INTERNATIONAL ATOMIC ENERGY AGENCY
UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION

INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
34100 TRIESTE (ITALY) - P.O.B. 586 - MIRAMARE - STRADA COSTIERA 11 - TELEPHONES: 224281/2/3/4/5/6
CABLE: CENTRATOM - TELEX 460392-1

H4.SMR/159 -16

THIRD COLLEGE ON MICROPROCESSORS:
TECHNOLOGY AND APPLICATIONS IN PHYSICS

7 October - 1 November 1985

ASSEMBLY LANGUAGE PROGRAMMING FOR THE MOTOROLA MC 6809

N. QUAYNOR

Digital Equipment Corporation
U.S.A.

ASSEMBLER LANGUAGE PROGRAMMING
FOR THE
MOTOROLA MC 6809
BY

Nii QUAYNOR

DIGITAL EQUIPMENT CORPORATION
U.S.A
(GHANA)

Programming in Assembler

- Low level Programming Methods
- Useful Algorithms
- Good Programming Practices
- Assembly Process
- Systems Programming Techniques
- Concepts for other machines

Software View of Comuter

HARDWARE ARCHITECTURE :=

{ FUNCTIONS, (OPCODES)
  STATE, (REGISTER, CONDITIONS)
  MEMORY, (ADDRESSING, ..)
  I/O (DEVICES, MAPPING, START, FINISH)
}

PASCAL

```
Z := V + W;
  :
```

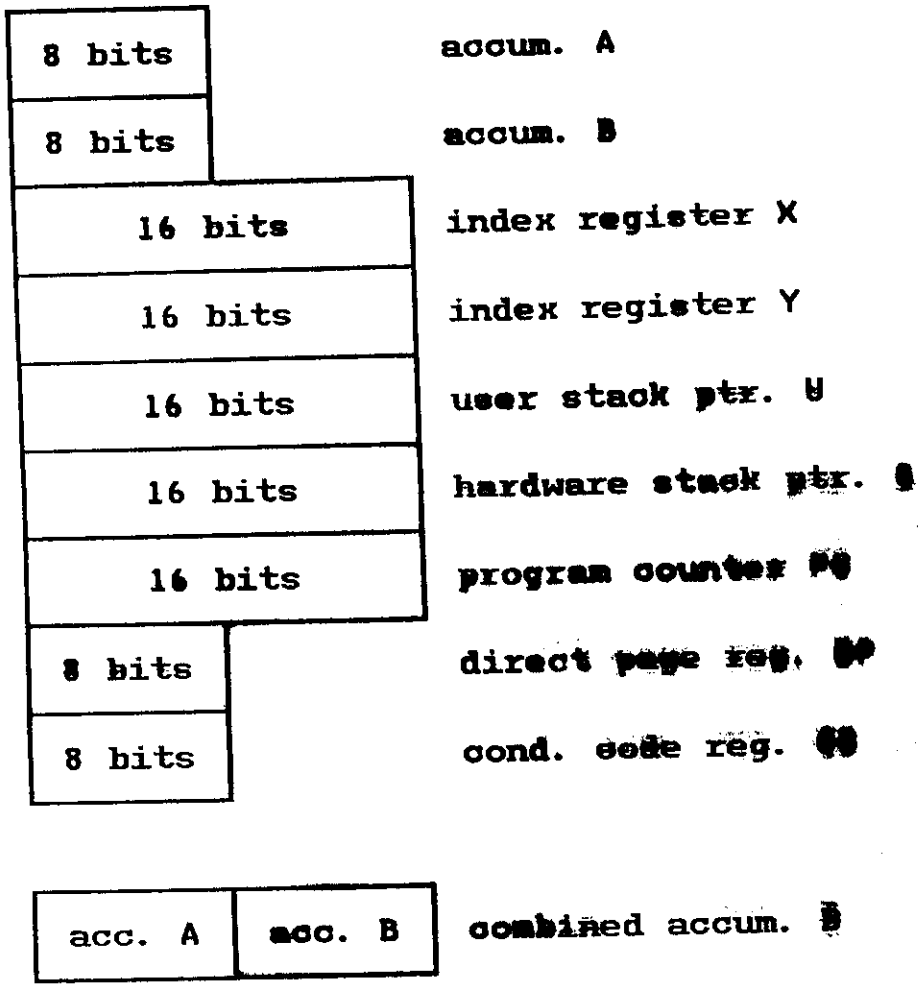
TRANSLATE ⟹

6809

```
  :
LDA   V
ADDA  W
STA   Z
  :
```

- MEMORY - PROGRAM / DATA
- PROGRAM COUNTER
- OPCODES
- REGISTERS
- MEMORY ADDRESSES
- CONDITION CODES

MC 6809 Register Package:



MC 6809 Condition Code Register:

7	6	5	4	3	2	1	0	bit No.
E	F	H	I	N	Z	V	C	flags

	0	1	
C:	no	with	carry or borrow
V:	no	with	magnitude overflow
Z:	not	is	zero
N:	not	is	negative
I:	enable	disable	normal interrupts
H:	no	with	half carry
F:	enable	disable	fast interrupts
E:	part	full	register stacking

First Programming Problem:

Move contents of memory location SOURCE ($0410)
into memory location DESTIN ($0411)

* 8-bit data transfer
* use accumulator A

Memory contents before execution:

