The Terma Emulator Evolution

Dr. Mattias Holm
Terma
Shuttersveld 9
2316 XG Leiden
Netherlands
maho@terma.com

March 2, 2015

Abstract

Emulators and virtual platforms are essential in the development of any type of embedded platform. This paper introduces T-EMU 2.0 the next generation micro processor emulator and full system simulator framework from Terma. T-EMU 2.0 attempts to solve a common problem, where an emulator is either timing accurate or has high performance, but not both. T-EMU 2, has been designed with both maximum flexibility, and high performance in mind. At the centre of the new emulator lies the LLVM compiler framework, which enables custom code transformations at both compile and run-time.

T-EMU 2 also introduces a new object system based Application Programming Interface (API) for device modelling. It provides a plugin-concept for modelling of devices, enabling device models to be modelled inside the emulator or outside (e.g. using SMP2).

1 Introduction

Micro-processor emulators are (mostly) PC-based software solutions, used in the development, testing, deployment and maintenance of software.

During development (and unit testing), emulators offers more flexibility and often a more cost effective means for developers to run code on the deployment platform. Another advantage when developing hardware and software together is that an emulator based solution is often available before stable hardware can be produced, so it is in the end the only way to run code on the target platform during development phase. This is not just an advantage in the space sector, for example, when NetBSD was ported to the AMD64, they used the Simics full system simulator, for the porting [2].

During testing of software, emulators are used in for example Software Validation Facilities to provide an accurate model of the hardware and environment without having the need for physical access to the actual hardware.

During deployment of a spacecraft, emulators are used inside operational simulators for training operators and testing operational procedures.

Finally, during maintenance, emulators are critically important as the hardware may not even be available anymore.

The emulators available today often vary in terms of accuracy and performance. While instruction behavioural accuracy is relatively easy to obtain without sacrificing too much performance, obtaining timing accuracy and high performance at the same time is often not possible when the target system has caches (e.g. LEON2 and later).

For operational simulators and simulators with hardware in the loop, real-time performance of the emulator is often required. While this is achievable when emulating the ERC32 and LEON2 which run at less than 100 MHz, the next generation flight processors will have clocks at several hundreds of MHz and come in multi-core configurations.

This results in a significant problem for emulators. Firstly, real-time performance must be boosted by a factor of 4 to handle at least a single core of the NGMP processor to be simulated in real-time, and secondly, multi-core emulation needs to be supported.

Boosting performance of an interpreted emulator such as TSIM and the ESOC Emulator should in principle be very simple, that is, one would add support for binary translation. The problem with this is that if an emulator has been designed for interpretation from the start, it may not be practical to add binary translation to the system later. As binary translation has an effect on the design on just about everything in the emulator core.

It was shown in [6] that the emulators currently used by the European Space sector do not have the highest performance that is achievable. While e.g. the ESOC Emulator provides relatively high performance given its architecture, it cannot handle the future multi-core processors in real-time as it stands at the moment.

Faster emulator systems (e.g. Simics and OVPSim) which have MIPS ratings five times that of the ESOC Emulator [4, 5], often drop accuracy by for example omitting accurate cache emulation or assuming an instruction takes only one cycle.

T-EMU 2.0 was born from the need to overcome all of the mentioned issues. Firstly, T-EMU 2.0 is designed to be flexible for the future, it is for example relatively easy to integrate both dynamic and static binary translation in the system. Secondly, while performance is key in T-EMU 2.0, it is flexible enough to allow for the integration of cycle accurate cache models if needed. Thirdly, the T-EMU 2.0 architecture is prepared with support for the emulation of multi-core processors. While supported by some systems (e.g. Simics and QEMU), multi-core support is currently lacking in both TSIM and the ESOC Emulator. Cycle exact multi-core support is available in GRSIM, but GRSIM runs at only 5 MIPS on high end PCs [1] and this is far below needed performance levels for applications such as operational simulators.

2 Architectural Overview

2.1 T-EMU and LLVM

LLVM is a compiler framework for whole program optimisation and code generation. It provides an *intermediate representation* (IR), which can be expressed in human readable text format as LLVM assembler or in a binary format known as LLVM bitcode.

The LLVM framework provides a C++ API for writing code transformation plugins that analyse and / or transforms the IR. It also provides tools such as TableGen that are suited for describing instruction formats and attributes.

