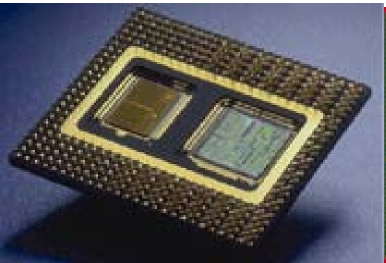
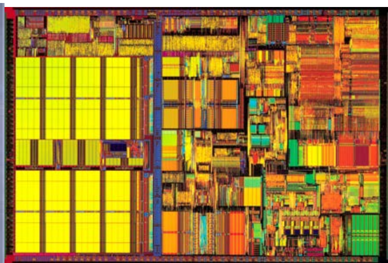
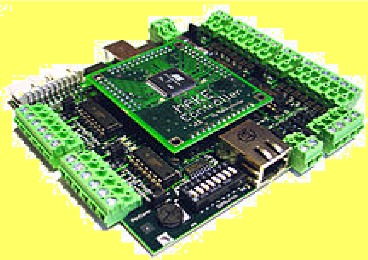
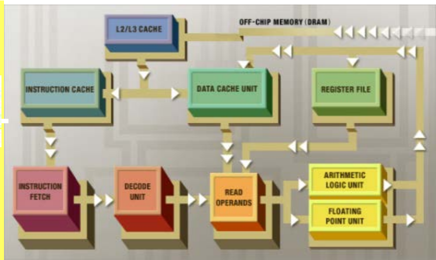
Lecture 6

Assembly Programming: Branch & Iteration

Bryan Ackland
Department of Electrical and Computer Engineering
Stevens Institute of Technology
Hoboken, NJ 07030

Adapted from HCS12/9S12 An Introduction to Software and Hardware Interfacing   Han-Way Huang, 2010

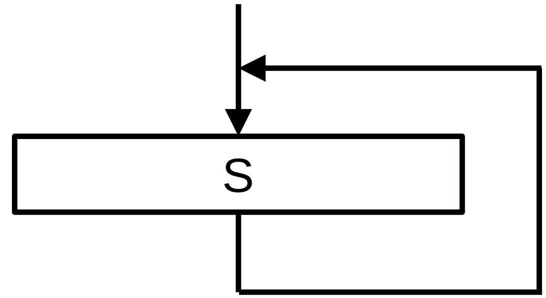



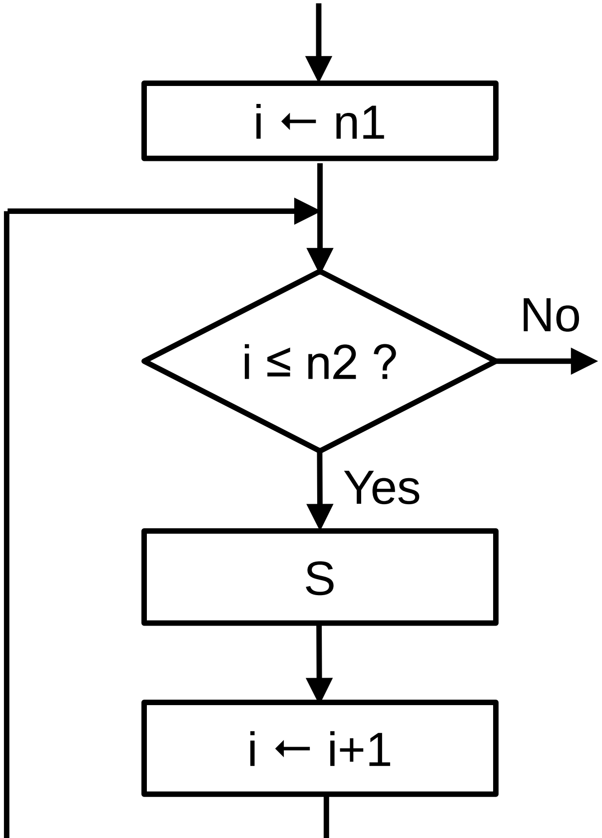
Iteration

- Most interesting algorithms involve some form of iteration
- Iteration allows us to work through large bit sequences, data structures, problem sets, data structures etc., repetitively using the same set of operations on different data in such a manner that we progress towards a final result.
- Iteration means we don’t have to explicitly code every operation that occurs in a program
- Iteration is implemented in assembly code (also in high level languages) as various kinds of loops
- Four basic loop constructs:

Program Loops

do operation S forever

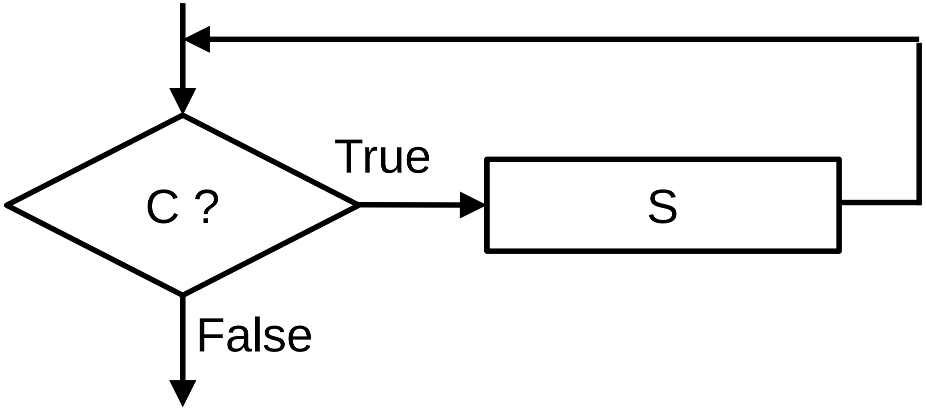


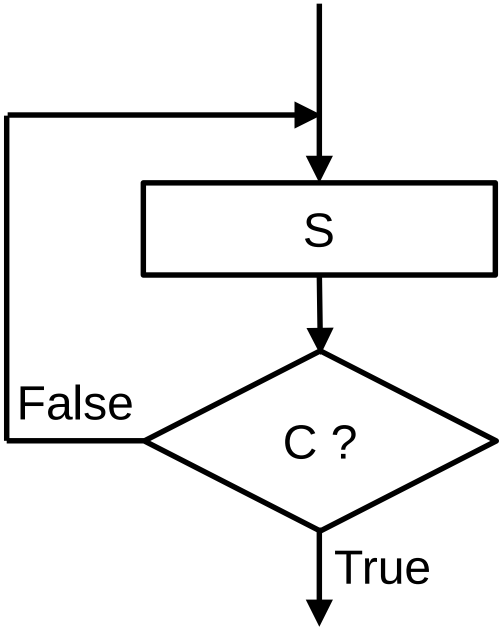


for i = n1 to n2 do operation S

Program Loops

while C do statement S

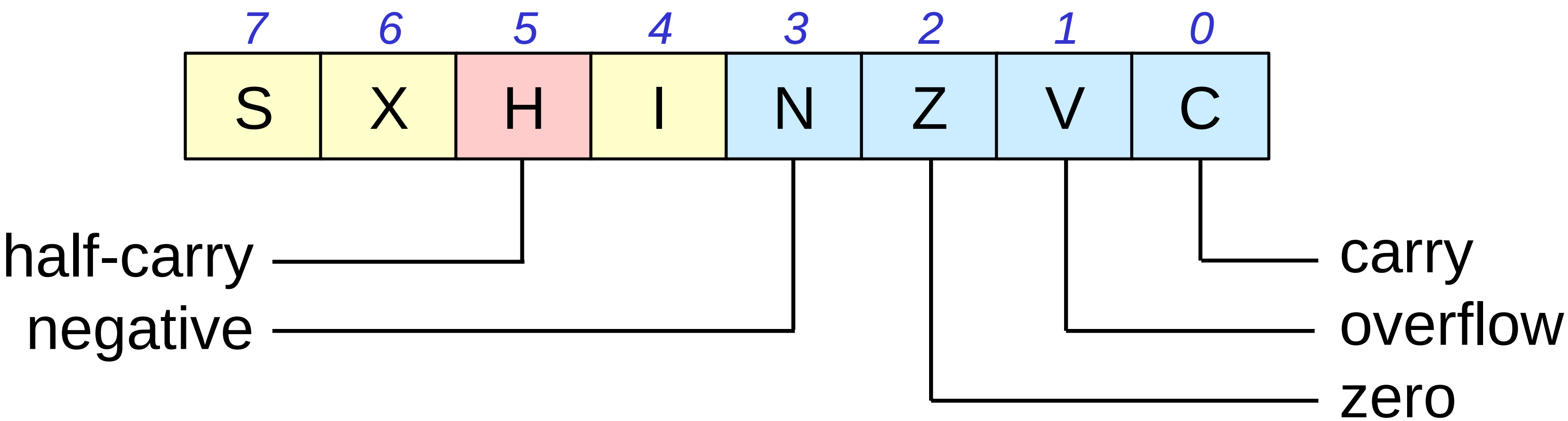




repeat statement S until C

Condition Code Register

- Program loops are implemented using conditional branch instructions
- Execution of these depends on contents of CCR



- CCR is updated whenever an arithmetic or compare instruction is executed

Condition Codes (Flags)

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

- **C**arry flag is set (reset) whenever an arithmetic or compare instruction causes (does not cause) a carry-out from MSBit of the result
- **Z**ero flag is set (reset) whenever an arithmetic or compare instruction generates a zero (non-zero) result
- **N**egative flag is set (reset) whenever an arithmetic or compare instruction generates a negative (positive or zero) result
- o**V**erflow flag is set (reset) whenever an arithmetic or compare instruction generates an incorrect or out-of-range (correct) result
  - assumes the operands and result are interpreted as two's complement signed quantities.

What’s the Difference between Carry and Overflow?

Unsigned *(range: 0 to15)*	Signed *(range: -8 to +7)*	CCR Result
1 0 1 0 (10) + 1 1 1 1 (15) 1 1 0 0 1 (9)	1 0 1 0 (-6) + 1 1 1 1 (-1) 1 1 0 0 1 (-7)	C = 1 V = 0
0 1 1 1 (7) + 0 0 1 1 (3) 0 1 0 1 0 (10)	0 1 1 1 (7) + 0 0 1 1 (3) 0 1 0 1 0 (-6)	C = 0 V = 1
1 1 0 0 (12) + 1 0 1 1 (11) 1 0 1 1 1 (7)	1 1 0 0 (-4) + 1 0 1 1 (-5) 1 0 1 1 1 (7)	C = 1 V = 1

