今天的主題主要分為兩個部分，zynq 7000 系列的主要架構介紹以及利用手上這塊基於 zynq 7000 的 zedboard 來進行 Vitis 實際開發流程的講解

P4. 這邊是 zynq 大致上的整體架構，我們可以看到分成了 Processing system 和 Programmable logic 兩個區塊，其中 PL 端指的是我們 FPGA 的部分，而 PS 端中包含了 APU、memory interface、hardcore IO 等等，其中 APU 內又包含了許多不只處理器在內的元件，都會於接下來的頁面一一來做介紹

P5. 這邊可以看到關於 zynq 系列的一些產品代號，PS 端架構主要的差異會在單雙核處理器以及速度上，而 PL 端於每個不同型號上都會有 logic cells、LUT 等等數量上明確的差別，基本上越往圖表右方的方向就會是越為高階的代表，因此在這之下我們也分為了 Artix 和 Kintex 這兩個系列

P6. 這邊可以看到說不論是 artix 或是 kintex 系列，我們使用的處理器都是以 ARM cortex-A9 為基底，並且主要的差別會在於 Artix™-based 只有 HR Bank 可以做使用，而 Kintex™-based 則有 HR 與 HP 的 Bank，因此在應用的層面上，不同的料件便有相對應適合的領域，可以參考右邊這張圖表的劃分

P7. 再來，剛剛提到了無論什麼系列，處理器都是基於 ARM cortex-A9，這邊會大制介紹一下關於 ARM 處理器系列的一些特點，這邊看到的不論是 ARM7、ARM9、ARM11 等等，都是在 2004 以前所謂的「經典處理器」，2004 年後所誕生的新架構被稱為 Cortex 系列，而其中 Cortex 又分門別類成三個專屬的應用領域，包含 application、Real time 以及 microcontroller

P8. 那這邊開始會進入到整個 PS 端的介紹，PS 端主要包含了四個部分

1. Application processor unit (APU)
2. Memory interfaces
3. I/O peripherals (IOP)
4. Interconnect

那一個一個來看

P9. 首先看到 APU 的部分，除了剛剛一直提到的 Cortex-A9，還包含了 NEON 指令集、FPU engine、L1 Cache、SCU 等等，接下來都會一一介紹

P10. 那相較於前面的架構圖，這邊的各元件間的連線可以更好地顯示出彼此相對的關係，因此我們先把 APU 的區塊放大來看

P11. 這邊就是 APU 內所包含的所有元件，像是剛剛提到的 FPU 與 NEON，
在現有的設計中，FPU 和 NEON 是被綁在一起無法各自分開的，而 NEON 基於 SIMD 的技術，主要目的會是來去加速數據的計算與處理，這邊稍後會有更詳細的說明

而因為我們處理器主要是面向 application，因此 MMU 的目的就在於說可以將虛擬地址映射到物理地址上，這麼做的好處是假設我們有一個大於內部記憶體的程式要做運行，那就可以決定說當下要運行的哪些地址內容再放到記憶體就好，並在需要的時候再與外部做程式片段的交換，來達成大程式在有限的記憶體中執行的結果

32kb 的 I-Cache 與 D-Cache 我們通常會稱為 L1 Cache，I 是指存取指令，D 則是資料，單 CPU 使用不共

享，這邊在後面會與 L2 Cache 有一個更詳細的差異比較

SWDT 和 AWDT 都是 watch dog，但一個是針對整個系統，一個是針對 ARM，簡單來說它會去偵測目前的程式或狀態是否發生了問題像是死迴圈，並有一個 timer 會去做計數，一旦超過時間便會認定異常並進行 reset

TTC 便是指這個計數器內有三個 channel 可以來做獨立計數

System-Level Control Registers 則是會由一組不同的控制器用來控制 PS 的行為

DMA 允許不同速度的硬體裝置來溝通，而不需要依賴 CPU 的大量中斷負載
DMA 傳輸常使用在將資料從一個記憶體區從一個裝置複製到另外一個,
或是將資料從周邊 I/O 搬移到記憶體
不經由 CPU 而是透過 DMA controller 完成

GIC 為一般的中斷控制器，

SCU 的作用在於維持 L1 與 L2 快取的資料一致性，會於後面篇幅再去做更詳細功用的說明

最後配有一個 256kb 的 OCM

P12. 這邊提到了剛剛所說 L1 快取與 L2 快取的差別
L1 雖然相較於 L2 小很多，但它是針對每個 CPU 獨立存取的，因此速度也是最快
而剛剛也有提到他分為兩塊：指令與資料的快取

