[SoC Lab] LabD

tags: SoC Lab, SOC Design

Team 13

Student ID	Name
311551095	林聖博
312551174	張祐誠

附上此篇Hackmd Link: https://hackmd.io/@Sheng08/r1vrU5Ud6

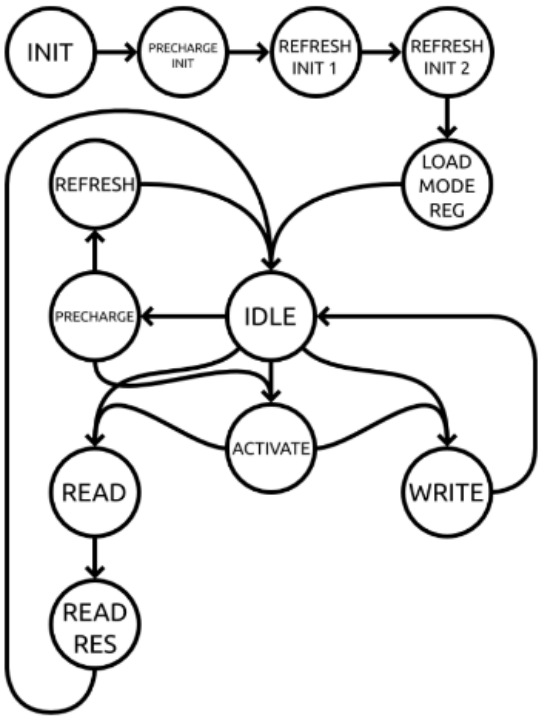
Lab D Spec

- Lab D (sdram)
  - https://github.com/bol-edu/caravel-soc_fpga-lab/tree/main/lab-sdram

1. SDRAM controller design

SDRAM Controller

- INIT→IDLE
- IDLE→ACTIVATE→WRITE→IDLE
- IDLE→ACTIVATE→READ→READ_RES→IDLE
- IDLE→WRITE→IDLE
- IDLE→READ→READ_RES→IDLE
- IDLE→PRECHARE→ACTIVATE→WRITE→IDLE
- IDLE→PRECHARE→ACTIVATE→READ→READ_RES→IDLE
- IDLE→PRECHARE→REFRESH→IDLE



Reference: https://alchitry.com/sdram-mojo

> Ref: https://github.com/bol-edu/caravel-soc_fpga-lab/blob/main/lab-sdram/LabD-sdram_workbook.pdf

- 各狀態說明：

狀態	說明
INIT (初始化)	初始化記憶體參數和設定。包括設定時序參數、模式註冊等
IDLE (閒置)	等待接收記憶體操作命令
ACTIVATE (激活)	激活特定記憶體 Row 以進行存取(Read 或 Write)
WRITE (寫入)	向已激活 Row 指定的 Col 寫入資料
READ (讀取)	從已激活 Row 指定的 Col 讀取資料
READ_RES (讀取回應)	處理從記憶體**讀取**的資料回應
PRECHARGE (緩啟動/預充電)	關閉記憶體 Row 以準備下一操作命令
REFRESH (刷新)	刷新記憶體以維持記憶體中存儲的資料穩定

> 上述皆由 controller 進行記憶體的管理與各項操作。
>
> 🚀 補充：
>
> - REFRESH 為了維持記憶體中存儲資料的穩定，需不斷刷新，也分成兩種模式：

