國立交通大學

資訊科學與工程研究所

碩士論文

應用於 Java 加速器的堆疊記憶體及系統軟體設計

研究生：林子剛

指導教授：蔡淳仁 教授

中 華 民 國 一 百 年 九 月

應用於 Java 加速器的堆疊記憶體及系統軟體設計

Design of Stack Memory Device and System Software for Java Accelerator IP

研 究 生：林子剛　　Student：Zi-Gang Lin

指導教授：蔡淳仁　　Advisor：Chun-Jen Tsai

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

June 2011

Hsinchu, Taiwan, Republic of China

中華民國一百年九月

摘要

本論文試著提出以軟硬體協同設計的方式，設計一個 JAVA 加速處理器(Java Accelerator IP, JAIP)配合任何通用處理器（General Purpose Processor, GPP）來執行 JAVA 程式。論文的重點分成軟體架構和硬體架構的設計。在硬體方面，我們設計了一個針對 Java Virtual Machine (JVM) 運作特性所設計的客製化 4-port memory，做為在 Java 加速器的 stack memory，在較低的硬體成本的情形下，可以降低 Java 加速器 double-issue 時因為 local variable accesses 所造成的結構危障(structure hazard)。

在軟體方面，我們是以 Java 語言的 dynamic class loading 的運作模式，來設計我們 GPP 和 JAIP 的系統軟體整合介面。我們設計的介面，僅需 GPP 系統平台提供中斷服務的功能以及標準 C 語言的函式庫，就可以讓我們整合 JAIP 至任何作業系統的環境裡。另外我們在系統軟體中設計了快速原生方法（Native Method）呼叫的功能，以支援 JAVA 系統物件型別（system classes）中的系統功能呼叫。整體而言，我們所提出的軟硬體協同設計的 Java 加速系統同時具備易整合以及高相容的特性。

配合這樣的架構底下，論文當中也在 Xilinx 的 FPGA 上實作出我們所提出的堆疊記憶體，以及完整的系統軟體以進行驗證。特別是我們完整的支援 JAVA 物件導向特性中的繼承與介面機制、以及動態連結等機制。並把 Java Micro Edition 中 CLDC 的大部份系統物件型別移植到我們的平台上。

誌謝

終於能夠寫到這一頁，我相信這是所有學生都期盼的一刻，而如果有幸能讓你翻開這篇論文，希望你能知道這頁也是我寫得最認真的部分之一，因為在我念大學的時候，有很長的一段時間都在圖書館打工，其中一項工作就是將每年畢業的碩博士論文建檔，那時候的我總會翻開來看的，就是這頁的內容，最大的理由當然不外乎是這頁我比較看得懂（笑），一邊看著誌謝一邊評論著這個人應該沒什麼朋友又或是這個人應該過得很充實之類的，所以我希望本文寫下的內容可以不至於讓我的青春留白。

首先能順利在一年半的時間內完成碩士學位，最要感謝的就是我的指導教授－蔡淳仁老師，感謝老師在求學期間對我的諄諄教誨，雖然在研究過程中遇到諸多瓶頸，但是沒有磨練也就不會有成長，老師常提的學校其實給了學生許多犯錯的機會，我想我將來一定會很懷念在老師指導的這段歲月，懷念老師對學生的寬容以及關心；再來要感謝的就是我的家人，感謝父母對我的栽培以及兄長在我人生及求學上的諸多指引，家人的支持是我總是能夠面對挫折與失敗的力量，而我求學的生涯能夠如此順遂，其實是因為我總是走在別人已經走過的路上，感謝上天賜給我一位比我還優秀的哥哥；另外還要感謝在交大幫助過我的許多人，包括實驗室的成員、NCTU 健身猛男壯女團、修課認識的朋友以及大學認識的同學，這些都是陪我在異鄉一同打拼的好夥伴，也要感謝陪我走過低潮的小學同學欣眉、大學同學裴呱、陳承聖以及實驗室學弟子敬。

文末，我很想學網路上的老梗說，我要感謝我的女朋友，感謝你這二十多年來沒有出現過，讓我得以致力於學業，但是沒想到在這邊我確實需要感謝這麼一個人，謝謝妳短暫的出現點綴了我的碩士生涯，讓我有哭有笑的渡過最後的學生生活，謝謝這些發生在我生命中，最大與最小、最重要也最微不足道的事。

目錄

圖目錄

表目錄

第一章　前言

1.1. 研究動機

自 1995 年 Java 程式語言問世至今，隨著其具備安全、跨平台、物件導向...等優點，一直備受矚目，而現今更因為其跨平台的特性，在愈來愈多的嵌入式平台應用上都廣泛的被選擇做為開發程式的主要語言。

跨平台的特性來自於透過 JAVA 虛擬機器（JVM）[2]的技術，使得相同的程式可執行於不同的平台上，達到高可攜性，然而為了實現這項特性，所付出的代價是效能上的損失，其原因在於 JAVA 程式在被執行前會先轉譯成位元組碼（bytecode），再透過各平台上的 JVM 來執行，在此特性下，我們要探討的一點就是當隨著消費性電子的需求增加，使得在愈來愈多的嵌入式系統中都需要具備能夠執行 JAVA 程式的環境，其伴隨著的就是我們需要移植 JVM 的平台也增加許多的問題。

而在此研究中，我們試著提出高整合性的解決方式，以 JAVA 加速處理器並透過軟硬體協同的設計架構來執行 JAVA 程式，這樣不同於一般在作業系統上安裝 JVM 的做法，可以避免在移植 JVM 上所需要花費的時間，而且透過我們設計簡易整合的介面，僅需系統平台提供中斷服務的功能以及標準 C 語言的函式庫，便可將我們的軟硬體整合至系統中，加上無需仰賴作業系統的設計，更可以讓我們整合至已安裝任何作業系統的環境裡，期望透過其具備易整合以及高相容的特性來解決前段所述的問題。

1.2. 研究目的及貢獻

在本篇論文中提出的貢獻，第一部分是在硬體上的實作及針對系統所使用的記憶體配置上做效能的測試，實作內容繼承至[4][5][6]的研究結果，修改 JAVA 處理器所使用的堆疊記憶體架構，因為在處理器的設計上我們使用的是雙指令的處理器架構，表示處理器可以在同一時間執行兩個指令，由於這項技術的關係，使得在每個周期時會產生兩組分別位在解碼級及

執行級的雙指令，可能會對堆疊記憶體進行存取的行為，造成系統出現結構危障的情形，雖然在任何處理器的設計上，危障的發生都是無法避免的問題，但是由於 JAVA 語言所產生的虛擬機碼是將指令與其操作的運算元，會合併一起放在轉譯完成的映像檔（.class 為副檔名）上，這樣交錯指令與運算元的設計，會讓電路在偵測對堆疊記憶體操作時，部分指令需要解讀完所需的運算元資訊才能判斷其危障產生與否。

而為了配合在新架構中提出將危障偵測機制提前至提取級處理的設計，針對這樣的考量，我們提出以增加四個暫存器做為堆疊中呼叫方法的前四個區域變數的快取設計，屏除新架構中仍會在解碼級及執行級出現結構危障發生的原因，最後再將 2-level 中的第二層堆疊記憶體包裝成模組，在其內部實作 forwarding 的機制，以簡化外部對堆疊記憶體的控制訊號。

第二部分則是針對系統軟體設計的實作，首先針對 JAVA 語言在物件導向的特性－繼承及介面，繼承這項特性在 JAVA 語言中指得是允許物件型別的部分內容是來於其他的物件型別所定義，而介面的特性指得是當我們在不同的物件型別中都實作相同的介面時，我們能透過這個統一的介面去操作這些所屬於不同物件型別的物件內容及方法。

我們透過對物件型別解析器（Class Parser）的修改實作出支援物件型別的繼承與介面呼叫兩項特性所需要的相關資料結構，完成系統這兩項重要的功能，並針對新修改的資料結構，我們需要再對原先系統已提供的 ISR 重新再做修正；最後我們實作在 JAVA 語言中呼叫由系統物件型別（System Class）裡所提供原生方法（Native Method）的機制，支援原生方法的呼叫在我們的設計中，一樣是先透過發出中斷的方式，使得 RISC Core 跳到其原生方法實作的 ISR 中，最後再根據我們系統平台所定義的資料結構去撰寫符合我們需求的原生方法，完成以上軟體的實作，也讓我們的系統對 JAVA 語言在支援上更完備及有彈性。

1.3. 論文架構

本論文一共分為六章，本章是一個概括性的導論，說明背景、動機以及貢獻；第二章為文獻探討，會先介紹目前增進對 JAVA 執行效能的做法，並接著介紹與論文實作相關的 JAVA 處理器堆疊架構的設計方式，之後再針對軟體的部分開始介紹動態載入的機制及處理器溝通介面等的相關研究；第三章開始是對全系統架構的說明，並以硬體的修改實驗為主；第四章則是從軟體部分對其系統延伸的功能實作部分做詳細介紹；第五章則分析各項實驗的結果；最後在第六章則是提出結論及未來可能的研究方向。

第二章　相關研究

2.1. 提升 JAVA 執行效率的方法

早期的 JAVA 虛擬機器在實作時都只是軟體解譯器（Interpreter），而這些虛擬機器又大多都需仰賴底層作業系統的幫忙，像是 Sun 的 CVM[7]與 KVM[8]就是在嵌入式裝置中利用軟體實作 JAVA 虛擬機器的直譯器[9]，但是透過實作解譯器（Interpreter）完成以堆疊為基礎的虛擬機器在執行 Java 程式時相當沒有效率。因此大部份商用的 Java 執行環境都會提用加速化的虛擬機器。目前常見的做法主要有三種[3][10]：不相容於標準 Java VM 的最佳化 interpreter (利如 Android 的 Delvik VM)、JIT（Just-in-Time compilation）[12][13]即時編譯的 Java 加速技術、及使用硬體加速的方式。

我們提出的系統，將針對嵌入式的環境底下，對記憶體資源的要求較為嚴苛的應用設計，所以我們是採用硬體加速的方式來設計 Java 加速系統。常見使用硬體執行 JAVA 的方式有四種[10]：獨立式的處理器像是 Sun 的 picoJava[17][18]及 aJile 的 aJ-100[15][16]屬於此類，其處理器的設計可單獨執行 JAVA 程式，協同式的架構像是 InSilicon 的 JVXtreme[19][20]屬於此類，在使用上需搭配其他處理器來完成執行、內嵌轉譯器（embedded Java translator）的像是 ARM 的 Jazelle[22]及 Nazomi 的 JSTAR[21]屬於此類，其具備可以切換成執行 JAVA 位元組碼的處理器設計，最後一種是以硬體實作即時編譯器的機制，像是 Chicory System 的 HotShot[14][23]屬於此類；而我們系統選擇以協同式架構設計，其最大的原因在於可以將其架構設計成具備易整合的特性，並且因為在我們的架構中，JAVA 位元組碼（byte code）的指令絕大部分執行上，是不需要仰賴 RISC Core 來執行 ，僅在有需要呼叫底層方法實作的指令，才會透過 IPC 介面呼叫 RISC Core 來協助，這樣的設計下使得我們無需像其他 JAVA 加速處理器的設計[10][11]，會有需要移植 KVM 至平台中讓 RISC Core 也能執行 JAVA 位元組碼（byte code）的負擔。

2.2. JAVA 處理器的堆疊架構

因為 JAVA 語言被設計為執行在堆疊機器[3]上，所以在執行的指令中充斥大量對堆疊的操作，這邊將介紹關於不同 JAVA 處理器，為了加速 JAVA 處理器在堆疊操作上，各平台所提出不同架構的堆疊快取機制，這邊將常見的架構分為三類，像是有使用 Register file 作為快取的設計、on-chip 的記憶體作為快取的設計以及同時使用 Register file 與 on-chip 記憶體作 2-level 的快取;使用 Register file 作為處理器的堆疊快取架構，舉凡像是 Sun 所設計的 picoJava[17]、aJile's JEMCore[24]及 Ignite[25] processor 皆是以這樣的架構去設計堆疊的快取，然而在這樣的設計下因為暫存器的資源昂貴，所以都會有數量上的限制，以上的處理器在使用 reigster file 作為堆疊快取的數量介於 16~64 個，當使用超過數量的資料時便需要對記憶體與堆疊的快取暫存器做般進（Fill）與搬出（Spill）的動作，當系統需要執行複雜程式時會這樣的情形也就容易發生。

第二類使用 on-chip 記憶體作為堆疊快取的像是 Komod[26]oand FemtoJava[27]，使用 on-chip 記憶體的好處在於快取大小的尺寸就不會受到第一類的限制那麼大，但是使用這樣架構的設計會衍生出的問題是，當處理器使用管線技術來加快效能時，在同一個週期內對堆疊存取會造成需要使用三個阜（port）的情形，舉例來說在 Store 指令後接的是 ALU 的動作，則執行第一個指令時會需要先在寫回堆疊資料時，而同時第二個指令需要抓取堆疊上的兩個值，會造成硬體上的結構危障，避開這樣情形的解決方案是使用 three-port 的 on-chip 記憶體，但相對的成本也就高出許多。

第三類使用 2-level 的架構，如 Schoberl [28]提出的 JOP（Java Optimized Processor）即採用此架構，第一層先使用兩個暫存器表是堆疊最上層的兩個運算元，在第二層的部分使用 on-chip 記憶體，這樣的設計機制可以在硬體僅需增加些微的成本卻能兼顧以上兩種的優點，（如圖 1 所示），當運算指令執行時會從 A、B 暫存器內取值出來計算，並寫回堆疊上適當的位置。

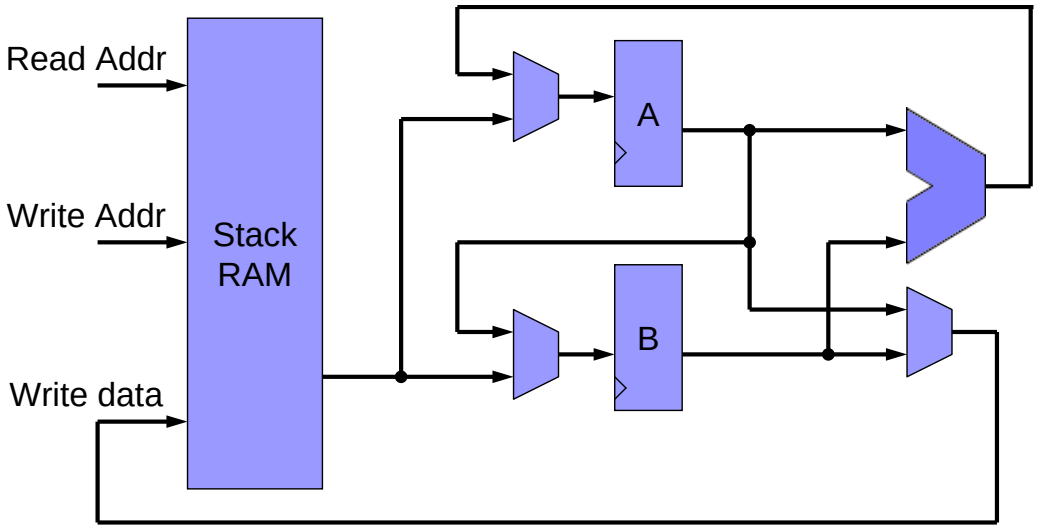

圖1. 2-level 堆疊記憶體架構

在此我們的堆疊架構承繼 JOP 的設計，並更進一步因為我們系統的 JAVA 處理器採用雙指令的架構，所以除了 2-level 以外，還需要再提供額外的設計，包含增加第一層暫存器的數量至三個，以供我們雙指令同時執行的運算，在第二層的將採用交錯式的記憶體架構，因為當堆疊最上層移除一個運算元時，我們需要對堆疊的內容做調整，將 B 暫存器移至 A，C 暫存器移至 B，並從第二層抓取最上層的值至 C 暫存器，而這樣調整的動作就需要使用交錯式記憶體的架構才能同時完成；另外針對 JAVA 位元組碼（byte code）的特性，去增加四個暫存器至我們堆疊的架構，以滿足特殊指令對堆疊的操作，(如圖 2 所示)。

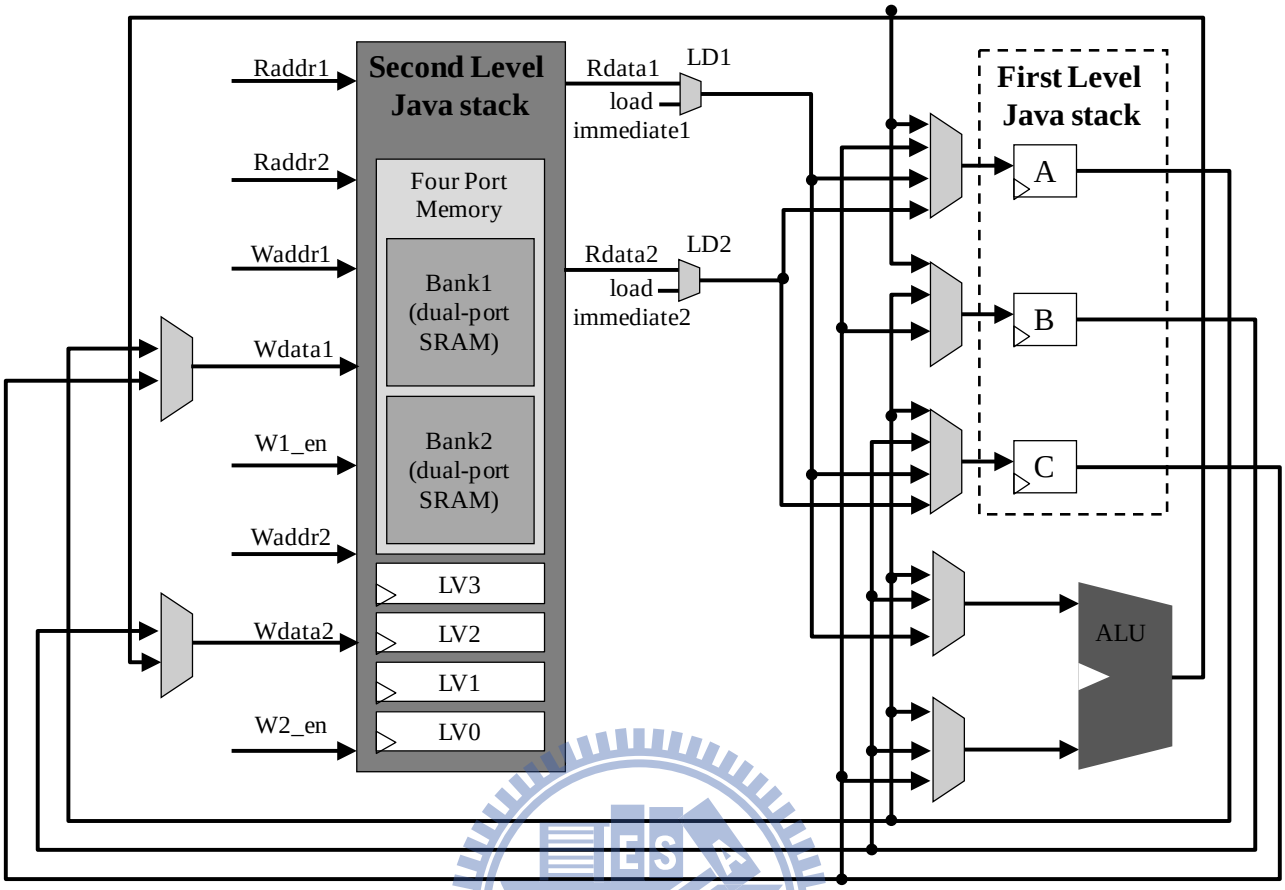

圖2. JAIP 中實作的 2-level 堆疊記憶體架構

2.3. 動態載入物件型別的機制

這部分要介紹的是關於動態載入 JAVA 程式的機制，在 JVM 的實作中提供了幾種關於將 JAVA 的物件型別檔載入時機[29][30]的可供選擇，像是非主動式（lazy）的載入行為或是由使用者自行定義載入的行為等，而 JVM 在嵌入式的平台預設的就是以非主動式方式去做載入，（如圖 3 所示）是 JVM 實作物件型別載入的機制。

