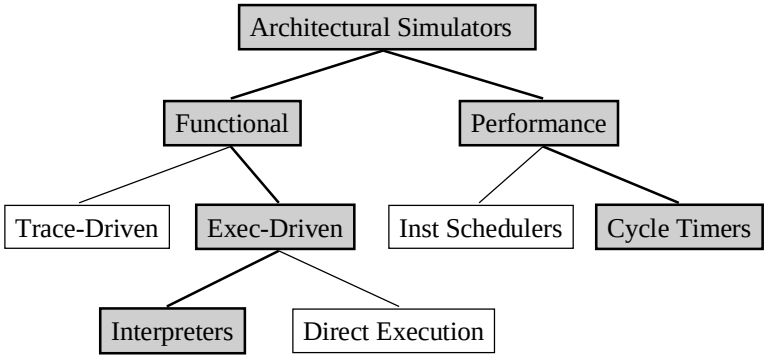
The SimpleScalar Tool Set as an Instructional Tool: Experiences and Future Directions

Todd M. Austin
Intel Microcomputer Research Labs
Oregon Graduate Institute
taustin@ichips.intel.com

Talk Overview

- SimpleScalar Tool Set Overview
- Initial Course Experiences
- Release 2.0 Enhancements
- Future Enhancements
- Summary and Contact Information

A Taxonomy of Simulation Tools



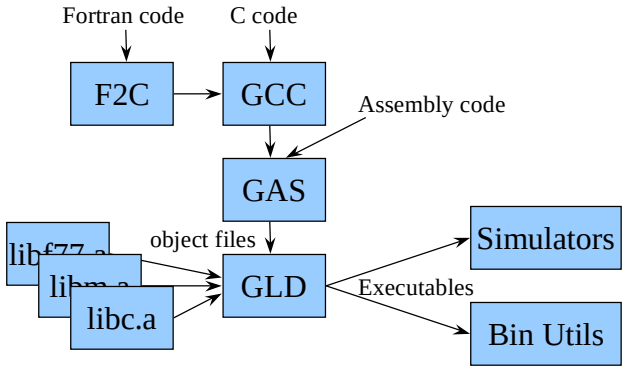
- shaded tools are included in the SimpleScalar Tool Set

The SimpleScalar Tool Set

- uniprocessor computer architecture test bed
  - compilers, assembler, linker, libraries, and simulators
  - targeted to the virtual SimpleScalar architecture
  - hosted on most common platforms
- developed during my dissertation work at UW-Madison
  - third generation simulation tool (Sohi → Franklin → SimpleScalar)
  - in development since '94, release 1 in July '96, with Doug Burger
  - release 2 in January '97
- backed by a growing community of users
  - researchers (40+ published papers) and instructors (10+ courses)
- freely available with source and docs from UW-Madison

  `http://www.cs.wisc.edu/~mscalar/simplescalar.html`

SimpleScalar Tool Set Overview



- compiler chain is GNU tools ported to SimpleScalar
- Fortran codes are compiled with AT&T's *f2c*
- libraries are GLIBC ported to SimpleScalar

Using the SimpleScalar Tool Set

- compiling a C program

```
ssbig-na-sstrix-gcc -g -O -o program foo.c -lm
```

- compiling a Fortran program

```
ssbig-na-sstrix-f77 -g -O -o program foo.f -lm
```

- compiling a SimpleScalar assembly program

```
ssbig-na-sstrix-gcc -g -O -o program foo.s -lm
```

