PiM simulation using Gem5 - Ramulator

Introduction to Gem5 & Ramulator

- **Gem5:** System level & Processor simulator

  - Memory timing simulation은 정확하지 않음.

  - 참고: https://www.gem5.org/documentation/general_docs/building

- **Ramulator:** Memory simulator

  - Memory timing simulation을 위한 simulator

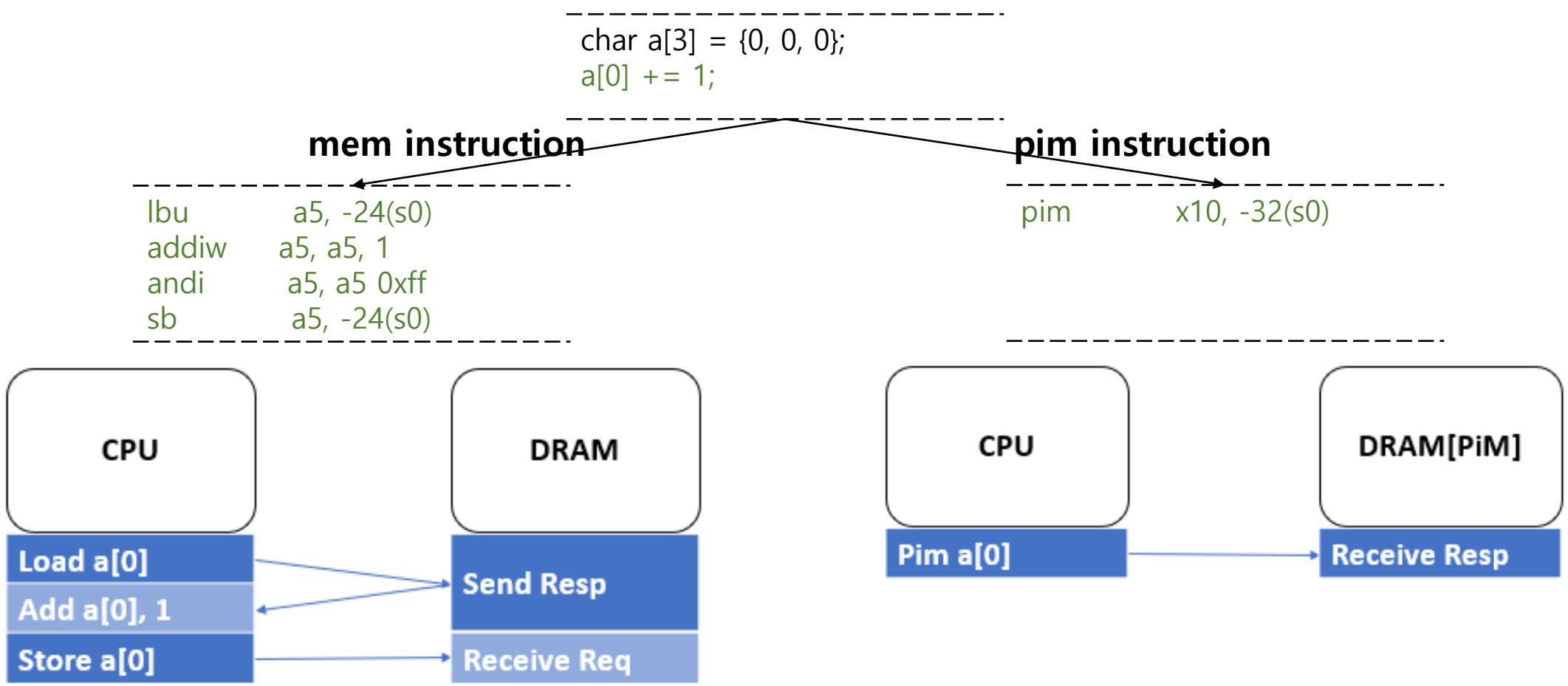
  - 참고: https://comsys-pim.tistory.com/15

Project details

- **Goal:** PiM을 Gem5-ramulator를 사용해 simulate 하고,
  performance를 측정하여 결과를 비교합니다.
  - PiM function: 주어진 address에 저장된 값에 +1을 수행
  - PiM spec: function 수행에 각 30/50/100cycles 소요 가정

Project details

- 목표로 하는 PIM 연산은 다음과 같이 compile 되어 실행됩니다.



Project details

- **CPU Architecture:** RISCV

X86	RISCV
범용적	범용적이지는 않으나, 아키텍처가 잘 알려져 있고, cross compiler 등 사용 가능
CISC instruction 사용으로, Gem5 에 custom instruction 추가 시 복잡 함.	RISC instruction 사용으로, Gem5 에 custom instruciton 추가 시 간단 함.

Project details

- **PiM Architecture:** Processing in Chip / DDR4

Processing Using Memory	Processing Near Memory	Processing In Chip
새로운 로직을 추가하지 않고, 기존 memory의 charget 및 activation 등의 signal을 통한 구현.	Memory 내부가 아닌, Memory 근처에 연산 모듈을 두고 (메모리 패키지 내부) 연산을 진행.	Memory의 Chip 내부에 연산 logic을 구현.
bandwidth에 좋음	연산 시 bandwidth 점유	Bandwidth에 좋음
Chip size 변화 없음	Chip size에 변화 없음	Chip size가 매우 커짐
비용이 상대적으로 적음	비용이 상대적으로 적음	개발 비용이 크다.

Project details

- **TODO:**
  1. Environment setup
  2. Custom instruction 제작하여 compiler에 추가
  3. Custom instruction을 ramulator에 추가
  4. Custom instruction을 Gem5에 추가
  5. Evaluation

1. Env. Setup

1. Env. Setup

- Server 환경: Ubuntu 20.04
- Ramulator download: https://github.com/CMU-SAFARI/ramulator
- Gem5-Ramulator download: https://github.com/Kirrito-k423/gem5
- Cross compiler for RISCV: https://github.com/riscv/riscv-gnu-toolchain.git

1. Env. Setup (Ramulator)

- Cluster 0에는 Ramulator를 위한 환경을 설정하였으며, Ramulator에서 PiM instruction을 추가하고자 할 때, Ramulator만 시뮬레이션 돌려보면 구현에 훨씬 도움이 됩니다.

```
1. Dependent package 설치
cluster00:~ $ sudo apt install build-essential git m4 scons zlib1g zlib1g-dev ₩
libprotobuf-dev protobuf-compiler libprotoc-dev libgoogle-perftools-dev ₩
python3-dev python-is-python3 libboost-all-dev pkg-config
cluster00:~ $ sudo vim /etc/apt/sources.list
vv 마지막 줄에 추가 vv
deb http://dk.archive.ubuntu.com/ubuntu/ xenial main
deb http://dk.archive.ubuntu.com/ubuntu/ xenial universe
^^^^^^^^^^^^^^^^^^^^^^^^
cluster00:~ $ sudo apt update
cluster00:~ $ sudo apt-get install clang libgoogle-perftools-dev swig3.0 mercurial g++-5 gcc-5
cmake
cluster00:~ $ sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-5 5
cluster00:~ $ sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-5 5
```

1. Env. Setup (Ramulator)

- Gem5와 Ramulator가 통합된 파일을 사용할 것이지만, 한번에 Gem5-Ramulator를 고치는 것보다 Ramulator에서 Pim instruction을 인식하여 실행될 수 있도록 구성하고, Gem5를 고친다음 Gem5와 Ramulator를 연결해주는 편이 더 안정적이고 빠르게 진행할 수 있을 것 같습니다.

