20171969

박건호

과제명 : 시스템 프로그래밍 Project #2 SIC/XE 시뮬레이터

출석번호 : 110

수업 구분 : 가

1. 동기 / 목적

이번 프로젝트의 주요 목적은 SIC/XE 시뮬레이터를 개발하는 것이다.

구현한 SIC/XE 시뮬레이터는 SIC/XE 컴퓨터 시스템의 기초적인 명령어 집합이 어떻게 실행되는지를 시각적으로 보여주는 도구이다.

텍스트 기반 시뮬레이터는 사용자가 명령어 실행 과정을 직관적으로 이해하기 어려울 수 있다. 따라서 그래픽 사용자 인터페이스(GUI)를 통한 시뮬레이터를 개발하면, 사용자가 각 명령어의 실행 과정을 보다 시각적으로 명확하게 이해할 수 있을 것이다. 이를 통해 저 수준에서 명령어가 어떻게 실행되는지 이해할 수 있다.

2. 설계 / 구현 아이디어

A 방법으로 구현하였다. - 가상으로 설정한 메모리를 직접 보여주고, 현재 수행되고 있는 명령어의 주소를 표시

설계

├─── Inst : 명령어 관리 위한 Pakage

│ ├─── Instruction

│ └─── InstructionSet

├─── Main : VisualSimulator 호출

├─── Device : Device 를 정의한 클래스

├─── Symbol : Symbol 을 정의한 클래스

├─── InstructionExecutor : 명령어 실행기

├─── ObjectCodeLoader : 로더

├─── ResourceManager : 실행 프로그램의 메모리, 레지스터 등을 저장하고 있는 리소스 매니저

├─── SICXESimulator : VisualSimulator 가 호출 / 명령어 실행기와 로더의 동작을 호출 관리

└─── VisualSimulator : GUI 창을 만들고 SICXESimulator에게 명령 호출

- GUI 구성

메모리 뷰어: 메모리 상태를 시각적으로 보여주는 패널.

레지스터 뷰어: 현재 각 레지스터의 값을 표시하는 패널.

명령어 로그: 실행된 명령어와 그 결과를 보여주는 로그 창.

디바이스 상태: 현재 사용 중인 입출력 디바이스의 상태를 표시하는 창.

- 명령어 실행 로직

각 명령어는 별도의 메서드로 구현하여 명령어의 기능을 시뮬레이션 한다.

명령어 실행 후, 리소스 매니저에서 관리되는 각 레지스터와 메모리의 상태를 업데이트하고 이를 GUI에 반영한다.

- 디바이스 관리

입출력 디바이스를 관리하는 별도의 클래스를 도입하여, 각 디바이스의 상태와 데이터를 관리하였다.

디바이스와의 데이터 송수신을 시뮬레이션 하여, 명령어 실행 시 디바이스에 행해지는 읽기 쓰기 동작을 확인 가능하다.

구현

- Java Swing을 이용한 GUI 구현

VisualSimulator 클래스에서 GUI 창을 구성하고, 실행할 명령어를 노란색으로 하이라이트 한다.

버튼에 따라 SICXESimulator의 동작을 호출한다.

JFrame과 JPanel을 이용하여 전체 창을 구성.

JTextField와 JTextArea를 이용하여 레지스터와 메모리의 상태를 표시.

JButton을 통해 명령어 실행 제어.

- 로더 실행 매커니즘

ObjecCodeLoader

GUI에서 파일 오픈 버튼이 눌리면 - SICXESimulator의 loadObjectCode를 호출

해당 오브젝트코드 파일을 2번의 PASS로 가상의 메모리 공간에 로드한다.

1PASS : 심볼테이블에 심볼을 등록 (ESTAB을 만드는 과정) , H와 E 레코드를 참고하여 정보를 저장, 프로그램 길이는 모든 심볼의 저장이 끝난 후 마지막 Section의 시작주소와 크기를 더하고 프로그램의 시작주소를 빼서 계산

2PASS : 만들어진 심볼테이블을 활용하여 실제로 가상의 메모리공간에 로드 - M 레코드로 인한 수정도 심볼테이블을 참고하여 수행 가능 - T 레코드의 내용을 로드, M 레코드에 따라 수정 - Current Section의 시작주소를 참고하여 수행한다.

로드 후에 GUI를 업데이트

- 명령어 실행 메커니즘

InstructionExecutor

GUI에서 명령어 실행 버튼이 눌리면 SICXESimulator의 executeNextInstruction 를 호출

명령어의 형식과 오퍼랜드를 해석하여 해당하는 명령어의 메서드를 호출.

각 명령어 메서드에서 레지스터와 메모리의 상태를 변경하고, 변경된 내용을 리소스매니저에 반영한다.

실행 후에 GUI를 업데이트

Cf) 모든 명령어 실행

종료조건을 while 문으로 검사하여 종료가 아니라면 명령어 실행을 계속 수행한다.

종료조건은 임의로 설정하였다.

본 프로젝트에서 실행하는 COPY프로그램이 일종의 서브루틴이고, 어디선가 호출을 하였을 것이다. 모든 작업이 끝나고 돌아가야 하는 곳은 원래 L레지스터에 저장되어 있었을 것이고, 그 주소를 0x3000 이라고 가정하여 코딩하였다.

- GUIupdate

리소스매니저로부터 필요한 값들을 받아와서 VisualSimulator 의 GUI 창에 담긴 필드들을 수정한다. 그리고 여기서 PC 값에 해당하는 메모리 영역을 노란색 하이라이트 하였다.

현재 프로그램의 큰 틀을 정리하면 다음과 같다.

Main : VisualSimulator 생성

VisualSimulator : GUI 생성, SICXESimulator 생성

버튼 입력에 따라 SICXESimulator 에게 동작을 요청

SICXESimulator : ResourceManager, ObjectCodeLoader, InstructionExecutor 생성, ResourceManager 인스턴스 공유를 통해 로드와 명령어 실행으로 리소스의 변경을 가능하게 하였음

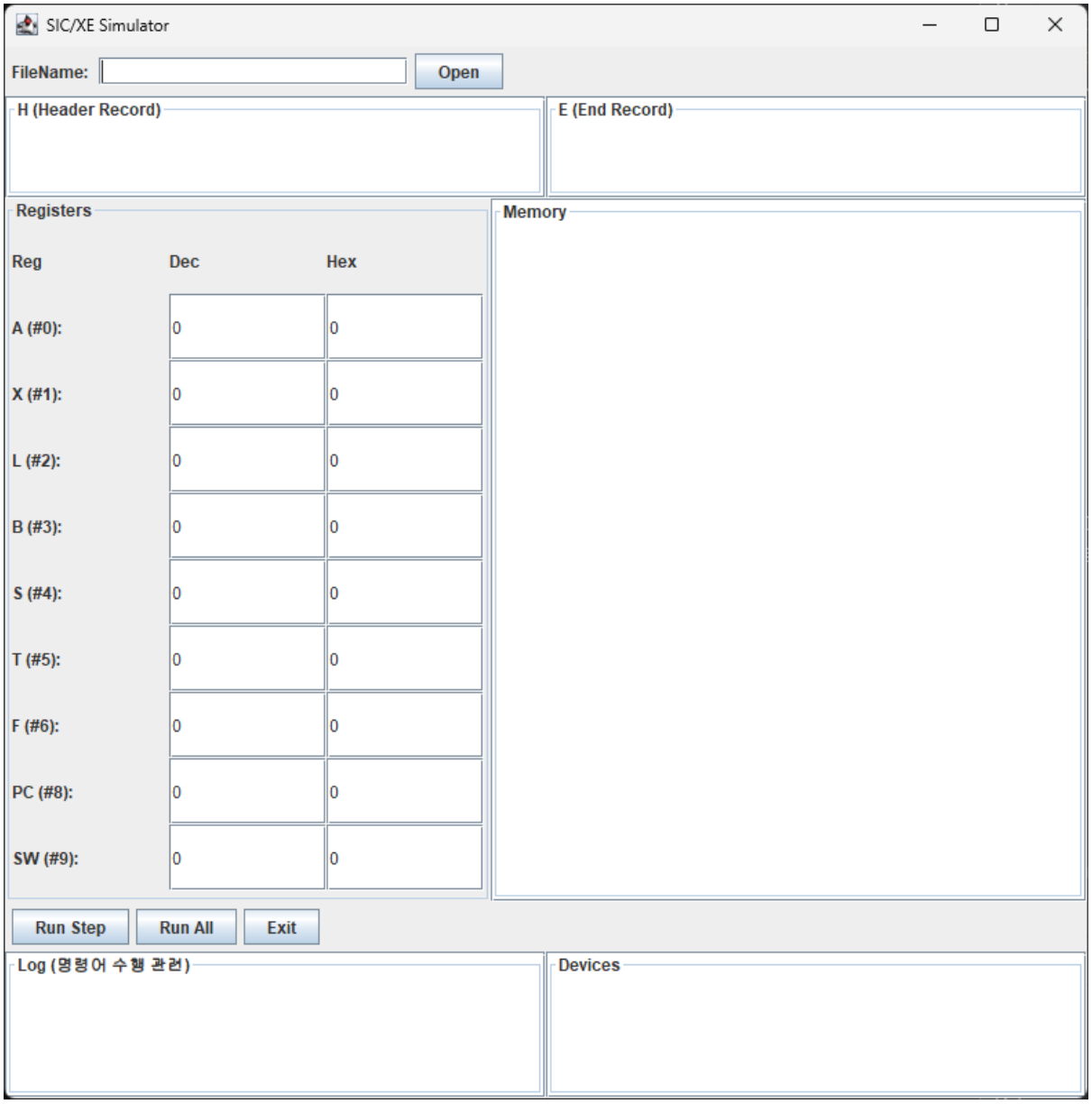
VisualSimulator 로부터 호출되어 ObjectCodeLoader, InstructionExecutor 에게 동작을 요청

