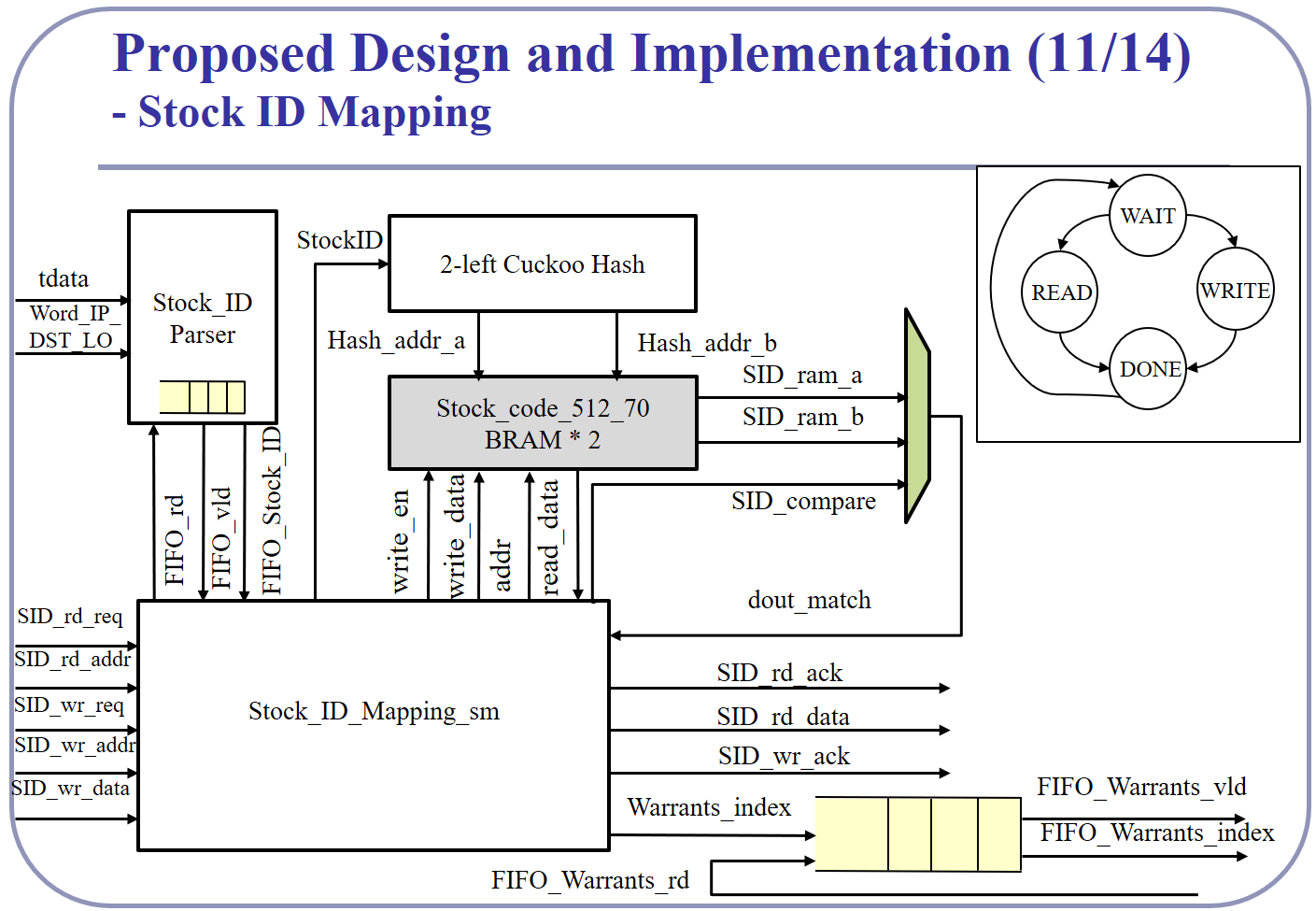
* stock\_id\_mapping

功能：

分別parse出ip\_feed和FIX order的資料作處理，如果為FIX Order封包，會parse出warrant ID與其他資料並存在fifo中，然後送至warrants\_id\_mapping處理，如果為IP Feed封包，則parse出Stock ID並存在另一個fifo中，當內部的state machine處理時判斷此fifo不為空，就會去fifo取資料並與stock\_id table比對，如果找到對應的entry則會傳回start和end兩個數字，意思為warrant table的起始index和結束index，然後傳至warrents\_id\_mapping繼續做比對的動作

詳述：

1. 這個module主要做3件事情，
2. stock code parser: parse出ip\_feed封包裡的股票代號，揭示項目註記，根據揭示項目註記取出即時行情裡的最佳五檔買價和賣價的第一檔價格，stockID(48 bits)+buy\_in(24bits)+sell\_in(24bits)，這些資料會用來讓SUME的dataflow更新stock\_code\_512x49裡面的table
3. order parser: parse出fix封包MsgType=D (New Order Single) 裡的  
   ClOrdID(96 bits)  
   OrderID(40 bits)  
   Symbol(48 bits)  
   Side(2 bits)  
   TWSE Fields(13 bits)  
   總共199 bits的資料，這些資料送往下個module做處理
4. State machine:如下圖分成4個State

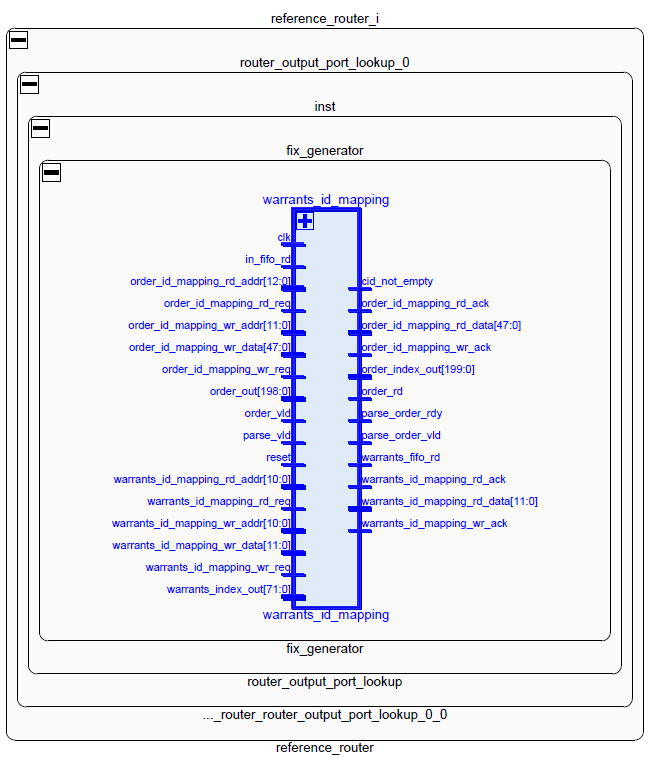
此state machine使用了current\_owner\_reg這個暫存器來lock住stock\_code\_512x70裡面的hash table，同時間只能由software(d’2)或dataflow(d’4)去讀寫

WAIT:

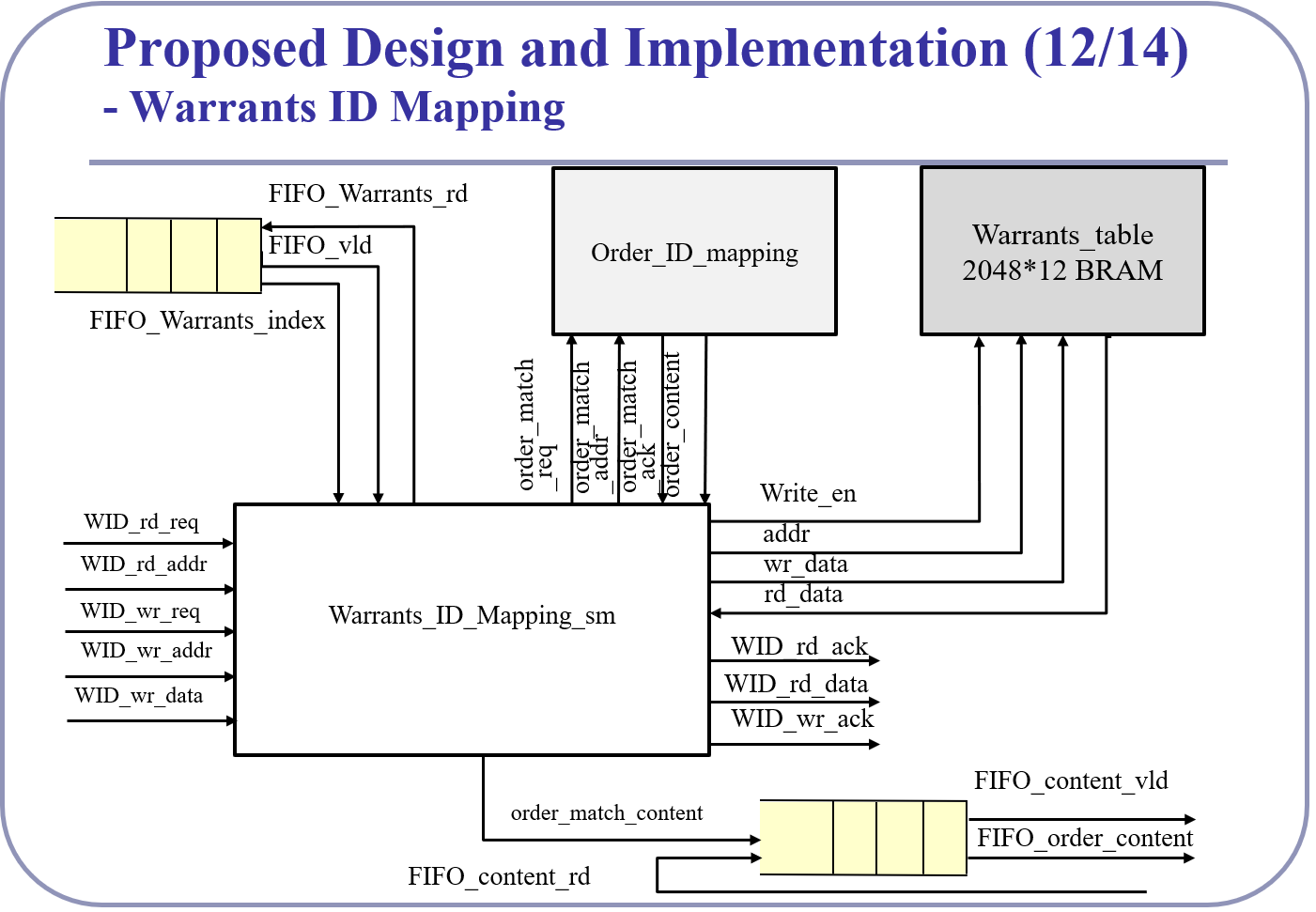
先判斷是否由software發送的request，如果是SID\_wr\_req則進到WRITE寫data，如果是SID\_rd\_req則進到READ讀data，如果software沒有request且現在stock table沒人再用，則會把剛剛ip\_feed parse出來的stockID做hash，這邊需要8個cycle才能計算完畢，然後進到READ，由於要進到BRAM讀資料所以要多一個cycle等待，

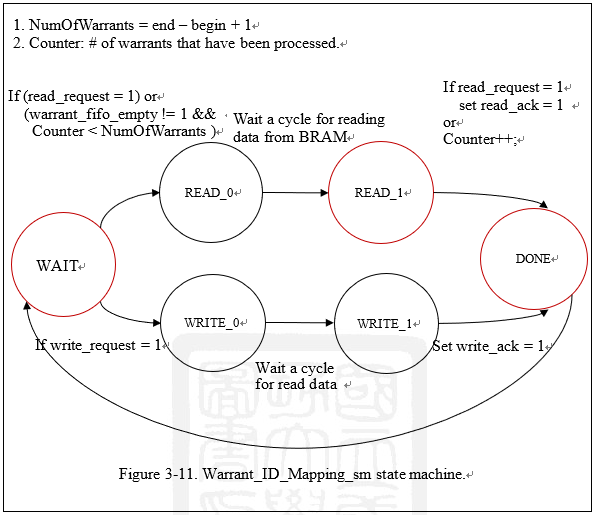
WRITE:這邊會把由software傳送的70bits資料寫到stock\_code\_512x70裡對應的位置(位置軟體會先算好)

READ:如果是由software發出的request，則將stock\_code\_512x49裡對應的entry讀出來送往下個module處理，如果是dataflow則先比較stockID是否正確(屬於stock\_code\_512x70中哪個hash table)，接著比較stock\_code\_512x49裡的買價賣價與ip\_feed parse出來的買價賣價，如果新的買價 > 舊的買價 ，新的賣價 < 舊的賣價，就更新stock\_code\_512x49裡table的值，並且把stock\_code\_512x70對應的70bits資料送往下個module。

* warrants\_id\_mapping

功能：

主要目的為從FIFO\_Warrants\_index取出begin和end，然後到warrants\_table找出對應的entry，接著再到order\_ID\_mapping裡的order\_content table取出data後，如果valid bit是1就將data存到FIFO\_order\_content中，最後傳至fix\_formater

詳述：

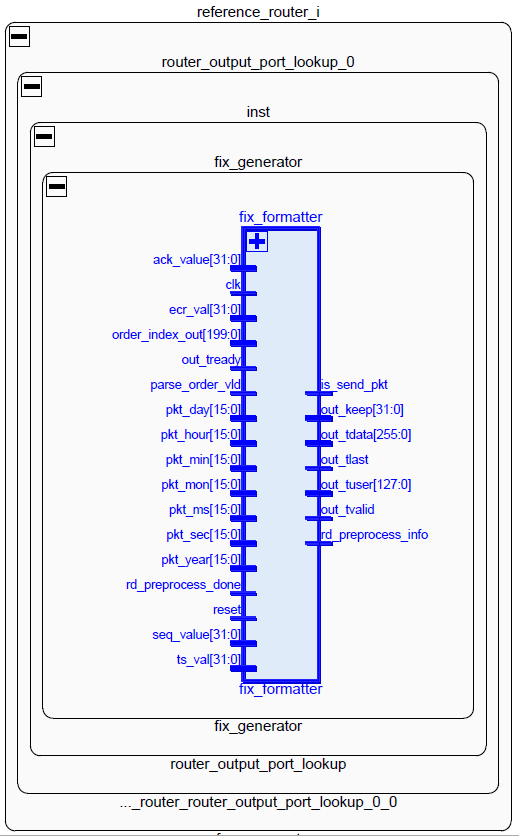
State machine跟stock\_ID\_mapping類似，主要說明不一樣的地方

WAIT：

由software發出的request做法與stock\_ID\_mapping相同，否則會從傳入的begin作為index到warrents table查找，查找次數為end-begin+1

READ：

同樣須等一個cycle(READ\_0)從BRAM讀出資料，在READ\_1從warrants table得到結果後將前面經stock\_ID\_table傳入的warrents\_ID\_out最後2 bits當作index到對應的order content table(共4個)，addr為剛剛從warrants table的結果送到order\_ID\_mapping，等到order\_ID\_mapping處理完後，判斷valid bit(order\_content的第0 bit)如果是1就把傳出的order\_content存到FIFO\_order\_content，並將counter+1。

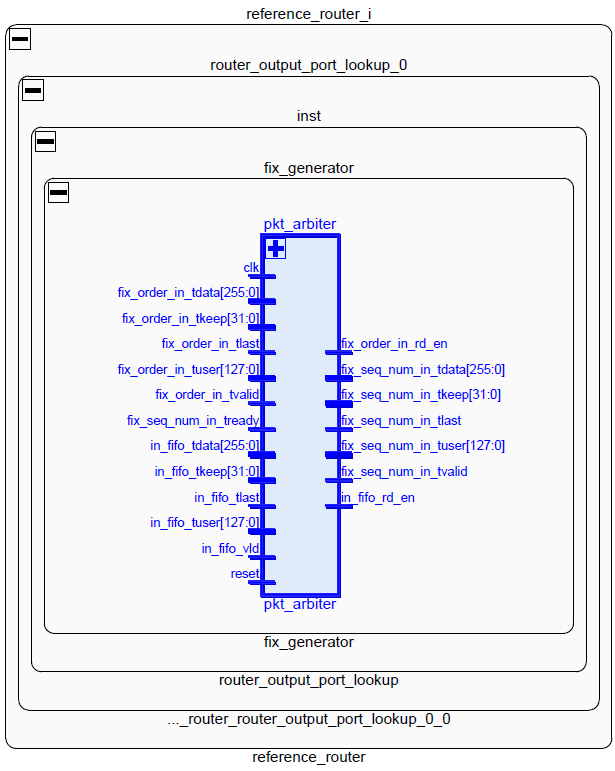
* fix\_formatter

功能：

透過傳入的order content table裡面的資料產生FIX格式的cancel request封包，但這邊先不處理checksum的計算，傳入pkt\_arbiter裡面的order cancel fifo儲存

