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

Todd M. Austin
taustin@ichips.intel.com
Intel MicroComputer Research Labs
M/S JF3-359, 2111 NE 25th Ave.
Hillsboro, OR 97124-5961

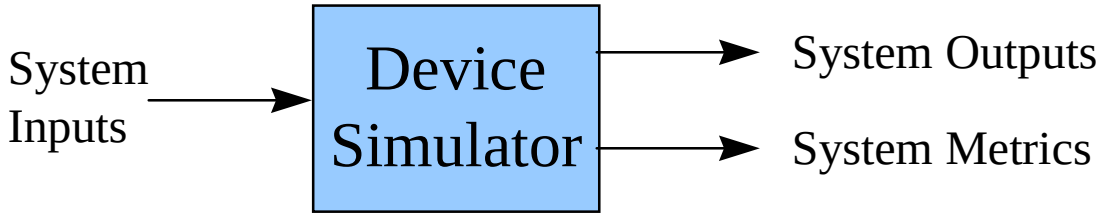
December, 1996

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

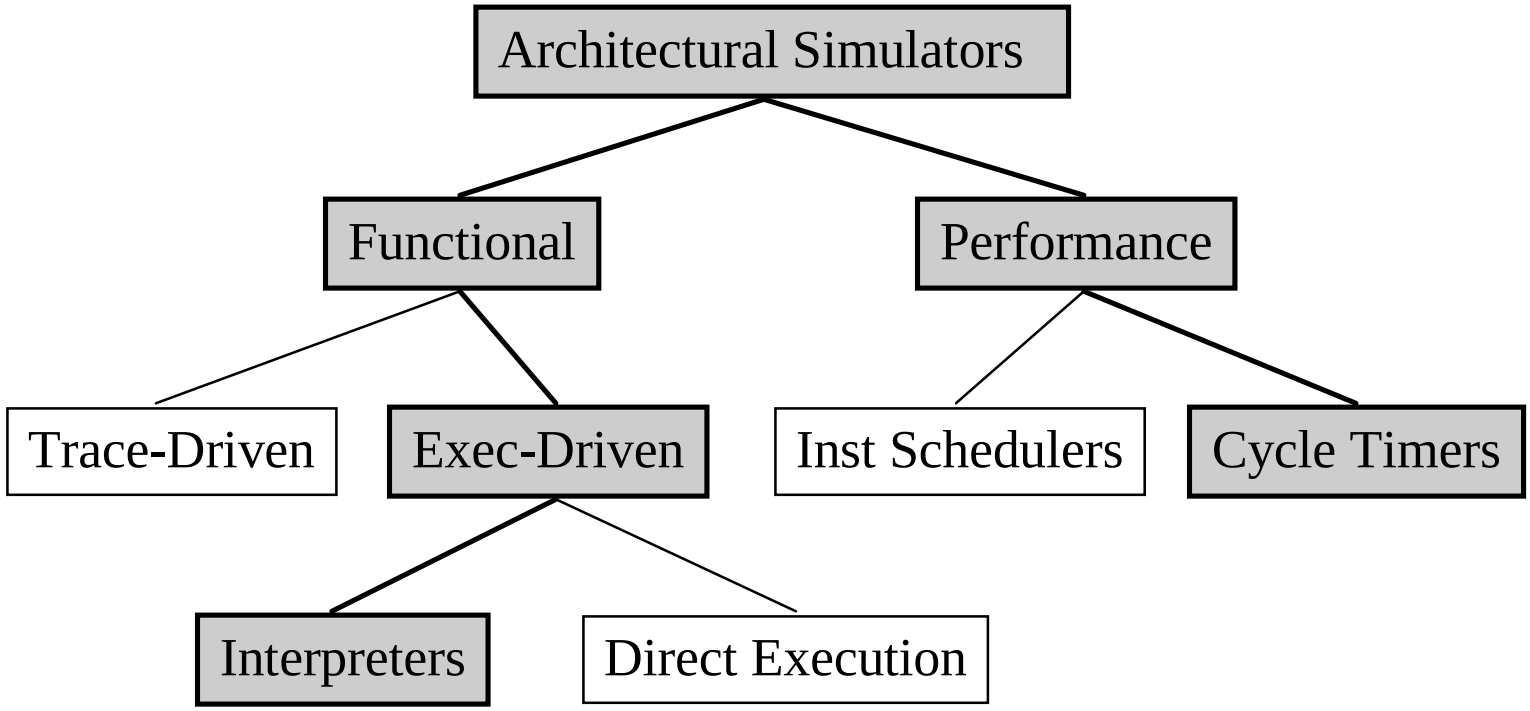
A Computer Architecture Simulator Primer

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



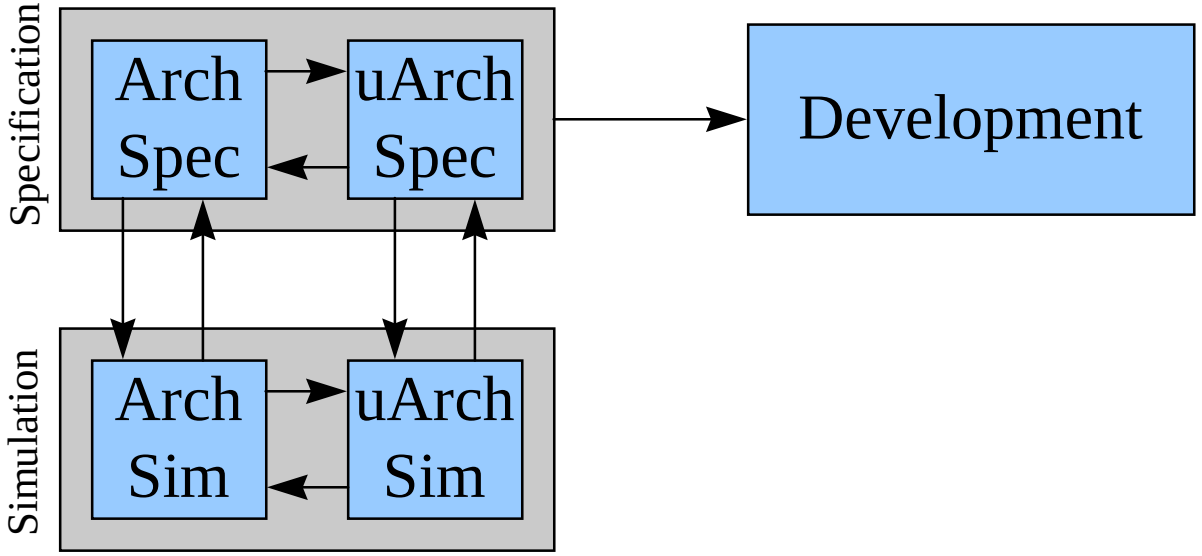
- Why use a simulator?
  - leverage faster, more flexible S/W development cycle
    - permits more design space exploration
    - facilitates validation before H/W becomes available
    - level of abstraction can be throttled to design task
    - possible to increase/improve system instrumentation

A Taxonomy of Simulation Tools



- shaded tools are included in the SimpleScalar tool set

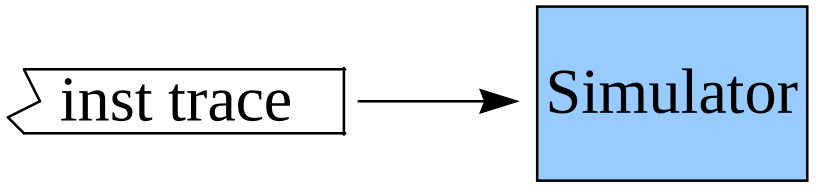
Functional vs. Performance Simulators



- 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

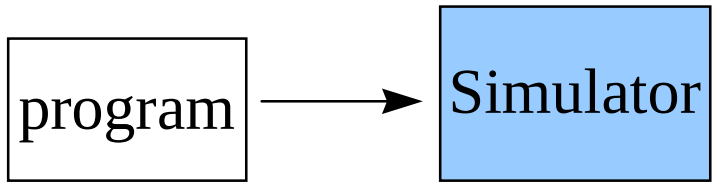
Execution- vs. Trace-Driven Simulation

- trace-based simulation:



