

Basic Assembly Language II

ICS312 - Spring 2009 Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

Control Structures

- So far we have seen instructions to
 - Move data back and forth between memory and registers
 - Do some data conversion
 - Perform arithmetic operation on that data
- Now we're going to learn about **control structures**, that is instructions that modify the order in which instructions are executed
 - i.e., we not necessarily execute the next instruction
- High-level programming languages provide control structures
 - for loops, while loop, if-then-else statements, etc.
- Assembly language provides much more basic control structures
- Mostly it provides a **goto!**
 - A really infamous instruction, that causes horrendous "spaghetti code"
- Luckily, high-level control structures can be cleanly translated into assembly code
- Therefore, one can write non-spaghetti assembly! (sort of)

Comparisons

- Control structures essentially decide which instruction should be executed next based on comparisons of data items
- In assembly, the result of a comparison is stored in the bits of the FLAGS register
- The basic comparison instruction is **cmp**
- **cmp** subtracts one operand from another, and sets the bits of FLAGS accordingly, **but the result of the subtraction is not stored anywhere**
- Other arithmetic instructions also set bits of FLAGS (add, sub, mul, etc.)

Unsigned Integers

- When you use unsigned integers the bits in the FLAGS register (also called "flags") that are important are:
 - ZF: The Zero Flag (set to 1 if result is 0)
 - CF: The Carry Flag
 - During an arithmetic operation, used to detect overflow or to do clever arithmetic since it may denote a carry or a borrow
- Consider: **cmp a, b** (which computes a-b)
 - If a = b: ZF is set, CF is not set
 - If a < b: ZF is not set, CF is set (borrow)
 - If you were computing the difference for real, this would mean an error!
 - If a > b: ZF is not set, CF is not set
- **Therefore, by looking at ZF and CF you can determine the result of the comparison!**
 - We'll see how we "look" at the flags shortly

Signed Integers

- For signed integers you should care about three flags:
 - ZF: zero flag
 - OF: overflow flag (set to 1 if the result overflows or underflows)
 - SF: sign flag (set to 1 if the result is negative)
- Consider: **cmp a, b** (which computes a-b)
 - If a = b: ZF is set, OF is not set, SF is not set
 - If a < b: ZF is not set, and SF ≠ OF
 - If a > b: ZF is not set, and SF = OF
- **Therefore, by looking at ZF, SF, and OF you can determine the result of the comparison!**

Signed Integers: SF and OF???

- Why do we have this odd relationship between SF and OF?
- Consider two signed integers a and b, and remember that we compute (a-b)
- If a < b
 - If there is no overflow, then (a-b) is a negative number!
 - If there is overflow, then (a-b) is (erroneously) a positive number
 - Therefore, in both cases SF ≠ OF
- If a > b
 - If there is no overflow, the (correct) result is positive
 - If there is an overflow, the (incorrect) result is negative
 - Therefore, in both cases SF = OF

Signed Integers: SF and OF???

- Example: $a = 80h$ (-128d), $b = 23h$ (+35d) ($a < b$)
 - $a - b = a + (-b) = 80h + DDh = 15Dh$
 - dropping the 1, we get 5Dh (+93d), which is erroneously positive!
 - So, SF=0 and OF=1
- Example: $a = F3h$ (-13d), $b = 23h$ (+35d) ($a < b$)
 - $a - b = a + (-b) = F3h + DDh = D0h$ (-48d)
 - D0h is negative and we have no overflow (in range)
 - So, SF=1 and OF=0
- Example: $a = F3h$ (-13d), $b = 82h$ (-126d) ($a > b$)
 - $a - b = a + (-b) = F3h + 7Eh = 171h$
 - dropping the 1, we get 71h (+113d), which is positive and we have no overflow
 - So, SF=0 and OF=0
- Example: $a = 70h$ (112d), $b = D8h$ (-40d) ($a > b$)
 - $a - b = a + (-b) = 70h + 28h = 98h$, which is erroneously negative
 - So, SF=1 and OF=1

In-Class Exercise

- What are the ZF, CF, SF, and OF flags for “comp a,b” for the following values
- $a = 0F3h$ and $b = 019h$
- $a = 074h$ and $b = 082h$
- $a = 0A3h$ and $b = 071h$

In-Class Exercise

- $a = 0F3h$ and $b = 019h$
 - ZF = 0
 - CF? (thinking of numbers as unsigned)
 - $a - b = 0F3h - 019h =$ something that's still >0
 - CF=0
 - SF? (thinking of numbers as signed)
 - $a + (-b) = F3h + E7h = 1DAh$, drop the 1
 - DAh is negative
 - SF = 1
 - OF? (thinking of numbers as signed)
 - a is negative, b is positive, DA is negative, we're good
 - OF = 0

