2조

2014314636 우제율

2016311821 한승하

“jasmine.h”

“jasmine.h”에서는 보드의 기본적인 세팅에 관하여 정의하고 있습니다.

DRAM의 사이즈, Bank의 Bitmap 주소, Clock의 속도와

여러 Option과 Channel과 Bank의 개수,

Bank, Block, Page, Sector의 개수 및 Byte 크기와

Virtual 값과 Physical 값이 모두 적혀있습니다.

```
#define FLASH_TYPE          K9LCG08U1M
#define DRAM_SIZE           65075200
#define BANK_BMP            0x00330033
#define CLOCK_SPEED         175000000

#define OPTION_2_PLANE              1
#define OPTION_ENABLE_ASSERT        0
#define OPTION_FTL_TEST             0
#define OPTION_UART_DEBUG           1
#define OPTION_SLOW_SATA            0
#define OPTION_SUPPORT_NCQ          0
#define OPTION_REDUCED_CAPACITY     0

#define CHN_WIDTH           2    //
#define NUM_CHNLS_MAX       4
#define BANKS_PER_CHN_MAX   8
#define NUM_BANKS_MAX       32
```

```
#define BYTES_PER_SECTOR        512

#define NUM_PSECTORS_8GB        16777216
#define NUM_PSECTORS_16GB       (NUM_PSECTORS_8GB*2)
#define NUM_PSECTORS_32GB       (NUM_PSECTORS_16GB*2)
#define NUM_PSECTORS_40GB       (NUM_PSECTORS_8GB*5)
#define NUM_PSECTORS_48GB       (NUM_PSECTORS_16GB*3)
#define NUM_PSECTORS_64GB       (NUM_PSECTORS_32GB*2)
#define NUM_PSECTORS_80GB       (NUM_PSECTORS_16GB*5)
#define NUM_PSECTORS_96GB       (NUM_PSECTORS_32GB*3)
#define NUM_PSECTORS_128GB      (NUM_PSECTORS_64GB*2)
#define NUM_PSECTORS_160GB      (NUM_PSECTORS_32GB*5)
#define NUM_PSECTORS_192GB      (NUM_PSECTORS_64GB*3)
#define NUM_PSECTORS_256GB      (NUM_PSECTORS_128GB*2)
#define NUM_PSECTORS_320GB      (NUM_PSECTORS_64GB*5)
#define NUM_PSECTORS_384GB      (NUM_PSECTORS_128GB*3)
#define NUM_PSECTORS_512GB      (NUM_PSECTORS_256GB*2)

#define BYTES_PER_PAGE          (BYTES_PER_SECTOR * SECTORS_PER_PAGE)
#define BYTES_PER_PAGE_EXT      ((BYTES_PER_PHYPAGE + SPARE_PER_PHYPAGE) * PHYPAGES_PER_PAGE)
#define BYTES_PER_PHYPAGE       (BYTES_PER_SECTOR * SECTORS_PER_PHYPAGE)
#define BYTES_PER_VBLK          (BYTES_PER_SECTOR * SECTORS_PER_VBLK)
#define BYTES_PER_BANK          ((UINT64) BYTES_PER_PAGE * PAGES_PER_BANK)
#define BYTES_PER_SMALL_PAGE    (BYTES_PER_PHYPAGE * CHN_WIDTH)
#if OPTION_2_PLANE
#define PHYPAGES_PER_PAGE       (CHN_WIDTH * NUM_PLANES)
#else
#define PHYPAGES_PER_PAGE       CHN_WIDTH
#endif
#define SECTORS_PER_PAGE        (SECTORS_PER_PHYPAGE * PHYPAGES_PER_PAGE)
#define SECTORS_PER_SMALL_PAGE  (SECTORS_PER_PHYPAGE * CHN_WIDTH)
#define SECTORS_PER_VBLK        (SECTORS_PER_PAGE * PAGES_PER_VBLK)
#define SECTORS_PER_BANK        (SECTORS_PER_PAGE * PAGES_PER_BANK)
#define PAGES_PER_BANK          (PAGES_PER_VBLK * VBLKS_PER_BANK)
#define PAGES_PER_BLK           (PAGES_PER_VBLK)
#if OPTION_2_PLANE
#define VBLKS_PER_BANK          (PBLKS_PER_BANK / NUM_PLANES)
#define SPARE_VBLKS_PER_BANK    (SPARE_PBLKS_PER_BANK / NUM_PLANES)
#else
#define VBLKS_PER_BANK          PBLKS_PER_BANK
#define SPARE_VBLKS_PER_BANK    SPARE_PBLKS_PER_BANK
#endif

#define NUM_VBLKS               (VBLKS_PER_BANK * NUM_BANKS)
#define NUM_VPAGES              (PAGES_PER_VBLK * NUM_VBLKS)
#define NUM_PSECTORS            (SECTORS_PER_VBLK * ((VBLKS_PER_BANK - SPARE_VBLKS_PER_BANK) * NUM_BANKS))
#define NUM_LPAGES              ((NUM_LSECTORS + SECTORS_PER_PAGE - 1) / SECTORS_PER_PAGE)

#define ROWS_PER_PBLK           PAGES_PER_VBLK
#define ROWS_PER_BANK           (ROWS_PER_PBLK * PBLKS_PER_BANK)
```

또한, 코드 내에서 사용되는 여러 변수들에 대한 정의를 담고 있는 "misc (miscellaneous)"도 역시 적혀있습니다.