Location Address		Symbolic Name
$040F	??	
$0410	6A	SOURCE
$0411	12	DESTIN

Instructions:

```
LDA  SOURCE     *LOAD DATA FROM $0410
STA  DESTIN     *TRANSFER TO $0411
```

Memory contents after execution:

Loc:			
	$040F	??	
	$0410	6A	SOURCE
	$0411	6A	DESTIN

First Programming Problem:

* Memory contains data
* Memory contains program to execute

Lets translate our program:

LDA SOURCE — we need extended addr - $0410

→ B6 04 10 (3 bytes)

STA DESTIN — we need extended addr - $0411

→ B7 04 11 (3 bytes)

Now lets load it into memory:

Address		Symbolic
$03FF	??	GARBAGE
$0400	B6	LDA
$0401	04	SOURCE ADDRESS
$0402	10	
$0403	B7	STA
$0404	04	DESTIN ADDRESS
$0405	11	

Second programming problem:

Add the first 15 integers

Constraints: $SUM = \sum_{i=1}^{15} i$

* program starts at loc. $0400 - START
* result stored into loc. $0420 - RESULT

Possible programming solutions:
(expressed in a Pascal like syntax)

```
VAR
 Count  : INTEGER;  to count repetitions
 EndVal : INTEGER;  to end repetitions
 Sum    : INTEGER;  to calculate the sum
```

--> WHILE <condition> DO <body>

```
Sum:= 0; Count:= 0;
WHILE Count < EndVal DO BEGIN
  Count:= Count + 1;
  Sum  := Sum + Count;
END;
```

--> FOR <iterative condition> DO <body>

```
Sum:= 0;
FOR Count:= 1 TO EndVal DO
  Sum:= Sum + Count;
```


SECOND PROGRAMMING PROBLEM:

→ REPEAT <BODY> UNTIL <CONDITION>

```
SUM:=0; COUNT:= ENDVAL;
REPEAT
   SUM:= SUM + COUNT;
   COUNT:= COUNT - 1;
UNTIL  COUNT = 0;
```

→ LETS PROGRAM FOR 6809

DATA:

SUM	→	LOC.	$0420
ENDVAL	→	LOC.	$0421
COUNT	→	LOC.	$0422

PROGRAM:

```
        CLR   SUM       SUM:=0
        LDA   ENDVAL    COUNT:= ENDVAL
        STA   COUNT
LOOP    LDA   COUNT     SUM:= SUM+ COUNT
        ADDA  SUM
        STA   SUM
        LDA   COUNT     COUNT:= COUNT - 1
        DECA
        STA   COUNT
        BNE   LOOP      UNTIL  COUNT=0;
```

Second programming problem:

--> Now let's translate it: (OPCODE, LABELS)

loc.	contents	opcode	addressing mode	
0400	7F 04 20	CLR	extended	SUM
0403	B6 04 21	LDA	extended	ENDVAL
0406	B7 04 22	STA	extended	COUNT
LOOP 0409	B' 04 22	LDA	extended	COUNT
040C	BB 04 20	ADDA	extended	SUM
040F	B7 04 20	STA	extended	SUM
0412	B6 04 22	LDA	extended	COUNT
0415	4A	DECA	inherent	
0416	B7 04 22	STA	extended	COUNT

* we have to compute the relative distance

  hence: $041B - $0409 --> $0012

* we branch backwards

  hence: - $0012 --> $FFEE

* signed short or long branch ?

15 8	7	6 0	bit No.
xxxx xxxx	x	<dist.>	

!! 0419 26 EE BNE relative
!! 041B ..

A VARIATION OF SECOND PROGRAM:

$$SUM_K = \sum_{i=K+1}^{K+15} i$$

DATA:

K → $0423

PROGRAM:

```
        CLR   SUM
        LDA   ENDVAL
        STA   COUNT
LOOP    LDA   COUNT
        ADDA  K
        ADDA  SUM
        STA   SUM
        LDA   COUNT
        DECA
        STA   COUNT
        BNE   LOOP
```

DESIGN METHODS – FLOW GRAPHS

- MANAGE COMPLEXITY OF DESIGNS
- STRUCTURED APPROACH TO DESIGN
  - GRAPHICAL AIDS FOR DESIGN
  - CONTROL STRUCTURES
    - SINGLE ENTRY
    - SINGLE EXIT
    - INDENT STRUCTURE
- CORRECTNESS

