64-041 Übung Rechnerstrukturen und Betriebssysteme WS26/27



Aufgabenblatt 4 Ausgabe: 27.10., Abgabe bis 04.11.2026 24:00

Gruppe	
Name(n)	Matrikelnummer(n)

Aufgabe 4.1 (Punkte 5+5+5+5)

Darstellung negativer Zahlen: Geben Sie für jedes der folgenden 8-bit Bitmuster die fünf dezimalen Werte an, die sich bei deren Interpretation als

1. (positive) ganze Zahl im Dualsystem,
2. Betrag und Vorzeichen,
3. Exzess-127 Codierung,
4. Einerkomplement,
5. Zweierkomplement

ergeben. Für die „Betrag und Vorzeichen“ Codierung gilt, dass (wie üblich) das MSB das Vorzeichen darstellt.

(a) 0000 0101

(b) 0111 1110

(c) 1001 0001

(d) 1111 1000

Aufgabe 4.2 (Punkte 5+5+5+5)

Subtraktion mit Komplementen: Führen Sie die folgenden Subtraktionen im Dezimalsystem unter Nutzung des 10-Komplements aus. Rechnen Sie dabei mit vier Dezimalstellen und notieren Sie die anfallenden Zwischenschritte (z.B. die Komplemente):

(a) $1487 - 839$

(b) $543 - 687$

Wiederholen Sie die beiden Berechnungen im Dualsystem im 2-Komplement mit 12 Binärstellen. Zuerst müssen Sie natürlich die Zahlen ins Dualsystem umwandeln. Bitte alle Zwischenschritte notieren:

(c) $1487 - 839$ (2-Komplement mit 12 Dualstellen)

(d) $543 - 687$ (2-Komplement mit 12 Dualstellen)

Aufgabe 4.3 (Punkte 10+10)

Subtraktion mit negativen Zahlen: In der vorherigen Aufgabe waren die beiden Operanden der Subtraktion beide positiv. Jetzt wollen wir ausprobieren, ob die Subtraktion im Zweierkomplement auch funktioniert, wenn einer oder sogar beide Eingangswerte selbst negativ sind.

Konvertieren Sie also zunächst die beiden Werte der folgenden Subtraktionen in die 2-Komplement Darstellung mit 12 Binärstellen. Führen Sie dann die Subtraktion (durch Addition des Komplements) aus, und vergleichen Sie den Wert mit der Berechnung im Dezimalsystem:

(a) $(-100) - 13$ (2-Komplement mit 12 Dualstellen)

(b) $(-128) - (-253)$ (2-Komplement mit 12 Dualstellen)

Aufgabe 4.4 (Punkte 5+5+10+20)

Akkumulator-Maschine. Die am Anfang der Vorlesung vorgestellte „Primitive Maschine“ verkörpert einen Digitalrechner der ersten Generation (ca. 1950). Es gibt nur ein Resultatregister (den „Akkumulator“) für die arithmetischen und logischen Operationen, und alle Datenoperationen und Sprungbefehle arbeiten mit absoluten Addressen. Es ist zwar unbequem, aber auch sehr lehrreich, ein paar Beispielprogramme für einen derartigen Rechner zu erstellen.

Die PRIMA verfügt über einen Hauptspeicher von 256 Bytes (Adressen 0..255) und insgesamt vier 8-bit Register: den Programmzähler (program counter, PC), das Befehlsregister (BR), ein Adressregister (AR), und den Akkumulator (ACCU). Der Befehlszyklus umfasst drei Takte: Befehl holen , Adresse holen, und Ausführung:

```
PRIMA Befehlszyklus:
   1. Holen:        BR := MEM[ PC ], PC := PC + 1
   2. Adresse:      AR := MEM[ PC ], PC := PC + 1
   3. Ausführung:   siehe Befehlssatz
```

Die Befehlsausführung startet nach einem Reset bei Speicheradresse 0. Als Beispiel für minimale Input/Output-Fähigkeiten gibt es zusätzlich einen von außen bedienten Schalter (switch, SW), der mit einem bedingten Sprungbefehl abgefragt werden kann.

Sie können die folgenden Aufgaben gerne mit Papier und Bleistift lösen, aber natürlich ist die Verwendung einer virtuellen Maschine zu empfehlen. Wie stellen ein Python-Skript zur Verfügung, dass die PRIMA simuliert: http://tams.informatik.uni-hamburg.de/lectures/2026ws/vorlesung/rsb/uebung/prima.py.

Die Simulation ist text-basiert, kann also auch remote im Terminal ausgeführt werden, etwa (Remote/SSH-Zugang) auf `rzssh1.informatik.uni-hamburg.de`:

```
ssh <username> rzssh1.informatik.uni-hamburg.de
wget tams.informatik.uni-hamburg.de/lectures/2026ws/vorlesung/rsb/doc/prima.py
python prima.py
```

