Chapter 2: Assembly Programming

Assembly Language Program Structure

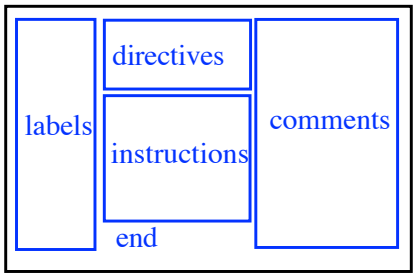
An assembly language program consists of a sequence of statements that tells the computer to perform the desired operations.

An assembly program consists of three sections: ***assembler directives***, ***assembly language instructions***, and ***comments***.

Assembler Directives: Instruct the assembler how to process subsequent assembly language instructions, provide a way to define program constants, and reserve space for dynamic variables:

- Define data and symbol.
- Reserve and initialize memory locations.
- Set assembler and linking condition.
- Specify output format.
- Specifies the end of a program.

structure of an assembly program in a text editor



Instructions: Consists of operations and operands. (See Appendix A of the textbook for complete list.)

Comments: Explain the function of a single or a group of instructions or directives, making a program more readable. Use **;** to indicate the beginning of a comment.

Each line of an assembly program, excluding certain special constructs, consists of four distinct fields: ***label***, ***operation***, ***operand***, and ***comment***.

The Label Field

- Labels are symbols defined by the user to identify memory locations in the programs and data areas of the assemble module
- Begin with a letter and followed by letters, digits, or special symbols (_ or .)
- Must start at column 1 if not ended with "**:**"
- Can start from any column if ended with "**:**"

The machine code of a program is stored in the memory line-by-line in the binary form → a programmer doesn't need to keep track of the memory address of each line, and simply referes as line by the line's "label."

```
$1034       begin:      LDAA #10          ; label begins in column 1
                …
            print:      JST    hexout     ; label is terminated by a colon
$11FB                   JMP    begin      ; jump to the line labeled as "begin"
```

For example, the line with the label "begin" is in the memory location with the address $1034 and the line with "print" is in $11FB, JMP instruction will redirect the back to address $1034.

The Operation Field

- Contains the mnemonic name for a machine instruction or an assembler directive.
- Separated from the label by at least one space.
- Consists of one opcode per line.

```
        ADDA        #$02        ; ADDA is the instruction mnemonic
true    EQU         1           ; equate directive EQU occupies the operation field
```

The Operand Field

- Separated from the operation field by at least one space.
- Contains operands for instructions or arguments for assembler directives.
- Consists of nothing, a number (prefixed with a #), or an address per line.

```
tcnt    EQU         $0084       ; $0080 is the operand field
        ADDA        $0090       ; $0090 is the operand field
```

The Comment Field [optional]

- Must have a * or ; prefixing any comment.
- Separated from the operand and operation field for at least one space.

```
; this program computes the square root of a 8-bit integer N
        org     $1000       ; set the location counter to memory location $1000
        dec     lp_cnt      ; decrement the loop counter
```

Example:

```
label       opcode      operand     comment
loop        ADDA        #$40        ; add $40 (hex) to accumulator A
```

- "ADDA" is an instruction mnemonic
- "#$40" is the operand (# = a number, not an address)
- "add #$40 to accumulator A" is a comment

Assembler Directives

Look just like instructions, but they **tell the assembler to do something** other than creating the machine code for an instruction.

end

- Ends a program to be processed by an assembler
- Any statement following END is ignored

org (origin)

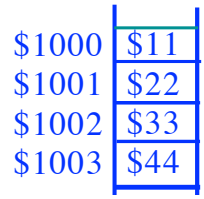
- ORG sets a new value for the location counter of the assembler.
- Mainly used to force a data table or a segment of instructions to start with a certain memory address.
- Should be used as infrequently as possible. Using too many ORG will make a program less reusable.

```
        ORG     $1000       ; forces the location counter to be set to $1000
        LDAB    #$FF        ; put the opcode byte at location $1000
```

dc.b (define constant byte)
db (define byte)
fcb (form constant byte)

- Define the value of a byte or bytes that will be placed at a given memory address.
- Often preceded by ORG.
- Multiple bytes can be defined at a time by using commas.

```
        org     $1000
array   dc.b    $11, $22, $33, $44
```



initializes the contents of memory locations at $1000, $1001, $1002, and $1003 to $11, $22, $33 and $44, respectively. The label, array, is used as the symbolic address of the first byte whose initial value is $11.

dc.w (define constant word; 1 word = 2 bytes)
dw (define word)
fdb (form double bytes)

- Define the value of a word or words that will be placed at a given memory address.

```
vec_tab     dc.w    $1234, $5678
```

initializes the two words starting from the current location counter to $1234 and $5678, respectively. After this statement, the location will be incremented by four.

fcc (form constant character)

- Define a string of characters (a message).
- Each character is represented by its ASCII form and stored in memory.
- A character string outputing to a LCD display is often defined using this directive.

```
msg:      fcc  "Please enter 1, 2, or 3:"     ; each character is stored in ASCII
alpha:    fcc  "def"                          ; d=$64, d=$65, f=$66 in ASCII
```

"mg and "alpha" are labels = symbolic addresses of the first byte of the two arrays = starting addresses of the two arrays in memory.

[See p. 14 of this lecture note for the ASCII table.]

fill (fill memory)

- Fill a certain number of memory locations with a given value.
- Syntax: **fill value, count**

```
space_line:   fill   $20, 40      ; a vetcor of 40 elemets with values of $20
```

fills 40 bytes with the value of $20 starting from the memory location reference to by the label "space_line". The address of the first byte is "space_line".

ds.b (define storage bytes)
ds (define storage)
rmb (reserve memory byte)

- Reserves a number of bytes given as the arguments to the directive. (for variables)
- The location counter will be incremented by the number that follows the directive mnemonic.

```
buffer:   ds.b   100      ; reserve memory space for a vector of 100 bytes
```

reserves 100 bytes in memory starting from the location represented by the label "buffer". After this directive, the location counter will be incremented by 100.

ds.w (define storage word)
rmw (reserve memory word)

- Each of these directives increments the location counter by the value indicated in the number-of-words argument ***multiplied by two***.

```
dbuf:     ds.w   20       ; a vector of 40 bytes
```

reserves 40 bytes starting from the momory location counter represented by the label "dbuf".

equ (equate)

- Assigns a constant value to a label.
- Define constants will makes a program more readable.

```
arr_cnt     equ     100             ; arr_cnt is now a constant = 100
```

informs the assembler that the symbol "arr_cnt" should be replaced with the value of 100 whenever it is encountered. arr_cnt contains a constant = 100, not an address location/label.

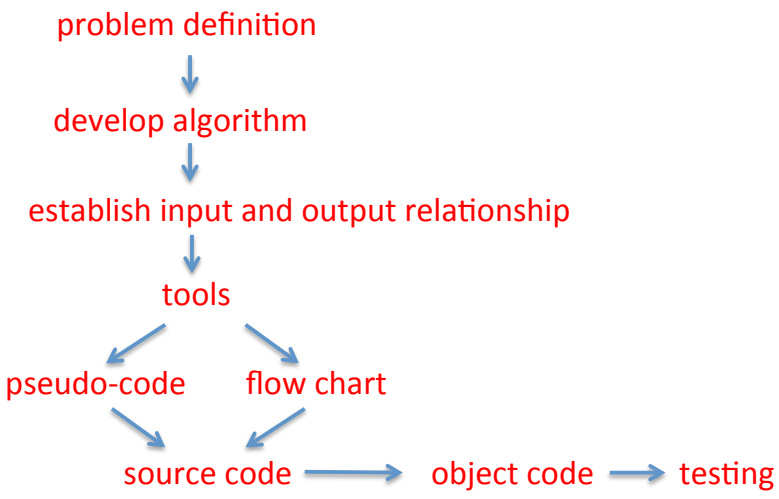
loc

- Increments and produces an internal counter used in conjunction with the backward tick mark (`).
- By using the loc directive and the ` mark, one can write program segments like the following example, without thinking up new labels:

```
        loc                                         loc
        ldaa    #2                                  ldaa    #2
loop`   deca                    ➔   loop001         deca
        bne     loop`                               bne     loop001
        loc                                         loc
loop`   brclr   0, X, $55, loop`    ➔   loop002     brclr   0, X, $55, loop002
```

The left segment will work perfectly fine because the first loop label will be seen as loop001, whereas the second loop label will be seen as loop002. The assembler actually sees the right segment.

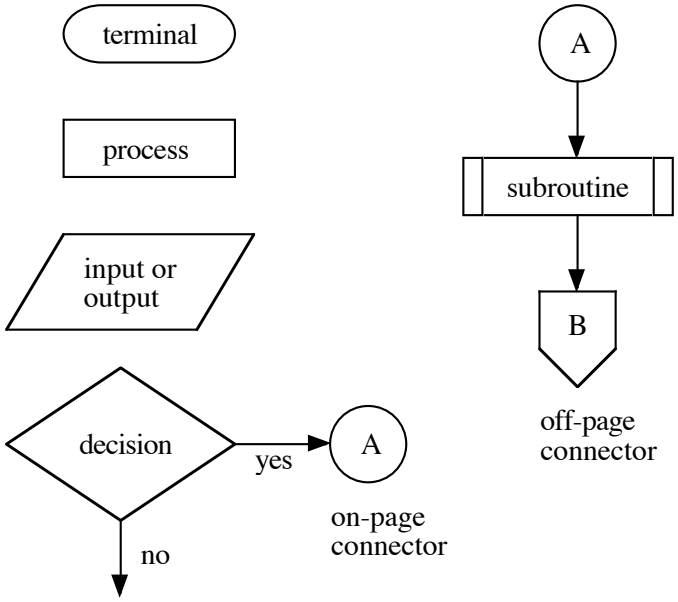
Software Development Issues



1. Software development starts with ***problem definition***, identifying to be done.
2. Develop the ***algorithm***, which presents the overall plan for solving the problem.
   - An algorithm is a sequence of computational steps that transforms the input into the output. That is, the desired ***input/output relationship***.
   - An algorithm, in **pseudo-code**, documents the software to be developed.

     Step 1
     …
     Step 2
     …

   - Another way to express overall plan is to use **flowchart**, showing the way a program operates, illustrating the logic flow of the program.



The ***terminal*** box is used at the beginning (with the word ***Start***) and end (with the word ***Stop***) of each program.

The ***process*** box indicates what must be done at this point in the program execution.

The ***input/output*** box is used to represent data that are either read or displayed by the computer.

The ***decision*** box contains a question that can be answered by either yes or no.

The ***on-page*** connector indicates that the flowchart continues elsewhere on the same page. The place where it is continued will have the same label as the on-page connector.