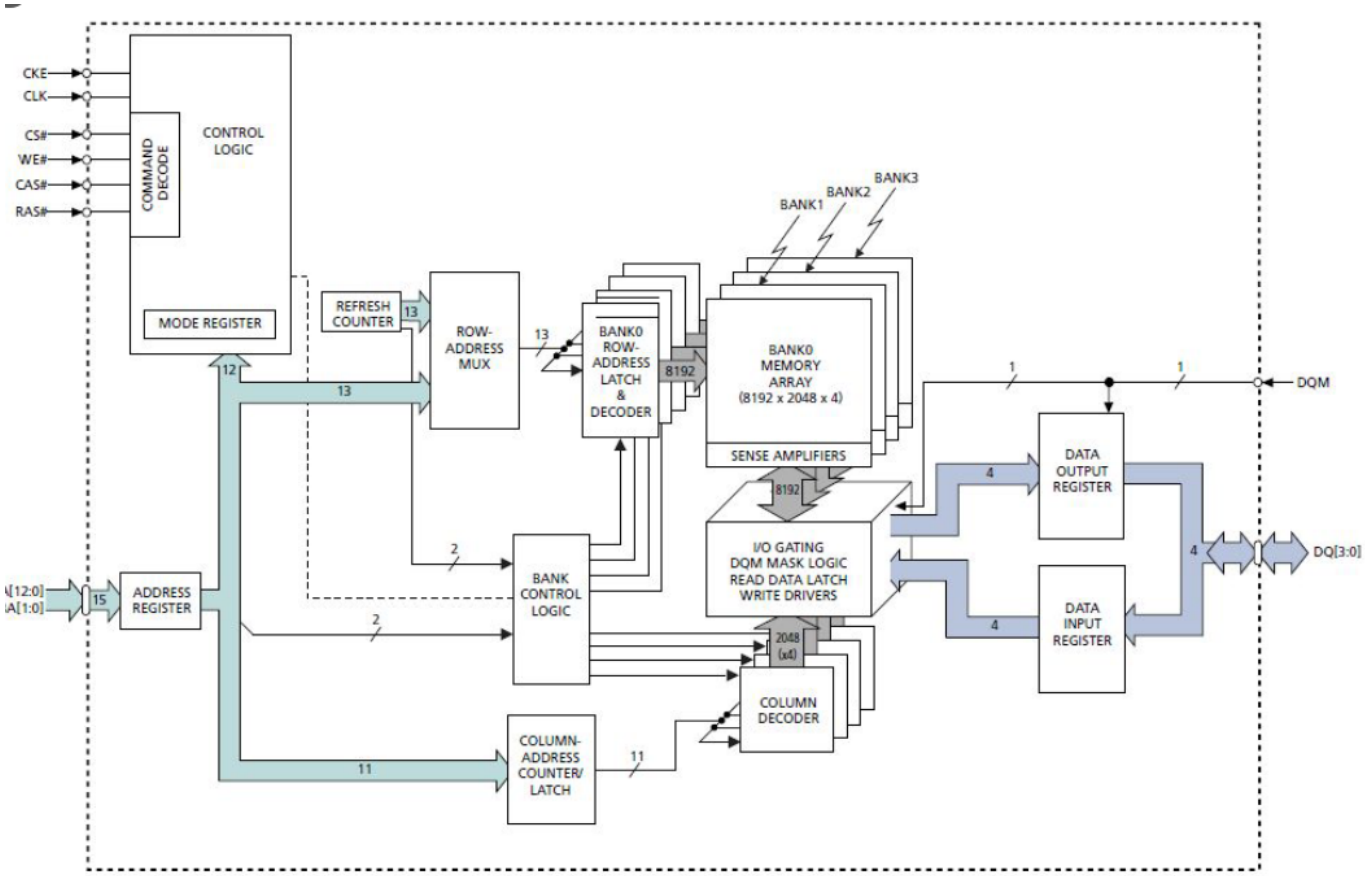
> 1. **Auto Refresh (自動刷新):** 在這種模式下，controller 負責定期發送刷新命令，以確保所有記憶體單元都獲得刷新。在執行自動刷新前，必須先將所有處於激活狀態的 bank 進行 PRECHARGE，以確保記憶體行對刷新命令做出正確反應。
> 2. **Self Refresh (自我刷新):** 此模式允許 SDRAM 獨立於外部 controller 進行刷新操作，通常在系統進入低功耗狀態時使用。自我刷新模式下，SDRAM 內部邏輯會自動進行刷新週期，不需外部 controller 介入。

