SPIKE Simulator는 어떻게 작동하는가?

핵심적인 동작은 총 4개의 file을 참고해야 한다고 판단했다.

execute.cc, processor.cc, mmu.cc 이다.

- Execute.cc

  핵심함수는 step이다.

- processor_t::step(size_t n)

```
#define advance_pc() \
 if (unlikely(invalid_pc(pc))) { \
   switch (pc) { \
     case PC_SERIALIZE_BEFORE: state.serialized = true; break; \
     case PC_SERIALIZE_AFTER: ++instret; break; \
     case PC_SERIALIZE_WFI: n = ++instret; break; \
     default: abort(); \
   } \
   pc = state.pc; \
   break; \
 } else { \
   state.pc = pc; \
   instret++; \
 }
```

  advance_pc()로, program counter를 advance (+4 or sth.else) 시켜서 다음 instruction에 접근가능하게끔 도와주는 함수이다.

```
if (unlikely(slow_path()))
{
  // Main simulation loop, slow path.
  while (instret < n)
  {
    if (unlikely(!state.serialized && state.single_step == state.STEP_STEPPED)) {
      state.single_step = state.STEP_NONE;
      if (!state.debug_mode) {
        enter_debug_mode(DCSR_CAUSE_STEP);
        // enter_debug_mode changed state.pc, so we can't just continue.
        break;
      }
    }

    if (unlikely(state.single_step == state.STEP_STEPPING)) {
      state.single_step = state.STEP_STEPPED;
    }

    insn_fetch_t fetch = mmu->load_insn(pc);
    if (debug && !state.serialized)
      disasm(fetch.insn);
    pc = execute_insn(this, pc, fetch);
    advance_pc();
  }
}
```

slow_path()에 해당된다면, (unlikely(x)는 (x)로 정의가 되어있으며,

```
bool processor_t::slow_path()
{
  return debug || state.single_step != state.STEP_NONE || state.debug_mode;
}
```

slow_path()도

로 정의가 되어있다)

fetch에 load_insn(pc)로 decode_insn(insn)까지 타고 들어가게 되어 decode_insn()을 실행한다.

fast_path()에 해당한다면, _mmu -> accesss_icache(pc) 에서 얻어온 icache에 대해서 decode_insn()을 수행하는데, 이는 debug모드를 켜지 않았을 때 fast_path()만을 타고 검사하게 되며,

slow_path()에 해당한다면 모든 instruction이 fetch되어 _mmu->load_insn()되어 decode_insn()을 통해 수행된다.

이 과정에서 trap (pagefault, ecall 등 exception)이 발생하낟면 밑의 take_trap()에 걸려 수행되는 것이다.

이 때, fast_path()와 slow_path() (slow_path()는 debug모드를 켰을 때 slow_path로 작동한다.) 의 코드 수가 다르다. fast_path()에선 무시되는 instruction이 존재한다고 가정, debug 모드로 코드를 돌려봤더니 다음과 같았다.

```
: until pc 0 0x10178
bbl loader
:
1
core   0: 0x0000000000010178 (0x06c000ef) jal     pc + 0x6c
:
2
core   0: 0x00000000000101e4 (0xff010113) addi    sp, sp, -16
:
3
core   0: 0x00000000000101e8 (0x00113423) sd      ra, 8(sp)
:
4
core   0: 0x00000000000101ec (0x00813023) sd      s0, 0(sp)
:
```

다음과 같이 debugmode에서 main 안의 instruction 개수를 센다.

```
35060
core   0: 0x0000000000010210 (0x01010113) addi    sp, sp, 16
35061
core   0: 0x0000000000010214 (0x00008067) ret
:
```

사용한 command는 untiln을 썼고, 35061줄이 출력된다.