The ***off-page connector*** indicates that the flowchart continues on another page with matching off-page connector.

Normal flow on a flowchart is from top to bottom and from left to right. Any line that does not follow this normal flow should have an ***arrowhead*** on it.

3. After you are satisfied with the algorithm or flowchart, convert it to source code in the assembly or high-level language.
4. Program testing.

Arithmetic Examples

Example: Add the numbers stored at memory locations $1000, $1001, and $1002, and save the sum at memory location $1100.

Pseudo code

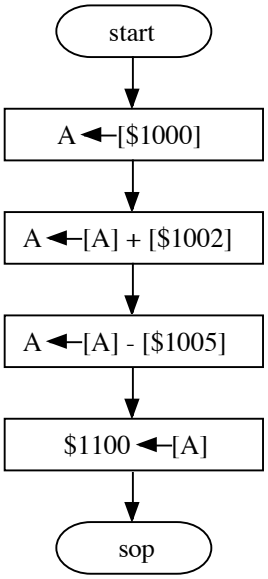
```
Step 1:    A ← m[$1000]          ; use LDAA (load into A)
Step 2:    A ← A + m[$1001]      ; use ADDA (add w/o carry)
Step 3:    A ← A + m[$1002]
Step 4:    $1100 ← A             ; use STAA (store to m[$1100])
```

```
ORG     $2000     ; starting memory location of the program
LDAA    $1000     ; place the contents at location $1000 into A
ADDA    $1001     ; add the contents at location $1001 into A
ADDA    $1002     ; add the contents at location $1002 into A
STAA    $1100     ; store the sum at location $1100
SWI               ; release the control back to D-Bug12
END               ; tell assembler this is the end of program
```

Example: Subtract the number stored at the memory location $1005 from the sum of the memory locations at $1000 and $1002, and store the difference at $1100.

use flow chart →

```
ORG     $2000
LDAA    $1000     ; A ← m[$1000]
ADDA    $1002     ; A ← A+m{$1002]
SUBA    $1005     ; A ← A-m[$1005]
STAA    $1100
SWI
END
```



SUBA = subtarct the number in a memory location from A w/o borrow; A ← A–m[????]

Example: Subtract 5 from two memory locations at $1000 and $1001.

In the HCS12, a memory location cannot be the destination of an ADD or SUB instruction.

Step 1: load the memory location into an accumulator. ; A ← m[$1000]
Step 2: add (or subtract) the number to (from) the accumulator ; A ← A - #5
Step 3: store the result at the specified memory location ; m[$1000] ← A

```
        ORG     $2000       ; start address of the program
1:      LDAA    $1000       ; copy the content at location $1000 to A
2:      SUBA    #5          ; subtract 5 from A
3:      STAA    $1000       ; store the result back to $1000
1:      LDAA    $1001       ; repeat the instructions with location $1001
2:      SUBA    #5          ;       "
3:      STAA    $1001       ;       "
        SWI
        END
```

Example: Add two 16-bit numbers that are stored at $1000-$1001 and $1002-$1003. Afterwards, store the sum at $1100-$1101.

```
        ORG     $2000       ;
        LDD     $1000       ; place the 16-bit number in D ←m[$1000:$1001]
        ADDD    $1002       ; add the 16-bit number at $1002-$1003 to D
                            ; D ← D + m[$1002:$1003]
        STD     $1100       ; save the sum from D → m[$1100:$1101]
        SWI
        END
```

The Carry/Borrow Flag

	N	V	Z	C

- Bit 0 of the CCR register
- Set to 1 when the addition operation produces a carry 1
- Set to 1 when the subtraction operation produces a borrow 1
- Enables the user to implement multiprecision (> 16 bits) arithmetic.

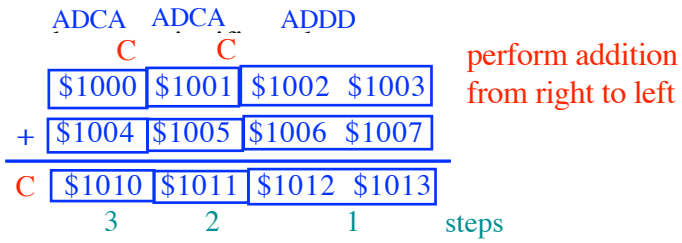
```
LDD     #$8654      ; 16-bit number needs to use accumulator D
ADDD    #$9978      ; C flag = 1 in the CCR register

          $8 6 4 5
        + $9 9 7 8
        $1 1 F B D
carry|    sum
```

Example: Add two 4-byte numbers that are stored at $ 1000-$1003 and $1004-$1007, and store the sum at $1010-$1013.

Note: no instruction for 16-bit addition with carry.

Addition starts from the least significant byte and proceeds toward the most significant byte.

```
                org     $2000           ; starting address of the program
                ldd     $1002           ; copy the lowest two bytes of the first operand in D
1:              addd    $1006           ; add the least significant two bytes
                std     $1012           ; save the sum of the least significant two bytes
; needs to use accumulator A because there is no "add with carry" in D
                ldaa    $1001           ; add and save the second most significant bytes
2:              adca    $1005           ; "add with carry to A" from bit 0 of the CCR
                staa    $1011           ;
; repeat the steps with the most significant bytes
                ldaa    $1000           ; add and save the most significant bytes
3:              adca    $1004           ;
                staa    $1010           ;
                swi
                end
```



Example: Subtract the 4-byte number stored at $1004-$1007 from the 4-byte number stored at $1000-$1003, and save the result at $1100-$1103.

Note: no instruction for 16-bit subtraction with borrow. Subtraction starts from the least significant byte and proceeds toward the most significant byte.

Simply replace ADDD wih SUBD and ADCA with SBCA.

```
                org     $2000
                ldd     $1002  ; copy the lowest two bytes of the minuend to D
1:              subd    $1006  ; subtract the lowest two bytes of the subtrahend from D
                std     $1102  ; save the lowest two bytes of the difference
; needs to use accumulator A because there is no "subtract with carry" in D
                ldaa    $1001  ; subtract the second to most significant bytes
2:              sbca    $1005  ; "subtract with the borrow from A"
                staa    $1101  ; save the difference of the second to most significant bytes
; repeat the steps with the most significant bytes
                ldaa    $1000  ; subtract and save the difference of the
3:              sbca    $1004  ; most significant bytes
                staa    $1100  ;
                swi
                end
```

Binary-Coded-Decimal (BCD) Addition

- Each ***decimal digit*** is encoded by ***4 bits***. Simplifies I/O conversion.
- Example: 2538 (= %100111101010) can represented by 0010 0101 0011 1000 in BCD format.
- The addition of two BCD numbers is performed by binary addition (ADDA, ADCA, or ABA) and followed by an adjust operation using the DAA instruction.
- DAA = 4-bit binary sum + 6 [See ENGG 32A book or lecture note]
- DAA does not work with subtraction.

```
LDAA    $1000    ; load two BCD digits into A
ADDA    $1001    ; add two BCD digits in A
DAA              ; decimal adjust the sum in A
STAA    $1002    ; save the sum in BCD form
```

Multiplication and Division [Be careful on which registers are used.]

Mnemonic	Function	Operation
EMUL	unsigned 16 by 16 multiply	(D) x (Y) → Y:D = **32 bits**
EMULS	signed 16 by 16 multiply	(D) x (Y) → Y:D
MUL	unsigned 8 by 8 multiply	(A) x (B) → A:B = **16 bits**
EDIV	unsigned 32 by 16 divide	(Y:D) ÷ (X) quotient → Y; remainder → D
EDIVS	signed 32 by 16 divide	(Y:D) ÷ (X) quotient → Y; remainder → D
FDIV	16 by 16 fractional divide [quotient in fraction between 0 & 1]	(D) ÷ (X) → X (quotient) remainder → D
IDIV	unsigned 16 by 16 integer divide [quotient in integer]	(D) ÷ (X) → X (quotient) remainder → D
IDIVS	signed 16 by 16 integer divide	(D) ÷ (X) → X (quotient) remainder → D

EMUL multiplies the 16-bit unsigned integers stored in D and Y. The upper 16 bits of the product are found in Y and the lower 16 bits of the product are found in D.

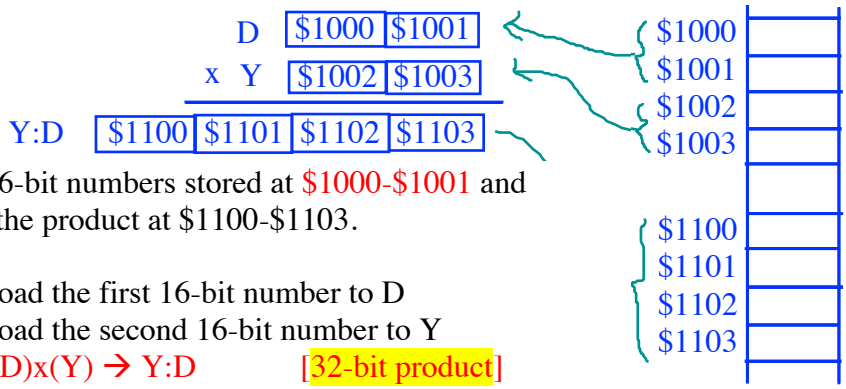
EDIV performs an unsigned 32-bit by 16-bit division. The upper 16-bit dividend is found in Y and the lower 16-bit dividend is found in D. X is the divisor. After division, the quotient and remainder are placed in Y and D, respectively.

D=$0003
X=$0006

FDIV ---> D/X = 0.5 (fraction is stored in X)

X: | 1 0 0 0 0 0 0 0 0 … 0 |

positional notation $2^{-1}\ 2^{-2}\ 2^{-3}\ 2^{-4}$ ...



Example: Multiply the 16-bit numbers stored at $1000-$1001 and $1002-$1003, and store the product at $1100-$1103.

```
LDD   $1000      ; load the first 16-bit number to D
LDY   $1002      ; load the second 16-bit number to Y
EMUL             ; (D)x(Y) → Y:D        [32-bit product]
STY   $1100      ; $1100 has the higher byte
STD   $1102      ; $1102 has the lower byte
```

Example: Multiply the contents of X and D. Store the product at $1000-$1003.