- FSM 說明：

狀態轉換序列	說明
INIT → IDLE	初始化所有 bank（設定時序、模式等），並進入閒置狀態，準備接收指令
IDLE → ACTIVATE → WRITE → IDLE	在特定 bank 的閒置狀態下，激活記憶體 Row，執行**寫入**操作，然後該 bank 回到閒置狀態
IDLE → ACTIVATE → READ → READ_RES → IDLE	在特定 bank 的閒置狀態下，激活記憶體 Row，執行**讀取**操作，處理讀取回應，然後該 bank 回到閒置狀態
IDLE → WRITE → IDLE	在特定 bank 的閒置狀態下，直接執行**寫入**操作（假設該 Row 已激活），然後該 bank 回到閒置狀態
IDLE → READ → READ_RES → IDLE	在特定 bank 的閒置狀態下，直接執行**讀取**操作，處理讀取回應，然後該 bank 回到閒置狀態
IDLE → PRECHARGE → ACTIVATE → WRITE → IDLE	在特定 bank 的閒置狀態下，進行 PRECHARGE，激活記憶體 Row，執行**寫入**操作，然後該 bank 回到閒置狀態
IDLE → PRECHARGE → ACTIVATE → READ → READ_RES → IDLE	在特定 bank 的閒置狀態下，進行 PRECHARGE，激活記憶體 Row，執行**讀取**操作，處理讀取回應，然後該 bank 回到閒置狀態
IDLE → PRECHARGE → REFRESH → IDLE	在特定 bank 的閒置狀態下，進行 PRECHARGE 和 REFRESH 以維持數據穩定，然後該 bank 回到閒置狀態

> SDRAM 通常被分割成多個 bank，每個 bank 可以獨立操作(允許 controller 同時在不同的 bank 上執行不同的操作)，從而提高整體存取效率和總體性能。

2. SDRAM bus protocol



- 外部訊號：

> 這些訊號用於控制 SDRAM 的操作和資料傳輸
>
> - SDRAM Device
>   - sdram_cle, sdram_cs, sdram_cas, sdram_ras, sdram_we
>   - sdram_dqm, sdram_ba, sdram_a
>   - sdram_dqi, sdram_dqo

訊號類型	訊號名稱	說明
時脈相關	`clk`	同步時脈訊號
	`Cke` (Clock Enable, `sdram_sle`)	為 clk 訊號的 enable
基本控制	`Cs_n` (Chip Select, `sdram_cs`)	Chip 選擇，控制裝置回應
	`Cas_n` (Column Address Strobe, `sdram_cas`)	控制 Column Address 的讀取
	`Ras_n` (Row Address Strobe, `sdram_ras`)	控制 Row Address 的讀取
	`We_n` (Write Enable, `sdram_we`)	write 的 enable 的訊號

訊號類型	訊號名稱	說明
資料控制	`Dqm` (Data I/O Mask, `sdram_dqm`)	資料位的讀寫控制，可控制 I/O port 取消那些輸入或輸出的資料
	`Ba` (Bank Address, `sdram_ba`)	Bank 的位址。用於選擇哪一個 Bank
	`Addr` (Address, `sdram_a`)	指定 Column 或 Row Address
資料傳輸	`Dqi` (Data Input, `sdram_dqi`)	資料輸入訊號
	`Dqo` (Data Output, `sdram_dqo`)	資料輸出訊號

