February 28 – March 1, 2012

Memory Debugging of Virtual Platforms

by
George F. Frazier
Neeti Bhatnagar
Qizhang Chao
Kathy Lang
Cadence Design Systems, Inc.

Virtual Prototypes (VP)

- VPs are models of systems that typically contain a hardware design and software.
- Embedded OS, device drivers, bare metal test programs, applications, etc.
- VPs model instructions on the processor(s) accurately so are appropriate for full scale pre-RTL software development.
- Types of software involved informs the debug strategy.
- Challenge is to create debug environments that provide a richness & visibility not available to environments that contain physical prototypes of the hardware.
- TLM standard can help.

SystemC VPs

- A SystemC-based VP is a SystemC/TLM-based model of the hardware along with software.
- Hardware components are written in SystemC or at least wrapped in TLM 2.0 interfaces.
- Includes hardware such as processor models and memories along with SystemC/TLM implementations of peripherals.
- Embedded software such as an embedded OS, drivers, and application software.
- TLM standard helps in creation of rich debug environments for both HW and SW for VP-based systems.

Role of Memory sub-systems in SystemC-based VPs

- Many possible memory implementations and configurations can exist.
- The program memory, data memory, and stack need not be stored in a single memory model.
- They can be stored in a single memory model but implemented with sparse memory or other advanced memory modeling techniques that are often predicated by performance.
- From a tooling perspective, we need a generic way to inspect and modify values of VP memories that is external to any implementation details of a particular system.
- This is where the TLM 2.0 standard assists us.

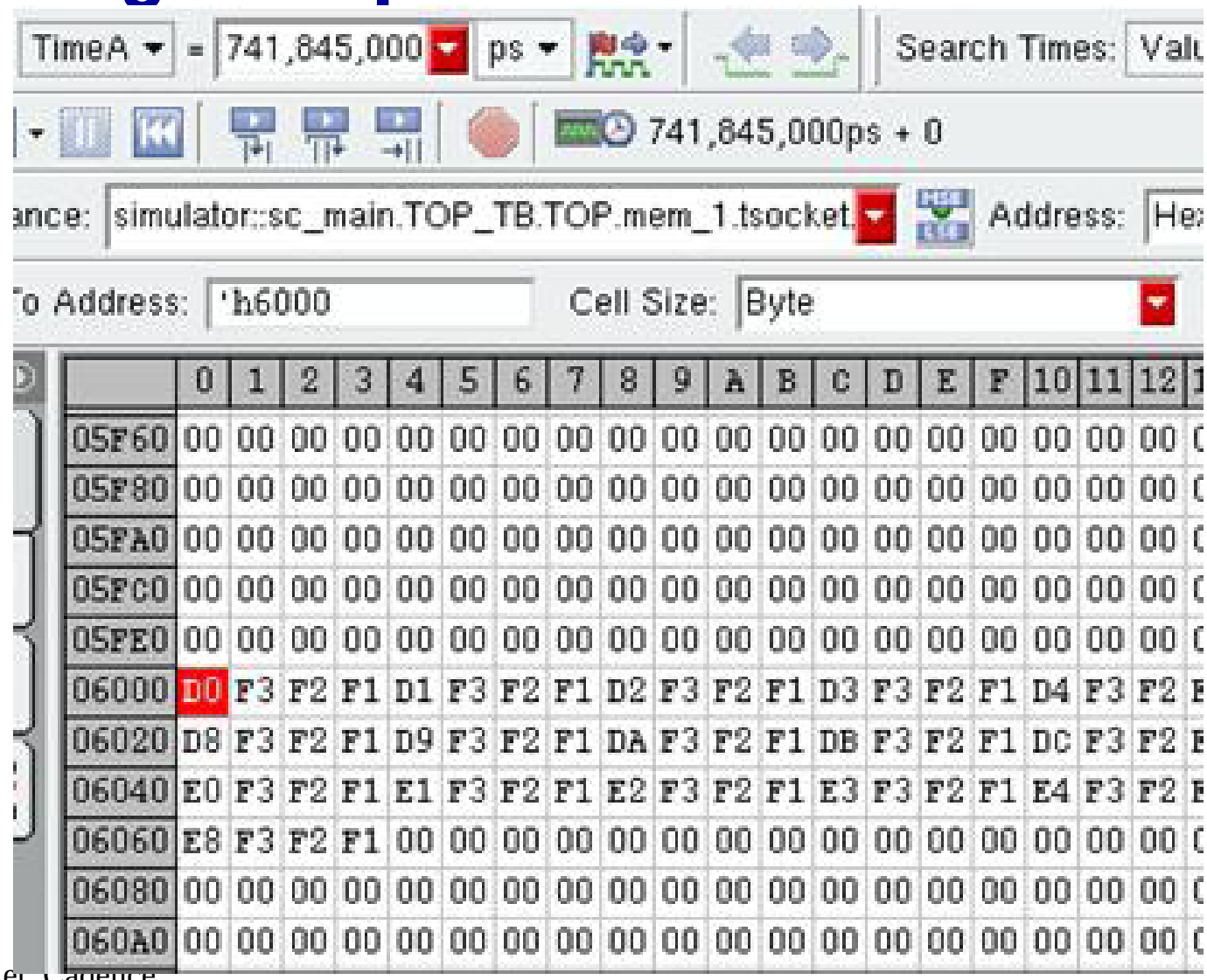
TLM 2.0 functions for inspecting target values

