

Design Report

SIC/XE Disassembler

마이크로컴퓨터 설계 (19-2, ECE452) / 이정원 교수님

201120696 최제헌

요구사항

- SIC/XE assembly program으로 disassemble 하는 프로그램
32p object program & SymbolTable → 10~12p SIC/XE (sp03_jwlee.ppt)

```

HCOPY 000000001077
T0000001D17202D69202D4B1010360320262900003320074B10105D3F2FEC032010
T00001D130F20160100030F200D4B10105D3E2003454F46
T0010361DB410B400B44075101000E32019332FFA0B2013A00433200857C003B850
T0010531D3B2FEA1340004F0000F1B410774000E32011332FFA53C003DF2008B850
T001070073B2FEF4F00005
M00000705
M00001405
M00002705
E000000

```

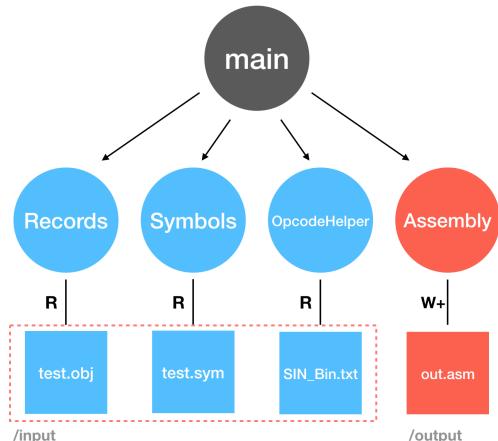


Line	Loc	Source statement			Object code
5	0000	COPY	START	0	
10	0000	FIRST	STL	RETADR	17202D
12	0003		LDB	#LENGTH	69202D
13			BASE	LENGTH	
15	0006	CLOOP	+JSUB	RDREC	4B101036
20	000A		LDA	LENGTH	032026
25	000D		COMP	#0	290000
30	0010		JEQ	ENDFIL	332007
35	0013		+JSUB	WRREC	4B10105D
40	0017		J	CLOOP	3F2FEC
45	001A	ENDFIL	LDA	EOF	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	RLOOP	TD	INPUT	E32019
140	1043		JEQ	RLOOP	332FFA
145	1046		RD	INPUT	DB2013
150	1049		COMPR	A,S	A004
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
190	.				
200	.		SUBROUTINE TO WRITE RECORD FROM BUFFER		
205	.				
210	105D	WRREC	CLEAR	X	B410
212	105F		LDT	LENGTH	774000
215	1062	WLOOP	TD	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	

프로그램 전체 구조

1. 파일 구조

```
SIC-XE-Disassembler      # root
├── Assembly.py
├── OpcodeHelper.py
└── Records.py
├── Symbols.py
└── __main__.py
├── input
│   └── SIC_Bin.txt      # OPTAB
└── output
    └── test.obj        # Object program
        └── test.sym      # SYMTAB
    └── out.asm          # Assembly file
```



*개발환경: Python 3.7.4 (별도 외부 디펜던시 X)

__main__.py에서 프로그램이 실행되며, Records.py, Symbols.py, OpcodeHelper.py, Assembly.py의 각 파일의 자료구조, 함수들을 사용하여 Disassemble을 수행한다.

Records.py, Symbols.py, OpcodeHelper.py는 이미 주어진 input 파일들을 다루며 Assembly.py는 Disassemble을 통해 만들어질 output Assembly 파일을 다룬다.

2. 프로그램 PseudoCode

```
# __main__.py:
begin
    # make dict by input files
    records = ReadRecord(OBJ_Prog)
    symbols = ReadSymbol(SYMTAB)
    opcodes = ReadOpcodes(OPTAB)

    # create assembly dict with Head, End records
    assembly = AssemblyDict(record.H, record.E)

    # add symbol infos
    assembly.addSymbols(symbols)

    # add Text records infos with opcodes
    assembly.addTextRecords(records.T, opcodes)

    # write assembly file
    assembly.writeFile()

end
```

root directory에서 __main__.py를 실행하면, OpcodeHelper, Records, Symbols의 각 Read 함수를 통해 OPTAB, Obj Program, SYMTAB 파일 데이터를 읽고 가공하여 dictionary를 만든다.