L2 具有 512kb，可以由 CPU1 和 CPU2 共享存取，並且單 SCU 就會配一個 L2 快取，同時用於資料與指令

兩者的位置與關係可以參考左下角的圖示

P13. 這邊介紹 SCU 主要的功能，SCU 主要會針對多顆處理器之間快取不同步去進行處理，假設今天 CPU1 的任務資料放在 L1 快取，並把任務交給 CPU2，CPU2 處理完之後便將更新的資料放在 CPU2 L1，這時當 CPU1 取用 CPU1 L1 快取時，SCU 就會告訴 CPU1 她要拿的並不是最新的資料，要它更新成 CPU2 L1 內的資料，來達到資料同步的目的

另外其他還有初始化 L2 AXI memory 存取，或是決定現在哪顆 CPU 可以來存取 L2
管理 ACP 的存取，以及透過 ACP 做於 PL 端與 PS 間快取一致性的處理
另外由於 OCM 支持兩個 64 bit AXI slave port，一個 port 就是專用於通過 SCU 對 CPU 的訪問，而另一個是由 PS 和 PL 內其他所有的 master 所共享的

P14. 再來介紹 NEON 指令集與 FPU
NEON 其實是 ARM 基於 SIMD 的思想來去建構的技術，因此與 SIMD 一樣，主要也是提供單指令就能

完成多資料運算與處理，而 FPU 的加入更拓展了 NEON 處理資料的範圍，因此你可以看到最後一行提到 NEON 可以處理的資料型態包含 signed/unsigned 8-bit, 16-bit, 32-bit, 64-bit, or 32-bit float

而在此架構中，NEON 具有 16 個 128bit 的向量暫存器，或是可以拆分成 32 個 64bit 的向量暫存器

下圖可以看到這是 NEON 運作時的圖示，若今天是 SISD 的話，要完成這四個動作就會需要四步指令，增加 delay 的時間，而 SIMD 則可以使所有通道的資料單用一指令就可以完成動作，增加效率

P15. NEON 主要應用到的層面

P16. APU 的部分大致介紹到此，接下來進入 memory interface

P17. 一樣根據前面的圖，這邊可以發現到 zynq 7 系列包含了幾種記憶體 interface 的支援，左方的像是這類非揮發的 flash 就有包含 SRAM/NOR、NAND 以及 QSPI，而右方的揮發性記憶體有包含了 DDR2/3、DDR3L、LPDDR2 這幾種支援的介面

P18. 再來是針對 IOP 的部分

P19. Zynq 7000 系列支持了 hardcore 上的這些 IOP，其中低速的部分像是 CAN、SPI、I2C 等等，高速則有 USB、SDIO、gigabit Ethernet，這些由於是 hardcore 因此不會占用到 PL 的資源

再來是 GPIO 的部分，zynq 7000 提供了 54 隻 PS 端的 MIO 可供使用，但若是使用者覺得還不夠用的話，則可以透過 EMIO 的方式從 PS 拉到 PL 端再拉成 output 的形式去擴展額外的 64 隻 IO

P20. 因此這邊就來比較一下 MIO 與 EMIO 的差異，MIO 屬於 PS 端的部分，因此 PL 端是不可見的，擁有兩種 HR 跟 HP bank 和三種電壓，由於 MIO 是已經定義好的腳位，因此不用再額外去設定 constraint，EMIO 則是從 PS 端拉到 PL 端再從 PL 端形成 output 去擴展 MIO 的方式，會需要去做 constraint 分配腳位

P21. 那並不是所有的 MIO 都可以去擴展成 EMIO，這邊有張表格來去說明哪些 MIO 可以去做 EMIO 擴展，或是擴展成 EMIO 後性能上有一定的限制

P22. MIO EMIO 示意圖，主要說明 EMIO 實際上還是從 PS 端拉出來的

P23. 再來說明 PS 與 PL 之間的連接之前，會先介紹一下由 Arm 定義的 AXI 接口，屬於 AMBA 下協議的一部分，除了 AXI 外，其他像是 APB、AHB 和 ATB 主要用於 ARM core 內部的連接，而 AXI 顧名思義，是為了連接其他外部像是 PL 端的 IP 所存在的，因此像是有些 IP 雖然不是專門做給 embedded 所使用的，但在支持 AXI 的協議下，就可以很方便地透過 AXI 進行連接，而 Xilinx 之所以採用 AXI，其中的一個目的便是統一元件在連接時的一致性

P24. AXI 協議的連接不像一般的 bus 一樣有很多個，AXI 只定義了單個 master to slave 屬於點到點之間的連接，因此不像一般 bus 會需要有某個時刻來去處理哪段 bus 可以用到此條資料的分配情形產生，