ObjectCodeLoader : SICXESimulator 로부터 호출되어 로드 수행

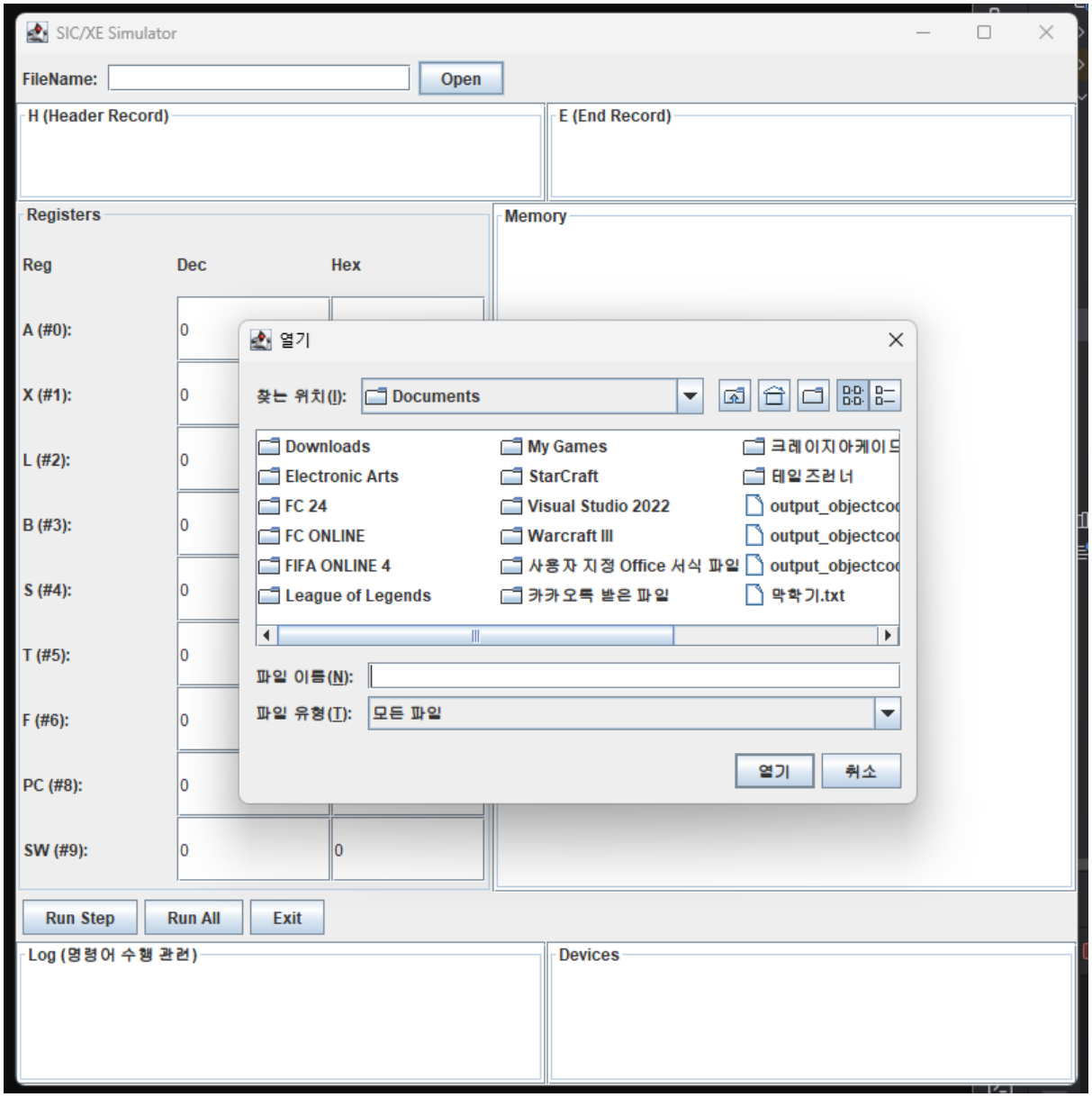
InstructionExecutor : SICXESimulator 로부터 호출되어 명령어 실행 수행

상세한 구현 로직들은 소스코드의 주석으로 달아놓았다.