No instruction to multiply X and D; need to transfer the content of X to Y.

```
STY   $1010      ; save Y in a temporary location
TFR   X, Y       ; transfer the contents of X to Y
EMUL             ; (D)x(Y) → Y: D       [32-bit product]
STY   $1000
STD   $1002
LDY   $1010      ; restore the value of Y
```

Example: Divide the unsigned 16-bit number stored at $1005-$1006 by the unsigned 16-bit number at $1020-$1021. Store the quotient and remainder at $1100 and $1102, respectively.

```
LDD   $1005      ; place the dividend in D
LDX   $1020      ; place the divisor in X
IDIV             ; (D) ÷ (X) → quotient X & remainder D
STX   $1100      ; store the quotient
STD   $1102      ; store the remainder
```

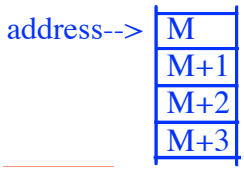
Exchange between accumulator D and index registers X and Y

Because most arithmetic operations are performed only on accumulators, we need to transfer the contents of index register X to accumulator D so that further division on the quotient can be performed.

XGDX exchanges the contents of D and X
XGDY exchanges the contents of D and Y

32-bit by 32-bit Multiplication (4 bytes x 4 bytes = 8-byte product)

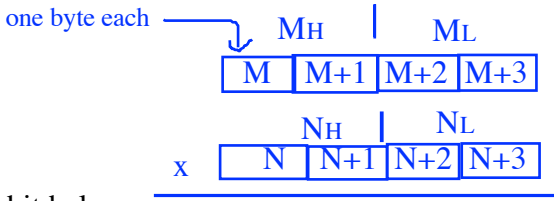
Multiply two unsigned 32-bit numbers stored at M~M+3 and N~N+3, respectively, and store the product at P~P+7.



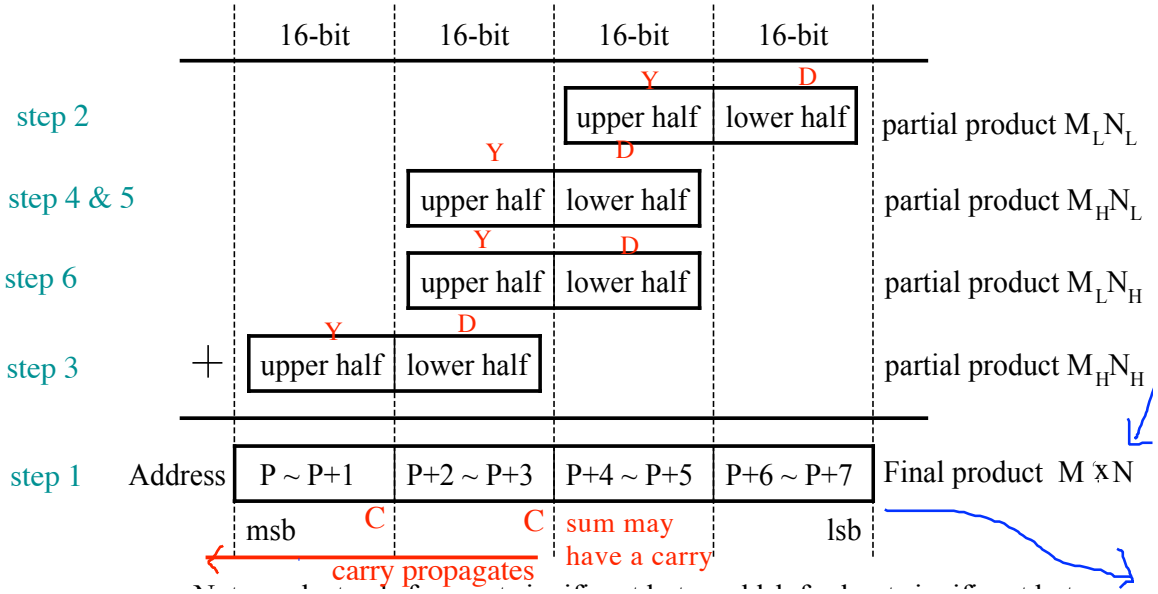
Two 32-bit numbers M and N are divided into two (16-bit) halves

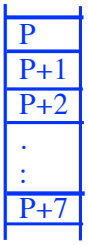
$M = M_H M_L$

$N = N_H N_L$



Use the four 16-bit by 16-bit multiplications [EMUL: D x Y → Y:D] to synthesize the operation. Their 4-byte (=32-bits) partial products are finally added together to give a 8-byte (=64-bits) product.





Step 1: Allocate 8 bytes to hold the product at locations P~P+7.
Step 2: Generate the partial product $M_L N_L$ (in Y:D) and save it at P+4~P+7.
Step 3: Generate the partial product $M_H N_H$ (in Y:D) and save it at P~P+3.
Step 4: Generate the partial product $M_H N_L$ (in Y:D) and add it to locations P+2~P+5. The C flag may be set to 1 after this addition.
Step 5: [Allow Carry to propagate to the left.] Add the C flag to location P+1 using ADCA (or ADCB). This addition may also set the C flag to 1 after this addition. So, again, add the C flag to location P.
Step 6: [Allow Carry to propagate to the left.] Generate the partial product $M_L N_H$ (in Y:D) and add it to locations P+2~P+5. The C flag may be set to 1 after this addition. So, add the C flag to location P+1 and then add it to location P.

```
                        org     $1000       ; starting address of the storage of the numbers
      $1000 -> M        ds.b    4           ; reserve 4 bytes to hold the multiplicand
step 1 $1004 -> N       ds.b    4           ; reserve 4 bytes to hold the multiplier
      $1008 -> P        ds.b    8           ; reserve 8 bytes to hold the product

-------------------------------------------------------------------------------
                        org     $2000       ; starting address of the program
                        ldd     M+2         ; place ML in D                 [M+2 = $1002]
step 2                  ldy     N+2         ; place NL in Y                 [N+2 = $1005]
                        emul                ; compute MLNL  (store in Y:D)
                        sty     P+4         ; save the upper 16 bits of the partial product MLNL
                        std     P+6         ; save the lower 16 bits of the partial product MLNL

-------------------------------------------------------------------------------
                        ldd     M           ; place MH in D
                        ldy     N           ; place NH in Y
step 3                  emul                ; compute MHNH
                        sty     P
                        std     P+2

-------------------------------------------------------------------------------
                        ldd     M           ; place MH in D
                        ldy     N+2         ; place NL in Y
                        emul                ; compute MHNL
; the following instructions add MHNL to memory locations P+2~P+5
                        addd    P+4         ; add lower half of MHNL and upper half of MLNL
step 4                  std     P+4         ; save the sum to P+4~P+5
                        tfr     Y,D         ; move the upper half of MHNL to A:B → P+2:P+3
                        adcb    P+3         ; "add with carry" upper half of MHNL and
                        stab    P+3         ; lower half of MLNL
                        adca    P+2
                        staa    P+2

-------------------------------------------------------------------------------
; the following instructions propagate carry to the most significant byte
                        ldaa    P+1
                        adca    #0          ; add carry to the location at P+1
step 5                  staa    P+1         ;       "
                        ldaa    P           ; add carry to the location at P
                        adca    #0          ;       "
                        staa    P           ;       "

-------------------------------------------------------------------------------
; the following instructions compute MLNH
                        ldd     M+2         ; place ML in D
                        ldy     N           ; place NH in
step 6                  emul                ; compute MLNH
; the following instructions add MLNH to memory locations P+2 ~ P+5
                        addd    P+4         ; add lower half of MLNH with the contents of P+4
                        std     P+4         ; save the sum to P+4~P+5
                        tfr     Y,D         ; transfer Y to D
```

```
        adcb    P+3             ; add upper half of M_L N_H with the contents of P+3
        stab    P+3
        adca    P+2
        staa    P+2
; the following instruction propagate carry to the most significant byte
        clra                    ; clear A
        adca    P+1             ; add with carry
        staa    P+1             ; save sum to location P
        ldaa    P
        adca    #0              ; add C flag to location P
        staa    P
        end
```

ASCII Table

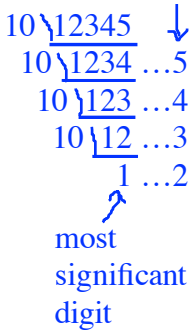
differ by $20

Decimal	Hex	ASCII	Decimal	Hex	ASCII	Decimal	Hex	ASCII	Decimal	Hex	ASCII
0	00	NUL	32	20	(blank)	64	40	@	96	60	`
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	"	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	'	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	(delete)

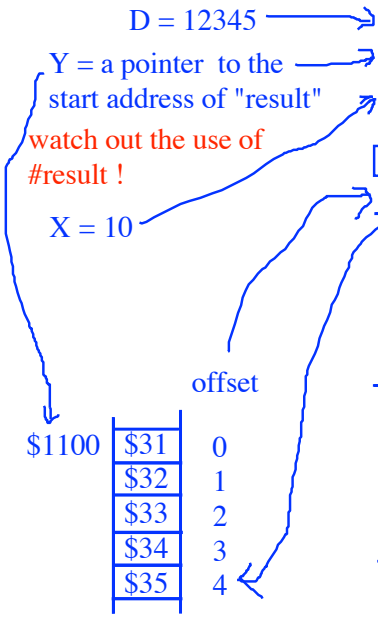
Example: Convert the 16-bit number stored at $1000-$1001 to BCD form, then convert each BCD digit into its ASCII code, and store the result at $1100-$1104.

[Application: Display a number stored in computer on a screen.]

- A binary number can be converted to BCD format using division-by-10.
- The largest 16-bit binary number is 65535, which has 5 decimal digits.
- The reminder of the first division-by-10 obtains the least significant digit, the second division-by-10 obtains the second least significant digit, ...
- The ASCII code is obtained by adding $30 to each BCD digit.




```
                org     $1000
data            dc.w    12345       ; place a decimal number to be tested in
                                    ; the label "data" (stored as a 16-bit binary number)
                org     $1100
result          ds.b    5           ; reserve 5 bytes to store the result

                org     $2000       ; starting address of the program
                ldd     data        ; copy the 16-bit number to D
                ldy     #result     ; store the memory location of result in Y = $1000
                ldx     #10         ; divide the number by 10
                idiv                ; (D)÷(X) → quotient X, remainder D=A:B  [A=0]
                addb    #$30        ; convert the digit stored in B into ASCII code
                stab    4, Y        ; save the least significant digit to location result+4
                xgdx                ; swap the quotient X to D
                ldx     #10
                idiv
                adcb    #$30        ; convert the digit into ASCII code
                stab    3,Y         ; save the second-to-least significant digit
                xgdx
                ldx     #10
                idiv
                addb    #$30
                stab    2,Y         ; save the middle digit
                xgdx
                ldx     #10
                idiv                ; separate the most significant and second-to-most
                addb    #$30        ; significant digits
                stab    1,Y         ; save the second-to-most significant digit
                xgdx                ; swap the most significant digit to B
                addb    #$30        ; convert the digit into ASCII code
                stab    0,Y         ; save the most significant digit
                swi                 ; ASCII form of 12345 ($31, $32, $33, $34, $35)
                end                 ; can be found in memory location $1100~$1104
```



[What happens if the decimal number is less than 5 digits? How to shorten the program?]

