國立交通大學

資訊科學與工程研究所

碩 士 論 文

多執行緒 Java 處理器設計

Design of the Multithreading Architecture for a Java Processor

研 究 生：蘇宏成

指導教授：蔡淳仁　教授

中 華 民 國　102　年　7　月

多執行緒 Java 處理器設計

Design of the Multithreading Architecture for a Java Processor

研 究 生：蘇宏成　　Student：Hung-Cheng Su

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 2012

Hsinchu, Taiwan, Republic of China

中華民國 102 年 7 月

摘要

在此篇論文中，我們提出了兩種多執行緒的 Java 執行環境，包括單一核心以硬體切換不同執行緒的 Temporal Multithreading (TMT)，以及多核心並行的 Simultaneous Multithreading(SMT)。首先在單一 Java 核心 TMT 架構方面，因為 Java 本身就是一種多執行緒的程式語言，而每個執行緒都擁有自己的執行資訊與運算堆疊。若是利用多組硬體堆疊來維護所有執行緒的狀態，對於嵌入式系統設計的成本過大。因此，在我們的設計中提出了兼具效能與成本的解決方案，將各執行緒所運作的堆疊資料在 DDR-SDRAM 中存有備份，並在 Java 處理器中利用兩組運算堆疊，當處理器在執行時使用其一的堆疊運算，並將另一堆疊準備好提供接續的執行緒可順利執行。對於執行緒的管理完全是由 Java 處理器核心負責。實驗顯示，所提供的架構在多執行緒環境下支援執行緒的切換是較有效率的。而第二種 SMT 則是多處理核心的執行環境，利用在系統中加入多組 Java 處理核心，使得執行緒的運作可以並行執行。並且由於在執行時的各處理器上的 Heap 資料可能會不一致，因此我們設計了一個處理資料一致性的機制來確保執行的正確性。結果顯示，在多核心的架構下使得執行緒平行執行確實可以提高執行效率。而我們所提出的 Java 處理器架構也在 Xilinx ML-605 FPGA 平台上實作驗證。

誌謝

在兩年研究所的學習與研究下終於產生了此篇論文。首先一定要感謝我的指導教授蔡淳仁教授。在研究所這短短的兩年間，教授給予了相關研究的經驗跟自我學習的機會，並在跟老師一起討論研究的過程中，常常能了解到自己研究上的缺失以及提升自己在解決問題的能力。雖然這兩年內常常在研究上遇到瓶頸，但老師還是很耐心的與我一起研究問題，在此很感謝老師的諄諄教誨。而在解決問題的過程中，學長與實驗室的同學們意見也幫助了我很多，藉由一起討論問題，可以從不同的觀點來分析自己的研究設計，感謝各位實驗室的同學們。再來要感謝我的家人們，在求學的學習路上並不簡單，感謝家人們的支持與指引。最後感謝我的一群好朋友，A20 的大家(阿升升、喬喬、阿噹噹、阿龍仔、宗儒、長線、鮪魚與綠包)，在這兩年內不論是遇到研究上的障礙還是對自己研究失去信心時，總是有你們給我鼓勵(？)與陪著我到處吃喝玩樂，讓我暫時可以忘卻研究上的煩惱，好好放空自己充個電。可以有動力的重新面對研究並再度全力以赴，謝謝。最後不免俗的，要感謝的人太多，那就謝天吧。

目錄

摘要..I

誌謝..II

目錄..III

圖目錄..V

表目錄..VII

第一章 前言...1

1.1. 研究動機...1

1.2. Java 執行環境..2

1.3. 論文架構...4

第二章 相關研究...5

2.1. 異質雙核心 Java 處理器....................................5

2.2. 多執行緒相關研究..9

第三章 Temporal Multithreading 系統架構.....................13

3.1. Thread Manager Unit.......................................14

3.1.1. 執行緒新增與終結..16

3.1.2. Thread Controller.......................................17

3.1.3. Thread Control Block....................................21

3.1.4. 執行緒佇列..23

3.1.5. Stack Manager...25

3.1.6. 執行緒同步..27

3.2. Ping-pong Java stack memory 28
3.3. 系統軟體 30
3.3.1. 呼叫方法流程 31
3.3.2. Method Flag 34
3.3.3. 類別解析器對於新增執行緒之設計 35
3.3.4. 硬體原生方法介面 37
第四章 多處理核心系統架構 39
4.1. 多處理器執行環境 40
4.2. Object Heap Cache 44
4.3. Data Coherence Controller 47
4.3.1. Heap Cache Coherence 之機制 48
4.3.2. Java 同步機制之支援 50
第五章 實驗結果 52
5.1. 實驗環境 52
5.2. Benchmark 分析 53
5.2.1. Single-Thread 效能分析 53
5.2.2. Temporal Multithreading 效能分析 54
5.2.3. 多處理核心效能分析 55
第六章 結論與未來展望 57
參考文獻 58

圖目錄

圖 1. 傳統 Java 執行環境 2
圖 2. 異質雙核心系統架構 5
圖 3. Four-stage pipelines of bytecode execution engine 6
圖 4. Two-level Java stack memory 7
圖 5. Dynamic resolutionru 機制有限狀態機 8
圖 6. Cross Reference Table 類別存放資訊 9
圖 7. 處理器支援多執行緒不同方法 10
圖 8. Dual stack cache 11
圖 9. The Komodo processor core. 11
圖 10. The JOP chip multi-processor system 12
圖 11. 多執行緒異質雙核心系統架構 13
圖 12. Thread Manager Unit 14
圖 13. 切換執行緒流程 15
圖 14. 新增執行緒之 dynamic resolution 機制 17
圖 15. Thread Controller 有限狀態機 18
圖 16. 執行緒首次執行之 dynamic resolution 機制 20
圖 17. Thread Control Block 21
圖 18. 執行緒佇列 23
圖 19. 環狀佇列運作示意圖行緒佇列 24
圖 20. Stack Manager 25
圖 21. Stack Manager 有限狀態機 26
圖 22. Ping-pong Java stack memory 28
圖 23. Ping-pong Java stack memory 控制示意圖 29
圖 24. 新增執行緒程式 31
圖 25. 呼叫方法流程圖 32
圖 26. Dynamic Method Loading 流程圖 33
圖 27. 方法資訊欄位更新 35
圖 28. Cross Reference Table 查找 start()方法資訊 37
圖 29. 多處理核心環境之 JAIP 系統架構 40
圖 30. 多處理核心系統架構 40
圖 31. JAIP Information Table 41
圖 32. 多處理核心系統運作 42
圖 33. JAIP Information Table 之改變 43
圖 34. Object Heap Cache 45
圖 35. Object Heap Cache 存取有限狀態機 46

圖 36. Cache Updata 有限狀態機......47
圖 37. Data Coherence Controller......48
圖 38. Heap Cache Coherence 運作機制......48
圖 39. Heap Cache Coherence 機制流程圖......49
圖 40. 同步機制運作流程......50
圖 41. Synchronized Manager 有限狀態機......51
圖 42. 執行 JemBench 單執行緒效能評比......54
圖 43 . JAIP Temporal Multithreading 機制與 CVM-JIT 效能評比......55

表目錄

表 1. Number of PLB Master Burst Read 27
表 2. Number of PLB Master Burst Write..27
表 3. Method Flag..34
表 4. java.lang.Thread 類別的原生方法 38
表 5. JAIP 合成資訊 52
表 6.多處理核心效能分析 56

第一章 前言

1.1. 研究動機

Java 程式語言自 1995 年被 Sun 公司推出至今，隨著其具有跨平台、物件導向、多執行緒...等優點，一直以來被受重視。而近幾年更因其跨平台的特性，只要將原始碼編譯成位元組碼（Bytecode），再透過各平台的 Java 虛擬機器（JVM）[1]的技術，即可使得相同的程式可執行於不同的作業系統或處理器上。因此，在越來越多的嵌入式平台應用上都選擇 Java 做為開發程式的主要語言。透過 Java 作為一個標準應用程式開發語言也能使用所支援的各式應用程式介面(Application Programming Interface)，並且 Java 程式語言為一個設計良好的物件導向程式語言，其產量比 C++高於 40%以上[2]。因此，有許多不同的組織提出支援不同的服務要求，例如由 Java 創造者 Sun 公司所推出的 Java Platform, Micro Edition (Java ME) [3] ，是一種為了嵌入式裝置所設計的架構。另一例子為近年來 Google 所提出的 Android 系統，它使用 Java 類別庫為基礎架構於 Apache Harmony，使得大多數的 Android 應用程式皆使用 Java。

Java 程式支援多執行緒的設計，在部分應用程式上使用循序執行即可解決問題，但有些應用程式卻須執行多個並行的程式，例如聊天室軟體需要不同執行緒去監控不同使用者的操作，甚至這些執行緒可能同時執行。當處理器能同時處理一組以上執行緒[4][5]或系統中存在多顆處理器[6][7]，即可使得多執行緒可以平行執行藉以提高執行效率。否則只能於單顆處理器上透過切換執行緒的機制，在同一時間執行的只有單一執行緒。在此論文提出的機制中，試著提出在 Java 加速處理器上支援多執行緒的解決方式，包含單一處理器多執行緒與多處理器設計。然而要使 Java 處理器能支援多執行緒的機制，必須要付出一定的成本[8]，每個執行緒必須保有自己的執行資訊，例如 program counter、運算堆疊等。因此，在嵌入式系統上，Java 加速處理器去設計一個有效率與低成本的多執行緒機制是個重要的議題。而且在多執行緒的環境之下，可能在各執行緒並行時互相干擾，導致資料的不一致性，所以如何保持一致性，也是此研究的重點之一。

1.2. Java 執行環境

Java 運行環境是由 Java 虛擬機器與一組標準類別庫所組成，在 Sun 公司所推出的 Java Platform, Micro Edition 目標裝置從工業控制到行動裝置等。此外 Java Platform, Micro Edition 還區分為兩類。用於小型行動裝置或用以實現較多功能的設備，如智慧型手機或電視機上盒等。把運算功能有限、電力有限的小型行動嵌入式裝置定義在 Connected Limited Device Configuration(CLDC)[9]規格之中，而另外功能較為強大的裝置則規範為 Connected Device Configuration(CDC)[10]規格。

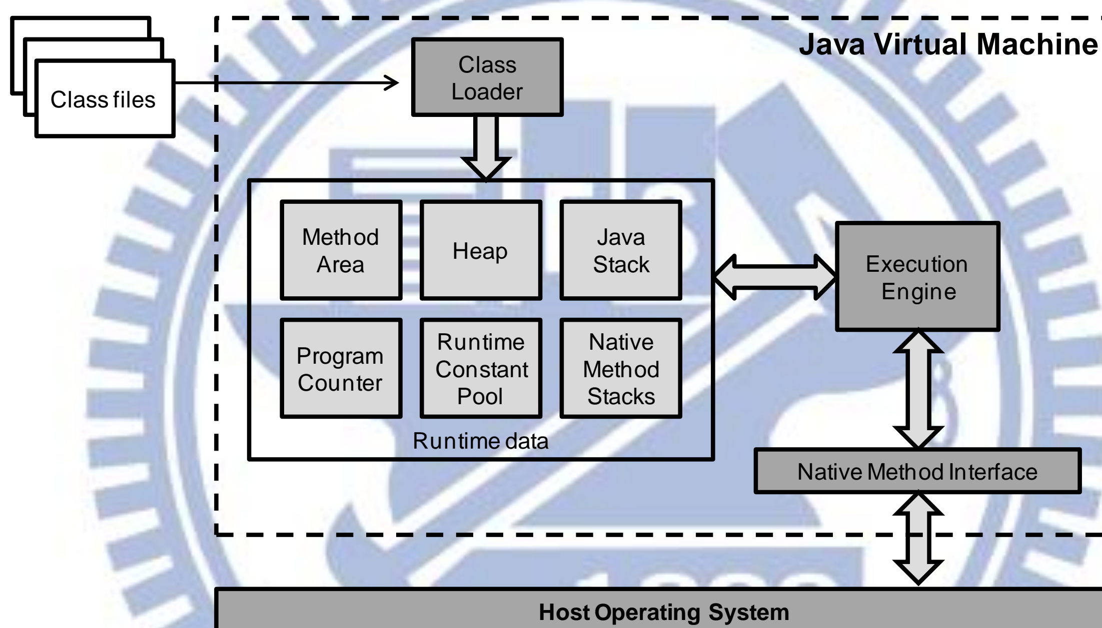


圖 1. 傳統 Java 執行環境

傳統 Java 執行環境(如圖 1 所示)包含軟體的 Java 虛擬機器，它依賴完整的作業系統去執行 Java 應用程式。在 Java 虛擬機器中有三個重要元件，Class Loader、Runtime Data 與 Execution Engine[11]。Class Loader 此元件負責載入指定的類別(Class)或介面(Interface)。Runtime Data 為提供 Java 虛擬機器在執行程式時所需資料，包括運算堆疊、執行方法等。Execution Engine 負責去執行所載入方法的指令。

雖然 Java 虛擬機器在實作時可以只用軟體直譯器（Interpreter）來執行 Java 程式，像是Sun 的CVM[12]與KVM[13]就是在嵌入式裝置中利用軟體實作Java 虛擬機器的直譯器[14]，但是直譯器在執行 Java 程式時相當沒有效率。因此大部份 Java 執行環境都

會使用加速的虛擬機器。有許多解決方法可以改善在嵌入式環境下 Java 應用程式的效能。這些解決方法提供了 Java 執行時空間或時間的改善。大致上可分為三類。單純以軟體加速的編譯器(例如：Just-in-Time 編譯器[21])、以硬體為基礎的協助處理器(例如：ARM Jazelle[15]與 Nazomi JSTAR[16])、獨立執行程式的 Java 處理器(例如：Sun picoJava[17][18]及 aJile aJ-100[19][20])。

利用 Just-in-Time 編譯器的技術藉由將 Java 位元組碼轉換成轉換為原生機器碼[22]並暫存於系統之中，可顯著的提高執行效率，但 Just-in-Time 需要額外的記憶體空間和在類別載入時所附加額外的編譯開銷。因此在對記憶體資源較為嚴格要求的嵌入式應用設計較不適合。另一提高 Java 執行效率的方法為單純以硬體為基礎所架構。像是內嵌轉譯器（Embedded Java Translator）的 ARM 的 Jazelle 及 Nazomi 的 JSTAR，其具備可以切換執行 Java 位元組碼的處理器設計、或像是獨立式的 Java 處理器 Sun picoJava、Komodo [23]及 JOP [24]，其處理器的設計可單獨執行 Java 程式。而這些處理器皆是依照 Java 虛擬機器的堆疊機制所設計，因此其執行的效率會比一般處理器所運行的 Java 直譯器來的高。然而，有些程式執行的運作在純硬體的環境下會顯得較為沒有效率，像是解析類別、記憶體管理、檔案存取等。因此我們的所設計的系統使用異質雙核心的架構設計[25][26]，此架構的 Java 運行環境包含一個 RISC 處理器與一個 Java 處理器。Java 處理器用於有效的執行 Java 位元組碼，而 RISC 處理器則是使用軟體運作的解決方法來執行純硬體較無效率的運作藉以提供更好的執行效率。在此架構下，Java 位元組碼的指令絕大部分不需要依賴 RISC 處理器來執行，僅有在需要呼叫底層方法實作的指令或是載入類別解析時，才會透過 Inter-process communication 介面呼叫 RISC 處理器來協助。這樣的設計下使得我們無需像其他 Java 加速處理器的設計[27]，會有需要移植 KVM 至平台中讓 RISC 處理器執行 Java 位元組碼的負擔。

1.3. 論文架構

在此篇論文中，以我們先前所研究的異質雙核心 Java 處理器為基礎[28][29][30]，呈現了兩種不同Java處理器的設計之多執行緒架構與機制。其一為 Temporal Multithreading 機制，在只有單一 Java 處理核心的情況下，同時間只能有一組執行緒被執行。而透過切換執行緒的機制，能達到類似執行緒並行執行的假象。每個執行緒都擁有自己的資訊與運算堆疊(Java Stack)，若要維護所有執行緒的狀態，對於 Java 處理器的成本過大。因此，在我們的設計中提出了通用解法，將各執行緒所使用的堆疊在 DDR-SDRAM 中存有一備份，並在 Java 處理器中利用兩組運算堆疊，當執行緒在對其一堆疊運算時，並將另一堆疊準備好以提供接續的執行緒可順利執行。對於執行緒的管理不須經由 RISC 處理器而完全是由 Java 處理核心負責，可避免處理器間溝通所造成的成本。另一種架構為多個 Java 處理核心執行環境，由於單一處理核心執行多執行緒並不能使執行效率提升，因此我們設計使用兩顆以上的 Java 處理核心來執行多執行緒程式，而由 RISC 處理核心作為執行緒的管理者。並在 Java 處理器上加入的快取記憶體(Cache)機制，然而在執行過程中會造成各處理器的快取記憶體不一致的狀況。因此，我們提供了一個 Data Coherence Controller 機制，去維護各記憶體的快取記憶體資料保持一致。

本論文一共分為六章，本章是一個概略性的導論，說明背景、動機以及論文大綱；第二章為相關研究，會先介紹先前所研究作為設計基礎的異質雙核心 Java 處理器，並接著介紹與 Java 多執行緒機制相關研究；第三章開始是對於在單一處理器上使用 Temporal Multithreading 系統架構的說明；第四章則是對於多處理器系統架構的說明，並提出快取記憶體與 Data Coherence Controller 的機制；第五章則分析實驗的結果；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

在此章節中，會先呈現我們設計時作為基礎所使用的嵌入式異質雙核心 Java 處理器。之後會介紹不同多執行緒機制的相關研究。

2.1. 異質雙核心 Java 處理器

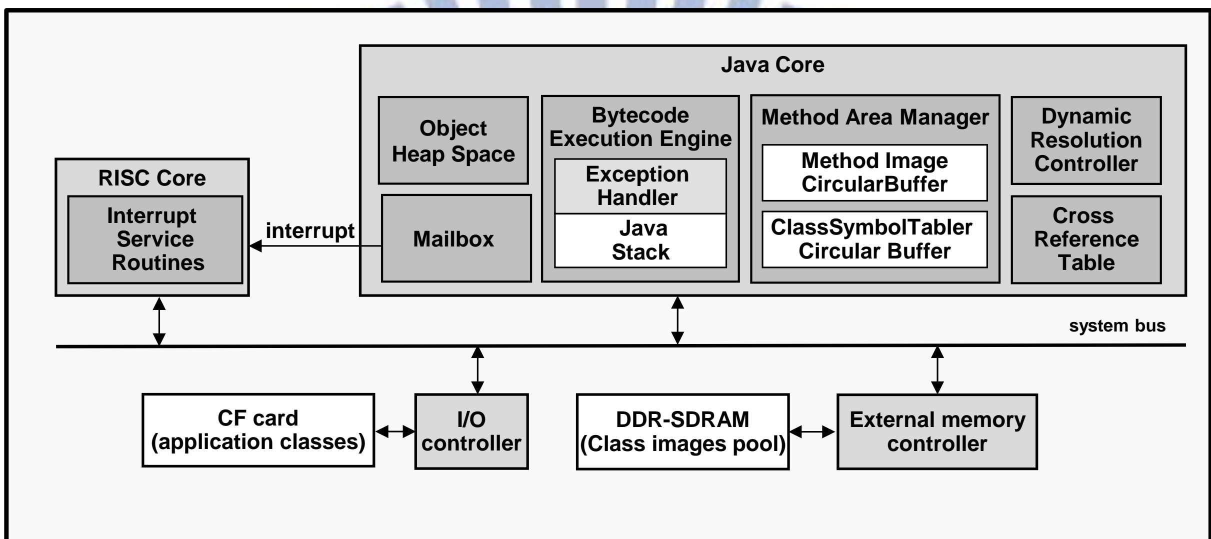


圖 2. 異質雙核心系統架構

在這篇論文裡，我們使用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 處理器，我們稱之為 Java Accelerator IP （JAIP）電路架構(如圖 2 所示)，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，並透過所提供的系統軟體[31]輔助來對完整的 Java 特性提供支援。對於整個系統分為兩個部分，第一部分為執行 Java 程式的執行引擎，用來執行程式的 Java 位元組碼，第二部分則是透過 RISC 處理器來執行我們所提供的系統軟體，這部分包括類別解析器（Class Parser）、對 Java 部分指令支援所提供的程序以及實作系統類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[32]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

有的作業系統或處理器中。在這個架構下，對於 Inter-process communication(IPC)採用較簡單的介面完成單方向呼叫的機制，即讓 JAIP 可以透過這個機制呼叫到 RISC 處理器所提供的服務。並在 JAIP 上利用 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 處理器。因此在執行 Java 應用程式時，絕大部分的 Java 位元組碼可在高效能的 Java 處理器上運作。只有少部分的位元組碼牽涉到 I/O 控制或 Java 類別檔案載入及解析，會透過中斷交由 RISC 處理器的中斷處理機制(Interrupt Service Routine)來實作。

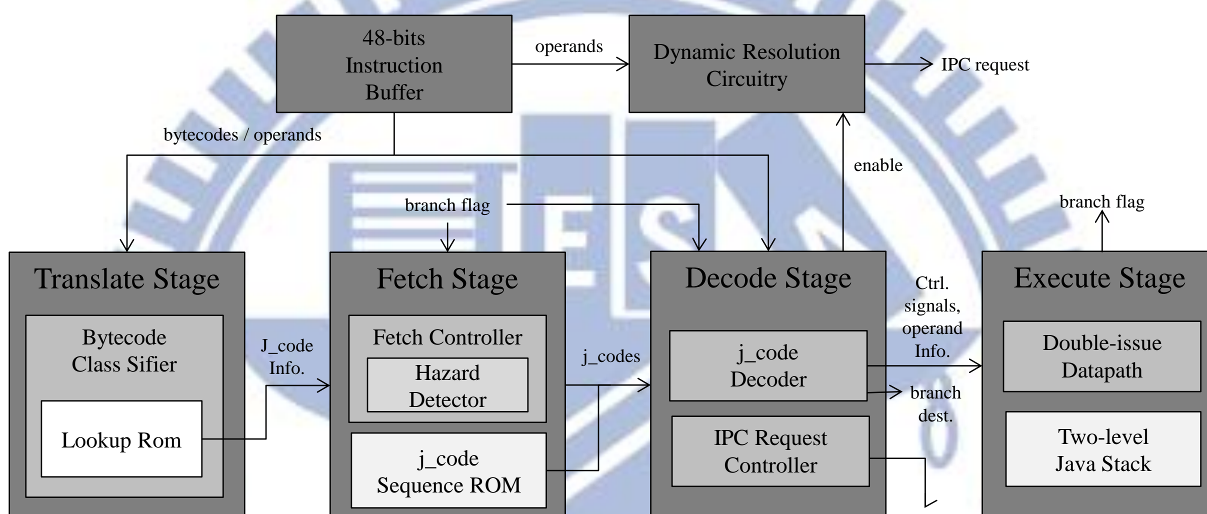


圖 3. Four-stage pipelines of bytecode execution engine

JAIP 的設計是可以同時間執行雙指令，Java 位元組碼運作於執行引擎(如圖 3 所示)。此位元組碼執行引擎採用 four-stage pipeline 架構，包含 Translate Stage、Fetch Stage、Decode Stage 和 Execute Stage。由於 JAIP 是一個獨立的 IP 並不需依賴任何處理器的架構，因此可以較容易的將 JAIP 整合到任何有支援利用中斷去執行 Inter-process communication 的處理器上。

