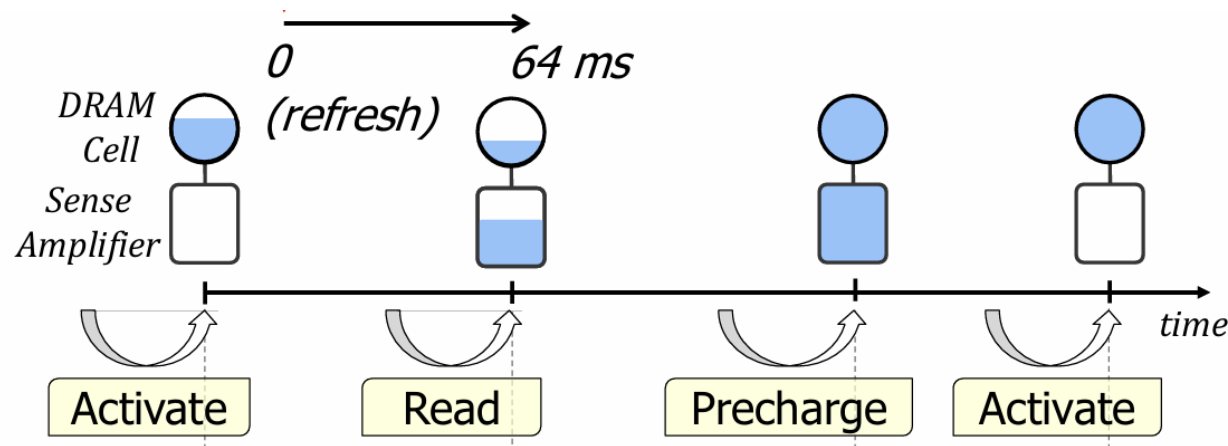

Ramulator 2.0 Summary

***Intelligent System
Laboratory***

DRAM Operations & States

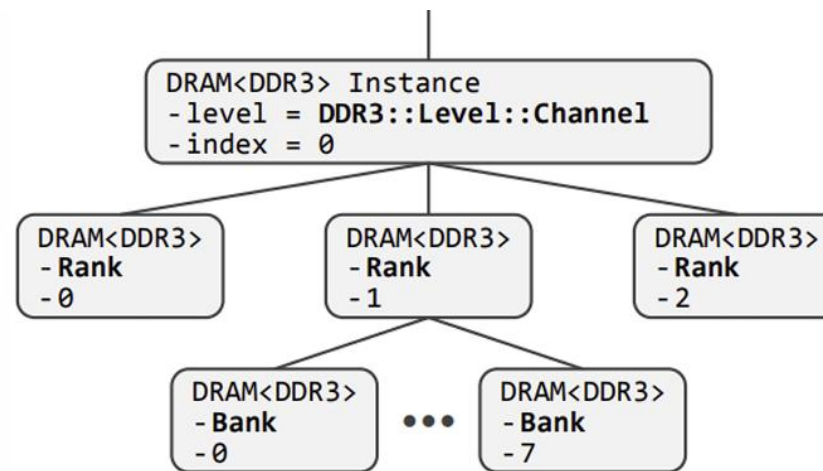
□ DRAM Operations & States



```
1 // DRAM.h
2 template <typename T>
3 class DRAM {
4     DRAM<T>* parent;
5     vector<DRAM<T>*>
6         children;
7     T::Level level;
8     int index;
9     // more code...
10 };
```

```
1 // DDR3.h/cpp
2 class DDR3 {
3     enum class Level {
4         Channel, Rank, Bank,
5         Row, Column, MAX
6     };
7
8     // more code...
9
10 };
```