읽어온 정보들로 일련의 처리과정을 거치며, Assembly의 각 line의 Loc을 key로 갖고, 정보를 value로 갖는 AssemblyDict를 완성하고, out.asm 파일을 만들고 쓰면서 disassemble을 완료하고 프로그램이 종료된다.

파일 별 자료구조 설명

Records.py

```
▼ c RecordType(Enum)
  f E
  f H
  f M
  f T

▼ c Record
  m __init__(self, rtype, name, addr, length, code=None)
  f code
  f length
  f programName
  f recordType
  f startingAddr
  f ReadRecords(OBJ_DIR)
```

• Header	H
Col. 1	Program name
Col. 2~7	Starting address of object program (hex)
Col. 8~13	Length of object program in bytes (hex)
Col. 14-19	
• Text	T
Col. 1	Starting address for object code in this record (hex)
Col. 2~7	Length of object code in this record in bytes (hex)
Col. 8~9	Object code, represented in hex (2 col. per byte)
Col. 10-69	
• End	E
Col. 1	Address of first executable instruction in object program (hex)
Col. 2~7	
	Modification record:
Col. 1	M
Col. 2~7	Starting location of the address field to be modified, relative to the beginning of the program (hexadecimal)
Col. 8~9	Length of the address field to be modified, in half-bytes (hexadecimal)

Object Program의 4가지 종류의 Record(Header, Text, End, Modification)을 모두 저장할 수 있는 Record class를 만들었고, 추가적으로 RecordType Enum을 만들어 사용하였다. ReadRecords 함수를 통해 Object Program 파일을 읽고, 각 Record별로 Record instance를 생성하여 startingAddr를 key로, Record를 value로 갖는 Records dictionary를 만든다.

Symbols.py

```
c Symbol
  m __init__(self, label, loc, flag, dType=None)
  f dType
  f flag
  f label
  f loc
  f ReadSymbols(SYM_DIR)
```

기본적으로 Symbol Table의 필수 정보들을 저장할 수 있도록 하였고 추가적으로 assembler directives Mnemonic 처리를 위한 WORD, BYTE, RESW, RESB 정보를 갖는 dType 멤버변수를 넣었다.

ReadSymbols 함수를 통해 SYMTAB 파일을 읽고, 각 symbol별로 Symbol instance를 생성하여 loc을 key로, Symbol을 value로 갖는 Symbols dictionary를 만든다.

OpcodeHelper.py

```
▼ v OPCODE_DIR
▼ c OpcodeFormatType(Enum)
  f ONE
  f THREE_OR_FOUR
  f TWO
▼ c Opcode
  m __init__(self, mnemonic, opFormat, opcode, op, effect, notes=None)
  f effect
  f mnemonic
  f notes
  f op
  f opFormat
  f opcode
▼ c OpcodeHelper
  m __init__(self)
  m readOpcodes(self, OPCODE_DIR)
  m getOpcode(self, opcode)
  m getRegisterMnemonic(self, value)
  f opcodes
```

Opcode Table의 각 라인당 필수 정보들을 저장할 수 있도록 Opcode class를 만들었다.

추가적으로 OpcodeFormatType Enum을 만들어 사용하였다.

OpcodeHelper instance를 만들면서 readOpcodes 함수를 통해 OPTAB 파일을 읽고

각 opcode별로 Opcode instance를 생성하여 opcode 값을 key로 갖는 opcodes dictionary를 만든다.

앞서 Records.py, Symbols.py와 조금 다른점은 Helper적인 method들을 함께 사용하도록 디자인하였기에
OpcodeHelper instance를 통해 opcodes dictionary에 접근할 수 있도록 하였다.

getOpcode는 opcode 값을 넣으면 해당하는 Opcode instance를 반환하는 method이고,

getRegisterMnemonic은 Register의 index를 넣으면 Register의 Mnemonic을 반환하는 method이다.

