Practical Session 1: Assembly

1 Getting started

Start by downloading and running Jasmin-1.5.8.jar. Some documentation on the interface, as well as the language accepted by Jasmin (a relatively large subset of the x86 instruction set) can be found online at: `http://wwwi10.lrr.in.tum.de/~jasmin/documentation.html`.

Jasmin is an assembly interpreter. You can type assembly instructions in the main window, execute your code (either step-by-step or the whole code at once) and observe the behaviour in the memory and registers. Don't forget to reset the virtual memory of Jasmin using the appropriate button after each run.

2 Using data

This section will introduce how to manipulate, store and read data in assembly.

2.1 Registers and Memory

In assembly, you have access to the CPU registers as well as the memory. The **MOV** instruction can be used to move data around.

Start by storing some numbers in the different registers. For instance:

1. Store the integer 4 in the 32-bit **EAX** register
2. Store the integer 10 in the 32-bit **EBX** register
3. Store the integer 5 in the 16-bit **CX** register
4. Store the integer 2 in the 8-bit **DL** register

Observe how each 16-bit register (e.g. **AX**) can be extended to a 32-bit register (**EAX**) or divided in two 8-bit registers for the high-order bits (**AH**) and low-order bits (**AL**).

You can use 32-bit registers as memory address pointers using brackets: [**EAX**].

Try to store the integer 25 in the memory address contained in **EAX** (i.e. `0x4`).

2.2 The stack

The x86 instruction set provides facilities to use the stack. In Jasmin the stack is located at the end of the memory: the stack pointer initially contains the address `0x1000`, and this address is decremented by 2 (resp. 4) if you push a 16-bit (resp. 32-bit) operand.

Try to **PUSH** some data on the stack and **POP** it into the **CX** register.

3 Performing computations

Write instructions to perform the following:

1. Add 4 to **EBX**.
2. Subtract the content of **EBX** from the integer located at the memory address stored in **EAX** (e.g. 0x4)
3. Multiply **EBX** by 8.
4. Divide **CX** by 7. The quotient must be stored in **ECX** and the remainder in **EDX**. **EAX** must be unchanged at the end of the operation (*hints*: use the stack to restore **EAX** and use an 8 bit register for the divider).

4 Control Flow

A simple `if` condition in x86 works in two steps :

1. First, compare two items between them.
2. Second, jump to a location in the code based on the result of the comparison.

Look up the **CMP** instruction as well as the conditional jump instructions (**JE**, **JNE**, **JZ**, ...) in the documentation of Jasmin or any x86 documentation of your choice.

The **CMP** instruction is basically a glorified **SUB**: it subtracts one operands to the other, then sets the CPU flags depending on the result. For instance, if two operands `arg1` and `arg2` are equal, their difference is $\texttt{arg1} - \texttt{arg2} = 0$ and the zero flag (`ZF`) is set to 1. If `arg1` is smaller, the result will be negative and the sign flag (`SF`) will be set, etc.

The conditional jump instruction then check the values of the flags and jump to an address if the condition they check for is met. For instance, the **JE** instruction, which checks that the operands were strictly equal (`arg1 == arg2`) will check that the zero flag is set. Check out for instance `http://unixwiz.net/techtips/x86-jumps.html` for the full list of which flags the jump instructions check.

Jump instructions take as operand an address which is the location in the code to jump to if necessary. Dealing with addresses directly can be terribly inconvenient for multiple reasons – how can you know the address of a particular location, for instance? To help with this problem, the assembly language provides *labels*. A label is a named reference to a user-defined location in the code, which the assembler will replace with the appropriate address when assembling the machine code.

Start by storing some example data we will use :

```
MOV [0xC], -35
```

Write a condition checking that the integer in address `0xC` is negative. If it is, multiply it by three. Otherwise, do nothing (*hint*: jump to a label to skip the multiplication if the number is positive.)

Writing a loop in assembly is not fundamentally different. To loop, you will simply jump to a label located before the conditional check.

```
MOV BX, 0x20
MOV CX, 0x70
```

Write a loop to generate a sequence of incrementing numbers starting from 0, in every address from the address in **EBX** up to the address in **ECX** (use 32-bit words). **EAX** and **EBX** must be unchanged after the operation.

5 AbsMean

We will now combine elements of the previous sections to write a small program that actually performs some computations.

Write a program that computes the mean of the absolute values of an array in memory. We assume the memory contains :

at `0x0`: n, a 32-bit integer: the length of the array.

from `0x4`: the array of n 32-bit signed integers.

The following instruction will write appropriate data in the memory of Jasmin, for an array of ten elements :

```
DD    10,   -12, 14, 65, -124, 57, 12, 13, -98, -41, 2
```

You can also directly edit the table in the right panel of the interface of Jasmin to manually change the sample values. For the example above, you should obtain a mean absolute value of 43 (we use integer division to simplify things)

6 Additional Exercise

This exercise is not mandatory.

We will introduce the basics of how assembly handles functions and how every function (i.e. in C for instance) can be translated in assembly.

Using functions in assembly is highly platform-specific. Many choices must be made such as were will the return value be stored and how to pass parameters. We will only gloss over how these problems can be solved.

Start by write a a piece of assembly code which implements the following function f:

$$f(n) = \begin{cases} n/2 & \text{if } n \text{ is even} \\ 3n+1 & \text{if } n \text{ is odd} \end{cases}$$

This piece of code should take its parameter n in **EAX** and store the result of $f(n)$ in the same register.
Now write a program to verify the Collatz conjecture, which states that, for any starting number n, if we repeatedly apply the f function defined above, we will always eventually reach 1.
Use a label to mark the beginning of the f function you defined above. Then use **CALL** and **RET** to call and return from your "function". You shoud write, in consecutive memory addresses starting from `0x0`, the full sequence of numbers generated until you get to 1.
For instance, if you start with $n = 13$, you should get the following sequence in memory: 13, 40, 20, 10, 5, 16, 8, 4, 2, 1.

We chose to pass the parameter in a register for simplicity here, but consider what would happen if we had more than 4 parameters? Actual systems usually

use the stack, but some caution is in order, since the top of the stack always contain the return address, pushed there by **CALL**. Consider also what would happen if we wished to use recursive functions. Obviously, using functions in assembly is feasible but can be very tricky.