詳述：

狀態WAIT\_PREPROCESS\_RDY等待前面的module處理完畢(stock\_id\_mapping, warrant\_id\_mapping, order\_id\_mapping)以及後面的module(pkt\_arbiter)可以接收資料時才會到後面的狀態產生FIX封包，後面的每個狀態都只有一個cycle負責填好封包的資料，資料來源有從warrant\_id\_mapping傳來的order\_index\_out(order content table data)、host端一開始設好的時間、ip\_feed\_fix\_parser裡由check\_tcp\_flag算出的ack\_value, seq\_value, ts\_value和ecr\_value，產生的格式包刮out\_tdata, out\_tvalid, out\_tlast\_next, out\_keep\_next等訊號線

* pkt\_arbiter

功能：

檢查兩個fifo(order\_cancel和connect fifo) 裡是否有資料有的話會把資料取出並計算checksum後傳到output queues，優先權order cancel request較高

詳述：

狀態WAIT\_PREPROCESS\_RDY檢查兩個fifo (order\_cancel和connect fifo)裡是否有資料且tready是否為1(計算checksum的module是否準備好)，分別進入狀態PASS\_ORDER\_CANCEL(優先權較高)、PASS\_TCP\_FIX\_CONNECT，將封包送往計算checksum的module，order cancel request需要填入seq number後計算FIX和IP的checksum，connect的封包只需要計算IP的checksum，都計算好後送往output queue

刪單交易與連線State Machine設計

* 從連線到送出Order Cancel Request之 State Machine

receive cmd from CPU

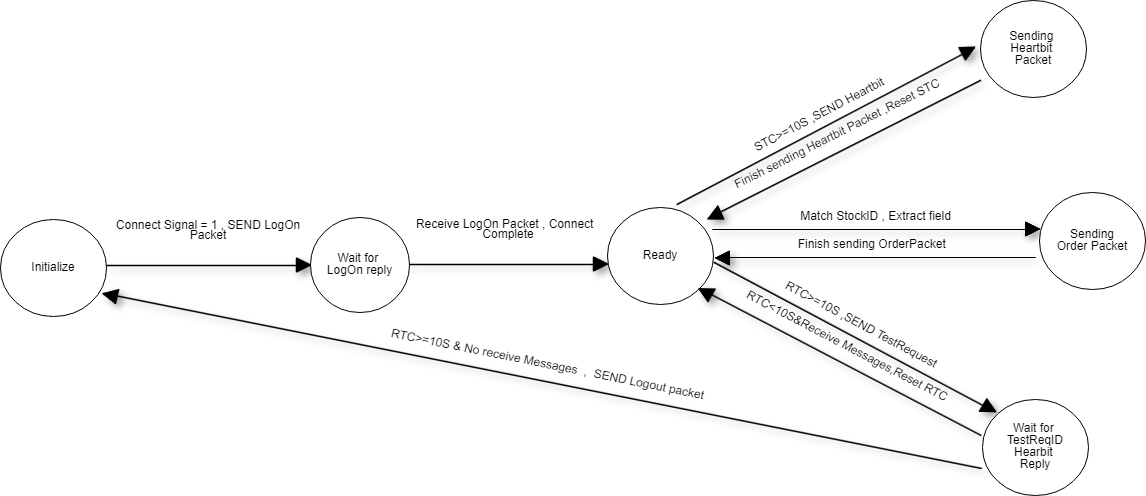
receive SYN-ACK

receive ACK

receive LOGON

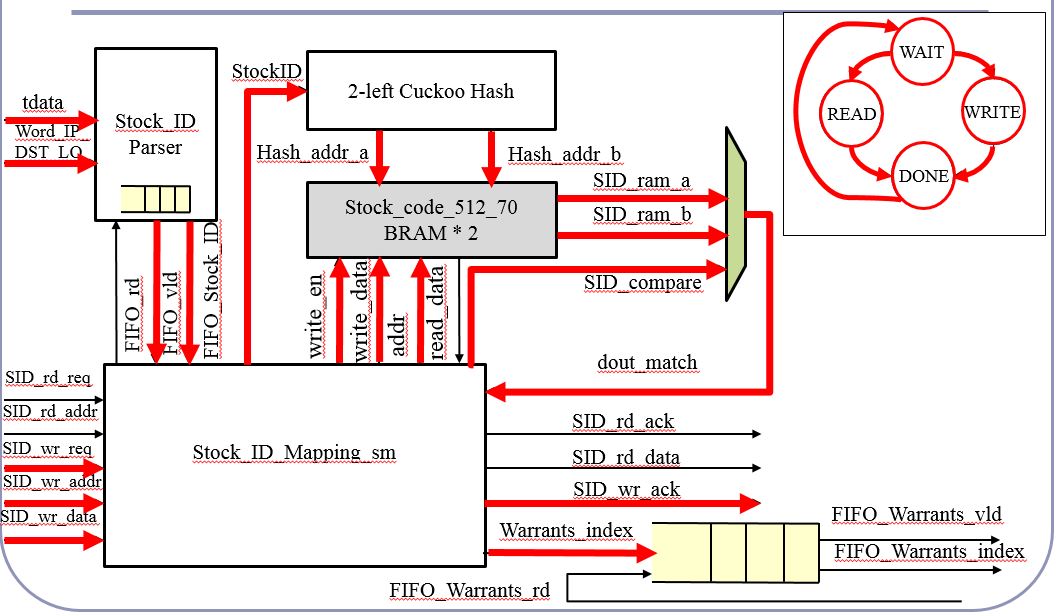
order cancel sent

receive Report

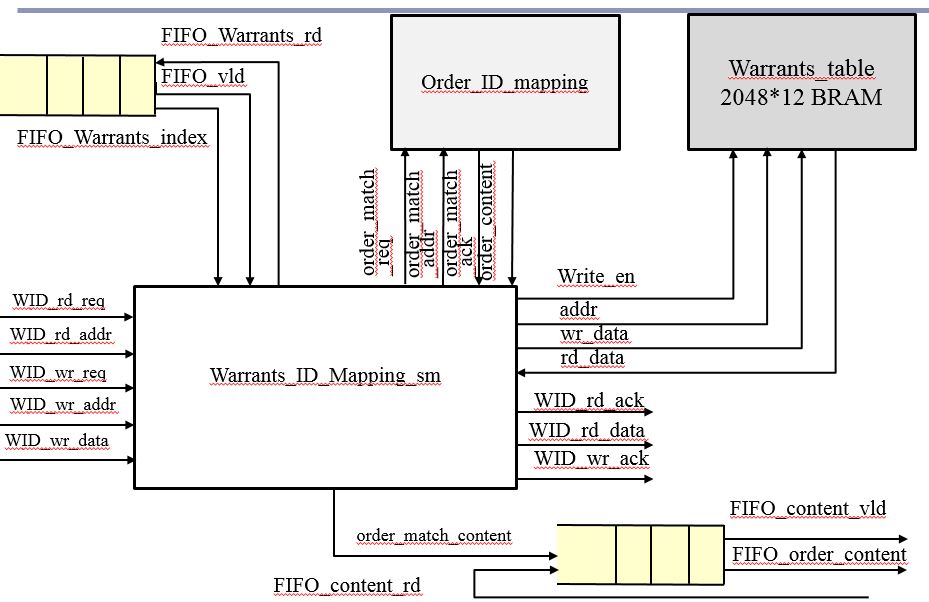
* 維持與Quick FIX 連線狀態之State Machine

Stock & Warrants ID Mapping流程設計

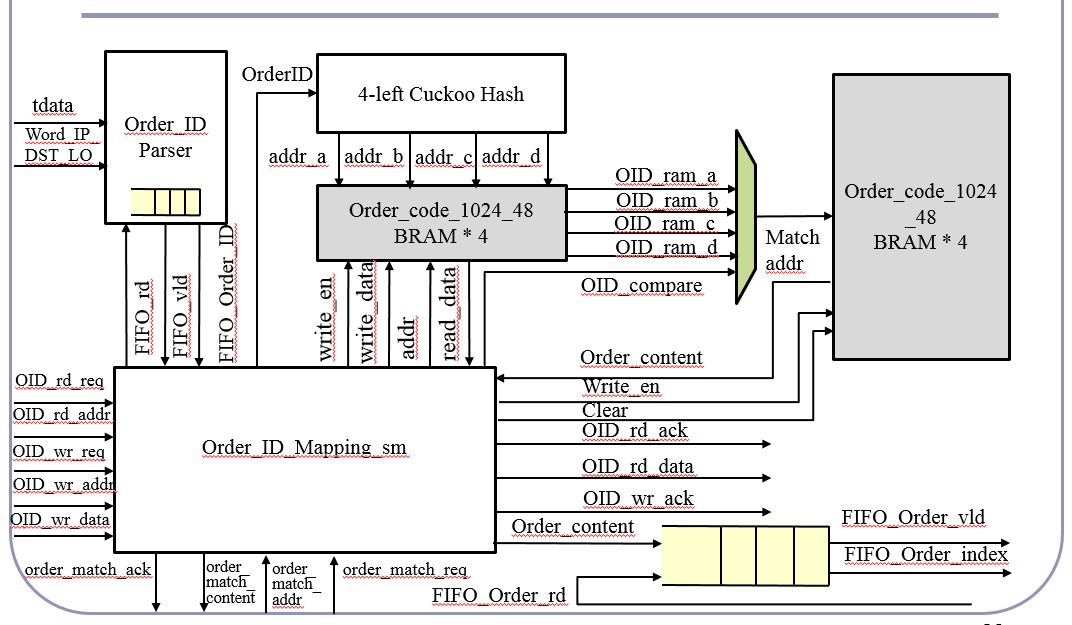
* Stock ID Mapping電路設計



* Warrants ID Mapping電路設計



* Order ID mapping電路設計



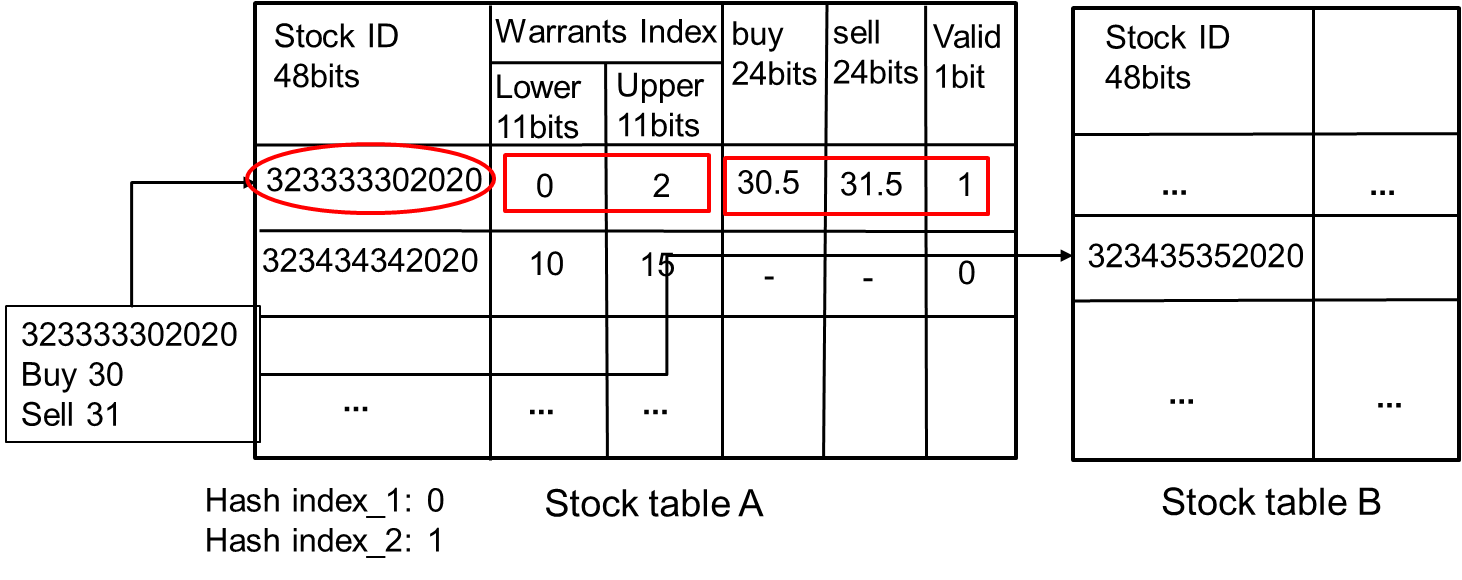
* 行情資訊之Stock ID Mapping流程

***Stock Table***

* Two 512-entries hash table
* Store Stock ID for full match and return the warrants index of warrants table

Buy Price New(30) < Old(30.5)Cancel buy order

Send {11’d0, 11’d2, 2’b01} to Next stage



* Warrants Table & Order Content Table Mapping設計

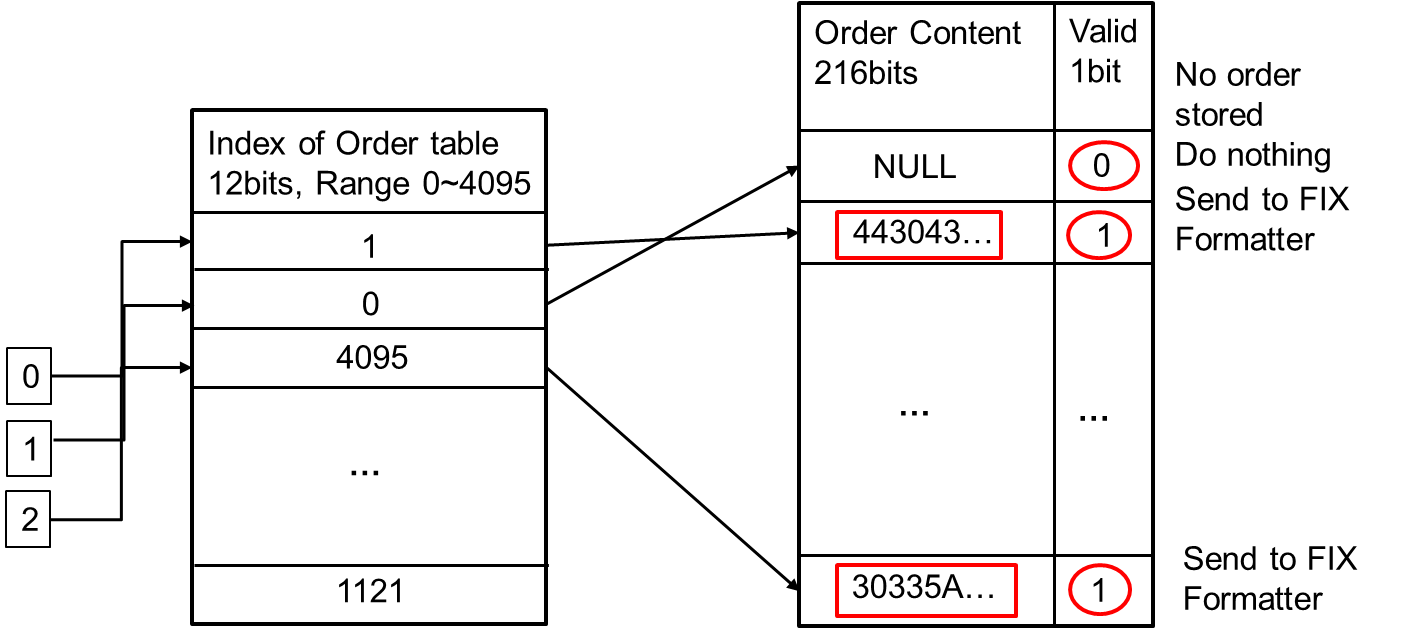
*Warrant Table*

* 1550 entries, only store index of *Order Content Table*

*Buy / Sell Order Content Table*

* 4096 entries, store buy/sell order content.

