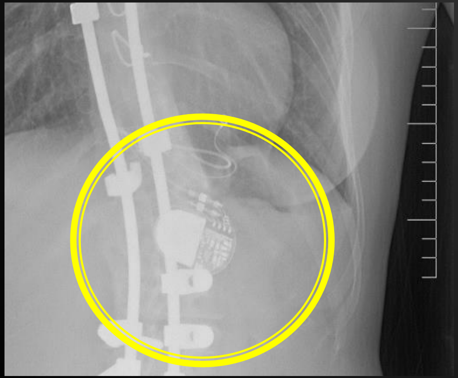
Karl Koscher, Tadayoshi Kohno, David Molnar

SURROGATES: Enabling Near-Real-Time Dynamic Analyses of Embedded Systems

Embedded Systems Security



Embedded Systems Security

- Many potential reasons for vulnerabilities
  - Time-to-market pressures
  - Limited patching abilities
  - Historic lack of adversarial pressure
  - Limited visibility of the whole system
  - Limited tools for security analysis

Embedded Systems Security

- Many potential reasons for vulnerabilities
  - Time-to-market pressures
  - Limited patching abilities
  - Historic lack of adversarial pressure
  - Limited visibility of the whole system
  - Limited tools for security analysis

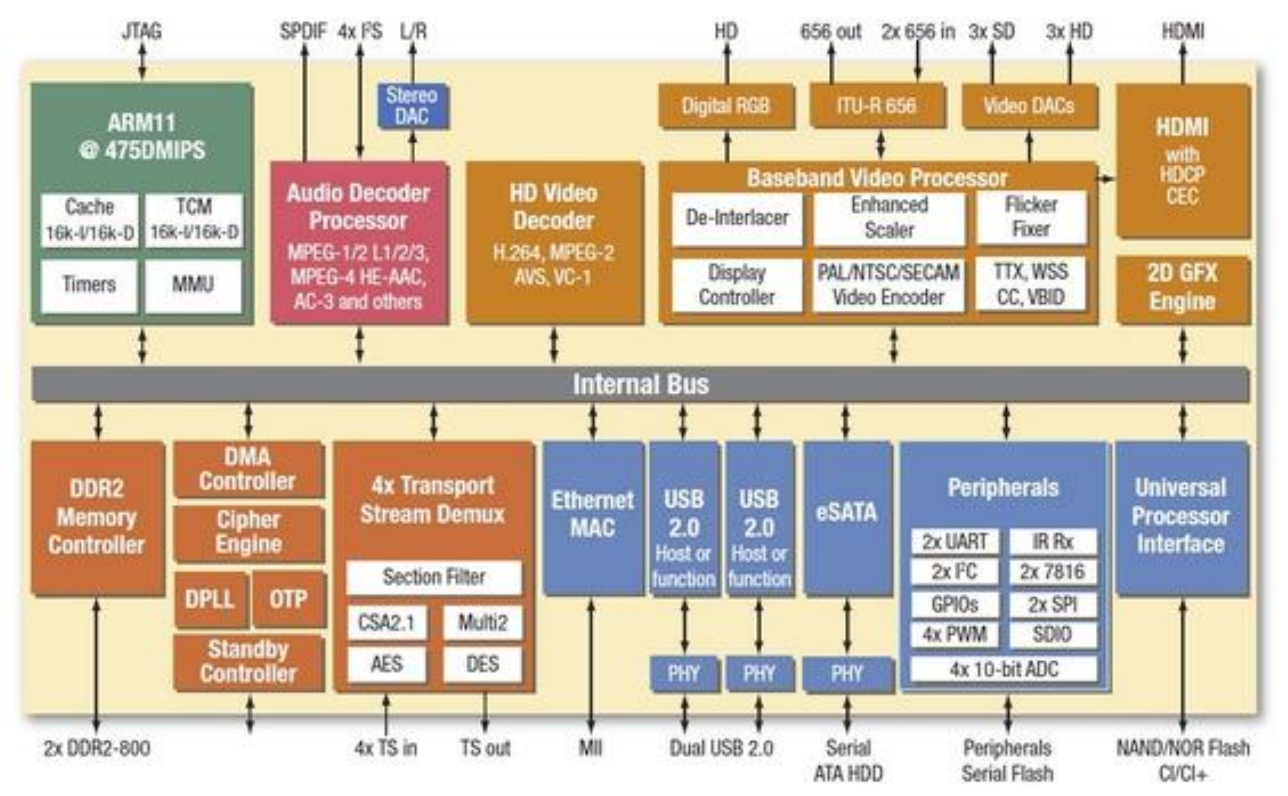
Security Analysis Tools

- For "traditional" software, we have many
  - Memory checkers (e.g., valgrind)
  - Fuzzers (e.g., Peach, SAGE, etc.)
- For embedded systems, not so much

Challenges for Embedded Systems

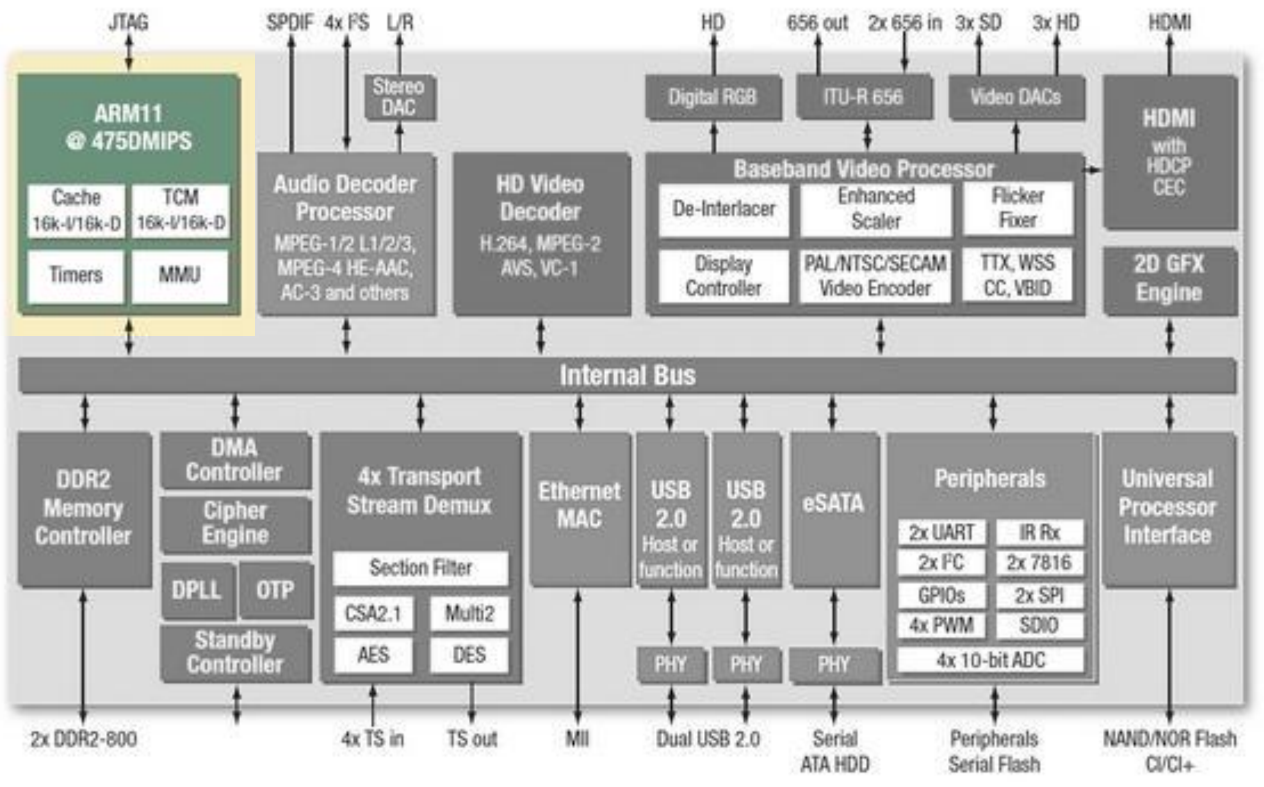
- Hard to add instrumentation to real systems
  - Limited resources
  - Lack of standard abstractions (e.g., OS APIs)
- Hard to emulate
  - Heterogeneity
  - Systems are tightly coupled with their environment