- running a program

```
sim-safe [-sim opts] program [-program opts]
```

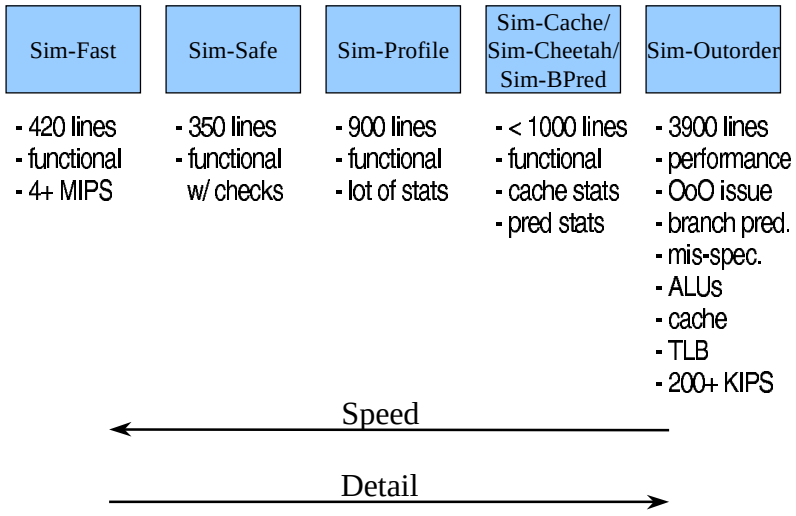
- debugging a program with DLite!

```
sim-safe -i [-sim opts] program [-program opts]
```

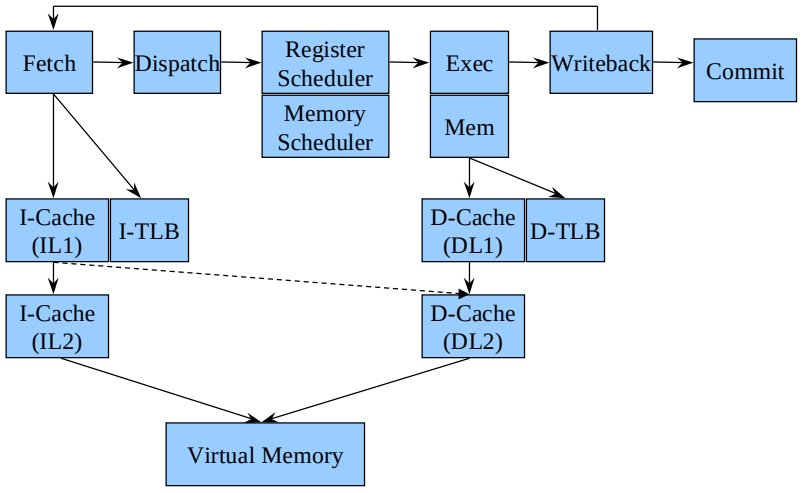
- disassembling a program

```
ssbig-na-sstrix-objdump -x -d -l program
```