2. 설치 & build

```
cluster00:~ $ git clone https://github.com/CMU-SAFARI/ramulator.git
cluster00:~/ramulator $ cd ramulator
cluster00:~/ramulator $ make –j 32
```

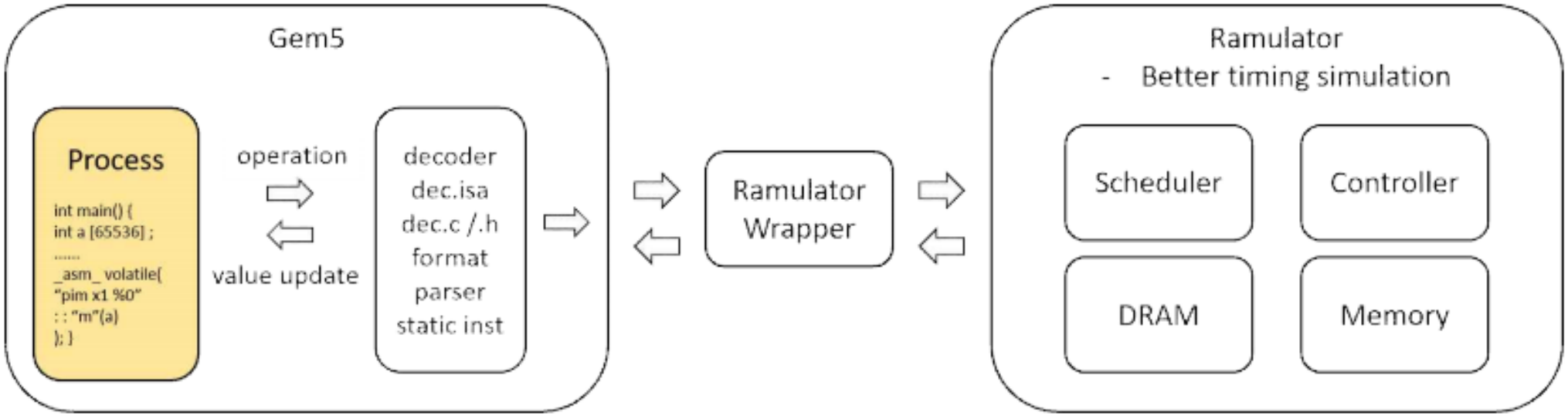
1. Env. Setup (Gem5-Ramulator)

- 최신 버전의 Gem5는 Ramulator와 호환이 잘 되지 않아서 이전 버전의, Gem5와 Ramulator가 integrate 되어있는 git repository를 clone 했습니다. Gem5의 build에 요구되는 조건과 ramulator에 요구되는 조건이 상이하므로, Ramulator를 위한 package는 cluster0 서버에, Gem5-ramulator를 위한 package는 cluster1 서버에 설치했습니다.

```
1. Dependent package 설치
cluster01:~ $ sudo apt install build-essential git m4 scons zlib1g zlib1g-dev ₩
libprotobuf-dev protobuf-compiler libprotoc-dev libgoogle-perftools-dev ₩
python3-dev python-is-python3 libboost-all-dev pkg-config

2, 설치 & build
cluster01:~ $ git clone https://github.com/Kirrito-k423/gem5.git
cluster01:~/gem5$ cd gem5
cluster01:~/gem5$ export C=clang
cluster01:~/gem5$ export C++=clang++
cluster01:~/gem5$ scons build/{ISA}/gem5.{variant} – j {cpus}
```

1. Env. Setup (Gem5-Ramulator)

- Wrapper로 연결된 Gem5와 Ramulator는 다음과 같은 구조를 갖습니다. Gem5에서 Wrapper로 memory request를 주면, wrapper에서 Ramulator로 request를 전달하고, Ramulator에서는 해당 request에 해당하는 timing 정보를 업데이트 하여 response로 Gem5에 돌려줍니다. 이를 위해 Gem5와 Ramulator에서 모두 PIM instruction을 인식하고 처리할 수 있도록 코드를 수정할 필요가 있습니다.



1. Env. Setup (RISCV cross compiler)

- Cross compiler는 gem5에서 수행할 binary를 만드는데 사용하므로, gem5-ramulator가 설치된 cluster 1에 설치했습니다. 본 프로젝트에서는 RISCV architecture를 사용하므로 RISCV cross compiler이며, pim instruction을 compile할 수 있도록 수정할 예정입니다.

1. Dependent package 설치

```
cluster01:~ $ sudo apt-get install autoconf automake autotools-dev curl python3 libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev device-tree-compiler
```

2, 설치 & build

```
cluster01:~ $ git clone --recurse-submodules https://github.com/riscv/riscv-gnu-toolchain.git
cluster01:~ $ git clone https://github.com/riscv/riscv-opcodes
cluster01:~ $ cd riscv-gnu-toolchain
cluster01:~/riscv-gnu-toolchain $ ./configure --prefix={custom compiler 설치할 절대 경로}
cluster01:~/riscv-gnu-toolchain $ make -j 32
```

2. Custom instruction to Compiler

2. Custom instruction to Compiler

- Cross compiler에 custom instruction을 추가하기 위해, mask와 match bit을 할당받아야 합니다. 이를 위해 riscv-opcodes/ directory에서 instruction을 추가할 class를 선택하고 make 합니다. 제가 구현하려는 pim instruction은 메모리 동작에 있어서 store instruction(sd)과 비슷하다고 생각했기 때문에 riscv-opcodes/rv64_i에서 아래 줄을 추가해주었습니다.

```
23 pim     imm12hi rs1 rs2 imm12lo 14..12=0 6..2=0x02 1..0=3
```

- 위와 같이 추가한 다음 make를 수행하고, encoding.out.h 파일 내에서 “PIM”을 찾으면, 아래와 같이 정의된 MATCH_PIM과 MASK_PIM의 definition을 찾을 수 있습니다.

```
1350 #define MATCH_PIM 0xb
1351 #define MASK_PIM 0x707f
```

2. Custom instruction to Compiler

- 앞서 확인한 MASK와 MATCH 값을 riscv-gnu-toolchain/binutils/include/opcode/riscv-opc.h 에 추가하고(2785, 2786 lines), DECLARE_INSN(pim, MATCH_PIM, MASK_PIM) 라인을 추가합니다(2789 lines).

```
2785 #define MATCH_PIM 0xb
2786 #define MASK_PIM 0x707f
2787 #endif /* RISCV_ENCODING_H */
2788 #ifdef DECLARE_INSN
2789 DECLARE_INSN(pim, MATCH_PIM, MASK_PIM)
```

2. Custom instruction to Compiler

- riscv-gnu-toolchain/binutils/opcodes/riscv-opc.c에 다음 라인을 추가합니다(314 line).

```
313 /* name, xlen, isa, operands, match, mask, match_func, pinfo.  */
314 {"pim", 0, INSN_CLASS_I, "t,q(s)", MATCH_PIM, MASK_PIM, match_opcode, I
    NSN_DREF|INSN_8_BYTE},
```

- name: custom instruction의 이름으로, 여기서는 “pim”입니다.
- xlen: 사용하는 register의 bit을 의미합니다. 64bit = 64, 32bit = 32, 둘 다 사용시 = 0.
- isa: 여기서는 I type instruction으로 추가했으므로, INSN_CLASS_I를 주었습니다.
- operands: t는 source2 register, q(s)는 source 1 register 값에 상수값을 더해서 가져오겠다는 의미로 사용했습니다.

2. Custom instruction to Compiler

- 이전 슬라이드와 같이 custom instruction에 대한 정보를 추가한 다음, compiler를 make 하여 build 하였고, 이를 사용할 때는 아래와 같은 커맨드를 사용해서 컴파일할 수 있습니다.

