Why Simulation Sucks

Micheal Frysinger <michael.frysinger@analog.com>
Robin Getz <robin.getz@analog.com>

The long lead in any embedded Linux® product is always the software - software takes the longest to develop, the longest to debug, and the longest to test/qualify. It is for these reasons that instruction set simulators (ISSs) have become an essential part of any embedded software development process. Developers need to get a head start on software development before the hardware exists. However, many ISSs are either slow, difficult to develop, and do not model enough of the system to really be valuable.

It is for these reasons that many embedded Linux developers use some sort of system level simulator, or emulator (qemu, armulator, skyeye, ovp, etc) to do their initial development. This development model has many benefits (including the ability to do development before the target hardware exists), but does not come without its drawbacks.

This paper will explore a few of the most common open source simulators and explain the benefits and pitfalls that exist with each. It will also explore some of the verification that developers are doing to ensure the simulation engines and the final target hardware match as well as what to watch out for when using these valuable tools.

The two main types of simulators are Cycle Accurate Simulators (CAS) and Instruction Set Simulators (ISS). A CAS must ensure that all operations are executed in the proper simulated time, branch misprediction, cache misses, fetches, pipeline flushes and stalls, thread context switching, bus stalls, wait states, interrupt latency, cache systems and many other subtle aspects of cores and SoC's. While accurate behavior with these operations are necessary when benchmarking or making microarchitecture decisions, these are not normally used when doing embedded software development where the goal is "functional correctness as fast as possible".

The simulators we will explore are focused on speed, so they typically ignore hardware penalties described above. The execution phase in these simulators encompasses all aspects of the instruction in question, whether it be loading from memory, storing to memory, performing some math operation, branching, or any combination of those operations which the CPU allows, which happens in a single simulated clock tick.

QEMU

Without a doubt, the current most popular open source simulator is QEMU[1]. QEMU has a large and active development community, with new code being submitted virtually every day[2]. It has also seen integration into the official Linux kernel (in the form of KVM), which further guarantees continued development.

The QEMU project provides two main entry points for the Linux developer. First, there is the Linux userland simulation environment. This allows execution of unmodified Linux userspace programs (when the host is also a Linux system) by translating system calls on the fly (there is support for other userlands, but that is not terribly important). Second, there is a full system simulation environment. This allows for booting of full (and unmodified) operating systems and using virtual peripherals to do driver testing.

Since QEMU's code base shares a lot with the Linux kernel, generally any executable format that the Linux kernel can execute, QEMU's userland simulation can as well. This is beneficial as these are the formats that Linux developers are typically concerned with. If a format is not yet supported by QEMU, its code base makes it easy to import the relevant binfmt files from the Linux kernel. However, this is not too

---

[1] http://www.qemu.org/

[2] http://git.qemu.org/

common an occurrence as the number of binary formats in active use tends to be fairly constant.

Code sharing is emphasized heavily throughout QEMU, so the available feature set is typically similar across different architectures and SoCs. The architecture maintainers are allowed to focus on translation of their instruction set architecture (ISA) since the userland and peripheral simulation is handled by someone else (and works for everyone). Ports typically start off with userland support only and then quickly move on to full system simulation. The lead time tends to be dependent upon implementation of models for the hardware devices that are specific to the architecture.

The central QEMU design is the often mentioned *dynamic translation functionality*. Target opcodes are read in, decoded by the respective architecture ports, turned into simple operations (add/subtract/etc), and then encoded into the host cpu's opcodes via the Tiny Code Generator (tcg). These translated host opcodes are then cached so that any time the same target code is executed, the cache can be used directly, thus bypassing the whole translation process. For example, if you have a simple "for" loop of 100 iterations, the first run through will incur the translation overhead from the target ISA to the host ISA, but the remaining 99 iterations will all execute at native cpu speeds.

The upside here is speed -- lots and lots of it. You can often see performance in the range of hundreds of mips (if not almost 1 GHz). This easily enables software development in the pre-hardware stages. For many people, the simulation environment might even be faster than the actual hardware. Keep in mind that performance is not just about how many cycles your core can execute; the system bandwidth between the core and external memory, as well as internal caches, and the contention between system devices (such as network cards and video displays) often times play just as significant a role. QEMU on your development PC with many dedicated processors, wider buses, and larger caches often enables people to do a lot more functional testing (at the "enter text in a box" level rather than "is my LCD wired correctly") in shorter periods of time.