하지만 debug mode 없이 main 내에서의 instruction 개수를 세면

```
3859 - 0x0000000000010434 (0x06013403) ld      s0, 96(sp)
3860 - 0x0000000000010438 (0x05813483) ld      s1, 88(sp)
3861 - 0x000000000001043c (0x00078513) mv      a0, a5
3862 - 0x0000000000010440 (0x07010113) addi    sp, sp, 112
3863 - 0x0000000000010444 (0x00008067) ret
3864 - 0x0000000000010200 (0x00000793) li      a5, 0
3865 - 0x0000000000010204 (0x00078513) mv      a0, a5
3866 - 0x0000000000010208 (0x00813083) ld      ra, 8(sp)
3867 - 0x000000000001020c (0x00013403) ld      s0, 0(sp)
3868 - 0x0000000000010210 (0x01010113) addi    sp, sp, 16
3869 - 0x0000000000010214 (0x00008067) ret
```

이 출력된다.

따라서 debugmode 없는 fast_path가 확실히 빠르게 instruction을 처리한다고 볼 수 있다.

이에 모든 instruction을 slow_path로 돌아가게끔 정의해뒀는데, 이는 추후에 서술하겠습니다.

- mmu.cc

  step 의 slow_path → load_insn() → refill_icache → decode_insn()으로 연결, step 의 fast_path → access_icache → refill_icache → decode_insn()으로 연결.

  결국 refill_icache 의 동작을 파악해야한다.

- refill_icache

```
inline icache_entry_t* refill_icache(reg_t addr, icache_entry_t* entry)
{
  auto tlb_entry = translate_insn_addr(addr);
  insn_bits_t insn = from_le(*(uint16_t*)(tlb_entry.host_offset + addr));
  int length = insn_length(insn);
```

  insn 의 endian 변환 (from_le) 를 시행한 후, insn 의 bits 에 관한 동작들을 해준 뒤

```
    insn_fetch_t fetch = {proc->decode_insn(insn), insn};
```

  decode_insn()을 시행하는게 핵심이다.

  결국 insn 을 사용할 수 있는 꼴로 잘 trim 한 후, decode_insn()으로 보내는 게 이 함수의 핵심이다.

  그럼 decode_insn()을 확인해보자.

- processor.cc

  decode_insn(), take_trap()이 핵심 함수이다.

- decode_insn()

```
size_t idx = insn.bits() % OPCODE_CACHE_SIZE;
insn_desc_t desc = opcode_cache[idx];
```

  주어진 instruction 의 bit 를 hash_map 에서 쓸 수 있게 idx 로 변환하며,

```
      opcode_cache[idx] = desc;
      opcode_cache[idx].match = insn.bits();
    }

    return xlen == 64 ? desc.rv64 : desc.rv32;
  }
```

  linear search 까지 해가며 얻어낸 instruction 을 실행할 수 있게 실행정보를 반환하는 함수이다.

  이 함수에서 중요한 요소는 다른 함수를 call 해서 그 함수를 사용한다는 점이 아닌,

  위에서 봤듯 step()이 걸림에 따라 모든 instruction 이 decode_insn()을 거친다는 것이다.

- take_trap

```
void processor_t::take_trap(trap_t& t, reg_t epc)
{
  if (debug) {
    std::stringstream s; // first put everything in a string, later send it to output
    s << "core " << std::dec << std::setfill(' ') << std::setw(3) << id
      << ": exception " << t.name() << ", epc 0x"
      << std::hex << std::setfill('0') << std::setw(max_xlen/4) << zext(epc, max_xlen) << std::endl;
    if (t.has_tval())
       s << "core " << std::dec << std::setfill(' ') << std::setw(3) << id
         << ":           tval 0x" << std::hex << std::setfill('0') << std::setw(max_xlen/4)
         << zext(t.get_tval(), max_xlen) << std::endl;
    debug_output_log(&s);
  }
```

trap t 를 catch 했다면 trap 이 발생했다고 inform 하며 그 정보를 print 한다.

이 함수는 instruction 이 진행됨에 따라 trap 이 발생했을 때 우리가 알 수 있게끔 해준다.

결국 전체적인 flow 는,

step()함수가 실행됨에 따라서, fast_path 또는 slow_path 로 instruction 이 구분되어 들어가며,

각 path 에 따라 실행되는 instruction 의 수와 종류가 조금씩 달라지지만,

각 instruction 은 결국 decode_insn()을 거쳐간다는 게 간단한 flow 설명이다.

