◈ 技術解說

M6800 Cross-Assembler

鄭 起 東*

—차 례—

Ⅰ. 序 論

Microprocessor의 出現은 電子計算機 구조에 일대 혁명을 일으켰으며 最近 몇년간 microcomputer가 世界的인 화제의 초점을 이루고 있고 그 應用은 극히 强力한 기대와 관심을 불러 일으키고 있다.

Microcomputer의 가장 큰 특징은 micrs computer 自體가 한 system의 component로 使用될 수 있다는 점이며, application software의 開發여하에 따라 無限한 可能性을 약속해 주고 있다.

本 論文에서는 M6800 application software를 開發하기 위하여 M6800 crossassembler를 作成하였다.

M6800은 Motorola에서 開發한 micrs computer로 全世界的으로 가장 널리 使用되고 있는 것 中의 하나이다.

Sourco program은 M6800 assembly language이고 이 program을 assembly 하기 위하여 CYBER FOR TRAN이 使用되며, object program은 결국 M6800 machine language program이 된다.

CYBER system은 대형 고속 computer이고 I/O 속도가 빠르므로 debugging시간을 절약할 수 있고 somce machine의 back-up machine 역할을 할 수도 있다.

이 같은 language간의 translation은 1962연 J.H. Gunn이 처음 시도 했으며 그는 Mercury computer와 Orion computier간의 machine language translation을 시도 하였다.

本 論文의 2장의 첫 부분에서는 M6800 assembly language가 소개되고 두번째 부분에서는 cross-assembler의 기본적인 特性이 기술되고 있다.

* 正會員 : 서울大 大學院

Ⅱ. 本 論

1. M6800 assemble language

M6800 assembler language의 하나의 source statement는 4個의 group으로 形成된다.

4個의 group은 다음과 같다.

(1) label
(2) operator
(3) operand
(4) comment

이들 4個의 group은 하나의 space로써 서로 구분된다.

1) label

label은 1~6個의 alphanumeric character로 形成되며 첫째 문자는 반드시 alphabet여야 한다. 또 하나의 character A.B.Y는 label로 사용될 수 없다.

2) operator

M6800 assembler langage는 72개의 mnemonic operator와 7개의 directive로 形成된다.

mnemonic operator는 對應하는 machine code가 있지만 driective는 machire wde가 生成되지 않는다.

M6800 assembly에서는 immediate, indexed, extended와 inherent의 4가지의 addressing mode가 있다. addressing mode는 operater와 operand를 조사하여 assembler가 決定한다.

表 1은 operator의 각 addressing mode에 對한 Hexadecinal machine code와 MPU cycle의 表, program byte수를 나타내고 있다.

ADDRESSING MODES

OPERATIONS	MNE MONIC	IMMED			DIRECT			INDEX			EXTND			INHER			BOOLEAN/ ARITHMETIC OPERATION
		OP	—	#	OP	—	#	OP	—	#	OP	—	#	OP	—	#	
Add	ADDA	8B	2	2	9B	3	2	AB	5	2	BB	4	3				A+M→A
	ADDB	CB	2	2	DB	3	2	EB	5	2	FB	4	3				B+M→B
Add Acmltrs	ABA													1B	2	1	A+B→A
Add with Carry	ADCA	89	2	2	99	3	2	A9	5	2	B9	4	3				A+M+C→A
	ADCB	C9	2	2	D9	3	2	E9	5	2	F9	4	3				B+M+C→B
And	ANDA	84	2	2	94	3	2	A4	5	2	B4	4	3				A.M→A
	ANDB	C4	2	2	D4	3	2	E4	5	2	F4	4	3				B.M→B
Bit Test	BITA	85	2	2	95	3	2	A5	5	2	B5	4	3				A.M
	BITB	C5	2	2	D5	3	2	E5	5	2	F5	4	3				B.M
Clear	CLR							6F	7	2	7F	6	3				00→M
	CLRA										4F	2	1	4F	2	1	00→A
	CLRB													5F	2	1	00→B
Compare	CMPA	81	2	2	91	3	2	A1	5	2	B1	4	3				AM
	CMPB	C1	2	3	D1	3	2	E1	5	2	F1	4	3				BM
Compare Acmtrs	CBA													11	2	1	A−B
Complement, 1's	COM							63	7	2	73	6	3				M→M
	COMA													43	2	1	A→A
	COMB													53	2	1	B→B
Complement, 2's	NEG							60	7	2	70	6	3				00−M→M
(Negate)	NEGA													40	2	1	00−A→A
	NEGB													50	2	1	00−B→B
Decimal Adjust. A	DAA													19	2	1	Converts Binary Add of BCD Characters into BCD format
Decrement	DEC							6A	7	2	7A	6	3				M−1→M
	DECA													4A	2	1	A−1→A
	DECB													5A	2	1	B−1→B
Exclusive OR	EORA	88	2	2	98	3	2	A8	5	2	B8	4	3				AM→A
	EORB	C8	2	2	D8	3	2	E8	5	2	F8	4	3				BM→B
Increment	INC							6C	7	2	7C	6	3				M+1→M
	INCA													4C	2	1	A+1→A
	INCB													5C	2	1	B+1→B
Load Acmltr	LDAA	86	2	2	96	3	2	A6	5	2	B6	4	3				M→A
	LDAB	C6	2	2	D6	3	2	E6	5	2	F6	4	3				M→B
Or, Inclusive	ORAA	8A	2	2	9A	3	2	AA	5	2	BA	4	3				A+M→A
	ORAB	CA	2	2	DA	3	2	EA	5	2	FA	4	3				B+M→B
Push Data	PSHA													36	4	1	A→MSP, SP−1→SP
	PSHB													37	4	1	B→MSP, SP−1→SP
Pull Data	PULA													32	4	1	SP+1→SP, MSP→A
	PULB													33	4	1	SP+1→SP, MSP→B
Rotate Left	ROL							69	7	2	79	6	3				
	ROLA													49	2	1	
	ROLB													59	2	1	

