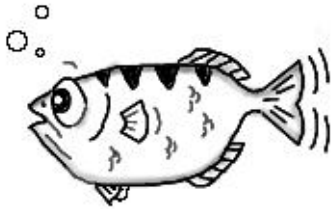
A Tale of Two Sims



GNU Sim
http://sourceware.org/gdb/

QEMU
open source processor emulator

QEMU
http://qemu.org/

Gentoo
http://gentoo.org/

Mike Frysinger
2012 Libre Software Meeting

Blackfin
http://blackfin.uclinux.org/

Agenda

- Introduction
  - Simulation vs Virtualization
  - Types of simulation
  - Other simulators
  - Blackfin architecture
- Simulator Comparisons
  - Overview
  - Execution modes
  - Tracing & Profiling
  - Memory handling
  - Device models
  - Target code simulation
  - Testing
- Conclusions (freedom!)
- Maybe extra stuff ...

Simulation != Virtualization

- Virtualization
  - Not what this talk is about
  - Run multiple native OS's in parallel on native hardware
  - Not useful for cross-compilation / embedded (generally)
  - Interesting, but not really
    - Instructions run natively (cheat)

- Simulation
  - Is what this talk is about (!)
  - Run any target on any host
    - Host: AMD 64bit CPU development desktop
    - Target: ARMv7 CPU embedded system
  - Instructions get decoded/translated/executed → simulated

- Blurry lines
  - What is Emulation?
  - Some projects (e.g. QEMU) do both

Cycle vs Instruction Accurate

- Cycle Accurate Simulators (CAS)
  - CAS must ensure that all operations are executed in the proper simulated time/cpu ticks
    - Branch misprediction, cache misses/flushes, instruction or data fetches, pipeline flushes and stalls, bus stalls, wait states, interrupt latency, cache systems, and many other subtle aspects of SoCs
  - Used to be all the rage

- Instruction Set Simulators (ISS)
  - a.k.a. ”Functional” Simulators
  - Ignores the time/clock aspects, and focuses on wall time execution speed
  - The execution phase in these simulators encompasses all aspects of the instruction in question
    - Loading from memory, storing to memory, performing math operations, branching, or any combination of those operations which the CPU allows, which happens in a single simulated clock tick
    - Often times ignores cache issues completely – pretends they don't exist

Other Projects

- A few semi-generalized simulators
  - Skyeye (mostly arm)
- Many specialized simulators out there
  - Might work better for the ISA in question
    - Might not have as many peripherals supported
  - Armulator (arm)
  - Coldfire (m68k/coldfire)
  - Softgun (arm)
  - Ski (ia64)
  - Hercules (s390)
- Many specialized virtualization projects
  - Bochs (x86)
  - VirtualBox (x86)
  - VMWare (x86)
  - Xen (arm, ia64, x86)

Intro to the Blackfin ISA

- Started off as the Micro Signal Architecture (MSA)
- Jointly developed by Analog Devices, Inc. and Intel Corporation
- RISC instruction set
  - 16, 32, and 64 bit fixed width insns
- Easy to understand (algebraic) assembly language
- Dedicated registers (generally 32 bit)
  - Data (R0 – R7)
  - Accumulators (A0 – A1)
  - Pointer (P0 – P5, FP, SP)
  - Circular buffers (I0 – I4, M0 – M4, B0 – B4, L0 – L4)
- Easy access to 8, 16, and 32 bit portions of data registers
  - R0.B, R0.H/R0.L, R0
- Geared towards digital signal processing (DSP)
  - Fully functional microcontroller as well

Example Blackfin Insns

- Moves

```
R0 = -1 (x);            R1.L = 0x1234;
CC = R0 = 0;            IF CC R3 = R1;
```

- Math

```
R0 = R1 + R2;           R0 = R0 – R1;
```

- Load

```
R3 = [P4];              R2.H = W[P3 + 0x100];
```

- Store

```
[P4 + 0x400] = R3;      B[P2] = R7.B;
```

- Branches

```
IF CC JUMP memcpy;      CALL printf;
JUMP (P0);              CALL (P3);
```

- Stack

```
R4 = [SP++];            [--SP] = R6;
LINK 0x100;             UNLINK;
```