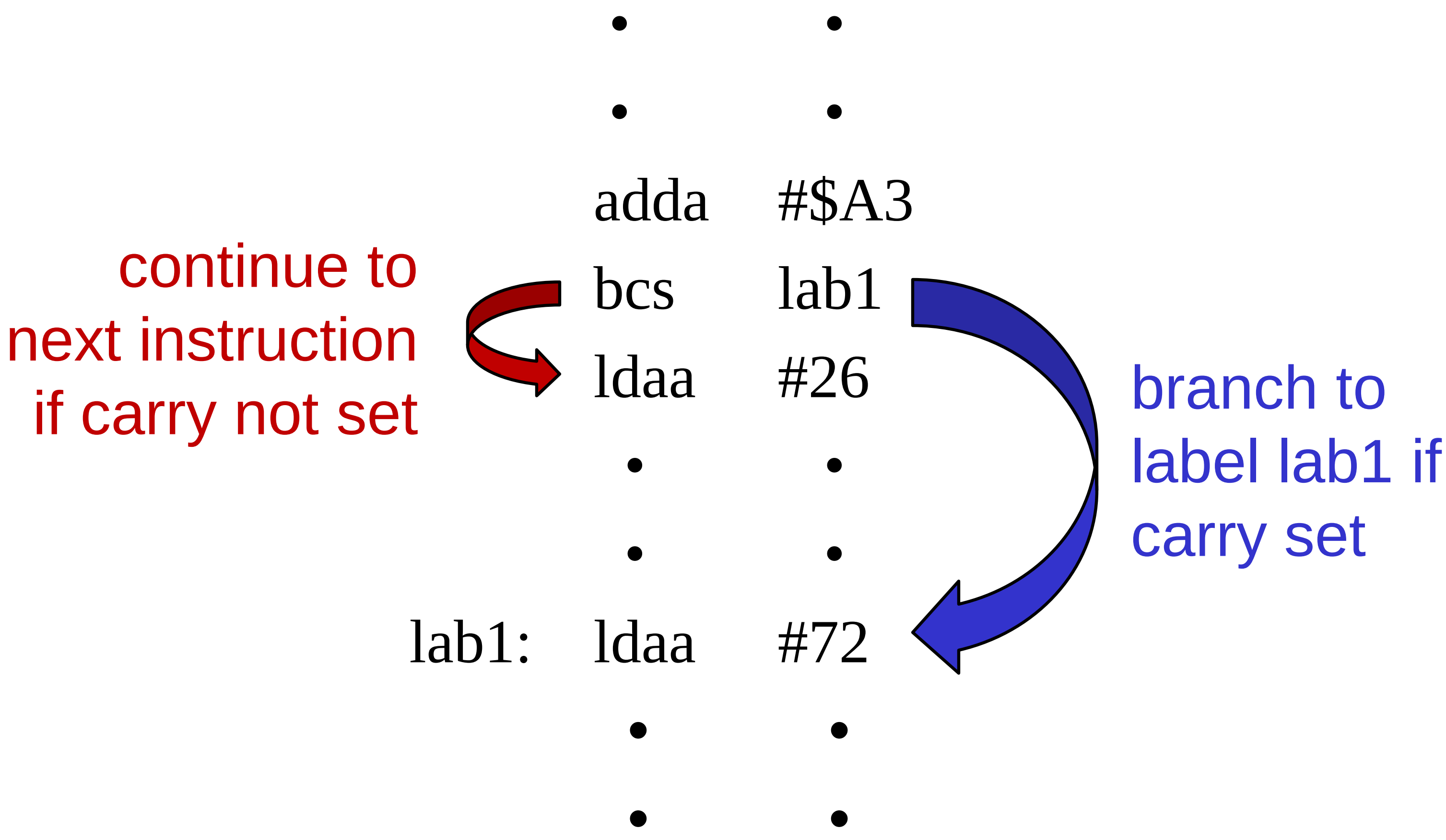
Condition Code Example

	N	Z	V	C
clra	0	1	0	0
adda #$20	0	0	0	0
suba #$30	1	0	0	1
tsta	1	0	0	0
cmpa #$F0	0	1	0	0
adda #$70	0	0	0	1
adda #$50	1	0	1	0

Branch Instructions

- Branch instructions (conditionally) modify the program counter (PC) so that the next instruction fetched may not be the instruction immediately following the current instruction
  - program will *either* branch to specified target address *or* continue to the next sequential instruction depending on the specified condition
- Branch instruction specifies a signed offset (in bytes)
- This offset is (conditionally) added to the PC to form the address of the instruction we are to branch to
  - positive offset branches forward
  - negative offset branches backward
- Programmer almost never specifies a numerical offset
  - We use labels and let the assembler work out the correct offset

Branch to Label



Branch Instructions

Four types of branch instructions:

- **Unconditional:** always execute
- **Simple:** branch depends on test of specific CCR bit
- **Unsigned:** branch taken after a comparison or test of unsigned numbers (uses combination of CCR bits)
- **Signed:** branch taken after a comparison or test of signed numbers (uses combination of CCR bits)

Two categories of branch instructions:

Short branches: 8-bit signed offset in range of -128 to +127 bytes

Long branches: 16-bit signed offset

Short Branch Instructions (1)

- Unconditional Branches:

Mnemonic	Function	Branch Test
bra	Branch always	True
brn	Branch never	False

- **bra** is an unconditional branch (*jump* or *goto*)

- **brn** is effectively a *nop* (no-operation)
  - just continue to next instruction
  - can be used as debugging instruction to temporarily replace a conditional branch instruction

Short Branch Instructions (2)