The downside of dynamic translation is a significant increase in overall complexity. Much of this is due to the fundamental disconnect between the translation of the target opcodes and the execution of the host opcodes. The processor state (such as registers) may only be updated by the latter, so the former spends a lot of time indirectly referring to things. While this is not overly complex for simple instructions such as "add two registers" or "shift a register by an immediate value",

```
r5 = r2 + r1 ;
r3 = r6 >> 4 ;
r3 >>= 17 ;
```

people can easily get confused when everything is indirect. For example, when working with a shift instruction where the shift magnitude is specified by a cpu register,

```
r3 >>= r0 ;
```

extra host opcodes have to be generated to make sure the shift magnitude is within a valid range. You cannot simply check the register value directly and then emit different op-codes based on that state during translation -- it must be done at execution time as the cpu state is only known at that point.

This complexity increase also shows up heavily when trying to track down misbehavior. If you write code that uses a bad pointer, you now have to coordinate the host opcodes, which actually executed and made the bad reference, back to target opcodes which your own high level code actually produced. QEMU includes a built-in gdb stub, but it is not bulletproof.

There are many QEMU based projects, one of note is QEMU-SystemC[3], which is a modification to QEMU which allows mixed HW/SW simulations. It simulates SystemC-described modules, as if they were connected to a system bus (PCI in Intel platforms, AMBA in ARM platforms, etc.).

---

[3] http://www.greensocs.com/Projects/QEMUSystemC

GNU Simulator
Another popular project is the GNU Simulator. Many people do not use it directly, or even realize that they are using it in the first place. While the GNU sim can be run from the command line (via the aptly named `run` program), it is typically utilized indirectly with the GNU Debugger (GDB) by using the "`target sim`" command. This tight integration is both a boon and a curse. On one hand, it means people are using it, but on the other, it suffers from lack of recognition and an active development community (specifically, the sim pieces -- obviously GDB itself is an extremely active and healthy community, probably bar none in the debugger world).

One unintentional downside of this tight integration to GDB and the entire GNU stack (libiberty, libbfd, libopcodes, etc), ends up restricting the GNU Simulator to only architectures and file formats that this stack supports. If your target already has support in the GNU bfd/assembler/disassembler, you should be all set. However, if your file format is not supported by bfd (such as the Linux embedded bFLT (binary FLAT) format), you are pretty much out of luck.

The simulator feature set varies greatly between target architectures. Some provide only partial instruction (ISA) level simulation, while others provide user mode simulation. Still others provide models of core devices (such as timers, interrupt controllers, user/supervisor modes, memory management unit, etc.) and a variety of peripherals that are typical to a SoC (UART, flash, network, SPI, etc). You will probably have to refer to the source code for your particular architecture to find out what exactly is available. The documentation for the GNU sim tends to be light at best to non-existent.

Fundamentally, all simulator ports operate in the same way. The code in question is loaded into the target memory space and then a main loop interprets each instruction one at a time. Think of this as a CPU without a pipeline (or cache) -- every simulated tick fetches one instruction, decodes it, and then executes it. After this, any pending events are handled (such as an exception or interrupt), and then things repeat.

The upside to this method is simplicity. For new simulator ports, it is fairly trivial to take the existing disassembler code out of the opcodes directory, stick a little bit of state to it, glue in the sim hooks, and have a functional ISA level port. Furthermore, when things go wrong, debugging a single stepping system that decodes and interprets the state at the same time is significantly more straight forward to understand and fix.

The obvious downside is speed. You will typically see performance in the range of 10 MIPS. Perhaps for tiny microcontrollers, this is blazing fast, but for modern targets that are expected to be running a full Linux environment and significant amount of userland code, this can be a hassle. It takes the simulation component from a full-fledged part of development and regales it to quick, one-off smoke tests.

