



---

# System Programming

(프로젝트 3 설명)

---

# 프로젝트 1 요약

- 프로젝트1에서 SIC/XE 머신이 실행될 수 있는 가상 환경(shell 환경)을 만들고 SIC/XE 머신이 작동될 수 있도록 준비함.
- Shell과 관련된 명령어, 메모리와 관련된 명령어, opcode와 관련된 명령어 크게 3가지를 구현함.
- Shell 관련 명령어에는 shell에서 사용할 수 있는 명령어들을 보여주는 명령어, 리눅스 ls 명령어와 같이 현재 shell이 작동하고 있는 디렉토리에 있는 파일을 보여주는 명령어, 사용자가 사용한 명령어를 출력해주는 명령어, shell 가상 환경을 종료하는 명령어 4가지를 구현함.
- 메모리와 관련된 명령어는 가상 환경이 사용할 1MB 메모리에 대한 명령어로 메모리 내용을 수정하거나, 확인하는 명령어를 구현했다. 이 메모리와 관련된 명령어는 프로젝트3에서 프로그램이 메모리에 잘 올라갔는지 확인하기 위해 사용됨.
- Opcode와 관련된 명령어는 SIC/XE가 사용하는 Opcode들에 대해서 opcode의 이름 ( ex) ADD )을 key 값으로 하는 해쉬 테이블로 구현한 optable이라는 곳에 미리 저장해두고 opcode를 확인하는 명령어를 구현함. 이 opcode는 프로젝트 2에서 assemble하는 데 사용됨.

## 프로젝트 2 요약

- 프로젝트 1에서 만든 shell에 **assemble** 기능을 추가하여 shell이 **SIC/XE** 어셈블리 코드에 대해서 어셈블할 수 있도록 구현함.
- **Shell** 관련 명령어에 대해서는 어셈블리 결과(object 파일등)를 shell 환경에서 확인할 수 있게 하기 위해서 파일의 내용을 보여주는 명령어를 구현함(리눅스의 **cat**과 같음).
- **Assemble**을 하는 동안 **object code**를 만드는 과정 속에서 **symbol** 주소가 필요한데 **symbol**에 대한 주소를 **1-pass**로 찾는 것이 어려움. 그러므로 어셈블 과정을 **2-pass**로 구현하고 **symbol**에 대한 주소를 **symbol table**에 저장함. 가장 최근에 성공한 어셈블에 대한 **symbol table**의 내용을 확인할 수 있는 명령어를 구현함.
- **Assemble**를 할 수 있게 하는 명령어를 구현함. **SIC/XE** 머신에서 사용하는 어셈블리 어로 작성된 소스 코드를 **input**으로 받으면 코드에 대한 **object code**의 내용이 담겨 있는 **object file**과 코드에 대한 **location counter** 값과 **object code**가 있는 **lst** 파일을 만듦.

## 프로젝트 3 목표

- 프로젝트 2 에서 구현한 셸(shell)에 **linking**과 **loading** 기능을 추가하는 프로그램으로, 프로젝트 2 에서 구현된 **assemble** 명령을 통해서 생성된 **object** 파일을 **link**시켜 메모리에 올리는 일을 수행한다. 다음 네 가지 기능 구현을 위해 필요한 자료구조 및 알고리즘을 구상하여 전체 프로그램을 설계하는 것이 이번 프로젝트의 목표이다.
  - 1) 프로그램을 실행하거나, **load**할 때의 시작 주소를 사용자가 설정할 수 있는 명령어를 구현함.
  - 2) 프로그램들을 **linking**하고 메모리에 **loading**하는 과정을 수행하는 명령어를 구현함.
  - 3) 메모리 상에 올라간 프로그램을 실행하는 명령어를 구현함.
  - 4) **Debug**에 사용되는 **breakpoint**를 지정하는 명령어를 구현함.

## 주소 지정 명령어 설명

- `progaddr [address(16진수)]`을 실행하면, 해당 `address`로 프로그램이 `load` 되거나, `run` 명령어를 수행할 때 해당 `address`에서부터 시작된다. 예를 들면 `progaddr 4000`을 하게 되면 프로그램의 시작 메모리 주소는 4000번지가 된다.
- `Address`에 대한 범위는 SIC/XE 머신은 1MB의 메모리 주소를 사용하기 때문에 16진수로 0~FFFFFF까지 들어올 수 있다.
- 처음 `sicsim shell`이 시작되면 시작 주소에 대한 값은 0x0으로 지정하도록 한다.