- Simple Branches
  - depend on value of a single condition code

Mnemonic	Function	Branch Test
bcc	Branch if carry clear	C=0
bcs	Branch if carry set	C=1
beq	Branch if equal*	Z=1
bne	Branch if not equal*	Z=0
bmi	Branch if minus	N=1
bpl	Branch if plus (or zero)	N=0
bvc	Branch if overflow clear	V=0
bvs	Branch if overflow set	V=1

* These instructions are called “branch if equal” and “branch if not equal” because they usually follow a compare instruction in which one operand is subtracted from another

Simple Branch Example

```
        •       •
        •       •
        ldaa    #8
abc:    ldx     2, Y+
        •       •
        •       •
        suba    #1
xyz:    bne     abc
        •       •
        •       •
```

what is this code doing?

Short Branch Instructions (3)

- Unsigned Branches:

'+' means logical OR

Mnemonic	Function	Branch Test
bhi	Branch if higher	C + Z = 0
bhs	Branch if higher or same	C = 0
blo	Branch if lower	C = 1
bls	Branch of lower or same	C + Z = 1

- These branches assume two unsigned quantities have already been subtracted (or compared) using one of:
  - SBCA, SBCB, SUBA, SUBB, SUBD, CMPA, CMPB, CPD, CPS, CPX or CPY. For example:

```
subb  #$27          cmpb  #$27
bhi   abc     or    bhi   abc
```

  - both will branch to label abc if contents of acc B are higher than (greater than in an unsigned sense) the number $27
  - subb will change contents of acc B, cmpb will not.

Short Branch Instructions (4)

- Signed Branches

'⊕' means logical EXOR

Mnemonic	Function	Branch Test
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	$N \oplus V = 1$

- These branches assume two signed quantities have already been compared or subtracted using one of:
  - CMPA, CMPB, CPD, CPS, CPX, CPY, SBCA, SBCB, SUBA, SUBB, or SUBD. For example:

```
subb    #$27
bgt     abc
```

or

```
cmpb    #$27
bgt     abc
```

  - will branch to label abc if contents of acc B are greater than (in a signed sense) the number $27
  - subb will change contents of acc B, cmpb will not.

Long Branch Instructions

- The short branch instructions described so far, can only branch to a location (-128 to +127) bytes relative to current value of PC.
  - used for local branches and small program loops
- To branch to a location outside of this range, need to use long branch instructions
- To make a long branch, simply add letter ‘l’ to front of mnemonic
  - e.g. bgt (short) becomes lbgt (long)
- Assembler will give an error if you try to use a short branch instruction to branch to a location that is out of short branch range.

Branch Instructions: Example

- Draw a flowchart to describe what the following instruction sequence accomplishes:

```
        cmpa    #$3B
        bgt     M23
        addb    #1
        bra     xyz
M23:    subb    #1
xyz:    stab    bval
```

Compare & Test Instructions

- Before a conditional branch can be executed, condition codes need to be set up
- Compare & test instructions set up CCR without storing result

Mnemonic	Function	Branch Test
cba	Compare A to B	[A] – [B]
cmpa <opr>	Compare A to memory	[A] – [M]
cmpb <opr>	Compare B to memory	[B] – [M]
cpd <opr>	Compare D to memory	[D] – [M]:[M+1]
cps <opr>	Compare SP to memory	[SP] – [M]:[M+1]
cpx <opr>	Compare X to memory	[X] – [M]:[M+1]
cpy <opr>	Compare Y to memory	[Y] – [M]:[M+1]

Mnemonic	Function	Branch Test
tst <opr>	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

Useful Instructions: Clear, Complement & Negate

Mnemonic	Function	Operation
clr <opr>	Clear memory to zero	M ← 0
clra	Clear A to zero	A← 0
clrb	Clear B to zero	B ← 0

Mnemonic	Function	Operation
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]

Branching: Example

- Write an instruction sequence that will form the absolute value of a signed 8-bit integer stored in location $1000 and store the result in location $1004

