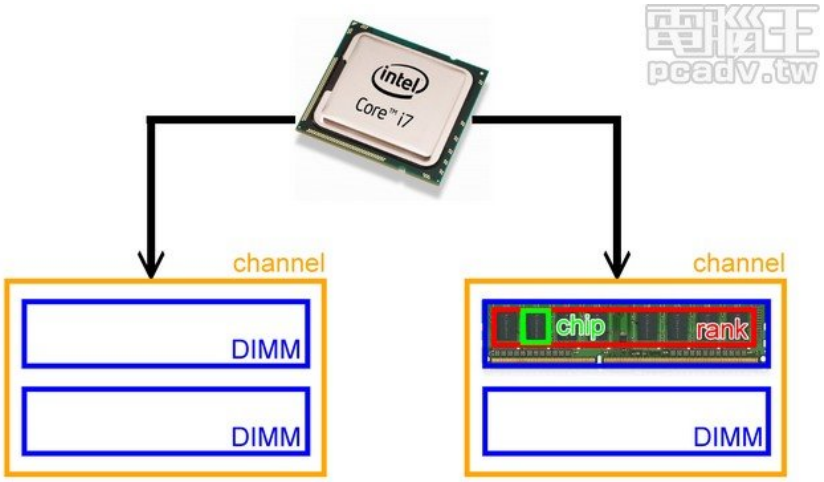
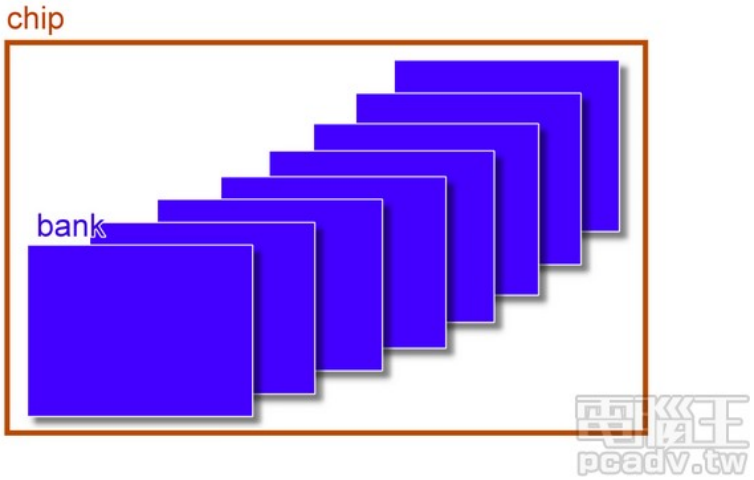
DRAMSim2 使用教程