- Main DRAM states
 - Activate
 - Read/Write
 - Precharge



src files \Leftrightarrow DRAM Operation

□ Simulation Configuration

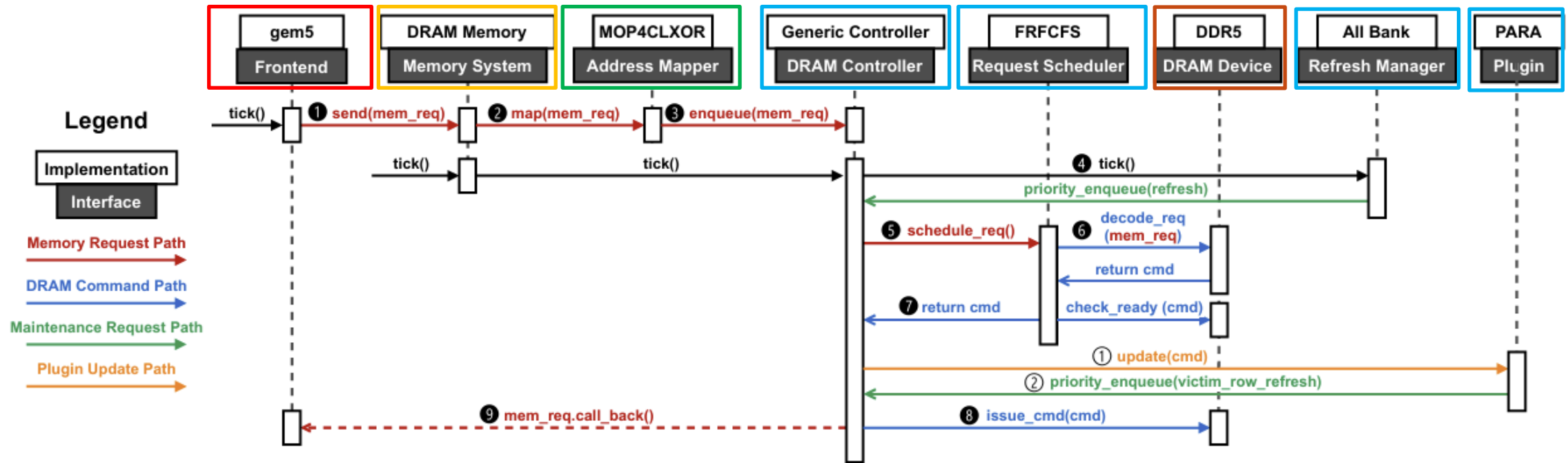
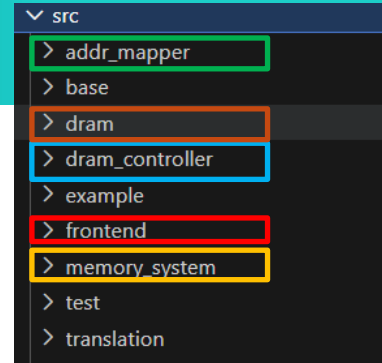


Fig. 1: High-level software architecture of Ramulator 2.0 using an example DDR5 system configuration

src files <=> DRAM Operation

□ Simulation Configuration

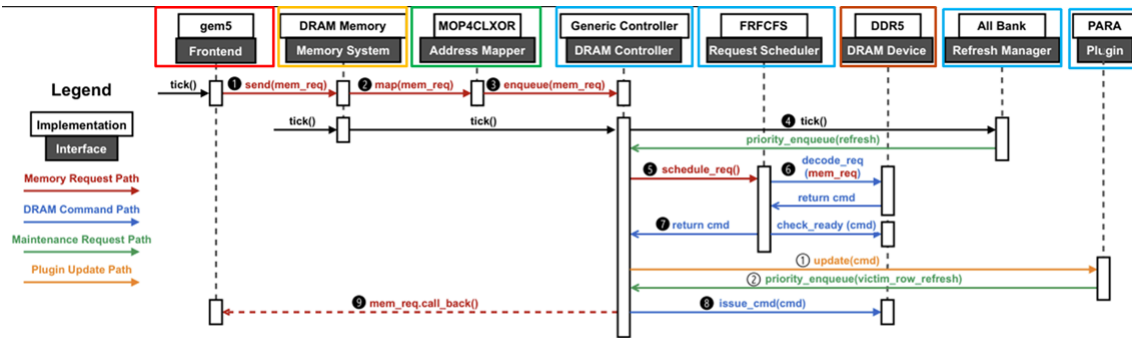
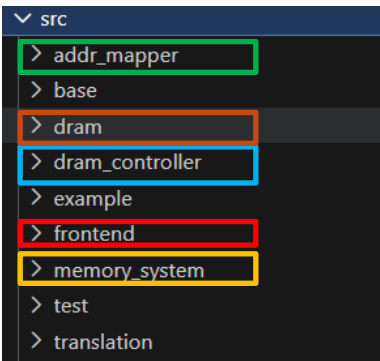


