ECET 310-001
Chapter 2, Part 2 of 3

W. Barnes, 9/2006, rev’d. 10/07

Ref. Huang, Han-Way, ***The HCS12/9S12: An Introduction to Software and Hardware Interfacing***, Thomson/Delmar.

In This Set of Slides:

1. **Introduction to Program Loops**
2. **3 Types of finite loops**
3. **CCR & Branching**
4. **Various Types of Branching**

Introduction to Program Loops

- Why? Efficiency and flexibility in programming
  - When doing a task three or more times, instead of rewriting the code, a loop is used
  - In example 2.13, the following code is used 4 times:

```
xgdx
ldx     #10
idiv
addb    #$30
stab    2,Y
```

These five instructions can be put into a loop, just making sure that the offset in the stab instruction decrements each time so that it goes from 4 to 0.

Program Loops Continued

<u>Infinite</u> (or endless or forever) **Loop:**

do *statement S* **forever**

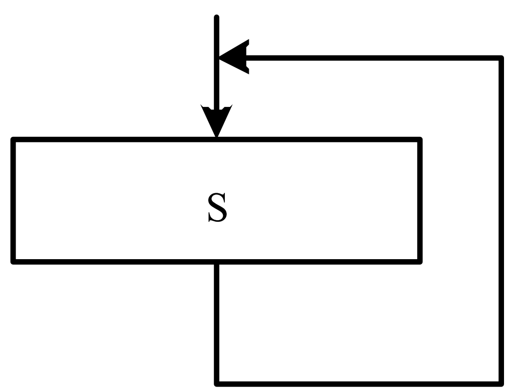


Figure 2.4 An infinite loop

Program Loops Continued

3 Finite Loop types:

1. **For** i = *n1* **to** *n2* **do** *statement S* or
   For i = *n2* **downto** *n1* **do** *statement S*
2. **While** *C* **do** *statement S*
3. **Repeat** *statement S* **until** *C*

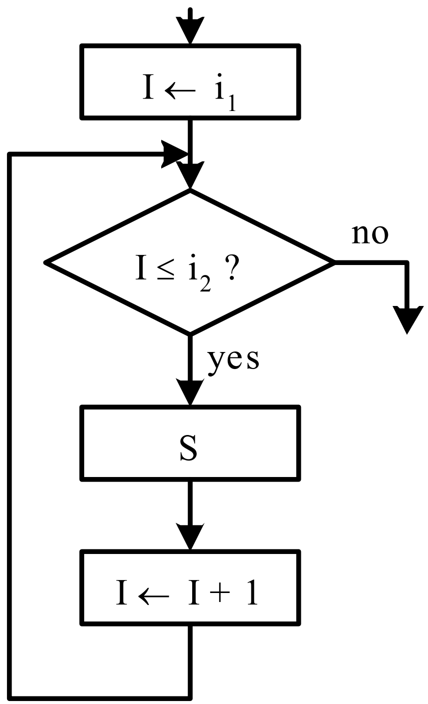
- **How is the exit from a loop implemented?** With conditional branch instructions, which depend on the CCR register flags for a decision.

Flowcharts of the 3 Types of finite loops

(1) Using an index (counter)