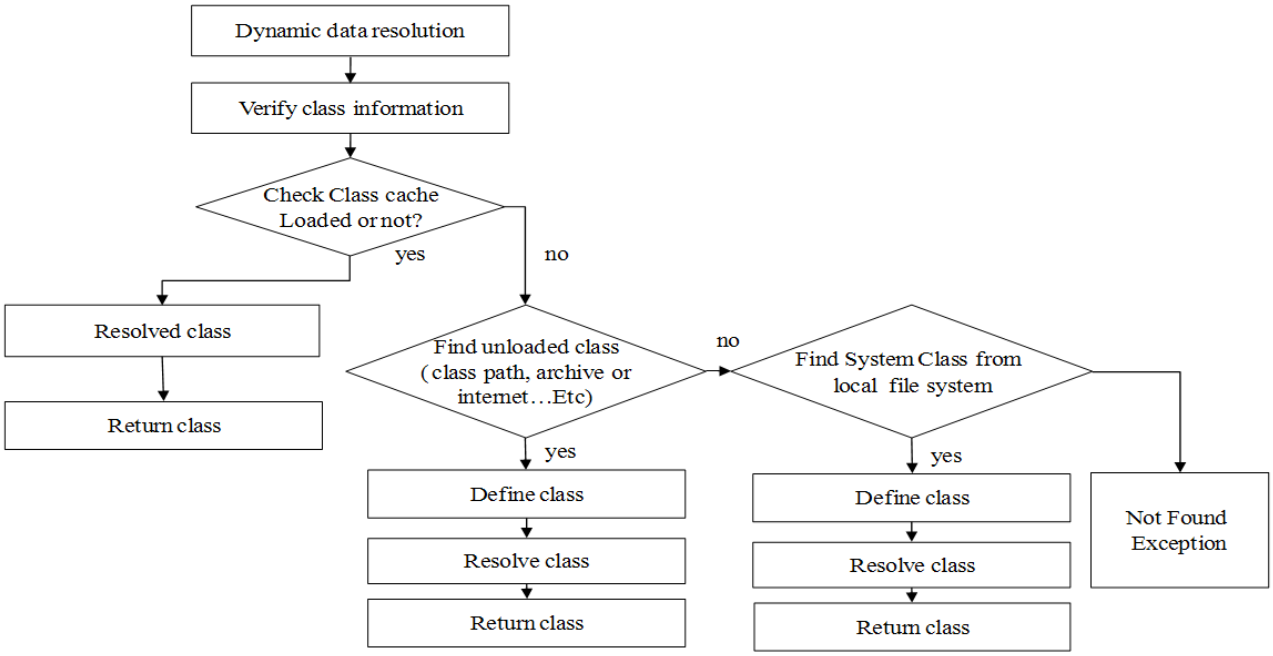

圖3. JVM 物件型別載入的機制

延續我們之前的研究，在我們的平台將動態載入物件型別的機制分為兩部分，第一部分是解析的動作，而為了實作繼承、介面的機制所以對我們系統中所提供的物件型別解析器（Class Parser）做了適當的修改，並在這部分新增了支援由原生方法中的－”forName”所觸發的動態載入物件型別時機；第二部分為載入的機制，這邊的載入動作指得是將執行解析過後的執行映像檔（Run time image）載入至我們 JAVA 加速電路中的快取記憶體，針對載入的時機仍保留 JVM 在嵌入式平台的設計，以非主動式（late-resolution 或 resolution-on-demand）的方式去載入。

2.4. 處理器溝通介面設計

在目前愈來愈複雜的行動多媒體應用的嵌入式系統，由於任務的多樣性，使用單一處理器的設計架構未必能滿足其所有需求，相反的在這樣複雜的系統下卻相當適合以異質（Heterogeneous）整合的雙核心甚至多核心平台來處理，常見以這樣方式為解決方案（solution）的應用像是整合了一個精簡指令集處理器（RISC Core）和數位訊號處理器（DSP）的架構[31]，這邊我們在設計執行 JAVA 這樣跨平台特性語言的複雜系統，也同樣提出了以異質雙核心的

架構來使用 JAVA 加速處理器協助執行 JAVA 程式。

而在實作關於 IPC（inter-process communication）的部分，其中一種方式是透過實作 Java Native Interface（JNI）的[32][33]機制，JNI 的機制是 JAVA 所提供的應用之一，其使用的方式有兩種，一種是讓 JAVA 撰寫的程式，能夠呼叫到由底層實作的程式語言撰寫的原生方法（native method），另一種則是可以讓原生方法撰寫的程式透過 JNI 存取 JAVA 的資源，因為其設計需考量到其通用性及彈性，在實作上會造成許多額外的負擔，像是去做堆疊記憶體資料結構的轉換以及要能動態去載入相關的函式庫（仰賴作業系統的輔助）等，故因其額外的負擔以及對作業系統仰賴的特性，不適用在我們的架構；所以在我們系統的實作上，我們採用較簡單的介面完成單方向呼叫的機制，即讓 JAVA 加速處理器可以透過這個機制呼叫到 RISC Core 所提供的服務，設計上我們採用 8 個暫存器做為 MailBox 的傳遞機制，以透過中斷處理的方式，將 JAVA 加速處理器所要傳遞的參數，傳遞給 RISC Core，並執行相對應觸發的 ISR，這樣簡易的設計也有助於我們未來整合系統至其他平台中。

第三章　硬體架構介紹

在這篇論文裡，我們提出一個可供應用於嵌入式環境底下，具備可重複性使用及易於整合的 Java Accelerator IP （JAIP）電路架構，JAIP 為一個用於加速執行使用 JAVA 程式語言所撰寫程式的執行引擎，並透過我們提供的系統軟體輔助來對完整的 JAVA 特性提供支援。

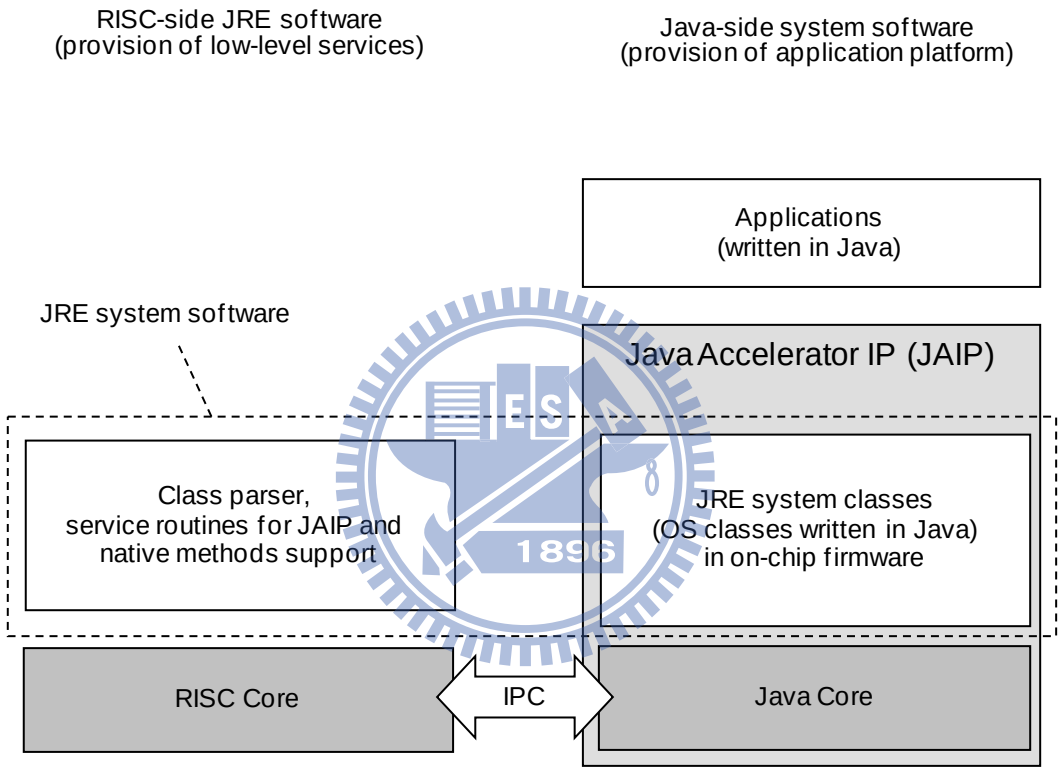

圖4. 系統軟硬體的功能描述

對於整個系統分為兩個部分，(如圖 4 所示)，第一部分為執行 JAVA 程式的電路執行引擎，用來執行我們 JAVA 位元組碼（byte code），第二部分則是透過 RISC Core 來執行我們所提供的系統軟體，這部分包括物件型別解析器（class parser）對 JAVA 部分指令支援所提供的程序以及實作系統物件型別所提供的原生方法（Native Method）實作，並且在系統軟體上的實作上，我們是採用 Sun Microsystems 所提出的 JavaOS model [1][2] 僅需仰賴平台提供的精簡系統函式庫，做為底層硬體的 Hardware Abstraction Layer (HAL)無需仰賴全功能作業系

統（如 Linux、WinCE 等等）的支援，這樣的特性也讓我們可以整合至任何現有的作業系統中。

這一章著重的方向為對硬體層面的描述，我們主要會在開始說明硬體實作部分前，先在 3.1 節簡介 JAIP 內的各個模組，等對 JAIP 有初步的認識之後，從 3.2 節之後再說明本論文對硬體架構實作上的貢獻。

3.1. Java 核心管線硬體架構

在 JAIP 內部的設計我們主要切分為四級的管線設計，(如圖 5 所示)。以下分別針對不同級的功能做介紹。

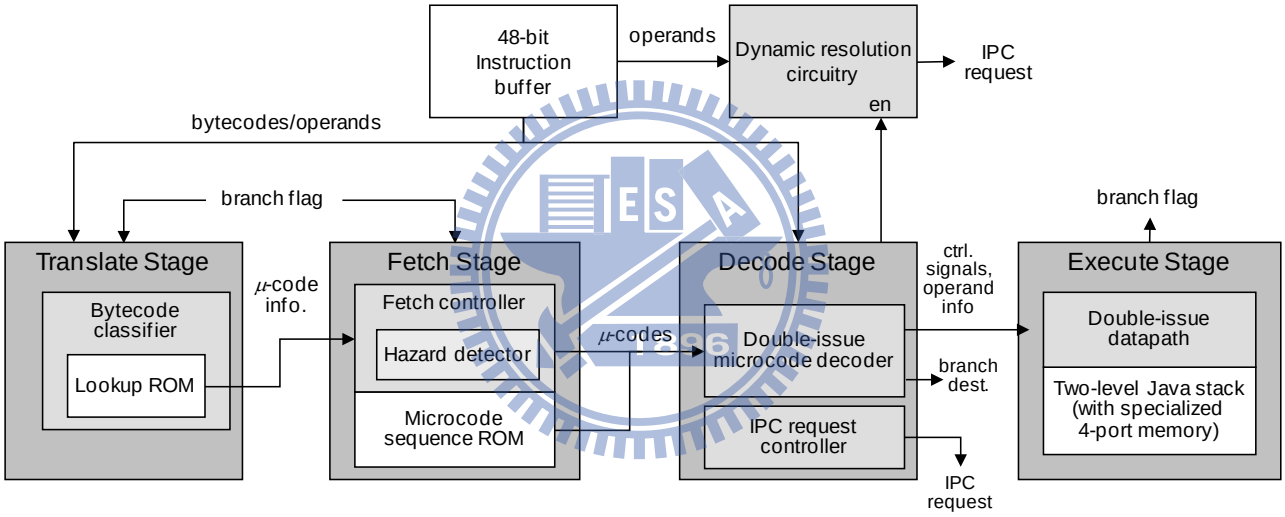

圖5. JAVA 位元組碼執行引擎的管線架構

3.1.1. Translate Stage 說明

為了實作上的簡化，我們自己定義了一套 ISA 以執行 Java bytecode。並將 Java bytecode 型態分類成 simple、complex、operand 三種。大部分 Java bytecode 的指令都是一對一的轉換，我們稱之為 simple instruction。少部分的指令（ex: invoke）則是轉換成 code sequence，我們稱之為 complex instruction。附帶在 instruction 後面的參數則稱之為 operand。

Translate stage 的任務很簡單，僅只是按照 Fetch stage 的訊號來選定 instruction buffer 內

幾個 bytecode，並將其轉換。Translation 是透過一個 ROM 來實作。使用 Java bytecode 當作 address 到 ROM 去讀取該 instruction 的資料。每筆資料都含有三項資訊：instruction 長度、IsComplex、mapping data。

Mapping data 會隨著 instruction 不同，而代表不同意義。Simple instruction 直接 map，所以 mapping data 即為 ISA instruction。若為 complex instruction，則 mapping data 為 u-code address，Fetch stage 便根據此 address 讀出一連串預先寫好的 instruction pair。而 Operand 的取用是由 decode stage 與 execute stage 依狀況至 instruction buffer 內取出，理論上並不用轉換 Operand，但是因為是以 ROM 為實作方式，在 Translate stage 無法得知更多訊息，所以 Operand 轉換後的資訊也會送到 Fetch stage，而到 Fetch stage 才有足夠訊息去判斷是否為 Operand，若是則忽略之。

3.1.2. Fetch Stage 說明

Fetch stage 的工作主要有三項，從傳送真正要執行的指令給 Decode stage、控制 Translate stage 的 instruction buffer 解讀方式以及當 branch 相關的 instruction 出現時，記錄該 instruction 所在的 program counter。

經由從 Translate stage 透過 ROM 解讀出的資料判斷分析後，即可分類由 Translate stage 傳來的 16bit 的資料，前後位元組各屬於 simple、complex、operand 哪一種，最後在根據這三種的排列組合傳送指令給 Decode stage，這邊包含傳送複雜指令轉換後的 u-code sequence、合併簡單指令...等等。

3.1.3. Decode Stage 說明

當 Fetch stage 把真正要執行的指令傳到 Decode stage 之後，Decode stage 便會 decode 傳過來的 instruction 並產生相應的 control signal 以及拉起某些 flag，Decode stage 大致可分成幾個部份 ，控制 operand source 的 MUX、special decode unit、normal decode unit、branch destination 計算。

控制 operand source 的 MUX，主要的功能是因為在 Fetch stage 解讀方式的不同，造成

operand 目前在 instruction buffer 內的位置也不同，而在 Decode stage 也需做出相對應的調整；一般 instruction 進到 Decode stage 後都會交由 normal decode unit 分析並產生 control signal，但是有些 instruction 的行為較特殊，不可與其它 instruction 並行執行(ex: branch)，所以其 control signal 須透過 special decode unit 來產生；一般而言 branch destination 的計算是愈早愈好，但是由於在 Java bytecode 的指令中，其計算需要使用到 operand，而且若為 condition branch，還需要參考到 top of stack 的 data，所以我們 branch destination 的計算安排在 Decode stage 中。

3.1.4. Execute Stage 說明

Execute stage 的工作為忠實的執行由 Decode stage 所傳來的 control signal 而在 Execute stage 最為重要的部分即 stack 與 stack operation 的實作，Java virtual machine 為 stack machine，所以 stack 與 stack operation 的實作亦為設計重點（我們在 3.2 節討論堆疊架構設計時會另外詳細說明這部分的架構）。

3.2. 堆疊記憶體架構設計

我們在 JAIP 使用的堆疊記憶體是採用 2-level 的設計架構，第一層使用的是三個暫存器表示在堆疊中最上面的三個值，分別以 A、B、C 從堆疊最上層開始依序命名，第二層則是使用兩塊 Dual-Port BRAM 來實作交錯式（Interleaving）的記憶體架構，並針對每一塊 BRAM 我們將 Dual-Port 分成 Read-Only 以及 Write-Only 的兩個 Port，雖然在 Dual-Port BRAM 的使用上原先並無這樣的限制，但這樣的使用方式可以簡化我們在解碼級（Decode Stage）的所需送出控制訊號線的複雜度，有助於我們的硬體設計。

3.2.1. 2-level 設計架構說明

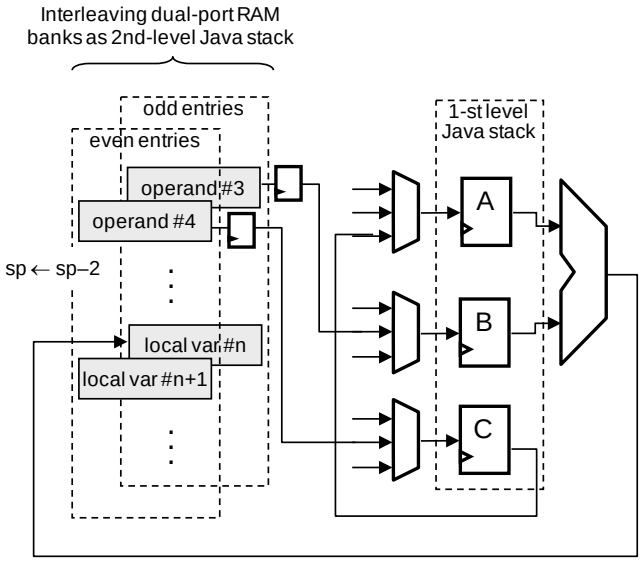

圖6. Store-ALU 對堆疊記憶體的操作

使用三個暫存器表示堆疊最上層三個值，這樣設計的理由是因為我們在雙指令架構處理器執行的過程中，排列各種指令的組合，以目前不支援長整數型態運算的設計下，最多只會同時使用到堆疊上的三個值，我們實際以 Store-ALU 指令的執行為例，(如圖 6 所示)，因為雙指令會同時執行的關係，處理器會將 A 暫存器的值存回堆疊中，B 暫存器跟 C 暫存器運算的結果會存入 A 暫存器，最後在同時從第二層交錯式的記憶體中讀取兩個值更新至 B、C 暫存器，接著調整堆疊的指標（SP）位置減二，完成執行 Store-ALU 後堆疊上資料變化的操作（這邊 A、B、C 暫存器內執行指令前的值以 operand#0～operand#2 表示）。

在原有的堆疊記憶體架構中，第二層的記憶體已採用交錯式的架構實作，為的是能夠支援我們雙指令架構處理器的設計，如前段所描述的操作，用來輔助三個暫存器的機制能從第二層記憶體讀取兩筆資料至 B、C 暫存器，也因為 A 暫存器的值需要寫回，所以同時間至少需要對第二層記憶體做三筆資料的存取，也是我們使用兩塊 Dual-Port Bram 的原因，而本篇論文在堆疊記憶體設計貢獻其中之一，即實作將交錯式記憶體的架構包裝成模組，抽象化包裝之後來簡化外部控制訊號的複雜度，像是 Forwarding 的機制在新設計中也將由記憶體內部來實作。

3.2.2. 使用 4 LV（local variable）registers 說明

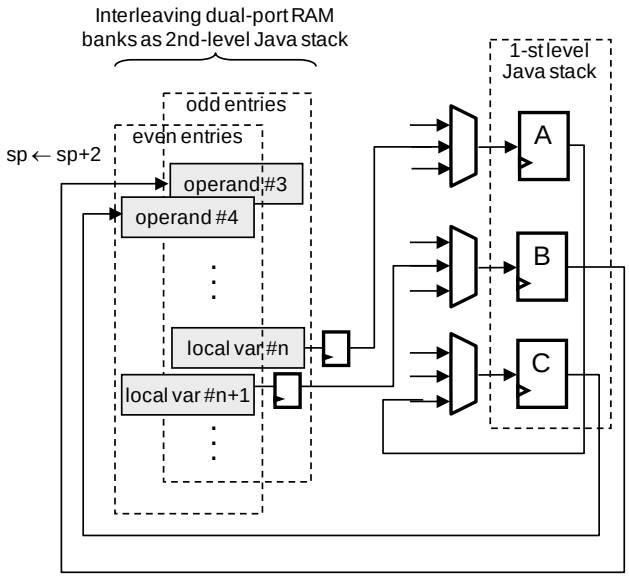

圖7. Load-Load 對堆疊記憶體的操作

另外一項貢獻為提出區域變數快取設計的機制，因為 Sun 在 JAVA 語言制定上了解在程式執行的過程中對前四個區域變數的使用率最高，所以在指令集設計中對前四個區域變數存取的指令是不需要使用 Operand，而這樣的設計會讓我們雙指令架構的處理器，會同時出現兩個存取堆疊的指令，在部分狀況下便有可能需要同時讀取同一個 BRAM，造成結構危障的發生，實際以 Load-Load 指令的執行為例，如（圖 7 所示），當我們執行 Load-Load 指令時，我們會需要將 A 暫存器值寫入 C 暫存器，而 B、C 暫存器值寫回第二層交錯式的記憶體中，接著從堆疊中取出區域變數，放置 A、B 暫存器中，但是在從堆疊中取出兩個區域變數的動作，即有可能造成對同一塊 BRAM 做讀取的動作，所以針對這樣的特性，我們在堆疊記憶體的電路中新增了四個暫存器作為前四個區域變數的快取，用來支援對前四個區域變數的存取指令，這樣的機制是因為會造成兩個指令都是讀取堆疊的指令的情形，僅在對區域變數前四個操作的指令才可能會發生，而需要跟取 Operand 的讀取堆疊值指令並不會與其他指令合併執行，新架構的堆疊記憶體會將對區域變數前四個操作的指令轉為對快取暫存器的讀取避開結構危障的產生，讓新架構硬體在實作上，也可以將危障偵測的機制往前到提取級（Fetch