Fig. 1: High-level software architecture of Ramulator 2.0 using an example DDR5 system configuration



1. **Requests are sent:** Front-end(trace file)에서 Memory Request를 보냄
2. **Memory Addresses are Mapped:** Address Mapper가 Request Address를 DRAM 구조에 맞게 변환
3. **Enqueue:** DRAM Ctrlr의 Buffer에 Request를 넣음
4. **DRAM Ctrlr - Ticking Refresh Manager:** Ctrlr가 Refresh Manager를 호출해 high-priority maintenance request(ex. Refresh)을 추가
5. **DRAM Ctrlr - Request Scheduling:** Request Scheduler에게 최적의 Request를 선택하라고 요청
6. **DRAM Device가 Request 확인:** Scheduler가 DRAM Device Model을 참조해 적합한 Command를 Decode
7. **Issue Command:** DRAM Ctrlr가 DRAM Command를 보냄
8. **Updates the behavior and timing information:** DRAM Command Issue시 State & Timing이 Update
9. **Notify the frontend:** Memory Request가 끝나면 callback으로 frontend에 알림

main function

□ main.cpp

```
13 ~ int main(int argc, char* argv[]) {  
14     // Parse command line arguments  
15     argparse::ArgumentParser program("Ramulator", "2.0");  
16     program.add_argument("-c", "--config").metavar("\ndumped YAML configuration")  
17     .help("String dump of the yaml configuration.");  
18     program.add_argument("-f", "--config_file").metavar("path-to-configuration-file")  
19     .help("Path to a YAML configuration file.");  
20     program.add_argument("-p", "--param").metavar("KEY=VALUE")  
21     .append()  
22     .help("Specify parameter to override in the configuration file. Repeat this option to change multiple parameters.");
```

:

```
88     // Connect the frontend and the memory system together,  
89     // this recursively calls the "setup" function in all instantiated components  
90     // so that they can get each other's parameters (if needed) after their initialization  
91     frontend->connect_memory_system(memory_system);  
92     memory_system->connect_frontend(frontend);  
93  
94     // Get the relative clock ratio between the frontend and memory system  
95     int frontend_tick = frontend->get_clock_ratio();  
96     int mem_tick = memory_system->get_clock_ratio();  
97  
98     int tick_mult = frontend_tick * mem_tick;  
99  
100    for (uint64_t i = 0;; i++) {  
101        if (((i % tick_mult) % mem_tick) == 0) {  
102            frontend->tick();  
103        }  
104  
105        if (frontend->is_finished()) {  
106            break;  
107        }  
108  
109        if ((i % tick_mult) % frontend_tick == 0) {  
110            memory_system->tick();  
111        }  
112    }  
113  
114    // Finalize the simulation. Recursively print all statistics from all components  
115    frontend->finalize();  
116    memory_system->finalize();  
117  
118    return 0;  
119 }
```

main.cpp

1. Argument 받는 부분

- Options

1. -c: command line dump
2. -f: YAML document
3. -p: overriding parameters in a YAML document