OPERATIONS	MNE MONIC	IMMED			DIRECT			INDEX			EXTND			INHER			BOOLEAN/ ARITHMETIC OPERATION
		OP	—	#	OP	—	#	OP	—	#	OP	—	#	OP	—	#	
Rotate Right	ROR							66	7	2	76	6	3				
	RORA													46	2	1	
	RORB													56	2	1	
Shift left, Arithmetic	ASL							68	7	2	78	6	3				
	ASLA													48	2	1	
	ASLB													58	2	1	
Shift Right, Arithmetic	ASR							67	7	2	77	6	3				
	ASRA													47	2	1	
	ASRB													57	2	1	
Shift Right. Logic	LSR							64	7	2	74	6	3				
	LSRA													44	2	1	
	LSRB													54	2	1	
Store Acmltr	STAA				97	4	2	A7	6	2	B7	5	3				A→M
	STAB				D7	4	2	E7	6	2	F7	5	3				B→M
Subtract	SUBA	80	2	2	90	3	2	A0	5	2	B0	4	3				A−M→A
	SUBB	C0	2	2	D0	3	2	E0	5	2	F0	4	3				B−M−B
Subract Acmltrs	SBA													10	2	1	A−B−A
Subtr With Carry	SBCA	82	2	2	92	3	2	A2	5	2	B2	4	3	3			A−M−C−A
	SBCB	C2	2	2	D2	3	2	E2	5	2	F2	4	3				B−M−C−B
Transfer Acmltrs	TAB													16	2	1	A−B
	TBA													17	2	1	B−A
Test, Zero or M Minus	TST							6D	7	2	7D	6	3				M−00
	TSTA													4D	2	1	A−00
	TSTB													5D	2	1	B−00

INDEX REGISTER AND STACK POINTER OPERATIONS	MNE MONIC	IMMED			DIRECT			INDEX			EXTND			IMHER			BOOLEAN/ ARITHMETIC OPERATION
		OP	—	#	OP	—	#	OP	—	#	OP	—	#	OP	—	#	
Compare index reg	CPX	8C	3	3	9C	4	2	AC	6	2	BC	5	3				
Decrement index reg	DEX													09	4	1	
Decrement Stack pntr	DEX													34	4	1	
Increment Index reg	INX													08	4	1	
Increment Stack Pntr	INS													31	4	1	
Load Index reg	LDX	CE	3	3	DE	4	2	EE	6	2	FE	5	3				
Load Stack Pntr	LDS	8E	3	3	9E	4	2	AE	6	2	BE	5	3				
Store Index Reg	STX				DF	5	2	EF	7	2	FF	6	3				
Store Stack Pntr	STS				9F	5	2	AF	7	2	BF	6	3				
Indx Reg-Stack Pntr	TXS													35	4	1	
Stack Pntr-Indx reg	TSX													30	4	1	

