M68000 instr's continued

7 - while + for

Assembler construct for the while-statement

Consider C-language's construct for a while-statement:

```
while (condition)
    statement
```

Assembler construct:

```
whileloopLabel:   [ evaluate while condition & set flags
                    (use cmp!) ]

                  branch to "while-end-label" if condition not met

                  [ statement (of while body) in assembler

                  branch to "whileloopLabel" (repeat)

while-end-label:
```

Example: division by repeated subtraction

```
A: ds.l 1
B: ds.l 1
Q: ds.l 1
R: ds.l 1
```

We find A / B by repeated subtraction.
($A > 0$ & $B > 0$ assumed).

C program:

```
Q = Ø;
while (A ≥ B)
   {  Q = Q+1;
      A = A - B;
   }
R = A;
```

eg: $A = 8$
$B = 3$

$Q = \emptyset$

iter 1: $Q = Q+1 = 1$
$A = A - B = 5$.

iter 2: $Q = Q+1 = 2$
$A = A - B = 2$

done

$Q = 2$, $R = 2$.

Assembler program structure:

instr's to do $Q = \emptyset$

whileStart: instr's to evaluate condition $A \geq B$

branch instr. that jumps to "whileEnd" if $A \geq B$ is false.

instructions to do $Q = Q + 1$ and $A = A - B$

Bra whilestart (repeat).

whileEnd: instr. to do $R = A$

Assembler program:

```
A: ds.l 1
B: ds.l 1
Q: ds.l 1
R: ds.l 1
```

whiles.s

(1) continue

(2) break point

```
            move.l #Ø, Q

whileStart: move.l  A, DØ
            cmp.l   B, DØ          test A against B.

            blt     WhileEnd

            addq.l  #1, Q      <   move.l Q, DØ
                                   add.l #1, DØ
                                   move.l DØ, Q.
            move.l  B, DØ
            sub.l   DØ, A

            bra     whilestart

WhileEnd:   move.l  A, R
```

~~[illegible]~~

Turbo charged assembler code - advanced

whilela.s

```
A:  ds.l 1
B:  ds.l 1
Q:  ds.l 1
R:  ds.l 1
```

- We avoid using variables values in memory.

Instead, we use temp. values in registers:

~~D7 = Q~~
~~D6 = A~~

D0 = A
D1 = B
D2 = Q.

```
            move.l  #0, D2

~~whilestart~~:  move.l  A, D0      ] prepare
            move.l  B, D1

whilestart: cmp.l   D1, D0         test A against B
            blt     whileend

            addq.l  #1, D2
            sub.l   D1, D0
            bra     whilestart
whileEnd:   move.l  D0, R
            move.l  D2, Q.
```

Example: sum 10 array elements:

```
sum = 0
i = 0
while (i < 10)
  { sum = A[i] + sum;
    i = i+1;
  }
```

Assembler code:

while2.s

```
Sum: ds.l   1
  i: ds.l   1
  A: ds.l   10


     move.l  #0, sum            (sum = 0)
     move.l  #0 , i             (i = 0 )

whileStart:  move.l  i, D0
             cmp.l  #10, D0     (test i against 10.)

             bge    whileEnd

                    ↓ start of while
                      body.
```

```
move.l   #A, A0
move.l   i, D0
muls     #4, D0
move.l   0(A0, D0.w), D7

add.l    D7, sum          (sum = sum + A[i])

addq.l   #1, i            (i = i+1)

bra      whileStart

whileEnd:
         nop.
```

→ notice i is just a way for C to identify the element A[i]. , walk down array A from 0 to N

→ we can walk down array elements more efficiently with addr. reg's.