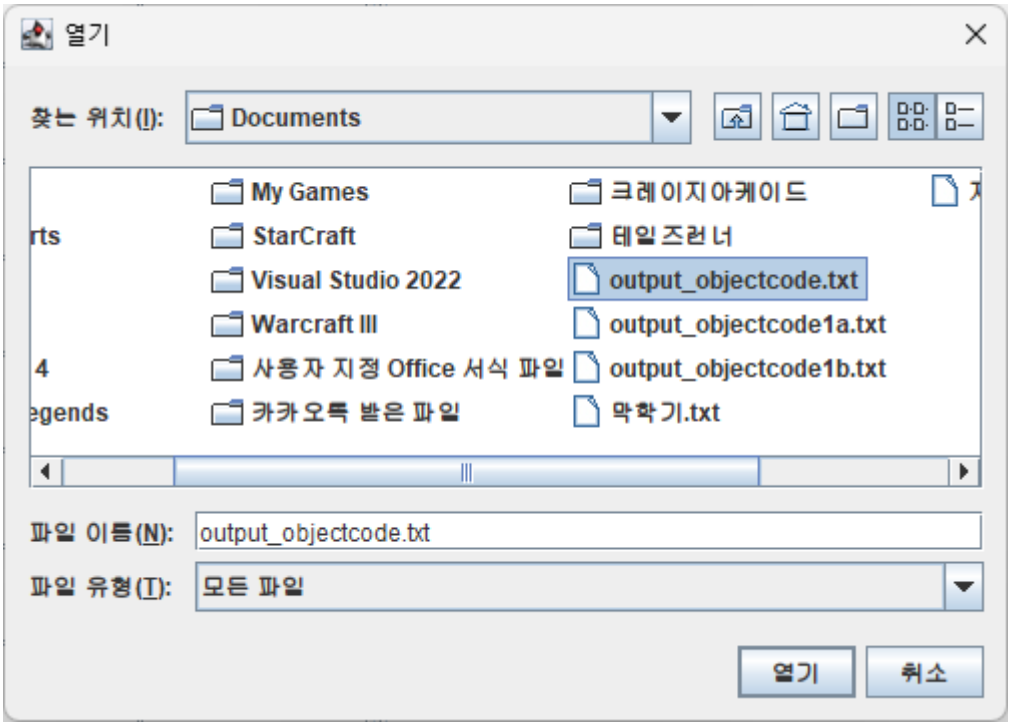
3. 수행결과



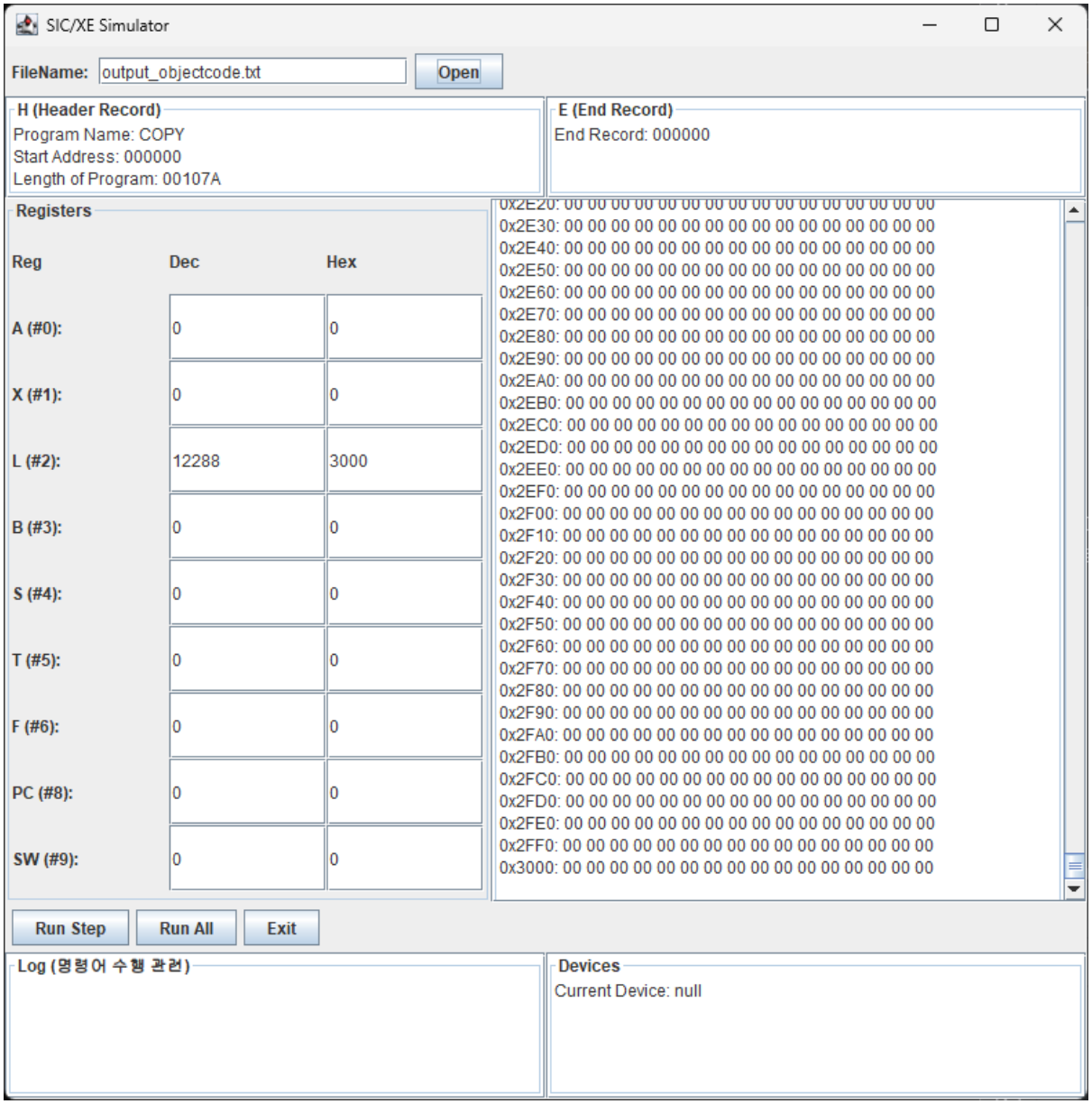
메인함수 실행 후 확인 가능한 GUI 창이다.



open 버튼을 클릭하면 파일을 선택하여 열기가 가능하다.



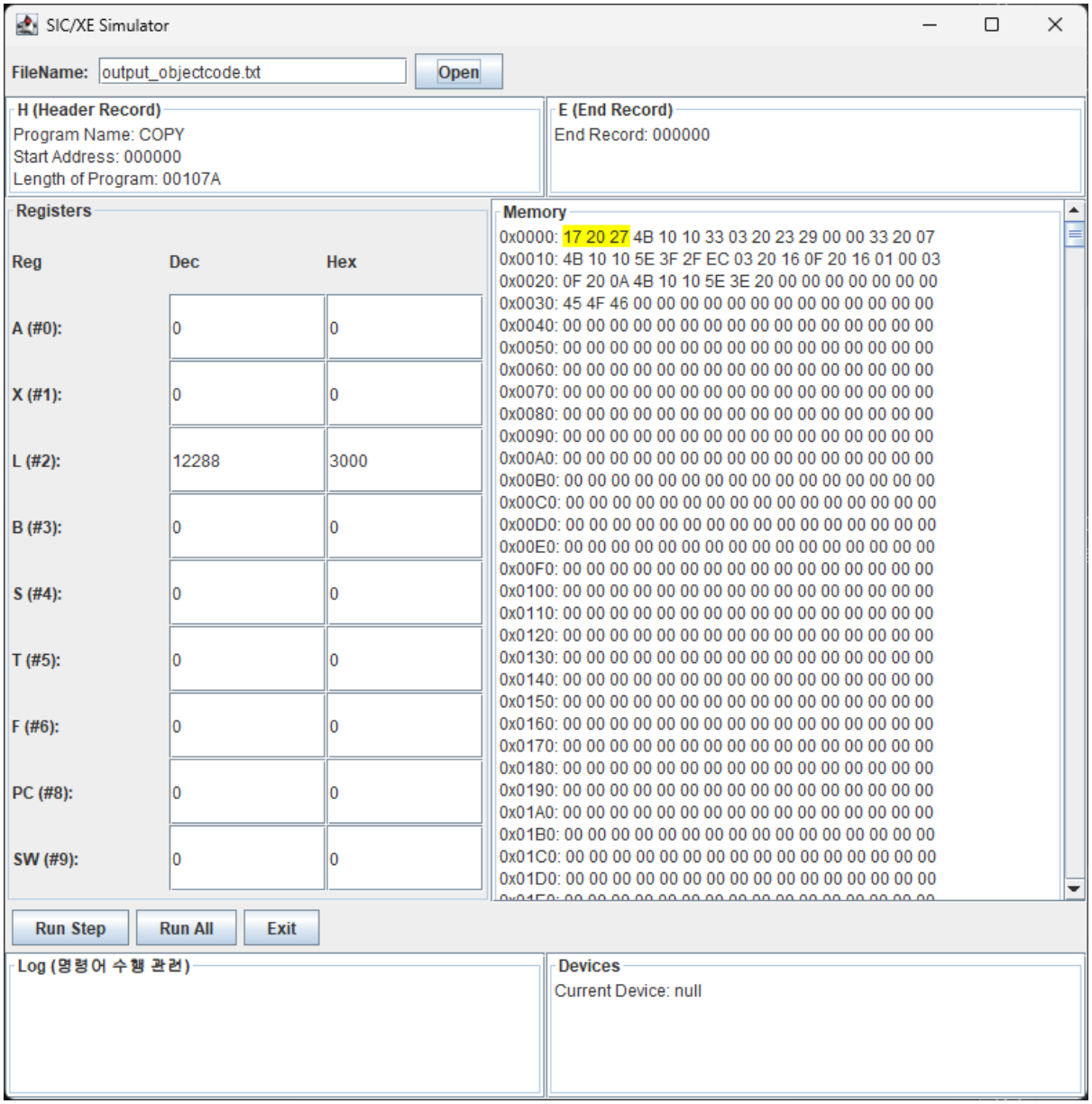
선택한 파일은 이전 프로젝트들의 결과물인 COPY 프로그램의 오브젝트 코드이다.



파일을 선택하여 열기를 수행하면 우선 로더가 실행되어 가상의 메모리 공간에 T Record의 내용이 올라간다.

그리고 H 와 E 레코드들을 확인하여 현재 프로그램의 이름, 시작주소, 길이, End 후 이동할 시작주소의 정보를 보여준다.

로드가 끝난 후에는 UpdateGUI 함수를 통해 로드 후 리소스 매니저로부터 정보를 받아온다.



로드 과정에서, 모든 로드가 끝나면 PC를 오브젝트코드에서 주어진 시작주소로 세팅한다.

Ex) H COPY 000000 001033 : 시작 주소인 000000 으로 PC 를 설정

그리고, 현재 오픈한 COPY 프로그램의 시작인 0x000000으로 PC가 세팅되었다는 것은, 어딘가에서 이 COPY 프로그램을 서브루틴으로서 Call 했다는 것이다.

