New Tools for STEM, Cyber, and Makers

www.lockelabs.net
Overview

• Motivation and Rationale
  – Execute Java source code directly on hardware
  – Concept inspired in part by modular features of Java 9

• Demo Configuration and Slides
  – Embedded Hardware
  – Host configuration

• Caveats
  – Proof of concept, necessary but not sufficient capabilities
    • Memory access, memory allocation and constraints, process interrupts
  – “Execution environment” is not a JVM

• Workflow and Package Structure

• Embedded Code Examples
  – Java main, memory allocation, a SoC module, memory access, interrupts
  – Memory corruption due to stack or heap overflow is currently prevented. Current heap protection approach is presented.

• Discussion
Motivation and Rationale

• Currently we have large numbers of cyber vulnerabilities and attacks
  – At the same time, we have an increase in the number of people interested in STEM, cyber technology, and makers interested in electronics

• Suggesting that these 3 communities have a similar requirement
  – *How to more easily experiment with the interaction between hardware and software*

• This proof of concept demonstrates that the Java language and pending changes (modularity) have several advantages over traditional embedded programming (C and flavors of Unix)
  – Reduce the attack surface by reducing the number of lines of code executing
  – Stronger typing
  – More consistent memory constraints
  – Per TIOBE, the most popular programming language
  – Reduce the effort of setting up a cross-platform toolchain
Demo Slides

- Demo Java application utilizes timers, interrupts, and General Purpose Input Output to blink LEDs on a fixed frequency
- Embedded Hardware
  - Initially targeting a Beagle Bone Black board with a TI SoC (AM3358B) with an ARM Cortex-A8 MPU
  - SoC includes:
    - UART
    - Dual Mode Timer
    - Interrupt Controller
    - General Purpose Input Output
    - Power, Reset, and Clock Management
    - Programmable Real-Time Unit and Industrial Communication Subsystem (dual 32-bit RISC cores)
    - Enhanced Direct Memory Access
    - 3-port gigabit ethernet switch
    - Pulse Width Modulation Subsystem
    - USB
    - I2C
    - Controller Area Network (CAN)
    - Multichannel Serial Port Interface (McSPI)
- Host configuration
  - 2010 MacBook Pro
  - Version 4.0.2 of ‘screen’ terminal emulator
  - USB to TTL Serial Cable and Drivers to BBB Console Serial UART
Start the runtime, then disable Timers

- Runtime waits for ‘g’ command to run
- Hardcoded to start 2 sec timer. Timer interrupt service routine advances to the next LED
  - ‘int’ in output denotes that INTC generated the ARM exception
  - ‘TIMER’ denotes that the interrupt request was generated by the timer
- Timer Disable (‘d’) command stops the timer
• On previous slide, a single LED was illuminated (LED 2)
• LED clear command (‘l 2 c’) writes 0 to the control bit, turning the LED off
**Interactively Turn a LED On**

- LED set command (‘l 4 s’) writes a 1 to the control bit, turning the LED on.
- Now LED 4 is illuminated.
Simple Console Port Debugger

• Java code also implements a simple command line debugger that interactively provides limited (only on NOP) breakpoints, with the ability to display:
  – registers
  – memory contents at address
  – call stack
  – Memory contents of method variables
• In between this and the previous slide, a warm reset of the board was performed by the ‘pr’ command.
• Instead of entering the ‘g’ command, a breakpoint was set at the NOP instruction at 0x8020a0b0.
• This instruction is in the return path for the sum of a series from N to 1.
• The debugger verified the instruction was a NOP, set the breakpoint and echoed corresponding status ‘Bkpt Set’.
• Enter ‘g’ to continue execution of the runtime.
• The timer fires as before, no difference in execution.
• ‘t d’ disables the timer.
• The breakpoint is in a recursive routine used to test stack memory management and overflows.
• The routine is demonstrated in following slides.
Hit a Breakpoint, But then Continue

