Simulating Embedded Hardware for Software Development

Class 410

Jakob Engblom, PhD
Virtutech
jakob@virtutech.com
Scope & Context of This Talk

- Supporting embedded software development
- Using simulation of computer systems
  - On computers (obviously)
- It is not about
  - Simulating computer hardware to design it (primarily)
  - Chip design
  - Mechanical system simulation and design
Why?
Because Hardware Is...

- Not yet available
- Flaky prototype stage
- Not available anymore

Photo: Freescale

Photo: Computer History Museum
Because Hardware Is...

- Inconvenient
- Dangerous
- Inaccessible

Photo: www.mil.se, Bromma Conquip

Photo: ESA
Because Hardware Is...

- Impractical in scale
- Limited
- Inflexible
Simulation Advantages

- “Just software”
- Availability
  - Easy to copy & distribute
  - Requires no special lab
  - Good for global reach
  - Available before hardware is completed
- Flexibility
  - Engineering computer can be “any” system
  - Fewer fixed lab setups
- Inspectability
  - Any variable or property can be observed, even if hidden in the real world
- Controllability
  - Any variable or property can be changed
  - Controlled experiments, not real-world random
- Configurability
  - Easy to change configuration and create new configurations
- Easy to vary parameters
Example: Early Hardware

- Hardware design and production
- Hardware/Software Integration and Test
- Hardware-dependent software development
- Simulator development
- Hardware/Software Integration and Test
- Hardware-dependent software development

First successful power-on and boot

Reduced project time-to-ship using simulated hardware
Example: Late Hardware

Yesterday: 32-bit PC
Host hardware
Host operating system: Windows
Simics: Sim for Win/x86
Target OS: PPC 750fx Card
Applications

Today: 64-bit PC
Host hardware
Host operating system: Linux
Simics: Sim for Linux/x86-64
Target OS: PPC 750fx Card
Applications

Future: X Hardware
Host hardware
Host operating system: Y OS
Simics: Sim for X/Y
Target OS: PPC 750fx Card
Applications

...but the simulated target hardware stays the same
Allowing continuous use and evolution of the same software stack

The host machines available change over time...
Example: Loading Flash Convenient

- Much shorter turn-around time for changing target software setup
- No risk of “bricking” a target
Technology
Embedded Computer System

Software stack:
- Applications
- Middleware, libraries
- Operating system
- BootROM, drivers, HAL

Networks

Controlled Environment

Human user interface
Simulating the board(s) and running their software is the focus of this presentation.
User Interface Simulation

Short overview
User Interface Simulation

- A category of tools of its own
- Part of many other simulation tools

- Many different levels:
  - Virtual screen & mouse
  - Clickable touch screens
  - Clickable button panels
  - Graphics displays
  - Serial console
  - Real panels connected to PC

- Target software representation:
  - Simulated by scripts
  - Special code for special API
  - Actual target code in some form of other simulator
Environment Simulation

Short overview
Environment Simulation

- Large field for powerful commercial tools
  - MatLab/Simulink
  - LabView/Matrixx
  - MSC software
  - ... and many more ...
- In-house models common
- Everybody is using it, CAD has been doing mechanical simulations for 50 years
- Commonly used for control algorithm development
- Key part of the model-driven architecture/model-driven design paradigm
- Interface to board simulation:
  - AD, DA converters
  - Digital inputs & outputs
Network Simulation

In a little more depth
Network Simulation Variants

- Connections between abstracted nodes, to study communication patterns
  - Contains models of node behavior, no actual code
- “Rest of network simulation” to provide the environment for a single node
  - Generates “real” traffic
  - Implements actual protocols
  - Bidirectional reactive traffic
- Dumb traffic generation
  - Generate traffic from rules
  - Unidirectional
- Virtual packet-level network links between simulated nodes
  - No protocol understanding
  - Nodes run network stacks
- Connect physical and simulated nodes
  - Virtual machines visible on physical network
- Network types:
  - Ethernet, AFDX, CAN, LIN, FlexRay, MOST, PCIe, I2C, LonWorks, ARINC 429, MIL-STD-1553, Serial, RapidIO, VME, SpaceWire, USB, FireWire, ...
Network Simulation Levels