Assembly.py

```
▼ c AssemblyLine
  m __init__(self, loc, label=None, mnemonic=None, operand=None, objCode=None)
  f extendedFlag
  f immediateFlag
  f indexAddrFlag
  f indirectFlag
  f label
  f loc
  f mnemonic
  f objCode
  f operand
▼ c AssemblyDict(dict)
  m __init__(self, header, end)
  m writeAssemblyFile(self, ASM_DIR)
  m makeString(self, asm)
  f end
  f header
```

어셈블리의 각 라인 정보를 갖는 AssemblyLine class와

Dict를 상속받은 AssemblyDict class를 만들었다.

AssemblyDict의 key는 Loc이고, value는 AssemblyLine instance이다.

AssemblyDict에는 현재 갖고 있는 AssemblyLine 정보들로 assembly 파일을 만드는 method와

AssemblyDict 내의 각 AssemblyLine을 가독성있게 formatting해주는 makeString method가 있다.

주요로직 설명

Read input files & Create dictionary

```
__main__.py
13 if __name__ == '__main__':
14     # input
15     hRecord, tRecords, mRecords, eRecord = rc.ReadRecords(OBJ_DIR)
16     symbols = sym.ReadSymbols(SYM_DIR)
17     ochelper = oh.OpcodeHelper()
```

프로그램을 실행하면 가장 먼저 Record, Symbol, Opcode 파일들을 읽고, dictionary를 생성하는 작업을 수행한다.

Records.py

```
18
19 def ReadRecords(OBJ_DIR):
20     hRecord = None
21     tRecords = dict()
22     mRecords = dict()
23     eRecord = None
24
25     f = open(OBJ_DIR, 'r')
26     for line in f.readlines():
27         line = line.replace('\n', '')
28
29         if line[0] == RecordType.H.value:
30             hRecord = Record(RecordType.H.value, line[1:7], int(line[7:13], 16), int(line[13:19], 16))
31
32         elif line[0] == RecordType.T.value:
33             tRecord = Record(RecordType.T.value, None, int(line[1:7], 16), int(line[7:9], 16), line[9:])
34             tRecords[tRecord.startingAddr] = tRecord
35
36         elif line[0] == RecordType.M.value:
37             mRecord = Record(RecordType.M.value, None, int(line[1:7], 16), int(line[7:9], 16))
38             mRecords[mRecord.startingAddr] = mRecord
39
40         elif line[0] == RecordType.E.value:
41             eRecord = Record(RecordType.E.value, None, int(line[1:7], 16), None)
42
43     f.close()
44
45     return (hRecord, tRecords, mRecords, eRecord)
```

Object Program의 각 라인별로 Record를 읽어오고, SIC/XE Object Record 스펙에 맞춰 각 Type을 구분하고 각 Type별로 정보들을 구분한 정보를 Record 인스턴스를 만들어 넣는다.

또한, 생성된 각 Record 인스턴스는 Records dictionary에 startingAddress를 key로 저장하였다. Symbols, Opcodes도 같은 과정을 거치며 dictionary를 생성한다.

* mRecord의 경우 Disassemble 할 때에 필요 없는 정보이지만 생성하였다.

Read input files & Create dictionary

```
__main__.py
19     # Header & End
20     header = asm.AssemblyLine(hRecord.startingAddr, hRecord.programName, 'START', hRecord.startingAddr)
21     end = asm.AssemblyLine(None, mnemonic='END', operand=symbols[eRecord.startingAddr].label)
22
23     # for output
24     BASE_R = 0
25     LOCCTR = 0
26     assembly = asm.AssemblyDict(header, end)
```

앞서 input files로 만든 dictionaries를 해석하며 어셈블리 정보를 저장할 용도로 사용할 assemblyDict를 생성하였다. 생성하며 header와, end AssemblyLine 정보를 함께 만들어서 미리 넣어두었다.

또한, output assembly의 정확한 operand location 계산을 위해 BASE 레지스터 값을 의미하는 BASE_R과 현재 해석 중인 라인의 Loc 정보를 갖고있는 LOCCTR 변수를 만들고, 초기화하였다.