2. Long for loop를 통한 tick() 기반 simul

1. frontend(core)가 발행한 예상 instructions들을 모두 처리시 is_finished()가 true가 됨

yaml file

□ example_config.yaml

```
1  Frontend:
2    impl: SimpleO3
3    clock_ratio: 8
4    num_expected_insts: 500000
5  traces:
6    - example_inst.trace
7
8  Translation:
9    impl: RandomTranslation
10    max_addr: 2147483648
11
12
13 MemorySystem:
14   impl: GenericDRAM
15   clock_ratio: 3
16
17   DRAM:
18     impl: DDR4
19     org:
20       preset: DDR4_8Gb_x8
21       channel: 1
22     rank: 2
23     timing:
24       preset: DDR4_2400R
25
26   Controller:
27     impl: Generic
28     Scheduler:
29       impl: FRFCFS
30     RefreshManager:
31       impl: AllBank
32     RowPolicy:
33       impl: ClosedRowPolicy
34       cap: 4
35     plugins:
36
37   AddrMapper:
38     impl: RoBaRaCoCh
```

1. Frontend Interface(IFrontEnd) 부분

- trace file에서 Instruction읽고, Memory Request 생성
- impl: SimpleO3
⇒ Simple Out-of-Order (O3) CPU
- clock_ratio: 8
⇒ global CLK 대비 Frontend CLK 속도
- num_expected_insts: 500000
⇒ Simulation이 해당 instruction 수에 도달 시 종료
- traces
⇒ Instruction trace file(include memory access Inst)
- impl: RandomTranslation
⇒ Physical Memory ↔ Virtual Memory 변환
⇒ System의 Page Table 등을 간단히 Modeling
- max_addr: 2147483648
⇒ Translation 시 address overflow 방지

yaml file

□ example_config.yaml

```
1  Frontend:
2    impl: SimpleO3
3    clock_ratio: 8
4    num_expected_insts: 500000
5  traces:
6    - example_inst.trace
7
8  Translation:
9    impl: RandomTranslation
10   max_addr: 2147483648
11
12
13  MemorySystem:
14    impl: GenericDRAM
15    clock_ratio: 3
16
17  DRAM:
18    impl: DDR4
19    org:
20      preset: DDR4_8Gb_x8
21      channel: 1
22      rank: 2
23    timing:
24      preset: DDR4_2400R
25
26  Controller:
27    impl: Generic
28    Scheduler:
29      impl: FRFCFS
30    RefreshManager:
31      impl: AllBank
32    RowPolicy:
33      impl: ClosedRowPolicy
34      cap: 4
35    plugins:
36
37  AddrMapper:
38    impl: RoBaRaCoCh
```

2. MemorySystem Interface 부분

- Frontend의 Request를 받아 DRAM Ctrlr을 통해 처리
- Latency, en/dequeue, Timing Constraints 처리
- impl: GenericDRAM
⇒ 기본 DRAM 기반 System, Ctrlr와 DRAM을 통합
- clock_ratio: 3
⇒ global CLK 대비 MemorySystem CLK 속도
⇒ 현재: DRAM이 CPU보다 느린 System (= 3:8)

