Control Structures

ICS312

Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

Translating high-level structures

- When programming with high-level languages we are used to using high-level structures rather than just branches
  - In fact, many languages don't allow branches (i.e., "goto")
  - C/C++ does though!
- Therefore, it's useful to know how to translate these structures in assembly, so that we can use the same patterns as when writing, say, Java code
  - A compiler does these translations for us with high-level code
  - But in this course, it's not because we write assembly directly that we should ignore everything we've learned with high-level languages and embrace spaghetti code. Quite the opposite.
- Let's start with the most common high-level control structure: if-then-else
  - We already did this in the previous set of slides

If-then-Else

- A generic if-then-else construct:

```
if (condition) then
    then_block
else
    else_block
```

- Translation into x86 assembly:

```
; instructions to set flags (e.g., cmp ...)
jxx       else_block   ; xx so that branch if
                       ; condition is false
; code for the then block
jmp endif
else_block:
; code for the else block
endif:
```

No Else?

- A generic if-then-else construct:

```
if (condition) then
    then_block
```

- Translation into x86 assembly:

```
; instructions to set flags (e.g., cmp ...)
jxx       endif   ; select xx so that branch
                  ; if condition is false
; code for the then block
endif:
```

Else but No Else?

- One can use the pattern in the previous slide even with an else, by doing possibly one extra **mov** instruction

```
if (ebx  == 12) {
  eax = 10;
} else {
  eax = 20;
}
```

```
  mov eax, 20
  cmp ebx, 12
  jne endif
  mov eax, 10
endif:
```

- If ebx == 12, the assembly code does an extra **mov** instruction to overwrite the “wrong” value

Short-circuiting

```
if (((ecx <= 2) || (ebx > 1)) && (edx == 8)){
    eax = 10;
} else {
    eax = 20;
}
```

- Translating the above to assembly isn't difficult, but some ways of doing it are much easier than others
- A simple way is to try short-circuiting
- For instance, in the above, one can first check wether edx == 8, and if not then we know the condition is false
- Let's write the code live right now…

Short-circuiting

```
if (((ecx <= 2) || (ebx > 1)) && (edx == 8)){
    eax = 10;
} else {
    eax = 20;
}
```

```
cmp edx, 8
jne else
cmp ecx, 2
jle then
cmp ebx, 1
jle else
then:
   mov eax, 10
   jmp end
else:
   mov eax, 20
end:
```

For Loops

- Let's translate the following loop:

```
sum = 0;
for (i = 0; i <= 10; i++)
      sum += i
```

- Translation

```
    mov eax, 0          ; eax is sum
    mov ebx, 0          ; ebx is i
loop_start:
    cmp ebx, 10         ; compare i and 10
    jg    loop_end      ; if (i>10) go loop_end
    add   eax, ebx      ; sum += i
    inc   ebx           ; i++
    jmp loop_start      ; goto loop_start
loop_end:
```

The loop instruction

- It turns out that, for convenience, the x86 assembly provides instructions to do loops!
  - The book lists 3, but we'll talk only about the 1st one
- There is a loop instruction
- It is used as: `loop <label>`
- and does
  - Decrement ecx (ecx **has** to be the loop index)
  - If (ecx != 0), branch to the label
    - Only a short jump!
- Let's try to do the loop in our previous example

For Loops

- Let's translate the following loop:

```
sum = 0;
for (i = 1; i <= 10; i++)
        sum += i
```

- The x86 loop instruction requires that
  - The loop index be stored in ecx
  - The loop index be decremented
  - The loop exits when the loop index is equal to zero
- Given this, we really have to think of this loop in reverse

```
sum = 0
for (i = 10; i > 0; i--)
        sum += i
```

- This loop is equivalent to the previous one, but now it can be directly translated to assembly using the loop instruction

Using the loop Instruction

- Here is our “reversed” loop

```
sum = 0
for (i = 10; i > 0; i--)
    sum += i
```

- And the translation

```
    mov  eax, 0          ; eax is sum
    mov  ecx, 10         ; ecx is i
loop_start:
    add eax, ecx         ; sum += i
    loop loop_start      ; if i > 0 then
                         ; go to loop_start
```

Using the loop instruction?