Program Loops

- For repetitive operations.
- Types of program loops: *finite* and *infinite* loops

1. **Do** *statement S* **forever**
   - An infinite loop in which statement S is repeated forever.
   - May add a condition statement "**If** condition *C=true* **then exit**" the infinite loop.

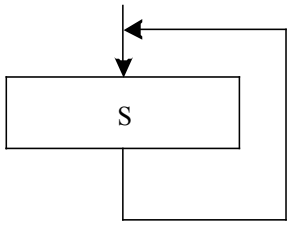
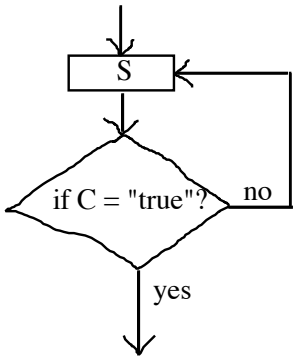


Figure 2.4 An infinite loop



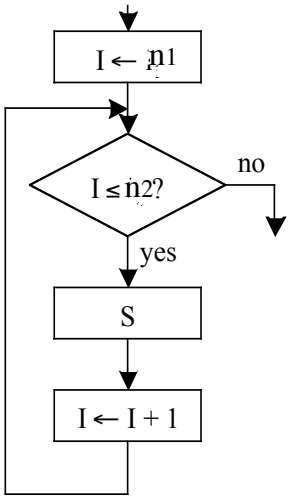
2. **For** *i = n1* **to** *n2* **do** *statement S* or **For** *i = n2* **downto** *n1* **do** *statement S*
   - Variable *i* is the loop counter, which can be incremented or decremented, and keeps track of the number of remaining times statement S is to be executed.
   - Statement S is repeated n2-n1+1 times, where n1 ≤ n2.
   - Four steps are required to implement a For loop:
     Step 1: Initialize the loop counter and other variables.
     Step 2: Compare the loop counter with the limit to see if it is within bounds. If it is, executive the specified operations. Otherwise, exit the loop.
     Step 3: Increment (or decrement) the loop counter.
     Step 4: Go to step2.

initialize

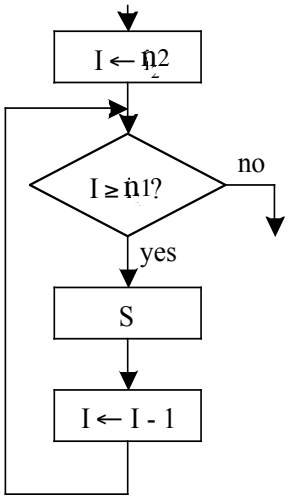
compare

execute

increase/decrease counter value by 1



(a) For I = i_1 to i_2 DO S



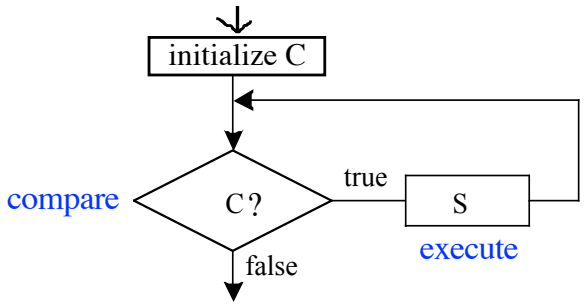
(b) For I = i_2 downto i_1 DO S

compare first, then execute

i = loop counter

3. **While** condition *C=true* **do** *statement S*

- The logical expression C is evaluated first before executing the While construct
- Four steps are required to implement a While loop:
  Step 1: Initialize the logical expression C. (C = condition)
  Step 2: Evaluate the logical expression C.
  Step 3: Execute the specified operations S, update C, and go to step 2 if C is true. If C is false, go to step 4 without exccuting S.
  Step 4: Exit the loop.

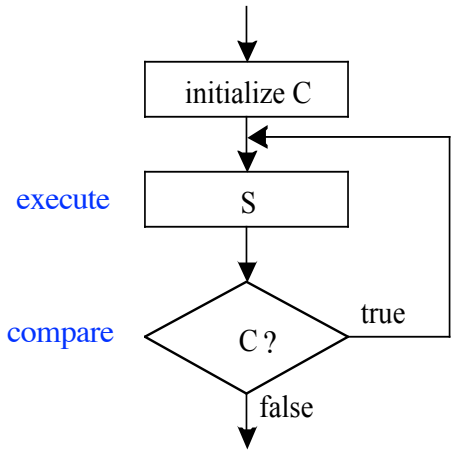


compare first, then execute

C = condition

4. **Repeat** *statement S* **until** *C=false*

- Statement S is first executed and then the logical expression C is evaluated.
- If C is false, the next statement S will not be executed anymore.
- Three steps are required to implement a While loop:
  Step 1: Initialize the logical expression C. (C = condition)
  Step 2: Execute statement S and update C.
  Step 3: Go to step 2 if C is true. Otherwise, exit.

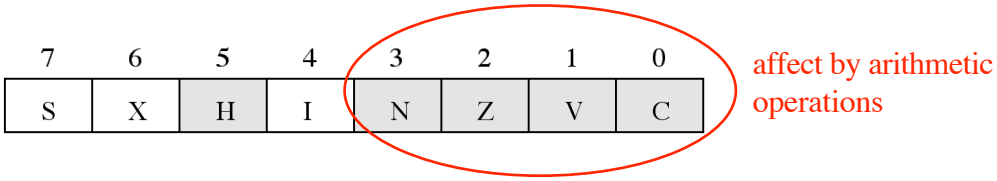


execute first, then compute

[Where does the conditon C come from?]

Condition Code Register (CCR)

Program loops are implemented by using the conditional branch instructions and the execution of these instructions depends on the contents of the CCR.



The shaded characters are condition flags that reflect the status of an operation.

C: the carry flag

- Whenever a carry is generated as the result of an operation, the flag C = 1.
- Otherwise, it will be cleared to 0.

V: the overflow flag

- Whenever the result of a 2's complement arithmetic operation is out of range, this flag V = 1.
- Otherwise, it will be cleared to 0.
- This flag will be set to 1 when the carry from the MSB and the second MSB differ as the result of an arithmetic operation.

Z: the zero flag

- Whenever the result of an operation is zero, the flag Z = 1.
- Otherwise, it will be cleared to 0.

N: the negative flag

- This flag indicates that the result of an operation is negative
- Whenever the MSB (sign bit) of the result of an operation is 1, the flag N = 1.
- Otherwise, it will be cleared to 0.

H: the half-carry flag

- Whenever there is a carry from the lower four bits to the upper four bits as the result of an operation, the flag H = 1.
- Otherwise, it will be cleared to 0.

Branch Instructions

- Branch instructions cause program flow to change when specific conditions exist.
- Four categories of branch instructions:
    1. **Unconditional branch:** always execute.
    2. **Simple branch:** a branch is taken when a specific bit of CCR is in a specific status as a result of a previous operation.
    3. **Unsigned branch**: a branch is taken when a comparison or test of ***unsigned*** numbers results in a specific combination of CCR bits.
    4. **Signed branch:** a branch is taken when a comparison or test of ***signed*** numbers results in a specific combination of CCR bits.
- Two kinds of branches
    1. **Short Branch:** a signed 8-bit offset is added to the value of the program counter (PC) when a specified condition is met. Program execution continues at the new address. The numeric range of the offset value is in the range of –128 ~ +127 (or $80 ~ $7F in hex) bytes.
    2. **Long Branches:** a signed 16-bit offset is added to the value of the program counter (PC) when a specified condition is met. Program execution continues at the new address. The numeric range of the offset value is in the range of –32768 ~ 32767 (or $8000 ~ $7FFF in hex) byte. This permits branching from any location in the 64-Kbyte address map.

Common formats:

```
loop:   …                    or      loop:   …
        .                                    .
        :                                    :
        Bcc loop                             LBcc loop
```

B = short branch; **LB** = long branch

From pp. 20 & 21, **cc** is one of the condition codes: CC, CS, EQ, MI, NE, PL, VC, VS, HI, HS, LO, LS, GE, GT, LS, and LT.

Usually a comparison or arithmetic instruction (see p. 22) is executed to set up the CCR bits before the conditional branch instruction.

Table 2.2 Summary of short branch instructions

Unary Branches		
Mnemonic	Function	Equation or Operation
BRA	Branch always	1 = 1
BRN	Branch never	1 = 0
Simple Branches		
Mnemonic	Function	Equation or Operation
BCC	Branch if carry clear (or borrow)	C = 0
BCS	Branch if carry set (or borrow)	C = 1
BEQ	Branch if equal	Z = 1
BMI	Branch if minus	N = 1
BNE	Branch if not equal	Z = 0
BPL	Branch if plus	N = 0
BVC	Branch if overflow clear	V = 0
BVS	Branch if overflow set	V = 1
Unsigned Branches		+ = or
Mnemonic	Function	Equation or Operation
BHI	Branch if higher	C + Z = 0
BHS	Branch if higher or same	C = 0
BLO	Branch if lower	C = 1
BLS	Branch if lower or same	C + Z = 1
Signed Branches		
Mnemonic	Function	Equation or Operation
BGE	Branch if greater than or equal	N ⊕ V = 0
BGT	Branch if greater than	Z + (N ⊕ V) = 0
BLE	Branch if less than or equal	Z + (N ⊕ V) = 1
BLT	Branch if less than	N ⊕ V = 1

use afer A-B or (A,B,D)-M, then check a CCR bit

compare two signed numbers

first perform A-B

if A-B>0, then no borrow and Z=0

Z = 0 or 1

if A-B,0, then borrow exists

compare two signed numbers

Need to run a comparison instruction before running a branch instruction! (see p. 22)

Table 2.3 Summary of long branch instructions

Unary Branches		
Mnemonic	Function	Equation or Operation
LBRA	Long branch always	1 = 1
LBRN	Long branch never	1 = 0
Simple Branches		
Mnemonic	Function	Equation or Operation
LBCC	Long branch if carry clear	C = 0
LBCS	Long branch if carry set	C = 1
LBEQ	Long branch if equal	Z = 1
LBMI	Long branch if minus	N = 1
LBNE	Long branch if not equal	Z = 0
LBPL	Long branch if plus	N = 0
LBVC	Long branch if overflow is clear	V = 0
LBVS	Long branch if overflow set	V = 1
Unsigned Branches		
Mnemonic	Function	Equation or Operation
LBHI	Long branch if higher	C + Z = 0
LBHS	Long branch if higher or same	C = 0
LBLO	Long branch if lower	C = 1
LBLS	Long branch if lower or same	C + Z = 1
Signed Branches		
Mnemonic	Function	Equation or Operation
LBGE	Long branch if greater than or equal	$N \oplus V = 0$
LBGT	Long branch if greater than	$Z + (N \oplus V) = 0$
LBLE	Long branch if less than or equal	$Z + (N \oplus V) = 1$
LBLT	Long branch if less than	$N \oplus V = 1$

Compare and Test Instructions

- CCR flags need to be set up before a conditional branch instruction is executed.
- A group of compare instructions for setting the CCR flags.
- A group of instructions for testing the CCR flags.
- Most instructions update CCR flags automatically.

