M
R
L

A User's and Hacker's Guide to the SimpleScalar Architectural Research Tool Set

(for tool set release 2.0)

Todd M. Austin
taustin@ichips.intel.com
Intel MicroComputer Research Labs

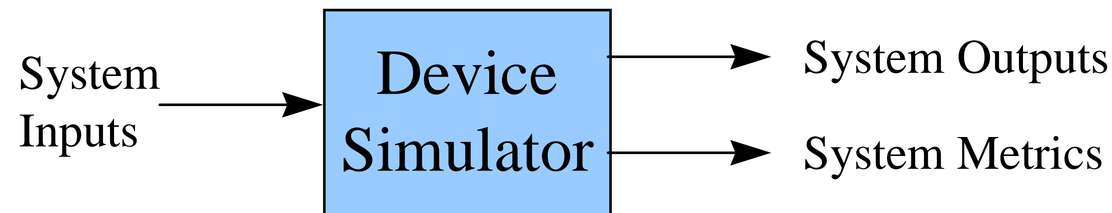
January, 1997

Tutorial Overview

- Computer Architecture Simulation Primer
- SimpleScalar Tool Set
  - Overview
  - User's Guide
- SimpleScalar Instruction Set Architecture
- Out-of-Order Issue Simulator
  - Model Microarchitecture
  - Implementation Details
- Hacking SimpleScalar
- Looking Ahead

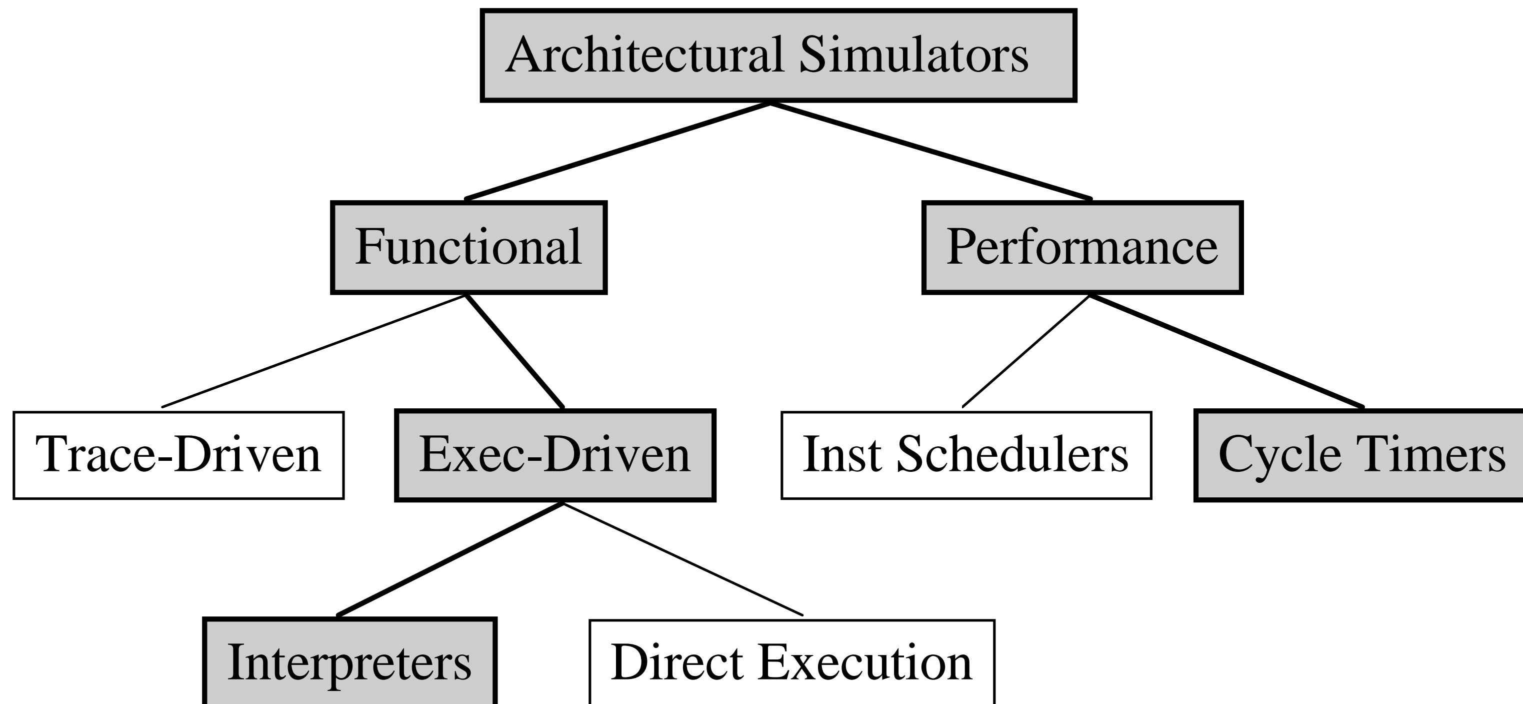
A Computer Architecture Simulator Primer

- What is an architectural simulator?
  - a tool that reproduces the behavior of a computing device



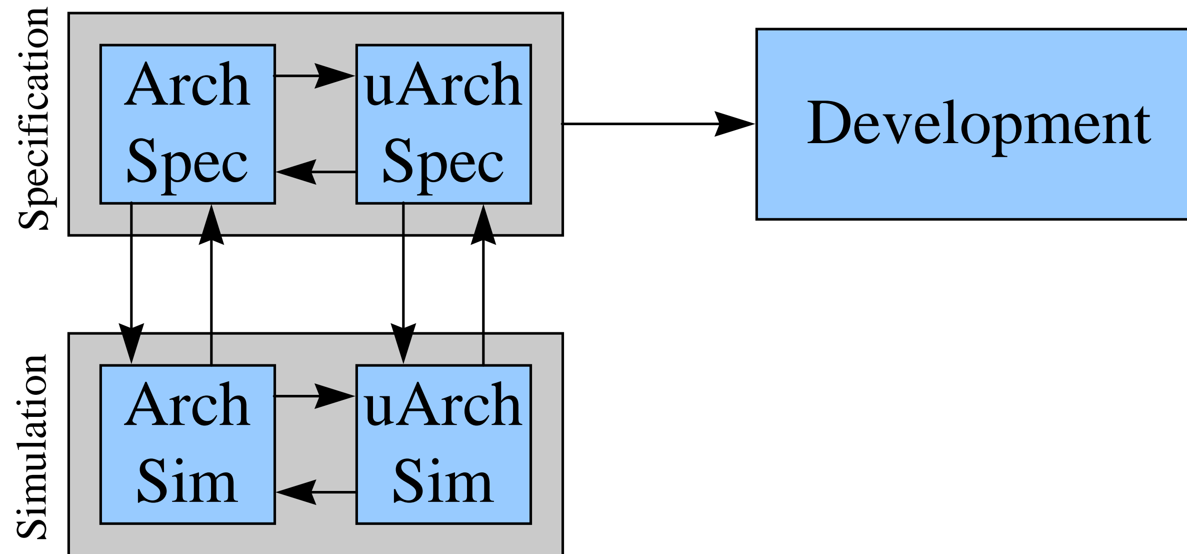
- Why use a simulator?
  - leverage faster, more flexible S/W development cycle
    - permits more design space exploration
    - facilitates validation before H/W becomes available
    - level of abstraction can be throttled to design task
    - possible to increase/improve system instrumentation