Function name	*description*	*Has side effects?*
b_transport	Blocking transport.	Yes
nb_transport_fw	Non-blocking forward transport	Yes
nb_transport_bw	Non-blocking backward transport	Yes
transport_dbg	Debug transport call	No

Provisioning a Memory "view" with transport_dbg

- If you know which blocks in a design represent memories, it is possible to collect the values of the memory for debugging purposes using transport_dbg calls.
- This can be done without side effects or advancing simulation time, and without modifying a user design.
- Values can be collected and presented as a "memory view"

Memory “Viewer” provisioned by TLM target inspection



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
05F60	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
05F80	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
05FA0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
05FC0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
05FE0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
06000	D0	F3	F2	F1	D1	F3	F2	F1	D2	F3	F2	F1	D3	F3	F2	F1	D4	F3	F2
06020	D8	F3	F2	F1	D9	F3	F2	F1	DA	F3	F2	F1	DB	F3	F2	F1	DC	F3	F2
06040	E0	F3	F2	F1	E1	F3	F2	F1	E2	F3	F2	F1	E3	F3	F2	F1	E4	F3	F2
06060	E8	F3	F2	F1	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
06080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
060A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

TLM VP that runs Android

- Two ARM processor models (processor models were generated by ARM tools and have a TLM wrapper).
- A9 and M3.
- TLM Simple Memory (RAM).
- Devices connected to the bus.
- System Memory Map.
- Android Goldfish.

Device name	*TLM Target*	*Local start address*	*Local end address*	*System base address*
ram	ram.tsocket	0	0x5FFFFFF	0
Interrupt controller	interrupt.tsocket	0	0xFFF	FF00000
timer	timer.tsocket	0	0xFFF	FF01000
tty	tty0.tsocket	0	0xFFF	FF02000
audio	audio.tsocket	0	0xFFF	FF03000
Battery	battery.tsocket	0	0xFFF	FF04000

Common Steps in VP development

- Choose your processors and how to model them (QEMU, Arm, Imperas, etc).
- Create SystemC peripherals as needed.
- Write drivers for any hardware peripherals you authored, if you are only extending a system you might only need drivers for new hardware blocks you added.
- Port embedded OS.
- Run the system, and test by running middleware on the OS.

Time to Debug!

