Branching and Looping

Edited from the work of Ray Seyfarth

August 22, 2015
Branching and looping

- So far we have only written “straight line” code
- Conditional moves gave us an avenue for trivial if like structures.
- But we really need
  - To handle code structures like if/else. So we need both conditional and unconditional branch statements
  - We need loops
An unconditional jump is equivalent to a goto

But jumps are necessary in assembly, while high level languages could exist without goto

The unconditional jump looks like

cmp label

The label can be any label in the program’s text segment

We might think of parts of the text segment as functions

▶ The computer will let you jump anywhere
▶ You can try to jump to a label in the data segment, which hopefully will fail

The assembler will generate an instruction register (rip) relative location to jump

▶ The simplest form uses an 8 bit immediate: -128 to +127 bytes
▶ The next version is 32 bits: plus or minus 2 GB
▶ The short version takes up 2 bytes; the longer version 5 bytes
▶ The assembler figures this out for you (Yay)
Unconditional jumps can vary

It is possible to use an unconditional jump to simulate a conditional jump.
- It is possible to jump to an address stored in a register.
- We can control the value of the register using a conditional move.

```assembly
mov rax, a
mov rbx, b
cmovl rax, rbx ; rather jmp to b if the sign flag is set
jmp rax

a:
    .....  
    .....  
    jmp end

b:
    .....  
    .....  

end:
```
Unconditional jumps can vary

- Though it is simpler to just use a conditional jump.
- However you can construct an efficient switch statement by expanding this idea
  - You need an array of addresses and an index for the array to select which address to use for the jump
Unconditional jump used as a switch

```assembly
segment .data
switch: dq main.case0
dq main.case1
dq main.case2
i: dq 2

segment .text

global main ; tell linker about main

main: mov rax, [i] ; move i to rax
      jmp [switch+rax*8] ; switch (i)

.case0:
mov rbx, 100 ; go here if i == 0
jmp .end

.case1:
mov rbx, 101 ; go here if i == 1
jmp .end

.case2:
mov rbx, 102 ; go here if i == 2

.end: xor eax, eax
ret
```

© Edited from the work of Ray Seyfarth
Conditional jump

- First you need to execute an instruction which sets some flags
- Then you can use a conditional jump
- The general pattern is `jCC label`
- The CC means a condition code

<table>
<thead>
<tr>
<th>instruction</th>
<th>meaning</th>
<th>aliases</th>
<th>flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>jz</td>
<td>jump if zero</td>
<td>je</td>
<td>ZF=1</td>
</tr>
<tr>
<td>jnz</td>
<td>jump if not zero</td>
<td>jne</td>
<td>ZF=0</td>
</tr>
<tr>
<td>jg</td>
<td>jump if &gt; zero</td>
<td>jnle ja</td>
<td>ZF=0, SF=0</td>
</tr>
<tr>
<td>jge</td>
<td>jump if ≥ zero</td>
<td>jnl</td>
<td>SF=0</td>
</tr>
<tr>
<td>jl</td>
<td>jump if &lt; zero</td>
<td>jnge js</td>
<td>SF=1</td>
</tr>
<tr>
<td>jle</td>
<td>jump if ≤ zero</td>
<td>jng</td>
<td>ZF=1 or SF=1</td>
</tr>
<tr>
<td>jc</td>
<td>jump if carry</td>
<td>jb jnae</td>
<td>CF=1</td>
</tr>
<tr>
<td>jnc</td>
<td>jump if not carry</td>
<td>jae jnb</td>
<td>CF=0</td>
</tr>
</tbody>
</table>
Compare operation

- It can become cumbersome to always have to preform a calculation and store the result simply to use condition jump.
- This is where the compare operation comes in handy
  - `cmp`
- `cmp` takes 2 operand.
- `cmp` subtracts the second operand from the first and sets the appropriate flags.
- But, the result is not actually stored.
- At most one operand can be an immediate value.
Simple if statement

```c
if ( a < b ) {
    temp = a;
    a = b;
    b = temp;
}
```

```assembly
mov rax, [a]
mov rbx, [b]
cmp rax, rbx
jge in_order
mov [a], rbx
mov [b], rax
in_order:
```
If statement with an else clause

```c
if ( a < b ) {
    max = b;
} else {
    max = a;
}
```

```assembly
mov rax, [a]
mov rbx, [b]
cmp rax, rbx
jnl else
mov [max], rbx
jmp endif
else:  mov [max], rax
endif:
```
You can construct any form of loop using conditional jumps.

We will model our code after C’s loops: while, do...while and for.

We will also consider break and continue.

Break and continue can be avoided in C, though sometimes the result is less clear.

The same consideration applies for assembly loops as well.
Sum 1 to 1000

```c
sum = 0;
i = 1;
while ( i <= 100 )
{
    sum +=i;
    i++; 
}
```

Now the assembler version (no optimization done to keep things simple)
segment .data
sum dq 0
segment .text
global _start
_start:
    mov rcx,1 ; i=1
while:
    cmp rcx,100
    jg ewhile
    add [sum],rcx
    inc rcx
    jmp while
 ewhile:
    mov rax, 60
    xor rdi, rdi
    syscall
Counting 1 bits in a quad-word

```c
sum = 0;
i = 0;
while ( i < 64 )
{
    sum += data & 1;
    data = data >> 1;
    i++;
}
```

- There are much faster ways to do this
- But this is easy to understand and convert to assembly
Assume we have the following data segment:

    segment .data
    data dq 0xfedcba9876543210
    sum   dq 0
segment .text

global main

main:    mov rax, [data] ; rax holds the data
         xor ebx, ebx ; clear since setc will fill in bl
         xor ecx, ecx ; i = 0;
         xor edx, edx ; sum = 0;

while:   cmp rcx, 64 ; while ( i < 64 ) {
         jnl end_while ; requires testing on opposite
         bt rax, 0 ; data & 1
         setc bl ; move result of test to bl
         add edx, ebx ; sum += data & 1;
         shr rax, 1 ; data = data >> 1;
         inc rcx ; i++;
         jmp while ; end of the while loop

end_while:
         mov [sum], rdx ; save result in memory
         xor eax, eax ; return 0 from main
         ret
To be more true to the C-code, we could replace

```
bt rax, 0
setc bl
add edx, ebx
```

with

```
mov r8, rax
and r8, 1
add edx, r8d
```