그러면 step()함수가 어디서 실행되냐, 하면 step()함수는

```
void sim_t::main()
{
  if (!debug && log)
    set_procs_debug(true);

  while (!done())
  {
    if (debug || ctrlc_pressed)
      interactive();
    else
      step(INTERLEAVE);
    if (remote_bitbang) {
      remote_bitbang->tick();
    }
  }
}
```

sim_t 의 main 함수에 정의되어 있다.

FLOW 정리

sim_t 의 main 함수가 실행이 된다.

debug 모드가 아닌 log 모드 (-l option)이 정의되어 있다면 set_procs_debug(true)를 세팅,

debug 모드가 정의되어 있다면 interactive()를 실행하며,

(interactive()는 interactive.cc 에 정의되어 있으며, -d 모드 전용 옵션이다.

until, while, r 등 -d 모드에서 입력할 수 있는 모든 커맨드를 구현해놨다.)

debug 모드도 아니고, -l option 도 안 켜져 있다면 step(INTERLEAVE)가 실행된다. (INTERLEAVE 는 5000 으로 정의되어 있고, 5000 회의 step 이 끝나도 !done()일 때 STEP 은 계속 진행된다)

STEP 이 실행되면, 현재 실행방식이 slow_path()에 해당하는지 아닌지를 판단한 후,

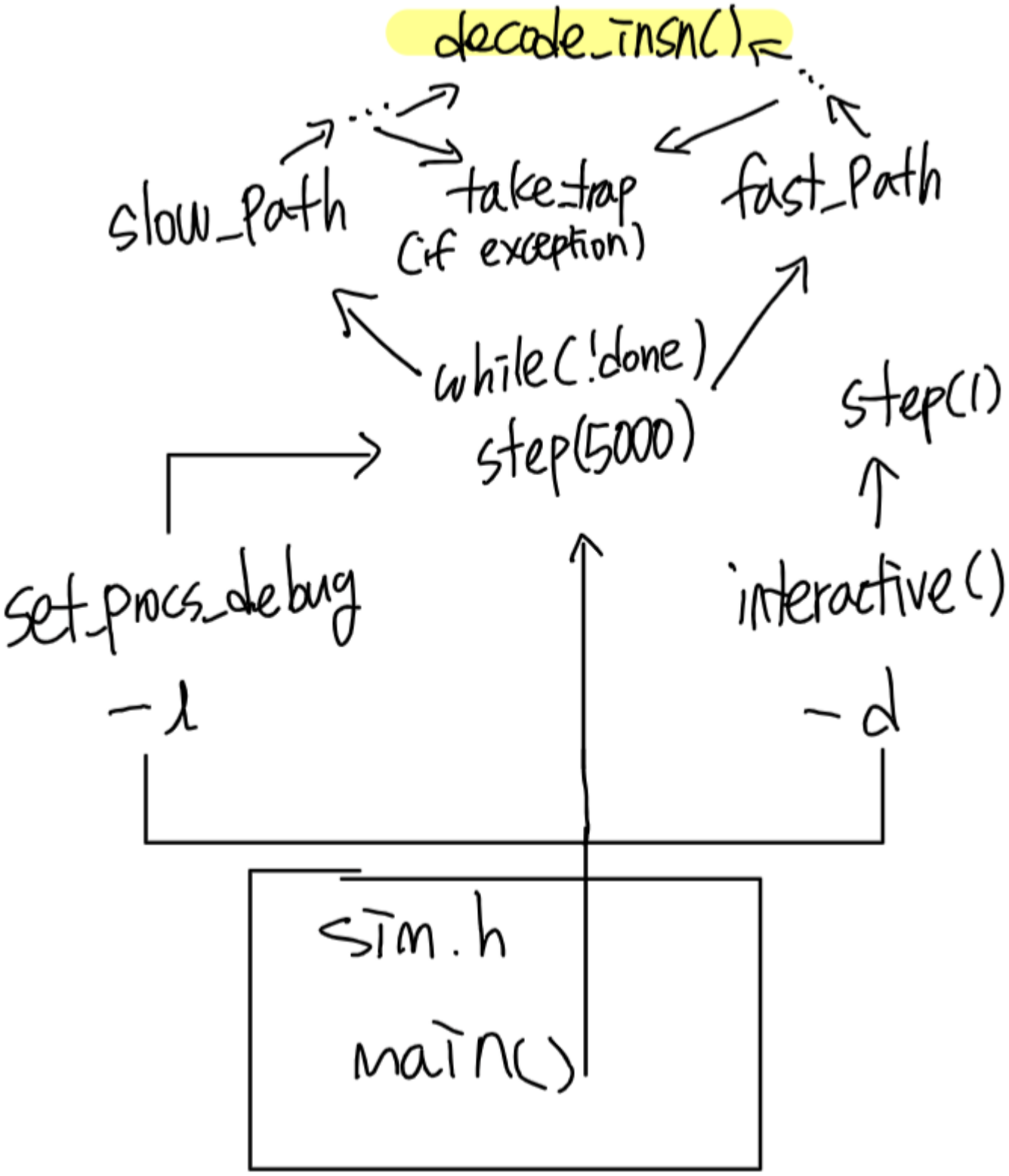
slow_path()에 해당하면 step -> load_insn() -> refill_icache() -> **decode_insn()**을 거친다