A Taxonomy of Simulation Tools



- shaded tools are included in the SimpleScalar tool set

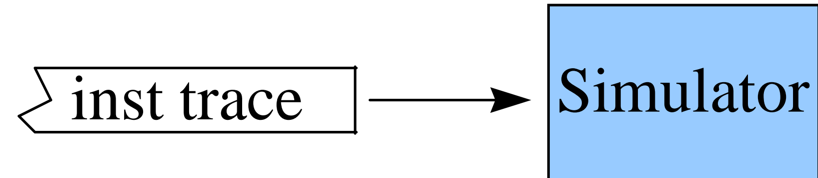
Functional vs. Performance Simulators



- functional simulators implement the architecture
  - the architecture is what programmer's see
- performance simulators implement the microarchitecture
  - model system internals (microarchitecture)
  - often concerned with time

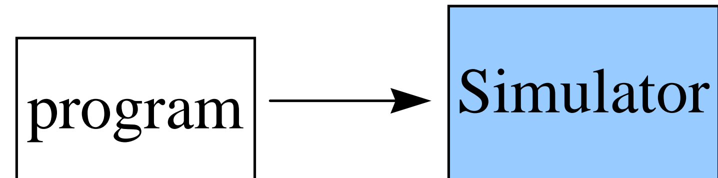
Execution- vs. Trace-Driven Simulation

- trace-based simulation:



  - simulator reads a "trace" of inst captured during a previous execution
  - easiest to implement, no functional component needed

- execution-driven simulation:

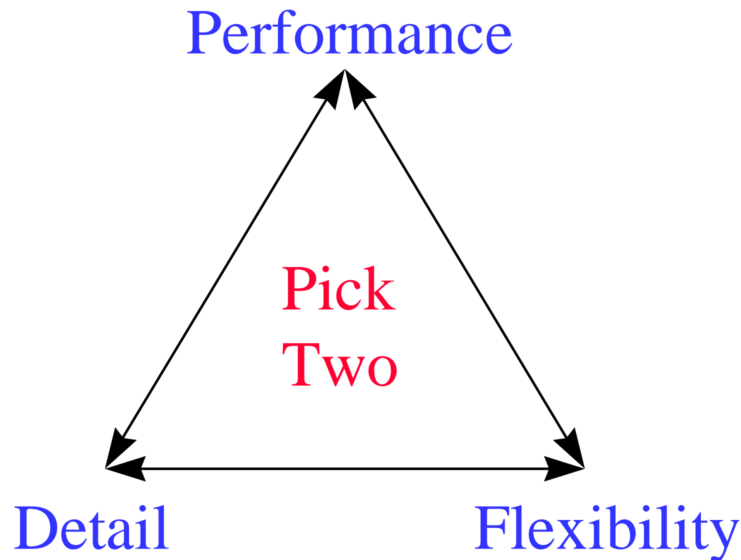


  - simulator "runs" the program, generating a trace on-the-fly
  - more difficult to implement, but has many advantages
  - direct-execution: instrumented program runs on host

Instruction Schedulers vs. Cycle Timers

- constraint-based instruction schedulers
  - simulator schedules instructions into execution graph based on availability of microarchitecture resources
  - instructions are handled one-at-a-time and in order
  - simpler to modify, but usually less detailed
- cycle-timer simulators
  - simulator tracks microarchitecture state for each cycle
  - many instructions may be “in flight” at any time
  - simulator state == state of the microarchitecture
  - perfect for detailed microarchitecture simulation, simulator faithfully tracks microarchitecture function

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 Architectural Research Tool Set
  - optimizes performance and flexibility
  - in addition, provides portability and varied detail

Tutorial Overview

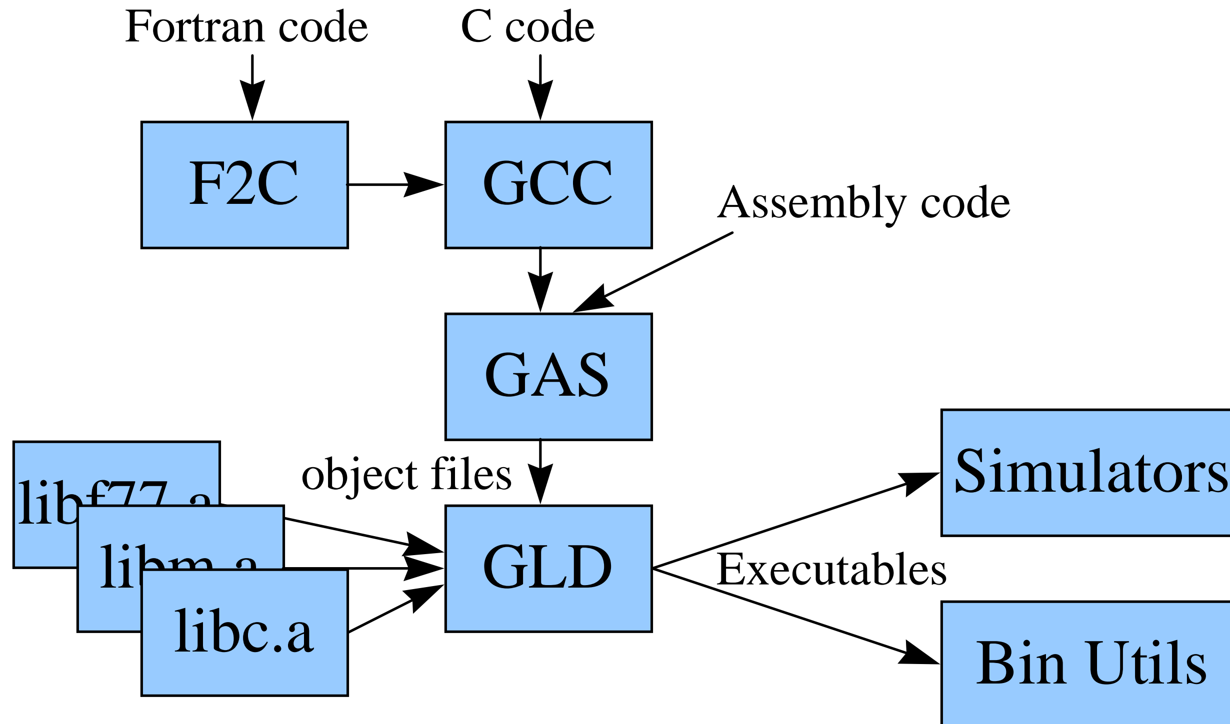
- Computer Architecture Simulation Primer
- SimpleScalar Tool Set
  - **Overview**
  - User's Guide
- SimpleScalar Instruction Set Architecture
- Out-of-Order Issue Simulator
  - Model Microarchitecture
  - Implementation Details
- Hacking SimpleScalar
- Looking Ahead