Turbo assembler code: — moving pointer!

int A[10];
int sum;

(Windows)

make variable sum equal to A[0] + A[1] + ... + A[9]

- We use M68000 registers as scratch areas! for speed.

```
         move.l   #0, D0        (clear sum)
setup    move.l   #A, A0        (make A0 points to next elem. to sum!)
         move.l   #0, D1        D1 = loop count

loop:    cmp.l    #10, D1       D1 ?? 10
         BEQ      Loopexit

         add.l    (A0), D0      add A[i] to sum
next array element
         adda.l   #4, A0        make A0 points to next array element
         add.l    #1, D1        increase loop count

         bra      loop
loopexit: move.l  D0, sum       - Put sum in it's rightful place.
```

Example: ~~while~~
```
int Found;
int x;
int A[10];
```
(compound while condition + nested if !!!)

x is assigned a value previously.
And we want to find x in Array A.

```
Found = 0; i = 0;
while ( i<10 && !Found)
    if (A[i] == x)
        {found = 1 ;}
         break;   <------
    else
        { i++; }
```

Found: ds.l 1
x: ds.l 1
i: ds.l 1
A: ds.l 10

- Note: !Found (not Found) is actually

the following comparison: ~~Found ≠ 0~~

$$Found == 0$$

```
            move.l   #Ø, Found                  (Found = Ø)
            move.l   #Ø, i                      (i = Ø).

whileloop:  move.l   i, dØ                  }  stop while loop
            cmp.l    #10, dØ                }  when i ≧ 10
            bge      whiledone              }

            move.l   Found, dØ              }  stop while loop
            cmp.l    #Ø, dØ                 }  when Found ≠ Ø.
            bne      while end              }

            ________________________________   start of if.
body    {   move.l   i, dØ
of      {   muls     #4, dØ
while   {   move.l   dØ, aØ
        {   move.l   A(aØ), dØ
        {   Cmp.l    x, dØ
        {   bne      elsepart
        {
        {   move.l   #1, Found
        {   bra      doneif ifdone  Whildone     break;
        {
elsepart:   move.l   i, dØ
        {   add.l    #1, dØ
        {   move.l   dØ, i
ifdone :    ________________________________   end of if.

            bra whileloop
whiledone:
```

Turbo version:

Goal: find x in A[i]
Found = 0 if not found
Found = 1 if found, i contains index: A[i] = x

Found = ∅

make A∅ point to start of array
D1 = ∅ (loop count)
D2 = x

whilestart: Test D1 < 0
exit if false

Test (A∅)+ against D2
Goto elsepart if false

Found = 1
i = D1
exit while loop

elsepart: D1 = D1 + 1. (increment D1).

Bra whilestart.

```
A: ds.l 10
Found: ds.l 1
 i : ds.l 1
 x : ds.l 1
            move.l  #0, Found

            move.l  #A, A0
            move.l  #0, D1
            move.l  x, D2

WhileStart:  cmp.l  #10, D1
             bge    whileEnd

             cmp.l  (A0)+, D2
             bne    elsePart

             move.l #1, Found
             move.l D1, i
             bra    WhileEnd

ElsePart:    addq.l #1, D1

             BRA  WhileStart

WhileEnd:
```

Assembler construct for for-statement

- For-statement :

```
for ( expr1 ; expr2 ; expr3 )
    statement
next statement
```

- This is semantically equivalent to:

```
expr1 ;
while (expr2)
  { statement
    expr3 ;
  }
next statement
```

- You already know how to translate while statement. For-statement can be handled in the similar way.

Example: search for a value x in an array A[10].

C-code:

```
Found = FALSE;
for (i = 0 ; i < 10
    if (A[i] == x )
       { Found = TR
         break;
       }
```

Better example:

```
max = A[0];
for(i=1; i<N; i++)
   if (A[i] > max)
      max = A[i];
```

First rewrite for-statement into while-statement:

```
Found = FALSE;

i = 0;
while (i < 10)
  { if (A[i] == x )
       { Found = TRUE;
         break;
       }
    i++;
  }
```

