# 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：吳宗漢

指導教授：蔡淳仁　教授

**中 華 民 國　103 年　6　月**

# 多執行緒 Java 處理器設計

# Design of the Multithreading Architecture for a Java Processor

研 究 生：吳宗漢　　Student：Tsung-Han Wu

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 2014

Hsinchu, Taiwan, Republic of China

中華民國 103 年 6 月

# 摘要

多執行緒在 Java 平台內是不可或缺的功能，例如 web-browser 內需要管理不同子視窗的連線、或者檔案下載軟體可能使用多個執行緒同時載入同個檔案內不同的 data block。在本論文中我們將以 Java Application IP (JAIP)為基礎，提出一個 Multicore multithreaded Java processor Architecture，藉由同時執行多個 Java 處理器並且每個 Java 處理用上啟用 temporal multithreading 機制，達到每個處理器資源使用最佳化並且能容納更多執行緒，為了在我們提出的設計之中解決 thread 的分配與同步問題，我們也提出新架構用以協同每個Java處理器運作並且處理程式執行的一致性。此架構在 Xilinx ML-605 FPGA 平台上驗證。實驗結果顯示，我們提出的架構比單一 Java 處理器啟用 temporal multithreading 的執行效能有顯著的提升，並且將大幅縮減電路資源使用。

# 誌謝

這篇論文的完成，首先必須要感謝我的指導教授蔡淳仁教授。在研究所的這兩年期間，除了老師提供了很多學習的經驗跟機會外，在與老師討論的過程中，我也了解到何謂邏輯的謹慎推演以及細節的注意。也感謝實驗室的同學，當我們一起進行研究時，讓我了解到自己仍有許多地方思考不夠嚴謹，也學習到如何清楚表達自己的想法給別人，以及如何與別人的溝通中提取重點。很高興在實驗室的同學願意和我一起努力，讓我自己在這兩年間成長了許多。感謝 MMES LAB 的大家。

# 目錄

# 圖目錄

# 表目錄

# 第一章 前言

## 1.1. 研究動機

近年來 Java 因為其可移植至不同作業系統與處理器的特性，而成為嵌入式平台的主流開發語言之一，其中一個知名範例是 Android 作業系統，其 Application Framework 的部分主要使用 Apache Harmony 這組 library 開發。Java 也提供許多 API (Application Programming Interface)以支援軟體專案的開發，其中一項重要應用為提供程式介面以支援多個 thread 並行執行，Java VM 規格書與 API 皆說明 thread 管理時使用的語法以及造成的效果，然而許多 multithreading 相關操作必須仰賴底層硬體架構或作業系統完成 thread 的執行資訊維護，因此設一個有效率的 multithreading 管理機制勢必為一項重要議題。

針對 Java multithreading 的設計，可簡單區分為由軟體或硬體支援：由軟體所支援的 Java 執行環境包括 Sun Microsystems 提出的 Java VM，以及可在嵌入式平台上執行的 CVM，對於 thread 的管理與同步機制運作大多透過直接執行 Java Bytecode 指令以及透過 native methods 來支援，而 Java 規格將 native method 實作細節留給底層作業系統決定，此種做法有兩項限制：(1)Time slice 數值通常以 millisecond 為計算單位，例如 Unix-like kernel 的 time slice 大約在 10 milliseconds，此項限制在某些領域會造成嚴重問題，例如可穿戴式電腦，當使用者有任何動作時，裝置的處理器必須能快速取得個別 sensor 收集到的資料以減少回應時間 (2)Context-switching 執行時間，thread context 內容取決於處理器與作業系統架構。一般 RISC core 處理器底下每一個 thread context 包含一組 CPU registers、program counter 與被分配到的 main memory space；Java thread 的 context 除了少數 special-purpose registers 之外還包含 stack frames，當執行 context-switching 時由作業系統觸發 interrupt 訊號到 interrupt service routine 切換 current thread 與 next ready thread 的 context，此過程至少需要 500 個 clock cycles 以上。

由硬體所支援的 Java 處理器環境包括 Pico-Java、KOMODO、JOP 等，主要特

性為不需要底層作業系統支援、直接由電路管理多個 Java thread 的 context，這表示開發者可以透過各種方式達到較少的 time slice 與 context-switching 執行時間。由於 Java thread context 的大小比一般 RISC core 的 thread context 高，因此設計一個節省硬體資源的 thread 管理方法是我們的研究重點。

近年來多核心處理器架構在嵌入式系統應用上日漸廣泛，在多核心 Java 處理器方面，雖然 JOP[24][28]已經有提出多核心處理器的研究但是仍然有效能的問題。為了實作多核心 Java 處理器架構並且每個處理器支援多個 threads，在 Java 執行環境中必須確認應用程式執行結果的一致性，設計有效率的同步機制協同多個 threads 並行執行也是我們的研究重點。

## 1.2. 研究目的與貢獻

本篇論文以一個異質雙核心 Java Application IP (JAIP)應用處理器[1][2]為基礎，提出一個 Multicore Multithreading Java Processors 架構。JAIP 為一個具高度可移植性的 Java IP core，負責處理所有 Java Bytecode 的執行，並且支援字串處理與 Temporal Multithreading 機制，為了讓多核心 JAIP 環境彼此間能夠正常執行，在[1]的研究中提出 Multi-core Coordinator 負責處理 thread synchronization 與 cache coherence 問題。

先前 JAIP 的 Temporal Multithreading 管理電路之中使用大量 registers 暫存 thread context 之中所有的 special-purpose registers 因此造成硬體資源使用量大增。另外在原先多核心 JAIP 處理器架構下[1]單一 JAIP 內部不存在 Temporal Multithreading 管理電路，意即單一 JAIP 只能容納一個 thread，假設其中一個 JAIP 的 current thread 取得 lock object 失敗時，Multi-core Coordinator 的實作上只需要讓這個 thread 暫停執行指令，因為單一個 JAIP 內只有一個 active thread 所以無法切換到下一個 ready thread 執行，導致這個 JAIP 閒置的情形發生。

為了讓 Java 支援底層硬體操作，[2]提出一個讓 JAIP 的 Bytecode Execution Engine 直接透過 Dynamic Resolution 機制來操控硬體設備或電路之介面—Hardware

Native Interface。Java 程式語言可以透過這個介面觸發 Temporal Multithreading 機制並執行 thread 相關管理工作。

本論文研究重點在於提出方法以支援多核心 JAIP 處理器環境同時整合 Temporal Multithreading 與 Hardware Native Interface 機制，為了達到此目的，我們也提出方法改進 Multi-core Coordinator 之中分配 thread 與同步功能，並且更改其名稱為 Data Coherence Controller，使單一 JAIP 處理器底下能容納更多 threads，並且最佳化 thread 管理的電路資源使用。

## 1.3. Java 執行環境

Java 執行環境是由 Java VM 與一組標準類別庫所組成，在 Sun Microsystem 所推出的 Java Platform, Micro Edition 應用範圍涵蓋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)[4]規格之中，另外功能較為強大的裝置則規範為 Connected Device Configuration(CDC)[5]規格。

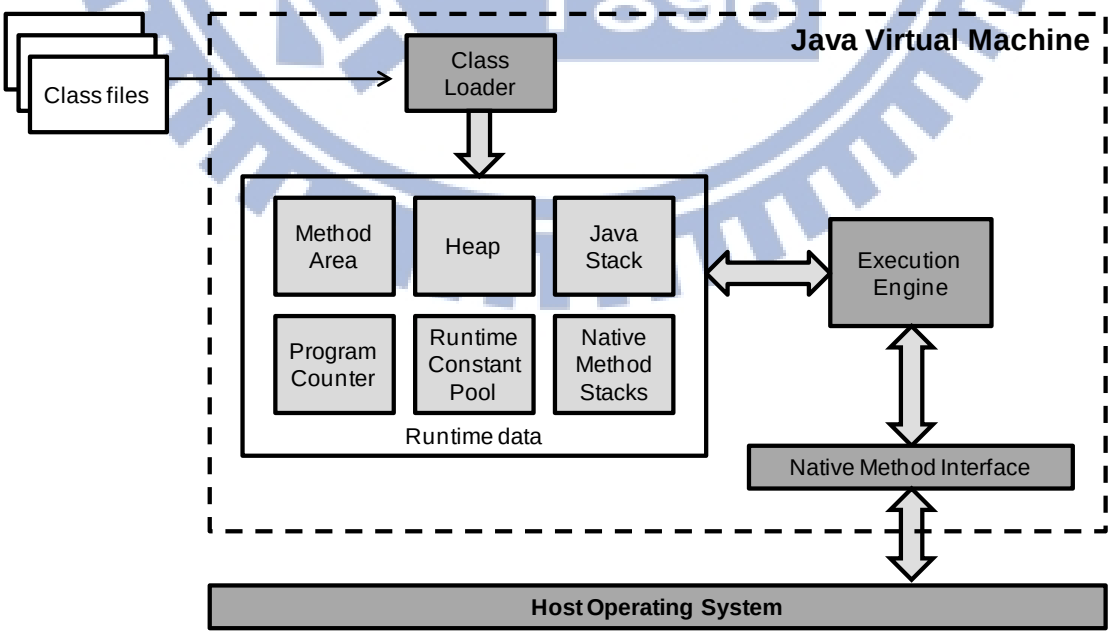



圖 1 傳統 Java 執行環境

傳統 Java 執行環境(如圖 1)包含軟體 Java VM，它依賴完整的作業系統去執行 Java 應用程式。Java VM 中有三個重要元件，Class Loader、Runtime Data 與 Execution Engine[6]。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 VM 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 VM 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[8]與 KVM[9]就是在嵌入式裝置中利用軟體實作 Java VM 的直譯器[10]，但是直譯器在執行 Java 程式時相當沒有效率。因此大部份 Java 執行環境會使用加速技巧優化執行效率。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 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[7])、協同式的 Java co-processor，例如 ARM Jazelle[11]與 Nazomi JSTAR[12]、standalone Java processor，例如 Sun picoJava [13][14][21]及 aJile aJ-102 [15][16]。

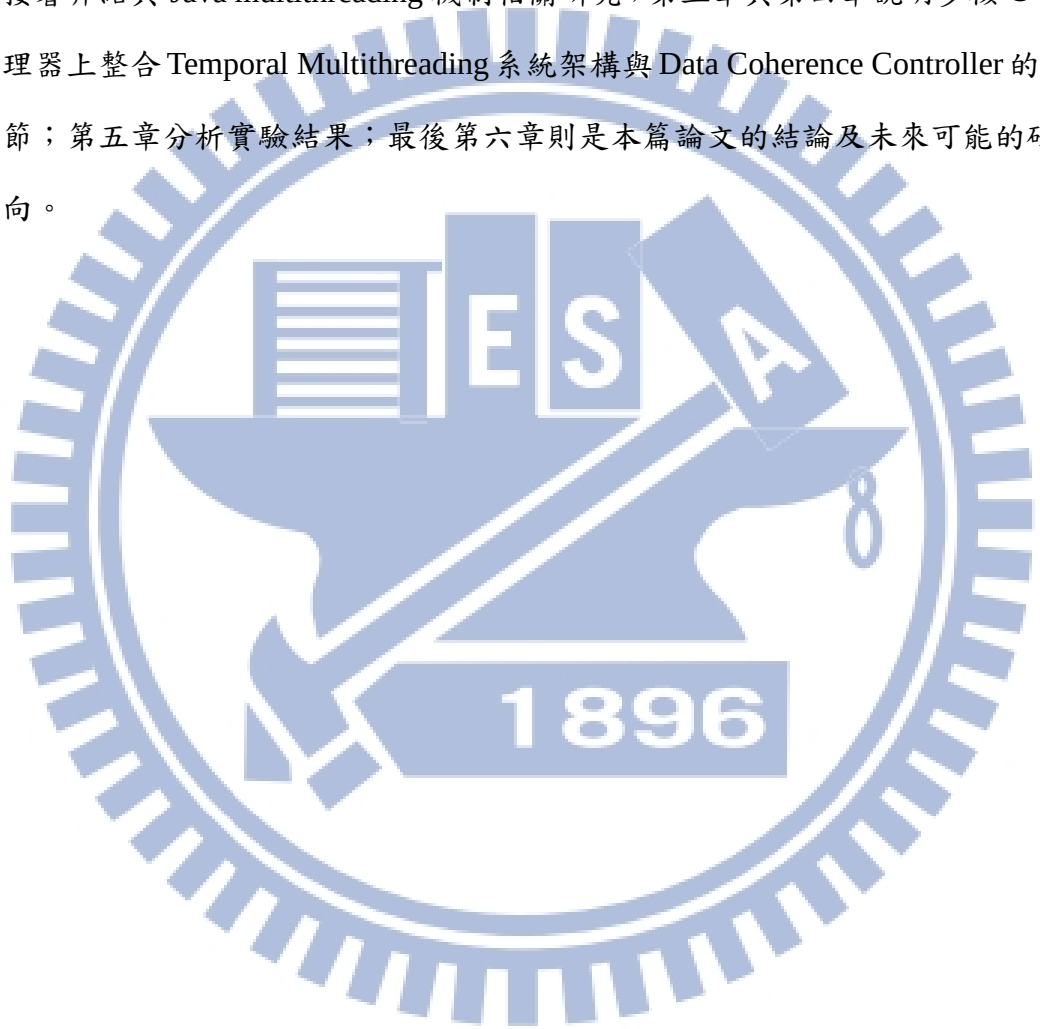
利用 Just-in-Time 編譯器的技術藉由執行時期將部分 Java bytecode 轉換成 native code[17]並暫存於系統之中，可顯著的提高執行效率，但 Just-in-Time 需要額外的記憶體空間和在 Class 被載入時所附加額外編譯時間。因此在對記憶體資源較為嚴格要求的嵌入式應用設計較不適合。另一提高 Java 執行效率的方法為協同式的 Java co-processor。像是具備內嵌轉譯器（Embedded Java Translator）的 ARM Jazelle 及 Nazomi JSTAR，其具備可以切換執行 Java bytecode 的處理器設計、或像是 standalone Java processor：Sun picoJava、Komodo[16]及 JOP [18]都可單獨執行 Java 程式。這些處理器皆是依照 Java VM 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。

JAIP 採用易於與其他處理器整合的弱協同式設計(weakly-coupled co-design)，只要 RISC 處理器支援 interrupt-driven 之 communication 便可與 JAIP 整合。JAIP 可以獨力完成幾乎所有的 Java 程式計算，只有程式需要低階 I/O 裝置控制例如 RS-232 controller，或從 Compact Flash Card 載入 class file 並且解析，才會需要使

用 RISC 處理器計算。

## 1.4. 論文架構

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，會先介紹在本論文中作為設計基礎的 JAIP 先前研究，並接著介紹與 Java multithreading 機制相關研究；第三章與第四章說明多核心 JAIP 處理器上整合 Temporal Multithreading 系統架構與 Data Coherence Controller 的設計細節；第五章分析實驗結果；最後第六章則是本篇論文的結論及未來可能的研究方向。



# 第二章 相關研究

此章節先說明 Java 執行環境並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 multithreading 機制的相關研究。

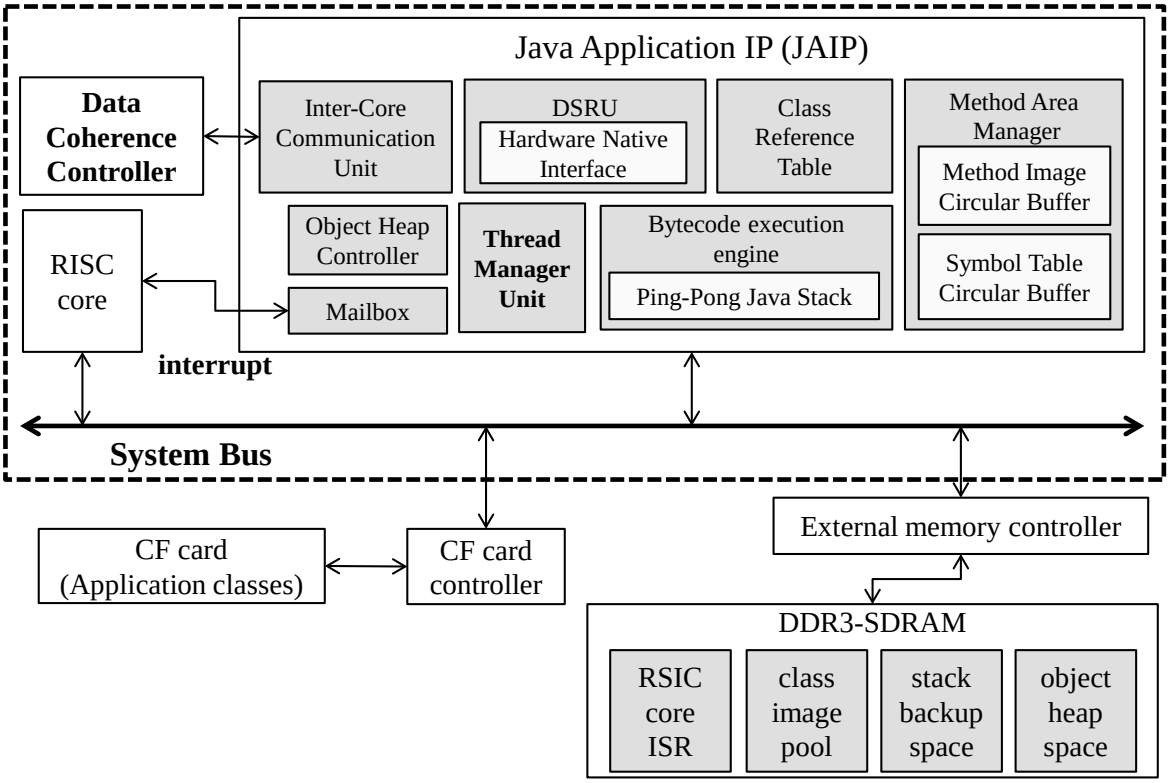



圖 2 The block diagram of JAIP

## 2.1. Previous Work on JAIP

本論文所採用的 JAIP 處理器架構[26]如圖 2 所示，包含 Bytecode Execution Engine(BEE)負責直接執行 Java bytecode 指令，部分指令運算結果、local variables 以及 return frame 被暫存到 Ping-Pong Java Stack；而 object fields 與 class fields 則被暫存到 Object Heap Space 與 2-way Set Associative Heap Cache Controller。根據 Java instruction set 定義，某些 bytecode 指令其後的 operand bytes 用來指向 class file 的 constant pool 特定位址，此時便需要 Dynamic Symbol Resolution Unit(DSRU)作符號解析的工作，每次解析到的新的 Class 則由 Method Area Manager 底下的 Class Symbol Table Controller (CST Controller)與 Method Area Controller (MA Controller)負責載入 constant pool data 與 method

bytecode，以及 DSRU 執行時會用到的 field info 與 method info 也將被寫入到 Cross Reference Table。另一個核心為 RISC-core，主要負責 class loading and parsing 和 serial port I/O 管理等工作。

BEE 是一個獨立的模組。它可以整合到任何的 host 端處理器，只要 host 端支援 interrupt-driven 之 inter-processor communication 機制。BEE 是一個 4-stage pipeline，包括 translate、fetch、decode 及 execution stages。Translate stage 透過 jpc 將 Method Area 內的 Java Bytecode 讀出，每次從 Method Image Circular Buffer 讀出兩個 byte 資料，每個 opcode 皆會經過 translation ROM 轉換為一對 j-code info.。j-code info.被送進 fetch stage 後，被分類成為 simple 及 complex 兩種類型 bytecode，若為 simple bytecode，會被轉換成一個 BEE 指令—j-code，兩個無 hazard 之 jocde 可以被 double-issued；若為 complex bytecode，會被轉換為一組 j-code-pair sequence。每個 j-code pair 會經由後面的 decode stage 和 execution stage 執行。為了讓指令平行度增加，[3]提出了 2-level stack memory 架構(圖 3)，為了支援不同 thread 之間快速切換 stack frames，[1]擴充 2-level stack memory 實作 Ping-Pong Java Stack(圖 4)。

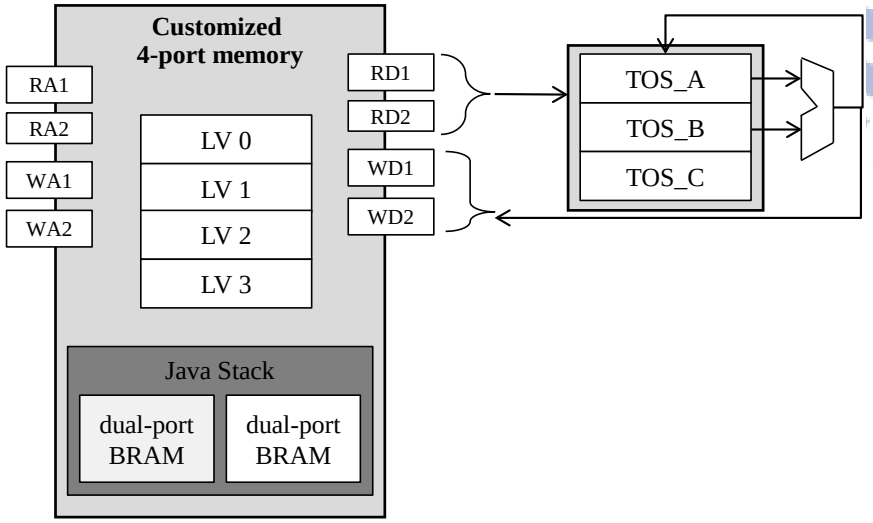



圖 3 The block diagram of 2-level stack memory

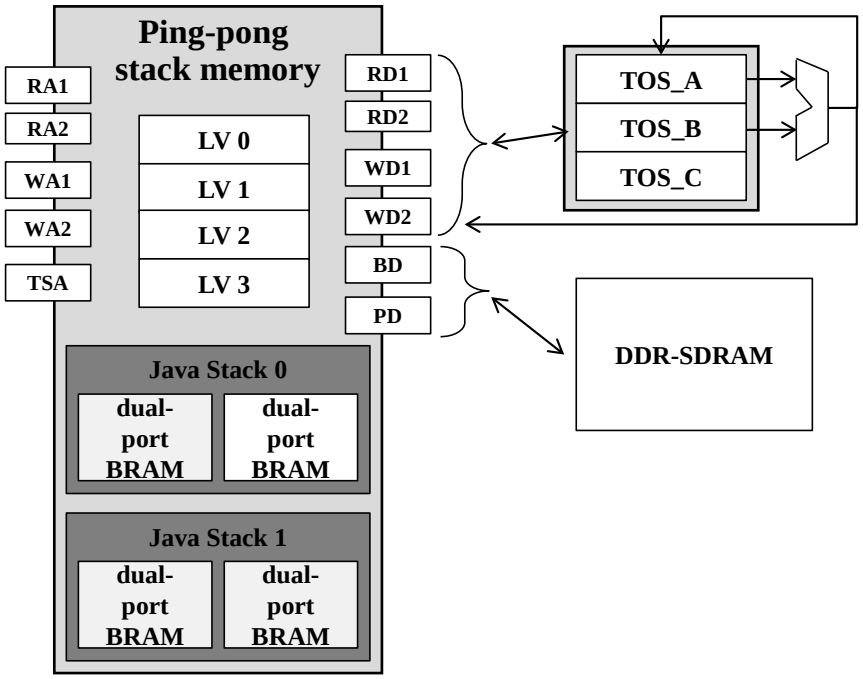



圖 4 The block diagram of Ping-Pong Java stack

DSRU 負責在 runtime 時解析 Symbol Table Circular Buffer 中每個 entry 的資訊、協助觸發額外加速電路[2]與 class loading 請求、傳遞 class ID 和 method ID 到 CST Controller 和 MA Controller 中。Symbol table 在一般程式語言中，是一個專門儲存 identifier 資訊之資料結構，可供 compiler、interpreter 或 linker 查詢變數型態、scope 或函式進入點。在我們的 JAIP 平台中，Symbol Table Circular Buffer 存放著 ldc bytecode 所用的 constant 資訊、new bytecode 需要的 class ID、field 操作和 invocation 時都需要的 Cross Reference Table 之 entry reference。

Cross referece table 內的結構則是由一個 entry 表示的 field info.，或是由 2 個 entries 組成的 method info。在 BEE 遇到 field 操作、load constant 或 method invocation 類型的 bytecode 指令時，會啟動 DSRU 來解析 Symbol Table Circular Buffer 與 Cross Reference Table 之內容。

## 2.2. 多執行緒與同步處理相關研究

其他 Multithreaded Java processors 相關研究中，[22]提出的 Java 加速電路包含 CPU

portion 與一個 thread lifetime unit 電路負責 active threads 管理與排班機制，並使用一個 timer 倒數紀錄 current thread 的執行時間，一旦 timer 的值小於等於 0，thread lifetime unit 會觸發訊號給 CPU portion，此時 current thread 在 CPU portion 下的 register sets(包含 current thread 的 operand stack values 與)將被清除並且備份到 memory，並且載入 Java Virtual Machine 到 CPU portion 執行 thread 排班。

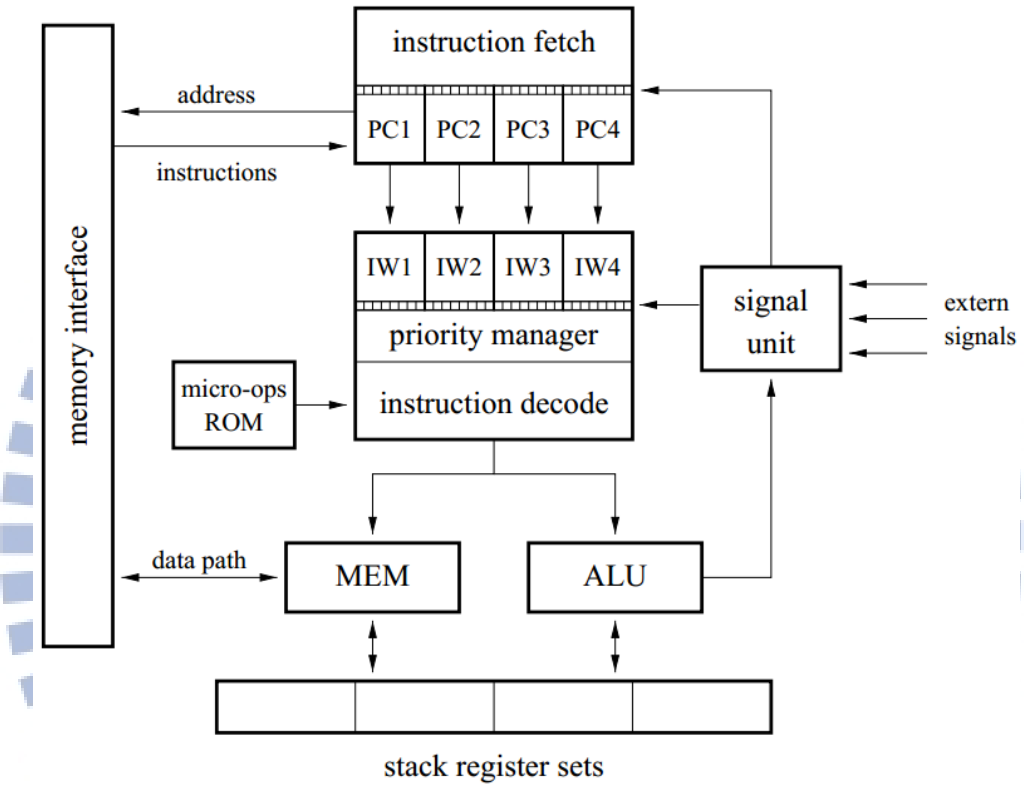



圖 5 The block diagram of KOMODO microcontroller

KOMODO[16]使用 4 個 thread slots 儲存每個 thread 執行資訊包括 program counter、Instruction window、一組 stack registers (圖 5)，每次從 Memory Interface 依序載入一組 4-byte instruction package 到 Intrsuction windows，此時這 4 個 instruction windows 可能儲存 0 到 4 個 opcode，接著由 Priority Manager 檢查每個 thread 的 characteristic value[19]判斷哪個 thread 對應的 instruction window 優先被執行並傳到 MEM 或 ALU。其中 characteristic value 為 Priority Manager 的 internal register 並且可視為 thread state，包含這個 thread 是否在等待 Atomic lock、以及這個 thread 的指令是否在執行 external memory

相關操作。在 KOMODO 處理器中，I/O 操作完成時(e.g. 使用外部 timer 計算每個 task 的執行時間、memory-accessing)會傳 external signal 並且觸發 Signal Unit，使 Priority Manager 修改 characteristic value。如果在此架構下要擴張 thread 的數量則容易造成電路資源使用過高。

Jamuth[20]以 KOMODO 為基礎，並且提出一些改進的設計(圖 6)例如：所有 hardware threads 共享一個 2k-entry stack cache、採用 4-KB direct-mapped instruction cache 存放部分 method bytecode 以減少載入指令的 latency。但是 2k-entry stack cache 被切成 4 個部分分別存放 4 個 threads 的 stack frames，此作法同時只能有一個 thread 執行指令並且存取 stack cache 而其他 threads 只能將個別 stack frames 搬入 stack cache 或搬出到 external memory。此作法同樣會增加電路的複雜度。Jamuth 皆以 complex trap routines 實作 thread 同步機制例如 Java 指令 monitorenter[6]因此會有比較高的 synchronization overhead。

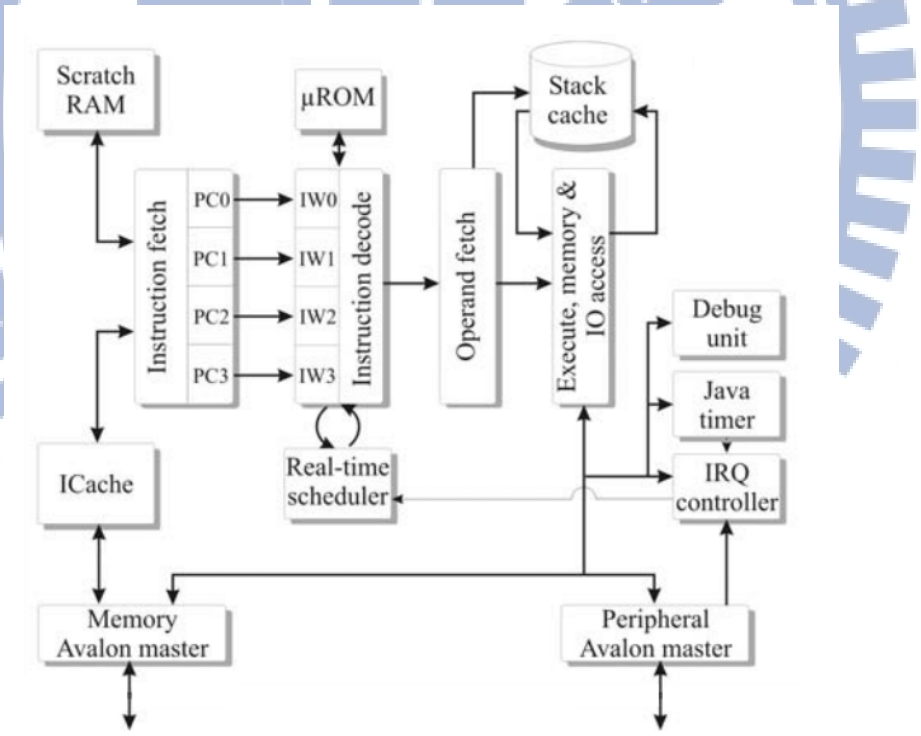



圖 6 The block diagram of jamuth

aJile aJ-102[16]包含一個 execution unit、Unified I-cache 與 D-cache 與 32KB microcode RAM，aJ-200 以 extended bytecode instruction 方式支援直接執行 thread 管理與同步功能，此作法也使用極少的硬體成本。aJ-200 官方文件指出 context-switching overhead 少於 1

microsecond。另外 aJ-200 提出 MJM (multiple JVM manager)功能，可支援 2 個 JVM 同步執行 2 個獨立的 Java 應用程式，每個 JVM 被分配一個內部 timer 用來計算 current thread 的 time slice，MJM 底下可以個別設定每個 JVM 的 thread 排班演算法。

