DAT 103
Datamaskiner og operativsystemer (Computers and Operating Systems)

Graded assignment – matrix multiplication.

Part II: Assembly translation (15%)

Deadline: 5.11.2021, 23:59

Your task for Assignment Part II is to translate `MatMulBinary.java` to assembly language. Specifically, you are to complete the skeleton `MatMulBinary.asm` so that it will replicate the behaviour of the Java program.

To pass the assignment (mandatory for admission to the exam), i.e., grade E, you need to have at least addressed Subtask 1 and Subtask 2, and your jumpTrace routine must work correctly. Your exact grade will be determined by a combination of how you succeed in Subtask 3, the quality and structure of your assembly code and comments, and on bonus points from the bonus task in Part I.

Subtask 1 Data input

In assembly, data in decimal form (like in the `A*.mat` files) is not only more computationally expensive, there also is no standard library that has this functionality built-in. Manually implemementing decimal input would require reading the individual digits, translating them to integers and adding them together each with the right power-of-10 factor. This is *not* required for this assignment.

Instead, you are to accept the data in the binary format that `toBinary.c` produces. Fortunately, this is already the format that the matrices will have in memory as x86-native arrays, so all that needs to be done is *copying* to the program's memory.

This is already implemented in the `readBinaryData` routine. Your task is to comment every line of this routine, explaining what it does and/or why.

Answer the following questions (in a comment after the `ret` statement):

- Would the routine work with matrices of *any* size? It does work with the A- and B matrices we are using, but what requirement makes this possible?
- What could be changed to actually make it work with any size? Discuss if your suggestion has any drawbacks.

Subtask 2 Pseudo-hash

The `jumpTrace` function reduces a matrix to a single integer, one that will change unpredictably if any of the matrix entries is varied.

The assembly template contains part of this function's implementation, demonstrating in particular how to set up a function and a loop. You need to complete the second half of the loop body. Make use of the provided macros, and comment your code extensively.

In order to check/debug this part, we recommend temporarily modifying both the assembly main-routine and the corresponding Java reference to directly show the jump-trace of one of the input matrices, without computing the matrix multiplication. Ensure that the assembly version then consistently gives the same output as the Java one (given different matrix inputs from the examples).

Subtask 3 Matrix multiplication

For the final part you are on your own: implement the nested loop that computes the product of matrices A and B and stores the result in C.

Obligatory submission Part II (due on 5.11.2021, 23:59)

When/Where to submit:

- You will have make your submission in Canvas on/before 5.11.2021, 23:59.

How to submit:

- You can work in a group of at most 3 people. Note that if you work in a group, the group should be formed and registered in Canvas **before the submission**. If you join a group after the submission, a new submission has to be made in order to receive the grade of the assignment.
- Pack the source code you wrote in **a single archive file** called `oblig2-`*studnr*`.tar`, where *studnr* is your student number. If you work in a group, use the group ID instead: `oblig2-group`*grpID*`.tar`.

  For example, `oblig1-4567.tar` if student 4567 submits alone, or `oblig1-group89` for group 89.

  (Please avoid including any spaces, uppercase or non-ASCII characters in the file name. The separator should be *a single hyphen/minus character*, as is good practice in any Unix project.)

  - This archive must contain the files:
    1. `readBinaryData.asm`,
    2. `jumpTrace.asm`, and
    3. `matmul.asm` if you did Subtask 3 also.
  - Each of the above should contain **only the routine it is concerned with**, for example the file `jumpTrace.asm` should
    * begin with

      `jumpTrace:`
    * and end with

      `funret3_1 eax`

    (plus possibly extra comments, but no header etc. code or a main routine).