陳庚佑

gengyouchen@gmail.com

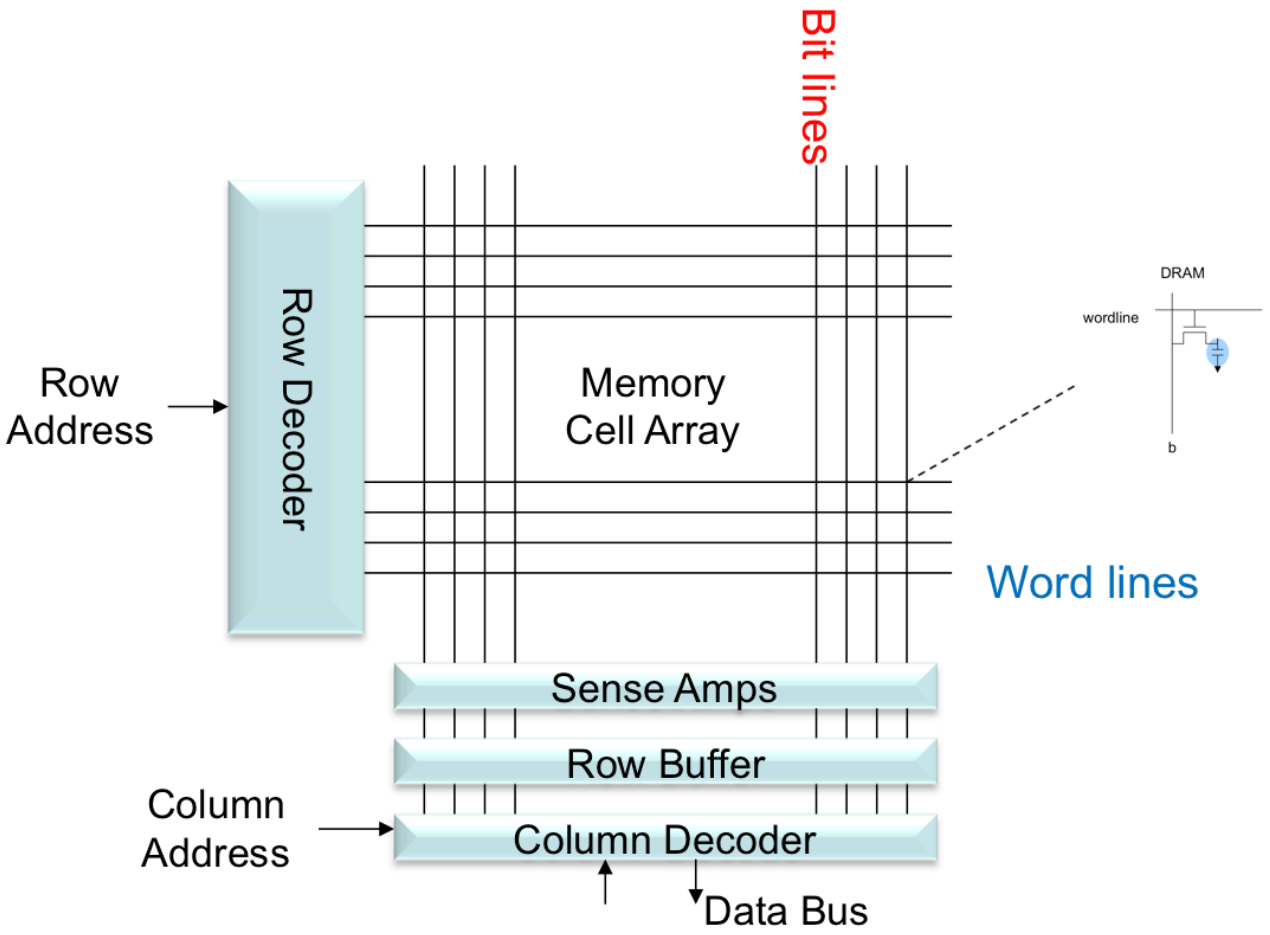
一張圖看懂 DRAM 各層單元的名稱





圖片來源:
http://www.techbang.com/posts/18381-from-the-channel-to-address-computer-main-memory-structures-to-understand

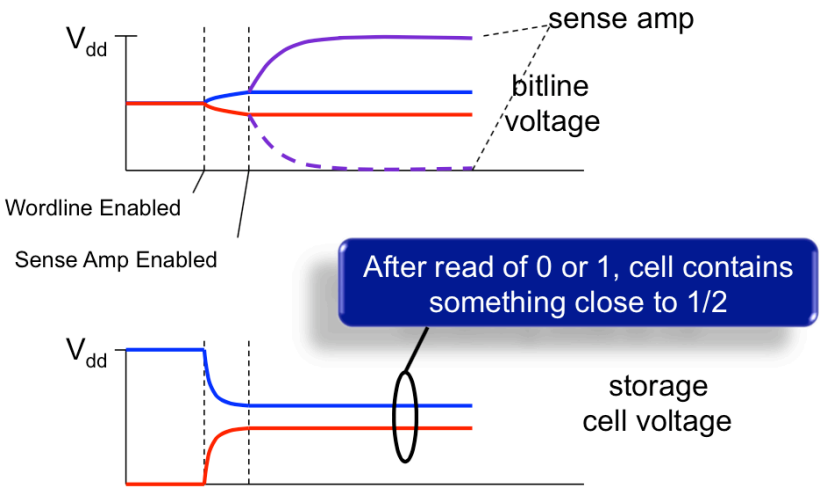
DRAM 平行運作的最小單元: Bank



圖片來源: http://www.cc.gatech.edu/~hyesoon/fall11/lec_dram.pdf

如何將 WL 資料讀到 Row Buffer?