  - simulator reads a “trace” of instructions captured during a previous execution
  - easiest to implement, no functional component needed

- execution-driven simulation:

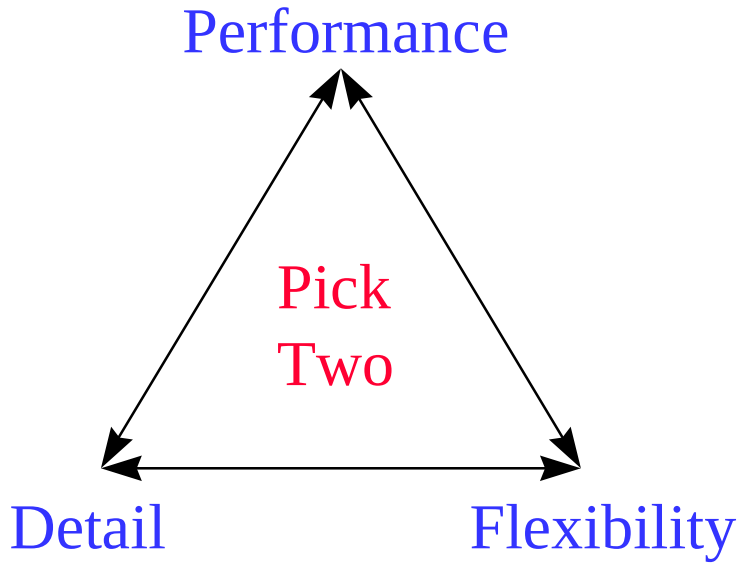


  - simulator “runs” the program, generating a trace on-the-fly
  - more difficult to implement, but has many advantages
  - direct-execution: instrumented program runs on host

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

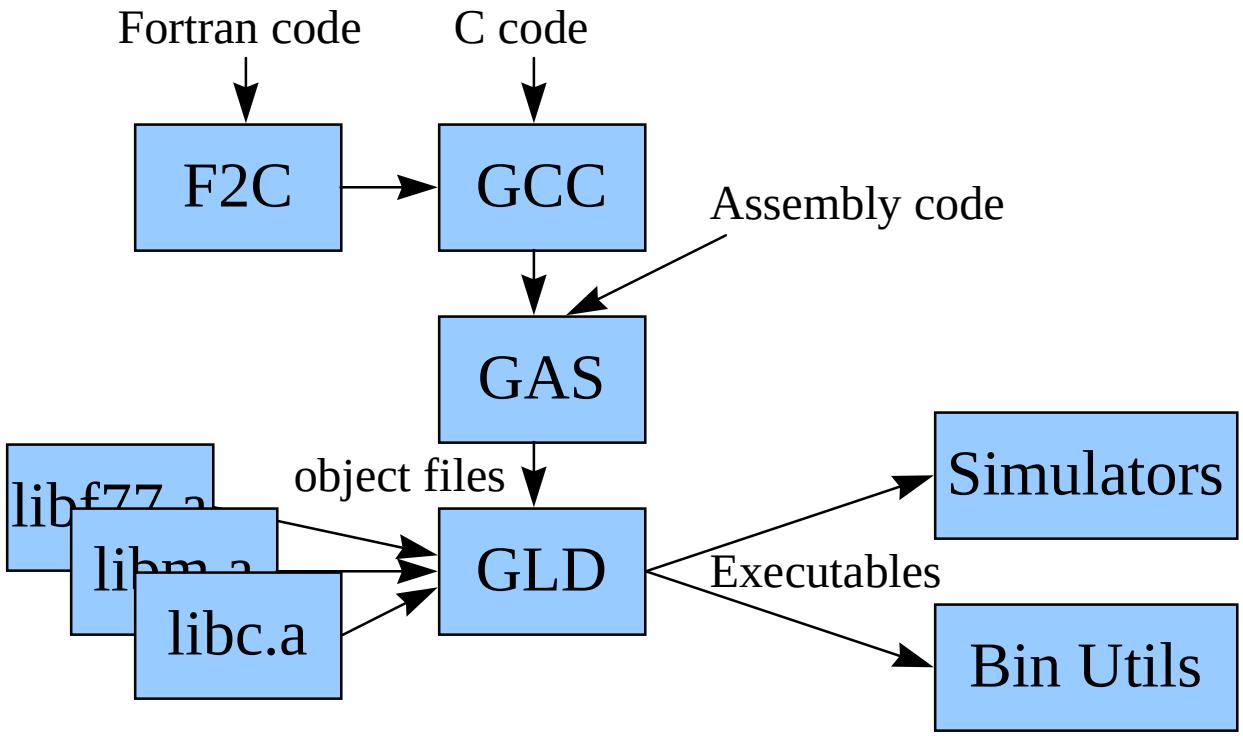
- Computer Architecture Simulation Primer
- SimpleScalar Tool Set
  - **Overview**
  - User's Guide
- SimpleScalar Instruction Set Architecture
- Out-of-Order Issue Simulator
  - Model Microarchitecture
  - Implementation Details
- Hacking SimpleScalar
- Looking Ahead

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
- actively developed and maintained
- 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

