Introduction to the μMPS2 Emulator and Development Environemt

Tomislav Jonjic

December 7, 2012

Outline

Introduction

- Your educational OS project will target μMPS2—a hypothetical streamlined computer architecture designed on purpose for this kind of application
- You will be using an emulator to run your programs, of course
- In this tutorial, we are going to cover:
  - The essentials of the μMPS2 architecture
  - The development environment: the cross-development toolchain, μMPS support libraries, etc.
  - Emulator usage

Outline

Processor Architecture

- μMPS2 processors implement a slight superset of the base MIPS-I architecture.
- Historical note: MIPS-I is the name retrospectively attributed to the ISA of the earliest MIPS processors—the R2000 and the R3000.
- μMPS2 differs somewhat from the R2000 in the system control coprocessor (CP0) interface:
  - It introduces two modes of operation of the MMU
    - Physical memory mode: address translation is not performed
    - Virtual memory mode: address translation is performed for all addresses, apart from those in the MMIO/ROM range
  - It includes an integrated interval timer (see the `Timer` CP0 register)
- New instructions: `cas` (compare-and-set) and `wait` (wait for interrupt)

Devices

- μMPS2 supports 5 types of peripheral devices:
  - disks
  - tapes
  - network adapters
  - printers
  - terminals
- Up to eight instances of each device type may be installed
- Device controllers support an interrupt-driven programming model
  - Devices of the same type share a single interrupt line
- All device controllers use MMIO and expose a similar programming interface

Multiprocessor Support

- A μMPS2 machine supports up to 16 processors
- Processors are controlled via a set of **machine control registers**
- μMPS2 includes a multiprocessor interrupt controller (MPIC)
  - The MPIC controls distribution of interrupts to available processors
  - For each interrupt source, the programmer can specify the policy which determines a target CPU for an interrupt from the source in question
  - The above is accomplished by programming a structure called the **interrupt routing table** (IRT)

The BIOS

- A μMPS2 machine needs to be provided with some basic ROM code in order to operate at all
- Most importantly, the memory region onto which the ROM code is mapped contains the exception entry points
- Two ROM modules must be supplied: a **bootstrap** ROM image and an **execution** ROM image
- The default execution ROM image is **`exec.rom.umps`**, installed under **`<prefix>/share/umps2`**
- Some convenient higher-level services are implemented by the **`exec.rom.umps`** code:
  - A more elaborate exception-handling scheme
  - A segmented/paged virtual memory scheme

Outline

Outline

Programming for μMPS2

- You are expected to implement your OS in the C language
- As we have seen, μMPS2 is centered around the MIPS architecture
- The above implies that we can use unmodified compilers which support a MIPS back-end to target μMPS2
- A the moment, the most viable (and perhaps the only) option is given by GCC

The Cross Development Toolchain

- You cannot use the host's compiler and associated set of development tools you (may) have installed on you system to build programs for μMPS2/MIPS.
- Note: a single GCC/Binutils build supports a single target
- We need separate tools, configured and built for a MIPS target:
  - The compiler (GCC)
  - GNU Binutils: `as`, `ld`, ...
- We refer to the above as a **cross toolchain**

Installing the Toolchain for Ubuntu Users

- There is an archive with μMPS2-related packages for Ubuntu: **https://launchpad.net/~jonjic/+archive/umps2**
- Simply add the relative **sources.list** entries to your system (see the above page for details).
- The toolchain consists of the following packages:
  - **umpsxt-binutils-mipsel**
  - **umpsxt-gcc-mipsel**
- Installed binaries follow the

  mipsel-elf-*tool*

  naming scheme; e.g., **mipsel-elf-gcc**, **mipsel-elf-ld**, ...

Building the Toolchain

- Binutils:
  - Configure as:

```
configure --prefix=<prefix> \
          --target=mipsel-elf \
          --disable-nls
```

  - Make and install as usual
- GCC:
  - Configure as:

```
configure --prefix=<prefix> \
          --target=mipsel-elf \
          --enable-languages=c \
          --without-headers \
          --disable-nls
```

  - We don't need to build/install all components:

```
make all-gcc all-target-libgcc
make install-gcc install-target-libgcc
```

Outline

libumps

- The emulator ships with a small utility library called `libumps`
- `libumps` provides:
  - Abstractions for BIOS service requests: `LDST()`, `INITCPU()`, ...
  - CP0 register accessors: `getPRID()`, `getSTATUS()`, `setSTATUS()`, ...
  - Simple wrappers for some processor instructions: `TLBWR()`, `WAIT()`, `CAS()`, ...
- All of this is in no way essential, but it's convenient, especially if one is not familiar with the MIPS assembly language

Using libumps in Your Programs

- Single header file: libumps.h
  - Installed in <prefix>/include/umps2/umps
  - Best practice: add <prefix>/include/umps2 to the list of searched dirs and use #include "umps/libumps.h"
- Implementation: libumps.S
  - Installed in <prefix>/share/umps2
  - Assemble and link to your kernel executable:

```
VPATH = $(UMPS_DATA_DIR)
...
%.o : %.S
        $(CC) $(CFLAGS) -c -o $@ $<

kernel : ... libumps.o ...
```

    - Where UMPS_DATA_DIR was set to the above dir

Backward Compatibility Notes

- The distribution also includes a pre-assembled relocatable module (**`libumps.o`**), installed under **`<prefix>/lib/umps2`**
- This is only done for backward compatibility reasons!
- This file is tied to a specific flavor of a specific MIPS ABI variant
  - For details, see the output of **`readelf -h libumps.o`**

Outline

μMPS .aout/.core Object File Formats

- A number of features are included with μMPS for user's convenience
- Among these is a specification for a simplified executable file format, along with the relative build-time/run-time support
- Two flavors of the executable format are provided:
  - The **.aout** format is aimed at regular executables
  - The **.core** format is aimed at kernel executables
- The `umps2-elf2umps` utility converts ELF object files to μMPS object files:
  - ELF relocatable modules can be converted to μMPS ROM images
  - ELF executable files can be converted to .aout/.core files

Linker Scripts for .aout/.core Executables

- Part of the support for .aout comes in terms of GNU ld linker scripts
- Each variant of the format has an associated linker script:

  umpsaout.ldscript
  : Standard linker script for .aout executables.

  umpscore.ldscript
  : Standard linker script for .core executables.
- The scripts are installed in **<prefix>/share/umps2**
- When using these scripts, the resulting ELF executable conforms to the memory layout requirements of the .aout format.

Outline

Program Startup

- Before your `main()` function can be called, the program environment has to be properly set up (e.g. critical registers, such as the stack pointer or the `$gp` register, have to be initialized).
- μMPS2 includes startup modules which should be adequate for most projects:

  `crtso.S` Startup code for .core executables

  `crti.S` Startup code for .aout executables
- The same backward compatibility/deprecation notes apply as for libumps!

Outline

MIPS ABIs and Related Issues

- A number of GCC options control MIPS-specific aspects of code generation
- The default values of these parameters can differ, and depend on how GCC itself was configured
- For example, in many cases GCC generates SVR4/Linux-style position independent code by default
  - While certainly good for its intended use, this is completely inadequate for your kernel!
- Rather than relying on compiler-provided defaults (in "cargo cult" fashion), learn to control code generation as appropriate

Sensible Compiler and Linker Flags for an OS Kernel

- GCC:

```
-ffreestanding
-mips1
-mabi=32
-mno-gpopt -G 0
-mno-abicalls -fno-pic
```

- Linker (`ld`):

```
-G 0
-nostdlib
```

Outline

Miscellaneous Headers

aout.h Definitions pertaining the .aout file header

cp0.h Definitions of processor control registers: `CP0_Index`, ..., `CP0_PRID` `STATUS_IEc`, ..., `STATUS_CU3`

arch.h Definitions of various machine registers (bus registers, interrupt controller, etc.)

types.h Definition of the CPU state structure (as specified by the BIOS exception handling interface), device registers

Note: like **umps.h**, all of these headers are found in **<prefix>/include/umps2/umps**

Outline

Hello World in μMPS2

- There is a "hello world" example included in the μMPS2 source distribution.
- The **Makefile** sums up part of the things we have covered thus far.

```
[tjonjic@soyuz hello-umps]$ ls
hello.c  Makefile
[tjonjic@soyuz hello-umps]$ make
mipsel-elf-gcc -ffreestanding -ansi -mips1 -mabi=32 -mno
mipsel-elf-gcc -ffreestanding -ansi -mips1 -mabi=32 -mno
mipsel-elf-gcc -ffreestanding -ansi -mips1 -mabi=32 -mno
mipsel-elf-ld -o kernel hello.o crtso.o libumps.o -G 0 -
umps2-elf2umps -k kernel
[tjonjic@soyuz hello-umps]$ ls
crtso.o  hello.o  kernel.core.umps  libumps.o
hello.c  kernel   kernel.stab.umps  Makefile
```

- **kernel.core.umps** is the .core kernel executable, ready to be loaded and run

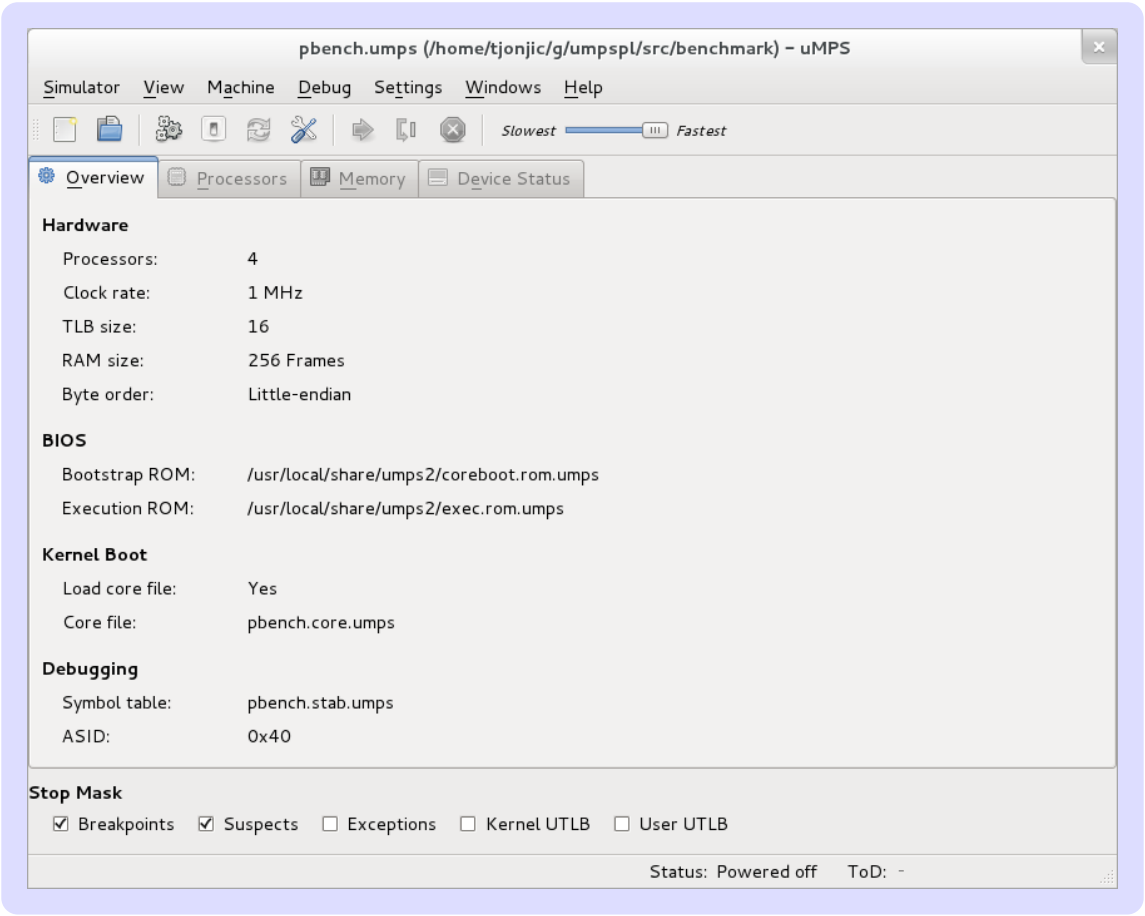
Outline

Machines and Machine Configurations

- A μMPS2 machine is described by a **machine configuration** file
- Machine configurations specify all user-settable machine parameters:
  - Number of CPUs and their characteristics (frequency, TLB size)
  - Installed RAM
  - Installed devices and the associated device files
  - ROM images
  - ...
- Machine configurations can be edited entirely within the emulator UI
  - Internally, these files employ a JSON-based syntax
- At any time, only a single machine configuration may be active (in a single instance of the emulator)