- **High-level application actions**: Load software, configure node, restart
- **Application protocol**: FTP, DHCP, SS7, CANopen
- **Network protocol**: TCP/IP etc.
- **Packet transmission**: Ethernet packets with MAC address, CAN packets, serial characters, VME data read/write
- **Bit stream**: Clocked zeros and ones, CAN with contention, Ethernet with CSMA model
- **Physical signaling**: Analog signals, bit errors, radio modeling

[Diagram showing the levels of network simulation]

Hardware/software boundary
Network Simulation Levels

- **High-level application actions**: Load software, configure node, restart
- **Application protocol**: FTP, DHCP, SS7, CANopen
- **Network protocol**: TCP/IP etc.
- **Packet transmission**: Ethernet packets with MAC address, CAN packets, serial characters, VME data read/write
- **Bit stream**: Clocked zeros and ones, CAN with contention, Ethernet with CSMA model
- **Physical signaling**: Analog signals, bit errors, radio modeling

Levels of main interest for embedded software work on simulators
Example: Rest-of-Network

Rest-of-network simulation solution for telecom networks.
UMTS, GSM, POTS. All protocols and all node types.
400 employees to develop and maintain.

Source: Nethawk Marketing Materials
Simulating a Board

Something you can actually run code on

Part I: Technology background
Cardinal Rule of Simulation

Scope of modeled system

Universe

Galaxy

Atom

Quarks

Planets

Galaxies

Reasonable to simulate: scope proportional to abstraction

Units of the simulation
Computer Simulation Use Cases

- **System-on-Chip Design**
  - Hardware designer needs
  - Architecture exploration
  - Sizing, performance, optimization of hardware

- **Fidelity** to target is primary driver for models
  - Bus structure
  - Bus protocols and arbitration
  - Timing
  - Bandwidth
  - Latency
  - Cycles

- All components are equals

- **Software Development**
  - Execute large workloads
  - Debug code
  - Get the system to work
  - (Optimize the software)

- **Speed** of execution is the primary driver for model
  - Abstract as far as possible
  - Approximate timing

- The system processor or processors key driver
  - Clear difference between processors and other devices

- Our focus today
### Simulation/Virtualization Techniques

<table>
<thead>
<tr>
<th>Scope of exec</th>
<th>Desktop/Server Virtualization</th>
<th>Para-virtualization</th>
<th>Emulation</th>
<th>ISS</th>
<th>API-Level Simulation</th>
<th>Full-system simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU A on A</td>
<td>System</td>
<td>System</td>
<td>Application</td>
<td>Processor</td>
<td>Application</td>
<td>System</td>
</tr>
<tr>
<td>CPU B on A</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Run full OS</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>OS A on A</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>OS B on A</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>N/A</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Run unmodified software stack</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Custom devices &amp; drivers</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Deterministic</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Complexity</td>
<td>Medium</td>
<td>Low</td>
<td>High</td>
<td>Medium</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Example</td>
<td><em>VmWare, LPAR, kvm</em></td>
<td><em>Xen</em></td>
<td><em>Rosetta</em></td>
<td><em>gdb ISS</em></td>
<td><em>VxSim</em></td>
<td><em>Simics</em></td>
</tr>
</tbody>
</table>

For embedded software work, FSS and API-level sim are really the best options.
**Full System vs API-Level**

- **“Java is java”: simulate using some common middleware/API set**

- **Classic OS API simulation**: compile to PC host, use special implementation of OS API

- **Low-level API-level simulation**: special device drivers and HAL for PC simulation, compile to host including the kernel

- **“Full-system simulation”:** simulate the hardware/software interface, unmodified software stack from target compilation
### Nota Bene: Workload Sizes