Compile command

```
cluster01:~ $ {설치경로}/riscv_custom/bin/riscv64-unknown-elf-gcc {target} -o {result}
```

3. Custom instruction to Ramulator

3. Custom instruction to Ramulator

- Custom instruction을 Ramulator에서 parsing하고, 이에 맞는 timing simulation을 수행하기 위한 작업입니다. gem5-ramulator가 integrate 파일에서 custom instruction을 디버깅하면서 프로그래밍 해도 되지만, Ramulator에서만 custom instruction을 먼저 넣어보는 것이 ramulator에서 발생한 에러를 디버깅하고 수정하는데 용이했습니다.
- 검증 방식은 dram trace를 input file로 넣어서 진행하였습니다.
- 따라서 Ramulator에 instruction을 넣는 작업은 cluster00 (ramulator를 위한 환경이 설치되어있음)에서 진행하였습니다.

3. Custom instruction to Ramulator

- dram trace file은 다음과 같이 구성되어있으며, address operation 순서입니다. R은 read, W는 write, P는 pim instruction으로, Ramulator에서 다음의 trace file에서 operation들을 읽어서 latency가 제대로 계산될 수 있도록 해야합니다.

```
1 0x12345680 R
2 0x12345670 W
3 0x4cbd56c0 W
4 0x35d46f00 R
5 0x696fed40 W
6 0x7876af80 R
7 0x10101010 P
8 0x77654320 W
9 0x20202200 R
```

DRAM trace file

- 일차적인 목표는 왼쪽의 파일을 읽어서 설정해준 pim instruction에 소요되는 시간 만큼이 timing simulation에 반영되어 결과가 도출되는 것입니다.
- 위 과정이 성공하면 gem5에서 ramulator로 pim instruction을 보내줄 수 있도록 연결하는 과정을 진행할 것입니다.

3. Custom instruction to Ramulator

- Ramulator에서의 dram simulation 동작 과정을 설명드리면 다음과 같습니다.

1. Main.cpp:: Main.cpp에서 ramulator 실행에 따른 parameter 받기
2. Main.cpp:: run_dramtrace에서 Processor.cpp:: get_dramtrace_request 실행
3. Processor.cpp:: get_dramtrace_request에서 request의 type이 read인지 write인지 판단하고, Main.cpp에서 해당 request의 count 증가.
4. Main.cpp에서 request를 queue에 집어넣고, memory.tick() 실행.
5. Memory.h와 Controller.h의 tick() 함수 실행.
6. 각 tick 함수에서 memory state 및 이와 관련된 timing state update.

3. Custom instruction to Ramulator

- pim instruction은 기본 동작이 write instruction과 매우 유사하므로, 바꿔줘야할 함수들의 전체적인 overview를 보기 위해, 터미널에서 ramulator/src 디렉토리로 이동한 후, grep –r “write”, grep –r “Write”, grep –r “WRITE” 세 명령어를 입력해 write instruction이 어떤 함수를 통해 구현되어있는지 확인합니다. 편의를 위해 Cache가 없는 구조를 상정하면, 아래와 같이 DRAM.h, Processor.cpp, Memory.h, Refresh.cpp, Controller.h에 구현되어 있는 함수들 위주로 확인하면 됩니다. 수정사항 중 중요 부분을 추려 설명드리겠습니다.

```
DRAM.h:         .desc("The sum of read and write requests that are served in
this DRAM element per memory cycle for level " + identifier + "_" + to_strin
g(id))
DRAM.h:         .desc("The average of read and write requests that are served
 in this DRAM element per memory cycle for level " + identifier + "_" + to_s
tring(id))
Processor.h:    // [# of bubbles(non-mem instructions)] [read address(dec or
 hex)] <optional: write address(evicted cacheline)>
Refresh.h:  bool ctrl_write_mode = false;
```

3. Custom instruction to Ramulator

```
22      enum class Type
23      {
24          READ,
25          WRITE,
26          REFRESH,
27          POWERDOWN,
28          SELFREFRESH,
29          PIM,
30          EXTENSION,
31          MAX
```

- 1. Request.h

  가장 먼저 Request.h 파일을 수정합니다. Memory operation request들의 자료구조로, 현재는 request의 type으로 WRITE, READ, REFRESH 등이 저장되어있으므로, PIM을 추가해줍니다.

3. Custom instruction to Ramulator

2. Main.cpp

```
45      int reads = 0, writes = 0, pims = 0, clks = 0;
```

(1) pim instruction의 count를 위한 변수입니다 (simulation result에 기록됩니다.).

```
62              if (!stall){
63                  if (type == Request::Type::READ) reads++;
64                  else if (type == Request::Type::WRITE) writes++;
65                  else if (type == Request::Type::PIM) pims++;
66              }
```

(2) pim instruction이 들어올 때 pim의 count를 증가시켜줍니다.

3. Custom instruction to Ramulator

3. Processor.cpp

get_dramtrace_request 함수의 다음 부분에 dramtrace로부터 ‘P’ 문자를 인식할 수 있도록 다음과 같이 수정합니다. ‘P’를 받으면 현재 memory request를 PIM TYPE으로 넣어줍니다.

```
469     if (pos == string::npos || line.substr(pos)[0] == 'R')
470         req_type = Request::Type::READ;
471     else if (line.substr(pos)[0] == 'W')
472         req_type = Request::Type::WRITE;
473     else if (line.substr(pos)[0] == 'P')
474         req_type = Request::Type::PIM;
475     else assert(false);
476     return true;
```

3. Custom instruction to Ramulator

4. Controller.h

Controller.h는 memory request packet들을 queue에 저장하고, tick마다 적절한 instruction을 뽑아 실행시켜주는 기능을 수행합니다. 현재 request type이 pim이라면, 다른 함수에서 get_queue를 호출 시 pimq에서 enqueue와 pop이 일어날 수 있도록 다음과 같이 수정합니다.

```
365     Queue& get_queue(Request::Type type)
366     {
367         switch (int(type)) {
368             case int(Request::Type::READ): return readq;
369             case int(Request::Type::WRITE): return writeq;
370             case int(Request::Type::PIM): return pimq;
371             default: return otherq;
372         }
373     }
```

3. Custom instruction to Ramulator

4. Controller.h

read request를 수행하는데 write 및 pim instruction이 들어올 때 coherency를 위한 부분입니다. 현재 request type이 READ일 때 write나 pim이 수행중이라면 pending으로 갈 수 있도록 하는 부분으로, pending에 가기 위한 조건에 pim instruction의 수행상태를 추가합니다..

```
383         // shortcut for read requests, if a write to same addr exists
384         // necessary for coherence
385         if (req.type == Request::Type::READ && (find_if(writeq.q.begin()
    , writeq.q.end(),
386                 [req](Request& wreq){ return req.addr == wreq.addr;}) !=
     writeq.q.end()
387                     ||  find_if(pimq.q.begin(), pimq.q.end(), [req](Requ
    est& preq){return req.addr == preq.addr;}) != pimq.q.end()) ){
388             req.depart = clk + 1;
389             pending.push_back(req);
390             readq.q.pop_back();
391         }
```

3. Custom instruction to Ramulator

```
420         /*** 3. Should we schedule writes? ***/
421         if (!write_mode) {
422             // yes -- write queue is almost full or read queue is empty
423             if (writeq.size() > int(wr_high_watermark * writeq.max) || r
    eadq.size() == 0)
424                 write_mode = true;
425         }
426         else {
427             // no -- write queue is almost empty and read queue is not e
    mpty
428             if (writeq.size() < int(wr_low_watermark * writeq.max) && (r
    eadq.size() != 0))
429                 write_mode = false;
430         }
431         if (!pim_mode){
432             if(pimq.size() > 0)
433                 pim_mode = true;
434         }
435         else {
436             if(pimq.size() == 0)
437                 pim_mode = false;
438         }
```

4. Controller.h

현재 작업을 하는 queue가 write queue인지 pim queue인지 판단하는 부분입니다. 저는 뒷 부분 구현의 편의를 위해 write request에 대해서는 write_mode를, pim_mode일 때는 write_mode와 pim_mode가 모두 true가 되도록 설계했습니다.