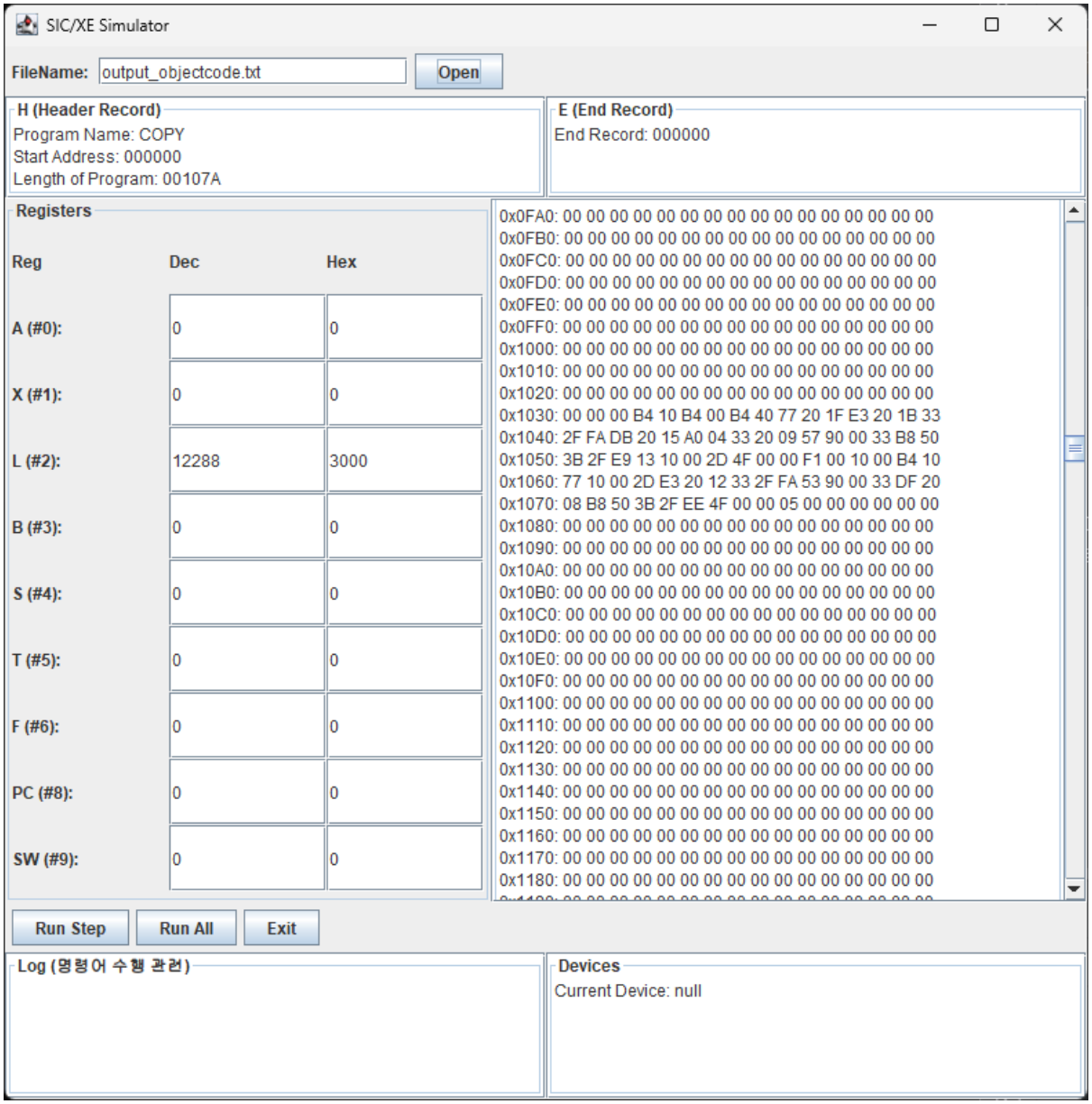
만약 COPY 서브루틴이 끝나고 RSUB한다면, COPY를 호출한 곳의 다음 명령어로 돌아와야 하는데, 그 주소를 임의로 설정하고 돌아왔는지 아닌지 판단하면 호출했던 COPY 프로그램이 끝났는지 안 끝났는지 판단할 수 있다. -> Run All 을 실행해서 COPY의 명령어가 모두 실행되었는지 판단하는 데 쓰일 것이다.

현재 프로젝트 구현에서는 간단하게 돌아올 곳을 0x3000으로 설정하기 위해 L 레지스터의 값을 0x3000으로 설정하였다.

그리고, 레지스터의 상태를 확인하면 L (Linkage) 레지스터에 0x3000이 담겨있는 것을 알 수 있다.

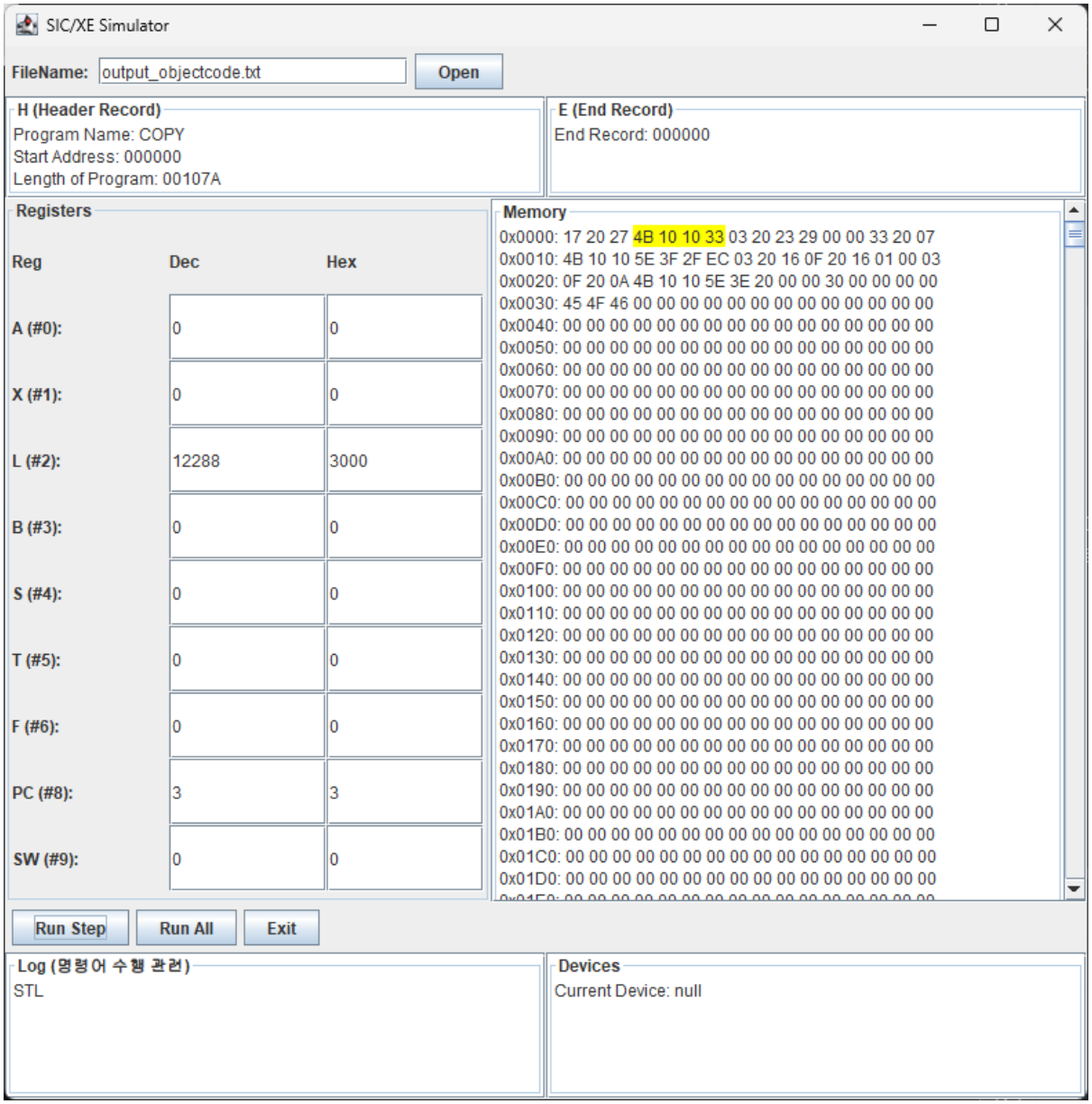
오른쪽에 위치한 가상의 메모리공간을 보면 노란색으로 표시된 명령어를 확인할 수 있는데, 저것은 PC : 실행될 명령어 위치 를 기준으로 표시되어 있는 것이다.