LLVM is used in T-EMU for a number of reasons. One of the reasons is that TableGen simplifies the modelling of a processor core in a way that is emulation method independent. The same instruction description can power both a decode-dispatch interpreter, a threaded interpreter and a binary translator. The TableGen files also simplifies maintenance of multiple instruction decoders, since the opcodes are all centralised to one location. Not only the instruction formats and opcodes are described, but TableGen is used as a macro processor for instruction semantics implemented in LLVM assembler.

An example instruction in LLVM TableGen and assembler format is illustrated in Listing 1. In that example two variants of an add instruction are described, the add-register-register and add-register-immediate variants.

Listing 1: Example TableGen Instruction Description

```
multiclass ri_inst_alu<bits<2> op, bits<6> op3, string asm, code sem>
{
  class _rr {
    let i = 0;
    let semantics = [{
        %r1 = call i32 @emu.getReg(i5 %rs1)
        %r2 = call i32 @emu.getReg(i5 %rs2)
    }] # sem # [{
        call void @emu.setReg(i5 %rd, i32 %res)
    }];
  }
  class _ri {
    let i = 1; // Immediate variant, note that inst{13} =  i
    let semantics = [{
        %r1 = call i32 @emu.getReg(i5 %rs1)
        ;; Sign extend instruction immediate field
        %r2 = sext i13 %simm13 to i32
    }] # sem # [{
        call void @emu.setReg(i5 %rd, i32 %res)
    }];
  }
}
```

```
defm add : ri_inst_alu<0b10, 0b1010101, "add", [{
    %res = add i32 %r1, %r2
}]>;
```

In addition to the use of TableGen for describing instruction formats and as a macro processor for LLVM assembler. The LLVM framework enables us to write specific compiler optimisations targeting just our emulator. This is used not just for optimisation but also for executing code transformations between different emulator mechanisms. LLVM also contain a JIT engine, which we plan to use in a future version of T-EMU.

2.2 Object System APIs

The emulator is written around a C API providing an object system for the emulator. The object system provides support for classes, objects, properties and interfaces. Both CPU, and device models are implemented using this system.

While interfaces can be supported in C++, it was decided to provide a C API for a number of reasons. Firstly the C API can be easily wrapped for scripting languages, enabling the user to make a model prototype in languages such as Python. Secondly, as C++ does not have introspection support, manual class and property registration would still be needed. In the future (when C++17 compile time reflection support is ready), we may revisit this and provide an alternate C++ modelling API as well. Thirdly, some of the interfaces (notably the *Memory Access Interface* (see Section 2.4) must be possible to invoke from a well defined ABI[1], and C++ does not provide such an ABI at this moment.

The object system is similar in some aspects to *gobject* and parts of *SMP2*, but is a lot more light weight. In essence, the programmer defines a record (which must be a *POD type*) and registers this with the object system. The best way is to illustrate the object system with example code (see 2). Note that the API is exposed as C for several reasons, one being that the emulator core must be able to interact with some of the interfaces.

In the example, a class called *MyDevice* is registered. The class contains two properties, a 32 bit register named *RegA* and an interface reference (a pair of an object pointer and an interface pointer). *MyDevice* also supports the Foo-interface, this interface is registered with the object system. The *register* function registers the classes (this is normally done by a plugin init function called when a shared library with a T-EMU plugin has been loaded). *CreateObjectAndRaise* illustrate how to create objects, connect them together and how to call interfaces implemented by other objects.

Listing 2: Object System Example

```
typedef struct {
  uint32_t RegA;
  temu_IrqObj IrqCtrl; // Interface-reference
} MyDevice;

temu_FooIface Foo = {
  bar, // void (*bar)(void *Obj, int A)
};

void
register(void)
{
  auto Cls = temu_registerClass("MyDevice", sizeof(MyDevice));
  temu_addProperty(Cls, "RegA", teTY_U32, offsetof(MyDevice, RegA),
                   nullptr, nullptr);

  temu_addProperty(Cls, "IrqCtrl", teTY_IfaceRef, offsetof(MyDevice, IrqCtrl),
                   nullptr, nullptr);

  temu_addIface(Cls, "Foo", &Foo);
}

void
createObjectAndRaise()
```

[1] Application Binary Interface

```
{
  // Create new object named "Device0" of class "MyDevice"
  MyDevice *Dev = temu_createObject("MyDevice", "Device0");

  // Find object named "IrqDevice0" created elsewhere.
  void *IrqObj = temu_getObject("IrqDevice0");

  // Connect the IrqCtrl property in Device0 to the IrqCtrlIface
  // from IrqDevice0
  temu_connect(Dev, "IrqCtrl", IrqObj, "IrqCtrlIface");

  // Invoke the raiseIrq function provided by the interface.
  Dev->IrqCtrl.Iface->raiseIrq(Dev->IrqCtrl.Obj, 1);
}
```

