Jumps and Branches

ICS312

Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

Modifying Instruction Flow

- 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 instructions that modify the order in which instructions are executed
  - i.e., we not always execute the next instruction
- High-level programming languages provide control structures
  - for loops, while loop, if-then-else statements, etc.
- Assembly language basically provides a goto
  - An infamous instruction, that causes “spaghetti code”

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
        . . .
```

These instructions 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

- The JMP instruction is an unconditional branch
- We also have conditional branch instructions
- These instructions jump to an address in the code segment (i.e., a label) based on the content of the FLAGS register
- As a programmer you don't modify the FLAGS register, instead it is updated by
  - All instructions that perform arithmetic operations
  - The cmp instruction, which subtracts one operand from another but doesn't store the result anywhere

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

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

Simple Conditional Branches

- There is a large set of conditional branch instructions that act based on bits in the FLAGS register
- 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 is SF is set

JNS branches is 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 wither register-like variables

```
if  (EAX == 0)
  EBX = 1;
else
  EBX = 2;
```

- Here it is in x86 assembly

```
        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 contains a value that we interpret as **signed**)

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

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

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

Comparison

Testing relevant flags

“Then” block

“Else” block

Another Example (continued)

- 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:
```

Another Example (continued)

```
        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
  EBX = 2;
```

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

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

Conclusion

- In the next set of lecture notes we'll see how to translate high-level control structures (if-then-else, while, for, etc.) into assembly based on what we just described
  - We've basically seen if-the-else already