Microcomputers I – CE 320

Mohammad Ghamari, Ph.D.
Electrical and Computer Engineering
Kettering University

Announcements

No announcements

Lecture 6: Simple Conditional Branches

Today’s Goals

- Understand the function of the Condition Code Register and how the bits are set.

- Use simple conditional branches to control the flow of programs.

Branch Instructions

- Branch instructions cause program flow to change when specific condition exist.

- Branches are used to perform:
  - Infinite execution
  - Conditional operations
  - Loops
  - Time delay (software controlled)

Branch Instructions

- HCS12 has three kinds of branch instructions
  - **Short** branches
  - **Long** branches
  - Bit conditional branches
- Branch instructions can be classified by the type of condition that must be satisfied in order for a branch to be taken.
  - Unary (unconditional) branch*: **Always** branch takes place.
  - Simple branch: Branch **if a condition is satisfied**.
    - A condition is satisfied if certain flags are set.
    - Usually there is a comparison or arithmetic operation to set up the flags before the branch instruction.
  - Unsigned & signed branches: Are taken when a comparison or test of unsigned/signed quantities results in a specific combination of condition code register bits.

Unary Branches

Mnemonic	Function	Equation or Operation
BRA	Branch always	1 = 1
BRN	Branch never	1 = 0

⇦ Unconditional branch

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

⇦ Branch is taken when a specific flag is 0 or 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 ⊕ 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

Compare and Test Instructions

- Condition flags need to be set up **before** conditional branch instruction are executed.
- The compare and test instructions **perform subtraction**, **set the flags based on the result**, and **does not store the result**. ONLY flags changes.
- Most instructions update the flags automatically so sometimes compare or test instructions are not needed.

Summary of compare and test instructions

Compare Instructions		
Mnemonic	**Function**	**Operation**
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)
Test instructions		
Mnemonic	**Function**	**Operation**
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

The memory and register does not change

<opr> can be an immediate value, or a memory location that can be specified using immediate, direct, extended, indexed addressing modes

Loop Primitive Instructions

- HCS12 provides a group of instructions that either decrement or increment a loop count to determine if the looping should be continued.
- The range of the branch is from $80 (-128) to $7F (+127).

Summary of loop primitive instructions

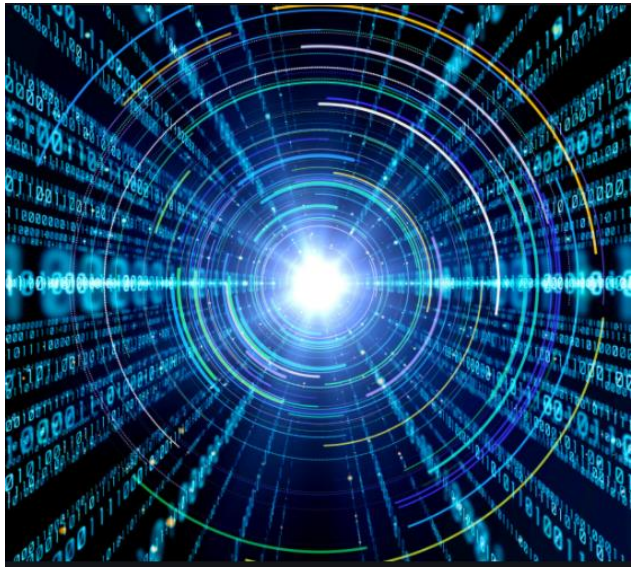
Mnemonic	Function	Equation or Operation
dbeq cntr, rel	Decrement counter and branch if = 0 (cntr = A, B, D, X, Y, or SP)	cntr ⇐ (cntr) - 1 If (cntr) = 0, then branch else continue to next instruction
dbne cntr, rel	Decrement counter and branch if ≠ 0 (cntr = A, B, D, X, Y, or SP)	cntr ← (cntr) - 1 If (cntr) ≠ 0, then branch else continue to next instruction
ibeq cntr, rel	Increment counter and branch if = 0 (cntr = A, B, D, X, Y, or SP)	cntr ← (cntr) + 1 If (cntr) = 0, then branch else continue to next instruction
ibne cntr, rel	Increment counter and branch if ≠ 0 (cntr = A, B, D, X, Y, or SP)	cntr ← (cntr) + 1 If (cntr) ≠ 0, then branch else continue to next instruction
tbeq cntr, rel	Test counter and branch if = 0 (cntr = A, B, D, X, Y, or SP)	If (cntr) = 0, then branch else continue to next instruction
tbne cntr, rel	Test counter and branch if ≠ 0 (cntr = A, B, D, X, Y, or SP)	If (cntr) ≠ 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.