QEMU

- Popular open source simulator
  - Vibrant community doing stuff every day
- Started by Fabrice Bellard
  - Pretty smart dude
- Shares a lot of code with compatible projects
  - the Linux kernel (maybe you've heard of it)
  - GPL-2 disassemblers
- Project Design focuses
  - Portable (target and host)
  - Fast
  - Functional
  - Recently was ported to javascript to run in your browser?!
- It's simply awesome
- Documentation can be severly lacking
  - Wiki and code comments slowly improving

QEMU Technology

- SMP & threading fully supported
- Can easily work with any format
  - Linux ELF & FLAT binfmts currently
- Fundamental disconnect between decode and execute stages

```
while (1) {
    if (translated_opcodes_not_cached())
        translate();
    execute_translated_opcodes();
    if (event_pending())
        process_events();
}
```

- Dynamic translation
  - Target opcodes → TCG → host opcodes → cache → execute
- Translation of opcodes cannot utilize CPU state
  - State is not known until execution time
- Once things are cached, run at native speeds!

QEMU Technology

- Super fast!
  - Hundreds of MIPS (if not 1 GHz+)
- Can be much more complicated
  - Correlate host opcodes with target opcodes with original language → makes you crazy
- Lots of code sharing
  - Device models not specific to a port
  - Arch maintainers focus on their arch
- Lots of device models ready to use
  - More than 300!
    - Many are arch-specific
    - Lots of common ones to play with though

GNU Sim

- Dormant development community
  - Many people not aware of its existence
  - Overshadowed by QEMU and other “new” projects
- Long history
  - Dates back to 1993
  - Huge code injection by IBM in 1999 (psim)
- Heavily integrated with GNU stack
  - Leverages a lot of common code
  - Code stays alive and counters bit roting
  - Allows focusing on simulator
- Easy to interact with GNU Debugger (GDB)
  - Just connect to the “sim” remote target
- No documentation
  - Read the code, then read it some more, then find someone who has read it and ask them

GNU Sim Technology

- Only Uniprocessor / single thread support
  - SMP framework is laid out, but disabled/TODO
- GNU stack sometimes limits usefulness
  - Stack needs to support your target
  - Can only work with known formats
    - ELF is in but bFLT is out
- Fundamentally very simple

```
while (get_next_insn()) {
   interpret_insn();          /* decode + execute */
   if (pending_events())
      process_events();
}
```

- Easy to debug and understand
- Easy to start new port
  - Take disassembler and add CPU state tracking

GNU Sim Technology

- Slow :(
  - ~10 MIPS
- Level of support varies greatly between ports
  - Some are ISA-only
  - Some support virtual mode
  - Some support full operating system mode
  - Some support user (Linux) mode
- Uses device tree syntax to describe layout
- Not a whole lot of off-the-shelf device models
  - ~60 models in total
    - <10 common ones
    - Half are just Blackfin/ADI models

QEMU Modes

- User mode simulation
  - Run target Linux/BSD/etc... programs natively
    - Target and host OS's must match
  - Converts system calls on the fly
    - Fairly complete system call coverage
    - Endian and bitwise translation
  - Great for simple program testing
  - Heavily used with cross-compiling
  - Accesses files/devices in your system directly
- (Operating) System mode simulation
  - Boot a full operating system
    - Specify a kernel, or a bootable device, or ...
    - Interact with it like another machine
  - All devices are virtual
    - Some level of “pass-thru” support

GNU Sim Modes

- Called ”environments”
- User mode simulation
  - Run target programs natively
  - Limited number of system calls currently supported
- Operating (System) mode simulation
  - Boot a full operating system
  - Only boot from “programs” and not devices
    - Give it a vmlinux ELF
- Virtual mode simulation
  - In between user and operating modes
  - Kind of like supervisor mode
    - Privileged insns ignored (CLI → NOP)
    - No device simulation
  - Useful for algorithm testing
    - Focus on ISA (used for compiler testing)
  - Quick & simple sandbox for playing

QEMU Tracing

- Little end-user tracing support
  - Mostly for QEMU developers
  - Hardware tracing in development
  - System calls (strace)
- No profiling support