For i = *n1* **to** *n2* **do** *statement S* or

For i = *n2* **downto** *n1* **do** *statement S*



(a) For I = i_1 to i_2 DO S (b) For I = i_2 downto i_1 DO S

Figure 2.5 For looping construct

Flowcharts of the 3 Types of Finite Loops cont'd

(2) **While** *C* **do** *statement S*

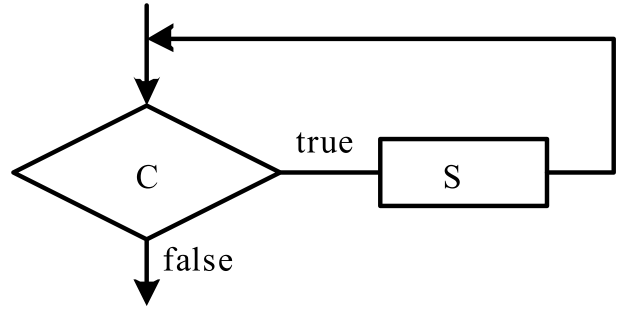


Figure 2.6 The While ... Do looping construct

(3) **Repeat** *statement S* **until** *C*

Q. What's wrong with the flow chart?

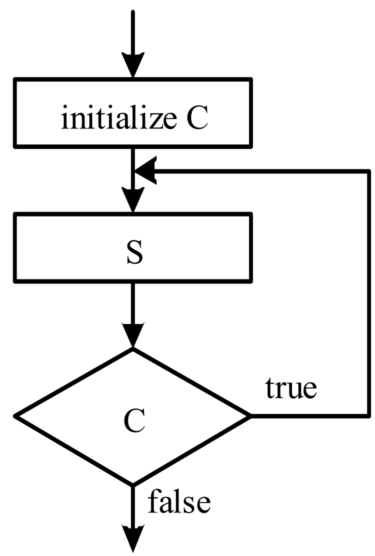


Figure 2.7 The Repeat ... Until looping construct

CCR & Branching

7	6	5	4	3	2	1	0
S	X	H	I	N	Z	V	C

Figure 2.8 Condition code register

- Four types of branch instructions
  - Unconditional branch: always executes
  - Simple branches: branch is taken based on a specific bit of CCR
  - Unsigned branches: branches are taken when a comparison or test of unsigned numbers results in a specific combination of CCR bits
    - Associate higher, lower, and same with <u>**unsigned**</u> numbers
  - Signed branches: branches are taken when a comparison or test of signed quantities are in a specific combination of CCR bits
    - Associate greater, less, and equal with <u>**signed**</u> numbers
- Two categories of branches (See next two slides)
  - Short branches (most used): in the range of -128 ~ +127 bytes
  - Long branches: in the range of 64KB (note L at beginning of each mnemonic)

Table 2.2 Summary of short branch instructions

Mnemonic	Function	Equation or Operation
Unary Branches		
BRA	Branch always	$1 = 1$
BRN	Branch never	$1 = 0$
Simple Branches		
Mnemonic	Function	Equation or Operation
BCC	Branch if carry clear	$C = 0$
BCS	Branch if carry set	$C = 1$
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		
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	$N \oplus V = 0$
BGT	Branch if greater than	$Z + (N \oplus V) = 0$
BLE	Branch if less than or equal	$Z + (N \oplus V) = 1$
BLT	Branch if less than	$N \oplus V = 1$

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	$N \oplus V = 1$

Branching continued

Compare and Test Instructions

– None of these instructions actually changes any values but they affect the flags and are then followed by conditional branch instructions

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

Branching Continued

- Loop Primitive Instructions
  These short branch instructions decrement or increment a loop counter to determine if the looping should continue.

Table 2.5 Summary of loop primitive instructions

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 If (counter) = 0, then branch 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

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

Branching & Loops cont'd

Example 2.14 Write a program to add an array of N 8-bit numbers and store the sum at memory locations $1500~$1501. Use the **For** i = n1 **downto** n2 **do** looping construct.

```
;webex2_14a, adds N  numbers and stores sum
;accesses and stores sum each time in the loop
;includes dbne and auto-incrementing of index
N       equ     3
        org     $1500
array   db      1,2,255    ;test data
sum     rmw     1
        org     $2000
        ldx     #array
        ldaa    #N
        movw    #0,sum     ;initially clear sum
loop    ldy     sum                    ;get sum
        ldab    1,x+                   ;     get x(i) and point to x(i+1),
        aby                ;        and add
        sty     sum
        dbne    a,loop     ;update counter and check if done
        swi
        end
```

<u>Exercise</u>: (1) Draw a flow chart for this solution and compare with figure 2.9 in text.
(2) label program and flow chart into sections: initialization, loop test, update counter, loop statements

Branching (& loops) continued

Example 2.14 <u>Second approach, not accessing sum each time (since we see Y undisturbed)</u>

```
;webex2_14b, adds N  numbers and stores sum
;includes dbne and auto-incrementing index
;does not rely on accessing sum from memory

N       equ     3
        org     $1500
array   db      1,2,255   ;test data
sum     rmw     1

        org     $2000
        ldx     #array
        ldaa    #N
        ldy     #0        ;prepare y to accumulate sum
loop    ldab    1,x+      ;get x(i)
        aby               ;    and add to sum
        dbne    a,loop
        sty     sum       ;no need to pre-clear
        swi
        end
```

Branching (& Loops) Continued

Example 2.15 Write a program to find the maximum element from an array of N 8-bit elements using the **repeat S until C** looping construct.