在 JAIP 的實作上定義了自己的一套 Instruction Set Architecture(ISA)稱之為 j_code 用以執行 Java 位元組碼，並將 Java 位元組碼型態分類成 simple、complex 以及 operand 三種。大部分指令都是一對一的轉換，稱之為 simple。少部分的指令(例如 invoke 指令等)則是轉換成 code sequence，稱之為 complex。附帶在指令後面的參數則稱之為 operand。當 Translate Stage 從 Instruction Buffer 中取得位元組碼後。如果取得的是 simple

指令，則單純地轉換成單一 j_code 並送到 Fetch Stage。如果取得的是 complex 指令，即會轉成對應於 Fetch Stage 的 j_code sequence 起始位址。若取得的是 operand，則此 operand 的值將會在 Decode Stage 直接由 Instruction Buffer 中提取。Fetch Stage 每個週期負責傳送一對 j_code 指令到 Decode Stage。如果從 Translate Stage 傳送過來的是 complex 指令的 j_code sequence 起始位址，則會從 j_code sequence ROM 中提取一對 j_code 指令送到 Decode Stage。然而在 j_code sequence ROM 中指令皆是成對的，有些成對的指令可能會使用到 NOP。當 Fetch Stage 把真正要執行的 j_code 傳到 Decode Stage 之後，Decode Stage 便會同時解析傳過來的一對指令並產生相應的控制訊號以及拉起某些特定訊號。最後 Execute Stage 即根據 Decode stage 所生成的控制訊號來執行相關運作。

Java 虛擬機器是為 stack-based 的設計，因此在 JAIP 的實作中堆疊架構與堆疊的運作也是設計的重點。為了實現雙指令執行的運作，JAIP 設計了一組特別的 Two-level Java stack memory(如圖 4 所示)，第一層是由三個暫存器所組成，負責去存放 Java stack 最上方的三筆資料。第二層的 Java stack 則是由兩塊 Dual-port On-chip BRAM 來實作交錯式（Interleaving）的記憶體架構，並針對每一塊 BRAM 將 Dual-Port 分成 Read 以及 Write 的兩個 Port。並使用四個暫存器作為存取較為頻繁的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，

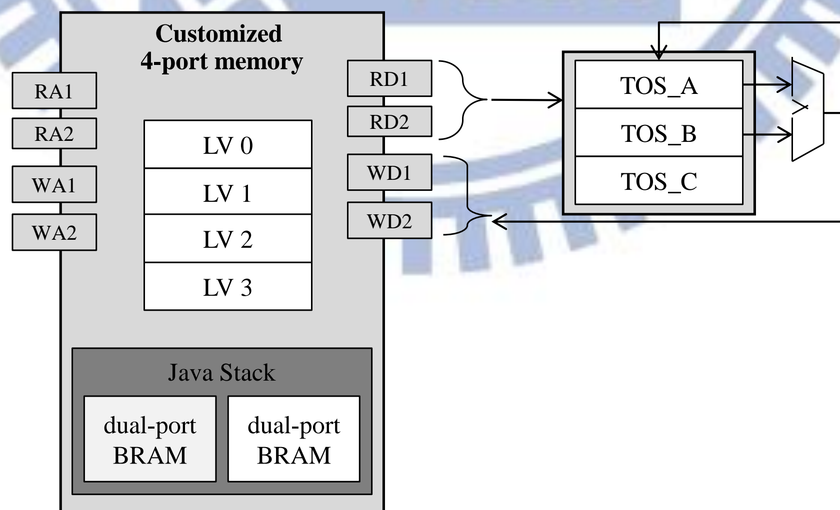


圖 4. Two-level Java stack memory

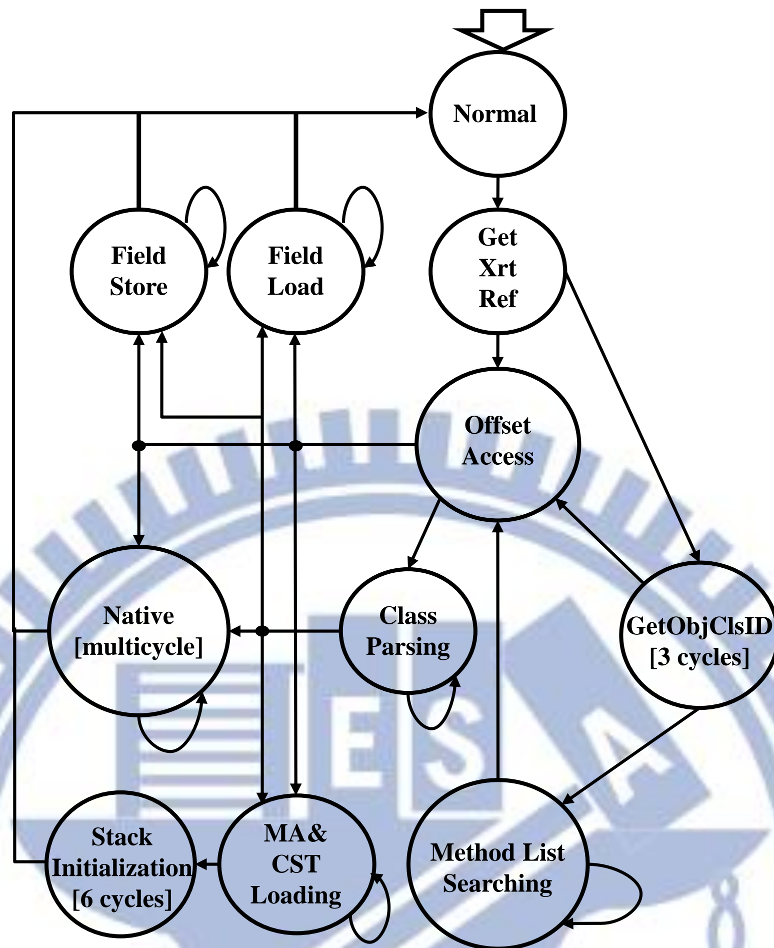


圖 5. Dynamic resolutionru 機制有限狀態機

當 JAIP 在執行 Java 程式時，是將 JAR 檔內所有的檔案先全部載入到記憶體中，但卻並不是一開始就將所有類別全部解析。而是當所要呼叫方法(Method)的類別尚未被解析時，才交由於 RISC 處理器上由系統軟體所實作的類別解析器（Class Parser）來做解析並產生適用於 JAIP 的執行映像檔（Run Time Image），並將解析完畢的映像檔會保留在記憶體中。此外會由類別解析器維護一張 Class Parsing Table[31]，此張表格主要用於紀錄各類別的詳細資訊，例如類別編號、名稱、記憶體位址、繼承與介面資訊、Field Data 與 Method 的參照資料等。由於當 Java 程式於 JAIP 上執行時，Field Data 存取及呼叫方法時都需要經常查找這個資訊，所以類別解析器會將這兩項資料整理並存放於由 On-chip BRAM 所組成的 Cross Reference Table 之中。因此當 JAIP 在執行 Field Data 存取及呼叫方法時，會先啟動 dynamic resolution 機制(如圖五所示)，會去查找 Cross Reference Table 內所需的資訊(如圖 6 所示)。若是存取 field data 會取得 offset 值去計算

存放於 Heap space 的 object data 位址。若是執行呼叫方法，則會取得所要呼叫方法的 Class ID 跟 Method ID。

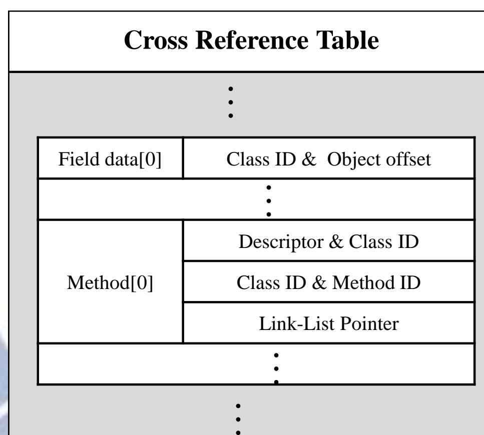


圖 6. Cross Reference Table 類別存放資訊

2.2. 多執行緒相關研究

多執行緒的設計研究，可簡單區分為由軟體或硬體來支援。由軟體所支援的例如 Java 虛擬機器，對於執行緒的管理與運作大多透過 Java 程式的執行直接操作以及透過原生方法來支援，而對於執行緒的相關執行方法都定義於 java.lang.Thread 此類別。但在 JAIP 的設計架構之下，若要仿造 Java 虛擬機器單純由系統軟體來支援管理執行緒，則每一次要對執行緒做操作時，都需透過中斷來交由 RISC 處理器的中斷處理機制來執行，而每一次的中斷發生都需耗費大量的執行時間，以至於使用系統軟體的成本過高。因此若使用單一 JAIP 來支援多執行緒架構以系統軟體來支援的作法並不推薦。

而由處理器來直接支援多執行緒 [33]，以單一處理器來執行管理多執行緒的架構而言，當系統中只有一條 data path，透過切換執行緒來達到似乎多個執行緒同時在執行的現象，並依照時間區段(Time-slice)的選擇可區分為粗粒度交替多執行緒(Block multi-threading)與細粒度交替式多執行緒(Interleaved multi-threading) (如圖 7.b、7.c 所示)。若單處理器上可以處理多條 data path，即可每一個週期從不同的執行緒上取得指令執行[34]，可真正達到執行緒真實並行(如圖 7.d 所示)。使用多處理器所組成的系統，可利用每個處理器去平行執行不同執行緒來提高執行效率(如圖 7.e 所示)。而當由處理器

直接管理多執行緒時，每個執行緒必須保持自己的執行資訊，例如 program counter、運算堆疊等。因此，必須提供一個機制來保存與恢復執行緒狀態。在嵌入式系統上，Java 處理器去設計一個有效率與低成本的多執行緒機制是重要的議題。

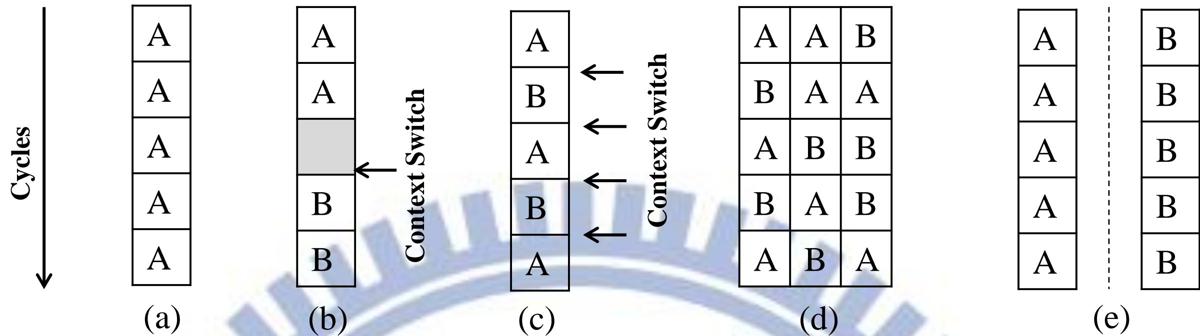


圖 7. 處理器支援多執行緒不同方法；(a) Single-threading；
(b) Block multi-threading； (c) Interleaved multi-threading；
(d) Simultaneous multi-threading with Single-processor； (e) Multi-processors

在 C.-M. Chung 所提出的 Dual-threaded Java Processor [4]的研究中使用存在兩條 data path 的 4-stage pipeline 處理器。呈現了最多可同時執行兩組執行緒的 Simultaneous Multithreading 處理器架構。因此當同時執行兩組執行緒時，處理器必須維持各執行緒的執行資訊。而在執行緒的運算堆疊的運作上，提出了兩種不同的做法，Logical dual stack cache 與 Physical dual stack cache。Logical dual stack cache(如圖 8.a 所示)是由一組 128 個暫存器所組成的集合，並在系統中保持四個暫存器，分別記錄最高、最低位址跟兩個執行所使用的堆疊指標。而當執行緒在使用堆疊做運算時，其一執行緒從 0 的位址開始存起而另一執行緒則是從此集合的中間開始存起。然而此種作法可能會造成兩個執行緒在堆疊的運作上衝突卻可提高 stack cache 的使用率。Physical dual stack cache(如圖 8.b 所示)則是由兩組獨立 64 個暫存器所組成的集合。執行緒則各擁有自己的運算堆疊。此種做法可避免兩執行緒的堆疊互相衝突卻可能降低 stack cache 的使用率。而在這種設計架構之下，系統最多只擁有兩組運算堆疊，造成應用程式所能運行與管理的執行緒被限制為兩組。

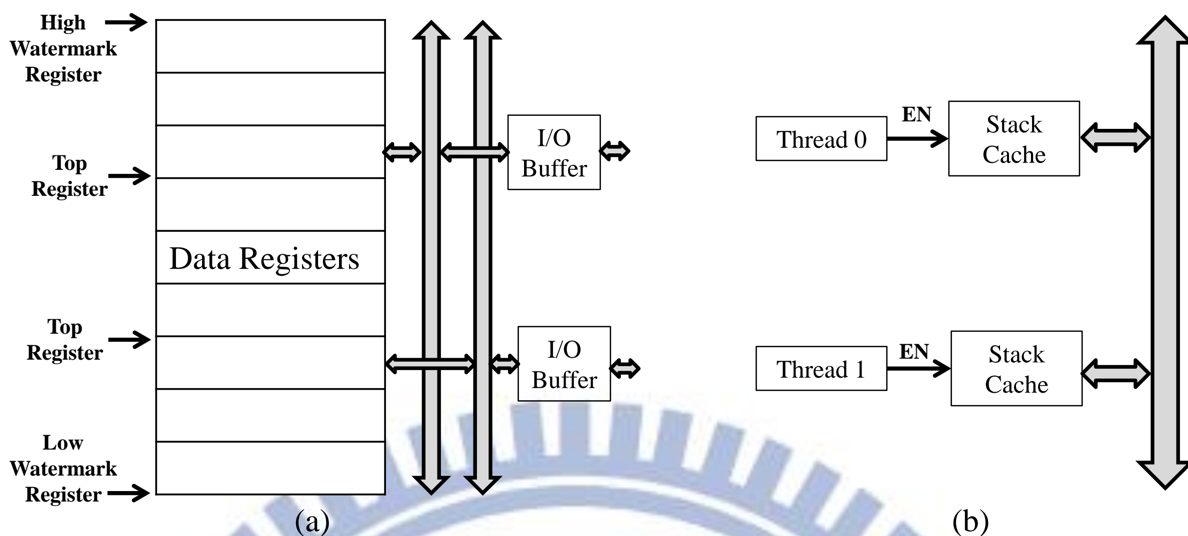


圖 8. Dual stack cache；(a) Logical dual stack cache；(b) Physical dual stack cache

若要提高系統中可執行的執行緒數量，就必須要增加管理執行緒的成本，用以記錄各執行緒的執行資訊。Java 處理器 Komodo[7]的設計架構下，可支援系統中存在最多四組執行緒，在設計中使用四組暫存器集合作為運算堆疊來執行四組執行緒的運作(如圖 9 所示)。然而在單一處理器設計中往往為了要擴展可執行的執行緒數量，而在設計裡加入多組暫存器集合與堆疊，使得每個執行緒都保有自己的執行資訊，令執行緒可以輕易的交互執行，但卻大幅增加在執行緒的管理成本。

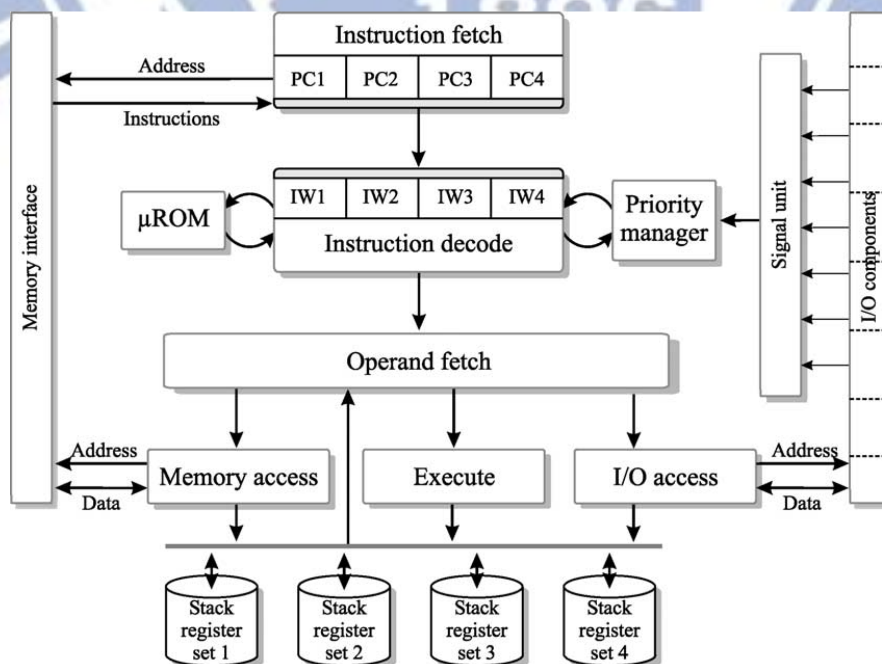


圖 9. The Komodo processor core.

而也有 Java 處理器的對於多執行緒的機制設計是採用多核心的架構來支援。如 Schoberl [35]提出的 Java Optimized Processor(JOP)即採用多核心執行環境的架構[6][7]。使用多組 Java Optimized Processor 加上對於共用記憶體的仲裁器與管理執行緒的機制(如圖 10 所示)。而每一個核心上都有一組 stack cache 與所要執行方法的 method cache。並統一管理所有執行緒並指派給處理核心去執行。然而藉由使用多個核心，使得每個核心能夠真正在同一週期平行執行不同的執行緒，藉此提高整體執行效率。然而在這樣的設計架構之下，Java 處理器執行所需的全部資料皆放置於共用的記憶體上，因此在存取共用記憶體將會成為多處理器的執行瓶頸。

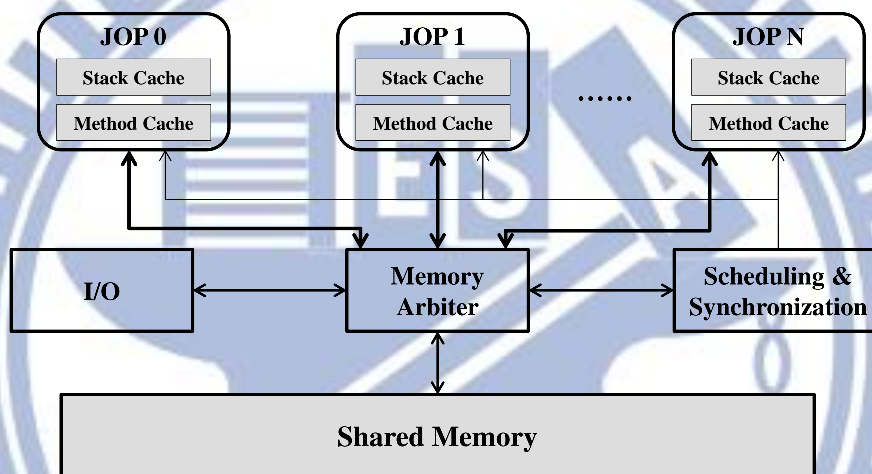


圖 10. The JOP chip multi-processor system

在設計 JAIP 的多執行緒機制時，首先規劃單一個核心內支援多執行緒的運作，然而在之前 JAIP 的設計下核心內同一時間只能處裡一組 data path，而若要支援多組執行緒能同時執行，則在核心內所要耗費的成本過高，因為每一個執行緒都必須要有自己的控制訊號以及運算堆疊。因此我們在單一核心內採用在一時間內只有一組執行緒能執行，並藉由執行緒的切換來達到支援多執行緒的功能，詳細的設計細節於第三章將做解說。但在單一核心內支援多執行緒並不能使多執行緒能增加效能的特點展現出來，因此在第四章裡我們呈現出由多個 JAIP 處理核心所組成多核心的系統架構，藉以讓執行緒能真正平行執行，使其能提高在執行時的效率。

第三章 Temporal Multithreading 系統架構

在本章節中，會呈現我們以原先設計之 Java Accelerator IP (JAIP)架構為基礎，在單一 JAIP 內設計能夠讓多組執行緒能交互執行之機制。因此對原先架構做了修改(如圖 11 所示)，加入 Thread Manager Unit 來管理執行緒的執行資訊與運作，包含控制執行緒的切換與排班，並使用一組暫存器集合來記錄個執行緒的執行資訊。而且為了將執行緒管理成本降低並使 JAIP 能支援多執行緒機制，在原先的設計中提供了一組 Four-port stack memory，我們再加入了一組 stack memory 組成 Ping-pong Java stack。並將各執行緒所運作的堆疊在 DDR-SDRAM 中存放一組備份，使得在切換執行緒前可以先從 DDR-SDRAM 中將準備執行的執行緒堆疊放置於 Ping-pong Java stack 中未使用的一組，等到可切換執行緒時即可直接切換系統所使用的運算堆疊。而不用在 JAIP 上記錄所有執行緒運算堆疊資料，如此便能大幅降低管理成本。

此外，對於執行緒的相關操作完全交由 JAIP 來處理，並不用透過 RISC 處理器來協助。因此我們修改系統軟體的類別解析器，針對相關執行緒類別做特別的處理，使得所產生的解析資料當 JAIP 在操作執行緒相關運作時，完全由 JAIP 來支援。相關設計細節會在之後小節中說明。