Note: rel is the relative branch offset and usually a label

Review: Program Loops

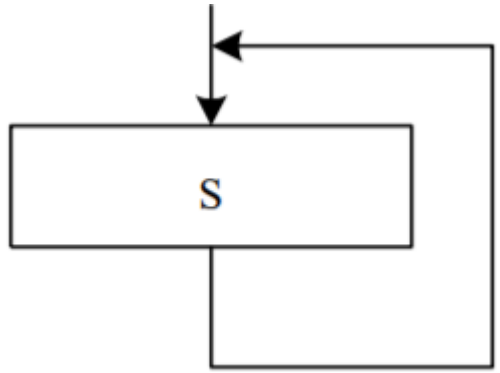
- Many applications require **repetitive operations**.
- We can write programs to tell computers to perform the **same operation over and over**.
- A **finite loop** is a sequence of instructions that will be executed by the computer **for a finite number of times**.
- An **endless loop** is a sequence of instructions that the computer **will execute forever**.
- There are four major loop constructs:
  - **Do statement S forever**
  - **For-loop**
  - **While C Do S**
  - **Repeat S Until C**



Endless Loop

- Do a sequence of instructions (S) forever.

```
Loop: ldaa 1,x+
      adda #$12
      bra Loop
```



An infinite loop

For Loops

```
For (i = n1, i <= n2, i++)
{a sequence of instructions (S) }
```

OR

```
For (i = n2, i >= n1, i--)
{a sequence of instructions (S) }
```

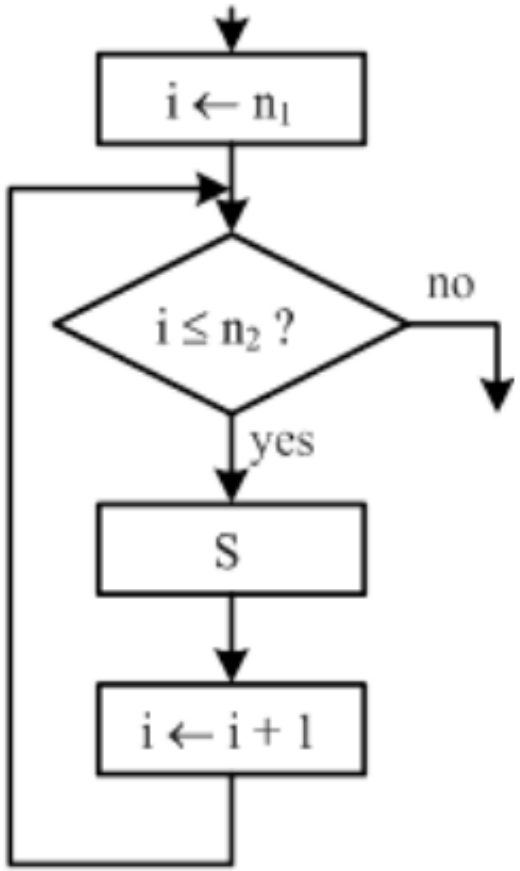
- i is loop counter that can be incremented (or decremented) in each iteration.
- Sequence S is repeated n2-n1+1 times
- n2 > n1

Steps:

1- Initialize loop counter
2- Compare the loop counter with the limit n2 (or n1) if it is not equal do the loop otherwise exit
3- increment (or decrement) the loop and go to step 2