<table>
<thead>
<tr>
<th>Workload</th>
<th>Size in number of instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Booting Linux 2.4 on a simple StrongARM machine</td>
<td>50 M</td>
</tr>
<tr>
<td>Booting a real-time operating system on a PPC440GP</td>
<td>100 M</td>
</tr>
<tr>
<td>Booting Linux 2.6 on a single-core MPC8548 SoC</td>
<td>1000 M</td>
</tr>
<tr>
<td>Booting Linux 2.6 on a dual-core MPC8641D SoC</td>
<td>3600 M</td>
</tr>
<tr>
<td>Running 10 million Dhrystone iterations on an UltraSPARC core</td>
<td>4000 M</td>
</tr>
<tr>
<td>One second in a 10-processor 1GHz rack system</td>
<td>10000 M</td>
</tr>
</tbody>
</table>
Building a Board Model
Building a Model

- Essentially, what you want to do is to replace development hardware with a simulator.
- The road there is building the system model.
Starting the Modeling Process

- Board documentation
- Board block diagram
  - Chips
  - Connections
  - Memory map
- Chip documentation
- Chip block diagram
  - Devices
  - Connections
  - Memory map
Board Documentation

- Common names:
  - "Technical Reference"
  - "User’s Guide"
  - "Programmer’s Manual"

- The document a programmer needs to write code for the board

- Typically contains a block diagram
Board Block Diagram

- Good initial overview
- Shows all the chips and their connections
  - Double-check with other documentation, though
- Determine what you need to model, and their relative priority
  - Usually, you do not actually need all the pieces
Chip Documentation

- With modern SoCs, each chip is like a small board in its own right
- Internal block diagrams
Components of an SoC

- Processor cores
- On-chip interconnects
- "System utilities"
  - Interrupts
  - Timers
  - DMA controllers
- Accelerators
- Bus controllers
- IO device
# Units of Simulation

<table>
<thead>
<tr>
<th>Processor Cores</th>
<th>Devices</th>
</tr>
</thead>
<tbody>
<tr>
<td>● The CPUs running code</td>
<td>● Anything that the system contains that does things and that is not a user-programmable CPU</td>
</tr>
<tr>
<td>● Special case to gain performance, simulated using ISS, JIT, API, etc. – buy or borrow!</td>
<td>● Examples: Timers, interrupt controllers, ADC, DAC, network interfaces, I²C controllers, serial ports, LEDs, displays, media accelerators, pattern matches, table lookup engines, memory controllers, ...</td>
</tr>
<tr>
<td>● Comparatively limited in variants, compared to devices</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memories</th>
<th>Interconnects</th>
</tr>
</thead>
<tbody>
<tr>
<td>● RAM, ROM, FLASH, EEPROM, ...</td>
<td>● Connecting devices, chips, boards, cabinets, systems together</td>
</tr>
<tr>
<td>● Store code and data</td>
<td></td>
</tr>
<tr>
<td>● Usually special simulation case for performance reasons, closely integrated with processor core simulators</td>
<td>● I²C, Serial, Ethernet, PCI, PCIe, RapidIO, ATM, CAN, FireWire, USB, MIL-STD-1553, MII, VME, HyperTransport, memory bus, ...</td>
</tr>
</tbody>
</table>
## Units of Simulation: In Practice

<table>
<thead>
<tr>
<th>Processor Cores</th>
<th>Devices</th>
</tr>
</thead>
<tbody>
<tr>
<td>● The CPUs running code</td>
<td>● Anything that the system contains that does things and is not a user-programmable CPU</td>
</tr>
<tr>
<td>● Special cases (simulated using CPU)</td>
<td>● Example: ADC, DAC, ports, matches, controllers, ...</td>
</tr>
<tr>
<td>● Complex: reuse existing simulators and the efforts of experts</td>
<td>● CPU and other devices</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memories</th>
<th>Interconnects</th>
</tr>
</thead>
<tbody>
<tr>
<td>● RAM, ROM, FLASH, EEPROM, ...</td>
<td>● Connecting devices, chips, boards, cabinets, systems</td>
</tr>
<tr>
<td>● Store code and data</td>
<td>● I²C, CAN, CANopen, MIL-STD-1553, MII, VME, HyperTransport, memory bus, ...</td>
</tr>
<tr>
<td>● For most purposes, very generic and reusable. You should not need to model this.</td>
<td>● Comparatively few and standardized, often possible to reuse existing simulators.</td>
</tr>
</tbody>
</table>
Memory Map