Now translate into assembler:

```
          move.l   #0, Found
          move.l   #0, i

whileloop:  move.l   i, d0           } test i<20
            cmp.l    #10, d0
            bge      whiledone

(if:)       move.l   #A, a0
            muls     #4, d0
            move.l   0(a0, d0.w) d0        get A[i]
                                 ↑
            cmp.l    x, d0        (d0.w) necessary.
            bne      ifdone

(thenpart:) move.l   #1, Found
            bra      while done            (break!)

ifdone:     addq.l   #1, i                 (i++ !!!)

            bra while loop


whiledone:  ____________

   x:   ds.l  1
   A:   ds.l  10
   i:   ds.l  1
```

Faster version:

```
        move.l  #0, Found

        move.l  #0 , D0
        move.l  #A , A0

for_start:
        cmp.l   #10, D0
        bge     for_end

        cmp.l   (A0)+ , X
        bne     for_start

        move.l  #1, Found
        move.l  D0 , i

for_end:
```

Complete set of conditional branch instructions in M68000

Standard construct:

```
CMP  <source>, Dn
Bcc  <label>
next instruction
```

Bcc	Effect
signed	
BEQ	Branch to <label> if Dn = <source>, otherwise continue with next instruction. (Z flag set)
BNE	Branch to <label> if Dn ≠ <source>. (Z flag reset)
BLT	Branch to <label> if Dn < <source> (signed)
BLE	Dn ≤ <source> (signed)
BGT	Dn > <source> (signed)
BGE	Dn ≥ <source> (signed)
unsigned	
BHI	(high) Dn > <source> (unsigned)
BLS	(low or same) Dn ≤ <source> (unsigned)
BCC	(carry clear) Branch if Carry Flag = ∅ (cleared). Also used as: Branch to <label> if Dn ≥ <source> (unsigned)
BCS	(carry set) Branch if carry flag = 1 (set). Also used as Branch to <label> if Dn < <source> (unsigned)

- Sign equal & not equal comparison is done with BEQ & BNE

BCC	Effect
BVC	Branch to <label> if Overflow flag = Ø (clear) Used to check overflow errors.
BVS	Branch to <label> if overflow flag = 1 (set).
BMI	(Minus) Branch to <label> if Negative flag = 1. (value is negative)
BPL	(plus) Branch to <label> if Negative flag = Ø.

Signed & unsigned comparison: Recall:

- Unsigned numbers don't use 2's complement representation.

  Instead, the value is always ≥ 0 and is computed as: $a_{n-1} a_{n-2} \dots a_0$.

$$\text{value} = \sum_{i=0}^{n-1} a_i \cdot 2^i$$

- Signed numbers use the 2's complement representation system

eg: bit pattern: 1000.0001

if this is a unsigned number, it's value is:

1000.0001

2^7 2^0

$128 + 1 = 129$.

if this is a signed number, it's value is:

1000.0001

↓

0111.1110

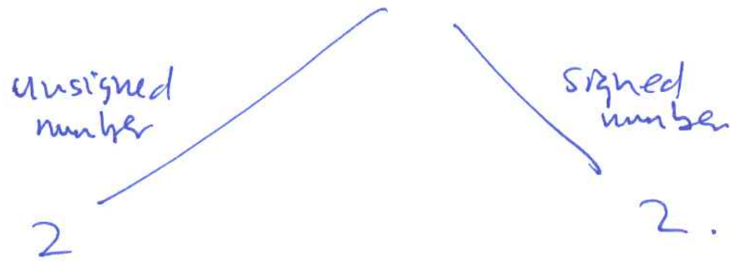
+1

0111.1111

$64+32+16+8+4+2+1 = 127$

Value $= -127$.

The bit pattern 0000.0010 = 2



When the patterns: 1000.0001 & 0000.0010 are compared, we have:

type of comparison	result.
signed	1000.0001 < 0000.0010 "−127 < 2"
unsigned	1000.0001 > 0000.0010 because 129 > 2.

Relational operator	signed	unsigned
==	BEQ	BEQ
!=	BNE	BNE
<	BLT	BCS
<=	BLE	BLS
>	BGT	BHI
>=	BGE	BCC