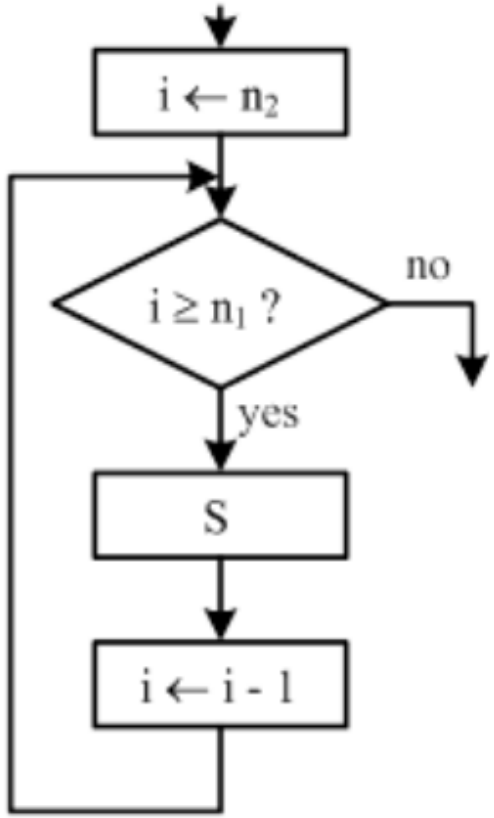
Implementation of for (i = n1, i <= n2, i++) {S}

```
n1  equ 1              ; starting index
n2  equ 20             ; ending index
i  ds.b  1             ; i is the loop counter

 movb  #n1,i           ;initialize i to n1

Loopf: ldaa   i        ;check index i
       cmpa   #n2
       bhi Next        ; if i > n2, exit the loop

         …             ; performs S
         …             ;       "

      inc i            ;increment loop index
      bra Loopf        ;go back to the loop body

Next:  …
```



Implementation of for (i = n2, i > = n1, i--) {S}

```
n1  equ 1              ; starting index
n2  equ 20             ; ending index
i   ds.b  1            ; i is the loop counter

 movb   #n2,i          ;initialize i to n2

Loopf: ldaa    i       ;check index i
       cmpa    #n1
       blo Next        ; if i < n1, exit the loop

         …             ; performs S
         …             ;        "

       dec i           ;decrement loop index
       bra Loopf       ;go back to the loop body

Next:  …
```



Since i is a byte, the max. number of iterations is 256. For more iterations:-

1- use nested loops - outer and inner For loops See next slide.

Or 2- i can be a word. See next slide.

i is word (up to 65,535 iterations)

```
n1  equ 1
n2  equ 6000
i   rmb 2

 movw  #n2,i

 ldd   i
Loopf: cpd #n1
       blo Next
           …
           …
       ldd i
       subd #1
       std i
       bra Loopf

Next:  …
```

Nested loops

```
n11 equ 1
n12 equ 20
n21 equ 1
n22 equ 20
i1 ds.b 1
i2 ds.b 1
movb #n12,i1
Loop1: ldaa   i1
       cmpa   #n11
       blo next1

movb #n22,i2
Loop2: ldaa   i2
       cmpa   #n21
       blo next2
        …….    ; performs S
        …….
        …….
       dec i2
       bra Loop2
next2:    dec i1
          bra Loop1
next1:
```

For loop using dbeq

up to 65,535 iterations

```
n equ  6000      ; number of
                   iterations
 ldx #n+1
Loopf: dbeq x,next
       ……..      ; performs S
       ……..
        bra Loopf


next:  …
```

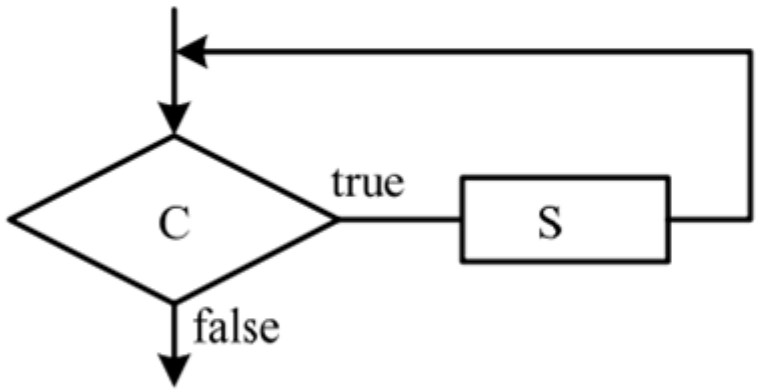
up to 256 iterations

```
n equ  60 ; number of
            iterations
 ldab #n+1
Loopf: dbeq b,next
       ……..   ; performs S
       ……..…
        bra Loopf


next:  …
```

While Loop

While (condition) { Sequence S; }

- The condition is evaluated first, if it is false, S will not be executed
- Unlike for loop, the number of iterations may not be known beforehand
- It will repeat until an event happens, e.g., user enter escape character



The While ... Do looping construct

While (icount ≠ 0) {Sequence S;}

```
N equ 10
icount ds.b 1
 movb #N,icount ; initial value
Wloop: ldaa #0
       cmpa icount
       beq  Next
        ……. ; perform S
        ……..
       bra  Wloop
Next: …
```

The update of icount is done by an interrupt service routine (not shown)

