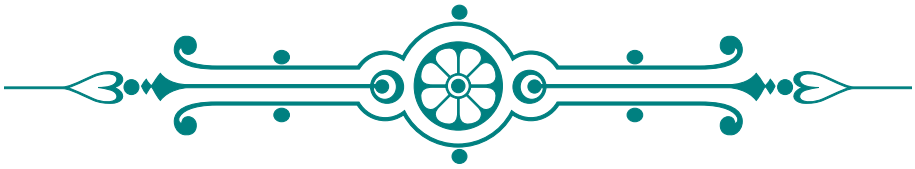


동국 SW 공모대전
SIC emulator 설명서

지도교수 : 홍영식

학번 : 2016110056

학과 : 불교학부

이름 : 박승원

email : zezeon@msn.com

날짜 : 2016년 11월 13일

차 례

제 1 장

개발 목표

- 현재 학교에서 SIC emulator로 쓰고 있는 프로그램은 윈도우에서 버추얼 박스 안에서 실행해야 하는 불편함이 있다.
- 어느 시스템에서나 쓸 수 있게끔 텍스트 파일을 직접 에디트 하고 텍스트 파일로 결과를 받아볼 수 있는 간단한 SIC emulator가 더 편리할 것이다.
- C나 C++로 짜서 어디서든 컴파일 할 수 있게 하고, 소오스를 스스로 고쳐보며 공부하는 것이 훨씬 나을 것이다.
- 스스로의 공부를 위해서도 에뮬레이터를 만들어 보는 과정이 큰 공부가 될 것이다.

제 2 장

소프트웨어 구성, 설명

제 1 절 SIC emulator에서의 Hello World 프로그램

매크로 정의 부분은 매크로 전처리 기능을 보여주기 위해서 넣었다. Hello World 기능과 아무 상관이 없다.

Listing 2.1: 1.s : Hello World in SIC

```
macro save  store , three
  stx   store
  ldx   three
  sta   store , x
  jsub  2
  jsub  2
  jsub  2
   stl   store , x
  ldx   store
mend
macro load  store , three
  ldx   three
  lda   store , x
  jsub  2
  jsub  2
  jsub  2
   ldl   store , x
  ldx   store
mend
   start   1000
loop  ldch   string , x
  wd string
  jsub  2  //x++
  comp end_string
  save  garage , three
```

```
 jeq  quit
 j  loop
 load  garage,three
quit  word 0
string  byte  c'Hello'
 byte  20
 byte  c'World!'
 byte  10
end_string  word 0
garage  resw 3
three  word 3
 end
```

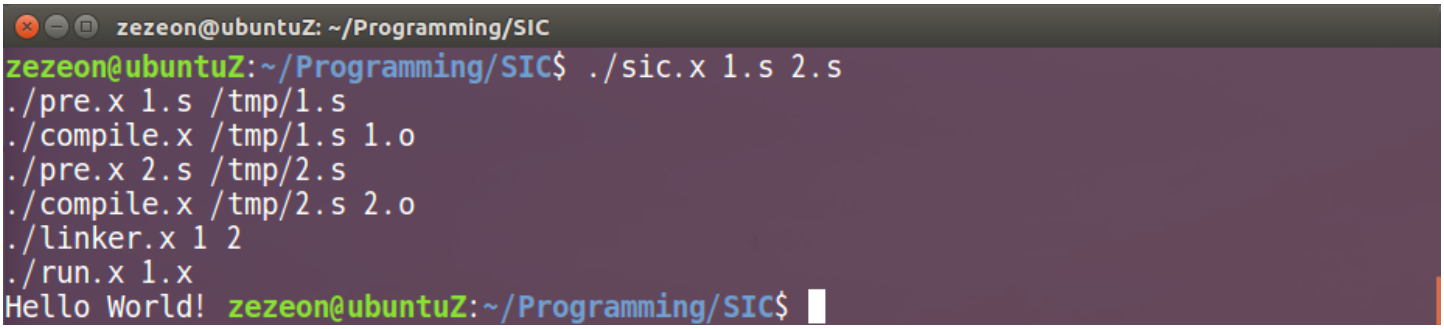
Listing 2.2: 2.s : X 레지스터의 값을 1 증가시키는 모듈

```
  start  1000  X+1 routine
 sta  backup
 stx  backup2
 lda  backup2
 add one
 sta  backup2
 lda  backup
 ldx  backup2
 rsub
backup word 0
backup2 word 0
one word 1
 end
```