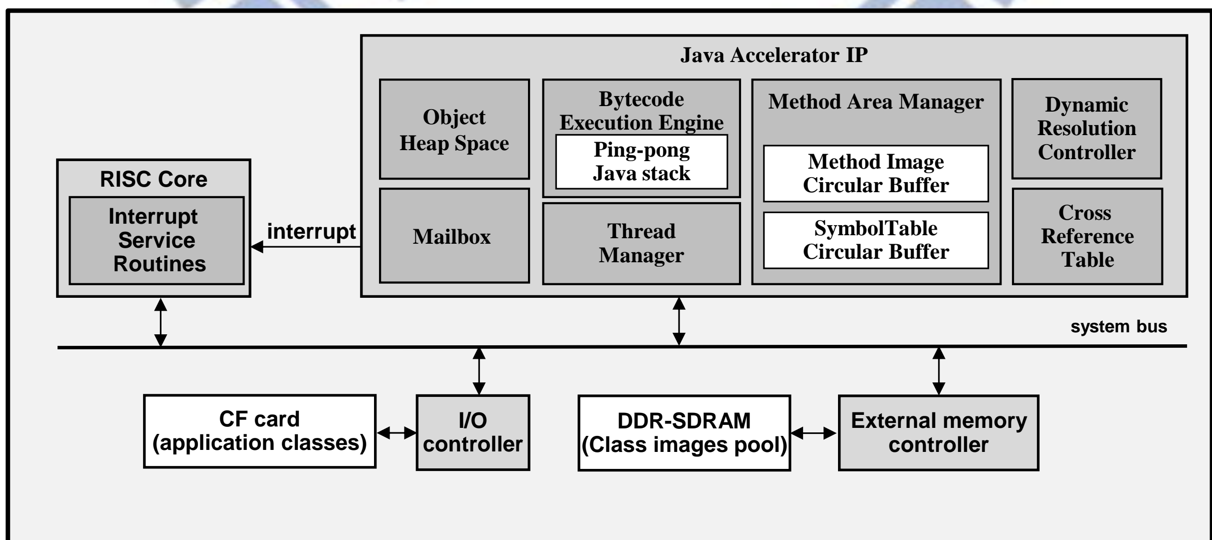


圖 11. 多執行緒異質雙核心系統架構

3.1. Thread Manager Unit

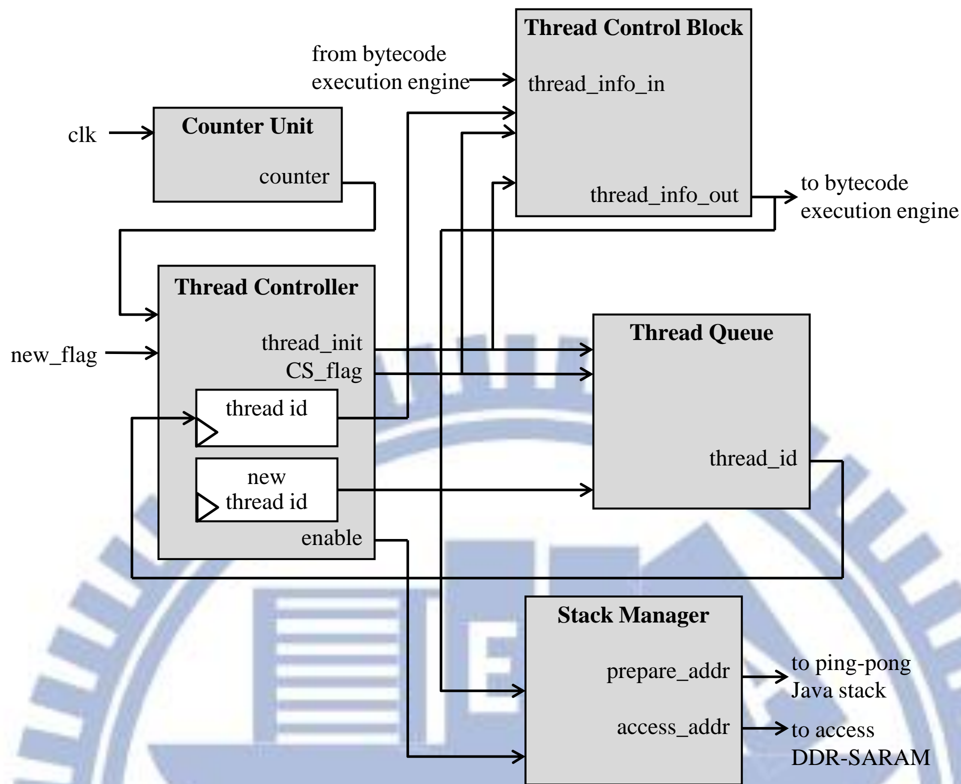


圖 12. Thread Manager Unit

為了使 JAIP 能有效支援多執行緒機制，我們提出了 Thread Manager Unit 的架構(如圖 12 所示)。這個提出的架構下，由幾個主要元件所組成，負責協助規劃執行緒執行順序的執行緒佇列(Thread Queue)。紀錄各執行緒之執行資訊的 Thread Control Block。管理各執行緒所使用堆疊從 Ping-pong Java stack 移入或移出的 Stack Manager。由於我們是採用靜態的粗粒度交替多執行緒(Block multi-threading)的技術，所以我們加入計算執行緒所執行時間的計數器(Counter Unit)。在此架構下，當有執行緒呼叫 java.lang.thread 類別的 start()方法時，藉由 dynamic resolution 機制的查找發現是新增執行緒的動作，會送出一訊號 new_flag 通知 Thread Manager Unit。Thread controller 會給此新的執行緒一個專屬編號，並發出訊號 thread_init 告知 Thread Control Block 將新執行緒的資訊初始化以及將新執行緒放入執行緒佇列中排班並等待取得執行權。在我們的設計中是採用粗粒度交替多執行緒，因此正在執行的執行緒執行時間已到一個時間片段時，JAIP 會令執行緒停止執行，同時存入此執行緒的執行資訊到 Thread Control Block 中，並將預備執行的執

行緒的執行資訊傳入 Bytecode Execution Engine 之中。在下一個週期就可執行不同執行緒的程式。由於 Thread Control Block 是利用暫存器實作，因此將執行緒資料放入 Thread Control Block 以及切換執行緒的動作也只需一個週期即可。此做法可以降低執行緒在切換時的成本。而在執行緒被切換後，Thread Controller 開始執行上一執行緒所使用堆疊資料備份與準備下一個被選定執行的執行緒堆疊準備的動作，啟動 Stack Manager 並將所需資訊傳入，包含執行緒堆疊在 DDR-SDRAM 中的位置以及堆疊的長度。

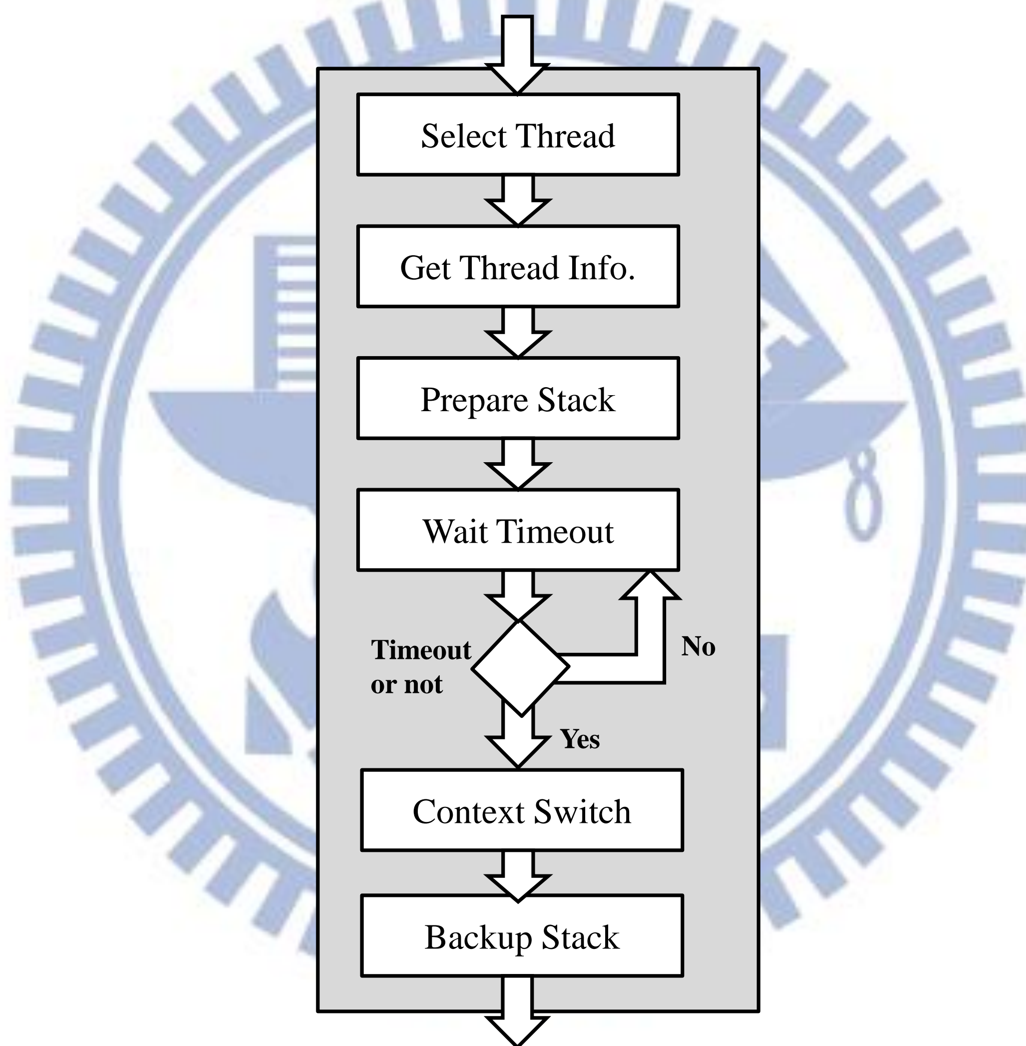


圖 13. 切換執行緒流程

當系統中存在一個以上的執行緒時，在執行緒就必須輪流切換執行，圖 13 為切換執行緒的流程，一開始必須先從執行緒佇列中取得下一個可取得執行權的執行緒編號，並利用此編號到 Thread Control Block 中查找取得此執行緒的執行資訊，接著從 DDR-SDRAM 中讀取其使用的運算堆疊內容到存放至 Ping-pong Java stack 上非執行中

的堆疊，而等到執行中的執行緒執行時間到達一個時間片段，執行 context switch 即將當前的執行緒資訊存回 Thread Control Block 之中，並將能取得執行權的執行緒資訊更新目前系統中執行狀態，而當此切換執行緒且開始執行時，便將上一個執行緒所使用的運算堆疊內容存到 DDR-SDRAM 中進行備份。

3.1.1. 執行緒新增與終結

當 Java 程式要新增執行緒時，必須透過呼叫 java.lang.Thread 類別的 start()方法。依照原本 Java 標準類別庫，此時會呼叫原生方法並交由作業系統來管理執行緒。而在我們 JAIP 的多執行緒架構設計下，對於執行緒的管理完全由 JAIP 負責而不用 RISC 處理器的系統軟體來支援。因此當呼叫 start()方法時，如同一般呼叫方法會啟動 dynamic resolution 機制(如圖 14 所示)進行查找方法資訊，並且為了在硬體上就能進行新增執行緒的運作，因此在查找呼叫方法的 dynamic resolution 機制中加入新的執行狀態 NewThread。而在查找的過程中到了 OffsetAccess 的狀態時會取得執行緒所要執行方法的類別編號(Class ID)與方法編號(Method ID)。並且在查找的過程中會判斷此次的呼叫方法是否進行新增執行緒的運作。若是便會進入 NewThread 狀態，在此狀態時 Dynamic Resolution Unit 會把找到的方法資訊(執行的類別編號與方法編號)與新增執行緒的信號發送給 Thread Manager Unit，通知要新增一組執行緒並且進行初始化，此時 Thread Controller 接收到新增執行緒的訊號後，會將目前系統中執行緒的數量加一，並給予新執行緒一個獨立的 16 bits 編號，作為執行緒在系統中識別的名稱。然後會在 Thread Control Block 找到一組空的欄位且初始化這些欄位，並填入執行緒編號與所要執行的方法資訊，再將此執行緒編號加到執行緒佇列中列入執行排班機制。此時便完成了一組執行緒的新增。

而當執行緒執行完畢要終結時，會發出訊號通知 Thread Manager Unit 此執行緒已經終止(Terminate)。Thread Controller 收到執行緒終止的訊號後，便會直接切換執行權到所選出下一個等待執行的執行緒，並且將終止的執行緒資訊直接移除，將系統中執行緒數

量減一。此執行緒將不會再被放入執行緒佇列而有取得執行權的資格。而且會刪除此終止執行緒在 Thread Control Block 內的存在資訊。

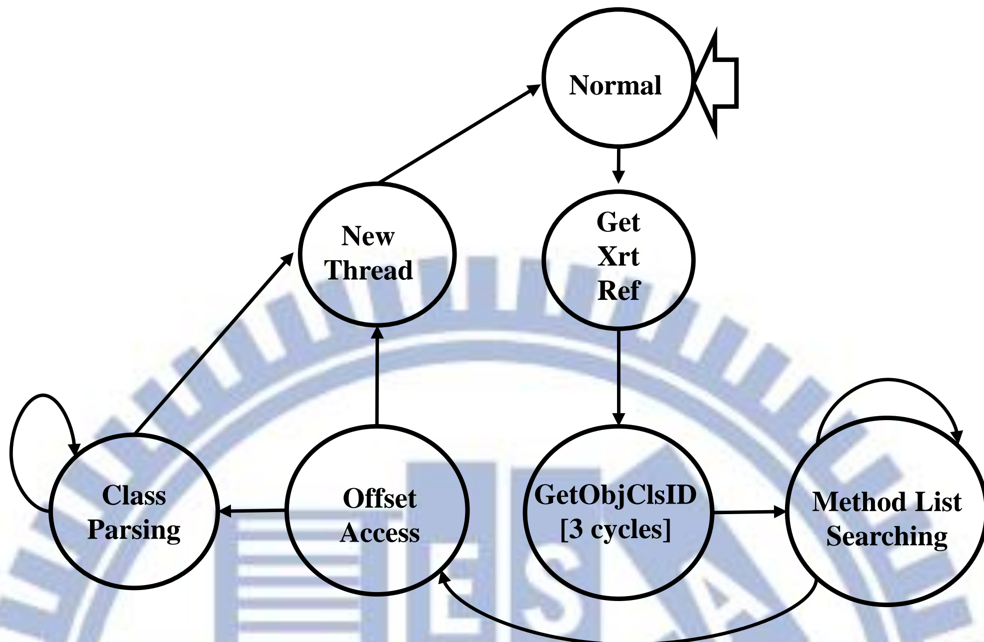


圖 14. 新增執行緒之 dynamic resolution 機制

3.1.2. Thread Controller

Thread Controller 主要接收所有傳入 Thread Manager Unit 對於執行緒運作的相關訊號(包含新增執行緒、執行緒終止等)。並產生相對應的訊號來控制執行緒佇列和 Thread Control Block。也負責管理目前系統中執行緒的排班與選擇以及當新執行緒產生時所需要的初始化資訊。而當執行緒需要切換執行權時，Thread Controller 也會產生相關訊號去通知 JAIP 各元件將執行資訊切換到下一個取得執行權的執行緒資訊(例如所執行的方法、堆疊的指標等)。在圖 15 表示的為 Thread Controller 在系統中對於執行緒管理時的狀態。由五個狀態所組成，Idle、CheckTimeout、ContextSwitch、BackupPreviousStack 以及 PrepareNextThread。由這五個狀態來控制目前系統中對於執行緒的操作。每個狀態詳細的設計於下段解說。

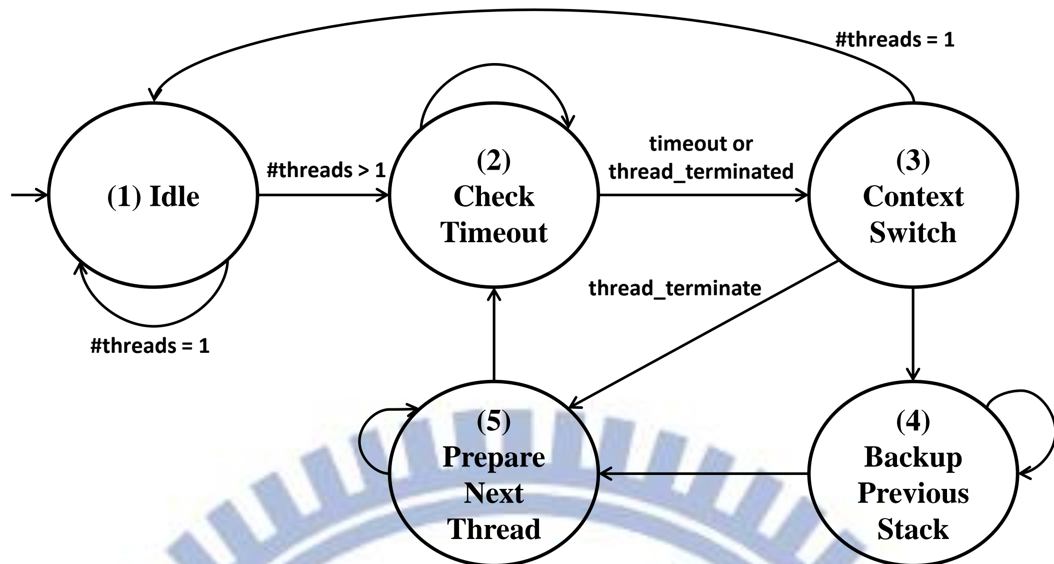


圖 15. Thread Controller 有限狀態機

在(1)Idle 狀態時，當系統中執行的只有一組執行緒時，將一直維持在此狀態。此時 Thread Manager Unit 並不會執行任何動作，因此不會對執行的程式造成任何的額外負擔。直到有新的執行緒被產生使得系統中執行緒數量大於一組，此時即會切換到下一個狀態，並在切換的時候取得新執行緒被初始化的資訊，用以等待執行權被切換。

在(2)CheckTimeout 此狀態時，Thread Controller 會檢查當時執行緒所已經執行的時間，如果執行時間到達我們所設的一個時間片段，則會停止從 Method Area Manager 中抓取此執行緒所要執行的位元組碼到 Instruction Buffer 裡。並將 Bytecode Execution Engineg 停止從 Translate Stage 傳送 j_code Information 以及 Fetch Stage 傳送所要執行的一對 j_code 到 Decode Stage。但是在這個時間點上，Decode Stage 以及 Execute Stage 尚存有正在解碼或執行的 j_code。因此為了讓執行緒在切換上更為簡便以及簡化所要記錄資訊的電路邏輯，在執行到達一個時間片段時，並不會抹除目前存在 Decode Stage 以及 Execute Stage 所要執行的 j_code，而是等此次的執行完畢之後再做執行緒的切換運作。如此一來便可降低管理執行緒之成本以及使執行緒切換的操作能更為單純。而若一執行緒在還沒執行到一個時間片段時已將所負責之程式執行完畢，並發出執行緒終止的訊號，則 Thread Controller 會直接停止所有執行，而進行執行緒切換的動作。

在(3)ContextSwitch 此狀態時，Thread Controller 會發出 context switch 的訊號通知並會將先前從 Thread Control Block 中取出的執行緒資訊送到 JAIP 相關元件去進行執行緒的切換運作。將可取得執行權的執行緒所要運作方法的類別編號與方法編號送到 Method Area Manager，檢查所要執行的方法是否已經載入 JAIP 之中。以及切換在 Ping-pong Java stack 中所使用的堆疊並調整堆疊的指標(Stack pointer 與 Variable pointer)與狀態。並且將系統在執行的 program counter 也切換成此執行緒上一次所執行到的 program counter。而被切換執行權的執行緒當下的執行狀態將會被回存到屬於它在 Thread Control Block 中的欄位。由於在進行執行緒切換前，Thread Controller 就會先將下一個將取得執行權的執行緒資訊從 Thread Control Block 中拿出，因此在執行緒切換的時候可以在一個週期就切換 JAIP 相關重要的資訊，並且初始化所有的暫存器資訊，下一個週期就可以開始執行新的執行緒。

而當所新增的執行緒尚未被執行過時，其堆疊的狀態尚未被初始化，也就是說並不清楚對於此執行方法的區域變數(Local variable)數量。而此資訊是當系統軟體利用類別解析器解析所執行類別時，整理並放置在執行映像檔內。通常是利用在 dynamic resolution 機制中在呼叫方法時對於堆疊做初始化的運作(如圖 5 所示)。因此我們對 dynamic resolution 的運作機制做出修改(如圖 16 所示)，使得切換執行緒的時候若執行緒是第一次取得執行權執行時會直接啟動 dynamic resolution 機制中的堆疊初始化的流程，藉此對於新執行緒所使用的運算堆疊能做到初始化的運作。

而在執行續進行切換並進入此狀態時，Thread Controller 會從執行緒佇列中取出下一個能取得執行權的執行緒編號，並利用此編號到 Thread Control Block 中取得此執行緒的資訊等待下一次的切換，並將被切換的執行緒編號放到執行緒佇列中，重新排班以等待取得執行權的機會。而若是由於被切換的執行緒原因已終止執行的狀況，則不會進入(4) BackupPreviousStack 此狀態。因為此執行緒已終止，所以系統中不需要存在此執行緒的相關資訊，可以直接開始準備下一個可取得執行權的執行緒運算堆疊。

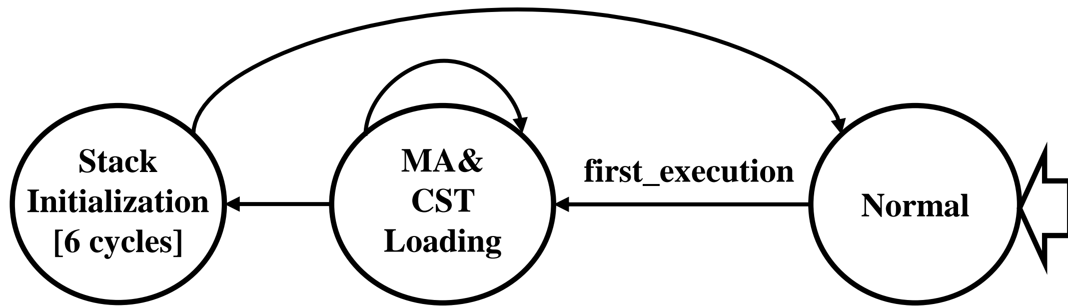


圖 16. 執行緒首次執行之 dynamic resolution 機制

在(4) BackupPreviousStack 此狀態時，由於在執行緒切換時，JAIP 會將 Ping-pong Java stack 中所使用的運算堆疊切換到另外一組堆疊。因此在上一個執行的執行緒所操作之運算堆疊資料還留在 Ping-pong Java stack 中。而在為了降低在處理器上管理執行緒所耗費之成本，我們將各執行緒所使用的運算堆疊資料在 DDR-SDRAM 中存放一組備份。因此我們在此狀態下會將被交出執行權的執行緒所用堆疊複製到 DDR-SDRAM 中。但由於 JAIP 的堆疊設計是採用 Two-level Java stack memory(如圖 4 所示)，主要是由兩塊 Dual-Port On-chip BRAM 所組成。所以若我們完整的將整個堆疊複製到 DDR-SDRAM 上，如此一來在傳輸的作業時間太長，對於系統的成本過高。因此我們採用執行緒的堆疊資訊 Stack pointer 來做為傳輸長度。因為在執行時真正在堆疊上的資料也只放置到 Stack pointer 這個位址而已。所以我們只複製有使用到的堆疊空間，藉以降低在傳輸堆疊資料時對於系統運作的影響。而若執行緒已經終止執行的話，Thread Controller 並不會來到這個狀態，可縮短在準備的作業時間。

而當上一個執行緒的運算堆疊已完整的複製到 DDR-SDRAM 中或是執行緒已終止，即會進入(5)PrepareNextThread 的狀態。在此狀態時會開始將此執行緒原本存放在 DDR-SDRAM 中的堆疊資料備份，將其讀進 JAIP 並存放於 Ping-pong Java stack 中目前在 JAIP 執行所使用的另一組執行堆疊上。與備份時相同，為了減短傳輸的作業時間，以降低對於正常執行的影響。我們在之前的狀態就已取出此執行緒的資訊。因此我們也使用此執行緒上一次執行時的 Stack pointer 來限制傳輸的長度。經由將執行緒的堆疊在執行緒切換前準備好，而當要進行 context switch 時，JAIP 只需改變所使用的運算堆疊即可。也不用讓每個執行緒都在 JAIP 都上保留一組堆疊。

