Aufgabenblatt 12

Ausgabe 24/01/2011, Abgabe bis 31/01/2011 12:00

Name(n):
Matrikelnummer(n):
Übungsgruppe:

Aufgabe 12.1 PC-relative Addressierung (10+10+10 Punkte)

Die x86-Architektur erlaubt bei Sprungbefehlen (`call`, `jmp`, `je` und Varianten) sowohl die Angabe absoluter Zieladressen, als auch die Berechnung relativ zum aktuellen Wert des Program-Counters `eip`. Diese verschiedenen Möglichkeiten werden als separate Befehle mit unterschiedlichen Opcodes codiert.

Bei PC-relativen Sprüngen wird der Offset vorzeichenbehaftet mit 1, 2, oder 4 Bytes kodiert, und wird relativ zur Startadresse des nachfolgenden (!) Befehls angegeben. (Dieses Verhalten ist darauf zurückzuführen, dass ältere x86-Prozessoren den Wert des Registers `eip` als ersten Schritt der Befehlsausführung inkrementierten.)

Überlegen Sie sich in den folgenden Beispielen die relevanten Adressen und ersetzen Sie die Platzhalter `xxxxxxx` jeweils durch die passenden Werte:

a) Was ist die Zieladresse des Befehls `jbe` (jump if below or equal) im folgenden Beispiel (Opcode 0x76, Offset 0xda im Zweierkomplement):

```
804001c:   76 da            jbe  xxxxxxx
804001e:   eb 24            jmp  8040044
```

b) Ergänzen Sie die Adressen:

```
xxxxxxx:   eb 54            jmp  8050d42
xxxxxxx:   c7 45 f8 10 00   mov  $0x10,0xfffffff8(%ebp)
```

c) Ergänzen Sie die Sprungadresse (4-Byte Offset, Byte-Order beachten):

```
8040000:   e9 cb 00 00 00   jmp  xxxxxxx
8040005:   90               nop
```

Aufgabe 12.2 x86-Assembler: Fibonacci-Folge (10+5+15+10 Punkte)

Die Fibonacci-Folge als unendliche Zahlenfolge, ist ein gutes und übersichtiches Beispiel zur Demonstration einfacher numerischer Operationen. Wir betrachten folgende Definition der Fibonacci-Folge:

$$f_n = \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n \in \{1, 2\} \\ f_{n-1} + f_{n-2}, & \text{falls } n > 2 \end{cases}$$

a) Schreiben Sie x86-Assemblercode für eine **iterative** Variante der Funktion zur Berechnung einer Fibonacci-Zahl n. Die Funktion erwartet einen Integer (32-bit) als Parameter und liefert das Resultat ebenfalls als Integerwert zurück, also `int fib( int n )`. Die iterative Funktion hat folgende Darstellung als C-Code:

```
int fib(int n) {
  int i = 3;
  int f_n = 0;
  int f_n1 = 1;
  int f_n2 = 1;

  if (n == 0)
    return 0;
  else if (n <= 2)
    return 1;
  else {
    for (i; i <= n; i++) {
      f_n = f_n1 + f_n2;
      f_n2 = f_n1;
      f_n1 = f_n;
    }
    return f_n;
  }
}
```

b) Welches ist der größte Wert von `n`, für den die Funktion noch ein korrektes Ergebnis (ohne Überlauf) liefert?

c) Schreiben Sie x86-Assemblercode für eine **rekursive** Variante der obigen Funktion zur Berechnung einer Fibonacci-Zahl n. Die rekursive Funktion hat folgende Darstellung als C-Code:

```
int fib(int n) {
  int tmp1, tmp2;

  if (n == 0)
    return 0;
  else if (n <= 2)
    return 1;
  else {
    tmp1 = fib(n - 1);
    tmp2 = fib(n - 2);
    return tmp1 + tmp2;
  }
}
```

d) Die rekursive Formulierung eines Problems ist in vielen Fällen elegant und gleichzeitig effizient. Leider ist dies für die obige Funktion nicht der Fall; diese ist zwar in der Tat elegant formuliert aber die Auswertung erfordert exponentiell mit `n` wachsenden Aufwand. Warum?

Skizzieren Sie den Stack im Zustand maximaler Verschachtelungstiefe nach dem Aufruf von `fib(6)`.

Aufgabe 12.3 x86-Assembler entschlüsseln (10+20 Punkte)

Wir betrachten die folgende Funktion `int mystery(int value)`, die einen 32-bit Integer-Parameter entgegennimmt und einen 32-bit Integerwert im Register `eax` zurückliefert:

```
080483e4 <mystery>:
 80483e4:  55                   push   %ebp
 80483e5:  89 e5                mov    %esp,%ebp
 80483e7:  8b 55 08             mov    0x8(%ebp),%edx
 80483ea:  89 d0                mov    %edx,%eax
 80483ec:  25 55 55 55 55       and    $0x55555555,%eax
 80483f1:  01 c0                add    %eax,%eax
 80483f3:  81 e2 aa aa aa aa    and    $0xaaaaaaaa,%edx
 80483f9:  d1 ea                shr    %edx
 80483fb:  09 d0                or     %edx,%eax
 80483fd:  89 c2                mov    %eax,%edx
 80483ff:  81 e2 33 33 33 33    and    $0x33333333,%edx
 8048405:  c1 e2 02             shl    $0x2,%edx
 8048408:  25 cc cc cc cc       and    $0xcccccccc,%eax
 804840d:  c1 e8 02             shr    $0x2,%eax
 8048410:  09 c2                or     %eax,%edx
 8048412:  89 d1                mov    %edx,%ecx
 8048414:  81 e1 0f 0f 0f 0f    and    $0xf0f0f0f,%ecx
 804841a:  c1 e1 04             shl    $0x4,%ecx
 804841d:  81 e2 f0 f0 f0 f0    and    $0xf0f0f0f0,%edx
 8048423:  c1 ea 04             shr    $0x4,%edx
 8048426:  09 d1                or     %edx,%ecx
 8048428:  89 c8                mov    %ecx,%eax
 804842a:  25 ff 00 ff 00       and    $0xff00ff,%eax
 804842f:  c1 e0 08             shl    $0x8,%eax
 8048432:  81 e1 00 ff 00 ff    and    $0xff00ff00,%ecx
 8048438:  c1 e9 08             shr    $0x8,%ecx
 804843b:  09 c8                or     %ecx,%eax
 804843d:  c1 c0 10             rol    $0x10,%eax
 8048440:  5d                   pop    %ebp
 8048441:  c3                   ret
```

a) Kommentieren Sie die einzelnen Assemblerbefehle.

b) Überlegen Sie sich, was die Funktion „berechnet“ und beschreiben Sie dann in eigenen Worten, wozu diese Funktion gut sein könnte. Welche Funktionswerte ergeben sich zum Beispiel für die Parameterwerte 0, 1, `0xcafebabe`?