  \+

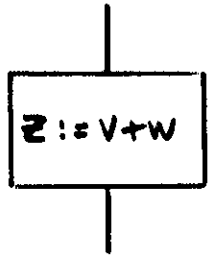
  SOME ALGORITMS

SEQUENTIAL

PASCAL

```
Z := V + W
```

NOTATION



6809

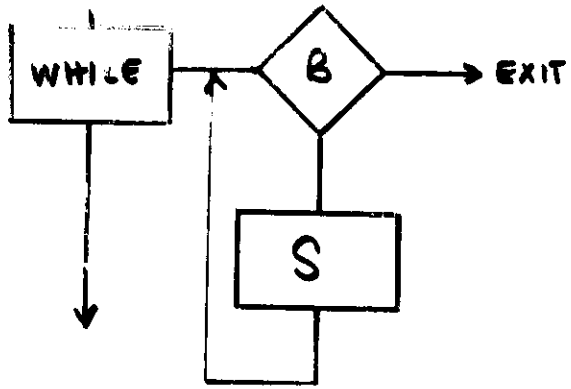
```
LDA   V
ADD   W
STA   Z
```

ITERATION 1

PASCAL

```
WHILE B DO S;
```

NOTATION



6809

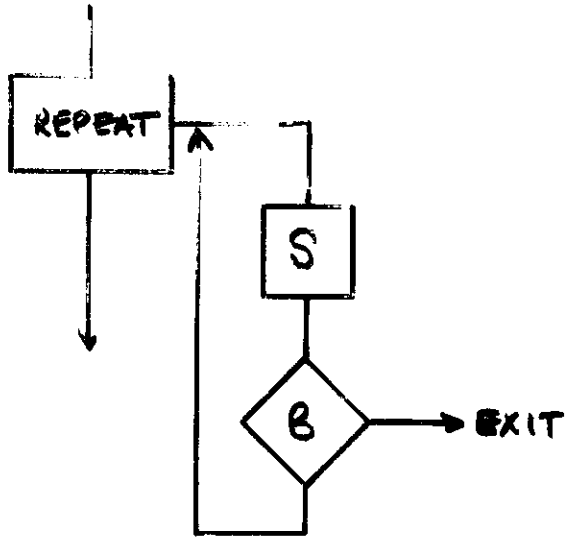
```
WHL   BRANCH ON B̄ TO EXIT
        S
      BRA  WHL
EXIT    ⋮
```


ITERATION 2

PASCAL

```
REPEAT S UNTIL B;
```

NOTATION



6809

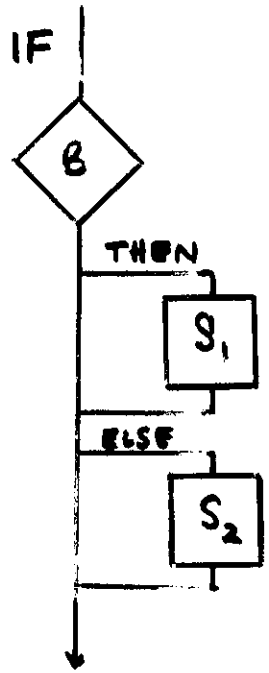
```
REP
      S
      BRANCH ON B̄ TO REP
EXIT
```

ALTERNATIVE

PASCAL

IF B THEN S_1 ELSE S_2

NOTATION



- ALSO CASE i OF $(S_1, S_2, \ldots, S_N)$

N-ALTERNATIVE (CONTD)