Simulation Suite Overview

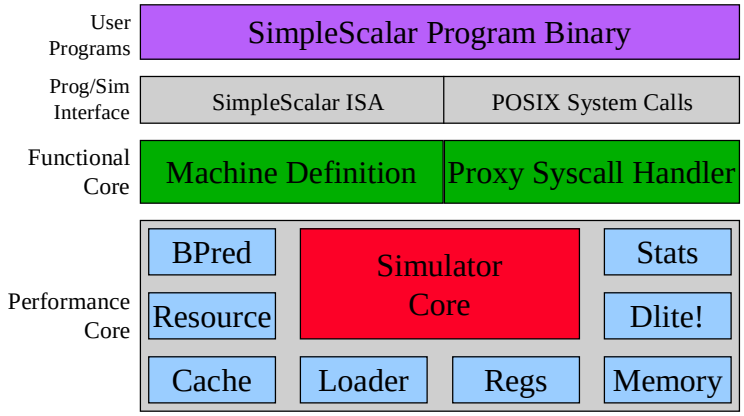


SIM-OUTORDER: Simulated Microarchitecture



- implemented in `sim-outorder.c` and components

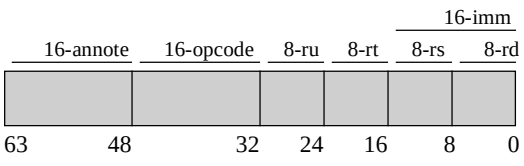
Simulator S/W Architecture



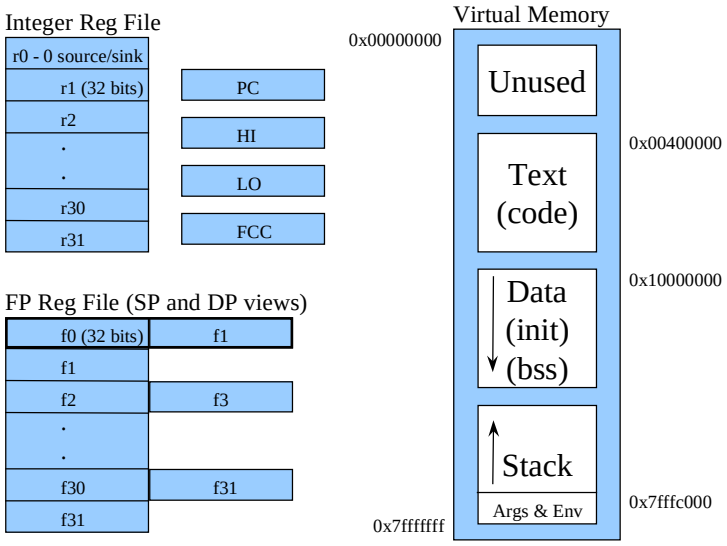
- most of performance core is optional
- most projects will enhance the "simulator core"

The SimpleScalar Instruction Set

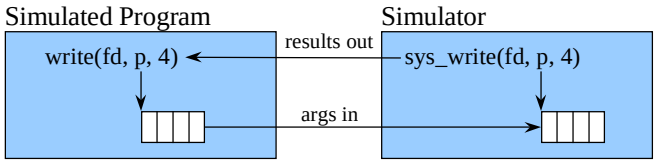
- clean and simple instruction set architecture
  - MIPS/DLX + more addressing modes - delay slots
- bi-endian instruction set definition
  - facilitates portability, build to match host endian
- 64-bit encoding facilitates instruction set research
  - 16-bit space for hints, new instructions, and annotations
  - four operand instruction format, up to 256 registers



SimpleScalar Architected State



Simulator I/O via Proxy System Calls



- a useful simulator must implement some form of I/O
  - I/O via SYSCALL instruction
  - subset of Ultrix system calls, proxied out to host
- basic algorithm
  - decode system call
  - copy arguments (if any) into simulator memory
  - perform system call on host
  - copy results (if any) into simulated program memory

Initial Course Experiences

- we were surprised so many instructors used SimpleScalar
  - at the time, the only detailed simulation tool widely available
  - portable code, could be built for most common platforms
  - simple transition from course project to research project
- the tool set was used
  - for "test drives" for undergraduate introductory courses
    - to introduce students to simulation tools and experiments
  - as a foundation for student projects in advanced graduate courses
    - students pick appropriate simulator for project baseline
- problems encountered (as evidenced by our mailboxes!):
  - insufficient internal documentation
  - too difficult to install

Release 2.0 Enhancements

- many improvements to the internal documentation
  - code base grew by 40% due to comments added!
  - Hacker's Guide written
  - second technical report includes details about S/W architecture
- install process streamlined
  - SPEC'95 benchmark binary release made (with permission)
  - 8 new host ports added (including Windows NT)
  - self-hosting test suite added
- other enhancements
  - pipeline visualization tools (pipe traces)
  - DLite! debugger, program profiling tools, more simulators, …

Future Enhancements

- still working towards the ultimate "out-of-box" experience
  - more host ports being contributed by users
  - EIO (external I/O) traces
    - a single file captures entire experiment, including code, data, arguments and external I/O
    - a technology-friendly trace format
  - SimpleScalar Tutorial
    - many more internal details now documented
    - limitations of the tool set succinctly specified
- improving the applicability of the tool set
  - parallel system simulation support

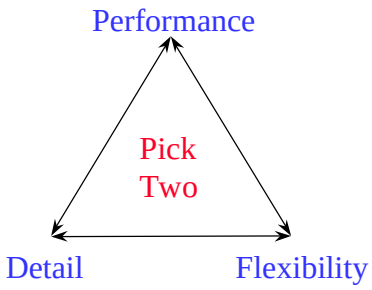
Summary and Contact Information

- uniprocessor computer architecture test bed
  - easy to install for most platforms, with pre-packaged experiments
  - well documented, user's and hacker's guides available
  - broad simulation suite applicable to research projects as well
- role in instruction
  - students in introductory courses can "test drive" the simulators
  - advanced students can base their projects off existing infrastructure
- freely available with source and docs from UW-Madison