Stage）實作，有助於我們減低在後幾級電路才偵測到危障發生時的負擔。圖 8 是我們設計出來的堆疊記憶體架構。

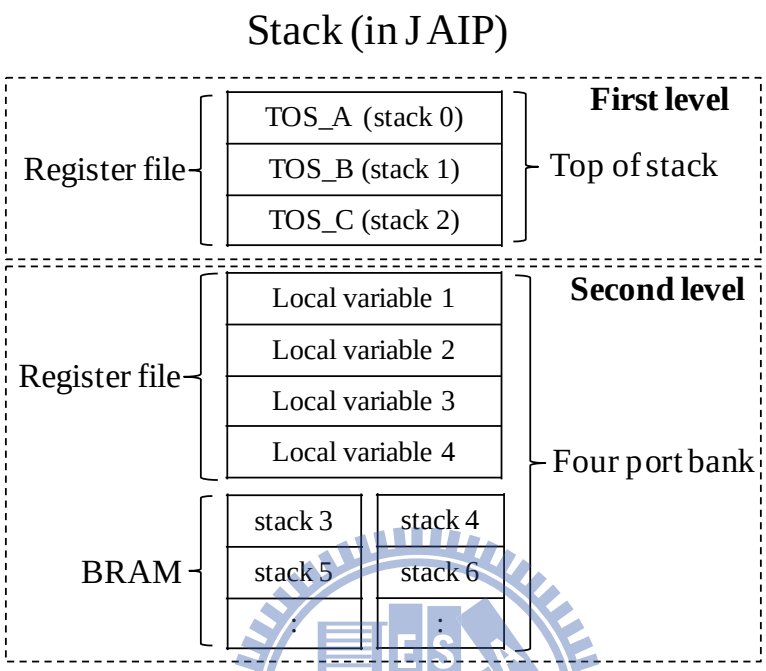

圖8. 堆疊記憶體架構圖

3.2.3. 呼叫方法（invoke）及結束方法（return）時的操作說明

使用暫存器為前四個區域變數作快取的機制，這邊要討論在呼叫方法及方法結束時的機制設計，需要額外做更新快取暫存器的值或是將快取暫存器值寫回 BRAM 中的處理，先介紹當方法結束時電路的機制，當方法結束時需要拉起 Mem2Reg 並送入適當的 BRAM 位址來將前一個呼叫方法的區域變數放回暫存器上，這邊如果前一個方法使用的區域變數少於四個，仍抓取四個連續記憶體的值放上暫存器，因為指令不會使用到超過擁有的區域變數，所以不影響運作，而當我們去呼叫方法的步驟就相對比較繁複，需先將目前方法的區域變數寫回 BRAM，如果目前方法使用小於四個區域變數時，仍會將暫存器上無效資料寫入 BRAM，而寫入的位址為記憶體最高位 ，並將呼叫的下一個方法需要傳入的參數寫到暫存器上，所以在呼叫方法時 Reg2Mem、Mem2Reg 兩個訊號都要拉起，目前設計為可以同時拉起，故我們呼

叫方法及結束方法時在理想情況都只需要兩個周期，但這邊拉起訊號的時機主要是配合電路的 code sequence 來運作，設計上會安排將更新區域變數暫存器的這些時間隱含在原先呼叫方法及結束方法的時間中，使得不影響電路執行的效能。

3.2.4. 外部訊號功能簡介

我們所設計的 4-port memory 並不是通用型的 4 通道記憶體，而是針對我們所設計的 double-issue JAIP 的堆疊存取需求所設計的。該邏輯單元的對外連接埠設計如圖 9 所示，而其內部架構則是在圖 11。

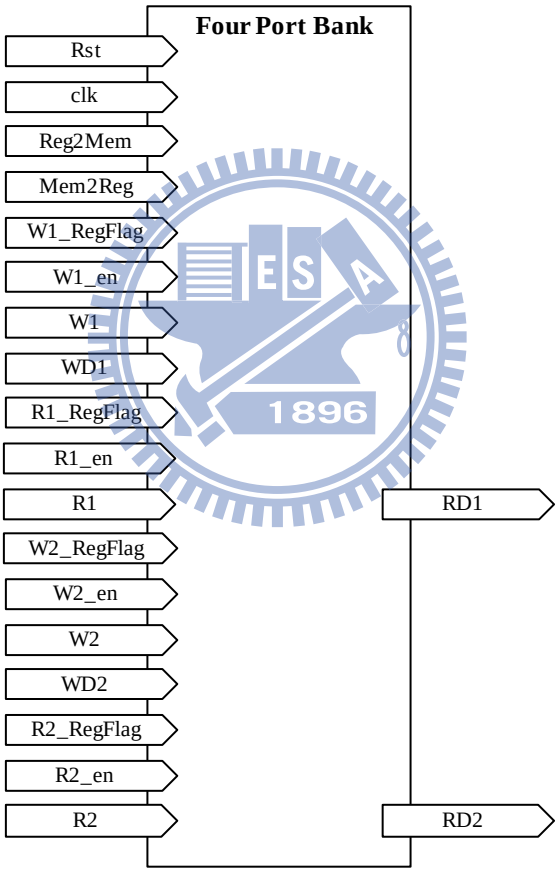

圖9. 抽象化後 Four-Port 記憶體的對外訊號線

因為管線（pipeline）架構的設計(參見圖 10)，我們在對堆疊作讀取的動作會在解碼級（Decode stage）送出訊號，而對堆疊作寫入的動作則保留在執行級（Execute stage），所以在每個周期內會有兩組雙指令封包（instruction package），一組來自於解碼級的對堆疊的操作可

能會有讀取R(Read)的需求,另一組來自於執行級的為對堆疊的操作可能會有寫入W(write)需求，而1跟2的編號在這邊分別表示來自於同一組雙指令封包內的第一個指令或是第2個指令。

	Translate Stage	Fetch Stage	Decode Stage		Execute Stage	
Now			R1	R2	W1	W2

	Translate Stage	Fetch Stage	Decode Stage		Execute Stage	
Next instruction package			R1	R2	W1	W2

圖10. 雙指令封包對堆疊記憶體存取指令在管線架構中的關係

Reg2Mem：用來觸發將暫存器上四個區域變數寫回BRAM的訊號線，外部仍需送寫入位址。

Mem2Reg：用來觸發從BRAM抓取四個區域變數寫到暫存器的訊號線，外部仍需送寫入位址。

(#U) = W (write) or R (read)

(#U)1_RegFlag：表示來自於目前周期雙指令中的第一個指令存取對象是暫存器或BRAM 。

(#U)2_RegFlag：表示來自於目前周期雙指令中的第二個指令存取對象是暫存器或BRAM。

(#U)1_en：表示來自於目前周期雙指令中的第一個指令存取是否有效。

(#U)2_en：表示來自於目前周期雙指令中的第二個指令存取是否有效。

(#U)1：表示來自於目前周期雙指令中的第一個指令存取的位址。

(#U)2：表示來自於目前周期雙指令中的第二個指令存取的位址。

(#U)D1：表示來自於目前周期雙指令中的第一個指令存取的資料。

(#U)D2：表示來自於目前周期雙指令中的第二個指令存取的資料。

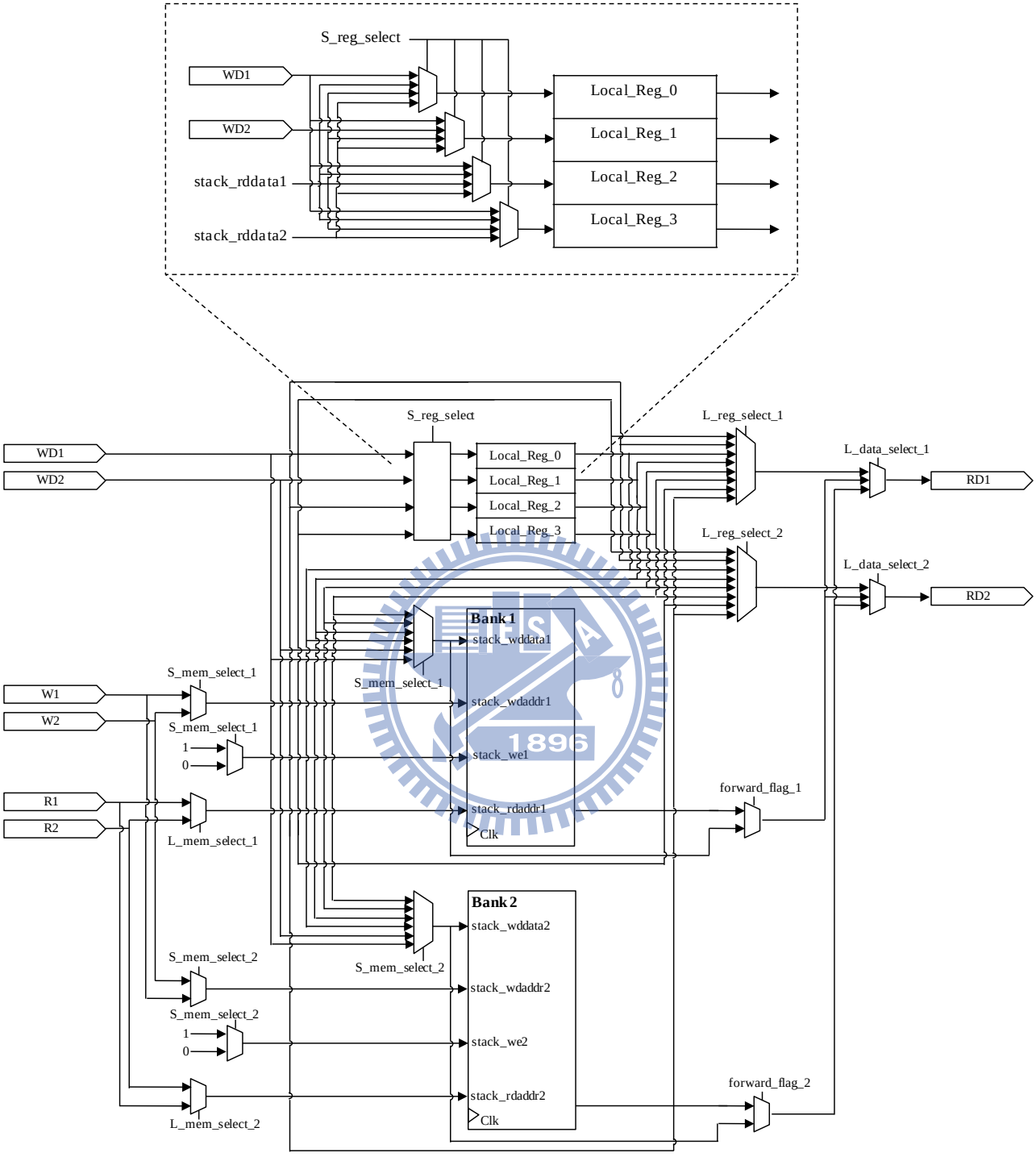

圖11. Four-Port Bank 電路架構圖

3.3. On-chip Memory 測試架構設計

為了對目前平台效能的評估，我們在實驗中也設計了使用不同記憶體配置來測試對整體效能的影響，包括參照資訊表（reference information table）所存放的空間以及 JAVA Heap 的空間都是放在 on-chip 記憶體上的(圖 12)。參照資訊表是當 JAIP 要存取 field data 或是 invoke method 時需要查找的資料，而 JAVA Heap 上的資料則包含了程式執行中的陣列、物件型態及靜 field data。這邊的設計是將 Local Bram Access Control Unit（包含存放上述兩塊資訊的記憶體空間）放在 Top Module 外面，其原因在於實際測試時，我們整合至 JAIP 中可以保留原有內部對外記憶體存取的控制訊號設計。

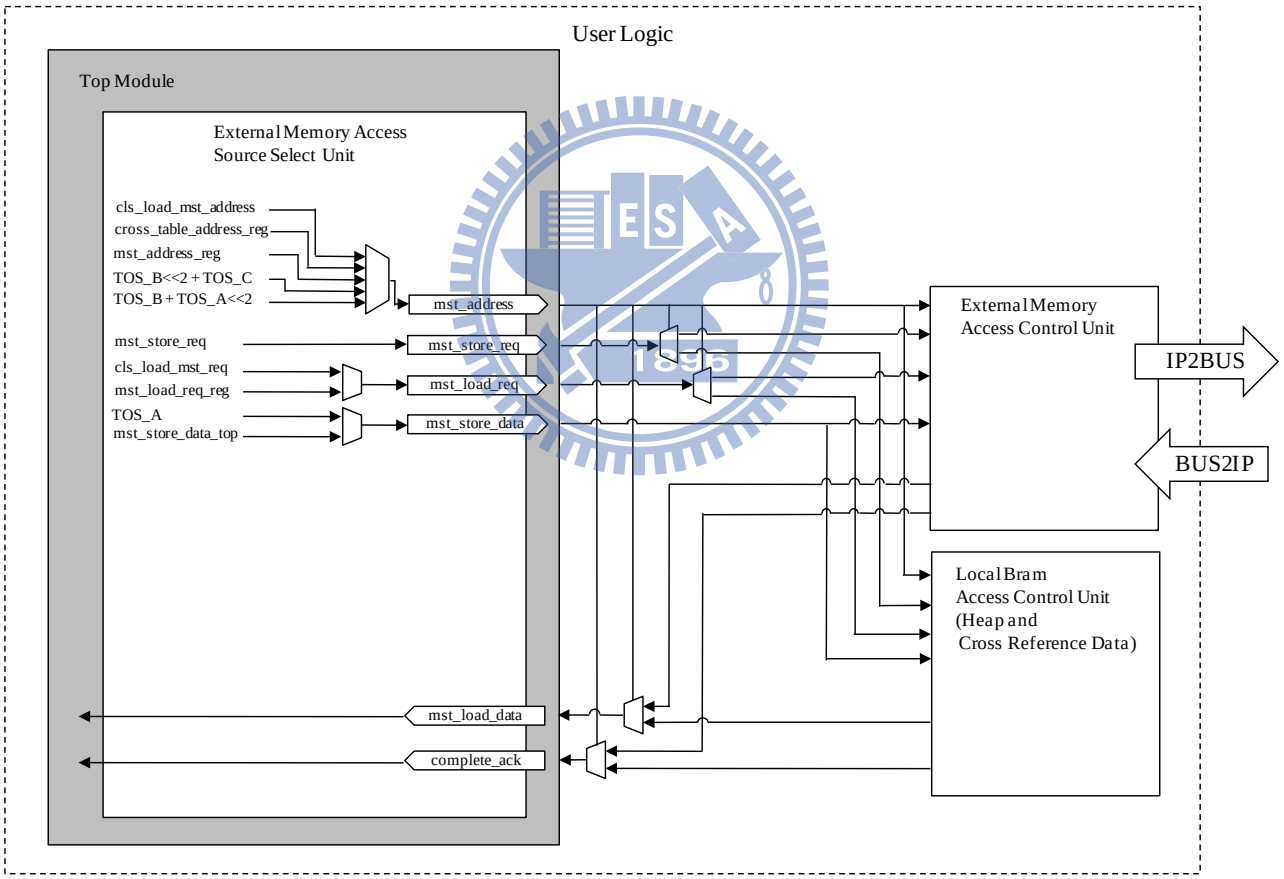

圖12. 使用 on-chip 記憶體架構

第四章　系統軟體介紹

4.1. 物件型別解析器（Class Parser）說明

關於物件型別解析器的功能在於當 JAIP 執行的過程中，對 JAIP 所需要執行的物件型別檔，產生適用於我們平台的映像檔（Run Time Image），以非主動的方式去對原始的物件型別檔做解析，並因為完成解析的物件型別其映像檔會在記憶體中保留一份，所以對物件型別作解析僅為一次性的動作，對於長時間在運作的系統也能接受這樣額外增加的負擔。

Cross reference table				
Class [x]	**Attributes**	Class ID , Class name, image address, etc		
	Parent_id	Class ID		
	Object Size	Size (byte)		
	IsParsed	1 or 0		
	Interface Info	IsInterface (1 or 0)	Interface Count	Interface List
	Field Data[i]	Field Name	Class ID & Field Offset	Field Tag
	Method[i]	Method Name	Class ID & Method Offset	
	Ldc[i]	Data	Type	
	String Pool	The whole string data stored in constant pool		
	String Pool Offset[i]	The offset for each string		

圖13. 在 Cross Reference Table 上每一個 Class 的資料結構

物件型別解析器會針對要解析的物件型別檔，先在 JAR 檔上做搜尋，我們的系統在啟動之後會先將 JAR 檔直接從 CF 卡上讀入記憶體中，以節省之後搜尋時的速度，最後根據是否存在於我們的 JAR 檔中，並判斷是否已經解析過，最後才去解析該物件型別檔及維護此物件型別檔在 XRT（Cross Reference Table）上的資料結構(圖 13)，這邊需要注意的部分是關於 XRT 資料結構上所提關於 Field Data 與 Method 的參照資料(Reference Data)，〔Class ID & Field Tag & Field Offset〕及〔Class ID & Method Offset〕這兩個欄位是用間接的方式存放在另外的

記憶體空間，這個資料結構欄位中放的只是指向另外記憶體空間的位址，會這樣做的原因在於硬體電路在做 Field Data 存取及 Invoke Method 都需要經常查找這個資訊，所以將存放這項資訊的記憶體會統一整理安排在 on-chip 的 BRAM 上用來加速查找的動作，在圖 13 上這兩個欄位先以簡化的方式表示其意義。

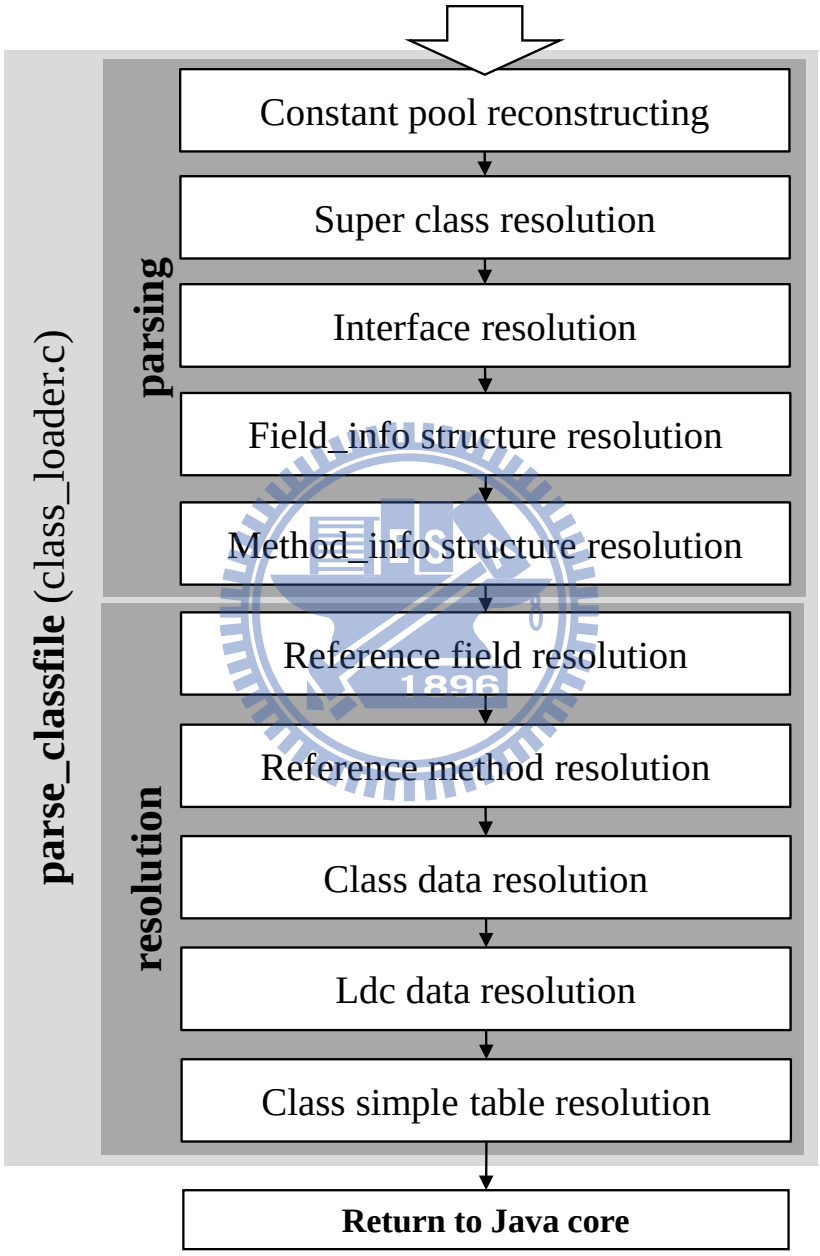

圖14. 物件型別解析器演算法流程

上圖 14 是我們物件型別解析器在先不考慮繼承以及介面機制的實作，只是針對單一物件

型別去做解析的流程，這邊將流程中的行為主要區分為 parsing 及 resolution 兩大類，其 parsing 所表示的意思是其行為發生在解析物件型別格式（class format）內容時會做的，而 resolution 則是指對常數索引區段中部分會使用到的資訊，做進一步的轉換，接著我們針對流程中每一個步驟去做說明。