- 內部訊號：

> 這些訊號為 SDRAM Controller 內部用來管理記憶體操作

```
// Commands Decode
wire      Active_enable    = ~Cs_n & ~Ras_n &  Cas_n &  We_n;
wire      Aref_enable      = ~Cs_n & ~Ras_n & ~Cas_n &  We_n;
wire      Burst_term       = ~Cs_n &  Ras_n &  Cas_n & ~We_n;
wire      Mode_reg_enable  = ~Cs_n & ~Ras_n & ~Cas_n & ~We_n;
wire      Prech_enable     = ~Cs_n & ~Ras_n &  Cas_n & ~We_n;
wire      Read_enable      = ~Cs_n &  Ras_n & ~Cas_n &  We_n;
wire      Write_enable     = ~Cs_n &  Ras_n & ~Cas_n & ~We_n;
```

訊號名稱	說明
`Activate_enable`	啟用(激活)特定 Row 的操作。並等待 `Read_enbale` 或 `Write_enable`
`Aref_enable`	自動刷新(auto refresh)操作的 enable 訊號
`Burst_term`	截斷 Read / Write 操作中的 Burst 傳輸
`Mode_reg_enable`	模式註冊的 enable 訊號
`Prech_enable`	PRECHARGE 操作的 enable 訊號
`Read_enable`	啟用(激活) Column Address 以進行**讀取**操作
`Write_enable`	啟用(激活) Column Address 以進行**寫入**操作

🚀 補充：

- 「Strobe」： 通常指的是一種用來同步或指示特定事件或操作的訊號。這種訊號通常用於控制數據的讀取或寫入時機，確保資料在正確的時刻被送入或從某個設備中讀出。例如：
  - **RAS (Row Address Strobe):** 用於控制行地址的讀取時機。
  - **CAS (Column Address Strobe):** 用於控制列地址的讀取時機。
- Burst Termination(`Burst_term`):
  - **終止當前突發傳輸：** 當執行突發終止命令時，正在進行的連續讀寫操作會被立即中斷
  - **適用於各類突發長度：** 無論是固定長度或整頁長度的突發傳輸，都可被此命令終止

    - **不涉及記憶體 Row 的關閉：** 執行突發終止命令不會 PRECHARGE 或關閉當前激活的記憶體 Row
    - **需單獨 PRECHARGE 以關閉 Row：** 若需關閉被激活的記憶體 Row，必須執行額外的 PRECHARGE 操作
- 突發傳輸（Burst Transfer）:
    - 是一種記憶體或資料傳輸的模式，其中資料以**連續的快速序列傳輸**，而不是單個 Byte 逐一傳輸。這種傳輸模式通常用於提高資料傳輸的效率，尤其是在大量資料需要被快速讀取或寫入記憶體時。
    - 在突發傳輸模式中，一旦開始傳輸，資料將連續不斷地流動，直到達到預設的傳輸數量（例如: 一定數量的 Byte 或 Word ）。這種方式減少了每次資料傳輸前後所需的設定時間，從而提高整體傳輸速率。
    - 在許多高速記憶體和儲存設備中，如: SDRAM(同步動態隨機存取記憶體)等，用於最大化資料處理效率和Bandwidth利用率。

3. Introduce the prefetch scheme

```
reg[22:0] prefetch_address;
reg prefetch_en;

always@(posedge clk)begin
    if(in_valid)
        prefetch_address <= addr + 22'd4;
    else if(!in_valid && out_valid_delay)
        prefetch_address <= 0;

    if((prefetch_address == addr) && in_valid && !rw)
        prefetch_en <= 1;
    else prefetch_en <= 0;
end
```

1. **定義暫存器**:
    - `reg[22:0] prefetch_address`: 此 23 bits 的暫存器用來存放 Prefetch 的位址
    - `reg prefetch_en`: 此暫存器是用來控制是否啟動 Prefetch
2. **更新 Address**:
    - `if(in_valid)`: 當 `in_valid` 訊號時，將 `prefetch_address` 設為目前的位址（`addr`）再加上 4，表示將準備讀取接下來的記憶體位置
    - `else if(!in_valid && out_valid_delay)`: 如果沒有 `in_valid` 輸入，且 `out_valid_delay` 成立，那麼就將 `prefetch_address` 歸零
3. **判斷是否啟用預取**:
    - `(prefetch_address == addr) && in_valid && !rw`: 如果 `prefetch_address` 和目前的 address 相同，且有 `in_valid` ，而且 `rw` 為 false（表示讀取操作），那麼就將`prefetch_en` 設為 1，代表先前已經完成該 address prefetch 的操作(通常在 IDLE 狀態進行判斷)
    - `prefetch_en <= 0`: 在其他情況下，關閉 Prefetch