6809

≡

BRANCH ON $\overline{B}$ TO ELSE

S_1

BRA EXIT

ELSE

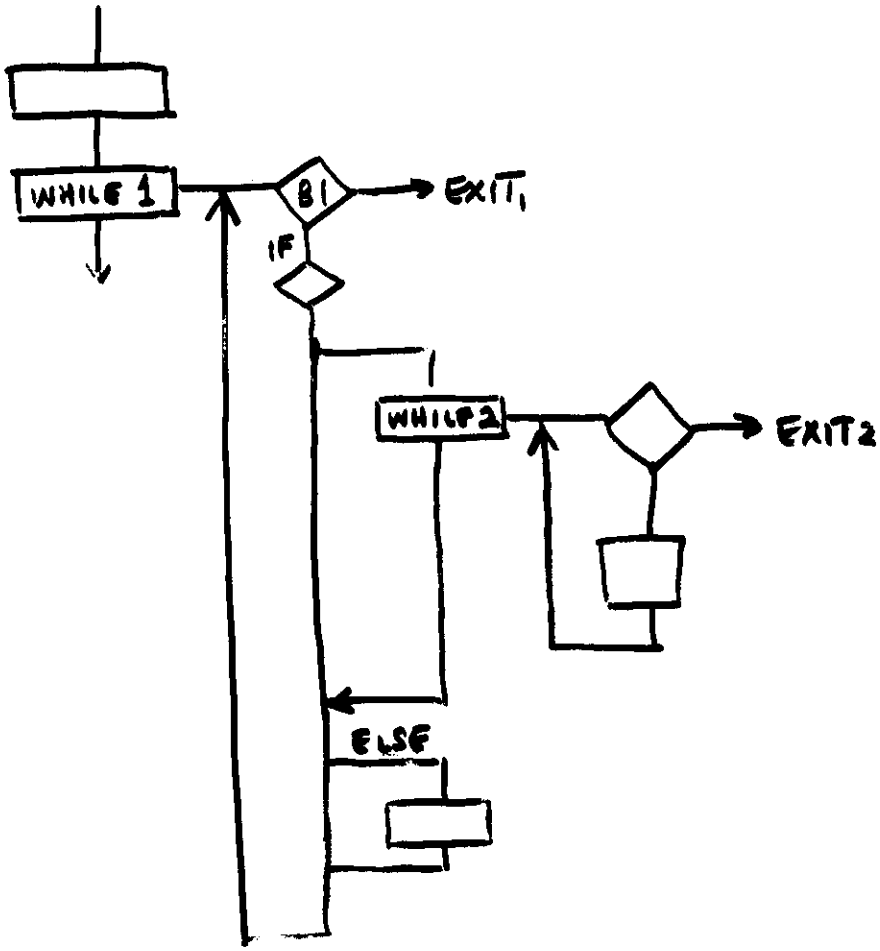
S_2

EXIT

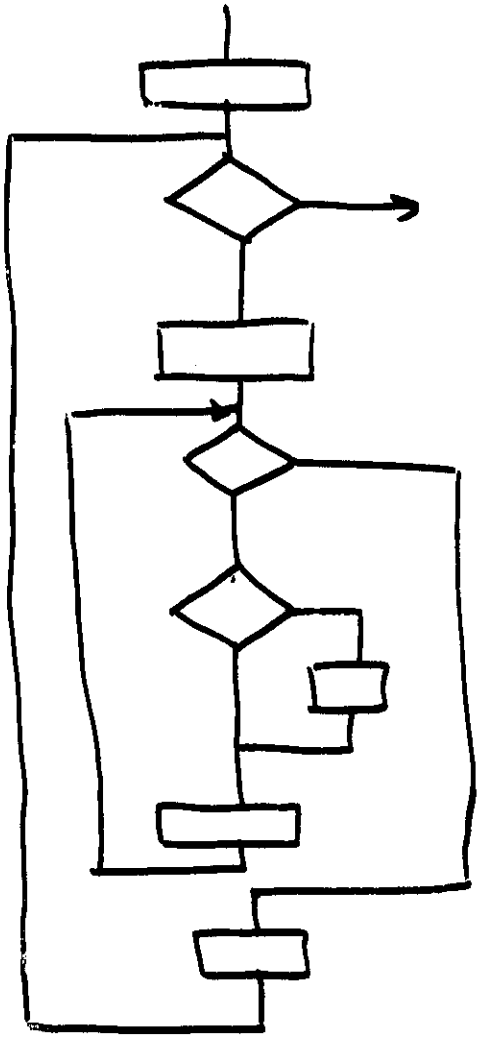
≈

IF B
THEN S_1
ELSE S_2

WELL FORMED FLOW GRAPH



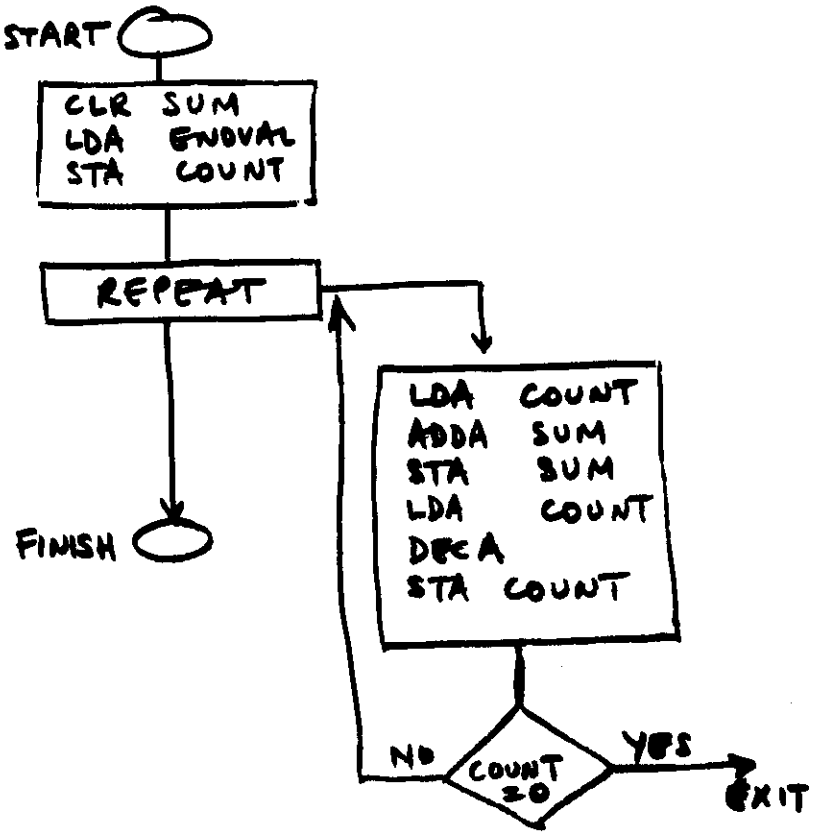
BADLY FORMED FLOW GRAPH



- MULTIPLE EXITS
- LOST STRUCTUR

Flow Diagrams - Examples

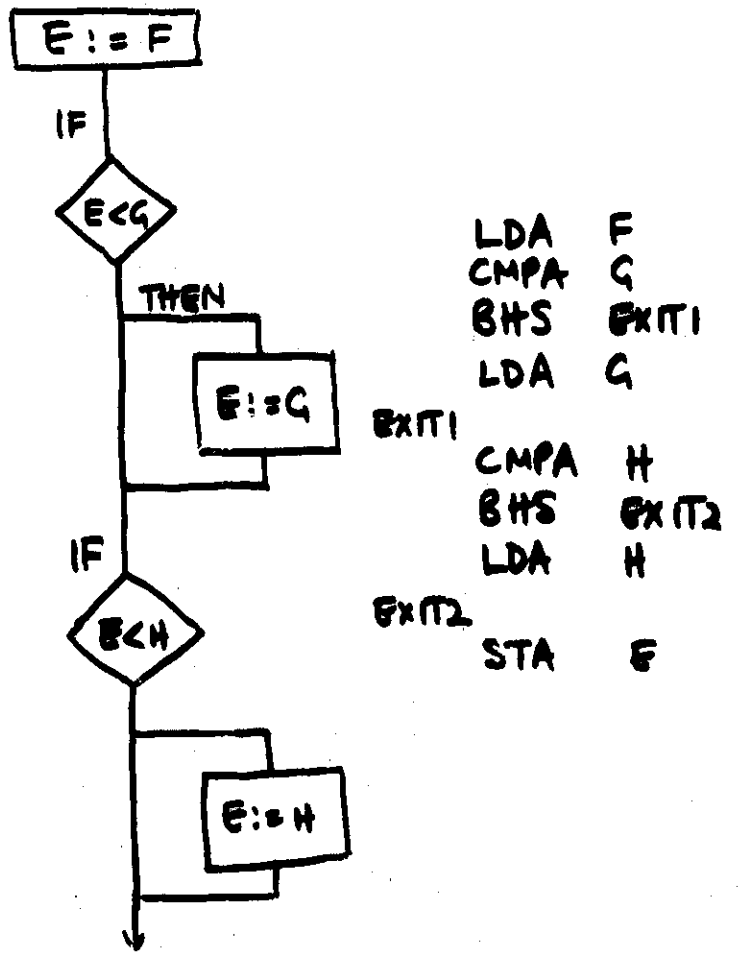
$$SUM = \sum_{i=1}^{15} i$$



Example – IF

$E = MAX(F, G, H)$

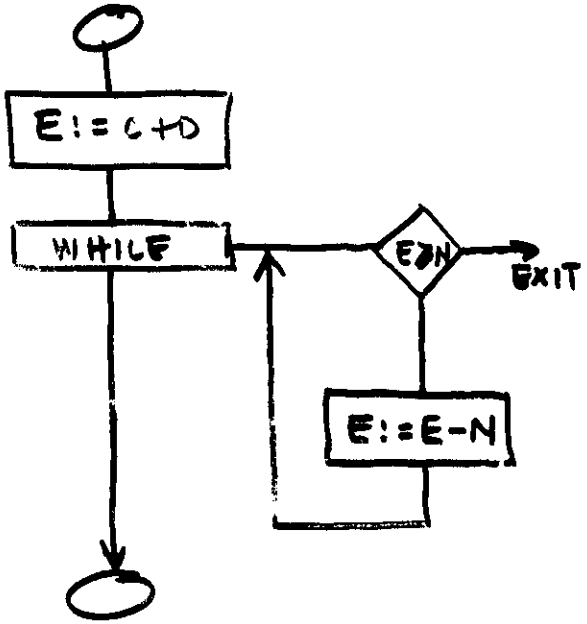
UNSIGNED INTEGERS




```