VP Debugging Terminology

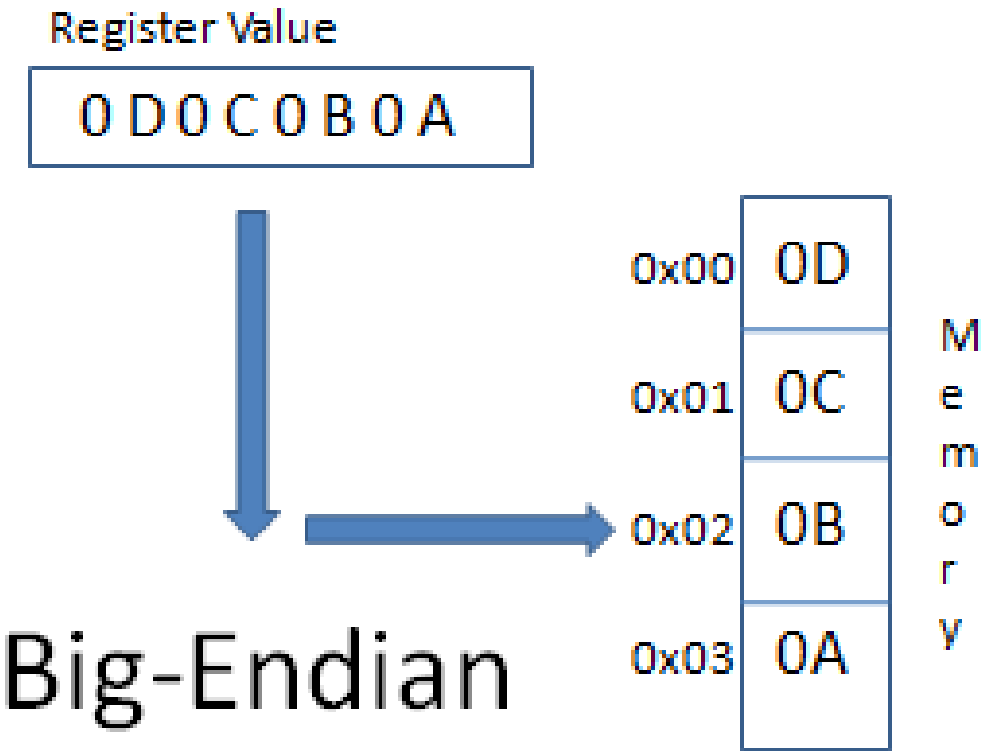
- Device drivers and embedded programs that run on the VP without an OS are called “bare metal” programs.
- Debugging a bare metal program is called “bare metal” debugging.
- If the debugger deals natively with OS constructs such as threads and signals it is “OS-aware.”
- The following examples apply to “bare metal.”

Bare Metal debugging of VPs is a low visibility endeavor. TLM-based Memory Debug can help.

Finding an endianness mismatch

- In any SOC design, it is important that processors, peripherals, and the embedded SW agree on endianness of data items passed between components.
- Sometimes it can be tricky to keep this straight.
- Tracking down an endianness mismatch can be very challenging.