Challenges for Embedded Systems



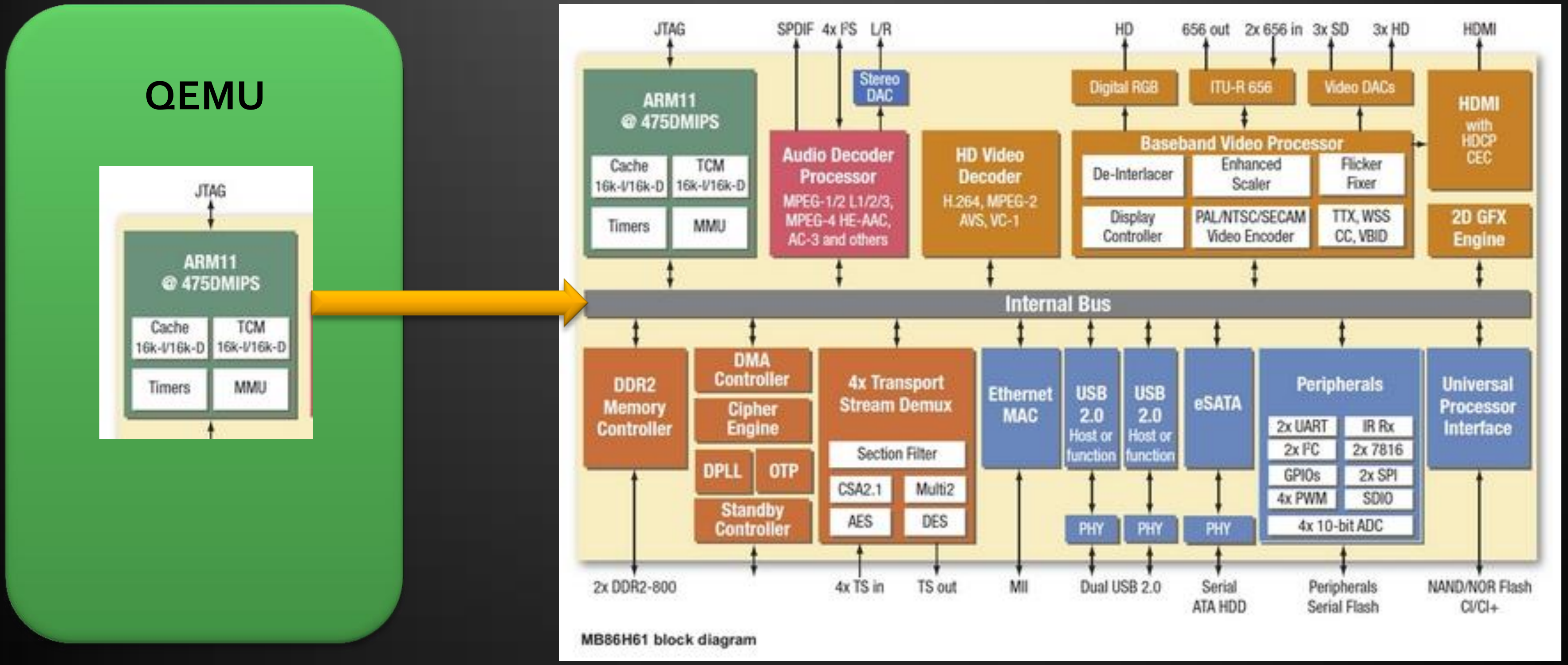
MB86H61 block diagram

Our Approach



MB86H61 block diagram

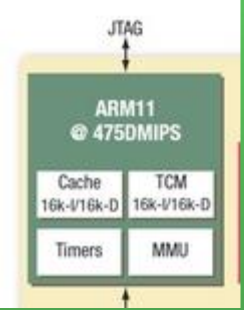
Our Approach

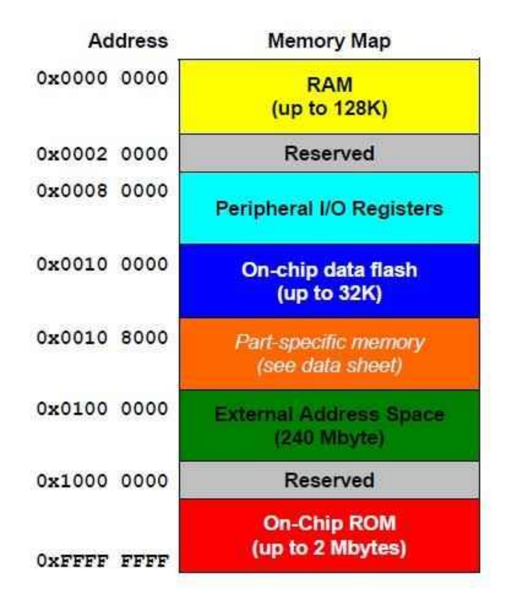


MB86H61 block diagram

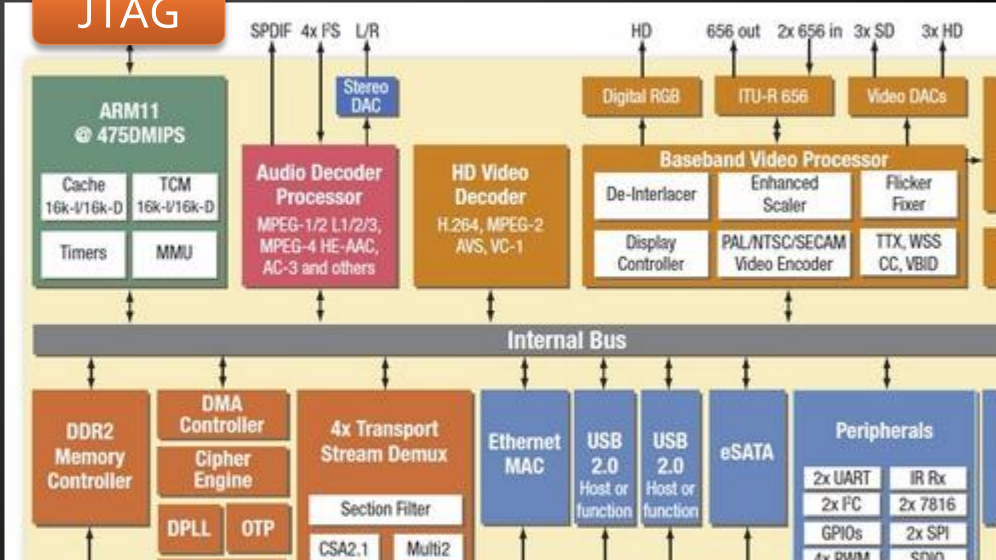
Our Approach

QEMU





JTAG



Our Approach

- This is not easy
  - We are not the first to propose this approach
  - Prior work is limited by I/O performance
  - A substantial amount of effort is needed to make this work in practice

Contributions

- We demonstrate the feasibility of near-real-time whole system emulation
- We discuss the engineering challenges and tradeoffs
- We identify and overcome new challenges when running a whole system under this type of emulation

Towards Real-Time Emulation

- Our design decisions are driven by our failures
- First approach: Connect QEMU to OpenOCD and issue memory reads/writes over JTAG
- This is *extremely* slow

Towards Real-Time Emulation

- Most JTAG debuggers implement memory operations by modifying the CPU state
  - Read CPU state (i.e., registers)
  - Update registers for operation (address, data)
  - Inject instruction to perform the operation
  - Single-step the CPU
  - Read out the result from a register
  - Restore CPU state

Towards Real-Time Emulation

- To overcome this performance bottleneck, a dedicated debug channel is provided