파일의 컴파일과 실행 절차

1. 전처리기(pre.x 1.s 1.s) : 1.s⟶1.s
2. 컴파일(compile.x 1.s 1.o) : 1.s ⟶ 1.o 2.s ⟶ 2.o
3. 링킹(linker.x 1 2) : 1.o + 2.o ⟶ 1.x
4. 실행(run.x 1.x) : 1.x ⟶ Hello World!

1+2+3+4=통합 명령 : sic.x 1.s 2.s⟶Hello World!



제 2 절 Preprocessor

source의 규칙

- 소오스는 탭 혹은 공백 없이 시작하면 라벨이나 마크로 정의로 인식한다.
- 탭 또는 공백이 연달아 오는 것은 하나의 공백으로 인식한다.
- 세 번째 공백문자 이후는 모두 주석으로 인식한다.
- 대소문자는 구별하지 않는다.
- SIC의 모든 명령어를 지원한다.(단, SIC/XE는 아님.)
- 마크로 정의는 macro로 시작하고 mend로 끝난다.
- 모듈파일은 반드시 프로그램의 시작 번지(start)보다 작은 숫자로 이름지어야 한다. 이 모듈은 jsub 2 이런 식으로 프로그램에서 호출할 수 있다.
- 소오스의 모든 숫자는 십육진수로 인식한다.

Listing 2.3: 1.s가 전처리된 소스 파일

```
 start  1000
loop ldch  string ,x
 wd string
 jsub  2
 comp end_string
 stx  garage
 ldx  three
 sta  store ,x
 jsub  2
 jsub  2
 jsub  2
 stl  store ,x
 ldx  garage
 jeq  quit
```

```
 j  loop
 ldx  three
 lda  store ,x
 jsub  2
 jsub  2
 jsub  2
 ldl  store ,x
 ldx  garage
quit  word 0
string  byte  48
 byte  65
 byte  6c
 byte  6c
 byte  6f
 byte  20
 byte  57
 byte  6f
 byte  72
 byte  6c
 byte  64
 byte  21
 byte  10
end_string  word 0
garage  resw 3
three  word 3
 end
```

macro가 치환되었고, byte c'Hello'가 byte + 16진수로 바뀌었다.

Listing 2.4: 2.s가 전처리된 소스 파일

```
 start  1000
 sta  backup
 stx  backup2
 lda  backup2
 add one
 sta  backup2
 lda  backup
 ldx  backup2
 rsub
backup word 0
backup2 word 0
```

```
one word 1
 end
```

제 3 절 Compiler

Listing 2.5: 1.o : 오브젝트 파일

```
 start  1000
data  103f
end 105e
1000  509042
1003  dc1042
1006  480002
1009  28104f
100c  101052
100f  04105b
1012  0c8000
1015  480002
1018  480002
101b  480002
101e  148000
1021  041052
1024  30103f
1027  3c1000
102a  04105b
102d  008000
1030  480002
1033  480002
1036  480002
1039  088000
103c  041052
103f  000000
1042  48
1043  65
1044  6c
1045  6c
1046  6f
1047  20
1048  57
1049  6f
```

```
104a 72
104b 6c
104c 64
104d 21
104e 10
104f 000000
1052 000000000000000000
105b 000003
105e 00
```

오브젝트 파일의 규칙

- 처음 세 줄은 시작번지, 데이터 시작 번지, 종료 번지이다.
- 다음부터는 주소와 기계어로 전처리된 소오스와 일대일 대응한다. 이 방식은 T 혹은 M 이후에 숫자를 주욱 나열하는 것보다 훨씬 가독성이 좋다.
- 0x1000번지를 보면 9042로 string,x의 X 레지스터 오프셋을 더하는 어드레스 지정방식이 적용된 것을 알 수 있다.
- jsub 2의 2는 링크 단계에서 주소를 찾아주게 된다.

