Branching and Looping Continued

Do-while loops

- Strict translation of a while loop uses 2 jumps

```
while:
    some compare
    conditional jump to "ewhile"
    .....
    .....
    jump to "while"
ewhile:
```

- However, a do-while only requires one jump.

```
do_while:
    .....
    .....
    .....
    some compare
    conditional jump to "do_while"
```

Do-while loops

Any **while** loop can be simulated by a **do-while** loop wrapped in an **if** statement. For example

```
while ( condition )
{
   statements;
}
```

can be simulated as

```
if ( condition )
{
   do
   {
      statements;
   } while ( condition );
}
```

Ugly C code to search through a character array

```
//Looking for an character x other than 0. Store index in n
// data is a null terminated character array
    i = 0;
    c = data[i];
    if ( c != 0 )
    do
    {
        if ( c == x )
          break;
        i++;
        c = data[i];
    } while ( c != 0 );

    n = c == 0 ? -1 : i;
```

Assume we have the following data segment

Assume we have the following data segment

```
   section .data
data   db    "hello world",0
n      dq     0
x      db    'w'
```

Assembly code to search through an array

```
        mov     bl, [x]          ; value being sought
        mov     rcx, 0           ; i = 0;
        mov     al, [data+rcx]   ; c = data[i]
        cmp     al, 0            ; if ( c != 0 ) {
        jz      end_do_while     ; skip loop for empty string
do_while:
        cmp     al, bl           ; if ( c == x ) break;
        je      found
        inc     rcx              ; i++;
        mov     al, [data+rcx]   ; c = data[i];
        cmp     al, 0            ; while ( c != 0 );
        jnz     do_while
end_do_while:
        mov     rcx, -1          ; If we get here, we failed
found:  mov     [n], rcx         ; Assign either -1 or the
                                 ; index where x was found
```

Assembly code to search through an array (Using only 64 bit registers)

```
        movzx   rbx, byte[x]            ;<----
        mov     rcx, 0
        movzx   rax, byte [data+rcx]    ;<----
        cmp     rax, 0                  ;<----
        jz      end_do_while
do_while:
        cmp     rax, rbx
        je      found
        inc     rcx
        movzx   rax, byte [data+rcx]    ;<----
        cmp     rax, 0                  ;<----
        jnz     do_while
end_do_while:
        mov     rcx, -1
found:  mov     [n], rcx
```

Counting loops

```
// assume we have 3 arrays of size n.
// Each containing longs (quad words)
```

```
    for ( i = 0; i < n; i++ )
    {
        c[i] = a[i] + b[i];
    }
```

Counting loops

```
//assume there are 3 contiguous segemnts in memory
// each containing n quad words.
   mov     rdx, [n]         ; use rdx for n
   xor     ecx, ecx         ; i = 0
for:
   cmp     rcx, rdx         ; i < n
   je      end_for          ; get out if equal
   mov     rax, [a+rcx*8]   ; get a[i]
   add     rax, [b+rcx*8]   ; a[i] + b[i]
   mov     [c+rcx*8], rax   ; c[i] = a[i] + b[i];
   inc     rcx              ; i++
   jmp     for
end_for:
```

Nested loops

Consider the double summation

$$\sum_{i=1}^{N} \sum_{j=1}^{i} j \tag{1}$$

ignoring the fact that

$$\sum_{i=1}^{N} \sum_{j=1}^{i} j = \frac{N(N+1)(N+2)}{6} \tag{2}$$

We will code this double sum in assembler.

Nested loops

Assuming we have:

```
segment .data
Sum:    dq 0
N:      dq 5
```

Nested loops

```
    mov rbx, [N]
    mov rax, 0          ; sum=0
    mov r8,1            ; i=1
loop1:
    cmp r8,rbx
    jg eloop1           ; !(i<=N)
    mov rcx,1           ; j=1
    loop2:
        cmp rcx, r8
        jg  eloop2      ; !(j<=i)
        add rax,rcx
        inc rcx
        jmp loop2
    eloop2:
    inc r8
    jmp loop1
eloop1:
  mov [Sum], rax
```

Loop instructions

- The CPU has instructions like
  - loop
  - loope
  - loopne

  which are designed for looping.
- They decrement `rcx` and do the branch if `rcx` is not 0
  - loope checks if zero flag is set as well.
  - loopne checks if the zero flag is not set as well
- It is faster to use `dec` and `jnz` instead
- The label must be within -128 to +127 bytes of `rip`
- Probably pointless on modern architecture. (it was fast on old architecture.)

Loop instructions

Add 5 to a sum 64 times.

```
    xor rax,rax; sum=0
    mov rcx, 64;
loop1:
    add rax, 5
    loop loop1
```

string (array)

- Let use first consider the simple array instruction

  `movsb`
- we must load the address of source data in `rsi`
- we must load the address of destination data in `rdi`
- on execution `movsb` will move the value at rsi to rdi, and increment both addresses by 1.

```
....
mes1: db "abcdefg"
mes2: db "1234567"
....
   mov rsi, mes1
   mov rdi, mes2
   movsb
```

mes2 will equal "a234567" and rsi=mes1+1 and rdi=mes2+1

Repeat string (array) instructions

But how is this useful?

- we utilize the string operation with the `rep` instruction.
- rep will repeatedly call the string operation until `rcx`= 0.

For example, let us copy an array of 1000 bytes.

```
lea rsi, [source]
lea rdi, [destination]
mov rcx, 1000
rep movsb
```

- `lea`?
- lea rsi, [source] = mov rsi, source

Repeat string (array) instructions

- What is if the array contains non bytes?
- We simple use the different size specifier
  - `movsw`
  - `movsd`
  - `movsq`
- Now instead of incrementing the addresses by 1,
- We will increase it by the size.
- e.g. with movsq we increment by 8

Repeat string (array) instructions

- Up to now we have relied on an incrementing the source and destination addresses.
- actually the address is only increased if the direction flag(DF)=0 (default)
- If DF=1, the addresses will decrement after each string instruction
- We can set the direction flag to 1 with

```
std
```

- or 0 with

```
cld
```

Store instruction

- The `stosb` instruction stores the byte in `al` at the address specified in `rdi` and increments `rdi`
- If the direction flag is set it decrements `rdi`
- There are also `stosw`, `stosd` and `stosq` to operate 2, 4 and 8 byte quantities

```
mov     eax, 1
mov     ecx, 1000000
lea     rdi, [destination]
rep     stosd       ; place 1000000 1's in destination
```

Scan instruction and Compare instruction

Scan: (`scas`)

- There are a collection of scan string instructions which scan data from the address pointed at by `rdi` and increment (or decrement) `rdi`
- They compare data against `al`, `ax`, `eax`, ...
- Used with `repne`, and will stop once data found or `rcx`=0

Compare: (`cmps`)

- The compare string instructions compare the data pointed at by `rdi` and `rsi`
- End once `rcx` has reached zero or a match is not found.
- Used with `repe`, and will stop once match found or `rcx`=0