Table 2.4 Summary of compare and test instructions

Compare instructions		
Mnemonic	Function	Operation
CBA	Compare A to B	(A) - (B)
CMPA	Compare A to memory	(A) - (M)
CMPB	Compare B to memory	(B) - (M)
CPD	Compare D to memory	(D) - (M:M+1)
CPS	Compare SP to memory	(SP) - (M:M+1)
CPX	Compare X to memory	(X) - (M:M+1)
CPY	Compare Y to memory	(Y) - (M:M+1)
Test instructions		
Mnemonic	Function	Operation
TST	Test memory for zero or minus	(M) - $00
TSTA	Test A for zero or minus	(A) - $00
TSTB	Test B for zero or minus	(B) - $00

Decrementing and Incrementing Instructions

- Often need to add or subtract one from a variable in a program.
- Instructions to increment or decrement a variable by one.
- Application: increase/decrease a loop counter or address pointer by 1

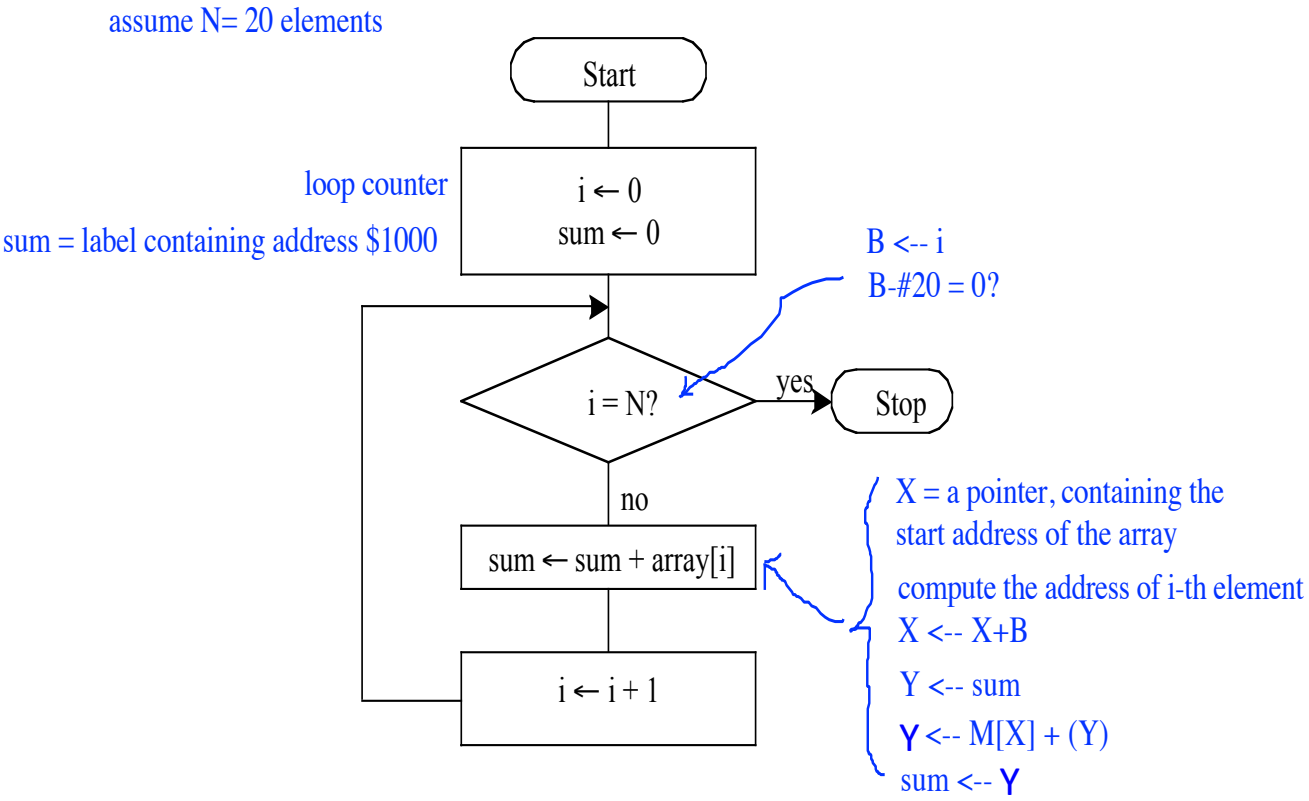
Decrement instructions		
Mnemonic	**Function**	**Operation**
DEC	Decrement memory by 1	M ← (M) - $01
DECA	Decrement A by 1	A ← (A) - $01
DECB	Decrement B by 1	B ← (B) - $01
DES	Decrement SP by 1	SP ← (SP) - $01
DEX	Decrement X by 1	X ← (X) - $01
DEY	Decrement Y by 1	Y ← (Y) - $01
Increment instructions		
Mnemonic	**Function**	**Operation**
INC	Increment memory by 1	M ← (M) + $01
INCA	Increment A by 1	A ← (A) + $01
INCB	Increment B by 1	B ← (B) + $01
INS	Increment SP by 1	SP ← (SP) + $01
INX	Increment X by 1	X ← (X) + $01
INY	Increment Y by 1	Y ← (Y) + $01

Table 2.6 ■ Summary of decrement and increment instructions

Example:

```
LDAA    i       ; this sequence is the same as  INC i
ADDA    #1
STAA    i
```

Example: Add an array of N 8-bit numbers and store the sum at memory locations $1000~$1001. Use the For *i = n1* to *n2* do looping construct.



Variable *i* is the loop counter, used to keep track of the number of iterations remained to be performed.

i can also be served as the array index.

Variable sum has 2 bytes to hold the sum of array elements.

```
constant -->  N       equ     20          ; array size = 20 = the number of 8-bit numbers
              org     $1000               ; starting address of on-chip SRAM
$1000 -->     sum     rmb     2           ; reserve 2 bytes for the sum
$1002 -->     i       rmb     1           ; reserve 1 byte for index i (loop counter)

              org     $2000               ; starting address of the program
              ldaa    #0                  ; clear A
initialize    staa    i                   ; load A to i for initializing loop counter to 0
              staa    sum                 ; load 0 to the lower byte of the sum
              staa    sum+1               ; load 0 to the upper byte of the sum

              ; iteration
loop          ldab    i                   ; copy index i to B. (i starts from 0)
              cmpb    #N                  ; is i = N?       [cmpb #N = (B) - #$20]
              beq     done                ; if equal, then branch to the address of the label done
              ldx     #array              ; if not, use index register X as a pointer to the array
              abx                         ; compute the address of array[i] (abx = B+X→X)
              ldab    0,X                 ; place array[i] in B      [B has the i-th element of the array]
              ldy     sum                 ; place sum in Y
              aby                         ; compute sum ← sum + array[i]  (aby = B+Y→Y)
              sty     sum                 ; update sum
              inc     i                   ; increment the loop count by 1
              bra     loop                ;

done          swi                         ; return to D-Bug12 monitor

              ; the array is defined in the following statement
array         dc.b    11, 22, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

              end
```

watch out the use of "#"

#array = a number = the start address of the array = the address of the first element.

What is LDX array?

Without the "#", the content of the address contained in the label "array" is loaded.
→ X = $1122 in this example because "11" is the first element of the array and X contains 2 bytes (from 2 consecutive addresses).

Loop Primitive Instructions

- Program loops are usually implemented by incrementing or decrementing the value of a loop counter.
- Loop-primitive instructions can decrement or increment a loop counter to determine if the looping should be continued.
- These instructions test a counter value in a register or accumulator (A, B, D, X, Y, or SP) for zero or nonzero value as a branch condition.
- The range of the branch is -128 ~ +127 (or $80 ~ $7F in hex) byte from the instruction immediately following the loop primitive instruction.

Combine increment/decrement of a loop counter and comparison in one instruction.