Exercise: label program and flow chart into sections: initialization, loop test, update counter, loop statements- ALSO, try signed numbers [note use of BGE (see table 2.2)in program-)]

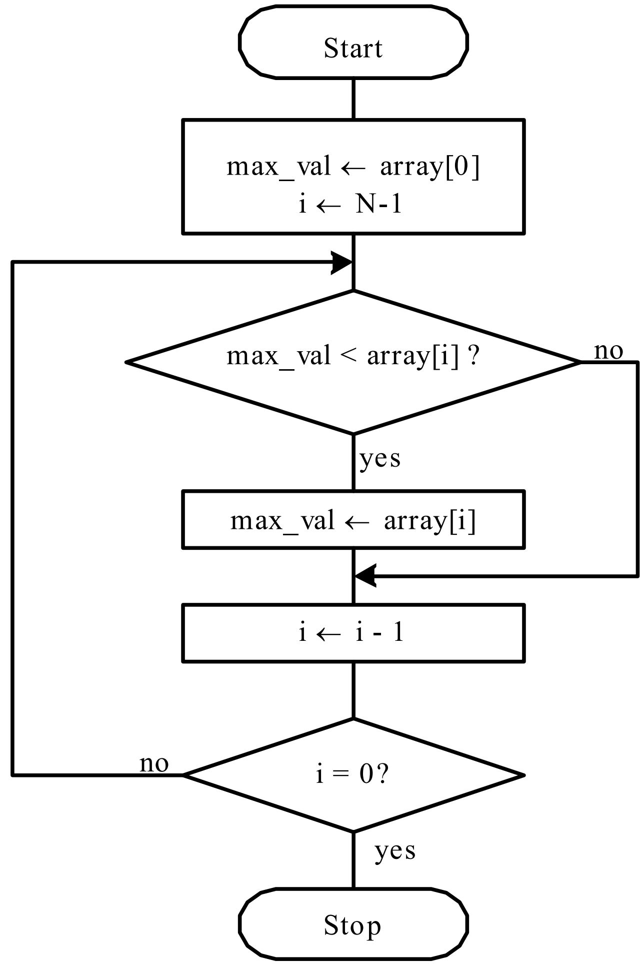


Figure 2.10 Logic flow of example 2.15

Branching & Loops continued

Ex. 2.15 code

```
1.  N          equ    10
2.             org    $1500
3.  max_val    rmb    1
4.             org    $2000
5.             ldaa   array         ; set array[0] as the initial max
6.             staa   max_val       ;       in this case max_val becomes 1
7.             ldx    #array+N-1  ; start from the end of the array
8.             ldab   #N-1          ; set loop count for N – 1 comparisons
9.  loop       ldaa   max_val
10.            cmpa   0,x           ;compare current max_val w/array[ i ]
11.            bge    chk_end       ;if > or = skip to chk_end
12.            ldaa   0,x
13.            staa   max_val
14. chk_end    dex
15.            dbne   b,loop      ; finish all the comparisons yet?
16. forever    bra    forever     ; swi better for debugging
17. array      db     1,3,5,6,19
18.            db     20,54,64,74,29
19.            end
```

Questions:

(1) What are the advantages and disadvantages of storing data at the end of the program?

(2) In what applications would line #16 be useful and not useful?

Branching (& loops) continued

In class exercises (be sure to consider what addressing modes to use for the following):

1. Write a program segment to move 5 bytes starting at $1500 to locations starting at $1600

2. Repeat (1) for words

3. Draw flowcharts.