Im Python-Interpreter kann dann interaktiv herumgespielt werden, aber natürlich können Sie auch Ihre eigenen Skripte schreiben und starten. Der Python-Interpreter läuft nach Fehleingaben normalerweise weiter, also können beliebig mit der Simulation herumspielen. Zum Beispiel:

```
prima = PRIMA()
prima.memset( 0, [42,11,2,3,4,5,6,7,8,9] ) # MEM[0] = 42, MEM[1] = 11, ...
prima.print_memory( 0, 127, 1 )            # MEM[0..127] list of hex-values
prima.print_registers()
prima.load_memory( "/tmp/demo.txt" )       # addr: hex hex hex hex ... \n
prima.clk()                                # clock pulse (phase 1 or 2 or 3)
prima.instruction()                        # one instruction (phase 1+2+3)
prima.set_SW( 0 )                          # set input switch to 0 (or 1)
for i in range( 1000 ):
    prima.instruction( print_registers=True )
    if prima.is_halted(): break            # machine stopped
print.print_memory( 0, 256, 2 )            # MEM[0..255] addr + hex-values
prima.save_memory( "/tmp/euklid.txt", start=0, end=256,
                   comment="Euklids Algorithmus von A.Author" )
```

Die folgende Tabelle fasst noch einmal den Befehlssatz des Rechners zusammen, wobei die Opcodes („operation code“) der einzelnen Befehle im Hexadezimalformat angegeben sind:

```
PRIMA Befehlssatz:
   Opcode     Mnemonic    Bedeutung

   0x00       nop         no-operation
   0x01       clear       accu = 0
   0x02       load        accu = MEM[ AR ]
   0x03       store       MEM[ AR ] = accu

   0x10       incr        accu = (accu + 1) & 0xff
   0x11       decr        accu = (accu - 1) & 0xff
   0x12       add         accu = (accu + MEM[AR]) & 0xff
   0x13       sub         accu = (accu - MEM[AR]) & 0xff

   0x20       neg         accu = ~accu & 0xff     (bitwise not)
   0x21       and         accu = accu & MEM[AR]   (bitwise and)
   0x22       or          accu = accu | MEM[AR]   (bitwise or)
   0x23       xor         accu = accu ^ MEM[AR]   (bitwise xor)

   0x40       jump        PC = AR           (unconditional jump)
   0x41       bz          if (accu == 0) PC = AR    (branch if zero)
   0x42       bneg        if (accu & 0x80) PC = AR  (branch if negative)
   0x45       bsw         if (SW) PC = AR           (branch if switch)

   0xff       halt        stop machine execution
```

Hier jetzt die Aufgaben:

(a) Offenbar sind nicht alle der (bei 8-bit) möglichen 256 Bitkombinationen auch tatsächlich mit Befehlen belegt. Es ist also möglich, dass der Rechner bei der Programmausführung auf eine „Illegal Instruction“ stößt, also einen nicht kodierten Befehl. Überlegen Sie sich, was die Maschine in so einem Fall am besten tun sollte?

(b) Es gibt für die PRIMA zwar den „Load“-Befehl (`0x02`), der im zweiten Schritt des Befehlszyklus das Adress-Register lädt, und dann im dritten Schritt den Inhalt der angegeben

Speicheradresse in den Akkumulator lädt.

Es gibt aber keinen separaten „Load Immediate“ Befehl, um einen direkt im Befehl kodierten Zahlenwert in den Akkumulator zu laden. Überlegen Sie sich, einen Befehl (oder eine Befehlsfolge), startend ab Speicheradresse `0x30`, die den Wert `0x67` in den Akkumulator lädt.

(c) Schreiben Sie ein Programm, dass den Wert in der Speicherstelle `0xff` zunächst einmal auf den Wert `0x03` initialisiert und anschließend in einer Endlosschleife immer weiter hochzählt.

Bitte geben Sie für diese und die folgenden Aufgaben das Programm jeweils als „ausführbare“ Textdatei (erzeugt mittels `prima.save_memory()`) und als kommentiertes Listing ab, wo sie die verwendeten Speicheradressen und jeden einzelnen Maschinenbefehl beschreiben.

(d) Euklid's Algorithmus zur Berechnung des größten gemeinsamen Teilers („GGT“) zweier positiver Zahlen gilt als der allererste „nicht-triviale“ Algorithmus der Geschichte (siehe https://de.wikipedia.org/wiki/Euklidischer_Algorithmus).

```
EUCLID_OLD(a,b)
1  wenn a = 0 dann
2    Ergebnis = b
3  sonst
4    solange b ungleich 0
5      wenn a > b dann
6        a = (a - b)
7      sonst
8        b = (b - a)
9  Ergebnis = a
```

Versuchen Sie, diesen Algorithmus für die PRIMA umzusetzen. Ihr Programm sollte ab Adresse `0x10` im Speicher abgelegt werden, und wir vereinbaren die festen Adressen `0xf0` und `0xf1` für die beiden Argumente a und b des Algorithmus. Das Endergebnis ggt soll an Adresse `0xf2` abgelegt werden, und schließlich soll das Programm stoppen.

Nach einem Reset der Maschine startet die Programmausführung an Adresse `0x00`, also wird Ihr Programm zusätzlich auch einen Sprung an die Adresse `0x10` benötigen.

Um den Befehl `bneg` (`0x42`) ohne weitere Komplikationen benutzen zu können, beschränken wir uns beim Zahlenbereich für die beiden Argumente a und b auf $0 \leq a, b, \leq 127$.

Probieren Sie Ihr Programm mit mehreren Wertekombinationen aus, z.B. $a = 16, b = 80$, $a = 100, b = 125$, $a = 84, b = 126$ usw. Liefert das Programm das richtige Ergebnis, wenn entweder a oder b oder sogar beide Argumente den Wert Null aufweisen?