Increment/Decrement Instructions

Mnemonic	Function	Operation
inc <opr>	Increment memory by 1	M ← [M] + 1
inca	Increment A by 1	A← [A] + 1
incb	Increment B by 1	B ← [B] + 1
ins	Increment SP by 1	SP ← [SP] + 1
inx	Increment X by 1	X ← [X] + 1
iny	Increment Y by 1	Y ← [Y] + 1

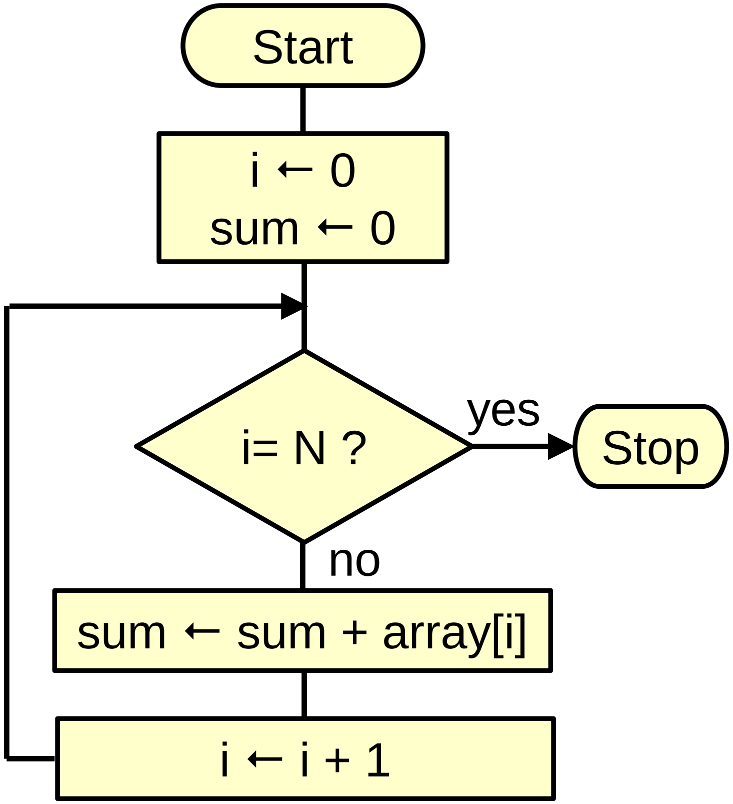
Mnemonic	Function	Operation
dec <opr>	Decrement memory by 1	M ← [M] – 1
deca	Decrement A by 1	A← [A] – 1
decb	Decrement B by 1	B ← [B] – 1
des	Decrement SP by 1	SP ← [SP] – 1
dex	Decrement X by 1	X ← [X] – 1
dey	Decrement Y by 1	Y ← [Y] – 1

Loop Example 1

- Write a program to add an array of N 8-bit numbers starting at address $1000 and store the 16-bit sum at memory locations $1100~$1101. Use a “for i = n1 to n2” looping construct

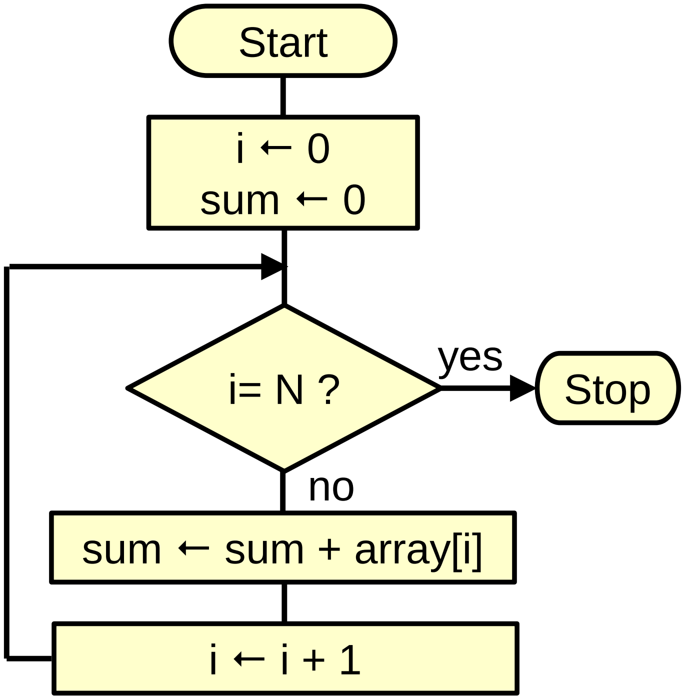
```
N:      EQU     20
        ORG     $1000
array:  DC.B    1,2,3,4,5,6,7,8,9,10,11,12
        DC.B    13,14,15,16,17,18,19,20
        ORG     $1100
sum:    DS.B    2
i:      DS.B    1

        ORG     $4000
        clr     i               ; i=0
        clr     sum             ; sum=0
        clr     sum+1
        ldx     #array          ; pointer to array
```