JUMP AND BRANCH OPERATIONS	MNE MONIC	RELATIVE			INDEX			EXTND			INHER			BRANCH TEST
		OP	—	#	OP	—	#	OP	—	#	OP	—	#	
Branch Always	BRA	20	4	2										None
Branch Carry Clear	BCC	24	4	2										C=0
Branch Carry Set	BCS	25	4	2										C=1
Branch if=Zero	BEQ	27	4	2										Z=1
Branch if Zero	BGE	2C	4	2										NoV=0
Branch if Zero	BGT	2E	4	2										Z+(NoV)=0
Branch if Higher	BHI	22	4	2										C+2=0
Branch if Zero	BLE	2F	4	2										Z+(NoV)=1
Branch if lower or Same	BLS	23	4	2										C+Z=1
Branch if Zero	BLT	2D	4	2										NoV=1
Branch if Minus	BMI	2B	4	2										N=1
Branch if not Equal Zero	BNE	26	4	2										Z=0
Branch if Overflow Clear	BVC	28	4	2										V=0
Branch if Overflow Set	BVS	29	4	2										V=1
Branch Plus	BPL	2A	4	2										N=0
Branch to Subrounine	BSR	8D	8	2										
Jump	JMP				6E	4	2	7E	3	3				
Jump to Subroutine	JSR				AD	8	2	BD	9	3				
No Operation	NOP										01	2	1	
Noturn from interrupt	RTI										38	10	1	
Return from Subroutine	RTS										39	5	1	
Software interrupt	SWI										3F	12	1	
Wait for interrupt	WAI										3E	9	1	

CONDITIONS CODE REGISTER OPERATIONS	MNEMONIC	INHER			BOOLEAN OPERATION
		OP	—	#	
Clear Carry	CIC	OC	2	1	O—C
Clear interrupt mask	CLI	OE	2	1	O—1
Clear overflow	CLV	OA	2	1	O—V
Set Carry	SEC	OD	2	1	1—C
Set interrupt mask	SEI	OF	2	1	1—1
Set Overflow	SEV	OB	2	1	1—V
Acmltr A-CCR	TAP	06	2	1	A—CCR
CCR-Acmltr A	TPA	07	2	1	CCR—A

表 1. Instruction Set

OP : operation code (Hexadecimal)

~ : Number of MPU cycles

: Number of program bytes

3) operand

operand의 syntax는 다음과 같다.

<operand> ::= <immedia te>|<indexed>|
<extended>|<inherent>|<directive-op1>|
<directive-op2>

<immediate> ::= #<expression>|
#'<alphanumeric>

<extended> ::= <expression>

<indexed> ::= <expression> <comma> X

<directive-op1> ::= <expression>|<directive-op1>
<comma> <expression>

<directive-op2> ::= <symbol 1>

<comma> ::= ,

<expression> ::= <term>|<expression> <a.o>

<term>
<term> ::= <symbol>|<number>|*
<symbol> ::= <alphabet>|<symbol>
<alphanumeric>
<symbol 1> ::= <alphanumeric>|<symbol 1>
<alphanumeric>
<alphanumeric> ::= <alphabet>|<digit>
<alphabet> ::= A|B|……|Z
<digit> ::= 0|1|2|……|9
<number> ::= <num>|<num> <post-op>|<prefix-op> <num>
<num> ::= <digit> <num> <digit>
<post-op> ::= H|O|Q|B
<prefix-op> ::= @|$|%
<a.o> ::= +|−|*|/
<inherent> ::=

또 使用되는 directive는 다음과 같다.

① ORG : source program을 assembly했을 때 machine code의 첫 byte의 numerical address를 定하여 준다.

② EQU : 한 symbol에 numerical value를 주기 위하여 使用된다.

③ FCB(Form Corstant Byte) : comma로 구분되는 operand의 값들이 각각 한個의 byte에 주어진다.

④ FCC(Form Constant Characters) : slask(1) 사이에 있는 character의 ASCⅡ Hexadecimal code가 character 表 만큼의 byte에 주어진다.

⑤ FDB(Form Double Constant Byte) : Comma로 구분되는 operand의 값들이 각각 2개의 byte에 들어간다.

⑥ RMB(Reserve Memory Byte) : operond에 표시된 값만큼 location counter의 값을 증가 시킨다. 즉 operand의 값만큼의 memory block을 잡는 역할을 한다.

⑦ END : Source program의 끝을 나타낸다.