4.1.1. 常數索引區段（Constant pool）的結構修改

物件型別解析器主要提供的功能其中一項是對 JAVA 物件型別檔中的常數索引區段（Constant Pool）進行結構修改，以簡化電路查找特定參考值的流程，原始的常數索引區段使用不同的標籤去表示每一個欄位的資料意義，並在每一筆資料內可以藉由常數索引區段的索引值再去找到其他欄位的資料來定義自己，如此反覆的查找最後才能知道完整的資訊，這邊 圖 15 是使用常數索引區段的索引值定義物件型別名稱的例子，而這樣在常數索引區段查找的動作（Resolution）會在執行解析的過程中頻繁出現，並且在不同標籤所代表的欄位資料在常數索引區段中的長度上也不是統一的，所以在查找的過程中需要逐一掃過小於索引值的所有欄位，當我們今天執行在執行的 JAVA bytecode 跟隨 Operand 是指向的是常數索引區段中的索引值時，這樣查找的動作更會影響我們在執行上的效能，所以在解析的過程中，我們這邊針對常數索引區段的處理會在第一次掃過所有常數索引區段時建立索引記錄每個欄位的起始位址，加快在後續解析物件型別時查找的動作，並將需要後續處理的一併資料紀錄起來。

index	tag	content
:	:	:
03	07 (class tag)	0004 (name index)
04	01 (utf8 tag)	0005 (length) 5349455645 (SIEVE)
:	:	:

圖15. 常數索引區段上一個以 utf8 字串定義物件型別名稱的範例

4.1.2. Super class resolution 及 Interface resolution

在 JAVA 語言中，其物件導向的特性具有繼承及介面功能（如 1.2 說明），而在物件型別的結構中（Class Format），裡面有兩個項目分別描述這個物件型別繼承的父物件型別為何以及實作的介面有哪些，這邊解析的動作會透過查找（Resolution）的方式建立此物件型別的父物件型別及有實作的介面資訊，並更新至 XRT 資料結構中。

4.1.3. Field_info/Method_info structure resolution

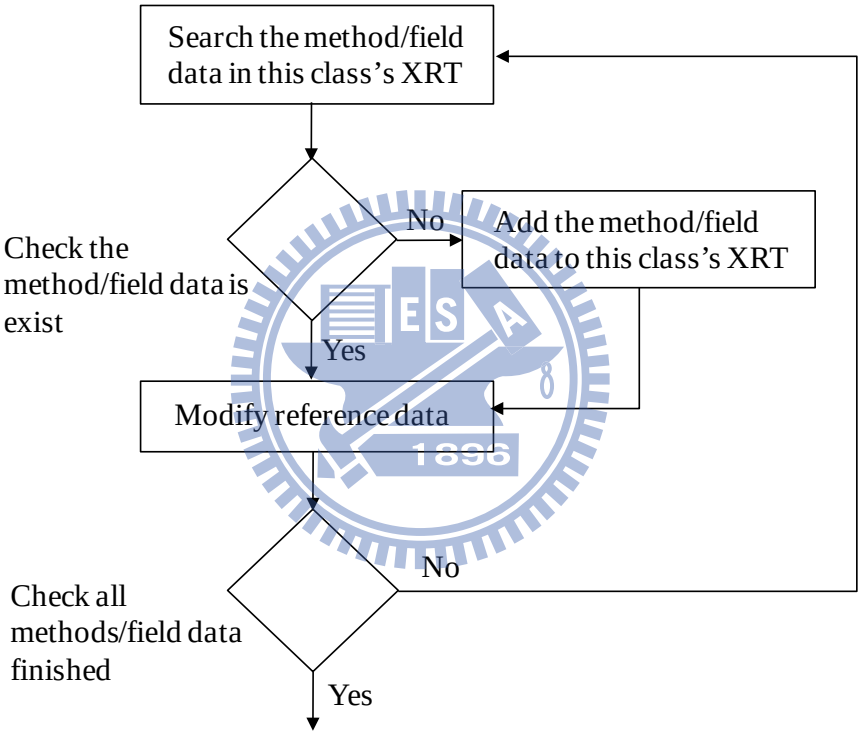

圖16. Field_info/Method_info structure resolution 演算法流程

上圖 16 是如何把某一個物件型別的 Field_info 及 Method_info 加入 XRT 的演算法，其中解析的欄位表示屬於目前解析的物件型別中所屬的 Field Data 或是 Method。因為我們使用的是 lazy 的方式去決定解析物件型別的時機，所以當別人有參照使用到此物件型別的 Field Data 或 Method 時，會先幫在此物件型別建立 Field_info 及 Method_info 的欄位並填入部分資訊，等到真正解析此物件型別時才會將正確的參照資料（Reference Data）

填入，所以當我們建立 Field_info 及 Method_info 時也要確認其參照資料是否已建立，這邊參照資料對於 Field Data 其 32bit 的內容代表的意義為所屬的物件型別、屬性及其相對於產生物件後，相對物件在記憶體中起位置的位移量，對於 Method 則是要表是其所屬的物件型別及執行的位元組碼在執行映像檔（Run Time Image）上的位置，不過在 Method 的參照資料處理上在建立的是物件型別跟介面會有些微不同的處理，詳細不同在 4.3 節關於介面部份會有更進一步的說明，演算法流程如圖 16。

4.1.4. Reference field/method resolution

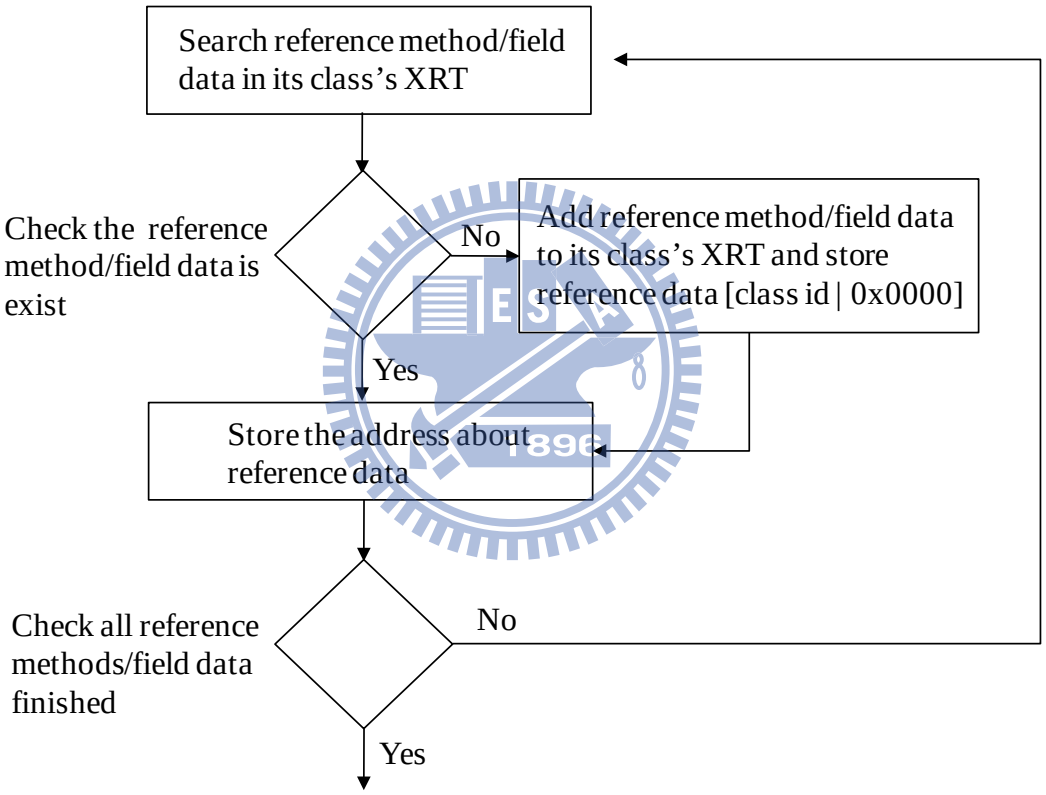

圖17. Reference field/method resolution 演算法流程

此資訊是列在常數索引區段中的 Field tag 及 Method tag 中，其所表示的意義為此物件型別中的方法會呼叫使用到的 Field Data 及 Method，可能屬於此物件型別，也可能是屬於其他物件型別，這部分的處裡主要是去搜尋確認 Reference field 及 Reference method，確認其所屬的物件型別以及該 field/method 的欄位是否皆已建立，若無則將此物件型別、欄位在 XRT 中

建立，並在參照資料（reference data）內填入〔Class ID | 0x0000〕，最後再將指向參照資料的位址做紀錄，之後一併整理至 Class Simple Table 中，4.1.7 會為 Class Simple Table 的功能做解釋。

4.1.5. Class data resolution

此資訊是列在常數索引區段中的 class tag 中，這部分的資訊透過查找後也會一併整理成 class info 至 Class simple table，class info 如圖 18、表 1，之所以會有維度資訊是因為在常數索引區段中關於陣列資訊也會以 class tag 表示，但最後這 class info 資訊並不會存放在 XRT 資料結構中，因為這資訊僅會在該物件型別的方法中像是 newobj、anewarray...等等的 bytecode 使用到。

Definition	type	dimension	Class ID
Space (bit)	8	8	16

圖 18. Class info 的資料表示

表 1 Class info 中的 Type 列舉說明

Primitive type	expressions
Object	0
Boolean	4
Char	5
Float	6
Double	7
byte	8
Short	9
Integer	10
Long	11

4.1.6. Ldc data resolution

此資訊是取出列在常數索引區段中的 Integer、Float 及 String 的 tag 中所記載的資訊，一樣會透過查找的方式將這些資訊存到 XRT 的資料結構中，為了之後當 JAVA 程式執行到 Ldc 的指令時，可以透過發出中斷觸發 Ldc 指令對應的 ISR，再將資料傳回 JAVA 處理器的堆疊上，以完成對 Ldc bytecode 的執行，所以需要在解析的階段，先將這些資訊整理至 XRT 的 Ldc data 中。

4.1.7. Run time image 格式介紹

最後要介紹的是完成解析的物件型別檔所產出的 run time image 結構，對 run time image 我們主要分為兩個部分，上半部稱為 class simple table，在 class simple table 中我們可以再細分切為三塊，最前面是我們自訂的 magic number 佔 4 個 byte，接著會使用每 2 個 byte 為一個欄位的方式，使用與常數索引區段欄位相同的個數，目的在於當 bytecode 使用的 Operand 指的是常數索引區段上的索引值時，可以直接查找到其對應的資料，這邊內容存放的是一個索引值，它指向位在第三塊上的資料，而在第三塊上的資料分為兩種，一種是物件型別方法所會參照到的 method 及 field data，在這邊會以 32bit 來表示其參照資料（reference data）所在記憶體的位置，第二種是直接存放 4.1.5 所提到的 class info，下半部則是將物件型別中所有的方法直接貼入，這邊在貼入時每個方法僅會保留 max_stack、max_local 及 bytecode 的資訊，最後產生的格式如圖 19 所示。

4D4D 4553 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0044 0000
0000 0000 0000 0000 0000 0000 003C 0000
0040 0000 0000 0000 0000 0000 0063 18D4
0063 18D8 0063 18C8 0001 0001 0001 0001
2AB7 000C B100 0001 0003 0002 0003 2A1B
B500 142A 1CBC 0AB5 0016 B100

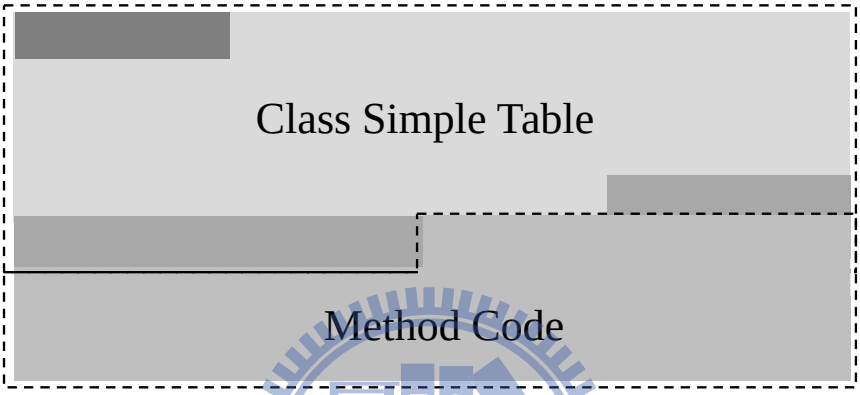

圖19. 實際 run time image 格式範例

4.2. 繼承（Inheritance）機制說明

在物件導向（Object-Oriented）的程式語言，繼承是相當重要的一個特性，在談實作繼承機制前我們先對繼承的行為做說明，所謂繼承，是指物件型別物件的資源可以延伸和重複使用，像是可以透過子物件型別或是子物件型別產生的物件呼叫繼承至父物件型別實作的方法、存取父物件型別宣告的 field data，在使用上，我們在程式中可以利用 extends 關鍵字來表示繼承關係，而 JAVA 的語法只允許單一繼承(Single Inheritance)，也就是說子物件型別同時只能繼承一個父物件型別，這邊需注意的一點在 JAVA 語言中雖然只允許單一繼承，但是當我們卻可以使用 extend 讓介面去繼承多個介面時，但是在物件型別格式（Class format）上形式會將繼承的介面資訊放在表示這個物件型別有實作的介面資訊上，而非放在表示這個物件型別的父物件型別資訊上，實作介面的資訊如何解讀這部份我們放在後面章節討論。

為了實現上述特性，我們在物件型別解析的流程設計上，需要在解析每個物件型別檔時

先完成其父物件型別的解析，以建立完整的子物件型別映像檔，這邊是以遞迴的方式去解析物件型別檔，所以愈上層的物件型別會愈先被完成解析，其遞迴演算法的流程如圖 20 所示。

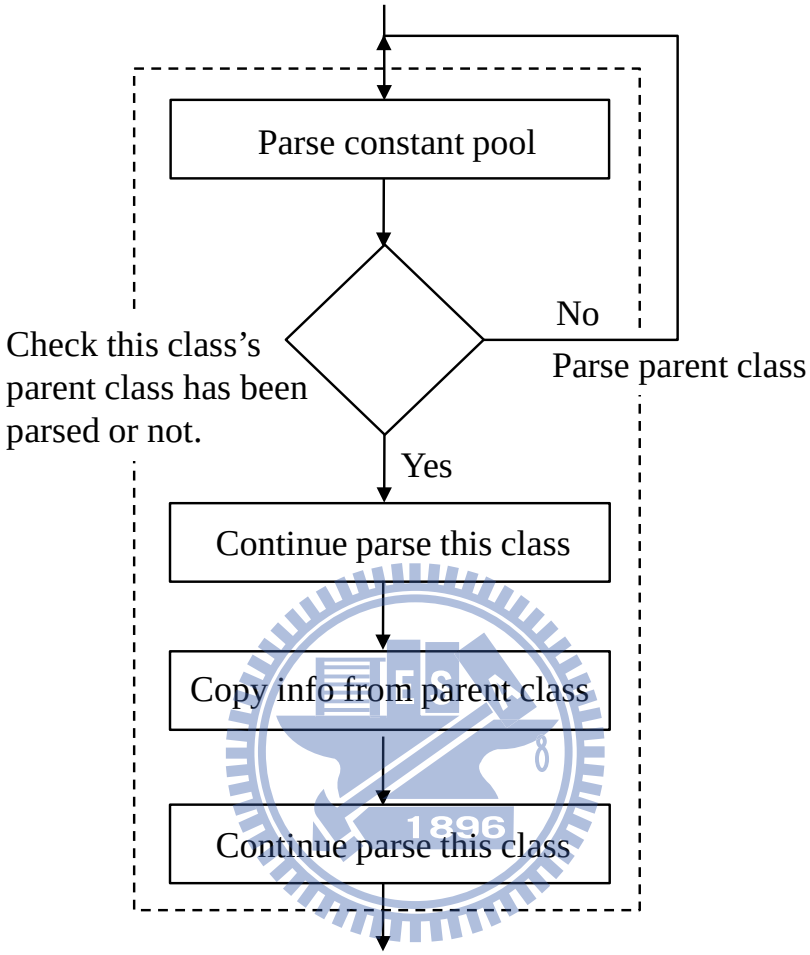

圖20. 在物件型別解析的流程中實作繼承的機制

除了解析父物件型別外，我們還需要從父物件型別的資料結構上複製部分資訊到目前解析物件型別的資料結構中，這邊包含了物件大小、介面資訊（interface info）、Field Data 以及 Merhod 的參照資料（Reference Data）。

透過複製父物件型別的物件大小的動作來繼續累加上子物件型別所屬的 Field Data，才能在產生新物件時正常配置記憶體空間，其配置記憶體的空間使用如圖 21 所示。

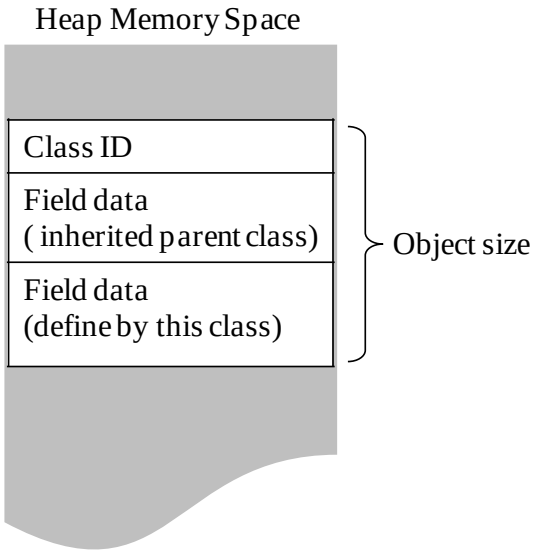

圖21. 產生新物件時在記憶體中配置的情形

介面資訊(interface info)主要複製內容包含實作的介面數量以及被實作介面的Class ID，讓呼叫介面方法時透過父物件型別實作的介面，也可以找到子物件型別有繼承或覆寫的方法，這邊關於介面的應用詳細內容以及複製介面資訊的原由我們保留在下一個章節 4.3 來討論。

另外這邊在說明複製 Field Data 資訊前，我們先針對 Field Data 的參照資料（Reference Data）上的資料結構再做詳細的描述，在 32bit 的 Field Data 參照資料中我們使用前 16bit 表示 Class ID 表示 field data 所屬的物件型別，當繼承至父物件型別時這部分會替換成子物件型別的 Class ID，而後半段的 16bit 拆成 4bit 表示 Field Tag，Field Tag 分別表示其是否為靜態的 Field Data、基本型態、長整數型態以及當 Field Data 存放的是記憶體中的位址時，所表示的是陣列或是物件的起始位置，另外在 XRT 上我們還新增了 32bit 的 Static Field Address 欄位用以存放當今天是靜態 Field Data 時，所有物件應該都會參照到的相同記憶體空間位址，以上結構的說明，(如圖 22、表 2 列舉 Field Tag 所表示的意義所示)，要特別注意的是除了長整數的 Field Data 在物件上是佔 64bit 的空間，其他型態的資料都是以 32bit 的空間來存放，在位移量也除了長整數型態是加二表示佔兩個 words 的空間，其餘每增加一個資料量為加一，最後我們以實際例子來描述資料結構及其對應到的記憶體配置關係，(如圖 23 所示)。

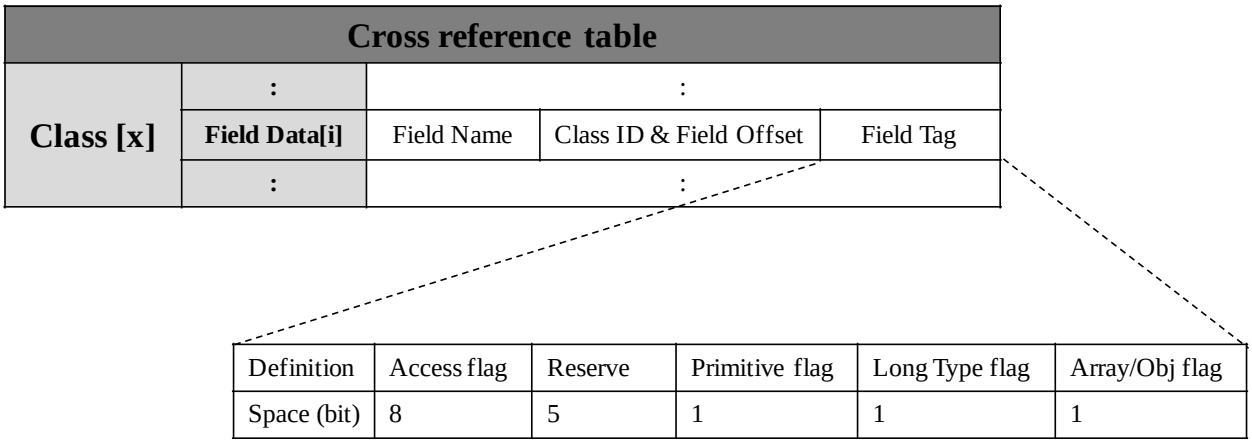

圖22. 關於物件型別所定義的資料在資料結構中的表式

這邊對於 Field Tag 的意義除了在幫助我們分辨是否為靜態 field data 外，在後面章節所提到的 Native Method 實作中也會需要參考其他 flag 資訊。