D = decrement
I = increment

B = branch

EQ = equal to 0

NE = not equal to 0

cntr = counter
rel = label

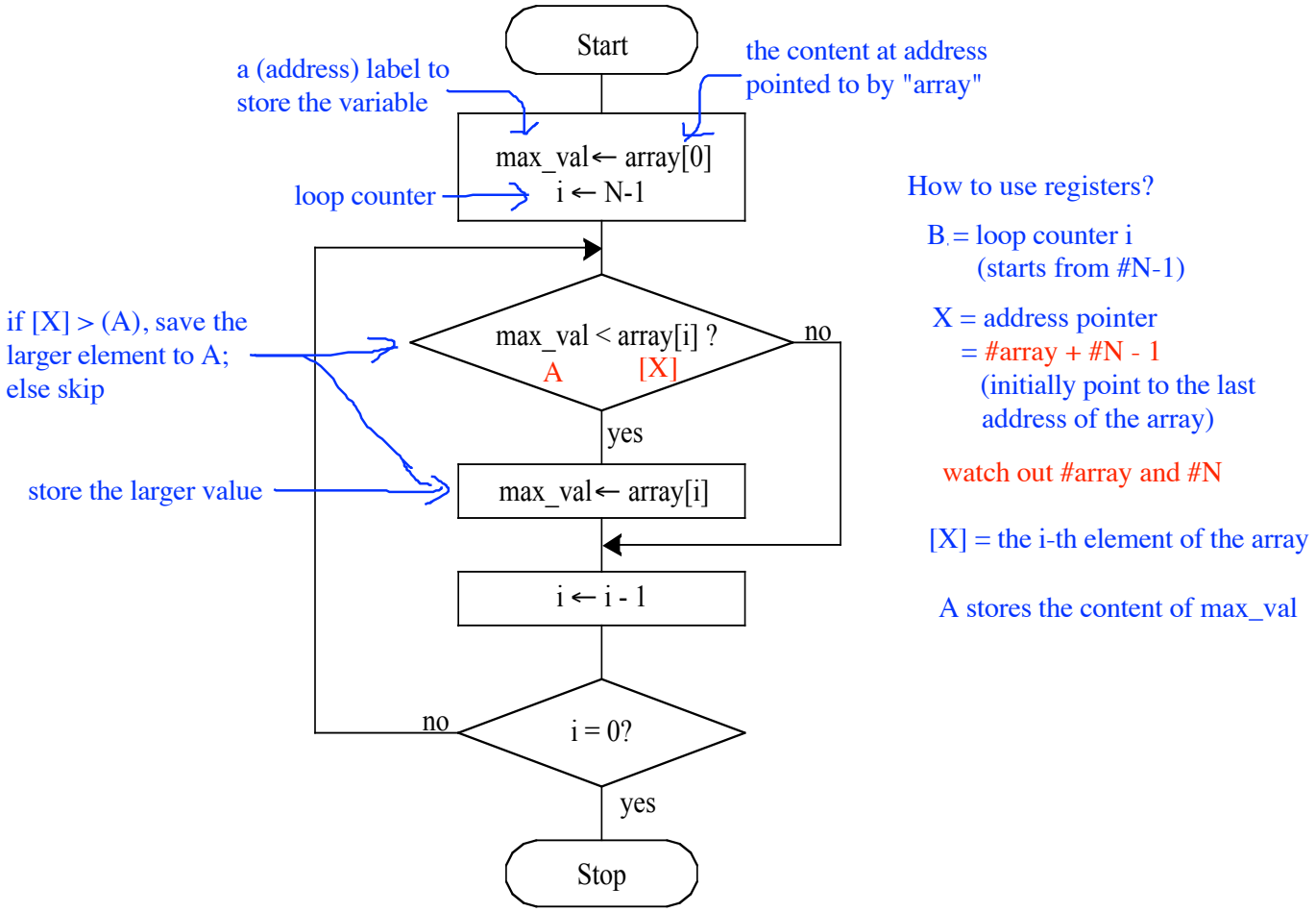
Mnemonic	Function	Equation or Operation
DBEQ cntr, rel	Decrement counter and branch if = 0 (counter = A, B, D, X, Y, or SP)	counter ← (counter) - 1 (do this first) If (counter) = 0, then branch (then this second) else continue to next instruction
DBNE cntr, rel	Decrement counter and branch if ≠ 0 (counter = A, B, D, X, Y, or SP)	counter ← (counter) - 1 If (counter) ≠ 0, then branch else continue to next instruction
IBEQ cntr, rel	Increment counter and branch if = 0 (counter = A, B, D, X, Y, or SP)	counter ← (counter) + 1 If (counter) = 0, then branch else continue to next instruction
IBNE cntr, rel	Increment counter and branch if ≠ 0 (counter = A, B, D, X, Y, or SP)	counter ← (counter) + 1 If (counter) ≠ 0, then branch else continue to next instruction
TBEQ cntr, rel	Test counter and branch if = 0 (counter = A, B, D, X, Y, or SP)	If (counter) = 0, then branch else continue to next instruction
TBNE cntr, rel	Test counter and branch if ≠ 0 (counter = A, B, D, X, Y, or SP)	If (counter) ≠ 0, then branch else continue to next instruction

need inc or dec counter first

Note. 1. **cntr** is the loop counter and can be accumulator A, B, or D and register X, Y, or SP.
2. **rel** is the relative branch offset and is usually a label

Example: Find the value of the maximum element from an array of N 8-bit elements using the **repeat** *S* **until** *C* looping construct.

Use the variable *i* as the array index and also as the loop counter. The variable *max_val* will be used to hold the array maximum.



```
N           equ   20          ; array size
            org   $1000       ; starting address of on-chip SRAM
max_val     ds.b  1           ; memory location to hold array maximum value

            org   $2000       ; starting address of program
            ldaa  array       ; load array[0] to A        [Why A=3?]
            staa  max_val     ; set array[0] as the temporary array maximum
                              ;                  [Why address $1000 contains 3?]
            ldx   #array+N-1  ; X has the address of the last element of the array
                              ;                          [N-1=19; why not 20?]
            ldab  #N-1        ; use B to set loop count to N–1   [=19]

loop        ldaa  max_val     ; A has the content of the max_val    [A←max_val]
            cmpa  0, X        ; compare max_val with array[i] = (A)-[X]
            bge   chk_end     ; no update if max_val is larger or equal
            ldaa  0, X        ; if not, move the address of array[i] to A
            staa  max_val     ; update the address of max_val

chk_end     dex               ; move the array pointer by decrement X by 1
            dbne  B, loop     ; decrement the loop count, branch if not zero yet
            swi               ;

array   db    3,1,5,6,19,41,53,28,13,42,76,14
        db    20,54,64,74,29,33,41,45

        end
```

or replace

no replace

or

0,X = offset mode = content of the address pointed to by X+0

X initially points to here

After executing one loop, max_val = 45 & B=18.

Continuously running the loop, max-val becomes 74, and finally 76.

array = a label contains the address of the first element of the array

Bit Conditional Branch Instructions

Make branch decisions based on the value of a few bits. (Application: check the status of an I/O port.)

- **BRCLR** performs bitwise logical AND on the contents of the specified memory location and the mask supplied with the instruction, then branch if all bits corresponding to 1s in the mask byte are 0s in the tested byte.

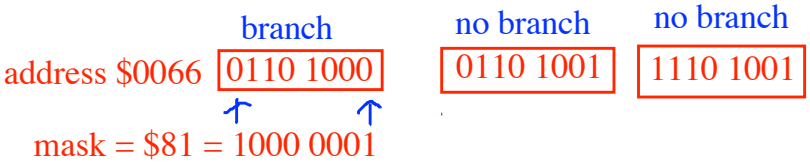
[<label>] BRCLR (opr) (msk) (rel)

opr specifies the memory location to be checked and can be specified using direct, extended, and all indexed addressing modes

msk is an 8-bit mask that specifies the bits of the memory location to be checked. The bits to be checked correspond to those bit positions that are ones in the mask.

rel is the branch offset and is specified in 8-bit relative mode.

```
here    brclr   $66, $81, here
        ldd     $70
```



the BRCLR instruction will be continued to execute if the MSB and LSB of the memory location at $0066 are both 0 because the mask $81=1000001. Otherwise, the next instruction will be executed.

- **BRSET** performs bitwise logical AND on the contents of the specified memory location and the mask supplied with the instruction, then branch if all bits corresponding to 1s in the mask byte are 1s in the tested byte.

[<label>] BRSET (opr) (msk) (rel)

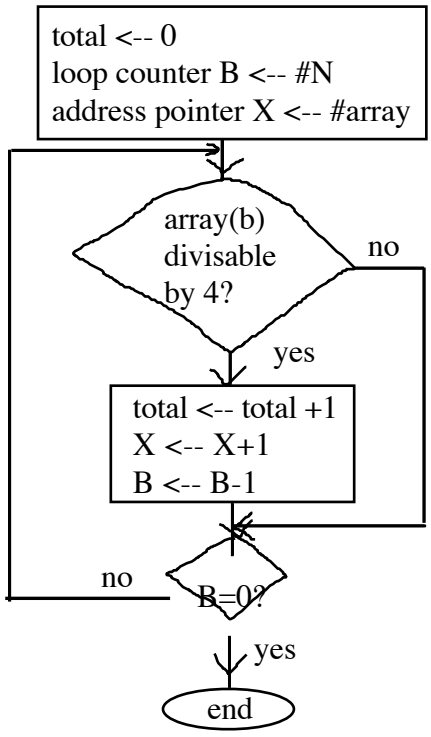
```
loop    inc     count
        …
        brset   $66, $E0, loop
```

the branch will be taken if the most significant three bits of the memory location at $0066 are all 1s because the mask $E0 = 11100000.

Example: Write a program to count the number of elements that are divisible by 4 in an array of N 8-bit numbers. Use the **repeat** S **until** C looping construct.

Method 1: number/4 and then check remainder = 0
Method 2: A number divisible by 4 would have the least significant two bits equal 00.

```
N         equ   20          ; there are 20 numbers to count
          org   $1000       ; start address of SRAM
total     ds.b  1           ; reserve one byte for the counter

          org   $2000
          clr   total       ; initialize total to 0
          ldx   #array      ; X as the array pointer;
                            ; X contains the start address of "array"
          ldab  #N          ; use B as the loop counter = 20

loop      brclr 0,X,$03,yes ; check bits 1 and 0; address = (X)+0
          bra   chkend
yes       inc   total       ; add 1 to the total
chkend    inx               ; move the array pointer to the next number
          dbne  B,loop      ; Is loop counter = N? If not, go to loop
          swi               ; if yes, stop

array     db    2,3,4,8,12,13,19,24,33,32,20,18,53,52,80,82,90,94,100,102
          end
```



Variable Initialization Instructions

1. [<label>] CLR opr ; specified memory location is initialized to zero

   opr is a memory location specified using the *extended* or *index* addressing (direct and indirect) modes. The.

2. [<label>] CLRA ; Accumulator A is cleared to 0.

3. [<label>] CLRB ; Accumulator B is cleared to 0.

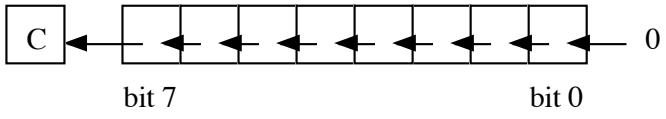
Shift & Rotate Instructions

- Shift and rotate instructions are useful for bit field manipulation.
- Used to speed up the integer multiply and divide operation if one of the operands is a power of two.
- Shift/rotate the operand by one bit.
- Apply to a memory location, accumulators A, B, and D.
- A memory operand must be specified using the extended addressing mode or (direct or indirect) indexed addressing modes.