Prefetch 機制流程

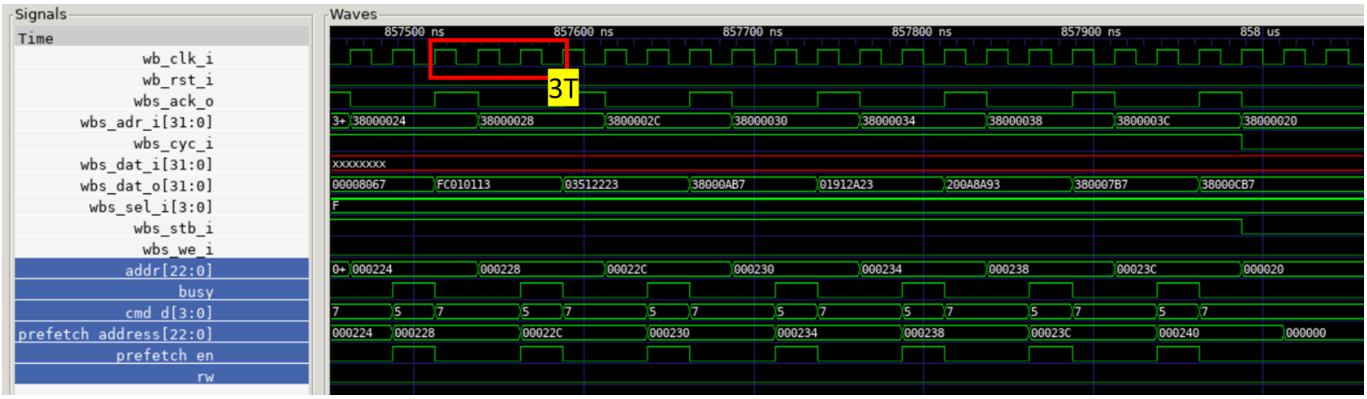
- 每當 clk 周期開始時，根據當前的輸入狀態和 address 來更新 `prefetch_address`
- 透過比較 `prefetch_address` 和當前的 address，以及讀寫(`rw`)狀態，來決定是否啟動 prefetch

優化

- IDLE 狀態中加入對 Prefetch bank 的訪問。如此當連續的讀取操作針對的是同一 bank 時，就無需再次 ACTIVATE(激活)，從而提高效率

效能提升

- **讀取延遲的降低** 原本連續兩次 Read 操作之間需要花費數個 clock (T)，但透過上述 Prefetch 機制達到只需 3T 間隔延遲。
- **總體效率的提升** 因此，READ_RES 狀態的時間成本降低，整體的記憶體訪問效率得到提升。此設計既減少了等待時間，又提高記憶體操作的回應速度。



4. Introduce the bank interleave for code and data

根據 Address 來判斷是否為 **code 段** 或是 **data 段**

```
assign request_data = (wbs_stb_i & wbs_cyc_i &(wbs_adr_i[11:8] == 4'b0 | wbs_adr_i[11:8] == 4'b1)) ? 1 : 0;
assign request_mprj = ((wbs_adr_i[31:24] == 8'h26)) ? 1 : 0;
assign request_code = (wbs_stb_i & wbs_cyc_i &(wbs_adr_i[11:8] == 4'b0010 | wbs_adr_i[11:8] == 4'b0011)) ? 1 : 0;

assign code_bank = (wbs_adr_i[9:8]>2'b01) ? (wbs_adr_i[9:8]-2'b10) : wbs_adr_i[9:8];
assign code_addr = (request_code) ? {wbs_adr_i[22:10], code_bank, wbs_adr_i[7:0]} : 0;
assign data_bank = (wbs_adr_i[9:8]<2'b10) ? (wbs_adr_i[9:8]+2'b10) : wbs_adr_i[9:8];
assign data_addr = (request_data) ? {wbs_adr_i[22:10], data_bank, wbs_adr_i[7:0]} : 0;
```