  `http://www.cs.wisc.edu/~mscalar/simplescalar.html`

Backups

The Zen of Simulator Design



Performance: speeds design cycle

Flexibility: maximizes design scope

Detail: minimizes risk

- *design goals* will drive which aspects are optimized
- the SimpleScalar Tool Set
  - optimizes performance and flexibility
  - in addition, provides portability and varied detail

Primary Advantages

- extensible
  - source included for everything: compiler, libraries, simulators
  - widely encoded, user-extensible instruction format
- portable
  - at the host, virtual target runs on most Unix-like boxes
  - at the target, simulators can support multiple ISA's
- detailed
  - execution-driven simulators
  - supports wrong path exec, control and data speculation, etc...
  - many sample simulators included
- performance (on P6-200)
  - Sim-Fast: 4+ MIPS, Sim-OutOrder: 150+ KIPS

SimpleScalar Instructions

Control:

j - jump
jal - jump and link
jr - jump register
jalr - jump and link register
beq - branch == 0
bne - branch != 0
blez - branch <= 0
bgtz - branch > 0
bltz - branch < 0
bgez - branch >= 0
bct - branch FCC TRUE
bcf - branch FCC FALSE

Load/Store:

lb - load byte
lbu - load byte unsigned
lh - load half (short)
lhu - load half (short) unsigned
lw - load word
dlw - load double word
l.s - load single-precision FP
l.d - load double-precision FP
sb - store byte
sbu - store byte unsigned
sh - store half (short)
shu - store half (short) unsigned
sw - store word
dsw - store double word
s.s - store single-precision FP
s.d - store double-precision FP

addressing modes:
(C)
(reg + C) (w/ pre/post inc/dec)
(reg + reg) (w/ pre/post inc/dec)

Integer Arithmetic:

add - integer add
addu - integer add unsigned
sub - integer subtract
subu - integer subtract unsigned
mult - integer multiply
multu - integer multiply unsigned
div - integer divide
divu - integer divide unsigned
and - logical AND
or - logical OR
xor - logical XOR
nor - logical NOR
sll - shift left logical
srl - shift right logical
sra - shift right arithmetic
slt - set less than
sltu - set less than unsigned

SimpleScalar Instructions

Floating Point Arithmetic:

add.s - single-precision add
add.d - double-precision add
sub.s - single-precision subtract
sub.d - double-precision subtract
mult.s - single-precision multiply
mult.d - double-precision multiply
div.s - single-precision divide
div.d - double-precision divide
abs.s - single-precision absolute value
abs.d - double-precision absolute value
neg.s - single-precision negation
neg.d - double-precision negation
sqrt.s - single-precision square root
sqrt.d - double-precision square root
cvt - integer, single, double conversion
c.s - single-precision compare
c.d - double-precision compare

Miscellaneous:

nop - no operation
syscall - system call
break - declare program error

Simulator S/W Architecture

- interface programming style
  - all ".c" files have an accompanying ".h" file with same base
  - ".h" files define public interfaces "exported" by module
    - mostly stable, documented with comments, studying these files
  - ".c" files implement the exported interfaces
    - not as stable, study these if you need to hack the functionality
- simulator modules
  - sim-*.c files, each implements a complete simulator core
- reusable S/W components facilitate "rolling your own"
  - system components
  - simulation components
  - "really useful" components

SIM-OUTORDER Pipetraces

- produces detailed history of all insts executed, including:
  - instruction fetch, retirement. and pipeline stage transitions
  - supported by sim-outorder
  - enabled via the "-ptrace" option: `-ptrace <file> <range>`
  - useful for pipeline visualization, micro-validation, debugging
- example usage:

`-ptrace FOO.trc`	- trace everything to file FOO.trc
`-ptrace BAR.trc 100:5000`	- trace from inst 100 to 5000
`-ptrace UXXE.trc :10000`	- trace until instruction 10000