If input\_data’s lowest significant two bits == 2’b01 then Cancel buy Order



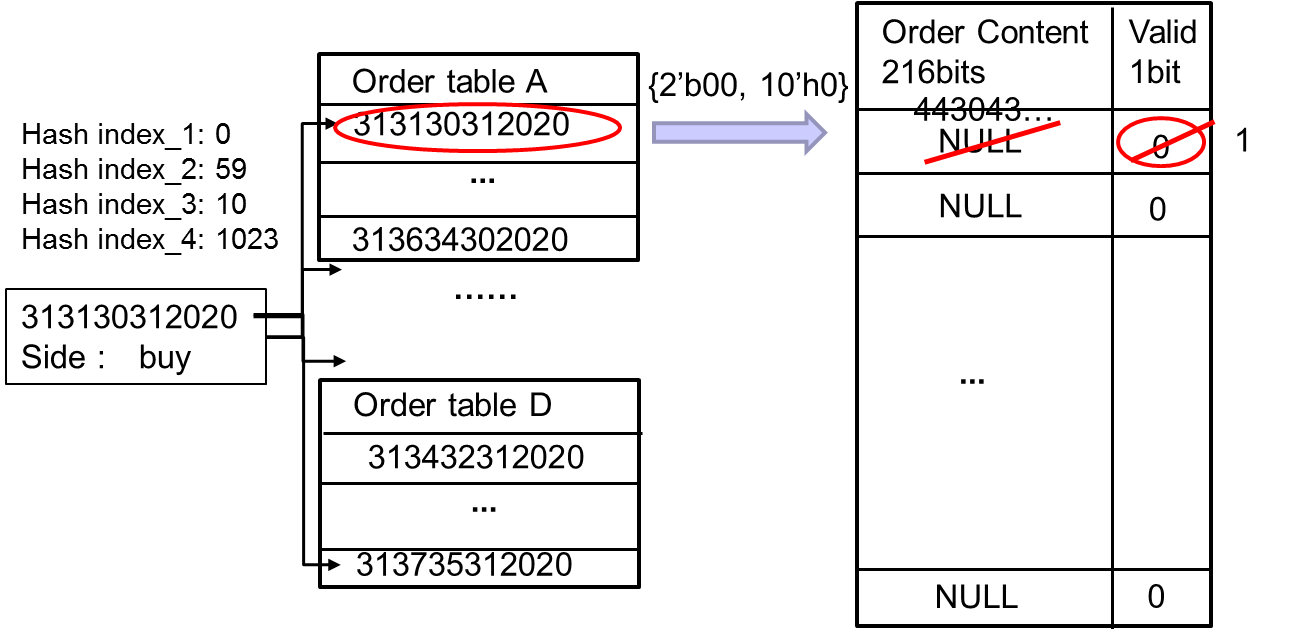
* Warrant ID & Order Content Table Mapping設計

***Order Table***

* Four 1024-entries hash table
* Store Warrants ID for full match
* Append 00, 01, 10, 11 to get the content table address

***Buy / Sell Order Content Table***

* 4096-entries table, Corresponding to order table



* Order Content Table的內容

Order的儲存，我們用價格排序的方式，買單是高→低，賣單是低→高，每欄最前面存的是SymbolID，因為同個權證的SymbolID皆相同，故我們不用重複存。接著每個Order的內容，我們存:ClOrdID、OrderID、TWSEfields，因為這些欄位皆為unique，每筆訂單皆不同。

Buy Table

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ADDR | Price(高→低) | | | | | |
| 0 | 002330 | Order1 | Order2 | Order3 | Order4 | Order5 |
| 1 | 002337 | Order1 | Order2 | Order3 | Order4 | Order5 |
| 2 | 002354 | Order1 | Order2 | Order3 | Order4 | Order5 |
| 3 | 002357 | Order1 | Order2 | Order3 | Order4 | Order5 |
| 4 | 002402 | Order1 | Order2 | Order3 | Order4 | Order5 |
| 5 | 002403 | Order1 | Order2 | Order3 | Order4 | Order5 |
| … | … | … | … | … | … | … |
| 4096 | … | … | … | … | … | … |

Sell Table

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Price(低→高) | | | | | |
| 0 | 002130 | Order1 | Order2 | Order3 | Order4 | Order5 |
| 1 | 002137 | Order1 | Order2 | Order3 |  |  |
| 2 | 002154 | Order1 | Order2 | Order3 | Order4 | Order5 |
| 3 | 002157 | Order1 | Order2 | Order3 | Order4 |  |
| 4 | 002502 | Order1 | Order2 |  |  |  |
| 5 | 002503 |  |  |  |  |  |
| … | … | … | … | … | … | … |
| 4096 | … | … | … | … | … | … |

* Order Price Table的內容

為了儲存多筆價格不同的Order，價格我們也必須儲存在BRAM中，為了加速儲存，我們將BRAM儲存Table分成OrderContentTable和OrderPriceTable，每當訂單進FPGA要儲存時，我們先到OrderPriceTable取出該權證所有儲存的訂單價格，接著用價格排序的方式，買單是高→低，賣單是低→高，排序完後，更新PriceTable和ContentTable，Price Table每欄最前面存的是Valid Bits，Valid Bits共3bits，因為我們總共最多存5個訂單，Valid Bits是用來表示此檔權證的買/賣單儲存數。

Buy Price Table

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ADDR | Price(高→低) | | | | | |
| 0 | Valid Bit | Price1 | Price2 | Price3 | Price4 | Price5 |
| 1 | Valid Bit | Price1 | Price2 | Price3 | Price4 | Price5 |
| 2 | Valid Bit | Price1 | Price2 | Price3 | Price4 | Price5 |
| 3 | Valid Bit | Price1 | Price2 | Price3 | Price4 | Price5 |
| 4 | Valid Bit | Price1 | Price2 | Price3 | Price4 | Price5 |
| 5 | Valid Bit | Price1 | Price2 | Price3 | Price4 | Price5 |
| … | … | … | … | … | … | … |
| 4096 | … | … | … | … | … | … |

Sell Price Table

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Price(低→高) | | | | | |
| 0 | Valid Bit | Price1 | Price2 | Price3 | Price4 | Price5 |
| 1 | Valid Bit | Price1 | Price2 | Price3 |  |  |
| 2 | Valid Bit | Price1 | Price2 | Price3 | Price4 | Price5 |
| 3 | Valid Bit | Price1 | Price2 | Price3 | Price4 |  |
| 4 | Valid Bit | Price1 | Price2 |  |  |  |
| 5 | Valid Bit |  |  |  |  |  |
| … | … | … | … | … | … | … |
| 4096 | … | … | … | … | … | … |

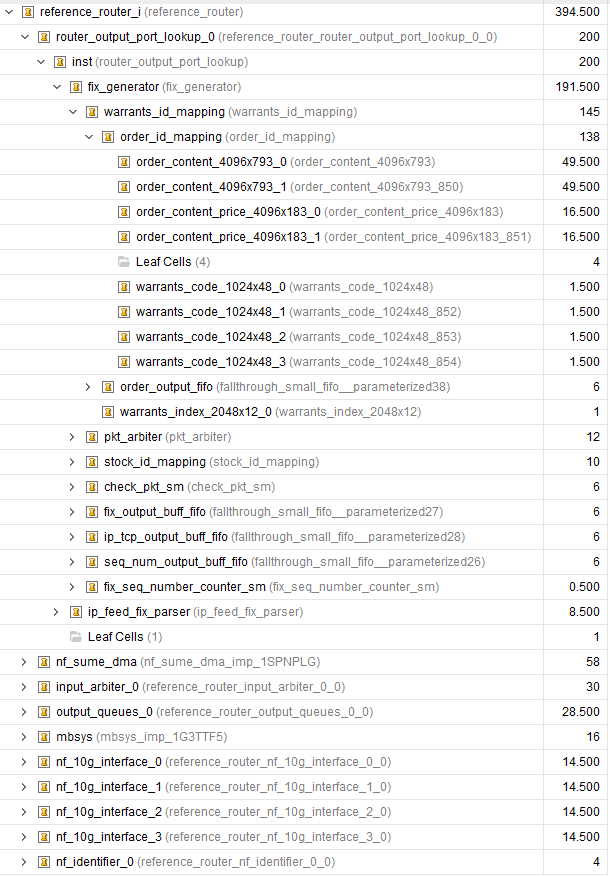
FPGA資源使用量

底下為目前整個專案的FPGA資源使用量:當中我們較關心BRAM的使用量，因為其他資源使用量基本上很少，而BRAM為我們儲存訂單資訊的Memory，故使用量是我們很重視的。



BRAM詳細用量與未來使用估算

底下是BRAM使用量的詳細統計，比起之前的專案，在warrants\_id\_mapping中的order\_output\_fifo BRAM使用量又更小了(之前是374塊36KB)，因為我將entry從2的16次方縮小成2的10次方。所以比起之前又節省了368塊36KB BRAM。



SUME上共1470塊36KB大小的BRAM

目前專案總共使用311塊36KB大小的BRAM

|  |  |  |
| --- | --- | --- |
|  | 36KB BRAM | 使用率 |
| 專案其餘必要的BRAM | 262.5 | 17.857% |
| Order content Table(Buy&Sell) | 99 | 6.735% |
| Order price Table(Buy&Sell) | 33 | 2.245% |
| 剩餘 | 1075.5 | 73.163% |
| 總共 | 1470 | 100% |

若以目前使用的Bram去估算最大可以存的order數:

目前扣除其他必要的BRAM使用，還剩下:

1470-262.5 = 1207.5塊36KB的BRAM可以使用，

Order Content Table : 8為symbol ID 149為1個order共需要存的總bits數

Price Table : 3為 validbits ，36為Price整理後可得下列公式 : N為一欄所存的Order數((48 + (149\*N) )+ (3 + (36\*N))/36)< 1207.5整理一下 : 51 + 185\*N < 1207.5\*36=> 185\*N <43470-51 => 185\*N < 43419 => N <

234.697 ，N最大整數為 234

故我們可以算出理論上，當我們有1540個權證ID時，同個Symbol ID 的權證最多買賣可以各存234個訂單。

根據公式，我們可以推測去以下幾種組合，在不同權證發行數時，我們可以儲存的訂單數:

* + 權證數量—訂單數組合表

|  |  |  |
| --- | --- | --- |
| 權證數量 | BRAM列數 | 一個權證可存不同價格之買賣訂單數 |
| 1500 | 4096 | 234 |
| 3000 | 8192 | 117 |
| 4500 | 16384 | 53 |
| 6000 | 32768 | 21 |

* + 環境架構圖

PC1

NetFPGA SUME (IP:140.116.82.189)

MAC0

MAC2

MAC3

MAC1

FIX Order Report封包、Order Cancel Reject封包、Session Reject封包

FIXOrderCanel封包、TCP/FIX 連線封包

UDP格式6行情封包

FIX Report封包、TCP/FIX 連線封包

FIX Order封包

PC2

Send行情

Server2

證交所

(IP:140.116.82.179)

PC4

Quick Fix

Server1

證交所

(IP:140.116.82.185)

PC3

Send Order

Request

網卡1

(IP:140.116.82.187)

Recieve OrderCancel Report

網卡2

(IP:192.168.1.121)

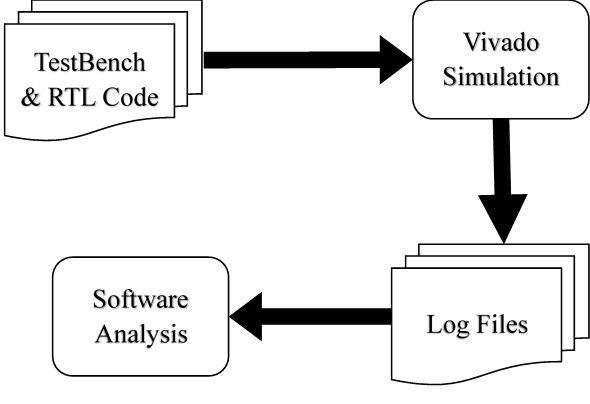
軟體測試

* + 前言

由於本計畫實作在NetFPGA SUME開發平台上，NetFPGA SUME的內部運轉時脈頻率為160MHz，換算後一個時脈週期為6.25ns。在我們的設計中，封包的處理不會超過200個時脈週期，也就是最多在1.2ms左右的範圍，而在這個時距下是非常難以使用一般的設備來量測整體設計的處理延遲以及吞吐量。因此為了能夠有效評估整體設計的效能，我們採用了軟體模擬的測試方式，使用這種方式我們可以得到整體設計的精確延遲以及驗證設計的正確性，以下會介紹軟體模擬的測試方法。

* + 測試方法

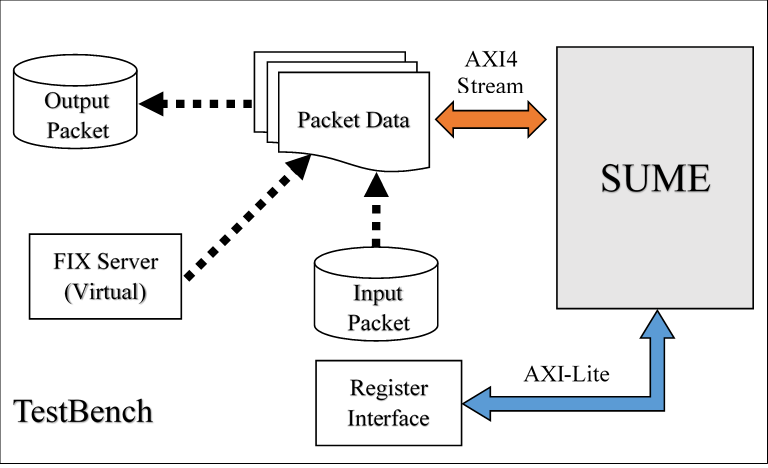
軟體模擬的測試方法主要透過Xilinx Vivado的Simulation功能，整體流程如下圖所示，開發者必須事先定義好符合需求的Testbench以及製作對應的測試資料，將Testbench與開發者設計的RTL Code經由Vivado Simulation功能進行模擬，得到電路設計的波形圖以及TCL Console所輸出的測試紀錄，波形圖可用作除錯，而測試紀錄可以再經由人工或軟體程式來計算整體設計的細節數據。



軟體模擬流程示意圖

* + 測試檔案

測試檔案包括Testbench、測試資料以及設計好的RTL Code，其中RTL Code為NetFPGA SUME中Datapath使用到的所有RTL Code。除此之外，開發者必須自行設計Testbench，模擬真實SUME運行使封包收送以及Host PC下指令的行為，將其寫入Testbench並產生對應的測試資料(封包、指令)，同時也必須在Testbench及RTL Code加註額外的Verilog程式碼輸出訊息至TCL Console，以獲得細部的測試數據。



TestBench檔案示意圖

* + 詳細流程

1. 撰寫Testbench檔案

Testbench檔案的撰寫可以參考程式碼裡的TWSE\_tb.v檔案，內容主要實現了上述示意圖的內容，將轉換後的封包資料及指令資料傳入SUME Datapath中，同時透過$display("…", $time);指令來輸出必要的訊息至TCL Console中。

2. 產生測試資料

測試資料的產生主要包括封包資料以及Host PC的指令資料，SUME Datapath分別採用AXI4 Stream及AXI-Lite這兩種格式來傳輸這兩個資料。通常我們所使用的封包資料格式為PCAP，指令資料則是使用者在執行時透過CLI軟體介面所下的指令，這兩種資料都必須透過額外的處理來轉換成對應的AXI4 Stream及AXI-Lite格式，其轉換方法可參考程式碼裡的的pcap-axi.c與cli-axi.c。

3. 加入額外語法

開發者可以在所設計的RTL Code中加入額外的$display語法，讓軟體模擬時可以在TCL Console輸出對應的資訊，其用法類似於在C語言程式碼中加入printf()輸出資訊。

4. 建立Xilinx Vivado專案

有了RTL Code、Testbench及測試資料後我們必須將其新增至Xilinx Vivado的專案中，其中RTL Code必須加入到Design Sources，Testbench及測試資料必須加入到Simulation Sources中。

5. 執行Simulation

在執行Simulation前我們需要在Simulation Settings中將xsim.simulate.runtimes設定成所需的模擬時間，必須注意如果時間設定太短會無法完整模擬。

6. 觀察波形圖

在Simulationt成功執行後Vivado會給出電路模擬時的完整波形圖，開發者可以透過檢視各個訊號線在每個時脈週期的行為來驗證設計的正確性並找出潛在的錯誤。

7. 紀錄TCL Consle輸出

Simulation執行後下方的TCL Console根據我們在Testbench以及RTL Code裡加入的$display語法印出對應的內容，我們可以透過分析其內容來得到想要的測試數據，內容可參考附件的Simulation Log資料夾。

結案數據

* + 軟體模擬數據

下表為經由軟體模擬所得到的測試數據，在表中列出了五種類型的封包處理延遲。第一種為Order封包，在接收到Order封包後會將其存入我們所設計的Order Table，其延遲時間由Datapath完整收到Order封包之時開始計算至Order被存入Table中為止。接下來的四種封包為UDP行情封包，分別列出了四種不同情況的處理延遲，First Out代表著一個有符合策略的行情並送出第一個刪單，而Follow-up代表著接著送出的第二、第三、…個刪單；Not format-6代表著不符合格式六要求的行情封包，Not Match代表著沒有符合策略的行情封包，以上四種計算方法一樣是由封包進入Datapath到封包丟棄或刪單封包傳出。

|  |  |  |
| --- | --- | --- |
| 封包類型 | 延遲 (cycles) | 延遲  (ns) |
| Order | 43~47 | 268~293 |
| IP\_Feed (First Out) | 98~105 | 616~656 |
| IP\_Feed (Follow-up) | 65~67 | 406~418 |
| IP\_Feed (Not format-6, drop) | 10~14 | 62~87 |
| IP\_Feed (Not Match, drop) | 39~43 | 243~268 |

軟體模擬數據(2) (多筆買賣單)