QEMU Tracing Example

```
qemu-bfin -d in_asm,out_asm,cpu,exec,int -singlestep ./a.out

Trace 0x601e9830 [000010d6]
...more cpu state...
RETI: 00000000  RETS: 000010b4   PC : 000010d8
 R0 : 000000b4   R4 : 000010b4   P0 : 4080004c   P4 : 00000000
 R1 : 00002800   R5 : 00000000   P1 : 00000000   P5 : 00000000
 R2 : 00002974   R6 : 00000000   P2 : 00000000   SP : 407ffe6c
 R3 : 00000000   R7 : 00000000   P3 : 00000000   FP : 00000000
...more cpu state...
_interp_insn_bfin: iw0:0x3040
decode_REGMV_0: gd:0 gs:1 dst:0 src:0
----------------
IN:
0x000010d8:  R0 = P0;

OUT: [size=50]
0x601e98e0:  mov    0x85e4(%r14),%ebp
0x601e98e7:  mov    $0x10da,%ebx
0x601e98ec:  mov    %ebx,0x80(%rsp)
0x601e98f3:  mov    %ebp,0x85c4(%r14)
0x601e98fa:  jmpq   0x601e98ff
0x601e98ff:  mov    $0x10da,%ebp
0x601e9904:  mov    %ebp,0x869c(%r14)
0x601e990b:  xor    %eax,%eax
0x601e990d:  jmpq   0x6222d616
```

GNU Sim Tracing

- Significant framework for tracing
  - Includes debugging information (symbols/linenum/etc...)
  - Instruction decode/extract/execute
  - Memory accesses
  - Branches
  - ALU/FPU/VPU
  - Core (bus) accesses
  - Hardware events
  - System calls (strace)
- Significant framework for profiling
  - Instructions
  - Memory
  - Hardware
- All code under compile time knobs
  - Disable it all for a nice speed boost

GNU Sim Tracing Example

```
main() { int i = 123; return i; }

0000011c <_main>:
 11c:   00 e8 0b 00     LINK 0x2c;
 120:   b8 b0           [FP + 0x8] = R0;
 122:   f9 b0           [FP + 0xc] = R1;
 124:   28 60           R0 = 0x5 (X);
 126:   f0 bb           [FP -0x4] = R0;
 128:   f0 b9           R0 = [FP -0x4];
 12a:   01 e8 00 00     UNLINK;
 12e:   10 00           RTS;

core:     0x000124 #2    main    -IBUS FETCH 2 bytes @ 0x00000124: 0x6028
extract:  0x000124 #2    main    -_interp_insn_bfin: iw0:0x6028
extract:  0x000124 #2    main    -decode_COMPI2opD_0: op:0 src:5 dst:0
decode:   0x000124 #2    main    -decode_COMPI2opD_0: imm7:0x5
insn:     0x000124 #2    main    -R0 = 0x5 (X);
reg:      0x000124 #2    main    -wrote R0 = 0x5
reg:      0x000124 #2    main    -wrote PC = 0x126
core:     0x000126 #2    main    -IBUS FETCH 2 bytes @ 0x00000126: 0xbbf0
extract:  0x000126 #2    main    -_interp_insn_bfin: iw0:0xbbf0
extract:  0x000126 #2    main    -decode_LDSTiiFP_0: W:1 offset:0x1f grp:0 reg:0
decode:   0x000126 #2    main    -decode_LDSTiiFP_0: negimm5s4:0xfffffffc
insn:     0x000126 #2    main    -[FP + -0x4] = R0;
core:     0x000126 #2    main    -DBUS STORE 4 bytes @ 0x07fffef8: 0x00000005
reg:      0x000126 #2    main    -wrote PC = 0x128
```