3.1.3. Thread Control Block

在 Thread Manager Unit 的架構中，設計了一組由暫存器所組成 Thread Control Block(如圖 17 所示)來存放各執行緒的執行資訊。由於在目前 JAIP 多執行緒設計架構上最多能支援到 16 組執行緒同時存在於系統之中，因此 Thread Control Block 目前可同時存放 16 組執行緒的資訊，而每一組資訊皆是由 8 個 32 bits 的暫存器所組成的集合。而每一個執行緒我們所記錄的執行資訊分別為：

(1) 執行緒編號

(2) 執行緒所執行的類別編號與方法編號

(3) 堆疊的使用情形，包括 Variable pointer 和 Stack pointer

(4) 執行緒之前執行狀態，包括 Java program counter 與執行方法的區域變數個數

(5) 堆疊最上方三個元素

(6) 執行緒的 object reference

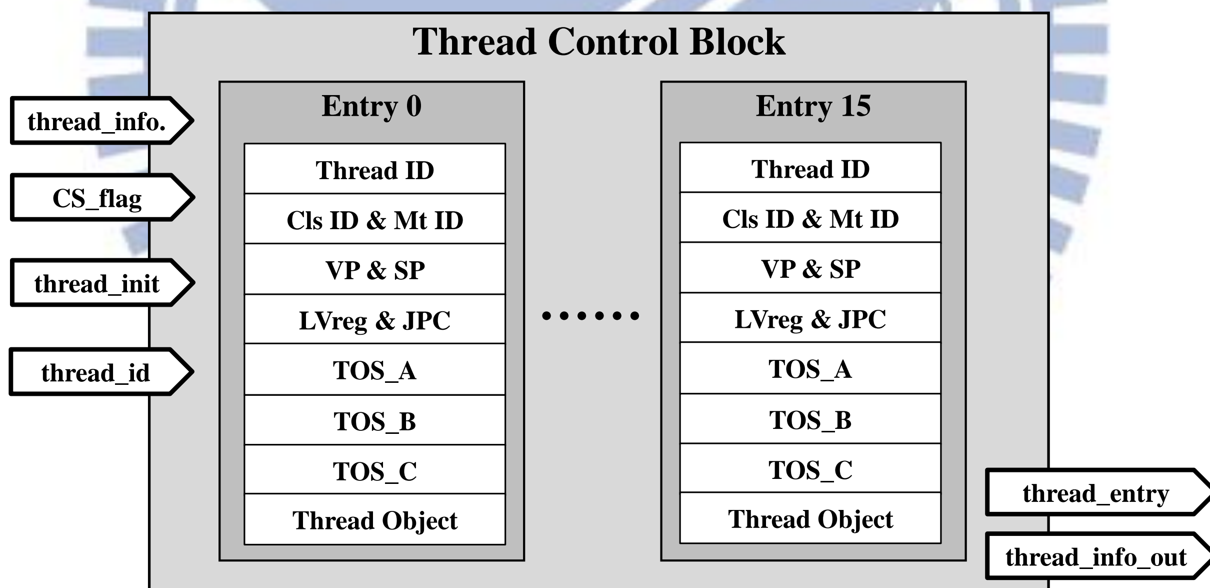


圖 17. Thread Control Block

然而選擇存放這些資訊的原因是透過存放執行緒的編號，可讓 Thread Controller 可直接透過執行緒的編號到 Thread Control Block 取得執行緒資訊。記錄類別與方法編號、Variable pointer、 Stack pointer 以及 Java program counter 這些訊息是 JAIP 在執行程式時的重要資訊。而在執行過程中執行方法的區域變數個數是藉由每次呼叫方法時，在

Dynamic Resolution Unit 運作時對於堆疊所做的操作。因為在原先 JAIP 中所使用的 Two-level Java stack memory(如圖 4 所示)，使用了四個暫存器去放置程式執行頻率較高的 local variable 0 到 3[31]，所以需要此項資訊來判斷當前所執行的方法對於這四個暫存器的運用情形。而在 Two-level Java stack memory 中第一層使用了三個暫存器來放置堆疊最上方的三筆資料，藉以提高 JAIP 的執行效率。因此為了減少在執行緒切換時造成的成本耗費，因此我們選擇直接將這三個暫存器放置於 Thread Control Block 中，使得在切換時只要直接替換 JAIP 所使用的這三個暫存器即可，而不用再從第二層的 stack memory 中取出來更新這三個暫存器。而當一執行緒要被新增之前，必定會先產生一個執行緒的物件參照資料(object reference)，用以查找此物件放置於 Heap 上的物件資料。然而雖然在新增執行緒的時候是使用 java.lang.Thread 類別的呼叫 start()方法，但在新增執行緒後，此 object reference 將會從堆疊上被 pop 掉，而不會存在於目前執行的執行緒運算堆疊中。因此當新執行緒要開始執行時，若它要取得此物件在 Heap 上的資料時，會因為沒有在它的堆疊中放置 object reference 而找不到資料。因此我們在 dynamic resolution 的呼叫方法機制時,若發現此次執行是為新增執行緒,則會將此執行緒的 object reference 連同執行方法資訊傳到 Thread Manager Unit。

而在執行緒被新增、切換與終止時，皆需要對 Thread Control Block 做修改的運作。當執行緒新增時,會由 Dynamic Resolution Unit 送出新增執行緒的訊號到 Thread Manager Unit，且伴隨著此執行緒的初始化資料。此時 Thread Controller 會給予新執行緒獨立的編號，並先在 Thread Control Block 中找尋在 16 個欄位中尚未被填入執行緒資訊的空白欄位，將此執行緒的編號與由 Dynamic Resolution Unit 所送來的資訊填入，且將其餘的資訊做初始化的動作。而在執行緒進行切換時,會將所要記錄的資訊送到 Thread Manager Unit，並依照當下執行的執行緒編號找到其欄位位址，並更新此欄位的所有資訊。而當執行緒已執行終止時，也是利用其執行緒編號來找尋所記錄的欄位，並將此欄位的執行緒編號刪除，使其欄位可以存放新執行緒之資訊。

由於我們是利用暫存器來組成 Thread Control Block，因此在取出與放入的操作皆可以在一個週期內完成，以提高在切換時的執行效率，若是利用 On-chip BRAM 來組成這

張表格，則必須要耗費四個週期才能將所有資訊寫入或讀出。而由於每個執行緒的編號皆不相同，因此我們只需拿執行緒的編號即可在 Thread Control Block 查詢資訊。目前在 JAIP 多核心系統架構下最多能支援到 16 組執行緒同時存在於系統中，而每一執行緒所記錄的資訊皆是由 8 個 32 bits 暫存器所組成。因此在此設計架構中，Thread Control Block 是使用的 512 bytes 的硬體成本。

3.1.4. 執行緒佇列

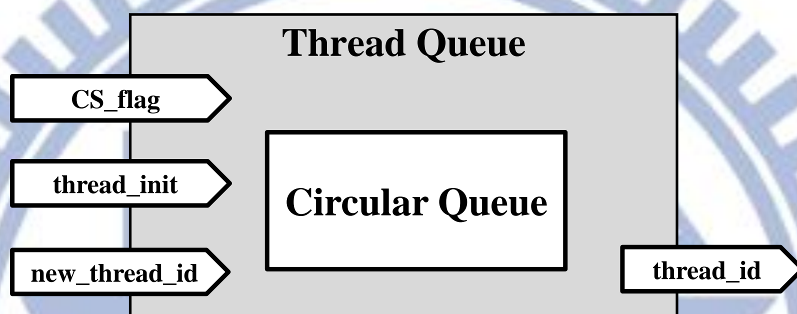


圖 18. 執行緒佇列

當系統中存在超過一組的執行緒時，必須要有個機制能對所有執行緒做排班，以規劃各執行緒能取得執行權的先後順序。而我們採用的排班演算法是循環分時排程(Round-Robin)，因此我們將執行時間切成一個個時間片段(Time-slice)，使得所有執行緒能夠公平地輪流執行。為此我們在 Thread Manager Unit 中設計了一組執行緒序列(Thread Queue)的元件(如圖 18 所示)，而在這個元件中存在著一組由暫存器所組成的環狀佇列(Circular Queue)(如圖 19.a 所示)來輔助排班機制。並在此機制中保持兩個指標，分別為 Ready pointer(RP)與 Tail pointer(TP)。在環狀佇列中 Ready pointer 指向的欄位內容代表下個可取得執行權的執行緒編號，而 Ready pointer 所指的前一欄位則是目前正在執行的執行緒編號。環狀佇列中 Tail Pointer 指向的欄位代表當有新的執行緒產生時所放置的欄位，或者是當執行緒被切換時需要重新排班所放置的欄位。

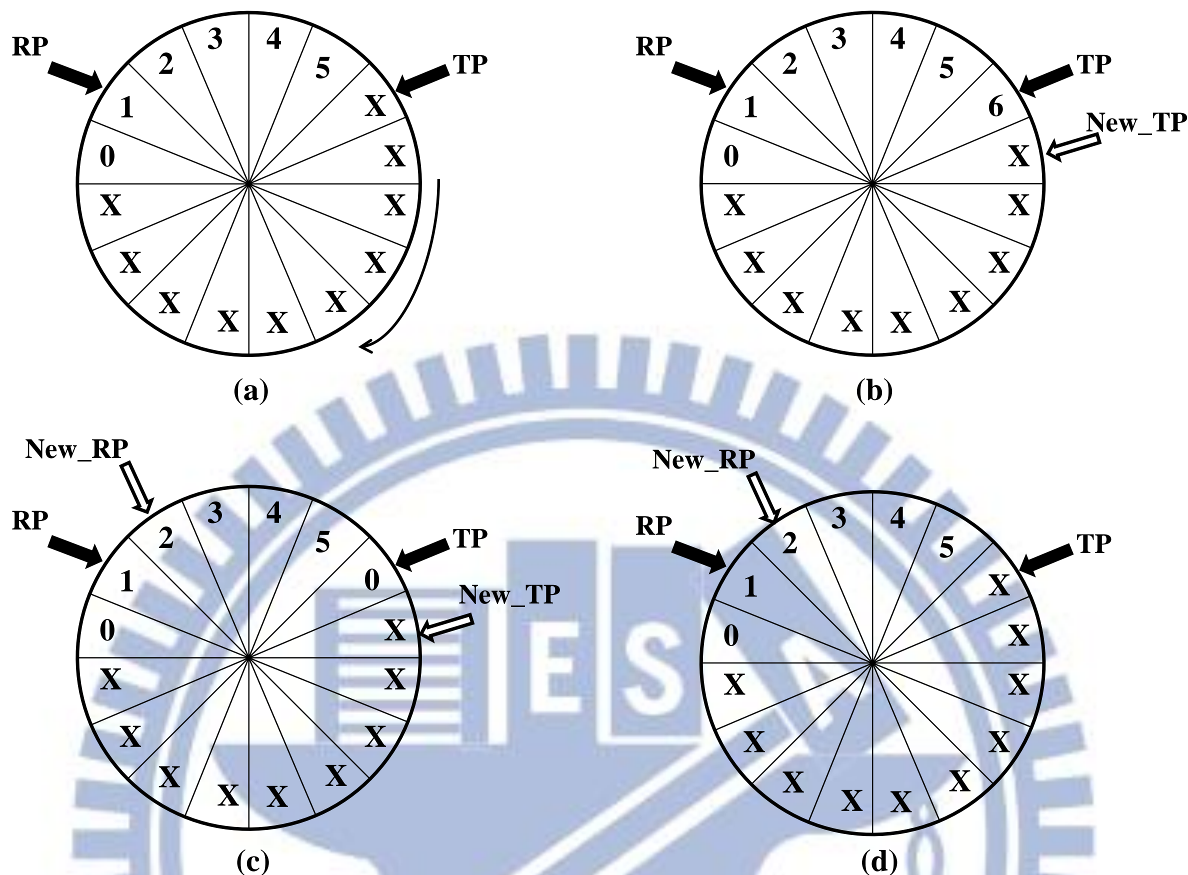


圖 19. 環狀佇列運作示意圖；(a) 執行緒佇列； (b) 新增執行緒；
(c) 執行緒切換； (d) 執行緒終止

環狀佇列的相關操作以圖 19 為例，當有一新執行緒被產生時(如圖 19.b 所示)，Thread Controller 會給予新執行緒獨立編號，在此例子編號為 6。並且會將新增執行緒的訊號與新執行緒編號送到執行緒序列中，當執行緒序列收到新增訊號時，會將新執行緒的編號擺放至 Tail pointer 所指向的欄位，等待取得執行權，並且讓 Tail pointer 加一往前指一個欄位。若是當下執行時間到達一個時間片段時，進行執行緒的切換運作(如圖 19.c 所示)，在此例子中被切換執行權的執行緒編號為 0。此時會皆被切換的執行緒放到 Tail pointer 所指向的欄位，並且讓 Tail pointer 與 Ready pointer 都分別加一往前指一個欄位。此時代表在這次切換後所能執行的執行緒編號為 1，而在下一次切換時所能取得執行權的是編號為 2 的執行緒。當目前在執行的執行緒以終止其執行時(如圖 19.d 所示)，在此例子中所執行終止的執行緒編號為 0。由於此執行緒已經終止，因此並不需要將此執行緒放入

排班規劃中，而是直接進行執行緒的切換。因此會直接讓 Ready pointer 加一往前指一個欄位。而使得執行緒編號等於 1 的執行緒能直接取得執行權。

在此執行緒序列的規劃下，使得 Thread Controller 能輕易地取得下一個可取得執行權的執行緒編號(Ready pointer 所指欄位)，以及能較為容易地完成執行緒排班的運作(將需要放入排班機制的執行緒編號放置於 Tail pointer 所指欄位)。由於目前在 JAIP 多核心系統架構下最多能支援到 16 組執行緒同時存在於系統中，而目前執行緒編號是由 16 bits 的長度所組成。因此在此設計架構中，環狀佇列是使用的 32 bytes 的硬體成本。

3.1.5. Stack Manager

在多執行緒的環境之下，每個執行緒都應記錄自己的執行資訊。然而若將執行緒所使用的堆疊都存放於處理器上，則在資源有限的嵌入式系統設計上有過多的成本。因此在前文中有提到，我們將各執行緒所使用的運算堆疊資料於 DDR-SDRAM 中存有一備份。使得在管理執行緒的資料上成本可以大幅降低。為此我們在 Thread Manager Unit 中設計 Stack Manager 機制(如圖 20 所示)，用以將執行緒所使用的堆疊從 DDR-SDRAM 移入或移出。當 Thread Controller 的有限狀態機(如圖 15 所示)執行到 BackupPreviousStack 以及 PrepareNextThread 這兩個狀態時，表示要對 Ping-pong Java stack 中非執行中的一組堆疊做備份或移入的動作。此時 Thread Controller 會啟動 Stack Manager 機制，並將執行緒的訊息傳入(包含執行緒的欄位編號以及 Stack pointer)。

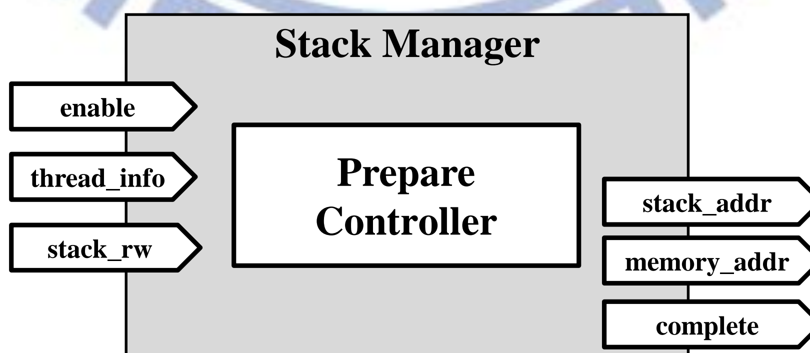


圖 20. Stack Manager

圖 21 為 Stack Manager 機制所使用的有限狀態機。當 JAIP 在一般執行時，並不需要對堆疊做任何的動作，因此會一直維持在 Idle 的狀態。而在執行緒進行切換後，Thread Controller 要對堆疊做備份與準備下一個可取得執行權的執行緒堆疊。因此會把執行緒堆疊備份於 DDR-SDRAM 的位置與所要傳輸的長度(Stack pointer)傳入。然而會在 CheckAccessRequest 狀態檢查是否已經滿足傳輸的長度，若堆疊的移入或移出尚未完成，則會進入 SDRAMAccessing 狀態再對於 DDR-SDRAM 提出存取的要求。這兩個狀態的循環直到執行緒的堆疊已經完整的移入或移出。

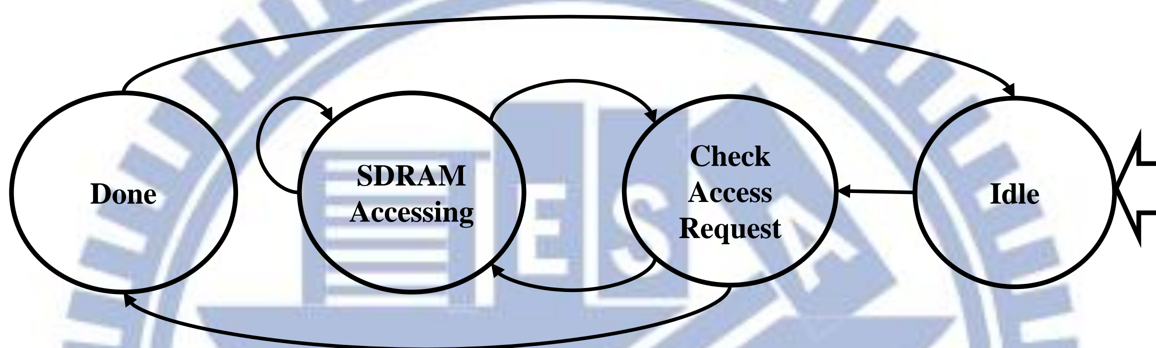


圖 21. Stack Manager 有限狀態機

而為了減少在傳輸時所耗費的成本，在原本 JAIP 的設計中對於 DDR-SDRAM 存取皆是採用 PLB Master Single 的方法，每一次對於 DDR-SDRAM 存取只有一個 word，如此的做法在傳輸大量資料的時候，每一次的要求就需要有固定的成本耗費在 Handshaking 上，卻每次都只取得單一個 word 的資料長度而已。因此為了降低在堆疊傳輸上所耗費的成本，我們對於 DDR-SDRAM 存取改為使用 PLB Master Burst 的方法。使得每一次對 DDR-SDRAM 發出存取的要求，只要進行一次的 Handshaking 即可取回大於一個 word 的資料長度。然而若一次取回過於大量的資料長度，在傳輸的過程中反而會造成嚴重的延誤，這樣的延誤可能會比 PLB Master Single 所耗費的成本還要來的高。因此如何選擇一個 Burst 要求的資料長度是為重點。表 1 與表 2 所呈現的數據是實際測試對於 DDR-SDRAM 存取利用不同長度的 Burst 模式要求。依照表格內數據的結果來看，在使用 PLB Master Burst 做存取要求時，並不是一次取回的資料量越大越好，而是大到某一個階段時，即展現不出使用 Burst 模式來存取所得到的好處。因此依照此數據，我

們在對 DDR-SDRAM 做存取時若使用 Burst 模式，則固定每次傳輸的資料長度皆為 16 個 words。以追求最大的傳輸效率降低堆疊移入移出對執行的影響。

表 1. Number of PLB Master Burst Read

PLB Master Burst Read						
Data length(words)	**1**	**2**	**4**	**8**	**16**	**32**
# clock cycles	36	37	39	43	51	108

表 2. Number of PLB Master Burst Write

PLB Master Burst Write						
Data length(words)	**1**	**2**	**4**	**8**	**16**	**32**
# clock cycles	17	16	18	22	30	102

3.1.6. 執行緒同步

在多執行緒機制下，可能會發生不同執行緒對於同一個共用資料做存取。由於 JAIP 的 object 資料是存放於 JAIP 上的 Heap Space，所有執行緒皆共用此空間。因此，不會發生資料的不一致性。但卻會發生競爭條件(Race Condition)的問題。為此 Java 語言提供兩種執行緒同步機制:(1)同步區塊，在程式中，*synchronized* 此關鍵字定義一個同步區塊。並使用 *monitorenter* 與 *monitorexit* 這兩個位元組碼，利用所放入的 object 來做為物件鎖(object lock)，鎖住要進入相同同步區塊的其他執行緒。(2)同步方法，利用 *synchronized* 此關鍵字修飾的方法，當被呼叫時，使用被呼叫方法的 object 做為鎖，鎖住要呼叫此 object 所有同步方法的執行緒。因此當有執行緒要啟動同步機制時，必須提出一個 object lock 來做為進入關鍵區域(Critical Sections)的鎖。而 Thread Controller 會記錄此 object lock，當有其他執行緒要啟動同步機制，就必須先檢查所提出的 object lock 是不是已經被記錄下來。若否，則執行緒可繼續執行。若是，則執行緒將進行切換。

3.2. Ping-pong Java stack memory

對於多執行緒程式的執行環境，每個執行緒都必須保持自己的執行狀態與堆疊內容。在通常的情況下，Java 處理器在紀錄執行資訊所使用暫存器數量往往有限而且可以有效率的移到主記憶體內。然而在 Java 程式執行時，所使用的運算堆疊資料實際的大小是會變動的，且資料數量勢必遠遠超過用於紀錄執行資訊的暫存器。如果我們將每個執行緒的運算堆疊都存放於處理器上的記憶體裡而不是存於 DDR-SDRAM，那在執行緒切換時，堆疊的切換上所花費的成本將是不可忽略的。

在原先 JAIP 處理器設計上，我們已經設計了一組程式執行的運算堆疊 Two-level Java stack memory(如圖 4 所示)，來使得我們可以同時間執行兩組指令。在此設計中，第一層的堆疊是由三個暫存器所組成，存放堆疊最上面的三筆資料。堆疊的第二層由兩個 Dual-port On-chip Block RAM 交互組織成實作交錯的記憶體架構。將四條 ports 區分為兩條 read-ports 與兩條 write-ports。並使用四個暫存器去存放所執行方法的前四個區域變數。這個設計比一般通用的 four-ports 記憶體具有更低的成本，但已能有效的支援大多數雙指令對於堆疊的操作。