비주얼 시뮬레이터는 updateGUI 를 할 때, PC를 기준으로 실행될 명령어의 크기를 계산하고, 그 크기에 따라 실행될 명령어 부분을 노란색으로 강조한다.



0x1033부터 RDREC의 요소가 로드되어있음을 확인할 수 있다.

0x1032까지는 COPY 섹션의 BUFFER이다.



Run Step을 클릭하여 노란색으로 표시되어있었던 172027 명령어가 실행되고 나서의 모습이다.

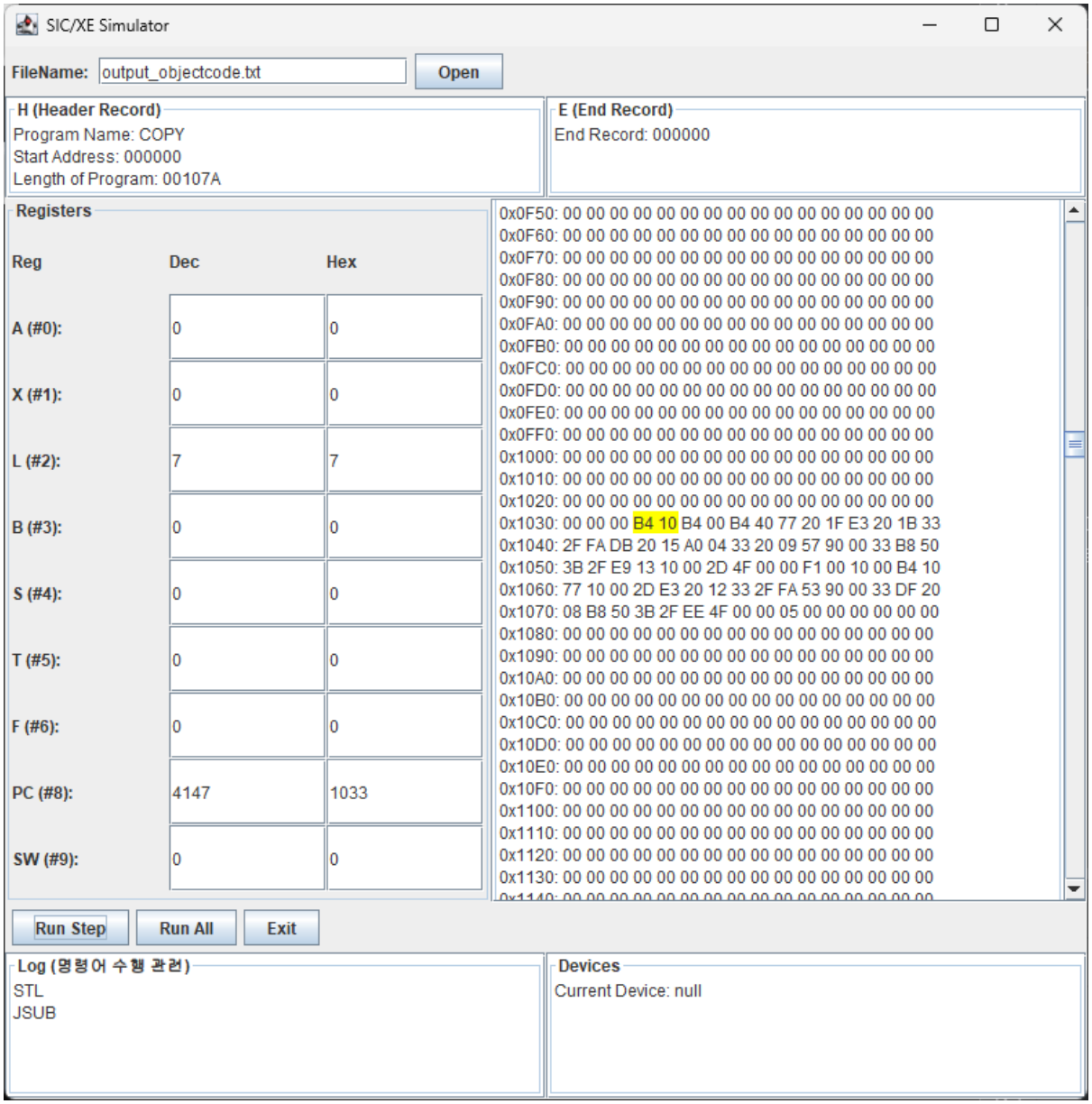
해당 명령어는 STL 이었고, 수행 후에 Log에 STL이 추가되어 출력된 것을 확인 가능하고, PC가 0x0003이 되어 다음 명령어가 노란색으로 강조되어있는 것을 확인 가능하다.

그리고 STL의 결과로 0x002A 의 메모리 부분 (RETADR 라는 WORD 공간)에 0x003000이 저장된 것을 확인 가능하다.

```
FIRST           STL             RETADR  SAVE RETURN ADDRESS

...

RETADR          RESW            1
```



다음 명령어 실행 후의 결과이다.

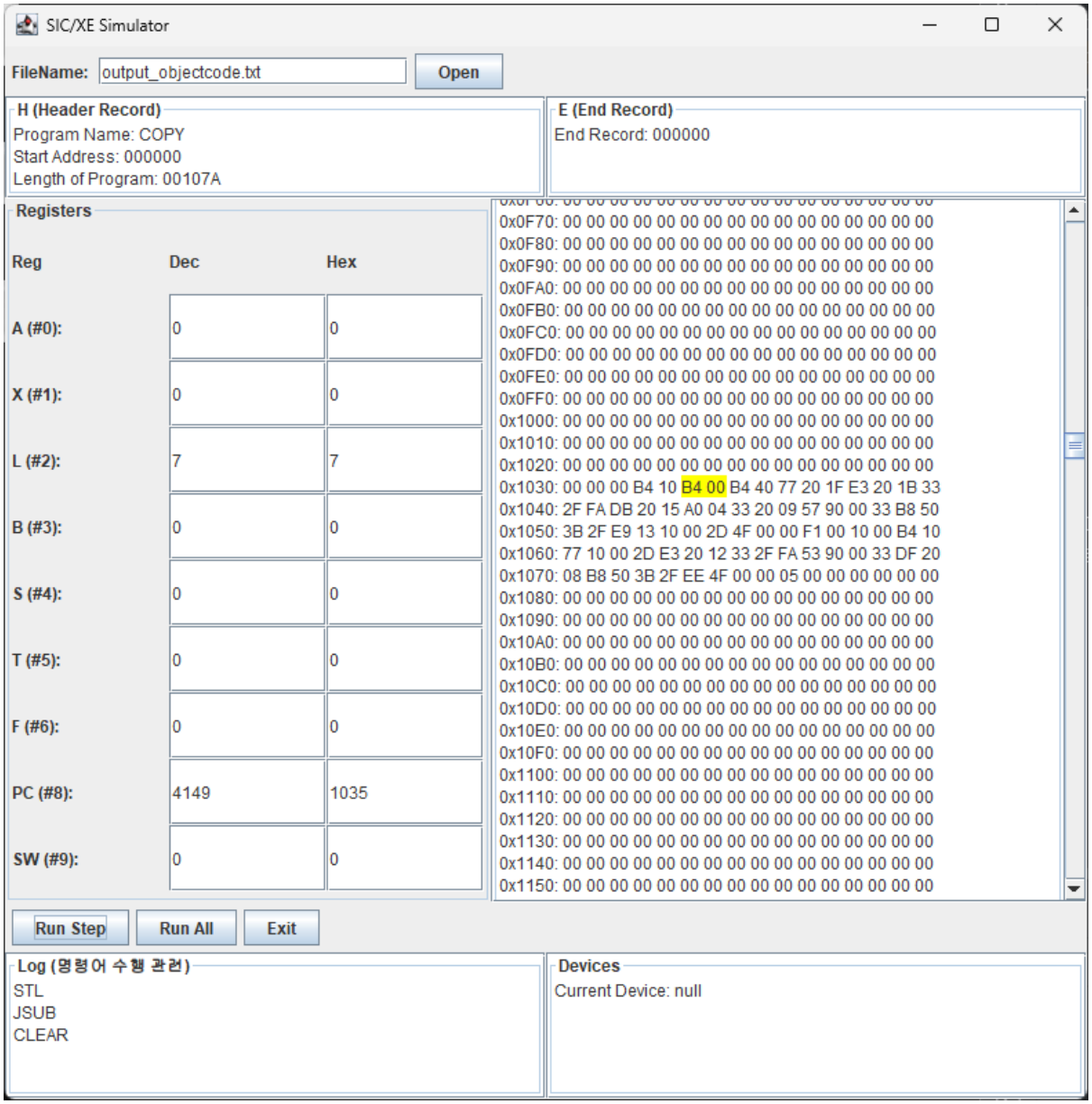
4B1 01033 : CLOOP +JSUB RDREC READ INPUT RECORD

이 실행된 결과이다.