  - Load small stub into cache/RAM
  - Begin executing the stub
  - Send commands/data over the dedicated debug channel

Towards Real-Time Emulation

- Next approach: A stub that accepts commands over ARM's DDC
  - Stub is 768 bytes (small enough to lock in cache)
- Multiple commands:
  - Single byte/word/dword write
  - Multiple byte/word/dword read/write
  - Set processor flags (e.g., interrupt enable flag)

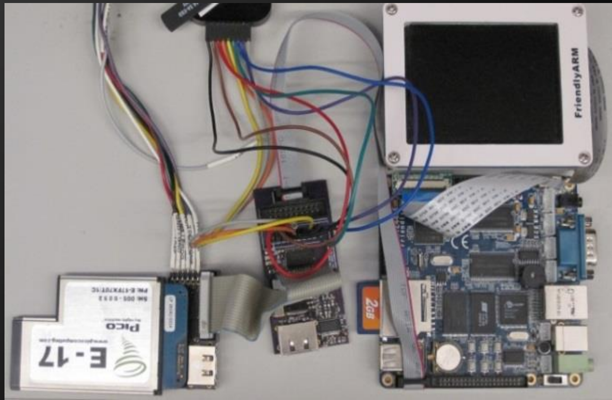
Towards Real-Time Emulation

- Much faster! … but still not fast enough
  - Devices with coprocessors or watchdogs can be very sensitive to timing
- Bottleneck? USB
  - USB 2 polls devices up to once every millisecond
  - => Under the best conditions, 500 ops/second
  - Execution depends on the environment, so this limits total performance

(╯°□°)╯︵ ┻━┻

Enabling Near-Real-Time I/O

- Idea: memory map the target device into the host's address space using an FPGA connected to the PCIe bus
  - No OS overhead – it's just (uncached) memory
  - Can map entire 32-bit address spaces into 64-bit emulator processes



- Great idea… in theory.

Enabling Near-Real-Time I/O

- Problem: PCIe requires that all 64-bit mappings are *prefetchable*
  - Side effect free
  - This is fundamentally incompatible with MMIO
- So just ask for a 32-bit mapping and have everything else move above 4 GB?