5. Introduce how to modify the linker to load

將 Code 與 Data 分別放在0x3800000與0x3800200不同起始位址

```
38000000 <__mulsi3>:
38000000: 00050613          mv  a2,a0
38000004: 00000513          li  a0,0
38000008: 0015f693          andi  a3,a1,1
3800000c: 00068463          beqz  a3,38000014 <__mulsi3+0x14>
38000010: 00c50533          add a0,a0,a2
38000014: 0015d593          srli  a1,a1,0x1
38000018: 00161613          slli  a2,a2,0x1
3800001c: fe0596e3          bnez  a1,38000008 <__mulsi3+0x8>
38000020: 00008067          ret
```

```
47      .data :
48      {
49          . = ALIGN(8);
50          _fdata = .;
51          *(.data .data.* .gnu.linkonce.d.*)
52          *(.data1)
53          _gp = ALIGN(16);
54          *(.sdata .sdata.* .gnu.linkonce.s.*)
55          . = ALIGN(8);
56          _edata = .;
57      } > sdrdata AT > flash
58
59      .bss :
60      {
61          . = ALIGN(8);
62          _fbss = .;
63          *(.dynsbss)
64          *(.sbss .sbss.* .gnu.linkonce.sb.*)
65          *(.scommon)
66          *(.dynbss)
67          *(.bss .bss.* .gnu.linkonce.b.*)
68          *(COMMON)
69          . = ALIGN(8);
70          _ebss = .;
71          _end = .;
72      } > sdrdata AT > flash
73
74      .mprjram :
75      {
76          . = ALIGN(8);
77          _fsram = .;
78          *libgcc.a:*(.text .text.*)
79
80      } > sdrcode AT > flash
81  }
82
```

Section	Memory
Code段	sdrcode 0x3800000
Data段(.data 和 .bss)	sdrdata 0x3800200

原始實作方式會導致會導致 code 放在 flash(0x10 SPIflash)上導致執行相當緩慢

```
MEMORY {
        vexriscv_debug : ORIGIN = 0xf00f0000, LENGTH = 0x00000100
        dff : ORIGIN = 0x00000000, LENGTH = 0x00000400
        dff2 : ORIGIN = 0x00000400, LENGTH = 0x00000200
        flash : ORIGIN = 0x10000000, LENGTH = 0x01000000
        mprj : ORIGIN = 0x30000000, LENGTH = 0x00100000
        mprjram : ORIGIN = 0x38000000, LENGTH = 0x00400000
        hk : ORIGIN = 0x26000000, LENGTH = 0x00100000
        csr : ORIGIN = 0xf0000000, LENGTH = 0x00010000
}
```

```
.text :
{
        _ftext = .;
        /* Make sure crt0 files come first, and they, and the isr */
        /* don't get disposed of by greedy optimisation */
        *crt0*(.text)
        KEEP(*crt0*(.text))
        KEEP(*(.text.isr))

        *(.text .stub .text.* .gnu.linkonce.t.*)
        _etext = .;
} > flash
```