The SimpleScalar Tool Set

- computer architecture research test bed
  - compilers, assembler, linker, libraries, and simulators
  - targeted to the virtual SimpleScalar architecture
  - hosted on most any Unix-like machine
- developed during my dissertation work at UW-Madison
  - third generation simulation system (Sohi → Franklin → Austin)
  - 2.5 years to develop this incarnation
  - first public release in July ‘96, made with Doug Burger
  - second public release in January ‘97
- freely available with source and docs from UW-Madison

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

SimpleScalar Tool Set Overview



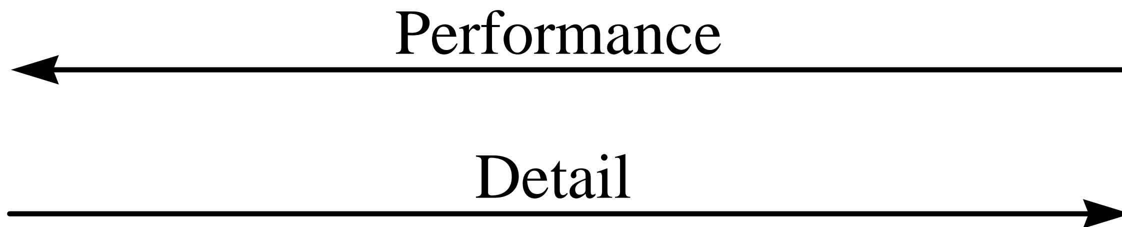
- compiler chain is GNU tools ported to SimpleScalar
- Fortran codes are compiled with AT&T's *f2c*
- libraries are GLIBC ported to SimpleScalar

Primary Advantages

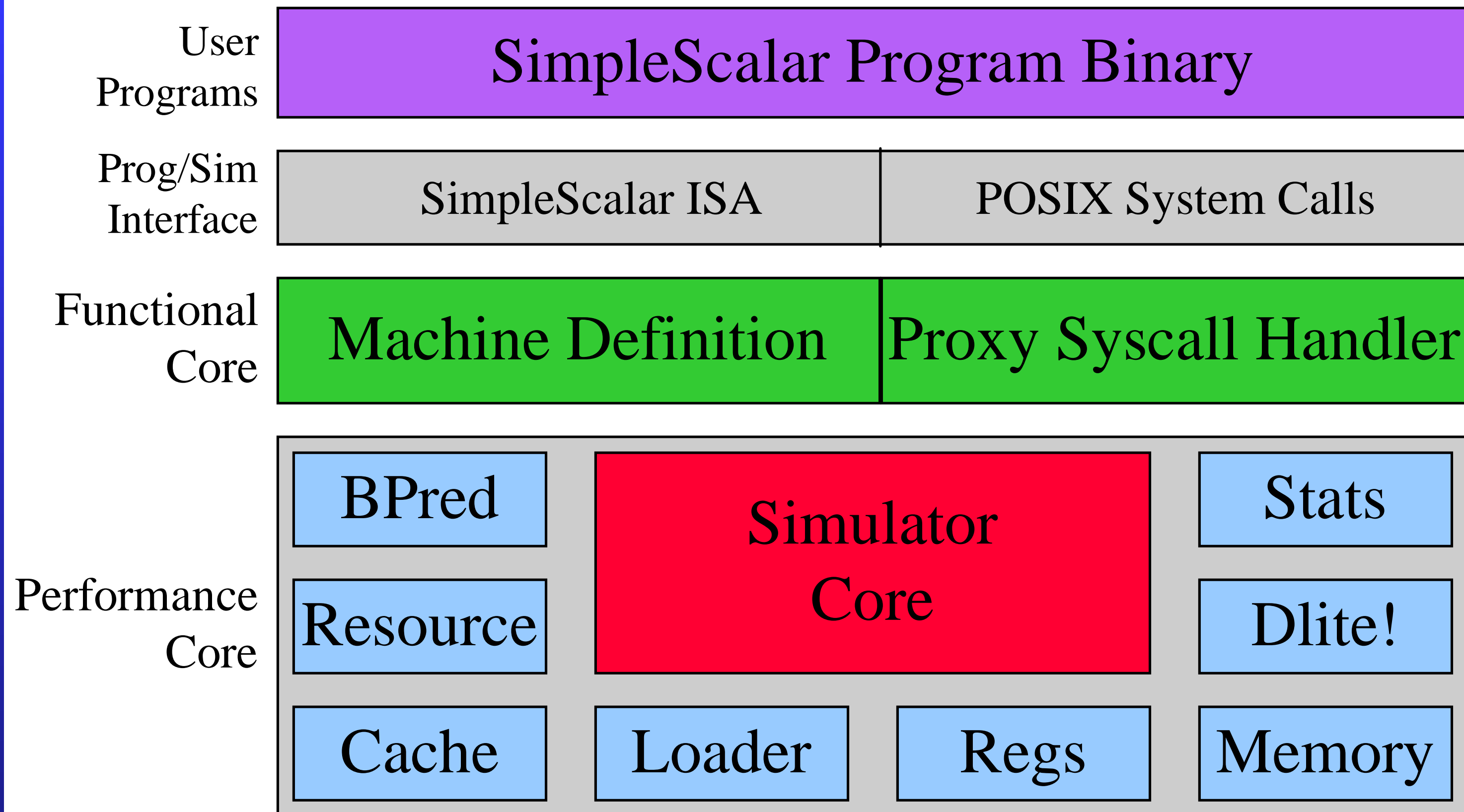
- extensible
  - source included for everything: compiler, libraries, simulators
  - widely encoded, user-extensible instruction format
- portable
  - at the host, virtual target runs on most Unix-like boxes
  - at the target, simulators can support multiple ISA's
- detailed
  - execution driven simulators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

Simulation Suite Overview

Sim-Fast	Sim-Safe	Sim-Profile	Sim-Cache/ Sim-Cheetah	Sim-Outorder
- 420 lines - functional - 4+ MIPS	- 350 lines - functional w/ checks	- 900 lines - functional - lot of stats	- < 1000 lines - functional - cache stats	- 3900 lines - performance - OoO issue - branch pred. - mis-spec. - ALUs - cache - TLB - 200+ KIPS



Simulator Structure



- modular components facilitate “rolling your own”
- performance core is optional

Tutorial Overview

- Computer Architecture Simulation Primer
- SimpleScalar Tool Set
  - Overview
  - **User's Guide**
- SimpleScalar Instruction Set Architecture
- Out-of-Order Issue Simulator
  - Model Microarchitecture
  - Implementation Details
- Hacking SimpleScalar
- Looking Ahead

Generating SimpleScalar Binaries

- compiling a C program, e.g.,

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

- compiling a Fortran program, e.g.,

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

- compiling a SimpleScalar assembly program, e.g.,

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

- running a program, e.g.,

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

- disassembling a program, e.g.,

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

- building a library, use:

```
ssbig-na-sstrix-{ar,ranlib}
```

Global Simulator Options

- supported on all simulators:

`-h`	- print simulator help message
`-d`	- enable debug message
`-i`	- start up in DLite! debugger
`-q`	- terminate immediately (use with `-dumpconfig`)
`-config <file>`	- read configuration parameters from `<file>`
`-dumpconfig <file>`	- save configuration parameters into `<file>`

- configuration files:
  - to generate a configuration file:
    - specify non-default options on command line
    - and, include “`-dumpconfig <file>`” to generate configuration file
  - comments allowed in configuration files:
    - text after “`#`” ignored until end of line
  - reload configuration files using “`-config <file>`”
  - config files may reference other configuration files

DLite!, the Lite Debugger

- a very lightweight symbolic debugger
- supported by all simulators (except sim-fast)
- designed for easily integration into SimpleScalar simulators
  - requires addition of only four function calls (see `dlite.h`)
- to use DLite!, start simulator with "`-i`" option (interactive)
- program symbols and expressions may be used in most contexts
  - e.g., "`break main+8`"
- use the "`help`" command for complete documentation
- main features:
  - `break`, `dbreak`, `rbreak`: set text, data, and range breakpoints
  - `regs`, `iregs`, `fregs`: display all, int, 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

Execution Ranges

- specify a range of addresses, instructions, or cycles
- used by range breakpoints and pipetracer (in sim-outorder)
- format:

    address range: @<start>:<end>
    instruction range: <start>:<end>
    cycle range: #<start>:<end>

- the end range may be specified relative to the start range
- both endpoints are optional, and if omitted the value will default to the largest/smallest allowed value in that range
- e.g.,
    - @main:+278 - main to main+278
    - #:1000 - cycle 0 to cycle 1000
    - : - entire execution (instruction 0 to end)

Sim-Safe: Functional Simulator

- the minimal SimpleScalar simulator
- no other options supported

Sim-Fast: Fast Functional Simulator

- an optimized version of sim-safe
- DLite! is not supported on this simulator
- no other options supported

Sim-Outorder Pipetraces

- produces detailed history of all instructions executed, including:
  - instruction fetch, retirement. and stage transitions
- supported in sim-outorder
- use the "-ptrace" option to generate a pipetrace
  - `-ptrace <file> <range>`
- example usage:

`-pcstat FOO.trc :`	- trace entire execution to FOO.trc
`-pcstat BAR.trc 100:5000`	- trace from inst 100 to 5000
`-pcstat UXXE.trc :10000`	- trace until instruction 10000

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

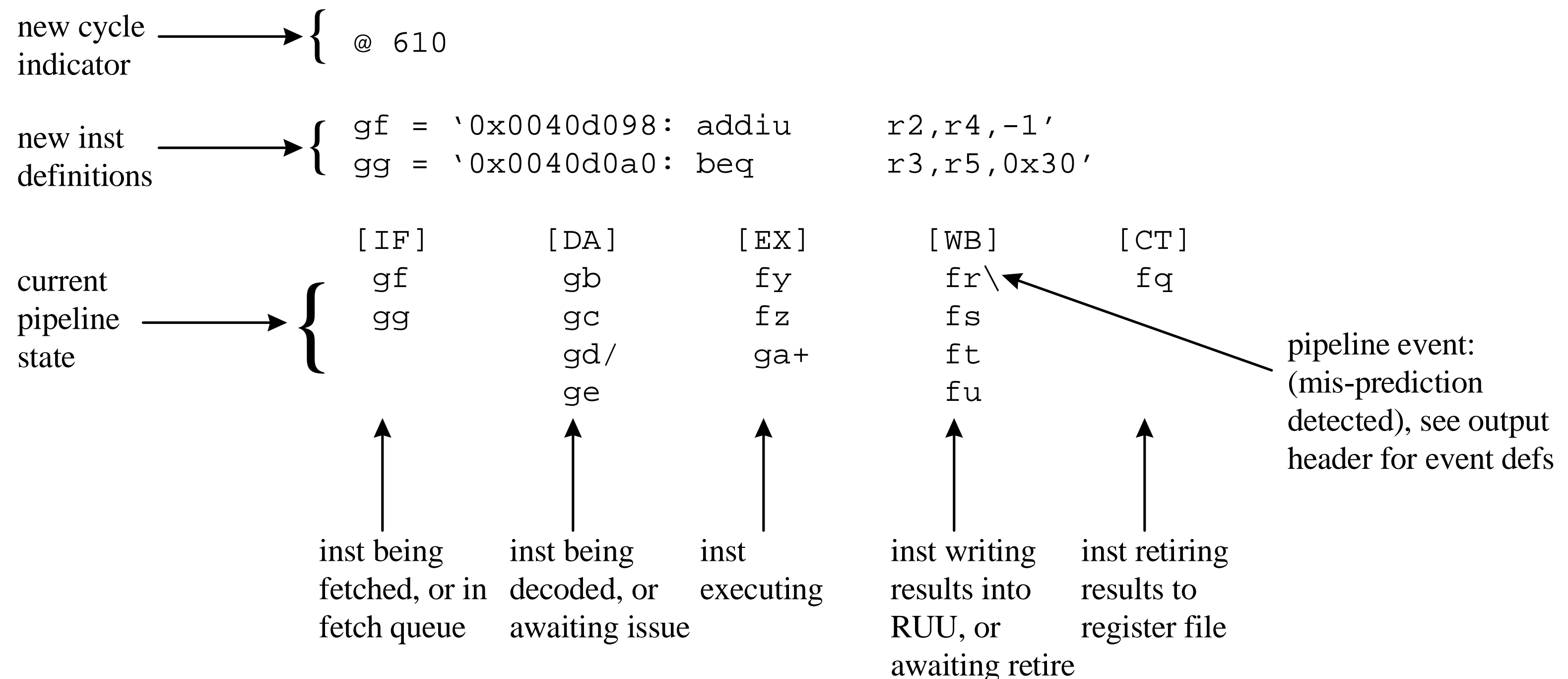
```
pipeview.pl <ptrace_file>
```

Sim-Outorder Pipetraces (cont.)

- example usage:

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

- example output:

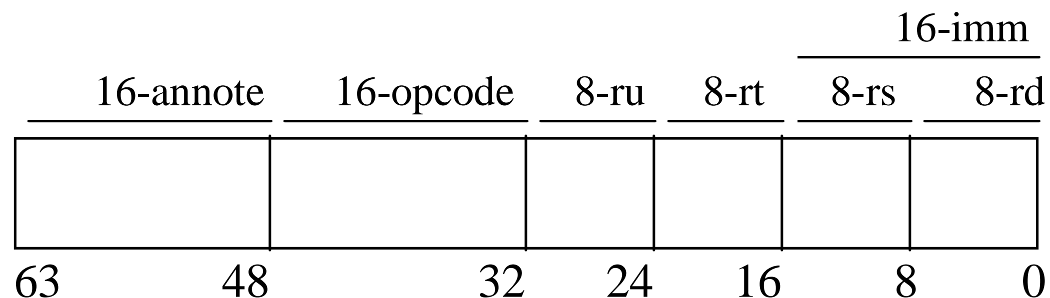


Tutorial Overview

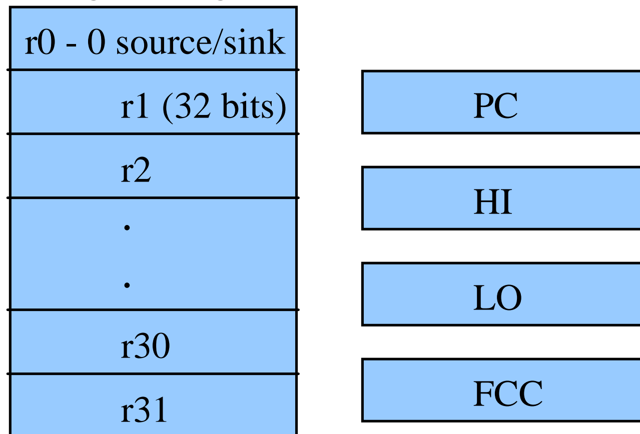
- Computer Architecture Simulation Primer
- SimpleScalar Tool Set
  - Overview
  - User's Guide
- **SimpleScalar Instruction Set Architecture**
- Out-of-Order Issue Simulator
  - Model Microarchitecture
  - Implementation Details
- Hacking SimpleScalar
- Looking Ahead

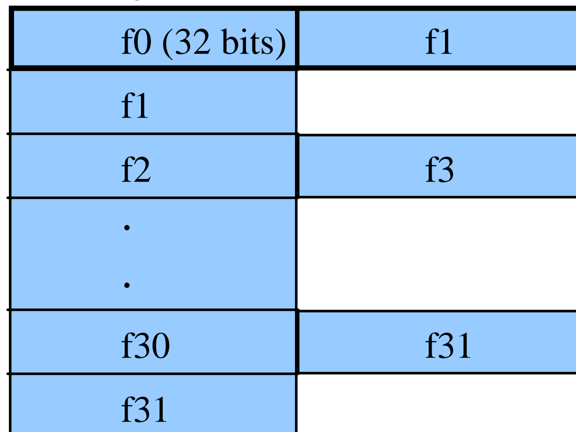
The SimpleScalar Instruction Set

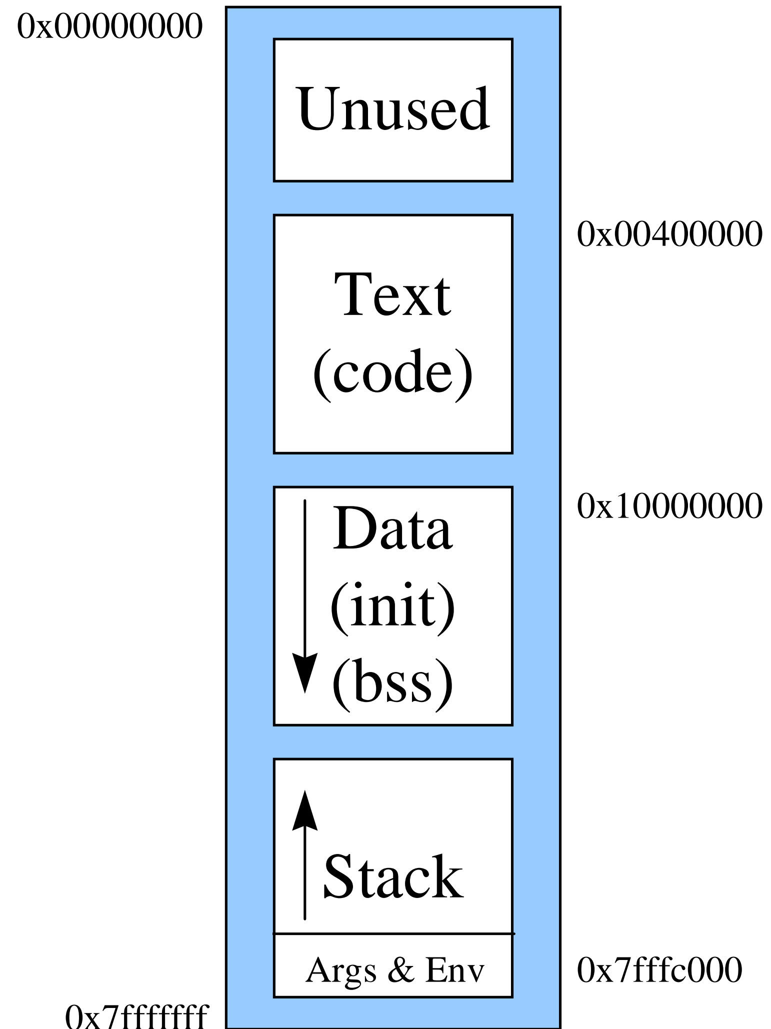
- clean and simple instruction set architecture:
  - MIPS/DLX + more addressing modes - delay slots
- bi-endian instruction set definition
  - facilitates portability, build to match host endian
- 64-bit inst encoding facilitates instruction set research
  - 16-bit space for hints, new insts, and annotations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



SimpleScalar Architected State







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

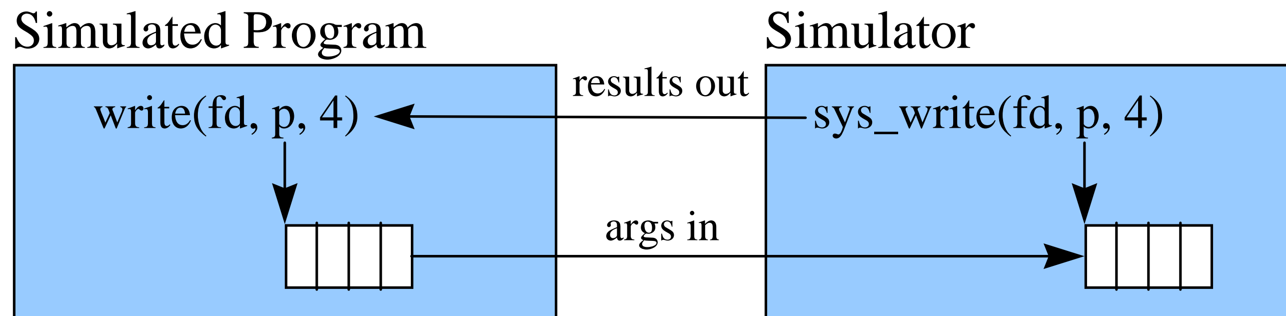
Miscellaneous:

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

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)`

Proxy System Call Handler

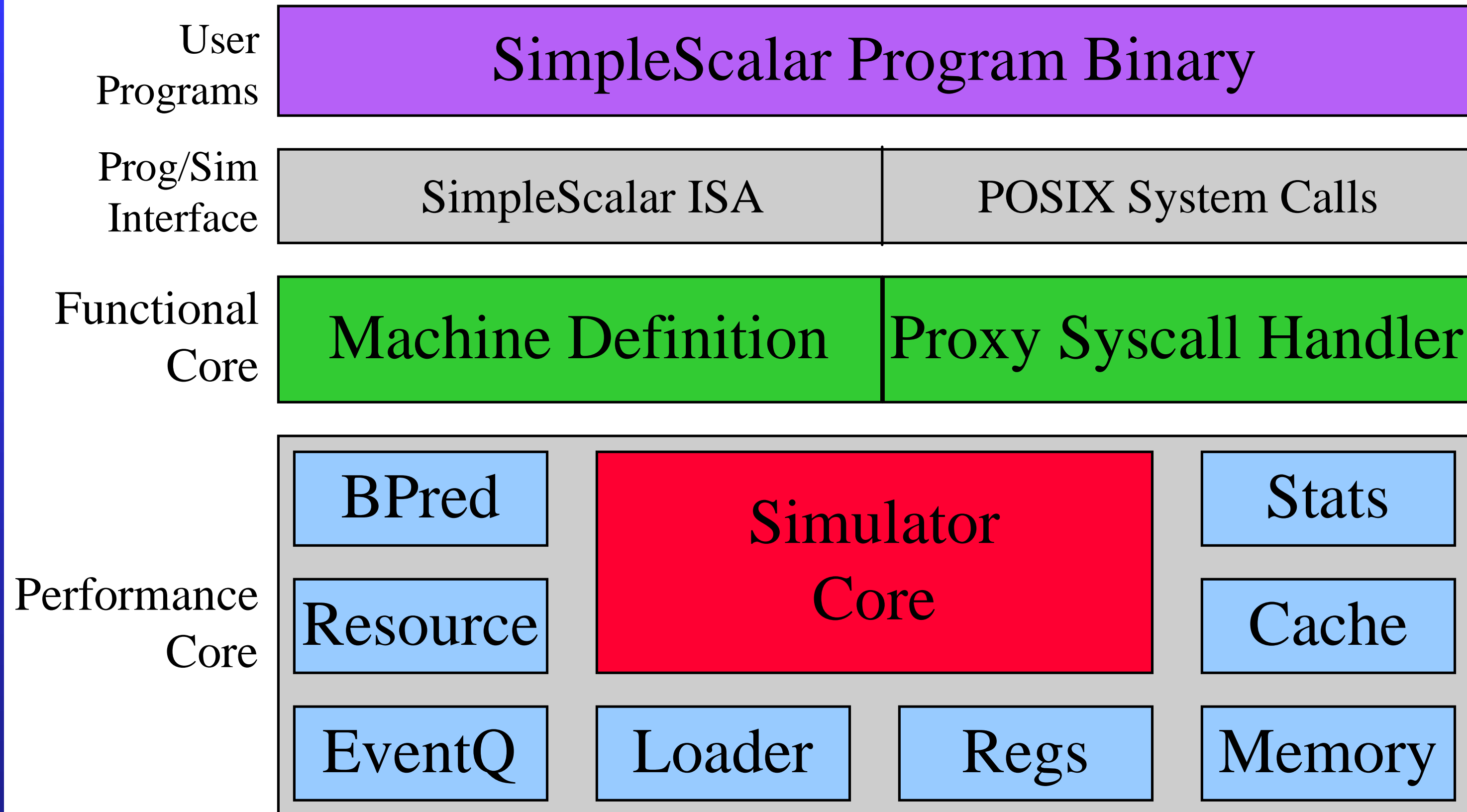


- syscall.c implements a subset of Ultrix Unix system calls
- basic algorithm:
  - decode system call
  - copy arguments (if any) into simulator memory
  - make system call
  - copy results (if any) into simulated program memory

Tutorial Overview

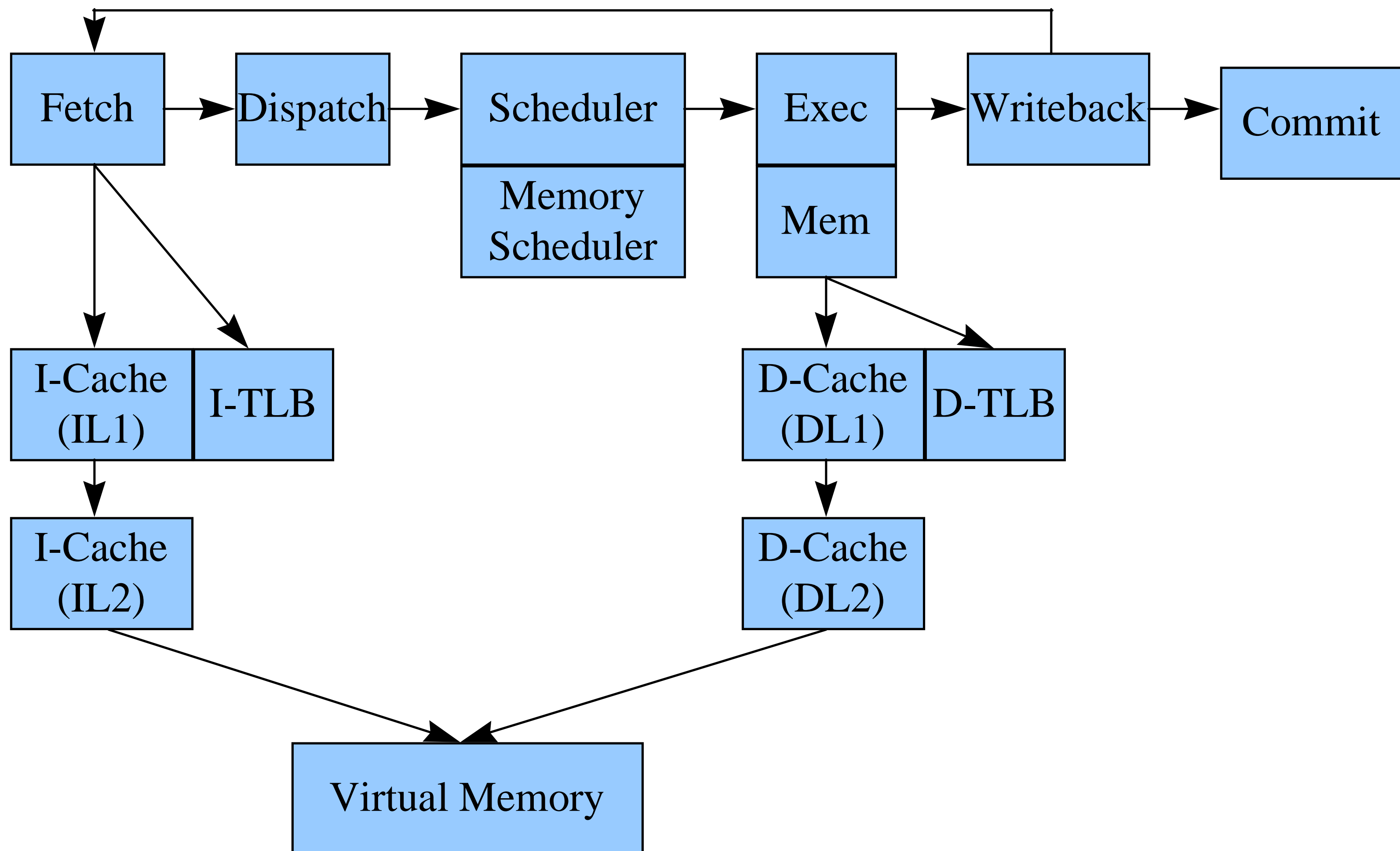
- Computer Architecture Simulation Primer
- SimpleScalar Tool Set
  - Overview
  - User's Guide
- SimpleScalar Instruction Set Architecture
- Out-of-Order Issue Simulator
  - **Model Microarchitecture**
  - Implementation Details
- Hacking SimpleScalar
- Looking Ahead