1. Three 8-bit ***arithmetic shift left*** instructions and one 16-bit arithmetic shift left instruction:

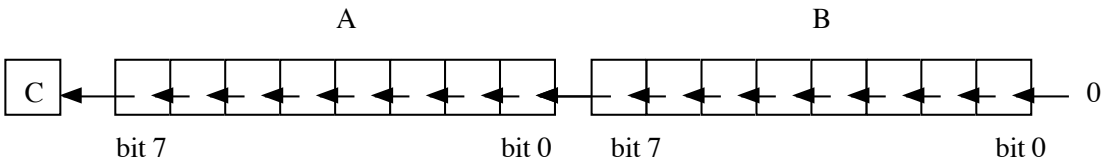
ASL <opr> ; memory location ***opr*** is shifted left one bit
ASLA ; accumulator A is shifted left one bit
ASLB ; accumulator B is shifted left one bit

same as logical shift left (LSL)
= x2 each shift
may cause overflow



affect CCR bits

ASLD ; accumulator D is shifted left one bit

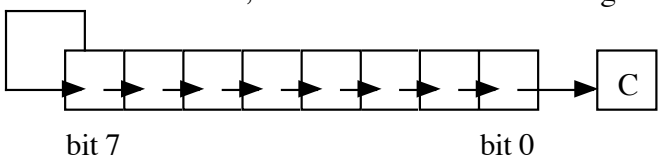


2. Three 8-bit ***arithmetic shift right*** instructions:

ASR <opr> ; memory location ***opr*** is shifted right one bit
ASRA ; accumulator A is shifted right one bit
ASRB ; accumulator B is shifted right one bit

= /2 each shift

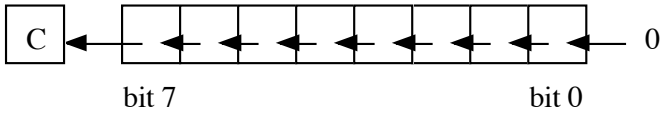
affect CCR bits



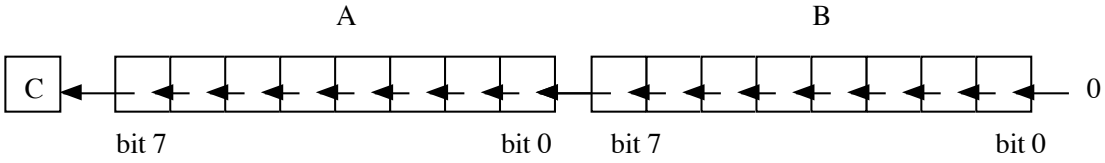
keep sign bit unchanged

3. Three 8-bit ***logical shift left*** instructions and one 16-bit logical shift left instruction:

LSL <opr> ; memory location ***opr*** is shifted left one bit
LSLA ; accumulator A is shifted left one bit
LSLB ; accumulator B is shifted left one bit

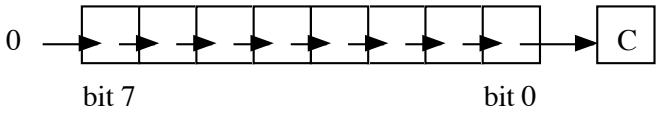


LSLD ; accumulator D is shifted left one bit

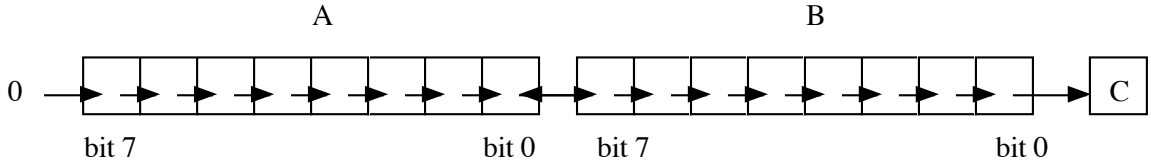


4. Three 8-bit ***logical shift right*** instructions and on 16-bit logical shift right instruction.

LSR <opr> ; memory location ***opr*** is shifted right one bit
LSRA ; accumulator A is shifted right one bit
LSRB ; accumulator B is shifted right one bit

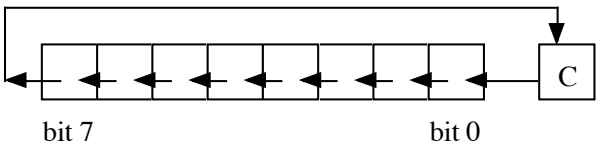


LSRD ; accumulator D is shifted right one bit



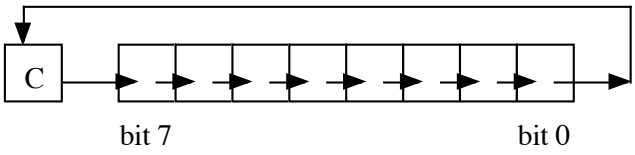
5. Three 9-bit ***rotate left*** instructions:

ROL <opr> ; memory location ***opr*** is rotated left one bit
ROLA ; accumulator A is rotated left one bit
ROLB ; accumulator B is rotated left one bit

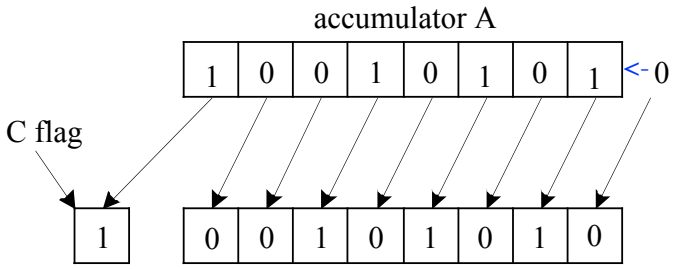


6. Three 9-bit ***rotate right*** instructions:

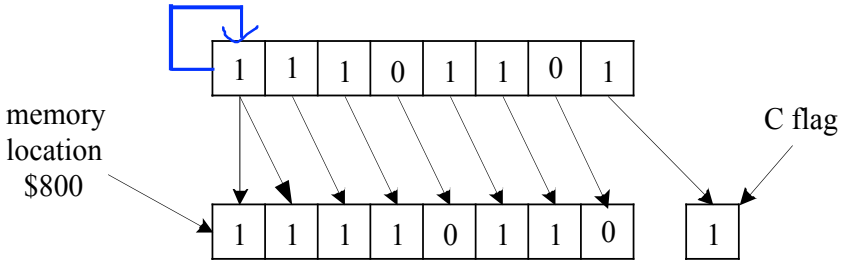
ROR <opr> ; memory location ***opr*** is rotated right one bit
RORA ; accumulator A is rotated right one bit
RORB ; accumulator B is rotated right one bit



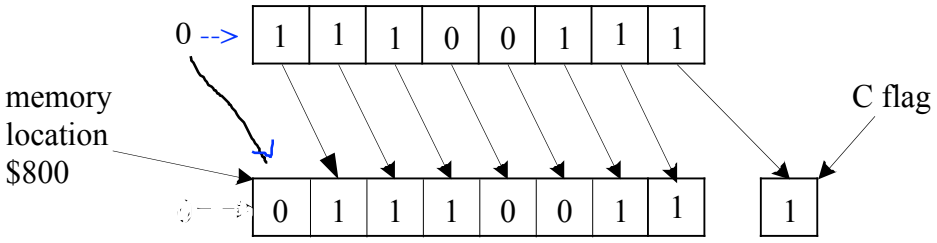
Example: Suppose that [A] = $95 and C = 1. Compute the new values of A and C after the execution of the instruction ASLA.



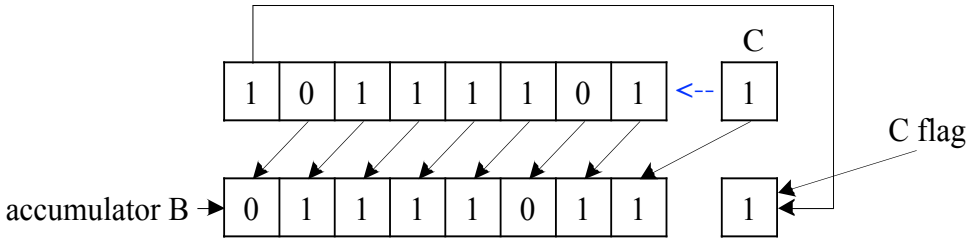
Example: Suppose that m[$800] = $ED and C = 0. Compute the new values of m[$800] and the C flag after the execution of the instruction ASR $800.



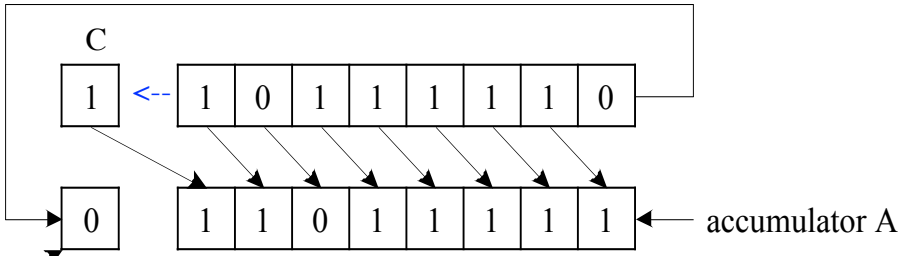
Example: Suppose that m[$800] = $E7 and C = 1. Compute the new contents of m[$800] and the C flag after the execution of the instruction LSR $800.



Example: Suppose that [B] = $BD and C = 1. Compute the new values of B and the C flag after the execution of the instruction ROLB.



Example: Suppose that [A] = $BE and C = 1. Compute the new values of A and C after the execution of the instruction RORA.



Example: Write a program to count the number of zeros in the 16-bit number stored at $1000-$1001 and save the result in $1005.

Step 1: Initialize the loop counter to 16 and the zero counter to 0.
Step 2: Load the 16-bit number into D.
Step 3: Shift D to the right one bit → least significance bit to the C flag in CCR
Step 4: If the bit shifted out into C is a 0, then increment the zero counter by 1.
Step 5: Decrement loop counter by 1.
Step 6: If loop counter is 0, the stop. Otherwise, go to step 3.

```
            org     $1000       ; starting address of the 16-bit number
            db      $23,$55     ; define the 16-bit number as $2355 (2 bytes)
            org     $1005       ; starting address of SRAM
zero_cnt    rmb     1           ; reserve 1 byte for zero counter   [Why not use A?]
lp_cnt      rmb     1           ; reserve 1 byte for loop counter   [Why not use B?]

            org     $2000
            clr     zero_cnt    ; initialize the zero counter to 0
step 1      ldaa    #16         ; initialize the loop counter to 16
            staa    lp_cnt      ;

step 2      ldd     $1000       ; place the number in D (2 bytes)  ← D=A:B is used

step 3 loop lsrd                ; shift the LSB of D to the C flag

            bcs     chkend      ; is the C flag a 0? If not, jump to chkend
step 4      inc     zero_cnt    ; if yes, increment zero counter by 1

step 5 chkend dec   lp_cnt      ; decrement loop counter by 1

step 6      bne     loop        ; if loop counter is 0, go back to the loop

            swi                 ;
            end
```

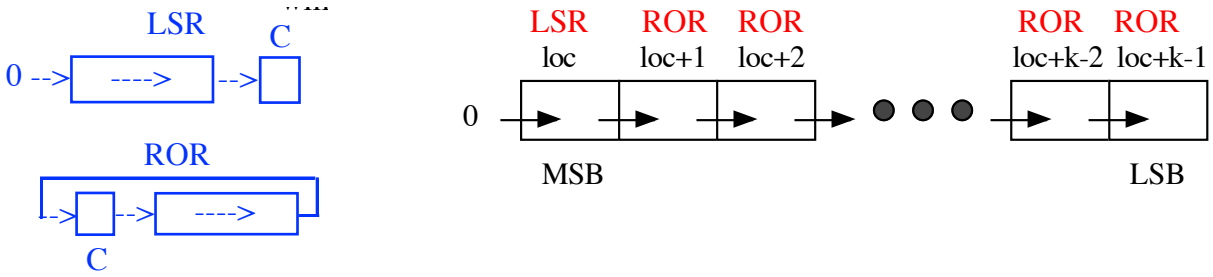
BCS = branch if carry is set (C flag = 1)
BNE = branch if not equal to zero (Z flag = 0).