        LDA   F        IF
        CMPA  G          E<G
        BHS   EXIT1
        LDA   G        THEN
EXIT1                  IF
        CMPA  H          E<H
        BHS   EXIT2
        LDA   H        THEN
EXIT2
        STA   F
```

EXAMPLE — WHILE

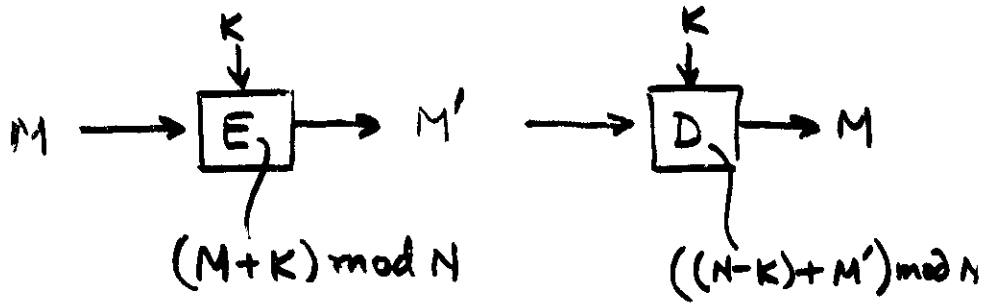
E := (C+D) MOD N [UNSIGNED INT]




```
        LDA   C          E := C+D
        ADDA  D
        STA   E
WHL
        CMPA  N          WHILE B DO
        BLO   EXIT
        SUBA  N          E := E-N
        BRA   WHL        ENDWHILE
EXIT
```