AssemblyDict에 Symbols 정보 추가

__main__.py

```
28 # Symbol
29 locList = list(symbols.keys())      # Symbol locs
30 locList.extend(tRecords.keys())    # Text locs
31 locList.append(hRecord.length)     # program max length
32 locList.sort()
33
34 for sLoc, s in symbols.items():
35     operand = None
36     if s.dType == 'BYTE' or s.dType == 'RESB' or s.dType == 'WORD' or s.dType == 'RESW':
37         nextLoc = 0
38         for loc in locList:
39             if sLoc < loc:
40                 nextLoc = loc
41                 break
42
43         size = nextLoc - sLoc
44         operand = size      # BYTE or RESB
45         if s.dType == 'WORD' or s.dType == 'RESW':
46             operand = int(size / 3)
47
48     assembly[sLoc] = asm.AssemblyLine(s.loc, s.label, s.dType, operand)
49
```

assemblyDict의 loc에 Symbol label을 미리 저장해서
Text Record를 해석할 때에 symbol label을 바로 처리할 수 있도록 하였다.

또한, BYTE, RESB, WORD, RESW의 dType을 가지는 Symbol의 경우,
Symbols Loc keys, TextRecords Loc keys, Program length 정보를 하나의 리스트에 합치고, 오름차순 정렬한 locList를 사용하여
다음 assembly line의 loc을 찾아내고, 현재 Symbol의 크기를 미리 계산하여 크기 정보를 operand에 할당하였다.

TextRecord 해석 및 AssemblyDict에 정보 추가

__main__.py

```
50 # Text
51 for tLoc, t in tRecords.items():
52
53     LOCCTR = tLoc
54     finishLoc = tLoc + t.length
55
56     while LOCCTR < finishLoc:
57         curAsm = None
58         if assembly.get(LOCCTR) == None:
59             curAsm = asm.AssemblyLine(LOCCTR)
60         else:
61             curAsm = assembly[LOCCTR]
62
```

Text Record는 전체 Length는 알지만, 몇 개의 instruction으로 이루어져있는지 알 수 없으므로
현재의 LOCCTR가 전체 Length에 도달할 때까지 TextRecord를 해석, AssemblyDict에 정보 추가하는 작업을 반복하도록 하였다.

앞서 symbols에서 처리한 적있던 동일한 loc의 정보를 처리할 때엔,
AssemblyDict에 symbols 정보를 추가하면서 만든 AssemblyLine 인스턴스를 사용하여 값을 업데이트하였다.

```

main__.py
63     startIdx = (LOCCTR - tLoc) * 2
64     curHalfByteLen = 0
65
66     # BYTE, WORD directives
67     if curAsm.mnemonic == 'BYTE':
68         curHalfByteLen = curAsm.operand * 2
69         curAsm.objCode = t.code[startIdx: startIdx + curHalfByteLen]
70
71     elif curAsm.mnemonic == 'WORD':
72         curHalfByteLen = curAsm.operand * 3 * 2
73         curAsm.objCode = t.code[startIdx: startIdx + curHalfByteLen]
74
75     else:
76         curOpcode = int(t.code[startIdx: startIdx + 2], 16) & 0b11111100
77         curFormat = ocHelper.getOpcode(curOpcode).opFormat
78
79         # format checking
80         if curFormat == oh.OpcodeFormatType.ONE.value:
81             curHalfByteLen = 2
82             curAsm.mnemonic = ocHelper.getOpcode(curOpcode).mnemonic.split(' ')[0]
83             curAsm.objCode = t.code[startIdx: startIdx + curHalfByteLen]
84
85
86         elif curFormat == oh.OpcodeFormatType.TWO.value:
87             curHalfByteLen = 4
88
89             m = ocHelper.getOpcode(curOpcode).mnemonic.split(' ')
90             curAsm.mnemonic = m[0]
91             curAsm.objCode = t.code[startIdx: startIdx + curHalfByteLen]
92
93             r12 = int(curAsm.objCode[2:4], 16)
94             r1 = ocHelper.getRegisterMnemonic(r12 >> 4)
95             r2 = ocHelper.getRegisterMnemonic(r12 & 0b1111)
96
97             if len(m) >= 2:
98                 if 'r1' in m[1]:
99                     curAsm.operand = r1
100
101            if len(m) >= 3:
102                if 'r2' in m[2]:
103                    curAsm.operand += ', ' + r2
104
105            if curAsm.mnemonic == 'CLEAR':
106                if r1 == 'B':
107                    BASE_R = 0
108