2. Cross assembler

Cross assembler는 pass 1, pass 2의 2 pass이다.

pass 1에서는 scanning을 하고 syntax checking을 한다. 또 operator와 operand를 종합하여 addressing mode를 決定하고 2 mode에 해당하는 op code를 生成한다.

pass 2에서는 operand field의 값을 計算하여 machine code의 address part를 決定한다.

또 assembly가 끝나면 symbol table을 print하고 error가 있으면 error message를 print한다.

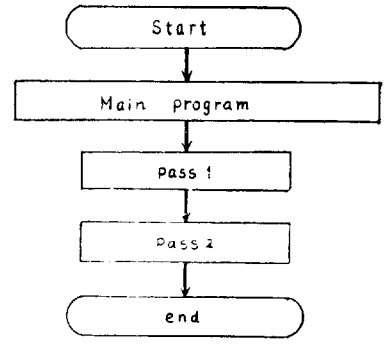

그림 1

assembler의 flow diagrom은 그림 1과 같다.

1) Main program

Main program은 assembly에 필요한 각종 table을 잡고, source program을 읽어 들이고 pass 1을 불러서 assembley 한다.

2) pass-1

pass-1은 scanner와 syntax analyzer 2 부분으로 되어 있다.

Scanner와 analyzer와의 關係는 그림 2와 같다.

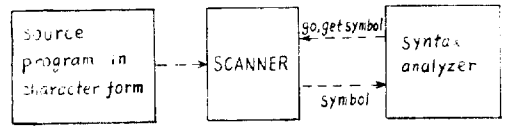

그림 2

pass-1의 全體的인 flow-diagram은 그림 3과 같다.

a) Scanner

Scanner는 source program을 constituent part로 나눈다.

또 label을 symbol table에 저장하고 operator의 character가 A～Z 사이의 문자인지 조사하고 operator의 addressing mode를 決定한다.

Operand를 scanning하고 다음 token을 使用한다.

*(program counter)	1
#	2
×	3
arithmetic operator (+, −, *, /)	4
,(comma)	8
‘(apostrophe)	9
symbol	10
constant	11
blank	14
terminator	15