  Double-check that your archive can be unpacked without errors, by moving it to an empty directory and running the command `tar xf` on the file. This should not give any error message, the exit code should be 0 (check with `echo $?`), and it should result in the necessary files being in the working directory now. Points will be deducted for those archives that cannot be unpacked successfully (i.e., exit code is 0).

x86 cheat sheet

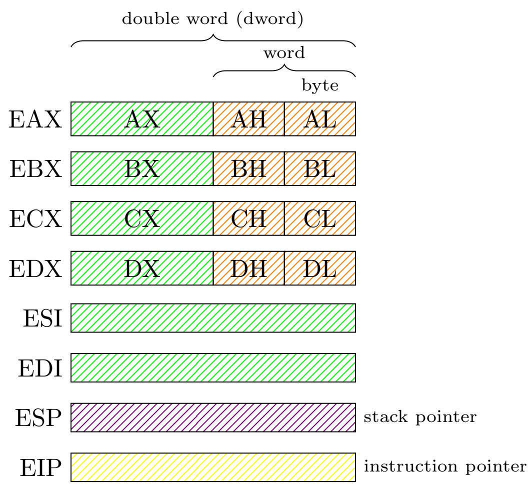

Figure 1: Layout of the registers. (The details are not important for the assignment.)

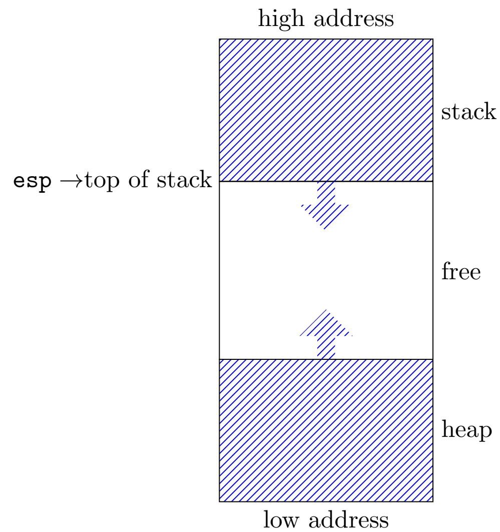

Figure 2: Memory usage. Note that pushing to the stack *lowers* the value of `esp`.

mov *dst*, *src*	$dst \leftarrow src$
mov `eax`, [`ebx`]	`eax` $\leftarrow$ dword at addr in `ebx`
mov [var], `ebx`	dword at addr var $\leftarrow$ dword in `ebx`
mov `eax`, [`esi` $-$ 4]	`eax` $\leftarrow$ dword at addr in `esi` $-$ 4
mov [`esi` + `eax`], cl	byte at addr in `esi` + `eax` $\leftarrow$ byte in cl
mov `edx`, [`esi` + 4 ∗ `ebx`]	`edx` $\leftarrow$ dword at addr `esi` + 4 ∗ `ebx`

add *dst*, *src*	$dst \leftarrow src + dst$
sub *dst*, *src*	$dst \leftarrow src + dst$
inc *dst*	$dst \leftarrow dst + 1$
dec *dst*	$dst \leftarrow dst - 1$
mul *dst*, *src*	$dst \leftarrow src * dst$
mul *dst*, *src*, *val*	$dst \leftarrow src * val$
div *src*	`eax` $\leftarrow \frac{\texttt{eax}}{src}$; `edx` $\leftarrow$ `eax` mod *src*; this requires that `ebx` = 0

push *src*	`esp` $\leftarrow$ `esp` $-$ 4; dword at esp $\leftarrow$ *src*
push `eax`	place dword in `eax` on top of stack
push [var]	place dword at addr in var on top of stack
push dword 0	place dword 0x0000 on top of stack

pop *dst*	*dst* $\leftarrow$ dword at `esp`; `esp` $\leftarrow$ `esp` + 4
pop *dst*	pop top of stack into *dst*
pop [`ebx`]	pop top of stack into dword at addr in `ebx`

cmp src_1, src_2	set flags depending on whether $src_1 ? src_2$ where ? is ==, < or >
jg *label*	jump to *label* if flag indicates $src_1 > src_2$
jl *label*	jump to *label* if flag indicates $src_1 < src_2$
je *label*	jump to *label* if flag indicates $src_1 == src_2$

call *label*	push next instruction address to stack; jump to *label*
ret	pop top of stack to instruction pointer
int *val*	call interrupt handler at *val*. In x86 Linux 0x80 interrupt handler is the kernel, and is used to make system calls to the kernel.