• The series summation is invoked by the ‘s t’, or stack test command.
• Here, N = 2.
• The breakpoint at 0x8020a0b0 is hit.
• Execution is continued by entering the ‘g’ command.
• The stack test routine prints the sum, 3.
• Here, the first series summation call received a value of 2, followed by a recursive call with argument of 1 [2 – 1 = 1], where the breakpoint was hit.
• Several lines down on the left, the stack test is repeated with the command ‘s t 00000003’.
• The breakpoint is hit, and all the current stack frames are listed with the ‘f l’ command.
• For each frame, the number of local variables and return address are listed
• A few lines from the top on the right, the register values at the breakpoint are displayed with the ‘dr’ command.
• On the lower half of the right, the local variables of several of the stack frames are displayed via the ‘f v l 0000000i’ command, where “i” is 0, 1, and 2.
Caveats (1 of 2)

• Proof of concept, necessary but not sufficient capabilities to be a compliant JVM
  – Memory access, memory allocation and constraints, process interrupts

• “Execution environment” is not a JVM
  – Only executes native code statically linked with the execution environment
  – Currently a very small subset of the Java language features are implemented
    • Static classes, methods, primitive fields (int and boolean), dynamic allocation of character arrays (no memory reclamation)
    • Utilizing custom annotations to integrate link time information, Java source, and a limited amount of hard coded native assembly source
    • No objects, exceptions, or threads yet
  – Have not created a target Java platform, using project specific packages for defining and testing the current capability of heap and stack errors.
Caveats (2 of 2)

- Java Specs
- Status of JVM Instruction implementation
  - Implemented to varying degrees - 
    `aload`, `arraylength`, `astore`, `bipush`, `caload`, `castore`, `dup`, `getstatic`, `goto`, `iadd`, `iand`, `iconst`, `if_icmpge`, `ifeq`, `iflt`, `iinc`, `iload`, `imul`, `invokestatic`, `ireturn`, `istore`, `isub`, `ldc`, `newarray`, `putstatic`, `return`
  - Currently no implementation includes – any array other than char array, double instructions, float instructions, related to objects such as invokespecial or invokevirtual, switch statements such as lookupswitch or tableswitch, synchronization such as monitorenter or monitorexit
Workflow (1 of 2)

• Cross Platform Toolchain
  – Cross compiled Minix ARM port to BeagleBone Black
  – From this, I am using u-boot and cross-platform GNU binutils – as, ld, objcopy, and objdump

• Target Build Process
  – Netbeans compile of embedded Java and native generation tool (also Java)
  – Run the native generation tool
    • Depends on BCEL and Velocity and generates ARM assembly
    • Running with JDK 8
  – Run as, ld, objcopy, and objdump
    • Ld is generating ELF, objcopy is transforming to binary
    • Objdump generates asm of linked executable to manually lookup addresses for breakpoints
Class files of embedded code

Templates
- Typical asm files
- JVM instruction implementation

BCEL Extract

Velocity Merge

ARM Assembly Files

as, ld, objcopy

Micro SD Adapter
Transfer between Host and Target

- This is a notional diagram of the native code generation process
- The flow ‘Templates, Merge, Assembly File’ is repeated for a variety of files and types.
- Some of these file types are the entry point to the executable, assembly routines that process ARM exceptions, templates for implementation of JVM instructions, etc.
- This native generation approach is very straightforward, no optimization is performed to reduce code size, number of operand stack accesses, etc
Package Structure

- Non-compliant placeholder namespace for Stack and Heap errors (jm.lang)
- User application code (lockelabs.examples)
- Support code, either offline or on target (eg, lockelabs.jm.annotations, lockelabs.jm.cpu, lockelabs.jm.memory.heap)
- Ahead of time generation of native code (lockelabs.jm.nativegen)
- ‘Drivers’ for modules on the SoC (lockelabs.jm.soc)
Embedded Code Examples

- Java main, memory allocation, a SoC module, memory access, interrupts
- Memory limit implementations
- *The following slides illustrate code that runs on the target.*
- *Information from the target build process is included with these slides to illustrate what was required to execute the Java source code.*
Java main - Initialization

```java
char[] helloPrompt = new char[]{' ', 'h', 'e', 'l', 'o', ' ','-',' '};

char[] lsrString = new char[HexString.requiredBufferLength];

char[] counterString = new char[HexString.requiredBufferLength];

char[] eol = new char[]{'n', '\r'};

int uartLineStatusRegister;
WatchDogTimer.disable();
Gen1PurposeInputOutput.initialize();
DualModeTimer.initialize();
InterruptController.initialize();
DualModeTimer.startTimer();

Adder adder = new Adder(3, 4);

int counter = 0;
while (true) {
```

• Line 63 illustrates allocation and initialization of char array. Will look at implementation in following slides.
• Lines 73 – 79 illustrate initializing modules on the TI SoC (eg, GPIO, Timers, Interrupt Controller). Examine Interrupt Controller initialization in following slides.
• Line 84 starts the main loop of the main method. Illustrated on next slide.
Java main - Loop

```java
int counter = 0;
while (true) {
    Instance.println(uart0BaseAddress, helloPrompt);
    Instance.readLine(uart0BaseAddress, fromUart);
    // InterruptController.printInfo();
    GenlPurposeInputOutput.processCommand(fromUart);
    DualModeTimer.processCommand(fromUart);
    PrmDevice.processCommand(fromUart);
    HeapManager.processCommand(fromUart);
    StackManager.processCommand(fromUart);
    // HexString.convert(add.re.add(), counterString);
    // Instance.println(uart0BaseAddress, counterString);
    // Instance.println(uart0BaseAddress, eol);
    counter++;
}
```

• Line 88 reads characters and carriage return entered on Mac keyboard and transmitted over USB-to-TTL to UART (aka console port) on Beagle Bone Black.
• Lines 92 – 100 pass the current ‘command’ to each of the processCommand static methods.
• Code above is from user main.
• At left are the disassembled JVM instructions for the allocation of the char array above.
• On the next slide, the current JVM instruction (newarray) is transformed to native code.
Memory Allocation (2 of 3)

• Code below only runs during native code generation.
• Each JVM instruction has a corresponding template. At left is the template for newarray.
• During code generation, template is merged with relevant data from BCEL and written to current asm file. In this case, the HelloWorld.S file.
• Below, the variable newArrayMethod in the template at left is replaced with the class and static method (NewArray.newCharArray on line 296). This method is shown on the next slide.
• This method runs only on the target and allocates char arrays from the heap and initializes each array with the length of the array. The presence of the length enables index range checking to verify that the bounds of the array are not being exceeded by application code.
SoC Module – Interrupt Controller

• Lines 75 – 77 at left are from the Java main method. One of the methods, InterruptController.initialize, is shown below.

• The code at left calls the DeviceMemory class to write the memory mapped control registers of the InterruptController.
Memory Access (1 of 3)

• In the code at left, each method is decorated with the `WholeMethodGeneration` annotation.
• During the native code generation process, any method with this annotation is generated by reading one template for the entire method instead of iterating through the JVM instructions generated by the Java compiler for the method.
• An example of this process is shown on the next slide.
Shown at line 57, forMethod generates the native assembly code for an embedded Java source code method.

- If the Java method has the WholeMethodGeneration annotation, the template is loaded and merged with the current context (shown in lines 36 – 54).
- The result is written to the ARM assembly file being generated for the current Java class.
Memory Access (3 of 3)

- The template at left is not the whole method for writeInt.
- There is a preMethod template that implements stack overflow checks and initializes the stack frame for the current method.
- The native generation process needs some refactoring. You can see at left that each method is responsible for popping the current frame.
- Template variables are used here as well so that I can easily change register convention when needed.
Interrupts (1 of 4)

During startup of the execution environment, initialize the ARM Vector Base Address Register

On the ARM MPU, when one of the exceptions (Reset, Data Abort, Interrupt from Interrupt Controller, etc) occurs, instruction execution jumps to the corresponding handler (Reset : exc1, Interrupt : exc7) by loading the PC with the address of that handler.
Locke Labs
for the common defense

Interrupts (2 of 4)

• The merged version of this file, which is part of the execution environment, is how Java source code is called when an interrupt occurs.