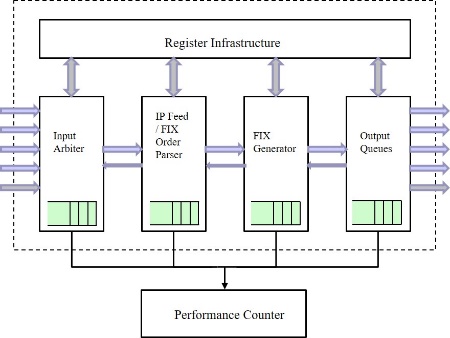
|  |  |  |
| --- | --- | --- |
| 封包類型 | 延遲 (cycles) | 延遲  (ns) |
| Order | 42~46 | 262~287 |
| IP\_Feed (First Out) | 98~105 | 616~656 |
| IP\_Feed (Follow-up) | 65~67 | 406~418 |
| IP\_Feed (Not format-6, drop) | 10~14 | 62~87 |
| IP\_Feed (Not Match, drop) | 39~43 | 243~268 |

軟體模擬數據(1) (單筆買賣單)

* + 單筆買賣單 vs 多筆買賣單

上表的兩種數據主要差異在於Order Content Table能夠儲存Order量的多寡，過去我們所使用的Order Content Table僅能儲存買單與賣單各一，其模擬數據數表(1)所示；而現今我們採用的是能夠存取最多五筆買單及五筆賣單的架構，其模擬數據如表(2)所示。兩者在處理IP\_Feed封包的延遲數據並無不同，唯一的差別在對於Order封包的處理時，多筆買賣單的架構必須多花一個時脈週期來完成排序並存進Order Content Table中。

* + 硬體測試數據



硬體測試架構

硬體測試數據主要透過在Datapath中加入Counter計算每次實驗所花費的時脈週期數，計算方法為使用硬體電路偵測Datapath中各個FIFO的使用狀況，如果任何FIFO不為空，就代表電路還在處理封包，則Counter數加一，透過Register Infrastructure取得實驗前與實驗後的Counter數字，相減即可得到所需的硬體測試數據。

由於無法在執行時間定義一個封包會產生軟體模擬所述的四種行為之一，因此硬體測試數據僅能提供Order類型的封包處理延遲。

|  |  |
| --- | --- |
| 封包類型 | 延遲(ns) |
| Order | 368.75 |

硬體測試數據

* + 延遲數據內容

上述兩段所給出的軟體模擬數據與硬體測試數據，均僅限於NetFPGA SUME內部Datapath之延遲計算，並未考慮到實際硬體上Datapath外如物理傳輸介面等電路之延遲。除此之外，由於軟體模擬數據僅計算Datapath的Input Arbiter至FIX Generator，因此硬體測試數據相較於軟體模擬數據，還會多出Output queue等相關電路之延遲，實際測試出來的Order封包處理延遲會多於軟體模擬數據約10個時脈週期的時間。

計畫成果

* 證交所自訂IP行情傳輸電文解析
* 與證交所之TCP & FIX4.4 之連線維持系統設計
* 與證交所之FIX4.4刪單請求交易
* 硬體效能量測
* 模擬測試功能(含Latency量測)

參與人員

主持人:張燕光教授

碩士研究生助理:李俊豫、王偉立、黃義方、林子傑、林宇翔、林鈺航

凱基代表: 107/1~ 目前 Kevin Chiao 喬振桓

106/4~106/12 Stan Su 蘇力群

附件

1.NetFPGA 1G Document

2.NetFPGA 1G to SUME Porting Document

3.NetFPGA 1G 環境建置

4.NetFPGA SUME 環境建置

5.NetFPGA Driver安裝與常見問題排除方法

6.NetFPGA SUME刪單系統操作手冊

7.會議PPT

NetFPGA 1G Document

**Project Name: Reference Router**

* **NetFPGA Top Level**

此Reference Router Documentation的環境是NetFPGA1G的板子，底下為Reference Router在整塊板子下的基本系統圖:

* **Overview of NetFPGA1G**

**PHY**

**PHY**

**PHY**

**RGMII**

**NETFPGA CORE**

**DDR2**

**SDRAM**

**DMA**

**CPCI**

**SRAM**

**DDR2**

**Controller**

**PHY**

**PHY**

**PHY**

**PHY**

**Virtex II Pro-50**

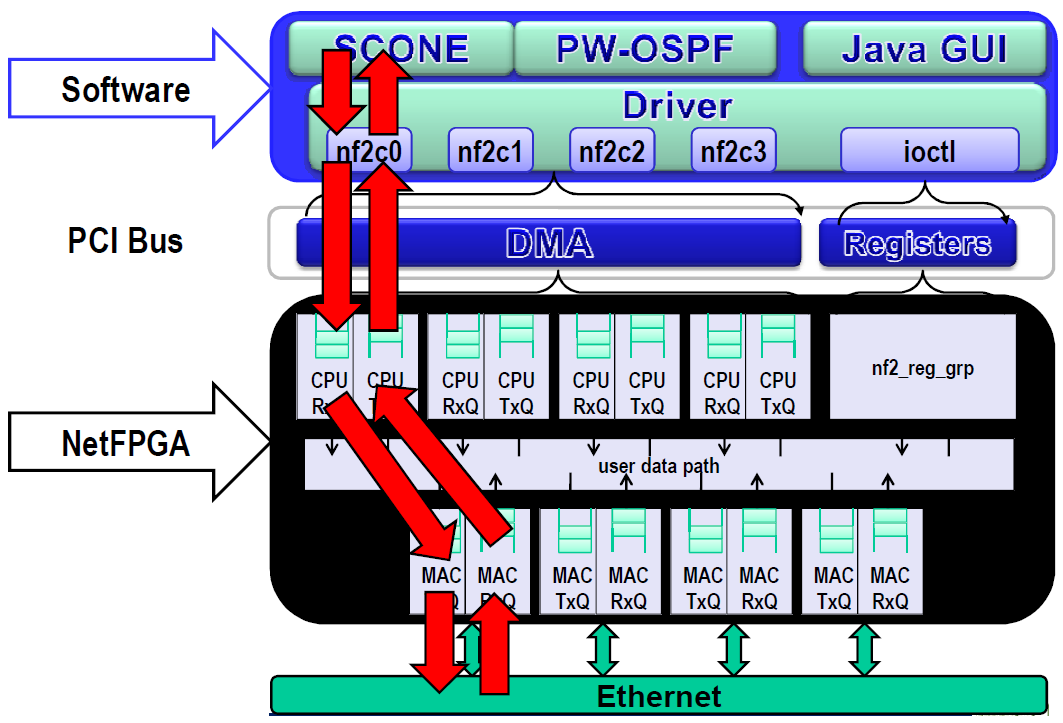
說明:

4個1G的Port完成Ethernet connect加上 PHY晶片完成Physical Layer的Data 處理；中間NetFPGA CORE主要由UserFPGA晶片來完成Packet Pipeline處理和Forwording，即Routing Forwording Engine；ControlPCI外接的是PC端，PC端的Application Layer 軟體用來實現各項網路協定(TCP、UDP…)，並對Router進行配置和管理，同時還處理那些不能正常Forwording的Packet(包含帶IP optionField和IP Header驗證錯誤的Packet)，即RoutingEngine；UserFPGA和PC端透過CPCI交換訊息和傳遞Packet。RefereneceRouter使用SRAM來實現Packet的

Output Buffer。

* **Exception Packet Path**

從整個軟硬體結構來看，由於我們希望透過ReferenceRouter 處理Packet，故也要先了解Packet在Reference Router的處理流程:



說明：

從上到下依次為:

1. PC上的軟體─分為User Interface和Driver兩部分。
2. PCI Port ，包括DMA和Register兩部分
3. NetFPGA，主要包含UserFPGA晶片上的Hardware設計和PHY的Data處理，我們主要討論的是前者

* **NetFPGA Core**

基於Reference Router專案，其中最重要的便是了解NetFPGACORE的運作，故我們先撇除其餘元件，將NetFPGA Core的內容結構設計仔細檢視：

* **NetFPGA Core Design**

**DMA Controller**

**MDIO Controller**

**User Data Path**

**SRAM**

**Controller**

**CPCI**

**Controller**

**MAC Queues**

**Register**

**Group**

**MAC Queues**

**MAC Queues**

**MAC Queue**

**CPU Queues**

**CPU Queues**

**CPU Queues**

**CPU Queue**

我們從Trace NetFPGA code(NetFPGA/lib/Verilog/core/nf2/nf2\_core.v)，來了解Reference Router的硬體設計:

* **Reference Router Hardware Design**

**nf\_dma**

**sram\_arbiter**

**MAC Queues**

**MAC Queues**

**MAC Queues**

**nf2\_mac\_grp**

**CPU Queues**

**CPU Queues**

**CPU Queues**

**Cpu\_dma\_queue**

Input\_arbiter

Output\_port\_lookup

Output\_queues

**SRAM**

**nf2\_mac\_grp**

從上圖可知User data path 是由三個Module組成(Input\_arbiter、Output\_port\_lookup和

Output\_queues)，User data path是主要分析、處理Packet的所在，故後面我們會特別分

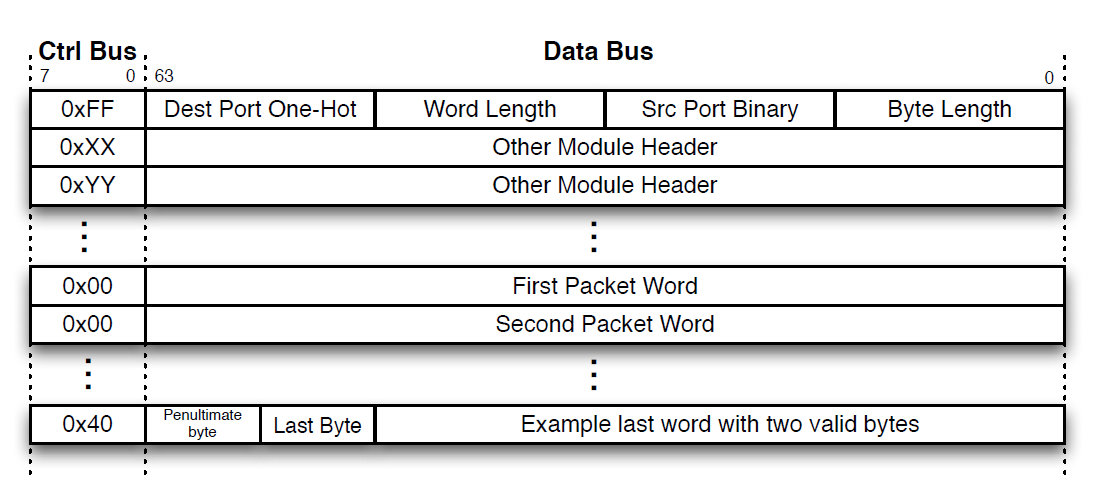
析包含在其中的Module功能。

若將ReferenceRouter包含的Ｍodule檔案做一個完整的Ｈierarchy的整理，方便Trace Code時能較有效率，如下：

* **Hierarchy File**

了解完整個ＲferenceRouter的軟硬體架構和Ｍodule組成後，我們接下來要進入的是較底層的Ｐacket格式分析：

* **Inter-Module Communication格式**



說明：

由於複雜系統中的Module要互相通信，Reference Router硬體設計利用了OSI參考模型，在Packet data path上傳輸的Packet也用一種特殊的格式，如下圖:從nf2\_mac\_grp開始，每個module都可以修改Packet word，同時也可以在Packet前端添加一個64bit的module header ，不同的module header用Ctrl Bus來區分，屬於後面Pipeline Stage的module可透過讀取這個module header來和前面Pipeline Stage的module通信。

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Ctrl Bus** | **Data Bus** | | | | | | | | |
| **2 bytes** | **2 bytes** | | **2 bytes** | | **2 bytes** | | **2 bytes** | | |
| 0xFF | Destination port | | Word length | | Source port | | Byte length | | |
| 0x00 | Dst MAC address | | | | | | Src MAC address | | |
| 0x00 | Src MAC address | | | | Ethertype | | IP ver | Head len | ToS |
| 0x00 | Total Length | | | Identification | Flags | Fragment offset | Time To Live | | Protocol |
| 0x00 | Header Checksum | | | Src IP address | | | Dst IP address | | |
| 0x00 | Dst IP address | | | Src Port | Dst Port | | Sequence number | | |
| 0x00 | Sequence number | | | Acknowledgement number | | | Header length | | Flags |
| 0x00 | Window size | | | Checksum | Urgent pointer | | Options | | |
| 0x00 | Options | | | | | | | | |
| 0x00 | Options | | | Payload | | | | | |
| 0x00 | ……. | | | | | | | | |
| 0x00 | …… | | | | | | | | |
| 0x80 | Last Byte |  | | | | | | | |

* **Packet實際擺放資料之例子**

**說明：**

Data Bus上的內容透過Ctrl\_Bus來區分，Ctrl Bus有三種定義:

1. 0xFF 表示Data Bus上的內容為nf2\_mac\_grp添加的module header

2. 0x00表示Data Bus上的內容為Packet word

3. 0x80表示Data Bus上的內容為Packet的last Byte位於Data Bus的第一個位置(eg:若0x40則表示位於第二個位置)

* **Data link Layer in NetFPGA**

Reference Router中處理Data Link Layer的處理module為nf2\_mac\_grp，而nf2\_mac\_grp包含了另外三個module: rx\_queue、tx\_queue和mac\_grp\_regs，rx\_queue和tx\_queue只是Packet在module內部的流向不同，功能基本是一樣的。我們挑rx\_queue來進行解說。

* Rx\_queue

此Ｍodule主要工作為根據Clock Rate配合FIFO來接收自入口MAC進入的Packets，由於此Ｍodule涉及不同Clock Field的Data同步問題：像是TEMAC EdgeClock為125/12.5/1.25MHz的rxcoreclk，user\_data\_path EdgeClock為System clock (clk)。Module內由pulse\_sychronizer來實現3個控制信號rx\_pkt\_bad、rx\_pkt\_good和rx\_pkt\_dropped的跨ClockField同步。底下介紹在不同的Ｅdge Clock下的Ｓtate Machine運作：

* **Rxcoreclk**

一個rxcoreclk下的Ｓtate Machine在FIFO有足夠的空間時會將Ｐacket寫入gmac\_rx\_fifo，在Packet尾端添加eop(end of packet，eop為1時表示這個是Packet的最後一個Data。)標記，同時將Packet State和length寫入pkt\_chk\_fifo ，底下為簡化後的seudo code:

**Case(rx\_state)**

**RX\_IDLE: Begin**

**Do something**

**end**

**RX\_RCV\_PKT: Begin**

**Do something**

**End**

**RX\_WAIT\_GOOD\_OR\_BAD: Begin**

**Do something**

**End**

**RX\_ADD\_PAD: Begin**

**Do something**

**End**

**RX\_DROP\_PKT: Begin**

**Do something**

**End**

**Endcase**

**說明：**

第一個State確認是否還有空間能接收Packet

第二個State接收Packet並確認Ｐacket的Size

第三個State判斷Frame的好壞

第四個State補上最後一個Word中缺少的bytes(因為我們讀Data用的是Word不是Bytes)

第五個State完成丟棄Packet的動作

* **Clk**

一個User\_data\_path的EdgeClock　clk下的State Machine，所做的處理為當pkt\_chk\_fifo不是空的和Packet沒有錯誤的狀況下，從gmac\_rx\_fifo中讀取出Packet並送出。底下為簡化後的seudo code:

**Case(out\_state)**

**OUT\_WAIT\_PKT\_AVAIL: Begin**

**Do something**

**End**

**OUT\_LENGTH: Begin**

**Do something**

**End**

**OUT\_WAIT\_PKT\_DONE: Begin**

**Do something**

**End**

**Endcase**

**說明：**

第一個狀態等待可用的Packet

第二個狀態添加Packet的Length資訊

第三個狀態正常Forwording Packet

* **User Data Path**

大致解說完rx\_queue中重要的功能後，接著要將重點放在User Data Path，此為Reference Router專案主要設計的地方，我們將User Data Path的系統流行圖特地畫出來看:

**Input Arbiter**

**Output Port**

**Lookup**

**Output Queues**

**Register**

**Master**

**說明：**

1. **Input Arbiter檢查並接收Input queues的Packets**
2. **Output Port Lookup則是Router的Forwording Engine**
3. **Output Queue將Packet根據Packet的Control Word傳送給對應的Queue**

* **Input Arbiter**

Input\_Arbiter從８個Packet Buffer Queue(4個為nf2\_mac\_grp和4個為cpu\_dma\_queue)中讀取Packet，其Module用一個簡單的State Machine來控制Packet的讀取，底下為簡化後的seudo code:

Case(state)

IDLE: Begin

Do something

End

WR\_PKT: Begin

Do something

End

Endcase

**說明：**

在IDLE狀態判斷8個in\_arb\_fifo中的哪一個為非空，即對應Input Queue的Packet有效，然後從相對應的in\_arb\_fifo中讀取Packet；在WR\_PKT狀態傳輸Packet到下一個Level Module。這裡要注意的是：Input\_Arbiter 用輪流查詢(RoundRobin)的方式來讀每個Queue，直到從一個Queue讀完一個完整的Packet才會繼續讀下一個Queue。

* **Output Port Lookup**
* **Packet processing State Machine**

Yes

發送到Destination Port

Packet從CPU Queue送出?