所以就像右下角的圖示一樣，不會存在多個 master 去對到單個 slave 的情形發生

P25. 那假設我今天就是想要有多對多的情形呢？Xilinx 針對此問題提出了 AXI interconnect 的 IP，這個 IP 主要的功用在於允許任意的 master 與任意的 slave 進行互連，並且可以根據 bit width 與 clock 進行轉換，達到實現 N 個 master to M 個 slave 的過程

P26. 了解了 AXI 基本的傳輸規則後，我們來看看 AXI 內部的資料、地址以及 response 是如何去做收發的，

第一：

AXI 對於每個資料都有獨立的 channel，因此在定義下你可以看到寫地址有寫地址專屬的 channel，寫 data 有寫 data 專屬的 channel，不過這邊可以發現到，只有寫的 response 有自己獨立的 channel，讀的 response channel 呢？這是因為 Xilinx 規定在讀資料前必須先讀地址，而從讀資料 channel 回來時就已確定了地址跟資料的匹配，因此 response 會隨著 DATA 一起回來，但 write 中都是從 master to slave 的，沒有從 slave 回來的 response，引此才需要多一條 write 的 response 來做回覆

P27. 第二：

非對齊資料的傳輸，有些時候突發性的資料傳輸並不會按照對齊的地址來做讀寫，像是圖中理想上 32bit 的資料要按照 4 個 4 個 byte 的地址來去排，但今天 0 的位置不做資料傳輸，就變成資料要從 1 的地址開始去做傳輸，就會形成非對齊資料的傳輸

P28. 第三：

Burst transactions，以圖解來說的話，AXI 支持讀寫一個地址就可以支持多筆資料的讀寫，像左下角途中就是一條讀寫地只配一條資料，右邊的則是讀寫一地址跑三條 data，當然資料的數量在定義上也有限制，這邊後面會再提到

P29. 第四：

Multiple outstanding reads，這邊是指連續發出 N 組寫地址（寫數據）命令，這期間如果寫響應沒有返回，則必須等待寫響應返回才能接著發寫地址（寫數據）命令，如果有寫響應返回，則返回了幾個，就可以接著發幾組，保持在傳輸上有 N 組的命令在執行，可以大大減少等待的時間，提升效率

P30. 了解了 AXI 的定義以及讀寫過程後，回到一開始的這張圖，AXI 底下其實還有分為

Memory map/full

Lite

Streaming

三類，剛剛所提到的定義都是 AXI full 的功能，而其餘兩種則是基於 AXI full 來去修改或簡化而成的像是 AXI full 就會用於一般需要資料傳輸的時候，AXI Lite 則是簡化過後的 AXI full，用於少資料的傳輸場景，而 streaming 只有 write data 的通道，沒有寫地址與讀的通道，因此多用在大量資料傳輸的場景像是影音的串流

P31. 那前面有說到 AXI 的定義基本上就是 AXI full 的定義，因此這邊來補足一下前面沒說到的 burst length 的部分，burst length 最多可以到 256 條資料，另外 Data width 有 32, 64, 128, 256, 512, 1024 這幾種，Xilinx 支援前面藍色標起來的部分，也支援重疊傳輸，讓 response 間的 latency 降低許多

P32. AXI Lite 剛剛有提到是 AXI full 的簡化版，主要是為了在小資料的傳輸上，並減少 power 與 space 的消耗，因此就比較適合用在單單控制訊號的傳輸，並且資料位寬限制在 32 與 64bit
P33. 最後一個是 AXI Streaming，AXI Streaming 如剛剛所提到的，只剩下 write data 的通道，並且只有一個資料的傳輸方向，不限制 burst length，而 streaming 有幾個名詞，在這邊解釋一下
Transfer 代表一條資料傳輸的過程
Packet 代表多個資料，比較像是前面所說的 burst length
Frame 則是最高層的資料形式，包含多個 Packet
而整體的資料傳輸過程就成為 stream

P34. Minimum control 只會出現在整段資料的最前段與最後段而已，就像此圖的例子，中間一大段都是資料的形式，只有在前端與後端會有其他包括地址的資訊與檢測數據傳輸的雜湊碼而已

P35. AXI 的介紹大致到這邊，接下來就會提到 PS 與 PL 之間是如何來去做傳輸的，PS 跟 PL 之間主要有兩種形式：HP 跟 GP，HP 有四條，適合高速傳輸使用，都是以 PL 為 master，PS 為 slave 的形式，直接連接到的是 OCM 和 DDR，右邊的 AMBA switches 專用於 HP bank 與 ddr 溝通