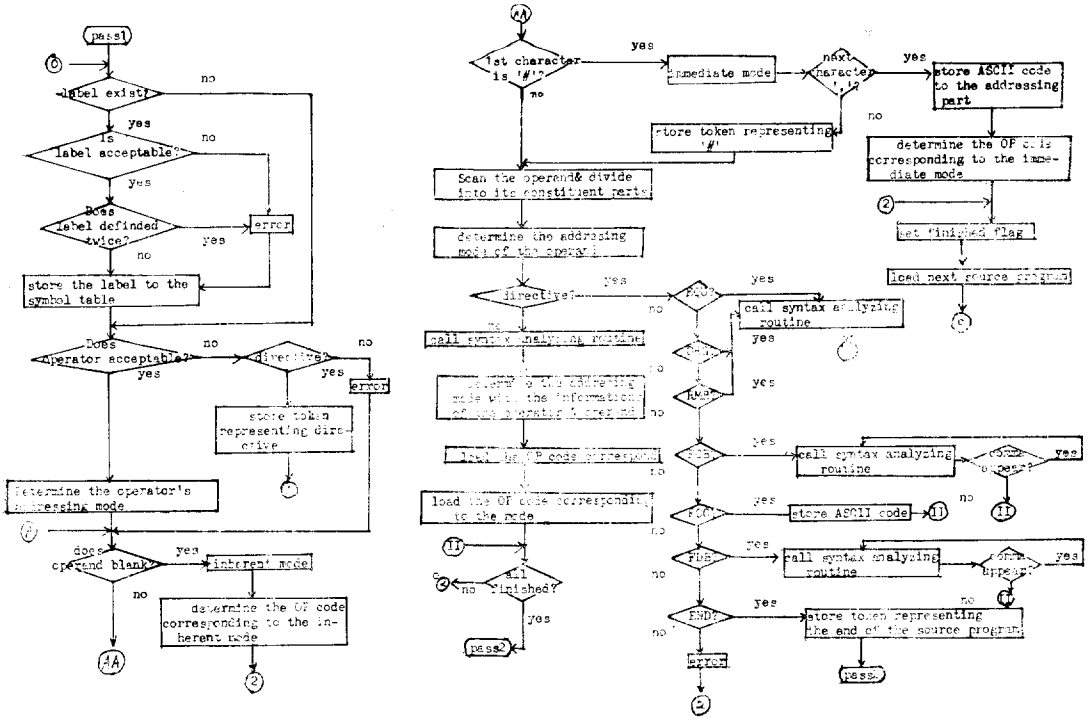

그림 3

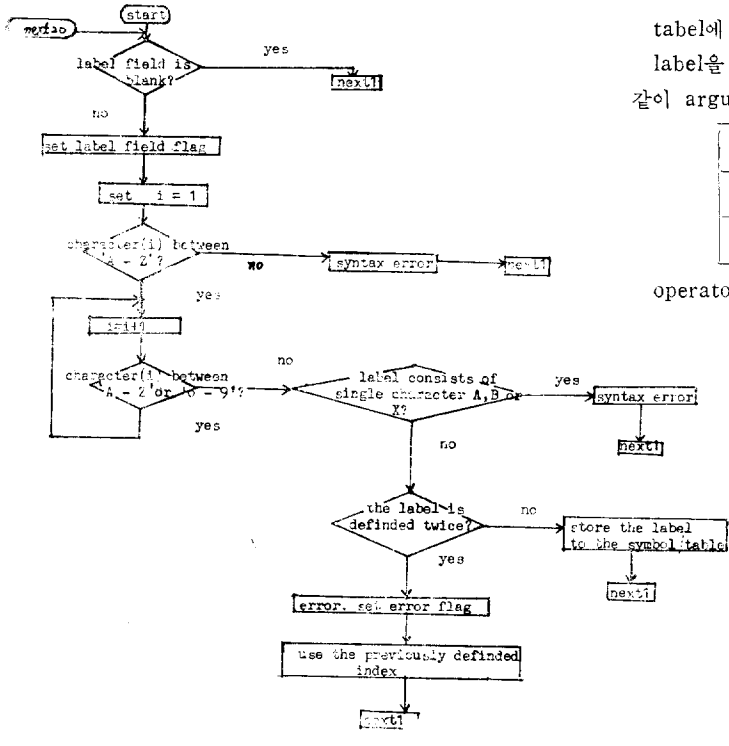

그림 4

tabel에 對한 scanning을 그림 4와 같다.

label을 저장하기 위한 symbol table은 아래 그림과 같이 argument와 volue의 2個 부분으로 되어 있다.

argument	value

operator에 對한 scanning은 그림 5와 같다.

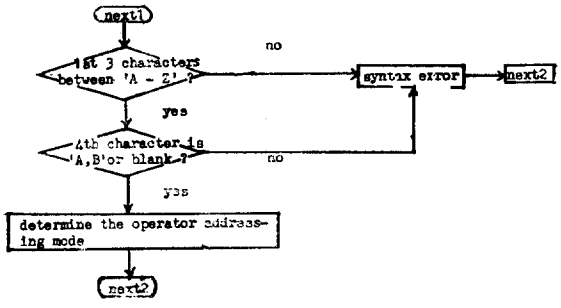

그림 5

operator의 mode를 결정하기 위하여 다음과 같은 token이 使用된다.

immediate, indexed, extended 3가지 다 가능하면 3, indexed와 extended만 가능하면 2, inherent 하나만 되면 1, relatrive이면 0, directive이면 4가 使用이다.

operand에 對한 scanning은 그림 7 flow-diagram과 같다.

operator와 operand를 使用하여 addressing mode를 決定하고 해당 mode에 필요한 만큼의 byte 길이를 결정하여 program counter의 값을 증가시킨다.

operand中 使用되는 constant의 형은 다음과 같다.

number(decimal), $ number (heradecimal) number H(Hexadecimal), @ number (Octal), number O(Octal), number Q(Octal), % number (binary), number B (Binary) constant의 크기는 0부터 65535까지이며 음수는 2's complement로 고쳐서 저장 된다.

constant table은 아래와 같다.