```

startIdx: 현재 해석을 시작하려는 code의 half-byte index 값을 의미

curHalfByteLen: 현재 해석을 시작하려는 code의 half-byte length를 의미

BYTE, WORD의 경우 opcode table을 통해서는 해석할 수 없지만

Record에 objCode 값이 포함되어 있기 때문에 먼저 확인하여 예외처리를 해준다.

앞서 Symbols 정보 추가 작업을 했으므로

BYTE, WORD directives인 경우 mnemonic에 BYTE, WORD 정보가 들어가 있다.

curHalfByteLen을 계산하고, objCode를 할당해준다.

다음으로 opcode를 해석하고, format을 찾아내고, AssemblyLine 정보를 업데이트한다.

BASE Register를 CLEAR하면 assembly에 영향을 미치므로 BASE_R 값 초기화로직만 따로 추가하였다.

위 코드는 formatType 1, 2까지 나와있는데, 이어서 formatType 3도 설명하겠다.

main_.py

```
109         else: # 3/4
110             ni = int(t.code[startIdx + 1], 16) & 0b11
111             x = int(t.code[startIdx + 2], 16)
112             xbpe = ni & 0b1000
113             x = xbpe & 0b1000
114             b = xbpe & 0b0100
115             p = xbpe & 0b0010
116             e = xbpe & 0b0001
117
118             curHalfByteLen = e == 1 and 8 or 6
119             m = ocHelper.getOpcode(curOpCode).mnemonic.split(' ')
120
121             curAsm.mnemonic = m[0]
122             curAsm.objCode = t.code[startIdx: startIdx + curHalfByteLen]
123
124             # set flags
125             if ni == 0b10:
126                 curAsm.indirectFlag = True
127             elif ni == 0b01:
128                 curAsm.immediateFlag = True
129                 if x != 0:
130                     curAsm.indexAddrFlag = True
131                 if e != 0:
132                     curAsm.extendedFlag = True
133
134             # calculate operand loc & find loc label
135             operandLoc = int(curAsm.objCode[3:], 16)
136
137             if p != 0:
138                 # -2048 <= disp <= 2047
139                 if operandLoc > pow(2, 11) - 1:
140                     operandLoc -= pow(2, 12)
141                 elif operandLoc < -pow(2, 11):
142                     operandLoc += pow(2, 12)
143                 operandLoc += LOCCTR + int(curHalfByteLen / 2)
144
145             elif b != 0:
146                 # 0 <= disp <= 4095
147                 operandLoc += BASE_R
148
149             if len(m) >= 2 and m[1] == 'm':      # except RSUB
150                 curAsm.operand = operandLoc
151
152             if symbols.get(operandLoc) != None and operandLoc != 0:
153                 curAsm.operand = symbols[operandLoc].label
154
155             # set BASE register (for relative loc label)
156             if curAsm.mnemonic == 'LDB':
157                 if ni == 0b10: # not working, TBD
158                     BASE_R = assembly[operandLoc].operand
159                 elif ni == 0b01:
160                     BASE_R = operandLoc
161                 else:
162                     BASE_R = LOCCTR
163
164             # update assembly line & set next loc
165             assembly[LOCCTR] = curAsm
166             LOCCTR += int(curHalfByteLen / 2)
```

formatType 3/4의 경우 SIC/XE 스펙에 따라 먼저 n, i, x, b, p, e 값을 찾아냈다.

e flag를 확인하여 format size를 확인하고

indirect, immediate, index, pc-relative, base-relative addressing에 맞게 flag 값을 세팅하고, operand Loc을 계산하였다.
pc-relative의 경우 python에서 overflow가 발생하지 않아서 11bits 2의 보수의 맞게 따로 계산을 하였다.

operandLoc 계산이 완료되면 operand에 m이 위치하는지 opcode mnemonic을 확인하고
operandLoc에 symbol label이 있으면 operand에 symbol label을 넣어주도록 하였다.