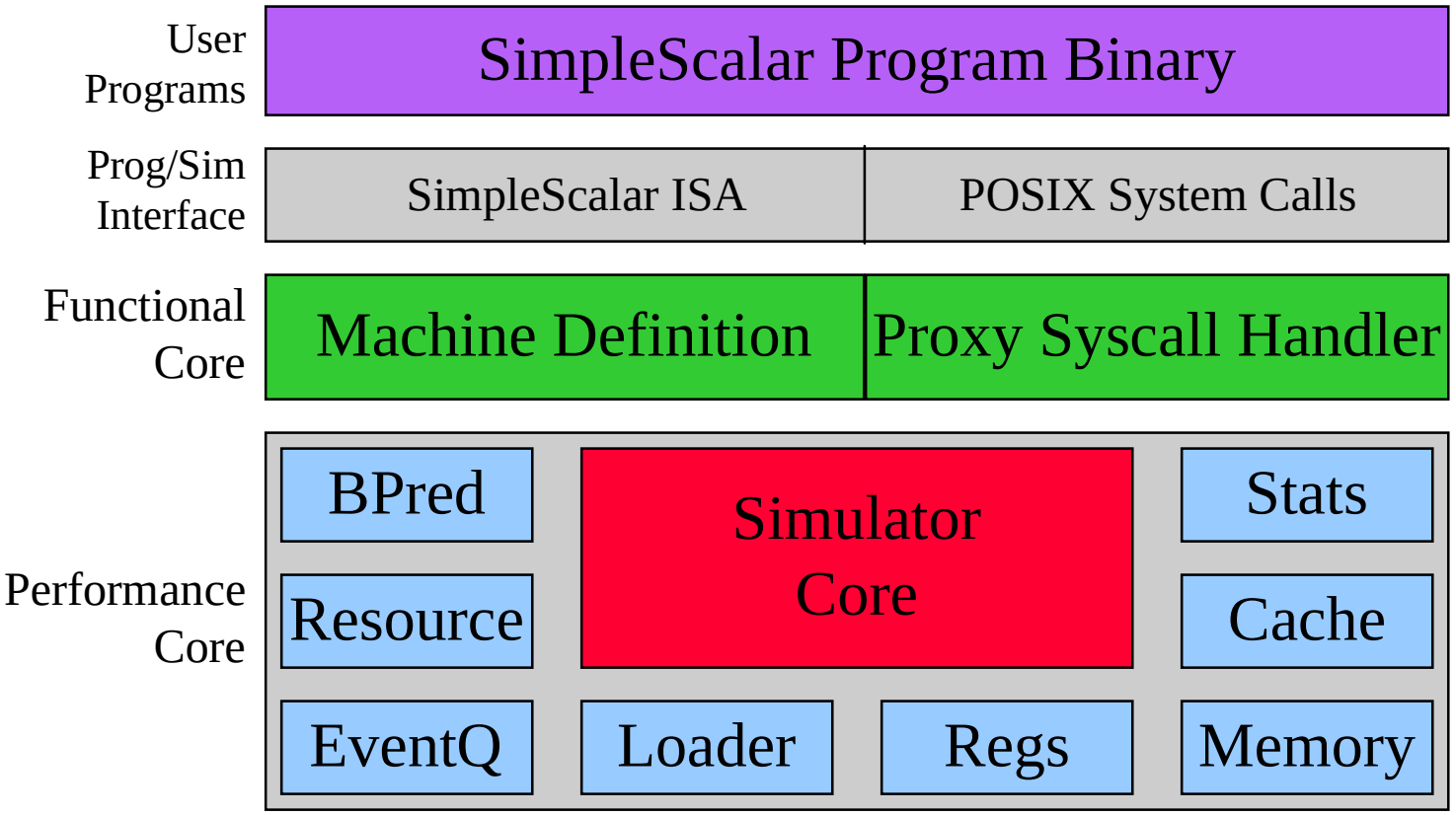
SimpleScalar Tool Set

- 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 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-Cache	Sim-Inorder	Sim-Outorder
- 270 lines - functional - 2 MIPS	- 272 lines - functional w/ checks	- 390 lines - functional - cache stats	- 1110 lines - performance - inorder issue - branch pred. - mis-spec. - ALUs - cache	- 2221 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

Installation Notes

- follow the installation directions in the tech report, and

  DON'T PANIC!!!!

- avoid building GLIBC
  - it's a non-trivial process
  - use the big- and little-endian, pre-compiled libraries in `ss-bootstrap/`
- if you have problems, send e-mail to the SimpleScalar mailing list: `simplescalar@cs.wisc.edu`
- please e-mail install mods to: `dburger@cs.wisc.edu`
- x86 port has limited functionality, portability
  - reportedly only works under little-endian Linux