In-Class Exercise

- $a = 074h$ and $b = 082h$
 - ZF = 0
 - CF? (thinking of numbers as unsigned)
 - $a - b = 074h - 082h =$ something that's <0
 - CF=1
 - SF? (thinking of numbers as signed)
 - $a + (-b) = 74h + 7Eh = F2h$
 - F2h is negative
 - SF = 1
 - OF? (thinking of numbers as signed)
 - a is positive, b is negative, F2 is erroneously negative
 - OF = 1

In-Class Exercise

- $a = 0A3h$ and $b = 071h$
 - ZF = 0
 - CF? (thinking of numbers as unsigned)
 - $a - b = 0A3h - 71h =$ something that's >0
 - CF=0
 - SF? (thinking of numbers as signed)
 - $a + (-b) = A3h + 8Fh = 152h$, drop the 1
 - 52h is positive
 - SF = 0
 - OF? (thinking of numbers as signed)
 - a is negative, b is positive, 52 is erroneously positive
 - OF = 1

The FLAGS register

- Is it very important to remember that many instructions change the bits of the FLAGS register
- So you should “act” on flag values immediately, and not expect them to remain unchanged inside FLAGS
 - or you can save them by-hand for later use perhaps

Summary

	cmp a,b	ZF	CF	OF	SF
unsigned	a=b	1	0	/	/
	a<b	0	1	/	/
	a>b	0	0	/	/
signed	a=b	1	/	0	0
	a<b	0	/	v	!v
	a>b	0	/	v	v

Branch Instructions

- A “branch” is basically a “goto” that says: instead of executing the next instruction, go execute that other one
- Two types of branches
 - **Unconditional** (often called a “jump”)
 - always branches
 - **Conditional**
 - branches only when some condition is true

The JMP Instruction

- JMP allows you to “jump” to a **code label**
- Example:

```

...
add eax, ebx
jmp here
sub al, bl
movsx ax, al
here:
call print_int
...
    
```

This instruction will never be executed!

The JMP Instruction

- The ability to jump to a **label** in the assembly code is convenient
- In machine code there is no such thing as a label: only addresses
- So one would constantly have to compute addresses by hand
 - e.g., “jump to the instruction +4319 bytes from here in the source code”
 - e.g., “jump to the instruction -18 bytes from here in the source code”
 - This is what programmers way back when used to do by hand, using **signed displacements in bytes**
 - The displacements are added to the EIP register (program counter)
- There are three versions of the JMP instruction in machine code:
 - **Short jump**: Can only jump to an instruction that is within 128 bytes in memory of the jump instruction (1-byte displacement)
 - **Near jump**: 4-byte displacement (any location in the code segment)
 - **Far jump**: very rare jump to another code segment
 - We won't use this at all

The JMP Instruction

- A **short jump**:


```

jmp label
or jmp short label
            
```
- A **near jump**:


```

jmp near label
            
```
- Why do we even have this?
 - Remember that instructions are encoded in binary
 - To jump one needs to encode the number of bytes to add/subtract to the program counter
 - If this number is large, we need many bits to encode it
 - If this number is small, we want to use few bits so that our program takes less space in memory
 - i.e., the encoding of a short jmp instruction takes fewer bits than the encoding of a near jmp instruction (3 bytes less)
 - In a code that has 100,000 near jumps, if you can replace 50% of them by short jumps, you save ~150KB (in the size of the executable)

Conditional Branches

- There is a large set of conditional branch instructions
- The simple ones just branch (or not) depending on the value of one of the flags:
 - ZF, OF, SF, CF, PF
 - PF: Parity Flag
 - Set to 0 if the number of bits set to 1 in the lower 8-bit of the “result” is odd, to 1 otherwise

Simple Conditional Branches

- JZ** branches if ZF is set
- JNZ** branches if ZF is unset
- JO** branches if OF is set
- JNO** branches if OF is unset
- JS** branches if SF is set
- JNS** branches if SF is unset
- JC** branches if CF is set
- JNC** branches if CF is unset
- JP** branches if PF is set
- JNP** branches if PF is unset

Example

- Consider the following C-like code


```
if (EAX == 0)
    EBX = 1;
else
    EBX = 2;
```
- Here it is in x86 assembler


```
cmp    eax, 0        ; do the comparison
jz     thenblock     ; if = 0, then goto thenblock
mov    ebx, 2        ; else clause
jmp    next          ; jump over the then clause
thenblock:
mov    ebx, 1        ; then clause
next:
```
- Could use `jnz` and be the other way around