Simulator Structure



- modular components facilitate “rolling your own”
- performance core is optional

Out-of-Order Issue Simulator



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

Tutorial Overview

- Computer Architecture Simulation Primer
- SimpleScalar Tool Set
  - Overview
  - User's Guide
- SimpleScalar Instruction Set Architecture
- Out-of-Order Issue Simulator
  - Model Microarchitecture
  - **Implementation Details**
- Hacking SimpleScalar
- Looking Ahead

Tutorial Overview

- Computer Architecture Simulation Primer
- SimpleScalar Tool Set
  - Overview
  - User's Guide
- SimpleScalar Instruction Set Architecture
- Out-of-Order Issue Simulator
  - Model Microarchitecture
  - Implementation Details
- **Hacking SimpleScalar**
- Looking Ahead

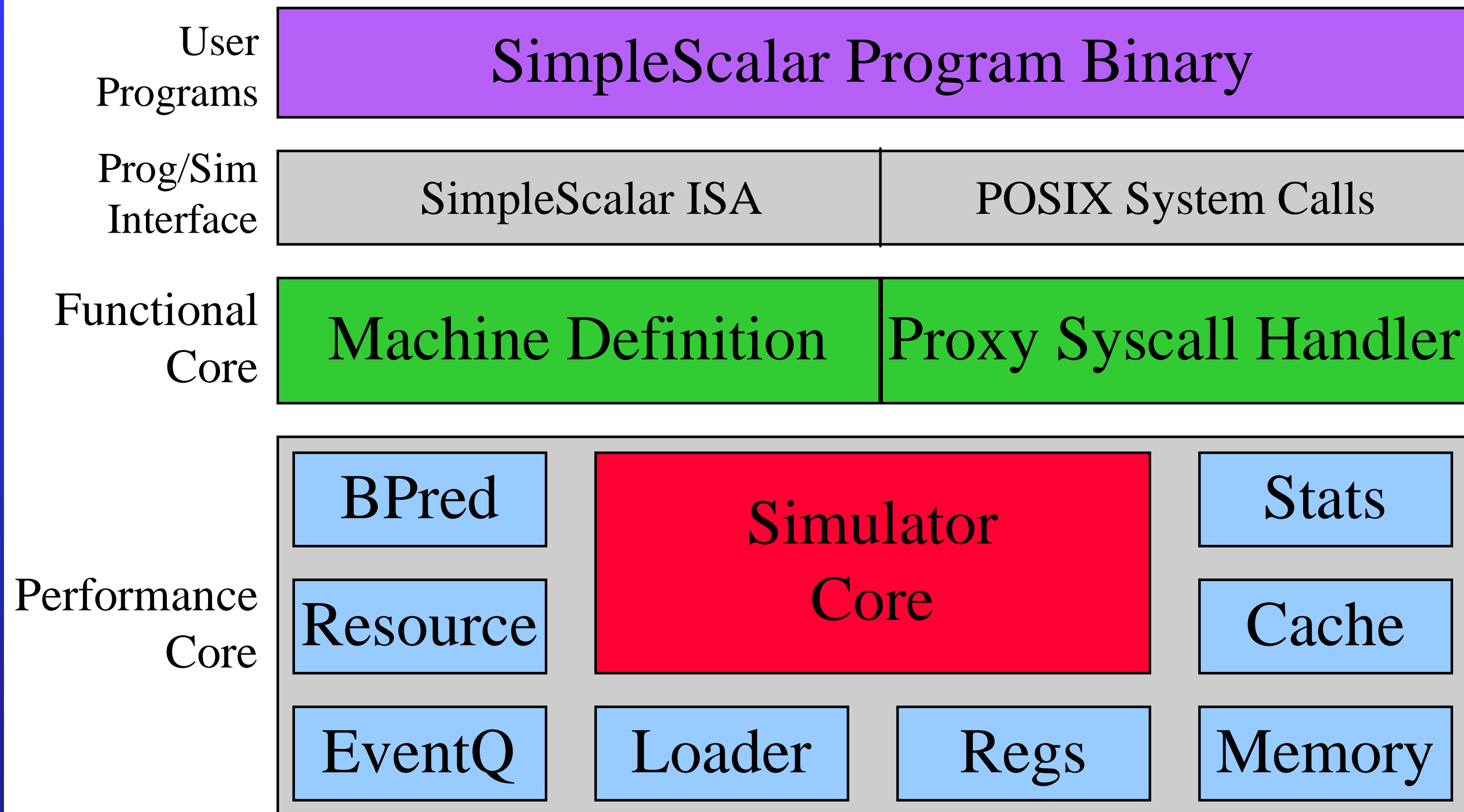
Hacker's Guide

- source code design philosophy:
  - infrastructure facilitates "rolling your own"
    - standard simulator interfaces
    - large component library, e.g., caches, loaders, etc...
  - performance and flexibility before clarity
- section organization:
  - compiler chain hacking
  - simulator hacking

Hacking the SimpleScalar Simulators

- two options:
  - leverage existing simulators (sim-*.c)
    - they are stable
    - very little instrumentation has been added to keep the source clean
  - roll your own
    - leverage the existing simulation infrastructure, i.e., all the files that do not start with 'sim-'
    - consider contributing useful tools to the source base
- for documentation, read interface documentation in ".h" files