No

Yes

Packet Type是否為IP?

Destination MAC是否正確?

Yes

No

丟棄

IP CheckSum是否正確?

Yes

Yes

Destination IP Address是否正確?

Yes

No

IP Version!=4

No

No

TTL>1?

No

發送到相對應CPU Queue

Yes

Routing Table HIT?

TTL值-1 /更新IP CheckSum

/設置Destination MAC/設置Source MAC

No

Yes

Yes

ARP HIT?

說明:

由於身為Router Forwarding Engine 的Output\_port\_lookup，首先要做的工作就是Packets的IP Layer分析與處理，檢查其Destination MAC Address 、Version、Checksum和Time-to-live(TTL)等資訊。過濾掉各種類型的錯誤Packet。然後根據Packet的Destination IP Address查找Forwarding Table，確定Packet的Next-Hop 設備和Output port，故還需要查找ARP表來獲取Next-Hop設備的MAC Address；最後根據查表結果對Packet進行修改或Encapsulation(打包封裝)。上圖為Packet在Output port lookup的處理流程，

* **Module 內部構造圖**

**ip\_lpm**

**ip\_arp**

**dest\_ip\_filter**

**ip\_checksum\_ttl**

**op\_lut\_hdr\_parser**

**eth\_parser**

**pre\_process\_  
control**

**op\_lut\_process\_sm**

**router\_op\_lut\_regs**

**fallthrough\_small\_fifo**

**說明:**

**Packet 同時進入fallthrough\_small\_fifo和pre\_process\_control，前者負責保證Packet在Output port lookup中可以被緩衝處理，後者是整個Output port lookup的控制中樞，根據Packet進入的順序同時啟動eth\_parser、ip\_checksum\_ttl、ip\_lpm、op\_lut\_hdr\_parser和dest\_ip\_filter，並且等待所有module完成處理後，才會到op\_lut\_process\_sm取出之前預處理的訊息並決定Packet傳輸的目的地。**

* **Preprocess Control**

Preprocess control Module是一個簡潔的Finite State Machine，底下分析一下source code:

Case(state)

SKIP\_MODULE\_HDRS: Begin

Do something

End

WORD\_1: Begin

Do something

End

WORD\_2: Begin

Do something

End

WORD\_3: Begin

Do something

End

WORD\_4: Begin

Do something

End

WORD\_EOP: Begin

Do something

End

Endcase

**說明：**

在SKIP\_MODULE\_HDRS狀態跳過Packet Header Control Word，後面的狀態依次根據in\_data上Packet Word 的內容送出後續的啟動和控制信號，像是:word\_MAC\_DA\_HI、word\_MAC\_DASA及word\_MAC\_SA\_LO等等。

連接preprocess control的4個Module(eth\_parser、ip\_checksum\_ttl、ip\_lpm及dest\_ip\_filter)只有在檢測到啟動信號時才開始工作。

* **Eth parser**

**說明:**

此Module確定Frame類型(ARP或IP)，檢查輸入Packet的Destination MAC Address，輸出到Match的Port。

* **IP\_checksum\_ttl**

說明:

此Module檢查Packet的checksum和TTL值，輸出新的checksum和TTL值。

* **Ip\_lpm**

**說明:**

此Module實現Reference Router的LPM(Longest Prefix Matches)，輸出NextHop的MAC Address和輸出Port Number，啟動IP\_arp

* **Dest\_ip\_filter**

**說明:**

此Module實現Packet Destination IP Address的過濾，使用的是一個32x32的CAM Core。

連接在ip\_lpm後的是ip\_arp，完成Reference Router的ARP Lookup，輸出NextHop的MAC Address和輸出Port Number，這裡同樣也用了一個CAM Core。

Ip\_lpm、IP\_arp和dest\_ip\_filter這3個Module，他們都使用相同的結構，包含一個CAM Core和一個用來控制CAM的module。

* **Op lut hdr parser**

**說明：**

檢查Packet是否從CPU送來，此Module功能較簡單，故不多加贅述。

* **Op lut process sm**

**說明：**

**此Module丟掉出錯的Packet，修改Packet Control Word，有的被傳送給CPU，有的則轉發出去，同時將Packet統計結果送給Register Module。Module為一個複雜的FSM，故用　Transition Graph表達比較易懂：**

* **Op\_lut\_process\_smTransition Graph**
* in\_fifo\_ctrl==IOQ\_STAGE\_NUM
* to\_from\_cpu=1
* in\_fifo\_vld=1
* eop=1

(out\_dry&&in\_fifo\_vld)=1

(out\_dry&&in\_fifo\_vld)=1

in\_fifo\_ctrl=0

* is\_for\_us=0
* !(ip\_checksum\_is\_good)=1
* !(ip\_ttl\_is\_good)=1
* (out\_dry&&in\_fifo\_vld)=1
* eop=1
* is\_for\_us= 1
* is\_from\_cpu=1
* is\_ip\_pkt=1
* ip\_checksum\_is\_good=1
* (dest\_ip\_hit||(ip\_hdr\_has\_options|!ip\_ttl\_is\_good|!arp\_lookup\_hit|!lpm\_lookup\_hit))=1

(out\_dry&&

in\_fifo\_vld)=1

MOVE\_PKT

SEND\_IP\_CHECKSUM

SEND\_IP\_TTL

SEND\_SRC\_MAC\_LO

MOVE\_MODULE\_HDRS

DROP\_PKT

WAIT\_PREPROCESS\_RDY

**說明：**

WAIT\_PREPROCESS\_RDY收集相應Module的分析結果，根據這些結果跳轉到DROP\_PKT或MOVE\_MODULE\_HDRS；如果是後者則代表完成Router規定的操作和修改，並傳輸修改後的Packet。

* **user\_data\_path**
  + **input\_arbiter**

決定要從哪一個queue拿到資料, 並且送出去(也送給in\_arb\_regs), 決定的方式是上數0 ~ 7 (round-robin), 但是目前的queue必須要清空了才會往上加，會有不公平的現象

* + - **in\_arb\_regs**

讓pci可以讀取register

* + **output\_port\_lookup**
    - **fallthrough\_small\_fifo**

在output port這個module裡面暫時擺放封包的fifo

* + - **preprocess\_control**

6個state做循環，用9條信號線告訴其他模組目前的64bit資料是什麼，像是MAC或IP

* + - **eth\_parser**

看ethertype來判斷是arp(0x0806)封包還是ip封包(0x0800)，並且用兩條信號線(is\_arp\_pkt, is\_ip\_pkt)告訴其他模組結果。因為吃完ethertype的資料，接下來search\_req為1，狀態從IDLE轉換到DO\_SEARCH，從4數到0時dst MAC分別搜尋FFF\_FFF, MAC3, MAC2, MAC1, MAC0，用一條信號線(is\_for\_us)告訴其他模組是否為傳送至reference router的封包。

* + - **ip\_lpm**

具有read write interface的模組，其子模組如下

* + - * srl\_cam\_unencoded\_32x32

dst ip搜尋TCAM找到match address (many hot)

* + - * unencoded\_cam\_lut\_sm

將match address解成lowest priority順序的address，在routing table找rule得到port, nexthop ip

* + - **ip\_arp** 
      * 具有read write interface的模組，用nexthop ip在arp table(CAM)裡面搜尋，得到nexthop mac。
    - **dest\_ip\_filter**

如果dst ip有在filter裡面，將其送到對應的CPU output port

* + - **ip\_checksum\_ttl**檢查目前的checksum有沒有問題,
    - **op\_lut\_hdr\_parser**

檢查封包是不是從cpu port來, 是的話從哪個port來都告訴別人

* + - **oq\_lut\_process\_sm**Take the information from the preprocess blocks, write a new module header for the output port, write the packet with the information from the preprocess. 如果is\_for\_us為0的話，封包一定會被丟掉。
    - **router\_op\_lut\_regs**這個模組負責register解析，根據輸入的address信號map到對應的模組裡面(像是lpm, arp)
      * **router\_op\_lut\_regs\_non\_cntr**

此模組告訴模組eth\_parser所定義的mac0, mac1, mac2, mac3是什麼，除此之外從電腦透過PCI要寫入的值 (*reg\_data\_in*) 也由此模組負責轉換，再告訴其他人 (*arp\_wr\_addr, lpm\_wr\_ip, lpm\_wr\_mask*)。

* + **output\_queues**
  + **udp\_reg\_master**

處理pci要讀寫的資料線(*core\_reg\_wr\_data, core\_reg\_rd\_data*)

**output\_port\_lookup physical wiring**

****

* **Reference NIC**

**Receive path**

NetFPGA

Application

**Send Path**

NetFPGA

Application

Linux kernel

Libnet

**Applications**

**Kernel**

**Hardware**

NetFPGA Driver

NetFPGA Driver

Libpcap

Linux kernel

NetFPGA 1G to SUME Porting Document

* **目的：**

針對目前專案在NetFPGA 1G上的設計，整理對應的輸入輸出與內部邏輯，規劃並設計應用於NetFPGA SUME上的專案。

* **1G V.S. SUME Data Flow比較**

1G內部封包資料傳輸為64BITS DATA + 8BITS CTRL

SUME則為256BITS DATA + 32BITS T\_KEEP(CTRL)

* **目前評估開發流程**
  1. 熟悉合成tool流程，與新增模組進專案
  2. 研究SUME Register Interface如何操作(包含軟硬體)
  3. 先針對IP\_Feed\_Filter的部分，做簡易的行情格式六Parser，並驗證
  4. 加入策略表與產生模擬的訂單封包，並驗證
  5. 建立與Server之連線
  6. 整合連線與訂單封包之fix\_formatter，並驗證
* **內部模組 – 分三部分討論**
  + **IP\_Feed**
    - Preprocess\_control.v

輸入: 64bits packet data, 8bits ctrl, 1bit in\_wr

輸出:皆為1bit訊號，包含word\_MAC\_DA\_HI, word\_MAC\_DASA, word\_MAC\_SA\_LO, word\_ETH\_IP\_VER, word\_IP\_LEN\_ID, word\_IP\_FRAG\_TTL\_PROTO, word\_IP\_CHECKSUM\_SRC\_HI, word\_IP\_SRC\_DST, word\_IP\_DST\_LO, word\_ECSCODE\_FORMAT, word\_IP\_SEQ\_ACK, word\_CHECKSUM\_OPT\_1, word\_OPT\_2, word\_OPT\_3

邏輯:根據輸入的64bits packet data與8bits ctrl判斷目前輸入的data包含哪些內容

Porting修改：

輸入: 256bits packet data, 1bit t\_last與1bit t\_valid

輸出:皆為1bit訊號，包含word\_IP\_DST\_HI, word\_IP\_DST\_LO, word\_OPT即包含所需的部分

邏輯: 由於輸入packet data為每cycle 256bits，故減少輸出的訊號線

* Eth\_ip\_parser.v

輸入: 64bits packet data, 1bit word\_MAC\_DA\_HI, 1bit word\_MAC\_DASA, 1bit word\_MAC\_SA\_LO, 1bit word\_ETH\_IP\_VER, 1bit eth\_parser\_rd\_info, 48bits mac\_0, 48bits mac\_1, 48bits mac\_2, 48bits mac\_3(from register interface)

輸出:1bit is\_ip\_pkt, 1bit is\_udp\_pkt, 1bit is\_for\_us, 1bit is\_multicast, 1bit eth\_parser\_info\_vld, 8bits mac\_dst\_port\_num

邏輯:透過輸入的封包資料與對應的訊號線，擷取出source, destination mac與ethertype。透過擷取的資料比對是否為送給我們的封包，並計算出對應的destination port number。另外也判斷是否為ip或udp封包。

Porting修改：

輸入: 256bits packet data, 1bit word\_IP\_DST\_HI與1bit eth\_parser\_rd\_info

輸出:同1G，後面模組還是需要這些訊號判斷

邏輯:只需1個cycle即可擷取出所有需要的資料，比對的部分不太需要修改。

* Ip\_checksum.v

輸入: 64bits packet data, 1bit in\_wr, 1bit word\_ETH\_IP\_VER, 1bit word\_IP\_LEN\_ID, 1bit word\_IP\_FRAG\_TTL\_PROTO, 1bit word\_IP\_CHECKSUM\_SRC\_HI, 1bit word\_IP\_SRC\_DST, 1bit word\_IP\_DST\_LO, 1bit rd\_checksum

輸出:1bit ip\_checksum\_vld, 1bit ip\_checksum\_is\_good, 1bit ip\_hdr\_has\_options

邏輯:驗證封包的ip\_checksum是否正確。擷取出計算ip checksum所需的欄位後，還需額外兩個cycle計算carry值

Porting修改：

輸入: 256bits packet data, 1bit valid, 1bit word\_IP\_DST\_HI, 1bit word\_IP\_DST\_LO, 1bit rd\_checksum

輸出:1bit ip\_checksum\_vld, 1bit ip\_checksum\_is\_good, 1bit ip\_hdr\_has\_options

邏輯: 僅需兩個cycle擷取所需資料：讀到word\_IP\_DST\_HI==1後IPver+Headlen+Tos+TotalLength+ID+Flags+Fragment offset +TTL +Protocol+HeaderChecksum +Src IP ADDR+Src IP ADDR + Dst IP ADDR；讀到word\_IP\_DST\_LO後All temp data+Dst IP ADDR

後續也需要兩cycle計算carry：第一個cycle addcarry==1將final checksum拆成[19:16]和[15:0]相加；第二個cycle addcarry\_d1==1，相加後的值是否等於0xffff，是則完成計算。否則drop packet。

udp\_checksum.v

輸入: 64bits packet data, 1bit in\_wr, 8bits ctrl, 1bit word\_IP\_FRAG\_TTL\_PROTO, 1bit word\_IP\_CHECKSUM\_SRC\_HI, 1bit word\_IP\_DST\_LO, 1bit rd\_checksum

輸出:1bit udp\_checksum\_vld, 1bit udp\_checksum\_is\_good

邏輯:從protocol開始計算至end of packet，一樣到end of packet需要額外兩個cycle計算carry

Porting修改：

輸入: 256bits packet data, 1bit valid, 1bit word\_IP\_DST\_HI, 1bit word\_IP\_DST\_LO, 1bit rd\_checksum, 1bit last

輸出:1bit udp\_checksum\_vld, 1bit udp\_checksum\_is\_good

邏輯: 相對於1G由於data bus較寬，擷取所需資料須較少cycle，cycle長度視Payload長度而定。前兩個Cycle：讀到word\_IP\_DST\_HI==1後，由於我們只需要Protocol +Src IP ADDR+Src IP ADDR + Dst IP，故我們把Header checksum這段的data用Protocol段的取代，這樣可以在一個cycle完成Protocol +Src IP ADDR+Src IP ADDR + Dst IP的計算。後續Payload的部分，依據刻好的16個26bits Register，每個cycle 256bits Data進入後，存到相對應位置的Register，每個Cycle將兩兩register相加後加上前一個Cycle暫存的checksum值。持續相加到end of packet，這裡不直接判斷tlast==1是因為會有delay導致checksum無法在正確的cycle算出，於是需要另外用一條訊號線tkeep\_prev\_is\_F控制。

後續需要3 cycle計算carry：add\_carry\_1==1為第一個cycle，將所有checksum\_word register相加，接著add\_carry2和add\_carry3將final checksum拆成[19:16]和[15:0]相加，相加後的值是否等於0xffff，是則完成計算。否則drop packet。

* op\_lut\_hdr\_parser.v

輸入: 4bits packet data, 1bit in\_wr, 8bits ctrl, 1bit rd\_hdr\_parser

輸出:1bit is\_from\_cpu, 8bits to\_cpu\_output\_port(目前沒用到), 8bits from\_cpu\_output\_port, 3bits input\_port\_num, 1bit is\_from\_cpu\_vld

邏輯:擷取module header(ctrl = 0xff)中的src\_port資料，判斷封包是否來自cpu，判斷方法為SRC\_Port為0~7的value，先轉為8bit one-hot表示法在與8’b10101010做and，若大於0則為from cpu(也就是0~7中偶數(0, 2, 4, 6)為mac queue 奇數(1, 3, 5, 7)為cpu queue)

Porting修改：

輸入: 256bits packet data, 128bits tuser, 1bit valid, 1bit tlast, 1bit rd\_hdr\_parser

輸出: 1bit is\_from\_cpu, 8bits to\_cpu\_output\_port(目前沒用到可以拿掉), 8bits from\_cpu\_output\_port, 3bits input\_port\_num, 1bit is\_from\_cpu\_vld

邏輯:SUME沒有Module header，src port的資訊儲存在tuser中，擷取出src\_port value後一樣轉成8bits one-hot表示做比較，另外因為to\_cpu\_output\_port後面op\_lut\_process\_sm沒用到所以可以拿掉