- Precharge 每條 BL 至 0.5
- 打開想讀的某一條 WL
  - 使得該 WL 上的所有 cell 各自與對應位置的 BL 導通
  - 如果某 Cell 原本是存 1, 與對應的 BL 分享電荷後, 其 BL 會變得比 0.5 稍高
  - 如果某 Cell 原本是存 0, 與對應的 BL 分享電荷後, 其 BL 會變得比 0.5 稍低
- Sense Amp 放大 BL 訊號
  - 將 BL 稍高或稍低於 0.5 的微小訊號變化放大到 1 或 0, 送往 Row Buffer 以供讀取



注意! 一旦 Cell 與 BL 分享了彼此電荷, Cell 原本 1 或 0 的電位就會被中和掉, 所以 controller 在換去讀其他 WL 之前, 要先把 Row Buffer 資料寫回目前的 WL

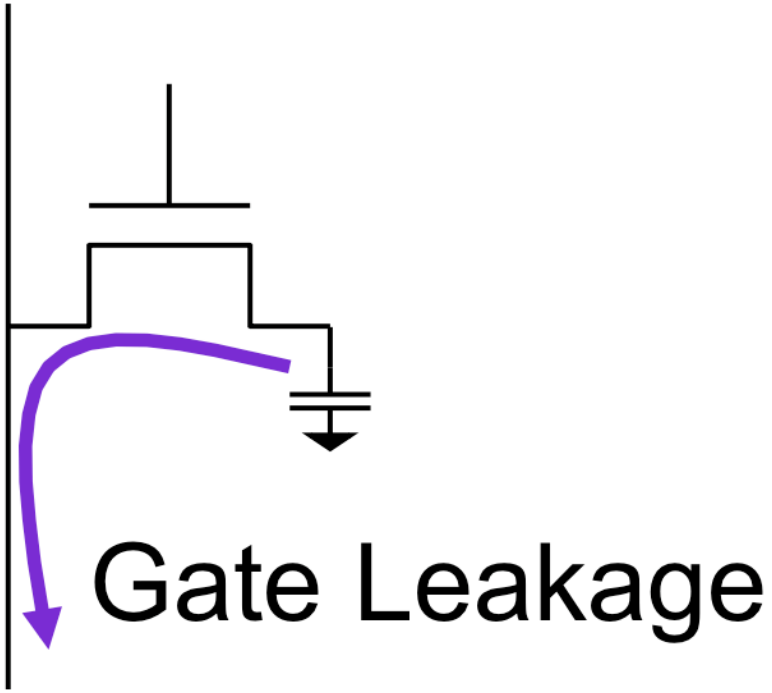
圖片來源: http://www.cc.gatech.edu/~hyesoon/fall11/lec_dram.pdf

如何將 Row Buffer 資料寫回 WL?

- 依據 Row Buffer 每個 bit 的值是 1 或 0,
  將對應的每一條 BL 給 charge 至電位 1
  或者 release 至電位 0
- 由於目前 WL 上的所有 cell 都各與對應的 BL 導通
  - 如果某 Cell 是與電位為 1 的 BL 導通,
    該 Cell 也會同時被 charge 到電位 1
  - 如果某 Cell是與電位為 0 的 BL 導通,
    該 Cell 也會同時被 release 到電位 0
- 如此, controller 就可以安全關上目前的 WL
  - 目前的 WL 關閉後, WL 上的所有 Cell 皆與 BL 斷開
  - 電位 1 或電位 0 的資訊將繼續保存在每個 Cell 內