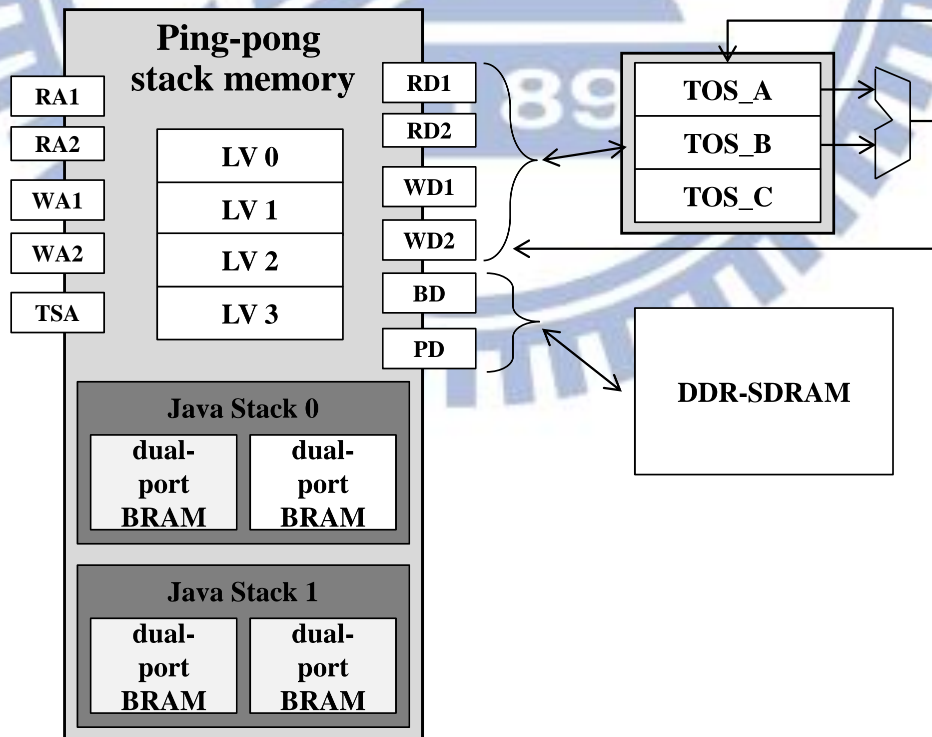


圖 22. Ping-pong Java stack memory

為了解決紀錄各執行緒的堆疊的問題，我們在原先的 Two-level Java stack memory 第二層又增加了一組由兩個 Dual-port On-chip Block RAM 交互組織成實作交錯的記憶體架構，形成 Ping-pong Java stack memory(如圖 22 所示)。換句話說，也就是目前系統中存在兩組 Java 運算堆疊。而當一執行緒正在執行時，Thread Manager Unit 將會藉由排班選出下一個執行可取得執行權的執行緒。而當目前執行緒尚未被切換時，則從 DDR-SDRAM 中將下一執行的執行緒所使用的運算堆疊資料移到另一組堆疊內。而後當 JAIP 切換到下一個執行緒時，可直接切換系統所使用的堆疊即可。而上一個執行緒所使用的堆疊將會被保存到 DDR-SDRAM 與當前執行緒的運作並行。在這個設計之下，執行緒切換時在備份或移入堆疊的開銷，可以利用重疊當下的執行而消除。

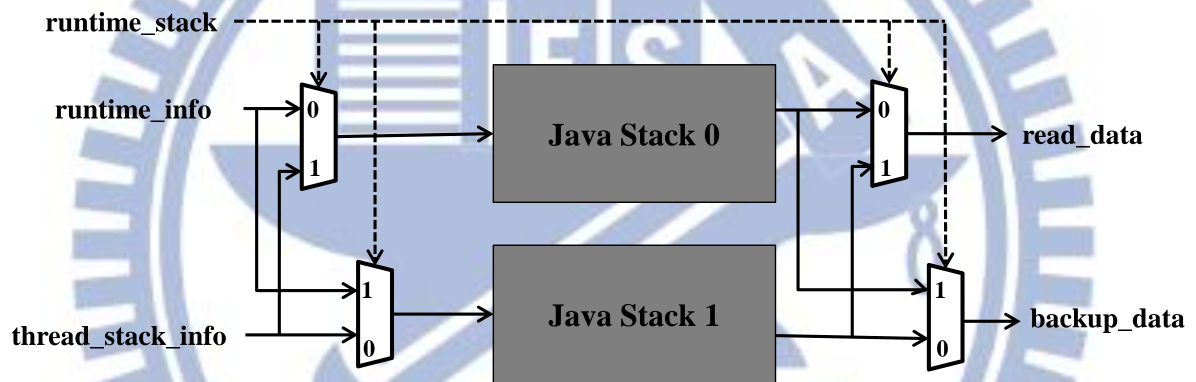


圖 23. Ping-pong Java stack memory 控制示意圖

而在 Ping-pong Java stack memory 的設計，除了在原本 Java stack memory 第二層中多放入一對 Block RAM 形成另一組 Java stack 外，並為了能同時存取兩組堆疊的的資料，我們再加入了一條信號線 Thread stack address(TSA)，用以控制非執行用堆疊的資料移入或移出的位址。以及一組從 DDR-SDRAM 中讀取資料傳進 stack memory 的 Prepare data(PD)與從 stack memory 將備份資料送到 DDR-SDRAM 的 Backup Data(BD)。而在第二層內的兩個 Java stack 則是利用一個暫存器 runtime_stack 來記錄目前系統執行所使用的是何組 Java stack，由 runtime_stack 來判斷將傳進的訊號線正確的拉到所需操作的堆疊。以圖 23 為例，在 JAIP 執行程式時，會傳入運算堆疊的控制訊號 runtime_info(包含讀取堆疊位址、寫入堆疊位址跟資料)進入 Ping-pong Java stack memory。同時間也可能

正在進行準備執行緒堆疊的運作，因此也會傳入另一組堆疊的控制訊號 thread_stack_info(包含堆疊位置與寫入資料)。則這兩組控制訊號會被利用 runtime_stack 來判斷的 MUX 所分辨並傳入其應正確操作的堆疊。而從堆疊讀出的資料亦然。

在使用 Ping-pong Java stack memory 此項機制下，在 JAIP 處理器上並不用保存所有執行緒的運算堆疊，而是將執行緒的堆疊備份於 DDR-SDRAM 中。如此便可減少處理器在管理執行緒上所耗費的資源量。並且可以在 JAIP 在執行時，當 JAIP 沒有需要對 DDR-SDRAM 做存取動作時，堆疊的移入或移出可同時執行。如此一來每次切換執行緒後並不同於其他執行資訊需一次存回 Thread Control Block 中，而大量的堆疊資料便可隨著程式的執行，並行的把堆疊移入或移出到 DDR-SDRAM，並不會額外增加負擔在執行緒切換上。而且可以當切換執行緒時，下一執行緒的資料已放置在另一堆疊上，因此只需改變 runtime_stack 的值即可順利切換 JAIP 執行時所使用的運算堆疊。

由於目前 JAIP 多執行緒的架構最多支援 16 個執行緒同時存在系統中，因此在 DDR-SDRAM 中每個執行緒的備份位置並不是使用執行緒編號來區分，而是直接用此執行緒在 Thread Control Block 的欄位來區分。舉例而言，若一執行緒於 Thread Control Block 放置於欄位 3，則對應到 DDR-SDRAM 的堆疊存放起始位址是為 0x5E003000，若為欄位 9 則是 0x5E009000。如此做法我們只需要知道此執行緒被放置於 Thread Control Block 的欄位編號馬上就可以對應它的運算堆疊存放於 DDR-SDRAM 的起始位址。而由於每一個執行緒所使用的堆疊都是由兩個 2K bytes 的 Block RAM 所組成，因此最多將會在 DDR-SARAM 中佔據 64K bytes 的空間來做為備份執行緒堆疊。

3.3. 系統軟體

在 Temporal Multithreading 的設計架構之下，對於執行緒的操作完全交由 JAIP 來執行，而不須 RISC 處理器的協助。而在 Java 程式語言的設計下，可藉由兩種方法來撰寫新增執行緒的動作。(1) 使用一類別去繼承 java.lang.Thread 類別並覆寫 run()方法(如圖 24.a 所示)；(2) 使用一類別去實作 Runnable 介面，Runnable 介面中只定義一個 run()方

法，然後新增一個 java.lang.Thread 物件時，並將實作 Runnable 介面的物件傳入建構子作為參數，而此物件會調用所實作的 run()方法(如圖 24.b 所示)。而以上兩種方式都須再透過呼叫 java.lang.Thread 類別的 start()方法來新增執行緒，當執行緒被新增後所執行的第一個方法即是被覆寫或實作的 run()方法。換句話說，run()方法對於新執行緒而言可視為 Java 程式開始執行的所呼叫的 main()方法。而按照 Java 標準類別庫，當呼叫 java.lang.Thread 類別的 start()方法時，Java 虛擬機器會透過呼叫原生方式來支援執行緒的運作。然而在 JAIP 的設計架構中，若要透過中斷來交由 RSIC 處理器來管理單一處理器上的執行緒耗費成本過高，因此我們改變系統軟體上的類別解析器(Class Parser)對於執行緒相關運作的解析結果，使得大多數執行緒的運作都執行能在 JAIP 上，而不需由 RSIC 處理器支援。在本節中會先介紹 JAIP 呼叫方法的查找機制。再來說明如何修改類別解析器使得新增執行緒的運作可直接於 JAIP 上執行。

```
class NewThread extends Thread {
    public void run() {
     // override this method
    }
}

public class Thread1 {
    public static void main(String[] args){
        NewThread    t = new NewThread ();
        t.start();
     }
}
```

(a)

```
class NewThread implements Runnable {
    public void run() {
     // implement this method
    }
}

public class Thread2 {
    public static void main(String[] args){
        NewThread    r = new NewThread ();
         Thread    t = new Thread(r);
         t.start();
    }
}
```

(b)

圖 24. 新增執行緒程式；(a)繼承 java.lang.Thread 類別；(b)實作 Runnable 介面

3.3.1. 呼叫方法流程

類別方法的呼叫是透過 Java 位元組碼 invoke 相關指令來實作，包含 invokevirtual、invokespecial 等。而若方法為靜態(static)則是使用 invokestatic 來呼叫。而當 JAIP 執行 invoke 相關指令，會經由 Decode Stage 經過解碼發現是要呼叫方法時，會啟動 Dynamic

Resolution Unit。而 Dynamic Resolution Unit 的狀態改變如圖 5，開始時都會停在 Normal 的狀態等待被啟動，而一旦發現是要執行呼叫方法時，會依據 Decode Stage 所傳過來的相關訊號線以及此次 invoke 指令的參數。此時 Dynamic Resolution Unit 會依照這些相關資訊來查找在處理器上保存的 Class Symbol Table 與 Cross Reference Table。

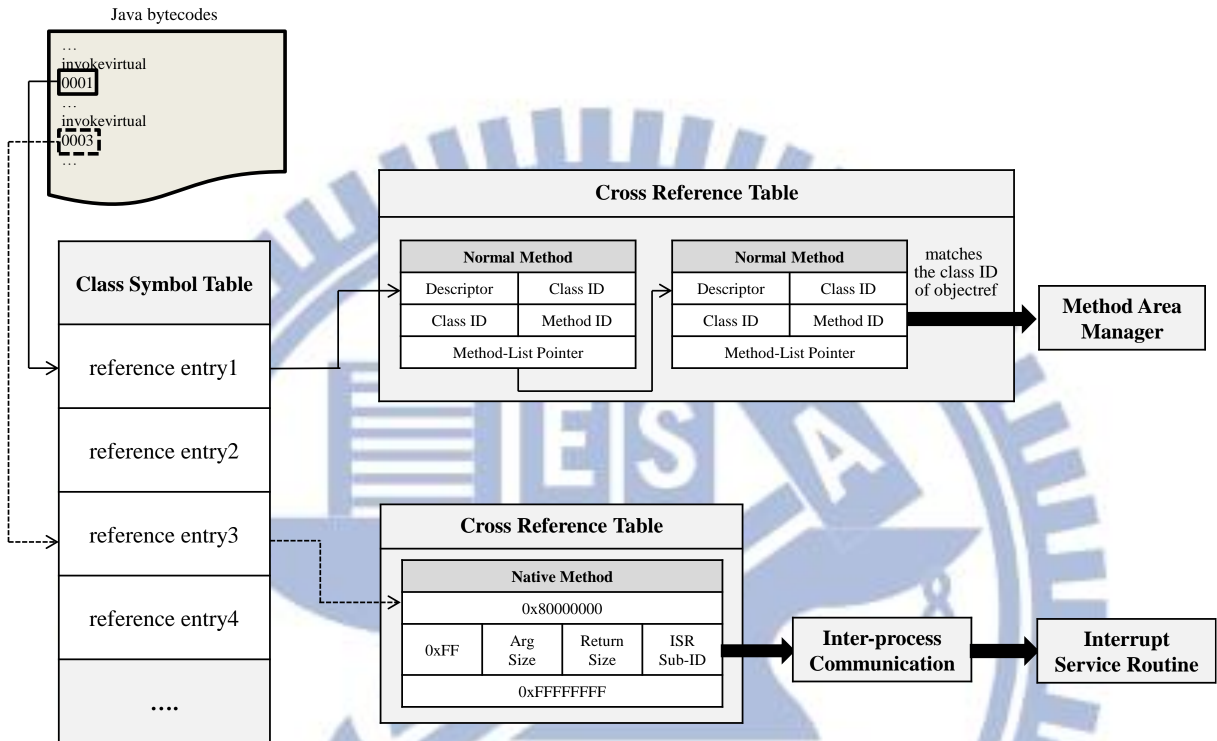


圖 25. 呼叫方法流程圖

如圖 25 所示，Dynamic Resolution Unit 會根據 invoke 指令的參數來查找該類別的 Class Symbol Table，而在此表格內的每一個欄位所存放的資料代表指向 Cross Reference Table 位址。而透過這些位址我們可以取得在 Cross Reference Table 上所要呼叫的方法資訊。而方法資訊區分為兩種(Normal method 與 Native method)。而在 Normal method 中所存放的資訊包含 Descriptor、類別編號，實作類別編號、方法編號以及 Method-List Pointer。由於 Java 程式語言中支援繼承與實作的機制，因此同一個方法可能會被不同的類別所實作，因此在類別解析器在解析類別的時候，會將被不同類別所實作的同一方法串成一組 Method-List，使得 Dynamic Resolution Unit 在查找正確的呼叫方法時，只要透過類別編號在 Method-List 上找尋真正所實作的方法即可。因此 Method-List Pointer 是指向下一個被覆寫的相同方法在 Cross Reference Table 裡的欄位。而 Native method 所存放

的資訊包含參數個數、回傳值數量以及在系統軟體上的原生方法編號。

以圖 25 為例，第一次的 invokevirtual 先查找 Class Symbol Table 的欄位 1 資訊，並依照此資訊索引到 Cross Reference Table 的方法資訊，發現此次呼叫的方法為 Normal method，然而此次取得的方法資訊在類別編號上並不相同，必須開始查找此方法的 Method-List。因此直接取得此方法資訊中的 Method-List Pointer 而並不會取出此方法的編號。之後取出下一個方法資訊，相同的一開始就先比對類別編號是否相同，而若發現類別編號相同時，即取出此方法資訊中的類別編號與方法編號，並將此資訊傳入 Method Area Manager[29]進行 Dynamic Method Loading 機制(如圖 26 所示)與更替 JAIP 所執行的方法。而第二次的 invokevirtual 先查找 Class Symbol Table 的欄位 3 資訊並憑此索引到 Cross Reference Table 的方法資訊，發現此次呼叫的方法資訊第一欄資訊為 0x80000000，憑此判斷是為 Native method。所以並不用進行查找 Method-List 的工作，而是直接取得第二欄的資訊，依據參數個數調整 Inter-process communication 所使用的參數暫存器[31]，並透過這個介面來執行 RISC 處理器上的 Interrupt Service Routine。而當系統軟體接到 JAIP 的中斷要求後，會先檢查中斷編號，若為執行原生方式則系統軟體會從 JAIP 暫存器上取得原生方法編號並執行其方法，執行完畢後將結果寫回 JAIP 並告知已完成。

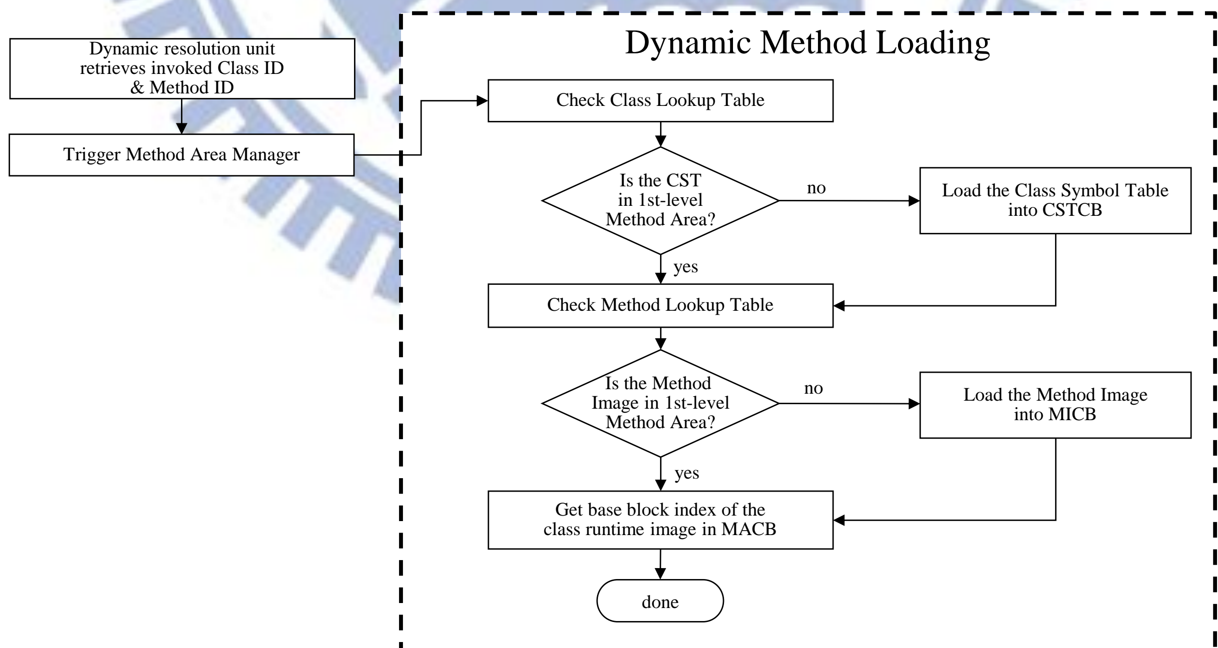


圖 26. Dynamic Method Loading 流程圖

3.3.2. Method Flag

為了讓 JAIP 在 Dynamic Resolution Unit 運作呼叫方法時，能夠分辨出所呼叫的方法是新增執行緒的操作，也就是呼叫 java.lang.Thread 類別的 start()方法。因此我們修改原先由類別解析器對於每個方法解析後存放於 JAIP 處理器上的 Cross Reference Table 欄位資訊。我們先定義一組 8 bits 的資訊為 Method Flag(如表 3 所示)。給予不同方法的性質做出區分，在多執行緒的環境之下，除了原本方法有 Normal 與 Native 之分，再新增了兩項對於方法的描述。首先是 Synchronized 的方法，由於在多執行緒的執行環境下，執行緒之間對於共用記憶體的存取可能會造成競爭條件(Race Condition)的問題。因此 Java 程式語言提供了 *synchronized* 這個關鍵字來修飾方法的宣告，用以解決執行緒同步問題。而在先前研究中皆是以單執行緒架構來設計，應該並未考慮此關鍵字的用法。因此我們再類別解析器中只要解析到有加上 *synchronized* 這個關鍵字的方法，必須再 Dynamic Resolution Unit 的查找過程中得知這是個同步方法，並進行相關同步機制。例如在 Dynamic Resolution Unit 查找中所取得 Normal Synchronized Method 的 Method Flag 即為 0x20、而 Native Synchronized Method 的 Method Flag 即為 0xA0。而第二種新增的方法描述為產生執行緒的方法，即專門為 java.lang.Thread 類別的 start()方法所設計的資訊。當 JAIP 執行到 start()方法時，Dynamic Resolution Unit 取出的 Method Flag 即為 0x40，此時會改變這個方法的查找過程，而不是如同 Java 標準類別庫中定義的去執行原生方法，使得新增執行緒的操作能完全在 JAIP 上執行而不需 RISC 處理器的協助。start()方法的查找過程會在下一小節說明。

表 3. Method Flag

Method Flag	Definition
0x00	Normal
0x20	Synchronized
0x40	New thread
0x80	Native

我們新定義了一組 Method Flag 資訊來判斷方法的類別，而也修改類別解析器放置於 JAIP 處理器上的 Cross Reference Table 欄位格式。而在 3.3.1 中有說明所存放的方法資訊的格式分為兩種(Normal method 與 Native method)。皆重新定義第一欄的前 8 個位元為 Method Flag 資訊(如圖 27 所示)。在如此修改下，當 Dynamic Resolution Unit 在查找方法資訊時，一開始便會取得第一欄的資訊用以判斷是否為原生方法或類別編號是否為所要執行的方法，此時可連同 Method Flag 的資訊一併取得。即可判斷此方法是否要執行同步化機制或是新增一組執行緒的操作。

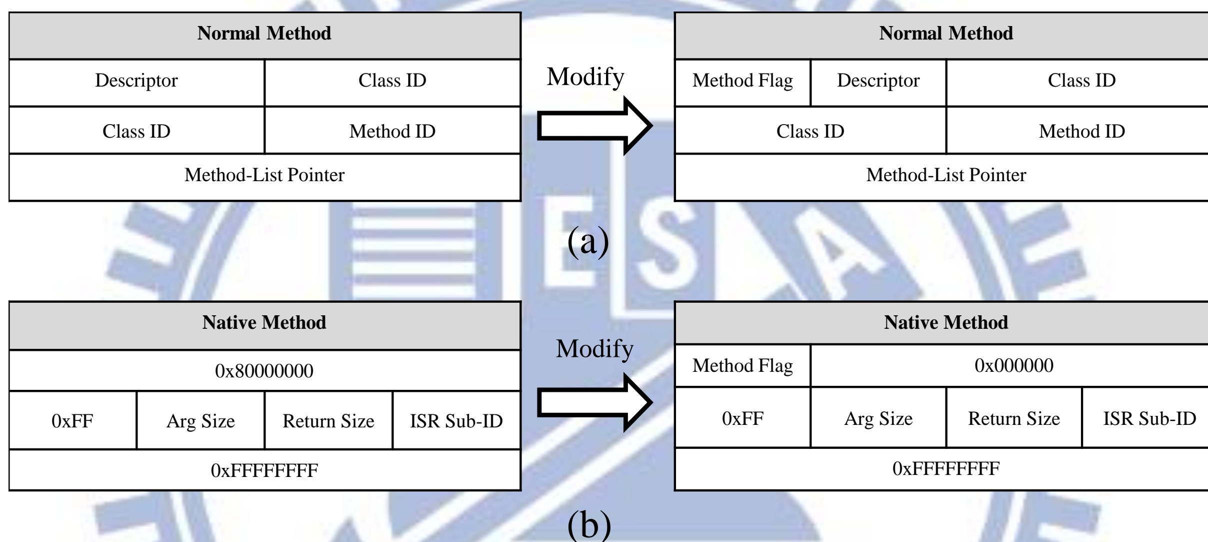


圖 27. 方法資訊欄位更新；(a) Normal method；(b) Native method

3.3.3. 類別解析器對於新增執行緒之設計

當要於系統中新增一組執行緒時，3.3 章有提出兩種程式撰寫方式。然而這兩種方式的相同點在於當要開始新增執行緒時，都必須要呼叫 java.lang.Thread 類別的 start()方法，利用原生方法來實作新增執行緒，並在底層的系統軟體管理此執行緒。而且當新執行緒在第一次取得執行權執行時，執行的是被覆寫或實作的 run()方法，也就是形同程式開始的 main()方法一般。而在 3.3.2 節中我們已經新增了一組 Method Flag 資訊，可以在執行呼叫方法時由 dynamic resolution 的機制來判斷出目前所執行的是 java.lang.Thread 類別的 start()方法，但是我們卻不能把此方法的類別編號與方法編號傳給 Thread Manager