또한, LDB opcode의 경우 따로 BASE Register 값을 갱신하도록 처리하였는데

앞서 CLEAR에 BASE Register를 따로 처리하였던 것처럼, BASE Register의 값은 assembly에 영향을 미치므로 추가한 것이다.
다만, indirect의 경우 아직 생성하지 않은 AssemblyLine의 주소를 가리킬 수 있으므로 작동하지 않을 수 있다.

마지막으로 LOCCTR의 assemblyDict를 갱신하고, 다음 instruction을 가리키도록 LOCCTR 값을 추가하였다.
TEXT Record를 전부 해석할 때까지 이 과정을 반복 수행한다.

Write output assembly file

__main__.py

```
168     # Write Assembly file
169     assembly.writeAssemblyFile(ASM_DIR)
170
171     print('finish disassemble!')
```

모든 정보들을 해석 완료한 후, writeAssemblyFile을 호출하여 assembly 파일을 만든다.

Assembly.py

```
15 class AssemblyDict(dict):
16     def __init__(self, header, end):
17         dict.__init__(self)
18         self.header = header
19         self.end = end
20
21
22     def writeAssemblyFile(self, ASM_DIR):
23         f = open(ASM_DIR, 'w+')
24         f.write('Loc\t\t\tSource statement\t\t\tObject code\n')
25         f.write(self.makeText(self.header))
26
27         keys = list(self.keys())
28         keys.sort()
29         for key in keys:
30             asm = self[key]
31             f.write(self.makeText(asm))
32             if asm.mnemonic == 'LDB':    # BASE directive
33                 f.write(self.makeText(AssemblyLine(None, mnemonic='BASE', operand=asm.operand)))
34
35         f.write(self.makeText(self.end))
36         f.close()
37
38
39     def makeText(self, asm):
40         sSpace = ' ' * 4
41         lspace = ' ' * 8
42
43         ni = asm.indirectFlag and '@' or asm.immediateFlag and '#' or ''
44         e = asm.extendedFlag and '+' or ''
45         x = asm.indexAddrFlag and ',X' or ''
46
47         loc      = asm.loc      != None and '%04X' % asm.loc      or sSpace
48         label    = asm.label    != None and '%-8s' % asm.label    or lspace
49         mnemonic = asm.mnemonic != None and '%-8s' % (e + asm.mnemonic) or lspace
50         operand  = asm.operand  != None and '%-8s' % (ni + str(asm.operand) + x) or lspace
51         objCode  = asm.objCode != None and '%-8s' % asm.objCode or lspace
52
53         return '{}\t{}\t{}\t{}\n'.format(loc, label, mnemonic, operand, objCode)
```

AssemblyDict의 인스턴스 생성시 받은 header, end 정보를 포함하여 assembly 파일을 만든다.

앞서 AssemblyDict의 인스턴스에 key, value를 추가할 때, key를 순서대로 넣지 않았으므로

list로 변환하여 sorting을 하는 과정이 있으며,

내부적으로 makeText 메서드를 사용하여 가독성있는 assembly 파일을 만들 수 있게 하였다.