Listing 2.6: 2.o : 오브젝트 파일

```
 start   1000
data  1018
end 1021
1000 0c1018
1003 10101b
1006 00101b
1009 18101e
100c 0c101b
100f 001018
1012 04101b
1015 4c0000
1018 000000
101b 000000
101e 000001
1021 00
```

제 4 절 Linker

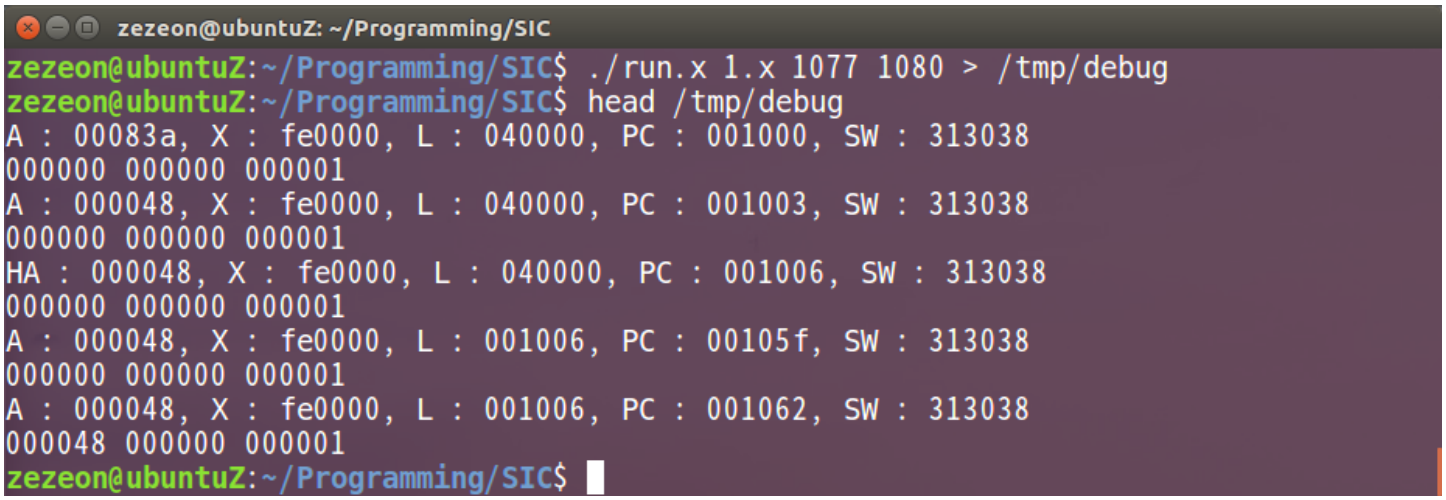
Listing 2.7: 1.x : 1.o와 2.o가 링크된 실행 파일

```
 start  1000
data  103f
end 1080
1000  509042
1003  dc1042
1006  48105f
1009  28104f
100c  101052
100f  04105b
1012  0c8000
1015  48105f
1018  48105f
101b  48105f
101e  148000
1021  041052
1024  30103f
1027  3c1000
102a  04105b
102d  008000
1030  48105f
1033  48105f
1036  48105f
1039  088000
103c  041052
103f  000000
1042  48
1043  65
1044  6c
1045  6c
1046  6f
1047  20
1048  57
1049  6f
104a  72
104b  6c
104c  64
104d  21
104e  10
104f  000000
```

```
1052 000000000000000000
105b 000003
105e 00
105f 0c1077
1062 10107a
1065 00107a
1068 18107d
106b 0c107a
106e 001077
1071 04107a
1074 4c0000
1077 000000
107a 000000
107d 000001
1080 00
```

1015주소를 보면 48105f로 jsub 모듈의 주소가 치환되었다. 2.o의 어드레스가 오프셋에 맞게 변환되었다.

제 5 절 실행기(Interpreter)



디버깅 실행시 옵션을 주면 레지스터와 메모리의 특정 어드레스를 표현할 수 있다.

./run.x [실행화일] [표시할 시작어드레스] [표시할 종료어드레스]

의 형식이다.

컴파일 시에 에러가 발생하는 부분은 그 구문을 에러로 표시해준다.