* ip\_feed\_filter.v

輸入: 64bits packet data, 1bit in\_wr, 8bits ctrl, 1bit word\_IP\_FRAG\_TTL\_PROTO, 1bit word\_IP\_DST\_LO, 1bit word\_ECSCODE\_FORMAT, 1bit rd\_ip\_feed\_filter\_result

Register interface:5bits listed\_stock\_format\_type\_stats\_rd\_addr,

1bit listed\_stock\_format\_type\_stats\_rd\_req,

5bit OTC\_stock\_format\_type\_stats\_rd\_addr,

1bit OTC\_stock\_format\_type\_stats\_rd\_req,

5bit listed\_warrant\_format\_type\_stats\_rd\_addr,

1bit listed\_warrant\_format\_type\_stats\_rd\_req,

5bit OTC\_warrant\_format\_type\_stats\_rd\_addr,

1bit OTC\_warrant\_format\_type\_stats\_rd\_req

輸出:1bit ip\_feed\_vilter\_vld, 1bit is\_ip\_feed

Register interface:16bits listed\_stock\_format\_type\_stats\_rd\_value,

1bit listed\_stock\_format\_type\_stats\_rd\_ack,

16bit OTC\_stock\_format\_type\_stats\_rd\_value,

1bit OTC\_stock\_format\_type\_stats\_rd\_ack,

16bit listed\_warrant\_format\_type\_stats\_rd\_value,

1bit listed\_warrant\_format\_type\_stats\_rd\_ack,

16bit OTC\_warrant\_format\_type\_stats\_rd\_value,

1bit OTC\_warrant\_format\_type\_stats\_rd\_ack

邏輯:透過訊號線判斷目前資料欄位，擷取所需部分(protocol, ecs\_code, terminal code, format filed等等)，判斷幾個部分，”是否為udp”，”ECSCODE是否為8’h1b(固定)”，”判斷terminal code是否為0d0a(固定)”，”格式的欄位是否為8’h06”。另外與Register interface溝通的部分主要是做一些行情封包的分析數據統計，

Porting修改：

輸入: 256bits packet data, 1bit valid, 1bit tlast, 1bit word\_IP\_DST\_HI, 1bit word\_IP\_DST\_LO

輸出:1bit ip\_feed\_filter\_vld, 1bit is\_ip\_feed

邏輯: 同1G，只是擷取資料的部分可以在兩個cycle處理完

* check\_tcp\_flag.v

輸入: 64bits packet data, 1bit in\_wr, 8bits ctrl,1bit word\_IP\_FRAG\_TTL\_PROTO, 1bit

word\_IP\_CHECKSUM\_SRC\_HI, 1bit

word\_IP\_SRC\_DST, 1bit word\_IP\_DST\_LO, 1bit word\_IP\_SEQ\_ACK, 1bit word\_CHECKSUM\_OPT\_1, 1bit word\_OPT\_2, 1bit word\_OPT\_3, 1bit rd\_check

輸出:1bit hand\_shake\_vld, 1bit is\_tcp\_handshake, 1bit is\_tcp\_ack, 1bit is\_tcp\_fin, 32bits seq\_val, 32bits ack\_val, 32bits ts\_val, 32bits ecr\_val

邏輯: 通過訊號線判斷目前資料欄位，判斷所需部分(pkt\_length,cp\_flag, sequence number, acknowledge number, timestamp, echo value)。判斷”protocol是否為tcp”,”tcp\_flag為syn,ack or ack or fin or psh”, “acknowledge number是否正確”。比較麻煩的是packet header長度不一，syn,ack為74，純ack與其他為66。另外ack num驗證的方式是以我們送出的syn封包當作初始做判斷。

Porting修改：

輸入: 256bits packet data, 1bit valid, 1bit tlast, 1bit word\_IP\_DST\_HI, 1bit word\_IP\_DST\_LO, 1bit word\_OPT

輸出: 1bit hand\_shake\_vld, 1bit is\_tcp\_handshake, 1bit is\_tcp\_ack, 1bit is\_tcp\_fin, 32bits seq\_val, 32bits ack\_val, 32bits ts\_val, 32bits ecr\_val

邏輯:三個cycle即可取得所有資料，在1G我是一邊讀封包一邊判斷，可以考慮改成將要的資料暫存起來，判斷的部分用組合電路做，一樣要考慮HEADER長度不同的情況。

* fix\_fileter.v

輸入: 64bits packet data, 1bit in\_wr, 8bits ctrl, 1bit rd\_check, 1bit resend\_ack(與fix\_formatter溝通，跳號重送處理)

輸出:1bit is\_fix, 1bit is\_logon, 1bit is\_report, 32bits fix\_server\_seq, 32bits resend\_begin, 32bits resend\_num, 1bit resend\_req(與fix\_formatter溝通，跳號重送處理)

邏輯:通過counter計算目前得到的packet data包含哪些欄位，一樣是一邊讀一邊在對應的位置做判斷。判斷包含” protocol是否為tcp”, “判斷tcp flag是否為psh,ack”, “判斷FIX Message為logon, report or resend”, “儲存FIX Server端的Sequence number”, “儲存resend begin sequence number”。一樣要針對不同長度的Payload去做對應的parse。可能需19個Cycle(若為resend packet)才能讀完所需的資料。

Porting修改：

輸入: 256bits packet data, 1bit valid, 1bit tlast

輸出: 1bit is\_fix, 1bit is\_logon, 1bit is\_report, 32bits fix\_server\_seq, 32bits resend\_begin, 32bits resend\_num, 1bit resend\_req(與fix\_formatter溝通，跳號重送處理)

邏輯:大致修改重點最長應該5個cycle可讀取完所需資料，一樣要注意不同長度的資料擷取與End of packet處理。

* connect\_check.v

輸入:6bits connect\_flag\_rd\_addr, 1bit connect\_flag\_rd\_req, 6bit connect\_flag\_wr\_addr, 1bit connect\_flag\_wr\_value, 1bit connect\_flag\_wr\_req

輸出: 1bit connect\_value(與op\_lut\_process\_sm連接作為啟動連線的訊號), 6bits connect\_flag\_rd\_value, 1bit connect\_flag\_rd\_ack, 1bit connect\_flag\_wr\_ack

邏輯:除了connect\_value外都是與register interface溝通的部分，主要概念就是透過register interface修改特定register的value，並在op\_process\_sm中判斷若為1則送出連線封包。

Porting修改：

* kernel\_time\_rw.v

輸入:1bit kernel\_time\_rd\_req, 1bit kernel\_time\_wr\_req, 16bits kernel\_time\_wr\_ms, 16bits kernel\_time\_wr\_s, 16bits kernel\_time\_wr\_min, 16bits kernel\_time\_wr\_hour, 16bits kernel\_time\_wr\_day, 16bits kernel\_time\_wr\_mon, 16bits kernel\_time\_wr\_year, 1bit pkt\_gen\_req(目前沒用到可以刪除)

輸出: 1bit kernel\_time\_rd\_ack, 1bit kernel\_time\_wr\_ack, 16bits kernel\_time\_rd\_ms, 16bits kernel\_time\_rd\_s, 16bits kernel\_time\_rd\_min, 16bits kernel\_time\_rd\_hour, 16bits kernel\_time\_rd\_day, 16bits kernel\_time\_rd\_mon, 16bits kernel\_time\_rd\_year, 16bits pkt\_ms, 16bits pkt\_s, 16bits pkt\_min, 16bits pkt\_hour, 16bits pkt\_day, 16bits pkt\_mon, 16bits pkt\_year(與op\_lut\_process\_sm, fix\_formatter連接，將clock time傳到要送出的封包)

邏輯:與register interface的部分讀取一次kernel time後，透過counter機制每125000個counter即增加1ms，維持硬體clock。並透過binary\_to\_bcd模組將value轉換成bcd以符合封包格式。另外時間也與op\_lut\_process\_sm, fix\_formatter連接，隨時將時間資料傳給新的封包。

Porting修改：

* binary\_to\_bcd.v

輸入:12bits binary(9999以下)

輸出:4bits thousand, 4bits hundred, 4bits ten, 4bits one(個位十位百位千位)

邏輯:將9999以下的binary value轉為bcd表示

Porting修改：

輸入:12bits binary(9999以下)

輸出:4bits thousand, 4bits hundred, 4bits ten, 4bits one(個位十位百位千位)

邏輯:應該不需要修改

* + - **op\_lut\_process\_sm.v**

輸入:

1bit in\_fifo\_vld,

64bits in\_fifo\_data,

8bits in\_fifo\_ctrl,

1bit is\_ip\_pkt,

1bit is\_udp\_pkt,

1 bit is\_for\_us,

1bit is\_multi\_cast,

1bit eth\_parser\_info\_vld,

3bits mac\_dst\_port\_num,

1bit is\_from\_cpu,

8bits to\_cpu\_output\_port,

8bit from\_cpu\_output\_port,

1bit is\_from\_cpu\_vld,

3bits input\_port\_num,

1bit ip\_checksum\_vld,

1bit ip\_checksum\_is\_good,

1bit ip\_hdr\_has\_options,

1bit udp\_checksum\_vld,

1bit udp\_checksum\_is\_good,

1bit ip\_feed\_filter\_vld,

1bit is\_ip\_feed,

1bit out\_rdy,

48bits mac\_0,

48bits mac\_1,

48bits mac\_2,

48bits mac\_3,

1bit connect\_signal,

1bit hand\_shake\_vld,

1bit is\_tcp\_hand\_shake,

1bit is\_tcp\_ack,

1bit is\_tcp\_fin,

1bit is\_fix,

1bit is\_logon,

1bit is\_report,

32bits seq\_value,

32bits ack\_value,

32bits ts\_value,

32bits ecr\_value,

16bits pkt\_year,

16bits pkt\_mon,

16bits pkt\_day,

16bits pkt\_hour,

16bits pkt\_min,

16bits pkt\_sec,

16bits pkt\_ms,

輸出:

1bit in\_fifo\_rd\_en,

1bit rd\_preprocess\_info,

1bit out\_wr,

64bits out\_data,

8bits out\_ctrl,

1bit pkt\_forwarded,

1bit pkt\_dropped\_checksum,

1bit pkt\_sent\_to\_cpu\_non\_ip,

1bit pkt\_dropped\_wrong\_dst\_mac,

1bit pkt\_gen\_req,

1bit syn\_sended,

1bit is\_syn\_ack,

1bit logon\_sended,

1bit is\_ack,

1bit ack\_sended,

1bit is\_fix\_logon,

1bit is\_fix\_report

邏輯:

**From CPU?**

**Forward pkt\_data**

**Udp & format 6?**

**Tcp or fix logon**

**fix report**

**Forward pkt\_data**

**Forward ack & drop pkt\_data**

**Is hand\_shake**

**Forward FIX logon**

**Forward ack**

**Forward pkt\_data(report) to another port**

**yes**

**yes**

**yes**

**yes**

**yes**

**no**

**no**

**no**

**no**

**drop**

* + **Order\_Generator**
  + **Register\_Interface**
    - router\_op\_lut\_regs.v

輸入:

輸出:

邏輯:包含兩個模組，router\_op\_lut\_regs\_cntr與router\_op\_lut\_regs\_non\_cntr，主要差異在於router\_op\_lut\_regs\_cntr是透過一些counter記錄電路內部數據(Ex. pkt\_forward數量) ，並且是遞增形式。user透過cli只能讀取相關數據無法修改。而router\_op\_lut\_regs\_non\_cntr則讓user可以透過cli對register作read/write，並可透過RAM / Register的方式儲存資料。

NetFPGA 環境設置

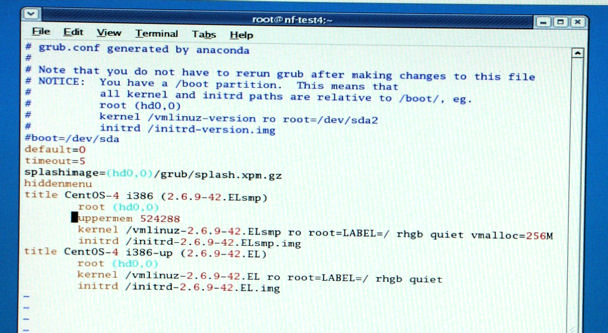
1. 用光碟安裝CentOS (裝6.5)

建議用英文體系 比較不會遇到問題

1. $ vim /boot/grub/grub.conf

在kernel行前面加上 uppermem 524288

在kernel行後面加上 vmalloc=256M



1. $ lspci

查看是否出現##:##.# Ethernet controller: Unknown device feed:0001

1. 到<https://github.com/NetFPGA/netfpga/wiki/Releases>下載NetFPGA Full 3.0.1

$ tar –xvzf netfpga\_full\_3\_0\_1.tar.gz

1. $ cat ~/netfpga/bashrc\_addon >> ~/.bashrc

$ source ~/netfpga/bashrc\_addon

1. $ yum –y install kernel-devel

重新開機

1. $ yum update

$ yum -y install gcc -----到這

$ yum -y install gcc-c++

$ yum -y install ncurses-devel

$ yum -y install libnet libnet-devel

$ yum -y install compat-libstdc++-296

$ yum -y install libpcap libpcap-devel

1. $ rpm –Uvh <http://repository.it4i.cz/mirrors/repoforge/redhat/el6/en/x86_64/rpmforge/RPMS/rpmforge-release-0.5.3-1.el6.rf.x86_64.rpm>

$ yum -y install perl-Net-Pcap

$ yum -y install perl-Net-RawIP  
$ yum -y install perl-Error.noarch

$ yum -y install perl-XML-Simple

1. 用光碟安裝Xilinx ISE\_DS 10.1
2. $ vim ~/bash.rc

增加一行source /opt/Xilinx/10.1/ISE/settings64.sh

1. $ cd ~/netfpga

$ make

1. $ cd ~/netfpga/projects/reference\_router/synth

$ make

Insmod nf2.ko

Ifconfig nf2c0~3 up

./nf\_download \*.bit

可透過sw/cli對板子上的router操作

* 1G PC: wade@140.116.82.186 port 2266 ykchang66 登入在su ykchang66
* QuickFIX Server PC: Yifang@140.116.82.187 port 2266 ykchang66 登入在su ykchang66
* QuickFIX Server
  + 先將eth1連接

設定routing table & arp table



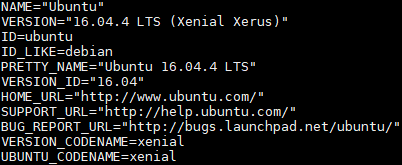
送封包的網卡設一下固定ip(隨便都可，因為如果broadcast會一直彈訊息讓netfpga運作受影響)

* + 將多餘的走eth1的rule刪掉(ex. 140.116.82.0)
  + 透過sudo 執行bin/run\_executor\_cpp.sh

NetFPGA SUME 環境配置

* OS版本 : Ubuntu 16.04.4 LTS

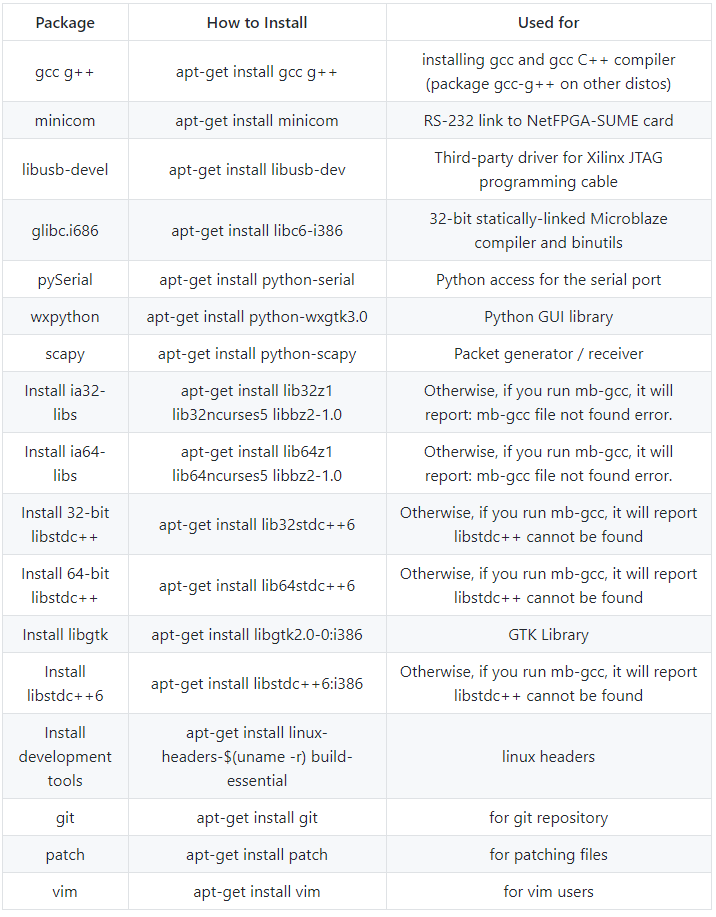
詳細資訊



* Kernel版本 : 4.15.0-29-generic



* Xilinx VIVADO 版本 : 2016.4 System Edition (過舊的版本會無法使用)
* Xilinx License :Vivado Design Suite – System edition(這裡面含有多個License，詳細的並沒有列出來 ，不過跟Xilinx買VIVADO Design Suite System Edition的License，應該就包含最完整的License 內容了) , EF-DI-25GEMAC-SITE(include LogiCORE, 10 Gigabit Ethernet MAC, Site License)，主要需要這兩個License即可合成NetFPGA SUME的電路
* 需要在OS上安裝的Packages