User's Guide

- 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 foo
```

- building a library, use `ssbig-na-sstrix-{ar,ranlib}`

Simulator Options

- sim-fast, sim-safe:

  no options

- sim-cache:

  -d name:sets:bsize:assoc:repl{r,l,f} - D-Cache
  -i name:sets:bsize:assoc:repl{r,l,f} - I-Cache
  -t name:sets:psize:assoc:repl{r,l,f} - D-TLB
  -f - flush caches on system calls

- sim-inorder:

  -w # - issue with
  -b # - predictor config
  -j # - mis-prediction penalty
  -m # - L1 miss latency

Simulator Options

- sim-outorder

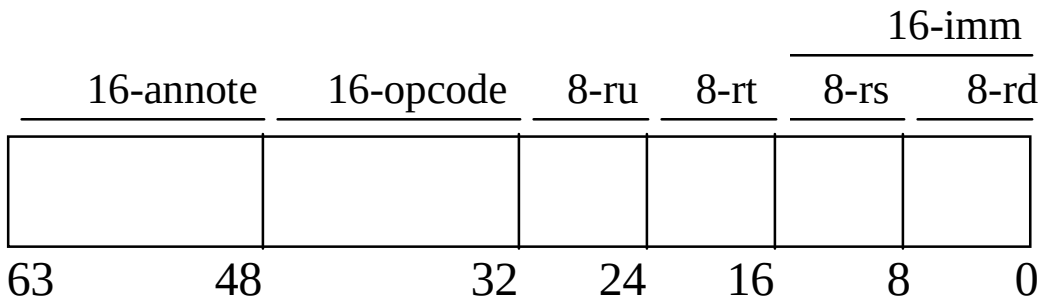
-D #	- decode width
-R #	- ROB size
-L #	- Load/Store queue size
-0	- no mis-spec modeling
-1	- force in-order issue

Tutorial Overview

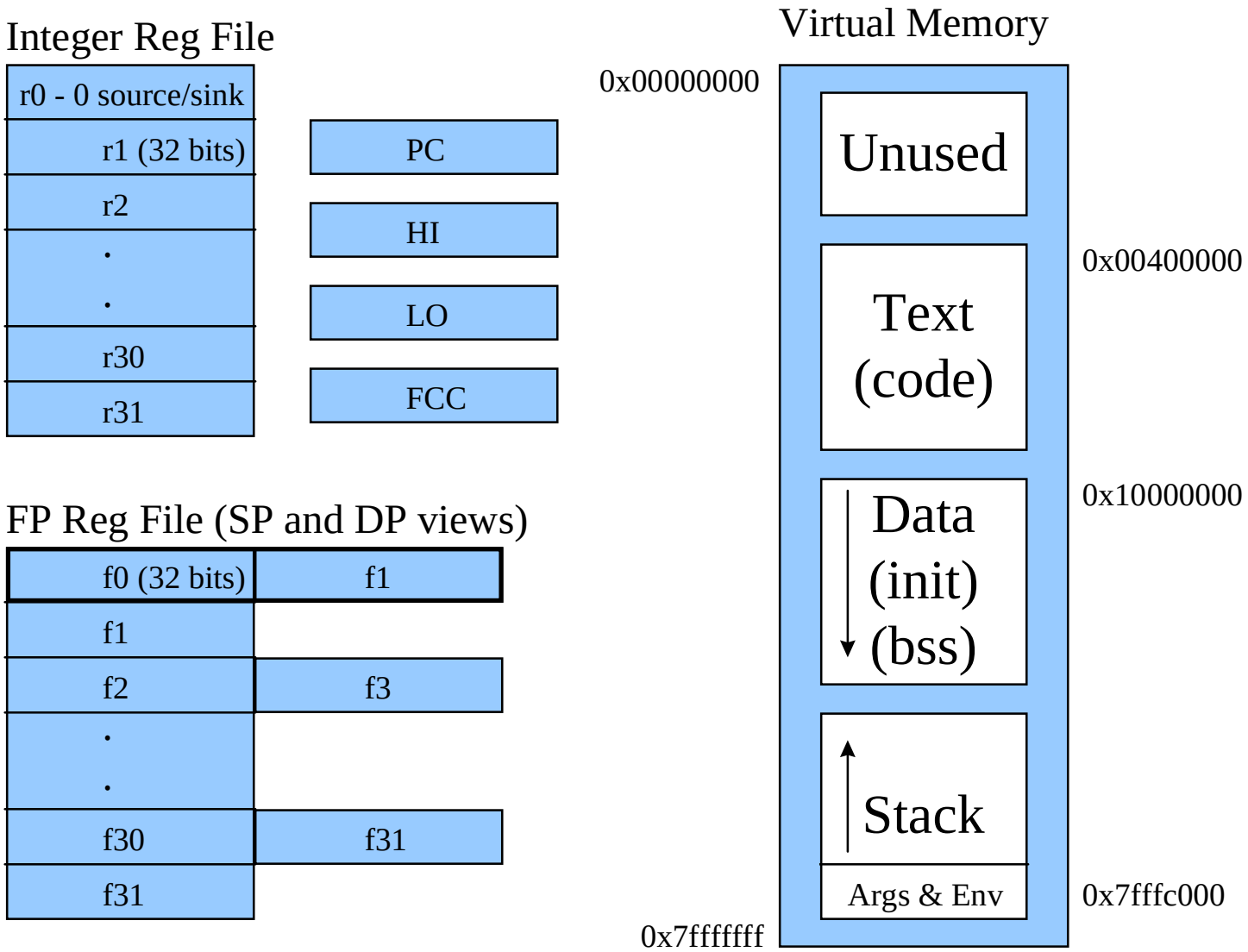
- Computer Architecture Simulation Primer
- SimpleScalar Tool Set
  - Overview
  - User's Guide
- **SimpleScalar Instruction Set Architecture**
- Out-of-Order Issue Simulator
  - Model Microarchitecture
  - Implementation Details
- Hacking SimpleScalar
- Looking Ahead

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

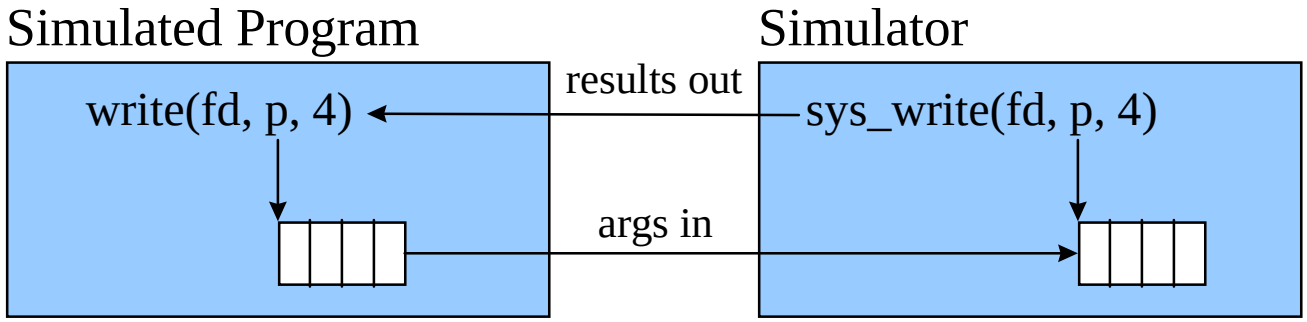
Miscellaneous:

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

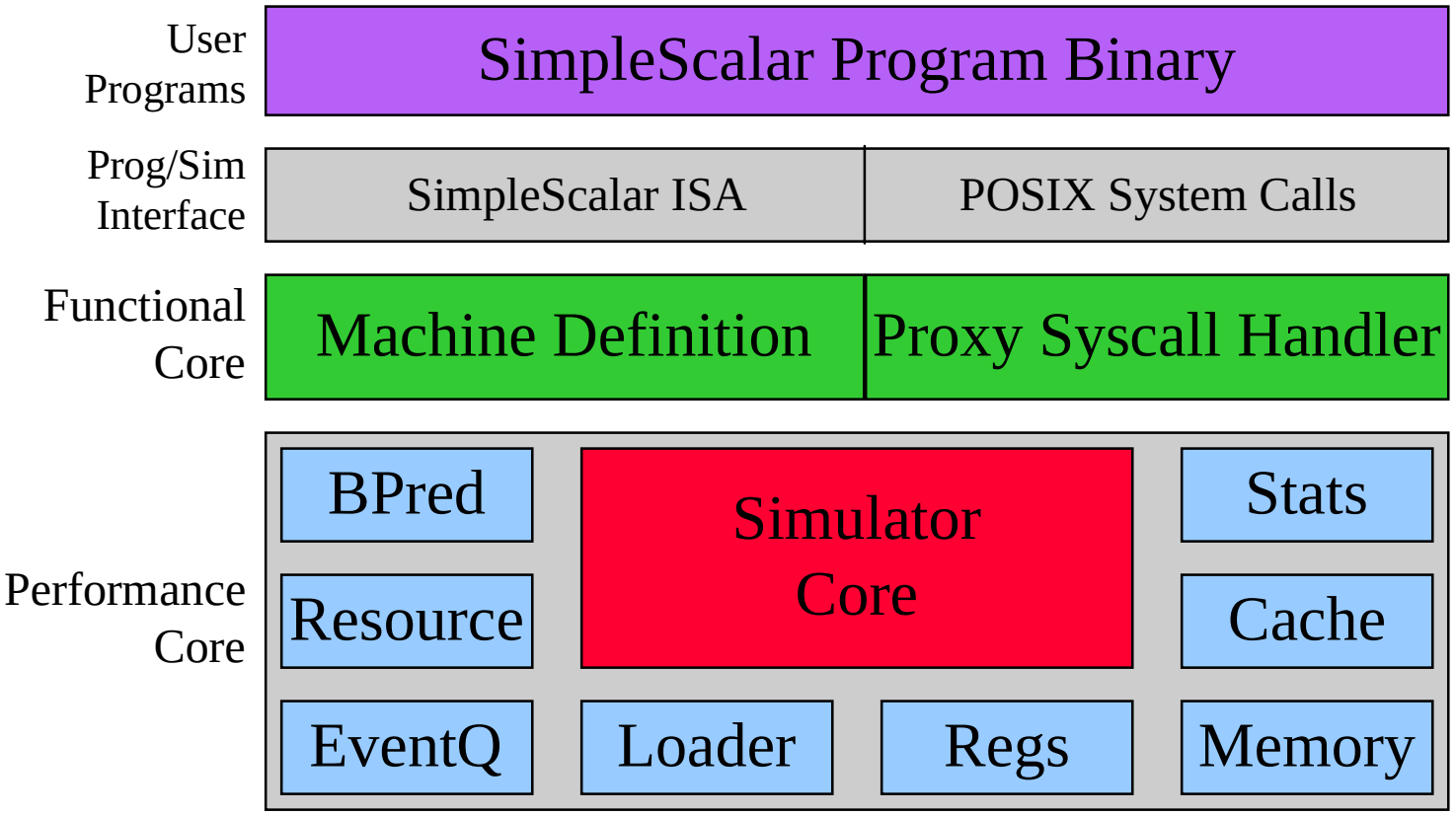


- syscall.c implements a subset of Ultrix Unix system calls
- basic algorithm:
  - decode system call
  - copy arguments (if any) into simulator memory
  - make system call
  - copy results (if any) into simulated program memory

Tutorial Overview

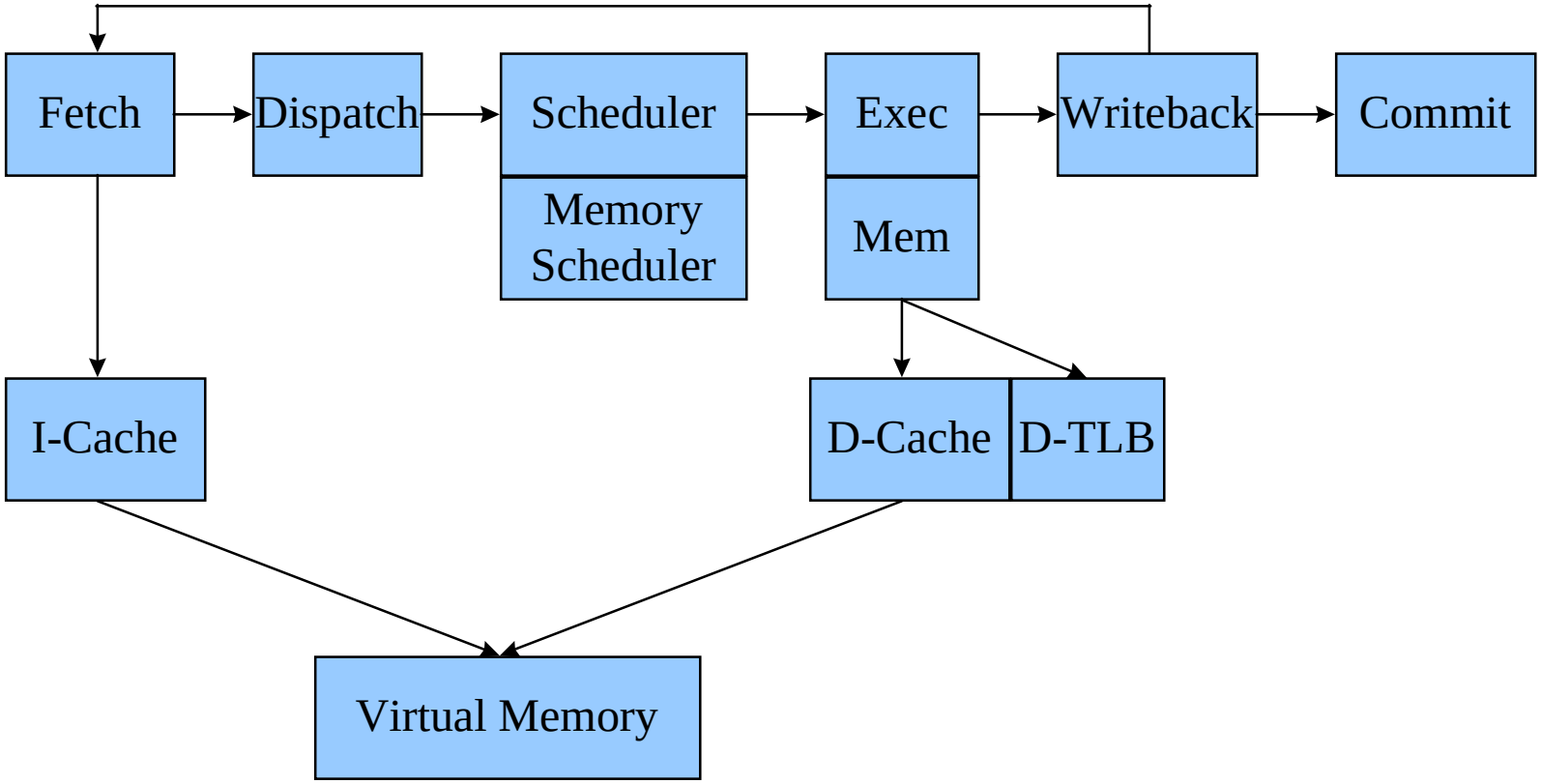
- Computer Architecture Simulation Primer
- SimpleScalar Tool Set
  - Overview
  - User's Guide
- SimpleScalar Instruction Set Architecture
- Out-of-Order Issue Simulator
  - **Model Microarchitecture**
  - Implementation Details
- Hacking SimpleScalar
- Looking Ahead

Simulator Structure



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

Out-of-Order Issue Simulator

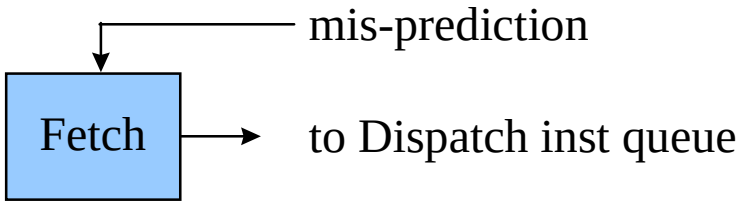


- implemented in `sim-outorder.c` and modules

Tutorial Overview

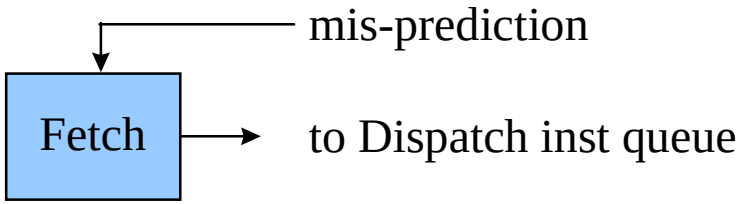
- Computer Architecture Simulation Primer
- SimpleScalar Tool Set
  - Overview
  - User's Guide
- SimpleScalar Instruction Set Architecture
- Out-of-Order Issue Simulator
  - Model Microarchitecture
  - **Implementation Details**
- Hacking SimpleScalar
- Looking Ahead

Out-of-Order Issue Simulator: Fetch



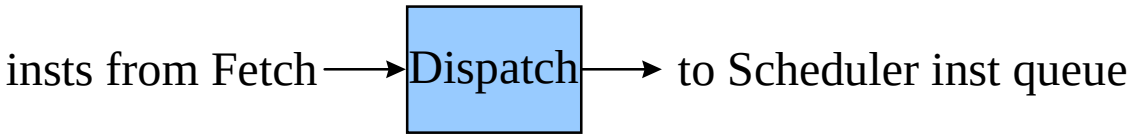
- implemented in `ruu_fetch()`
- models machine fetch bandwidth
- inputs:
  - program counter
  - predictor state (see bpred.[hc])
  - mis-prediction detection from branch execution unit(s)
- outputs:
  - fetched instructions to Dispatch queue

Out-of-Order Issue Simulator: Fetch



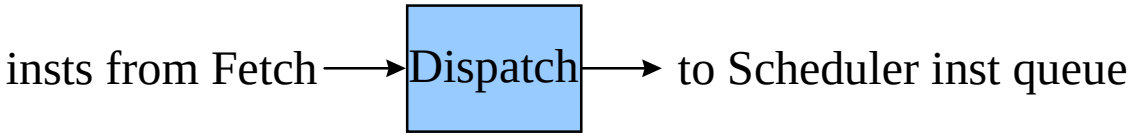
- procedure (once per cycle):
  - fetch insts from *one* I-cache line, block until misses are resolved
  - queue fetched instructions to Dispatch
  - probe line predictor for cache line to access in next cycle

Out-of-Order Issue Simulator: Dispatch



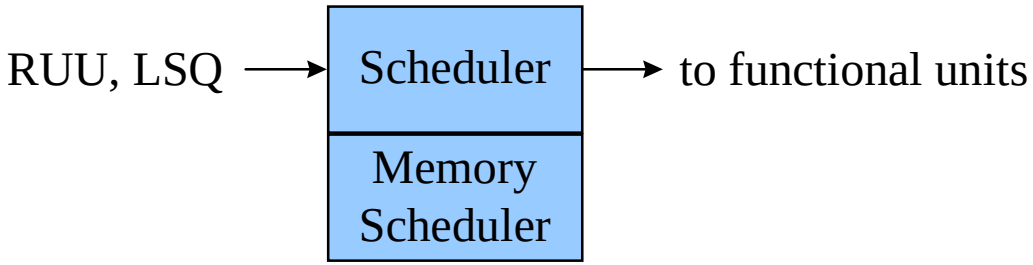
- implemented in `ruu_dispatch()`
- models machine decode, rename, allocate bandwidth
- inputs:
  - instructions from input queue, fed by Fetch stage
  - RUU
  - rename table (create_vector)
  - architected machine state (for execution)
- outputs:
  - updated RUU, rename table, machine state

Out-of-Order Issue Simulator: Dispatch



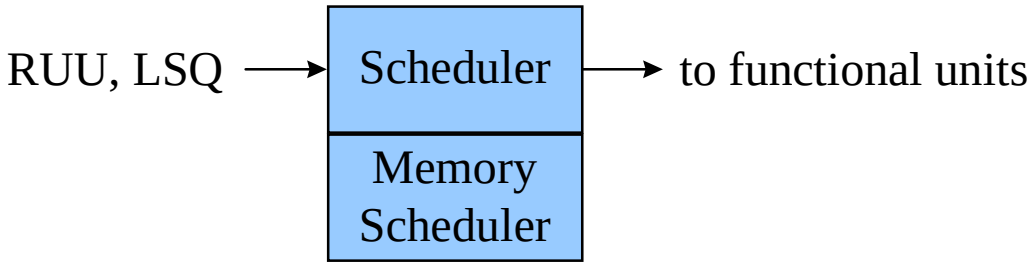
- procedure (once per cycle):
  - fetch insts from Dispatch queue
  - decode and *execute* instructions
    - facilitates simulation of data-dependent optimizations
    - permits early detection of branch mis-predicts
  - if mis-predict occurs:
    - start copy-on-write of architected state to speculative state buffers
  - enter and link instructions into RUU and LSQ (load/store queue)
    - links implemented with RS_LINK structure
    - loads/stores are split into two insts: ADD → Load/Store
    - improves performance of memory dependence checking

Out-of-Order Issue Simulator: Scheduler



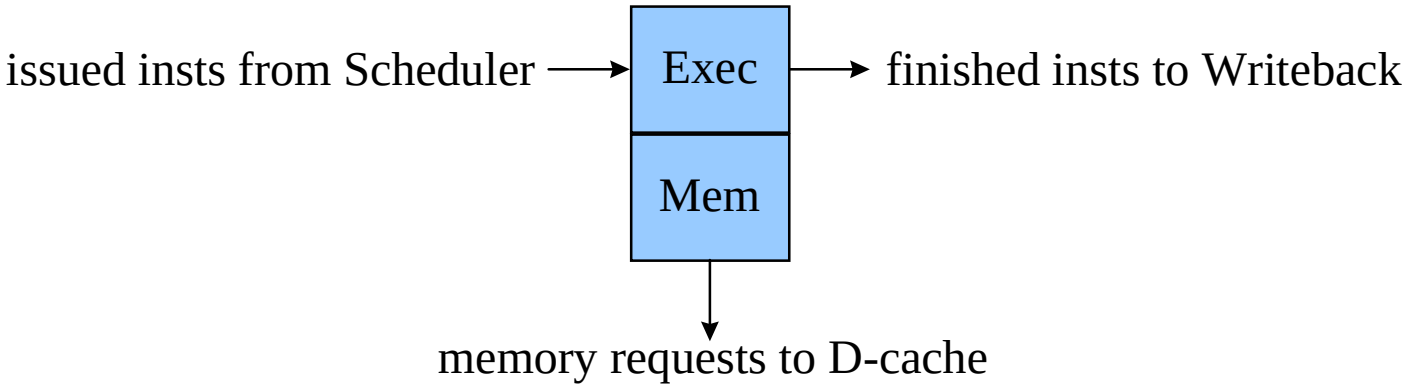
- implemented in `ruu_issue()`and `lsq_refresh()`
- models instruction, wakeup, and issue to functional units
  - separate schedulers to track register and memory dependencies
- inputs:
  - RUU, LSQ
- outputs:
  - updated RUU, LSQ
  - updated functional unit state

Out-of-Order Issue Simulator: Scheduler



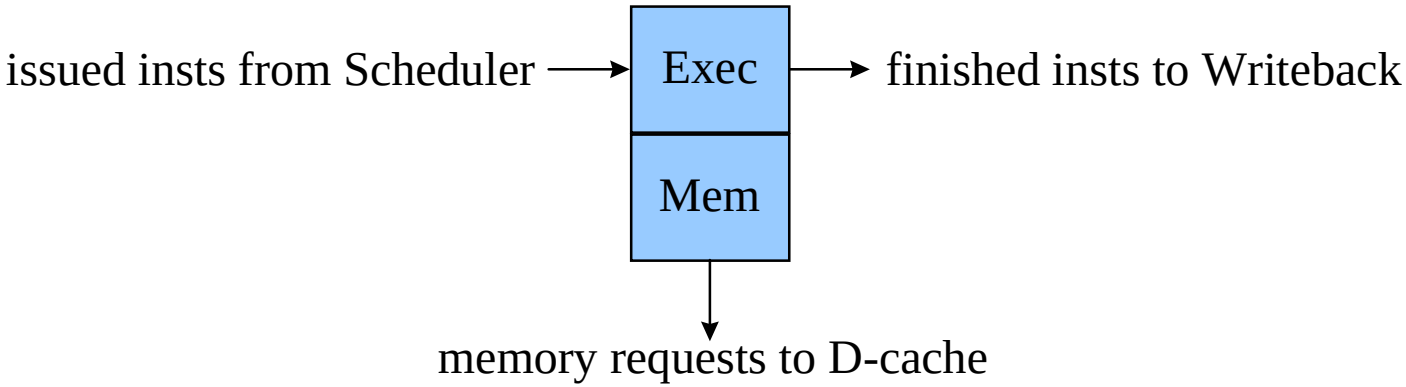
- procedure (once per cycle):
  - locate instructions with all register inputs ready
    - in ready queue, inserted during dependent inst's wakeup walk
  - locate instructions with all memory inputs ready
    - determined by walking the load/store queue
    - if earlier store with unknown addr → stall issue (and poll)
    - if earlier store with matching addr → store forward
    - else → access D-cache

Out-of-Order Issue Simulator: Execute



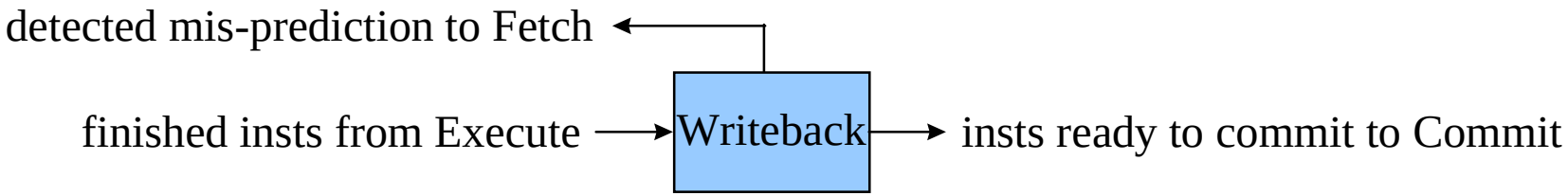
- implemented in `ruu_issue()`
- models func unit and D-cache issue and execute latencies
