Control Structures

ICS312

Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

Translating high-level structures

- We are used to using high-level structures rather than just branches
- Therefore, it's useful to know how to translate these structures in assembly, so that we can just use the same patterns as when writing, say, C code
  - A compiler does such translations for us
- 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:
```

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
- The instruction is called loop
- It is used as: loop   <label>
- and does
  - Decrement ecx   (ecx **has** to be the loop index)
  - If (ecx != 0), branches to the label
- Let's try to do the loop in our previous example

For Loops

- Let's translate the following loop:

```
sum = 0;
for (i = 0; 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
```

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
```

Computing Prime Numbers

- The book has an example of an assembly program that computes prime numbers
- Let's look at it in detail
- 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

Computing Primes: High-Level

```
unsigned int guess;
unsigned int factor;
unsigned int limit;

printf(“Find primes up to: “);
scanf(“%u”,&limit);
printf(“2\n3\n”);                  // prints the first 2 obvious primes
guess = 5;                         // we start the guess at 5
while (guess <= limit) {           // look for numbers up to the limit
  factor = 3;                      // initial potential factor
  // we only look at potential factors < sqrt(guess)
  while ( factor*factor < guess  &&  guess % factor != 0 )
    factor += 2;             // skip even factors
  if ( guess % factor != 0 ) // we never found a factor
    printf(“%d\n”,guess);    // print the number, which is prime!
  guess += 2;       // skip even numbers since they are never prime
}
```

Computing Primes in Assembly

```
unsigned int guess;
unsigned int factor;
unsigned int limit;
```

bss segment

```
printf("Find primes up to: ");
scanf("%u",&limit);
printf("2\n3\n");          // prints the first 2 obvious primes
guess = 5;                 // we start the guess at 5
```

data segment (message)
easy text segment

```
while (guess <= limit) {
  factor = 3;              // look for a possible factor
  // we only look at factors < sqrt(guess)
  while ( factor*factor < guess && guess % factor != 0 )
    factor += 2;
  if ( guess % factor != 0 )         // we never found a factor
    printf("%d\n",guess);
  guess += 2;              // skip even numbers
}
```

more difficult text segment

Computing Primes in Assembly

```
unsigned int guess;
unsigned int factor;
unsigned int limit;
```

bss segment

```
printf("Find primes up to: ");
scanf("%u",&limit);
printf("2\n3\n");          // prints the first 2 obvious primes
guess = 5;                 // we start the guess at 5
```

data segment (message)
easy text segment

```
%include "asm_io.inc"
segment .data
Message   db    "Find primes up to: ", 0
segment .bss
Limit     resd      1     ; 4-byte int
Guess     resd      1     ; 4-byte int
segment .text
    global asm_main
asm_main:
          enter     0, 0
          pusha
```

```
mov    eax, Message            ; print the message
call   print_string
call   read_int                ; read Limit
mov    [Limit], eax
mov    eax, 2                  ; print "2\n"
call   print_int
call   print_nl
mov    eax, 3                  ; print "3\n"
call   print_int
call   print_nl
mov    dword  [Guess], 5       ; Guess = 5
```

Computing Primes in Assembly

```
while (guess <= limit) {
. . .
}
```

unsigned numbers

```
while_limit:
        mov     eax, [Guess]
        cmp     eax, [Limit]         ; compare Guess and Limit
        jnbe    end_while_limit      ; If !(Guess <= Limit) Goto end_while_limit

        . . .                        ; body of the loop goes here

        jmp     while_limit
end_while_limit:

        popa                         ; clean up
        mov     eax, 0               ; clean up
        leave                        ; clean up
        ret                          ; clean up
```

Computing Primes in Assembly

```
        mov     ebx, 3              ; ebx is factor
while_factor:
        mov     eax, ebx            ; eax = factor
        mul     eax                 ; edx:eax = factor * factor
        cmp     edx, 0              ; compare edx and 0
        jne     end_while_factor    ; factor too big
        cmp     eax, [Guess]        ; compare  factor*factor and guess
        je      endif               ; if == then number is perfect square
        jnb     end_while_factor    ; if !< then the number is prime
        mov     edx, 0              ; edx = 0
        mov     eax, [Guess]        ; eax = [Guess]
        div     ebx                 ; divide edx:eax by factor
        cmp     edx, 0              ; compare the remainder with 0
        je      endif               ; if == 0 goto endif
        add     ebx, 2              ; factor += 2
        jmp     while_factor        ; loop back
end_while_factor:
        mov     eax, [Guess]        ; print guess
        call    print_int           ; print guess
        call    print_nl            ; print guess
endif:
        add     dword [Guess], 2    ; guess += 2
```

```
factor = 3;        // look for a possible factor
 // we only look at factors < sqrt(guess)
 while ( factor*factor < guess &&
         guess % factor != 0 )
   factor += 2;
 if ( guess % factor != 0 ) // no found factor
   printf("%d\n",guess);
 guess += 2;    // skip e
```

if edx != 0, then we're too big

don't forget to initialize edx

We don't chose eax for factor because eax is used by a lot of functions/routines

The Book's Program

- There are a few differences between this program and the one in the book:
  - e.g., Instead of checking that edx=0 after the multiplication, the book simple checks for overflow with "jo end_while_factor"
    - When doing a multiplication of 2 32-bit integers and getting the 64-bit result in edx:eax, the OF flag is set if the result does not fit solely in eax
    - In the previous program I just explicitly tested that indeed all bits of edx where zeros
- Note that we do not have a straight translation from the C code
  - We do not test (guess % factor) twice like in the C code!
  - This is a typical "assembly optimization"
    - Can of course lead to bugs

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 ; An array of 20 double words
- 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:
    ; 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 ++
    ; Done?
    cmp   ecx, 20      ; compare ecx to 20
    jl    main_loop    ; if <20, then loop back
```

Computing the Sum of an Array

```
; SHORTER/SIMPLER VERSION

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

main_loop:
      ; Increment the sum
      add   ebx, [eax] ; sum += element
      ; Move to the next element
      add   eax, 4     ; eax += 4
      inc   ecx        ; ecx ++
      ; Done?
      cmp   ecx, 20    ; compare ecx to 20
      jl    main_loop  ; if <20, then loop back
```

Conclusion

- Make sure you understand the prime number example 100%
- 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