PicoJava-II [13][21]之中每個 thread context 包括一組 registers，用來記錄以下資訊：紀錄 current method 的 base address、stack 頂端元素的位置、local varaible 起始位置、每個 thread 被分配到的 external memory address range、constant pool 的 base address、program status register、program counter 等。picoJava 採用 interrupt 的方式執行額外 context-switching code，切換 current thread 與 next ready thread 的 context，並且需要額外軟體程式支援 thread scheduling，此做法會導致 context-switching overhead 增加。

picoJava 支援 hardware synchronization 機制，同時可執行最多 2 個 lock object 權限取得或釋放的工作，在執行同步機制操作時會使用特定用途的 registers 暫存 lock object 的參考位置、目前是否存在其他 waiting thread、以及 lock object 擁有者重複取得 lock 的次數。在 memory 之中每個 object header 皆包含 lock bit 用來判斷每個 object 是否已被某個 thread 取得權限。[21]提出在 Java object 內維護一個 pointer 指向自行定義 Java class 資料結構，用以儲存每個 lock object 相關 waiting threads。此作法雖然使用極少的硬體成本，但是執行同步操作時維護 waiting threads 會增加 object fields 存取次數，執行時間將會增加。

在多核心執行環境下，Java Optimized Processor(JOP) [24][28]使用最多 8 核心 Java 處理器加上 shared memory[23][27](圖 7)。而每一個處理器上都有一組 stack cache 與 method cache，分別使用 on-chip memory 實作並且自行定義 microcode 執行堆疊操作。當開始執行 Java 程式時，多核心 JOP 環境的其中一個處理器(假設為 JOP0)負責初始化整個系統的工作，完成初始化之後開始執行 bytecode 指令，每個處理器會被分配一個 ID number，當系統有新的 thread 被產生，一律由第一個處理器(JOP0)分配 ID number 與 threads 給其他處理器，在此架構下不支援 thread migration。每個處理器輪替執行不同 threads 並且以內部 timer 檢查執行時間，當每個 JOP 處理器需要執行 context-switching 時，藉由 local timer interrupt 觸發 thread scheduler，thread scheduler 由 Java 程式實作並

且執行 thread scheduling 與 2 個 threads 的 stack frames 切換，由於 JOP 以 on-chip memory 實作 stack cache 因此 threads 使用的 stack frames 數量將決定資料來回搬移到 main memory 的時間，也影響 context-switching 整體時間。Hardware synchronization 方面，Arbiter 依序處理每個 JOP 處理器對 shared memory 的 read/write operation，Synchronization unit 只包含一個 global lock，所有 threads 進入 synchronized method 或 synchronized code block 之前必須先到此模組取得 global lock，當每個 JOP 處理器取得 lock 失敗之後則必須等待，此時在此 JOP 處理器下無法切換到另一個 ready thread 繼續執行 bytecode。

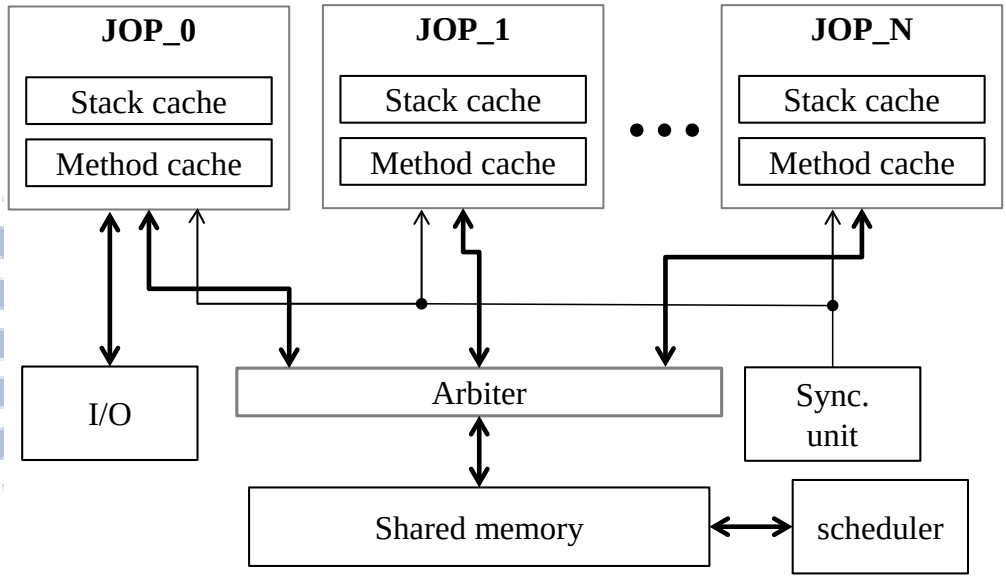



圖 7 CMP system based on JOP

# 第三章 Multicore Multithreading Java 處理器架構

Java 語言定義 java.lang.Thread class 使得任何 Java 平台上可以管理多個 threads 同步執行應用程式。任何 Java 平台開始執行應用程式前會先取得 main method 相關資訊，依照不同平台實作方式會先解析必要的 Java Class 並且載入到 method area、初始化 heap 與 program counter[6]、接著產生 main thread 且開始執行 main method 的 Bytecode 指令，如果 main thread 執行過程中產生多個 threads，此時依照不同 Java 平台實作決定以下事項：(1)多核心 Java 處理器環境下該如何記錄並且分配 new thread 的執行資訊到哪一個處理器 (2)單一處理器下，當 current thread 結束執行或者取得 lock object 失敗時，如何切換到另一個 thread 執行 (3)多核心 Java 環境下，2 個不同處理器的 threads 同時修改 shared data 時造成的 race condition 問題 (4)當某個 thread 釋放 lock object 之後，必須決定下個取得此 lock object 的 waiting thread。

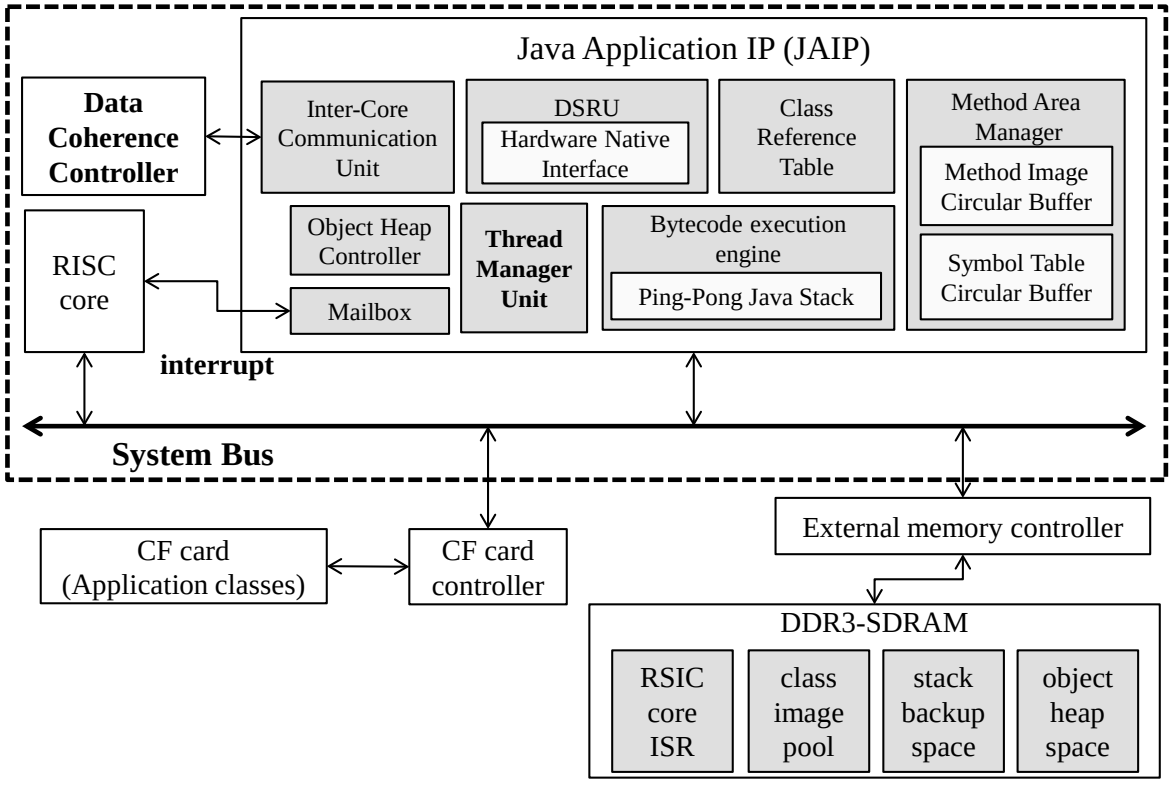



圖 8. The block diagram of JAIP

本章節我們將以先前 Java Application IP (JAIP) 的單核多執行緒架構 [1][2]為基

礎，介紹 Multicore multithreaded Java 處理器架構(單一 JAIP 核心架構，如圖 8)，藉由多個 JAIP 處理器同時執行且每一個 JAIP 啟用 temporal multithreading 機制，達到多個 threads 真正同步執行並且能容納更多 threads。為了使我們提出的架構能順利執行，我們將修改之前 Thread Manager Unit 對以下功能的實作。

1. Java 語言的 Thread class 提供 Thread.start() method 作為產生 new thread 的方式，當 current thread 呼叫 start method 時，Java VM 透過內部機制產生 new thread 的相關執行資訊例如 stack pointer、program counter、current class ID/method ID 等，由 Java VM 決定何時要切換到這個 new thread 並且執行其第一個 method：Thread.run()，當每一個 thread 在 run method 執行 return 指令時，表示這個 thread 結束執行。當 start method 被呼叫後，如何管理這些 threads 執行資訊?如何設計 context switch 機制抓取下個 ready thread?這些將留給個別 Java 平台實作。
2. Java VM 定義 monitor 來協調多個 threads 執行時遇到的同步問題，每一個 Object 只包含一個 lock，當多個 threads 要進入 critical section 之前，必須先從 monitor(Object) 取得 lock，此時必定只有一個 thread 成功取得 lock 且進入 critical section，而其他 threads 必須等待。當一個 thread 已取得 lock object 時可以在 critical section 內執行程式，直到這個 lock object 被釋放且分配給這些 waiting threads 的其中一個，如果不存在對應的 waiting threads 則該 lock 歸還給 monitor。

   為解決同步執行問題，Java 語言提供 Synchronized 保留字，其使用方式如圖 9 與圖 10，修改 shared data (field shared1、shared2)的程式碼被放在 Synchronized statement 或 Synchronized method，一旦任何 thread 要更改 shared data，都必須呼叫 Synchronized method A2、A3，或者執行 Synchronized(lock){ ..... 這段程式。將圖 9 的 class B 編譯過後，method B2、B3 的 bytecode 會包含 monitorenter/monitorexit 指令[6]，這兩個指令可被視為每段 critical section 的第一個與最後一個指令。以圖 10 為範例，假設 thread t1 呼叫 method B2 並且進入 critical section 之前，t1 的 Java stack 的 top element 會存放對應的 object reference (e.g.由前一個指令 push 到 stack 上)，接著 t1 執行 critical section 的第一個指令 monitorenter，在 Java VM 內部取得 lock

object，如果 Java VM 決定將 lock 權限交給 t1，則 t1 才可以進入 critical section 往下執行指令，否則必須暫停執行直到其他 thread 釋放 lock 並且將 lock 權限轉移給 t1。當 t1 離開 critical section 時，同樣在 Java stack 的 top element 會存放對應的 object reference，此時執行 monitorexit 指令以進行 lock 權限轉移或歸還的工作。

```
Class A{
        int shared1 = 0;
        int shared2 = 0;

        public A (){
        }

        public synchronized void A2 (int r){
                shared1 += r;
        }

        public synchronized int A3 (int r){
                shared2 = shared2 * r;
                return 1;
        }
}
```

```
Class B{
        int shared1 = 0;
        int shared2 = 0;
        Object L0;

        public B (){
                lock = new Object ();
        }

        public void B2 (int r){
                r = r * r;
                synchronized(L0){
                        shared1 += r;
                }
        }

        public int B3 (int r){
                r = r << 1;
                synchronized(L0){
                        shared2 = shared2 * r;
                }
                return 1;
        }
}
```

圖 9. Synchronized method 與 Synchronized Statement 範例

當 t1 取得 object A 或 B 的內部 lock ，此時 t1 有權限重複取得相同的 lock，Java VM 對每一個 Object 會維護一個內部 counter 記錄目前每個 lock 擁有者取得此 lock object 的次數，如果 lock object 的 counter=0 表示任意 threads 都有機會取得這個 lock object；如果 t1 多次重複取得 lock object，則 lock object 對應的 counter 會逐次累加，這表示之後 t1 必須釋放相同的 lock object 達到相同次數後 (每釋放 1 次 lock object，counter 遞減一個單位)，直到 lock object 的 counter=0 才算是把 lock 真正釋放並且把權限轉移給其他 waiting threads。monitorenter/monitorexit 被執行時，個別 Java 平台必須提供 monitor 相關實作機制、包括如何紀錄 waiting threads、ready threads、lock object 狀態 (e.g.目前 lock 擁有者、對應的 counter)、決定 thread 是否

成功取得 lock object、lock object 權限轉移或歸還。另外，圖 9 的 class A 編譯後，method A2、A3 的 bytecode 並不會包含 monitorenter / monitorexit 指令，這表示 synchronized method 相關實作由個別 Java 平台自行決定。

```
Class TestTH extends java.lang.Thread
        B obj_b = new B();
        A obj_a = new A();

        public void run (){
                ….
                While(…){
                If(…){
                        obj_a.A3(37);
                        obj_b.B2(885);
                }
                Else if(…){
                        obj_b.B3(701);
                        obj_a.A2(802);
                }
                }
                …
        }

        public static void main(int r){
                TestTH t1 = new TestTH ();
                TestTH t2 = new TestTH ();
                t1.start();
                t2.start();
        }
}
```

圖 10. 產生 thread 並且同時呼叫 Synchronized 程式區段的範例

上述功能在先前 Multi-core Coordinator[1]設計中僅適用於 temporal multithreading 機制未被啟用的多核心 JAIP 處理器環境，每一個處理器只能容納一個 thread，此時當一個 thread 取得 lock object 失敗時，這個 thread 即暫停執行指令，因為 temporal multithreading 機制未啟用所以無法切換到下一個 ready thread 執行，導致 Bytecode Execution Engine 閒置的情形發生。

在多核心 JAIP 處理器環境下如要啟用每一個 JAIP 的 temporal multithreading 機制，

我們將修改先前的設計並且更改其名稱為 Data Coherence Controller (圖 11)，當 Data Coherence Controller 收到來自 JAIP 處理器的 request 時:(1)先由 Mutex Controller (圖 25) 檢查 Data Coherence Controller 目前是否正在執行其他處理器的 request signal，確定每一個 JAIP 的 request 都能正確地被執行 (2) Mutex Controller 會解碼 request signal 啟動不同電路模組。若進來的 request 是要分配 new thread 到其他 JAIP，則啟動 Thread Assignment Controller (圖 28)；若進來的 request 是要取得/釋放 lock object，則我們提出 Lock Object Accessing Controller 的設計(圖 31)，可以檢查所有 threads 等待所有 lock objects 的狀態。

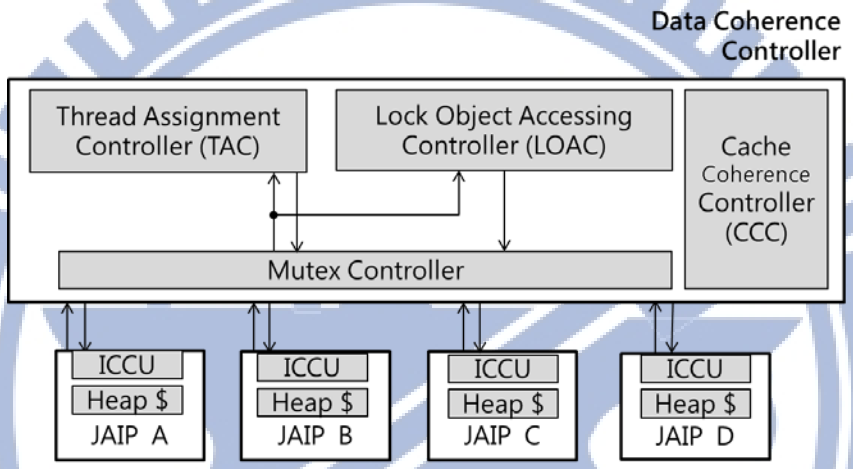



圖 11 Data Coherence Controller 與各個 JAIP 處理器的關係圖

為了使 Data Coherence Controller 能控制每個 JAIP 處理器底下的 Thread Manager Unit，我們也在每個 JAIP 處理器內提出 Inter-Core Communication Unit (圖 11，以下簡稱 ICCU)的設計。同時為了使 ICCU 能正確執行，我們也將修改原本 JAIP 的 DRSU 與 Thread Manager Unit 部分元件例如：Thread Control Block、Thread Queue 與 Thread Controller，Hardware Native Interface 設計也將被整合到 JAIP。

本章節可大略分為二個部分；第一部分先說明單一 JAIP 處理器 thread 的新增分配流程以及同步操作的設計，最後將會觸發 ICCU；第二部份再介紹 Data Coherence Controller 以及每個 JAIP 處理器底下的 ICCU 與 Thread Manager Unit 如何互相運作，以維護 threads 執行資訊。

## 3.1. Hardware Native Interface

Java 平台的 native method 大多數以其他語言或組合語言的 library 實做，透過呼叫這些 library 完成工作並且回傳數值。為了使 Java Thread Class 能正確地操作 ICCU、Thead Manager Unit 或者其他加速電路，在不修改原有 Java 應用程式情況下，我們加入 Hardware Native Interface[2]的設計，這個介面由 DSRU 觸發，使得 native method 的執行介面達到一致性，如果開發者要為某一個 native method 新增一個等價電路時，便可以簡單地連接 Hardware Native Interface 的輸出輸入訊號。

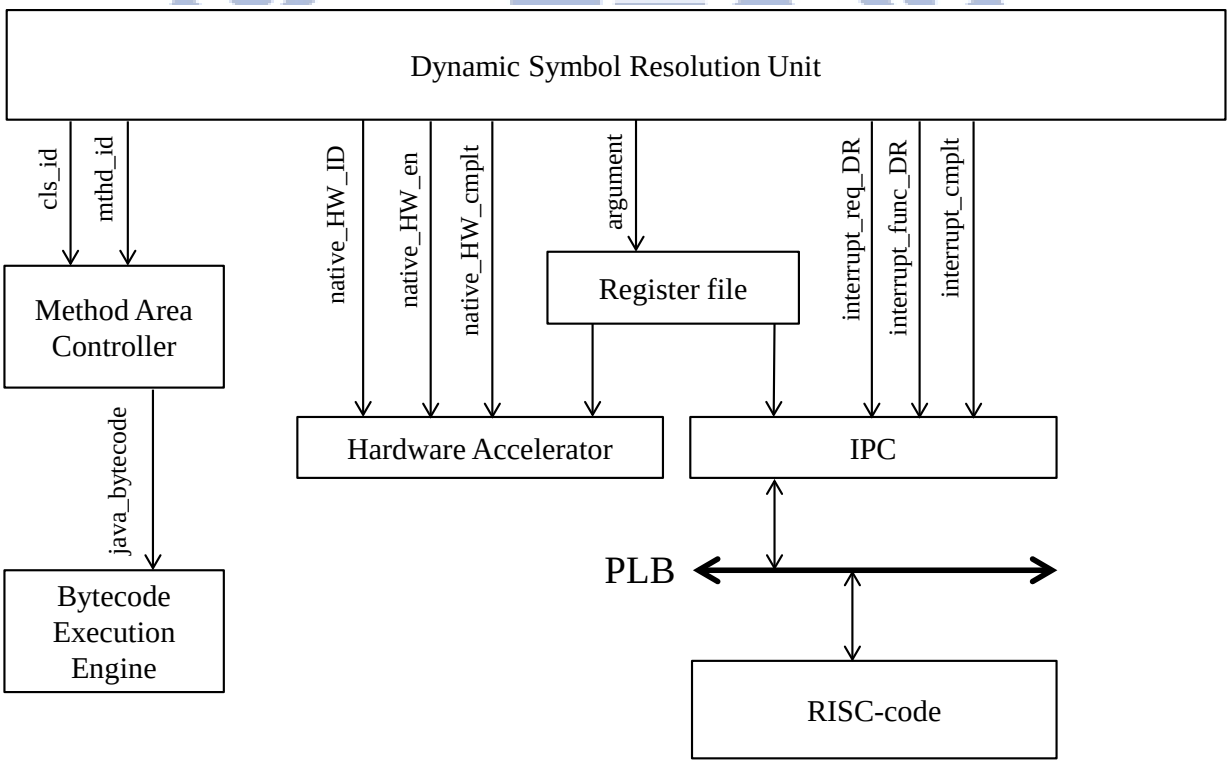



圖 12 JAIP 調用的 3 種介面

(1) Bytecode Execution Engine (2) RISC-core (3) Hardware Accelerator

圖 12 說明 JAIP 處理器內 2 種 natvie method 的實作方式：其一是啟動 interrupt 通知 RISC-core，RISC-core 提供對應的 native C function 來完成工作，執行過程中由 RISC-code 讀取 JAIP 的 stack memory 頂端 data words；第二種方式是 Hardware Native Interface，當 DSRU 解析過程中發現即將呼叫一個 native method，且該 native method ID

對應到某個 JAIP 處理器支援的 Hardware Accelerator，則啟動對應的電路、傳入 method 參數並執行工作、把回傳值寫到 stack memory 頂端，當其對應的電路工作結束時，發送完成訊號到 DSRU，此時完成 native method invocation。此設計的好處是可降低 native method 執行時間。

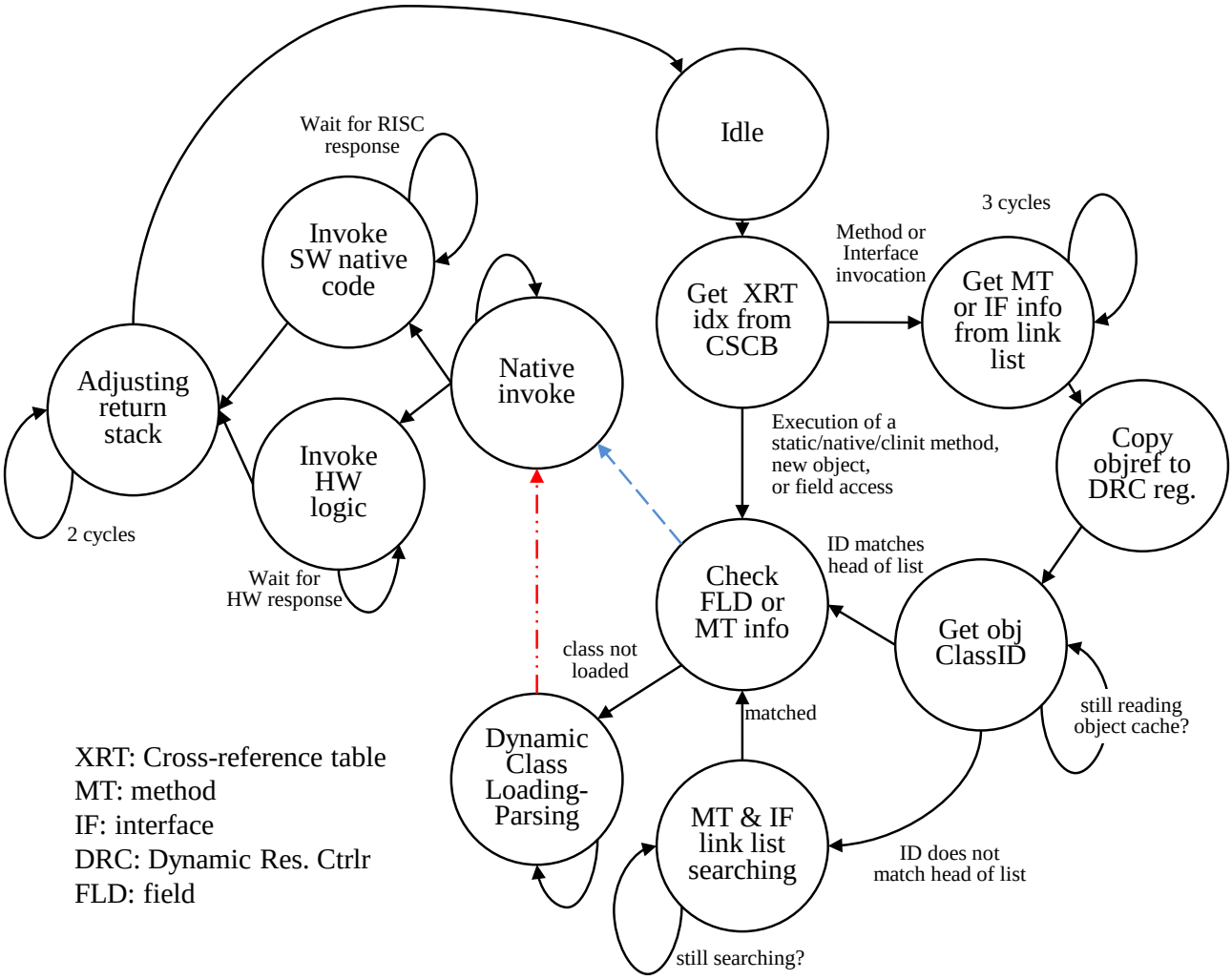



圖 13 在 DSRU 中呼叫 Hardware Native Interface 的 state diagram

在我們的設計下當 current thread 呼叫 native method 會執行 invokevirtual、invokestatic 等指令，當 Decode Stage 解析發現要執行這些指令時便會啟動 Dynamic Symbol Resolution Unit(DSRU)，此時 DSRU 狀態從 Idle 到 Get_XRT_idx_from_CSCB(如圖 13)，同時將 operand bytes 及 control flags 從 Decode stage 傳給 DSRU，DSRU 之後根據 decode stage 給的 control flags 來決定工作內容，如果發現是要呼叫 method 的動作，則 DSRU

從 Cross Reference Table 找到對應的 method info list，逐一檢查每個 method info 的 cls_id、mthd_id、native_flag (圖 14)。當 DSRU 搜尋到正確的 class ID 與 method ID 時，其狀態便進入到 Check_FLD_or_MT_info，檢查目前這個 method 相關資料是否已被載入到 Method Area Circular Buffer 與 Class Symbol Table Circular Buffer，如果沒有則先進入 DynamicClass_Loading_Parsing 狀態，啟動 Class parser 把 class data 與 method bytecode 分別載入 Class Symbol Table Circular Buffer 與 Method Area Circular Buffer 之後，再檢查 native_flag 的值，決定要下個狀態是 MACB_CSCB_Loading 或是 Native_invoke。如果呼叫的 method bytecode 已被載入到 Method Area Circular Buffer，則不需啟動 Class Parser，直接判斷 native_flag 決定下個狀態是 Native_invoke 或是 MACB_CSCB_Loading。由於在此我們舉例 current thread 呼叫 native method，因此，native_flag=1，DSRU 下個狀態進入 Native_invoke。

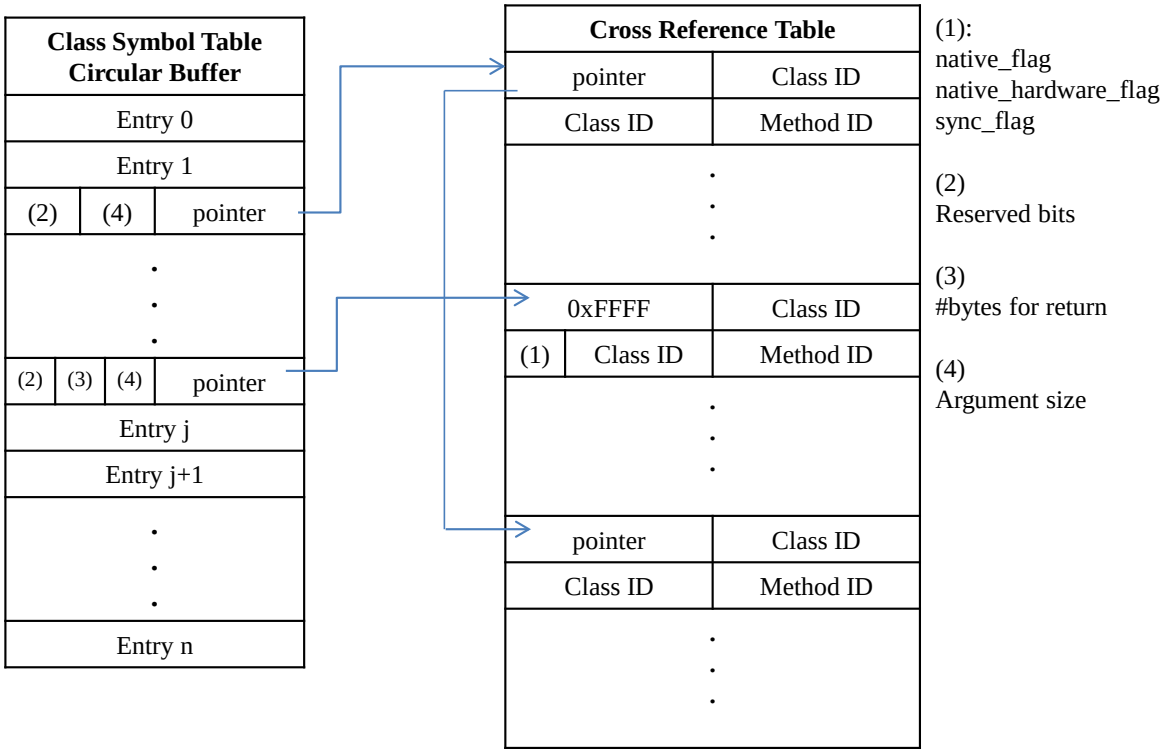



圖 14 method info list 訪問過程與 method info 格式

native_hardware_flag(圖 14)被用來判斷此 native method 是由 RISC-core 實作與否。如果其值為 0，表示這個 method 以 RISC-core 實作 interrupt service routine，此時 mthd_id

表示 interrupt service routine ID 編號，且 DSRU 下個狀態進入 Invoke_SW_native_code；如果其值為 1，表示這個 method 是由 hardwired 電路或加速器實作，此時把 method info 的 method ID 欄位寫入 native_HW_ID 並且 DSRU 下個狀態進入 Invoke_HW_logic。當 DSRU 狀態為 Invoke_HW_logic 時啟動 Hardware Native Interface，把 native_HW_en、native_HW_ID 這 2 個 Hardware Native Interface 輸出訊號連接個別加速電路模組內。

以圖 10 為例，當 main thread 呼叫 t1.start()或者 t2.start()時會啟動 DSRU，如上一段描述當 DSRU 到 Cross Reference Table 讀取 Thread.start()的 method info 時發現 native_flag 與 native_hardware_flag 的值皆為 1，因此 DSRU 下個狀態進入 Invoke_HW_logic。根據圖 12，Hardware Native Interface 的 native_HW_en、native_HW_ID 兩條訊號連接到 ICCU，當 DSRU 狀態為 Invoke_HW_logic，如果 native_HW_en 等於 1 則透過 Hardware Native Interface 觸發 ICCU 執行產生 new thread 的動作，ICCU (圖 19)詳細步驟即將在 section 3-3 描述。當 ICCU 完成產生 new thread 工作之後，native_HW_cmplt 等於 1 且 native_HW_en 等於 0，此時 DSRU 的狀態從 Invoke_HW_logic 轉到 Adjusting_return_stack，把 2-level Java Runtime Stack 回復到呼叫 method 之前的狀態，最後完成呼叫 start method 的工作。