- It's totally up to you whether you use the loop instruction or not
  - You should know it though
- In some cases it's very convenient
- In other cases the mental gymnastics needed to modify the loop to use the loop instruction is tedious and error-prone

While Loops

- A generic while loop

```
while (condition) {
    body
}
```

- Translated as:

```
while:
    ; instructions to set flags (e.g., cmp...)
    jxx   end_while   ; branches if
                      ; condition=false
    ; body of loop
    jmp   while
end_while:
```

Do While Loops

- A generic do while loop

```
do {
    body
} while (condition)
```

- Translated as:

```
do:
    ; body of loop
    ; instructions to set flags (e.g., cmp...)
    jxx   do    ; branches if condition=true
```

The Compiler

- The compiler does these types of translations for us when we give it high level code
- It can be enlightening to look at the assembly code that the compiler produces
- Let's look at the "Disassembling code with NASM" reading on the course Web site…

Computing Prime Numbers

- The book has an example of an assembly program that computes prime numbers
- Principle:
  - Try possible prime numbers in increasing order starting at 5
  - Skip even numbers
  - Test whether the possible prime number (the “guess”) is divisible by any number other than 1 and itself
    - If yes, then it’s not a prime, otherwise, it is
- You may want to look at it

Computing the Sum of an Array

- Let's write a (fragment of a) program that computes the sum of an array
- Let us assume that the array is "declared" in the .bss segment as:
  - `array resd 20 ; 20 4-byte values`
- And let us assume that its elements have been set to some values
- We want to compute the numerical sum of all its elements into register ebx
- Let's try to write the code together live...

Computing the Sum of an Array

```
        mov   ebx, 0        ; ebx = 0 (sum)
        mov   ecx, 0        ; ecx = 0 (loop index)

main_loop:
        ; Done?
        cmp   ecx, 20.      ; compare ecx to 20
        je    end_main_loop ; if == 20, we're done
        ; Compute address of current element
        mov   eax, array    ; eax points to 1st element
        mov   edx, ecx      ; edx = ecx (loop index)
        imul  edx, 4        ; edx = 4 * ecx
        add   eax, edx      ; eax = array + 4 * ecx
        ; Increment the sum
        add   ebx, [eax]    ; sum += element
        ; Move to the next element
        inc   ecx           ; ecx ++
        jmp   main_loop     ; loop back
end_main_loop:
```

Computing the Sum of an Array

```
; SHORTER/SIMPLER VERSION WITH A POINTER

        mov   ebx, 0        ; ebx = 0 (sum)
        mov   ecx, 0        ; ecx = 0 (loop index)
        mov   eax, array    ; eax = array
main_loop:
        ; Done?
        cmp   ecx, 20       ; compare ecx to 20
        je    end_main_loop ; if == 20, we're done
        ; Increment the sum
        add   ebx, [eax]    ; sum += element
        ; Move to the next element
        add   eax, 4        ; eax += 4
        inc   ecx           ; ecx ++
        jmp   main_loop     ; loop back
end_main_loop:
```

Going Through a Linked List?

- How about going through a linked list of integers and computing the sum of the integers?
  - Each item in the list contains: (i) **`item.value`**: a 4-byte integer; (ii) **`item.next`**: the 4-byte address of the next item (or null)
- Let's try to live code if time…
- The next slide has a version….

Going Through a Linked List

```
   mov   eax, 0              ; The sum
   mov   ecx, L              ; Our pointer to an item

loop_begin:

   cmp   ecx, 0              ; If nullptr…
   je    end_loop            ; then we're done

   add   eax, [ecx]          ; sum += item.value
   mov   ecx, [ecx + 4]      ; pointer = item.next

   jmp   loop_begin          ; jump back
end_loop:
```

How about “binning” integers

- (Let’s do this if time)
- Say you’re given a stream of positive integers between 0 and 299, and the last integer is -1
- Print how many integers are between 0 and 9, how many between 10 and 19, …., how many are between 290 and 299

Important Takeaways

- One should avoid spaghetti code in assembly even though we have a “goto”
- High-level control structures are just branch instructions, but using some known/standard patterns
- In an if-then-else one must not forget to “jump over” the second clause
- For for loops, using the `loop` instruction is optional
- While and do-while loops are probably the easiest to implement!

Conclusion

- Make sure you understand the “sum of an array example” 100%
- Writing control structures in assembly isn’t as easy as in high-level languages
- But as long as you follow consistent patterns and use reasonable label names it should be manageable

- We have an optional homework #5…
- Let’s also look at some of the practice problems…
- We’ll have an **in-class quiz** on this module next week
- This completes all material for Midterm #2