3. Custom instruction to Ramulator

4. Controller.h

작업할 queue를 정하는데, pim instruction에 우선권을 줘서, pim_mode인지 우선적으로 판별합니다. (이전 슬라이드에 pim과 write를 같이 올렸기 때문에 다음과 같이 구현했습니다.)

```
454         if (!is_valid_req) {
455             queue = !pim_mode ? (!write_mode ? &readq : &writeq): &pimq;
```

Ramulator는 Channel, Rank, BankGroup, Bank 단계로 메모리를 추상화하는데, 메모리의 가장 상위 레벨인 channel단계에서 update가 일어나도록 if문에 PIM 조건을 추가합니다.

```
480         if (req->is_first_command) {
481             req->is_first_command = false;
482             int coreid = req->coreid;
483             if (req->type == Request::Type::READ || req->type == Request
    ::Type::WRITE || req->type == Request::Type::PIM) {
484               channel->update_serving_requests(req->addr_vec.data(), 1,
    clk);
485             }
```

3. Custom instruction to Ramulator

4. Controller.h

PIM instruction일 때 statistics 정보들을 update 해줍니다.

```
511             } else if (req->type == Request::Type::PIM){
512               if (is_row_hit(req)) {
513                   ++pim_row_hits[coreid];
514                   ++row_hits;
515               } else if(is_row_open(req)) {
516                   ++pim_row_conflicts[coreid];
517                   ++row_conflicts;
518               } else {
519                   ++pim_row_misses[coreid];
520                   ++row_misses;
521               }
522               pim_transaction_bytes += tx;
523             }
```

3. Custom instruction to Ramulator

4. Controller.h

cmd_issue_autoprecharge 함수에 다음 부분을 추가합니다.

```
681                 if(num_row_hits == 1) {
682                     if(cmd == T::Command::RD)
683                         cmd = T::Command::RDA;
684                     else if (cmd == T::Command::WR)
685                         cmd = T::Command::WRA;
686                     else if (cmd == T::Command::PIM)
687                         cmd = T::Command::PIMA;
688                     else
```

request의 command가 PIM -> PIMA로 전환될 수 있도록 합니다. 해당 transition의 latency는 구현하고자 하는 메모리 모델이 DDR4라면, DDR4.cpp에서 수정할 수 있습니다. Command에 정의된 state들은 DDR4.h에 정의되어있습니다.

3. Custom instruction to Ramulator

```
310     void tick()
311     {
312         ++num_dram_cycles;
313         int cur_que_req_num = 0;
314         int cur_que_readreq_num = 0;
315         int cur_que_writereq_num = 0;
316         int cur_que_pimreq_num = 0;
317 
318         for (auto ctrl : ctrls) {
319           cur_que_req_num += ctrl->readq.size() + ctrl->writeq.size() + ctr
    l->pending.size() + ctrl->pimq.size();
320           cur_que_readreq_num += ctrl->readq.size() + ctrl->pending.size();
321           cur_que_writereq_num += ctrl->writeq.size();
322           cur_que_pimreq_num += ctrl->pimq.size();
323         }
324         in_queue_req_num_sum += cur_que_req_num;
325         in_queue_read_req_num_sum += cur_que_readreq_num;
326         in_queue_write_req_num_sum += cur_que_writereq_num;
327         in_queue_pim_req_num_sum += cur_que_pimreq_num;
328         bool is_active = false;
329         for (auto ctrl : ctrls) {
330           is_active = is_active || ctrl->is_active();
331           ctrl->tick();
332         }
333         if (is_active) {
334           ramulator_active_cycles++;
335         }
336     }
```

5. Memory.h

tick마다 pim instruction에 의한 pim request number, queue size 등의 state들을 update 합니다.

3. Custom instruction to Ramulator

6. DDR4.h

저는 PiM을 시뮬레이션하기 위한 메모리 모델을 DDR4로 잡았으므로, DDR4.h와 DDR4.cpp를 수정하였습니다. HBM을 사용하신다면, HBM.h와 HBM.cpp를 수정해야 합니다. 앞서 Controller.cpp에서 수정 시 언급하였던 command의 state에 PIM과 PIMA를 추가합니다.

```
42      string command_name[int(Command::MAX)] = {
43          "ACT", "PRE", "PREA",
44          "RD",  "WR",  "RDA",  "WRA",
45          "REF", "PDE", "PDX",  "SRE", "SRX",
46          "PIM", "PIMA"
47      };
```

3. Custom instruction to Ramulator

6. DDR4.h

PIM과 PIMA를 마지막 entry로 추가해주었는데, 이에 대응되는, 해당 command가 적용 될 memory의 level을 정하는 곳입니다. 이 부분은 확실하지 않아서 WR와 WRA를 참고하였는데, 이들이 Level::Column scope로 지정되어있었기 때문에, PIM, PIMA를 위해 마지막에 Level::Column을 두개 추가해주었습니다.

```
49      Level scope[int(Command::MAX)] = {
50          Level::Row,    Level::Bank,   Level::Rank,
51          Level::Column, Level::Column, Level::Column, Level::Column,
52          Level::Rank,   Level::Rank,   Level::Rank,   Level::Rank,
53          Level::Rank,   Level::Column, Level::Column
54      };
```

3. Custom instruction to Ramulator

```
66      bool is_accessing(Command cmd)
67      {
68          switch(int(cmd)) {
69              case int(Command::RD):
70              case int(Command::WR):
71              case int(Command::RDA):
72              case int(Command::WRA):
73              case int(Command::PIM):
74              case int(Command::PIMA):
75                  return true;
76              default:
77                  return false;
78          }
79      }
80
```

6. DDR4.h

command가 PIM과 PIMA인 경우에 대해 access와 close 상태가 true를 출력할 수 있도록 다음과 같이 추가해줍니다.

3. Custom instruction to Ramulator

7. DDR4.cpp

PIM에서 동작하고자 하는 내용은 기본적으로 WR (write)와 동일하므로,

DDR4::init_rowopen()과 rowhit(), init_rereq() 등의 함수에서 PIM에 대한 부분을 WR로 선언되어있는 부분들을 참고하여 그대로 만들어줍니다. 아래 예시를 보면 PIM에 대한 prereq가 WR에서와 동일하게 선언되어있음을 확인할 수 있습니다.

```
139     prereq[int(Level::Rank)][int(Command::WR)] = prereq[int(Level::Rank)][i
    nt(Command::RD)];
140     prereq[int(Level::Bank)][int(Command::WR)] = prereq[int(Level::Bank)][i
    nt(Command::RD)];
141     prereq[int(Level::Rank)][int(Command::PIM)] = prereq[int(Level::Rank)][
    int(Command::RD)];
142     prereq[int(Level::Bank)][int(Command::PIM)] = prereq[int(Level::Bank)][
    int(Command::RD)];
```

3. Custom instruction to Ramulator

7. DDR4.cpp

DDR4.cpp에서 가장 중요한 부분은 init_timing()입니다.

```
260     /*** Channel ***/
261     t = timing[int(Level::Channel)];
262     int nPIM = 50;
263     // CAS <-> CAS
264     t[int(Command::RD)].push_back({Command::RD, 1, s.nBL});
265     t[int(Command::RD)].push_back({Command::RDA, 1, s.nBL});
266     t[int(Command::RDA)].push_back({Command::RD, 1, s.nBL});
267     t[int(Command::RDA)].push_back({Command::RDA, 1, s.nBL});
268     t[int(Command::WR)].push_back({Command::WR, 1, s.nBL});
269     t[int(Command::WR)].push_back({Command::WRA, 1, s.nBL});
270     t[int(Command::WRA)].push_back({Command::WR, 1, s.nBL});
271     t[int(Command::WRA)].push_back({Command::WRA, 1, s.nBL});
272     t[int(Command::PIM)].push_back({Command::PIM, 1, s.nBL});
273     t[int(Command::PIM)].push_back({Command::PIMA, 1, s.nBL});
274     t[int(Command::PIMA)].push_back({Command::PIM, 1, s.nBL});
275     t[int(Command::PIMA)].push_back({Command::PIMA, 1, s.nBL});
```

t에 Channel에 대한 timing 정보를 저장한 상태로 t의 각 entry에 timing parameter들을 추가해주는 방식입니다. 저는 PIM을 수행하는데 30,50,100 cycle이 수행될 경우 performance를 측정할 예정으로, 해당 timing을 위해 nPIM을 선언했습니다.