## 3.2. Synchronization Operation

為了在 multithreading 環境下支援同步機制，本章節前面有舉例提到 Java 語言提供 Synchronized 保留字，相關實作方式包含 synchronized statement (圖 9(a))與 synchronized method (圖 9(b))，其中 synchronized method 的細節留給個別 Java 平台實做[6]。圖 9 的 class B 用 Java compiler 所產生的 method B2、B3 的 bytecode 會包含 monitorenter 與 monitorexit 指令，因此當 TestTH class (圖 10)產生兩個 threads t1, t2 之後每次執行 method B2 或 B3 時，必然會執行到 monitor 相關指令；然而 Class A 編譯過後所產生的 method A2、A3 的 bytecode 並不包含這兩個指令，因此我們在 JAIP 處理器做少部分的修改以實現 synchonized method 功能。

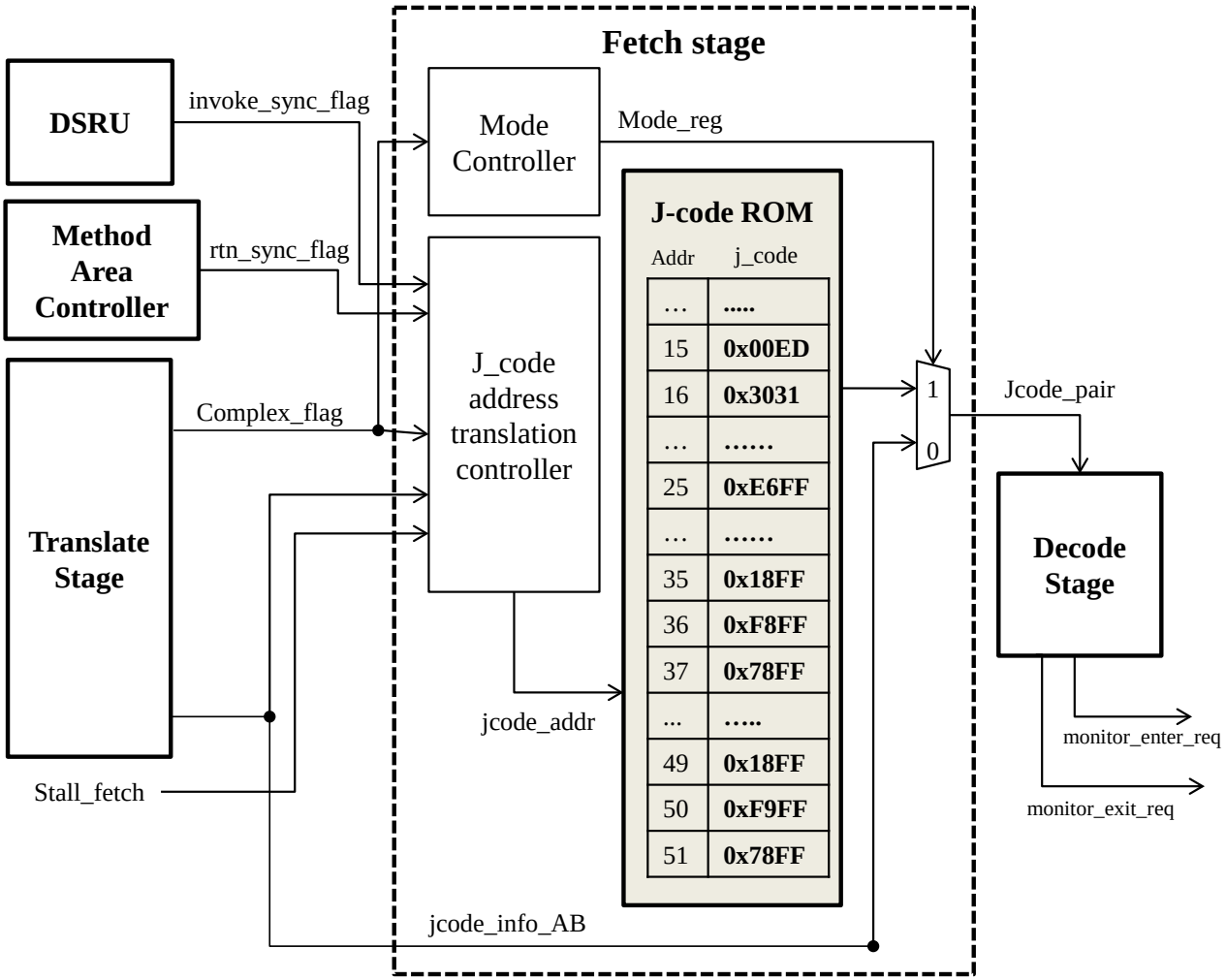



圖 15 Fetch stage 解析複雜指令示意圖

以下先說明 JAIP 處理器的 Bytecode Execution Engine 如何解析 monitorenter 與 monitorexit 並啟動 ICCU，再說明 synchronized method 電路實作方式。圖 15 說明 JAIP 的 Fetch stage 讀取 bytecode 複雜指令時，把原本 bytecode 指令轉換成一系列 j_code 指令再依序傳到 Decode Stage 解析，某些指令如果無法在 1 clock 內於 Fetch、Decode、Execute stage 完成工作的都被視為複雜指令。根據 Java instruction set 定義，執行 monitorente 與 monitorexit 這 2 個指令之前 Java Stack 的頂端元素必定存放 lock object 的參考位置，當 current thread 執行這 2 個指令時，便會觸發 Java VM 內部機制，讀取這個 lock object 且檢查是否已經被其他 thread 占用，再回傳訊息給 current thread 判斷 thread state 是否會改變，最後 Java VM 內部執行 pop 的動作將 lock object 參考位置從 Java stack

頂端移除。而 JAIP 與 Data Coherence Controller 無法在 1 clock 內完成以上動作，因此 monitorenter 與 monitorexit 指令在 JAIP 內部被歸類為複雜指令。

當 monitorenter 與 monitorexit 進入 Translate Stage 時，Two-level Java Stack memory 的頂端元素為 lock object 的參考位置，Translate Stage 的輸出訊號 complex_flag 等於 1，此時 jcode_info_AB 的內容代表 j_code ROM 的參考位置，指向 monitorenter 與 monitorexit 的 j_code 序列中第一個 j_code 位置。j_code ROM 為一個 on-chip block RAM 用來儲存每個 bytecode 複雜指令所對應到的 j_code 序列，j_code 指令 0xF8 被用來觸發 Decode Stage 的 monitor_enter_req(圖 3-3)，j_code 指令 0xF9 被用來觸發 Decode Stage 的 monitor_exit_req，而 j_code 指令 0x78 代表從 Two-level Java Stack memory 移除一個頂端元素。0xF8 與 0x78 這 2 個 j_code 在此被用來實做 monitorenter 複雜指令，0xF9 與 0x78 這 2 個 j_code 在此被用來實做 monitorexit 複雜指令，以圖 15 為例，monitorenter 與 monitorexit 對應的 j_code 序列儲存在 j_code ROM 的起始位置分別為 36 與 50。

JAIP Fetch Stage 處理指令 monitorenter 與 monitorexit 的流程皆類似，在此一併描述其流程。假設目前 Fetch Stage 正在處理 monitorenter 指令，則 mode_reg 的值變成 1，jcode_info_AB=36 傳入 j_code address translation controller (圖 15)，透過其內部電路決定把 jcode_info_AB 的值傳給 jcode_addr，同時 mode_reg=1 表示 Fetch Stage 目前執行的是複雜指令。因此從 j_code ROM 讀取的 j_code 會藉由 multiplexer 傳給 j_code_pair，下個 clock j_code ROM 的訊號輸出為 0xF8 與 0xFF，當 Decode Stage 解析這組 j_code 時會拉起 monitor_enter_flag，如 section 3-1 描述也會進一步啟動 ICCU，同時 stall_fetch=1 使 j_code ROM 暫停往下讀取下一個 j_code。等到 ICCU 整個執行完畢後，便會恢復整個 Bytecode Execution Engine 正常運作，意即 stall_fetch=0。Fetch stage 累加 jcode_addr 其值為 37，並且到 j_code ROM 往下讀取一對 j_code 0x78 與 0xFF，移除 Two-level Java Stack memory 頂端元素。之後 complex_flag 與 mode_reg 等於 0，Fetch Stage 開始處理下一個 bytecode 指令。

為了實做 synchronized method 電路，我們將在 method info 之中加入 synchronized flag (如圖 14 的 sync_flag)，呼叫 method 過程中 DSRU 可以藉由 sync_flag 數值判斷是否為

synchronized method，修改 Fetch Stage 的 j_code address translation controller，使 DSRU 與 Method Area Controller 可以傳送信號控制 jcode_addr，我們在 j_code ROM 之中定義 2 組 j_code 序列並命名為 invoke_sync_mthd 與 rtn_sync_mthd，如圖 15 所敘述，這 2 個 j_code 序列的起始位置分別為 35 與 49，結束位置分別為 37 與 51，以此方式重複使用 monitorenter 與 monitoerexit 的 j_code 序列；最後在 return frame 中加入 synchronized flag (圖 17)，當執行 return 相關指令時 Method Area Controller 可判斷 synchronized flag 決定是否需要執行釋放 lock object。

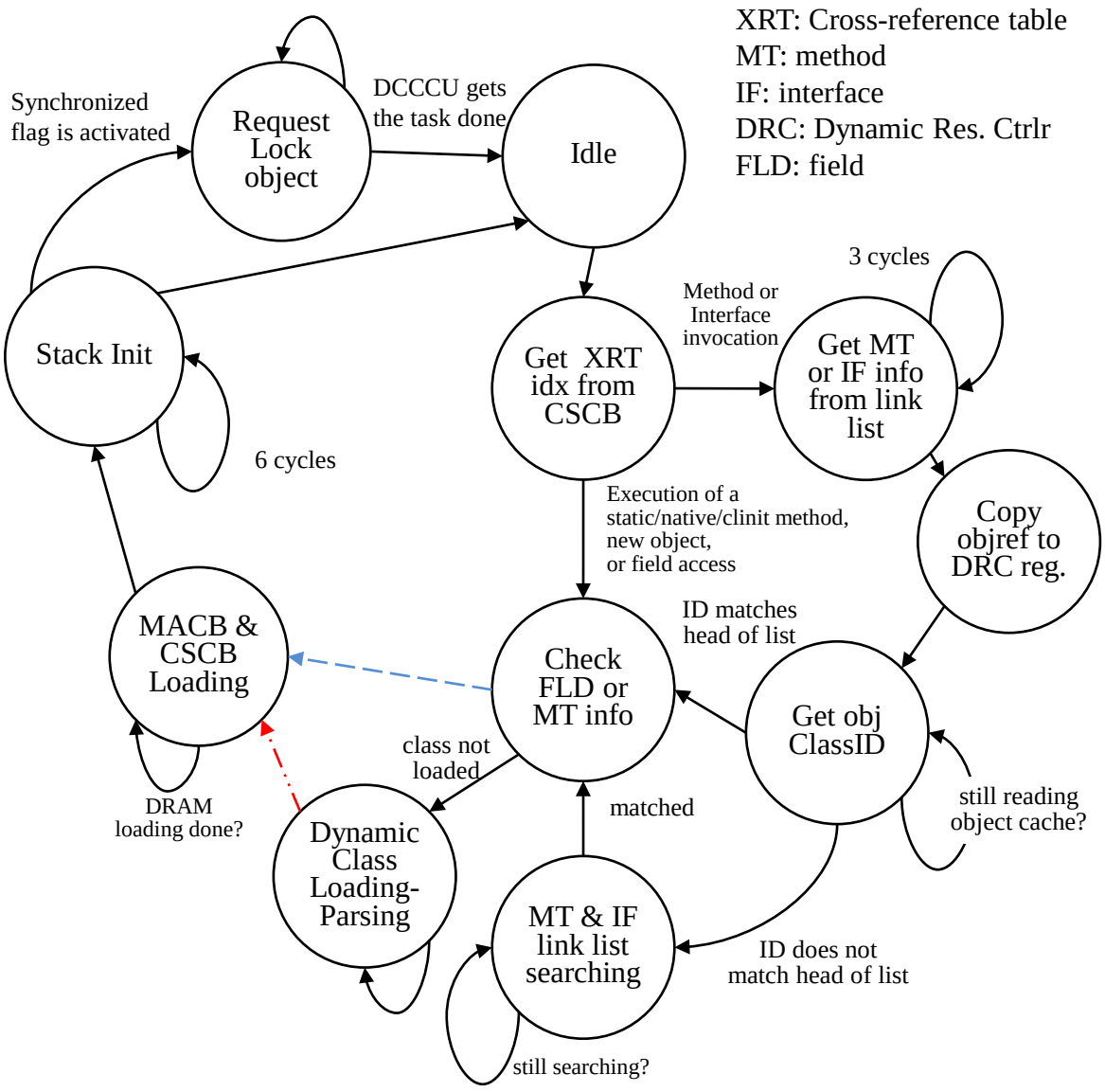



圖 16 在 DSRU 中呼叫 synchronized method 的流程

當 Class Parser 解析一個 class 過程中如果遇到 synchonized method，則該 method info

的 sync_flag=1，Class parser 最後把所有 method_info 資訊依序寫到 cross reference table。當 current thread 呼叫 method 時會執行相關複雜指令例如 invokevirtual、invokeinterface、invokestatic 等，Fetch Stage 的 mode_reg=1，啟動 DSRU，其流程如圖 16 所示，此時 DSRU 狀態從 Idle 到 Get_XRT_idx_from_CSCB，此時將 return frame 放到 Two-level Java stack memory 的頂端元素 (TOS_A、TOS_B，圖 19)，同時將 operand bytes 及 control flags 從 Decode stage 傳給 DSRU，DSRU 之後根據 decode stage 給的 control flags 來決定工作內容，如果發現是要呼叫 method 的動作，並且 DSRU 搜尋到正確的 class ID 與 method ID 時，其狀態進入到 Check_FLD_or_MT_info，檢查目前這個 method 相關資料是否已被載入到 Method Area Circular Buffer 與 Class Symbol Table Circular Buffer，如果沒有則先進入 DynamicClass_Loading_Parsing 狀態，啟動 Class parser 把 class data 與 method bytecode 分別載入 JAIP 之後，再進入下個狀態 MACB_CSCB_Loading。如果 method bytecode 已被載入到 Method Area Circular Buffer，則直接進入 MACB_CSCB_Loading。

| | | | | | |
|---|---|---|---|---|---|
| 1st. Entry | Cls_id | Lv_cnt | (1) | (2) | vp |
| 2nd. Entry | Mthd_id | jpc | | | |

(1) Reserved bit
(2) Synchronized bit

圖 17 return frame 格式

Method Area Controller 把呼叫的 method 完全載入到 Method Area Circular Buffer 之後，DSRU 狀態進入到 Stack_init，此時 stall_fetch=0，所以 j_code ROM 輸出的 j_code 被傳到 jcode_pair 進入 Decode Stage。調整完 local variable pointer (vp)與 stack pointer (sp)之後，此時 DSRU 內部 register 保存 method info 的訊息，並且呼叫 method 的工作已完成，vp 指向 current method 的第 1 個 local variable(object 參考位置)。接著檢查 method info 的 sync_flag 訊號，如果是呼叫 synchronized method 則 sync_flag=1，DSRU 狀態從 Stack_init 轉成 Request_Lock_object。此時 2-level Java stack memory 的頂端元素為 return frame 的第一個 entry (圖 17)，故直接把 stack 頂端元素的 synchronized bit 設成 1，DSRU

的 invoke_sync_flag=1，啟動 fetch stage 的 J_code address translation controller(如圖 15)，此時 jcode_addr=35 指向 invoke_sync_mthd 的 j_code 序列中第一個位置。並且下個 clock 讀取 j_code 0x18 與 0xFF，j_code 0x18 會把 vp 指向的 stack entry(第一個 local variable，lock object 參考位置)加到 stack 頂端，接著 J_code address translation controller 再累加 jcode_addr 的值為 36 使下一組 j_code 0xF8 與 0xFF 被讀取。後續步驟跟上一段執行 monitorenter 的 j_code 序列流程一樣。等到 monitorenter 的 j_code 序列整個執行完畢後，不管在 Data Coherence Controller 是否成功取得 lock object 權限，此時 mode_reg=0，DSRU 狀態回到 Idle。

當 current thread 在 synchronized method 底下執行 return 相關指令時，2-level Java Stack memory 的頂端 2 個元素(TOS_A、TOS_B，圖 19)分別存放 synchronized method 的 2 個 return frames(圖 17)，Decode Stage 發送訊號讀取 TOS_A 的 synchronized bit 判斷是否要執行 monitorexit 指令，結束之後再從 Class Symbol Table Circular Buffer 與 Method Area Circular Buffer 載入前一個 method image 與 symbol information(圖 18)，Class Symbol Table Controller 的執行流程與 Method Area Controller 類似，在此一併說明。當 Method Area Controller 狀態為 Get_Offset，如果 return frames 的 synchronized bit = 1 則下個狀態為 Release_lock_object；否則下個狀態為 Check_Offset。當 Method Area Controller 狀態為 Release_lock_object，rtn_sync_flag=1 並且啟動 fetch stage 的 J_code address translation controller，此時 jcode_addr=49 指向 rtn_sync_mthd 的 j_code 序列中第一個位置。並且在下個 clock 讀取 j_code 0x18 與 0xFF，此時 vp、sp 與 synchronized method 的 local variables 存放的位置未被清除，因此透過 vp 載入第一個 local variable(lock object 參考位置)加到 stack 頂端，接著 J_code address translation controller 再累加 jcode_addr 的值為 50 使下一組 j_code 0xF9 與 0xFF 被讀取。後續步驟跟上一段執行 monitorexit 的 j_code 序列流程一樣。等到 monitorexit 的 j_code 序列整個執行完畢後，此時 mode_reg=0，Method Area Controller 狀態進入 Check_Offset，開始載入 parent method 的 bytecode。

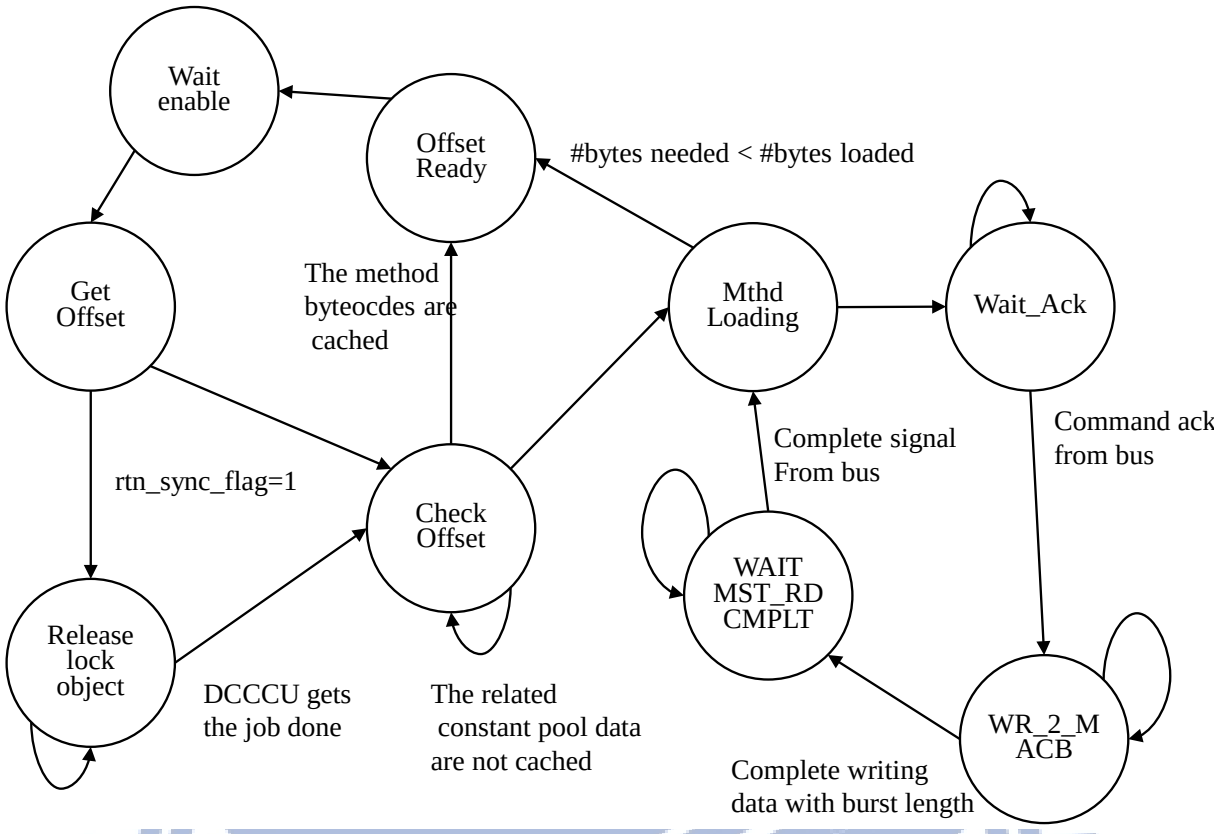



圖 18 The state diagram of Method Area Controller

## 3.3. Inter-Core Communication Unit (ICCU)

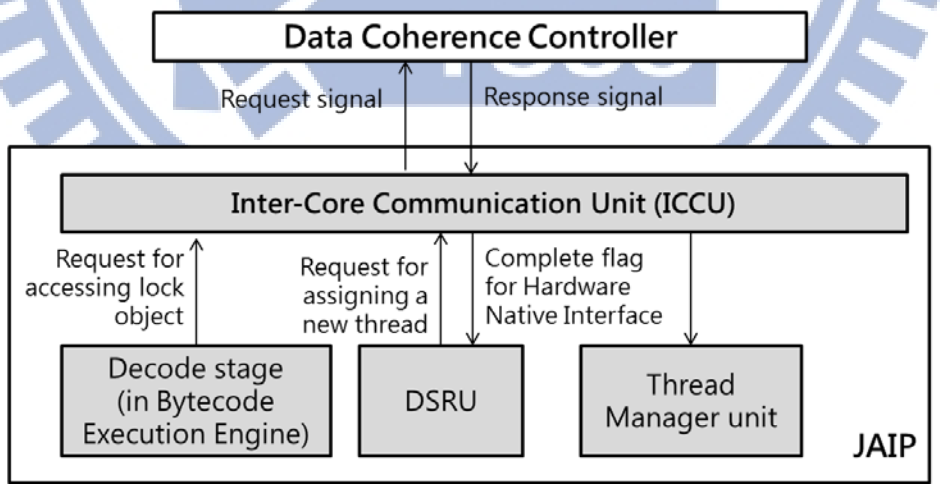



圖 19 The block diagram of Inter-Core Communication Unit

每個 JAIP 處理器底下的 Inter-Core Communication Unit (ICCU，圖 19)用來負責與 Data Coherence Controller 以及 JAIP 處理器內各個電路模組的溝通。當 DSRU 或者 Decode Stage 送出 request 時，ICCU 會判斷 request 類型，抓取對應的資訊(例如：new thread 的

class ID、method ID、thread object 參考位置、lock object 參考位置以及 current thread ID)，再發送 request signal 到 Data Coherence Controller 執行工作；而當 Data Coherence Controller 完成處理器發送的 request signal 後便會發送回應訊號給此處理器的 ICCU，ICCU 收到回應訊號便會開始解析回應內容，拉起對應的訊號觸發 DSRU 或 Thread Manager Unit 內某些元件。

| JAIP2DCC_cmd | Command Description |
|---|---|
| 00 | Default value, nothing happened. |
| 01 | Acquiring a lock object for the specific thread |
| 10 | Release a lock object for the specific thread |
| 11 | Assigning new thread to one processor, which contains minimum number of threads for all of processors. |

表 1 ICCU 支援的 request type

| Response Status | Corresponding JAIP2DCC_cmd | Description |
|---|---|---|
| 010 | 11 | When a processor gets the response, it means current thread of a processor is invoking Thread.start(), this processor contains minimum number of threads for all of processors at that time, note both of them could happen on this processor. |
| 100 | 01 | Current thread succeeds in acquiring the specific lock object and owns it, the thread can execute instructions in critical section. |
| 101 | 01 | Current thread fails to acquire the specific lock object and must wait. |
| 110 | 10 | When a processor gets the response, it means the other thread, which is from other processors and owns the lock right now, released the lock, and a thread of this processor become the next owner |
| 111 | 10 | When a processor get the response, it means no other threads are waiting for the lock at the time after current thread releases the lock. |

表 2 ICCU 支援的 response type

表 1 與表 2 分別說明目前 ICCU 支援的 request 類型與回應訊息類型，以下說明每種 request 發送訊息的流程。以圖 10 為範例，假設 main thread 呼叫 t2.start()產生 new thread 時，DSRU 會啟動 Hardware Native Interface，把 native_HW_en 設成 1 並且啟動 Hardware Native Interface，把輸出訊號傳到 ICCU 同時記錄 DSRU 的 2 個輸出訊號：TH_objref 與 Cls_id_mthd_id 的值，分別表示 thread object 參考位置與 new thread 起始執行的 class ID 與 method ID，藉由 JAIP2DCC_info 把這些資料傳到 Data Coherence Controller。最後 JAIP2DCC_cmd 的值設定為 11，此時發送 request 至 Data Coherence Controller 的工作已完成。

以圖 10 為範例，假設當 thread t2 呼叫 method B3 進入 synchronized statement 欲取得/釋放 lock object 時，首先 Bytecode Execution Engine 暫停執行指令，根據之前的 2-level Java stack memory 設計(圖 3)，JAIP stack memory 的頂端元素為 TOS_A，此時儲存 lock object 參考位置，同時 Decode Stage 解析到對應的 j_code：monitor_enter_req 或 monitor_exit_req 值為 1，此時 ICCU 記錄 TOS_A 值與 current thread ID，藉由 JAIP2DCC_info 把這些資料傳到 Data Coherence Controller。最後 JAIP2DCC_cmd 的值為 01 或 10，發送 request 至 Data Coherence Controller 的工作已完成。

| **Receiver core ID [8 : 6]** | **Sender core ID [5 : 3]** | **Response status[2 : 0]** |
|---|---|---|

| **core ID** | **Description** |
|---|---|
| 000 | Java Core 0 |
| 001 | Java Core 1 |
| 010 | Java Core 2 |
| 011 | Java Core 3 |
| 100 | RISC Core (Microblaze) |
| 111 | Default value |

圖 20 DCC2JAIP_response_msg 的格式

接著說明 ICCU 接收與解析 DCC2JAIP_response_msg 的格式，當 Data Coherence

Controller 完成工作時會修改 DCC2JAIP_response_msg 並且根據 Sender core ID 與 Receiver core ID 欄位把訊息傳給對應的 JAIP 處理器 (圖 20)。我們將根據表 2 個別舉例說明：

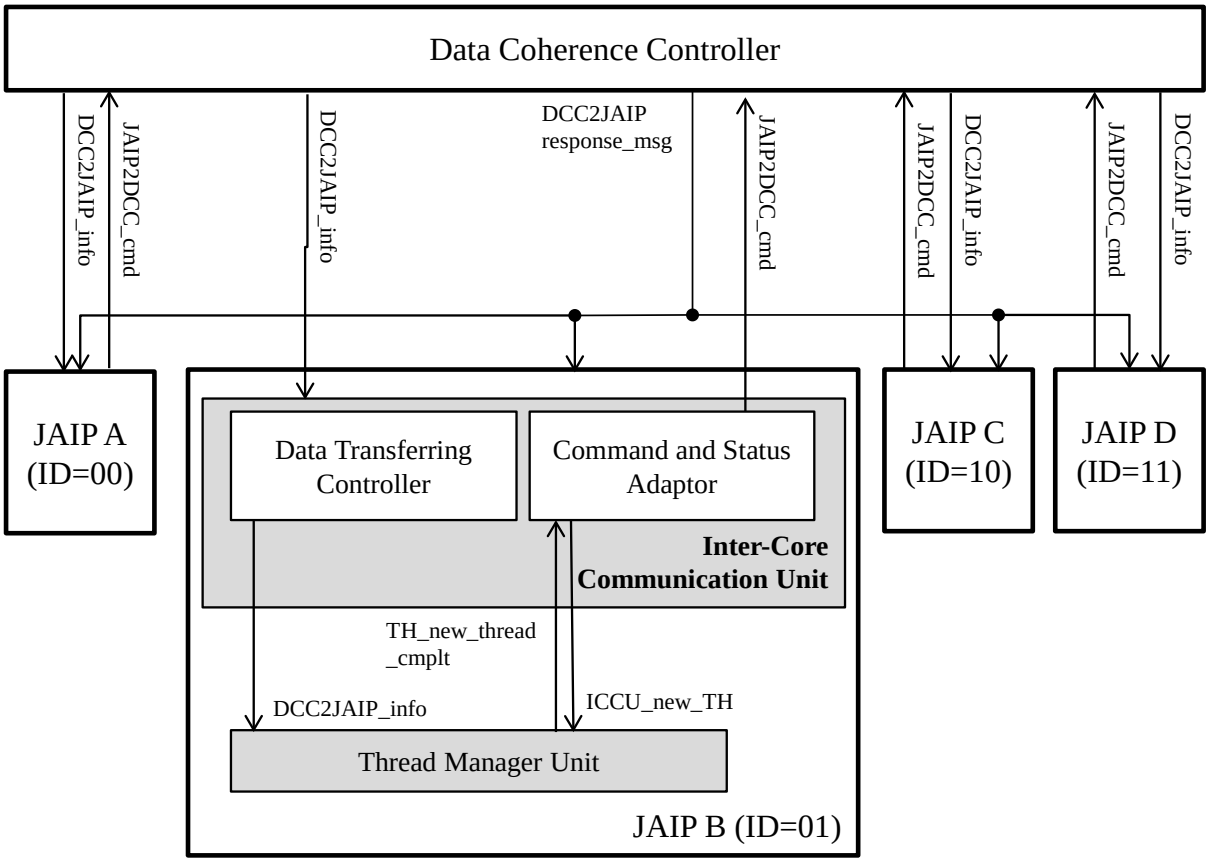



圖 21 ICCU 將 response status 回傳給處理器之示意圖

如圖 21，假設先前 JAIP A 發送的 request 是產生 new thread (JAIP2DCC_cmd = 11)，並且依照 Data Coherence Controller 內部機制決定將 new thread 分配到 JAIP B，此時 DCC2JAIP_response_msg 為 001000010，Receiver core ID = 001(JAIP B)，Sender core ID = 000(JAIP A)，Data Coherence Controller 透過 DCC2JAIP_info 把 new thread 的 Thread Object 參考位置、class ID 與 method ID 傳到 JAIP B。JAIP B 的 ICCU 分析 DCC2JAIP_response_msg 得知 new thread 執行資訊將被存放在這個處理器內，於是觸發 Data Transferring Controller，接收 DCC2JAIP_info 的 data，並且把 ICCU_new_TH 設成 1 啟動 Thread Controller，利用 Thread Control Block 儲存 new thread 執行資訊，完成工作後再發送信號到 ICCU。而 JAIP A 收到 DCC2JAIP_response_msg 的回應訊號之後，會檢

查 Receiver core ID 是否等於 Sender core ID，如果相同則 JAIP A 會啟動 Thread Controller，細節步驟皆與上一段描述的相同；否則 Receiver core ID 不等於 Sender core ID，最後 JAIP A 的 ICCU 觸發 Hardware Native Interface 表示執行完 Thread.start()的工作。