APPLICATIONS OF MODULO ARITH

ENCRYPTION / DECRYPTION

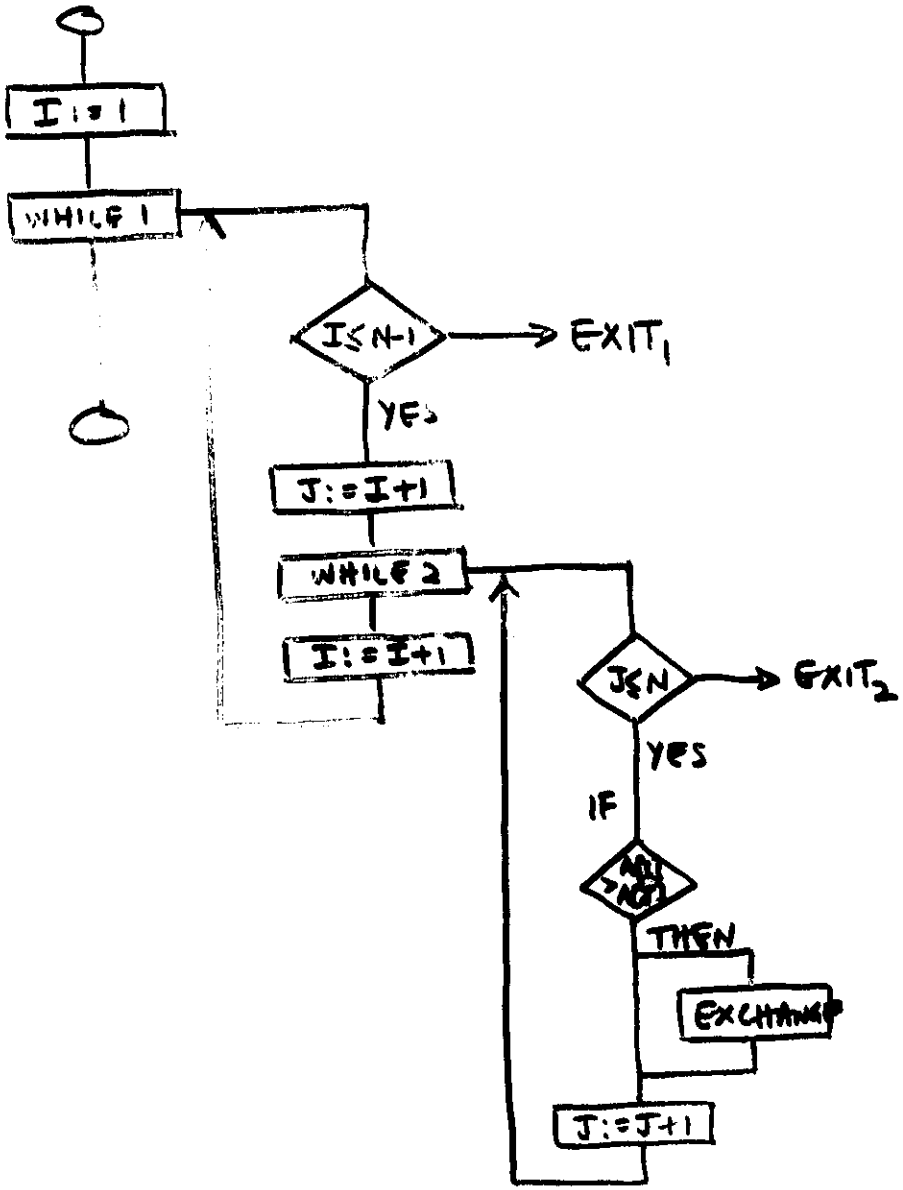


EG: N = 8, K = 5, M = 4 → M' = 1 → M = 4

RANDOM NUMBER GENERATION

$$R_{n+1} = (C_0 + C_1 * R_n) \bmod N$$

BUBBLE SORT



Second programming problem:

Handcoding or assembly language?

Are there any problems with handcoding?

* easy to make mistakes, nobody checks!
* address calculation/allocation tediou
* just imagine you want to change.....

What is Assembly Language?

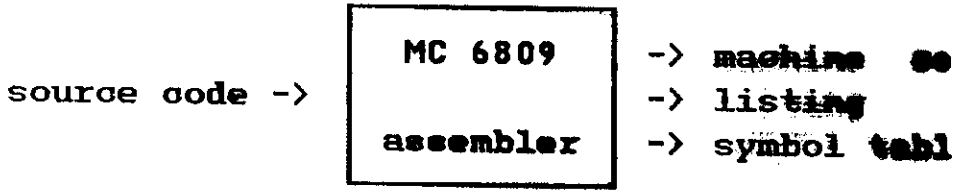
* easier to create and modify a progra
* easier to read and understand than handcode
* symbolic machine instructions (opcodes
* symbolic identifiers for registers
* symbolic address identifiers (labels)
* assembler directives for extra servic

foi

Flex MC 6809 Mnemonic Assembler:

The assembler is a computer program:

- # reading lines containing the source language of a MC 6809 assembler progra
- # translating source language into machi language for the MC 6809 microprocessc
- # producing an annotated listing of the program
- # producing a symbol table output



From where does it read?

Where does it store the information?

format.

Each source line may have up to 4 fields:

<label> <operation> <operand> <comment>

Label field:

starts in column 1, if with:

'*' (star) : --> comment line

' ' (space): --> label field is empty

identifier : --> label symbol

--> ERROR

Operation field:

follows label field with 1 blank minimum

opcode (machine instruction mnemonic)

pseudo (assembler directive)

macro call (evaluated body inserted her

Assembler source line format:

Operand field:

after operation field with 1 blank minimum

contents depends on operation, maybe:

* nothing
* register specifications (address modes)
* numeric constants, symbols, expressions

Comment field:

after operand field with 1 blank minimum

* optional but essential!!!
* comment flow of program
* opcode explanation is useless...

Assembly language elements:

Identifier:

* composed with:

  letters A-Z and a-z
  numbers 0-9
  underscore

* first character must be a letter
* upper/lower case letters are distinct
* first six characters are significant
* reserved identifiers:

"A", "B", "D"	- accumulators
"X", "Y"	- index registers
"U", "S"	- stack pointers
"DP"	- direct page reg.
"CC"	- condition code reg.
"PC", "PCR", "*"	- program counter

Assembly language elements:

Constants:

* decimal constant, (0 - 9)
  -32768 <= N <= 32767
* hexadecimal constant, (0 - 9), (A - F)
  0 <= N <= $FFFF
* octal constant (0 - 7)
  0 <= N <= @177777
* binary constant (0 - 1)
  0 <= N <= %1111111111111111
* character constant, ASCII character
  prefixed with "'" (apostrophe)

Opcodes:

* mnemonic for machine instruction
  (CLR, DEC, etc.)
* one to one correspondence between
  opcodes and machine instructions
* assembler verifies opcode and it's
  operands

Assembly language elements:

Assembler directives: (or: pseudo codes)

are instructions to the assembler!