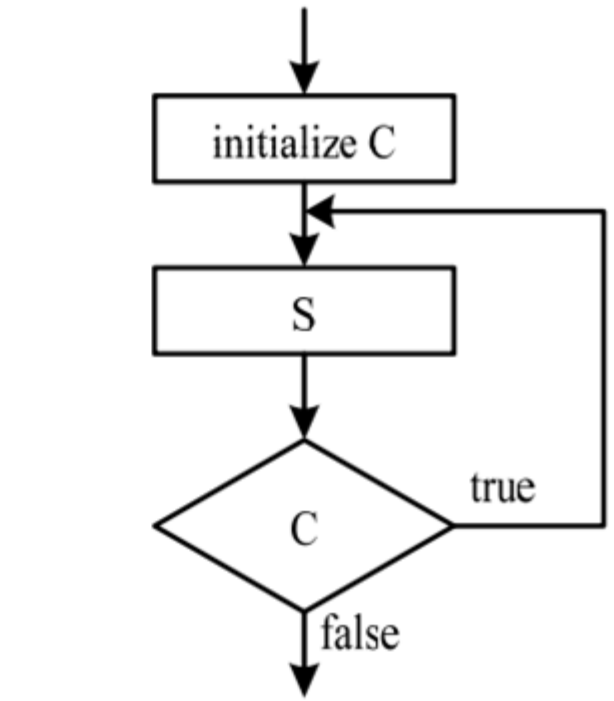
Do - While loop

Do { Sequence S; } While (condition)

- The main difference between while and do-while loops is that do-while loop can execute S at least once because it is executed first and then the condition is evaluated.

Do {Sequence S;} While (icount ≠ 0)

```
N equ 10
icount ds.b 1
 movb #N,icount
Wloop:
        ……..     ; perform S
        ……..     ; "

        ldaa  #0
        cmpa  icount
        bne Wloop
       ……..
```



The Repeat ... Until looping construct

Other examples:-

Do {Sequence S;}
While (m1 == m2)

```
m1 ds.b 1
m2 ds.b 1
movb  #5,m1  ; initial value
movb  #5,m2  ; initial value
Wloop:
        .....     ; perform S
        .....     ;  "

        ldaa  m1
        cmpa  m2
        beq Wloop
       .....
```

I = 1;
Do { Sequence S;
I++;}
While (I<= 10)

```
I rmb 2
movw #1,I
Wloop:
        .....     ; perform S
     ldd I
     addd #1
     std I
     cpd #10
     bls Wloop
```

OR

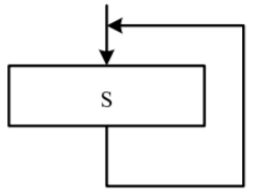
```
ldy #10        ;Y is loop counter (I) = 10
Wloop: …
        …
     dbne Y,Wloop      ;Y = Y-1
                       ;loop if Y ≠ 0
```

Program Loops

```
ldaa 1,x+      ;Loads Array1 into accumulator A and
               increments value in X.

adda           ;adds the contents of memory location M to
               accumulator A and places the result in A.
```

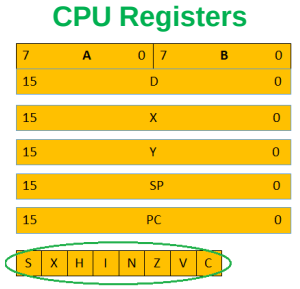
- Program loops are implemented by using the **unconditional** and **conditional** branch instructions.
  - The execution of **unconditional** branch instruction can be done by endless loops.

```
Loop: ldaa 1,x+
      adda #$12
      bra Loop
```