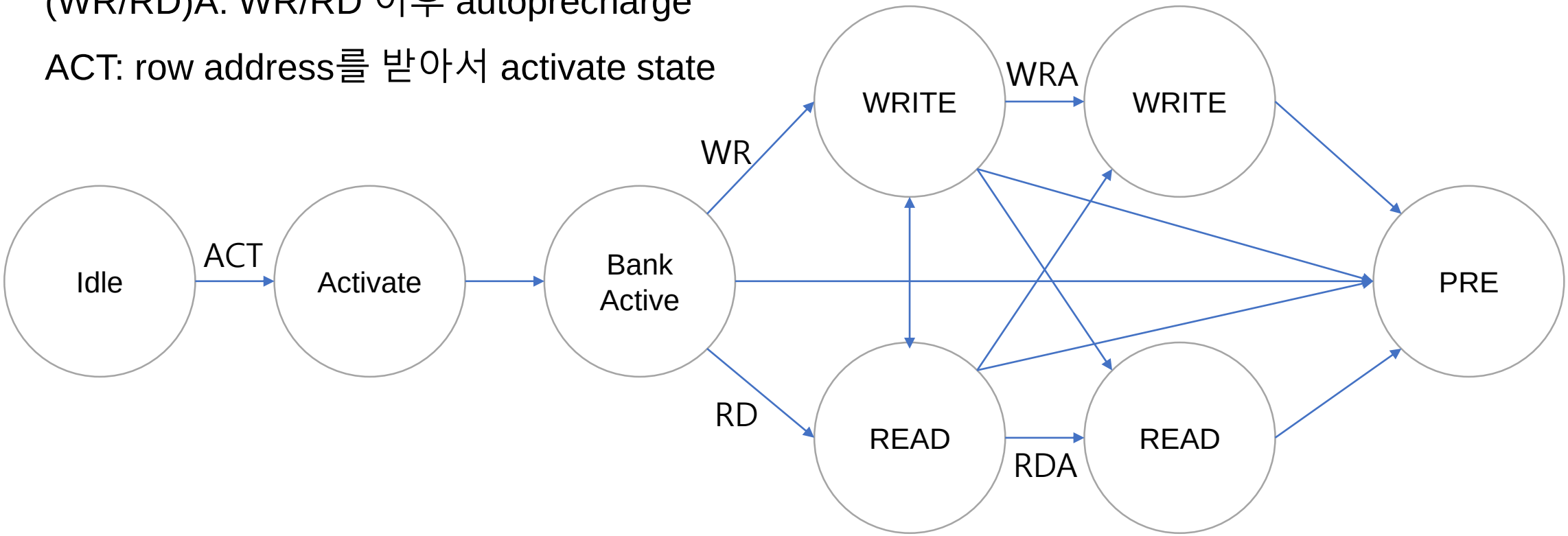
3. Custom instruction to Ramulator

7. DDR4.cpp

DRAM command에 따른 state transistion은 다음과 같은 sequence를 따릅니다.

(WR/RD)A: WR/RD 이후 autoprecharge

ACT: row address를 받아서 activate state

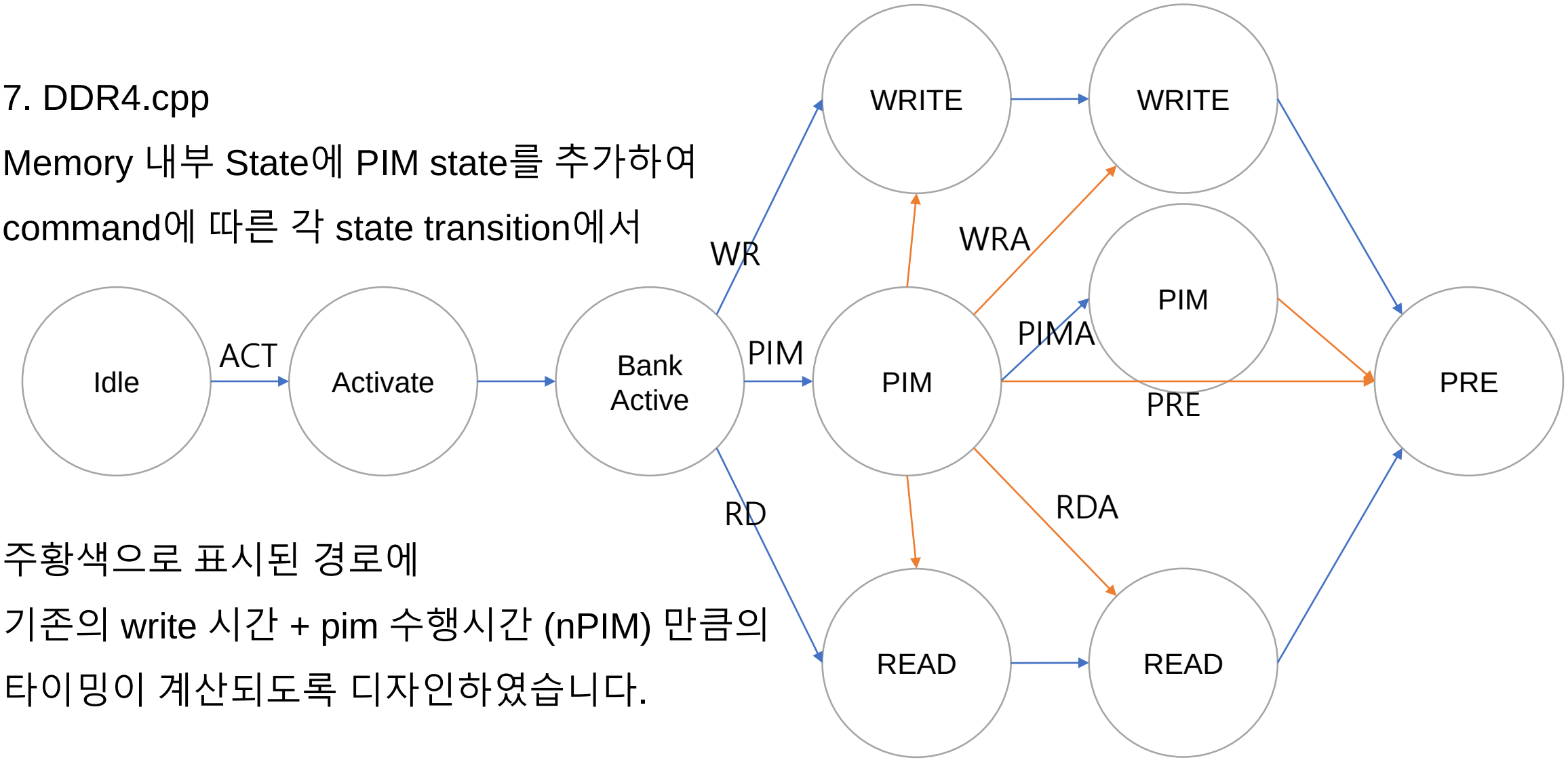


3. Custom instruction to Ramulator

7. DDR4.cpp

Memory 내부 State에 PIM state를 추가하여

command에 따른 각 state transition에서



주황색으로 표시된 경로에

기존의 write 시간 + pim 수행시간 (nPIM) 만큼의

타이밍이 계산되도록 디자인하였습니다.