* module identification
  (NAM, END)
* origin control
  (ORG)
* listing control
  (ERR, OPT, PAG, SPC, TTL, STTL)
* data generation and storage reservatio
  (FCB, FDB, FCC, RMB)
* symbol definition
  (EQU, REG, SET)
* object code control
  (RPT, SETDP)
* conditional assembly
  (IF, IFN, IFC, IFNC, ELSE, ENDIF)
* macro definition
  (MACRO, DUP, ENDD, EXITM, ENDM)

SEE: chapter V of TSC 6809 Assembler
manual for more details ...

Assembly language elements:

Expressions:

* an expression is a combination of:

symbols	constants
operators	parentheses

* arithmetic operators:

+	unary or binary addition
-	unary or binary subtraction
*	multiplication
/	division

* logical operators:

&	logical AND
|	logical OR
!	logical NOT

* shift operators:

>>	shift right
<<	shift left

MC 6809 addressing modes:

operand format	MC 6809 addressing mode
no operand	accumulator and inheren
<expression>	direct, extended, relativ
<<expression>	forced direct
> <expression>	forced extended
[<expression>]	extended indirect
<expression>,R	indexed
<<expression>,R	8-bit offset indexed
> <expression>,R	16-bit offset indexed
<accumulator>,Q	accum. offset indexed
[<expression>,R]	indexed indirect
<[<expression>,R]	same with 8-bit offset
>[<expression>,R]	same with 16-bit offset
[<accumulator>,Q]	same with accum. offset
Q+	auto increment by 1
Q++	auto increment by 2
[Q++]	auto increment indirect
-Q	auto decrement by 1
--Q	auto decrement by 2
[--Q]	auto decrement indirect
#<expression>	immediate
<register list>	immediate

with: R = S U X Y PC PCR
and : Q = S U X Y

Second programming problem:

Program our problem in assembly language

```
          NAM   Add first 15 integers
* Define begin of program area:
          ORG   $400
Start     CLR   Sum        Sum:= 0;
          LDA   EndVal     initialize coun
          STA   Count
Loop      LDA   Sum        accumulated sum
          ADDA  Count      add loop count
          STA   Sum        save it
          LDA   Count      re-load count
          DECA             next iteration
          STA   Count      save it
          BNE   Loop       repeat until ze
          ???              and now what?
* Define begin of data area:
          ORG   $420
Sum       RMB   1          to contain sum
EndVal    FCB   15         initial loop val
Count     RMB   1          to contain count

          END
```

2-PASS ASSEMBLY

- FEW OPCODES
- ADDRESSES ARE: EXTENDED
  INHERENT
  RELATIVE (BRANCHES)
- FEW PSEUDO OPS

SYMBOL-TABLE:

NAME	TYPE	VALUE
CLR	I	7F
LDA	I	B6
CLRA	I	4F
⋮	⋮	
ORG	P	—
COUNT	L	?422
⋮		

INITIALIZ

TOP

LOCATION COUNTER:

OBJECT:

PTR

BASIC ROUTINES

```
SCAN (SYMBOL)
SEARCH (SYMBOL, TYPE, VALUE, INDEX)
ENTER (SYMBOL, TYPE, VALUE, INDEX)

PASS1_IP;                    DEFINE LABELS
   CASE SYMBOL OF
   'ORG': LC:= NUMBER(NEXT_SYMBOL);
   'RMB': LC:= LC + NUMBER(NEXT_SYMBOL);
'FCB','CLRA','DECA',
    'SWI': LC:= LC+1;
'BRA','BNE',... : LC:= LC+2;
 OTHERWISE   LC =:LC+3;
   ENDCASE;

PASS1_LABEL;

   SYMBOL_TABLE[TOP].NAME:= SYMBOL
                    .TYPE:= L
                    .VALUE:= LC
```


PASS2 PROCEDURE (GENERATE OBJECT)

```
PASS2_IP;
   CASE SYMBOL OF
   'RMB': PTR:= PTR + NUMBER(NEXT_SYMBOL);
   'ORG': PTR:= NUMBER(NEXT_SYMBOL);
   'END': DONE:= TRUE
   'FCB': OBJECT[PTR] ⇐ NUMBER(NEXT_
                               SYMBOL,
   'CLRA': OBJECT[PTR]
                ⇐ TABLE['CLRA'].VALUE
     ⋮
   'BRA': [OBJECT[PTR]
                ⇐ TABLE['BRA'].VALUE;
           OBJECT[PTR]
                ⇐ TABLE[NEXT_SYMBOL].VALUE
                      - PTR+1
          ]
   'LDA': [OBJECT[PTR]
              ⇐ TABLE['LDA'].VALUE
           OBJECT[PTR]
              ⇐ TABLE[NEXT_SYMBOL].VALUE
          ]
     ⋮
ENDCASE;
```

START	L	400
LOOP	L	406
SUM	L	420
ENDVAL	L	421
COUNT	L	422

ORG → 400 CLR → 403 LDA → 406 STA → 409

409 ADDA → 40C STA → 40F LDA → 412

412 DECA → 413 BNE → 415 SWI → 416 RTS → 417

417

ORG → 420 RMB → 421 RMB → 422 RMB → 423

PTR = 400

400	7F	CLR
1	04	
2	20	
3	B6	LDA
4	04	
5	21	
6	B7	STA
7	04	
8	22	
9	BB	ADDA
A	04	
B	20	
C	B7	STA
D	04	
E	20	
F	B6	LDA
10	04	
11	20	
12	4A	DECA
13	26	
14	FF	
415 →	3F	
	00	

