Lecture Notes on Assembly Language - J. Vaughan

16. If statements

if (a == b) { statements }

```
        mov eax, dword[a]
        mov ebx, dword[b]
    cmp eax, ebx
        jnz continue
        ; statements
continue:
```

if (a == b) { statements1 }
else { statements2 }

```
        mov eax, dword[a]
        mov ebx, dword[b]
        cmp eax, ebx
        jnz else
        ; statements1
        jmp continue
else:
        ; statements2
continue:
```

if (a == b) { statements1 }

else if (c == d) { statements2 }

else { statements3 }

```
            mov eax, dword[a]
            mov ebx, dword[b]
            cmp eax, ebx
            jnz elseif
            ; statements1
            jmp continue
elseif:
            mov eax, dword[c]
            mov ebx, dword[d]
            cmp eax, ebx
            jnz else
            ; statements2
            jmp continue
else:
            ; statements3
continue:
```

if (a > b) { statements }

```
            mov eax, dword[a]
            mov ebx, dword[b]
            cmp eax, ebx
            jc continue         ; a< b
            jz continue         ; a== b
            ; statements
continue:
```

if (a < b) { statements }

```
          mov eax, dword[a]
          mov ebx, dword[b]
          cmp eax, ebx
          jnc continue
          ; statements
continue:
```

if (a >= b) { statements }

```
          mov eax, dword[a]
          mov ebx, dword[b]
          cmp eax, ebx
          jc continue
          ; statements
continue:
```

if (a <= b) { statements }

```
          mov eax, dword[a]
          mov ebx, dword[b]
          cmp eax, ebx
          jz perform      ; a == b
          jc perform      ; a < b
          jmp continue
perform:
          ; statements
continue:
```

17. Loops

while (a == b) { statements }

```
            mov eax, dword[a]
            mov ebx, dword[b]
while:      cmp eax, ebx
            jnz continue
            ; statements
            jmp while
continue:
```

for (expr1; a == b; expr3) { statements }

is equivalent to

```
expr1;
while (a==b) {
   statements;
   expr2;
}
```

for (i=0; i <100; i++) { statements }

```
            mov word[i], 0
for:        mov ax, word[i]
            cmp ax, 100
            jnc continue
            ; statements
            inc word[i]
            jmp for
continue:
```

do { statements } while (a == b)

```
dowhile:
          ; statements
          mov eax, dword[a]
          mov ebx, dword[b]
          cmp eax, ebx
          jz dowhile
continue:
```

for (i=0; i <100; i++) { statements }

using the LOOP instruction

```
          mov ecx, 100
for:
          ; statements
          LOOP for
continue:
```

Note that this is not exactly the same as the previous implementation.

Note also the following quote from the NASM manual:

" `LOOP` decrements its counter register (either `CX` or `ECX` – if one is not specified explicitly, the `BITS` setting dictates which is used) by one, and if the counter does not become zero as a result of this operation, it jumps to the given label. The jump has a range of 128 bytes.

`LOOPE` (or its synonym `LOOPZ`) adds the additional condition that it only jumps if the counter is nonzero *and* the zero flag is set. Similarly, `LOOPNE` (and `LOOPNZ`) jumps only if the counter is nonzero and the zero flag is clear."

"The `BITS` directive specifies whether NASM should generate code designed to run on a processor operating in 16–bit mode, or code designed to run on a processor operating in 32–bit mode. The syntax is `BITS 16` or `BITS 32` .

In most cases, you should not need to use `BITS` explicitly. The `aout` , `coff` , `elf` and `win32` object formats, which are designed for use in 32–bit operating systems, all cause NASM to select 32–bit mode by default."