本次 Lab 參考以下討論進行優化，將 code 與 data 放在 0x38 區段:

```
MEMORY {
    vexriscv_debug : ORIGIN = 0xf00f0000, LENGTH = 0x00000100
    dff : ORIGIN = 0x00000000, LENGTH = 0x00000400
    dff2 : ORIGIN = 0x00000400, LENGTH = 0x00000200
    flash : ORIGIN = 0x10000000, LENGTH = 0x01000000
    mprj : ORIGIN = 0x30000000, LENGTH = 0x00100000
    sdrcode : ORIGIN = 0x38000000, LENGTH = 0x00000200
    sdrdata : ORIGIN = 0x38000200, LENGTH = 0x00000200
    hk : ORIGIN = 0x26000000, LENGTH = 0x00100000
    csr : ORIGIN = 0xf0000000, LENGTH = 0x00010000
}
```

```
.mprjram :
{
    . = ALIGN(8);
    _fsram = .;
    *libgcc.a:*(.text .text.*)

} > sdrcode AT > flash
```

- 參考串: https://github.com/bol-edu/HLS-SOC-Discussions/discussions/189#discussioncomment-7915925

6. Observe SDRAM access conflicts with SDRAM

- SDRAM 儲存的資料會隨著時間逐漸衰減，因此需要定期進行 Refresh 來重置存儲單元狀態，確保資料的穩定性與可靠性。
- 衝突情況:
    - 當試圖訪問一個記憶體區域(bank)，而該區域正在進行 Refresh 時，就會發生衝突。這種衝突可能會導致效能降低，因為存取請求必須等待 Refresh 完成

解決 conflicts 策略

1. 調整 Refresh 週期:
    - 降低 Refresh 週期可以減少 Refresh 操作與系統存取之間的衝突機會
    - 不是減少Refresh的頻率，而是優化 Refresh 操作的行程，使其更好地配合系統的存取模式
2. 增加存取時間:
    - 藉由減少這些衝突(conflicts)，系統在存取記憶體時可以擁有更多的 time window，從而增加了有效的存取時間
3. 提升 SDRAM 效能:
    - 減少存取衝突不僅減少了等待時間，也提高了記憶體的整體效能和回應速度

總結 觀察和管理 SDRAM 的訪問衝突，尤其是與 Refresh 相關的 conflicts，是提高系統效能的重要環節。通過良好的調整 Refresh 週期和優化存取策略，可以顯著減少這些衝突的發生，從而確保記憶體操作的效率和速度。這樣的策略不僅提高了 SDRAM 的性能，也保證了資料的穩定性和可靠性。

結果 本次 Lab 可以透過 `-O2` 或 `-Os` compiler 優化達到更好的效能，但 `-O3` 上仍存在問題(無法順利完成矩陣運算)

```
ubuntu@ubuntu2004:~/caravel-soc_fpga-lab/lab-sdram/testbench/counter_la_mm$ source run_sim
Reading counter_la_mm.hex
counter_la_mm.hex loaded into memory
Memory 5 bytes = 0x6f 0x00 0x00 0x0b 0x13
VCD info: dumpfile counter_la_mm.vcd opened for output.
LA Test 1 started
counter_la_mm_tb.uut.mprj.mprj.user_bram : at time           957413000 ERROR: Bank is not Activated for Read
counter_la_mm_tb.uut.mprj.mprj.user_bram : at time          1054688000 ERROR: Bank is not Activated for Read
counter_la_mm_tb.uut.mprj.mprj.user_bram : at time          1151888000 ERROR: Bank is not Activated for Read
counter_la_mm_tb.uut.mprj.mprj.user_bram : at time          1248738000 ERROR: Bank is not Activated for Read
Call function matmul() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x003e
Call function matmul() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x0044
Call function matmul() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x004a
Call function matmul() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x0050
LA Test 2 passed
            1349413
```

TODO 本次Lab後，預計期末進一步優化項目：

1. 成功使用 `-O3` 進行編譯並完成目標運算
2. 雖目前仍完成矩陣運算，但執行時仍存在有`ERROR: Bank is not Activated for Read`訊息，推測某些prefetch判斷有些問題，預計於期末專案實作時一併解決本次 SDRAM 設計問題

Github link

- https://github.com/NYCU-SOC-Design-Team13/SoC-LabD