* 安裝篇
* 設定環境變數:

source ~/NetFPGA-SUME-live/tools/settings.sh

source ~/opt/Xilinx/Vivado/2016.1/settings64.sh

* 第一次安裝SUME:

cd $SUME\_FOLDER

make

* 如果之前已經有make過，則不需要每次都執行make，除非有改IP cores。

如果有改sume的 library cores：執行 make sume即可更新library

* 如果只更改單一project則到該project資料夾執行make
* 如果要更換vivado版本執行，記得環境變數要重新設定
* 如果要更換project執行，同上
* 執行篇

產生Bitfile後，接著要Load到NetFPGA SUME上，執行

$ cd bitfiles

$ xmd

進入Xilinx tool的介面後，執行

fpga –f reference\_router.bit

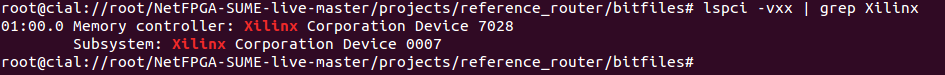
會看到下列畫面：



接著為了確定bitfie有Load到SUME上，要執行下列指令:

lspci –vxx | grep Xilinx

如果有看到Xilinx device代表成功，如下圖:



* 之後還要安裝一些SUME的Driver，執行:

$ cd $DRIVER\_FOLDER

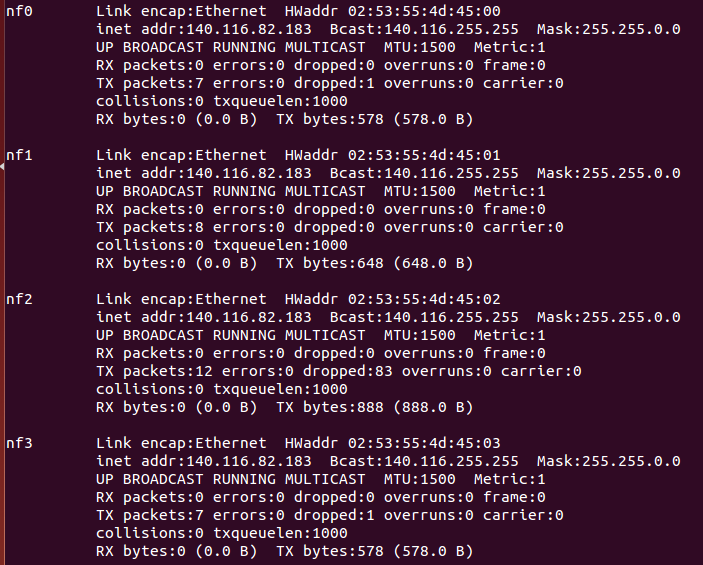
$ make all

$ make install

$ modprobe sume\_riffa

$ lsmod sume\_riffa

接著Run ifconfig -a 如果有看到nfX Device如下:



代表Driver有安裝好，可以讀到SUME在正常運作。

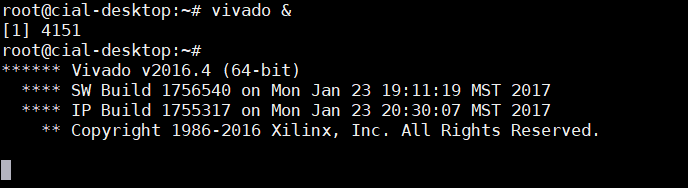
* 若沒有看到nfx的device:到 tools/scripts/reconfigure/執行pci\_rescan\_run.sh
* 再執行一次ifconfig –a 就可以了
* 測試篇
* 進project/test看測試的python檔名both\_前\_後
* ./nf\_test.py hw --major 前 –minor 後

NetFPGA Driver安裝與常見問題排除方法

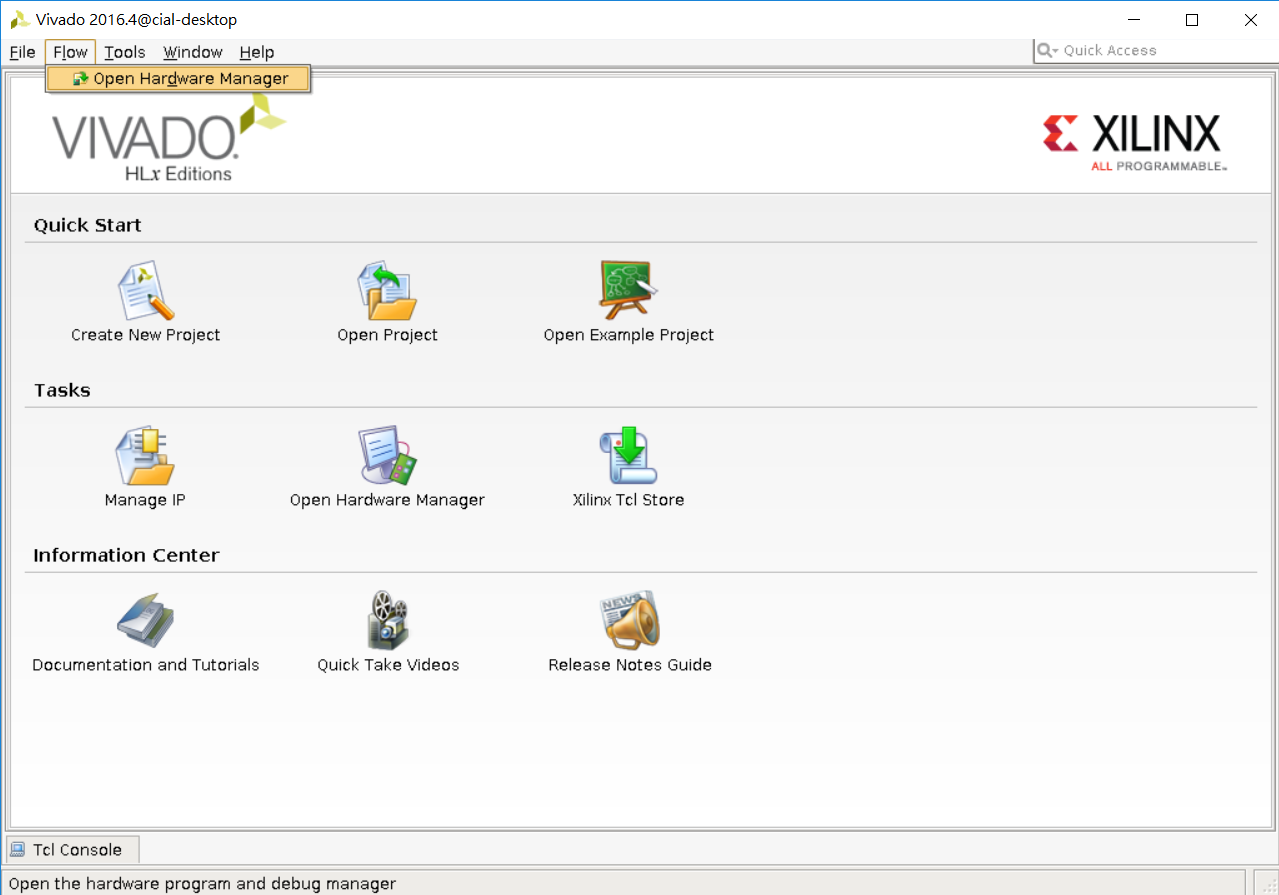
* 說明

NetFPGA SUME與Host PC之PCIe溝通管道使用了RIFFA[1][2] 這個framework，其中RIFFA的原始碼被包含在nf\_riffa\_dma這個IP中，Host PC端的driver程式同樣使用了RIFFA的原始碼，因此Host PC若想透過PCIe偵測到NetFPGA SUME的網路介面(nf0 ~ nf3)，NetFPGA SUME上必須要執行一個包含nf\_riffa\_dma這個IP的專案，兩邊的通訊才可以成功建立。因此以下教學步驟說明了如何在NetFPGA SUME端燒錄一個可行的專案(reference nic)，讓Host PC端的driver成功偵測到NetFPGA SUME的網路介面。

