Branching and Looping

Ray Seyfarth

August 10, 2011
Branching and looping

- So far we have only written “straight line” code
- Conditional moves helped spice things up
- In addition conditional moves kept the pipeline full
- But conditional moves are not always faster than branching
- But we need loops to process each bit in a register
- Repeated code can be faster, but there is a limit
- In the next chapter we will work with arrays
- Here we will need to process differing amounts of data
- Repeated code is too inflexible
- We need loops
- To handle code structures like if/else we need both conditional and unconditional branch statements
Outline

1. Unconditional jump
2. Conditional jump
3. Looping with conditional jumps
4. Loop instructions
5. Repeat string (array) instructions
Unconditional jump

- 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
  \[ \text{jmp label} \]
- The label can be any label in the program’s text segment
- Humans 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
Unconditional jumps can vary

- An unconditional jump can jump to a location specified by a register’s content or a memory location.
- You could use a conditional move to hold either of 2 locations in a register and jump to the proper location.
- 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
```
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</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>
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   [temp], rax
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 could 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
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
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
Code generated by gcc -O3 -S countbit.s

```assembly
movq    data(%rip), %rax
movl    $64, %ecx
xorl    %edx, %edx

.L2:

movq    %rax, %rsi
sarq    %rax
andl    $1, %esi
addq    %rsi, %rdx
subl    $1, %ecx
jne     .L2
```

- AT&T syntax: operands are reversed and names are more explicit
- The compiler counted down from 64
- Converted the loop to test at the bottom
- Loop has 2 fewer instructions
- Is it faster to use movq and andl?
Learning from the compiler

- The compiler writers know the instruction set very well
- Most likely movq and andl is faster
- Testing would tell if the other method is superior
- I also tried the compiler option “-funroll-all-loops”
- The compiler added up values for 8 bits in 1 loop iteration
- 8 bits in a 24 instruction loop vs 1 bit in a six instruction loop
- This makes it twice as fast, but the instructions use many different registers allowing parallel execution in 1 core
- Loop unrolling can help a lot with 16 registers
- Examining the generated code should mean than you do no worse
- Clever reorganization can beat the compiler
Do-while loops

- Strict translation of a while loop uses 2 jumps
- It save a jump to the top if you use a do-while loop

\[
\text{do } \{ \\
\text{    statements; } \\
\} \text{ while ( condition );}
\]

- A do-while loop always executes the loop body at least once
- You can always place an if statement around a do-while to make it behave like a while loop

\[
\text{if ( condition ) } \{ \\
\text{    do } \{ \\
\text{        statements; } \\
\} \text{ while ( condition ); } \\
\}
\]

- Don’t do this in C - let the compiler do it for you
Ugly C code to search through an array

```c
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;
```
Assembly code to search through an array

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

©2011 Ray Seyfarth
Counting loops

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

mov    rdx, [n] ; use rdx for n
xor    ecx, ecx ; i (rdx) = 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             ; too bad, loop has 2 jumps

end_for:

- We could use a test before the loop
- We could do loop unrolling
Loop instructions

- The CPU has instructions like `loop` and `loopne` which are designed for loops.
- They decrement `rcx` and do the branch if `rcx` is not 0.
- It is faster to use `dec` and `jnz` instead.
- The label must be within -128 to +127 bytes of `rip`.
- Probably pointless.

```
mov    ecx, [n]
sub    ecx, 1
more:  cmp    [data+rcx],al
loopne more
mov    [loc], ecx
```
The repeat instruction (rep) works in conjunction with string (array) instructions.

You first set rcx to be the number of repetitions.

You set rsi to the address of source data.

And set rdi to be the address of destination data.

Then you use a command like:

```
rep movsb
```

The previous command would copy an array of bytes.

Some string instructions include tests for early termination.

The string instructions can also be used without rep.
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

```assembly
mov   eax, 1
mov   ecx, 1000000
lea   rdi, [destination]
rep   stosd ; place 1000000 1’s in destination
```
Store instruction

- There are a collection of load string instructions which copy data from the address pointed at by `rsi` and increment (or decrement) `rsi`.
- Using `rep lodsb` seems pointless.
- The code below uses `lodsb` and optionally `stosb` to copy none carriage return characters.

```
lea rsi, [source]
lea rdi, [destination]
mov ecx, 1000000 ; number of iterations
more: lodsb ; get the next byte in al
cmp al, 13 ; if al is not 13 store al
je skip
stosb ; store al in destination
skip: sub ecx, 1 ; count down
jnz more
```
Scan instruction

- There are a collection of scan string instructions which scan data from the address pointed at by rsi and increment (or decrement) rsi.
- They compare data against al, ax, ...
- Below is a version of the C `strlen` function

```assembly
 segment .text
 global strlen

strlen:  cld          ; prepare to increment rdi
         mov  rcx, -1  ; maximum number of iterations
         xor  al, al   ; will scan for 0
         repne scasb  ; repeatedly scan for 0
         mov  rax, -2  ; start at -1, end 1 past the end
         sub  rax, rcx
         ret
```
The compare string instructions compare the data pointed at by rdi and rsi

The code below implements the C `memcmp` function

```assembly
segment .text
global memcmp

memcmp: mov rcx, rdx
    repe cmpsb       ; compare until end or difference
    cmp rcx, 0
    jz equal         ; reached the end
    movzx eax, byte [rdi-1]
    movsx ecx, byte [rsi-1]
    sub eax, ecx
    ret

equal: xor eax, eax
    ret
```
Setting and clearing the direction flag

- The string operations increment their addresses if the direction flag is 0
- They decrement their address is the direction flag is 1
- Use `cld` to prepare for increasing addresses
- Use `std` to prepare for decreasing addresses
- Functions are expected to leave the direction flag set to 0