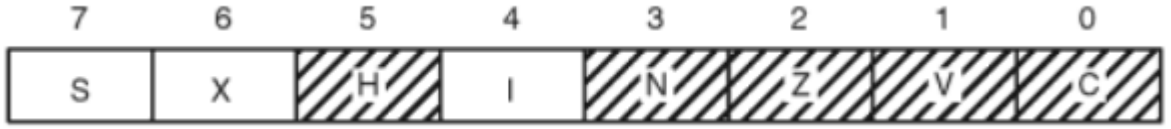


An infinite loop

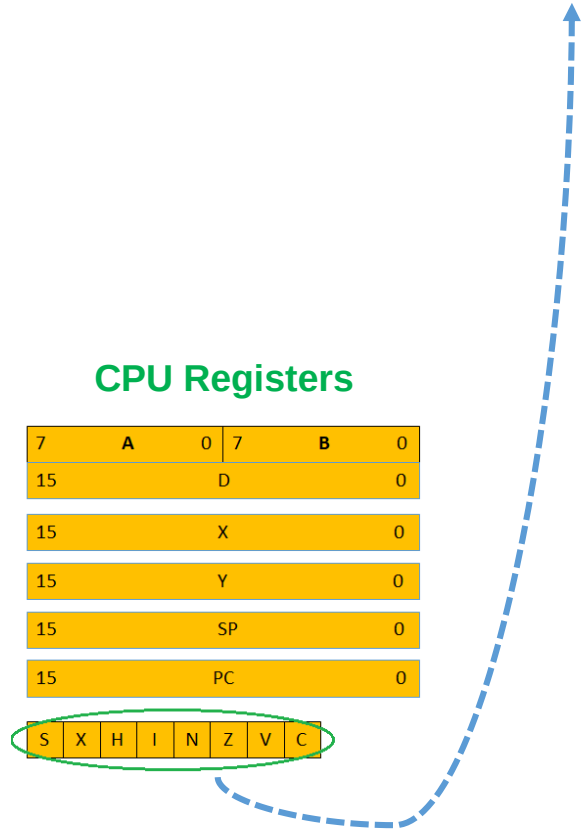
  - The execution of **conditional** branch instructions depends on the contents of the CCR register.
    - When executing conditional branch instructions, the HCS12 **checks the condition flags** in the **CCR register**.



Program Loops



- **Condition Code Register**:
  - The shaded characters are **condition flags** that reflect the status of an operation.
    - ***Carry flag (C)*:** Whenever a carry is generated as the result of an operation, this flag will be set to 1. Otherwise, it will be cleared to 0.
    - ***Overflow flag (V):*** Whenever the result of a two's complement arithmetic operation is out of range, this flag will be set to 1. Otherwise, it will be set to 0. The V flag is set to 1 when the carry from the most significant bit and the second most significant bit differ as the result of an arithmetic operation.
    - **Zero flag (Z):** Whenever the result of an operation is zero, this flag will be set to 1. Otherwise, it will be set to 0.
    - **Negative flag (N):** Whenever the most significant bit of the result of an operation is 1, this flag will be set to 1. Otherwise, it will be set to 0. This flag indicates that the result of an operation is negative.
    - **Half-carry flag (H):** Whenever there is a carry from the lower four hits to the upper four hits as the result of an operation, this flag will be set to 1. Otherwise, it will be set to 0.



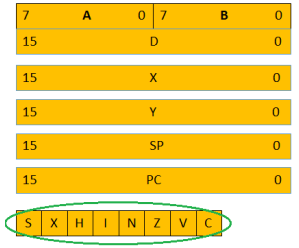
Making Decisions

- We've learned unconditional branch (BRA) already.
- Conditional branches
  - Programs often need to decide which portion will be executed **based on conditions**.

- In microcontrollers, two steps are required to make decisions

  1. Evaluating a Boolean statement and generate a true or false result.

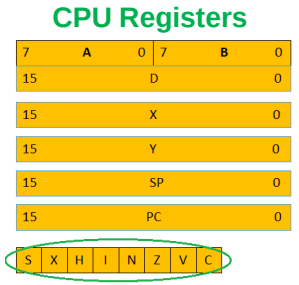
  2. Using a conditional branch that uses the Boolean result as a condition.
     - If the result is true, the branch changes the PC.
     - Otherwise the PC remains and continues on the next sequential instruction.
- Each of these steps is done with separate lines of code.



- CCR is one byte register that **stores the results of the Boolean statements** used for branching.

- Once some of these bits have been set, the conditional branches are used to inspect them.

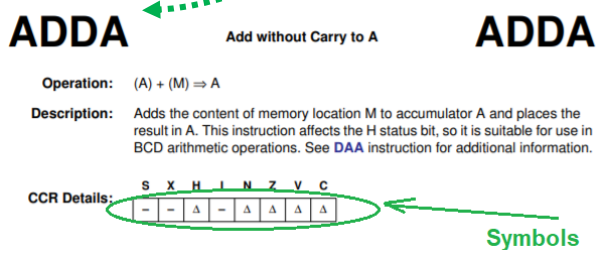
Condition Code Register (CCR)