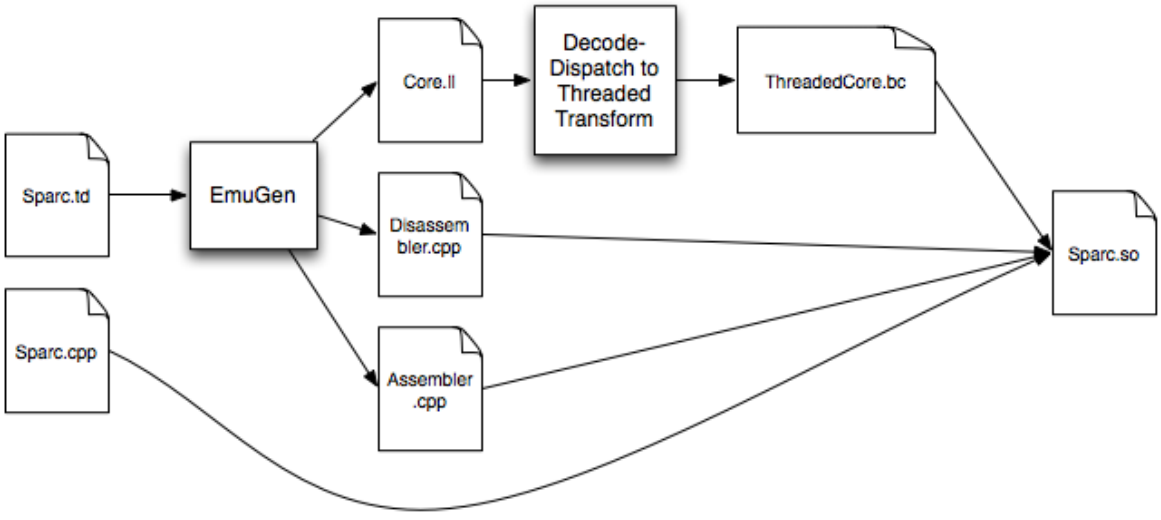
2.3 Microprocessor Models

Microprocessor models in T-EMU 2 are currently written by Terma due to the non-triviality in developing these. At the moment the ERC32, LEON2 and LEON3 are supported, but the system has been designed so additional CPUs, even of different architectures such as the ARM can easily be added. To do this we simply add a new TableGen file describing the new CPU architecture.

When building T-EMU 2, TableGen files describing a processor model (registers, instruction opcodes and instruction semantics) are loaded by the *emulator generator* which in turn generates instruction decoders from the opcodes, instruction implementation functions from the semantics and CPU types from the register descriptions. The tool also (semi-automatically) generates an assembler and disassembler.

The general compilation pipeline used in T-EMU to build a CPU model is illustaded in Figure 1. The TableGen file (Sparc.td) is loaded by the emulator generator, the emulator generator generates three files, one assembler, one disassembler and one emulator core (Core.ll in LLVM assembler). The core file can be either compiled as a decode-dispatch based interpreter (but this is only done during core development), or it can be transformed to a threaded interpreter. This transformation inlines all functions implementing instruction semantics into one single emulator function and converts the decode-dispatch instruction decoder so it decodes to labels in the emulator function and not to individual functions.

Figure 1: Compilation Pipeline



In addition to the TableGen file, a C++ file implementing high level functionality for the processor model is compiled and linked with the threaded code function. This file implements for example the plugin interface and the relevant CPU interfaces, e.g. the interrupt controller interface.

An emulator core does not use any global state, and as such it is easy to run multiple CPU cores in the same emulator system. In fact, in addition to performance and flexibility goals, T-EMU 2 has been designed to support multi-core systems.

2.4 Memory Models

One of the more performance sensitive subsystems in an emulator is the memory system. The memory system has to deal, not only with memory (RAM and ROM), but also with device models. All of which will be mapped to a memory address range. To ensure the efficiency of the memory emulation, T-EMU 2 maintains a 2 level page table in a memory space object. This memory space object maps out the full 36 bit physical address space of the SPARC architecture. When a device or memory object is created, it is a free standing object, that must be mapped to some memory space object. The mapping is done by assigning the object a physical address and size. Any write or read to these addresses from the software running on the processor will result in a call to the memory space objects read, write or fetch function.