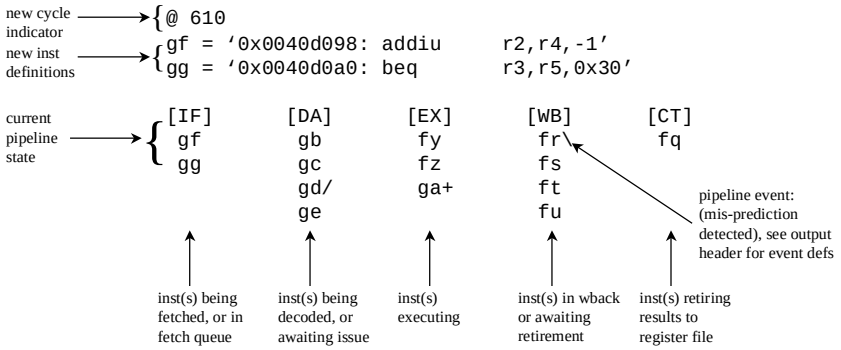
- view with the pipeview.pl Perl script
  - it displays the pipeline for each cycle of execution traced
  - usage: `pipeview.pl <ptrace_file>`

Displaying Pipetraces

- example session:

```
sim-outorder -ptrace FOO.trc :1000 test-math
pipeview.pl FOO.trc
```

- example output:



PC-Based Statistical Profiles

- produces a text segment profile for any integer statistical counter
  - supported on sim-cache, sim-profile, and sim-outorder
  - specify counter to be monitored using "-pcstat" option
    - e.g., `-pcstat sim_num_insn`
- example applications:

`-pcstat sim_num_insn` - execution profile
`-pcstat sim_num_refs` - reference profile
`-pcstat il1.misses` - L1 I-cache miss profile
`-pcstat bpred_bimod.misses` - branch pred miss profile

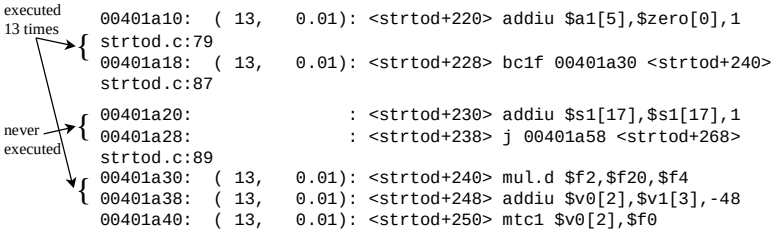
- view with the textprof.pl Perl script
  - it displays pc-based statistics with program disassembly
  - usage: `textprof.pl <dis_file> <sim_output> <stat_name>`

PC-Based Statistical Profiles (cont.)

- example session:

```
sim-profile -pcstat sim_num_insn test-math >&! test-math.out
objdump -dl test-math >! test-math.dis
textprof.pl test-math.dis test-math.out sim_num_insn_by_pc
```

- example output:




```
00401a10:  ( 13,   0.01): <strtod+220> addiu $a1[5],$zero[0],1
strtod.c:79
00401a18:  ( 13,   0.01): <strtod+228> bc1f 00401a30 <strtod+240>
strtod.c:87
00401a20:              : <strtod+230> addiu $s1[17],$s1[17],1
00401a28:              : <strtod+238> j 00401a58 <strtod+268>
strtod.c:89
00401a30:  ( 13,   0.01): <strtod+240> mul.d $f2,$f20,$f4
00401a38:  ( 13,   0.01): <strtod+248> addiu $v0[2],$v1[3],-48
00401a40:  ( 13,   0.01): <strtod+250> mtc1 $v0[2],$f0
```

- works on any integer counter registered with the stats package, including those added by users!