DRAM 需要定期 Refresh 資料

- 即使某 WL 是關上的, 每一個 Cell 內的電荷仍然會慢慢漏掉
- 為確保資料, 當某 WL 被關上一段時間之後, controller 就必須重新打開該 WL 再關上
  - 這個動作稱為 refresh



圖片來源: http://www.cc.gatech.edu/~hyesoon/fall11/lec_dram.pdf

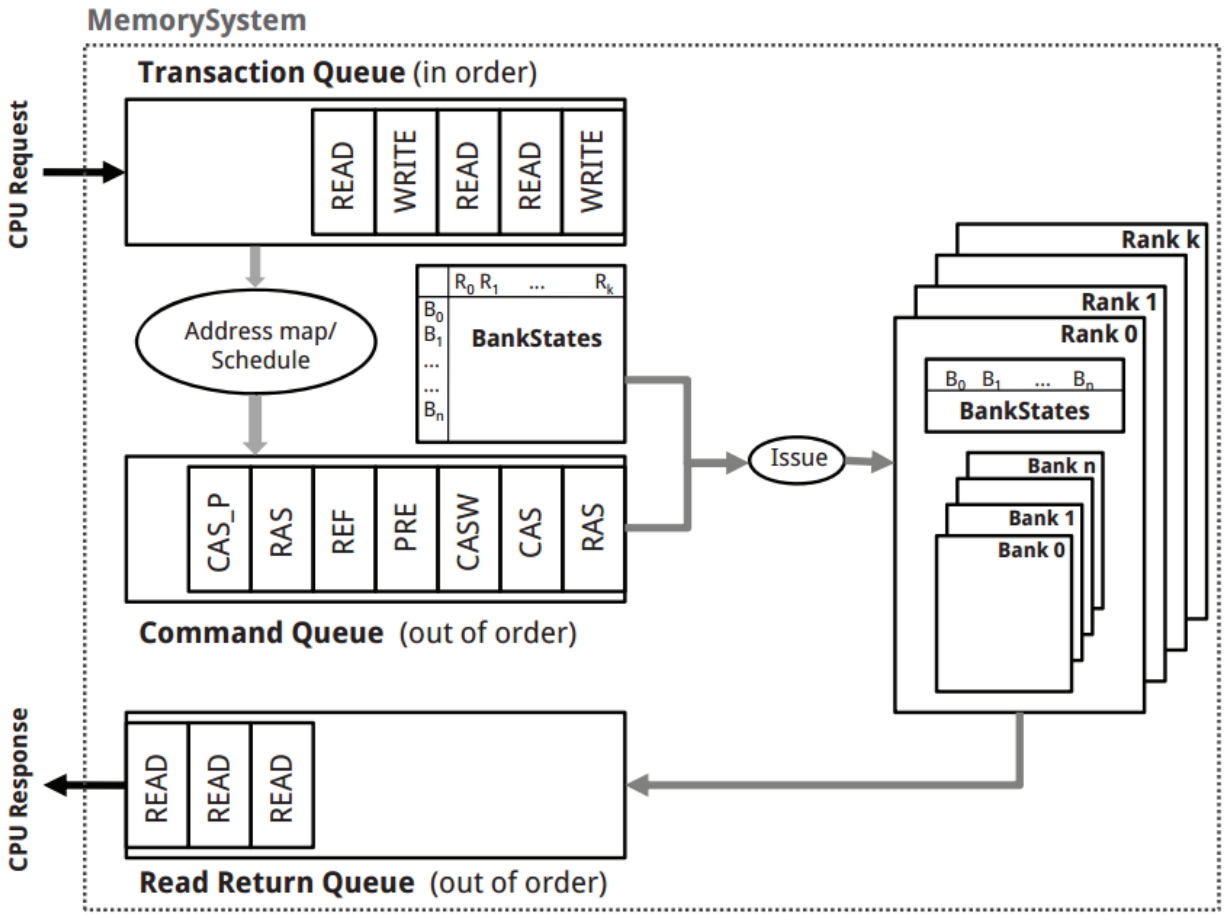
分析 Bank 的存取 Latency

- Case 1: 目前有一個 WL 已經在 Row Buffer 打開, 而且我們想存取的資料就在 Row Buffer 上
  - Bank Latency: CAS (Column Access Strobe)
    - CAS: 直接對 Row Buffer 某個 column 進行 read/write
- Case 2: 目前沒有 WL 打開
  - Bank Latency: RAS (Row Access Strobe) + CAS
    - RAS: 將想要存取的 WL 打開並讀到 Row Buffer 上
- Case 3:目前有一個 WL 在 Row Buffer 打開, 但不巧我們想存取的資料是放在別的 WL 上
  - Bank Latency: PRE (Precharge) + RAS + CAS
    - PRE: 將 Row Buffer 寫回 WL 並關上, 然後對 BL 重新 precharge

什麼時間點關上 WL 比較聰明?

- Close Page Policy
  - 每次 Column Access 後, 如果目前沒有其他等待的 request 需要存取目前的 WL, 那麼就預先關上目前的 WL (i.e. 立刻執行 PRE)
  - 優點: 如果未來出現的 request 要存取其他 WL, 可以省去等待 PRE 的時間 (i.e. case 2)
- Open Page Policy
  - 每次 Column Access 後, 保留目前的 WL 不關上
  - 優點: 如果未來出現的 request 又剛好要存取目前的 WL, 可以省去等待 RAS 的時間 (i.e. case 1)

DRAMSim2 架構



圖片來源: *DRAMSim2: A Cycle Accurate Memory System Simulator*

下載及編譯 DRAMSim2

- 取得最新版本的 DRAMSim2
  - git clone https://github.com/dramninjasUMD/DRAMSim2.git