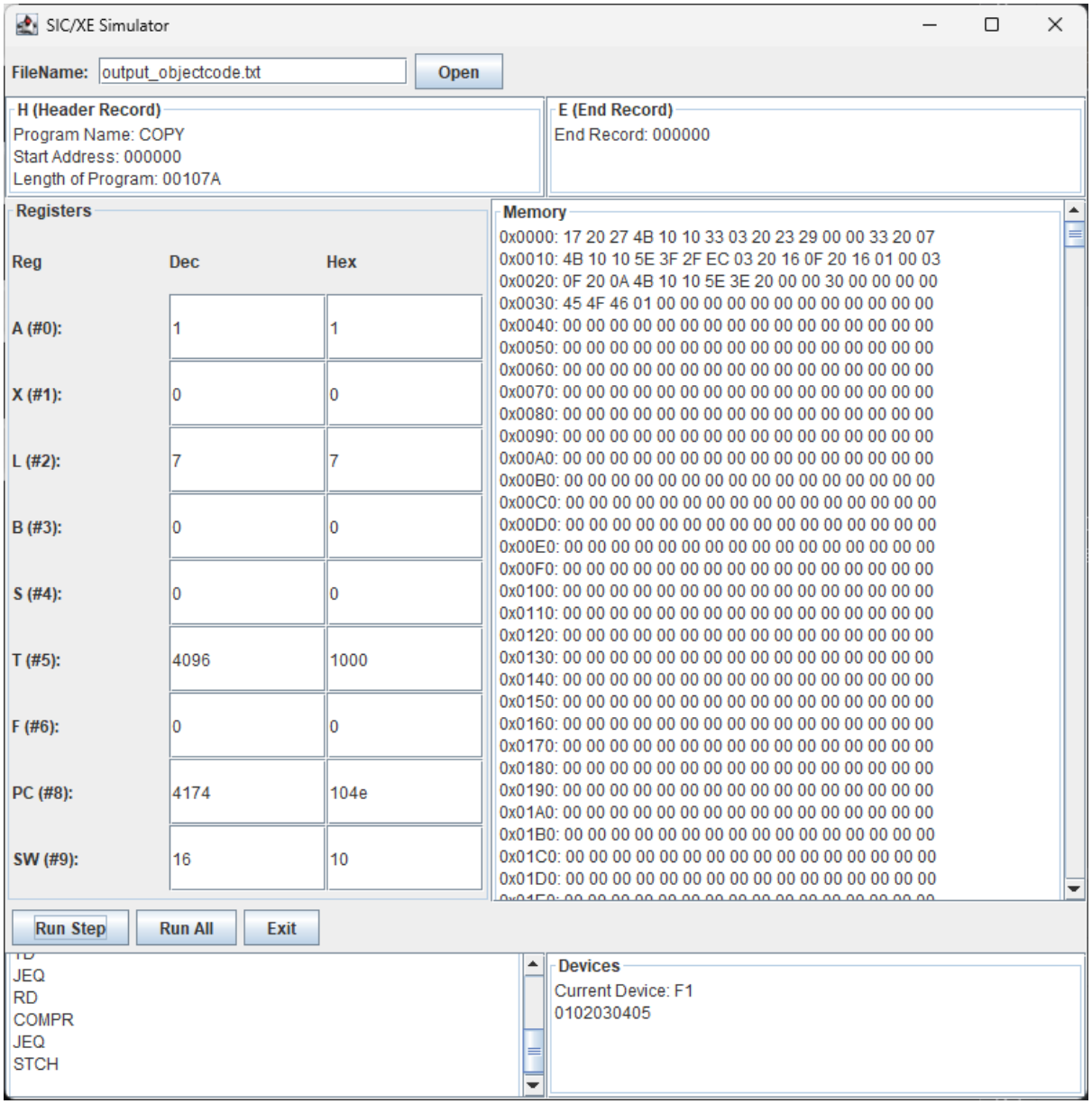
JSUB이 Log에 추가되었다.

그리고 다음으로 실행 할 PC가 0x1033 으로 세팅된 것을 확인 가능하다.

이 영역은 RDREC이고 이 루틴이 끝나서 COPY로 돌아갈때는 L에 저장된 0x000007로 돌아갈 것임을 알 수 있다.



명령어 한개 실행의 결과 B4 10 의 CLEAR가 Log에 추가되었다.



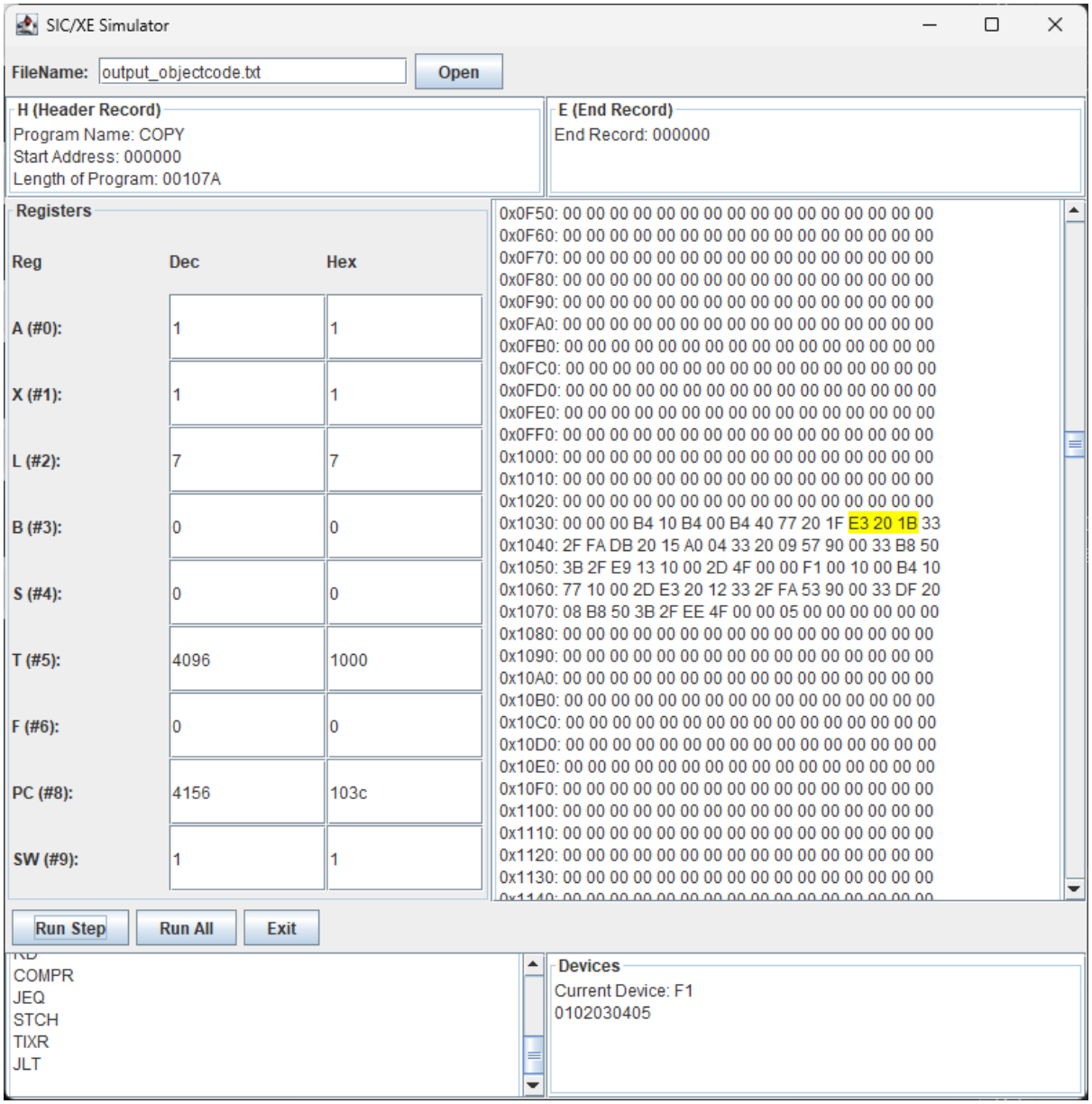
실행을 쭉 하여... STCH 까지 수행 된 결과이다.

