Introduction to Intel x86-64 Assembly, Architecture, Applications, & Alliteration

Xeno Kovah – 2014
xkovah at gmail
All materials is licensed under a Creative Commons “Share Alike” license.

- http://creativecommons.org/licenses/by-sa/3.0/

Attribution condition: You must indicate that derivative work

"Is derived from Xeno Kovah’s ‘Intro x86-64’ class, available at http://OpenSecurityTraining.info/IntroX86-64.html"
Control Flow

- Two forms of control flow
  - Conditional - go somewhere if a condition is met. Think “if”s, switches, loops
  - Unconditional - go somewhere no matter what. Procedure calls, goto, exceptions, interrupts.

- We've already seen procedure calls manifest themselves as call/ret, let's see how goto manifests itself in asm.
// Goto example
#include <stdio.h>

int main()
{
    goto mylabel;
    printf("skipped\n");
    mylabel:
    printf("goto ftw!\n");
    return 0xf00d;
}
JMP - Jump

• Change rip to the given address
• Main forms of the address
  – Short relative (1 byte displacement from end of the instruction)
    • "jmp 000000140001023" doesn’t have the number 000000140001023 anywhere in it, it’s really “jmp 0x0E bytes forward”
    • Some disassemblers will indicate this with a mnemonic by writing it as “jmp short”
  – Near relative (4 byte displacement from current eip)
  – Absolute (hardcoded address in instruction)
  – Absolute Indirect (address calculated with r/m32)
• jmp -2 == infinite loop for short relative jmp :)
GotoExample.c takeaways

• goto == jmp in asm :)

//Goto example
#include <stdio.h>
int main(){
    goto mylabel;
    printf("skipped\n");
mylabel:
    printf("goto ftw!\n");
    return 0xf00d;
}

main:
0000000140001010  sub        rsp,28h
0000000140001014  jmp        0000000140001023
0000000140001016  lea         rcx,[40006000h]
000000014000101D  call        qword ptr [40008368h]
$mylabel:
0000000140001023  lea         rcx,[40006010h]
000000014000102A  call        qword ptr [40008368h]
0000000140001030  mov       eax,0F00Dh
0000000140001035  add        rsp,28h
0000000140001039  ret
int main()
{
    int a=1, b=2;
    if(a == b){
        return 1;
    }
    if(a > b){
        return 2;
    }
    if(a < b){
        return 3;
    }
    return 0xdefea7;
}
Ghost of Xmas Future: Tools you won’t get to use today generate a Control Flow Graph (CFG) which looks much nicer. Not that that helps you. Just sayin’ :)

---

```
main: proc near

var a: dword ptr -16h
var b: dword ptr -16h

sub rsp, 16h
mov [rsp+16h+var a], 1
mov [rsp+16h+var a], 7
mov eax, [rsp+16h+var a]
cmp [rsp+16h+var a], eax
inc short loc t4000103
```

---

```
loc t4000103:
  mov eax, [rsp+16h+var a]
cmp [rsp+16h+var a], eax
jle short loc t4000103
```

---

```
loc t4000103:
  mov eax, [rsp+16h+var a]
cmp [rsp+16h+var a], eax
je short loc t4000103
```

---

```
loc t4000103:
  mov eax, 3
```

---

```
loc t4000103:
  mov eax, 7
```

---

```
loc t4000103:
  mov eax, [rsp+16h+var a]
cmp [rsp+16h+var a], eax
jle short loc t4000103
```

---

```
loc t4000103:
  mov eax, [rsp+16h+var a]
cmp [rsp+16h+var a], eax
```

---

```
loc t4000103:
  mov eax, [rsp+16h+var a]
cmp [rsp+16h+var a], eax
```

---

```
loc t4000103:
  add rsp, 16h
ret
main: cendp
```
Jcc - Jump If Condition Is Met

• There are more than 4 pages of conditional jump types! Luckily a bunch of them are synonyms for each other.
• JNE == JNZ (Jump if not equal, Jump if not zero, both check if the Zero Flag (ZF) == 0)

Book p. 137
Some Notable Jcc Instructions

- JZ/JE: if ZF == 1
- JNZ/JNE: if ZF == 0
- JLE/JNG: if ZF == 1 or SF != OF
- JGE/JNL: if SF == OF
- JBE: if CF == 1 OR ZF == 1
- JB: if CF == 1

Note: Don’t get hung up on memorizing which flags are set for what. More often than not, you will be running code in a debugger, not just reading it. In the debugger you can just look at rflags and/or watch whether it takes a jump.
Flag setting

• Before you can do a conditional jump, you need something to set the condition flags for you.
• Typically done with CMP, TEST, or whatever instructions are already inline and happen to have flag-setting side-effects
CMP - Compare Two Operands

- “The comparison is performed by subtracting the second operand from the first operand and then setting the status flags in the same manner as the SUB instruction.”
- What’s the difference from just doing SUB? Difference is that with SUB the result has to be stored somewhere. With CMP the result is computed, the flags are set, but the result is discarded. Thus this only sets flags and doesn’t mess up any of your registers.
- Modifies CF, OF, SF, ZF, AF, and PF
- (as we already saw, SUB modifies all those too)

Book p. 138
IfExample.c takeaways

- Conditional logic, like if statements, manifests in assembly as conditional jumps (Jcc). “If condition true, jump there, else fall through”
- Conditions involving in/equality are often checked with a CMP instruction, which is the same thing as a SUB, but it just throws the results away after the relevant RFLAGS bits are set
- The RFLAGS bits are fundamentally what are checked by the Jccs

```c
int main(){
    int a=1, b=2;
    if(a == b){
        return 1;
    }
    if(a > b){
        return 2;
    }
    if(a < b){
        return 3;
    }
    return 0xdefea7;
}
```
#define MASK 0x100

int main(){
    int a=0x1301;
    if(a & MASK){
        return 1;
    }
    else{
        return 2;
    }
}
TEST - Logical Compare

• “Computes the bit-wise logical AND of first operand (source 1 operand) and the second operand (source 2 operand) and sets the SF, ZF, and PF status flags according to the result.”

• Like CMP - sets flags, and throws away the result

Book p. 232
BitmaskExample.c takeaways

• Conditions depending on bit tests (which is often expressed with boolean logic instructions) will often see the RFLAGS set with the CMP instruction. CMP is like AND, but throws the results away.
• The reason for the extraneous jmp here is because it’s unoptimized code so it’s following a simpler set of asm construction rules.

```c
#define MASK 0x100

int main(){
    int a=0x1301;
    if(a & MASK){
        return 1;
    } else{
        return 2;
    }
}
```

```assembly
main:
0000000140001010 sub rsp,18h
0000000140001014 mov dword ptr [rsp],1301h
000000014000101B mov eax,dword ptr [rsp]
000000014000101E and eax,100h
0000000140001023 test eax,eax
0000000140001025 je 0000000140001030
0000000140001027 mov eax,1
000000014000102C jmp 0000000140001035
000000014000102E jmp 0000000140001035
0000000140001030 mov eax,2
0000000140001035 add rsp,18h
0000000140001039 ret
```
Instructions we now know (17)

• NOP
• PUSH/POP
• CALL/RET
• MOV
• ADD/SUB
• IMUL
• MOVZX/MOVZX
• LEA
• JMP/Jcc (family)
• CMP/TEST