Loop Example 1 (cont.)

```
loop:   ldaa    i           ; i
        cmpa    #N          ; is i=N?
        beq     done
        ldab    a,x         ; array[i]
        ldy     sum         ; sum in Y
        aby                 ; sum=sum+array[i]
        sty     sum         ; update sum
        inc     i           ; increment index
        bra     loop
done:   bgnd                ; return to monitor
        END
```



Loop Primitive Instructions

- These increment/decrement loop counter (stored in register) and conditionally branch in one instruction
  - Range of branch is 9-bit (-256 to +255)

Mnemonic	Function	Operation
dbeq cntr, tar	Decrement counter and branch if = 0	cntr ← [cntr] – 1 if [cntr] = 0 then branch
dbne cntr, tar	Decrement counter and branch if ≠ 0	cntr ← [cntr] – 1 if [cntr] ≠ 0 then branch
ibeq cntr, tar	Increment counter and branch if = 0	cntr ← [cntr] + 1 if [cntr] = 0 then branch
ibne cntr, tar	Increment counter and branch if ≠ 0	cntr ← [cntr] + 1 if [cntr] ≠ 0 then branch
tbeq cntr, tar	Test counter and branch if = 0	if [cntr] = 0 then branch
tbne cntr, tar	Test counter and branch if ≠ 0	if [cntr] ≠ 0 then branch

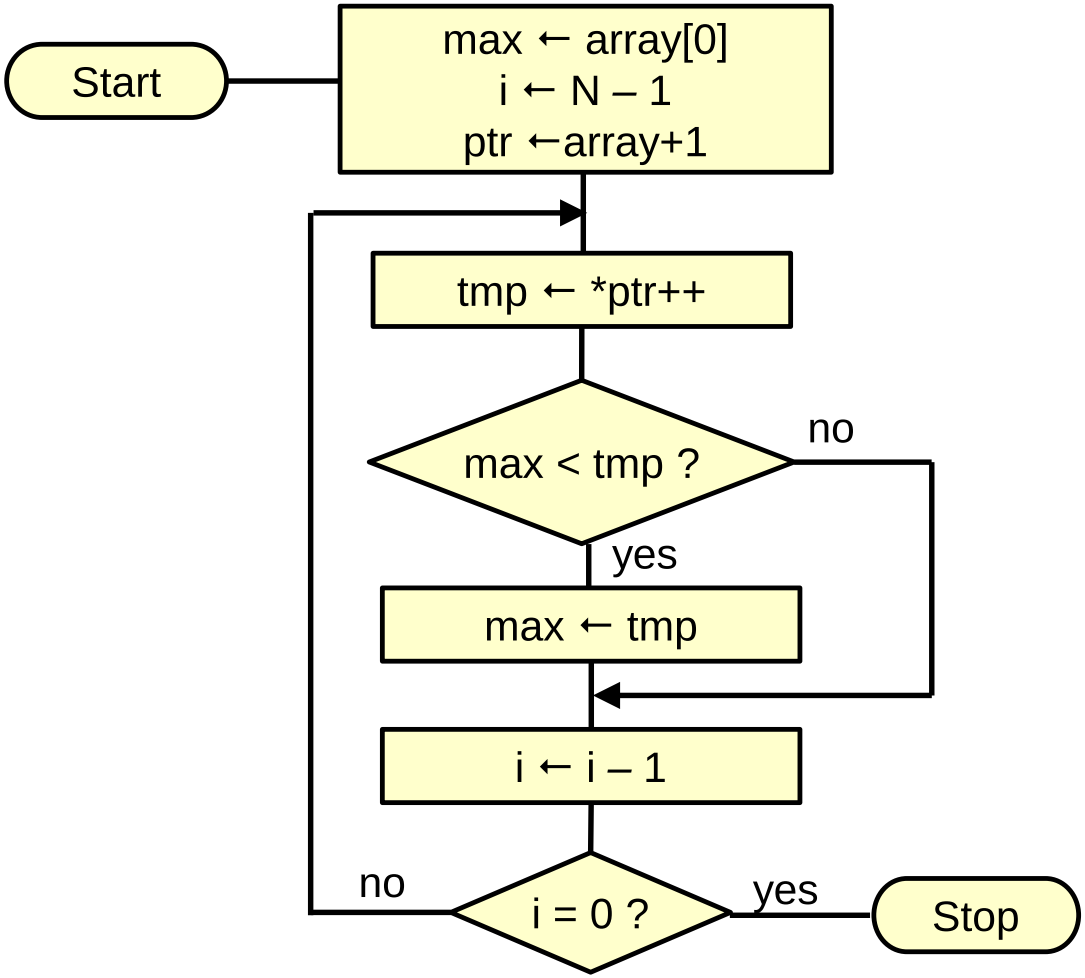
- **cntr** can be A, B, D, X, Y or SP, **tar** is branch target (label)