- inputs:
  - ready insts as specified by Scheduler
  - functional unit and D-cache state
- outputs:
  - updated functional unit and D-cache state
  - updated event queue, events notify Writeback of inst completion

Out-of-Order Issue Simulator: Execute



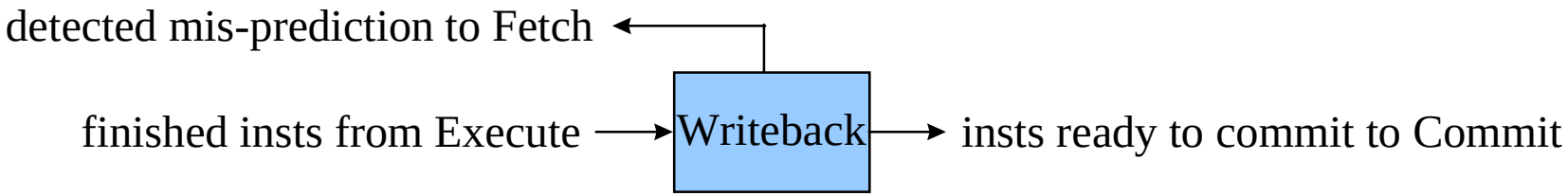
- procedure (once per cycle):
  - get ready instructions (as many as supported by issue B/W)
  - probe functional unit state for availability and access port
  - reserve unit it can issue again
  - schedule writeback event using operation latency of functional unit
    - for loads satisfied in D-cache, probe D-cache for access latency
    - also probe D-TLB, stall future issue on a miss
    - D-TLB misses serviced at commit time with fixed latency

Out-of-Order Issue Simulator: Writeback



- implemented in `ruu_writeback()`
- models writeback bandwidth, detects mis-predictions, initiated mis-prediction recovery sequence
- inputs:
  - completed instructions as indicated by event queue
  - RUU, LSQ state (for wakeup walks)
- outputs:
  - updated event queue
  - updated RUU, LSQ, ready queue
  - branch mis-prediction recovery updates

Out-of-Order Issue Simulator: Writeback



- procedure (once per cycle):
  - get finished instructions (specified in event queue)
  - if mis-predicted branch:
    - recover RUU
      - walk newest inst to mis-pred branch
      - unlink insts from output dependence chains
    - recover architected state
      - roll back to checkpoint
  - wakeup walk: walk dependence chains of inst outputs
    - mark dependent inst's input as now ready
    - if all reg dependencies of the dependent inst are satisfied, wake it up (memory dependence check occurs later in Issue)

Out-of-Order Issue Simulator: Commit

- implemented in `ruu_commit()`
- models in-order retirement of instructions, store commits to the D-cache, and D-TLB miss handling
- inputs:
  - completed instructions in RUU/LSQ that are ready to retire
  - D-cache state (for committed stores)
- outputs:
  - updated RUU, LSQ
  - updated D-cache state

Out-of-Order Issue Simulator: Commit

- procedure (once per cycle):
  - while head of RUU is ready to commit (in-order retirement)
    - if D-TLB miss, then service it
    - then if store, attempt to retire store into D-cache, stall commit otherwise
    - commit inst result to the architected register file, update rename table to point to architected register file
    - reclaim RUU/LSQ resources

Out-of-Order Issue Simulator: Main