다른 Header들도 여기서 Include되고 있습니다.

특히, "NUM_LSECTORS"는 Logical Sector의 개수를 정의하며, 실제 Physical Sector의 값과 차이를 주는 Over-provisioning이 가능하도록 해줍니다.

Logical Sector의 개수를 알려주어 실제로 OS에서 사용 가능한 메모리의 크기를 알려줍니다.

0x10000 = 65536 개의 Sectors

1개의 Sector는 512-Byte 크기이며,

전체 크기는 32MB입니다.

```
#define FLASH_ID_BYTES          5

// 4 byte ECC parity is appended to the end of every 128 byte data
// The amount of DRAM space that you can use is reduced.
#define DRAM_ECC_UNIT           128

#ifdef TRUE
#undef TRUE
#endif

#ifdef FALSE
#undef FALSE
#endif

#ifdef NULL
#undef NULL
#endif


#define TRUE        1
#define FALSE       0
#define NULL        0
#define OK          TRUE
#define FAIL        FALSE
#define INVALID     0xABABABAB // use ftl_faster
#define INVALID8    ((UINT8) -1)
#define INVALID16   ((UINT16) -1)
#define INVALID32   ((UINT32) -1)

typedef unsigned char       BOOL8;
typedef unsigned short      BOOL16;
typedef unsigned int        BOOL32;
typedef unsigned char       UINT8;
typedef unsigned short      UINT16;
typedef unsigned int        UINT32;
typedef unsigned long long  UINT64;

#define MIN(X, Y)               ((X) > (Y) ? (Y) : (X))
#define MAX(X, Y)               ((X) > (Y) ? (X) : (Y))

void delay(UINT32 const count);

#include "flash.h"
#include "sata.h"
#include "sata_cmd.h"
#include "sata_registers.h"
#include "mem_util.h"
#include "target.h"
#include "bank.h"
```

```
#define SCAN_LIST_SIZE              BYTES_PER_SMALL_PAGE
#define SCAN_LIST_ITEMS             ((SCAN_LIST_SIZE / sizeof(UINT16)) - 1)

typedef struct
{
    UINT16  num_entries;
    UINT16  list[SCAN_LIST_ITEMS];
}scan_list_t;

// original
// #define NUM_LSECTORS (21168 + ((NUM_PSECTORS) / 2097152 * 1953504)) // 125045424, 9172304(provisioning ratio: 7.3%)

// #define NUM_LSECTORS (NUM_PSECTORS  / 100 * 86) // 14% provisioning
#define NUM_LSECTORS    (0x10000)//(NUM_PSECTORS  / 100 * 93) // 7% provisioning

#include "ftl.h"
#include "misc.h"

#ifndef PROGRAM_INSTALLER
#include "uart.h"
#endif
```

“ftl.h”

```
#define NUM_RW_BUFFERS      ((DRAM_SIZE - DRAM_BYTES_OTHER) / BYTES_PER_PAGE - 1)
#define NUM_RD_BUFFERS      (((NUM_RW_BUFFERS / 8) + NUM_BANKS - 1) / NUM_BANKS * NUM_BANKS)
#define NUM_WR_BUFFERS      (NUM_RW_BUFFERS - NUM_RD_BUFFERS)
#define NUM_COPY_BUFFERS    NUM_BANKS_MAX
#define NUM_HIL_BUFFERS     1

#define DRAM_BYTES_OTHER    ((NUM_COPY_BUFFERS + NUM_HIL_BUFFERS) * BYTES_PER_PAGE + (0x2000000))

#define WR_BUF_PTR(BUF_ID)  (WR_BUF_ADDR + ((UINT32)(BUF_ID)) * BYTES_PER_PAGE)
#define WR_BUF_ID(BUF_PTR)  ((((UINT32)BUF_PTR) - WR_BUF_ADDR) / BYTES_PER_PAGE)
#define RD_BUF_PTR(BUF_ID)  (RD_BUF_ADDR + ((UINT32)(BUF_ID)) * BYTES_PER_PAGE)
#define RD_BUF_ID(BUF_PTR)  ((((UINT32)BUF_PTR) - RD_BUF_ADDR) / BYTES_PER_PAGE)

#define _COPY_BUF(RBANK)    (COPY_BUF_ADDR + (RBANK) * BYTES_PER_PAGE)
#define COPY_BUF(BANK)      _COPY_BUF(REAL_BANK(BANK))
```

Read&Write, Read, Write, Copy, Host Interface Layer Buffer의 사이즈가 정의되어 있습니다.

(단위는 Page이며, Read&Write 버퍼는 Read Buffer와 Write Buffer의 사이즈를 구하는 데에 쓰입니다..)

“DRAM_BYTES_OTHER”에서는 Copy Buffer와 HIL Buffer 이외의 DRAM 영역을 Read Buffer와 Write Buffer에 할당하도록 하는데에 쓰입니다.