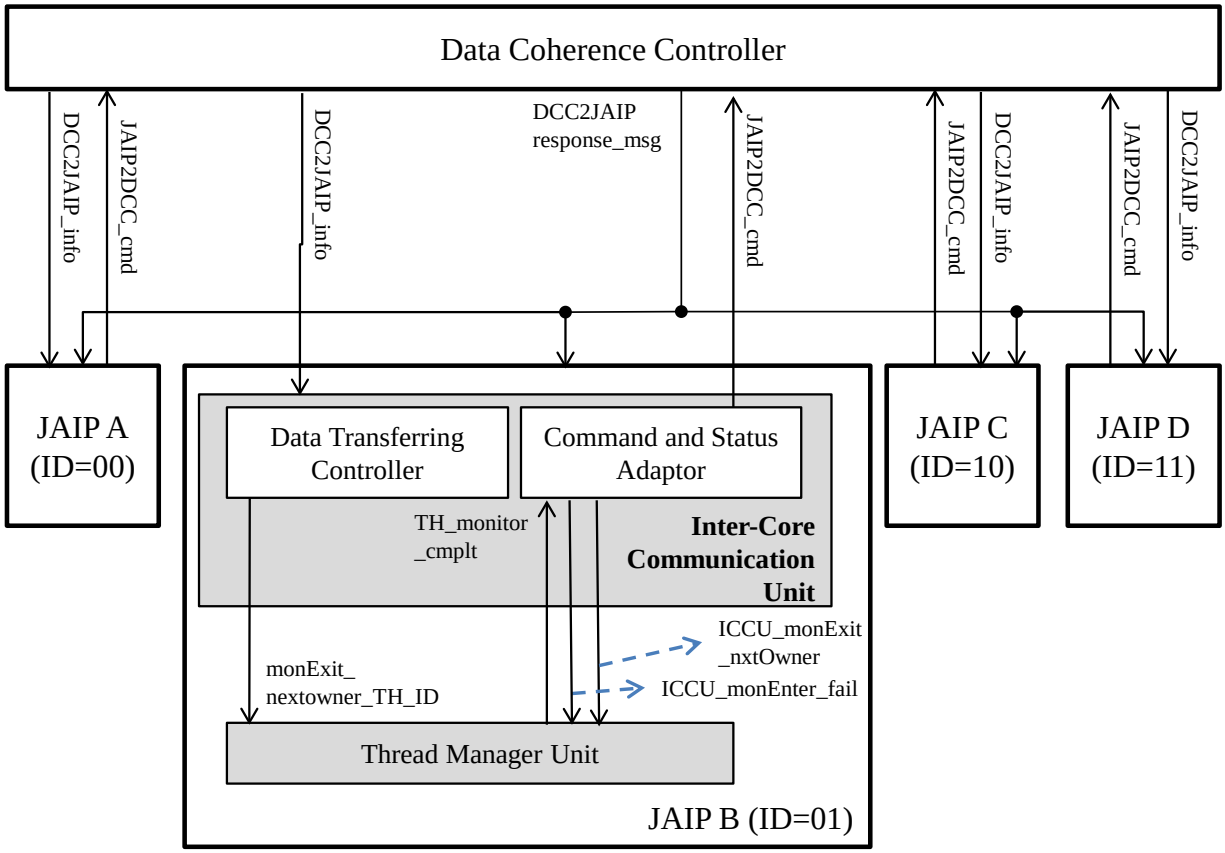



圖 22.ICCU 將 response status 回傳給處理器之示意圖

假設 JAIP B 的 current thread 要取得 lock object L(JAIP2DCC_cmd = 01)，以圖 22 為例，Data Coherence Controller 完成工作後將 DCC2JAIP_response_msg 的值改為 001001101，Receiver core ID = Sender core ID=001(JAIP B)，而 Response status 為 100 或 101，取決於目前 Data Coherence Controller 是否已存在 lock object L 的紀錄，表示 lock object L 目前已被其他 thread 取得權限。最後更改 ICCU_monEnter_fail 的值，如果該 thread 未成功取得 lock object L 則 ICCU_monEnter_fail = 1 表示 JAIP B 的 current thread 即將進入 waiting state，因此當 JAIP B 的 Thread Manager Unit 執行 context switch 之後會修改這個 thread 儲存在 Thread Control Block 的執行資訊；否則 ICCU_monEnter_fail = 0，JAIP B 的 ICCU 收到這個訊號後 Bytecode Execution Engine 恢復執行指令。

假設 JAIP A 的 current thread 釋放 lock ojbect L(JAIP2DCC_cmd=10)，以圖 22 為例，假設在 Data Coherence Controller 發現存在一個以上 waiting thread (s)正在等待取得 lock object L，則 Data Coherence Controller 會決定接下來哪個 waiting thread 能取得 lock object L，在此假設 Data Coherence Controller 將 lock L 的權限轉移到 JAIP B 的某個 waiting thread，則 Data Coherence Controller 完成工作後把下一個取得 lock object L 的 waiting thread ID 傳至 JAIP B、DCC2JAIP_response_msg 的值改為 001000110，Receiver core ID = 001，Sender core ID = 000，Response status = 110，此時 ICCU 收到訊號會將 monExit_nxtOwner 設成 1，表示 JAIP B 的某個 waiting thread 為 lock object L 的下一個擁有者。monExit_nextowner_TH_ID 傳入 Thread Manager Unit 把這個 thread 改成 ready state；最後 JAIP A 的 Bytecode Execution Engine 恢復執行指令；如果 Data Coherence Controller 內部不存在任何正在等待 lock object L 的 waiting threads，則 Response status = 111、monExit_nxtOwner = 0，JAIP A 的 Bytecode Execution Engine 直接恢復執行指令。

## 3.4. Data Coherence Controller

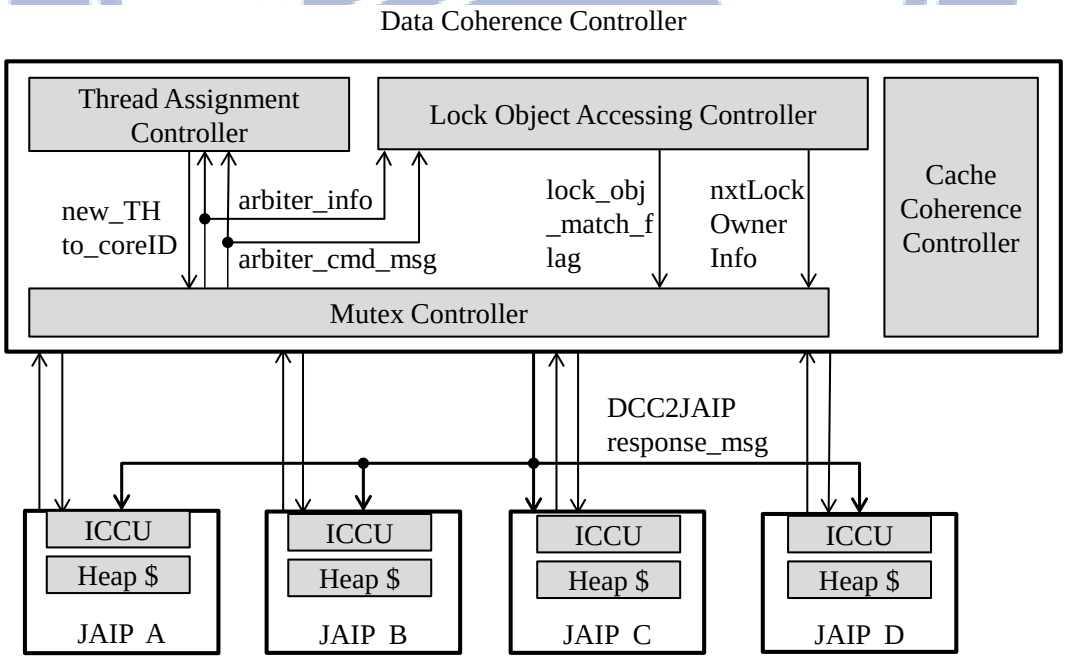



圖 23 The block diagram of Data Coherence Controller

Data Coherence Controller (圖 23)可分為以下幾個部分：

- Cache Coherence Controller，Java VM 定義 heap space 存放執行時期 Object 與 Array 相關資料，先前多核心 JAIP 處理器環境下每一個 JAIP 內存放一個 2-way set associative Heap Cache 並且採用 Least Recently Used replacement policy 備份資料到 DDR memory。一旦某個 JAIP 執行 heap 相關操作指令例如 new、newarray、anewarray、arraylength、getfield、putfield，為了確保其他 JAIP 的 heap cache 內容一致性，這個 JAIP 處理器將透過 JAIP2DCC_info 與 DCC2JAIP_info 把 heap address 與 data 寫入到其他 JAIP 的 heap cache，本論文中保留其原有機制。
- Mutex Controller，如果 2 個以上 JAIP 同時透過 ICCU 發送 request，Data Coherence Controller 必須決定哪一個 request 訊號優先被處理。
- Thread Assignment Controller，先前 JAIP information table [1]內每一個對應 entry 以 core ID 作為索引值用來判斷個別 JAIP 目前是否已被啟動或者閒置。當某個 JAIP 產生 new thread，Thread Assignment Controller 會查詢 JAIP information table 每一個 entry 決定 new thread 被分配到哪一個 JAIP。我們將延伸其 table 設計並用在我們的架構內。
- Lock Object Accessing Controller，先前 Multi-core coordinator[1]設計中提出一個 Synchronized Manager 處理 Java 同步機制，圖 24 描述 Synchronized Manager 的 state controller，當某個 thread A 要取得 lock object，便將 object 參考位置透過 JAIP 傳到 Synchronized Manager 檢查，此時 state 從 Idle 進入 Analysis，如果其他 thread 已先取得該 lock object 則 thread A 將暫停執行指令，此時 state 從 Analysis 進入 Update Table 等待，直到其他 thread 釋放該 lock object 給 thread A 之後，此時 state 從 Update Table 進入 Idle，thread A 才能繼續往下執行指令；當 thread A 要釋放 lock object，此時 state 從 Idle 進入 Free Lock，搜尋是否有其他 waiting thread 等待這個 lock object，若存在此 waiting thread 則將 lock object 權限轉移，此時 state 從 Free Lock、Update Table 回到 Idle。由於先前的設計並未啟用 temporal multithreading 機制，每一個 JAIP 只包含一個 ready thread，所以先前的設計僅使用 registers 暫存每一個 thread 欲取得的 lock object 狀態，並未考慮如果多個 threads 先後取得單一 lock object

時該如何記錄，如果每一個 thread 執行過程中會取得/釋放不同的 lock object，這會增加紀錄這些資訊的複雜度。本論文將提出另一設計可以使資源使用最佳化，並且同時支援多個 lock objects 與 waiting threads。

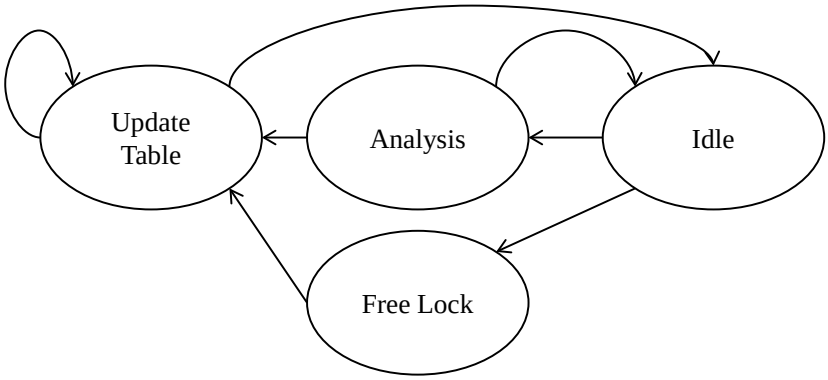



圖 24 The state diagram of Synchronized Manager in [1]

### 3.4.1. Mutex Controller

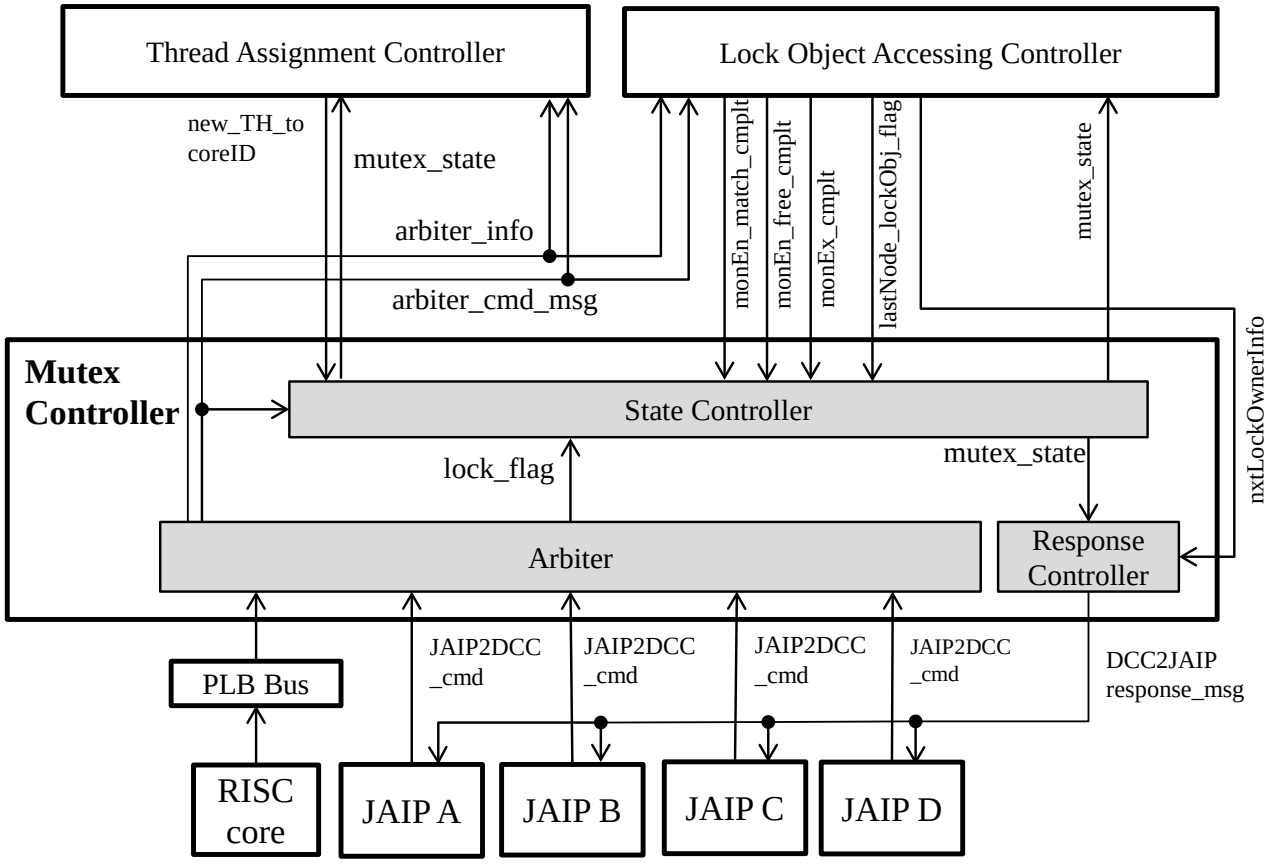



圖 25 The block diagram of Mutex Controller

圖 25 與圖 26 描述 Mutex Controller 的架構，當 2 個以上 JAIP 處理器同時發送 request 時，Arbiter 目的是優先挑選其中一個 request 使這些不同來源的 request 能循序被其他子模組處理；arbitor_cmd_msg 為 Arbiter 輸出，格式如下：arbitor_cmd_msg[4:2]為 core ID(如圖 20)，arbitor_cmd_msg[1:0]為 command ID(如表 1)，其儲存的值表示某個 JAIP 處理器取得權限並且參考 command ID 欄位與 mutex_state 準備觸發 Thread Assignment Controller 或者 Lock Object Accessing Controller，當沒有任何 JAIP 處理器啟動 Mutex Controller 執行工作時，arbitor_cmd_msg 預設值是 11100

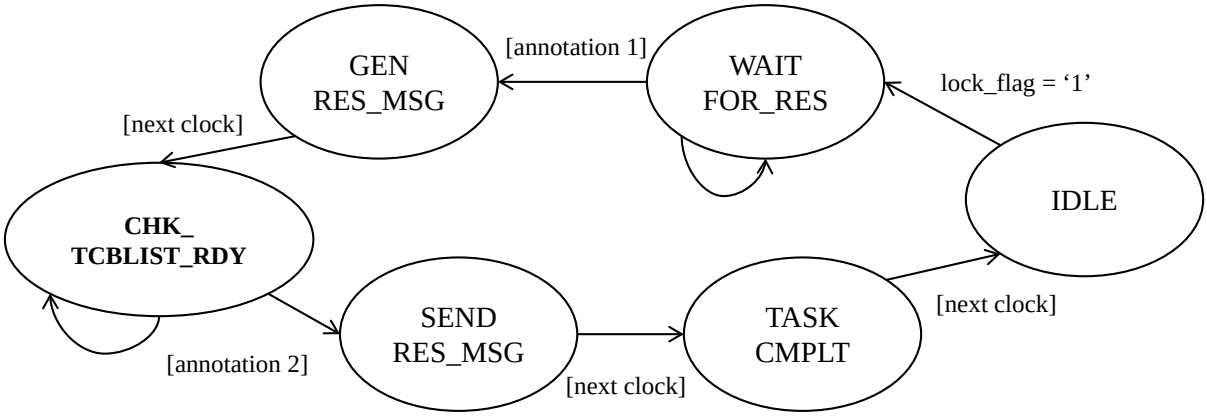



圖 26 The state diagram of Mutex Controller

**[annotation 1]**
根據圖 25 的訊號，當 monEn_match_cmplt 為 true、monEn_free_cmplt 為 true、monEx_cmplt 為 true、new_TH_to_coreID 不等於 111 其中一個條件成立時，轉移到下個狀態

**[annotation 2]**
判斷 DCC2JAIP_response_msg 的 Receiver core ID，檢查對應 JAIP 的 JAIP2DCC_delay_resMsg 數值

lock_flag 為 Arbiter 輸出訊號，其初始值為 0。Arbitor 會決定要優先處理哪個 request signal、lock_flag 由 0 轉為 1 並且 mutex_state 從 Idle 轉為 WAIT_FOR_RES，當 mutex_state = TASK_CMPLT、 lock_flag 值回到 0 才允許從 JAIP 處理器讀取下一個 request 並且修改 arbitor_cmd_msg；Arbiter_Info 只有當 arbitor_cmd_msg 不等於 111 時，根據不同 core ID 抓取對應 JAIP 處理器的 JAIP2DCC_info；mutex_state 表示目前 state controller 狀態並且表示目前 Data Coherence Controller 各模組工作階段；new_TH_to_coreID 為 Thread Assignment Controller 工作完成後輸出的信號，其數值代表的意義如同圖 20

DCC2JAIP_response_msg 的 Receiver core ID；monEn_match_cmplt、monEn_free_cmplt、monEx_cmplt 為 Lock Object Accessing Controller 工作完成後輸出的信號；JAIP2DCC_delay_resMsg (圖 26 的 annotation 2)為每一個 JAIP 的 ICCU 連接到 Mutex Controller 的信號，表示個別 JAIP 底下的 Thread Manager Unit 正在更新的 Thread Control Block (e.g. 增加 new thread 的執行資訊、Thread Controller 備份 previous thread 的執行資訊)；Response Controller 負責傳送回應訊號 DCC2JAIP_response_msg 到各個 JAIP 處理器的 ICCU。

以圖 10 為範例，當 thread t1 與 t2 分別透過 JAIP A 與 JAIP B 同時呼叫 synchronized method A2，此時由 ICCU 發送 request signal 到 Data Coherence Controller：此時 JAIP A 與 JAIP B 的 JAIP2DCC_cmd 同時不等於 0，mutex_state = Idle、由 Arbiter 判斷讓哪個 JAIP 的 JAIP2DCC_cmd 與 JAIP2DCC_info (圖 19) 分別寫入 arbitor_cmd_msg 與 Arbiter_Info，同時設定 lock_flag 值為 1。假設 Arbiter 優先選擇 JAIP A 的 request，則其他 JAIP 處理器的 request 必須等到之後 lock_flag 值為 0 且 mutex_state = Idle 時才有機會將 request 與 data 寫入 arbitor_cmd_msg 與 Arbiter_Info。當 lock_flag 值為 1，mutex_state 從 Idle 到 WAIT_FOR_RES，在這個範例中 arbitor_cmd_msg 值傳入並且啟動 Lock Object Accessing Controller 執行工作。

而當 Thread Assignment Controller 或者 Lock Object Accessing Controller 工作完成後各個電路回傳訊號(例如 monEn_match_cmplt、monEn_free_cmplt、new_TH_to_coreID)到 state controller，mutex_state 值從 WAIT_FOR_RES 到 GEN_RES_MSG。當 mutex_state = GEN_RES_MSG，產生回應訊號並且暫存到 Response Controller 的內部暫存器，其格式與 DCC2JAIP_response_msg 相同；當 mutex_state 為 CHK_TCBLIST_RDY，此時檢查 Response Controller 中用來儲存回應訊號的內部暫存器，檢查 Receiver core ID 與對應 JAIP 的 JAIP2DCC_delay_resMsg，如果 JAIP2DCC_delay_resMsg = 0 表示 Receiver core ID 對應的 JAIP 處理器其底下 Thread Controller Block 目前沒有被其他電路存取，則 mutex_state 進入 SEND_RES_MSG；否則 JAIP2DCC_delay_resMsg = 1，此 Thread Controller Block 目前已有其他電路更新其內容，Response Controller 必須延後將回應訊

息送給 DCC2JAIP_response_msg。當 mutex_state = SEND_RES_MSG，Response Controller 將回應訊息送給 DCC2JAIP_response_msg，傳到個別 JAIP 處理器的 ICCU。當 mutex_state = TASK_CMPLT、lock_flag 值回到 0、mutex_state 值從 TASK_CMPLT 回到 Idle，此時 Arbiter 檢查其他 JAIP 處理器的 JAIP2DCC_cmd，如果其他 JAIP 處理器的 JAIP2DCC_cmd 不等於 111，則把 JAIP2DCC_cmd 寫入 arbitor_cmd_msg 其餘執行流程相同，否則 arbitor_cmd_msg 被改為預設值。

### 3.4.2. Thread Assignment Controller

| core ID | active_TH_num |
|---|---|
| 000 | 2 |
| 001 | 2 |
| 010 | 2 |
| 011 | 1 |

圖 27 JAIP information table

圖 27 顯示我們修改的 JAIP information table 欄位格式，active_TH_num 為目前某個 JAIP 處理器下 active threads 數量。當 mutex_state = WAIT_FOR_RES 並且 arbitor_cmd_msg 的 command ID field 等於 11，此時 Arbiter_Info 訊號值代表的是 thread object 的參考位置、起始 class ID 與 method ID，並且開始搜尋 JAIP information table 找出目前哪一個 JAIP 處理器包含的 thread 數量最少，以決定 new_TH_to_coreID 的值。之後參考 new_TH_to_coreID 將 Arbiter_Info 輸出到對應 JAIP 的 DCC2JAIP_info，當 mutex_state = SEND_RES_MSG 將回應訊息透過 DCC2JAIP_response_msg 分別傳給呼叫原本發送此 request 的 JAIP 處理器以及即將儲存 new thread 執行資訊的 JAIP 處理器。

以圖 28 為例，在這個例子中當系統一開始啟動時，JAIP information table 的所有 entries 被歸零，此時所有 JAIP 處於閒置狀態，RISC-core 初始化且由 Class Parser 載入以及解析必要 Java class images，完成後 RISC-core 把 main method 的 class ID 與 method ID 透過 bus 傳給 Mutex Controller 的 arbiter_info，再依照 arbitor_cmd_msg 格式將二進位數值 10011 寫入 Mutex Controller 的 arbitor_cmd_msg (圖 28(a)的 step 1)，藉此方式傳

送新增 thread 的指令。

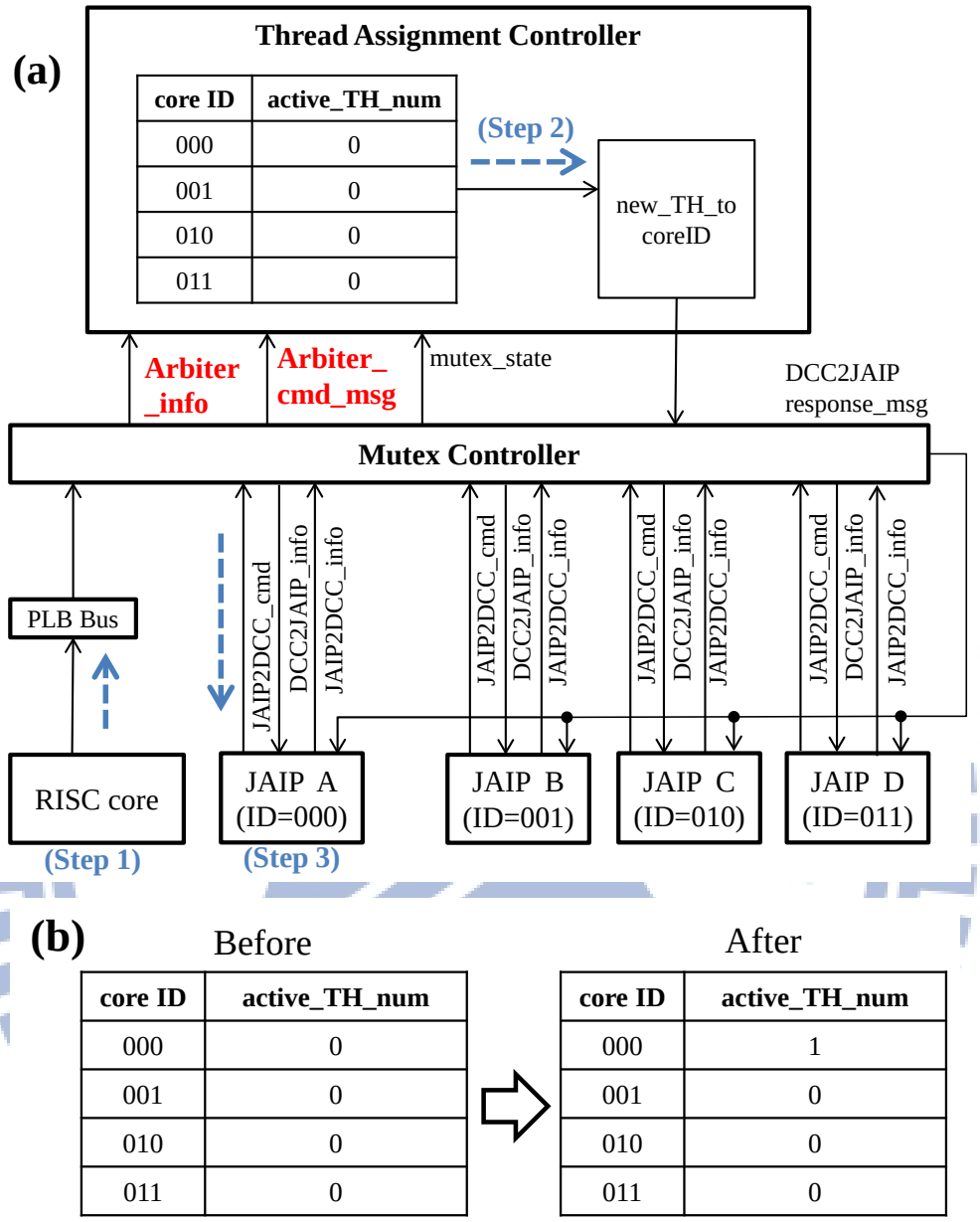



圖 28 Thread assignment controller 工作流程，範例 1

(a)Thread assignment controller 工作流程示意圖 (b)JAIP information table 變化過程

接著將 arbitor_cmd_msg 的內容傳到 Thread Assignment Controller，此時搜尋 JAIP information table 內每個 entry 的 active_TH_num 值，並記錄 active_TH_num 最小值所在的 entry index，如果存在 2 個以上 entries 的 active_TH_num 為最小值，則 Thread Assignment Controller 預設選擇 entry index 較小的(圖 28 (a)的 step 2)，搜尋 JAIP information table 完成後選擇 JAIP A (entry index = 0)，於是 new_TH_to_coreID = 000，把

arbiter_info 傳到 JAIP A 的 DCC2JAIP_info。當 mutex_state = SEND_RES_MSG，DCC2JAIP_response_msg 值為 000100010，代表這個 request 由 RISC-core 送出(DCC2JAIP_response_msg[5:3]=100)，由 JAIP A (DCC2JAIP_response_msg [8:6] = 000)儲存 main thread 執行資訊並且開始執行 main thread (DCC2JAIP_response_msg [2:0]=010)，DCC2JAIP_response_msg 訊號不需連接到 RISC-core，RISC-core 在此只用來啟動 main thread。最後 Thread Assignment Controller 將 new_TH_to_coreID 的數值作為 JAIP information table 的 entry index，更新 JAIP information table 內每個 entry 儲存的數值，如圖 28(b)。

假設 main thread 在 JAIP A 上執行，當 main thread 呼叫 Thread.start()產生 new thread (如圖 29(a)的 step 1)，此時 JAIP A 的 JAIP2DCC_cmd = 11，其 thread object 的參考位置、Class ID 與 method ID 等資訊被傳到 Data Coherence Controller，如同前一個例子 Mutex Controller 依照 Arbiter 機制將特定 JAIP 處理器的 JAIP2DCC_info 與 JAIP2DCC_cmd 分別傳到 arbiter_info 與 arbitor_cmd_msg。Thread Assignment Controller 此時搜尋 JAIP information table 內每個 entry 的 active_TH_num 值，記錄 active_TH_num 最小值所對應的 entry index。在此範例中完成搜尋後選擇 JAIP B (entry index=001) 容納且執行此 new thread，於是 new_TH_to_coreID = 001，其餘的行為與上個範例相同。當 mutex_state = SEND_RES_MSG，DCC2JAIP_response_msg = 001000010，代表此 request 由 JAIP A 送出且 new thread 被分配至 JAIP B。

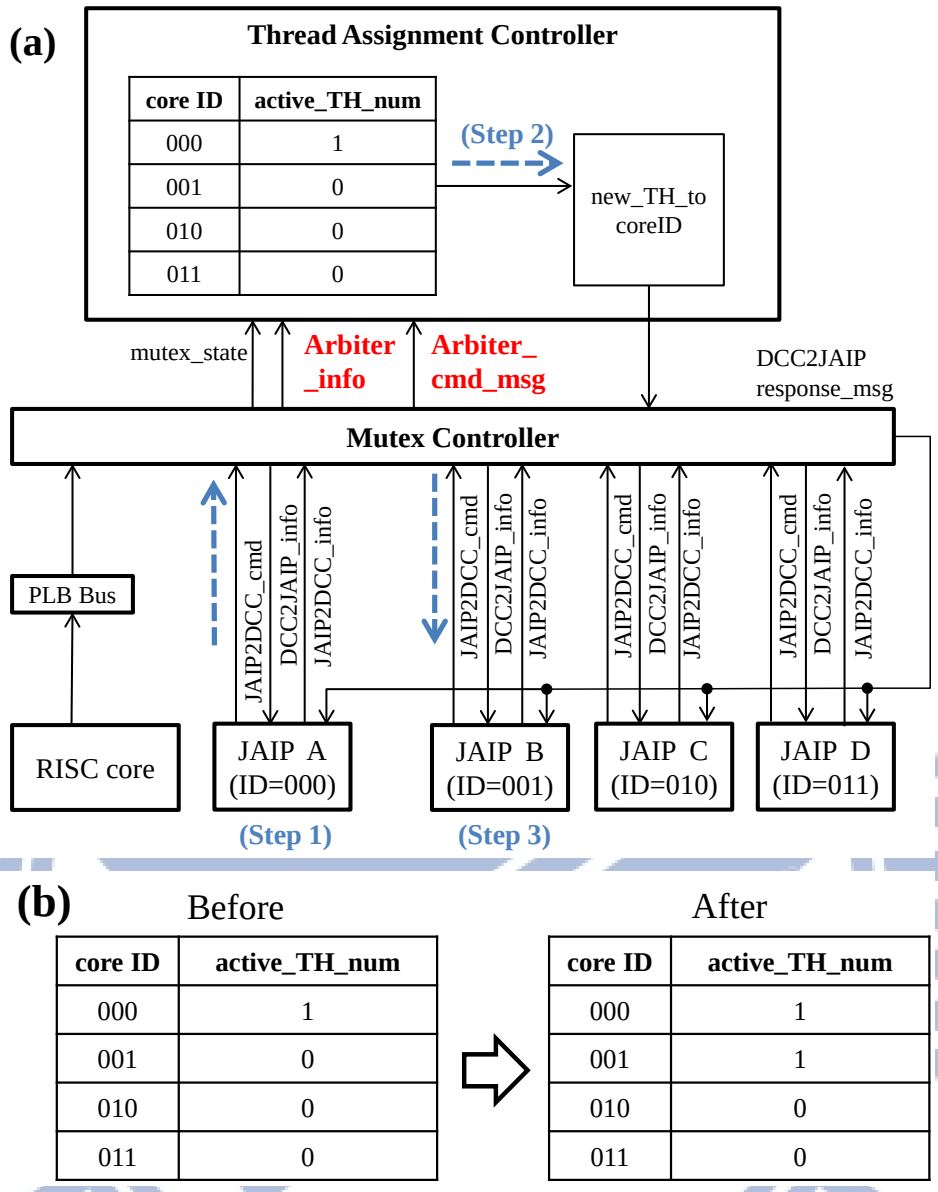



圖 29 Thread assignment controller 工作流程，範例 2

(a)Thread assignment controller 工作流程示意圖 (b) JAIP information table 變化過程

同樣在圖 30 範例中，當 JAIP C 的 current thread 呼叫 Thread.start()產生 new thread，如同圖 29(a)相關資訊被傳到 Data Coherence Controller，當 Thread Assignment Controller 查找 JAIP information table (圖 30(a)的 step 2)，發現所有 entries 的 active_TH_num 值都相同，故進一步選擇 entry index 值最小的 JAIP A。當 mutex_state = SEND_RES_MSG，DCC2JAIP_response_msg = 000010010，代表此 request 由 JAIP C 送出且 new thread 被分配至 JAIP A，此時 JAIP A 的 Thread Manager Unit 發現有 2 active threads 需要輪替執行指令因此 temporal multithreading 機制被啟動。

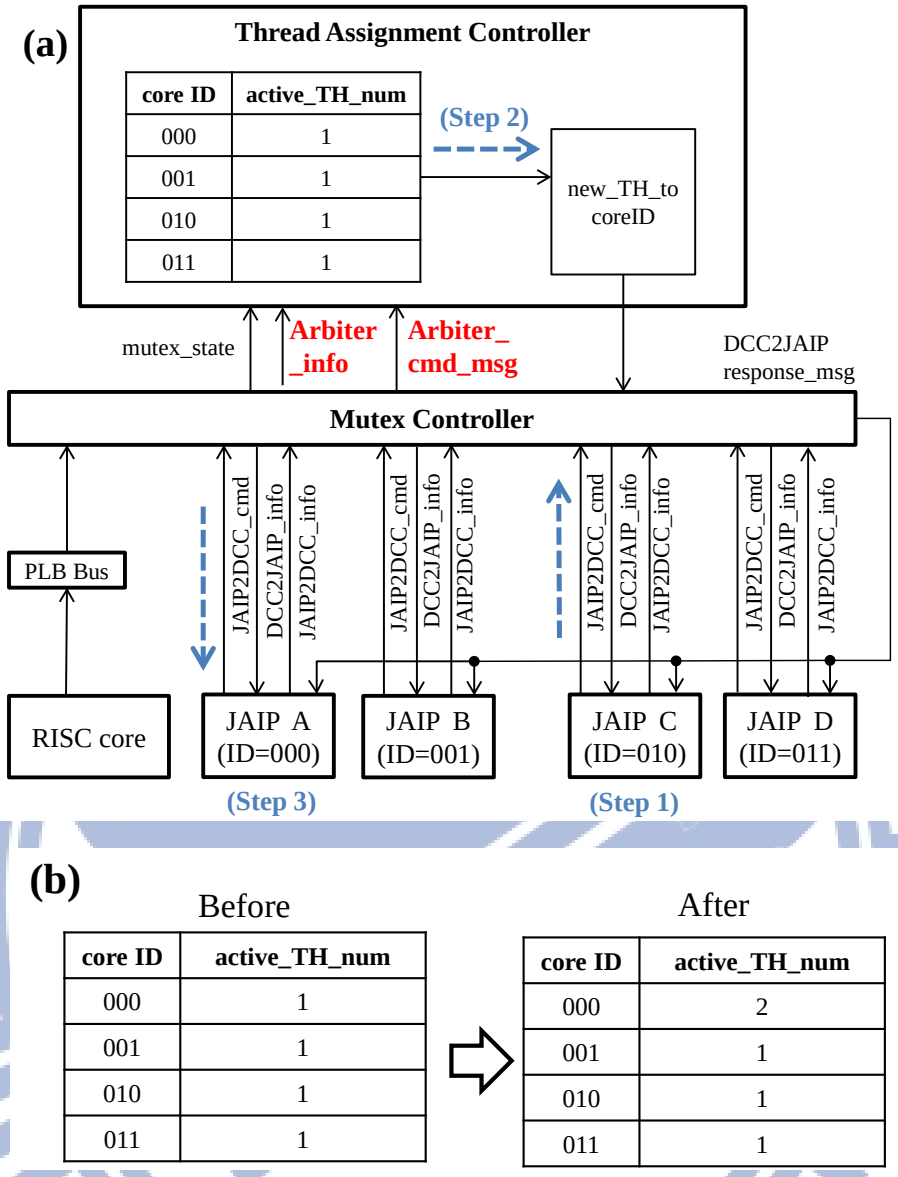



圖 30 Thread assignment controller 工作流程，範例 3

(a) Thread assignment controller 工作流程示意圖 (b) JAIP information table 變化過程

### 3.4.3. Lock Object Accessing Controller

圖 31 說明 Lock Object Accessing Controller 的設計。在 Java 程式執行期間，lockObj_curCount 儲存目前被某些 threads 占用的 lock object 個數，當多核心 JAIP 處理器執行指令時，如果有 2 個以上 threads 想要取得某個 lock object 並且目前沒有被其他 thread 占用，則第一個取得此權限的 thread 便會累加 lockObj_curCount 的值；當某個 thread 釋放該 lock object 且同時不存在相關 waiting threads 時遞減 lockObj_curCount 的值，其餘狀況皆不更改 lockObj_curCount 的數值。

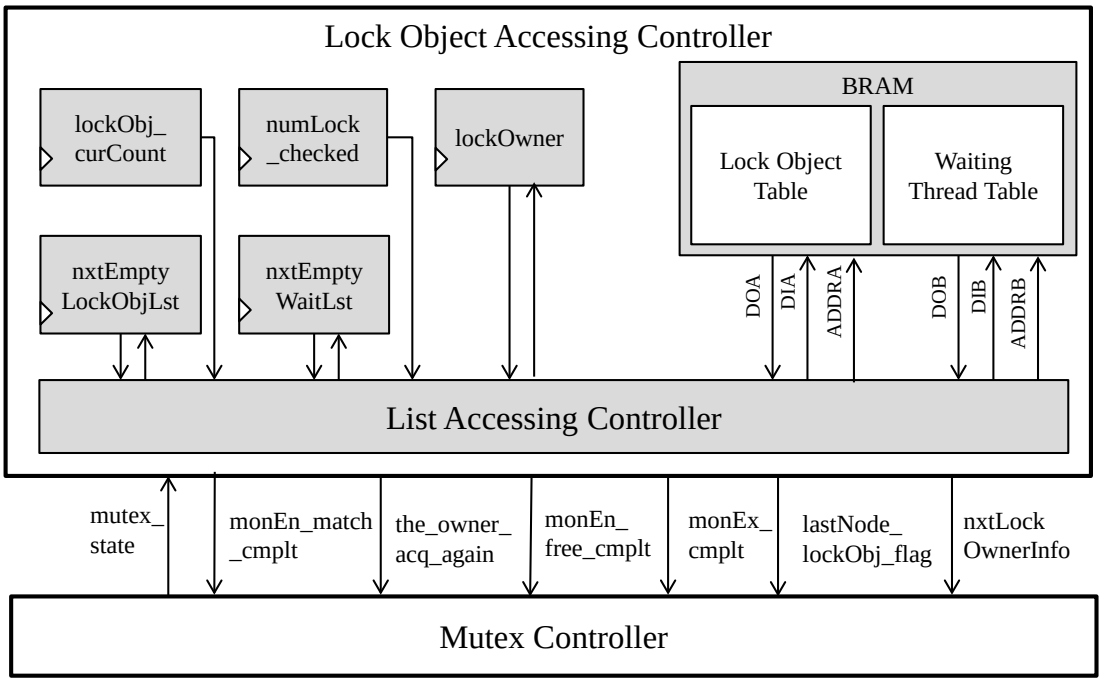



圖 31 The block diagram of Lock Object Accessing Controller

Waiting Thread Table 與 Lock Object Table 分別存放目前所有 waiting threads 與所有被占用的 lock objects 相關資訊，當執行 Java 程式時，這 2 個 tables 會用到多少 entries 取決於其應用程式複雜度，如果這些資訊全部使用 register 儲存，雖然可以加快查詢速度但是很容易使電路資源使用量過大。在此我們以一個 dual-port on-chip memory 實作 Waiting Thread Table 與 Lock Object Table，當 JAIP2DCC_cmd = 01 或 10，LOAC 可藉由 32-bit input port (DIA、DIB)與 32-bit output port (DOA、DOB)同時讀寫 Waiting Thread Table 與 Lock Object Table 的 entry，這 2 種 list entry 格式如圖 32 所示，valid bit 代表目前這個 entry 是否已被使用，其預設值為 0；next node address 為一個 7-bit address 指向另一個 waiting thread entry，所有等待同一個 lock object 的 waiting threads 在 Waiting Thread Table 內部將形成一條 singly linked list、每一個 lock object entry 的 next node address 欄位將會指向 Waiting Thread Table 內部某條 linked list 的 head node、在此 head node 指的是擁有 lock 權限的 thread (lock 擁有者)其 ID number，在我們的設計中 Waiting Thread Table 內部每一條 linked list 的 tail node 的 next node address 欄位其值為 0x7F 代表為預設值 NULL；counter 欄位代表目前 lock 擁有者重複取得幾次 lock object，當一個

waiting thread entry 的 valid bit = 0，此時該 counter 欄位為預設值 0；valid bit = 1 時，該 counter 欄位的值為 1，在我們的設計中任意 thread 成功取得 lock object 之後，它有資格重複取得相同 lock object 多次，累加 counter 欄位的值。

(a)

| Valid bit [31] | Core ID [30:29] | Thread ID [28:22] | Counter [21:16] | Reserved [15:7] | Next node Address [6 : 0] |
|---|---|---|---|---|---|

(b)

| Valid bit [31] | Object reference address [30 : 7] | Next node Address [6 : 0] |
|---|---|---|

圖 32 The format of (a) waiting thread entry and (b) lock object entry

以圖 33 為例，Lock Object Table 第 1 個 entry 儲存 Object L0 的參考位置，其 next node address 欄位指向 Waiting Thread Table 第 6 個 entry，這表示 Waiting Thread Table 內部存在一條與 Object L0 有關的 linked list，該 list 的 head node 在第 6 個 entry，其 core ID = 10 與 thread ID = 0000000 代表目前 lock 擁有者的資訊，第 6 個 entry 的 next node address 欄等於二進位數值 0000000 將指向第 1 個 entry；Waiting Thread Table 的第 1 個 entry 其 core ID = 01 以及 thread ID = 0000100 代表目前 lock 擁有者釋放 lock 之後，會將 lock 權限轉交到這個 thread，Object L0 在 Waiting Thread Table 內部從 head node 到 tail node 拜訪順序分別是第 6、第 1、第 5 個 entry，並且整個 linked list 長度為 3。Lock Object Table 第 4 個 entry 儲存 Object L2 的參考位置，其 next node address 欄位指向 Waiting Thread Table 第 7 個 entry，而第 7 個 entry 的 core ID = 00、thread ID = 0000001、counter = 000011，表示該 thread 成功取得 Object L2 的 lock 之後又重複取得 2 次 lock object L2。

使用 on-chip memory 的設計再配合圖 32 定義的資料格式，我們可以完整地紀錄 waiting threads 與 lock objects 資訊並且用少數的電路資源實作 LOAC。

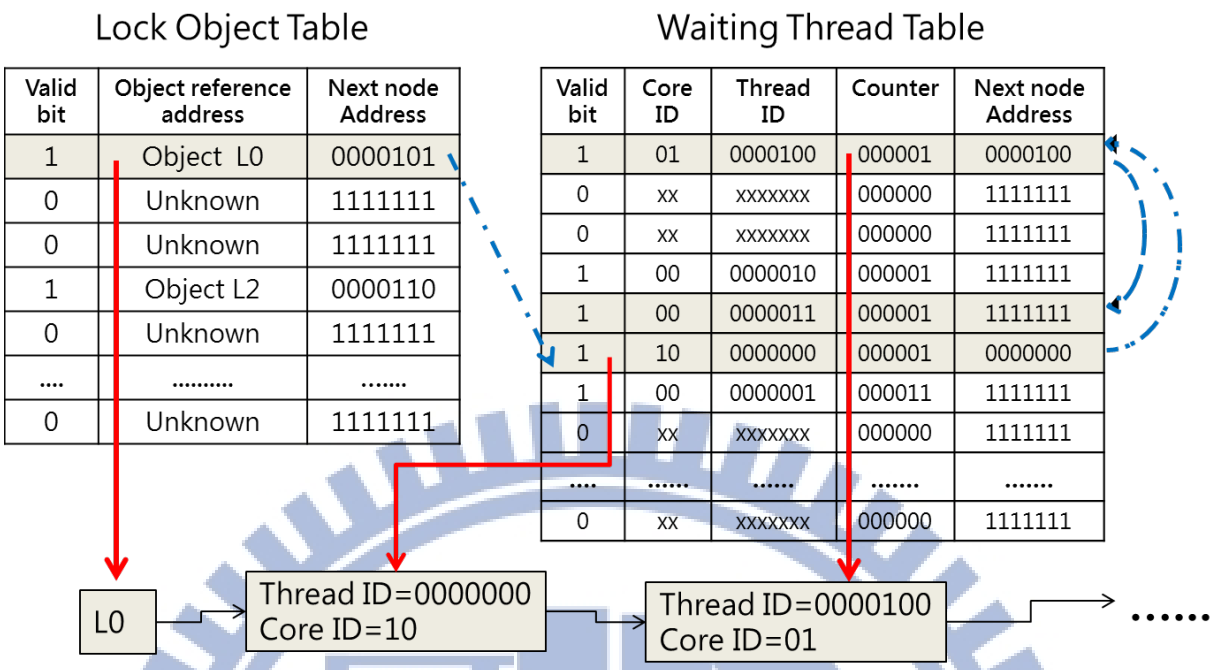
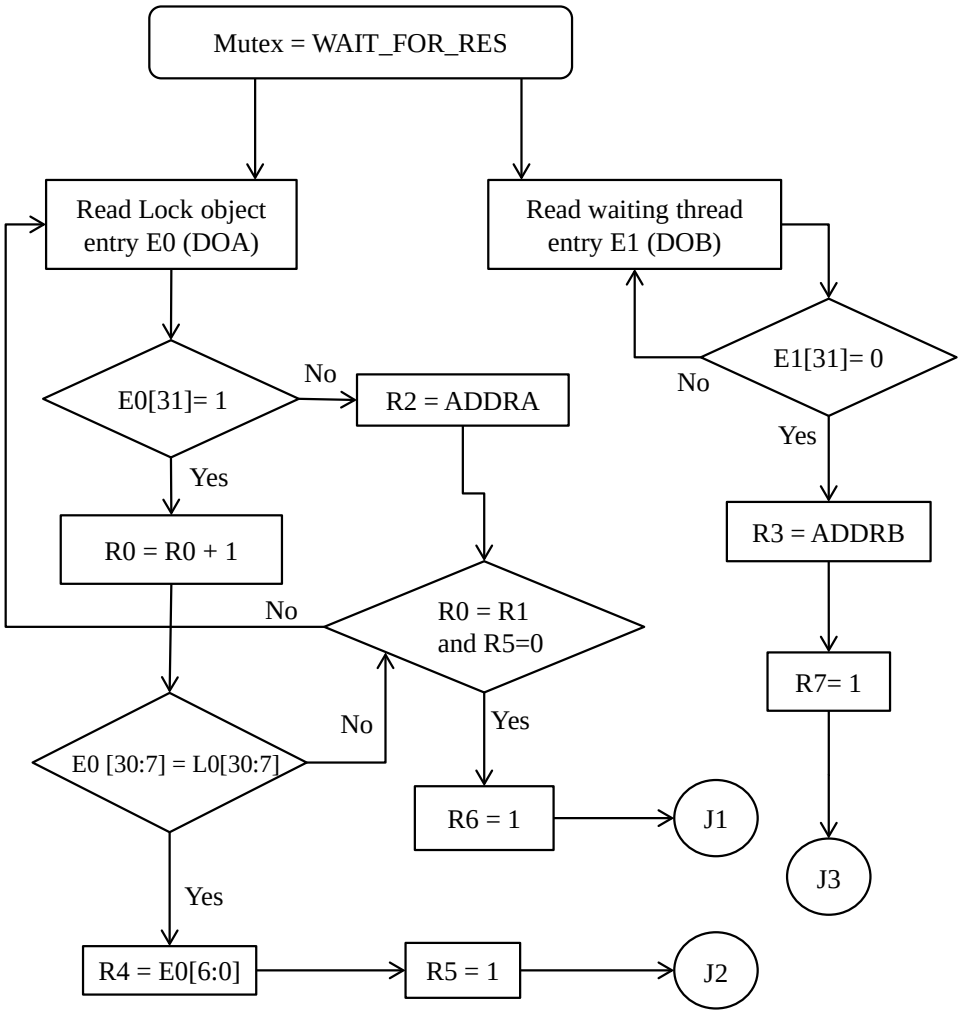



圖 33 waiting thread entry 與 lock object entry 與關係圖

我們以圖 9 與圖 10 舉例說明 LOAC 如何維護 Waiting Thread Table、Lock Object Table 以及其他內部 registers，假設 thread t1 從 JAIP A 處理器呼叫 method B3，此時藉由這個 JAIP 處理器發送 request signal 到 Data coherence Controller 請求取得或釋放 class B 底下的 lock object L0，JAIP A 的 Arbiter_Info 包含 L0 的參考位置與 t1 的 ID 資訊、LOAC 被啟動並且開始執行工作，以下幾種情況分別說明 LOAC 不同的執行流程。

**Case A**：當 t1 要請求取得 L0，arbitor_cmd_msg 的 command ID 欄位為 01，則 mutex_state 從 IDLE 轉成 WAIT_FOR_RES，所有內部 registers 皆被歸零。此時 LOAC 以一個內部 counter 作為 Waiting Thread Table (ADDRB，如圖 31) 與 Lock Object Table (ADDRA)索引位置，從第 1 個 entry 開始，每個 clock 往下一個 entry 同步查詢 Lock Object Table 與 Waiting Thread Table(圖 34)。如果讀取到的 lock object entry 其 valid bit = 1 (圖 34 的 E0[31]=1 為 true) 則累加 numLock_checked 表示已經檢查過這筆資料，否則目前 lock object entry 其 valid bit = 0，內部暫存器 nxtEmpty_LockObjLst 被用來記錄目前 lock object entry 對應的索引位置(ADDRA)。




每個代號皆表示 Lock Object Accessing Controller 內部暫存器名稱，詳見圖 31

| | |
|---|---|
| L0 : reference address of Object L0 | T1 : thread information (core ID, thread ID) |
| R0 = numLock_checked | R5 = lock_obj_match_flag |
| R1 = lockObj_curCount | R6 = lock_obj_free_flag |
| R2 = nxtEmpty_LockObjLst | R7 = find_empty_waitTHlist |
| R3 = nxtEmpty_WaitLst | R8 = search_waitLst_flag |
| R4 = lockOwner | R9 = the_owner_acq_again |

圖 34 LOAC 內部逐一比對 lock object 流程圖

當目前讀取到的 lock object entry 其 valid bit = 1，接著比對 lock object entry 的 Object reference address 欄位與 L0 的參考位置是否相同，如果相同則 lock_obj_match_flag 等於

1，最後比對 numLock_checked 與 lockObj_curCount。同時如果讀取到的 waiting thread entry 其 valid bit = 0，這表示目前此 waiting thread entry 沒有被使用，則 find_empty_waitTHlist 等於 1 且利用內部 register nxtEmptyWaitLst 記錄目前 waiting thread entry 對應的索引位置。此時會遇到 2 種情況：

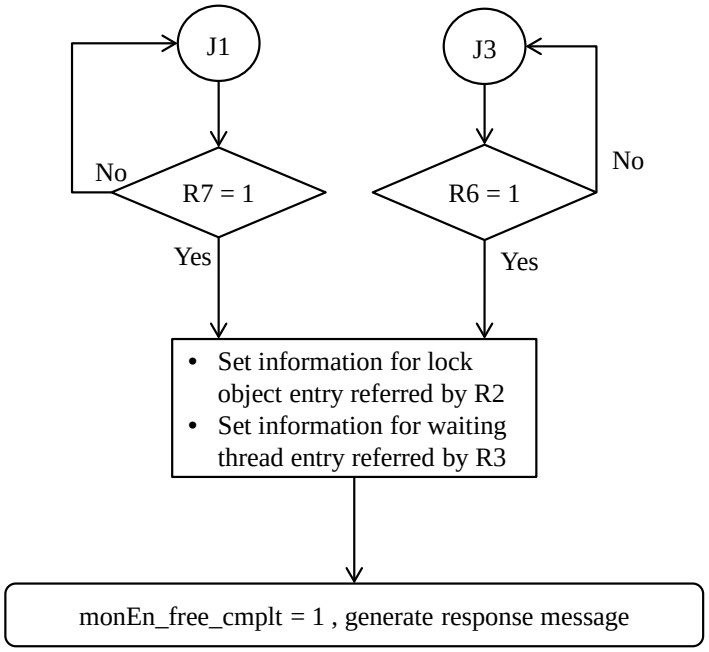



圖 35 LOAC 內部成功取得 lock object 之流程圖

J1、J3 表示流程連接點，請參考圖 34

**Case A-1：**。

如果 Lock Object Table 內沒有儲存 L0 參考位置，這表示 lock_obj_match_flag = 0 而且 numLock_checked = lockObj_curCount，t1 成功取得 L0，則內部訊號 lock_obj_free_flag 等於 1，當 find_empty_waitTHlist 等於 1 與 lock_obj_free_flag 等於 1 皆成立時期流程如圖 35，LOAC 以 nxtEmpty_LockObjLst 作為 Lock Object Table 的索引位置，寫入 L0 的參考位置與 nxtEmpty_WaitLst 的值，並且 valid bit 改成 1；同時以 nxtEmpty_WaitLst 作為 Waiting Thread Table 的索引位置，寫入 thread t1 的 core ID、thread ID(圖 36)。當 mutex_state = GEN_RES_MSG，LOAC 累加 lockObj_curCount 的值，並且由 Response Controller 產生回應訊號到 DCC2JAIP_response_msg，DCC2JAIP_response_msg 的 Response status 為 100，代表 t1 已成功取得 L0。

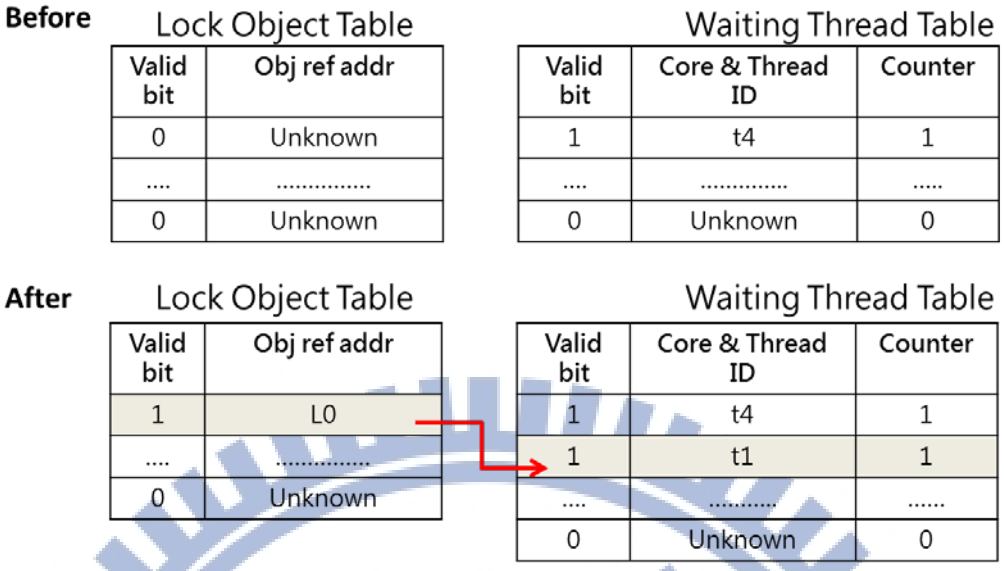



圖 36 範例：在 Waiting Thread Table 產生一條 Object L0 的 linked list N0

**Case A-2：**

如果 Lock Object Table 內儲存 L0 參考位置，這表示 L0 可能已先被其他 thread 取得權限 (e.g. thread t2) ，lock_obj_match_flag = 1，此時 LOAC 內部 register lockOwner 會暫存 DOA 的 next node address 欄位，它代表一個指向 Waiting Thread Table 內與 L0 相關的 linked list (以下簡稱 N0)，且 N0 的 head node 儲存 L0 的擁有者的 thread 資訊。當 find_empty_waitTHlist 與 lock_obj_match_flag 皆為 1，此時再觸發另一個內部信號 search_waitLst_flag 等於 1 (圖 37，R8 = 1)，LOAC 開始在 N0 內查詢 lock 擁有者的 thread ID 與 core ID，首先以 register lockOwner 當作 Waiting Thread Table 的索引位置，抓出 N0 的 head node，如果目前指向的 waiting thread entry (DOB)的 core ID 與 thread ID 欄位等於 t1 的 core ID 與 thread ID (圖 37 的 E1[30:22] = t1 為 true)，則內部信號 the_owner_acq_again 等於 1，否則等於 0，根據 the_owner_acq_again 的值，可以再細分下列 2 種情況：

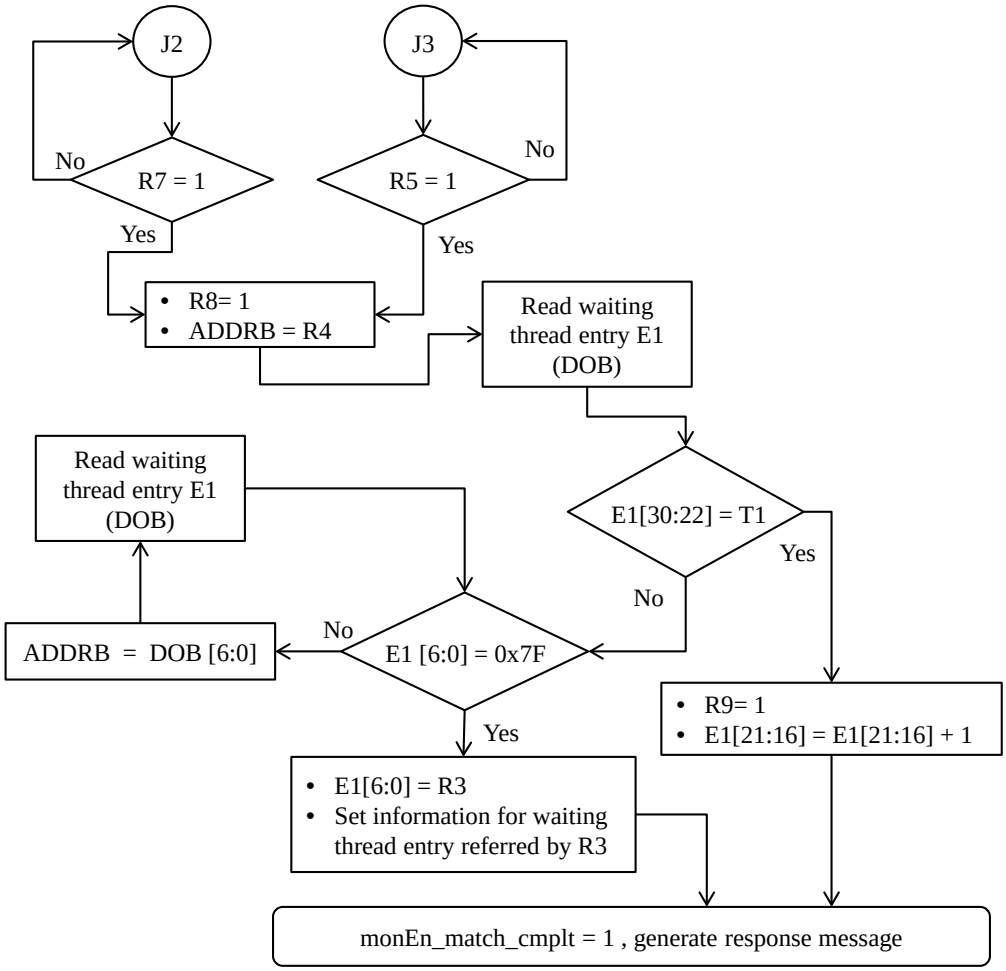



圖 37 查詢 Waiting Thread Table 內特定 lock object 的 linked list

J2、J3 表示流程連接點，請參考圖 34

**Case A-2-1：**the_owner_acq_again = 0。

此時可以確定 L0 被其他 threads 占用(如圖 38)，LOAC 必須在 Waiting Thread Table 內分配一個未使用的 entry 儲存 t1 的 core ID 與 thread ID，並且將這個 entry 加入到 linked list N0 末端，其查詢流程如圖 37。為了執行以上步驟，首先 LOAC 必須從 N0 中抓出目前 tail node 的索引位置並修改其 next node address 欄位，上一段提到 LOAC 內部 register nxtEmpty_WaitLst 數值在 search_waitLst_flag 等於 1 之前就已經先準備好，也提到 search_waitLst_flag 等於 1 時表示 LOAC 開始依序拜訪 N0 的每一個 node，LOAC 在每個 clock 開始，參考 DOB 的 next node address 欄位且檢查其數值是否為 NULL 或者指向下一個 waiting thread entry，如果不是 NULL 則以此 next node address 欄位當作 Waiting Thread Table 的索引位置，同樣地在 N0 拜訪下一個 node，依此類推；如果發現目前拜

訪的 entry 其 next node address 欄位是 NULL，此時表示已找到 N0 的 tail node，monEn_match_cmplt = 1、LOAC 必須將 nxtEmpty_WaitLst 的值寫到 tail node 的 next node address 欄位，同時將 nxtEmpty_WaitLst 的值當作 Waiting Thread Table 的輸入索引位置，儲存 t1 的 core ID 與 thread ID，注意 nxtEmpty_WaitLst 指向的 entry 其 next node address 數值為 0x7F，在 N0 內新增一個 node 紀錄 waiting thread 的工作就此完成。最後 LOAC 回傳訊號到 Mutex Controller 更改狀態，產生回應訊號輸出到 DCC2JAIP_response_msg，其 Response status 為 101，代表 t1 未成功取得 L0。

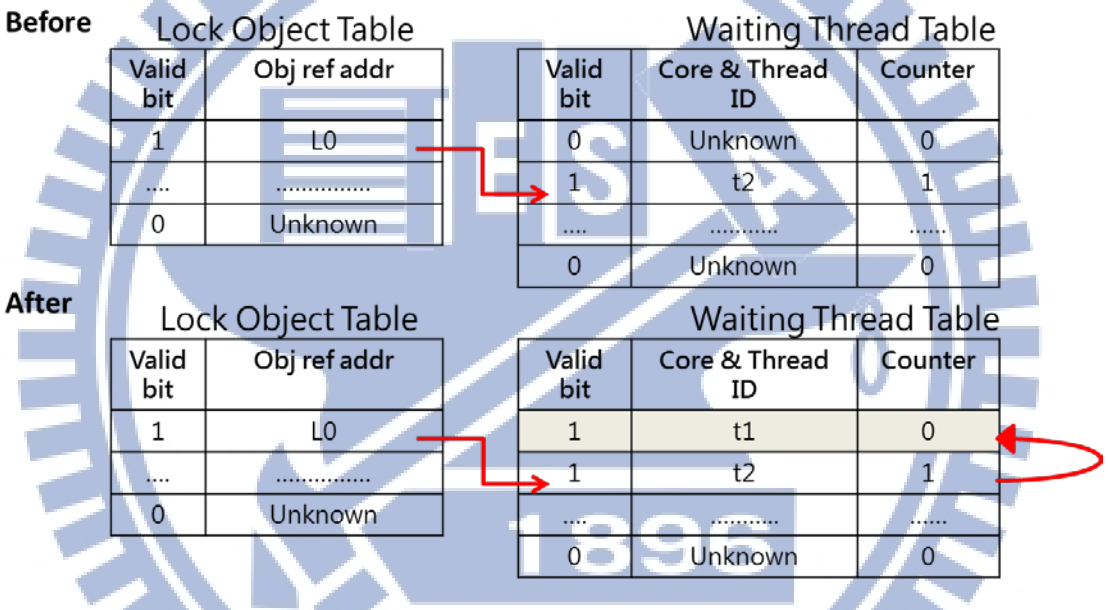



圖 38 範例：在 linked list N0 內新增一個 tail node

**Case A-2-2**：the_owner_acq_again = 1

此時可以確定先前 t1 已取得 L0 並且執行重複取得的工作(如圖 37 的判斷條件 E1[30:22]=T1)，則 LOAC 只需要將 lockOwner 的值當作 Waiting Thread Table 索引位置，累加 N0 的 head node 的 counter 欄位。最後 monEn_match_cmplt 等於 1，LOAC 回傳訊號到 Mutex Controller 產生回應訊號到 DCC2JAIP_response_msg，其 Response status 為 100 代表 t1 成功地重複取得 L0。

**Case B**：假設 thread t2 要釋放 L0 並且 thread t1 正在等待 L0，則 arbitor_cmd_msg 的

command ID 欄位為 10，且 mutex_state 從 IDLE 轉成 WAIT_FOR_RES，此時 LOAC 查詢 Lock Object Table 的流程與圖 34 相同，但是不需要到 Waiting Thread Table 查詢未使用的 entry，LOAC 一定可以找到 L0 在 Lock Object Table 對應的 entry (意即 lock_obj_match_flag 必定為 1)，因此當 lock_obj_match_flag = 1，則 search_waitLst_flag = 1，此時 ADDRA 指向 Lock Object Table 內 L0 的 entry，因此將 ADDRA 的數值暫時記錄下來，同時開始搜尋 linked list N0 (如圖 39)，由於先前每個 thread 取得 lock object 時 LOAC 已經維護各個 lock object 在 Waiting Thread Table 內對應的 linked list，所以在此我們只需要讀取 N0 的 head node，檢查 next node address 是否為 NULL，以決定是否把 L0 轉移給其他 thread。首先以 register lockOwner 當作 Waiting Thread Table 的索引位置，讀取 N0 的 head node，接著檢查 LOAC 取的 waiting thread entry (DOB)的 next node address 欄位是否等於 NULL (如圖 39 的 E1[6:0]=0x7F 是否為 true)、以及 counter 欄位是否大於 1 (如圖 39 的 E1[21:16]>1 是否為 true)。因此可分成以下 3 種情況：

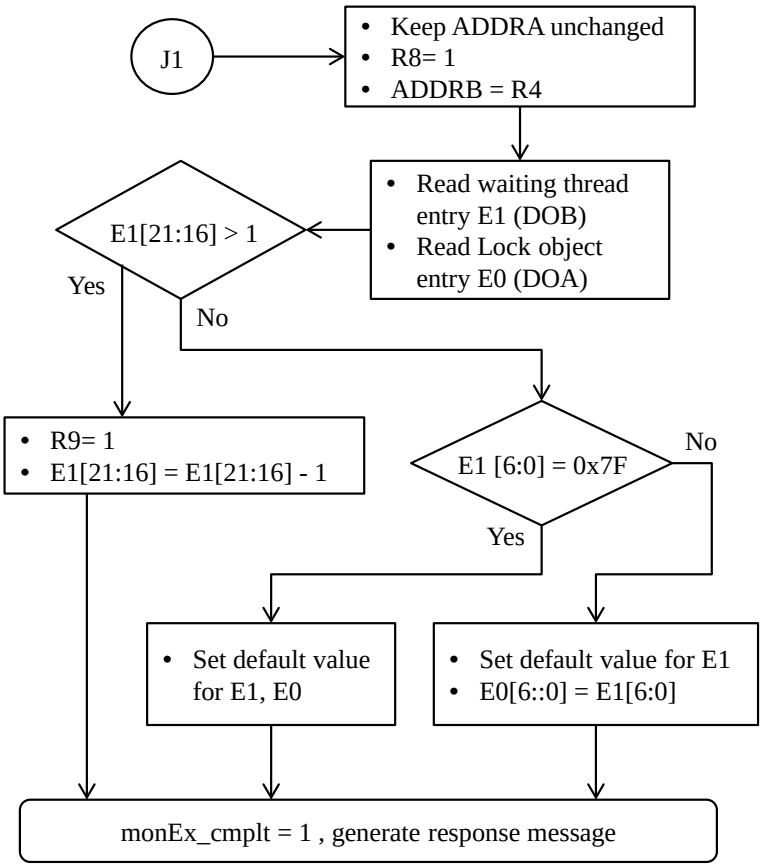



圖 39 lock 擁有者釋放 lock object 流程圖，J1 表示流程連接點，請參考圖 34

**Case B-1：**

waiting thread entry (thread t2)的 counter 欄位大於 1，此時 the_owner_acq_again 等於 1 表示先前 t2 重複取得 L0，我們只需要將 waiting thread entry (t2)的 counter 欄位值遞減一個單位即可，t2 仍然保持 L0 的權限。當 mutex_state = GEN_RES_MSG，產生回應訊號到 DCC2JAIP_response_msg，其 Response status 為 111，代表 t2 已完成釋放 L0 並且不作任何權限轉移。

**Case B-2：**

waiting thread entry (thread t2)的 counter 欄位等於 1 並且 next node address 指向下一個 waiting thread entry(如圖 40 的 thread t1)，這表示還有其他 thread 正在等待取得 L0，所以把 next node address 值寫到目前讀取的 lock object entry (L0)的 next node address 欄位使 lock object entry 直接指向 t1 的 waiting thread entry，最後把目前讀取的 waiting thread entry 改回預設值。

**Case B-3：**

waiting thread entry (t2)的 counter 欄位等於 1 並且 next node address 等於 NULL (如圖 41)，這表示當下沒有其他 waiting threads 正在等待取得 L0，此時 N0 僅剩下 2 個 nodes：一個 lock object entry 用來儲存 L0 的資訊，以及另一個 waiting thread entry 儲存 T1 的資訊，所以只需要將這 2 個 entries 還原預設值。

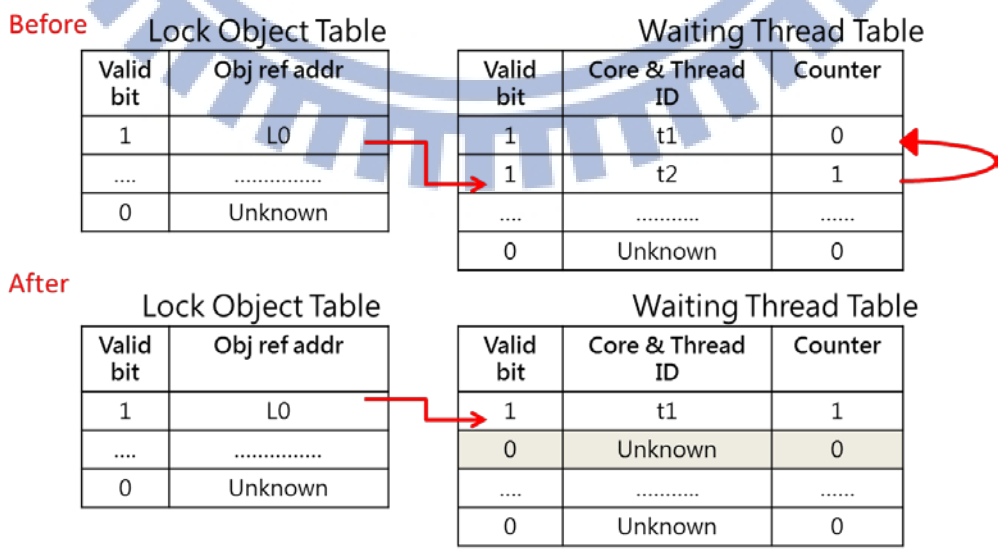



圖 40 範例：在 Waiting Thread Table 的 linked list N0 內移除 head node

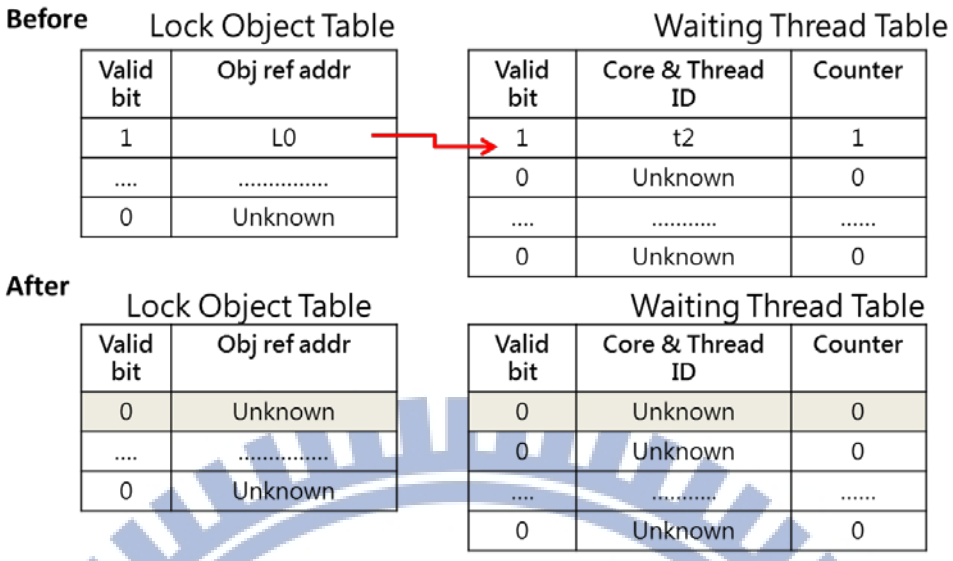



圖 41 範例：在 Waiting Thread Table 移除 Object L0 與 thread t2

# 第四章 Java 處理器 Temporal Multithreading 架構

本章節主要說明我們所修改的 Thread Manager Unit 底下 thread context 備份與回復流程、以及前一個章節中 ICCU 觸發 Thread Manager Unit 之後可能的電路行為。每一個 JAIP 處理器支援 multithreading 機制[1]， Thread Manager Unit(圖 42)由幾個主要元件所組成：Thread Queue 負責紀錄每個 thread 執行順序；Thread Control Block 紀錄每個 thread 之執行資訊；Stack Manager 用來復原或備份每個 thread 私有的 stack frames；Ping-pong Java stack(圖 4)內有 2 組 Two-level Runtime Java Stack，multithreading 執行環境下 Bytecode Execution Engine 只會存取其中一組 runtime stack memory，另一組 stack memory 由 Stack Manager 作復原或備份 stack frames 的動作。

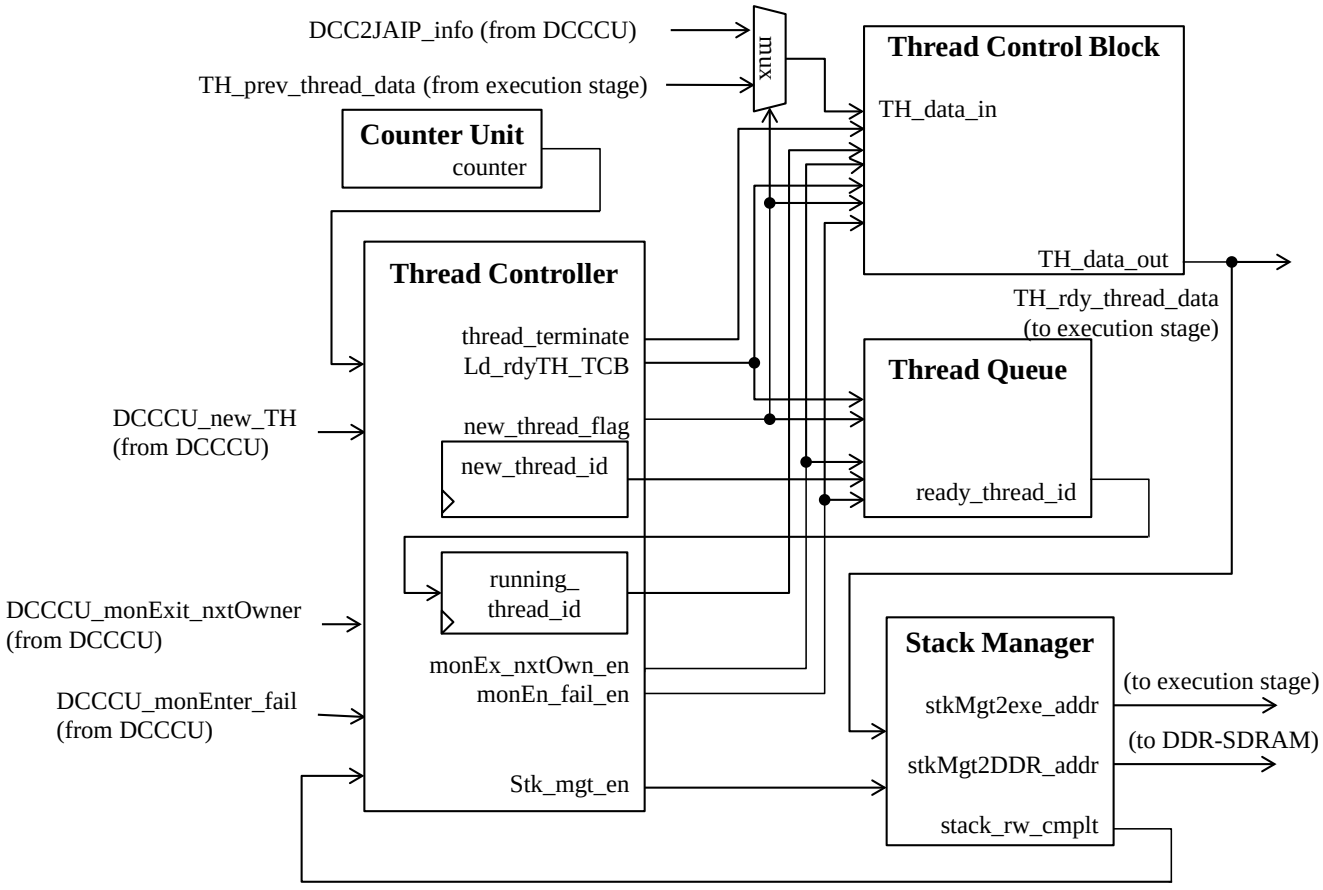



圖 42 The block diagram of Thread Manager Unit

Thread Manager Unit 是採用 Block multi-threading 的技術，使用 Counter Unit 計算

current thread 執行的時間。然而多核心版本 JAIP 處理器設計中[1]尚未支援 temporal multithreading 機制，意即每個 JAIP 未啟用 Thread Manager Unit 作 thread 的切換與輪替執行，並且先前 Thread Control Block 完全以 registers 儲存所有 Java threads 的執行資訊，這也會增加許多電路資源的使用，因此本小節我們將說明 Thread Manager Unit 中哪些模組被修改。

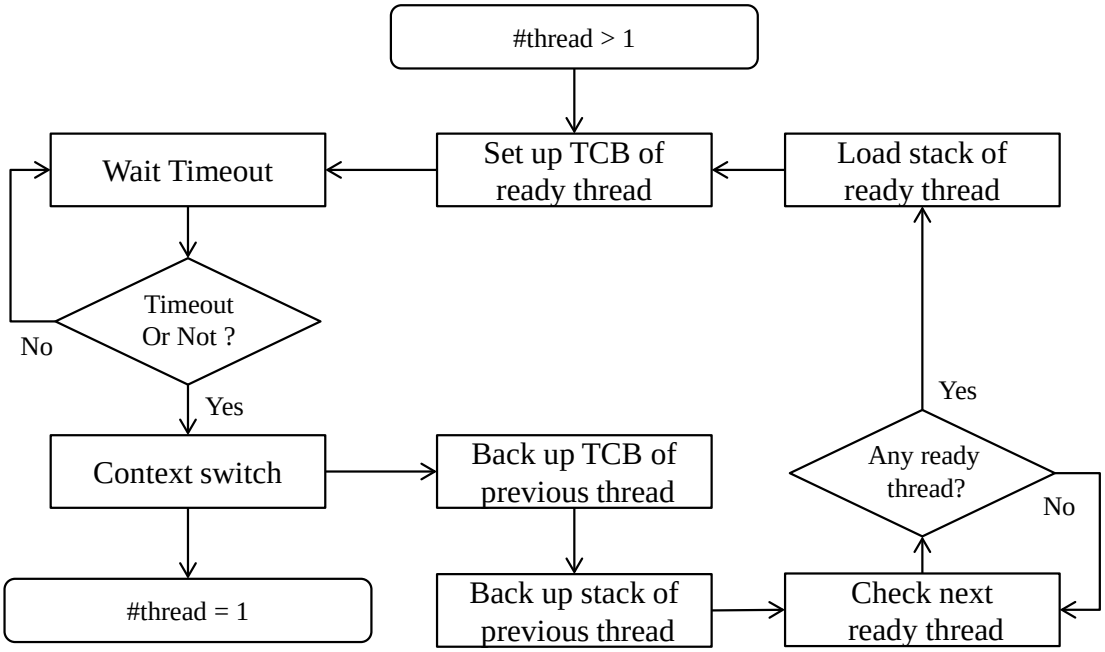



圖 43　切換 thread 流程

Section3-1 提到，任何 JAIP 處理器的 current thread 呼叫 start method 時，DSRU 的 state controller 會把 native_HW_en 設成 1，啟動 Hardware Native Interface，把輸出訊號傳到 ICCU。由 ICCU 彙整訊號送出請求到 Data Coherence Controller，Thread Assignment Controller 決定 new thread 被分配到哪個 JAIP 處理器，再觸發該 JAIP 處理器的 ICCU，把 ICCU_new_TH 設成 1 啟動 Thread Manager Unit (圖 19)。Thread Controller 把 new_thread_id 數值分配給此 new thread 作為 ID number，並告知 Thread Control Block 把 new thread 的執行資訊如 stack pointer、variable pointer、program counter、method ID、new thread 的 object reference 等初始化以及把 new thread 的 ID number 放入 Thread Queue 中排班並等待取得執行權。儲存 new thread 工作完成後 Thread Controller 再發送信號到 ICCU 與 Hardware Native Interface，直接把 native_HW_cmplt 設為 1 完成呼叫 start method

的動作。

一旦 Thread Manager Unit 偵測到存在 2 個以上的 active threads 時，每個 thread 就必須輪流切換執行(圖 43)，在我們的設計中是採用 Block multi-threading 的技術[1]，所以當 current thread 在 Bytecode Execution Engine 執行到一定的時間片段時 Thread Controller 會傳輸訊號使 running thread 停止執行，由 Thread Controller 執行 context-switching，相關動作如下：(1)Thread Controller 啟動 Method Area Manager，載入 ready thread 的 current method image，如果 Method Area Circular Buffer 沒有暫存這個 method image 則由 Method Area Manager 控制 system bus 訊號從 DDR memory 載入此 method (2)Thread Controller 發送訊號，讓 Bytecode Execution Engine 讀取 next ready thread 的執行資訊，同時 current thread 的執行資訊先暫存至另一組 registers，之後再依序寫回 Thread Control Block (3)切換 Bytecode Execution Engine 使用的 stack frames，使 ready thread 執行指令時可以讀寫正確的 stack frames。下個 clock 就可以開始執行另一個 thread 的指令，此做法可以降低 context-switching 的時間。

而在 context-switching 完成後，Thread Controller 動作如下：(1)把 previous thread 執行資訊寫回到 Thread Control Block (2)觸發 Stack Manager 開始備份 previous thread 所使用的 stack frames 傳到 DDR memory (3)如果 Thread Queue 內存在一個 ready thread，此時 Ld_rdyTH_TCB 等於 1 (圖 42)，從 Thread Queue 中讀取 next ready thread 的 ID number (4) 用步驟(3)讀取到的 ID number 啟動 Stack Manager 準備 next ready thread 使用的 stack frames (5)用步驟(3)讀取到的 ID number 到 Thread Control Block 讀取 next ready thread 的執行資訊。在步驟(2)跟(5)中 Thread Controller 必須先把所需資訊傳入 Stack Manager，包含每一個 thread 的 stack frames 在 DDR memory 中備份的位置、stack pointer、以及 previous thread 在 Ping-pong Java stack 內非執行中的 stack memory，把 previous thread 的 stack frames 內容寫回 DDR memory，或從 DDR memory 讀取的內容寫入 Ping-Pong Java Stack 之中被 Stack Manager 控制存取的 stack memory。

而當 current thread 執行完畢要終結時，會發出訊號通知 Thread Manager Unit 此 current thread 已經終止。Thread Controller 收到 thread 終止的訊號後，會等待上一段步驟

(1)至步驟(5)完成後再切換執行權到 Thread Queue 之中所選出的 next ready thread，Thread Manager Unit 中 thread 數量遞減一個單位，此 thread 將不再被放入 Thread Queue，並且刪除此 thread 儲存在 Thread Control Block 的執行資訊。

## 4.1. Thread Controller

Thread Controller 主要接收所有傳入 Thread Manager Unit 對於 threads 運作的相關訊號，並產生相對應的控制訊號到 Thread Queue、Thread Control Block 或其他內部元件。Thread Controller 的工作包含管理 new thread 產生時所需要的初始化資訊、管理 thread 終止執行、處理同步機制的電路被啟動時，產生訊號到 Thread Queue 與 Thread Control Block 修改某個 thread 的狀態、當 Thread Controller 執行 context-switching 時也會修改相關訊號到 Bytecode Execution Engine 各元件切換到 next ready thread 的執行資訊。圖 44 說明 Thread Controller 的狀態圖。由這些狀態來控制單一 JAIP 處理器內 threads 的操作，每個狀態說明如下：

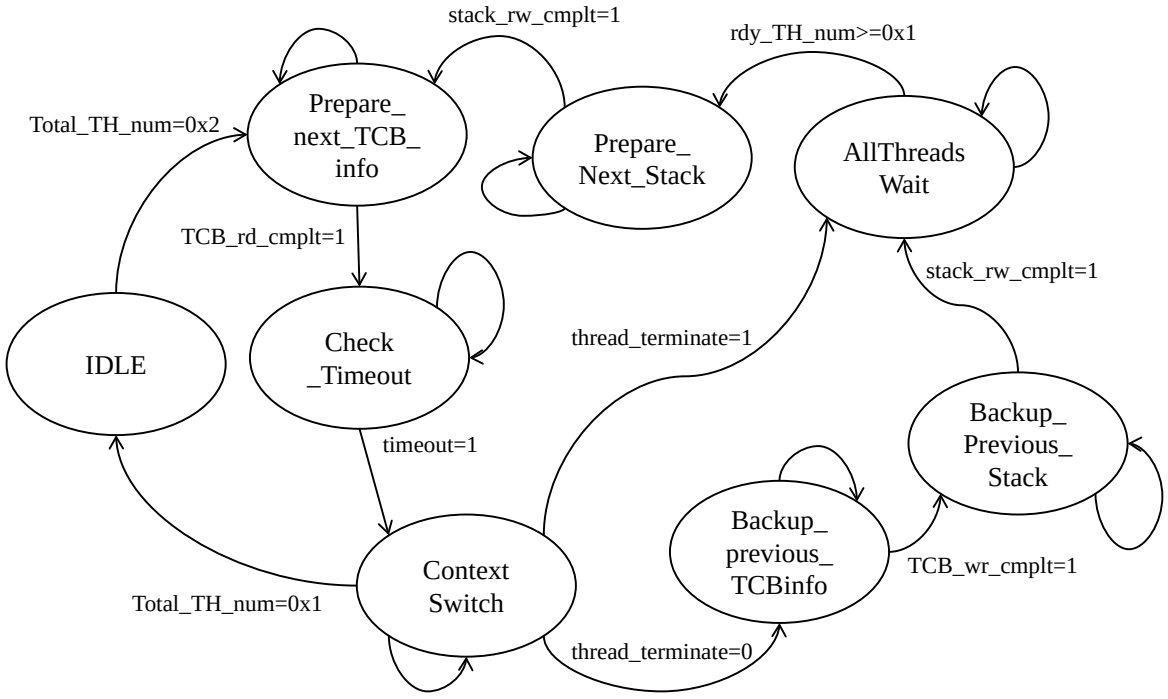



圖 44 The state diagram of Thread Controller

- **Idle：**

在 Idle 狀態時，表示此 JAIP 處理器底下只有一個 active thread，Thread Controller 將一直維持在此狀態，Thread Manager Unit 其他元件不會有任何動作，因此不會對執行 Java 程式造成任何影響。直到 new thread 被產生使得 Thread Manager Unit 要維護大於 2 個 active threads，此時才會轉換到下一個狀態。

- **Prepare_next_TCB_info：**

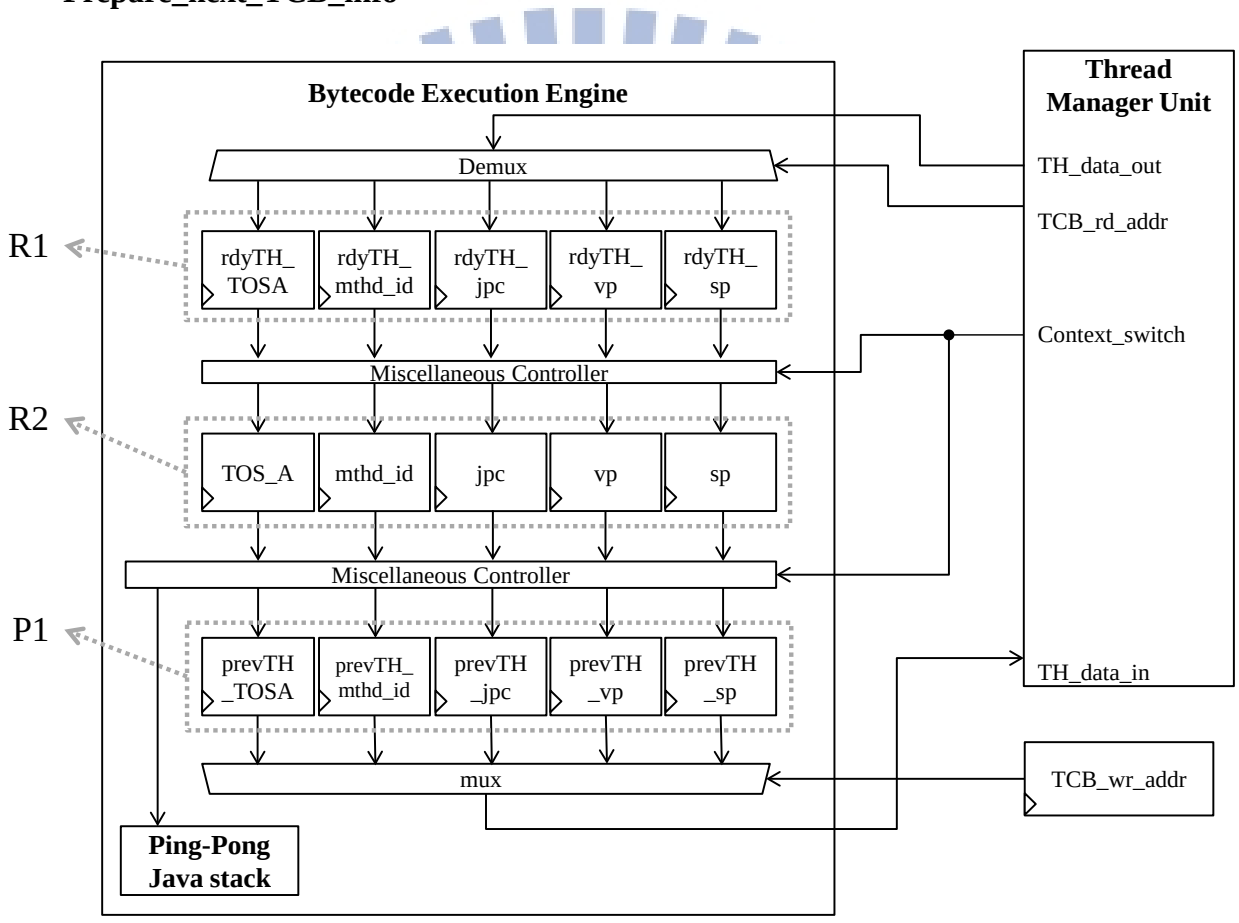



圖 45 執行 context-switching 時，threads 執行資訊切換示意圖

這個狀態主要功用是從 Thread Control Block 讀取 next ready thread 的執行資訊，以圖 45 為例，透過 TH_data_out 把這些執行資訊依序傳到 Execute stage 的一組 internal registers rdyTH_xxx (如圖 45 的 R1 區塊)，其中 Miscellaneous Controller 代表修改 R1、R2、P1 這些區塊的 internal registers 相關電路。我們使用一個 on-chip memory 來實作 Thread Control Block，由於 on-chip memory 的 data ports 限制並且每

個 thread 的執行資訊皆由多個 32-bits special-propose registers 組成，所以任何一個 ready thread 的執行資訊從 Thread Control Block 讀取到 Execution Engine 的 internal registers 必須花 8 clocks。之後便拉起 Thread Controller 內部信號 TCB_rd_cmplt (圖 44)進入下個狀態。

- **Check_Timeout：**

主要用來檢查 current thread 在此回合剩下多少可使用的 time slice，這個狀態下 Thread Controller 會持續讀取 Counter Unit 的輸出並且會遇到 3 種情形：(1)如果 current thread 在此回合執行時間已經大於或等於我們設定的 time slice 數值，而且 Fetch Stage 解析出來的 bytecode 不是複雜指令，則 Thread Controller 發送控制訊號給 Bytecode Execution Engine 記錄 current thread 目前執行到哪個 method (e.g. class ID、method ID)的哪一段指令(Java program counter)，以及記錄 stack frames 狀態(e.g. stack pointer、variable pointer) (2)如果 current thread 在還沒執行到一個時間片段時已經先發出 thread 終止的訊號，Thread Controller 直接把內部信號 timeout (圖 44)設成 1，進入下個狀態 (3)如果 current thread 執行 monitorenter 指令並且由 Data Coherence Controller 回傳的訊息表示取得 lock object 失敗，此時 Thread Controller 把內部信號 timeout (圖 44)設成 1。當目前狀態為 Check_Timeout 並且 timeout=1 的時候，Thread Controller 把 current thread 標記為 waiting state，最後進入下個狀態。

如果 current thread 此次使用的執行時間已經大於或等於我們設定的 time slice 參數，在這個時間點 Decode Stage 以及 Execute Stage 可能尚存有正在解碼或執行的 j_code。因此為了讓 thread 切換上更為簡便以及簡化所要記錄執行資訊的電路，在執行到達一個時間片段時，並不會清除目前已存在於 Decode Stage 以及 Execute Stage 所要執行的 j_code，而是等待這 2 個 stage 的指令執行完畢之後，Thread Controller 內部信號 timeout=1，到下一個狀態再執行 context-switching。如此一來便可降低管理 threads 的資源。

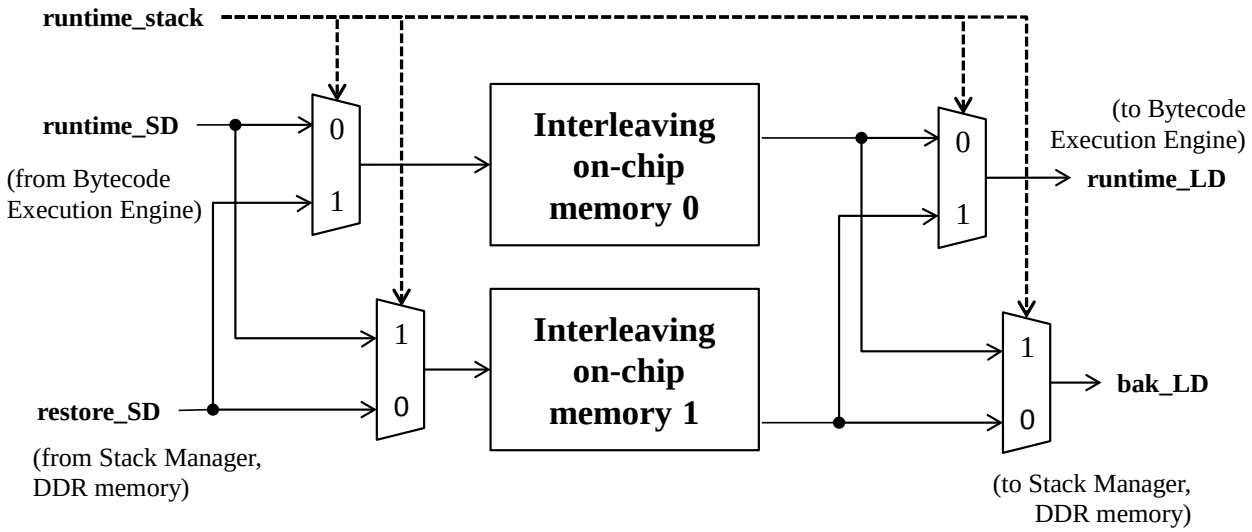



圖 46 Ping-pong Java stack memory 控制示意圖

- **ContextSwitch：**

首先將 ready thread 的 current class ID 與 method ID 將傳入 Method Area Manager (圖 8) 並且把 Method Area Manager 的內部訊號 ClsLoading_stall 設定為 1，此時便暫停 Bytecode Execution Engine 執行，啟動這兩個電路載入對應的 method bytecode，還原 ready thread 上次執行的環境；當 Method Area Manager 完成工作後，此時 ClsLoading_stall 等於 0，Thread Controller 必須切換 Bytecode Execution Engine 讀取 thread 執行資訊的來源。以圖 45 為例，ready thread 的執行資訊此時儲存在 R1 區塊，而 Bytecode Execution Engine 執行指令時預設參考 R2 區塊的內容，也就是 current thread 的執行資訊。此時將 R1 區塊所有 internal registers 內容複製到 R2，同時 R2 區塊的內容複製到 P1 區塊，此時也會改變 Ping-pong Java stack (圖 46)的內部訊號 runtime_stack 的數值，來回切換 Bytecode Execution Engine 執行指令時使用的 interleaving stack memory。故 current thread 與 next ready thread 的 stack frames 切換也只需要 1 clock 即可完成。

在這個狀態下當 Method Area Manager 的 ClsLoading_stall 等於 0，如果 Thread Manager Unit 維護的 thread 數量小於或等於 1，則下個狀態為 Idle；否則再細分以下 2 個情況：如果 thread_terminate (圖 44)等於 1，這表示 previous thread 已經終止

執行，不需要備份 previous thread 執行資訊與 stack frames，故下個狀態為 AllThreadsWait；否則需要備份 previous thread 的執行資訊與 stack frames，所以下個狀態為 Backup_previous_TCBinfo。

- **Backup_previous_TCBinfo：**

這個狀態主要功用是把 previous thread 的執行資訊更新到 Thread Control Block 對應的位置，以圖 45 的 P1 區塊為例，Thread Controller 透過 TH_data_in、TCB_wr_addr 把這些執行資訊依序寫到 Thread Control Block 對應的位置。同樣把任何一個 thread 的執行資訊全部寫回去 Thread Control Block 必須花 8 clocks。之後便拉起 Thread Controller 內部信號 TCB_wr_cmplt 進入下個狀態。

- **Backup_Previous_Stack：**

由於在先前執行 context-switching 時 JAIP 會將 Ping-pong Java stack 中所使用的 stack memory 切換到另一組 stack memory，此時 previous thread 所操作的 stack frames 還保留在 Ping-pong Java stack 中(圖 46)。為了降低在 JAIP 內部管理 stack frames 所耗費之成本，在此狀態下 Thread Controller 觸發 Stack Manager，將 previous thread 的 stack frames 備份到 DDR-SDRAM 特定位置。由於 JAIP 處理器的是採用 Two-level Java Runtime Stack [3]的設計，主要是由 2 塊 Dual-Port on-chip memories 所組成，所以若 Stack Manager 完整的將整個 stack frames 複製到 DDR-SDRAM 上則傳輸作業時間會被拉長，因此 Stack Manager 會參考 previous thread 的 stack pointer (如圖 45 的 prevTH_sp)來做為傳輸長度，因為在執行時 stack memory 真正有用到的範圍也只是從 0 到 Stack pointer 這個位址，Stack Manager 只傳輸有使用的部分，藉以降低傳輸 stack frames 的時間。Stack Manager 完成 stack frames 備份的工作後觸發 stack_rw_cmplt 到 Thread Controller 改變狀態到 AllThreadsWait。

- **AllThreadsWait：**

多核心 JAIP 處理器環境下，考慮所有 threads 在某個時間內要取得某個 lock object，則有可能某個 JAIP 處理器底下 current thread 執行 monitorenter 取得 lock object 失敗並且同一個處理器底下所有其他 threads 已經先進入 waiting state，意即

這個 JAIP 處理器的 ready thread 數量等於 0，此時勢必由其他 JAIP 處理器的 current thread 把 lock object 權限轉移給這個 JAIP 處理器的某個 thread 恢復成為 ready state。這個狀態主要用來檢查(1)Thread Manager Unit 的是否存在 2 個以上 ready threads 或者(2)ready thread 數量等於 1 但是 current thread 取得 lock object 失敗。當這兩個條件都不成立時，Thread Controller 的狀態會停留在 AllThreadsWait，因為此時 Thread Queue 不存在任何 ready thread ID 可以讓 Thread Controller 切換到下一個 thread 執行指令；如果上述兩個條件中至少一個成立，則 Thread Controller 的 Ld_rdyTH_TCB=1 (圖 42)，到 Thread Queue 抓取 ready thread ID，利用這個 thread ID 到 Thread Control Block 讀取這個 ready thread 的 stack pointer。最後 Thread Controller 狀態切換到 Prepare_Next_Stack。

- **Prepare_Next_Stack：**

  在此狀態 Stack Manager 會開始將此 ready thread 原本存放在 DDR-SDRAM 中的 stack frames 傳輸到 JAIP 處理器並寫入到 Ping-pong Java stack 中目前由 Stack Manager 控制的 stack memory。與備份時相同，為了減短傳輸時間以降低對於正常執行的影響。在上個狀態就已取出此 ready thread 的 stack pointer 作為傳輸長度。在 context-switching 被執行前先把 ready thread 的 stack frames 準備好，而當要進行 context-switching 時，Thread Controller 只需切換 Bytecode Execution Engine 所使用的 stack memory 即可。利用此種作法，Thread Manager Unit 也不需要用額外硬體資源維護每個 thread 的 stack frames。同樣地 Stack Manager 完成 stack frames 載入的工作後觸發 stack_rw_cmplt 到 Thread Controller 改變狀態到 Prepare_next_TCB_info。

## 4.2. Thread Control Block

所有 Java VM threads 共享[6]Run-Time Constant Pool、method area、heap 的 data，而每個 thread 擁有自己的 program counter、stack pointer、variable pointer、stack frames、記錄每個 thread 目前在哪個 method 執行指令並且被分配一塊獨立的記憶體空間。當 Java VM 執行 context-switching 時必須備份 current thread 的相關資訊並且載入 next ready

thread 的執行資訊，Java VM 將每個 thread 執行資訊的儲存機制留給個別平台實作。

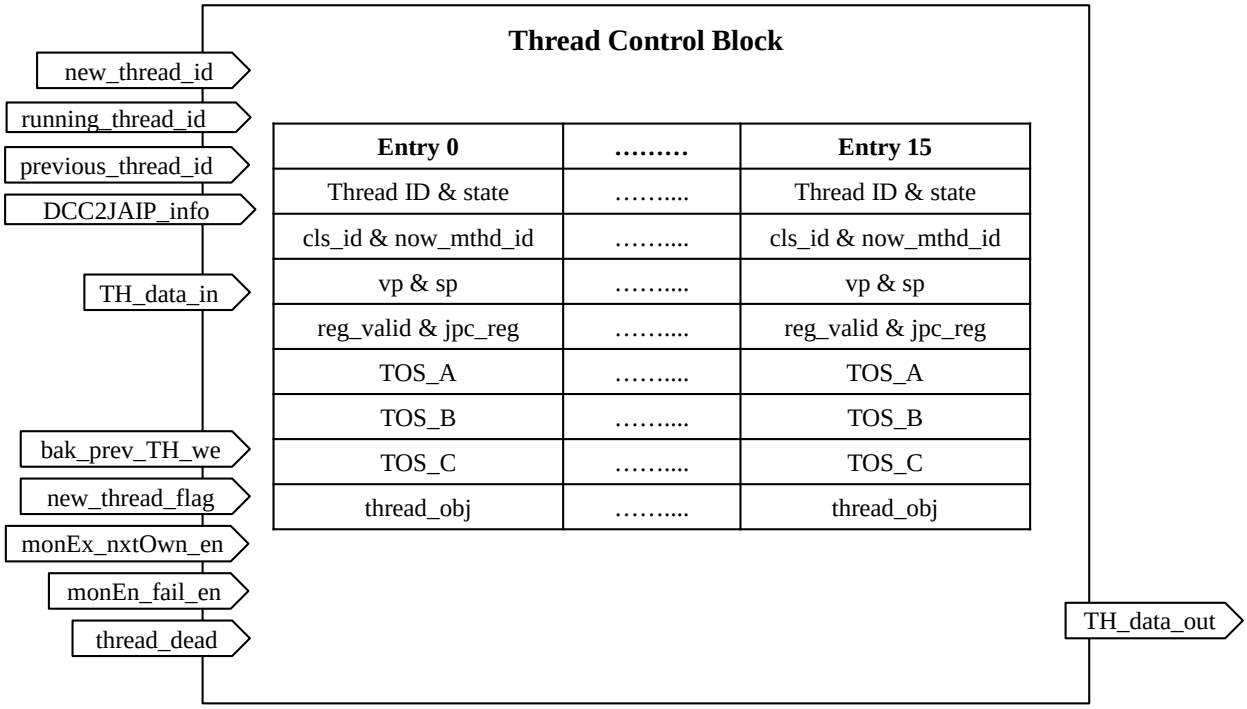



圖 47 每個 Thread 儲存在 Thread Controller Block 內的執行資訊

圖 47 描述 Thread Control Block 的儲存結構。每個 thread 的執行資訊皆由 8 個 32-bits special-purpose registers 所組成的集合。而每個 thread 所需要被記錄的執行資訊分別為：

(1) Thread ID

(2) Thread state

(3) thread 目前所執行的 class 與 method ID

(4) Java program counter

(5) stack memory 使用情形，包括 Variable pointer 和 Stack pointer

(6) thread 目前所執行的 method 其 local variable 個數

(7) Two-level Java Runtime Stack 最上方的三筆資料(TOS_A、TOS_B、TOS_C)

(8) Thread object reference

不同於先前的設計，我們使用一個大小為 2KB 的 on-chip block RAM 來暫存上述第

(2)項至第(8)項所有 threads 的執行資訊，使用 on-chip block RAM 暫存即時性運算資料可減少 registers 與 LUTs 的使用，並且只需花 1 clock 讀寫儲存的資料。Thread Control Block 可容納的 thread 數量目前最多 16 個，如果要擴充 Thread Control Block 可容納的 active thread 數量，我們只需加入幾條位置訊號線就可以存取 on-chip block RAM 中其餘未被使用的位置。

然而選擇存放這些資訊的原因是：透過存放 thread ID，可讓 Thread Controller 可直接透過 thread ID 到 Thread Control Block 取得執行緒資訊。為了支援同步機制的電路，在我們的設計下使用 2 bits 表示 thread state，二進位數值 00 代表這個 thread 已經終止執行或者不存在，01 代表 waiting state，10 代表 ready state。記錄 class 與 method ID、Variable pointer、Stack pointer 以及 Java program counter 這些訊息是 JAIP 在執行程式時的必要資訊。而 thread 目前所執行的 class 與 method ID，以及 current method 的 local variable 個數，是藉由每次呼叫方法時 DSRU 對 stack memory 的操作。因為在 Ping-Pong Java Stack 之中每個 2-level runtime stack 包含 4 registers 放置讀寫頻率較高的 local variable 0 到 3 [3]，所以需要 local variable 個數來判斷 current method 對於這 4 registers 的運用情形。而在 Two-level Java stack memory 中第一層僅使用 3 registers 來放置 stack memory 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 thread 切換時造成的成本耗費，因此我們選擇直接將這 3 registers 放置於 Thread Control Block，使得在切換時只要直接替換 JAIP 所使用的這 3 registers 即可，而不用再從第二層的 interleaving stack memory 中取出來更新這三個暫存器。以下個別說明 Thread Controller 更新 Thread Control Block 的幾種情況：

- 產生 new thread。根據圖 21 與圖 42，當 ICCU_new_TH 等於 1 時，new_thread_flag 等於 1 並且啟動 Thread Controller 新增 thread，直接把 register new_thread_id 的數值分配給 new thread 之後再自行累加一個單位。再從 Thread Control Block 內部尋找閒置的 entry (圖 47)，分配給 new thread 寫入下列執行資訊：(1)DCC2JAIP_info 此時儲存 new thread 必要的執行資訊例如起始 class ID、method ID 與 Thread object reference，將這些資訊寫到 Thread Control Block 內部的 on-chip block RAM (2)其他

執行資訊如 vp、sp、jpc_reg 等皆寫入預設值即可(3)new thread 的 ID number 與 thread state 同樣寫入到 Thread Control Block。

- 當 Thread Controller 狀態為 Backup_previous_TCBinfo，此時 previous thread 最新的執行資訊存放在 Bytecode Execution Engine(如圖 45)，Thread Controller 參考 previous thread ID 的值(previous_thread_id)到 Thread Control Block 查詢對應的 entry，再把 previous thread 執行資訊從 Bytecode Execution Engine 透過 TH_data_in 依序寫入 Thread Control Block。
- Thread Controller 利用內部 register running_thread_state 與 prev_thread_state 記錄 current thread state，當 current thread 執行 monitorenter 指令取得 lock obejct 失敗時，ICCU 輸出訊號 ICCU_monEnter_fail = 1 將會觸發 Thread Controller 的內部信號 timeout=1，此時必須等到 Thread Controller 的狀態為 Check_Timeout 並且 timeout 等於 1 才可以把 running_thread_state 的值改為 01(表示 waiting state)。下個狀態進入 ContextSwitch，running_thread_state 內容此時傳給 prev_thread_state，接著再進入 Backup_previous_TCBinfo 狀態。此時如同前一個情況，把 prev_thread_state 的數值寫到 Thread Control Block 裡面 previous thread 對應的 entry。
- 當 current thread 執行 monitorexit 指令，要將 lock object 權限轉移到別的 JAIP 底下某個 thread 的時候，DCC2JAIP_info 的值代表下一個 lock 擁有者的 thread ID，首先該 JAIP 的 ICCU 的 ICCU_monExit_nxtOwner 訊號等於 1，表示此 Thread Manager Unit 底下有某個 waiting thread 將被改回 ready state，DCC2JAIP_info 的值被傳到 Thread Control Block，查詢對應的 entry，最後修改 thread state 為 01(表示 ready state)。
- 當 current thread 終止執行時，Thread Controller 的 thread_terminate 等於 1(圖 42)並且讀取 running_thread_id 的值到 Thread Controller 查詢對應的 entry (圖 47)，此時 thread state 改成 00 即可把這個 entry 回復到閒置的狀態。

## 4.3. Thread Queue

當 Thread Manager Unit 維護 2 個以上的 thread 執行資訊時，必須建立一個機制能對

所有 threads 做排班，以決定每個 thread 取得 Bytecode Execution Engine 執行權的先後順序，在此我們採用的排班演算法是循環分時排程(Round-Robin scheduling)，因此執行時間被切成一個個時間片段(Time slice)，使每個 thread 能被分配到相同時間公平地輪流執行。為此我們在 Thread Manager Unit 中設計了 Thread Queue 元件，這個元件中存在著一組由 registers 所組成的環狀佇列(Circular Queue)(如圖 48)來輔助排班機制，並且維護兩個指標 Ready pointer(RP)與 Tail pointer(TP)。Ready pointer 指向 Circular Queue 中某個欄位內容，代表下個可取得 Bytecode Execution Engine 執行權的 ready thread 的 ID number；而 Ready pointer 所指的前一欄位內容則是 current thread 的 ID number；Tail Pointer 同樣會指向 Circular Queue 中某個欄位內容，這表示當產生 new thread 時需要存入 new thread ID 的欄位、或是當執行 context-switching 時需要存入 previous thread ID 的欄位。

先前的 Thread Manager Unit 架構[1]並未考慮 thread state 的情況，因為在多核心 JAIP 環境下每個 JAIP 僅有一個 thread 執行指令，所以不管這個 thread 是在 ready state 或是 waiting state，Thread Manager Unit 得不會有任何行為，Thread Queue 也不會被啟動。在我們的設計下若要把同步機制的電路完整地加入到 Thread Manager Unit，使得多核心 JAIP 處理器下每個 JAIP 能維護 2 個以上的 threads，則 Thread Manager Unit 必須判斷目前維護的 threads 相關資料之中那些是 waiting thread，並且避免 waiting thread 從 Thread Queue 再次被載入到 Bytecode Execution Engine 執行指令。其中一種做法是新增一個 waiting thread queue 存放所有 waiting threads 的 ID number，但是此作法不適用於硬體設計上，因為隨著 waiting thread 數量增加，儲存與搜尋 waiting thread ID 的電路資源勢必會隨著增加。

因此我們提出的修改如下：Thread Queue 只用來儲存 ready thread 的 ID number，當 Thread Controller 狀態為 ContextSwitch 的時候，如果 current thread 在 waiting state，則 Thread Controller 不會將 current thread ID 寫入到 Thread Queue 中 TP 指向的位置。如果這個 JAIP 的某個 thread 從 waiting state 轉換到 ready state，則由 Thread Controller 修改 Thread Control Block 內對應的 thread state，以及把這個 thread ID 寫到 Thread Queue 之

中 TP 指向的位置。

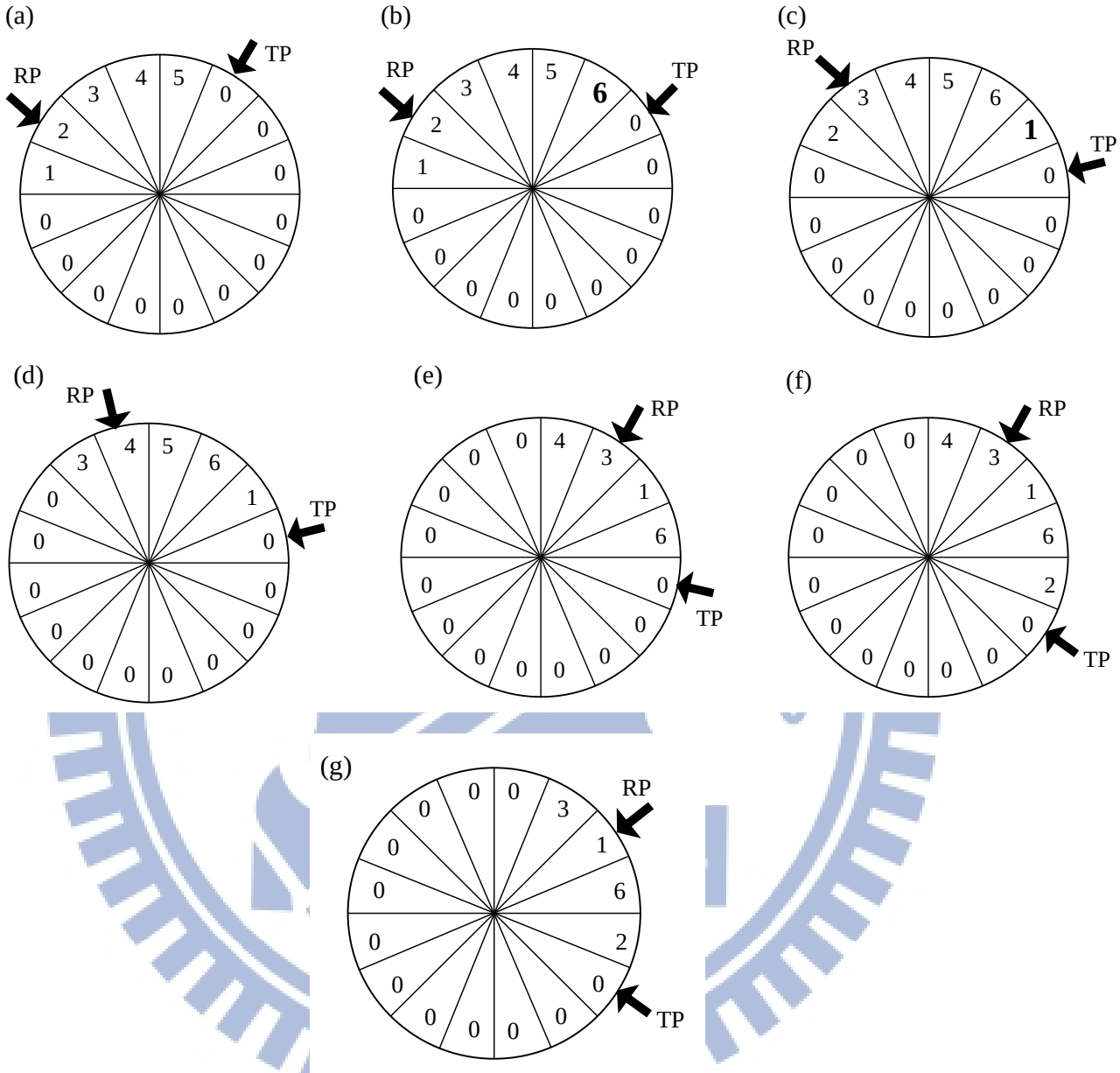



圖 48 Thread Queue 操作示意圖

Thread Queue 的操作以圖 48 為例，當 new thread 被產生時，Thread Controller 的 new_thread_flag=1 (圖 42)並且直接把 register new_thread_id 的數值分配給 new thread，在此假設 new thread ID 為 6。並且將 new thread ID 寫到 Thread Queue 的 Tail pointer 所指向的欄位，如圖 48(a)與(b)，最後累加 Tail pointer 的值往前指一個位置。

如果 current thread 執行時間到達一個片段時，則 Thread Controller 狀態進入 ContextSwitch 切換 Bytecode Execution Engine 讀取 current thread 的執行資訊，在此假設 ready thread ID 為 2、current thread ID 為 1，且 current thread 在 ready state，此時把 current thread ID 寫入到 Tail pointer 所指向的位置，Tail pointer 與 Ready pointer 都分別累加一往前指一個位置，如圖 48(b)與(c)，此時代表在這次 context-switching 執行完成後 current thread ID 為 2，而下次執行 context-switching 時，ID number 為 3 的 ready thread 即將可以取得 Bytecode Execution Engine 執行權。

如果 current thread 執行時間到達一個片段並且 current thread 在 waiting state，則 Thread Controller 狀態進入 ContextSwitch，假設 ready thread ID 為 3、current thread ID 為 2，根據 section 3-3-2 的說明，Thread Controller 的內部 register running_thread_state 把值傳給 prev_thread_state，此時 prev_thread_state=10 表示 current thread 在 waiting state，Thread Queue 不需要將 current thread ID 寫到 TP(Tail Pointer)指向的位置、不需要累加 TP 的值，指向 Circular Queue 的下個位置，如圖 48(c)與(d)。當 Thread Controller 的狀態為 ContexstSwitch 會累加 RP(Ready Pointer)的值，表示 current thread 的 ID 為 3、下個 ready thread 的 ID 為 4。

考慮目前 ready thread ID 為 4、current thread ID 為 3，假設其他 JAIP 的 current thread 釋放 lock object(e.g. 執行 monitorexit 指令)並且把 lock 權限轉移到這個 JAIP 之中 ID number 為 2 的 waiting thread。此時 DCC2JAIP_info 的值代表下一個 lock 擁有者的 threadID，首先 ICCU 的 ICCU_monExit_nxtOwner=1，Thread Controller 把 DCC2JAIP_info 的值寫入 Thread Queue 之中 TP 指向的位置，最後累加 TP 的值往前指一個位置，如圖 48(d)與(e)。

當 current thread 以終止其執行時，如圖 48(e)與(f)，在此假設執行終止的 thread 其 ID number 為 4。由於此 thread 已經終止，Thread Controller 的 thread_terminate=1 並且其狀態進入 ContextSwitch，此時不需要將 previous thread 寫入 Thread Queue，而是直接累加 Ready pointer 使它往前指一個位置。使 ID number 等於 3 的 ready thread 能直接進入 Bytecode Execution Engine 執行指令。

# 第五章 實驗結果

## 5.1. 實驗環境

本論文所提出之 JAIP 處理器包含 ICCU 與 Thread Manager Unit 實作在 Xilinx ML-605 Evaluation Platform(XC6VLX240T)，此開發板的 FPGA 模組有充足的電路資源使我們可以建立 4 個 JAIP 處理器與 1 個 Data Coherence Controller IP 在這個平台上，我們採用 MicroBlaze soft IP core 作為系統上的 RISC-core。系統頻率設置為 83.3MHz。Data Coherence Controller、JAIP 的 RTL model 皆以 VHDL 語言實作並以 Xilinx Synthesis Tool 合成。此外透過 Xilinx ISE Design Suite 13.4 提供的 Chipscope Pro Analyzer 與相關 Debugging IP 驗證我們實作的電路。軟體部分我們使用 Xilinx SDK 來建立 Board Support Package (BSP)，這是一個輕巧、低階且獨立無須作業系統支援的軟體層，裡面包括了基本的操控各種硬體裝置(例如 UART 或 Memory controller)之 library 以及 C standard library。RISC-core 的 runtime 環境(JAIP 初始化、class parser 或各種 interrupt service routine 等軟體)是建立在此 BSP 之上。

| **Device: vertex-6 (XC6VLX240T)** | |
|---|---|
| Number of LUTs | 12580 |
| Number of Flip-flops | 5912 |
| Number of 2K BRAM | 34 |
| Maxinum frequency | 83.6MHz |

(a)

| **Device: vertex-6 (XC6VLX240T)** | |
|---|---|
| Number of LUTs | 663 |
| Number of Flip-flops | 449 |
| Number of 2K BRAM | 1 |
| Maxinum frequency | 83.6MHz |

(b)

表 3 Resource Utilization (a)proposed JAIP (b) proposed DCC

表 3 分別說明 JAIP 與 Data Coherence Controller 在 ML605 平台的電路資源使用，包含 Flip-Flop 及 LUT 數量。由於電路資源使用量在不同版本合成工具和不同電路之整合上會有不同的數據，合成工具會根據特定 FPGA 平台的電路狀況，在 data path 和 resource 使用量之間做整體優化。

| Device: vertex-6 (XC6VLX240T) | |
|---|---|
| Number of LUTs | 9730 |
| Number of Flip-flops | 4654 |
| Number of 2K BRAM | 26 |

(a)

| Device: vertex-6 (XC6VLX240T) | |
|---|---|
| Number of LUTs | 663 |
| Number of Flip-flops | 449 |
| Number of 2K BRAM | 1 |

(b)

| Device: vertex-6 (XC6VLX240T) | |
|---|---|
| Number of LUTs | 16138 |
| Number of Flip-flops | 11176 |
| Number of 2K BRAM | 32 |

(c)

| Device: vertex-6 (XC6VLX240T) | |
|---|---|
| Number of LUTs | 1193 |
| Number of Flip-flops | 478 |
| Number of 2K BRAM | 0 |

(d)

表 4 Resource Utilization 比較表

(a) JAIP (without Arrycopy Accelerator) (b) Data Coherence Controller

(c) JAIP from [1] (d) Data Coherence Controller from [1]

表 4 比較本篇論文與[1]的電路資源參考數據，包含 LUTs 及 flip flops。表 3(a)與表 4(a)中 JAIP 資源使用數量的差別在於：(1)因為[1]的設計並未包含 Arraycopy 加速電路[2]，為了盡量達到相同的條件因此在表 4(a)我們拿掉了 Arraycopy 加速電路並且重新合成 JAIP (2)相較於[1]，我們提出的設計以硬體支援以下功能，例如 heap pointer controller、新增 object、新增 array、以及 Hardware Native Interface。

結果顯示，我們修改的 Thread Manager Unit 相對於[1]的設計，多使用一個 on-chip block RAM，但是可以減少使用 6522 個 flip flop，以及 6408 個 LUT。先前[1]的設計中提到 Thread Control Block 最多支援 16 threads 並且多儲存一個 thread 的執行資訊就必須多使用 256 bits flip flop，而在我們的設計下最多可支援 64 threads 並且所有 threads 的執行資訊皆放在 on-chip block RAM。

Data Coherence Controller 電路資源比較結果，我們修改的設計下多使用一個 on-chip block RAM，但是相較於[1]的設計可以減少使用 29-bits flip flop，以及 530-bits LUT。由於[1]的 多核心 JAIP 處理器架構下並未啟動 Thread Manager Unit，所以 Data Coherence Controller 的 synchronized manager 僅用 4 registers 儲存 lock object，以及用 synchronized

manager 內部的組合電路判斷 thread 同步問題，因此所佔的資源數量較大，如果要暫存比較多的 lock object，勢必會增加 Flip-Flop 與 LUTs 的數量。相較之下我們提出的 Lock Object Accessing Controller 最多可容納 128 waiting threads 與 lock object 參考位置，且使用少許電路資源執行 lock object 查詢與維護的動作。

## 5.2. Benchmark 分析

為了比較 JAIP 的 temporal multithreading 機制與其他平台的效能，我們另外建立 Sun’s CVM 平台並且執行相同測試程式。CVM 為嵌入式的 Java VM 並且支援 Just-In-Time (JIT) compilation，在此 CVM 實作在 Xilinx ML-405 的 PowerPC 處理器上，為了使 CVM 能支援 multithreaded Java programs 執行，底層作業系統使用嵌入式 Linux Kernel 2.6.38，工作頻率同為 83.3 MHz。

在本小節我們用 JemBench Suite 的 parallel benchmark programs 來測試 JAIP 的 multi-threading 機制的執行效能。JemBench 是一個開放原始碼的嵌入式平台 Java benchmark，此 benchmark 包含了絕大部分的功能測試並且可區分為 computational kernel benchmark(Bubble、Sieve)、control application benchmark(Klf、Lift、udpip)以及 parallel benchmark(Dummy Test、Matrix Multiplication、NQueens)等。除了 JemBench 內含的三組 parallel benchmark programs，我們還將 CaffeineMark 中的 Logic 測試程式抽出並仿照 JemBench 的寫法改寫為 Multi-Logic benchmark 來進行測試。

本小節分成四階段，首先說明單一核心 JAIP 處理器在 temporal multi-threading 環境下程式執行效能，並且與 CVM、CVM-JIT 的執行效能做出比較；接著分析多核心 JAIP 處理器的程式執行效能，當我們多使用一個 JAIP 處理器，或者每個 JAIP 處理器多容納一個 thread，benchmark program 執行結果的差異；最後分析 Thread Manager Unit 與 Data Coherence Controller 內部執行 multi-threading 相關功能時所需的平均時間、最大與最小時間。

### 5.2.1. Temporal Multithreading 效能分析

此階段我們在 benchmark program 加入多個 threads，分別執行在單一核心 JAIP 處理器與 CVM 上，請注意在 CVM 環境下，我們每次皆啟用 JIT compilation 協助執行 Java benchmark program。此階段 benchmark program 將 thread 的數量遞增到 16 組並且觀察效能的變化。此次效能比較中 Thread Manager Unit 用於判斷執行緒切換的 time slice 為 20 microseconds。

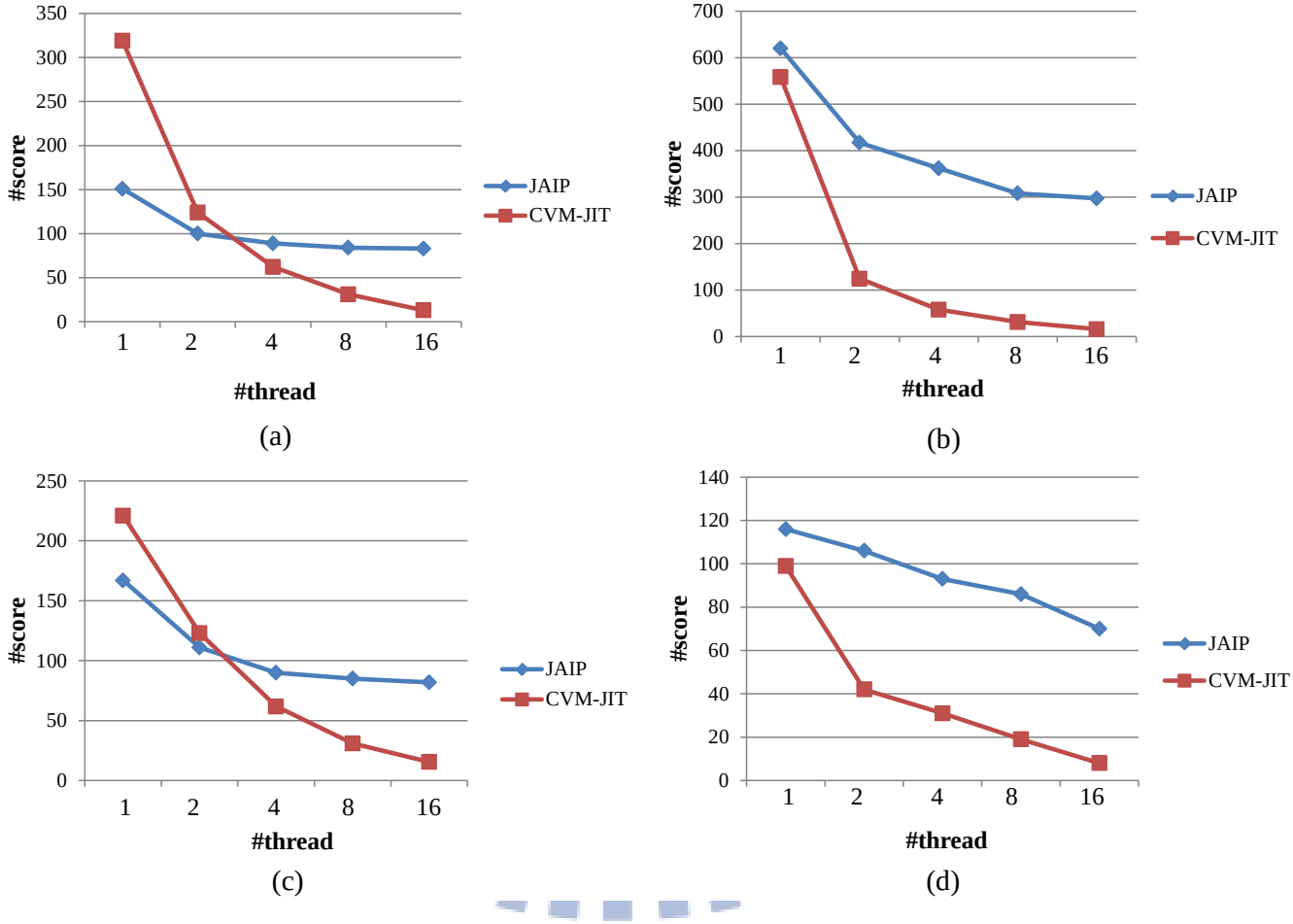



圖 49 JAIP 與 CVM 在 temporal multithreading 機制下的效能比較結果
(a) Dummy Test (b) Multi-Logic (c) Matrix Multiplication (d) N-Queens

圖 49 為我們使用的 4 個 benchmark 之分數比較圖，這些 benchmark 皆是測試 Java 程式在長時間重複動作操作下，系統效能的表現。在 single-thread 測試條件下 CVM 執行 Dummy Test 與 Matrix Multiplicaiton 的分數皆大於 JAIP，然而隨著 benchmark programs

中 thread 的數量遞增，CVM 與 JAIP 執行這些 benchmark programs 的分數逐漸下降，當 thread 的數量等於 4，JAIP 在執行分數上已經大於 CVM；而 NQueens 與 Multi-Logic 不論在 single-thread 或者 multi-thread 測試條件下，JAIP 執行的分數始終超過 CVM，這表示其 method bytecode 尚有可以優化的空間。另外，當 benchmark programs 的 thread 數量增加時，造成執行時間增加的原因不僅只有 context-switching overhead 另外還有 synchronization operations，我們在後面會分別說明 JAIP 執行跟 multi-threading 相關功能時所花費的時間。

### 5.2.2. Multicore multithreading 效能分析

此階段我們在 4 核心 JAIP 處理器上執行 Jembench parallel benchmark 與 Multi-Logic benchmark，並且加入多個 threads 檢查效能變化。表 5 與表 6 分別說明 4 核心架構下，在每個 JAIP 上執行一個 thread 與多個 threads 輪替切換執行時，對於 benchmark program 執行分數的影響。此次效能比較 JAIP 用來判斷執行緒切換的 time slice 為 20 microseconds。

結果顯示，在表 5 內可以觀察到當 JAIP 處理器個數增加時每個 benchmark programs 執行分數也會明顯地增加。由於此階段測試環境不再只是由單一 JAIP 處理器的 Temporal multithreading 機制來執行，在此多核心的運作環境下可達到平行執行的狀態，因此當 JAIP 數量增加時效能也會向上提升。

| Benchmark (score) / #thread | Dummy Test | Multi-logic | Matrix Multiplication | N-Queens |
|---|---|---|---|---|
| 1 | 151 | 620 | 167 | 116 |
| 2 | 298 | 1195 | 240 | 225 |
| 3 | 374 | 1455 | 395 | 330 |
| 4 | 491 | 1722 | 498 | 428 |

表 5 多核心 JAIP 處理器執行效能 (未啟用 temporal multithreading)

| Benchmark (score) / #thread | Dummy Test | Multi-logic | Matrix Multiplication | N-Queens |
|---|---|---|---|---|
| 5 | 410 | 1665 | 425 | 311 |
| 6 | 373 | 1416 | 412 | 251 |
| 8 | 362 | 1411 | 399 | 262 |
| 12 | 359 | 1260 | 366 | 212 |
| 16 | 340 | 1105 | 327 | 195 |

表 6 多核心 JAIP 處理器執行效能 (已啟用 temporal multithreading)

表 6 說明在 4 核心 JAIP 處理器環境下，如果每個 JAIP 的 temporal multithreading 機制皆被啟用，對於 benchmark programs 執行分數的影響。當 benchmark program 的 thread 數量從 4 個遞增到 16 個，此時每個 JAIP 內每個 thread 需要被輪替切換執行，因此效能開始下降。如同之前敘述，造成 benchmark 執行時間增加的原因不僅只有 context-switching，其可能原因有：

(1)在此使用的 benchmark program 底下，當任何一個 thread 進入 synchronized method 或者 synchronized statement 的時候，JAIP 會觸發 Data Coherence Controller 處理 thread 同步問題，如同 section3-2-3 所敘述，多個 threads 在請求取得同個 lock object 的時候，Mutex Controller 與 Lock Object Accessing Controller 會依序處理這些來自不同 JAIP 的請求，因此會造成額外的等待時間。

(2)為了使在 DDR memory 中的 Object Heap Space 的資料保持一致性，Heap Cache Controller 在此採用 write-through 機制，如果任何指令會修改 JAIP 處理器上的 Object Heap Cache 時(e.g. putfield, putstatic)，則更新完 Object Heap Cache 之後一律將修改的資料寫入 DDR memory。因為存取 DDR memory 必須透過 system bus，所以每個 JAIP 處理器必須等到 system bus 為空閒狀態時才可以開始使用，當處理器數量越多對程式效能造成的影響也越大。

(3)Benchmark program 的程式寫法會造成不同效能測試結果，例如執行時間的計算與同步程式的寫法。以 Jembench Suite 的 N-Queens benchmark program 為例，此程式使

用 2 個 synchronized methods，由於同個 class file 內所有 synchronized method 都必須參考同一個 lock object，並且這 2 個 synchronized methods 分別保護不同的共享變數，在這種情況下我們修改 N-Queens 程式，藉由使用 2 個 lock objects 實作這兩個同步區塊來達到較好的執行效能，並且能維持程式執行的正確性，表 7 為改寫後的 N-Queens benchmark program 在 4 核心 JAIP 處理器環境上執行分數，當 threads 的數量到 5 個以上，每個 JAIP 處理器啟動 temporal multithreading 機制後，我們可以觀察到測試程式的寫法對執行時間造成的影響。

| Benchmark (score) / #thread | N-Queens |
|---|---|
| 1 | 116 |
| 2 | 229 |
| 3 | 337 |
| 4 | 439 |
| 5 | 382 |
| 6 | 337 |
| 8 | 332 |
| 12 | 272 |
| 16 | 241 |

表 7 N-Queens 程式執行效能 (使用 2 個 lock objects)

### 5.2.3. Thread Manager Unit 效能分析

此階段我們使用 Jembench Suite 的 parallel benchmark programs 與 Multi-logic 這 4 個 benchmark programs 測試 context-switching 執行時間，以及不同 time slice 參數對於 JAIP temporal multithreading 機制所造成的影響。

根據 section 3-3 的敘述，當 Thread Controller 執行 context-switching 時會使用 1 clock 切換 ready thread 與 current thread 的 special-purpose registers 例如 stack pointer，local variable pointer，Java program counter 與 current thread ID 等資料；同時切換 Ping-pong Java

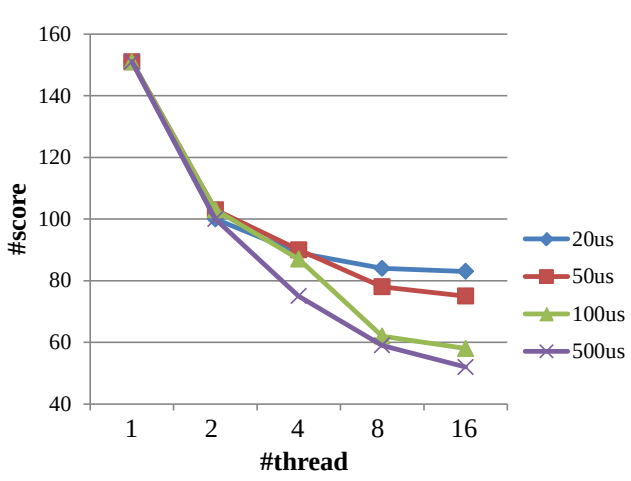
Stack 底下的 2 組 interleaving on-chip memories；但是切換 ready thread 與 current thread 的 current method 最快需要 5 個 clock，任意 2 個 methods 切換必須先檢查下個被呼叫的 method image 是否還保留在 Method Area Circular buffer 之中(如圖 18)。表 8 說明在 4 核心 JAIP 處理器環境下執行所有 parallel benchmark programs 時，JAIP 處理器的 context-switching 平均執行時間。當 thread 數量達到 8 個以上，每個 JAIP 處理器的 Thread Manager Unit 開始排序不同 threads 使它們輪替執行，由此表可見 context-switching 平均執行時間都趨近於 5，這表示 context-switching 並非執行這些 benchmark programs 時主要的效能瓶頸。

| #thread / time | Dummy Test | | | Matrix Multiplication | | |
|---|---|---|---|---|---|---|
| | 8 | 12 | 16 | 8 | 12 | 16 |
| #clock | 700330 | 698554 | 745249 | 628350 | 693833 | 777639 |
| #count | 140029 | 139674 | 149013 | 125633 | 138730 | 155490 |
| Avg. Clock | 5.0013 | 5.0013 | 5.0012 | 5.0015 | 5.0013 | 5.00122 |

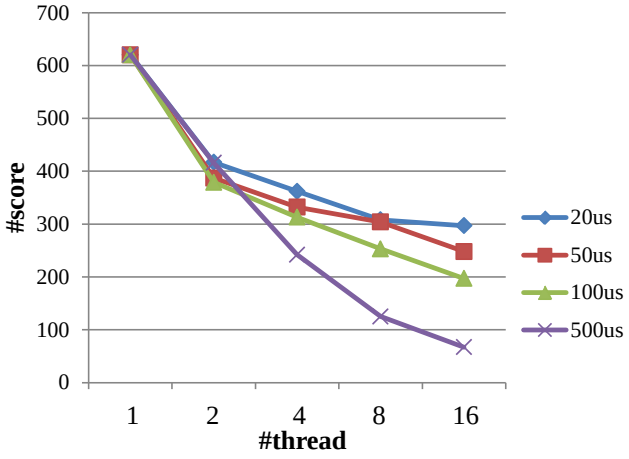
| #thread / time | Multi-logic | | | N-Queens | | |
|---|---|---|---|---|---|---|
| | 8 | 12 | 16 | 8 | 12 | 16 |
| #clock | 721969 | 800786 | 924149 | 251066 | 256930 | 320619 |
| #count | 144356 | 160121 | 184794 | 50195 | 51368 | 64105 |
| Avg. Clock | 5.00131 | 5.00113 | 5.00097 | 5.00189 | 5.001751 | 5.001466 |

表 8 不同 benchmark programs 下 context-switching 執行時間

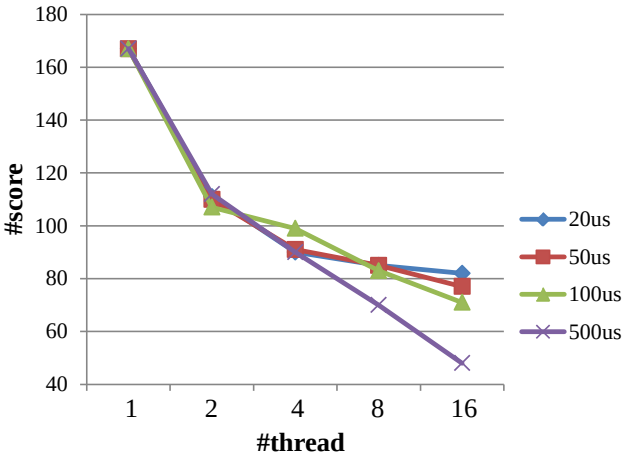
在單一核心 JAIP 環境下我們分別使用 20、50、100 與 500 microseconds 四種不同 time slice 參數，測試 Thread Manager Unit 的執行效能，圖 50 顯示其執行結果，當 time slice 為 20 microseconds 時大部分 benchmark programs 執行分數可以達到最高，因此在本小節 Thread Manager Unit 皆以 20 microseconds 分配個別 thread 相同執行時間並且測試執行結果。另外當 time slice 數值增加時執行分數反而下降，其中一個原因為 benchmark program 控制每個 thread 執行主要程式功能的方式。



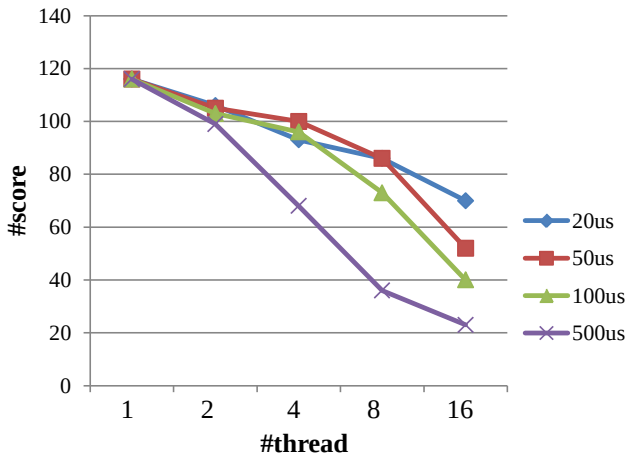

(a)




(b)




(c)




(d)

圖 50 不同 time slice 參數對於 benchmark program 執行分數的影響
(a)Dummy Test (b) Multi-Logic (c) Matrix Multiplication (d) N-Queens

本章節使用的 benchmark programs 皆由 main thread 紀錄執行時間，當測試程式開始執行時首先 main thread 進入 measure method (圖 51(a))，依照設定產生特定數量的 threads，接著每回合 main thread 會呼叫 Util.getTimeMillis()讀取 JAIP 內部 timer 數值並且執行 executeParallel。圖 51(a)的 r.run()與圖 51(b)的 work.run()代表每個 thread 開始執行 benchmark 主要程式功能，所有 threads (class Worker)皆以一個 object 欄位 finished 控制每個 thread 執行 work.run()。當 main thread 進入 executeParallel 之後先把每個 thread 的變數 finished 改成 false，使得其他 threads 可以開始執行 work.run()。接著 main thread 執行完 r.run()之後進入迴圈檢查所有 threads 的變數 finished 其值是否被改成 true，如果

所有 threads 完成執行主程式功能，則 main thread 才會離開迴圈並且從 executeParallel 返回，最後再由 main thread 呼叫 Util.getTimeMillis()讀取 JAIP 內部 timer 數值並計算執行時間與分數。每回合執行效能計算時，假設 main thread 第一次讀取 JAIP 內部 timer 後尚未將其他 thread 的變數 finished 改成 false，就先被切換到下一個 thread 執行，此時其他 threads 會卡在 run method(圖 51(b))的 while-loop 內無法執行 work.run()主要程式功能，這種情況下當 time slice 越大反而造成 performance 越低。

```
public int measure() {
        …
        start = Util.getTimeMillis();
        for (int i=0; i<cnt; ++i){
                executeParallel(r);
        }
        end = Util.getTimeMillis();;
        …
}

public void executeParallel(Runnable r) {
        for (int i=0; i<cpus-1; ++i) {
                runner[i].setExecute(r);
                runner[i].finished = false;
        }
        r.run();
        boolean allFinished;
        do {
                allFinished = true;
                for (int i=0; i<cpus-1; ++i) {
                        allFinished &=
                          runner[i].finished;
                }
        }
        while (!allFinished);
}
```

(a)

```
class Worker extends Thread {
        volatile boolean finished;
        Runnable work;
        …
        void setExecute(Runnable r) {
                work = r;
        }
        …
        public void run() {
                while (1) {
                        if (!finished) {
                                work.run();
                                finished = true;
                        }
                        else {
                                if (requestStop)
                                { break;}
                        }
                }
        }
        …
}
```

(b)

圖 51 Jembench Suite 的 parallel benchmark program 計算執行時間之示意圖

## 5.2.4. Data Coherence Controller 效能分析

根據 section 3-2 敘述，Data Coherence Controller 主要實作 Thread Assignment Controller、Lock Object Accessing Controller、Cache Coherence Controller，此階段我們在

4 核心 JAIP 處理器環境下分析 Data Coherence Controller 執行前兩項功能所需要的時間，並且同樣以 Jembench Suite 的 parallel benchmark programs 與 Multi-logic 作為效能測試程式。最後一項 Cache Coherence Controller 效能分析不在本論文範圍內。

| Time \ #thread | 4 | 8 | 12 | 16 |
|---|---|---|---|---|
| #clock | 33 | 77 | 121 | 165 |
| #count | 3 | 7 | 11 | 15 |
| avg. clock | 11 | 11 | 11 | 11 |

表 9 產生 new thread 所需的執行時間

表 9 說明產生 new thread 所需的執行時間，在此執行時間的計算是從 Hardware Native Interface 觸發 ICCU 開始，直到 Data Coherence Controller 的 DCC2JAIP_response_msg 回傳有效值為止。由於我們執行的 benchmark programs 一律經由 main thread 呼叫 Thread.start()產生 new thread，於是 Data Coherence Controller 在每個 benchmark programs 之下平均執行時間皆為 11 clocks，所以產生 new thread 在 benchmark programs 總執行時間內只佔一小部分。

| #clock \ #thread | Dummy Test | | | | |
|---|---|---|---|---|---|
| | 4 | 6 | 8 | 12 | 16 |
| Avg. | 22.5 | 22.9 | 23.8 | 27.4 | 29.1 |
| Worst-case | 34 | 85 | 86 | 92 | 104 |
| Best-case | 9 | 9 | 9 | 9 | 9 |

| #clock \ #thread | Matrix Multiplication | | | | |
|---|---|---|---|---|---|
| | 4 | 6 | 8 | 12 | 16 |
| Avg. | 23.1 | 25.4 | 25.4 | 28.9 | 28.9 |
| Worst-case | 43 | 89 | 108 | 108 | 110 |
| Best-case | 9 | 9 | 9 | 9 | 9 |

| #clock \ #thread | Multi-Logic | | | | |
|---|---|---|---|---|---|
| | 4 | 6 | 8 | 12 | 16 |
| Avg. | 22.2 | 23.89 | 24.6 | 29.0 | 29.3 |
| Worst-case | 35 | 80 | 109 | 110 | 102 |
| Best-case | 9 | 9 | 9 | 9 | 9 |

| #clock \ #thread | NQueens | | | | |
|---|---|---|---|---|---|
| | 4 | 6 | 8 | 12 | 16 |
| Avg. | 22.1 | 24.9 | 24.5 | 24.8 | 24.5 |
| Worst-case | 42 | 90 | 104 | 94 | 102 |
| Best-case | 9 | 9 | 9 | 9 | 9 |

表 10 current thread 取得 lock object 所需的執行時間

| #thread / #clock | Dummy Test | | | | |
|---|---|---|---|---|---|
| | 4 | 6 | 8 | 12 | 16 |
| Avg. | 21.1 | 21.4 | 21.4 | 27.9 | 29.1 |
| Worst-case | 38 | 70 | 77 | 78 | 98 |
| Best-case | 10 | 10 | 10 | 10 | 10 |

| #thread / #clock | Matrix Multiplication | | | | |
|---|---|---|---|---|---|
| | 4 | 6 | 8 | 12 | 16 |
| Avg. | 20.2 | 21.6 | 21.8 | 28.4 | 28.7 |
| Worst-case | 43 | 76 | 85 | 92 | 98 |
| Best-case | 10 | 10 | 10 | 10 | 10 |

| #thread / #clock | Multi-Logic | | | | |
|---|---|---|---|---|---|
| | 4 | 6 | 8 | 12 | 16 |
| Avg. | 20.5 | 21.4 | 25.9 | 28.6 | 29.1 |
| Worst-case | 40 | 68 | 89 | 88 | 94 |
| Best-case | 10 | 10 | 10 | 10 | 10 |

| #thread / #clock | NQueens | | | | |
|---|---|---|---|---|---|
| | 4 | 6 | 8 | 12 | 16 |
| Avg. | 21.9 | 23.0 | 23.5 | 23.9 | 24.2 |
| Worst-case | 38 | 85 | 95 | 95 | 98 |
| Best-case | 10 | 10 | 10 | 10 | 10 |

表 11 current thread 釋放 lock object 所需的執行時間

表 10 與表 11 分別說明 current thread 取得或者釋放 lock object 時，Lock Object Accessing Controller 的相關執行時間，在此執行時間的計算是從 Decode Stage 觸發 ICCU 開始，直到 Data Coherence Controller 的 DCC2JAIP_response_msg 回傳有效值為止。

從這兩張表我們可觀察到：取得 lock object 所需的最小時間為 9 clocks，最大時間為 110 clocks；釋放 lock object 所需的最小時間為 10 clocks，最大時間為 98 clocks。取得 lock object 所需的平均時間大於或等於釋放 lock object 的平均時間，這是因為當 current thread 取得 lock object 時，必須在 Lock Object Accessing Controller 內搜尋 Waiting Thread Table 與 Lock Object Table，根據 section 3-2-3 敘述，搜尋時間取決於 lock object 在 Waiting Thread Table 內對應的 linked list 長度，以及 benchmark programs 內總共使用的 lock object 數量(如圖 38 的範例說明)；然而當 current thread 釋放 lock object 時，Lock Object Accessing Controller 只需要讀取 lock object 在 Waiting Thread Table 內對應 linked list 的前 2 個節點(如圖 40 的範例說明)即可完成工作。另外，任意 2 個來自不同 JAIP 處理器的 current threads 同時發出取得 lock object 的請求，由於 Data Coherence Controller 的 Mutex Controller 會依序把這些請求傳到 Lock Object Accessing Controller，因此取得/釋放 lock object 的執行時間也包含等待 Mutex Controller 的處理每項請求的時間。

# 第六章 結論與未來展望

本論文我們以先前 JAIP 為基礎提出 Multicore multithreading Java Processor 架構。每個 JAIP 處理器的可容納多個 threads、達到極小的 context-switching overhead、time slice 數值可以壓縮到 20 milliseconds，並且大幅減少電路資源的使用；在多核心 JAIP 處理器方面，我們提出 Inter-Core Communication Unit 與 Data Coherence Controller 新的設計，用以支援 thread 的分配、lock object 與 waiting thread 的維護，修改每個處理器的 Thread Manager Unit 以支援 waiting thread 管理與排班。實驗結果顯示我們提出的架構可大幅縮減 Java 處理器電路資源使用並且有效支援更多 threads。

未來設計方向有 3 點：(1)使用不同 thread scheduling 方法實作成電路例如 priority queue，當 Java 應用程式內每個 thread 的指令數量或重要性不一致時，能夠設定每個 thread 的 priority 並且作為 scheduling 電路模組的參數 (2)藉由 JAIP 的 Hardware Native Interface 支援更多 Java native methods (3)多核心 JAIP 處理器下 Object Heap Cache 與存取效率的改進，目前 JAIP 處理器的 Heap Cache Controller 採用 Write-through 機制，由於每次執行 Object Heap Cache 寫入的指令時皆需要傳資料到 DDR memory，因此需要設計 Write-back 或是更進階的 flush 機制減少 memory accessing overhead。

# 參考文獻


[1] Hung-Cheng Su, “Design of Multithreading Architecture for a Java Processor, ” Master thesis, NCTU, 2013.

[2] Chia-Che Hsu, “Performance Evaluation and Optimization of String Manipulation on a Java Processor, ” Master thesis, NCTU, 2013.

[3] 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.

[4] Kent, K. B., & Serra, M. “Hardware/software co-design of a Java virtual machine.” In *Rapid System Prototyping, 2000. RSP 2000. Proceedings. 11th International Workshop on* (pp. 66-71). IEEE, 2000

[5] T. Lindholm and F. Yellin, The Java Virtual Machine Specification, 2nd Ed., Addison-Wesley, 1999.B. R. Montague, “JN: OS for an Embedded Java Network Computer,” *IEEE Micro*, 17, 3, 1997, pp. 54-60.

[6] Venners, Bill. *Inside the Java virtual machine*. McGraw-Hill, Inc., 1996. ch.5 ch.6 ch.7 ch.8.

[7] Krall, Andreas. "Efficient Java VM just-in-time compilation." *Parallel Architectures and Compilation Techniques, International Conference on*. IEEE, Oct. 1998, pp. 205-212

[8] Microsystem, S. U. N. "Connected Device Configuration (CDC) of J2ME; JSR 36, JSR 218." *URL: http://java. sun. com/products/cdc/index. jsp* (2006).

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

[10] 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.

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

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

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

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

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

[16] aJile Inc, “aj-102 technical reference manual v2.4.”, 2009

[17] 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

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

[19] Kreuzinger, Jochen, et al. "Real-time event-handling and scheduling on a multithreaded Java microcontroller." *Microprocessors and Microsystems* 27.1 (2003): 19-31.

[20] Uhrig, Sascha, and Jörg Wiese. "jamuth: an IP processor core for embedded Java real-time systems." *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*. ACM, 2007.

[21] Sun Microsystems Inc,”picoJava-II Programer’s Reference Guide”, Mar. 1999.

[22] Patel, Mukesh K., Udaykumar R. Raval, and Harihar J. Vyas. "Java hardware accelerator using thread manager." U.S. Patent No. 6,826,749. 30 Nov. 2004.

[23] M. Schoeberl, "Design rationale of a processor architecture for predictable real-time execution of Java programs." *Proc. of International Conference On Real-Time And Embedded Computing Systems And Applications*. 2004.

[24] Pitter, Christof, and Martin Schoeberl. "A real-time Java chip-multiprocessor."*ACM Transactions on Embedded Computing Systems (TECS)* 10.1 (2010): 9.

[25] Stoif, Christian, et al. "Hardware synchronization for embedded multi-core

processors." *Circuits and Systems (ISCAS), 2011 IEEE International Symposium on*. IEEE, 2011.

[26] C.-J. Tsai, H.-W. Kuo, Z. Lin, Z.-J. Guo, J.-F. Wang, "A Java Processor IP Design for Embedded SoC," *ACM Transactions on Embedded Computing Systems, accepted on January 4, 2014*. An electronic manuscript of the paper is available from: http://www.cs.nctu.edu.tw/~cjtsai/research/jaip.

[27] Ko, Hou-Jen, and Chun-Jen Tsai. "A double-issue Java processor design for embedded applications." *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*. IEEE, 2007.

[28] Schoeberl, Martin, and Juan Ricardo Rios. "Safety-critical Java on a Java processor." *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*. ACM, 2012.

[29] Gruian, Flavius, and Martin Schoeberl. "Hardware support for CSP on a Java chip multiprocessor." *Microprocessors and Microsystems* 37.4 (2013): 472-481.