Other Simulators
A few more projects worth mentioning in passing are Skyeye, coldfire, softgun, and armulator projects. The coldfire and softgun are fairly complete system simulators, but lack active development and acceptance by the wider community. The Skyeye and armulator simulators basically boil down to taking an old version of the GNU simulator, forking it, customizing it heavily for the desired targets, and presto! These projects have seen varying degrees of development and success since their inception, but we will refrain from voicing too much of an opinion on the respective designs (or lack thereof). Perhaps part of the reason for forking rather than working with the original GNU simulator project to keep things integrated can be traced back to the small (and unresponsive) upstream development community. If your system is explicitly supported by one of these projects, and you want to focus on your own code, then perhaps these will be a good fit. However, for people who are interested in something a bit longer term, or perhaps contributing to development of a simulator itself, we would recommend one of the earlier projects discussed in this paper.

There are also other non-GNU sim, and non-QEMU projects like ovp[4], Simit-ARM[5], SimpleScaler[6], and many others which are distributed under a variety of licenses, some approved open source or free software license, and some not -- but they might have a feature that you need.

We have ignored simulation projects which are not generally useful to the embedded developer such as bochs and VirtualBox. While they are both great projects, they are limited to x86 based cpus running on x86 based cpus.

Architecture Introduction

Before discussing the exact implementation and testing methodologies used, a brief understanding of the instruction set under test is necessary. In these examples, we will look at the Blackfin architecture (originally the Micro Signal Architecture (MSA) core developed jointly by Analog Devices, Inc. and Intel Corporation), which has an easy to understand assembly language. The Blackfin processor core contains two 16-bit multipliers, two 40-bit accumulators, two 40-bit arithmetic logic units (ALUs), four 8-bit video ALUs, and a 40-bit shifter. The processor can handle 8-, 16-, or 32-bit data from the register file. The Blackfin processor instruction set uses a combination of 16, 32 and 64-bit instructions, optimized so that 16-bit opcodes represent the most frequently used instructions. Complex DSP instructions are encoded into 32-bit opcodes as multifunction instructions. Blackfin products support a limited multi-issue capability, where a 32-bit instruction can be issued in parallel with two 16-bit instructions. This allows the programmer to use many of the core resources in a single instruction cycle. All these examples can occur within a single CPU cycle.

```
R2 = R2 +|+ R4, R4 = R2 -|- R4 (ASR) || I0 += M0 (BREV) || R1 =[I0] ;
        /* Add/subtract two 16-bit vector values while incrementing an Ireg
         * and loading a data register. */
A1=R2.L*R1.L, A0=R2.H*R1.H || R2.H=W[I2++] || [I3++]=R3 ;
        /* Multiply and accumulate to Accumulator while loading a data
         * register and storing a data register using an Ireg pointer. */
R3.H=(A1+=R0.L*R1.H), R3.L=(A0+=R0.L*R1.L) || R0=[P0++] || R1=[I0] ;
        /* Multiply and accumulate while loading two data registers.
         * One load uses an Ireg pointer. */
```

The processor's computational units have three definitive register groups:

- Data Register File consists of eight registers, each 32 bits wide (R7:0), and two dedicated, 40-bit accumulator registers, called A0 and A1.
- Pointer Register File, P[5:0], SP (stack pointer), FP (Frame Pointer)
- Data Address Generation (DAG) registers. I[3:0], M[3:0], B[3:0], L[3:0]

The Blackfin's 16-bit instruction's are encoded from this table, where instruction groups are built from multiple fields - including a fixed portion, and typically source (src) and destination (dst) registers, register groups (grp), operations (op), arithmetic operations (aop), and offsets.

```
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
+---+---+---+---|---+---+---+---|---+---+---+---|---+---+---+---+
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |.prgfunc.......|.poprnd........| ProgCtrl
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |.a.|.op....|.reg.......| CaCTRL
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |.W.|.grp.......|.reg.......| PushPopReg
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |.d.|.p.|.W.|.dr........|.pr........| PushPopMultiple
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |.T.|.d.|.s.|.dst.......|.src.......| ccMV
| 0 | 0 | 0 | 0 | 0 | 1 |.I.|.opc.......|.G.|.y.........|.x.........| CCflag
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |.op....|.reg.......| CC2dreg
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |.D.|.op....|.cbit..............| CC2stat
| 0 | 0 | 0 | 1 |.T.|.B.|.offset................................| BRCC
| 0 | 0 | 1 | 0 |.offset........................................| UJUMP
| 0 | 0 | 1 | 1 |.gd........|.gs........|.dst.......|.src.......| REGMV
| 0 | 1 | 0 | 0 | 0 | 0 |.opc...........|.src.......|.dst.......| ALU2op
```

[4] http://www.ovpworld.org/

[5] http://simit-arm.sourceforge.net/

[6] http://simplescalar.com/

```
| 0 | 1 | 0 | 0 | 0 | 1 | 0 |.opc.......|.src.......|.dst.......| PTR2op
| 0 | 1 | 0 | 0 | 1 |.opc.......|.src.................|.dst.......| LOGI2op
| 0 | 1 | 0 | 1 |.opc.......|.dst.......|.src1......|.src0......| COMP3op
| 0 | 1 | 1 | 0 | 0 |.op|..src.....................|.dst.......| COMPI2opD
| 0 | 1 | 1 | 0 | 1 |.op|.src.......................|.dst.......| COMPI2opP
| 1 | 0 | 0 | 0 |.W.|.aop...|.reg.......|.idx.......|.ptr.......| LDSTpmod
| 1 | 0 | 0 | 1 | 1 | 1 | 0 |.br| 1 | 1 |.op|.m.....|.i.....| dagMODim
| 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 |.op....|.i.....| dagMODik
| 1 | 0 | 0 | 1 | 1 | 1 |.W.|.aop...|.m.....|.i.....|.reg.......| dspLDST
| 1 | 0 | 0 | 1 |.sz....|.W.|.aop...|.Z.|.ptr.......|.reg.......| LDST
| 1 | 0 | 1 | 1 | 1 | 0 |.W.|.offset............|.reg...........| LDSTiiFP
| 1 | 0 | 1 |.W.|.op....|.offset........|.ptr.......|.reg.......| LDSTii
+---+---+---+---|---+---+---+---|---+---+---+---|---+---+---+---+
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
```

QEMU and GNU Sim implementation comparisons

A very basic Blackfin instruction - the Move Conditional instruction, which moves source register contents into a destination register, depending on the value of CC (Condition Code) bit. The source and destination registers can be any data register (R7:0) or any pointer register (P5:0, SP, FP)

```
Assembly Mnemonic             /* what it does   */
IF CC DPreg = DPreg ;         /* move if CC = 1 */
IF !CC DPreg = DPreg ;        /* move if CC = 0 */
```

In the GNU simulator, things are a very straight forward implementation in C. If the condition is set, write the destination register with the contents of the source register..

```
static void
decode_ccMV_0 (SIM_CPU *cpu, bu16 iw0)
{
  /* ccMV
     +---+---+---+---|---+---+---+---|---+---+---+---|---+---+---+---+
     | 0 | 0 | 0 | 0 | 0 | 1 | 1 |.T.|.d.|.s.|.dst.......|.src.......|
     +---+---+---+---|---+---+---+---|---+---+---+---|---+---+---+---+  */
  int s  = ((iw0 >> CCmv_s_bits) & CCmv_s_mask);
  int d  = ((iw0 >> CCmv_d_bits) & CCmv_d_mask);
  int T  = ((iw0 >> CCmv_T_bits) & CCmv_T_mask);
  int src = ((iw0 >> CCmv_src_bits) & CCmv_src_mask);
  int dst = ((iw0 >> CCmv_dst_bits) & CCmv_dst_mask);
  int cond = T ? CCREG : ! CCREG;

  if (cond)
    reg_write (cpu, d, dst, reg_read (cpu, s, src));
}
```

In QEMU, the basics are still there, but it is a little more complicated since TCG mnemonics must be generated, and then compiled. Since TCG does not support a conditional move as a single instruction, it is implemented as a conditional jump, which jumps over the move if the condition is not true. A temporary label must be generated to handle this possible jump.

```
static void
decode_ccMV_0(DisasContext *dc, uint16_t iw0)
{
  /* ccMV
     +---+---+---+---|---+---+---+---|---+---+---+---|---+---+---+---+
     | 0 | 0 | 0 | 0 | 0 | 1 | 1 |.T.|.d.|.s.|.dst.......|.src.......|
     +---+---+---+---|---+---+---+---|---+---+---+---|---+---+---+---+  */
  int s  = ((iw0 >> CCmv_s_bits) & CCmv_s_mask);
  int d  = ((iw0 >> CCmv_d_bits) & CCmv_d_mask);
  int T  = ((iw0 >> CCmv_T_bits) & CCmv_T_mask);
  int src = ((iw0 >> CCmv_src_bits) & CCmv_src_mask);
  int dst = ((iw0 >> CCmv_dst_bits) & CCmv_dst_mask);
  int l;
  TCGv reg_src, reg_dst;

  reg_src = get_allreg(dc, s, src);
  reg_dst = get_allreg(dc, d, dst);
  l = gen_new_label();
```

```
    tcg_gen_brcondi_tl(TCG_COND_NE, cpu_cc, T, l);
    tcg_gen_mov_tl(reg_dst, reg_src);
    gen_set_label(l);
}
```

One may think that it would be better to find the src and destination registers only if the test condition was met (after the `tcg_gen_brcondi`), but the optimization paths in TCG will do this for you.

Simulator Performance comparisons

Running a few examples benchmarks cachebench[7] , drhystone[8], and whetstone[9], on 3 different systems:
- native on embedded hardware
- native on x86 host
- qemu equivalent of the embedded hardware on the x86 host

Comparison provides some insight as to why how larger caches, and faster clock speeds on host machines help out with overall performance of simulation both on qemu, and the GNU Simulator.

Dhyrstone results

Blackfin CPU Core Speed	Dhrystone (DMIPS) (Hardware)	Dhrystone (DMIPS) (GNU Sim)	Dhrystone (DMIPS) (QEMU)
700MHz [10]		606.3	
600MHz	519.7		
500MHz	433.1		
400MHz	346.5		
300MHz	259.8		
200MHz	173.2		
15MHz [11]			12.9

Whetstone results

cachebench results

results

Testing Simulators

In general, the trouble of developing on any system other than the one you intend to actually ship on is a mismatch in behavior. There are a variety of issues in play to demoralize you, any of which can be a significant show stopper. The people developing the simulator are rarely the people designing the

[7] http://icl.cs.utk.edu/projects/llcbench/cachebench.html

[8] http://en.wikipedia.org/wiki/Dhrystone

[9] http://en.wikipedia.org/wiki/Whetstone_%28benchmark%29

[10] equivalent CPU speed, based on Dhrystone benchmark results

[11] equivalent CPU speed, based on Dhrystone benchmark results

CPU, which means the simulator developers are working off of documentation of the ISA (and the people writing that documentation oftentimes are not the designers either). Even if the hardware designers were the simulator developers, that does not guarantee accuracy when moving from actual hardware back to a software representation. Especially when the intended hardware specification/behavior does not even match the actual hardware behavior (often times referred to as "anomalies" or "bugs" or "oh well, close enough, ship it anyways"). This is all at the instruction set level, and things often get harder with more complicated hardware logic as is found on microcontrollers (timers, interrupts, etc) and SoCs (UARTs, SPI/I2C buses, etc). Even at the hardware level, verifying expected behavior with unit tests requires utilizing multiple devices simultaneously (which sort of defeats the point of unit tests). Think of the DMA controller which reads data from memory and feeds it to a high speed serial peripheral block which wiggles pins connected to external hardware devices (like an LCD).

So how, do we bring sanity to this environment? Fuzz, lots and lots of fuzz. Think hand grenade in a Muppets show.

It is not enough to just boot a kernel, or run some user-space application to test a simulator when you are under pressure to make a product release. The last thing you want to find, is that an unknown difference in the simulator and hardware cause you to spend time investigating a phantom issue in your codebase. It's not just a city in Italy; assuming a simulator is a 100% accurate representation of the hardware is Baloney.

To better understand the simulator accuracy, we will look at the way that simulators are tested.

Directed tests
Directed testing is the minimum test to determine correctness of simulator. Correctness testing will need some type of oracle, to tell the right behavior from the wrong one -- normally this is the person writing the tests. For example, this Blackfin test, loads the value 5 into a register, subtracts one, and checks the result (4) . If the result is not 4, the `DBGA` instruction asserts a failure.

```
# mach: bfin
.include "testutils.inc"
        start

        R0 = 5;
        R0 += -1;
        DBGA ( R0.L , 4 );
        pass
```

To better understand the number (and size) of various architectures directed tests the GNU sim directory, here's a table!

architecture	GNU Sim Number tests	GNU Sim Size of tests	QEMU Number tests	QEMU Size of tests
alpha			6	84k
arm	178	3.0M		
bfin	814	15M	420	5.5M
cris	363	5.0M	107	920K
fr30	107	1.8M		
frv	842	17M		
h8300	64	2.4M		

m32r	109	1.8M		
mips	16	396K		
sh	81	1.5M		

None of this means with any certainty that any one architecture is any better or worse tested than another, but it may give an idea of the public regression testing that is going on on your architecture of choice. With a rapidly developing simulator like QEMU, having a substantial regression test has a greater chance of providing a more stable, and more accurate to the hardware release.