yaml file

example_config.yaml

```
1 Frontend: 13 MemorySystem:
2   impl: SimpleO3 14   impl: GenericDRAM
3   clock_ratio: 8 15   clock_ratio: 3
4   num_expected_insts: 500000 16
5   traces: 17 DRAM:
6     - example_inst.trace 18   impl: DDR4
7 8 Translation: 19   org:
9   impl: RandomTranslation 20     preset: DDR4_8Gb_x8
10   max_addr: 2147483648 21     channel: 1
11 22     rank: 2
23     timing:
24     preset: DDR4_2400R
25
26 Controller:
27   impl: Generic
28   Scheduler:
29     impl: FRFCFS
30   RefreshManager:
31     impl: AllBank
32   RowPolicy:
33     impl: ClosedRowPolicy
34     cap: 4
35   plugins:
36
37 AddrMapper:
38   impl: RoBaRaCoCh
```

```
26 inline static const std::map<std::string, std::vector<int>> timing_presets = {
27   // name rate nBL nCL nRCD nRP nRAS nRC nWR nRTP nCWL nCCDS nCCDL nRRDS nRRDL nWTRS nW
28   {"DDR4_1600J", {1600, 4, 10, 10, 10, 28, 38, 12, 6, 9, 4, 5, -1, -1, 2,
29   {"DDR4_1600K", {1600, 4, 11, 11, 11, 28, 39, 12, 6, 9, 4, 5, -1, -1, 2,
30   {"DDR4_1600L", {1600, 4, 12, 12, 12, 28, 40, 12, 6, 9, 4, 5, -1, -1, 2,
31   {"DDR4_1866L", {1866, 4, 12, 12, 12, 32, 44, 14, 7, 10, 4, 5, -1, -1, 3,
32   {"DDR4_1866M", {1866, 4, 13, 13, 13, 32, 45, 14, 7, 10, 4, 5, -1, -1, 3,
33   {"DDR4_1866N", {1866, 4, 14, 14, 14, 32, 46, 14, 7, 10, 4, 5, -1, -1, 3,
34   {"DDR4_2133N", {2133, 4, 14, 14, 14, 36, 50, 16, 8, 11, 4, 6, -1, -1, 3,
35   {"DDR4_2133P", {2133, 4, 15, 15, 15, 36, 51, 16, 8, 11, 4, 6, -1, -1, 3,
36   {"DDR4_2133R", {2133, 4, 16, 16, 16, 36, 52, 16, 8, 11, 4, 6, -1, -1, 3,
37   {"DDR4_2400P", {2400, 4, 15, 15, 15, 39, 54, 18, 9, 12, 4, 6, -1, -1, 3,
38   {"DDR4_2400R", {2400, 4, 16, 16, 16, 39, 55, 18, 9, 12, 4, 6, -1, -1, 3,
39   {"DDR4_2400U", {2400, 4, 17, 17, 17, 39, 56, 18, 9, 12, 4, 6, -1, -1, 3,
40   {"DDR4_2400T", {2400, 4, 18, 18, 18, 39, 57, 18, 9, 12, 4, 6, -1, -1, 3,
```

2. MemorySystem Interface 부분

• DRAM Section

• impl: DDR4

⇒ tick() 시 Timing Check / Command Issue 실행.

