Aufgabe 1

Wir benutzen die verschiedenen Adressierungsarten `Displacement`, `Indirect` und `Direct`:

Variante 1: `Displacement`

```
LW    R3, 180(R1)     // Lade 32-Bit Integerzahl aus Speicher
                         mit Adresse 492-495
```

Variante 2: `Indirect`

```
LW    R3, 0(R2)       // Lade 32-Bit Integerzahl aus Speicher
                         mit Adresse 492-495
```

Variante 3: `Direct`

```
LW    R3, 492(R0)     // Lade 32-Bit Integerzahl aus Speicher
                         mit Adresse 492-495
```

Aufgabe 2

Das folgende Programm vertauscht die Werte zweier ganzzahliger 32-Bit-Zahlen in Register `R1` und `R2`:

```
ADD R3,R0,R1      // Schreibe R1 nach R3
ADD R1,R0,R2      // Schreibe R2 nach R1
ADD R2,R0,R3      // Schreibe R3 nach R2
```

Zuerst wird der Wert in Register `R1` in Register `R3` zwischengespeichert. Dies wird realisiert, indem durch den Befehl `ADD R3, R0, R1` der Wert aus Register `R1` und der Wert aus Register `R0` (dieses ist stets 0) addiert werden und das Ergebnis nach `R3` geschrieben wird. Nun kann der Wert aus Register `R2` auf dieselbe Weise in das Register `R1` geschrieben werden. Schließlich wird der alte, in `R3` zwischengespeicherte Wert von `R1` durch den Befehl `ADD R2, R0, R3` nach `R2` geschrieben.
Alternative Version mit nur zwei Registern (auf eine ausführliche Beschreibung wird hier verzichtet):

```
ADD R1,R1,R2    // Speichere die Summe von R1 und R2 in R1
SUB R2,R1,R2    // Ziehe R2 von R1 ab, und speichere das Ergebnis in R2
                // R2 enthält nun das alte R1
                // R1 die Summe von R1 und R2
SUB R1,R1,R2    // Ziehe R2 von R1 ab, R1 enthält nun das alte R2
```

Aufgabe 3

Wir kopieren die zu bearbeitende Zahl nach R3 und ziehen solange 15 ab, bis die Zahl kleiner als 15 ist:

```
        ADD     R3,R2,R0    //Kopiere R2 nach R3
loop:   SUBI    R3,R3,#15   //Ziehe 15 von R3 ab
        SLTI    R4,R3,#15   //Ist R3 schon kleiner als 15?
        BEQZ    R4,loop     //Wenn nicht, springe zu loop
```

Problem: Der Algorithmus funktioniert nur für Zahlen größer gleich 15, daher addieren wir erst solange 15, bis diese Bedingung erfüllt ist:

```
        ADD     R3,R2,R0    //Kopiere R2 nach R3
loop1:  ADDI    R3,R3,#15   //Addiere 15 zu R3
        SLTI    R4,R3,#15   //Ist R3 immer noch kleiner als 15?
        BNEZ    R4,loop1    //Dann springe wieder zu loop1
                            //R3 ist nun größer als 15
loop2:  SUBI    R3,R3,#15   //Ziehe 15 von R3 ab
        SLTI    R4,R3,#15   //Ist R3 schon kleiner als 15?
        BEQZ    R4,loop2    //Wenn nicht, springe zu loop2
```

Problem: Bei großen Zahlen kann dieser Algorithmus über 100.000.000 Zyklen durchlaufen. Dauert uns das zu lange, können wir ihn beschleunigen, indem wir ihn so modifizieren, dass er zunächst Vielfache von 15 subtrahiert, und den Subtrahenden dann jeweils verkleinert, wenn er größer wäre als der Minuend. Dafür benutzen wir statt der #15 nun R1, das wir mit $15 \cdot 2^{24}$ initialisieren und jeweils durch 16 teilen.
Die blauen Zeilen in folgendem Programm könnten also weggelassen werden, beschleunigen das Programm aber von $\mathcal{O}(n)$ auf $\mathcal{O}(\log n)$:

```
        ADD     R3,R2,R0    //Kopiere R2 nach R3
        ADDI    R1,R0,#15   //Speichere 15 in R1 (Subtrahend)
        SLLI    R1,R1,#24   //Erhöhe den Subtrahenden auf ein Vielfaches von 15
loop1:  ADD     R3,R3,R1    //Addiere R1 zu R3
        SLT     R4,R3,R1    //Ist R3 immer noch kleiner als R1?
        BNEZ    R4,loop1    //Dann springe wieder zu loop1
                            //R3 ist nun größer als R1
loop2:  SUB     R3,R3,R1    //Ziehe R1 von R3 ab
        SLT     R4,R3,R1    //Ist R3 schon kleiner als R1?
        BEQZ    R4,loop2    //Wenn nicht, springe zu loop2
        SRLI    R1,R1,#4    //Verkleinere den Subtrahenden
        BNEZ    R1,loop1    //Solange bis R1==0
        HALT
```