Random Testing
Random testing is a form of functional testing that is useful when the time needed to write and run directed tests is too long (or the complexity of the problem makes it impossible to test every combination). One of the big issues of random testing is to know when a test fails. As with all testing, a golden model is needed. With directed testing, you can rely on the oracle to place code assertions as the golden model. In other situations, common with hardware or simulator development, you have two different implementations of the same specification (one is “the golden model”, the other is “the implementation”). If both implementation and golden model agree to a defined accuracy/result, then the test passes.

When doing random testing you must, of course, ensure that your tests are sufficiently random, and that they cover the spec. Repeating the same test for two weeks is two weeks of test time wasted.

When using random tests in a test suite, it is necessary to ensure that failures can be reproduced (perhaps by logging the random number seed, or logging the test snippet). A non-reproducible bug does not help anybody! Furthermore, if random testing reveals bugs, it would be wise to create non-random unit tests from these cases, because tomorrow's random input will not necessarily protect you against regressions.

The reason it is not practical to do directed tests on a complex machine, like any modern processor architecture, is the complexity of the instructions types. For example, a single 32-bit instruction on the Blackfin (a dsp32mac instruction), is represented as:

```
R1 = (A1 -= R3.L * R4.H) (M), R0 = (A0 += R3.H * R4.L) (IS);
```

This instruction does two Multiply-ACcumulates (MAC) to the accumulators and extracts the results to a data register (saturating the extraction at 32 bits). The inputs to this single instruction is A0 (40 bits), A1 (40 bits), R3 (32 bits), and R4 (32 bits). To test every combination of input for this single instruction, would take $9.62143091 \times 10^{48}$ iterations. Running at 1GHz, it would still take $1.83056144 \times 10^{34}$ years to test one instruction thoroughly (expected lifetime of the sun is only another 10^{11} years).

It would be impractical to think every combination of different values and register possibilities could be tested in a reasonable amount of time, which forces random test generation/checking.

The random testing that was done for the Blackfin architecture was done in multiple steps.

The first was to generate a table of “known good instructions”. This was done by running a small bare metal application on the hardware, which single stepped/cycled through every 16, and 32-bit instruction, and combinations of 64 bit instructions, from user-mode to determine if the instruction caused an illegal instruction exception, supervisor-only instruction exception, or was a valid instruction.

A small standalone application was built from this table, which could be tested on the simulators under test. This test determined if the op-codes were being decoded properly, and “bad instructions” were generating illegal instruction exception in the simulation environments. Doing this found many issues in the simulators, as well as issues in the documentation (op-codes in the manuals were missing, or left out,

incorrect length of bit fields in the instruction encoding). However, it did not ensure correctness of the instruction, only that it faulted or excepted in the simulators the same as it did in the hardware.

The next step was to do correctness testing at the instruction mnemonic level. A small C application, gdb test scripts, and bash shell script was written which:

1. Creates a random 32-bit number
2. Applies various bit masks to ensure that a certain instruction group is tested. This is done to ensure a fair distribution of tested instructions.
3. Check this cooked value against the known good instructions, to make sure it actually runs on the hardware. If the it is a invalid op-code, ensure it disassembles as “illegal instruction.”
4. Determine the instruction mnemonic this checked and cooked value is associated with. Since this runs on the hardware, both the assembler and disassembler are checked, and logged if an error occurs.
5. Save the instruction mnemonic into an .S file for later assembly.
6. Repeat steps 1 -5 for 2000 (or more) times.
7. Generate a random register loads, and place this into the .S file
8. Assemble this .S file, and run it on the hardware in gdb under the control of a small gdb script which single steps through the .S file, checking for register differences, and denotes the differences in a log file. snippet of gdb script:

```
if $$old_r0 != $r0
        printf "%c\tcheckreg R0, 0x%08x;\n", $$pre, $r0
end
```