• org: DDR4 8Gb x8

⇒ 현재 **DRAM preset** - 8Gb 용량, x8bit data bus

⇒ **Channel/Rank 설정** 시 기본 Preset설정을 Override 함

• timing: DDR4 2400R

⇒ **Timing preset** - nRCD등의 Timing Constraint 정의

⇒ 이를 이용해 tick() 시 Latency 계산

```
class DDR4 : public IDRAM, public Implementation {
    RAMULATOR_REGISTER_IMPLEMENTATION(IDRAM, DDR4, "DDR4", "DDR4 Device Model")

public:
    inline static const std::map<std::string, Organization> org_presets = {
        // name density DQ Ch Ra Bg Ba Ro Co
        {"DDR4_2Gb_x4", {2<<10, 4, {1, 1, 4, 4, 1<<15, 1<<10}}},
        {"DDR4_2Gb_x8", {2<<10, 8, {1, 1, 4, 4, 1<<14, 1<<10}}},
        {"DDR4_2Gb_x16", {2<<10, 16, {1, 1, 2, 4, 1<<14, 1<<10}}},
        {"DDR4_4Gb_x4", {4<<10, 4, {1, 1, 4, 4, 1<<16, 1<<10}}},
        {"DDR4_4Gb_x8", {4<<10, 8, {1, 1, 4, 4, 1<<15, 1<<10}}},
        {"DDR4_4Gb_x16", {4<<10, 16, {1, 1, 2, 4, 1<<15, 1<<10}}},
        {"DDR4_8Gb_x4", {8<<10, 4, {1, 1, 4, 4, 1<<17, 1<<10}}},
        {"DDR4_8Gb_x8", {8<<10, 8, {1, 1, 4, 4, 1<<16, 1<<10}}},
        {"DDR4_8Gb_x16", {8<<10, 16, {1, 1, 2, 4, 1<<16, 1<<10}}},
        {"DDR4_16Gb_x4", {16<<10, 4, {1, 1, 4, 4, 1<<18, 1<<10}}},
        {"DDR4_16Gb_x8", {16<<10, 8, {1, 1, 4, 4, 1<<17, 1<<10}}},
        {"DDR4_16Gb_x16", {16<<10, 16, {1, 1, 2, 4, 1<<17, 1<<10}}},
    };
};
```

```
class DDR4 : public IDRAM, public Implementation {
    void tick() override {

    void init() override {
        RAMULATOR_DECLARE_SPECS();
        set_organization();
        set_timing_vals();

        set_actions();
        set_preqs();
        set_rowhits();
        set_rowopens();
        set_powers();

        create_nodes();
    };
};
```


yaml file

□ example_config.yaml

```
1 Frontend:
2   impl: SimpleO3
3   clock_ratio: 8
4   num_expected_insts: 500000
5   traces:
6     - example_inst.trace
7
8   Translation:
9     impl: RandomTranslation
10    max_addr: 2147483648
11
12
13 MemorySystem:
14   impl: GenericDRAM
15   clock_ratio: 3
16
17   DRAM:
18     impl: DDR4
19     org:
20       preset: DDR4_8Gb_x8
21       channel: 1
22       rank: 2
23     timing:
24       preset: DDR4_2400R
25
26   Controller:
27     impl: Generic
28     Scheduler:
29       impl: FRFCFS
30     RefreshManager:
31       impl: AllBank
32     RowPolicy:
33       impl: ClosedRowPolicy
34       cap: 4
35     plugins:
36
37   AddrMapper:
38     impl: RoBaRaCoCh
```

2. MemorySystem Interface 부분

- **Controller Section**
- impl: Generic
 - ⇒ Generic - 기본 Ctrlr
 - ⇒ Request Queue/Scheduling 등 관리
- Scheduler - impl: FRFCFS
 - ⇒ FRFCFS - First-Ready First-Come-First-Serve
 - ⇒ 준비된 Request를 Queue에서 꺼내 우선 처리
- RefreshManager - impl: AllBank
 - ⇒ AllBank - 모든 Bank simultaneous Refresh
- RowPolicy - impl: ClosedRowPolicy
 - ⇒ ClosedRowPolicy - 사용 후 Row 즉시 닫음(Precharge)
 - ⇒ cap:4 - 열려있는 Row 최대 수 제한
- plugins
 - ⇒ 현재 Ramulator에서는 Row Hammering 완화 기법을 plugin으로 제공해줌

yaml file

□ example_config.yaml

```
1  Frontend:
2    impl: SimpleO3
3    clock_ratio: 8
4    num_expected_insts: 500000
5  traces:
6    - example_inst.trace
7
8  Translation:
9    impl: RandomTranslation
10   max_addr: 2147483648
11
12
13 MemorySystem:
14   impl: GenericDRAM
15   clock_ratio: 3
16
17   DRAM:
18     impl: DDR4
19     org:
20       preset: DDR4_8Gb_x8
21       channel: 1
22       rank: 2
23     timing:
24       preset: DDR4_2400R
25
26   Controller:
27     impl: Generic
28     Scheduler:
29       impl: FRFCFS
30     RefreshManager:
31       impl: AllBank
32     RowPolicy:
33       impl: ClosedRowPolicy
34       cap: 4
35     plugins:
36
37   AddrMapper:
38     impl: RoBaRaCoCh
```