Unit 來對執行緒做初始化的動作，因為 start()方法並不是我們真正所要執行的方法，而對於 JAIP 來說呼叫 start()方法只是得知目前系統中要新增一組執行緒，而我們需要額外的修改來得知執行緒所真正要執行的方法。

由於在執行緒程式撰寫上，會覆寫 java.lang.Thread 類別的 run()方法來定義此執行緒應該進行何種操作。依照先前所設計的物件解析器的解析流程，會被不同類別所覆寫的相同方法組成一個 Method List，便於查找真正所要執行的方法。因此我們利用此種設計，在解析 java.lang.Thread 類別時，在處理器上的 Cross Reference Table 將 Method Flag 0x40 填入 start()方法的方法資訊，並且將此方法的 Method-List Pointer 直接指向同類別的 run()方法的 Method-List，使得我們可以直接去查找所要執行的 run()方法。

以圖 28 為例，有兩個類別 A.class 與 B.class 繼承 java.lang.Thread 類別，並覆寫 run()方法來定義他們對於執行緒所要執行的程式。因此可以看到系統軟體的類別解析器會在解析這些類別後，於 Cross Reference Table 建立如圖中之關聯。首先是由於 A、B 兩類別皆覆寫了 run()方法，因此從 java.lang.Thread 類別的 run()方法開始串起一組 Method List。並且將 java.lang.Thread 類別的 start()方法的 Method-List Pointer 指向所形成的 run()方法的 Method List。圖 28 是舉例當要使用 B 類別來產生新執行緒，則當 B 類別呼叫 start()方法的時候，會利用在取得 Class Symbol Table 的索引指向 Cross Reference Table 取得 start()方法的方法資訊。Dynamic Resolution Unit 會發現 Method Flag 的值為 0x40，代表這次的呼叫方式是要進行新增執行緒的動作。而 Dynamic Resolution Unit 會記錄此資訊並直接開始查找 run()方法的 Method List，在查找的過程中會發現 java.lang.Thread 類別與 A 類別的方法資訊類別編號都不相同，直到找到 B 類別的方法資訊編號相同。此時會取出這個方法的類別編號與方法編號，因為之前紀錄此次呼叫方法為新增執行緒的運作，因此會將所查找到的資訊以及產生新增執行緒的訊號送到 Thread Manager 中初始化執行緒資訊。等到這個執行緒取得執行權後，所執行的即是 B 類別所覆寫的 run()方法。由於我們對於系統軟體的類別解析器解析過程的修改，使得執行緒的新增不再需要透過原生方式來支援。而是直接由 JAIP 藉由呼叫方法在 dynamic resolution 的機制運行中就可以取得新增執行緒的所有資訊。

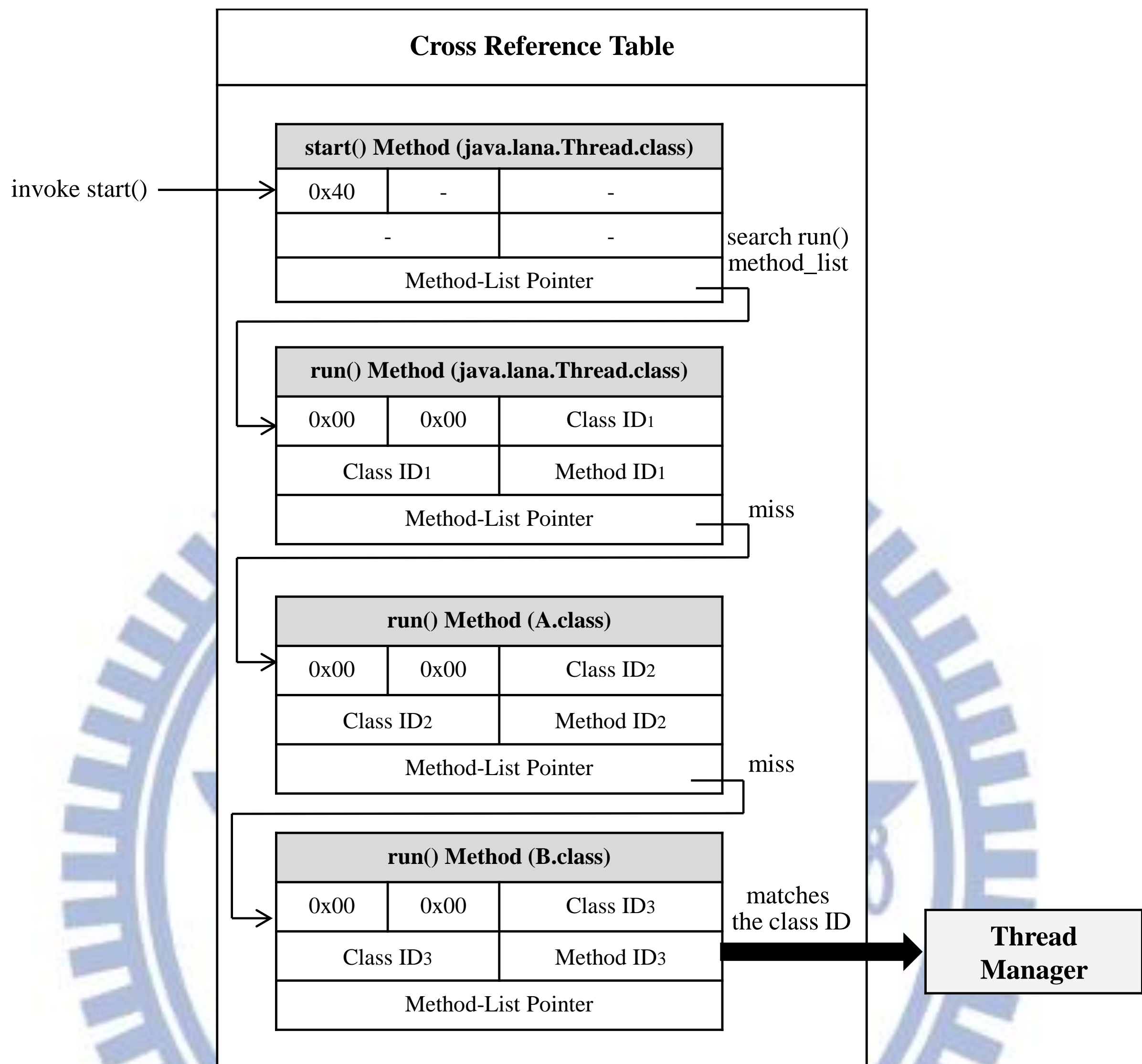


圖 28. Cross Reference Table 查找 start()方法資訊

3.3.4. 硬體原生方法介面

我們將管理執行緒的機制移到 JAIP 上，而不需要 RISC 處理器的協助。然而在 java.lang.Thread 類別中還定義了對於操作執行緒的原生方法(如 yield()、activeCount() 等)。若是由原本使用系統軟體來支援原生方式來支援這些方法的話，則還需要透過中斷，且 RSIC 處理器還需要向管理執行緒的 JAIP 索取資料，如此的作法所耗費的成本過高。因此我們呈現一種設計機制，使得能使部分的原生方法能交由 JAIP 在硬體上即可支援這些原生方法的執行。

因此我們將原生方式的方法資訊(如圖 27.b 所示)第二個欄位的 Sub-ID 再做出分類。而利用這 8 bits 資訊的最高位原來判斷此原生方法是否由 JAIP 來實作。換句話說就是如果這個 Sub-ID 所存放的值是低於 128 的話，將會依照之前所設計的方法來透過中斷通知 RISC 處理器來執行 Interrupt Service Routine，由系統軟體來支援原生方法的執行。而若此 Sub-ID 所存放的值是大於 128 的話，此方法將會由硬體的電路邏輯或加速器來實作，而不會再需要 RISC 處理器的協助。此外，若原生方法為 RISC 處理器實作，則 Sub-ID 為系統軟體所執行原生方法的編號；若為由硬體來支援，此 ID 代表硬體原生方法之編號。表 4.列出再 JAIP 所使用的執行類別庫中 java.lang.Thread 中的原生方法，而這些方法都是直接對執行緒的資料操作，因此若有使用到這些方法則可利用此節所敘述之硬體原生介面來支援這些原生方法。

表 4. java.lang.Thread 類別的原生方法

Thread.activeCount()
Thread.currentThread()
Thread. interrupt0()
Thread.isAlive()
Thread. setPriority0(int i)
Thread.sleep()
Thread.yield()

第四章 多處理核心系統架構

在上一章中，我們提出了在單一處理器中能執行多執行緒的系統架構。然而在這樣的作法下，的確能使得多執行緒的機制被支援，然而執行緒卻不是並行執行的，只是透過執行權的切換讓執行緒可以相互交替執行，造成執行緒並行執行的假象。因此這樣的作法只能執行多執行緒的程式，卻無法真正使得執行效率提高。因此本章節提出多處理核心執行環境和新設計架構藉以達到真正執行緒並行，以提高執行效率。

為此我們以原先設計之 Java Accelerator IP （JAIP）架構為基礎，利用在每個 JAIP 上執行一組執行緒而組成多處理核心的執行環境。使得在多處理核心的環境下執行緒可以分別在每個 JAIP 上運作達到並行。原先 JAIP 在 heap 空間的處理上有兩個不同的作法。第一種是把 heap 直接放在 on-chip memory 上，因為 on-chip memory 大小有限，這作法並不適合多執行緒的 Java 應用。第二種架構是把 heap 放在 DDR-SDRAM，並利用一個整合在 JAIP 內部的 two-way set associative cache 來加快 heap objects 的存取速度(如圖 29 所示)。第二種作法雖然比較適合改成多核心、多執行緒的架構，但是之前的 cache controller 並沒有處理多核心下的 cache coherence 問題，所以在這邊，我們必須對原先的 JAIP 架構做修改，以解決各處理核心在執行時，當 Heap 的資料修改時，只會在處理核心自己的 Heap Cache 上修改，而其他的處理核心上還是未修改的值，因此造成 Heap 資料的不一致。為了處理資料一致性(Data Coherence)的問題，以及管理執行緒的分配執行，我們設計了一個 Multi-core Coordinator 元件。而其中包含了兩個主要元件，Data Coherence Controller 與 JAIP Manager。而 Multi-core Coordinator 是獨立於處理核心之外的一個 IP。其中 Data Coherence Controller 對於執行緒所需運算資料主要包含有兩個功能，其一是在於當有某一 JAIP 處理核心要對 Heap 上的資料作修改時，Data Coherence Controller 將保持其他處理器的 Heap Cache 上的資料保持一致性。另一作用是在於支援 Java 程式語言的同步機制。

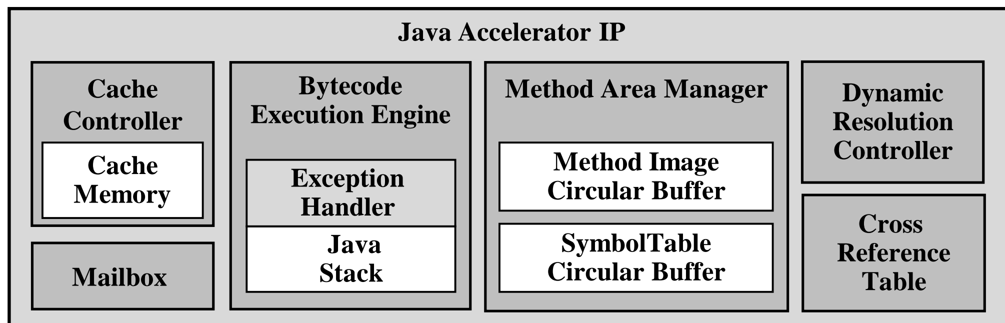


圖 29. 多處理核心環境之 JAIP 系統架構

然而在此多處理核心的環境之下，不再像上一章一樣將執行緒的管理工作放置於 JAIP 處理核心上。而是由 Multi-core Coordinator 來支援。而 Java 處理器對於執行緒的相關操作也將交由 JAIP Manager 來做處理。在本章以下小節內，說明多處理器的執行環境、Java 處理器上的 Heap Cache Controller 以及 Data Coherence Controller 的設計。

4.1. 多處理器執行環境

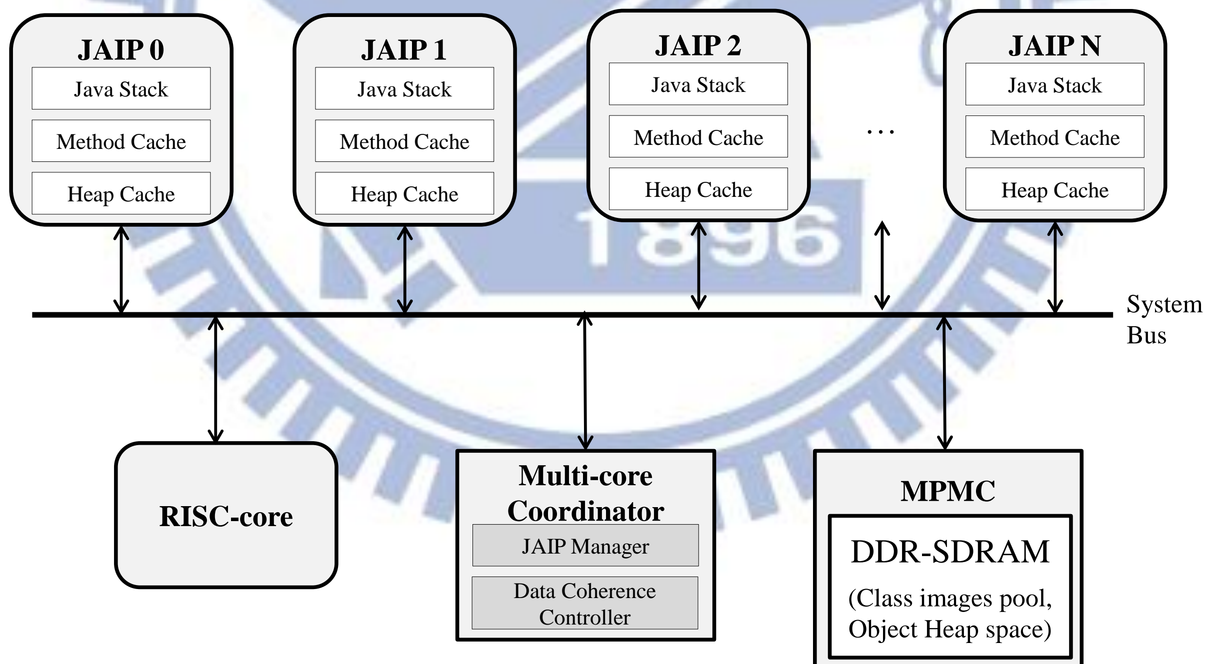


圖 30. 多處理核心系統架構

為了讓執行緒能夠真正達到並行，因此我們提出了多處理核心的執行環境架構(如圖 30 所示)。在系統加入多個 JAIP，而每個 JAIP 處理核心上分別有自己獨立的執行堆疊、Method Cache 與 Heap Cache。這些記憶體元件將只屬於在處理核心上所執行的執行

緒來運作。並同原先 Java 處理器的設計，在系統中會存在一個 RISC 處理核心，主要負責於執行系統軟體來協助執行，包含類別解析器、原生方法等。由於 Object Heap 上的資料在運作時是共同使用的，因此我們將完整的 Object Heap Space 改移到 DDR-SDRAM，使得每個處理核心上不需要有完整的 Object Heap Space，而只需要藉由 Heap Cache 的機制來在處理核心上自己所使用到的資料即可。因此對於 JAIP 處理核心而言，DDR-SDRAM 上不再只有由系統軟體的類別解析器所產生每個類別的執行映像檔，而還包含存放了 Java 程式執行時完整的 Object Heap Space。而 Multi-core Coordinator 中的 Data Coherence Controller 此元件負責協助處理核心間的 Heap Cache 上的資料一致，並管理程式執行的同步機制。且在多處理核心環境中，Multi-core Coordinator 還須負責執行緒的管理，包含新增執行緒、指派處理核心執行等。

對於執行緒的管理，在此系統架構上是由 Multi-core Coordinator 中的 JAIP Manager 來支援。因此在此元件上會記錄各處理核心目前的狀態，以及目前系統中所存在的執行緒狀態。由於我們目前的設計架構是在單一處理核心上只能執行一條執行緒，且並無執行緒切換之機制。因此在 JAIP Manager 上，並不需要紀錄每個執行緒在執行過程中的資訊。所以目前管理執行緒的作法較為單純，負責管理紀錄每個在系統中的 JAIP 處理核心當下是否已經有執行緒在執行中。若是當有一個新增執行緒的事件發生，會去選出目前還在閒置的 JAIP 處理核心，並將執行緒的資訊傳到該處理核心中並驅動此核心使其能開始進行執行動作。而且 JAIP Manager 會將各處理核心狀態記錄下來，形成 JAIP Information Table(圖 31 所示)，此表是利用 JAIP 處理核心的編號做索引，分別記錄該處理核心目前的運作狀態(0 表示 JAIP 正在閒置，1 表示 JAIP 目前有執行緒在運作中)，以及所執行的執行緒編號。

JAIP ID	Active(0 or 1)	Thread ID
0		
1		
2		
⋮		

圖 31. JAIP Information Table

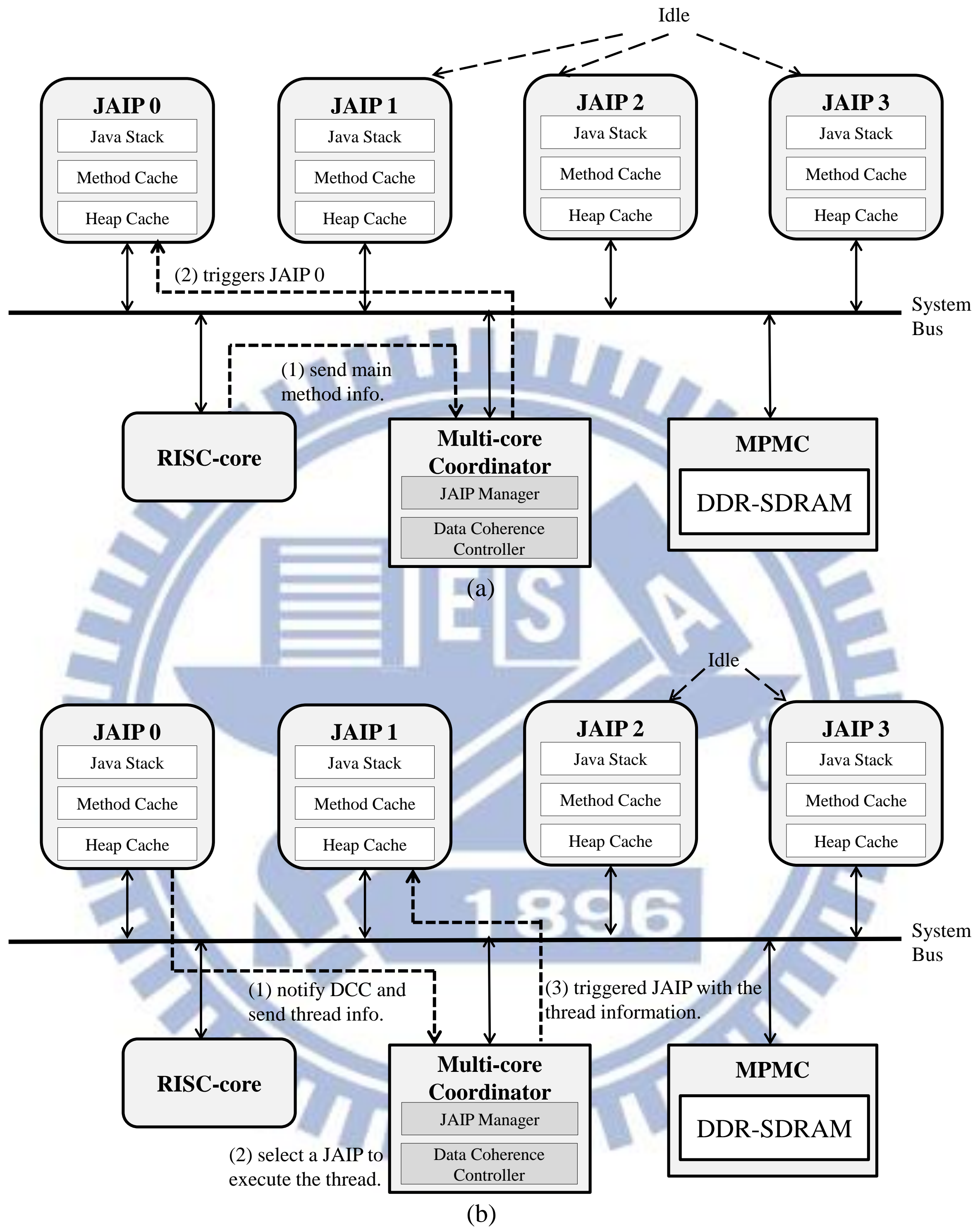


圖 32. 多處理核心系統運作；(a) 系統啟動；(b) 新增執行緒

以圖 32 為例，在這個例子中我們在系統裡面放入 4 個 JAIP 處理核心。當系統一開始啟動時，系統軟體給定每個處理核心一個獨立的編號，且把所有在系統中的 JAIP 處理核心判斷是否開始運作的暫存器設為閒置狀態。並使得 JAIP Manager 中的 JAIP

Information Table 中所有欄位初始化(如圖 33.a 所示)，且由系統軟體會先將一開始所要執行的類別作解析。而當系統軟體已完成解析後，會將解析的資訊與系統開始執行的訊號傳到 Multi-core Coordinator。而 JAIP Manager 當收到系統開始執行訊號時，會先選定其一在閒置中的 JAIP 處理核心，在此例中首先指定 JAIP 0 作為第一個所運作的處理核心，而將之前由 RISC 處理核心所送來的 Java 程式一開始所執行 main()方法的類別編號與方法編號送到 JAIP 0，並改變 JAIP 0 判斷運作的暫存器(如圖 32.a 所示)，使得 JAIP 0 更改為運作狀態並開始執行程式。而且此時會在 JAIP Information Table 記錄編號為 0 的 JAIP 處理核心已在運作，以及填入主執行緒的編號(系統啟動後首先執行的主執行緒編號固定為 0)。而其他三個處理核心尚未開始執行依然是閒置狀態(如圖 33.b 所示)。

JAIP ID	Active	Thread ID
0	0	X
1	0	X
2	0	X
3	0	X

(a)

JAIP ID	Active	Thread ID
0	1	0
1	0	X
2	0	X
3	0	X

(b)

JAIP ID	Active	Thread ID
0	1	0
1	1	1
2	0	X
3	0	X

(c)

圖 33. JAIP Information Table 之改變 ；
(a) 初始化；(b) 開始執行 Java 程式；(c) 新增執行緒

而在新增執行緒的例子而言(如圖 32.b 所示)，在多處理核心的執行環境下，RISC 處理核心上執行的系統軟體是依照 3.3 節所敘述的方式來修改，因此當 JAIP 0 處理器呼叫到 java.lang.Thread 類別的 start()方法時，所以在 dynamic resolution 機制查照過程中，會得知此次的呼叫是為了新增一組執行緒。所以在查找的過程會找出此執行緒所要執行的方法，並將其記錄於暫存器中。而在多處理器的系統架構下將修改 dynamic resolution 機制如圖 14 一般，新增一狀態為 NewThread。但功能卻不同於 3.1.1.節所敘述。而是由 JAIP 0 直接來通知 Multi-core Coordinator 要執行新增執行緒的動作並傳送此執行緒的相關資訊。因此當 JAIP Manager 收到新增執行緒要求後，會先給予新執行緒一個獨立的編號(此例中編號為 1)，並記錄由 JAIP 0 所傳送過來此新增執行緒所要執行的方法編號與類別編號，且檢視系統中是否有尚未運作而閒置的處理核心。在此例中 JAIP 處理核心編號 1、2、3 皆處於閒置狀態(如圖 33.b 所示)，因此選定處理核心 JAIP 1 來運作此執行