X

420 → 15

Second programming problem:

Optimize our assembly language program:

```
          NAM    Add first 15 integers
* Define monitor request
Rosy      EQU    0              to return to Ro
* Define begin of program area:
          ORG    $400
Start     CLR    Sum            Sum:= 0;
          LDA    EndVal         initialize coun
Loop      STA    Count          save current co
          ADDA   Sum            accumulated su
          STA    Sum            save it
          LDA    Count          re-load count
          DECA                  next iteratio
          BNE    Loop           repeat until
          SWI                   monitor call
          FCB    Rosy           return to Rosy
* Define begin of data area:
          ORG    $420
Sum       RMB    1              to contain sum
EndVal    FCB    15             initial loop va
Count     RMB    1              to contain coun
          END
```


Third programming problem:

Initialize an array to zero

Constraints:

* program starts at loc $0200
* array "Bin", 5 bytes long, starts at loc $0010

Let's do it the hard way:

```
          NAM    Initialize an array
          ORG    $10            data area
Bin       RMB    5              reserve 5 bytes
          ORG    $200           program area
Init      CLRA                  zero array "Bin"
          STA    Bin+0          Bin(1)
          STA    Bin+1          Bin(2)
          STA    Bin+2          Bin(3)
          STA    Bin+3          Bin(4)
          STA    Bin+4          Bin(5)
          SWI                   return to Rosy
          FCB    0
          END    Init
```

foil

Third programming problem:

Now let's use indexed addressing

Structure:

--> FOR <iterative condition> DO <bod

```
FOR Index:= 1 TO 5 DO
 Bin [Index] := 0;
```

-->

```
        NAM   Initialize an array
        ORG   $10          data area
Bin     RMB   5            reserve 5 bytes
        ORG   $200         program area
Init    CLRA               zero array "Bin"
        LDX   #0           our index
Loop    STA   Bin,X        zero next element
        LEAX  1,X          step index
        CMPX  #5           all done?
        BNE   Loop         if not
        SWI                else --> Resy
        FCB   0
        END   Init
```

12

foil

MC 6809 effective addressing modes:

--> effective address

is the address ultimately used by the MC 6809 microprocessor for it's operation

we use <ea> as abbreviation

--> inherent addressing

operation code specifies address

DECA --> 0400 4A

effective address for DECA is accumulator A

--> immediate addressing

data follows immediately operation code

```
LDX  #0     --> 0400  8E
                0401  00   <-- <ea>
                0402  00
```

effective address is in program counter once instruction has been decoded

MC 6809 effective addressing modes:

--> extended addressing

address of data follows operation code

```
LDA  $0421  --> 0400  B6
                0401  04
                0402  21
```

effective address is contained in the two bytes following operation code

--> direct addressing

address of data is computed using:

- contents of byte after opcode
- contents of direct page register

```
STA  $0010  --> 0400  97
                0401  10
```

assume direct page register contains

<ea> --> 05 10

contents of accum. A is stored into memory address 0510

MC 6809 effective addressing modes:

--> indexed addressing

base register --> register to be used for <ea> calculation

offset --> constant value or contents of accum. to be used for <ea> calculation

post byte --> byte following opcod defines the indexed addressing mode

* the effective address is calculated using the contents of base register and the offset (if specified)

* for indirect indexed addressing mode the effective address is used to read the word at the memory location it is pointing to and use it's contents as final effective address

MC 6809 effective addressing modes:

Indexed addressing:

--> constant offset from base register

```
STA  Bin,X     0400  A7
               0401  88 (1 00 0 1000)
               0402  10
```

<ea> --> contents of register X
+ signed 8-bit offset in loc. 0402

or:

```
STA  Bin-1,X   0400  A7
               0401  0F (0 00 0 1111)
```

<ea> --> contents of register X
+ signed 5-bit offset in post byte

or:

```
LEAX 1,X       0400  30
               0401  01 (0 00 0 0001)
```

<ea> --> contents of register X
+ signed 5-bit offset in post byte

MC 6809 effective addressing modes:

Indexed addressing:

--> accumulator offset from base register

* the effective address is calculated by adding the signed contents of accumulator A, B or D to the conten of the base register X, Y, S or U
* this addressing mode is very useful to handle lookup tables

Example:

Assume we are reading a character fro a terminal in ASCII format and have t convert it for further processing.

We use the value of the ASCII charact as an index into a table containing t conversion value:

```
LDX  #Table    code table base addre
LDB  CHAR      ASCII character code
LDA  B,X       corresponding code
```

Note: ASCII begins with code "00"!

MC 6809 effective addressing modes:

--> program counter relative addressing

* this address mode is implied for conditional branch instructions:

  we are branching to an instruction that is "some locations" away from where we are

  the branch instruction contains a constant signed displacement that is added to the program counter when the branch is taken

* following the same principle there is another indexed addressing mode:

  " constant offset
  from
  program counter "

MC 6809 effective addressing modes:

Indexed addressing:

--> constant offset from program counter

* the effective address is calculated by adding the signed offset to the contents of the program counter

* the offset is either an 8-bit or a 16-bit offset, there is no special 5-bit or accumulator offset

* the assembler will calculate the offset if we designate the address to be "program counter relative" using "PCR":

```
LEAX  L-Value,PCR
```

* this addressing mode is very useful to write position-independent code.