2. MemorySystem Interface 부분

- AddrMapper Section

- impl: RoBaRaCoCh

⇒ Row-Bank-Rank-Column-Channel Mapping Scheme

⇒ Requested Address 변환(Physical → DRAM Vector)

[Physical → DRAM Vector Example]

⇒ Physical Address: 0x12345678

⇒ DRAM Vector:

[Channel:0, Rank:1, Bank:2, Row:128, Column:512]

□ example_inst.trace & trace.cpp & core.cpp

- simpleO3 CPU model기준
 - Ramulator는 “Memory” Simulator
 - Memory 명령어만 취급하기에, 3가지로만 Instruction을 분리한다.

1. Not Memory Operation

2. Load

3. Store

- 따라서, simpleO3 기반 trace파일:
 - 1st column은 Not Memory Operation Cycle 수 (or ticks 수)
 - 2nd column은 load operation address
 - 3rd column은 store operation address

example_inst.trace		
1	3	20734016
2	1	20846400
3	6	20734208
4	1	20846400
5	8	20841280 20841280
6	0	20734144
7	2	20918976 20734016
8	1	20846400
9		

- 각 line은 load(& store)동작을 나타낸다.
- 1st line : 3cycle동안 stall → load
- 5th line : 8cycle동안 stall → load → store

□ example_inst.trace & trace.cpp & core.cpp

The screenshot displays three code files: `trace.cpp`, `core.cpp`, and `example_inst.trace`. In `trace.cpp`, the `while` loop processes tokens from the trace file. Lines 114-124 show token parsing, with `int bubble_count = std::stoi(tokens[0]);` and `Addr_t load_addr = std::stoll(tokens[1]);` highlighted in red. Lines 125-134 show store address parsing, with `Addr_t store_addr = std::stoll(tokens[2]);` highlighted in green. A red box highlights the `if (has store)` block. In `core.cpp`, the `SimpleO3Core::tick()` function is shown. Lines 125-126 show `int num_inserted_insts = 0;` and `while (m_num_bubbles > 0)` highlighted in red. Lines 127-134 show the `if (num_inserted_insts == m_window.m_ipc)` block highlighted in green. Lines 135-144 show the `if (m_load_addr != -1)` block highlighted in green. In `example_inst.trace`, the first line is `3 20734016`, with the bubble count '3' highlighted in red and the address '20734016' highlighted in green. A red box highlights the first line, and a green box highlights the address. A red arrow points from the `int bubble_count` line in `trace.cpp` to the bubble count '3' in the trace file. A green arrow points from the `Addr_t load_addr` line in `trace.cpp` to the address '20734016' in the trace file. A green arrow points from the `Addr_t store_addr` line in `trace.cpp` to the address '20841280' in the trace file. A red arrow points from the `if (has store)` block in `trace.cpp` to the `if (num_inserted_insts == m_window.m_ipc)` block in `core.cpp`. A green arrow points from the `if (m_load_addr != -1)` block in `core.cpp` to the `if (m_load_addr != -1)` block in `core.cpp`. A green arrow points from the `if (m_writeback_addr != -1)` block in `core.cpp` to the `if (m_writeback_addr != -1)` block in `core.cpp`.

```
src > frontend > impl > processor > simpleO3 > trace.cpp > ...
std::string line;
while (std::getline(trace_file, line)) {
    std::vector<std::string> tokens;
    tokenize(tokens, line, " ");

    int num_tokens = tokens.size();
    if (num_tokens != 2 & num_tokens != 3) {
        throw ConfigurationError("Trace {} format invalid!", file_path_str);
    }

    int bubble_count = std::stoi(tokens[0]);
    Addr_t load_addr = std::stoll(tokens[1]);

    bool has_store = num_tokens == 2 ? false : true;
    if (has_store) {
        Addr_t store_addr = std::stoll(tokens[2]);
        m_trace.push_back({bubble_count, load_addr, store_addr});
    } else {
        m_trace.push_back({bubble_count, load_addr, -1});
    }
}

trace_file.close();
m_trace.length = m_trace.size();

src > frontend > impl > processor > simpleO3 > core.cpp > {} Ramulator > tick()
void SimpleO3Core::tick() {
    m_clk++;

    s_insts_retired += m_window.retire();
    if (!reached_expected_num_insts) {
        if (s_insts_retired >= m_num_expected_insts) {
            reached_expected_num_insts = true;
            s_cycles_recorded = m_clk;
        }
    }

    // First, issue the non-memory instructions.
    int num_inserted_insts = 0;
    while (m_num_bubbles > 0) {
        if (num_inserted_insts == m_window.m_ipc) {
            return;
        }
        if (m_window.is_full()) {
            return;
        }
        m_window.insert(true, -1);
        num_inserted_insts++;
        m_num_bubbles--;
    }

    // Second, try to send the load to the LLC
    if (m_load_addr != -1) {
        if (num_inserted_insts == m_window.m_ipc) {
            return;
        }
        if (m_window.is_full()) {
            return;
        }
    }

    Request load_request(m_load_addr, Request::Type::Read, m_id, m_callback);
    if (!m_translation->translate(load_request)) {
        return;
    }

    if (m_llc->send(load_request)) {
        m_window.insert(false, load_request.addr);
        m_load_addr = -1;
        if (m_writeback_addr != -1) {
            // If there is still writeback, return without getting the next trace line
            // The write back will be issued in the next cycle
            // TODO: Should we allow both load and writeback to issue at the same cycle?
            return;
        }
    } else {
        return;
    }

    // Third, try to send the writeback to the LLC
    if (m_writeback_addr != -1) {
        Request writeback_request(m_writeback_addr, Request::Type::Write, m_id, m_callback);
        if (!m_translation->translate(writeback_request)) {
            return;
        }
    }

    if (m_llc->send(writeback_request)) {
        return;
    }

    auto inst = m_trace.get_next_inst();
    m_num_bubbles = inst.bubble_count;
    m_load_addr = inst.load_addr;
    m_writeback_addr = inst.store_addr;
}
```