여기에 0x2000000 (=32MB)만큼의 값을 더해주어 저희가 사용할 32MB의 공간을 확보하였습니다.

```
#define RD_BUF_ADDR         DRAM_BASE                                   // base address of SATA read buffers
#define RD_BUF_BYTES        (NUM_RD_BUFFERS * BYTES_PER_PAGE)

#define WR_BUF_ADDR         (RD_BUF_ADDR + RD_BUF_BYTES)                // base address of SATA write buffers
#define WR_BUF_BYTES        (NUM_WR_BUFFERS * BYTES_PER_PAGE)

#define COPY_BUF_ADDR       (WR_BUF_ADDR + WR_BUF_BYTES)                // base address of flash copy buffers
#define COPY_BUF_BYTES      (NUM_COPY_BUFFERS * BYTES_PER_PAGE)

#define HIL_BUF_ADDR        (COPY_BUF_ADDR + COPY_BUF_BYTES)                // a buffer dedicated to HIL internal purpose
#define HIL_BUF_BYTES       (NUM_HIL_BUFFERS * BYTES_PER_PAGE)

#define DRAM_NEW            (HIL_BUF_ADDR + HIL_BUF_BYTES)
```

각 Buffer들의 주소 값과 사이즈가 정의되어 있으며,

“DRAM_NEW”는 제가 정의한 변수로 저희가 사용할 32MB의 범위를 정의하고 있습니다.

```
void ftl_open(void);
void ftl_read(UINT32 const lba, UINT32 const num_sectors);
void ftl_write(UINT32 const lba, UINT32 const num_sectors);
void read_dram(UINT32 const lpn, UINT32 const sect_offset, UINT32 const num_sectors_to_read);
void write_dram(UINT32 const lpn, UINT32 const sect_offset, UINT32 const num_sectors_to_write);
void ftl_test_write(UINT32 const lba, UINT32 const num_sectors);
void ftl_flush(void);
void ftl_isr(void);
```

“ftl.c”에서 사용되는 함수들이 선언되어 있으며,

“read_dram” 과 “write_dram” 함수는 DRAM에 데이터를 저장하기 위해 제가 선언한 함수입니다.

“ftl.c”

```
UINT32 g_ftl_read_buf_id;
UINT32 g_ftl_write_buf_id;
```

이 두 변수는 Read Buffer 와 Write Buffer의 Pointer가 어느 Index에 있는 지를 알려주는 변수입니다.

```
void ftl_read(UINT32 const lba, UINT32 const total_sectors)
{
    UINT32 num_sectors_to_read;

    UINT32 lpn              = lba / SECTORS_PER_PAGE;   // logical page address
    UINT32 sect_offset      = lba % SECTORS_PER_PAGE;   // sector offset within the page
    UINT32 sectors_remain   = total_sectors;

    while (sectors_remain != 0) // one page per iteration
    {
        if (sect_offset + sectors_remain < SECTORS_PER_PAGE)
        {
            num_sectors_to_read = sectors_remain;
        }
        else
        {
            num_sectors_to_read = SECTORS_PER_PAGE - sect_offset;
        }

        UINT32 next_read_buf_id = (g_ftl_read_buf_id + 1) % NUM_RD_BUFFERS;

        read_dram(lpn, sect_offset, num_sectors_to_read);

        while (next_read_buf_id == GETREG(SATA_RBUF_PTR));  // wait if the read buffer is full (slow host)

        SETREG(BM_STACK_RDSET, next_read_buf_id);   // change bm_read_limit
        SETREG(BM_STACK_RESET, 0x02);               // change bm_read_limit

        g_ftl_read_buf_id = next_read_buf_id;

        sect_offset = 0;
        sectors_remain -= num_sectors_to_read;
        lpn++;
    }
}
```

“ftl_read”함수는 “lba (Logical Block Address)”와 “total_sectors”를 입력 받아 “lba”를 기준으로 “total_secters”의 개수만큼 데이터를 읽습니다.

“lba”는 Sector단위이며, Buffer는 Page 단위이기 때문에 “sect_offset” 변수를 만들어 Page 단위 Read를 할 때, 데이터가 끊기지 않도록 사용합니다.

(단, 이 곳의 Code에서는 DRAM에 데이터를 저장하기 때문에 Page단위 Read를 구현하지 않았습니다.)

“lpn (Logical Page Number)”는 Logical Page 값을 갖고 있습니다.

If문에서는 앞으로 읽어야할 데이터가 Page 단위일 때와 아닐 때를 경우를 나눠주었습니다.

“read_dram”을 통하여 DRAM에서 “mem_copy()”를 통해 데이터를 읽어들여 Buffer에 넣어줍니다.

While문에서는 현재 ftl_read_ptr 다음 Index가 비어 있는지 sata_read_ptr와 비교하여 판단하며, 비어있지 않았을 경우 계속 기다려줍니다.

```
void read_dram(UINT32 const lpn, UINT32 const sect_offset, UINT32 const num_sectors_to_read)
{
    // Read from DRAM NEW & Write to SATA Read Buffer.
    UINT32 left_bytes = num_sectors_to_read * BYTES_PER_SECTOR;
    UINT32 offset_bytes = sect_offset * BYTES_PER_SECTOR;
    UINT32 start_byte = lpn * BYTES_PER_PAGE;

    mem_copy(RD_BUF_PTR(g_ftl_read_buf_id) + offset_bytes, DRAM_NEW + start_byte + offset_bytes, left_bytes);
}
```

"left_bytes"는 Read Buffer에 넣어줘야 하는 Sector들의 Byte값을 가지고 있습니다.

"offset_bytes"는 Page 단위인 Read Buffer를 위하여 이 값 만큼 Shift를 해주어 데이터를 읽고, 입력합니다.

"start_byte"는 Logical Page Number의 Byte값을 가지고 있으며, 읽어 들일 위치의 Byte값을 나타냅니다.

"RD_BUF_PTR()"은 입력한 Read Buffer ID의 Address를 반환합니다.