DLite!, the Lite Debugger

- a very lightweight symbolic debugger
- supported by all simulators (except sim-fast)
- designed for easily integration into new simulators
  - requires addition of only four function calls (see `dlite.h`)
- to use DLite!, start simulator with "`-i`" option
  - use the "`help`" command for complete documentation
- program symbols and expressions may be used in most contexts
  - e.g., "`break main+8`"

DLite! Commands

- main features:
  - `break`, `dbreak`, `rbreak`:
    - set text, data, and range breakpoints
  - `regs`, `iregs`, `fregs`:
    - display all, integer, and FP register state
  - `dump <addr> <count>`:
    - dump `<count>` bytes of memory at `<addr>`
  - `dis <addr> <count>`:
    - disassemble `<count>` insts starting at `<addr>`
  - `print <expr>`, `display <expr>`:
    - display expression or memory
  - `mstate`: display machine-specific state
    - `mstate` alone displays options, if any

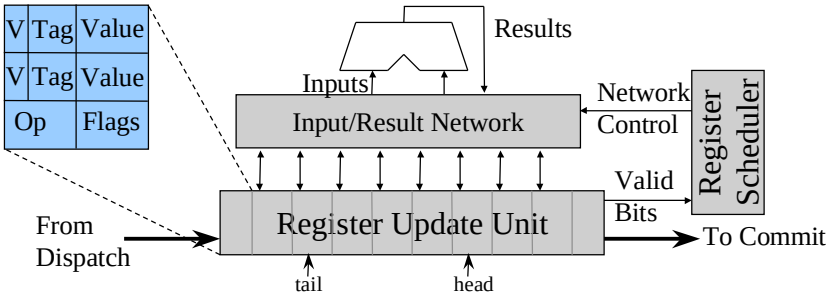
DLite!, Breakpoints and Expressions

- breakpoints:
  - code:
    - `break <addr>`, e.g., `break main`, `break 0x400148`
  - data:
    - `dbreak <addr> {r|w|x}`
    - r = read, w = write, x = execute, e.g., `dbreak stdin w`, `dbreak sys_count wr`
  - range:
    - `rbreak <range>`, e.g., `rbreak @main:+279`, `rbreak 2000:3500`
- DLite! expressions, may include:
  - operators: +, -, /, *
  - literals: 10, 0xff, 077
  - symbols: main, vfprintf
  - registers: e.g., $r1, $f4, $pc, $lo

Annotating SimpleScalar Instructions

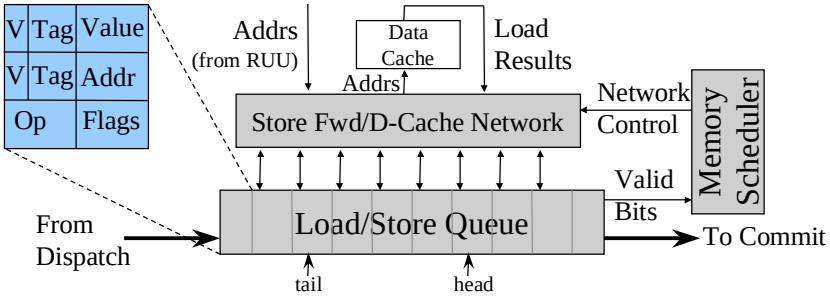
- useful for adding
  - hints, new instructions, text markers, etc...
  - no need to hack the assembler
- bit annotations:
  - /a - /p, set bit 0 - 15
  - e.g., `ld/a $r6,4($r7)`
- field annotations:
  - /s:e(v), set bits s->e with value v
  - e.g., `ld/6:4(7) $r6,4($r7)`

The Register Update Unit (RUU)



- RUU handles register synchronization/communication
  - unifies reorder buffer and reservation stations
    - managed as a circular queue
    - entries allocated at Dispatch, deallocated at Commit
  - out-of-order issue, when register and memory deps satisfied
    - memory dependencies resolved by load/store queue (LSQ)

The Load/Store Queue (LSQ)



- LSQ handles memory synchronization/communication
  - contains all loads and stores in program order
    - load/store primitives really, address calculation is separate op
    - effective address calculations reside in RUU (as ADD insts)
  - loads issue out-of-order, when memory deps known satisfied
    - load addr known, source data identified, no unknown store address

Machine Definition File (ss.def)

- a single file describes all aspects of the architecture
  - used to generate decoders, dependency analyzers, functional components, disassemblers, appendices, etc.
  - e.g., machine definition + ~30 line main = functional sim
  - generates fast and reliable codes with minimum effort
- instruction definition example: