CSP 701
ASSIGNMENT 1

ARMv8 Simulator

In this assignment you will develop an ARMv8 instruction set simulator. Given an ELF format binary for the ARM architecture your simulator should simulate it. The instructions appearing in the binary will be a subset of the ARMv8 instuction set. The simulator will have to parse the ELF format binary, extract the instructions (along with other information), decode them and take actions accordingly. Your simulator is expected to keep track of the state of the memory and all the registers

Along with the simulator you are expected to develop and integrate a tiny debugger. The assignment will have 2 phases and has to be done in groups of **two.**

PHASE 1 [Deadline: Aug 12 2014 23:59 PM]

In the first phase you are expected to have the basic structure ready. At the end of this phase you must have written code to parse the ELF file, extract the instructions and have data structures for registers. Your simulator must be able to simulate straight line code (without any branches). You not expected to handle load stores yet but other instructions such as arithmetic/logic must be handled.

A basic skeleton of the debugger must be included. Functionality such as printing contents of registers etc. should be there.

PHASE 2 [Deadline: Aug 20 2014 23:50 PM]

You will add support for branch instructions and handle instruction addresses as well. Appropriate data structures for the memory need to be written. Loads/Stores and support for all other instructions must be provided along with a fully functional debugger.

- **It is recommended to start early**
- You are allowed to use a library[1] of your choice to parse the ELF binary.
- Submissions for each phase must be made on SAKAI as well as SVN before the respective deadlines. See the submissions section for more detail.
- Refer to the ARMv8 reference manual[2] for information on the instruction set and the encoding for the different instructions.
- A list of instructions that will appear in the binary will be provided in a separate file. We will also provide you with a few binaries and their disassembled output.
- You will want to generate your own test cases. For this you will need to setup a cross compiler. Refer to the cross compiler section for this.
- More detailed notes on how the debugger should function can be found in the debugger section.
- It is recommended to use git for local development.

SETTING UP A CROSS-COMPILER
(for writing testcases)

You cannot use ordinary gcc to compile ARM code (since it compiles for x86). You need to use a separate tool chain (or a gcc that is configure with target as ARM). You can use Linaro-gcc[3].

1. Download "gcc-linaro-aarch64-linux-gnu-4.9-2014.07_linux.tar.xz".
2. Extract it and you are good to go.

Note: 64-bit machines may require ia32-libs, lib32ncurses and lsb packages to be installed (apt-get install lsb ia32-libs lib32ncurses5).

You can now compile using *aarch64-linux-gnu-gcc,* disassemble the binary using *aarch64-linux-gnu-objdump* and debug using *aarch64-linux-gnu-gdb*.

For those of you who want to run your ARM code, you could use QEMU[4]. QEMU is a hosted hypervisor but also has user-mode emulation for ARMv8. Download the latest version of QEMU and follow the steps:

1. tar -xvf qemu-2.1.0.tar.gz
2. cd qemu-2.1.0
3. mkdir build
4. cd build
5. ../configure –target-list=aarch64-linux-user
6. make

./aarch64-linux-user/qemu-aarch64 <path-to-binary>
Note: The binary must be statically linked so use the - -static flag while compiling.

DEBUGGING with GDB+QEMU

Generate debug symbols when you compile (-g compiler flag).

Add the -g 1234 option to qemu (./aarch64-linux-user/qemu-aarch64 -g 1234 <path-to-binary>). The -g option pauses it so that you can attach a debugger to it on port 1234. Go ahead and attach aarch64-linux-gnu-gdb to qemu's gdb stub on port 1234 by launching aarch64-linux-gnu-gdb and the typing the following commands at the gdb prompt:

- gdb) file <path-to-binary>
- gdb) target remote localhost:1234
- gdb) list

And then you can proceed debugging (continue) .

SUBMISSION DETAILS

Along with your sources, you must also submit a **makefile** that will build your simulator along with any **external library files** that you use and a few **testcases** of your own. A good testcase would be a program that sums up all the integers in an array (written in ARM assembly, compiled without stdlib using the cross compiler). Before each deadline you must make a commit on SVN.

Submissions for sakai must be in the form of zip files with the format.
<roll1>_<roll2>_CSPA1.zip.

Instructions for setting up SVN:

```
$ ssh ssh1.iitd.ernet.in   # use your proxy password
$ mkdir svn
$ cd svn
$ svnadmin create armv8sim-repo
```

You should see a subdirectory called armv8sim-repo in the svn directory. Read armv8sim-repo/README.txt. Do not add, delete, or modify any files in this subdirectory. Edit the svn/authz file to provide read-write permissions to armv8sim-repo to yourself and your partners, and read permissions to mcs132552, mcs132543, mcs138296, mcs132569. For example, if users cs1012345 and cs1067890 are partners and user cs1012345 is currently logged in, here is a sample authz file to use:

[/]
[armv8sim-repo:/]
cs1012345@IITD.ERNET.IN = rw

cs1067890@IITD.ERNET.IN = rw

mcs132552@IITD.ERNET.IN = r

mcs132543@IITD.ERNET.IN = r

mcs132569@IITD.ERNET.IN = r

mcs138296@IITD.ERNET.IN = r

Go to one of the machines in the department cluster (or your local machine). Use the following command to checkout the files from your repository:

$ svn checkout https://svn.iitd.ernet.in/~cs1012345/armv8sim-repo

You will be prompted for your CSC password. Do not offer to save the password in plaintext format in your home directory as that is insecure. You can add the following line to~/.subversion/servers to avoid getting prompted for saving passwords in future:

store-plaintext-passwords = no

DEBUGGER

Your simulator must have a debugging mode. Preferably it should be activated by passing the “--debug” option on the command line. When in debug mode the simulator should stop so that the user can set breakpoints. Use GDB and get a feel for it. Your debugger must have similar experience too.

The following commands must be supported:
- break <ip> : puts a breakpoint at address <ip>. Ex: “`break 0x40ab0400`”
- del <ip>: removes breakpoint at address <ip> Ex: “`del 0x40ab0400`”
- run : starts the program execution
- s : executes 1 machine instruction and pauses again.
- c: continues execution till the next breakpoint is hit (execution is paused again)

- print <x/d> [<eg> : Prints the contents of the register “reg” in either hexadecimal (x) or decimal (d)
Ex: “print x r3” should print the value of register r3 in hexadecimal.

- print <num><b/w/d> <x/d> <mem-add>: Prints “num” bytes(b – 8 bits) / word(w – 32 bits) / double word(d – 64 bits) in hexadecimal (x) or decimal (d) from start of memory address <mem-addr>.
Ex: “`print 4w x 0x40ab0400`” will print `4` words from the start of `0x40ab0400` in hexadecimal. *Bonus for pretty printing (space after each byte/word/doubleword depending on whether it was b/w/d etc.).*

- [Bonus] watch <reg>: Stop execution after the first instruction that changes the value of register <reg>.

REFERENCES

[1] ELFIO http://elfio.sourceforge.net/
[2] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.architecture/index.html (registration required to download the PDF)
[3] http://releases.linaro.org/latest/components/toolchain/binaries/
[4] http://qemu.org

Git book: http://git-scm.com/book
ELF Format: http://www.cse.iitd.ac.in/~sbansal/os/ref/elf.pdf
SVN Manual: http://svnbook.red-bean.com/en/1.7/index.html
MakefileTutorial: http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

For anything not mentioned - Google and Stackoverflow are your best friends