제 3 장

개발 효과

제 1 절 교육용 ADT로서의 소오스

Listing 3.1: sic.h 헤더화일

```
// sic/xe simulator 헤더파일입니다 .
#include<vector>
#include<functional>
#include<string>
#include<map>

struct Register //24 bit
{
    unsigned char opcode;
    short address; // 1 bit ( direct0 , indexed 1) + 15 bit address
    Register & operator=(int n);
    operator int() ;
};

class SICException : public std :: exception
{
public:
    SICException(std :: string s) : message(s) {}
    virtual const char* what() const throw() {
        return message.data() ;
    }
protected:
    std :: string message;
};

// typedef char byte ;// byte c'z'
```

```
// typedef char[3] word;// 상수 정의 five(addr) word 5

class SIC
{
public:
    Register A, X, L, PC, SW;

protected:
    unsigned char memory[32768];//2**15
    bool is_opcode(std::string s);
    int fetch(short addr) const;
    void store(short addr, Register r);

    std::map<std::string, unsigned char> op_table = {
        {"lda", 0x00}, {"ldx", 0x04}, {"ldl", 0x08},
        {"sta", 0x0c}, {"stx", 0x10}, {"stl", 0x14},
        {"add", 0x18}, {"sub", 0x1c}, {"mul", 0x20}, {"div", 0x24},
        {"comp", 0x28}, {"tix", 0x2c},
        {"jeq", 0x30}, {"jgt", 0x34}, {"jlt", 0x38}, {"j", 0x3c},
        {"jsub", 0x48}, {"rsub", 0x4c},
        {"ldch", 0x50}, {"stch", 0x54},
        {"rd", 0xd8}, {"wd", 0xdc}, {"td", 0xe0}
    };

    std::map<unsigned char, std::function<void(short)>> po_table = {
        {0x00, std::bind(&SIC::LDA, this, std::placeholders::_1)},
        {0x04, std::bind(&SIC::LDX, this, std::placeholders::_1)},
        {0x08, std::bind(&SIC::LDL, this, std::placeholders::_1)},
        {0x0c, std::bind(&SIC::STA, this, std::placeholders::_1)},
        {0x10, std::bind(&SIC::STX, this, std::placeholders::_1)},
        {0x14, std::bind(&SIC::STL, this, std::placeholders::_1)},
        {0x18, std::bind(&SIC::ADD, this, std::placeholders::_1)},
        {0x1c, std::bind(&SIC::SUB, this, std::placeholders::_1)},
        {0x20, std::bind(&SIC::MUL, this, std::placeholders::_1)},
        {0x24, std::bind(&SIC::DIV, this, std::placeholders::_1)},
        {0x28, std::bind(&SIC::COMP, this, std::placeholders::_1)},
        {0x2c, std::bind(&SIC::TIX, this, std::placeholders::_1)},
        {0x30, std::bind(&SIC::JEQ, this, std::placeholders::_1)},
        {0x34, std::bind(&SIC::JGT, this, std::placeholders::_1)},
        {0x38, std::bind(&SIC::JLT, this, std::placeholders::_1)},
```

```
        {0x3c, std::bind(&SIC::J, this, std::placeholders::_1)},
        {0x48, std::bind(&SIC::JSUB, this, std::placeholders::_1)},
        {0x4c, std::bind(&SIC::RSUB, this, std::placeholders::_1)},
        {0xd8, std::bind(&SIC::RD, this, std::placeholders::_1)},
        {0xdc, std::bind(&SIC::WD, this, std::placeholders::_1)},
        {0xe0, std::bind(&SIC::TD, this, std::placeholders::_1)},
        {0x50, std::bind(&SIC::LDCH, this, std::placeholders::_1)},
        {0x54, std::bind(&SIC::STCH, this, std::placeholders::_1)}
    };

private:
    //pseudo instruction
    void start();
    void end();
    void resw(); //1word space allocate
    void resb(); //alpha resb 2; 2byte alloca

    //load and store
    void LDA(short address);
    void LDX(short);
    void LDL(short);
    void STA(short address);
    void STX(short);
    void STL(short);

    // arithmetic
    void ADD(short);
    void ADDX(short);
    void SUB(short);
    void MUL(short);
    void DIV(short);

    //comparison
    void COMP(short);
    void TIX(short);

    // conditional jump
    void JLT(short);
    void JEQ(short);
    void JGT(short);
```

```
    void J(short);

    // subroutine  linkage
    void JSUB(short);
    void RSUB(short);

    //IO.  test  divice .  read  write  device
    void TD(short);
    void RD(short);
    void WD(short);

    //char  handle
    void LDCH(short);
    void STCH(short);
};
```

Listing 3.2: sic.cc 구현화일

```
//SIC/XE  구현부
#include<iostream>
#include"sic.h"
using namespace std;

Register& Register::operator=(int n)
{
    address = n & 0xffff;
    opcode = (n & 0xff0000) >> 16;
    return *this;
}

Register::operator int()
{
    int r = opcode;
    r <<= 16;
    r |= address;
    return r;
}
void SIC::store(short addr, Register r)
{
    memory[addr] = r.opcode;
    memory[addr + 1] = r.address >> 8;
```

```
    memory[addr + 2] = r.address & 0xff;
}
int SIC::fetch(short addr) const
{
    int r = memory[addr];
    r <<= 16;
    int k = memory[addr + 1];
    k <<= 8;
    r |= k;
    r |= memory[addr + 2];
    return r;
}
bool SIC::is_opcode(string s) { return op_table.find(s) != op_table.end(); }

void SIC::LDA(short addr) { A = fetch(addr); }
void SIC::LDL(short addr) { L = fetch(addr); }
void SIC::LDX(short addr) { X = fetch(addr); }
void SIC::STX(short addr) { store(addr, X); }
void SIC::STA(short addr) { store(addr, A); }
void SIC::STL(short addr) { store(addr, L); }

void SIC::ADD(short addr) { A = A + fetch(addr); }
void SIC::SUB(short addr) { A = A - fetch(addr); }
void SIC::MUL(short addr) { A = A * fetch(addr); }
void SIC::DIV(short addr) { A = A / fetch(addr); }

void SIC::LDCH(short addr) { A = memory[addr]; }
void SIC::STCH(short addr) { memory[addr] = A.address & 255; }

void SIC::COMP(short addr)
{
    int n = fetch(addr);
    int a = A;
    if(a == n) SW.opcode = 0;
    else if(a < n) SW.opcode = 1;
    else SW.opcode = 2;
}
void SIC::TIX(short addr) {
    X = X + 1;
    int a = A;
```

```
    A = X;
    COMP(addr);
    A = a;
}

void SIC::JEQ(short addr) { if(!SW.opcode) PC = addr - 3; }
void SIC::JGT(short addr) { if(SW.opcode == 2) PC = addr - 3; }
void SIC::J(short addr) { PC = addr - 3; }
void SIC::JLT(short addr) { if(SW.opcode == 1) PC = addr - 3; }
void SIC::JSUB(short addr)
{
    int pc = PC;
    int k = addr;
    L = pc;
    PC = k - 3;
}
void SIC::RSUB(short addr) { PC = L; }

void SIC::RD(short addr) {
    char c;
    cin >> c;
    A = (int)c;
}
void SIC::WD(short addr) { cout << (char)(A.address & 0xff); }
void SIC::TD(short addr) {}
```

교육과정상 C++을 먼저 배우고, 시스템 소프트웨어를 배우는데, C++을 이해하면 위의 헤더파일만 봐도 어느 정도 SIC의 작동원리를 이해할 수 있다. 하드웨어를 추상적으로 이해하는 데에 소오스 자체가 추상자료형을 제공하는 셈이다.

제 2 절 SIC 에뮬레이터로서의 간편함

- 커맨드 라인 환경이기에 어느 시스템에서나 호환성이 높다.
- 이미 앞서서 오브젝트 파일에서 보았듯이, 기계어 코드가 매우 시인성이 높게 볼 수 있으며, 단순한 텍스트 파일이기 때문에, 얼다든지 직접 에디트하여 실행을 할 수도 있다.
- 중간 처리 과정을 하나하나 분리하여 그 중간 결과물들을 텍스트로 볼 수 있다.
- 소스를 직접 고쳐보며 배울 수 있다.