Another Example

- Say we have the following C code (let us assume that EAX is signed)


```
if (EAX >= 5)
    EBX = 1;
else
    EAX = 2;
```
- This is much less straightforward
- Let's go back to our table for signed numbers

	cmp a,b	ZF	OF	SF
signed	a=b	1	0	0
	a<b	0	v	!v
	a>b	0	v	v

After executing `cmp eax, 5`
 if (OF = SF) then a >= b

Another Example

- `a >= b` if (OF = SF)
- Skeleton program


```
cmp    eax, 5        Comparison
                        Testing relevant flags
                        ???
thenblock:
mov    ebx, 1        "Then" block
jmp    end
elseblock:
mov    ebx, 2        "Else" block
end:
```

Another Example

- `a >= b` if (OF = SF)
- Program:


```
cmp    eax, 5        ; do the comparison
jo     oset           ; if OF = 1 goto oset
js     elseblock      ; (OF=0) and (SF = 1) goto elseblock
jmp    thenblock      ; (OF=0) and (SF=0) goto thenblock
oset:
jns    elseblock      ; (OF=1) and (SF = 0) goto elseblock
jmp    thenblock      ; (OF=1) and (SF=1) goto thenblock
thenblock:
mov    ebx, 1
jmp    end
elseblock:
mov    ebx, 2
end:
```

let's check that it works

Another Example

- Program:


```
cmp    eax, 5        ; do the comparison
jo     oset           ; if OF = 1 goto oset
js     elseblock      ; (OF=0) and (SF = 1) goto elseblock
jmp    thenblock      ; (OF=0) and (SF=0) goto thenblock
oset:
jns    elseblock      ; (OF=1) and (SF = 0) goto elseblock
jmp    thenblock      ; (OF=1) and (SF=1) goto thenblock
thenblock:
mov    ebx, 1
jmp    end
elseblock:
mov    ebx, 2
end:
```

Unneeded instruction, we can just "fall through"

The book has the same example, but their solution is the other way around

A bit too hard?

- One can play tricks by putting the else block before the then block
 - See example in the book
- The previous two examples are really awkward, and it's very easy to introduce bugs
- Consequently, x86 assembly provides other branch instructions to make our life much easier :)
- Let's look at these instructions

More branches

cmp x, y			
signed		unsigned	
Instruction	branches if	Instruction	branches if
JE	x = y	JE	x = y
JNE	x != y	JNE	x != y
JL, JNGE	x < y	JB, JNAE	x < y
JLE, JNG	x <= y	JBE, JNA	x <= y
JG, JNLE	x > y	JA, JNBE	x > y
JGE, JNL	x >= y	JAE, JNB	x >= y

Redoing our Example

```
if (EAX >= 5)
    EBX = 1;
else
    EAX = 2;

    cmp  eax, 5
    jge  thenblock
    mov  eax, 2
    jmp  end
thenblock:
    mov  ebx, 1
end:
```

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 than when writing, say, C code
 - A compiler does such translations
- Let's start with a high-level control structure we just talked about: if-then-else

If-then-Else

- A generic if-the-else construct:

```
if (condition) then
    then_block
else
    else_block;
```
- Translation into x86 assembly:

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

No Else?

- A generic if-the-else construct:

```
if (condition) then
    then_block
```
- Translation into x86 assembly:

```
; instructions to set flags (e.g., cmp ...)
jxx  endif; ; select xx so that branches if condition==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) goto end_loop
    add eax, ebx ; sum += i
    inc ebx     ; i++
    jmp loop_start ; goto loop
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 called 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 exists 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, 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 in C

```

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) {
    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
}
    
```

Computing Primes in Assembly

```

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) {
    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
}
            
```

bss segment

data segment (message)
easy text segment

more difficult text segment

Computing Primes in Assembly

```

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
            
```

bss segment

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_limit:
    mov eax,[Guess]
    cmp eax,[Limit] ; compare Guess and Limit
    jnb end_while_limit ; If !(Guess <= Limit) Goto end_while_limit
    ... ; body of the loop goes here
end_while_limit:
    popa
    mov eax,0 ; clean up
    leave ; clean up
    ret ; clean up
    
```

```

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

unsigned numbers

Computing Primes in Assembly

```

while_factor:
    mov ebx,3 ; ebx is factor
    mov eax,ebx ; eax = factor
    mul eax ; edx:eax = factor * factor
    cmp edx,0 ; compare edx and 0
    jne endif ; factor too big
    cmp eax,[Guess] ; compare factor*factor and guess
    jnb endif ; if !< goto endif (factor too big)
    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 end_while_factor ; if == 0 goto end_while_factor
    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 ) // we never found a factor
    printf("%d\n",guess);
guess += 2; // skip even numbers
            
```

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 simply 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` were 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

Homework

- Homework #4 will be posted shortly...