GNU Sim Profile Example

```
Instruction Statistics
  Total: 13,445 insns
      ProgCtrl_nop:   210: **
   ProgCtrl_branch:    77: *
      ProgCtrl_cec:     4:
        PushPopReg:    21:
   PushPopMultiple:    61:
              ccMV:     4:
           CCflag: 1,880: **************************
          CC2dreg:     2:
          CC2stat:    20:
             BRCC: 1,902: ***************************
            UJUMP:   482: ******
            REGMV: 2,852: ****************************
           ALU2op:   761: **********
           PTR2op:    20:
          LOGI2op:    44:
          COMP3op: 1,830: *************************
        COMPI2opD:   826: ***********
        COMPI2opP:   134: *
         dagMODim:     1:
          dspLDST:    52:
             LDST:   910: ************
         LDSTiiFP:    14:
           LDSTii:   794: ***********
        LoopSetup:    27:
        LDIMMhalf:    40:
            CALLa:    58:
         LDSTidxI:    74: *
          linkage:    87: *
         dsp32mac:    36:
         dsp32alu:     8:
      dsp32shift:     9:
   dsp32shiftimm:   205: **
```

$ bfin-elf-run -p -v --env user ./a.out.static

```
CORE Statistics
  Total:  16,211 accesses
      read:     1,747: ****
     write:       475: *
      exec:    13,989: ********************

Model bf537 Timing Information
  Taken branches:                    1,690
  Untaken branches:                  764
  Cycles stalled due to branches:    0
  Cycles stalled due to loads:       0
  Total cycles (*approximate*):      0

Simulator Execution Speed
  Total instructions:       13,370
  Total execution time:     < 1 second
```

QEMU Memory Layout

- User mode mmaps files directly
  - Makes execution fast
  - Complicates host env greatly
    - Must make sure that host address mappings do not collide
    - QEMU is built as a PIE similar to the LDSO
  - Simplifies target env greatly
    - No need to do address space translation
    - No need to do hacky target relocation
  - Can get into trouble when mappings do collide
    - Tries to mitigate by mimicing the target kernel's default mmap settings and loading into areas typically unused
    - NOMMU kernels sometimes use addresses lower than the host's mmap_min_addr setting (default of 64KiB)
    - Some custom apps assume certain virtual addresses are free
      - If host loads a library there, you're out of luck
      - Uncommon practice, but not unheard of
- Operating mode has full softmmu
  - Will never run into address collisions

GNU Sim Memory Layout

- Sim core allocates chunk of memory address space
  - Regardless of operating mode
  - Translates addresses all the time
- Read all programs (code/data/etc...) into memory
  - No files are mapped after load phase
- Attach all hardware devices to address space
  - Only for operating mode
- Execute!
- Portable, but slower
  - Can simulate user mode apps under Windows/etc...

QEMU Device Modeling

- Uses variety of command line options for specifying device layout and options
  - Work is on going to clean up & unify with QDEV model
- Quite a lot of open coding
  - Work is on going to clean up & unify
- I/O read/write device memory (via sysbus)
  - Often used to model registers
- DMA is hard to use
- IRQs setup via sysbus
- GPIOs setup via QDEV

GNU Sim Device Modeling

- Uses device tree syntax for laying out hardware
  - Getting popular in the embedded world
  - Automatically instantiates & connects all models

- Clean device model
- Bus topology
  - Core → SPI bus → SPI clients
    \→ PCI bus → PCI clients
- I/O read/write device memory
  - Often used to model registers
- DMA read/write device memory
- ”Ports” between devices
  - Model signal lines
    - e.g. Interrupts, GPIOs, etc...
  - Control line levels

Implementation Comparison

- Simple Blackfin insn like conditional move
  - **`IF CC Dpreg = Dpreg;`**
    - CC is the Condition Code register
    - Dpreg can be any data or pointer register
      - R0 – R7, P0 – P5, FP, SP

- **`IF CC R0 = R1;`**
  - If CC=0, then do nothing
  - If CC=1, then move R1 into R2

- **`IF !CC P0 = P1;`**
  - If CC=1, then do nothing
  - If CC=0, then move P1 into P0

GNU Sim Implementation Example

```
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));
}
```

QEMU Implementation Example

```
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);
}
```

```
movi_i32 tmp0,$0x1
brcond_i32 CC,tmp0,ne,$0x0

mov_i32 R0,R1

set_label $0x0
```

Testing

- The trouble of developing on any system other than the one you intend to actually ship on is a mismatch in behavior
- It is not enough to just boot a kernel, or run some user-space application to test a simulator
  - Common methodology though :(
- Lack of testing can burn users bad
  - Difference between simulator and hardware can easily waste a lot of time investigating a phantom issue in codebase
- How to make sure it's OK?
  - Directed tests
  - Random tests