value

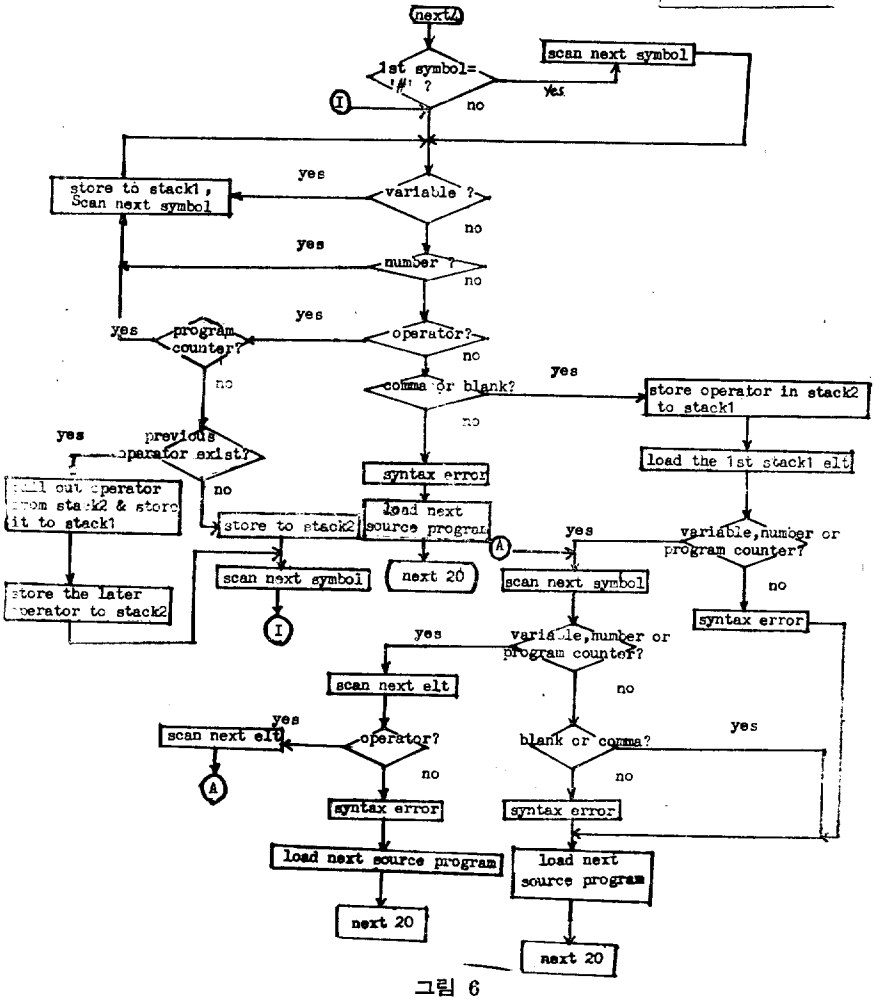

그림 6

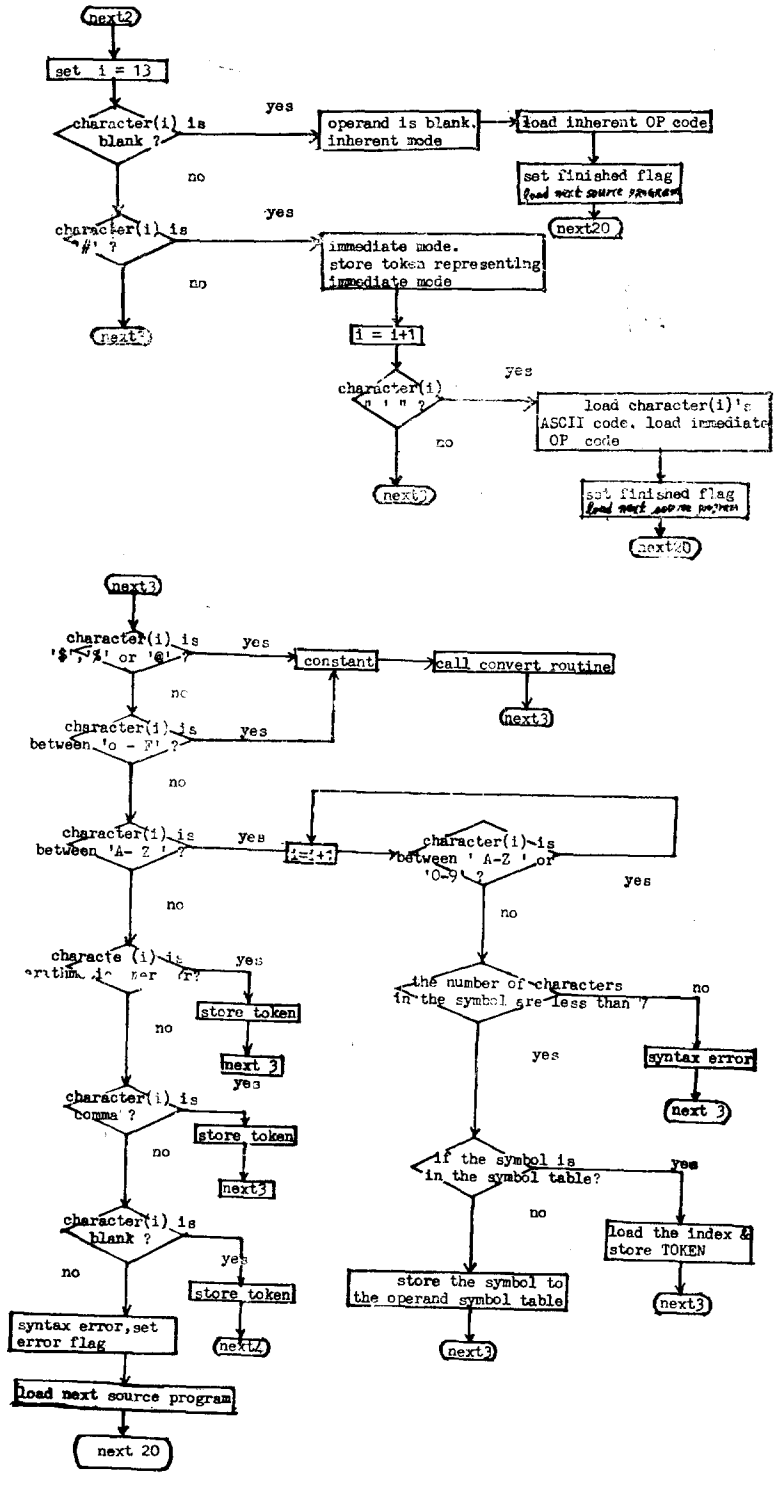
next2
set i = 13
character(i) is blank ?
yes
operand is blank. inherent mode
load inherent OP code
set finished flag
next20
no
character(i) is '#' ?
yes
immediate mode. store token representing immediate mode
no
next3
i = i+1
character(i) ' ' ?
yes
load character(i)'s ASCII code, load immediate OP code
no
next3
set finished flag
next20
next3
character(i) is '$', '%' or '@' ?
yes
constant
call convert routine
next3
no
character(i) is between '0 - F' ?
yes
no
character(i) is between 'A- Z ' ?
yes
i=i+1
character(i) is between ' A-Z ' or '0-9' ?
yes
no
no
store token
next 3
yes
character(i) is comma ?
store token
no
next3
character(i) is blank ?
yes
store token
no
syntax error, set error flag
next2
load next source program
next 20
the number of characters in the symbol are less than 7
no
yes
syntax error
next 3
if the symbol is in the symbol table?
yes
no