GP 則有兩條是 PS 為 master to PL slave，另外兩條反之，因為是 general purpose 的關係，性能受到 master port 和 slave port 的限制，因此今天 PS 若想將資料透過高速主動傳給 PL 端，只能先將資料儲存到 OCM 或是 DDR，再由 PL 透過 HP 去做存取

P36. 因此這邊可以看到上圖是 HP 的表示，PL 為 Master 會連接到 DDR 與 OCM 去做溝通，下圖則是 GP

P37. ACP 因為不用透過 switches 的關係，因此延遲會比 AXI HP 小，但傳輸資料也較小
DMA 允許不同速度的硬體裝置來溝通，而不需要依賴 CPU 的大量中斷負載
DMA 傳輸常使用在將資料從一個記憶體區從一個裝置複製到另外一個,
或是將資料從周邊 I/O 搬移到記憶體
不經由 CPU 而是透過 DMA controller 完成

GIC 為通用的中斷控制，PL 到 PS 共有 16 組，Vivado 內可以看到
PS 可以提供四組 reset 到 PL 端以及 4 路時鐘 (FCLKs)，並且每一路時鐘可以獨立配置並且完全異步

P38. 再來這邊就會說明剛剛提到的 PS clock 以及 reset 部分，首先先來看到 clock
PS 的 clock resource 可以來自於外在的 pin 以及三個 PLL：ARM、DDR、IO

P39. Reset 方面，PS 除了最前面有提到的 watch dog timer reset 外，zynq 7000 也支持上殿後 reset，PS_POR_B 是整個晶片最高級 reset（Power-on Reset），通俗點來說，就是整個晶片都會 reset，並重新讀取 mode pin 從 flash 抓資料
PS 端方面的 PS_SRST_B 行為與 PS_POR_B 差不多，差別在於它是 Non_Por 的，並不會 reset 所有的 register，等於是跳過了 POR 在 initial 整個晶片與 PL 的步驟，主要用於系統軟體的調整
warm-reset 代表我去重新 reset 整個系統地執行
p40. 再來講到 zynq 的開機配置，一般而言 zynq 的開機可以分為 secure 與 non-secure，最大的分

別在
於能不能使用 JTAG 進行開機，能用 JTAG 進行開機的話稱為 non-secure，一般都是在 Debug 或是開發
的環境下去運行
再者，zynq 有主要四個開機的 device：QSPI、NAND、NOR 以及 SD 卡
如果 zynq 從這些地方找不到可以開機的資訊的話，便會從 USB、Ethernet 這些 Secondary boot devices
去找
P41. Zynq 在開機的順序主要分成三個階段，第一個階段會從 BootRom 中去執行裡面原本就有的 code
，這份 code 會去確認說目前 BootMode Pins 是什麼狀態，然後來決定要從 SD card、Nor Flash、NAND Flash、QSPI Flash 還是 JTAG 進行開機，注意這部分是 Xilinx 已經寫好的，不能被修改
P42. 假設我們今天是從 SD Card 開機
Boot Loader 會試著將 SD Card 內的 BOOT.BIN 檔載入到 OCM 並且開始執行這個在 OCM 內的 BOOT.BIN
那麼 BOOT.BIN 裡包含了什麼呢？
總共有三個：
1. FSBL
2. PL Logic bit files
3. SSBL
P43. 再來第二階段我們會去執行放在 OCM 裡面的 FSBL，FSBL 是由 SDK 所產生的，它的作用在於會
將 PL logic bit file 來載入到 PL 端，並且設定 PS 端的 PLL 以及進行基本外設 IO 的配置，最後會呼叫
SSBL 並將整個控制權交給 SSBL 與進行整個 OS 的載入
P44. 那這邊 Bitstream 也就是 PL 的配置是可選的原因為，因為 PL 的配置相對來說較久，有些開發者
會先想讓某些程式先開始執行後再去配置 PL，因此這邊的 PL 配置可以選擇放在 FSBL 這邊或是 SSBL
的時候再去執行
P45. 最後一個階段是執行 SSBL，SSBL 的檔案一般為 u-boot.elf，U-Boot 可從 SD card 獲得 kernel image
預設上 U-Boot 會開始執行 autoboot 的程式，並且會再從 BootMode Pins 上去確認要從哪裡載入 Kernel
Image
P46. 一般來說載入的步驟會有三個：
1. 第一，從 SD Card 上讀取 kernel image (zImage) 並將其 copy 到記憶體上
2. 第二，會讀取 DTB File (devicetree.dtb)
3. 第三，會讀取 zipped ramdisk file system (ramdisk8M.image.gz)
全部載入完畢後，U-Boot 就會開始執行 kernel image
那如果在沒有 OS 的環境下，SSBL 就會轉而執行要直接 run 的程式