表 2 列舉 Field Tag 所表示的意義

Field Tag		
Long	Access Flag	00000 | 110
Int		00000 | 100
Short		00000 | 100
Char		00000 | 100
Array		00000 | 001
Object		00000 | 000

所以在實作繼承機制時，子物件型別透過先複製父物件型別的靜態與非靜態 Field Data 部分資訊，來得到繼承的 Field Data 在子物件型別產生物件時，所配置空間相對於物件起始位址的位移量以及其所屬的 Field Tag，最後再加上解析子物件型別後得到自行宣告的 Field Data，才能產生完整的 Field Data 資訊。

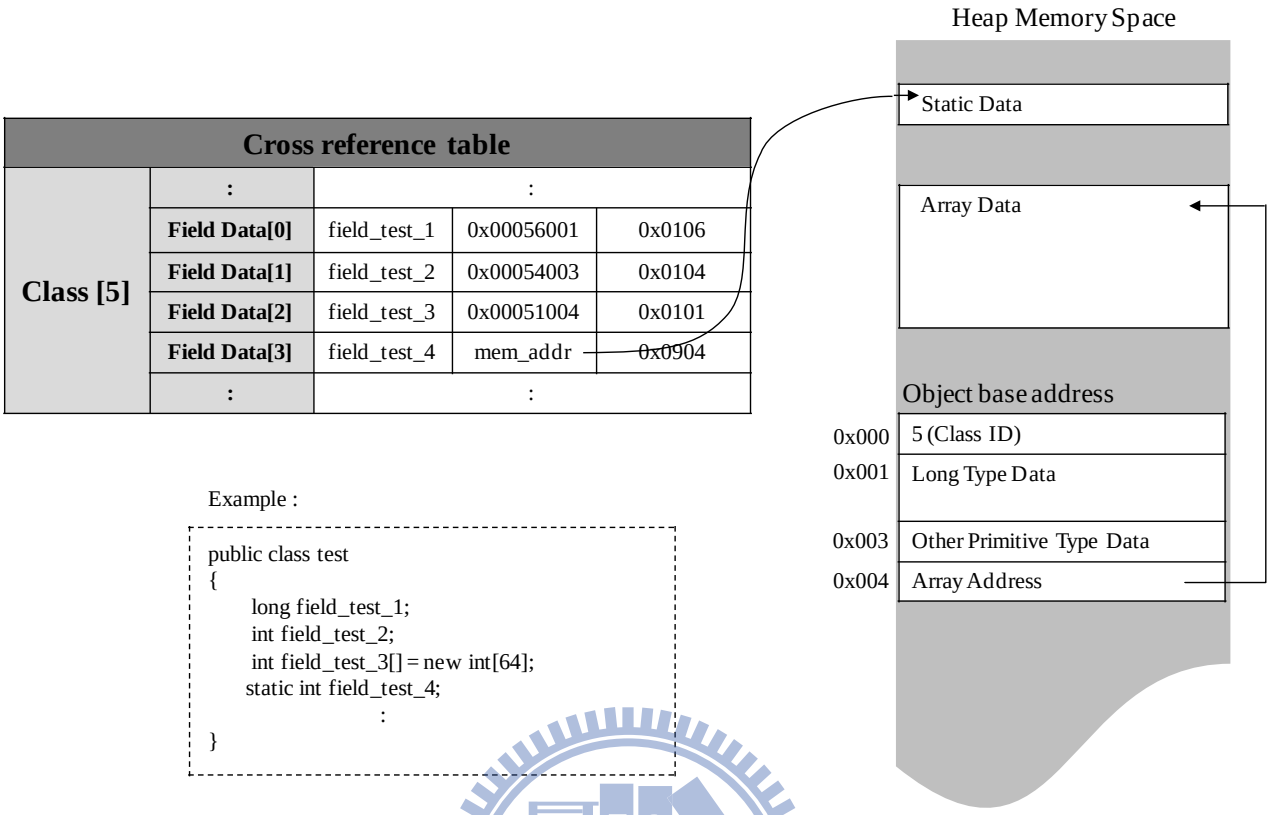

圖23. 舉例描述物件型別資料結構中 Field Data 對應物件在 Heap Memory 上的關係

最後實作繼承至父物件型別的方法，僅需在子物件型別複製一份完整父物件型別方法的資訊，包含方法的名稱、描述以及父物件型別方法完整的參照資訊〔父物件型別 Class ID | Method Offset〕，這邊之所以不用替換 Class ID，是因為在我們設計的機制中當 invoke method 時，執行的位元組碼（bytecode）仍是使用位在父物件型別的執行映像檔（run time image）上某個方法中。

4.3. 介面（Interface）機制說明

在前面已經提過繼承是物件導向語言中的重要特性之後，這邊還要介紹另一項特性－介面，而在說明實作部分之前，我們這邊一樣也先對介面的功能以及意義做說明，最後再接著描述這些機制在我們平台上實作的方式。

介面的宣告像是在定義一個東西應有的行為，所以介面本身只有行為名稱，卻沒有描述行為的詳細動作，若是某個物件型別實做了一個介面，則該物件型別必須把介面的所有方法都實作出來，這麼的規範也表示任何實做了某介面的物件型別，我們可確信其必具有介面應有的方法，就算方法內容的動作大不相同，我們仍然可以呼叫，而在使用上，因為在 JAVA 語言本身並不提供多重繼承的功能,所以一般也會透過介面的使用上來達到多重繼承的效果,因為 JAVA 支援一次實作多個介面，就像是同時繼承多個抽象物件型別（實際上這是 C++中多重繼承的一個實際運用方式），以下（圖 24）是一個介面的例子，我定義了一個 Interface 叫 phone，他必具備一個叫 dial 的方法，但是在介面裡並不會詳述要怎麼動作，而右邊是實做了這個介面的物件型別，在這物件型別裡面就必須把介面裡的方法完整的描述出來。

Interface: phone

```
public interface phone {
    public void dial();
}
```

Class: nokia

```
public class nokia implements phone{
    public void dial(){
        …
    };
}
```

圖24. 介面與實作其方法的物件型別例子

實作介面的物件型別在操作上並沒有與其他未實作介面的物件型別有所不同，而是多了一種方式呼叫該物件型別的方法，即是我們可以使用透過呼叫介面方法（Invokeinterface）來使用實作其的物件型別方法，這樣的方式在呼叫實作相同介面的不同物件型別方法，指令上會相同,而且在我們目前機制的資料結構(XRT)中查找到的其方法參考資訊(reference data)也會是一樣的，因為都是指向建立在屬於介面的資料結構裡，並非像是一般呼叫物件型別方

法（invokespecial、invokevirtual、invokestatic）會直接連結到屬於該物件型別方法的參考資訊（reference data）。

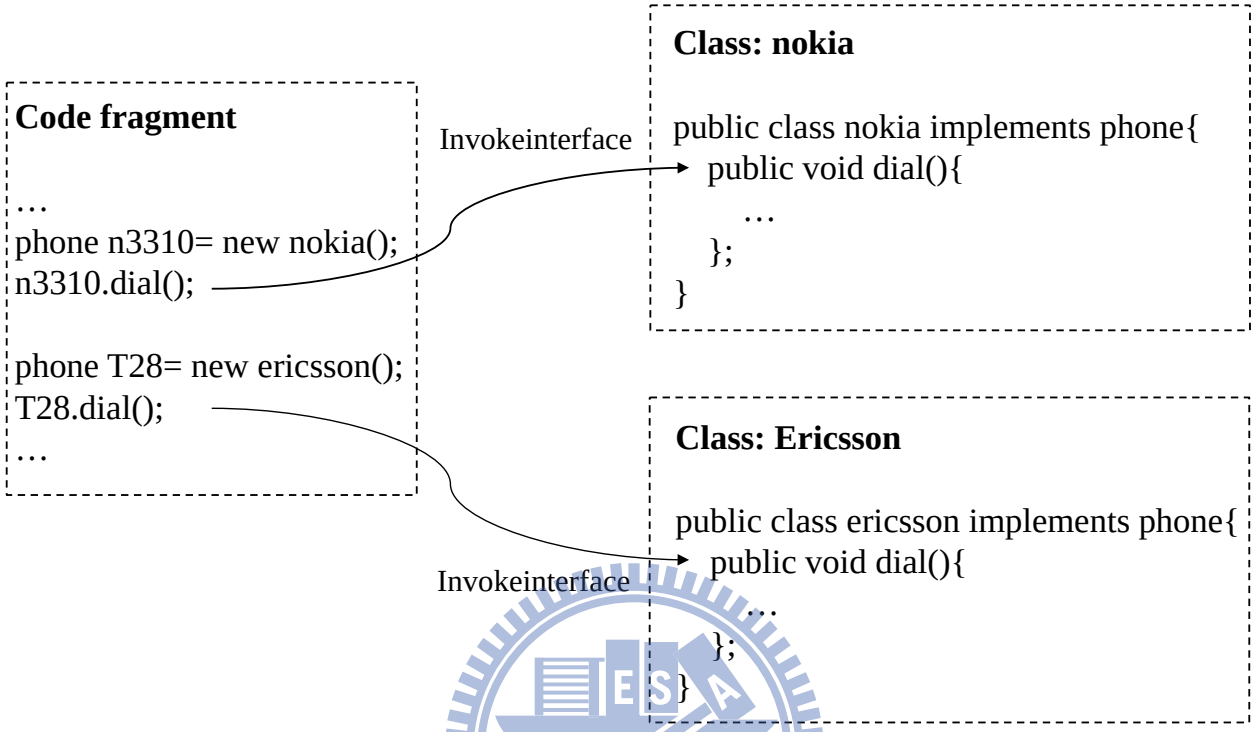

圖25. 透過呼叫介面方法的方式呼叫實作其的物件型別方法

上圖 25 是一個實際呼叫介面方法的例子，左半邊是一個程式片段，接續前段的例子，這邊另外又多加了個物件型別，也同樣實做了 phone 這個介面，這邊 n3310 與 T28 是兩個具有相同介面屬性的物件，而他們真正的身分是個別是 nokia 與 ericsson，接下來呼叫使用該介面裡定義的行為 dial，所以就透過了 Invokeinterface dial 來呼叫，然而因為 n3310 跟 T28 是不同身分的物件，所以在 Java 雖然使用同樣的 byte code，最後卻會呼叫到的 dial 其 code 會不同。

如何透過指向相同的參照資料（reference data）找到不同的方法，在這邊我們對資料結構提出新的修改，建立介面的資料結構時，會將 XRT 上指向一般物件型別方法的參照資料改成指向鏈結串列的資料結構，而這串列上每個節點有三個資訊，包含此方法所屬的 Class ID、方法的參照資料、下一個節點的位置，注意這邊之所以除了原來方法的參照資料還增加所屬的 Class ID，目的在於解決子物件型別繼承至父物件型別方法時，參照資料上的 Class ID 會

是來自於父物件型別，而在語法中我們可以透過父物件型別實作的介面來呼叫子物件型別的方法，這邊子物件型別的方法定義包含子物件型別有覆寫（over write）的方法或是僅繼承至父物件型別，當子物件型別未覆寫其繼承的方法時，若不增加此欄位的話，我們當 Invokeinterface 時會在 method list 上找不到屬於子物件型別 Class ID 的方法，所以之後我們在解析一個物件型別時，會將它每個實作的方法加到其有實作介面的 method list 上面，新的資料結構如圖 26 所示。

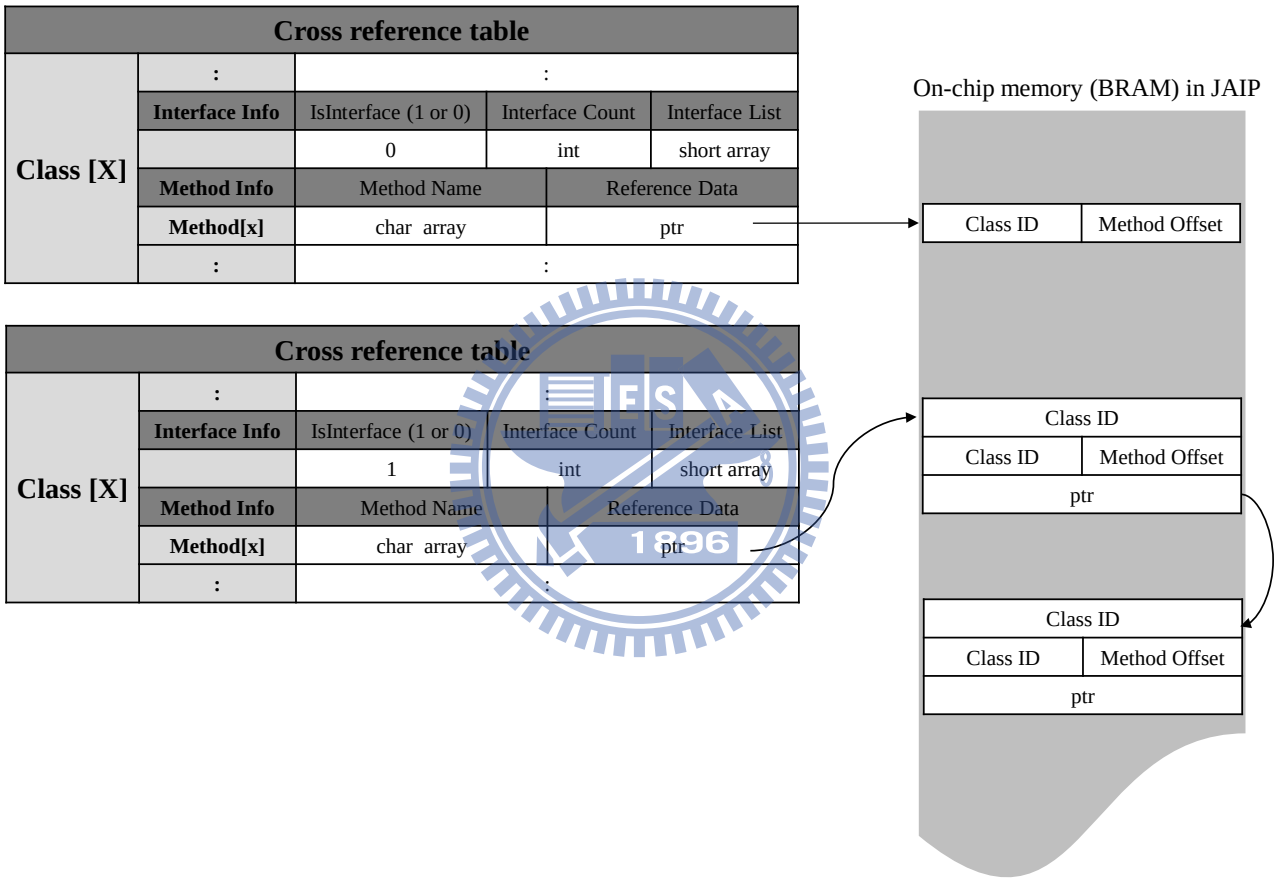

圖26. 一般物件型別與介面在方法參照資料上的差異

最後還要解決一個問題，這個問題在於我們要決定一個物件型別其所實作的介面有哪些，除了會有本身實作的介面外，還有其繼承的父物件型別所實作的介面，最後還有要加上本身實作的介面其繼承的其他介面，上述這三種介面都能透過呼叫介面方法的方式來使用到這個物件型別有實作其的方法，因為需要建立完整的介面資訊（Interface Info），所以在解析物件型別實的流程也會跟著調整，遞迴演算法的流程如圖 27 所示，透過這樣的流程可以保證我們

在解析子物件型別實作介面前，會先完成其父物件型別及其實作介面的解析，可以順利建立完整的介面資訊。

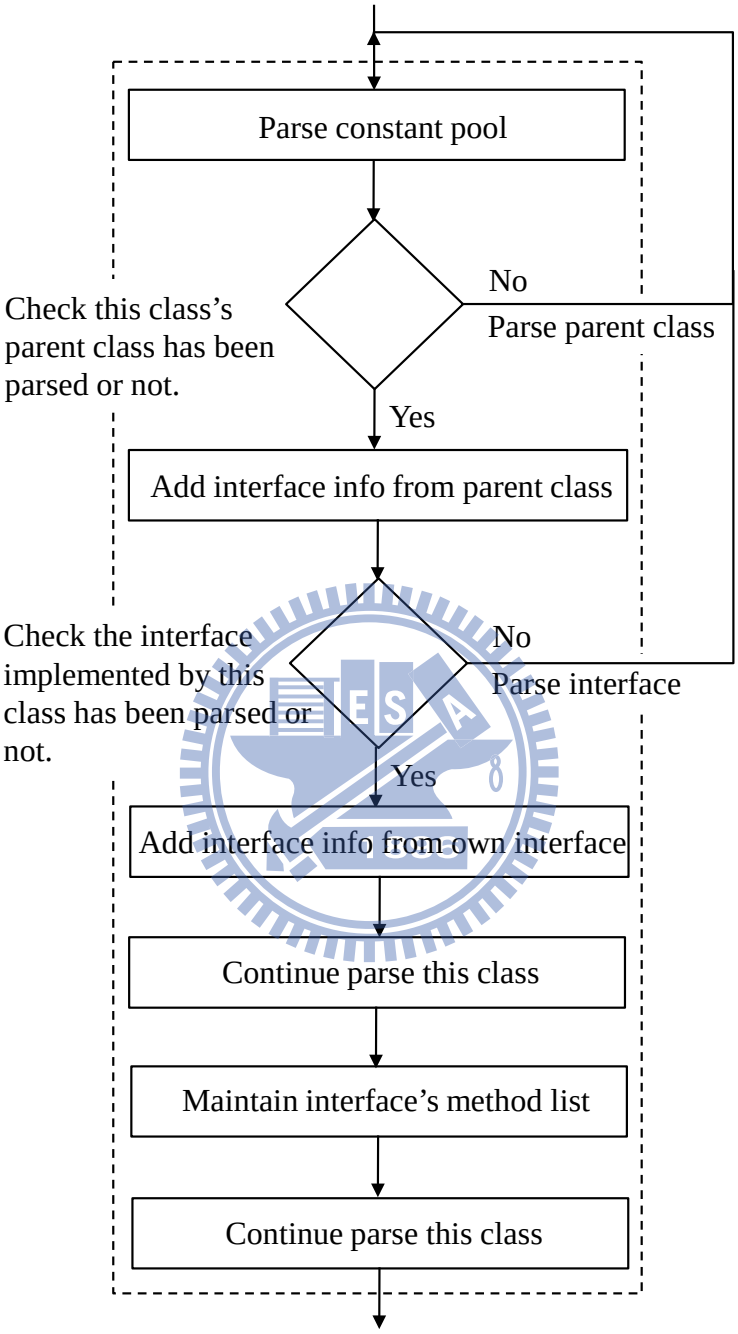

圖27. 在物件型別解析的流程中實作介面的機制

4.4. ISR 介紹

在我們系統所提供 ISR 的部分，這邊將它們分為兩類，其中一類是由 JAIP 執行電路的過程所啟動的，另一類則是由系統物件型別(system class)所提供的原生方法(native method)，在開始說明這些 ISR 的內容流程前，先對 JAIP 與 RISC core 溝通的介面做介紹，之後我們會再對其在執行 ISR 時的狀態變化做說明，最後再解釋各 ISR 實作的內容。