9. Take the log file which was created in step 8, with the test assertions (shown below), and run it on the hardware again (to ensure things are OK)

```
R7.H = (A1 = R7.L * R0.L) (M), R7.L = (A0 -= R7.H * R0.H) (ISS2);
checkreg R7, 0x00007fff;
checkreg A0.w, 0x31b5efeb;
checkreg A0.x, 0x0000007e;
checkreg A1.w, 0x00000000;
checkreg A1.x, 0x00000000;
checkreg ASTAT, ( _VS | _V | _AV1S | _AV0S | _CC | _V_COPY | _AC0_COPY);
```

10. Run the test assertions on the simulator, and log any test failures. If the test passes, throw out the test, and repeat. It is possible to cycle steps 7-10 while the new test (steps 1-5) is being created.

The limitations of random tests in this method are plenty.

- The tests on the hardware are run as userspace under the Linux kernel (this excludes the testing of any supervisor only instructions or supervisor managed resources, including peripherals, interrupts, DMA, etc).
- Test can be located anywhere in memory, so the program counter is not checked; therefore change of flow (Branches, Jumps, Calls, Returns, hardware loops, etc) are not tested.
- When randomly manipulating the stack in Linux userspace, it is pretty easy to crash the application, so any stack operations/modifications are excluded (and therefore not tested).
- Random reads/writes into random parts of memory will cause Linux userspace to crash, so memory reads/writes are limited to specific parts of the application’s memory space, reducing test coverage.
- Multiple opcodes can disassemble to the same mnemonic. For example, the 32-bit op-code encoding for the dsp32mac instruction:

```
 31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16
+---+---+---+---|---+---+---+---|---+---+---+---|---+---+---+---+
| 1 | 1 | 0 | 0 |.M.| 0 | 0 |.mmod.........|.MM|.P.|.w1|.op1...|
|h01|h11|.w0|.op0...|h00|h10|.dst.......|.src0......|.src1......|
+---+---+---+---|---+---+---+---|---+---+---+---|---+---+---+---+
 15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
```

  where `op1` (MAC1) and `op0` (MAC0) control the operation of the DUAL macs:

```
00 operation is “=”
01 operation is “+=”
10 operating is “-=”
11 operation is NOP
```

  Therefore, the hardware, when it sees that `op0` and `op1` are both “11” does nothing, and the

instruction is the same as a “MNOP” instruction (32-bit NOP). This is how it is disassembled by the disassembler. However, when the assembler assembles the “MNOP” instruction it translates it into the specific MNOP “`0xC0031800`“ op-code, and this is the one that is tested on the hardware and simulator.

Instructions which are not simulated

If we examine the most “fixed” instruction from 16-bit opcode table table, (Cache Control), it only has 5 bits out of 16 which can change the function of the instruction. Looking at the disassembler can provide some insight to the way the instruction works.

```
static int
decode_CaCTRL_0 (TIword iw0, disassemble_info *outf)
{
  /* CaCTRL
       +---+---+---+---|---+---+---+---|---+---+---+---|---+---+---+---+
       | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |.a.|.op....|.reg.......|
       +---+---+---+---|---+---+---+---|---+---+---+---|---+---+---+---+  */
  int a   = ((iw0 >> CaCTRL_a_bits) & CaCTRL_a_mask);
  int op  = ((iw0 >> CaCTRL_op_bits) & CaCTRL_op_mask);
  int reg = ((iw0 >> CaCTRL_reg_bits) & CaCTRL_reg_mask);

  switch (op)
       {
       case 0 : OUTS (outf, "PREFETCH["); break;
       case 1 : OUTS (outf, "FLUSHINV["); break;
       case 2 : OUTS (outf, "FLUSH["   ); break;
       case 3 : OUTS (outf, "IFLUSH["  ); break;
       default: return 0;
       }

  OUTS (outf, pregs (reg));

  if (a == 1)
       OUTS (outf, "++]");
  else
       OUTS (outf, "]");

  return 2;
}
```