Shift a Multibyte Number

- Shift a number larger than 16 bits.
- Suppose there is a k-byte number that is stored at *loc* to *loc+k-1*. (MSB is in *loc* and LSB is in *loc+k-1*.)

Method for logical shifting right (by one bit)

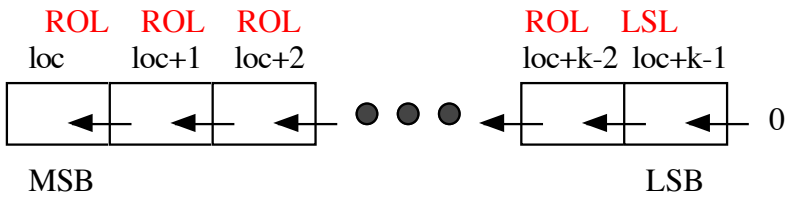
- Bit 7 of each byte will receive bit 0 of the byte on its immediate left with the exception of the most significant byte, which will receive a 0.
- Each byte will be shifted to the right by one bit. Bit 0 of the least significant byte will be lost.



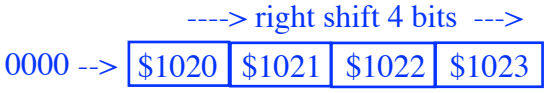
Step 1: Shift the byte at *loc* to the right one place. (Use LSR <opr>)
Step 2: Rotate the byte at *loc+1* to the right one place. (Use ROR <opr>)
Step 3: Repeat Step 2 for the remaining bytes.

Method for logical shifting left (by one bit)

- Bit 0 of each byte will receive bit 7 of the byte on its immediate right with the exception of the least significant byte, which will receive a 0.
- Each byte will be shifted to the left by one bit. Bit 7 of the most significant byte will be lost.



Step 1: Shift the byte at *loc+k-1* to the left one place. (Use LSL <opr>)
Step 2: Rotate the byte at *loc+K-2* to the left one place. (Use ROL <opr>)
Step 3: Repeat Step 2 for the remaining bytes.



Example: Write a program to shift the 32-bit number stored at $1020-$1023 to the right four places. (The most significant to the least significant bytes are stored at $1020~$1023, respectively.)

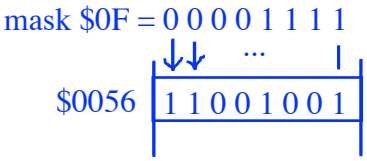
```
        ldab    #4          ; set up B as the loop counter to shift 4 times
        ldx     #$1020      ; use X as the pointer to the leftmost byte

again   lsr     0,X         ; shift right the left most byte
        ror     1,X         ; rotate right
        ror     2,X         ;
        ror     3,X         ; the rightmost byte

        dbne    B, again    ; decrement B by 1, branch if B is not zero
```

Boolean Logic Instruction

- BITWISE logic operations can be used to change a few I/O port pins easily.
- The operand ***opr*** can be specified using all except the relative addressing modes.
- **AND** instruction to ***clear*** one or a few bits.
- **OR** instruction to ***set*** one or a few bits.
- **XOR** instruction to ***toggle*** one or a few bits.
- **COMA** and **COMB** that perform ***1's complement*** can be used if all the port pins needed to be toggled.

<opr> acts like a mask

Mnemonic	Function	Operation
ANDA <opr>	AND A with memory	A ← (A) • (M)
ANDB <opr>	AND B with memory	B ← (B) • (M)
ANDCC <opr>	AND CCR with memory (clear CCR bits)	CCR ← (CCR) • (M)
EORA <opr>	Exclusive OR A with memroy	A ← (A) ⊕ (M)
EORB <opr>	Exclusive OR B with memory	B ← (B) ⊕ (M)
ORAA <opr>	OR A with memory	A ← (A) + (M)
ORAB <opr>	OR B with memory	B ← (B) + (M)
ORCC <opr>	OR CCR with memory	CCR ← (CCR) + (M)
CLC	Clear C bit in CCR	C ← 0
CLI	Clear I bit in CCR	I ← 0
CLV	Clear V bit in CCR	V ← 0
COM <opr>	One's complement memory	M ← $FF - (M)
COMA	One's complement A	A ← $FF - (A)
COMB	One's complement B	B ← $FF - (B)
NEG <opr>	Two's complement memory	M ← $00 - (M)
NEGA	Two's complement A	A ← $00 - (A)
NEGB	Two's complement B	B ← $00 - (B)



Examples:

```
LDAA    $56     ; this instruction sequence clears the
ANDA    #$0F    ; upper four pins of the I/O port located at $56
STAA    $56     ; because $0F = 00001111
                ; $0056 contains 0000 1001

LDAA    $56     ; this instruction sequence sets the
ORAA    #$0F    ; bit 0 of the I/O port located at $56
STAA    $56     ; because $01 = 00000001
                ; $0056 contains 1100 1111

LDAA    $56     ; this instruction sequence toggle the
EORA    #$0F    ; lower four bits of the I/O port located at $56
STAA    $56     ; $0056 contains 1100 0110
```

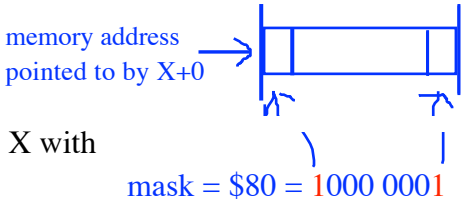
Bit Test and Manipulation Instructions

- Use a mask value to test or change the value of individual bits in an accumulator or in a memory location.
- **BITA** and **BITB** provide a convenient means of testing bits without altering the value of either operand.

Mnemonic	Function	Operation
BCLR <opr>[2], msk8	Clear bits in memory	M ← (M)x($\overline{mm}$)
BITA <opr>[1]	Bit test A	(A)x(M)
BITB <opr>[1]	Bit test B	(B)x(M)
BSET <opr>[2], msk8	Set bits in memory	M ← (M)x(mm)

Note:
1. <opr> can be specified by using all except relative addressing modes for BITA and BITB
2. <opr> can specified by using direct, extended, and indexed (exclude indirect) addressing modes.
3. msk8 is an 8-bit value.

BCLR 0,X, $81 ; 0,X = offset addressing mode



clears the MSB and LSB of the memory location pointed to by X with 0 offset.

```
BITA    #$44            ; mask = $44 = 0100 0100
```

tests bit 6 and bit 2 of A, and updates the Z and N flags of CCR accordingly. The V flag in CCR is cleared.

```
        BITB   #$22          ; mask $22 = 0010 0010
```
tests bit 5 and bit 1 of B, and updates the Z and N flags of CCR accordingly. The V flag in CCR is cleared.

```
        BSET  0,Y,$33        ; mask $33 = 0011 0011
```
sets bits 5, 4, 1, and 0 of the memory location pointed to by Y with 0 offset.

Program Execution Time

- HCS12 uses the **ECLK** signal as a timing reference (clock).
- Frequency of ECLK (24 MHz) is ***half*** of that of the crystal oscillator (48 MHz).
- Execution times of instructions are measured in the number of ***E cycles***.
- Many applications require the generation of time delays.
- Program loops are often used to create some amount of (not so accurate) delay:
  1. Select a sequence of instructions that takes a certain amount of time to execute
  2. Repeat the selected instruction sequence for an appropriate number of times

Example: the following instruction sequence takes 40 E cycles to execute.

The HCS12 has a E-clock of 24 MHz. This corresponds to 1E cycle = 1/24,000,000 sec. The following instruction sequence will take about 40/24,000,000 = 1.6667 μs to execute.

```
loop    psha                ; 2 E cycles
        pula                ; 3 E cycles
        psha
        pula
        psha
        pula
        psha                ; 7 pairs of psha and pula gives 35 E cycles
        pula
        psha                ; these instructions do nothing useful;
        pula                ; just set up times in order to genereate time delay
        psha
        pula
        psha
        pula
        ---------------------------------------------------------------
        nop                 ; 1 E cycle
        nop                 ; 1 E cycle
        dbne   X,loop       ; 3 E cycles
```

Example: A delay of 100 ms can be created by repeating the previous loop 60,000 times because 100 ms / [40 / 24,000,000] = 100 ms / 1.6667 μs = 60,000).

```
        ldx     #60000      ; 2 E cycles
loop    psha                ; 2 E cycles
        pula                ; 3 E cycles
        psha
        pula
        psha
        pula
        psha                ; 7 pairs of psha and pula gives 35 E cycles
        pula
        psha
        pula
        psha
        pula
        psha
        pula
        nop                 ; 1 E cycle
        nop                 ; 1 E cycle
        dbne    X,loop      ; 3 E cycles
```

Example: By repeating the previous instruction sequence 100 times, we can create a delay of 10 seconds. This is done by creating a 2-layer loop.

```
        ldab    #100        ; outer loop  (1 E cycle overhead)
out_loop
        ldx     #60000      ; inner loop  (2 E cycles overhead)
in_loop
        psha                ; 2 E cycles
        pula                ; 3 E cycles
        psha                ; …
        pula                ; …
        …
        …
        psha                ; 7 pairs of psha and pula gives 35 E cycles
        pula
        psha
        pula
        nop                 ; 1 E cycle
        nop                 ; 1 E cycle
        dbne    X,in_loop   ; 3 E cycles
        dbne    B,out_loop  ; 3 E cycles  (overhead)
```

The time delay created by using loops is not accurate. Some overhead is required to set up the loop count. For example, the one-layer loop has a 2-E-cycle overhead while the two-layer loop has

Overhead = 1 E cycle (caused by "ldab #100")
+ 100 x 2 E cycles (caused by out_loop "ldx #60000")
+ 100 x 3 E cycles (caused by "dbne B,out_loop")
= 501 E cycles
= 20.875 us (at 24 MHz E-clock) per 10sec delay

This overhead can be reduced by placing a larger value in X and a smaller value in B.

If higher accuracy is required, you should use one of the timer functions in the HCS12 to create the desired time delay.

End of Chapter 2