緒。而將新執行緒所執行類別編號與方法編號送入 JAIP 1 的中記錄，並改變 JAIP 1 判斷運作的暫存核心，使得 JAIP 1 更改為運作狀態並開始執行程式。而且此時會在 JAIP Information Table 記錄編號為 1 的 JAIP 處理核心已在運作狀態，以及填入此次所新增執行緒的編號(如圖 33.c 所示)。

然而在之前的設計中 JAIP 處理核心終止執行時，JAIP 處理核心並不會主動去通知 RISC 處理核心已經執行終止。而是會直接將處理核心上判斷執行的暫存器設為閒置狀態，由 RISC 處理核心上的系統軟體會一直監看著處理核心的執行狀態，若有發現處理核心已經將自己設為閒置狀態，此狀況就是已經執行完畢而終止。然而在多核心的執行環境之中，我們包含了多個 JAIP 處理核心，且各核心的狀態是由 JAIP Manager 來負責管理。因此當 JAIP 處理核心內執行完畢之後，除了將自己設為閒置狀態以外，還需通知 Multi-core Coordinator 此處理核心已經執行終止。當收到終止執行的訊號後，會將此處理核心在 JAIP Information Table 中的資訊清除，並改為閒置狀態，藉以等待下一次被分配執行緒執行。此外在 JAIP Manager 也維護一個暫存器來記錄是否所有 JAIP 處理核心都已是閒置狀態，代表系統所要執行的程式已執行完畢。而 RISC 處理核心只要檢查此暫存器即可，若發現暫存器已被設為執行完畢，則結束整個系統的運作。

4.2. Object Heap Cache

Java 應用程式在執行時，所使用的 Object Data 皆是存放於 Heap space 上。如本章一開頭的描述，在多處理核心環境下，我們必須保證 heap cache 在各核心之間的資料內容是一致的，也就是要解決 cache coherence 的問題。簡單地說，當各處理核心一旦將 Heap space 上的資料快取起來，往後的執行只要有存放於處理核心上 Object Heap Cache 上的資料就會直接存取，因此可能會造成處理核心間的資料不一致。因此我們才在 JAIP 多處理核心的設計架構上加入 Data Coherence Controller 元件來協助處理資料的一致性，而此元件會於 4.3 節說明。

在 JAIP 處裡核心中，heap cache 機制是利用兩組由暫存器集合仿照 On-Chip Block RAM 方法組成 2-way set associative cache(如圖 34 所示)。利用原先對外存取的 32 bits 位址最低的 25 bits 資料作為查找 Heap Cache 的資訊。而一次所快取的資料區塊(data block)為 8 筆資料，且兩組暫存器集合所形成的 Cache 個別最多都能存放 64 個資料區塊。因此將存取的位址分別區分為 14 bits 的 Tag、6 bits 的 Index 以及 5 bits 的 Offset。並在 Cache 中對於每個資料區塊加入一筆資訊。是為 valid bit，用以判斷 Cache 中此資料區塊的欄位是否已有快取資料存放。

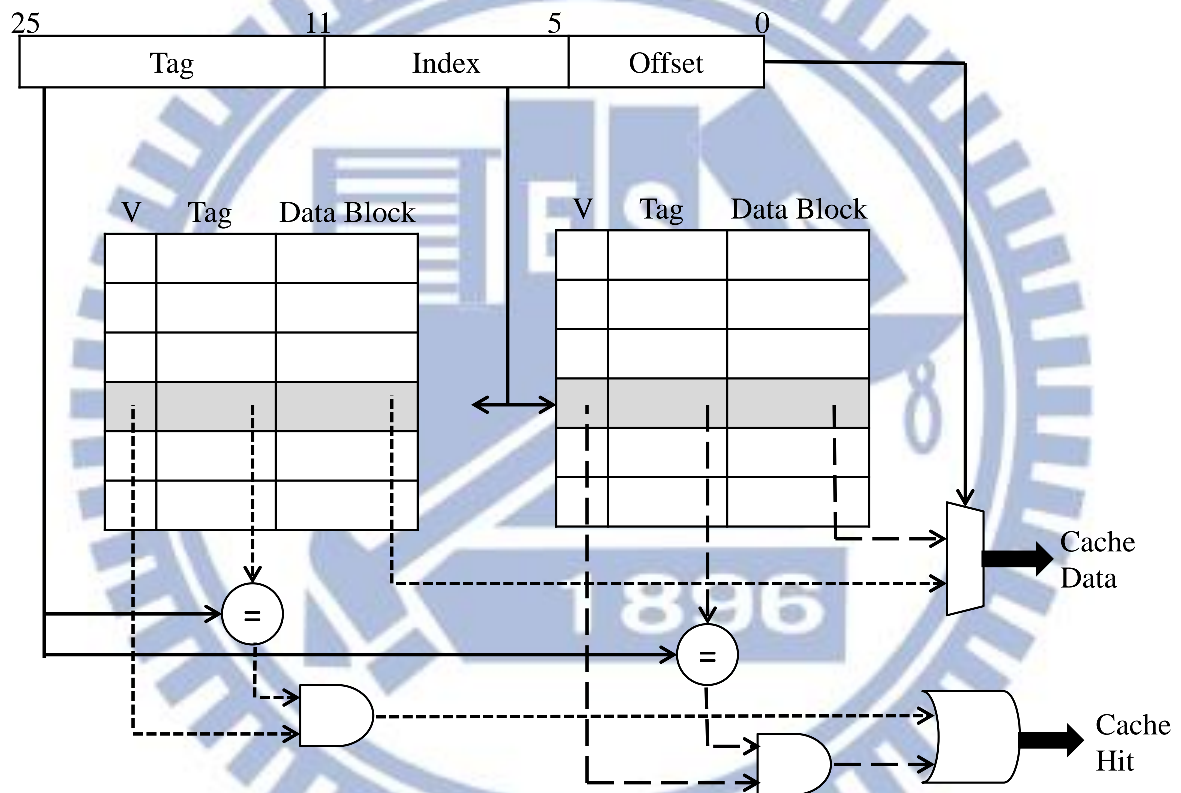


圖 34. Object Heap Cache

圖 35.所敘述的是當 JAIP 在執行程式需要存取 Heap 資料時，對處理核心上的 Object Heap Cache 發出存取需求後，Cache Controller 的操作流程。而對於 Object Heap Cache 的操作可分為讀寫兩類，並以所存取的 Heap 資料是否已快取到 Object Heap Cache 又可分為兩類，總共會有四種情況。以下將說明這四種情形。

(1)讀取資料但資料不在 Object Heap Cache 中：當 Cache Controller 收到 JAIP 執行時要讀 Heap 資料時，會離開 Idle 狀態進入到 Analysis 狀態，此時經過查找機制(如圖 34

所示)會發現所需讀取資料尚未被快取到 Object Heap Cache 中，因此會進入 RdFromMem 狀態，對 DDR-SDRAM 發出讀取的要求。由於一個資料區塊是為 8 筆資料，因此利用 PLB Burst 模式可在一次要求中取回所有資料。當把載入的資料更新 Object Heap Cache 同時也將並回傳 JAIP 執行時所需讀取的資料。

(2)讀取資料而資料已存在於 Object Heap Cache 中：當 Cache Controller 收到 JAIP 執行時要讀 Heap 資料時，會進入 Analysis 狀態來查找。而當找到所需讀取的資料已被快取。此時會直接進入 ChitFinish 狀態，直接從 Cache 中取出並回傳 JAIP 執行時所需讀取的資料。

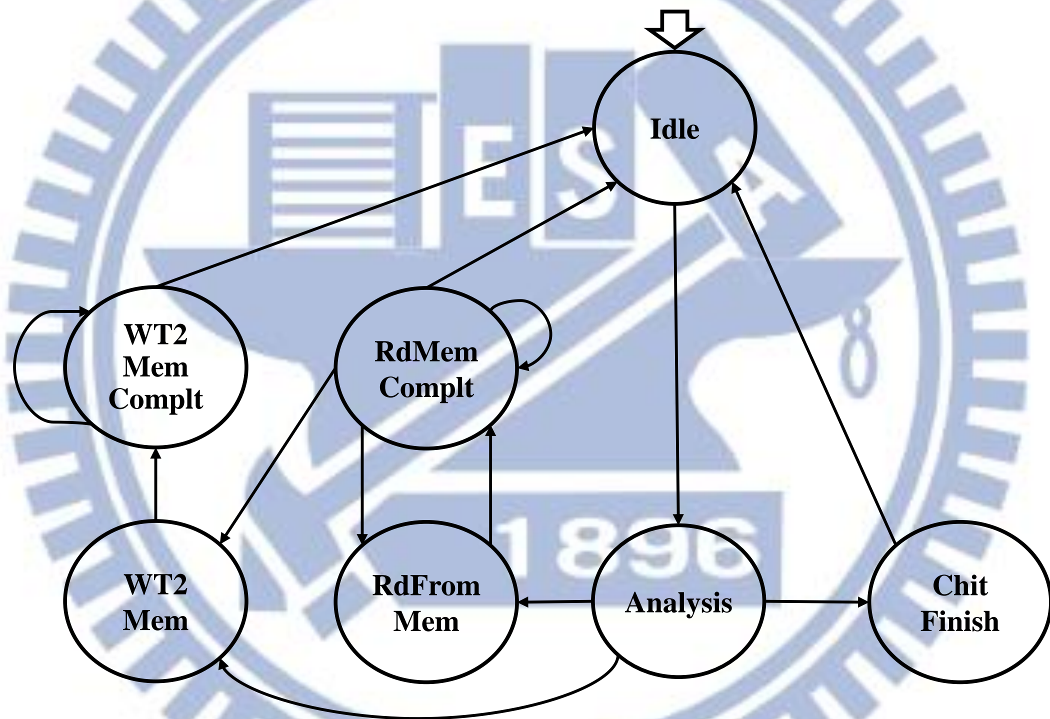


圖 35. Object Heap Cache 存取有限狀態機

(3)寫入資料但資料不在 Object Heap Cache 中：當 JAIP 在執行需要修改 Heap 資料時，會發出寫入訊號給 Cache Controller。則執行狀態會進入 Analysis 狀態，發現資料尚未被快取到 Object Heap Cache 中。而會先從 DDR-SDRAM 讀取此資料所屬的資料區塊並更新 Cache。當對 DDR-SDRAM 存取完畢後即將所需修改的資料一併寫到 Object Heap Cache 之中。然而為了在多處理核心的環境之下保持各 JAIP 處理核心以及 DDR-SDRAM 的 Heap 資料一致，因此會進入 WT2Mem 狀態，將所修改的資訊寫回到 DDR-SDRAM

中的 Object Heap Space，使其之後要 cache 此資料的處理核心能取得正確的資料。並將所修改的資訊(包含 Heap 位址與修改資料)送到 Multi-core Coordinator，由 Multi-core Coordinator 中 Data Coherence Controller 來運作保持各 JAIP 處理核心資料一致性。

(4) 寫入資料而資料已存在於 Object Heap Cache 中：JAIP 在執行需要修改 Heap 資料時，而此位址的資料已被快取。因此 Cache Controller 不須從 DDR-SDRAM 中載入資料直接進入 WT2Mem 狀態。修改 DDR-SDRAM 中的 Object Heap Space，並將所修改 Heap 資料的資訊送到 Data Coherence Controller 即可。並且同時更新在處理核心 Object Heap Cache 上的資料。

而當被修改的 Object Heap 相關資訊送到 Data Coherence Controller 之後，Data Coherence Controller 會再將被修改的資訊轉送到各 JAIP 處理核心上，並啟動檢查是否需要更新的機制(如圖 36 所示)。當 JAIP 處理核心收到由 Data Coherence Controller 所送來要更新 Object Heap Cache 的資訊時。會先檢查所被更改的資料是否快取在自己處理核心上的 Cache 內。若已快取此筆資料，則利用所送來的資訊更新 Cache 上的資料。若否，則不須執行任何動作。如此一來當有 JAIP 處理核心對 Heap 資料做修改時，也可以反映到所以處理核心藉以達到資料的一致性。

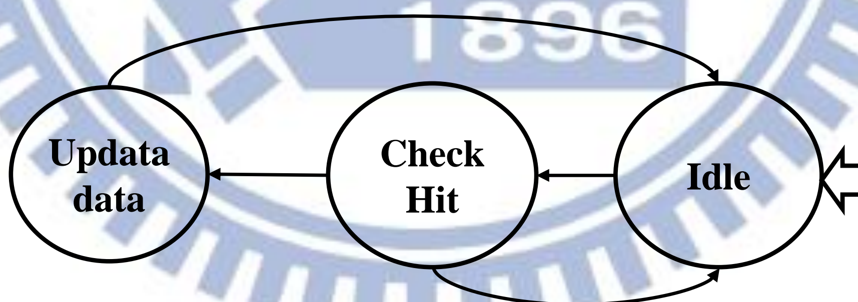


圖 36. Cache Updata 有限狀態機

4.3. Data Coherence Controller

Data Coherence Controller 對於維護各處理核心所需運算的資料主要有兩個功能(如圖 37 所示)。其一是在於當 JAIP 處理核心要對 Heap 上的資料作修改時，除了更新自己的 Object Heap Cache 上的資料外，還須將修改的位址與資料傳送到 Data Coherence

Controller，並由此元件負責通知其他 JAIP 處理核心所修改的資訊，維持每個 JAIP 處理核心上的 Object Heap Cache 內所保留的資料一致性;另一作用在於支援執行緒的同步機制，當有處理器上的執行緒要呼叫同步方法或要進入關鍵區域(Critical Sections)時，則必須將所做為 Object Lock 的 object reference 傳到 Data Coherence Controller 來判斷是否能執行這個同步的運作。而此功能並非 JAIP 處理核心的一部分而是建立在一個獨立的 IP 內。針對此兩個運作功能，將於以下兩小節來說明設計架構。

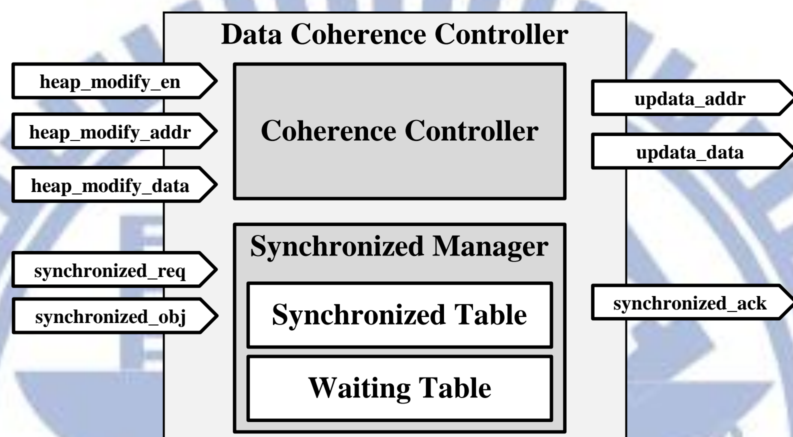


圖 37. Data Coherence Controller

4.3.1. Heap Cache Coherence 之機制

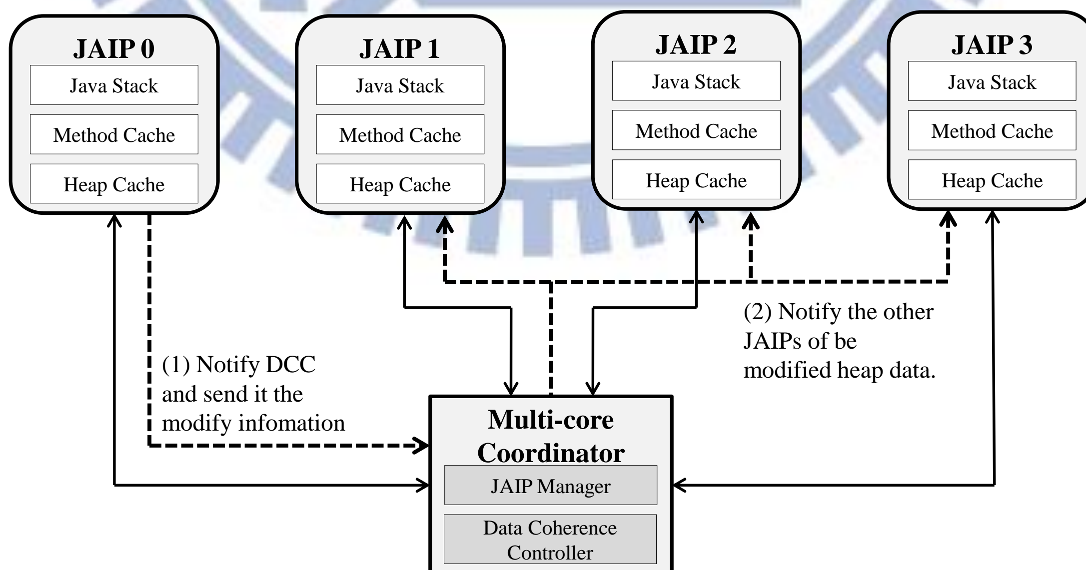


圖 38. Heap Cache Coherence 運作機制

在多處理核心的環境下，我們將 Object Heap Space 放置於 DDR-SDRAM 上，並且在每個 JAIP 處理核心上加入 Object Heap Cache 的機制。由於 Heap 內的資料是屬於所有執行緒所共享，因此每個 JAIP 處理核心皆可能會修改 Heap 上的值，則可能會造成各 JAIP 處理核心上 Heap Cache 的資料不一致的現象。

為了解決 Object Heap Space 資料會因為 JAIP 處理核心只修改自己所快取的內容所造成其他處理核心上資料不一致的問題。我們在系統中加入支援維持各 JAIP 處理核心資料一致性的機制。圖 38.為例，當編號 0 的 JAIP 處理核心對 Heap 的資料作修改時，會藉由 4.2 節所敘述的 Object Heap Cache 機制，會先對在處理核心上的 Cache 做修改，並且將所修改 Heap 資料的相關資訊(包含所修改的 Heap 位址與資料)傳到 Data Coherence Controller 中。而當 Data Coherence Controller 收到來自 JAIP 處理核心的修改要求後，會將此更新 Heap 的資訊轉送到其他的 JAIP 處理核心。各 JAIP 處理核心收到更新的資訊後，會檢查此資料是否存放於 Object Heap Cache 之中，若已快取此筆資料則更新。而圖 39.則列出當有某一 JAIP 處理核心對 Object Heap 資料做修改時，為了維持其他處理核心上所已經快取到的資料一致性，而透過 Data Coherence Controller 來支援的流程。然而藉由這個機制，使得各個 JAIP 處理核心上的 Object Heap Cache 所快取的資料能夠保持相同。

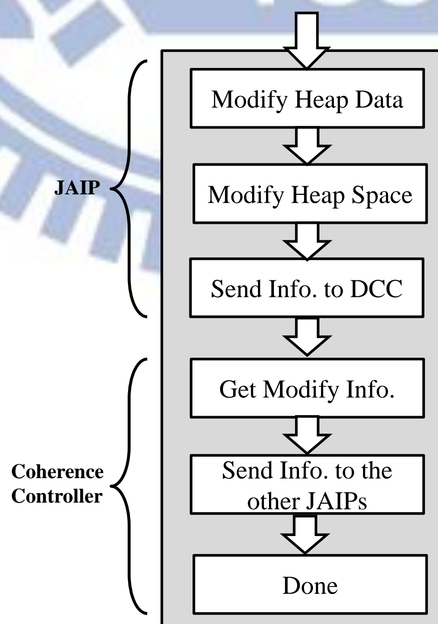


圖 39. Heap Cache Coherence 機制流程圖

4.3.2. Java 同步機制之支援

在 3.1.6 節提到，在 Java 程式語言中程式撰寫者可以對某些資料的讀寫或方法的呼叫，強制規定只有一個執行緒能運作。因此提供了 *synchronized* 此關鍵字來修飾方法或定義一個關鍵區域(Critical Sections)。而當要程式執行中要啟動此同步機制時，會使用一個 object reference 來作為 Object Lock。當有其他執行緒要啟動同步機制但所使用的 object reference 已被某一執行緒作為同步的 Object Lock 時，此執行緒必須等到它離開由此 reference 所鎖住的同步機制後，才能再開始執行。而 Java 同步機制是用於當某些資料可能會由多個執行緒皆對它做出修改而造成的競爭條件(Race Condition)問題，規定一次只有一組執行緒能進入由同一個 Object Lock 所鎖住的同步機制，使得在機制內的資料只會被單一執行緒所修改。

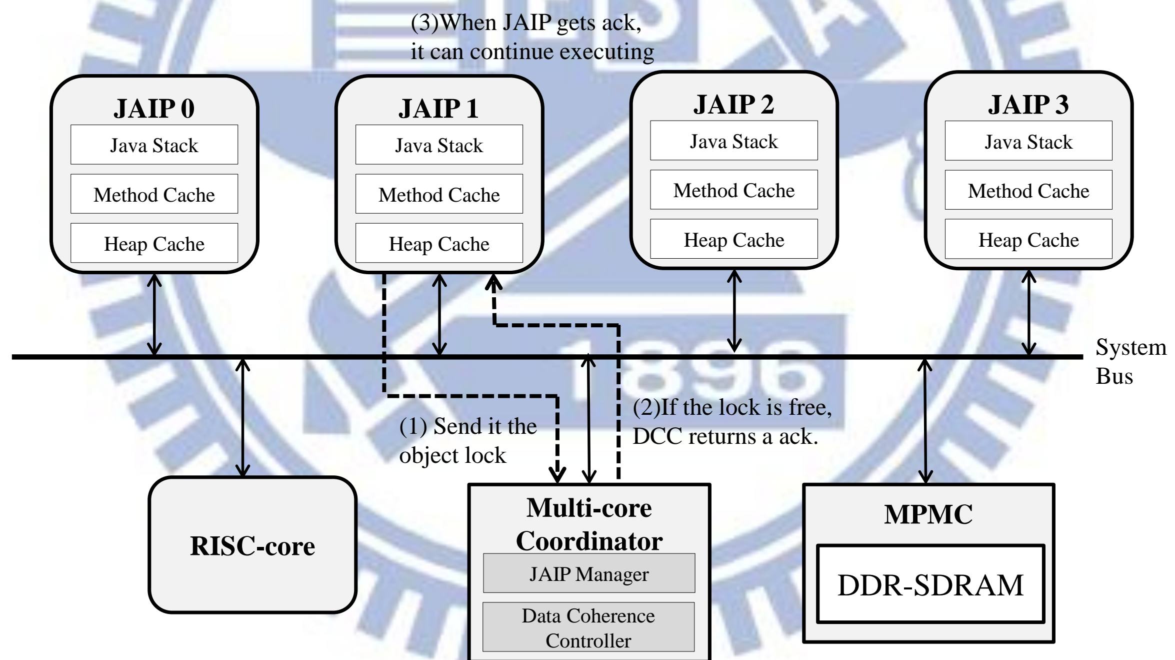


圖 40. 同步機制運作流程

為了支援同步機制，我們在 Data Coherence Controller 中加入了 Synchronized Manager，負責來管理處理器間的執行同步機制。並在此單元中將由執行緒所提出 Object Lock 紀錄為 Synchronized Table。以圖 40.為例，當有執行緒想要啟動同步機制時，會先暫停住本身執行緒的執行，並將要啟動同步所做為 Object Lock 的 object reference 送到 Data Coherence Controller。而當 Data Coherence Controller 收到要求後，會在 Synchronized