"mem_copy()"는 Source 주소부터 Byte 개수만큼 복사하여 Destination에 입력하여줍니다.

```
void ftl_write(UINT32 const lba, UINT32 const total_sectors)
{
    UINT32 num_sectors_to_write;

    UINT32 lpn              = lba / SECTORS_PER_PAGE;   // logical page address
    UINT32 sect_offset      = lba % SECTORS_PER_PAGE;
    UINT32 remain_sectors   = total_sectors;

    while (remain_sectors != 0)
    {
        if (sect_offset + remain_sectors >= SECTORS_PER_PAGE)
        {
            num_sectors_to_write = SECTORS_PER_PAGE - sect_offset;
        }
        else
        {
            num_sectors_to_write = remain_sectors;
        }

        while (g_ftl_write_buf_id == GETREG(SATA_WBUF_PTR));    // bm_write_limit should not outpace SATA_WBUF_PTR

        write_dram(lpn, sect_offset, num_sectors_to_write);

        g_ftl_write_buf_id = (g_ftl_write_buf_id + 1) % NUM_WR_BUFFERS;     // Circular buffer

        SETREG(BM_STACK_WRSET, g_ftl_write_buf_id); // change bm_write_limit
        SETREG(BM_STACK_RESET, 0x01);               // change bm_write_limit

        sect_offset = 0;
        remain_sectors -= num_sectors_to_write;
    }
}
```

"ftl_write"는 "ftl_read"와 비슷하게 동작하지만, Write Buffer로부터 데이터를 읽어 DRAM에 저장한다는 것이 다릅니다.

"ftl_read"와 다르게 While문 이후에 "write_dram"을 넣은 이유는 ftl_write_ptr는 sata_write_ptr가 데이터를 Write Buffer에 입력한 뒤에 입력된 데이트를 읽어야 하므로 sata_write_ptr가 데이터를 쓰고 있을 경우 기다린 뒤, 데이터를 DRAM에 옮겨주어야하기 때문입니다.

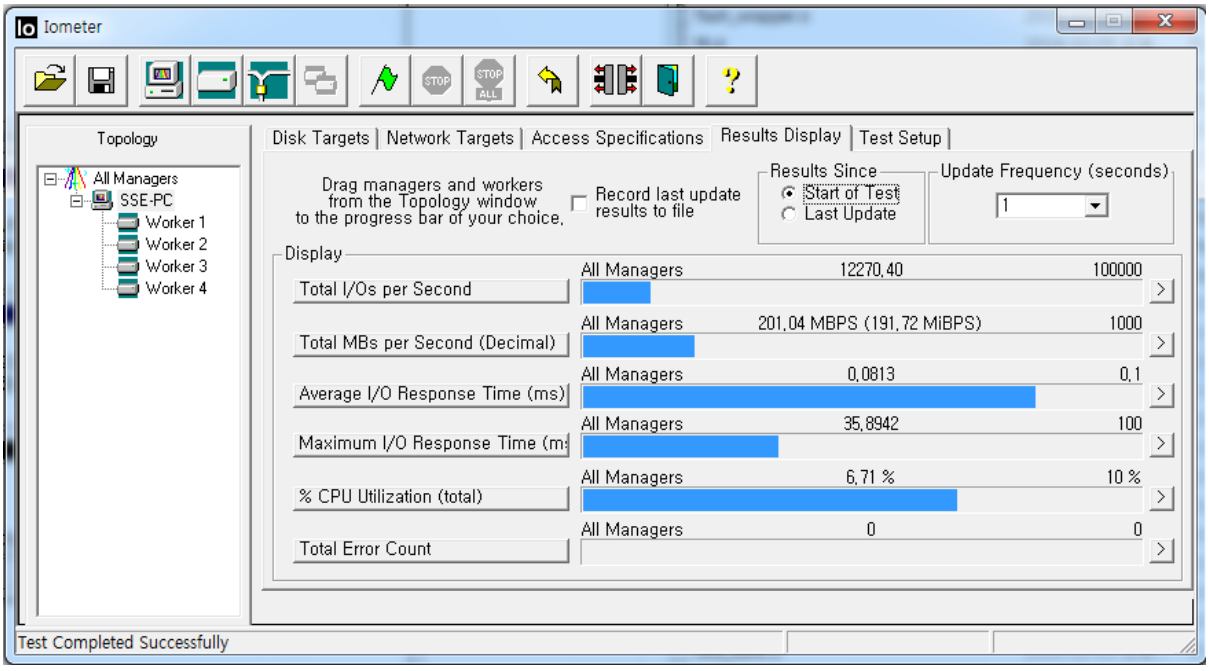
```
void write_dram(UINT32 const lpn, UINT32 const sect_offset, UINT32 const num_sectors_to_write)
{
    // Read from SATA Write Buffer & Write to Empty Space.
    UINT32 left_bytes = num_sectors_to_write * BYTES_PER_SECTOR;
    UINT32 offset_bytes = sect_offset * BYTES_PER_SECTOR;
    UINT32 start_byte = lpn * BYTES_PER_PAGE;

    mem_copy(DRAM_NEW + start_byte + offset_bytes, WR_BUF_PTR(g_ftl_write_buf_id) + offset_bytes, left_bytes);
}
```