Calling a function for every memory access would be too costly however, so in order to speed up memory accesses to RAM and ROM, the mapping between virtual target address and host address is saved into a small cache called *address translation cache* (ATC). This cache is checked on every memory access, and in case of a hit the memory can be accessed directly, in case of a miss, the CPU core will call the relevant function from its memory space object. This speeds up the access to RAM and ROM memories considerably. The cache is indexed by virtual address, and this implies that when an MMU model is enabled (e.g. on the LEON3), the ATC does an extra job as a software TLB.

The use of a single interface for memory accesses, enables the insertion of (multiple levels of) cache models before e.g. RAM and ROM memories, therefore creating a memory hierarchy. Exact cache models naturally slow down the memory simulation, and forces all accesses to go though the memory space system, but the ATC is still useful in this case as it not only tracks virtual to host address mappings, but also virtual to physical address mappings. Thus one can decide whether on want the emulator to run as fast as possible (by omitting the cache models) or if one wants accuracy (by inserting an accurate cache model). Even without a cache model, it is possible to model a cache with a rough statistical model, where every memory page has an individual access penalty. These penalty figures can for example be derived by running an accurate cache model and computing hit probabilities.

Memory and memory mapped device and models all implement the *Memory Access Interface*. This interface is relatively simple. The model developer writes three functions, one each for fetches, reads and writes. The emulator core will in case of a miss in the address cache, call the memory system and invoke the relevant function. The memory access functions gets as a parameter a memory transaction pointer, which is created and allocated by the emulator core. This transaction contain a number of fields. The use of a pointer to a transaction object, speeds up nested calls in the memory hierarchy as only a single pointer needs to be passed along. The memory access interface is shown in Listing 3.

Listing 3: Memory Access Interface

```
typedef struct temu_MemTransaction {
  uint64_t Va;       //!< 64 bit virtual for unified 32/64 bit interface.
  uint64_t Pa;       //!< 64 bit physical address
  uint64_t Value;    //!< Resulting value (or written value)

  //! Log size of the transaction size
  uint8_t Size;

  //! Used for device models, this will be filled in with the offset
  //! from the start address of the device (note it is in practice
  //! possible to add a device at multiple locations (which may happen in
  //! some rare cases)).
  uint64_t Offset;
  void *Initiator; //!< Initiator of the transaction
  void *Page;      //!< Page pointer (for address caching)
  uint64_t Cycles; //!< Cycle cost for memory access
} temu_MemTransaction;


typedef struct temu_MemAccessIface {
  void (*fetch)(void *Obj, temu_MemTransaction *Mt);
```

```
    void (*read)(void *Obj, temu_MemTransaction *Mt);
    void (*write)(void *Obj, temu_MemTransaction *Mt);
} temu_MemAccessIface;
```

2.5 Device Models

Device models are components in T-EMU, which are mapped into a memory address space object. Device models can be written by the emulator user using the public T-EMU C API. They are loaded as plugins and connected to the memory system at run-time following the user's emulator configuration.

Device models have access to common services from the emulator, such as posting of timed events on a processor's event queue, reading and writing memory blocks, raising interrupts, etc.

Due to the workings of the microprocessor models and the memory system, an access to a device model is practically free, it involves an ATC-miss, followed by an address translation (via a two level page table), and a function call to the device's memory access function. Note that this can be contrasted to several other emulators that either calls a single I/O function (leaving the address decoding to the simulator), or some that needs to flush hundreds of bytes to a stack before leaving the emulator core, e.g. T-EMU 1.0 and the ESOC Emulator must flush over half a kilobyte of data to the emulator core stack before calling an external MMIO model.

If native device models are used in T-EMU 2.0, address decoding is automatic and also fast. This can also be achieved with external models by implementing the memory access interface in or together with the external model. This means that the emulator core can do the address decoding in a highly efficient way. This differ from the normal TSIM and ESOC Emulator approach which utilise a single global MMIO callback function.

It is also possible to integrate the simulator with the emulator on the external bus level instead of MMIO level. T-EMU 2, being capable of full system simulation has its own bus models (e.g. for serial ports and 1553 busses).

Currently the MEC (ERC32), LEON2 on-chip devices and several LEON3/GRLIB models have been implemented. GRLIB models can easily be remapped to different addresses and the AHB/APB plug-and-play mechanism is easy to work with (i.e. more or less automatic for the user).

3 Performance

Performance of an emulator is best measured in MIPS (Millions of emulated Instructions Per Second), while executing some payload application. The standard payload application for measuring emulator performance is normally the Dhrystone application. While the Dhrystone application results is typically expressed in a figure called DMIPS, this rating is slightly artificial since it is highly dependent on compiler behaviour and the instruction set of the target platform.