Big Endian


Register Value
0 D 0 C 0 B 0 A
0x00
0D
0x01
0C
0x02
0B
0x03
0A
Memory
Big-Endian

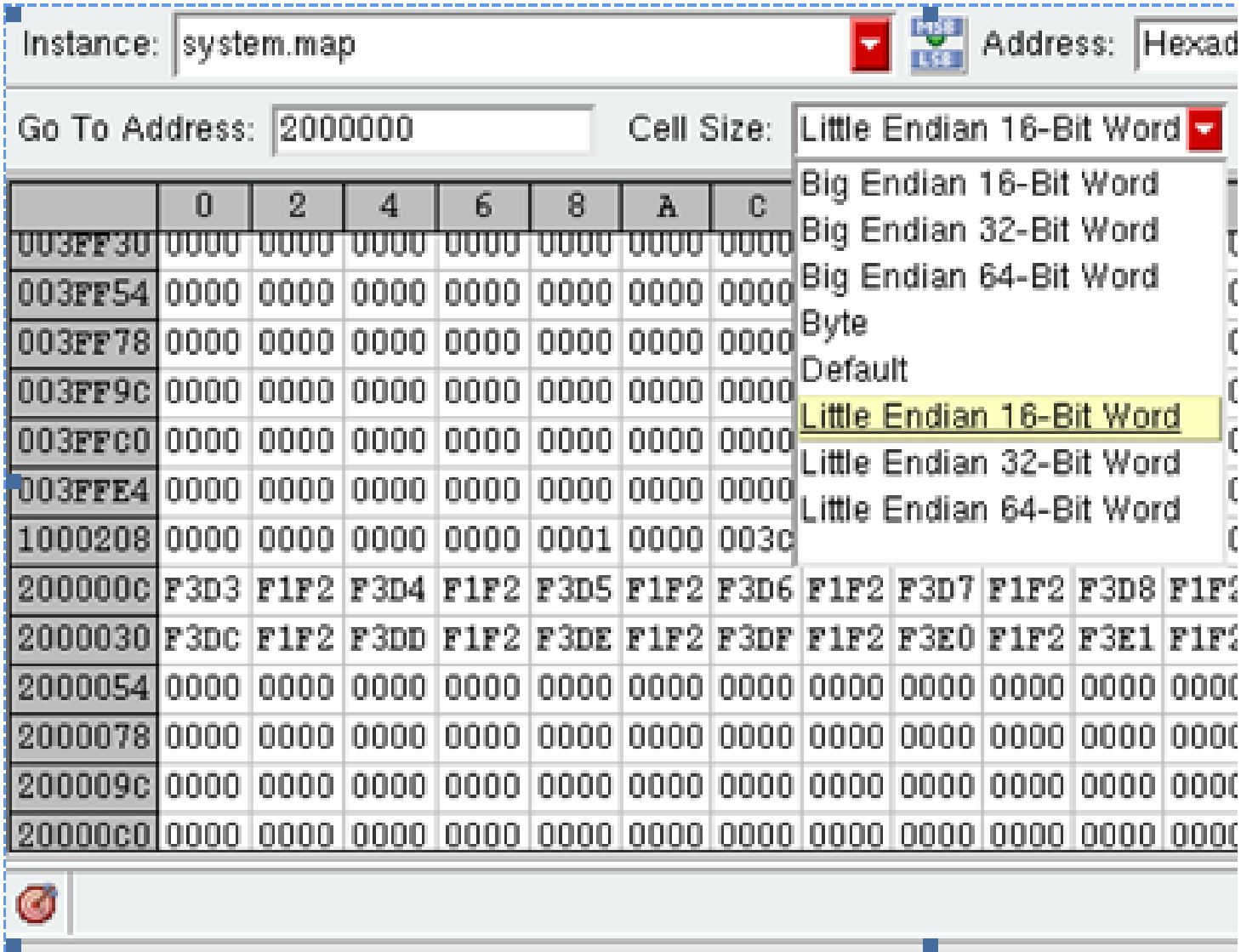
Configuring endianness for ARM A9s

- At least 2 different ways.
- When we generated the A9 processor, host-endianness of Linux was explicitly chosen (little endian).
- TLM standard suggests that for debug, choose endianness of host (LT models need to be fast). In this case both initiator and target share host endianness.

On boot-up, the system crashed

- So early in boot process no traditional embedded software debugging even of assembly language is possible.
- Instead we investigated memory using TLM-based memory debug.
- Running the bare metal design, we noticed memory content register value was in big endian.