load the index & store TOKEN
next3
store the symbol to the operand symbol table
next3

그림 7

Operand 中에 나타나는 variable은 다음과 같이 한 개의 word로 된 table에 저장하여 놓고 pass 2에서 사용한다.

operand의 symtol table은 아래와 같다.

argument

Operand의 syntax를 check하기 위하여 top-down parsing이 使用되고 operand中에 나타나는 arithmetic expression은 post-fix palish notation으로 바꿔어 진다. 이에 對한 flow-diagram은 그림 6과 같다.

Directive에 對한 syntax check는 ORG, EQU, RMB, END는 위와 같고 FCB, FCC, FBB는 다르다.

또 위와 같이 syntax를 check하는 中에 numerical value로만된 operand가 있으면 이 값들을 계산하여 저장하고 그 結果를 pass 2에서 利用한다.

이상으로 pass 1이 끝난다.

(3) pass-2

pass 2에서는 machine code의 addressing part를 決定한다.

特히 Branch instrnation들의 branch point를 정하여야 한다.

pass 2의 flow-diagram은 그림 8과 같다.

Branching address는 다음 式에 의하여 計算된다.

$$R=D-(PC+2)$$

PC : 현재 instruction의 program counter의 값

D : branching해야 할 instruction의 처음 byte의 address

모든 instruction에 對한 addressing part이 計算이 끝나면 assembler는 symbol table을 print하고 error가 있으면 error가 發生한 source program의 line