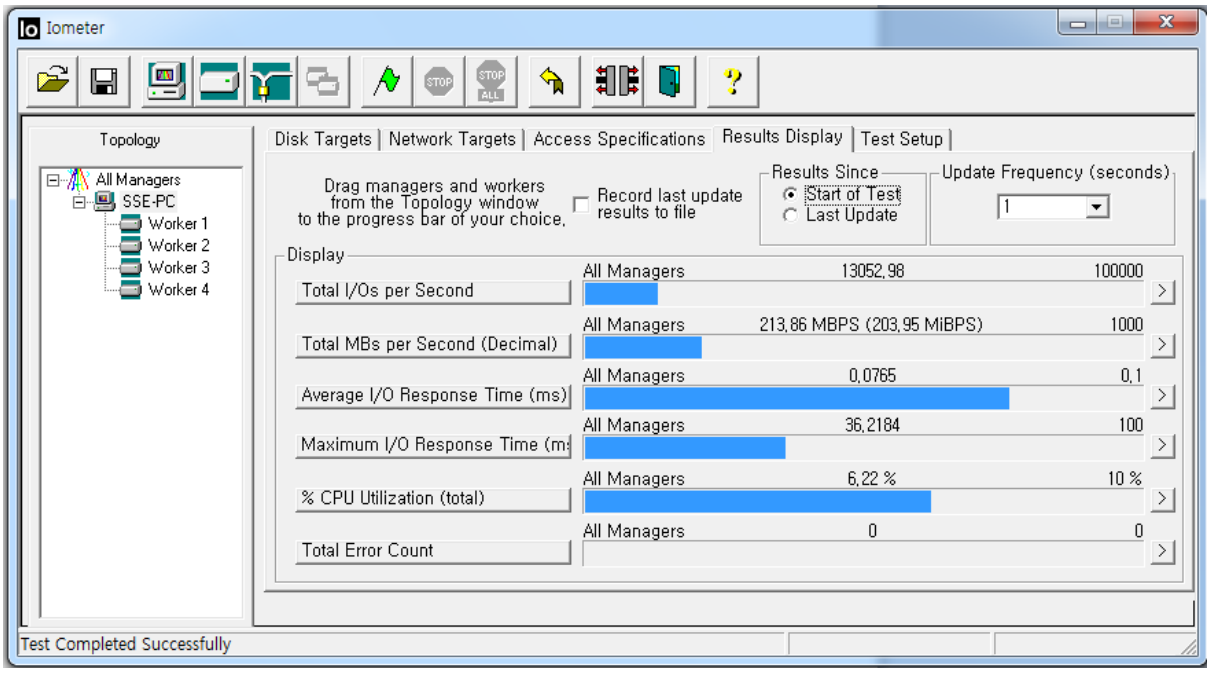
"read_dram"과 거의 유사하게 작성하였으며, Write Buffer가 Source가 되고 RAM이 Destination이 된다는 점에서만 다릅니다.