TD의 실행으로 설정된 현재 사용중인 'F1' 디바이스가 Devices 정보 창에서 확인가능하다.

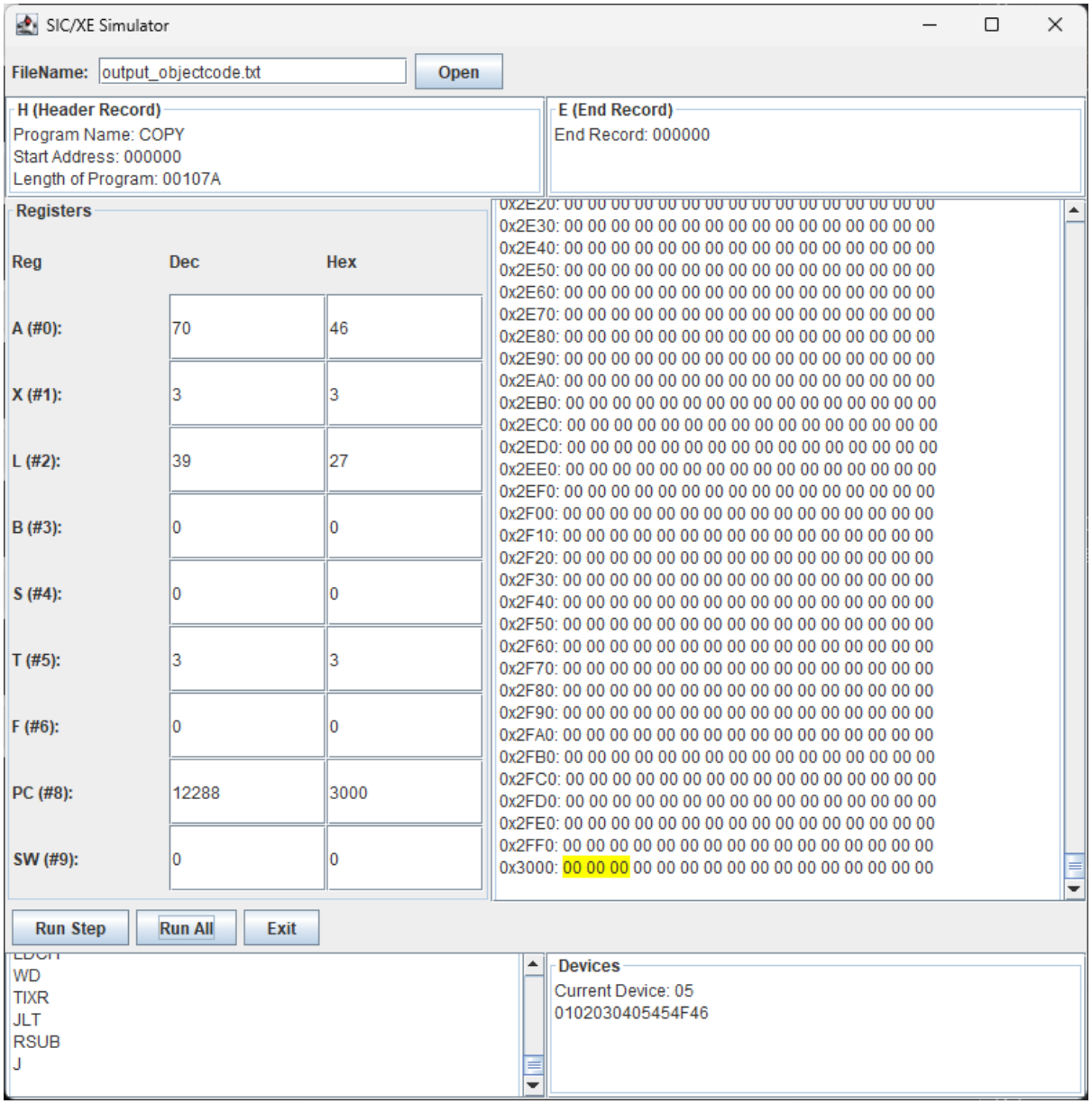
Devices에서는 해당 디바이스의 내용물까지 출력을 해준다.

'F1' 디바이스에는 내용물을 임의로 01 02 03 04 05로 담아놓았기 때문에 담겨있음을 확인 가능하다.

여기서 STCH 의 실행 결과로 0x0033부터 시작한 BUFFER의 영역에 F1 디바이스의 첫 바이트인 01 이 저장되어있음을 확인 가능하다.



RDREC 루틴 속에서 TIXR 과 JLT 명령어로 계속 루프를 돌고있음을 알 수 있다.



Run All 을 실행하여 남아있는 모든 명령어를 실행한 결과이다.

WRREC 의 결과로 '05' 디바이스에 'F1' 의 내용 01 02 03 04 05 가 저장되었고, 마지막에 EOF 인 45 4F 46 까지 저장되었음을 알 수 있다.

```
ENDFIL  LDA     =C'EOF' INSERT END OF FILE MARKER
        STA     BUFFER
        LDA     #3      SET LENGTH = 3
        STA     LENGTH
```

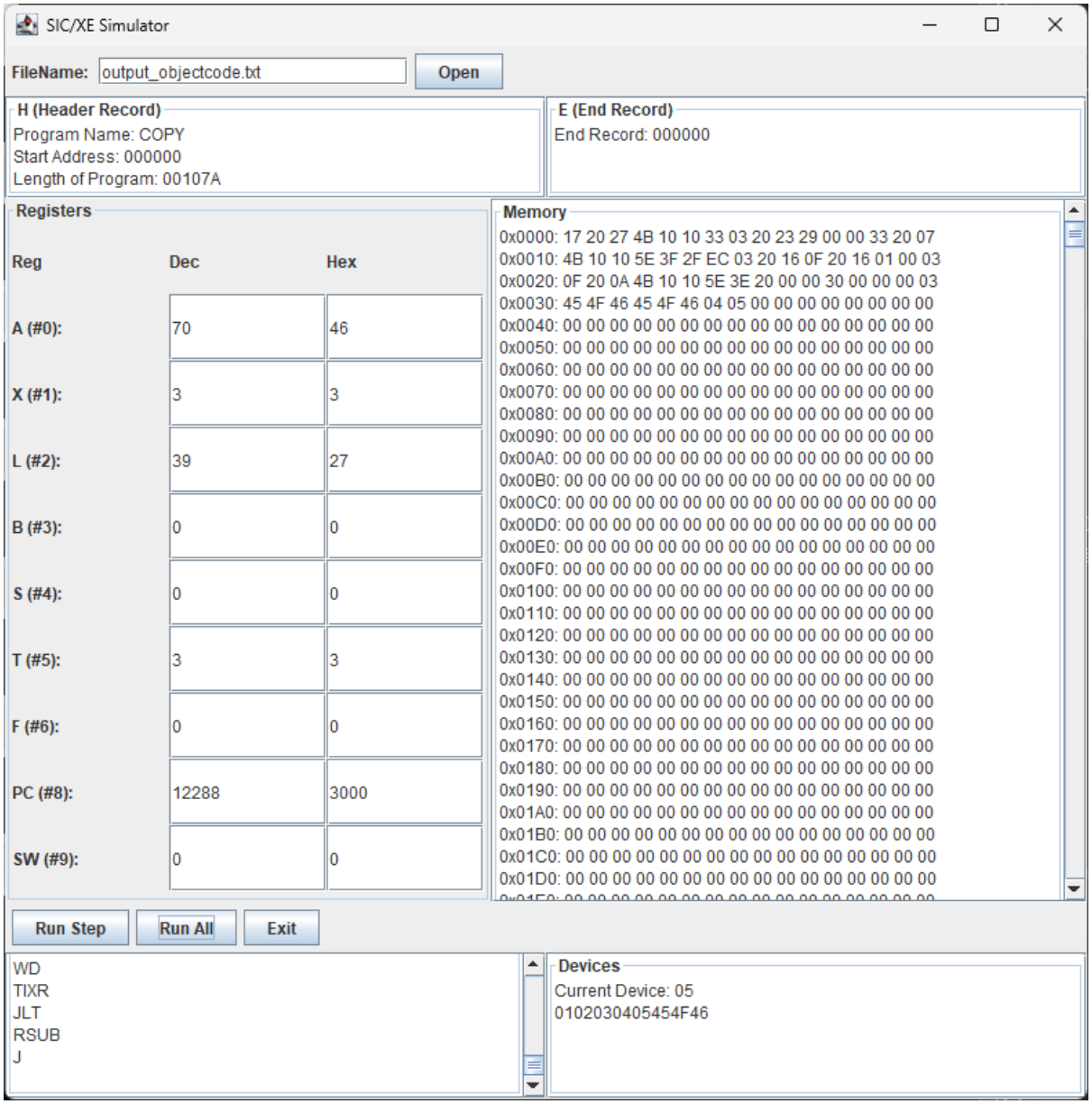
+JSUB WRREC WRITE EOF

이 작업의 결과이다.