fast_path()에 해당하면 step -> access_icache() -> refill_icache() -> **decode_insn()**을 거친다.

이 과정에서 exception 이 발생하면 step 에서 catch 된 trap 이 take_trap()을 발생시킨다.

그렇게 모든 instruction 을 실행하면 코드가 종료된다.

개요도가 밑의 그림에 정리되어있다.


decode_insn()
slow_path
take_trap
(if exception)
fast_path
while(!done)
step(5000)
step(1)
set_procs_debug
-l
interactive()
-d
sim.h
main()

Modifications

1. execute.cc 의 step()에서, fast_path()로 향하는 경우 instruction 이 생략되어 계산된다고 판단, 모든 instruction 이 실행되는 slow_path()로 fast_path()를 병합했다.

```
if (unlikely(slow_path()))
{
  // Main simulation loop, slow path.
  while (instret < n)
  {
    if (unlikely(!state.serialized && state.single_step == state.STEP_STEPPED)) {
      state.single_step = state.STEP_NONE;
      if (!state.debug_mode) {
```

원래 코드. slow_path()일 시 밑의 block 이 실행된다.

```
else while (instret < n)
{
  // Main simulation loop, fast path.
  for (auto ic_entry = _mmu->access_icache(pc); ; ) {
    auto fetch = ic_entry->data;
```

원래 코드. else (fast_path())일 시 밑의 block 이 실행된다.

```
//if (unlikely(slow_path())) //combine fast path with slow path. every instructions are considered slow
if (1)
{
  // Main simulation loop, slow path.
  while (instret < n)
  {
    if (unlikely(!state.serialized && state.single_step == state.STEP_STEPPED)) {
      state.single_step = state.STEP_NONE;
```

Modified code. 항상 블록이 실행된다.

```
}
/* // eliminate fastpath. all instructions are considered slow
else while (instret < n)
{
  // Main simulation loop, fast path.
  for (auto ic_entry = _mmu->access_icache(pc); ; ) {
```

fast_path()의 part 는 전체 주석처리 했다. 항상 slow_path()의 code 가 실행된다.

(이 실행을 위해 slow_path()와 fast_path()가 겹치지 않음을 검증했다.)

2. processor_cc 의 decode_insn()은 instruction 을 decode 하는 함수이며 큰 기능이 없지만, 모든 instruction 이 decode_insn()을 거쳐가는 말단 function 의 역할을 한다.

   이에, decode_insn()에서 거쳐가는 모든 instruction 의 opcode 와 정보를 알 수 있음을 확인하여, decode_insn() 내부에 instruction 을 판단하여 cycle 을 계산하는 코드를 implement 했다.

```
insn_func_t processor_t::decode_insn(insn_t insn)
{
  // look up opcode in hash table
  size_t idx = insn.bits() % OPCODE_CACHE_SIZE;
  insn_desc_t desc = opcode_cache[idx];
```

원래 코드.

```
uint64_t bits = insn.bits() & ((1ULL << (8 * insn_length(insn.bits()))) - 1);
if (zext(state.pc, max_xlen) == 0x10178) // main started. from next instruction, every instructions are valid.
{
        validinstr = false;
        valid_from_next = true;
        inside_main = true;
}
if (zext(state.pc, max_xlen) == 0x1017C) // main finished. valid or not, every instructions shouldn't be counted.
{
        validinstr = false;
        inside_main = false;
        std::cout << std::dec << cnt+4 << std::endl;
        // cnt is the cnt of 'instructions', but I mixed it with 'cycles' in the counting process.
        // since one instruction has 5 counts, cnt + 4
}

if ((bits & 0x7F) == 0x73)
{
      if (bits != 0x00000073) validinstr = false; // if exceptions, don't print. but, print ecall.
      else
      {
              //std::cout << "ecall" << std::endl;
              validinstr = true;
      }
}
if (bits == 0x10200073) valid_from_next = true;
```

Modified code (1)

원래 decode_insn()의 code 는 수정하지 않았다. 단, decode_insn()의 원래 코드가 실행되기 전 cycle 을 판별하는 코드를 추가했을 뿐이다.

이 Screenshot 은, code 의 bits 를 뜯어와 저장한 후, pc 가 0x10178 (main 함수 시작) 일 때부터

pc 가 0x1017c (main 함수 끝) 일 때까지 instruction 들에 대한 cycle 계산을 취한다.

물론 PC 에 대한 비교를 해서, PC 의 opcode 가 0x73 이라면, (ecall 인 경우 또는 csrrw 인 경우 (pagefault))

ecall 일 시 valid, ecall 이 아닐 시 invalid instruction 으로 취급한다.

```
if (validinstr && inside_main)
{
        /*
        std::cout << std::dec << ++cnt << std::hex << " - 0x" << std::setfill('0') << std::setw(max_xlen/4)
        << zext(state.pc, max_xlen) << " (0x" << std::setw(8) << bits << ") "
        << disassembler->disassemble(insn) << std::endl;
        //This for-debug code is implemented from disasm.
        */
        ++cnt;
        if (prevcnt && prevprevcnt)
        {
                auto prevbits = prev.bits() & ((1ULL << (8 * insn_length(prev.bits()))) - 1);
                auto prevprevbits = prevprev.bits() & ((1ULL << (8 * insn_length(prev.bits()))) - 1);
                if (prevbits == bits)
                {
                        //std::cout << "pagefault" << std::endl;
                        cnt--; // since pagefault, prev instr == now instr, so don't count as new instruction
                }
                if ((prevprevbits & 0x7F) == 0x3) // prevprev instruction is load
                {
                        if ((bits&0x7F) == 0x63) // now inst is branch
                        {
                                if (prevprev.rd() == insn.rs1() || prevprev.rd() == insn.rs2())
                                {
                                        cnt++;
                                }
                        }
                }
```

valid 한 instruction (main 함수 내의 instruction + exception 이 아님)인 경우,

```
if ((prevbits & 0x7F) == 0x3) // prev instruction is load .. we're searching for load-use hazards
{
        if ((bits & 0x7F) == 0x63) // if now inst is branch
        {
               if (prev.rd() == insn.rs1() || prev.rd() == insn.rs2())
               {
                      cnt += 2;
               }
        }
        else if ((bits & 0x7F) == 0x33 || (bits & 0x7F) == 0x3B) // now inst is R-Type (including RV-64I)
        {
                if (prev.rd() == insn.rs1() || prev.rd() == insn.rs2())
                {
                        cnt++;
                }
        }
        else if ((bits & 0x7F) == 0x13 || (bits & 0x7F) == 0x1B) // now inst is I-type (including RV-64I)
        {
```

```
}
else if ((prevbits & 0x7F) == 0x63) // prev instruction is branch
{
        if (prev_pc + 4 != zext(state.pc, max_xlen)) // if branch is taken
        {
                cnt++;
        }
}
else if ((prevbits & 0x7F) == 0x6F) // prev instruction is jal
{
        if (prev_pc + 4 != zext(state.pc, max_xlen))
        {
                cnt++;
        }
}
else if ((prevbits & 0x7F) == 0x67 && ((prevbits>>11) & 0x7) == 0) // prev inst is jalr
{
        if (prev_pc + 4 != zext(state.pc, max_xlen))
        {
                cnt++;
        }
}
else
{
        if ((bits & 0x7F) == 0x63) // now branch
        {
                if ((prevbits & 0x7F) == 0x33 || (prevbits & 0x7F) == 0x13
                                || (prevbits & 0x7F) == 0x1B || (prevbits & 0x7F) == 0x3B) // prev execute -> use
                {
                        if (prev.rd() == insn.rs1() || prev.rd() == insn.rs2())
                        {
                                cnt++;
                        }
                }
        }
}
```

LOAD-USE, BRANCH prediction, Jump 로 인한 stall, Use-branch 로 인한 stall 등을 계산하는 코드를 삽입했다.

그 외

```
void processor_t::disasm(insn_t insn)
{
  uint64_t bits = insn.bits() & ((1ULL << (8 * insn_length(insn.bits())))-1);
  if (last_pc != state.pc || last_bits != bits) {
    std::stringstream s;  // first put everything in a string, later send it to output

#ifdef RISCV_ENABLE_COMMITLOG
    const char* sym = get_symbol(state.pc);
    if (sym != nullptr)
    {
      s << "hello1 " << std::dec << std::setfill(' ') << std::setw(3) << id
        << ": >>>>  " << sym << std::endl;
    }
#endif

    if (executions != 1) {
      s << "hello2 " << std::dec << std::setfill(' ') << std::setw(3) << id
        << ": Executed " << executions << " times" << std::endl;
    }
```

어느 부분에서 disasm 의 output 이 실행되는지 확인하기 위해 core 를 hello1, hello2 로 바꿔 디버깅할 때 사용한 수정 코드 등,

전반적인 작동을 바꾸지 않고 디버깅만을 위해 삽입 / 수정한 output stream 용 코드들이 존재한다.

Cycle detection

- 모든 cycle detection 은 decode_insn() 내에서 진행했다.
- cycle detection 을 위해 필요한 코드는, disasm()에서 가져와 사용했다.

기본적인 구성은 다음과 같다.

- ecall 은 nop 처리 (1 instruction 어치를 맡되 ecall 로 인한 stall 이 발생 X)
- LOAD-USE 시 1 cycle stall
- LOAD – (instruction) – BRANCH 시 1 cycle stall
- LOAD – BRANCH 시 2 cycle stall
- BRANCH 가 Taken 될 시 1 cycle stall
- Jump, Jalr 가 발생될 시 1 cycle stall
- USE – BRANCH 시 1 cycle stall (use 는 EX 가 끝난 후 정보가 확정되는데, BRANCH 는 ID 단계에서 정보가 필요하니 1 cycle stall)

이를 계산하기 위해 필요한 정보는 다음과 같이 얻어냈다.

1. 현재 instruction 의 bit →

   ```
   uint64_t bits = insn.bits() & ((1ULL << (8 * insn_length(insn.bits()))) - 1);
   ```

   (코드 출처 : processor_t::disasm())

2. 이전, 이전이전 instruction 의 bit → insn_t prevbits, insn_t prevprevbits 를 정의 후,

   각 instruction 에 대한 계산이 끝날 때마다 갱신했다.

3. 현재 instruction 의 PC (main 인지 또는 branch 가 taken 되었는지 확인하기 위해 필요함.)

   ```
   if (zext(state.pc, max_xlen) == 0x10178)
   ```

   zext(state.pc, max_xlen)으로 현재 PC 를 계산했다. (코드 출처 : processor_t::disasm())

4. 이전, 이전이전 instruction 의 PC → state.pc()의 반환형이 reg_t 임에 착안,

   reg_t prev_pc, reg_t prevprev_pc 를 정의해 각 계산이 끝날 때 마다 갱신했다.

5. Stall 이 발생할지 말지를 확인하기 위해 레지스터 정보 확인

   decode.h()에 정의되어 있다. insn_t type 의 레지스터 정보는 ".rs1(), ".rd() 처럼 접근이 가능하다.

   reg1.rd() == reg2.rd() 처럼 비교도 가능하기에, 이걸 사용해줬다.

```
if (zext(state.pc, max_xlen) == 0x10178) // main started. from next instruction, every instructions are valid.
{
        validinstr = false;
        valid_from_next = true;
        inside_main = true;
}
if (zext(state.pc, max_xlen) == 0x1017C) // main finished. valid or not, every instructions shouldn't be counted.
{
        validinstr = false;
        inside_main = false;
        std::cout << std::dec << cnt+4 << std::endl;
        // cnt is the cnt of 'instructions', but I mixed it with 'cycles' in the counting process.
        // since one instruction has 5 counts, cnt + 4
}
```

현재 instruction 의 PC 가 0x10178 이라면 print 를 시작하게끔 flag 를 조정한다.

현재 instruction 의 PC 가 0x1017c 라면 print 를 멈추게끔 flag 를 조정한다.

```
if ((bits & 0x7F) == 0x73)
{
      if (bits != 0x00000073) validinstr = false; // if exceptions, don't print. but, print ecall.
      else
      {
              //std::cout << "ecall" << std::endl;
              validinstr = true;
      }
}
if (bits == 0x10200073) valid_from_next = true; // sret. After sret, all instructions are valid
```

현재 instruction 의 opcode (bits & 0x7F) 가 0x73 (ecall, ebreak, csrrw 등..) 인 경우,

instruction 이 ECALL 이라면 무시하지 않고 nop 처리 해야 하기에 냅두고, 아니라면 flag 를 false 처리.

instruction 이 sret 이라면 그 다음엔 다시 usercode 로 돌아오기에,

'이 다음부터 usercode 에요' 를 알려주는 flag 를 true 처리.

```
if ((prevprevbits & 0x7F) == 0x3) // prevprev instruction is load
{
        if ((bits&0x7F) == 0x63) // now inst is branch
        {
                if (prevprev.rd() == insn.rs1() || prevprev.rd() == insn.rs2())
                {
                        cnt++;
                }
        }
}
```

LOAD-(instruction)-BRANCH 의 경우, LOAD 의 rd()가 BRANCH 에서 사용된다면 1 cycle stall.

```
if ((prevbits & 0x7F) == 0x3) // prev instruction is load .. we're searching for load-use hazards
{
        if ((bits & 0x7F) == 0x63) // if now inst is branch
        {
                if (prev.rd() == insn.rs1() || prev.rd() == insn.rs2())
                {
                        cnt += 2;
                }
        }
```

LOAD-BRANCH 의 경우, LOAD 의 rd 가 BRANCH 에서 사용된다면 2 cycle stall.

```
else if ((bits & 0x7F) == 0x33 || (bits & 0x7F) == 0x3B) // now inst is R-Type (including RV-64I)
{
        if (prev.rd() == insn.rs1() || prev.rd() == insn.rs2())
        {
                cnt++;
        }
}
else if ((bits & 0x7F) == 0x13 || (bits & 0x7F) == 0x1B) // now inst is I-type (including RV-64I)
{
        if (prev.rd() == insn.rs1())
        {
                cnt++;
        }
}
else if ((bits & 0x7F) == 0x23) // now inst is S-Type
{
        if (prev.rd() == insn.rs1() || prev.rd() == insn.rs2())
        {
                cnt++;
        }
}
else if ((bits & 0x7F) == 0x37 || (bits & 0x7F) == 0x17) // now inst is AUIPC or LUI
{
        if (prev.rd() == insn.rd())
        {
        cnt++;
        }
}
else if ((bits & 0x7F) == 0x6F) // now inst is J-Type
{
        if (prev.rd() == insn.rd())
        {
                cnt++;
        }
}
```

그 외의 LOAD-USE 의 경우, LOAD 가 끝난 후 그 레지스터를 use 해줘야 하니 전부 1 cycle stall 을 해준다.

```
else if ((prevbits & 0x7F) == 0x63) // prev instruction is branch
{
         if (prev_pc + 4 != zext(state.pc, max_xlen)) // if branch is taken
         {
                 cnt++;
         }
}
else if ((prevbits & 0x7F) == 0x6F) // prev instruction is jal
{
         if (prev_pc + 4 != zext(state.pc, max_xlen))
         {
                 cnt++;
         }
}
else if ((prevbits & 0x7F) == 0x67 && ((prevbits>>11) & 0x7) == 0) // prev inst is jalr
{
         if (prev_pc + 4 != zext(state.pc, max_xlen))
         {
                 cnt++;
         }
}
```

BRANCH (taken)의 경우 1 개의 instruction 을 flush 해줘야 하니 1 cycle stall 이 생긴다.

jump, JALR 역시 같은 이유로 1 cycle stall 을 해준다.

```
else
{
        if ((bits & 0x7F) == 0x63) // now branch
        {
                if ((prevbits & 0x7F) == 0x33 || (prevbits & 0x7F) == 0x13
                                || (prevbits & 0x7F) == 0x1B || (prevbits & 0x7F) == 0x3B) // prev execute -> use
                {
                        if (prev.rd() == insn.rs1() || prev.rd() == insn.rs2())
                        {
                                cnt++;
                        }
                }
        }
}
```

USE-BRANCH 의 경우 Branch 는 ID 단계에서 필요로 하는 레지스터를 USE 가 EX 단계가 끝나서야 갱신해 줄 수 있으므로 1 cycle stall 이 필수적.

```
if (!prevprevcnt)
{
      prevprev = insn;
      prevprev_pc = zext(state.pc, max_xlen);
      prevprevcnt = true;
}
else if (prevprevcnt && !prevcnt)
{
        prev = insn;
        prev_pc = zext(state.pc, max_xlen);
        prevcnt = true;
}
else if (prevprevcnt && prevcnt)
{
        prevprev = prev;
        prevprev_pc = prev_pc;
        prev = insn;
        prev_pc = zext(state.pc, max_xlen);
}
```

현재의 instruction 에 대한 cycle 계산이 끝났을 때,

현재 instruction 에 대한 정보를 prev 와 prev_pc 에 담고,

이전 instruction 에 대한 정보를 prevprev 과 prevprev_pc 에 담는다.

- Details

```
if (prevbits == bits)
{
        //std::cout << "pagefault" << std::endl;
        cnt--; // since pagefault, prev instr == now instr, so don't count as new instruction
}
```

이전 instruction 이 현재의 instruction 과 완전히 같다면,

Pagefault 로 인해 (instr) -> pagefault -> (instr) 처럼 한 instruction 이 두번 실행된 것으로 간주,

추가했던 cnt 를 줄여서 instruction 취급하지 않는다.

```
if (zext(state.pc, max_xlen) == 0x1017C) // main finished. valid or not, every instructions shouldn't be counted.
{
        validinstr = false;
        inside_main = false;
        std::cout << std::dec << cnt+4 << std::endl;
        // cnt is the cnt of 'instructions', but I mixed it with 'cycles' in the counting process.
        // since one instruction has 5 counts, cnt + 4
}
```

main 이 끝났을 때 cnt (= cycle) 수를 print 하는데,

이 때 cnt 는 instruction 의 수와 단순히 동기화 되어있기 때문에,

하나의 instruction 만 존재한다면 cycle 수가 5 가 아닌 1 이 print 된다.

따라서 cnt 에 4 를 더해서 출력했다.

- 실행결과

```
2020311937@swji:/home/2020311937/Spike_Assignment$ ./build/spike ./pk/pk bench/test1
bbl loader
Hello, RISCV!
1350
2020311937@swji:/home/2020311937/Spike_Assignment$ ./build/spike ./pk/pk bench/test2
bbl loader
Bubble sort done
37025345
2020311937@swji:/home/2020311937/Spike_Assignment$ ./build/spike ./pk/pk bench/test3
bbl loader
Largest prime 999983
62536876
2020311937@swji:/home/2020311937/Spike_Assignment$ ./build/spike ./pk/pk bench/test4
bbl loader
fibonacci done
3563
2020311937@swji:/home/2020311937/Spike_Assignment$ ./build/spike ./pk/pk bench/test5
bbl loader
matmul done
99825368
```