Thus, when measuring emulator performance, the actual instruction count is measured and not the benchmark scores. In fact, the reported benchmark score will reflect the virtual clock frequency of the emulated target and not the performance in wall-clock time.

Further discussion on performance can be found in [6] and raw MIPS figures for TSIM are published at [3]. Figures 2 and 3 summarise the obtained performance with three different emulators. Note that performance numbers for TSIM with the MMU enabled where not available, so these are omitted from Figure 3.

Figure 2: Raw MIPS when running Dhrystone on a 3.4 GHz Intel (MMU disabled)

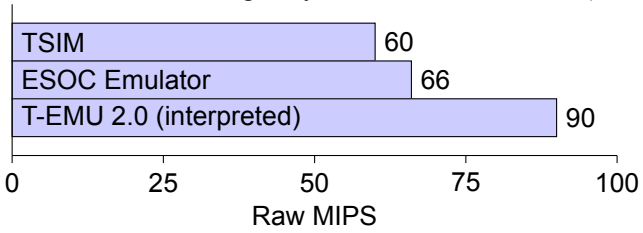
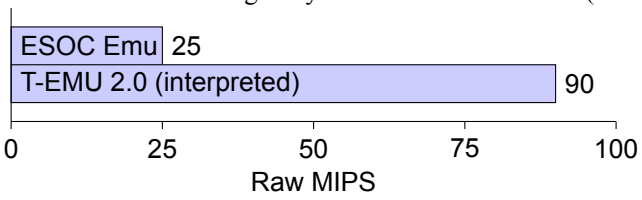


Figure 3: Raw MIPS when running Dhrystone on a 3.4 GHz Intel (MMU enabled)



4 Future Directions

T-EMU 2 has as a long-term goal is to build up a larger standard device model library, and the integration of additional CPU architectures (e.g. ARM, PowerPC, etc).

T-EMU 2 can be extended to contain an SMP2 scheduler and enable the integration of other types of device models such as for example System-C models. For SMP2, we are also looking into handling publication of registers and device state automatically. Most of the needed information already exists in the T-EMU 2 object system.

Binary translation is an obvious future extension to T-EMU, and has the potentiality to improve performance significantly for code that is executed repeatedly (which should be the case for most code running in an operational simulator for example). Our aim is to at least achieve performance like Simics which is capable of over 300 MIPS speeds (see [5, 4]). By using the LLVM framework for implementing the instruction semantics, we already have the instructions in a binary-translation friendly format. It should be added that T-EMU 2 can be extended to do both dynamic AND static binary translation. Static binary translation is of particular interest for operational simulators, while dynamic binary translation is a general purpose system.

A device/MMIO modelling *domain specific language* (DSL) is useful in the long term. Due to the high amount of boilerplate (independent on doing a model in C or C++), a DSL could be developed that simplifies the modelling of devices and registers. Such a DSL could in principle target multiple emulators using different DSL backends. We can imagine a backend targeting T-EMU 2+, and a second one targeting TSIM / ESOC Emu. This approach would potentially enable model portability while still not trying to force every emulator to expose the same API and therefore avoiding integrating the device in an suboptimal way on some emulators.

Current emulators, including T-EMU 2, implements floating point support using soft-float emulation. This is needed in order to ensure that architecture specific floating point rules are followed. However, for targets implementing only IEEE floating point (e.g. SPARC) running on IEEE floating point hosts, it is possible to add hardware driven floating point support (by relaxing some rules such as being exact in the emulation of NaN propagation). Note that this would not be done by replacing the current soft-float system, but rather provided as an extra option as it slightly modify the behaviour of floating point instructions compared to the actual hardware.

References

[1] GRSIM. `http://www.gaisler.com/index.php/products/simulators/grsim`.

[2] NetBSD/amd64. `http://wiki.netbsd.org/ports/amd64`.

[3] TSIM2 ERC32/LEON simulator. `http://www.gaisler.com/index.php/products/simulators/tsim`.

[4] Jakob Engblom and Dan Ekblom. Simics: a commercially proven full-system simulation framework. In *Simulation & EGSE Facilities for Space Programmes*, 2006.

[5] C. W. Mattias Holm. A Fully Virtual Multi-Node 1553 Bus Computer System. In *Data Systems in Aerospace (DASIA)*, 2006.

[6] Mattias Holm. Emulator Performance Study. In *Simulation & EGSE Facilities for Space Programmes*, 2015.