Dummy FTL

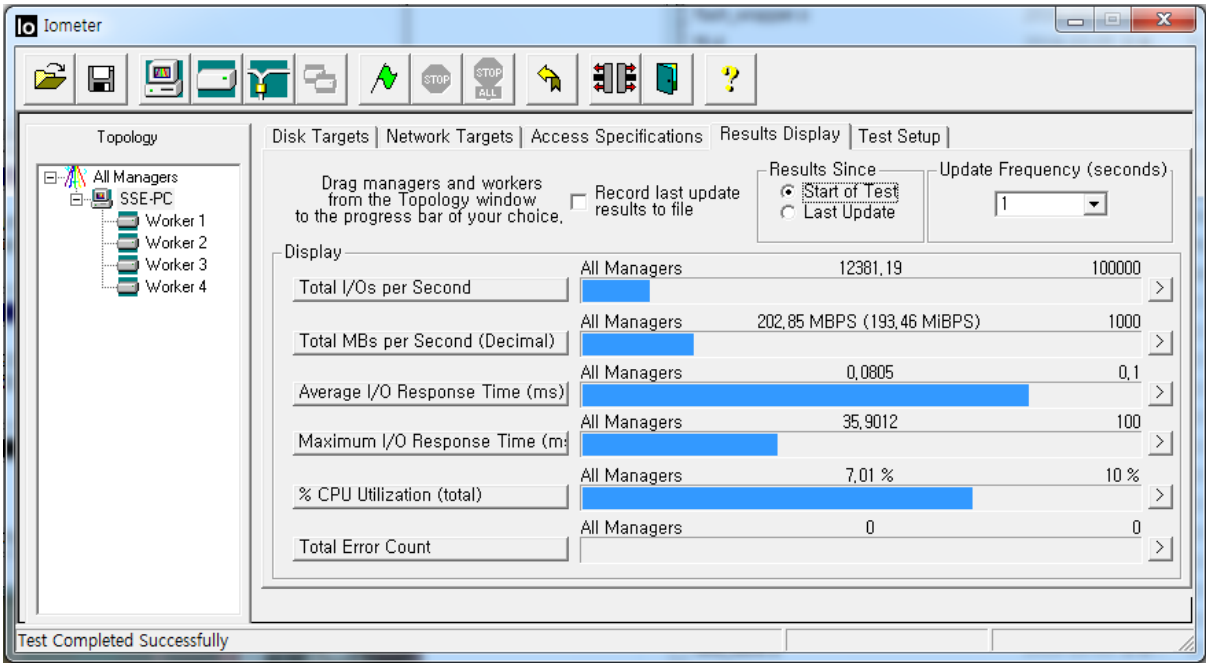
Sequential Write 16KB



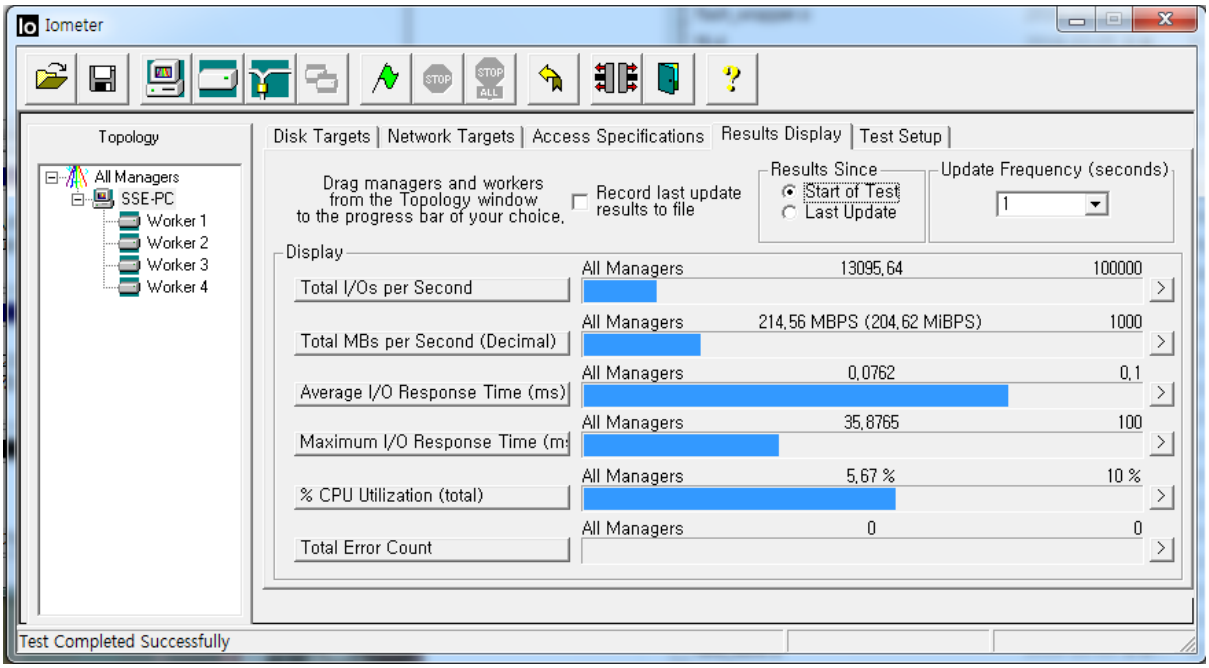
Sequential Read 16KB



Random Write 16KB

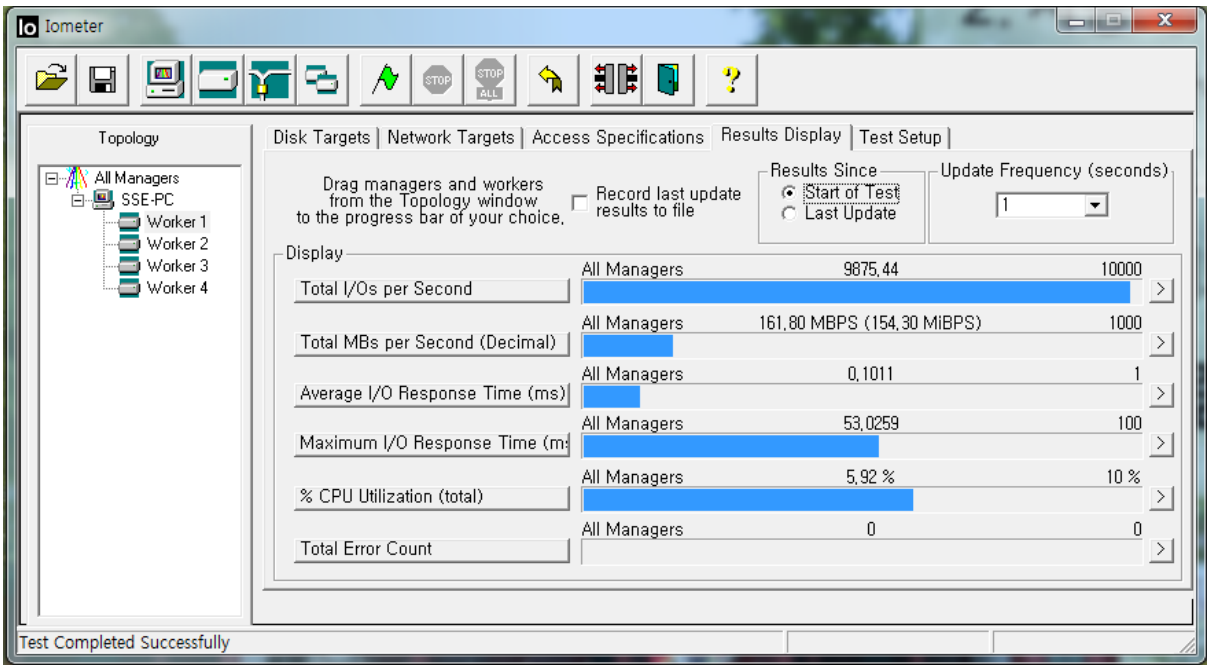


Random Read 16KB

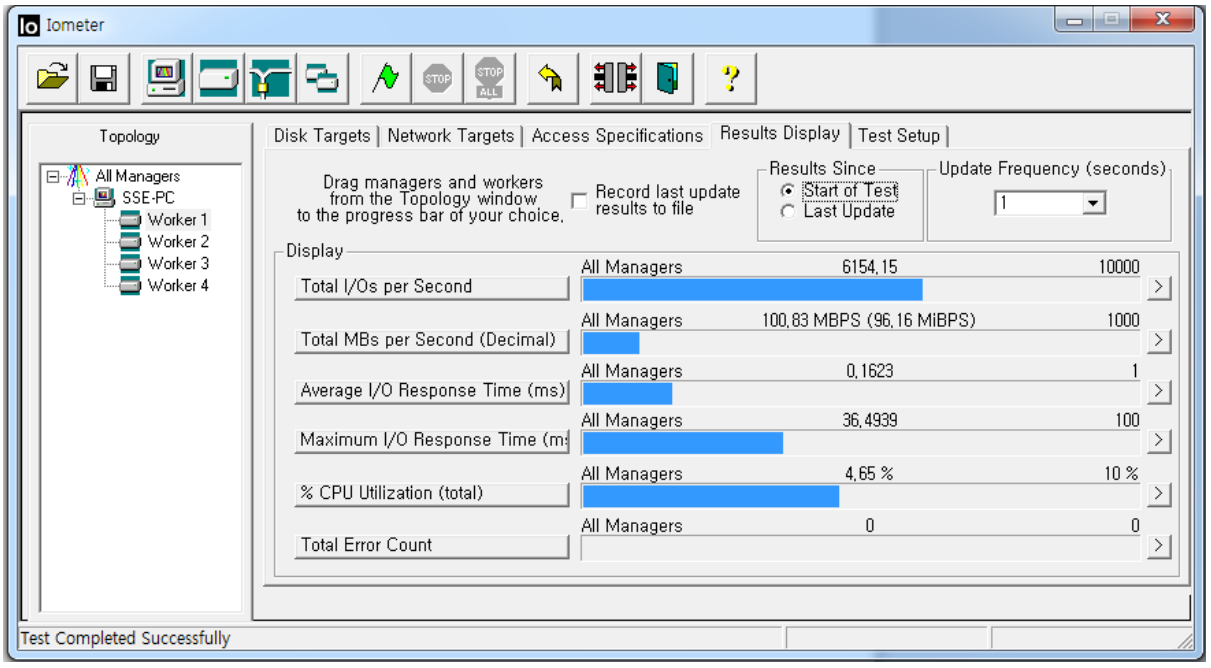


RAM FTL

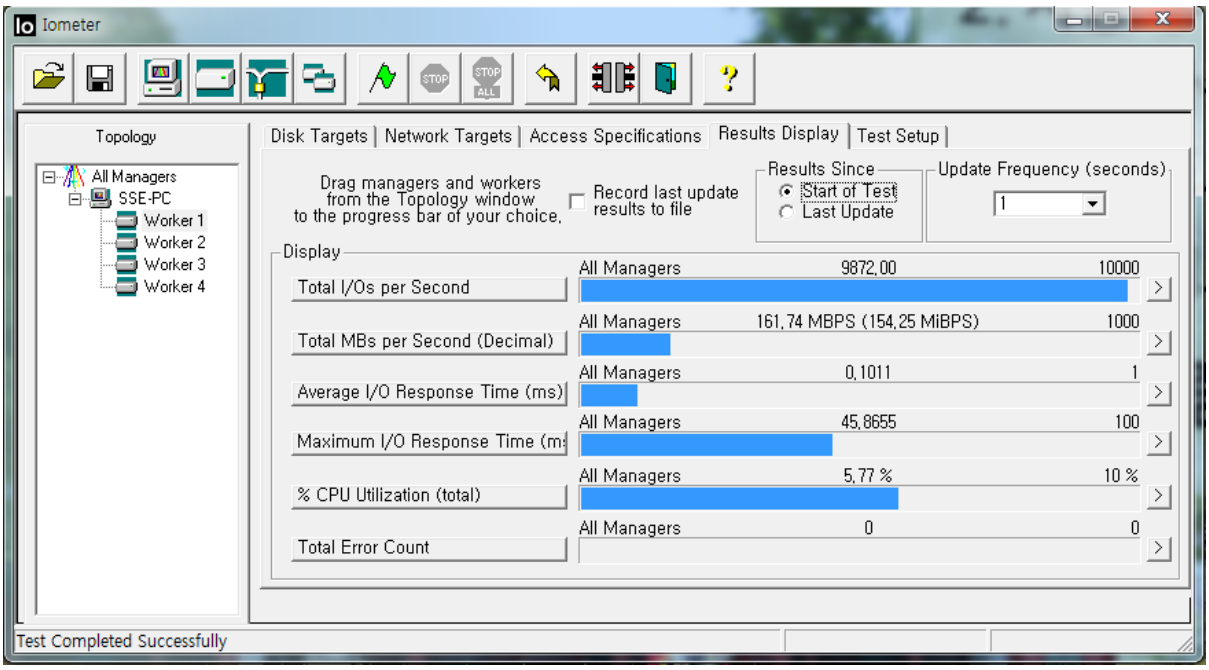
Sequential Write 16KB



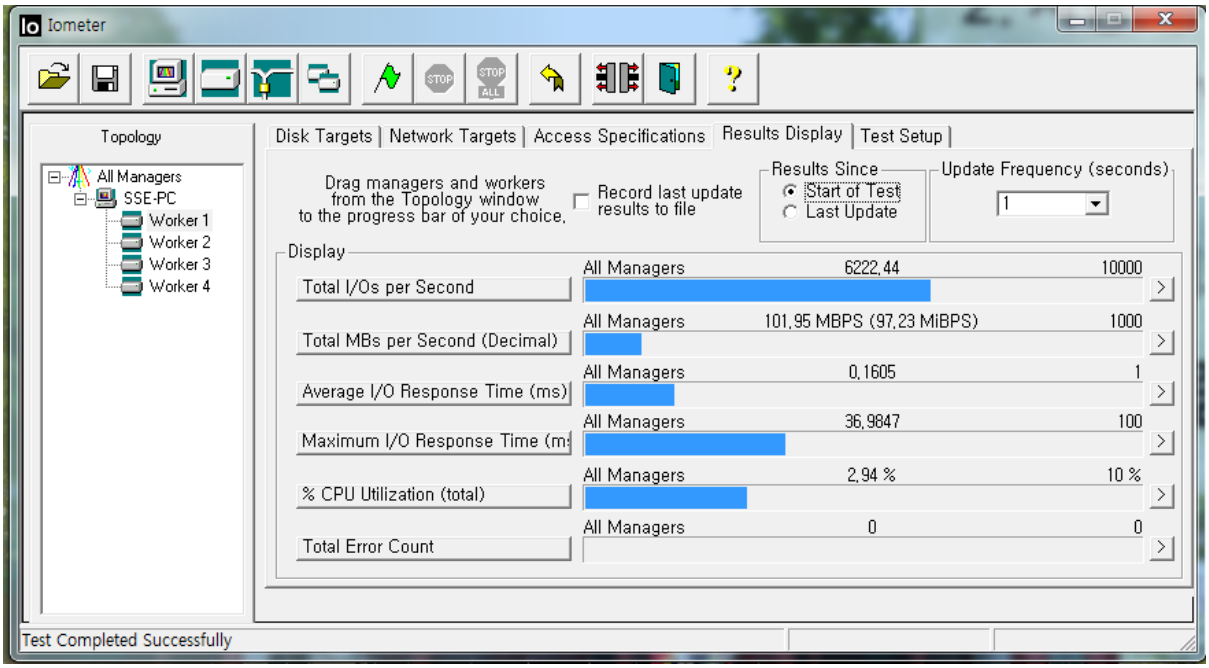
Sequential Read 16KB



Random Write 16KB



Random Read 16KB



Comparison (Dummy FTL VS RAM FTL)

먼저, 두 FTL의 Performace를 각각 분석하도록 하겠습니다.

Dummy FTL은

Sequential과 Random 모두 Read가 Write보다 Total I/Os per Second와 Total MBs per Second가 높았고,

Average I/O Response Time와 Maximum I/O Response Time이 짧았습니다.