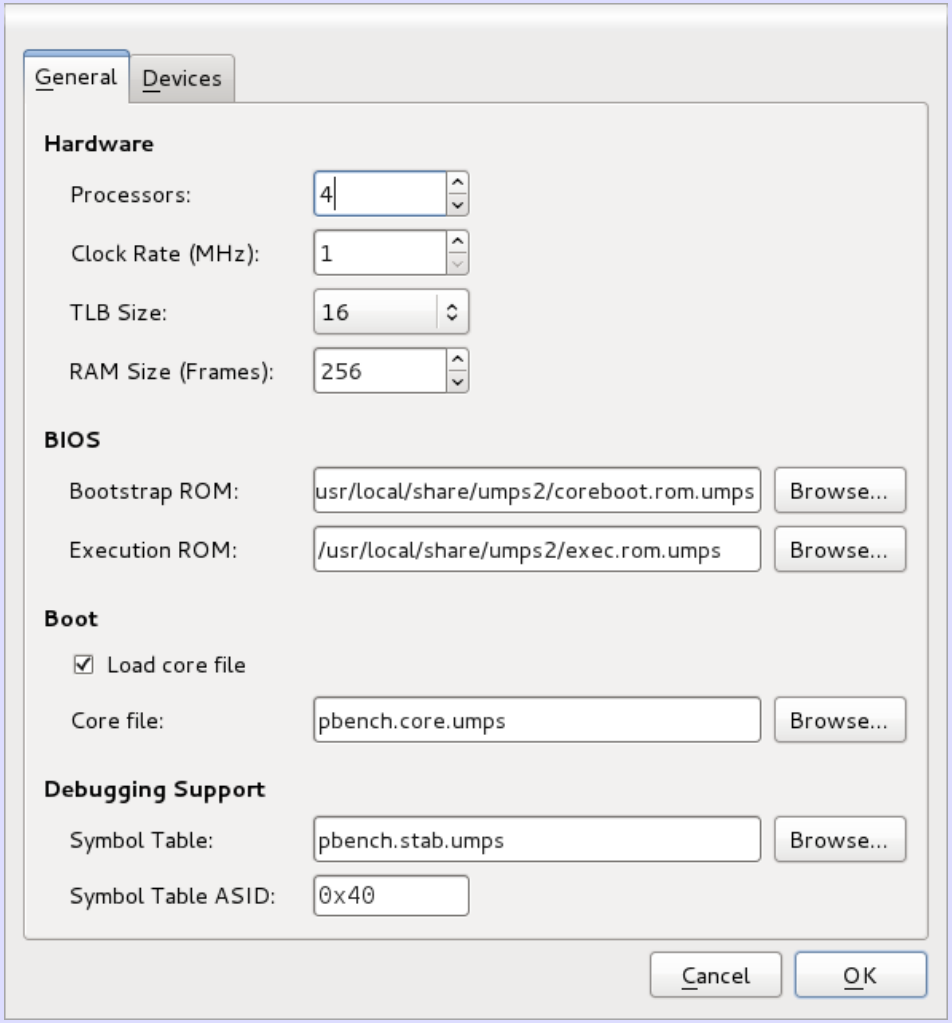
Outline

The Main Window



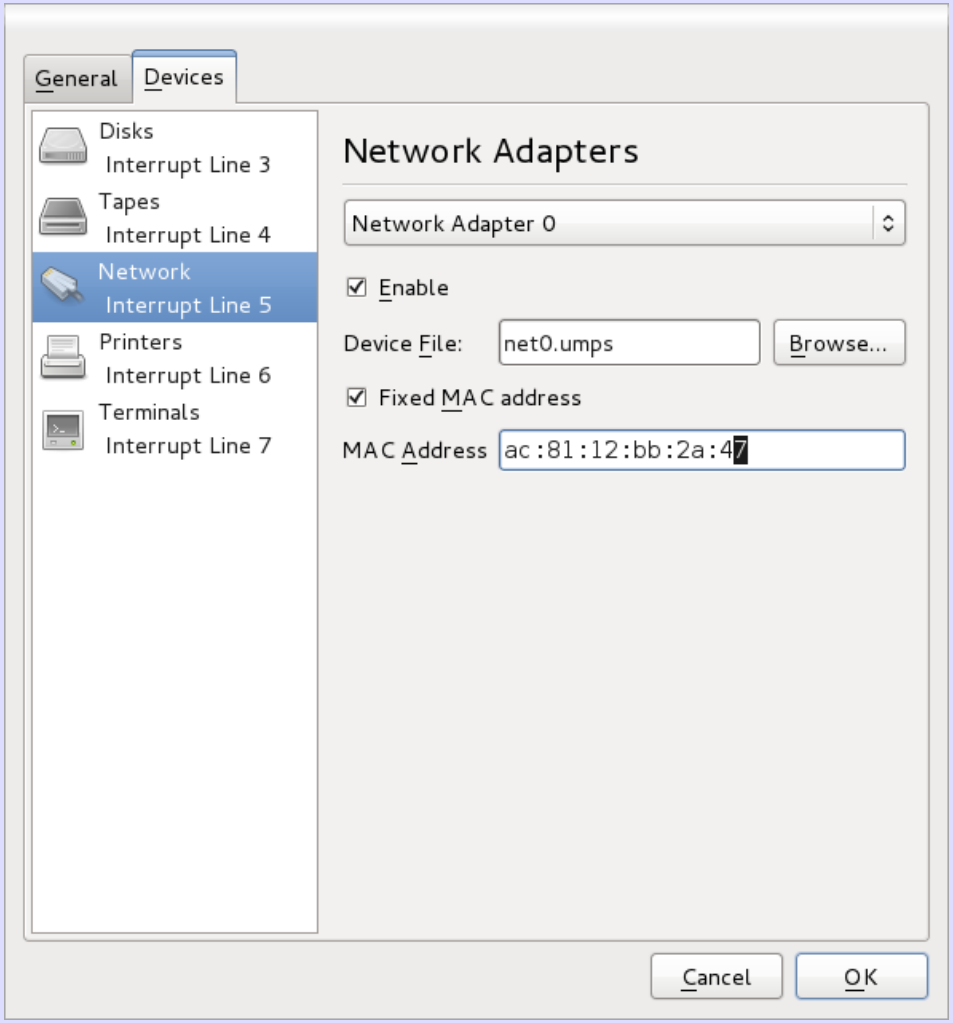
The central application window in μMPS2; it contains most execution/debugger related elements.

