Tutorial FTL

Sejun Kwon
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu

Contents

- NAND Flash Memory
- NAND Flash Operation
- NAND Flash Architecture
- NAND Controller
- Tutorial FTL

NAND Flash Memory

- K9LCG08U1M (Dual die)
  - Samsung 35 nm 2-bit MLC flash
  - 16 sectors per page (8 KB + 640 B)
  - 128 pages per block (1 MB + 80 KB)
  - 4096 + 56 blocks per die
  - Page read : 250 us
  - Page program : 1.3 ms
  - Block erase : 1.5 ms

NAND Flash Memory

- NAND Flash Code Information
  - http://www.samsung.com/global/business/semiconductor/support/label-code-info/code-info/memory-component

K	9	X	X	X	X	X	X	X	X	-	X	X	X	X	X	X	X
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

3. Small Classification

(SLC : Single Level Cell, MLC : Multi Level Cell, SM : SmartMedia, S/B : Small Block)

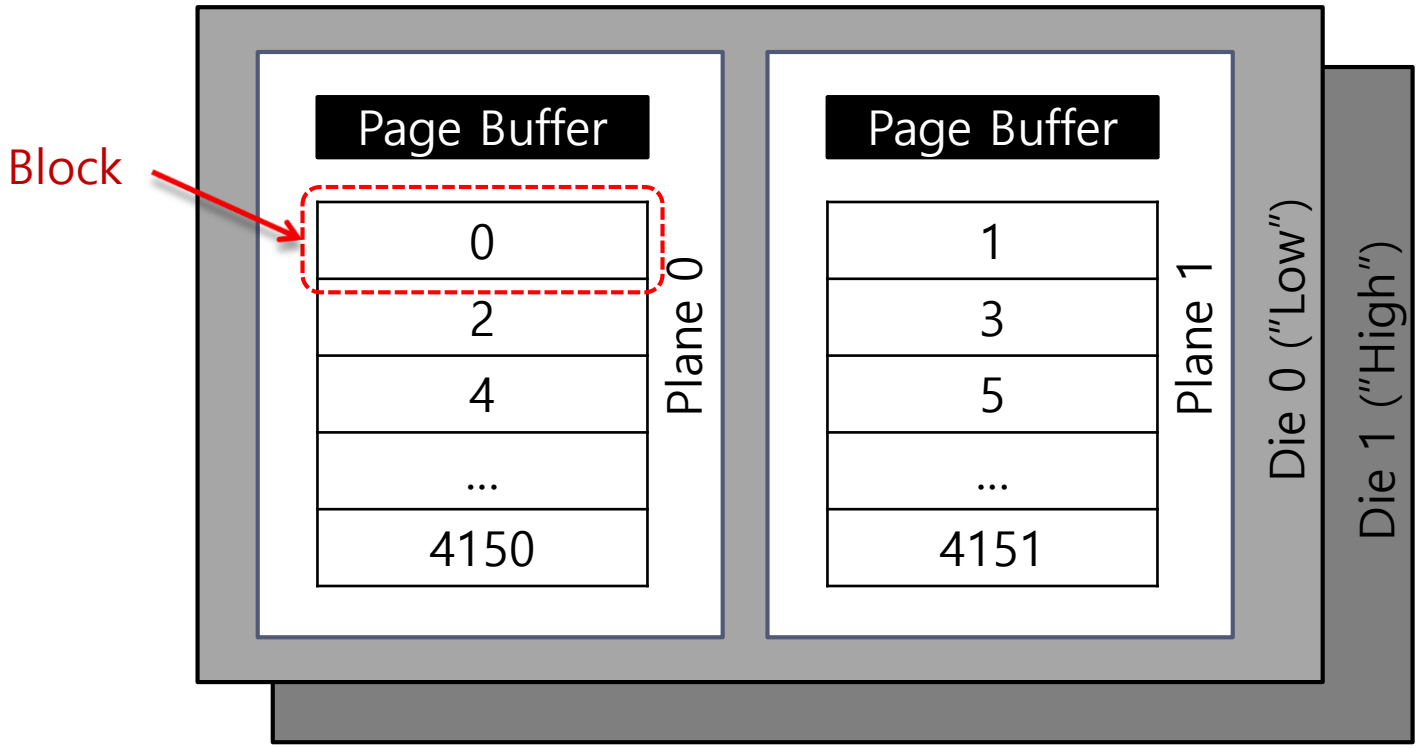
K : SLC Die Stack
L : MLC DDP
M : MLC DSP
N : SLC DSP
O : 3bit MLC ODP

4~5. Density

12 : 512M	16 : 16M	28 : 128M
32 : 32M	40 : 4M	56 : 256M
64 : 64M	80 : 8M	1G : 1G
2G : 2G	4G : 4G	8G : 8G
AG : 16G	BG : 32G	CG : 64G
DG : 128G	EG : 256G	FG : 256G
GG : 384G	HG : 512G	LG : 24G
NG : 96G	ZG : 48G	00 : NONE

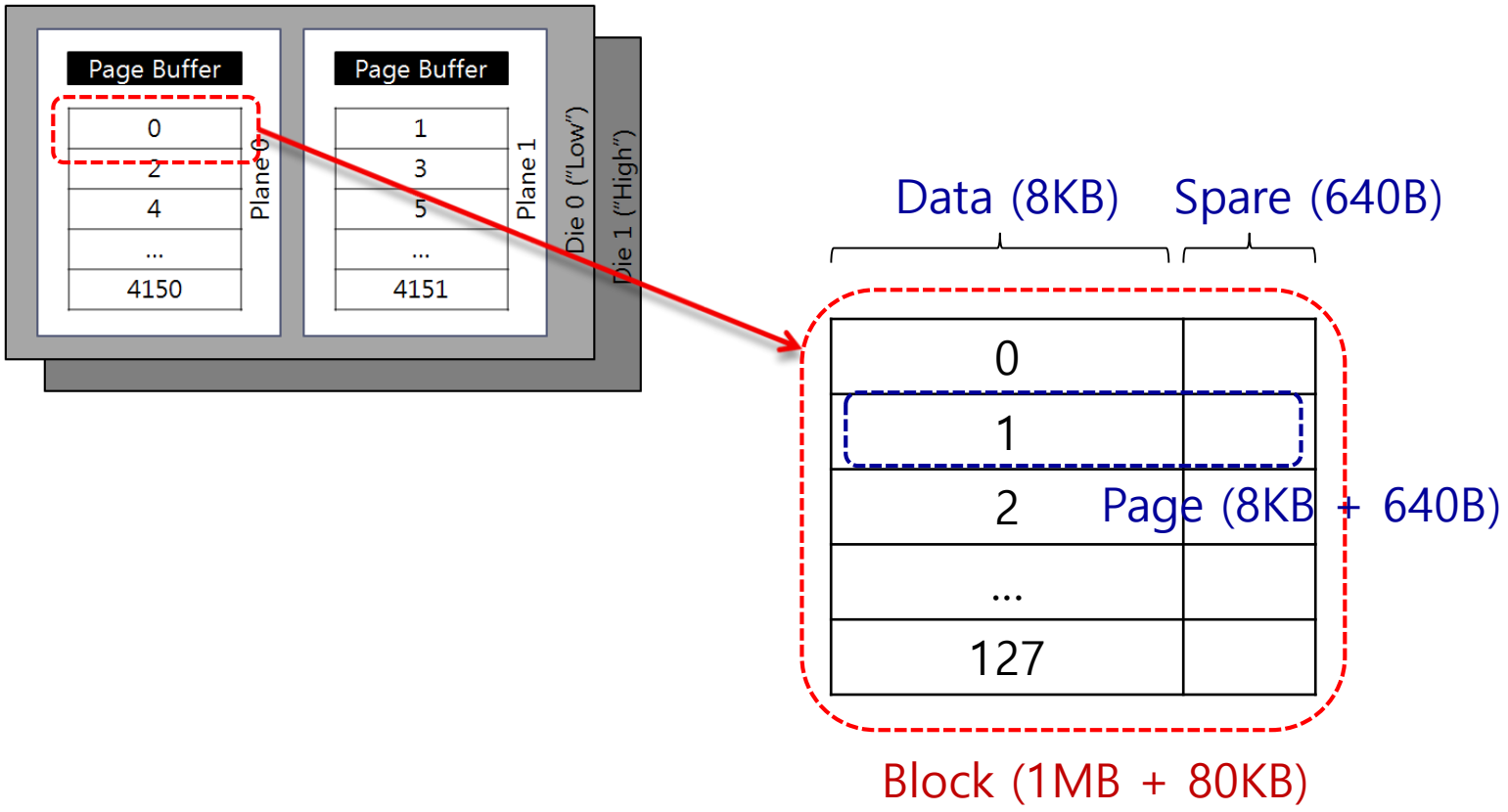
NAND Flash Memory

- NAND Flash Organization



NAND Flash Memory

- NAND Flash Organization

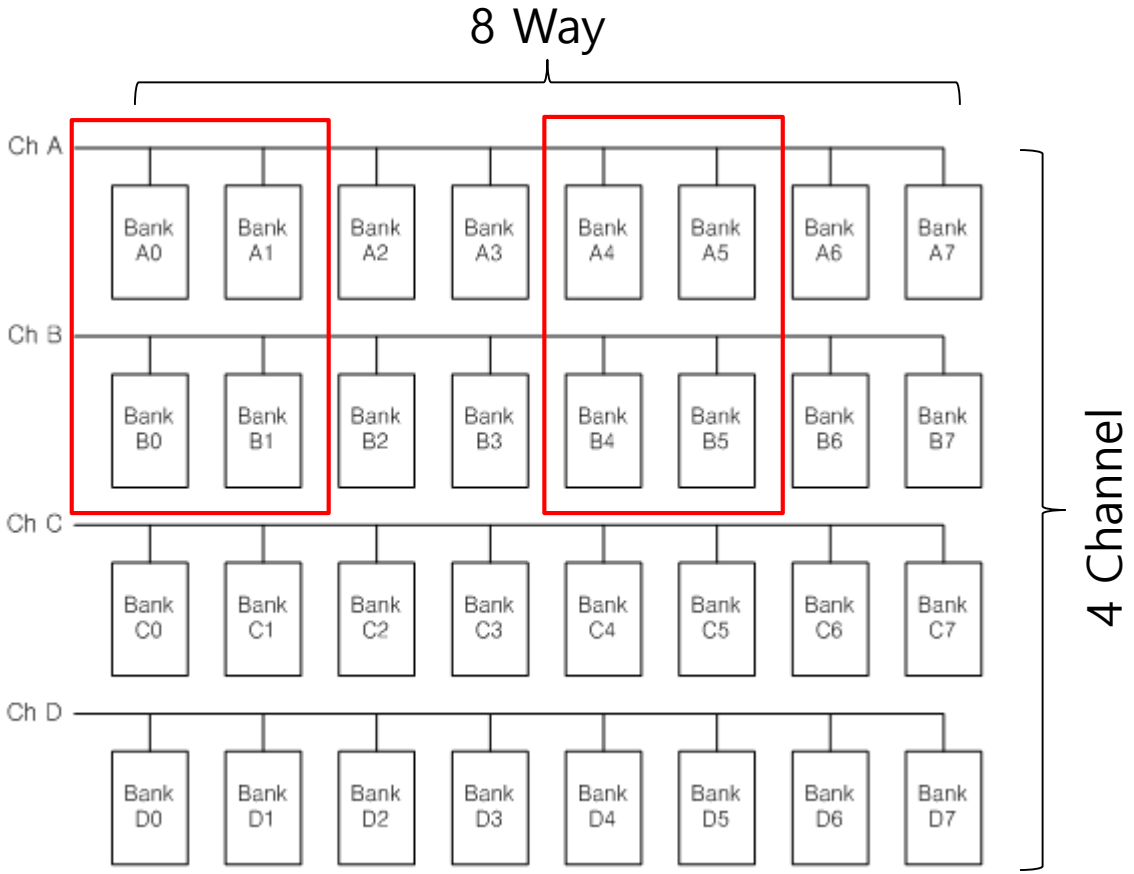


NAND Flash Operation

- Page read
  - Cell -> Page Buffer -> RAM
- Page program
  - RAM -> Page Buffer -> Cell
- Page copy-back
  - Cell (Src) -> Page Buffer -> Cell (Dst)
- Block erase

NAND Flash Configuration

- Channel / Way



NAND Flash Configuration

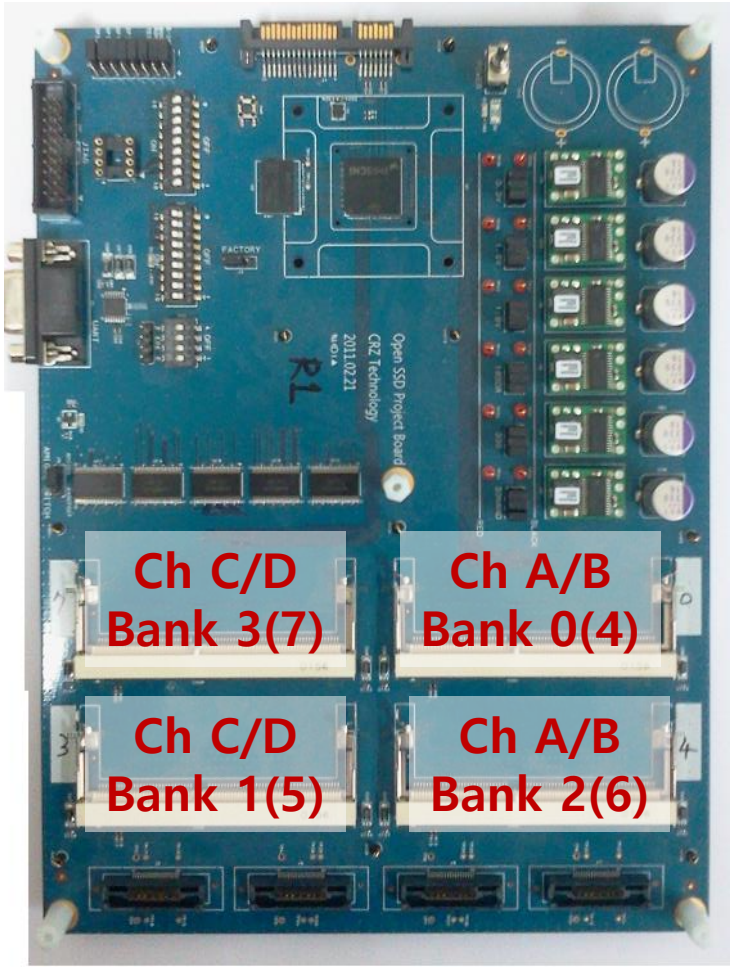
- Jasmine Firmware abstracts channel and way to “bank”

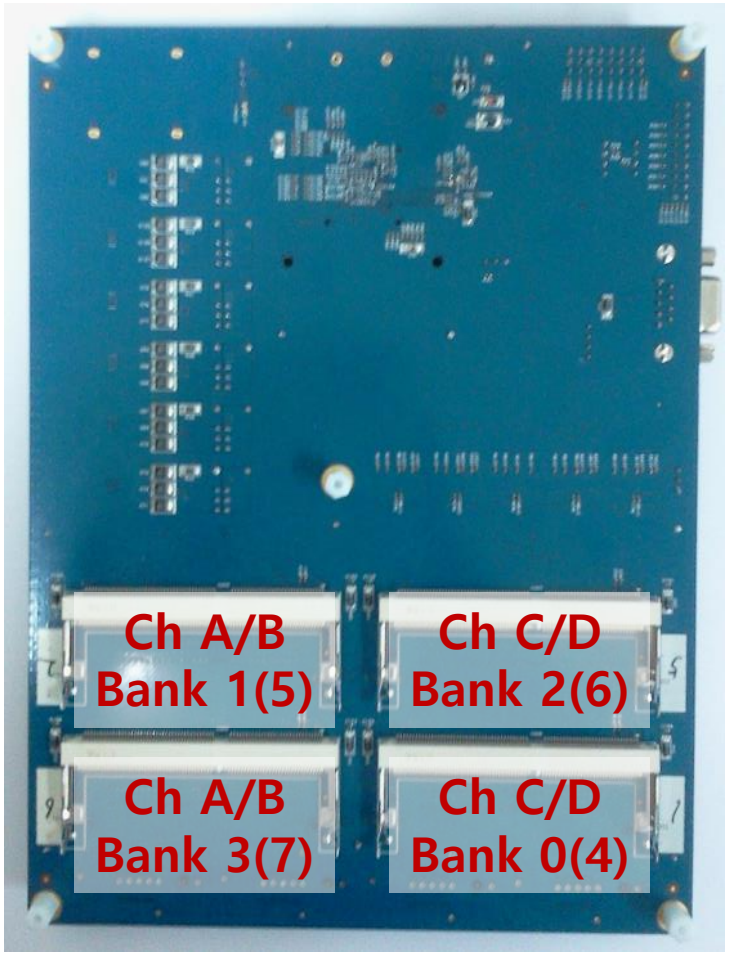
- 4way * 2 channel = 8 banks
- bank number = physical page number % number of banks
- Each bank has blocks

NAND Flash Configuration

- Banks share the same IO bus
- Each bank can perform cell operation in parallel
  - Cell to Page Buffer operation, Block erase
- Barefoot has only 4 R/B signal inputs per channel
  - A0 and A4 are tied together
  - Maximum 4 way interleaving

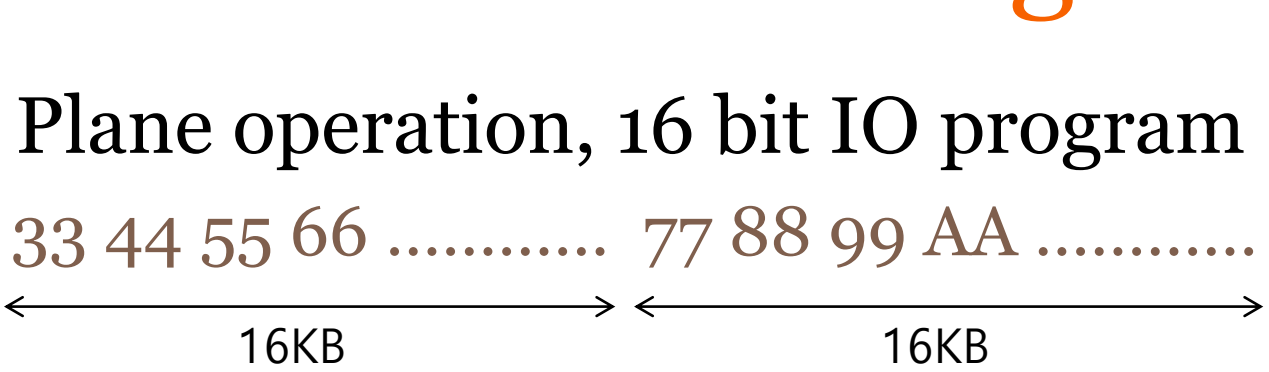
NAND Flash Configuration





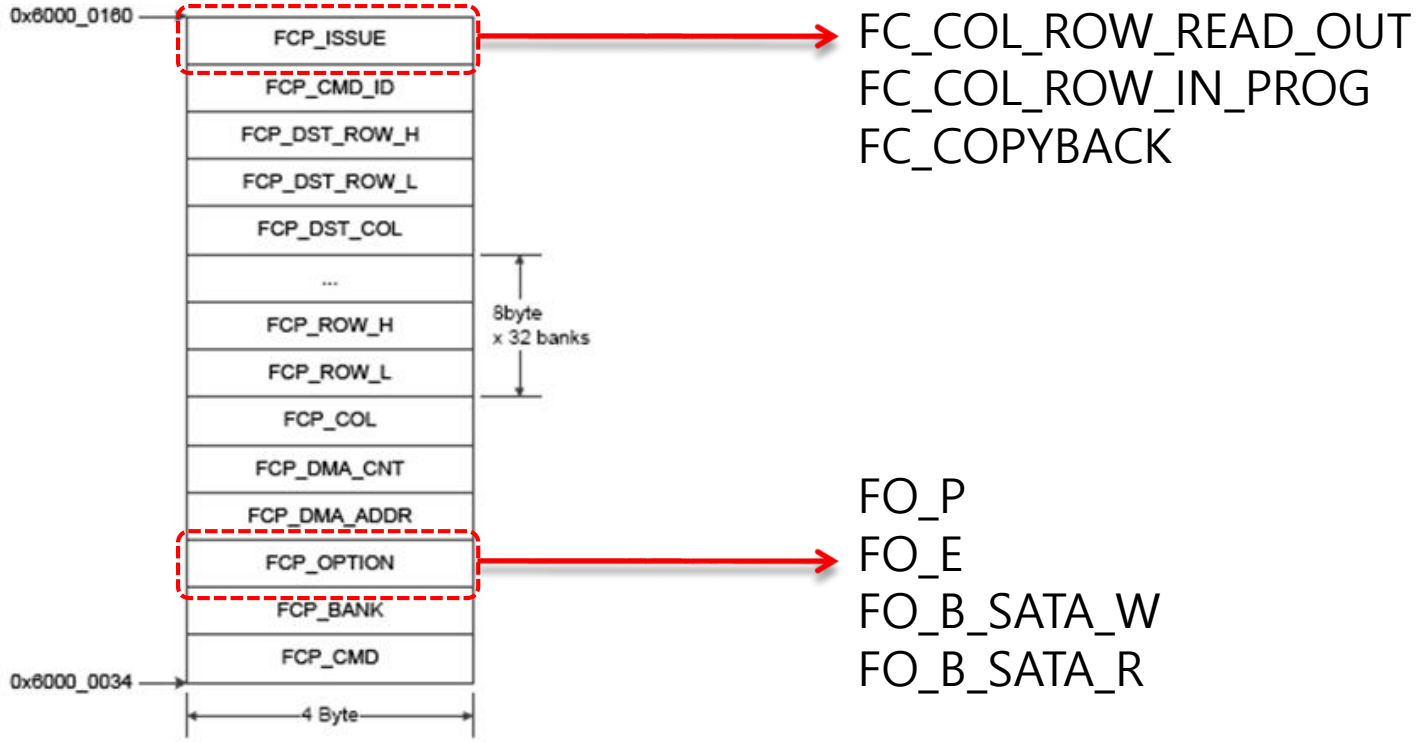
NAND Flash Configuration

- 2 Plane operation, 16 bit IO program
  - 33 44 55 66 ..... 77 88 99 AA .....



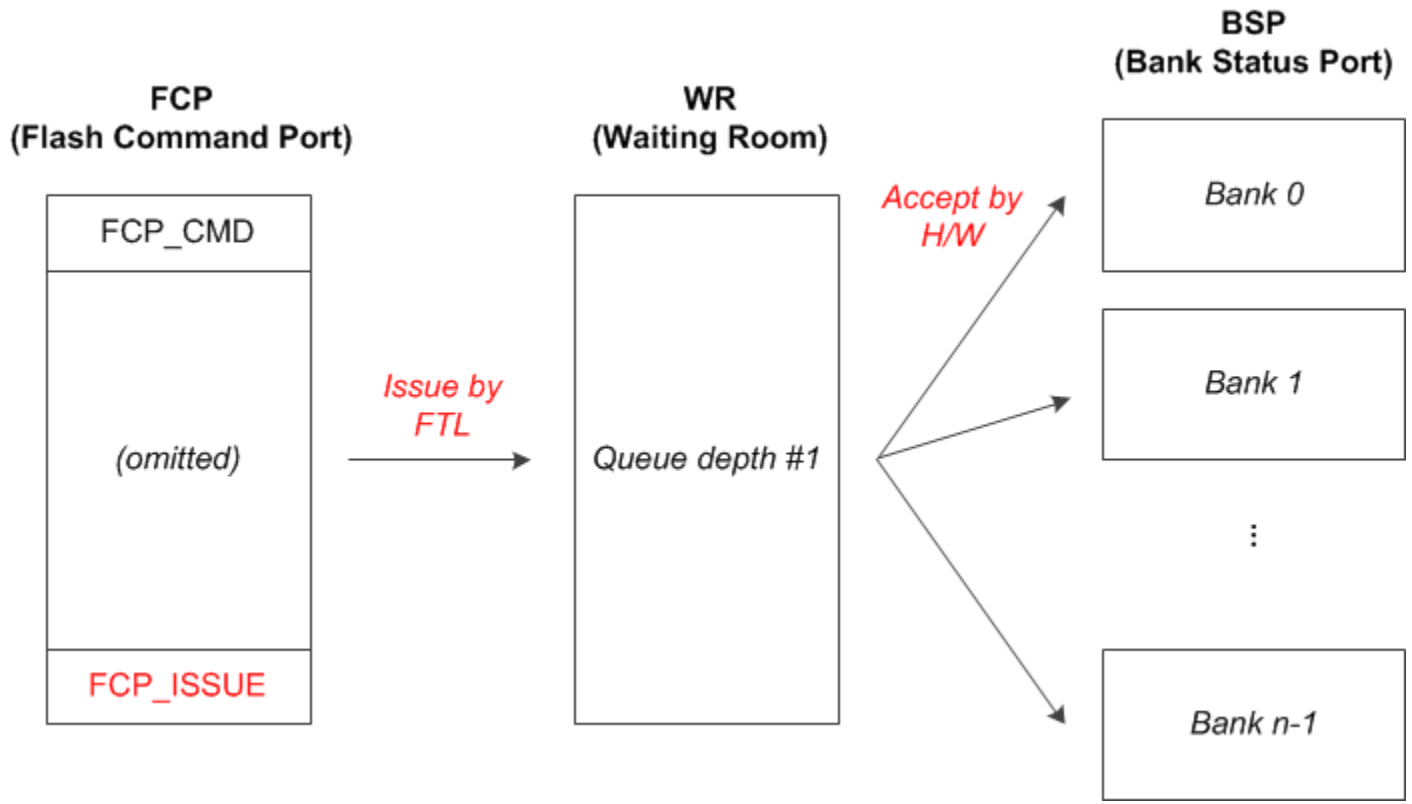
NAND Flash Controller

- To issue NAND Flash operation
  - Defined in ./include/flash.h



NAND Flash Controller

- To issue NAND Flash operation



Tutorial FTL

- ./ftl_tutorial
  - ftl.c, ftl.h

- Page Mapping FTL
  - Write data from DRAM to NAND
  - Read data from NAND to DRAM
  - But no garbage collection

Page Mapping Table

- LPN to PPN map
  - LPN: Logical Page Number
    - LPN = LBA / Sectors Per Page
  - PPN: Physical Page Number

Index	0	1	2	3	…	2097151
Value	100	256	0	INVALID	…	20000

```
static UINT32 get_physical_address(UINT32 const lpage_addr)
{
    // Page mapping table entry size is 4 byte.
    return read_dram_32(PAGE_MAP_ADDR + lpage_addr * sizeof(UINT32));
}

static void update_physical_address(UINT32 const lpage_addr, UINT32 const new_bank, UINT32 const new_row)
{
    write_dram_32(PAGE_MAP_ADDR + lpage_addr * sizeof(UINT32), new_bank * PAGES_PER_BANK + new_row);
}
```

Read Operation

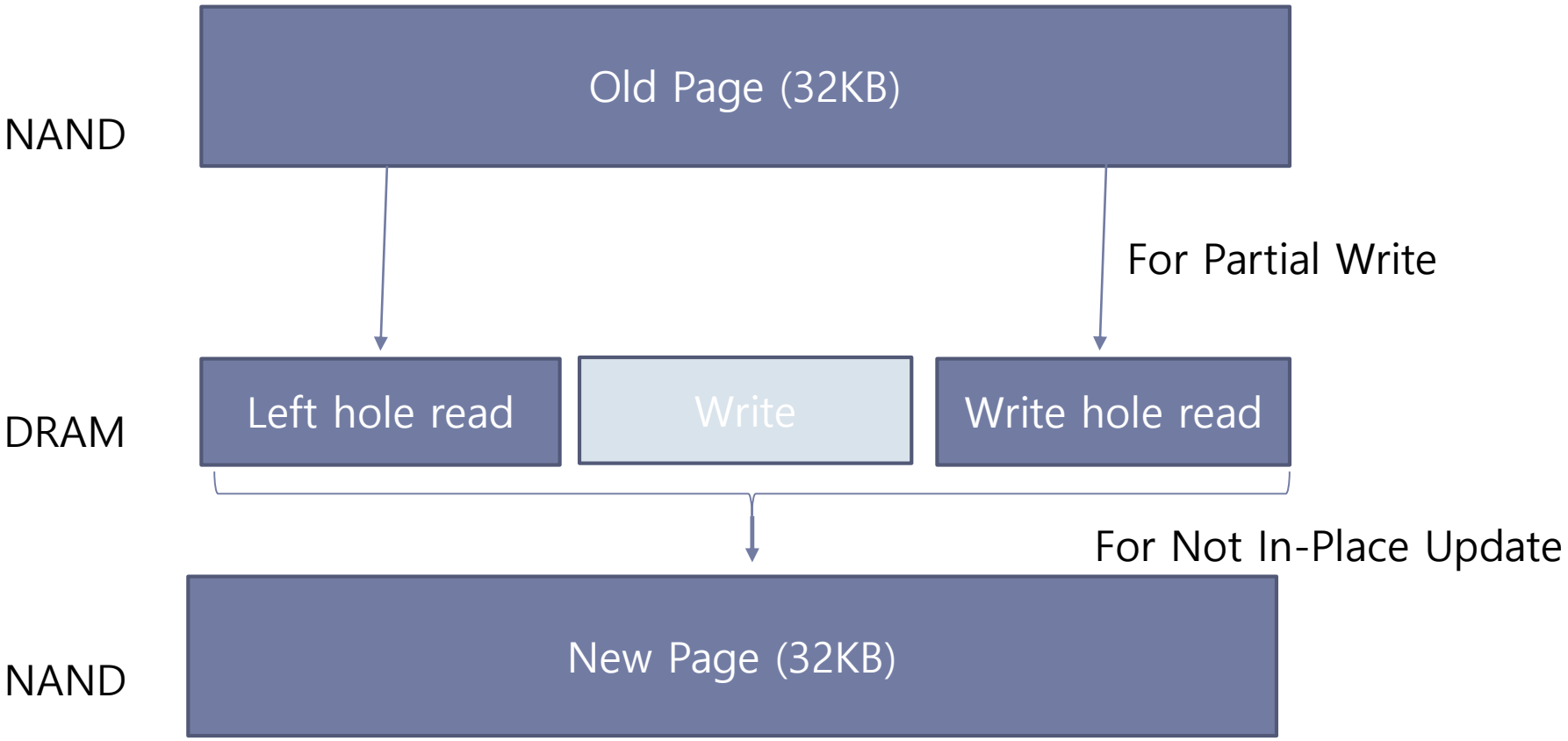
```
void ftl_read(UINT32 const lba, UINT32 const total_sectors)
{
    UINT32 bank, row, num_sectors_to_read, temp;

    UINT32 lpage_addr       = 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;
        }

        temp = get_physical_address(lpage_addr);     // logical to physical mapping

        if (temp != NULL)
        {
            bank = temp / PAGES_PER_BANK;    // most significant bits represent bank number
            row = temp % PAGES_PER_BANK;     // and remaining bits represent page number within the bank

            SETREG(FCP_CMD, FC_COL_ROW_READ_OUT);
            SETREG(FCP_DMA_CNT, num_sectors_to_read * BYTES_PER_SECTOR);
            SETREG(FCP_COL, sect_offset);
            SETREG(FCP_DMA_ADDR, RD_BUF_PTR(g_ftl_read_buf_id));
            SETREG(FCP_OPTION, FO_P | FO_E | FO_B_SATA_R);
            SETREG(FCP_ROW_L(bank), row);
            SETREG(FCP_ROW_H(bank), row);

            g_ftl_read_buf_id = (g_ftl_read_buf_id + 1) % NUM_RD_BUFFERS;    // circular buffer

            while (1)
            {
                UINT32 sata_id = GETREG(SATA_RBUF_PTR); // wait if the read buffer is full (slow host)

                if (g_ftl_read_buf_id != sata_id)
                    break;
            }

            // Now that all the FCP registers contain valid values, we can call flash_issue_cmd().
            flash_issue_cmd(bank, RETURN_ON_ISSUE);
        } ? end if temp! =NULL ?
```

Write Operation

- ./ftl_tutorial/ftl.c

```
void ftl_write(UINT32 const lba, UINT32 const total_sectors)
{
    UINT32 new_bank, new_row, old_row, num_sectors_to_write, right_hole_sectors, left_hole_sectors;
    UINT32 old_phys_page, old_bank, read_old_data;

    UINT32 lpage_addr   = lba / SECTORS_PER_PAGE;
    UINT32 sect_offset  = lba % SECTORS_PER_PAGE;
    UINT32 remain_sectors = total_sectors;

    while (remain_sectors != 0)
    {
        new_bank = g_target_bank;
        g_target_bank = (g_target_bank + 1) % NUM_BANKS;

        new_row = get_free_page(new_bank);                  // row address of the page to write to
        old_phys_page = get_physical_address(lpage_addr);   // row address of the page the old data was written to
        old_bank = old_phys_page / PAGES_PER_BANK;
        old_row = old_phys_page % PAGES_PER_BANK;

        if (sect_offset + remain_sectors >= SECTORS_PER_PAGE)
        {
            right_hole_sectors = 0;
            num_sectors_to_write = SECTORS_PER_PAGE - sect_offset;
        }
        else
        {
            UINT32 end_sect_offset = sect_offset + remain_sectors - 1;
            right_hole_sectors = SECTORS_PER_PAGE - end_sect_offset - 1;
            num_sectors_to_write = remain_sectors;
        }

        left_hole_sectors = sect_offset;
        read_old_data = FALSE;
```

Write Operation



Write Operation

- ./ftl_tutorial/ftl.c

```
if (old_phys_page != NULL)
{
    if (left_hole_sectors != 0)
    {
        // Left hole data will be read from flash to DRAM, at the address FCP_DMA_ADDR

        SETREG(FCP_CMD, FC_COL_ROW_READ_OUT);
        SETREG(FCP_OPTION, FO_P | FO_E);
        SETREG(FCP_DMA_ADDR, WR_BUF_PTR(g_ftl_write_buf_id));
        SETREG(FCP_DMA_CNT, left_hole_sectors * BYTES_PER_SECTOR);
        SETREG(FCP_COL, 0);
        SETREG(FCP_ROW_L(old_bank), old_row);
        SETREG(FCP_ROW_H(old_bank), old_row);

        flash_issue_cmd(old_bank, RETURN_ON_ISSUE);

        read_old_data = TRUE;
    }

    if (right_hole_sectors != 0)
    {
        // Right hole data will be read from flash to DRAM, at the address
        // FCP_DMA_ADDR + BYTES_PER_SECTOR * (SECTORS_PER_PAGE - right_hole_sectors)

        SETREG(FCP_CMD, (left_hole_sectors != 0) ? FC_COL_OUT : FC_COL_ROW_READ_OUT);
        SETREG(FCP_OPTION, FO_P | FO_E);
        SETREG(FCP_DMA_ADDR, WR_BUF_PTR(g_ftl_write_buf_id));
        SETREG(FCP_DMA_CNT, right_hole_sectors * BYTES_PER_SECTOR);
        SETREG(FCP_COL, SECTORS_PER_PAGE - right_hole_sectors);
        SETREG(FCP_ROW_L(old_bank), old_row);
        SETREG(FCP_ROW_H(old_bank), old_row);

        flash_issue_cmd(old_bank, RETURN_ON_ISSUE);

        read_old_data = TRUE;
    }
} ? end if old_phys_page! =NULL ?
```

Write Operation

- ./ftl_tutorial/ftl.c

```
        SETREG(FCP_CMD, FC_COL_ROW_IN_PROG);
        SETREG(FCP_OPTION, FO_P | FO_E | FO_B_SATA_W);

        SETREG(FCP_DMA_ADDR, WR_BUF_PTR(g_ftl_write_buf_id));
        SETREG(FCP_DMA_CNT, (left_hole_sectors + num_sectors_to_write + right_hole_sectors) * BYTES_PER_SECTOR);
        SETREG(FCP_COL, (left_hole_sectors != 0) ? 0 : sect_offset);
        SETREG(FCP_ROW_L(new_bank), new_row);
        SETREG(FCP_ROW_H(new_bank), new_row);

        if (read_old_data && old_bank != new_bank)
        {
            while ((GETREG(WR_STAT) & 0x00000001) != 0);     // wait for the old bank to accept the read command
            while (BSP_FSM(old_bank) != BANK_IDLE);          // wait for the old bank to finish the read command
        }

        flash_issue_cmd(new_bank, RETURN_ON_ISSUE);

        update_physical_address(lpage_addr, new_bank, new_row);

        sect_offset = 0;
        remain_sectors -= num_sectors_to_write;
        lpage_addr++;

        g_ftl_write_buf_id = (g_ftl_write_buf_id + 1) % NUM_WR_BUFFERS;      // Circular buffer
    } ? end while remain_sectors! =0 ?
} ? end ftl_write ?
```

NAND Flash Operation

- ./target_spw/flash.c

```
void nand_page_read_to_host(UINT32 const bank, UINT32 const vblock, UINT32 const page_num)
{
    UINT32 row;

    ASSERT(bank < NUM_BANKS);
    ASSERT(vblock < VBLKS_PER_BANK);
    ASSERT(page_num < PAGES_PER_BLK);

    row = (vblock * PAGES_PER_BLK) + page_num;

    SETREG(FCP_CMD, FC_COL_ROW_READ_OUT);
    SETREG(FCP_DMA_ADDR, RD_BUF_PTR(g_ftl_read_buf_id));
    SETREG(FCP_DMA_CNT, BYTES_PER_PAGE);

    SETREG(FCP_COL, 0);
    SETREG(FCP_OPTION, FO_P | FO_E | FO_B_SATA_R);

    SETREG(FCP_ROW_L(bank), row);
    SETREG(FCP_ROW_H(bank), row);

    g_ftl_read_buf_id = (g_ftl_read_buf_id + 1) % NUM_RD_BUFFERS;

    flash_issue_cmd(bank, RETURN_ON_ISSUE);
} ? end nand_page_read_to_host ?
```

NAND Flash Operation

- ./target_spw/flash.c

```
void flash_issue_cmd(UINT32 const bank, UINT32 const sync)
{
    UINT32 rbank = REAL_BANK(bank);

    SETREG(FCP_BANK, rbank);

    // You should not issue a new command when Waiting Room is not empty.
    while ((GETREG(WR_STAT) & 0x00000001) != 0);

    // If you write any value to FCP_ISSUE, FCP register contents are copied to Waiting Room.
    // The hardware does not read FCP registers unless you write any value to FCP_ISSUE.
    // The hardware never changes the values of FCP registers unless reset.
    // The value written to FCP_ISSUE is not significant.
    SETREG(FCP_ISSUE, NULL);

    if (sync == RETURN_ON_ISSUE)
        return;

    // wait until the new command is accepted by the target bank
    while ((GETREG(WR_STAT) & 0x00000001) != 0);

    if (sync == RETURN_ON_ACCEPT)
        return;

    // wail until the target bank finishes the command
    while (_BSP_FSM(rbank) != BANK_IDLE);
} ? end flash_issue_cmd ?
```

NAND Flash Operation

- ./target_spw/flash_wrapper.c

```
void nand_page_read(UINT32 const bank, UINT32 const vblock, UINT32 const page_num, UINT32 const buf_addr);
void nand_page_ptread(UINT32 const bank, UINT32 const vblock, UINT32 const page_num,
                      UINT32 const sect_offset, UINT32 const num_sectors, UINT32 const buf_addr, UINT32 const issue_flag);

void nand_page_read_to_host(UINT32 const bank, UINT32 const vblock, UINT32 const page_num);
void nand_page_ptread_to_host(UINT32 const bank, UINT32 const vblock, UINT32 const page_num,
                              UINT32 const sect_offset, UINT32 const num_sectors);

void nand_page_program(UINT32 const bank, UINT32 const vblock, UINT32 const page_num, UINT32 const buf_addr);
void nand_page_ptprogram(UINT32 const bank, UINT32 const vblock, UINT32 const page_num,
                         UINT32 const sect_offset, UINT32 const num_sectors, UINT32 const buf_addr);

void nand_page_program_from_host(UINT32 const bank, UINT32 const vblock, UINT32 const page_num);
void nand_page_ptprogram_from_host(UINT32 const bank, UINT32 const vblock, UINT32 const page_num,
                                   UINT32 const sect_offset, UINT32 const num_sectors);

void nand_page_copyback(UINT32 const bank, UINT32 const src_vblock, UINT32 const src_page,
                        UINT32 const dst_vblock, UINT32 const dst_page);
void nand_page_modified_copyback(UINT32 const bank, UINT32 const src_vblock, UINT32 const src_page,
                                 UINT32 const dst_vblock, UINT32 const dst_page,
                                 UINT32 const sect_offset, UINT32 dma_addr, UINT32 const dma_count);

void nand_block_erase(UINT32 const bank, UINT32 const vblock);
void nand_block_erase_sync(UINT32 const bank, UINT32 const vblock);
```

Error Handling

- ./ftl_tutorial/ftl.c

```
void ftl_isr(void)
{
    UINT32 bank;
    UINT32 bsp_intr_flag;

    for (bank = 0; bank < NUM_BANKS; bank++)
    {
        while (BSP_FSM(bank) != BANK_IDLE);

        bsp_intr_flag = BSP_INTR(bank);

        if (bsp_intr_flag == 0)
        {
            continue;
        }

        UINT32 fc = GETREG(BSP_CMD(bank));

        CLR_BSP_INTR(bank, bsp_intr_flag);

        if (bsp_intr_flag & FIRQ_DATA_CORRUPT)
        {
            g_read_fail_count++;
        }

        if (bsp_intr_flag & (FIRQ_BADBLK_H | FIRQ_BADBLK_L))
        {
            if (fc == FC_COL_ROW_IN_PROG || fc == FC_IN_PROG || fc == FC_PROG)
            {
                g_program_fail_count++;
            }
            else
            {
                ASSERT(fc == FC_ERASE);
                g_erase_fail_count++;
            }
        }
    } ? end for bank=0;bank<NUM_BANKS... ?

    SETREG(APB_INT_STS, INTR_FLASH);
} ? end ftl_isr ?
```

ftl_open()

```
SETREG(INTR_MASK, FIRQ_DATA_CORRUPT | FIRQ_BADBLK_L | FIRQ_BADBLK_H);
SETREG(FCONF_PAUSE, FIRQ_DATA_CORRUPT | FIRQ_BADBLK_L | FIRQ_BADBLK_H);
```

Any Questions?