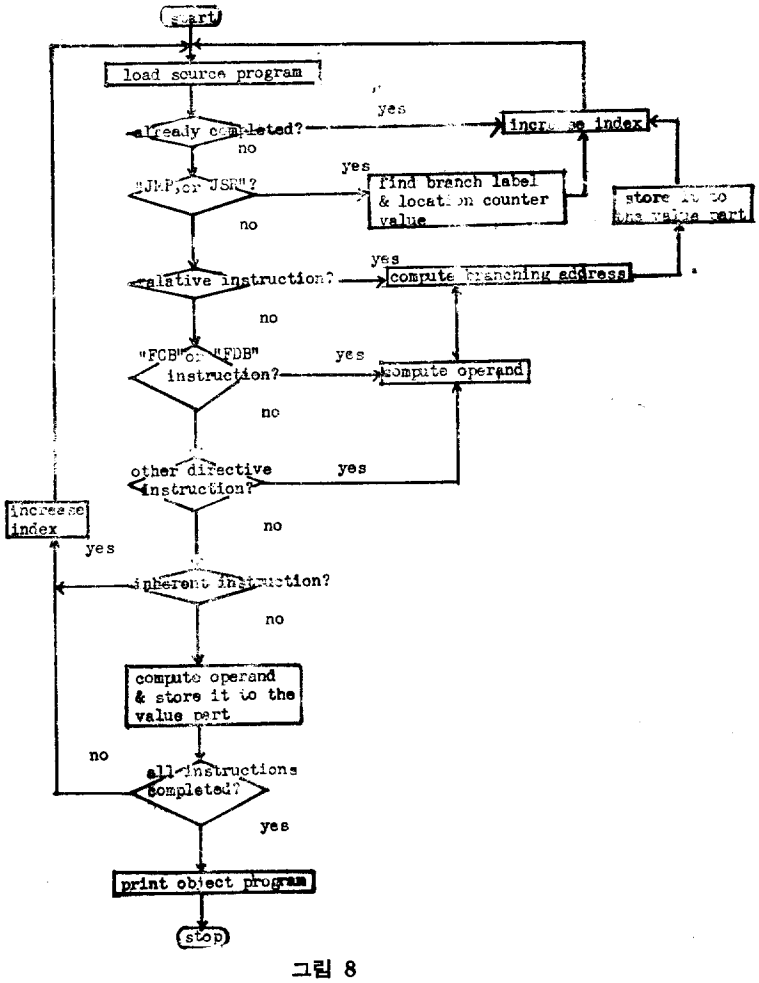

그림 8

```
10          0008        NB      EQU     8
20  1000                        ORG     $1000
30  1000    C6  08      BCD     LDAB    #NB
40  1002    FE  0100            LDX     ADDR
50  1005    0C                  CLC
60  1006    A6  07      NEXT    LDAA    NB-1,X
70  1008    A9  0F              ADCA    2*NB-1,X
80  100A    19                  DAA
90  100B    A7  17              STAA    3*NB-1,X
100 100D    09                  DEX
110 100E    5A                  DECB
120 100F    26  F5              BNE     NEXT
130 1011    39                  RTS
140 1100                        ORG
150 1100    8E  013F            LDS     #$13F
160 1103    CE  0102            LDX     #P
170 1106    FF  0100            STX     ADDR
180 1109    BD  1000            JSR     BCD
190 110C    01                  NOP
200 110D    20  FE              BRA     *
210 0100                        ORG     $0100
220 0100    0002        ADDR    RMB     2
230 0102    0008        P       RMB     NB
240 010A    0008        Q       RMB     NB
250 0112    0008        RES     RMB     NB
260                             END
SYMBOL TABLE
ADDR  0100  BCD  1000  NB    0008  NEXT  1006
P     0102  Q    010A  RES   0112
```

그림 9

number와 error message를 print 한다.

結果를 print 할 때는 line number를 제외한 모든 값들을 Hexadecimal 고쳐서 print 한다.

다음은 실제의 M6800 assembly pronam과 그 output을 보여주고 있다.

Ⅲ. 結 論

M6800 cross-assembler의 가장 큰 장점은 대형, 고속 system을 使用하므로 debugging 시간을 절약할 수 있다는 것이다. execution time과 memory는 앞으로의 硏究에 依하여 더욱 절약할 수 있다.

또한 backing machine으로서의 역할을 효율적으로 利用한다면 훌륭한 soft wore를 開發할 수 있을 것이다.

이 M6800 assembly를 利用하여 FORTRAN, COBCL, PL/1 등과 같은 high-level language를 開發할 수 있으며 論子計算機의 國産化라는 과제를 생각해 볼 때 이 방면으로의 硏究가 더욱 활발해져야 할 것이다.

참 고 문 헌

1. D. Gries., Compiler construction for digital computer, John Wiley & Sons Inc., 1971.
2. J.J. Donovan., Systems programming, McGraw-Hill Book company, 1972.
3. Adam Osborne., An introduction to microcomputers., 1975.
4. M6800 microprocessor appliation manual.
5. 韓國情報科學會誌, 제 1권 1호, 1974.8.