- In simulation for the benefit of software, we work from the processor outwards
- The most important information initially is the basic memory map of the system
  - This is what the software and drivers see
- Static or flexible
  - Depends on chips in use, interconnects, etc.
  - Simplest case, all things are in fixed and documented positions
Simple Memory Map

- Microcontrollers tend to have fixed memory maps
- Some flexibility:
  - External memory amounts
  - Related product variants with different amounts of memory
  - Related product variants with slightly different peripheral devices

Note on terminology: “device” in this presentation is a single functional peripheral device, typically part of a microcontroller or SoC. “Device” is also used as the term for variants of microcontrollers.

Source: Texas Instruments MSP430x1xxx Family User’s Guide
High-end devices can be very complex
- Different memory size & types
- Configurable mappings
- Negotiated mappings for PCI
- Open to custom boards, multiprocessor combos, etc.

Note that for any particular board + OS combination, it is mostly fixed
- Model does not need to be as flexible as the real world

Example Decomposition
The computer systems connect to the environment using devices that expose a programming interface to the processors and connect to the environment simulation at the other end.
Device Modeling Methodology

How to create a device model
Follow the Hardware

- Structure your simulation like the hardware
  - With appropriate abstraction where possible
- Most blocks in block diagrams should have a representative in the simulation
  - Model devices, connections, memories, processors
Follow the Software

- Only model the devices used by the software
  - Rare to find all devices on a complex chip used
  - Assumes some idea of the software stack being used
    - If nothing else, follow a deployment roadmap

- Only model the modes used by the software
  - Polled vs interrupt-driven serial ports
  - Transparent vs non-transparent PCIe modes

- Put in warnings for anything not implemented!
  - You can always come back later and add functions
Follow the Boot Order

- Especially important for complex systems
  - Multiple boards, network boots, multiprocessors, ...

- Implement system component models in the order the system boot uses them
  - Start with the core board/processor/devices
  - Get the initial boot code to run
  - Build outwards as the boot progresses
Reuse and Adapt Existing Models

- The best way to get a model is to use something that already exists
  - Check the device library of your modeling system!
  - Vendors often reuse components across SoCs
- Exact same device (or chip)
- Similar device (or chip)
  - Other generation of same family
  - Superset/subset of devices in an SoC
Skip Unnecessary Details

- Details cost development time and execution speed
- Implement the *what* and not the *how*
  - Remember our use case: simulation for software dev, not for hardware design and validation
  - Improves model reusability
- Do work in largest possible units
  - Send entire Ethernet packets
  - Move a DMA block in a single step
  - Also called "transaction-level modeling"
- Simplify system timing
  - Bus contention, bandwidth limits, caches, cache coherency, and other functionally invisible effects can (usually) be ignored
Abstraction and Optimization

Getting fast, really fast
Full-System Simulation

- Detail level determines speed
  - The more detail, the slower the simulation
- Abstraction: timing precision, implementation details
- Functionality must always be correct!

<table>
<thead>
<tr>
<th>Simulation detail level</th>
<th>Typical slowdown</th>
<th>Approximate speed in MIPS</th>
<th>Time to simulate one real-world minute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gate-level simulation</td>
<td>1000000</td>
<td>0.002</td>
<td>2 years</td>
</tr>
<tr>
<td>Computer architecture</td>
<td>10000</td>
<td>0.2</td>
<td>7 days</td>
</tr>
<tr>
<td>Cycle-approximate simulation</td>
<td>500</td>
<td>4</td>
<td>8 hours</td>
</tr>
<tr>
<td>Fast full-system simulation</td>
<td>5</td>
<td>400</td>
<td>5 minutes</td>
</tr>
</tbody>
</table>
Skip Unnecessary Details

- "Know when to bluff"
  - You are in a poker game against the software 😊

- It is an art to implement just enough to fool the software, but not more
Endianness – Has to be Modeled

- Correct endianness
- Incorrect endianness
Temporal Decoupling

- Context-switch overhead is a killer!
- Do not call all components each cycle/step
  - *(which is the correct solution for HW design)*
  - Give each an uninterrupted time slice
- Normally imperceptible by software

System being simulated

![Simulation execution time for four steps](chart)

- Time saved from not switching contexts as often
Temporal Decoupling

- Experimental data
  - 4 virtual PPC440 boards
  - Booting Linux
  - Aggregate MIPS
  - Execution quanta of 1, 10, 100, ... 1M cycles

- Notable points:
  - 10x performance increase from 10 to 1000 quantum
  - +30% from 1000 to 1M quantum

- Quantum of 1000-10000 considered normal
Memory Map: Full Bus Hierarchy

- Core0
  - L1 cache
- Core1
  - L1 cache
- Core2
  - L1 cache
- Coherency module
- Shared L2 cache
- Fast interconnect for high-bandwidth devices
- Bridge
- Slower interconnect for other devices
- DDR MC
- DDR SDRAM
- DDR MC
- DDR SDRAM
- Flash MC
- Flash memory

- Ethernet
- Accelerator
- RapidIO
- Timer
- UART
- I2C
Memory Map: Streamlined

This is an extreme case of transaction-oriented modeling of memory buses.
Bus Hierarchy or Streamlined?

- Depends on the application and use case
  - For large-scale software development, use streamlined (provides speed and simplicity)
  - For tight hardware/software performance analysis and hardware validation, full bus hierarchy
    - Mostly the domain of SoC designers, not embedded SW

- Simulation runs faster with streamlined model
  - Fewer active parts
  - Fewer intermediaries in each memory operation
**Dummy Devices**

- Many devices lack interesting behavior
  - From the perspective of the software for a particular system
    - Only affect low-level system timing
    - Not used by current software setup
    - No interesting effects from using them
  - Replace with dummies that do nothing
    - But do not give “access out of memory” errors

- Examples: memory timing setup, performance counters, error detection registers, ...
  - Note that you can add them later if effects are needed
Stubs

- Not all parts of a system need to be modeled
- Replace by **stubs**
  - Find an appropriate (narrow) interface to cut at
  - Replace complete model with its behavior
For simulation of rack management, stub out the DSPs on the line cards.

For testing control-plane algorithms, stub out the entire line card.
Methodology Flowchart
Methodology

1. Collect documents
2. Map out system

- Existing devices
  - Reuse
- Similar devices
  - Adapt
- Dummy devices
  - Create dummy
- New devices
  - Create basic register map

- Setup machine model
- Test with target software
- Implement missing features in devices and add missing devices
- Fully functioning machine model
Getting to Code

How does this look, actually?
Framework Rules

- You need to learn your framework
  - What is the API for devices?
  - How do you setup the memory map of a system?
  - How can you add a new device to a system?
Memory Map Setup

- **In device code**
  - The device knows its own set of addresses
  - Looks at all accesses to determine which to handle

- **Makes devices hard to move between systems**

- **Outside device**
  - Some memory map mechanism
  - Declare start & length
  - Give device (preferably) a zero-based address

- **Best for reuse and multiple instances of device models**
Memory Map Outside

Different address from the processor, but the same offset into the device from the memory map.
Handling a Memory Access

- C/C++/SystemC: “the big switch”

```c
if(op is read) {
    switch(addr) {
        case 0x00:
            // Read at 0x00: do sth
            break;
        case 0x04:
            ...
        default:
            error “unknown offset”
            break;
    }
} else { // op is write
    switch(addr) {
        case 0x00:
            // Write at 0x00: do sth
            ...
        }
}
```

- Domain-specific tools:

```c
bank {
    register A size 4 @ 0x00 {
        write(value) {
            // do write
        }
        read ()->(value) {
            // do read
        }
    }
    register B size 4 @ 0x04 {
        ...
    }
}
```

- Or graphical views
- Or table views
Solution Examples
CC SimTech

- For vehicle systems
  - Control & User Interface
  - Handful of compute nodes
- API-level simulation
  - Special middleware API hides details of RTOS
  - Compiles all program code for the host
  - No target timing
- Connects to physical CAN and real-world control panels

Pictures: CC-Systems, www.cc-systems.se
Google Android Emulator

- Full-system simulator
  - Qemu-based, ARM9 ISS
  - Single-processor
  - Touch screen & keyboard
- Not a model of a real board, rather an idealized mobile phone
  - Simpler device programming interfaces
  - Faked mobile connection
- Special BSP for emulator
Virtutech Simics Telecom Rack

- Full-system simulator
  - PowerPC & TI DSPs
  - 2-70 boards
  - 20-500 processors
  - Rack backplane
  - Multiple board types
- Some boards stubbed
- Some boards simplified with partial stubs
- User interface
  - Serial consoles
  - Ethernet to physical world
    - O&M, traffic, SCTP, ...
Where Next?
How can you start simulating?

- Use C/C++ and build system simulations from scratch
- You might already have something to start with
  - API-level simulator for your RTOS
  - ISS built into most embedded tool chains
- Commercial mechanical, network, user interface simulation tools
  - Plenty on the ESC show floor
- Heavy-duty commercial computer simulation tools
  - Virtutech, VaST, Synopsys, CoWare, ARM, ...
- Open-source system simulators
  - Qemu, Bochs, SystemC TLM2, ...
A New Category of Tools?
Innovating on Tools IS Allowed

nano? REAL PROGRAMMERS USE emacs

HEY, REAL PROGRAMMERS USE vim.

WELL, REAL PROGRAMMERS USE ed.

NO, REAL PROGRAMMERS USE cat.

REAL PROGRAMMERS USE A MAGNETIZED NEEDLE AND A STEADY HAND.

EXCUSE ME, BUT REAL PROGRAMMERS USE BUTTERFLIES.

THEY OPEN THEIR HANDS AND LET THE DELICATE WINGS FLAP ONCE.

THE DISTURBANCE RIPPLES OUTWARD, CHANGING THE FLOW OF THE EDDY CURRENTS IN THE UPPER ATMOSPHERE.

WHICH ACT AS LENSES THAT DEFLECT INCOMING COSMIC RAYS, FOCUSING THEM TO STRIKE THE DRIVE PLATTER AND FLIP THE DESIRED BIT.

NICE. "OF COURSE, THERE'S AN EMACS COMMAND TO DO THAT."

OH YEAH! GOOD OL' C-x M-c M-butterfly...

DAMMIT, EMACS.
This is nothing new, really

- Ad in Embedded Systems Programming, Issue 1, Volume 1, January 1988

- The reason to simulate has stayed the same

- The power and scope of simulation and target systems has increased tremendously!
  - Single 1MHz 8-bit...
  - ...to 100 1GHz 32-bit
  - In 20 years
This seriously is nothing new

The diagnosis of mistakes in programmes on the EDSAC

BY S. GILL
Mathematical Laboratory
University of Cambridge

(R.S.—Received 13 December 1950)

Their solution basically used a system simulator to run target instructions one at a time under debugger control. Took some 32 target instructions to implement!
Misconceptions

By far, the biggest disadvantage of a simulator is that it simulates only the processor. Embedded systems frequently contain one or more important peripherals. Interaction with these devices can sometimes be imitated with simulator scripts or other workarounds, but such workarounds are often more trouble to create than the simulation is worth. So you probably won’t do too much with the simulator once the actual embedded hardware is available.
Questions?
Thank You!

Please remember to fill in the course evaluation forms completely!
Spares
Transaction-Level Modeling (TLM)

- Concept from chip design/EDA field
- Gain efficiency by abstracting from bus details
  - Necessary for anything resembling fast simulation

Traditional low-level hardware modeling

Transaction-oriented modeling