Enabling Near-Real-Time I/O

- Problem: Address space below 4 GB is scarce
  - You can't map more than 128 – 256 MB
  - MMIO shouldn't use that much space
- Problem: MMIO often *does* use that much
  - Sparse memory layout for easy address decoding

(╯°□°)╯︵ ┻━┻

Enabling Near-Real-Time I/O

- Memory-map FPGA registers to initiate I/O requests over JTAG
  - e.g., writing an address to the “read address” register will initiate a read over JTAG
  - Read the result out of the “read result” register
  - PCIe stalls can be used to avoid polling for the result…
    - … unless your system is buggy

Enabling Near-Real-Time I/O

- MMIOs are now limited by JTAG speed
  - For a 4 MHz JTAG clock, we can do ~16000 read ops/sec and ~17000 write ops/sec
    - v.s. 5 ops/sec in Avatar
- Now we have several more issues to address

Interrupts

- When an IRQ is raised on the target, the stub disables IRQs and sends a notification to the host
- The FPGA converts this to a host interrupt, which is passed as a signal to qemu

Interrupts

- Emulated IRQ handler runs
  - Acknowledges the IRQ
  - Re-enables interrupts
    - This sends a command to the stub to do so as well
- What if a second interrupt occurs?

Interrupts

- Multiple interrupts can occur on real hardware, so systems are designed for it
  - e.g. an interrupt controller which tracks unacknowledged IRQs

Interrupts

- Some SoCs use plain ARM IRQs/FIQs, and some use vectored IRQs
  - Vectored IRQs are often implemented in ROM as they are unsupported in the ISA
  - This ROM can be emulated as well, so we only have to handle IRQ/FIQ!

DMA

- For performance, RAM is not mirrored on the target
  - This breaks DMA
- No standard DMA interface

DMA

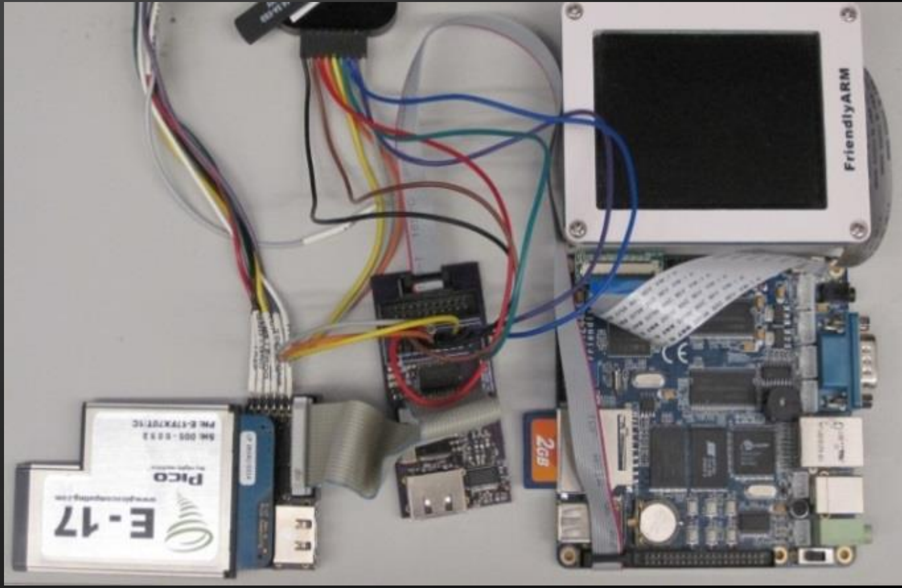
- Two approaches:
  - Add a small amount of logic to qemu to emulate DMA controller(s) of the SoC
  - Exploit the fact that DMA buffers can’t be cached

Other Annoying Details

- Watchdogs should generally be disabled
- Changing the SoC’s clock generator will cause the JTAG adapter to lose sync
- Use Surrogates to find the write that causes it to crash and refuse to pass that write

Implementation & Evaluation

- Pico Computing E17 PCIe card
  - Xilinx Virtex5 FX70T
    - ~14% utilization
  - ~1100 lines of Verilog
    - ~1000 lines of tests
- FriendlyARM 2440 dev board (S3C2440)



DEMO

Future Work

- Full integration with dynamic analyses tools (e.g., S2E)
- Learn approximate models of hardware
  - Can search for bugs in parallel
  - Verify potential bugs against real hardware
- Cheaper hardware (USB 3?) / Open Source

Summary

- We demonstrate the feasibility of near-real-time whole system emulation
- We discuss the engineering challenges and tradeoffs
- We identify and overcome new challenges when running a whole system under this type of emulation

Thank You!