$$\begin{aligned} 16^5 - 1 \\ = (2^4)^5 - 1 = 2^{20} - 1 \\ 1M \end{aligned}$$

# Linking Loader에 대한 설명

- 명령어 형식은 **loader [object filename1] [object filename2] [...]**이다. 입력 받은 순서대로 **Filename1,filename2,...**에 대한 **object** 파일을 읽고 **linking**을 해서 **SIC/XE**가 사용하는 가상 메모리(1MB)에 **load**를 한다.
- **Linking loader** 또한 **assemble** 과정과 마찬가지로 **2-pass** 방식으로 수행하도록 구현한다. **Pass1**에서는 외부 심볼에 대한 주소를 지정하고 **pass2**에서는 **linking**과 **loading**을 수행하도록 한다.
- **Linking loader**가 정상적으로 수행이 된다면 수행 후에 **load map**을 화면에 출력하도록 한다. (강의자료 **chapter3 pp.18-21** 참조) **load map**에는 **control section**의 시작 주소와 길이, **symbol**의 주소, 그리고 전체적인 프로그램의 길이가 나타나야 한다.

## 예제를 통한 설명

- 아래 그림은 프로그램 주소를 4000으로 지정하고 **proga.obj**, **progb.obj**, **progc.obj**를 **linking**하고 **loading**한 후에 성공적으로 작동하고 **load map**이 출력된 상황이다.
- **Proga.obj**를 첫번째 인자로 받았기 때문에 **PROGA**이 먼저 올라오게 되는데 주소를 보면 4000으로 시작한다. 이는 **load**하기 전에 **progaddr** 명령어를 통해서 시작 주소를 4000으로 지정했기 때문이다.

```
sicsim>progaddr 4000
sicsim>loader proga.obj progb.obj progcb.obj
control symbol address length
section name
-----
PROGA          4000      0063
      LISTA      4040
      ENDA      4054
PROGB          4063      007F
      LISTB      40C3
      ENDB      40D3
PROGC          40E2      0051
      LISTC      4112
      ENDC      4124
-----
              total length 0133
```

# 예제를 통한 설명

- 성공적으로 프로그램이 메모리 상에 로드 되었을 때 **dump** 명령어로 프로그램이 사용자가 지정한 시작 주소부터 잘 올라와 있는지 확인할 수 있다.
- 아래 그림은 전 슬라이드에서 설명한 **loader proga.obj progb.obj progc.obj**을 성공적으로 실행한 후의 메모리인데, 프로그램의 총 길이가 133이므로 프로그램이 올라가는 시작 주소부터 시작 주소 + 133만큼 메모리 상에 프로그램이 올라가게 된다.

```
sicsim>dump 4000, 4133
04000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
04010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
04020 03 20 1D 77 10 40 C7 05 00 14 00 00 00 00 00 00 ; . w.@.....
04030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
04040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
04050 00 00 00 00 00 41 26 00 00 08 00 40 51 00 00 04 ; ....A&....@Q...
04060 00 00 83 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
04070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
04080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
04090 00 00 00 00 00 00 00 00 00 00 03 10 40 40 77 20 27 ; .....@w '
040A0 05 10 00 14 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
040B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
040C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
040D0 00 00 00 00 41 26 00 00 08 00 40 51 00 00 04 00 ; ....A&....@Q...
040E0 00 83 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
040F0 00 00 00 00 00 00 00 00 00 00 00 03 10 40 40 77 10 ; .....@w.
04100 40 C7 05 10 00 14 00 00 00 00 00 00 00 00 00 00 ; @.....
04110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
04120 00 00 00 00 00 41 26 00 00 08 00 40 51 00 00 04 ; ....A&....@Q...
04130 00 00 83 00 ; .....
```



## Break point에 대한 설명

- 비주얼 스튜디오 등의 **IDE**에서 지원하는 **debug** 기능과 유사하게 **break point**를 설정하는 기능을 구현한다.
- 가상 **shell**에서 **sic/xe** 어셈블리 코드가 잘 작동하는 지 확인하기 위해 **break point**를 설정하고 프로그램을 실행 시 **break point**에서 프로그램을 정지함으로써 디버그를 할 수 있는 기능을 만든다.
- **Breakpoint**를 만나고 정지 후 다시 프로그램을 **run**하면 정지된 **breakpoint**부터 프로그램을 시작해서 다음 **breakpoint**를 만날 때까지 프로그램을 실행한다. **Breakpoint**가 없다면 프로그램 끝까지 실행한다.