- 依照使用需求,有不同的編譯方式
  - 把 DRAMSim2 編譯成一支單獨程式, 這支程式可以讀入 trace 檔進行模擬 (trace-driven)
    - make
  - 把 DRAMSim2 編譯成一個 library, 然後由其他 simulator (例如 MARSSx86) 執行時, 即時產生 memory access 給 DRAMSim2 模擬
    - make libdramsim.so

兩種使用 DRAMSim2 的方式

- 如上頁所示, 有兩種使用方式:
  - 第一種方式: 單獨執行 DRAMSim2, 透過參數列來指定 trace 檔案名稱以開始模擬
    - 請參考 DRAMSim2 的 README 檔了解命令列格式
    - 程式由 DRAMSim2 的 main 開始執行, 需要修改的人請參考 TraceBasedSim.cpp 檔案
  - 第二種方式: 外部的 simulator 在執行模擬時, 即時產生 memory access 並調用 DRAMSim2 的 library 去同步模擬 (execution-driven)
    - 範例程式請看 example_app 資料夾
- 時間因素, 在此只講解第二種方式 (應用較廣)

DRAMSim2 範例程式: Akarin

- Akarin 是一支極度簡化的 CPU 模擬器
  - 單核心且只跑一支程式 (single thread)
    - 程式可以一直執行, 不會被 context-switched 走
  - 程式只會做三件事:
    - 計算 (需要消耗若干個 CPU cycle)
    - 讀資料 (假設 CPU 沒有 cache, 要直接讀 DRAM)
    - 寫資料 (假設 CPU 沒有 cache, 要直接寫 DRAM)
- 視資料的急迫性而定, 存取 DRAM 可能會讓 Akarin 暫時處於 CPU stalled 的狀況

下載及編譯 Akarin

- 在繼續前, 請確認你的 DRAMSim2 已事先編譯成 library 了 (參考前面幾頁的說明)
- 取得最新版本的 Akarin 原始碼
  - git clone https://github.com/gengyouchen/akarin.git
- 在 Makefile, 修改第一行 dramsim_path 的值為你自己放 DRAMSim2 資料夾的正確路徑
- 一切準備就緒後, 就可以編譯 Akarin 了
  - make

Akarin 是如何調用 DRAMSim2 的?

