Maxeler Off-Site Programming Exercise

=====================================

This exercise consists of two tasks which both involve writing small computer

programs. The first task revolves around creating a simple virtual-machine

which processes files provided with this document. The second task will be

revealed when you use your VM implementation to execute the 'task1.bin' file

provided.

For these exercises there is no requirement that you use any specific

programming language/tools/operating systems etc. As such you may feel free to

use whatever tools you feel are most appropriate. However, you should submit all

relevant source code, scripts and data you rely on for both test and build.

You should aim for it to be possible for us to re-compile/create your solution

using: what you have sent us, off the shelf products, and/or programs

downloadable from the Internet.

Please be considerate of the fact a human will need to read any source code

you provide and as such you should try to keep this neat and commented where

appropriate. Furthermore, bear in mind that if you are invited for an

in-office interview you may be asked to discuss your solution and justify the

approaches you have taken.

We expect the time needed to completed both parts of this exercise to be

around 4-6 hours with slightly more time generally needed for the second task.

If you find that you are unable to complete both parts by the submission

deadline, please submit what you have done so far with an explanation of what

does and does not work.

Submitting Your Answers

=======================

To submit your answers you need to include both the results from running your

programs and their source-code in a single zip-file. This file should have the

following format (/ indicates directory):

task1/

result.txt - Output from running your VM on the task1.bin file.

source-code/

overview.txt - Brief overview of the files in this

directory.

<other files> - Source code to your VM implementation

task2/

result.txt - Output from your program for task2.

source-code/

overview.txt - Brief overview of the files in this

directory.

<other files> - Source code to your solution to task2.

The zip-file should be named with your full name. For example: "John

Smith.zip". An example zip-file has been provided.

Please DO NOT:

\* Include ANY non ASCII-text files in your submission.

\* Supply any files in any directories other than those listed above.

\* Create any further sub-directories in the source-code directories.

Internally we use a script to turn this zip-file into a report we use when

assessing your answer. As such, please follow the above specification as

rigorously as possible and examine the example zip-file provided carefully.

Details of where to send your zip-file should have been provided with in the

e-mail accompanying this specification.

Task 1 - Creating the VM

========================

For this part of the exercise the primary goal is to create a VM following the

specification below. Once implemented, the VM will be able to execute the

provided 'task1.bin' program image which when run will reveal the second task.

Executing the provided program images is a two step process. Firstly, the files

must be loaded into memory. Once loaded, their contents must then be

interpreted as a mix of instructions and data values. As instructions are

processed output will be produced one-byte-at-time as ASCII text.

Loading VM images

-----------------

The VM images themselves are ASCII text with UNIX-style line-endings. They

consist of a series of 32-bit signed integers in hexadecimal only. To load these

files, use the following process:

1. Read the first 32-bit signed value. We shall call this value

'data-size'.

2. Allocate an array called 'data' of signed 32-bit integers capable of holding

'data-size' elements. This array should be indexable from 0 to (data-size - 1)

(inclusive).

3. Read the second 32-bit signed value. We shall call this value 'image-size'.

4. Read 'image-size' values sequentially into the 'data' array

starting from index 0.

Interpreting VM images

-----------------------

Once loaded into memory, the VM images are executed by interpreting their

instructions one at a time from the 'data' array.

During execution program state will be held in the data array and two 32-bit

signed variables (registers):

\* 'ip' - initialised to 0.

\* 'sp' - initialised to 'data-size'.

General execution in the VM operates as expressed below in pseudo-code:

Do

current instruction = data[ip]

ip = ip + 1

decode instruction

perform action based on operation (see below)

Loop forever

Instructions are decoded by breaking them into three parts: 'binop',

'operation' and 'optional data'. These are stored as bit-fields within a

single 32-bit instruction as follows:

binop (1 bit)

/

+--+----------+-----------------------------+

| |operation | optional data |

| |(7 bits) | (24 bits) |

+--+----------+-----------------------------+

Bit: 31 30 ... 23 ... 0

Actions to be performed based on the 'operation' are defined below.

VM Instructions

---------------

VM Operations are divided into two classes based on the 'binop' bit at the

top of the instruction. Instructions in both classes are defined below.

Note that many of these use the functions "f" and "g" which are defined by the

following pseudo-code:

f(v):

sp = sp - 1

data[sp] = v

g():

v = data[sp]

sp = sp + 1

return v

Binop = 0 instructions

----------------------

operation = 0 - "pop"

sp = sp + 1

operation = 1 - "push <const>"

f(optional data)

operation = 2 - "push ip"

f(ip)

operation = 3 - "push sp"

f(sp)

operation = 4 - "load"

addr = g()

f(data[addr])

operation = 5 - "store"

st\_data = g()

addr = g()

data[addr] = st\_data

operation = 6 - "jmp"

cond = g()

addr = g()

if cond is not equal to zero then set ip = addr

operation = 7 - "not"

if g() equals 0 then f(1) else f(0)

operation = 8 - "putc"

output exactly one byte = (g() & 0xff) to stdout

Note: Output from the supplied VM images will always be ASCII text when

functioning correctly and will use '\n' (= 0x0A) to indicate new-line.

operation = 9 - "getc"

x = read exactly one byte from stdin

x = cast x to 32bits

f(x & 0xff)

operation = 10 (0x0A) - halt

Stop execution

Binop = 1 instructions

----------------------

All the instructions in this class involve getting two values, a and b,

using g() and storing the result using f() as follows:

b = g()

a = g()

f(a <op> b)

The exact <op> depends on the value of the 'operation' field as follows:

0 => add, <op> is +

1 => sub, <op> is -

2 => mul, <op> is \*

3 => div, <op> is /

4 => and, <op> is &

5 => or, <op> is |

6 => xor, <op> is ^

7 => eq, <op> is =

8 => lt, <op> is <

Notes:

\* For all operations, a and b and the result of <op> should be treated as

signed 32-bit integers.

\* For eq and lt, the result of the <op> should be exactly 0 or 1 only.

E.g. 1 < 10 => 1 and 10 = 10 => 1.

\* Division must be evaluated as integer division, with the result

truncated. E.g. 3 / 2 = 1.

Test Suite

----------

As implementing any software from a spec is potentially error prone, a set of

tests have been provided to help you debug your VM implementation.

These test programs have names of the form testXX.bin. When executed, all of

these tests should print a single ASCII 'X' character and exit immediately. A

set of testXX.asm files illustrate which aspects of the VM the corresponding

.bin file is testing. It is NOT necessary as part of this exercise to

implement software to process the .asm files.

Note there is no test of 'getc' functionality.