Endian display in Memory View



Led to investigation of other ways to set endianness for the ARM A9

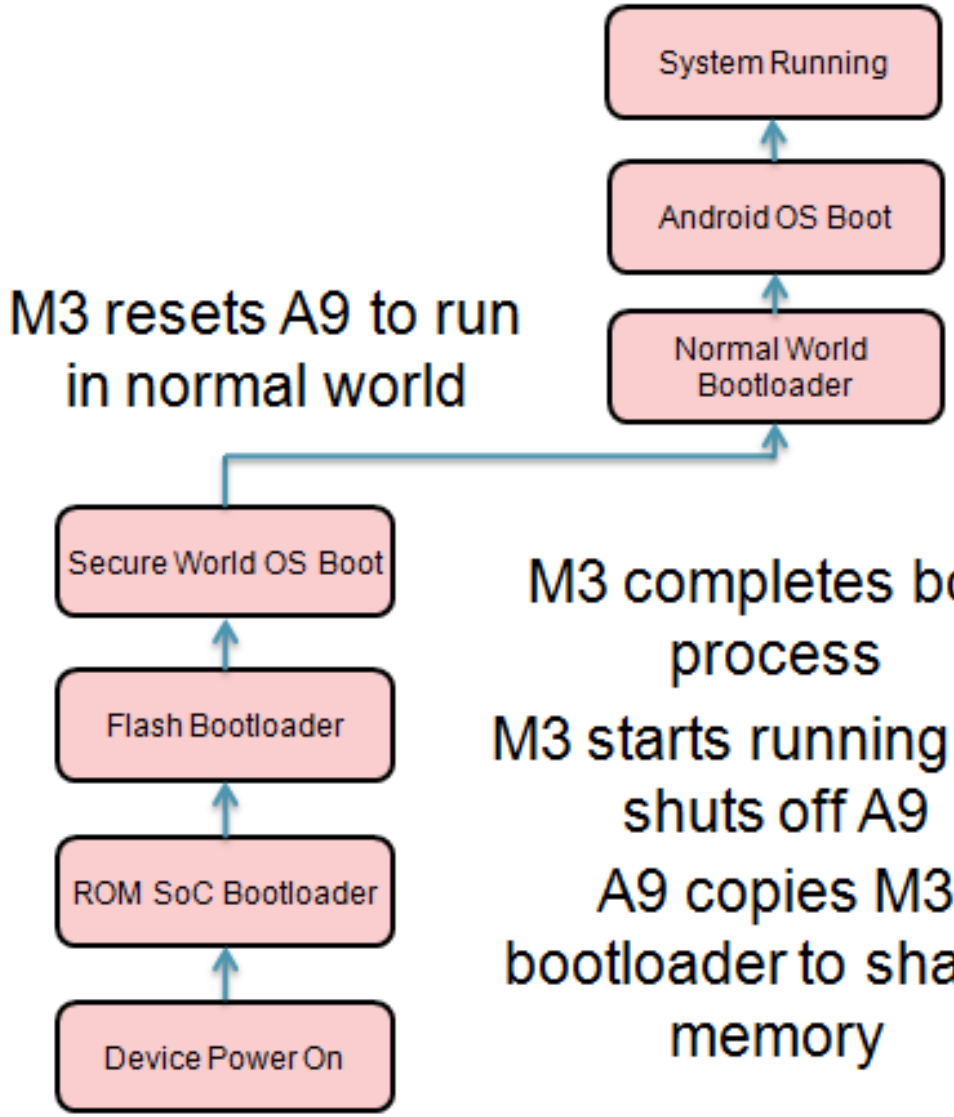
Arm documentation pointed to assembly language code that sets the core to big-endian.

```
MRC p15, 0, r0, c1, c0, 0
ORR r0, r0, #0xf8
MCR p15, 0, r0, c1, c0, 0
```

- This was from code we inherited.
- TLM-based debug gathers the memory values of the RAM and supports displays in different endianness.

Finding problems in dual-processor boot up

- This issue related to the multi-phase boot sequence for dual ARM Cores.
- We were working from a spec from ARM that explains how the bootup works.


System Running
Android OS Boot
Normal World Bootloader
M3 resets A9 to run in normal world
Secure World OS Boot
Flash Bootloader
ROM SoC Bootloader
Device Power On
M3 completes boot process
M3 starts running and shuts off A9
A9 copies M3 bootloader to shared memory

If the boot fails, the system freezes

- Early access to memory contents invaluable in this case.
- We examined the memory in the phase where A9 boots and copies the rest of the bootup code into memory so the M3 can read it.
- Only part of the bootup code made it.

Possible Causes of the Problem

- Problem in synchronization of accesses to the RAM being used by the A9 and M3.
- A problem with the multi-processor communication hardware block.
- A problem with the interrupt configuration.
- Without TLM Memory debug, you would have to instrument the memory to dump its values and change the model.

An ARM General Interrupt Controller programming error.

- We used values in the address registers to identify the location of the bootup code in RAM.
- Even with this, we had to go down several fruitless paths.
- By tracing interrupt signals, we found a missing signal which led to discovery of an error in GIC (Generic Interrupt Controller) programming.
- INTS[0] corresponds to GIC Interrupt ID 32, but the software wrote to a different bit in the register.

Redirecting Android kernel messages to memory.

- Possible using TLM Memory Debug.
- Very early in the Android boot before the TTY is initialized, “printk” messages sit in the ring buffer in memory.
- Ring buffer is a static array:

```
static char __log_buf[__LOG_BUF_LEN];
static char *log_buf = __log_buf;
```

Conclusion

- Debugging a SystemC-based VP can be a difficult challenge involving problems caused by the hardware models, the low-level software, the application software, or all three.
- TLM standard provides a powerful methodology for creating, via the transport_dbg interface, TLM-based debug tools such as Memory Debug.