하지만, Sequential과 Random 모두 Write가 Read보다 % CPU Utilization이 높았습니다.

반면, RAM FTL은

특이하게도 Sequential과 Random 모두 Write가 Read보다 Total I/Os per Second, Total MBs per Second가 높았고, Average I/O Response Time와 Maximum I/O Response Time이 짧았으며, % CPU Utilization이 높았습니다.

Write가 Read보다 Performance가 좋은 것은 Write의 경우, 상대적으로 빠른 SATA Write Pointer가 Buffer에 먼저 정보를 적은 뒤 상대적으로 더 느린 FTL Write Pointer가 따라가기 때문에 전체적인 Write Operation이 일찍 끝났다고 OS에서 인식하게 된다고 생각합니다.

반대로, Read의 경우, 상대적으로 더 느린 FTL Read Pointer가 먼저 Read Buffer에 정보를 적은 뒤, 상대적으로 빠른 SATA Read Pointer가 OS로 정보를 전달하기 때문에 OS에서는 정보 전달이 Write보다 느리게 처리된다고 인식한다고 생각합니다.

또한, "jasmine.h"를 보면 Write Buffer와 Read Buffer가 서로 크기가 다르며, Write Buffer가 사이즈가 더 큰데 이것은 제 의견을 더 뒷받침한다고 생각합니다.