The Machine Configuration Dialog



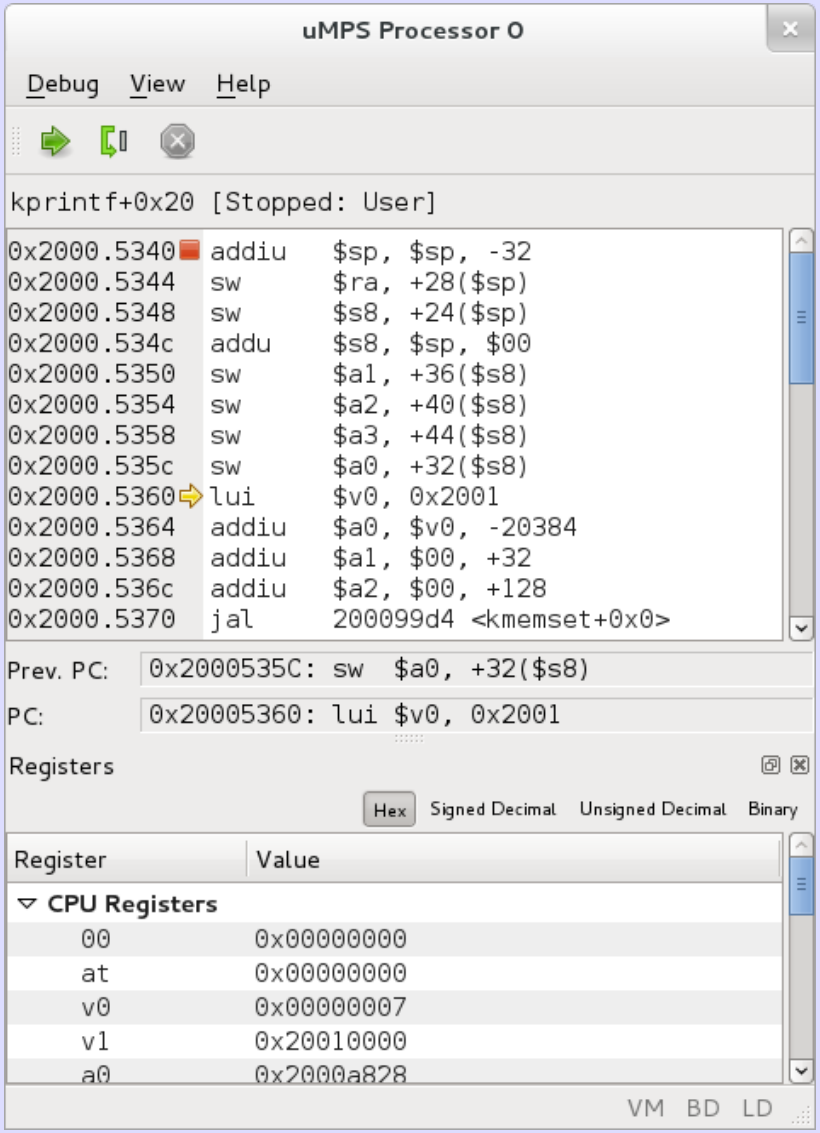
As already noted, machine parameters can be set within the UI.

The Machine Configuration Dialog



As already noted, machine parameters can be set within the UI.

Processor Windows



A processor window displays:

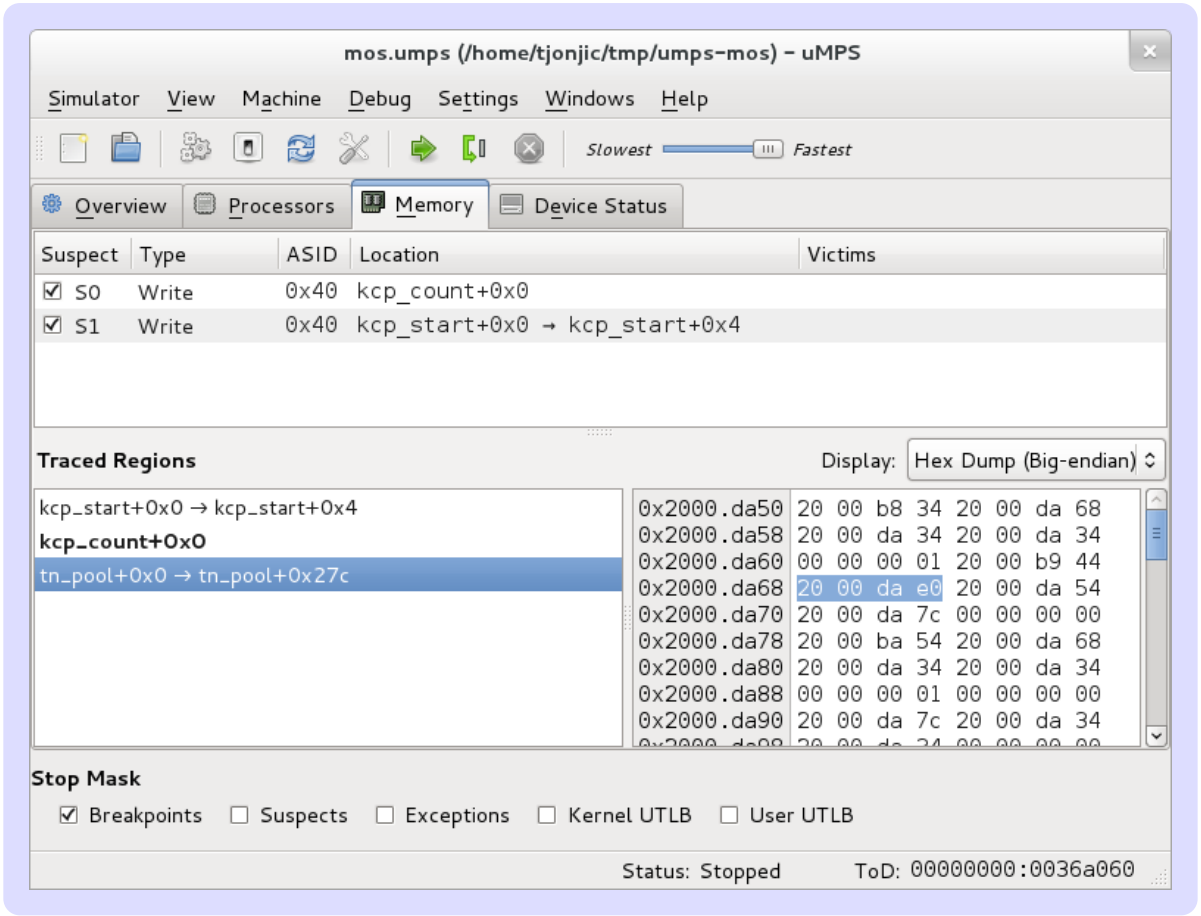
- The processor's status
- Architectural state (general purpose registers, CP0 registers, TLB, etc.)
- A disassembly of the currently executing function

Outline

Debugger Support in μMPS2

- The emulator includes fairly complete debugging support:
  - Breakpoints
  - Suspects
    - Commonly known as "data breakpoints", or "watches"
    - Machine execution is suspended on a read or (more commonly) write access to a memory location in a specified range
  - Memory tracepoints
- Machine execution is (almost) entirely deterministic
  - Under equal device input conditions, two emulation runs of a single machine will be identical
- Certain features aid debugging parallel systems
  - For example, it is easy to identify which CPU triggered a particular breakpoint or suspect

Memory View



Traced memory regions may be displayed in two ways: ASCII and hex; the selected region may be edited in-place.

Debugger Variables and Workflow

- A typical debugging session consists of one or more iterations of the following steps:
  - A condition that exhibits the bug is triggered, and an attempt is made to infer its cause
  - The required change to the source is applied, and the program is recompiled
  - The machine is reset (to reload the kernel)
- After recompilation, the old memory addresses associated with breakpoints might not be correct
- The emulator attempts to relocate breakpoints/suspects/tracepoints to the correct locations