각 줄이 하나의 Instruction처럼 CPU 모델에 들어감

```
example_inst.trace
1 3 20734016
2 1 20846400
3 6 20734208
4 1 20846400
5 8 20841280 20841280
6 0 20734144
7 2 20918976 20734016
8 1 20846400
9
```

- Trace file을 arg로 받아, Frontend (ex: simpleO3.cpp)에서 처리되어 메모리 접근 request를 생성
- simpleO3 CPU model기준: trace의 token[0]: bubble / token[1]: load address / token[2]: store address

src/frontend folder

□ frontend

- abcd

src/frontend folder

□ frontend

- abcd

src/frontend folder

□ frontend

- abcd

src/memory_system folder

□ **memory_system**

- **abcd**

src/memory_system folder

□ **memory_system**

- **abcd**

src/addr_mapper folder

□ **addr_mapper**

- **abcd**

src/addr_mapper folder

□ **addr_mapper**

- **abcd**

src/translation folder

□ translation

- abcd

src/translation folder

□ translation

- abcd

src/translation folder

□ translation

- abcd

src/dram folder

□ dram

- abcd

src/dram folder

□ dram

- abcd

src/dram folder

□ dram

- abcd

src/dram_controller folder

□ **dram_controller**

- **abcd**

src/dram_controller folder

□ **dram_controller**

- **abcd**

src/dram_controller folder

□ **dram_controller**

- **abcd**

src/dram_controller folder

□ **dram_controller**

- **abcd**

src/base folder

□ **base**

- **abcd**