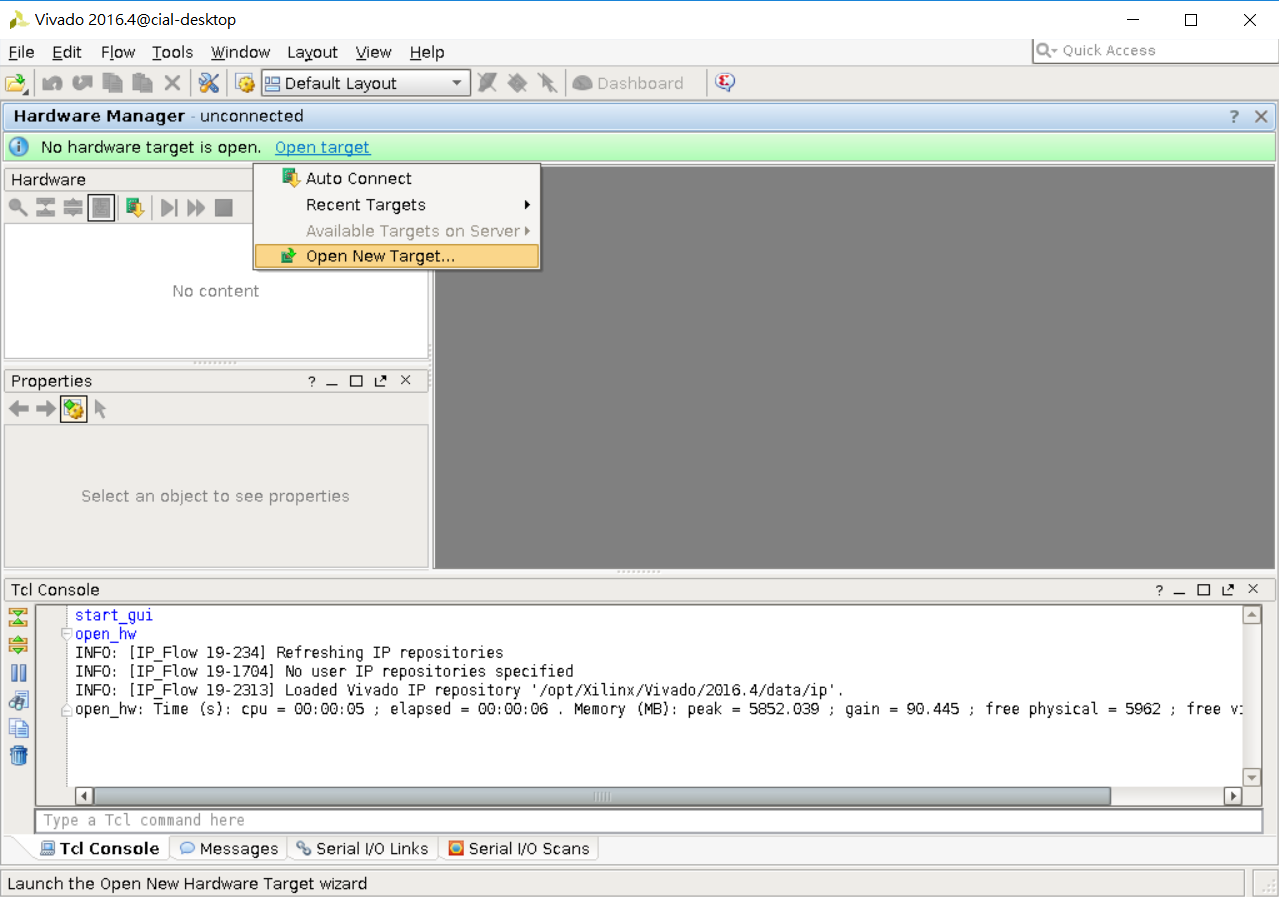
1. 開啟Vivado 2016.4



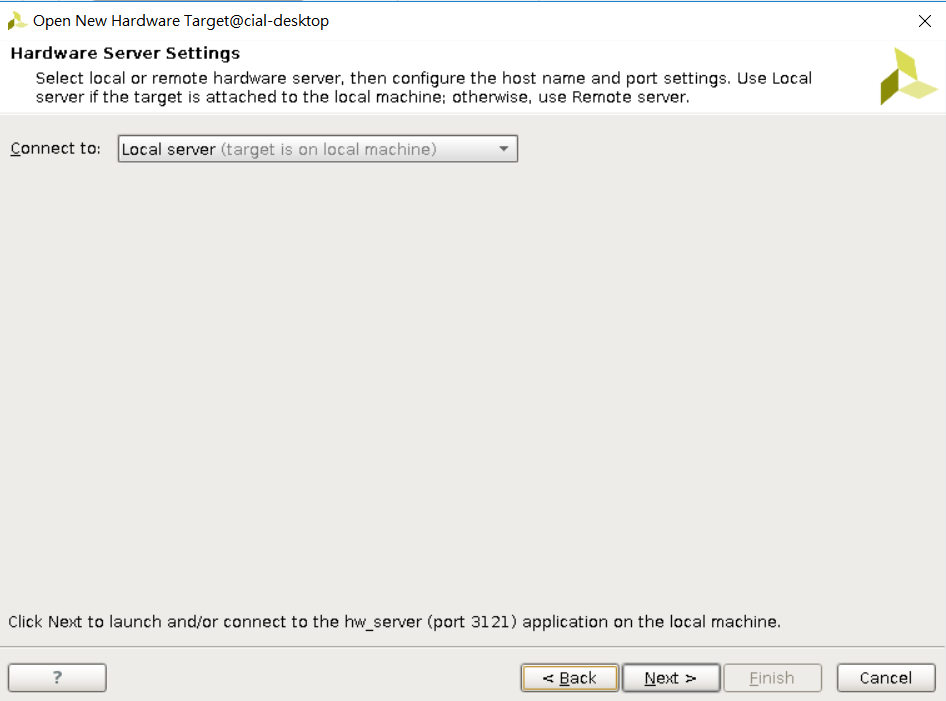
2. 點選Flow -> Open Hardware Manager



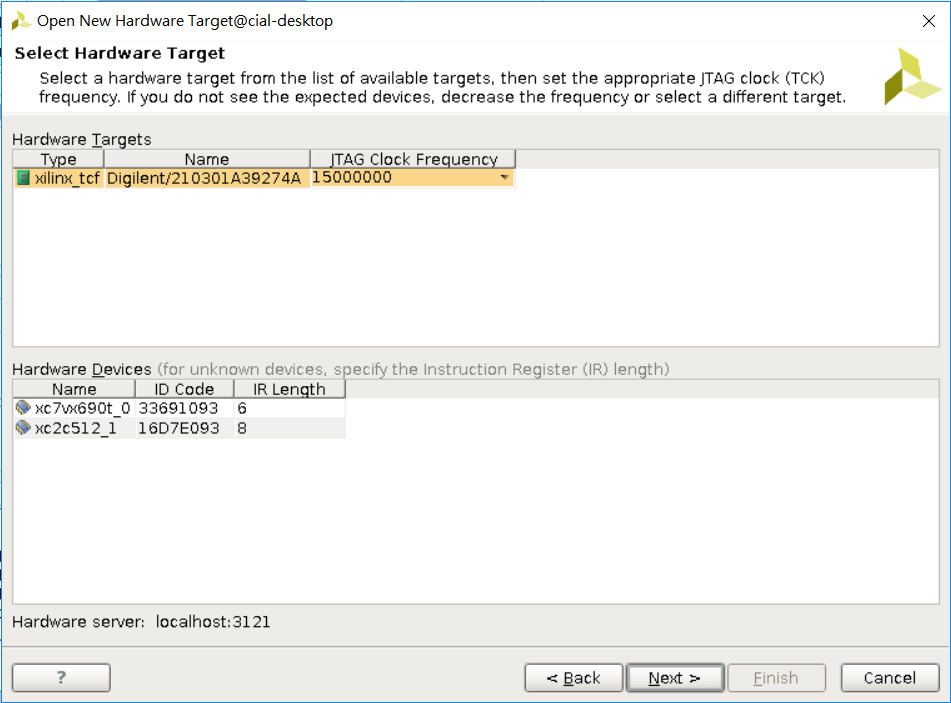
3. 點選Open target -> Open New Target …



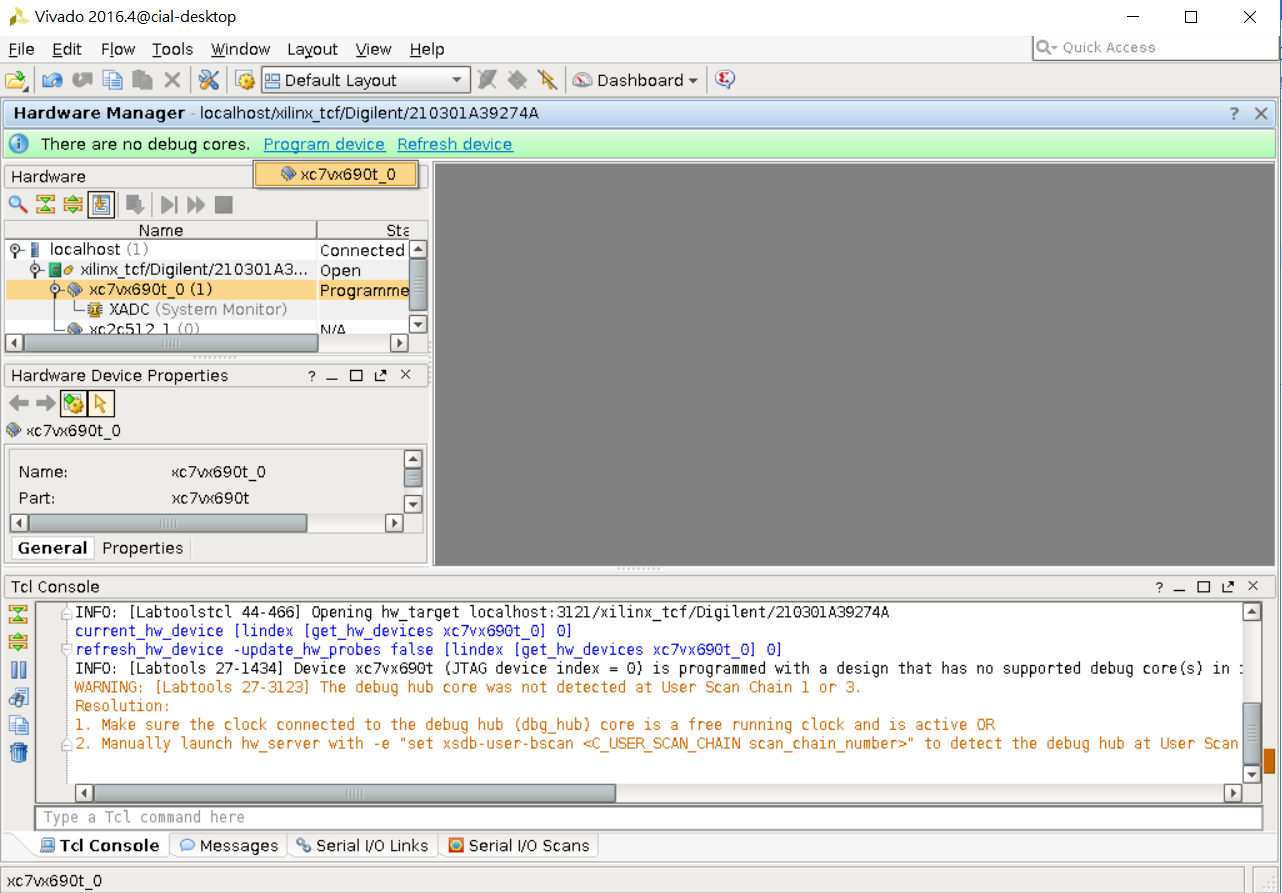
4. 確認Connect to 的選擇為Local server，點選Next



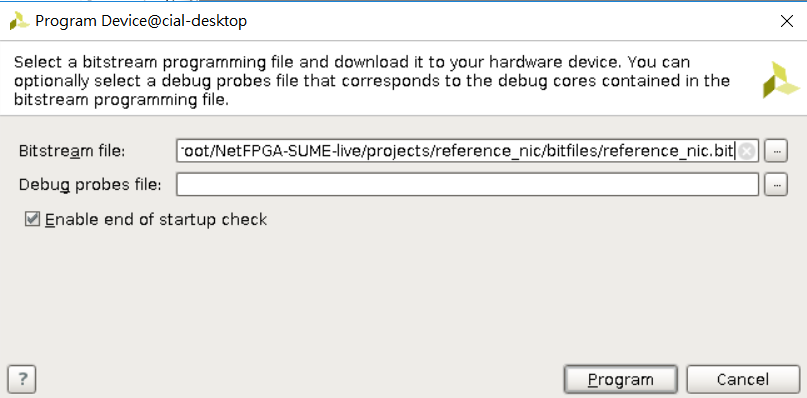
5. 經過數秒的等待後，會出現如下圖的頁面，確認硬體資訊是否與下圖相同



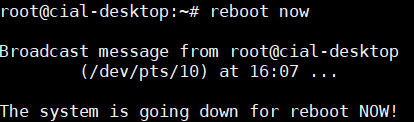
6. 完成Open New Hardware Target後點選Program device ->xc7vx690t\_0

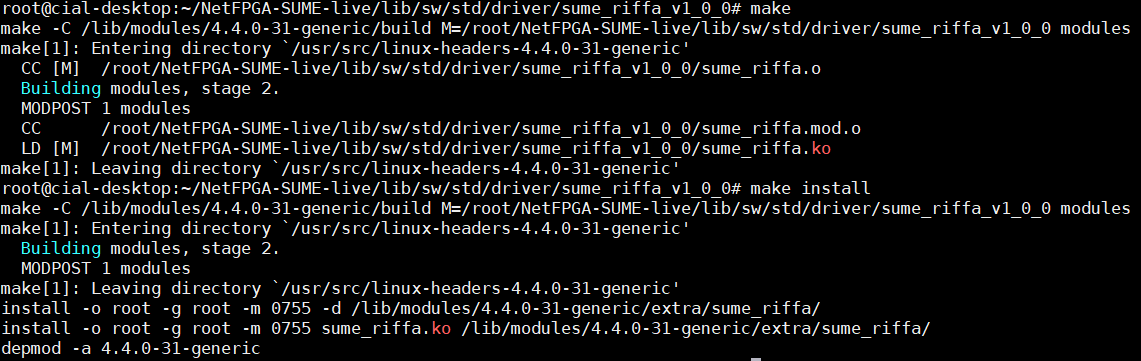


7. Bitstream file選擇已經合成好的reference\_nic.bit (或reference\_router等其他可運行專案)，點選Program

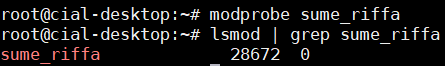


8. 將機器重新開機

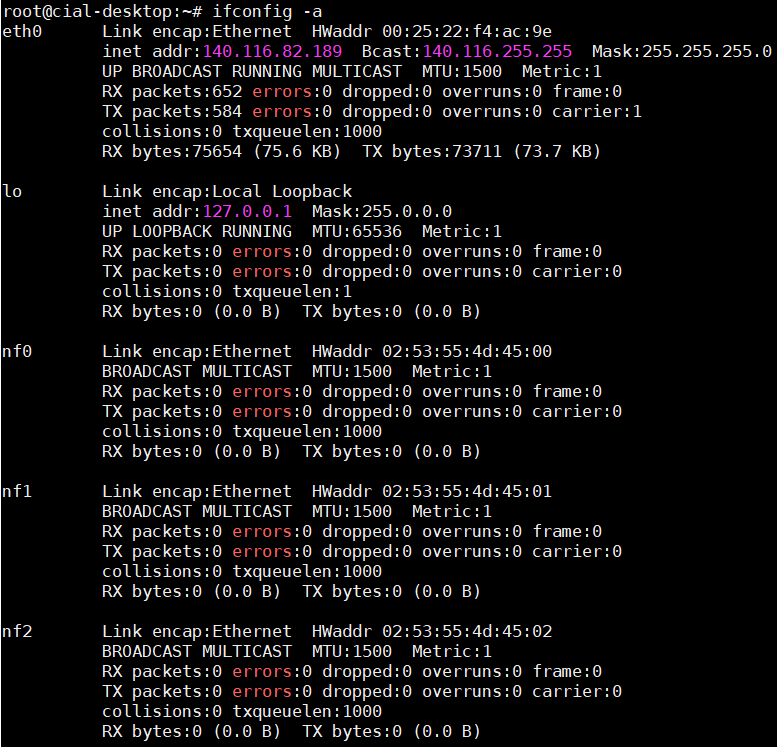


9. 編譯並安裝driver程式 (只需做一次，若機器有經過重灌變更後才需要重新編譯安裝，若無則跳至步驟10)

10. 使用modprobe指令載入driver的kernel module，並用lsmod確認是否正確載入。



11. 最後輸入ifconfig -a查看網路介面資訊，應該可以得到nf0 ~ nf3的資訊，代表host PC已經可以偵測到NetFPGA SUME。





[1] [RIFFA: A Reusable Integration Framework for FPGA Accelerators](https://ieeexplore.ieee.org/document/6239817)

[2] [RIFFA 2.0: A reusable integration framework for FPGA accelerators](https://ieeexplore.ieee.org/document/6645504)

NetFPGA SUME 刪單系統操作手冊

* Tcpdump是一個執行在命令列下的嗅探工具。它允許用戶攔截和顯示傳送或收到過網路連線到該電腦的TCP/IP和其他封包。tcpdump 是一個在BSD授權條款下釋出的自由軟體。

Server有幫你裝了可以直接用，tcpdump –h 可以看看

* Wireshark是一個免費開源的網路封包分析軟體。網路封包分析軟體的功能是截取網路封包，並盡可能顯示出最為詳細的網路封包資料。 在過去，網路封包分析軟體是非常昂貴，或是專門屬於營利用的軟體，Wireshark的出現改變了這一切。

[**下載連結**](https://www.wireshark.org/download.html)

Server有幫你裝了可以直接用，tshark –h可以看看

* 環境架設：目前透過DMA(nf3)送FIX Order，MAC\_0收行情，MAC\_3送刪單
  + SUME PC
    - [cial@140.116.82.183](mailto:cial@140.116.82.183) port 2266 cial66
    - 開發專案 reference\_router
    - 開機後
      1. cd $DRIVER\_FOLDER
      2. make all
      3. make install
      4. modprobe sume.riffa
      5. ifconfig nf0~3 up
      6. cd /root/NetFPGA-SUME-live-master/tools/scripts/
      7. sh run\_load\_image.sh ~/NetFPGA-SUME-live-master/projects/reference\_router/bitfiles/reference\_router.bit
    - 若需修改hdl source code，主要都是修改/root/NetFPGA-SUME-live-master/lib/hw/std/cores/router\_output\_port\_lookup\_v1\_0\_0/hdl中的verilog file
    - 權證資料儲存於 /root/NetFPGA-SUME-live-master/projects/reference\_router/sw/host/cli/stock\_data/kgi\_commodity\_fix.txt 中 格式為：0050(股票代號) 06205P(權證代號)
    - root/NetFPGA-SUME-live-master/projects/reference\_router/sw/host/cli/cli為Host PC主要與NetFPGA SUME溝通介面，執行cli後即呼叫三個函式**init\_str\_array();** - 初始化宣告的陣列(has\_table)

**stock\_id\_transfer();** - 讀stock\_data/kgi\_commodity\_fix.txt，並將股票代號轉換成MSB\_24BITS與LSB\_24BITS的形式儲存。另外也計算出每一檔股票代號對應的權證代號Index Range. Ex. 0050 0 6 代表股票代號0050對應的權證為權證table中的index 0~6這7檔

**commodity\_id\_transfer();** 轉換權證代號成MSB\_24BITS與LSB\_24BITS的形式儲存。

Cli command:

* + - 1. **loadstocktable –** 透過/stock\_data/Stock\_ID.txt這個文件去計算出289個Stock\_ID對應的Hash index，並透過將Stock\_ID與對應的warrants index range傳入BRAM(stocktable)中並寫入對應的位置
      2. **loadwarrants –** 透過/stock\_data/Commodity.txt這個文件去計算出1550個Warrants\_ID對應的Hash index，並寫入BRAM中(warrants\_table & order\_table)。
      3. **liststocktable –** 列出stocktable中股票的前一次儲存的最佳一檔買價與賣價
      4. **listwarrantstable –** 列出warrants table中1550之權證對應到order table(hash table)的index為何
      5. **listordertable –** 列出目前ordertable(目前只讀取buy table，之後會加上sell table的部分)儲存情況與儲存總單數。
    - /root/NetFPGA-SUME-live-master/projects/reference\_router/sw/host/apps/register\_read.sh會秀出NetFPGA內Counter的資訊
    - tcpreplay -i nf3 –p 50000 order\_16447.pcap 傳FIX Order給SUME NF3(DMA)
    - #listordertable : buy 1430
  + NIC PC
    - [root@140.116.82.179](mailto:root@140.116.82.179) port 22 cial66
    - tcpreplay -i eth7 –p 50000 stock\_quote\_0712.cap. 傳所有行情給SUME MAC\_0
    - tcpreplay -i –p 50000 eth7 decoder\_0821.cap. 傳6298格式六行情給SUME MAC\_0
    - 送行情 => #listordertable : buy 1236, 送出194個封包
* HDL設計架構 – 主要分五個部分(可以透過/root/NetFPGA-SUME-live-master/projects/reference\_router/hw/tcl/reference\_router\_bd\_connection.tcl看到module間connection的情況)
  + TX / RX Queues – RX為收到的封包過PHY層後最先進入的Module，TX則為要送出的封包到PHY層前最後經過的模組，這邊我們沒修改。
  + Input Arbiter – Round Robin選取RX Queue中CPU0~3 & MAC0~3，這個模組我們沒修改
  + IP\_Feed / FIX Parser
    - 透過preprocess\_controv告訴其他submodules目前進來的封包資料包是整個封包的第幾個word。(一個word有256bits，從封包的頭開始計算)
    - 做一些checksum的檢查，也判斷是否為行情，是否為FIX封包
    - 我目前額外判斷的東西是格式六行情、FIX order才能通過此stage，整合的時候要修改
  + FIX\_Generator
    - 若為行情則先cuckoo hash比對我們的stock table，有中則比較前一次的買賣價，若符合條件(新買價<舊買價 => 刪買單；新賣價>舊賣價 => 刪賣單)則將後面的order content index傳出。
    - 若為FIX Order則做cuckoo hash比對order table，找到對應的位置後將order內容存入order content table
    - Stock\_id\_mapping
      1. parse出股票代號(固定位置)
      2. parse最佳一檔買價賣價(共10種可能，有無即時成交價量(2種) \* 最佳一至五檔(5種))
      3. parse出order content，儲存的型式如下
      4. Parse出來的股票代號做cuckoo hash，取得maapping結果後比對買賣價，若new buy price < old but price => 刪買單(傳stock\_code的11bits start + 11bits end + 2’b01)，new sell price > old sell price =>刪賣單(傳stock\_code的11bits start + 11bits end + 2’b10)。
      5. 寫入stock\_code的資料 => 48bits股票代號 + 11bits對應權證的start index + 11bits 對應權證的end index
* Warrants\_id\_mapping
  1. 寫入warrants\_index的資料 => 12bits的對應order content的index
  2. 此module主要就是讀出此order\_contene的index並取order\_id\_mapping這個module中的order\_content BRAM access出對應的資料送至fix\_formatter
* Order\_id\_mapping
  1. 寫入order\_table的資料 => 48bits的股票代號
  2. 將收到order之symbol在此做cuckoo hash，並透過mapping得到的結果將order content存入order\_content BRAM
  3. 透過warrants\_id\_mapping傳入的order\_content BRAM的index access order\_content BRAM，並將內容回傳至warrants\_id\_mapping module
     + Fix\_formatter–將讀出來的order\_content在打包成FIX Cancel Request的形式送出
  + Output Queues
    - 將封包暫時buffer住，並送至對應的TX Queue
* 架構圖

｜router\_output\_port\_lookup\_

｜—ip\_feed\_fix\_parser

｜｜—preprocess\_control

｜｜—eth\_parser

｜｜—ip\_checksum

｜｜—op\_lut\_hdr\_parser

｜｜—udp\_checksum

｜｜—ip\_feed\_filter

｜｜—fix\_filter

｜｜—op\_lut\_process\_sm

｜—fix\_generator

｜｜—preprocess\_control

｜｜—stock\_id\_mapping

｜｜｜—one\_at\_a\_time\_0

｜｜｜—one\_at\_a\_time\_1

｜｜｜—(stock\_code\_512\*70) \*2

｜｜｜—(stock\_price 512\*49) \*2

｜｜—warrants\_id\_mapping

｜｜｜—order\_id\_mapping

｜｜｜｜—one\_at\_a\_time\_0

｜｜｜｜—one\_at\_a\_time\_1

｜｜｜｜—one\_at\_a\_time\_2

｜｜｜｜—one\_at\_a\_time\_3

｜｜｜｜—(warrants\_code 1024\*48) \* 4

｜｜｜｜—(order\_content 4096\*217) \* 2 (buy and sell)

｜｜｜—warrants\_index\_2048x12

｜｜—fix\_formatter

｜—output\_port\_lookup\_cpu\_regs