IOMeter에서는 16KB 단위로 16MB를 입력하기 때문에 Write Buffer가 전부 가득 차지 않을 것이라고 생각이 됩니다.

만약, Write Buffer가 가득 차는 양의 I/O가 필요하다면, Read와 Write의 속도가 비슷하거나 Read가 더 빠를 것이라고 생각합니다.

이제, Dummy FTL과 RAM FTL를 비교하도록 하겠습니다.

Maximum I/O Response Time을 제외하면, Dummy FTL이 RAM FTL보다 Performance가 좋다는 것을 알 수 있습니다.

상식적으로는 DRAM이 SSD보다 빨라야 한다고 생각이 되었지만, 다른 결과가 나와 의아하였습니다.

제 예상으로는 Dummy FTL에서는 아무 작업을 하지 않고 Buffer에 있는 정보를 제거하지만, 저희가 작성한 RAM FTL은 실제로 정보를 쓰거나 읽기 때문에 더 느리다고 생각합니다.

Dummy FTL과 RAM FTL 모두에서 Sequential과 Random 데이터 입/출력이 의미있는 큰 차이를 나타내지는 않아습니다.

입력되는 데이터의 사이즈가 Block 사이즈보다 작아질수록 Sequential Access가 Random Access보다 빠르게 동작하지만, 전체 공간을 32MB로 작게 설정하였을 뿐 아니라 Page 사이즈 역시 16KB였으므로, 큰 차이가 나타나지 않았다고 생각됩니다.