3. Custom instruction to Ramulator

7. DDR4.cpp

PIM의 수행을 위해 nPIM을 추가한 부분의 예시입니다.

<Bank Level>

```
470     t[int(Command::PIM)].push_back({Command::PRE, 1, s.nCWL + s.nBL + s.nWR
    + nPIM});
471     t[int(Command::RDA)].push_back({Command::ACT, 1, s.nRTP + s.nRP});
472     t[int(Command::WRA)].push_back({Command::ACT, 1, s.nCWL + s.nBL + s.nWR
    + s.nRP});
473     t[int(Command::PIMA)].push_back({Command::ACT, 1, s.nCWL + s.nBL + s.nW
   R + s.nRP + nPIM});
```

3. Custom instruction to Ramulator

8. Others

다른 부분들은 write와 동일한 기능을 수행하도록 write에 대한 함수들을 참고해서 수정해주면 됩니다. 대표적으로, Refresh, Scheduler 모듈이 그러하며, 이외에도 ramulator가 compile되기 위해서는 DDR4 뿐만이 아닌 다른 메모리 모델(DDR3, HBM, etc)들에 대해서도 header 파일의 Command와 command_name, scope에 PIM, PIMA를 추가해줘야 합니다.

4. Custom instruction to Gem5

4. Custom instruction to Gem5

- 지금까지 진행한 것은

  (1) cross compiler에서 pim instruction을 정의해주었고,

  (2) Ramulator에서 dram trace 모드에서 pim instruction을 인식하고 수행하여 timing 정보를 update할 수 있도록 코드를 수정하였습니다.

  이번 슬라이드부터는 Gem5에서 pim instruction을 인식할 수 있도록 하는 과정을 진행하겠습니다.

4. Custom instruction to Gem5

- Gem5에서 instruction을 수행하는 부분은 gem5/src/arch/riscv/isa/ 디렉토리의 decoder.isa 에서 수정할 수 있습니다. 이전에 compiler에 추가한 pim instruction은 아래와 같았습니다.

```
23 pim    imm12hi rs1 rs2 imm12lo 14..12=0 6..2=0x02 1..0=3
```

- 위의 instruction을 수행하기 위해 다음과 같이 추가합니다. COPCODE가 0x3인 상태에서, OPCODE가 0x02이면 Pim format의 instruction을 아래와 같이 수행하라는 뜻입니다.

```
432     0x02: decode FUNCT3{
433         format Pim {
434                 0x0: pim_add({{
435                     Mem_uk = Mem_uk+1;
436                 }});
437         }
```

4. Custom instruction to Gem5

- Pim format의 정의는 arch/riscv/isa/formats/mem.isa에서 다음과 같이 정의해주었습니다.
  기본 코드의 구조는 Store format에 정의된 것과 동일합니다.

```
301 def format Pim(memacc_code, ea_code={{EA = Rs1 + offset;}},
302         offset_code={{offset = sext<12>(IMM5 | (IMM7 << 5));}},
303         mem_flags=[], inst_flags=[]) {{
304     (header_output, decoder_output, decode_block, exec_output) = \
305         LoadStoreBase(name, Name, offset_code, ea_code, memacc_code, mem
    _flags,
306         inst_flags, 'Pim', exec_template_base='Pim')
307 }};
```

4. Custom instruction to Gem5

- Pim format이 호출한 LadStoreBase에서는 PimExecute, PimInitiateAcc, PimCompleteAcc을 호출하기 때문에, 위 함수들을 정의해주어야하는데요, Store과 동일한 동작을 수행하되, flag만 다르게 해서 pim과 store의 timing 구분을 지어줄 것이기 때문에, 함수 내부 구조는 Store의 것을 그대로 따라가면 됩니다. 아래는 PimCompleteAcc의 예시입니다.

```
275 def template PimCompleteAcc {{
276     Fault
277     %(class_name)s::completeAcc(PacketPtr pkt, ExecContext *xc,
278         Trace::InstRecord *traceData) const
279     {
280         return NoFault;
281     }
282 }};
```

4. Custom instruction to Gem5

- PimExecute, PimInitiateAcc는 pimMemTimingLE, pimAtomicLE를 호출하는데, 이는 arch/generic/memhelpers.hh에서 정의해주면 됩니다.

```
266 template <class XC, class MemT>
267 Fault
268 pimMemTimingLE(XC *xc, Trace::InstRecord *traceData, MemT mem, Addr addr
    ,
269                 Request::Flags flags, uint64_t *res)
270 {
271     return pimMemTiming<ByteOrder::little>(
272             xc, traceData, mem, addr, flags, res);
273 }
```

4. Custom instruction to Gem5

- instruction set architecture를 구성하는 일은 끝났고, 이제 cpu에서 지정한 isa를 읽어서 memory request로 보내주는 작업을 수행해야합니다. 이는 gem5/src/cpu/ 디렉토리에서 simple/atomic.hh, simple/base.hh, simple/exec_context.hh, simple/timing.cc를 수정하면 됩니다.

```
169     virtual Fault
170     pimMem(uint8_t* data, unsigned size, Addr addr, Request::Flags flags
    ,
171             uint64_t* res,
172             const std::vector<bool>& byte_enable=std::vector<bool>())
173     {
174         panic("writeMem() is not implemented\n");
175     }
```

4. Custom instruction to Gem5

- 실제 pimMem의 정의는 cpu/simple/timing.cc에서 진행합니다.

```
589 Fault
590 TimingSimpleCPU::pimMem(uint8_t *data, unsigned size,
591                            Addr addr, Request::Flags flags, uint64_t *re
    s,
592                            const std::vector<bool>& byte_enable)
593 {
594     SimpleExecContext &t_info = *threadInfo[curThread];
595     SimpleThread* thread = t_info.thread;
596     uint8_t *newData = new uint8_t[size];
597     const Addr pc = thread->pcState().instAddr();
598     unsigned block_size = cacheLineSize();
599     BaseMMU::Mode mode = BaseMMU::Pim;
```

4. Custom instruction to Gem5

- 마지막으로 cpu/ driectory의 StaticInstFlags.py를 열어 IsPim flag를 추가해줍니다.

```
 94
 95          # hardware transactional memory
 96          'IsHtmStart',        # Starts a HTM transaction
 97          'IsHtmStop',         # Stops (commits) a HTM transaction
 98          'IsHtmCancel',        # Explicitely aborts a HTM transaction
 99          'IsPim'
100          ]
```

4. Custom instruction to Gem5

- gem5/src/mem 디렉토리에서는 packet.hh, abstract_memory.hh/cc를 수정해야하고, ramulator와의 연결을 위해 ramulator.cc 역시 수정해야 합니다.
- 먼저, packet.hh에서 Command에 PimReq, PimResp를 추가하고, Attribute에 IsPim을 추가해줍니다. 그리고 public function으로 isPim을 isWrite과 동일한 구조로 define 합니다.

4. Custom instruction to Gem5

- 다음은 abstract_memory.cc의 수정사항입니다. abstract_memory는 memory 모델이 결정되는 것 상위의 개념으로, 구체적인 메모리 모델에 적용되기 전 범용성 있는 함수들을 모아놓은 모듈입니다. access함수에서 저는 다음과 같이 구성하여 functionality는 write의 functionality를 일부 재사용할 수 있도록 구성했습니다.

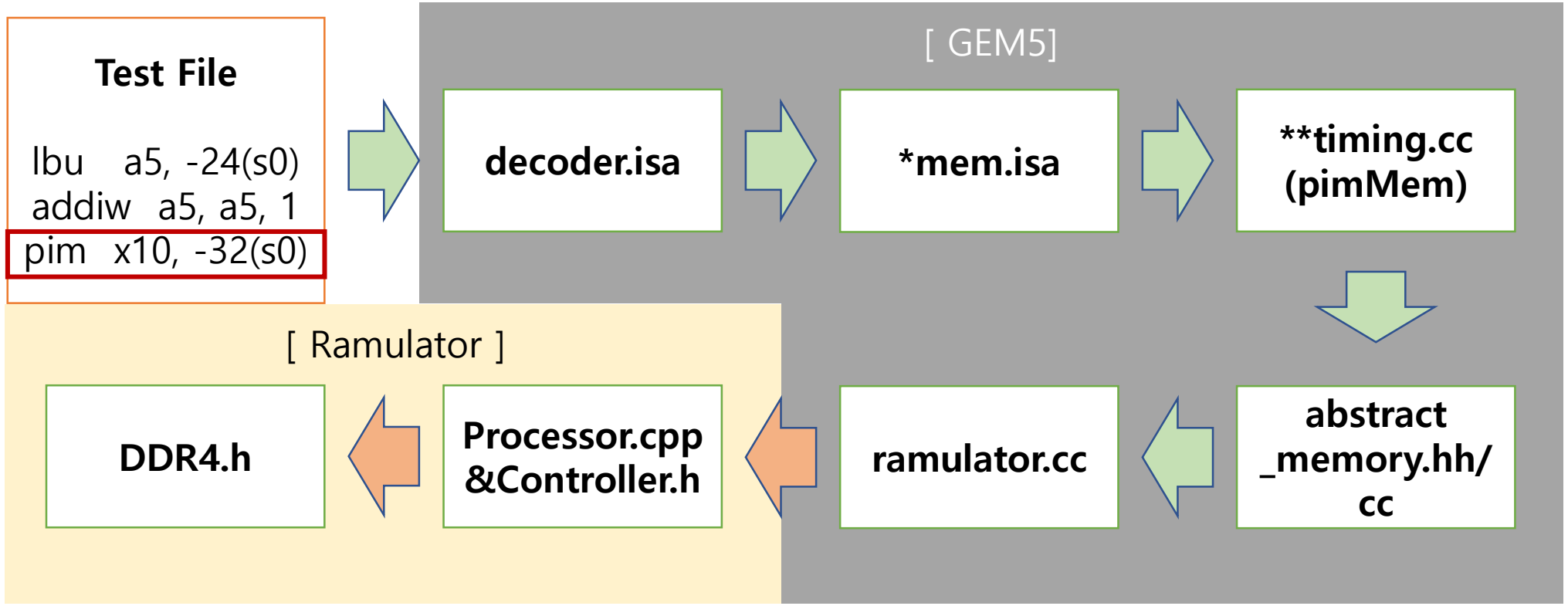
```
459     } else if (pkt->isWrite() || pkt->isPim()) {
460         if (writeOK(pkt)) {
461             if (pmemAddr) {
462                 pkt->writeData(host_addr);
463                 DPRINTF(MemoryAccess, "%s write due to %s\n",
464                         __func__, pkt->print());
465             }
466             assert(!pkt->req->isInstFetch());
467             TRACE_PACKET("Write");
468             stats.numWrites[pkt->req->requestorId()]++;
469             stats.bytesWritten[pkt->req->requestorId()] += pkt->getSize(
    );
470         }
```

4. Custom instruction to Gem5

- ramulator.cc에서는 recvTimingReq에서 ramulator module에 PIM type의 request를 보낼 수 있도록 다음을 추가했습니다. 나머지 부분(172 line 이하) 역시 pkt->isWrite() case에서 선언된 부분을 참고하여 동일한 구조를 따라가되 구현하려는 pim의 functionality에 맞게 수정해주면 됩니다.

```
166     } else if (pkt->isPim()) {
167         // Detailed CPU model always comes along with cache model enable
    d and
168         // write requests are caused by cache eviction, so it shouldn't
    be
169         // tallied for any core/thread
170         ramulator::Request req(pkt->getAddr(),
171                 ramulator::Request::Type::PIM, write_cb_func, 0);
172         accepted = wrapper->send(req);
```

4. Custom instruction to Gem5



*mem_isa: timing.cc에서 사용하는 pimMem의 prototype (pimMemTimingLE, etc) 정의
**timing.cc: decoder에서 받은 request packet으로 packing (buildPacket 함수)

4. Custom instruction to Gem5

- Ramulator와 Gem5의 setting이 완료되었으면 scons build/RISCV/gem5.opt –j 32를 사용하여 RISCV용 gem5를 compile합니다.

5. Evalulation

5. Evaluation

- gem5/ext/ramulator/Ramulator/src/DDR4.cpp에서 정의한 nPIM값을 바꾸면서 evaluation 합니다. pim instruction에 소요되는 latency는 기존 memory의 write instruction의 latency + 30/50/100 cycle이 소요된다고 생각하고 세가지 케이스에 대해 성능 비교를 진행했습니다.
- 30/50/100 cycle에 대한 근거는, DDR4-1600의 clock frequency가 800MHz로 2.0~3.2GHz의 clock frequency를 갖는 cpu clock frequency에 비해 매우 낮으므로, 일반적인 adder를 chip마다 구현한다고 생각하면 adder가 무리 없이 안정적으로 잘 수행될 것이라고 생각하였습니다. 가장 비효율적인 ripple carry adder로 64bit에 대한 연산을 수행한다고 할 때, 64 cycle이 소요되며, 메모리 내에서 데이터 안정화를 위한 작업이 필요하다고 할 때 최대 100 cycle이 걸릴 수 있을 것이라 판단했습니다. 위와 같은 논리로, 최적화 된 adder, 일반적인 adder 등을 사용했을 때 latency를 30, 50으로 생각했습니다.

5. Evaluation

- 다음과 같은 코드를 구성하여 성능을 테스트해보았습니다. 성능 비교의 타겟이 되는 코드는 주어진 코드의 6번째 라인의 수행 시간으로, 이를 위해 세 가지 다른 버전의 코드를 만들어 컴파일하였습니다.

```
1  int main() {
2      int a[65535];
3      for(int I = 0; I < 65535; I ++)
4          a[i] = 0;
5      for(int I = 0; I < 65535; I ++)
6          a[i] += 1;
7      return 0;
8  }
```

5. Evaluation

- 성능 test를 위한 test code는 다음과 같습니다.

  (1) pim instruction을 test합니다. (pim_test)

  (2) 일반적인 memory store와 read하고 +1 instruction을 수행합니다. (mem_test)

  (3) for loop 안에서 아무런 instruction을 수행하지 않습니다. (nop_test)

```c
#include <stdio.h>

int main(){
    int a[65535];
    for(int i = 0; i < 65535; i ++){
        a[i] = 0;
    }
    printf("value stored in a before ++: %d", a[0]);
    for(int i = 0; i < 65535; i ++){
        __asm__ volatile (
            "pim x10, %0"
            : : "m" (a[i])
        );

    }
    printf("value stored in a after ++: %d", a[0]);



    return 0;
}
```

pim_test

```c
#include <stdio.h>

int main(){
    int a[65535];
    for(int i = 0; i < 65535; i ++){
        a[i] = 0;
    }
    printf("value stored in a before ++: %d", a[0]);
    for(int i = 0; i < 65535; i ++){
        a[i]++;
    }
    printf("value stored in a after ++: %d", a[0]);



    return 0;
}
```

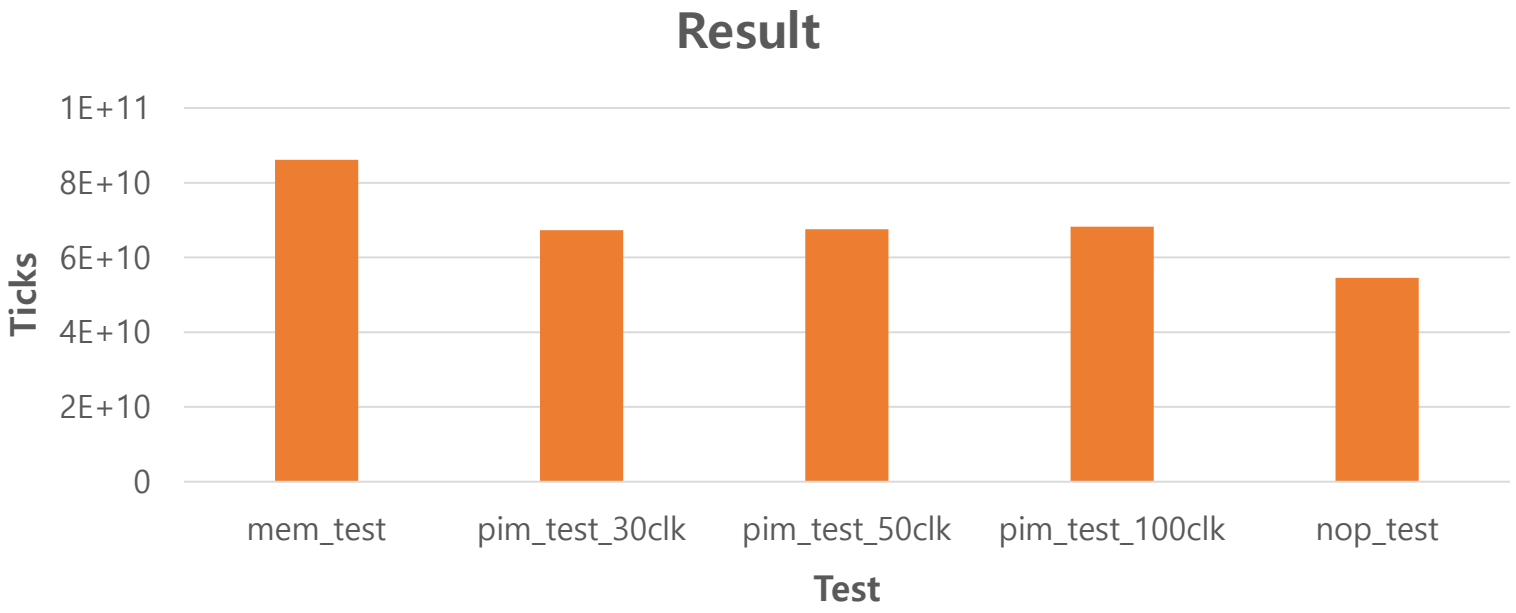
mem_test

```c
#include <stdio.h>

int main(){
    int a[65535];
    for(int i = 0; i < 65535; i ++){
        a[i] = 0;
    }
    printf("value stored in a before ++: %d", a[0]);
    for(int i = 0; i < 65535; i ++);
    printf("value stored in a after ++: %d", a[0]);



    return 0;
}
```

nop_test

5. Evaluation

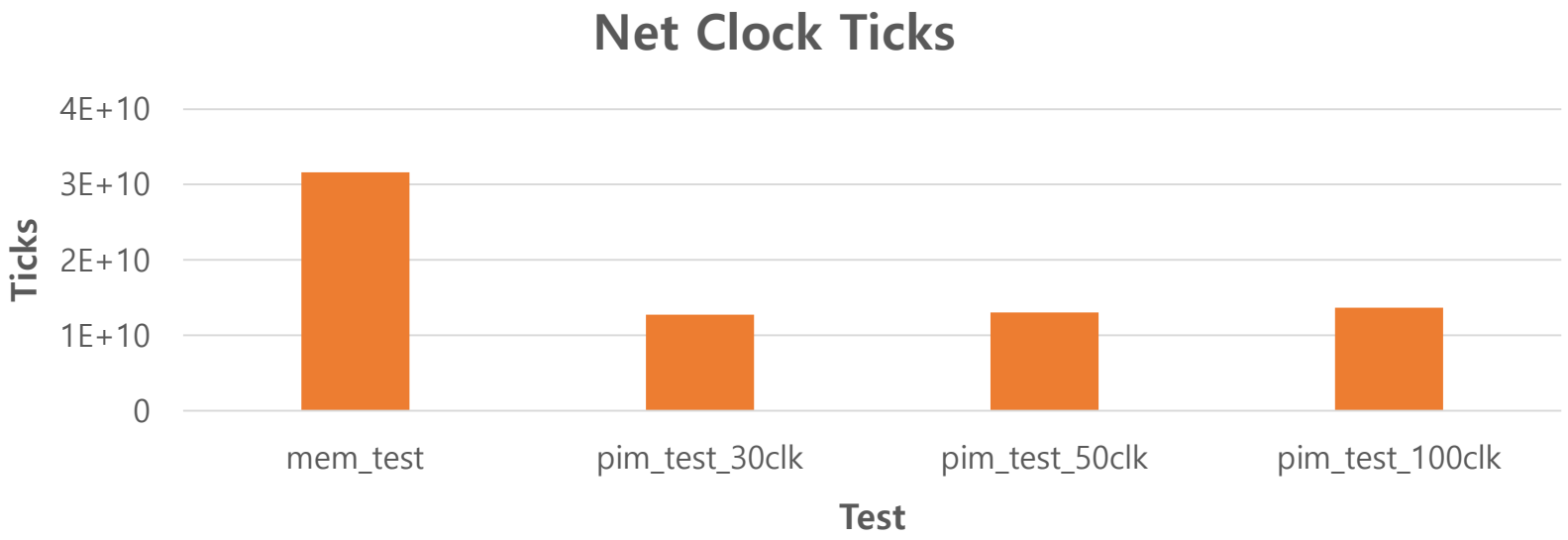
- memory에서 일반적인 memory instruction (a[i]++)의 수행과 pim instruction의 수행 (pim_test_30,50,100clk), memory와 pim instruction으로 비교하고자 하는 부분을 제외하여 아무런 instruction을 수행하지 않을 때(nop_test)의 결과를 아래와 같이 표현하였습니다.



	Result (tick)
mem_test	86165626000
pim_test_30clk	67268447000
pim_test_50clk	67562127000
pim_test_100clk	68200102000
nop_test	54534999000

5. Evaluation

- mem_test와 pim_test에서 nop_test의 결과를 빼주어, 비교하고자 하는 구간의 수행 결과 비교입니다. target operation을 수행하는데 있어서 pim_test_100clk와 mem_test의 결과를 비교했을 때 약 131%의 performance 향상을 확인할 수 있습니다. 또한 pim_test_30_clk로 mem_test 결과를 비교하면 약 148%의 performance 향상을 확인할 수 있습니다.



	Result (tick)
mem_test	31630627000
pim_test_30clk	12733448000
pim_test_50clk	13027128000
pim_test_100clk	13665103000

5. Evaluation

- 본 프로젝트에서는 pim instruction을 설계할 때 메모리의 병렬성을 고려하지 않았으나, pim instruction을 설계할 때 메모리의 병렬성을 고려하여 설계한다면 더 높은 performance gain을 얻을 수 있을 것으로 보입니다. 예를 들어, 다음을 고려해볼 수 있습니다.
  1. Near memory processing을 진행한다면, for문을 통해 연속된 메모리의 값들을 1씩 증가시키는 연산을 할 때, 한번에 row buffer에 올릴 수 있는 단위만큼의 데이터 값들을 한번에 1씩 증가시켜주는 연산을 수행하는 pim instruction의 설계
  2. Processing in chip을 진행한다면, pim instruction이 한 address를 보내는 것이 아니라, address의 범위를 보냄으로써 해당 address 범위에 속한 memory 저장값들에 대해 연산을 진행하는 방식을 생각해볼 수 있습니다.