Table 中檢查此 Object Lock 是否已被紀錄。若尚未被記錄則會直接回傳一個回覆給要求的 JAIP 處理核心，而當 JAIP 處理核心收到回覆即代表它能執行此次的同步機制，因而可以繼續往下執行程式。然而若此 Object Lock 已被紀錄於 Synchronized Table 之中，代表已經有其他執行緒已經利用此 Object Lock 啟動同步機制，因此 Synchronized Manager 只會將此次的要求紀錄於 Waiting Table 之中，而並不會回覆給 JAIP 處理核心。等待有執行緒結束同步機制而釋放這個 Object Lock，才會通知可啟動同步機制。

圖 41.為 Synchronized Manager Unit 在執行時有限狀態機的變化，一開始皆處於 Idle 狀態，當有執行緒要啟動同步機制時，會進到 Analysis 來利用所提供的 Object Lock 來查找 Synchronized Table，若已被紀錄則將此次要求的資訊(JAIP 處理核心的編號與所提出的 Object Lock 存放於 Waiting Table，並直接回到 Idle 狀態；若尚未被記錄，則會進入 UpdataTable 狀態來將此 Object Lock 放置到 Synchronized Table 之中，並產生一個回覆給提出要求的 JAIP 處理核心使其能夠繼續執行。而當有執行緒要解除同步機制時，會進入 FreeLock 狀態來找尋 Waiting Table 中是否有正在等待此 Object Lock 的 JAIP 處理核心。若無，在 UpdataTable 狀態時，僅會將此 Object Lock 從 Synchronized Table 中移除；若有，則保留此紀錄於 Synchronized Table 並且根據 Waiting Table 所記錄的處理器編號回傳一個回覆給 JAIP 處理核心，表示可開始啟動同步機制並繼續執行。然而要解除同步機制的 JAIP 處理核心只需要將同步相關資訊傳到 Data Coherence Controller 即可繼續執行程式，不需要等待任何回覆。

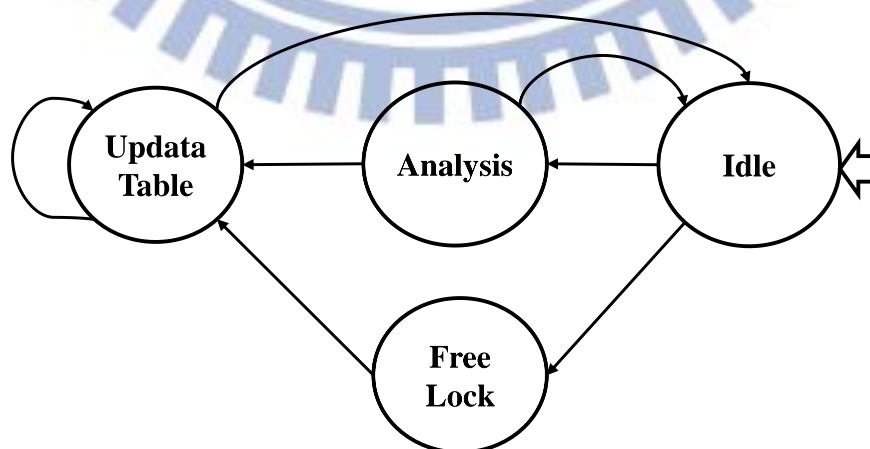


圖 41. Synchronized Manager 有限狀態機

第五章 實驗結果

5.1. 實驗環境

本論文所設計的系統架構實作採用 Xilinx ML-605 SoC emulation platform 之上做為開發平台。此平台上包含了 Xilinx Virtex-6 FPGA (XC6VLX240T)。RISC 處理器是採用 Microblaze，它是種嵌入式平台上由軟體所合成 RISC 處理器。負責去執行系統軟體協助 JAIP 的 I/O 操作以及原生方法。所有設計的 RTL model 是使用 VHDL 程式語言所寫。而 FPGA 的合成軟體是採用 Xilinx Synthesis Technology(XST) 13.1 並使用 Xilinx ISE Design Suite 所提供的 ChipScope 來驗證。然而目前最大工作頻率為 94.2 MHz，而在我們開發平台上設定 83.3 MHz 作為工作頻率，也能滿足 DDR3-SDRAM 的工作頻率。表 5.為 JAIP 處理核心在系統中 FPGA 電路資源的使用量。

表 5. JAIP 合成資訊

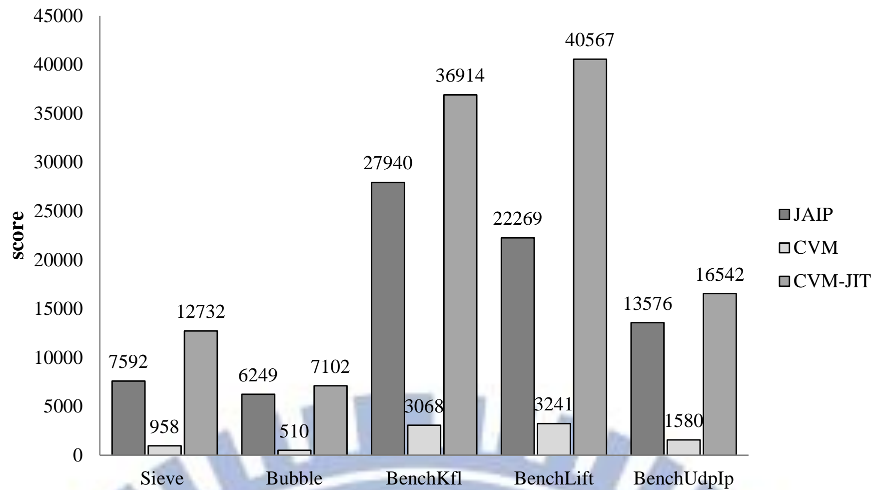
Device: vertex-6 (XC6VLX240T)	
Number of LUTs	12924
Number of Flip-flops	8183
Number of 2K BRAM	30
Maxinum frequency	94.2 MHz

5.2. Benchmark 分析

在評估所設計的 Java 處理器執行效能，我們利用 JemBench 來測試。JemBench 是一個開放原始碼的嵌入式平台 Java benchmark，此 benchmark 包含了絕大部分位元組碼與機制的測試，可區分為 computational kernel benchmark(Bubble、Sieve)、control application benchmark(Klf、Lift、udpip)以及 multi-threading benchmark(Dummy Test、Matrix Multiplication、NQueens)等。同樣的 benchmark 我們也跑在 Sun’s CVM 之上作為比較，CVM 為嵌入式的 Java 虛擬機器執行於嵌入式 Linux 之上。而此作業系統是 porting 於工作頻率同為 83.3 MHz Xilinx ML-405 的 PowerPC 處理器之上。並且在 CVM 上有支援 Just-In-Time (JIT)的軟體加速功能。本章節會先針對在 JAIP 上執行僅有單一執行緒的測試，並與 CVM 跟 CVM-JIT 的執行效能做出比較分析，再來會執行 JemBench 中的 multi-threading benchmark 並將執行緒數量增加，藉以分析在多執行緒的環境之下 JAIP 執行 Temporal Multithreading 的效能。最後是建立一個多處理核心的執行環境，並在每個核心上執行一組執行緒，來達到執行緒平行執行，並藉由處理核心的增加狀況來觀察在執行效能上的變化情形。

5.2.1. Single-Thread 效能分析

在這階段的測試，我們使用 JemBench 來測試當執行環境中只有單一執行緒時的效能評估。測試結果如圖 42 所示，雖然 JAIP 在執行此 JemBench 的程式時，絕大部分效能都低於有軟體加速編譯器的 CVM-JIT，但卻也比單純只使用直譯器(Interpreter)的效能來的高出許多倍。而在效能評比的結果中，JAIP 有一支程式 NQueens benchmark 的分數反而超過 CVM-JIT。這是由於 NQueens 在程式執行的內容上大多數都是進行邏輯運算(例如利用 and 、or 等運算元)。因此這使得單純由硬體來實作 Java 位元組碼運作的 JAIP，在邏輯運算上可以大幅的加速程式的運作，使得執行結果的分數能超越單純以軟體來實作加速編譯器的 Just-In-Time。因此當所要執行的程式內容包含大量的邏輯運算時，使用 JAIP 所執行的效率相較於 Just-In-Time 的加速技術來得較高一點。



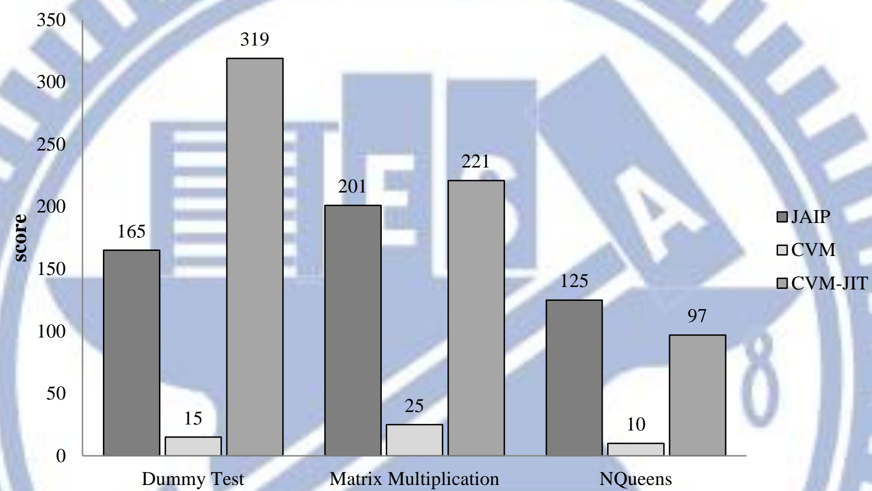


圖 42. 執行 JemBench 單執行緒效能評比

5.2.2. Temporal Multithreading 效能分析

在這階段的測試，我們將執行環境加入多組執行緒。使得 JAIP 可以利用 Temporal Multithreading 的機制去執行。而測試的程式除了 JemBench 內含的三組 multi-threading benchmark 以外，我們還將 CaffeineMark 中的 Logic 測試程式抽出並仿照 JemBench 改寫為 Multi-Logic benchmark 來進行測試。如第 3 章所述，目前 JAIP 的 Temporal Multithreading 機制可支援到 16 組執行緒同時存在於系統之中，因此此節的測試將以執行緒的數量遞增(最多為 16 組)的狀況下，觀察效能的變化。然而此次評比中，JAIP 所用於判斷執行緒切換的時間區段為 100 microseconds.

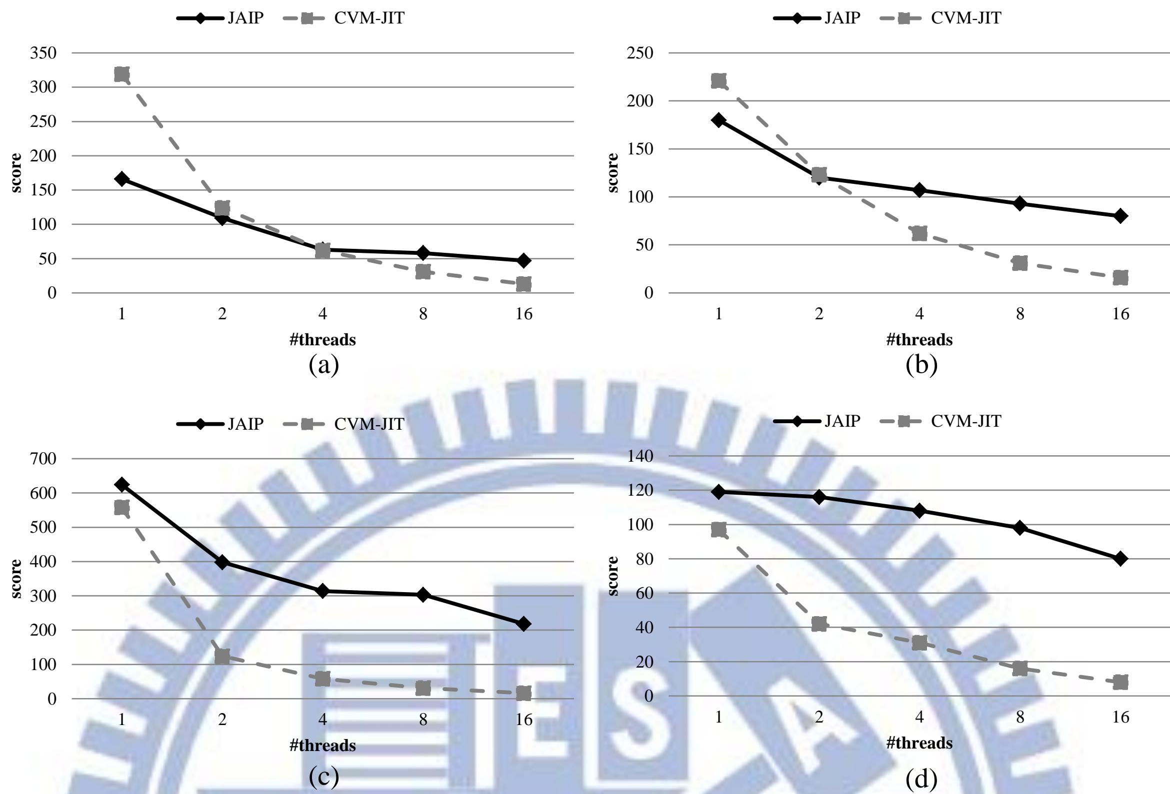


圖 43 . JAIP Temporal Multithreading 機制與 CVM-JIT 效能評比；(a) Dummy Test；(b) Matrix Multiplication；(c) Multi-Logic；(d) NQueens

測試結果如圖 43 所示，原本在單執行緒時 JAIP 在 Dummy Test 以及 Matrix Multiplication 兩支程式的執行效能低於 CVM-JIT。但隨著執行緒的增加，JAIP 在執行的分數上漸漸追上 CVM-JIT 甚至是超越其執行分數。然而在 NQueens 以及 Multi-Logic 由於這兩支程式皆大部分為邏輯運算，本來執行效率就會高於 CVM-JIT，因此在多執行緒的環境之下其執行效能還是較佳。所以由 JAIP 直接於處理核心內實作 Temporal Multithreading 的管理多執行緒機制效能是較為高的。

5.2.3. 多處理核心效能分析

在此章節的測試，我們建立一個多 JAIP 處理核心的執行環境，並在系統中加入 Multi-core Coordinator 元件來管理各處理核心以及協助 Cache Coherence 機制。並且為了讓在 DDR-SDRAM 中的 Object Heap Space 的資料也保持一致則在修改 JAIP 處理核心上

Cache 的資料時，採用 write-through 的機制來將所修改的資料寫回到 DDR-SDRAM 中的 Object Heap Space。而在多核心環境中測試所使用程式為 JemBench 的 multi-threading benchmark 以及所改寫的 Multi-Logic benchmark。在本節將測試 JAIP 處理核心逐漸增加，並只在每個核心上只運作一組執行緒。藉以觀察在處理核心的數量增加下，對於效能所造成的影響並分析。

表 6.多處理核心效能分析

#JAIP-cores / Benchmark(score)	1	2	3	4
Dummy Test	166	330	397	485
Matrix Multiplication	180	278	429	583
Multi-Logic	622	1216	1460	1901
NQueens	118	230	329	413

測試效能結果如表 6 所示，可以明顯地觀察到當處理核心增加時效能也會隨之增加。是由於多組執行緒不再像由 Temporal Multithreading 機制來執行，而使得多執行緒只能在單一核心內使用執行緒輪流執行的方式，然而在此多處理核心的運作環境下可達到平行執行的狀態，因此當核心數量增加時效能也會向上提升。

然而當核心增加時，與單一核心的效能相比下卻不是呈現倍數的成長，因為多個 JAIP 處理核心雖然是平行執行的，但卻因為存取 DDR-SDRAM 必須透過 System Bus 的溝通，以至於各處理核心都必須等到 System Bus 是空閒的狀態才能爭取使用。對效能所造成的影響在處理核心越多的狀況下越加的明顯，因此在多核心的執行環境下在對 System Bus 的使用情形將成為多核心架構下的限制。

第六章 結論與未來展望

在本論文實作了兩種不同的多執行緒機制，Temporal Multithreading 以及多處理核心執行環境。Temporal Multithreading 可以有效的在硬體上管理多組執行緒並進行執行緒的切換執行。對於執行緒所有的操作皆由 JAIP 來支援，而不需要 RISC 處理器的協助，節省了處理器間溝通耗費的成本。而在管理執行緒的資訊上，我們設計了一組 Ping-pong Java stack memory，讓 JAIP 上存有兩組 Java stack，並將執行緒所用的運算堆疊備份於主記憶體。可以當 JAIP 在執行程式時同時也能準備下一個執行緒的所需要之堆疊。因此於處理器上的管理成本可以大幅降低，而以實驗結果來看在切換執行緒執行的效能上有較好的表現。至於多處理核心的執行環境下，藉由在系統中放入多個 JAIP，使得執行緒能夠真正的並行執行，但卻可能會造成 JAIP 處理核心之間的資料不一致。因此設計出 Multi-core Coordinator 元件，其中由 Data Coherence Controller 來協助處理核心之間的資料不一致的問題。並讓 JAIP Manager 來管理多個處理核心。使得多處理核心的執行環境能夠正常的運作。而結果呈現來看當處理核心增加時，效能也會跟著增加。

而在未來的實作上，可能有幾個方向需要去實作。首先是修改關鍵路徑(critical path)，並試著提高 JAIP 工作的頻率。第二是加入不同的排班機制，由於我們目前對於執行緒的排班採用的是 Round-Robin 的方式，然而可加入其他執行緒排班的機制，例如優先權的管理機制，使得當有些執行緒希望被優先執行時，在目前的機制中還是要進入 Round-Robin 的排班機制中。第三是設計並加入在 Java 程式語言中對多執行緒管理的相關函式(notify()、notifyAll()等)，可藉由 3.3.4 節所提出的介面於處理器上直接控制執行緒狀態。最後是採用已經實作 Temporal Multithreading 機制的 JAIP 建立多處理核心的執行環境，在每個 JAIP 處理核心上皆可以存在多組的執行緒，而不再是每個處理核心只能單獨處理一組執行緒，可提高在系統中所支援的執行緒數量。

參考文獻

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

[2] J.M. O'Connor and M. Tremblay, “picoJava-I: the Java virtual machine in hardware,” Micro, IEEE, vol. 17, no. 2, pp. 45-53, Mar./Apr. 1997.

[3] 3. Sun Microsystems, J2ME Technology, Sun Developer Network URL: http://java.sun.com/javame/technology/, 1994-2009.

[4] C.-M. Chung and S.-D, Kim, “A dualthreaded Java processor for Java multithreading,” In Proc. IEEE on Parallel and Dist. Sys, pp. 693-700, 1998.

[5] J. Kreuzinger, et al, “Real-time event-handling and scheduling on a multithreaded Java microcontroller. Microprocessors and Microsystems,” , 27.1: pp.19-31, 2003.

[6] C. Pitter and M. Schoeberl, “Towards a Java multiprocessor,” In Proc. of the 5th ACM int. workshop on Java technologies for real-time and embedded systems, pp. 144-151,2007.

[7] A. Wellings and M. Schoeberl, “Thread-local scope caching for real-time Java, IEEE ISORC'09, pp. 275-282, 2009.

[8] M. W. El-Kharashi and F. Elguibaly, “Java Microprocessors: Computer Architecture Implications,” Proc. of IEEE Pacific Rim Conf. on Comm., Computers, and Signal Proc., vol. 1, pp. 277-280, Aug. 1997..

[9] K. B. Kent and M. Serra, “Hard/Software Co-Design of a Java Virtual Machine,” Proc. of IEEE Int. Workshop on Rapid Systems Prototyping (RSP), June, 2000.Sun Microsystems, Connected, Limited Device Configuration Specification, ver. 1.0a, Sun Microsystems White Paper, May 2000.

[10] White, James. "An introduction to Java 2 micro edition (J2ME); Java in small things. “ Proceedings of the 23rd international conference on Software engineering,” IEEE Computer Society, 2001.

[11] Bill Venners, Inside the Java 2 Virtual Machine, New York: McGraw-Hill, 2001, ch.5 ch.6 ch.7 ch.8.A. Krall, “Efficient Java Just-in-Time Compilation,” Proc. of Int. Conf. on Parallel Architectures and Compilation Techniques, pp. 205-212, Paris, Oct. 1998.

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

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

[14] 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.

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

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

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

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

[21] A. Krall, “Efficient Java Just-in-Time Compilation,” Proc. of Int. Conf. on Parallel Architectures and Compilation Techniques, pp. 205-212, Paris, Oct. 1998.Sun, picoJava-II Microarchitecture Guide, Sun Microsystems, March 1999.

[22] 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.H. McGhan and M. O’Connor, “PicoJava: A Direct Execution Engine for Java Bytecode,” Computer, Vol. 31, Issue 10, pp. 22-30, Oct. 1998.

[23] U. Brinkschulte, C. Krakowski, J. Kreuzinger, and T. Ungerer, “A Multithreaded Java Microcontroller for Thread-Oriented Real-Time Event-Handling,” Proc. of 1999 Int. Conf. on Parallel Architectures and Compilation Techniques (PACT’99), pp. 34-39, Newport Beach, Oct. 1999

[24] M. Schoebel, “Evalution of a Java Processor,” Tagungsband Austrochip 2005, pp. 127-134, Oct. 2005.

[25] H.-J. Ko and C.-J. Tsai, “A Double-issue Java Processor Design for Embedded Application,” Proc. of IEEE Int. Symp. on Circuits and Systems(ISCAS’08), Seattle, May. 2007.

[26] H.-J. Ko, “A Double-issue Java Processor Design for Embedded Application, ” Mater thesis, NCTU, 2007.

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

[28] H.-W. Kuo, “Design of Java Accelerator IP for Embedded Systems, ” Mater thesis, NCTU, 2011.

[29] Z.-J. Guo, “Design of Dual-Core Java Application Processor for Embedded Systems, ” Mater thesis, NCTU, 2012

[30] . Z.-G. Lin, H.-W. Kuo, Z.-J. Guo, and C.-J. Tsai, “Stack Memory Design for a Low-Cost Instruction Folding Java Processor,” IEEE ISCAS, May, 2012.

[31] Z.-G. Lin, “Design of stack memory device and system software for java accelerator IP, ”Mater thesis, NCTU, 2011.

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

[33] Ungerer, Theo, Borut Robič, and Jurij Šilc. “Multithreaded processors,” The Computer Journal 45.3 pp.320-348, 2002.

[34] Tullsen, Dean M., et al. “Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor,”ACM SIGARCH Computer Architecture News. Vol. 24. No. 2. ACM, 1996.

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