실행 결과 (assembly file)

Loc		Source statement	Object code	Line	Loc	Source statement	Object code
0000	COPY	START 0		5	0000	COPY START 0	
0000	FIRST	STL RETADR	17202D	10	0000	FIRST STL RETADR	17202D
0003		LDB #LENGTH	69202D	12	0003	LDB #LENGTH	69202D
		BASE LENGTH		13		BASE LENGTH	
0006	CLOOP	+JSUB RDREC	4B101036	15	0006	CLOOP +JSUB RDREC	4B101036
000A		LDA LENGTH	032026	20	000A	LDA LENGTH	032026
000D		COMP #0	290000	25	000D	COMP #0	290000
0010		JEQ ENDFIL	332007	30	0010	JEQ ENDFIL	332007
0013		+JSUB WRREC	4B10105D	35	0013	+JSUB WRREC	4B10105D
0017		J CLOOP	3F2FEC	40	0017	J CLOOP	3F2FEC
001A	ENDFIL	LDA EOF	032010	45	001A	ENDFIL LDA EOF	032010
001D		STA BUFFER	0F2016	50	001D	STA BUFFER	0F2016
0020		LDA #3	010003	55	0020	LDA #3	010003
0023		STA LENGTH	0F200D	60	0023	STA LENGTH	0F200D
0026		+JSUB WRREC	4B10105D	65	0026	+JSUB WRREC	4B10105D
002A		J @RETADR	3E2003	70	002A	J @RETADR	3E2003
002D	EOF	BYTE 3	454F46	80	002D	EOF BYTE C'EOF'	454F46
0030	RETADR	RESW 1		95	0030	RETADR RESW 1	
0033	LENGTH	RESW 1		100	0033	LENGTH RESW 1	
0036	BUFFER	RESB 4096		105	0036	BUFFER RESB 4096	
1036	RDREC	CLEAR X	B410	110		.	SUBROUTINE TO READ RECORD INTO BUFFER
1038		CLEAR A	B400	125	1036	RDREC CLEAR X	B410
103A		CLEAR S	B440	130	1038	CLEAR A	B400
103C		+LDT #4096	75101000	132	103A	CLEAR S	B440
1040	RLOOP	TD INPUT	E32019	133	103C	+LDT #4096	75101000
1043		JEQ RLOOP	332FFA	135	1040	RLOOP TD INPUT	E32019
1046		RD INPUT	DB2013	140	1043	JEQ RLOOP	332FFA
1049		COMPR A,S	A004	145	1046	RD INPUT	DB2013
104B		JEQ EXIT	332008	150	1049	COMPR A,S	A004
104E		STCH BUFFER,X	57C003	155	104B	JEQ EXIT	332008
1051		TIXR T	B850	160	104E	STCH BUFFER,X	57C003
1053		JLT RLOOP	3B2FEA	165	1051	TIXR T	B850
1056	EXIT	STX LENGTH	134000	170	1053	JLT RLOOP	3B2FEA
1059		RSUB	4F0000	175	1056	STX LENGTH	134000
105C	INPUT	BYTE 1	F1	180	1059	RSUB	4F0000
105D	WRREC	CLEAR X	B410	185	105C	INPUT BYTE X'F1'	F1
105F		LDT LENGTH	774000	195		.	SUBROUTINE TO WRITE RECORD FROM BUFFER
1062	WLOOP	TD OUTPUT	E32011	200		.	
1065		JEQ WLOOP	332FFA	205		.	
1068		LDCH BUFFER,X	53C003	210	105D	WRREC CLEAR X	B410
106B		WD OUTPUT	DF2008	212	105F	LDT LENGTH	774000
106E		TIXR T	B850	215	1062	WLOOP TD OUTPUT	E32011
1070		JLT WLOOP	3B2FEF	220	1065	JEQ WLOOP	332FFA
1073		RSUB	4F0000	225	1068	LDCH BUFFER,X	53C003
1076	OUTPUT	BYTE 1	05	230	106B	WD OUTPUT	DF2008
		END	FIRST	235	106E	TIXR T	B850
				240	1070	JLT WLOOP	3B2FEF
				245	1073	RSUB	4F0000
				250	1076	OUTPUT BYTE X'05'	05
				255		END FIRST	

Loc 002D 등의 BYTE operand 및 주석 외에는 동일한 assembly 파일이 만들어진 것을 확인할 수 있다.
 BYTE operand의 경우 Disassembly에서 X type인지 C type인지 구분할 수 있는 명확한 정보가 있는게 아니라 추가 처리하지 않았다.