This instruction only operates on P registers, and can have optional post increments. But looking at the simulators - the instructions are mostly skipped/NOP’ed (since there is not any cache simulation - it would be painfully slow).

```
static void
decode_CaCTRL_0 (SIM_CPU *cpu, bu16 iw0)
{
  /* CaCTRL
       +---+---+---+---|---+---+---+---|---+---+---+---|---+---+---+---+
       | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |.a.|.op....|.reg.......|
       +---+---+---+---|---+---+---+---|---+---+---+---|---+---+---+---+  */
  int a   = ((iw0 >> CaCTRL_a_bits) & CaCTRL_a_mask);
  int op  = ((iw0 >> CaCTRL_op_bits) & CaCTRL_op_mask);
  int reg = ((iw0 >> CaCTRL_reg_bits) & CaCTRL_reg_mask);
  bu32 preg = PREG (reg);

  if (INSN_LEN == 8)
       /* None of these can be part of a parallel instruction */
       illegal_instruction_combination (cpu);

  /* No cache simulation, so these are all NOPs. */

  if (a)
       SET_PREG (reg, preg + BFIN_L1_CACHE_BYTES);
}
```

It’s impossible to test things that are not actually simulated.

When to stop testing?
Testing is potentially endless. Nobody can test until all the defects are unearthed and removed -- it is simply impossible. At some point, the software must be released -- testing must end sometime. The question is when.

In a commercial setting, testing is a trade-off between budget, time and quality, driven by profit models. The most often used approach is to stop testing whenever some, or any of the allocated resources -- time, budget, or test cases -- are exhausted.

In an open source setting, testing for some architectures is a never ending, on-going process which is managed by merge windows, and release schedules. For other architectures, testing stops when someone loses interest in doing the testing, and decides to re-purpose their test machine into a BZFlag server. Unfortunately it is up to the user to determine which test methodology is being deployed on the architecture of interest.

The Pewter Lining
While we have covered many reasons for why software developers should be aware of and actively using system level simulators, the reality is that there are some things that cannot replace actual hardware. We will mention many of the issues, but will not delve into detail as most are related to where the software meets the hardware. That can be put off for another time (and paper?).

Many open source simulators have the advantage of being widely used. However, anyone who has read a recent financial report knows, "*past performance is no guarantee of future results*". Unless you are only planning on doing exactly the same as other users, don't be surprised if you run into a new issue. This could be related to that fancy "display" which is wired up "*correctly*" in simulation, which just means it matched the developers PCB, just not yours. There's no way to verify your PCB has its wires connected properly to the SoC. That can only be validated with actual hardware.

Similarly, interesting and difficult to track down issues can easily arise in the analog space that are completely hidden by working in a pure digital environment. Over heating, electromagnetic interference under certain component usage, brown outs, and short circuits just to name a few. Extensive QA testing on the final board is the only way to ferret out these problems.

Some of the hardware components that are on the final board might not have models available in the simulation environment. Although this can be overcome somewhat by being a good open source citizen and implementing those models and sending them back to the project you're working with.

One feature touted as an advantage earlier is also a disadvantage. Namely, having a simulated system that performs better than actual hardware. Those larger buses, caches, etc, can sometimes come back to bite you if you allow the software development to grow beyond the budget of the target system. While interfaces might seem snappy in a simulator, the routines may take up too much cpu/memory/etc resources and lead to an unusable end user experience.

Along those lines, utilization and possible contention on shared resources (such as a cache, or system bus) cannot be truly known until the final integration. How much you will be impacted by pipeline stalls, and what your true performance will be is always a little bit of a mystery in a non-cycle accurate simulation. While you can calculate theoretical limits and make sure the system comes in under that budget, the actual answer is hardly ever known. While there might be enough system bandwidth to accommodate a DMA driven display most of the time, the edge cases such as incoming ethernet packets combined with playing an audio alert and handling core cache misses can only be noticed on the actual hardware.

Finally, we get back to the focus on this paper -- namely, instruction set accuracy. Not every simulator and port will have full coverage of the instruction set, especially when it comes to uncommon or esoteric options. Further, there might even be some design trade-offs made where certain hardware correctness

is sacrificed in the name of simulation speed. Hopefully these caveats are clearly spelled out so you can make educated decisions rather than getting caught with your proverbial pants down.

Simulators absolutely occupy a critical place in the software development cycle, but they are not the sole component.

Robin Getz has been involved with open source development for the better part of 10 years, making contributions to the Linux kernel, GNU binutils, GNU simulator, qemu, and various other open source applications.

Mike Frysinger has been a founder, contributor, and maintainer of many open source projects, including (but not limited to) many you've heard of, many you haven't heard of, and more. You probably don't care either way.