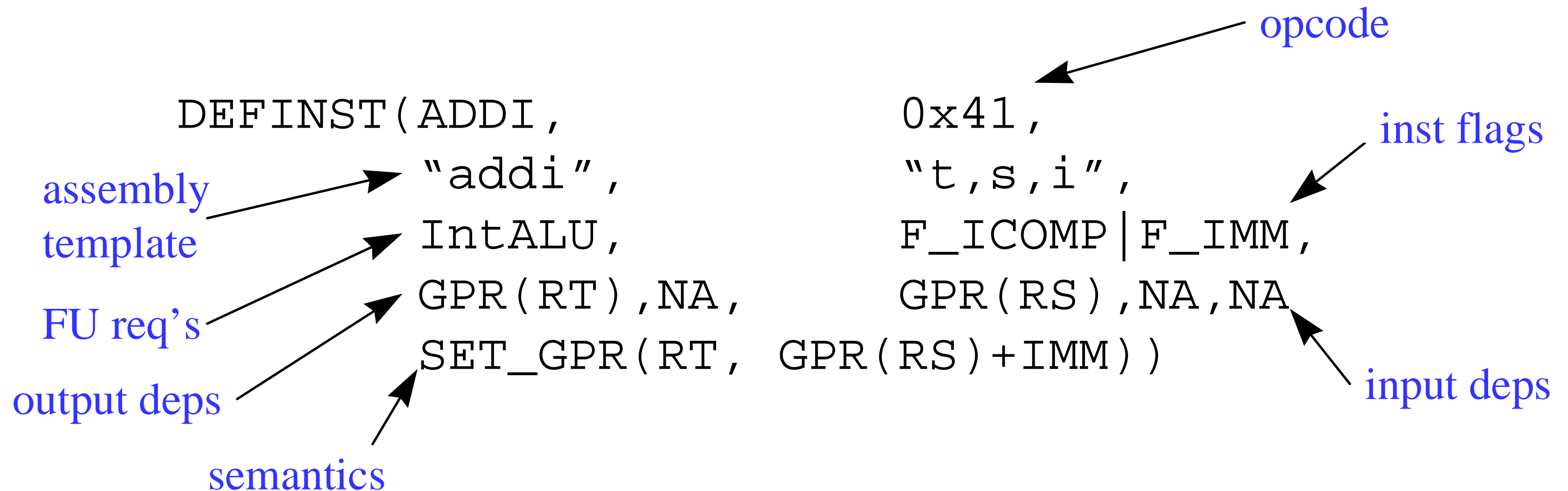
Simulator Structure



- modular components facilitate “rolling your own”
- performance core is optional

Machine Definition

- a single file describes all aspects of the architecture
  - used to generate decoders, dependency analyzers, functional components, disassemblers, appendices, etc.
  - e.g., machine definition + 10 line main == functional sim
  - generates fast and reliable codes with minimum effort
- instruction definition example:



Crafting a Functional Component

```
#define GPR(N)                    (regs_R[N])
#define SET_GPR(N,EXPR)           (regs_R[N] = (EXPR))
#define READ_WORD(SRC, DST)       (mem_read_word((SRC))


switch (SS_OPCODE(inst)) {
#define DEFINST(OP,MSK,NAME,OPFORM,RES,FLAGS,O1,O2,I1,I2,I3,EXPR)  \
      case OP:                                                       \
        EXPR;                                                        \
        break;
#define DEFLINK(OP,MSK,NAME,MASK,SHIFT)                              \
      case OP:                                                       \
        panic("attempted to execute a linking opcode");
#define CONNECT(OP)
#include "ss.def"
#undef DEFINST
#undef DEFLINK
#undef CONNECT
    }
```

Crafting an Decoder

```
#define DEP_GPR(N)              (N)

switch (SS_OPCODE(inst)) {
#define DEFINST(OP,MSK,NAME,OPFORM,RES,CLASS,O1,O2,I1,I2,I3,EXPR)    \
      case OP:                                                      \
        out1 = DEP_##O1; out2 = DEP_##O2;                           \
        in1 = DEP_##I1; in2 = DEP_##I2; in3 = DEP_##I3;             \
        break;
#define DEFLINK(OP,MSK,NAME,MASK,SHIFT)                              \
      case OP:                                                      \
        /* can speculatively decode a bogus inst */                 \
        op = NOP;                                                   \
        out1 = NA; out2 = NA;                                       \
        in1 = NA; in2 = NA; in3 = NA;                               \
        break;
#define CONNECT(OP)
#include "ss.def"
#undef DEFINST
#undef DEFLINK
#undef CONNECT
      default:
        /* can speculatively decode a bogus inst */
        op = NOP;
        out1 = NA; out2 = NA;
        in1 = NA; in2 = NA; in3 = NA;
    }
```

Options Module (option.[hc])

- options are registers (by type) into an options data base
  - see opt_reg_*() interfaces
- produce a help listing:
  - opt_print_help()
- print current options state:
  - opt_print_options()
- add a header to the help screen:
  - opt_reg_header()
- add notes to an option (printed on help screen):
  - opt_reg_note()

Stats Package (stats.[hc])

- one-stop shopping for statistical counters, expressions, and distributions
- counters are "registered" by type with the stats package:
  - see stat_reg_*() interfaces
  - stat_reg_formula(): register a stat that is an expression of other stats
    - stat_reg_formula(sdb, "ipc", "insts per cycle", "insns/cycles", 0);
- simulator manipulates counters using standard in code, e.g.,

  ```
  stat_num_insn++;
  ```
- stat package prints all statistics (using canonical format)
  - stat_print_stats()
- distributions also supported:
  - stat_reg_dist(): register an array distribution
  - stat_reg_sdist(): register a sparse distribution
  - stat_add_sample(): add a sample to a distribution

Proxy Syscall Handler (syscall.[hc])

- algorithm:
  - decode system call
  - copy arguments (if any) into simulator memory
  - make system call
  - copy results (if any) into simulated program memory
- you'll need to hack this module to:
  - add new system call support
  - port SimpleScalar to an unsupported host OS

Branch Predictors (bpred.[hc])

- various branch predictors
  - static
  - BTB w/ 2-bit saturating counters
  - 2-level adaptive
- important interfaces:
  - bpred_create(class, size)
  - bpred_lookup(pred, br_addr)
  - bpred_update(pred, br_addr, targ_addr, result)

Cache Module (cache.[hc])

- ultra-vanilla cache module
  - can implement low- and high-assoc, caches, TLBs, etc...
  - efficient for all geometries
  - assumes a single-ported, fully pipelined backside bus
- important interfaces:
  - cache_create(name, nsets, bsize, balloc, usize, assoc
    repl, blk_fn, hit_latency)
  - cache_access(cache, op, addr, ptr, nbytes, when, udata)
  - cache_probe(cache, addr)
  - cache_flush(cache, when)
  - cache_flush_addr(cache, addr, when)

Event Queue (event.[hc])

- generic event (priority) queue
  - queue event for time t
  - returns events from the head of the queue
- important interfaces:
  - eventq_queue(when, op...)
  - eventq_service_events(when)

Program Loader (loader.[hc])

- prepares program memory for execution
  - loads program text
  - loads program data sections
  - initializes BSS section
  - sets up initial call stack
- important interfaces:
  - ld_load_prog(mem_fn, argc, argv, envp)

Main Routine (main.c, sim.h)

- defines interface to simulators
- important (imported) interfaces:
  - sim_options(argc, argv)
  - sim_config(stream)
  - sim_main()
  - sim_stats(stream)

Physical/Virtual Memory (memory.[hc])

- implements large flat memory spaces in simulator
  - uses single-level page table
  - may be used to implement virtual or physical memory
- important interfaces:
  - mem_access(cmd, addr, ptr, nbytes)

Miscellaneous Functions (misc.[hc])

- lots of useful stuff in here, e.g.,
  - fatal()
  - panic()
  - warn()
  - info()
  - debug()
  - getcore()
  - elapsed_time()
  - getopt()

- architected register variable definitions

Resource Manager (resource.[hc])

- powerful resource manager
  - configure with a resource pool
  - manager maintains resource availability
- resource configuration:
  { “name”, num, { FU_class, issue_lat, op_lat }, ... }
- important interfaces:
  - res_create_pool(name, pool_def, ndefs)
  - res_get(pool, FU_class)