```
ruu_init()
for (;;) {
  ruu_commit();
  ruu_writeback();
  ruu_dispatch();
  lsq_refresh();
  ruu_issue();
  ruu_fetch();
}
```

- implemented in `sim_main()`
- walks pipeline from Commit to Fetch
  - backward pipeline traversal eliminates relaxation problems, e.g., provides correct inter-stage latch synchronization
- loop is execute via a `longjmp()` to `main()` when simulated program executes an `exit()` system call

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 Compiler (GCC)

- see GCC.info in the GNU GCC release for details on the internals of GCC
- all SimpleScalar-specific code is in the config/ss in the GNU GCC source tree
- use instruction annotations to add new instruction, as you won't have to then hack the assembler
- avoid adding new linkage types, or you will have to hack GAS, GLD, and libBFD.a, all of which are very painful

Hacking the Assembler (GAS)

- most of the time, you should be able to avoid this by using instruction annotations
- new instructions are added in libopcode.a, new instructions will also be picked up by disassembler
- new linkage types require hacking GLD and libBFD.a, which is very painful

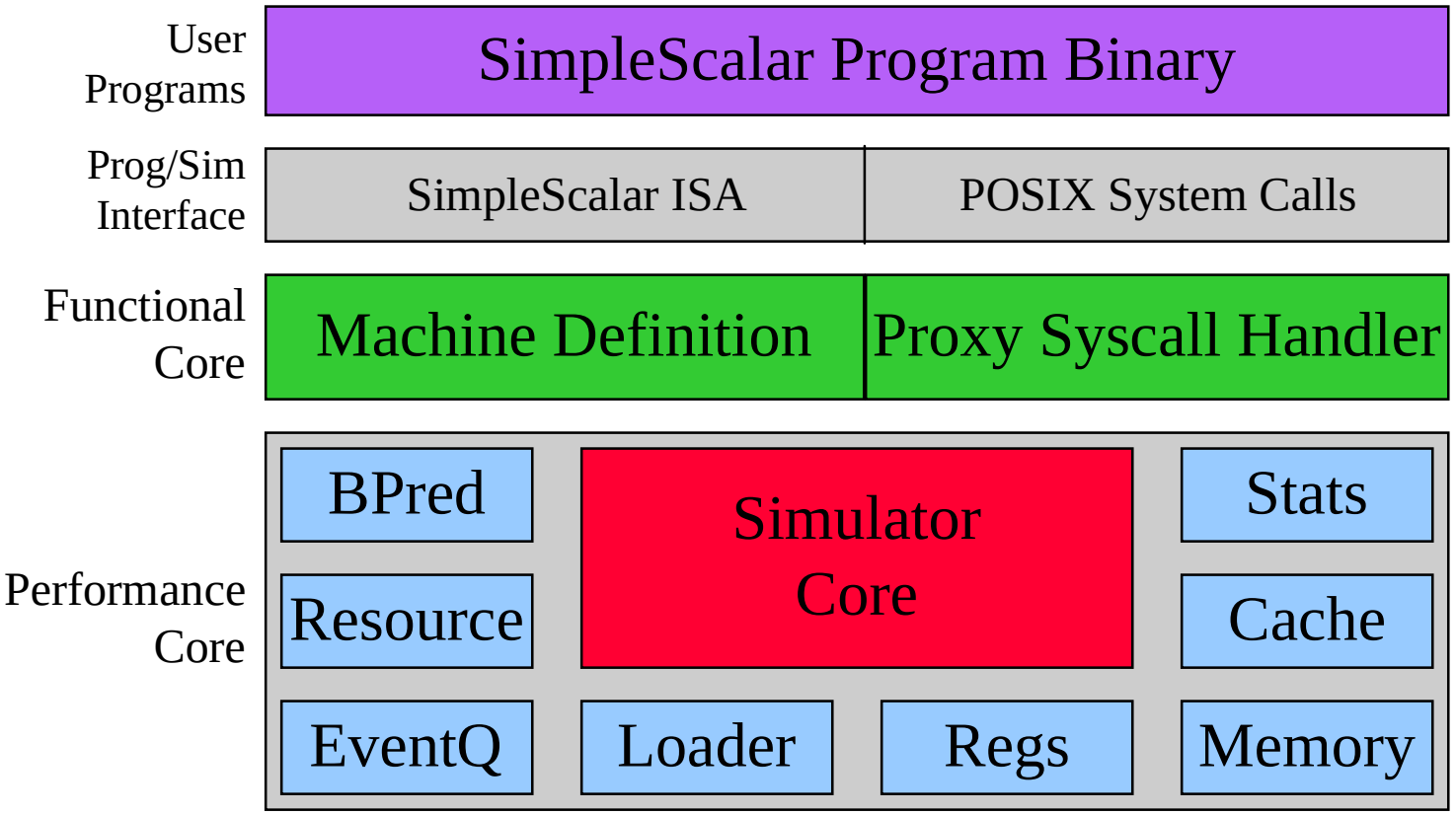
Hacking the Linker (GLD and libBFD.a)

- avoid this if possible, both tools are difficult to comprehend and generally delicate
- if you must...
  - emit a linkage map (-Map mapfile) and then edit the executable in a postpass
  - KLINK, from my dissertation work, does exactly this

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

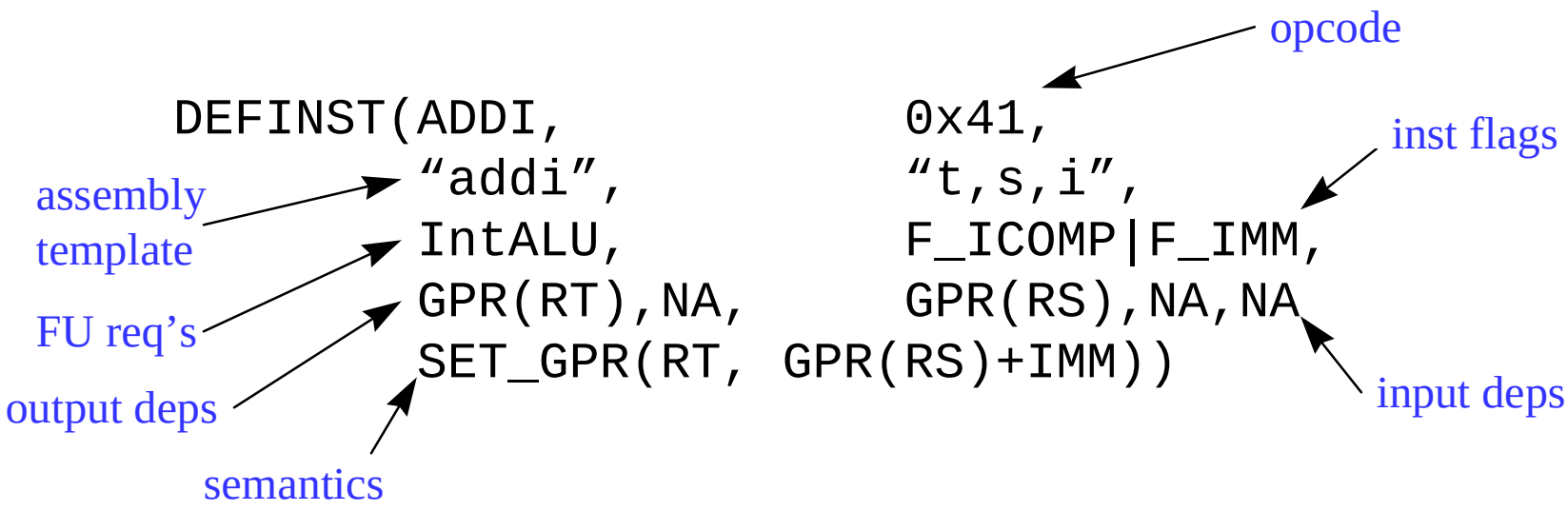
Simulator Structure



- 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 DEFFU(FU,DESC)
#define DEFLINK(OP,MSK,NAME,MASK,SHIFT)                             \
      case OP:                                                      \
        panic("attempted to execute a linking opcode");
#define CONNECT(OP)
#include "ss.def"
#undef DEFFU
#undef DEFINST
#undef DEFLINK
#undef CONNECT
    }
```