Directed Tests

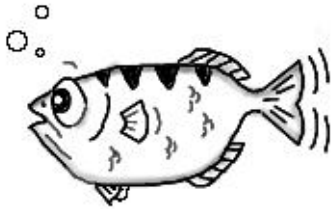
- Lots of tests doesn't necessarily mean better testing, but it sure helps to make sure regressions don't slip in
- Only showing arches with both QEMU and GNU Sim ports
- Mostly ISA tests, but includes some device tests too

architecture	GNU Sim number	GNU Sim size	QEMU Number	QEMU size
arm	169	684K	6	84k
blackfin	839	8.8M	417	2.7M
cris	384	1.6M	103	416K
lm32	0		64	256K
microblaze	0		0	
mips	14	1.1M	0	
powerpc	0		0	
superh	78	368K	0	

Concluding Conclusions

- QEMU
  - Fast!
  - Lots of people hacking on it
  - Not the easiest to develop for
    - Things getting better
- GNU Sim
  - Slow, but simple!
  - Great low level debugging
  - Easy to get to started

Questions ?



GNU Sim
http://sourceware.org/gdb/

QEMU
open source processor emulator

QEMU
http://qemu.org/

Gentoo
http://gentoo.org/

Mike Frysinger
2012 Libre Software Meeting

Blackfin
http://blackfin.uclinux.org/

Extra Material

- Even more crap to ramble about!

Why should you care?

- Used to be hardware development dominated overall
  - Software was an after thought
  - Not a whole lot with simple devices
- Trend has now reversed
  - Devices are complicated and do a lot more
  - Much more software running in a system
    - Requires more time to develop/test/verify/etc...
  - Software development now dominates overall
    - Typical project : 80% of embedded development costs are software
- What to do?
  - Nothing
    - Lose money/product/market/job/family/...
    - Cry a lot
  - Something
    - Probably still cry a lot

Why you should be using simulation

- Parallelize development
  - Hardware & software groups are rarely the same
  - Get “hardware” from day 1
- Simulation environment is easier to use
  - Focus on software bugs and not hardware bugs
  - No ugly wires, soldering irons, ...
  - No complicated board bring up
  - No contention for limited number of initial samples
- Simulation environment is faster
  - Desktops/Laptops have gobs of resources
    - CPU
    - Memory
    - Fat buses
    - Fast peripherals
  - Everything is in memory

Pitfalls

- Simulators are great, but can't do it all
- Can't verify actual board design
  - PCB layout/wiring
  - Noise (EMI)
  - Signal integrity
  - Heat
  - Power (brown outs, short circuits, ...)
- Model and hardware mismatch
  - Different components modeled
  - “Wired” differently
  - No relevant models available at all
  - Write a new model and submit it!

Yet More Pitfalls

- Simulation environment is too good
  - Larger buses, caches, etc...
  - Easy to develop past budget of actual hardware
  - Runs fast in simulation, but takes up too much cpu/memory/etc... on the final system
  - Terrible user experience
- Simulation environment not cycle accurate
  - Too slow
  - Functional matters more
- Resource contention not modeled
  - Simulation environment has “dedicated” buses between components but final system shares resources
  - Contention significantly higher
  - Theoretical calculations never line up to real world
  - Cache misses + DMA display + ethernet + audio → choke

My God It's Full of Pitfalls

- ISA itself might not be accurate
  - Experience greatly depends on the architecture port
  - Testing is an after thought
  - Uncommon insns
  - Esoteric options
  - Simulator design trade offs
    - Correctness may be sacrificed by speed
- Common cases are important (satisfy 99%)
- Uncommon cases are a pain (satisfy the whiny 1%)
- Check the documentation
  - Maybe that too was an after thought
  - Pray to the computer gods

Concluding Conclusions

- Simulation
  - You need it
    - Maybe you don't realize it
  - You want it
    - Maybe you can't admit it
- Critical to modern development cycles
  - Best to accept it
- Testing of simulator matters
  - But it isn't easy as it takes:
    - Time
    - Planning
    - More time
    - Effort
    - Dedicated test infrastructure (host PCs, target hardware, etc)
    - People
    - All the little details you'd rather forget