4.4.1. ISR 狀態變化圖示說明

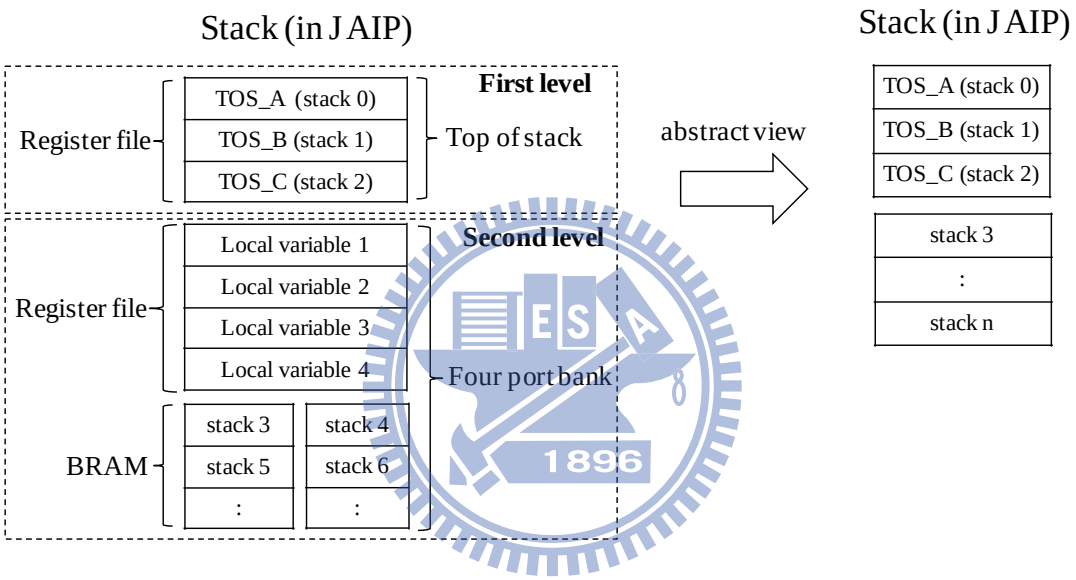

圖28. 堆疊抽象化的簡圖

ISR 所需傳遞的參數其中一部分資訊是透過 stack 中最上層的三個暫存器來傳遞，這邊會以抽象的圖示來表示 JAIP 所使用的 2-level 及交錯式的記憶體，之後再以抽象圖，(如圖 28 所示)，來表達執行 ISR 時的 stack 狀態變化。

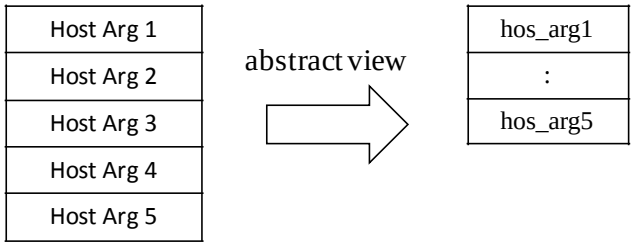

圖29. Host arg 抽象化的簡圖

JAIP 用來傳遞參數的 register file，這裡的參數指的不是出現在 operand 上的，而是另外用來幫助我們軟體執行的額外資訊，在實做原生方法的機制時也用來當呼叫方法所傳遞參數來使用，抽象圖如圖 29 所示。

Bytecode

Type	Bytecode
:	:
:	:

圖30. 用來表示目前執行的指令及其 operand 意義

這邊的圖 30 表示用以表示觸發 ISR 時，目前執行的 bytecode，以及之後跟隨著 operand 意義，而 type 中的 u(N)，N 指得是資訊佔多少 byte。

4.4.2. ISR 流程圖

首先在這邊說明 newarray 與 anewarray 的 ISR 內容時，先解釋 array 在我們平台實作上的資料結構，我們對 array 的表示會分為三個欄位，以圖 31 說明，array tag 依序表示的有陣列存放的是不是 primitive 的 tpye、每個元素佔的空間，這邊不同於 field data 的處理，在 field data 中除了長整數是佔 64bit，其餘皆是 32bit 的大小，而在陣列上的元素除了 boolean 是佔 8bit 以外，其他元素皆是配剛好跟資料所需空間相符的大小，這樣的設定也是符合 JAVA 在 SPEC 中的規範，接著的 flag 如同 field data 會表示元素中存放的是位址時，指向的是物件還是陣列的起始位址，下一個欄位是 length，用來支援 arraylength（0xBE）的 bytecode，之後接的才是實際陣列的資料，之後圖 32－圖 39 分別表示執行其 ISR 時系統的變化以及演算法的流程。

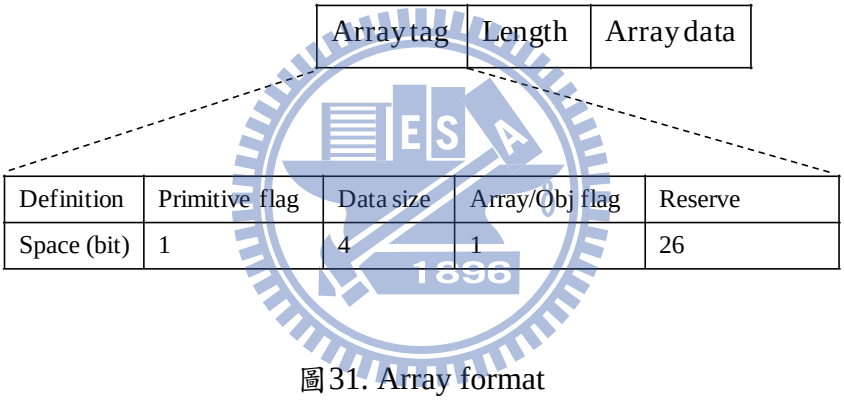

圖31. Array format

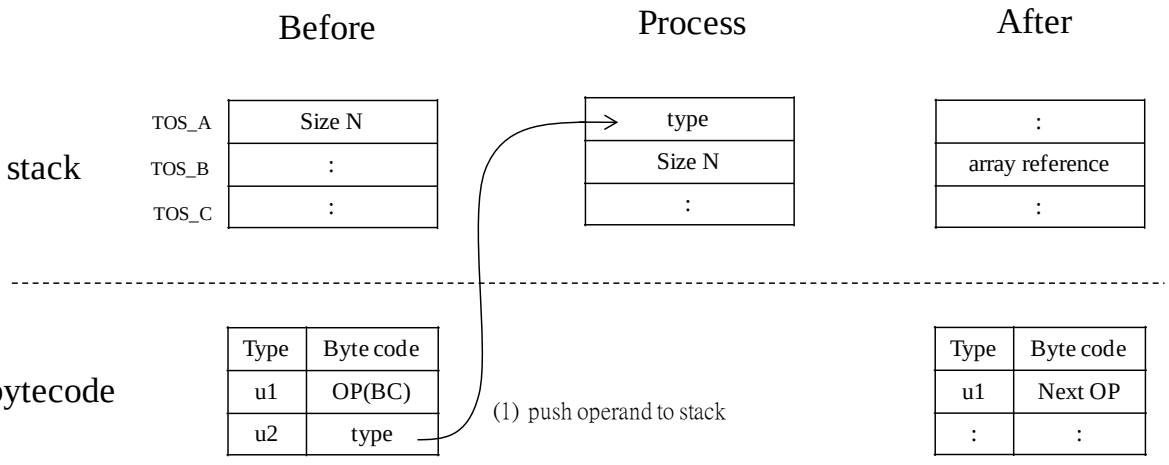

圖32. newarray ISR 執行時的狀態變化

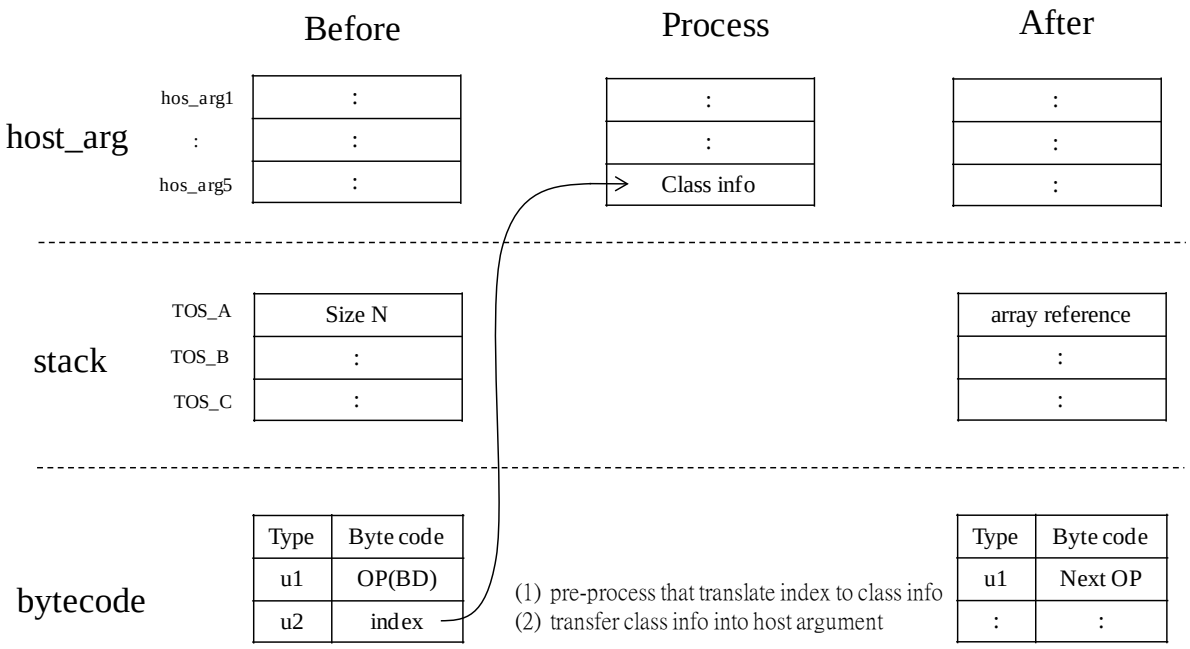

圖33. anewarray ISR 執行時的狀態變化

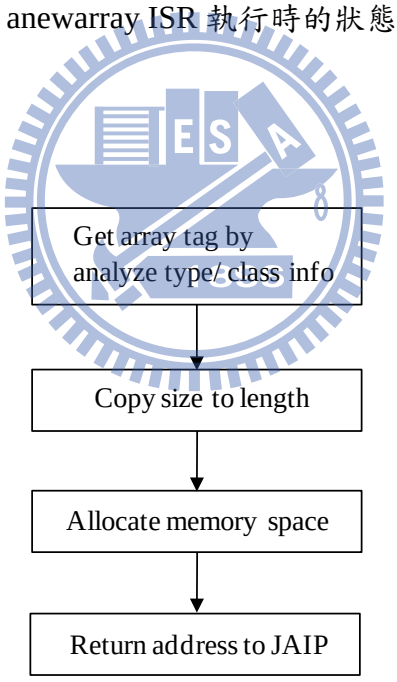

圖34. newarray/anewarray ISR 程式實作流程

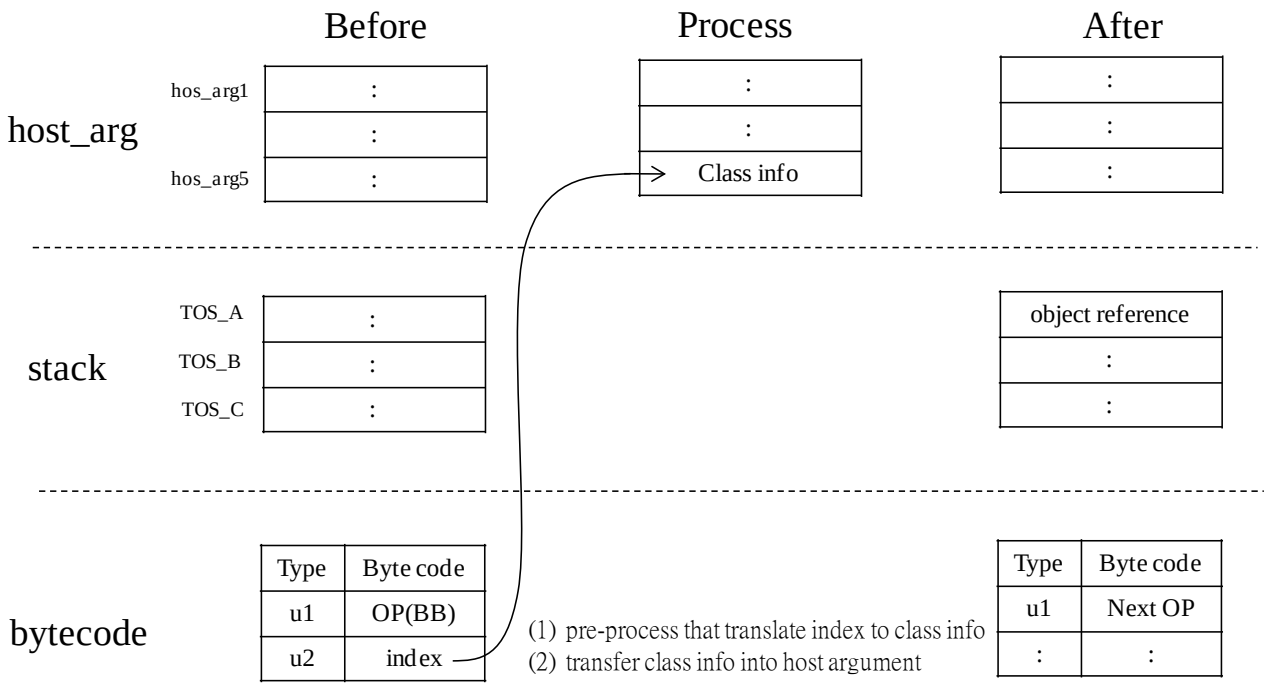

圖35. newobj ISR 執行時的狀態變化

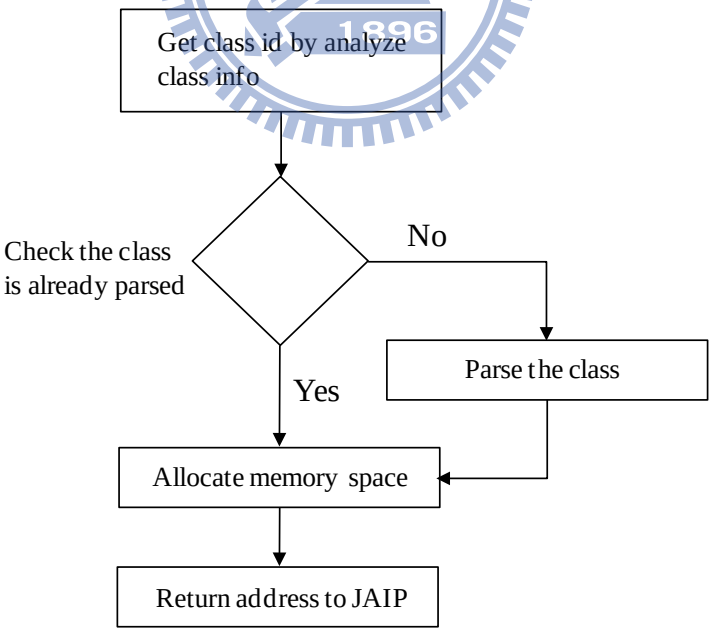

圖36. newobj ISR 程式實作流程

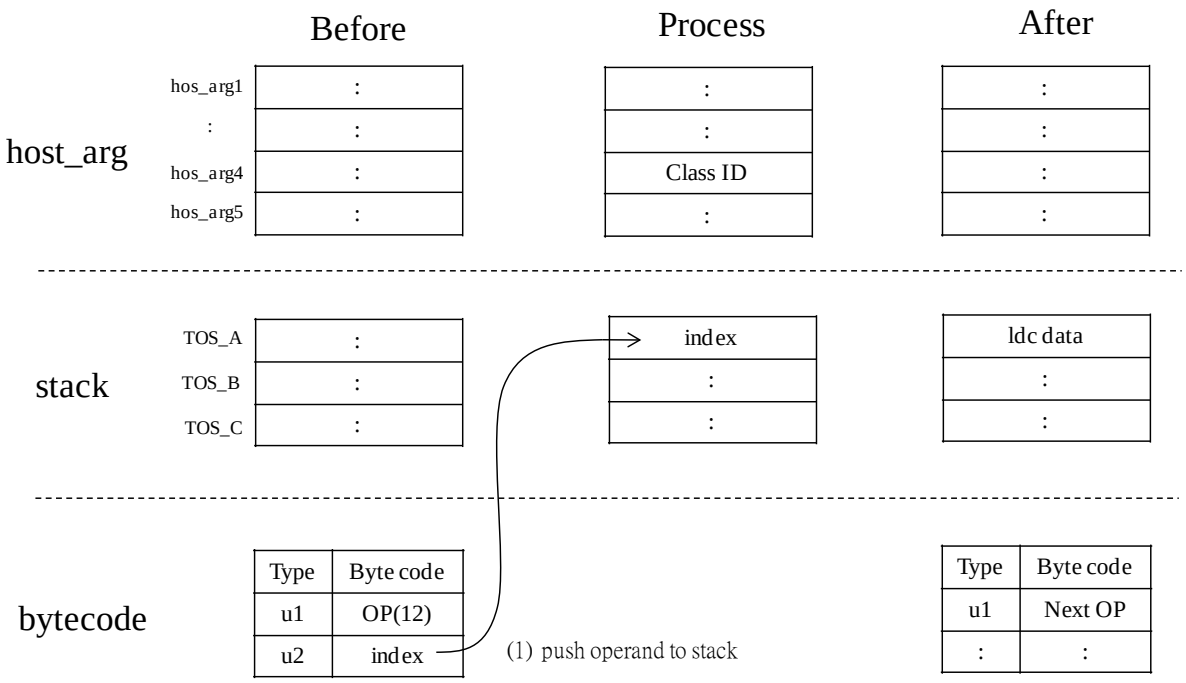

圖37. ldc ISR 執行時的狀態變化

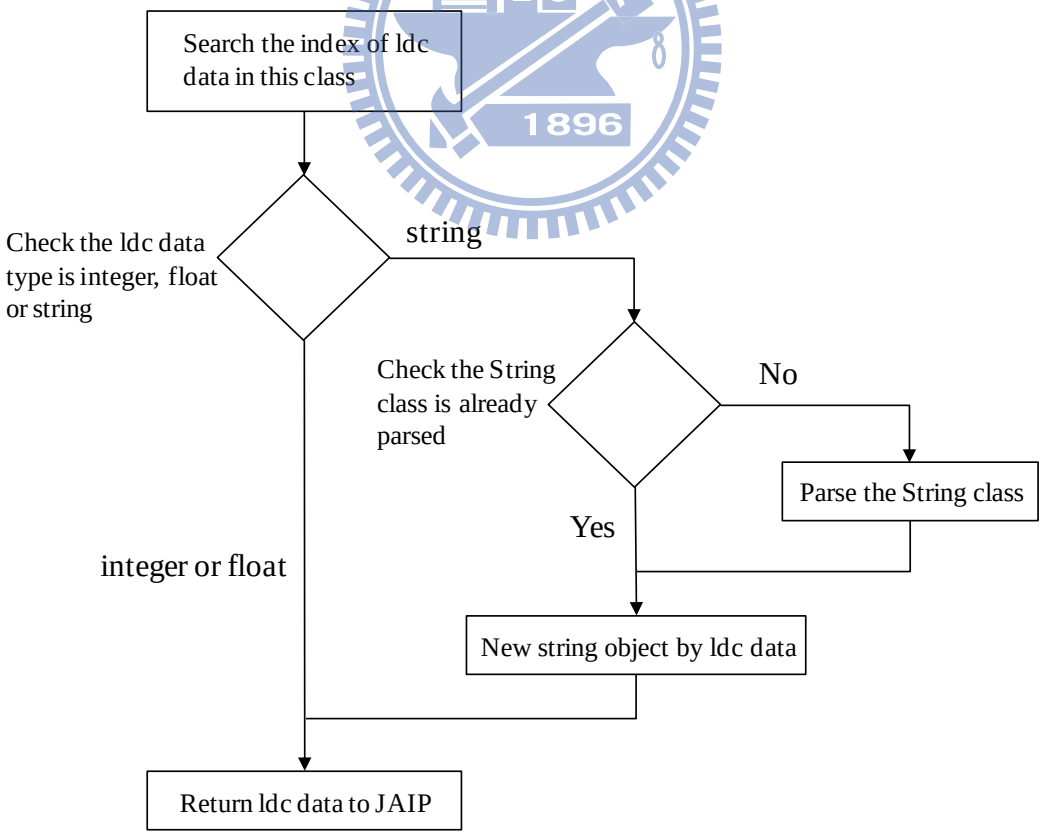

圖38. ldc ISR 程式實作流程

最後介紹的這個 ISR 稱作 bad offset，觸發此中斷的原因在於我們對物件型別作解析的動作為非主動式的，所以當 JAVA bytecode 做 field data 存取或是方法的呼叫，其參照資料為未被解析的物件型別，則會抓取到〔Class ID | 0x0000〕的參照資料，這邊需要透過對物件型別作解析才能抓取到正確的參照資料。