Crafting an Decoder

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

switch (op) {
#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 DEFFU(FU,DESC)
#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 DEFFU
#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;
    }
```

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()

Register State (regs.[hc])

- 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)

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**

Looking Ahead

- upcoming maintenance release (in December)
  - install fixes
  - much more documentation
  - DLite - the light debugger
  - stats package
  - options package
- SimpleScalar port to WinNT
- MP/MT support for SimpleScalar simulators
- Linux port to SimpleScalar
  - with device-level emulation and user-level file system
- Alpha and SPARC target support (SimpleScalar and MIPS currently exist)

To Get Plugged In

- Technical Report:
  - "*Evaluating Future Microprocessors: the SimpleScalar Tools Set*", UW-Madison Tech Report #1308, July 1996
- SimpleScalar Public Release
  - Public Release 0 is available from: `http://www.cs.wisc.edu/~mscalar/simplescalar.html`
  - the "early bird" release
  - to be followed in Dec by Public Release 1, includes more documentation, a hacker's guide, install fixes
- SimpleScalar mailing list:
  - contact Doug Burger (`dburger@cs.wisc.edu`) to join

Backups

Experiences and Insights

- the history of SimpleScalar:
  - Sohi's CSim begat Franklin's MSim begat SimpleScalar
  - first public release in July '96, made with Doug Burger
- key insights:
  - major investment req'd to develop sim infrastructure
    - 2.5 years to develop, while at UW-Madison
  - modular component design reduces design time and complexity, improves quality
  - fast simulators improve the design process, although it does introduce some complexity
  - virtual target improves portability, but limits workload
  - execution-driven simulation is worth the trouble

Advantages of Execution-Driven Simulation

- execution-based simulation
  - faster than tracing
    - fast simulators: 2+ MIPS, fast disks: < 1 MIPS
  - no need to store traces
  - register and memory values usually not in trace
    - functional component maintains precise state
    - extends design scope to include data-value-dependent optimizations
  - support mis-speculation cost modeling
    - on control and data dependencies
  - may be possible to eliminate most execution overheads

Example Applications

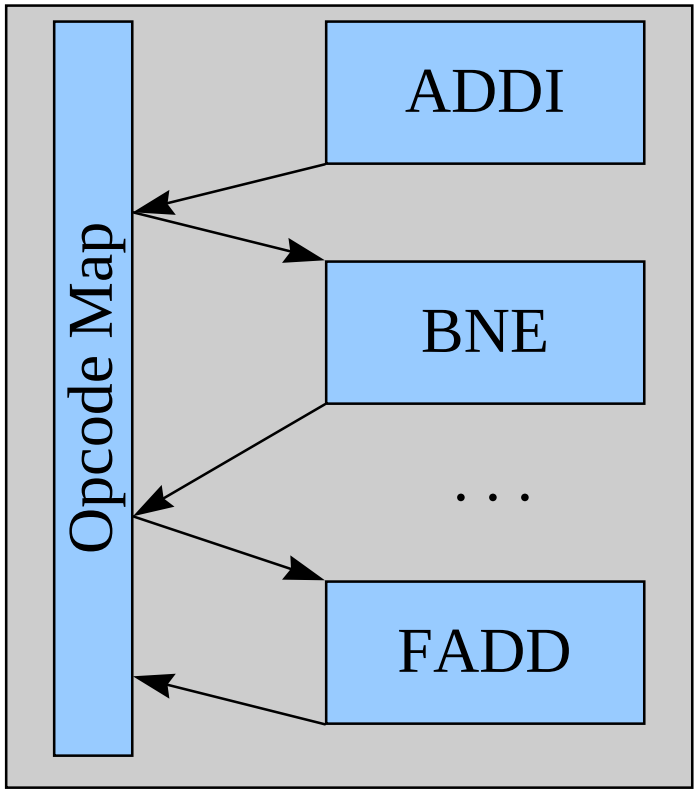
- my dissertation: “H/W and S/W Mechanisms for Reducing Load Latency”
  - fast address calculation
  - zero-cycle loads
  - high-bandwidth address translation
  - cache-conscious data placement
- other users:
  - SCI project
  - Galileo project
  - more coming on-line

Related Tools

- SimOS from Stanford
  - includes OS and device simulation, and MP support
  - not portable, currently only runs on MIPS hosts
  - little source code since much of the tool chain is commercial code, e.g., compiler, operating system
- functional simulators:
  - direct execution via dynamic translation: Shade, FX32!
  - direct execution via static translation: Atom, EEL, Pixie
  - machine interpreters: Msim, DLXSim, Mint

Fast Functional Simulator

sim_main()



- on SPARC host, some globals are register allocated

SimpleScalar Wish List

- Linux port to SimpleScalar
  - with device-level emulation
- MP/MT support for SimpleScalar sims
- Port SimpleScalar tool set to Windows NT
- simple debugger support for SimpleScalar sims
  - fast code/data breakpoints, machine state access, hooks for microarchitecture state access
- GDB port to SimpleScalar
  - would allow source-level debugging
- Alpha, SPARC target support (MIPS exists)
- NOTE: ping the mailing list before starting a project