Project 3: Simulating Pipelined Execution

Due 23:59, 21th May, 2024

TA: Seonmu Oh(smoh@dgist.ac.kr), Hyunkyun Shin(hkshin@dgist.ac.kr)

Sungju Kim(sungju_kim@dgist.ac.kr), Kyeonghyeon Ryu(khryu@dgist.ac.kr)

Introduction

세 번째 과제는 두 번째 과제에서 구현하였던 에뮬레이터의 확장으로, MIPS ISA를 대상으로 다섯 단계의 파이프라인을 구현하는 것이다. 파이프라인은 인스트럭션을 처리하는 과정을 여러 단계로 나누어, 각 단계가 병렬적으로 동작할 수 있게 하는 기법이다. 에뮬레이터가 지원해야 하는 인스트럭션은 첫 번째 과제 및 두 번째 과제와 동일하며, 본 과제 설명에 나와있지 않은 내용은 수업 자료를 따른다.

1. Pipeline Implementation

A. Pipeline Stages

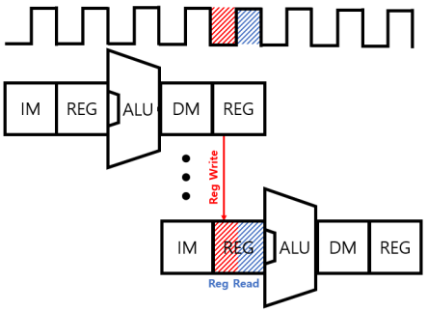
구현할 파이프라인 단계는 이하의 5단계로 구성된다.

- IF (Instruction Fetch) : 인스트럭션 하나를 메모리에서 읽어 온다.
- ID (Instruction Decode) : 읽어 온 인스트럭션을 해석(decode)하고 레지스터에서 필요한 값을 읽는다.
- EX (Execution) : ALU 연산을 실행한다.
  - 산술 논리 연산(arithmetic and logical operation)및 'load'와 'store' 인스트럭션에 대한 주소 계산을 수행한다.
- MEM (Memory) : 'load', 'store' 인스트럭션에 대한 메모리 접근(access)을 한다.
- WB (Write Back) : 연산 결과를 레지스터에 쓴다.

파이프라인 과정에서 발생하는 해저드(hazard)를 막고 전방전달(forwarding)을 구현하기 위해 각 파이프라인 단계 사이에 파이프라인 상태 레지스터(pipelining state register)와 같은 내용을 구현해야 한다.

B. Simultaneous Register Access

MIPS에서는 한 사이클의 첫 절반 단계에서는 WB, 나머지 절반 단계에서는 ID가 레지스터에 접근한다. 서로 다른 타이밍에 레지스터를 접근하는 것으로 파이프라인 과정에서 발생하는 구조적 해저드(structural hazard)를 방지할 수 있어야 한다.



C. Memory

이번 과제에서는 IF 단계에서 메모리를 읽는 동작과 MEM 단계에서의 메모리 읽기/쓰기 동작이 한 사이클에 동시에 이루어지는 이상적인 메모리를 사용한다고 가정한다. 다시 말해, IF와 MEM 단계에서 동시에 메모리 접근을 시도하는 것으로 인해 발생하는 구조적 해저드는 없는 것으로 가정한다. 또한, 인스트럭션들이 저장된 메모리 영역이 한 사이클 도중에 바뀌는 경우는 없다고 가정한다. 따라서 같은 사이클에, MEM 단계에 있는 store 인스트럭션이 IF 단계에서 불러온 인스트럭션을 변경하는 경우는 발생하지 않아야 한다.

D. Data Hazard (Forwading)

데이터 해저드를 막기 위해서는 지연(stall)을 넣거나 파이프라인 상태 레지스터를 이용하여 데이터 전방전달(forwarding)을 수행해야 한다. 이번 과제에서 사용할 파이프라인에서는 'MEM/WB to EX', 'EX/MEM to EX', 'MEM/WB to MEM'의 세 종류의 데이터 포워딩을 지원해야한다. 또한, load 등의 인스트럭션에 뒤따라오는 인스트럭션 중, EX 단계에서 해당 레지스터 값을 사용해야 하는 인스트럭션에 의해 발생하는 데이터 의존성(data dependency)는 한 사이클의 지연이 필요한 것을 고려하여 구현해야 한다.