- 建立一個新的 DRAMSim2 物件實體
  - 調用 DRAMSim::getMemorySystemInstance
- 每個 CPU cycle 都要調用 DRAMSim2 的 update
  - DRAMSim2 內部會自己去換算幾次 CPU cycle 相當於一次的 DRAM cycle, 並進行相對應的模擬動作
  - 初始化時需調用 DRAMSim2 的 setCPUClockSpeed 來告知 DRAMSim2 目前 CPU 的絕對速度 (單位:Hz)
- 調用 DRAMSim2 的 addTransaction 發送 request
  - 初始化時可調用 DRAMSim2 的 RegisterCallbacks 來告知 DRAMSim2 當有 request 做完後, 應該被調用的 callback 函式 (我們會在 callback 中解除 CPU stalled)

建立並初始化 DRAMSim2 實體

```
Akarin::Akarin(
     const string &dev, const string &sys, const string &pwd, const string &trc,
     unsigned megsOfMemory, uint64_t cpuClkFreqHz) :
          readFinishedCallback(this, &Akarin::onReadFinished),
          writeFinishedCallback(this, &Akarin::onWriteFinished),
          currentCPUCycle(0), stalledReadAddress(-1), stalledWriteAddress(-1)
{
     memorySource = DRAMSim::getMemorySystemInstance(
                                          dev, sys, pwd, trc, megsOfMemory);
     memorySource->setCPUClockSpeed(cpuClkFreqHz);
     memorySource->RegisterCallbacks(
                          &readFinishedCallback, &writeFinishedCallback, NULL);
}
```

於 callback 解除 CPU stalled 狀況

```
void Akarin::onReadFinished(
     unsigned channelID, uint64_t logicalAddress, uint64_t currentDRAMCycle)
{
     if (stalledReadAddress == logicalAddress)
          stalledReadAddress = -1;
}

void Akarin::onWriteFinished(
     unsigned channelID, uint64_t logicalAddress, uint64_t currentDRAMCycle)
{
     if (stalledWriteAddress == logicalAddress)
          stalledWriteAddress = -1;
}
```

CPU 每前進一 cycle 就調用 update

```
void Akarin::doComputing(
                    uint64_t busyCPUCycle)
{
    for (int i = 0; i < busyCPUCycle; i++) {
        currentCPUCycle++;
        memorySource->update();
    }
}
```

使用 addTransaction 發送 request

```
void Akarin::accessMemory(uint64_t logicalAddress, bool isWrite, bool isStalled)
{
      memorySource->addTransaction(isWrite, logicalAddress);
      if (isStalled) {
            if (isWrite) {
                  stalledWriteAddress = logicalAddress;
                  do {
                        currentCPUCycle++;
                        memorySource->update();
                  } while (stalledWriteAddress != -1);
            } else {
                  stalledReadAddress = logicalAddress;
                  do {
                        currentCPUCycle++;
                        memorySource->update();
                  } while (stalledReadAddress != -1);
            }
      }
}
```

如果 request 會造成 CPU stalled,
持續前進 CPU cycle 並調用 update,
直到 CPU stalled 狀況解除才返回.

Request 邏輯地址對應的物理位址

- 可在 system.ini 選擇一種 mapping scheme
  1. [request 邏輯地址] -> [ch | rank | row | col | bank | offset]
  2. [request 邏輯地址] -> [ch | row | col | bank | rank | offset]
  3. [request 邏輯地址] -> [ch | rank | bank | col | row | offset]
  4. [request 邏輯地址] -> [ch | rank | bank | row | col | offset]
  5. [request 邏輯地址] -> [ch | row | col | rank | bank | offset]
  6. [request 邏輯地址] -> [ch | row | bank | rank | col | offset]
  7. [request 邏輯地址] -> [row | col | rank | bank | ch | offset]
- 一般來說, CPU 發送 DRAM request 皆使用邏輯地址,
  上面跑的程式並不需要知道 DRAM 實際的物理位址
  - accessMemory(*logicalAddress*, isWrite, isStalled)
- 但為了方便學習, Akarin 也可直接用物理位址來發 request
  - accessMemory(*channelID*, *rankID*, *bankID*, *rowID*, *columnID*, isWrite, isStalled)

編寫上層應用程式的行為

```
int main()
{
      Akarin myCPU(“ini/DDR2_micron_16M_8b_x8_sg3E.ini”, “system.ini”, “.”, “hello_world”,
                        4096, /* Memory: 4GB */
                        3.2e9 /* CPU: 3.2GHz */
      );
      myCPU.accessMemory(0, 0, 0, 1, 3, true, false);
      myCPU.doComputing(5);
      myCPU.accessMemory(0, 0, 0, 2, 0, true, false);
      myCPU.doComputing(5);
      myCPU.accessMemory(0, 0, 0, 1, 8, true, false);
      myCPU.doComputing(5);
      myCPU.accessMemory(0, 0, 0, 2, 4, true, false);
      myCPU.doComputing(5);
      myCPU.accessMemory(0, 0, 0, 1, 1, true, false);
      myCPU.doComputing(5);
      myCPU.accessMemory(0, 0, 0, 2, 1, true, false);
      myCPU.doComputing(975);
```

在此處編寫上層應用程式的行為,
可命令 CPU 計算, 或讀/寫 DRAM,
並指定讀/寫是否造成 CPU stalled

觀察讀寫 DRAM 的次序與時間

- 編譯完 Akarin 後可輸入 ./Akarin 來執行, 或者輸入 make exec 也可以
- 以下是上一頁程式碼會打印出的訊息
  - Start writing 0x40c00 (ch=0, rank=0, bank=0, row=1, col=3) at CPU cycle 1000
  - Start writing 0x80000 (ch=0, rank=0, bank=0, row=2, col=0) at CPU cycle 1005
  - Start writing 0x42000 (ch=0, rank=0, bank=0, row=1, col=8) at CPU cycle 1010
  - Start writing 0x81000 (ch=0, rank=0, bank=0, row=2, col=4) at CPU cycle 1015
  - Start writing 0x40400 (ch=0, rank=0, bank=0, row=1, col=1) at CPU cycle 1020
  - Start writing 0x80400 (ch=0, rank=0, bank=0, row=2, col=1) at CPU cycle 1025
  - Finish writing 0x40c00 (ch=0, rank=0, bank=0, row=1, col=3) at CPU cycle 1114
  - Finish writing 0x42000 (ch=0, rank=0, bank=0, row=1, col=8) at CPU cycle 1133
  - Finish writing 0x40400 (ch=0, rank=0, bank=0, row=1, col=1) at CPU cycle 1153
  - Finish writing 0x80000 (ch=0, rank=0, bank=0, row=2, col=0) at CPU cycle 1325
  - Finish writing 0x81000 (ch=0, rank=0, bank=0, row=2, col=4) at CPU cycle 1345
  - Finish writing 0x80400 (ch=0, rank=0, bank=0, row=2, col=1) at CPU cycle 1364
- 打印訊息說明
  - Start 是指 CPU 將 request 送入 DRAM controller 的時間點
  - Finish 是 DRAM controller 實際完成該 request 的時間點

問題討論

- 請閱讀 main.cpp 內附的 5 組上層範例程式
  1. non-blocking writes with some locality
  2. non-blocking writes with no locality
  3. non-blocking read with some locality
  4. non-blocking read with no locality
  5. blocking read with some locality
- 觀察 CPU 把每個 request 送進 DRAM controller 的時間以及他們 finish 的時間, 想想看 DRAM controller 是怎麼 schedule 這些 request 的?
- 觀察 CPU 執行完每一組範例程式的總時間, 想想看為什麼某幾組範例程式明顯要比另外幾組範例程式要花更多時間才能執行完?

THANK YOU FOR LISTENING!!