TDT4205 Problem Set 4
Spring 2011

PART 1 - Theory due Wed Mar. 7th, 20:00.
PART 2 - Programming due Wed Mar. 14th, 20:00.

Part 1 needs to be passed in order for Part 2 to be evaluated!
ALL answers are to be submitted to *itslearning*

ALL OF THIS ASSIGNMENT IS TO BE DONE INDIVIDUALLY. Cheating ("koking"), including using partial solutions from other students, will cause a failing grade for the whole course. We will be checking for this using a plagerism detecting as well as looking for suspiciously similar submissions.

All submitted source code MUST be able to compile and run on asti. **This assignment counts towards your final grade.** Please read the assignment guidelines on itslearning before starting to work on the assignment. Requests for clarifications can be posted on the itslearning forum.

What to turn in
When turning in assignments, please turn in these files:

- (your_username)_answers.pdf : Answers to non-programming questions (Part 1)
- (your_username)_assembly.s : Solution to the assembly programming problem (Part 1)
- (your_username)_code.zip,tar.gz,tgz : All your code for this assignment, including makefiles and other necessary code (Part 2)

Part 1 - Theory and Assembly Programming

Task 1.1 - Stack frames (10%)

1. What is a stack frame and what is the purpose of this concept?
2. Illustrate the layout of the stack frame of the following VSL function and its block statement.

   ```
   FUNC saxpy(a,x,y)
   {
       VAR x1, x2
       x1 := a*x
       x2 := x1+y

       RETURN x2
   }
   ```

3. Describe the process of setting up and tearing down the stack frame after a function is called. E.g. what happens to the stack and registers?

Task 1.2 - x86 Assembly Programming (20%)

The program

```
#include <stdio.h>
#include <stdlib.h>

int foo(int N)
{
    int sum = 0;
    for (int i = 1; i < N; i++)
    {
        if (i % 3 == 0 || i % 5 == 0)
        sum += i;
    }
    return sum;
}

int main(int argc, char**argv)
{
    int N = atoi(argv[1]);
    int ans = foo(N);
    printf("Sum is %d.\n", ans);

    return 0;
}
```

finds the sum of all numbers below N that is dividible by either 3 or 5. Implement this function in x86 assembly. You should use the file `foo_skeleton.s` as a base, which reads and parses the command line argument for you, as well as setting up the stack frame for main(). For your convenience, a list of the neccessary instructions is found in the table in appendix A. As with the rest of the programming assignments, the code must be able to run on asti, and the assembly MUST be able to compile with AS assembler, with the supplied makefile.

For full score, you must properly set up a stack frame for foo(), as well as tearing it down after the function completes.

For error checking, you may compile the C program as listed above using `gcc --std=c99 foo.c -o program`, and run with ./program N where N is an integer larger than 0.

Task 1.3 - Symbol tables (10%)

The offset from the stack frame's base pointer for each local variable in a stack frame is stored in the symbol table. In earlier versions of VSL the first local variable has an offset of -4 from the base pointer, the next has an offset of -8 and so on. As an added complexity, in VSL 2012 it is possible to allocate arrays on the stack.

1. In the VSL program below, what would the stack offset be for each of the symbols `a`, `b` and `c`?

```
FUNC foo()
{
    VAR a
    VAR b[5]
    VAR c

    // irrelevant code
}
```

2. In part 2 of this assignment you will implement symbol tables for VSL. Each symbol there has three relevant fields for variables: Stack offset, lexical depth, and variable name. Why is it neccesary to know the lexical depth[1] of a symbol, in addition to the offset from the stack frame's base pointer?

[1]The lexical depth of a symbol is simply the number of nested scopes a variable is in. See appendix C for an example.

Appendix A - Relevant instructions

Instruction	Description	Notes
pushl A	Pushes the value A onto the stack.	
popl A	Pops the top value on the stack and puts it in A.	
movl A, B	Assigns the value of a to B.	
cmp A, B	Compares the value of A and B.	The results can be used with e.g. je, jge, jne, ...
divl A	Divides EDX:EAX by A. (EDX:EAX) / A is put in EAX, while (EDX:EAX) mod A is put in EDX.	After moving the divisor into EAX, use `cdq` to extend the sign into EDX.
addl A, B	Adds A to B.	
incl A	Adds 1 to A	
jmp L	Unconditional jump to label L	
je/jne L	Jump to label L if operands of last comparison was equal/not equal resp.	
jg/jge/jl/jle L	Jump to label L if last operand of the last comparison was greater, greater-or-equal, less and less-or-equal, resp.	
cdq	Extend the sign of EAX into EDX. Must be done before any 32 bit division.	

Appendix B - Registers

The names for these registers indicates its typical use; however, all of these registers may be used however you wish as they are considered general purpose. However, note that some registers, esp. the ESP and EBP registers are modified as an effect of certain instructions, e.g. push and pop will increment/decrement ESP, and LEAVE will do a copy of EBP into ESP and finally pop into EBP.

Register	Description
EAX	Accumulator.
EBX	Base index.
ECX	Counter.
EDX	Data.
ESI	Source index.
EDI	Destination index.
EBP	Base pointer. Points to the base of the current stack frame.
ESP	Stack poitner. Points to the top of the stack.

Appendix C - Lexical depth - an example

```
int main(int argc, char** argv)
{
    int i = 0;
    for (i = 0; i < 100; i++)
    {
        int a = 4;
        int b = i*2;
        if (a < b)
        {
            int c = b-a;
            printf("Something ODD is happening here!
        }
    }
    return 0;
}
```

Here, the function main() may be considered to have a lexical depth of 0. *i* has a lexical depth of 1, while *a* and *b* has a depth of 2. *c* has a depth of 3.