E. Control Hazard

점프 인스트럭션(J, JAL, JR)이 fetch 될 때에 발생하는 제어 해저드(control hazard)를 막기 위해, 점프 인스트럭션 이후에 한 사이클 동안 지연을 발생시켜 해당 문제를 해결해야 한다.

조건 분기 인스트럭션(BEQ, BNE)에 의해 발생하는 제어 해저드를 막기 위해서 분기 예측기(branch predictor)를 사용한다. 이번 과제에서는 정적 분기 예측기(static branch predictor)인 Always Taken, Always Not Taken을 구현해야 한다. 이번 과제에서는 ID 단계 분기 전방전달(ID stage branch forwarding)은 구현하지 않고, MEM 단계에서 분기가 결정되는 것으로 생각한다.

- Always Taken : 항상 모든 분기가 일어날 것(taken)이라고 예측한다. 이를 위해, ID 단계에서 분기될 주소(target address)를 계산한다. 분기 예측이 성공하면 1 사이클의 지연이 발생한다. 분기 예측이 실패하면 3 사이클 동안 쓸어내기(flushing)로 인해 지연이 일어나도록 구현한다.

- Always Not Taken : 항상 모든 분기가 일어나지 않을 것(not taken)이라고 예측하고, 지연 없이 바로 다음 instruction을 가져온다. 따라서, 분기 예측이 성공하면 지연이 발생하지 않는다. 실패할 경우, 쓸어내기로 인해 3 사이클 동안 지연이 일어나도록 구현한다.

F. The end of the pipeline

우리가 작성할 프로그램(시뮬레이터)은 모든 인스트럭션이 WB 단계를 끝낸 이후 종료되어야 한다. 첫 번째 사이클(cycle 1)에서, 첫 번째 인스트럭션을 메모리에서 불러오고(첫 번째 인스트럭션의 IF 단계), 마지막 인스트럭션이 N 번째 사이클에서 WB 단계에 있다면, 최종 사이클 카운트는 cycle N이다.

G. Other Conditions

두 번째 과제와 같이 이번 과제에서 만들어야 하는 프로그램은 MIPS 인스트럭션을 실행시키며, 시스템(레지스터 및 메모리)의 상태를 올바르게 유지하여야 한다. 프로그램은 바이너리 파일을 읽어 text 영역 및 data 영역의 크기를 읽고 메모리를 생성한다. 이전 과제와 같이 text 영역은 0x400000, data 영역은 0x10000000의 주소부터 시작한다. 이번 과제에서도 역시 stack 영역을 생성하지 않는다.

초기 상태(initial states):

- PC : PC의 초기값은 text 영역의 시작 주소인 0x400000이다.
- Registers : 모든 레지스터(R0-R31)는 초기값으로 0을 갖는다. jal 등의 인스트럭션에서 사용하는 $ra 레지스터는 R31에 해당한다.
- Memory : 불러온 data 영역과 text 영역을 제외하고 모든 메모리는 초기값으로 0x0을 갖는다.

2. Pipeline Register States

각 단계 사이에 파이프라인 상태 레지스터(pipeline state register)를 추가해야 한다. 다음은 파이프라인 상태 레지스터의 예시이다.

A. IF_ID.Instr : 32비트 인스트럭션

B. IF_ID.NPC : 다음 32비트 PC 값(PC+4)

C. ID_EX.NPC : 다음 32비트 PC 값

D. ID_EX.rs : rs에 해당하는 레지스터의 번호

E. ID_EX.rt : rt에 해당하는 레지스터의 번호

F. ID_EX.IMM : 상수 값(Immediate value)

G. ID_EX.rd : 목적지(destination)에 해당하는 레지스터의 번호

H. EX_MEM.ALU_OUT : ALU에서 출력(output)된 값.

I. EX_MEM.BR_TARGET : 분기 주소(branch target address)

J. MEM_WB.ALU_OUT : ALU 출력

K. MEM_WB.MEM_OUT : 메모리 출력(memory output)

위에 예시 이외에도 구현 과정에서 추가로 파이프라인 상태 레지스터가 필요할 수 있다. 과제를 진행하는 학생들은 인스트럭션을 수행하는 과정에서 어떤 내용이 파이프라인 상태 레지스터에 필요한지, 어떤 내용을 전방전달 할 수 있도록 해야 하는지에 대해서 정확한 설계를 해야 한다. 학생들은 과제를 진행하는 과정에서 **파이프라인 상태 레지스터의 설계를 어떻게 하였는지, 그리고 왜 그렇게 설계하였는지에 대해 보고서에 설명을 적어 같이 제출하도록 한다.**

3. Coding and Execution

A. Environment

제출된 코드는 Ubuntu 20.04, Windows Subsystem for Linux(Ubuntu 20.04) 환경에서 테스트될 것이다. 위 환경 중 하나에서 코드가 실행 가능하면 채점 가능하다. 이외의 OS 에서 코딩을 진행하는 경우 OS API 등의 코드로 인해 코드 컴파일이 불가능한 경우가 있을 수 있으니, 위 환경 중 최소 하나 이상에서 테스트한 후 제출하여야 한다.

B. Program Language

프로그래밍 언어는 C, C++만이 허용된다. 이 언어들 중에 어떤 것을 사용하여도 좋다.

- C 혹은 C++의 경우, gcc 9.4.0, g++ 9.4.0 버전의 컴파일러를 이용하여 채점을 진행하게 된다.
- C 혹은 C++의 경우 Microsoft Visual Studio를 사용해야만 컴파일이 되는 경우, **컴파일이 불가능한 것으로 간주**한다. 또한, Microsoft Visual Studio 프로젝트 파일을 제출하는 것은 인정되지 않는다.

B. Execution command

옵션이 없는 기본적인 실행 형태는 아래와 같다.

```
$ ./runfile <-atp 또는 -antp> [-m addr1:addr2] [-d] [-p] [-n num_instr] <binary file>
```

- `-atp`: Always Taken 분기 예측기를 사용한다.
- `-antp`: Always Not Taken 분기 예측기를 사용한다.
- `-m`: 프로그램이 종료될 때 메모리 주소 범위(addr1 ~ addr2)에 있는 내용들을 출력한다. 해당 주소가 data section의 주소(0x10000000에서 시작) 혹은 text section의 주소(0x400000에서 시작)를 벗어난 범위를 가리키고 있을 경우, 해당 주소에는 값이 할당되지 않았으므로 0x0을 출력한다. 디폴트 값은 없기 때문에 본 플래그가 있을 때는 항상 주소 값을 입력해주어야 한다.
- `-d`: 매 사이클 마다 모든 레지스터의 내용을 출력한다. –m 옵션으로 출력할 메모리 주소 범위가 지정되어 있다면 지정된 메모리 범위의 내용도 출력한다.
- `-p`: 매 사이클 마다 각 파이프라인 단계에서 실행되고 있는 인스트럭션의 PC를 아래와 같은 형태로 출력한다.

  e.g.) `{0x400010|0x40000c|0x400008|0x400004|0x400000}`
- `-n`: 수행될 명령어의 개수를 지정한다. 디폴트 값은 없기 때문에 본 플래그가 있을 때는 항상 개수를 지정해주어야 한다.
- 입력 값 중 대괄호('[]')로 나타낸 부분은 옵션을 포함하지 않을 경우 기본 값으로 실행되는 옵션들이다. '<>'로 나타낸 옵션은 프로그램을 실행할 때 반드시 필요한 옵션이므로, 옵션이 들어가 있지 않을 경우 실행 오류로 처리한다.

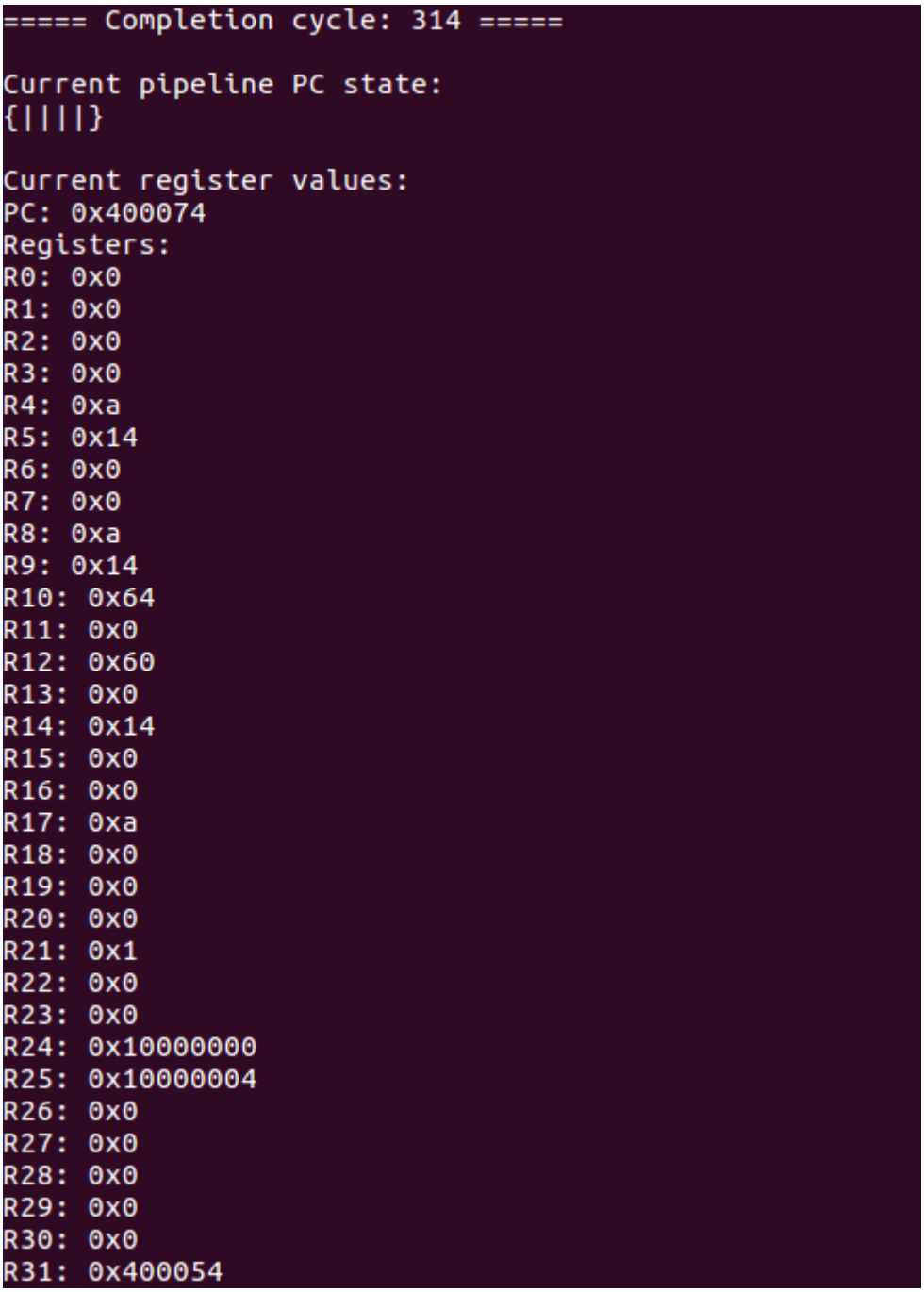
C. Input format

입력 파일은 두 번째 과제에서 사용한 object 파일(*.o)과 같은 파일이다.

D. Output format

별도의 옵션이 지정되지 않았을 경우, 마지막 인스트럭션이 WB 단계를 끝낸 시점에서의 결과를 아래와 같이 출력한다. 출력해야 하는 정보들은 다음과 같다.

- 프로그램이 끝날 때까지 필요한 사이클의 수
- 출력하는 시점에서의 파이프라인 상태
- 프로그램이 종료된 시점의 레지스터의 내용



[Figure 1. Output format at the end of the program]

-p 옵션이 사용되었을 경우, 매 사이클마다 파이프라인의 상태를 출력한다.

-d 옵션이 사용되었을 경우, 매 사이클마다 레지스터의 상태를 출력한다. -m 옵션이 같이 사용되었을 경우에는 메모리의 상태도 출력한다.

옵션이 사용되었을 때의 출력 예시는 다음과 같다.

```
===== Cycle 3 =====
Current pipeline PC state:
{0x400008|0x400004|0x400000||}

Current register values:
PC: 0x40000c
Registers:
R0: 0x0
R1: 0x0
R2: 0x0
R3: 0x0
R4: 0x0
R5: 0x0
R6: 0x0
R7: 0x0
R8: 0x0
R9: 0x0
R10: 0x0
R11: 0x0
R12: 0x0
R13: 0x0
R14: 0x0
R15: 0x0
R16: 0x0
R17: 0x0
R18: 0x0
R19: 0x0
R20: 0x0
R21: 0x0
R22: 0x0
R23: 0x0
R24: 0x0
R25: 0x0
R26: 0x0
R27: 0x0
R28: 0x0
R29: 0x0
R30: 0x0
R31: 0x0
```

[Figure 2. Output format at cycle 3 with -p -d]

```
===== Cycle 7 =====
Current pipeline PC state:
{0x400018|0x400014|0x400010|0x40000c|0x400008}

===== Cycle 8 =====
Current pipeline PC state:
{0x40001c|0x400018|0x400014|0x400010|0x40000c}

===== Cycle 9 =====
Current pipeline PC state:
{0x400020|0x40001c|0x400018|0x400014|0x400010}
```

[Figure 3. Output format at cycle 7, 8, 9 with -p]

4. Report

과제를 수행한 학생들은 다음 내용이 들어있는 보고서를 작성하여 소스 파일과 함께 제출하여야 한다.

- 자신이 작성한 과제에 대한 간략한 설명
- 과제의 컴파일 방법 및 컴파일 환경(사용한 OS, 컴파일러 버전)
- 과제의 실행 방법 및 실행 환경

컴파일 방법 및 컴파일 환경에 대한 보고서 작성 예시는 다음과 같다.

- ✓ WSL Ubuntu 20.04 환경에서 gcc 9.4.0 버전을 이용하여 다음 명령어로 컴파일 하였다.
  `gcc -o main main.c -std=c++14`
- ✓ Ubuntu 20.04 환경에서 g++ 9.4.0 버전을 이용하여 컴파일 하였다. 첨부한 makefile 을 소스 코드와 같은 폴더에 넣고, 리눅스 커맨드라인에서 `make` 명령어로 컴파일 할 수 있다.

실행 방법 및 실행 환경도 위와 같은 형식으로 작성한다. 단, 4-C Execution Command 에 지정한 명령어와 다른 명령어로 프로그램이 실행되는 경우, 감점의 요인이 될 수 있다. **4-B Programming Language에 명시 해놓은 gcc, g++ 이외의 컴파일러를 사용한 컴파일 방법을 보고서에 컴파일 방법으로 기재하였거나, 보고서에 컴파일 방법을 명시 해놓지 않은 경우 소스 코드를 컴파일 할 수 없는 것으로 간주한다.**

보고서는 ".pdf" 형식으로 변환하여 제출한다. 이를 지키지 않는 경우 감점된다. 더불어 보고서 내에 소스 코드를 기재할 필요는 없다.

5. Submission

A. **프로젝트 결과물(소스 파일) 및 보고서를 zip 형식으로 압축**하여 LMS에 해당 과제 게시글로 제출하도록 한다. **압축 파일의 이름은, 학번_이름.zip** 으로 **꼭 양식을 지키도록** 한다. 양식에 맞지 않을 경우, 감점의 요인이 된다.

B. e.g. 202411999 김철수 학생의 경우, **202411999_김철수.zip** 형식으로 제출.

6. Notice

A. Due date: **23:59, 21th May, 2024**

B. Penalty

- Late due

Case	Penalty
1일 지연	점수 10% 감점
2일 지연	점수 30% 감점
3일 지연	점수 50% 감점
4일 이상 지연	0점 처리

- Cheating 적발 시 D+ 이하의 학점 부여
- Compile 혹은 실행이 마지막 Late due까지 안 될 경우 0점 처리.
- 파일 이름, 실행 방법 등이 규정한 양식과 다를 경우 감점 처리.
- 보고서에 컴파일 방법을 작성하지 않거나, 과제에서 규정한 컴파일러를 사용한 컴파일 방법이 적혀 있지 않을 경우, 컴파일이 불가능한 것으로 처리.

C. TA Contact

TA에게 질문 사항 또는 기타 용건이 있다면, LMS Q&A 게시판을 통해 문의하거나, 메일로 문의할 것(e.g. 수신 1명, 수신자 제외 참조 1명). 또한 TA를 직접 만나서 이야기하고 싶다면, 메일로 미리 약속을 잡을 것.

- **Seonmu Oh(smoh@dgist.ac.kr)**
- **Hyunkyun Shin(hkshin@dgist.ac.kr)**

7. FAQ

Q. jal이나 jr을 구현할 때 레지스터의 값을 읽거나 쓰는 경우 어떤 단계에서 해당 작업을 수행해야 하나요?

A. 레지스터의 읽기는 ID 단계에서, 쓰기는 WB 단계에서 수행됩니다.

Q. jal 명령어 이후 jr $ra 명령어가 실행될 경우 1 사이클 stall을 하더라도 hazard가 발생합니다. 어떻게 해야 하나요?

A. jal 명령어로 점프한 곳에 jr $ra 명령어가 있을 경우, 1 사이클 stall을 하더라도 jal 명령어가 WB 단계에서 $ra 레지스터를 업데이트 하기 전에 jr 명령어가 ID 단계에서 $ra 레지스터를 읽으므로 잘못된 PC 값을 fetch할 수 있습니다. 그러나 이번 샘플에서 jal 명령어 다음 jr 명령어가 실행되는 경우는 없으므로 따로 hazard 처리를 하지 않아도 됩니다.

Q. 분기 예측에서 flush와 stall이 정확히 어떻게 발생하는 지 이해되지 않습니다.

A. always taken의 예측기의 경우 항상 모든 분기가 일어나는 것으로 예측하여 ID 단계에

서 분기될 주소를 계산하고 분기하되, j, jal, jr 명령어와 마찬가지로 컨트롤 해저드를 막기 위해 1 사이클의 stall을 발생시켜야 합니다. 만약 분기 예측이 실패하면 taken 후 그 동안 fetch한 명령어 2개를 flush해야 하므로 총 3 사이클의 지연이 발생합니다. always not taken 예측기의 경우 항상 모든 분기가 일어나지 않는다고 예측합니다. 따라서 예측에 성공한 경우 별도의 stall이 발생하지 않습니다. 그러나 예측이 실패했을 경우 (즉 분기가 일어났을 경우), 분기를 해야하므로 fetch한 명령어의 flush로 인해 총 3 사이클의 지연이 발생하게 됩니다. 분기 예측기의 경우 강의 자료와 교과서에 더 자세한 설명이 나와있으니 참고 바랍니다.

Q. flush나 stall로 인해 noop이 파이프라인에 있는 경우 이를 어떻게 처리해야 하나요?

A. noop은 no operation의 줄임말로, 아무런 인스트럭션이 없는 것과 동일합니다. 더 이상 읽어올 instruction이 없어 파이프라인이 비워지는 것은 noop으로 파이프라인이 차는 것과 같다고 생각하면 됩니다. 동일하게 flush나 stall로 인해 파이프라인에 noop을 추가했을 때 해당 단계에 명령어가 없는 것으로 처리하고, 출력의 경우 {0x400010||0x400008|0x400004|0x400000}과 같이 해당 단계를 비우고 출력하면 됩니다.

Q. 파이프라인을 구현할 때 과제 설명에 언급된 것 이외의 상태 레지스터가 필요 한 경우 임의로 구현해도 상관없나요?

A. 필요에 따라 원하는 방법으로 파이프라인 상태 레지스터를 구현한 후 보고서에 해당 설명을 적으시면 됩니다.

Q. instruction이 모두 fetch된 후 파이프라인이 비워지는 중에도 PC가 증가해야 하나요?

A. 그렇지 않습니다. 마지막 instruction을 fetch하고 PC를 증가시킨 다음에는 더 이상 읽어올 명령어가 없기 때문에 PC를 증가시키지 않아야 합니다.

Q. 별도의 옵션이 없을 경우 마지막 사이클에 지정된 내용을 출력하라고 되어있는데 옵션을 준 경우에는 어떻게 해야 하나요?

A. 옵션이 있을 경우 마지막 사이클이 끝난 후 해당 내용을 출력하셔도 되고 안하셔도 됩니다.

Q. -n 옵션의 경우 fetch할 인스트럭션의 수를 지정하는 것인지 완료한 인스트럭션의 수를 지정하는 것인지 궁금합니다.

A. -n에서 입력한 숫자만큼의 인스트럭션이 모두 완료될 때까지 수행하고 결과를 출력하시기 바랍니다.