## Break point에 대한 설명

- **bp [address]**를 shell에 입력하게 되면 shell은 breakpoint로 입력 받은 address를 지정하게 된다.

```
sicsim> bp 3
[ok] create breakpoint 0003
sicsim> bp 2a
[ok] create breakpoint 002A
sicsim> bp 1046
[ok] create breakpoint 1046
```

- **bp clear**를 입력하면 이때까지 지정한 breakpoint를 전부 삭제한다.

```
sicsim>bp clear
[ok] clear all breakpoints
```

- **bp**만 입력하게 되면 현재 지정된 breakpoint를 화면으로 출력하도록 한다.

```
sicsim>bp
breakpoint
-----
3
1046
2A
```

## 프로그램 실행(run)에 대한 설명

- **Loader** 명령어를 통해서 메모리에 올라간 프로그램을 실행한다. **Progaddr**로 지정한 시작 주소부터 프로그램을 실행하도록 한다.
- 결과로는 **SIC/XE**에서 사용하는 7가지 레지스터(**A, X, L, PC, B, S, T**)의 값이 화면에 출력하도록 한다.
- 메모리에 올라온 프로그램을 실행하다 보면 레지스터의 값이 바뀌게 되는데 이를 구현하면 된다. (예를 들면 **LDA #3**이면 **A register**의 값이 3으로 바뀌게 된다)
- 프로그램은 **breakpoint**를 만나면 정지하고 **breakpoint**가 없으면 끝까지 실행되도록 한다.

# Breakpoint 및 run 명령에 대한 예제

- 아래 코드로 작성된 프로그램(record를 buffer를 통해서 읽고 그 내용을 출력하도록 하는 프로그램)을 0번지부터 올린 후에 breakpoint를 3, 2A, 1046으로 지정을 한 다음에 run 명령어를 4번 수행한 결과

Line	Loc	Source statement	Object code
5	0000	COPY START 0	
10	0000	FIRST STL RETADR	17202D
12	0003	LDB #LENGTH	69202D
13		BASE LENGTH	
15	0006	CLOOP +JSUB RDREC	4B101036
20	000A	LDA LENGTH	032026
25	000D	COMP #0	290000
30	0010	JEQ ENDFIL	332007
35	0013	+JSUB WRREC	4B10105D
40	0017	J CLOOP	3F2FEC
45	001A	ENDFIL LDA ROP	032010
50	001D	STA BUFFER	0F2016
55	0020	LDA #3	010003
60	0023	STA LENGTH	0F200D
65	0026	+JSUB WRREC	4B10105D
70	002A	J @RETADR	3E2003
80	002D	EOF BYTE C'EOF'	454F46
95	0030	RETADR RESW 1	
100	0033	LENGTH RESW 1	
105	0036	BUFFER RESB 4096	
110			
115		SUBROUTINE TO READ RECORD INTO BUFFER	
120			
125	1036	RDREC CLEAR X	B410
130	1038	CLEAR A	B400
132	103A	CLEAR S	B440
133	103C	+LDT #4096	75101000
135	1040	TD INPUT	E32019
140	1043	JEQ RLOOP	332FFA
145	1046	RD INPUT	DB2013
150	1049	COMPR A, S	A001
155	104B	JEQ EXIT	332008
160	104E	STCH BUFFER, X	57C003
165	1051	TIXR T	B850
170	1053	JLT RLOOP	3B2FEA
175	1056	EXIT STX LENGTH	134000
180	1059	RSUB	4F0000
185	105C	INPUT BYTE X'F1'	F1
195			
200		SUBROUTINE TO WRITE RECORD FROM BUFFER	
205			
210	105D	WRREC CLEAR X	B410
212	105F	LDT LENGTH	774000
215	1062	WLOOP OUTPUT	E32011
220	1065	JEQ WLOOP	332FFA
225	1068	LDCH BUFFER, X	53C003
230	106B	WD OUTPUT	DF2008
235	106E	TIXR T	B850
240	1070	JLT WLOOP	3B2FEF
245	1073	RSUB	4F0000
250	1076	OUTPUT BYTE X'05'	05
255		END FIRST	

Ne  
Reloc  
usi  
Modifi  
Rec

# Breakpoint 및 run 명령에 대한 예제

- 첫 번째 breakpoint인 3을 만났을 때 레지스터의 값

```
A : 000000 X : 000000
L : 001077 PC : 000003
B : 000000 S : 000000
T : 000000
```

Stop at checkpoint[3]

- 두 번째 breakpoint인 1046 를 만났을 때 레지스터의 값 (코드에 의하면 1036으로 점프하기 때문에 1046이 먼저 나옴)

```
A : 000000 X : 000000
L : 00000A PC : 001046
B : 000033 S : 000000
T : 001000
```

Stop at checkpoint[1046]

- 세 번째 breakpoint인 2A을 만났을 때 레지스터의 값

```
A : 000046 X : 000003
L : 00002A PC : 00002A
B : 000033 S : 000000
T : 000003
```

Stop at checkpoint[2A]

- 네 번째로 run을 수행했을 때 더 이상 breakpoint가 없으므로 프로그램 끝까지 수행됨. 최종 레지스터의 값

```
A : 000046 X : 000003
L : 00002A PC : 001077
B : 000033 S : 000000
T : 000003
```

End Program