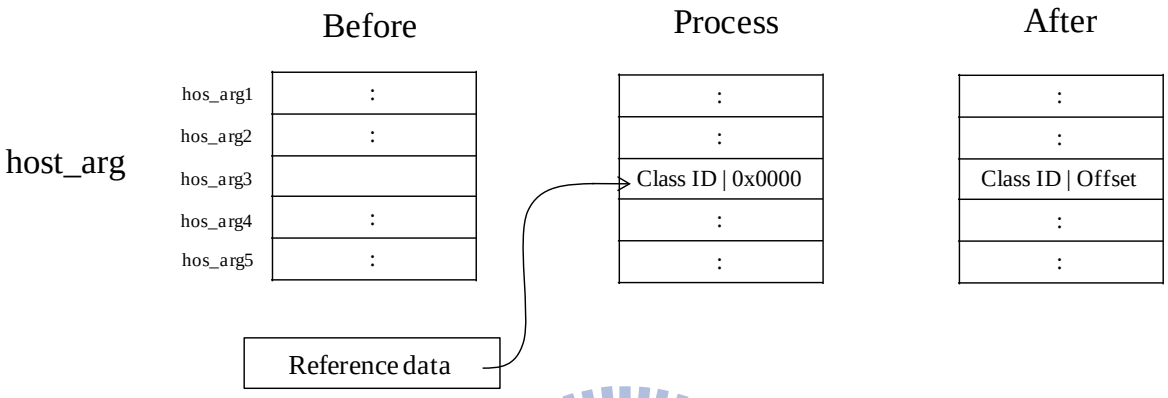

圖39. Bad index ISR 執行時的狀態變化

4.4.3. 原生方法的機制（Native method）說明

因為原生方法的呼叫在 bytecode 層級來看是與一般呼叫方法是相同的，透過 invokevirtual、invokestatic 指令後面接的常數索引區段索引值，一樣以間接的方式最後取到參照資料在記憶體中的位置，並抓取參照資料，而我們實作原生方法的機制是透過在物件型別解析時，檢察所屬其物件型別的方法是否為原生方法，如果是的話，我們則會在參照資料裡上放入屬於原生方法格式的資料，格式如圖 40，我們會犧牲 0xFF 開頭的 Class ID 用以表示此參照資料為原生方法，讓 JAIP 取回參考資料時可以進入到呼叫原生方法的狀態機。

Definition	Native method tag (0xFF)	Parameter count	Return value count	ISR ID
Space (bit)	8	8	8	8

圖40. 原生方法參照資料（reference data）的格式

我們在 ISR 的 ID 上會用前 16bit 以 0 或 1 區分是否會為 Native method 所觸發，當 JAIP 發出中斷時，我們會先確認其是否為 Native method 所觸發的，之後跳去執行我們建立的 native method table 上對應到的 function，機制如 0 所示，其執行 Native method 實的系統變化如 0 所示。

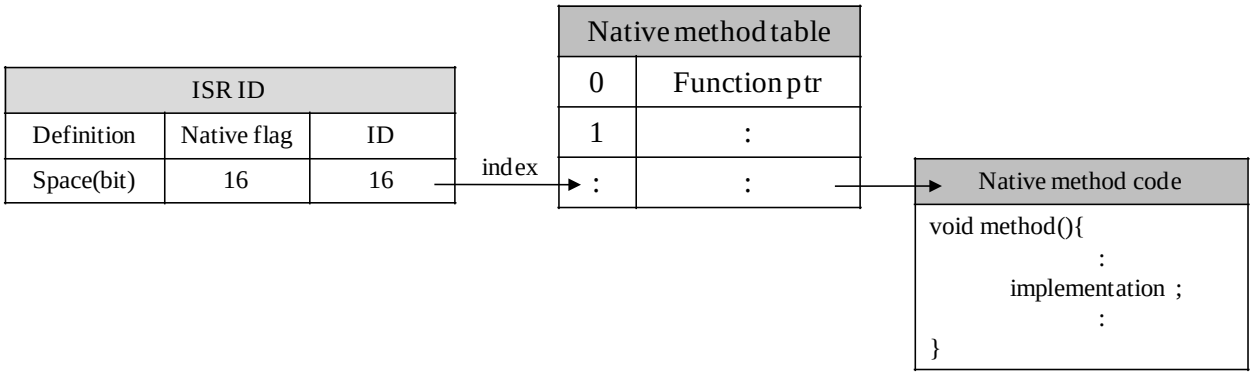

圖41. Native method 機制

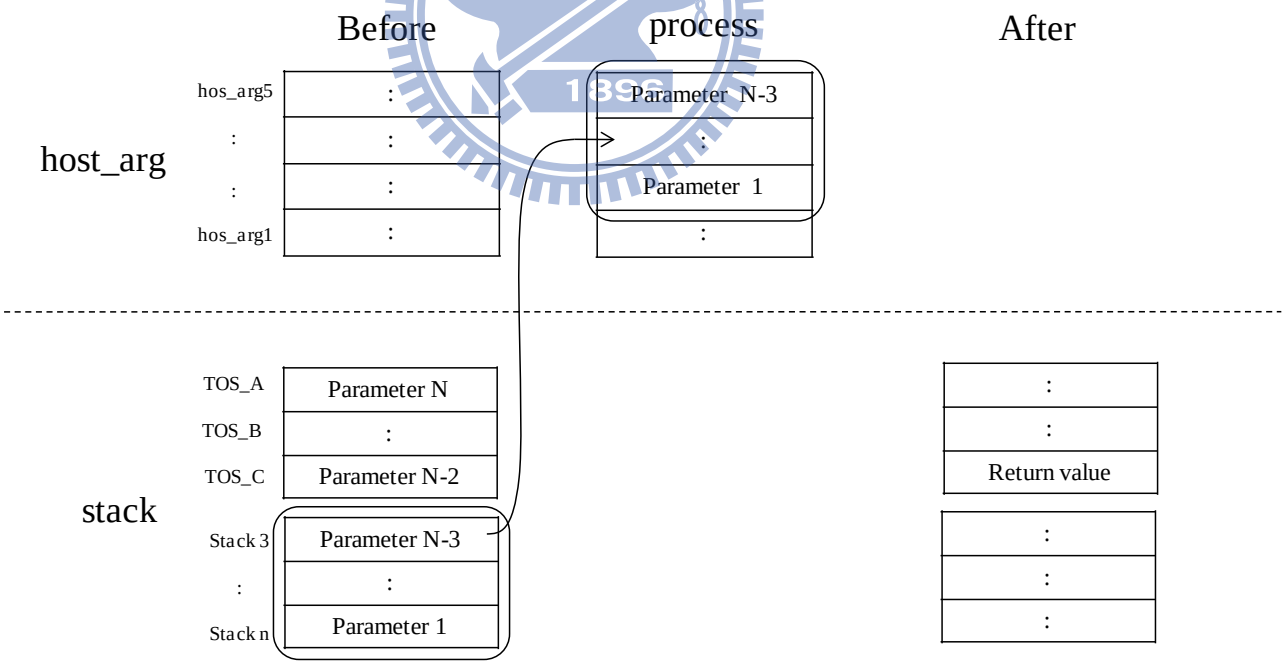

圖42. Invoke native method 執行時的狀態變化

而為了配合電路在 native method 的結束時對 stack 的整理操作方便，我們在傳遞不同個數的參數時，擺放在 register file 上的順序也不同，擺放順序如表 3 所示，而目前實作的原生方法如表 4 所列。

表 3 傳遞不同參數個數時，參數位在 register file 上的位置，最多支援 8 個參數

Parameter order								
Parameter count	1	2	3	4	5	6	7	8
TOS_A			③	④	⑤	⑥	⑦	⑧
TOS_B		②	②	③	④	⑤	⑥	⑦
TOS_C	①	①	①	②	③	④	⑤	⑥
Host_Arg 5								⑤
Host_Arg 4							④	④
Host_Arg 3						③	③	③
Host_Arg 2					②	②	②	②
Host_Arg 1				①	①	①	①	①

表 4 目前對 System Class 所支援的原生方法

Java_java_lang_Object_getClass
Java_java_lang_Class_forName
Java_java_lang_Class_newInstance
Java_java_lang_Thread_yield
Java_java_lang_System_arraycopy
Java_java_lang_System_currentTimeMillis
Java_java_lang_System_getProperty0
Java_java_lang_String_charAt
Java_java_lang_String_indexOf__I
Java_java_lang_String_indexOf__II
Java_java_lang_String_equals
Java_java_lang_StringBuffer_append__I
Java_java_lang_StringBuffer_append__Ljava_lang_String_2
Java_java_lang_StringBuffer_toString

實作原生方法的依據主要參考自 KVM 當中實作的原生方法的邏輯，並撰寫符合我們平台的版本，接下來會說明各原生方法的實作內容。

java/lang/Object/getClass

因為在我們配置一個新產生的物件時，其在記憶體中最前面 32bit 的資料就是存放此物件所屬的 Class ID，所以當我們今天取得 Object reference 時，我們即可以透過這個位址讀取 Class ID，並回傳給 JAIP。

java/lang/Class/forName

透過取得的 String Object 參數，我們取出其字串的 field data，並以此為要解析的物件型別名稱呼叫物件型別解析器，最後再回傳一個 Class Object 給 JAIP，但是其所屬的 Class ID 我們會填入此次解析物件型別的 Class ID，而非 Class 這個物件型別的 Class ID。

java/lang/Class/newInstance

透過傳遞出的由 Class 造出的 Object，我們取出其存放的 Class ID，再透過此 Class ID 去配置一個新物件記憶體空間，並回傳 object reference 給 JAIP。

java/lang/Thread/yield

目前平台尚未提供多執行緒的功能，但此方法會去確認目前系統中的執行緒個數，當個數大於一時,我們會去設定切換執行緒的訊號,之後可以延伸做為觸發電路做切換執行緒的機制。

java/lang/System/currentTimeMillis

這邊我是透過讀取 JAIP 內部的 timer 暫存器的值，再去對系統頻率換算得到 JAIP 啟動之後的毫秒數，再將這個時間資訊傳回給 JAIP。

java/lang/System/arraycopy

arraycopy 會使用到五個參數，其中兩個分別代表陣列的記憶體位址、另外兩個表示從陣列上要開始複製及貼上的起始位置，最後一個表示要複製資料的長度，在這部分實作兩個遞迴呼

叫的函式 copy_obj 及 copy_array 來實作，這兩個函式的流程如圖 43、圖 44 所示，當呼叫 arraycopy 時會根據 array tag 去決定呼叫 copy_obj、copy_array 或是直接複製內容，也因為實作這個原生方法，所以我們對陣列及 field data 都做了額外的 tag 表示其中的意義，來幫助我們在複製物件及陣列時，使用正確的複製的方式。

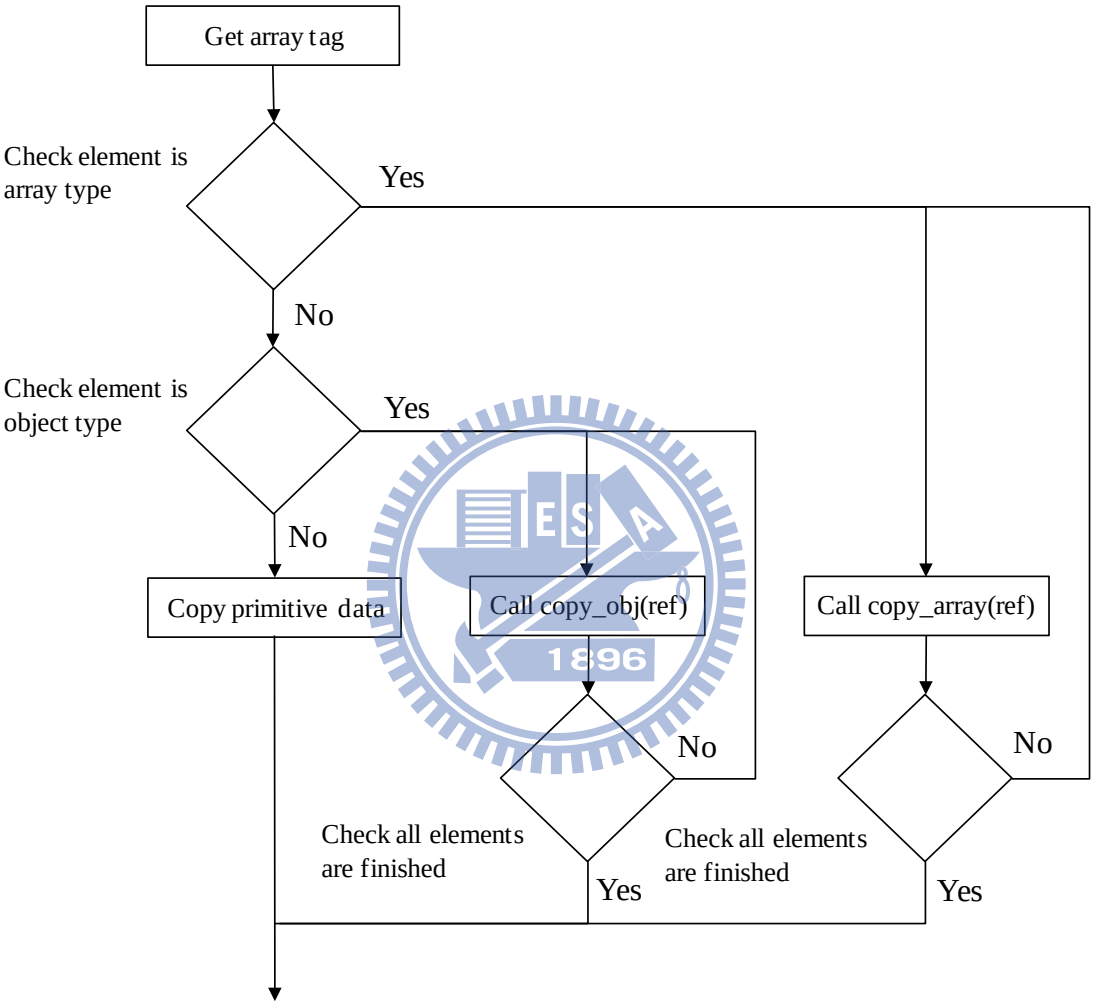

圖43. copy_array(ref)流程

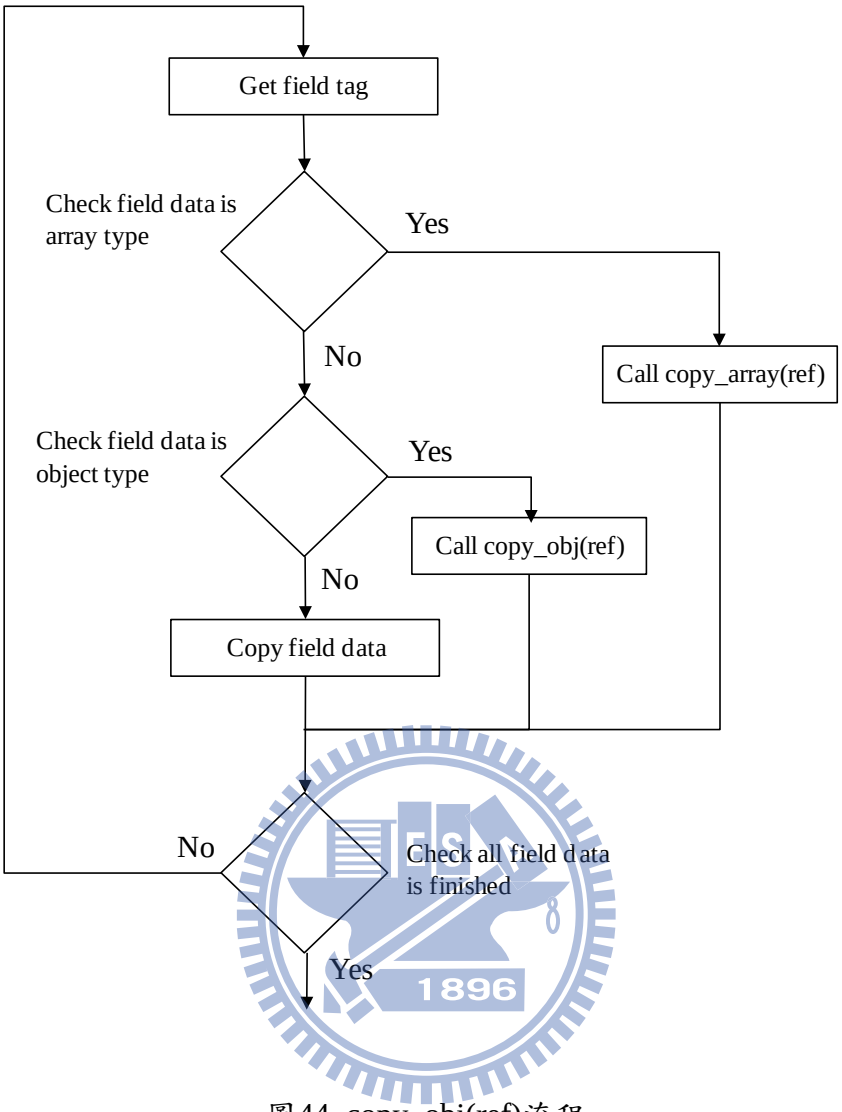

圖44. copy_obj(ref)流程

java/lang/System/getProperty0

透過取得的 String Object 參數，取出其字串內容的資訊，這邊參考自 KVM 中實作的方式會再呼叫 getSystemProperty 的函式，取得表示系統資訊的字串，並以此字串產生一個 String Object 回傳給 JAIP。

java/lang/String/charAt

在 String 的 field data 中，其中一項是用來描述另一個 field data 字元陣列的有效起始位置，當我們得到 String Object 參數時，會在其字元陣列中，將其起始位移量加上索引值得字元回傳給 JAIP。

java/lang/String/equals

比對兩個 String Object 在字元陣列的起始位置之後的內容是否相同，並將結果回傳給 JAIP。

java/lang/String/indexOf__I

在 String Object 上的 field data 中，從字元陣列的起始位置開始搜尋指定字元出現的索引值，並回傳給 JAIP。

java/lang/String/indexOf__II

在 String Object 上的 field data 中，從字元陣列的起始位置加上一個指定搜尋的位移量，開始搜尋指定字元出現的索引值，並回傳給 JAIP。

java/lang/StringBuffer/append__I

產生新的字元陣列，並將數值轉成字元附加在原來在 String Object 的字元陣列之後，再存回原 String Object 的 field data 中。

java/lang/StringBuffer/append__Ljava_lang_String_2

將欲附加的 String Object 的字元陣列附加在原來在 String Object 的字元陣列之後，再存回原 String Object 的 field data 中。

java/lang/StringBuffer/toString

透過取得的 StringBuffer Object 參數，修改其所屬 C lass ID 並對其 field data 的順序做調整，修改成符合 String 的格式。

第五章　效能評估

5.1. 實驗環境

在本論文中我們的系統實作採用 Xilinx Virtex5 ML507 的開發板做為我們的系統平台，架構如所圖 45 示，其硬體資源如下：FPGA XC5VFX70T 包含 hardcore PowerPC 440、 44800 slices、128 DSP8E functional units、5328 Kbits BRAM 及 256 MB DDR2 memory，而我們實作的 JAVA 加速處理器電路時脈為 100MHZ，合成所使用的硬體資源如圖 46 所示，其作為比較系統效能的對照組，CVM 與 CVM-JIT 使用平台為 Xilinx Virtex-4 ML405，其系統執行時脈也是同樣使用 100MHz。

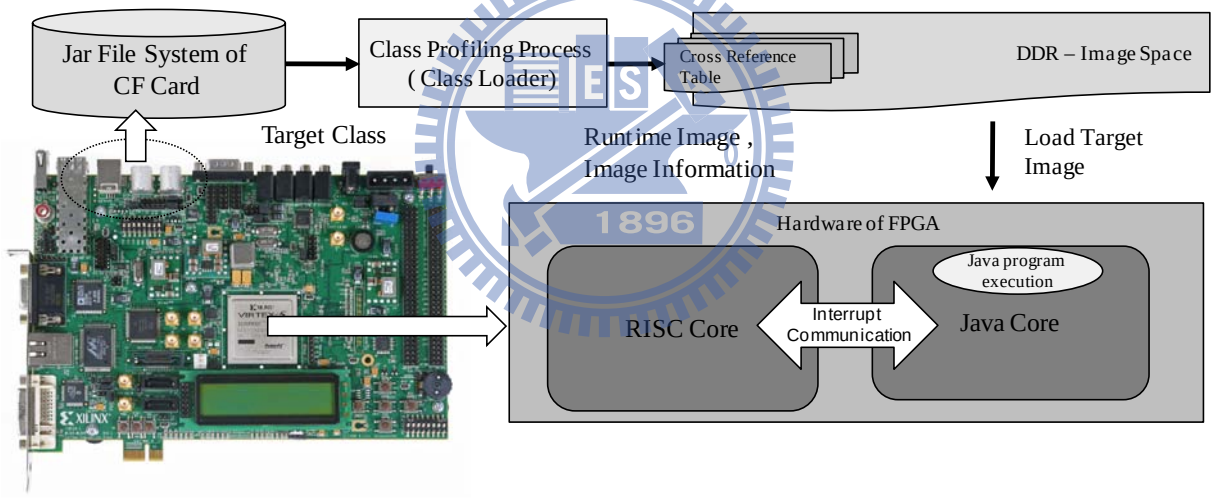

圖45. 系統架構圖

Selected Device : 5vfx70tff1136-1		
Number of Slices:	9252(3528) out of 44800	20(7)%
Number of Slice 6 input LUTs:	8755(4044) out of 44800	19(9)%
Number used as logic:	8390(4404)	
Number of IOs:	212(0)	
Number of bonded IOBs:	120 (0) out of 640	18%
Number of Block RAM/FIFO	35 (19) out of 148	23(12)%
Number using Block RAM only	35(19)	
Number of PPC440:	1 out of 1	100%
Minimum period: 10.681ns	Maximum Frequency: 93.624MHz The value of parentheses is only Java execution engine	