• The file shown above, exc.S, is the ‘template’ version.
• Line 220 is an example of two Velocity ‘variables’ in this template.
• At native code generation time, $className$ and $interruptJavaHandler$ are replaced with their actual names as supplied by Java source code. Example of this on the next slide, line 157 and method setMethodNames.
• Both of the files above are relevant only during native code generation.
• The file on the left illustrates reading the template exc.S and generating corresponding merged ARM assembly language file exc.S.
Interrupts (4 of 4)

- This file was rearranged to place relevant details on one screen.
- This illustrates the Java source code that processes an interrupt.
- As shown in previous slides, the ArmExceptionHandlers annotation is used during native code generation to insert the name of the Java source methods to call from the execution environment.
- The method interruptHandler is the method invoked when the ARM MPU responds to the Interrupt Controller exception, the exc7 label on a previous slide.
• HeapManager.allocateBytes, shown above, is called in the context of allocating a char array.
• On line 121 above, an OutOfMemoryError exception is thrown when a heap allocation fails.
• The current implementation of new and throwing exceptions is only a proof of concept that illustrates Java detection and halt when stack or heap allocation fails.
• Current implementations of new and throw are not presented.
• Implementation of the heapEndAddress is illustrated on following slides.
Memory Limits – Heap (2 of 3)

- This slide and the following illustrate how the heap end address is currently integrated with Java source.
- LLMAIN shown at left is the entry point for the executable started by u-boot.
- On line 78, the end address of the heap is statically defined.

- The annotation at left is utilized to integrate the assembly language heap end address and the Java source references to the same address.
During class init, the address of the provided label is stored in the annotated int.
- This is implemented with a ‘special case’ template for the putstatic JVM instruction.
- This special case drops the value provided by the class file and instead uses the address of the label provided by the annotation.
Discussion

• Working towards a Kickstarter campaign for June 2017
  – Prep and release all code as open source, considering a BSD 4-clause license
  – Write a book with 2 general topics
    • Step by step instructions for novice to reproduce the capabilities described here
    • Detailed description of design and implementation
  – Identify a stretch goal of a Java implementation of required GNU binutils. Primarily, as, ld, objcopy, objdump. Believe these would have to be released as GPL.

• Questions
Classes Java main doesn’t depend on?

- InterruptHandlers
- HeapManager
- HeapMemory
- NewArray

Native code generation visits all dependencies of Java main and generates corresponding native code. The classes above don’t appear as dependencies and therefore have to be included manually in the list of classes to generate.
How Much ‘Plain’ Assembly?

• LLMAIN.S – defines the method started by u-boot.
• exc.S – defines the exception vector table and initial service routines.
• jvmMain.S – invokes methods to initialize static data of classes and invokes Java main method.
• Files to read and write system control / status registers: flush caches to implement setting breakpoint, address of instruction fault, read and write int values to memory, etc.
Register Convention

Some of the above are notional at this point.
- R0-R3, R4, R5 are the ones listed most frequently in the examples.
- R0-R3 referenced as ocReg1 – ocReg3 in templates (oc = op code)
- R4, R5, and LR are currently pushed to stack between calls.
### Notional Stack Contents

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>zzz</td>
<td>R4 - VP - end of caller stack (Var Ptr)</td>
</tr>
<tr>
<td>aaa</td>
<td></td>
</tr>
<tr>
<td>bbb</td>
<td></td>
</tr>
<tr>
<td>ccc</td>
<td></td>
</tr>
<tr>
<td>ddd</td>
<td></td>
</tr>
<tr>
<td>eee</td>
<td></td>
</tr>
<tr>
<td>fff</td>
<td>Local Variables</td>
</tr>
<tr>
<td>LR</td>
<td></td>
</tr>
<tr>
<td>R5</td>
<td>R5 - FP - caller's local variables (Frame Ptr)</td>
</tr>
<tr>
<td>R4</td>
<td></td>
</tr>
<tr>
<td>ggg</td>
<td></td>
</tr>
<tr>
<td>hhh</td>
<td></td>
</tr>
<tr>
<td>iii</td>
<td></td>
</tr>
<tr>
<td>jjj</td>
<td></td>
</tr>
<tr>
<td>kkk</td>
<td></td>
</tr>
<tr>
<td>lll</td>
<td></td>
</tr>
<tr>
<td>mmm</td>
<td></td>
</tr>
<tr>
<td>nnn</td>
<td>SP - Callee local stack</td>
</tr>
</tbody>
</table>