example:

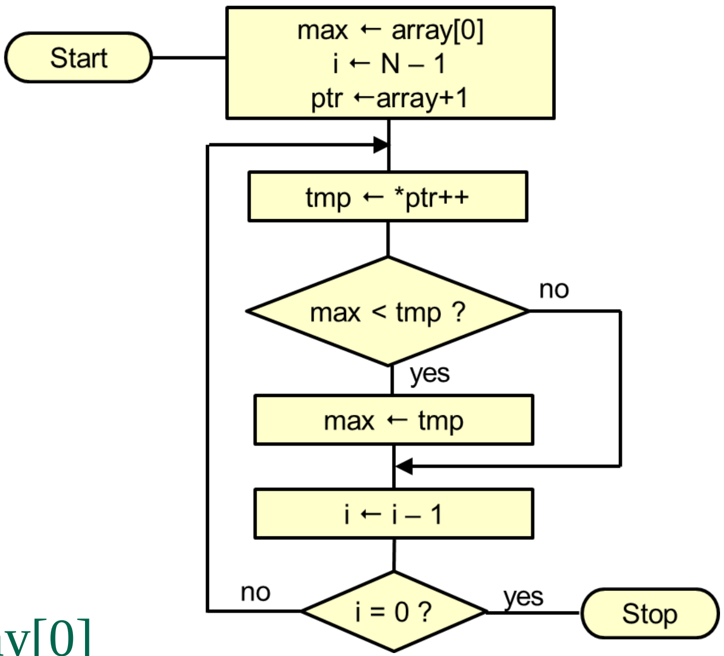
dbeq Y, loop1 ; decrement Y and branch to loop1 if Y=0

Loop Example 2

- Write a program to find the maximum element from an array of N 8-bit signed numbers starting at addr. $1000 and store that value at memory location $1100. Use a “repeat until C” looping construct.



Loop Example 2 (cont.)




```
N:      EQU     20
        ORG     $1000
array:  DC.B    1,-3,5,-6,19,41,-53,28,-13,-42
        DC.B    14,20,76,-29,-93,33,41,-8,61,4
        ORG     $1100
max:    DS.B    1


        ORG     $4000
        movb    array, max      ; init max=array[0]
        ldx     #array + 1      ; X points to array[1]
        ldab    #N –1           ; set loop count to N-1
loop:   ldaa    1,X +           ; load array value and incr pointer
        cmpa    max             ; compare to current max
        ble     skip            ; if <= max then skip update
        staa    max             ; update max=array value
skip:   dbne    b,loop          ; done yet?
here:   bra     here            ; stay here
```