University of Applied Sciences Ulm
Institute of Computer Science – Lab Exercises

CTS-ICTS, Prof. F. Steiper

Lab Exercises to Chapter 5 of the Lecture „Introduction to Computer Science“

Content

Preparation: Starting the “Microprocessor Simulator“

Switch on your computer and choose the “Windows Boot Manager (on /dev/sda2)”. Log on to the Windows 10 system using your standard university account. The executable file **sms32v50.exe** of the Microprocessor Simulator is located in the folder **D:\SteiperGr1\smz32v50**. Double-click the executable file and start the simulator. Use this simulator to solve the following tasks. Have a look at chapter 5.3 of the lectures as well.

（If you want to install the simulator on your own Windows machine:
The installation file **sms32v50.exe** of the simulator can be downloaded from http://www.softwareforeducation.com . Look for the entry “Version 5.0 - smz32v50.exe - Self extracting ZIP file…”. Copy the self-extracting file “smz32v50.exe” into a **separate** folder and extract the files. Start the program “sms32v50.exe” in this folder.)

Part 1: High Level Programming Languages, Assembly Languages and Machine Languages

High Level Languages vs. Assembly Languages

The assembly language can be seen as an intermediate step between a high level language like C++ and a machine language. In an assembly language there is exactly one assembler instruction for each machine instruction. The assembly language uses exactly those instructions a processor understands. For this reason the assembler programmer needs to divide each task – no matter how simple – into very little subtasks, which are able to be expressed in machine commands. Error messages similar to those in high level languages like C++ or Java hardly exist. Incorrectly implemented assembler programs usually result in program crashes. For control structures like “if, for, while, etc.”, which are common in high level languages, there are nearly no direct equivalents in any machine language. A C++ compiler implements the abstract C++ instructions using the simpler machine instructions of

a specific processor. Especially in complex problems high level languages allow solutions that are much better structured and readable, less susceptible for errors, shorter and independent of the used processor.

Tasks:

1. Look at the following Java program:

```
public class ifAnweisung {
        public static void main(String[] args) {
                int a=3;
                if (a<5) {
                        a++;
                }
                System.out.println(a);
        }
}
```

   - Firstly create an assembler program which reproduces the function of the Java program above!
   - To output the program´s result in a terminal use the **Visual Display Unit (VDU)** of the simulator in your assembler program. The following page of the simulator´s online manual contains related information:
     "**Help > Contents > Reference > Peripheral Devices**"
   - Test your program using an alternative initial value of the variable a: **int a = 3;**

2. Extend the assembler program you created in task 1 in a way that the following "if-else" control structure is reconstructed:

```
public class ifAnweisung {
        public static void main(String[] args) {
                int a=5;
                if (a==3) {
                        a++;
                }
                else {
                        a += 2;
                }
                System.out.println(a);
        }
}
```

3. Now reconstruct the following “for”-loop in an assembler program!

```
public  class  forAnweisung {
        public static void main(String[] args) {
                int a;
                int b=0;
                for ( a=3 ; a<6 ; a++) {
                        b += 3;
                }
                System.out.println(b);
        }
}
```

4. As a final example, reconstruct the following Java program!

```
public  class  Schleife {
        public static void main(String[] args) {
                int a;
                int b=0;
                for ( a=0 ; a<6 ; a++) {
                        if (a==3) {
                                b += 3;
                        }
                }
                System.out.println(a);
                System.out.println(b);
        }
}
```

Part 2: Implementation of Procedures in an Assembler Language

Procedures

Procedures are a possible variant of subroutine calls. A procedure consists of a set of assembler instructions, which are stored one after the other, beginning at a certain physical address in the memory. Using our Microprocessor Simulator the start address of a procedure is determined by the instruction “**ORG** <Start-Address>”. That is, the “ORG” instruction is always the first assembler instruction of a procedure. Afterwards, there are the assembler instructions which represent the function of the procedure. The very last instruction of each procedure is “**RET**”.

The procedure is called out of the main program using the “**CALL** <Start-Address>” instruction. This CALL instruction ensures that the value of the program counter is saved on the stack as the return address before jumping into the procedure. After that the procedure is executed. The final “RET” command reconstructs the saved return address in the program counter, so that the main program can be continued at the right place. The associated tutorial can be found at “**Help > Contents F1 > Tutorials > 06 Procedures”**.

We must also keep in mind that the general purpose registers AL-DL and the Status Register can be modified in the main program as well as in the procedure. Therefore, it makes sense to store the current values of these registers in the stack first with the commands "PUSH" and "PUSHF" within the procedure. At the end of the procedure, they must be restored with the "**POP**" and "**POPF**" commands **in reverse order**! This way the main program can continue exactly in the same state that existed before calling of the procedure.

Parameter Handover

With the call of the procedure usually parameters need to be handed over to the procedure. Parameters can be handed over in three different ways:

1. You can use the general-purpose registers AL-DL. In this case the number of possible parameters is limited to 4.
2. You can use storage cells inside the RAM at certain physical memory addresses.
3. You can use the stack memory and the "PUSH" and "POP" instructions.

Tasks

1. The 3 methods of parameter handling are described in the example program **09PARAM.ASM** of the simulator. The associated tutorial can be found at "**Help > Contents F1 > Tutorials > 09 Parameters"**. Work through the tutorial. Sketch for each method which steps need to be carried out in which order to implement the parameter handover.

2. Write an assembler program which implements the modulo function:
   r = modulo(a,b). This corresponds to the Java instruction "r = a % b;".
   - Output the result of the calculation in a terminal using the Visual Display Unit (VDU).
   - Test the assembler program with different integer values of a and b where $0_{16} \leq a \leq 7F_{16}$ and $02_{16} \leq b \leq 09_{16}$.

3. The following Java method implements the Euclidian Algorithm concerning the determination of the greatest common divisor (GCD) of two integers:

```
public  int  ggt ( int i1, int i2 ) {
            int r;
            do {
                        r  = i1 % i2;
                        i1 = i2;
                        i2 = r;
            } while ( i2 != 0 );
            return i1;
}
```

Write an assembler program that calculates the largest common divisor of two integers a and b according to the above program:

- Modify the above “modulo” program so that it can be used as a procedure!
- Use the stack memory to handover parameters!
- Display the largest common divisor on the visual display unit.
- Implement the following extensions:

    a) Use the simple method for interpreting keyboard input, which is described in “**Help > Contents F1 > Tutorials > 05 Keyboard Input”**, so that you don´t have to define them statically inside the program. Please note that the variables can also consist of two digits. Because of this you must read a variable number of characters until the “return” button occurs.
    b) If the range of b is enhanced to $02_{16} \leq b \leq 0F_{16}$, you must keep in mind that the values are not limited to the digits 2-9, there can also be some like “A”. Change your program according to that.