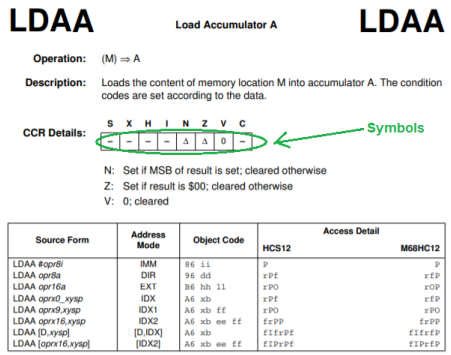


- **How the CCR bits are set?**
  - **Arithmetic instructions** (addition and subtraction) affect the **N**, **Z**, **V**, **C** and **H** bits.
  - **Data transfer instructions** affect **N**, **Z**, and **V** bits.
  - Branches don't affect any CCR bits.
  - The instruction set details the effect of each instruction on all of the CCR bits.

Symbol	Operation
-	Unaffected
1	Always "set" to 1
0	Always "cleared" to 0
↕ or Δ	Set or cleared based on result

LDAA is a data transfer instruction





Simple Conditional Branches

- Simple conditional branches **examine** only one CCR bit.

- There are **two instructions** for each of the N, Z, V, and C bits.
  - E.g. For N bit: **BMI & BPL**

CCR Bit	Branch Taken if Bit is 1	Branch Taken if Bit is 0
N	BMI	BPL
Z	BEQ	BNE
V	BVS	BVC
C	BCS	BCC

Example

```
        LDAA    #0
        BEQ     LABEL_M
        do something
LABEL_M:  do something else
```

Instructions

- BMI, BPL, LBMI, LBPL – Branch to a new instruction based on the N bit
- BEQ, BNE, LBEQ, LBNE – Branch to a new instruction based on the Z bit
- BVS, BVC, LBVS, LBVC – Branch to a new instruction based on the V bit
- BCS, BCC, LBCS, LBCC – Branch to a new instruction based on the C bit
- DEC, DECA, DECB – decrement an 8-bit number in memory or accumulator A or B, used to decrement an 8-bit loop counter (affects N, Z, and V but NOT C)
- DEX, DEY – decrement a 16-bit number in register X or Y, used to decrement a 16-bit loop counter (ONLY affects Z)
- INC, INCA, INCB – increment an 8-bit number in memory or accumulator A or B, used to increment an 8-bit loop counter (affects N, Z, and V but NOT C)
- INX, INY – increment a 16-bit number in register X or Y, used to increment a 16-bit loop counter (ONLY affects Z)
- TST, TSTA, TSTB – Compare an 8-bit value in memory, A, or B to 0, allowing branches that check the Z bit or N bit.

Source Form	Operation	Addr. Mode
BMI *rel8*	Branch if Minus (if N = 1)	REL

Simple Conditional Branches

- Example: Write code that executes a loop 3 times.

```
1: 86 03      LDAA #03          ($2000)
2: 27 04      BEQ $04           ($2002)
3: 43         DECA              ($2004) sets Z bit to 1 if decrements to 0.
4: 20 FB      BRA -5            ($2005)
5:  20 FE     BRA -2            ($2007)
```

Simple Conditional Branches

- Example: Write code that executes a loop 3 times.

```
1: 86 03      LDAA #03          ($2000)
2: 27 04      BEQ $04           ($2002)
3: 43         DECA              ($2004) sets Z bit to 1 if decrements to 0.
4: 20 FB      BRA -5            ($2005)
5:  20 FE     BRA -2            ($2007)
```

- **Q1:** Which lines are the setup?
- **Q2:** Which lines are the actual loop?
- **Q3:** If the loop was used to perform a function three times, where should this code be inserted?

Program Trace

- Trace the previous program showing PC, A, and the CCR (N, Z, V, and C).
- Assume the program begins at address $2000.

Trace Line	Code Line	PC	A	N	Z	V	C
1	1	2002	03	0	0	0	-
2	2	2004	03	0	0	0	-
3	3	2005	02	0	0	0	0
4	4	2002	02	0	0	0	0
5	2	2004	02	0	0	0	0
6	3	2005	01	0	0	0	0
7	4	2002	01	0	0	0	0
8	2	2004	01	0	0	0	0
9	3	2005	00	0	1	0	0
10	4	2002	00	0	1	0	0
11	2	2007	00	0	1	0	0
12	5	-	-	-	-	-	-

Wrap-up

What we've learned

- Bits in the Condition Code Register (CCR)

- Simple conditional branches:
  - BMI, BPL, BEQ, BNE, BVS, BVC, BCS, BCC

What to Come

- Comparison branches