Die Registerbelegung ist dabei wie folgt gegeben:

```
R2: Enthält das Argument; wird nach R3 kopiert, damit es nicht verändert wird
R3: Kopie des Arguments, wird solange verändert, bis es das Ergebnis enthält
R4: Hilfsregister für Vergleiche
R1: Enthält den Wert, der von R3 abgezogen/zu R3 addiert werden soll
```

Aufgabe 4

Zunächst kommentieren wir gegebenes Programm zeilenweise:

```
      LW    R1, 1024(R0)     // Lade 32-Bit Integerzahl aus Speicher mit
                             //   Adresse 1024-1027
      ADDI  R2, R0, #1       // Initialisiere Register R2 mit der Konstanten 1
      ADD   R4, R0, R0       // Initialisiere Register R4 mit der Konstanten 0
Loop: AND   R5, R2, R1       // Prüfe, ob das letzte Bit der Zahl im Register R1
                             //   eine 1 ist. Setze in dem Fall R5=1, sonst R5=0.
      XOR   R4, R4, R5       // Prüfe, ob jeweils die letzte Ziffer von R4 und
                             //   R5 ungleich sind. Falls R5=1 ist, kippe also das
                             //   Bit in R4, sonst bleibt R4 gleich.
      SRL   R1, R1, R2       // Schiebe Registerinhalt von R1 logisch um 1 Bit
                             //   nach rechts
      BNEZ  R1, Loop         // Solange R1 ungleich 0, springe zu Loop
      SW    1028(R0), R4     // Schreibe den Wert aus Register R4 in den
                             //   Speicher mit Adresse 1028-1031
      HALT                   // Programmende
```

Das Programm gibt an, ob eine gerade oder ungerade Anzahl von Einsen in der 32-Bit Integerzahl im Speicher mit der Adresse 1024-1027 vorkommt.
Enthält die Zahl eine gerade Anzahl von Einsen, wird eine 0 an der Adresse 1028-1031 gespeichert, anderenfalls eine 1.
Das Programm lädt eine 32-Bit Integerzahl aus dem Speicher in das Register R1 und initialisiert die Register R2 und R4 mit 1 respektive 0.
Der Befehl `AND R5, R2, R1` prüft für jedes der 32 Bits, ob die jeweiligen Bits von R2 und R1 beide 1 sind. Da der Wert vom Register R2 mit 1 initialisiert ist, kann eine 1 nur beim letzten Bit von R5 auftreten; und zwar genau dann, wenn das letzte Bit (LSB) von R1 eine 1 ist. R5 enthält nun das LSB von R1, ist also entweder 0 oder 1. Der Befehl `XOR R4, R4, R5` setzt das LSB von R4, wenn die jeweiligen LSB der Zahlen aus den Registern R4 und R5 verschieden sind. Dies wird nun für jedes Bit von R1 durchgeführt, indem R1 um ein Bit nach rechts geshiftet wird und dann zu `"Loop:"` gesprungen wird. Falls das Register R1 nur noch aus Nullen besteht, führt das Programm nach der Schleife (nächste Zeile) fort.
Zum Ende des Programms wird mit dem Befehl `SW 1028(R0), R4` der Wert aus dem Register R4 in den Speicher mit der Adresse 1028-1031 geschrieben. Dabei kann leidiglich das letzte Bit ungleich 0 sein.
Die Registerbelegung ist dabei wie folgt gegeben:

```
R1:  Enthält zu Beginn das geladene Wort, es wird immer das LSB untersucht
     und das Wort um ein Bit weiter nach rechts geshiftet
R2:  Hilfsregister, enthält den konstanten Wert 1
R4:  Hilfsregister, wird getoggelt, falls aktuelles Bit eine 1 ist,
     enthält am Ende eine 0, falls Anzahl der 1en gerade, sonst eine 1
R5:  Hilfsregister für Vergleiche
```