圖46. 在 XC5VFX70T 上的系統合成報告

5.2. 關於軟體部份的分析

5.2.1. 軟體大小的比較

針對軟體部份的分析我們首先與 CVM/CVM-JIT 比較在軟體上的大小，因為其也代表系統對記憶體的使用，比較結果如表 5，可以明顯看出我們系統無需仰賴作業系統的好處。

表 5 我們系統與 CVM/CVM-JIT 的軟體執行檔大小比較

	JAIP	CVM（CVM-JIT）
Software Size	282KB	5661KB

5.2.2. 動態物件型別載入（Dynamic Class Loading）機制的效能分析

對於 Dynamic Class Loading 的效能評估我們分為兩個部分，一個為解析的時間與原始物件型別大小及程式結構有關，另一個為載入的時間，因為載入時間通常僅取決於執行映像檔（Run Time Image）的大小，所以我們只針對解析時間去做分析，分別統計 CLDC 與 AWT 中的系統物件型別作為代表（共 135 的物件型別）分析其物件型別大小與我們系統轉換後的執行映像檔（Run Time Image）大小統計關係及其平均，如

表 6 所示，其統計的結果可以知道在我們系統設計底下，對物件型別轉換後的大小會有壓縮的效果，相較於 JIT 機制對記憶體的需求，我們的設計會更適用於對記憶體資源講究的嵌入式環境。

表 6 CLDC and AWT 與其轉換後的執行映像檔大小統計及取平均後的大小

Item / Size(byte)	Class	Run time image
0 - 1000	60.7 %	84.4 %
1000 - 2000	20.0 %	8.2 %
2000 - 3000	8.2 %	4.4 %
3000 - 4000	3.7 %	1.5 %
4000 – 500	3.7 %	0.7 %
5000 – 6000	1.5 %	0.0 %
6000 – 7000	0.7 %	0.0 %
7000 - 8000	0.0 %	0.0 %
8000 - 9000	0.0 %	0.0 %
>10000	1.5 %	0.7 %

Item	Class	Run time image
Average size (byte)	1288	568

接著我們再使用隨機產生的方式，隨機從 CLDC 與 AWT 中的物件型別挑選 100 個去做解析，分析其在解析上的效能及穩定性，其結果如所示圖 47、圖 48。

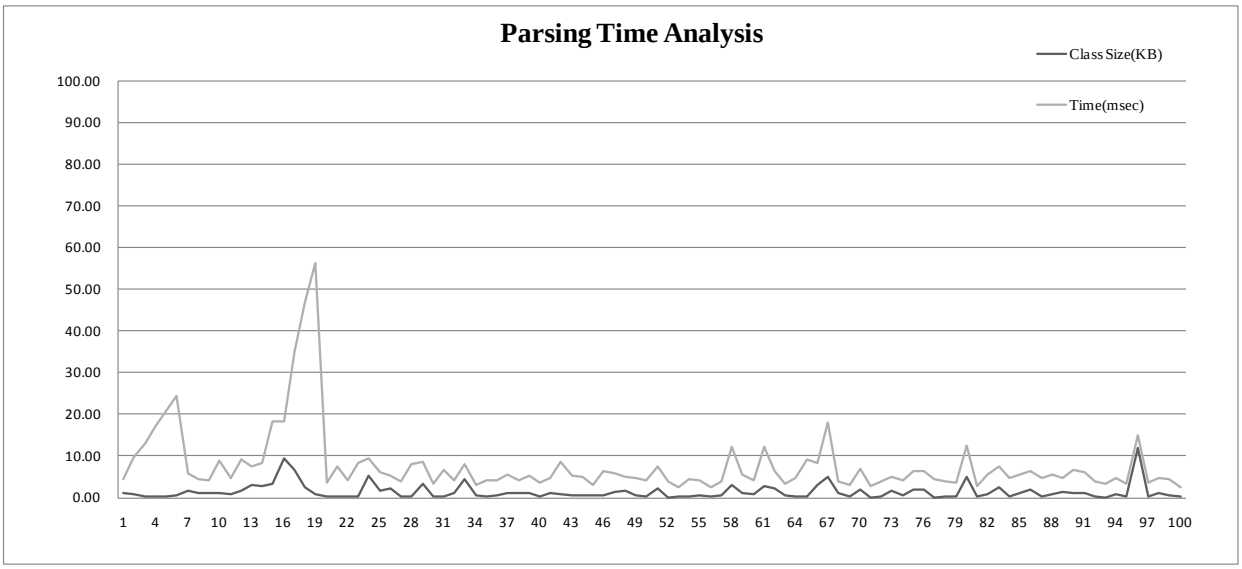

圖47. 對動態載入物件型別的機制分析其解析時間造成的負擔，載入類別為隨機挑選

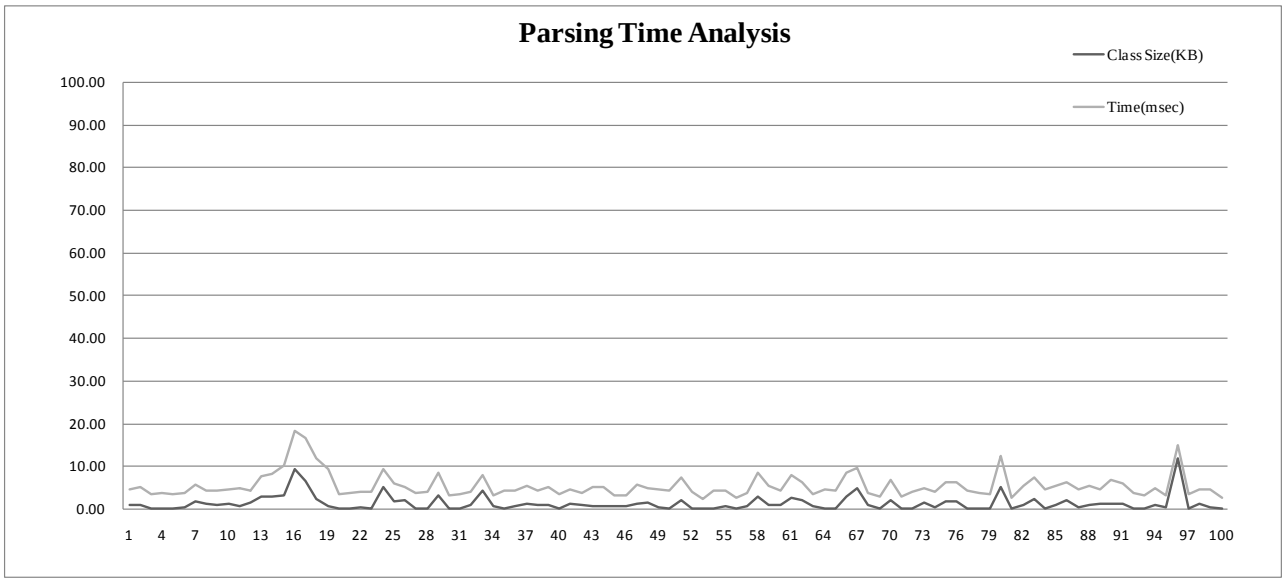

圖48. 同圖 47 的數據，減去解析物件型別時其解析父物件型別的時間

這邊圖 47、圖 48 的縱座標代表兩個數值，分別是物件型別大小(KB)以及解析時間(msec)，透過此測試數據可知瞭解系統並未因已解析的物件型別增加而有特別上升的趨勢，並解析的時間仍跟物件型別大小保持穩定的正相關。

5.3. 整體系統分析評比

在評估效能的第一部份，我們先使用在嵌入式環境底下最著名的 CaffeinMark，測試我們系統與 CVM、CVM-JIT 做比較，並針這部份的測試如 3.3 所提到的，我們也測試不同記憶體對我們平台效能的影響，測試結果如圖 49 所示，這邊我們僅保留將 JAIP 的 Heap Space 分別放在 DRAM 以及 BRAM 上作測試的效能，參照資料皆已固定放至 JAIP 的 on-chip 記憶體中，進而推估當系統未來支援 Data Cache 時的效能，在效能上我們可以明顯看到比 CVM 好上許多，然而與 CVM-JIT 相比仍然有改善的空間，但如 5.2.1 所提，我們的系統也有其優勢所在。

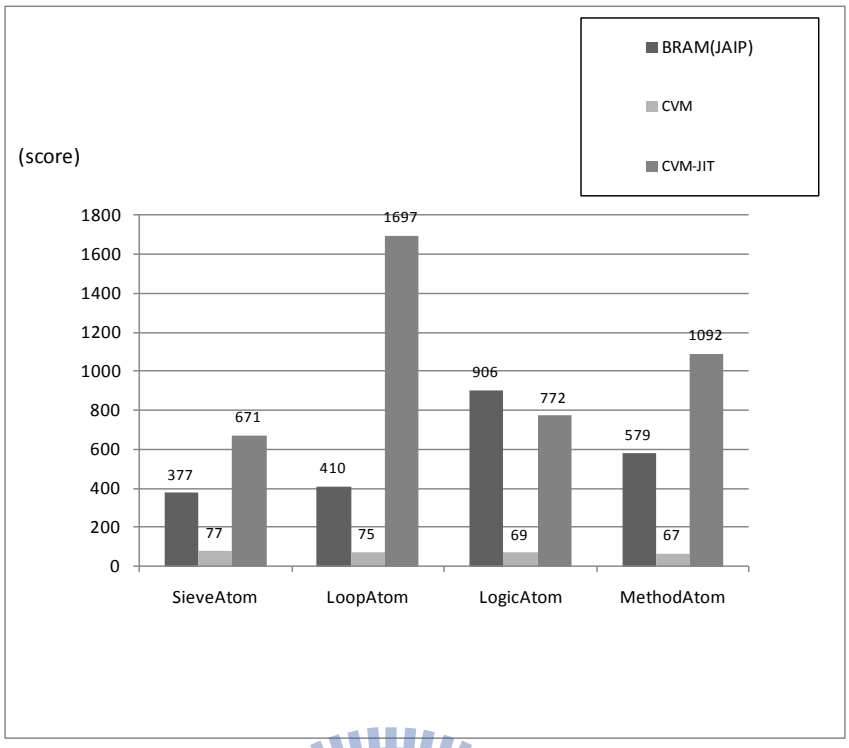

圖49. 執行 CaffeinMark 與 CVM/CVM-JIT 評比效能

透過這部份的實驗，在我們所使用的開發平台(Xilinx Vitex-5 ML507)上，我們也測試原平台中預設掛在 PLB Bus 上的 BRAM，但是經由我們將 Heap Space 存放在不同記憶體區塊，再執行 JAVA 程式對 Heap Space 存取測試得到的效能，得知實際透過 PLB Bus 存取 BRAM 的所需花費約 7~8cycles（結果如表 7），故我們最後所使用的 BRAM 皆為建立在 JAIP 內部的 on-chip 的記憶體而非經由 PLB BUS 所存取的。

表 7 測試平台對不同記憶體存取的效能

Memory / Time	DDR2	BRAM (PLB)	BRAM (on-chip)
Cycle	18.87	8.39	1

最後我們設計了兩個程式來再對 JIT 的機制去做測試，其中之一針對 JIT 可能會將遞迴呼叫的程式展開成迴圈來執行的底成機器語言來執行的特性，所以我們設計了一連串根據輸入值來決定循環呼叫次數的程式，使得 JIT 在編譯階段無法決定何時會結束，另外則是側是純粹執行計算時我們系統與 CVM-JIT 的效能評比，結果如圖 50 所示。

循環呼叫的驗證程式說明：

程式設計為 40 個 class 循環呼叫屬於別的 class 的 static method，class1.class 會呼叫 class2.class 的 static method，而 class2.class 會呼叫 class3.class 的 static method，以此類推至 class40.class 會呼叫 class1.class 的 static method，並在 class 循環呼叫的同時會為下一個呼叫的 method 傳入參數，而傳遞給下一個 method 的參數為上一個 class 呼叫目前執行的 method 時的所傳遞過來的參數值減一，循環呼叫直到傳遞的參數值等於 0 才停止呼叫下一個 method，驗證程式內容為 initial 起始傳入參數為 100 來開始循環呼叫，並再讓這樣循環呼叫的測試執行 1000 個迴圈。

計算的驗證程式說明：

使用測試我們系統中雙指令同時執行的效率，其程式內容為計算 Pi 值的程式，用來計算 Pi 值至精準到小數點以下第 499 位，含整數部分為 500 位數，將相同計算的過程在迴圈內反覆執行三次。

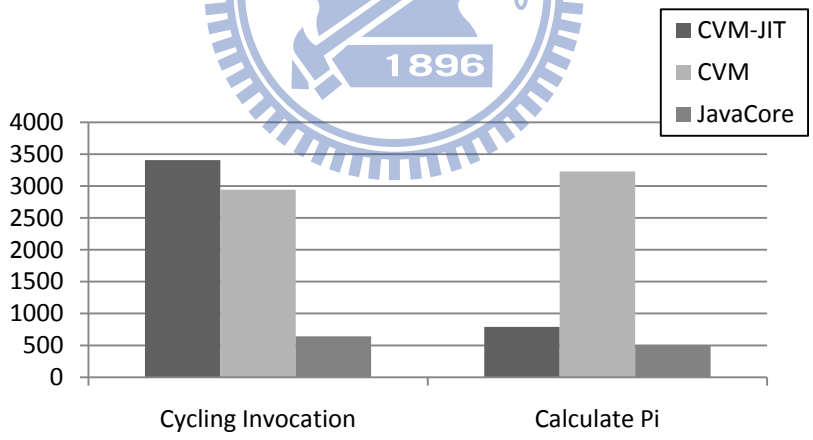

圖50. 執行循環呼叫及計算 PI 程式的效能評估

由我們自行設計的程式在評比效能皆能比 CVM-JIT 有更好的表現知道，光是測試 CaffeinMark 仍未反應出程式的所有特性，未來我們將嘗試測試更複查的 Benchmark 來加入我們實驗結果中。

第六章　結論

在本論文中實作的針對新架構 JAVA 加速處理器當中所需的堆疊記憶體，可以有效解決在堆疊存取時結構危障的發生，進而提升雙指令同時執行的比例，並透過對系統記憶體配置的實驗測試結果，在成本及效能上的考量後，實作上將會把參照資料（Reference Data）放入 on-chip 記憶體的設計納入系統架構中，接著透過對系統軟體的修改，完成繼承及介面的功能，提供對 JAVA 物件導向語言特性的支援；最後再實作對原生方法（Native Method）呼叫的介面，提供更完整的 JRE（Java Runtime Environment）。

在未來的實作上，因為我們有了對呼叫原生方法的支援，將可以完整實作支援多執行緒功能所需的物件型別，在 JAVA 語言中要產生一個執行緒，可以透過繼承 Thread 這個物件型別或是實作 Runnable 介面，兩種方式的差異在於繼承 Thread 物件型別的方式，可以直接呼叫其繼承至 Thread 的原生方法－”start”，在系統中增加一個執行緒，而透過實作 Runnable 介面的方式則是需將實作 Runnable 介面的物件作為引數，去實例化一個 Thread 型別的物件，而其所產生的物件會將實作 Runnable 介面中名為”run”的方法設為此執行緒的程式進入點，但是如果要使系統真正增加一個執行緒，一樣仍需透過這個以 Thread 物件型別所產生的物件去呼叫其原生方法－”start”，透過以上流程的介紹，可以知道在多執行緒上的功能，實作對原生方法呼叫的支援其必要性，之後也將以實作多執行緒及對系統物件型別提供更完整的支援為目標繼續研究。

參考文獻

[1] S. Ritchie, “Systems Programming in Java,” *IEEE Micro*, 17, 3 (Mar.), 1997, pp. 30-35.

[2] B. R. Montague, “JN: OS for an Embedded Java Network Computer,” *IEEE Micro*, 17, 3, 1997, pp. 54-60.

[3] T. Lindholm and F. Yellin. The Java Virtual Machine Specification. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[4] Hou-Jen Ko and Chun-Jen Tsai, A Double-issue Java Processor Design for Embedded Application, Proc. of IEEE Int. Symp. on Circuits and Systems, May. 2007.

[5] Hou-Jen Ko, A Double-issue Java Processor Design for Embedded Application, NCTU, 2007.

[6] Chien-Fong Huang, Design of Dual-Core Java Processor for Interactive 3-D GUI Platform, Mater thesis, NCTU, 2010.

[7] The Java Community Process Program, JSR 36: Connected Device Configuration, ver. 1.0b, Dec 20, 2005.

[8] Connected, Limited Device Configuration Specification Version 1.0a, Sun Microsystems White Paper, May. 2000.

[9] Jun Qin, Qiaomin Lin, and Xiujin Wang, Research on Embedded Java Virtual Machine and its Porting, IJCSNC International Journal of Computer Science and Network Security, Vol.7 No.9, September 2007.

[10] Yi-Ting Wang, A Java Accelerator, NCKU, 2006.

[11] Cheng-Che Chen, Ying-Tien Huang, Chen-Hung Yang, Java Virtual Machine on CCL Java Coprocessor, CCL Technical Journal, no. 103, Mar 2003, pp56-67.

[12] C.-H. Hsieh, J. C. Gyllenhaal, and W. W. Hwu, Java Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results, Proc. of 29th Annual ACM/IEEE Int. Symp. on Microarchitecture (MICRO’29), pp. 90-99, Paris, Dec. 1996.

[13] A. Krall, “Efficient Java Just-in-Time Compilation,” Proc. of Int. Conf. on Parallel Architectures and Compilation Techniques, pp. 205-212, Paris, Oct. 1998.

[14] Design And Reuse Communication Inc, D＆R Headline News, Communication Inc, Dation,” Proc.eviews Silicon-Base Acceleration Technology for Wireless Internet Applications”, Also available on the website: http://www.us.design-reuse.com/news/news1186.html.

[15] Ajile Inc, “aJile Java Processor Core JEMCore”, 2001.

[16] Ajile Inc, “aJ-100 TM Real-time Low Power Java TM Processor”,Processor Datasheet, 2001.

[17] Sun Inc, ”picoJava-II Processor Core”, Datasheet,1999.

[18] Harlan McGhan, Mike O’Connor, “PicoJava: A Direct Execution Engine For Java Bytecode”, IEEE 1998.

[19] InSilicon Inc, JVXtreme Java Accelerator Core, 2001.

[20] Markus Levy, “Java to go: Part 2”, Microprocessor Report, March, 2001.

[21] Nazomi Communication, inc, “JSTAR-Java Coprocessor for ARM Microprocessors”.

[22] ARM inc, “Jazelle technology: ARM acceleration technology for the Java Platform”, 2004.

[23] Markus Levy, “Java to go: Part 3”, Microprocessor Report, March, 2001.

[24] D.S. Hardin. Real-Time Objects on the Bare Metal: An Efficient Hardware Realization of the JavaTM Virtual Machine. In Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing, page 53. IEEE Computer Society, 2001.

[25] PTSC. IGNITE Processor Brochure, Rev 1.0. Available at http://www.ptsc.com.

[26] R. Zulauf. Entwurf eines Java-Mikrocontrollers und prototypische Implementierung auf einem FPGA. Masterasterasts, University of Karlsruhe, 2000.

[27] S.A. Ito, L. Carro, and R.P. Jacobi. Making Java Work for Microcontroller Applications. IEEE Design & Test of Computers, 18(5):100):100rs, 18

[28] Martin Schoberl, JOP: A Java Optimized Processor for Embedded Real-Time Systems, Ph.D. Thesis, Tech. Universitaet Wien, Jan 2005.

[29] Z. Qian, A. Goldberg, and A.Coglio, CA Formal Specification of Java Class Loading,ication of Javao, o, r EmbHillview Avenue, Palo Alto, CA 94304 July 21, 2000.

[30] S. Liang and G. Bracha, acha, and G. Bracha, 4304 July 21, 200021, 2000l-Time SysteMicrosystem Inc. 901 San Antonio Road, CUP02-302 Palo Alto, CA 94303.

[31] A. Gatherer, T. Stetzler, M. McMahan, and E. Auslander, slander, n, and E. Auslander, lander, d E. Auslas: past, present and future,uture,futuCommun. Mag., vol.38, pp.84-90, Jan. 2000.

[32] S. Liang, The Java Native Interface: ProgrammermerProgrammerslas: past, present and future,: pas

[33] D. Kurzyniec and V. Sunderam, Effcient Cooperation between Java and Native codes - JNI performance benchmark, present and future,lo Alto, CA 94303 94303, CA 94303430303 94303o Alto, CA 94303y, 2001.9, Paris9, Paris9, Parisris