그리고 WRREC 서브루틴이 끝나고 돌아와서 실행된

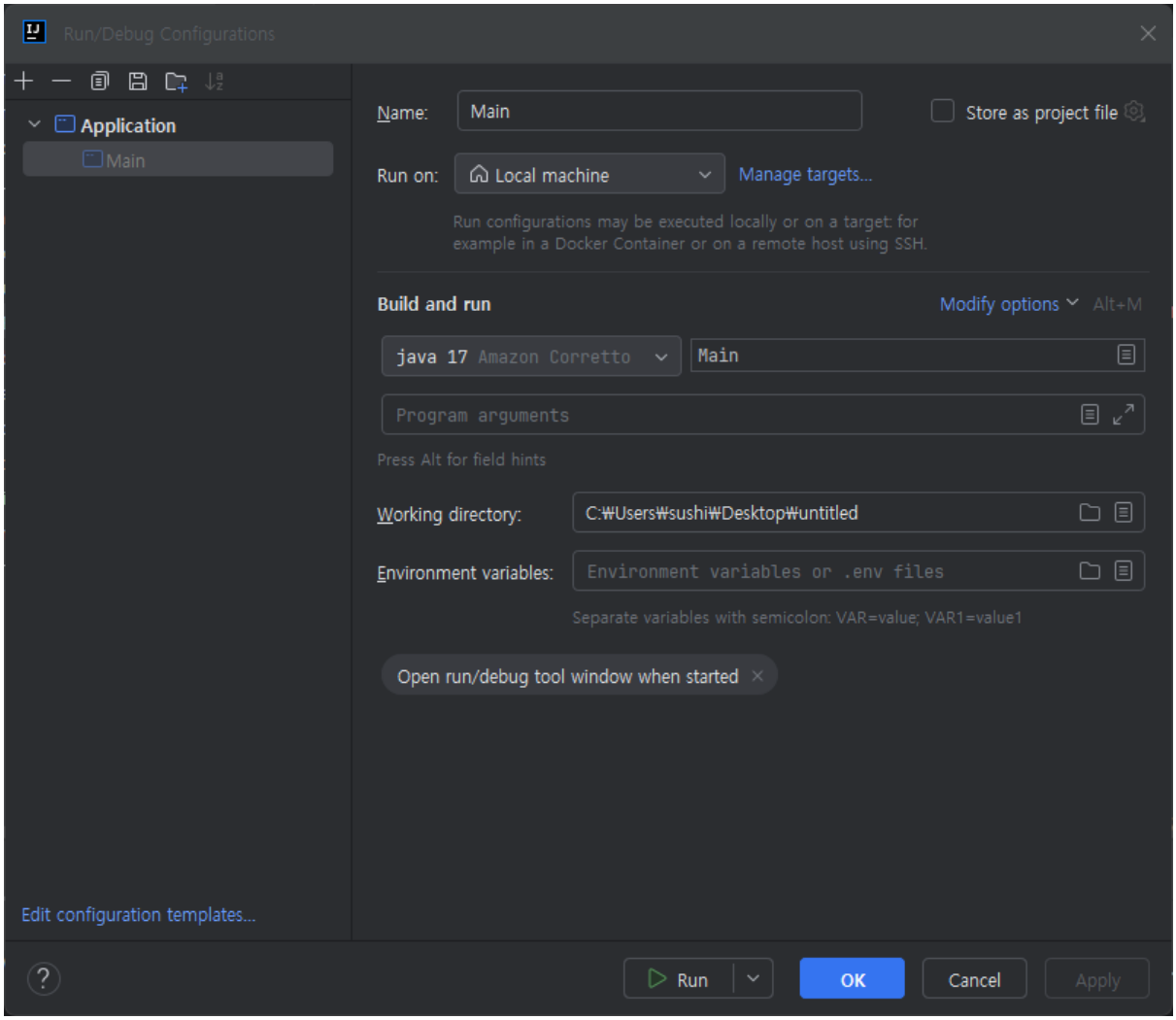
J @RETADR RETURN TO CALLER

의 실행 결과로 맨 처음 RETADR에 저장한 0x003000으로 PC가 설정되었음을 알 수 있다.



모든 실행 후 COPY Section 의 BUFFER 메모리 영역을 확인해보면 45 4F 46 04 05 가 저장되어있음을 알 수 있다.

이로써 COPY 가 정상적으로 작동하여 'F1' 디바이스의 내용을 '05' 디바이스에 복사하고, 그 끝에 EOF 를 추가하였음을 알 수 있다.



실행환경 : 윈도우 11 IntelliJ IDEA 2024 / JAVA 17 Amazon Corretto

4. 결론 및 보충할 점

결론

이번 프로젝트를 통해 SIC/XE 시뮬레이터를 GUI로 구현하여 명령어 실행 과정을 시각적으로 이해할 수 있는 도구를 완성했다. 이를 통해 어셈블리 언어로 작성된 소스코드의 오브젝트 코드가 실제로 어떻게 메모리에 로드되고, 명령어 실행의 작동방식을 직관적으로 이해할 수 있었다.

보충할 점

제한된 명령어 세트

현재 구현된 시뮬레이터는 COPY 프로그램에 사용된 명령어 세트만을 지원한다.

더 많은 명령어를 지원하기 위해 추가적인 구현이 필요하다.

그렇게 되면, 명령어 실행기의 사이즈가 커지기 때문에 명령어별 실행을 정의한 클래스를 분리하여 관리함이 유지 보수에 더 좋을 듯 하다.

입출력 디바이스 시뮬레이션의 단순함

디바이스 시뮬레이션이 간단하게 구현되어 있으며, 실제 하드웨어와의 상호작용을 완벽히 시뮬레이션하지는 않는다. 실제로 디바이스를 TD 하고 RD WD 하는 과정을 해당 디바이스 이름에 해당하는 파일을 오픈하여 거기서 내용을 읽고, 쓰는 작업으로 만들도록 수정하면 좋을 듯 하다.

지금은 COPY 프로그램이 순차적으로 잘 작동하는지 확인하기 위해서 디바이스의 시뮬레이션을 최대한 간단하게 만들어 두었다.

인터페이스 개선

인터페이스를 더 직관적이고 사용하기 쉽게 개선할 필요가 있다.

현재는 임의로 만든 0x3010 크기의 메모리 영역을 보여주고 있다. (ResourceManager.javja : 34 line)

그래서 메모리 영역과 실행중인 명령어를 확인하기 위해 해당 지역까지 스크롤을 해서 확인해야 하는 불편함이 있다. -> 메모리에 내용이 있는 부분만 보여주고, 내용이 없는 부분은 생략하는 로직을 추가하면 메모리 영역이 달라지는 것을 시각적으로 확인하기가 더 좋을 듯 하다. (내용이 차있고 안차있고의 Flag를 설정하여 구현하는 방법 등)

명령어 표시 추가

GUI 에서 노란색으로 하이라이트 된 부분이 Run Step으로 실행할 명령어이다. 그러나 지금의 구현에서는 해당 명령어의 이름을 GUI 에 추가하지는 않았고, 실행 후 Log에 적어 내려 가도록 구현하였다. Instruction 클래스들을 import하여 pc의 opcode를 분석하여 이름을 알아내서 GUI에 추가하는 것도 좋을 듯 하다.